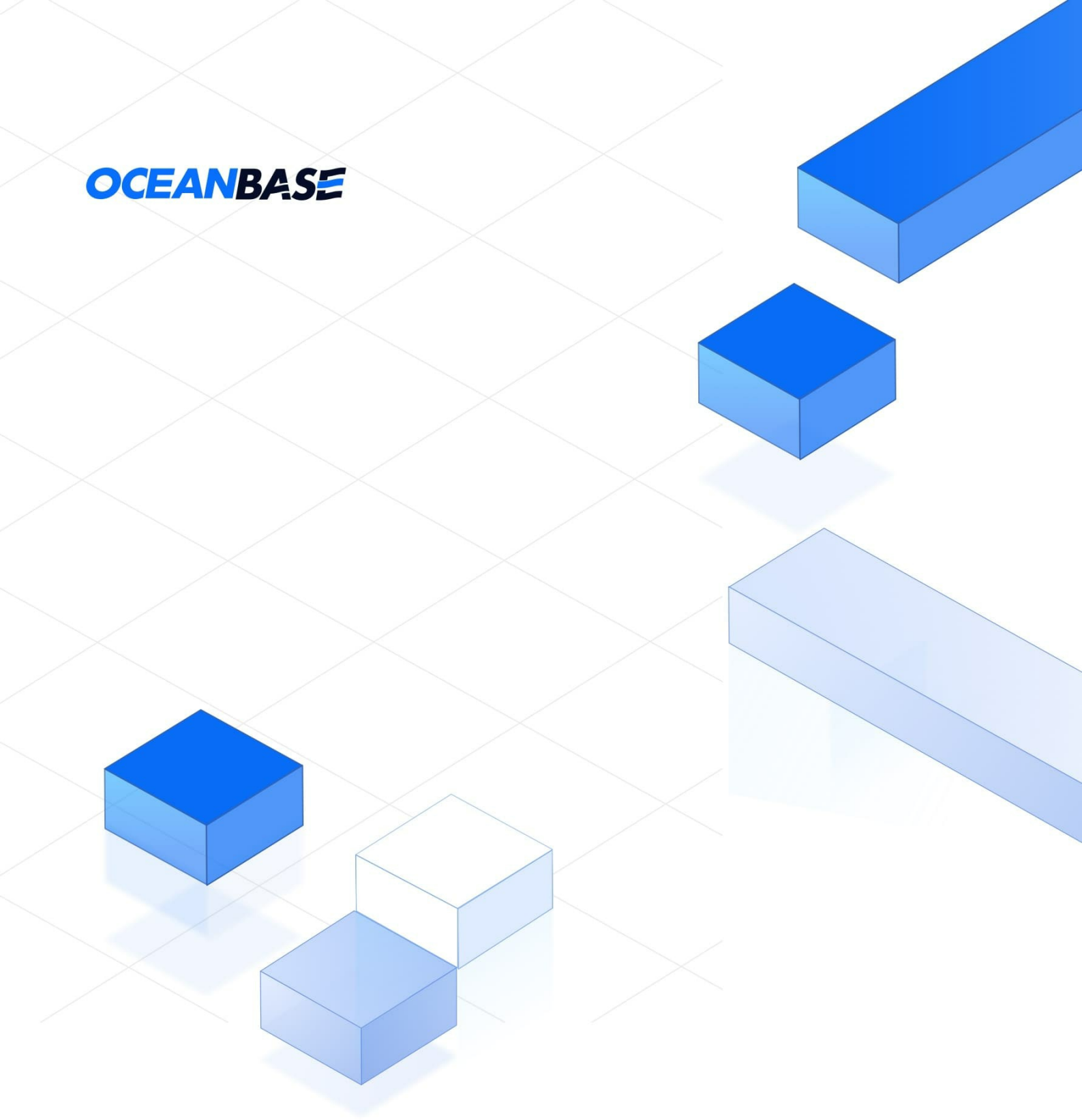


OceanBase 数据库

OceanBase 数据库系统概念

| 产品版本：V3.2.2 | 文档版本：20220304

声明

蚂蚁集团版权所有©2020，并保留一切权利。

商标声明

免责声明

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.OceanBase 数据库简介

1.1. 了解 OceanBase 数据库

OceanBase 数据库是蚂蚁集团完全自主研发的原生分布式关系数据库软件。它在普通服务器集群上实现金融级稳定性和高可用，首创“三地五中心”城市级故障自动无损容灾新标准，具备基于原生分布式的卓越的水平扩展能力。OceanBase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目前承担蚂蚁集团支付宝 100% 核心链路，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。

透明可扩展

OceanBase 数据库独创的总控服务和分区级负载均衡能力使系统具有极强的可扩展性，可以在线进行平滑扩容或缩容，并且在扩容后自动实现系统负载均衡，对应用透明，确保系统的持续运行。

此外，OceanBase 数据库支持超大规模集群（节点超过 1500 台，最大单集群数据量超过 3 PB，单表数量达到万亿行级别）动态扩展，在 TPC-C 场景中，系统扩展比可以达到 1:0.9，使用户投资的硬件成本被最大化的利用。

高可用

OceanBase 数据库采用基于无共享集群的分布式架构，通过 Paxos 共识协议实现数据多副本的一致性。整个系统没有任何单点故障，保证系统的持续可用。支持单机、机房、城市级别的高可用和容灾，可以进行单机房、双机房、两地三中心、三地五中心部署。经过实际测试，可以做到城市级故障 RPO=0，RTO<30 秒，达到国际标准灾难恢复能力最高级别 6 级。OceanBase 数据库还提供了基于日志复制技术的主备库特性，为客户提供更加灵活的高可用和容灾能力。

混合事务和分析处理

OceanBase 数据库独创的分布式计算引擎，能够让系统中多个计算节点同时运行 OLTP 类型的应用和复杂的 OLAP 类型的应用。OceanBase 数据库真正实现了用一套计算引擎同时支持混合负载的能力，让用户通过一套系统解决 80% 的问题，最大化利用集群的计算能力。

多租户

OceanBase 数据库采用了单集群多租户设计，天然支持云时代多租户业务的需求，支持公有云、私有云、混合云等多种部署形式。OceanBase 数据库通过租户实现资源隔离，让每个数据库服务的实例不感知其他实例的存在，并通过权限控制确保不同租户数据的安全性，配合 OceanBase 数据库强大的可扩展性，能够提供安全、灵活的 DBaaS 服务。

高兼容性

OceanBase 数据库针对 Oracle、MySQL 这两种应用最为广泛的数据库生态都给予了很好的支持。对于 MySQL 数据库，OceanBase 数据库兼容 MySQL 5.5/5.6/5.7，基于 MySQL 的应用能够平滑迁移。

对于 Oracle 数据库，OceanBase 数据库能够支持绝大部分的 Oracle 语法和几乎全量过程性语言功能，可以做到大部分的 Oracle 业务进行少量修改后自动迁移。在蚂蚁集团内部和多家金融行业客户的业务中成功完成平滑迁移。

完整自主知识产权

OceanBase 数据库由蚂蚁集团完全自主研发，不基于 MySQL 或 PostgreSQL 等任何开源数据库，能够做到完全自主可控，不会存在基于开源数据库二次开发引起的产品技术限制问题。完全自研，也使得我们能够快速响应客户需求和解决客户问题，没有技术障碍。

高性能

OceanBase 数据库作为准内存数据库，通常只需要操作内存中的数据，并且采用了独创的基于 LSM-Tree 结构的存储引擎，充分利用新存储硬件特性，读写性能均远超传统关系型数据库。

OceanBase 数据库的分布式事务引擎严格支持事务的 ACID 属性，并且在整个集群内严格支持数据强一致性，是全球唯一一家通过了标准 TPC-C 测试的原生分布式关系型数据库产品。在保证分布式事务对应用透明的同时，进行了大量性能优化。

安全性

OceanBase 数据库在调研了大量企业对于数据库软件的安全需求，并参考了各种安全标准之后，实现了企业需要的绝大部分安全功能，支持完备的权限与角色体系，支持 SSL、数据透明加密、审计、Label Security、IP 白名单等功能，并通过了等保三标准测试。

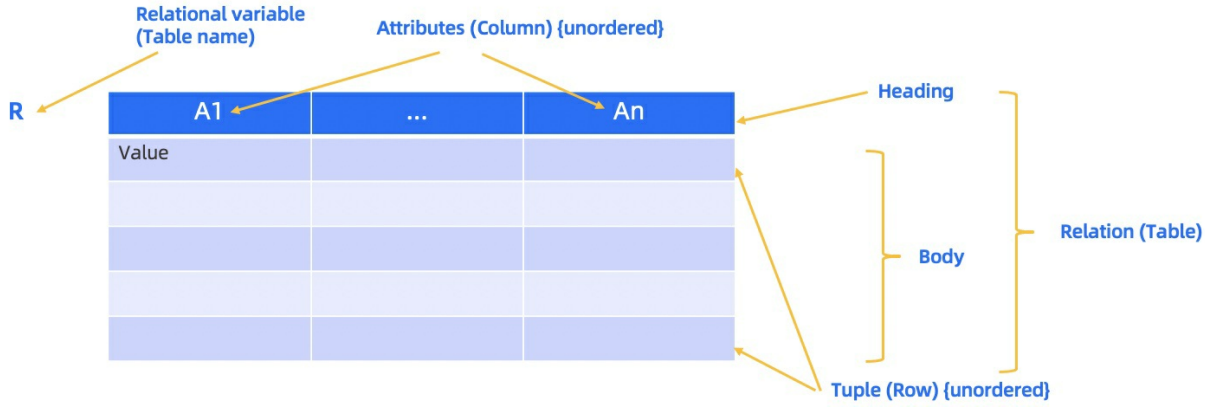
1.2. 了解 OceanBase 数据库

1.3. 数据库基础概念

在开始深入了解 OceanBase 数据库的技术架构和原理之前，您可以简单了解下数据库相关的一些基础概念，以便您对数据库有一些基础的了解，方便更好的阅读后续的章节。

数据库管理系统是一种负责数据的存储、组织和检索的软件系统。数据库的应用程序通过一组接口操作和访问数据库中的数据。关系数据库是一类最为广泛使用的数据库系统，它遵循关系模型理论。关系模型由 E.F.Codd 于 1970 年提出，它具备如下要点：

- 结构化：所有数据遵循良好定义的结构解释其内容。
- 规范操作：应用程序使用良好定义的操作来修改和访问数据。
- 完整性规则：任何操作需要保持预定义的完整性规则。



关系

在关系模型中，数据由一组关系（Relation）组成。

关系是元组（Tuple）的集合，元组是属性（Attribute Value）的集合。在关系数据库中，关系对应于表，表是关系的二维表示，表由若干行（元组）数据组成，每行由相同的有序的列（属性）组成。创建表的时候可以指定一组完整性约束。每列数据具有确定的数据类型。应用程序可以使用结构化查询语言 SQL 对表中的数据进行查询和修改。

SQL

SQL 语言由国际标准化组织定义了标准，但各种数据库提供的 SQL 语言方言又有各种细微的差别。

SQL 语言是一种声明式语言，它只描述要什么，而不需要指明怎么做。数据库系统使用 SQL 优化器选择最好的执行逻辑。SQL 语言包含一系列 SQL 语句类型，最常用的是 SELECT、INSERT、UPDATE、DELETE。SELETE 语句可以访问表数据并进行复杂的计算操作。INSERT、UPDATE、DELETE 等语句对数据进行修改，统称 DML 语句（Data Manupulation Language）。此外，DDL 语句可以定义和修改表的结构等，DCL 语句执行事务控制。传统关系数据库运行在单个计算机节点上。

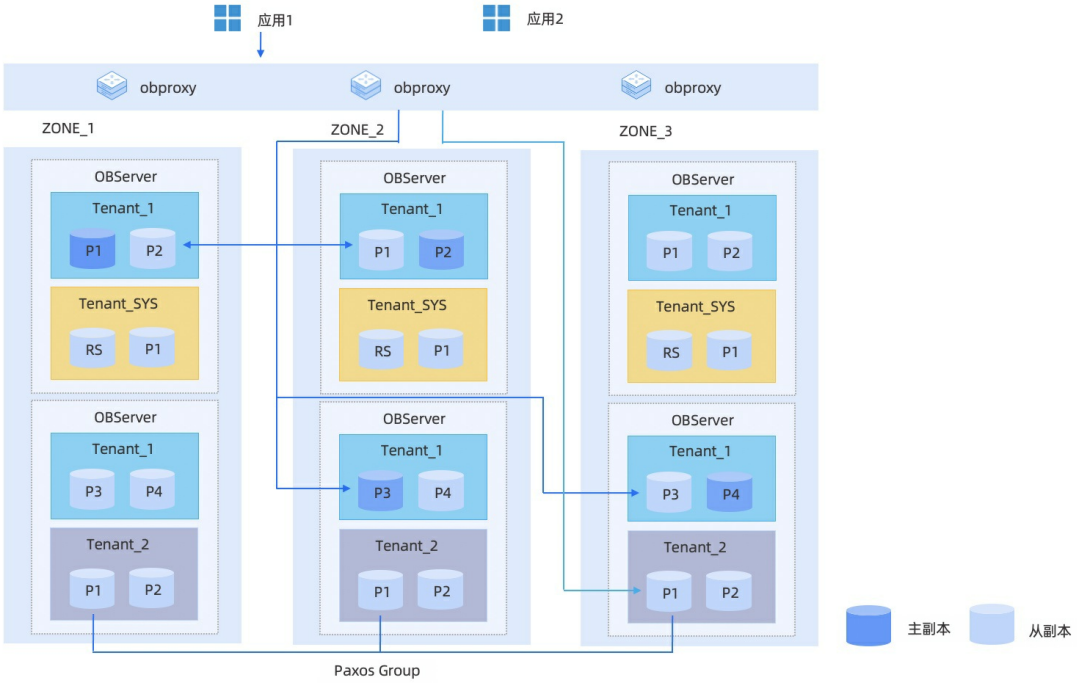
集群

计算机集群，简称集群，是一种通过一组松散集成的计算机软件或硬件连接起来高度紧密地协作完成计算工作的计算机系统。

OceanBase 数据库是一种分布式关系数据库，它使用分布式存储、分布式事务、分布式 SQL 执行、分布式共识等技术，在一个集群上提供关系数据库服务，对应用程序呈现的是与集中式数据库一致的使用接口。OceanBase 数据库使用普通服务器作为组成节点，每个节点拥有完整的 CPU、内存、本地磁盘，节点之间通过普通数据中心网络进行互联。分布式数据库比集中式数据库更易于低成本地扩展其处理能力。

1.4. OceanBase 数据库整体架构

OceanBase 数据库采用 Shared-Nothing 架构，各个节点之间完全对等，每个节点都有自己的 SQL 引擎、存储引擎，运行在普通PC服务器组成的集群之上，具备可扩展、高可用、高性能、低成本、云原生等核心特性。



OceanBase 数据库的一个集群由若干个节点组成。这些节点分属于若干个可用区（Zone），每个节点属于一个可用区。可用区是一个逻辑概念，表示集群内具有相似硬件可用性的一组节点，它在不同的部署模式下代表不同的含义。例如，当整个集群部署在同一个数据中心（IDC）内的时候，一个可用区的节点可以属于同一个机架，同一个交换机等。当集群分布在多个数据中心的时候，每个可用区可以对应于一个数据中心。每个可用区具有 IDC 和地域（Region）两个属性，描述该可用区所在的 IDC 及 IDC 所属的地域。一般地，地域指 IDC 所在的城市。可用区的 IDC 和 Region 属性需要反映部署时候的实际情况，以便集群内的自动容灾处理和优化策略能更好地工作。根据业务对数据库系统不同的高可用性需求，OceanBase 集群提供了多种部署模式，参见 高可用架构概述。

在 OceanBase 数据库中，一个表的数据可以按照某种划分规则水平拆分为多个分片，每个分片叫做一个表分区，简称分区（Partition）。某行数据属于且只属于一个分区。分区的规则由用户在建表的时候指定，包括hash、range、list等类型的分区，还支持二级分区。例如，交易库中的订单表，可以先按照用户 ID 划分为若干一级分区，再按照月份把每个一级分区划分为若干二级分区。对于二级分区表，第二级的每个子分区是一个物理分区，而第一级分区只是逻辑概念。一个表的若干个分区可以分布在一个可用区内的多个节点上。

为了能够保护数据，并在节点发生故障的时候不中断服务，每个分区有多个副本。一般来说，一个分区的多个副本分散在多个不同的可用区里。多个副本中有且只有一个副本接受修改操作，叫做主副本（Leader），其他副本叫做从副本（Follower）。主从副本之间通过基于 Multi-Paxos 的分布式共识协议实现了副本之间数据的一致性。当主副本所在节点发生故障的时候，一个从节点会被选举为新的主节点并继续提供服务。为了权衡成本和性能等因素，OceanBase 数据库还提供了多种副本类型，参见 数据分区和分区副本概述。

在集群的每个节点上会运行一个叫做 observer 的服务进程，它内部包含多个操作系统线程。节点的功能都是对等的。每个服务负责自己所在节点上分区数据的存取，也负责路由到本机的 SQL 语句的解析和执行。这些服务进程之间通过 TCP/IP 协议进行通信。同时，每个服务会监听来自外部应用的连接请求，建立连接和数据库会话，并提供数据库服务。关于 observer 服务进程的更多信息，参见 线程简介。

为了简化大规模部署多个业务数据库的管理并降低资源成本，OceanBase 数据库提供了独特的多租户特性。在一个 OceanBase 集群内，可以创建很多个互相之间隔离的数据库“实例”，叫做一个租户。从应用程序的视角来看，每个租户是一个独立的数据库。不仅如此，每个租户可以选择 MySQL 或 Oracle 兼容模式。应用连接到 MySQL 租户后，可以在租户下创建用户、database，与一个独立的 MySQL 库的使用体验是一样的。同样的，应用连接到 Oracle 租户后，可以在租户下创建 schema、管理角色等，与一个独立的 Oracle 库的使用体验是一样的。一个新的集群初始化之后，就会存在一个特殊的名为 sys 的租户，叫做系统租户。系统租户中保存了集群的元数据，是一个 MySQL 兼容模式的租户。

为了隔离租户的资源，每个 observer 进程内可以有多个属于不同租户的虚拟容器，叫做资源单元（UNIT）。每个租户在多个节点上的资源单元组成一个资源池。资源单元包括 CPU 和内存资源。

为了使 OceanBase 数据库对应用程序屏蔽内部分区和副本分布等细节，使应用访问分布式数据库像访问单机数据库一样简单，我们提供了 obproxy 代理服务。应用程序并不会直接与 OBServer 建立连接，而是连接 obproxy，然后由 obproxy 转发 SQL 请求到合适的 OBServer 节点。obproxy 是无状态的服务，多个 obproxy 节点通过网络负载均衡（SLB）对应用提供统一的网络地址。

1.5. OceanBase 数据库的发展历程

OceanBase 数据库是随着阿里巴巴电商业务的发展孕育而生，随着蚂蚁集团移动支付业务的发展而壮大，经过十多年各类业务的使用和打磨才终于破茧成蝶，推向了外部市场。本章节简述 OceanBase 数据库发展过程中一些里程碑意义的事件。

- **诞生**

  2010 年，OceanBase 创始人阳振坤博士带领初创团队启动了 OceanBase 项目。第一个应用是淘宝的收藏夹业务。如今收藏夹依然是 OceanBase 的客户。收藏夹单表数据量非常大，OceanBase 用独创的方法解决了其高并发的大表连接小表的需求。

- **关系数据库**

  早期的版本中，应用通过定制的 API 库访问 OceanBase 数据库。2012 年，OceanBase 数据库发布了支持 SQL 的版本，初步成为一个功能完整的通用关系数据库。

- **初试金融业务**

  OceanBase 进入支付宝（后来的蚂蚁集团），开始应用于金融级的业务场景。2014 年“双 11”大促活动，OceanBase 开始承担交易库部分流量。此后，新成立的网商银行把所有核心交易库都运行在 OceanBase 数据库上。

- **金融级核心库**

2016 年，OceanBase 数据库发布了架构重新设计后的 1.0 版本，支持了分布式事务，提升了高并发写业务中的扩展，同时实现了多租户架构，这个整体架构延续至今。同时，到 2016 年“双 11”时，支付宝全部核心库的业务流量 100% 运行在 OceanBase 数据库上，包括交易、支付、会员和最重要的账务库。

- **走向外部市场**

  2017 年，OceanBase 数据库开始试点外部业务，成功应用于南京银行。

- **商业化加速**

  2018 年，OceanBase 数据库发布 2.0 版本，开始支持 Oracle 兼容模式。这一特性降低应用改造适配成本，在外部客户中快速推广开来。

- **登峰造极**

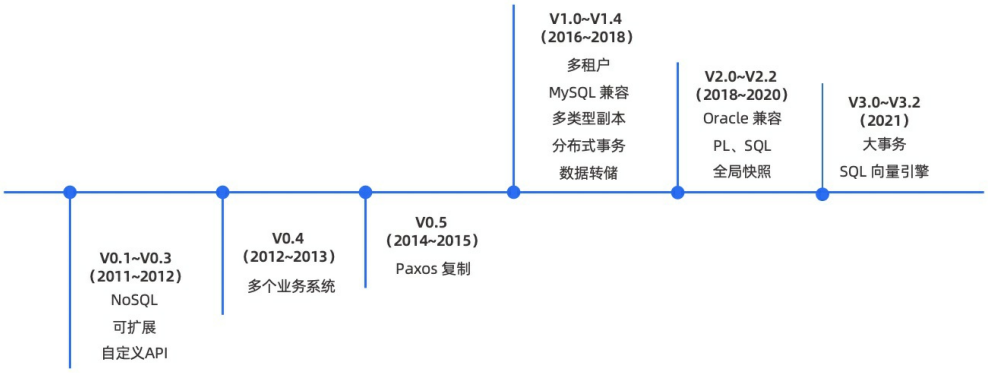
  2019 年，OceanBase 数据库 V2.2 版本参加代表 OLTP 数据库最权威的 TPC-C 评测，以 6000 万 tpmC 的成绩登顶世界第一。随后，在 2020 年，又以 7 亿 tpmC 刷新纪录，截止目前依然稳居第一。这充分证明了 OceanBase 数据库优秀的扩展性和稳定性。OceanBase 数据库是第一个也是截止目前唯一一个上榜 TPC-C 的中国数据库产品。

- **HTAP 混合负载**

  2021 年，OceanBase 数据库 V3.0 基于全新的向量化执行引擎，在 TPC-H 30000GB 的评测中以 1526 万 QphH 的成绩刷新了评测榜单。这标志着 OceanBase 数据库一套引擎处理 AP 和 TP 混合负载的能力取得了基础性的突破。

- **开源开放**

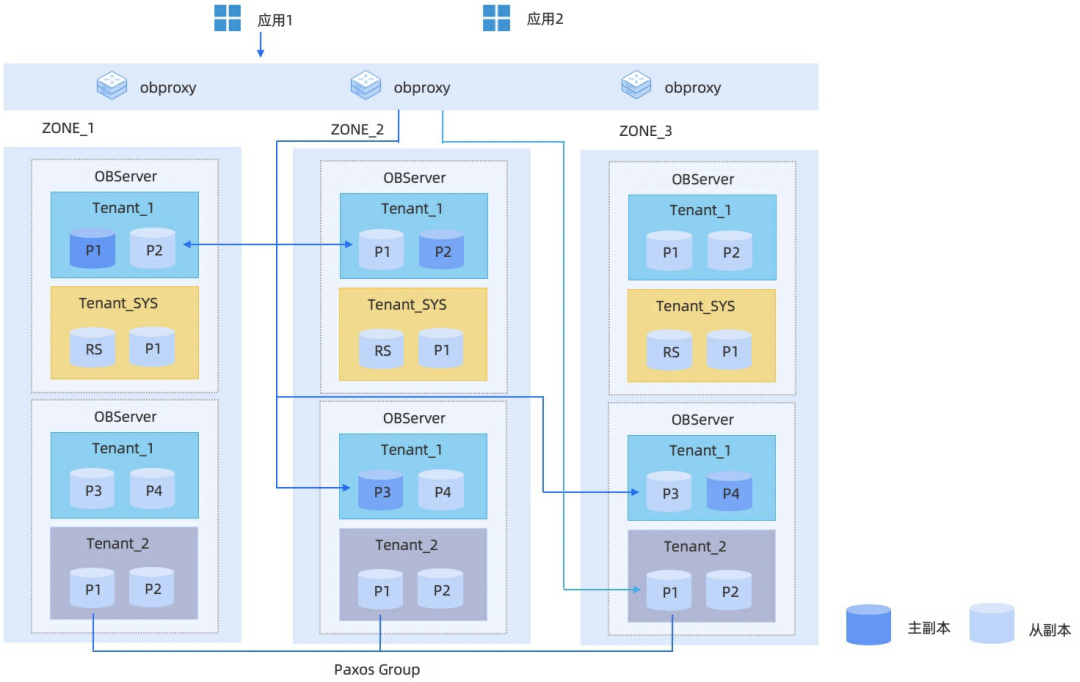
  2021 年六一儿童节，OceanBase 数据库宣布全面开源，开放合作，共建生态。



2.多租户架构

2.1. 多租户架构概述

OceanBase 数据库采用了单集群多租户设计，天然支持云数据库架构，支持公有云、私有云、混合云等多种部署形式。



OceanBase 数据库通过租户实现资源隔离，让每个数据库服务的实例不感知其他实例的存在，并通过权限控制确保租户数据的安全性，配合 OceanBase 数据库强大的可扩展性，能够提供安全、灵活的 DBaaS 服务。

租户是一个逻辑概念。在 OceanBase 数据库中，租户是资源分配的单位，是数据库对象管理和资源管理的基础，对于系统运维，尤其是对于云数据库的运维有着重要的影响。租户在一定程度上相当于传统数据库的“实例”概念。租户之间是完全隔离的。在数据安全方面，OceanBase 数据库不允许跨租户的数据访问，以确保用户的数据资产没有被其他租户窃取的风险。在资源使用方面，OceanBase 数据库表现为租户“独占”其资源配额。总体上来说，租户（tenant）既是各类数据库对象的容器，又是资源（CPU、Memory、IO 等）的容器。

2.2. 兼容模式

OceanBase 数据库在一个系统中可同时支持 MySQL 模式和 Oracle 模式两种模式的租户。用户在创建租户时，可选择创建 MySQL 兼容模式的租户或 Oracle 兼容模式的租户，租户的兼容模式一经确定就无法更改，所有数据类型、SQL 功能、内部视图等相应地与 MySQL 数据库或 Oracle 数据库保持一致。

MySQL 模式

MySQL 模式是为降低 MySQL 数据库迁移至 OceanBase 数据库所引发的业务系统改造成本，同时使业务数据库设计人员、开发人员、数据库管理员等可复用积累的 MySQL 数据库技术知识经验，并能快速上手 OceanBase 数据库而支持的一种租户类型功能。OceanBase 数据库兼容 MySQL 5.5/5.6/5.7，基于 MySQL 的应用能够平滑迁移。

Oracle 模式

OceanBase 数据库从 V2.x.x 版本开始支持 Oracle 兼容模式。Oracle 模式是为降低 Oracle 数据库迁移 OceanBase 数据库的业务系统改造成本，同时使业务数据库设计开发人员、数据库管理员等可复用积累的 Oracle 数据库技术知识经验，并能快速上手 OceanBase 数据库而支持的一种租户类型功能。Oracle 模式目前能够支持绝大部分的 Oracle 语法和过程性语言功能，可以做到大部分的 Oracle 业务进行少量修改后的自动迁移。

2.3. 系统租户

系统租户也称为 sys 租户，是 OceanBase 数据库的系统内置租户。

系统租户主要有以下几个功能：

- 系统租户承载了所有租户的元信息存储和管理服务。例如，系统租户下存储了所有普通租户系统表的对象元数据信息和位置信息。
- 系统租户是分布式集群集中式策略的执行者。例如，只有在系统租户下，才可以执行轮转合并、删除或创建普通租户、修改系统配置项、资源负载均衡、自动容灾处理等操作。
- 系统租户管理和维护集群资源。例如，系统租户下存储了集群中所有 OBServer 的信息和 Zone 的信息。

系统租户在集群自举过程中创建，系统租户信息和资源的管理都是在 RootService 服务上完成，RootService简称 RS，是启动在系统租户下 `__all_core_table` 表上主副本上的一组服务，`__all_core_table` 是系统的 1 号表，系统中所有的表都可以通过 1 号表索引到。

系统表分类

系统租户是一个 MySQL 兼容租户，系统租户下的系统表主要分为以下几类：

- 对象元数据表

  系统租户下存储了所有系统表的元数据。例如，`__all_core_table` 中存储了 `__all_table` 表的元数据，`__all_table` 表中存储了其他系统表的元数据。

- 分区位置信息表

  系统租户存储了系统表的位置信息。例如，`__all_core_table` 中存储了 `__all_root_table` 的位置信息，`__all_root_table` 中记录了其他系统表的位置信息。

- 集群资源相关表

  系统租户维护了所有机器的位置信息和分布信息。例如，在 `__all_zone` 表中记录了所有 Zone 的信息，在 `__all_server` 表中维护了 OBServer 的信息。

- 租户元信息和资源相关表

  在系统租户下可以看到所有租户的元数据信息。例如，在 `__all_tenant` 表中可以看到集群中所有的租户以及租户的分布信息。

RS 服务

RootService 的功能主要包括集群自举、集群资源管理、DDL 操作以及分布式集群集中式策略执行。

RootService 各功能的特点如下：

- 集群自举

  集群自举是在 OBServer 启动成功后，创建系统租户和初始化配置的过程，也称 Bootstrap。

  系统自举时需要指定 RootService 的位置信息，Bootstrap 命令在 RootService 位置上创建 `__all_core_table` 。`__all_core_table` 的 Leader 所在的 OBServer 自动提供 RootService 服务。

  RS 启动后就可以创建系统租户、系统表、初始系统数据和集群配置。

- 集群资源管理

  集群资源管理主要包括：

  - Zone 的管理

    在系统租户下，可以新增一个 Zone，删除一个 Zone、修改 Zone 的信息，或者停止一个 Zone 的服务。

  - Unit 管理

    Unit 是资源的最小分隔单位。一组 Unit 构成一个资源池，一个资源池可以被分配给一个租户，一个租户可以有多个资源池。在系统租户下，用户可以通过调整 Unit 规格来调整资源池大小从而调整租户资源。

    通过调整 Unit 规格来调整租户资源的详细介绍信息请参见 租户资源垂直扩缩容。

- OBServer 管理

  每个 OBServer 都需要通过心跳来与 RS 保持通信。RS 会根据心跳信息感知 OBServer 是否在线以及是否可以提供服务。系统租户下可以进行 OBServer 的增加、删除或者停止服务等操作。

- DDL 操作

  所有的 DDL 操作都会在 RS 上执行。

- 分布式系统集中式策略执行

  主要包括：

  - 主备库集群角色切换

    OceanBase 集群包括一个主集群和多个备集群，可以在系统租户下操作无损切换和有损切换。

  - 合并管理

    在系统租户下维护各个版本的合并信息，RS 会根据这些信息进行合并调度。

  - 集群级配置项只能在系统租户下修改

    例如：`enable_rebalance` 、`enable_rereplication` 等。

  - 租户管理

    在系统租户下可以增加、修改和删除普通租户。

高级特性

除了上述特点外，系统租户还具备以下高级特性：

- 全局虚拟表

系统租户是一个 MySQL 兼容租户，相关视图与 MySQL 兼容。但是在系统租户下存在一些全局视图，例如，系统租户下的 `__all_virtual_meta_table` 表可以查询到所有租户下 `__all_tenant_meta_table` 表的内容；`__all_virtual_ddl_operation` 表可以查询到所有租户下 `__all_ddl_operation` 表的内容。

- 租户切换

  在系统租户下，您可以通过命令切换到任意普通租户下，但是只有普通租户下系统表的读写权限。一个普通租户不能切换到另一个普通租户。

- `ALTER SYSTEM` 命令

  `ALTER SYSTEM` 命令一般用于集群级别的操作，例如主备库切换、配置项修改等。

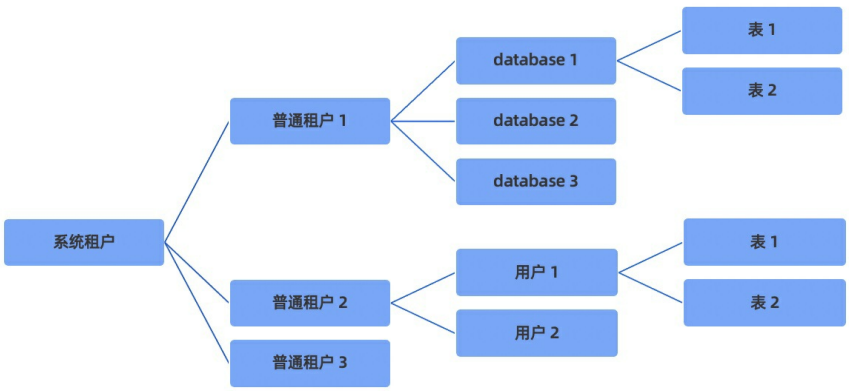
2.4. 普通租户

普通租户与通常所见的数据库管理系统相对应，可以被看作是一个数据库实例。它由系统租户根据业务需要所创建出来。

普通租户具备一个实例所应该具有的所有特性，主要包括：

- 可以创建自己的用户
- 可以创建数据库（database）、表（table）等所有客体对象
- 有自己独立的系统表和系统视图
- 有自己独立的系统变量
- 数据库实例所具备的其他特性

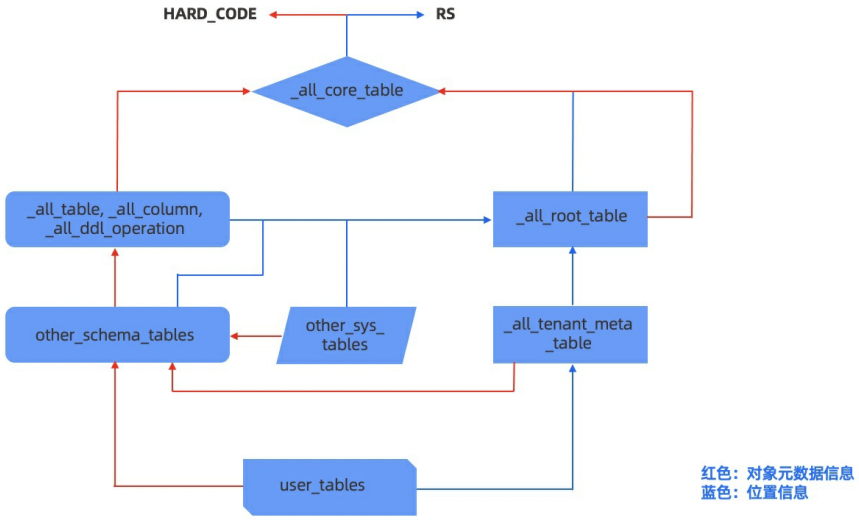
所有用户数据的元信息都存储在普通租户下，所以每个租户都有自己的名字空间，并且彼此隔离不可访问。系统租户管理所有普通租户，系统租户与普通租户之间的层级关系如下图所示。



租户级系统表

虽然普通租户与系统租户是独立且隔离的，但普通租户无法完成自举。普通租户系统表的元数据信息是引用了系统租户的元数据信息；普通租户系统表的位置信息存储在系统租户的 `__all_root_table` 表中。

普通租户系统表的对象元数据信息和位置信息的分布如下图所示。



根据上图可知：

- 副本位置信息的发现

  租户下用户表的位置信息存储在本租户的 `__all_tenant_meta_table` 表中。所有租户的系统表的位置信息都存储在系统租户下的 `__all_root_table` 表中。`__all_root_table` 表的位置信息存储在 `__all_core_table` 表中。`__all_core_table` 表的位置信息常驻在 RS 的内存中。

  当需要发现一个用户表分区的副本位置时，需要找到 `__all_tenant_meta_table` 表，而找到 `__all_tenant_meta_table` 表需要找到 `__all_root_table` 表，再根据 `__all_root_table` 表找到 `__all_core_table` 表的位置，最后追溯到 RS 的位置。

- 对象元数据信息

  租户下用户表的对象元数据信息存储在本租户下的对象元数据相关的表中。普通租户下系统表的对象元数据信息是引用了系统租户的系统表。

  例如，租户下的 `__all_table` 表存储了本集群下所有的用户表的信息，租户下的 `__all_table` 表的对象元数据信息是直接引用了系统租户的 `__all_table` 表，而系统租户的 `__all_table` 表的对象元数据信息是存储在 `__all_core_table` 表中，`__all_core_table` 的表结构为硬编码。

创建用户

在普通租户下创建的用户，只能登录到本租户，对其他租户不可见。

对于 MySQL 兼容模式的租户，可以从 `mysql.user` 视图中查询用户信息。

对于 Oracle 兼容模式的租户，可以从 `ALL_USERS` 视图中查询用户信息。

创建用户表

在普通租户下可以创建表，创建后对其他租户不可见。

对于 MySQL 兼容模式的租户，可以从 `information_schema.tables` 视图中查询本租户所有用户表的信息。

对于 Oracle 兼容模式的租户，可以从 `ALL_TABLES` 视图中查询本租户所有用户表的信息。

修改系统变量

普通租户只能在本租户下修改本租户的系统变量。

对于 MySQL 兼容模式的租户，可以从 `information_schema.global_variables` 和 `information_schema.session_variables` 视图中查询系统变量信息。也可以通过 `SHOW VARIABLES` 语句查询。

对于 Oracle 兼容模式的租户，可以通过 `SHOW VARIABLES` 语句来查询本租户所有用户表的信息。

位置信息

每个普通租户都分别包含一张 `__all_tenant_meta_table` 表，记录租户下用户表的物理位置元信息。

2.5. 租户与资源管理

2.5.1. 租户的资源管理

OceanBase 数据库是多租户的数据库系统，一个集群内可包含多个相互独立的租户，每个租户提供独立的数据库服务。在 OceanBase 数据库中，使用资源配置（unit_config）、资源池（Resource Pool）和资源单元（Unit）等三个概念，对各租户的可用资源进行管理。

租户资源的创建

创建租户前，需首先确定租户的资源配置、使用资源范围等。租户创建的通用流程如下：

- 创建资源配置
- 创建资源池
- 创建租户

创建资源配置

资源配置是描述资源池的配置信息，用来描述资源池中每个资源单元可用的 CPU、内存、存储空间和 IOPS 等的规格。修改资源配置可动态调整资源单元的规格。这里需要注意，资源配置指定的是对应资源单元能够提供的服务能力，而不是资源单元的实时负载。

创建资源配置的示例语句如下：

```
obclient> CREATE RESOURCE UNIT uc1 MAX_CPU 5,MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMORY '32G',
MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
```

在该语句中，当前仅 `MIN_CPU` 、 `MAX_CPU` 、 `MIN_MEMORY` 和 `MAX_MEMORY` 会产生实际作用，其他字段需要填写，但系统不会做任何处理。其中， `MIN_CPU` 和 `MIN_MEMORY` 表示使用该资源配置的资源单元能够提供 CPU 或 Memory 的下限。

创建资源池

资源池由若干个资源单元组成，通过给资源池指定资源配置，可指定资源池下各资源单元的物理资源。创建资源池的示例语句如下：

```
obclient> CREATE RESOURCE POOL rp1 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('zone1', 'zone2');
```

在该示例语句中，创建了一个资源池 `rp1` ，该资源池有三个要素，三个要素缺一不可：

- `UNIT 'uc1'` 表示为该资源池指定的资源配置为 `uc1` ，该资源池下的每个资源单元使用 `uc1` 的规格进行配置。
- `ZONE_LIST ('zone1','zone2')` 是为资源池指定的使用范围，表示该资源池要在 `zone1` 和 `zone2` 上创建资源单元。
- `UNIT_NUM 2` 是为资源池指定资源单元的个数，表示在 `ZONE_LIST` 内的每个 Zone 上都创建 2 个资源单元。

任何一个资源单元一定需要放置在资源足够容纳下它的物理机上，并且单台物理机上最多能放置同一个资源池下的一个资源单元，如果 `zone1` 或 `zone2` 上的物理机个数小于 2，或物理机的资源小于 `uc1` 的规格，上述创建资源池的示例语句将无法执行成功，资源池最终会创建失败。

创建租户

创建好资源池后，可以继续创建租户，一个资源池仅能属于一个租户，一个租户可拥有一个或多个资源池，租户在同一个 Zone 上仅能有一个资源池，即属于同一个租户的多个资源池的 `ZONE_LIST` 彼此不允许有交集。一个租户的所有资源池下的全部资源单元的集合描述了该租户可以使用的全部物理机资源。

创建租户的示例语句如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('z1', 'z2');
obclient>CREATE RESOURCE POOL pool2 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z3');
obclient>CREATE TENANT tt resource_pool_list=('pool1','pool2');
```

示例语句中首先创建了两个资源池 `pool1` 和 `pool2` ，并在此基础上创建了租户 `tt` ，租户 `tt` 在 `z1` 、 `z2` 上各有 2 个资源单元，在 `z3` 上有 1 个资源单元，各资源单元的规格均使用 `uc1` 指定的资源配置。

租户资源的变更

租户的资源变更通过调整租户下各资源的三要素即可完成，具体表现为，可以分别单独调整资源池的 Unit 配置、 `ZONE_LIST` 和 `UNIT_NUM` 来进行租户资源的变更。除此之外，还支持对资源池进行 Split 或 Merge 两个特殊的变更操作。

修改资源配置

修改资源池的资源配置即直接调整资源配置的 CPU 或 Memory 等的值，进而直接影响租户在该资源池上的资源规格和服务能力。
修改资源池的资源配置的示例语句如下：

```
obclient> CREATE RESOURCE UNIT uc1 MAX_CPU 5, MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMORY '32G'
, MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
obclient> CREATE RESOURCE POOL pool1 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('z1', 'z2');
obclient> CREATE RESOURCE POOL pool2 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z3');
obclient> CREATE TENANT tt resource_pool_list=('pool1','pool2');
obclient> ALTER RESOURCE UNIT uc1 MAX_CPU 6, MIN_MEMORY '36G';
```

示例中，资源池 `pool1` 和 `pool2` 的资源配置是 `uc1`，语句 `ALTER RESOURCE UNIT uc1 MAX_CPU 6, MIN_MEMORY '36G';` 将资源配置 `uc1` 的 `MAX_CPU` 调整为 `6`；`MIN_MEMORY` 调整为 `36G`，其他配置选项不变。通过调整资源配置的各个选项，可调整租户的资源池在对应 Zone 上的资源规格，进而影响租户的服务能力。

切换资源配置

切换资源池的资源配置可以调整资源池下每个资源单元的资源规格，进而调整租户在该资源池上的资源规格和服务能力。

切换资源池的资源配置的示例语句如下：

```
obclient> ALTER RESOURCE POOL rp1 UNIT 'uc2';
```

假设资源池 `rp1` 之前的资源配置为 `uc1`，则示例语句将 `rp1` 的资源配置从 `uc1` 变更为 `uc2`。理论上OceanBase 数据库支持对资源规格 `MIN_CPU`、`MAX_CPU`、`MIN_MEMORY` 以及 `MAX_MEMORY` 同时修改。但通常情况下，修改资源配置和切换资源配置都能对租户的服务能力进行调整。对应到租户层面，实际上是调整了租户资源单元的规格。对资源规格的修改通常有以下两种场景：

- 调大资源规格

  调大资源规格主要用在租户的资源扩容场景中，可分别对 CPU 和 Memory 进行资源扩容。

  示例 1：

  ```
  obclient> CREATE RESOURCE UNIT u_c0 MAX_CPU 5, MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMORY '3
  2G', MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
  obclient> CREATE RESOURCE POOL pool1 unit='u_c0', unit_num=3, zone_list=('z1','z2','z3');
  obclient> ALTER RESOURCE UNIT u_c0 MAX_CPU 10, MIN_CPU 8, MAX_MEMORY '72G', MIN_MEMORY '6
  4G';
  ```

  上述示例 1 中创建了一个资源配置 `u_c0`，并创建了一个资源池 `pool1`，`pool1` 使用 `u_c0` 作为自己的资源配置，之后调大 `u_c0` 的 `MIN_CPU`、`MAX_CPU`、`MIN_MEMORY` 或 `MAX_MEMORY`。该调整旨在调大资源池 `pool1` 的资源规格，目的是提高相应租户的服务能力。

  示例 2:

```
obclient> CREATE RESOURCE UNIT u_c0 MAX_CPU 5, MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMORY '3
2G', MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
obclient> CREATE RESOURCE UNIT u_c1 MAX_CPU 10, MIN_CPU 8, MAX_MEMORY '72G', MIN_MEMORY '
64G', MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
obclient> CREATE RESOURCE POOL pool1 unit='u_c0', unit_num=3, zone_list=('z1','z2','z3');
obclient> ALTER RESOURCE POOL pool1 unit='u_c1';
```

上述示例 2 中创建了两个资源配置 `u_c0` 和 `u_c1` 。并创建了一个资源池 `pool1` ， `pool1` 最初使用 `u_c0` 作为自己的资源配置，之后调整为资源配置 `u_c1` 。该调整旨在调大资源池 `pool1` 的资源规格，目的是提高相应租户的服务能力。

- 调小资源规格

  根据调大资源规格的示例，系统还支持调小资源规格，方法与调大资源规格的方法相同。

变更 UNIT_NUM

调整资源池的 `UNIT_NUM` 可以调整资源池下每个 Zone 内资源单元的数量，进而通过调整资源单元的数量来提高或降低该租户在对应 Zone 上的服务能力。

调整资源池的 `UNIT_NUM` 的示例语句如下：

```
obclient> CREATE RESOURCE POOL rp1 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('zone1', 'zone2');
obclient> ALTER RESOURCE POOL rp1 UNIT_NUM 3; // 调大unit num
obclient> ALTER RESOURCE POOL rp1 UNIT_NUM 2; // 调小unit num
obclient> ALTER RESOURCE POOL rp1 UNIT_NUM 1 DELETE UNIT = (1001, 1003); // 指定资源单元调小u
nit num
```

变更 `UNIT_NUM` 也分为两类，即调大 `UNIT_NUM` 和调小 `UNIT_NUM` 。其中，

`ALTER RESOURCE POOL rp1 UNIT_NUM 3;` 是调大 `UNIT_NUM` ；

`ALTER RESOURCE POOL rp1 UNIT_NUM 2;` 和

`ALTER RESOURCE POOL rp1 UNIT_NUM 1 DELETE UNIT = (1001, 1003);` 为调小 `UNIT_NUM` 。

变更 ZONE_LIST

调整资源池的 `ZONE_LIST` 可以调整资源池在 Zone 维度的使用范围，从而调整租户数据在 Zone 维度的服务范围。

调整资源池的 `ZONE_LIST` 的示例语句如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT_NUM=3, UNIT='unit_config', ZONE_LIST=('z1','z2','
z3');
obclient> CREATE RESOURCE POOL pool2 UNIT_NUM=3, UNIT='unit_config', ZONE_LIST=('z1','z2','
z3');
obclient> ALTER RESOURCE POOL pool1 ZONE_LIST=('z1','z2','z3','z4');
obclient> ALTER RESOURCE POOL pool2 ZONE_LIST=('z1','z2');
```

变更 `ZONE_LIST` 也分为两类，即扩大 Zone 维度的使用范围和缩小 Zone 维度的使用范围。其中，`ALTER RESOURCE POOL pool1 ZONE_LIST=('z1','z2','z3','z4');` 是扩大 Zone 维度的使用范围；`ALTER RESOURCE POOL pool2 ZONE_LIST=('z1','z2');` 是缩小 Zone 维度的使用范围。

分裂资源池

分裂资源池操作可将一个资源池分裂为多个资源池，分裂资源池操作的基本语法和示例如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z1','z2','z3');
obclient> ALTER RESOURCE POOL pool1 SPLIT INTO ('pool10','pool11','pool12') ON ('z1','z2','
z3');
obclient> ALTER RESOURCE POOL pool10 UNIT='uc1';
obclient> ALTER RESOURCE POOL pool11 UNIT='uc2';
obclient> ALTER RESOURCE POOL pool12 UNIT='uc3';
```

合并资源池

合并资源池操作可将多个资源池合并为一个资源池，合并资源池的基本语法和示例如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z1');
obclient> CREATE RESOURCE POOL pool2 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z2');
obclient> CREATE RESOURCE POOL pool3 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z3');
obclient> ALTER RESOURCE POOL MERGE ('pool1','pool2','pool3') INTO ('pool0');
```

2.5.2. 租户间的资源隔离

OceanBase 数据库是多租户的数据库系统，为了确保租户间不出现资源争抢保障业务稳定运行，OceanBase 数据库针对租户间的资源进行了隔离。

OceanBase 数据库中把 Unit 当作给租户分配资源的基本单位，一个 Unit 可以类比于一个 Docker 容器。一个 OBServer 上可以创建多个 Unit，在 OBServer上每创建一个 Unit 都会占用一部分该 OBServer 的 CPU、内存等物理资源，OBServer 的资源分配情况会记录在内部表中以便 DBA 查看。

一个租户可以在多个 OBServer上放置多个 Unit，但一个特定的租户在某个 OBServer 上只能有一个 Unit。一个租户的多个 Unit 相互独立，OceanBase 数据库目前没有汇总多个 Unit 的资源占用进行全局的资源控制，具体来讲，不会因为一个租户在某个 OBServer 上的资源没得到满足，就让它在另一个 OBServer 上去抢其它租户的资源。

所谓资源隔离，就是 OBServer 控制本地多个 Unit 间的资源分配的行为，它是 OBServer 本地的行为。类似的技术是 Docker 和虚拟机，但 OceanBase 数据库并没有依赖 Docker 或虚拟机技术，而是在数据库内部实现资源隔离。

OceanBase 数据库租户隔离的优势

相比 Docker 和虚拟机，OceanBase 数据库的租户隔离更加轻量，并且便于实现优先级等高级特性。
从 OceanBase 数据库的需求来看，Docker 或虚拟机的主要有以下几个问题：

- Docker 或虚拟机运行环境的开销太重，OceanBase 数据库需要支持轻量级租户。
- Docker 或虚拟机规格变化以及迁移开销比较大，OceanBase 数据库希望租户的规格变化和迁移尽量快。
- Docker 或虚拟机不便于租户间的资源共享，例如，对象池的共享。
- Docker 或虚拟机的资源隔离很难定制，例如，租户内的优先级支持。

除此之外，Docker 或虚拟机的实现不便于暴露统一视图。

普通用户的视角看隔离效果

从普通用户的角度，我们可以看到的隔离效果如下：

- 内存完全隔离。

  具体包括：

  - SQL 执行过程各种算子使用的内存是分离的，一个租户的内存耗尽不会影响到另一个租户。

    SQL 执行过程的各种算子的详细信息请参见 用户接口和查询语言 章节。

  - Block Cache 和 MEMTable 是分离的，一个租户的内存耗尽不会影响到另一个租户的写入和读取。

    Block Cache 的详细介绍信息请参见 多级缓存。

    MEMTable 的详细介绍信息请参见 MEMTable。

- CPU 通过用户态调度实现隔离。

  一个租户能使用的 CPU 资源是由 Unit 规格决定的，不过 OBServer 目前是允许租户 CPU 超卖的。

- 大部分数据结构是分离的。

  具体包括：

  - SQL 的 Plan Cache 是租户分离的，一个租户的 Plan Cache 淘汰不会影响另一个租户。
  - SQL 的 Audit 表是分离的，一个租户的 QPS 太高，不会冲洗掉另一个租户的 Audit 信息。

- 事务相关的数据结构是分离的。

  具体包括：

  - 一个租户的行锁挂起，不会影响到其他租户。
  - 一个租户的事务挂起，不会影响到其他租户。
  - 一个租户的回放出问题，不会影响到其它租户。

- Clog 是共享的。

  一个 OBServer 上的不同租户共享 Clog 文件，这个设计主要是为了让事务的 Group Commit 能有更好的效果。

资源分类

从设计上，一个 Unit 可以指定 CPU、Memory、IOPS、Disk Size、Session Number 这 5 种资源的下限和上限。

```
obclient> CREATE RESOURCE UNIT box1 MIN_CPU 4, MAX_CPU 4, MIN_MEMORY 34359738368, MAX_MEMOR
Y 34359738368, min_iops 128, MAX_IOPS 128, MIN_DISK_SIZE '5G', MAX_DISK_SIZE '5G', MIN_SESS
ION_NUM 64, MAX_SESSION_NUM 64;
```

目前 IOPS、Disk Size、Session Number 这三种资源是没有被管理的，即目前的资源隔离主要考虑的是CPU 和 Memory。

OBServer 的可用 CPU

OBServer 在启动时会探测物理机或容器的在线 CPU 个数，如果 OBServer 探测得不准确（例如，在容器化环境里)，您也可以通过 `cpu_count` 配置项来指定。

由于 OBServer 会为后台线程预留两个 CPU，故实际可以分给租户的 CPU 总数会少两个。

OBServer 的可用 Memory

OBServer 在启动时会探测物理机或容器的内存，由于 OBServer 需要为其它进程预留一部分内存，故 observer 进程的可用内存等于 **物理内存** `* memory_limit_percentage` 。您也可以通过配置项 `memory_limit` 直接配置 observer 进程可用的总内存大小。

observer 进程可用的内存需要进一步扣除掉内部共用模块的那一部分，这部分内存大小由配置项 `system_memory` 指定，剩下的内存才是租户可用的总内存。

关于内存配置的更多信息，请参见 内存管理 章节。

查看每个 OBServer 的可用资源

您可以通过 `oceanbase.__all_virtual_server` 表查看每个 OBServer 的可用资源，示例如下：

```
obclient> SELECT * FROM __all_virtual_server_stat \G
*************************** TODO 1. row ***************************
               svr_ip: 10.10.10.1
             svr_port: 20900
                 zone: z1
            cpu_total: 14
         cpu_assigned: 2.5
 cpu_assigned_percent: 17
            mem_total: 53687091200
         mem_assigned: 12884901888
 mem_assigned_percent: 24
           disk_total: 10737418240
        disk_assigned: 10737418240
disk_assigned_percent: 100
             unit_num: 1
   migrating_unit_num: 0
       merged_version: 1
         leader_count: 1283
                 load: 0.21379327157484151
           cpu_weight: 0.4266211604095563
        memory_weight: 0.5733788395904437
          disk_weight: 0
                   id: 1
           inner_port: 20901
        build_version: 3.2.1_1-3c4c42fc31ba322cd7dfd441a8a71f648835eced(Sep  1 2021 16:16:0
6)
        register_time: 0
  last_heartbeat_time: 1631074048327156
block_migrate_in_time: 0
   start_service_time: 1631073245084853
    last_offline_time: 0
            stop_time: 0
 force_stop_heartbeat: 0
         admin_status: NORMAL
     heartbeat_status: alive
      with_rootserver: 1
       with_partition: 1
           mem_in_use: 0
          disk_in_use: 12582912
      clock_deviation: -11
    heartbeat_latency: 182
    clock_sync_status: SYNC
         cpu_capacity: 14
     cpu_max_assigned: 5
         mem_capacity: 53687091200
     mem_max_assigned: 16106127360
 ssl_key_expired_time: 0
```

资源隔离

CPU 和内存的超卖

如果一个 OBServer 上放置的所有 Unit 的 `MAX_CPU` 的值加起来超过这个 OBServer 的可用 CPU 总数，则表示该 OBServer 上的 CPU 是超卖的。同理，如果所有 Unit 的 `MAX_MEMORY` 加起来超过该 OBServer 可以分配的 Memory，则表示该 OBServer 上的 Memory 是超卖的。

OBServer 资源超卖的比例受配置项 `resource_hard_limit` 的控制，假设 `resource_hard_limit=200`，那就意味着可以超卖成 2 倍，即 16 个 CPU 可以超卖成 32 个，16G 的内存可以超卖成 32G。

超卖是通过牺牲稳定性来获取更高的资源利用率的方案，是否要开启超卖需要根据应用特性和应用的 SLA 要求仔细评估。例如，一个比较适合的场景就是业务的研发测试环境。

注意，即使在分配资源的时候没有超卖，在实现 CPU 资源隔离时，Unit 可以使用的 CPU 并没有严格限制在 `MAX_CPU` 的值，这个意义上的 CPU 默认是超卖的。更多 CPU 超卖的介绍信息可参见本文中的 **CPU 隔离**。

内存隔离

内存是个刚性的资源，也就是内存一旦被占用了，很难保证能快速收回。所以内存是不适合超卖的。

Unit 的内存上限由 `MAX_MEMORY` 决定，Unit 的 `MIN_MEMORY` 仅用来决定 Block Cache 的挤占时机，除此之外没有其它用途。

生产系统中推荐将 `MIN_MEMORY` 和 `MAX_MEMORY` 设置为相等的值。

CPU隔离

相比于内存，CPU 是更弹性的资源，所以 OBServer 目前允许一个 Unit 使用的物理 CPU 超过分配给它的配额。

在 OceanBase 数据库 V3.1.x 版本之前，主要通过控制线程数来控制 CPU 的占用；在 OceanBase 数据库 V3.1.x 及之后的版本，允许配置 Cgroup 来控制 CPU 的占用。

线程分类

OBServer 会启动很多不同功能的线程，细化的分类请参考ObServer线程的相关章节，本节按照最粗略的标准可以分为以下两类：

- 一类是处理 SQL 和事务提交的线程，统称为 Worker 线程。
- 其余的是处理网络 IO、磁盘 IO、Compaction 以及定时任务的线程。

Worker 线程是分租户的，非 Worker 线程是所有租户共享的。本节的租户 CPU 隔离都是针对 Worker线程的。

基于线程数的 CPU 隔离

Unit 的 CPU 隔离是通过一个 Unit 的活跃 Worker 线程数实现的。

由于 SQL 执行过程中可能会有 IO 等待、锁等待等，所以一个线程无法用满一个物理 CPU，故在缺省配置下，OBServer 会给每个 CPU 启动 4 个线程，4 这个倍数可以通过配置 `cpu_quota_concurrency` 来控制。

这就意味着如果一个 Unit 的 `MAX_CPU` 是10，那么它能同时运行的活跃线程是 40，最大物理 CPU 的占用是 400%，也就是 10 个 CPU 被超卖成了 40 个 CPU。

基于 Cgroup 的 CPU 隔离

开启 Cgroup 后最大的变化是不同租户的 Worker 线程放到不同的 Cgroup 目录内，租户间的 CPU 隔离效果会更好。最后的隔离效果如下：

- 如果一个 OBServer 上只有一个租户负载很高，其余租户比较空闲，那么这个负载高的租户的 CPU 可以超过 `MAX_CPU` 的限制，超卖倍数仍然受 `cpu_quota_concurrency` 控制，这个行为主要是为了兼容。
- 延续上面的场景，如果有多个空闲的租户的负载上升了，导致物理 CPU 不够了，Cgroup 会按照权重分配时间片。

大查询处理

我们认为相比于大查询，让短查询尽快返回对用户更有意义，即大查询的查询优先级更低，当大查询和短查询同时争抢 CPU 时，系统会限制大查询的 CPU 使用。

当一个线程执行的 SQL 查询耗时太长，这条查询就会被判定为大查询，一旦判定为大查询，执行大查询的 Worker 会等在一个 Pthread Condition 上，这样就为其它的 Worker 线程让出了 CPU。

具体实现上，OBServer 在代码中插入了很多检查点，Worker 线程在运行过程中会通过检查点定期检查自己的状态，如果判断应该挂起，那么线程就会等待在一个 Pthread Conditon 上，等到合适的时机再被唤醒。

如果同时有大查询和小查询，大查询最多占用 30% 的 Worker 线程，30% 这个百分比值可以通过配置项 `large_query_worker_percentage` 来设置。

有两点需要说明：

- 当没有小查询的时候，大查询可以用到 100% 的 Worker 线程。只有当同时有大查询和小查询时，30% 的比例才生效。
- 一个 Worker 因为执行大查询被挂起时，作为补偿，系统可能会新创建一个 Worker 线程，但是总的 Worker 线程不能超过 `MAX_CPU` 的 10 倍，10 这个倍数可以通过配置项 `workers_per_cpu_quota` 来设置。

提前识别大查询

由于 OBServer 挂起一个大查询线程，就会启动一个新的 Worker 线程，但是如果有大量大查询涌入，OBServer 新创建的线程还是被用来处理大查询，很快达到 Worker 数上限，在这批大查询消耗完之前就没有机会再处理短查询了。

为了优化这个场景，OBServer 会在 SQL 开始执行之前预判它是不是大查询，预判的本质就是估计 SQL 的执行时间。预判主要依据以下假设场景：如果两条 SQL 的执行 Plan 是一样的，可以猜测它们的执行时间也是相似的，这样就可以用 Plan 最近的执行时间来判断 SQL 会不会是大查询。

如果某条 SQL 被预判为大查询，那么该查询就会被放入一个特殊的大查询队列，其 Worker 线程会被释放，系统就会接着执行后面的请求了。

监控项和日志

在日志中搜索 `dump tenant info` 关键字，可看到租户的规格、线程、队列及请求统计等信息，且这条日志每个租户每 10s 打印一次。

日志示例：

```
grep 'dump tenant info.*tenant={id:1002' log/observer.log.*[2021-05-10 16:56:22.564978] INF
O  [SERVER.OMT] ob_multi_tenant.cpp:803 [48820][2116][Y0-0000000000000000] [lt=5] dump tena
nt info(tenant={id:1002, compat_mode:1, unit_min_cpu:"1.000000000000000000e+01", unit_max_c
pu:"1.500000000000000000e+01", slice:"0.000000000000000000e+00", slice_remain:"0.0000000000
00000000e+00", token_cnt:30, ass_token_cnt:30, lq_tokens:3, used_lq_tokens:3, stopped:false
, idle_us:4945506, recv_hp_rpc_cnt:2420622, recv_np_rpc_cnt:7523808, recv_lp_rpc_cnt:0, rec
v_mysql_cnt:4561007, recv_task_cnt:337865, recv_large_req_cnt:1272, tt_large_quries:3648648
, actives:35, workers:35, nesting workers:7, lq waiting workers:5, req_queue:total_size=481
83 queue[0]=47888 queue[1]=0 queue[2]=242 queue[3]=5 queue[4]=48 queue[5]=0 , large queued:
12, multi_level_queue:total_size=0 queue[0]=0 queue[1]=0 queue[2]=0 queue[3]=0 queue[4]=0 q
ueue[5]=0 queue[6]=0 queue[7]=0 , recv_level_rpc_cnt:cnt[0]=0 cnt[1]=0 cnt[2]=0 cnt[3]=0 cn
t[4]=0 cnt[5]=165652 cnt[6]=10 cnt[7]=0 })
```

示例日志中部分字段的说明如下表所示。

字段	说明
Id	租户 ID。
unit_min_cpu	表示最小 CPU 核数，即保证能提供的 CPU 核数。
unit_max_cpu	表示最大 CPU 核数，即限制上限。
slice	无实际意义。
slice_remain	无实际意义。
token_cnt	表示调度器分配的 Token 数，一个 Token 会转换为一个工作线程。
ass_token_cnt	租户当前确认的 Token 数，可以根据 `token_cnt` 来确认，一般两者的值相等。
lq_tokens	大请求的 Token 个数，根据 `token_cnt` 的值乘以大请求的比例设置。
used_lq_tokens	当前持有大查询 Token 的 Worker 数。
stopped	表示租户的 Unit 是否正在删除。
idle_us	一轮（10秒）中工作线程空闲的总时间和，所谓空闲实际只统计了等待队列的时间。

字段	说明
recv_hp/np/lp_rpc_cnt	租户累计收到不同级别（hp(High)、np(Normal)、lp(Low)）的 RPC 请求数。
recv_mysql_cnt	租户累计收到的 MySQL 请求数。
recv_task_cnt	租户累计收到的内部任务数。
recv_large_req_cnt	租户累计预判的大请求数，只会递增，不会清零。实际是重试时是递增的。
tt_large_quries	租户累计处理的大请求数，只会递增，不会清零。实际是打点 Check 的时候递增的。
actives	活跃工作线程数，一般和 Worker 数相等，它们的差包含：租户工作线程缓存+带工作线程的大请求缓存。
workers	租户持有的工作线程数。
nesting workers	租户持有的嵌套请求专用线程数，共 7 个线程对应 7 个嵌套层级。
lq waiting workers	处于等待调度的工作线程。
req_queue	不同优先级的工作队列，数字越小优先级越高。
large queued	当前预判出的大请求个数。
multi_level_queue	存放嵌套请求的工作队列，1~7 对应 7 个嵌套层级，其中，`queue[0]` 暂时不用。

2.5.3. 资源单元的均衡

资源单元（Unit）是 OceanBase 数据库系统内用户资源在 OBServer 上的容器（或虚拟机），它是 OceanBase 数据库作为多租户分布式数据库架构的重要概念。RS 模块需要对资源单元进行管理，并通过把资源单元在多个 OBServer 间调度，对系统资源进行有效利用。

RS 对资源单元的管理包括：

- 资源单元的分配，即新建一个资源单元时，RS 需要决定这个 Unit 分配到哪个 OBServer 上。

- 资源单元的均衡，即在系统运行过程中，RS 根据 Unit 的资源规格等信息对 Unit 进行再平衡的一个调度过程。

当前，Unit 的分配和均衡策略的目标如下：

- 考虑能够支持多种资源的分配和均衡调度，主要为 CPU 资源和 Memory 资源。
- 支持 Zone 内 OBServer 间资源均衡的分配。

多种资源

在当前的 Unit 分配和均衡算法中，主要考虑两类资源（CPU 资源和 Memory资源）的分配和均衡，多种资源同时存在时，资源的分配和均衡可能存在一定问题。为简化问题描述，先考察仅有单一资源的分配和均衡算法，以下以 CPU 资源为例，讨论单一资源的分配和均衡算法。

CPU 单一资源的均衡示例

- 示例背景

  假设存在两个 OBServer：OBS0（10 个 CPU）和 OBS1（10 个 CPU）。其中，OBS0 上包含 6 个 Unit，每个 Unit 的资源规格为 1 个 CPU；OBS1 上包含 4 个 Unit，每个 Unit 的资源规格为 1 个 CPU。

- 均衡目标

  通过在 OBServer 间迁移 Unit，使得各 OBServer 的 CPU 占用率尽可能接近。

- 均衡过程

  从该示例场景可以看到，OBS0 的 CPU 占用率为 `(6 / 10) * 100%= 60%`，OBS1 的 CPU 占用率为 `(4 / 10) * 100% = 40%`。两个 OBServer 的 CPU 占用率差值为 0.2，将 OBS0 上的一个 Unit 迁移到 OBS1 上，迁移后的 OBS0 的 CPU 占用率为 `(5 / 10) *100% = 50%`，OBS1 的 CPU 占用率为 `(5 / 10) *100% = 50%`，两个 OBServer 的 CPU 相等，与迁移 Unit 前相比，两个 OBServer 的资源占用率更平均。

多种资源占用率的计算

当系统中有多种资源需要进行分配和均衡时，仅使用其中一种资源的占用率去进行分配和均衡可能不够准确，也很难达到较好的分配和均衡效果，为此，OceanBase 数据库在多种资源（CPU 资源和 Memory 资源）均衡和分配时，使用了如下的分配和均衡方法，即为参与分配和均衡的每种资源分配一个权重，作为计算 OBServer 总的资源占用率时该资源所占的百分比。每种资源的权重使用如下方法计算，即某种资源使用的越多，则该资源的权重就越高。

例如，某集群中总的 CPU 资源为 50 个CPU，Unit 共占用 20 个 CPU，则 CPU 总的占用率为 40%。该集群中总的 Memory 资源为 1000 GB，Unit 共占用 Memory 资源 100 GB，则 Memory 占用率为 10%，集群中没有其他资源参与均衡。归一化后，CPU 和 Memory 资源的权重分配为 80% 和 20%，各 OBServer 根据该权重计算各自的资源占用率，然后再通过迁移降低各 OBServer 之间的资源占用率差值。

资源单元的分配

创建一个新的 Unit 时，需要为该 Unit 选择一个 OBServer 宿主机，分配宿主机所采用的方法是：先根据上面多资源占用率的计算规则，计算出每一个 OBServer 的资源占用率，然后选取资源占用率最小的那台 OBServer，作为新建 Unit 的宿主机。

资源单元的均衡

资源单元均衡是通过在 OBServer 间迁移 Unit 的方式使得各 OBServer 的资源占用率相差尽量小，使用上述多种资源占用率的算法，可以计算出每台 OBServer 的资源占用率，并尝试不断迁移 Unit，使得迁移 Unit 完成后，各 OBServer 之间的资源占用率比迁移 Unit 前更小，即完成了资源单元的均衡。

资源单元均衡的控制

OceanBase 数据库通过以下 3 个配置项控制资源单元的均衡：

- `enable_rebalance`

  该配置项为负载均衡的总开关，用于控制资源单元的均衡和分区副本均衡的开关。当 `enable_rebalance` 的值为 `False` 时，资源单元均衡和分区副本均衡均关闭；当 `enable_rebalance` 的值为 `True` 时，资源单元均衡需参考配置项 `resource_soft_limit` 的配置。

  有关分区副本均衡的详细信息请参见 自动负载均衡。

- `resource_soft_limit`

  该配置项为资源单元均衡的开关。当 `enable_rebalance` 的值为 `True` 时，资源单元的均衡是否开启需要参考该配置项的设置。

  当 `enable_rebalance` 的值为 `True` 且 `resource_soft_limit` 的值小于 `100` 时，资源单元均衡关闭；当 `enable_rebalance` 的值为 `True` 且 `resource_soft_limit` 的值大于等于 `100` 时，资源单元均衡开启。

- `server_balance_cpu_mem_tolerance_percent`

  该配置项为触发资源单元均衡的阈值，当某些 OBServer的资源单元负载与平均负载的差值超过 `server_balance_cpu_mem_tolerance_percent` 设置的值时，开始调度均衡，直到所有 OBServer 的资源单元的负载与平均负载的差值都小于配置项 `server_balance_cpu_mem_tolerance_percent` 的值。

手动迁移资源单元

除了上述资源单元的自动均衡以外，OceanBase 数据库还支持资源单元的手动迁移，即数据库管理员可以通过 SQL 指令对资源单元进行手动迁移，具体迁移语句如下。

```
obclient> ALTER SYSTEM MIGRATE UNIT $unit_id DESTINATION '$server';
```

其中：

- $ `unit_id` ：填写待迁移的 Unit 的 ID，可通过 `oceanbase.gv$unit` 视图查询。
- $ `server` ：填写将 Unit 迁移到的目标 Server 地址，格式为 `ip` 地址:端口号。 例如，`10.10.10.1:2882` 。

处于迁移中的资源单元，您也可以手动取消该资源单元的迁移，具体语句如下。

```
obclient> ALTER SYSTEM CANCEL MIGRATE UNIT $unit_id;
```

3.数据库对象

3.1. Oracle 模式

3.1.1. 数据库对象介绍

3.1.1.1. 数据库对象概述

数据库对象是数据库的组成部分，是指在数据库中可以通过 SQL 进行操作和使用的对象。本章节主要介绍 OceanBase 数据库 Oracle 模式下所支持的数据库对象类型、存储方式和数据库对象之间的依赖。

OceanBase 数据库 Oracle 模式下的数据库对象主要包括：表（Table）、视图（View）、索引（Index）、分区（Partition）、序列（Sequence）、同义词（Synonyms）、触发器（Trigger）以及各种 PL 对象（PL 子程序和包）等。

Oracle 模式下的用户（User）在被赋予相关权限后可以连接数据库并拥有数据库对象。

Oracle 模式下的一个 Schema 是指一个用户所拥有的数据库对象的集合，用于权限管理和命名空间隔离。一般来说，一个 Schema 往往对应于一个应用。用户在创建时会拥有一个缺省的 Schema，其 Schema 名就等于用户名。

用户还可以访问和使用其他的 Schema。在访问一个 Schema 中的对象时，如果没有指明该对象属于哪一个 Schema，系统就会自动给对象加上缺省的 Schema 名称。

Schema 对象是指在某个 Schema 中的数据库对象，例如 Schema 中的表、视图、索引等；非 Schema 对象是指不属于某个 Schema 的数据库对象，例如用户、角色、表空间、目录等。

3.1.1.2. 数据库对象类型

数据库对象（即 Schema 对象）属于某一个 Schema。Schema 为数据库对象的集合，一个用户有一个缺省的 Schema，其 Schema 名称就等于用户名。用户还可以访问和使用其他的 Schema。如果访问一个 Schema 中的对象时，没有指明该对象属于哪一个 Schema ，系统就会自动给对象加上缺省的 Schema 名称。

有关 OceanBase 数据库 Oracle 模式下的数据库对象的详细信息如下表所示。

对象类型	描述
表（Table）	数据库里最基础的存储单元，表里的数据由行和列组织排列的。
视图（View）	视图是一个虚拟的表，其内容由查询定义。 同真实的表一样，视图包含一系列带有名称的列和行数据。但是，视图并不在数据库中以存储的数据值集合的形式存在。行和列数据可以自由定义视图的查询所引用的表，并且在引用视图时动态生成。

对象类型	描述
索引（Index）	索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。 例如，按照指定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。
分区（Partition）	OceanBase 数据库可以把普通的表的数据按照一定的规则划分到不同的区块内，将同一区块的数据物理上存储在一起。这种划分区块的表叫做分区表。 与 Oracle 中的 Partition 概念相同，在 OceanBase 数据库中只有水平分区，表的每一个分区包含一部分记录。根据行数据到分区的映射关系不同，分为 Hash 分区、Range 分区（按范围）和 List 分区等。 每一个分区，还可以用不同的维度再分为若干分区，称为二级分区。例如，交易记录表按照用户 ID 分为若干 Hash 分区，每个一级 Hash 分区再按照交易时间分为若干二级 Range 分区。
序列（Sequence）	序列（Sequence）是序列号生成器，可以自动生成序列号，产生一组等间隔的数值 (数值类型为数字)。 每个 Sequence 对象实例能够生成一个数据序列，无论在单机还是多机环境下，都可以保证生成的每一个数据不相同。同时，为了满足不同应用场景的需求，在保证每个数据不相同的前提下，Sequence 还给用户提供了更加丰富的选项。例如保证数据按照生成的先后顺序确保递增，尽可能避免两个数据之间相差太大等。 需要注意的是，对 Sequence 要求越严格，数据库付出的性能代价也越大。
同义词（Synonyms）	同义词是数据库对象的一个别名，用于简化 Schema 对象访问和提高对象访问的安全性。 在使用同义词时，OceanBase 数据库 Oracle 模式下会将它翻译成对应 Schema 对象的名字。与视图类似，同义词并不占用实际存储空间，只在数据字典中保存了同义词的定义。数据库中的大部分数据库对象，例如表、视图、同义词、序列等，数据库管理员都可以根据实际情况为他们定义同义词。 **注意** PL 中的对象，例如存储过程、包等暂不支持创建同义词。
触发器（Trigger）	触发器是由系统在指定事件发生时自动调用的一组存储过程。 与普通存储过程不同的是，触发器可以被启用或禁用，不能被显式调用。

对象类型	描述
PL 对象（PL 子程序和包）	PL 是一种过程化程序语言（Procedural Language）。它是对 SQL 的扩展，在普通 SQL 语句的基础上增加了编程语言的特点，把数据操作和查询语句组织在 PL 代码的过程化代码中，通过逻辑判断、循环等操作实现复杂的功能。 PL 源程序的基本单元是块，由声明和语句组成，内部允许声明变量和常量，并可以在表达式中使用。PL 程序包由逻辑上相关的 PL 类型、变量、常量、子程序、游标和异常组成。

3.1.1.3. 数据库对象存储

OceanBase 数据库将一些数据库对象的数据存储在一个或者多个 SSTable 中，一个 SSTable 包含一个或多个宏块，每个宏块中包含一个或多个微块，每个微块包含一行或者多行。有些数据库对象（例如视图）由于只有元数据信息，所以不会存储实际的数据。

不同于一些传统数据库，OceanBase 数据库的表空间是系统自动管理的，用户不需要显式创建和管理表空间。一些功能支持表空间的语法（例如使用透明加密功能时）允许用户设置表空间相关的属性。

3.1.1.4. 数据库对象之间的依赖

数据库对象之间的依赖就是描述数据库对象之间的依赖和引用关系。本文主要介绍 OceanBase 数据库 Oracle 模式下如何创建和查看数据库对象之间的依赖关系。

创建和查看数据库对象之间的依赖关系

某些类型的 Schema 对象可以在其定义中引用其他的对象。例如，一个视图的定义可能是一个引用了表或者其他视图的查询；一个存储过程的过程体中包含的 SQL 可以引用数据库的其他对象。如果对象 A 的定义引用了对象 B，则 A 是（B 的）依赖对象，B 是（A 的）引用对象。

数据库对象之间的依赖关系在某些场景下非常有用。例如，在数据库迁移的场景下，需要在另一个数据库中重建 Schema 对象。如果需要重建的 Schema 对象是一个视图，那就必须要先重建该视图依赖的其他对象，否则重建该视图时会报错“依赖的对象不存在”。如果能获取到数据库对象之间的依赖关系，那就可以很轻松的完成重建对象的工作。

创建数据库对象之间的依赖关系成功之后可以通过 USER_DEPENDENCIES、ALL_DEPENDENCIES 和 DBA_DEPENDENCIES 视图查看数据库对象之间的依赖关系。

示例 1： 创建一个视图 `view2`，该视图引用了表 `tbl1` 和另一个视图 `view1`。

```
obclient> CREATE TABLE tbl1(col1 INT, col2 INT);
Query OK, 0 rows affected

obclient> CREATE TABLE tbl2 (col1 INT, col2 INT);
Query OK, 0 rows affected

obclient> CREATE VIEW view1 AS SELECT * FROM tbl2;
Query OK, 0 rows affected

obclient> CREATE VIEW view2 AS SELECT t.col1 AS col1, t.col2 AS col2 FROM tbl1 t, view1 v W
HERE
      t.col1 = v.col2;
Query OK, 0 rows affected
```

示例 2：通过 USER_DEPENDENCIES 视图查看数据库对象之间的依赖关系。

```
obclient> SELECT * FROM USER_DEPENDENCIES;
+------+------+-----------------+----------------+----------------+--------------------
-+------------------+-----------------+
| NAME | TYPE | REFERENCED_OWNER | REFERENCED_NAME | REFERENCED_TYPE | REFERENCED_LINK_NAME
| SCHEMAID         | DEPENDENCY_TYPE |
+------+------+-----------------+----------------+----------------+--------------------
-+------------------+-----------------+
| VIEW1   | VIEW | SYS              | TBL2              | TABLE           | NULL
| 1100611139403782 | HARD            |
| VIEW2   | VIEW | SYS              | TBL1              | TABLE           | NULL
| 1100611139403782 | HARD            |
| VIEW2   | VIEW | SYS              | VIEW1              | VIEW            | NULL
| 1100611139403782 | HARD            |
+------+------+-----------------+----------------+----------------+--------------------
-+------------------+-----------------+
3 rows in set
```

上述示例中，视图的查询结果中描述了数据库对象之间的依赖关系。例如，视图 `view1` 引用了表 `tbl2`，视图 `view2` 引用了表 `tbl1` 和另一个视图 `view1`。

注意事项

数据库对象之间的依赖关系在数据库中维护的并不是一个实时并且绝对准确的关系，引用对象被删除、重建时对应的依赖关系的更新依赖于下一次对依赖对象的查询或 DML 触发。

如下示例中，创建一个视图 `view1`，该视图引用了表 `tbl1`。当删除表 `tbl1` 之后查询 USER_DEPENDENCIES 视图依然可以查到它们之间的依赖关系。直到对视图 `view1` 触发了一次 DML 操作，再次查询 USER_DEPENDENCIES 视图可以发现它们之间的依赖关系已经被删除。

```
obclient>CREATE TABLE tbl1 (col1 INT, col2 INT);
Query OK, 0 rows affected

obclient>CREATE VIEW view1 AS SELECT * FROM tbl1;
Query OK, 0 rows affected

obclient>SELECT * FROM USER_DEPENDENCIES;
+------+------+-----------------+----------------+----------------+--------------------
-+------------------+-----------------+
| NAME | TYPE | REFERENCED_OWNER | REFERENCED_NAME | REFERENCED_TYPE | REFERENCED_LINK_NAME
| SCHEMAID         | DEPENDENCY_TYPE |
+------+------+-----------------+----------------+----------------+--------------------
-+------------------+-----------------+
| VIEW1    | VIEW | SYS              | TBL1              | TABLE           | NULL
| 1100611139403782 | HARD            |
+------+------+-----------------+----------------+----------------+--------------------
-+------------------+-----------------+
1 row in set

obclient>DROP TABLE tbl1;
Query OK, 0 rows affected

obclient>SELECT * FROM USER_DEPENDENCIES;
+------+------+-----------------+----------------------------+----------------+--------
-------------+------------------+-----------------+
| NAME | TYPE | REFERENCED_OWNER | REFERENCED_NAME              | REFERENCED_TYPE | REFEREN
CED_LINK_NAME | SCHEMAID         | DEPENDENCY_TYPE |
+------+------+-----------------+----------------------------+----------------+--------
-------------+------------------+-----------------+
| VIEW1    | VIEW | __recyclebin     | RECYCLE_$_1_1635668004963688 | TABLE           | NULL
| 1100611139403782 | HARD            |
+------+------+-----------------+----------------------------+----------------+--------
-------------+------------------+-----------------+
1 row in set

obclient>SELECT * FROM view1;
ORA-04063: view 'SYS.view1' has errors
obclient>SELECT * FROM USER_DEPENDENCIES;
Empty set
```

3.1.2. 表

3.1.2.1. 表概述

数据库表是一系列二维数组的集合，用来代表和储存数据对象之间的关系。本章节主要介绍表所涉及的数据库对象以及表的相关使用特性，包括表存储、表压缩、表组和临时表等。

在数据库中，表是数据组织的基础单元，它由纵向的列和横向的行组成。例如，一个有关作者信息的表格名称为 “作者”，表中每个列包含的是所有作者的某个特定类型的信息，例如姓氏，而每行则包含了某个指定作者的所有信息，例如姓、名、住址等等。对于指定的数据库表，列的数目一般在创建表时确定，各列之间可以由列名来识别，而行的数目可以随时动态变化。

表分为普通表和临时表。OceanBase 数据库 Oracle 模式下的临时表生命周期同普通表一样，临时表被创建后，对任何会话（ `session` / `sess` ）都可见，只有显式执行 `DROP` 命令才会被清理。临时表占用了普通表的命名空间，对象名不能和普通表名字相同，且被执行 `DROP` 后不会进入回收站。

临时表的数据清除策略分为会话级别和事务级别，会话级别的临时表数据会将每个会话各自保存一份，会话之间互不干扰，在会话断开时数据会被清理；而事务级别的临时表在事务提交时数据会被清理。

列

在数据库中，列（Column）用于记录一张表上某个属性的字段的值，用户给每个属性起的名称即为列名。除了列名以外，列上还有数据类型以及数据类型的最大长度（精度）等信息。

除了普通列以外，OceanBase 数据库 Oracle 模式下还包括虚拟列（Virtual Column/Generated Column）和隐藏列（Invisible Column）。

虚拟列不像普通列一样具有真实的物理存储空间，而是在查询时通过用户在虚拟列上定义的一个表达式或函数来计算得到结果的。

隐藏列只有当查询时被显式指定出列名时才会对用户可见，如果使用 `SELECT *` 进行查询表数据，则隐藏列对用户不可见。隐藏列多用于数据迁移或者对已经上线的应用进行功能扩展。

行

在数据库中，行（Row）表示表中单条记录中所有列的数据集合。简单来说，数据库可以认为是由列和行组成的。表中的每一行代表一组相关数据，并且表中的每一行具有相同的结构。

例如，一张与公司信息相关的数据表，每一行代表一个公司，对应的列可能代表诸如公司名称、公司街道地址、增值税号等内容。

3.1.2.2. 数据类型

3.1.2.2.1. 数据类型概述

在 OceanBase 数据库中，表中的每一列都有一个数据类型，每个数据类型都对应独特的存储格式、约束以及取值范围。本章节主要介绍最常用的 OceanBase 数据库内置数据类型。

OceanBase 数据库提供一些内置的数据类型，其中最常用的为以下四种类型：

- 字符数据类型
- 数值数据类型
- 日期时间数据类型
- Rowid 数据类型

其他重要的数据类型还包括 `RAW` 、大对象数据类型以及集合数据类型。PL 中还可以为常量和变量指定一些其他的类型，包括 `BOOLEAN` 、引用类型、复合类型（包括 `COLLECTION` 和 `RECORD` 类型）以及用户自定义类型。

数据的类型将一组固定的属性与数据相关联，这些属性使得 OceanBase 数据库有区别地对待不同类型的数据。例如，您可以对 `NUMBER` 类型的数据进行乘法操作，但不能对 `RAW` 类型的数据进行此类操作。

用户创建一张表时，必须为其中的每一列都指定数据类型，之后插入列中的每条数据都需要与类型相符。

3.1.2.2.2. 字符数据类型

字符类型将数字、字母等字符以字符串的形式存储。本节主要介绍最常用的字符数据类型。

字符串中的字节序列使用了某种字符编码方式，该字符编码方式通常被称为字符集。数据库使用字符集是在租户创建时指定的，例如 UTF8、GBK 字符集等。

字符类型一般包括长度语义、字节语义和字符语义。字符类型的长度语义定义了如何度量字符串的长度，按字节长度或者按字符个数。字节语义是将字符串当做字节序列处理，这是字符类型默认的选择。字符语义是将字符串当做字符序列，字符序列中的字符对应数据库字符集中的码位。

VARCHAR2 和 CHAR 类型

在数据库中，最常用的字符数据类型是 `VARCHAR2`，这也是最高效的存储方式。`VARCHAR2` 类型存储不定长的字符串常量，例如 'LILA'、'St. George Island' 和 '101' 都是字符串常量，它们的长度不相同。与数字常量（例如 5001）不同，字符串常量用单引号包裹起来，这样数据库可以区分出两者。

当创建一张表包含 `VARCHAR2` 列，用户可以定义 `VARCHAR2` 类型能存储字符串的最大长度，例如 `VARCHAR2(25)` 表示最多存储 25 个字节。

对每一行，数据库能够存储列中的不同长度的值，如果数据长度超过了列定义的最大长度就会报错。向字符集为 UTF8 且最大长度为 25 的列中插入 10 个字符，列中存储的字符数为 10，而不是 25，因此使用 `VARCHAR2` 可以减少存储空间消耗。相比之下，`CHAR` 存储定长的字符串，默认的定义长度为 1 字节，当实际的长度小于类型定义的长度时，数据库会使用空格将数据填充到定义的长度。

在 OceanBase 数据库 Oracle 模式下比较 `VARCHAR2` 类型时，按照非填充空格的模式进行比较；而比较 `CHAR` 类型时，按照填充空格的模式比较。

NCHAR 和 NVARCHAR2 类型

`NCHAR` 和 `NVARCHAR2` 类型用于存储 Unicode 字符数据。

Unicode 是通用的字符集，用一个字符集可以存储任何语言的字符。`NCHAR` 用于存储定长的字符串，`NVARCHAR2` 用于存储变长字符串，它们使用的字符集是国家字符集（National Character Set），国家字符集可以在租户创建时进行设置。国家字符集可以用于存储用户当地特殊字符，为了统一标准，国家字符集只能使用基于 Unicode 编码的字符集，例如 AL16UTF16 字符集或 UTF8 字符集。

当用户创建一张表包含 `NCHAR` 或 `NVARCHAR2` 的列时，它们的总是遵循字符语义。

3.1.2.2.3. 数值数据类型

数值数据类型使用数值字面量指定固定数和浮点数的值，同时支持 0、无限和非数字（"Not a Number" 或者 NaN）。本节主要介绍数字类型（NUMBER）和浮点数字。

数字类型（NUMBER）

`NUMBER` 数据类型存储定点数和浮点数。该类型可以存储几乎任何数量级的数字。

- 精度（Precision）

精度用来表示数字的总位数。如果未指定精度，按提供原始值存储，不进行任何四舍五入。

- 小数位数（Scale)

  小数位数指定从小数点到最低有效位的位数。正小数位数计算小数点右侧的数字，直到最低有效数字。负小数位数计算小数点左侧的数字。如果您指定一个没有小数位数的精度，例如 `NUMBER(6)` ，则小数位数为 0。

数字字面量最大可以储存精度为 38 位的数字。如果字面量要求的精度比 `NUMBER` 、 `BINARY_FLOAT` 或 `BINARY_DOUBLE` 所提供的精度更高，则 OceanBase 数据库将截断该值。如果字面量的范围超出 `NUMBER` 、 `BINARY_FLOAT` 或 `BINARY_DOUBLE` 支持的范围，则 OceanBase 数据库会抛出错误。

例如， `NUMBER(8,2)` 表示精度为 8，小数位数为 2。因此，数据库将 100,000 存储为 100000.00。

以下是一些有效的 `NUMBER` 类型的数字：

```
12
+6.87
0.5
25e-03
-9
```

以下是一些有效的 FLOAT-Point 类型的浮点数：

```
25f
+6.34F
0.5d
-1D
```

浮点数字

OceanBase 数据库专门为浮点数提供两种数值数据类型： `BINARY_FLOAT` 和 `BINARY_DOUBLE` 。

这些类型支持 `NUMBER` 数据类型提供的所有基本功能。但是， `NUMBER` 使用十进制精度，而 `BINARY_FLOAT` 和 `BINARY_DOUBLE` 使用二进制精度，这可以实现更快的算术计算并且通常会降低存储要求。

`BINARY_FLOAT` 和 `BINARY_DOUBLE` 是近似数值数据类型。它们存储十进制值的近似表示，而不是精确表示。例如，值 0.1 不能由 `BINARY_DOUBLE` 或 `BINARY_FLOAT` 精确表示。它们经常用于科学计算。它们的行为类似于 Java 中的数据类型 `FLOAT` 和 `DOUBLE` 。

以下是一些有效的浮点数字：

```
1.2
-1.1
0.
```

数值数据类型除了提供数字浮点类型外，还提供了正负无穷和非数字（not a number / NaN) 等特殊表示。详细信息如下：

- binary_double_infinity
- binary_double_nan
- binary_float_infinity
- binary_float_nan

3.1.2.2.4. 日期时间数据类型

OceanBase 数据库支持日期时间数据类型，包括 DATE 数据类型和 TIMESTAMP 数据类型。

DATE 数据类型

`DATE` 数据类型存储了日期和时间。尽管日期时间可以使用字符或数字类型表示，但是使用 `DATE` 类型时拥有独特的属性。

OceanBase 数据库内部使用 64 位 INT 存储 `DATE` 和 `TIMESTAMP` 类型的数据，记录该时间点距离 1970 年 01 月 01 日 00 时 00 分 00 秒的微秒数。

> 注意
>
> `DATE` 和 `TIMESTAMP` 类型完全支持算术运算，用户可以像对待 `NUMBER` 类型一样，对其进行加法或减法操作。

数据库以特定的格式展示日期时间类型的数据。对于 `DATE` 类型，标准的格式是 `DD-MON-RR`。例如，2011 年 1 月 1 日展示为 `01-JAN-11`。用户可以修改集群级别和会话级别的默认格式。

OceanBase 数据库默认以 24 时的格式 `HH24::MI:SS` 展示时间，如果用户没有指定时间部分的值，缺省值为 0 时 0 分 0 秒。对于只包含时间部分的数据，日期部分的缺省值为当月的第一天。

TIMESTAMP 数据类型

`TIMESTAMP` 数据类型是 `DATE` 类型的一个扩展，额外地保存了时间的小数部分，因此该类型适合存储精确的时间值。例如，用于必须记录所有事件的顺序的应用中。

`TIMESTAMP WITH TIME ZONE` 和 `TIMESTAMP WITH LOCAL TIME ZONE` 两个类型是包含时区的，该类型的数据在展示时会调整为会话的时区。此数据类型可用于记录和分析跨地理区域的日期信息。

3.1.2.2.5. Rowid 数据类型

存储在数据库中的每一行数据都有一个地址，OceanBase 数据库使用 Rowid 数据类型来存储数据库中每一行的地址。

Rowid 包括无主键表 ROWID 和有主键表 ROWID。UROWID（Universal Rowid）支持存储以上两种类型的 ROWID。

在 OceanBase 数据库中，每个表都有一个名为 ROWID 的伪列。

伪列的行为类似一个表的普通列，但实际上并没有存储在表中。可以通过伪列执行 `SELECT` 操作，但不能对它们的值执行 `INSERT` 、 `UPDATE` 、 `DELETE` 操作。伪列也类似于没有参数的 SQL 函数。不带参数的函数通常为结果集中的每一行返回相同的值，而伪列通常为每一行返回不同的值。

ROWID 伪列的值是每行数据的 Key（对于有主键表是 Primary Key，对于无主键表是系统自动生成的序列），通过 Base64 编码转换而成。可以通过使用保留字 ROWID 作为列名的 SQL 来访问每一行的 Rowid。

3.1.2.2.6. 大对象数据类型

OceanBase 数据库目前支持的大对象数据类型包括 CLOB 和 BLOB。

一个 LOB 对象由数据和定位数据的相关信息（Locator）组成，相关信息包括表 ID 和行 ID 等，可以确定该对象对应哪张表的哪一行数据。目前大对象的存储和 `VARCHAR` 类型一样，只存储了数据部分，查询时由上层接口生成 Locator 部分。

`LOB` 对象可以赋值给任意相同类型的对象。通常情况下，用户可以在应用中使用 `LOB` 对象而不用考虑 Locator 的语义。在部分场景中，Locator 是必需的部分，而在一些其他场景中，可以包含冗余的 Locator 信息。

读取和修改 `LOB` 对象时可以使用 Locator，也可以不使用 Locator。

不使用 Locator 的方式

在许多操作中， `LOB` 对象近似于 `VARCHAR2` 和 `RAW` 类型，这些操作只使用了数据，而忽略了 Locator 部分。这类操作包括：

- SQL 和 PL 的内置函数
- 允许用户查询 `LOB` 列中的数据，或向 `LOB` 列中插入数据的如下接口：
  - 通过绑定 `LOB` 列的变量向 `CLOB` 列中插入字符数据，或向 `BLOB` 列中插入 `RAW` 数据。例如，在 PL 中可以将一个 `VARCHAR2` 缓冲区插入 `CLOB` 列中。
  - 在应用中定义输出为从 `CLOB` 中查询的字符数据，或是从 `BLOB` 中查询的 `RAW` 数据。例如，在 PL 中可以将查询到的 `CLOB` 结果存入 `VARCHAR2` 缓冲区中。

使用 Locator 的方式

使用 OceanBase 数据库提供的 LOB API，并且传入 LOB 对象作为参数。这些 API 大多包含在 DBMS_LOB 包中，包括 DBMS_LOB.READ、DBMS_LOB.WRITE、DBMS_LOB.UPDATE 等等。

3.1.2.2.7. 格式模型

3.1.2.2.7.1. 格式模型概述

在 OceanBase 数据库 Oracle 模式下，可以使用格式模型指定存储在数据库中的日期时间数据或数值数据的格式。

格式模型不会更改数据库中值的内部表示形式。当数据库将字符串转换为日期或数字时，格式模型将决定如何转换并存储该字符串。

在 SQL 语句中，您可以使用格式模型作为 `TO_CHAR` 、 `TO_NUMBER` 和 `TO_DATE` 等函数的参数来指定如下格式：

- 数据库返回值的格式
- 存储在数据库中值的格式

OceanBase 数据库支持以下两种数据格式模型：

- 数字格式模型
- 日期时间格式模型

> **说明**
>
> OceanBase 数据库暂不支持格式化修饰符 `FX` 和 `FM` 。

3.1.2.2.7.2. 数字格式模型

数字格式模型（Number Format Models）指定了存储在数据库中的定点数和浮点数的格式。当您需要将 SQL 语句中出现 NUMBER、BINARY_FLOAT 或 BINARY_ DOUBLE 值转换为 VARCHAR2 数据类型时，可以使用函数中的数字格式模型。

数字格式模型的元素

数字格式模型由一个或多个数字格式元素组成。负数返回值自动包含前导负号，正值会自动包含前导空格，除非格式模型包含了 MI、S 或 PR 等表示数字正负的格式元素。

如下表格列出了数字格式包含的元素以及具体的用法。

数字格式元素	示例	描述
,（逗号）	9,999	在指定位置返回逗号。可以在数字格式模型中指定多个逗号。 限制如下： • 逗号元素不能出现在数字格式模型开头。 • 在数字格式模型中，逗号不能出现在十进制字符或句号的右侧。
.(小数点)	99.99	返回小数点，即在指定的小数点位置。 限制如下： 在数字格式模型中只能指定一个小数点。
$	$9999	返回带前导美元符号的值。

数字格式元素	示例	描述
0	0999 9990	返回前导的零。 返回尾随的零。
9	9999	返回指定位数的值。如果为正，则返回前导空格；如果为负，则返回前导减号。前导零为空，但零值除外，该值为定点数的小数部分返回零。
B	B9999	当结果为零时，返回空格。
C	C999	在指定位置返回 ISO 货币符号(当前 NLS_ISO_CURRENCY 参数所代表的值)。
D	99D99	跟“.”的作用相同，只能出现一次。所不同的是，该元素会使用参数 `NLS_NUMERIC_CHARACTER` 的默认值。
EEEE	9.9EEEE	返回使用科学计数法表示的值。
G	9G999	跟“.”的作用相同，所不同的是，该元素会使用参数 `NLS_NUMERIC_CHARACTER` 的默认值。
L	L999	在指定位置返回本地货币符号，会使用参数 `NLS_CURRENCY` 当前所代表的值。
Mi	9999MI	返回带有尾随减号（-）的负值。 返回带尾随空格的正值。 限制如下： MI 格式元素只能出现在数字格式模型的最后一个位置。
PR	9999PR	返回<尖括号>中的负值。 返回带前导和尾随空格的正值。 限制如下： PR 格式元素只能出现在数字格式模型的最后一个位置。

数字格式元素	示例	描述
RN(rn)	RN rn	以大写罗马数字形式返回值。 以小写罗马数字形式返回值。 限制如下： 取值可以是介于 1 和 3999 之间的整数。
S	S9999 9999S	• 返回带前导减号（-）的负值。 • 返回带前导加号（+）的正值。 • 返回带有尾随减号（-）的负值。 • 返回带尾随加号（+）的正值。 限制如下： S 格式元素只能出现在数字格式模型的第一个或最后一个位置。
TM	TME TM9	默认值为 `TM9`，除非输出超过64 个字符，否则以固定符号返回数字。如果输出超过 64 个字符，将自动以科学记数法返回该数字。 当数字长度超过 64 位时候，`TM9` 的输出等同于 `TME` 的输出。
U	U9999	在指定位置返回由 `NLS_dual_currency` 参数代表的当前值确定的欧元（或其他）双货币符号。
V(v)	999V99	返回一个乘以 10n 的值（如有必要，将其四舍五入），其中 n 是 V 后面的数字 9 的个数。
X(x)	XXXX xxxx	返回指定位数的十六进制值。如果指定的数字不是整数，则将其舍入为整数。 限制如下： • 此元素只接受正值或 0。负值返回一个错误。 • 只能在该元素前面加 0（返回前导零）或 FM。任何其他元素都会返回错误。如果使用 X 既不指定 0 也不指定 FM，则返回值始终有一个前导空格。
FM(fm)	FM999	去除前面的空格。

数字格式模型可以在以下的函数中使用：

- 在 TO_CHAR 函数中，将 `NUMBER` 、 `BINARY_FLOAT` 或 `BINARY_DOUBLE` 数据类型的值转换为 `VARCHAR2` 数据类型。
- 在 TO_NUMBER 函数中，将 `CHAR` 或 `VARCHAR2` 数据类型的值转换为 `NUMBER` 数据类型。
- 在 TO_BINARY_FLOAT 和 TO_BINARY_DOUBLE 函数中，将 `CHAR` 和 `VARCHAR2` 表达式转换为 `BINARY_FLOAT` 或 `BINARY_DOUBLE` 类型。

示例

- ‘S’格式元素只能出现在数字格式模型的第一个或最后一个位置。当出现在开始位置时，为正数返回带前导加号（+）的值，为负数返回带前导减号（-）的值。当出现在结束位置时，为正数返回带尾随加号（+）的值，为负数返回带尾随减号（-）的值。

```
obclient> SELECT TO_CHAR(-1234567890, '9999999999S') FROM DUAL;
+-----------------------------------+
| TO_CHAR(-1234567890,'9999999999S') |
+-----------------------------------+
| 1234567890-                       |
+-----------------------------------+
1 row in set

obclient> SELECT TO_CHAR(1234567890, '9999999999S') FROM DUAL;
+-----------------------------------+
| TO_CHAR(1234567890,'9999999999S') |
+-----------------------------------+
| 1234567890+                       |
+-----------------------------------+
1 row in set
```

- ‘9’格式模型返回指定位数的值，如果为正，则返回前导空格；如果为负，则返回前导减号。前导零值为空，但零值除外，该值为定点数的小数部分返回零。如下例所示，小数部分补了两个‘0’。

```
obclient> SELECT TO_CHAR(0, '99.99') FROM DUAL;
+--------------------+
| TO_CHAR(0,'99.99') |
+--------------------+
|    .00             |
+--------------------+
1 row in set

obclient> SELECT TO_CHAR(0.1, '99.99') FROM DUAL;
+----------------------+
| TO_CHAR(0.1,'99.99') |
+----------------------+
|    .10               |
+----------------------+
1 row in set

obclient> SELECT TO_CHAR(-0.2, '99.99') FROM DUAL;
+-----------------------+
| TO_CHAR(-0.2,'99.99') |
+-----------------------+
|   -.20                |
+-----------------------+
1 row in set (0.00 sec)
```

- 当结果为零时，‘B’格式模型返回空格。

```
obclient> SELECT TO_CHAR(0, 'B9999') FROM DUAL;
+--------------------+
| TO_CHAR(0,'B9999') |
+--------------------+
|                    |
+--------------------+
1 row in set
```

- ‘FM’格式模型表示去除前导空格。

```
obclient>SELECT TO_CHAR(123.456, 'FM999.009') FROM DUAL;
+------------------------------+
| TO_CHAR(123.456,'FM999.009') |
+------------------------------+
| 123.456                      |
+------------------------------+
1 row in set
```

- ‘EEEE’格式模型表示使用科学计数法表示值。

```
obclient> SELECT TO_CHAR(123.456, '9.9EEEE') FROM DUAL;
+----------------------------+
| TO_CHAR(123.456,'9.9EEEE') |
+----------------------------+
|   1.2E+02                  |
+----------------------------+
1 row in set

obclient> SELECT TO_CHAR(123.456, 'FM9.9EEEE') FROM DUAL;
+-----------------------------+
| TO_CHAR(123.456,'FM9.9EEEE') |
+-----------------------------+
| 1.2E+02                     |
+-----------------------------+
1 row in set
```

- ‘L ’格式模型表示在指定位置返回本地货币符号，会使用参数 `NLS_CURRENCY` 当前所代表的值。

```
obclient> SELECT TO_CHAR(123.456, 'FML999.99') FROM DUAL;
+-----------------------------+
| TO_CHAR(123.456,'FML999.99') |
+-----------------------------+
| $123.46                     |
+-----------------------------+
1 row in set
```

3.1.2.2.7.3. 日期时间格式模型

日期时间格式模型（Datetime Format Models）指定了存储在数据库中日期时间数据的格式。

日期时间格式模型的总长度不能超过 22 个字符。当您需要将非默认格式的字符值转换为日期时间格式的值时，可以使用函数中的日期时间格式模型。

日期时间格式模型可以在以下函数中使用：

- 将非默认格式的字符值转换为日期时间值时，需要使用 `TO_DATE` 、 `TO_TIMESTAMP` 和 `TO_TIMESTAMP_TZ` 函数的参数指定日期时间的格式。
- 将日期时间值转换为非默认格式的字符值时，需要使用 `TO_CHAR` 函数的参数指定输出的字符串的格式。

当用户没有指定格式时，使用默认格式进行转换。您可以通过系统变量 `NLS_DATE_FORMAT` 、 `NLS_TIMESTAMP_FORMAT` 或 `NLS_TIMESTAMP_TZ_FORMAT` 显式指定默认日期时间格式。用户可以在 `V$NLS_PARAMETERS` 系统视图中查询这些变量的值，也可以通过 `ALTER SESSION` 或者 `ALTER SYSTEM` 命令修改变量的值。

日期时间格式模型的元素

日期时间格式模型由一个或多个日期时间格式模型元素组成。OceanBase 数据库支持的格式模型元素请查阅**日期时间格式模型元素表**。

- 如果格式用于解析输入的字符串，相同的格式化元素不能出现两次，表示类似信息的格式化元素不能组合。例如，您不能在一个日期时间格式中同时使用 `SYYYY` 和 `BC` 元素。这条规则是为了避免信息冗余导致冲突。
- 部分元素不能在 `TO_CHAR` 、`TO_DATE` 、`TO_TIMESTAMP` 和 `TO_TIMESTAMP_TZ` 函数中使用。
- 日期时间格式模型元素 `FF` 、`TZD` 、`TZH` 、`TZM` 和 `TZR` 可以出现在时间戳和间隔类型的格式中，但不能出现在 `DATE` 类型的格式中。
- 部分元素会生成固定长度的字符串，长度不足时会在末尾补充空格。

> **注意**
>
> 建议您使用 4 位数的年份元素（ `YYYY` ），较短的年份元素会影响查询优化，因为年份只能在运行时确定。

日期时间格式化元素表

元素	是否可以用于 TO_* 函数	描述
- / , 。 ; : “文字”	是	符号字符串常量。
AD A.D.	是	表示公元纪年法。可以带有或不带有点号。
AM A.M.	是	表示上午。可以带有或不带有点号。
BC B.C.	是	表示公元前的年份。可以带有或不带有点号。
CC SCC	否	表示世纪。

元素	是否可以用于 TO_* 函数	描述
D	是	一周的第几（1~7）天。
DAY	是	天的名称。结果的长度是固定的，与名称的最长长度的相同。
DD	是	一个月的第几（1~31）天。
DDD	是	一年的第几（1~366）天。
DY	是	天的名称的缩写。
FF [1..9]	是	小数秒。使用数字 1~9 来指定返回值的小数秒部分的位数，即精度。 默认为日期时间数据类型指定的精度。在时间戳和间隔格式中有效，但在 `DATE` 格式中无效。
HH	是	小时(1~12)。
HH12	否	小时（1~12）。12 小时制
HH24	是	小时（0~23）。24 小时制
IW	否	一年的第几周(1~52 或 1~53)，基于 ISO 标准。
IYY IY I	否	ISO 年份的后 3、2 或 1 位数字。
IYYY	否	包含 4 位数字的年份，基于 ISO 标准。
J	是	儒略日

元素	是否可以用于 TO_* 函数	描述
MI	是	分钟（0~59）。
MM	是	月（01~12）。一月份表示为 01，以此类推 。
MON	是	月份的缩写。
MONTH	是	月份名称。结果的长度是固定的，与名称的最长长度的相同。
PM P.M.	否	表示下午。可以带有或不带有点号。
Q	否	季度（1、2、3、4）。；1月~3月是第 1 季度，以此类推。
RR	是	年份的后两位， 所在世纪取决于输入的两位以及当前年份处于上半世纪还是下半世纪。 如果现在处于上半世纪，那么输入 0~49 年时，结果处于当前世纪，输入 50~99 时结果处于上一世纪。 例如，当前为 2060 年，那么输入 0~49 年时结果为下一世纪，输入 55~99 时结果处于当前世纪。
RRRR	是	年份。接受 4 位或 2 位输入。2位时与 RR 相同。
SS	是	秒（0~59）。
SSSSS	是	当天的秒数（0~86400）。

元素	是否可以用于 TO_* 函数	描述
TZD	是	包含夏令时信息的时区缩写。 例如，PST 表示 US/Pacific 标准时间）；PDT 表示 US/Pacific 夏令时时间）。 在时间戳和间隔格式中有效，但在 `DATE` 格式中无效。
TZH	是	时区的小时部分。 在时间戳和间隔格式中有效，但在 `DATE` 格式中无效。
TZM	是	时区的分钟部分。 在时间戳和间隔格式中有效，但在 `DATE` 格式中无效。
TZR	是	时区的区域信息。例如US/Pacific。 在时间戳和间隔格式中有效，但在 `DATE` 格式中无效。
WW	否	一年的第几（1-53）周。
W	否	一个月的第几（1~5）周。
X	是	本地分隔小数部分的字符。 例如“HH:MI:SSXFF”。
Y,YYY	是	带逗号的年份。
YEAR SYEAR	否	详细说明的年份。
YYYY SYYYY	是	4 位数字的年份。前缀 S 代表用一个负号表示公元前的日期。

元素	是否可以用于 TO_* 函数	描述
YYY YY Y	是	年份的后 3、2 或 1 位数字。

> **说明**
>
> 日期时间函数指的是 `TO_CHAR` 、 `TO_DATE` 、 `TO_TIMESTAMP` 和 `TO_TIMESTAMP_TZ` 。

> **注意**
>
> - 表格中展示的转化要求输入的字符串日期能够与格式元素相匹配，否则会报错。
> - OceanBase 数据库中默认的日期格式为 DD-MON-RR，如果要显示为上面的格式，可以通过执行 `ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';` 修改当前会话的日期时间格式。

当您的格式字符串漏掉了一些元素时，会得到系统的报错信息：

```
obclient> SELECT TO_DATE( '31 Aug 2020', 'DD MON YYY' ) FROM DUAL;
ORA-01830: date format picture ends before converting entire input string
```

日期格式模型元素中的大写字母

格式中元素的大小写会影响生成的字符串的大小写。例如，日期格式元素 `DAY` 生成的 `MONDAY` 也大写， `Day` 生成 `Monday` ， `day` 生成小写的 `monday` 。示例如下。

```
obclient> SELECT TO_CHAR(sysdate,'mon')  AS nowMonth FROM DUAL;
+-----------+
| NOWMONTH |
+-----------+
| mar      |
+-----------+
1 row in set

obclient> SELECT TO_CHAR(sysdate,'MON')  AS nowMonth FROM DUAL;
+-----------+
| NOWMONTH |
+-----------+
| MAR      |
+-----------+
1 row in set
```

日期时间格式模型中的标点符号和字符串常量

用户可以在格式中包含以下字符，这些字符出现在返回值中的位置与格式模型中字符的位置相同：

- 标点符号，例如连字符、斜杠、逗号、句号和冒号。
- 字符串常量，使用双引号包含起来。

OceanBase 数据库可以灵活的将字符串转换为日期。当您使用 `TO_DATE` 函数时，如果输入字符串中的每个数字元素都包含格式模型允许的最大位数，则格式字符串将与输入的字符串匹配。

示例

- **示例 1**：格式元素 **MM/YY**，其中 **02** 对应 **MM**，**07** 对应 **YY**。

```
obclient> SELECT TO_CHAR(TO_DATE('0207','MM/YY'),'MM/YY') FROM DUAL;
+-----------------------------------------+
| TO_CHAR(TO_DATE('0207','MM/YY'),'MM/YY') |
+-----------------------------------------+
| 02/07                                   |
+-----------------------------------------+
1 row in set
```

- **示例 2**：OceanBase 数据库允许非字母数字字符与格式化中的标点字符匹配，**#** 对应 **/**。

```
obclient> SELECT TO_CHAR (TO_DATE('02#07','MM/YY'), 'MM/YY') FROM DUAL;
+-----------------------------------------+
| TO_CHAR(TO_DATE('02#07','MM/YY'),'MM/YY') |
+-----------------------------------------+
| 02/07                                   |
+-----------------------------------------+
1 row in set
```

3.1.2.3. 完整性约束

完整性约束（Integrity Constraint）用来确保数据遵从业务规则的要求，防止数据表中出现不满足规则的非法数据。本文主要介绍如何使用完整性约束以防止数据库的表中被插入非法数据。

OceanBase 数据库 Oracle 模式下的租户可以使用完整性约束以防止用户向数据库的表中插入非法数据。完整性约束的作用是确保数据库内存储的信息遵从一定的业务规则，如果 DML 语句的执行结果违反了完整性约束，将回滚语句并返回错误消息。对视图和表的同义词（Synonym）执行的操作需要遵从基表（Base Table）上的完整性约束。

在 OceanBase 数据库 Oracle 模式下，用户可以通过修改约束的 `ENABLE` 和 `DISABLE` 选项控制该约束是否生效。

例如，用户在 `employees` 表的 `salary` 列上定义了完整性约束。此完整性约束规定 `salary` 列的数字值大于 10000 的数据行不能插入到 `salary` 表。如果某个 `INSERT` 或 `UPDATE` 语句违反了此完整性约束，将回滚语句并返回错误消息。

有关完整性约束的具体内容，请参见 数据完整性。

3.1.2.4. 表存储

OceanBase 数据库中使用宏块来存储数据，每张表可能包含多个宏块，每个宏块占用 2M 空间。宏块内包含一个或者多个微块，每个微块内包含一行或者多行数据。

表的组织形式

OceanBase 数据库中的每张表都有存储键，对于无主键表，存储键是分区键和自动生成的隐藏自增列，而对于有主键表，存储键是用户指定的主键。OceanBase 数据库中使用聚集索引表模型来组织，也就是存储时按照表的存储键顺序来存储数据。当往表中插入一行数据时，会按照其存储键的顺序插入到表中。

每行存储的列顺序是一样的。OceanBase 数据库通常是按照建表时的列顺序来存储，新加的列都放在最后一列存储。

行存储

行存储在微块中，一般情况下，行的所有列都是存储在一起的。当表中有大对象类型时，其所占用的存储空间会超过宏块大小，此时会将大对象类型溢出存储到其他的宏块中。

RowIDs

OceanBase 数据库中只有聚集索引表的存储模型，ROWID 实际包含的数据是表的存储键列。

空值存储

OceanBase 数据库中会存储空值，在存储中会使用一个字节来标记某列是否为空。

3.1.2.5. 表压缩

OceanBase 数据库支持两种方式进行压缩数据，一种是采用数据编码的方式，另一种是使用通用的压缩算法，包括 lz4_1.0、zstd_1.0 等。

OceanBase 数据库在建表或者修改表属性时，通过指定 `COMPRESS` 的级别来设置是否打开数据编码，以及采用哪种压缩算法。相关 SQL 语句如下：

- 关闭数据编码，并且关闭通用压缩算法。

```
CREATE TABLE table_name NOCOMPRESS;

ALTER TABLE table_name [MOVE] NOCOMPRESS;
```

- 关闭数据编码，使用 lz4_1.0 压缩算法。

```
CREATE TABLE table_name COMPRESS [BASIC];

ALTER TABLE table_name [MOVE] COMPRESS [BASIC];
```

- 关闭数据编码，使用 lz4_1.0 压缩算法。

```
CREATE TABLE table_name COMPRESS FOR OLTP;

ALTER TABLE table_name [MOVE] COMPRESS FOR OLTP;
```

- 打开数据编码，使用 zstd_1.0 压缩算法。

```
CREATE TABLE table_name COMPRESS FOR QUERY [LOW|HIGH];

ALTER TABLE table_name [MOVE] COMPRESS FOR QUERY [LOW|HIGH];
```

- 打开数据编码，使用 zstd_1.0 压缩算法。

```
CREATE TABLE table_name COMPRESS FOR ARCHIVE [LOW|HIGH];

ALTER TABLE table_name [MOVE] COMPRESS FOR ARCHIVE [LOW|HIGH];
```

详细信息请参考 压缩与编码。

3.1.2.6. 分区表

OceanBase 数据库支持普通表和分区表。

在 OceanBase 数据库中，分区是指根据一定的规则，把一个表或者索引分解成多个更小的、更容易管理的部分。每个分区都是一个独立的对象，具有自己的名称和可选的存储特性。

分区表由一个或多个分区组成，这些分区是单独管理的，可以独立于其他分区运行。表包括已分区或未分区。即使已分区表仅由一个分区组成，该表也不同于未分区表，非分区表不能添加分区。

OceanBase 数据库将每个表分区的数据存储在自己的 SStable 中。每个 SStable 包含表数据的一部分。

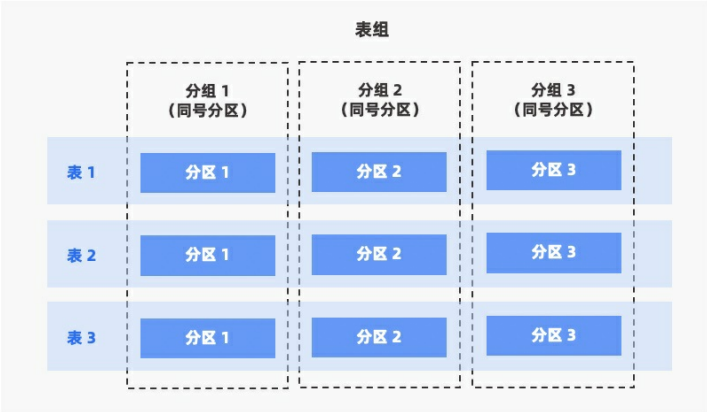
OceanBase 数据库提供了多种分区策略来使用多种业务的需求。由于分区是完全透明的，业务不需要进行过多的修改，就可以将分区功能应用到绝大多数业务。

3.1.2.7. 表组

对于分布式数据库，多个表中的数据可能会分布在不同的机器上，这样在执行 Join 查询或跨表事务等复杂操作时就需要涉及跨机器的通信，而表组功能可以避免这种跨机器操作，从而提高数据库性能。本文主要介绍表组功能及其工作原理。

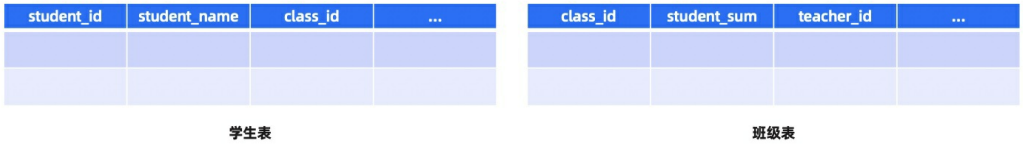
作为分布式数据库，为了满足扩展性和多点写入等需求，OceanBase 数据库支持分区功能，即将一个表的数据分成多个分区来存储。表中用来参与具体数据分区的列称为分区键，通过一行数据的分区键值，对其进行 Hash 计算（数据分区的方式有多种，这里以 Hash 分区为例），能够锁定其所属的分区。

让用户将分区方式相同的表聚集到一起就形成了表组（以 Hash 分区为例，分区方式相同等价于分区个数相同，当然计算分区的 Hash 算法也是一样的），表组内每个表的同号分区称为一个分区组，如下图所示。



OceanBase 数据库在分区创建以及之后可能发生的负载均衡时，会将一个分区组的分区放到一个机器，这样即便存在跨表操作，只要操作数据所在的分区是属于同一个分区组，那么就不存在跨机器的操作。那么如何保障操作的数据在同一个分区组呢？OceanBase 数据库无法干涉用户的操作，但是可以根据业务特点大概率地保障某些操作涉及的跨表数据在同一分区组中。

以学生表和班级表为例，学生表和班级表中都有班级 `id`，可以将班级 `id` 列作为分区键，两表中同一班级的数据聚集到同一个分区中，这样在对班级 `id` 列作 JOIN 的时候，只需要在这个分区所在的机器上处理，不需要所有分区跨机锁定某一班级 `id` 对应的数据。对于分布式写事务的场景，如果增加一个同学信息，需要在学生表中增加一条数据，并在班级表中更新学生总数。因为这两个数据不在一个表，自然也不在一个分区，因此需要执行分布式写事务才能进行一致的更新。由于两个分区在同一台 OBServer，OceanBase 数据库对于同一台 OBServer 上的分布式事务执行优化，因此相对跨机的分布式事务效率更高。



因为分区方式相同的表才能聚集到一个表组中，所以在表组里的表不支持打破分区规则的分区操作，但是可以通过对表组作分区操作来更改其内所有表的分区。

目前的表组功能设计中，一个分区对应一个日志流，日志流通过 Paxos 算法将数据变动日志由主副本同步到备副本上，如果涉及多分区的事务，就对应多个日志流的写入，因此需要分布式事务才能完成一致的操作。尽管单机分布式事务相对跨机已经有一些优化，OceanBase 数据库支持绑定表组作为进一步的优化方式：分区组的分区不仅在一台机器上，而且将分区的改动写在一个日志流里，这样对一个分区组的跨表写事务就可以在一个日志流里原子的提交，将分布式事务优化为单机事务，可以达到更好的优化效果。由于分区组对应一个日志流，导致表从表组里删除变得很困难，OceanBase 数据库目前还不支持绑定表组来删除表。用户可以根据特点灵活使用表组。

综上，在典型的业务场景下，OceanBase 数据库支持的表组功能优化了分布式查询和分布式事务的场景，但是也引入一些限制。例如，不支持表组里的表单独做分区更改操作，在此基础上，又支持绑定表组，这使得在典型分布式事务场景下可以拥有更好的性能，但是也增加了一些限制，例如不支持从表组中删除表。

3.1.2.8. 主键表和无主键表

OceanBase 数据库支持主键表和无主键表。本文主要介绍主键以及有主键表和无主键表的使用规则。

主键（Primary Key）

主键是在数据库的表中能够唯一标识一行的列的集合。主键需要满足以下规则：

- 值不能为 `NULL` 或空串
- 在全表范围内主键列集合的值唯一
- 主键的值不允许变更

有主键表

有主键表即表中包含主键的表，在 OceanBase 数据库中需要满足以下规则：

- 每个表最多拥有一个主键列集合
- 主键列的数量不能超过 64 列，且主键数据总长度不能超过 16 KB

在创建有主键表后，会自动为主键列创建一个全局唯一索引，可以通过主键快速定位到行。

如下例所示，创建了一个以 `emp_id` 为主键的表 `emp_table`，它属于有主键表。

```
CREATE TABLE emp_table (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(100),
  emp_age INT NOT NULL
);
```

无主键表

表中未指定主键的表称为无主键表，由于没有全局索引，无主键表通常使用 `ROWID` 来快速定位到行。

如下例所示，表 `student_table` 未指定主键，它属于无主键表。

```
CREATE TABLE student_table (
  student_id INT NOT NULL,
  student_name VARCHAR(100),
  student_age INT NOT NULL
);
```

OceanBase 数据库的无主键表采用自增列作为隐藏主键。无主键表利用了自增列的多分区全局唯一的原则，以此保证无主键表隐藏主键的唯一性。

OceanBase 数据库的自增列是兼容 MySQL 的自增列模块，满足以下三个原则：

- 多分区全局唯一
- 语句内连续递增
- 生成的自增列值大于用户显式插入的值

3.1.2.9. 临时表

临时表保存仅在事务或会话期间存在的数据。本文主要介绍 OceanBase 数据库的全局临时表（GTT）及其创建方法。

全局临时表简介

全局临时表用于保存一段时间内的数据，这里的一段时间可以是事务的生命周期也可以是 Session 的生命周期。

全局临时表可以对所有 Session 可见。但是，全局临时表中的数据是 Session 私有的，当前 Session 只能看到和修改自己的数据。

例如，课程安排应用程序使大学生能够创建可选的学期课程表。全局临时表中的一行代表每个课程安排。在会话期间，课程安排数据是私有的。当学生确认课程安排时，应用程序将所选时间表的行移动到普通表中。在会话结束时，数据库会自动删除全局临时表中的数据。

全局临时表的创建方法

用户可以通过 `CREATE GLOBAL TEMPORARY TABLE` 语句创建全局临时表，通过 `ON COMMIT` 子句控制是事务临时表还是 Session 临时表。

全局临时表可以创建索引，全局临时表的数据也是临时性的，数据也是 Session 私有的。

示例：创建一个事务级临时表。

```
obclient> CREATE GLOBAL TEMPORARY TABLE tbl1(col1 INT) ON COMMIT DELETE ROWS;
Query OK, 0 rows affected
```

3.1.3. 索引

3.1.3.1. 索引简介

索引也叫二级索引，是一种可选的结构，用户可以根据自身业务的需求，来决定在某些字段创建索引，从而加快在这些字段的查询速度。本章节主要介绍使用索引的优点和缺点、索引的可用性和可见性以及索引和键的关系。

OceanBase 数据库采用的聚集索引表模型，对于用户指定的主键会自动生成主键索引，而对于用户创建的其他索引，则是二级索引。

如下例所示，创建表 `employee` ，并插入三组数据。

```
obclient> CREATE TABLE employee(id INT, name VARCHAR(20), PRIMARY KEY(id));
Query OK, 0 rows affected

obclient> INSERT INTO employee VALUES(1,'John'),(2,'Alice'),(3,'Bob');
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0

obclient> SELECT * FROM employee;
+----+-------+
| ID | NAME  |
+----+-------+
|  1 | John  |
|  2 | Alice |
|  3 | Bob   |
+----+-------+
3 rows in set
```

在 `employee` 表中，按照用户指定的 `id` 有序存储数据，在查找数据时，可以根据 `id` 以二分查询的方式快速定位到具体的数据。如果需要按照 `name` 字段快速查找，可以在 `name` 字段建立一个二级索引，如下例所示：

```
CREATE INDEX name_index ON employee(name);
```

索引表中的数据如下：

```
name: Alice, id: 2
name: Bob, id :3
name: John, id: 1
```

在索引表 `name_index` 中，按照 `name` 字段有序存储，当用户指定 `name` 字段查询时，能以二分查询的方式快速地定位到具体的数据。

索引的优缺点

索引的优点如下：

- 用户可以在不修改 SQL 语句的情况下，可以加速查询，只需要扫描用户所需要的部分数据。
- 索引存储的列数通常较少，可以节省查询 IO。

索引的缺点如下：

- 选择在什么字段上创建索引需要对业务和数据模型有较深地理解。
- 当业务发生变化时，需要重新评估以前创建的索引是否满足需求。
- 写入数据时，需要维护索引表中的数据，消耗一定的性能代价。
- 索引表会占用内存、磁盘等资源。

索引的可用性和可见性

索引可用性

在 Drop Partition 场景，如果没有指定 `rebuild index` 字段，会将索引标记为 `UNUSABLE` ，即索引不可用。此时，在 DML 操作中，索引是无须维护的，并且该索引也会被优化器忽略。

索引可见性

索引的可见性是指优化器是否忽略该索引，如果索引是不可见的，则优化器会忽略该索引，但在 DML 操作中索引是需要维护的。一般在删除索引前，可以先将索引设置成不可见，来观察对业务的影响，如果确认无影响后，再将索引删除。

索引和键的关系

键是指一组列或者表达式，用户可以在键上创建索引。但索引和键是不同的，索引是存储在数据中的对象，而键是逻辑上的概念。

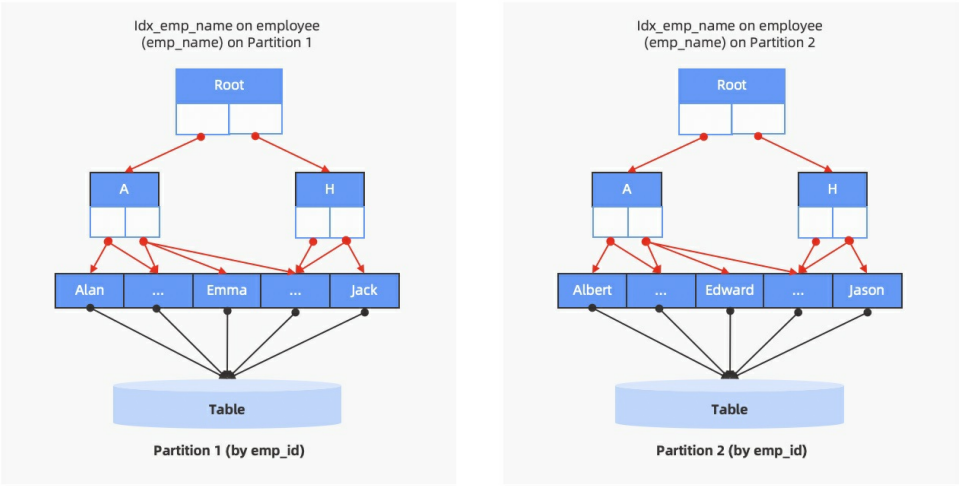
3.1.3.2. 局部索引和全局索引

OceanBase 数据库支持局部索引和全局索引。本文主要介绍局部索引和全局索引的概念以及创建索引时的默认行为。

局部索引

分区表的局部索引和非分区表的索引类似，索引的数据结构还是和主表的数据结构保持一对一的关系，但由于主表已经做了分区，主表的每一个分区都会有自己单独的索引数据结构。对每一个索引数据结构来说，里面的键（Key）只映射到自己分区中的主表数据，不会映射到其它分区中的主表，因此这种索引被称为局部索引。

从另一个角度来看，这种模式下索引的数据结构也做了分区处理，因此有时也被称为局部分区索引（Local Partitioned Index）。局部索引的结构如下图所示。



在上图中，`employee` 表按照 `emp_id` 做了 Range 分区，同时也在 `emp_name` 上创建了局部索引。

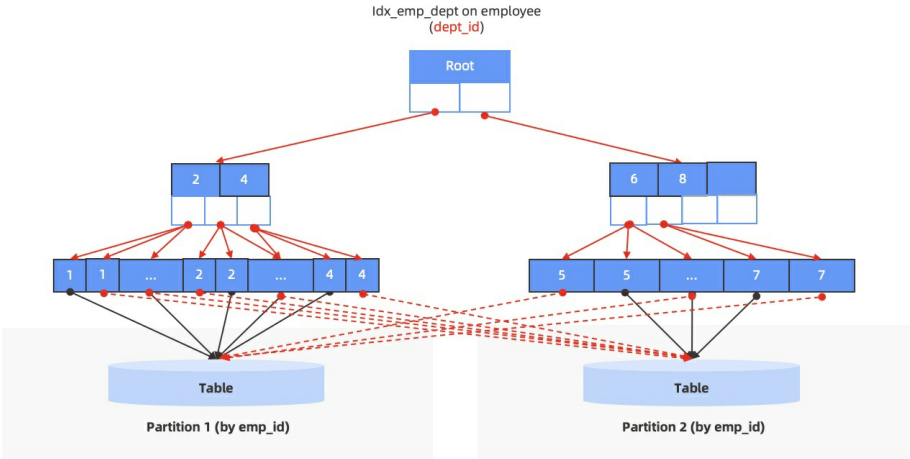
全局索引

和分区表的局部索引相比，分区表的全局索引不再和主表的分区保持一对一的关系，而是将所有主表分区的数据合成一个整体来看，索引中的一个键可能会映射到多个主表分区中的数据（当索引键有重复值时）。更进一步，全局索引可以定义自己独立的数据分布模式，既可以选择非分区模式也可以选择分区模式；在分区模式中，分区的方式既可以和主表相同也可以和主表不同。

因此，全局索引又分为以下两种形式：

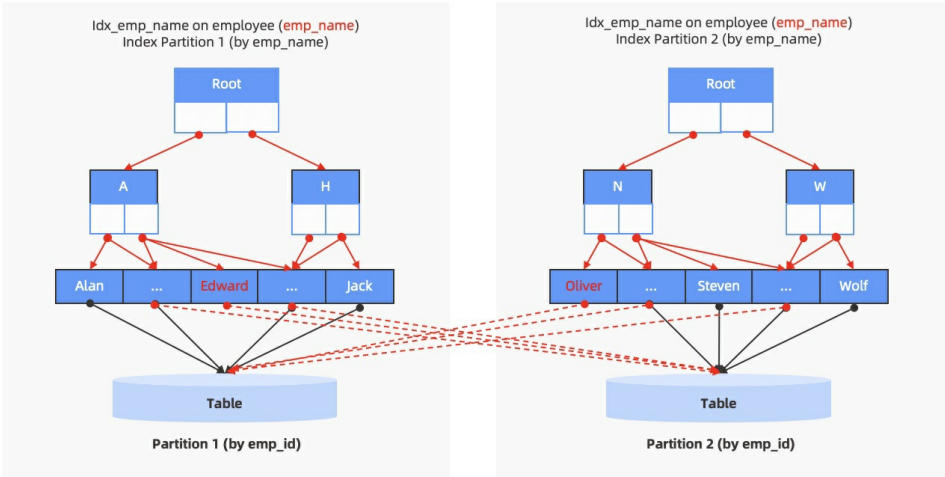
- 全局非分区索引（Global Non-Partitioned Index）

  索引数据不做分区，保持单一的数据结构，和非分区表的索引类似。但由于主表已经做了分区，因此会出现索引中的某一个键映射到不同主表分区的情况，即一对多的对应关系。全局非分区索引的结构如下图所示。



- 全局分区索引（Global Partitioned Index）

  索引数据按照指定的方式做分区处理，例如做哈希（Hash）分区或者范围（Range）分区，将索引数据分散到不同的分区中。但索引的分区模式是完全独立的，和主表的分区没有任何关系，因此对于每个索引分区来说，里面的某一个键都可能映射到不同的主表分区（当索引键有重复值时），索引分区和主表分区之间是多对多的对应关系。全局分区索引的结构如下图所示。



在上图中，`employee` 表按照 `emp_id` 做了 Range 分区，同时在 `emp_name` 上做了全局分区索引。可以看到同一个索引分区里的键，会指向不同的主表分区。

由于全局索引的分区模式和主表的分区模式完全没有关系，看上去全局索引更像是另一张独立的表，因此也会将全局索引叫做索引表，理解起来会更容易一些（和主表相对应）。

> **说明**
>
> 非分区表也可以创建全局分区索引。但如果主表没有分区的必要，通常来说索引也就没有必要分区了。

推荐使用全局索引的场景包括：

- 业务上除了主键外，还有其他列的组合需要满足全局唯一性的强需求，这个业务需求仅能通过全局性的唯一索引来实现。
- 业务的查询无法得到分区键的条件谓词，且业务表没有高并发的同时写入，为避免进行全分区的扫描，可以根据查询条件构建全局索引，必要时可以将全局索引按照新的分区键来分区。

需要注意的是，全局索引虽然为全局唯一、数据重新分区带来了可能，解决了一些业务需要根据不同维度进行查询的强需求，但是为此付出的代价是每一笔数据的写入都有可能变成跨机的分布式事务，在高并发的写入场景下它将影响系统的写入性能。当业务的查询可以拥有分区键的条件谓词时，OceanBase 数据库依旧推荐构建局部索引，通过数据库优化器的分区裁剪功能，排除掉不符合条件的分区。这样的做法可以同时兼顾查询和写入的性能，让系统的总体性能表现更优。

创建索引时的默认行为

当用户创建索引时，如果没有指定 `LOCAL` 或者 `GLOBAL` 关键字，在分区表上会默认会创建出全局索引。

3.1.3.3. 唯一索引和非唯一索引

索引表分为唯一索引和非唯一索引，其中唯一索引保证在表内的索引列上不存在两行有完全相同的值，而非唯一索引则可能存在完全相同的值。在 OceanBase 数据库中，NULL 值也会存储在索引中。

对于非唯一索引，索引表的存储键是用户指定的索引列和主表主键；而对于唯一索引列，索引表的存储键是用户指定的索引列和可变的主表主键列，这里可变的含义是主键列的值是随索引列值的不同而不同。

如下例所示，对于非唯一索引表 `i1` ，它的存储键是 `c2` 和 `c1` ，而对于唯一索引表 `i2` ，它的存储键是 `c2` ，而 `c1` 是可变的。当 `c2` 为 `NULL` 时，可变的 `c1` 列值是主表 `c1` 列的值；当 `c2` 不为 `NULL` 时，可变的 `c1` 列值为 `NULL` 。

```
CREATE TABLE t1(c1 INT, c2 INT, c3 INT, c4 INT, PRIMARY KEY(c1));
CREATE INDEX i1 ON t1(c2);
CREATE UNIQUE INDEX i2 ON t1(c3);
```

3.1.3.4. 索引的使用

当用户创建好索引后，OceanBase 数据库会自动维护该索引，所有的 DML 操作都会实时更新索引表相应的数据记录，同时优化器也会根据用户的查询来自动地选择是否使用索引。本文主要介绍如何进行索引扫描。

当 SQL 查询语句指定谓词条件查询的是索引列时，数据库会自动地抽取谓词条件作为查询索引的范围，也就是查询索引表的起始键和终止键。数据库根据起始键能定位到数据开始的位置，根据终止键能定位出数据结束位置，而开始和结束位置范围内所包含的数据是需要被此查询扫描的数据。

对于索引表，OceanBase 数据库存储时使用 MemTable 和 SSTable 来存储数据，其中 MemTable 使用的是 B+ 树结构，而 SSTable 使用的是宏块结构。在 MemTable 或者 SSTable 都按照上述扫描过程，扫描出相应的数据，而最终的数据行是由 MemTable 和 SSTable 的数据行融合成完整的数据行。

因此，OceanBase 数据库查询索引表数据的完整过程如下：

1. 在 MemTable 中查询数据。
2. 在 SSTable 中查询数据。
3. 将 MemTable 和 SSTable 中的数据融合，得到完整的行。

当 SQL 查询语句只涉及到索引表中的列时，那么数据库会根据用户指定的列，按照上述查询过程，查询相应索引表的 MemTable 和 SSTable，得到完整的数据行。

当 SQL 查询语句除了包含索引表中的列，还包含其他列时，数据库会先通过索引表，查询出相关的行，并根据行上的主键，按照上述查询过程，到主表中查询所需要的数据列，这个过程也称为回表。

3.1.3.5. 索引的存储

在 OceanBase 数据库中，索引表的存储类似于普通的数据表，其数据也是存储在宏块和微块的结构中。由于 OceanBase 数据库使用的是聚集索引表模型，因此在索引的行中除了用户指定的索引列之外，还会存储主表的主键列，以方便进行回表。

3.1.4. 分区

3.1.4.1. 分区概述

在 OceanBase 数据库中，分区是指根据一定的规则，把一个表分解成多个更小的、更容易管理的部分。每个分区都是一个独立的对象，具有自己的名称和可选的存储特性。本章节主要介绍分区的相关概念以及使用分区的好处。

对于访问数据库的应用而言，逻辑上访问的只有一个表或一个索引，但是实际上这个表可能由数十个物理分区对象组成，每个分区都是一个独立的对象，可以独自处理访问，也可以作为表的一部分处理访问。分区对应用来说是完全透明的，不影响应用的业务逻辑。

从应用程序的角度来看，只存在一个 Schema 对象。访问分区表不需要修改 SQL 语句。分区对于许多不同类型的数据库应用程序非常有用，尤其是那些管理大量数据的应用程序。

分区表可以由一个或多个分区组成，这些分区是单独管理的，可以独立于其他分区运行。表可以是已分区或未分区的，即使已分区表仅由一个分区组成，该表也不同于未分区表，非分区表不能添加分区。

分区表也可以由一个或多个表分区段组成。OceanBase 数据库将每个表分区的数据存储在自己的 SStable 中，每个 SStable 包含表数据的一部分。

使用分区的好处如下：

- 提高了可用性

  分区不可用并不意味着对象不可用。查询优化器自动从查询计划中删除未引用的分区。因此，当分区不可用时，查询不受影响。

- 更轻松地管理对象

  分区对象具有可以集体或单独管理的片段。DDL 语句可以操作分区而不是整个表或索引。因此，可以对重建索引或表等资源密集型任务进行分解。例如，可以一次只移动一个分区。如果出现问题，只需要重做分区移动，而不是表移动。此外，对分区进行 `TRUNCATE` 操作可以避免大量数据被 `DELETE` 。

- 减少 OLTP 系统中共享资源的争用

  在 TP 场景中，分区可以减少共享资源的争用。例如，DML 分布在许多分区而不是一个表上。

- 增强数据仓库中的查询性能

  在 AP 场景中，分区可以加快即席查询的处理速度。分区键有天然的过滤功能。例如，查询一个季度的销售数据，当销售数据按照销售时间进行分区时，仅仅需要查询一个分区或者几个分区，而不是整个表。

- 提供更好的负载均衡效果

  OceanBase 数据库的存储单位和负载均衡单位都是分区。不同的分区可以存储在不同的节点。因此，一个分区表可以将不同的分区分布在不同的节点，这样可以将一个表的数据比较均匀的分布在整个集群。

3.1.4.2. 分区键

分区键是一个或多个列的集合，用于确定分区表中的每一行所在的分区。每一行都明确地分配给一个分区。

例如，对于一个用户表，可以指定 `user_id` 列作为 Range 分区的键。数据库根据此列中的用户号是否在指定范围内，将行分配给分区。

OceanBase 数据库使用分区键自动将插入、更新和删除操作定向到相应的分区。

3.1.4.3. 分区类型

分区策略

OceanBase 数据库提供了多种分区策略，用于控制数据库如何将数据放入分区。

OceanBase 数据库的基本分区策略包括范围（Range）分区、列表（List）分区和哈希（Hash）分区。

一个一级分区仅限使用一种数据分配方法。例如，仅使用 List 分区或仅使用 Range 分区。

在进行二级分区时，表首先通过一种数据分配方法进行分区，然后使用第二种数据分配方法将每个分区进一步划分为二级分区。例如，一个表中包含 `create_time` 列和 `user_id` 列，您可以在 `create_time` 列上使用 Range 分区，然后在 `user_id` 列上使用 Hash 进行二级分区 。

Range 分区

Range 分区是最常见的分区类型，通常与日期一起使用。在进行 Range 分区时，数据库根据分区键的值范围将行映射到分区。

每个 Range 分区都包含一个 `VALUES LESS THAN` 子句，该子句用于指定分区的非包容性上限，等于或高于此文本的分区键的任何值都将添加到下一个更高的分区。除了第一个分区之外，所有分区都有一个由上一个分区的 `VALUES LESS THAN` 子句指定的隐式下限。

Range 分区支持为最高分区定义 `MAXVALUE` 。`MAXVALUE` 表示一个虚拟无限值，其排序高于分区键的任何其他可能值，包括空值。

Hash 分区

在进行 Hash 分区时，数据库根据数据库应用于用户指定的分区键的哈希算法将行映射到分区。行的目标分区是由内部 Hash 函数计算出一个 Hash 值，再根据 Hash 分区个数来确定的。当分区数量为 2 的幂次方时，哈希算法会创建所有分区中大致均匀的行分布。Hash 算法在分区之间均匀分布行，使分区的大小大致相同。

Hash 分区是在节点之间均匀分布数据的理想方法。Hash 分区也是范围分区的一种易于使用的替代方法，特别是当要分区的数据不是历史数据或没有明显的分区键的场景。

Hash 分区在具有极高更新冲突的 OLTP 系统里面非常有用。这是因为 Hash 分区将一个表分成几个分区，将一个表的修改分解到不同的分区修改，而不是修改整个表。

List 分区

在进行 List 分区时，数据库使用离散值列表作为每个分区的分区键。分区键由一个或多个列组成。

可以使用 List 分区来控制单个行如何映射到指定分区。

当不方便根据分区键进行排序时，可以使用 List 分区对数据进行分组和管理。

组合分区

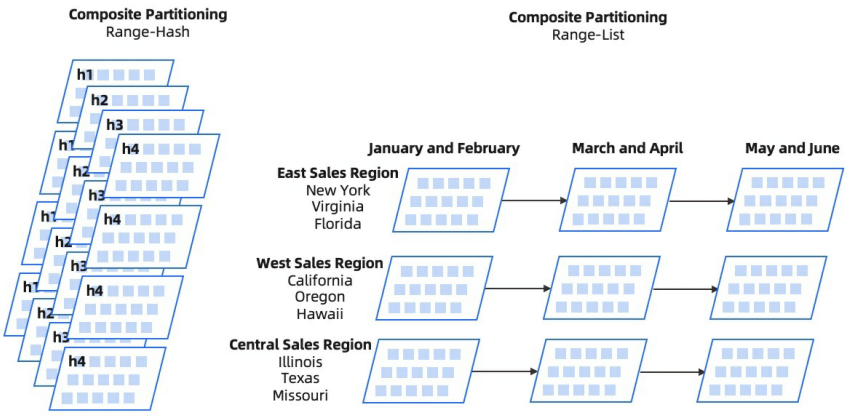
范围（Range）分区、列表（List）分区和哈希（Hash）分区都可以作为组合分区表的二级分区策略。

在进行组合分区时，表通过一种数据分配方法进行分区，然后使用第二种数据分配方法将每个分区进一步细分为二级分区。因此，组合分区结合了基本的数据分发方法。指定分区的所有二级分区代表数据的逻辑子集。

组合分区有如下优点：

- 根据 SQL 语句，在一维或二维上进行分区修剪可能会提高性能。
- 查询可以在任一维度上使用全分区或部分分区连接。
- 您可以对单个表执行并行备份和恢复。
- 分区的数量大于单层分区，这可能有利于并行执行。
- 您可以实现一个滚动窗口来支持历史数据，如果许多语句可以从分区修剪或分区连接中受益，则仍然可以在另一个维度上进行分区。
- 您可以根据分区键的标识以不同方式存储数据。例如，您可能决定以只读的压缩格式存储特定产品类型的数据，并保持其他产品类型的数据不压缩。

下图展示了 Range-Hash 和 Range-List 组合分区。



3.1.4.4. 分区索引

在数据库中，用户可以通过建立索引的方式来提升查询性能，当数据表中的数据量越来越大时，则可以通过分区的方式将数据表拆分成若干分片，利用负载均衡将数据分片打散到整个集群中来提高集群整体服务能力。本文主要介绍局部索引、全局索引和唯一索引。

索引类型

OceanBase 数据库主要涉及如下索引类型：

- 局部分区索引：在创建索引时指定关键字 `LOCAL` 的索引为局部索引，局部索引无须指定分区规则，它的分区属性和主表的属性一致，也会跟随主表的分区操作而发生变更。
- 全局分区索引：在创建索引时指定关键字 `GLOBAL` 的索引为全局索引，全局索引可以按照分区规则进行分表。
- 唯一索引：索引键值具有唯一性，用关键字 `UNIQUE` 表示。
- 前缀索引：当分区索引表的分区键是索引列的左前缀时，该索引称为前缀索引。这不同于 MySQL 长字符串的前缀索引。例如，索引表 `idx` 建立在 `c1` 和 `c2` 列上，如果该索引表的分区键为 `c1`，那么该索引称为前缀索引；如果该索引表的分区键为 `c2` 或者其他列，则称为非前缀索引。
- 非前缀索引：相对于分区前缀索引而言，如果分区索引不是前缀索引，那么称为非前缀索引。

局部索引

局部索引根据分区键划分为局部前缀索引和局部非前缀索引。

局部前缀索引

如果分区键是索引的左前缀，并且索引包含二级分区键，那么这个索引为局部前缀索引。局部前缀索引可以是唯一索引或者非唯一索引。

指定索引键的查询利用局部前缀索引可以唯一定位到一个索引分区，非常适合于结果集比较小，但是需要进行分区裁剪的场合。

例如，表 A 上有个局部索引 `idx(c1,c2,c3)` ，主表根据 `c1` 进行分区，根据局部索引的特性，索引表和主表的分区方式一样，因此 `idx` 也以 `c1` 为分区键，根据定义知道这是一个局部前缀索引，当处理指定索引键的查询时，可以通过分区键 `c1` 的值定位到唯一一个索引分区，大大减少了索引分区的访问。

局部非前缀索引

如果索引表不是局部前缀索引，那就是局部非前缀索引。可能的情况是，索引表的分区键不是索引的左前缀，或者索引没有包含二级分区键。

如果分区键不是索引的子集，那么局部非前缀索引不能是唯一索引。 指定索引键的查询利用局部非前缀索引，不能定位到索引分区，而是需要访问所有索引分区，因此比较适合访问数据量比较大，注重并发的场景。

例如，表 A 上有个局部索引 `idx(c1,c2,c3)` ，主表根据 `c4` 列进行分区，根据定义获知这是一个局部非前缀索引，当用户查询指定索引键时，不能通过分区键定位到索引分区，因此需要访问所有的索引分区才能获得结果，并发执行在这种场景下可以发挥重要的作用。

全局索引

全局索引拥有自己独立的分区定义，不需要跟主表一样。同时，全局索引表的分区也可以进行分裂和合并。一般情况下，如果主表和全局索引的分区方式完全一样的话，除去具有唯一性的非前缀索引，其他索引建议定义成局部索引，全局索引在分区管理和维护上代价要远远大于局部索引，并且从查询代价、分区裁剪上来说，具有相同分区方式的全局索引和局部索引的效果是一样的。

全局前缀索引

如果全局分区索引的分区键是索引键的左前缀，那么这个索引称为全局前缀索引。

全局前缀索引可以是唯一索引或者非唯一索引。

全局前缀索引只在用 Range 分区时有意义，对于 Hash 分区索引无意义。原因在于，如果是用户选择 Hash 分区索引，那么用户查询模式一定是指定索引键的点查询，索引键如果覆盖分区键的话，那么是否为前缀索引并无意义，都能够通过用户指定的索引键值算出索引分区；如果用户没有指定全部分区键值，Hash 分区索引则需要访问所有的分区数据，而 Range 分区可以进行一定程度的分区裁剪。

全局非前缀索引

OceanBase 数据库不支持全局非前缀索引，全局非前缀索引对于查询优化并没有太多意义。

例如，表 A 上有一个全局索引 `idx(c1,c2)` ， `idx` 通过 `c2` 进行分区，那么 `idx` 为一个非全局索引。这种情况下，只有当用户指定全部索引键值的时候才能进行分区裁剪，其他情况均需要扫描所有的索引分区，因此用户没有理由不直接用 `c1` 键做分区，使用 `c1` 还能通过前缀过滤进行分区裁剪。

唯一索引

唯一索引可以定义为全局索引或者是局部索引。

唯一索引定义为局部索引的时候，需要满足一定的条件，即索引键要覆盖索引分区键。 例如，表 A 上有个唯一索引 `idx(c1,c2,c3)` ，索引表 `idx` 按照 `(c1,c2)` 进行分区，这样能保证相同的 `(c1,c2)` 一定能进入同一分区，只需要在单个分区内维护唯一性。如果索引表 `idx` 按照 `(c2,c4)` 进行分区，索引没有覆盖分区键，就不能通过局部索引来保证分区之间索引键的唯一性，这样的局部唯一索引不能被创建。

索引创建策略

在创建索引的时候，需要将用户查询模式、索引管理、性能、可用性等方面的需求综合起来考虑，选择一个最合适自身业务的索引方式。

- 如果用户需要唯一索引，并且索引键覆盖所有分区键，则可以定义成局部索引，否则需要用全局索引。
- 如果主表的分区键是索引的子集，则可以采用局部索引。
- 如果主表分区属性和索引的分区属性相同，建议采用局部索引。
- 如果用户比较关注索引分区管理的代价，主表分区总是不断进行分区删减时，尽量避免创建全局索引。这是由于主表分区删减操作会导致全局索引变更太大，难以恢复，而且可能会导致索引不可用。
- 如果用户查询总是指定所有索引分区键值，那么只需要在其他列上建索引，而无须包含分区键，减少维护代价和存储代价；对于不指定分区键的查询，前缀索引更合适做分区裁剪、数据量比较小的场景，前缀索引一般更适合做分区并行、数据量比较大的场景。

分区选择策略

在进行索引分区的时候，如果单个索引表的数据量太大则需要进行分区，以便查询更好的进行分区并行以及负载均衡。

- 如果用户查询多为指定索引键的单点查询，从访问分区的数目和访问并发角度来说，Hash 分区和 Range 分区的查询代价都相差不大。但是如果数据具有热点区的话，用 Hash 分区能更好的避免热点问题。
- 如果用户的查询多为指定索引键的范围查询，则从访问的分区数目来说，Range 分区要优于 Hash 分区；但从并发的角度考虑，Hash 分区可以更好的利用并发查询的优势，在查询结果集很大的情况下，Hash 分区的并发查询应该能有更大的性能优势。

查询时索引选择策略

- 如果用户更注重分区裁剪的话，选择前缀索引更好，在用查询过滤条件里面指定分区前缀的值能够最大程度的进行分区裁剪，减少索引分区数据的读取。
- 如果用户注重吞吐量，访问的数据量比较大时，选择非前缀索引更能提高性能，可以通过分区并行的方式来解决分区键上的范围查询；局部非前缀索引可以并发访问所有的分区，但是局部前缀索引会进行分区裁剪，由少量分区处理大量的数据集合，响应时间也许会比并发查询要差一些。

3.1.5. 视图

3.1.5.1. 视图概述

视图是从一个或多个表导出的虚拟的表。本质上，视图是一个存储好的查询，用户通过访问这个视图来获取该视图定义的数据。本章节主要介绍视图相关的基本概念和使用方法。

视图从它所基于的表（称为基表）中获取数据。基表可以是表或其他视图。对视图执行的所有操作实际上都会影响基表。您可以在使用表的大多数场景中使用视图。

> **注意**
>
> 物化视图与标准视图使用不同的数据结构。

视图的优势

视图使您能够为不同类型的用户定制数据的呈现方式。视图拥有如下优势：

- 把经常使用的数据定义为视图以简化操作。数据库的查询大多要使用聚合函数，同时还要显示其它字段的信息，可能还会需要关联到其它表，这时涉及的 SQL 语句可能比较复杂。如果需要频繁执行此查询，就可以通过创建视图简化查询操作，之后只需要执行 `SELECT * FROM view_name` 就可以获得预期结果。
- 视图限制用户只能查询和修改可视的数据，提高了数据的安全性。视图是动态的数据集合，数据随着基表的更新而更新。但是视图是虚拟的，物理上是不存在的，只是存储了数据的集合，所以可以将基表中重要的字段信息屏蔽，不通过视图展示给用户。
- 视图拥有逻辑上的独立性，屏蔽了真实表的结构带来的影响。视图可以使应用程序和数据库的表在一定程度上互相独立。如果没有视图，应用一定是建立在表上的。创建了视图之后，程序可以建立在视图之上，从而程序与数据库表被视图分割开来。

视图的特点

与表不同，视图没有分配存储空间，视图也不包含数据。相反，视图是从视图引用的基表中提取或派生数据。因此除了用于定义数据字典中视图的查询的存储之外，它不需要其他存储。

视图依赖于其引用的对象，如果视图所依赖的对象被删除、更新或者重建，则数据库会确定新对象是否可以被视图定义所接受。

3.1.5.2. 视图的数据操作

因为视图是从表中派生出来的，所以它和表有很多相似之处。用户可以查询视图，并且在一定限制下，对视图执行 DML 操作。对视图执行的操作会影响视图的某些基表中的数据，并受基表的完整性约束和触发器的约束控制。

如下示例中，在表 `employees` 上定义视图 `staff_dept_10` 。这个视图定义了部门是 10 的员工信息。

使用关键字 `CHECK OPTION` 创建视图，表明该视图上的 DML 操作只能作用在视图所定义的数据中。因此，可以插入部门为 10 中的员工的行数据，但不能插入部门为 30 中的员工的行数据。

```
CREATE VIEW staff_dept_10 AS
   SELECT employee_id, last_name, job_id, manager_id, department_id
   FROM   employees
   WHERE  department_id = 10
   WITH CHECK OPTION CONSTRAINT staff_dept_10_cnst;
```

3.1.5.3. 视图的数据访问

OceanBase 数据库会在内部的数据字典中保存视图对应的查询语句。

当用户尝试通过视图读取数据时，OceanBase 数据库会执行以下操作步骤：

1. 解析用户查询，如果在解析过程中遇到视图名称时，从数据字典中获取视图对应的查询语句并解析。
2. 尝试将该视图与用户查询合并，合并之后有可能生成更好的执行计划。
3. 生成执行计划，并执行该语句。

示例：

1. 创建一个名为 `staff_dept_10` 的视图，该视图的定义如下所示。

```
CREATE VIEW staff_dept_10 AS
    SELECT employee_id, last_name, job_id, manager_id, department_id
    FROM   employees
    WHERE  department_id = 10
```

2. 用户执行以下查询访问 `staff_dept_10` 。

```
SELECT last_name FROM  staff_dept_10 WHERE  employee_id = 200;
```

3. OceanBase 数据库首先会将以上查询解析成如下查询。

```
SELECT last_name
FROM   (SELECT employee_id, last_name, job_id,
                      manager_id, department_id
                               FROM   employees
                               WHERE  department_id = 10) staff_dept_10
WHERE  employee_id = 200;
```

4. 查询改写会尝试将视图定义与主查询合并。

```
SELECT last_name
FROM  employees
WHERE  employee_id = 200 and department_id = 10;
```

5. OceanBase 数据库的 SQL 引擎会生成以上 SQL 对应的执行计划并执行。

3.1.5.4. 可更新的视图

用户除了通过视图读取数据外，还可以通过视图更新数据。这些被更新的视图称之为可更新视图。

例如，对于视图 `staff_dept_10` ，用户可以执行如下删除操作：

```
DELETE staff_dept_10 WHERE employee_id = 200;
```

OceanBase 数据库在接受到这个请求后，处理的流程如下：

1. 解析该语句，并将 `staff_dept_10` 替换成视图的定义。
2. 检查该视图作为被更新的对象，是否满足一些必要的约束。
3. 尝试将该视图合并到主语句中。
4. 生成计划并执行。

在 OceanBase 数据库内部，实际执行的删除语句是：

```
DELETE employees WHERE employee_id = 200 and department_id = 10;
```

一个视图是否可以被更新类语句修改需要满足一些必要的条件，如下：

- 一个修改类操作只能修改一张实体表。
- 对于被修改表中的每一行，在视图的查询中，每一行仅出现一次。多对多的连接查询不具备这样的性质。

- 被修改的视图没有 `WITH READ ONLY` 的标记。

3.1.6. 其他对象

3.1.6.1. 序列

序列（Sequence）是 Oracle 租户的数据库对象，可以产生不重复的有顺序的值，在表需要不重复的列做主键时很有用。本文主要介绍序列的基本特性以及 OceanBase 数据库与 Oracle 数据库序列的差异。

基本特性

序列可以提供两个伪列 `CURRVAL` 和 `NEXTVAL` ，每次查询会返回当前的序列值和下一个序列值。每当查询 `NEXTVAL` 都会推进 `CURRVAL` 值。

序列的定义里可以指定序列名、序列步长以及增序还是降序等。

`CREATE SEQUENCE` 语句用来创建序列，语法格式如下：

```
CREATE SEQUENCE sequence_name
  [MINVALUE value | NOMINVALUE]
  [MAXVALUE value | NOMAXVALUE]
  [START WITH value]
  [INCREMENT BY value]
  [CACHE value | NOCACHE]
  [ORDER | NOORDER]
  [CYCLE | NOCYCLE];
```

> **说明**
>
> - `MINVALUE` 和 `MAXVALUE` 指定最小值和最大值。
> - `START WITH` 指定起始值。
> - `INCREMENT BY` 指定步长，可以为负数，默认是 1。
> - `CACHE` 是为了性能缓存部分序列值，在并发高的时候使用。
> - `CYCLE` 指定序列值是否循环。如果循环，需要指定最大值或最小值。

序列创建成功后，可以通过 USER_SEQUENCES、ALL_SEQUENCES、DBA_SEQUENCES 视图查看自己创建的序列。

`ALTER SEQUENCE` 语句用来修改序列的属性。序列的起始值不能修改，其他属性如最小值、最大值、步长、循环属性都可以修改。语法格式如下：

```
ALTER SEQUENCE sequence_name
  [MINVALUE value | NOMINVALUE]
  [MAXVALUE value | NOMAXVALUE]
  [INCREMENT BY value]
  [CACHE value | NOCACHE]
  [ORDER | NOORDER]
  [CYCLE | NOCYCLE];
```

`DROP SEQUENCE` 语句用来删除序列，语法格式如下：

```
DROP SEQUENCE sequence_name;
```

序列是一个独立的对象，同一个序列可以用在不同的表上。如下示例中，第一个 `INSERT` 语句会向表 `t1` 中插入 1，第二个 `INSERT` 语句会向表 `t2` 中插入 2。

```
CREATE SEQUENCE s1 MINVALUE 1 MAXVALUE 100 INCREMENT BY 1;
INSERT INTO t1 (id) VALUES (s1.nextval);
INSERT INTO t2 (id) VALUES (s1.nextval);
```

注意事项

关于 CACHE 和 ORDER

关于 `CACHE` 和 `ORDER`，OceanBase 数据库与 Oracle 数据库序列的差异如下表所示。

模式	Oracle 数据库	OceanBase 数据库
NOCACHE with ORDER	所有 Instance 不缓存任何序列，使用时从全局 `CACHE` 中获取，`CACHE` 根据请求的先后顺序返回序列值。	所有 Instance 不缓存任何序列，使用时从全局 `CACHE` 中获取，`CACHE` 根据请求的先后顺序返回序列值。
NOCACHE with NOORDER	所有 Instance 不缓存任何序列，使用时从全局 `CACHE` 中获取，`CACHE` 可以根据繁忙程度延迟满足一些请求，导致 `NOORDER`。	仅语法兼容，实际效果与 `NOCACHE with ORDER` 相同。

模式	Oracle 数据库	OceanBase 数据库
CACHE with ORDER	每个 Instance 都缓存相同的序列，使用之前需要通过全局锁来同步序列中的下一个可用位置。	仅语法兼容，实际效果与 `NOCACHE with ORDER` 相同。
CACHE with NOORDER	每个 Instance 都缓存不同的序列，并且不需要全局同步 `CACHE` 状态。此时的值自然是 `NOORDER` 。	每个 Instance 都缓存不同的序列，并且不需要全局同步 `CACHE` 状态。此时的值自然是 `NOORDER` 。

说明

- 出于性能考虑，OceanBase 数据库建议您使用默认的 `CACHE with NOORDER` 方式。
- OceanBase 数据库中，不建议使用 `ORDER with NO CACHE` 。这种模式下，每次调用 `NEXTVAL` 都会触发一次内部表 `SELECT` 与 `UPDATE` 操作，会影响数据库的性能。
- 在创建序列时，由于默认的 `CACHE` 值只有 20，需要手动声明一个比较大的值。对于单机 TPS 为 100 时， `CACHE SIZE` 建议设置为 360000。

关于 CURRVAL

在一个节点上通过 `NEXTVAL` 取到的序列值，不支持在另一个节点上用 `CURRVAL` 获得。常见于 Proxy 将 `NEXTVAL` 获取的语句和 `CURRVAL` 获取的语句发给了不同节点的情况。为了避免这个情况，可以让 `NEXTVAL` 取值和 `CURRVAL` 取值在同一个事务中，确保 Proxy 会始终把 Query 发给同一个节点。

3.1.6.2. 同义词

同义词 （Synonym）是 Oracle 租户中表、视图、物化视图、序列、存储过程、函数、包、类型、用户自定义类型，或是其他的同义词的别名。由于其只是一个别名，所以除了在数据字典中的定义不占任何空间。本文主要介绍同义词的分类、基本特性和权限要求。

分类

同义词有两种类型，Public 同义词和 Private 同义词。

Public 同义词属于 `PUBLIC` 组，每个用户都可以访问。Private 同义词属于对象所有者，只有其显式授权后其他用户才可访问。

公有同义词一般由 DBA 创建，普通用户如果希望创建公有同义词，则需要 `CREATE PUBLIC SYNONYM` 系统权限。

基本特性

同义词扩展了数据库的使用范围，能够在不同的数据库用户之间实现无缝交互。经常用于简化对象访问和提高对象访问的安全性。

创建同义词

`CREATE SYNONYM` 语句用来创建同义词，语法格式如下：

```
CREATE [ OR REPLACE ] [ PUBLIC ]
SYNONYM [ schema. ]synonym
FOR [ schema. ]object;
```

参数说明

参数	描述
OR REPLACE	表示如果要创建的同义词名称已存在，则使用新的定义替换同义词。
PUBLIC	指定 `PUBLIC` 来创建公共同义词，所有用户都可以使用。用户必须对基础对象具有相应的权限才能使用该同义词。 在解析对象的引用时，仅当对象没有指定 Schema 时，才会使用公共同义词。 如果不指定 `PUBLIC` ，则同义词是私有的，只能由当前 Schema 访问，并且同义词名称在当前 Schema 中必须唯一。
[schema.]synonym	Schema 指定当前同义词属于哪个用户。如果指定了 `PUBLIC` ，则对同义词不能指定用户。 `synonym` 表示同义词的名称。
[schema.]object	表示同义词对应对象的名称。

同义词创建成功后，可以通过 USER_SYNONYMS、ALL_SYNONYMS、DBA_SYNONYMS 视图查看自己创建的同义词。

示例

- 创建一个 Private 同义词。

```
obclient> CREATE TABLE t1(c1 INT);
obclient> CREATE SYNONYM s1 FOR t1;
obclient> INSERT INTO s1 VALUES(1);
obclient> SELECT * FROM s1;
+------+
| c1   |
+------+
|    1 |
+------+
1 row in set
```

- 创建一个 Public 同义词。

```
obclient> CREATE PUBLIC SYNONYM syn_pub FOR t1;
obclient> SELECT * FROM syn_pub;
+------+
| c1   |
+------+
|    1 |
+------+
1 row in set
```

删除同义词

`DROP SYNONYM` 语句用来删除同义词，语法格式如下：

```
DROP [PUBLIC] SYNONYM [ schema. ]synonym;
```

参数说明

参数	描述
PUBLIC	指定 `PUBLIC` 来删除公共同义词。如果不指定 `PUBLIC` ，则删除私有同义词。
[schema.]synonym	Schema 指定当前同义词属于哪个用户。 如果指定了 `PUBLIC` ，则对同义词不能指定用户。 `synonym` 表示同义词的名称。

示例

- 删除一个 Private 同义词。

```
obclient> DROP SYNONYM test.s1;
Query OK, 0 rows affected
```

- 删除一个 Public 同义词。

```
obclient> DROP PUBLIC SYNONYM syn_pub;
Query OK, 0 rows affected
```

注意事项

创建同义词时，需要满足如下权限要求：

- 在当前的用户下创建私有的同义词，需要 `CREATE SYNONYM` 权限。
- 在非当前用户下创建私有的同义词，需要 `CREATE ANY SYNONYM` 权限。
- 创建 `PUBLIC` 的同义词，需要 `CREATE PUBLIC SYNONYM` 权限。
- 不需要具有要创建同义词的对象的访问权限。

```
/*连接 sys 用户*/
obclient> CREATE USER syn_user IDENTIFIED BY syn_user;
obclient> GRANT CREATE ON syn_user.* TO syn_user;
obclient> GRANT SELECT ON syn_user.* TO syn_user;

/*连接 syn_user，创建同义词失败*/
obclient> CREATE SYNONYM syn_1 FOR t1;
ERROR-00600: internal error code, arguments: -5036, Access denied; you need (at least one o
f) the CREATE SYNONYM privilege(s) for this operation

/*连接 sys 用户，授予 CREATE SYNONYM 权限*/
obclient> GRANT CREATE SYNONYM ON *.* TO syn_user;

/*连接 syn_user*/
obclient> CREATE SYNONYM syn_1 FOR t1;
```

3.1.7. 系统视图

3.1.7.1. 字典视图

OceanBase 数据库 Oracle 模式下的数据字典是一组只读的表，提供有关数据库的管理元数据。

字典视图简介

OceanBase 数据库 Oracle 模式下的数据字典由系统表和字典视图组成，它是数据库的重要组成部分。

系统表存储有关数据库的信息，只有 OceanBase 数据库能写入和读取这些表。用户很少直接访问系统表，因为它们是面向内部实现机制的，不易读，而且版本之间不保证外部兼容。

字典视图将系统表数据解码为有用的信息，例如将用户名或者表名作为联接键和 `WHERE` 子句的过滤条件来简化系统表信息。视图包含数据字典中所有对象的名称和描述。某些视图可供所有数据库用户访问，而其他视图仅供管理员使用。

数据字典包含但不限于如下信息：

- 数据库中每个 Schema 对象的定义，包括列的默认值和完整性约束信息。

- 为 Schema 对象分配和当前使用的存储空间。
- 数据库用户的名称、授予用户的权限和角色以及与用户相关的审计信息。

字典视图的组成

字典视图一般由下表所示的信息组成。

前缀	权限	内容	其他信息
DBA_	数据库管理员	所有对象	某些 `DBA_` 视图具有包含对管理员有用的信息的附加列。
ALL_	所有用户	用户持有权限的对象	包括用户拥有的对象。
USER_	所有用户	用户拥有的对象	带有前缀 `USER_` 的视图通常不包括列 `OWNER` 。

带有 ALL_ 前缀的视图

前缀为 `ALL_` 的视图用于用户对数据库的整体概览。除了用户拥有的 Schema 对象之外，这些视图还返回通过公开或显式授权给用户的 Schema 对象。

带有 DBA_ 前缀的视图

前缀为 `DBA_` 的视图显示整个数据库中的所有相关信息。 `DBA_` 视图需要使用管理员权限访问。

带有 USER_ 前缀的视图

普通数据库用户最常用的视图是前缀为 `USER_` 的视图。

此类视图具有如下特点：

- 在数据库中引用用户的私有环境，包括关于用户创建的模式对象、用户授予的权限等的元数据。
- 仅显示与用户相关的行，返回 `ALL_` 视图中信息的子集。
- 具有与其他视图相同的列，但是隐含列 `OWNER` 。
- 为方便起见，可以缩写 `PUBLIC` 同义词。

字典视图的工作机制

Oracle 模式下用户的 SYS 用户拥有数据字典的所有基表和用户可访问的视图。每个 Oracle 租户在创建时候，都默认会有 SYS 用户。

在数据库操作过程中，数据库系统读取数据字典，以确定 Schema 对象是否存在，以及用户是否有适当的访问权限。在 Oracle 模式下，通过不断更新数据字典来反映数据库结构、审核、授权和数据的变化。

例如，如果用户 `hr` 创建一个名为“实习生”的表，则数据库会向数据字典添加新行，以反映新表、列、段、数据块以及 `hr` 对该表拥有的权限。下次查询字典视图时，此新信息仍然可见。

数据字典基表中的数据是 Oracle 模式下正常运行所必需的。只有数据库本身应写入或更改数据字典信息。任何 Oracle 模式下用户都不应更改 SYS 用户的 Schema 中包含的行或 Schema 对象，因为此类操作可能会损害数据完整性，所以安全管理员必须严格控制。

字典视图的存储

在 OceanBase 数据库中，将每个租户元数据作为一个整体存储的数据字典仅存储在每个租户字典信息中。

由于存储指定租户元数据的数据字典存储在此租户专用的字典表中，因此每组数据字典都存储在自己租户的专用字典表中。

3.1.7.2. 性能视图

动态性能视图由数据库服务器维护，可供数据库管理员用户 SYS 访问，会在打开和使用数据库时不断更新，并且其内容主要与性能有关。

性能视图简介

Oracle 模式下包含一组基础视图，这些视图由数据库服务器维护，可供数据库管理员用户 SYS 访问。因为它们在打开和使用数据库时不断更新，并且其内容主要与性能有关，所以这些视图被称为动态性能视图。

数据库管理员可以查询和创建表上的视图，并将这些视图的访问权授予其他用户。

用户 SYS 拥有动态性能表，其名称以 `V$` 开头。在这些表上创建视图，然后使用前缀为 `V$` 的公共同义词。例如，V$DBLINK 视图包含有关 Database Link 的信息。

性能视图的种类

性能视图包括 `V$` 视图和 `GV$` 视图。

V$ 视图

实际的动态性能视图由前缀 `V$` 标识。这些视图的公共同义词具有前缀 `V$` 。数据库管理员和其他用户应仅访问 `V$` 对象。

实例启动后，可以访问从内存读取的 `V$` 视图。

GV$ 视图

每个 `V$` 视图几乎都有相应的 `GV$` 视图，即全局的 `V$` 视图。在 OceanBase 集群中，查询 `GV$` 视图将返回所有符合条件的 `V$` 视图信息。

性能视图的存储

动态性能视图是基于数据库内存结构而构建的虚拟表。动态性能视图也称为固定视图，数据库管理员无法更改或删除它们。

动态性能视图不是存储在数据库中的常规表，因为数据是动态更新的，所以无法保证视图的读取一致性。尽管这些视图似乎是常规数据库表，但这些视图仅提供有关内部磁盘结构和内存结构的数据。您可以从这些视图中进行选择，但不能对其进行更新或更改。

因为动态性能视图不是真正的表，所以数据内容取决于集群和服务结点的状态。

3.1.8. 数据完整性

3.1.8.1. 数据完整性概述

OceanBase 数据库 Oracle 模式下的租户可以使用完整性约束（Integrity Constraint）防止用户向数据库的表中插入非法数据。本章节主要介绍完整性约束类型和应用场景等。

完整性约束的作用是确保数据库内存储的信息遵从一定的业务规则。例如，如果 DML 语句的执行结果违反了完整性约束，将回滚语句并返回错误消息。

在视图和表的同义词（Synonym）执行的操作需要遵从基表（Base Table）上的完整性约束。

例如，用户在 `employees` 表的 `salary` 列上定义了完整性约束。此完整性约束规定 `salary` 列的数字值大于 10,000 的数据行不能插入 `employees` 表。如果某个 `INSERT` 或 `UPDATE` 语句违反了此完整性约束，将回滚语句并返回错误消息。

3.1.8.2. 完整性约束类型

3.1.8.2.1. 完整性约束类型概述

OceanBase 数据库支持多种类型的完整性约束。

用户可以使用以下完整性约束对输入的列值加以限制：

- `NOT NULL` 完整性约束
- `UNIQUE KEY` 完整性约束
- `PRIMARY KEY` 完整性约束
- 引用完整性约束
- `CHECK` 完整性约束

NOT NULL 完整性约束

非空规则（Null Rule）是定义在某一列上的规则，其作用是禁止将要被插入或更新的数据行的列值为空值 `NULL` 。

UNIQUE KEY 完整性约束

唯一值规则（Unique Value Rule）是定义在某一列（或某一列集）上的规则，其作用是确保将要被插入或更新的数据行此列（或列集）的值是唯一的。

PRIMARY KEY 完整性约束

主键值规则（Primary Key Value Rule）是定义在某一键（Key）（键指一列或一个列集）上的规则，其作用是确保表内的每一数据行都可以由某一个键值唯一地确定。

引用完整性约束

引用完整性规则（Referential Integrity Rule）是定义在某一键（Key）（键指一列或一个列集）上的规则，其作用是确保任意键值都能与相关表（Related Table）的某一键值，即引用值（Referenced Value）相匹配。

引用完整性约束时，对引用值可以进行哪些类型的数据操作（Data Manipulation），以及这些操作将如何影响依赖值（Dependent Value）。请参见如下具体规则：

- 限制（Restrict）：不允许对引用值进行更新与删除。
- 置空（Set to Null）：当因引用值被更新或删除后，所有受影响的依赖值都将被置为 `NULL`。
- 置默认值（Set to Default）：当引用值被更新或删除后，所有受影响的依赖值都将被赋予一个默认值。
- 级联操作（Cascade）：当引用值被更新后，所有受影响的依赖值也将被更新为相同的值。当引用数据行（Referenced Row）被删除后，所有受影响的依赖数据行（Dependent Row）也将被删除。
- 无操作（No Action）：不允许对引用值进行更新与删除。此规则与 `RESTRICT` 有所不同，它只在语句结束时进行检查，如果约束被延迟（Deferred）则在事务结束时进行检查。（Oracle 的默认操作为无操作。）

CHECK 完整性约束

一种相对复杂完整性检查（Complex Integrity Checking），是一种用户定义的规则，针对某一列（或某一列族）。其作用是依据用户自定义的表达式或函数对数据行的列值来进行计算，根据计算结果决定是否允许插入或更新此数据行。

3.1.8.2.2. NOT NULL 完整性约束

在 OceanBase 数据库中，NOT NULL 约束可用于限制一个列中不能包含 NULL 值。

在 OceanBase 数据库 V3.1.X 版本之前，`NOT NULL` 可以视作列的属性，仅能在建表时通过以下方式指定，并且建表后不支持将 `NULLABLE` 的列改为 `NOT NULL`。

```
CREATE TABLE employee(
   id NUMBER NOT NULL,
   name VARCHAR(100) NOT NULL,
   mgr_id NUMBER);
```

从 V3.1.X 版本开始，OceanBase 数据库真正地支持了 `NOT NULL` 约束，主要特性如下：

- 建表时可以指定约束名以及约束状态。用户如果没有为 `NOT NULL` 约束指定名字，数据库会利用系统当前时间自动生成约束名。状态包括 `RELY` / `NORELY`、`ENABLE` / `DISABLE` 和 `VALIDATE` / `NOVALIDATE`。建表时如果不为 `NOT NULL` 约束指定状态，默认为 `NORELY ENABLE VALIDATE`。
  - `RELY` 表示数据库可以将该列视作不含 `NULL` 值，以执行计划优化等。

- `ENABLE` 表示不能向该列中插入 `NULL` 值。
- `VALIDATE` 表示该列的现有数据中不包含 `NULL` 值。

```
CREATE TABLE employee(
   id NUMBER CONSTRAINT ID_NOT_NULL NOT NULL NORELY ENABLE VALIDATE,
   name VARCHAR(100) CONSTRAINT NAME_NOT_NULL NOT NULL,
   mgr_id NUMBER);
```

- 可以在系统视图 `ALL_CONSTRAINTS` 中查到 `NOT NULL` 约束，约束类型为 'C'，也可以通过 DBMS_METADATA.GET_DDL 获取 `NOT NULL` 约束的信息。

```
SELECT * FROM ALL_CONSTRAINTS WHERE TABLE_NAME = 'employee';
SELECT DBMS_METADATA.GET_DDL('CONSTRAINT', 'ID_NOT_NULL') FROM DUAL;
```

- 可以通过 `ALTER TABLE` 语句添加或删除 `NOT NULL` 约束，也支持修改 `NOT NULL` 约束的状态。如果希望向 `id` 列中插入 `NULL` 值，可以执行下面的语句将约束 `ID_NOT_NULL` 的状态改为 `DISABLE` 或直接删除约束。

```
ALTER TABLE employee MODIFY CONSTRAINT ID_NOT_NULL DISABLE;
ALTER TABLE employee DROP CONSTRAINT ID_NOT_NULL;
```

如果希望禁止向 `mgr_id` 列中插入 `NULL` 值，可以执行下面的语句在 `mgr_id` 列上添加一个 `NOT NULL` 约束。

```
ALTER TABLE employee MODIFY mgr_id NOT NULL;
```

执行上述语句修改 `mgr_id` 列为 `not null` 时，会检查已有数据是否含 `NULL` 值，如果包含 `NULL` 值会报错。

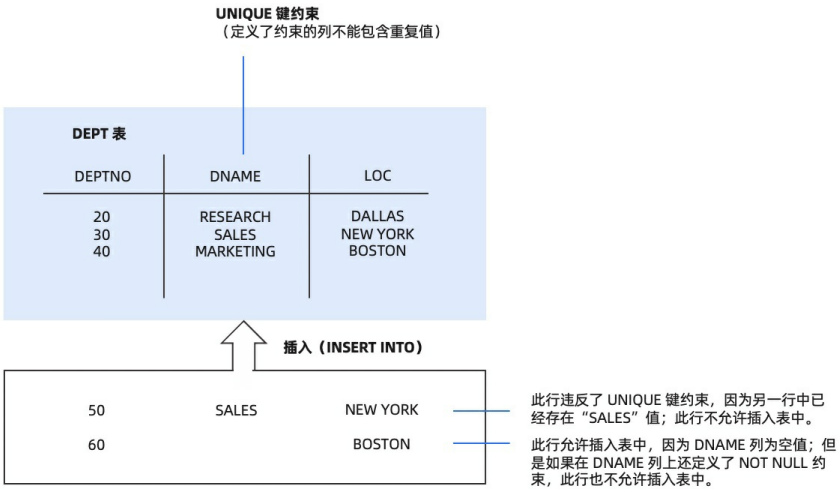
3.1.8.2.3. 唯一性约束

唯一键（UNIQUE key）约束是关于唯一列值（Unique Column Value）的规则，定义在某一列或某一列族上，其作用是确保将要被插入或更新的数据行的列或列族的值是唯一的，表的任意两行的某列或某个列集的值不重复。

本文通过以下具体示例来介绍唯一性约束的主要特性。

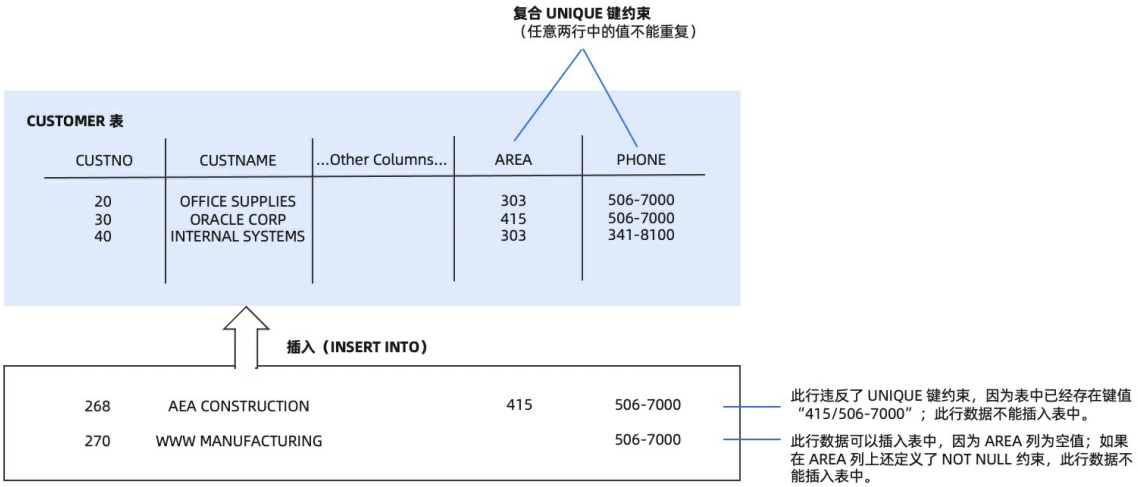
```
obclient> CREATE TABLE t1(c1 INT, c2 INT, CONSTRAINT t1_uk_c1_c2 UNIQUE(c1, c2));
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(1, 1);
ORA-00001: unique constraint '1-1' for key 'T1_UK_C1_C2' violated

obclient> INSERT INTO t1 VALUES(null, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(null, 1);
ORA-00001: unique constraint 'NULL-1' for key 'T1_UK_C1_C2' violated

obclient> INSERT INTO t1 VALUES(null, null);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(null, null);
Query OK, 1 row affected
```

又如，在下图中，`dept` 表的 `dname` 列上定义了 `UNIQUE` 键约束，则不允许此表存在重复的部门名称，但是允许用户向 `dname` 列中插入空值。如果 `dname` 列上还定义了 `NOT NULL` 约束，则不允许用户向 `dname` 列中插入空值。



如果 `UNIQUE` 键由多列构成，那么这组数据列被称为复合唯一键（Composite Unique Key）。如下图所示，`customer` 表上定义的 `UNIQUE` 键约束使用了复合唯一键，包含 `area` 和 `phone` 两列，这要求了任意两行中键的值不能重复。用户可以向列中输入空值。但是，如果列上还定义了 `NOT NULL` 约束，则不允许输入空值。

用户可以向 `customer` 表插入任意条记录，但依据上述 `UNIQUE` 键约束的限制，表中各行的区码（Area Code）与电话号码（Telephone Number）的组合不能重复。这能避免因疏忽造成电话号码重复问题。



3.1.8.2.4. 主键约束

主键值规则（Primary Key Value Rule）是定义在某一键 Key（键指一列或一个列集）上的规则，其作用是确保表内的每一数据行都可以由某一个键值唯一地确定。

每个数据库表上最多只能定义一个 `PRIMARY KEY` 约束。构成此约束的列（一列或多列）的值可以作为一行数据的唯一标识符，即每个数据行可以由此主键值命名。

> **说明**
>
> OceanBase 数据库只支持在建表时通过 `CREATE TABLE` 创建主键约束，暂不支持通过 `ALTER TABLE` 追加、删除、修改主键约束。

本文通过以下具体示例来介绍主键约束的主要特性。

```
obclient> CREATE TABLE t1(c1 INT, c2 INT, CONSTRAINT pk_c1_c2 PRIMARY KEY(c1, c2));
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(1, 1);
ORA-00001: unique constraint '1-1' for key 'PK_C1_C2' violated

obclient> INSERT INTO t1 VALUES(null, 1);
ORA-01400: cannot insert NULL into '(C1)'

obclient> INSERT INTO t1 VALUES(null, null);
ORA-01400: cannot insert NULL into '(C1)'
```

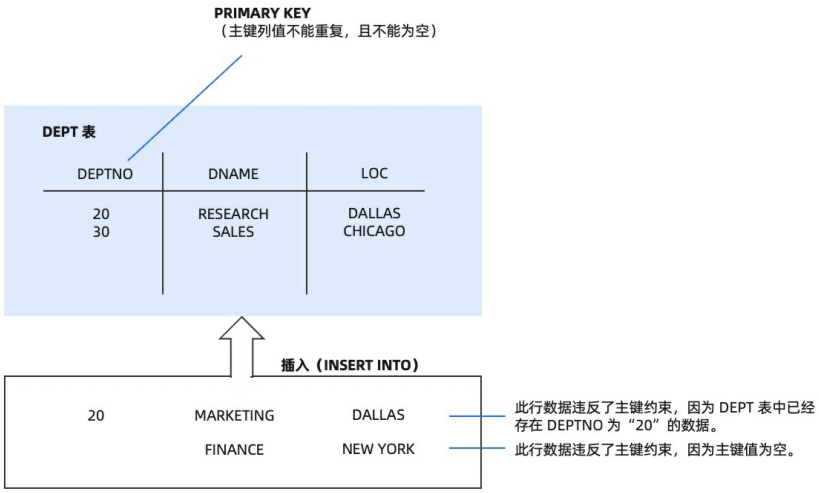
`PRIMARY KEY` 完整性约束（Integrity Constraint）能够确保表的数据遵从以下两个规则：

- 任意两行数据的 `PRIMARY KEY` 约束列（一列或多列）不存在重复值。
- 主键列的值不为空。即用户必须为主键列输入值。

数据库不强制用户为表定义主键，但使用主键可以使表内的每行数据可以被唯一确定，且不存在重复的数据行。

下图展示了定义在 `dept` 表上的 `PRIMARY KEY` 约束，以及违反此约束的数据行。`dept` 表内包含 3 列，分别是 `deptno` 、 `dname` 和 `loc` 。在 `deptno` 列上定义了主键约束，则此列不能有重复数据，且不能为空。

图中还展示了两行因为违反主键约束而无法插入 `dept` 表的数据，一行数据的主键值与已有数据重复，另一行数据的主键列为空值。



3.1.8.2.5. 外键约束

每当两个表包含一个或多个公共列时，OceanBase 数据库 Oracle 模式下可以通过外键约束（也称为参照完整性约束）强制执行两个表之间的关系。

外键约束的特性

外键约束要求，对于定义约束的列中的每个值，与另一个指定的表和列中的值必须匹配。语法格式如下：

```
REFERENCES [ schema. ] object [ (column_name [, column_name...]) ] [ON DELETE { CASCADE | SET NULL } ]
```

参照完整性约束相关的术语如下表所示。

术语	定义

术语	定义
外键（Foreign key）	引用 Referenced Key 的约束定义中包含的列或列集。 外键可以定义为多列。但是，复合外键必须引用具有相同列数和相同数据类型的复合主键或唯一键。 外键的值可以匹配引用的主键值或唯一键值，也可以为空。如果复合外键的任何列为空，则键的非空部分不必与父键的任何相应部分匹配。
引用键（Referenced Key）	外键引用的表的唯一键或主键。
子表（Dependent/Child Table）	包含外键的表。此表取决于引用的唯一键或主键中存在的值。
父表（Referenced/Parent table）	子表的外键所引用的表。正是这个表的引用键决定了在子表中是否允许指定的插入或更新。

示例：创建包含外键约束的表。

```
obclient> CREATE TABLE supplier_groups(
     group_id NUMBER GENERATED BY DEFAULT AS IDENTITY,
     group_name VARCHAR2(127) NOT NULL,
     PRIMARY KEY (group_id)
      );
Query OK, 0 rows affected

obclient> CREATE TABLE suppliers (
     supplier_id NUMBER GENERATED BY DEFAULT AS IDENTITY,
     supplier_name VARCHAR2(127) NOT NULL,
     group_id NUMBER NOT NULL,
     PRIMARY KEY(supplier_id),
     CONSTRAINT fk_name FOREIGN KEY(group_id) REFERENCES supplier_groups(group_id)
     );
Query OK, 0 rows affected
```

外键约束的主要特性如下：

- 自引用完整性约束

  自引用完整性约束是引用同一表中父键的外键。建表示例如下：

  ```
  obclient> CREATE TABLE employee (
     employee_id INT PRIMARY KEY,
     employee_name VARCHAR(30),
     salary VARCHAR(30),
     manager_id INT,
     CONSTRAINT sr_fk_emp_man FOREIGN KEY (manager_id) REFERENCES employee(employee_id)
     );
  ```

- **空值和外键**

  关系模型允许外键的值匹配引用的主键或唯一键值，或者为空。如果复合外键的任何列为空，则键的非空部分不必与父键的任何相应部分匹配。

- **父键修改和外键**

  外键和父键之间的关系对删除父键有影响。修改父键时，参照完整性约束，可以指定要对子表中的受影响的行执行 `DELETE NO ACTION` 或者 `DELETE CASCADE` 操作。

  下表概述了对父表中的键值和子表中的外键值的不同引用操作所允许的 DML 语句。

DML 语句	针对父表的操作	针对子表的操作
INSERT	父表键唯一则允许写入	当外键值存在于父键中，或者部分（或全部）为空时才允许写入。
DELETE NO ACTION	如果子表中没有匹配的记录，则允许对父表对应键进行删除操作。	无限制条件
DELETE CASCADE	无限制条件	无限制条件

- **索引和外键**

  通常来说，外键列应该是索引列。唯一的例外是匹配的唯一键或主键从未更新或删除。索引子表中的外键有以下好处：

  - 防止子表上的全表锁定。数据库只需要获取索引上的行锁。
  - 无需对子表进行全表扫描。

> **注意**
>
> OceanBase 数据库 Oracle 模式下外键暂不支持如下行为：
>
> - 不支持 `SET NULL` 行为。
> - 不支持语句级检测。
> - 不支持延迟约束检查。

外键约束的常用操作

- **将外键约束添加到表中。**

```
ALTER TABLE child ADD CONSTRAINT fk_name_1
     FOREIGN KEY (col1) REFERENCES parent (col1);
```

- 删除外键约束。

```
ALTER TABLE child DROP CONSTRAINT fk_name_1;
```

- 禁用外键约束。

```
ALTER TABLE child DISABLE CONSTRAINT fk_name_1;
```

- 启用外部约束。

```
ALTER TABLE child ENABLE CONSTRAINT fk_name_1;
```

3.1.8.2.6. CHECK 约束

CHECK 完整性约束（Integrity Constraint）定义于列或列集上，此约束要求数据行满足用户定义的检查条件或条件判断结果为不确定（Unknown）。如果一个 DML 语句使 CHECK 完整性约束的检查结果为假（False），则此语句将被回滚（Rolled Back）并报错。

以下示例展示了 CHECK 约束语法的主要特性。

```
obclient> CREATE TABLE t1(c1 INT, c2 INT, CONSTRAINT t1_cst_c1_equals_c2 CHECK(c1 = c2));
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(2, 3);
ORA-02290: check constraint violated
```

用户可以使用 `CHECK` 约束定义检查条件（Check Condition）来实现特殊的完整性规则（Integrity Rule）。定义 `CHECK` 约束的检查条件时有以下限制：

- 检查条件必须是用于评估被插入或被更新数据行内的值的布尔表达式（Boolean Expression）。
- 检查条件中不能包含：子查询（Subquery）、序列（Sequence）、ROWNUM 等伪列（Pseudo column），以及 SYSDATE()、UID()、USER() 、USERENV() 等在不同环境或不同场景下执行可能得到不同计算结果的系统函数。

同一列可以被多个 `CHECK` 约束的条件定义所引用，即用户为某一列定义的 `CHECK` 约束的数量不受限制。如果用户为在一列上创建了多个 `CHECK` 约束，必须确保各个约束的检查条件不会相互冲突，`CHECK` 约束的检查顺序是不确定的，也不会检查各个 `CHECK` 约束是否为互斥的（Mutually Exclusive）。

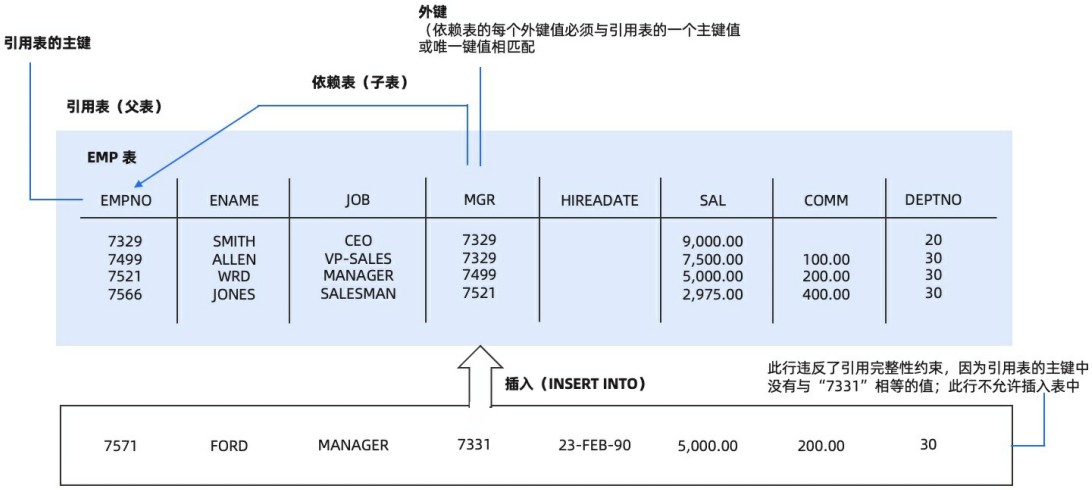
3.1.8.3. 完整性约束的使用

本节主要介绍完整性约束的检查时机和约束选项。

约束检查时机

OceanBase 数据库何时执行约束检查（Checking of Constraint），有助于明确存在各种约束时允许执行的操作类型。本文通过具体示例来介绍约束检查的时机。

如下图所示，定义 `emp` 表。在 `emp` 表上定义自引用约束（Self-Referential Constraint），`mgr` 列的值依赖于 `empno` 列的值。为了简化示例，以下内容只针对 `emp` 表的 `empno`（`employee_id`）以及 `mgr`（`manager_id`）列。



现在向 `emp` 表插入第一条数据。由于此时表内没有数据，`mgr` 列无法引用 `empno` 列已有的值，可以根据如下场景执行数据插入：

- 如果 `mgr` 列上没有定义 `NOT NULL` 约束，可以在第一行的 `mgr` 列上输入一个空值。由于外键约束允许空值，所以此行能成功插入表中。
- 可以向第一行的 `empno` 及 `mgr` 列输入一个相同的值。此种情况说明是在语句运行完成后执行的约束检查（Constraint Checking）。如果在第一行的父键（Parent Key）及外键（Foreign Key）插入相同的值，必须首先运行语句（即插入数据行），再检查此行数据的 `mgr` 列值是否能与此表内的某个 `empno` 列值相匹配。
- 执行一个多行的 `INSERT` 语句，例如与 `SELECT` 语句结合的 `INSERT` 语句，将插入存在相互引用关系的多行数据。例如，第一行的 `empno` 列值为 200，`mgr` 列值为 300，而第二行的 `empno` 列值为 300，`mgr` 列值为 200。

此种情况也说明数据库会将约束检查延迟直至语句运行结束。所有数据行首先被插入，之后逐行检查是否存在违反约束的情况。

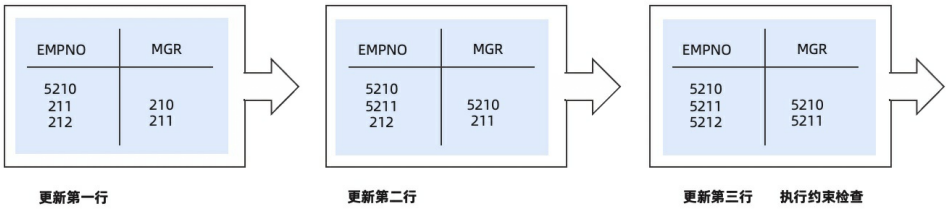
用上述的自引用约束再举一个例子。假设公司被收购，所有员工编号需要被更新为当前值加 5000，以便和新公司的员工编号保持一致。由于经理编号也是员工编号，所以此值也需要加 5000。下图为被更新之前的 `emp` 表，其中包含 `empno` 与 `mgr` 两列。`empno` 列有 3 个值：210、211 和 212。`mgr` 列有两个值：210 和 211。

EMPNO	MGR
210	
211	210
212	211

对 `emp` 表执行以下 SQL:

```
UPDATE EMP
SET empno = empno + 5000,
mgr = mgr + 5000;
```

尽管 `emp` 表上定义的约束要求每个 `mgr` 值必须能和一个 `empno` 值相匹配，此语句仍旧可以执行，因为在语句执行后才进行约束检查。下图表明执行了 SQL 语句的全部操作之后才进行约束检查。



首先为每个员工编号加 5000，再为每个经理编号加 5000。在第一步中，`empno` 列的值 210 被更新为 5210。在第二步中，`empno` 列的值 211 被更新为 5211，`mgr` 列的值 210 被更新为 5210。在第三步中，`empno` 列的值 212 被更新为 5212，`mgr` 列的值 211 被更新为 5211。最后执行约束检查。

上述示例说明了 `INSERT` 及 `UPDATE` 语句的约束检查机制。事实上各类 DML 语句的约束检查机制均相同，这些 DML 语句包括 `UPDATE` 、`INSERT` 及 `DELETE` 等语句。

约束选项

OceanBase 数据库 Oracle 模式下的外键约束、`CHECK` 约束、`NOT NULL` 约束均支持 `ENABLE` / `DISABLE` 和 `VALIDATE` / `NOVALIDATE` 两种选项。`ENABLE` / `DISABLE` 决定是否对表中通过 DML 插入或更新后的新数据进行是否满足约束条件的检查，`VALIDATE` / `NOVALIDATE` 表示表中已有的数据是否都已经满足约束条件。

约束选项的默认值

创建约束时，`ENABLE` / `DISABLE` 选项默认是 `ENABLE` 。

如果 `ENABLE` / `DISABLE` 选项为 `ENABLE` ，则默认的 `VALIDATE` / `NOVALIDATE` 选项为 `VALIDATE` 。

如果 `ENABLE` / `DISABLE` 选项为 `DISABLE` ，则默认的 `VALIDATE` / `NOVALIDATE` 选项为 `NOVALIDATE` 。

约束选项组合的效果

- `ENABLE VALIDATE` 会检查新数据的合法性，也会检查表上已有数据的合法性。如果已有数据不满足约束，则不允许修改约束状态为 `ENABLE VALIDATE` 或创建状态为 `ENABLE VALIDATE` 的约束。

- `ENABLE NOVALIDATE` 不会检查已有数据的合法性，只检查表中新数据的合法性。
- `DISABLE VALIDATE` 会使整张表不被允许执行 DML 操作。
- `DISABLE NOVALIDATE` 指的是修改约束为无效约束，不会检查已有数据的合法性，也不检查表中新数据的合法性。

约束选项举例

```
obclient> CREATE TABLE t1(c1 INT, c2 INT);
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(0, 1);
Query OK, 1 row affected

obclient> ALTER TABLE t1 ADD CONSTRAINT cst CHECK(c1 = c2) ENABLE VALIDATE;
ORA-02293: cannot validate (TEST.CST) - check constraint violated

obclient> ALTER TABLE t1 ADD CONSTRAINT cst CHECK(c1 = c2) DISABLE VALIDATE;
ORA-02293: cannot validate (TEST.CST) - check constraint violated

obclient> ALTER TABLE t1 ADD CONSTRAINT cst CHECK(c1 = c2) ENABLE NOVALIDATE;
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(0, 1);
ORA-02290: check constraint violated

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected

obclient> ALTER TABLE t1 MODIFY CONSTRAINT cst DISABLE NOVALIDATE;
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(0, 1);
Query OK, 1 row affected

obclient> DELETE FROM t1 WHERE c1 != c2;
Query OK, 2 rows affected

obclient> ALTER TABLE t1 MODIFY CONSTRAINT cst DISABLE VALIDATE;
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(1, 1);
ORA-25128: No insert/update/delete on table with constraint (TEST.CST) disabled and validat
ed

obclient> ALTER TABLE t1 MODIFY CONSTRAINT cst ENABLE VALIDATE;
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(0, 1);
ORA-02290: check constraint violated

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected
```

3.2. MySQL 模式

3.2.1. 数据库对象介绍

3.2.1.1. 数据库对象概述

数据库对象是数据库的组成部分，是指在数据库中可以通过 SQL 进行操作和使用的对象。本章节主要介绍 OceanBase 数据库 MySQL 模式下所支持的数据库对象类型、存储方式和数据库对象之间的依赖。

OceanBase 数据库 MySQL 模式下的数据库对象主要包括：表（Table）、视图（View）、索引（Index）、分区（Partition）、序列（Sequence）、触发器（Trigger）以及存储程序等。

MySQL 模式下的用户（User）在被赋予相关权限后可以连接数据库、访问或操作数据库对象。库（Database）是数据库对象的集合，用于权限管理和命名空间隔离。Schema 是 Database 的同义词，SQL 中可以使用 Schema 关键字代替 Database 关键字，例如使用 `CREATE SCHEMA` 代替 `CREATE DATABASE` 等。

3.2.1.2. 数据库对象类型

本节主要介绍 OceanBase 数据库 MySQL 模式下所支持的数据库对象类型。

有关 OceanBase 数据库 MySQL 模式下的数据库对象的详细信息如下表所示。

对象类型	描述
表（Table）	数据库里最基础的存储单元，表里的数据由行和列组织排列的。
视图（View）	视图是一个虚拟的表，其内容由查询定义。 同真实的表一样，视图包含一系列带有名称的列和行数据。但是，视图并不在数据库中以存储的数据值集合的形式存在。行和列数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。
索引（Index）	索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的指定信息。 例如，想按指定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。
分区（Partition）	OceanBase 数据库可以把普通的表的数据按照一定的规则划分到不同的区块内，同一区块的数据物理上存储在一起。这种划分区块的表叫做分区表。 与 Oracle 中的 Partition 概念相同，在 OceanBase 数据库中只有水平分区，表的每一个分区包含一部分记录。根据行数据到分区的映射关系不同，分为 Hash 分区、Range 分区（按范围）和 List 分区等。 每一个分区，还可以用不同的维度再分为若干分区，叫做二级分区。例如，交易记录表按照用户 ID 分为若干 Hash 分区，每个一级 Hash 分区再按照交易时间分为若干二级 Range 分区。

对象类型	描述
触发器（Trigger）	触发器是由系统在指定事件发生时自动调用的一组存储过程。 与普通存储过程不同的是，触发器可以被启用或禁用，不能被显式调用。
存储程序（Stored Program）	存储程序（Stored Program）是一组为了完成指定功能的 SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并指定参数（如果该存储过程带有参数）来调用执行它。 存储程序是可编程的函数，在数据库中创建并保存，由 SQL 语句和控制结构组成。

3.2.1.3. 数据库对象存储

OceanBase 数据库将一些数据库对象的数据存储在一个或者多个 SSTable 中，一个 SSTable 包含一个或多个宏块，每个宏块中包含一个或多个微块，每个微块包含一行或者多行。有些数据库对象，例如视图、虚拟表等，由于只有元数据信息，所以不会存储实际的数据。

3.2.1.4. 数据库对象之间的依赖

数据库对象之间的依赖就是描述数据库对象之间的依赖和引用关系。本文主要介绍 OceanBase 数据库 MySQL 模式下如何创建和查看数据库对象之间的依赖关系。

某些类型的 Schema 对象可以在其定义中引用其他的对象。例如，一个视图的定义可能是一个引用了表或者其他视图的查询。如果对象 A 的定义引用对象 B，则 A 是（B 的）依赖对象，B 是（A 的）引用对象。

MySQL 租户下描述了视图对视图、视图对表的依赖关系，并可以通过 information_schema 数据库的 VIEW_TABLE_USAGE 视图查询。

示例 1：创建一个视图 `view2`，该视图引用了表 `tbl1` 和另一个视图 `view1`。

```
obclient> CREATE TABLE tbl1(col1 INT, col2 INT);
Query OK, 0 rows affected

obclient> CREATE TABLE tbl2 (col1 INT, col2 INT);
Query OK, 0 rows affected

obclient> CREATE VIEW view1 AS SELECT * FROM tbl2;
Query OK, 0 rows affected

obclient> CREATE VIEW view2 AS SELECT t.col1 AS col1, t.col2 AS col2 FROM tbl1 t, view1 v W
HERE
      t.col1 = v.col2;
Query OK, 0 rows affected
```

示例 2：通过 information_schema 数据库的 VIEW_TABLE_USAGE 视图查询数据库对象之间的依赖关系。

```
obclient> SELECT * FROM information_schema.VIEW_TABLE_USAGE;
+--------------+-------------+-----------+--------------+------------+---------------+
| VIEW_CATALOG | VIEW_SCHEMA | VIEW_NAME | TABLE_SCHEMA | TABLE_NAME | TABLE_CATALOG |
+--------------+-------------+-----------+--------------+------------+---------------+
| def          | test        | view1     | test         | tbl2       | def           |
| def          | test        | view2     | test         | view1      | def           |
|
| def          | test        | view2     | test         | tbl1       | def           |
+--------------+-------------+-----------+--------------+------------+---------------+
3 rows in set
```

上述示例中，视图的查询结果中描述了数据库对象之间的依赖关系。例如，视图 `view1` 引用了表 `tbl2` ，视图 `view2` 引用了表 `tbl1` 和另一个视图 `view1` 。

视图对视图、视图对表的依赖关系在某些场景下非常有用。例如，在数据库迁移的场景下，需要在另一个数据库中重建 Schema 对象。如果需要重建的 Schema 对象是一个视图，那就必须要先重建该视图依赖的其他对象，否则重建该视图时会报错依赖的对象不存在。如果能获取到视图对视图、视图对表的依赖关系，那就可以很轻松的完成重建对象的工作。

3.2.2. 表

3.2.2.1. 表概述

数据库表是一系列二维数组的集合，用来代表和储存数据对象之间的关系。本章节主要介绍表所涉及的数据库对象以及表的相关使用特性，包括表存储、表压缩、表组和临时表等。

在数据库中，表是数据组织的基础单元，它由纵向的列和横向的行组成。例如，一个有关作者信息的表格名称为 “作者”，表中每个列包含的是所有作者的某个特定类型的信息，例如姓氏，而每行则包含了某个指定作者的所有信息，例如姓、名、住址等等。对于指定的数据库表，列的数目一般在创建表时确定，各列之间可以由列名来识别；而行的数目可以随时动态变化。

表分为普通表和临时表。OceanBase 数据库 MySQL 模式下的临时表生命周期仅存在于 Session 期间，Session 断开时创建的所有临时表自动被 `DROP` ，如果 `sess#1` 创建了临时表则对 `sess#2` 不可见。

不同 Session 可以创建同名临时表，临时表可以和现有的普通名重名，但是不能基于临时表创建视图。当同时存在同名的临时表和普通表时， `SHOW CREATET TABLE` 、 `DESC` 、DML 等操作都执行于临时表，此时普通表被隐藏。

列

在数据库中，列（Column）用于记录一张表上某个属性的字段的值，用户给每个属性起的名称即为列名。除了列名以外，列上还有数据类型以及数据类型的最大长度（精度）等信息。

除了普通列以外，OceanBase 数据库 Oracle 模式下还包括虚拟列（Virtual Column/Generated Column）和自增列（Auto_increment Column）。

虚拟列不像普通列一样具有真实的物理存储空间，而是在查询时通过用户在虚拟列上定义的一个表达式或函数来计算得到结果的。

自增列满足以下三个条件：

- 多分区全局唯一
- 语句内连续递增
- 生成的自增列值大于用户显式插入的值

行

在数据库中，行（Row）表示表中单条记录中所有列的数据集合。简单来说，数据库可以认为是由列和行组成的。表中的每一行代表一组相关数据，并且表中的每一行具有相同的结构。

例如，一张与公司信息相关的数据表，每一行代表一个公司，对应的列可能代表诸如公司名称、公司街道地址、增值税号等内容。

3.2.2.2. 数据类型

3.2.2.2.1. 数据类型概述

在 OceanBase 数据库中，表中的每一列都有一个数据类型，每个数据类型都对应独特的存储格式、约束以及取值范围。

OceanBase 数据库支持的 SQL 数据类型包含以下三种：

- 数值类型
- 日期时间类型
- 字符类型

> **说明**
>
> 目前暂不支持空间类型和 JSON 类型。

3.2.2.2.2. 数值类型

OceanBase 数据库支持标准 SQL 数值类型，包括精确数值类型、近似数值类型、存储位值的 BIT 数据类型和扩展类型。

数值类型概述

数值类型分类

OceanBase 数据库当前版本支持的数值类型可以划分为如下四类：

- 整数类型：`BOOL/BOOLEAN/TINYINT`、`SMALLINT`、`MEDIUMINT`、`INT/INTEGER` 和 `BIGINT`。
- 定点类型：`DECIMAL` 和 `NUMERIC`。
- 浮点类型：`FLOAT` 和 `DOUBLE`。
- Bit-Value 类型：`BIT`。

数值类型在定义时可以指定 Precision（精度，即字段长度）和 Scale（范围，即小数位数），不同数值类型的 Precision 和 Scale 的含义可能有所不同，详情请参见各类型的说明。

ZEROFILL 属性

数值类型在定义时可以通过 `ZEROFILL` 关键字指定最小显示宽度，同时将该类型隐式定义为 `UNSIGNED` 。在数据实际显示宽度不足最小显示宽度时，通过先将小数部分补零到 Scale 上限，然后将整数部分补零到 Precision 上限的方式，将显示宽度补足到最小显示宽度。

示例如下：

- `INT(5) ZEROFILL` 表示当数据值为 123 时，将显示为 00123。
- `DECIMAL(10, 5) ZEROFILL` 表示当数据值为 123.456 时，将显示为 00123.45600。

整数类型

整数类型为定长、精确的数值类型，值域取决于类型长度以及是否为无符号，Precision 只表示最小显示宽度。

OceanBase 数据库所支持的每种整数类型所需的存储长度和值域如下表所示。

类型	长度（字节）	值域（有符号）	值域（无符号）
`BOOL` / `BOOLEAN` / `TINYINT`	1	$[-2^7, 2^7 - 1]$	$[0, 2^8 - 1]$
`SMALLINT`	2	$[-2^{15}, 2^{15} - 1]$	$[0, 2^{16} - 1]$
`MEDIUMINT`	3	$[-2^{23}, 2^{23} - 1]$	$[0, 2^{24} - 1]$
`INT` / `INTEGER`	4	$[-2^{31}, 2^{31} - 1]$	$[0, 2^{32} - 1]$
`BIGINT`	8	$[-2^{63}, 2^{63} - 1]$	$[0, 2^{64} - 1]$

TINYINT

`TINYINT` 用于表示一个非常小的整数。语法如下：

```
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
```

其中， `M` 表示最大显示宽度。最大显示宽度为 255。显示宽度与可以存储的值范围无关。如果为数值列指定 `ZEROFILL` ，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。

BOOL/BOOLEAN

`BOOL/BOOLEAN` 类型是 `TINYINT` 的同义词。零值表示错误，非零值表示正确。示例如下：

```
obclient> SELECT IF(0, 'true', 'false');
+------------------------+
| IF(0, 'true', 'false') |
+------------------------+
| false                  |
+------------------------+
1 row in set

obclient> SELECT IF(1, 'true', 'false');
+------------------------+
| IF(1, 'true', 'false') |
+------------------------+
| true                   |
+------------------------+
1 row in set

obclient> SELECT IF(2, 'true', 'false');
+------------------------+
| IF(2, 'true', 'false') |
+------------------------+
| true                   |
+------------------------+
1 row in set

obclient> SELECT IF(2 = FALSE, 'true', 'false');
+--------------------------------+
| IF(2 = FALSE, 'true', 'false') |
+--------------------------------+
| false                          |
+--------------------------------+
1 row in set
```

SMALLINT

`SMALLINT` 用于表示一个小值整数。语法如下：

```
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
```

其中，`M` 表示最大显示宽度。最大显示宽度为 255。显示宽度与可以存储的值范围无关。如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。

MEDIUMINT

`MEDIUMINT` 用于表示一个中等大小的整数。语法如下：

```
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
```

其中，`M` 表示最大显示宽度。最大显示宽度为 255。显示宽度与可以存储的值范围无关。如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。

INT/INTEGER

`INT` 或 `INTEGER` 用于表示一个正常大小的整数。语法如下：

```
INT[(M)] [UNSIGNED] [ZEROFILL] INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

其中，`M` 表示最大显示宽度。最大显示宽度为 255。显示宽度与可以存储的值范围无关。如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。

BIGINT

BIGINT 用于表示一个大整数。语法如下：

```
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
```

其中，`M` 表示最大显示宽度。最大显示宽度为 255。显示宽度与可以存储的值范围无关。如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。

关于 `BIGINT`，需要注意以下事项：

- 所有运算需要使用有符号的 `BIGINT` 或 `DOUBLE` 值，因此不应使用大于 9223372036854775807（63 位）的无符号大整数，BIT 函数除外。否则，在将 `BIGINT` 值转换为 `DOUBLE` 时会出现舍入错误，导致结果中的最后一位数字可能出错。
- 通过使用字符串存储 `BIGINT`，这样始终可以将精确的整数值存储在 `BIGINT` 列中。在这种情况下，在执行字符串到数字的转换时，不涉及双精度的中间转换。
- 当两个操作数都是整数值时，-、+ 和 * 运算符使用 `BIGINT` 运算。如果将两个大整数（或函数返回的整数结果）相乘，当结果大于 9223372036854775807 时可能会得到异常的结果。

定点类型

定点类型为变长、精确数值类型，值域和精度取决于 Precision 和 Scale，以及是否为无符号。

`DECIMAL` 等价于 `NUMERIC`。语法如下：

```
DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
```

其中，`M` 是可以存储的总位数（Precision），`D` 是小数点后的位数（Scale）。小数点和负数符号“-”不计入 `M`。如果 `D` 为 0，则值没有小数点或小数部分。整数部分最大有效位数等于 `M` 减去 `D` 的值，即 Precision 减去 Scale 的值。

`DECIMAL` 的 `M` 最大值为 65，`D` 最大值为 30。如果省略 `D`，则默认为 0。如果省略 `M`，则默认为 10。当 `M` 和 `D` 都未指定时，所有带有 `DECIMAL` 列的基本计算（+、-、*、/）都使用 65 位的精度。

如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。如果指定 `UNSIGNED`，则不允许为负值。

例如 `DECIMAL(5, 2)` 表示整数部分和小数部分最大有效位数分别为 3 和 2，所以值域为 [-999.99, 999.99]。如果同时定义为 `UNSIGNED`，则值域为 [0, 999.99]。

如下类型也是 `DECIMAL` 的同义词。其中，`FIXED` 可用于与其他数据库系统兼容。

```
DEC[(M[,D])] [UNSIGNED] [ZEROFILL], NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL],
 FIXED[(M[,D])] [UNSIGNED] [ZEROFILL]
```

浮点类型

浮点类型为定长、非精确数值类型，值域和精度取决于类型长度、Precision 和 Scale，以及是否为无符号。

Precision 和 Scale 分别表示十进制下的总最大有效位数、小数部分最大有效位数，整数部分最大有效位数等于 Precision 减去 Scale 的值，其中 Precision 最大值为 53，Scale 的最大值为 30。

> **注意**
>
> 浮点类型的精度只是 IEEE 标准中规定的理论值，实际情况可能因硬件或操作系统限制略有不同。

下表为不指定 Precision 和 Scale 时浮点类型所需的存储长度和值域。

类型	长度（字节）	值域	精度
FLOAT	4	[-3.402823466E+38, -1.175494351E-38]、0 和 [1.175494351E-38,3.402823466E+38]	7 位

类型	长度（字节）	值域	精度
DOUBLE	8	[-1.7976931348623157E+308, -2.2250738585072014E-308]、0 和 [2.2250738585072014E-308,1.7976931348623157E+308]	15 位

如果指定 Precision 和 Scale，则值域确定方法与定点类型相同。

FLOAT

FLOAT 用于表示一个小的（单精度）浮点数。语法如下：

```
FLOAT [UNSIGNED] [ZEROFILL]
```

`M` 是可以存储的总位数，`D` 是小数点后的位数。如果省略 `M` 和 `D`，则将值存储到硬件允许的限制范围内。单精度浮点数精确到大约 7 个小数位。

如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。如果指定 `UNSIGNED`，则不允许为负值。

> **说明**
>
> - `FLOAT(M,D)` 是 MySQL 的过时语法，不建议用户使用。
> - OceanBase 数据库中，`D` 的值只支持 0，当设置为非 0 时，会报错“Feature not supportted”。

`FLOAT` 还支持以下语法：

```
FLOAT(p) [UNSIGNED] [ZEROFILL]
```

其中，`p` 表示以位为单位的精度，但仅使用此值来确定结果数据类型为 `FLOAT` 或是 `DOUBLE`。如果 `p` 为 0 到 24，则数据类型变为 `FLOAT`，没有 `M` 或 `D` 值；如果 `p` 为 25 到 53，则数据类型变为 `DOUBLE`，没有 `M` 或 `D` 值。结果列的范围与本节前面描述的单精度 `FLOAT` 或双精度 `DOUBLE` 数据类型相同。

DOUBLE

`DOUBLE` 用于表示正常大小（双精度）浮点数。语法如下：

```
DOUBLE [UNSIGNED] [ZEROFILL]
```

其中，`M` 是可以存储总位数，`D` 是小数点后的位数。如果省略 `M` 和 `D`，则将值存储到硬件允许的限制范围内。双精度浮点数精确到大约 15 位小数。

如果为数值列指定 `ZEROFILL`，OceanBase 数据库会自动向该列添加 `UNSIGNED` 属性。如果指定 `UNSIGNED`，则不允许为负值。

> 说明
>
> - `DOUBLE[(M,D)]` 是 MySQL 的过时语法，不建议用户使用。如果用户需要精确查找，建议使用 `DECIMAL` 类型。
> - OceanBase 数据库中，`D` 的值只支持 0，当设置为非 0 时，会报错“Feature not supportted”。

DOUBLE PRECISION

`DOUBLE PRECISION` 是 `DOUBLE` 的同义词。语法如下：

```
DOUBLE PRECISION [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL]
```

> 说明
>
> `DOUBLE PRECISION [(M,D)]` 是 MySQL 的过时语法，不建议用户使用。如果用户需要精确查找，建议使用 `DECIMAL` 类型。

BIT-Value 类型

`BIT` 数据类型用于存储位值。

位值通过 `b'value'` 的形式指定，`value` 是用 0 和 1 来指定的，例如，`b'111'` 表示 7，`b'10000000'` 表示 128。

语法如下：

```
BIT[(M)]
```

`M` 表示每个值的位数，范围为 [1，64]。如果省略 `M`，则默认为 1。

当向 `BIT(M)` 列插入值时，如果插入值的长度小于 `M`，则会在左侧填充 0。例如：将 `b'101'` 插入到 `BIT(6)` 时，相当于插入了 `b'000101'`。

3.2.2.2.3. 日期时间数据类型

OceanBase 数据库当前版本所支持的日期时间相关的数据类型包括 DATE、TIME、DATETIME、TIMESTAMP 和 YEAR，每个类型的值拥有一个合法范围。

使用日期时间类型时，需要考虑以下几点：

- OceanBase 数据库以标准的格式输出日期时间类型，但是会尝试以多种格式解析用户输入的值。
- 尽管 OceanBase 数据库尝试以多种格式解析用户输入的字面量，日期部分必须符合“年-月-日”的顺序，例如 '98-09-04'，不能是“月-日-年”或者“日-月-年”的顺序，例如 '09-04-98' 和 '04-09-98'。如果希望转换不满足“年-月-日”顺序的字符串，可以使用 STR_TO_DAT 函数。
- 如果输入的年份是两位数是具有歧义的，因为不确定是哪个世纪。OceanBase 数据库转换一位或两位年份的原则是，“70-99”之间的年份转换为“1970-1999”，“00-69”之间的年份转换为“2000-2069”。
- 当发生日期时间类型之间的转换时，如果源类型是 `DATE` 类型而目的类型是 `DATETIME` 或 `TIMESTAMP` 类型，那么转换后时间部分为 '00:00:00'，如果源类型是 `TIME` 类型而目的类型是 `DATETIME` 或 `TIMESTAMP` 类型，那么转换后日期部分为系统当前日期，该日期受系统变量 `timestamp` 的影响。
- 如果在数值场景中使用日期时间相关类型，例如用作加法的参数，OceanBase 数据库会将其转换数值类型，反之亦然。
- 通常情况下，当一个值超出日期或时间类型的合法范围或者是无效的值时，OceanBase 数据库会将其转换为该类型的零值。例如对于 `DATETIME` 类型，转换结果为 '0000-00-00 00:00:00'，对于 `DATE` 类型，转换结果为 '0000-00-00'。一个例外是超出范围的 `TIME` 值会被转换为 `TIME` 类型合法范围的边界值。
- 设置 SQL Mode 可以控制 OceanBase 数据库支持哪些日期值。例如在 SQL Mode 中添加 `ALLOW_INVALID_DATES` 可以使数据库接受 '2009-11-31' 这样的日期值，此时数据库只会检查月是否出于 1~12 之间，日是否出于 1~31 之间。从 SQL Mode 中删去 `NO_ZERO_IN_DATE`，可以使数据库接受 '2020-00-01' 和 '2020-01-00'这样的日期值，此时使用 DATE_SUB 和 DATE_ADD 等函数可能无法得到正确的结果。
- OceanBase 数据库允许日期部分为 '0000-00-00'，如果类型是 `DATETIME` 或 `TIMESTAMP`，那么时分秒部分必须同样为 0。

下面的表格展示了日期时间相关的每个类型的零值。用户可以显式使用表格中的值表示零值，也可以使用 0 或 '0'，后者更加方便。

类型	零值格式
DATE	'0000-00-00'
TIME	'00:00:00'

类型	零值格式
DATETIME	'0000-00-00 00:00:00'
TIMESTAMP	'0000-00-00 00:00:00'
YEAR	0000

3.2.2.2.4. 字符类型

符类型将数字、字母等字符以字符串的形式存储。本节主要介绍最常用的字符数据类型，包括 CHAR 和 VARCHAR 类型以及 BINARY 和 VARBINARY 类型。

CHAR 和 VARCHAR 类型

`CHAR` 和 `VARCHAR` 类型是相似的，他们的区别在于存储和取回的方式不同以及对末尾空格的处理方式不同。

`CHAR` 和 `VARCHAR` 类型都可以指定最大长度。例如 `CHAR(30)` 最多存储 30 个字符。

`CHAR` 类型的列可以指定的长度范围是 0~255，`CHAR` 类型是将写入的值末尾连续空格去掉后进行存储的，当取回时，如果 `PAD_CHAR_TO_FULL_LENGTH` SQL 模式是开启状态，会将取回的值用空格填充到最大长度。

`VARCHAR` 类型的列可以指定的长度范围是 0~65536，有效的最大长度为 65536 个字节。

对 `CHAR` 和 `VARCHAR` 类型，如果插入的数据长度超过了最大长度，当 `sql_mode` 包含 `STRICT_TRANS_TABLES` 的设置时报错，当不包含该设置时会自动截取到最大长度并报警告。

在存储 `VARCHAR` 和 `CHAR` 类型时，比较方法是根据字符序的设置。

BINARY 和 VARBINARY 类型

`BINARY` 和 `VARBINARY` 类型与 `CHAR` 和 `VARCHAR` 类型相似，区别在于他们存储的是二进制数据。

`BINARY` 和 `VARBINARY` 类型的最大长度与 `CHAR` 和 `VARCHAR` 类型一样，不同的是它们的长度是按字节长度计算。

当存储 `BINARY` 类型时，待存储的值会在右侧补充 0x00（即 '\0'）到最长长度。

当开启严格模式时，插入超过最大长度会报错。当开启非严格模式时会进行截断并报出警告。

`BINARY` 类型的比较是会显式比较末尾的空字节的，0x00 与空格不同，0x00 排序会排在空格之前。

3.2.2.2.5. 大对象数据类型

OceanBase 数据库支持大对象数据类型，包括 BLOB 和 TEXT 两类。

`BLOB` 类型包括 `TINYBLOB` 、 `BLOB` 、 `MEDIUMBLOB` 和 `LONGBLOB` 。 `TEXT` 类型包括 `TINYTEXT` 、 `TEXT` 、 `MEDIUMTEXT` 和 `LONGTEXT` 。

`BLOB` 类型中存储二进制字符串， `BLOB` 的比较和排序基于其中每个字节的数值。 `TEXT` 类型存储非二进制字符串， `TEXT` 类型的比较与排序基于其字符集的字符序。如果为 `TEXT` 类型指定 `BINARY` 字符集，那么和 `BLOB` 是等价的。

在非严格模式下，用户可以向 `BLOB` 或 `TEXT` 列中插入超过列最大长度限制的字符串，超出的部分会被截断。在严格模式下，这种场景会报错。如果超出最大长度部分都是空格，那么无论是否开启严格模式都不会报错。

对于 `TEXT` 和 `BLOB` 类型，插入时不会在末尾补充空格，查询时也不会删除末尾空格。

大多数情况下，可以把 `BLOB` 当作长度足够大的 `VARBINARY` 类型，类似地把 `TEXT` 当作 `VARCHAR` 类型，不同之处在于 `BLOB` 和 `TEXT` 列不能有默认值。

> **注意**
>
> 不支持在 `TEXT` 或 `BLOB` 列上建索引。

`TINYTEXT` 、 `TEXT` 、 `MEDIUMTEXT` 和 `LONGTEXT` 四种类型中只有 `TEXT` 可以显式指定长度，OceanBase 数据库会根据用户指定的长度，在四种类型中推导出合适的类型。 `BLOB` 类型也类似。示例如下：

```
obclient> CREATE TABLE t(c1 TEXT(30), c2 TEXT(300), c3 TEXT(30000), c4 TEXT(10000000));
Query OK, 0 rows affected

obclient> DESC t;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| c1    | tinytext   | YES  |     | NULL    |       |
| c2    | text       | YES  |     | NULL    |       |
| c3    | mediumtext | YES  |     | NULL    |       |
| c4    | longtext   | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
```

3.2.2.3. 完整性约束

完整性约束（Integrity Constraint）用来确保数据遵从业务规则的要求，防止数据表中出现不满足规则的非法数据。本文主要介绍如何使用完整性约束以防止数据库的表中被插入非法数据。

OceanBase 数据库 MySQL 模式下的租户可以使用完整性约束以防止用户向数据库的表中插入非法数据。完整性约束的作用是确保数据库内存储的信息遵从一定的业务规则，如果 DML 语句的执行结果违反了完整性约束，将回滚语句并返回错误消息。

例如，用户在 `employees` 表的 `salary` 列上定义了完整性约束。此完整性约束规定 `salary` 列的数字值大于 10000 的数据行不能插入到 `salary` 表。如果某个 `INSERT` 或 `UPDATE` 语句违反了此完整性约束，将回滚语句并返回错误消息。

有关完整性约束的具体内容，请参见 数据完整性。

3.2.2.4. 表存储

OceanBase 数据库中使用宏块来存储数据，每张表可能包含多个宏块，每个宏块占用 2M 空间。宏块内包含一个或者多个微块，每个微块内包含一行或者多行数据。

表的组织形式

OceanBase 数据库中使用聚集索引表模型来组织，也就是存储时按照表的主键顺序来存储数据。当往表中插入一行数据时，会按照其主键的顺序插入到表中。

每行存储的列顺序是一样的。OceanBase 数据库通常是按照建表时的列顺序来存储，新加的列都放在最后一列存储。

行存储

行存储在微块中，一般情况下，行的所有列都是存储在一起的。当表中有大对象类型时，其所占用的存储空间会超过宏块大小，此时会将大对象类型溢出存储到其他的宏块中。

空值存储

OceanBase 数据库中会存储空值，在存储中会使用一个字节来标记某列是否为空。

3.2.2.5. 表压缩

OceanBase 数据库支持两种方式进行压缩数据，一种是采用数据编码的方式，另一种是使用通用的压缩算法，包括 lz4_1.0、zstd_1.0 等。

OceanBase 数据库在建表或者修改表属性时，通过设置 `row_format` 来打开或关闭数据编码，而设置 `COMPRESSION` 属性可以控制使用哪种压缩算法。

使用 ROW_FORMAT 控制是否打开数据编码

- 关闭数据编码。SQL 语句如下：

```
CREATE TABLE table_name ROW_FORMAT = REDUNDANT
ALTER TABLE table_name [SET] ROW_FORMAT = REDUNDANT

CREATE TABLE table_name ROW_FORMAT = COMPACT
ALTER TABLE table_name [SET] ROW_FORMAT = COMPACT

CREATE TABLE table_name ROW_FORMAT = DYNAMIC
ALTER TABLE table_name [SET] ROW_FORMAT = DYNAMIC
```

- 打开数据编码。SQL 语句如下：

```
CREATE TABLE table_name ROW_FORMAT = COMPRESSED
ALTER TABLE table_name [SET] ROW_FORMAT = COMPRESSED

CREATE TABLE table_name ROW_FORMAT = DEFAULT
ALTER TABLE table_name [SET] ROW_FORMAT = DEFAULT
```

使用 COMPRESSION 控制使用的压缩算法

使用 `COMPRESSION` 设置压缩算法的 SQL 语句如下：

```
CREATE TABLE table_name COMPRESSION 'compress_func_name'
ALTER TABLE table_name [SET] COMPRESSION 'compress_func_name'
```

其中，`compress_func_name` 可以配置成 `none` 、`zstd_1.0` 和 `lz4_1.0` ，`none` 代表不使用压缩算法。

详细信息请参考 压缩与编码。

3.2.2.6. 分区表

OceanBase 数据库支持普通表和分区表。

在 OceanBase 数据库中，分区是指根据一定的规则，把一个表或者索引分解成多个更小的、更容易管理的部分。每个分区都是一个独立的对象，具有自己的名称和可选的存储特性。

分区表由一个或多个分区组成，这些分区是单独管理的，可以独立于其他分区运行。表包括已分区或未分区。即使已分区表仅由一个分区组成，该表也不同于未分区表，非分区表不能添加分区。

OceanBase 数据库将每个表分区的数据存储在自己的 SStable 中。每个 SStable 包含表数据的一部分。

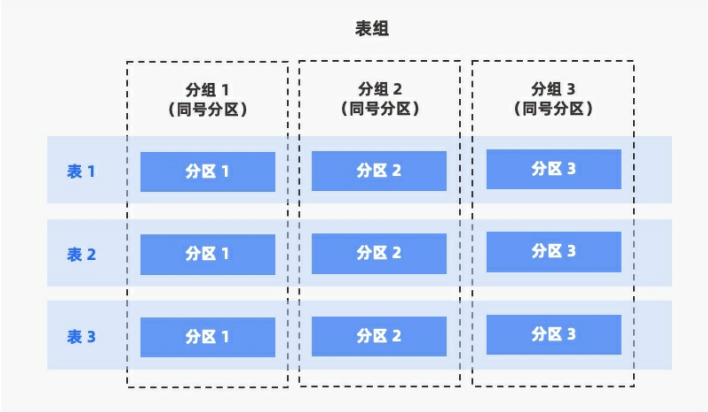
OceanBase 数据库提供了多种分区策略来使用多种业务的需求。由于分区是完全透明的，业务不需要进行过多的修改，就可以将分区功能应用到绝大多数业务。

3.2.2.7. 表组

对于分布式数据库，多个表中的数据可能会分布在不同的机器上，这样在执行 Join 查询或跨表事务等复杂操作时就需要涉及跨机器的通信，而表组功能可以避免这种跨机器操作，从而提高数据库性能。本文主要介绍表组功能及其工作原理。

作为分布式数据库，为了满足扩展性和多点写入等需求，OceanBase 数据库支持分区功能，即将一个表的数据分成多个分区来存储。表中用来参与具体数据分区的列称为分区键，通过一行数据的分区键值，对其进行 Hash 计算（数据分区的方式有多种，这里以 Hash 分区为例），能够锁定其所属的分区。

让用户将分区方式相同的表聚集到一起就形成了表组（以 Hash 分区为例，分区方式相同等价于分区个数相同，当然计算分区的 Hash 算法也是一样的），表组内每个表的同号分区称为一个分区组，如下图所示。



OceanBase 数据库在分区创建以及之后可能发生的负载均衡时，会将一个分区组的分区放到一个机器，这样即便存在跨表操作，只要操作数据所在的分区是属于同一个分区组，那么就不存在跨机器的操作。那么如何保障操作的数据在同一个分区组呢？OceanBase 数据库无法干涉用户的操作，但是可以根据业务特点大概率地保障某些操作涉及的跨表数据在同一分区组中。

以学生表和班级表为例，学生表和班级表中都有班级 `id`，可以将班级 `id` 列作为分区键，两表中同一班级的数据聚集到同一个分区中，这样在对班级 `id` 列作 JOIN 的时候，只需要在这个分区所在的机器上处理，不需要所有分区跨机锁定某一班级 `id` 对应的数据。对于分布式写事务的场景，如果增加一个同学信息，需要在学生表中增加一条数据，并在班级表中更新学生总数。因为这两个数据不在一个表，自然也不在一个分区，因此需要执行分布式写事务才能进行一致的更新。由于两个分区在同一台 OBServer，OceanBase 数据库对于同一台 OBServer 上的分布式事务执行优化，因此相对跨机的分布式事务效率更高。

因为分区方式相同的表才能聚集到一个表组中，所以在表组里的表不支持打破分区规则的分区操作，但是可以通过对表组作分区操作来更改其内所有表的分区。

目前的表组功能设计中，一个分区对应一个日志流，日志流通过 Paxos 算法将数据变动日志由主副本同步到备副本上，如果涉及多分区的事务，就对应多个日志流的写入，因此需要分布式事务才能完成一致的操作。尽管单机分布式事务相对跨机已经有一些优化，OceanBase 数据库支持绑定表组作为进一步的优化方式：分区组的分区不仅在一台机器上，而且将分区的改动写在一个日志流里，这样对一个分区组的跨表写事务就可以在一个日志流里原子的提交，将分布式事务优化为单机事务，可以达到更好的优化效果。由于分区组对应一个日志流，导致表从表组里删除变得很困难，OceanBase 数据库目前还不支持绑定表组来删除表。用户可以根据特点灵活使用表组。

综上，在典型的业务场景下，OceanBase 数据库支持的表组功能优化了分布式查询和分布式事务的场景，但是也引入一些限制。例如，不支持表组里的表单独做分区更改操作，在此基础上，又支持绑定表组，这使得在典型分布式事务场景下可以拥有更好的性能，但是也增加了一些限制，例如不支持从表组中删除表。

3.2.2.8. 主键表和无主键表

OceanBase 数据库支持主键表和无主键表。

主键（Primary Key）

主键是在数据表中能够唯一标识一行的列的集合。主键需要满足以下规则：

- 值不能为 `NULL` 或空串
- 在全表范围内主键列集合的值唯一
- 主键的值不允许变更

有主键表

有主键表即数据表中包含主键的表，在 OceanBase 数据库中需要满足以下规则：

- 每个数据表最多拥有一个主键列集合。
- 主键列的数量不能超过 64 列，且主键数据总长度不能超过 16 KB。

在创建有主键表后，会自动为主键列创建一个全局唯一索引，可以通过主键快速定位到行。

如下例所示，创建了一个以 `emp_id` 为主键的表 `emp_table`，它属于有主键表。

```
CREATE TABLE emp_table (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(100),
  emp_age INT NOT NULL
);
```

无主键表

数据表中未指定主键的表称为无主键表。

如下例所示，数据表 `student_table` 未指定主键，它属于无主键表。

```
CREATE TABLE student_table (
  student_id INT NOT NULL,
  student_name VARCHAR(100),
  student_age INT NOT NULL
);
```

OceanBase 数据库的无主键表采用自增列作为隐藏主键。无主键表利用了自增列的多分区全局唯一的原则，以此保证无主键表隐藏键的唯一性。

OceanBase 数据库的自增列是兼容 MySQL 的自增列模块，满足以下三个原则：

- 多分区全局唯一
- 语句内连续递增
- 生成的自增列值大于用户显式插入的值

3.2.3. 索引

3.2.3.1. 索引简介

索引是一种可选的结构，用户可以根据自身业务的需求来决定在某些字段创建索引，从而加快在这些字段的查询速度。本文主要介绍使用索引的优点和缺点以及索引的可用性和可见性。

索引的优缺点

索引的优点如下：

- 用户可以在不修改 SQL 语句的情况下，可以加速查询，只需要扫描用户所需要的部分数据。
- 索引存储的列数通常较少，可以节省查询 IO。

索引的缺点如下：

- 选择在什么字段上创建索引需要对业务和数据模型有较深地理解。
- 当业务发生变化时，需要重新评估以前创建的索引是否满足需求。
- 写入数据时，需要维护索引表中的数据，消耗一定的性能代价。
- 索引表会占用内存、磁盘等资源。

索引的可用性和可见性

索引可用性

在 Drop Partition 场景，如果没有指定 `rebuild index` 字段，会将索引标记为 `UNUSABLE` ，即索引不可用，此时，在 DML 操作中，索引是无须维护的，并且在该索引也会被优化器忽略。

索引可见性

索引的可见性是指优化器是否忽略该索引，如果索引是不可见的，则优化器会忽略该索引，但在 DML 操作中索引是需要维护的。一般在删除索引前，可以先将索引设置成不可见，来观察对业务的影响，如果确认无影响后，再将索引删除。

索引和键的关系

键是指一组列或者表达式，用户可以在键上创建索引。但索引和键是不同的，索引是存储在数据库中的对象，而键是逻辑上的概念。

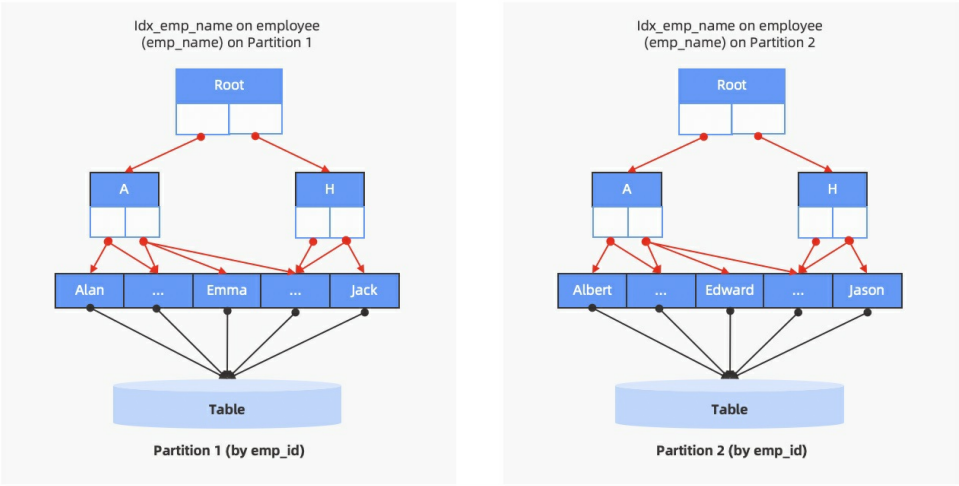
3.2.3.2. 局部索引和全局索引

OceanBase 数据库支持局部索引和全局索引。本文主要介绍局部索引和全局索引的概念以及创建索引时的默认行为。

局部索引

分区表的局部索引和非分区表的索引类似，索引的数据结构还是和主表的数据结构保持一对一的关系，但由于主表已经做了分区，主表的每一个分区都会有自己单独的索引数据结构。对每一个索引数据结构来说，里面的键（Key）只映射到自己分区中的主表数据，不会映射到其它分区中的主表，因此这种索引被称为局部索引。

从另一个角度来看，这种模式下索引的数据结构也做了分区处理，因此有时也被称为局部分区索引（Local Partitioned Index）。局部索引的结构如下图所示。



在上图中，`employee` 表按照 `emp_id` 做了 Range 分区，同时也在 `emp_name` 上创建了局部索引。

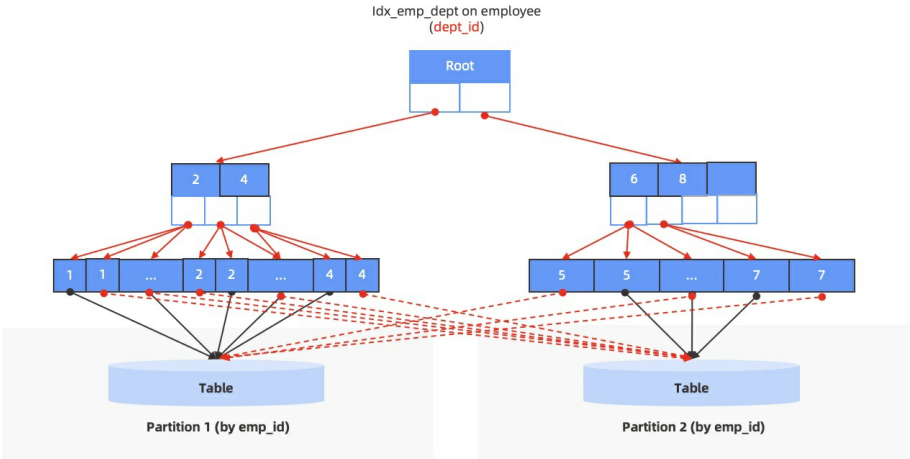
全局索引

和分区表的局部索引相比，分区表的全局索引不再和主表的分区保持一对一的关系，而是将所有主表分区的数据合成一个整体来看，索引中的一个键可能会映射到多个主表分区中的数据（当索引键有重复值时）。更进一步，全局索引可以定义自己独立的数据分布模式，既可以选择非分区模式也可以选择分区模式；在分区模式中，分区的方式既可以和主表相同也可以和主表不同。

因此，全局索引又分为以下两种形式：

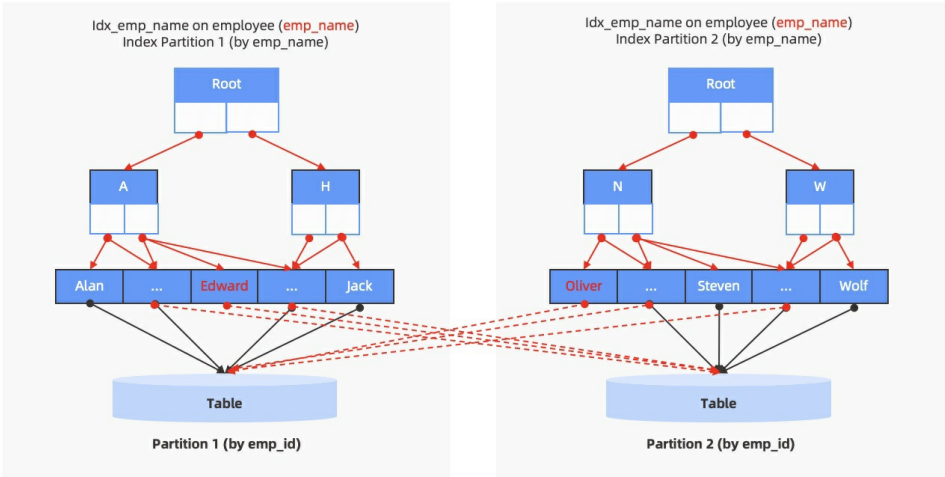
- 全局非分区索引（Global Non-Partitioned Index）

  索引数据不做分区，保持单一的数据结构，和非分区表的索引类似。但由于主表已经做了分区，因此会出现索引中的某一个键映射到不同主表分区的情况，即一对多的对应关系。全局非分区索引的结构如下图所示。



- 全局分区索引（Global Partitioned Index）

  索引数据按照指定的方式做分区处理，例如做哈希（Hash）分区或者范围（Range）分区，将索引数据分散到不同的分区中。但索引的分区模式是完全独立的，和主表的分区没有任何关系，因此对于每个索引分区来说，里面的某一个键都可能映射到不同的主表分区（当索引键有重复值时），索引分区和主表分区之间是多对多的对应关系。全局分区索引的结构如下图所示。



在上图中，`employee` 表按照 `emp_id` 做了 Range 分区，同时在 `emp_name` 上做了全局分区索引。可以看到同一个索引分区里的键，会指向不同的主表分区。

由于全局索引的分区模式和主表的分区模式完全没有关系，看上去全局索引更像是另一张独立的表，因此也会将全局索引叫做索引表，理解起来会更容易一些（和主表相对应）。

> **说明**
>
> 非分区表也可以创建全局分区索引。但如果主表没有分区的必要，通常来说索引也就没有必要分区了。

推荐使用全局索引的场景包括：

- 业务上除了主键外，还有其他列的组合需要满足全局唯一性的强需求，这个业务需求仅能通过全局性的唯一索引来实现。
- 业务的查询无法得到分区键的条件谓词，且业务表没有高并发的同时写入，为避免进行全分区的扫描，可以根据查询条件构建全局索引，必要时可以将全局索引按照新的分区键来分区。

需要注意的是，全局索引虽然为全局唯一、数据重新分区带来了可能，解决了一些业务需要根据不同维度进行查询的强需求，但是为此付出的代价是每一笔数据的写入都有可能变成跨机的分布式事务，在高并发的写入场景下它将影响系统的写入性能。当业务的查询可以拥有分区键的条件谓词时，OceanBase 数据库依旧推荐构建局部索引，通过数据库优化器的分区裁剪功能，排除掉不符合条件的分区。这样的做法可以同时兼顾查询和写入的性能，让系统的总体性能表现更优。

创建索引时的默认行为

当用户创建索引时，如果没有指定 `LOCAL` 或者 `GLOBAL` 关键字，在分区表上会默认会创建出全局索引。

3.2.3.3. 唯一索引和非唯一索引

索引表分为唯一索引和非唯一索引，其中唯一索引保证在表内的索引列上不存在两行有完全相同的值，而非唯一索引则可能存在完全相同的值。在 OceanBase 数据库中，NULL 值也会存储在索引中。

对于非唯一索引，索引表的存储键是用户指定的索引列和主表主键；而对于唯一索引列，索引表的存储键是用户指定的索引列和可变的主表主键列，这里可变的含义是主键列的值是随索引列值的不同而不同。

如下例所示，对于非唯一索引表 `i1` ，它的存储键是 `c2` 和 `c1` ，而对于唯一索引表 `i2` ，它的存储键是 `c2` ，而 `c1` 是可变的。当 `c2` 为 `NULL` 时，可变的 `c1` 列值是主表 `c1` 列的值；当 `c2` 不为 `NULL` 时，可变的 `c1` 列值为 `NULL` 。

```
CREATE TABLE t1(c1 INT, c2 INT, c3 INT, c4 INT, PRIMARY KEY(c1));
CREATE INDEX i1 ON t1(c2);
CREATE UNIQUE INDEX i2 ON t1(c3);
```

3.2.3.4. 索引的使用

当用户创建好索引后，OceanBase 数据库会自动维护该索引，所有的 DML 操作都会实时更新索引表相应的数据记录，同时优化器也会根据用户的查询来自动地选择是否使用索引。本文主要介绍如何进行索引扫描。

当 SQL 查询语句指定谓词条件查询的是索引列时，数据库会自动地抽取谓词条件作为查询索引的范围，也就是查询索引表的起始键和终止键。数据库根据起始键能定位到数据开始的位置，根据终止键能定位出数据结束位置，而开始和结束位置范围内所包含的数据是需要被此查询扫描的数据。

对于索引表，OceanBase 数据库存储时使用 MemTable 和 SSTable 来存储数据，其中 MemTable 使用的是 B+ 树结构，而 SSTable 使用的是宏块结构。在 MemTable 或者 SSTable 都按照上述扫描过程，扫描出相应的数据，而最终的数据行是由 MemTable 和 SSTable 的数据行融合成完整的数据行。

因此，OceanBase 数据库查询索引表数据的完整过程如下：

1. 在 MemTable 中查询数据。
2. 在 SSTable 中查询数据。
3. 将 MemTable 和 SSTable 中的数据融合，得到完整的行。

当 SQL 查询语句只涉及到索引表中的列时，那么数据库会根据用户指定的列，按照上述查询过程，查询相应索引表的 MemTable 和 SSTable，得到完整的数据行。

当 SQL 查询语句除了包含索引表中的列，还包含其他列时，数据库会先通过索引表，查询出相关的行，并根据行上的主键，按照上述查询过程，到主表中查询所需要的数据列，这个过程也称为回表。

3.2.3.5. 索引的存储

在 OceanBase 数据库中，索引表的存储类似于普通的数据表，其数据也是存储在宏块和微块的结构中。由于 OceanBase 数据库使用的是聚集索引表模型，因此在索引的行中除了用户指定的索引列之外，还会存储主表的主键列，以方便进行回表。

3.2.4. 分区

3.2.4.1. 分区概述

在 OceanBase 数据库中，分区是指根据一定的规则，把一个表分解成多个更小的、更容易管理的部分。每个分区都是一个独立的对象，具有自己的名称和可选的存储特性。本章节主要介绍分区的相关概念以及使用分区的好处。

对于访问数据库的应用而言，逻辑上访问的只有一个表或一个索引，但是实际上这个表可能由数十个物理分区对象组成，每个分区都是一个独立的对象，可以独自处理访问，也可以作为表的一部分处理访问。分区对应用来说是完全透明的，不影响应用的业务逻辑。

从应用程序的角度来看，只存在一个 Schema 对象。访问分区表不需要修改 SQL 语句。分区对于许多不同类型的数据库应用程序非常有用，尤其是那些管理大量数据的应用程序。

分区表可以由一个或多个分区组成，这些分区是单独管理的，可以独立于其他分区运行。表可以是已分区或未分区的，即使已分区表仅由一个分区组成，该表也不同于未分区表，非分区表不能添加分区。

分区表也可以由一个或多个表分区段组成。OceanBase 数据库将每个表分区的数据存储在自己的 SStable 中，每个 SStable 包含表数据的一部分。

使用分区的好处如下：

- 提高了可用性

  分区不可用并不意味着对象不可用。查询优化器自动从查询计划中删除未引用的分区。因此，当分区不可用时，查询不受影响。

- 更轻松地管理对象

  分区对象具有可以集体或单独管理的片段。DDL 语句可以操作分区而不是整个表或索引。因此，可以对重建索引或表等资源密集型任务进行分解。例如，可以一次只移动一个分区。如果出现问题，只需要重做分区移动，而不是表移动。此外，对分区进行 `TRUNCATE` 操作可以避免大量数据被 `DELETE` 。

- 减少 OLTP 系统中共享资源的争用

  在 TP 场景中，分区可以减少共享资源的争用。例如，DML 分布在许多分区而不是一个表上。

- 增强数据仓库中的查询性能

  在 AP 场景中，分区可以加快即席查询的处理速度。分区键有天然的过滤功能。例如，查询一个季度的销售数据，当销售数据按照销售时间进行分区时，仅仅需要查询一个分区或者几个分区，而不是整个表。

- 提供更好的负载均衡效果

  OceanBase 数据库的存储单位和负载均衡单位都是分区。不同的分区可以存储在不同的节点。因此，一个分区表可以将不同的分区分布在不同的节点，这样可以将一个表的数据比较均匀的分布在整个集群。

3.2.4.2. 分区键

分区键是一个或多个列的集合，用于确定分区表中的每一行所在的分区。每一行都明确地分配给一个分区。

例如，对于一个用户表，可以指定 `user_id` 列作为 Range 分区的键。数据库根据此列中的用户号是否在指定范围内，将行分配给分区。

OceanBase 数据库使用分区键自动将插入、更新和删除操作定向到相应的分区。

3.2.4.3. 分区类型

分区策略

OceanBase 数据库提供了多种分区策略，用于控制数据库如何将数据放入分区。

OceanBase 数据库的基本分区策略包括范围（Range）分区、列表（List）分区和哈希（Hash）分区。

一个一级分区仅限使用一种数据分配方法。例如，仅使用 List 分区或仅使用 Range 分区。

在进行二级分区时，表首先通过一种数据分配方法进行分区，然后使用第二种数据分配方法将每个分区进一步划分为二级分区。例如，一个表中包含 `create_time` 列和 `user_id` 列，您可以在 `create_time` 列上使用 Range 分区，然后在 `user_id` 列上使用 Hash 进行二级分区 。

Range 分区

Range 分区是最常见的分区类型，通常与日期一起使用。在进行 Range 分区时，数据库根据分区键的值范围将行映射到分区。

Range 分区的分区键只支持一列，并且只支持 `INT` 类型。

如果要支持多列的分区键，或者其他数据类型，可以使用 Range Columns 分区。

Range Columns 分区作用跟 Range 分区基本类似，不同点如下：

- Range Columns 拆分列结果不要求是整型，可以是任意类型。
- Range Columns 拆分列不能使用表达式。
- Range Columns 拆分列可以写多个列（即列向量）。

Hash 分区和 Key 分区

Hash 分区

在进行 Hash 分区时，数据库根据数据库应用于用户指定的分区键的哈希算法将行映射到分区。

行的目标分区是由内部 Hash 函数计算出一个 Hash 值，再根据 Hash 分区个数来确定的。当分区数量为 2 的幂次方时，哈希算法会创建所有分区中大致均匀的行分布。

哈希算法在分区之间均匀分布行，使分区的大小大致相同。

Hash 分区是在节点之间均匀分布数据的理想方法。Hash 分区也是 Range 分区的一种易于使用的替代方法，特别是当要分区的数据不是历史数据或没有明显的分区键的场景。

Hash 分区在具有极高更新冲突的 OLTP 系统里面非常有用。这是因为 Hash 分区将一个表分成几个分区，将一个表的修改分解到不同的分区修改，而不是修改整个表。

Hash 分区键的表达式必须返回 `INT` 类型。

Key 分区

Key 分区和 Hash 分区类似。主要区别如下：

- Hash 分区的分区键可以是用户自定义的表达式，而 Key 分区的分区键只能是列，或者不指定。
- key 分区的分区键不限于 `INT` 类型。

key 分区可以指定或不指定列，也可以指定多个列作为分区键。如果表上有主键，那么这些列必须是表的主键的一部分，或者全部。如果 Key 分区不指定分区键，那么分区键就是主键列。如果没有主键，有 `UNIQUE` 键，那么分区键就是 `UNIQUE` 键。

List 分区

在进行 List 分区时，数据库使用离散值列表作为每个分区的分区键。分区键由一个或多个列组成。

您可以使用 List 分区来控制单个行如何映射到指定分区。当不方便根据分区键进行排序时，可以使用 List 分区对数据进行分组和管理。

List 分区仅支持单分区键，分区键可以是一列，也可以是一个表达式。分区键的数据类型仅支持 `INT` 类型。

如果要使用多列的 List 分区，或者其他数据类型的 List 分区，可以使用 List Columns 分区。

List Columns 分区是 List 分区的一个扩展，支持多个分区键，并且支持 `INT` 数据、 `DATE` 类型和 `DATETIME` 类型。

组合分区

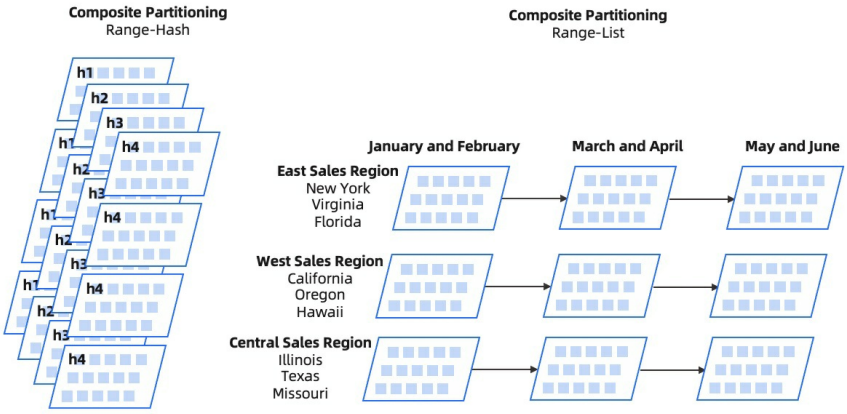
范围（Range）分区、列表（List）分区和哈希（Hash）分区都可以作为组合分区表的二级分区策略。

在进行组合分区时，表通过一种数据分配方法进行分区，然后使用第二种数据分配方法将每个分区进一步细分为二级分区。因此，组合分区结合了基本的数据分发方法。指定分区的所有二级分区代表数据的逻辑子集。

组合分区有如下优点:

- 根据 SQL 语句，在一维或二维上进行分区修剪可能会提高性能。
- 查询可以在任一维度上使用全分区或部分分区连接。
- 您可以对单个表执行并行备份和恢复。
- 分区的数量大于单层分区，这可能有利于并行执行。
- 您可以实现一个滚动窗口来支持历史数据，如果许多语句可以从分区修剪或分区连接中受益，则仍然可以在另一个维度上进行分区。
- 您可以根据分区键的标识以不同方式存储数据。例如，您可能决定以只读的压缩格式存储特定产品类型的数据，并保持其他产品类型的数据不压缩。

下图展示了 Range-Hash 和 Range-List 组合分区。



3.2.4.4. 分区索引

在数据库中，用户可以通过建立索引的方式来提升查询性能，当数据表中的数据量越来越大时，则可以通过分区的方式将数据表拆分成若干分片，利用负载均衡将数据分片打散到整个集群中来提高集群整体服务能力。本文主要介绍局部索引、全局索引和唯一索引。

索引类型

OceanBase 数据库主要涉及如下索引类型:

- 局部分区索引：在创建索引时指定关键字 `LOCAL` 的索引为局部索引，局部索引无须指定分区规则，它的分区属性和主表的属性一致，也会跟随主表的分区操作而发生变更。

- 全局分区索引：在创建索引时指定关键字 `GLOBAL` 的索引为全局索引，全局索引可以按照分区规则进行分表。
- 唯一索引：索引键值具有唯一性，用关键字 `UNIQUE` 表示。
- 前缀索引：当分区索引表的分区键是索引列的左前缀时，该索引称为前缀索引。这不同于 MySQL 长字符串的前缀索引。例如，索引表 `idx` 建立在 `c1` 和 `c2` 列上，如果该索引表的分区键为 `c1`，那么该索引称为前缀索引；如果该索引表的分区键为 `c2` 或者其他列，则称为非前缀索引。
- 非前缀索引：相对于分区前缀索引而言，如果分区索引不是前缀索引，那么称为非前缀索引。

局部索引

局部索引根据分区键划分为局部前缀索引和局部非前缀索引。

局部前缀索引

如果分区键是索引的左前缀，并且索引包含二级分区键，那么这个索引为局部前缀索引。局部前缀索引可以是唯一索引或者非唯一索引。

指定索引键的查询利用局部前缀索引可以唯一定位到一个索引分区，非常适合于结果集比较小，但是需要进行分区裁剪的场合。

例如，表 A 上有个局部索引 `idx(c1,c2,c3)`，主表根据 `c1` 进行分区，根据局部索引的特性，索引表和主表的分区方式一样，因此 `idx` 也以 `c1` 为分区键，根据定义知道这是一个局部前缀索引，当处理指定索引键的查询时，可以通过分区键 `c1` 的值定位到唯一一个索引分区，大大减少了索引分区的访问。

局部非前缀索引

如果索引表不是局部前缀索引，那就是局部非前缀索引。可能的情况是，索引表的分区键不是索引的左前缀，或者索引没有包含二级分区键。

如果分区键不是索引的子集，那么局部非前缀索引不能是唯一索引。 指定索引键的查询利用局部非前缀索引，不能定位到索引分区，而是需要访问所有索引分区，因此比较适合访问数据量比较大，注重并发的场景。

例如，表 A 上有个局部索引 `idx(c1,c2,c3)`，主表根据 `c4` 列进行分区，根据定义获知这是一个局部非前缀索引，当用户查询指定索引键时，不能通过分区键定位到索引分区，因此需要访问所有的索引分区才能获得结果，并发执行在这种场景下可以发挥重要的作用。

全局索引

全局索引拥有自己独立的分区定义，不需要跟主表一样。同时，全局索引表的分区也可以进行分裂和合并。一般情况下，如果主表和全局索引的分区方式完全一样的话，除去具有唯一性的非前缀索引，其他索引建议定义成局部索引，全局索引在分区管理和维护上代价要远远大于局部索引，并且从查询代价、分区裁剪上来说，具有相同分区方式的全局索引和局部索引的效果是一样的。

全局前缀索引

如果全局分区索引的分区键是索引键的左前缀，那么这个索引称为全局前缀索引。

全局前缀索引可以是唯一索引或者非唯一索引。

全局前缀索引只在用 Range 分区时有意义，对于 Hash 分区索引无意义。原因在于，如果是用户选择 Hash 分区索引，那么用户查询模式一定是指定索引键的点查询，索引键如果覆盖分区键的话，那么是否为前缀索引并无意义，都能够通过用户指定的索引键值算出索引分区；如果用户没有指定全部分区键值，Hash 分区索引则需要访问所有的分区数据，而 Range 分区可以进行一定程度的分区裁剪。

全局非前缀索引

OceanBase 数据库不支持全局非前缀索引，全局非前缀索引对于查询优化并没有太多意义。

例如，表 A 上有一个全局索引 `idx(c1,c2)`，`idx` 通过 `c2` 进行分区，那么 `idx` 为一个非全局索引。这种情况下，只有当用户指定全部索引键值的时候才能进行分区裁剪，其他情况均需要扫描所有的索引分区，因此用户没有理由不直接用 `c1` 键做分区，使用 `c1` 还能通过前缀过滤进行分区裁剪。

唯一索引

唯一索引可以定义为全局索引或者是局部索引。

唯一索引定义为局部索引的时候，需要满足一定的条件，即索引键要覆盖索引分区键。 例如，表 A 上有个唯一索引 `idx(c1,c2,c3)`，索引表 `idx` 按照 `(c1,c2)` 进行分区，这样能保证相同的 `(c1,c2)` 一定能进入同一分区，只需要在单个分区内维护唯一性。如果索引表 `idx` 按照 `(c2,c4)` 进行分区，索引没有覆盖分区键，就不能通过局部索引来保证分区之间索引键的唯一性，这样的局部唯一索引不能被创建。

索引创建策略

在创建索引的时候，需要将用户查询模式、索引管理、性能、可用性等方面的需求综合起来考虑，选择一个最合适自身业务的索引方式。

- 如果用户需要唯一索引，并且索引键覆盖所有分区键，则可以定义成局部索引，否则需要用全局索引。
- 如果主表的分区键是索引的子集，则可以采用局部索引。
- 如果主表分区属性和索引的分区属性相同，建议采用局部索引。
- 如果用户比较关注索引分区管理的代价，主表分区总是不断进行分区删减时，尽量避免创建全局索引。这是由于主表分区删减操作会导致全局索引变更太大，难以恢复，而且可能会导致索引不可用。
- 如果用户查询总是指定所有索引分区键值，那么只需要在其他列上建索引，而无须包含分区键，减少维护代价和存储代价；对于不指定分区键的查询，前缀索引更合适做分区裁剪、数据量比较小的场景，前缀索引一般更适合做分区并行、数据量比较大的场景。

分区选择策略

在进行索引分区的时候，如果单个索引表的数据量太大则需要进行分区，以便查询更好的进行分区并行以及负载均衡。

- 如果用户查询多为指定索引键的单点查询，从访问分区的数目和访问并发角度来说，Hash 分区和 Range 分区的查询代价都相差不大。但是如果数据具有热点区的话，用 Hash 分区能更好的避免热点问题。
- 如果用户的查询多为指定索引键的范围查询，则从访问的分区数目来说，Range 分区要优于 Hash 分区；但从并发的角度考虑，Hash 分区可以更好的利用并发查询的优势，在查询结果集很大的情况下，Hash 分区的并发查询应该能有更大的性能优势。

查询时索引选择策略

- 如果用户更注重分区裁剪的话，选择前缀索引更好，在用查询过滤条件里面指定分区前缀的值能够最大程度的进行分区裁剪，减少索引分区数据的读取。

- 如果用户注重吞吐量，访问的数据量比较大时，选择非前缀索引更能提高性能，可以通过分区并行的方式来解决分区键上的范围查询；局部非前缀索引可以并发访问所有的分区，但是局部前缀索引会进行分区裁剪，由少量分区处理大量的数据集合，响应时间也许会比并发查询要差一些。

3.2.5. 视图

3.2.5.1. 视图概述

视图是从一个或多个表导出的虚拟的表。本质上，视图是一个存储好的查询，用户通过访问这个视图来获取该视图定义的数据。本章节主要介绍视图相关的基本概念和使用方法。

视图从它所基于的表（称为基表）中获取数据。基表可以是表或其他视图。对视图执行的所有操作实际上都会影响基表。您可以在使用表的大多数场景中使用视图。

> **注意**
>
> 物化视图与标准视图使用不同的数据结构。

视图的优势

视图使您能够为不同类型的用户定制数据的呈现方式。视图拥有如下优势：

- 把经常使用的数据定义为视图以简化操作。数据库的查询大多要使用聚合函数，同时还要显示其它字段的信息，可能还会需要关联到其它表，这时涉及的 SQL 语句可能比较复杂。如果需要频繁执行此查询，就可以通过创建视图简化查询操作，之后只需要执行 `SELECT * FROM view_name` 就可以获得预期结果。
- 视图限制用户只能查询和修改可视的数据，提高了数据的安全性。视图是动态的数据集合，数据随着基表的更新而更新。但是视图是虚拟的，物理上是不存在的，只是存储了数据的集合，所以可以将基表中重要的字段信息屏蔽，不通过视图展示给用户。
- 视图拥有逻辑上的独立性，屏蔽了真实表的结构带来的影响。视图可以使应用程序和数据库的表在一定程度上互相独立。如果没有视图，应用一定是建立在表上的。创建了视图之后，程序可以建立在视图之上，从而程序与数据库表被视图分割开来。

视图的特点

与表不同，视图没有分配存储空间，视图也不包含数据。相反，视图是从视图引用的基表中提取或派生数据。因此除了用于定义数据字典中视图的查询的存储之外，它不需要其他存储。

视图依赖于其引用的对象，如果视图所依赖的对象被删除、更新或者重建，则数据库会确定新对象是否可以被视图定义所接受。

3.2.5.2. 视图的数据操作

因为视图是从表中派生出来的，所以它和表有很多相似之处。用户可以查询视图，并且在一定限制下，对视图执行 DML 操作。对视图执行的操作会影响视图的某些基表中的数据，并受基表的完整性约束和触发器的约束控制。

如下示例中，在表 `employees` 上定义视图 `staff_dept_10` 。这个视图定义了部门是 10 的员工信息。

使用关键字 `CHECK OPTION` 创建视图，表明该视图上的 DML 操作只能作用在视图所定义的数据中。因此，可以插入部门为 10 中的员工的行数据，但不能插入部门为 30 中的员工的行数据。

```
CREATE VIEW staff_dept_10 AS
   SELECT employee_id, last_name, job_id, manager_id, department_id
   FROM   employees
   WHERE  department_id = 10
   WITH CHECK OPTION CONSTRAINT staff_dept_10_cnst;
```

3.2.5.3. 视图的数据访问

OceanBase 数据库会在内部的数据字典中保存视图对应的查询语句。

当用户尝试通过视图读取数据时，OceanBase 数据库会执行以下操作步骤：

1. 解析用户查询，如果在解析过程中遇到视图名称时，从数据字典中获取视图对应的查询语句并解析。
2. 尝试将该视图与用户查询合并，合并之后有可能生成更好的执行计划。
3. 生成执行计划，并执行该语句。

示例：

1. 创建一个名为 `staff_dept_10` 的视图，该视图的定义如下所示。

   ```
   CREATE VIEW staff_dept_10 AS
       SELECT employee_id, last_name, job_id, manager_id, department_id
       FROM   employees
       WHERE  department_id = 10
   ```

2. 用户执行以下查询访问 `staff_dept_10` 。

   ```
   SELECT last_name FROM  staff_dept_10 WHERE  employee_id = 200;
   ```

3. OceanBase 数据库首先会将以上查询解析成如下查询。

   ```
   SELECT last_name
   FROM   (SELECT employee_id, last_name, job_id,
                          manager_id, department_id
                                   FROM   employees
                                   WHERE  department_id = 10) staff_dept_10
   WHERE  employee_id = 200;
   ```

4. 查询改写会尝试将视图定义与主查询合并。

   ```
   SELECT last_name
   FROM  employees
   WHERE  employee_id = 200 and department_id = 10;
   ```

5. OceanBase 数据库的 SQL 引擎会生成以上 SQL 对应的执行计划并执行。

3.2.5.4. 可更新的视图

用户除了通过视图读取数据外，还可以通过视图更新数据。这些被更新的视图称之为可更新视图。

例如，对于视图 `staff_dept_10` ，用户可以执行如下删除操作：

```
DELETE staff_dept_10 WHERE employee_id = 200;
```

OceanBase 数据库在接受到这个请求后，处理的流程如下：

1. 解析该语句，并将 `staff_dept_10` 替换成视图的定义。
2. 检查该视图作为被更新的对象，是否满足一些必要的约束。
3. 尝试将该视图合并到主语句中。
4. 生成计划并执行。

在 OceanBase 数据库内部，实际执行的删除语句是：

```
DELETE employees WHERE employee_id = 200 and department_id = 10;
```

3.2.6. 系统视图

3.2.6.1. 字典视图

OceanBase 数据库 MySQL 模式下拥有数据字典，数据字典表受到保护，只能使用 DEBUG 调试获取到相关信息。字典视图是用来访问数据字典的，在 MySQL模式下支持通过 INFORMATION_SCHEMA 表和 SHOW 语句访问存储在数据字典表中的数据。

字典视图的组成

OceanBase 数据库 MySQL 模式下的字典视图包含 `information_schema.*` 相关视图、`oceanbase.CDB_` 前缀视图以及 `mysql.*` 视图。

带有 INFORMATION_SCHEMA 前缀的视图

INFORMATION_SCHEMA 提供对 MySQL 租户中数据库元数据（例如数据库或表的名称、列的数据类型或访问权限）的访问。 有时也叫做数据字典或系统目录。INFORMATION_SCHEMA 是每个 MySQL 租户中的一个 Database/Schema，用于存储有关 MySQL 租户维护的所有其他数据库的信息。INFORMATION_SCHEMA 数据库包含几个只读表。它们实际上是视图，而不是基表。

带有 mysql 前缀的视图

带有 `mysql` 前缀的视图都是系统视图，包含存储 OceanBase 数据库 MySQL 模式下服务器运行时所需信息的表。一般地，带 `mysql` 前缀的视图包含存储数据库对象元数据的数据字典表和用于其他操作目的的系统表。OceanBase 数据库兼容了部分 `mysql` 前缀的视图，例如 `help_*` 视图包含了服务器端的一些帮助信息，`time_zone*` 记录了时区相关的信息，`USER` 视图和 DB 视图记录了用户权限相关的一些信息。

带有 oceanbase.CDB 前缀的视图

OceanBase 数据库对于每个 `DBA_*` 视图，都对应在系统租户下定义了一个 `CDB_*` 视图。`CDB_*` 视图可用于获取包含在系统租户和普通租户中的部分数据库对象的有关信息。

字典视图的工作机制

字典对象缓存是一个共享的全局缓存，它将以前访问过的数据字典对象存储在内存中，以实现对象重用并最小化磁盘 I/O。字典对象缓存使用基于 LRU 的逐出策略从内存中逐出最近最少使用的对象。

字典对象缓存包括存储不同对象类型的缓存分区。一些缓存分区大小限制是可配置的，而其他的则是硬编码的。

字典视图的存储

OceanBase 数据库中，将每个租户元数据作为一个整体存储的数据字典仅存储在系统租户中。

由于存储指定租户元数据的数据字典存储在此租户专用的字典表中，因此每组数据字典表都存储在自己的指定的租户中，不同租户不可以跨租户访问到其他租户字典信息。

3.2.6.2. 性能视图

和 Oracle 模式一样，MySQL 模式下同样包含一组动态性能视图。

性能视图简介

因为它们在打开和使用数据库时不断更新，并且其内容主要与性能有关，所以这些视图被称为动态性能视图。

尽管这些视图似乎是常规数据库表，但这些视图仅提供有关内部磁盘结构和内存结构的数据。您可以从这些视图中进行选择，但不能对其进行更新或更改。

OceanBase 数据库拥有动态性能表，其名称以 `V$` 开头。在这些表上创建视图，然后使用前缀为 `V$` 的公共同义词。

每个 `V$` 视图几乎都有相应的 `GV$` 视图，即全局的 `V$` 视图。查询 `GV$` 视图将返回所有符合条件的 `V$` 视图信息。

性能视图的存储

动态性能视图是基于数据库内存结构而构建的虚拟表。

动态性能视图不是存储在数据库中的常规表，因为数据是动态更新的，所以无法保证视图的读取一致性。

因为动态性能视图不是真正的表，所以数据内容取决于集群和服务结点的状态。

3.2.7. 数据完整性

3.2.7.1. 数据完整性概述

OceanBase 数据库 MySQL 模式下的租户可以使用完整性约束（Integrity Constraint）防止用户向数据库的表中插入非法数据。本章节主要介绍完整性约束类型和应用场景等。

完整性约束的作用是确保数据库内存储的信息遵从一定的业务规则。例如，如果 DML 语句的执行结果违反了完整性约束，将回滚语句并返回错误消息。

在视图执行的操作需要遵从基表（Base Table）上的完整性约束。

例如，用户在 `employees` 表的 `salary` 列上定义了完整性约束。此完整性约束规定 `salary` 列的数字值大于 10,000 的数据行不能插入 `employees` 表。如果某个 `INSERT` 或 `UPDATE` 语句违反了此完整性约束，将回滚语句并返回错误消息。

3.2.7.2. 完整性约束类型

3.2.7.2.1. 完整性约束类型概述

OceanBase 数据库支持多种类型的完整性约束。

用户可以使用以下完整性约束对输入的列值加以限制：

- `NOT NULL` 完整性约束
- `UNIQUE KEY` 完整性约束
- `PRIMARY KEY` 完整性约束
- 引用完整性约束

NOT NULL 完整性约束

非空规则（Null Rule）是定义在某一列上的规则，其作用是禁止将要被插入或更新的数据行的列值为空值 `NULL` 。

UNIQUE KEY 完整性约束

唯一值规则（Unique Value Rule）是定义在某一列（或某一列集）上的规则，其作用是确保将要被插入或更新的数据行此列（或列集）的值是唯一的。

PRIMARY KEY 完整性约束

主键值规则（Primary Key Value Rule）是定义在某一键（Key）（键指一列或一个列集）上的规则，其作用是确保表内的每一数据行都可以由某一个键值唯一地确定。

引用完整性约束

引用完整性规则（Referential Integrity Rule）是定义在某一键（Key）（键指一列或一个列集）上的规则，其作用是确保任意键值都能与相关表（Related Table）的某一键值，即引用值（Referenced Value）相匹配。

引用完整性约束时，对引用值可以进行哪些类型的数据操作（Data Manipulation），以及这些操作将如何影响依赖值（Dependent Value）。请参见如下具体规则：

- 限制（Restrict）：不允许对引用值进行更新与删除。
- 置空（Set to Null）：当因引用值被更新或删除后，所有受影响的依赖值都将被置为 `NULL` 。
- 置默认值（Set to Default）：当引用值被更新或删除后，所有受影响的依赖值都将被赋予一个默认值。
- 级联操作（Cascade）：当引用值被更新后，所有受影响的依赖值也将被更新为相同的值。当引用数据行（Referenced Row）被删除后，所有受影响的依赖数据行（Dependent Row）也将被删除。

- 无操作（No Action）：不允许对引用值进行更新与删除。此规则与 `RESTRICT` 有所不同，它只在语句结束时进行检查，如果约束被延迟（Deferred）则在事务结束时进行检查。

3.2.7.2.2. NOT NULL 完整性约束

在 OceanBase 数据库中，NOT NULL 约束可用于限制一个列中不能包含 NULL 值。

语法如下：

```
column_name data_type NOT NULL;
```

用户无法向包含 `NOT NULL` 约束的列中插入 `NULL` 值，或更新旧值为 `NULL` 。

把 `NOT NULL` 称作列的一个属性比称之为约束更准确，因为在 INFORMATION_SCHEMA 中和约束相关的表中，查询不到 `NOT NULL` 相关的数据，这些表包括 CHECK_CONSTRAINTS、REFERENTIAL_CONSTRAINTS、TABLE_CONSTRAINTS 和TABLE_CONSTRAINTS_EXTENSIONS。

OceanBase 数据库目前支持删除列上的 `NOT NULL` 约束，但不支持在列上追加 `NOT NULL` 约束。删除的语法如下：

```
ALTER TABLE table_name MODIFY column_name data_type;
```

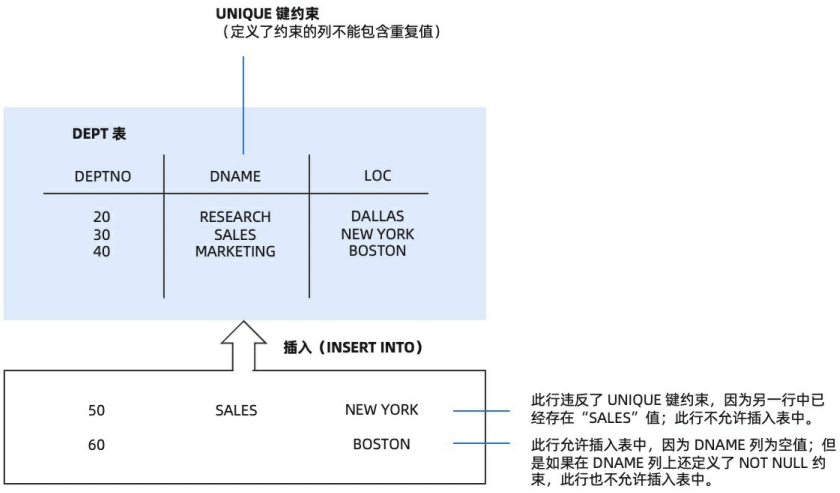
3.2.7.2.3. 唯一性约束

唯一键（UNIQUE key）约束是关于唯一列值（Unique Column Value）的规则，定义在某一列或某一列族上，其作用是确保将要被插入或更新的数据行的列或列族的值是唯一的，表的任意两行的某列或某个列集的值不重复。

本文通过以下具体示例来介绍唯一性约束的主要特性。

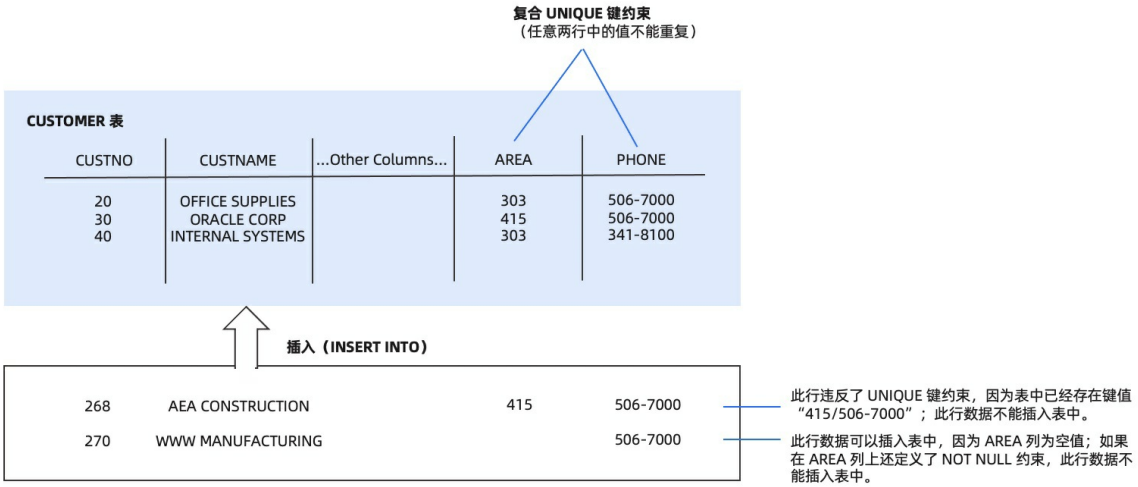
```
obclient> CREATE TABLE t1(c1 INT, c2 INT, CONSTRAINT t1_uk_c1_c2 UNIQUE(c1, c2));
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(1, 1);
ORA-00001: unique constraint '1-1' for key 'T1_UK_C1_C2' violated

obclient> INSERT INTO t1 VALUES(null, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(null, 1);
ORA-00001: unique constraint 'NULL-1' for key 'T1_UK_C1_C2' violated

obclient> INSERT INTO t1 VALUES(null, null);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(null, null);
Query OK, 1 row affected
```

又如，在下图中，`dept` 表的 `dname` 列上定义了 `UNIQUE` 键约束，则不允许此表存在重复的部门名称，但是允许用户向 `dname` 列中插入空值。如果 `dname` 列上还定义了 `NOT NULL` 约束，则不允许用户向 `dname` 列中插入空值。



如果 `UNIQUE` 键由多列构成，那么这组数据列被称为复合唯一键（Composite Unique Key）。如下图所示，`customer` 表上定义的 `UNIQUE` 键约束使用了复合唯一键，包含 `area` 和 `phone` 两列，这要求了任意两行中键的值不能重复。用户可以向列中输入空值。但是，如果列上还定义了 `NOT NULL` 约束，则不允许输入空值。

用户可以向 `customer` 表插入任意条记录，但依据上述 `UNIQUE` 键约束的限制，表中各行的区码（Area Code）与电话号码（Telephone Number）的组合不能重复。这能避免因疏忽造成电话号码重复问题。



3.2.7.2.4. 主键约束

主键值规则（Primary Key Value Rule）是定义在某一键 Key（键指一列或一个列集）上的规则，其作用是确保表内的每一数据行都可以由某一个键值唯一地确定。

每个数据库表上最多只能定义一个 `PRIMARY KEY` 约束。构成此约束的列（一列或多列）的值可以作为一行数据的唯一标识符，即每个数据行可以由此主键值命名。

> **说明**
>
> OceanBase 数据库只支持在建表时通过 `CREATE TABLE` 创建主键约束，暂不支持通过 `ALTER TABLE` 追加、删除、修改主键约束。

本文通过以下具体示例来介绍主键约束的主要特性。

```
obclient> CREATE TABLE t1(c1 INT, c2 INT, CONSTRAINT pk_c1_c2 PRIMARY KEY(c1, c2));
Query OK, 0 rows affected

obclient> INSERT INTO t1 VALUES(1, 1);
Query OK, 1 row affected

obclient> INSERT INTO t1 VALUES(1, 1);
ORA-00001: unique constraint '1-1' for key 'PK_C1_C2' violated

obclient> INSERT INTO t1 VALUES(null, 1);
ORA-01400: cannot insert NULL into '(C1)'

obclient> INSERT INTO t1 VALUES(null, null);
ORA-01400: cannot insert NULL into '(C1)'
```

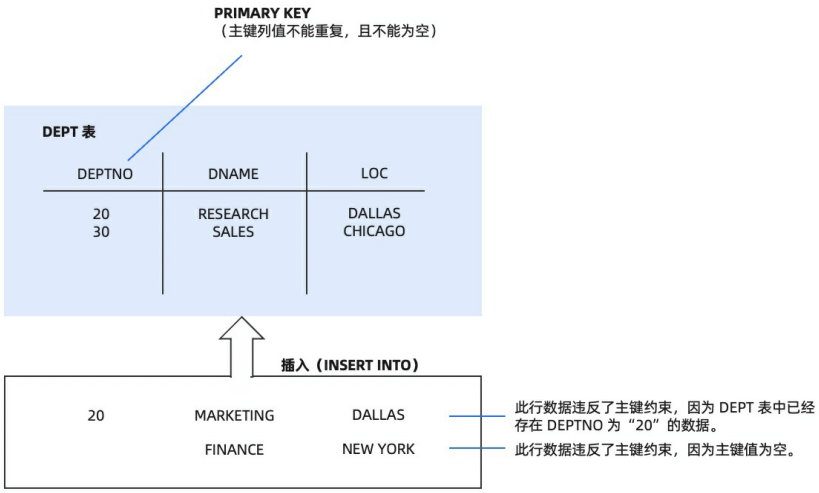
`PRIMARY KEY` 完整性约束（Integrity Constraint）能够确保表的数据遵从以下两个规则：

- 任意两行数据的 `PRIMARY KEY` 约束列（一列或多列）不存在重复值。
- 主键列的值不为空。即用户必须为主键列输入值。

数据库不强制用户为表定义主键，但使用主键可以使表内的每行数据可以被唯一确定，且不存在重复的数据行。

下图展示了定义在 `dept` 表上的 `PRIMARY KEY` 约束，以及违反此约束的数据行。`dept` 表内包含 3 列，分别是 `deptno`、`dname` 和 `loc`。在 `deptno` 列上定义了主键约束，则此列不能有重复数据，且不能为空。

图中还展示了两行因为违反主键约束而无法插入 `dept` 表的数据，一行数据的主键值与已有数据重复，另一行数据的主键列为空值。



3.2.7.2.5. 外键约束

OceanBase 数据库 MySQL 模式支持外键，允许跨表交叉引用相关数据，外键约束有助于保持相关数据的一致性。

外键约束的特性

外键关系涉及一个包含初始列值的父表，以及一个包含引用父列值的列值的子表。在子表上定义了外键约束。

在 `CREATE TABLE` 或者 `ALTER TABLE` 语句中定义外键约束的基本语法如下：

```
[CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (col_name, ...)
    REFERENCES tbl_name (col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]

reference_option:
    RESTRICT | CASCADE | NO ACTION | SET DEFAULT
```

外键约束的相关特性如下：

- 身份标识
- 条件和限制
- 引用行为

身份标识

外键约束命名由以下规则控制：

- 如果 `CONSTRAINT symbol` 已定义，则使用该值。
- 如果 `CONSTRAINT symbol` 子句未定义，或者 `CONSTRAINT` 关键字后面未包含符号，则会自动生成约束名称名称。
- 该值（如果已定义）在数据库中必须是唯一的。重复的值会导致以下内容的错误：`ERROR 1005 (HY000): Can't create table 'test.fk1' (errno: 121)` 。

`FOREIGN KEY ... REFERENCES` 子句中的表和列标识符可以用反引号 (`)引用。

条件和限制

外键约束受以下条件和限制的约束：

- 创建外键约束需要有 `REFERENCES` 父表的权限。
- 外键和引用键中对应的列必须具有相似的数据类型。 `INTEGER` 和 `DECIMAL` 等固定精度类型的大小和符号必须相同。字符串类型的长度不必相同。对于非二进制（字符）字符串列，字符集和排序规则必须相同。
- MySQL 模式下需要外键和引用键的索引，以便外键检查可以快速并且不需要表扫描。在引用表中，必须有一个索引，其中外键列以相同的顺序列为**第**一列。如果引用表不存在，则会在引用表上自动创建此类索引。如果您创建另一个可用于强制执行外键约束的索引，则此索引可能会在稍后被静默删除。如果指定 `index_name` ，则如前所述的方法进行使用。
- 不支持外键列的索引前缀。因此， `BLOB` 和 `TEXT` 列不能被包括在一个外键，因为对这些列的索引必须总是包含一个前缀长度。
- 外键约束不能引用虚拟生成的列。

引用行为

当 `UPDATE` 或 `DELETE` 操作影响在子表中具有匹配行的父表中的键值时，结果取决于 `FOREIGN KEY` 子句的 `ON UPDATE` 和 `ON DELETE` 子句指定的引用操作。引用行为包括：

- `CASCADE` ：从父表中删除或更新行并自动删除或更新子表中的匹配行。

支持 `ON DELETE CASCADE` 和 `ON UPDATE CASCADE` 。在两个表之间，不要定义多个作用于父表或子表中同一列的 `ON UPDATE CASCADE` 子句。 如果在外键关系中的两个表上都定义了 `FOREIGN KEY` 子句，使两个表成为父表和子表，则必须为另一个 `FOREIGN KEY` 子句定义 `ON UPDATE CASCADE` 或 `ON DELETE CASCADE` 子句，以便进行级联操作成功。如果仅为一个 `FOREIGN KEY` 子句定义了 `ON UPDATE CASCADE` 或 `ON DELETE CASCADE` 子句，则级联操作将失败并显示错误。

> **注意**
>
> 级联外键操作不会激活触发器。

- `RESTRICT` ：拒绝父表的删除或更新操作。

  指定 `RESTRICT` (或 `NO ACTION`) 与省略 `ON DELETE` 或 `ON UPDATE` 子句相同。

- `NO ACTION` : 标准 SQL 中的关键字。

  在 OceanBase 数据库 MySQL 模式中，相当于 `RESTRICT` 。如果被引用的表中存在相关的外键值，OceanBase 数据库将拒绝对父表的删除或更新操作。有些数据库系统有延迟检查，并且 `NO ACTION` 是延迟检查。在 MySQL 中，会立即检查外键约束，因此 `NO ACTION` 与 `RESTRICT` 含义相同。

对于未指定的 `ON DELETE` 或 `ON UPDATE` ，默认操作始终为 `NO ACTION` 。默认情况下，显式指定的 `ON DELETE NO ACTION` 或 `ON UPDATE NO ACTION` 子句不会出现在 `SHOW CREATE TABLE` 的输出中。

外键约束的常用操作

添加外键约束

使用如下 `ALTER TABLE` 语法向现有表添加外键约束：

```
ALTER TABLE table_name
    ADD [CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (col_name, ...)
    REFERENCES tbl_name (col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

外键可以是自引用的（指同一个表）。当您使用 `ALTER TABLE` 向表添加外键约束时，请务必首先在外键引用的列上创建索引。

删除外键约束

使用如下 `ALTER TABLE` 语法删除外键约束：

```
ALTER TABLE table_name DROP FOREIGN KEY fk_symbol;
```

如果在创建约束时 `FOREIGN KEY` 子句定义了一个 `CONSTRAINT` 名称，则可以引用该名称来删除外键约束。否则，会在内部生成约束名称，而且您必须使用该值。要确定外键约束名称，请使用 `SHOW CREATE TABLE` 。

外键检查

外键检查由 `foreign_key_checks` 变量控制，该变量默认启用。通常，在正常操作期间启用此变量以强制执行参照完整性。

在以下情况下禁用 `foreign_key_checks` 会对数据库带来正面影响：

- 删除由外键约束引用的表。只有在禁用 `foreign_key_checks` 后才能删除引用的表。删除表时，表上定义的约束也会被删除。
- 以不同于外键关系所需的顺序重新加载表。
- 进行数据导入操作时，可以关闭外键检查，以加速数据导入。
- 对具有外键关系的表执行 `ALTER TABLE` 操作。

同样，禁用 `foreign_key_checks` 也会带来一些负向影响：

- 允许删除包含带有外键的表的数据库，这些表被数据库外的表引用。
- 允许删除带有其他表引用的外键的表。

启用 `foreign_key_checks` 不会触发表数据的扫描，这意味着当 `foreign_key_checks` 重新启用时，在禁用 `foreign_key_checks` 时添加到表中的行不会检查一致性。

外键定义和元数据

要查看外键定义，请使用 `SHOW CREATE TABLE` ，示例如下：

```
obclient> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
Table: child
Create Table: CREATE TABLE `child` (
`id` int(11) DEFAULT NULL,
`parent_id` int(11) DEFAULT NULL,
CONSTRAINT `child_OBFK_1633952161788605` FOREIGN KEY (`parent_id`) REFERENCES `test`.`paren
t`(`id`) ON UPDATE RESTRICT ON DELETE CASCADE ,
KEY `par_ind` (`parent_id`) BLOCK_SIZE 16384 GLOBAL
) DEFAULT CHARSET = utf8mb4 ROW_FORMAT = DYNAMIC COMPRESSION = 'zstd_1.3.8' REPLICA_NUM = 1
BLOCK_SIZE = 16384 USE_BLOOM_FILTER = FALSE TABLET_SIZE = 134217728 PCTFREE = 0
```

可以从 INFORMATION_SCHEMA.KEY_COLUMN_USAGE 表中获取有关外键的信息。查询示例如下：

```
obclient> SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGEWHERE REFERENCED_TABLE_SCHEMA IS NOT NULL;+--------
------+------------+-------------+----------------------------+| TABLE_SCHEMA | TABLE_NAME
| COLUMN_NAME | CONSTRAINT_NAME             |+--------------+------------+-------------+---
-------------------------+| test         | child      | parent_id   | child_OBFK_163395216
1788605 |+--------------+------------+-------------+----------------------------+
```

外键约束示例

- 通过单列外键关联父表和子表。

```
CREATE TABLE parent (
 id INT NOT NULL,
 PRIMARY KEY (id)
)

CREATE TABLE child (
 id INT,
 parent_id INT,
 INDEX par_ind(parent_id),
 FOREIGN KEY (parent_id) REFERENCES parent (id) ON DELETE CASCADE
)
```

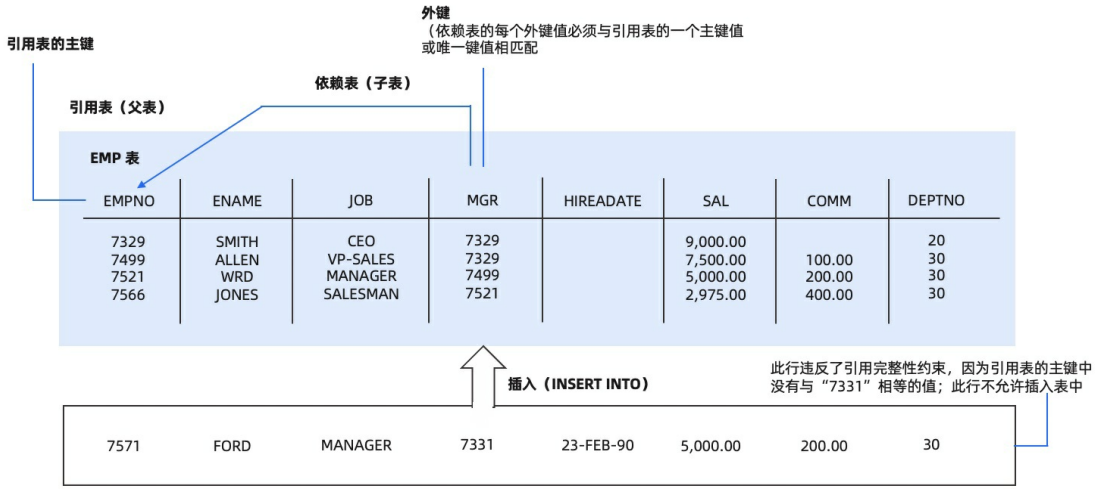
- `product_order` 表具有其他两个表的外键。一个外键引用产品表中的两列索引，另一个引用客户表中的单列索引。

```
CREATE TABLE product (
 category INT NOT NULL,
 id INT NOT NULL,
 price DECIMAL,
 PRIMARY KEY (category, id)
)

CREATE TABLE customer (
 id INT NOT NULL,
 PRIMARY KEY (id)
)

CREATE TABLE product_order (
 no INT NOT NULL AUTO_INCREMENT,
 product_category INT NOT NULL,
 product_id INT NOT NULL,
 customer_id INT NOT NULL,
 PRIMARY KEY (no),
 INDEX(product_category, product_id),
 INDEX(customer_id),
 FOREIGN KEY (product_category, product_id) REFERENCES product (category, id) ON DELETE R
ESTRICT ON UPDATE CASCADE,
 FOREIGN KEY (customer_id) REFERENCES customer (id)
)
```

3.2.7.3. 完整性约束的使用

OceanBase 数据库何时执行约束检查（Checking of Constraint），有助于明确存在各种约束时允许执行的操作类型。

如下图所示，定义 `emp` 表。在 `emp` 表上定义自引用约束（Self-Referential Constraint），`mgr` 列的值依赖于 `empno` 列的值。为了简化示例，以下内容只针对 `emp` 表的 `empno`（`employee_id`）以及 `mgr`（`manager_id`）列。



现在向 `emp` 表插入第一条数据。由于此时表内没有数据，`mgr` 列无法引用 `empno` 列已有的值，可以根据如下场景执行数据插入：

- 如果 `mgr` 列上没有定义 `NOT NULL` 约束，可以在第一行的 `mgr` 列上输入一个空值。由于外键约束允许空值，所以此行能成功插入表中。
- 可以向第一行的 `empno` 及 `mgr` 列输入一个相同的值。此种情况说明是在语句运行完成后执行的约束检查（Constraint Checking）。如果在第一行的父键（Parent Key）及外键（Foreign Key）插入相同的值，必须首先运行语句（即插入数据行），再检查此行数据的 `mgr` 列值是否能与此表内的某个 `empno` 列值相匹配。
- 执行一个多行的 `INSERT` 语句，例如与 `SELECT` 语句结合的 `INSERT` 语句，将插入存在相互引用关系的多行数据。例如，第一行的 `empno` 列值为 200，`mgr` 列值为 300，而第二行的 `empno` 列值为 300，`mgr` 列值为 200。

此种情况也说明数据库会将约束检查延迟直至语句运行结束。所有数据行首先被插入，之后逐行检查是否存在违反约束的情况。

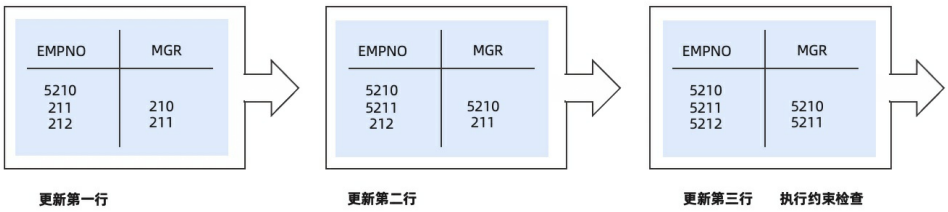
用上述的自引用约束再举一个例子。假设公司被收购，所有员工编号需要被更新为当前值加 5000，以便和新公司的员工编号保持一致。由于经理编号也是员工编号，所以此值也需要加 5000。下图为被更新之前的 `emp` 表，其中包含 `empno` 与 `mgr` 两列。`empno` 列有 3 个值：210、211 和 212。`mgr` 列有两个值：210 和 211。

EMPNO	MGR
210 211 212	210 211

对 `emp` 表执行以下 SQL:

```
UPDATE EMP
SET empno = empno + 5000,
mgr = mgr + 5000;
```

尽管 `emp` 表上定义的约束要求每个 `mgr` 值必须能和一个 `empno` 值相匹配，此语句仍旧可以执行，因为在语句执行后才进行约束检查。下图表明执行了 SQL 语句的全部操作之后才进行约束检查。



首先为每个员工编号加 5000，再为每个经理编号加 5000。在第一步中，`empno` 列的值 210 被更新为 5210。在第二步中，`empno` 列的值 211 被更新为 5211，`mgr` 列的值 210 被更新为 5210。在第三步中，`empno` 列的值 212 被更新为 5212，`mgr` 列的值 211 被更新为 5211。最后执行约束检查。

上述示例说明了 `INSERT` 和 `UPDATE` 语句的约束检查机制。事实上各类 DML 语句的约束检查机制均相同，这些 DML 语句包括 `UPDATE` 、 `INSERT` 以及 `DELETE` 等语句。

4.分布式数据库对象

4.1. 分布式数据库对象概述

OceanBase 数据库是分布式关系型数据库。区别于集中式数据库，OceanBase 数据库的核心特点是分布式能力，即具备极高的可扩展性和高可用性。

OceanBase 数据库采用 shared-nothing 架构，用户数据以分区方式进行分片分布在多台机器上。系统根据分区数、数据量、以及机器负载等因素动态迁移分区数据以达到负载均衡目的。用户可以进行资源调整，实现计划内的扩容和缩容。

在分布式环境下，为保证数据读写服务的高可用，OceanBase 数据库会把同一个分区的数据拷贝到多个机器。不同机器同一个分区的数据拷贝称为副本（Replica）。同一分区的多个副本使用 Paxos 一致性协议保证副本的强一致，每个分区和它的副本构成一个独立的 Paxos 组，其中一个分区为主分区（Leader），其它分区为备分区（Follower）。主分区具备强一致性读和写能力，备分区具备弱一致性读能力。

4.2. 集群架构

基本概念

集群

OceanBase 数据库集群由一个或多个 Region 组成，Region 由一个或多个 Zone 组成，Zone 由一个或多个 OBServer 组成，每个 OBServer 有若干个 Partition 的 Replica。

Region

Region 对应物理上的一个城市或地域，当 OceanBase 数据库集群由多个 Region 组成时，数据库的数据和服务能力就具备地域级容灾能力；当集群只有一个 Region 时，如果出现整个城市级别的故障，则会影响数据库的数据和服务能力。

Zone

Zone 一般情况（不考虑机房级容灾可部署一中心三副本）下对应一个有独立网络和供电容灾能力的数据中心，在一个 Region 内的多个 Zone 间 OceanBase 数据库集群拥有 Zone 故障时的容灾能力。

OBServer

运行 observer 进程的物理机。一台物理机上可以部署一个或者多个 OBServer（通常情况下一台物理机只部署一个 OBServer）。在 OceanBase 数据库内部，Server 由其 IP 地址和服务端口唯一标识。

Partition

OceanBase 数据库以分区（Partition）为单位组织用户数据，分区在不同机器上的数据拷贝称为副本（Replica）。同一分区的多个副本使用 Paxos 一致性协议保证副本的强一致，每个分区和它的副本构成一个独立的 Paxos 组，其中一个分区为主分区（Leader），其它分区为备分区（Follower）。主分区具备强一致性读和写能力，备分区具备弱一致性读能力。

部署模式

为保证单一机器故障时同一分区的多数派副本可用，OceanBase 数据库会保证同一个分区的多个副本不调度在同一台机器上。由于同一个分区的副本分布在不同的 Zone/Region 下，在城市级灾难或者数据中心故障时既保证了数据的可靠性，又保证了数据库服务的可用性，达到可靠性与可用性的平衡。OceanBase 数据库创新的容灾能力有三地五中心可以无损容忍城市级灾难，以及同城三中心可以无损容忍数据中心级故障。

三地五中心部署

同城三中心部署

OceanBase 数据库的无损容灾能力还可以方便集群的运维操作，当数据中心或者服务器需要替换和维修时，可以直接下线对应的数据中心或服务器进行替换和维修，并补充进新的数据中心或服务器，OceanBase 数据库会自动进行分区副本的复制和均衡，整个过程可以保证数据库服务的使用不受影响。

RootService

OceanBase 数据库集群会有一个总控服务（RootService），其运行在某个 OBServer上。当 RootService 所在机器故障时，其余 OBServer 会选举出来新的 RootService。RootService 主要提供资源管理、负载均衡、schema 管理等功能，其中：

- 资源管理

  包括 Region/Zone/OBServer/Resource Pool/Unit 等元信息的管理，比如：上下线 OBServer、改变 Tenant 资源规格等。

- 负载均衡

  决定 Unit/Partition 在多个机器间的分布，均衡机器上主分区个数，在容灾场景下通过自动复制/迁移补充缺失的 Replica。

- Schema 管理

  负责处理 DDL 请求并生成新 Schema。

Locality

租户下分区在各个 Zone 上的副本分布和类型称为 Locality。我们可以通过建租户指定 Locality 的方式决定租户下分区初始的副本类型和分布，后续可通过改变租户 Locality 的方式进行修改。

下边的语句表示创建 `mysql_tenant` 租户，并且其租户下的分区在 `z1` 、 `z2` 、 `z3` 上都是全能型副本。

```
obclient> CREATE TENANT mysql_tenant RESOURCE_POOL_LIST =('resource_pool_1'), primary_zone
= "z1;z2;z3", locality ="F@z1, F@z2, F@z3" setob_tcp_invited_nodes='%',ob_timestamp_service
='GTS';
```

下边的语句表示变更 mysql_tenant 的 locality，使其租户下的分区在 z1、z2 是全能型副本，z3 是日志型副本。OceanBase 数据库会基于租户新旧 locality 的对比，决定是否创建/删除/转换对应 zone 的副本。

```
ALTER TENANT mysql_tenant set locality = "F@z1, F@z2, L@z3";
```

- 同一个 OBServer 不会有同一个分区一个以上的副本。
- 对于一个分区而言，其在一个 Zone 内最多存在一个 Paxos 副本，可以有若干个非 Paxos 副本。可以在 Locality 中指定非 Paxos 副本，如：
  - `locality = "F{1}@z1, R{2}@z1"` ：表示 z1 有 1 个全能型副本，2 个只读型副本。
  - `locality = "F{1}@z1, R{ALL_SERVER}@z1"` ：表示 z1 有一个全能型副本，并在同 zone 其余机器上创建只读型副本(可以没有只读型副本)。

RootService 会根据用户设置的 Locality，通过副本创建/删除/迁移/类型转换的方式，使分区的副本分布和类型满足用户配置的 Locality。

Primary Zone

用户可通过一个租户级的配置，使租户下分区 Leader 分布在指定的 Zone 上，此时称 Leader 所在的 Zone 为 Primary Zone。

Primary Zone 是一个 Zone 集合，用分号（;）分割表示不同的优先级，用逗号（,）分隔表示相同的优先级。RootService 会根据用户设置的 Primary Zone，按照优先级高低顺序，尽可能把分区 Leader 调度到更高优先级的 Zone 内，并在同一优先级的 Zone 间将 Leader 打散在不同的机器上。不设置 Primary Zone 的场合，会认为租户的所有 Zone 都是同一优先级，RootService 会把租户分区 Leader 打散在所有 Zone 内的机器。

用户可通过租户级配置，设置或修改租户的 Primary Zone。

示例如下：

- 租户创建时设置 Primary Zone，优先级 `z1` = `z2` > `z3` 。

```
obclient> CREATE TENANT mysql_tenant RESOURCE_POOL_LIST =('resource_pool_1'), primary_zon
e = "z1,z2;z3", locality ="F@z1, F@z2, F@z3" setob_tcp_invited_nodes='%',ob_timestamp_ser
vice='GTS';
```

- 变更租户 Primary Zone，优先级 `z1` > `z2` > `z3` 。

```
obclient> ALTER TENANT mysql_tenant set primary_zone ="z1;z2;z3";
```

- 变更租户 Primary Zone，优先级 `z1` = `z2` = `z3` 。

```
obclient> ALTER TENANT mysql_tenant set primary_zone =RANDOM;
```

> **说明**
>
> Primary Zone 只是其中一种选主参考因素，分区对应 Zone 的副本是否能成为 Leader 还需要参考副本类型、日志同步进度等因素。

4.3. 数据分区和分区副本

4.3.1. 数据分区和分区副本概述

OceanBase 数据库参考传统数据库分区表的概念，把一张表格的数据划分成不同的分区（Partition）。在分布式环境下，为保证数据读写服务的高可用，OceanBase 数据库会把同一个分区的数据拷贝到多个机器。不同机器同一个分区的数据拷贝称为副本（Replica）。同一分区的多个副本使用 Paxos 一致性协议保证副本的强一致，每个分区和它的副本构成一个独立的 Paxos 组，其中一个分区为主分区（Leader），其它分区为备分区（Follower）。主分区具备强一致性读和写能力，备分区具备弱一致性读能力。

Location Cache

OceanBase 数据库按分区组织用户数据，且每个分区有多个副本用于容灾。因此，在 SQL 请求执行过程中需要知道分区的位置信息，用于路由到特定机器读写对应分区副本的数据。分区的位置信息即称为 Location，每个 observer 进程会有一个服务，用于刷新及缓存本机需要的分区 Location 信息，该服务称为 Location Cache 服务。

分区的 Location 信息持久化到了 OceanBase 数据库内置的表中，持久化 Location 的内置表称为 Meta 表。为解决集群自举的问题，不同类型的表的 Location 信息是按层次组织的，不同类型的表的 Location 会持久化到不同的 Meta 表中。各级 Meta 表所记录的内容如下：

- `__all_virtual_core_root_table`：记录 `__all_root_table` 的 Location。
- `__all_root_table`：记录集群中所有内置表的 Location。
- `__all_virtual_meta_table`：记录集群中所有租户的用户创建的表的分区的 Location。

副本类型

在 OceanBase 数据库中，为了数据安全和提供高可用的数据服务，每个分区数据在物理上存储多份，每一份叫做分区的一个副本。每个副本包含了存储在磁盘上的静态数据（SSTable）、存储在内存的增量数据（MemTable）以及记录事务的日志等三类主要数据。根据存储数据种类的不同，副本有几种不同的类型，以支持不同业务在数据安全、性能伸缩性、可用性、成本等之间的选择。

当前，OceanBase 数据库支持以下四种类型的副本：

- 全能型副本（FULL/F）
- 日志型副本（LOGONLY/L）
- 加密投票型副本（ENCRYPTVOTE/E）
- 只读型副本（READONLY/R）

全能型、日志型或加密投票型副本又称为 Paxos 副本，对应的副本可构成 Paxos 成员组；而只读型副本又称为非 Paxos 副本，对应的副本不可构成 Paxos 成员组。

分布式一致性协议

OceanBase 数据库使用 Paxos 协议在同一分区各副本间同步事务日志，多数派同步成功才能提交。缺省情况下读、写操作在主分区上保证强一致；备副本支持弱一致性读，允许读取某一个稍旧版本的数据。

数据均衡

OceanBase 数据库通过 RootService 管理租户内各个资源单元间的负载均衡。不同类型的副本对资源的需求各不相同，RootService 在执行分区管理操作时需要考虑的因素包括每个资源单元的 CPU、磁盘使用量、内存使用量以及 IOPS 使用情况。经过负载均衡，最终会使得所有机器的各类型资源占用都处于一种比较均衡的状态，充分利用每台机器的所有资源。

RootService 主要通过以下两种方式实现数据均衡：

- 副本均衡

  RootService 会通过 Unit 迁移、副本复制或迁移的方式，调整租户在各机器上的资源占用情况。

- Leader 均衡

  在副本均衡的基础上，RootService 会根据租户的 Primary Zone 等因素，均衡各机器分区 Leader 的数目。具体表现为，通过将分区 Leader 聚集到同一机器上，减少分布式事务执行的可能，减少业务请求的 RT（Response Time），或者，通过将分区 Leader 在多机上打散，最大限度地利用机器资源，提高系统吞吐能力。

4.3.2. 分区副本类型

4.3.2.1. 分区副本概述

为了数据安全和提供高可用的数据服务，每个分区数据在物理上存储多份，每一份叫做分区的一个副本。每个副本，包括存储在磁盘上的静态数据（SSTable）、存储在内存的增量数据（MEMTable）、以及记录事务的日志三类主要的数据。根据存储数据种类的不同，副本有几种不同的类型，以支持不同业务在在数据安全、性能伸缩性、可用性、成本等之间的选择。

目前 OceanBase 数据库支持以下四种类型的副本：

- 全能型副本（FULL/F）
- 日志型副本（LOGONLY/L）
- 加密投票型副本（ENCRYPTVOTE/E）
- 只读型副本（READONLY/R）

我们把全能型、日志型、加密投票型副本称为 Paxos 副本，对应副本可构成 Paxos 成员组；只读型副本称为非 Paxos 副本，对应副本不可构成 Paxos 成员组。

副本类型属性

目前，指定副本类型和位置分布的 locality 一般语法如下：

```
locality='F@z1,F@z2,F{1},R{ALL_SERVER}@z3,L@z4,L@z5'
```

在该语法中，允许在副本类型之后的花括弧内限定副本的其他属性。花括弧内的数字表示对应 zone 对应副本类型的副本个数，其中：

- 对于 F、L、E 型副本：该值只能为 1，或者不指定（不指定视作 1）。
- 对于 R 型副本：该值上限为租户对应 zone 的可用 unit 数目，或者指定为 ALL_SERVER（表示尽可能在对应 zone 创建 R 副本），或者不指定（不指定视作 1）。

为了限定副本的 memtable 内存属性，引入一个属性名为 memstore_percent，表示该**副本 memstore 占用内存上限与 leader 占用内存上限的比值**。目前，memstore_percent 属性可以**作用于 F、R 型副本**，语法示例如下：

```
locality='F@z1,F{memstore_percent:0}@z2,F{1,memstore_percent:100},R{ALL_SERVER}@z3,L@z4,L@z
5'
```

目前该属性只有 0 和 100 两种取值：

- 对于不给定 memstore_percent 属性的副本，等价于 `memstore_percent=100`，代表不限制 mestore 内存使用。
- 特别的，对于 `memstore_percent=0` 的副本，这个副本没有 memstore，不做日志回放（节省了回放日志和转储的 CPU 开销）；该副本需要定期从其他副本拉取转储数据，以保障宕机后的恢复时间可控。

对于设置了 `memstore_percent=0` 属性的副本，我们称之为 **D 副本（Data Replica/D）**。对于 D 副本：

- F 型的 D 副本作为新主上任时，需要更长（分钟级）的宕机自动恢复时间。
- D 副本的 memtable 内存占用为 0，不回放日志。

对资源受限的业务场景，若其对宕机恢复时间不敏感，通过设置 D 副本可以显著减少内存资源占用。

4.3.2.2. 全能型副本

全能型副本是目前最广泛使用的副本类型，它拥有事务日志、MEMTable 和 SSTable 等全部完整的数据和功能。

全能型副本具备以下特点：

- 是目前最广泛使用的副本类型，它拥有事务日志、MEMTable 和 SSTable 等全部完整的数据和功能。
- 可以随时快速切换为 Leader 对外提供服务。
- 可以构成 Paxos 成员组，并且要求 Paxos 成员组多数派必须为全能型副本。
- 可以转换为除加密投票型副本以外的任意副本类型。

更多特性及其说明如下表所示。

特性项	描述
副本名称及缩写	FULL(F)
是否有 Log	有，参与投票(SYNC_CLOG)
是否有 MEMTable	有(WITH_MEMSTORE)
是否有 SSTable	有(WITH_SSSTORE)
数据安全	高
恢复为 Leader 的时间	快
资源成本	高

特性项	描述
服务	Leader 提供读写，Follower 可非一致性读
副本类型转换限制	可转换为除加密投票型副本以外的任意副本类型

4.3.2.3. 日志型副本

日志型副本仅包含日志的副本，没有 MEMTable 和 SSTable。

日志型副本主要具备以下特点：

- 仅包含日志的副本，没有 MEMTable 和 SSTable，对内存和磁盘占用最少。
- 参与日志投票并对外提供日志服务，可以参与其他副本的恢复。
- 不能变为主提供数据库服务。
- 可构成 Paxos 成员组。
- 无法转换为其他副本类型。

更多特性及其说明如下表所示。

特性项	说明
副本名称及缩写	LOGONLY(L)
是否有 Log	有，参与投票(SYNC_CLOG)
是否有 MEMTable	无(WITHOUT_MEMSTORE)
是否有 SSTable	无(WITHOUT_SSSTORE)
数据安全	低
恢复为 Leader 的时间	不支持
资源成本	低
服务	不可读写
副本类型转换限制	无法与任何类型的副本转换

4.3.2.4. 加密投票型副本

加密投票型副本本质上是加密后的日志型副本，没有 MEMTable 和 SSTable。

加密投票型副本主要具备以下特点：

- 本质上是加密后的日志型副本，没有 MEMTable 和 SSTable。
- 参与日志投票并对外提供日志服务，可以参与其他副本的恢复。
- 不能变为主提供数据库服务。
- 可构成 Paxos 成员组。
- 无法转换为其他副本类型。

更多特性及其说明如下表所示。

特性项	描述
副本名称及缩写	ENCRYPTVOTE（E）
是否有 Log	有，参与投票（STNC_CLOG）
是否有 MEMTable	无（WITHOUT_MEMSTORE）
是否有 SSTable	无（WITHOUT_SSSTORE）
数据安全	高
恢复为 Leader 的时间	不支持
资源成本	低
服务	不可读写
副本类型转换限制	无法与任何类型的副本转换

4.3.2.5. 只读型副本

只读型副本包含完整的日志、MEMTable 和 SSTable 等。

只读型副本主要具备以下特点：

- 包含完整的日志、MEMTable 和 SSTable 等。

- 不可构成 Paxos 成员组，它不作为 Paxos 成员参与日志的投票，而是作为一个观察者实时追赶 Paxos 成员的日志，并在本地回放，故不会造成投票成员增加导致事务提交延时的增加。
- 在业务对读取数据的一致性要求不高的时候可提供只读服务。
- 可转换成全能型副本。

更多特性及其说明如下表所示。

特性项	描述
副本名称及缩写	READONLY(R)
是否有 LOG	有，是异步日志，但不属于 Paxos 组，只是 Listener (ASYNC_CLOG)
是否有 MEMTable	有(WITH_MEMSTORE)
是否有 SSTable	有(WITH_SSSTORE)
数据安全	中
恢复为 Leader 的时间	不支持
资源成本	高
服务	可非一致性读
副本类型转换限制	只能转换成全能型副本

4.3.3. 多副本一致性协议

本节主要介绍 OceanBase 数据库的两种一致性协议及各协议的作用。

在 OceanBase 数据库中，同一行数据在多个节点上存储和提供服务。数据包括已经持久化到磁盘的数据，也包含尚在内存中的数据。如何维护这些数据副本之间的一致性，依靠的就是“一致性协议”。OceanBase 数据库有两种一致性协议：

- 基于 Multi-Paxos 的分布式一致性协议，它在主副本故障时能保证备副本的数据无损恢复。它是 OceanBase 数据库高可用能力的基础。更多详细信息请参见 Paxos协议。
- 基于主从复制（异步复制）的一致性协议，它在主副本故障是能保证备副本的数据有损恢复。它是只读副本和主备库的基础。更多详细信息请参见 分布式选举。

上述协议都是通过复制事务 REDO 日志的方式实现的，更多事务 REDO 日志的信息请参见 多副本日志同步。

4.3.4. 数据均衡

4.3.4.1. 分区副本均衡

4.3.4.1.1. 自动负载均衡

分区副本的自动负载均衡是指在租户拥有的 Unit 内调整分区副本的分布使得 Unit 的负载差值尽量小。分区副本的自动负载均衡是租户级别的行为，发生在单个可用区（Zone）内。即 RootService 调度某个租户的数据副本在 Zone 内发生迁移，达到该租户在该 Zone 上的全部 Unit 的负载均衡。

均衡组

分区副本均衡的目标是将某个租户在单个可用区（Zone）上的全部副本（Partition）均衡调度到该 Zone 的全部 Unit 上。在具体的分区副本均衡中，系统将该 Zone 内的全部 Partition 划分成若干个组，每个组作为均衡调度的一个基本单位，这样的组就称为一个均衡组。均衡组是一组 Partition 的集合。各均衡组之间相互独立，当每个均衡组内的 Partition 在组内达到负载均衡时，系统认为该 Zone 内的分区副本达到整体负载均衡。

OceanBase 数据库中目前共划分了以下 3 类均衡组：

- 第一类均衡组：包含多个分区的一个 Table Group下的所有 Partition 被认定为一个均衡组。
- 第二类均衡组：不属于任何 Table Group 的某个多分区表下的全部 Partition 被认定为一个均衡组。
- 第三类均衡组：除上述 Partition 以外的全部其他 Partition 被认定为一个均衡组。针对一个租户，此类均衡组在某个 Zone 内仅有一个。

均衡规则

在 OceanBase 数据库中，通过配置项 `balancer_tolerance_percentage` 来指定磁盘均衡的灵敏度参数，该配置项的值是一个百分数，取值范围为 [1, 100]。

各类均衡组结合配置项 `balancer_tolerance_percentage` 所遵循的均衡规则如下：

- 第一类均衡组

  此类均衡组中的全部 Partition 来自于一个 Table Group，这类均衡组的均衡目标是，将 Partition Group 按照个数平均调度到 Zone 内的全部 Unit 上，使得各 Unit 上的 Partition Group 的数量差值最大不超过 1。在 Partition Group 个数满足要求的前提下，通过在 Unit 间交换 Partition Group，使各 Unit 磁盘使用量的差值小于配置项 `balancer_tolerance_percentage` 设置的值。

  有关 Partition Group 的更多信息请参见 Table Group。

- 第二类均衡组

  此类均衡组中的全部 Partition 来自于一个多分区表。这类均衡组的均衡目标是，将该多分区表的全部 Partition 按照个数平均调度到 Zone 内的全部 Unit 上，使得各 Unit 上的 Partition 数量差值最大不超过 1。在 Partition 个数满足要求的前提下，通过在 Unit 间交换 Partition，使各 Unit 磁盘使用量的差值小于配置项 `balancer_tolerance_percentage` 设置的值。

- 第三类均衡组

此类均衡组是除第一类和第二类均衡组以外的其他全部 Partition，系统将这些 Partition 按照个数调度到 Zone 内的全部 Unit 上，使得各 Unit 上的 Partition 数量差值最大不超过 1。在 Partition 个数满足要求的前提下，通过在 Unit 间交换 Partition，使各 Unit 磁盘使用量的差值小于配置项 `balancer_tolerance_percentage` 的值。

4.3.4.1.2. Table Group

OceanBase 数据库支持通过设置 Table Group 来配置数据的分布。

什么是 Table Group

Table Group是一个逻辑概念，它描述的是 Table 的一个集合。属于该集合的所有 Table，均需要满足如下约束，即集合中的所有 Table 必须拥有相同的 Locality（包括副本类型、个数及位置）、相同的 Primary Zone（包括 Leader 位置及其优先级）和相同的分区方式。属于同一个 Table Group 的 Table，每个 Table 都拥有相同数量的分区（Partition）。

什么是 Partition Group

假设存在一个 Table Group `TG0`，`TG0` 中包含 k 个 Table，名称依次为 `T1`、`T2` ... `Tk`。假设每个 Table 都包含 m 个分区，给这些分区标号如下：

- `T1` 表的全部分区：P11, P12, P13, P14 ... P1m
- `T2` 表的全部分区：P21, P22, P23, P24 ... P2m
- `T3` 表的全部分区：P31, P32, P33, P34 ... P3m
- ...
- `Tk` 表的全部分区：Pk1, Pk2, Pk3, Pk4...Pkm

将偏移量相同的一组分区称为一个 Partition Group，则 `TG0` 中包含 m 个 Partition Group，将这些 Partition Group 的名称分别命名为 `pg1`、`pg2`、`pg3` ... `pgm`，则各 Partition Group 包含的分区如下：

- `pg1` 包含的分区：P11、P21、P31、P41 ... Pk1
- `pg2` 包含的分区：P12、P22、P32、P42 ... Pk2
- `pg3` 包含的分区：P13、P23、P33、P43 ... Pk3
- ...
- `pgm` 包含的分区：P1m、P2m、P3m、P4m ... Pkm

Partition Group 的使用

根据上面的示例，Table Group `TG0` 中共包含 `pg1` 、 `pg2` 、 `pg3` ... pgm 等 m 个 Partition Group。OceanBase 数据库假设，处于同一个 Partition Group 的多个分区有较大概率在同一个事务中被修改。为使同一个事务的修改尽量发生在同一个 OBServer 上，减少分布式事务发生的概率，RootService 会将属于同一个 Partition Group 的分区尽量调度到同一个 OBServer 上去，即：

- `pg1` 的 k 个分区 P11、P21、P31、P41 ... Pk1 会被尽量调度到同一个 OBServer 上。
- `pg2` 的 k 个分区 P12、 P22、P32、P42 ... Pk2 会被尽量调度到同一个 OBServer 上。
- `pg3` 的 k 个分区 P13、P23、P33、P43 ... Pk3 会被尽量调度到同一个 OBServer 上。
- ...
- `pgm` 的 k 个分区 P1m、P2m、P3m、P4m ... Pkm 会被尽量调度到同一个 OBServer 上。

4.3.4.2. Leader 均衡

4.3.4.2.1. 自动负载均衡

前面的章节中介绍了均衡组的含义，同时通过个数均衡和磁盘均衡的手段，OceanBase 数据库实现了均衡组内的副本均衡。在分区副本均衡的基础之上，OceanBase 数据库还实现了 Leader 维度的均衡。

Leader 均衡仍然以均衡组为基本均衡单元，旨在将一个均衡组的所有分区的 Leader 平均调度到该均衡组 Primary Zone 的全部 OBServer上，使得 Primary Zone 的各 OBServer 的 Leader 数量差值不超过 1，进而将该均衡组的 Leader 写入负载平均分配到 Primary Zone 的全部 OBServer 上。有关 Primary Zone 的详细介绍信息请参见 Primary Zone。

> **说明**
>
> 有关均衡组的详细介绍信息请参见 **分区副本均衡** 中的 自动负载均衡。

均衡效果

本节以一个简单示例来介绍 Leader 的自动均衡效果。

示例背景

假设系统中存在一个集群包含 3 个可用区 `Zone1` 、 `Zone2` 、 `Zone3` 。每个 Zone 部署 2 台 OBServer，有 一个均衡组包含 12 个 Partition，12 个 Partition 在各 Zone 内的副本分布已经均衡。

下面通过三个场景介绍在不同的 Primary Zone 配置下，Leader 均衡的效果。

场景一

Primary Zone 配置为 `PrimaryZone='Zone1'` ，Leader 平铺到 `Zone1` 的全部 OBServer，Leader 均衡后的效果如下图所示。

从图中可知，全部 12 个 Partition 的 Leader 都分布在 `Zone1` 上，`Zone1` 的每个 OBServer 上各有 6 个 Leader。

场景二

Primary Zone 配置为 `PrimaryZone='Zone1, Zone2'`，Leader 平铺到 `Zone1` 和 `Zone2` 的全部 OBServer，Leader 均衡后的效果如下图所示。

从图中可知，全部 12 个 Partition 的 Leader 平均分布到 `Zone1` 和 `Zone2` 上，每个 OBServer 上各有 3 个 Leader。

场景三

Primary Zone 配置为 `PrimaryZone='Zone1, Zone2, Zone3'`，Leader 平铺到 `Zone1`、`Zone2` 和 `Zone3` 的全部 OBServer，Leader 均衡后的效果如下图所示。

从图中可知，全部 12 个 Partition 的 Leader 平均分布到 `Zone1` 、 `Zone2` 和 `Zone3` 上，每个 OBServer 上各有 2 个 Leader。

4.3.4.2.2. Primary Zone

OceanBase 数据库支持通过设置 Primary Zone 来设置 Leader 副本的偏好。

基本概念

Primary Zone 表示 Leader 副本的偏好位置。指定 Primary Zone 实际上是指定了 Leader 更趋向于被调度到哪个 Zone 上，即假设某张表 `t1` 的 `primary_zone="zone1"` ，则 RS 会尽量将 `t1` 表的 Leader 调度到 `Zone1` 上来。

引申概念

Primary Zone 实际上是一个 Zone 的列表，列表中包含多个 Zone。该列表用使用如下方式为 Zone 配置了优先级。

当 Primary Zone 列表包含多个 Zone 时，用 `;` 分隔的具有从高到底的优先级； `,` 分隔的具有相同优先级。例如： `'hz1,hz2;sh1,sh2;sz1'` 表示 `hz1` 和 `hz2` 具有相同的优先级，并且优先级高于 `sh1` / `sh2` 和 `sz1` ； `sh1` 和 `sh2` 具有相同优先级，并且优先级高于 `sz1` 。

Region 信息

在 OceanBase 数据库中，Zone 有一个 Region 属性，表示该 Zone 所处的地区，每个 Zone 仅能配置一个 Region，但一个 Region 内可包含多个 Zone。Primary Zone 的设置隐含的包含了 Leader 偏好的 Region 位置。具体指用户设置 primary_zone 包含两层语义：

- 被指定的 Primary Zone 为 Leader 的偏好 Zone 的位置。
- 被指定 Primary Zone 所在的 Region 为 Leader 偏好的 Region。

具体地，Leader 会被优先调度到最高优先级的 Zone 上去，如果最高优先级的 Zone 上的副本不能成为 Leader，会优先选择同一个 Region 内的其他 Zone 作为 Leader 的位置。

Primary Zone 改写

用户设置的 Primary Zone 会由 OceanBase 数据库内部，基于各个 Zone 所在的 Region 进行改写，改写规则如下：

- 将用户设置的 Primary Zone 中的所有 Zone 对应的 Region 列出。例如：primary_zone 为 `'hz1,hz2;sh1,sh2;sz1'` 对应的 primary_region 的列表为 `'hz,hz;sh,sh;sz'` 。
- 将 primary_region 中重复的 Region 去掉，去掉规则为保留第一个出现的 Region，其他 Region 后续重复的 Region 移除，则上述 primary_region 转化为 primary_region 列表 `'hz;sh;sz'` 。
- 依据 primary_region 中各 Region 的优先级，对 primary_zone 进行补充，补充规则如下：将 Primary Zone 中各 Region 对应的 Zone 都取出，重新排列，高优先级 Region 内 Zone 比低优先级 Region 内 Zone 的优先级高，同一 Region 内 Zone 的优先级高低参考原始 Primary Zone 中的优先级。

继承关系

当前有 Table 级、Table Group 级、Database 级（MySQL 模式）/Schema级（Oracle 模式）以及 Tenant 级 primary_zone。

除 Tenant 级别外，每个级别均可以自由指定 primary_zone 的分布情况，如果不指定那么默认向上继承，tenant 维度必须指定 primary_zone（此处的必须指定含义是租户级的 primary_zone 不能为空，如果创建时未指定 primary_zone，默认填写为 RANDOM，表示各个 Zone 优先级相同）。

- Table 级
  - 首先看自身的 primary_zone，若 不为空，则使用自身的 primary_zone。
  - 若为空，判断是否属于 Table Group。若属于则查看 Table Group 的 primary_zone 是否为空，若不为空，则使用 Table Group 的 primary_zone。
  - 若 Table Group 为空或不属于 Table Group，那么查看 Database（MySQL模式）/Schema（Oracle 模式）的 primary_zone 是否为空，若不为空，则使用 Database（MySQL 模式)/Schema（Oracle 模式）primary_zone。
  - 若均为空，那么使用 Tenant 级的 primary_zone。
- Table Group 级
  - 检查自身 primary_zone，若不为空，则使用自身的 primary_zone。
  - 若为空，则使用 Tenant 的 primary_zone。
- Database 级（MySQL 模式）/Schema 级（Oracle 模式）
  - 检查自身 primary_zone，若不为空，则使用自身的 primary_zone。
  - 若为空，则使用 Tenant 的 primary_zone。

示例

假设共有 9 个 Zone，`sh1` 、`sh2` 、`sh3` 三个 Zone 在 Region `SH` ，`hz1` 、`hz2` 、`hz3` 三个 Zone 在 Region `HZ` ，`sz1` 、`sz2` 、`sz3` 三个 Zone 在 Region `SZ` 。

示例 1

用户设置的 primary_zone 为 `'sh1;hz1;hz2;sz1';` 按照改写规则 1 得到 primary_region 为 `'SH;HZ;HZ;SZ'` 。按照改写规则 2 得到 primary_zone 为 `'SH;HZ;SZ'` 。按规则 3 得到改写后 primary_zone 为 `'sh1;sh2,sh3;hz1;hz2;hz3;sz1;sz2,sz3'` 。

解释如下：

三个 Region 的优先级为 `SH` > `HZ` > `SZ` 。Region `SH` 中的 Zone 的优先级高于 Region `HZ` 和 Region `SZ` 中 Zone 的优先级。Region `HZ` 中的 Zone 的优先级高于 Region `SZ` 中 Zone 的优先级。各 Region 内每个 Zone 的优先级为：在 Region `SH` 中 `sh1` > `sh2` = `sh3` ，在 Region `HZ` 中 `hz1` > `hz2` > `hz3` ，在 Region SZ 中 `sz1` > `sz2` = `sz3` 。因此最终得到新的 Primary Zone 为 `'sh1;sh2,sh3;hz1;hz2;hz3;sz1;sz2,sz3'` 。Leader 会优先分布在 `sh1` 上，当 `sh1` 发生故障时，Leader 会依照上面的 Primary Zone 优先级依次分布在 `sh2` 和 `sh3` 。

示例 2

用户设置的 primary_zone 为 `'sh1,sh2;hz1;hz2;sz1';` 按照改写规则 1 得到 primary_region 为 `'SH,SH;HZ;HZ;SZ'` 。按照改写规则 2 得到 primary_zone 为 `'SH;HZ;SZ'` 。按规则 3 得到改写后 primary_zone 为 `'sh1,sh2;sh3;hz1;hz2;hz3;sz1;sz2,sz3'` 。

解释如下：

三个 Region 的优先级为 `SH` > `HZ` > `SZ` 。Region `SH` 中的 Zone 的优先级高于 Region `HZ` 和 Region `SZ` 中 Zone 的优先级。Region `HZ` 中的 Zone 的优先级高于 Region `SZ` 中 Zone 的优先级。各 Region 内每个 Zone的优先级为：在 Region `SH` 中 `sh1` = `sh2` > `sh3` ，在 Region `HZ` 中 `hz1` > `hz2` > `hz3` ，在 Region `SZ` 中 `sz1` > `sz2` = `sz3` 。因此最终得到新的 Primary Zone 为 `'sh1,sh2;sh3;hz1;hz2;hz3;sz1;sz2,sz3'` 。Leader 会优先平均分布在 `sh1` 和 `sh2` 上，当 `sh1` 和 `sh2` 发生故障时，Leader 会依照上面 Primary Zone 优先级依次分布在 `sh3` 。

示例 3

用户设置的 primary_zone 为 `'sh1,hz1;hz2;sz1';` 按照改写规则 1 得到 primary_region 为 `'SH,HZ;HZ;SZ'` 。按照改写规则 2 得到 primary_zone 为 `'SH,HZ;SZ'` 。按规则 3 得到改写后 primary_zone 为 `'sh1,hz1;hz2;sh2,sh3,hz3;sz1;sz2,sz3'` 。

解释如下：

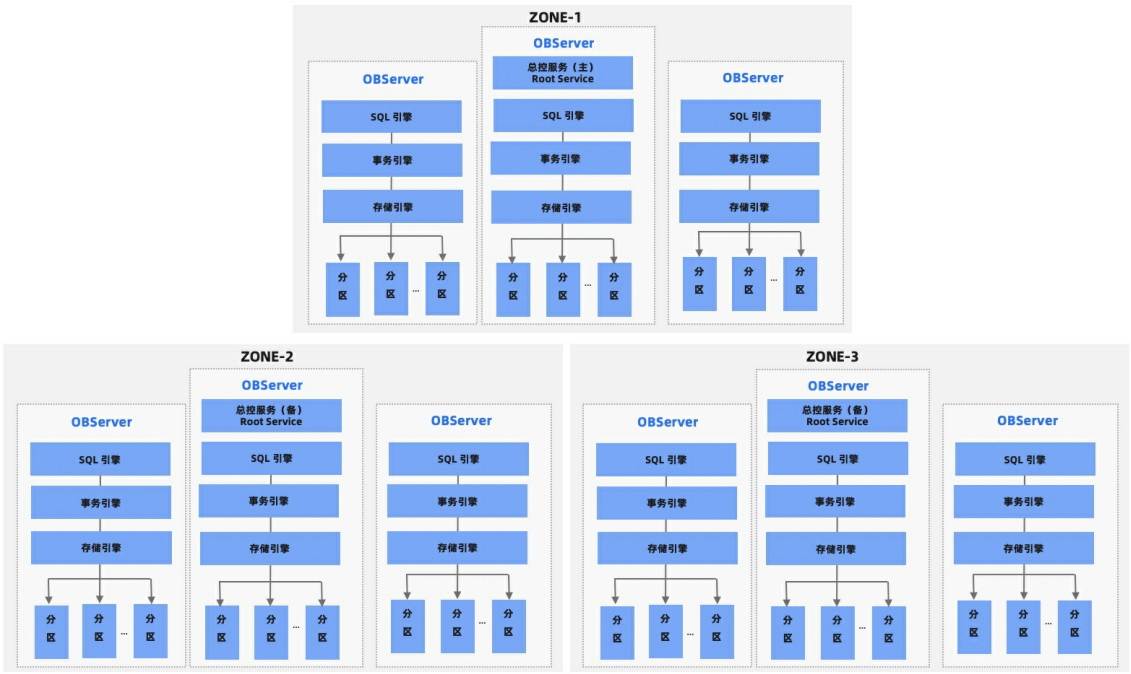
三个 Region 的优先级为 `SH` = `HZ` > `SZ` 。Region `SH` 和 Region `HZ` 中的 Zone 的优先级高于 Region SZ 中 Zone 的优先级。`sh1` = `hz1` > `hz2` > `sh2` = `sh3` = `hz3` > `sz1` > `sz2` = `sz3` 。因此最终得到新的 Primary Zone 为 `'sh1,hz1;hz2;sh2,sh3,hz3;sz1;sz2,sz3'` 。Leader 会优先平均分布在 `sh1` 和 `hz1` 上，当 `sh1` 和 `hz1` 发生故障时，Leader 会依照上面 Primary Zone 优先级依次分布在 `hz2` 。

4.4. 动态扩容和缩容

4.4.1. 集群级别的扩容和缩容

基于分布式架构的 OceanBase 数据库提供灵活的在线扩展性。在集群持续可用的前提下，提供在线扩缩容。

OceanBase 数据库是分布式的数据库系统，通常由多个可用区（Zone）组成，每个可用区（Zone）内包含若干多台物理服务器（OBServer）。OceanBase 数据库的整体结构入下图所示。



与传统单机数据库相比，基于分布式架构的 OceanBase 数据库提供灵活的在线扩展性。在集群持续可用的前提下，提供在线扩缩容。当集群的容灾需求发生变化时，可通过调整可用区数量（加/减 Zone）的方式提高或降低集群的容灾能力。当集群的外部负载发生变化时，可通过调整可用区内物理机的数量（加/减 OBServer）的方式改变集群的负载能力。以下从两个层面详细介绍 OceanBase 数据库的扩缩容能力。

可用区（Zone）动态调整

OceanBase 集群中的每一份数据都维护了多个副本，一份数据的多个副本通过 paxos 协议组成一个基本的高可用单元。通常情况下，会在每个可用区内部署至多一个数据副本，在少数可用区发生故障时，剩余可用区内的副本仍可以通过 paxos 协议，在保证数据完整的前提下，继续提供服务。可通过增加可用区的数量来增加数据的副本数，进而提高系统可用性，具体示例如下。

存在一个 OceanBase 集群，共包含 3 个可用区 Zone1、Zone2、Zone3。集群中的每一份数据包含 3 个副本，分别部署在以上 3 个可用区内。为进一步提高系统的可用性，期望升级数据副本数为 5 副本，可通过扩容可用区数量的方式达到这个目标。首先为集群添加两个新的可用区 Zone4 和 Zone5。然后在新的可用区 Zone4 和 Zone5 内添加物理机。随后可根据用户需求，在可用区 Zone4 和 Zone5 上部署新的数据副本，完成 Zone 级别的动态扩容。相反地，可用通过减少可用区的数量，来实现可用区级别的动态缩容操作。

物理机的扩缩容

为支持数据库服务能力的线性扩展，OceanBase 数据库支持可用区内物理机的动态扩缩容操作，即在可用区内调整物理机的数量来扩展服务能力，OceanBase 数据库实现服务能力线性扩展的，具体示例如下。

存在一个 OceanBase 集群，共包含 3N 个 OBServer（包含 3 个可用区 Zone1、Zone2、Zone3，每个 Zone 包含 N 个 OBServer）提供读写服务，当集群提供的服务能力不能完全满足读写请求时，需要对该 OceanBase 集群扩容以提高集群服务能力。为集群扩容三台新 OBServer，每个可用区扩容一台，扩容后每个可用区包含 N+1 台 OBServer。通过动态扩容，集群中的 OBServer 数量相应增加，RootService 会根据内部的负载均衡机制，将集群内原有的数据和负载依次均衡到新扩容的 OBServer 上，具体的负载均衡机制与策略，可参考 资源单元的均衡 和 数据均衡。

相反地，当集群中的 OBServer 的服务能力对当前负载有较多冗余时，可考虑通过缩减 OBServer 来降低集群成本，称即将被缩减掉的 OBServer 为待删除 OBServer。具体地，可执行集群管理指令，将待删除 OBServer 上的数据和负载迁移到其他可用 OBServer 上去，然后回收待删除 OBServer，完成物理机缩容。

4.4.2. 租户内资源的扩容和缩容

4.4.2.1. 租户内资源扩容和缩容概述

在 OceanBase 数据库中，租户内资源管理的方式主要有水平管理、垂直管理和跨 Zone 资源管理。

租户资源的水平管理

租户资源的水平管理是指，通过调整租户资源池的 `UNIT_NUM` 数量，增加或减少集群中为该租户提供服务的 OBServer 的数量，从而改变租户整体的服务能力，资源池的 `UNIT_NUM` 可根据要求动态调大或调小。资源水平管理的详细描述请参见 租户资源水平扩缩容。

租户资源的垂直管理

租户资源的垂直管理是指，通过调整租户内每个资源单元的资源大小，改变租户在各 OBServer 上的服务能力，进而改变租户整体的服务能力。改变租户资源单元的资源大小有以下两种手段：

- 给资源单元（Unit）切换新的资源规格（unit_config）
- 调整资源单元（Unit）当前资源规格（unit_config）的资源大小

资源垂直管理的详细描述请参见 租户资源垂直扩缩容。

跨 Zone 资源管理

租户的跨 Zone 资源管理是指，通过改变租户资源单元的 `ZONE_LIST`，来改变租户资源的分布范围，可以因此调整租户每一份数据的副本数量，进而改变租户的容灾能力。租户跨 Zone 资源管理的详细描述请参见 租户跨 Zone 资源管理。

4.4.2.2. 租户资源水平扩缩容

租户资源的水平管理主要是通过调整资源池的 `UNIT_NUM` 来动态调整租户的可用资源。具体表现为，您可以通过调整资源池的 `UNIT_NUM` 来调整资源池下每个 Zone 内资源单元的数量，进而提高或降低该租户在对应 Zone 上的服务能力。

调整资源池 `UNIT_NUM` 的示例语句如下：

```
obclient> CREATE RESOURCE POOL rp1 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('zone1', 'zone2');
obclient> ALTER RESOURCE POOL rp1 UNIT_NUM 3; // 调大 UNIT_NUM
obclient> ALTER RESOURCE POOL rp1 UNIT_NUM 2; // 调小 UNIT_NUM
obclient> ALTER RESOURCE POOL rp1 UNIT_NUM 1 DELETE UNIT = (1001, 1003); // 指定资源单元调小
UNIT_NUM
```

`UNIT_NUM` 的变更也分为两类，即调大 `UNIT_NUM` 和调小 `UNIT_NUM` 。其中，示例语句 `ALTER RESOURCE POOL rp1 UNIT_NUM 3;` 是调大 `UNIT_NUM` ，示例语句 `ALTER RESOURCE POOL rp1 UNIT_NUM 2;` 和 示例语句 `ALTER RESOURCE POOL rp1 UNIT_NUM 1 DELETE UNIT = (1001, 1003);` 为调小 `UNIT_NUM` 。

调大 UNIT_NUM

调大 `UNIT_NUM` 通常出现在租户扩容的场景中，例如，租户的当前工作状态如下所示。

```
obclient> CREATE RESOURCE POOL r_p0 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('zone1', 'zone2', 'z
one3');
obclient> CREATE TENANT tt RESOURCE_POOL_LIST = ('r_p0');
```

示例中，租户 `tt` 使用资源池 `r_p0` 。在 `zone1` 、 `zone2` 、 `zone3` 每个 Zone 上分别有 2 个资源单元，随着业务量的不断变大，每个 Zone 上 2 个资源单元无法承载当前的业务量，因此需要考虑调大 `unit num` 来提高租户的服务能力，以满足新的业务需求。

您可以使用以下语句将租户在资源池 `r_p0` 的资源单元数量调整为 `3` ，进而提高租户的服务能力。

```
obclient> ALTER RESOURCE POOL r_p0 UNIT_NUM = 3;
```

调小 UNIT_NUM

调小 `UNIT_NUM` 实际就是删除资源单元，由于被删除的资源单元上可能存在对应租户的数据，因此，在执行删除资源单元的语句成功返回后，系统还需要将租户数据从被删除的资源单元上搬走。具体表现为，OceanBase 数据库会将本次删除资源单元的操作记录到 `__all_rootservice_job` 内部表中，查询该表可以观察本次资源单元的删除进度。`UNIT_NUM` 被调小后，即将被删除的资源单元会被标记为 `DELETING` 状态，可以通过查询 `__all_unit` 表的 `status` 列来获取资源单元的状态。当租户数据从被删除资源单元全部搬走后，OceanBase 数据库会更新 `__all_rootservice_job` 表，标记本次删除资源单元的任务完成，同时清理 `__all_unit` 表，将处于 `DELETING` 状态的资源单元删除。

下面以几个示例进行进一步进行说明：

- 示例 1：调小 `UNIT_NUM` 时，不指定 `unit_id`

  i. 创建资源池 `test_pool`，且该资源池包含 3 个 Zone `z1`、`z2` 和 z3，每个 Zone 内三个资源单元。

  ```
  obclient> CREATE RESOURCE POOL test_pool UNIT='uc', UNIT_NUM=3, ZONE_LIST=('z1','z2',
  'z3');
  ```

  ii. 执行以下语句，将每个 Zone 内的 `UNIT_NUM` 调整为 `1`。

  ```
  obclient> ALTER RESOURCE POOL test_pool UNIT_NUM=1;
  ```

  由于本示例中未明确指定待删除的资源单元的 `unit_id`，系统会自动在每个 Zone 内选取 2 个资源单元进行删除。

- 示例 2：调小 `UNIT_NUM` 时，指定 `unit_id`

  i. 创建资源池 `test_pool`，且该资源池包含 3 个 Zone `z1`、`z2` 和 z3，每个 Zone 内三个资源单元。

  假设 `z1` 上的 `unit_id` 为 `1001`、`1002`、`1003`，`z2` 上的 `unit_id` 为 `1004`、`1005`、`1006`，`z3` 上的 `unit_id` 为 `1007`、`1008`、`1009`。

  ```
  obclient>CREATE RESOURCE POOL test_pool UNIT='unit_config', UNIT_NUM=3, ZONE_LIST=('z
  1','z2','z3');
  ```

  ii. 执行以下语句，将每个 Zone 内的 `UNIT_NUM` 调整为 `2`。

  ```
  obclient> ALTER RESOURCE POOL test_pool UNIT_NUM=2 DELETE UNIT=(1001,1004,1007);
  ```

  由于本示例中明确指定了待删除的资源单元的列表为 `(1001,1004,1007)`，则系统会直接删除指定的资源单元。

下面再给出 2 个非法调小 `UNIT_NUM` 的反例：

- 反例 1：调小 `UNIT_NUM 时` ，各 Zone 内删除的资源单元数量不相等。

  i. 创建资源池 `test_pool` ，且该资源池包含 3 个 Zone `z1` 、 `z2` `和` z3，每个 Zone 内三个资源单元。

  假设 `z1` 上的 `unit_id` 为 `1001` 、 `1002` 、 `1003` ， `z2` 上的 `unit_id` 为 `1004` 、 `1005` 、 `1006` ， `z3` 上的 `unit_id` 为 `1007` 、 `1008` 、 `1009` 。

  ```
  obclient> CREATE RESOURCE POOL test_pool UNIT='unit_config', UNIT_NUM=3, ZONE_LIST=('
  z1','z2','z3');
  ```

  ii. 执行以下语句，在 `z1` 、 `z2` 上各删除 1 个资源单元， `z3` 上不删除资源单元。

  ```
  obclient> ALTER RESOURCE POOL test_pool UNIT_NUM=2 DELET UNIT=(1001,1004);
  ```

  由于各 Zone 内删除的资源单元数量不相同，是一个非法操作，系统会报错。

- 反例 2：调小 `UNIT_NUM 时` ，各 Zone 内删除的资源单元数量与 `UNIT_NUM` 不匹配。

  i. 创建资源池 `test_pool` ，且该资源池包含 3 个 Zone `z1` 、 `z2` `和` z3，每个 Zone 内三个资源单元。

  假设 `z1` 上的 `unit_id` 为 `1001` 、 `1002` 、 `1003` ， `z2` 上的 `unit_id` 为 `1004` 、 `1005` 、 `1006` ， `z3` 上的 `unit_id` 为 `1007` 、 `1008` 、 `1009` 。

  ```
  obclient> CREATE RESOURCE POOL test_pool UNIT='unit_config', UNIT_NUM=3, ZONE_LIST=('
  z1','z2','z3');
  ```

  ii. 执行以下命令，指定 `UNIT_NUM` 的数量为 `1` ，并在 `z1` 、 `z2` 、 `z3` 每个 Zone 上删除 1 个资源单元。

  ```
  obclient> ALTER RESOURCE POOL test_pool UNIT_NUM=1 DELETE UNIT=(1001,1004,1007);
  ```

  由于在每个 Zone 上各删除 1 个资源单元， `UNIT_NUM` 的变化值应该为 `2` ，各 Zone 内删除资源单元的数量与 `UNIT_NUM` 的变化值不匹配，是一个非法操作，系统会报错。

4.4.2.3. 租户资源垂直扩缩容

租户资源的垂直管理主要是通过调整租户资源池的资源规格来调整租户的服务能力。调整租户资源池的规格有两种方法：修改资源配置和切换资源配置。

修改资源配置

修改资源池的资源配置，即直接调整资源配置的 CPU 或 Memory 等的值，进而直接影响租户在该资源池上的资源规格和服务能力。

假设创建了资源池 `pool1` 和 `pool2` ，且其资源配置都是 `uc1` ，可以通过修改资源池的资源配置的方式，将资源配置 `uc1` 的 `MAX_CPU` 调整为 `6` ； `MIN_MEMORY` 调整为 `36G` ，其他选项不变。示例语句如下：

```
obclient> CREATE RESOURCE UNIT uc1 MAX_CPU 5, MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMORY '32G'
, MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
obclient>CREATE RESOURCE POOL pool1 UNIT 'uc1', UNIT_NUM 2, ZONE_LIST ('z1', 'z2');
obclient>CREATE RESOURCE POOL pool2 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z3');
obclient>CREATE TENANT tt resource_pool_list=('pool1','pool2');
obclient>ALTER RESOURCE UNIT uc1 MAX_CPU 6, MIN_MEMORY '36G';
```

通过调整资源配置的各个选项，可调整租户的资源池在对应 Zone 上的资源规格，进而影响租户的服务能力。

切换资源配置

切换资源池的资源配置，即调整资源池下每个资源单元的资源规格，进而调整租户在该资源池上的资源规格和服务能力。

假设资源池 `rp1` 之前的资源配置为 `uc1` ，则通过切换资源池的资源配置将 `rp1` 的资源配置从 `uc1` 变更为 `uc2` 的示例语句如下：

```
obclient> ALTER RESOURCE POOL rp1 UNIT 'uc2';
```

理论上 OceanBase 数据库支持对资源规格 `MIN_CPU` 、 `MAX_CPU` 、 `MIN_MEMORY` 以及 `MAX_MEMORY` 同时进行修改。

示例及使用限制

通常情况下，通过修改资源配置和切换资源配置的方式都能对租户的服务能力进行调整。对应到租户层面，实际上是调整了租户资源单元的规格。对资源规格的修改通常有两种场景，即调大资源规格和调小资源规格。

调大资源规格

调大资源规格主要用在租户的资源扩容场景中，可分别对 CPU 和 Memory 进行资源扩容。

示例：

- 示例 1
  i. 创建一个资源配置 `u_c0` ，并创建一个资源池 `pool1` ， `pool1` 使用 `u_c0` 作为自己的资源配置。

  ```
  obclient> CREATE RESOURCE UNIT u_c0 MAX_CPU 5, MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMOR
  Y '32G', MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
  obclient> CREATE RESOURCE POOL pool1 unit='u_c0', unit_num=3, zone_list=('z1','z2','z
  3');
  ```

  ii. 调大 `u_c0` 的 `MIN_CPU` 、 `MAX_CPU` 、 `MIN_MEMORY` 或 `MAX_MEMORY` 。

```
obclient> ALTER RESOURCE UNIT u_c0 MAX_CPU 10, MIN_CPU 8, MAX_MEMORY '72G', MIN_MEMOR
Y '64G';
```

该调整旨在调大资源池 `pool1` 的资源规格，目的是提高相应租户的服务能力。

- 示例 2

  i. 创建两个资源配置 `u_c0` 和 `u_c1` ，并创建一个资源池 `pool1` ， `pool1` 使用 `u_c0` 作为自己的资源配置。

  ```
  obclient> CREATE RESOURCE UNIT u_c0 MAX_CPU 5, MIN_CPU 4, MAX_MEMORY '36G', MIN_MEMOR
  Y '32G', MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
  obclient> CREATE RESOURCE UNIT u_c1 MAX_CPU 10, MIN_CPU 8, MAX_MEMORY '72G', MIN_MEMO
  RY '64G', MAX_IOPS 128, MIN_IOPS 128, MAX_DISK_SIZE '2T', MAX_SESSION_NUM 64;
  obclient> CREATE RESOURCE POOL pool1 unit='u_c0', unit_num=3, zone_list=('z1','z2','z
  3');
  ```

  ii. 将 `pool1` 的资源配置调整成 `u_c1` 。

  ```
  obclient> ALTER RESOURCE POOL pool1 unit='u_c1';
  ```

  该调整旨在调大资源池 `pool1` 的资源规格，目的是提高相应租户的服务能力。

注意，在调大资源规格时，无论是通过修改资源配置还是切换资源配置，调整后的资源总量都满足以下要求：

```
Sum(min_cpu) <= CPU_CAPACITY;
Sum(min_memory) <= MEM_CAPACITY;
Sum(max_cpu) <= CPU_CAPACITY * resource_hard_limit;
Sum(max_mem) <= CPU_CAPACITY * resource_hard_limit;
```

否则，系统会报错，提示扩容失败。

调小资源规格

根据上面调大资源规格的示例，系统还支持将资源池的资源规格调小。调小资源规格能够保证机器资源规格足够用，但仍然存在限制，该限制主要出现在调小资源规格的 `MIN_MEMORY` 选项上，一个租户在某物理机上的实际内存使用量记录为 `MEM_USED` ，在调小其资源规格时，对于为租户提供服务的每台物理机，需要满足资源配置的 `MIN_MEMORY` 大于等于租户在该物理机上当前的实际使用内存。如果在调整时， `MIN_MEMORY` 的值小于 `MEM_USED` ，则本次资源规格的调整会失败，需要进行一次数据的 Checkpoint 操作来释放内存，然后再重新尝试调小资源池的配置规格。

4.4.2.4. 租户跨 Zone 资源管理

租户的跨 Zone 资源管理是一种通过调整资源池的 Zone 分布范围来调整租户每一份数据的副本数，进而改变租户数据容灾能力的资源管理方式。

对资源池的 `ZONE_LIST` 支持如下集中类型的操作：

- `ZONE_LIST` 变更
- 分裂资源池
- 合并资源池

ZONE_LIST 的变更

调整资源池的 `ZONE_LIST` ，可以调整资源池在 Zone 维度的使用范围，从而调整租户数据在 Zone 维度的服务范围。

调整资源池 `ZONE_LIST` 的示例语句如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT_NUM=3, UNIT='unit_config', ZONE_LIST=('z1','z2','
z3');
obclient> CREATE RESOURCE POOL pool2 UNIT_NUM=3, UNIT='unit_config', ZONE_LIST=('z1','z2','
z3');
obclient> ALTER RESOURCE POOL pool1 ZONE_LIST=('z1','z2','z3','z4');
obclient> ALTER RESOURCE POOL pool2 ZONE_LIST=('z1','z2');
```

`ZONE_LIST` 的变更可以分为两类，即扩大 Zone 维度的使用范围和缩小 Zone 维度的使用范围。示例语句 `ALTER RESOURCE POOL pool1 ZONE_LIST=('z1','z2','z3','z4');` 是扩大 Zone 维度使用范围，示例语句 `ALTER RESOURCE POOL pool2 ZONE_LIST=('z1','z2');` 是缩小 Zone 维度使用范围。

扩大 Zone 维度的使用范围

扩大 Zone 维度的使用范围通常使用在以下两个场景中：

- 场景 1： 租户副本数升级，以提高租户的可用性。
- 场景 2：数据从一个机房（Zone）搬迁到另一个机房（Zone）时，在新机房中进行数据补全。

下面通过示例介绍这 2 种使用场景：

- 场景 1：副本数升级示例

  需要将租户 `tt` 从 3 副本升级为 5 副本，数据范围从 `'z1', 'z2', 'z3'` 扩大到 `'z1','z2','z3','z4','z5'` ，以便提高租户的可用性。

  ```
  obclient>CREATE RESOURCE POOL r_p0 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z1', 'z2', 'z3');
  obclient>CREATE TENANT tt RESOURCE_POOL_LIST = ('r_p0');

  obclient>ALTER RESOURCE POOL r_p0 ZONE_LIST=('z1','z2','z3','z4');
  obclient>ALTER TENANT tt locality="F@z1,F@z2,F@z3,F@z4";//在 z4 上为租户 tt 增加数据

  obclient>ALTER RESOURCE POOL r_p0 ZONE_LIST=('z1','z2','z3','z4','z5');
  obclient> ALTER TENANT tt locality="F@z1,F@z2,F@z3,F@z4,F@z5";//在 z5 上为租户 tt 增加数据
  ```

- 场景 2：在新机房中补全数据

  新机房内的数据补全是数据在机房内搬迁的一个子操作，假设租户期望将机房 `z1` 的数据搬迁到 `z4` ，则新机房的数据补全的具体操作如下：

```
obclient> CREATE RESOURCE POOL r_p0 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z1', 'z2', 'z3');
obclient> CREATE TENANT tt RESOURCE_POOL_LIST = ('r_p0');

// 以下为新机房中数据补全子操作
obclient> ALTER RESOURCE POOL r_p0 ZONE_LIST=('z1','z2','z3','z4');
obclient> ALTER TENANT tt locality="F@z1,F@z2,F@z3,F@z4";//在 z4 上为租户 tt 增加数据
// 接着再从旧机房中删除数据的子操作可以参考缩小 Zone 维度的使用范围
```

以上操作帮助租户先在 `z4` 机房内添加数据，完成了数据在机房间搬迁过程中，在新机房补全数据的操作。

缩小 Zone 维度的使用范围

缩小 Zone 维度的使用范围通常使用在以下 2 个场景中：

- 场景 1：租户副本数降级，以减少租户的存储资源使用量。
- 场景 2：数据从一个机房（Zone）搬迁到另一个机房（Zone）时，在旧机房中删除数据。

下面通过示例介绍这 2 种使用场景：

- 场景 1：副本数降级示例

  需要将租户 tt 从 5 副本降级为 3 副本，数据范围从 `'z1','z2','z3','z4','z5'` 缩小到 `'z1','z2','z3'` 。减少租户存储资源使用量。

```
obclient> CREATE RESOURCE POOL r_p0 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z1','z2','z3','z4
','z5');
obclient> CREATE TENANT tt RESOURCE_POOL_LIST = ('r_p0');

obclient> ALTER TENANT tt locality="F@z1,F@z2,F@z3,F@z4";//在 z5 上为租户 tt 删除数据
obclient> ALTER RESOURCE POOL r_p0 ZONE_LIST=('z1','z2','z3','z4'); // 为资源池删除 z5
obclient> ALTER TENANT tt locality="F@z1,F@z2,F@z3";//在 z4 上为租户 tt 删除数据
obclient> ALTER RESOURCE POOL r_p0 ZONE_LIST=('z1','z2','z3'); // 为资源池删除 z4
```

- 场景 2：从旧机房中删除数据示例

  旧机房内的数据删除是数据在机房内搬迁的另一个子操作，假设租户期望将机房 `z1` 的数据搬迁到 `z4` 。则旧机房的数据删除的具体操作如下：

```
obclient> CREATE RESOURCE POOL r_p0 UNIT 'uc1', UNIT_NUM 1, ZONE_LIST ('z1', 'z2', 'z3');
obclient> CREATE TENANT tt RESOURCE_POOL_LIST = ('r_p0');

// 以下为新机房中数据补全子操作
obclient> ALTER RESOURCE POOL r_p0 ZONE_LIST=('z1','z2','z3','z4');
obclient> ALTER TENANT tt locality="F@z1,F@z2,F@z3,F@z4";//在 z4 上为租户 tt 增加数据

// 以下为旧机房中数据删除子操作
obclient> ALTER TENANT tt locality="F@z2,F@z3,F@z4";//先在 z1 上为租户 tt 删除数据
obclient> ALTER RESOURCE POOL r_p0 ZONE_LIST=('z2','z3','z4');
```

上面的操作帮助租户先在 z4 机房内添加数据，然后在 z1 机房内删除数据，将租户数据从 z1 搬迁到 z4 。

分裂资源池

分裂资源池操作可将一个资源池分裂为多个资源池，分裂资源池操作的基本语法和示例如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z1','z2','z3');
obclient> ALTER RESOURCE POOL pool1 SPLIT INTO ('pool10','pool11','pool12') ON ('z1','z2','
z3');
obclient> ALTER RESOURCE POOL pool10 UNIT='uc1';
obclient> ALTER RESOURCE POOL pool11 UNIT='uc2';
obclient> ALTER RESOURCE POOL pool12 UNIT='uc3';
```

分裂资源池的操作主要应用在如下场景中：资源池 pool1 的当前使用范围是 z1 、 z2 、 z3 ，而资源配置规格均为 uc0 ，由于 z1 、 z2 、 z3 等 3 个 Zone 上的物理机规格可能有较大差别，3 个 Zone 内如果使用同一个资源规格 uc0 ，无法充分利用每个 Zone 内物理机的资源。分裂资源池可以将一个多 Zone 资源池分裂为多个单 Zone 资源池，再为每个单 Zone 资源池配置各自的资源配置规格。

在该示例中，将一个 'z1','z2','z3' 跨度的多 Zone 资源池 pool1 分裂成三个单 Zone 的资源池 pool10 （ ZONE_LIST 为 z1 ，并继承 pool1 之前在 z1 上的资源单元）、 pool11 （ ZONE_LIST 为 z2 ，并继承 pool1 之前在 z2 上的资源单元）和 pool12 （ ZONE_LIST 为 z3 ，并继承 pool1 之前在 z3 上的资源单元）。分裂完成后， pool10 、 pool11 、 pool12 的默认资源配置仍然为 pool1 的资源配置 uc0 ，用户可根据各 Zone 的资源情况自行调整各新资源池的资源配置。

合并资源池

合并资源池操作可以将多个资源池分裂为一个资源池，合并资源池操作的基本语法和示例如下：

```
obclient> CREATE RESOURCE POOL pool1 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z1');
obclient> CREATE RESOURCE POOL pool2 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z2');
obclient> CREATE RESOURCE POOL pool3 UNIT='uc0', UNIT_NUM=1, ZONE_LIST=('z3');
obclient> ALTER RESOURCE POOL MERGE ('pool1','pool2','pool3') INTO ('pool0');
```

合并资源池是分裂资源池的逆操作。合并资源池的操作主要应用在如下场景中：资源池 pool1 的当前使用范围是 z1 ，资源配置为 uc0 ；资源池 pool2 的当前使用范围是 z2 ，资源配置为 uc0 ，资源池 pool3 的当前使用范围是 z3 ，资源配置规格为 uc0。由于 pool1 、 pool2 、 pool3 这 3 个资源池使用相同的资源配置规格，可以将多个资源池合并为一个资源池，降低资源池的管理和运维成本。

在该示例中，将三个单 Zone 资源池 pool1 、 pool2 、 pool3 合并成一个多 Zone 资源池 pool0 。

合并资源池有以下两个限制：

- 被和并的资源池的 `UNIT_NUM` 需要相等。
- 被合并的资源池的资源配置需要是同一个。

5.数据链路

5.1. 数据链路概述

OceanBase 数据链路提供了从客户端到数据库端的最佳数据访问能力，对用户屏蔽 OceanBase 分布式数据库的感知，提供分布式数据库的可扩展、高性能和高可用的服务能力。数据链路包含两个部分：数据库代理和数据库驱动。

数据库代理

OceanBase 数据库代理 ODP（OceanBase Database Proxy，又称 OBProxy）是 OceanBase 专用的代理服务。 OceanBase 数据库中的数据会以多副本的形式存放在各个 OBServer 上，ODP 则负责接收用户的 SQL 请求，结合请求中涉及数据的分布，将用户 SQL 请求转发到最佳 OBServer 上，在 OBServer 执行完成后接受结果并将执行结果返回给用户。

作为 OceanBase 数据库的关键组件，ODP 具有以下特性：

- **高性能转发**

  ODP 完整兼容 MySQL 协议，并支持 OceanBase 自研协议，采用多线程异步框架和透明流式转发的设计，保证了数据的高性能转发，同时确保了自身对机器资源的最小消耗。

- **最佳路由**

  ODP 会充分考虑用户请求涉及的副本位置、用户配置的读写分离路由策略、OceanBase 多地部署的最优链路，以及 OceanBase 各机器的状态及负载情况，将用户的请求路由到最佳的 OBServer，最大程度的保证了 OceanBase 整体的高性能运转。

- **连接管理**

  针对一个客户端的物理连接，ODP 维持自身到后端多个 OBServer 的连接，采用基于版本号的增量同步方案维持了每个 OBServer 连接的会话状态，保证了客户端高效访问各个 OBServer。

- **专有协议**

  ODP 与 OBServer 默认采用了 OceanBase 专有协议，如：增加报文的 CRC 校验来保证 OBServer 链路的正确性，增强传输协议以支持 Oracle 兼容需要的数据类型和交互模型。

- **易运维**

  ODP 本身无状态，支持无限水平扩展，支持同时访问多个 OceanBase 集群。可以通过丰富的内部命令实现对自身状态的实时监控，提供极大的运维便利性。

有关数据库代理的详细内容，请参见 数据库代理。

数据库驱动

数据库驱动程序由数据库厂商或者其他一些专门开发数据库驱动程序的厂商提供，它们提供给外部应用程序一个访问该数据库的接口。OceanBase 数据库目前提供了 MySQL 模式数据库（兼容 MySQL ）和 Oracle 模式数据库（兼容 Oracle ）。在 OceanBase MySQL 模式下，用户可以直接使用 MySQL 官方提供的 Connector 来使用 OceanBase 数据库（暂不支持 8.0 的驱动）。在 OceanBase Oracle 模式下，需要使用 OceanBase 自研的数据库驱动。OceanBase 数据库驱动同时支持 OceanBase 的 MySQL 和 Oracle 两种模式，在使用时可以自动识别 OceanBase 的运行模式是 MySQL 还是 Oracle，无需额外设置。

OceanBase 数据库支持各类语言的数据库驱动，主要有如下产品：

- **OBCI**

  兼容 OCI 接口，支持 OceanBase Oracle 模式的 C 语言驱动。

有关 OBCI 的详细介绍，请参见 OBCI。

- **OceanBase Connector/C**

  兼容 ODBC 接口，支持 OceanBase MySQL 和 OceanBase Oracle 模式的 C 语言驱动。

  有关 OceanBase Connector/C 的详细介绍，请参见 OceanBase Connector/C。

- **OceanBase Connector/J**

  兼容 JDBC 接口，支持 OceanBase MySQL 和 OceanBase Oracle 模式的 Java 语言驱动。

  有关 OceanBase Connector/J 的详细介绍，请参见 OceanBase Connector/J。

有关数据库驱动的详细内容，请参见 数据库驱动。

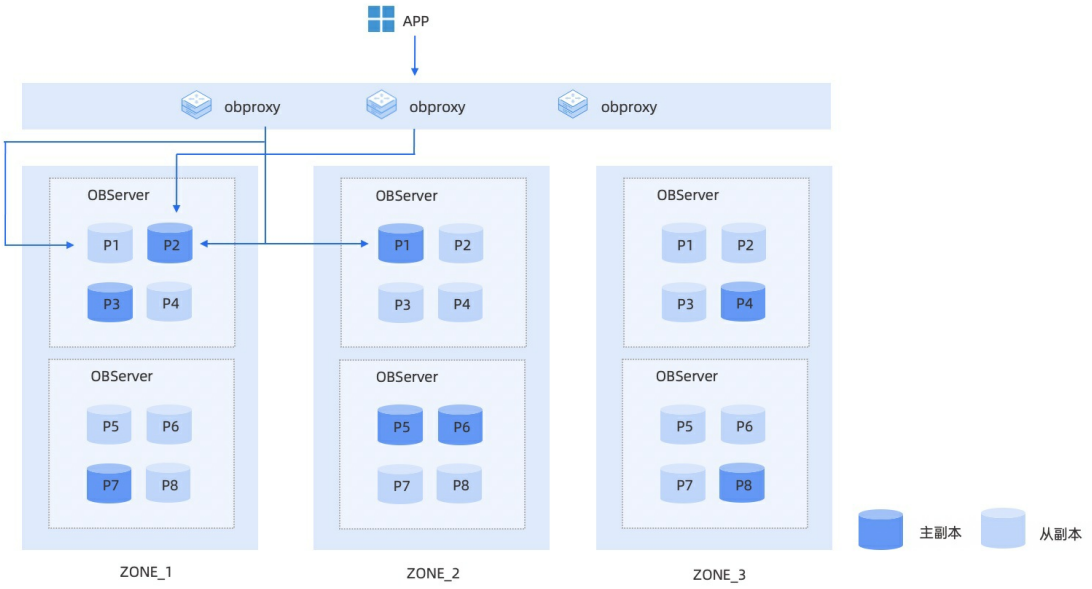
5.2. 数据库代理

5.2.1. 代理概述

OceanBase Database Proxy（简称 ODP ）是 OceanBase 数据库专用的代理服务器。OceanBase 数据库的用户数据以多副本的形式存放在各个 OBServer 上，ODP 接收用户发出的 SQL 请求，并将 SQL 请求转发至最佳目标 OBServer，最后将执行结果返回给用户。

为什么需要 ODP?

ODP 是代理服务器，代理服务器会让访问数据库的链路多一跳，那为什么需要 ODP 呢？我们以下图为例进行说明。



图中 APP 是我们的业务程序，APP 前面有三台 OBPoxy（ ODP 的进程名叫做 obproxy ），在实际部署中，OBPoxy 和 APP 之间一般会有一个负载均衡（如：F5 ）将请求分散到多台 OBProxy 上面，OBProxy 下面是 OBServer，图中有 6 台 OBServer。

需要使用 ODP 的原因如下：

- 连接管理：OBServer 集群规模庞大，机器、软件出现问题或者本身运维机器上线、下线概率较大，如果直连 OBServer，遇到上面的情况客户端就会发生断连。ODP 屏蔽了 OBServer 本身分布式的复杂性，客户连接 ODP，ODP 可以保证连接的稳定性，自身对 OBServer 的复杂状态进行处理。

- 数据路由：ODP 可以获取到 OBServer 中的数据分布信息，可以将用户 SQL 高效转发到数据所在机器，执行效率更高。如：表 t1 数据在图中 P1 内，表 t2 数据在图中 P2 内，表 t3 数据在图中 P3 内，红色表示主副本，蓝色表示备副本。对于 insert into t1 语句 ODP 可以将 SQL 转发到 ZONE_2 中含有 P1 主副本的机器上。对于 update t2 语句 ODP 可以将 SQL 转发到 ZONE_1 中含有 P2 主副本的机器上。

ODP 可以实现像使用单机数据库一样使用分布式数据库。

ODP 特性

作为 OceanBase 数据库的关键组件，ODP 具有如下特性：

- 高性能转发

  ODP 完整兼容 MySQL 协议，并支持 OceanBase 自研协议，采用多线程异步框架和透明流式转发的设计，保证了数据的高性能转发，同时确保了自身对机器资源的最小消耗。

- 最佳路由

  ODP 会充分考虑用户请求涉及的副本位置、用户配置的读写分离路由策略、OceanBase 数据库多地部署的最优链路，以及 OceanBase 数据库各机器的状态及负载情况，将用户的请求路由到最佳的 OBServer，最大程度的保证了 OceanBase 数据库整体的高性能运转。

- 连接管理

  针对一个客户端的物理连接，ODP 维持自身到后端多个 OBServer 的连接，采用基于版本号的增量同步方案维持了每个 OBServer 连接的会话状态，保证了客户端高效访问各个 OBServer。

- 专有协议

  ODP 与 OBServer 默认采用了 OceanBase 专有协议，如增加报文的 CRC 校验保证与 OBServer 链路的正确性，增强传输协议以支持 Oracle 兼容性的数据类型和交互模型。

- 安全可信

  ODP 支持使用 SSL 访问数据，并和 MySQL 协议做了兼容，满足客户安全需求。

- 易运维

  ODP 本身无状态支持无限水平扩展，支持同时访问多个 OceanBase 集群。可以通过丰富的内部命令实现对自身状态的实时监控，提供极大的运维便利性。

许可证

ODP 社区版完全开源，使用 MulanPubL - 2.0 许可证，您可以免费复制和使用源代码。当您修改或分发源代码时，请遵守木兰协议。

5.2.2. SQL 路由

ODP 会充分考虑用户请求涉及的副本位置、用户配置的读写分离路由策略、OceanBase 数据库多地部署的最优链路，以及 OceanBase 数据库各机器的状态及负载情况，将用户的请求路由到最佳的 OBServer，最大程度的保证了 OceanBase 数据库整体的高性能运转。

开始阅读本节内容之前，您可以先了解一些路由相关的概念（参见 附录：OceanBase 数据库基础概念），便于您更好的了解下面的内容。

- Zone
- Region
- Server List
- RS List

- Location Cache
- 副本
- 合并
- 强一致性读/弱一致性读
- 读写Zone/只读Zone
- 分区表
- 分区键

OceanBase 数据库执行计划

执行计划分为三种：Local 、Remote 、Distribute。ODP 的作用主要是尽量避免 Remote 计划（效率低，性能差），将路由尽可能准确变为 Local 计划。

ODP 路由作用

理解了上述 AZ/Region/分区/副本的基本概念和物理含义后，就能理解 ODP 的路由思路了。从分区的设计到其物理分布，以及本地执行计划的高效性考虑，ODP 需要尽可能对 SQL 进行准确路由，主要过程将经历：SQL 解析、分区计算、分区信息获取、副本策略选择等。

非分区表路由

非分区表可以直接利用 Location Cache 中的副本信息。ODP 保存了分区和 OBServer 地址的映射，通过解析 SQL 中的表名，根据表名查询 ODP 缓存中的分区对应的机器 IP。缓存的有效性，有以下三种情况：

- 缓存中找不到，此时需要访问 OBServer 查询最新映射并缓存。
- 缓存中存在但不可用，此时需要重新去 OBServer 查询并更新。
- 缓存中存在且可用，此时可以直接使用。

分区表路由

分区表的路由相比非分区表而言，增加了分区 ID 及其相关的计算和查询过程。在获取了 Location Cache 后，分区表需要继续判定表的一级/二级分区，根据不同分区键类型和计算方式，计算分区 ID 并获取对应主备副本信息。

做分区计算时，通过表结构可以得知分区键及其类型，之后通过解析 SQL 语句获取对应分区键的值，并根据表结构和分区键类型做分区计算，从而能转发到对应分区所在的机器。

正常情况下，通过分区计算，ODP 可以将 SQL 路由到分区对应的机器上，从而避免 Remote 执行，提升效率。在 ODP V3.2.0 版本，已针对分区表且无法计算分区路由的场景进行优化，由随机选择租户的机器路由，优化成随机从分布了分区的机器中随机路由，提升命中率，尽可能减少 Remote 执行。

副本路由选择（常规部署）

对于强一致性读，且 SQL 指定了表名，将路由到该表对应分区的 Leader OBServer；对于弱一致性读、登录认证请求、强一致性读但没有指定表名等情况，存在主备均衡路由（默认）、备优先路由、不合并备优先路由这三种路由策略。

主备均衡路由（默认）

将按照以下优先级进行路由选择：

1. 相同 Region，相同 IDC，不在合并状态的 OBServer。
2. 相同 Region，不同 IDC，不在合并状态的 OBServer。

3. 相同 Region，相同 IDC，正在合并状态的 OBSserver。
4. 相同 Region，不同 IDC，正在合并状态的 OBSserver。
5. 不同 Region，不在合并状态的 OBServer。
6. 不同 Region，正在合并状态的 OBSserver。

备优先路由

常规部署下，支持备优先读策略。通过用户级别的系统变量 `proxy_route_policy` 控制，仅在常规部署和弱一致性读的情况下生效，优先读 Follower 而非主备均衡路由。

在常规模式部署和弱一致性读时，设置 `set @proxy_route_policy='follower_first';` ，路由将优先发往备 OBServer，即使 OBServer 处于正在合并状态。将按照以下优先级进行路由选择：

1. 相同 Region，相同 IDC，不在合并状态的备 OBServer。
2. 相同 Region，不同 IDC，不在合并状态的备 OBServer。
3. 相同 Region，相同 IDC，正在合并状态的备 OBServer。
4. 相同 Region，不同 IDC，正在合并状态的备 OBServer。
5. 相同 Region，相同 IDC，不在合并状态的主 OBServer。
6. 相同 Region，不同 IDC，不在合并状态的主 OBServer。
7. 不同 Region，不在合并状态的备 OBServer。
8. 不同 Region，正在合并状态的备 OBServer。
9. 不同 Region，不在合并状态的主 OBServer。
10. 不同 Region，正在合并状态的主 OBServer。

不在合并的备优先路由

常规部署下，弱一致性读，设置 `set @proxy_route_policy='unmerge_follower_first';` 时，路由将优先发往不在合并的备 OBServer。将按照以下优先级进行路由选择：

1. 相同 Region，相同 IDC，不在合并状态的备 OBServer。
2. 相同 Region，不同 IDC，不在合并状态的备 OBServer。
3. 相同 Region，相同 IDC，不在合并状态的主 OBServer。
4. 相同 Region，不同 IDC，不在合并状态的主 OBServer。
5. 相同 Region，相同 IDC，正在合并状态的备 OBServer。
6. 相同 Region，不同 IDC，正在合并状态的备 OBServer。
7. 不同 Region，不在合并状态的备 OBServer。
8. 不同 Region，不在合并状态的主 OBServer。
9. 不同 Region，正在合并状态的备 OBServer。
10. 不同 Region，正在合并状态的主 OBServer。

其他

常规部署下，弱一致性读，当 `proxy_route_policy` 变量取其他值时，路由选择将退化为普通的弱一致性读主备均衡路由策略，即主备均衡路由(默认)。

副本路由选择（读写分离部署）

即使用了只读副本和只读Zone的部署模式。读写分离部署时，不存在备优先读的路由策略，路由依赖系统变量 `ob_route_policy` 。主要存在以下几种情况：

1. 对于强一致性读语句，且 SQL 指定了表名，那么直接路由到该表涉及的分区的 Leader OBServer 执行。
2. 对于强一致性读语句，SQL 为“select 不指定表名/use database/set session 级别系统变量”时，忽略 Zone 属性，此时等同于常规部署下的主备均衡路由（默认）。
3. 对于强一致性读语句，不包含登录认证请求，且去除 1 & 2 场景的请求，将按照以下策略进行路由：
   i. 相同 Region，相同 IDC，不在合并状态的读写 Zone 中的 OBServer。
   ii. 相同 Region，不同 IDC，不在合并状态的读写 Zone 中的 OBServer。
   iii. 相同 Region，相同 IDC，正在合并状态的读写 Zone 中的 OBServer。
   iv. 相同 Region，不同 IDC，正在合并状态的读写 Zone 中的 OBServer。
   v. 不同 Region，不在合并状态的读写 Zone 中的 OBServer。
   vi. 不同 Region，正在合并状态的读写 Zone 中的 OBServer。

只读 Zone 优先路由（默认）

对于弱一致性语句和登录认证请求，将根据系统变量 `ob_route_policy` 的不同取值进行路由。设置 `ob_route_policy = readonly_zone_first` （默认值）时，将按照以下策略进行路由：

1. 相同 Region，相同IDC，不在合并状态的只读 Zone 中的 OBServer。
2. 相同 Region，不同IDC，不在合并状态的只读 Zone 中的 OBServer。
3. 相同 Region，相同IDC，正在合并状态的只读 Zone 中的 OBServer。
4. 相同 Region，不同IDC，正在合并状态的只读 Zone 中的 OBServer。
5. 相同 Region，相同IDC，不在合并状态的读写 Zone 中的 OBServer。
6. 相同 Region，不同IDC，不在合并状态的读写 Zone 中的 OBServer。
7. 相同 Region，相同IDC，正在合并状态的读写 Zone 中的 OBServer。
8. 相同 Region，不同IDC，正在合并状态的读写 Zone 中的 OBServer。
9. 不同 Region，不在合并状态的只读 Zone 中的 OBServer。
10. 不同 Region，正在合并状态的只读 Zone 中的 OBServer。
11. 不同 Region，不在合并状态的读写 Zone 中的 OBServer。
12. 不同 Region，正在合并状态的读写 Zone 中的 OBServer。

仅发送到只读 Zone 的路由

设置 `ob_route_policy = only_readonly_zone` 时，将按照以下策略进行路由：

1. 相同 Region，相同 IDC，不在合并状态的只读 Zone 中的 OBServer。

2. 相同 Region，不同 IDC，不在合并状态的只读 Zone 中的 OBServer。
3. 相同 Region，相同 IDC，正在合并状态的只读 Zone 中的 OBServer。
4. 相同 Region，不同 IDC，正在合并状态的只读 Zone 中的 OBServer。
5. 不同 Region，不在合并状态的只读 Zone 中的 OBServer。
6. 不同 Region，正在合并状态的只读 Zone 中的 OBServer。

不在合并状态的 Zone 优先路由

设置 `ob_route_policy = unmerge_zone_first` 时，将按照以下策略进行路由：

1. 相同 Region，相同 IDC，不在合并状态的只读 Zone 中的 OBServer。
2. 相同 Region，不同 IDC，不在合并状态的只读 Zone 中的 OBServer。
3. 相同 Region，相同 IDC，不在合并状态的读写 Zone 中的 OBServer。
4. 相同 Region，不同 IDC，不在合并状态的读写 Zone 中的 OBServer。
5. 相同 Region，相同 IDC，正在合并状态的只读 Zone 中的 OBServer。
6. 相同 Region，不同 IDC，正在合并状态的只读 Zone 中的 OBServer。
7. 相同 Region，相同 IDC，正在合并状态的读写 Zone 中的 OBServer。
8. 相同 Region，不同 IDC，正在合并状态的读写 Zone 中的 OBServer。
9. 不同 Region，不在合并状态的只读 Zone 中的 OBServer。
10. 不同 Region，不在合并状态的读写 Zone 中的 OBServer。
11. 不同 Region，正在合并状态的只读 Zone 中的 OBServer。
12. 不同 Region，正在合并状态的读写 Zone 中的 OBServer。

读写分离部署注意事项

不能仅仅在异地 Region 设置只读 Zone，否则可能导致每次请求都与 OBServer 进行建链，耗时增加。

原因是 OBProxy 内部有个策略，如果 Session 上设置了弱读，并且这个租户有只读 Zone，会判断上一次访问的 Server 是不是处于只读 Zone。如果不是的话，会关掉这个 Server Session。同时，根据上面的路由策略，如果同 Region 下是读写 Zone，异地是只读 Zone，OBProxy 始终会路由到同 Region 下的读写 Zone。因此，这样就可能会导致不断地与同 Region 下的读写 Zone 的 Server 建连，然后关闭链接，下次请求又继续重复。

5.2.3. 连接管理

针对一个客户端的物理连接，OBProxy 维持自身到后端多个 OBServer 的连接，采用基于版本号的增量同步方案维持每个 OBServer 的连接在同一状态，保证了客户端高效访问各个 OBServer。连接管理的另外一个功能是连接保持，在 OBServer 宕机/升级/重启时，客户端与 OBProxy 的连接不会断开，OBProxy 可以迅速切换到健康的 OBServer 上，对应用透明。

创建连接

OBProxy 的 Session 分为两类：Client Session 和 Server Session。Client Session 指的是客户端与 OBProxy 之间建立的连接；Server Session 指的是 OBProxy 与 OBServer 之间创建的连接。当 OBProxy 向 OBServer 转发客户端请求时，如果与 OBServer 之间没有创建连接，则需要初始化一个 Session 实例。

创建 Session 时需要做认证操作，OBProxy 无法决定该 Session 将来要访问哪些 OBServer，所以认证过程中，OBProxy 只能任意选择一台 OBServer 进行认证，将客户端发送的认证数据包转发给 OBServer，并将 OBServer 返回的结果转发给客户端，同时将客户端认证的数据包都缓存在 Client Session 内部，以便 OBProxy 与其它 OBServer 建立该 Client Session 关联的 Server Session 时，将这些认证数据表发送给 OBServer，便于与 OBServer 顺利进行认证。

存储连接

每当客户端向 OBProxy 发送请求，需要根据客户端连接信息查询获取 Client Session。在非连接池模式下，Server Session 不能脱离 Client Session 独立存在，只有根据 Client Session 来查询其关联的 Server Session，或者查询某个 Server Session 是否是与某个 Client Session 关联。

OBProxy 接收到客户端 SQL 请求时，会通过查询 partition table cache 来获取到 OBServer 的地址，然后查询 Client Session 保存的 Server Session 有没有与该 OBServer 关联的 Server Session 存在，如果存在则使用该 Server Session，如果不存在，则与该 OBServer 创建连接，并将 Server Session 加入 Client Session 关联的 Server Session 存储结构中。

事务状态维护

OBProxy 以事务来绑定 Client Session 与 Server Session。事务的第一条语句到达 OBProxy 时，OBProxy 选择一个 Server Session，并绑定到 Client Session 上，以后在整个事务处理过程中，都使用该 Server Session 转发数据到 OBServer，在事务中不能切换 Server Session，所以需要记录事务的状态到 Client Session中。

在事务开启时，记录事务开启状态到 Client Session，事务结束时将该事务状态重置。那么怎么来判断事务是否结束了呢？如果使用 autocommit，每个请求都要解析 OBServer 的返回包，如果一旦数据转发完成，就将该请求的事务状态重置。如果是长事务，只需要在 commit/rollback 请求后解析 OBServer 数据包来判断事务是否结束。

维护事务状态的目的主要是保证在事务中不切换 Server Session，也就是说，如果在事务中，OBServer 宕机，OBProxy 需要感知 OBServer 状态，并将未完结的事务结束掉。因为 MySQL 协议没有超时机制，OBProxy 必须主动通知客户端事务已经终止。OBServer 也没有对事务做同步，暂时也不能实现事务迁移功能，也就是说事务只能由接受请求的那台 OBServer 负责响应结果，换一台 OBServer 就不能处理，基于这个原因，在 OBProxy 做主备切换时，一定要保证正在处理的事务完成后才能切换，或者返回特殊错误，便于 OBProxy 通知客户端终止事务。

维护事务状态还有一个目的就是在 OBProxy 升级时，旧 OBProxy 经历一段时间，活跃的 Client Session 比较少时，采用 Kill 旧 OBProxy 的方式停止服务，停止服务也要保证在所有事务完结后才能进行，尽量减小对用户的影响。

连接变量管理

Client Session 需要记录客户端在该 Session 上所有设置过的变量，每次修改一个变量，在 Client Session 中记下修改时间。当 OBProxy 选择一个 Server Session 转发请求时，首先检查该 Server Session 上 Session 变量的修改时间是否大于 Client Session 中记录的修改时间。若小于，则意味着该 Session 没有使用最新的 Session 变量。OBProxy 会先重置该 Server Session 上的所有 Session 变量，批量将当前 Client Session 中保存的 Session 变量设置到该 Server Session 上后，再通过该 Server Session 转发请求；反之，则直接通过 Server Session 转发。

对于一些常见的 Session 变量，如：autocommit 等，客户端可能频繁设置，所以不能使用根据 modify time 来决定是否批量重置，而是每个 Server Session 存储这些常见 Session 变量的值，每次请求都对比 Client Session 中的变量值与 Server Session 中的变量值是否一致，不一致则重新设置这些变量值。

闪断避免

闪断避免指的是 OBProxy 与 OBServer 的 Server Session 异常不会被客户端感知，客户端与 OBProxy 之间的 Client Session 正常，客户端能够正常的读写数据。需要处理 Server Session 异常的情况如下：

- OBServer 发生 Leader 切换，OBProxy 还没有获取到新的 Leader。这种情况主要由 OBServer 来处理，Leader 切换一定要保证将正在处理的事务都完成，OBProxy 只是尽力保证将请求发送给数据所在的 OBServer，OBServer 发生了 Leader 切换，那么即使数据不在该 OBServer 上，该 OBServer 也需要负责处理该请求，并把结果返回给 OBProxy。
- OBServer 发生宕机，那么 OBProxy 与该 OBServer 的连接就会断开。如果该 Server Session 正在处理事务中，那么 OBProxy 需要发送一个错误响应给客户端；如果该 Server Session 处于空闲状态，只需将该 Server Session 从其对应的 Client Session 中标记删除即可。新的请求将不再使用该 Server Session 转发请求。
- OBProxy 与 OBServer 通信超时，MySQL 协议本没有超时机制，但 OBServer 有超时机制，所以 OBserver 超时会通知 OBProxy，OBProxy 将错误通知给客户端。如果 OBServer 发现 Server Session 长时间不活动，也会 Kill 该 Session，这种情况的处理参考第 2 项的处理方式。
- OBProxy 与 OBServer 之间的网络连接断开或发生网络分区，这种情况处理参考第 2 项，即使发生网络分区，如果 Server Session 上有事务正在执行，OBServer 在一段时间后终止该 Session，所以 OBProxy 在 Server Session 断开一段时间后，给客户端报错，结束为完成的事务，避免 Session 长时间被挂住。
- OBProxy 升级，新启动的 OBProxy 将负责客户端发起的新 Session，旧 OBProxy 上的 Session 数量会越来越少，当旧 OBProxy 上的 Session 数量低于某个阈值时，需要将旧 OBProxy 上的 Session 全部都终止掉（当然要保证正在处理的事务完成，长事务需要等一段时间，超时仍然没有完成也只能强制 Kill 掉），然后停止旧 OBProxy。这个过程无法完全避免客户端连接闪断。
- OBProxy 宕机，这种情况下，OBProxy 上的连接都会断掉，可能很快 OBProxy 被重新启动，或者该 OBProxy 负责的连接被其它的 OBProxy 处理，闪断无法避免。

使用 OBProxy 能够避免大部分异常情况下的连接闪断，特别是在 OBServer 发生 Leader 切换、主备集群切换、OBProxy 升级等后端维护的情况下，能保持客户端连接正常，对客户端的影响比较小。

连接复用

当 Client Session 关闭后，与 Client Session 关联的 Server Session 是否需要全部关闭？如果按常理处理，会关闭所有关联的 Server Session，但如果同一个客户端再次来连接 OBProxy，那么 Server Session 又需要重新建立一遍。创建 Session 是比较耗时的操作，特别是有应用的客户端代码，创建一个 Client Session，发送一条 SQL 请求获取到数据后关闭 Client Session，反复创建/关闭 Session 给 OBProxy 带来比较大的资源消耗。其中一种解决办法就是 Session 复用。

Session 复用是在 Client Session 关闭后，不关闭与之关联的 Server Session，而是将该 Client Session 加入 free list 中，如果该 Client 再创建 Client Session，则先查询是否该 Client 有空闲的 Client Session 可以使用，如果有，则重用该 Client Session，当需要使用与之关联的 Server Session 时，需要重置其 Session 变量，并设置新的 Session 变量后，就可以使用该 Server Session 转发客户端请求。

通过 Session 复用可以减少 OBProxy 与 OBServer 创建 Session 的频率，是一种优化的手段，并且 OBProxy 支持 Client Session 与 Server Session 不强绑定，作为一个公共的池子（连接池），所有的 Client Session 都可以从该连接池中获取可用的连接，当不再使用时，就将该连接释放归还连接池，从而可以被别的请求使用。

Kill Session 处理

MySQL 协议没有超时机制，但 MySQL 会 Kill 长时间不活跃的 Session，如果客户端一直等不到服务端的响应，会一直等待，遇到这种挂住的情况，一般 DBA 会介入，调用 MySQL 的 Kill Session 命令将 Session 关闭。

OBServer 有超时机制，一般情况下，无论是服务端宕机，还是网络分区，或者 OBProxy 与服务端的连接断开，OBProxy 都能够通知 Client Session 事务处理失败。但如果超时时间设置得过长，或者 OBProxy 处理有遗漏，Client Session 确实挂住了，那么应用机会要求做 Kill Session 操作。OBProxy 需要能够对 Kill Session 做特殊处理，不仅要处理 OBProxy 上记录的 Client Session 状态，还需要通知 OBServer关闭 Server Session。

5.2.4. 配置管理

ODP 的配置控制着 ODP 的行为，您可以修改 ODP 的配置、查看配置等。

ODP 会将配置保存到本地文件中，该文件放在 obproxy 启动目录的 `etc` 下，名称为 `obproxy_config.bin`，查看该文件也可以看到简单的配置内容，但该文件不能修改，否则会导致配置异常。

目前 ODP 的配置都是进程级别，设置后影响所有访问 ODP 的请求，无法进行租户、集群、用户等细粒度配置。

参数说明

ODP 参数可以分为以下四类：

- 支持动态修改的参数。

  此类参数修改后立即生效。详细信息，参考 支持动态修改的参数。
- 不支持动态修改的参数。

  此类参数修改后需要重启 ODP 才能生效。详细信息，参考 不支持动态修改的参数。
- 对普通用户不可见的参数。

  此类参数包括 ODP 内部使用的一些参数以及从通用配置中继承的配置。您无需配置此类参数。此类参数仅支持特殊权限用户使用内部命令修改。详细信息，参考 无需修改的参数。
- 内存级参数。

  此类参数的修改仅生效一次。详细信息，参考 内存级参数。

查看 ODP 配置

当运行 ODP 时，使用 `root@proxysys` 账号登录 ODP，密码由用户安装 ODP 时自己设置。执行以下命令获取 ODP 配置：

```
SHOW proxyconfig [LIKE '%var_name%'];
```

当需要查看所有配置时直接 `show proxyconfig` 即可，此时结果会以表格的形式展现，包含如下信息：

- name：配置的名称。
- value：配置的值。
- info：配置的介绍。通过该介绍可以了解配置的用途。
- need_reboot：是否重启生效。取值为 true 时，表示修改后需要重启才能够生效。
- visible_level：暂未使用

修改配置

您可以通过以下两种方式修改 ODP 的配置：

- 启动时在 `-o` 参数后面加上配置的内容，如 `-o <var_name>=<var_value>` 。
- 使用 `root@proxysys` 登录，通过 `alter proxyconfig set <var_name>=<var_value>` 修改配置后，对于need_reoobt 的配置不会动态生效，必须要重启。

ODP 还有一类配置，只会生效一次，参考下面表格：

参数	默认值	说明
refresh_json_config	false	用于设置是否重新从 config server 获取 json 配置。
refresh_rslist	false	用于设置是否重新从 config server 获取 rslist 配置。
refresh_idc_list	false	用于设置是否重新从 config server 获取 idc 配置。
refresh_config	false	用于设置是否重新从 MetaDB 更新所有配置。
partition_location_expire_relative_time	0	过 n 秒后所有 location cache 失效， n 为配置值

有关 ODP 的其它配置，可以通过 `show proxyconfig` 查看或参考 ODP 文档。

驱动支持

ODP 的 `root@proxysys` 操作完全兼容 MySQL 协议，使用了 MySQL 协议的 `COM_QUERY` 字段，ODP 配置管理也可以通过 Java 等 MySQL 驱动进行管理。

5.2.5. 日志与监控

ODP 运行过程中会把各类请求信息和错误信息打印到对应的日志文件中，便于您进行针对性的分析。此外，OCP 也提供了ODP 的各类监控信息。

obproxy_digest.log 日志

大于参数 `query_digest_time_threshold` 设置值（默认 100ms，主站是 2ms）的请求、错误响应请求会打到 `obproxy_digest.log` 日志中。

> **说明**
>
> `query_digest_time_threshold` 参数用于设置请求执行时间阈值，超过阈值即打印一条日志到 `obproxy_digest.log` 文件。有关该参数的详细介绍，参见 ODP 支持动态修改的参数。

日志示例

```
2020-03-18 21:26:54.871053,postmen,,,,postmen40:postmen_new0497_3279:postmen_r497,OB_MYSQL,
,postmen_push_msg_4977,COM_QUERY,SELECT,success,,SELECT    id%2C    gmt_create%2C    gmt_mo
dified%2C    msg_id%2C    principal_id%2C    app_name%2C    target_utdid%2C    biz_id%2C
host%2C    status%2C    expire_time%2C    msg_data    FROM    postmen_push_msg_4977    WHERE
principal_id = 'W/jEKoLOxnwDABWmhzjgmK1V' AND    app_name = 'KOUBEI' AND    expire_time > 1
584538014858 AND    status = 1,11765us,52us,0us,11632us,Y0-7F16DD5BEBC0,,
```

日志信息说明

上述日志示例中的信息的含义依次为：

- 日志打印时间
- 当前应用名
- TraceId
- RpcId
- 逻辑数据源名称：该信息在 ODP 的 1.x 版本中留空
- 物理库信息（cluster:tenant:database）
- 数据库类型：取值为 OB 或 RDS
- 逻辑表名：该信息在 ODP 的 1.x 版本中留空
- 物理表名
- SQL 命令：COM_QUERY、COM_STMT_PREPARE 等
- SQL 类型（CRUD）
- 执行结果：取值为 success 或 failed
- 错误码：当执行结果为 success 时，该信息为空
- SQL 执行总耗时：单位为 ms，包括内部 SQL 执行耗时
- 预执行时间
- 链接建立时间
- 数据库执行时间
- 当前线程名：ODP 的内部线程 ID
- 系统穿透数据：系统灾备信息等
- 穿透数据

obproxy_stat.log 日志

OBProxy 请求统计日志，默认每分钟输出一次。

日志示例

```
2020-03-18 21:26:59.504487,postmen,,postmen50:postmen_new0563_3279:postmen_r563,OB_MYSQL,SE
LECT,success,,1,0,0,0,9.383ms,0.029ms,9.263ms
2020-03-18 21:26:59.504509,postmen,,postmen50:postmen_new0528_3279:postmen_r528,OB_MYSQL,SH
OW,success,,2,0,0,0,14.280ms,0.056ms,14.067ms
2020-03-18 21:26:59.504545,postmen,,postmen30:postmen_new0363_3279:postmen_r363,OB_MYSQL,SE
LECT,success,,5,0,0,0,0.158ms,0.000ms,0.000ms
2020-03-18 21:26:59.504563,postmen,,postmen50:postmen_new0561_3279:postmen_r561,OB_MYSQL,SE
LECT,failed,1054,1,0,0,0,8.487ms,0.048ms,8.281ms
2020-03-18 21:26:59.504604,postmen,,postmen50:postmen_new0561_3279:postmen_r561,OB_MYSQL,OT
HERS,success,,1,0,0,0,27.148ms,17.902ms,9.102ms
```

日志信息说明

上述日志示例中的信息的含义依次为：

- 日志打印时间
- 当前应用名
- 逻辑数据源名称 ： 该信息在 ODP 的 1.x 版本中留空
- 物理库信息（cluster:tenant:database）
- 数据库类型：取值为 OB 或 RDS
- SQL 类型(CRUD)
- 执行结果：取值为 success 或 failed
- 错误码：当执行结果为 success 时，该信息为空
- 总请求数量
- 30 ms ~ 100 ms 请求数量
- 100 ms ~ 500 ms 请求数量
- 大于 500 ms 请求数量
- 执行总耗时：单位为 ms，包括内部 SQL 执行耗时
- 预执行时间
- 数据库执行时间

obproxy_slow.log 日志

大于参数 `slow_query_time_threshold` 设置值（默认 500ms）的请求，会打到 `obproxy_slow.log` 日志中。

> **说明**
>
> 参数 `slow_query_time_threshold` 用于设置慢请求执行时间阈值。超过阈值即打印一条日志到 `obproxy_slow.log` 文件。有关该参数的详细介绍，参见 ODP 支持动态修改的参数。

日志示例

```
2020-03-16 21:31:32.125967,postmen,,,,postmen20:postmen_new0200_3279:postmen_r200,OB_MYSQL,
,,COM_LOGIN,,success,,,507680us,507049us,0us,515us,Y0-7F16DD5743A0,,
2020-03-16 21:31:49.561865,postmen,,,,postmen50:postmen_new0580_3279:postmen_r580,OB_MYSQL,
,,COM_LOGIN,,success,,,1258599us,1249321us,0us,9130us,Y0-7F16DB9752C0,,
```

日志信息说明

上述日志示例中的信息的含义依次为：

- 日志打印时间
- 当前应用名
- TraceId
- RpcId
- 逻辑数据源名称：该信息在 ODP 的 1.x 版本中留空
- 物理库信息（cluster:tenant:database）
- 数据库类型：取值为 OB 或 RDS
- 逻辑表名：该信息在 ODP 的 1.x 版本中留空
- 物理表名
- SQL 命令：COM_QUERY、COM_STMT_PREPARE 等
- SQL 类型（CRUD）
- 执行结果：取值为 success 或 failed
- 错误码：当执行结果为 success 时，该信息为空
- SQL 执行总耗时：单位为 ms，包括内部 SQL 执行耗时
- 预执行时间
- 链接建立时间
- 数据库执行时间
- 当前线程名：ODP 的内部线程 ID
- 系统穿透数据：系统灾备信息等
- 穿透数据

obproxy_error.log 日志

执行错误的请求会打印到该日志中，包括 OBProxy 自身错误和 OBServer 返回错误。

日志示例

```
2020-03-18 21:28:10.497945,postmen,,,,postmen70:postmen_new0781_3279:postmen_r781,OB_MYSQL,
,postmen_connection_info_7819,COM_QUERY,INSERT,failed,1054,INSERT INTO  postmen_connection
_info_7819 (   gmt_create%2C gmt_modified%2C app_name%2C principal_id%2C utdid%2C link_inf
o_key%2C product_id%2C product_version%2C   os_type%2C os_version%2C network%2C brand%2C m
odel%2C last_active_time%2C status%2C ext_attr%2C worker_id%2C sid%2C   postmen_ip%2C link
_version%2C protocol_type%2C zone_name%2C sync_version%2C background_time%2C app_status%2C
device_id%2C   push_switch  )  VALUES  ( now(6)%2C now(6)%2C 'KOUBEI'%2C '2088212505319
787'%2C 'W/jEKoLOxnwDABWmhzjgmK1V'%2C '11.233.24.240_3883D71A4ECCAE0F_AB18'%2C 'KOUBEI_APP_
ANDROID'%2C '7.1.87.000001'%2C  'android'%2C '9'%2C 'wifi'%2C 'xiaomi'%2C 'MI 8 SE'%2C 158
4538090493%2C 1%2C null%2C '11.233.24.240_3883D71A4ECCAE0F'%2C 43800%2C  '11.235.166.113'%
2C 1%2C 'MMTP'%2C 'RZ99S'%2C 6%2C 1584538090477%2C 1%2C 'W/jEKoLOxnwDABWmhzjgmK1V'%2C  '1'
)  ON DUPLICATE KEY UPDATE  gmt_modified = now(6)%2C  utdid = 'W/jEKoLOxnwDABWmhzjgmK1V'
%2C  link_info_key = '11.233.24.240_3883D71A4ECCAE0F_AB18'%2C  product_id = 'KO,1138us,29
us,0us,984us,Y0-7F16DD5BDF00,,,Unknown column 'push_switch' in 'field list'
```

日志信息说明

上述日志示例中的信息的含义依次为：

- 日志打印时间
- 当前应用名
- TraceId
- RpcId
- 逻辑数据源名称：该信息在 ODP 的 1.x 版本中留空
- 物理库信息（cluster:tenant:database）
- 数据库类型：取值为 OB 或 RDS
- 逻辑表名：该信息在 ODP 的 1.x 版本中留空
- 物理表名
- SQL 命令：COM_QUERY、COM_STMT_PREPARE 等
- SQL 类型（CRUD）
- 执行结果：取值为 success 或 failed
- 错误码：当执行结果为 success 时，该信息为空
- SQL 执行总耗时：单位为 ms，包括内部 SQL 执行耗时
- 预执行时间
- 链接建立时间
- 数据库执行时间
- 当前线程名：ODP 的内部线程 ID
- 系统穿透数据：系统灾备信息等
- 穿透数据
- 错误详情

obproxy_limit.log 日志

OBProxy 限流日志，如果发生限流，会将被限流的请求打印到日志中。

日志格式：

```
限流状态（RUNNING/OBSERVE） SQL 限流规则名称
```

日志示例

```
2020-03-18 21:26:54.871053,postmen,,,,postmen40:postmen_new0497_3279:postmen_r497,OB_MYSQL,
,postmen_push_msg_4977,COM_QUERY,SELECT,RUNNING,SELECT id%2C gmt_create%2C gmt_mod
ified%2C msg_id%2C principal_id%2C app_name%2C target_utdid%2C biz_id%2C
host%2C status%2C expire_time%2C msg_data FROM postmen_push_msg_4977 WHERE
principal_id = 'W/jEKoLOxnwDABWmhzjgmK1V' AND app_name = 'KOUBEI' AND expire_time > 1
584538014858 AND status = 1,LIMIT_RULE_
```

日志信息说明

上述日志示例中的信息的含义依次为：

- 日志打印时间
- 当前应用名
- TraceId
- RpcId
- 逻辑数据源名称：该信息在 ODP 的 1.x 版本中留空
- 物理库信息（cluster:tenant:database）
- 数据库类型：取值为 OB 或 RDS
- 逻辑表名：该信息在 ODP 的 1.x 版本中留空
- 物理表名
- SQL 命令：COM_QUERY、COM_STMT_PREPARE 等
- SQL 类型（CRUD）
- 限流状态（RUNNING/OBSERVE）
- SQL 限流规则名称

日志示例：

ODP 监控

除了通过 ODP 的各类日志之外，您也可以通过 OceanBase 提供的数据库管理运维产品 OCP 来查看 ODP 的各类监控信息。

OBProxy 集群性能监控提供了服务监控和系统监控功能。

服务监控

服务监控提供了 ODP 集群维度、ODP 关联的 OB 集群维度以及单个 ODP IP 维度的 **每秒事务数**、**请求数**、**客户端连接数**、**服务端链接数**、**SQL 处理耗时**、**每秒 ERROR 响应包数**、**路由表** 和 **网络请求字节数** 信息。

系统监控

系统监控提供了 ODP 集群维度以及单个 ODP IP 维度的 **Linux 系统负载** 、**CPU 使用率**、**平均每秒 IO 次数**、**平均每次 IO 耗时**、**平均每秒 IO 数据量**、**网络吞吐率**、**内存** 和 **磁盘** 信息。

有关 ODP 监控的详细信息，参见《OCP 用户指南》中 **ODP 集群性能监控** 章节。

5.3. 数据库驱动

5.3.1. 数据库驱动概述

在 OceanBase 数据库 MySQL 模式下，用户可以直接使用 MySQL 官方提供的 Connector 来使用 OceanBase 数据库（暂不支持 8.0 的驱动），在 OceanBase 数据库 Oracle 模式下，需要使用 OceanBase 自研的数据库驱动。下面介绍 OceanBase 数据库提供的几种驱动。

OBCI 驱动

OBCI 驱动是兼容 Oracle 数据库 OCI 接口的 C 语言的驱动。基于 Oracle OCI 开发的用户和应用可以使用 OBCI 驱动，平滑的迁移使用 OceanBase 数据库的 Oracle 模式。

有关 OBCI 的详细介绍，请参见 OBCI。

OceanBase Connector/J

JDBC（Java Database Connectivity）是 Java 应用程序访问数据库的标准 API（应用程序编程接口），数据库驱动的实现会将 JDBC 标准编程接口转换成对应数据库厂商的 SQL 实现。OceanBase 数据库 JDBC 驱动兼容 JDBC 4.0、4.1、4.2 标准。OceanBase 提供了自研的驱动，使用该驱动可以同时使用 OceanBase 数据库的 MySQL 模式和 Oracle 模式。

有关 OceanBase Connector/J 的详细介绍，请参见 OceanBase Connector/J。

OceanBase Connector/C

OceanBase Connector/C 是一个基于 C/C++ 的 OceanBase 客户端开发组件，支持 C API Lib 库。

允许 C/C++ 程序以一种较为底层的方式访问 OceanBase 分布式数据库集群，以进行数据库连接、数据访问、错误处理和 Prepared Statement 处理等操作。

有关 OceanBase Connector/C 的详细介绍，请参见 OceanBase Connector/C。

5.3.2. OBCI

OBCI（OceanBase Call Interface）是 OceanBase 架构环境中，与 Oracle OCI 兼容的 C 语言接口调用工具，它提供了与 Oracle OCI 完全兼容的功能特性。

OBCI 介绍

OCI（Oracle Call Interface）是 Oracle 公司开发的一个 C 端应用程序开发工具套件，是一个能够访问 Oracle 数据库的服务器，并通过控制各类 SQL 语句的执行，进而控制数据库所有操作的应用程序接口（API）。它支持 SQL 所有的数据定义、数据操作、查询和事务管理等操作，支持 C 和 C++ 的数据类型、调用、语法和语义。它提供了一组可对 Oracle 数据库进行存取的接口子例程（函数）。 OBCI 是参照 OCI 的接口标准，结合自身的特点，为开发人员提供向 Oracle 兼容功能的一款接口产品。在接口能力及用户行为上与 Oracle OCI 处理保持一致，满足开发用户在 C 端（OTL、Tuxdo、ECOB 等）快速访问 OceanBase Oracle 模式集群的需求。

相关概念如下：

- OBCI：C 端访问 OceanBase 数据库的一套应用程序接口。
- OCI：C 端访问 Oracle 数据库的一套应用程序接口。

- libobclient/libobclnt：OBCI 依赖 C 端操作数据库一套访问接口。
- OceanBase Oracle模式：OceanBase 数据库兼容 Oracle 语法的处理模式。

OBCI 主要功能

OBCI 使您可以使用 C 语言来访问操作 OceanBase 数据库（ Oracle 模式）中的数据。它以动态链接库（ OCI 库）的形式提供了标准数据库访问和索引功能，应用程序在运行阶段链接此库就可以使用这些功能。

OBCI提供的主要功能如下：

- OBCI 提供一套标准的 OceanBase Oracle 数据库访问、处理接口，主要包括数据的定义、数据管理、查询处理、事务控制等。
- OBCI 的 API 接口可用于支持可扩展、多线程的应用。
- 支持 SQL 访问函数。可用于管理数据库访问，处理 SQL 语句，和操作从 OceanBase 数据库服务器获取的对象。
- 支持数据类型映射和操作函数。可用于操作 OceanBase 数据类型属性。
- 支持数据加载函数。可直接将数据加载到数据库中而无需使用 SQL 语句。
- 支持外部过程函数。可以在 PL/SQL 主体中指定 C 语言回调函数。

OBCI 的优势

OBCI 相对于直接通过数据库协议或直接访问 C 接口进行操作 OceanBase 数据库等的有如下优势：

- 高度封装对数据库访问操作，简化业务实现逻辑。
- 可以支持第三方框架能力（Tuxdo、OTL 等）。
- 可以使用 Connection Pool、Statement cache，提高程序的扩展性。
- 能够支持动态的绑定参数及结果回调函数处理，便于程序动态处理参数及结果数据。
- 提供暴露 Meta 信息的接口。
- 提供 DML（ INSERT、UPDATE、DELTE ）中操作访问 Array 等复杂类型数据的能力。
- 支持 Prefetch 能力，可以减少与后端的交互。
- 保证线程安全，减少业务对互斥量的使用。

支持的 SQL

- Data Definition Language (DDL)
- Control Statments:
  - Transaction Control
  - Session Control
  - System Control
- Data Manipulation Language（DML)
- Queries
- PL SQL/Non-block

数据类型

OBCI 支持 OceanBase Oracle 模式下的内部数据类型和 C 语言中 `<oci.h>` 定义的外部类型（兼容Oracle OCI）。

内部类型

内部类型指的是 OceanBase 数据库 Oracle 模式下可使用的数据类型。

OBCI 支持的常见内部类型如下表所示：

名字	描述	限制
char	固定长度字符串	最大长度 2000
varchar2	可变长度字符串	最大长度 32767
number	数值类型	精度取值范围 1 ~ 38 位数取值范围 -84 ~ 127
int	整形数值	最大值 38 位
binary_float	32 位浮点数	最小值：1.17549E-38F 最大值：3.40282E+38F
binary_double	64 位浮点数	最小值：2.22507485850720E-308 最大值：1.79769313486231E+308
date	日期类型	YYYY-MM-DD HH:MI:SS
timestamp	时间类型	YYYY-MM-DD HH:MI:SS [.FFFFFFFFF]

外部类型

外部类型是用于指定宿主变量存储数据的类型，当输入数据到数据库时，OBCI 会将输入的宿主变量的外部类型和内部数据类型进行转换。当输出数据到外部程序时，OBCI 会将数据库中表的内部数据类型和输出的宿主外部数据类型进行转换。

OBCI 支持的常用外部类型及其对应关系如下表所示：

外部类型	编码	宿主变量数据类型	OBCI 类型
VARCHAR2	1	char[n]	SQLT_CHR

外部类型	编码	宿主变量数据类型	OBCI 类型
NUMBER	2	unsigned char[21]	SQLT_NUM
8-bit signed INTEGER	3	signed char	SQLT_INT
16-bit signed INTEGER	3	signed short, signed int	SQLT_INT
32-bit signed INTEGER	3	signed int, signed long	SQLT_INT
FLOAT	4	float, double	SQLT_FLT
LONG	8	char[n]	SQLT_LNG
NULL-terminated STRING	5	char[n+1]	SQLT_STR
LONG	8	char[n]	SQLT_LNG
VARCHAR	9	char[n+sizeof(short integer)] SQLT_VCS	VARCHAR
DATE	12	unsigned char[n]	SQLT_BIN
RAW	23	char[7]	SQLT_DAT
CHAR	96	char[n]	SQLT_AFC
REF	110	OCIRef	SQLT_REF
Character LOB descriptor	112	OCILobLocator (see note 2)	SQLT_CLOB
Binary LOB descriptor	113	OCILobLocator (see note 2)	SQLT_BLOB
ANSI DATE descriptor	184	OCIDateTime *	SQLT_DATE
TIMESTAMP descriptor	187	OCIDateTime *	SQLT_TIMESTAMP

外部类型	编码	宿主变量数据类型	OBCI 类型
TIMESTAMP WITH TIME ZONEdescriptor	188	OCIDateTime *	SQLT_TIMESTAMP_TZ
TIMESTAMP WITH LOCAL TIME ZONEdescriptor	232	OCIDateTime *	SQLT_TIMESTAMP_LTZ

数据类型转换

在 OBCI 程序中，宿主变量（ Host Variables ）使用了外部数据类型，当从数据库读取数据到外部变量或者将 外部数据存储到数据库中时，会发生数据类型转换。

当前支持的数据类型转换如下表所示：

	VARCHAR2/CHAR	INT	NUMBER	FLOAT	BINARY_FLOAT	BINARY_DOUBLE	DATE	TIMESTAMP
CHAR/UNSIGNED CHAR	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT		
INT/UNSIGNED INT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT		
LONG/UNSIGNED LONG	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT		
LONG LONG/UNSIGNED LONG LONG	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT		
FLOAT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT		
DOUBLE	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT		

	VARCHAR2/CHAR	INT	NUMBER	FLOAT	BINARY_FLOAT	BINARY_DOUBLE	DATE	TIMESTAMP
CHAR[n]/VARCHAR[n]/CHAR*	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN

> 说明
>
> - 对于一些错误下表未标明。如 `VARCHAR2` 转 C 语言 `CHAR` 时可能会发生溢出或有非法数字等错误。
> - IN 表示支持从 C 语言转成数据库类型，也就是写入数据到数据库。OUT 表示支持数据库类型转成 C 语言，也就是读取数据。

对象及函数接口

OBCI 对象接口

OBCI 支持的 OCI 内部的常规对象接口如下表所示：

对象	对象描述
OCIEnv	OCI 环境句柄
OCIError	OCI 错误处理句柄，获取处理过程中错误信息的对象
OCISvcCtx	OCI 服务上下文对象（内存、连接等）
OCIStmt	OCI 语句处理对象
OCIBind	OCI 参数变量信息
OCIDefine	OCI 结果变量信息
OCIDescribe	OCI 定义类型信息
OCIServer	OCI 后端 OBServer 对象

对象	对象描述
OCISession	OCI 连接 Session 信息
OCITrans	OCI 事务对象
OCICPool	OCI CPool 连接池对象
OCIAuthInfo	OCI 连接验证信息
OCILobLocator	OCI Lob 对象，Lob 函数处理
OCIParam	OCI 参数对象，可以获取对象 Meta 信息
OCIRowid	OCI OB Rowid 对象
OCIDate	OCI 日期对象
OCIDateTime	OCI 时间戳对象
OCIInterval	OCI 时间间隔对象
OCINumber	OCI 高精度数据对象
OCIString	OCI 字符串对象

OBCI 函数接口

初始化相关

处理初始化、连接认证等相关的函数如下表所示：

函数	函数描述
OCIEnvCreate	创建并初始化环境句柄。

函数	函数描述
OCIEnvNlsCreate	创建并初始化一个环境句柄，以使其在 OCI 函数下工作。它是 OCIEnvCreate（）函数的增强版本。
OCIEnvInit	分配并初始化环境句柄。
OCIInitialize	初始化 OCI 应用环境， OCI 会在这个函数中初始化内部的全局变量和加载一些配置信息。
OCILogoff	断开通过 OCILogon/OCILogon 与服务器建立的连接。
OCILogon	根据用户名和密码，登录到一个指定的数据库服务上，并初始化相关的上下文句柄。
OCILogon2	根据用户名和密码，登录到一个指定的数据库服务上，并初始化相关的上下文句柄，可以使用CPOOL连接进行处理。
OCIServerAttach	附加一个数据库服务到指定的连接句柄上。
OCIServerDetach	解除连接句柄和数据库服务名之间的关联。
OCISessionBegin	使用登录信息在指定的上下文句柄上打开与数据库服务的连接。
OCISessionEnd	结束 OCISessionBegin 函数中上下文句柄与数据库服务之间的连接。
OCIPing	确定连接和服务处于活动状态。只有服务运行且连接存在时才成功。
OCITerminal	终止线程中处理

对象初始化相关

Handle 和 Descriptor 的相关函数如下表所示：

函数	函数描述
OCIAttrGet	获取句柄上的属性值。

函数	函数描述
OCIAttrSet	设置句柄的属性。
OCIDescriptorAlloc	分配一个存贮大字段描述符句柄。
OCIDescriptorFree	释放 OCIDescriptorAlloc 生成的描述符。
OCIHandleAlloc	分配和初始化各种句柄。
OCIHandleFree	释放由 OCIHandleAlloc 所分配的句柄。
OCIParamGet	获取描述符句柄上指定位置的描述符句柄。

参数绑定结果处理相关

Bind、 Define 和 Describe 等参数、结果集空间的相关函数如下表所示：

函数	函数描述
OCIBindArrayOfStruct	以数组方式进行参数绑定。
OCIBindByName	按参数名称绑定 SQL 语句中的参数（同名只需要绑定一次）。
OCIBindByPos	按参数在 SQL 语句中出现的位置进行绑定。
OCIDefineArrayOfStruct	以数组方式进行结果列绑定。
OCIDefineByPos	按位置来绑定查询返回结果集中每一列的取值空间。

请求语句处理相关

语句处理相关函数如下表所示：

函数	函数描述
OCIStmtPrepare	准备一条 SQL 语句，初始化Stmtp对象，随后调用 OCIStmtExecute 来执行。

函数	函数描述
OCIStmtExecute	执行准备好的语句。
OCIStmtFetch	提取 SQL 生成的结果集中的行集。当执行一条查询以后，可以多次调用该函数来返回结果集中的所有行，直到该函数返回 OCI_NO_DATA。
OCIStmtRelease	释放通过调用 OCIStmtPrepare2（）获得的语句句柄。

事务处理相关

事务处理相关函数如下表所示：

函数	函数描述
OCITransStart	启动事务。
OCITransCommit	提交 SQL 的执行动作。
OCITransRollback	回滚 SQL 的执行动作。

数据类型相关

各种数据类型相关的函数如下表所示：

数据对象	函数	函数描述
OCIString	OCIStringAllocSize	获取以代码点（Unicode）或字节为单位的字符串内存分配大小。
	OCIStringAssign	将一个字符串分配给另一个字符串。
	OCIStringAssignText	将一个字符串分配给另一个字符串。
	OCIStringPtr	获取指向给定字符串文本的指针。
	OCIStringSize	获取指向给定字符串文本的大小。
	OCIStringResize	获取指向给定字符串文本的大小。

数据对象	函数	函数描述
OCILoblocator	OCILobGetLength	返回大字段的长度，按字节计算。
	OCILobRead	读取某个大字段中指定长度的内容。
	OCILobWrite	连续写入内容到一个大字段存贮描述符中。
	OCILobLocatorIsInit	判断给定的 LOB/BFILE 句柄是否已经初始化。
	OCILobOpen	打开一个 LOB/BFILE 对象
	OCILobClose	关闭一个 LOB/BFILE 对象
	OCILobIsOpen	测试一个 LOB/FILE 对象是否已打开。
	OCILobTrim	将 LOB 值截短到较短的长度。
	OCILobTrim2	将 LOB 值截断为较短的长度。OceanBase 目前仅支持最大 48M 的 LOB。
OCIDate	OCIDateSysDate	获取客户端的当前系统日期和时间。
	OCIDateToText	将日期类型转换为指定格式字符串。
	OCIDateFromText	根据指定格式将字符串转换为日期类型。
	OCIDateTimeToText	根据指定的格式将给定的日期转换为字符串。
	OCIDateTimeFromText	根据指定的格式将给定的日期转换为字符串。
	OCIDateTimeConstruct	构造一个日期时间。

OCIDateTime 数据对象	函数	函数描述
	OCIDateTimeGetTime	从日期时间值中获取时间（小时、分钟、秒和小数秒）。
	OCIDateTimeGetDate	获取日期时间值的日期（年，月，日）部分。
	OCIDateTimeGetTimeZoneOffset	取日期时间值的时区（小时，分钟）部分。
OCINumber	OCINumberToInt	将 Oracle `NUMBER` 类型转换为整数。
	OCINumberToReal	将 Oracle `NUMBER` 类型转换为实数。
	OCINumberToText	根据指定格式将 Oracle `NUMBER` 转换为字符串。
	OCINumberIsInt	测试 OCINumber 是否为整数。
	OCINumberFromText	将字符串转换为 Oracle NUMBER。

其他函数

常用的一些其他函数如下表所示：

函数	函数描述
OCIBreak	该调用将立即终止当前的（异步）运行执行与服务器关联的 OBCI 功能。
OCIClientVersion	返回运行时客户端库的版本号。
OCIServerVersion	返回运行时服务端库的版本号。
OCIErrorGet	在提供的缓冲区中返回错误消息和 OceanBase 错误代码。
OCIPasswordChange	修改账户密码。

函数	函数描述
OCIPing	对服务器进行往返通话，以确认连接和服务器处于活动状态。

更多信息

有关 OBCI 的详细介绍和使用说明，参见《 OceanBase C 语言调用接口》。

5.3.3. OceanBase Connector/J

OceanBase 数据库通过 OceanBase Connector/J 为基于 Java 开发的客户端应用程序提供连接。Java 数据库连接器（ Java Database Connectivity，JDBC ）提供了 Java 连接关系数据库的接口，是一种 Java 标准。OceanBase Connector/J 是一种实现 JDBC API 的驱动程序。

JDBC 标准由 Sun Microsystems 定义，通过标准java.SQL接口实现，支持各个提供程序使用自己的 JDBC 驱动程序来实现和扩展标准。JDBC 是基于 X/Open SQL 的调用级别接口（ Call Level Interface，CLI ）。

主要功能

OceanBase Connector/J 主要模块的功能如下：

- DriverManager：用于加载驱动程序（ Driver ），并根据调用请求返回相应的数据库连接（ Connection ）。
- Driver：驱动程序，会被加载到 DriverManager 中，负责处理请求并返回相应的数据库连接（ Connection ）。
- Connection：数据库连接，负责与数据库进行通讯，提供 SQL 执行以及事务处理的 Connection 环境，创建和执行 Statement。
- Statement：
  - Statements：用以单次执行 SQL 查询和更新。
  - PreparedStatement：用以执行已缓存的 Statement，其执行路径已经预先确定，支持重复执行以提高执行效率。
  - CallableStatement：用以执行数据库中的存储过程。
- SQLException：显示在与数据库创建、关闭连接，或者执行 SQL 语句时发生的错误。

OceanBase Connector/J 驱动程序

OceanBase Connector/J 驱动程序属于 JDBC Type 4 驱动类型，可以通过本地协议直接与数据库引擎通信。OBProxy/OBServer 支持 OceanBase Connector/J 驱动程序，同时完全兼容 MySQL 原生的 JDBC 驱动（ MySQL Connector Java ）。OceanBase Connector/J 驱动程序完全兼容 MySQL JDBC 的使用方式，可以自动识别 OceanBase 数据库的运行模式是 MySQL 还是 Oracle，并在协议层同时兼容这两种模式。

OceanBase Connector/J 驱动程序兼容 OB2.0 协议。

> **注意**
>
> OBServer 会依据 JDBC 驱动连接时的租户名称判断运行模式为 MySQL 或者 Oracle。Oracle 模式的租户只允许使用 Oracle 兼容的 SQL 语法。

除了支持标准的 JDBC 应用程序编程接口（ API ），OceanBase Connector/J 还兼容 Oracle Driver 的使用方式，OBServer 的 Oracle 模式兼容 Oracle 的大部分语法。

MySQL Protocol

MySQLProtocol 实现 MySQL 驱动和 MySQLServer 之间的通信，它通过相同的序列化和反序列化规定，让数据的发送端和接收端可以准确高效地对数据进行 PRC 调用。WireShark 软件支持按照 MySQLProtocol 的格式解码，所以抓包相对来说很容易，如下图所示：

Protocol	Length	Info
MySQL	144	Server Greeting proto=10 version=5.7.25
MySQL	408	Login Request user=tester@oracle db=UNITTESTS
MySQL	226	Response OK
MySQL	141	Request Query { set autocommit=1, sql_mode = concat(@@sql_mode
MySQL	77	Response OK
MySQL	85	Request Query { set names utf8 }
MySQL	77	Response OK
MySQL	236	Request Query { SELECT @@max_allowed_packet,@@system_time_zone
MySQL	487	Response
MySQL	109	Request Query { select count(*) from V$TIMEZONE_NAMES; }
MySQL	140	Response
MySQL	90	Request Query { drop table blobtest }
MySQL	77	Response OK
MySQL	110	Request Query { create table blobtest(c1 varchar2(200)) }
MySQL	77	Response OK
MySQL	105	Request Prepare Statement
MySQL	118	Response
MySQL	90	Request Unknown (162)
MySQL	90	Request Unknown (162)
MySQL	90	Request Unknown (162)
MySQL	84	Request Execute Statement
MySQL	354	Response Error 600
MySQL	77	Response OK

可以比较清晰的查看到驱动和 OBServer（ OBProxy ）之间的交互过程和发包信息。

因为 MySQLProtocol 的存在，只要支持了它，并且表现上和 MySQLServer 一致我们就可以做到兼容 MySQL JDBC 及其他基于 MySQLProtocol 的驱动（如：MaraiDB JDBC ），所以 OBServer 是兼容 MySQL JDBC 的，且在 MySQL JDBC、MariaDB JDBC 的基础上开发了支持 OceanBase 的 Oracle Mode 的 OBJDBC。

文本协议和二进制协议

MySQL Protocol 有一个很重要的概念就是文本协议和二进制协议，简单的来说可以总结为：

- 文本协议：将 JDBC 的所有动作转换为标准的 SQL 语句，将 SQL 语句以字符串的形式发送到 OBServer，OBServer 执行后再返回相应的结果。
- 二进制协议：通过协议中的 COM_STMT_PREPARE 命令来 Prepare 相应的 SQL 语句，OBServer 解析 SQL 语句，返回相应参数的描述信息，驱动方面填充相应的数据信息，再发送到 OBServer。

文本协议的好处是不用执行一次 Prepare，而是只执行相应的 SQL；二进制协议的好处是可以 Prepare 后多次执行。

与之紧密相关的配置是 useServerPrepStmts，当它为 False 的时候采用文本协议，为 True 的时候会采用二进制协议。

如：

```
PreparedStatement ps = conn.prepareStatement("select * from selecttest");
ResultSet rs = ps.executeQuery();
```

如上的测试语句，在 useServerPrepStmts=true 和 useServerPrepStmts=false 时表现是不同的：

```
Request Prepare Statement
Response
Request Execute Statement
Response
```

通过抓包数据可以看到 useServerPrepStmts=true 的时候做的操作是 prepare + execute，而 useServerPrepStmts=false 时，是执行了 SQL 语句。

```
Request Query {select * from selecttest}
Response
```

OBJDBC 版本差异

- 1.X 基于 MySQL JDBC 开发，开源协议为 GPL，JDK 版本要求为 JDK7~8。
- 2.X 基于 MaraiDB JDBC 开发，开源协议为 LGPL，JDK 要求 1.8。

JDBC 标准实现

Connection 相关

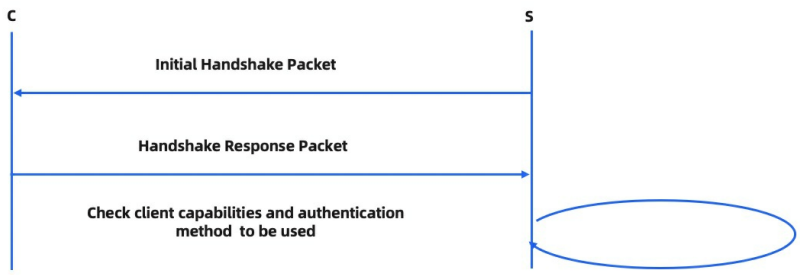
建立连接

OceanBase 数据库通过租户来区分 Oracle 模式还是 MySQL 模式，MySQL 模式的情形和 MySQL Server 表现基本一致，也可以使用 MySQL JDBC 直连 OBServer，这里我们不讨论 MySQL 模式情况，只分析 Oracle 模式的情形。

MySQL 建连接的交互过程称之为 Handshake，整个流程如下（这里描述的流程是在底层的 TCP 连接创建之后）：

1. OBServer 端主动向 Client 端发送 Handshake Packet， 包里携带里 OBServer 端的版本、字符集和支持的能力等信息。
2. Client 端根据 Handshake Packet 携带的信息构造 Handshake Response Packet， 发给 OBServer 去做认证。
3. 认证成功之后，OBServer 返回一个 OK Packet 给 Client，此时建立连接交互完成。

OK 包中携带了 2 个字节的 Status Flags，用来表示当前连接的状态信息，一共 16 位，每一位代表一种功能。



在 Handshake Response 包中，开头的 4 个字节 capability flags 描述了 Client 所支持的能力，长度是 32 bit，每一个 bit 代表一种功能，目前 MySQL 用了前面 26 位，我们使用第 27 位来表示 Client 是否支持 Oracle 模式，具体的内容如下：

```
const (
 clientLongPassword clientFlag = 1 << iota
 clientFoundRows
 clientLongFlag
 clientConnectWithDB
 clientNoSchema
 clientCompress
 clientODBC
 clientLocalFiles
 clientIgnoreSpace
 clientProtocol41
 clientInteractive
 clientSSL
 clientIgnoreSIGPIPE
 clientTransactions
 clientReserved
 clientSecureConn
 clientMultiStatements
 clientMultiResults
 clientPSMultiResults
 clientPluginAuth
 clientConnectAttrs
 clientPluginAuthLenEncClientData
 clientCanHandleExpiredPasswords
 clientSessionTrack
 clientDeprecateEOF
 clientSupportOracleMode = 1 << 27
)
```

OBServer 接收到 Handshake Response 包之后，会解析出 username 里的租户信息，如果是 Oracle 租户，并且 capability flags 中的第 27 位是 1，会返回一个 OK 包给 Client， 否则会返回一个 Error 包给 Client，错误信息是“Oracle tenant for current client driver is not supported”。在返回的 OK 包中，有 2 个字节的 Status Flags，代表当前连接的状态信息，一共 16 位，其中第 3 位 MySQL 没有使用，我们直接复用这一位来表示当前连接是 Oracle 模式。

初始化 Session 变量

OBJDBC 与 OBServer/OBProxy 建连成功后，会进行一系列的变量初始化。初始化的变量的值是从默认的配置得到的，可以被 URL 中的配置覆盖。

一般初始化时会执行 SQL 来做变量的初始化，示例如下：

```
set autocommit=1, sql_mode = concat(@@sql_mode,',STRICT_TRANS_TABLES')
set names utf8
```

初始化本地变量

OBJDBC 通过存储某些变量的值来和 OBServer 的 Session 保证一致，如：ReadOnly 和 txIsolation 等。在设置完 Session 变量后，会查询相应的变量值来初始化这些 Local Variables，并且在之后调用 API 的时候也修改相应的值以保证 OBServer 端和 JDBC 端的信息一致。

示例如下：

```
SELECT @@max_allowed_packet,@@system_time_zone,@@time_zone,@@auto_increment_increment,@@tx_
isolation AS tx_isolation,@@session.tx_read_only AS tx_read_only from dual
select count(*) from V$TIMEZONE_NAMES // 该语句比较特殊是为了确定server是否有导入时区表
```

Connection 常用接口

Connection 作为一个最基础的接口主要创建其他接口或者获得 Connection 相关的信息。最常用的接口如下：

- createStatement
- prepareStatement
- prepareCall
- getMetaData
- commit/rollback

Statement 功能

Statement 是功能比较单一的接口，主要就用来执行 SQL 语句，也可以用来查询数据，和使用 OBClient 等命令行工具执行 SQL 没有本质的区别。

Statement 的常用接口如下：

- execute
- executeQuery
- setQueryTimeout
- addBatch
- executeBatch
- setFetchSize

PrepareStatement 功能

PrepareStatement JDBC 是最常用的接口之一，他可以实现 Statement 几乎所有的功能。
PrepareStatement 表示预编译的 SQL 语句的对象，SQL 语句被预编译并存储在对象中，被封装的 SQL 语句代表某一类操作，SQL 语句中允许包含动态参数“？”，在执行时可以为“？”动态设置参数值。

值得注意的是，要实现真正的预编译效果需要设置 `url option` 中的 `useServerPreparedStmts = true` ，否则即使使用 PrepareStatement 也会降级成文本协议而不是二进制协议。

PrepareStatement 的常见用法如下：

```
ps = conn.prepareStatement("insert into pstest values(?,?)");
ps.setString(1,"string1");
ps.setInt(2,1);
ps.execute();
ps.setString(1,"string2");
ps.setInt(2,2);
ps.execute();
```

预编译之后的数据只需要发送 Execute 就可以，这大大提升了效率。

PrepareStatement 的常用接口如下：

- addBatch
- executeBatch
- setFetchSize

CallableStatement 功能

CallableStatement 可以用来执行 Procedure 和 Function，并设置相应的 IN、OUT 参数，获取其返回值等。本质上 CallableStatement 是 PrepareStatement 的一个子类，也可以当做 PrepareStatement 来使用。

在执行 Connection 中的方法 PrepareCall 的时候，会解析相应的 SQL 语句，如果是调用 Procedure 或者 Function 的 SQL 语句，会使用相应的解析工具解析出 Procedure/Function 的名字，然后通过名字在 all_argument 中查询出相应的 IN、OUT 类型，通过这些信息拼装出相应的 Procedure/Function 的描述信息。

CallableStatement 中最常用的方法如下：

- registerOutParameter
- getXXX

ResultSet

通常通过执行查询数据库的语句生成一个表示数据库结果集的数据表。

ResultSet 对象维护一个指向其当前数据行的游标。游标最初位于第一行之前，使用 next 方法可以将光标移动到下一行，当 ResultSet 对象中没有更多行时会返回 False，故可以在 While 循环中使用它来遍历结果集。

默认的 ResultSet 对象是不可更新的，并且只有一个向前移动的游标。因此，只能从第一行到最后一行遍历一次结果集，但是可以生成可滚动或可更新的 ResultSet 对象。

ResultSet 接口提供了从当前行检索列值的Getter方法（GetBoolean、GetLong等）。可以使用列的索引号或列名来检索值。一般来说，使用列索引会更有效率。列从 1 开始编号。为了获得最大的可移植性，应按从左到右的顺序读取每行中的结果集列，并且每列应仅读取一次。

对于 Getter 方法，JDBC 驱动程序尝试将底层数据转换为 Getter 方法中指定的 JAVA 类型并返回合适的 JAVA 值。

Getter 方法用作输入的列名不区分大小写。当使用列名调用 Getter 方法并且多个列具有相同名称时，将返回第一个匹配的列值。列名选项在生成结果集的 SQL 查询中使用。对于查询中未明确命名的列，最好使用列号。如果使用列名，程序员可以通过 SQL 的 AS 子句来确保它们被唯一的列引用。

相关方法如下：

- next absolute previous
- getMetaData

ResultSetMetaData 功能

ResultSetMetaData 可用于获取有关 ResultSet 对象中列的类型和属性的信息，包括很多接口可以获得非常多的描述信息。

Oracle 兼容

Oracle 兼容性主要包括两方面：数据类型兼容和功能兼容，前面的标准实现也匹配了 Oracle 兼容性。比如为了兼容 Oracle 的某些数据类型 ResultSetMetaData 的方法返回值就需要做出调成。实现 CheckSum 需要在原有的MySQL Protocol 的基础上实现 CheckSum 功能，利用 HandShake 包的 Status Flag 做 Oracle MODE Flag 等。

数据类型兼容

OceanBase 兼容的 Oracle 数据类型如下：

- 字符类型
  - VARCHAR2
  - NVARCHAR2
  - CHAR
  - NCHAR
- 数字类型
  - NUMBER
  - NUMBER_FLOAT
  - BINARY_FLOAT
  - BINARY_DOUBLE
- 日期类型
  - DATE
  - TIMESTAMP
  - TIMESTAMP WITH TIME ZONE
  - TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL
  - INTERVAL YEAR TO MONTH
  - INTERVAL DAY TO SECOND
- LOB
  - CLOB
  - BLOB
- 复杂类型
  - Struct
  - Array
  - RefCursor
- RAW
- ROWID

功能兼容

PS Checksum

Oracle 模式下支持开启 PS 后的 CheckSum 校验，UseServerPsStmtChecksum 这个参数负责开启或者关闭这个功能。

在 Preapre 的时候通过 OBCrc32C (CRC32 checksum 算法的纯 Java 实现，它使用 CRC32-C 多项式，iSCSI 使用的多项式与在许多支持 SSE4.2 的英特尔芯片组上实现的多项式相同)计算一个 CheckSum，并在只收的 Execute 中都将其发送过去，如果 CheckSum 出错则 OBServer 会报错。

更多信息

有关 OceanBase Connector/J 的详细介绍和使用方法，请参见《 OceanBase Connector/J 》。

5.3.4. OceanBase Connector/C

OceanBase Connector/C 是一个基于 C/C++ 的 OceanBase 客户端开发组件，支持 C API Lib 库。

OceanBase Connector/C 允许 C/C++ 程序以一种较为底层的方式访问 OceanBase 分布式数据库集群，以进行数据库连接、数据访问、错误处理和 Prepared Statement 处理等操作。

OceanBase Connector/C 也称为 LibobClient，用于应用程序作为独立的服务器进程通过网络连接与数据库服务器 OBServer 进行通信。客户端程序在编译时会引用 C API 头文件，同时可以连接到 C API 库文件。

接口信息

支持的 C API 函数如下表所示：

有关以下函数的详细介绍，参见 OceanBase Connector/C 的 C API 函数文档。

函数名称	功能
my_init	初始化 OceanBase 所需的全局变量。
mysql_affected_rows	返回上一次由 UPDATE、DELETE 或 INSERT 语句进行更改、删除或插入的行数。
mysql_autocommit	用于设置自动提交模式。
mysql_change_user	更改指定连接的用户和数据库。
mysql_character_set_name	返回当前连接的默认字符集名称。
mysql_client_find_plugin	返回指向已加载插件的指针。
mysql_client_register_plugin	用于注册一个插件，将插件添加到所加载的插件列表中。
mysql_close	用于关闭先前打开的连接。
mysql_commit	用于提交当前事务。

函数名称	功能
mysql_connect	用于连接服务器。
mysql_create_db	用于创建由 db 参数命名的数据库。
mysql_data_seek	用于查找结果集中的任意行。
mysql_debug	使用指定的字符串执行 DBUG_PUSH。
mysql_drop_db	用于删除由 db 参数命名的数据库。
mysql_dump_debug_info	用于引发服务器将调试信息写入错误日志。
mysql_eof	用于确定是否已读取结果集的最后一行。
mysql_errno	返回最近调用的 API 函数执行成功或失败的错误代码。
mysql_error	返回一个以空字符结尾的字符串，其中包含最近调用失败的 API 函数的错误消息。
mysql_escape_string	用于转义字符串中的特殊字符，使其可以在 SQL 语句中使用。
mysql_fetch_field	返回下一个表字段的类型。
mysql_fetch_field_direct	返回指定字段编号的字段类型。
mysql_fetch_fields	用于以数组的形式返回结果集的所有 MYSQL_FIELD 结构。MYSQL_FIELD 结构为结果集的列提供字段定义。
mysql_fetch_lengths	返回结果集中当前行的列的长度。
mysql_fetch_row	用于检索结果集的下一行。
mysql_field_count	返回最近查询的列数。

函数名称	功能
mysql_field_seek	用于查找结果集行中的列。
mysql_field_tell	返回最后一次调用 mysql_fetch_field() 的字段游标的位置。
mysql_free_result	用于释放结果集内存。
mysql_get_character_set_info	用于提供有关默认客户端字符集的信息。
mysql_get_client_info	返回表示 OceanBase 客户端库版本的字符串。
mysql_get_client_version	返回一个表示 OceanBase 客户端库版本的整数。
mysql_get_host_info	返回描述当前连接类型的字符串，包括服务器主机名。
mysql_get_proto_info	返回当前连接使用的协议版本。
mysql_get_server_info	返回表示 OceanBase 服务器版本的字符串。
mysql_get_server_version	返回一个表示 OceanBase 服务器版本的整数。
mysql_get_ssl_cipher	返回指定服务器连接的加密密码名称。
mysql_hex_string	用于创建合法的 SQL 字符串。
mysql_info	返回最近执行语句的有关信息的字符串。
mysql_init	用于分配或初始化一个适用于 mysql_real_connect() 的 MYSQL 对象。
mysql_insert_id	返回前一个 INSERT 或 UPDATE 语句为 AUTO_INCREMENT 列生成的值。
mysql_kill	用于要求服务器终止 pid 指定的线程。
mysql_library_end	用于结束使用 C API 库。

函数名称	功能
mysql_library_init	用于初始化 C API 库。
mysql_list_dbs	返回由 wild 参数指定的，与简单正则表达式相匹配的数据库名称组成的结果集。
mysql_list_fields	返回由 wild 参数指定的，与简单正则表达式相匹配的字段名称。
mysql_list_processes	返回描述当前服务器线程的结果集。
mysql_list_tables	返回由 wild 参数指定的，与简单正则表达式相匹配的表名组成的结果集。 添加H3+
mysql_load_plugin	用于加载由名称和类型指定的 OceanBase 客户端插件。
mysql_load_plugin_v	用于加载 OceanBase 客户端插件。
mysql_more_results	用于检查是否存在更多的结果。
mysql_next_result	用于执行由多个语句组成的单个语句字符串，或者用于在存储过程中使用 CALL 语句返回多个结果集的场景。
mysql_num_fields	用于返回结果集中的列数。
mysql_num_rows	用于返回结果集中的行数。
mysql_options	用于设置额外的连接选项并影响连接的行为。
mysql_options4	用于设置额外的连接选项并影响连接的行为。
mysql_ping	用于检查与服务器的连接是否正常。
mysql_plugin_options	用于将选项类型和值传递给插件。
mysql_query	用于执行由空终止字符串 stmt_str 指向的 SQL 语句。

函数名称	功能
mysql_real_connect	用于与主机上运行的 OBServer 建立连接。
mysql_real_escape_string	对语句的字符串中的特殊字符进行编码，创建 SQL 语句的合法字符串。
mysql_real_query	用于执行由 stmt_str 指向的 SQL 语句。
mysql_refresh	用于刷新或重置表和缓存。
mysql_reload	用于要求服务器重新加载授权表。
mysql_rollback	用于回滚当前事务。
mysql_row_seek	用于在查询结果集中查找任意行。
mysql_row_tell	返回结果集行游标的当前位置。
mysql_select_db	用于使 db 指定的数据库成为 mysql 指定连接上的默认（当前）数据库。
mysql_server_end	用于结束使用 C API 库。
mysql_server_init	用于初始化 OceanBase 客户端库。
mysql_set_character_set	用于设置当前连接的默认字符集。
mysql_set_local_infile_default	将 LOAD DATA LOCAL 回调函数设置为 C 客户端库内部使用的默认值。
mysql_set_local_infile_handler	用于安装指定应用程序的 LOAD DATA LOCAL 句柄回调。
mysql_set_server_option	用于启用或禁用连接选项。
mysql_shutdown	用于关闭数据库服务器。
mysql_sqlstate	返回一个以空字符结尾的字符串，其中包含最近执行的 SQL 语句的 SQLSTATE 错误代码。

函数名称	功能
mysql_ssl_set	用于使用 SSL 建立加密连接。
mysql_stat	返回包含服务器信息的字符串（类似于 mysqladmin status 命令），包括以秒为单位的正常运行时间以及正在运行的线程、问题、重载和打开表的数量。
mysql_stmt_affected_rows	返回使用上次准备的 UPDATE、DELETE 或 INSERT 语句进行更改、删除或插入的行数。
mysql_stmt_attr_get	用于获取语句的属性值。
mysql_stmt_attr_set	用于设置 Prepared Statement 属性值。
mysql_stmt_bind_param	将应用程序数据缓冲器与 Prepared Statement 中的参数标记相关联。
mysql_stmt_bind_result	用于将结果集中的输出列关联（即绑定）到数据缓冲区和长度缓冲区。
mysql_stmt_close	用于关闭 Prepared Statement。
mysql_stmt_data_seek	用于查找语句结果集中的任意行。
mysql_stmt_errno	返回最近调用语句的 API 函数的错误代码。
mysql_stmt_error	返回最近调用语句的 API 函数的错误信息。
mysql_stmt_execute	执行与语句句柄关联的预处理查询。
mysql_stmt_fetch	获取结果集的下一行并返回所有绑定列的数据。
mysql_stmt_fetch_column	用于获取当前结果集行的一列。
mysql_stmt_field_count	返回最新 Prepared Statement 的列数。
mysql_stmt_free_result	用于释放执行 Prepared Statement 产生的结果集相关的内存。

函数名称	功能
mysql_stmt_init	用于创建并返回一个 MYSQL_STMT 句柄。
mysql_stmt_insert_id	返回由预处理的 INSERT 或 UPDATE 语句为 AUTO_INCREMENT 列生成的值。
mysql_stmt_next_result	在多结果 Prepared Statement 执行中返回或启动下一个结果。
mysql_stmt_num_rows	返回结果集中的行数。
mysql_stmt_param_count	返回 Prepared Statement 中存在的参数数量。
mysql_stmt_param_metadata	将参数元数据作为结果集返回。
mysql_stmt_prepare	预处理由字符串 stmt_str 指向的 SQL 语句并返回一个状态值。
mysql_stmt_reset	用于重置服务器上的语句、使用 mysql_stmt_send_long_data() 发送的数据、未缓冲的结果集和当前错误。
mysql_stmt_result_metadata	用于获取 Prepared Statement 的结果集元数据。
mysql_stmt_row_seek	用于在 Prepared Statement 结果集中寻找任意行。
mysql_stmt_row_tell	返回 Prepared Statement 结果集行游标的当前位置。
mysql_stmt_send_long_data	使应用程序能够将参数数据分段（或“块”）发送到服务器。
mysql_stmt_sqlstate	最近调用的 Prepared Statement API 函数的 SQLSTATE 值。
mysql_stmt_store_result	用于检索并存储整个结果集。
mysql_store_result	检索并存储整个结果集。
mysql_thread_end	用于结束使用线程句柄。
mysql_thread_id	返回当前连接的线程 ID。

函数名称	功能
mysql_thread_init	用于初始化线程句柄。
mysql_thread_safe	用于指示客户端库的编译是否为线程安全的。
mysql_use_result	用于逐行检索结果集。
mysql_warning_count	返回在执行前一个 SQL 语句期间生成的错误、告警和注释的数量。

C API 对象列表

对象	说明
MYSQL	处理对象
MYSQL_RES	结果对象
MYSQL_ROW	结果行对象
MYSQL_FIELD	结果列信息对象
MYSQL_FIELD_OFFSET	结果列的偏移对象
my_ulonglong	基础类型
my_bool	基础类型

更多信息

有关 OceanBase Connector/C 的详细介绍和使用方法，请参见《 OceanBase Connector/C 》。

6.用户接口和查询语言

6.1. SQL

6.1.1. SQL 介绍

6.1.1.1. SQL 简介

结构化查询语言（Structured Query Language）简称 SQL，是一种用于特殊目的的编程语言，是一种数据库查询和程序设计语言，用于存取数据以及查询、更新和管理关系数据库系统。程序和用户都通过 SQL 语句存取数据。

SQL 提供了访问关系数据库 (例如 OceanBase 数据库) 的接口。

SQL 作为一种通用的统一语言，可以完成以下任务:

- 创建、替换、更改和删除对象
- 插入、更新和删除表行
- 查询数据
- 控制对数据库及其对象的访问
- 保证数据库的一致性和完整性

SQL 可以通过交互式（用户直接将 SQL 语句发送到 OceanBase 数据库）使用，也可以嵌入到不同语言（例如 C 或者 Java）中使用。

6.1.1.2. SQL 的访问

计算机语言有两个广泛的家族，即说明型语言和程序型语言。

说明型语言是非过程的，仅描述做什么。程序型语言描述怎么做事情，例如 C + + 和 Java。

从这个分类角度来看，SQL 属于说明型语言。用户指定了想要的结果，而不是如何产生数据。例如，使用以下语句查询张姓的员工的记录。数据库负责执行获取数据的过程，同时将检索后的数据返回给用户。SQL 的说明特性使您能够在逻辑层面处理数据。只有在操作数据时，才需要关注实现细节。

```
SELECT last_name, first_name
   WHERE last_name LIKE '张 %' ORDER BY last_name, first_name;
```

数据库取得满足条件的所有行，其中的条件也称为谓词，由 `WHERE` 子句指定。 数据库可以将这些行作为一个单元传递给用户、其他 SQL 语句或应用程序。应用程序不需要逐个处理每一行，开发人员不需要知道这些数据行在物理上如何存储的，也不需要知道这些数据行如何被检索出来的。

所有 SQL 语句都使用优化器，优化器是数据库的一个组件，它确定了访问所请求数据的最有效方式。OceanBase 数据库还提供了如何利用优化器来更好完成用户各种需求的技术。

6.1.1.3. SQL 的标准

OceanBase 数据库遵循业界公认的标准。

目前，行业认可的委员会是美国国家标准协会 (ANSI) 和国际标准化组织 (ISO)。 ANSI 和 ISO/IEC 都接受 SQL 作为关系数据库的标准语言。

SQL 标准由十个部分组成。其中一部分 (SQL/RPR:2012) 最新的版本在 2012 年修订。五个部分在 2011 年做了修订。其他四个部分，沿用 2008 修订的版本。

OceanBase 数据库的 SQL 对 ANSI/ISO 标准 SQL 语言进行了许多扩展。工具 OBClient、ODC 使您可以运行所有基于 OceanBase 数据库的 ANSI/ISO 标准 SQL 语句以及任何扩展的语句或函数。

6.1.2. SQL 语句

6.1.2.1. SQL 语句概述

对 OceanBase 数据库的运维任务和数据处理等所有操作都使用 SQL 语句。

SQL 语句是一种计算机语言或者指令，由标识符、参数、变量、数据类型和 SQL 保留字组成。

SQL 语句必须是完整的 SQL 子句，例如：

```
SELECT last_name, department_id FROM employee;
```

OceanBase 数据库仅运行完整的 SQL 语句。如下例所示的片段会生成一个错误，并指示需要更多文本：

```
SELECT last_name;
```

OceanBase 数据库的 SQL 语句分为以下三类：

- 数据定义语言（DDL）语句
- 数据操作语言（DML）语句
- 数据控制语句（DCL）语句

6.1.2.2. DDL

数据定义语言 DDL 语句用于定义、更改和删除 Schema 对象。

DDL 使您能够在不改变访问对象的应用程序的情况下更改对象的属性。例如，您可以向人力资源应用程序访问的表中添加列，而无需重写该应用程序。 当数据库用户在数据库中执行工作时，也可以使用 DDL 更改对象的结构。

更具体地说，DDL 语句使您能够完成如下操作：

- 创建、更改和删除 Schema 对象和其他数据库对象结构，包括数据库本身和数据库用户。大多数 DDL 语句以关键字 `CREATE` 、 `ALTER` 或 `DROP` 开头。
- 删除 Schema 对象中的所有数据，而不删除这些对象的结构 (非 `TRUNCATE` 命令)。
- 打开和关闭审核选项 (关键字 `AUDIT` 和 `NOAUDIT`)。
- 为数据库对象添加注释。

数据库执行 DDL 语句之前，会进行一个隐式提交。执行完 DDL 语句后，会立即执行一个提交或回滚动作。

以下示例使用 DDL 语句创建 `cusotmer` 表，使用 DML 语句在表中插入两行。然后，使用 DDL 语句更改表结构，使用 DCL 语句向用户授予和回收此表的读取权限，最后删除该表。

```
CREATE customer(cust_id INT PRIMARY KEY,common_name VARCHAR2(15));

# DML 语句
INSERT INTO customer VALUES (1,'Tom');
INSERT INTO customer VALUES (2,'Mary');

# DDL 语句
ALTER TABLE customer ADD ( cust_name VARCHAR2(40) );

# DCL 语句
GRANT SELECT ON customer to User2;
REVOKE SELECT ON customer from Users;

# DDL 语句
DROP TABLE customer;
```

上述示例中，两个插入语句后跟随一个 `ALTER TABLE` 语句，因此数据库提交两个插入操作。如果 `ALTER TABLE` 语句执行成功，则数据库提交此 DDL 语句；否则，数据库将回滚此 DDL 语句。无论是哪种情况，这两种情况的插入操作已经提交。

6.1.2.3. DML

数据操作语言 DML 语句用于查询或操作现有 Schema 对象中的数据。

DDL 语句改变数据库结构，而DML 语句可以查询或改变内容。例如， `ALTER TABLE` 更改表的结构，而 `INSERT` 语句向表中添加一行或多行。

DML 语句是最常用的 SQL 语句，使您能够完成如下操作：

- 从一个或多个表或视图查询（ `SELECT` 语句）。
- 通过值插入，或者查询插入的方式，将新的数据行添加到表或视图中（ `INSERT` 语句)
- 更改表或视图现有行中的列值（ `UPDATE` 语句)。
- 有条件地更新或插入行到表或视图（ `MERGE` 语句）。
- 从表或视图中移除行（ `DELETE` 语句)。
- 查看 SQL 语句的执行计划（ `EXPLAIN` 语句)。

以下示例使用 DML 查询 `customer` 表，使用 DML 将行插入到 `customer` ，更新此行，然后将其删除。

```
SELECT * FROM customer;
INSERT INTO customer VALUES(1234, 'Tom', 'Jacy');
UPDATE customer SET cust_name = 'Tomy' WHERE cust_id = 11;
DELETE customer WHERE cust_id = 11;
```

构成逻辑工作单元的一组 DML 语句的集合称为事务。与 DDL 语句不同，DML 语句不会隐式提交当前事务。

例如，转账交易可能涉及三种离散操作，包括减少储蓄账户余额，增加支票账户余额，并将转账记录在账户历史表中， 这种转账交易就可以称为事务。

6.1.2.4. DCL

数据控制语言 DCL 语句是对数据访问权限控制的命令，可以控制指定账号对指定数据库资源的访问权限。

一些用户默认具有访问一些数据的权限。例如，在 Oracle 模式下，数据具有 Owner 的概念，某个 Schema 下的数据的 Owner 是同名用户。数据 Owner 可以不需要任何授权就可以访问数据，非 Owner 需要授权。而 MySQL 模式下，数据没有 Owner 概念，访问数据时需要授权。

示例 1：将 Schema A 下的表 `ta` 的查询权限授予用户 B。

```
GRANT SELECT ON a.ta TO b;
```

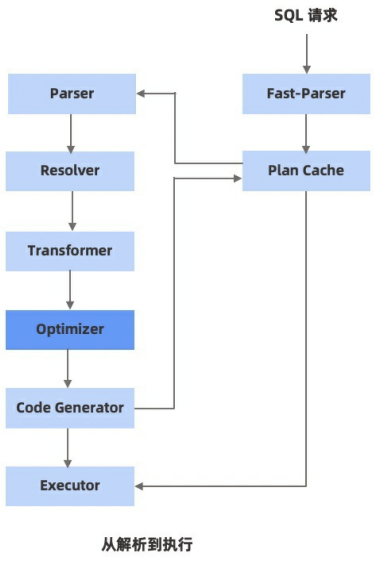
示例 2：回收用户 B 在 Schema A 下的表 `ta` 的查询权限。

```
REVOKE SELECT ON a.ta FROM b;
```

6.1.3. SQL 请求执行流程

SQL 引擎从接受 SQL 请求到执行遵循典型的流程。

流程如下图所示：



> **说明**
>
> 此执行流程适用于 DML 和 `SELECT` 语句，其他语句（例如 DCL ）没有优化器等步骤。

下表为 SQL 请求执行流程的步骤说明。

步骤	说明
Fast-parser	仅使用词法分析对文本串直接参数化，获取参数化后的文本及常量参数。
Parser（词法/语法解析模块）	在收到用户发送的 SQL 请求串后，Parser 会将字符串分成一个个的“单词”，并根据预先设定好的语法规则解析整个请求，将 SQL 请求字符串转换成带有语法结构信息的内存数据结构，称为语法树（Syntax Tree）。
Plan Cache（执行计划缓存模块）	执行计划缓存模块会将该 SQL 第一次生成的执行计划缓存在内存中，后续的执行可以反复执行这个计划，避免了重复查询优化的过程。
Resolver（语义解析模块）	Resolver 将生成的语法树转换为带有数据库语义信息的内部数据结构。在这一过程中，Resolver 将根据数据库元信息将 SQL 请求中的 Token 翻译成对应的对象（例如库、表、列、索引等），生成的数据结构叫做 Statement Tree。
Transfomer（逻辑改写模块）	分析用户 SQL 的语义，并根据内部的规则或代价模型，将用户 SQL 改写为与之等价的其他形式，并将其提供给后续的优化器做进一步的优化。Transformer 的工作方式是在原 Statement Tree 上做等价变换，变换的结果仍然是一棵 Statement Tree。
Optimizer（优化器）	优化器是整个 SQL 请求优化的核心，其作用是为 SQL 请求生成最佳的执行计划。在优化过程中，优化器需要综合考虑 SQL 请求的语义、对象数据特征、对象物理分布等多方面因素，解决访问路径选择、联接顺序选择、联接算法选择、分布式计划生成等多个核心问题，最终选择一个对应该 SQL 的最佳执行计划。
Code Generator（代码生成器）	将执行计划转换为可执行的代码，但是不做任何优化选择。
Executor（执行器）	启动 SQL 的执行过程。 • 对于本地执行计划，Executor 会简单的从执行计划的顶端的算子开始调用，根据算子自身的逻辑完成整个执行的过程，并返回执行结果。 • 对于远程或分布式计划，将执行树分成多个可以调度的子计划，并通过 RPC 将其发送给相关的节点去执行。

6.1.4. SQL 执行计划

执行计划（Execution Plan）是对一条 SQL 查询语句在数据库中执行过程的描述。

用户可以通过 `EXPLAIN` 命令查看优化器针对指定 SQL 生成的逻辑执行计划。如果要分析某条 SQL 的性能问题，通常需要先查看 SQL 的执行计划，排查每一步 SQL 执行是否存在问题。所以读懂执行计划是 SQL 优化的先决条件，而了解执行计划的算子是理解 `EXPLAIN` 命令的关键。

EXPLAIN 命令格式

OceanBase 数据库的执行计划命令有三种模式：`EXPLAIN BASIC`、`EXPLAIN` 和 `EXPLAIN EXTENDED`。这三种模式对执行计划展现不同粒度的细节信息：

- `EXPLAIN BASIC` 命令用于最基本的计划展示。
- `EXPLAIN EXTENDED` 命令用于最详细的计划展示（通常在排查问题时使用这种展示模式）。
- `EXPLAIN` 命令所展示的信息可以帮助普通用户了解整个计划的执行方式。

命令格式如下：

```
EXPLAIN [BASIC | EXTENDED | PARTITIONS | FORMAT = format_name] explainable_stmt
format_name: { TRADITIONAL | JSON }
explainable_stmt: { SELECT statement
| DELETE statement
| INSERT statement
| REPLACE statement
| UPDATE statement }
```

执行计划形状与算子信息

OceanBase 数据库执行计划展示如下：

```
|ID|OPERATOR              |NAME       |EST. ROWS|COST |
-----------------------------------------------------------
|0 |LIMIT                 |           |100      |81141|
|1 | TOP-N SORT           |           |100      |81127|
|2 |  HASH GROUP BY       |           |2924     |68551|
|3 |   HASH JOIN          |           |2924     |65004|
|4 |    SUBPLAN SCAN      |VIEW1      |2953     |19070|
|5 |     HASH GROUP BY    |           |2953     |18662|
|6 |      NESTED-LOOP JOIN|           |2953     |15080|
|7 |       TABLE SCAN     |ITEM       |19       |11841|
|8 |       TABLE SCAN     |STORE_SALES|161      |73   |
|9 |    TABLE SCAN        |DT         |6088     |29401|
===========================================================
```

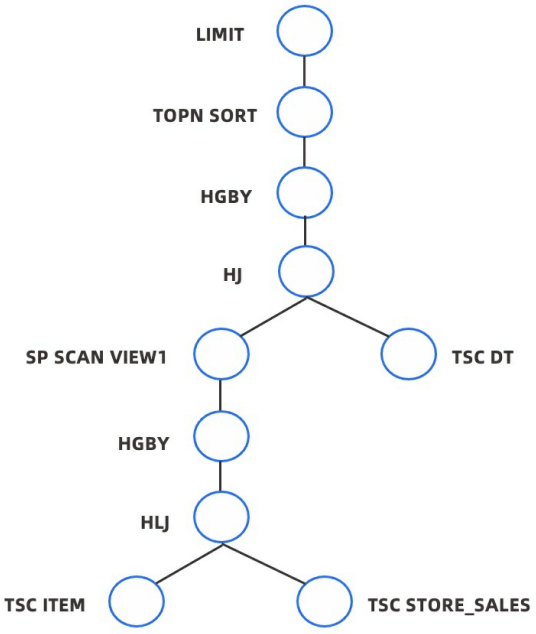
由示例可见，OceanBase 数据库的计划展示与 Oracle 数据库类似。OceanBase 数据库执行计划中的各列的含义如下表所示。

列名	含义
ID	执行树按照前序遍历的方式得到的编号（从 0 开始）。
OPERATOR	操作算子的名称。
NAME	对应表操作的表名（索引名）。
EST. ROWS	估算该操作算子的输出行数。
COST	该操作算子的执行代价（微秒）。

> **说明**
>
> 在表操作中，`NAME` 字段会显示该操作涉及的表的名称（别名），如果是使用索引访问，还会在名称后的括号中展示该索引的名称，例如 `t1(t1_c2)` 表示使用了索引 `t1_c2` 。如果扫描的顺序是逆序，还会在后面使用 `RESERVE` 关键字标识，例如 `t1(t1_c2,RESERVE)` 。

OceanBase 数据库 `EXPLAIN` 命令输出的第一部分是执行计划的树形结构展示。其中每一个操作在树中的层次通过其在算子中的缩进予以展示。树的层次关系用缩进来表示，层次最深的优先执行，层次相同的算子以指定算子的执行顺序为标准来执行。

上述示例查询的计划展示树如下图所示。



OceanBase 数据库 `EXPLAIN` 命令输出的第二部分是各操作算子的详细信息，包括输出表达式、过滤条件、分区信息以及各算子的独有信息（包括排序键、联接键、下压条件等）。示例如下：

```
Outputs & filters:
-------------------------------------
  0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), sort_keys([t1.c1, ASC], [t1.
c2, ASC]), prefix_pos(1)
  1 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil),
      equal_conds([t1.c1 = t2.c2]), other_conds(nil)
  2 - output([t2.c1], [t2.c2]), filter(nil), sort_keys([t2.c2, ASC])
  3 - output([t2.c2], [t2.c1]), filter(nil),
      access([t2.c2], [t2.c1]), partitions(p0)
  4 - output([t1.c1], [t1.c2]), filter(nil),
      access([t1.c1], [t1.c2]), partitions(p0)
```

6.1.5. 分布式执行计划

6.1.5.1. 分布式执行和并行查询

分布式执行简介

OceanBase 数据库基于 Shared-Nothing 的分布式系统构建，具有分布式执行计划生成和执行能力。

由于一个关系数据表的数据会以分区的方式存放在系统里面的各个节点上，所以对于跨分区的数据查询请求，必然会要求执行计划能够对多个节点的数据进行操作。OceanBase 数据库的优化器会自动根据查询和数据的物理分布生成分布式执行计划。对于分布式执行计划，分区可以提高查询性能。如果数据库关系表比较小，则不必要进行分区，如果关系表比较大，则需要根据上层业务需求谨慎选择分区键，以保证大多数查询能够使用分区键进行分区裁剪，从而减少数据访问量。

同时，对于有关联性的表，建议使用关联键作为分区键，并采用相同分区方式，使用表组将相同的分区配置在同样的节点上，以减少跨节点的数据交互。

并行查询简介

并行查询是指通过对查询计划的并行化执行，提升对每一个查询计划的 CPU 和 IO 处理能力，从而缩短单个查询的响应时间。并行查询技术可以用于分布式执行计划，也可以用于本地查询计划。

当单个查询的访问数据不在同一个节点上时，需要通过数据重分布的方式，相关数据执行分发到相同的节点进行计算。以每一次的数据重分布节点为上下界，OceanBase 数据库的执行计划在垂直方向上被划分为多个 DFO（Data Flow Object），而每一个 DFO 可以被切分为指定并行度的任务，通过并发执行以提高执行效率。

一般来说，当并行度提高时，查询的响应时间会缩短，更多的 CPU、IO 和内存资源会被用于执行查询命令。对于支持大数据量查询处理的 DSS（Decision Support Systems）系统或者数据仓库型应用来说，查询时间的提升尤为明显。

整体来说，并行查询的总体思路和分布式执行计划有相似之处，即将执行计划分解之后，将执行计划的每个部分由多个执行线程执行，通过一定的调度的方式，实现执行计划的 DFO 之间的并发执行和 DFO 内部的并发执行。并行查询特别适用于在线交易（OLTP）场景的批量更新操作、创建索引和维护索引等操作。

当系统满足以下条件时，并行查询可以有效提升系统处理性能：

- 充足的 IO 带宽

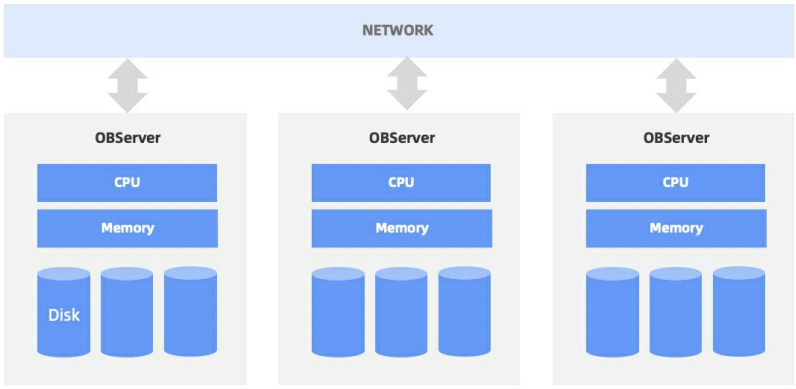
- 系统 CPU 负载较低
- 充足的内存资源

如果系统没有充足的资源进行额外的并行处理，使用并行查询或者提高并行度并不能提高执行性能。相反，在系统过载的情况下，操作系统会被迫进行更多的调度，例如执行上下文切换可能会导致性能的下降。

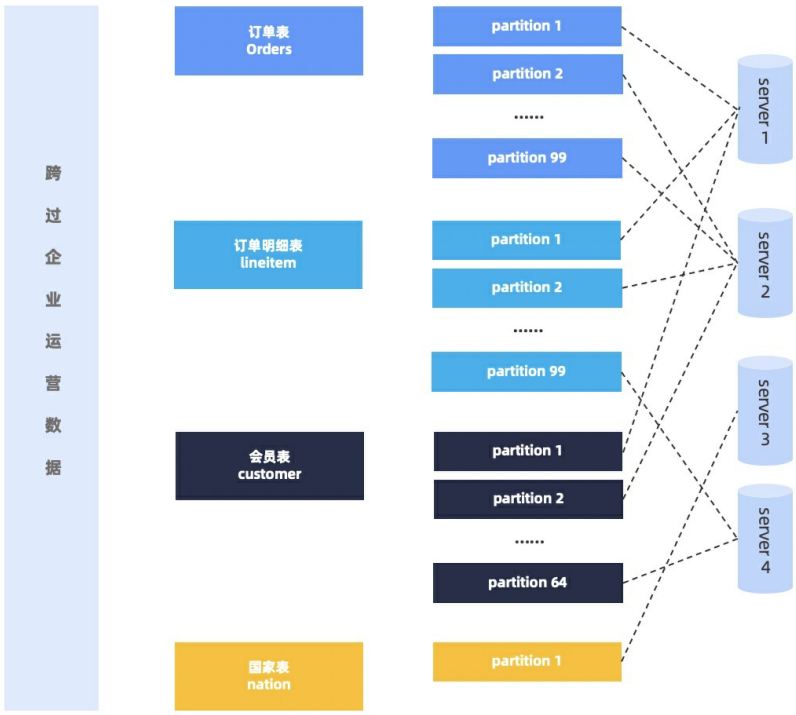
通常在 DSS 系统中，需要访问大量数据，这时并行执行能够提升执行响应时间。对于简单的 DML 操作或者涉及数据量比较小的查询来说，使用并行查询并不能很明显的降低查询响应时间。

并行查询和分布式查询原理

OceanBase 数据库的数据以分片的形式存储于每个节点，节点之间通过千兆、万兆网络通信。一般会在每个节点上部署一个叫做 observer 的进程，它是 OceanBase 数据库对外服务的主体。如下图所示。



OceanBase 数据库会根据一定的均衡策略将数据分片均衡到多个 observer 进程上，因此对于一个并行查询一般需要同时访问多个 observer 进程。如下图所示。

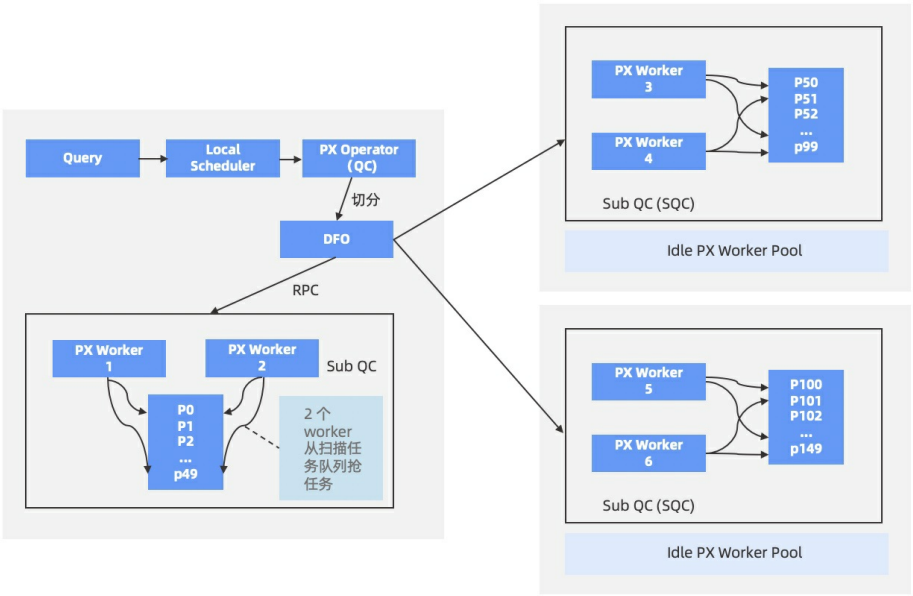


SQL 语句并行执行流程

当用户指定的 SQL 语句需要访问的数据位于 2 台或 2 台以上 OBServer 时，就会启用并行执行，用户所连接的这个 OBServer 将承担查询协调者 QC（Query Coordinator）的角色，执行步骤如下：

1. QC 预约足够的线程资源。
2. QC 将需要并行的计划拆成多个子计划，即 DFO（Data Flow Operation）。每个 DFO 包含若干个串行执行的算子。例如，一个 DFO 里包含了扫描分区、聚集和发送算子的任务，另外一个 DFO 里包含了收集、聚集算子等任务。
3. QC 按照一定的逻辑顺序将 DFO 调度到合适的 OBServer 上执行，OBServer 上会临时启动一个辅助协调者 SQC（Sub Query Coordinator），SQC 负责在所在 OBServer 上为各个 DFO 申请执行资源、构造执行上下文环境等，然后启动 DFO 在各个 OBServer 上进行并行执行。
4. 当各个 DFO 都执行完毕，QC 会串行执行剩余部分的计算。例如，一个并行的 `COUNT` 算法最终需要 QC 将各个机器上的计算结果做一个 `SUM` 运算。
5. QC 所在线程将结果返回给客户端。

优化器负责决策生成一个怎样的并行计划，QC 负责具体执行该计划。例如，两分区表 JOIN，优化器根据规则和代价信息，可能生成一个分布式的 PARTITION WISE JOIN 计划，也可能生成一个 HASH HASH 打散的分布式 JOIN 计划。计划一旦确定，QC 就会将计划拆分成多个 DFO，进行有序的调度执行。QC 的执行步骤如下图所示。



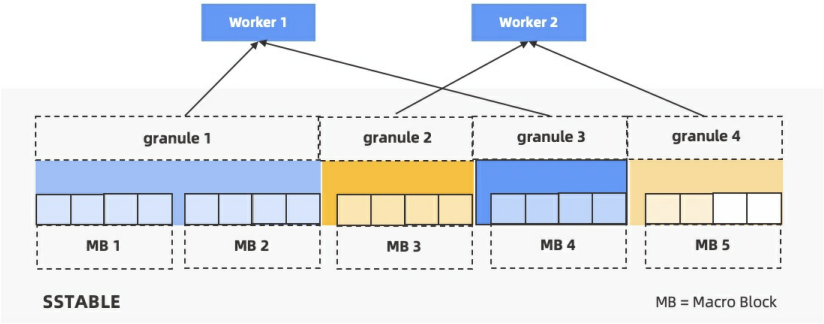
并行度与任务划分方法

并行度 DOP（Degree Of Parallelism）可以指定使用多少个线程（Worker）来执行一个 DFO。目前 OceanBase 数据库通过 `PARALLEL` Hint 来指定并行度。确定并行度后，会将 DOP 拆分到需要运行 DFO 的多个 OBServer 上。

对于包含扫描的 DFO，会计算 DFO 需要访问哪些分区，这些分区分布在哪些 OBServer 上，然后将 DOP 按比例划分给对应的 OBServer。例如，DOP 为 6，DFO 要访问 120 个分区，其中 server1 上有 60 个分区，server2 上有 40 个分区，server3 上有 20 个分区，那么会分 3 个线程给 server1，分 2 个线程给 server2，分 1 个线程给 server3，达到平均每个线程可以处理 20 个分区的效果。如果 DOP 和分区数不能整除，OceanBase 数据库会做一定的调整，以达到长尾尽可能短的目的。

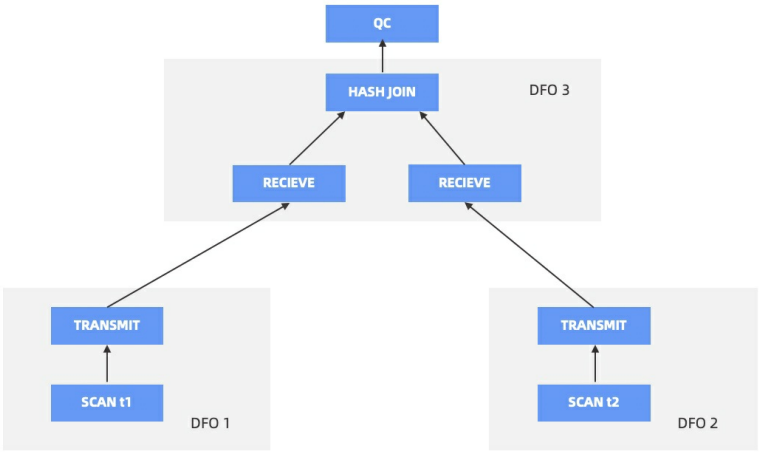
如果每个机器上分得的 Worker 数远大于分区数，会自动做分区内并行。每个分区会以宏块为边界切分成若干个扫描任务，由多个 Worker 争抢执行。

为了将这种划分能力进行抽象和封装，引入 **Granule** 的概念。每个扫描任务称为一个 Granule，这个扫描任务既可以是扫一个分区，也可以扫描分区中的一小块范围。如下图所示。



并行调度方法

优化器生成并行计划后，QC 会将其切分成多个 DFO。如下图所示，`t1` 表和 `t2` 表做 HASH JOIN，切分成了 3 个 DFO，DFO 1 和 DFO 2 负责并行扫描数据，并将数据 HASH 到对应节点，DFO 3 负责做 HASH JOIN，并将最终的 HASH 结果汇总到 QC。

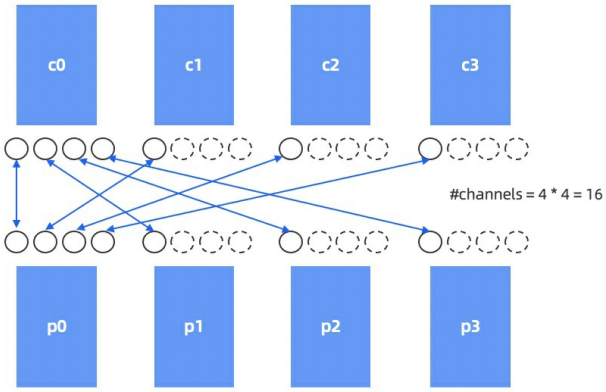


QC 会尽量使用两组线程来完成计划的调度，上述示例中的调度流程如下：

1. QC 首先会调度 DFO 1 和 DFO 3，DFO 1 开始执行后就开始扫数据，并吐给 DFO 3。
2. DFO 3 开始执行后，首先会阻塞在 HASH JOIN 创建 Hash Table 的步骤上，也就是会一直会从 DFO 1 收集数据，直到全部收集完成，建立 Hash Table 完成。然后 DFO 3 会从右边的 DFO 2 收集数据。这时候 DFO 2 还没有被调度起来，所以 DFO 3 会等待在收数据的流程上。DFO 1 在把数据都发送给 DFO 3 后就可以让出线程资源退出了。
3. 调度器回收了 DFO 1 的线程资源后，立即会调度 DFO 2。
4. DFO 2 开始运行后就开始发送数据给 DFO 3 ，DFO 3 每收到一行 DFO 2 的数据就回到 Hash Table 中查表，如果命中，就会立即向上输出给 QC，QC 负责将结果输出给客户端。

网络通信方法

对于一对有关联的 Child DFO 和 Parent DFO，Child DFO 作为生产者分配了 M 个 Worker 线程， Parent DFO 作为消费者分配了 N 个 Worker 线程。他们之间的数据传输需要用到 M * N 个网络通道。如下图所示。



为了更好的理解这种网络通信形式，引入数据传输层 DTL（Data Transfer Layer）的概念，即任意两点之间的通信连接使用通道（Channel）的概念来描述。

通道分为发送端和接收端，在最初的实现中我们允许发送端无限地给接收端发送数据，但发现如果接收端无法立即消费掉这些数据，可能会导致接收端内存爆掉，所以加入了流控逻辑。每个 Channel 接收端预留了三个槽位，当槽位被数据占满时会通知发送端暂停发送数据，当有接收端数据被消费空闲槽位出现时通知发送端继续发送。

6.1.5.2. 生成分布式计划

OceanBase 数据库的优化器会分为两大阶段来生成分布式的执行计划。

这两阶段生成执行计划的方式如下：

1. 第一阶段：不考虑数据的物理分布，生成所有基于本地关系优化的最优执行计划。在本地计划生成后，优化器会检查数据是否访问了多个分区，或者是否访问的是本地单分区表但是用户使用 Hint 强制采用了并行查询执行。
2. 第二阶段：生成分布式计划。根据执行计划树，在需要进行数据重分布的地方，插入 `EXCHANGE` 节点，从而将原先的本地计划树变成分布式执行计划。

生成分布式计划的过程就是在原始计划树上寻找恰当位置插入 `EXCHANGE` 算子的过程，在自顶向下遍历计划树的时候，需要根据相应算子的数据处理情况以及输入算子的数据分区情况，来决定是否需要插入 `EXCHANGE` 算子。

如下示例为最简单的单表扫描。当表 `t1` 是一个分区表，可以在 `TABLE SCAN` 上插入配对的 `EXCHANGE` 算子，从而将 `TABLE SCAN` 和 `EXCHANGE OUT` 封装成一个 Job，可以用于并行的执行。

```
obclient>CREATE TABLE t1 (v1 INT, v2 INT) PARTITION BY HASH(v1) PARTITIONS 5;
Query OK, 0 rows affected

obclient>EXPLAIN SELECT * FROM t1\G;
*************************** 1. row ***************************
Query Plan:
============================================
|ID|OPERATOR               |NAME    |EST. ROWS|COST  |
--------------------------------------------------------
|0 |PX COORDINATOR         |        |500000   |545109|
|1 | EXCHANGE OUT DISTR    |:EX10000|500000   |320292|
|2 |  PX PARTITION ITERATOR|        |500000   |320292|
|3 |   TABLE SCAN          |T1      |500000   |320292|
========================================================

Outputs & filters:
-------------------------------------
  0 - output([T1.V1], [T1.V2]), filter(nil)
  1 - output([T1.V1], [T1.V2]), filter(nil), dop=1
  2 - output([T1.V1], [T1.V2]), filter(nil)
  3 - output([T1.V1], [T1.V2]), filter(nil),
      access([T1.V1], [T1.V2]), partitions(p[0-4])
```

单输入可下压算子

单输入可下压算子主要包括 `AGGREGATION` 、 `SORT` 、 `GROUP BY` 和 `LIMIT` 算子等，除了 `LIMIT` 算子以外，其余所列举的算子都会有一个操作的键，如果操作的键和输入数据的数据分布是一致的，则可以做一阶段聚合操作，也即 Partition Wise Aggregation。如果操作的键和输入数据的数据分布是不一致的，则需要做两阶段聚合操作，聚合算子需要做下压操作。

一阶段聚合操作如下例所示：

```
obclient>CREATE TABLE t2 (v1 INT, v2 INT) PARTITION BY HASH(v1) PARTITIONS 4;
Query OK, 0 rows affected

obclient>EXPLAIN SELECT SUM(v1) FROM t2 GROUP BY v1\G;
*************************** 1. row ***************************
Query Plan:
| =====================================================
|ID|OPERATOR               |NAME    |EST. ROWS|COST   |
-----------------------------------------------------
|0 |PX COORDINATOR         |        |101      |357302|
|1 | EXCHANGE OUT DISTR    |:EX10000|101      |357297|
|2 |  PX PARTITION ITERATOR|        |101      |357297|
|3 |   MERGE GROUP BY      |        |101      |357297|
|4 |    TABLE SCAN         |t2      |400000   |247403|
=====================================================

Outputs & filters:
-------------------------------------
  0 - output([T_FUN_SUM(t2.v1)]), filter(nil)
  1 - output([T_FUN_SUM(t2.v1)]), filter(nil), dop=1
  2 - output([T_FUN_SUM(t2.v1)]), filter(nil)
  3 - output([T_FUN_SUM(t2.v1)]), filter(nil),
      group([t2.v1]), agg_func([T_FUN_SUM(t2.v1)])
  4 - output([t2.v1]), filter(nil),
      access([t2.v1]), partitions(p[0-3])
```

二阶段聚合操作如下例所示：

```
| ============================================================
|ID|OPERATOR                      |NAME    |EST. ROWS|COST   |
------------------------------------------------------------
|0 |PX COORDINATOR                |        |101      |561383|
|1 | EXCHANGE OUT DISTR           |:EX10001|101      |561374|
|2 |  HASH GROUP BY               |        |101      |561374|
|3 |   EXCHANGE IN DISTR          |        |101      |408805|
|4 |    EXCHANGE OUT DISTR (HASH)|:EX10000|101      |408795|
|5 |     HASH GROUP BY            |        |101      |408795|
|6 |      PX PARTITION ITERATOR   |        |400000   |256226|
|7 |       TABLE SCAN             |t2      |400000   |256226|
============================================================

Outputs & filters:
-------------------------------------
  0 - output([T_FUN_SUM(T_FUN_SUM(t2.v1))]), filter(nil)
  1 - output([T_FUN_SUM(T_FUN_SUM(t2.v1))]), filter(nil), dop=1
  2 - output([T_FUN_SUM(T_FUN_SUM(t2.v1))]), filter(nil),
      group([t2.v2]), agg_func([T_FUN_SUM(T_FUN_SUM(t2.v1))])
  3 - output([t2.v2], [T_FUN_SUM(t2.v1)]), filter(nil)
  4 - (#keys=1, [t2.v2]), output([t2.v2], [T_FUN_SUM(t2.v1)]), filter(nil), dop=1
  5 - output([t2.v2], [T_FUN_SUM(t2.v1)]), filter(nil),
      group([t2.v2]), agg_func([T_FUN_SUM(t2.v1)])
  6 - output([t2.v1], [t2.v2]), filter(nil)
  7 - output([t2.v1], [t2.v2]), filter(nil),
      access([t2.v1], [t2.v2]), partitions(p[0-3])
```

二元输入算子

二元输入算子主要考虑 `JOIN` 算子的情况。对于 `JOIN` 算子来说，主要基于规则来生成分布式执行计划和选择数据重分布方法。`JOIN` 算子主要有以下三种联接方式：

- Partition-Wise Join

  当左右表都是分区表且分区方式相同，物理分布一样，并且 `JOIN` 的联接条件为分区键时，可以使用以分区为单位的联接方法。如下例所示：

```
obclient>CREATE TABLE t3 (v1 INT, v2 INT) PARTITION BY HASH(v1) PARTITIONS 4;
Query OK, 0 rows affected

obclient>EXPLAIN SELECT * FROM t2, t3 WHERE t2.v1 = t3.v1\G;
*************************** 1. row ***************************
Query Plan:
============================================================
|ID|OPERATOR               |NAME    |EST. ROWS |COST      |
|0 |PX COORDINATOR         |        |1568160000|1227554264|
|1 | EXCHANGE OUT DISTR    |:EX10000|1568160000|930670004 |
|2 |  PX PARTITION ITERATOR|        |1568160000|930670004 |
|3 |   MERGE JOIN          |        |1568160000|930670004 |
|4 |    TABLE SCAN         |t2      |400000    |256226    |
|5 |    TABLE SCAN         |t3      |400000    |256226    |
============================================================

Outputs & filters:
-------------------------------------
  0 - output([t2.v1], [t2.v2], [t3.v1], [t3.v2]), filter(nil)
  1 - output([t2.v1], [t2.v2], [t3.v1], [t3.v2]), filter(nil), dop=1
  2 - output([t2.v1], [t2.v2], [t3.v1], [t3.v2]), filter(nil)
  3 - output([t2.v1], [t2.v2], [t3.v1], [t3.v2]), filter(nil),
      equal_conds([t2.v1 = t3.v1]), other_conds(nil)
  4 - output([t2.v1], [t2.v2]), filter(nil),
      access([t2.v1], [t2.v2]), partitions(p[0-3])
  5 - output([t3.v1], [t3.v2]), filter(nil),
      access([t3.v1], [t3.v2]), partitions(p[0-3])
```

- Partial Partition-Wise Join

  当左右表中一个表为分区表，另一个表为非分区表，或者两者皆为分区表但是联接键仅和其中一个分区表的分区键相同的情况下，会以该分区表的分区分布为基准，重新分布另一个表的数据。如下例所示：

```
obclient>CREATE TABLE t4 (v1 INT, v2 INT) PARTITION BY HASH(v1) PARTITIONS 3;
Query OK, 0 rows affected

obclient>EXPLAIN SELECT * FROM t4, t2 WHERE t2.v1 = t4.v1\G;
*************************** 1. row ***************************
Query Plan:
 ============================================================
|ID|OPERATOR                     |NAME    |EST. ROWS|COST |
------------------------------------------------------------
|0 |PX COORDINATOR               |        |11880    |17658|
|1 | EXCHANGE OUT DISTR          |:EX10001|11880    |15409|
|2 |  NESTED-LOOP JOIN           |        |11880    |15409|
|3 |   EXCHANGE IN DISTR         |        |3        |37   |
|4 |    EXCHANGE OUT DISTR (PKEY)|:EX10000|3        |37   |
|5 |     PX PARTITION ITERATOR   |        |3        |37   |
|6 |      TABLE SCAN             |t4      |3        |37   |
|7 |   PX PARTITION ITERATOR     |        |3960     |2561 |
|8 |    TABLE SCAN               |t2      |3960     |2561 |
============================================================

Outputs & filters:
-------------------------------------
  0 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil)
  1 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil), dop=1
  2 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil),
      conds(nil), nl_params_([t4.v1])
  3 - output([t4.v1], [t4.v2]), filter(nil)
  4 - (#keys=1, [t4.v1]), output([t4.v1], [t4.v2]), filter(nil), dop=1
  5 - output([t4.v1], [t4.v2]), filter(nil)
  6 - output([t4.v1], [t4.v2]), filter(nil),
      access([t4.v1], [t4.v2]), partitions(p[0-2])
  7 - output([t2.v1], [t2.v2]), filter(nil)
  8 - output([t2.v1], [t2.v2]), filter(nil),
      access([t2.v1], [t2.v2]), partitions(p[0-3])
```

- 数据重分布

  当联接键和左右表的分区键都没有关系的情况下，可以根据规则计算来选择使用 `BROADCAST` 还是 `HASH HASH` 的数据重分布方式，如下例所示：

  > **注意**
  >
  > 只有在并行度大于 1 时，以下示例中两种数据重分发方式才有可能被选中。

```
obclient>EXPLAIN SELECT /*+ PARALLEL(2)*/* FROM t4, t2 WHERE t2.v2 = t4.v2\G;
*************************** 1. row ***************************
Query Plan:
============================================================
|ID|OPERATOR                         |NAME    |EST. ROWS|COST  |
------------------------------------------------------------
|0 |PX COORDINATOR                   |        |11880    |396863|
|1 | EXCHANGE OUT DISTR              |:EX10001|11880    |394614|
```

```
|2 |  HASH JOIN                        |        |11880    |394614|
|3 |   EXCHANGE IN DISTR               |        |3        |37    |
|4 |    EXCHANGE OUT DISTR (BROADCAST)|:EX10000|3        |37    |
|5 |     PX BLOCK ITERATOR             |        |3        |37    |
|6 |      TABLE SCAN                   |t4      |3        |37    |
|7 |   PX PARTITION ITERATOR           |        |400000   |256226|
|8 |    TABLE SCAN                     |t2      |400000   |256226|
=======================================================================

Outputs & filters:
-------------------------------------
  0 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil)
  1 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil), dop=2
  2 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil),
      equal_conds([t2.v2 = t4.v2]), other_conds(nil)
  3 - output([t4.v1], [t4.v2]), filter(nil)
  4 - output([t4.v1], [t4.v2]), filter(nil), dop=2
  5 - output([t4.v1], [t4.v2]), filter(nil)
  6 - output([t4.v1], [t4.v2]), filter(nil),
      access([t4.v1], [t4.v2]), partitions(p[0-2])
  7 - output([t2.v1], [t2.v2]), filter(nil)
  8 - output([t2.v1], [t2.v2]), filter(nil),
      access([t2.v1], [t2.v2]), partitions(p[0-3])


obclient>EXPLAIN SELECT /*+ PQ_DISTRIBUTE(t2 HASH HASH) PARALLEL(2)*/* FROM t4, t2
              WHERE t2.v2 = t4.v2\G;
*************************** 1. row ***************************
Query Plan:
 ============================================================
|ID|OPERATOR                      |NAME    |EST. ROWS|COST   |
------------------------------------------------------------
|0 |PX COORDINATOR                |        |11880    |434727|
|1 | EXCHANGE OUT DISTR           |:EX10002|11880    |432478|
|2 |  HASH JOIN                   |        |11880    |432478|
|3 |   EXCHANGE IN DISTR          |        |3        |37    |
|4 |    EXCHANGE OUT DISTR (HASH)|:EX10000|3        |37    |
|5 |     PX BLOCK ITERATOR        |        |3        |37    |
|6 |      TABLE SCAN              |t4      |3        |37    |
|7 |   EXCHANGE IN DISTR          |        |400000   |294090|
|8 |    EXCHANGE OUT DISTR (HASH)|:EX10001|400000   |256226|
|9 |     PX PARTITION ITERATOR    |        |400000   |256226|
|10|      TABLE SCAN              |t2      |400000   |256226|
============================================================

Outputs & filters:
-------------------------------------
  0 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil)
  1 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil), dop=2
  2 - output([t4.v1], [t4.v2], [t2.v1], [t2.v2]), filter(nil),
      equal_conds([t2.v2 = t4.v2]), other_conds(nil)
  3 - output([t4.v1], [t4.v2]), filter(nil)
  4 - (#keys=1, [t4.v2]), output([t4.v1], [t4.v2]), filter(nil), dop=2
  5 - output([t4.v1], [t4.v2]), filter(nil)
```

```
  6 - output([t4.v1], [t4.v2]), filter(nil),
      access([t4.v1], [t4.v2]), partitions(p[0-2])
  7 - output([t2.v1], [t2.v2]), filter(nil)
  8 - (#keys=1, [t2.v2]), output([t2.v1], [t2.v2]), filter(nil), dop=2
  9 - output([t2.v1], [t2.v2]), filter(nil)
  10 - output([t2.v1], [t2.v2]), filter(nil),
      access([t2.v1], [t2.v2]), partitions(p[0-3])
```

6.1.5.3. 启用和关闭并行查询

本文主要通过具体示例介绍 OceanBase 数据库如何启用分区表并行查询、非分区表并行查询、多表并行查询和关闭并行查询，以及并行执行相关的系统视图。

启用分区表并行查询

针对分区表的查询，如果查询的目标分区数大于 1，系统会自动启用并行查询，并行度 DOP 值由系统默认指定为 1。

如下例所示，创建一个分区表 `ptable`，对 `ptable` 进行全表数据的扫描操作，通过 `EXPLAIN` 命令查看生成的执行计划。通过执行计划可以看出，分区表默认的并行查询的 `dop` 值为 1。如果 OceanBase 集群一共有 3 个 OBServer，表 `ptable` 的 16 个分区分散在 3 个 OBServer 中，那么每一个 OBServer 都会启动一个工作线程（Worker Thread）来执行分区数据的扫描工作，一共需要启动 3 个工作线程来执行表的扫描工作。

```
obclient>CREATE TABLE PTABLE(c1 INT , c2 INT) PARTITION BY HASH(c1) PARTITIONS 16;
Query OK, 0 rows affected

obclient>EXPLAIN SELECT * FROM ptable\G;
Query Plan: ====================================================
|ID|OPERATOR               |NAME    |EST. ROWS|COST   |
------------------------------------------------------
|0 |PX COORDINATOR         |        |1600000  |737992|
|1 | EXCHANGE OUT DISTR    |:EX10000|1600000  |618888|
|2 |  PX PARTITION ITERATOR|        |1600000  |618888|
|3 |   TABLE SCAN          |ptable  |1600000  |618888|
======================================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil), dop=1
  2 - output([ptable.c1], [ptable.c2]), filter(nil)
  3 - output([ptable.c1], [ptable.c2]), filter(nil),
      access([ptable.c1], [ptable.c2]), partitions(p[0-15])
```

针对分区表，通过添加 `PARALLEL` Hint 启动并行查询，并指定 `dop` 值，通过 `EXPLAIN` 命令查看生成的执行计划。

```
obclient>EXPLAIN SELECT /*+ PARALLEL(8) */ * FROM ptable\G;
Query Plan: ==================================================
|ID|OPERATOR           |NAME    |EST. ROWS|COST   |
--------------------------------------------------
|0 |PX COORDINATOR     |        |1600000  |737992|
|1 | EXCHANGE OUT DISTR|:EX10000|1600000  |618888|
|2 |  PX BLOCK ITERATOR|        |1600000  |618888|
|3 |   TABLE SCAN      |ptable  |1600000  |618888|
==================================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil), dop=8
  2 - output([ptable.c1], [ptable.c2]), filter(nil)
  3 - output([ptable.c1], [ptable.c2]), filter(nil),
      access([ptable.c1], [ptable.c2]), partitions(p[0-15])
```

通过执行计划可以看出，并行查询的 `dop` 值为 8。如果查询分区所在的 OBServer 的个数小于等于 `dop` 值，那么工作线程（总个数等于 `dop` 值）会按照一定的策略分配到涉及的 OBServer 上；如果查询分区所在的 OBServer 的个数大于 `dop` 值，那么每一个 OBServer 都会至少启动一个工作线程，一共需要启动的工作线程的数目会大于 `dop` 值。

例如，当 `dop` 值为 8 时，如果 16 个分区均匀的分布在 4 台 OBServer 节点上，那么每一个 OBServer 上都会启动 2 个工作线程来扫描其对应的分区（一共启动 8 个工作线程）；如果 16 个分区分布在 16 台 OBServer 节点上（每一个节点一个分区），那么每一台 OBServer 上都会启动 1 个工作线程来扫描其对应的分区（一共启动 16 个工作线程）。

如果针对分区表的查询，查询分区数目小于等于 1，系统不会启动并行查询。如下例所示，对 `ptable` 的查询添加一个过滤条件 `c1=1` 。

```
obclient>EXPLAIN SELECT * FROM ptable WHERE c1 = 1\G;
*************************** 1. row ***************************
Query Plan:
======================================
|ID|OPERATOR  |NAME  |EST. ROWS|COST |
--------------------------------------
|0 |TABLE SCAN|ptable|990      |85222|
======================================

Outputs & filters:
-------------------------------------
  0 - output([ptable.c1], [ptable.c2]), filter([ptable.c1 = 1]),
      access([ptable.c1], [ptable.c2]), partitions(p1)
```

通过执行计划可以看出，查询的目标分区个数为 1，系统没有启动并行查询。如果希望针对一个分区的查询也能够进行并行执行，就只能通过添加 `PARALLEL` Hint 的方式进行分区内并行查询，通过 `EXPLAIN` 命令查看生成的执行计划。

```
obclient>EXPLAIN SELECT /*+ PARALLEL(8) */ * FROM ptable WHERE c1 = 1\G;
Query Plan: ==============================================
|ID|OPERATOR           |NAME    |EST. ROWS|COST|
----------------------------------------------
|0 |PX COORDINATOR     |        |990      |457 |
|1 | EXCHANGE OUT DISTR|:EX10000|990      |383 |
|2 |  PX BLOCK ITERATOR|        |990      |383 |
|3 |   TABLE SCAN      |ptable  |990      |383 |
==============================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil), dop=8
  2 - output([ptable.c1], [ptable.c2]), filter(nil)
  3 - output([ptable.c1], [ptable.c2]), filter(nil),
      access([ptable.c1], [ptable.c2]), partitions(p1)
```

> **说明**
>
> 如果希望在查询分区数等于 1 的情况下，能够采用 Hint 的方式进行分区内并行查询，需要对应的 DOP 值大于等于 2。 如果 DOP 值为空或者小于 2 将不启动并行查询。

启用非分区表并行查询

非分区表本质上是只有 1 个分区的分区表，因此针对非分区表的查询，只能通过添加 `PARALLEL` Hint 的方式启动分区内并行查询，否则不会启动并行查询。

如下例所示，创建一个非分区表 `stable` ，对 `stable` 进行全表数据的扫描操作，通过 `EXPLAIN` 命令查看生成的执行计划。

```
obclient>CREATE TABLE stable(c1 INT, c2 INT);
Query OK, 0 rows affected

obclient>EXPLAIN SELECT * FROM stable\G;
*************************** 1. row ***************************
Query Plan:
======================================
|ID|OPERATOR  |NAME  |EST. ROWS|COST |
--------------------------------------
|0 |TABLE SCAN|stable|100000   |68478|
======================================

Outputs & filters:
-------------------------------------
  0 - output([stable.c1], [stable.c2]), filter(nil),
      access([stable.c1], [stable.c2]), partitions(p0)
```

通过执行计划可以看出，非分区表不使用 Hint 的情况下，不会启动并行查询。

针对非分区表，添加 `PARALLEL` Hint 启动分区内并行查询，并指定 `dop` 值（大于等于 2），通过 `EXPLAIN` 命令查看生成的执行计划。

```
obclient>EXPLAIN SELECT /*+ PARALLEL(4)*/  * FROM stable\G;
Query Plan: ================================================
|ID|OPERATOR           |NAME    |EST. ROWS|COST |
-----------------------------------------------
|0 |PX COORDINATOR     |        |100000   |46125|
|1 | EXCHANGE OUT DISTR|:EX10000|100000   |38681|
|2 |  PX BLOCK ITERATOR|        |100000   |38681|
|3 |   TABLE SCAN      |stable  |100000   |38681|
===============================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(stable.c1, stable.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(stable.c1, stable.c2)]), filter(nil), dop=4
  2 - output([stable.c1], [stable.c2]), filter(nil)
  3 - output([stable.c1], [stable.c2]), filter(nil),
      access([stable.c1], [stable.c2]), partitions(p0)
```

启用多表并行查询

在查询中，多表 `JOIN` 查询最为常见。对于多表的场景，如果查询的分区数都大于 1，每张表都会采用并行查询。

如下例所示，首先创建两张分区表 `p1table` 和 `p2table` 。

```
obclient>CREATE TABLE p1table(c1 INT ,c2 INT) PARTITION BY HASH(c1) PARTITIONS 2;
Query OK, 0 rows affected

obclient>CREATE TABLE p2table(c1 INT ,c2 INT) PARTITION BY HASH(c1) PARTITIONS 4;
Query OK, 0 rows affected
```

查询 `p1table` 与 `p2table` 的 `JOIN` 结果，`JOIN` 条件是 `p1table.c1=p2table.c2`，所得执行计划如下所示。

```
************************* 1. row *************************
Query Plan: ============================================================
|ID|OPERATOR                     |NAME    |EST. ROWS|COST     |
------------------------------------------------------------
|0 |PX COORDINATOR               |        |784080000|295962589|
|1 | EXCHANGE OUT DISTR          |:EX10001|784080000|179228805|
|2 |  HASH JOIN                  |        |784080000|179228805|
|3 |   PX PARTITION ITERATOR     |        |200000   |77361    |
|4 |    TABLE SCAN               |p1      |200000   |77361    |
|5 |   EXCHANGE IN DISTR         |        |400000   |184498   |
|6 |    EXCHANGE OUT DISTR (PKEY)|:EX10000|400000   |154722   |
|7 |     PX PARTITION ITERATOR   |        |400000   |154722   |
|8 |      TABLE SCAN             |p2      |400000   |154722   |
============================================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(p1.c1, p1.c2, p2.c1, p2.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(p1.c1, p1.c2, p2.c1, p2.c2)]), filter(nil), dop=1
  2 - output([p1.c1], [p1.c2], [p2.c1], [p2.c2]), filter(nil),
      equal_conds([p1.c1 = p2.c2]), other_conds(nil)
  3 - output([p1.c1], [p1.c2]), filter(nil)
  4 - output([p1.c1], [p1.c2]), filter(nil),
      access([p1.c1], [p1.c2]), partitions(p[0-1])
  5 - output([p2.c1], [p2.c2]), filter(nil)
  6 - (#keys=1, [p2.c2]), output([p2.c1], [p2.c2]), filter(nil), dop=1
  7 - output([p2.c1], [p2.c2]), filter(nil)
  8 - output([p2.c1], [p2.c2]), filter(nil),
      access([p2.c1], [p2.c2]), partitions(p[0-3])
```

默认情况下针对 `p1table` 与 `p2table`（两张表需要查询的分区数都大于 1）都会采用并行查询，默认的 `dop` 值为 1。同样，也可以通过使用 `PARALLEL` Hint 的方式来改变并行度。

如下例所示，改变 `JOIN` 的条件为 `p1table.c1=p2table.c2` 和 `p2table.c1=1`，这样针对 `p2table` 仅仅会选择单个分区，执行计划如下所示：

```
obclient>EXPLAIN SELECT  * FROM p1table p1 JOIN p2table p2 ON p1.c1=p2.c2 AND p2.c1=1\G;
*************************** 1. row ***************************
Query Plan: ============================================================
|ID|OPERATOR                      |NAME    |EST. ROWS|COST   |
------------------------------------------------------------
|0 |PX COORDINATOR                |        |1940598  |1394266|
|1 | EXCHANGE OUT DISTR           |:EX10001|1940598  |1105349|
|2 |  MERGE JOIN                  |        |1940598  |1105349|
|3 |   SORT                       |        |200000   |657776 |
|4 |    PX PARTITION ITERATOR     |        |200000   |77361  |
|5 |     TABLE SCAN               |p1      |200000   |77361  |
|6 |   SORT                       |        |990      |2092   |
|7 |    EXCHANGE IN DISTR         |        |990      |457    |
|8 |     EXCHANGE OUT DISTR (PKEY)|:EX10000|990      |383    |
|9 |      TABLE SCAN              |p2      |990      |383    |
============================================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(p1.c1, p1.c2, p2.c1, p2.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(p1.c1, p1.c2, p2.c1, p2.c2)]), filter(nil), dop=1
  2 - output([p1.c1], [p1.c2], [p2.c1], [p2.c2]), filter(nil),
      equal_conds([p1.c1 = p2.c2]), other_conds(nil)
  3 - output([p1.c1], [p1.c2]), filter(nil), sort_keys([p1.c1, ASC]), local merge sort
  4 - output([p1.c1], [p1.c2]), filter(nil)
  5 - output([p1.c1], [p1.c2]), filter(nil),
      access([p1.c1], [p1.c2]), partitions(p[0-1])
  6 - output([p2.c1], [p2.c2]), filter(nil), sort_keys([p2.c2, ASC])
  7 - output([p2.c1], [p2.c2]), filter(nil)
  8 - (#keys=1, [p2.c2]), output([p2.c1], [p2.c2]), filter(nil), is_single, dop=1
  9 - output([p2.c1], [p2.c2]), filter(nil),
      access([p2.c1], [p2.c2]), partitions(p1)

1 row in set
```

通过计划可以看出，`p2table` 仅需要扫描一个分区，在默认情况下不进行并行查询；`p1table` 需要扫描两个分区，默认情况下进行并行查询。同样，也可以通过添加 `PARALLEL` Hint 的方式改变并行度，使 `p2table` 针对一个分区的查询变为分区内并行查询。

关闭并行查询

分区表在查询的时候会自动启动并行查询（查询分区个数大于 1），如果不想启动并行查询，可以使用添加 Hint `/*+ NO_USE_PX */` 来关闭并行查询。

例如，针对分区表 `ptable`，添加 Hint `/*+ NO_USE_PX */` 来关闭并行查询，通过生成的执行计划可以看出对 `ptable` 表的扫描没有进行并行查询。

```
obclient>EXPLAIN SELECT /*+ NO_USE_PX */ * FROM ptable\G;
*************************** 1. row ***************************
Query Plan: ==============================================
|ID|OPERATOR           |NAME  |EST. ROWS|COST   |
----------------------------------------------
|0 |EXCHANGE IN DISTR  |      |1600000  |737992|
|1 | EXCHANGE OUT DISTR|      |1600000  |618888|
|2 |  TABLE SCAN       |ptable|1600000  |618888|
==============================================

Outputs & filters:
-------------------------------------
  0 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil)
  1 - output([INTERNAL_FUNCTION(ptable.c1, ptable.c2)]), filter(nil)
  2 - output([ptable.c1], [ptable.c2]), filter(nil),
      access([ptable.c1], [ptable.c2]), partitions(p[0-15])

1 row in set
```

并行执行相关的系统视图

OceanBase 数据库提供了系统视图 `gv$sql_audit` 和 `v$sql_audit` 来查看并行执行的运行状态以及一些统计信息。

`gv$sql_audit` 和 `v$sql_audit` 包含字段较多，其中与并行执行相关的字段为：`qc_id` 、`dfo_id` 、`sqc_id` 和 `worker_id` 。

详细信息请参考**《SQL 调优指南》**。

6.1.5.4. 控制分布式执行计划

分布式执行计划可以使用 Hint 管理，以提高 SQL 查询性能。

分布式执行框架支持的 Hint 包括 `NO_USE_PX` 、`PARALLEL` 、`ORDERED` 、`LEADING` 、`USE_NL` 、`USE_HASH` 和 `USE_MERGE` 等。

NO_USE_PX Hint

如果某个查询确定不走并行执行框架，使用 `NO_USE_PX` 拉回数据并生成本地执行计划。示例如下：

```
obclient> SELECT /*+ no_use_px parallel(8) */ * FROM(
    SELECT /*+ no_use_px parallel(8) */ no_w_id, no_d_id, MAX(no_o_id) max_no_o_id, MIN(no_
o_id) min_no_o_id, COUNT(*) count_no
    FROM nord
    GROUP BY no_w_id, no_d_Id
) x
WHERE max_no_o_id - min_no_o_id+ 1!= count_no;
```

PARALLEL Hint

指定分布式执行的并行度。

启用 3 个 Worker 并行执行扫描，如下例所示：

```
obclient>SELECT /*+ PARALLEL(3) */ MAX(L_QUANTITY) FROM table_name;
```

> **说明**
>
> 在复杂查询中，调度器可以调度 2 个 DFO 并行流水执行，此时，启用的 Worker 数量为并行度的 2 倍，即 `PARALLEL * 2` 。

ORDERED Hint

`ORDERED` Hint 指定并行查询计划中 `JOIN` 的顺序，严格按照 `FROM` 语句中的顺序生成。

如下例所示，强制要求 `customer` 为左表， `orders` 为右表，并且使用 `NESTED LOOP JOIN` ：

```
obclient>CREATE TABLE lineitem(
    l_orderkey           NUMBER(20) NOT NULL ,
    l_linenumber         NUMBER(20) NOT NULL ,
    l_quantity           NUMBER(20) NOT NULL ,
    l_extendedprice      DECIMAL(10,2) NOT NULL ,
    l_discount           DECIMAL(10,2) NOT NULL ,
    l_tax                DECIMAL(10,2) NOT NULL ,
    l_shipdate           DATE NOT NULL,
    PRIMARY KEY(L_ORDERKEY, L_LINENUMBER));
Query OK, 1 row affected

obclient>CREATE TABLE customer(
    c_custkey            NUMBER(20) NOT NULL ,
    c_name               VARCHAR(25) DEFAULT NULL,
    c_address            VARCHAR(40) DEFAULT NULL,
    c_nationkey          NUMBER(20) DEFAULT NULL,
    c_phone              CHAR(15) DEFAULT NULL,
    c_acctbal            DECIMAL(10,2) DEFAULT NULL,
    c_mktsegment         CHAR(10) DEFAULT NULL,
    c_comment            VARCHAR(117) DEFAULT NULL,
    PRIMARY KEY(c_custkey));
Query OK, 1 row affected

obclient>CREATE TABLE orders(
    o_orderkey           NUMBER(20) NOT NULL ,
    o_custkey            NUMBER(20) NOT NULL ,
    o_orderstatus        CHAR(1) DEFAULT NULL,
    o_totalprice         DECIMAL(10,2) DEFAULT NULL,
    o_orderdate          DATE NOT NULL,
    o_orderpriority      CHAR(15) DEFAULT NULL,
    o_clerk              CHAR(15) DEFAULT NULL,
    o_shippriority       NUMBER(20) DEFAULT NULL,
    o_comment            VARCHAR(79) DEFAULT NULL,
    PRIMARY KEY(o_orderkey,o_orderdate,o_custkey));
Query OK, 1 row affected
```

```
Query OK, 1 row affected

obclient> INSERT INTO lineitem VALUES(1,2,3,6.00,0.20,0.01,'01-JUN-02');
Query OK, 1 row affected

obclient> INSERT INTO customer VALUES(1,'Leo',null,null,'13700461258',null,'BUILDING',null)
;
Query OK, 1 row affected

obclient> INSERT INTO orders VALUES(1,1,null,null,'01-JUN-20',10,null,8,null);
Query OK, 1 row affected

obclient>SELECT /*+ ORDERED USE_NL(orders) */o_orderdate, o_shippriority
        FROM customer, orders WHERE c_mktsegment = 'BUILDING' AND
         c_custkey = o_custkey GROUP BY o_orderdate, o_shippriority;

+-------------+----------------+
| O_ORDERDATE | O_SHIPPRIORITY |
+-------------+----------------+
| 01-JUN-20   |              8 |
+-------------+----------------+
1 row in set
```

在编写 SQL 时，`ORDERED` 较为有用，用户知道 `JOIN` 的最佳顺序时，可以将表按照顺序写在 `FROM` 的后面，然后加上 `ORDERED` Hint。

LEADING Hint

`LEADING` Hint 指定并行查询计划中最先 `JOIN` 哪些表，`LEADING` 中的表从左到右的顺序，也是 `JOIN` 的顺序。它比 `ORDERED` 有更大的灵活性。

> **说明**
>
> 如果同时使用 `ORDERED` 和 `LEADING`，仅 `ORDERED` 生效。

PQ_DISTRIBUTE Hint

PQ Hint 即 `PQ_DISTRIBUTE`，用于指定并行查询计划中的数据分布方式。PQ Hint 会改变分布式 `JOIN` 时的数据分发方式。

PQ Hint 的基本语法如下：

```
PQ_DISTRIBUTE(tablespec outer_distribution inner_distribution)
```

参数解释如下：

- `tablespec` 指定关注的表，关注 `JOIN` 的右表。

- `outer_distribution` 指定左表的数据分发方式。
- `inner_distribution` 指定右表的数据分发方式。

两表的数据分发方式共有以下六种：

- `HASH` , `HASH`
- `BROADCAST` , `NONE`
- `NONE` , `BROADCAST`
- `PARTITION` , `NONE`
- `NONE` , `PARTITION`
- `NONE` , `NONE`

其中，带分区的两种分发方式要求左表或右表有分区，而且分区键就是 `JOIN` 的键。如果不满足要求的话，PQ Hint 不会生效。

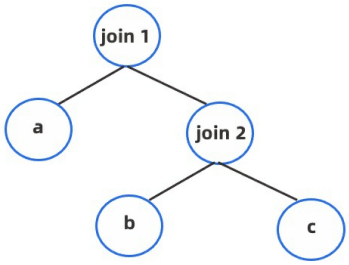
```
obclient>CREATE TABLE t1(c1 INT PRIMARY KEY, c2 INT, c3 INT, c4 DATE);
Query OK, 0 rows affected

obclient>CREATE INDEX i1 ON t1(c3);
Query OK, 0 rows affected

obclient>CREATE TABLE t2(c1 INT(11) NOT NULL, c2 INT(11) NOT NULL, c3 INT(11)
       NOT NULL, PRIMARY KEY (c1, c2, c3)) PARTITION BY KEY(c2) PARTITIONS 4;
Query OK, 0 rows affected

obclient>EXPLAIN BASIC SELECT /*+USE_PX PARALLEL(3) PQ_DISTRIBUTE
        (t2 BROADCAST NONE) LEADING(t1 t2)*/ * FROM t1 JOIN t2 ON
         t1.c2 = t2.c2\G;
*************************** 1. row ***************************
Query Plan: ==============================================
|ID|OPERATOR                          |NAME    |
----------------------------------------------
|0 |PX COORDINATOR                    |        |
|1 | EXCHANGE OUT DISTR               |:EX10001|
|2 |  HASH JOIN                       |        |
|3 |   EXCHANGE IN DISTR              |        |
|4 |    EXCHANGE OUT DISTR (BROADCAST)|:EX10000|
|5 |     PX BLOCK ITERATOR            |        |
|6 |      TABLE SCAN                  |t1      |
|7 |   PX BLOCK ITERATOR              |        |
|8 |    TABLE SCAN                    |t2      |
==============================================
```

USE_NL Hint

`USE_NL` Hint 指定使用 `NESTED LOOP JOIN`，并且需要满足 `USE_NL` 中指定的表是 `JOIN` 的右表。

如下例所示，如果希望 `join1` 为 `NESTED LOOP JOIN`，则 Hint 写法为 `LEADING(a, (b,c)) USE_NL((b,c))`。

当 `USE_NLJ` 和 `ORDERED`、`LEADING` Hint 一起使用时，如果 `USE_NLJ` 中注明的表不是右表，则 `USE_NLJ` Hint 会被忽略。



USE_HASH Hint

`USE_HASH` Hint 指定使用 `HASH JOIN`，并且需要满足 `USE_HASH` 中指定的表是 `JOIN` 的右表。

> 说明
>
> 如果没有使用 `ORDERED` 和 `LEADING` Hint，并且优化器生成的联接顺序中指定的表之间不是直接 `JOIN` 的关系，那么 `USE_HASH` Hint 会被忽略。

USE_MERGE Hint

`USE_MERGE` Hint 指定使用 `MERGE JOIN`，并且需要满足 `USE_MERGE` 中指定的表是 `JOIN` 的右表。

> 说明
>
> 如果没有使用 `ORDERED` 和 `LEADING` Hint，并且优化器生成的联接顺序中指定的表之间不是直接 `JOIN` 的关系，那么 `USE_MERGE` Hint 会被忽略。

6.1.5.5. 并行查询的参数调优

Oceanbase 数据库并行查询（PX）的参数决定了并行查询的速度，主要包括并行度和 EXCHANGE 等相关参数 。

并行度参数

并行度相关参数主要决定每个查询并发时的 Worker 个数。详细信息如下表所示。

参数名称	描述	取值范围	默认值	配置建议
parallel_servers_target	当准备排队之前，控制检查查询要求的并行度和已统计的 Worker 总和是否超过该值。如果超过该值，则查询需要排队，否则查询继续执行。	[0, 1800]	10（目前会根据 CPU 个数计算得到，以实际大小为准）	该参数主要是控制 PX 场景下，当准备进行并行查询时，如果没有足够 Worker 处理该查询，决定是否继续进行还是排队等待。
_force_parallel_query_dop	该参数在会话中指定查询 SQL 的默认并行度，在没有指定 PARALLEL Hint 情况下，查询SQL 的并行度受此变量控制。	[1, +∞]	1	根据实际需要。例如同一个会话中要运行一批并行查询 SQL 又不想手动给每条 SQL 加上 Hint 时，建议使用此参数。
_force_parallel_dml_dop	该参数在会话中指定 DML SQL 的默认并行度，在没有指定 PARALLEL Hint 情况下，DML SQL 的并行度受此变量控制。	[1,+∞]	1	根据实际需要。例如同一个会话中要运行一批并行DML SQL 又不想手动给每条 SQL 加上 Hint 时，建议使用此参数。

可以通过 `SHOW VARIABLES` 来查看这些参数的值。

EXCHANGE（Shuffle）参数

`EXCHANGE` （Shuffle）参数主要用来控制在每个 DFO 之间进行数据传输时的参数控制，也就是数据进行 Shuffle 时的内存控制。OceanBase 数据库将数据传输封装成了叫做 DTL（Data Transfer layer）的模块。

参数名称	描述	取值范围	默认值	配置建议

参数名称	描述	取值范围	默认值	配置建议
dtl_buffer_size	控制 EXCHANG 算子之间（即 Transmit 和 Receive 之间）发送数据时，每次发送数据的 Buffer 的大小。即当数据达到了该值上限才进行发送，减少每行传输的代价。	[0, 1800]	10（目前会根据 CPU 个数计算得到，以实际大小为准）	PX 场景下，EXCHANGE 之间发送数据依赖于该参数大小，一般不需要调整该参数，如果是为了减少发送数据次数等可以尝试进行修改，一般不建议修改该值大小。

可以通过 `SHOW PARAMETERS` 来查看参数的值，如下例所示：

```
obclient>SHOW PARAMETERS LIKE '%dtl%';
+-------+----------+----------------+----------+-----------------+-----------+-------+-----
----------+----------+---------+---------+------------------+
| zone  | svr_type | svr_ip         | svr_port | name            | data_type | value | info
| section  | scope   | source  | edit_level        |
+-------+----------+----------------+----------+-----------------+-----------+-------+-----
----------+----------+---------+---------+------------------+
| zone1 | observer | 100.81.152.114 |    36500 | dtl_buffer_size | NULL      | 64K   | to b
e removed | OBSERVER | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
+-------+----------+----------------+----------+-----------------+-----------+-------+-----
----------+----------+---------+---------+------------------+
1 row in set
```

其他并行相关参数

参数名称	描述	取值范围	默认值	配置建议
_enable_px_batch_rescan	控制在 NLJ 生成分布式 PX RESCAN 计划执行时是否使用 BATCH RESCAN, 可以获得更好的性能.	• True • False	True	默认开启会获得更好的性能，但会消耗更多的内存。

参数名称	描述	取值范围	默认值	配置建议
_bloom_filter_enabled	控制在 HASH JOIN 场景下是否开启 BLOOM FILTER。	• True • False	True	在并行度大于 1 的情况下默认开启，如果 HASH JOIN 的连接条件过滤性不佳，打开 BLOOM FILTER 会有额外的开销，在此场景可以考虑关闭此功能。

6.2. PL

6.2.1. PL 概念

6.2.1.1. Oracle 模式

6.2.1.1.1. 子程序

子程序是包含很多 SQL 和 PL 语句的 PL 单元，以解决特定的问题或者执行一组相关的任务。

子程序可以包含参数，具体值由调用者传入。子程序可以是一个存储过程或者函数。典型的用法是使用存储过程进行一个操作，使用函数进行计算并返回一个值。

存储子程序是存储在数据库内部的子程序，为很多不同数据库应用程序实现复杂的逻辑运算。存储子程序包含如下三类：

- 独立的子程序，即在 Schema 内创建的子程序。
- 包内部的子程序，即在包体内部创建的子程序。
- 嵌套的子程序，即在 PL 块内创建的子程序。

独立的子程序对测试程序逻辑很方便，但是大量的子程序不是很方便管理。因此建议在程序逻辑确定后，将独立的子程序按照业务模块放到不同的包里。

子程序是其他可维护性功能的重要组成部分，例如程序包和触发器等。

子程序结构

子程序以子程序标题开头，该标题指定其名称和其参数列表（可选）。

子程序结构跟 PL 块结构一致，包括：

- 声明部分（可选）

  声明部分包括类型、常量、变量、异常、显式游标和嵌套子程序的声明。这些项对于子程序都是本地的，当子程序执行结束时就都不存在了。

- 执行部分（必选）

  执行部分包括赋值语句、控制执行语句和数据操作语句。

- 异常处理部分（可选）

  异常处理部分包括处理异常（运行时错误）的代码。

在子程序中添加注释会增加程序的可读性，注释可以出现在子程序的任意位置，编译器会忽略注释。单行注释是以双横线（--）开头的，注释范围到行尾截止。多行注释可以以斜杠和星号（/*）开头，以星号和斜杠（*/）结尾。

一个存储过程的结构如下：

```
PROCEDURE name [ ( parameter_list ) ]
{ IS | AS }
[ declarative_part ]
BEGIN    -- 开始执行部分
statement; [ statement; ]...
[ EXCEPTION ]
exception_handler; [ exception_handler; ]... ]
END;
```

一个函数的结构如下，与存储过程相比，多了一个或多个 `RETURN` 子句：

```
FUNCTION name [ ( parameter_list ) ] RETURN data_type [ clauses ]
{ IS | AS }
[ declarative_part ]
BEGIN
statement; [ statement; ]...
[ EXCEPTION ]
exception_handler; [ exception_handler; ]... ]
END;
```

存储过程和函数跟 `IS | AS` 之间的代码是子程序的声明，声明部分、执行部分和异常处理部分是子程序的内容体。

子程序优点

相比于客户端程序，子程序有如下优点：

- 提高性能

  减少网络传输带来的代价；可以提前编译，甚至使用 Cache 机制缓存，减少执行时期的性能消耗。

- 减少内存消耗

  数据库的共享内存机制会在多个用户执行同一个存储过程时减少内存的消耗。

- 提高生产力

  存储过程可以为用户减少代码逻辑，提高用户的生产力。

- 提高安全性

  通过指定存储过程的权限信息，提高安全性。

- 具有可继承性

  用户可以通过获取权限来访问其他用户定义好的存储过程。

子程序的执行

可以通过以下三种方式执行子程序：

- 使用客户端工具（OBClient）
- 在应用程序中直接调用子程序
- 在子程序或者触发器中调用

依赖关系管理

在子程序的包体中引用的对象构成子程序的依赖对象。数据库自动跟踪和管理这些依赖关系。

例如，如果用户修改了被子程序依赖的表定义，那么子程序必须被重新编译以确定子程序是否依然有效。通常这种依赖变更后子程序的重新编译由数据库依赖管理单元自动完成。

6.2.1.1.2. 存储过程

存储过程是 SQL 语句和可选控制流语句的预编译集合，PL 引擎将其作为一个单元进行处理。一个存储过程可以引用其它存储过程，并且可以返回多个变量。

存储过程的结构

一个存储过程的结构如下：

```
PROCEDURE name [ ( parameter_list ) ]
{ IS | AS }
[ declarative_part ]
BEGIN    -- 开始执行部分
statement; [ statement; ]...
[ EXCEPTION ]
exception_handler; [ exception_handler; ]... ]
END;
```

存储过程的创建

可以通过 `CREATE PROCEDURE` 语句创建存储过程。创建存储过程的语法如下：

```
CREATE [OR REPLACE] PROCEDURE Procedure_name
[ (argment [ { IN | IN OUT }] Type,
      argment [ { IN | OUT | IN OUT } ] Type ]
    [ AUTHID DEFINER | CURRENT_USER ]
{ IS | AS }
delarification_block
BEGIN
    procedure_body
EXCEPTION
  exception_handler
END;
```

无参数的存储过程示例如下：

```
obclient> CREATE TABLE loghistory
      (userid VARCHAR2(20),
      logdate DATE DEFAULT SYSDATE);
Query OK, 0 rows affected

obclient> CREATE OR REPLACE PROCEDURE userlogin
      IS
  BEGIN
      INSERT INTO loghistory (userid) VALUES (USER);
  END;
  /
Query OK, 0 rows affected
```

存储过程的调用

存储过程建立完成后，用户通过授权可以在 OBClient 、OceanBase Developer Center 或第三方开发工具中来调用运行。

示例如下：

```
obclient> SELECT * FROM loghistory;
Empty set

obclient> BEGIN
     userlogin;
     END;
     /
Query OK, 0 rows affected

obclient> SELECT * FROM loghistory;
+--------+-----------+
| USERID | LOGDATE   |
+--------+-----------+
| HR     | 27-SEP-20 |
+--------+-----------+
1 row in set

obclient> COMMIT;
Query OK, 0 rows affected
```

6.2.1.1.3. 函数

函数与存储过程完全相同，唯一的不同点是函数在执行结束后会返回一个值，而存储过程不会。

函数的结构

函数的结构如下，与存储过程相比，函数在创建时需要声明返回值的类型：

```
FUNCTION name [ ( parameter_list ) ] RETURN sp_type
{ IS | AS }
[ declarative_part ]
BEGIN -- executable part begins
statement; [ statement; ]...
[ EXCEPTION ]
exception_handler; [ exception_handler; ]... ]
END;
```

函数的创建

通过 `CREATE FUNCTION` 语句创建函数，语法如下：

```
CREATE OR REPLACE FUNCTION func RETURN INT
IS
BEGIN
RETURN 1;
END;
/
```

在包内部创建函数，语法如下：

```
CREATE OR REPLACE PACKAGE pack IS
  FUNCTION func RETURN INT;
 END;
 /
 CREATE OR REPLACE PACKAGE BODY pack IS
   FUNCTION func RETURN INT
   IS
   BEGIN
   RETURN 1;
   END;
 END;
 /
```

在子程序内部定义函数，语法如下：

```
DECLARE
  FUNCTION func RETURN INT
  IS
  BEGIN
  RETURN 1;
  END;
BEGIN
  NULL;
END;
/
```

6.2.1.1.4. 触发器

触发器是 OceanBase 数据库中提供的功能，它与存储过程和函数类似，包括声明、执行和异常处理过程的 PL 块，使用 PL 编写的编译存储单元。

触发器是个独立的对象，当某个触发语句执行时自动地隐式运行，而且触发器不能接收参数。这里的触发语句指的是对数据库的表进行的 `INSERT` 、 `UPDATE` 以及 `DELETE` 操作。

触发器的优点

正确使用触发器可以使应用程序的构建和部署更加简单且健壮。

您可以通过触发器强制所有客户端程序执行底层业务逻辑。例如，多个客户端应用程序都访问某张表，如果这个表上的触发器保证了插入数据需要执行的逻辑，那么这个业务逻辑就不需要在每个客户端都执行，由于应用程序无法绕过触发器，因此会自动使用触发器中的业务逻辑。

请在需要的情况下使用触发器，避免触发器过度使用。过度使用触发器会导致复杂的互相依赖关系，在大型应用程序中会变得难以维护。例如，当触发器被触发时，其中执行的 SQL 语句可能会触发其他的触发器，从而导致级联触发，可能会产生预期之外的结果。

触发器类型

OceanBase 数据库可以创建以下几种类型触发器：

- 行级触发器

  行级触发器创建在实体表上，每次表受到触发语句影响时会触发一个行级触发器。例如，一条语句更新多行数据，每条受影响的数据都会触发一次触发器。如果触发语句不影响任何行数据，则不会运行触发器。

- 语句级触发器

  语句级触发器创建在实体表上，每次执行触发语句时都会自动触发一次，无论该触发语句是否影响了表中的任何行数据。例如，一条语句更新了表中的 100 条数据，则语句级 `UPDATE` 触发器仅触发一次。

- INSTEAD OF 触发器

  INSTEAD OF 触发器创建在视图上，当对视图执行触发语句时自动触发。INSTEAD OF 触发器可以用来修改无法直接通过 DML 语句修改的视图。

触发器的触发时机

您可以定义触发器的触发时机，触发时机是指触发操作在触发语句之前还是之后运行。

行级触发器和语句级触发器可以指定如下触发时机：

- 在触发语句执行之前
- 在每行数据被触发语句修改之前
- 在每行数据被触发语句修改之后
- 在触发语句执行之后

对于语句级和行级触发器来说，BEFORE 触发器可以在对数据进行更改之前增强安全性并执行业务规则。AFTER 触发器非常适合用来记录操作日志。

一个简单的触发器的触发时机有如下四种：

- 在事件执行之前（语句级别的 BEFORE 触发器）
- 在事件执行之后（语句级别的 AFTER 触发器）
- 在每行被事件执行影响之前（行级的 BEFORE 触发器）

- 在每行被事件执行影响之后（行级的 AFTER 触发器）

触发器的触发顺序如下：

- 同一类型的触发器的执行顺序是不确定的，当前暂不支持指定触发器执行顺序。
- 一个 DML 语句可能会触发多个简单的触发器，触发执行的先后顺序是：语句级 BEFORE 触发器 -> 行级 BEFORE 触发器 -> 行级 AFTER 触发器 -> 语句级 AFTER 触发器。

> **注意**
>
> OceanBase 数据库 V2.2.7x 版本及以下只支持表上的行级触发器。

创建触发器

创建触发器的基本语法结构如下：

```
CREATE [OR REPLACE] TRIGGER trigger_name triggering_statement
  [trigger_restriction]
BEGIN
 triggered_action;
END
```

一个触发器的基本结构如下：

- 触发器名字（Trigger Name）

  在同一个 Database 下，触发器名字必须是唯一的。例如，触发器名字可能是 row_triggger_on_employees。

- 触发语句（Triggering Statement）

  触发语句是导致调用触发器的 SQL 语句。例如，用户更新一个表。

- 触发器限制（Trigger Restriction）

  触发器限制是指定义一个布尔表达式，该表达式为真时才能触发触发器。例如，员工表上的触发器定义一个限制，只有家住北京的员工才能触发此触发器。

- 触发器动作（Trigger Action）

  触发器动作是指触发语句执行并且触发器限制为真时运行的触发器内部的代码块。例如，在员工表中插入一条数据。

创建触发器示例

在表 `emp_msg` 上创建一个触发器，当对表执行 `INSERT` 、 `UPDATE` 、 `DELETE` 语句时触发该触发器。

当往 `emp_msg` 表中插入数据时，会同时往 `employees` 中插入一条数据，当删除 `emp_msg` 表中的数据时，会同时删除 `employees` 表中 `id` 相同的数据，当更新 `emp_msg` 表中的数据时，会更新 `employees` 表中 `id` 相同的数据。

```
CREATE TABLE employees (id INT, name VARCHAR2(20), WORK_YEAR int);

CREATE TABLE emp_msg (id INT PRIMARY KEY, name VARCHAR2(20), address VARCHAR2(100));

CREATE OR REPLACE TRIGGER tri_emp_msg BEFORE INSERT OR UPDATE OR DELETE ON emp_msg
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO employees VALUES (:NEW.id, :NEW.name, 0);
  ELSIF DELETING THEN
    DELETE FROM employees WHERE id = :OLD.id;
  ELSE
    UPDATE employees SET name = :NEW.name WHERE id = :NEW.id;
  END IF;
END;
/
```

创建完触发器后，执行如下示例中的 DML 语句：

```
obclient> INSERT INTO emp_msg VALUES (1, 'Curry', 'BeiJing');
Query OK, 1 row affected

obclient> SELECT * FROM emp_msg WHERE id = 1;
+----+-------+---------+
| ID | NAME  | ADDRESS |
+----+-------+---------+
|  1 | Curry | BeiJing |
+----+-------+---------+
1 row in set

obclient> SELECT * FROM  employees WHERE id = 1;
+------+-------+-----------+
| ID   | NAME  | WORK_YEAR |
+------+-------+-----------+
|    1 | Curry |         0 |
+------+-------+-----------+
1 row in set


obclient> UPDATE emp_msg SET name = 'Stephen Curry' WHERE id = 1;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

obclient> SELECT * FROM emp_msg WHERE id = 1;
+----+---------------+---------+
| ID | NAME          | ADDRESS |
+----+---------------+---------+
|  1 | Stephen Curry | BeiJing |
+----+---------------+---------+
1 row in set

obclient> SELECT * FROM  employees WHERE id = 1;
+------+---------------+-----------+
| ID   | NAME          | WORK_YEAR |
+------+---------------+-----------+
|    1 | Stephen Curry |         0 |
+------+---------------+-----------+
1 row in set


obclient> DELETE FROM emp_msg WHERE id = 1;
Query OK, 1 row affected

obclient> SELECT * FROM emp_msg WHERE id = 1;
Empty set

obclient> SELECT * FROM  employees WHERE id = 1;
Empty set
```

6.2.1.1.5. 程序包

程序包 Package（简称包）是一组相关的子程序，包括子程序使用的游标和变量，一起存储在数据库中作为一个结合单元来使用。

OceanBases 数据库提供了许多系统包，这些包扩展了数据库功能，您可以在创建应用程序的时候使用系统包。例如，DBMS_RANDOM 包可以很容易的获取一个随机值。

包的优点

PL 包为开发应用程序提供了很多便利和优势。优点如下：

- 封装功能

  包具有面向对象程序设计语言的特点，您可以将存储过程、变量、自定义数据类型等封装在一个包内，这样更容易组织和管理这些更小的单元。包类似于 C++ 和 Java 语言中的类，其中变量相当于类中的成员变量，存储过程和函数相当于类方法。包封装还简化了权限管理，授予包的访问权限，被授予者就可以访问包内的各个结构。

- 较高的数据安全性

  包内定义的变量、游标、存储过程可以设置为公共的或私有的，“公共”表示包的用户可以直接访问它，“私有”表示包的用户不可以直接访问它。

  例如，一个包有三个变量和五个存储过程，您可以将两个变量和三个存储过程设置为私有的，那么这两个变量和三个存储过程只能在包体内访问，包的使用者只能访问剩下的一个变量和两个存储过程。

创建包

创建包需要两部分内容，包头和包体。包头需要声明所有公共的结构，而包体需要定义包的所有结构，包括公共部分和私有部分。

如下示例中，创建了一个 `students` 表，和 `students_adm` 的包。包中有两个存储过程，当有学生入学时，调用 `students_adm.stu_entrance` 会往 `students` 表中插入学生的信息，当有学生毕业时，调用 `students_adm.graduate` 会从 `students` 表中删除学生的信息。

```
CREATE TABLE students (id INT, name VARCHAR2(100), class INT, grade INT, dorm VARCHAR2(100)
, year INT);

CREATE OR REPLACE PACKAGE students_adm AS
  PROCEDURE stu_entrance(id INT, name VARCHAR2, class INT, grade INT, dorm VARCHAR2, year I
NT);
  PROCEDURE graduate(id INT);
END students_adm;
/

CREATE OR REPLACE PACKAGE BODY students_adm AS
 PROCEDURE stu_entrance(id INT, name VARCHAR2, class INT, grade INT, dorm VARCHAR2, year IN
T) AS
  BEGIN
    INSERT INTO students VALUES (id, name, class, grade, dorm, year);
  END;
PROCEDURE graduate(id INT) AS
  BEGIN
    DELETE from students s WHERE s.id = id;
  END;
END students_adm;
/

obclient> CALL students_adm.stu_entrance(1001, 'Curry', 5, 10, '1-1-3', 12);
Query OK, 0 rows affected

obclient> SELECT * from students WHERE id = 1001;
+------+-------+-------+-------+-------+------+
| ID   | NAME  | CLASS | GRADE | DORM  | YEAR |
+------+-------+-------+-------+-------+------+
| 1001 | Curry |     5 |    10 | 1-1-3 |   12 |
+------+-------+-------+-------+-------+------+
1 row in set

obclient> CALL students_adm.graduate(1001);
Query OK, 0 rows affected

obclient> SELECT * from students WHERE id = 1001;
Empty set
```

6.2.1.2. MySQL 模式

6.2.1.2.1. 子程序

子程序是包含很多 SQL 和 PL 语句的 PL 单元，以解决特定的问题或者进行一组相关的任务。

子程序可以包含参数，具体值由调用者传入。子程序可以是一个存储过程或者函数。典型的用法是使用存储过程执行一个操作，使用函数计算并返回一个值。

存储程序是存储在数据库内部的子程序，为很多不同数据库应用子程序实现复杂的逻辑运算。MySQL 模式的子程序只有独立的程序，即在 Schema 内创建的程序。

> **注意**
>
> MySQL 模式的子程序遵循 SQL 标准中存储程序的标准，与 Oracle 模式的子程序（PL）在语法与功能上都有显著的区别。

子程序结构

子程序结构跟 PL 块结构一致，包括：

- 声明部分（可选）

  声明部分包括类型、常量、变量、异常、显式游标和嵌套程序的声明。这些项对于子程序都是本地的，当子程序执行结束时就都不存在了。

- 执行部分（必选）

  执行部分包括赋值语句、控制执行语句和数据操作语句。

- 异常处理部分（可选）

  异常处理部分包括处理异常（运行时错误）的代码。

在子程序中添加注释会增加程序的可读性，注释可以出现在子程序的任意位置，编译器会忽略注释。单行注释是以双横线（--）开头的，注释范围到行尾截止。多行注释可以以斜杠和星号（/*）开头，以星号和斜杠（*/）结尾。

一个存储过程的结构如下：

```
PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

一个函数的结构如下，与存储过程相比，多了 `RETURNS` 子句：

```
FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

子程序优点

相比于客户端程序，子程序有如下优点：

- 提高性能

  减少网络传输带来的代价；可以提前编译，甚至使用 Cache 机制缓存，减少执行时期的性能消耗。

- 减少内存消耗

  数据库的共享内存机制会在多个用户执行同一个存储过程时减少内存的消耗。

- 提高生产力

  存储过程可以为用户减少代码逻辑，提高用户的生产力。

- 提高安全性

  通过指定存储过程的权限信息，提高安全性。

- 具有可继承性

  用户可以通过获取权限来访问其他用户定义好的存储过程。

子程序的执行

可以通过以下三种方式执行子程序：

- 使用 MySQL 工具
- 通过数据库应用程序调用
- 通过另一个存储过程或者触发器调用

子程序的元数据

可以通过以下方法获取子程序相关的元数据：

- 查询 `INFORMATION_SCHEMA` 的 ROUTINES 视图，可以获取所有的子程序。

- 通过 `SHOW CREATE PROCEDURE` 可以查看某个存储过程的定义
- 通过 `SHOW CREATE FUNCTION` 可以查看某个函数的定义
- 通过 `SHOW CREATE TRIGGER` 可以查看某个触发器的定义

6.2.1.2.2. 存储过程

存储过程是一种没有直接返回值的子程序。当参数类型为 OUT 时，存储过程也可以给调用者返回值。

存储过程的结构

一个存储过程的结构如下：

```
CREATE
    [DEFINER = user]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

存储过程的创建

存储过程通过 `CREATE PROCEDURE` 语句创建存储过程。MySQL 模式下与 Oracle 模式比较显著的区别如下：

- `DECLARE` 模块要放在 `BEGIN END;` 模块中，并且所有定义声明完成后才能定义其他语句。
- 可以定义 DETERMINISTIC / LANGUAGE SQL 等 sp_create_chistic 信息以丰富存储过程的使用。
- MySQL 模式下的存储过程不存在重载情况，所以删除时不用考虑参数类型。

示例如下：

```
obclient> CREATE TABLE city(CountryCode CHAR(3))/
Query OK, 0 rows affected

obclient> CREATE PROCEDURE citycount (IN country CHAR(3), OUT cities INT)
     BEGIN
        SELECT COUNT(*) INTO cities FROM city
         WHERE CountryCode = country;
     END/
Query OK, 0 rows affected
```

存储过程的调用

创建好的存储过程，需要使用 `CALL` 语句来调用，但是它不能作为 SQL 表达式的一部分被调用。

```
obclient> CALL citycount('JPN', @cities);/
+--------+
| cities |
+--------+
|      0 |
+--------+
1 row in set

obclient> SELECT @cities/
+----------+
| @cities |
+----------+
|       0 |
+----------+
1 row in set

obclient> DROP PROCEDURE citycount/
Query OK, 0 rows affected
```

6.2.1.2.3. 函数

函数可以理解为一种特殊的子程序，函数可以有返回值，并且所有参数都是 IN 类型。

函数的结构

函数的结构如下，与存储过程相比，至少多一个或多个 `RETURNS` 子句：

```
CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

函数的优点

函数的优点与子程序基本相同，但是因为函数有返回值所以函数可以作为表达式使用。将功能定义为函数，并在表达式中使用，这种用法更灵活、更丰富、更符合编程习惯。

函数的创建

函数也要通过 `CREATE` 语句进行创建，并需满足如下条件：

- 函数需要考虑返回值的设置。
- 所有函数必须至少有一个 `RETURNS` 语句。

```
obclient> CREATE FUNCTION hello (s CHAR(20)) RETURNS CHAR(50) DETERMINISTIC
     RETURN CONCAT('Hello, ',s,'!');/
Query OK, 0 rows affected

obclient> SELECT hello('world');/
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set
```

- 函数要作为表达式被调用，需要与其他语句结合使用。

6.2.1.2.4. 触发器

OceanBase 数据库 MySQL 模式兼容 MySQL 5.6 的触发器。触发器是与表相关的数据库对象，触发器可以在语句插入、更新或删除关联表中的行时被激活。例如，您可以通过 INSERT 或 LOAD DATA 语句插入行，并且每插入一行就会激活一次 INSERT 触发器。如果批量插入两行数据，则会出发两次触发器。

触发器可以设置为在触发事件之前或之后激活。例如，您可以在插入表的每一行之前或在更新的每一行之后激活触发器。

触发器类型

OceanBase 数据库 MySQL 模式当前主要支持以下类型的触发器：

- `INSERT` 型触发器：表示插入某一行时激活触发器，可以通过 `INSERT` 、 `LOAD DATA` 、 `REPLACE` 语句触发。
- `UPDATE` 型触发器：表示更改某一行时激活触发器，可以通过 `UPDATE` 语句触发。
- `DELETE` 型触发器：删除某一行时激活触发器，可以通过 `DELETE` 、 `REPLACE` 语句触发。

比较特殊的是 `INSERT INTO ... ON DUPLICATE KEY UPDATE` 语句，每个行都会激活一个 `BEFORE INSERT` 触发器，接着是 `AFTER INSERT` 触发器或者 `BEFORE UPDATE` 与 `AFTER UPDATE` 触发器，是 `AFTER INSERT` 触发器还是 `BEFORE UPDATE` 与 `AFTER UPDATE` 触发器主要取决于行是否有重复的键。

创建触发器

您可以使用 `CREATE TRIGGER` 语句创建触发器。

创建触发器的用户需要具备以下权限：

- 当前触发器关联的表的权限，包括 `SELECT` 、 `INSERT` 、 `UPDATE` 、 `DELETE` 等权限。
- 触发器权限，即 `CREATE TRIGGER` 权限。
- 触发器激活后要执行的语句的权限。

创建触发器的 SQL 语法如下：

```
CREATE
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }
```

语法说明：

- 触发器名称 `trigger_name` 必须保持唯一。

- `tbl_name` 表示建立触发器的表名，即在哪张表上建立触发器。
- `BEFORE` 或 `AFTER` 表示触发动作的时间，例如触发器将在每一行插入到表中之前还是之后激活。
- `INSERT` 、 `UPDATE` 或 `DELETE` 表示触发事件，即激活触发器的操作类型。
- `FOR EACH ROW` 用于定义触发器的主体，每次触发器激活时就会执行该语句，对于受触发事件影响的每一行都会执行一次。

OceanBase 数据库中也定义了 `NEW.columnName` 和 `OLD.columnName` ：

- 在 `INSERT` 型触发器中， `NEW.columnName` 用来表示将要（ `BEFORE` ）或已经（ `AFTER` ）插入的新数据。其中， `columnName` 为相应数据表的某一列名。
- 在 `UPDATE` 型触发器中， `OLD.columnName` 用来表示将要或已经被修改的原数据， `NEW.columnName` 用来表示将要或已经修改后的新数据。
- 在 `DELETE` 型触发器中， `OLD.columnName` 用来表示将要或已经被删除的原数据。
- `OLD.columnName` 是只读的，而 `NEW.columnName` 则可以在触发器中使用 `SET` 赋值。

示例：创建一个触发器 `test_trg` ，并将其与表 `test` 相关联，以激活 `INSERT` 操作，且触发器充当累加器，将插入到表的列的值求和。

```
obclient> CREATE TABLE test (user_id INT, user_num DECIMAL(10,2));
Query OK, 0 rows affected

obclient> CREATE TRIGGER test_trg BEFORE INSERT ON test
       FOR EACH ROW SET @sum = @sum + NEW.user_num;
Query OK, 0 rows affected
```

此外，如果需要定义的触发器有多条执行语句时，可以使用 `BEGIN … END` 语句，分别表示整个代码块的开始和结束。

`BEGIN … END` 语句的语法为：

```
BEGIN
[statement_list]
END
```

其中， `statement_list` 表示一个或多个语句的列表，列表内的每条语句都必须使用分号（;）结尾。由于在 SQL 语句中，分号（;）是语句结束的标识符，遇到分号表示该段语句已经结束，系统就会开始执行该段语句，最终导致在执行过程中，解释器因为找不到与 `BEGIN` 匹配的 `END` 而报错。为了避免发生此类错误，可以使用 `DELIMITER` 命令来修改语句结束符。

`DELIMITER` 命令示例如下：

```
DELIMITER new_delemiter
```

其中，`new_delemiter` 可以设置为 1 个或多个字节长度的符号，默认为分号（;），您可以将其修改为其他符号，例如 `#` 。

添加 `DELIMITER` 命令后，`DELIMITER` 命令后的语句再使用分号则不会报错，直到遇到设置的结束符（`#`），才认为该语句结束。

> **注意**
>
> 使用 `DELIMITER` 命令修改结束符后，待语句执行结束，请务必将结束符修改为默认符号分号（;）。

示例如下：

```
obclient> CREATE TABLE test (user_id INT, user_num DECIMAL(10,2));
Query OK, 0 rows affected

obclient> DELIMITER #

obclient> CREATE TRIGGER test_trg BEFORE UPDATE ON test
               FOR EACH ROW
               BEGIN
               IF NEW.user_num < 1 THEN
              SET NEW.user_num  = 1;
              ELSEIF NEW.user_num > 45 THEN
              SET NEW.user_num= 45;
             END IF;
             END;#
Query OK, 0 rows affected

obclient> DELIMITER ;
```

触发器的使用限制

MySQL 模式的触发器有以下使用限制：

- 对于同一个表，不能有多个触发器具备相同的触发事件和触发动作的时间。
- 只能在永久表上创建触发器，不能对临时表创建触发器。
- 触发器不能使用 `CALL` 语句调用将数据返回给客户端或使用动态 SQL 的存储过程，但是允许存储过程或者函数通过 `OUT` 或者 `IN OUT` 类型的参数将数据返回触发器。

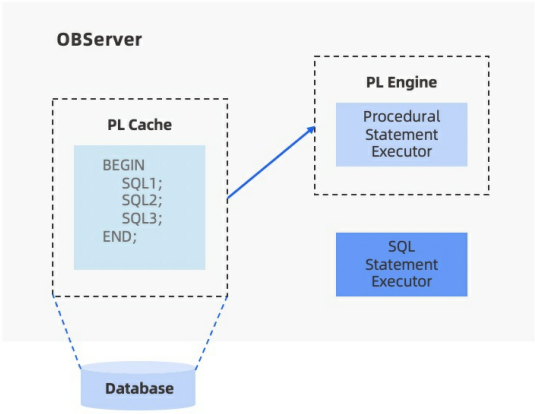
- 触发器中不能使用开启或结束事务的语句段。例如，开始事务（ `START TRANSACTION` ）、提交事务（ `COMMIT` ）或回滚事务（ `ROLLBACK` ），但是可以回滚到一个保存点，即 Savepoint 是允许的，因为回滚到保存点不会结束事务。
- 外键不会激活触发器。
- 触发器中不允许返回值，因此触发器中不能有返回语句，如果需要立即停止一个触发器，则需要使用 `LEAVE` 语句。

6.2.2. PL 执行机制

OceanBase 数据库的 PL 代码仅支持编译执行。本文主要介绍 PL 引擎与 OBServer 和 SQL 引擎间的执行机制。

编译执行相对于解释执行有更好的性能优势。尤其是在 PL 引擎内部计算元素比较多的场景下，性能提升会更加明显。

在编译执行的框架下，PL 源代码会被直接编译为对应平台的可执行代码。PL 引擎执行时，OBServer 直接调用编译好的可执行代码。同时编译好的 PL 代码会被缓存在 PL 缓存中，因此同一个 PL 代码仅需要编译一次。



PL 代码在定义时被存储在 OBServer 中，当应用程序调用存储过程时，OBServer 首先从数据字典中读取 PL 代码的定义，然后将 PL 代码定义编译为可执行的二进制代码，将编译好的 PL 二进制代码存储进 PL 缓存。执行时 PL 引擎从 PL 缓存获取可执行代码，配合 SQL 引擎完成 PL 代码的执行。

存储过程间可以嵌套调用，例如当前 PL 块可以调用另外的存储过程。

6.3. 客户端编程语言

ECOB（Embedded SQL C Preprocessor for OceanBase）是 OceanBase 数据库架构环境中，与 Oracle Pro*C 兼容的 OceanBase 预编译器，提供了与 Proc*C 兼容的完整功能特性。

Pro*C 是 Oracle 数据库生态提供的一种开发应用的工具。当使用 C 语言编写应用程序时，可以在程序源代码中直接嵌入 SQL 语句，这些嵌入式 SQL 语句可以使用宿主 C 程序中的 C 语言变量作为输入输出。然后，Pro*C 预编译程序 proc 将对源代码进行预处理以进行完整的语法分析，并把嵌入式 SQL 语句和指令转换为对运行时库 sqllib 的函数调用。最后，输出一个 C 语言程序源代码文件。这个文件再通过 C 语言编译器进行编译链接后即可生成可执行程序。因此一个 proc 项目会由大量 `.pc` 源文件组成，并通过 Makefile 等方式构建，OceanBase 数据库为了提供对 Pro*C 的支持，开发了类似的包含预编译程序 ecob 和运行时库 ecoblib 的预编译器 ECOB，同时为了迁移代价最小化，我们的预编译程序 ecob 还提供了和 proc 程序相同的命令行选项和功能。

相关概念

- **.pc**：对于含有嵌入式 SQL 的 C 语言程序文件，其文件后缀名为 `.pc` 。
- **ECOB**：OceanBase 数据库提供的一款用于编译含有嵌入式 SQL 的 C 语言程序的预编译器。
- **ecoblib**：OceanBase 数据库提供的用于 ECOB 的运行时库，该库提供 ECOB 需要的各种函数接口。
- **Pro*C**：Oracle 提供的一款用于编译含有嵌入式 SQL 的 C 语言程序的预编译器。
- **sqllib**：Oracle 提供的用于 Pro*C 的运行时库，该库提供 Pro*C 需要的各种函数接口。

功能特性

- 与 OceanBase 数据库 Oracle 模式完全兼容的 SQL 语法支持。
- 支持基本嵌入式 SQL 语句，例如 `COMMIT` 、 `CONNECT` 、 `DELETE` 、 `EXECUTE` 、 `EXECUTE IMMEDIATE` 、 `INSERT` 、 `SELECT` 、 `UPDATE` 、 `WHENEVER` 、 `CALL` 、 `PREPARE` 和 `ROLLBACK` 等语句。
- 支持游标（Cursor）相关的 `DECLARE` 、 `OPEN` 、 `CLOSE` 和 `FETCH` （包括 `NEXT` 、 `ABSOLUTE` 、 `WITH HOLD` 、 `CURRENT OF` ）等语句。
- 支持 ANSI 标准的动态 SQL 所需要的 `DESCRIPTOR` 和与之相关的 `ALLOCATE` 、 `DEALLOCATE` 、 `GET` 、 `SET` 、 `DESCRIBE` 和 `FETCH` 等语句。
- 宿主变量无需在 `BEGIN DECLARE SECTION` 和 `END DECLARE SECTION` 内声明即可直接使用。
- 支持对 C 预处理器宏 #ifdef、#ifndef、#else、#endif 和 #define 的识别和处理。
- ecoblib 库支持全部上述语句的运行时行为，不仅提供了 proc 程序兼容行为（如 CHAR_MAP）还支持程序在 Tuxedo 环境下的运行。
- ecob 程序识别全部 proc 程序选项和兼容语义（例如 PARSE=FULL），可无缝替换 proc 程序使用。

数据类型与变量

ECOB 支持了 Oracle 内部数据类型和 C 语言的外部数据类型，同时支持它们之间的数据类型转换。并且 ECOB 当前支持指示器、VARCHAR、结构体和字符串指针等变量。

内部数据类型

ECOB 中内部类型指的是在 OceanBase 数据库中可使用的数据类型，当前 ECOB 支持的内部数据类型如下表所示。

内部数据类型	说明
CHAR	固定长度字符串。其最大长度为 2000。
VARCHAR2	可变长度字符串。其最大长度为 32767。
NUMBER	数值类型。精度取值范围为 [1,38]，位数取值范围为 [-84,127]。
INT	整型数值，其最大值为 38 位。
BINARY_FLOAT	32 位浮点数。最小值为 1.17549E-38F，最大值为 3.40282E+38F。
BINARY_DOUBLE	64 位浮点数。最小值为 2.22507485850720E-308，最大值为 1.79769313486231E+308。
DATE	日期类型。格式为 `YYYY-MM-DD HH:MI:SS` 。
TIMESTAMP	时间类型。格式为 `YYYY-MM-DD HH:MI:SS [.FFFFFFFFF]` 。

外部数据类型

ECOB 中外部类型被用于指定宿主变量存储数据的类型（C 语言中的数据类型），当输入数据到数据库时，ECOB 会将输入的宿主变量的外部类型和内部数据类型进行转换。当输出数据到外部程序时，ECOB 会将数据库中表的内部数据类型和输出的宿主外部数据类型进行转换。

下述表格展示了 ECOB 支持的外部类型，并展示了和宿主变量数据类型之间的对应关系。

外部数据类型	说明	宿主变量数据类型
ECOBt_char	用于存储定长字符串。	char、char *、char []
ECOBt_unsigned_char	用于存储定长字符串。	unsigned char
ECOBt_short	用于存储有符号短整型。	int、int *、 int[]
ECOBt_unsigned_short	用于存储无符号短整型。	unsigned int

外部数据类型	说明	宿主变量数据类型
ECOBt_int	用于存储有符号整型。	short 、short * 、short[]
ECOBt_unsigned_int	用于存储无符号整型。	unsigned short
ECOBt_long	用于存储有符号长整型。	long、long * 、long[]
ECOBt_unsigned_long	用于存储无符号长整型。	unsigned long
ECOBt_long_long	用于存储有符号长长整型。	long long、long long * 、long long[]
ECOBt_unsigned_long_long	用于存储无符号长长整型。	unsigned long long
ECOBt_float	用于存储 32 位浮点数。	float 、float * 、float[]
ECOBt_double	用于存储 64 位浮点数。	double、double *、double[]
ECOBt_varchar	用于存储变长字符串。	varchar varchar * varchar[]
ECOBt_struct	用于存储结构体。	struct 、struct *、struct[]

数据类型转换

在 ecob 程序中，宿主变量（Host Variables）使用了外部数据类型，当从数据库读取数据到 C 语言变量或者将 C 语言数据写入到数据库中时，会发生外部数据类型与内部数据类型的转换。

下表展示了数据类型间的转换关系，其中 **IN** 表示支持从 C 语言转成数据库类型（写入数据到数据库），**OUT** 表示支持数据库类型转成 C 语言（读取数据）。

	varchar2/char	int	NUMBER	FLOAT	BINARY_FLOAT	BINARY_DOUBLE	Date	TIMESTAMP
char/unsigned char	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	-	-

	varchar2/char	int	NUMBER	FLOAT	BINARY_FLOAT	BINARY_DOUBLE	Date	TIMESTAMP
int/unsigned int	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	-	-
long/unsigned long	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	-	-
long long/unsigned long long	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	-	-
float	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	-	-
double	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	-	-
char[n]/varchar[n]/char*	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN/OUT	IN

指示器变量

指示器变量是用来处理数据库 `NULL` 值的变量。当执行 `SELECT` 或者 `FETCH` 语句时，如果不使用指示器变量并且列返回的值为 `NULL` 时，会显示错误信息。指示器变量必须采用 short 类型定义，并且指示器变量必须跟在宿主变量后面。

指示器变量的语法如下所示：

```
:host_variable [INDICATOR] :indicator_variable
```

在使用指示器变量后，根据以下规则可以检测出返回的列是否为 `NULL`：

- 当指示器变量返回 -1 时，表示数据库返回 `NULL` 值。
- 当指示器变量返回 0 时，表示列值被赋给了输出宿主变量。
- 当指示器变量返回的值大于 0 时，表示将被截断列值赋给了输出宿主变量，并且指示器变量存放着数据列值的实际长度。

- 当指示器变量返回值为 -2 时，表示将被截断列值赋给了输出宿主变量，但是实际长度不能确定。

VARCHAR 变量

`VARCHAR` 变量是使用 VARCHAR 类型定义的变长字符串。`VARCHAR` 类型只能用于定义宿主变量，而不能用于定义普通 C 变量，所以定义 `VARCHAR` 变量时，必须要指定长度，示例语句如下所示：

```
VARCHAR name[20];
```

上面的 VARCHAR 变量预编译后会生成一个如下所示的 C 结构体：

```
struct{
 unsigned short len;
 unsigned char arr[20];
}name;
```

结构体中 `arr` 对应字符串，`len` 表示字符串的实际长度。

结构体与字符串指针

在 ecob 程序中使用结构体与字符串指针作为宿主变量（Host Variables）时，有几个地方需要注意下：

- 结构体（Struct）作为宿主变量时，可以作为一个整体传递。预编译时会自动将结构体拆分到每一个数据库列中。
- 当指针作为结果输出时，需要初始化非空。示例语句如下：

```
char * p1 ;
char * p2;
p1=(char *) malloc(11);
p2=(char*) malloc(11);
strcpy(p1,"     ");
strcpy(p2,"00000");
EXEC SQL SELECT c1,c2 into :p1,:p2 from t1 where rownum < 2;
```

嵌入式 SQL

ECOB 支持的嵌入式 SQL 语句包含全部的标准 SQL 语句，并包含若干扩展 SQL 语句用于在宿主程序与 OBServer 服务之间传递数据。在 ECOB 中使用嵌入式 SQL 语句的语法如下所示，需在语句前添加 EXEC SQL。

```
EXEC SQL <standard SQL statement>|<extension SQL statement>
```

> **说明**
>
> 本章节主要介绍 ECOB 支持的扩展 SQL 语句与 ECOB 特有的 SQL 语句，有关标准 SQL 语句的语法可参见《OceanBase 数据库 SQL 参考（Oracle 模式）》手册。

变量声明

在 ECOB 中，DECLARE SECTION 语句用于声明宿主变量，这是 ECOB 特有的 SQL 语句。语法如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
...
EXEC SQL END DECLARE SECTION;
```

示例语句：

```
EXEC SQL BEGIN DECLARE SECTION;
int a;
char * b;
EXEC SQL END DECLARE SECTION;
```

在声明块中只允许使用如下元素：

- 宿主变量或指示器变量
- C 语言的注释
- EXEC SQL INCLUDE 语句
- 关键字 typedef

需要注意的是，当选项 parse=full（缺省值为 full）时，宿主变量在大部分情况下都不需要在 DECLARE SECTION 语句中声明，但当结构体中有 `VARCHAR` 变量时需要在声明块中声明。当选项 parse=NONE 或 parse=partial 时，宿主变量需要在 DECLARE SECTION 语句中声明。

CONNECT 数据库

ECOB 目前支持的扩展 SQL 语句中， `CONNECT` 语句用于与 OceanBase 数据库建立连接连接。语法如下所示：

```
语法一：EXEC SQL CONNECT <:username> identified by <:password> (using <:dbstring>)
语法二：EXEC SQL CONNECT <:user_password> (using <:dbstring>)
语法三：EXEC SQL CONNECT <:username> identified by <:password> (AT <:dbname>) (using <:dbstr
ing>)
```

其中各变量的含义如下所示：

- **<:username>：**

  用于连接 OceanBase 数据库 Oracle 模式租户的用户名，格式为**用户名@租户名**，例如 `test@oracle` 。如果使用 OBProxy 连接，需要加上集群名，格式为**用户名@租户名#集群名**，例如 `test@oracle#cluster1` 。

- **<:password>：**

  用于连接 OceanBase 数据库 Oracle 模式租户的密码。

- **<:dbstring>：**

  用于连接 OceanBase 数据库 Oracle 模式租户的服务名。此服务名是 `TNS_ADMIN` 环境变量中指定的 `tnsnames.ora` 文件连接串的名字。假如 `tnsnames.ora` 文件中的连接串如下所示：

```
demo=
      (DESCRIPTION=
        (ADDRESS=(PROTOCOL = TCP)(HOST = 10.10.10.10)(PORT = 30035))
        (CONNECT_DATA=
          (SERVICE_NAME=TEST))
)
```

那么 `<:dbstring>` 的值即为 `demo` 。如使用 ip+port 方式的连接串， `<:dbstring>` 的值也可以写成 'ip:port/dbname' ，如 '10.10.10.10:30035/test' 。

- **<:user_password>:**

  用于连接 OceanBase 数据库 Oracle 模式租户的用户名和密码，例如 test@oracle/welcome1 ，其中用户名和密码中间用反斜杠（/）进行分隔。

  示例语句如下：

```
//connect method 1
char * username ="demo@oracle";
char * password = "welcome1";
char * servicename = "obdb";
EXEC SQL CONNECT :username identified by :password using :servicename;

//connect method 2
char * userpass = "demo@oracle/welcome1";
char * servicename = "obdb";
EXEC SQL CONNECT :userpass using :servicename;

//connect method 3
char * username ="demo@oracle";
char * password = "welcome1";
char * servicename = "obdb";
char * connname = "democonn";
EXEC SQL CONNECT :username identified by :password AT :connname using :servicename;
```

基础 SQL 语句

ECOB 目前支持的扩展 SQL 语句中，基础 SQL 语句有 `SELECT` 、 `INSERT` 、 `UPDATE` 、 `DELETE` 、 `COMMIT` 和 `ROLLBACK` 语句。

- `SELECT` 语句用于执行查询语句并输出查询结果到外部宿主变量。
- `INSERT` 语句用于执行插入语句，可以将外部宿主变量输入到数据库的表列中。
- `UPDATE` 语句用于执行更新语句，可以将外部数据变量的值更新到数据库的表列中，也可以选择只更新当前游标列。
- `DELETE` 语句用于执行删除语句，可以将数据库中的一行或者多行删除，也可以选择只删除当前游标列。
- `COMMIT` 语句用于提交一个事务语句，可以选择是否释放资源并关闭数据库连接。

- `ROLLBACK` 语句用于回滚一个事务语句，可以选择是否释放资源并关闭数据库连接。

预编译语句

ECOB 目前支持的扩展 SQL 语句中，与预编译语句（Prepared Statement）相关的语句有 `PREPARE` 语句和 `EXECUTE` 语句。它们是一种动态 SQL 语句，可以包含固定个数的输入输出宿主变量。

- `PREPARE` 语句用于 PS 模式下预编译一个 SQL 语句。
- `EXECUTE` 语句用于执行编译好的 SQL 语句。

存储过程

ECOB 目前支持的扩展 SQL 语句中，`CALL` 语句用于执行一个存储过程。在当前版本只支持没有参数 `OUT` 的存储过程。

游标

ECOB 目前支持的扩展 SQL 语句中，与游标（Cursor）相关的语句有 `DECLARE CURSOR` 、 `OPEN` 、 `FETCH` 和 `CLOSE` 等语句。

- `DECLARE CURSOR` 语句用于定义一个游标变量，当前游标变量只支持查询语句。
- `OPEN` 语句用于打开一个游标变量。
- `FETCH` 语句用于获取游标变量保存的结果集，对于滚动游标，可以指定 `FETCH` 的位置。
- `CLOSE` 语句用于关闭一个游标变量。

简单动态 SQL

ECOB 目前支持的扩展 SQL 语句中，`EXECUTE IMMEDIATE` 语句用于执行动态的 SQL 字符串。这种语句中，SQL 语句可以在运行时生成，但不能使用宿主变量。

ANSI 动态 SQL

ECOB 目前支持的扩展 SQL 语句中，包含了兼容 ANSI 标准的动态 SQL 语句。目前支持 `ALLOCATE DESCRIPTOR` 、 `DEALLOCATE DESCRIPTOR` 、 `DESCRIBE INPUT DESCRIPTOR` 、 `DESCRIBE OUTPUT DESCRIPTOR` 、 `GET DESCRIPTOR` 、 `SET DESCRIPTOR` 、 `OPEN USING DESCRIPTOR` 和 `FETCH INTO DESCRIPTOR` 等语句。当 SQL 语句的输入或输出宿主变量个数在预编译时未知的时候，可以使用 ANSI 动态 SQL 语句。

- `ALLOCATE DESCRIPTO` R 语句用于分配 SQL 描述区。
- `DEALLOCATE DESCRIPTOR` 语句用于释放SQL描述区。
- `DESCRIBE INPUT DESCRIPTOR` 语句用于绑定变量信息。

- `DESCRIBE OUTPUT DESCRIPTOR` 语句用于获取输出列的信息。
- `GET DESCRIPTOR` 语句用于获取 SQL 描述区的 Item 信息。
- `SET DESCRIPTOR` 语句用于设置 SQL 描述区的 Item 信息。
- `OPEN USING DESCRIPTOR` 语句用于在 ANSI 动态 SQL 中打开游标变量。
- `FETCH INTO DESCRIPTOR` 语句使用动态语句描述符获取游标变量保存的结果集。

错误处理

SQLCA 结构

目前 ECOB 实现了 The SQL Communications Area（SQLCA）数据结构，目前的实现中不需要显式使用 #include<sqlca.h> 或 EXEC SQL INCLUDE SQLCA 语句。SQLCA 结构体有 sqlcode、sqlerrm 和 sqlerrd 等成员。

sqlcode 的取值范围如下所示：

- 值等于 0 时，表示成功。
- 值小于 0 时，表示语句在数据库中执行失败，其具体错误码可能和 Oracle 有差别，在应用程序中希望直接进行 `sqlca.sqlcode<0` 的判断。
- 值大于 0 时，表示在数据库中未找到合适的数据，目前大于 0 的值只支持 1403，对应信息 `DATA NOT FOUND` 。

sqlerrm 结构体有两个成员 sqlerrml 和 sqlerrmc，其中 sqlerrml 用于存放错误信息的长度，sqlerrmc 用于保存报错信息。

sqlerrd 是一个 long 数组，长度为 6，下标为 0、1、3、4 和 5 的元素值为空，下标为 2 的元素保存的是 `INSERT` 、 `UPDATE` 和 `DELETE` 等语句处理的行数。

其中提供了 sqlgls 函数用于获取上一次执行的 SQL 语句和 sqlglm 获取上一次执行的错误信息。

其中 `sqlgls` 返回的是 SQL 语句， `stmlen` 返回的是 SQL 语句的长度， `sqlfc` 返回的是 SQL FUNCTION code； `sqlglm` 返回的是错误信息， `buffer_size` 是 buffer 的最大长度， `message_length` 是错误信息的实际长度。

WHENEVER 语句

`WHENEVER` 语句用于指定错误和警告条件的处理方法。

语法如下所示：

```
EXEC SQL WHENEVER (SQLERROR | NOT FOUND) ( DO (routine | BREAK | CONTINUE) | CONTINUE | GO TO <label> | STOP )
```

其中 `SQLERROR` 表示当当前执行语句发生错误时采取语句中定义 Action 来做错误处理，而 `NOT FOUND` 表示当前执行语句未找到数据时采取语句中定义的 Action 进行异常处理。

支持的 Action 如下：

- `CONTINUE`：继续执行，也是默认行为，相当于没处理任何异常。
- `DO`：执行错误处理函数。
- `DO BREAK`：相当于在程序中增加了 BREAK 语句，用在 LOOP 循环中。
- `DO CONTINUE`：相当于在程序中增加了 CONTINUE 语句，用在 LOOP 循环中。
- `GOTO label_name`：跳转到程序的 `label_name` 处。
- `STOP`： 程序终止，为提交的事务都会回滚，类似执行了 `EXIT()` 函数。

需要注意的是， `WHENEVER` 语句会对语句之后所有的嵌入语句生效，特别是对于 Action DO BREAK 和 DO CONTINUE，这两个 Action 要求在循环体内，否则报错，所以对于在 Action DO BREAK 或 DO CONTINUE 后不在循环体内的 DML 语句，需要在语句之前重置异常处理的方式。

```
{
EXEC SQL WHENEVER SQLERROR DO sqlerror();
EXEC SQL INSERT INTO t1 VALUES(1,'ABC');
...
}
void sqlerror(){
...
}
```

又如

```
int c1val;
EXEC SQL DECLARE cur CURSOR FOR select c1 from t1;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
for(;;){
EXEC SQL FETCH cur INTO :c1val;
}
```

7.事务管理

7.1. 事务

7.1.1. 事务简介

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。

数据库事务有两个作用：

- 为数据库操作序列提供一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。
- 为数据库的多个并发访问提供隔离的方法，避免多个并发操作导致数据库进入一个不一致的状态。

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。

- 原子性

  OceanBase 数据库是一个分布式系统，分布式事务操作的表或者分区可能分布在不同机器上，OceanBase 数据库采用两阶段提交协议保证事务的原子性，确保多台机器上的事务要么都提交成功要么都回滚。

- 一致性

  事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

- 隔离性

  OceanBase 数据库支持 Oracle 和 MySQL 兼容模式。在 Oracle 模式下，支持 Read Committed 隔离级别和 Serializable 隔离级别。在 MySQL 模式下，支持 Read Committed 隔离级别和 Repeatable Read 隔离级别。

- 持久化
  对于单个机器来说，OceanBase 数据库通过 redo log 记录了数据的修改，通过 WAL 机制保证在宕机重启之后能够恢复出来。保证事务一旦提交成功，事务数据一定不会丢失。对于整个集群来说，OceanBase 数据库通过 paxos 协议将数据同步到多个副本，只要多数派副本存活事务数据一定不会丢失。

7.1.2. 事务的结构

一个数据库事务包含一条或者多条 DML 语句，事务有明确的起始点及结束点。

开启事务

以 MySQL 模式为例，数据库在执行以下语句时会开启一个事务。

```
begin
start transaction
insert …
update …
delete …
select … for update...
```

当事务开启时，OceanBase 数据库为事务分配一个事务 ID，用于唯一的标识一个事务。通过 `oceanbase.__all_virtual_trans_stat` 可以查询事务的状态。以下例子说明，`update` 语句开启了一个事务。

```
session 1:
obclient> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)

obclient> UPDATE t SET c="b" WHERE i=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

session 2:
obclient> SELECT trans_id FROM __all_virtual_trans_stat;
+-----------------------------------------------------------------------------------------+
| trans_id                                                                                |
+-----------------------------------------------------------------------------------------+
| {hash:17242042390259891950, inc:98713, addr:"100.88.106.107:24974", t:1632636623536459} |
+-----------------------------------------------------------------------------------------+
```

语句执行

在语句执行过程中，OceanBase 数据库在语句访问到的每个分区上创建一个事务上下文，用于记录语句执行过程中的数据快照版本号以及语句对该分区所做的修改。

结束事务

事务结束时收集事务执行过程中修改过的所有分区，根据不同的场景对这些分区发起提交事务或者回滚事务。以下场景会触发事务的提交或者回滚。

- 用户显式的发起 Commit 或者 Rollback。用户发起 Commit 时，OceanBase 数据库会将事务所做的修改持久化到 clog 文件中。
- 用户执行 DDL 操作。包括 CREATE、DROP、RENAME、或者 ALTER。当用户在事务中发起这些 DDL 操作，OceanBase 数据库会隐式的发起一个 Commit 请求，后续的语句会开启一个新的事务。
- 客户端断开连接。当客户端在事务执行过程中断开连接，OceanBase 数据库会隐式的发起 Rollback 请求，将事务回滚。

一个事务结束之后，后续的请求会开启一个新的事务。以下例子说明，`UPDATE` 语句开启了一个事务，`ROLLBACK` 语句结束事务。后续的 `UPDATE` 语句又开启了一个新的事务。

```
session 1:
obclient> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)

obclient> UPDATE t SET c="b" WHERE i=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

session 2:
obclient> SELECT trans_id FROM __all_virtual_trans_stat;
+-------------------------------------------------------------------------------------------+
| trans_id                                                                                  |
+-------------------------------------------------------------------------------------------+
| {hash:17242042390259891950, inc:98713, addr:"100.88.106.107:24974", t:1632636623536459} |
+-------------------------------------------------------------------------------------------+

session 1:
obclient> ROLLBACK;
Query OK, 0 rows affected

obclient> UPDATE t SET c="c" WHERE i=1;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

session 2:
obclient> SELECT trans_id FROM __all_virtual_trans_stat;
+-------------------------------------------------------------------------------------------
+
| trans_id
|
+-------------------------------------------------------------------------------------------
+
| {hash:11752179762787656100, inc:104885, addr:"100.88.106.107:24974", t:1632636737417119}
|
+-------------------------------------------------------------------------------------------
+
1 row in set
```

7.1.3. 语句级原子性

OceanBase 数据库支持语句级的原子性，语句级的原子性指的是一条语句的操作要么都成功要么都失败，不会存在部分成功部分失败的情况。

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。语句级回滚有如下特点：

- 语句回滚时仅回滚本语句的修改，不会影响当前事务该语句之前语句所做的修改。

  例如，一个事务有 2 条 `UPDATE` 语句，第一条 `UPDATE` 语句执行成功，第二条 `UPDATE` 语句执行失败，则只有第二条 `UPDATE` 语句会发生语句级回滚，第一条 `UPDATE` 语句所做的修改会被保留。

- 语句级回滚的效果等价于语句没有被执行过，即语句执行过程中涉及的全局索引、触发器、行锁等均属于语句的操作，语句回滚需要将这些操作都回滚到语句开启之前的状态。

常见的语句级回滚主要有：

- Insert 操作出现主键冲突会导致语句级回滚。
- 单条语句执行时间过长，语句执行超时可能出现语句级回滚。
- 多个事务存在行锁冲突导致死锁，某个事务被死锁检测机制 Kill 掉可能会出现语句级回滚。

SQL 语句的语法解析报错不涉及到语句级回滚，因为语句解析报错尚未对数据进行修改。

7.1.4. 全局时间戳

全局时间戳服务（Global Timestamp Service，简称 GTS），OceanBase 数据库内部每个租户启动一个全局时间戳服务，事务提交时通过本租户的时间戳服务获取事务版本号，保证全局的事务顺序。

服务高可用

GTS 是集群的核心，需要保证高可用。

- 对于用户租户而言，OceanBase 数据库使用租户级别内部表 __all_dummy 的 leader 作为 GTS 服务提供者，时间来源于该 leader 的本地时钟。GTS 默认是三副本的，其高可用能力跟普通表的能力一样。
- 对于系统租户，使用 `__all_core_table` 的 leader 作为 GTS 服务的提供者，高可用能力与普通表一样。

时间戳正确性保证

GTS 维护了全局递增的时间戳服务，异常场景下依然能够保证正确性：

- **有主改选**

  原 Leader 主动发起改选的场景，我们称为有主改选。新 leader 上任之前先获取旧 leader 的最大已经授权的时间戳作为新 leader 时间戳授权的基准值。因此该场景下，GTS 提供的时间戳不会回退。

- **无主选举**

  原 leader 与多数派成员发生网络隔离，等 lease 过期之后，原 follower 会重新选主，这一个过程，我们称为无主选举。选举服务保证了无主选举场景下，新旧 Leader 的 lease 是不重叠的，因此能够保证本地时钟一定大于旧主提供的最大时间戳。因此新 leader 能够保证 GTS 提供的时间戳不回退。

GTS 获取优化

- 语句快照获取优化事务提交的时候都会更新其所在机器的 Global Committed Version，当一条语句可以明确其查询所在机器时，如果是一台机器，则直接使用该机器的 Global Committed Version 作为 Read Version，降低对于全局时间戳的请求压力。
- 事务提交版本号获取优化多个事务可以合并获取全局时间戳，并且获取时间戳的请求可以提早发送，缩短事务提交时间。

7.1.5. 事务控制

7.1.5.1. 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。更多 Savepoint 相关信息请参见 Savepoint。

事务大小

OceanBase 数据库 V2.x 版本上单个事务大小有限制，通常是 100M。事务的大小与两个配置项有关，分别是租户级配置项 `_tenant_max_trx_size` 和集群级配置项 `_max_trx_size`，`_tenant_max_trx_size` 优先生效。

OceanBase 数据库 V3.x 版本因支持了大事务，不再受此限制。

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。

事务查询

虚拟表 `__all_virtual_trans_stat` 可用于查询系统中当前所有的活跃事务。有关活跃事务的详细信息请参见 活跃事务。

7.1.5.2. 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。虚拟表 __all_virtual_trans_stat 里的 state 字段标识了事务所处的状态。

`state` 字段对应的值所表示的含义如下表所示。

state 的值	说明
0	表示事务处于活跃状态，所有修改对其他事务不可见。

state 的值	说明
1	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）。
2	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）。
3	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。
4	表示事务已经提交或回滚结束，处于 `CLEAR` 状态。
101	表示单分区事务提交完成，处于 `COMMIT` 状态，其他事务可以看到该事务的修改。
102	表示单分区事务已经回滚，其他事务看不到该事务的修改。

活跃事务的数据存储相关信息，请参见多版本读一致性。

7.1.5.3. 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。

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

命令	解释
`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。

7.1.5.4. 事务控制语句

7.1.5.4.1. MySQL 事务控制

开启事务

OceanBase 数据库的事务控制语句与 MySQL 数据库兼容，开启事务可以通过以下方式来完成：

- 执行 `START TRANSACTION` 命令
- 执行 `BEGIN` 命令
- 执行 `SET autocommit = 0` 之后再执行的第一条语句

语法如下：

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

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]

SET autocommit = {0 | 1}
```

提交事务

提交事务通过 `COMMIT` 命令来完成，具体语法如下所示：

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

此外，如果 `autocommit = 0` ，那么执行一条开启事务的语句也会隐式地提交当前进行中的事务。

回滚事务

回滚事务通过 `ROLLBACK` 命令来完成。

回滚事务的 SQL 语法如下：

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

自动提交

自动提交是指当 `autommit` 这个 Session 变量的值为 `1` 时，此时每条语句执行结束后，OceanBase 数据库将会自动的把这条所在的事务提交。这样一来，一条语句就是一个事务。

隐式提交

隐式提交是用户未发出 COMMIT/ROLLBACK 等结束事务的语句给 OceanBase 数据库，而 OceanBase 数据库自动将当前活跃的事务执行 COMMIT 提交的过程。

隐式提交发生在以下情形：

- 执行一个开启事务的语句
- 执行 DDL

自动回滚

自动回滚是用户未发出 ROLLBACK 指令，而是 OceanBase 数据库内部发起的回滚，通常发生在以下情形：

- session 断开
- 事务执行超时（ob_tx_timeout）
- 活跃事务的 session 超过一定时长没有语句执行（ob_tx_idle_timeout）

> **说明**
>
> - ob_tx_timeout 用于设置事务超时时间，单位为微秒。
> - ob_tx_idle_timeout 用于设置事务空闲超时时间，即事务中两条语句之间的执行间隔超过该值时超时，单位为微秒。
>
> 有关变量的详细介绍，参见《OceanBase 数据库参考指南》中 **系统变量** 章节。

这些情况下，事务自动被 OceanBase 数据库回滚，如果用户再次在当前 session（未断开）上执行 SQL 则会提示事务已经中断（无法继续）需要回滚，此时用户需要执行 ROLLBACK 来结束当前事务。

事务被中断

当事务执行过程中发生内部错误，如参与者节点宕机或者其它导致事务无法继续时，此时当前事务无法继续成功地执行语句，只能回滚。

此种情况发生时，用户执行 SQL 语句将会收到“transaction need rollback”的错误，用户需要执行 ROLLBACK 来结束当前事务。

7.1.5.4.2. Oracle 事务控制

开启事务

开启方式

OceanBase 数据库在 Oracle 模式下开启事务共有以下两种方式：

- 自动开启

  在无事务开启的 Session 端执行 DML 语句（除了 `SELECT` ）、DCL 语句或 DDL 语句时，会默认开启一个事务。

- 手动开启

  除了兼容 Oracle 数据库的使用方式外，OceanBase 数据库还支持手动开启事务，语法如下：

```
START TRANSACTION
    [READ ONLY | READ WRITE];
BEGIN [WORK] ;
```

执行效果

事务开启后，系统会根据用户指定或默认设置对事务信息进行初始化：

1. 设置用户信息。
2. 设置隔离级别。

   默认事务的隔离级别为 `READ COMMITTED` ，您可以通过 `SET TRANSACTION` 等语句指定隔离级别。

3. 为 `SERIALIZABLE` 级别的事务设置事务级快照版本。

提交事务

提交当前事务的语法如下：

```
obclient> COMMIT [WORK];
```

输入 `COMMIT` 语句后即进入事务提交流程，提交事务修改并结束事务。

在 OceanBase 数据库中，典型情况下提交事务需要执行“两阶段提交”，两阶段提交详细说明请参见 二阶段提交。

回滚事务

回滚当前事务的语法如下：

```
ROLLBACK [WORK];
```

输入 ROLLBACK 语句后进入事务回滚流程，清除事务修改并结束事务。

具体流程如下：

1. 各参与者清除 MemTable 上自己的修改并释放行锁，清理参与者上下文。
2. 参与者清除成功后返回消息给调度者，调度者收集到所有参与者清理成功的消息后，清理调度者上下文。

只读事务

只读不开事务

为了针对只读语句进行优化，在 Session 无事务开启的情况下，执行 `SELECT` 语句即可进入只读不开事务路径。只读不开事务跳过事务上下文等流程，只读取所需的必要信息，将数据返回用户。

只读不开事务的快照时间获取方式由当前 Session 的隔离级别决定。

只读不开事务的示例如下：

```
SELECT * FROM t1;//只读不开事务
SELECT * FROM t2;//只读不开事务
INSERT INTO t1 VALUES(1);//开启事务
```

只读事务

输入如下语句，可开启只读事务。

```
obclient> START TRANSACTION READ ONLY;
```

只读事务不能存在对数据的修改，也因此相对读写事务可进行优化，只读事务不创建事务上下文，故在回滚时也无需执行任何清理操作。

7.1.6. Redo 日志

本节主要介绍 Redo 日志保证数据持久化的机制及 Redo 日志的归档方式。

概述

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。

OceanBase 数据库的 Redo 日志有两个主要作用：

- 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 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。

- 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。

日志文件类型

OceanBase 数据库采用了分区级别的日志流，每个分区的所有日志要求在逻辑上连续有序。而一台机器上的所有日志流最终会写入到一个日志文件中。

OceanBase 数据库的 Redo 日志文件包含如下两种类型：

- Clog

  全称 Commit log，用于记录 Redo 日志的日志内容，位于 `store/clog` 目录下，文件编号从 1 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。

- ilog

  全称 index log，用于记录相同分区相同 Log ID 的已经形成多数派日志的 Commit log 的位置信息。位于 `store/ilog` 目录下，文件编号从 1 开始并连续递增，文件 ID 不会复用，单个日志文件的大小非定长。

  这个目录下的日志文件是 Clog 的索引，本质上是对日志管理的一种优化，ilog 文件删除不会影响数据持久性，但可能会影响系统的恢复时间。 ilog 文件和 Clog 文件没有对应关系，由于 ilog 针对单条日志记录的内容会比 Clog 少很多，因此一般场景下 ilog 文件数目也比 Clog 文件数目少很多。

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。

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。

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。

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_disk_usage_limit_percentage`

  该配置项用于控制 Clog 或 ilog 磁盘空间的使用上限，默认值为 `95`，表示允许 Clog 或 ilog 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，超过此值后该 OBServer 不再允许任何新事务的写入，同时不允许接收其他 OBServer 同步的日志。对外表现是所有访问此 OBServer 的读写事务报 `"transaction needs rollback"` 的错误。

- `clog_disk_utilization_threshold`

  该配置项用于控制 Clog 或 ilog 磁盘的复用下限。在系统工作正常时，Clog 或 ilog 会在此水位开始复用最老的日志文件，默认值是 Clog 或 ilog 独立磁盘空间的 80%，不可修改。因此，正常运行的情况下，Clog 或 ilog 磁盘空间占用不会超过 80%，超过则会报 `"clog disk is almost full"` 的错误，提醒 DBA 处理。

7.1.7. 本地事务

本地事务是相对于跨机分布式事务而言的，特指事务所操作的表的分区 Leader 全部在同一个 Server 上，并且与 Session 建立的 Server 具有相同的事务。

根据操作的表的数量，本地事务可以继续细分为本地单分区事务和本地多分区事务。

单分区事务

本地单分区事务需要满足以下两个条件：

- 事务涉及的操作总共涉及一个分区。
- 分区的 Leader 与 Session 创建的 Server 相同。

本地单分区事务是最简单的模型，事务的提交采用了极高的优化。

单机多分区事务

类似于本地单分区事务，本地多分区事务也需要满足两个条件：

- 事务涉及的表所涉及的多个分区， 其 Leader 在同一个 Server 上。
- 分区 Leader 与 Session 创建的 Server 相同。

由于 OceanBase 数据库分区级日志流的设计，单机多分区事务本质上也是分布式事务。为了提高单机的性能，OceanBase 数据库对事务内参与者副本分布相同的事务做了比较多的优化，相对于传统两阶段提交，大大提高了单机事务提交的性能。

7.1.8. 分布式事务

7.1.8.1. 分布式事务概述

OceanBase 数据库的事务类型由事务 session 位置和事务涉及的分区 leader 数量两个维度来决定，主要分为分布式事务和单分区事务。

- 分布式事务

  以下两种场景的事务均称为分布式事务：

  - 事务涉及的分区数量大于一个。
  - 事务涉及的分区数量只有一个，且分区 leader 和事务 session 位置不在同一个 server。

- 单分区事务

  事务涉及的分区数量只有一个，且分区 leader 和事务 session 在同一个 server。

  分布式事务满足事务的所有属性，同样需要满足 ACID 的特性。在多机数据修改，且要保证原子性的场景，分布式事务能够发挥重要作用。

为了保证上述特性，通常采用两阶段提交协议。有关两阶段提交协议的详细介绍，参见 两阶段提交。

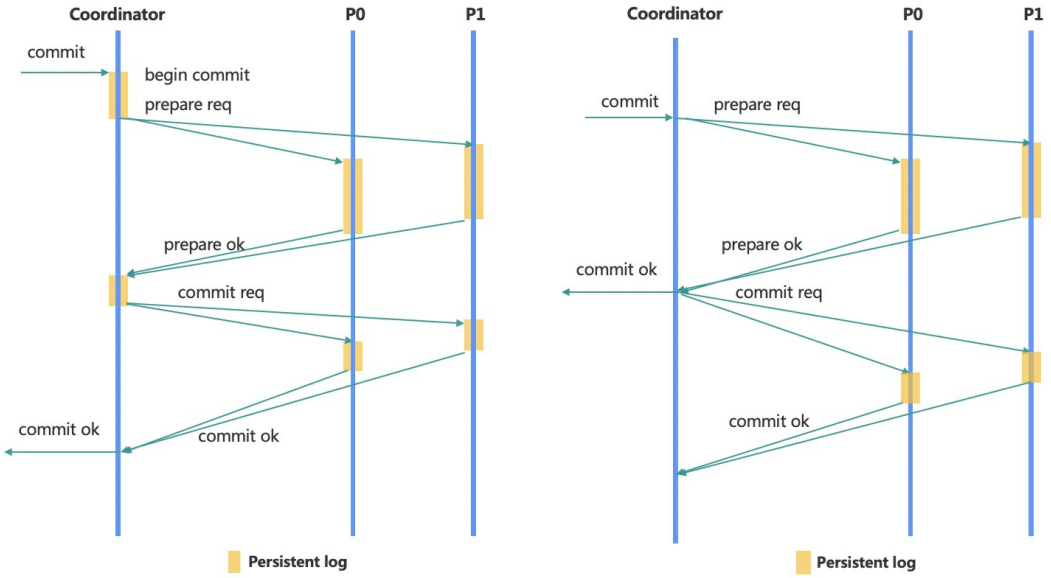
7.1.8.2. 两阶段提交

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。

两阶段提交协议中包含两种角色，协调者（Coordinator）和参与者（Participant）。协调者负责整个协议的推进，使得多个参与者最终达到一致的决议。参与者响应协调者的请求，根据协调者的请求完成 prepare 操作及 commit/abort 操作。

分布式事务提交流程

传统和 OceanBase 数据库两阶段提交的流程如下图所示。



PREPARE 阶段

协调者：协调者向所有的参与者发起 prepare request

参与者：参与者收到 prepare request 之后，决定是否可以提交，如果可以则持久化 prepare log 并且向协调者返回 prepare 成功，否则返回 prepare 失败。

COMMIT阶段

协调者：协调者收齐所有参与者的 prepare ack 之后，进入 COMMIT 状态，向用户返回事务 commit 成功，然后向所有参与者发送事务 commit request。

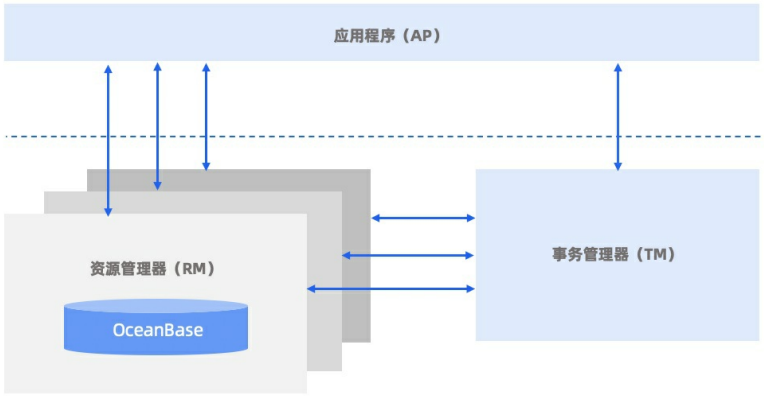
参与者：参与者收到 commit request 之后释放资源解行锁，然后提交 commit log，日志持久化完成之后给协调者回复 `commit ok` 消息，最后释放事务上下文并退出。

7.1.9. XA 事务

XA 协议是由 X/Open 公司于 1991 年发布的一套标准协议。XA 是 eXtended Architecture 的缩写，因此该协议旨在解决如何在异构系统中保证全局事务的原子性。

分布式事务处理模型

分布式事务处理（Distributed Transaction Processing，DTP）模型定义了一个标准化的分布式事务处理的体系结构以及交互接口。应用程序（Application Program，AP）能够访问由多个资源管理器（Resource Manager，RM）提供的资源。其中，每个资源管理器都具有独立性，且不必是同构的。全局事务的原子性由事务管理器来负责，各个模块的交互如下图所示。



如图所示，在分布式事务处理模型中，OceanBase 数据库系统作为资源管理器，负责管理部分数据资源。为了与事务管理器进行交互，OceanBase 数据库提供了一套标准的 XA 协议接口。为了实现这些接口，OceanBase 数据库内部有一套子程序来保证每一个 XA 协议接口的外部语义的标准性。事务管理器通过调用接口来触发 OceanBase 数据库内部的子程序，子程序处理完毕后将结果返回给事务管理器，事务管理器将会根据返回的结果来进行下一步处理。

基本概念

在进一步了解分布式事务处理模型前，您需要了解以下基本概念：

- 应用程序（application program，AP）

  应用程序定义了全局事务的边界，并指定了构成全局事务的相关数据访问操作。

- 资源管理器（resource manager，RM）

  资源管理器管理了一个可共享的、可恢复的数据资源，例如传统的关系型数据库管理系统。当发生故障后，资源管理器可以将数据资源恢复到一致状态。OceanBase 数据库系统可以作为资源管理器。

- 事务分支（branch）

  通常，一个全局事务将会访问多个资源管理器上的数据，该全局事务在某一个资源管理器上的执行可以看作该全局事务的一个分支（branch）。

- 事务管理器（transaction manager，TM）

  事务管理器提供了用于指定事务边界的接口，并负责全局事务的提交和恢复。当一个全局事务涉及多个资源管理器，为了保证该全局事务提交的原子性，事务管理器采用两阶段提交协议。

原子性保证

当全局事务进入到提交阶段后，事务管理器将需要保证该全局事务的原子性。事务管理器采用了两阶段提交协议，主要流程如下：

1. 准备阶段：事务管理器向该全局事务涉及的资源管理器发送准备请求，每个资源管理器收到准备请求后，将持久化本地 XA 事务分支的修改，待持久化成功后，将成功消息返回给事务管理器。在该阶段中，OceanBase 数据库系统将会持久化对应事务分支的 Redo 日志。
2. 提交/回滚阶段：当收齐所有该全局事务涉及的资源管理器的成功响应后，事务管理器将向这些资源管理器发送提交请求；否则，发送回滚请求。当资源管理器收到提交或回滚请求后，提交或回滚对应的事务分支。在该阶段中，OceanBase 数据库系统将会根据请求的类型，为对应的事务分支持久化相应的日志。具体来说，如果是提交请求，则将持久化提交日志；如果是回滚请求，则将持久化回滚日志。在持久化结束后，释放该事务分支的资源。

紧耦合模式

在分布式事务处理模型中，可能存在多个资源管理器底层访问同一个数据库。因此，存在一个全局事务的多个事务分支访问同一个数据库。在这种情况下，如果将该全局事务设置为紧耦合模式，那么访问同一个数据库的多个事务分支将共享资源。也就是说，这些分支之间可以互相看见彼此的修改。在 OceanBase 数据库系统中，如果两个分支为紧耦合分支，那么这两个分支将共享锁资源。具体来说，当一个事务分支对某个数据项加锁之后，另一个事务分支便可以认为自己已对该数据项加锁。

优化机制

一阶段提交

如果某个全局事务访问的数据仅涉及一个资源管理器上的数据资源，那么该全局事务不必采用两阶段提交机制。具体来说，当某个全局事务进入到提交阶段后，如果 TM 端发现该全局事务仅访问了一个资源管理器上的数据，那么事务管理器可以向对应的资源管理器发送一阶段提交请求。如果该资源管理器为OceanBase数据库系统，收到一阶段提交请求后，便可以直接提交对应的事务分支，该请求处理结束后，OceanBase数据库系统可以释放事务持有的资源。当收到该一阶段提交请求的响应后，事务管理器便可以返回结果，不必向资源管理器再发送任何请求。

只读事务分支

对于某个全局事务，如果其涉及的某个资源管理器仅提供了读取服务，那么当该全局事务进入到提交阶段后，该资源管理器不需要为该全局事务持久化做任何的数据修改。当 OceanBase 数据库系统收到某个事务分支的准备请求后，如果发现该事务分支没有修改任何数据，那么可以返回一个特殊的响应消息，然后便可以释放该事务分支持有的资源。当事务管理器收到该特殊响应后，便可以知道该事务分支为只读事务分支，后续将不再向该资源管理器发送两阶段提交中的提交或回滚请求。

7.2. 事务并发和一致性

7.2.1. 数据并发性和一致性概述

数据并发性

为了更好地提高事务的处理能力，数据库允许用户通过事务并发地访问与修改同一个数据，我们需要为这种数据并发性来定义语义。

数据一致性

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。

> **说明**
>
> 数据一致性不等价于 ACID 中的 C (Consistency)。

并发控制

以上两种语义我们称为并发控制模型，也就是 ACID 中的 I (Isolation level).

最简单的并发控制就是串行（serial）执行，其中串行执行是指一个进程在另一个进程执行完(收到触发操作的回应)一个操作前不会触发下一个操作. 但这明显不符合高并发的需求. 因此学者们提出了一种可串行化（serializable），即可以通过并行(非串行)地执行事务内的多个操作，但是最终需要达到和串行执行相同的结果。

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。

但是实际上上述两种实现机制提供可串行化隔离级别会极大地影响性能，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，也是 Oceanbase 数据库的选择。如何定义隔离级别的抽象，能给予使用者在性能上和语义易用性上的平衡感是设计事务隔离级别（Isolation）的关键之一。

7.2.2. 多版本读一致性

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：

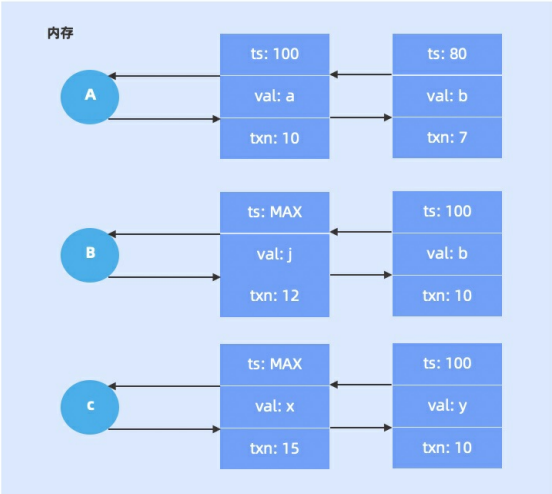
- 未提交事务：不能读到非本事物的未提交事务，否则若对应事务回滚，就会产生脏读（dirty read）。
- 事务一致性快照：要读取小于读取版本号的所有提交数据，来保证一个用户可理解的一致性点，否则我们就会产生会返回一半事务（fractured read）。
- 读写不互斥：在满足未提交事务与事务一致性点的前提下，依旧要保证读写不互斥。

多版本读一致性使用

多版本读一致性在数据库内部是广泛使用的，也是实现并发控制的关键之一：

- 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。
- 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。
- 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。
- 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。

用户在使用多版本的过程中，如下图所示：



如上图左所示，数据 A 包含 100 版本已经提交的数据 a，其对应的事务为事务 10 以及已经提交的数据 b，其对应的事务为事务 7；数据 B 包含未决定版本的数据 j 以及对应事务事务 12 以及已经提交的数据 b，其对应的事务为事务 10；数据 C 包含未决定版本的数据 x 以及对应事务事务 15 以及已经提交的数据 y，其对应的事务为事务 10。

多版本读一致性实现

事务表

事务上下文

事务	状态	版本
6	ABORT	MAX
7	COMMIT	80
10	COMMIT	100
12	RUNNING	MAX
15	RUNNING	130
...		

正在执行中的事务会存放在事务表内，根据不同的事务状态，来决定是否要读取到对应的数据。其中数据状态包含提交（COMMIT）、执行（RUNNING）、回滚（ABORT）。对于执行（RUNNING）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` 代表事务最终的版本，也是我们一致性位点的决定因素。

如上图右所示，事务 6 处于回滚状态；事务 7 处于提交状态， `全局提交版本号` 为 80；事务 12 处于执行状态，不存在 `本地提交版本号` ；事务 15 处于执行状态， `本地提交版本号` 为 130。

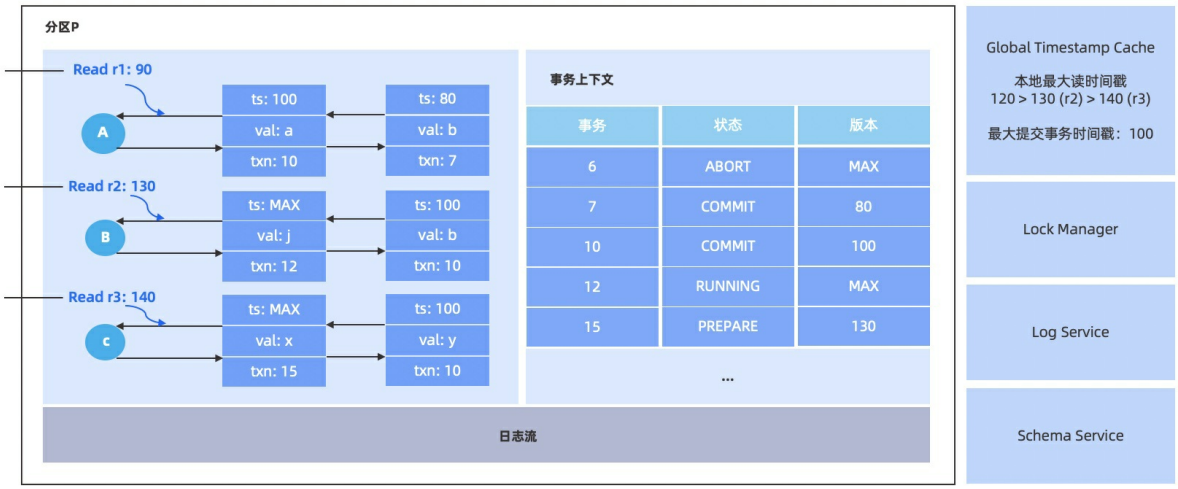
读请求处理

在读取的时候，我们会使用读版本号来读取对应的数据。

接下来我们分开来分析，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。如下图所示，读取请求 r1 以 90 作为读版本号进行读取，根据快照读的策略，会选择版本号为 80，数据为 b 的数据进行读取。

当读取到 RUNNING 状态的事务时，可以安全地跳过这个数据。如下图所示，读取请求 r2 以 130 作为读版本号进行读取，可以安全跳过未进入两阶段提交的事务 12 读取到版本号为 100，数据为 b 的数据。

当读取到 PREPARE 状态的事务时，无法确定事务是否会提交，因此会等在这行的事务上。如下图所示，读取请求 r3 以 140 作为读版本号进行读取，等待两阶段提交状态且 本地提交时间戳 为 130 的时候最后决定 全局提交时间戳 和读时间戳 140 的关系。



7.2.3. 并发控制

7.2.3.1. 并发控制概述

并发控制模型

每个事务包含多个读写操作，操作对象为数据库内部的不同数据。最简单的并发控制就是串行（serial）执行，指一个进程在另一个进程执行完（收到触发操作的回应）一个操作前不会触发下一个操作。但这明显不符合高并发的需求。因此学者们提出了一种可串行化（serializable），即可以通过并行（非串行）地执行事务内的多个操作，但是可以达到和串行执行相同的结果。

我们可以利用事务中的读写操作来为事务来建立依赖关系（依赖关系代表事务串行化成串行执行序后的事务定序，若事务 B 依赖事务 A，事务 A 应该排在事务 B 前面）：

- 写写冲突（Write Dependency）：当事务 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。
- 写读冲突（Read Dependency）：当事务 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。
- 读写冲突（Anti Dependency）：当事务 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。

通过冲突来定义的可串行化，一般称为冲突可串行化（conflict serializable），可以轻易地通过以上的冲突机制来分析。当事务间的冲突关系没有成环的话，就可以保证冲突可串行化。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成环的事务保证不会产生环。

但是实际上实现可串行化隔离级别的商业数据库少之又少，其中上述两种实现机制都有极大的性能代价，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，这种异常也就是我们经常说的写偏斜（Write Skew）。这是快照读暴露给用户的异常。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。如何定义隔离级别的抽象，能给予使用者在性能上和语义易用性上的平衡感是设计事务隔离级别（Isolation）的关键之一。

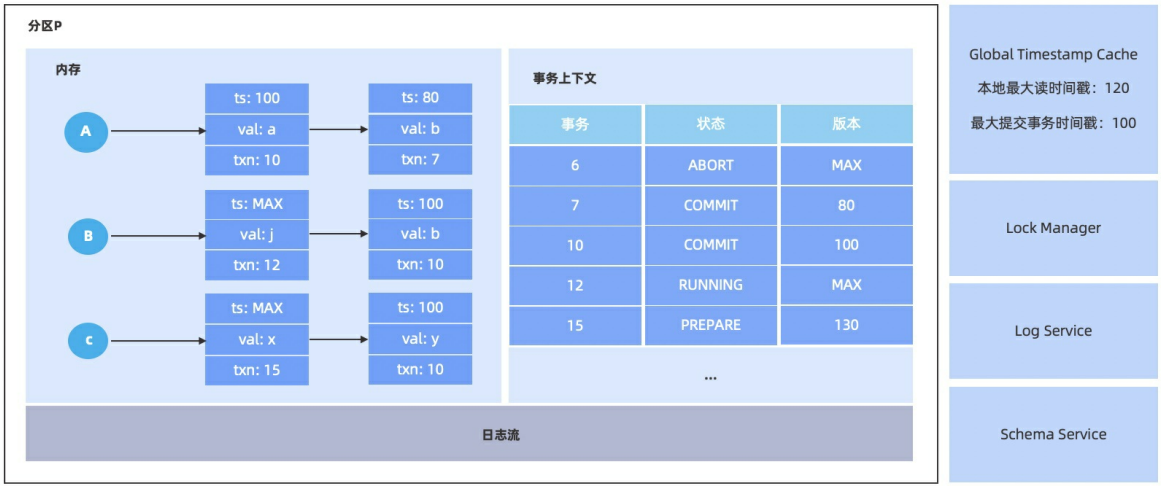
OceanBase 数据库的并发控制模型

OceanBase 数据库现在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&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，如图所示的 `本地最大读时间戳` 和 `最大提交时间戳` 。其次，在内存中会为每一次更新记录一个新的版本（可以做到读写不互斥）。

如图所示，在内存中有三行数据 A、B 和 C；其中每次通过版本（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。

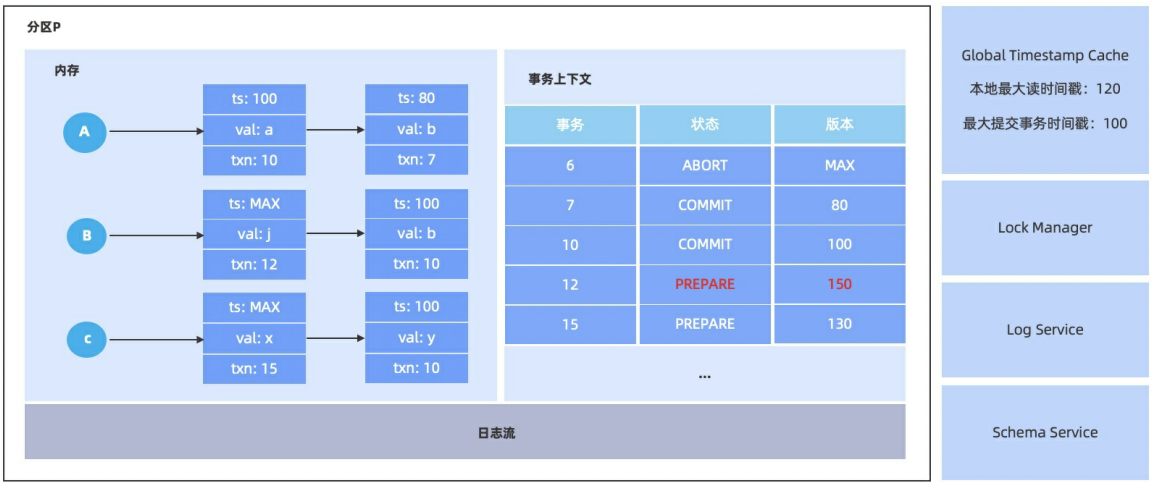


由图中可知，全局时间戳获取服务维护了最大遇到的事务的读时间戳和最大在已经提交的事务的提交时间戳，分别为 120 与 100（之后会提到这两个时间戳的作用）。在内存中，数据 A 包含 100 版本已经提交的数据 a。其对应的事务为事务 10；类似数据 B 包含未决定版本的数据 j 以及对应事务事务 12 以及数据 C 包含未决定版本的数据 x 以及对应事务事务 15。事务表中包含了对应事务以及其对应状态，如事务事务 15 正在以 130 版本进入两阶段提交状态。

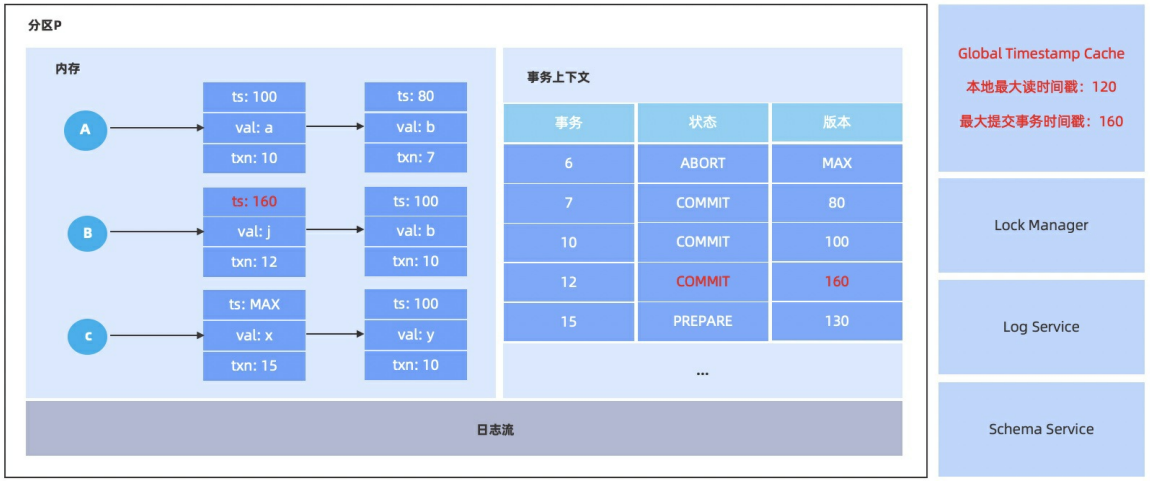
提交请求处理

OceanBase 数据库 的分布式事务简单来说有三个状态，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。因此我们对于事务中维护一个 本地提交版本号 （local commit version，即 prepare version），事务的 全局提交版本号 （global commit version，即 commit version）是由所有分区本地提交版本号的最大值决定的。对于每个分区的保证是最后事务的 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 的。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。

当事务提交时，我们会走对应的两阶段提交，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。这个保证是为了做单值的读写冲突（anti dependency），根据保证，可以得到我们的提交时间戳一定大于之前所有的读取，因此我们在串行执行中可以在这些之前的读取后面。如图所示，事务 12 进入提交阶段，并设置状态为 PREPARE，设置本地事务版本号为本地 最大读时间戳 120 与取 GTS 为 150 的最大值 150 作为 本地事务版本号 。

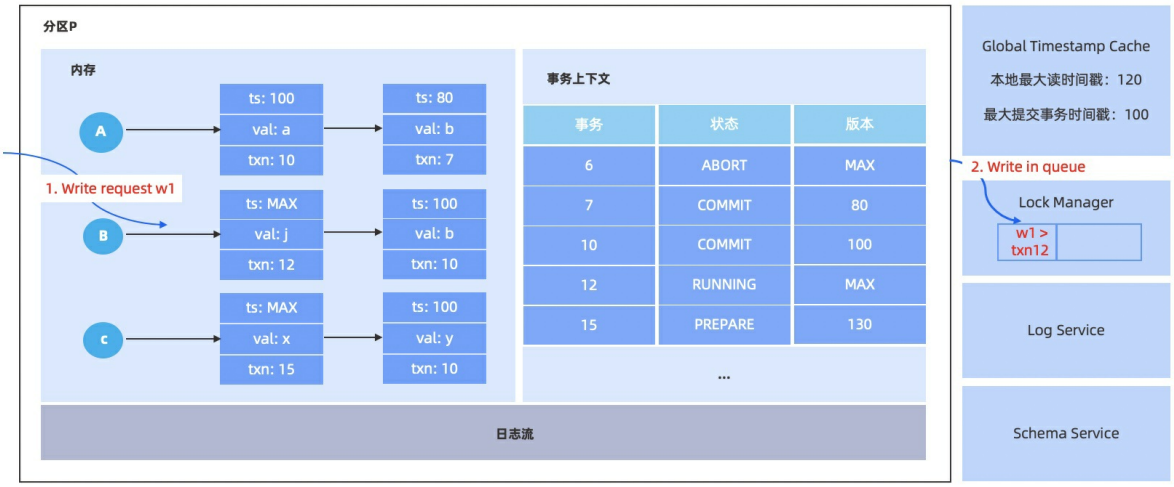


在两阶段结束之前，我们的保证只有 全局提交时间戳 大于等于 本地提交时间戳 ，当我们收到两阶段提交的提交消息后，便能知道全局的提交时间戳，如图所示，回填状态为提交，时间戳为 160，并异步地回填到更新数据上，保证之后不需要再来查事务表。其次，会更新 最大提交事务时间戳 来提供之后的读请求优化，并在锁队列中唤醒对应的事务。



写请求处理

在写入数据的时候，为了保证写写冲突（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，就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，实现了等待队列，通过锁或超时来唤醒这个写请求。如图所示当准备更新数据 B 时，由于存在活跃事务事务 12 正在修改数据 B，因此会将这次的写入请求放到锁队列中，等待事务事务 12 的唤醒。



快照读隔离级别为了防止读写冲突（anti dependency）和写读请求（read dependency）成环，即丢失更新(lost update)，尽管写入或唤醒后加锁成功，会用读时间戳，跟本行上维护的 **最大提交事务时间戳** 作比较，如果读时间戳小于行上的最大提交事务时间，则会回滚掉此事务。比如说若上图写操作的读时间戳是 100，事务 12 以时间戳 160 提交，那么就会触发写操作对应事务的回滚，对应的报错(`TRANSACTION_SET_VIOLATION`)。

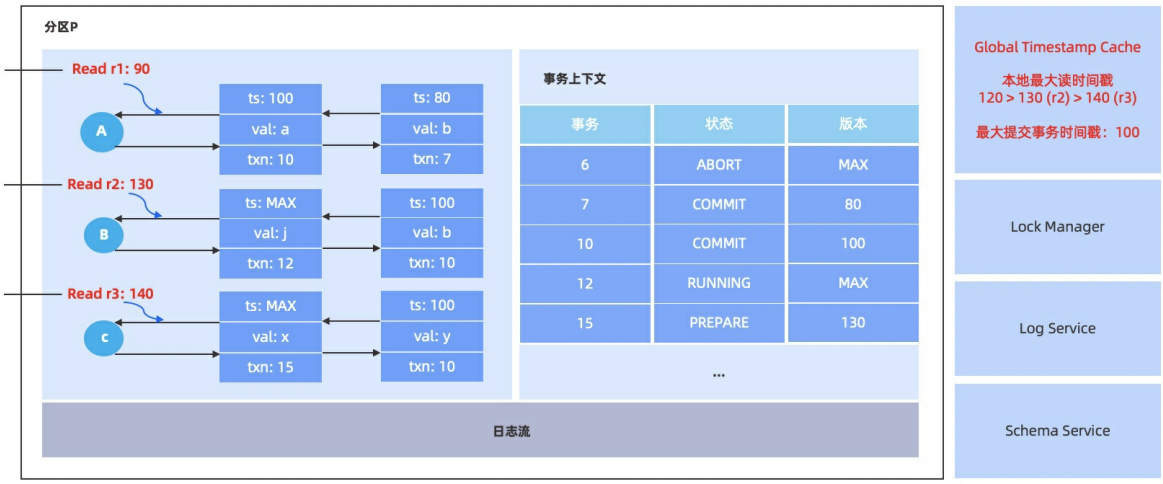
读请求处理

在读取的时候，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** 。依赖之前给予的保证，可以很优雅地在分布式场景下处理读请求。

接下来分开来分析，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。如下图所示，读取请求 r1 以 90 作为读版本号进行读取，根据快照读的策略，会选择版本号为 80，数据为 b 的数据进行读取。

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。如下图所示，读取请求 r2 以 130 作为读版本号进行读取，会先推高 **最大读事务时间戳** ，保证之后事务以大于 130 的 **本地提交版本/全局提交版本** 提交，然后跳过未进入两阶段提交的事务 12 读取到版本号为 100，数据为 b 的数据。

当读取到 PREPARE 状态的事务时，由于给予的保证，对于 **本地时间戳** 大于读时间戳的事务，和之前的分析一样，可以保证不用读到。但是若 **本地时间戳** 小于读时间戳，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，因此解决方案是，优雅地等在这行的事务上（内部称为 `lock for read`），两阶段提交在 OceanBase 数据库的假设中应该会是很快完成的过程。如下图所示，读取请求 r3 以 140 作为读版本号进行读取，会先推高 **最大读事务时间戳**，保证之后事务以大于 140 的 **本地提交版本/全局提交版本** 提交，然后等待两阶段提交状态且 **本地提交时间戳** 为 130 的时候最后决定 **全局提交时间戳** 和读时间戳 140 的关系。



7.2.3.2. 锁机制

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，锁机制是保证正确的数据并发性和一致性很重要的一点。

OceanBase 数据库的锁机制使用了以数据行为级别的锁粒度。同一行不同列之间的修改会导致同一把锁上的互斥；而不同行的修改是不同的两把锁，因此是无关的。类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，选择将锁存储在行上（可能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。其次，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。

> **注意**
>
> - `SELECT ... FOR UPDATE` 无法做到读写不互斥。
> - 在事务提交过程中，为了维护事务的一致性快照，会有短暂的读写互斥，我们称之为 lock for read。

OceanBase 数据库锁机制的使用

在深入之前，我们先来看一下如何使用 OceanBase 数据库的行锁能力。如下所示是一个很常见的业务 SQL 来更新货物的信息。

```
UPDATE GOODS
SET    PRICE = ?, AMOUNT = ?
WHERE  GOOD_ID = ?
AND    LOCATION = ?;
```

在上述 SQL 中，根据用户填入的货物 ID 和 地址，去更新对应的价格和存量。对应事务中的一个 SQL，在事务结束前，对应货物 ID 和 地址的数据的一行会被加上行锁，所有并发的更新都会被阻塞并等待。从而预防并法的修改导致的脏写（Dirty Write）。由此可见用户在更新数据的同时，隐式地为修改的数据行上加上了对应的锁，用户无需显示地指示锁的范围等的情况下，就可以依赖 OceanBase 数据库内部的机制做到并发控制的效果。

当然用户可以显式地制定使用锁机制。如下所示是一个很常见的业务 SQL 来互斥地获取货物的信息。

```
SELECT PRICE = ?, AMOUNT = ?
FROM   GOODS
WHERE  GOOD_ID = ?
AND    LOCATION = ?
FOR UPDATE;
```

在上述 SQL 中，根据用户填入的货物 ID 和 地址，去获取对应的价格和存量。对应事务中的一个 SQL，在事务结束前，对应货物 ID 和地址的数据的一行会被加上行锁，所有并发的更新都会被阻塞并等待。从而做到用户指定的显示加锁。在不同的业务需求下，是极其重要的一点。

OceanBase 数据库锁机制的粒度

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。

OceanBase 数据库锁机制的互斥

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以需要获取行锁的操作产生互斥。

OceanBase 数据库锁机制的存储

OceanBase 数据库的锁存储在行上，从而减少内存中所需要维护的锁数据结构带来的开销。在内存中，当事务获取到行锁时，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。

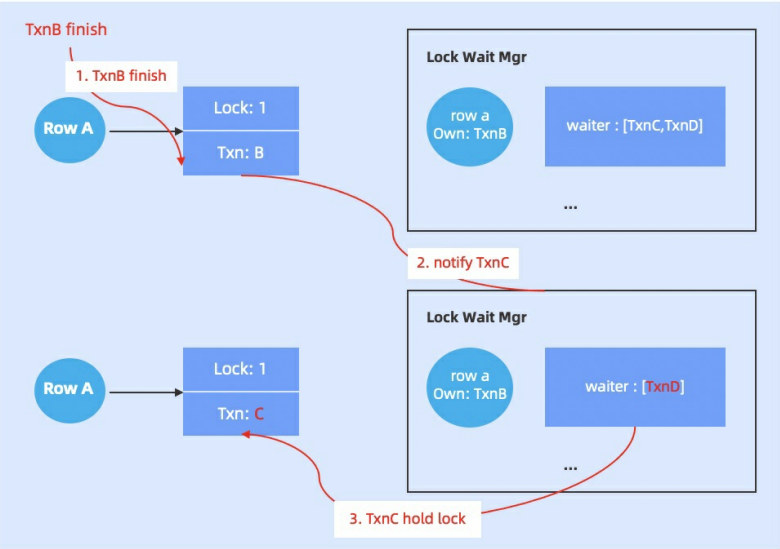
当数据被转储当 SSTable 后，在宏块内部的数据上，记录着对应的事务标记。其余事务依旧需要通过事务标识来辨识是否可以允许访问对应的数据。与内存中的锁机制不同的是，由于 SSTable 不可变的特性，无法在事务释放行锁后，立即清楚宏块内部的数据上的事务标记。当然依旧可以通过事务标识来确认找到对应的事务信息来确认事务是否已经解锁。

OceanBase 数据库锁机制的释放

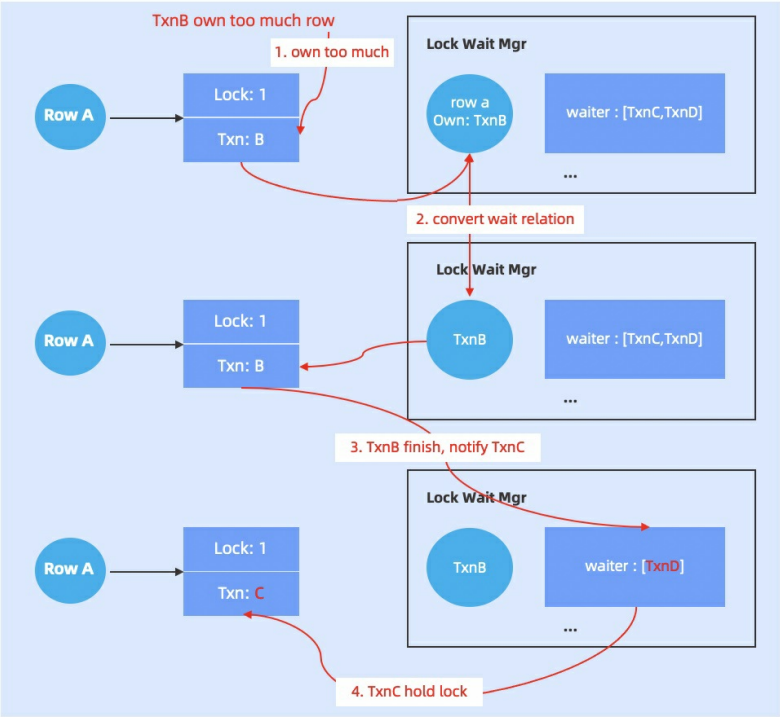
类似于大部分的两阶段锁实现，OceanBase 数据库的锁在事务结束（提交或回滚）的时候释放的，从而避免数据不一致性的影响。OceanBase 数据库还存在其余的释放时机，即 SAVEPOINT，当用户选择回滚至 SAVEPOINT 后，事务内部会将 SAVEPOINT 及之后所有涉及数据的行锁，全部根据 OceanBase 数据库锁机制的互斥 中介绍的机制进行释放。

OceanBase 数据库锁机制的唤醒

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。



除了行与事务的等待关系，OceanBase 数据库可能会维护事务与事务的等待关系。为了减小对于内存的占用，OceanBase 数据库内部可能会将行与事务的等待关系转换为事务与事务的等待关系。如图所示，行 A 被事务 B 持有，被事务 C 与事务 D 等待，并被转换为事务 B 被事务 C 与事务 D 等待。当事务 B 结束后，由于不明确知道行之间的锁等待关系，会同时唤醒事务 C 与事务 D。



OceanBase 数据库锁机制的死锁

锁机制的实现会导致死锁，死锁是指对于资源的循环依赖，举例来说，当事务 A 与事务 B 同时获取资源 C 与 D 的情况下，若事务 A 优先获取到资源 C 并去获取资源 D；而事务 B 优先获取到资源 D 并去获取资源 C。此时若没有任何人愿意放弃自己已经获取到的资源，就没有事务可以正常结束。

基于超时的死锁解决

OceanBase 数据库 V3.2 版本之前，未包含主动死锁检测的能力。因此我们主要是依赖超时回滚机制来解决业务逻辑上的死锁。

存在三种超时机制用来解决对应的问题：

- 锁超时机制：配置项名称为 `ob_trx_lock_timeout`，默认为语句超时时间，若加锁等待超过锁超时时间，则会回滚对应的语句，并返回锁超时对应的错误码。此时，由于某一个循环依赖中的资源依赖已经消失，因此就不再存在死锁。以事务 B 获取资源 C 超时为例，只要事务 B 结束，则事务 A 就可以获取到对应的资源 D。
- 语句超时机制：配置项名称为 `ob_query_timeout`，默认为 `10s`，若加锁等待超过语句超时时间，则会回滚对应的语句，并返回语句超时对应的错误码。此时，由于某一个循环依赖中的资源依赖已经消失，因此就不再存在死锁。以事务 B 获取资源 C 超时为例，只要事务 B 结束，则事务 A 就可以获取到对应的资源 D。
- 事务超时机制：配置项名称为 `ob_trx_timeout`，默认为 `100s`，若加锁等待超过事务超时时间，则会回滚对应的事务，并返回语句事务对应的错误码。由于某一个循环依赖中的资源依赖已经消失，因此就不再存在死锁。以事务 B 超时为例，由于事务 B 结束，事务 A 就可以获取到对应的资源 D。

主动死锁检测

从 OceanBase 数据库 V3.2 版本开始，除了以上基于超时的死锁解决机制，我们还实现了主动死锁检测机制。

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。

基于优先级是指，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。

多出度是指，每一个事务都可以同时等待超过一个的其他事务。

分布式死锁检测是指，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。

实现原理

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。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。

在边跟踪算法中死锁探测的基本原理是令牌值可以经由自己发出后回到自己，但是在多出度场景下采用单令牌值设计时可能出现死锁环路中最大的令牌值并不属于这个环路中的任何一个节点的情况，此情况下将检测不到死锁，该场景称为“环外污染”，因此 LCL 死锁检测方案引入了“路径深度”概念，每个节点维护一个路径深度值，在环路中的节点的路径深度值随时间推移可以无限增长，而不在环路中也不被环路中节点所触达的节点的路径深度值存在增长上限，约束节点只能接受路径深度至少和自己一样大的节点传递的令牌来避免“环外污染”，并通过定期清理节点上当前的令牌值来消除在算法运行早期阶段已经出现的“环外污染”，由此保证算法工作在多出度下的正确性。

具体实现

当一个事务 A 遇到加行锁失败时，事务 A 在等待行锁解开的同时，会获取持有行锁的事务 B 的 ID，并创建出一个死锁检测节点 a（下文称为 Detector(a)），为 Detector(a) 记录到事务 B 的 Detector(b) 的单向依赖关系。

每个节点要维护以下状态：

当一个 Detector 节点被建出时生成两个令牌状态，一个公共令牌值（public label），一个私有令牌值（private label），它根据自己的优先级生成一个全局唯一性令牌（优先级越高的节点，令牌值越大），并用其初始化两个令牌值，并同时生成一个初始为 0 的深度值 lclv ((Lock Chain Length)。

每一个 Detector 节点都维护一个依赖列表，列表中记录了依赖的其他节点的网络位置信息。

从时间轴划分，每 1.4s 划分为一个 LCL 周期。

在每个周期开始的时候，每个节点重置自己的 public label 为自己的 private label。

LCL 周期的前 700ms 称为 LCLP 周期（Lock Chain Length Proliferating），每个节点在该周期中定期将自己的状态的 lclv 值发送给依赖列表中的所有下游节点，每个节点在接收到上游节点发送来的 lclv 值后，将自己的 lclv 值更新为 max(lclv, received_lclv + 1)。

LCL 周期的后 700ms 称为 LCLS 周期（Lock Chain Length Spreading），每个节点在该周期中定期将自己的状态的 { lclv, public label } 发送给依赖列表中的所有下游节点，每个节点在接收到上游节点发送来的值后，若其 lclv 值不小于自己当前的 lclv 值，则更新自己的 public label 为 min(public label, received public label)，更新自己的 lclv 值为 received lclv。

检测到死锁：当一个节点收到的 public label 是自己的 private lavel 时，它就发现了死锁。

视图

视图 `__all_virtual_deadlock_event_history` 记录了所有发生过的死锁事件以及参与这些事件的事务，并注明了在一个死锁事件中哪个事务最终被 kill 掉。

7.2.4. 事务隔离级别

7.2.4.1. 事务隔离级别概述

隔离级别是根据事务并发执行过程中必须防止的现象来定义的。

可防止的现象包括：

- 脏读（Dirty Read）：一个事务读到其他事务尚未提交的数据。
- 不可重复读（Non Repeatable Read）：曾经读到的某行数据，再次查询发现该行数据已经被修改或者删除。例如：`select c2 from test where c1=1;` 第一次查询 `c2` 的结果为 `1`，再次查询由于其他事务修改了 `c2` 的值，因此 `c2` 的结果为 `2`。

- 幻读（Phantom Read）：只读请求返回一组满足搜索条件的行，再次执行发现另一个提交的事务已经插入满足条件的行。

基于上述三种现象，ANSI 和 ISO/IEC 定义了四种隔离级别，这四种隔离级别如下：

- 读未提交（Read Uncommitted）
- 读已提交（Read Committed）
- 可重复读（Repeatable Read）
- 可串行化（Serializable）

四种隔离级别比较如下所示。

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不会	可能	可能
可重复读	不会	不会	可能
可串行化	不会	不会	不会

OceanBase 数据库目前支持了以下几种隔离级别：

- Oracle 模式
  - 读已提交（Read Committed）
  - 可串行化（Serializable）
- MySQL 模式
  - 读已提交（Read Committed）
  - 可重复读（Repeatable Read）

OceanBase 数据库默认的隔离级别为读已提交（Read Committed）。

7.2.4.2. Oracle 模式

Oracle 模式支持的隔离级别

Oceanbase 数据库在 Oracle 模式下，支持两种隔离级别：

- 读已提交（Read Committed）：一个事务执行的查询，只能看到这次查询开始之前提交的数据。读已提交无法防止不可重复读和幻读两种异常现象。如果冲突的事务比较少，简单高效的读已提交隔离级别，对应用来说是足够的。
- 可串行化（Serializable）：一个事务的查询，只能看到事务开始之前提交的数据。这是最严格的隔离级别，可以防止脏读、不可重复读和幻读三种异常现象，事务看起来就像是串行执行的。

OceanBase 数据库默认的隔离级别为读已提交（Read Committed）。

隔离级别设置方法

设置隔离级别有两种方式，分别为事务级别及 session 级别。

- Transaction level

```
obclient> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- Session level

```
obclient> ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
```

使用限制

- 不能在事务执行过程中设置隔离级别，否则会报如下错误：

```
ERROR:ORA-01453: SET TRANSACTION must be first statement of transaction
```

- 在开启可串行化隔离级别时需要确保全局时钟服务（Global Timestamp Service）是打开的。
- Session 需要维护 session 级别的事务隔离级别，在开启事务时获取 session 级别的事务隔离级别，该隔离级别可以被事务级别的隔离级别覆盖。

7.2.4.3. MySQL 模式

MySQL 模式支持的隔离级别

OceanBase 数据库在 MySQL 模式下，支持两种隔离级别：

- 读已提交（Read Committed）：一个事务执行的查询，只能看到这次查询开始之前提交的数据。读已提交无法防止不可重复读和幻读两种异常现象。如果冲突的事务比较少，简单高效的读已提交隔离级别对应用来说是足够的。
- 可重复读（Repeatable Read）：事务内不同时间读到的同一批数据是一致的。无法防止幻读这种异常现象。

OceanBase 数据库默认的隔离级别为读已提交（Read Committed）。

隔离级别设置方法

设置隔离级别有两种方式，分别为全局 Global 级别及 Session 级别。

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

使用限制

OceanBase 数据库 MySQL 模式的 Repeatable Read 隔离级别的实现，能比 MySQL 数据库防止更多的异常。

7.2.5. 弱一致性读

OceanBase 数据库提供了两种一致性级别（Consistency Level）：STRONG 和 WEAK。STRONG 指强一致性，读取最新数据，请求路由给主副本；WEAK 指弱一致性，不要求读取最新数据，请求优先路由给备副本。OceanBase 数据库的写操作始终是强一致性的，即始终由主副本提供服务；读操作默认是强一致性的，由主副本提供服务，用户也可以指定弱一致性读，由备副本优先提供服务。

一致性级别的指定方式

有两种方式指定一致性级别：

- 通过 `ob_read_consistency` 系统变量指定
  - 设置 Session 变量，影响当前 Session

```
obclient> SET ob_read_consistency = WEAK;
obclient> SELECT * FROM t1;  -- 弱一致性读
```

  - 设置 Global 变量，影响之后新建的所有 Session

```
obclient> SET GLOBAL ob_read_consistency = STRONG;
```

- 指定 Hint 方式
  - 指定 WEAK Consistency，优先级高于 `ob_read_consistency`

```
obclient> SELECT /*+READ_CONSISTENCY(WEAK) */ * FROM t1;
```

  - 指定 STRONG Consistency

```
obclient> SELECT /*+READ_CONSISTENCY(STRONG) */ * FROM t1;
```

SQL 语句的一致性级别

- 写语句 DML (INSERT/DELETE/UPDATE)：强制使用 STRONG Consistency，要求基于最新数据进行修改。
- `SELECT FOR UPDATE`（SFU）：与写语句类似，强制使用 STRONG Consistency。
- 只读语句 `SELECT`：用户可以配置不同的 Consistency Level，满足不同的读取需求。

事务的一致性级别

弱一致性读的最佳实践是为不在事务中的 SELECT 语句指定 WEAK 一致性级别，它的语义是确定的。对于显式开启事务的场景，语法上，OceanBase 数据库允许不同的语句配置不同的一致性级别，这样会让用户很困惑，而且如果使用不当，SQL 会报错。

原则：

- 一致性级别是事务级的，事务内所有语句采用相同的一致性级别。
- 由事务的第一条语句决定事务的一致性级别，后续的 SELECT 语句如果指定了不同的一致性级别，则强制改写为事务的一致性级别
- 写语句和 SFU 语句只能采用 STRONG，如果事务级一致性级别为 WEAK，则报错 `OB_NOT_SUPPORTED`。

下面举例说明。

```
BEGIN;
-- 修改语句，consistency_level=STRONG，整个事务应该是 STRONG
insert into t1 values (1);

-- SQL自身的 consistency_level=WEAK，但由于第一条语句为 STRONG，
-- 因此这条语句的 consistency_level 强制设置为 STRONG
select /*+READ_CONSISTENCY(WEAK) */ from t1;
COMMIT;

BEGIN;
-- SFU属于修改语句，consistency_level=STRONG，整个事务应该也是STRONG
select * from t1 for update;

-- SQL自身的consistency_level=WEAK，但由于第一条语句为STRONG，
-- 因此这条语句的consistency_level强制设置为STRONG
select /*+READ_CONSISTENCY(WEAK) */ from t1;
COMMIT;

BEGIN;
-- 第一条语句为WEAK
select /*+READ_CONSISTENCY(WEAK) */ from t1;

-- 虽然本条语句为STRONG，但是会继承第一条语句的consistency level，会强制设置为WEAK
select * from t1;
COMMIT;

BEGIN;
-- 第一条语句为WEAK
select /*+READ_CONSISTENCY(WEAK) */ from t1;

-- 修改语句，必须为STRONG，由于第一条语句为WEAK，这里会报错：NOT SUPPORTED
insert into t1 values (1);

-- SFU属于修改语句，必须为STRONG，这里同样会报错：NOT SUPPORTED
select * from t1 for update;
COMMIT;
```

因此，对于单条 SQL 而言，一致性级别的确定规则优先级从大到小可以概括为：

1. 根据语句类型确定的一致性级别，例如 DML 和 SFU 必须采用 STRONG。
2. 事务的一致性级别，如果语句在事务中，而且不是第一条语句，则采用事务的一致性级别。
3. 通过 Hint 指定的一致性级别。
4. 系统变量指定的一致性级别。
5. 缺省采用 STRONG。

与隔离级别关系

- STRONG 支持所有的隔离级别。
- WEAK 仅支持读已提交 `READ COMMITTED` 隔离级别，其他隔离级别下会报错 `OB_NOT_SUPPORTED` 。

弱一致性读配置项

2.2.x 及以后版本

名称	范围	语义
enable_monotonic_weak_read	租户级	是否开启单调读，默认为 false
max_stale_time_for_weak_consistency	租户级	弱一致性读最大落后时间，默认值是 5 秒
weak_read_version_refresh_interval	集群级	弱一致性读版本号刷新周期，默认值 50 毫秒

各个配置项具体含义如下：

- `enable_monotonic_weak_read` ：租户级单调读开关

  弱一致性读会路由到不同副本上，不同副本上读到的数据新旧没有保证；单调读开关打开后，OceanBase 数据库保证读到的数据版本不回退，保证单调性。一个典型的应用场景是保证因果序：两个事务 T1 和 T2，T1 提交之后，T2 才提交，如果客户端读到了 T2 事务的修改，那么之后一定可以读到 T1 事务的修改。

- `max_stale_time_for_weak_consistency` ：弱一致性读最大落后时间

  OceanBase 数据库弱一致性读提供有界旧保证，即保证读到的数据最多落后 `max_stale_time_for_weak_consistency` 时间，默认配置值是 5 秒，支持租户级配置。

  正常情况下，各个分区的备副本落后时间在 100 毫秒到 200 毫秒，弱一致性读的时效性在百毫秒级别；当出现网络抖动、无主等情况，弱一致性读的时效性会降低，一旦一个副本落后时间超过 `max_stale_time_for_weak_consistency` ，该副本将不可读，内部重试机制会重试其他有效副本；如果所有副本都不可读，则持续重试，直到语句超时。

  当开启单调读开关后，OceanBase 数据库内部会为每个租户维护一个 Cluster 级别的弱一致性读版本号，该版本号也满足 `max_stale_time_for_weak_consistency` 约束。它的生成方式是统计租户下所有分区副本回放进度的最小值，如果某些分区副本落后时间超过 `max_stale_time_for_weak_consistency` ，则不统计该副本。目前机制下，一个落后的副本会影响整体单调读的版本号，例如有两个分区的两个副本，一个副本落后 100ms，一个副本落后 1 秒，那么整体的单调读版本号是 1 秒。我们认为副本长时间落后不会是常态，正常情况下，单调读版本号都应该在百毫秒级别。

- `weak_read_version_refresh_interval` ：弱一致性读版本号刷新周期

  弱一致性读版本号刷新周期影响读取数据的新旧程度，它配置的值不能大于 `max_stale_time_for_weak_consistency` 。当它配置为 0 时，弱一致性单调读功能关闭，即不再维护 Cluster 级别弱一致性读版本号。另外，它是集群级别配置项，不支持租户级别配置。

2.2.x 之前版本

名称	范围	语义
enbale_causal_order_read	集群级	是否开启单调读，默认为 false
max_stale_time_for_weak_consistency	集群级	弱一致性读最大落后时间，默认为 5 秒

2.2 之前版本仅支持 Proxy 级别单调读，即只要客户端始终访问同一个 Proxy，就可以保证单调读。具体实现方式是在 Proxy 上维护单调递增的弱一致性读版本号。如果客户端跨 Proxy 访问，则不保证单调读。如果用户需要集群级别单调读，需要使用 2.2 及之后的版本。

弱一致性读时间戳

弱一致性读，通常是指备副本、备库上的查询语句。OceanBase 数据库的弱一致性读依旧返回事务一致性点，不会出现返回未提交事务和一半事务的情况。

弱一致性读时间戳，包括两个方面：

- 弱读执行之前，需要确定一个读快照数据。
- 分区自身实时维护一个最大安全可读的位点，凡是读快照大于该位点的读请求，均不能读该副本。

时间戳生成方式

OceanBase 数据库弱读一致性分为两大类：单调读和非单调读。不同的能力，时间戳生成的方式不同。

单调读

单调读是指根据读请求发起的绝对时间，后发起的请求使用的读快照一定不比前者小，确保读到的数据不会出现回退。这里的单调是针对语句读快照而言的。单调读的保证依赖集群级别单调弱读版本号，全局版本号需要做到不回退。

全局版本号的生成来源于每个 OBServer 维护的最小最大安全版本号，该版本号的生成依赖本机日志同步的进度。对于 OceanBase 数据库而言，事务的日志数据，由三个模块来维护，Clog、replay_engine 和 Transaction：

1. Clog 负责事务日志的本地落盘、发送备机、接收 Leader 同步的日志等操作。对备机而言，它为每个 Partition 维护了一个滑动窗口，将收到的日志，按照 logid 从小到大管理起来。日志从滑动窗口中顺序滑出，提交给 replay_engine 去回放。
2. `replay_engine` 负责日志的回放，它将同一个事务的多条日志 Hash 到同一个工作线程，保证同一个事务的日志顺序回放，但是同一个分区的多个事务是并发回放的。对一个分区的日志而言，日志按照提交时间戳有序串到链表中，由多个线程并发回放。
3. Transaction 负责事务状态的管理，Transaction 执行日志的回放操作，即将 Redo 数据回放到 MEMStore，并记录事务的状态。

由上描述可知，一条日志从 Clog 接收到提交 `replay_engine` 回放任务，再通过 Transaction 回放事务日志，为保证单个分区备机读不会读到一半的事务，需要找到该分区备机读的安全版本号，保证该版本号之前的所有事务都已经回放完成。计算公式如下：

```
slave_read_ts = min(clog_ts, replay_engine_ts, trans_service_ts) - 1
```

其中 `clog_ts` 表示 Clog 滑动窗口中下一条将要滑出或将要生成的日志 timestamp，`replay_engine_ts` 表示该 Partition 尚未回放日志的 timestamp 最小值，`trans_service_ts` 表示当前所有正在回放事务的 `prepare_log_ts` 的最小值。

举例说明：假设 clog 的滑动窗口中存在两条日志，logid 和日志的提交时间戳分别为（10，100），（11，200）；`replay_engine` 中，该 Partition 尚未回放的日志分别为 `（7，70）`，`（8，80）`；transaction 中，只有一个正在回放的事务，刚刚回放完 redo 和 prepare log，并且 `prepare log` 的 logid 为 `9`，`prepare_log_ts` = 90；此时该分区 **安全版本号**=`min(100, 70, 90) - 1 = 69`。

非单调读

相对于单调读，非单调读保证的能力较弱。非单调读则不保证前后发起的请求快照递增，由于同一份数据不同副本同步进度的差异，如果前后两次读取不同的副本，数据可能出现回退。

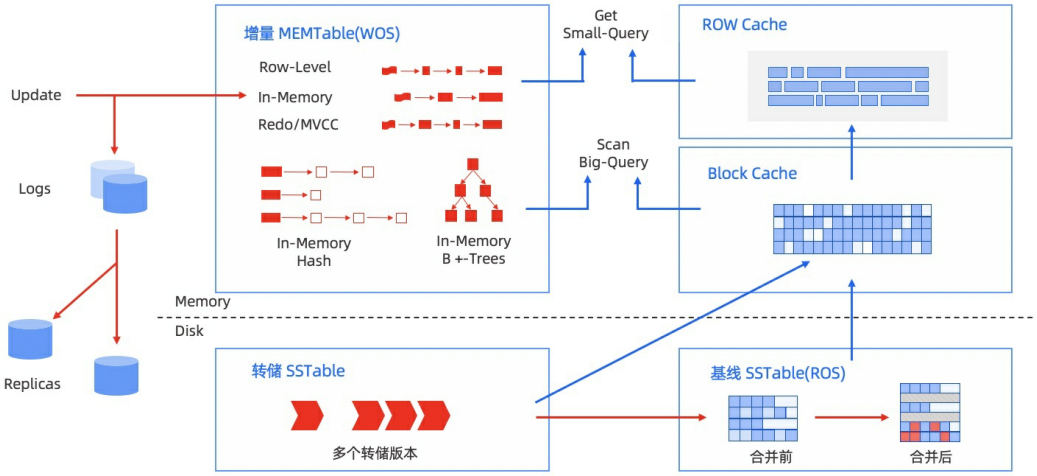
原子性

不论单调读还是非单调读，对单个分区同一个事务修改的数据，均能保证原子性。不论何种场景，只要读快照小于单个分区维护的最大安全可读版本号，读到的数据一定具备原子性，这是原子性保证的原则。

因此设计上，需要引入分区是否可读的校验，对于不可读的分区，让 SQL 执行引擎重试其他副本。

8.存储架构

8.1. 存储架构概述



OceanBase 数据库的存储引擎基于 LSM Tree 架构，将数据分为静态基线数据（放在 SSTable 中）和动态增量数据（放在 MemTable 中）两部分，其中 SSTable 是只读的，一旦生成就不再被修改，存储于磁盘；MemTable 支持读写，存储于内存。数据库 DML 操作插入、更新、删除等首先写入 MemTable，等到 MemTable 达到一定大小时转储到磁盘成为 SSTable。在进行查询时，需要分别对 SSTable 和 MemTable 进行查询，并将查询结果进行归并，返回给 SQL 层归并后的查询结果。同时在内存实现了 Block Cache 和 Row cache，来避免对基线数据的随机读。

当内存的增量数据达到一定规模的时候，会触发增量数据和基线数据的合并，把增量数据落盘。同时每天晚上的空闲时刻，系统也会自动每日合并。

OceanBase 数据库本质上是一个基线加增量的存储引擎，在保持 LSM-Tree 架构优点的同时也借鉴了部分传统关系数据库存储引擎的优点。

传统数据库把数据分成很多页面，OceanBase 数据库也借鉴了传统数据库的思想，把数据文件按照 2MB 为基本粒度切分多一个个宏块，每个宏块内部继续拆分多多个变长的微块；而在合并时数据会基于宏块的粒度进行重用，没有更新的数据宏块不会被重新打开读取，这样能够尽可能减少合并期间的写放大，相较于传统的 LSM-Tree 架构数据库显著降低合并代价。另外，OceanBase 数据库通过轮转合并的机制把正常服务和合并时间错开，使得合并操作对正常用户请求完全没有干扰。

由于 OceanBase 数据库采用基线加增量的设计，一部分数据在基线，一部分在增量，原理上每次查询都是既要读基线，也要读增量。为此，OceanBase 数据库做了很多的优化，尤其是针对单行的优化。OceanBase 数据库内部除了对数据块进行缓存之外，也会对行进行缓存，行缓存会极大加速对单行的查询性能。对于不存在行的“空查”，我们会构建布隆过滤器，并对布隆过滤器进行缓存。OLTP 业务大部分操作为小查询，通过小查询优化，OceanBase 数据库避免了传统数据库解析整个数据块的开销，达到了接近内存数据库的性能。另外，由于基线是只读数据，而且内部采用连续存储的方式，OceanBase 数据库可以采用比较激进的压缩算法，既能做到高压缩比，又不影响查询性能，大大降低了成本。

结合借鉴经典数据库的部分优点，OceanBase 数据库提供了一个更为通用的 LSM-tree 架构的关系型数据库存储引擎，具备以下特性：

- 低成本，利用 LSM-tree 写入数据不再更新的特点，通过自研行列混合编码叠加通用压缩算法，OceanBase 数据库的数据存储压缩率能够相较传统数据库提升 10+ 倍。
- 易使用，不同于其他 LSM-tree 数据库，OceanBase 数据库通过支持活跃事务的落盘保证用户的大事务/长事务的正常运行或回滚，多级合并和转储机制来帮助用户在性能和空间上找到更佳的平衡。

- 高性能，对于常见的点查，OceanBase 数据库提供了多级 cache 加速来保证极低的响应延时，而对于范围扫描，存储引擎能够利用数据编码特征支持查询过滤条件的计算下压，并提供原生的向量化支持。
- 高可靠，除了全链路的数据检验之外，利用原生分布式的优势，OceanBase 数据库还会在全局合并时通过多副本比对以及主表和索引表比对的校验来保证用户数据正确性，同时提供后台线程定期扫描规避静默错误。

存储引擎的功能

从功能模块划分上，OceanBase 数据库存储引擎可以大致分为以下几个部分。

数据存储

- 数据组织

  和其他 LSM-tree 数据库一样，OceanBase 数据库也将数据分为内存增量数据（MemTable）和存储静态数据（SSTable）两个层次，其中 SSTable 是只读的，一旦生成就不再被修改，存储于磁盘；MEMTable 支持读写，存储于内存。数据库 DML 操作插入、更新、删除等首先写入 MEMTable，等到 MEMTable 达到一定大小时转储到磁盘成为 SSTable。

  另外在 OceanBase 数据库内，SSTable 会继续细分为 Mini SSTable、Minor SSTable、Major SSTable 三类，MEMTable 转储后形成的我们称为 Mini SSTable，多个 Mini SSTable 会定期 compact 成为 Minor SSTable，而当 OceanBase 数据库特有的每日合并开始后，每个分区所有的 Mini SSTable 和 Minor SSTable 会整体合并为 Major SSTable。

- 存储结构

  在 OceanBase 数据库中，每个分区的基本存储单元是一个个的 SSTable，而所有存储的基本粒度是宏块，数据库启动时，会将整个数据文件按照 2MB 定长大小切分为一个个宏块，每个 SSTable 实质就是多个宏块的集合。

  每个宏块内部又会继续切分为多个微块，微块的概念和传统数据库的 page/block 概念比较类似，但是借助 LSM-Tree 的特性，OceanBase 数据库的微块是做过压缩变长的，微块的压缩前大小可以通过建表的时候指定 `block_size` 来确定。

  而微块根据用户指定存储格式可以分别以 encoding 格式或者 flat 格式存储，encoding 格式的微块，内部数据会以行列混合模式存储；对于 flat 格式的微块，所有数据行则是平铺存储。

- 压缩编码

  OceanBase 数据库对于微块内的数据会根据用户表指定的模式分别进行编码和压缩。当用户表打开 encoding 时，每个微块内的数据会按照列维度分别进行列内的编码，编码规则包括字典/游程/常量/差值等，每一列压缩结束后，还会进一步对多列进行列间等值/子串等规则编码。编码不仅能帮助用户对数据进行大幅压缩，同时提炼的列内特征信息还能进一步加速后续的查询速度。

  在编码压缩之后，OceanBase 数据库还支持进一步对微块数据使用用户指定的通用压缩算法进行无损压缩，进一步提升数据压缩率。

转储合并

- 转储

  OceanBase 数据库中的转储即 Minor Compaction 概念可以理解和其他 LSM-tree 架构数据库的 Compaction 概念类似，主要负责 MEMTable 刷盘转成 SSTable 以及多个 SSTable 之间的 Compaction 策略选择以及动作。OceanBase 数据库中采用的是 leveled 结合 size tired 的 Compaction 策略，大致可以分为三层，其中 L1 和 L2 就是固定的 leveled 层次，L0 层是 size tired，L0 内部还会继续根据写放大系数以及 SSTable 个数进行内部 Compaction 动作。

- 合并

合并也就是 Major Compaction，在 OceanBase 数据库中也叫每日合并，概念和其他 LSM-Tree 数据库稍有不同。顾名思义，这个概念诞生之初是希望这个动作放到每天凌晨 2 点左右整个集群做一次整体的 Compaction 动作。合并一般是由 RS 根据写入状态或者用户设置发起调度，每次合并都会选取一个全局的快照点，集群内所有的分区都会用这个快照点的数据做一次 Major Compaction，这样每次合并集群所有的数据都基于这个统一的快照点生成相应的 SSTable，通过这个机制不仅能帮助用户定期整合增量数据，提升读取性能，同时还提供了一个天然的数据校验点，通过全局的一致位点，OceanBase 数据库能够在内部对多副本以及主表索引表进行多维度的物理数据校验。

查询读写

- 插入

  在 OceanBase 数据库中，所有的数据表都可以看成索引聚簇表，即使是无主键堆表，在内部也会为其维护一个隐藏主键。因此当用户插入数据时，在向 MEMTable 中写入新的用户数据前，需要先检查当前数据表中是否已经存在相同数据主键数据，为了加速这个重复主键查询性能，对于每个 SSTable 会由后台线程针对不同宏块判重频率来异步调度构建 Bloomfilter。

- 更新

  作为 LSM-Tree 数据库，OceanBase 数据库中的每次更新同样会插入一行新数据，和 Clog 不同， 在 MEMTable 中更新写入的数据只包含更新列的新值以及对应的主键列，即更新行并不一定包含表全部列的数据，在不断的后台 Compaction 动作中，这些增量更新会不断的融合在一起加速用户查询

- 删除

  和更新类似，删除操作同样不是直接作用在原数据上，而是使用删除行的主键写入一行数据，通过行头标记来标明删除动作。大量的删除动作对于 LSM-Tree 数据库都是不友好的，这样会导致即使一个数据范围被完全删除后，数据库还是需要迭代这个范围内所有删除标记行，再做完熔合后才能确认删除状态。针对这个场景，OceanBase 数据库提供了内在的范围删除标记逻辑来规避这种场景，另外也支持让用户显示指定表模式，这样可以通过特殊的转储合并方式来提前回收这些删除行加速查询。

- 查询

  由于增量更新的策略，查询每一行数据的时候需要根据版本从新到旧遍历所有的 MEMTable 以及 SSTable，将每个 Table 中对应主键的数据熔合在一起返回。数据访问过程中会根据需要利用 Cache 加速，同时针对大查询场景，SQL 层会下压过滤条件到存储层，利用存数据特征进行底层的快速过滤，并支。向量化场景的批量计算和结果返回。

- 多级缓存

  为提升性能，OceanBase 数据库支持了多级的缓存系统，对于查询提供针对数据微块的 Block Cache，针对每个 SSTable 的 Row Cache，针对查询熔合结果的 Fuse Rrow Cache，针对插入判空检查的 bloomfilter cache 等，同一个租户下的所有缓存共享内存，当 MEMTable 写入速度过快时，可以灵活的从当前各种缓存对象中挤占内存给写入使用。

数据校验

作为金融级关系数据库，OceanBase 数据库一直将数据质量和安全放在第一位，全数据链路每一个涉及持久化的数据部分都会增加数据校验保护，同时利用多副本存储的内在优势，还会增加副本间的数据校验进一步验证整体数据一致。

- 逻辑校验

  在常见部署模式下，OceanBase 数据库的每个用户表在集群中都会存在多副本，在集群每日合并时， 所有的副本都会基于全局统一的快照版本生成一致的基线数据，利用这个特性，所有副本的数据会在合并完成时比对数据的校验和，保证完全一致。更进一步，基于用户表的索引，还会继续比对索引列的校验和，确保最后返回用户的数据不会因为程序内在问题出错。

- 物理校验

针对数据存储，OceanBase 数据库从数据存储最小 I/O 粒度微块开始，在每个微块/宏块/SSTable/ 分区上都记录了相应的校验和，每次数据读取时都会进行数据校验；为了防止底层存储硬件问题，在转储合并写入数据宏块时也会在写入后马上重新数据进行校验；最后每个 Server 后台会有定期的数据巡检线程对整体数据扫描校验，以提前发现磁盘静默错误。

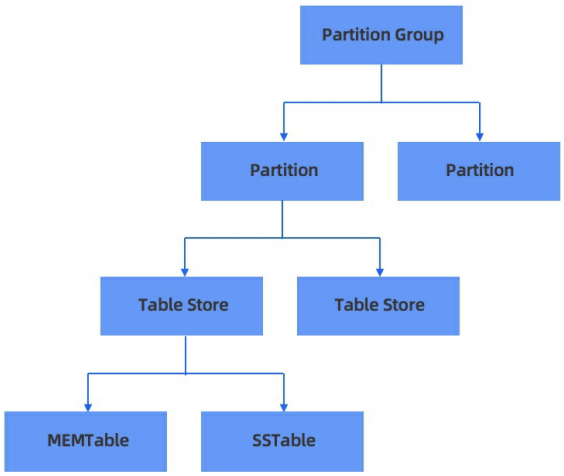
8.2. 数据存储

8.2.1. 数据存储概述

在 OceanBase 数据库的存储视角看，存储结构里最上层是 Partition Group，对应一个分区组，很多时候我们也将其简称为 PG。PG 是一个为了取得极限性能而抽象出来的一个概念。我们知道在一个用户的事务中，可能会操作很多张不同的表，在 OceanBase 数据库的分布式架构下，很难保证这些不同的表在相同的服务器上，那么这就势必会带来分布式事务，而分布式事务是依赖两阶段提交的，会有更大的开销；如果这些不同的表都在相同的服务器上，我们就有可能对这个事务做一阶段优化，以取得更好的性能。但大多数情况下，对表的位置其实是没有办法保证的。对于互联网下的很多应用，我们发现业务都会根据 User Id 做表的分区，并且多张表的分区规则都是相同的，对于这些表我们提供了语法来构建 Table Group，对于 Table Group 中的相应分区我们称之为 Partition Group，OceanBase 数据库会保证同一个 Partition Group 中的多个 Partition 始终绑定在一起，那么对于同一个 Partition Group 的事务操作就会被优化为单机事务，以取得更好的性能。

在一个 Partition Group 中可能包含多个 Partition，注意这些 Partition 的分区键和分区规则要完全相同。Partition Group 是 OceanBase 数据库的 Leader 选举和迁移复制的最小单位。Partition 就是表的一个分区，和 Oracle/MySQL 对于分区的定义基本相同。表的分区规则可能有很多种，例如 Hash 分区、Range 分区、List 分区甚至二级分区等等，但对于存储层来说，并不关心以上分区规则，都一视同仁为 Partition。

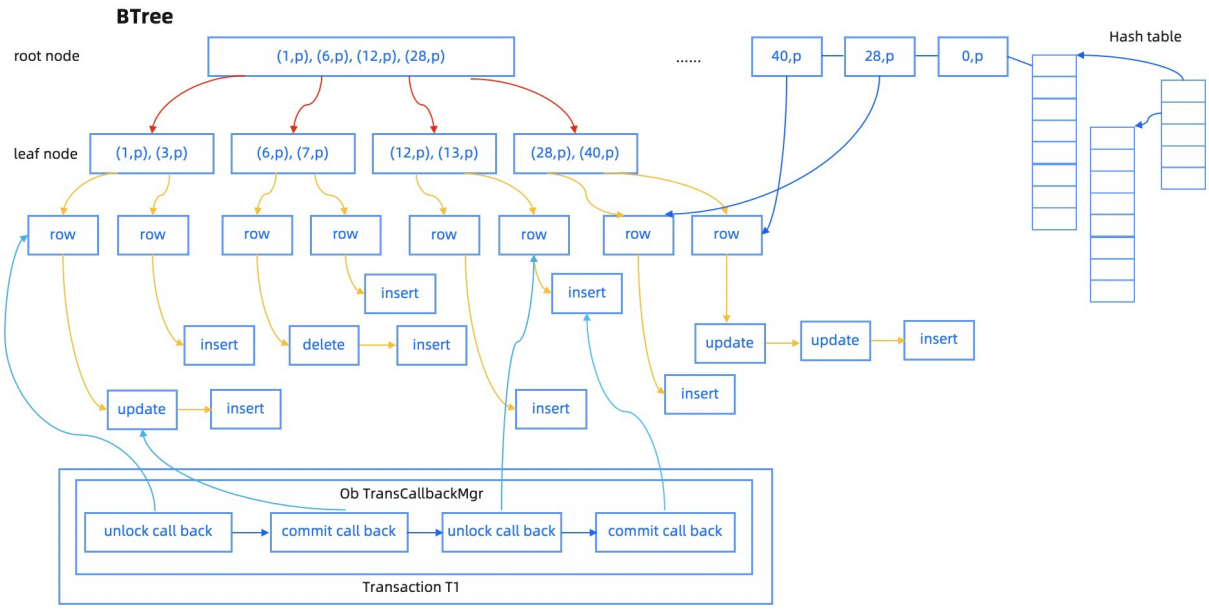
在 OceanBase 数据库中，对于用户表支持创建局部索引，局部索引的特征就是在存储上会和主表绑定在同一个 partiton 内部存储，主表和每个索引在内部会独立存储在一个 Table Store 内，在每一个Table Store 中会包含多个 SSTable 和 MEMTable。MEMTable 存储于内存，存储动态数据，提供读写操作；SSTable 存储于磁盘，存储静态数据并且只读。



8.2.2. MEMTable

MEMTable 中的数据结构

OceanBase 数据库的内存存储引擎 MEMTable 由 BTree 和 Hashtable 组成，在插入/更新/删除数据时，数据被写入内存块，在 HashTable 和 BTree 中存储的均为指向对应数据的指针。

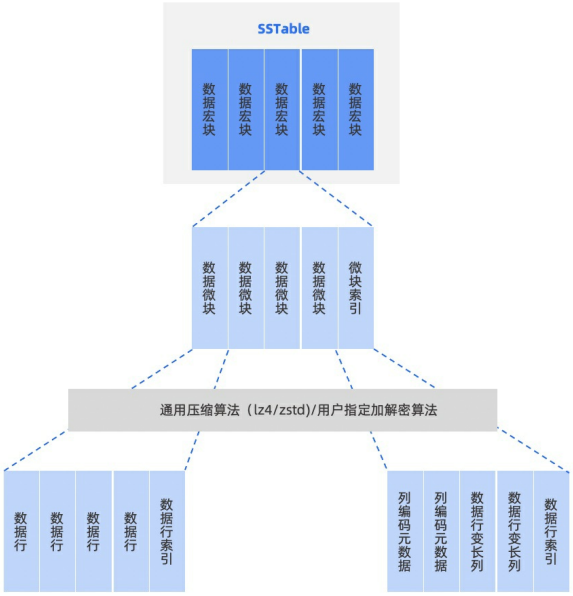


两种数据结构的特点

数据结构	优点	缺点
HashTable	插入一行数据的时候，需要先检查此行数据是否已经存在，当且仅当数据不存在时才能插入，检查冲突时，用 Hashtable 要比 BTree 快。 事务在插入或更新一行数据的时候，需要找到此行并对其进行上锁，防止其它事务修改此行，OceanBase 数据库的行锁放在行头数据结构中，需要先找到它，才能上锁。	不适合对范围查询使用 HashTable。
BTree	范围查找时，由于 BTree 中的数据都是有序的，因此只需要搜索局部的数据就可以了。	单行的查找，也需要进行大量的主键比较，从根结点找到叶子结点，而主键比较性能是较差的，因此理论上性能比 HashTable 慢很多。

8.2.3. SSTable

在 OceanBase 数据库中，对于用户表每个分区管理数据的基本单元就是 SSTable，当 MEMTable 的大小达到某个阈值后，OceanBase 数据库会将 MEMTable 冻结，然后将其中的数据转存于磁盘上，转储后的结构就称之为 SSTable 或者是 Minor SSTable。当集群发生全局合并时，每个用户表分区所有的 Minor SSTable 会根据合并快照点一起参与做 Major Compaction，最后会生成 Major SSTable。每个 SSTable 的构造方式类似，都是由自身的元数据信息和一系列的数据宏块组成，每个数据宏块内部则可以继续划分为多个微块，根据用户表模式定义的不同，微块可以选择使用平铺模式或者编码格式进行数据行的组织。



- **宏块**

  OceanBase 数据库将磁盘切分为大小为 2MB 的定长数据块，称之为宏块（Macro Block），宏块是数据文件写 IO 的基本单位，每个 SSTable 就由若干个宏块构成, 宏块2M固定大小的长度不可更改, 后续转储合并重用宏块以及复制迁移等任务都会以宏块为最基本粒度。

- **微块**

  在宏块内部数据被组织为多个大小为 16KB 左右的变长数据块，称之为微块（Micro Block），微块中包含若干数据行（Row），微块是数据文件读 IO 的最小单位。每个数据微块在构建时都会根据用户指定的压缩算法进行压缩，因此宏块上存储的实际是压缩后的数据微块，当数据微块从磁盘读取时，会在后台进行解压并将解压后的数据放入数据块缓存中。每个数据微块的大小在用户创建表时可以指定，默认 16KB，用户可以通过语句指定微块长度，但是不能超过宏块大小，语句如下。

  ```
  ALTER TABLE mytest SET block_size = 131072;
  ```

  一般来说微块长度越大，数据的压缩比会越高，但相应的一次 IO 读的代价也会越大；微块长度越小，数据的压缩比会相应降低，但相应的一次随机 IO 读的代价会更小。另外根据用户表模式的不同，每个微块构建的时候可能以平铺模式（Flat）或编码模式（Encoding）分别进行构建。在目前版本中，只有基线数据可以指定使用编码模式组织微块，对于转储数据全部默认使用平铺模式进行数据组织。

8.2.4. 压缩与编码

数据库系统中普遍都会对数据进行不同程度的压缩来减小存储成本，一些面向 AP 的列存数据库的按列编码压缩还能够提高某些查询的性能。但对于大部分压缩算法，越高的压缩率也就意味着更复杂的计算和更慢的压缩/解压速度。在传统的 B 树存储结构下的数据库中，数据压缩可能会给数据写入带来 CPU 的计算压力，影响写入性能。但 OceanBase 数据库的 LSM-Tree 架构使数据的压缩只发生在 Compaction 阶段，不会影响数据的写入，同时也使能够使用压缩率更高的压缩方法，在一些客户的应用场景中也证明了 OceanBase 数据库在压缩能力上的优势。

在 OceanBase 数据库中，当 MEMTable 占用内存空间大小达到一定阈值或每日合并时会触发转储/合并，MEMTable 中的数据落盘并合并成静态的 SSTable 数据。相对于 MEMTable，SSTable 中的数据量会更大，冷数据也相对更多。在合并不断产生新的 SSTable的过程中 OceanBase 数据库 会对 SSTable 中的数据进行压缩和编码，来节省数据在硬盘上的存储空间，同时减小对 SSTable 进行查询时产生的 IO。在 SSTable 中，数据是以块为单位来组织的，2Mb 定长的宏块方便对存储空间进行管理，宏块内部变长的微块则方便我们对数据进行压缩。微块有两种存储格式，flat 微块（未编码）和 encoding 微块（编码），OceanBase 数据库的数据压缩与编码都是在微块的粒度上进行的。对于一个 encoding 格式的微块，需要经历 填充行 > 编码 > 通用压缩（可选）> 加密（可选）的流程后完成构建，成为最后落盘的数据块，并写入到定长的宏块中。这个流程中的编码和通用压缩就是 OceanBase 数据库对数据进行压缩的两种方式。

通用压缩

通用压缩指的是压缩算法对数据内部的结构没有了解的情况下，直接对数据块进行压缩。这种压缩往往是根据二进制数据的特征进行编码来减少存储数据的冗余，并且压缩后的数据不能随机访问，压缩和解压都要以整个数据块为单位进行。对数据块 OceanBase 数据库支持 **zlib**、**snappy**、**lz4** 和 **zstd** 四种压缩算法。zstd、lz4 的压缩等级是 1，zlib 压缩等级为 6，snappy 使用默认压缩等级。在 OceanBase 数据库内部对默认 16kb 大小的微块进行压缩的测试中，snappy 和 lz4 压缩速度都比较快，但压缩率比较低，zlib 和 zstd 压缩率比较高但是压缩速度要更慢一些。lz4 和 snappy 的压缩率相似但 lz4 压缩解压的速度会更快一些，同样 zstd 压缩率与 zlib 相似但压缩解压速度都更快。在 MySQL 模式下，支持用户指定单独选择上述的压缩算法，在 Oracle 模式下，兼容 Oracle 的压缩选项，只支持用户选择 lz4 或 zstd 压缩算法。

数据编码（Encoding）

在通用压缩的基础上，OceanBase 数据库自研了一套对数据库进行行列混存编码的压缩方法（encoding）。和通用压缩不同，encoding 的基础建立在压缩算法感知数据块内部数据的格式和语义的基础上。OceanBase 数据库是一个关系型数据库，数据是以表的形式来组织的，表中的每一列都有固定的类型，这就保证了同一列数据在逻辑上存在着一定的相似性；而且在一些场景下，业务的表中相邻的行之间数据也可能会更相似，所以如果将数据按列进行压缩并存储在一起就可以带来更好的压缩效果。因此，OceanBase 数据库引入了一种 encoding 格式的微块，与所有数据逐行序列化到块中的 flat 格式的微块不同，encoding 格式的微块是行列混存的，逻辑上仍然是一组行的数据存在微块里，但微块会按列对数据进行编码，编码后的定长数据存储在微块内部的列存区，部分变长数据还是按行存储在变长区。而且在 encoding 微块中，数据是可以随机访问的，当需要读微块中的一行数据时，可以只对这一行数据进行解码，避免了部分解压算法读一部分数据要解压整个数据块的计算放大；在向量化执行的过程中也可以对指定的列进行解码，降低了投影的开销。

OceanBase 数据库提供了多种按列进行压缩的编码格式，包括列存数据库中常见的字典编码，游程编码（Run-Length Encoding），整形差值编码（Delta Encoding）等。当存储的列是一个定长的数值，如 timestamp、bigint 等时，且这个微块中的数据都分布在一个值域内，整形差值编码会带来比较好的压缩效果，通过只存储每一行的值与微块中最小值的差值，然后做 bit-packing 来减少实际存储的数据量。当微块内的数据的基数（Cardinality）比较小时，字典编码和RLE编码能通过在微块内部构建字典，存储每行的引用来进行压缩。更极端的情况下，一个微块内的一列可能基本都是相同的数据，这时 OceanBase 数据库会通过常量编码（Const）只存储常量和微块内不等于常量的值，进一步提高压缩率。

除了这些比较常见的编码外，OceanBase 数据库还对字符串设计了一些编码格式。包括当一列数据中有着相似的前缀时使用前缀编码（prefix encoding），存储前缀和每行的尾缀。当一列数据是定长的字符串，并且其中几个字节相同时，可以使用定长字符串差值编码（string diff encoding），存储模式串和每行的差值数据。当微块中一列字符串数据的字符基数小于 16 时，可以使用一个十六进制数来表示这个字符，进行十六进制编码（Hex encoding）。这些字符串相关的编码对于较长的业务 ID，带格式的字符串数据等有着很好的压缩效果。

在业务存储的表中，除了同一列数据之间存在相似性，不同列的数据之间也可能有一定的关系。因此ob引入了列间编码(span-column Encoding)，当两列数据大部分相同时，使用列间等值编码(Column equal encoding)，这样一整列都是另外一列的引用。当一列数据是另外一列数据的前缀时，也可以使用列间子串编码(Column prefix encoding)，只存储完整的一列和一列的后缀。这种列间编码可以降低在数据表设计上导致的一些数据冗余，对于一些重复的时间戳，复合列等有比较好的压缩效果，能够整体提高宏块的压缩率。但是这种列间编码在编码和解码时都会更复杂，编码时需要对不同列数据是否符合编码规则进行探测，解码时需要根据引用访问被引用的列的数据，进行处理后解码出数据，相对于其他的编码对cpu不友好一些。同时在有些情况下可能出现不同列之间可以级联引用的情况，对这中情况需要进行特殊处理。

在每列自己编码的基础上，OceanBase 数据库还支持对一列数据采用多种编码方式进行压缩，比如 hex 编码可以与其他的字符串编码叠加，但相应地也会让编码解码变得更复杂。对于 null 值的存储，不同的编码方式，列存行存都有有一些不同，但大多数都使用了一个 null bitmap，来表示该列对应行的数据是不是 null。OceanBase 数据库支持的编码格式不仅与表的 schema 相关，同时还与在一个微块内的值域等数据本身的特征相关，这也就意味着比较难以通过 DBA 设计表数据模型时指定列编码来实现最好的压缩效果，所以 OceanBase 数据库支持在对合并过程中自适应地探测更合适的编码方式来对数据进行编码来达到更高的压缩率。对 n 列数据 m 种编码进行探测理论上需要 m*n 次编码才能发现对每一列最优的编码方式，引入列间编码后又会变得更复杂，因此在编码选择算法上，OceanBase 数据库也进行了一些优化，来提高合并时数据编码的效率。

在 V3.2 版本后，encoding 又支持了向量化执行和 filter 下压，能够在编码后的数据上根据编码的特征进行过滤，降低了一些 overhead，对于一些编码也能够提高过滤效率，对部分列存的定长数据，也支持使用 AVX2 指令集进行 simd 的加速过滤。同时对于一些微块内部按列存存储的数据，向量化执行中直接按列来进行解码对 cache 和分支预测也都更加友好。

当然，除了这些优势外，encoding 也会带来一些问题，其中最直接的是编码解码带来的额外开销，包括合并过程中带来的 CPU 计算压力，查询逐行迭代时复杂的解码带来的 CPU 开销和解码器本身的开销等。OceanBase 数据库在这些问题上进行了一些优化，包括将解码器和对应的数据一起 cache 在内存中等。但是对于一些复杂的编码格式，解码本身带来的额外性能开销也是无法避免的。OceanBase 数据库也会一直尝试在查询性能，内存占用，存储成本上进行更多的权衡。

修改压缩选项

OceanBase 数据库支持通过 DDL 来对表级别的压缩/编码方式来进行配置。但对已经生成了 SSTable 的表进行压缩选项的变更时，为了避免给一次合并带来太大的 IO 写入压力，需要通过渐进合并的方式逐渐重写全部的微块数据，来完成压缩选项的变更。渐进合并的轮次可以通过表级配置项 `progressive_merge_num` 来设置。

- 建表时指定压缩选项
  - MySQL 模式：

    ```
    create table xxx row_format = $value compression = $value;
    ```

  - Oracle 模式：

    ```
    create table xxx $value;
    ```

- 修改一个表的压缩选项
  - MySQL 模式：

    ```
    alter table xxx [set] row_format = $value compression = $value;
    ```

- Oracle 模式：

```
alter table xxx [move] $value;
```

MySQL 模式中 `compression` 的选项取值如下：

- none
- lz4_1.0
- snappy_1.0
- zlib_1.0
- zstd_1.0
- zstd_1.3.8
- lz4_1.9.1

MySQL 模式中 `row_format` 的选项取值如下表所示。

值	微块格式
redundant	flat
compact	flat
dynamic	encoding
compressed	encoding
condensed	selective encoding (encoding 的子集，只使用查询更友好的编码方式)

Oracle 模式中的取值及其说明如下表所示。

值	通用压缩	微块格式
nocompress	none	flat
compress basic	lz4_1.0	flat
compress for oltp	zstd_1.3.8	flat

值	通用压缩	微块格式
compress for query	lz4_1.0	encoding
compress for archive	zstd_1.3.8	encoding
compress for query low	lz4_1.0	selective encoding (encoding 的子集，只使用查询更友好的编码方式)

8.3. 转储和合并

8.3.1. 转储和合并概述

OceanBase 数据库的存储引擎基于 LSM-Tree 架构，数据大体上被分为 MemTable 和 SSTable 两部分，当 MemTable 的大小超过一定阈值时，就需要将 MemTable 中的数据转存到 SSTable 中以释放内存，这一过程称之为**转储**。转储会生成新的 SSTable，当转储的次数超过一定阈值时，或者在每天的业务低峰期，系统会将基线 SSTable 与之后转储的增量 SSTable 给合并为一个 SSTable，这一过程称之为**合并**。

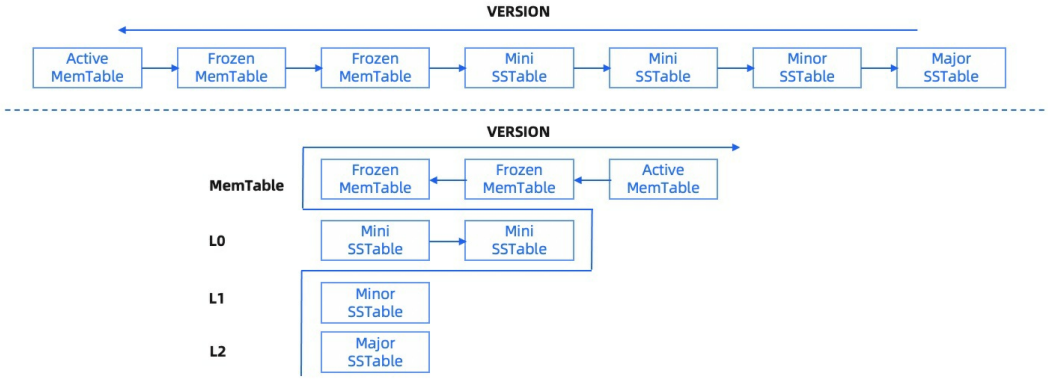
- 有关转储的详细介绍，参见 转储。
- 有关合并的详细介绍，参见 合并。

8.3.2. 转储

OceanBase 数据库的存储引擎基于 LSM-Tree 架构，数据大体上被分为 MemTable 和 SSTable 两部分，当 MemTable 的大小超过一定阈值时，就需要将 MemTable 中的数据转存到 SSTable 中以释放内存，我们将这一过程称之为转储。

分层转储

在 OceanBase 数据库 V2.1 版本之前，在同一时刻只会维护一个转储 SSTable，当 MemTable 需要进行转储时，会将 MemTable 中的数据和转储 SSTable 中的数据进行归并，这会带来一个问题，随着转储次数不断增多，转储 SSTable 的大小也越来越大，而一次转储需要操作的数据量也越来越多，导致转储速度越来越慢，进而导致 MemTable 内存爆。从 V2.2 版本开始，OceanBase 数据库引入了分层转储策略。



参考业界的部分实现，结合目前 OceanBase 数据库架构，OceanBase 数据库的分层转储方案可以理解为常见的 tiered-leveled compaction 变种方案，L0 层是 size-tiered Compaction, 内部继续根据不同场景分裂多层，L1 和 L2 层基于宏块粒度来维持 Leveled compaction。

L0 层

L0 层内部称为 Mini SSTable，根据不同转储策略需要的不同参数设置，L0 层 SSTable 可能存在可以为空。对于 L0 层提供 server 级配置参数来设置 L0 层内部分层数和每层最大 SSTable 个数，L0 层内部即分为level-0 到 level-n 层，每层最大容纳 SSTable 个数相同。当 L0 层 level-n 的 SSTable 到达一定数目上限或阈值后开始整体 compaction，合并成一个 SSTable 写入 level-n+1 层。当 L0 层 max level 内 SSTable 个数达到上限后，开始做 L0 层到 L1 层的整体 compaction 释放空间。在存在 L0 层的转储策略下，冻结 MemTable 直接转储在 L0-level0 写入一个新的 Mini SSTable，L0 层每个 level 内多个 SSTable 根据 base_version 有序，后续本层或跨层合并时需要保持一个原则，参与合并的所有 SSTable 的 version 必须邻接，这样新合并后的 SSTable 之间仍然能维持 version 有序，简化后续读取合并逻辑。

L0 层内部分层会延缓到 L1 的 compaction，更好的降低写放大，但同时会带来读放大，假设共 n 层，每层最多 m 个 SSTable，则最差情况 L0 层会需要持有（n X m+ 2）个 SSTable，因此实际应用中层数和每层 SSTable 上限都需要控制在合理范围。

L1

L1 层内部称为 Minor SSTable，L1 层的 Minor SSTable 仍然维持 rowkey 有序，每当 L0 层 Mini SSTable 达到 compaction 阈值后，L1 层 Minor SSTable 开始参与和 L0 层的 compaction。为了尽可能提升 L1 Compaction 效率, 降低整体写放大, OceanBase 数据库内部提供写放大系数设置, 当 L0 层 Mini SSTable 总大小和 L1 Minor SSTable 大小比率达到指定阈值后, 才开始调度 L1 Compaction, 否则仍位置 L0 层内部 Compaction。

L2

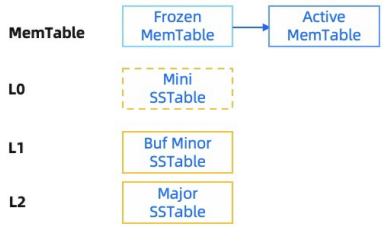
L2 层是基线 Major SSTable，为保持多副本间基线数据完全一致，日常转储过程中 Major SSTable 仍保持只读，不发生实际 compaction 动作。

Queuing 表转储

在 LSM Tree 架构下，对数据行的删除并不是原地删除，而是通过写入一行 Delete 操作（tombstore）来标记删除，如果某个用户业务表主要操作为频繁的插入与删除，从绝对数据量上来看可能并不大，但是由于 LSM Tree 的架构特征，会导致对这类表的增量数据中存在大量的 Delete 标记, 即使用户的实际扫描范围很小，但是 SSTable 内部无法快速识别这些数据是否删除，仍然要扫描大量包含删除标记的数据，导致一个小查询的响应时间可能远远超过用户预期。为了解决这类表的查询性能问题，OceanBase 数据库为用户特别提供了一种支持自定义的表模式，称为 **Queuing 表**（从业务角度有时候也会称为 **Buffer 表**），对其实现了特殊的转储策略。可以通过如下方式指定 queuing 表模式。

```
obclient> ALTER TABLE user_table table_mode = 'queuing';
```

对于 queuing 表具体来说，OceanBase 数据库引入一种自适应的 buffer 表转储策略，由存储层在每次转储时根据转储的统计信息来自主判断是否需要对该表采用 buffer 表转储策略，当发现一个表存在类似 buffer 表行为时，接下来会尝试对这个表做 buffer minor merge 的调度，对这个表基于 Major SSTable 和最新的增量数据以当前的读快照时间生成一个 Buf Minor SSTable，这次 Compaction 动作会消除掉增量数据里的所有 Delete 标记，后续查询基于新生成的 Buf Minor SSTable 就可以避免原有的大量无效扫描动作。



转储触发

转储有两种触发方式：自动触发与手动触发。

当一个租户的 MemTable 内存的使用量达到 `memstore_limit_percentage * freeze_trigger_percentage` 所限制使用的值时，就会自动触发冻结（转储的前置动作），然后系统内部再调度转储。

您也可以通过以下的运维命令手动触发转储。

> **说明**
>
> `memstore_limit_percentage` 用于设置租户使用 memstore 的内存占其总可用内存的百分比。有关该配置项的详细介绍，参见《OceanBase 数据库参考指南》中 **系统配置项** 章节。

```
ALTER SYSTEM MINOR FREEZE [zone] | [server_list] | [tenant_list] | [replica]

tenant_list:
    TENANT [=] (tenant_name_list)

tenant_name_list:
    tenant_name [, tenant_name ...]

replica:
    PARTITION_ID [=] 'partition_idx%partition_count@table_id'

server_list:
    SERVER [=] ip_port_list
```

示例：

- 集群级别转储

```
obclient> ALTER SYSTEM MINOR FREEZE;
```

- Server 级别转储

```
obclient> ALTER SYSTEM MINOR FREEZE SERVER='10.10.10.1:2882';
```

- 租户级别转储

```
obclient> ALTER SYSTEM MINOR FREEZE TENANT='prod_tenant';
```

- Replica 级别转储

```
obclient> ALTER SYSTEM MINOR FREEZE ALTER PARTITION_ID = '8%1@1099511627933';
```

需要注意的是，尽管允许只针对单个分区手动触发 Minor Freeze，但由于多个不同的分区可能共用相同的内存块，因此对单个分区的 Minor Freeze 可能并不能有效地释放内存，而针对租户的 Minor Freeze 可以有效地释放对应租户 MEMTable 的内存。

8.3.3. 合并

与转储相比，合并通常是一个比较重的操作，时间也相对较长，在最佳实践中，一般期望在一天只做一次合并操作，并且控制在业务低峰期进行，因此有时也会把合并称之为每日合并。

合并操作（Major Compaction）是将动静态数据做归并，会比较费时。当转储产生的增量数据积累到一定程度时，通过 Major Freeze 实现大版本的合并。它和转储最大区别在于，合并是集群上所有的分区在一个统一的快照点和全局静态数据进行合并的行为，是一个全局的操作，最终形成一个全局快照。

转储（Minor Compaction）	合并（Major Compaction）
Partition 或者租户级别，只是 MemTable 的物化。	全局级别，产生一个全局快照。
每个 OBServer 的每个租户独立决定自己 MemTable 的冻结操作，主备分区不保持一致。	全局分区一起做 MEMTable 的冻结操作，要求主备 Partition 保持一致，在合并时会对数据进行一致性校验。
可能包含多个不同版本的数据行。	只包含快照点的版本行。
转储只与相同大版本的 Minor SSTable 合并，产生新的 Minor SSTable，所以只包含增量数据，最终被删除的行需要特殊标记。	合并会把当前大版本的 SSTable 和 MemTable 与前一个大版本的全量静态数据进行合并，产生新的全量数据。

合并虽然比较费时，但是同时为数据库提供了一个操作窗口，在这个窗口内 OceanBase 数据库可以利用合并特征完整多个计算密集任务，提升整体资源利用效率。

- **数据压缩**

  合并期间 OceanBase 数据库会对数据进行两层压缩，第一层是数据库内部基于语义的编码压缩，第二层是基于用户指定压缩算法的通用压缩，使用 lz4 等压缩算法对编码后的数据再做一次瘦身。压缩不仅仅节省了存储空间，同时也会极大地提升查询性能。目前 OceanBase 数据库支持（snappy、lz4、lzo，zstd）等压缩算法，允许用户在压缩率和解压缩时间上做各自的权衡。MySQL 和 Oracle 在一定程度上也支持对数据的压缩，但和 OceanBase 相比，由于传统数据库定长页的设计，压缩不可避免的会造成存储的空洞，压缩效率会受影响。而更重要的是，对于 OceanBase 数据库这样的 LSM-tree 架构的存储系统，压缩对数据写入性能是几乎无影响的。

- **数据校验**

通过全局一致快照进行合并能够帮助 OceanBase 数据库很容易的进行多副本的数据一致校验，合并完成后多个副本可以直接比对基线数据来确保业务数据在不同副本间是一致的。另一方面还能基于这个快照基线数据会做主表和索引表的数据校验，保障数据在主表和索引表之间是一致的。

- **统计信息收集**

  排查宏块重用的场景，合并过程需要对每个用户表全表扫描，这个过程中能够顺便完成对每一列以及全表的统计信息收集，并提供给优化器使用

- **Schema 变更**

  对于加列、减列等 Schema 变更，OceanBase 数据库可以在合并中一起完成数据变更操作，DDL 操作对业务来说更加平滑。

合并方式

合并有很多种不同的方式，具体的描述如下。

- 全量合并

  全量合并是 OceanBase 数据库最初的合并算法，和 HBase 与 RocksDB 的 major compaction 过程是类似的。在全量合并过程中，会把当前的静态数据都读取出来，和内存中的动态数据合并后，再写到磁盘上去作为新的静态数据。在这个过程中，会把所有数据都重写一遍。全量合并会极大的耗费磁盘 IO 和空间，除了 DBA 强制指定外，目前 OceanBase 数据库一般不会主动做全量合并。

- 增量合并

  在 OceanBase 数据库的存储引擎中，宏块是 OceanBase 数据库基本的 IO 写入单位，在很多情况下，并不是所有的宏块都会被修改，当一个宏块没有增量修改时，合并可以直接重用这个数据宏块，OceanBase 数据库中将这种合并方式称之为增量合并。增量合并极大地减少了合并的工作量，是OceanBase 数据库目前默认的合并算法。更进一步地，OceanBase 数据库会在宏块内部将数据拆分为更小的微块，很多情况下，也并不是所有的微块都会被修改，可以重用微块而不是重写微块。微块级增量合并进一步减少了合并的时间。

- 渐进合并

  为了支持业务的快速发展，用户不可避免地要做加列、减列、建索引等诸多 DDL 变更。这些 DDL 变更对于数据库来说通长是很昂贵的操作。MySQL 在很长一段时间内都不能支持在线的 DDL 变更（直到 5.6 版本才开始对 Online DDL 有比较好的支持），而即使到今天，对于 DBA 来说，在 MySQL 5.7 中做 Online DDL 仍然是一件比较有风险的操作，因为一个大的 DDL 变更就会导致 MySQL 主备间的 replication lag。

  OceanBase 数据库在设计之初就考虑到了 Online DDL 的需求，目前在 OceanBase 数据库中加列、减列、建索引等 DDL 操作都是不阻塞读写的，也不会影响到多副本间的 paxos 同步。加减列的 DDL 变更是实时生效的，将对存储数据的变更延后到每日合并的时候来做。然而对于某些 DDL 操作如加减列等，是需要将所有数据重写一遍的，如果在一次每日合并过程中完成对所有数据的重写，那么对存储空间和合并时间都会是一个比较大的考验。为了解决这个问题，OceanBase 数据库引入了渐进合并，将 DDL 变更造成的数据重写分散到多次每日合并中去做，假设渐进轮次设置为 60，那么一次合并就只会重写 60 分之一的数据，在 60 轮合并过后，数据就被整体重写了一遍。渐进合并减轻了 DBA 做 DDL 操作的负担，同时也使得 DDL 变更更加平滑。

- 并行合并

  在 OceanBase 数据库 V1.0 中增加了对分区表的支持。对于不同的数据分区，合并是会并行来做的。但是由于数据倾斜，某些分区的数据量可能非常大。尽管增量合并极大减少了合并的数据量，对于一些更新频繁的业务，合并的数据量仍然非常大，为此 OceanBase 数据库引入了分区内并行合并。合并会将数据划分到不同线程中并行做合并，极大地提升了合并速度。

- 轮转合并

一般来说每日合并会在业务低峰期进行，但并不是所有业务都有业务低峰期。在每日合并期间，会消耗比较多的 CPU 和 IO，此时如果有大量业务请求，势必会对业务造成影响。为了规避每日合并对业务的影响。OceanBase 数据库借助多副本分布式架构引入了轮转合并的机制。一般配置下，OceanBase 数据库会同时有 3 个数据副本，当一个数据副本在进行合并时，会将这个副本上的查询流量切到其他没在合并的集群上面，这样业务的查询就不受每日合并的影响。等这个副本合并完成后，再将查询流量切回来，继续做其他副本的合并。为了避免流量切过去后，cache 较冷造成的 rt 波动，在流量切换之前， OceanBase 数据库还会做 cache 的预热。

合并触发

合并触发有三种触发方式：自动触发、定时触发与手动触发。

- 当集群中任一租户的 Minor Freeze 次数超过阈值时，就会自动触发整个集群的合并。
- 也可以通过设置参数来在每天的业务低峰期定时触发合并。

```
obclient> ALTER SYSTEM SET major_freeze_duty_time = '02:00';
```

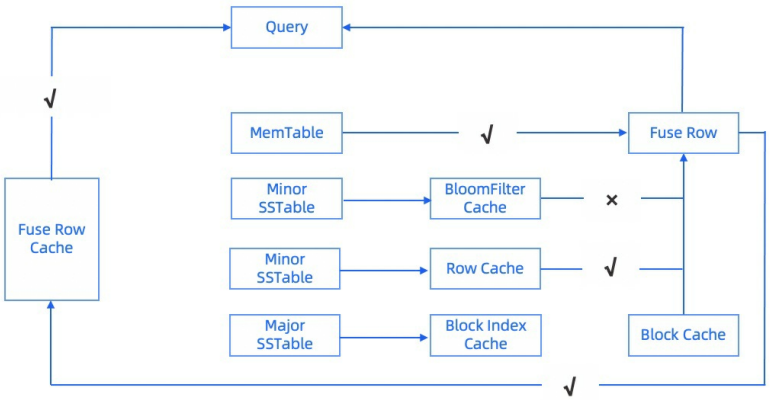
- 也通过以下的运维命令手动触发合并。

```
obclient> ALTER SYSTEM MAJOR FREEZE;
```

8.4. 多级缓存

OceanBase 数据库的 Cache 设计与 Oracle、MySQL 相比有很大的不同。由于 OceanBase 数据库的存储引擎是基于 LSM-tree 架构的，所有的修改只会写入 MemTable，而 SSTable 是只读的，这就意味着 OceanBase 数据库的 Cache 是一个只读 Cache，没有刷脏页的相关逻辑，这相对于传统数据库来说会简单一些；但同时我们对存储在 SSTable 内的数据会进行数据编码和压缩，这意味着我们需要存储的数据是变长而非定长的，和传统数据库相比 Cache 的内存管理又要复杂很多。除此之外，OceanBase 数据库还是一个面向多租户的分布式数据库系统，除了和传统数据库一样要处理 Cache 的内存淘汰之外，在 Cache 内部还需要进行多租户的内存隔离。

和 Oracle 与 MySQL 类似，在 OceanBase 数据库内部，也会有很多种不同类型的 Cache，除了用于缓存 SSTable 数据的 Block Cache（类似于 Oracle 和 MySQL 的 buffer cache）之外，还有 row cache（用于缓存数据行）、log cache（用于缓存 redo log）、location cache（用于缓存数据副本所在的位置）、schema cache（用于缓存表的 Schema 信息）、bloom filter cache（用于缓存静态数据的 bloomfilter，快速过滤空查）等等。OceanBase 数据库设计了一套统一的 Cache 框架，所有不同租户的不同类型的 Cache 都由框架统一管理。对于不同类型的 Cache，会配置不同的优先级，不同类型的 Cache 会根据各自的优先级以及数据访问热度做相互挤占；对于不同租户，会配置对应租户内存使用的上限和下限，不同租户的 Cache 会根据各自租户的内存上下限以及 Server 整体的内存上限做相互挤占。



上图以一个典型的 table get 示例了 OceanBase 数据库中目前服务于查询流程的各种 cache。

- BloomFilter Cache

  OceanBase 数据库的 BloomFilter 是构建在宏块上的，根据用户实际空查率按需自动构建，当一个宏块上的空查次数超过某个阈值时，就会自动构建 BloomFilter，并将 BloomFilter 放入 Cache。

- Row Cache

  针对每个 SSTable 缓存具体的数据行，在进行 Get/MultiGet 查询时，可以将对应查到的数据行放入 Row Cache，这样在下次走到对应行查询时就可以避免多次二分定位对行的查找。

- Block Index Cache

  缓存微块的索引，因为每个 SSTable 虽然以宏块组织，但是 2M 的粒度对于用户查询来说往往粒度太大，因此需要根据用户查询的范围在宏块中定位实际需要的微块，微块索引就是描述对每个宏块中所有的微块的范围，当需要访问某个宏块的微块时，需要提前装载这个宏块的微块索引，因为进行了前缀压缩，因此大小通常较小，并且在 OceanBase 数据库内部给予其较高优先级，因此一般命中率较高。

- Block Cache

  类似于 Oracle 的 Buffer Cache，缓存具体的数据微块，每个微块都会解压后装载到 Block Cache 中，因此每个 cache 大小都是变长的。

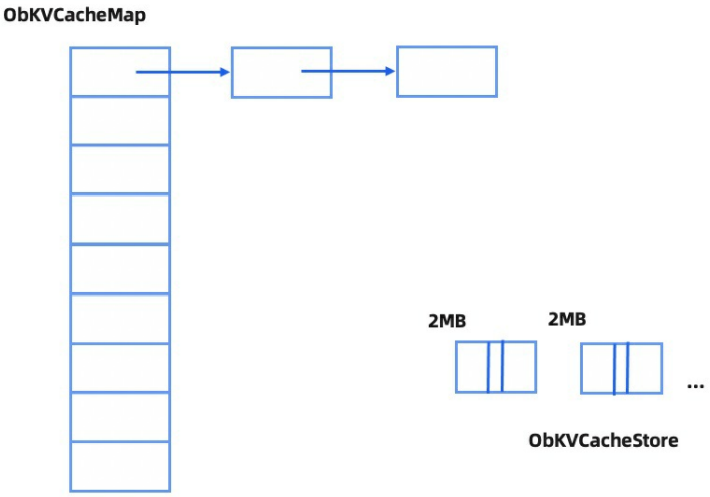
- Fuse Row Cache

  在 LSM-Tree 架构中，同一行的修改可能存在于不同的 SSTable 中，OceanBase 数据库为了进一步优化存储占用，每次用户的更新都只会存储增量数据，因此在查询时需要对各个 SSTable 查询的结果进行熔合，当用户不再触发新的更新时，这个熔合结果对查询都是一直有效的，因此 OceanBase 数据库也提供了对于熔合结果缓存的 Fuse Row Cache，更大幅度支持部分用户的热点行查询。

OceanBase 数据库除了以上用户查询相关的多级缓存之外，还支持多种更多不同类型的 cache，例如：

- Partition Location Cache：用于缓存 Partition 的位置信息，帮助对一个查询进行路由。
- Schema Cache：缓存数据表的元信息，用于执行计划的生成以及后续的查询。
- Clog Cache：缓存 clog 数据，用于加速某些情况下 Paxos 日志的拉取。
- ……

为了能够更加通用的支持这么多种类的 Cache，需要处理变长数据的问题，OceanBase 数据库的底层 Cache 框架将内存划分为多个 2MB 大小的内存块，对内存的申请和释放都以 2MB 为单位进行。变长数据被简单 pack 在 2MB 大小的内存块内。为了支持对数据的快速定位，在 Hashmap 中存储了指向对应数据的指针，整体结构如下图所示。



Cache 内存是以 2MB 为单位整体淘汰的，OceanBase 数据库会根据每个 2MB 内存块上各个元素的访问热度为其计算一个分值，访问越频繁的内存块的分值越高，同时有一个后台线程来定期对所有 2M 内存块的分值做排序，淘汰掉分值较低的内存块。对于一个整体分值不高但是内部存在热点数据的 2MB 内存块，OceanBase 数据库会将其热点数据从它所在的“冷块”移动到“热块”中，避免热点数据被淘汰。在数据结构上并没有维持类似于 Oracle 和 MySQL 的 LRU 链表，因此对于数据读取，OceanBase 数据库的 Cache 访问几乎是无锁的（除了 HashMap 上的 Bucket 锁之外），对于热点数据的高并发访问更加友好。在淘汰时，我们会考虑租户的内存上限及下限，控制各个租户中 Cache 内存的用量。

8.5. 查询处理

OceanBase 数据库目前使用拉取模式的 table_scan 迭代流程，每一行要迭代吐出到 SQL 层实际上是经过了以下几个阶段的：

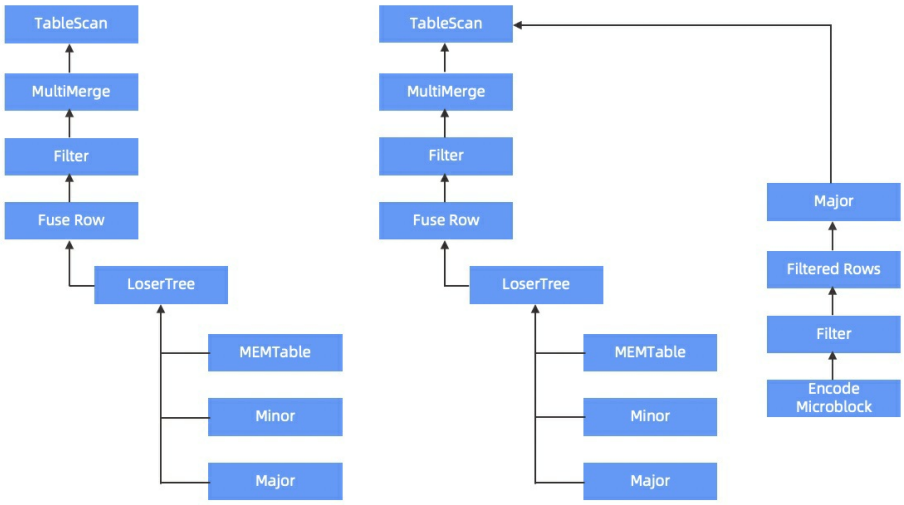
1. 每个 SSTable 通过 query range 定位需要扫描的宏块以及相应的微块，对每个微块需要打开扫描，将需要行整体吐出。
2. memtable/minor sstable/major sstable 多个 iterator 需要分别迭代行。
3. 通过败者树根据多个 iterator 当前行的 rowkey 是否相同来决定需要拿多少行来参与 fuse。
4. fuse 后的最终结果行根据需求投影最终结果，拿到最终结果行。
5. 用最终结果行调用 SQL 的 filter 回调函数进行检查，满足条件继续向上返回。

查询下压

对于 LSM-Tree 架构的存储引擎，在查询的流程一般都会遇到以下几个问题：

- 过多且频繁的迭代以及主键比较，在较多场景尤其是 OLAP 业务中，大部份表很少更新，数据基本都在基线 major SSTable 中，MEMTable 和 minor SSTable 中只存在很少的增量或者更新数据。理想情况是大部分路径应该都能快速的直接从 major SSTable 扫描，只有当主键存在可能交集时，才尝试几方做 fuse，而目前实现中时需要逐行的和转储以及 MEMTable 中的 iterator 判断主键是否相等。这样一方面是效率显著较低，另一方面也基本丧失了为了向量化的扩展可能。
- 过滤算子计算的时间过晚，结合上一点，为了保证最终结果的正确，所有行都必须经过 fuse 之后才能保证数据是最新状态，因此目前的过滤算子的检查都是放在每行迭代的最后进行的。
- 大量无用的投影，从微块扫描吐上来的行需要保证包含最终用户所有需要的列已经投影，经过多次迭代以及 fuse 之后，经过 filter 计算符合条件后可以直接返回，但是如果不符合过滤条件，那么除了 filter 所需列以外的其它投影列都是浪费的。

综上，OceanBase 数据库通过实现过滤算子下压到存储层来解决以上问题，如下图所示：



1. 算子下压，OceanBase 数据库能够快速区分数据没有交集的状态，即能够感知 major SSTable 和增量数据无交集的数据区间，由于该区间的数据只需要访问 major SSTable 即能拿到最新最终状态，因此可以对这段数据直接将过滤算子下压。
2. 算子过滤，每个下压的 filter 表达式进行解析拆分成存储层能够理解的表达式树，里面包含了对应列信息以及相应的过滤条件表达式。另外另外针对过滤条件的复杂度，还能进一步细分为黑盒和白盒两种模式。
   i. 基于 SQL 层的过滤(黑盒)，例如 a * 2 > 3 这种带表达式计算类型的过滤条件，存储层是无法处理的，还是需要回调 SQL 层函数处理。此时算子下压到微块 decoder 层后就无法再进一步下压，需要根据 filter node 信息投影出对应列然后调用 SQL 层回调函数来计算结果。
   ii. 基于存储的过滤(白盒)，例如 a> 1 或者 b = 'abc' 这种简单常见的过滤条件，存储层能够理解语义并处理的 filter node，SQL 层会帮忙进一步解析对应的操作符以及相应的常量表达式，存储层就可以把这个对应的 filter node 继续下压到每一列对应的解码规则中，可以进一步省去投影开销，同时能够充分利用 encoding 信息加速。

8.6. 数据完整性

8.6.1. 发现磁盘的静默错误

磁盘静默错误（Silent Data Corruption）是指存储系统向应用程序提供损坏的数据，而未发出任何警告。例如由于介质损坏，磁盘中出现了坏块，在应用程序读取该块的时候，就会读到错误的数据。

磁盘静默错误发生的概率并不太高，但其可怕之处在于错误发生时没有任何警告，如果应用程序没有对数据的正确性做校验，那就会导致奇怪的应用异常，例如进程崩溃、丢数据等。对数据库这类应用程序而言，不仅系统稳定性非常重要，更是无法容忍数据丢失，数据的正确性和完整性是数据库系统的生命线。

OceanBase 数据库如何防范磁盘静默错误

OceanBase 数据库存放在磁盘中数据包括两部分，第一部分是 RedoLog，即事务日志，RedoLog 经过回放后会构成内存中的 MEMTable。第二部分是 SSTable，即 LSM-Tree 中的静态数据。MEMTable 和 SSTable 共同组成用户数据的一个副本。

多副本机制

OceanBase 数据库作为分布式数据库，采用了多副本的容灾方式。由于磁盘静默错误发生的概率并不高，所以同一个数据块在多个副本同时出现静默错误的概率微乎其微。只要我们能知道某个副本出现了磁盘静默错误，就可以从剩余的正常副本中拷贝数据来修复这个错误。目前 OceanBase 数据库支持副本粒度的修复，出现磁盘静默错误时，运维同学可以先删除错误副本，再从其他机器补齐正确副本。

RedoLog 的校验机制

在每条 RedoLog 的头部，我们会记录这条日志的校验和。在做网络传输和日志回放时，都会强制对每条日志的校验和进行校验。这样我们保证了三副本同步到的日志是正确且一致的，如果一条日志中的数据出现了静默错误，那么这条日志一定不会被同步到其他副本。

SSTable 的校验机制

SSTable 的数据存放在一个个宏块中，宏块的长度固定为 2MB，在宏块的头部会记录这个宏块的校验和。宏块内部会拆分多个微块，微块长度不固定，通常为 16KB，在微块的头部也会记录这个微块的校验和。SSTable 的读 IO 以微块为基本单位，写 IO 以宏块为基本单位。在读取微块时，会强制校验微块头的校验和，保证用户读到的微块数据是正确的。在迁移、备份等复制宏块的场景，目的端写宏块前，也会强制校验宏块的校验和，保证写入的数据是正确的，防止磁盘静默错误的扩散。

除了在读写时检查数据块的正确性，我们还希望尽早发现磁盘静默错误。OceanBase 数据库可以在后台开启巡检任务，周期性扫描全部宏块并检查其校验和，一旦发现磁盘静默错误便会告警。

冷备机制

在没有多副本的部署场景，或者真的多个副本同时发生了磁盘静默错误，那应该怎么拯救数据呢？OceanBase 数据库提供了备份恢复的功能，可以将数据备份到 NFS、OSS 等外部介质，发生磁盘静默错误后，可以从外部介质将正确的数据再恢复到数据库中。

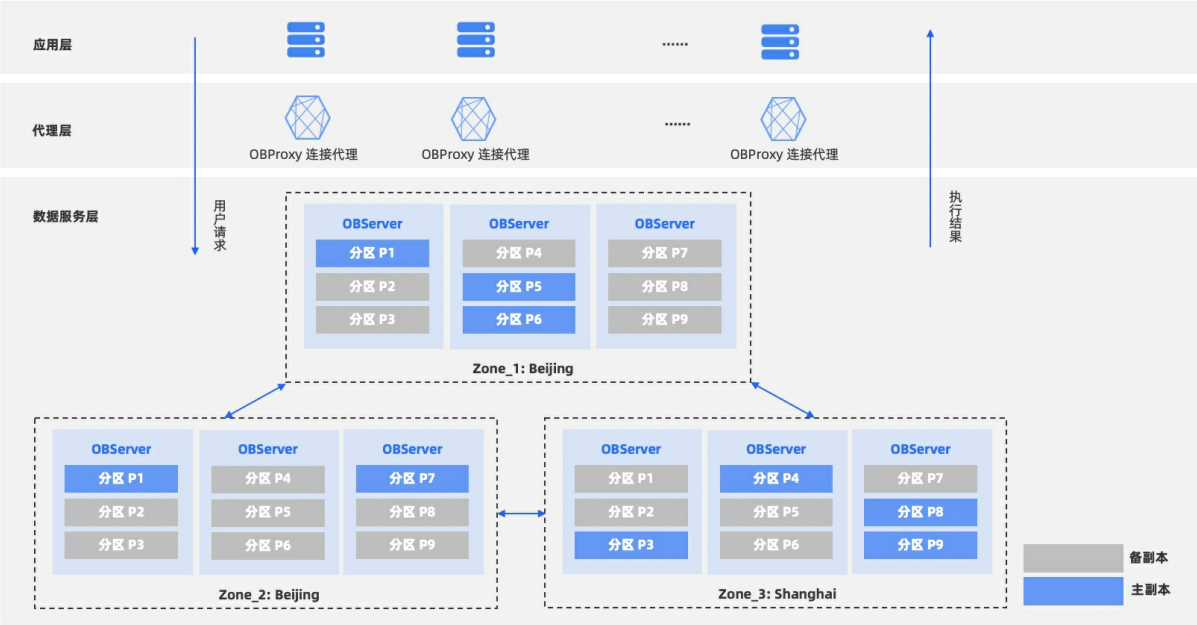
9.数据可靠性和高可用

9.1. 高可用架构

9.1.1. 高可用架构概述

OceanBase 数据库的体系架构和实现方式，保证了数据的强一致性、完整性和高可用。

OceanBase 数据库体系架构



如图所示，数据服务层表示一个 OceanBase 数据库集群。该集群由三个子集群（Zone）组成，一个 Zone 由多台物理机器组成，每台物理机器称之为数据节点（OBServer）。OceanBase 数据库采用 Shared-Nothing 的分布式架构，每个数据节点都是对等的。

OceanBase 数据库中存储的数据分布在一个 Zone 的多个数据节点上，其它 Zone 存放多个数据副本。如图所示的 OceanBase 数据库集群中的数据有三个副本，每个 Zone 存放一份。这三个 Zone 构成一个整体的数据库集群，为用户提供服务。

根据部署方式的不同，OceanBase 数据库可以实现各种级别容灾能力：

- 服务器（Server）级无损容灾：能够容忍单台服务器不可用，自动无损切换。
- 机房（Zone）级无损容灾：能够容忍单个机房不可用，自动无损切换。
- 地区（Region）级无损容灾：能够容忍某个城市整体不可用，自动无损切换。

当数据库集群部署在一个机房的多台服务器时，实现服务器级别容灾。当集群的服务器在一个地区的多个机房中时，能够实现机房级别容灾。当集群的服务器在多个地区的多个机房中时，能够丝线地区级别容灾。

OceanBase 数据库的容灾能力可以达到 RPO=0，RTO=30 秒的国标最高的 6 级标准。

高可用

OceanBase 分布式集群的多台机器同时提供数据库服务，并利用多台机器提供数据库服务高可用的能力。在上图中，应用层将请求发送到代理服务（ODP，也称为 OBProxy），经过代理服务的路由后，发送到实际服务数据的数据库节点（OBServer），请求的结果沿着反向的路径返回给应用层。整个过程中不同的组件通过不同的方式来达到高可用的能力。

在数据库节点（OBServer）组成的集群中，所有的数据以分区为单位存储并提供高可用的服务能力，每个分区有多个副本。一般来说，一个分区的多个副本分散在多个不同的 Zone 里。多个副本中有且只有一个副本接受修改操作，叫做主副本（Leader），其他叫做从副本（Follower）。主从副本之间通过基于 Multi-Paxos 的分布式共识协议实现了副本之间数据的一致性。当主副本所在节点发生故障的时候，一个从节点会被选举为新的主节点并继续提供服务。

选举服务是高可用的基石，分区的多个副本通过选举协议选择其中一个作为主副本（Leader），在集群重新启动时或者主副本出现故障时，都会进行这样的选举。选举服务依赖集群中各台机器时钟的一致性，每台机器之间的时钟误差不能超过 200 毫秒，集群的每台机器应部署 NTP 或其他时钟同步服务以保证时钟一致。选举服务有优先级机制保证选择更优的副本作为主副本，优先级机制会考虑用户指定的 Primary Zone，考虑机器的异常状态等。

当主副本开始服务后，用户的操作会产生新的数据修改，所有的修改都会产生日志，并同步给其他的备副本（Follower）。OceanBase 数据库同步日志信息的协议是 Multi-Paxos 分布式共识协议。Multi-Paxos 协议保证任何需要达成共识的日志信息，在副本列表中的多数派副本持久化成功后即可保证，在任意少数派副本故障时，信息不会丢失。Multi-Paxos 协议同步的多个副本保证了在少数节点故障时系统的两个重要特性：数据不会丢失、服务不会停止。用户写入的数据可以容忍少数节点的故障，同时，在节点故障时，系统总是可以自动选择新的副本作为主副本继续数据库的服务。

OceanBase 数据库每个租户还有一个全局时间戳服务（GTS），为租户内执行的所有事务提供事务的读取快照版本和提交版本，保证全局的事务顺序。如果全局时间戳服务出现异常，租户的事务相关操作都会受到影响。OceanBase 数据库使用与分区副本一致的方案保证全局时间戳服务的可靠性与可用性。租户内的全局时间戳服务实际会由一个特殊的分区来决定其服务的位置，这个特殊分区与其他分区一样也有多副本，并通过选举服务选择一个主副本，主副本所在节点就是全局时间戳服务所在节点。如果这个节点出现故障，特殊分区会选择另一个副本作为主副本继续工作，全局时间戳服务也自动转移到新的主副本所在节点继续提供服务。

以上是数据库集群节点实现高可用的关键组件，代理服务（ODP，也称为 OBProxy）也需要高可用能力来保证其服务。用户请求首先到达的是代理服务，如果代理服务不正常用户请求也无法被正常服务。代理服务还需要处理数据库集群节点故障，并做出响应的容错处理。

代理服务不同于数据库集群，代理服务没有持久化状态，其工作依赖的所有数据库信息都来自于对数据库服务的访问，所以代理服务故障不会导致数据丢失。代理服务也是由多台节点组成集群服务，用户的请求具体会由哪个代理服务节点来执行，应由用户的F5或者其他负载均衡组件负责，同时代理服务的某台节点故障，也应由负载均衡组件自动剔除，保证之后的请求不会再发送到故障节点上。

代理服务工作过程会实时监控数据库集群的状态，一方面代理服务会实时获取集群系统表，通过系统表了解每台机器的健康状态和分区的实时位置，另一方面代理服务会通过网络连接探测数据库集群节点的服务状态，遇到异常时会标记相应节点的故障状态，并进行相应的服务切换。

9.1.2. 代理高可用

OceanBase 数据库代理 ODP（OceanBase Database Proxy，又称 OBProxy）是 OceanBase 专用的代理服务。ODP 自身就有高可用设计。

ODP 部署模式

合并部署

- 以 OCP 方式启动

- 可以访问多个集群。
- 若 OCP 挂掉后，已访问过的集群可以正常访问，未访问过的集群不可访问。

- 以 RSList 方式启动（只可以访问单集群，IP 为 127.0.0.1）
  - 固定访问某个集群，不依赖 OCP。本机有问题后，SLB 探测并切流，无需 ODP 额外动作。
  - 可处理部分机器故障场景。

独立部署

- 以 OCP 方式启动
  - 可以访问多个集群。
  - 若 OCP 挂掉后，已访问过的集群可以正常访问，未访问过的集群不可访问。
- 以 RSList 方式启动
  - 固定访问某个集群，不依赖 OCP。RS List 如果有问题，可能导致集群不可访问。

ODP 容灾能力

发现故障

- 定时任务，刷新 OBServer、Zone、主备集群状态。
  - 方案限制：依赖 OBServer 的状态更新。可能存在 OBServer 出问题（如：磁盘 hang 住），但是没有 inactive 或者 stop server，这个时候状态不会变化。
- Server 连接保持 与 KeepAlive 探活机制，探测 OBProxy 与 后端 OBServer 的连接状况，如果空闲连接异常断开，OBProxy 可以及时发现，剔除并新建。
- Client KeepAlive 探活机制，探测 OBProxy 与 Client 的连接状况，同时可以避免 SLB 等负载均衡的空闲超时。

处理故障

处理进行中的请求
异步中止机制，当检测到机器故障后，及时中止，避免长时间等待，业务连接池被打爆
处理新请求

- 黑名单机制，避免新请求发送到故障机器。
  - OBServer 多次访问失败会加入黑名单。在加入黑名单一段时间后，会发起探测请求，探测该机器是否可访问。

    方案限制：如果客户端先超时断开连接，不会加入黑名单。
  - OBServer 状态变更为非 active，也会加入黑名单。

    方案限制：RTO 依赖 【发现故障-定时任务】 的周期（目前是 20s，可配置） 和 OBServer 的探测时间
- 异步刷新 table locatin cache 机制，检测副本是否切换，如果机器故障后发生了副本切换，可以及时路由到新的副本上

  方案限制：依赖 OBServer 告诉 ODP 路由有问题，如果 OBServer 有问题，一直没返回，table location cache 不会刷新
- 两中心主备库场景，有主备库切换机制，如果主集群发生了切换，可以快速切换到新的主集群

方案限制：RTO 依赖 【发现故障-定时任务】 的周期（目前是 20s，可配置） 和 OBServer 的切换时间

9.1.3. 分布式选举

在分布式系统的设计中，要解决的最主要问题之一就是单点故障问题（single point of failure (SPOF)）。为了能在某个节点宕机后，系统仍然具备正常工作的能力，通常需要对节点部署多个副本，互为主备，通过选举协议在多副本中挑选出主副本，并在主副本发生故障后通过选举协议自动切换至备副本。

主副本称为 Leader，备副本称为 Follower。

在分布式系统中，一个工作良好的选举协议应当符合两点预期：

- 正确性

  即当一个副本认为自己是 Leader 的时候，不应该有其他副本同时也认为自己是 Leader，在集群中同时有两个副本认为自己是 Leader 的情况称为“脑裂”，如 Raft 协议中的选举机制，通过保证为每个 Term 只分配一个 Leader 来避免脑裂，但是原生 Raft 中同一时刻可能有多个副本认为自己是 Leader（尽管它们分管不同的 Term，且更小的 Term 的主副本已经失效但是其不自知），使用原生的 Raft 协议必须读取多数派的内容来保证读取到的数据最新，OceanBase 数据库通过 Lease 机制避免对多数派的访问，确保在任意的时间点上只有一个副本能认为自己是 Leader。

- 活性

  即任意时刻，当 Leader 宕机时，只要集群中仍然有多数派的副本存活，那么在有限的时间内，存活副本中应当有副本能够成为 Leader。

在满足正确性和活性的基础上，OceanBase 数据库的选举协议还提供了优先级机制与切主机制，优先级机制在当前没有 Leader 的情况下在当前可当选 Leader 的多个副本中，选择其中优先级最高的副本称为 Leader；切主机制在当前有 Leader 的情况下可以无缝将 Leader 切换至指定副本。

假设与容错

OceanBase 数据库假定其运行的物理环境满足一定的约束：

- 任意两台机器间的单程网络延迟应当满足小于某个上限值，称为 MAX_TST。
- 集群中所有机器的时钟与 NTP Server 进行同步，且与 NTP Server 的时钟偏差小于给定的最大偏差，称为 MAX_DELAY。

目前 OceanBase 数据库的系统参数为，MAX_TST = 200ms，MAX_DELAY = 100ms。

注意到，若任意机器与 NTP 服务器的时钟偏差不超过 100ms，则任意两台机器间的时钟偏差不超过 200ms。

当时钟偏差和消息延迟的假设不满足时，将发生无主，但是由于安全接收窗口的检查，无论是否满足环境约束，选举协议的正确性都是可以保证的。

优先级

在选举的第一阶段，所有副本都会在同一时刻广播自己优先级，这将使得每个副本都具备足够的信息比较所有副本的优劣，集中投票给同一副本，这一方面避免了选举的分票，另一方面也让选举协议的结果更符合预期，更好预测。

选举优先级是一系列字段的集合，这些字段都有各自的比较规则，这些字段优先级从高到低包括：

1. 成员列表版本号（越大优先级越高）
2. 副本所在的 region 的属性（primary region > 非 primary region）
3. 内存满状态（未满 > 已满）

4. 程序初始化状态（已初始化 > 未初始化）
5. 选举黑名单状态（不在黑名单 > 在黑名单）
6. Rebuild 状态 （不需要 Rebuild > 需要 Rebuild）
7. 数据盘状态 （数据盘无错误 > 数据盘有错误）
8. 租户内存满状态 （租户内存未满 > 租户内存满）
9. CLOG 盘状态 （CLOG 盘无错误 > CLOG 盘有错误）
10. 副本类型（F 副本 > 非 F 副本）
11. CLOG 盘满状态 （CLOG 盘未满 > CLOG 盘满）
12. CLOG 日志数量 (越多优先级越高)

切主

用户可以指定分区 Leader，RS 会给对应的分区发消息以完成 Leader 的变更，这种变更通常是很快的。

用户也可以变更 primary region，RS 将指挥分区将 Leader 切换至新的 primary region 上。

当用户开启 enable_auto_leader_switch 选项后，RS 会在 primary zone 中使用切主的方式平滑的分散 leader 的分布。

9.1.4. 多副本日志同步

日志服务作为关系数据库的基础组件，在 OceanBase 数据库中的重要性主要体现在如下方面：

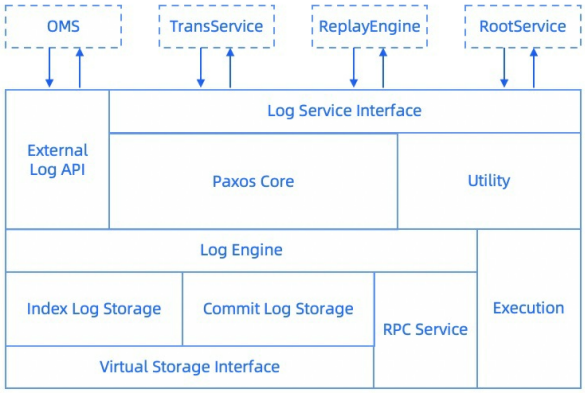
- 通过在事务提交时持久化 Memtable Mutator 的内容以及事务状态信息，为事务的原子性（Atomic）和持久性（Durability）提供支持。
- 通过生成 trans_version，并通过 keepalive 消息同步到所有备机（以及只读副本），为事务的隔离性（Isolation）提供支持。
- 通过实现 Paxos 协议将日志在多数派副本上实现强同步，为分布式数据库的数据容灾以及高可用提供支持；并进而支持各类副本类型（只读副本、日志副本等）。
- 通过维护权威的成员组和 Leader 信息，为 OceanBase 数据库中的各个模块所使用。为 RootService 的负载均衡、轮转合并等复杂策略提供底层机制。
- 提供外部数据服务，为 OMS 和增量备份等外部工具提供增量数据。

OceanBase 数据库日志服务的挑战

与传统关系型数据库的日志模块相比，OceanBase 数据库的日志服务主要有如下几点挑战：

- 通过 Multi-Paxos 替换传统的主备同步机制，进而实现系统的高可用和数据的高可靠。需要支持全球部署，多个副本之间的网络时延达到几十甚至数百 ms。
- 作为分布式数据库，OceanBase 数据库以 Partition 作为数据同步的基本单位。单机要求支持十万量级的 Partition，在此基础上，需要支持日志数据的高效写入和读取。同时，任何功能本身均需要考虑操作的批量化，以加速执行速度。
- 日志服务维护副本的成员列表、Leader 等状态信息，需要为分布式系统的丰富功能提供高效的支持。

整体结构



Paxos Core

Paxos Core 是日志服务的核心。日志服务实现了一套标准的 Multi-Paxos 协议，保证在未发生多数派永久性故障的前提下，所有提交成功的数据均可恢复。

Paxos Core 实现了乱序日志提交，保证事务之间无依赖，同时对异地部署非常友好，避免单个网络链路故障影响后续所有事务的 rt。

为确保 Multi-Paxos 的正确性，日志服务提供了多层次的 checksum 校验：

- 单条日志的 checksum 校验。
- Group Commit 后 Block 整体的 checksum 校验。
- RPC packet 的 checksum 校验。
- log_id 的累积 checksum 校验，每一条日志都校验当前 Partition 之前所有日志的正确性。

正确、高效的协议实现、不间断运行的容灾切换测试用例以及实时完备的 Checksum 校验，一同确保日志服务的正确性。

成员组管理

日志服务维护着每个 Partition 的成员组信息。成员组信息包括当前 Paxos Group 的 member_list 以及 Quorum 信息。

支持的功能如下：

- add_member/remove_member， 支持加减成员。负载均衡发起的迁移、宕机引起的补副本操作，均需要通过上述功能修改 Partition 的成员组。
- modify_quorum，修改 Paxos Group 的法定副本数。执行 Locality 变更（例如， 3F->5F），地域级故障降级容灾（5F->3F），均需要通过上述功能修改 paxos group 的法定副本数。

上述功能均提供了批量操作接口，允许将多个 Partition 的操作批量执行，极大的提升了相关操作的执行效率。

如通过 batch_modify_quorum 接口，单机 1 万个分区 5F->3F 的降级容灾操作执行时间由 20 分钟优化到只需要 20s，极大的提升了系统在故障场景下的容灾能力和响应时间。

有主改选

OceanBase 数据库作为使用 LSM tree 实现的数据库系统，需要周期性的将 Memtable 的内容转储或合并到磁盘，在执行合并时需要消耗大量 CPU。为了避免合并时影响正常的业务请求，也为了加快合并速度，OceanBase 数据库通过轮转合并，在执行合并前将 Leader 切换到同 Region 的另外一个副本。

日志服务为有主改选提供支持。有主改选执行分两步：

1. RS 查询每个待改选 Partition 的有效 candidates。
2. RS 根据 candidates list 以及 Region、Primary Zone 等信息，确定改选目标，下发命令，执行改选请求。

故障恢复

OceanBase 数据库在线上实际运行过程中难免遇到各类故障（磁盘、网络故障，机器异常宕机等）。作为高可用的分布式数据库系统，必须能够在各种故障场景应对自如，保证 RPO = 0，RTO 尽量短。这里按照不同的故障类型分别讲述日志服务在其中所起到的作用：

1. Leader 节点以外的其他节点出现少数派宕机：由于 Paxos 要求日志只需要同步到 majority 即可，对可用性无影响，RPO=0，RTO=0。
2. 包括 Leader 节点在内出现少数派宕机：通过无主选举+paxos 恢复流程，保证在 lease 过期后很短的时间内即选出新的 Leader 提供服务，且数据无任何丢失，RPO = 0，RTO < 30s。
3. Leader 节点以外的其他少数派节点出现网络分区：由于 paxos 要求日志只需要同步到 Majority 即可，对可用性无影响，RPO=0，RTO=0。
4. 包括 Leader 节点在内的少数派节点出现网络分区：通过无主选举+paxos 恢复流程，保证在 lease 过期后很短的时间内即选出新的 Leader 提供服务，且数据无任何丢失，RPO = 0，RTO < 30s。
5. 多数派节点宕机：此时服务中断，需要重启宕机进行副本恢复。
6. 集群全部宕机：此时服务中断，需要重启宕机进行副本恢复。

除上述描述的宕机或网络分区场景外，真实业务的故障场景会更加多样化。OceanBase 数据库除了可以处理上述场景外，对于多种磁盘故障（完全 hang 住或写入缓慢）可以进行自动检测，在 1 分钟内切走 Leader 和备机读流量，保证业务及时恢复。

只读副本、级联同步以及副本类型转换

日志服务通过保证 log_id 和 trans_version 偏序，以及百毫秒级的 Keepalive，用以支持备机和只读副本的弱读功能。

只读副本不是 Paxos Group 的成员，我们通过自适应的结合 Locality 信息的级联算法，自动为只读副本构建上下游关系，实现数据的自动同步。

除只读副本外，日志服务支持日志副本（日志副本有特殊的日志回收策略）以及三种副本类型之间的相互转换。

9.1.5. Paxos 协议

OceanBase 数据库使用 Paxos 的优化 Multi-Paxos 实现多副本数据同步以及集群的高可用。

Paxos一致性协议的原理

有关 Paxos 的原理，可以参考以下链接中的内容。

- Wiki: Paxos (computer science)
- Paxos Made Simple

OceanBase 数据库的 Paxos 协议与选举协议的关系

OceanBase 数据库的 Paxos 实现和选举协议一起构成了一致性协议（日志服务）的实现。两者有一定的相关性，但在实现上又尽量做到减少耦合。

选举协议会首先在多个副本中选出一个 Leader 节点，并通过 Lease 机制保证 Leader 的合法性。日志会基于选出的 Leader 推进 Multi-Paxos 的状态机，做未确认日志的恢复，并在恢复阶段完成后开始提供服务。

如何通过Paxos协议避免脑裂

什么是“脑裂”？

在传统数据库主备同步容灾方案中，系统正常工作时，会存在一个主节点（又称作 Master）对外提供数据读写服务，其余节点作为备节点（又称作 Slave）从主节点以日志的形式同步数据，作为容灾节点备用。

在系统出现异常时，以主节点和备节点之间出现网络分区为例：

主节点和同机房的应用服务器连接正常，仍作为主节点提供数据的读写服务。

作容灾决策的人或自动管控工具，在检查到主节点网络不通的场景下，为了保证服务的连续性，会尝试将备节点提升成主节点提供数据读写服务。

在这种场景下，同时存在两个或多个主节点，应用出现多写，会造成严重的数据正确性问题。这个现象被称作“脑裂”。

如何避免“脑裂”？

OceanBase 数据库基于 Paxos 协议实现了高可用选举和日志同步协议，一方面保证数据安全，另一方面提供了很好的服务连续性保证。

Paxos 协议是基于多数派的协议，简单来说，任何决策的达成均需要多数派节点达成一致。

OceanBase 数据库的高可用选举，保证在任一时刻，只有得到多数派的认可，一个节点才能成为主节点提供读写服务。由于集合中任意两个多数派均会存在交集，保证不会同时选举出两个主节点。

在一个节点当选为主节点后，通过租约（又称作 Lease）机制保证服务的连续性。

- 在少数派的备节点出现故障时，主节点的服务不受任何影响。
- 在主节点故障或网络分区时，多数派的备节点会首先等待租约过期；在租约过期后，原主节点保证不再提供读写服务，此时 OceanBase 数据库会自动从剩余节点集合中选举一个新的主节点继续提供服务。

OceanBase 数据库的日志同步协议，要求待写入的数据在多数派节点持久化成功。以 OceanBase 数据库典型的同城三机房部署为例，任意事务持久化的日志，均需要同步到至少两个机房，事务才会最终提交。

- 在少数派的备节点出现故障时，同样，主节点的服务不受任何影响，数据不会丢失。
- 在主节点故障或网络分区时，余下节点中仍保留有完整的数据；高可用选举会首先选出一个新的主节点，该节点会执行恢复流程，从余下节点中恢复出完整数据，在此之后可以继续提供服务，整个过程是完全自动的。

基于上述分析，OceanBase 数据库通过基于 Paxos 协议实现的高可用选举和日志同步协议，避免了“脑裂”，同时保证了数据安全性和服务连续性。

9.1.6. 节点故障的自动处理

RootService 高可用

在 OceanBase 集群中，RootService 提供集群的各类管理服务，RootService 服务的高可用使用如下的方式实现：RootService 服务使用 Paxos 协议实现高可用，可以通过集群配置，指定 RootService 服务副本数，RootService 的各副本基于 Paxos 协议选举 leader，leader 副本上任后为集群提供 RootService 服务。当 RootService 的当前 leader 发生故障卸任时，其他的 RootService 副本重新选举产生新的 leader，并继续提供 RootService 服务，依次实现 RootService 的高可用。RootService 的各副本不是一个单独的进程，仅是某些 OBServer 上启动的一个服务。

OBServer 状态监测

作为集群的中控服务，RootService 负责集群的 OBServer 管理，各 OBServer 通过心跳数据包（heartbeat）的方式，定期（每 2s）向 RootService 汇报自己的进程状态，RootService 通过监测 OBServer 的心跳数据包，来获取当前 OBServer 进程的工作状态。

OBServer 心跳状态相关配置项

- lease_time：当 RootService 累计超过 lease_time 时间没有收到过某 OBServer 的任意心跳数据包时，RootService 认为该 observer进程短暂断线，RootService 会标记该 OBServer 的心跳状态为 lease_expired。
- server_permanent_offline_time：当 RootService 累计超过 server_permanent_offline_time 时间没有收到过某 OBServer 的任意心跳数据包时，RootService 认为该 observer 进程断线，RootService 会标记该 OBServer 的心跳状态为 permanent_offline。

RootService 对 OBServer 节点故障的处理

RootService 根据心跳数据包可以获得 OBServer 如下的工作状态：

- OBServer 心跳数据包存在，心跳数据包中的 OBServer 磁盘状态正常。此种状态下，RootService 认为 OBServer 处于正常工作状态。
- OBServer 心跳数据包存在，心跳数据包中的 OBServer 磁盘状态异常。此种状态下，RootService 认为 observer 的进程还在，但 OBServer 磁盘故障。此种状态下，RootService 会尝试将该 OBServer 上的全部 leader 副本切走。
- OBServer 心跳数据包不存在，OBServer 心跳数据包的丢失时间还比较短，OBServer 心跳状态为 lease_time，此种状态下，RootService 仅将 OBServer 的工作状态设置为 inactive，不做其他处理。
- OBServer 心跳数据包不存在，OBServer 心跳数据包丢失时间超过 server_permanent_offline_time，OBServer 的心跳状态为 permanent_offline，此种情况下，RootService 会对该 OBServer 上的数据副本进行处理，将该 OBServer 上包含的数据副本从 Paxos 成员组中删除，并在其他可用 OBServer 上补充数据，已保证数据副本 Paxos 成员组完整。

故障机器的恢复

集群中发生故障的 OBServer 存在两种情况：

- 故障 OBServer 可以重新启动。这种情况下，不论故障机器之前处于那种心跳状态，重新启动后，OBServer与 RootService 之间的心跳数据包恢复以后，OBServer 可重新提供服务。
- 故障 OBServer 损坏，无法重新启动。这种情况下，在确认 OBServer 损坏无法重新启动后，需要数据库管理员执行集群管理操作，将该 OBServer 中删除。删除流程可参考 alter system delete server 管理操作。

9.1.7. GTS 高可用

全局时间戳服务（Global Timestamp Service，简称 GTS），OceanBase 数据库内部为每个租户启动一个全局时间戳服务，事务提交时通过本租户的时间戳服务获取事务版本号，保证全局的事务顺序。

GTS 是集群的核心，需要保证高可用。

- 对于用户租户而言，OceanBase 数据库使用租户级别内部表 __all_dummy 表的 leader 作为 GTS 服务提供者，时间来源于该 leader 的本地时钟。GTS 默认是三副本的，其高可用能力跟普通表的能力一样，保证单节点故障场景下 RTO<30s。

GTS 维护了全局递增的时间戳服务，异常场景下依然能够保证正确性：

- **有主改选**

原 Leader 主动发起改选的场景，我们称为有主改选。新 leader 上任之前先获取旧 leader 的最大已经授权的时间戳作为新 leader 时间戳授权的基准值。因此该场景下，GTS 提供的时间戳不会回退。

- **无主选举**

  原 leader 与多数派成员发生网络隔离，等 lease 过期之后，原 follower 会重新选主，这一个过程，我们称为无主选举。选举服务保证了无主选举场景下，新旧 Leader 的 lease 是不重叠的，因此能够保证本地时钟一定大于旧主提供的最大时间戳。因此新 leader 能够保证 GTS 提供的时间戳不回退。

OceanBase 数据库 3.x 及之前的版本 GTS 生成的时间戳并未持久化，依赖于宕机恢复时间 > 最大可能的时钟跳变时间来保证正确性；后续 OceanBase 数据库版本会持久化 GTS 分配的时间戳，解除上述对时钟跳变的依赖。

9.2. 容灾部署方案

OceanBase 数据库提供多种部署模式，可根据对机房配置以及性能和可用性的需求进行灵活选择。

部署方案	容灾能力	RTO	RPO
同机房三副本	机器级无损容灾 / 机架级无损容灾	30s 内	0
同城双机房主备库	机房级容灾	分钟级	大于 0
同城三机房	机房级无损容灾	30s 内	0
两地两中心主备库	地域级容灾	分钟级	大于 0
三地三中心五副本	地域级无损容灾	30s内	0

为了达到不同级别的容灾能力，OceanBase 数据库提供了两种高可用解决方案：多副本高可用解决方案和主备库高可用解决方案。多副本高可用解决方案基于 Paxos 协议实现，在少数派副本不可用情况下，能够自动恢复服务，并且不丢数据，始终保证 RTO 在 30 秒内，RPO 为 0。主备库高可用解决方案是基于传统的主-备架构来实现的高可用方案，是多副本高可用方案的重要补充，可以满足双机房和双地域场景下的容灾需求；它不能保证数据不丢，RPO 大于 0，RTO 为分钟级别。

同机房三副本

如果只有一个机房，可以部署三副本或更多副本，来达到机器级无损容灾。当单台 server 或少数派 server 宕机情况下，不影响业务服务，不丢数据。如果一个机房内有多个机架，可以为每个机架部署一个 Zone，从而达到机架级无损容灾。

同城双机房主备库

如果同城只有双机房，又想达到机房级容灾能力，可以采用主备库，每个机房部署一个集群。当任何一个机房不可用时，另一个机房可以接管业务服务。如果备机房不可用，此时业务数据不受影响，可以持续提供服务；如果主机房不可用，备机房集群需要激活成新主集群，接管业务服务，由于备集群不能保证同步所有数据，因此可能会丢失数据。

同城三机房

如果同城具备三机房条件，可以为每个机房部署一个 Zone，从而达到机房级无损容灾能力。任何一个机房不可用时，可以利用剩下的两个机房继续提供服务，不丢失数据。这种部署架构不依赖主备库，不过不具备地域级容灾能力。

两地两中心主备库

用户希望达到地域级容灾，但是每个地域只有一个机房时，可以采用主备库架构，选择一个地域作为主地域，部署主集群，另一个地域部署备集群。当备地域不可用时，不影响主地域的业务服务；当主地域不可用时，备集群可以激活为新主集群继续提供服务，这种情况下可能会丢失业务数据。

更进一步，用户可以利用两地两中心实现双活，部署两套主备库，两个地域互为主备。这样可以更加高效利用资源，并且达到更高的容灾能力。

三地三中心五副本

为了支持地区级无损容灾，通过 Paxos 协议的原理可以证明，至少需要 3 个地区。OceanBase 数据库采用的是两地三中心的变种方案：三地三中心五副本。该方案包含三个城市，每个城市一个机房，前两个城市的机房各有两个副本，第三个城市的机房只有一个副本。和两地三中心的不同点在于，每次执行事务至少需要同步到两个城市，需要业务容忍异地复制的延时。

三地五中心五副本

和三地三中心五副本类似，不同点在于，三地五中心会把每个副本部署到不同的机房，进一步强化机房容灾能力。

9.3. 主备库

9.3.1. 概述

OceanBase 集群的多副本机制可以提供丰富的容灾能力，在机器级、机房级、城市级故障情况下，可以实现自动切换，并且不丢数据，RPO = 0。OceanBase 数据库的主备库高可用架构是 OceanBase 数据库高可用能力的重要补充。当主集群出现计划内或计划外（多数派副本故障）的不可用情况时，备集群可以接管服务，并且提供无损切换（RPO = 0）和有损切换（RPO>0）两种容灾能力，最大限度降低服务停机时间。

OceanBase 数据库支持创建、维护、管理和监控一个或多个备集群。备集群是生产库数据的热备份。管理员可以选择将资源密集型的报表操作分配到备集群，以便提高系统的性能和资源利用率。

主集群

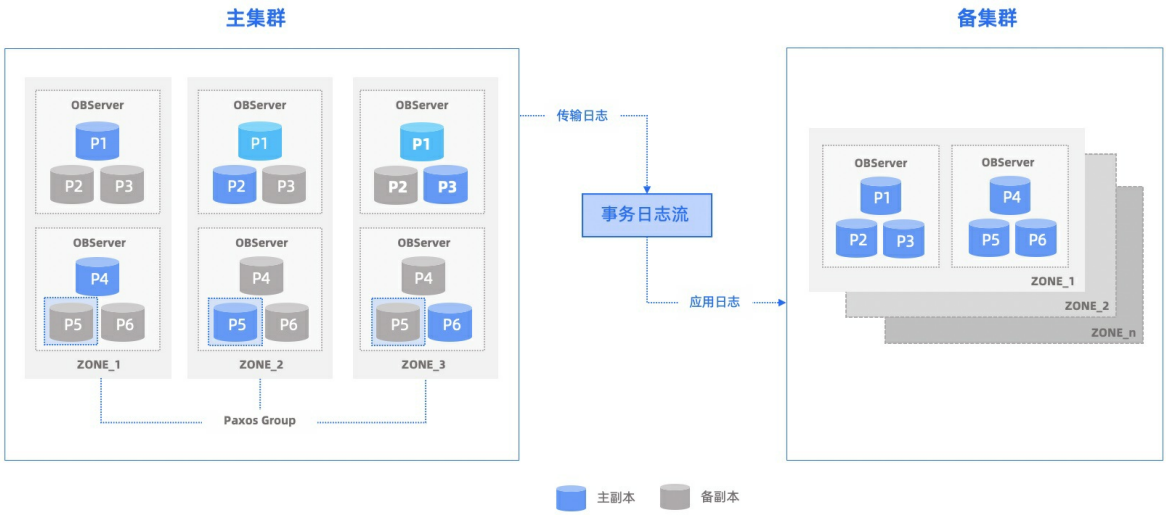
主集群，即生产集群，是唯一接受业务写入和强一致性读的集群，它的角色是 `PRIMARY` 。

备集群

备集群是主集群的一个物理拷贝，保证事务一致性，它的角色是 `PHYSICAL STANDBY` 。主集群会自动向备集群传输 REDO 日志；备集群收到 REDO 日志后会进行持久化，然后执行 REDO 回放操作，恢复用户数据和 schema，保证与主集群是物理一致的。

配置示例

下图展示了一个典型的主备库配置，包含一个主集群和一个备集群，通过传输 REDO 日志的方式进行数据同步。主备集群可以部署在不同的位置，以满足业务不同级别的容灾需求。



9.3.2. 典型应用场景

以机房级容灾和地域级容灾为例，介绍主备库典型的应用场景。

双机房

机房作为独立的容灾单元，用户有机房级容灾的需求。只有两个机房的情况下，用户希望在任何一个机房不可用时，另一个机房可以接管服务。

单集群部署情况下，如果跨机房部署，必然有一个机房中有多数派副本。多数派副本所在机房不可用情况下，主集群会停服务。因此，单集群部署满足不了双机房的容灾需求。

一主一备部署方案可以解决双机房容灾问题。主集群部署在主机房，多副本架构，满足机器级容灾需求。备集群部署在备机房，可以选择单副本，降低部署成本，也可以选择多副本，让备集群也具备机器级容灾能力。

备机房不可用时，不影响主集群服务；主机房不可用时，备集群可以 Failover 成新主集群，接管服务。这样就满足了机房级容灾需求。

两地三中心

两地三中心是指机房分布在两个地域，一个地域有两个机房，一个地域有一个机房。用户期望不仅能达到机房级容灾能力，也能够达到地域级容灾能力，即任何一个地域故障，另一个地域可以接管服务。

OceanBase 数据库有成熟的两地三中心解决方案：两地三中心五副本。举例说明，上海有两个机房，杭州有一个机房，上海每个机房部署两个副本，杭州单机房部署一个副本。任何一个机房不可用时，多数派副本都是存活的，可以实现无损容灾（RTO = 0）。不过如果出现上海地域级故障，多数派副本将不可用，主集群会停服务。因此，两地三中心模式下，单集群部署满足不了地域级容灾需求。

一主一备部署方案可以解决两地三中心地域级容灾问题。杭州机房可以部署一个备集群，单副本或多副本。杭州地域级故障时，不影响主集群服务；上海地域级故障时，备集群可以 Failover 成新主集群，接管服务。这样就满足了地域级容灾需求。

9.3.3. 保护模式

OceanBase 数据库提供了三种保护模式：最大性能，最大保护和最大可用，并且支持三种保护模式之间的切换。

最大性能（Maximum Performance）

这是默认的保护模式。它在最大限度地确保主集群性能的同时，还能保护用户的数据。在这种保护模式下，事务只需要等 REDO 日志在主集群持久化成功就可以立即提交。REDO 日志会异步的同步到备集群，但是不会阻塞主集群事务提交。因此，主集群的性能不会受备集群的同步延时影响。

最大保护（Maximum Protection）

这种保护模式提供了最高级别的数据保护，可以确保主集群出现故障时不会发生数据丢失。在这种保护模式下，事务需要等 REDO 日志在主集群和强同步的备集群上都持久化成功才能提交。

最大保护模式下只支持配置一个强同步的备集群，其他备集群只能处于异步同步模式。如果强同步的备集群不可用，主集群会停止写服务。

最大可用（Maximum Availability）

这种保护模式在不牺牲集群可用性的前提下提供了最高级别的数据保护。默认情况下，事务要等 REDO 日志在主集群和强同步的备集群上都持久化成功才能提交。当感知到强同步的备集群出现故障时，主集群不再等日志强同步成功，和最大性能一样优先恢复主集群服务，保证集群可用性。直到强同步的备集群恢复服务，主集群会自动恢复为强同步模式，提供最高级别的数据保护。

最大可用模式下只支持配置一个强同步的备集群，其他备集群只能处于异步同步模式。

9.3.4. 日志传输服务

OceanBase 数据库日志传输服务负责自动将主集群 REDO 日志传输到备集群。

它提供以下能力：

- 主集群向备集群传输 REDO 日志。
- 备集群落后情况下，自动从主集群拉取 REDO 日志。
- 管理所有备集群同步参数，支持级联传输日志。

OceanBase 数据库支持以下两种传输模式：

- 强同步模式 `SYNC`

  该模式下，主集群的 REDO 日志要强同步到目标备集群。一条 REDO日志要等主集群和 `SYNC` 模式的备集群都持久化成功之后才认为持久化成功。事务的提交时延会增加主备集群的网络延时和备集群持久化日志的时间。

  最大保护和最大可用模式下，主集群仅支持配置一个 `SYNC` 模式的备集群。最大性能模式下，该传输模式不生效，用户可以配置任意数量的备集群为 `SYNC` 模式。

  主集群可以将自己配置为 `SYNC` 模式，只有当自己切换为备集群时才有意义。最大保护和最大可用模式下，执行 switchover 前，要求将主集群配置为 `SYNC` 模式，保证 switchover 之后，仍然存在一个 `SYNC` 模式的备集群。

- 异步同步模式 `ASYNC`

  该模式下，主集群的 REDO 日志会异步同步到目标备集群。事务的提交时延不受目标备集群影响。

日志传输模式描述了主集群日志是否要强同步给目标备集群，OceanBase 数据库支持一主多备，不同的备集群可以配置不同的日志传输模式。不同的保护模式下，日志传输模式也有不同的含义。下表列出了所有可能出现的场景，以及主集群是否会强同步日志给目标备集群。

	强同步模式备集群 `SYNC`	异步同步模式备集群 `ASYNC`
最大保护模式	强同步	异步同步
最大性能模式	异步同步	异步同步
最大可用模式	强同步 / 异步同步	异步同步

- 最大保护模式下，`SYNC` 模式备集群是强同步的，`ASYNC` 模式备集群是异步同步的。
- 最大性能模式下，无论配置何种传输模式，都是异步同步的。
- 最大可用模式下，`SYNC` 模式备集群会在强同步和异步同步之间切换；`ASYNC` 模式备集群是异步同步的。

9.3.5. 角色切换

OceanBase 集群角色包括两种：主集群（PRIMARY）和备集群（PHYSICAL STANDBY）。通过 Switchover 和 Failover 操作，用户可以动态改变集群角色。

- Switchover

  允许主集群与其中一个备集群交换角色，保证数据无损。

- Failover

  主集群不可用情况下，将一个备集群角色切换为主集群角色。

9.3.6. 备集群读服务

备集群各个分区的全功能型副本或者只读副本会实时回放 REDO 日志，并且能够承载读服务。OceanBase 数据库支持单分区读和跨分区一致性读，始终保证读取结果是一个快照，保证事务一致性。

由于不同分区以及不同副本的回放进度不同，默认情况下，可能出现先后两次读取的数据版本回退。为此，OceanBase 数据库提供了单调读开关。开启单调读情况下，内部会维护租户级别单调递增的可读版本号，每次读取都会获取全局最新的可读版本号作为快照版本，保证读取的数据越来越新，不会回退。

OceanBase 数据库支持有界旧（Bounded Staleness）一致性读，保证读取的数据不会落后于主集群最新数据特定时间，默认为 5 秒。如果备集群落后超过阈值，则读操作会重试等待，直到超时或者备集群追上。

更多详细信息，参见 弱一致性读。

9.3.7. 主备库的优势

主备库是 OceanBase 数据库高可用能力的重要组成部分，提供了更加灵活的灾难恢复和高可用解决方案。

主备库的优势包括以下几方面：

- 主库和备库互相不影响，故障隔离

  相比于单集群多副本，主备库是物理上互相独立的库，互相不影响，故障隔离。

- 灵活的高可用能力

  主集群在计划内和计划外不可用场景下，支持主备集群角色切换能力，包括 Switchover 和 Failover。提供双机房和两地容灾解决方案。

- 灵活的数据保护能力

  提供最大保护模式，最大性能模式，最大可用模式，帮助用户平衡数据的可用性和性能需求。

- 数据自动同步

  主集群自动向备集群发送 REDO 日志，保持主备同步。当主备网络断开后，备集群会落后主集群。等网络恢复后，备集群会自动找主集群同步 REDO 日志，达到同步状态，不需要人工干预。

- 数据物理一致

  主集群和备集群的数据和 Schema 是物理一致的，主备集群会自动进行 Checksum 校验，保证数据一致性。

- 数据实时回放

  备集群支持实时回放 REDO 日志，对外可以承担读服务。用户可以将报表、汇总计算等对实时性要求不高的业务放在备集群，降低主集群负载，高效利用 CPU 和 I/O 资源。

- 主备异构部署，独立运维

  主集群和备集群支持配置不同规格的资源，包括 CPU 和内存，以及副本数。主备集群各自独立运维。

9.4. 数据保护

9.4.1. 数据保护概述

OceanBase 数据库提供了丰富的特性用来保护数据在各种人为或非人为的异常场景下都安全可靠不丢失、不出错。主要包括提供高可用服务的多副本的复制、主备库，以及回收站、闪回和备份恢复等。

硬件异常

当磁盘发生故障的时候，存储在磁盘上的数据可能发生错误。OceanBase 数据库的 redo 日志和 SSTable 中的数据都保存了校验和，读取数据的时候执行严格的校验检查，及时发现错误。磁盘静默错误检测机制还能够发现哪些长期不读取的冷数据块中的错误。更多磁盘静默错误检测机制的介绍信息请参见 发现磁盘的静默错误。

当交换机等网络设备发生故障的时候，数据包可能发生错误。无论是客户端与 OceanBase 服务器之间，还是 OceanBase 集群内部的通讯，所有的网络数据包都有严格的数据校验和检查。

人为错误

当数据库对象被用户误删除的时候，可以通过回收站功能，从回收站中恢复误删除的对象。

当应用程序出现软件缺陷的时候，或者发生操作错误或恶意攻击的时候，表内数据被修改为错误数据。使用闪回查询功能，可以读取表数据的历史快照，从而恢复正确的数据。使用 restore point 特性，还可以在一些重大变更之前记录数据快照，以备异常的时候恢复到快照点。

当然，OceanBase 数据库的备份恢复功能提供了高效和丰富的备份所有数据和元数据的机制，它是保护用户数据最可靠的方式，更多备份恢复的介绍信息请参见 备份恢复概述。

9.4.2. 闪回查询

UNDO_RETENSION

`undo retension` 字面意思是 Undo 的保留位点，即从当前时间回溯多长时间的 Undo 日志是保留下来的。对于 OceanBase 数据库来说，是将该段时间的所有数据多版本保留下来。可以通过系统变量 `undo_retention` 进行设置，默认单位为秒。

Flashback Query

闪回查询（Flashback Query）允许用户查询某个历史版本的数据。OceanBase 数据库提供了如下语法进行查询：

- Oracle 模式：`AS OF SCN` 和 `AS OF TIMESTAMP` 。
- MySQL 模式：`AS OF SNAPSHOT` 。

用户配置系统变量 `undo_retention` 后，可以使用 Flashback Query 查询当前时间 `T` 到 `T - undo_retention` 时间范围内的任意多版本。如果用户保留了 Restore Point，则可以查询 Restore Point 点对应的数据版本。此外，OceanBase 数据库还支持查询集群合并点的数据。

查询副本选择

当用户指定查询某个历史版本数据时，并不会像普通查询一样总去查 Leader，而是优先查本地机器的副本，如果本地副本不存在，则优先查本 Zone 其他机器上的副本，最后再去查 Leader。

如下图所示，集群包含两个 Zone，数据分为三副本，其中副本 Replica 1 和 Replica 2 都在 Zone1，副本 Replica 3 在 Zone2，Leader 是 Replica 3。如果处理查询的 SQL 层在 server 1 上，则查询副本优先选择 Replica 1，其次选择 Replica 2，最后才会选择 Replica 3。

注意事项

如果在 `T - undo_retention` 时间段内有 DDL 操作，OceanBase 数据库会用最新的数据字典解释表的数据。对于 DDL 操作发生前产生的数据，删掉列的数据将变成删掉的状态，新添加列的数据则设为 Default 值。

9.4.3. Restore Point 功能

多版本

OceanBase 数据库是一个多版本的分布式数据库。对于任意一行来说，都有一个 Rowkey 与之对应，行的每个修改称为该行的一个版本。在内存中，行的一个版本用 TransNode 记录，多个版本串成链表，从新到旧连在一起。

以下图为例，行 A 有上有 4 个多版本，版本号分别为：12、11、8、2。

当内存中的数据量积累到一定程度后，我们会通过转储将其转储到磁盘，转储到磁盘的文件称为 SSTable。SSTable 中也保留了数据的多个版本，其中， multi_version_start 指明了该 SSTable 中多版本的起始位点，也即从这个点开始保留了所有数据的多版本信息。base_version 和 snapshot_version 分别指明了 SSTable 中包含数据的事务的提交版本号的左边界和右边界。

Restore Point 保留多版本

Restore Point 指定一个名字，该名字关联到了一个版本号，这个版本号即 Restore Point 创建时的版本号，该版本号对应的数据会被保存起来，以便查询该版本号对应的数据，甚至是将整个数据库回退到该位点。OceanBase 数据库暂不支持将数据回退到 Restore Point，仅支持查询。此外，Restore Point 在 OceanBase 数据库中是租户级别的配置，创建 Restore Point 时，只会保留相应租户的数据。

创建 Restore Point 时，会取当前时间戳作为要保留的版本号，并会等到 GTS 推过后才返回。此时就可以通过 Flashback Query 的语法直接读取多版本数据了。Flashback Query 相关信息请参见 闪回查询。

当 Restore Point 要查询的数据转储成 SSTable 后，后台线程会将这些 SSTable 管理起来，并与 Restore Point 相关联。在 SSTable GC 时，如果发现有 Restore Point 依赖，则不允许将它删除。

注意事项

- 创建 Restore Point 以后，对创建 Restore Point 之前就存在的表，不允许执行 DDL 语句。
- Restore Point 依赖全局一致性快照，因此在使用 Restore Point 时，需要开启 GTS。
- 为了防止 Restore Point 占用过多资源，每个租户限制最多创建 10 个 Restore Point。

9.4.4. 回收站

回收站从原理上来说就是一个数据字典表，用于放置用户删除的数据库对象信息，包括数据库和表等信息。用户删除的信息被放入回收站后，其实仍然占据着物理空间，除非您手动清除（PURGE）或者对象定期被数据库系统删除。

回收站支持的对象

如下表所示，在当前版本中，支持进入回收站的对象有索引、表和库。

模式	索引（Index）	表（Table）	数据库（Database）	租户（Tenant）
MySQL	√	√	√	√
Oracle	√	√	×	×

注意

- 直接 `DROP` 索引不会进入回收站，删表时表上的索引会跟随主表一起进入回收站。
- 不能对回收站的对象做任何查询和 DML 操作，DDL 操作中只能进行 Purge 和 Flashback。
- 租户回收站的管理主要由 `sys` 租户来完成。

查看回收站

租户管理员可以通过如下命令，查看回收站中的对象：

```
obclient> SHOW RECYCLEBIN;
```

开关回收站

租户被创建后，默认回收站为关闭状态。如果开启了回收站，在对数据库对象进行 `TRUNCATE` 或 `DROP` 操作后，对象会进入回收站，控制回收站开启关闭的命令分为 Global 级别和 Session 级别：

- Global 级别的开启、关闭语句如下所示：

```
obclient> SET GLOBAL recyclebin = ON/OFF;
```

- Session 级别的开启、关闭语句如下所示：

```
obclient> SET @@recyclebin = ON/OFF;
```

回收站恢复

使用 `FLASHBACK` 命令可恢复回收站中的数据库和表对象，只有租户的管理员用户才可以使用该命令。恢复时可修改对象的名称，但是不要和已有对象重名。

下述示例语句展示了如何恢复回收站中的租户、数据库及表对象：

- 恢复租户

```
obclient> FLASHBACK TENANT tenant_name TO BEFORE DROP [RENAME to new_tenant_name];
```

- 恢复对象数据库

```
obclient> FLASHBACK DATABASE object_name TO BEFORE DROP [RENAME TO database_name];
```

- 恢复对象表

```
obclient> FLASHBACK TABLE object_name TO BEFORE DROP [RENAME to table_name];
```

命令的使用限制如下：

- `FLASHBACK` 数据库对象的顺序需要符合从属关系，即：Database > Table。
- 恢复表会连同索引一并恢复。
- 通过 `PURGE` 命令可以单独删除索引，但是 `FLASHBACK` 命令不支持单独恢复索引。

- 如果一张表在进回收站之前属于某个表组，则删除该表组后再恢复改表会导致它不属于任何一个表组。如果表组还在，则恢复后该表还在原表组中。

回收站清理

频繁删除数据库对象并重建，会在回收站产生大量数据，这些数据可以通过 `PURGE` 命令清理。需要注意的是，`PURGE` 操作会删除对象和从属于该对象的对象（Database > Table > Index ）。

执行 `PURGE` 命令后，在 OceanBase 数据库的回收站中将再也查不到对象的信息，真实数据也最终会被垃圾回收。当一个对象的上层对象被 Purge，则当前回收站中关联的下一层对象也会被 Purge。

下述示例展示了一些如何通过 `PURGE` 命令清理回收站中的数据：

- `sys` 租户从回收站把指定租户物理删除。

```
obclient> PURGE TENANT tenant_name;
```

- 从回收站把指定库物理删除。

```
obclient> PURGE DATABASE object_name;
```

- 从回收站中把指定表物理删除。

```
obclient> PURGE TABLE object_name;
```

- 从回收站中把指定索引表物理删除。

```
obclient> PURGE TABLE object_name;
```

- 清空整个回收站。

```
obclient> PURGE RECYCLEBIN;
```

9.5. 备份恢复

9.5.1. 备份恢复概述

备份恢复是 OceanBase 数据高可靠的核心组件，通过纯 SQL 的命令就可以使用完整的备份和恢复功能。在 OceanBase 数据库的世界里，数据的高可靠机制主要有多副本的容灾复制、回收站、主备库和备份恢复等，备份恢复是保护用户数据的最后手段。

常见的数据异常问题如下：

- 单机问题：常见的有磁盘错误、磁盘损坏、机器宕机等场景，这些场景一般通过多副本的容灾复制能力就能恢复正常。
- 多机问题：常见的是交换机损坏、机房掉电等场景。
  - 少数派副本的问题：OceanBase 数据库的多副本机制能够保证缺少数派副本的时候正常运行，并且故障节点恢复正常后能自动补全数据。

  - 多数派副本的问题：这种场景下多副本机制无法自动的恢复数据，一般来说冷备的恢复时间会比热备的备库恢复耗时长。如果部署有备库，优先建议使用备库切主作为恢复服务的应急措施；如果没有部署备库，建议使用备份恢复来恢复数据。
- 人为操作：常见的是删表、删库、删行、错误的程序逻辑造成的脏数据等操作
  - 对于一般的误删表、库的操作建议通过回收站的功能恢复数据。
  - 对于行级别的误操作或者更为复杂的程序逻辑错误造成的大规模数据的污染，建议通过备份恢复功能来恢复数据。

OceanBase 数据库的备份按照备份的形式区分，主要分为数据备份和日志备份两种：数据备份是指存储层的基线和转储数据，也就是备份时刻的 Major SSTable + Minor SSTable；日志备份是指事务层生成的 Clog，包含了 SSTable 之后修改的数据。

目前支持集群级别的备份和租户别级的恢复。恢复的时候允许通过白名单机制指定恢复的表或库。

9.5.2. 备份恢复的元信息管理

OceanBase 数据库的备份数据主要分为数据备份和日志备份：

- 单次的数据备份对应一个 backup_set。每次用户运行 `alter system backup database` 都会生成一个新的 backup_set 的目录，该目录包含了本次备份的所有数据。
- 日志备份可以由用户指定是否按天切分目录：如果是配置了按天拆分目录，每一天的日志数据目录对应一个 backup_piece；如果没有配置拆分目录，那么整个备份的日志数据目录对应一个 backup_piece。

在 OceanBase 数据库中，备份恢复的元信息是指 backup_set 的信息、backup_piece 的信息和租户信息：

- backup_set：在备份介质上保存在租户下面的 `tenant_data_backup_info` 文件中，在原备份集群保存在 `oceanbase.__all_tenant_backup_set_files` 内部表中。
- backup_piece：在备份介质上保存在租户下面的 `tenant_clog_backup_info` 文件中，在原备份集群保存在 `oceanbase.__all_tenant_backup_piece_files` 内部表中。
- 租户信息：在备份介质上保存在 `tenant_name_info` 文件中，在源集群上保存在 `oceanbase.__all_tenant` 和 `oceanbase.__all_tenant_history` 内部表中。

9.5.3. 备份架构

本节主要介绍物理备份的方式及其架构。

物理备份

OceanBase 数据库提供了在线物理备份的功能，该功能由日志备份+数据备份两大子功能组成。日志备份持续的维护了集群产生的日志，数据备份维护了快照点的备份，两者结合可以提供恢复到备份位点之后任意时间的能力。

日志备份

OceanBase 数据库提供了集群级别的日志备份能力。日志备份是 log entry 级别的物理备份，并且能够支持压缩和加密的能力。

日志备份的工作由 Partition 的 Leader 副本负责。每个 OBServer 负责本机 Leader 副本日志的读取、按照 Partition 拆分、日志聚合后发送的工作。

日志备份的周期默认为 2 分钟，提供了准实时的备份能力。

V2.2.77 版本以后，日志备份提供了按天拆分目录的功能，方便用户对于备份数据的管理。

数据备份

OceanBase 数据库提供了集群级别的数据备份的能力。数据备份目前是基于 restore point 的能力做的数据快照保留，保证了备份期间的数据能够保持全局一致性。数据保留带来了额外的磁盘空间的消耗，如果备份机器的磁盘水位线超过配置的警戒值，会导致数据备份失败。

数据备份的流程都是 RootService 节点调度的，将 1024 个分区作为一组任务发送给 OBServer 备份。备份数据包括分区的元信息和宏块数据。物理备份是指宏块数据的物理备份，元信息是内存序列化后的值。

OceanBase 数据库的基线宏块具有全局唯一的逻辑标识，这个逻辑标识提供了增量备份重用宏块的能力。在 OceanBase 数据库中，一次增量备份指的是全量的元信息的备份+增量的数据宏块的备份。增量备份的恢复和全量备份的恢复流程基本上是一致的，性能上也没有差别，只是会根据逻辑标识在不同的 backupset 之间读取宏块。

数据清理

OceanBase 数据库提供了当前配置路径下自动清理的功能，该功能由 RootService 定期检查用户配置的 Recovery Window，从而删除不需要数据备份。在删除数据备份的同时，会自动的根据保留的数据备份中最小的回放位点来删除不需要的日志备份。

备份的备份

备份的备份是指数据库备份的二次备份能力，通常一次备份为了较好的性能，会存放在性能较好的备份介质上，而这种介质的容量会比较有限，保留备份的时间会比较短；二次备份则是把一次备份的数据挪到空间更大、保留时间更长、成本更低的介质上。

目前 OceanBase 数据库提供了内置的备份的备份的功能，支持用户调度 OBServer 将一次备份的数据搬迁到指定的目录中。 目前支持 OSS 和 NFS 两种介质。

逻辑备份

V2.2.7 版本以后 OceanBase 数据库不再提供集群级别的逻辑备份功能，提供了数据的逻辑备份的 OBDUMPER 工具。该工具是 java 开发的一个并行的客户端导出工具，支持 SQL 或者 CSV 格式的数据导出，也支持全局的过滤条件。

有关逻辑备份工具的具体介绍，请参见 OBDUMPER 文档。

9.5.4. 恢复架构

本节主要介绍物理恢复流程及架构。

恢复概述

OceanBase 数据库提供租户级别的恢复能力，支持微秒级别的恢复精度。

租户恢复保证了跨表、跨分区的全局一致性。

OceanBase 数据库的恢复主要包含下面几步：

1. 恢复系统表的数据。
2. 恢复系统表的日志。

3. 修正系统表的数据。
4. 恢复用户表的数据。
5. 恢复用户表的日志。

对于单个分区来说，备份+恢复的流程和重启的流程比较类似，主要就是加载数据和应用日志。

恢复系统表的数据

RootService 根据数据备份的系统表的列表创建对应的分区，然后依次调度分区的 Leader 从备份的介质上拷贝分区的元信息、宏块数据。

恢复系统表的日志

日志恢复和重启后回放日志的流程比较类似，恢复的分区的 Leader 在完成数据恢复后，会主动从备份介质上拉取备份的分区级别的日志并保存到本地的 Clog 目录。Leader 将恢复的日志保存到本地的同时，Clog 回放的线程会同时开始回放数据到 MEMStore。等所有的 Clog 都恢复完成以后，一个分区的恢复就完成了。

修正系统表的数据

系统表恢复完成后，RootService 会进行系统表数据的修复：

1. 清理未建完的索引表。
2. 老版本的备份被恢复到新版本的集群上的兼容补偿：
    i. 补建新加的系统表
    ii. 补偿跨版本的升级任务

恢复用户表

用户表的数据、日志的恢复流程和系统表类似，唯一的区别是创建分区的列表依赖的数据源不同。恢复用户表的时候，RootService 是从已经恢复的系统表中读取相关列表的。

恢复事务的一致性

OceanBase 数据库的物理备份恢复强依赖租户的 GTS 功能，GTS 保证了备份和恢复的数据是全局一致的。

逻辑恢复

V2.2.7 版本以后 OceanBase 数据库不再提供集群级别的逻辑备份恢复功能，提供了数据的逻辑备份的 OBDUMPER 工具和逻辑恢复的 OBLOADER 工具。该工具是 java 开发的客户端导入工具，提供了支持 SQL 或者 CSV 格式的数据导入，也支持导入的限流、断点恢复的能力。

有关逻辑备份工具的具体介绍，请参见 OBDUMPER 文档。

有关逻辑恢复工具的具体介绍，请参见 OBLOADER 文档。

10.数据库安全

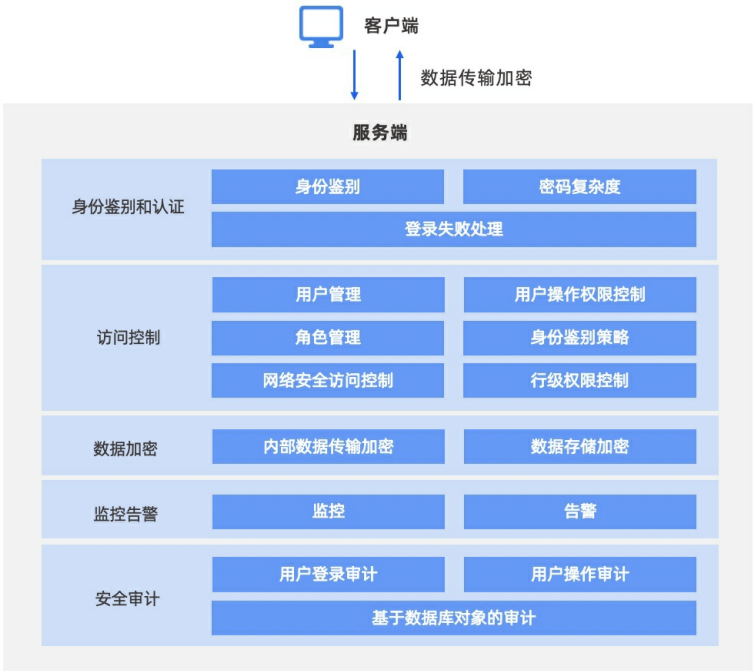
10.1. 安全机制总览

数据库系统的重要指标之一是要确保系统安全，通过数据库管理系统防止非授权使用数据库，保护数据库的文件和数据。本文为您介绍 OceanBase 数据库的安全体系。

数据是企业最重要的资产，经常面临来自于各个方面的威胁，数据泄露的危害越来越大，范围也越来越广。

OceanBase 数据库作为数据的最终载体，时刻关注保护数据的安全。数据库的安全特性从网络传输、用户认证、操作审计、存储安全、高可用等多个方面，全面保护您的数据安全。

OceanBase 数据库的安全体系包括身份鉴别和认证、访问控制、数据加密、监控告警、安全审计。



OceanBase 数据库已经支持比较完整的企业级安全特性，可以有效保证用户的数据安全。OceanBase 数据库安全体系的组成如下：

- 身份鉴别和认证

  OceanBase 数据库支持身份标识和鉴别、用户管理和角色管理，提升数据库的安全性。详情请参见 身份鉴别和认证。

- 访问控制

  OceanBase 数据库通过定义各种系统、对象权限以及角色来控制用户对数据的访问。详情请参见 访问控制。

- 数据加密

  OceanBase 数据库支持在数据传输和存储过程中，对数据进行加密。对于传输层，OceanBase 数据库支持全链路数据加密。同时，在数据存储时，OceanBase 数据库支持透明加密特性，即使存储介质丢失，依然可以保证数据不会丢失，最大化保护用户的数据安全。详情请参见 数据传输加密 和 数据存储加密。

- 监控告警

OceanBase 数据库通过 OceanBase 云平台（OceanBase Cloud Platform，OCP）进行监控告警。OCP 是专门为金融级分布式关系型数据库 OceanBase 打造的管控平台，包括资源和容器管理、集群和实例生命周期管理、OpenAPI 以及基于实时计算的性能监控等功能模块。详情请参见 监控告警。

- 安全审计

  OceanBase 数据库可以对数据库用户的行为进行审计，确保用户的操作都会被记录。同时，OceanBase 数据库还支持对数据库对象的审计操作，可以在用户访问数据时，产生审计信息，确保数据访问被真实记录。详情请参见 安全审计。

10.2. 身份鉴别和认证

身份标识与鉴别功能用于对登录数据库访问数据的用户进行身份验证，确认该用户是否能够与某一个数据库用户进行关联，并根据关联的数据库用户的权限，对该用户在数据库中的数据访问活动进行安全控制。OceanBase 同时支持了 MySQL 和 Oracle 两种模式的身份鉴别功能。

MySQL 模式

身份鉴别

MySQL 中，一个 user 由 user_name 和 host 共同体组成。使用下列语句创建 3 个具有相同用户名的用户的示例如下：

```
create user 'u1'@'%' identified by '123';
create user 'u1'@'1.1.1.1' identified by '111';
create user 'u1'@'2.2.2.2' identified by '222';
```

用户登录时，OBServer 会根据 user_name、client_ip、password 匹配，来控制用户登录。

密码复杂度

为了防止恶意的密码攻击，OceanBase 数据库用户可以根据需要设置密码的复杂度函数，验证用户登录身份，来提升数据库的安全性。

MySQL 模式下，用户可以通过设置一系列系统变量规定密码复杂度规则，这些配置是租户级的。创建用户或修改用户的密码时会根据系统变量的配置对密码进行校验，未通过校验则会报错。涉及的系统变量如下表所示：

变量名	功能	使用说明
validate_password_check_user_name	检查用户密码，是否可以和用户名相同	当为 on 时，代表用户密码可以和用户名相同。
validate_password_length	设置用户密码最小长度	-
validate_password_mixed_case_count	设置用户密码至少包含的大写字母和小写字母个数	-

变量名	功能	使用说明
validate_password_number_count	设置用户密码至少包含的数字个数	-
validate_password_policy	设置密码检查策略	• LOW 策略仅包含密码长度检测。 • MEDIUM 策略包括密码长度检测、大写字母个数检测、小写字母个数检测、数字个数检测、特殊字符个数检测、用户名密码相同检测。
validate_password_special_char_count	用户密码至少包含的特殊字符个数	-

登录失败处理

OceanBase 数据库会锁定多次登录失败的用户，主要目的在于防止恶意的密码攻击，保护数据库，提升数据库的安全性。

MySQL 模式下，使用系统变量 `connection_control_failed_connections_threshold` 来指定用户错误登录尝试的阀值。该变量是租户级的，为整数类型，取值范围为 [0，2147483647]，默认值为 0。当用户将参数设置为 0 时表示关闭该功能，即不对用户的错误登录尝试进行处理。当用户错误登录次数超过这个变量定义的值之后，会对账户进行锁定，锁定的时间参考下面的两个参数设置。

- `connection_control_min_connection_delay`：指定了超过错误登录次数阀值之后的第一次错误锁定账户的时长。之后的第二次错误登录的锁定时长为 `min（connection_control_min_connection_delay+1000，1000* trunc（connection_control_min_connection_delay/1000,0））`。再之后每次错误登录的锁定时间都在原来的基础上再加 1000 毫秒（1 秒）。
- `connection_control_max_connection_delay`：指定了错误链接锁定时长的最大值，当时长达到这个最大值之后就不再增长。

Oracle 模式

身份鉴别

Oracle 模式中，用户名称在租户内是唯一的，不同租户的用户可以同名，所以通过 `用户名@租户名` 的形式在系统全局唯一确定一个用户。

密码复杂度

为了防止恶意的密码攻击，OceanBase 数据库用户可以根据需要设置密码的复杂度函数，验证用户登录身份，来提升数据库的安全性。

Oracle 模式下，用户可以使用 Profile 中的 PASSWORD_VERIFY_FUNCTION 属性来验证密码复杂度是否符合要求。用户需要先创建一个用于校验密码复杂度的 PL Function，需要满足如下接口：

```
FUNCTION verify_function (username     IN VARCHAR2,
                          password     IN VARCHAR2,
                          old_password IN VARCHAR2)
RETURN BOOLEAN;
```

创建用户或修改密码时，会调用该校验函数，根据函数执行结果来判断新密码是否符合复杂度要求。

登录失败处理

OceanBase 数据库会锁定多次登录失败的用户，主要目的在于防止恶意的密码攻击，保护数据库，提升数据库的安全性。

Oracle 模式利用 User Profile 实现用户锁定功能。User Profile 的密码提供两个参数，来实现连续登录失败后锁定用户：

- failed_login_attempts：连续登录失败的次数。
- password_lock_time：锁定时间，单位为天。

10.3. 访问控制

一个完善的数据库系统，除了需要管理用户外，也需要设置普通用户来进行数据库对象的操作，但普通用户必须拥有对应数据库对象的访问和操作权限，才能进行数据库对象的访问和操作，否则，用户无法进行任何操作。

Oracle 模式权限

权限分类

Oracle 模式的权限分为两类：

- 对象权限：对特定对象的操作权限，例如：某个表对象的 Alter、Select、Update 等权限。
- 系统权限：允许用户执行在一个 Schema 或者任何 Schema 上进行特定的数据库操作的权限。

系统权限提供的权限比对象权限大得多。

权限转授

权限转授解决了授权者集中的问题。通过在授权时指定 with admin/grant option，可以同时授予用户将对应权限转授给其他用户的权限。回收对象权限时，要同时回收该用户转授给其他用户的对应权限，即 A 授予 B 权限，B 授予 C 权限，如果 A 收回 B 的权限，C 的权限也会被回收。回收系统权限时，不会级联回收转授的权限。

角色

为了方便权限的管理，设定了角色。角色是一组系统权限、对象权限的组合，角色中也可以包含其他角色。可以把角色授予用户，用户就拥有了角色里面的所有权限。系统在创建新租户时内置了三个默认的角色：

- DBA：拥有绝大多数的系统权限。
- RESOURCE：拥有 Resource 角色的用户只可以在自己的 Schema 中创建数据库对象。
- CONNECT：拥有 Connect 权限的用户只可以登录数据库。

间接权限

用户的权限，包含直接授予的系统权限或者对象权限，也包含授予角色后，通过角色包获得的权限。大部分操作需要的权限，不论是直接授予的还是通过角色间接拥有的权限，都可以满足条件。对于下列场景，需要有直接权限才可以：

- 创建视图时，访问视图中的对象所需的权限。
- 执行定义者权限的有名 PL 块中的语句所需的权限。

权限检查

在 Resolver 阶段，解析出一条 SQL 语句中所需的所有权限，逐个检查用户是否拥有对应的权限。对于系统操作的权限，如果权限不足则直接报错权限不足。对于访问对象的权限，如果用户在这个对象上没有任何权限，则报错对象不存在；如果用户在这个对象上有其他权限，只是没有需要的权限，则报错权限不足。

MySQL 模式权限

与 Oracle 模式异同

MySQL 模式没有角色，权限模型相对简单。权限检查逻辑和 Oracle 一致。

权限分类

MySQL 模式的权限分为了 3 个级别：

- 管理权限：可以影响整个租户的权限，例如：修改系统设置、访问所有的表等权限。
- 数据库权限：可以影响某个特定数据库下所有对象的权限，例如：在对应数据库下创建删除表，访问表等权限。
- 对象权限：可以影响某个特定对象的权限，例如：访问一个特定的表、视图或索引的权限。

网络安全访问控制

OceanBase 数据库提供白名单策略，实现网络安全访问控制。租户白名单通过系统变量 `ob_tcp_invited_nodes` 控制，支持列表形式取值，列表值之前使用英文逗号（,）分隔，例如：A,B,C,D。

用户登录时，OBServer 会将用户的 IP 地址依次和 A、B、C、D 进行匹配校验。如果全部不匹配，则拒绝访问。只要有任意一个匹配成功，则表示通过白名单。

列表值支持以下赋值：

- IP 地址，例如：192.168.1.1。匹配时采用等值匹配，即用户 Client IP 等于该 IP 值时，才算匹配。
- 包含百分号（%）或下划线（_）的 IP 地址，例如：192.168.1.% 或 192.168.1._。匹配时采用模糊匹配，即类似 LIKE 语法。
- IP/NETMASK 地址，例如：192.168.1.0/24 或者 192.168.1.0/255.255.255.0。匹配时采用掩码匹配，只有满足 Client_IP & NetMask == IP 才算匹配成功，类似于 MySQL 的掩码匹配。

行级访问权限控制

OceanBase 数据库兼容了 Oracle 的 Label Security 功能，可以在行级别对访问进行控制，保证读写数据的安全。

Label Security 是强制访问控制的一种方式，通过在表中添加一个 Label 列来记录每行的 Label 值，在访问时通过比较用户的 Label 和数据的 Label，达到约束主体（用户）对客体（表中的数据）访问的目的。

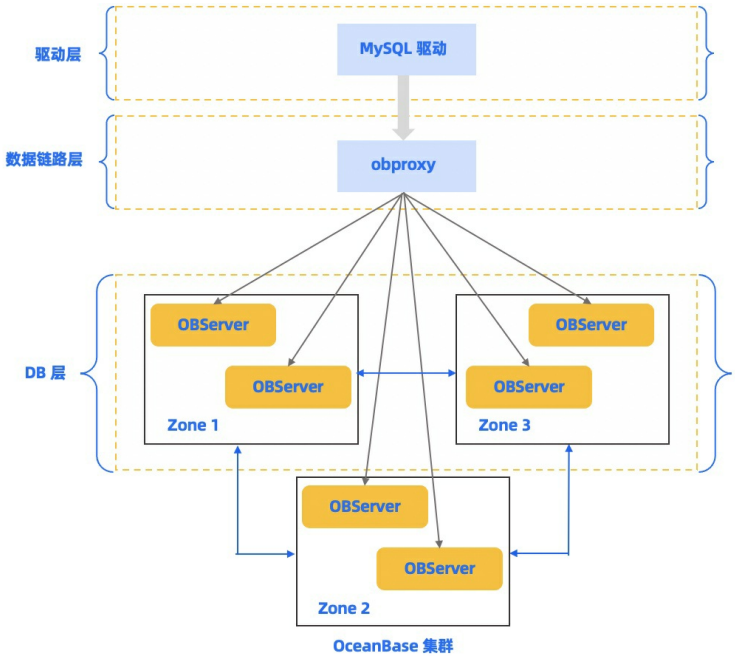
OceanBase 数据库提供了内置的安全管理员 `LBACSYS` 来管理和使用该功能，安全管理员可以通过创建安全策略、定义策略中的 Label、设置用户的 Label，来定制自己的安全策略。一个安全策略可以应用到多张表上，一张表也可以应用多个安全策略。每当一个安全策略被应用，这张表上自动会添加一列，用于该安全策略的访问控制。

> 注意
>
> 目前仅 Oracle 模式支持 Label Security。

10.4. 数据传输加密

安全传输层协议（TLS）用于在两个通信应用程序之间提供保密性和数据完整性。OceanBase 数据库在原有的 TCP 通信上扩展支持 SSL/TLS 协议，解决通信加密的问题。

OceanBase 数据库从产品架构层面可以分为以下三个基本部分。



其中采用两种协议进行通信：

- MySQL 协议：驱动层 <=> 数据链路层，数据链路层 <=> DB 层之间的通信采用扩展的 MySQL 协议。
- OB-RPC 协议：OBServer 与 OBServer 之间/OBServer 与 liboblog、ob_admin 等之间的通信采用 OBServer 自有的 RPC 协议通信。

其中数据链路层，DB 层的 OBServer、liboblog、ob_admin 等组件均支持 SSL/TLS 加密通信，底层均依赖 OpenSSL 或者第三方的 SSL 库，为业务提供安全的加密传输服务。

支持的私钥/证书加载方式

OBServer、liboblog、obadmin 等组件由于底层通信均依赖 Libeasy 库，因此支持的私钥/证书加载方式均相同，一种是本地文件模式，另一种是 BKMI 模式。

- 本地文件模式

将 CA 证书、用户证书、私钥放在 `wallet` 文件夹下，根据配置项开启时从该目录下读取加载，安全性较差。

- BKMI 模式

  BKMI（基础密钥服务）可以理解为密钥、证书产生和托管的仓库，当需要加载时可以通过 BKMI 分配给用户(OBServer 等）的应用身份私钥，用户身份信息等通过 URL 访问的方式，从 BKMI 获取加密后的私钥，之后通过应用身份私钥解密解出应用私钥来进行加载配置，安全性较高。

开启方式

OBServer 传输加密的开启通过多个配置项配合使用。

1. 指定私钥/证书/CA 证书的获取方式。

   本地文件模式

   ```
   alter system set ssl_external_kms_info = '
   {
   "ssl_mode":"file"
   }';
   ```

   或者

   ```
   alter system set ssl_external_kms_info = '';
   ```

   BKMI 模式

   ```
   alter system set ssl_external_kms_info = '
   {
   "ssl_mode":"bkmi",
   "kms_host":"http://xxx",  bkmi主站的url
   "root_cert": CA证书内容
   "private_key": bkmi分配给用户的应用身份私钥，用来解密从bkmi获取的加密的应用私钥
   "PRIVATE_KEY_PHRASE":"123456", bkmi分配给用户的应用身份私钥的解密密码
   "SCENE":"ANT", 应用场景
   "CALLER":"xxx", 进行服务调用的用户名，由bkmi分配
   "CERT_NAME":"xxx", 用户证书的名称
   "PRIVATE_KEY_NAME":"xxx", 用户私钥的名称
   "KEY_VERSION":"1"
   }';
   ```

2. 配置对应协议的 SSL 使能。

   ```
   alter system set ssl_client_authentication = 'TRUE';
   配置为TRUE后，mysql通信ssl即时开启
   ```

3. 配置 RPC 通信的 SSL 白名单，由于 OBServer 之间 TCP 连接都是长连接，因此需要重启 OBServer 后 RPC SSL 加密通信才能开启。

   ```
   rpc通信ssl需要配置白名单
   alter system set _ob_ssl_invited_nodes='ALL'; 整个集群都开启
   alter system set _ob_ssl_invited_nodes='123.12.21.12, 128.191.11.12'; 指定ip的observer开启
   ssl
   ```

OBServer 如何确定通信加密开启成功

最简单的方法是对 MySQL 端口和 RPC 端口进行抓包，查看是否开启加密，此外可以通过以下方式：

- OBClient/MySQL 登陆 OBServer，系统租户检索 `oceanbase.__all_virtual_server_stat` 表的 `ssl_key_expired_time` 字段确认是否开启。
- 该字段记录当前 OBServer 开启 SSL 时，当前 OBServer 使用的 SSL 证书过期时间， utc 时间，单位微秒。

```
obclient> select svr_ip, svr_port,zone, ssl_key_expired_time, from_unixtime(ssl_key_expir
ed_time/1000000) from oceanbase.__all_virtual_server_stat;
+--------------+----------+-------+---------------------+------------------------------
--------------+
| svr_ip       | svr_port | zone  | ssl_key_expired_time | from_unixtime(ssl_key_expired_
time/1000000) |
+--------------+----------+-------+---------------------+------------------------------
--------------+
| 100.83.6.189 |    13212 | zone1 |           1871860075 | 2029-04-26 09:07:55
|
+--------------+----------+-------+---------------------+------------------------------
--------------+
1 row in set (0.00 sec)
```

检查 MySQL 端口是否开启 SSL

通过 MySQL/OBClient 登录，执行 `\s` 查看 SSL 字段的示例如下：

```
obclient> \s
--------------
obclient  Ver 1.1.8 Distrib 5.7.24, for Linux (x86_64) using  EditLine wrapper

Connection id:  3221506046
Current database: test
Current user:  root@11.166.79.39
SSL:   Cipher in use is DHE-RSA-AES128-GCM-SHA256
Current pager:  less
Using outfile:  ''
Using delimiter: ;
Server version:  5.7.25 OceanBase 2.2.60 (r1-63cbd3084a3283523f09d6ba20795f77b95e046b) (Bui
lt Jun 30 2020 10:10:29)
Protocol version: 10
Connection:  100.83.6.189 via TCP/IP
Server characterset: utf8mb4
Db     characterset: utf8mb4
Client characterset: utf8mb4
Conn.  characterset: utf8mb4
TCP port:  13213
Active   --------------
```

检查 RPC 端口是否开启 SSL

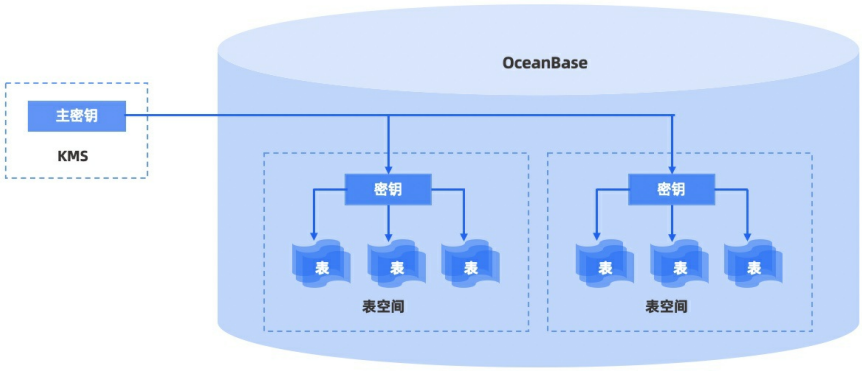
在 `OBServer.log` 中的日志，搜索关键字 "RPC connection accept" 查看 use_ssl = True 还是 False 判定是否 RPC SSL 开启成功。

10.5. 数据存储加密

透明数据加密（简称 TDE）可以对存储在磁盘的上的敏感数据（包括基线数据和clog）进行加密。数据在写入存储设备前自动进行加密，读取时自动解密，该过程对用户是透明的，经过数据库认证的用户可以不受限制地访问数据。当存储介质被盗时，透明数据加密可以保证存储在介质上的敏感数据无法被未经认证的用户访问。

两级密钥体系

透明数据加密使用两级密钥体系实现加解密功能。开启加密的最小粒度为数据库中的一个表，需要开启加密的表需要放到一个加密的表空间（tablespace）中。每个加密的表空间设置有加密算法及对应的数据密钥，用于给表空间中的数据进行加密。每个租户有一个主密钥，用于对表空间密钥进行加密，为了防止未经授权的解密操作，通常将主密钥存储在外部密钥管理服务（简称 KMS）中。



开启加密时，使用主密钥对数据密钥进行加密后得到加密的数据密钥密文，该密文会存储在内部表、宏块头部、clog 头部中，数据密钥不会明文存储在任何地方。需要加解密数据时，使用主密钥解密该密文后得到数据密钥，从而对宏块或 clog 中的用户数据进行加解密操作。

加密生效机制

用户存储在磁盘上的数据包括了 clog 和宏块两类，只有在将数据实际写到磁盘上时加密才会生效，内存中的数据是不加密的。若将原本不加密的表改为加密后，原先已经存储在磁盘上的未加密的 clog 和宏块数据不会变为加密，后续新写到磁盘上的数据才是加密的。由于每一条 clog 和每一个宏块都会记录加密元信息，因此加密和未加密的数据可以同时存在。

clog 是采用追加写的方式写到磁盘上的，因此旧的未加密的 clog 永远不会变为加密。等开启加密一段时间，旧的 clog 都被回收后，磁盘上的 clog 数据就会都成为加密数据了。对于宏块数据，只有在转储或合并时才会发生写入，可以手动触发全量合并将未加密的宏块数据重写为加密的。

支持的加密算法

算法	密钥长度	模式

算法	密钥长度	模式
AES	128 bits 192 bits 256 bits	ecb
SM4	128 bits	cbc

10.6. 监控告警

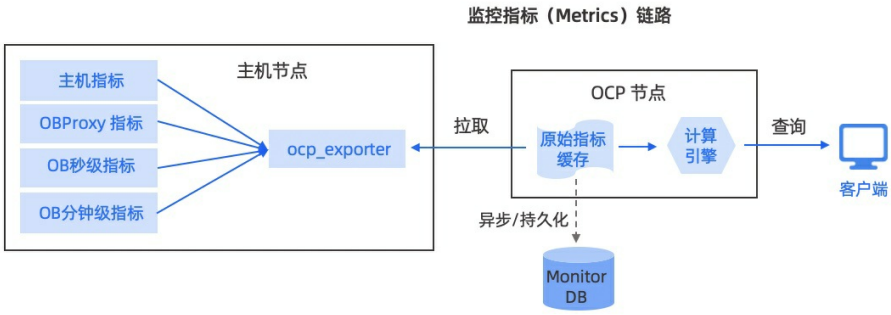
OceanBase 数据库监控目前主要依赖 OCP 的监控功能，支持数据库集群维度、租户维度、节点维度的性能、容量、运行状态等指标 7*24 监控采集，图表可视化展现，帮助用户全面了解 OceanBase 集群使用状况，及时发现集群异常，触发事件及时预警，确保数据库稳定、高效的正常运行。

监控

OceanBase 数据库监控从处理链路方面可以分为以下几个部分组成：

- Metric 链路（常规监控指标）：OBServer 状态监控、OBProxy 状态监控、主机指标监控。
- OB SQL 链路：包括 OceanBase 数据库相关的 SQL、Plan 指标。
- OB 资源水位链路：采集 OceanBase 集群、租户资源水位。

Metric 链路

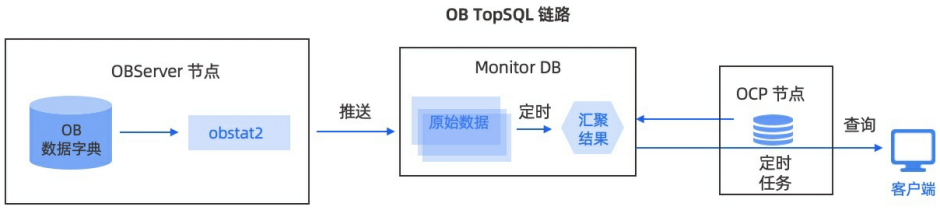


此链路根据采集了以下几种指标：

- 主机指标：包括主机、主机上部署的相关服务（如 OBServer、OBProxy） 的 CPU、磁盘、IO 、LOAD 等信息。
- OBProxy 指标：OBProxy 的相关请求、会话、事务等信息。
- OB 秒（分钟）级别指标：对应 OB 节点的资源状态、QTPS 等性能监控监控信息。

Metric 链路依赖部署 OCP 管理的主机上的 OCP Agent 的 ocp_exporter 程序来进行采集，ocp_exporter 对外提供了一组 RESTFul 服务来进行监控采集，接口提供 Prometheus 规范的监控指标，内部依赖 NodeExporter（主机监控）、OBProxyExporter（OBProxy 监控）、OBCollector（OceanBase 监控）来采集多种指标。OCP 在采集到指定类型监控指标后会将监控指标汇总转换后保存到监控库（MonitorDB），监控计算引擎会根据监控表达式（基于 Prometheus 的表达式）到监控库查询监控数据并进行计算后返回给客户端，客户端根据返回的计算后的信息展示监控图表。

OB SQL 链路

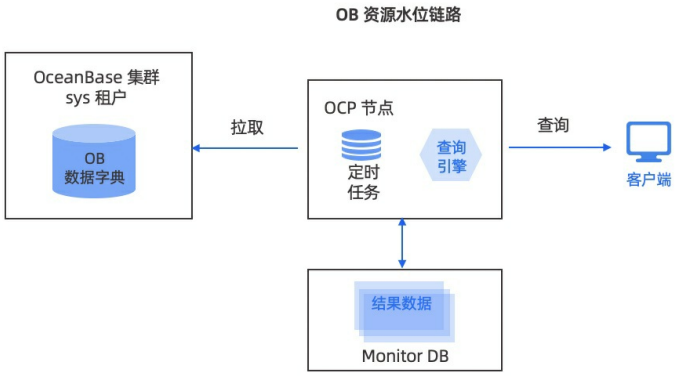


此链用来采集各个 OB 集群的 SQL 数据和 SQL 执行计划数据，主要从以下OB 数据字典采集数据：

- v$sql_audit：记录 SQL 的执行审计信息，
- v$plan_cache_plan_explain：记录的是执行计划的每个算子的信息。
- v$plan_cache_plan_stat：记录执行计划的审计信息。

考虑到 SQL 数据和 SQL 执行计划的数据量比较大，为了提升性能和降低资源消耗，此链路的采集程序 obstat2 是使用 C++ 开发的高性能、轻量级的程序。 它会根据采集频率配置，在采集周期内从 OB 数据字典采集 SQL 和执行计划信息，并在本地汇总计算后存入OCP 的监控库 （MonitorDB）。 OCP 后台任务会定期去监控库中进行汇总计算，将最小粒度的 SQL audit 和 SQL plan 数据汇总成大时间粒度的结果，并保存下来。当从web页面进行查询时，如果选择时间区间较小时，直接查询原始上报的表，如果时间粒度较大的则查询汇总表。

OB 资源水位链路



此链路负责采集 OceanBase 集群的资源水位，主要从以下数据源采集数据：

- CPU 信息：
  - __all_virtual_server_stat：CPU 总核数、已分配核数。
- 内存信息
  - __all_virtual_server_stat：内存总大小、已使用大小。
- 磁盘信息
  - __all_virtual_disk_stat：磁盘总大小、已使用信息。
- 系统事件信息
  - __all_rootservice_event_history：OceanBase 集群系统事件信息。
- 分区副本信息
  - __all_meta_table：OceanBase 数据库 1.4.x 版本的分区副本信息表。

- __all_virtual_meta_table：OceanBase 数据库 2.0 访问全部租户信息。
- __all_virtual_tenant_partition_meta_table：OceanBase 数据库 2.2.x 访问 PG 级别的信息。

OB 资源水位链路是由 OCP 的定时任务触发，使用各个集群的 sys 租户按照集群、租户、数据库、表等维度采集 OBServer 上的 CPU、Memory、Disk、分区副本等使用率，并将数据保存到监控库。在客户端发起查询请求时查询引擎会按照集群、租户、数据库、表等维度进行统计，将统计后的数据返回给前端展示。

告警

OceanBase 数据库主要通过 OCP 告警实现对于生产中主机、数据库风险、故障的预警。当数据库及数据库所处主机环境即将发生故障或发生故障时，内置的告警项会检测到异常，并通过告警通道将告警发送给告警订阅者。主要从告警项配置、告警检测、告警聚合、告警订阅四个方面为您介绍告警功能。

告警项

OCP 内置了约60个告警项，每个告警项描述了告警的基本信息，如告警名称、等级、概述模板、详情模板等，以及告警检测相关的规则信息。

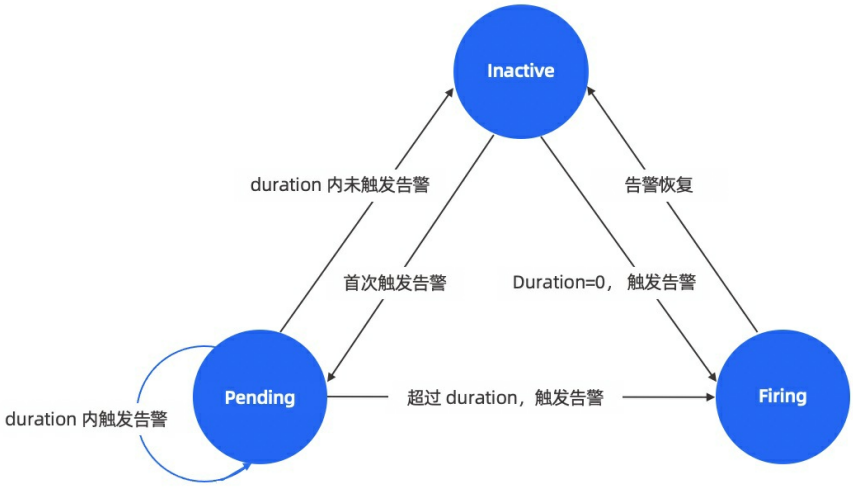
根据告警的风险程度，划分了5个告警等级：停服、严重、警告、注意、提醒。产生告警时，会生成一些相关的模板变量，模板变量可以配置在概述模板和详情模板中，以展示必要的上下文信息。告警检测规则是基于监控表达式的检测规则，如检测时长、检测周期、告警恢复周期、检测表达式配置等。

这些内置的告警项大致是从数据库资源、数据库事件、主机资源、OCP事件等方面描述告警，如数据库CPU、内存、memstore、磁盘使用率等数据库资源告警，合并超时、悬挂事务等数据库事件告警，主机网络、磁盘告警，以及OCP的监控API状态异常、metadb同步OB集群等告警。

告警检测

告警检测是检测内置的告警项并触发告警的过程，分为 2 种检测：基于监控表达式的检测和定时任务的逻辑检测。告警检测之后，会产生告警事件，也就是产生了告警；但该告警事件是否需要通知到用户，还依赖后续的告警聚合逻辑：可能会需要对告警做一些抑制操作，避免发生大量告警消息。

基于监控表达式的告警，可以通过监控API从不同维度聚合监控数据，对API的查询结果作阈值匹配，满足告警触发规则就会产生告警事件。



上图是基于监控表达式的告警检测中的状态流转，duration 是指告警的检测时长，当在 duration 时长内连续触发告警，则会产生告警事件。duration 常用于对异常状态的容错场景：偶发的一次异常不期望立即触发告警，只有持续异常才触发告警。

定时任务的逻辑检测是对应一些复杂场景，需要用一些脚本语言来检测。这类检测方法是直接调用告警 API，产生告警事件，如 OB 日志告警、OMS 告警等，适用于外部系统（OCP 以为的系统）的基于事件的告警。

告警聚合

聚合告警是指，把告警消息以预设规则合并成少量告警消息（称之为聚合消息），以防止告警风暴。

如下，是告警的聚合配置，这是一个深度优先匹配规则，如 OB 日志告警（ob_log_alarm）的聚合维度是 alarm_type（告警项）+ob_error_code（日志错误码）+obregion（OB 集群名）。

```
aggregate:

  # root 层为默认聚合，按照 告警类型，对象 进行聚合
  match: {}
  group_by:
    - "alarm_type"
  aggregate_wait_seconds: 10
  aggregate_interval_seconds: 60
  repeat_interval_seconds: 3600

  aggregates:

    # 对于 OB 告警，按照 告警类型，OB集群 进行聚合
    - match:
        app: "OB"
      group_by:
        - "alarm_type"
        - "obregion"
      aggregate_wait_seconds: 10
      aggregate_interval_seconds: 60
      repeat_interval_seconds: 3600

      aggregates:

        # 对于 OB日志告警，按照 告警类型，日志错误码，OB集群 进行聚合
        - match:
            alarm_type: "ob_log_alarm"
          group_by:
            - "alarm_type"
            - "ob_error_code"
            - "obregion"
          aggregate_wait_seconds: 10
          aggregate_interval_seconds: 60
          repeat_interval_seconds: 3600
```

- aggregate_wait_seconds 是首次产生高警时等待时长，该时间内产生的相同聚合维度的告警将会聚合为一条告警消息。
- aggregate_interval_seconds 是相同聚合维度的聚合周期，即：多久新产生一条聚合的告警消息。
- repeat_interval_seconds 是同一告警（相同告警 id，告警未恢复是 id 不会递增）的发送周期，即：同一告警要在下个 repeat_interval_seconds 周期才会被聚合。

告警订阅

告警订阅功能是为了方便将告警消息发送给不同的用户。

首先会将告警项划分为不同的分组，订阅时可直接订阅该告警分组，当前 OCP 划分了如下 6 个告警分组：

- ocp：OCP 相关告警项。
- dba：OceanBase 数据库相关告警项。
- info：操作类（Info 级别）告警项。
- oms：OMS 应用告警项。
- backup：备份恢复告警项。
- dev：运维相关告警项。

可以按照集群、级别订阅告警，将不同的告警发送到不同的告警通道。告警通道定义了如何将告警发送出去，现在支持通过 bash/python 脚本发送，或通过 HTTP API 发送；也支持在通道上设置限流策略，避免发送过多告警。

10.7. 安全审计

数据加密和访问控制可以大幅降低安全风险，但对于具备权限的用户，仍然需要记录其操作，以防止用户登录信息泄露，或者访问权限被滥用。审计功能可以加强企业对数据安全、合规等方面的要求，是跟踪用户行为最主要的工具。

审计开启

通过配置项 audit_trail 开启审计功能，执行完后立即生效。

- NONE：关闭审计。
- OS：审计记录写本地文件。
- DB：审计记录写内部表。
- DB,EXTENDED：审计记录写内部表且记录包含执行的 SQL 语句。

管理员用户拥有最高权限，能执行很多操作，所以针对管理员有单独配置审计。租户配置项 audit_sys_operations 决定是否对管理员的操作进行审计记录。

设置审计规则

通过内置的审计管理用户 ORAAUDITOR 可以配置审计规则，包括：

- 语句审计：对特定的操作进行审计，不指定具体对象，可以指定对具体用户或对所有用户生效。
- 对象审计：对特定的对象上执行的特定操作进行审计，可以指定对具体用户或对所有用户生效。

审计规则是通过 DDL 语句 audit/noaudit 语句来配置，审计规则也是一种 schema 对象。

审计流程

审计检查的时机位于一条用户 SQL 执行完毕后准备回包之前。审计检查的流程如下：

1. 检查对当前用户是否进行审计，根据租户、用户名、配置项综合判断。
2. 解析 SQL 语句中可被审计的操作，一条 SQL 语句可以包含多个不同的操作，例如 `insert into t1 select * from t2, t3` 中包含了 `(insert, t1), (select, t2), (select t3)` 3 个操作。
3. 对每个操作单独判断是否命中任意一条审计规则。
4. 每个命中规则的操作单独产生一条审计记录。根据配置项，审计记录可以记录在内部表或文件中。

审计记录

审计文件会放在 `audit` 目录中，`observer_${pid}_${timestamp}.aud` 文件名形式存储。写文件使用了 ObLogger 提供的接口，其功能和其他系统日志相同，当文件大小达到 256M 时，会将其进行切分。

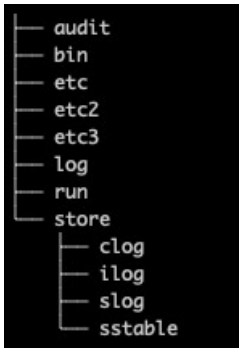
审计记录写入内部表有如下特性：

- AUDIT_TRAIL 取值 DB 时不记录用户 SQL，取值 DB,EXTENDED 会记录用户 SQL。
- 审计记录的插入独立于用户事务，即使用户事务回滚，审计记录依然保留。
- 审计写表失败时，会尝试审计写文件方式保存审计记录。

11.OBServer 节点架构

11.1. OBServer 安装目录结构

OBServer 工作目录下通常有 audit、bin、etc、etc2、etc3、log、run、store 这 8 个目录，但这 8 个目录并非都是安装必须的。在启动 OBServer 前我们需要保证 etc、log、run、store 这 4 的目录存在，同时 store 下应该有clog、ilog、slog、sstable 这 4 个目录。etc2、etc3 是备份配置文件用的，由 OBServer 创建。audit 下存放的是审计日志，也由 OBServer 创建。bin 目录用于存放 observer binary 文件，方便后续定位。



配置文件目录

etc、etc2、etc3 都是配置文件目录。这三个目录里的内容是完全一致的，区别是后两个目录是 OBServer 创建的，第一个目录是启动前需要准备的。etc2 和 etc3 是配置文件额外保存的目录，由配置项 config_additional_dir 控制。当配置修改以后，除了会写标准的 etc/observer.config.bin 以外，还会额外在这些目录创建配置项文件。server 启动不会读取额外目录的配置项文件，只是作为额外备份。额外目录如果有权限会自动创建，没有权限则日志中报 ERROR。

配置文件中储存了集群和租户级配置项的增量信息。集群配置可以通过连接到数据库上执行 alter system 修改，也可以通过命令行传入配置项。命令行传入配置项会覆盖原有配置。命令行可以通过以下参数修改 OBServer 配置项：

```
-z,--zone ZONE            zone name 节点所在的zone的名字
-p,--mysql_port PORT      mysql port SQL服务协议端口号
-P,--rpc_port PORT        rpc port 集群内部通信的端口号
-n,--appname APPNAME      application name 本OceanBase集群名
-c,--cluster_id ID        cluster id 本OceanBase集群ID
-d,--data_dir DIR         OceanBase data directory 存储sstable等数据的目录
-i,--devname DEV          net dev interface 服务进程绑定的网卡设备名
-r,--rs_list RS_LIST      root service list root service列表
-l,--log_level LOG_LEVEL server log level 系统日志级别
-o,--optstr OPTSTR        extra options string 额外配置项格式为 配置名=新值,配置名=新值
```

日志文件目录

log 目录是存放运行日志的目录，里面包含了 observer 日志、RS 日志和选举日志。单个日志文件大小为 256M。

当 `enable_syslog_recycle` 设置为 true、`max_syslog_file_count` 大于 0 时日志文件会自动回收。其中，`enable_syslog_recycle` 用于设置是否开启自动回收，`max_syslog_file_count` 用于设置每种日志的最大日志数量。

数据文件目录

store 是数据文件目录，包含了 clog、ilog、slog、sstable 这 4 个子目录。其中 clog、ilog、slog 是事务日志目录，slog 存储静态数据写入的事务日志，clog 存储动态数据写入的事务日志，ilog 存储日志目录。sstable 是基线数据目录。sstable 目录下会有一个 block_file，这个文件在 observer 启动后就会被创建，文件的大小由 `datafile_size` 或 `datafile_disk_percentage` 控制。

> 说明
>
> - `datafile_size` 用于设置数据文件的大小。
> - `datafile_disk_percentage` 表示占用 `data_dir` 所在磁盘（ `data_dir` 所在磁盘将被 OceanBase 数据库系统初始化用于存储数据）总空间的百分比。
>
> 有关两个配置项的详细介绍，参见《OceanBase 数据库参考指南》中的 **系统配置项** 章节。

11.2. 配置文件

OceanBase 数据库的配置项分为集群级配置项和租户级配置项。OBServer 会将所有的配置项序列化后保存到工作目录下的配置文件 `etc/observer.conf.bin` 中，之后在这个工作目录下启动 OBServer 都会读取这个配置文件。

配置项查询

- 普通租户查询配置项，通过虚拟表的形式展示当前租户的配置项信息，包含集群配置和租户配置。其中集群级配置项都是只读的，租户级配置项根据配置项属性决定是否可以被修改。

  普通租户使用 `SHOW PARAMETERS` 语句查看本租户级配置项信息的 SQL 语句如下：

  ```
  SHOW PARAMETERS [SHOW_PARAM_OPTS]
  ```

- 系统租户查询配置项

  系统租户可以使用 `SHOW PARAMETERS` 语句查看集群级配置项和租户级配置项信息。并可通过增加 `TENANT` 关键字信息查看指定租户的配置项。

  - 如果没有指定 `TENANT`，则展示系统租户级配置项信息，包含集群配置和租户配置。集群配置和租户配置都根据其属性决定是否可以被修改。
  - 如果指定为某个普通租户，则和普通租户展示的配置项内容完全一致。

  SQL 语句如下：

  ```
  SHOW PARAMETERS [SHOW_PARAM_OPTS] TENANT = tenant_name
  ```

配置项的详细信息如下表所示：

列名	含义
svr_ip	机器 IP
svr_port	机器端端口
name	配置项名
value	配置项值
type	配置项数据类型（NUMBER,STRING,CAPACITY...）
info	配置项解释
section	配置项分类
scope	配置项范围属性（Tenant|Cluster）
source	当前值来源（Tenant|Cluster|CommandLine|ObAdmin|File）
editable	是否可修改
dynamic	是否支持在线动态修改

配置项修改

> **说明**
>
> 集群级配置项仅支持在系统租户下配置。

MySQL 模式下，修改配置项的 SQL 语法如下：

```
ALTER SYSTEM [SET] parameter_name = expression [PARAM_OPTS]
```

系统租户下，可以通过指定 `TENANT` 关键字来修改全部或指定租户的配置项。SQL 语法如下：

```
ALTER SYSTEM [SET] parameter_name = expression [PARAM_OPTS] TENANT = all|tenant_name
```

> **说明**
>
> 执行成功以后，所有被指定的租户的配置项均会被修改。

Oracle 模式下，修改配置项的 SQL 语法如下：

```
ALTER SYSTEM SET parameter_name = expression [PARAM_OPTS]
```

> **说明**
>
> `PARAM_OPTS` 表示设置配置项所使用的其它限定条件，比如：指定 Zone、指定 OBServer 等。具体请参见《 SQL 参考（MySQL）》和《 SQL 参考（Oracle）》。

配置项存储

OBServer 持久化使用配置文件，运行时则在会维护内部表。

配置项有两种实体表：集群级配置项表和每个租户的租户级配置项表。当用户执行 SQL 修改配置项时修改集群级配置项时就写入集群级配置项表，否则写入对应的租户级配置项表。读取时，集群级配置项只需要使用集群级配置项表，租户级配置项则需要读取租户级配置项表。

配置文件格式

```
ObRecordHeader
[CLUSTER]
CLUSTER_PARAMETER_UPDATES
[TENANT_ID1]
TENANT_ID1_PARAMETER_UPDATES
[TENANT_ID2]
TENANT_ID2_PARAMETER_UPDATES
```

更新

更新是指各个 OBServer 根据需要，从内部表或者配置文件读取持久化的增量修改加载到内存中的过程。当 OBServer 进程刚启动的时候，它会从配置文件中加载配置项；正常运行过程中如果用户主动修改了配置项，OBServer 则会从内部表中进行加载。

从配置文件中加载

OBServer 启动的时候配置项会优先在配置文件中进行加载。加载配置项的顺序是：

1. 根据代码内容和运行环境计算出默认配置。
2. 使用配置文件中的配置逐项设置。
3. 租户级配置项是从集群级配置项克隆一份后再使用租户配置值进行覆盖。
4. 根据命令行参数传入的配置项值进行设置。

从内部表加载

配置项最主要的更新场景还是从内部表进行加载，也就是当前已经拥有一份可正常运行的配置项的同时，需要修改某些配置项的值。

当一条 `ALTER SYSTEM SET` 的 SQL 通过验证以后，对应配置项所在的内部表会进行更新，并且 RS 会通过心跳的方式通知各 OBServer 需要重新读取配置项内部表以更新它们的配置项。当 OBServer 收到 RS 的通知以后，后台线程会启动一个“更新配置项”的任务。该任务主要流程是：

1. 读取内部表中的配置项值。
2. 克隆一份当前配置项。
3. 将内部表中的配置项逐项应用到克隆出来的配置项副本中。
4. 结束后检查副本配置项的值是否合法。
5. 将内部表值持久化到配置文件。
6. 使用配置项副本代替原来配置项。

租户级配置项的流程和集群级配置项一致，唯一的区别是配置项所在的内部表不同。

11.3. observer 线程模型

11.3.1. 线程简介

OBServer 是单进程的。OBServer 中会起很多种线程，大的方面，可以把线程分成工作线程和后台线程。

工作线程

sql/transaction worker 处理 SQL 和事务请求的线程。是分租户的，也即每个租户都有自己的一套 sql/transaction worker，也称租户工作线程。

后台线程

OBServer 包含如下后台线程：

- net io: 处理网络 io 的线程。
- disk io: 处理磁盘 io 的线程。
- dag 线程: 用于 partition 的转储，合并，迁移等任务的执行。
- clog writer: 写 clog 的线程。
- election worker: 选举线程。
- misc timer：包括多个后台定时器线程，主要负责清理资源。

除此之外还有一批 RootServer 专有的线程，这里不再做细分。最后还有一些特定用途的后台线程，在进一步深入了解 OBServer 之前，可以先忽略。

11.3.2. 工作线程

处理 SQL 和事务请求的线程也称为工作线程，是分租户的，也即每个租户都有自己的一套 sql/transaction worker。

多租户线程池

线程池简介

多租户共享一个大线程池，租户从线程池中申请工作线程。线程池跟随多租户进行初始化和销毁，多租户初始化时，预先申请一定数目的线程，后续在租户运行过程中，还支持动态的扩展线程池。

初始线程池大小与 OBServer 的 CPU 数，系统租户预留线程数，虚拟租户预留线程数有关，部分可通过配置项配置。后续扩展线程池时的线程数上限受 observer 线程数上限和多租户线程数上限共同限制，可通过配置项配置。

线程池相关配置项

- _ob_max_thread_num

  决定 observer 的线程数上限。

  默认值为 4096，范围 4096 ~ 10000，非动态生效。

- system_cpu_quota

  决定系统租户的虚拟 CPU 个数，与机器物理 CPU 无直接关系，仅影响多租户线程池的初始大小和上限大小。

  默认值为 10，范围 0 ~ 16，非动态生效。

- server_cpu_quota_min

  决定 500 租户的最小虚拟 CPU 个数，与机器物理 CPU 无直接关系，仅影响多租户线程池的初始大小。

  默认值为 2.5，范围 0 ~ 16，非动态生效。

- server_cpu_quota_max

  决定 500 租户的最大虚拟 CPU 个数，与机器物理 CPU 无直接关系，仅影响多租户线程池的上限大小。

  默认值为 5，范围 0 ~ 16，非动态生效。

- election_cpu_quota

  决定 election 租户的虚拟 CPU 个数，与机器物理 CPU 无直接关系，仅影响多租户线程池的初始大小和上限大小。

  默认值为 3，范围 0 ~ 16，非动态生效。

- location_cache_cpu_quota

  决定 location cache 租户的虚拟 CPU 个数，与机器物理 CPU 无直接关系，仅影响多租户线程池的初始大小和上限大小。

  默认值为 5，范围 0 ~ 16，非动态生效。

租户线程

租户线程构成

单个租户的线程包含，处理嵌套请求的 7 个专有线程，处理一般请求的若干个普通线程。线程可处于活跃，挂起两种状态。由于嵌套请求的专用线程对外基本无感知，以下介绍只涉及普通线程。

活跃线程数概念

活跃线程表示能正常处理请求的线程，与挂起线程区分，活跃线程数用于限制单个租户的 CPU 使用。租户运行时会维持活跃线程数恒定，租户的活跃线程数受配置项和 unit 规格共同决定。活跃线程数 = unit_min_cpu * cpu_quota_concurrency。

大查询的线程挂起逻辑

用户发给 observer 的 SQL，可以粗略分为两类：一类 SQL 访问和操作的数据量小，所以执行很快；另一类 SQL 要访问大量的数据或者要写入大量数据，所有执行耗时长。我们把第二类耗时长的 SQL 叫做大查询。

大查询的特殊处理基于一个直观的用户效用假设，从影响用户数和延迟的 QoS 要求来合理猜想：1000 个小查询被延迟的影响要远大于 1 个大查询被延迟。换言之，与其花 1s 时间处理一个大查询，不如把同样的 CPU 时间用来处理 1000 个小查询。

请求被判定为大查询的条件是，处理时间超过大查询阈值，此阈值可通过配置项调整。

租户线程中持有被判定为大查询请求的线程，一部分可直接获得继续执行权，其余的需要挂起等待。可获得继续执行权的线程的比例由配置项决定。

最大线程数概念

为了维持租户活跃线程数恒定，同时考虑到大查询线程挂起的发生，租户就需要动态的从多租户线程池中申请线程。最大线程数用于限制单个租户的内存开销，每个租户总共可持有的最大线程数受配置项和 unit 规格共同决定。最大线程数 = unit_max_cpu * worker_per_cpu_quota。

相关配置项

- cpu_quota_concurrency

  决定租户活跃线程数与租户 unit 规格的倍数关系。

  默认值为 4，范围 1 ~ 10，动态生效。

- worker_per_cpu_quota

  决定租户最大线程数与租户 unit 规格的倍数关系。

  默认值为 10，范围 2 ~ 20，动态生效。

- large_query_worker_percentage

  决定租户线程中享有大查询继续执行权的线程的百分比。

  默认值为 30，范围 0 ~ 100，动态生效。

- large_query_threshold

  请求判定为大查询的处理时间阈值。

  默认值为 5s，范围 1ms ~ +∞，动态生效。

日志诊断

通过 `grep 'dump tenant info' observer.log` 指令可以获得租户的工作线程数情况和请求排队情况等。

```
grep 'dump tenant info.*tenant={id:1002' log/observer.log.*
[2021-05-10 16:56:22.564978] INFO  [SERVER.OMT] ob_multi_tenant.cpp:803
[48820][2116][Y0-0000000000000000] [lt=5] dump tenant info
(tenant={id:1002, compat_mode:1, unit_min_cpu:"1.00000000000000000e+01",
unit_max_cpu:"1.50000000000000000e+01", slice:"0.00000000000000000e+00",
slice_remain:"0.00000000000000000e+00", token_cnt:30, ass_token_cnt:30,
lq_tokens:3, used_lq_tokens:3, stopped:false, idle_us:4945506,
recv_hp_rpc_cnt:2420622, recv_np_rpc_cnt:7523808, recv_lp_rpc_cnt:0,
recv_mysql_cnt:4561007, recv_task_cnt:337865, recv_large_req_cnt:1272,
tt_large_quries:3648648, actives:35, workers:35, nesting workers:7,
lq waiting workers:5, req_queue:total_size=48183 queue[0]=47888 queue[1]=0 queue[2]=242 que
ue[3]=5 queue[4]=48 queue[5]=0 ,
large queued:12, multi_level_queue:total_size=0 queue[0]=0 queue[1]=0 queue[2]=0 queue[3]=0
queue[4]=0 queue[5]=0 queue[6]=0 queue[7]=0 ,
recv_level_rpc_cnt:cnt[0]=0 cnt[1]=0 cnt[2]=0 cnt[3]=0 cnt[4]=0 cnt[5]=165652 cnt[6]=10 cnt
[7]=0 })
```

此章节只介绍工作线程相关情况，且只列出对诊断有帮助的项目。

- unit_min_cpu 和 unit_max_cpu：租户 unit 规格。
- token_cnt：租户目标达到的活跃线程数。
- ass_token_cnt：租户实际的活跃线程数。
- lq_tokens：租户线程可享有大查询请求继续执行权的线程数。
- used_lq_tokens：租户线程已经享有大查询请求继续执行权的线程数。
- lq waiting workers：租户线程处理大查询请求并被挂起的线程。

11.3.3. 后台线程

下表列举出 OBServer 的一部分后台线程及其功能描述。大部分场景下用户无需关注其实现细节。

> **说明**
>
> 在 OBServer 版本迭代中会对后台线程持续优化，因此有些后台线程会在版本升级的过程出现消失、合并等情况，也有可能会出现新的后台线程。

线程名	归属模块	线程数量	功能描述
TsMgr	Trx	1	用于本地 gts cache 的刷新。
WeakReadSvr	Trx	1	用于计算本地 server 级别备机读时间戳。
HAGtsSource	Trx	1	用于 ha_gts 的刷新。 刷新频率由配置项 gts_refresh_interval 控制。
TransCheckpoint Worker	Trx	1	用户事务的 checkpoint 操作。
GCPartAdpt	Trx	1	定期检查 all_tenant_gc_partition_info 表，获取分区的 gc 情况，用于 GC 和事务 2PC 的推进。

线程名	归属模块	线程数量	功能描述
PartWorker	Trx	1	遍历本机所有的 partition，做以下操作： 1. 每隔 50ms 尝试写事务层 checkpoint log。 2. 每隔 15s 计算迁移的快照点。 3. 每隔 10s 驱动本 partition 上所有活跃事务，为其检查 scheduler 的状态，用于事务上下文的 GC。 4. 每隔 10s 检查下该 partition 复制表的状态。 5. 每隔 10s 驱动刷新本 partition 上缓存的 clog 日志。
LockWaitMgr	Trx	1	用于热点行场景下，等锁队列兜底的唤醒保障，比如语句超时，session 被 kill。
ClogAdapter	Trx	8	用于事务层异步提交 clog。
ObTransService	Trx	6	主要负责以下功能： • 用于事务提交过程处理 error msg • 用于语句回滚失败之后的重试。 • 用于提前解行锁场景，后继事务的唤醒操作。
GCCollector	storage	1	用于定期检查本机所有 partition 是否能够 GC，如果可以则触发 GC 动作。
RebuildSche	storage	1	定期驱动需要 rebuild 的 partition 任务。
PurgeWorker	stroage	1	后台 memtable 的标记删除操作。
DAG	storage	16	dag 的工作线程，用于 partition 的转储、合并、迁移等任务的执行。
DagScheduler	storage	1	dag 的调度线程。
STableChecksumUp	storage	1	用来 batch 提交汇报 sstable checksum 的任务。

线程名	归属模块	线程数量	功能描述
PartitionScheduler	storage	2	用来调度转储和调度合并任务。
MemstoreGC	storage	1	用来回收转储预热完成之后的 frozen memstore。
ObStoreFile	storage	1	用来检查坏块和宏块回收。
PartSerCb	storage	40	用于 partition leader 上任、卸任任务处理（20 个线程）。
ObPartitionSplit Worker	storage	1	用于处理 partition 分裂的后台任务。
FreezeTimer	storage	1	检查 memstore 内存，触发冻结。
RebuildTask	storage	1	调度 D 副本拉取数据。
IndexSche	storage	1	调度建索引任务。
FreInfoReload	storage	1	同步内部表中 major freeze 相关信息。
TableMgrGC	storage	1	收 memtable，释放内存。
BackupInfoUpdate	storage	1	用于定时刷 backup 的相关数据。
LogDiskMon	storage	1	用于检测系统多盘的情况。
KVCacheWash	storage	1	用于 kvcache 的内存 wash。 通过配置项 _cache_wash_interval 来控制检测周期，默认是 200ms，范围 [1ms,1m]。

线程名	归属模块	线程数量	功能描述
KVCacheRep	storage	1	用于整理 cache 的内存碎片回收内存。 通过配置项 _cache_wash_interval 来控制检测周期，默认是 200ms，范围 [1ms,1m]。
RSMonitor	RS	1	用来监控当前 RS 的状态。
CacheCalculator	RS	1	用来给 location_cache 和 schema_cache 预留内存。
ObDailyMergeScheduler	RS	1	用来调度每日合并。
ObEmptyServerChecker	RS	1	用来检测 OBS 上面是否不存在副本。
ObFetchPrimaryDDLOperator	RS	1	备库逻辑同步主库系统租户 schema。
ObFreezeInfoUpdater	RS	1	推动冻结进度和管理快照点回收点。
ObGlobalIndexBuilder	RS	1	调度全局索引的构建。
ObGlobalMaxDecidedTransVersionMgr	RS	1	统计全局最大事务版本号。
ObLeaderCoordinator	RS	1	管理集群中副本 leader 分布。
ObLogArchiveScheduler	RS	1	负责日志归档。
ObMajorFreezeLauncher	RS	1	负责每日发起定时合并。

线程名	归属模块	线程数量	功能描述
ObPartitionSpliter	RS	1	负责调度分区合并。
ObRebalanceTaskMgr	RS	1	负责调度负载均衡任务的执行。
ObRootBackup	RS	1	负责调度备份任务。
ObRootBalancer	RS	1	负责生成负载均衡任务。
ObRsGtsMonitor	RS	1	监控 gts 实例的分布情况的，并基于 gts 副本的分布，生成迁移或补 gts 副本任务。
ObRsGtsTaskMgr	RS	1	执行 gts 副本任务的线程。
ObHeartbeatChecker	RS	1	负责检查和 OBS 之间的心跳状态。
ObStandbyClusterSchemaProcessor	RS	1	备库副本同步主库的普通租户 schema。
ObRestoreScheduler	RS	1	负责恢复流程的调度。
ObWorkQueue	RS	4	执行 RS 的异步任务。 线程数通过配置项 rootservice_async_task_thread_count 来控制，范围 [1,10]。
ObAsyncTaskQueue	RS	16	执行构建索引的任务。
ObSwitchTaskQueue	RS	6	主备库切换的任务队列，线程数通过配置项 switchover_process_thread_count 来控制，范围 [1, 1000]。
LocalityReload	RS	1	定期刷新 locality 信息。

线程名	归属模块	线程数量	功能描述
RSqlPool	RS	1	备库用来定期维护主库的 rs_list。
ServerTracerTimer	RS	1	用来维护和其他 server 状态的定时任务。
LogEngine	CLOG	1	统计监控 clog 盘空间使用情况，执行 clog 文件回收操作。
CLGWR	CLOG	1	负责 clog item 写盘。
ClogHisRep	CLOG	3	partition 上下线时更新 clog history 内部表。
BatchSubmitCtx	CLOG	1	负责拆分一阶段优化的事务。
LogScanRunnable	CLOG	4	observer 启动时负责扫描所有 clog 文件。
LogStateDri	CLOG	1	负责 clog 模块主备角色切换，检查副本级联状态。
ObElectionGCThread	election	1	负责 election 对象内存回收。
Blacklist	CLOG	1	负责探测与通信目的端 server 之间的网络是否联通。
BRPC	CLOG	5	负责后台执行 batch_rpc 聚合包发包。
CLOGReqMinor	CLOG	1	根据 clog 磁盘空间的情况触发 minor freeze，加快 clog 回收。
LineCache	CLOG	1	用于优化历史数据的同步，Liboblog 场景使用。
EXTLogWash	CLOG	1	频率：100ms。
CLogFileGC	CLOG	1	用于用于定期回收 clog 文件。

线程名	归属模块	线程数量	功能描述
CKPTLogRep	CLOG	1	用于日志副本的 checkpoint 操作。
RebuildRetry	CLOG	1	用于重试 rebuild partition 的任务。
ReplayEngine	CLOG	物理核	用于备机 CLOG 的回放。
PxPoolTh	SQL	0	并行执行的线程池，线程数通过系统变量来控制 parallel_max_servers，默认为 0。
sql_mem_timer	SQL	1	用于自动内存管理。
OmtNodeBalancer	common	1	负责后台刷新多租户信息。
MultiTenant	common	1	负责刷多租户 CPU 配比，用于资源调度。
SignalHandle	common	1	信号处理线程。
LeaseUpdate	common	3	负责 rs 与各个 server 直接的 lease 监控。
RsDDL	common	1	负责 RS 的 DDL。
MysqlIO	common	12	负责 MySQL 的链接处理线程。
all_meta_table	common	8	用于 PG 或者 partition 状态的汇报。
all_pg_partition_meta_table	common	8	用于 PG 内 partition 的状态的汇报。
TimerMonitor	common	1	监控内部定时线程动作，对执行时间异常的任务报警。
ConfigMgr	common	1	用于配置项的刷新。
OB_ALOG	common	1	用于系统异步日志的日志落盘。

线程名	归属模块	线程数量	功能描述
ELE_ALOG	election	1	用于选举模块异步日志的日志落盘。

11.4. 日志

observer 的日志存放在 `observer` 的安装目录的 `log` 目录下面，分为三种日志 `observer.log` 、`rootservice.log` 和 `election.log` ，分别对应 observer 日志、RS 日志和选举日志。

对于每一种日志，如 `observer.log` ，按照文件名大致分为以下几种：

- observer.log
- observer.log.20210901123456
- observer.log.wf
- observer.log.wf.20210901123456

当 `observer.log` 达到 256M 时，会将其 rename 为第二种日志，后面的数字为时间戳。wf 日志的含义见下文 enable_syslog_wf 配置项。

日志格式

这里以一条日志举例说明。

```
[2021-09-01 11:31:18.605433] INFO  [STORAGE] ob_pg_sstable_garbage_collector.cpp:170 [38715
][0][Y0-0000000000000000-0-0] [lt=15] [dc=0] do one gc free sstable by queue(ret=0, free ss
table cnt=0)
```

对应的日志信息如下：

[时间] 日志级别 [所属模块] 文件:行号 [线程id][0][trace_id] [lt=上条日志耗费时间（微秒）][丢弃日志数量]

日志级别

从低到高有 6 种，DEBUG、TRACE、INFO、WARN、USER_ERR、ERROR。

其中 ERROR 日志比较特殊，会将打日志时所在的堆栈打印出来（需要通过符号表解析）。

> **注意**
>
> 开启 DEBUG 日志将耗费大量资源，在较新版本中，DEBUG 日志在 release 编译下会自动去掉，即使开启也无法生效。

所属模块

有很多种，可以大致区分日志所属模块。

线程id

打印日志所在的线程的 ID，可以用于跟踪对应线程的行为，对排查线程 hang 住、超时等问题比较有用。

trace_id

OceanBase 数据库内部的 SQL 级别的 ID，默认为 Y0-0000000000000000-0-0，可以通过 trace_id 来找到一条 SQL 的执行过程，是排查问题的重要手段。

相关配置项

以下所有配置项均为集群级别，需要在系统租户下使用。可以通过以下方法修改。

```
alter system set enable_syslog_recycle = False;
```

- enable_syslog_recycle

  是否开启日志回收，默认为 False。

  开启后将会自动删除多余的日志，具体逻辑详见 max_syslog_file_count。

- enable_syslog_wf

  是否启用 wf 日志，默认为 True。

  开启后会每种日志中 WARN 级别以上的日志备份到 wf 日志中，如 observer.log.wf。

- enable_async_syslog

  是否启用异步写日志功能，默认为 True。

  关闭后将使用同步方式写日志，可以保证 observer 宕机前写完所有日志，但性能较差，不建议关闭。

- max_syslog_file_count

  每种日志的最大日志数量，默认为 0，当且仅当该配置项大于 0 且 enable_syslog_recycle 为 True 时生效。

  > 注意
  >
  > 尽管该配置项的范围是 [0, +∞)，但 observer 源码中约束了该值不能超过 MAX_LOG_FILE_COUNT，即 10240，超过时依然可以设置成功但实际生效值为 MAX_LOG_FILE_COUNT。

- syslog_io_bandwidth_limit

  日志限流量，默认为 30M。

  > 注意
  >
  > 当日志打印速度超过限制时，将打印以下信息。
  >
  > ```
  > REACH SYSLOG RATE LIMIT
  > ```

- syslog_level

  打印的日志的最低级别，日志级别见上文，默认为 INFO。

11.5. 内存管理

11.5.1. 内存管理概述

OceanBase 数据库是多租户设计的数据库，同一个进程会运行着多个租户的请求，从租户资源划分上可以分为三类，500 租户内存、系统租户内存、业务租户内存。

- 500 是个特殊的虚拟租户，共享性的、非实体租户消耗的内存都被 OceanBase 数据库划归 500 租户。
- 系统租户是 OceanBase 数据库自动创建的第一个实体租户，管理着集群相关的内部表，这些内部表上的请求触发的内存就划归到了 sys 租户。
- 业务租户就是集群安装后有 DBA 创建的跑业务流量的实体租户，这些租户的请求触发的内存就划归租户自己。

11.5.2. 内存相关参数

OceanBase 数据库支持通过参数来设置和管理内存的使用。

> **说明**
>
> 有关以下提到的参数的详细介绍，参见《OceanBase 数据库参考指南》中 **系统配置项** 章节。

OBServer 总内存

表示 OBServer 所有内存的上限，即所有租户累加不可能超过的上限，由两个参数控制：

- memory_limit_percentage：物理机或者容器物理内存的百分比。
- memory_limit：绝对数值。

当 memory_limit 非 0 时，使用 memory_limit 作为上限，否则使用 memory_limit_percentage 计算后的值作为上限。

高优先级预留内存

表示给一些特别重要的底层模块预留的一块空间。

可以通过 memory_reserved 参数进行设置。

500 租户内存

表示 500 租户预留内存，可以通过 system_memory 参数进行设置。

注意它不是 limit 的语义，目前的实现上 500 租户实际可使用内存是可能突破 system_memory 的，但其他租户可使用内存不会超过 OBServer 总内存（即 system_memory 的值）。

业务租户内存

业务租户内存由租户创建 unit 时指定。命令示例如下。

```
create resource unit if not exists sp_trans_test max_cpu=1, min_memory='1G',max_memory='1G'
,max_disk_size='1G',max_iops=1000,max_session_num=1000;
create resource pool if not exists sp_trans_test unit='sp_trans_test', unit_num=1;
create tenant if not exists sp_trans_test RESOURCE_POOL_LIST=('sp_trans_test') set ob_tcp_i
nvited_nodes='%';
```

OceanBase 数据库支持动态调整 unit 规格（缩容需谨慎）。

```
alter RESOURCE UNIT sp_trans_test max_memory='2G',min_memory='2G';
```

可以通过系统租户内部表查询 unit 规格。

```
MySQL [oceanbase]> select *From __all_resource_pool where tenant_id = 1;
+----------------------------+----------------------------+-----------------+----------+--
----------+---------------+-----------+-----------+--------------+--------------------+
| gmt_create                 | gmt_modified               | resource_pool_id | name     | u
nit_count | unit_config_id | zone_list | tenant_id | replica_type | is_tenant_sys_pool |
+----------------------------+----------------------------+-----------------+----------+--
----------+---------------+-----------+-----------+--------------+--------------------+
| 2021-09-09 16:38:33.503570 | 2021-09-09 16:38:33.506882 |               1 | sys_pool |
1 |             1 | z1        |         1 |            0 |                  0 |
+----------------------------+----------------------------+-----------------+----------+--
----------+---------------+-----------+-----------+--------------+--------------------+
MySQL [oceanbase]> select * From __all_unit_config where unit_config_id = 1;
+----------------------------+----------------------------+----------------+---------------
--+---------+---------+-------------+-------------+----------+----------+---------------+--
-------------------+
| gmt_create                 | gmt_modified               | unit_config_id | name
| max_cpu | min_cpu | max_memory  | min_memory  | max_iops | min_iops | max_disk_size | max
_session_num     |
+----------------------------+----------------------------+----------------+---------------
--+---------+---------+-------------+-------------+----------+----------+---------------+--
-------------------+
| 2021-09-09 16:41:52.803406 | 2021-09-09 16:41:52.803406 |              1 | sys_unit_confi
g |       5 |     2.5 | 17179869184 | 12884901888 |    10000 |     5000 | 4398046511104 | 9
223372036854775807 |
+----------------------------+----------------------------+----------------+---------------
--+---------+---------+-------------+-------------+----------+----------+---------------+--
-------------------+
```

租户内存分类

租户内存分为两类，memstore 与 SQL 执行。

- memstore

  主要用于保存数据库增量数据。memstore 上限由参数 memstore_limit_percentage 控制，它表示租户 memStore 最多占租户总内存上限的百分比。freeze_trigger_percentage 参数用于控制转储时机，它表示当 memstore 内存达到 memstore 上限的某个百分比时触发转储，租户 memstore 的使用可以通过内部表查看 。

```
MySQL [oceanbase]> select * From gv$memstore;
+-----------+----------------+-------+-----------+-----------+---------------------+-----
----------------+------------+
| TENANT_ID | IP             | PORT  | ACTIVE    | TOTAL     | FREEZE_TRIGGER      | MEM_
LIMIT           | FREEZE_CNT |
+-----------+----------------+-------+-----------+-----------+---------------------+-----
----------------+------------+
|         1 | 100.81.113.215 | 46824 | 169792200 | 171966464 |          5153960700 |
10307921440 |          0 |
|       500 | 100.81.113.215 | 46824 |         0 |         0 | 3689348814741910300 | 7378
697629483820640 |          0 |
|      1001 | 100.81.113.215 | 46824 |  33539200 |  35651584 |           382730200 |
858993440 |          0 |
|      1002 | 100.81.113.215 | 46824 |         0 | 228589568 |         34359738350 |
68719476720 |          0 |
+-----------+----------------+-------+-----------+-----------+---------------------+-----
----------------+------------+
```

- **SQL 执行**

  表示 SQL 执行所需要的内存以及为了提高吞吐做的缓存，比如 SQL 执行计划等。

  ob_sql_work_area_percentage 是 SQL 工作区内存参数，表示 SQL 可能耗费大量内存的阻塞性算子所使用的内存。

sys 租户内存

前面提到，sys 租户就是实体租户，它的管理机制与业务租户完全一致。

超卖

OceanBase 数据库保证一个 OBServer 的资源使用尽量到达 soft_limit，并且严格小于 hard_limit，相关参数如下。

```
resource_hard_limit=100
resource_soft_limit=50
```

分配 unit 时会根据 resource_hard_limit 来决定内存是否超卖（默认值为 100，超过表示允许超卖）。 unit 有 min_memory 和 max_memory 两个选项，一般部署时 min_memory 和 max_memory 总是相同的。如果需要超卖，可以保持 min_memory 不变，max_memory 调大。

- 没有打开 enable_global_freeze_trigger 之前，一定不要调大 resource_hard_limit 和 max_memory，否则会出现所有租户内存加起来超过物理内存的问题。一旦出现这种情况，只能通过 ob_admin 临时调大 memory_limit 解决。
- 只能在 1:1:1 的环境下用，否则如果有迁移，很容易出现占用内存多的 partition 迁到一台机器上，导致被迁入的机器内存爆的问题。

kvcache

OBServer 的 kvcache 会尽可能地多用内存，并且能自动挤占，kvcache 在以下几种情况下会触发 wash：

- 内存申请不出来时。
- 进程剩余内存不足 cache_wash_threshold 时。
- 租户内存（max_memory）使用到达 90% 时。

11.5.3. 内存相关内部表

OceanBase 数据库支持通过内部表来查看内存的使用情况。

视图	描述	
__all_virtual_tenant_ctx_memory_info	context 级别的统计信息（context 是面向开发者的概念），它是 2M 粒度的统计。	
__all_virtual_memory_info	OBServer 内存标签的统计信息。	
__all_virtual_tenant_memstore_info	memstore 统计信息。	
__all_virtual_kvcache_info	kvcache 统计信息。	

11.5.4. 内存相关日志

OceanBase 数据库支持通过日志来查看内存的使用情况。

租户统计

对租户的总内存进行统计，每行代表一个租户。

```
[2021-09-09 17:13:17.232782] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 501, limit: 1,073,741,824 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.232962] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 502, limit: 1,073,741,824 hold: 4,194,304 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.233133] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 503, limit: 1,073,741,824 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.233308] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 504, limit: 1,073,741,824 hold: 8,388,608 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.233485] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 505, limit: 1,073,741,824 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.233659] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 506, limit: 4,294,967,296 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.233832] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 507, limit: 1,073,741,824 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.234014] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 508, limit: 1,073,741,824 hold: 6,291,456 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.234187] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=4] [dc=0] [MEMORY] tenant: 509, limit: 1,073,741,824 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.234358] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 510, limit: 2,147,483,648 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
[2021-09-09 17:13:17.234532] INFO  [LIB] ob_malloc_allocator.cpp:505 [43566][0][Y0-0000000000000000-0-0] [lt=5] [dc=0] [MEMORY] tenant: 512, limit: 1,073,741,824 hold: 2,097,152 rpc_hold: 0 cache_hold: 0 cache_used: 0 cache_item_count: 0
```

租户 ctx 统计

租户下各个 ctx 的内存统计，ctx 是粗粒度的划分方式，一个租户会有多个 ctx。

```
[MEMORY] ctx_id=           DEFAULT_CTX_ID hold_bytes=    534,773,760
[MEMORY] ctx_id=          MEMSTORE_CTX_ID hold_bytes=    123,731,968
[MEMORY] ctx_id=         TRANS_CTX_MGR_ID hold_bytes=     12,582,912
[MEMORY] ctx_id=     REPLAY_STATUS_CTX_ID hold_bytes=     20,971,520
[MEMORY] ctx_id=        PLAN_CACHE_CTX_ID hold_bytes=     14,680,064
[MEMORY] ctx_id=                WORK_AREA hold_bytes=      2,097,152
```

租户 ctx 下 mod 统计

租户 ctx 下更精细的内存统计，一个 ctx 下会有多个 mod。

```
[MEMORY] tenant_id=    1 ctx_id=           DEFAULT_CTX_ID hold=    534,773,760 used=    431
,805,528
[MEMORY] hold=    142,591,488 used=    140,579,840 count=      30 avg_used=      4,685,994
mod=MysqlRequesReco
[MEMORY] hold=    125,798,400 used=    125,767,680 count=      60 avg_used=      2,096,128
mod=SqlExecutor
[MEMORY] hold=    104,857,600 used=    104,855,040 count=       1 avg_used=    104,855,040
mod=TransAudit
[MEMORY] hold=     23,867,392 used=     19,660,800 count=      14 avg_used=      1,404,342
mod=LOCALDEVICE
[MEMORY] hold=     12,579,840 used=     12,576,768 count=       6 avg_used=      2,096,128
mod=PartitLogServic
[MEMORY] hold=     10,483,200 used=     10,480,640 count=       5 avg_used=      2,096,128
mod=Election
[MEMORY] hold=      4,496,000 used=      4,483,464 count=      67 avg_used=         66,917
mod=ResultSet
[MEMORY] hold=      4,193,280 used=      4,192,256 count=       2 avg_used=      2,096,128
mod=OB_KVSTORE_CACHE
[MEMORY] hold=      2,154,368 used=        408,600 count=  24,952 avg_used=             16
mod=Number
[MEMORY] hold=      2,096,640 used=      2,096,128 count=       1 avg_used=      2,096,128
mod=ElectionGroup
[MEMORY] hold=      1,700,736 used=      1,687,552 count=     206 avg_used=          8,192
mod=MemtableCallbac
```

租户下 PM 统计

PM 内存都是 SQL 执行过程中临时内存，SQL 执行结束释放。

```
[MEMORY][PM] tid=     48768 used=              0 hold=      2,097,152 pm=0x7f4e3066eec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48770 used=      2,096,192 hold=      2,097,152 pm=0x7f4e3005aec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48771 used=      2,161,792 hold=      4,194,304 pm=0x7f4e2fd50ec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48772 used=    124,691,512 hold=    127,926,272 pm=0x7f4e2fa46ec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48773 used=              0 hold=      4,194,304 pm=0x7f4e2f73cec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48774 used=              0 hold=      2,097,152 pm=0x7f4e2f432ec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48775 used=              0 hold=      2,097,152 pm=0x7f4e2f128ec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48776 used=              0 hold=      2,097,152 pm=0x7f4e2ee1eec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48777 used=              0 hold=      2,097,152 pm=0x7f4e2eb14ec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48780 used=              0 hold=      2,097,152 pm=0x7f4e2dff6ec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=     48784 used=              0 hold=      2,097,152 pm=0x7f4e2d3ceec0 ctx
_name=DEFAULT_CTX_ID
[MEMORY][PM] tid=   summary used=    128,949,496 hold=    153,092,096
```

11.5.5. 内存问题诊断

内存泄漏动态诊断

`__all_virtual_memory_info` 显示了整个 OBServer 通过 ObMalloc 分配内存的情况，每一块内存都有一个标签叫做 label 或者 mod_id，通过查看 mod 的统计信息，可以分析系统内存的使用情况，并初步确认疑似存在内存泄露的模块。

为了便于排查内存泄露问题，OceanBase 数据库实现了内存泄漏动态诊断机制，其工作原理是在每一次跟踪模块进行内存分配的时候，记录其申请内存的地址及相应的调用栈；在所申请内存被释放的时候，将该条记录清空。这样正常申请、释放内存的调用栈会被很快清空，而出现内存泄露的调用栈的记录将一直存在，最后我们按照调用栈进行分组，将每个不同的调用栈当前存在的申请记录进行累加，显示在 `__all_mem_leak_checker_info` 中。换言之，累计次数多的调用栈很可能出现了内存泄露（当然也有可能是由于各种缓存而造成的，这个需要具体问题具体分析）。

打开 memory_leak

```
obclient> alter system set leak_mod_to_check='OB_COMMON_ARRAY';
```

像这样，就指定了跟踪模块为 OB_COMMON_ARRAY，设置完毕后就可以开始监控 `__all_mem_leak_checker_info` 表的信息，查看可能出问题的调用栈。

查询

```
obclient> select * from __all_virtual_mem_leak_checker_info order by alloc_count desc;
```

一般来说存在泄露的调用栈计数都很大（而且越来越大），将这样的调用栈利用 addr2line 打印出。

```
obclient> addr2line -pCbfe bin/oserver  0xe77df5 0xedf5a4 0xee14b0 0xee0923 1 0xeea558 0x4a
05c3 0x1485cd9 0x1485223 0x1483c4b 0x1526f2a 0x1401359 0x1403075 0x140325a 0x1406416 0x14a5
1bb 0x14a5130 0x140x48f3aa7db8 0x14a4cc8 0x14a50f4 0x14a5cb1 0x130166e 0xd08bc9 0xd069d2 0x
d01e97 0xeff033 0xefe6
$oceanbase_root/src/lib/utility/utility.cpp:58
$oceanbase_root/src/lib/../../src/lib/allocator/ob_mem_leak_checker.h:125
$oceanbase_root/src/lib/allocator/ob_tc_malloc.cpp:399
$oceanbase_root/src/lib/allocator/ob_tc_malloc.cpp:218
$oceanbase_root/src/observer/../../src/lib/allocator/ob_malloc.h:38
$oceanbase_root/src/lib/allocator/ob_malloc.cpp:121
$oceanbase_root/src/lib/../../src/lib/allocator/ob_malloc.h:116
$oceanbase_root/src/sql/../../src/lib/container/ob_array.h:295
$oceanbase_root/src/sql/../../src/lib/container/ob_array.h:291
$oceanbase_root/src/sql/../../src/lib/container/ob_array.h:468
$oceanbase_root/src/sql/optimizer/ob_log_table_scan.cpp:85
$oceanbase_root/src/sql/optimizer/ob_log_plan.cpp:1183
$oceanbase_root/src/sql/optimizer/ob_log_plan.cpp:1484
$oceanbase_root/src/sql/optimizer/ob_log_plan.cpp:1557
$oceanbase_root/src/sql/optimizer/ob_log_plan.cpp:2092
$oceanbase_root/src/sql/optimizer/ob_select_log_plan.cpp:467
$oceanbase_root/src/sql/optimizer/ob_select_log_plan.cpp:455
$oceanbase_root/src/sql/optimizer/ob_select_log_plan.cpp:890
$oceanbase_root/src/sql/optimizer/ob_select_log_plan.cpp:378
$oceanbase_root/src/sql/optimizer/ob_select_log_plan.cpp:450
$oceanbase_root/src/sql/optimizer/ob_select_log_plan.cpp:568
$oceanbase_root/src/sql/optimizer/ob_optimizer.cpp:23
$oceanbase_root/src/sql/ob_sql.cpp:1160
$oceanbase_root/src/sql/ob_sql.cpp:887
$oceanbase_root/src/sql/ob_sql.cpp:152
$oceanbase_root/src/observer/mysql/obmp_query.cpp:263
```

关闭 memory_leak

打开 checker 后执行性能会显著下降，问题定位后及时将 checker 关闭。

```
obclient> alter system set leak_mod_to_check='';
```

memory_context 内存动态泄漏检查

对于长生命周期的内存管理，memory_leak 可以提供很好的诊断能力，方便找到调用栈定位问题。但对于短生命周期模型，因为内存总是会被很快释放，因此 memory_leak 往往无能为力，这就需要有一个新的功能来解决这个问题。

memory_context 是 OceanBase 数据库基于生命周期管理内存的接口，memory_context 自身就代表了内存的生命周期，因此可以扩展 memory_context 的能力，用它来实现对短周期内存泄漏的诊断，需要注意的是，memory_context 能保证即使有泄漏，内存也可以被完全释放干净，因此准确的说，我们支持的是“短生命周期内存的动态泄漏”。

找到 static_id

当日志打出了“HAS UNFREE”日志，找到日志里的 static_id。

```
[2020-10-22 15:30:57.923806] ERROR has_unfree_callback (object_set.cpp:27) [67779][462][Y40
DC64589025-0005B23D7165EB9F] [lt=24] [dc=0] HAS UNFREE PTR!!!label: mytest1,static_id: 12,s
tatic_info: {filename_:"obmp_query.cpp", line_:475, function_:"process_single_stmt"}, dynam
ic_info: {tid_:67779, cid_:462, create_time_:1603351857840041}
```

打开 memory_leak

```
obclient> alter system set leak_mod_to_check='mytest1@12';
```

表示跟踪 static_id 为 12 的 memory_context 下的 mytest1 的内存申请，当然也支持通配符，但线上不建议这么做，毕竟有性能与空间代价，尽可能缩小范围才合理。

```
obclient> alter system set leak_mod_to_check='*@12';
```

这样写表示跟踪 static_id 为 12 的 memory_context 下的所有内存申请。

等待复现

下一次出现 HAS UNFREE 时，就会打出调用栈。

```
[2020-10-22 15:41:02.530881] INFO  object_set.cpp:523 [67784][472][Y40DC64589025-0005B23D71
35EBC2] [lt=15] [dc=0] CONTEXT MEMORY LEAK. ptr: 0x7f8f8a4145b0, size: 200, label: mytest1,
lbt: 0xb979d8b 0xb72d239 0xb71ad7a 0x3a88b97 0x7fd735c 0x7fcee89 0x7fcd41d 0xbac5fad 0x8044
c7a 0x80443b5 0x804001a 0x803f2ff 0x32e2cd7 0x32e2b7c 0x2d49d6d 0xb76f163 0xb76cd74 0xb76bd
ae
```

同样，可以用 addr2line 继续分析。

关闭 memory_leak

打开 checker 后执行性能会显著下降，问题定位后及时将 checker 关闭。

```
obclient> alter system set leak_mod_to_check='';
```

内存元数据 dump

可以把所有内存的元数据写到磁盘。

操作步骤

1. 打开 OceanBase 数据库的部署目录。
2. 新增 `etc/dump.config` 文件，写入目标指令（具体指令见下文）。
3. `kill -62 $pid`，查看 `log/memory_meta` 文件。

指令汇总

- 输出所有内存信息

  `dump chunk all`

- 输出指定租户 +ctx 的内存信息

`dump chunk tenant_id=$tenant_id,$ctx_id=ctx_id` （注意是纯数字）

查看 glibc 接口申请的内存

malloc 等 glibc 内存申请会被透传到 OceanBase 数据库的内存管理接口，mod_id 为 glibc_malloc，可以通过以下方式查看：

```
obclient> select *from __all_virtual_memory_info where mod_name = 'glibc_malloc' and hold >
0;
+-----------+----------------+----------+--------+--------------+----------+----------+----
----+--------------+------+-----------+-----------+-------+-------------+------------+
| tenant_id | svr_ip         | svr_port | ctx_id | label        | ctx_name | mod_type | mod
_id | mod_name     | zone | hold      | used      | count | alloc_count | free_count |
+-----------+----------------+----------+--------+--------------+----------+----------+----
----+--------------+------+-----------+-----------+-------+-------------+------------+
|       500 | 100.81.113.215 |    46824 |     26 | glibc_malloc | GLIBC    | user     |
0 | glibc_malloc | z1   | 249800208 | 219156754 | 41247 |           0 |          0 |
+-----------+----------------+----------+--------+--------------+----------+----------+----
----+--------------+------+-----------+-----------+-------+-------------+------------+
```

12.附录：OceanBase 数据库基础概念

B | C | D | E | F | G | H | J | K | L | M | N | O | P | Q | R | S | T | U | W | X | Y | Z

B

本地执行（Local Execution）

接收客户端请求生成执行计划的数据库服务器和计划实际执行的服务器是同一个。

表（Table）

最基本的数据库对象。每个表由若干行记录组成，每一行有相同的预先定义的列。用户通过 SQL 语句对表进行增、删、查、改等操作。通常，表的若干列会组成一个主键，主键在整个表的数据集合内唯一。

表组（Table Group）

对经常会被同时访问的一组表，为了优化性能，需要将它们相同类型的副本存储在同一个 OceanBase 数据库服务器中。通过定义一个 Table group，并且将这一组表放在这个 Table group 中来达到这个目的。此外，同一个 Table group 的多个分区表具有相同的分区数和分区规则。假设一个 Table group 里的表都有 N 个分区，所有这些表的第 i 个分区的集合组成一个 Partition group。同一个 Partition group 里的分区，主副本总是位于同一个 Server上。

并行合并（Parallel Compaction）

特指对单个分区的并行合并。

并行执行（Parallel Execution）

一个执行计划在执行前，根据需要访问的分区情况，被切分成一个或者多个任务来执行。在执行过程中，调度器可以串行调度计划的多个任务，也可以让多个任务同时被调度执行。这种多个任务被同时调度执行的方式称为并行执行。

布隆过滤器缓存（Bloom Filter Cache）

布隆过滤器缓存，用于快速判断行在基线数据或转储数据是否存在，当结果为不存在时，可以减少磁盘 IO 和 CPU 消耗。

备份元数据库/恢复元数据库（Backup Metadb/ Restore Metadb）

备份元数据库包含参数表 `backup_base_profile` 以及控制备份任务的四张表，分别是 `base_data_backup` 、 `base_data_backup_task` 、 `base_data_backup_task_history` 和 `inc_data_backup` 。恢复元数据库包含控制恢复任务的四张表，分别是 `oceanbase_restore` 、 `base_data_restore` 、 `inc_data_restore` 和 `oceanbase_restore_history` 。通常会将备份元数据库和恢复元数据库部署在同一个数据库中。

备份工具程序/恢复工具程序（AgentServer/agentrestore.jar）

AgentServer 是备份工具，是一个常驻进程，每隔一段时间查询元数据库 MetaDB 中的 base_data_backup 表有无备份任务，来控制整个基线、增量数据备份的发起、取消，也会随着任务的推进更新备份任务四张表的状态。agentrestore.jar 是恢复工具，顾名思义是 Java 编写的 Jar 包，也是常驻进程，每隔一段时间查询元数据库 MetaDB 中的控制表，负责调度整个恢复任务的发起，也会随着任务的推进更新恢复任务四张表的状态。

C

Commit Log (CLOG)

操作日志，用来记录对数据库对象的修改，通过 WAL 协议，满足事务原子性和持久性的要求。 一个分区的一个事务在执行过程中会产生一条或多条 Commit Log，并通过 Multi-Paxos 协议复制到这个分区的其它副本上。一个分区完整的 Commit Log 序列代表了这个分区完整的修改历史。

查询改写（Query Rewrite/Query Transformation）

通过对用户查询做等价的改写以便优化器生成最佳执行计划的过程。

存储过程（Stored Procedure）

服务器端提供的用户编程方式。

D

DFO (Data Flow Object)

分布式并行执行计划中，若干个需要在一起流水线执行的算子集合，有时也被称为子计划。

DIO（Direct Input-Output）

直接输入输出。

Data Transfer Layer (DTL)

数据传输层，分布式并行执行框架中用于提供各执行线程之间数据传输的网络传输框架。

大版本冻结（Major Freeze）

集群中所有的节点在一个统一的快照点冻结当前的活跃 Memtable，不再接受新开启事务的写操作，新事务的写操作在新的活跃 Memtable 中进行。

大版本冻结版本号（Major Freeze Version）

大版本冻结的版本号。

地域（Region）

Region 指一个地域或者城市（例如杭州、上海、深圳等），一个 Region 包含一个或者多个 Zone，不同 Region 通常距离较远。OceanBase 数据库支持一份数据的多个副本 Region 部署。

地域/互联数据中心（Region/IDC）

每台 OBServer 服务器都具有 Region/IDC 属性，其中 Region 记录 OceanBase 集群地域信息，通常代表一个城市，IDC 记录 OceanBase 集群的机房信息。一个 OceanBase 集群包含若干个 Region，每个 Region 包含若干个 IDC，每个 IDC 部署若干个 OBServer 服务器。根据不同 Region 和 IDC，OceanBase 客户端与 OBServer 服务器，或者 OBServer 服务器 OBServer 服务器之间的位置关系可以分为 3 种：同 Region 同 IDC、同 Region 不同 IDC 和不同 Region。三者优先级依次降低。

冻结Memtable（Frozen Memtable）

Active Memtable 达到一定的内存阈值，进行冻结生成冻结 Memtable，冻结的 Memtable 不能再写入增量数据。

冻结版本（Frozen Version）

发生冻结操作时的版本号。

读写 Zone/只读 Zone

传统的读写分离架构中，写库与读库是完全不同的数据库实体，它们天然是隔离的，通过数据同步工具实时同步。OceanBase 数据库通过只读副本和只读 Zone 的功能，实现了单个数据库实体下的读写分离支持。其中，读写 Zone 可以接受任何读写请求，只读 Zone 只能接受“登录认证/弱一致性读请求/select 不指定表名的请求 /use database/set session variables 语句”。

自适应游标共享（Adaptive Cursor Sharing）

一种可以让优化器每一个参数化 SQL 存储多个执行计划，并根据 SQL 语句中谓词的选择性来选择合适的计划的机制。

E

二级索引（Secondary Index）

访问数据表的一种辅助数据结构。与主键不同，二级索引通常包括由用户显式或隐式指定的一组键值。在 OceanBase 数据库中，二级索引一般实现为与主表关联的数据表。

F

访问路径（Access Path）

数据库中访问某张表的特定方式，通常分为主键访问和二级索引访问。

分布式执行（Distributed Execution）

执行计划在多台数据库服务器上执行，每台服务器完成其中的一部分工作。

分裂分区（Partition Split）

通过 Schema 变更调整表的分区数，将一个表由一个分区变成多个分区或者多分区的表格变成更多的分区，会触发表已有分区上数据按照新的分区方式重新组织。分裂操作可以在表（Table）上操作，也可以在表组（Tablegroup）上操作，在表组上操作等同于对表组中所有的表按照同样的方式进行分裂。

分区（Partition）

与 Oracle 中 Partition 的概念相同，在 OceanBase 数据库中只有水平分区，表的每一个分区包含一部分记录。根据行数据到分区的映射关系不同，分为 hash 分区、range 分区（按范围）和 list分区等。每一个分区，还可以用不同的维度再分为若干分区，叫做二级分区。例如，交易记录表，按照用户 ID 分为若干 hash 分区，每个一级 hash 分区再按照交易时间分为若干二级 range 分区。

分区表

OBServer 可以将普通表的数据按照一定规则划分到不同区块内，同一区块内的数据物理上存储在一起，这种划分区块的表叫做“分区表”。

分区裁剪（Partition Pruning）

数据库根据查询条件，避免访问无关分区的优化过程。分区裁剪可以分为静态和动态两种类型。

分区键

数据表的每一行中用于计算这一行属于哪个分区的列的集合，叫做分区键。当前分区表支持一级分区和二级分区两种。由分区键构成的用于计算该行属于哪个分区的表达式，叫做分区表达式。

如果是非分区表，那么一张表对应一个分区；如果是分区表，则一张表对应多个分区。

分区类型支持：KEY、HASH、LIST、RANGE。

分区键数据类型支持：数值、字符串、Date、Timestamp、二进制、ROWID。

副本（Replica）

为了数据安全和提供高可用的数据服务，每个分区数据在物理上存储多份，每一份叫做分区的一个副本。每个副本，包括存储在磁盘上的静态数据（SSTable）、存储在内存的增量数据（MemTable）、以及记录事务的日志三类主要的数据。根据存储数据种类的不同，副本有几种不同的类型，以支持不同业务在在数据安全、性能伸缩性、可用性、成本等之间的选择。

- 全能型副本：也就是目前支持的普通副本，拥有事务日志，MemTable 和 SSTable 等全部完整的数据和功能。它可以随时快速切换为 Leader 对外提供服务。

- 日志型副本：只包含日志的副本，没有 MemTable 和 SSTable。它参与日志投票并对外提供日志服务，可以参与其他副本的恢复，但自己不能变为主提供数据库服务。
- 只读型副本：包含完整的日志，MemTable 和 SSTable 等，但是它的日志比较特殊。它不作为 Paxos 成员参与日志的投票，而是作为一个观察者实时追赶 Paxos 成员的日志，并在本地回放。这种副本可以在业务对读取数据的一致性要求不高的时候提供只读服务。因其不加入 Paxos 成员组，又不会造成投票成员增加导致事务提交延时的增加。

OceanBase 数据库支持的副本类型如下表所示。

类型	LOG	MemTable	SSTable	数据安全	恢复为 leader 时间	资源成本	服务	名称(简写)
全能型	有，参与投票（SYNC_CLOG）	有（WITH_MEMSTORE）	有（WITH_SSSTORE）	高	快	高	Leader 提供读写，Follower 可非一致性读	FULL(F)
日志型	有，参与投票（SYNC_CLOG）	无（WITHOUT_MEMSTORE）	无（WITHOUT_SSSTORE）	低	不支持	低	不可读写	LOGONLY(L)
只读型	有，异步日志，但不属于 paxos 组，只是 listener（ASYNC_CLOG）	有（WITH_MEMSTORE）	有（WITH_SSSTORE）	中	不支持	高	可非一致性读	READONLY(R)

副本根据负载和特定的策略，由系统自动调度分散在多个 Server 上。副本支持迁移、复制、增删、类型转换等管理操作。

主/从副本（Leader/Follower）

主/从副本定义了某一时刻某表分区副本的角色。对于 OceanBase 数据库，每个分区至少三副本，副本之间使用 Paxos 协议同步 Leader 的 Redo 到 Follower，策略上三个成员里只要有超过半数以上成员接收到 Redo 并落盘成功确认后，Leader 上的事务才可提交。每个分区的全部副本（三副本，五副本等）是一个独立的 Paxos Group，交付自行独立选主的能力，当一个 OceanBase 节点故障时，只有对应 OceanBase 数据库节点内部作为 Leader 的 observer 服务中断，OceanBase 集群服务会自动选出新的 Leader。

负载均衡（Load Balance）

系统根据一定的策略，通过动态调整 UNIT 的位置和 UNIT 内副本的位置，使得同一个 Zone 内所有 Server 的资源使用率达到均衡的过程。负载均衡策略要考虑很多因素，目前分为两级调度。具体信息请参见本节 **Partition 调度** 和 **Unit 调度** 的信息。

复制表功能（Copy Table-Oceanbase 2.X）

为应对应用访问频率高更新低频率同时总能访问到最新数据的小表访问需求，同时要保证数据一致性，目前只能选择强一致性读访问 Leader 数据的方案。但由于访问频率高，Leader 容易成为性能瓶颈。为了解决“小表广播”需求场景问题，Oceanbase 数据库 V2.x 版本结合自身架构提供了复制表功能，将相关小表的副本复制到表所属租户的所有 OBServer 上。该表称为复制表，这些副本称为复制副本。复制表的更新事务在提交时保证将数据同步到所有的全功能副本及复制副本，确保在更新事务 Commit 成功之后，在租户任意 OBServer 上能读到该事务修改的数据。

G

Granule

分布式并行执行计划中，对于表和索引扫描的最小工作任务粒度，可以是一个分区，或者是一个 query range。

工作线程（Worker Threads）

OceanBase 数据库中用以处理租户请求的线程。属于同一个租户的一组工作线程通过共享一个任务队列来服务用户的请求。

H

Hint

数据库中用以直接指定优化器行为的用户原语。

行

一个行由若干列组成，有些时候其中的部分列构成主键（row key）并且整个表按主键顺序存储。有些表（例如，SELECT 指令的结果）可以不包含主键。

行版本合并（Row Compact）

将增量行数据中多个修改版本的数据合并成一行数据。

行缓存（Row Cache）

基线数据和转储数据的行数据缓存，用于提升查询性能。

合并/每日合并（Major Compaction）

将内存 MemTable 中的增量数据以及转储的增量数据和持久化存储上的基线数据进行合并，形成新的基线数据的过程。

合并分区（Merge Partition ）

聚合是分裂的逆操作。用户通过 Schema 变更调整表格分区数触发，调整后的分区数比调整前的少。系统会把多个分区按照新的分区方式合并在一起。聚合操作可以在表格（Table）上操作，也可以在表组（Table group）上操作，在表组上操作是对表组中所有的表格按照同样的方式进行聚合。

宏块（Macro Block）

OceanBase 数据库数据文件的内部管理单位，包含若干微块，是 OceanBase 数据库存储系统写入的最小单元。

宏块分裂（Macro Block Split）

由于宏块范围内插入和更新数据导致空间不足，需要将数据存放到多个宏块中。

宏块合并（Macro Block Merge）

由于删除数据，相邻的几个宏块中的所有行可以在一个宏块中存放，把相邻的多个宏块转换成一个宏块的过程。

宏块空间回收（Macro Block Recycle）

合并过程中会根据原基线数据和变更数据生成新的基线数据，原基线数据的宏块被重复利用的过程称为回收。

宏块预读（Macro Block Prefetch）

在范围查询的过程中，根据需要提前预读取相邻的宏块。

活跃 Memtable（Active Memtable）

当前活跃的 Memtable，可以写入增量数据。与 Frozen Memtable 相对应。

J

基线数据（Baseline Data）

每日合并生成的存储于持久化介质上的只读有序数据。

基线数据版本（Baseline Data Version）

基线数据的版本。

渐进合并（Progressive Compaction）

为了降低全量合并对系统的影响，一轮合并操作只合并部分的宏块，经过几轮以后，完成所有宏块的合并。采用渐进合并的主要目的是控制单次合并的时间，在表模式发生变化（如增加、删除列）的时候，需要在合并过程中更新所有行，如果是一个大表，更新所有行对合并时间有很大的影响。在这种情况下，采用渐进合并能有效控制每次合并的时间。

K

可用区/区（Availability Zone/ Zone）

Zone 是 Availability Zone 的简称。一个 OceanBase 集群，由若干个可用区（Zone）组成。通常由一个机房内的若干服务器组成一个 Zone。为了数据安全性和高可用性，一般会把数据的多个副本分布在不同的 Zone 上，可以实现单个 Zone 故障不影响数据库服务。

库，数据库（Database）

数据库对象，一个数据库对象中可以包含表、视图等其他数据库对象。

快速参数化（Faster Parsing）

OceanBase 数据库特有的针对实参 SQL 快速扣取参数（参数化）的过程。快速参数化结合OceanBase 数据库计划缓存的固有特点，通过增加约束条件等方法，避免了输入 SQL 再次执行时的语义分析过程，加速了计划匹配的效率。

L

Location Cache
表分区的副本信息。
连接顺序（Join Order）

执行计划中连接多表时的执行顺序。

轮转合并

为了减少合并对业务的影响，各个 Zone 按照指定的顺序依次合并，正在合并的 Zone 不对外提供服务。

逻辑数据中心（Logical Data Center（LDC））

LDC 是对 IDC（Internal Data Center, 互联网数据中心）的一种逻辑划分。OceanBase 数据库在多地多中心部署时，会以 Region 和 Zone 的单位对所有 OBServer 服务器进行划分，此时 OceanBase 数据库的客户端路由和 OceanBase 数据库内部的 RPC 路由被称为 LDC 路由。

M

MVCC（Multi-Version Concurrency Control）

多版本并发控制。

慢查询

超过指定时间的 SQL 语句查询。

Multi-Paxos

一种执行多 Paxos 实例的优化协议，OceanBase 数据库使用 Multi-Paxos 协议实现 Commit Log 的多机持久化。

Membership Log

一种特殊的记录一个分区成员组变更记录的 Commit Log。

MySQL 模式（MySQL Mode）

MySQL 兼容性模式。为降低 MySQL 数据库迁移 OceanBase 数据库引发的业务系统改造成本，使业务数据库设计人员、开发人员、数据库管理员等可复用积累的技术知识经验，快速上手使用 OceanBase 数据库所做的一种租户类型功能，目前高度兼容 MySQL 语法，常用系统表、函数保持一致。

N

Nop Log

Commit Log 中的一种特定类型的日志，表示空操作。Nop Log 产生于 Multi-Paxos 协议的恢复阶段，如果一条日志没有在多数派副本上持久化成功，在恢复阶段就可能产生一条 Nop Log 作为这条日志的内容。

O

OceanBase（简称 OB）

由蚂蚁集团、阿里巴巴完全自主研发的金融级分布式关系数据库。

OBProxy

OceanBase 数据库高性能反向代理服务器，它接收客户端的应用请求，并转发给 OBServer，然后 OBServer 将数据返回给 OBProxy，OBProxy 将数据转发给应用客户端。具有防连接闪断、屏蔽后端异常（宕机、升级、网络抖动）、MySQL 协议兼容、强校验、支持热升级和多集群等功能。

OBServer

OceanBase 数据库服务器。Server 是运行 OBServer 进程的物理机，一台物理机上可以部署一个或者多个 OBServer。在 OceanBase 数据库内部，Server 由其 IP 地址和服务端口唯一标识。

OceanBase 云平台（OceanBase Cloud Platform）

简称 OCP，提供可视化监控、运维、报警等功能。

OLAP（On-Line Analytical Processing）

联机分析处理。

OLTP（On-Line Transaction Processing）

联机事务处理。

Oracle 模式（Oracle mode-Oceanbase 2.X）

Oracle 兼容性模式。为降低 Oracle 数据库迁移 Oceanbase 数据库引发的业务系统改造成本，使业务数据库设计人员、开发人员、数据库管理员等可复用积累的技术知识经验，快速上手使用 OceanBase 数据库所做的一种租户类型功能，逐步兼容 Oracle 语法，常用系统表、函数保持一致。

P

Partition 调度（Partition 负载均衡）

对于每一个租户每个 Zone 中的若干个 UNIT，通过在 UNIT 之间迁移副本，达到每个 UNIT 内资源使用率的均衡。

其中：

- 属于同一个分区表的若干不同分区，会均匀分散在不同的 UNIT 上，使得每个 UNIT 有相同个数的分区
- 属于同一个 Partition Group 的若干分区，会聚集在同一个 UNIT 上
- 在保证上述每组分区个数均衡的基础上，通过交换这组分区内的分区，降低 Server 的磁盘水位线
- 通过迁移非分区表的分区，降低 Server 的磁盘水位线

PX Worker

分布式并行执行下，在每一个机器上参与执行一个分布式并行计划的工作线程。

Q

Query Coordinator（QC）

在分布式并行执行计划中，主控节点上的一个线程，用于调度、协调整体分布式并行执行计划的执行。

迁入

Move in 是用户通过 Schema 变更把一个没有表组（table group）属性的表格设置上表组属性。数据库内会检查被操作表格的分区方式和目标表组的分区方式，只有完全一致的情况，DDL才会成功。

迁出（Move out）

迁出是迁入的逆操作，用户通过 Schema 变更把一个表格的表组（table group）属性删除，即把这张表从表组内迁出。用户通过 Schema 变更把一个表格的表组属性从 A 修改为 B，表格会从 A 中迁出，然后迁入 B。

迁移（Migration）

将分区的副本从一个节点迁到另一个节点，先在目标节点上增加一个副本，完成后再将源节点上的副本删除。

全量合并（Full Compaction）

进行全量合并时，无论分区的宏块是否修改都重新生成一遍。可以在表粒度指定是否采用全量合并，通常在表模式发生变化（如增加、删除列）或者存储属性发生变更（如修改压缩级别）时，对该表进行全量合并。在需要对表进行全量合并时，通常采用渐进合并的方式来控制单次合并的时间。

全局索引（Global Index）

OceanBase 数据库中跨分区的数据索引。全局索引通过全局化的存储数据键值与主键信息，可以快速定位到数据所在的分区。

全局一致性快照（Global Consistent Snapshot-Oceanbase 2.X）

在没有实现全局一致性快照前，分布式数据库无法实现跨节点的一致性读和保证因果序。OceanBase 数据库 V1.4.x 版本中，应用系统设计和开发人员需要保证一条 SQL 语句中访问的多个表、多个分区都在同一个 OceanBase 数据库节点上，同时对于依赖操作顺序的业务系统，无法保证两个前后事务分别修改两个节点上两张表的数据反映顺序。OceanBase 数据库 V2.0 版本实现的全局一致性快照功能，从根本上解决了这些问题。相对于 Google Truetime 基于原子钟的硬件实现，OceanBase 数据库的全局时间戳（GTS）服务是纯软件实现的，不依赖特定的硬件设备，也不对客户方的部署环境提额外的要求，使得 OceanBase 数据库能够服务更广泛的专有云客户。OceanBase 数据库 V2.x 版本的全局时间戳打开后，跨节点读写、因果序的行为和单机数据库完全一致。

强一致性读/弱一致性读（Strong Read Consistentcy/ Weak Read Consistentcy）

强一致性读是默认的一种 OceanBase 数据库的 SQL 执行方式，即 SQL 必须转发到涉及 Partition 的 Leader 所在的 OBServer 服务器上才能执行，用以保证获取到实时最新数据。弱一致性读是相对于强一致性读来说的。即 SQL 只需在涉及 Partition 的 OBServer 服务器上执行即可，不强制是 Leader。如需使用弱一致性 `SELECT`，主要通过两种方式。一种是带 read_consistency(weak) Hint 的 `SELECT` 语句，另一种是在当前 Session 中修改 `ob_read_consistency` 的系统变量取值为 weak。

R

Redo 日志索引

Ilog 操作日志（Commit Log）在文件中不是根据日志 ID 有序存放的，为顺序读取操作日志，在ilog 中存放了每条操作日志在日志文件中的位置，ilog 中的记录是按照日志 ID 有序的。

Rowid

分区局部索引的基线数据行中保存的主表行数据的位置信息，用于快速定位相应的主表行。

Row Merge

将基线行数据和增量行数据进行融合形成新版本行的过程。

Row Purge/Range Purge

标识一些行被删除了。

RPC（Remote Process Call）

远程方法调用。

RS（RootServer）

主控服务器。主要进行集群管理、数据分布和副本管理。

RS List

OceanBase 集群中启动了 Root Service 服务的机器的 IP 列表（一般是每个 Zone一个）。

RS List 涉及集群的创建，与 `sys` 租户关闭密切。

RS List 有两种集合形态，一种是直接从 Config Server 获取的 RS List；另一种是用 Server List 更新后的 RS List。

Reconfirm

Reconfirm 即再次确认，Multi-Paxos 协议中 Leader 上任需要执行的一个流程。分区 Leader 上任之后，要把之前这个分区多数派上持久化成功的日志再次确认一遍，并把这些确认过的日志同步到所有正常工作的副本，确认多数派都持久化成功所有日志后 Reconfirm 才算成功。

S

Schema

通常来说，指的是数据库对象（如表、视图、索引等）的模式；在 OceanBase 数据库服务端，也指所有数据库对象模式的集合。

Schema Version

在 OceanBase 数据库服务端，维护了一个全局的 Schema 版本号，数据库对象的每次变化都会引起全局 Schema 版本的升高。每一个数据库对象也有一个自己的版本号，该版本号是该对象最近一次修改（创建、变更等）对应的全局 Schema 版本号。

Slog/SSTable Log

节点为了维护本机基线数据一致性而使用的日志。

Server List

OceanBase 集群所有机器的 IP 列表。

Sub Query Coordinator（SQC）

分布式并行执行计划中，每一台参与执行的服务器上会有一个本地的协调者，用于接收从 QC 发送来的调度指令，获取本地执行工作线程，生成本地执行任务的 granule，协调本地执行。

SSTable

用于存储基线数据或转储数据，行数据有序存储。

Start Working Log

分区 Leader 上任成功，处理新事务之前写的第一条特殊日志，这条日志包含新 Leader 的上任时间。

数据库（Database）

数据库是按数据结构来组织、存储和管理数据的仓库。数据库下包括若干表、索引，以及数据库对象的元数据信息。

数据倾斜（Data Skew）

数据中对于某一个或某几个值出现的次数特别多，占据了对应数据较大的比例，在分布式执行过程中，数据倾斜会引起执行的长尾，导致被分配处理这些值的执行线程会用更多的时间完成执行。

刷新 Schema（Schema Refresh）

OceanBase 数据库的 DDL 操作对系统对象的变更都发生在 Root Server 上，为使得集群中的每一个节点都能获取到最新的 Schema 信息，Root Server 定期将最新的 schema version 广播给集群中每一个节点。节点在收到 schema version 后，和本地缓存的 schema version 比对，如果本地落后于全局版本，则从系统表中获取变更来更新本地缓存的系统对象信息，这个过程叫刷新 Schema。

算子（Operator）

组成执行计划的基本单元。一般通过多个算子构成一棵执行树完成用户 SQL 请求。

T

统计信息（Statistics）

一个描述数据库中表和列信息的数据集合。在 OceanBase 数据库中，统计信息有表统计信息（table level statistics）和列统计信息（column level statistics）两种。

U

UNIT 调度（Unit 负载均衡）

对于每个 Zone，根据 UNIT 的动态调度，达到均衡的策略：

- 属于同一个租户的若干个 UNIT，会均匀分散在不同的 Server 上
- 属于同一个租户组的若干个 UNIT，会尽量均匀分散在不同的 Server 上
- 当一个 Zone 内机器整体磁盘使用率超过一定阈值时，通过交换或迁移 UNIT 降低磁盘水位线
- 否则，根据 UNIT 的 CPU 和内存规格，通过交换或迁移 UNIT 降低 CPU 和内存的平均水位线

Universe

表示跨地域部署的所有 OceanBase 数据库。

W

微块（Micro Block）

宏块内部的管理单位，数据读的最小单位。

微块缓存（Block Cache）

微块在内存中的缓存，用于减少微块被频繁访问的 IO，提升查询性能。

微块索引缓存（Block Index Cache）

微块索引在内存中的缓存，用于提升频繁访问的查询性能。

位置（Locality）

用来描述一个表的副本类型以及分布位置的方式。基本语法结构型为 `replicas@location`，由如下一些元素组成：

- 副本类型（replicas）：例如， F 表示全功能副本；L 表示日志型副本。
- 位置（location）：位置是系统已知的一组枚举值。位置是zone的名称，如hz1, bj2等。
- 量词：不指定量词时，表示一个副本；用 `{n}` 表示 n 个副本。有一种特殊的量词 `{all_server}` 表示副本数和可用的 Server 数相同。目前，由于一些实现上的考虑，一个分区在一个 Zone 中最多有一个全功能或日志型副本（这些类型的副本是 Paxos 复制组的成员），只读型副本在同一个 Zone 可以有多个。

位置信息缓存（Location Cache）

分区位置信息缓存。每个节点维护分区位置信息，在执行计划生成阶段，根据位置信息生成不同类型的执行计划；在执行阶段，根据位置信息将计划发送到相应的节点执行。位置信息缓存的更新是按需的，如果在语句执行过程中发生由于位置失效产生的错误，则刷新相应分区的位置信息。

无主选举

分区没有 Leader 的情况下，这个分区的多副本进行的选主流程。无主选举的两个触发场景是集群重启分区第一次选举和分区原 Leader 故障，无主选举需要等原 Leader 的 Lease 过期后才能开始。

X

小版本冻结（Minor Freeze）

分区在内存中的增量数据超过阈值时，冻结该分区当前活跃 Memtable，不再接受新开启事务的写操作，新事务的写操作在该分区新的活跃 Memtable 中进行。

Y

远程执行（Remote execution）

接收用户请求和生成执行计划的数据库服务器和计划执行的数据库服务器不是同一个，并且只有一台数据库服务器执行该计划。

优化器（Optimizer）

优化器是决定用户查询执行计划的核心模块。通过访问相关数据的统计信息，结合 OceanBase 数据库内置的规则与代价模型，优化器为用户查询生成最佳的执行计划。

有主改选

分区有 Leader 的情况下，把分区 Leader 切换到指定 OBServer 的流程。有主改选不需要等原有 Leader Lease 过期。

Z

Zone

是 Availability Zone 的缩写。

一个 OceanBase 集群由若干个 Zone 组成。Zone 的含义是可用性区，通常指一个机房（数据中心或 IDC）。为了数据的安全和高可用性，一般会把数据的多个副本分布在多个 Zone 上。这样，对于 OceanBase 数据库来说，可以实现单个 Zone 的故障不影响数据库服务。一个 Zone 包括若干物理服务器。

增量合并（Incremental Compaction）

进行增量合并时，只重新生成所包含的行发生修改的宏块。

增量数据（Incremental Data）

执行增、删、改（insert、update、delete）等操作产生的修改数据，该部分数据尚未与基线数据合并，包括 Memtable 数据和转储数据。

增量数据表（Memtable）

内存中增量修改记录的集合。

租户（Tenant）

OceanBase 数据库通过租户实现资源隔离，采用单集群多租户的管理模式。OceanBase 集群的一个租户相当于一个 MySQL 或者 Oracle 的实例。OceanBase 数据库的租户之间的资源和数据都是隔离的。租户拥有一组计算和存储资源，提供一套完整独立的数据库服务。

OceanBase 数据库上有系统租户和普通租户。系统租户下存放 OceanBase 数据库管理的各种内部元数据信息；普通租户下存放用户的各种数据和数据库元信息。

执行计划/计划（Execution Plan/Plan）

数据库中用以执行用户 SQL 请求的物理代码的集合，通常是一棵由算子构成的执行树。

执行计划缓存/计划缓存（Plan Cache）

执行计划在每台 Server 上的缓存。SQL 语句的优化是一个比较耗时的过程，为了避免反复执行优化过程，生成的执行计划会加入到执行计划缓存中，以便再次执行该 SQL 时使用。每一个租户在每一台 Server 上都有一个独立的执行计划缓存，用以缓存在此 Server 上处理过的执行计划。

执行计划绑定（Execution Plan Binding）

用户通过 outline 绕过优化器而直接指定 SQL 的执行计划的过程，通常用于优化器生成的执行计划错误或者不够高效的场景。

执行计划匹配（Execution plan matching）

数据库为用户 SQL 选择在执行计划缓存中合适的执行计划的过程。

主键（Rowkey）

与传统关系数据库的主键（primary key）相同，表中每一行数据的标识符，用于唯一标识表中的行，表中的数据按照 RowKey 排序。

主可用区（Primary Zone）

Primary Zone 指的是分区的主副本所在的 Zone，可以为分区指定一个 Zone 的列表，当分区需要切主的时候，容灾策略会按照这个列表的顺序决定新主的偏好位置。

如果不设定 Primary Zone，系统会根据负载均衡的策略，在多个全功能副本里自动选择一个作为 Leader。

只读可用区（Read Zone）

只读 Zone 是一种特殊的 Zone，在这个 Zone 里，只部署只读副本。通常当多数派副本故障的时候，OceanBase 数据库会停止服务，但在这种情况下，只读 Zone 能继续提供弱一致性读，即读 Zone、读库。这也是 OceanBase 数据库提供读写分离的一种方案。

转储（Dump）

将小版本冻结的 Frozen Memtable 中的增量数据与前一个版本转储数据（如果存在）合并后持久化存储到磁盘的过程称为转储。在合并过程中，转储数据也参与和基线数据的合并，形成新的基线数据。

转储数据版本号（Minor Freeze Version）

转储数据的版本号。

子计划（Sub Plan）

在分布式并行执行场景下，和 DFO 意思相同。

资源池（Resource Pool）

一个租户拥有若干个资源池，这些资源池的集合描述了这个租户所能使用的所有资源。一个资源池由具有相同资源规格（Unit Config）的若干个 UNIT（资源单元）组成。一个资源池只能属于一个租户。每个 UNIT 描述了位于一个 Server 上的一组计算和存储资源，可以视为一个轻量级虚拟机，包括若干 CPU 资源、内存资源和磁盘资源等。

一个租户在同一个 Server 上最多有一个 UNIT。实际上，从概念上讲，副本是存储在 UNIT 之中，UNIT 是副本的容器。

组提交（Group Commit）

组提交是为了优化写日志时的刷磁盘问题，将多个事务的日志聚在一起用一次 IO 完成持久化。