
视频讲解

第5章

OceanBase数据库连接与OBProxy管理

5.1 OBProxy是什么?

OceanBase是一个分布式数据库,与传统数据库最大的不同是数据库是分布式,数据副本被打散分布在不同的Server上。因此,客户端在访问OceanBase时,也与传统数据库不同。

在分布式数据库系统中,客户端需要与多个数据库节点进行通信。这就带来了一些挑战,包括负载均衡、连接池管理、安全性和性能优化等。那么,如何做到像访问传统数据库那样访问分布式数据库呢?

一种连接应用于OceanBase数据库的桥梁孕育而生,它就是OBProxy。它主要有以下两方面的作用:

(1) 连接管理。OBServer集群规模庞大,服务器、软件出现问题或者本身运维服务器上线、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可以保证连接的稳定性,自身对OBServer的复杂状态进行处理。

(2) 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服务器,执行效率更高。

5.2 OBProxy管理

5.2.1 OBProxy简介

OBProxy产品从2014年开始设计研发,至今已经有近10年的历史了,其产品在蚂蚁内部、专有云场景、公有云场景都有广泛使用,在访问链路上发挥了重要的作用。

OBProxy全称为OceanBase DataBase Proxy(ODP),是OceanBase数据库专用的服务代理。使用OBProxy可以屏蔽后端OBServer集群本身的分布式带来的复杂性,让访问分布式数据库像访问单机数据库一样简单。用户的SQL会先发送给ODP节点,由ODP选择合适的OBServer(OceanBase数据库进程名)进行转发,并将结果返回给用户。

OceanBase数据库作为典型的分布式数据库与传统单机数据库不同,每个表甚至每个表的不同分区都可能存放在不同的服务器上。想要对表进行读写,必须先要定位到数据所属的表或是分区的主副本位置,然后才能执行相应的SQL DML语句,这在应用层面而言是几乎不

可能做到的。OBProxy 作为 OceanBase 数据库专用的反向代理软件，其核心功能就是路由以将客户端发起的数据访问请求转发到正确的 OBServer 上。

首先，从整体架构对 OBProxy 进行一个了解。OBProxy 架构如图 5-2-1 所示。

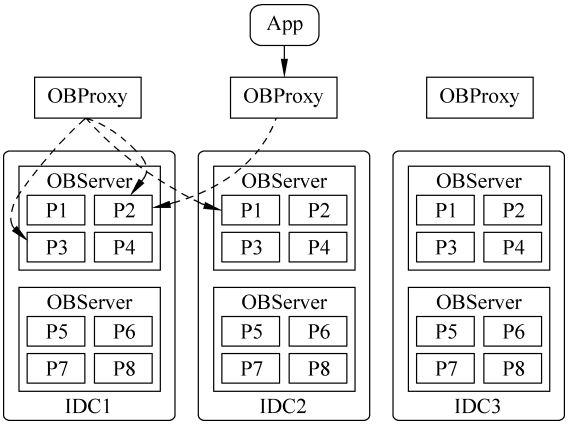

图 5-2-1　OBProxy 架构

图 5-2-1 中 App 表示业务程序，App 前面有三台 OBProxy(进程名叫作 OBProxy)。在实际部署中，App 和 OBProxy 之间一般会有负载均衡器，如 F5 将请求分散到多台 OBProxy 上面，OBProxy 的后面就是 OBServer，图 5-2-1 中一共有 6 台 OBServer。OBProxy 知道 OBServer 中的数据分布信息，可以将用户 SQL 高效转发到数据所在服务器，这样执行效率比转发到没数据的节点执行效率更高。如表 t1 数据在图 5-2-1 中 P1 内，表 t2 数据在 P2 内，表 t3 数据在 P3 内；红色表示主副本，蓝色表示备副本。对于 insert into t1 语句，OBProxy 可以将 SQL 转发到 IDC2 中含有 P1 主副本的 OBServer 服务器上。

为什么 OBProxy 需要将 SQL 发送到数据所在节点呢？因为发到数据所在节点后，SQL 的执行计划可以在本地执行，没有了远程 RPC 调用，性能会更好。在实际生产环境中，除了考虑数据分布，OBProxy 还会考虑服务器的地理位置分布，避免请求跨机房、跨城等情况出现。

1. 如何使用 OBProxy？

在部署好 OceanBase 数据库集群(包含部署 OBProxy)后，用户就可以使用数据库服务了。以 JDBC 访问数据库举例：

```
final String URL = "jdbc:mysql://127.0.0.1:2883/test?useSSL=false&useServerPrepStmts=true";
```

在建立连接时，需要先初始化相关的连接信息。上面 URL 包含了数据库的 IP、PORT、访问的库名 test、连接属性等信息。使用 OBProxy 访问和直接连接 OBServer 访问的区别在于 IP 和 PORT 的不同，其他信息不需要任何改动。后续使用时，OBProxy 对用户完全透明。

因此，使用 OBProxy 会让问题变得简单，用户无须关心数据库系统的分布式架构，这样设计的好处有以下三点。

(1) OBProxy 兼容 MySQL 协议，用户可以使用 MySQL 标准驱动。

(2) 从 MySQL 数据库切换到 OceanBase 数据库，用户访问数据库的代码无须改动。

(3) OBProxy 对用户屏蔽了后端分布式系统的复杂性：如服务器的变更、服务器宕机、租户的 Unit 分布、每日合并等，保证客户端和 OBProxy 连接的稳定性。

2. OBProxy 的执行流程

对功能模块有进一步认识之后，再从 SQL 请求去看 OBProxy 的执行流程。OBProxy 的执行流程如图 5-2-2 所示。

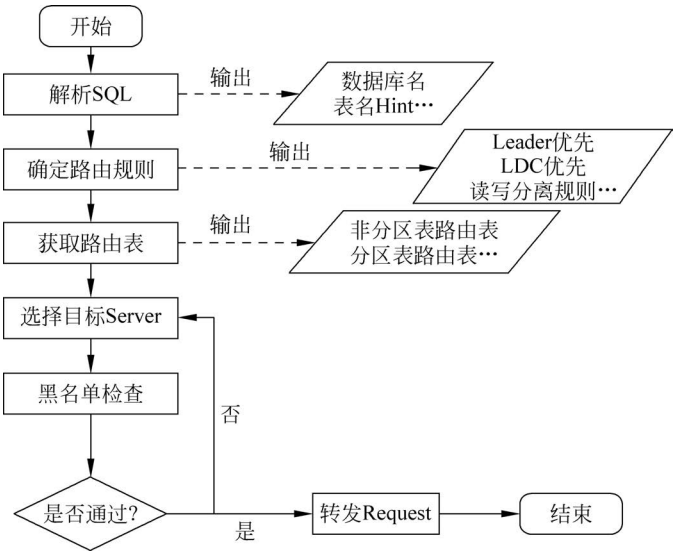

图 5-2-2　OBProxy 的执行流程

执行流程如下：

（1）客户端和 OBProxy 建立 TCP 连接，OBProxy 通过 epoll（网络通信库中实现）处理套接字的读写事件。

（2）从 TCP 读取字节流保存到 buffer 缓冲中，并进行 MySQL 协议报文解析，先解析报头再决定是否解析后续内容。

（3）从报文中读取 SQL，进行 SQL 解析。

（4）根据表名和 Location Cache（表分区信息）找到表数据分布，结合路由规则等计算出 SQL 语句发往的目的 OBServer 主机。

（5）从数据库连接中找到对应的 OBServer 连接，并做 OBServer 的容灾管理检查。

（6）使用选中的连接，通过高性能转发框架和后端 OBServer 进行交互。

（7）将从 OBServer 收到的数据进行协议层处理，并返回给客户端。

3. OBProxy 的关键特性

了解了 OBProxy 执行一条 SQL 时的主要工作后，下面总结一下 OBProxy 的主要关键特性。

（1）高性能转发。OBProxy 是数据访问流程中的重要部分，采用多线程异步框架和透明流式转发的设计，对关键路径代码做深入优化，同时确保了自身对服务器资源的最小消耗。

（2）协议支持。支持多种格式协议，如 MySQL 协议、Oracle 兼容协议和自主研发协议。目前，在增强自主研发协议实现更多强大功能。

（3）连接管理。保持客户端连接稳定是非常重要的事情，直观感受是业务不报连接错误，OBProxy 会去屏蔽后端的问题，保持和客户端连接的稳定。

（4）数据路由。数据路由影响性能和高可用，与部署架构、数据分布等关系密切，对 SQL

执行有很大影响，路由正确也是大家非常关心的特性。

5.2.2 OBProxy 连接管理

OBProxy 为用户提供了接入和路由功能，用户连接 OBProxy 就可以正常使用 OceanBase 数据库。用户在使用数据库功能时，OBProxy 和 OBServer 进行交互，且交互流程对用户透明，连接管理就是该交互过程中的关键点。

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。

(1) 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 ClientSession 内部，以便 OBProxy 与其他 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 进行顺利认证。

(2) 存储连接。

每当客户端向 OBProxy 发送请求时，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 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。

(3) 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 就不能处理了。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。

(4) 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 中记录下修改时间。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。

(5) 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，客户端能够正常地读写数据。需要处理 Server Session 异常的情况有：

① 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。

② 当 OBServer 发生宕机时，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。

③ 当 OBProxy 与 OBServer 通信超时，MySQL 协议本身没有超时机制，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。

④ 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，结束未完成的事务，避免 Session 长时间被挂住。

⑤ 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。

⑥ 在OBProxy宕机情况下，OBProxy上的连接都会断掉，可能很快OBProxy被重新启动，或者该OBProxy负责的连接被其他的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。

（6）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，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，这是一种优化的手段，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。

（7）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。

1. OBProxy的连接管理

登录成功后，客户端↔OBProxy↔OBServer之间的网络连接便建立起来，此时OBProxy只是和其中一台OBServer建立了连接。随着SQL请求的到来，如果路由到新的OBServer，会和新的OBServer建立连接。在此过程中涉及连接的映射关系、状态同步和连接功能特性。

1）连接的映射关系

连接映射主要指客户端连接和服务端连接之间的关系，我们先从一个客户端连接说起。当客户端和OBProxy建立一个连接后，OBProxy会和后面N个server建立连接，整个连接的映射关系如图5-2-3所示。

可以看到，OBProxy按需和两台OBServer建立了连接。这两个连接只属于这一个客户端连接，不会被其他客户端连接复用。连接映射的关键点就是需要用id标志出每一个连接并记录id之间的映射关系，可以将图5-2-3抽象成如下模型：

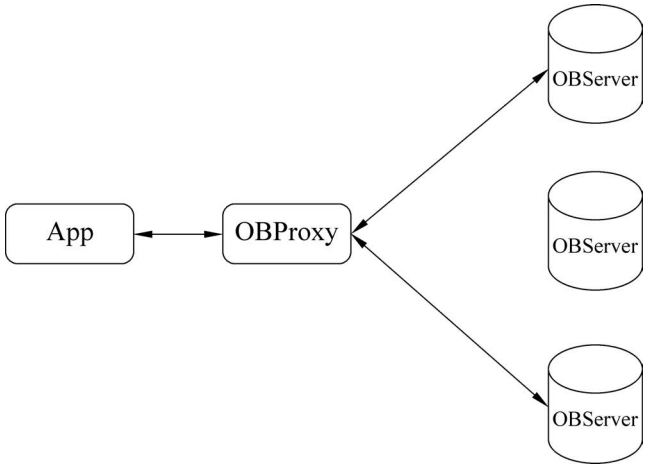

图 5-2-3　连接的映射关系

App←[proxy_sessid1]→OBProxy←[server_sessid1]→OBServer1←[server_sessid3]→OBServer3

这样就可以用 proxy_sessid 唯一标志 App 和 OBProxy 之间的连接，用 server_sessid 唯一标志 OBProxy 和 OBServer 之间的连接。当 SQL 执行错误、执行慢等情况出现时，会将映射关系打印到日志中，这样就将 App 和 OBServer 关联起来，进行全链路问题定位。

2）状态同步

一个客户端连接对应多个服务端连接，要保证执行结果的正确性，就要求多个服务端连接的 Session 状态是一致的。那么，状态不同步会导致什么问题？我们举个反例，假设用户执行下面的 SQL 命令：

```
set autocommit=1;
insert into t1 values(1);
insert into t2 values(2);
```

执行过程如下：

（1）set autocommit=0 发给 OBServer1。

（2）insert into t1 values(1)发给 OBServer1。

（3）进行连接切换，insert into t2 values(2)发给 OBServer2。

状态同步执行过程如图 5-2-4 所示。

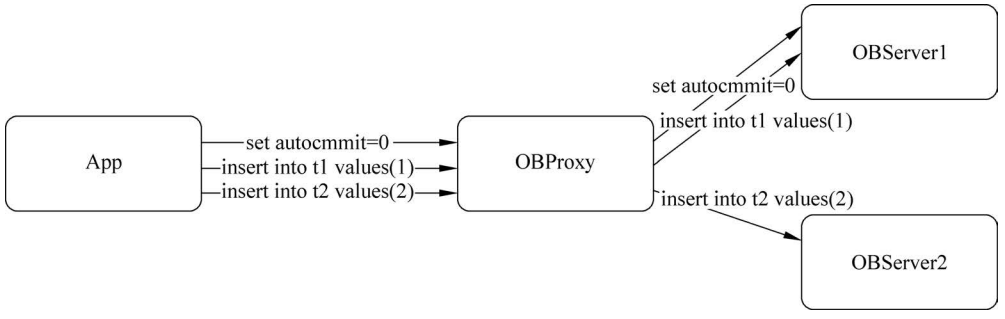

图 5-2-4　状态同步执行过程

对于第三条 SQL，OBProxy 和 OBServer2 的连接并未同步连接状态 autocommit=1，这

样就可能导致第三条语句 insert into t2 并未提交事务。

正确的步骤是 OBProxy 在给 OBServer2 发送 INSERT SQL 前，先同步 autocommit 变量的值。OBProxy 通过版本号机制解决了状态同步的问题，实现了 database、sessionvariables、last_insert_id、ps prepare 语句的状态同步，保证功能的正确性。

3）连接功能特性

与单机数据库不同，OBProxy 改变了连接的映射关系为 $M:N$，因此有些连接功能需要做额外处理。举个例子，用户通过 Show Processlist 查看连接数，此时他希望看到的是客户端和 OBProxy 之间的连接数，而不是 OBProxy 和 OBServer 之间的连接数。下面我们对常见的连接功能展开详细介绍。

（1）连接黏性。OBProxy 还未实现所有功能的状态同步，如事务状态、临时表状态、cursor 状态等。对于这些功能，OBProxy 只会将后续请求都发往状态开始的节点，这样就不需要进行状态同步，而缺点是无法充分发挥分布式系统的优势。

（2）Show Processlist 和 Kill 命令配套使用。Show Processlist 用于展示客户端和服务端之间的连接，对于 OBProxy，Show Processlist 只展示客户端和 OBProxy 之间的连接，不展示 OBProxy 和 OBServer 之间的连接。Kill 命令用于杀死一个客户端连接，客户端连接关闭后，OBProxy 也会关闭对应的服务端连接。对于 OBProxy 的 Kill 命令，需要先获取对应的 Id，如图 5-2-5 的 Id 列内容（Show Proxysession 和 Show Processlist 功能类似，Show Proxysession 是 OBProxy 专属命令）。

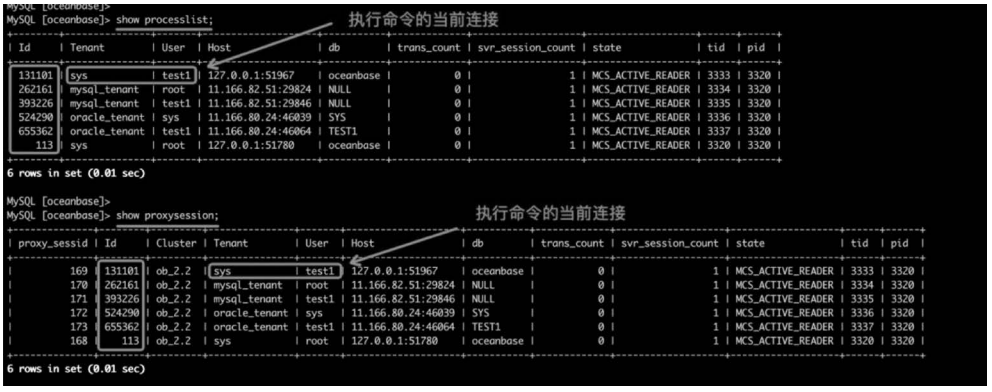

图 5-2-5　连接功能特性

（3）负载均衡影响。因为 OBProxy 对 Show Processlist 和 Kill 命令做了处理，所以 Show Processlist 和 Kill 命令只有都发往同一台 OBProxy 才能正常工作。

2. 查看物理连接

本文介绍两种查看 ODP 上物理连接的方法。

通过 SHOW PROCESSLIST 语句查询当前租户的会话数量及会话 ID。

示例如下：

```
obclient > SHOW PROCESSLIST;
```

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，那么应用就会要求做 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。

通过 SHOW PROXYNET CONNECTION 语句查看 ODP 上所有网络连接的内部属性状态。

SQL 语句如下：

```
SHOW PROXYNET CONNECTION [thread_id [LIMIT xx]]
```

参数说明：

不指定 thread_id 时，展示 ODP 上所有网络连接的内部属性状态。

指定 thread_id 时，展示指定 thread 上的连接状态。支持指定 LIMIT [offset,] rows 和 LIMIT rows OFFSET offset 参数，使格式与 MySQL 完全兼容。当 rows==-1 时，展示全部行。

示例如下：

```
obclient > SHOW PROXYNET CONNECTION\G
```

3. 展示全部 Session

sys 租户通过 SHOW PROXYSESSION 语句，可以查看 ODP 上所有租户连接的全部 Client Session 的内部状态；租户通过 SHOW PROXYSESSION 语句，可以查看 ODP 上当前租户连接的全部 Client Session 的内部状态。

通过 ODP 连接的方式连接 OceanBase 数据库，并执行 SHOW PROXYSESSION 命令：

```
obclient -h xx.xx.xx.xx -uusername@tenant_name#cluster_name -P2883 -p******** -c -A oceanbase
> SHOW PROXYSESSION;
```

4. 展示 Session 详细状态

通过 SHOW PROXYSESSION ATTRIBUTE 语句可以查看指定 Client Session 的详细内部状态，包括该 Client Session 上涉及的相关 Server Session。SQL 语句如下：

```
SHOW PROXYSESSION ATTRIBUTE [id [like 'xxx']]
```

参数说明：

(1) 不指定 id 时，显示当前 Session 的详细状态(ODP 1.1.0 版本起开始支持)，支持模糊查询当前 Session 指定属性名称的 value(Proxy 1.1.2 版本起开始支持)。

(2) 指定 id 时，显示指定 id 对应 Client Session 的详细状态(ODP 1.1.0 版本起开始支持)，支持模糊查询指定属性名称的 value(ODP 1.1.2 版本起开始支持)。

(3) id 既可以是 cs_id，也可以是 connection_id，显示结果相同。

(4) cs_id 为 ODP 内部标记的每个 Client 的 id 号，connection_id 为整个 OceanBase 数据库标记的每个 Client 的 id 号。

(5) like 模糊匹配字段名称，支持'%'和'_'。

5. 展示 Session 变量

通过 SHOW PROXYSESSION VARIABLES 语句可以查看指定 Client Session 的

Session 变量。语句的详细信息如下：

```
SHOW PROXYSESSION VARIABLES [[all] id [like 'xx']]
```

参数说明：

(1) 不带 all 参数时，展示指定 Client Session 的本地 Session 变量(包括修改过的系统变量和用户变量)。

(2) 带 all 参数时，展示指定 Client Session 的全部 Session 变量(包括所有系统变量和用户变量)。

(3) 不指定 id 时，显示当前 Session 的 Session 变量(ODP 1.1.0 版本起开始支持)，支持模糊查询当前 Session 指定属性名称的 value(Proxy 1.1.2 版本起开始支持)。

(4) 指定 id 时，显示指定 id 对应 Client Session 的 Session 变量(ODP 1.1.0 版本起开始支持)，支持模糊查询指定属性名称的 value(ODP 1.1.2 版本起开始支持)。

(5) id 既可以是 cs_id，也可以是 connection_id，显示结果相同。

(6) cs_id 为 ODP 内部标记的每个 client 的 id 号，connection_id 为整个 OceanBase 数据库标记的每个 Client 的 id 号。

(7) like 模糊匹配，支持'%'和'_'。

6. 终止 Server Session

可以通过 KILL PROXYSESSION(cs_id|connection_id)语句终止指定 Client Session。

参数说明：

(1) cs id 为 ODP 内部标记的每个 Client 的 id 号，connection_id 为整个 OceanBase 数据库标记的每个 Client 的 id 号。

(2) 与 KILL connection_id 的作用一致。有关 KILL 语句的详细介绍，参见 KILL(MySQL 模式)或 KILL(Oracle 模式)。

也可以通过 KILL PROXYSESSION(cs_id|connection_id)ss_id 语句，来终止指定 Client Session 上的 Server Session。

参数说明：

(1) id 既可以是 cs_id，也可以是 connection_id，显示结果相同。

(2) cs_id 为 ODP 内部标记的每个 Client 的 id 号，connection_id 为整个 OceanBase 数据库标记的每个 Client 的 id 号。

(3) ss_id 表示 ODP 内部标记每个 Server 端会话(Server Session)的 id 号，可以从 SHOW PROXYSESSION ATTRIBUTE id 中获取。详细的获取操作参见展示 Session 详细状态。

5.2.3 OBProxy 路由管理

OBProxy 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 数据库各服务器的状态及负载情况，将用户的请求路由到最佳的 OBServer，最大程度地保证了 OceanBase 数据库整体的高性能运转。

开始阅读本节内容之前，先回顾一些路由相关的概念：

(1) Zone。

（2）Region。

（3）Server List。

（4）RS List。

（5）Location Cache。

（6）副本。

（7）合并。

（8）强一致性读/弱一致性读。

（9）读写 Zone/只读 Zone。

（10）分区表。

（11）分区键。

路由实现了根据 OBServer 的数据分布精准访问到数据所在的服务器。同时还可以根据一定的策略将一致性要求不高的读请求发送给副本服务器，充分利用服务器的资源。路由选择输入的是用户的 SQL、用户配置规则和 OBServer 状态，路由选择输出的是一个可用 OBServer 地址。其路由逻辑如图 5-2-6 所示。

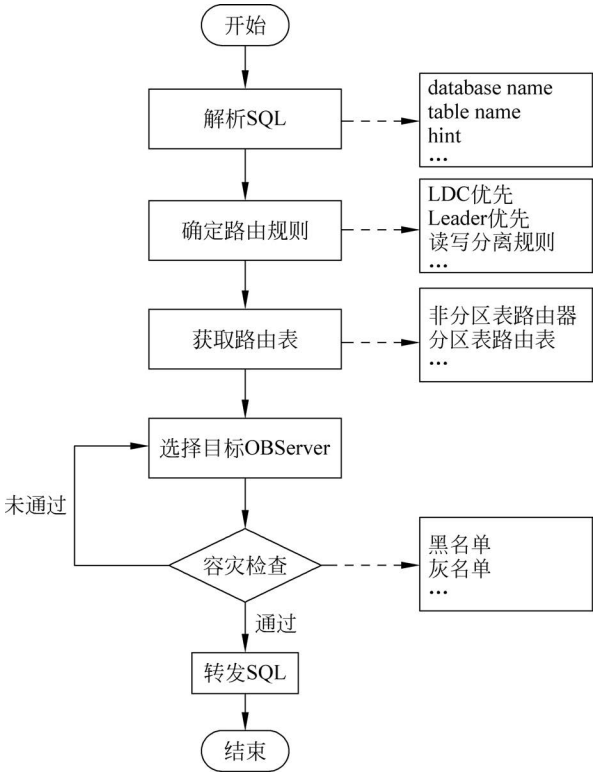

图 5-2-6　OBProxy 的路由逻辑

OBProxy 包含简单的 SQL Parser 功能，可进行轻量的 SQL 解析，即先从客户端发出的 SQL 语句中解析出数据库名和表名，然后根据用户的租户名、数据库名、表名以及分区 ID 信息等信息，向 OBServer 拉取表分区的路由表。

所谓的路由表是指表分区的主/从副本所在 OBServer 的 IP 地址列表信息。有了这些信息之后，OBProxy 将请求路由至主副本所在的服务器上，同时将副本的位置信息更新到自己的 Location Cache 中，当再次访问时，会命中该 Cache 以加速访问。

1. LDC 概述

逻辑数据中心(Logical Data Center,LDC)路由可用于解决分布式关系型数据库多地多中心部署时产生的异地路由延迟问题。

OceanBase 数据库作为典型的高可用分布式关系型数据库,使用 Paxos 协议进行日志同步,天然支持多地多中心的部署方式以提供高可靠的容灾保证。但当真正多地多中心部署时,任何数据库都会面临异地路由延迟问题。

逻辑数据中心(Logical Data Center,LDC)路由正是为了解决这一问题而设计的,通过给 OceanBase 集群的每个 Zone 设置 Region 属性和 IDC 属性,并给 OBProxy 指定 IDC 名称配置项后,当数据请求发送到 OBProxy 时,OBProxy 将按以下优先级顺序进行路由转发。

(1) 选取本机房不在合并的副本。

(2) 选取同地域机房不在合并的副本。

(3) 选取本机房在合并的副本。

(4) 选取同地域机房在合并的副本。

(5) 随机选取非本地域机房不在合并的副本。

(6) 随机选取非本地域机房在合并的副本。

LDC 路由的设置需要进行以下 3 个步骤。

(1) OceanBase 集群的 LDC 配置。

(2) OBProxy 的 LDC 配置。

(3) 应用配置弱一致性读。

2. 非分区表路由算法

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服务器 IP。缓存的有效性有以下三种情况。

(1) 缓存中找不到,此时需要访问 OBServer 查询最新映射并缓存。

(2) 缓存中存在但不可用,此时需要重新去 OBServer 查询并更新。

(3) 缓存中存在且可用,此时可以直接使用。

3. 分区表路由算法

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服务器。

正常情况下,通过分区计算,ODP 可以将 SQL 路由到分区对应的服务器上,从而避免 Remote 执行,提升效率。

在 ODP v3.2.0 版本中,已针对分区表且无法计算分区路由的场景进行优化,由随机选择租户的服务器路由,优化成随机从分布了分区的服务器中随机路由,提升命中率,尽可能减少 Remote 执行。

以二级分区为例,路由分为以下 10 个步骤。

(1) 解析 SQL 获取表名。

(2) 根据表名访问 OceanBase 内部表,确认是分区表。

(3) 解析 SQL 中的列表达式(如 c1=1)。

(4) 访问 OceanBase 内部表获取分区表信息。

(5) 访问 OceanBase 内部表获取一级分区信息。

(6) 根据列表达式计算一级分区的 partition_id。

(7) 访问 OceanBase 内部表获取二级分区信息。

(8) 根据列表达式计算二级分区的 partition_id。

(9) 计算最终的 partition_id。

(10) 访问 OceanBase 内部表获取对应 partition_id 的位置信息。

4. 强一致性读路由策略

在分布式系统中,为了容灾高可用,会采用多副本机制。OceanBase 副本之间需要保证数据一致性,采用了 Paxos 算法。在工程实践中,有一个特殊副本,该副本数据最新,并控制数据在副本间的同步,这个副本叫作主副本,其他副本统称为备副本。

由于 OceanBase 数据库只有一个主副本,因此,主副本路由策略就是发往该副本。此处以 select c1 from t1 语句为例,介绍主副本路由需要满足的两个条件。

(1) SQL 语句操作(查询、插入、更新和删除等)实体表,如上例中的 t1 表。

(2) 请求必须读到最新数据,即强读。

默认的路由策略为强一致性读路由策略,适用于对读写一致性要求高的场景。需要读取分区的 Leader 副本的数据,即 SQL 必须转发到涉及分区的 Leader Server 上执行,以此保证获取到实时最新的数据。

5. 弱一致性读路由策略

和主副本路由相似,备副本路由也需要满足以下两个条件。

(1) SQL 语句查询实体表,如 select c1 from t1 中的 t1 表。

(2) 请求要求弱读即可,即不要求读到最新数据。

这两个条件和主副本路由需要满足的条件是有区别的。

(1) 对于条件 1,备副本路由只支持查询语句,不支持其他语句,这也是 Paxos 算法的实现要求。

(2) 对于条件 2,需要主动设置弱读标记 ob_read_consistency=weak,可以通过 hint、session 等设置。

对于备副本路由,SQL 发往主副本和备副本都可以正常工作,因此备副本路由的选择变多了。

1) 主备均衡路由策略(默认)

OBProxy 默认的路由方式为主备均衡路由。首先,考虑 Region,相同 Region 的 OBServer 优先于不同 Region 的 OBServer;其次,考虑合并状态,不处于合并状态的 OBServer 优先于处于合并状态的 OBServer;最后,考虑 IDC 关系,相同 IDC 的 OBServer 优先于不同 IDC 的 OBServer。

详细的路由顺序如下。

(1) 相同 Region,相同 IDC 并且不处于合并状态的 OBServer。

(2) 相同 Region,不同 IDC 并且不处于合并状态的 OBServer。

(3) 相同 Region,相同 IDC 并且处于合并状态的 OBServer。

(4) 相同 Region,不同 IDC 并且处于合并状态的 OBServer。

(5) 不同 Region 并且不处于合并状态的 OBServer。

(6) 不同 Region 并且处于合并状态的 OBServer。

2) 备优先读策略

OBProxy 支持备优先读路由策略,通过用户级别系统变量 proxy_route_policy 控制备优先读路由。备优先读仅在弱一致性读时生效,且优先读 follower 而非主备均衡选择。使用 root 用户登录集群的 sys 租户后,运行下述语句对用户系统变量 proxy_route_policy 进行设置:

设置命令 SET @proxy_route_policy='[policy]';

验证命令 select @proxy_route_policy;|show proxysession variables;

取值为 follower_first 时,路由逻辑是优先发给备机(即使集群在合并状态)。

取值为 unmerge_follower_first 时,路由逻辑是优先发不在集群合并状态的备机。

3) 读写分离策略

读写分离部署,要求客户端将写请求路由到 ReadWrite zone 主副本,将弱一致性读请求路由到 ReadOnly zone。OceanBase 数据库由于不是所有 zone 里的副本数据都是实时一致的,因此业务对只读操作需接受弱一致性读。

使用 root 用户登录到集群的业务租户,运行下述语句设置 Global 级别系统变量 ob_route_policy 为对应取值即可,默认取值为 readonly_zone_first。读写分离策略如表 5-2-1 所示。

表 5-2-1 读写分离策略

值	路由策略	说明
readonly_zone_first	①本地常态读库 PS→ ②同城常态读库 PS→ ③本地合并读库 PS→ ④同城合并读库 PS→ …	默认值,读库优先访问 优先级:zone 类型>合并状态>IDC 状态
only_readonly_zone	①本地常态读库 PS→ ②同城常态读库 PS→ ③本地合并读库 PS→ ④同城合并读库 PS→ ⑤本地常态读库 TS→ …	只访问读库 优先级:合并状态>IDC 状态
unmerge_zone_first	①本地常态读库 PS→ ②同城常态读库 PS→ ③本地常态写库 PS→ ④同城常态写库 PS→ …	优先访问不再合并的 zone 优先级:合并状态>zone 类型>IDC 状态

注意:PS 表示 OBProxy 中有目标 Partition 的路由信息;TS 表示 OBProxy 中没有目标 Partition 的路由信息,只有租户的路由信息。

读写分离部署应注意的事项如下: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 进行建连,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，然后关闭连接，下次请求又继续重复。

5.2.4　OBProxy 运维管理

1. OBProxy 重启

OBProxy 部署成功后，可以通过 OCP 启动单个服务器上的 OBProxy 进程。

详细的操作步骤如下：

首先，登录 OCP 集群，找到 OBProxy 集群。OBProxy 页面如图 5-2-7 所示。

图 5-2-7　OBProxy 页面

其次，进入 OBProxy 集群，找到需要重启的 OBProxy 主机。最后选择重启，如图 5-2-8 所示。

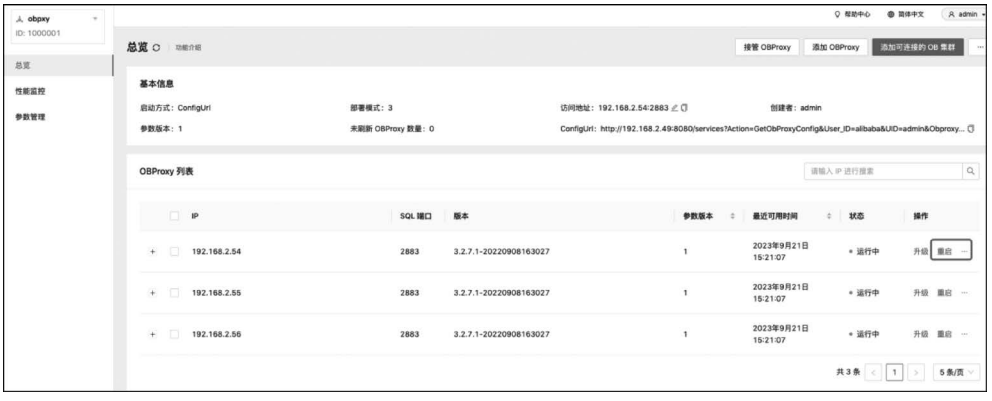

图 5-2-8　OBProxy 集群页面

2. 命令行停止 OBProxy

OBProxy 部署成功后，默认进程为启动状态，可以通过命令行停止单个 OBProxy 进程。

停止 OBProxy 进程：使用 admin 用户登录到 OBProxy 进程所在的服务器。执行以下命

令，查看 OBProxy 进程的进程号。

```
$ps -ef|grep obproxy
admin     37360         0     6 11:35 ?          00:00:09 bin/obproxy
admin     43055     36750     0 11:37 pts/10     00:00:00 grep --color=auto obproxy
root      85623         1     0 Jun02 ?          00:15:19
/home/admin/ocp_agent/obagent/obstat2 -o http://xx.xx.xx.xx:81 -c test323 __obproxy__ -f 20
```

查询到 OBProxy 的进程号为 37360。

执行以下命令，根据查询到的进程号，停止 OBProxy 进程。

```
$kill -9 37360
```

停止成功后，再次执行以下命令，确认 OBProxy 进程已不存在。

```
$ps -ef|grep obproxy
admin     45795     36750     0 11:39 pts/10     00:00:00 grep --color=auto obproxy
root      85623         1     0 Jun02 ?          00:15:19
/home/admin/ocp_agent/obagent/obstat2 -o http://xx.xx.xx.xx:81 -c test323 __obproxy__ -f 20
```

3. OBProxy 捕获慢查询

OBServer 的慢查询阈值默认为 100ms，表示超过 100ms 的慢查询才会记录在 OBServer 上。而 OBProxy 也有其慢查询的阈值，通过设置 OBProxy 的慢查询阈值，可以捕获到通过该 OBProxy 连接的相关慢查询。

1）OBProxy 慢查询相关参数

OBProxy 有慢查询日志打印功能，通过 OBProxy 的以下配置项来控制打印到日志中的 SQL 或事务的处理时间阈值：

（1）slow_transaction_time_threshold：指慢查询或事务的整个生命周期的时间阈值，超过了该时间，就会打印相关日志。

（2）query_digest_time_threshold：指 OBProxy 得到 SQL 直到返回给客户端之前的这段时间的阈值，超过了该时间，也会打印相关日志。

（3）slow_query_time_threshold：OBServer 上慢查询的阈值，默认是 500ms，超过 500ms 的 SQL 会打印相关的日志。该参数应当与 OBServer 的配置项 trace_log_slow_query_watermark 设置为相同的值。

2）修改 OBProxy 慢查询配置项

通过 OBProxy 登录 OceanBase 数据库的 sys 租户。查看 OBProxy 当前配置：

```
obclient>SHOW proxyconfig LIKE 'slow_transaction_time_threshold';
obclient>SHOW proxyconfig LIKE 'slow_query_time_threshold';
obclient>SHOW proxyconfig LIKE 'query_digest_time_threshold';
```

根据实际需求修改 OBProxy 配置项：

```
obclient>ALTER proxyconfig SET slow_transaction_time_threshold='100ms';
obclient>ALTER proxyconfig SET query_digest_time_threshold='5ms';
```

一般情况下，只需要修改配置项 slow_transaction_time_threshold。配置项 slow_proxy_process_time_threshold 使用默认值 2ms，slow_query_time_threshold 使用默认值 500ms 适

用于绝大多数场景。

慢查询举例：

```
[2023-08-20 22:39:00.824392] WARN [PROXY.SM] update_cmd_stats (ob_mysql_sm.cpp:4357)
[28044][Y0-7F70BE3853F0] [14]
Slow query:
client_ip=127.0.0.1:17403          // 执行 SQL client IP
server_ip=192.168.2.54:45785       // SQL 被路由到的 OBServer
conn_id=2147549270
cs_id=1
sm_id=8
md_size_stats=
client_request_bytes:26            // 客户端请求 SQL 大小
server_request_bytes:26            // 路由到 OBServer SQL 大小
server_response_bytes:1998889110   // OBServer 转发给 OBProxy 数据大小
client_response_bytes:1998889110   // OBProxy 转发给 client 数据大小
cmd_time_stats=
client_transaction_idle_time_us=0 // 在事务中该条 SQL 与上一条 SQL 执行结束之间的间隔时间，即
                                  // 客户端事务中 SQL 间隔时间
client_request_read_time_us=0      // OBProxy 读取客户端 SQL 耗时
server_request_write_time_us=0     // OBProxy 将客户端 SQL 转发给 OBServer 耗时
client_request_analyze_time_us=15  // OBProxy 解析本条 SQL 消耗时间
cluster_resource_create_time_us=0 //如果执行该条 SQL，需要收集 OB cluster 相关信息(一般发生在
                                  // 第一次连接到该集群的时候)，建立集群相关信息耗时
pl_lookup_time_us=3158             // 查询 partition location 耗时
prepare_send_request_to_server_time_us=3196 // 从 OBProxy 接收到客户端请求，到转发到 OBServer
                                            // 执行前总计时间，正常应该是前面所有时间之和
server_process_request_time_us=955 // OBServer 处理请求的时间，对于流式请求就是从 OBServer 处
                                   // 理数据，到第一次转发数据的时间
server_response_read_time_us=26067801 // OBServer 转发数据到 OBProxy 耗时，对于流式请求
//OBServer 是一边处理请求，一边转发数据(包含网络上的等待时间)
client_response_write_time_us=716825    // OBProxy 将数据写到客户端耗时
request_total_time_us=26788969          // 处理该请求的总时间
sql=select * from sbtest1               // 请求 SQL
```

4. OBProxy 常见问题

(1) 启动失败问题的排查步骤。

① 服务器是否存在 hostname。输入 hostname-i，确认 host ip 是否存在。

② 目录是否存在，权限是否正确。确保当前目录下有读、写、执行的权限。

③ 端口是否被占用。使用 obproxyd.sh 启动 OBProxy，使用的端口为 2883。

④ 启动环境是否指定正确。如果通过 obproxyd.sh 启动，需要使用-e 参数指定 OBProxy 运行环境。

(2) OBProxy 连接问题的排查步骤。

① 尝试直连 OceanBase 数据库，看是否可以连接成功。注意直连 OceanBase 数据库时不需要带集群名，且端口号默认为 2881。

```
[admin@hostname/] $ obclient -h $IP -uroot@sys -P2881 -p
```

② 如果直连可以成功，确认 OBProxy 管理员账户 proxyro 是否创建。

执行以下命令创建 proxyro 用户，如果命令执行成功，则表示连接失败是由于 OBProxy 管理员账户未创建。

```
obclient > CREATE USER proxyro IDENTIFIED BY 'xxxxxxxx';
```

(3) OBProxy 应该部署在哪里?

OBProxy 是一个反向代理,可以部署在任意节点(OBServer 服务器、应用服务器、网络中其他服务器等)。OBProxy 是通过第一次启动时,指定 OceanBase 集群的 RootService 地址完成集群注册的,因此可以不关注具体部署在哪些服务器上。

(4) 通过 OBProxy 可以对 OceanBase 数据库进行探活吗?

OceanBase 数据库不具有探活机制,但 OBProxy 会定期从 OceanBase 集群中获取 rs_list。

5. OBProxyTCP 参数配置

通过前面章节的学习,我们了解了 OBProxy 的操作步骤及其原理,如果想了解更深层次的连接原理,我们需要追溯到 TCP 协议。因为 OBProxy 的连接基于 TCP 协议,了解 TCP 协议的连接机制就能更透彻地掌握连接管理的功能及连接问题定位等知识。

OBProxy 在代码层面设置了 TCP 的 no_delay 和 keepalive 属性,以保证低延迟、高可用等特性。

(1) no_delay。属性通过禁用 TCP Nagle 算法解决延迟问题。在 Linux 的网络栈中默认启用 Nagle 算法,用于解决网络报文小分组出现,但会导致网络报文发送延迟。我们曾在生产环境中遇到 TCP 未禁用 Nagle 算法的情况,导致一条 SQL 发送耗时 40ms 左右,这是不满足业务要求的。

(2) keepalive 属性用于故障探测。及时发现服务器故障,将无效的连接关闭,使 TCP 层高可用一部分。

这两个属性可以通过 OBProxy 的配置项进行配置,推荐配置如下:

```
## 和 OBServer 的 TCP 连接设置,主要控制 ODP → OBServer 节点连接机制的开启
sock_option_flag_out = 3; --这是个二进制位参数,bit 0 表示不启用 no_delay,bit 1 表示启用
keepalive.3 的二进制是 11,表示启用 no_delay 和 keepalive
server_tcp_keepidle = 5; -- 启动 keepalive 探活前的 idle 时间,5s
server_tcp_keepintvl = 5; --两个 keepalive 探活包之间的时间间隔,5s
server_tcp_keepcnt = 2; -- 最多发送多少个 keepalive 包,2 个.最长 5+5*2=15s 发现 dead_socket

## 与客户端的 TCP 连接设置,主要控制 client → ODP 连接机制的开启
client_sock_option_flag_out = 2; -- 启用 keepalive,不启用 no_delay
client_tcp_keepidle = 5; -- 同上
client_tcp_keepintvl = 5; -- 同上
client_tcp_keepcnt = 2; -- 同上
```

6. JDBC 连接超时参数配置

我们在排查问题时,偶尔会遇到 TCP 连接断了的情况,但无法确定是客户端断连还是服务端断连。根据经验判断,往往是超时机制触发的。这里介绍几种超时机制。

1) socketTimeout

socketTimeout 是 Java Socket 的超时时间,指的是业务程序和后端数据库之间 TCP 通信的超时时间,如果业务程序发送一个 MySQL Packet 后,超过 socketTimeout 的时间还没有从后端数据库收到 Response 报文,这时候 JDBC 会抛出异常,程序执行失败。

socketTimeout 单位是毫秒，可以通过以下方式设置：

```
jdbc:mysql://$ip:$port/$database?socketTimeout=60000
```

2）connectTimeout

connectTimeout 是业务程序使用 JDBC 跟后端数据库建立 TCP 连接的超时时间，也相当于是在 connect 阶段的 socketTimeout，如果 TCP 连接超过这个时间没有创建成功，JDBC 会抛出异常。connectTimeout 的单位是毫秒，可以通过以下方式设置：

```
jdbc:mysql://$ip:$port/$database?socketTimeout=60000&connecTimeout=5000
```

3）queryTimeout

queryTimeout 是业务程序执行 SQL 时，JDBC 设置的本地的超时时间。在业务调用 JDBC 的接口执行 SQL 时，JDBC 内部会开启这个超时机制，超过 queryTimeout 没有执行结束，JDBC 会在当前连接上发送一个 Kill query 给后端数据库，并给上层业务抛一个 MySQLTimeoutException。queryTimeout 单位是秒，可以通过 JDBC 的 Statement.setQueryTimeout 接口来设置：

```
int queryTimeout = 10;
java.sql.Statement stmt = connection.CreateStatement();
stmt.setQueryTimeout(queryTimeout);
```