|  |  |
| --- | --- |
| 产品名称 | 密级 |
| CEDAR | 机密 |
| 产品版本 | 共页 |
|  |

CEDAR Scalable Commit设计文档

CEDAR：Scalable Commit

|  |  |  |  |
| --- | --- | --- | --- |
| 拟制 |  | 日期 | yyyy-mm-dd |
| 审核 |  | 日期 | yyyy-mm-dd |
| 批准 |  | 日期 | yyyy-mm-dd |



**OceanBase项目组**

**2016年6月18日**

数据科学与工程研究院保密资料 请勿外泄

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 修订版本 | 修改描述 | 作者 |
| 2016-06-21 | 1.0 | 初稿 | 周欢 胡爽 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1. 概述 4](#_Toc454279455)

[1.1 目的 4](#_Toc454279456)

[1.2 文档结构 4](#_Toc454279457)

[2. 环境约束 5](#_Toc454279458)

[2.1 开发系统环境 5](#_Toc454279459)

[2.2 测试系统环境 5](#_Toc454279460)

[2.3 名词解释 5](#_Toc454279461)

[3. 概要设计 5](#_Toc454279462)

[3.1 OceanBase事务执行流程 5](#_Toc454279463)

[3.2 OceanBase事务提交瓶颈 6](#_Toc454279464)

[3.3 Scalable Commit总体架构 8](#_Toc454279465)

[4. 详细设计 10](#_Toc454279466)

[4.1 占位设计 10](#_Toc454279467)

[4.1.1 设计概述 10](#_Toc454279468)

[4.1.2 设计实现 11](#_Toc454279469)

[4.2 日志缓冲区设计 12](#_Toc454279470)

[4.3 特殊写事务设计 14](#_Toc454279471)

[4.4 提交线程设计 15](#_Toc454279472)

[4.4.1 设计概述 15](#_Toc454279473)

[4.4.2 工作线程的私有队列 16](#_Toc454279474)

[4.4.3 日志持久化线程 16](#_Toc454279475)

[4.4.4 数据结构和函数 17](#_Toc454279476)

[5. 注意事项 19](#_Toc454279477)

[6. 修改文件一览 19](#_Toc454279478)

[7. 总结 19](#_Toc454279479)

# 概述

## 1.1 目的

OceanBase是可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务并支持数十万的TPS和数百万的QPS。其内存事务引擎使用多版本并发控制实现读写互不阻塞，提供快照读的事务隔离级别；使用经典的两阶段行锁避免写写冲突，提供读已提交的事务隔离级别；使用logging ahead方法记录redo log来持久化事务，并且通过实时同步备机来保证服务的持续可用。事务执行分为3个阶段，分别是事务处理，事务提交和事务发布，其中事务处理阶段保证了事务的一致性和隔离性，事务提交阶段保证了事务的原子性和持久性。虽然OceanBase采用成组提交，预解锁等技术来优化事务提交性能，但是单线程处理事务提交的设计仍然限制了内存事务引擎UpdateServer的提交性能。为了提高UpdateServer的吞吐量，我们使用可扩展的事务提交（Scalable Commit）技术来优化事务提交过程。

本文档详细介绍了Scalable Commit技术的设计与实现，读者通过对本文档的阅读与学习，能够明了其技术原理与实现。

## 1.2 文档结构

第一部分介绍文档的目的与结构。

第二部分介绍开发环境与名词解释。

第三部分介绍Scalable Commit概要设计。

第四部分介绍Scalable Commit详细设计。

# 环境约束

## 2.1 开发系统环境

|  |  |
| --- | --- |
| **软件产品** | **软件版本** |
| OB开发版本 | CEDAR Version:0.1 |
| 服务器操作系统 | CentOS 6.5 内核: 2.6.32-431.el6 |
| 编程语言 | C++ |
| 开发工具 | gcc ; vim |
| 测试工具 | gdb; gtest |

## 2.2 测试系统环境

|  |  |
| --- | --- |
| **软件产品** | **软件版本** |
|  |  |
|  |  |

## 2.3 名词解释

* 主控服务器（RootServer, RS）:OceanBase集群的主控节点，负责管理集群中的所有服务器，以及维护tablet信息。
* 更新服务器（UpdateServer, UPS）:OceanBase的内存事务引擎。
* 基准数据服务器（ChunkServer, CS）:OceanBase的分布式存储引擎。
* 合并服务器（MergeServer, MS）:负责接收并解析客户端的SQL请求，经过词法分析、语法分析、查询优化等一系列操作后发送到CS和UPS，最后合并请求结果。

# 概要设计

## 3.1 OceanBase事务执行流程

OceanBase数据库的内存事务引擎采用Flush Pipelining将事务执行过程分为3个阶段：事务处理，事务提交和事务发布，分别对应三种线程事务处理线程（worker）、事务提交线程（committer）和事务发布线程（publisher），每一个阶段完成之后将当前任务压入下一个阶段的任务队列之后再处理下一个任务请求。OceanBase的事务执行流程如图1：

1. 事务处理阶段，由多线程对并发的事务执行预处理，包括加行锁，进行逻辑判断（是否能执行insert等操作）；然后将待更新数据写入事务上下文的临时空间；之后在事务提交时确定每个待提交事务的事务版本号（trans\_id），并按trans\_id的先后顺序将任务压入提交队列；最后采用Early Lock Release技术提前释放行锁，维护多版本并发控制信息（更新memtable链表和版本号）；
2. 事务提交阶段，由单线程处理当前待提交的事务，包括更新系统已提交的事务版本号（trans\_committed\_id）、生成日志校验和（checksum）、并按事务先后顺序确定日志信息（log\_id,file\_id和file\_offset）;然后将事务日志顺序地拷贝到集中式日志缓冲区中；之后当缓冲区中的日志大小超过2M或者系统空闲时，单线程触发日志写盘和同步备机操作；同时提交线程循环地判断同步到备机的最大日志号，确定已经完成提交的事务，然后更新系统的发布版本号（trans\_published\_id）和memtable信息（row\_counter\_、checksum和last\_trans\_id）,最后将已完成提交的事务加入事务发布线程池中；
3. 事务发布阶段，由多线程对已完成提交的事务进行处理，包括释放事务上下文并响应客户端。



图1 OceanBase 事务执行流程

在事务提交方面，OceanBase较于传统数据库不仅避免了事务处理线程因为日志I/O而触发的上下文切换（尤其在如今计算机的物理核数不断增大的情况下）而且减少了系统内部对共享数据（日志缓冲区）的冲突访问。

## 3.2 OceanBase事务提交瓶颈

OceanBase采用流水线式事务执行架构虽然避免了传统事务执行架构的弊端，但单线程处理事务提交的设计使得系统性能受限于提交线程的处理能力。随着客户端auto-commit事务请求的增多，大量事务将因提交线程性能达到极限而堆积在提交队列中，而此时事务处理线程和事务发布线程均未达到极限。因此单线程提交限制了系统整体的事务吞吐量。

图2，3所示是在8个MS、3个CS和1个RS、UPS的集群环境下进行replace操作的实验结果。实验结果显示，当OceanBase的处理性能（TPS）达到13万之后，事务在提交队列中的等待时间由1519us突增到2539us,而此时事务提交线程的CPU利用率已达到97%。由此可以看出，当前OceanBase提交线程的处理量在13万左右，当超过13万之后大量事务在提交队列中堆积从而导致吞吐量降低最后趋于稳定。

图2 事务提交等待时间

图3 OceanBase replace处理性能

通过对事务提交线程处理函数void TransExecutor::*handle\_commit*(void \*ptask, void \*pdata)的详细分析可知事务提交过程主要包括更新已提交事务号id（update\_committed\_id）、生成memtable校验和（checksum）、填充日志缓冲区（fill\_log）、写磁盘并同步备机（commit\_log）和发布事务（publish）操作。各操作的主要任务如下：

1. Update\_committed\_id更新已提交事务号

事务在提交时首先获取当前事务的上下文session\_ctx，并修改事务管理器SessionMgr已提交的事务号commited\_trans\_id\_；

1. 生成memtable校验和

计算事务执行前后memtable的校验和并写入日志，从而保证主备日志的一致性。校验和的计算过程为：根据当前事务的事务版本号trans\_id计算出uc\_checksum，再根据uc\_checksum和memtable的uncommitted\_checksum计算出事务执行之后memtable的cheksum；

1. Fill\_log填充日志缓冲区

由日志管理器ObUpsLogMgr生成事务日志信息（log\_id,file\_id和file\_offset）并填入集中式日志缓冲区ObLogGenerator,最后根据日志号将当前任务压入发布队列flush\_queue\_中，等待日志写磁盘并同步备机成功之后被唤醒;

1. Commit\_log日志写盘并同步备机

当满足group\_commit条件（当缓冲区中的日志大小超过2M或者系统空闲）时，先将日志缓冲对齐（write\_nop），必要时进行日志切换（switch\_log），再插入结束标志符（eof）；然后读取缓冲区中的日志并将其发送备机并写入本地磁盘，最后清空缓冲区（commit）并完成*write\_log\_hook*；

1. Publish发布事务

提交线程循环地读取flush\_queue\_队列中任务的日志号log\_id，并将其与备机已完成日志同步的日志号c\_log\_id进行比较。如果log\_id<c\_log\_id，则表示该事务已完成提交操作，可以被唤醒并响应客户端。提交线程对每个已完成提交的事务进行以下操作：首先获取事务的上下文，并用事务版本号(trans\_id)更新事务管理器SessionMgr的published\_trans\_id\_(可见的事务版本号)，同时更新memtable信息；然后将事务压入CommitEndHandlePool响应客户端线程池的队列；最后从flush\_queue\_队列中取出。

通过统计事务提交过程中各操作消耗的时间可以发现，随着客户端数量的增多（UPS压力的逐渐增大），update\_commiited\_id和生成checksum时间基本不变，commit\_log时间越来越少，而fill\_time和publish时间却逐渐增多，最后稳定在一定的范围内。如图5所示为2000客户端时各操作的时间比，其中fill\_log和publish操作所占时间比例最大，分别为42%和37%。Fill\_log操作中的大量内存拷贝和publish操作中的大量循环判断是限制提交线程处理性能的主要原因。

图4 事务提交各操作函数时间比

## 3.3 Scalable Commit总体架构

为了提高提交线程的处理能力，需要优化提交过程中的大量内存拷贝和计算，在保证日志提交顺序的基础上，实现多线程填充日志缓冲区并减少对待发布事务的计算。在集中式内存事务引擎中，Logging是唯一的全局同步点，因此在updateserver事务提交阶段需要严格保证日志的先后顺序，即日志内容在日志缓冲区中的位置。为了实现多线程并发填充缓冲区，工作线程需要在填充缓冲区之前确定日志顺序和日志内容，主要包括日志号（log\_id/LSN）、事务版本号（trans\_id）、日志所在文件号（file\_id）、日志所在的文件偏移（file\_offset）以及memtable校验和。具体实现过程中需要满足以下条件：

1. 并发释放行锁，维护多版本控制信息之前必须决定log\_id,trans\_id,其中trans\_id是递增的时间戳；
2. 并发填充缓冲区（fill\_log）之前，必须给日志分配一个缓冲区位置，这步操作成为占位操作；
3. 并发填充缓冲区之前必须决定memtable的checksum；
4. 必须提前给NOP和SWITCH\_LOG日志预留log\_id和缓冲区位置；
5. 写日志之前必须决定日志所在的file\_id和file\_offset;
6. 每次写盘的日志不能超过2M。

为了充分利用多核扩展性实现并发的关键是用原子操作来决定日志顺序。

优化之后的UpdateServer中存在两种线程分别为多个事务处理线程（workers）和单事务提交线程（committer），处理线程负责并发处理事务、占位、解行锁、生成日志、填充缓冲区、提交写盘任务，最后响应客户端并回收事务上下文，事务提交线程负责日志写盘、同步备机、判断待发布的事务并唤醒工作线程响应客户端。在Scalable Commit架构中，事务执行过程分为四个阶段分别是事务处理、事务预提交、事务提交和事务发布。具体的执行流程如图5：

1. 事务处理阶段，由多线程对并发的事务执行预处理，包括加行锁，进行逻辑判断（是否能执行insert等操作）；然后将待更新数据写入事务上下文的临时空间；
2. 事务预提交阶段，当事务提交时多线程并发的为待提交事务抢占缓冲区的位置；如果抢占成功之后则生成日志信息，包括log\_id、trans\_id、file\_id、file\_offset和checksum，如果不成功则继续抢占；然后采用Early Lock Release技术提前释放行锁，维护多版本并发控制信息（更新memtable链表和版本号）；最后填充缓冲区并将预提交事务压入工作线程的私有队列中等待提交线程唤醒，再由缓冲区中最后一个完成日志填充的工作线程更新trans\_committed\_id并将缓冲区作为一个任务压入提交线程队列中；
3. 事务提交阶段，由单线程处理当前提交线程队列中的任务，如刷磁盘并同步备机；同时提交线程循环地判断同步到备机的最大日志号，确定已经完成提交的事务，然后用已完成提交事务的最大trans\_id更新系统的发布版本号（trans\_published\_id）并清空当前缓冲区，最后唤醒工作线程私有队列中等待的事务；
4. 事务发布阶段，由多线程循环地判断私有队列中事务是否被唤醒，如果被唤醒则多线程并发的更新memtable信息（row\_counter\_、checksum和last\_trans\_id）,然后释放事务上下文并响应客户端。



图5 Scalable Commit 事务执行流程

Scalable Commit采用原子操作进行并发占位保证了日志的连续性，实现多线程并发填充缓冲区提高了事务提交的性能。

# 详细设计

## 4.1 占位设计

### 4.1.1 设计概述

占位操作是指工作线程并发地抢占缓冲区位置，以便在填充缓冲区之前根据位置信息生成日志内容，包括log\_id,、trans\_id、file\_id、file\_offset和checksum。并发的关键是用原子操作决定id和偏移，但是上述所有id和偏移都至少是64bit，无法用一个原子操作决定。为了实现并发性，减少抢id和offset的冲突，占位操作主要的设计思路如下：

1. 将待提交的事务分成多个group并保证每个group内的log\_id和trans\_id都连续。所以只需确定group内第一个事务的log\_id和trans\_id，其余事务的log\_id和trans\_id可以根据自己在group内的相对编号(rel\_id\_)计算出来；
2. 每个group对应一个2M log\_buffer，log\_buffer的总数为GROUP\_ARRAY\_SIZE。每个group有一个递增且连续的group\_id，根据group\_id%GROUP\_ARRAY\_SIZE可以定位group对应log\_buffer的起始位置。Group内的事务可以根据group\_id和自己在group内的相对偏移(rel\_offset\_)计算出填充缓冲区的位置；
3. group内的日志是成组提交的，因此在写磁盘时，只需确定group内第一个事务和最后一个事务的file\_id、file\_offset（对应OB中ObLogCursor数据结构）以及group内日志的总长度即可；
4. group之间的memtable checksum需要保证连续。Group之间采用不可交换的算法计算出memtable checksum，而group内的每条日志采用可交换的算法计算出checksum。（注：在当前版本中没有实现checksum）；
5. group内最后一条日志预留给NOP或者SWITCH\_LOG。

因此，占位操作是对group\_id和group内部的编号（rel\_id）和偏移(rel\_offset)的抢占。

### 4.1.2 设计实现

用如下结构体记录占位信息：



其中append(FLogPos& pos, int32\_t len, int64\_t limit, int 64\_t& switch\_flag)函数为计算缓冲区位置，具体说明如下：

* 1. 如果工作线程向当前group追加长度为len的日志后还未超过limit（2M）,则追加成功并返回新追加日志的pos，然后用返回的Pos去抢占位置；
  2. 如果工作线程向当前group追加长度为len的日志后超过limit，则切换到下一个group。当前工作线程需要预留两条日志空间，一是当前group最后一条NOP或者SWITCH\_LOG日志；二是下一个group的第一条日志。成功时返回当前group最后一条日志的pos;
  3. 切换group时，设置swith\_flag =OB\_GROUP\_SWITCHED，并推算出下一个group第一条日志的pos，然后进行CAS抢占;
  4. 处理特殊写任务时，传递一个无效的len,表示冻结当前group，为当前group追加一条NOP或SWITCH\_LOG日志，并切换group;
  5. 实际调用时给append()传递的limit参数需要预留足够的空间写NOP或者SWITCH\_LOG，此时SWITCH\_LOG也有日志对齐功能。

因为ClogPos刚好128bit，所以可以使用CAS保证原子性。具体占位流程如下：

1. 全局变量FLogPos next\_pos\_,初始化时group\_id\_、rel\_id\_、rel\_offset\_都为0；
2. 工作线程根据自己的日志长度len，采用append方法乐观计算出自己的缓冲区位置cur\_pos；
3. 工作线程根据group是否切换，用对应的pos值原子抢占更新全局next\_pos\_，如果失败则返回（2）；
4. 抢占成功，则根据group\_id%GROUP\_ARRAY\_SIZE定位所属的group，生成日志信息并填充缓冲区。

## 4.2 日志缓冲区设计

日志缓冲区是内存中临时存储日志的结构，它由多个2M大小的buffer组成，每个buffer称为一个group。每个group中记录了日志长度len\_、日志个数count\_、起始事务号start\_timestamp\_、起始日志号start\_log\_id\_、group所属文件的起始start\_cursor\_和终止位置end\_cursor以及group状态标识。Group结构体定义如下：



Group结构体说明：

1. 每个group包含四种状态，分别为可重用，可设值，可填充，可提交。当ts\_seq\_ = group\_id时表示group为可重用状态；当last\_ts\_seq\_ = group\_id + READY时表示group为可设值状态；当ts\_seq\_= group\_id + READY时表示group为可填充状态；当log\_cursor\_seq\_ = group\_id + READY时表示group为可提交状态；
2. 初始化时，group数组中第i项的ts\_seq\_ = i表示这些group结构体直接可用。通常情况下，一个group的ts\_seq\_需要等group中所有日志都刷入磁盘并同步备机成功之后才能被重新设置，即由提交线程设置ts\_seq\_ = group\_id + GROUP\_ARRAY\_SIZE;
3. 初始化和主备切换时，更新当前group的start\_log\_id\_和last\_timestamp\_，当前group是指由全局next\_pos\_.group\_id\_指定的group。更新完start\_log\_id\_和last\_timestamp\_之后设置last\_ts\_seq\_ = next\_pos\_.group\_id + READY表示当前group可以设值。通常情况下，一个group B的start\_log\_id\_、last\_timestamp\_和last\_ts\_seq\_是由上一个group A最后一条日志（NOP/SWITCH LOG）的处理线程设置的，线程在设置start\_log\_id等变量之前需要等待group B为可重用状态；
4. 一个group的start\_timestamp\_是由当前group第一条日志的处理线程设置的。线程需等待当前group的last\_ts\_seq\_ = group\_id + READY之后，更新start\_timestamp\_为last\_timestamp\_＋１和当前时间戳中的较大者，然后再设置自己的ts\_seq\_ = group\_id + READY表示当前group为可填充状态；
5. 工作线程在填充日志缓冲区之前需要等待当前group为可填充状态；
6. 初始化和主备切换时，更新当前group的start\_cursor\_。通常情况下，group B的start\_cursor\_是由上一个group A最后一条日志的处理线程设置的，线程在设置start\_cursor\_时需要等待group B为可用状态，然后设置group B的start\_cursor\_等于group A的end\_cursor\_; group B的end\_cursor\_是由自己最后一条日志的处理线程设置的。更新完end\_cursor\_之后设置log\_cursor\_seq\_= group\_id + READY表示当前group为可提交状态；
7. 提交线程刷磁盘之前需要等待当前group为可提交状态；
8. len\_和count\_是由group最后一条日志的处理线程设置；
9. 工作线程填充完日志内容后对当前group的ref\_cnt\_原子加1，当ref\_cnt\_ = count\_时表示当前工作线程是最后一个完成group日志填充的线程，该线程需要发起刷磁盘任务；
10. 在OB系统中，写日志事务类型分为特殊写事务类型和普通事务类型。特殊写事务类型不需要响应客户端因此need\_ack\_ = false,而普通事务类型需要响应客户端因此need\_ack\_=true，通常特殊写事务和普通事务的日志存于不同的group中；
11. OB系统中，事务同步备机分为异步和同步两种模式。通过Sync\_to\_slave来区分两种模式并调用相应的刷磁盘函数（flush\_log/async\_flush\_log）。

Group操作的主要步骤如下如6所示：



图6 缓冲区操作

1. 初始化或主备切换时，更新全局变量next\_group\_.group\_id\_所对应group(G0)的start\_log\_id\_,last\_timestamp\_和start\_log\_cursor\_,然后设置last\_ts\_seq\_=next\_pos\_.group\_id\_+READY,同时group数组中第i项的ts\_seq\_ = i;(标识当前group G0可重用，可设值，G1、G2可重用)；
2. group切换时，填充G0最后一条占位日志线程先设置G0的end\_cursor\_和log\_cursor\_seq = group\_id + READY（标识G0可提交）,然后等G1可重用之后，设置G1的start\_log\_id\_，last\_timestamp\_和start\_cursor\_,然后设置G1的last\_ts\_seq\_=group\_id + READY(标识G1为可设值状态)。填充G1第一条占位日志线程需要等G1的last\_ts\_seq\_=group\_id + READY（标识G1可设值）之后，设置G1的start\_timestamp\_为last\_timestamp\_ + 1和当前时间戳中的较大者，然后设置ts\_seq\_=group\_id + READY（标识G1可填充）;
3. group填充时,G0的最后一个填充线程更新全局的committed\_trans\_id\_ = start\_timestamp\_ + count\_ - 1，然后将G0压入提交线程队列中;
4. group提交时，提交线程需等待当前group为可提交状态，然后等刷盘并同步备机成功之后设置当前group G0的ts\_seq\_ = group\_id + GROUP\_ARRAY\_SIZE（标识G0可重用）。

## 4.3 特殊写事务设计

在OceanBase系统中，与日志相关的写事务分为三类分别为普通写事务、系统维护型写事务和每日合并写事务。原OB对所有写事务的处理逻辑如图7所示

图7 OB写事务处理逻辑

其中黄色标记的是普通写事务如OB\_PHY\_PLAN\_EXECUTE，绿色标记的是系统维护型写事务如OB\_SWITCH\_SCHEMA，红色标记的是每日合并写事务如OB\_FREEZE\_MEM\_TABLE。普通写事务由事务执行器trans\_executor\_.handle\_packet处理，先由事务处理线程处理之后交由事务提交线程进行事务提交；系统维护型写事务先压入写线程队列中，然后再由写线程调用trans\_executor\_.handle\_packet将其交由事务提交线程进行写日志，其中写线程只是作为一个转接口；每日合并写事务先压入写线程队列中，然后再由写线程调用trans\_executor\_.handle\_packet找到其对应的处理函数（处理函数涉及写日志操作），其中写线程为主要工作线程。虽然普通写事务和系统维护型写事务都由事务提交线程处理，但是它们的处理流程却是不一样的。为了提升吞吐量普通写事务采用了成组提交，而为了保证系统的正确性，每处理一个系统维护型写事务就刷盘一次甚至有的维护型写事务在写日志缓冲区之前必须保证当前日志缓冲区为空。同时为了保证事务日志的连续性，OB采用spinlock对写线程和提交线程进行互斥从而保证了同一时刻只有一个线程访问集中式日志缓冲区和刷磁盘。

在Scalable Commit架构中，事务提交线程只负责刷磁盘和同步备机而确定日志顺序，填充集中式日志缓冲区是由处理线程完成的。为了简化写事务逻辑和实现，系统维护型写事务由写线程直接处理。Scalable commit写事务处理逻辑如图8所示，其中写线程处理系统维护型写事务和每日合并写事务时仍然采用CAS占位操作来确定事务日志顺序，在占位之前需要保证当前group为空即切换group，填充日志缓冲区之后切换group并把当前group压入提交线程队列中，再由提交线程处理刷磁盘和同步备机。为了简化写线程的处理逻辑，采用读写锁spinRWLock互斥写线程和处理线程中的占位操作。因为系统维护型写事务出现的频率不高，其中OB\_FAKE\_WRITE\_FOR\_KEEP\_ALIVE是每隔50ms才产生一次，因此采用互斥锁不会对系统性能造成太大的影响。值得注意的是系统维护型写事务中OB\_SEND\_LOG是主UPS同步备机时产生的任务类型，然后由备UPS处处理该事务将主UPS日志缓冲区的内容拷贝到备UPS的内存缓冲区中，因此可以不用加写锁。



图8 Scalable Commit写事务处理逻辑

说明：在scalable commit架构中备UPS写日志时只使用Group[0]即第一个日志缓冲区，因为原OB系统中只有一个日志缓冲区，在不改变备UPS回放与提交流程的基础上仍采用一个日志缓冲区处理备UPS填充日志和刷盘操作。

## 4.4 提交线程设计

### 4.4.1 设计概述

优化的目的在于消除单线程提交性能上的瓶颈。优化后事务提交由多个处理线程和单个日志刷盘线程来完成。提交时，多个处理线程根据事务上下文信息并发生成日志，并对日志缓冲区进行占位和填充，再将上下文信息记入线程的私有队列当中（以等待publish）；多个事务的日志将被填充到一个组（Group）中，日志刷盘线程将各组填入缓冲区buffer的日志持久化、同步备机，并通知一个刷盘成功的日志号，处理线程根据日志号将完成提交的事务publish。



图9

### 4.4.2 工作线程的私有队列

以图9中的S2MQueueThread线程池中某个工作线程thread1为例，thread1拥有用于存储事务上下文信息的私有队列ObCommitQueue：

1. 工作线程得到一个写任务Task，如果是特殊写任务，则在处理该特殊任务后，生成该任务的日志，排他抢占一整个Group，并提交到日志持久化线程；如果是事务处理的写任务，则在处理该任务后，生成的日志与其他线程抢占并填充缓冲区，同时，将事务上下文的信息（通过trans\_id获得）压入私有队列，等待日志持久化成功后做Publish&Free
2. 工作线程会判定日志持久化的成功进度OB\_GLOBAL\_SYNC\_LOG\_ID（刷盘且同步备机后更新），将日志号小于此进度的事务上下文从私有队列逐个弹出，并进行publish、释放上下文以及回复客户端等流程。
3. 其中1,2均在工作现场的函数循环中依次执行，如果工作线程此时未得到任何的写任务Task，则线程thread1会阻塞导致事务未及时提交释放，因此日志持久化线程还会利用回调唤醒signal所有工作线程。



### 4.4.3 日志持久化线程

这部分的基本流程概述为：首先是由单线程依次处理待提交的事务的group，进行刷磁盘和同步备机；在刷磁盘和同步备机完成后更新trans\_published\_id；最后是唤醒已经完成事务提交的处理线程。

以上图中的日志持久化线程GroupQueueThread为例：

1. 日志持久化线程为单线程，拥有一个队列flush\_queue，在每次日志写盘和同步备机后，将该group推入这个队列当中。这样使得group在同步备机成功后及时被重用，并用于更新事务的published\_trans\_id。

handle\_flush\_queue\_task()方法：与备集群通信获取备机同步成功的日志id，与队列flush\_queue中的group比较，小于成功同步的日志号的group被弹出，同时根据其信息将trans\_published\_id更新到group->start\_timestamp\_ + group->count\_ - 1;将日志同步进度赋值给全局变量OB\_GLOBAL\_SYNC\_LOG\_ID；唤醒所有处理线程。

1. 日志持久化线程的主要流程在handle\_group函数中实现：

单线程根据group\_id依次处理待提交的事务，分为两种情况：系统正常工作时间的普通group和系统空闲时间的特殊group。为了防止系统空闲时期flush\_queue中的group信息没有及时处理，导致处理线程没有及时将事务任务提交返回客户端，在空闲时期会在持久化线程中提交一个包含特殊日志号(-1)的group。

如果是特殊的group。则直接调用handle\_flush\_queue\_task()的方法，以及时完成事务的提交。

如果是普通的group，在调用handle\_flush\_queue\_task()完成之后，将当前日志缓冲区中的group进行持久化和同步备机操作。若当前的group的事务需要返回客户端(need\_ack为true时)，将对应的group压入flush\_queue中；若当前的group中的事务不需要返回客户端(need\_ack为false时)，立即释放group。



### 4.4.4 数据结构和函数

#### 4.4.4.1 工作线程

Struct ObCommitQueue：

存储需要返回客户端的普通写事务信息的私有队列，由ObFixQueue经过typedef得到。

Struct ICommitTask：

新构造的一个基类，使写事务任务Task继承此类，便于在common命名空间使用，且能够调用对应的方法，如得到该写任务结果的日志号

Int64\_t OB\_GLOBAL\_SYNC\_LOG\_ID：

以一个全局整型来表示日志持久化线程已经成功完成备机同步日志的id，工作线程将以写事务任务中的完成log\_id与此比较，用以判断该事务任务是否可以完成释放上下文并返回客户端。

*Void handle\_local\_task*(void \*pdata):

工作线程处理私有队列中的事务上下文。其主要的流程为从私有队列中取出一个事务任务，对比该任务生成的log\_id与同步备机完成的log\_id，释放并回复比同步备机的日志小的事务任务。



Void TransHandlePool:: wake\_up(int64\_t idx)：

唤醒工作线程的方法，idx为工作线程的线程号。实际实现中，日志持久化线程将在处理完其flush\_queue队列的group后调用此函数，并唤醒所有的工作线程。

int SessionMgr::*update\_commited\_trans\_id*(int64\_t trans\_id):

更新commited\_trans\_id\_，其值为cur\_group->start\_timestamp\_ + cur\_group->count\_ - 1。使用CAS抢占实现多线程同步。在工作线程将group成功提交给日志持久化线程后调用。

#### 4.4.4.2 日志持久化线程

class GroupQueueThread：

实现中移除了原先的CommitEndHandlePool，将publish和free的流程移至写任务处理工作线程。另外，由于日志持久化处理的是填充在日志缓冲区buffer的group，因此TransCommitThread改为由新实现的此类替代，实现日志持久化的大部分功能。

该线程处理任务队列，这个任务队列允许多线程加入任务，任务队列里的内容为group所在的指针。

LogGroup nop\_pos：

系统空闲时期产生的特殊group，其group\_id为-1，将此组加入日志持久化线程中，用以驱动日志持久化线程唤醒处理线程并及时更新全局日志号。

ObTicketQueue flush\_queue\_：

存储group信息的队列，取出队列中的组信息用以更新published\_trans\_id\_，组信息gtask在日志刷盘之后生成。

int SessionMgr::*update\_published\_trans\_id*(int64\_t trans\_id)

更新published\_trans\_id\_，其值为LogGroup的start\_timestamp\_ + count\_ - 1。在日志持久化线程与备集群通信得到同步成功的日志号后，与flush\_queue中的日志比较，若较小则改为新值。

int TransExecutor::handle\_flush\_queue\_task()：

见4.4.3内容

void TransExecutor::*handle\_group*(void \*ptask, void \*pdata)

处理一个从工作线程提交过来的group，其流程如4.4.3所述。

# 注意事项

* ScalableCommit实现过程中去掉了memtable checksum的生成，并通过修改ob\_update\_server\_config.h中replay\_checksum\_flag的配置信息屏蔽了备机UPS回放时对checksum的校验。

# 修改文件一览

|  |  |
| --- | --- |
| **h** | **cpp** |
| src/common/ob\_log\_entry.h | src/common/ob\_log\_entry.cpp |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| src/common/ob\_define.h |  |
|  |  |
|  |  |
|  |  |

# 总结

该文档主要介绍了Scalable Commit的设计与实现。Scalable Commit是为了提升原OceanBase提交线程性能而提出的一种可扩展性并行事务提交，该方案采用CAS原子操作来保证日志顺序，使得事务处理线程和提交线程得到最大化的并行从而提升了OceanBase整体的提交性能。