**日志同步详细设计文档**

**Table of Contents**

* [1 综述](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-1)
  + [1.1 名词解释](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-1-1)
  + [1.2 系统结构](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-1-2)
  + [1.3 同步信息](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-1-3)
  + [1.4 commit log同步的目标](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-1-4)
* [2 commit log同步状态和主要流程](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2)
  + [2.1 UPS状态](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-1)
  + [2.2 启动流程分解](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-2)
  + [2.3 日志同步的相关线程，服务和任务](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-3)
  + [2.4 各种服务什么时候可用](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-4)
  + [2.5 全局的一次性初始化工作](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-5)
  + [2.6 确定角色为主之后](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-6)
  + [2.7 确定角色为备之后](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-2-7)
* [3 写线程处理写请求的过程](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-3)
* [4 备UPS追赶主UPS的流程](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-4)
  + [4.1 replay线程的内部状态和处理逻辑](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-4-1)
  + [4.2 retrieve\_replay\_log(start\_cursor,...) 的实现](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-4-2)
  + [4.3 收日志线程的处理流程](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-4-3)
  + [4.4 recent\_log\_cache 的实现](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-4-4)
  + [4.5 fetched\_log\_buffer 的管理](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-4-5)
* [5 commit log获取服务](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-5)
  + [5.1 取日志协议](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-5-1)
  + [5.2 取日志请求处理流程](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-5-2)
* [6 ObPosLogReader](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-6)
* [7 ObLogLocator 的实现](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-7)
  + [7.1 位置缓存](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-7-1)
  + [7.2 实际定位](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-7-2)
* [8 recent\_log\_cache](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-8)
  + [8.1 模块描述](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-8-1)
  + [8.2 关键数据结构和算法](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-8-2)
* [9 如何获得本地最大的日志编号](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-9)
* [10 兼容0.21版UPS方案](http://10.232.35.40:8001/home/yuanqi.xhf/org/archive/log-sync-doc/log-sync-detail3.html#sec-10)

**1 综述**

UPS是OB的单点更新模块，为了保证可靠性，采用主备实现实时同步。 UPS采用单线程更新(更新包括对数据和元数据的修改)，每次更新操作会产生一条commit log，并追加到已有的commit log序列之后。完整的commit log序列代表了OB的更新历史。 UPS通过持久化commit log实现写操作持久化, 通过同步commit log实现主备复制。

本文描述了commit log同步方案的详细设计。

**1.1 名词解释**

1. OB：全称OceanBase，核心系统研发部的海量存储系统；
2. UPS：全称Update Server，OB系统更新子系统，通过主备保证可靠性；
3. Commit Log：操作日志，UPS为了保证可靠性，首先将更新操作序列化到主备机的文件中，称为操作日志；
4. UPS Master(Master)：全称Update Server Master，处理实际读写请求的Update Server；
5. UPS Slave(Slave)：全称Update Server Slave，用于实时备份的Update Server；
6. clog cursor：commit log代表了写操作历史，在灾难恢复和主备同步时， 需要表示恢复的进度或同步的进度，这就需要明确日志点的表示方法。 一般来说每条日志都有唯一的日志编号，连续的日志项，其编号也连续( SWITCH\_LOG 是例外，它与上一条日志相同)。 日志会不停的追加，显然无法把所有日志保存到单个文件里，所以当一个日志文件过大时，就需要新产生一个日志文件。 多个日志文件放在同一个目录下，每个日志文件都用一个序号区分，日志文件的编号是连续的。

日志点由日志ID唯一的确定。

typedef int64\_t ObLogCursor;

**1.2 系统结构**

一个应用可能会用到多个互为热备的OB instance, 这些instance有主备之分，任一时刻最多只有一个为主instance，其余为备instance。一个OB instance内部包含至少一个UPS(通常情况下为多个)，这些UPS也有主备之分，任一时刻最多只有一个为主UPS，其余为备UPS。只有主instance的主UPS可以提供写服务，接收客户端写请求，并生成commit log，然后通过日志同步机制将commit log复制到各个UPS。备instance的主UPS同步主instance的主UPS的commit log，instance内备UPS同步主UPS的commit log，最终实现所有UPS的状态一致。

OB instance的主备和instance内UPS的主备由RS管理，在运行过程中UPS的角色会发送变化。

**1.3 同步信息**

以下信息需要在主备UPS之间同步

1. 所有提交过的写事务
2. 日志文件的切分点，

日志的切分点通过 SWITCH\_LOG 同步到备。

1. Major/Minor freeze点和版本号: Major/Minor freeze作为一种特殊的写操作类型被同步到备UPS
2. schema: 切换schema也作为一种特殊的写操作同步到备UPS。

对日志同步而言，需要注意的是备UPS写日志时需要处理 SWITCH\_LOG , 重放日志时需要处理 Major/minor freeze 及 switch schema 操作。

**1.4 commit log同步的目标**

1. 支持多个备UPS连接同一个主UPS
2. 区分两种备机，主UPS给备机push新生成的日志时，一种需要主UPS等响应，称为实时备机，一种不需要主UPS等响应，称为非实时备机。

**2 commit log同步状态和主要流程**

**2.1 UPS状态**

对日志同步来说, 可以认为UPS有两种角色:

1. SLAVE: 角色为备，不能提供写服务，从主那里同步日志。
2. MASTER: 角色为主，可以提供写服务，生成日志。

UPS的主备由RS管理, 角色通过RS的lease通知给UPS。 UPS在没有RS授予的lease的情况下(未注册或lease过期)，角色为备。

UPS使用一个定时器，定期检查RS授予UPS的lease和主UPS的keep alive状态，在没有lease或keep alive包，或它们已过期的情况下，发起向RS和主UPS的注册。注册不影响UPS的主流程。

定时器处理过程中设置UPS应该进入的角色为 target\_role 。 UPS启动时角色为SLAVE，然后在一个循环中检查 target\_role 必要时进行状态切换，切换完成之后修改UPS角色为 target\_role 。

UPS会有以下几种状态:

1. REPLAYING:
   1. 主备UPS在重放本地日志；
   2. 备UPS在从主UPS那里同步日志，并且落后于主UPS较多；
2. ACTIVE:
   1. 主UPS重放完本地日志；
   2. 备UPS基本上追上主UPS的日志；
3. ERROR: 表示出现错误
4. STOP: 表示正常退出

UPS的完整状态可以用: "角色:状态" 表示, 比如: "MASTER:ACTIVE" 表示当前角色为主，状态为ACTIVE。

1. MASTER:ACTIVE意味着可以提供读写服务，
2. SLAVE:ACTIVE意味着能提供读服务，但不保证与主UPS完全一致。
3. 除非DBA显示设定 allow\_stale\_read 为true，否则MASTER:REPLAYING/SLAVE:REPLAYING状态不能提供读服务。

备UPS用 log\_sync\_type 表示自己是实时备机还是非实时备机。备UPS向主UPS注册时，需要告诉主UPS自己的 log\_sync\_type 。

备UPS通过两种机制同步主UPS的日志:

1. 备向主请求某个日志cursor之后的日志，主返回日志cursor之后的一批日志。
2. 主每次产生一批新日志，主动向备push。

正常情况下，备UPS的日志点不应该落后主很多，备向主请求日志时，总是请求最近产生日志，为了优化这一步骤，主UPS引入了日志缓冲区用于保存最近主生成的一批日志。主UPS把最近生成的日志push给备UPS时，备UPS不一定能立即处理(因为备UPS可能还有一段老日志落后于主)， 所以备UPS也需要用一个日志缓冲区保存主UPS push过来的日志。实际上主备UPS的日志缓冲区需求一样，使用同一个模块实现，不妨把它称为 recent\_log\_cache 。

备UPS重放和向主UPS取日志可以并行，为此要引入一个缓冲区 fetched\_log\_buffer ，用于缓存取来的日志。

取日志服务是无状态的，所以UPS不管状态，也不管主备，只要初始化完成就可以提供取日志服务。取日志请求要指定起始日志cursor，处理取日志请求时，先尝试从 recent\_log\_cache 中读，读不到时再根据cursor尝试从文件中读，所以UPS还需要实现一个根据cursor读日志的模块, 不妨把它称为 pos\_log\_reader 。

为了描述日志同步的进度，UPS需要记录如下的日志点位置:

1. 主UPS将要提交的日志点: master\_clog\_cursor
2. 备已重放的日志点: replayed\_clog\_cursor

对主UPS而言， master\_clog\_cursor/replayed\_clog\_cursor 总是相等的，主UPS每次写盘并提交日志后，更新 master\_clog\_cursor/replayed\_clog\_cursor 。备UPS每次收到主UPS的日志后，更新 master\_clog\_cursor , 每次重放日志之后更新 replayed\_clog\_cursor 。

简单描述一下备机同步状态的变化原则:

1. 备机replay线程重放时发现, replayed\_clog\_cursor 等于 master\_clog\_cursor 时, 认为备机同步。
2. 备机replay线程重放时发现, master\_clog\_cursor > replayed\_clog\_cursor 并且要取的日志在 recent\_log\_cache 没有， 需要向主机请求时，认为备机不同步。
3. 备机replay线程重放时发现, mutator的生成时间与现在时间差超过了某个阈值, 认为备机不同步。

更详细的规则在 **备UPS追赶主UPS的流程** 一节说明。

**2.2 启动流程分解**

UPS主线程如下所示：

init\_first\_time();

while(!need\_stop()) {

switch(ups\_role) {

case MASTER: run\_as\_master(); break;

case SLAVE: run\_as\_slave(); break;

default: report\_error(); break;

}

};

所以UPS主线程逻辑分解为三部分:

1. 第一次启动时一次性的初始化工作;
2. 确定角色为主之后的处理;
3. 确定角色为备之后的处理;

**2.3 日志同步的相关线程，服务和任务**

1. 重放本地日志的任务: 系统第一次启动时, 提交一次重放本地日志的任务， 直到本地日志重放完或显式要求停止(被kill或有错误发生)之后任务执行完成。 重放过程中更新 replayed\_clog\_cursor 。 重放完本地日志后， 需要调用 log\_mgr\_.start\_log(replayed\_clog\_cursor) ，表示从这个点以后生成日志和写日志。
2. 取日志服务: 相关数据结构初始化完之后就可以启用，不管UPS是什么角色和状态。 但是要限制取日志服务对磁盘的使用率，以免影响正常读写。
3. 写服务: 只有主机处于ACTIVE状态时才能启用，由单个线程处理。 负责处理所有写请求(包括 SWITCH\_LOG/MAJOR\_FREEZE/MINOR\_FREEZE/SWITCH\_SCHEMA 等)， 串行地改memtable，生成日志，写日志，同步备机，提交事务，响应客户端，新日志进缓冲区。 更新 master\_clog\_cursor/replayed\_clog\_cursor 。
4. 读服务: 主机或备机处于ACTIVE状态时能启用。负责处理get/scan请求及各种监控状态的读取。
5. 备机注册/退出服务: 主UPS可以提供，不管自己是否处于ACTIVE状态。 主机用 slave\_mgr 管理所有备机，备机注册时告诉主机自己的地址，自己的 log\_sync\_type, 主机将备机加入 slave\_mgr (需要处理并发和重复加入的问题), 备机注册成功，主机会将自己拥有的sstable和log文件列表及最大日志cursor返回给备机。 处理备机退出消息时，主机将它从 slave\_mgr 中删除。
6. 收日志服务: 备UPS启用，收到主机push的日志后立即给主机响应， 把收到的日志放入到 recent\_log\_cache, 缓冲区满时，直接覆盖老日志。 收到日志后更新 master\_clog\_cursor 。
7. replay线程: 备UPS使用，在本地日志重放完之后，接着本地replay之后的日志点 replayed\_clog\_cursor 开始， 负责所有后续日志的写盘和重放。replay线程的日志来源有两个: recent\_log\_cache 和 fetched\_log\_buffer, 先尝试从 recent\_log\_cache 中取日志，如果取不到再尝试从 fetched\_log\_buffer 中读， 如果检查到 fetched\_log\_buffer 有足够的空间，提交一次取日志任务，取日志任务异步地填充 fetched\_log\_buffer 。 切换为主之前要停掉replay线程，备机重新注册时不需要停replay线程。重放过程中更新 replayed\_clog\_cursor 。
8. fetch日志任务：备UPS使用，由replay线程提交。根据要取的日志开始位置向主机请求日志， 并将取回的日志填充到 fetched\_log\_buffer 中。

**2.4 各种服务什么时候可用**

UPS在全局的一次性初始化工作完成之后, tbnet线程和工作线程，写线程等就已经启动，但是否处理各种请求是由UPS的状态决定的: 当 target\_role 和UPS当前角色不一致时，可以认为UPS处于切换状态。

1. 获取日志请求总是可以服务的；
2. 读请求：
   1. 如果是 ACTIVE 状态，可以服务；
   2. 如果是 REPLAYING 状态，即主机未重放完本地日志或备机未追上主， 除非DBA设置参数 allow\_stale\_read 为true，否则不能服务；
3. 写请求：
   1. 在 MASTER:ACTIVE 状态且 target\_role 和UPS角色一致， 且 lease 在未来的一段时间内有效(保证可以完成当前的一批写操作)，提供服务；
   2. 其他状态返回 OB\_NOT\_MASTER ；
4. 备机注册和退出, 确定角色为主且lease有效就可以服务。
5. 收日志请求：确定角色为备且lease有效就可以服务。
6. 不认识的请求返回错误。

上面的状态检查在分发请求和处理请求时都要进行。必须保证写请求和收日志请求不能同时处理，写请求和replay线程也不能同时处理。

1. 在主切换为备之前，需要等待正在处理的写请求完成；
2. 在备切换为主之前，需要等待正在处理的收日志请求完成，replay线程停下。

在实现上写服务和收日志服务用同一个线程处理。等待正在处理的任务完成，目前实现为等待线程停下，清空任务队列，然后重新启动线程。

**2.5 全局的一次性初始化工作**

1. 初始化 obi\_role/role\_mgr , 各种 thread\_queue , recent\_log\_cache, pos\_log\_reader, log\_mgr , slave\_mgr 等。
2. **设置UPS状态为SLAVE:REPLAYING** 。
3. 提交重放本地日志的任务

**2.6 确定角色为主之后**

1. 非阻塞地等待重放本地日志的任务结束, 若未结束，直接返回。
2. **UPS状态置为ACTIVE**
3. 如果 target\_role 不等于当前的角色:
   1. 等待正在处理的写操作完成。
   2. 调用 log\_mgr\_.flush\_log() 确保还未写盘和提交的日志都被写盘并提交。
   3. 更新当前角色为 target\_role .

**2.7 确定角色为备之后**

1. 非阻塞地等待重放本地日志的任务结束, 若未结束，直接返回。
2. 如果未启用replay线程，启用它。
3. 如果 target\_role 不等于当前角色:
   1. 等待正在执行的收日志任务处理完毕。
   2. 检查自己是否与主UPS同步，如果不同步，应该报警， 但是不能认为这是错误(比如主UPS意外宕机，每个备都未同步，但还是要选择一个主)，应该继续处理。
   3. 如果已启用replay线程, 等待replay线程将 recent\_log\_cache 中的所有日志处理完毕，然后停掉replay线程.
   4. 设置当前角色为 target\_role 。

在角色确定为备之后:

1. 收日志服务可用，备收到主的日志后，如果是实时备机，立即给主响应，如果是非实时备机，不用给主响应。 不管什么类型的备机，都要把日志放入到 recent\_log\_cache 。
2. **replay线程判断自己与主的日志落后不多时，UPS状态置为ACTIVE,判断自己与主的日志落后较多之后再次把状态修改为REPLAYING，报警**.

**3 写线程处理写请求的过程**

处理写请求使用了 group commit 优化， 开始事务后，apply一批mutator，并不立即提交，直到 log\_writer 内部缓冲区满，然后开始处理这一批日志，遇到 SWITCH\_LOG/SWITCH\_SCHEMA/MINOR\_FREEZE/MAJOR\_FREEZE 这些特殊操作，必须将已开始的一批事务处理完之后，才能开始处理这些特殊操作。

处理一批日志的流程如下:

1. 将这批日志写盘，如果失败，报警，回滚当前的一批事务，结束这次写流程。
2. 将这一批日志加入到 recent\_log\_cache ，如果失败，报警但UPS不退出。
3. 将这批日志push给备机, 需要等待备机响应，但不管备机是否超时，push操作都算成功。
4. 提交事务同时更新 master\_clog\_cursor/replayed\_clog\_cursor 。
5. 根据可靠性要求和备机对push日志的响应决定是否应答客户端。现在实现为只要主UPS写成功，就应答客户端成功。

主机给备机push日志的过程:

主机用 slave\_mgr\_ 记录了所有的备UPS的地址和状态，备UPS的状态包括备机的 log\_sync\_type 。主机push日志时，先遍历 slave\_mgr\_ 得到同集群的备UPS列表，优先将日志同步给同集群的备，然后再同步给剩余的备UPS。由于备UPS不可能太多，所以push日志先做成串行的，push给每个UPS日志时根据备机的 log\_sync\_type 决定是否需要等待响应，如果需要等备UPS响应，但等待超时，则将备UPS从 slave\_mgr\_ 中删除。

**4 备UPS追赶主UPS的流程**

replay线程开始追赶主UPS之前，本地的日志已经重放完毕，收日志服务已经被启用了。不妨把重放之后的日志点记为 replayed\_cursor ， replay线程负责 replayed\_cursor 后续日志的写盘和重放。并且 master\_clog\_cursor 在注册时由主机返回给了备机。

replay线程每次调用 retrieve\_replay\_log(start\_cursor,...) 函数获取后续需要处理的日志，其中 start\_cursor 表示后续日志的起始点。备UPS每次收到主UPS push过来的日志后立即响应主机并把日志放入 recent\_log\_cache 。 retrieve\_replay\_log 每次先尝试从 recent\_log\_cache 中取日志，若取不到，再尝试从 fetched\_log\_buffer 中取。每次从 fetched\_log\_buffer 中取日志之前，会检查要取的日志是否能与 fetched\_log\_buffer 内部记录的点连接上，如果接不上，需要重新初始化 fetched\_log\_buffer ；不管这次取日志是否成功，都会检查 fetched\_log\_buffer 是否已空出了足够的空间，如果有，尝试提交一个异步取日志任务接着 fetched\_log\_buffer 最后的日志点取日志填充到 fetched\_log\_buffer 。

**4.1 replay线程的内部状态和处理逻辑**

重放过程中，replay线程用 replayed\_clog\_cursor 记录重放日志点，重放过程中需要调用 retrieve\_replay\_log() 取日志并修改UPS同步状态。

replay线程是一个循环：

1. 使用 retrieve\_replay\_log(replayed\_clog\_cursor,...) 取出后续日志，若返回 OB\_READ\_NOTHING 表明目前没有更多的日志可取， sleep一段时间之后开始下一轮循环。
2. 取日志成功后写日志，重放，更新 replayed\_clog\_cursor , 开始下一轮循环； 备机在重放日志的过程中，如果发现 mutator 生成的时间与现在的时间差超过了一个阈值，说明备机的日志落后于主机较多， 若这时备机状态为ACTIVE，把它修改为REPLAYING，并报警。

**4.2 retrieve\_replay\_log(start\_cursor,...) 的实现**

retrieve\_replay\_log(start\_cursor,...) 取 start\_cursor 之后的日志，若暂时取不到，返回 OB\_READ\_NOTHING , 取日志时需要引用 recent\_log\_cache 和 fetched\_log\_buffer ，取日志时也会修改备UPS同步状态.

1. 检查 master\_clog\_cursor 是否有效并等于 start\_cursor ，如果是，表明目前没有更多的日志需要处理， 返回一个特殊的错误码 OB\_READ\_NOTHING 。
2. 尝试从 recent\_log\_cache 中读取 start\_cursor 之后的日志：
   1. 如果读取成功，并且读之后日志点等于 master\_clog\_cursor ，说明备机的日志追上了主机， 若这时备机状态为REPLAYING, 把它修改为ACTIVE；
   2. 如果读取失败，表明备机日志落后于主较多，若此时备机状态为ACTIVE, 把状态修改为REPLAYING，并报警, 尝试下一步;
3. 尝试从 fetched\_log\_buffer 中读取 start\_cursor 之后的日志，并返回结果。

**4.3 收日志线程的处理流程**

收日志后需要把日志放入 recent\_log\_cache .

1. 接受主机push的日志后，如果是实时备机，立即响应主机，如果是非实时备机，不用响应主机。
2. 收到日志后解析一遍，得到日志的开始和结束位置 start\_cursor 和 end\_cursor ，
3. 检查 start\_cursor 是否等于 master\_clog\_cursor ，如果接不上，报警。
4. 将日志放入 recent\_log\_cache, recent\_log\_cahce 内部也记录了一个日志点，主切换为备或备机重新上线时， push的日志可能接不上 recent\_log\_cache 保存的日志, 这时需要重置 recent\_log\_cache 。 ( recent\_log\_cache 被实现为重置时也可安全地读取).
5. 更新 master\_clog\_cursor 。

**4.4 recent\_log\_cache 的实现**

后面有单独的一节描述。

**4.5 fetched\_log\_buffer 的管理**

fetched\_log\_buffer 需要支持单读者单写者，读者根据日志ID读取一批日志，写者向缓冲区中追加日志，当读取日志不连续时，需要重置缓冲区。日志保存在固定大小的缓冲区内， 内部记录了读写的位置。由于每次取日志任务都是由replay线程主动发起的，所以 fetched\_log\_buffer 不可能覆盖未处理的日志。任意时刻，只能有一个取日志任务执行，为此， replay线程每次发起一个取日志任务之前，要设置一个变量 started 为true，取日志任务结束后，设置它为false，这样每次replay线程提交取日志任务之前要检查 started 是否为true, 若为true，不必提交新的任务。取日志任务执行时等待主UPS的响应超时时间要设置合理，以保证不管取日志是否成功，任务在合理的时间一定会结束。

class ObFetchedLogBuffer {

public:

// retrieve\_replay\_log() 需要调用此接口， 在取日志时检查到缓冲区有足够的剩余空间时，提交异步取日志任务填充到缓冲区。

// read\_pos\_在每次读之后会更新。

// 要读取的日志起始点若与缓冲区内记录的read\_cursor不相等，则缓冲区需要重置。

int get\_log(const ObLogCursor& start\_cursor, ObLogCursor& end\_cursor,

char\* buf, int64\_t len, int64\_t& read\_count);

// 取日志任务执行时，成功取回日志后把日志放入缓冲区调用此接口。

int push\_log(const ObLogCursor& start\_cursor, const ObLogCursor& end\_cursor, char\* buf, int64\_t len);

protected:

int reset();

private:

char\* buf\_;

int64\_t buf\_len\_;

// write\_pos\_和read\_pos\_在遇到缓冲区末尾时不会置零，而是不断累计计数。

int64\_t write\_pos\_;

int64\_t read\_pos\_;

ObLogCursor read\_cursor\_;

ObLogCursor write\_cursor\_;

};

fetched\_log\_buffer 中保存的日志是由 read\_cursor 和 write\_cursor 组成的左闭右开的区间。

reset() 实现:

1. write\_cursor\_/read\_cursor\_ 置为无效值
2. write\_pos\_ 和 read\_pos\_ 都置为0

get\_log(start\_cursor, end\_cursor, buf, len, read\_count) 执行流程如下:

1. 检查 start\_cursor 是否与 read\_cursor\_ 相等，若不相等，等待当前的取日志任务结束，重置缓冲区，继续下一步。
2. 检查 read\_pos\_ 和 write\_pos\_ 是否相等，若相等，表明缓冲区中无日志可用。这次读日志失败，继续下一步。 若有日志可读，将日志填充到 buf 内，设置好 read\_count ，并更新 read\_pos\_/read\_cursor\_, 继续下一步.
3. 为了预防代码的bug，读取日志后需要检查 write\_pos\_ 是否覆盖了 read\_pos\_ 。
4. 如果 read\_pos\_ + buf\_len\_ - write\_pos\_ 够大，比如超过2M， 检查当前是否有另一个取日志任务在执行，如果没有，提交一个异步取日志任务。

push\_log(start\_cursor, end\_cursor, buf, len) 执行流程如下:

1. 如果 write\_cursor\_ 有效并且和 start\_cursor 不等，报警，结束这次调用。
2. 计算缓冲区剩余的空间 remain\_size 等于 read\_pos\_ + buf\_len\_ - write\_pos\_
3. 若 remain\_size 大于 len ，报警，把 buf 中的日志截断为不超过 remain\_size 的长度。
4. 从 write\_pos\_ 开始，将 buf 中的日志拷贝到缓冲区，拷贝完之后更新 write\_cursor\_/write\_pos\_

**5 commit log获取服务**

原则上每台UPS在任意时刻都可以提供取日志服务，在实现上，日志获取服务是由读线程处理的，所以只有处于ACTIVE状态时，才能提供日志获取服务。

**5.1 取日志协议**

1. 取日志请求的参数包括: 起始日志ID，允许返回日志的最大尺寸。
2. 取日志的响应包括: ERROR Code, 日志长度, 日志内容。

**5.2 取日志请求处理流程**

1. 尝试从 recent\_log\_cache 根据指定日志ID取一批日志: 如果能取到日志，直接返回结果; 如果不能取到日志，尝试步骤2;
2. 如果步骤1失败，尝试用 pos\_log\_reader 从日志文件中根据指定的日志ID读取日志: 如果能取到日志，直接返回结果; 如果不能取到日志，返回特殊的错误码 OB\_READ\_NOTHING ;

**6 ObPosLogReader**

ObPosLogReader 根据日志ID读取日志文件，返回给定ID(包括该ID)之后的一批日志。要完成读取功能，需要两个功能:

1. 根据日志ID快速定位日志文件ID和文件内偏移，这部分功能由 ObLogLocator 实现。
2. 根据日志文件ID和文件偏移快速读取日志的类，这部分功能由 ObLocationLogReader 完成。

日志的位置信息定义如下:

struct ObLogLocation: AtomicType<ObLogLocation> {

int64\_t log\_id\_;

int64\_t file\_id\_;

int64\_t offset\_;

};

其中 AtomicType 是一个用乐观锁实现原子性读取和修改的类。

template<typename T>

struct AtomicType {

int64\_t version\_; // version\_小于等于0时无效

int copy(T& that);

int set(T& that);

};

1. 在修改T之前，先把当前的 verson\_ 保存为 old\_version\_, 再把 version\_ 置为无效值，修改完成之后将 version\_ 置为 old\_version+1.
2. 在读取T之前，先把当前的 version\_ 保存为 old\_version, 如果 old\_version 无效，读取失败； 否则，拷贝T到另一个变量，拷贝完成之后，再次检查 version 号是否等于 old\_version, 如果不等，拷贝可能无效，如果相等，拷贝一定是有效。

ObLogLocator 和 ObLocationLogReader 接口定义如下:

class IObLogLocator {

public:

int get\_location(const int64\_t log\_id, ObLogLocation& location);

int enter\_location(const ObLogLocation& location);

};

class IObPosLogReader {

public:

int read(const ObLogLocation& location, ObLogLocation& end\_location, char\* buf, const int64\_t len, int64\_t& read\_count);

};

ObPosLogReader 实现如下:

class ObPosLogReader {

public:

int get\_log(const int64\_t start\_log\_id, int64\_t& end\_log\_id,

char\* buf, const int64\_t len, int64\_t& read\_count);

private:

IObLogLocator\* log\_locator\_\_;

IObLocationLogReader\* location\_log\_reader\_;

};

get\_log(start\_log\_id, end\_log\_id, buf, len, read\_count)

1. 使用 log\_locator\_.get\_location(log\_id, location) 得到文件ID和偏移
2. 使用 location\_log\_reader\_.read(location, end\_location, buf, len, read\_count) 读取文件内容到 buf
3. 调用 log\_locator\_.enter\_location(end\_location) 缓存最后的日志位置。

ObLocationLogReader 目前实现为每次重新打开指定的文件，定位到给定的位置，读取日志。

**7 ObLogLocator 的实现**

为了根据日志ID快速定位日志文件ID和文件内偏移，使用两个类：

1. ObDefaultLogLocator 读取文件定位日志
2. ObCachedLogLocator 缓存一部分日志的位置

这两个类都实现了 IObLogLocator 接口

class ObLogLocator : public IObLogLocator {

public:

// 先尝试调用 cached\_locator.get\_location()

// 失败后再调用 backup\_locator.get\_location()

int get\_location(int64\_t log\_id, ObLogLocation& location);

// 直接调用 cached\_locator\_.enter\_location(location)

int enter\_location(const ObLogLocation& location);

private:

ObLogLocator\* cached\_locator\_;

ObDefaultLogLocator\* backup\_locator\_;

};

**7.1 位置缓存**

class ObCachedLogLocator : public IObLogLocator {

public:

int get\_location(int64\_t log\_id, ObLogLocation& location);

int enter\_location(const ObLogLocation& location);

private:

int64\_t n\_location\_indexes\_;

ObLogLocation\* location\_indexes\_;

};

get\_location(log\_id, location)

1. 原子性拷贝 location\_indexes\_[log\_id % n\_location\_indexes\_] 到 location ，若拷贝失败，返回 OB\_ENTRY\_NOT\_EXIST ；
2. 检查 location.log\_id\_ 是否等于 log\_id ，若不等，返回 OB\_ENTRY\_NOT\_EXIST ; 若相等，返回成功。

enter\_location(location)

1. 检查 location\_indexes\_[location.log\_id\_ % n\_location\_indexes\_].log\_id\_ 是否大于 location.log\_id\_ ，若前者更大，直接返回。
2. 原子性更新 location\_indexes\_[location.log\_id\_ % n\_location\_indexes\_] 为 location

**7.2 实际定位**

1. 根据日志ID得到文件ID，由 ObLogFileIdLocator 完成
2. 在单个日志文件内得到给定ID的日志的偏移, 由 ObLogFileOffsetLocator 完成。

class ObDefaultLogLocator {

public:

int get\_offset();

private:

ObLogFileIdLocator file\_id\_locator\_;

ObLogFileOffsetLocator file\_offset\_locator\_;

};

* 由日志ID查询文件ID
* class ObLogFileIdLocator {
* struct FirstId : public AtomicType<FirstId> {
* int64\_t file\_id\_;
* int64\_t log\_id\_; // 小于等于0表示无效
* };
* public:
* // max\_log\_file\_id 表示当前最大的日志文件编号，可以从log\_mgr获得。
* int get\_file\_id(const int64\_t max\_log\_file\_id, int64\_t log\_id, int64\_t& file\_id);
* int get\_start\_log\_id(int64\_t file\_id, int64\_t& start\_log\_id);
* private:
* int64\_t n\_indexes\_;
* FirstId\* indexes\_;
* };

get\_file\_id(max\_log\_file\_id, log\_id, file\_id)

* 1. 设置 try\_file\_id 的值为 max\_log\_file\_id
  2. 调用 get\_start\_log\_id(try\_file\_id, start\_log\_id) 尝试获得 try\_file\_id 对应的日志文件的第一条日志的ID， 若这条日志ID小于 log\_id , 设置 file\_id 为 try\_file\_id ，返回成功; 否则，将 try\_file\_id 减1，继续尝试，直到 try\_file\_id 对应的文件不存在，返回失败。

get\_start\_log\_id(file\_id, start\_log\_id)

* 1. 先尝试从 indexes\_[file\_id % n\_indexes\_] 中取得第一条日志位置， 若对应的 file\_id\_ 匹配，设置 start\_log\_id 为 log\_id\_ 直接返回; 若对应的 file\_id\_ 不匹配，进行下一步；
  2. 尝试打开编号为 file\_id 的日志文件，取出第一条日志的编号， 若成功，设置 start\_log\_id 为 log\_id\_ , 如果 indexes\_[file\_id % n\_indexes\_].file\_id\_ 小于 file\_id , 更新索引，返回。
* 文件内部定位偏移
* class ObLogFileOffsetLocator {
* public:
* int get\_offset(const int64\_t log\_id, const int64\_t file\_id, int64\_t& offset);
* };

get\_offset(log\_id, file\_id, offset)

* 1. 打开编号为 file\_id 的日志文件，
  2. 从当前位置读取一条日志，假设当前日志编号为 cur\_id :
     1. 如果 cur\_id 大于 log\_id, 返回错误；
     2. 如果 cur\_id 等于 log\_id-1, 设置 offset 为当前位置, 返回；
     3. 如果 cur\_id 等于 log\_id, 要读的日志是这个文件的第一条日志，设置 offset 为0, 返回；
     4. 如果 cur\_id 小于 log\_id-1, 重复第二步

在遇到错误或读到文件末尾时退出第二步循环。

**8 recent\_log\_cache**

**8.1 模块描述**

recent\_log\_cache 用于缓存主UPS最近生成的一批日志或备UPS最近收到主UPS push过来的一批日志。 recent\_log\_cache 要支持三种操作:

1. 向 recent\_log\_cache 中追加一批日志，追加日志的操作必须被串行化，追加的日志必须与已有的日志是连续的, 在追加日志时如果发现与缓冲区中的日志不连续，需要重置 recent\_log\_cache 。 为此 recent\_log\_cache 要维护已保存的最大日志ID, 追加日志时除了日志内容还需要给出日志的开始和结束日志ID。 追加写可以无条件的覆盖缓冲区的老日志。
2. 从 recent\_log\_cache 中取出一批日志，读取日志时需要给出日志的开始日志ID和保存日志的缓冲区的最大长度， 读取成功后返回读取的日志总字节数和结束的日志点位置; 或者判断要读取的日志不在缓冲区内，返回 OB\_READ\_NOTHING . 单个读者读取 recent\_log\_cache 时总是连续读取的，所以可以由读者维护读取位置。 读写是并发的，读者在读取过程中缓冲区中日志可能变得无效，所以读取之后还要再检查一遍读取的日志是否已被覆盖。
3. 重置 recent\_log\_cache , 在追加日志时可能被调用。

**8.2 关键数据结构和算法**

在实现上， recent\_log\_cache 使用固定大小的缓冲区，日志在缓冲区中连续存放。每次追加日志时写到了缓冲区末尾，则折回到缓冲区开始位置覆盖以前写入的老日志。

用 start\_pos\_/end\_pos\_ 记录有效日志的起始和结束位置。 start\_pos\_ 和 end\_pos\_ 在遇到缓冲区末尾，折回到缓冲区开始和重置缓冲区时不清零而是不断累积计数。假定缓冲区的大小为 log\_buffer\_len\_ , 则可以计算当前写日志的位置, end\_pos\_ % log\_buffer\_len\_ 。

class ObRecentLogCache {

struct Pointer : public AtomicType<Pointer> {

int64\_t buf\_version\_; // 每次重置recent\_log\_cache, buf\_verson\_加1

int64\_t pos\_;

int64\_t log\_id\_;

};

public:

int init(int64\_t buf\_len, char\* log\_buf, Pointer\* indexes, int64\_t n\_index);

int push\_log(const int64\_t start\_id, const int64\_t end\_id, const char\* buf, const int64\_t len);

int get\_log(const int64\_t start\_id, int64\_t& end\_id, char\* buf, const int64\_t len, int64\_t& read\_count);

int reset();

protected:

int get\_pointer(const int64\_t log\_id, Pointer& pointer);

// 包含start\_pointer不包含end\_pointer

int check\_range\_validity(const Pointer& start\_pointer, const Pointer& end\_pointer);

private:

volatile int64\_t version\_; //充当版本号的作用，每次重置缓冲区加1

volatile int64\_t start\_pos\_;

volatile int64\_t end\_pos\_;

int64\_t start\_log\_id\_;

int64\_t end\_log\_id\_;

int64\_t log\_buffer\_len\_;

char\* log\_buffer\_;

Pointer\* indexes\_;

int64\_t n\_indexes\_;

};

* 检查pointer是否有效
* pointer.buf\_version\_ == version\_ && pointer.pos\_ <= end\_pos\_ && pointer.pos\_ >= start\_pos\_
* 检查range是否有效  
  check\_range\_validity(start\_pointer, end\_pointer): 等价于 start\_pointer/end\_pointer 同时有效。
* 读取索引  
  get\_index(log\_id, pointer): 原子性读取 indexes\_[log\_id % n\_indexes\_] 到 pointer 即可。
* 读日志  
  get\_log(start\_id, end\_id, buf, len, read\_count)
  1. 读取 start\_id 对应的索引到变量 start\_pointer
  2. 检查 start\_pointer 是否有效，若无效，返回 OB\_READ\_NOTHING
  3. 从 start\_pointer 开始拷贝多条日志对应的内容到 buf , 拷贝内容不能超过 len , 也不能越过 end\_pos\_ ，记拷贝的最后位置为 end\_pointer\_;
  4. 检查 start\_pointer 和 end\_pointer 组成的range是否依然有效，若无效，返回 OB\_READ\_NOTHING , 否则，拷贝成功，设置好 end\_id/read\_count 后返回。
* 追加日志  
  push\_log(start\_id, end\_id, buf, len)
  1. 如果 len 大于 log\_buffer\_len\_ , 返回错误；
  2. 如果 end\_log\_id\_ 有效且不等于 start\_id , 说明日志不连续，重置缓冲区；
  3. 设置 start\_pos\_ 为 max(start\_pos\_, end\_pos\_+len-log\_buffer\_len\_) ；
  4. 从 end\_pos 指示的位置拷贝日志；
  5. 设置 end\_pos 的值为 end\_pos+len ；
  6. 更新索引；
  7. 更新 start\_log\_id\_/end\_log\_id\_
* 重置缓冲区
  1. 将 max\_log\_id\_/min\_log\_id\_ 置为无效值；
  2. 将 start\_pos\_ 置为 end\_pos\_ 的值；
  3. 将 version\_ 加1

**9 如何获得本地最大的日志编号**

RS选主时，需要询问每个UPS本地已写盘或已收到的最大日志编号，选择编号最大的一个为主。获取UPS本地最大的日志编号的方法如下:

1. 如果本地日志未重放完成，返回本地日志文件保存日志的最大编号
2. 如果本地日志已重放完成
   1. 若 replayed\_clog\_cursor 小于 recent\_log\_cache.start\_log\_cursor\_ ， 表明缓冲区内容无效，返回 replayed\_clog\_cursor ；
   2. 若 replayed\_clog\_cursor 大于 recent\_log\_cache.start\_log\_cursor\_ ， 表明缓冲区内容有效，返回 recent\_log\_cache.end\_log\_cursor\_ 。

**10 兼容0.21版UPS方案**

与0.21版UPS兼容性包括两个层次：

1. 日志级别的兼容性: 这是一定需要保证的，否则无法回滚。 因为新的日志同步方案并不需要添加新的日志种类,日志级别的兼容性自然满足。
2. 协议级别的兼容性:
   1. 新增了备UPS向主UPS请求获取日志的协议，这是无法兼容的。
   2. 备UPS注册，需要主机返回最后的日志cursor，老版主机不返回cursor, 新版备机不能认为这是错误;
   3. 主UPS把日志push给备, 可以考虑带上日志的开始和接收位置，为了兼容暂时不带上这些信息。
   4. 备UPS注册时主UPS不需要切分日志, 为了兼容可以暂时保留切分日志特性。

升级和回滚方案:

1. lsync: 只要保证了日志级别的兼容性，原则上都可以通过lsync实现同步和主备切换，完成升级和回滚。 是否利用lsync完成同步是在配置文件中指定的，只需要保留lsync同步的代码不变即可。
2. 新版的备接0.21版的主，以下问题需要注意:
   1. 备需要向0.21版的主发起获取日志请求，但0.21版的主无法响应。 可以在0.21版的主所在的机器上启动一个程序提供日志获取服务(类似于 lsync, 但协议不一样, 不妨称这个程序为 lsync2)， 备机向 lsync2 发起取日志请求， lsync2 的地址可以写在备机的配置文件中。
3. 0.21版的备接新版的主，完全兼容。