并发fill log和commit

并发fill log和commit改造的目的是为了消除单提交线程的瓶颈，改造完成之后可以实现多线程解行锁，串链表，生成日志, 并发拷贝到buffer中，异步写盘和同步备机，最后由多线程响应客户端并回收 `session_ctx` .

首先明确如下的限制:

1. 并发解行锁，串链表，生成日志之前必须决定 `log_id` , `trans_id` 。其中 `trans_id` 也就是递增的时间戳.
2. 要并发fill log, 必须要提前给日志分配一个缓冲区位置。
3. 必须提前给 `NOP` 和 `SWITCH_LOG` 预留 `log_id` 和缓冲区位置, 并且每次写盘的日志不能超过2M。
4. 异步写日志之前，要决定写日志的文件编号和文件偏移。

总结一下: 并发的关键是用原子操作决定这些id和偏移. 但上面提到的所有ID和偏移都至少是64bit，无法用一个原子操作决定。

为了减少抢id和偏移的冲突，我们把待提交的事务分成很多group:

1. 保证一个group内 `log_id` 和 `trans_id` 都连续，所以只要这个group的第一个事务的 `log_id` 和 `trans_id` 决定了，其余事务的 `log_id` 和 `trans_id` 可以根据自己在group内部的编号计算出来
2. 每个group对应一个2M的buffer, 这个group的日志都写在这个buffer内(保证一个group的所有日志不会超过2M)。每个group有一个递增且连续的 `group_id`, 根据 `group_id` 可以定位到自己对应的2M `buffer` 的起始位置，，这个group内的所有事务都可以根据 `group_id` 和自己在group内的偏移计算出自己 `fill log` 的位置。
3. group内的最后一条日志是给 `NOP` 或 `SWITCH_LOG` 预留的

用如下的结构体记录 `group_id` 和group内部的id和偏移:

```
struct CLogPos
{
   int64_t group_id_;
   int32_t rel_id_; // 相对id
   int32_t rel_offset_; // 相对偏移
   int append(GroupPos& pos, int32_t len, int64_t limit);
};
```

其中 `append(pos, len, limit)` 函数的含义是:

1. 如果当前group追加长度为len的日志后还没有超过limit, 那么追加成功，返回新追加日志的pos.
2. 如果当前group追加长度为len的日志后超过了limit, 那么切换到下一个group，这种情况下，实际为两条日志预留了空间，一是当前group最后一条 `NOP` 或 `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, 下一个group的第一条日志的位置可以根据pos推算出来。
3. 特别的可以传递一个无效的len，表示不想追加"有用的" 日志，只是想冻结当前group，给当前group追加一条 `NOP` 或 `SWITCH_LOG` 日志，并切换到下一个group。
4. 当切换group的时候， `append()` 返回一个特殊的错误码: `OB_BLOCK_SWITCHED`
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 `SWITCH_LOG` 。

因为 `CLogPos` 刚好128bit, 所以可以使用CAS保证原子性。

具体来说有一个全局的 `next_pos_`, 初始化时 `groupd_id_`,`rel_id_`,`rel_offset_` 都为0, 多个线程根据自己的 `len` 乐观的计算出下一个 `new_pos`, 然后用128bit CAS操作抢占更新全局 `next_pos_` 的机会, 如果失败，需要重试。

为group定义一个结构体，记录这个group的 `ref_cnt`, `group_id`, `start_log_id_`, `start_timestamp_`, `start_log_cursor_` 等信息 全局有一个Group结构体数组，Group数组长度至少为2，一旦为一条日志抢到了一个 `cur_pos` ，那么就可以根据 `cur_pos.group_id_ % GROUP_ARRAY_SIZE` 定位到自己所属的 `group` 结构体。

Group结构体定义如下:

```
struct Group
{
  enum { READY=1 };
  int64_t ref_cnt_; // 初始化为0，一个group有n条日志的话，fill前n-1条日志的线程处理完之后，都把ref_cnt_加1，处理最后一条日志
  // 处理一个group的最后一条日志时设置len_和count_
  int64_t len_; // 日志长度
  int64_t count_; // 日志条数
  // ts_seq_ == group_id时，当前group是可用的，ts_seq == group_id + READY时，当前group的start_log_id_和start_trans_i
  // 最后把ref_cnt_减为0的线程需要负责把ts_seq_置为group_id + GROUP_ARRAY_SIZE以重用当前group结构体
  int64_t ts_seq_ CACHE_ALIGNED;
  int64_t start_timestamp_;
  int64_t start_log_id_;
  // 处理一个group的最后一条日志时设置下一个group的last_timestamp_, 设置好之后把下一个group的last_ts_seq_置为next_group_i
  int64_t last_ts_seq_ CACHE_ALIGNED;
  int64_t last_timestamp_; // 上一条日志使用的时间戳
  // 处理一个group的最后一条日志时设置下一个group的start_log_cursor_, 设置好之后把下一个group的log_cursor_seq_置为next_gr
  int64_t log_cursor_seq_ CACHE_ALIGNED;
```

```
  ObLogCursor start_log_cursor_; // 写日志的位置，包括file_id_, file_offset_
};
```

0.1 group结构体的重用

重用group是根据 `ref_cnt_` 和 `ts_seq_` 决定的。group数组中的第i项的 `ts_seq_` 初始化为 `i`，标志这些group结构体直接可用。 `ref_cnt_` 初始化为 `0` ， 当一个group有 `n` 条日志的时候，前 `n-1` 条日志处理完之后都把 `ref_cnt_` 加1， 最后一条日志处理完之后把 `ref_cnt_` 减 `n-1`,最后一个把 `ref_cnt_` 变为0的那个线程负责重用当前group结构体，方法是把当前group的 `ts_seq_` 置为 `cur_group_id_ + GROUP_ARRAY_SIZE` .

0.2 `timestamp` 和 `log_id` 的维护

group的 `ts_seq_` 变为 `cur_group_id_ + READY` 时，当前group的 `start_timestamp_` 和 `start_log_id_` 都可用了，这样group内的第 `i` 条日志对应的 `timestamp` 为 `start_timestamp_ + i`,`log_id_` 为 `start_log_id_ + i`.

初始化和备切成主的时候，更新当前group的 `start_log_id_` 和 `start_timestamp_` ，当前group是指由全局的 `next_pos_.group_id_` 指定的group。 更新完 `start_log_id_`,`start_timestamp_` 之后设置 `ts_seq_` 为 `next_pos_.group_id_ + READY` 。

正常情况下，一个group的 `start_log_id_` 是由上一个group的最后一条日志的处理线程设置的，但 `start_timstamp_` 却只能在处理这个group的第一条日志时决定，上一个group的最后一条日志处理完之后会 设置下一个group的 `last_timestamp_` 。

具体来说, 比如有A,B两个连续的group，两个group的group id分别是 `cur_group_id`,`next_group_id`：

1. 处理group A的最后一条日志时，需要等group B可用，也就是要等group B的 `ts_seq_` 变成 `next_group_id`，然后设置group B的 `start_log_id_` 和 `last_timestamp_` 设置完成之后修改group B的 `last_ts_seq_` 为 `next_group_id + READY`.
2. 处理group B的第一条日志时，需要自己的 `last_ts_seq_` 变成 `next_group_id + READY`,然后更新自己的 `start_timestamp_` 为 `last_timestamp_ + 1` 和当前时间戳中的较大者。最后设置自己的 `ts_seq_` 为 `cur_group_id + READY`

0.3 `log_cursor` 的维护

初始化或备切成主时，更新当前group的 `start_log_cursor_`，当前group是指由全局的 `next_pos_.group_id_` 指定的group。更新完 `start_log_cursor_` 之后设置 `log_cursor_seq_` 为 `next_pos_.group_id_ + READY`

正常情况下, 处理当前group最后一条日志的线程会设置下一个group的 `start_log_cursor_` ，设置完之后更新 `log_cursor_seq_` 指示 `start_log_cursor_` 可用.

0.4 并发性分析

上面的描述中有几个等待：

1. 等group结构体变得可用，这里是否需要等待，取决于重用是否及时。只要把结构体数组设得够大，这个部分是不会有冲突的。
2. 等 `start_log_cursor_` 变得有效，设置 `start_log_cursor_` 是上一个group的最后一条日志的处理线程做的，等待是在一个group的所有日志都处理完之后才开始的，所有真正需要等待的可能也不大。
3. 等 `start_log_id_` 和 `start_timestamp_` ，这一步是最主要的等待。但是应该也不至于成为瓶颈，不过这里确实需要一个优化，因为通常都是一个线程发现当前group放不下后切换到下一个group，这个线程一次抢占了两条日志的位置，所以实际上系统繁忙时，上一个group的最后一条日志和下一个group的第一条日志的是被同一个线程先后处理的。这种情况下，这个线程可以直接设置好下一个group的 `start_timestamp_` ，而不必先设 `last_timestamp_` 然后等真正处理下一个group的第一条日志的时候才设置 `start_timestamp_` .

0.5 异步写盘和同步备机

把 `ref_cnt_` 变为0的线程要负责提交异步写盘和同步备机的任务给 `log_writer`,在写盘之前要等待 `start_log_cursor_` 变得有效。如果某个group及其之前的所有group的日志都刷盘成功，并且同步备机成功或超时， `log_writer_` 就会调用回调函数，在回调函数里要重用group结构体。

0.6 关于 `commited_trans_id_` 和 `published_trans_id_`

1. 把 `ref_cnt_` 变成0的线程还要负责更新全局的 `commited_trans_id_` 为 `start_timestamp_ + count_ - 1` ，但要注意因为是多线程更新 `commited_trans_id_` 所以这里使用CAS操作进行。这个CAS只是为了保险，实际也不

会有什么冲突。

1. `published_trans_id_` 是在 `log_writer_` 的回调函数重用group结构体的时候更新的，同样这里要用CAS操作，但实际冲突很小。

0.7 给客户端回包和 `session_ctx` 的回收

工作线程在分配 `group_id` ， 并发fill log完成之后，把 `group_id` 记录在 `session_ctx` 的上下文中，然后把 `session_ctx` 加到自己线程私有的一个队列中去。等到对应group及之前group的日志都持久化成功之后就可以应答客户端并回收 `session_ctx` 了，所以在 `log_writer` 的回调函数里还需要唤醒工作线程。

1 接口定义

首先一个待提交的任务需要提供以下三个接口:

```
  class ICommitTask
```

```
{
  public:
    virtual int32_t get_log_len();
    virtual int fill_log(uint64_t log_id, uint64_t timestamp, char* buf, int64_t len) = 0;
    virtual int end_session() = 0;
};
```

其次需要有更新全局 `commited_trans_id_,published_trans_id_` 的方法

```
class ISessionMgr
{
  public:
    int update_commited_trans_id(int64_t trans_id);
    int update_published_trans_id(int64_t trans_id);
};
```

最后，我们定义一个类 `ObCommitQueue`：

1. 所有待commit的任务都用 `submit()` 接口提交给它
2. 当最后日志持久化成功的时候会调用 `flush_ok_callback` 回调，在这个回调里要唤醒工作线程。
3. 工作线程在每次循环的时候都要调用一次 `handle_unfinished_task()` 函数, 这个函数负责给客户端回包并回收 `session_ctx.`

```
class ObCommitQueue
{
  public:
    int init(IAckCallback* flush_ok_callback, ISessionMgr* session_mgr);
    int submit(ICommitTask* task);
    int handle_unfinished_task();
};
```

如果系统比较空闲，会有一个线程调用 `submit(NULL)` 函数，让当前的group立即冻结，切换到下一个group，使得当前group的事务可以及时提交。

1. 判断系统已经空闲的标准是： 一个线程在fill完log之后发现全局的 `next_pos_` 还没有变化。
2. 调用 `submit(NULL)` 的条件是：最后一个 `group` 不为空。