MySQL · 引擎特性 · InnoDB redo log漫游

前言

InnoDB 有两块非常重要的日志，一个是undo log，另外一个是redo log，前者用来保证事务的原子性以及InnoDB的MVCC，后者用来保证事务的持久性。

和大多数关系型数据库一样，InnoDB记录了对数据文件的物理更改，并保证总是日志先行，也就是所谓的WAL，即在持久化数据文件前，保证之前的redo日志已经写到磁盘。

LSN(log sequence number) 用于记录日志序号，它是一个不断递增的 unsigned long long 类型整数。在 InnoDB 的日志系统中，LSN 无处不在，它既用于表示修改脏页时的日志序号，也用于记录checkpoint，通过LSN，可以具体的定位到其在redo log文件中的位置。

为了管理脏页，在 Buffer Pool 的每个instance上都维持了一个flush list，flush list 上的 page 按照修改这些 page 的LSN号进行排序。因此定期做redo checkpoint点时，选择的 LSN 总是所有 bp instance 的 flush list 上最老的那个page（拥有最小的LSN）。由于采用WAL的策略，每次事务提交时需要持久化 redo log 才能保证事务不丢。而延迟刷脏页则起到了合并多次修改的效果，避免频繁写数据文件造成的性能问题。

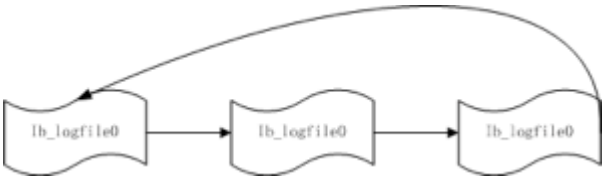
由于 InnoDB 日志组的特性已经被废弃（redo日志写多份），归档日志(InnoDB archive log)特性也在5.7被彻底移除，本文在描述相关逻辑时会忽略这些逻辑。另外限于篇幅，InnoDB崩溃恢复的逻辑将在下期讲述，本文重点阐述redo log 产生的生命周期以及MySQL 5.7的一些改进点。

本文的分析基于最新的MySQL 5.7.7-RC版本。

InnoDB 日志文件

InnoDB的redo log可以通过参数`innodb_log_files_in_group`配置成多个文件，另外一个参数`innodb_log_file_size`表示每个文件的大小。因此总的redo log大小为`innodb_log_files_in_group * innodb_log_file_size`。

Redo log文件以`ib_logfile[number]`命名，日志目录可以通过参数`innodb_log_group_home_dir`控制。Redo log 以顺序的方式写入文件文件，写满时则回溯到第一个文件，进行覆盖写。（但在做redo checkpoint时，也会更新第一个日志文件的头部checkpoint标记，所以严格来讲也不算顺序写）。



在InnoDB内部，逻辑上`ib_logfile`被当成了一个文件，对应同一个space id。由于是使用512字节block对齐写入文件，可以很方便的根据全局维护的LSN号计算出要写入到哪一个文件以及对应的偏移量。

Redo log文件是循环写入的，在覆盖写之前，总是要保证对应的脏页已经刷到了磁盘。在非常大的负载下，Redo log可能产生的速度非常快，导致频繁的刷脏操作，进而导致性能下降，通常在未做checkpoint的日志超过文件总大小的76%之后，InnoDB 认为这可能是个不安全的点，会强制的preflush脏页，导致大量用户线程stall住。如果可预期会有这样的场景，我们建议调大redo log文件的大小。可以做一次干净的shutdown，然后修改Redo log配置，重启实例。

除了redo log文件外，InnoDB还有其他的日志文件，例如为了保证truncate操作而产生的中间日志文件，包括 truncate innodb 表以及truncate undo log tablespace，都会产生一个中间文件，来标识这些操作是成功还是失败，如果truncate没有完成，则需要在 crash recovery 时进行重做。有意思的是，根据官方worklog的描述，最初实现truncate操作的原子化时是通过增加新的redo log类型来实现的，但后来不知道为什么又改成了采用日志文件的方式，也许是考虑到低版本兼容的问题吧。

关键结构体

log_sys对象

`log_sys`是InnoDB日志系统的中枢及核心对象，控制着日志的拷贝、写入、checkpoint等核心功能。它同时也是大写入负载场景下的热点模块，是连接InnoDB日志文件及log buffer的枢纽，对应结构体为`log_t`。

其中与 redo log 文件相关的成员变量包括：

变量名	描述
log_groups	日志组，当前版本仅支持一组日志，对应类型为 `log_group_t` 数、每个文件的大小、space id等信息
lsn_t log_group_capacity	表示当前日志文件的总容量，值为:(Redo log文件总大小 - redo LOG_FILE_HDR_SIZE) * 0.9，LOG_FILE_HDR_SIZE 为 4*512
lsn_t max_modified_age_async	异步 preflush dirty page 点
lsn_t max_modified_age_sync	同步 preflush dirty page 点

lsn_t max_checkpoint_age_async	异步 checkpoint 点
lsn_t max_checkpoint_age	同步 checkpoint 点

上述几个sync/async点的计算方式可以参阅函数`log_calc_max_ages`，以如下实例配置为例：

```
innodb_log_files_in_group=4
innodb_log_file_size=4G
```

总文件大小：`17179869184`

各个成员变量值及占总文件大小的比例：

```
log_sys->log_group_capacity = 15461874893 (90%)

log_sys->max_modified_age_async = 12175607164 (71%)

log_sys->max_modified_age_sync = 13045293390 (76%)

log_sys->max_checkpoint_age_async = 13480136503 (78%)

log_sys->max_checkpoint_age = 13914979615 (81%)
```

通常的：

当当前未刷脏的最老lsn和当前lsn的距离超过`max_modified_age_async`（71%）时，且开启了选项`innodb_adaptive_flushing`时，page cleaner线程会去尝试做更多的dirty page flush工作，避免脏页堆积。 当当前未刷脏的最老lsn和当前Lsn的距离超过`max_modified_age_sync`(76%)时，用户线程需要去做同步刷脏，这是一个性能下降的临界点，会极大的影响整体吞吐量和响应时间。 当上次checkpoint的lsn和当前lsn超过`max_checkpoint_age`(81%)，用户线程需要同步地做一次checkpoint，需要等待checkpoint写入完成。 当上次checkpoint的lsn和当前lsn的距离超过`max_checkpoint_age_async`（78%）但小于`max_checkpoint_age`（81%）时，用户线程做一次异步checkpoint（后台异步线程执行CHECKPOINT信息写入操作），无需等待checkpoint完成。

`log_group_t`结构体主要成员如下表所示：

变量名	描述
ulint n_files	lb_logfile的文件个数

lsn_t file_size	文件大小
ulint space_id	Redo log 的space id, 固定大小，值为SRV_LOG_SPACE_FIRST_ID
ulint state	LOG_GROUP_OK 或者 LOG_GROUP_CORRUPTED
lsn_t lsn	该group内写到的lsn
lsn_t lsn_offset	上述lsn对应的文件偏移量
byte** file_header_bufs	Buffer区域，用于设定日志文件头信息，并写入ib logfile。当切换到新的ib_lo lsn，写入头部。 头部信息还包含： LOG_GROUP_ID, LOG_FILE_START_LSN LOG_FILE_WAS_CREATED_BY_HOT_BACKUP(函数log_group_file_header_f
lsn_t scanned_lsn	用于崩溃恢复时辅助记录扫描到的lsn号
byte* checkpoint_buf	Checkpoint缓冲区，用于向日志文件写入checkpoint信息（下文详细描述）

与redo log 内存缓冲区相关的成员变量包括：

变量名	描述
ulint buf_free	Log buffer中当前空闲可写的位置
byte* buf	Log buffer起始位置指针
ulint buf_size	Log buffer 大小，受参数innodb_log_buffer_size控制，但可能会自动e
ulint max_buf_free	值为log_sys->buf_size / LOG_BUF_FLUSH_RATIO - LOG_BUF_FLUSH LOG_BUF_FLUSH_RATIO=2, LOG_BUF_FLUSH_MARGIN=(4 * 512 + 为16k,当buf_free超过该值时，可能触发用户线程去写redo；在事务拷re 值，如果超过buf_free，设置log_sys->check_flush_or_checkpoint为tr
ulint buf_next_to_write	Log buffer偏移量，下次写入redo文件的起始位置，即本次写入的结束位
volatile bool is_extending	Log buffer是否正在进行扩展 （防止过大的redo log entry无法写入buf log长度超过buf_size/2时，就会去调用函数log_buffer_extend,一旦扩展 了！
ulint write_end_offset	本次写入的结束位置偏移量(从逻辑来看有点多余，直接用log_sys->buf_

和Checkpoint检查点相关的成员变量：

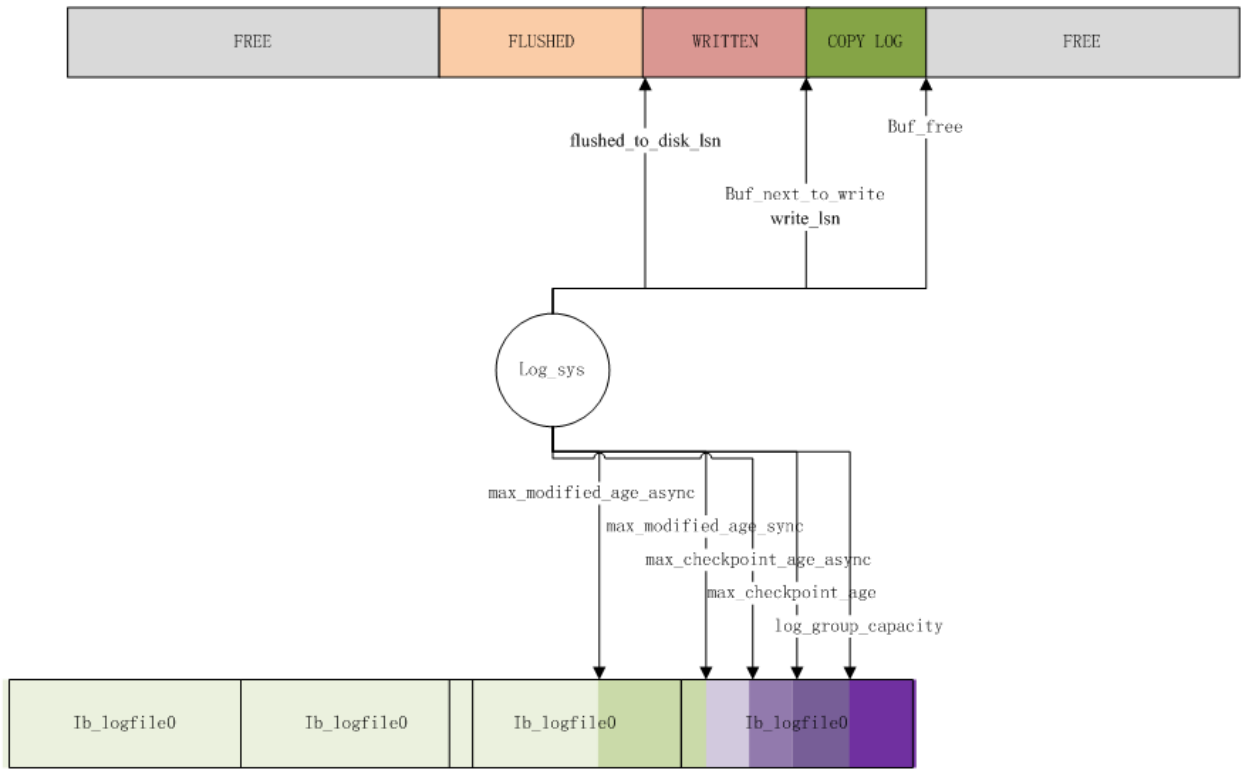
变量名	描述
ib_uint64_t next_checkpoint_no	每完成一次checkpoint递增该值
lsn_t last_checkpoint_lsn	最近一次checkpoint时的lsn，每完成一次checkpoint，将nex last_checkpoint_lsn
lsn_t next_checkpoint_lsn	下次checkpoint的lsn（本次发起的checkpoint的lsn）

mtr_buf_t* append_on_checkpoint	5.7新增，在做DDL时（例如增删列），会先将包含MLOG_FILE 这个变量上。 在DDL完成后，再清理掉。(log_append_on_ch crash产生的数据词典不一致。 该变量在如下commit加上: a5ecc38f44abb66aa2024c70e37d1f4aa4c8ace9
ulint n_pending_checkpoint_writes	大于0时，表示有一个checkpoint写入操作正在进行。用户发起 台线程完成checkpoint写入后，递减该值(log_io_complete)
rw_lock_t checkpoint_lock	checkpoint锁，每次写checkpoint信息时需要加x锁。由异步io
byte* checkpoint_buf	Checkpoint信息缓冲区，每次checkpoint前，先写该buf，再将

其他状态变量

变量名	描述
bool check_flush_or_checkpoint	当该变量被设置时，用户线程可能需要去检查释放要刷log buffer、 等以防止Redo 空间不足
lsn_t write_lsn	最近一次完成写入到文件的LSN
lsn_t current_flush_lsn	当前正在fsync到的LSN
lsn_t flushed_to_disk_lsn	最近一次完成fsync到文件的LSN
ulint n_pending_flushes	表示pending的redo fsync，这个值最大为1
os_event_t flush_event	若当前有正在进行的fsync，并且本次请求也是fsync操作，则需要等

log_sys与日志文件和日志缓冲区的关系可用下图来表示:



Mini transaction

Mini transaction(简称mtr)是InnoDB对物理数据文件操作的最小事务单元，用于管理对Page加锁、修改、释放、以及日志提交到公共buffer等工作。一个mtr操作必须是原子的，一个事务可以包含多个mtr。每个mtr完成后需要将本地产生的日志拷贝到公共缓冲区，将修改的脏页放到flush list上。

mtr事务对应的类为`mtr_t`, `mtr_t::Impl`中保存了当前mtr的相关信息，包括:

变量名	描述
mtr_buf_t m_memo	用于存储该mtr持有的锁类型
mtr_buf_t m_log	存储redo log记录
bool m_made_dirty	是否产生了至少一个脏页
bool m_inside_ibuf	是否在操作change buffer
bool m_modifications	是否修改了buffer pool page
ib_uint32_t m_n_log_recs	该mtr log记录个数
mtr_log_t m_log_mode	Mtr的工作模式，包括四种： MTR_LOG_ALL：默认模式，记录所有会修 MTR_LOG_NONE：不记录redo，脏页也不放到flush list上；MTR_LO 但脏页放到flush list上；MTR_LOG_SHORT_INSERTS：插入记录操作R 贝到另外一个新建的page时用到，此时忽略写索引信息到redo log中。 page_cur_insert_rec_write_log)
fil_space_t* m_user_space	当前mtr修改的用户表空间
fil_space_t* m_undo_space	当前mtr修改的undo表空间
fil_space_t* m_sys_space	当前mtr修改的系统表空间
mtr_state_t m_state	包含四种状态: MTR_STATE_INIT、MTR_STATE_COMMITTING、 MTI

在修改或读一个数据文件中的数据时，一般是通过mtr来控制对对应page或者索引树的加锁，在5.7中，有以下几种锁类型（`mtr_memo_type_t`）：

变量名	描述
MTR_MEMO_PAGE_S_FIX	用于PAGE上的S锁
MTR_MEMO_PAGE_X_FIX	用于PAGE上的X锁
MTR_MEMO_PAGE_SX_FIX	用于PAGE上的SX锁，以上锁通过mtr_memo_push 保存到mtr中

MTR_MEMO_BUF_FIX	PAGE上未加读写锁，仅做buf fix
MTR_MEMO_S_LOCK	S锁，通常用于索引锁
MTR_MEMO_X_LOCK	X锁，通常用于索引锁
MTR_MEMO_SX_LOCK	SX锁，通常用于索引锁，以上3个锁，通过mtr_s/x/sx_lock加锁，通

mtr log生成

InnoDB的redo log都是通过mtr产生的，先写到mtr的cache中，然后再提交到公共buffer中，本小节以INSERT一条记录对page产生的修改为例，阐述一个mtr的典型生命周期。

入口函数：`row_ins_clust_index_entry_low`

开启mtr

执行如下代码块

```
mtr_start(&mtr);
mtr.set_named_space(index->space);
```

mtr_start主要包括：

1. 初始化mtr的各个状态变量
2. 默认模式为MTR_LOG_ALL，表示记录所有的数据变更
3. mtr状态设置为ACTIVE状态（MTR_STATE_ACTIVE）
4. 为锁管理对象和日志管理对象初始化内存（mtr_buf_t）,初始化对象链表

`mtr.set_named_space` 是5.7新增的逻辑，将当前修改的表空间对象`fil_space_t`保存下来：如果是系统表空间，则赋值给`m_impl.m_sys_space`, 否则赋值给`m_impl.m_user_space`。

> Tips： 在5.7里针对临时表做了优化，直接关闭redo记录：mtr.set_log_mode(MTR_LOG_NO_REDO)

定位记录插入的位置

主要入口函数： `btr_cur_search_to_nth_level`

不管插入还是更新操作，都是先以乐观方式进行，因此先加索引S锁 `mtr_s_lock(dict_index_get_lock(index),&mtr)`，对应`mtr_t::s_lock`函数 如果以悲观方式插入记录，意味着可能产生索引分裂，在5.7之前会加索引X锁，而5.7版本则会加SX锁（但某些情况下也会退化成X锁） 加X

锁：`mtr_x_lock(dict_index_get_lock(index), mtr)`,对应`mtr_t::x_lock`函数 加SX锁：`mtr_sx_lock(dict_index_get_lock(index),mtr)`，对应`mtr_t::sx_lock`函数

对应到内部实现，实际上就是加上对应的锁对象，然后将该锁的指针和类型构建的`mtr_memo_slot_t`对象插入到`mtr.m_impl.m_memo`中。

当找到预插入page对应的block，还需要加block锁，并把对应的锁类型加入到mtr：`mtr_memo_push(mtr, block, fix_type)`

如果对page加的是MTR_MEMO_PAGE_X_FIX或者MTR_MEMO_PAGE_SX_FIX锁，并且当前block是clean的，则将`m_impl.m_made_dirty`设置成true，表示即将修改一个干净的page。

如果加锁类型为MTR_MEMO_BUF_FIX，实际上是不加锁对象的，但需要判断临时表的场景，临时表page的修改不加latch，但需要将`m_impl.m_made_dirty`设置为true（根据block的成员`m_impl.m_made_dirty`来判断），这也是5.7对InnoDB临时表场景的一种优化。

同样的，根据锁类型和锁对象构建`mtr_memo_slot_t`加入到`m_impl.m_memo`中。

插入数据

在插入数据过程中，包含大量的redo写cache逻辑，例如更新二级索引页的max trx id、写undo log产生的redo(嵌套另外一个mtr)、修改数据页产生的日志。这里我们只讨论修改数据页产生的日志，进入函数`page_cur_insert_rec_write_log`:

Step 1: 调用函数`mlog_open_and_write_index`记录索引相关信息

1. 调用`mlog_open`，分配足够日志写入的内存地址，并返回内存指针
2. 初始化日志记录：`mlog_write_initial_log_record_fast` 写入 `|类型=MLOG\_COMP\_REC\_INSERT, 1字节|space id | page no|` space id和page no采用一种压缩写入的方式（`mach_write_compressed`），根据数字的具体大小，选择从1到4个字节记录整数，节约redo空间，对应的解压函数为`mach_read_compressed`
3. 写入当前索引列个数，占两个字节
4. 写入行记录上决定唯一性的列的个数，占两个字节(`dict_index_get_n_unique_in_tree`) 对于聚集索引，就是PK上的列数；对于二级索引，就是二级索引列+PK列个数

5. 写入每个列的长度信息，每个列占两个字节 如果这是 varchar 列且最大长度超过255字节, len = 0x7fff；如果该列非空，len |= 0x8000；其他情况直接写入列长度。

Step 2: 写入记录在page上的偏移量，占两个字节

```
mach_write_to_2(log_ptr, page_offset(cursor_rec));
```

Step 3: 写入记录其它相关信息 （rec size, extra size, info bit，关于InnoDB的数据文件物理描述，我们以后再介绍，本文不展开）

Step 4: 将插入的记录拷贝到redo文件，同时关闭mlog

```
memcpy(log_ptr, ins_ptr, rec_size);
mlog_close(mtr, log_ptr + rec_size);
```

通过上述流程，我们写入了一个类型为MLOG_COMP_REC_INSERT的日志记录。由于特定类型的记录都基于约定的格式，在崩溃恢复时也可以基于这样的约定解析出日志。

这里只举了一个非常简单的例子，该mtr中只包含一条redo记录。实际上mtr遍布整个InnoDB的逻辑，但只要遵循相同的写入和读取约定，并对写入的单元（page）加上互斥锁，就能从崩溃恢复。

更多的redo log记录类型参见`enum mlog_id_t`。

在这个过程中产生的redo log都记录在`mtr.m_impl.m_log`中，只有显式提交mtr时，才会写到公共buffer中。

提交mtr log

当提交一个mini transaction时，需要将对数据的更改记录提交到公共buffer中，并将对应的脏页加到flush list上。

入口函数为`mtr_t::commit()`，当修改产生脏页或者日志记录时，调用`mtr_t::Command::execute`，执行过程如下：

Step 1: `mtr_t::Command::prepare_write()`

1. 若当前mtr的模式为MTR_LOG_NO_REDO 或者MTR_LOG_NONE，则获取`log_sys->mutex`，从函数返回
2. 若当前要写入的redo log记录的大小超过log buffer的二分之一，则去扩大log buffer，大小约为原来的两倍。
3. 持有`log_sys->mutex`
4. 调用函数`log_margin_checkpoint_age`检查本次写入： 如果本次产生的redo log size的两倍超过redo log文件capacity，则打印一条错误信息；若本

次写入可能覆盖检查点，还需要去强制做一次同步chekpoint

5. 检查本次修改的表空间是否是上次checkpoint后第一次修改，调用函数（`fil_names_write_if_was_clean`） 如果space->max_lsn = 0，表示自上次checkpoint后第一次修改该表空间： a. 修改`space->max_lsn`为当前`log_sys->lsn`； b. 调用`fil_names_dirty_and_write`将该tablespace加入到`fil_system->named_spaces`链表上； c. 调用`fil_names_write`写入一条类型为MLOG_FILE_NAME的日志，写入类型、spaceid, page no(0)、文件路径长度、以及文件路径名。

在mtr日志末尾追加一个字节的MLOG_MULTI_REC_END类型的标记，表示这是多个日志类型的mtr。

Tips：在5.6及之前的版本中，每次crash recovery时都需要打开所有的ibd文件，如果表的数量非常多时，会非常影响崩溃恢复性能，因此从5.7版本开始，每次checkpoint后，第一次修改的文件名被记录到redo log中，这样在重启从检查点恢复时，就只打开那些需要打开的文件即可（WL#7142）

6. 如果不是从上一次checkpoint后第一次修改该表，则根据mtr中log的个数，或标识日志头最高位为MLOG_SINGLE_REC_FLAG，或附加一个1字节的MLOG_MULTI_REC_END日志。

注意从`prepare_write`函数返回时是持有`log_sys->mutex`锁的。

至此一条插入操作产生的mtr日志格式有可能如下图所示：

Step 2: mtr_t::Command::finish_write

将日志从mtr中拷贝到公共log buffer。这里有两种方式

1. 如果mtr中的日志较小，则调用函数`log_reserve_and_write_fast`，尝试将日志拷贝到log buffer最近的一个block。如果空间不足，走逻辑b)，否则直接拷贝

2. 检查是否有足够的空闲空间后，返回当前的lsn赋值给`m_start_lsn`（`log_reserve_and_open(len)`），随后将日志记录写入到log buffer中。

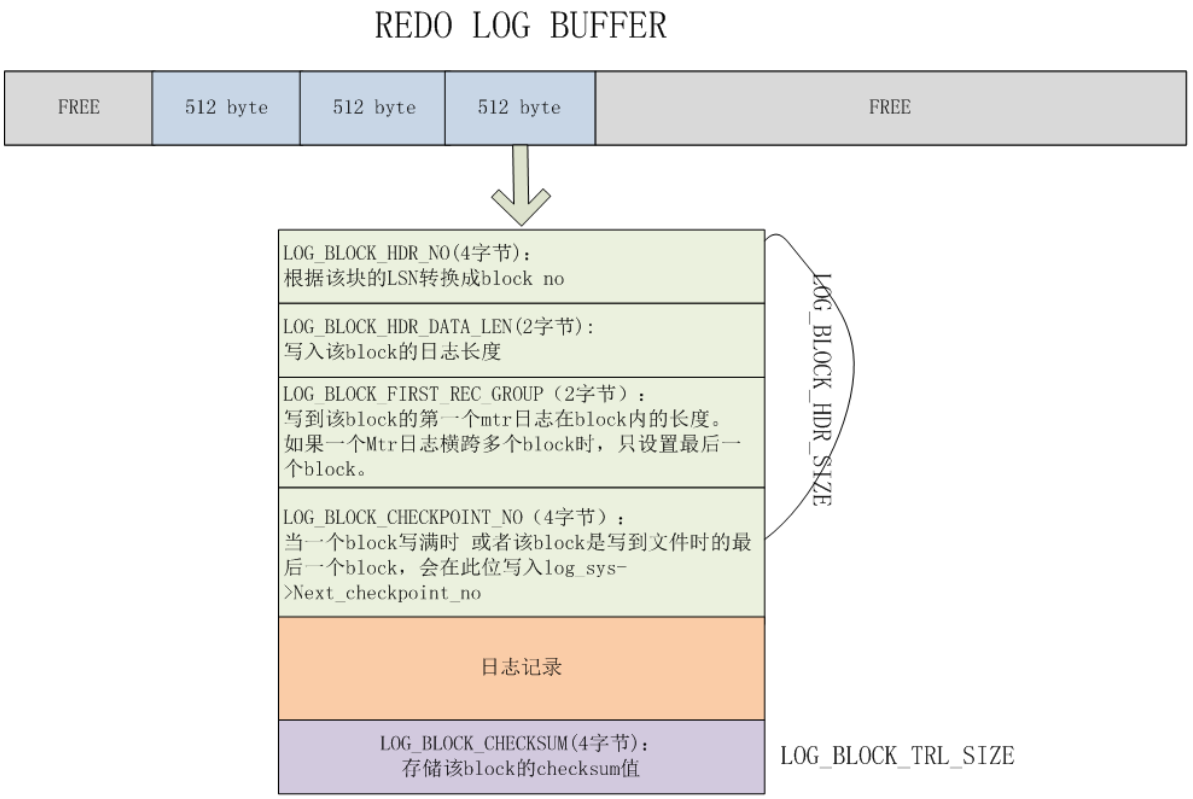
```
m_start_lsn = log_reserve_and_open(len);
mtr_write_log_t write_log;
m_impl->m_log.for_each_block(write_log);
```

3. 在完成将redo 拷贝到log buffer后，需要调用`log_close`, 如果最后一个block未写满，则设置该block头部的LOG_BLOCK_FIRST_REC_GROUP信息；满足如下情况时，设置`log_sys->check_flush_or_checkpoint`为true：

- 当前写入buffer的位置超过log buffer的一半
- bp中最老lsn和当前lsn的距离超过log_sys->max_modified_age_sync
- 当前未checkpoint的lsn age超过log_sys->max_checkpoint_age_async
- 当前bp中最老lsn为0 （没有脏页）

当`check_flush_or_checkpoint`被设置时，用户线程在每次修改数据前调用`log_free_check`时，会根据该标记决定是否刷redo日志或者脏页。

注意log buffer遵循一定的格式，它以512字节对齐，和redo log文件的block size必须完全匹配。由于以固定block size组织结构，因此一个block中可能包含多个mtr提交的记录，也可能一个mtr的日志占用多个block。如下图所示：



Step 3：如果本次修改产生了脏页，获取`log_sys->log_flush_order_mutex`，随后释放`log_sys->mutex`。

Step 4. 将当前Mtr修改的脏页加入到flush list上，脏页上记录的lsn为当前mtr写入的结束点lsn。基于上述加锁逻辑，能够保证flush list上的脏页总是以LSN排序。

Step 5. 释放`log_sys->log_flush_order_mutex`锁

Step 6. 释放当前mtr持有的锁（主要是page latch）及分配的内存，mtr完成提交。

Redo 写盘操作

有几种场景可能会触发redo log写文件：

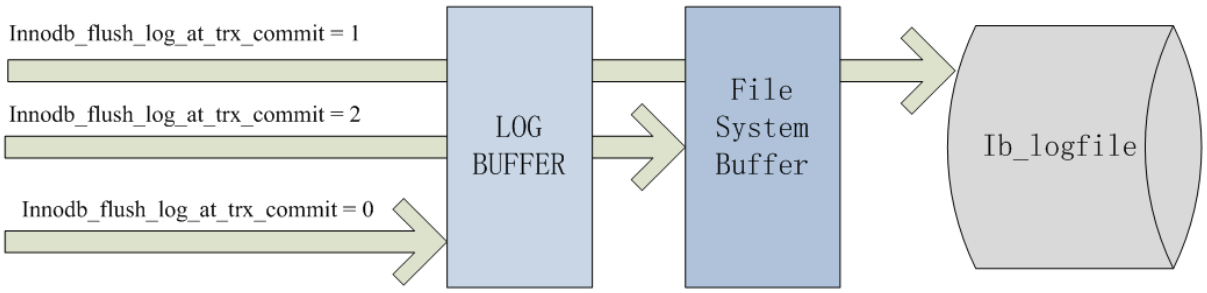
1. Redo log buffer空间不足时
2. 事务提交
3. 后台线程
4. 做checkpoint
5. 实例shutdown时
6. binlog切换时

我们所熟悉的参数`innodb_flush_log_at_trx_commit` 作用于事务提交时，这也是最常见的场景：

- 当设置该值为1时，每次事务提交都要做一次fsync，这是最安全的配置，即使宕机也不会丢失事务；

- 当设置为2时，则在事务提交时只做write操作，只保证写到系统的page cache，因此实例crash不会丢失事务，但宕机则可能丢失事务；
- 当设置为0时，事务提交不会触发redo写操作，而是留给后台线程每秒一次的刷盘操作，因此实例crash将最多丢失1秒钟内的事务。

下图表示了不同配置值的持久化程度：



显然对性能的影响是随着持久化程度的增加而增加的。通常我们建议在日常场景将该值设置为1，但在系统高峰期临时修改成2以应对大负载。

由于各个事务可以交叉的将事务日志拷贝到log buffer中，因而一次事务提交触发的写redo到文件，可能隐式的帮别的线程“顺便”也写了redo log，从而达到group commit的效果。

写redo log的入口函数为`log_write_up_to`，该函数的逻辑比较简单，这里不详细描述，但有几点说明下。

log_write_up_to逻辑重构

首先是在该代码逻辑上，相比5.6及之前的版本，5.7在没有更改日志写主要架构的基础上重写了`log_write_up_to`，让其代码更加可读，同时消除一次多余的获取`log_sys->mutex`，具体的(WL#7050)：

- 早期版本的innodb支持将redo写到多个group中，但现在只支持一个group，因此移除相关的变量，消除`log_write_up_to`的第二个传参；
- write redo操作一直持有`log_sys->mutex`, 所有随后的write请求，不再进入condition wait, 而是通过log_sys->mutex序列化；
- 之前的逻辑中，在write一次redo后，需要释放`log_sys->mutex`，再重新获取，更新相关变量，新的逻辑消除了第二次获取 `log_sys->mutex`；
- write请求的写redo无需等待fsync，这意味着写redo log文件和fsync文件可以同时进行。

理论上该改动可以帮助优化`innodb_flush_log_at_trx_commit=2`时的性能。

log write ahead

上面已经介绍过，InnoDB以512字节一个block的方式对齐写入ib_logfile文件，但现代文件系统一般以4096字节为一个block单位。如果即将写入的日志文件块不在OS Cache时，就需要将对应的4096字节的block读入内存，修改其中的512字节，然后再把该block写回磁盘。

为了解决这个问题，MySQL 5.7引入了一个新参数：`innodb_log_write_ahead_size`。当当前写入文件的偏移量不能整除该值时，则补0，多写一部分数据。这样当写入的数据是以磁盘block size对齐时，就可以直接write磁盘，而无需read-modify-write这三步了。

注意`innodb_log_write_ahead_size`的默认值为8196，你可能需要根据你的系统配置来修改该值，以获得更好的效果。

Innodb redo log checksum

在写入redo log到文件之前，redo log的每一个block都需要加上checksum校验位，以防止apply了损坏的redo log。

然而在5.7.7版本之前版本，都是使用的InnoDB的默认checksum算法（称为InnoDB checksum），这种算法的效率较低。因此在MySQL5.7.8以及Percona Server 5.6版本都支持使用CRC32的checksum算法，该算法可以引用硬件特性，因而具有非常高的效率。

在我的sysbench测试中，使用`update_non_index`，128个并发下TPS可以从55000上升到60000（非双1），效果还是非常明显的。

Redo checkpoint

InnoDB的redo log采用覆盖循环写的方式，而不是拥有无限的redo空间；即使拥有理论上极大的redo log空间，为了从崩溃中快速恢复，及时做checkpoint也是非常有必要的。

InnoDB的master线程大约每隔10秒会做一次redo checkpoint，但不会去preflush脏页来推进checkpoint点。

通常普通的低压力负载下，page cleaner线程的刷脏速度足以保证可作为检查点的lsn被及时的推进。但如果系统负载很高时，redo log推进速度过快，而page cleaner来不及刷脏，这时候就会出现用户线程陷入同步刷脏并做同checkpoint的境地，这种策略的目的是为了保证redo log能够安全的写入文件而不会覆盖最近的检查点。

redo checkpoint的入口函数为`log_checkpoint`，其执行流程如下：

Step1. 持有log_sys->mutex锁，并获取buffer pool的flush list链表尾的block上的lsn，这个lsn是buffer pool中未写入数据文件的最老lsn，在该lsn之前的数据都保证已经写入了磁盘。

Step 2. 调用函数fil_names_clear

1. 如果`log_sys->append_on_checkpoint`被设置，表示当前有会话正处于DDL的commit阶段，但还没有完成，向redo log buffer中追加一个新的redo log记录 该逻辑由commita5ecc38f44abb66aa2024c70e37d1f4aa4c8ace9引入，用于解决DDL过程中crash的问题
2. 扫描`fil_system->named_spaces`上的`fil_space_t`对象，如果表空间`fil_space_t->max_lsn`小于当前准备做checkpoint的Lsn，则从链表上移除并将max_lsn重置为0。同时为每个被修改的表空间构建MLOG_FILE_NAME类型的redo记录。（这一步未来可能会移除，只要跟踪第一次修改该表空间的min_lsn，并且min_lsn大于当前checkpoint的lsn，就可以忽略调用`fil_names_write`）
3. 写入一个MLOG_CHECKPOINT类型的CHECKPOINT REDO记录，并记入当前的checkpoint LSN

Step3 . fsync redo log到当前的lsn

Step4. 写入checkpoint信息

函数：`log_write_checkpoint_info --> log_group_checkpoint`

checkpoint信息被写入到了第一个iblogfile的头部，但写入的文件偏移位置比较有意思，当`log_sys->next_checkpoint_no`为奇数时，写入到LOG_CHECKPOINT_2（3 *512字节）位置，为偶数时，写入到LOG_CHECKPOINT_1（512字节）位置。

大致结构如下图所示：

在crash recover重启时，会读取记录在checkpoint中的lsn信息，然后从该lsn开始扫描redo日志。

Checkpoint操作由异步IO线程执行写入操作，当完成写入后，会调用函数`log_io_complete`执行如下操作：

1. fsync 被修改的redo log文件
2. 更新相关变量：

```
log_sys->next_checkpoint_no++
log_sys->last_checkpoint_lsn = log_sys->next_checkpoint_lsn
```

3. 释放log_sys->checkpoint_lock锁

然而在5.7之前的版本中，我们并没有根据即将写入的数据大小来预测当前是否需要做checkpoint，而是在写之前检测，保证redo log文件中有”足够安全”的空间（而非绝对安全）。假定我们的ib_logfile文件很小，如果我们更新一个非常大的blob字段，就有可能覆盖掉未checkpoint的redo log， 大神Jeremy cole 在buglist上提了一个Bug#69477。

为了解决该问题，在MySQL 5.6.22版本开始，对blob列做了限制: 当redo log的大小超过 (`innodb_log_file_size *innodb_log_files_in_group`)的十分之一时，就会给应用报错，然而这可能会带来不兼容问题，用户会发现，早期版本用的好好的SQL，在最新版本的5.6里居然跑不动了。

在5.7.5及之后版本，则没有5.6的限制，其核心思路是每操作4个外部存储页，就检查一次redo log是否足够用，如果不够，就会推进checkpoint的lsn。当然具体的实现比较复杂，感兴趣的参考如下comit：

f88a5151b18d24303746138a199db910fbb3d071

文件日志

除了普通的redo log日志外，InnoDB还增加了一种文件日志类型，即通过创建特定文件，赋予特定的文件名来标示某种操作。目前有两种类型：undo table space truncate操作及用户表空间truncate操作。通过文件日志可以保证这些操作的原子性。

Undo tablespace truncate

我们知道undo log是MVCC多版本控制的核心模块，一直以来undo log都存储在ibdata系统表空间中，而从5.6开始，用户可以把undo log存储到独立的tablespace中，并拆分成多个Undo log文件，但无法缩小文件的大小。而长时间未提交事务导致大量undo空间的浪费的例子，在我们的生产场景也不是一次两次了。

5.7版本的undo log的truncate操作是基于独立undo 表空间来实现的。在purge线程选定需要清理的undo tablespace后，开始做truncate操作之前，会先创建一个命名为`undo_space_id_trunc.log`的文件，然后将undo tablespace truncate 到10M大小，在完成truncate后删除日志文件。

如果在truncate过程中实例崩溃重启，若发现该文件存在，则认为truncate操作没有完成，需要重做一遍。注意这种文件操作是无法回滚的。

User tablespace truncate

类似的，在5.7版本里，也是通过日志文件来保证用户表空间truncate操作的原子性。在做实际的文件操作前，创建一个命名为`ib_space-id_table-id_trunc.log`的文件。在完成操作后删除。

同样的，在崩溃重启时，如果检查到该文件存在，需要确认是否重做。

InnoDB shutdown

实例关闭分为两种，一种是正常shutdown（非fast shutdown），实例重启时无需apply日志，另外一种是异常shutdown，包括实例crash以及fast shutdown。

当正常shutdown实例时，会将所有的脏页都刷到磁盘，并做一次完全同步的checkpoint；同时将最后的lsn写到系统表ibdata的第一个page中（函数`fil_write_flushed_lsn`）。在重启时，可以根据该lsn来判断这是不是一次正常的shutdown，如果不是就需要去做崩溃恢复逻辑。

参阅函数`logs_empty_and_mark_files_at_shutdown`。

关于异常重启的逻辑，由于崩溃恢复涉及到的模块众多，逻辑复杂，我们将在下期月报单独进行描述。