### 代码执行流程

CREATE TABLE `t` (

`id` int DEFAULT NULL,

`name` varchar(20) DEFAULT NULL

) ENGINE=InnoDB;

insert into t(id,name) value(1,'a');

insert into t(id,name) value(2,'bb');

bt函数调用

|  |
| --- |
| #0 ha\_commit\_trans (thd=0x7fa488000940, all=true, ignore\_global\_read\_lock=true)  at /data/mysql/mysql820/mysql-8.0.20/sql/handler.cc:1589  #1 0x00000000041ddf03 in System\_table\_access::close\_table (this=0x7fa4a67faaf0, thd=0x7fa488000940,  table=0x7fa488006d20, backup=0x7fa4a67fab10, error=false, need\_commit=true)  at /data/mysql/mysql820/mysql-8.0.20/sql/rpl\_table\_access.cc:132  #2 0x00000000041c2315 in Gtid\_table\_access\_context::deinit (this=0x7fa4a67faaf0, thd=0x7fa488000940,  table=0x7fa488006d20, error=false, need\_commit=true)  at /data/mysql/mysql820/mysql-8.0.20/sql/rpl\_gtid\_persist.cc:183  #3 0x00000000041c3057 in Gtid\_table\_persistor::save (this=0x97f72e0, gtid\_set=0x7fa4a67fb230, compress=false)  at /data/mysql/mysql820/mysql-8.0.20/sql/rpl\_gtid\_persist.cc:401  #4 0x0000000004aa3c21 in Clone\_persist\_gtid::write\_to\_table (this=0x7fa8982ec6a8, flush\_list\_number=9,  table\_gtid\_set=..., sid\_map=...) at /data/mysql/mysql820/mysql-8.0.20/storage/innobase/clone/clone0repl.cc:386  #5 0x0000000004aa3f67 in Clone\_persist\_gtid::flush\_gtids (this=0x7fa8982ec6a8, thd=0x7fa488000940)  at /data/mysql/mysql820/mysql-8.0.20/storage/innobase/clone/clone0repl.cc:438  #6 0x0000000004aa440f in Clone\_persist\_gtid::periodic\_write (this=0x7fa8982ec6a8)  at /data/mysql/mysql820/mysql-8.0.20/storage/innobase/clone/clone0repl.cc:519  #7 0x0000000004aa48a9 in clone\_gtid\_thread (persist\_gtid=0x7fa8982ec6a8)  at /data/mysql/mysql820/mysql-8.0.20/storage/innobase/clone/clone0repl.cc:599  #8 0x0000000004aa77f4 in std::\_\_invoke\_impl<void, void (\*&)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*&> (  \_\_f=@0x7fa4a67fbb80: 0x4aa4891 <clone\_gtid\_thread(Clone\_persist\_gtid\*)>)  at /opt/rh/devtoolset-9/root/usr/include/c++/9/bits/invoke.h:60  #9 0x0000000004aa774d in std::\_\_invoke<void (\*&)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*&> (  \_\_fn=@0x7fa4a67fbb80: 0x4aa4891 <clone\_gtid\_thread(Clone\_persist\_gtid\*)>)  at /opt/rh/devtoolset-9/root/usr/include/c++/9/bits/invoke.h:95  #10 0x0000000004aa765e in std::\_Bind<void (\*(Clone\_persist\_gtid\*))(Clone\_persist\_gtid\*)>::\_\_call<void, , 0ul>(std::t  uple<>&&, std::\_Index\_tuple<0ul>) (this=0x7fa4a67fbb80,  \_\_args=<unknown type in /usr/local/mysql820/bin/mysqld, CU 0x1128a0a1, DIE 0x113328bb>)  at /opt/rh/devtoolset-9/root/usr/include/c++/9/functional:400  #11 0x0000000004aa75b4 in std::\_Bind<void (\*(Clone\_persist\_gtid\*))(Clone\_persist\_gtid\*)>::operator()<, void>() (  this=0x7fa4a67fbb80) at /opt/rh/devtoolset-9/root/usr/include/c++/9/functional:484  #12 0x0000000004aa7517 in Runnable::operator()<void (\*)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*>(void (\*&&)(Clone\_  persist\_gtid\*), Clone\_persist\_gtid\*&&) (this=0x99498d8,  f=<unknown type in /usr/local/mysql820/bin/mysqld, CU 0x1128a0a1, DIE 0x11332a9e>)  at /data/mysql/mysql820/mysql-8.0.20/storage/innobase/include/os0thread-create.h:101  #13 0x0000000004aa73f3 in std::\_\_invoke\_impl<void, Runnable, void (\*)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*>(std  ::\_\_invoke\_other, Runnable&&, void (\*&&)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*&&) (  ::\_\_invoke\_other, Runnable&&, void (\*&&)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*&&) (  \_\_f=<unknown type in /usr/local/mysql820/bin/mysqld, CU 0x1128a0a1, DIE 0x11332bfd>)  at /opt/rh/devtoolset-9/root/usr/include/c++/9/bits/invoke.h:60  #14 0x0000000004aa7330 in std::\_\_invoke<Runnable, void (\*)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*>(Runnable&&, vo  id (\*&&)(Clone\_persist\_gtid\*), Clone\_persist\_gtid\*&&) (  \_\_fn=<unknown type in /usr/local/mysql820/bin/mysqld, CU 0x1128a0a1, DIE 0x11332e0d>)  at /opt/rh/devtoolset-9/root/usr/include/c++/9/bits/invoke.h:95  #15 0x0000000004aa7263 in std::thread::\_Invoker<std::tuple<Runnable, void (\*)(Clone\_persist\_gtid\*), Clone\_persist\_gt  id\*> >::\_M\_invoke<0ul, 1ul, 2ul> (this=0x99498c8) at /opt/rh/devtoolset-9/root/usr/include/c++/9/thread:244  #16 0x0000000004aa7202 in std::thread::\_Invoker<std::tuple<Runnable, void (\*)(Clone\_persist\_gtid\*), Clone\_persist\_gt  id\*> >::operator() (this=0x99498c8) at /opt/rh/devtoolset-9/root/usr/include/c++/9/thread:251  #17 0x0000000004aa71e6 in std::thread::\_State\_impl<std::thread::\_Invoker<std::tuple<Runnable, void (\*)(Clone\_persist  \_gtid\*), Clone\_persist\_gtid\*> > >::\_M\_run (this=0x99498c0)  at /opt/rh/devtoolset-9/root/usr/include/c++/9/thread:195  #18 0x00000000050fa5f0 in execute\_native\_thread\_routine ()  #19 0x00007fa8ae426ea5 in start\_thread (arg=0x7fa4a67fc700) at pthread\_create.c:307  #20 0x00007fa8aca159fd in clone () at ../sysdeps/unix/sysv/linux/x86\_64/clone.S:111 |

b mysql\_execute\_command

b ha\_commit\_trans

ha\_commit\_trans(THD \*thd, bool all, bool ignore\_global\_read\_lock) 语句级别的提交

//all意味着这要么是用户发出的显式提交，要么是DDL发出的隐式提交。但这里all的参数是false，说明是语句的提交动作，而非真正的事务commit

rw\_ha\_count = ha\_check\_and\_coalesce\_trx\_read\_only(thd, ha\_info, all) 检查我们是否需要两阶段提交，rw\_ha\_count=2表示一共有两个引擎参与：binlog&&innodb

实际是从ha\_info(ha\_list类型)取出m\_flags，然后m\_flags & (int)TRX\_READ\_WRITE，如果为真rw\_ha\_count++

**Prepare阶段：**

**error = tc\_log->prepare(thd, all) 在事务协调器中准备事务：该函数将在存储引擎中准备事务（通过调用ha\_prepare\_low），它将准备记录写入日志缓冲区。**

int MYSQL\_BIN\_LOG::prepare(THD \*thd, bool all)

thd->durability\_property = HA\_IGNORE\_DURABILITY; 忽略引擎层的持久性，只写入内存

error = ha\_prepare\_low(thd, all)

for (; ha\_info && !error; ha\_info = ha\_info->next()) {

err = ht->prepare(ht, thd, all) => static int binlog\_prepare(handlerton \*, THD \*thd, bool all) for循环第一次调用的是这个函数，binlog层；该函数将在存储引擎中准备事务（通过调用ha\_prepare\_low），它将准备记录写入日志缓冲区。

thd->get\_transaction()->store\_commit\_parent(mysql\_bin\_log.m\_dependency\_tracker.get\_max\_committed\_timestamp());

|  |
| --- |
| (gdb) p mysql\_bin\_log.m\_dependency\_tracker.get\_max\_committed\_timestamp()  $36 = 18  thd->get\_transaction()->store\_commit\_parent(18)  void store\_commit\_parent(int64 last\_arg) { last\_committed = last\_arg; }  last\_committed=18  (gdb) p thd->get\_transaction()->last\_committed  $20 = 18 |

return all && is\_loggable\_xa\_prepare(thd) ? mysql\_bin\_log.commit(thd, true):0; 这个结果return 0

all是false；is\_loggable\_xa\_prepare(thd) 的返回值如果是true，表示当正在准备的事务应该被记录在日志中时。否则没有记录日志，此时为false

err = ht->prepare(ht, thd, all) => innobase\_xa\_prepare(handlerton \*hton, THD \*thd, bool prepare\_trx) for循环第二次调用这个函数，此函数用于准备X/Open XA分布式事务。

\*trx = check\_trx\_exists(thd); 获取MySQL处理程序对象的InnoDB事务句柄trx，如果相应的MySQL线程结构仍然缺少一个InnoDB事务结构，则创建一个InnoDB事务结构。

innobase\_trx\_init(thd, trx) 初始化InnoDB事务对象中的一些字段。外键检查和唯一键检查

thd\_get\_xid(thd, (MYSQL\_XID \*) trx->xid); 从thd中获取xid

err = trx\_prepare\_for\_mysql(trx) 为mysql执行事物做准备

db\_err = trx\_undo\_gtid\_add\_update\_undo(trx, true, false) 对于GTID持久性，我们需要更新undo段。return (DB\_SUCCESS)

alloc = gtid\_persistor.trx\_check\_set(trx, prepare, rollback) return false

gtid\_exists = check\_gtid\_prepare(thd, trx, gtid\_exists, alloc\_check) 检查XA的prepare； return false;

**trx\_prepare(trx) 准备一个事物**

lsn = trx\_prepare\_low(trx, &trx->rsegs.m\_redo, false) 为给定的回滚段准备一个事务。

mtr\_start\_sync(&mtr)=>mtr\_t::start(bool sync, bool read\_only) 开启一个min事物，初始化m\_impl，sync=true，read\_only=false

trx\_undo\_gtid\_set(trx, undo\_ptr->update\_undo) 重置GTID标志；undo->flag &= ~TRX\_UNDO\_FLAG\_GTID;

trx\_undo\_set\_state\_at\_prepare(trx, undo\_ptr->insert\_undo, false, &mtr) 在XA PREPARE或XA ROLLBACK设置undo日志段的状态。

undo\_page = trx\_undo\_page\_get(page\_id\_t(undo->space, undo->hdr\_page\_no), undo->page\_size, mtr); 获取一个undo日志页面的地址

trx\_undo\_gtid\_write(trx, undo\_header, undo, mtr) 写入GTID信息到undo中

**undo->set\_prepared(trx->xid) 将undo segment设置为prepared state并设置XID。**

**mlog\_write\_ulint(seg\_hdr + TRX\_UNDO\_STATE, undo->state, MLOG\_2BYTES, mtr) 写undo->state，undo->state=TRX\_UNDO\_PREPARED=5**

**mlog\_write\_ulint(undo\_header + TRX\_UNDO\_FLAGS, undo->flag, MLOG\_1BYTE, mtr) 写undo->flag，undo->flag=TRX\_UNDO\_FLAG\_XID=1**

**trx\_undo\_write\_xid(undo\_header, &undo->xid, mtr) 写XID，在undo log header写入X/Open XA事务标识 (XID)。**

**return undo\_page**

mtr\_commit(&mtr) 提交一个小事物

cmd.execute() => mtr\_t::Command::execute() redo落盘、将脏页添加到刷新列表、释放资源。

m\_impl->m\_log.for\_each\_block(write\_log) redo落盘

add\_dirty\_blocks\_to\_flush\_list(handle.start\_lsn, handle.end\_lsn) 将在此事物中修改的脏页添加到flush list刷新列表中。

release\_all(); 释放这个事物中获得的latch和blocks。

release\_resources() 释放一些资源，mtr\_buffer回收

lsn = mtr.commit\_lsn() 获取commit()的LSN，return lsn

**tc\_log->prepare(thd, all)总结：**

1. binlog\_prepare：获取了last\_committed的值
2. innobase\_xa\_prepare

* 设置undo页
  + 设置undo的状态TRX\_UNDO\_PREPARED
  + 设置undo的flag TRX\_UNDO\_FLAG\_XID
  + 写undo的XID
* Redo落盘
* 将修改的脏页添加到flush list中

**Commit阶段：**

**tc\_log->commit(thd, all) => TC\_LOG::enum\_result MYSQL\_BIN\_LOG::commit(THD \*thd, bool all) 在事务协调器中提交事务。**

ordered\_commit(thd, all, skip\_commit) => MYSQL\_BIN\_LOG::ordered\_commit(THD \*thd, bool all, bool skip\_commit)

//第0阶段：确保从线程提交顺序，因为它们出现在从节点的中继日志中以将事务刷新到二进制日志。

if (Commit\_order\_manager::wait\_for\_its\_turn\_before\_flush\_stage(thd) || //是DML语句直接返回false

ending\_trans(thd, all) || //检查语句是否即将提交，true表示事物正在提交

Commit\_order\_manager::get\_rollback\_status(thd)) {

if (Commit\_order\_manager::wait(thd)) { // return false

|  |
| --- |
| has\_commit\_order\_manager(THD \*thd) //return is\_mts\_worker(thd) && thd->rli\_slave->get\_commit\_order\_manager() != nullptr  is\_mts\_worker(thd) 如果thd属于Worker 线程，则为true，否则为false。此为false  Commit\_order\_manager \*get\_commit\_order\_manager() { return commit\_order\_mngr; } |

return thd->commit\_error;

}

}

//第1阶段：将事务刷新到二进制日志

process\_flush\_stage\_queue(&total\_bytes,&do\_rotate,&wait\_queue) => MYSQL\_BIN\_LOG::process\_flush\_stage\_queue

THD \*first\_seen = fetch\_and\_process\_flush\_stage\_queue(); 获取整个flush队列并清空它，以便下一批有一个leader。

ha\_flush\_logs(true) 我们将准备好的事务记录刷新到组中的存储引擎日志（例如，InnoDB重做日志），然后再将它们刷新到二进制日志。

plugin\_foreach(nullptr, flush\_handlerton, MYSQL\_STORAGE\_ENGINE\_PLUGIN, static\_cast<void \*>(&binlog\_group\_flush)

plugin\_foreach\_with\_mask(thd, funcs, type, state\_mask, arg)

if (plugin && (\*funcs)(thd, plugin\_int\_to\_ref(plugin), arg)) => flush\_handlerton(THD \*, plugin\_ref plugin, void \*arg)

hton->flush\_logs(hton, \*(static\_cast<bool \*>(arg)))) => innobase\_flush\_logs(handlerton \*hton, bool binlog\_group\_flush) 将InnoDB的redo log刷新到文件系统。

//将redo log buffer刷新到redo log file。如果我们在FLUSH LOGS中，或者如果innodb\_flush\_log\_at\_trx\_commit=1则将其同步到磁盘。

log\_buffer\_flush\_to\_disk(!binlog\_group\_flush ||srv\_flush\_log\_at\_trx\_commit == 1) **组提交的时候一起落盘？？**

std::pair<int, my\_off\_t> result = flush\_thread\_caches(head) => std::pair<int, my\_off\_t> MYSQL\_BIN\_LOG::flush\_thread\_caches(THD \*thd) 刷新会话缓存。

binlog\_cache\_mngr \*cache\_mngr = thd\_get\_cache\_mngr(thd) 获取session级的binlog cache

cache\_mngr->flush(thd, &bytes, &wrote\_xid) => int flush(THD \*thd, my\_off\_t \*bytes\_written, bool \*wrote\_xid) 将两个缓存刷新到binary log的便捷方法。

stmt\_cache.flush(thd, &stmt\_bytes, wrote\_xid) => binlog\_cache\_data::flush(THD \*thd, my\_off\_t \*bytes\_written,bool \*wrote\_xid) 直接返回0，没做任何操作

trx\_cache.flush(thd, &trx\_bytes, wrote\_xid) =>binlog\_cache\_data::flush(THD \*thd, my\_off\_t \*bytes\_written,bool \*wrote\_xid) 将缓存刷新到二进制日志

trn\_ctx->sequence\_number = mysql\_bin\_log.m\_dependency\_tracker.step() 获取sequence\_number

mysql\_bin\_log.write\_transaction(thd, this, &writer) 将Gtid\_log\_event写入二进制日志（在写入语句或事务缓存之前）。

flush\_error = flush\_cache\_to\_file(&flush\_end\_pos) => int MYSQL\_BIN\_LOG::flush\_cache\_to\_file(my\_off\_t \*end\_pos\_var)**将I/O缓存刷新到文件。实际看到binlog落盘**

m\_binlog\_file->flush()

//第2阶段：将二进制日志文件同步到磁盘

Commit\_stage\_manager::get\_instance().wait\_count\_or\_timeout 判断组提交的参数

std::pair<bool, bool> result = sync\_binlog\_file(false) => std::pair<bool, bool> MYSQL\_BIN\_LOG::sync\_binlog\_file(bool force) 调用 fsync() 将文件同步到磁盘。

return std::make\_pair(false, synced) synced=true

//第3阶段：按顺序提交所有事务。

process\_commit\_stage\_queue(thd, commit\_queue) =>MYSQL\_BIN\_LOG::process\_commit\_stage\_queue(THD \*thd, THD \*first) 提交一系列会话

ha\_commit\_low(head, all, false) 提交会话未完成的事务

ht->commit(ht, thd, all))

binlog\_commit(handlerton \*, THD \*, bool) 第一次循环：什么都没做，return 0

innobase\_commit(handlerton \*hton, THD \*thd, bool commit\_trx) 第二次循环：在InnoDB 数据库中提交事务或标记SQL语句结束。

innobase\_commit\_low(trx) 在innodb层提交一个事物

trx\_commit\_for\_mysql(trx) 执行mysql事物是否提交

trx\_commit(trx) 提交一个事物

trx\_commit\_low(trx, mtr) 提交一个事物和一个小事物

serialised = trx\_write\_serialisation\_history(trx, mtr) 如果事务涉及更新，则添加回滚段。

undo\_hdr\_page = trx\_undo\_set\_state\_at\_finish(trx->rsegs.m\_redo.update\_undo, mtr)更改undo的状态为TRX\_UNDO\_CACHED=2，并返回undo page的指针

trx\_commit\_in\_memory(trx, mtr, serialised)

trx\_release\_impl\_and\_expl\_locks(trx, serialised) 更改trx的状态trx->state = TRX\_STATE\_COMMITTED\_IN\_MEMORY

**tc\_log->commit(thd, all)总结：**

第0阶段：return null

第1阶段：

* 调用redo log buffer同步到磁盘（根据innodb\_flush\_log\_at\_trx\_commit参数）**用到组提交的时候刷盘？？**
* Flush binlog cache到binlog file（落盘）

第2阶段：将binlog同步到（fsync）到磁盘

第3阶段：

* 更改undo的状态为TRX\_UNDO\_CACHED=2
* 更改trx的状态trx->state = TRX\_STATE\_COMMITTED\_IN\_MEMORY
* InnoDB commit；