# 1.相关文件

**innobase/handler**

ha\_innodb.cc //innodb存储引擎与mysql server层接口

**innobase/include**

trx0types.h // Transaction system global type definitions

trx0trx.h // The transaction

**innobase/log**

log0log.cc //REDO日志，通过Mini-transaction的mtr\_commit()把日志信息写到REDO日志文件中

log0recv.cc //恢复操作

**mtr**

mtr0log.cc // Mini-transaction logging routines

mtr0mtr.cc // Mini-transaction buffer

**trx**

trx0trx.cc //事务管理相关的内容，如事务开始、事务提交、事务信息打印输出等动作

# 2.架构概述

**MySQLServer层**

提供了显式开启事务（trans\_begin（）函数）、提交事务（trans\_commit（）函数）、回滚事务（trans\_rollback（）函数）、设置保存点（trans\_savepoint（）函数）等操作。

**InnoDB层**

通过MySQL的handler接口，调用innodb\_init（）注册了InnoDB层的事务管理相关的函数。

/\*\* Initialize the InnoDB storage engine plugin. \*/

static int innodb\_init(void \*p) {

…

innobase\_hton->savepoint\_set = innobase\_savepoint;

innobase\_hton->savepoint\_rollback = innobase\_rollback\_to\_savepoint;

…

innobase\_hton->savepoint\_release = innobase\_release\_savepoint;

innobase\_hton->commit = innobase\_commit;

innobase\_hton->rollback = innobase\_rollback;

…

}

# 3.事务管理

## 3.1 事务状态

事务可能的状态如下：

/\*\* Transaction states (trx\_t::state) \*/

enum trx\_state\_t {

TRX\_STATE\_NOT\_STARTED, //没有事务，即事务没有开始

TRX\_STATE\_FORCED\_ROLLBACK, //事务被回滚，这是从ACTIVE状态变迁过来的。

TRX\_STATE\_ACTIVE, //事处于ACTIVE状态，表明事务正在执行的过程中

TRX\_STATE\_PREPARED, //事务提交阶段，支持XA， 2PC的第一阶段即PREPARE阶段

TRX\_STATE\_COMMITTED\_IN\_MEMORY //事务已经提交，这是事务的提交标识。只有事务被设置为提交标识后，才可以释放锁等资源

};

对于处于活动状态（ACTIVE）的事务，其状态变化，还存在几个子状态标识：

/\*\* Transaction execution states when trx->state == TRX\_STATE\_ACTIVE \*/

enum trx\_que\_t {

TRX\_QUE\_RUNNING, // 事务正在执行

TRX\_QUE\_LOCK\_WAIT, // 事务正在等待一个锁

/

TRX\_QUE\_ROLLING\_BACK, // 事务正在回滚过程中

TRX\_QUE\_COMMITTING // 事务正在提交过程中

};

事务状态转换：

Regular transactions:

\* NOT\_STARTED -> ACTIVE -> COMMITTED -> NOT\_STARTED

Auto-commit non-locking read-only:

\* NOT\_STARTED -> ACTIVE -> NOT\_STARTED

XA (2PC):

\* NOT\_STARTED -> ACTIVE -> PREPARED -> COMMITTED -> NOT\_STARTED

Recovered XA:

\* NOT\_STARTED -> PREPARED -> COMMITTED -> (freed)

XA (2PC) (shutdown or disconnect before ROLLBACK or COMMIT):

\* NOT\_STARTED -> PREPARED -> (freed)

## 3.2 数据结构

事务的数据结构trx\_t，把事务的属性如隔离级别和事务的状态、与并发相关的锁和MVCC机制的快照隔离、UNDO日志等关联起来：

struct trx\_t {

…

bool abort; //事务被Abort

trx\_id\_t id; //事务的标识，事务ID

trx\_id\_t no; //事务的序列号

trx\_state\_t state; //事务的状态，状态变迁参见图10-1

ReadView\* read\_view; //活动事务的快照，与并发控制的MVCC机制关联.

trx\_lock\_t lock; //事务上的锁信息，与并发控制的锁机制关联

ulint isolation\_level; //事务的隔离级别

bool ddl\_operation; //事务包含DDL操作

lsn\_t commit\_lsn; //事务提交时刻的LSN，与REDO日志相关，

//要求事务结束前REDO日志落盘...

que\_t \*graph; //查询图...

undo\_no\_t undo\_no; //下一个UNDO日志的记录号，

//也表示事务中被修改或插入的行数

trx\_savept\_t last\_sql\_stat\_start; //保存点，记录上一个SQL的UNDO日志的记录号

trx\_rsegs\_t **rsegs;** //UNDO日志回滚段

bool read\_only; //是否只读事务

bool auto\_commit; //是否autocommit

bool internal; //是否是一个内部事务

…

}

## 3.3 UDNO日志

An undo log is a collection of undo log records associated with **a single read-write transaction**. An undo log record contains information about how to undo the latest change by a transaction to a clustered index record. If another transaction needs to see the original data as part of a consistent read operation, the unmodified data is retrieved from undo log records.

Undo logs exist within **undo log segments**, which are contained within **rollback segments**. Rollback segments reside in **undo tablespaces** and in **the global temporary tablespace**.

Undo logs that reside in the global temporary tablespace are used for transactions that modify data in user-defined **temporary tables**. These undo logs are **not redo-logged**, as they are not required for crash recovery. They are used **only for rollback** while the server is running. This type of undo log benefits performance by avoiding redo logging I/O.

/\*\* Rollback segments assigned to a transaction for undo logging. \*/

struct trx\_rsegs\_t {

/\*\* undo log ptr holding reference to a rollback segment that resides in

system/undo tablespace used for undo logging of tables that needs

to be recovered on crash. \*/

trx\_undo\_ptr\_t m\_redo;

/\*\* undo log ptr holding reference to a rollback segment that resides in

temp tablespace used for undo logging of tables that doesn't need

to be recovered on crash. \*/

trx\_undo\_ptr\_t m\_noredo;

};

/\*\* Represents an instance of rollback segment along with its state variables.\*/

struct trx\_undo\_ptr\_t {

trx\_rseg\_t \*rseg; /\*!< rollback segment assigned to the

transaction, or NULL if not assigned

yet \*/

trx\_undo\_t \*insert\_undo; /\*!< pointer to the insert undo log, or

NULL if no inserts performed yet \*/

trx\_undo\_t \*update\_undo; /\*!< pointer to the update undo log, or

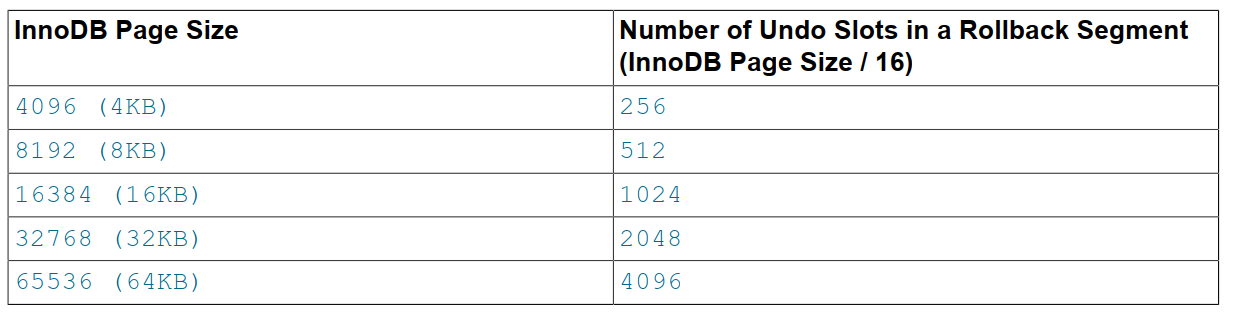
NULL if no update performed yet \*/

};

Each undo tablespace and the global temporary tablespace individually support a maximum of **128 rollback segments**. The innodb\_rollback\_segments variable defines the number of rollback segments.

The number of transactions that a rollback segment supports depends on **the number of undo slots** in the rollback segment and **the number of undo logs** required by each transaction.

The number of undo slots in a rollback segment differs according to InnoDB **page size**.



A transaction is assigned **up to four undo logs**, one for each of the following operation types:

1. INSERT operations on user-defined tables

2. UPDATE and DELETE operations on user-defined tables

3. INSERT operations on user-defined temporary tables

4. UPDATE and DELETE operations on user-defined temporary tables

If each transaction performs either an INSERT or an UPDATE or DELETE operation, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

**(innodb\_page\_size / 16) \* innodb\_rollback\_segments \* number of undo tablespaces**

If each transaction performs an INSERT and an UPDATE or DELETE operation, the number of

concurrent read-write transactions that InnoDB is capable of supporting is:

**(innodb\_page\_size / 16 / 2) \* innodb\_rollback\_segments \* number of undo tablespaces**

If each transaction performs an INSERT operation on a temporary table, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

**(innodb\_page\_size / 16) \* innodb\_rollback\_segments**

If each transaction performs an INSERT and an UPDATE or DELETE operation on a temporary table, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

**(innodb\_page\_size / 16 / 2) \* innodb\_rollback\_segments**

/\*\* The rollback segment memory object \*/

struct trx\_rseg\_t {

ulint id;

/\*\* space ID where the rollback segment header is placed \*/

space\_id\_t space\_id;

/\*\* page number of the rollback segment header \*/

page\_no\_t page\_no;

…

/\*\*执行UPDATE操作产生的UODO日志，事务完成，信息依然被保留，用于MVCC机制下的一致性读\*/

/\* Fields for update undo logs \*/

UT\_LIST\_BASE\_NODE\_T(trx\_undo\_t) update\_undo\_list;

/\*执行INSERT操作产生的UODO日志，这些信息是临时的，事务结束后就被清理\*/

/\* Fields for insert undo logs \*/

UT\_LIST\_BASE\_NODE\_T(trx\_undo\_t) insert\_undo\_list;

…

}

## 3.4 REDO日志

/\*\* Type used for all log sequence number storage and arithmetics. \*/

typedef uint64\_t lsn\_t;

## 3.5 Mini-Transaction

An internal phase of InnoDB processing, when making changes at the **physical** level to internal data structures during **DML** operations. A mini-transaction (mtr) has no notion of **rollback**; **multiple** mini-transactions can occurwithin **a single** **transaction**. Mini-transactions write information to the **redo log** that is used during **crash recovery**. A mini-transaction can also happen **outside** the context of a regular transaction, for example during **purge** processing by background threads.

/\*\* Mini-transaction handle and buffer \*/

**struct mtr\_t {**

/\*\* State variables of the mtr \*/

struct Impl {

mtr\_buf\_t m\_memo; /\*\* memo stack for locks etc. \*/

mtr\_buf\_t m\_log; /\*\* mini-transaction log \*/

bool m\_made\_dirty; /\*\* true if mtr has made at least one buffer pool page dirty \*/

bool m\_inside\_ibuf; /\*\* true if inside ibuf changes \*/

bool m\_modifications; /\*\* true if the mini-transaction modified buffer pool pages \*/

ib\_uint32\_t m\_n\_log\_recs; /\*\* Count of how many page initial log records have been

written to the mtr log \*/

mtr\_log\_t m\_log\_mode; /\*\* specifies which operations should be logged; default

value MTR\_LOG\_ALL \*/

mtr\_state\_t m\_state; /\*\* State of the transaction \*/

FlushObserver \*m\_flush\_observer; /\*\* Flush Observer \*/

mtr\_t \*m\_mtr; /\*\* Owning mini-transaction \*/

};

…

Impl m\_impl;

lsn\_t m\_commit\_lsn; /\*\* LSN at commit time \*/

bool m\_sync; /\*\* true if it is synchronous mini-transaction \*/

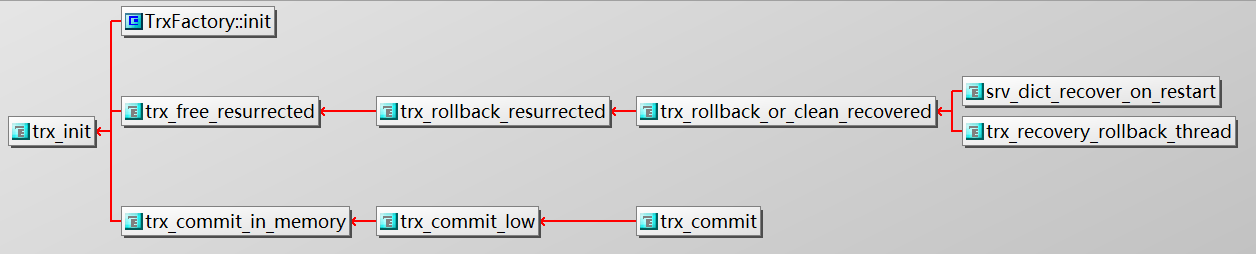
…

**}**

# 4.事务操作

## 4.1 事务初始化

事务开始，首先要初始化事务，在InnoDB中，调用trx\_init（）函数完成事务的初始化。在这个函数中，设置了事务结构体（trx\_t\* trx）的各个元素的值，如默认给定的隔离级别是“可重复读”。从函数被调用的关系图可以看出，事务被初始化，有三种途径：



一是源自于事务工厂TrxFactory调用的初始化init（）。事务工厂就是一个事务池，预先创建、初始化好一些事务对象，避免事务对象地频繁创建和删除。·事务工厂在InnoDB被初始化的时候通过innobase\_init\_files() →…→srv\_boot() →…→trx\_pool\_init() 逐层实现多个事务的池化。

内部的操作可以通过调用trx\_create\_low() 函数从事务池内获取一个事务，（内部的一些操作也是可以被当作事务来处理的，使得一些写系统表的内部操作也事务化）。事务被初始化的时候，事务的状态被设置为TRX\_STATE\_NOT\_STARTED，表示事务没有开始。

/\*\* @return a trx\_t instance from trx\_pools. \*/

static trx\_t \*trx\_create\_low() {

trx\_t \*trx = trx\_pools->get();

…

}

二是用于系统恢复时，调用trx\_free\_resurrected() →trx\_init() 实现事务的初始状态设置。

三是事务提交或回滚之后，在某个session上的事务对象被重新初始化，以备后用（事务实体没有被销毁而是后续备用）。

**#0 trx\_init** // 重新初始化，以备后用

#1 trx\_commit\_in\_memory

#2 trx\_commit\_low

#3 trx\_commit

#4 trx\_commit\_for\_mysql

#5 innobase\_commit\_low

#6 innobase\_commit

## 4.2 事务启动

事务的启动函数比较简单，根据用户的命令做简单区分，然后设置事务的一些基本属性信息。如对于读写事务分配回滚段，对于只读事务进行事务ID的分配等。事务启动的函数trx\_start\_low() 代码分析如下：

static void **trx\_start\_low**(trx\_t \*trx, bool read\_write) /\*\* Starts a transaction. \*/

{ …

++trx->version;

/\* Check whether it is an AUTOCOMMIT SELECT \*/

trx->auto\_commit = (trx->api\_trx && trx->api\_auto\_commit) ||

thd\_trx\_is\_auto\_commit(trx->mysql\_thd);

trx->read\_only = (trx->api\_trx && !trx->read\_write) ||

(!trx->internal && thd\_trx\_is\_read\_only(trx->mysql\_thd)) ||

srv\_read\_only\_mode;

…

if (!trx->read\_only && // 非只读事务

(trx->mysql\_thd == 0 || read\_write || trx->ddl\_operation)) {

trx\_assign\_rseg\_durable(trx);

/\* Temporary rseg is assigned only if the transaction updates a temporary table \*/

//对于非只读类型的事务，事务刚开始不能确定是否要用到临时表，所以临时表要使用到的临时回滚段暂且不分配

trx\_sys\_mutex\_enter();

trx->id = trx\_sys\_get\_new\_trx\_id(); // 获取一个事务ID

trx\_sys->rw\_trx\_ids.push\_back(trx->id);

…

trx->state = TRX\_STATE\_ACTIVE; // 事务开始

…

else {…

if (!trx\_is\_autocommit\_non\_locking(trx)) {

/\* If this is a read-only transaction that is writing to a temporary table then it needs a transaction id to write to the temporary table.

//只读事务，需要写临时表，而临时表需要通过事务的ID做区分

if (read\_write) {

trx\_sys\_mutex\_enter();

trx->id = trx\_sys\_get\_new\_trx\_id();

…

trx\_sys\_mutex\_exit();

}

trx->state = TRX\_STATE\_ACTIVE;

} else {

trx->state = TRX\_STATE\_ACTIVE;

}

…

}

**#0 trx\_start\_low**

#1 trx\_start\_if\_not\_started\_xa\_low

#2 row\_insert\_for\_mysql\_using\_ins\_graph

#3 row\_insert\_for\_mysql

#4 ha\_innobase::write\_row

#5 handler::ha\_write\_row

#6 write\_record

#7 Sql\_cmd\_insert\_values::execute\_inner

#8 Sql\_cmd\_dml::execute

#9 mysql\_execute\_command

## 4.3 提交事务

1. 整体过程

#0 trx\_commit\_in\_memory// 释放锁、修改事务状态、释放插入的UNDO日志，

// 生成新的LSN、刷日志（1）、

// 重新初始化事务对象以备再用

#1 trx\_commit\_low // Mini-Transaction 事务提交

// 调用trx\_commit\_in\_memory()完成事务

// 在内存中的提交等操作

#2 trx\_commit // 调用trx\_commit\_low()完成事务提交

// 也会被trx\_rollback\_finish()调用，用于回滚操作

#3 trx\_commit\_for\_mysql // 根据事务的状态，执行不同的操作。

// 调用trx\_commit是因为事务的状态

// 需要从TRX\_STATE\_ACTIVE或TRX\_STATE\_PREPARED

// 转变为TRX\_STATE\_COMMITTED\_IN\_MEMORY

#4 innobase\_commit\_low // 调用trx\_commit\_for\_mysql()完成事务提交

#5 innobase\_commit // 调用innobase\_commit\_low完成事务提交

// 设置事务提交标志并释放事务相关的锁

// 调用trx\_commit\_complete\_for\_mysql()刷出日志(2)

// 以上是InnoDB层的事务提交相关代码

#6 ha\_commit\_low // MySQL Server层通过handle接口对底层的存储进行事务

// 管理操作，通过函数指针ht->commit()

// 调用InnoDB的innobase\_commit()函数。

#7 TC\_LOG\_DUMMY::commit // 伪接口

#8 ha\_commit\_trans // mysq层进行的事务提交

#9 trans\_commit // mysq层进行的事务提交，

//trans\_commit多语句提交，trans\_commit\_stmt单语句提交

#10 mysql\_execute\_command

/\*\* Commits a transaction in memory. \*/

static void **trx\_commit\_in\_memory**(trx\_t \*trx, const mtr\_t \*mtr, bool serialised)

{

…

trx\_release\_impl\_and\_expl\_locks(trx, serialised); // 释放锁、修改事务状态

…

if (mtr != NULL) {

if (trx->rsegs.m\_redo.insert\_undo != NULL) {

trx\_undo\_insert\_cleanup(&trx->rsegs.m\_redo, false); // 释放插入的UNDO日志

}

…

}

lsn\_t lsn = mtr->commit\_lsn(); //生成新的LSN

…

// 根据innodb\_flush\_log\_at\_trx\_commit参数值，

// 确定是否调用trx\_flush\_log\_if\_needed()刷出日志

} else if ((srv\_flush\_log\_at\_trx\_commit == 0 ||

thd\_requested\_durability(trx->mysql\_thd) ==

HA\_IGNORE\_DURABILITY) &&

(!trx->ddl\_operation)) {

/\* Do nothing \*/

} else {

trx\_flush\_log\_if\_needed(lsn, trx); //

}

trx->commit\_lsn = lsn;

…

/\* Free all savepoints, starting from the first. \*/

trx\_named\_savept\_t \*savep = UT\_LIST\_GET\_FIRST(trx->trx\_savepoints);

trx\_roll\_savepoints\_free(trx, savep); // 释放保存点

…

if (trx->abort) {

trx->abort = false;

// 如果是回滚操作，设置事务状态为TRX\_STATE\_FORCED\_ROLLBACK

trx->state = TRX\_STATE\_FORCED\_ROLLBACK;

} else {

trx->state = TRX\_STATE\_NOT\_STARTED;

}

…

trx\_init(trx); //重新初绍化事务对象

…

}

static void **trx\_release\_impl\_and\_expl\_locks**(trx\_t \*trx, bool serialized) {

…

trx->state = TRX\_STATE\_COMMITTED\_IN\_MEMORY; //修改为提交状态

…

lock\_trx\_release\_locks(trx); // 释放锁

}

static void **trx\_flush\_log\_if\_needed**(lsn\_t lsn, trx\_t \*trx)

{

…

if (trx->ddl\_operation || trx->ddl\_must\_flush) {

log\_write\_up\_to(\*log\_sys, lsn, true);

} else {

trx\_flush\_log\_if\_needed\_low(lsn);

}

…

}

static void **trx\_flush\_log\_if\_needed\_low**(lsn\_t lsn)

{

…

switch (srv\_flush\_log\_at\_trx\_commit) {

case 2:

/\* Write the log but do not flush it to disk \*/

flush = false;

case 1:

wait\_stats = log\_write\_up\_to(\*log\_sys, lsn, flush);

MONITOR\_INC\_WAIT\_STATS(MONITOR\_TRX\_ON\_LOG\_, wait\_stats);

return;

case 0:

/\* Do nothing \*/

return;

}

}

void trx\_commit\_complete\_for\_mysql(trx\_t \*trx) // 第二写日志的机会

{

…

trx\_flush\_log\_if\_needed(trx->commit\_lsn, trx);

trx->must\_flush\_log\_later = false;

trx->ddl\_must\_flush = false;

}

## 4.4 日志落盘

/\*\* Waits until redo log is written up to provided lsn (or greater).

static Wait\_stats **log\_wait\_for\_write**(const log\_t &log, lsn\_t lsn) {

…

const size\_t slot = log\_compute\_write\_event\_slot(log, lsn);

const auto wait\_stats =

os\_event\_wait\_for(log.write\_events[slot], max\_spins,

srv\_log\_wait\_for\_write\_timeout, stop\_condition);

MONITOR\_INC\_WAIT\_STATS(MONITOR\_LOG\_ON\_WRITE\_, wait\_stats);

return (wait\_stats);

}

/\*\* Waits until redo log is flushed up to provided lsn (or greater).

static Wait\_stats **log\_wait\_for\_flush**(const log\_t &log, lsn\_t lsn) {

…

const size\_t slot = log\_compute\_flush\_event\_slot(log, lsn);

const auto wait\_stats =

os\_event\_wait\_for(log.flush\_events[slot], max\_spins,

srv\_log\_wait\_for\_flush\_timeout, stop\_condition);

MONITOR\_INC\_WAIT\_STATS(MONITOR\_LOG\_ON\_FLUSH\_, wait\_stats);

return (wait\_stats);

}

Flushes the write buffers of a given file to the disk.

bool **os\_file\_flush\_func**(os\_file\_t file) {

…

ret = os\_file\_fsync\_posix(file); // int ret = fsync(file);

…

}

## 4.5 回滚事务