|  |
| --- |
|  |
| MySQL笔记 |
| [键入文档副标题] |

|  |
| --- |
| Mike  2018/8/24 |

[MySQL启动 5](#_Toc532910022)

[MySQL中的几个重要的数据结构 7](#_Toc532910023)

[handler 、handlerton、 TABLE 、TABLE\_SHARE四者的关系 8](#_Toc532910024)

[锁 8](#_Toc532910025)

[metadata lock 9](#_Toc532910026)

[表锁 17](#_Toc532910027)

[Innodb 18](#_Toc532910028)

[Innodb 数据结构 18](#_Toc532910029)

[Innodb 启动过程 18](#_Toc532910030)

[innodb内存管理 24](#_Toc532910031)

[建表步骤 26](#_Toc532910032)

[Innodb各种page结构 28](#_Toc532910033)

[Ibdata1文件 29](#_Toc532910034)

[FIL\_HEADER 29](#_Toc532910035)

[FSP\_HDR 29](#_Toc532910036)

[INDEX\_HEADER 30](#_Toc532910037)

[*XDES\_ENTRY* 31](#_Toc532910038)

[*Page 0 ,space header* 31](#_Toc532910039)

[*IBUF\_BITMAP* 32](#_Toc532910040)

[*Page 1,* *IBUF BITMAP PAGE* 32](#_Toc532910041)

[*Inode entry* 33](#_Toc532910042)

[*Page 2 ,Inode page* 34](#_Toc532910043)

[*Segment header结构* 35](#_Toc532910044)

[*IBUF\_TREE\_SEG\_HEADER* 35](#_Toc532910045)

[*Page 3, IBUF HEAD page(insert buffer header页)* 36](#_Toc532910046)

[*Page 4, IBUF ROOT page(insert buffer root页)* 36](#_Toc532910047)

[*Page 5, TRX\_SYS PAGE* 37](#_Toc532910048)

[*Page 6, FSP\_FIRST\_RSEG\_PAGE(rollback segment header)* 39](#_Toc532910049)

[*索引页的page格式* 40](#_Toc532910050)

[*物理记录格式* 40](#_Toc532910051)

[*Compact格式* 40](#_Toc532910052)

[*大记录格式* 44](#_Toc532910053)

[*Redundant格式* 44](#_Toc532910054)

[*Compressed与Dynamic行记录格式* 45](#_Toc532910055)

[Undo log 结构 45](#_Toc532910056)

[*UNDO LOG PAGE HEADER* 46](#_Toc532910057)

[*UNDO LOG SEGMENT HEADER的格式* 46](#_Toc532910058)

[*UNDO LOG HEADER* 47](#_Toc532910059)

[*首个Undo log page* 47](#_Toc532910060)

[*普通Undo log page* 48](#_Toc532910061)

[Redo log 结构 48](#_Toc532910062)

[*Redo log 日志格式* 49](#_Toc532910063)

[*普通的redo log block(512字节) header的组成* 50](#_Toc532910064)

[*redo log 文件的第0个 page(512字节)[ log file header]* 50](#_Toc532910065)

[*redo log 文件的第1个 page(512字节)* 51](#_Toc532910066)

[*redo log 文件的第2个 page(512字节)* 51](#_Toc532910067)

[*redo log 文件的第3个 page(512字节)* 52](#_Toc532910068)

[*Redo log 类型* 52](#_Toc532910069)

[*Undo log 整体结构图* 54](#_Toc532910070)

[锁 57](#_Toc532910071)

[*lock\_sys\_t , lock\_t, lock\_rec\_t ,trx\_t，dict\_table\_t 五者的关系* 57](#_Toc532910072)

[*关于锁 record的几个重要函数* 59](#_Toc532910073)

[*隐式锁和显示锁* 60](#_Toc532910074)

[*Clustered index 记录的隐式锁* 60](#_Toc532910075)

[*Secondary index 记录的隐式锁* 61](#_Toc532910076)

[*Secondary index 记录通过clustered index 记录判断是否有implicit lock的思想* 61](#_Toc532910077)

[*行锁的维护(insert,update ,purge,分裂,合并 时)* 66](#_Toc532910078)

[B+树索引 72](#_Toc532910079)

[*btr\_cur\_search\_to\_nth\_level(index, level, mode, latch\_mode, cursor,has\_search\_latch)* 72](#_Toc532910080)

[*DML操作* 78](#_Toc532910081)

[*插入* 79](#_Toc532910082)

[*非主键更新(update)* 81](#_Toc532910083)

[*主键更新* 84](#_Toc532910084)

[*删除* 84](#_Toc532910085)

[*持久游标(persistent cursor)* 85](#_Toc532910086)

[*哈希索引* 86](#_Toc532910087)

[Insert buffer(change buffer) 87](#_Toc532910088)

[*存储结构* 87](#_Toc532910089)

[*Insert buffer逻辑控制* 89](#_Toc532910090)

[缓存池(buffer pool) 91](#_Toc532910091)

[事务处理 92](#_Toc532910092)

[*事务系统段* 93](#_Toc532910093)

[*trx\_sys\_t和trx\_t类* 93](#_Toc532910094)

## MySQL启动

简化步骤

int mysqld\_main(int argc, char \*\*argv) {

1.load\_defaults 函数把配置文件中【mysqld】中的配置加载到argv中去

if (load\_defaults(MYSQL\_CONFIG\_NAME, load\_default\_groups, &argc, &argv))

return 1;

2. 根据com\_status\_vars填写sql\_statement\_names

init\_sql\_statement\_names();

3.把all\_sys\_vars填写到system\_variable\_hash中去

sys\_var\_init();

4.

handle\_early\_options()

5.

init\_error\_log\_mutex();

6．

mysql\_audit\_initialize();

7．

logger.init\_base();

8.

init\_common\_variables()

9.

my\_init\_signals();

10.

init\_server\_components()

init\_server\_components中调用

1. mdl\_init();
2. table\_def\_init()
3. hostname\_cache\_init(host\_cache\_size)
4. randominit
5. init\_thr\_lock()
6. xid\_cache\_init()
7. delegates\_init()
8. 初始化binlog、relay\_log的文件
9. process\_key\_caches(&ha\_init\_key\_cache)
10. ha\_init\_errors()
11. gtid\_server\_init()
12. plugin\_init(…) 这个函数中初始化各种plugin和存储引擎

在plugin\_init函数中初始化builtins plugs ，这些builtins plugin包括 binlog,password,sha256\_password,CSV,MEMORY,MyISAM, MRG\_MYISAM, ARCHIVE, BLACKHOLE, FEDERATED, InnoDB, INNODB\_TRX, INNODB\_LOCKS, INNODB\_LOCK\_WAITS, INNODB\_CMP, INNODB\_CMP\_RESET, INNODB\_CMPMEM, INNODB\_CMPMEM\_RESET, INNODB\_CMP\_PER\_INDEX, INNODB\_CMP\_PER\_INDEX\_RESET, INNODB\_BUFFER\_PAGE, INNODB\_BUFFER\_PAGE\_LRU, INNODB\_BUFFER\_POOL\_STATS, INNODB\_METRICS, INNODB\_FT\_DEFAULT\_STOPWORD, INNODB\_FT\_DELETED, INNODB\_FT\_BEING\_DELETED, INNODB\_FT\_CONFIG, INNODB\_FT\_INDEX\_CACHE, INNODB\_FT\_INDEX\_TABLE, INNODB\_SYS\_TABLES, INNODB\_SYS\_TABLESTATS, INNODB\_SYS\_INDEXES, INNODB\_SYS\_COLUMNS, INNODB\_SYS\_FIELDS, INNODB\_SYS\_FOREIGN, INNODB\_SYS\_FOREIGN\_COLS, INNODB\_SYS\_TABLESPACES,

INNODB\_SYS\_DATAFILES, PERFORMANCE\_SCHEMA, partition

1. handle\_options(&remaining\_argc, &remaining\_argv, no\_opts,mysqld\_get\_one\_option)
2. ha\_init()
3. initialize\_storage\_engine(default\_storage\_engine, "",&global\_system\_variables.table\_plugin)
4. initialize\_storage\_engine(default\_tmp\_storage\_engine, " temp",&global\_system\_variables.temp\_table\_plugin)
5. ha\_recover(0) 从crash中恢复
6. init\_max\_user\_conn()
7. init\_update\_queries()

11.

init\_ssl()

12.

network\_init();

13.

mysql\_rm\_tmp\_tables() || acl\_init(opt\_noacl) ||

my\_tz\_init((THD \*)0, default\_tz\_name, opt\_bootstrap

14.

grant\_init()

15.

servers\_init(0)

16.

init\_status\_vars();

17.

init\_slave()

18.

initialize\_information\_schema\_acl();

19.

execute\_ddl\_log\_recovery();

20.

Events::init(opt\_noacl || opt\_bootstrap)

21.

read\_init\_file(opt\_init\_file)

22.

crate\_shutdown\_thread();

23,。

start\_handle\_manager();

24

handle\_connections\_sockets()

}

## MySQL中的几个重要的数据结构

1. SELECT\_LEX: 这个struct 对象表示一个 select语句或则子查询语句，通常在mysql\_new\_select函数中创建
2. SELECT\_UNIT: 这个struct 用来关联各个SELECT\_LEX 的上下前后的关系
3. TABLE\_LIST： 这个struct 用来表示SELECT\_LEX中的一个table对象

## handler 、handlerton、 TABLE 、TABLE\_SHARE四者的关系

每个引擎，在一个MySQL进程中只有一个handlerton对象，这个handlerton对象在启动的过程就创建了。

每一个表(frm文件)对应一个TABLE\_SHARE，这个TABLE\_SHARE缓存在tabel\_def\_cache中

但是每一个TABLE\_SHARE对象可以对应多个handler和TABLE,

1个TABLE对象对应1个handler对象

每次创建一个TABLE对象，会缓存到table\_cache\_manager中去，这个table\_cache\_manager中的TABLE是整个进程共享的，所以不同的THD对象，在不同的时刻，可能会用到相同的TABLE对象

TABLE对象负责MySQL层面的状态和操作

Handler对象负责引擎层面的状态和操作

handleton的创建时机

在系统初始化调用

win\_main(int argc, char \* \* argv)

init\_server\_components()

plugin\_init(int \* argc, char \* \* argv, int flags)

plugin\_initialize(st\_plugin\_int \* plugin)

ha\_initialize\_handlerton(st\_plugin\_int \* plugin) 在这个函数中创建各个引擎的handlerton

并且调用plugin->plugin->init(handlerton)这个函数初始化引擎的全局状态，innodb的init函数(innodb\_init)有点长

这个handlerton 存储在installed\_htons[128]和hton2plugin[M AX\_HA] 中

## 锁

在parser工作的时候，根据文法上下确定，TABLE\_LIST对象的 table\_list-> lock\_type= lock\_type和

table\_list-> mdl\_request(MDL\_key::TABLE, ptr->db, ptr->table\_name, mdl\_type, MDL\_TRANSACTION)

MySQL首先获得 mdl锁，然后获得 table lock，最后在engine层获得row lock

MySQL中涉及到锁

metadata lock

数据库的操作语句有DDL（create,alter之类的）和MDL(insert ,update,select)，如果两个客户端在同一时刻对同一个表发起事物操作，如果没有控制，结果将是不可预测的。

比如：

客户端s1:

BEGIN;

INSERT INTO t ... ;

COMMIT;

客户端 s2:

DROP TABLE t;

在master进程中是s1,s2这个顺序执行完毕的，但是binlog中的记录顺序会是

DROP TABLE t;

BEGIN;

INSERT INTO t ... ;

COMMIT;

那么这个次序在slave上执行是要报错的，第二个事物执行时，表已经被删除了。

所以引入了metadata lock，用于保护元数据(数据的数据，在mysql中就是sechema，table，procedure,function之类的)，事物一旦获得mdl锁，就要等到这个事物执行完毕，才释放mdl，

这样就能确保s2不会在s1执行到一半的时候，就开始执行。

在MySQL进程中所有的mdl lock都被保存在mdl\_locks中，mdl\_locks是一个MDL\_map对象，查找key的组成方式【db\_name+ table\_name】，就是MDL\_request->key.ptr() + 1之后的数据。

一个会话连接在实现中对应一个THD实体，一个THD对应一个MDL\_CONTEXT，表示当前thd的mdl锁资源，

一个MDL\_CONTEXT中包含多个MDL\_REQUEST，一个MDL\_REQUEST即是对一个对象的某种类型的lock请求。

每个mdl\_request上有一个ticket对象，ticket中包含lock。

MDL\_CONTEXT包含一个m\_tickets数组，包含该thd取得的所有ticket

mdl加锁调用堆栈

mysql\_execute\_command

execute\_sqlcom\_select

open\_normal\_and\_derived\_tables (sql\_base.cc)

open\_tables (sql\_base.cc)  
 lock\_table\_names (sql\_base.cc)

mdl\_context.acquire\_locks(&mdl\_requests, lock\_wait\_timeout)

open\_and\_process\_table (sql\_base.cc)

open\_table (sql\_base.cc)

open\_table\_get\_mdl\_lock (sql\_base.cc)

MDL\_context::acquire\_lock (mdl.cc)

try\_acquire\_lock\_impl (mdl.cc)

find\_ticket (mdl.cc)

MDL\_ticket::create (mdl.cc)

mdl\_locks.find\_or\_insert(key) (mdl.cc)

mysql\_prlock\_wrlock(&lock->m\_rwlock); (mysql\_thread.h)

lock->can\_grant\_lock(mdl\_request->type, this, false)

MySQL中mdl锁涉及三个重要的枚举类型

1. enum\_mdl\_namespace

enum enum\_mdl\_namespace { GLOBAL=**0**,

SCHEMA,

TABLE,

FUNCTION,

PROCEDURE,

TRIGGER,

EVENT,

COMMIT,

/\* This should be the last ! \*/

NAMESPACE\_END };

2. enum\_mdl\_duration

enum enum\_mdl\_duration {

/\*\*

Locks with statement duration are automatically released at the end

of statement or transaction.

\*/

MDL\_STATEMENT= 0,

/\*\*

Locks with transaction duration are automatically released at the end

of transaction.

\*/

MDL\_TRANSACTION,

/\*\*

Locks with explicit duration survive the end of statement and transaction.

They have to be released explicitly by calling MDL\_context::release\_lock().

\*/

MDL\_EXPLICIT,

/\* This should be the last ! \*/

MDL\_DURATION\_END };

3. enum\_mdl\_type

enum enum\_mdl\_type {

MDL\_INTENTION\_EXCLUSIVE= 0,

MDL\_SHARED,

MDL\_SHARED\_HIGH\_PRIO,

MDL\_SHARED\_READ,

MDL\_SHARED\_WRITE,

MDL\_SHARED\_NO\_WRITE,

MDL\_SHARED\_NO\_READ\_WRITE,

MDL\_EXCLUSIVE,

MDL\_TYPE\_END};

首先：enum\_mdl\_namespace 表示mdl\_request的作用域，比如alter table操作，需要获取TABLE作用域。

然后：enum\_mdl\_duration 表示mdl\_request的持久类型，比如alter table操作，类型是MDL\_STATEMENT，即语句结束，就释放mdl锁。又比如autocommit=0；select 操作，类型是MDL\_TRANSACTION，必须在显示的commit，才释放mdl锁。

最后：enum\_mdl\_type 表示mdl\_request的lock类型，根据这个枚举类型，来判断是否兼容和互斥。

初始化mdl\_request的函数

MDL\_request::init(MDL\_key::enum\_mdl\_namespace mdl\_namespace,

const char \*db\_arg,

const char \*name\_arg,

enum\_mdl\_type mdl\_type\_arg,

enum\_mdl\_duration mdl\_duration\_arg)

初始化MDL\_key的函数

mdl\_key\_init(enum\_mdl\_namespace mdl\_namespace,

const char \*db, const char \*name)

MDL\_request初始化结束后，获取锁的函：

mdl\_context.acquire\_lock(&protection\_request,timeout)

**2.3 按照操作的对象**

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 事例 |
| YYPS为parser\_state,这个对象在dispatch\_command函数中初始化  M\_lock\_type和 m\_mdl\_type的默认值为：  m\_lock\_type= TL\_READ\_DEFAULT;  m\_mdl\_type= MDL\_SHARED\_READ; | | |
| MDL\_INTENTION\_EXCLUSIVE(IX) | 意向排他锁用于global和commit的加锁。 | truncate table t1;  会加如下锁  (GLOBAL, MDL\_STATEMENT, MDL\_INTENTION\_EXCLUSIVE）  Truncate文法解析的时候：  YYPS->m\_lock\_type= TL\_WRITE;  YYPS->m\_mdl\_type= MDL\_EXCLUSIVE;  这两个值成为add\_table\_to\_list时的参数  insert into t1 values(3,'abcde');  分别会加如下锁  (SCHEMA, MDL\_TRANSACTION, MDL\_INTENTION\_EXCLUSIVE)  在set\_lock\_for\_tables中确定  m\_lock\_type = TL\_WRITE\_CONCURRENT\_INSERT  m\_mdl\_type = MDL\_SHARED\_WRITE ，在最后获取的时候确实是MDL\_INTENTION\_EXCLUSIVE了 |
| MDL\_SHARED(S) | 只访问元数据 比如表结构，不访问数据 | set golbal\_read\_only =on 加锁  (GLOBAL，MDL\_EXPLICIT，MDL\_SHARED） |
| MDL\_SHARED\_HIGH\_PRIO(SH) | 用于访问information\_scheam表，不涉及数据 | select \* from information\_schema.tables;  show create table xx;  desc xxx;  会加如下锁：  (TABLE,MDL\_TRANSACTION,MDL\_SHARED\_HIGH\_PRIO) |
| MDL\_SHARED\_READ(SR) | 访问表结构并且读表数据 | select \* from t1;  lock table t1 read;  会加如下锁:  (TABLE，MDL\_TRANSACTION，MDL\_SHARE\_READ） |
| MDL\_SHARED\_WRITE(SW) | 访问表结构并且写表数据 | insert/update/delete/select .. for update  会加如下锁：  (TABLE，MDL\_TRANSACTION，MDL\_SHARE\_WRITE) |
| MDL\_SHARED\_UPGRADABLE(SU) | 是mysql5.6引入的新的metadata lock，  在alter table/create index/drop index会加该锁；可以说是为了online ddl才引入的。特点是允许DML，防止DDL。 | 加如下锁：  （TABLE，MDL\_TRANSACTION，MDL\_SHARED\_UPGRADABLE） |
| MDL\_SHARED\_NO\_WRITE(SNW) | 可升级锁，访问表结构并且读写表数据，并且禁止其它事务写。 | alter table t1 modify c bigint;(非onlineddl)  会加如下锁:  (TABLE，MDL\_TRANSACTION，MDL\_SHARED\_NO\_WRITE） |
| MDL\_SHARED\_NO\_READ\_WRITE(SNRW) | 可升级锁，访问表结构并且读写表数据，并且禁止其它事务读写。 | lock table t1 write;  加锁  (TABLE，MDL\_TRANSACTION,MDL\_SHARED\_NO\_READ\_WRITE)  此处的add\_table\_to\_list的参数为  TL\_READ\_NO\_INSERT 和 MDL\_SHARED\_UPGRADABLE |
| MDL\_EXCLUSIVE(X) | 防止其他线程读写元数据 | CREATE/DROP/RENAME TABLE，  其他online DDL在rename阶段也持有X锁  (TABLE，MDL\_TRANSACTION，MDL\_EXCLUSIVE）  create 时 add\_table\_to\_list 的参数是TL\_WRITE 和MDL\_SHARED |

表锁

# Innodb

## Innodb 数据结构

## Innodb 启动过程

innobase\_init(void\* p)

innobase\_commit\_concurrency\_init\_default

innobase\_start\_or\_create\_for\_mysql()

srv\_boot();

srv\_normalize\_init\_values();

srv\_general\_init();

ut\_mem\_init();

UT\_LIST\_INIT(ut\_mem\_block\_list);

recv\_sys\_var\_init();

os\_sync\_init();

sync\_init();

mem\_init(srv\_mem\_pool\_size);

mem\_comm\_pool = mem\_pool\_create(size/\* 8 M \*/)

pool = static\_cast<mem\_pool\_t\*>(ut\_malloc(sizeof(mem\_pool\_t)));

pool->buf = static\_cast<byte\*>(ut\_malloc\_low(size, TRUE));

pool->size = size;

area = (mem\_area\_t\*)(pool->buf + used);

que\_init();

row\_mysql\_init();

UT\_LIST\_INIT(row\_mysql\_drop\_list);

srv\_init();

首先初始化srv\_sys 对象

dict\_ind\_init(); 创建表 SYS\_DUMMY1 和SYS\_DUMMY2

srv\_conc\_init(); 创建srv\_conc\_queue

trx\_i\_s\_cache\_init(trx\_i\_s\_cache);

ut\_crc32\_init();

dict\_mem\_init();

dict\_temp\_file\_num = ut\_crc32(buf, sizeof(now));

srv\_mon\_create();

创建read\_io\_thread和 write\_io\_thread

os\_aio\_init(io\_limit,srv\_n\_read\_io\_threads,srv\_n\_write\_io\_threads,

SRV\_MAX\_N\_PENDING\_SYN C\_IOS)

os\_io\_init\_simple(void)

srv\_reset\_io\_thread\_op\_info()

os\_aio\_array\_create(n\_read\_segs \* n\_per\_seg, n\_read\_segs)

fil\_init(srv\_file\_per\_table ? 50000 : 5000, srv\_max\_n\_open\_files)

buf\_pool\_init(srv\_buf\_pool\_size, srv\_buf\_pool\_instances)

根据初始化innodb\_buffer\_pool\_size和innodb\_buffer\_pool\_instaces初始化 Buffer pool ，

每个 buf\_pool\_instance的大小= srv\_buf\_pool\_size / srv\_buf\_pool\_instances

分配buf\_pool\_ptr[srv\_buf\_pool\_instances]

buf\_pool\_init\_instance(ptr, size, i)

buf\_pool\_set\_sizes();

buf\_LRU\_old\_ratio\_update(100 \* 3/ 8, FALSE);

btr\_search\_sys\_create(buf\_pool\_get\_curr\_size() / sizeof(void\*) / 64)

btr\_search\_sys = (btr\_search\_sys\_t\*)mem\_alloc(sizeof(btr\_search\_sys\_t));

btr\_search\_sys->hash\_index = ib\_create(hash\_size, 0,MEM\_HEAP\_FOR\_BTR\_SEARCH, 0)

fsp\_init()

log\_init() 初始化 日志系统 包括日志写入、LSN管理，checkpoint 、日志刷新以及数据恢复等操作

log\_sys->lsn = LOG\_START\_LSN

lock\_sys\_create(srv\_lock\_table\_size) 创建lock\_sys对象，保存innodb系统中的锁

os\_thread\_create(io\_handler\_thread,n + i, thread\_ids + i) //创建innodb\_write\_io\_threads + innodb\_read\_io\_threads 个io线程

recv\_sys\_create()

recv\_sys\_init(buf\_pool\_get\_curr\_size()) 初始化日志恢复系统

open\_or\_create\_data\_files() 用来创建或则打开系统数据文件ibdata，并且读取一些头文件信息，如LSN

buf\_dblwr\_init\_or\_load\_pages()

fil\_read\_first\_page()

fsp\_flags\_set\_page\_size()

fil\_space\_create()

fil\_node\_create()

fil\_open\_log\_and\_system\_tablespace\_files()

srv\_undo\_tablespaces\_init(create\_new\_db,srv\_undo\_tablespaces,&srv\_undo\_tablespaces\_open)

for (i = 0; i < n\_undo\_tablespaces; ++i) {

//fsp\_header\_init 的作用是在系统文件ibdata的一开始出分配空间，以便可以存储管理一些系统模块，比如事物系统，Inode页面，回滚段系统页面以及数据字典页面，这些页面都存储在ibdata的头几个页面中

fsp\_header\_init(

undo\_tablespace\_ids[i],

SRV\_UNDO\_TABLESPACE\_SIZE\_IN\_PAGES, &mtr);

}

dict\_stats\_thread\_init()

trx\_sys\_file\_format\_init()

trx\_sys\_create()

if (create\_new\_db) {

mtr\_start(&mtr)

fsp\_header\_init(0, sum\_of\_new\_sizes, &mtr)

mtr\_commit(&mtr)

trx\_sys\_create\_sys\_pages() //对事物系统存储初始化，事物系统使用5号页面（也就是ibdata的第六个页面），这些页面一个比较重要的东西就是事物ID，因为事物ID在MVCC及事物的ACID管理中很重要，并且不能被重复，所以这个值被固化在页面中

trx\_sys\_init\_at\_db\_start()

trx\_purge\_sys\_create(srv\_n\_purge\_threads, ib\_bh)

dict\_create() //因为是新建库，所以需要新建的数据字典，这里所做的操作，与上面的相似，首先分配一个ibdata文件的第8个页面，用来存储数据字典使用到的几个ID值,分别是ROWID,表ID,索引ID,当前最大的表空间ID等等，这些ID与前面的ID比较相似，所以有相同的处理。然后为每一个系统表创建一个B树，用来存储在系统启动之后，用户创建的数据库对象，比如表，索引等。最后通过调用dict\_boot把所有的系统表加载到内存中，以便处理DDL请求

dict\_hdr\_create(&mtr)

dict\_boot()

dict\_insert\_initial\_data()

log\_get\_lsn()

buf\_flush\_wait\_batch\_end(NULL, BUF\_FLUSH\_LIST)

fil\_write\_flushed\_lsn\_to\_data\_files(max\_flushed\_lsn, 0)

fil\_flush\_file\_spaces(FIL\_TABLESPACE)

create\_log\_files\_rename(logfilename, dirnamelen,max\_flushed\_lsn, logfile0)

} else if (srv\_archive\_recovery) {

recv\_recovery\_from\_archive\_start()

dict\_boot()

trx\_sys\_init\_at\_db\_start()

trx\_purge\_sys\_create(srv\_n\_purge\_threads, ib\_bh)

srv\_startup\_is\_before\_trx\_rollback\_phase = FALSE

recv\_recovery\_from\_archive\_finish()

} else {

trx\_sys\_file\_format\_max\_check(srv\_max\_file\_format\_at\_startup)

buf\_pool\_invalidate()

//下面这个函数非常非常的重要

recv\_recovery\_from\_checkpoint\_start(LOG\_CHECKPOINT, LSN\_MAX,min\_flushed\_lsn, max\_flushed\_lsn)

dict\_boot()//将所有的系统表加载到内存中，而不会涉及创建数据字典及初始化字典存储页面

trx\_sys\_init\_at\_db\_start()//初始化事物系统，将所有回滚段中需要处理的事物加载进来，包括INSERT回滚段及UPDATE回滚段，用来为后面的操作作准备，

trx\_purge\_sys\_create(srv\_n\_purge\_threads, ib\_bh)

recv\_recovery\_from\_checkpoint\_finish()

dict\_check\_tablespaces\_and\_store\_max\_id(dict\_check)

log\_get\_lsn()

buf\_flush\_wait\_batch\_end(NULL, BUF\_FLUSH\_LIST)

log\_buffer\_flush\_to\_disk()

fil\_flush(SRV\_LOG\_SPACE\_FIRST\_ID)

fil\_write\_flushed\_lsn\_to\_data\_files(max\_flushed\_lsn, 0)

fil\_flush\_file\_spaces(FIL\_TABLESPACE)

fil\_close\_log\_files(true)

log\_group\_close\_all()

create\_log\_files\_rename(logfilename, dirnamelen,

max\_flushed\_lsn, logfile0)

recv\_recovery\_rollback\_active()

trx\_sys\_file\_format\_tag\_init()

}

if (!create\_new\_db && sum\_of\_new\_sizes > 0) {

/\* New data file(s) were added \*/

mtr\_start(&mtr);

fsp\_header\_inc\_size(0, sum\_of\_new\_sizes, &mtr);

mtr\_commit(&mtr);

/\* Immediately write the log record about increased tablespace

size to disk, so that it is durable even if mysqld would crash

quickly \*/

log\_buffer\_flush\_to\_disk();

}

if (buf\_dblwr == NULL) {

/\* Create the doublewrite buffer to a new tablespace \*/

buf\_dblwr\_create();

}

trx\_sys\_create\_rsegs(srv\_undo\_tablespaces, srv\_undo\_logs)

dict\_create\_or\_check\_foreign\_constraint\_tables()

dict\_create\_or\_check\_sys\_tablespace()

os\_thread\_create(srv\_master\_thread,NULL, thread\_ids + (1 + SRV\_MAX\_N\_IO\_THREADS))

创建srv\_master\_thread线程，这个线程每个一秒进行一次后台循环，在空闲与繁忙的阶段分别作不太相同的事情，但就其根本而言是基本相同的，所做的事情主要包括：后台删除废弃表，检查日志空间是否足够，后台合并Insert Buffer 缓存、日志刷盘，做检查点。

os\_thread\_create(srv\_purge\_coordinator\_thread ,NULL, thread\_ids + 5 + SRV\_MAX\_N\_IO\_THREADS)

os\_thread\_create(srv\_worker\_thread, NULL,thread\_ids + 5 + i + SRV\_MAX\_N\_IO\_THREADS);

srv\_purge\_coordinator\_thread和srv\_worker\_thread两个线程通过相互配合，来完成整个InnoDB系统的PURGE操作，

os\_thread\_create(buf\_flush\_page\_cleaner\_thread, NULL, NULL)

buf\_flush\_page\_cleaner\_thread这个线程的主要工作时在后台每隔一秒钟，试图去刷一次Buffer页面，但具体刷多少，需要根据当前的系统负载来决定。如果innodb处于活动状态，则每次都刷一个比列的页面就可以，如果innodb处于空闲状态，则每次都刷100%的页面，当然，这只是后台刷，刷盘包括好几个时机

## innodb内存管理

类型:

层级关系

struct buf\_pool\_t 包含 chuncks

struct buf\_chunk\_t 包含 blocks

struct buf\_block\_t 包含一个page和frame

struct buf\_page\_t

typedef struct mem\_block\_info\_t mem\_block\_t;

typedef mem\_block\_t mem\_heap\_t;

全局变量:

buf\_pool\_t\* buf\_pool\_ptr ; // buffer pool

struct buf\_pool\_t {

ib\_mutex\_t mutex;

ib\_mutex\_t zip\_mutex;

ulint old\_pool\_size;

ulint curr\_pool\_size;

ulint LRU\_old\_ratio;

ulint buddy\_n\_frames;

ulint mutex\_exit\_forbidden;

ulint n\_chunks;

buf\_chunk\_t\* chunks;

ulint curr\_size;

hash\_table\_t\* page\_hash;

hash\_table\_t\* zip\_hash;

ulint n\_pend\_reads;

ulint n\_pend\_unzip;

time\_t last\_printout\_time;

buf\_buddy\_stat\_t buddy\_stat[BUF\_BUDDY\_SIZES\_MAX + 1];

buf\_pool\_stat\_t stat;

buf\_pool\_stat\_t old\_stat;

ib\_mutex\_t flush\_list\_mutex;

buf\_page\_t\* flush\_list\_hp;

UT\_LIST\_BASE\_NODE\_T(buf\_page\_t) flush\_list;

ibool init\_flush[BUF\_FLUSH\_N\_TYPES];

ulint n\_flush[BUF\_FLUSH\_N\_TYPES];

os\_event\_t no\_flush[BUF\_FLUSH\_N\_TYPES];

ib\_rbt\_t flush\_rbt;

ulint freed\_page\_clock;

ibool try\_LRU\_scan;

UT\_LIST\_BASE\_NODE\_T(buf\_page\_t) free;

UT\_LIST\_BASE\_NODE\_T(buf\_page\_t) LRU;

buf\_page\_t\* LRU\_old;

ulint LRU\_old\_len;

UT\_LIST\_BASE\_NODE\_T(buf\_page\_t) unzip\_LRU;

}

在mysql 5.6中，innodb的底部内存主要分成三个组成部分

1. mem\_pool\_t\* mem\_comm\_pool , 这里分配小内存

2. buf\_pool\_t\* buf\_pool\_ptr ， 这里分配大内存

3. UT\_LIST\_BASE\_NODE\_T(ut\_mem\_block\_t) ut\_mem\_block\_list

4. malloc 分配的内存存放在 ut\_mem\_block\_list 中

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

内存分配过程

mem\_alloc\_func(ulint n /\*预期分配的大小\*/ , ulint\* size /\*实际分配的大小\*/)

heap = mem\_heap\_create\_at(n, file\_name, line);

block = UT\_LIST\_GET\_LAST(heap->base)

block = mem\_heap\_add\_block(heap, n);//如果block的空闲不够，需要添加一个block到heap中去

buf = mem\_heap\_alloc(heap, n);

## 建表步骤

1. create\_table\_impl(THD \* thd, const char \* db, const char \* table\_name, const char \* error\_table\_name, const char \* path, st\_ha\_create\_information \* create\_info, Alter\_info \* alter\_info, bool internal\_tmp\_table, unsigned int select\_field\_count, bool no\_ha\_table, bool \* is\_trans, st\_key \* \* key\_info, unsigned int \* key\_count)

2. rea\_create\_table(THD \* thd, const char \* path, const char \* db, const char \* table\_name,

st\_ha\_create\_information \* create\_info, List<Create\_field> & create\_fields,

unsigned int keys, st\_key \* key\_info, handler \* file, bool no\_ha\_table)

3. ha\_create\_table(THD \* thd, const char \* path, const char \* db,

const char \* table\_name, st\_ha\_create\_information \* create\_info,

bool update\_create\_info, bool is\_temp\_table)

4. handler::ha\_create(const char \* name, TABLE \* form,

st\_ha\_create\_information \* info)

5. ha\_innobase::create(const char \* name, TABLE \* form,

st\_ha\_create\_information \* create\_info)

6.create\_table\_def(trx\_t \* trx, const TABLE \* form, const char \* table\_name,

const char \* temp\_path, const char \* remote\_path,

unsigned long flags, unsigned long flags2)

7. row\_create\_table\_for\_mysql(dict\_table\_t \* table, trx\_t \* trx, bool commit)

8. que\_run\_threads(que\_thr\_t \* thr)

9. que\_run\_threads\_low(que\_thr\_t \* thr)

10. que\_thr\_step(que\_thr\_t \* thr)

11. dict\_create\_table\_step(que\_thr\_t \* thr)

12. dict\_build\_table\_def\_step(que\_thr\_t \* thr,

tab\_node\_t \* node)

# Innodb各种page结构

几个特殊的page number

|  |  |  |
| --- | --- | --- |
|  |  | 备注 |
| FSP\_HDR | 0 |  |
| FSP\_IBUF\_BITMAP\_OFFSET | 1 |  |
| FSP\_FIRST\_INODE\_PAGE\_NO | 2 |  |
| FSP\_IBUF\_HEADER\_PAGE\_NO  (IBUF\_HEADER\_PAGE\_NO) | 3 | 类型为FIL\_PAGE\_TYPE\_SYS |
| FSP\_IBUF\_TREE\_ROOT\_PAGE\_NO  (IBUF\_TREE\_ROOT\_PAGE\_NO) | 4 | 类型为 FIL\_PAGE\_INDEX 用于存储change buffer的根page，change buffer目前存储于Ibdata中, IBUF HEADER Page 和Root Page联合起来对ibuf的数据页进行管理 |
| FSP\_TRX\_SYS\_PAGE\_NO | 5 |  |
| FSP\_FIRST\_RSEG\_PAGE\_NO | 6 |  |
| FSP\_DICT\_HDR\_PAGE\_NO | 7 |  |

## Ibdata1文件

## FIL\_HEADER

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_PAGE\_SPACE | 4 | ID of the space the page is in |
| FIL\_PAGE\_OFFSET | 4 | ordinal page number from start of space |
| FIL\_PAGE\_PREV | 4 | offset of previous page in key order |
| FIL\_PAGE\_NEXT | 4 | offset of next page in key order |
| FIL\_PAGE\_LSN | 8 | log serial number of page's latest log record |
| FIL\_PAGE\_TYPE | 2 | current defined types are:  FIL\_PAGE\_INDEX 17855  FIL\_PAGE\_UNDO\_LOG 2  FIL\_PAGE\_INODE 3  FIL\_PAGE\_IBUF\_FREE\_LIST 4  FIL\_PAGE\_TYPE\_ALLOCATED 0  FIL\_PAGE\_IBUF\_BITMAP 5  FIL\_PAGE\_TYPE\_SYS 6  FIL\_PAGE\_TYPE\_TRX\_SYS 7  FIL\_PAGE\_TYPE\_FSP\_HDR 8  FIL\_PAGE\_TYPE\_XDES 9  FIL\_PAGE\_TYPE\_BLOB 10  FIL\_PAGE\_TYPE\_ZBLOB 11  FIL\_PAGE\_TYPE\_ZBLOB2 12 |
| FIL\_PAGE\_FILE\_FLUSH\_LSN | 8 | "the file has been flushed to disk at least up to this lsn" (log serial number), valid only on the first page of the file |
| FIL\_PAGE\_ARCH\_LOG\_NO | 4 | the latest archived log file number at the time that FIL\_PAGE\_FILE\_FLUSH\_LSN was written (in the log) |
| 总共38字节 |  |  |
|  |  |  |

## FSP\_HDR

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FSP\_SPACE\_ID | 4 |  |
| FSP\_NOT\_USED | 4 |  |
| FSP\_SIZE | 4 |  |
| FSP\_FREE\_LIMIT | 4 |  |
| FSP\_SPACE\_FLAGS | 4 |  |
| FSP\_FRAG\_N\_USED | 4 |  |
| FSP\_FREE | 16字节 |  |
| FSP\_FREE\_FRAG | 16字节 |  |
| FSP\_FULL\_FRAG | 16字节 |  |
| FSP\_SEG\_ID | 8字节 | 当前文件中最大Segment ID + 1，用于段分配时的seg id计数器 |
| FSP\_SEG\_INODES\_FULL | 16字节 | 已被完全用满的Inode Page链表 |
| FSP\_SEG\_INODES\_FREE | 16字节 | 至少存在一个空闲Inode Entry的Inode Page被放到该链表上 |

## INDEX\_HEADER

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| PAGE\_N\_DIR\_SLOTS | 2 | Page directory中槽的数量 |
| PAGE\_HEAP\_TOP | 2 | 堆中空闲的位置(偏移量) |
| PAGE\_N\_HEAP | 2 | 堆中的记录数量 |
| PAGE\_FREE | 2 | 指向页中空闲空间的位置 |
| PAGE\_GARBAGE | 2 | 已删除记录的字节数，即行记录结构中delete flag=1的记录大小的总数 |
| PAGE\_LAST\_INSERT | 2 | 最后插入记录的位置 |
| PAGE\_DIRECTION | 2 | 最后插入的方向:  PAGE\_LEFT(0x01)  PAGE\_RIGHT(0x02)  PAGE\_SAME\_REC(0x03)  PAGE\_SAME\_PAGE(0x04)  PAGE\_NO\_DIRECTIONI(0x05) |
| PAGE\_N\_DIRECTION | 2 | 一个方向连续插入记录的数量 |
| PAGE\_N\_RECS | 2 | 该页中用户记录(user record)的数量 |
| PAGE\_MAX\_TRX\_ID | 8 | 修改当前页的事务id，该值仅在辅助页中使用 |
| PAGE\_LEVEL  (PAGE\_HEADER\_PRIV\_END) | 2 |  |
| PAGE\_INDEX\_ID | 8 | 索引id，表示当前页属于哪个索引 |
| PAGE\_BTR\_SEG\_LEAF | 10 | B+树叶子节点所在段的segment header.该值仅在B+树的root页中定义 |
| PAGE\_BTR\_SEG\_TOP | 10 | B+树非叶子节点所在段的segment header.该值仅在B+树的root页中定义 |
| 总共36+20字节 | | |

## XDES\_ENTRY

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| XDES\_ID | 8 | The identifier of the segment to which this extent belongs |
| XDES\_FLST\_NODE | 12 | The list node data structure for the descriptors |
| XDES\_STATE | 4 | 取值：  XDES\_FREE  XDES\_FREE\_FRAG  XDES\_FULL\_FRAG  XDES\_FSEG |
| XDES\_BITMAP | 16字节 | 每个page用2个bit描述，  取值：  XDES\_FREE\_BIT 0  XDES\_CLEAN\_BIT 1 |
| 总共 40字节 |  |  |

## Page 0 ,space header

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38字节 | PAGE\_TYPE=FSP\_HDR |
| FSP\_HEADER | 112字节 |  |
| ------------------------------------------------------------------------------------------------------------------------- | | |
|  | | |
| 38+112=150字节 | | |
| ------------------------------------------------------------------------------------------------------------------------- | | |
| XDES\_ENTRY | 40字节 |  |
| XDES\_ENTRY | 40字节 |  |
| XDES\_ENTRY | 40字节 |  |
| XDES\_ENTRY | 40字节 |  |
| 。。。。。。 |  |  |
| FSP\_HDR页面中保存256个XDES\_ENTRY | | |

## IBUF\_BITMAP

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| IBUF\_BITMAP\_FREE | 2bits | 使用2个bit来描述page的空闲空间范围：  0（0 bytes）、  1（512 bytes）、  2（1024 bytes）、  3（2048 bytes） |
| IBUF\_BITMAP\_BUFFERED | 1bit | 用于表示该page是否有操作缓存，在ibuf\_insert\_low函数中，准备插入ibuf btree前设置成true。二级索引物理页读入内存时会根据该标记位判断是否需要进行ibuf merge操作 |
| IBUF\_BITMAP\_IBUF | 1bit | 表示该数据页是否是ibuf btree的一部分，该标记位主要用于异步AIO读操作。InnoDB专门为change buffer模块分配了一个后台AIO线程，如果page属于change buffer的b树，则使用该线程做异步读，参考函数：ibuf\_page\_low |
|  |  |  |

## Page 1, IBUF BITMAP PAGE

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38字节 |  |
|  |  |  |
| IBUF\_BITMAP | 4bits |  |
| IBUF\_BITMAP | 4bits |  |
| IBUF\_BITMAP | 4bits |  |
| IBUF\_BITMAP | 4bits |  |
| …….总共16384个ibuf\_bitmap,占8192字节，描述16384个page | | |
|  |  |  |
|  |  |  |
|  |  |  |

## Inode entry

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FSEG\_ID | 8字节 | 该Inode归属的Segment ID，若值为0表示该slot未被使用 |
| FSEG\_NOT\_FULL\_N\_USED | 4字节 | number of used segment pages in the FSEG\_NOT\_FULL list |
| FSEG\_FREE | 16字节 | list of free extents of this segment |
| FSEG\_NOT\_FULL | 16字节 | list of partially free extents |
| FSEG\_FULL | 16字节 | list of full extents |
| FSEG\_MAGIC\_N | 4字节 |  |
| FSEG\_FRAG\_ARR 0 | 4字节 | 属于该Segment的独立Page。总是先从全局分配独立的Page，当填满32个数组项时，就在每次分配时都分配一个完整的Extent，并在XDES PAGE中将其Segment ID设置为当前值 |
| FSEG\_FRAG\_ARR 1 | 4字节 | 这个数组是用来管理从表空间直接分配的页的页号 |
| FSEG\_FRAG\_ARR 2 | 4字节 |  |
| FSEG\_FRAG\_ARR 3 | 4字节 |  |
| FSEG\_FRAG\_ARR 4 | 4字节 |  |
| FSEG\_FRAG\_ARR 5 | 4字节 |  |
| FSEG\_FRAG\_ARR 6 | 4字节 |  |
| FSEG\_FRAG\_ARR 7 | 4字节 |  |
| FSEG\_FRAG\_ARR 8 | 4字节 |  |
| FSEG\_FRAG\_ARR 9 | 4字节 |  |
| FSEG\_FRAG\_ARR 10 | 4字节 |  |
| FSEG\_FRAG\_ARR 11 | 4字节 |  |
| FSEG\_FRAG\_ARR 12 | 4字节 |  |
| FSEG\_FRAG\_ARR 13 | 4字节 |  |
| FSEG\_FRAG\_ARR 14 | 4字节 |  |
| FSEG\_FRAG\_ARR 15 | 4字节 |  |
| FSEG\_FRAG\_ARR 16 | 4字节 |  |
| FSEG\_FRAG\_ARR 17 | 4字节 |  |
| FSEG\_FRAG\_ARR 18 | 4字节 |  |
| FSEG\_FRAG\_ARR 19 | 4字节 |  |
| FSEG\_FRAG\_ARR 20 | 4字节 |  |
| FSEG\_FRAG\_ARR 21 | 4字节 |  |
| FSEG\_FRAG\_ARR 22 | 4字节 |  |
| FSEG\_FRAG\_ARR 23 | 4字节 |  |
| FSEG\_FRAG\_ARR 24 | 4字节 |  |
| FSEG\_FRAG\_ARR 25 | 4字节 |  |
| FSEG\_FRAG\_ARR 26 | 4字节 |  |
| FSEG\_FRAG\_ARR 27 | 4字节 |  |
| FSEG\_FRAG\_ARR 28 | 4字节 |  |
| FSEG\_FRAG\_ARR 29 | 4字节 |  |
| FSEG\_FRAG\_ARR 30 | 4字节 |  |
| FSEG\_FRAG\_ARR 31 | 4字节 |  |
| 总共192字节 |  |  |

## Page 2 ,Inode page

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38字节 |  |
| FSEG\_INODE\_PAGE\_NODE | 12字节 | 只有prev和next |
| Inode entry 0 | 192字节 |  |
| Inode entry 1 | 192字节 |  |
| Inode entry 2 | 192字节 |  |
| Inode entry 3 | 192字节 |  |
| Inode entry 4 | 192字节 |  |
| Inode entry 5 | 192字节 |  |
| Inode entry …… | 192字节 |  |
| Inode entry 84 | 192字节 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Segment header结构

为了找到segment inode，需要有segment header来存储某个segment inode的信息，segment header 指向了segment inode，segment header 的结构如下

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 说明 |
| FSEG\_HDR\_SPACE | 4 | Segment inode 页所在的space id |
| FSEG\_HDR\_PAGE\_NO | 4 | Segment inode 页所在表空间的偏移量 |
| FSEG\_HDR\_OFFSET | 2 | Segment inode 在页中的偏移量 |
|  |  |  |

对用户表来说，segment header总是保存在其索引的root页中，被指向了非节点的segment inode和非叶子节点的segment inode。然而segment header 并不是总保存在root页中，可以放在一个单独的页中，比如insert buffer的处理方式。

函数fseg\_create\_general用来创建一个段，初始化segment inode以及segment header。函数变量page表示segment header的位置。如果page==NULL，则表示segment header在创建的段中，更准确的说，是在段的碎片页中，例如insert buffer的方式。

## IBUF\_TREE\_SEG\_HEADER

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FSEG\_HDR\_SPACE | 4 |  |
| FSEG\_HDR\_PAGE\_NO | 4 |  |
| FSEG\_HDR\_OFFSET | 2 |  |

## Page 3, IBUF HEAD page(insert buffer header页)

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38字节 |  |
| INDEX\_HEADER | 36字节 |  |
| 这里有一个10字节FSEG | 10 |  |
| 这里还有一个10字节FSEG |  |  |
| IBUF\_TREE\_SEG\_HEADER |  | 这里放的是相应inode的文件地址<space, page, offset> |
|  |  |  |

## Page 4, IBUF ROOT page(insert buffer root页)

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38字节 |  |
| INDEX\_HEADER | 36字节 |  |
| PAGE\_BTR\_IBUF\_FREE\_LIST  (PAGE\_BTR\_SEG\_LEAF) | 16字节 | Insert buffer B+树中使用的page从这个链表中分配,  这个和之前的索引树的段头信息是不一样的。 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Insert buffer B+树的索引id(PAGE\_INDEX\_ID)总是从ff ff ff ff 00 00 00 00 开始的。

**INSERT BUFFER PAGE**

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 说明 |
| FIL\_HEADER | 38 |  |
| PAGE\_BTR\_IBUF\_FREE\_LIST\_NODE | 12 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Page 5, TRX\_SYS PAGE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段 | 偏移 | | 长度 | 备注 |
| FIL\_HEADER |  | | 38 | PAGE\_TYPE = FIL\_PAGE\_TYPE\_TRX\_SYS |
| TRX\_SYS\_TRX\_ID\_STORE |  | | 8 | 每次写入的都是  trx\_sys->max\_trx\_id |
| TRX\_SYS\_FSEG\_HEADER |  | | 10 |  |
| TRX\_SYS\_RSEGS(128)个rollback segments   |  |  | | --- | --- | | RSEG\_SPACE(space\_id值) | RSEG\_PAGE\_NO(page\_no值) | | RSEG\_SPACE | RSEG\_PAGE\_NO | | RSEG\_SPACE | RSEG\_PAGE\_NO | |  | | 128 \* 8 | 实际上只使用一个rollback segment |
|  |  | |  |  |
|  |  | |  |  |
|  | 以下是Page内UNIV\_PAGE\_SIZE - 2000的偏移位置  TRX\_SYS\_MYSQL\_MASTER\_LOG\_INFO | | | |
| TRX\_SYS\_MYSQL\_LOG\_MAGIC\_N\_FLD |  | 4 | | Magic Num ，值为873422344 |
| TRX\_SYS\_MYSQL\_LOG\_OFFSET\_HIGH |  | 4 | | 事务提交时会将其binlog位点更新到该page中，这里记录了在binlog文件中偏移量的高位的4字节 |
| TRX\_SYS\_MYSQL\_LOG\_OFFSET\_LOW |  | 4 | | 同上，记录偏移量的低4位字节 |
| TRX\_SYS\_MYSQL\_LOG\_NAME |  |  | | 记录所在的binlog文件名 |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  | | | |
|  | 以下是Page内UNIV\_PAGE\_SIZE - 1000的偏移位置  (TRX\_SYS\_MYSQL\_LOG\_INFO) | | | |
| TRX\_SYS\_MYSQL\_LOG\_MAGIC\_N\_FLD |  | 4 | | Magic Num ，值为873422344 |
| TRX\_SYS\_MYSQL\_LOG\_OFFSET\_HIGH |  | 4 | | 事务提交时会将其binlog位点更新到该page中，这里记录了在binlog文件中偏移量的高位的4字节 |
| TRX\_SYS\_MYSQL\_LOG\_OFFSET\_LOW |  | 4 | | 同上，记录偏移量的低4位字节 |
| TRX\_SYS\_MYSQL\_LOG\_NAME |  |  | | 记录所在的binlog文件名 |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  | | | |
|  |  | | | |
|  | 以下是Page内UNIV\_PAGE\_SIZE - 200 的偏移位置 | | | |
| TRX\_SYS\_DOUBLEWRITE\_FSEG |  | 10 | |  |
| TRX\_SYS\_DOUBLEWRITE\_MAGIC |  | 4 | |  |
| TRX\_SYS\_DOUBLEWRITE\_BLOCK1 |  | 4 | |  |
| TRX\_SYS\_DOUBLEWRITE\_BLOCK2 |  | 4 | |  |
| TRX\_SYS\_DOUBLEWRITE\_MAGIC  (TRX\_SYS\_DOUBLEWRITE\_REPEAT) |  | 4 | |  |
|  |  |  | |  |
|  |  |  | |  |
| TRX\_SYS\_DOUBLEWRITE\_SPACE\_ID\_STORED |  | 4 | |  |
|  |  |  | |  |
|  |  |  | |  |
| TRX\_SYS\_FILE\_FORMAT\_TAG | UNIV\_PAGE\_SIZE-16 | 8 | |  |
|  |  |  | |  |

## Page 6, FSP\_FIRST\_RSEG\_PAGE(rollback segment header)

这个页在trx\_sysf\_create()->trx\_rseg\_header\_create()中创建

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38 |  |
| TRX\_RSEG\_MAX\_SIZE  (TRX\_RSEG) | 4 |  |
| TRX\_RSEG\_HISTORY\_SIZE | 4 |  |
| TRX\_RSEG\_HISTORY | 16 | 根据事务的提交顺序存放undo日志，当进行purge操作时，首先读取该链表尾端的undo日志，判断是否可以回收该空间 |
| TRX\_RSEG\_FSEG\_HEADER |  |  |
| UNDO\_SLOT |  | 指向UNDO LOG PAGE |
| UNDO\_SLOT |  |  |
| UNDO\_SLOT |  |  |
| UNDO\_SLOT |  |  |
| 一共1024个 |  |  |
|  |  |  |
|  |  |  |

## 索引页的page格式

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER | 38 |  |
| INDEX\_HEADER | 56 |  |
| Infimum记录 |  |  |
| supremum记录 |  |  |
| 用户记录数据 1 |  |  |
| 用户记录数据 2 |  |  |
| 用户记录数据 3 |  |  |
|  |  |  |
|  |  |  |

## 物理记录格式

## Compact格式

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 非NULL变长字段长度列表 | NULL标志位 | 记录头信息(40个位) 5字节实际上是6字节，还有一个字节空着，具体的看解析rec的函数rec\_init\_offsets | Rowid | Transaction id | Rollback ptr | 列1数据 |
|  |  | |  |  |  | | --- | --- | --- | | 名称 | 大小(bit) | 描述 | | () | 1 | 未知 | | () | 1 | 未知 | | deleted\_flag | 1 | 改行是否已被删除 | | min\_rec\_flag | 1 | 为1，如果该记录是预先被定义为最小记录 | | n\_owned | 4 | 该记录拥有的记录数,page directory中一个slot对应一个记录，这个记录负责管理n\_owned个记录的次序，每个slot有4-8条记录  第一个slot只有一个记录，即infimum本身，  最后一个slot到是可以包含1-8个记录 | | heap\_no | 13 | 索引堆中的该条记录的排序记录 | | record\_type | 3 | 索引记录了类型  000=普通  001=索引节点指针  010=infimum  0  011=supermum，1xx表示保留 | | next\_record | 16 | 页中下一条记录的相对位置 | |  |  |  | |  |  |  |  |

Compact行格式的首部是一个非NULL变长字段长度列表，而且是按照列的顺序逆序放置的。当列的长度小于255字节，用1字节表示，若大于255个字节，用2个字节表示，变长字段的长度最大不可以超过2个字节（这也很好地解释了为什么MySQL中varchar的最大长度为65 535，因为2个字节为16位，即216=1=65 535）。第二个部分是NULL标志位，该位指示了该行数据中是否有NULL值，用1表示。该部分所占的字节应该为bytes。接下去的部分是为记录头信息（record header），固定占用5个字节（40位），每位的含义见下表4-1。最后的部分就是实际存储的每个列的数据了，需要特别注意的是，NULL不占该部分任何数据，即NULL除了占有NULL标志位，实际存储不占有任何空间。另外有一点需要注意的是，每行数据除了用户定义的列外，还有两个隐藏列，事务ID列和回滚指针列，分别为6个字节和7个字节的大小。若InnoDB表没有定义Primary Key，每行还会增加一个6字节的RowID列。

create table mytest (

　　t1 varchar(10),

　　t2 varchar(10),

　　t3 char(10),

　　t4 varchar(10)

) engine=innodb charset=latin1 row\_format=compact;

insert into mytest values('a','bb','bb','ccc');

insert into mytest values('d','ee','ee','fff');

insert into mytest values('d',NULL,NULL,'fff');

0000c070 73 75 70 72 65 6d 75 6d 03 02 01 00 00 00 10 00|supremum……

0000c080 2c 00 00 00 2b 68 00 00 00 00 00 06 05 80 00 00|，……+h……

0000c090 00 32 01 10 61 62 62 62 62 20 20 20 20 20 20 20|.2..abbbb

0000c0a0 20 63 63 63 03 02 01 00 00 00 18 00 2b 00 00 00|ccc……+……

0000c0b0 2b 68 01 00 00 00 00 06 06 80 00 00 00 32 01 10|+h……2..

0000c0c0 64 65 65 65 65 20 20 20 20 20 20 20 20 66 66 66|deeeefff

0000c0d0 03 01 06 00 00 20 ff 98 00 00 00 2b 68 02 00 00|……+h……

0000c0e0 00 00 06 07 80 00 00 00 32 01 10 64 66 66 66 00|……2..dfff.

该行记录从0000c078开始，若整理如下，相信你会有更好的理解：

03 02 01 /\*变长字段长度列表，逆序\*/

00 /\*NULL标志位，第一行没有NULL值\*/

00 00 10 00 2c /\*记录头信息，固定5字节长度\*/

00 00 00 2b 68 00 /\*RowID我们建的表没有主键，因此会有RowID\*/

00 00 00 00 06 05 /\*TransactionID\*/

80 00 00 00 32 01 10 /\*Roll Pointer\*/

61 /\*列1数据'a'\*/

62 62 /\*列2'bb'\*/

62 62 20 20 20 20 20 20 20 20 /\*列3数据'bb'\*/

63 63 63 /\*列4数据'ccc'\*/

现在第一行数据就展现在我们眼前了。需要注意的是，变长字段长度列表是逆序存放的，03 02 01，而不是01 02 03。还需要注意的是InnoDB每行有隐藏列。同时可以看到，固定长度char字段在未填充满其长度时，会用0x20来进行填充。再来分析一下，记录头信息的最后4个字节代表next\_recorder，0x6800代表下一个记录的偏移量，当前记录的位置+0x6800就是下一条记录的起始位置。所以InnoDB存储引擎在页内部是通过一种链表的结构来串联各个行记录的。

现在我们关注有NULL值的第三行：

03 01/\*变长字段长度列表，逆序\*/

06 =(0000 0110) /\*NULL标志位，第2,3列有NULL值\*/

00 00 20 ff 98/\*记录头信息\*/

00 00 00 2b 68 02/\*RowID\*/

00 00 00 00 06 07/\*TransactionID\*/

80 00 00 00 32 01 10/\*Roll Pointer\*/

64/\*列1数据'd'\*/

66 66 66/\*列4数据'fff'\*/

第三行有NULL值，因此NULL标志位不再是00而是06了，转换成二进制为00000110，为1的值即代表了第2列和第3列的数据为NULL，在其后存储列数据的部分，我们会发现没有存储NULL，只存储了第1列和第4列非NULL的值。这个例子很好地说明了：不管是char还是varchar类型，NULL值是不占用存储空间的。

## 大记录格式

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Col offset list | Record header | Col1 | Col2 |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 127 byte prefix | BTR\_EXTERN\_SPACE\_ID | BTR\_EXTERN\_PAGE\_NO | BTR\_EXTERN\_OFFSET | BTR\_EXTERN\_LEN |

|  |
| --- |
| FIL\_HEADER |
| PAGE\_HEADER |
| BTR\_BLOB\_HDR\_PART\_LEN |
| BTR\_BLOB\_HDR\_NEXT\_PAGE\_NO |
| 数据 |
| FIL\_END |

|  |
| --- |
| FIL\_HEADER |
| PAGE\_HEADER |
| BTR\_BLOB\_HDR\_PART\_LEN |
| BTR\_BLOB\_HDR\_NEXT\_PAGE\_NO |
| 数据 |
| FIL\_END |

## Redundant格式

|  |  |  |  |
| --- | --- | --- | --- |
| 字段长度偏移列表 | 记录头信息 (6字节) | 列1数据 | 列2数据 |
|  | |  |  |  | | --- | --- | --- | | 名称 | 大小(位) | 描述 | | () | 1 | 未知 | | () | 1 | 未知 | | delete\_flag | 1 | 记录是否已经被删除 | | min\_rec\_flag | 1 |  | | n\_owned | 4 | 该记录拥有的记录数 | | heap\_no | 13 | 索引堆中该记录的排序记录 | | n\_fields | 10 | 记录中列的数量 | | 1byte\_offs\_flag  (short flag) | 1 | 偏移列表为1个字节还是2个字节 | | next\_record | 16 | 页中下一条记录的相对位置 | |  |  |

## Compressed与Dynamic行记录格式

# Undo log 结构

Undo page 可以分为以下几个部分：

1. FIL\_HEADER
2. UNDO LOG PAGE HEADER
3. UNDO LOG SEGMENT HEADER
4. UNDO 日志数据

其中的UNDO LOG SEGMENT HEADER 只保存在undo 段的第一个undo页中，所以undo页的格式可以分为以下两种：

先列出UNDO LOG PAGE HEADER、UNDO LOG HEADER 和UNDO LOG SEGMENT HEADER的格式

## UNDO LOG PAGE HEADER

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| TRX\_UNDO\_PAGE\_TYPE | 2 | Undo日志的类型，有效值为   * TRX\_UNDO\_INSERT * TRX\_UNDO\_UPDATE(update 和delete都用这个) |
| TRX\_UNDO\_PAGE\_START | 2 | Undo页中最新一个事务undo日志所在的位置 |
| TRX\_UNDO\_PAGE\_FREE | 2 | Undo页中空闲空间的偏移量 |
| TRX\_UNDO\_PAGE\_NODE | 12 | Undo页的链表 |
|  |  |  |
|  |  |  |

## UNDO LOG SEGMENT HEADER的格式

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 说明 |
| TRX\_UNDO\_STATE | 2 | UNDO段的状态，有效值为：   * TRX\_UNDO\_ACTIVE * TRX\_UNDO\_CACHED * TRX\_UNDO\_TO\_FREE * TRX\_UNDO\_TO\_PURGE * TRX\_UNDO\_PREPARED |
| TRX\_UNDO\_LAST\_LOG | 2 | 最近一个undo log header在页中的偏移量位置 |
| TRX\_UNDO\_FSEG\_HEADER | 10 | Undo 段的segment header |
| TRX\_UNDO\_PAGE\_LIST | 16 | Undo页的链表头(指向 UNDO LOG PAGE HEADER中的 TRX\_UNDO\_PAGE\_NODE字段) |
|  |  |  |
|  |  |  |
|  |  |  |

## UNDO LOG HEADER

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 说明 |
| TRX\_UNDO\_TRX\_ID | 8 | 产生undo日志的事务id |
| TRX\_UNDO\_TRX\_NO | 8 | 将undo记录放入HISTORY 链表时，事务的no仅在update undo 日志中存在 |
| TRX\_UNDO\_DEL\_MARKS | 2 | TRUE若有delete mark操作，后续需要被purge线程清理 |
| TRX\_UNDO\_LOG\_START | 2 | Undo日志body的开始位置，由于purge是从第一个undo日志body开始，因此该值不一定等于undo log header结束位置的偏移量 |
| TRX\_UNDO\_XID\_EXISTS | 1 |  |
| TRX\_UNDO\_DICT\_TRANS | 1 |  |
| TRX\_UNDO\_TABLE\_ID | 8 | 若是DDL操作，操作表的id |
| TRX\_UNDO\_NEXT\_LOG | 2 | 下一个 undo log header的偏移 |
| TRX\_UNDO\_PREV\_LOG | 2 | 前一个 undo log header的偏移 |
| TRX\_UNDO\_HISTORY\_NODE | 12 | HISTORY链表 |
| TRX\_UNDO\_LOG\_OLD\_HDR\_SIZE | 不占空间 | 没有XA Transaction是，undo log header到此就结束了 |
| TRX\_UNDO\_XA\_FORMAT | 4 |  |
| TRX\_UNDO\_XA\_TRID\_LEN | 4 |  |
| TRX\_UNDO\_XA\_BQUAL\_LEN | 4 |  |
| TRX\_UNDO\_XA\_XID | 128 |  |
|  |  |  |
|  |  |  |

## 首个Undo log page

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER |  |  |
| UNDO LOG PAGE HEADER |  |  |
| UNDO LOG SEGMENT HEADER |  |  |
| UNDO LOG |  |  |
| UNDO LOG |  |  |
| UNDO LOG |  |  |
|  |  |  |

## 普通Undo log page

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| FIL\_HEADER |  |  |
| UNDO LOG PAGE HEADER |  |  |
| UNDO LOG |  |  |
| UNDO LOG |  |  |
| UNDO LOG |  |  |
|  |  |  |

# Redo log 结构

Redo log 恢复的大概调用图

recv\_recovery\_from\_checkpoint\_start

recv\_find\_max\_checkpoint (从redo log file中查找最大的那个checkpoint值 )

log\_group\_read\_checkpoint\_info

fil\_io

log\_group\_read\_checkpoint\_info

recv\_group\_scan\_log\_recs

log\_group\_read\_log\_seg

recv\_scan\_log\_recs

recv\_parse\_or\_apply\_log\_rec\_body

recv\_sys\_add\_to\_parsing\_buf

recv\_parse\_log\_recs

recv\_parse\_log\_rec

recv\_parse\_or\_apply\_log\_rec\_body ( 根据日志类型不同调用不同的函数进行恢复 )

fil\_op\_log\_parse\_or\_replay

recv\_add\_to\_hash\_table(将由recv\_parse\_or\_apply\_log\_rec\_body分析得到的重做日志加入到哈希表中)

调用recv\_recovery\_from\_checkpoint\_start后，会去查找检查点值。之前已经描述过，Innodb存储引擎保存两个检查点值，因此需要通过调用函数recv\_find\_max\_checkpoint找到最大的那个值。在得到checkpoint值后，通过函数recv\_recovery\_from\_checkpoint\_start传入的参数min\_flushed\_lsn和max\_flushed\_lsn就能得知是否需要进行恢复操作了，恢复的区间应为(checkpoint, last\_redo\_lsn )。在恢复上述区间日志的数据时，其通过小批量的方式读取重做日志文件中的log block到log\_sys->buf中。每次读取的量为64KB，即64KB/512=128个重做日志块，在innodb存储引擎中这是由宏RECV\_SCAN\_SIZE (4 \*16K )定义的。

在读取重做日志的日志块后，innodb存储引擎会对其进行分析，判断重做日志块中的日志是否包含上一个日志的内容。这时因为一个日志可能存放于多个重做日志块中，因此在恢复时需要对重做日志进行判断，将日志复制到recv\_sys->buf中。在将日志复制到recv\_sys->buf中后，就可对日志进行分析，之后根据日志对应的<space, offset>的哈希值插入到recv\_sys->addr\_hash中去，然后根据hash表中的日志进行恢复。

## Redo log 日志格式

必须有的

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Space | Page\_no | body |

MLOG\_REC\_DELETE格式的(就是删除操作对应的redo log)

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Space | Page\_no | offset |

MLOG\_REC\_INSERT格式的(就是insert操作对应的redo log)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type | Space | Page\_no | Cur\_rec\_offset | Len&extra\_info\_flag | Info\_bits | Origin\_offset | Mis\_match\_index | Original rec body |

有些mtr涉及多个page的操作，因此innodb在type列上用MLOG\_SINGLE\_REC\_FLAG标志来表示该mtr是否仅及1个页的操作。若一个mtr需要同时维护多个页之间的数据一致性，那么其在mtr结束时会额外写入一个字节大小的MLOG\_MULTI\_REC\_END信息，表示该mtr记录了多个页的日志。

## 普通的redo log block(512字节) header的组成

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 说明 |
| LOG\_BLOCK\_HDR\_NO | 4 | Log buffer 由log block组成，在内部log buffer 就好似一个数组，因此LOG\_BLOCK\_HDR\_NO用来标记这个数组中的位置。它是递归并且循环使用的，其占用4个字节，但是由于第一位用来判断是否是flush bit，所以最大的值为2GB |
| LOG\_BLOCK\_HDR\_DATA\_LEN | 2 | 表示log block所占用的大小。当log block 被写满时，该值为0x200，表示使用全部log block 空间，即占用512个字节 |
| LOG\_BLOCK\_FIRST\_REC\_GROUP | 2 | 表示log block中第一个日志所在的偏移量，若该值与LOG\_BLOCK\_HDR\_DATA\_LEN的值相同，说明当前log block不含有新的日志。 |
| LOG\_BLOCK\_CHECK\_POINT\_NO | 4 | 表示该log block 最后被写入时的log\_sys->next\_checkpoint\_no低4个字节的值 |
|  |  |  |
|  |  |  |
|  |  |  |

## redo log 文件的第0个 page(512字节)[ log file header]

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 说明 |
| LOG\_GROUP\_ID | 4 | Redo log group的ID号 |
| LOG\_FILE\_START\_LSN | 8 | 表示每个redo log的第一个日志的LSN |

## redo log 文件的第1个 page(512字节)

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| LOG\_CHECKPOINT\_NO | 8 | 单调递增的值，每次checkpoint操作完成后进行自增操作 |
| LOG\_CHECKPOINT\_LSN | 8 | Checkpoint值，即LSN小于等于该值的页都已经被写入到磁盘，若需要恢复，则只需要恢复LOG\_CHECKPOINT\_LSN后面的redo log。  由于checkpoint block有两个，并交替进行checkpoint值的更新。因此在恢复时，需要读取这两个块，判断各自的LOG\_CHECKPOINT\_LSN值，并且仅需恢复LSN大于最大LOG\_CHECKPOINT\_LSN的重做日志，这个过程由recv\_find\_max\_checkpoint实现 |
| LOG\_CHECKPOINT\_OFFSET\_LOW32 | 4 | Checkpoint值对应的在redo log中的偏移量 |
| LOG\_CHECKPOINT\_LOG\_BUF\_SIZE | 4 | 记录checkpoint时，log buffer的大小，该值没什么用处 |
| LOG\_CHECKPOINT\_ARCHIVED\_LSN | 8 | 记录checkpoint时，归档日志已经归档到的lsn值 |
| LOG\_CHECKPOINT\_GROUP\_ARRAY | 256 | 归档redo log日志的信息 |
|  |  |  |
| LOG\_CHECKPOINT\_CHECKSUM\_1 | 4 | 记录checksum值，该部分对前32字节进行fold操作 |
| LOG\_CHECKPOINT\_CHECKSUM\_2 | 4 | 对从LOG\_CHECKPOINT\_LSN部分开始的284个字节进行fold |

## redo log 文件的第2个 page(512字节)

redo log 文件的第二个512字节是空白

## redo log 文件的第3个 page(512字节)

|  |  |  |
| --- | --- | --- |
| 字段 | 长度 | 备注 |
| LOG\_CHECKPOINT\_NO | 8 | 单调递增的值，每次checkpoint操作完成后进行自增操作 |
| LOG\_CHECKPOINT\_LSN | 8 | Checkpoint值，即LSN小于等于该值的页都已经被写入到磁盘，若需要恢复，则只需要恢复LOG\_CHECKPOINT\_LSN后面的redo log。  由于checkpoint block有两个，并交替进行checkpoint值的更新。因此在恢复时，需要读取这两个块，判断各自的LOG\_CHECKPOINT\_LSN值，并且仅需恢复LSN大于最大LOG\_CHECKPOINT\_LSN的重做日志，这个过程由recv\_find\_max\_checkpoint实现 |
| LOG\_CHECKPOINT\_OFFSET\_LOW32 | 4 | Checkpoint值对应的在redo log中的偏移量 |
| LOG\_CHECKPOINT\_LOG\_BUF\_SIZE | 4 | 记录checkpoint时，log buffer的大小，该值没什么用处 |
| LOG\_CHECKPOINT\_ARCHIVED\_LSN | 8 | 记录checkpoint时，归档日志已经归档到的lsn值 |
| LOG\_CHECKPOINT\_GROUP\_ARRAY | 256 | 归档redo log日志的信息 |
|  |  |  |
| LOG\_CHECKPOINT\_CHECKSUM\_1 | 4 | 记录checksum值，该部分对前32字节进行fold操作 |
| LOG\_CHECKPOINT\_CHECKSUM\_2 | 4 | 对从LOG\_CHECKPOINT\_LSN部分开始的284个字节进行fold |

## Redo log 类型

|  |  |  |
| --- | --- | --- |
| 字段 | 值 | 备注 |
| MLOG\_SINGLE\_REC\_FLAG | 128 | if the mtr contains only one log record for one page,  i.e., write\_initial\_log\_record has been called only once,  this flag is ORed to the type of that first log record |
| MLOG\_1BYTE | 1 | one byte is written |
| MLOG\_2BYTES | 2 | 2 bytes |
| MLOG\_4BYTESs | 4 | 4 bytes |
| MLOG\_8BYTES | 8 | 8 bytes |
| MLOG\_REC\_INSERT | 9 | record insert |
| MLOG\_REC\_CLUST\_DELETE\_MARK | 10 | mark clustered index record deleted |
| MLOG\_REC\_SEC\_DELETE\_MARK | 11 | mark secondary index record deleted |
| MLOG\_REC\_UPDATE\_IN\_PLACE | 13 | update of a record, preserves record field sizes |
| MLOG\_REC\_DELETE | 14 | delete a record from a page |
| MLOG\_LIST\_END\_DELETE | 15 | delete record list end on index page |
| MLOG\_LIST\_START\_DELETE | 16 | delete record list start on index page |
| MLOG\_LIST\_END\_COPY\_CREATED | 17 | copy record list end to a new created index page |
| MLOG\_PAGE\_REORGANIZE | 18 | reorganize an index page in ROW\_FORMAT=REDUNDANT |
| MLOG\_PAGE\_CREATE | 19 | create an index page |
| MLOG\_UNDO\_INSERT | 20 | insert entry in an undo log |
| MLOG\_UNDO\_ERASE\_END | 21 | erase an undo log page end |
| MLOG\_UNDO\_INIT | 22 | initialize a page in an undo log |
| MLOG\_UNDO\_HDR\_DISCARD | 23 | discard an update undo log header |
| MLOG\_UNDO\_HDR\_REUSE | 24 | reuse an insert undo log header |
| MLOG\_UNDO\_HDR\_CREATE | 25 | create an undo log header |
| MLOG\_REC\_MIN\_MARK | 26 | mark an index record as the predefined minimum record |
| MLOG\_IBUF\_BITMAP\_INIT | 27 | initialize an ibuf bitmap page |
| MLOG\_LSN | 28 | current LSN |
| MLOG\_INIT\_FILE\_PAGE | 29 | this means that a file page is taken into use and the prior contents of the page should be ignored: in recovery we must not trust the lsn values stored to the file page |
| MLOG\_WRITE\_STRING | 30 | write a string to a page |
| MLOG\_MULTI\_REC\_END | 31 | if a single mtr writes several log records, this log record ends the sequence of these records |
| MLOG\_DUMMY\_RECORD | 32 | dummy log record used to pad a log block full |
| MLOG\_FILE\_CREATE | 33 | log record about an .ibd file creation |
| MLOG\_FILE\_RENAME | 34 | log record about an .ibd file rename |
| MLOG\_FILE\_DELETE | 35 | log record about an .ibd ile deletion |
| MLOG\_COMP\_REC\_MIN\_MARK | 36 | mark a compact index record as the predefined minimum record |
| MLOG\_COMP\_PAGE\_CREATE | 37 | create a compact index page |
| MLOG\_COMP\_REC\_INSERT | 38 | compact record insert |
| MLOG\_COMP\_REC\_CLUST\_DELETE\_MARK | 39 | mark compact clustered index record deleted |
| MLOG\_COMP\_REC\_SEC\_DELETE\_MARK | 40 | mark compact secondary index record deleted; this log record type is redundant, as MLOG\_REC\_SEC\_DELETE\_MARK is independent of the record format. |
| MLOG\_COMP\_REC\_UPDATE\_IN\_PLACE | 41 | update of a compact record, preserves record field sizes |
| MLOG\_COMP\_REC\_DELETE | 42 | delete a compact record from a page |
| MLOG\_COMP\_LIST\_END\_DELETE | 43 | delete compact record list end on index page |
| MLOG\_COMP\_LIST\_START\_DELETE | 44 | delete compact record list start on index page |
| MLOG\_COMP\_LIST\_END\_COPY\_CREATED | 45 | copy compact record list end to a new created index page |
| MLOG\_COMP\_PAGE\_REORGANIZE | 46 | reorganize an index page |
| MLOG\_FILE\_CREATE2 | 47 | log record about creating an .ibd file, with format |
| MLOG\_ZIP\_WRITE\_NODE\_PTR | 48 | write the node pointer of a record on a compressed non-leaf B-tree page |
| MLOG\_ZIP\_WRITE\_BLOB\_PTR | 49 | write the BLOB pointer of an externally stored column on a compressed page |
| MLOG\_ZIP\_WRITE\_HEADER | 50 | write to compressed page header |
| MLOG\_ZIP\_PAGE\_COMPRESS | 51 | compress an index page |
| MLOG\_ZIP\_PAGE\_COMPRESS\_NO\_DATA | 52 | compress an index page without logging it's image |
| MLOG\_ZIP\_PAGE\_REORGANIZE | 53 | reorganize a compressed page |
| MLOG\_BIGGEST\_TYPE | 53 |  |

## Undo log 整体结构图

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRX\_SYS\_PAGE\_NO (5)   |  | | --- | | TRX\_SYS\_TRX\_ID\_STORE | | TRX\_SYS\_FSEG\_HEADER | | |  |  | | --- | --- | | RSEG\_SPACE | RSEG\_PAGE\_NO | | RSEG\_SPACE | RSEG\_PAGE\_NO | | RSEG\_SPACE | RSEG\_PAGE\_NO | | RSEG\_SPACE | RSEG\_PAGE\_NO | | RSEG\_SPACE | RSEG\_PAGE\_NO | | | FSP\_FIRST\_RSEG\_PAGE(6)   |  | | --- | | TRX\_RSEG\_MAX\_SIZE | | TRX\_RSEG\_HISTORY\_SIZE | | TRX\_RSEG\_HISTORY | | TRX\_RSEG\_FSEG\_HEADER | | UNDO\_SLOT | | UNDO\_SLOT | | UNDO\_SLOT | | UNDO\_SLOT | |
| UNDO PAGE   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | FIL\_HEADER | |  | | --- | | TRX\_UNDO\_PAGE\_TYPE | | TRX\_UNDO\_PAGE\_START | | TRX\_UNDO\_PAGE\_FREE | | TRX\_UNDO\_PAGE\_NODE | |  | | |  | | --- | | UNDO LOG HEADER | |  | | UNDO LOG RECORD | |  | | UNDO LOG  RECORD | | |  | | --- | | TRX\_UNDO\_TRX\_ID | | TRX\_UNDO\_TRX\_NO | | TRX\_UNDO\_DEL\_MARKS | | TRX\_UNDO\_LOG\_START | | TRX\_UNDO\_DICT\_OPERATION | | TRX\_UNDO\_TABLE\_ID | | TRX\_UNDO\_NEXT\_LOG | | TRX\_UNDO\_PREV\_LOG | | TRX\_UNDO\_HISTORY\_NODE | |  | | | UNDO LOG PAGE HEADER | | UNDO LOG SEGMENT HEADER | | UNDO LOG | |  | | --- | | TRX\_UNDO\_STATE | | TRX\_UNDO\_LAST\_LOG | | TRX\_UNDO\_FSEG\_HEADER | | TRX\_UNDO\_PAGE\_LIST | | | UNDO LOG | | UNDO LOG | | UNDO LOG | | |

# 锁

record lock: 单个索引上的记录

gap lock: 间隙锁，锁定一个范围，但不包含记录本身

next-key lock: gap lock + record lock,锁定一个范围，但是需要加上记录本身。

行锁实际上一个索引记录锁。

根据锁的实现方式不同，可以分为implicit lock 和 explicit lock

## lock\_sys\_t , lock\_t, lock\_rec\_t ,trx\_t，dict\_table\_t 五者的关系

## 关于锁 record的几个重要函数

1. lock\_rec\_lock(ibool impl,

ulint mode,

const buf\_block\_t\* block,

ulint heap\_no,

dict\_index\_t\* index,

que\_thr\_t\* thr)

1. lock\_rec\_lock\_fast(ibool impl,

ulint mode,

const buf\_block\_t\* block,

ulint heap\_no,

dict\_index\_t\* index,

que\_thr\_t\* thr)

1. lock\_rec\_lock\_slow(ibool impl,

ulint mode,

const buf\_block\_t\* block,

ulint heap\_no,

dict\_index\_t\* index,

que\_thr\_t\* thr)

## 隐式锁和显示锁

explicit lock 可以分为 gap explicit lock 和no gap explicit lock，这里的gap通过type\_mode中的LOCK\_GAP来进行设置。No gap explicit lock 锁住的是记录以及记录之前的范围。explicit lock 可以是s-lock也可以是x-lock，然而implicit lock 总是x-lock

explicit lock 是非常直接的，例如select \* from t where rowed=xxx for update,那就需要对这个记录进行加锁，锁的类型为x-lock,这个过程通过函数lock\_rec\_create实现

implicit lock 是指索引记录逻辑上有x-lock，但是实际在内存对象中并不含有这个锁信息，这意味着implicit lock没有任何的内存开销，从而进一步减少innodb的内存开销。行锁本质上索引记录锁，那么当锁定一行聚集索引记录时，若该记录上还有辅助索引，根据谓词锁的要求还应该对相应辅助索引上的记录进行加锁，在innodb中，implicit lock即可以存在clustered index 中也可以存在于secondary index中。

对于clustered index 记录，例如用户插入一个 row id =4的新记录，但是事务还未提交，这是row id=4 的记录就包含一个implicit lock。然而在全局的lock\_sys变量中查询不到此新记录的锁，因此这个锁是隐式的。

更改也是同样的状况，存在隐式的锁。但是并不是每次更改都是这样的，这取决于加锁是否需要等待，若需要等待则产生explicit lock.原因很简单，只有通过创建需要等待的explicit lock对象，那么当锁释放后才能进行唤醒操作。

由于implicit lock的存在，在对某记录进行加锁之前，可能需要调用函数lock\_rec\_convert\_impl\_to\_expl将implicit lock 转换成explicit lock，并将锁信息加入到全局变量lock\_sys的哈希表中去。

## Clustered index 记录的隐式锁

聚集索引记录中implicit lock判断较为简单，因为每个clustered index 记录都有一个事务id的隐藏列，

只需要通过该事务id判断当前是否为活跃事务就能得知是否有implicit lock。若通过事务id查询得到该事务为活跃事务，则此clustered index有implicit lock.反之，就不含有implicit lock。

## Secondary index 记录的隐式锁

辅助索引记录是不含有事务id隐藏列的，因此判断辅助索引记录是否有implicit lock就是得复杂的多。然而，每个辅助索引页通过page header的PAGE\_MAX\_TRX\_ID保存一个最大事务id，当辅助索引中的任何记录被更新后，都需要更新PAGE\_MAX\_TRX\_ID的值。因此辅助索引记录的implicit lock判断可以分为两个步骤进行：

1. 根据辅助索引页的PAGE\_MAX\_TRX\_ID值进行判断
2. 通过聚集索引记录进行判断

由于每次辅助索引记录的修改都需要更新页的PAGE\_MAX\_TRX\_ID值，因此若当前PAGE\_MAX\_TRX\_ID的值小于当前活跃事务的最小id时，则此辅助索引记录不含有implicit lock，即之前已经提交的事物修改了该记录。若PAGE\_MAX\_TRX\_ID大于等于当前活跃事务的最小id时，则存在以下这些可能性：

1. 存在某活跃事务，修改了辅助索引记录，从而导致PAGE\_MAX\_TRX\_ID的更新。
2. 存在事务(可能是活跃也可能已经完成提交操作)，修改了页中其他的辅助索引记录，从而导致PAGE\_MAX\_TRX\_ID的更新。

此时，需要通过辅助索引记录对应的聚集索引来判断是否含有implicit lock，这个过程通过row\_vers\_impl\_x\_locked实现

判断辅助索引记录是否持有implicit lock，只需要判断是否存在未提交的活跃事务对记录进行INSERT，DELETE,UPDATE的操作。若有，则必然持有implicit lock，并返回该活跃事务对象trx\_t

## Secondary index 记录通过clustered index 记录判断是否有implicit lock的思想

clust\_rec聚集索引记录

secondary rec 辅助索引记录

trx\_id

prev\_version

entry

prev\_trx\_id

通过辅助索引记录可以得到聚集索引记录clust\_rec,由于聚集索引记录有隐藏列事务id，因此还可以得到对应的trx\_id。此外，可以通过undo log来构造聚集索引记录之前版本的聚集索引记录prev\_version以及对应的辅助索引记录entry。有了上述信息之后，对于implicit lock的判断过程如下所示

1. 若trx\_id对应的事务为不活跃的事务，则辅助索引记录rec不含有implicit lock。
2. 若prev\_version为null，表示没有之前版本的记录，即是当前事务插入的记录，则辅助索引记录rec含有implicit lock。
3. 若rec==entry，两个版本的索引记录相等，但是两个记录的delete flag位不同，则表示某活跃事务删除了记录，因此辅助索引记录rec含有implicit lock。
4. 若rec!=entry，两个版本的辅助索引不相同，且记录rec的delete flag为0，表示某活跃事务更新了辅助索引记录，因此辅助索引记录含有implicit lock。
5. 若rec==entry并且两个记录的delete flag位是相同的，则既可能是当前某活跃事务修改了记录rec，也可能是之前已提交的事务修改了记录rec。若trx\_id!=prev\_id,则表示之前的事务已经修改了记录，因此记录rec上不含有implicit lock。否则，需要通过再之前的记录版本进行判断。
6. 若rec!=entry并且rec的delete flag位为1，则既可能是当前某活跃事务修改了记录rec，也可能是之前已提交的事务修改了记录rec。若trx\_id != prev\_id，则表示之前的事务已经修改了记录，因此记录rec上不含有implicit lock。否则，需要通过再之前的记录版本进行判断。

上述6中情况的6个demo

表结构

Create table t(

a int not null,

b int not null,

c int not null,

primary key(a),

key(b)

)type=innodb

Insert into t values (1, 2, 3);

情况1

若这时进行查询: select b from t where b=2 for update; 则辅助索引记录的状态如下

trx\_id(not active)

(1, 2, 3)

2

由上图可以看到查询得到的事务id是不活跃的，故b=2的辅助索引记录上不含有implicit lock，上述SQL查询不会被阻塞。

情况2

|  |  |
| --- | --- |
| 会话A | 会话B |
| BEGIN;  INSERT INTO t values (2, 3, 4); | SELECT b FROM t where b=3 for update;(阻塞) |

这时辅助索引记录状态

trx\_id

(2, 3, 4)

3

Prev\_trx\_id

NULL

NULL

由上表可以看到记录b对应的聚集索引为(2, 3, 4)，而这个聚集索引的事务id表示其是活跃(回话A中的事务还未提交)。通过聚集索引记录构造之前版本的记录为NULL，因此b=3的索引记录上含有implicit lock，回话B会被阻塞。

情况3

|  |  |
| --- | --- |
| 会话A | 会话B |
| BEGIN;  DELETE from t where a=1; | SELECT b FROM t where b=2 for update;(阻塞) |

这时辅助索引记录的状态如下

trx\_id(active)

(1, 2, 3)

3

NULL

(1, 2, 3) vers\_del=0

Prev\_trx\_id

可以看到当前记录与之前版本辅助索引记录的值都是相等的，即b的值都为2.不同的是两者delete flag标志位不同。故b=2的辅助索引记录上含有implicit lock，回话B会被阻塞。

情况4

|  |  |
| --- | --- |
| 会话A | 会话B |
| BEGIN;  Update t set b=4 where a=1; | Select b from t where b=4 for update;(阻塞) |

4 rec\_del=0

trx\_id(active)

(1, 4, 3)

Prev\_trx\_id

(1,2,3) vers\_del=0

2

可以看到当前记录与之前版本辅助记录的值是不相等的，并且当前记录的delete flag 标志位为0.

故b=4的辅助索引记录上含有implicit lock，会话B会被阻塞。

情况5

|  |  |
| --- | --- |
| 会话A | 会话B |
| BEGIN;  update t set b=4 where a=1;  update t set c=4 where a=1; | Select b from t where b=4 for update;(阻塞) |

trx\_id(active)

4(rec\_del=0)

(1,4,4)

4

(1,4,3) vers\_del=0

Prev\_trx\_id

2

(1,2,3) vers\_del=0

Prev\_trx\_id

在上述的情况下，需要通过遍历2次行记录版本才能判断辅助索引记录上是否有implicit lock。首先在产生的行记录版本下，当前记录与之前版本辅助记录的值是相等的(rec==entry)，并且两者的delete flag标志位都为0。由于两者之间的事务id也相同，因此需要再遍历之前的记录版本，而这时会出现上述的第4种情况。故b=4的辅助索引记录含有implicit lock，回话B会被阻塞。

情况6

|  |  |
| --- | --- |
| 会话A | 会话B |
| BEGIN;  Update t set b=4 where a=1;  Update t set b=2 where a=1;  Delete from t where a=1; | Select b from t where b=4 for update;(阻塞) |

状态如下图所示

4(rec\_del=1)

(1,4,3)

trx\_id(active)

2

(1,2,3) vers\_del=0

prev\_trx\_id

4

prev\_trx\_id

(1,4,3) vers\_del=0

对于这个情况，innodb存储引擎需要通过遍历2次行记录版本才能判断辅助索引记录上是否有implicit lock。首先当前辅助索引记录与之前版本辅助索引记录的值不相等时(rec !=entry),然而辅助索引记录的delete flag标志位为1，由于两个记录版本的事务id相同，这就是上述分析的第6种情况。因此需要遍历再之前的记录版本，而这时会出现上述分析的第3种情况。故b=4的辅助索引记录上含有implicit lock，会话B会被阻塞。

## 行锁的维护(insert,update ,purge,分裂,合并 时)

**insert记录时的维护锁**

对innodb存储引擎进行插入操作时，需要进行如下步骤的操作：

1. 首先对表加上IX锁。
2. 根据PAGE\_CUR\_LE定位记录next\_rec
3. 判断记录next\_rec是否有锁，有的话等待锁的释放，否则直接插入。

插入操作需要定位插入记录的下一条记录，这时next-key locking算法所要求，因为该算法下锁定的不仅仅是记录本身，锁定的是区间。列入下面的记录：

1, 2, 3, 4, 5, 7, 8

若要插入6这个记录，首先根据查询模式PAGE\_CUR\_LE定位到记录5，接着判断5这条记录的下一条记录是否有锁，因为如果有锁，则根据next-key locking算法，其表示锁定的范围是(5, 7]或者(5,7)(gap标志位为1)。因此若记录7上有锁，则不允许在这个范围内进行插入操作。所以插入记录6的操作将被阻塞。对于innodb存储引擎而言，若记录next\_rec上没有锁，则直接插入，不产生任何锁对象。否则调用函数lock\_rec\_enqueue\_waiting,等待记录next\_rec上锁的释放，这时会产生锁的对象，锁定的记录为next\_rec，锁的类型为LOCK\_X|LOCK\_GAP。

此外，若下一条记录next\_rec上有锁，不管持有该锁的是否为插入操作事务本身，当插入操作完成后（无需事务提交），需要调用函数lock\_update\_insert来更新锁定的范围。列如上面的例子，若插入了6这条记录，则原来锁定的范围从(5,7]更新为(5,6)，(6, 7]。这样就阻止了其他事务在(5, 6)的范围内进行插入操作。

需要注意的是，若插入的表上有辅助索引，那么还需要对辅助索引记录进行锁的判断，其方法与步骤2，步骤3相同。只是在判断可以进行插入后，还需要更新辅助索引页page header中的PAGE\_MAX\_TRX\_ID的值。

函数lock\_rec\_insert\_check\_and\_lock用来判断next\_rec上的锁，参数inherit用来判断是否在插入完成后调用函数lock\_update\_insert来对已经锁定的范围进行更新。

**Update记录时维护锁**

当事物需要对记录进行更新(包括删除操作)前，首先尝试对更新的记录加上X(implicit lock),若待更新的记录上存在其他锁时，则事务被阻塞，需要等待其他记录上的锁被释放。

函数lock\_clust\_rec\_modify\_check\_and\_lock和lock\_sec\_rec\_modify\_check\_and\_lock分别对聚集索引和辅助索引进行加锁，两者的过程基本相同，都是调用函数lock\_rec\_lock对更新的记录进行加锁操作，不同的是:

* 聚集索引记录加锁前首先需要将记录上的implicit lock 转化为explicit lock。
* 辅助索引记录加锁成功后，还需要更新辅助索引页page header中PAGE\_MAX\_TRX\_ID的值

在事务对记录进行更新的过程中，若记录不能进行原地更新，则需要对锁进行维护，其步骤如下：

1. 将更新记录的锁信息移动到页的infimum记录上(函数lock\_rec\_store\_on\_page\_infimum);
2. 删除原记录
3. 插入更新完成后的记录
4. 将页的infimum记录上的锁重新移动到新插入的记录上。

这里可以发现伪记录infimum的作用，其在更新时临时保存更新记录锁的信息。这样做的好处是可以提高更新锁的效率，否则可能需要删除锁对象然后再创建新的锁对象。

**PURGE**

Innodb存储引擎对于记录的删除操作，首先是通过将其记录的record header中的delete flag标志位设置为1，然后通过后台的purge线程将记录进行真正的删除，这个步骤是通过函数page\_cur\_delete\_rec完成

当purege真正删除记录完成后，删除记录的下一个记录需继承删除记录的锁定范围，并且其模式是GAP的，同时释放并重置删除记录上等待锁的信息。上述过程通过lock\_update\_delete完成

此外，对于标记为delete flag的加锁记录，innodb存储引擎在定位记录后需要进一步扫描记录才能确定查询是否为结束。来看一个例子，首先根据如下清单创建表a：

Create table a (

a int primary key,

b varchar(30)

) type=innodb;

Insert into a values (1, ‘a’);

表a有一个主键值为1的记录，若这时执行下面的操作:

BEGIN:

UPDATE a set b=repeat(‘a’, 30) where a=1;

Select \* from a where a<=1 for update;

由于更新记录的大小发生了改变，这时不能进行原地更新，因此innodb存储引擎对于update操作首先是进行了delete mark操作，然后再插入了一条主键值为1的新记录。这意味着当purge操作发生之前，一个页中包含2个主键值为1的记录。这也符合innodb存储引擎多版本并发控制的要求。但是在未执行purge操作删除之前主键值为1的记录，执行了事务的第二条select语句，则innodb存储引擎会对两个主键值为1都加上一个x-lock。因为第一次加锁的记录，其record header 的delete flag被标记为1了，因此还需要访问record header中的next record来判断是否已经结束的扫描。

然而这有时会导致一些用户可能难以理解的问题，例如下面的示例，同样采用前面创建的测试表a:

Drop table if exists a;

Create table a (

a int primary key,

b varchar(30)

) type = innodb;

Insert into a values (1, ‘a’) ; # heap\_no: 2

Insert into a values (2, ‘b’) ; # heap\_no: 3

Insert into a values (3, ‘c’) ; # heap\_no: 4

Insert into a values (4, ‘d’) ; # heap\_no: 5

接着首先在会话A中运行事务，注意事务未提交:

BEGIN;

SELECT \* FROM a where a=4 for update;

然后再会话B中运行下列事务:

BEGIN;

SELECT \* FROM a where a<=2 lock in share mode;

Delete from a where a=3;

SELECT \* FROM a where a<=2 lock in share mode;--阻塞

用户会“惊奇的”发现会话B中的第二个SELECT语句会被阻塞，而同样第一次的SELECT操作已经加锁成功的。其实导致这个现象的原因是SELECT游标锁定的最大记录被标记为了删除(未被真正purge删除)，因此，当第二次再次执行SELECT操作时，需要进一步锁定记录，也是a等于4的记录，而该记录已经在会话A的事务中被锁定了。

同一范围的GAP类型锁是可以相互兼容的，例如两个事务分别持有(2, 4)范围的GAP锁是被允许的，GAP锁只是用来阻止在这个范围内进行插入，甚至是持有GAP锁的事务本身。例如有2, 3, 4 这三个记录，事务A持有(2, 3)这个GAP锁，类型为X。而事务B持有(3, 4)这个GAP锁，类型为S。不同的事务可能持有不同兼容性的GAP锁。如下所示

事务B，S锁

2

3

4

事务A，X锁

delete & purge

3

4

2

事务C

事务B，S锁

事务A,X锁

从上图可以看到，若事务A, B开始持有不同类型的GAP锁，若事务C删除了记录3，并且事务提交后进行了purge操作。这时会对锁定的范围进行合并，而事务A和B这时都锁定了(2, 4)这个范围，然而持有的锁的类型完全不同。

**一致性的锁定读**

默认情况下，innodb存储引擎使用一致性的非锁定读(consistent non-locking read),即读取不会被阻塞。然而有些情况下用户希望通过锁定读取的方式来保证数据的一致性，这时可以通过语法LOCK IN SHARE MODE 和for update 主动对读取进行加锁操作，称这种方式为一致性的锁定读(consistent locking read)。函数lock\_clust\_rec\_read\_check\_and\_lock和lock\_sec\_rec\_read\_check\_and\_lock分别用来对聚集索引和辅助索引记录进行加锁。

对于聚集索引记录，只需对主键值进行加相应的锁就可以了。而对于辅助索引记录，除了需要对辅助索引记录本身加锁，还需要对主键索引记录加锁，如上文件创建的表a，根据如下语句创建辅助索引:

Alter table a add key idx\_b(b);

若运行下述的SQL语句:

SELECT \* FROM a where b=’a’ lock in share mode;

那么除了辅助索引上对b等于’a’的记录进行加s-lock操作外，还需要对聚集索引上对于a等于1的记录同样加锁s-lock。此外，还需要对辅助索引记录’a’的下一条记录加上一个s-lock。这是为了避免其他事务插入b等于’a’的记录，从而导致幻读问题的产生。若列b上的索引是唯一索引，那就不需要对记录’a’的下一条记录上s-lock了，因为唯一性包子不会有两个记录其列b的值都为’a’，也就不会有幻读的问题。

**页的分裂(btr\_page\_split\_and\_insert)**

Innodb存储引擎可以往左以及往右分裂，下面的步骤是往右分裂的步骤

1. 首先确定分裂点记录split\_rec
2. 调用函数lock\_rec\_move将记录splic\_rec到记录supremum之间所有的锁移动到分裂产生的新页(right page)中，修改对应的lock bitmap中的值。
3. 调用函数lock\_rec\_move将原来页中的记录supremum持有的锁移动到新页的记录supremum;
4. 调用函数lock\_rec\_inherit\_to\_gap将新页第一条记录的锁继承给原页的记录supremum(类型为gap)

上述过程在函数lock\_update\_split\_right中实现 ， split的整个过程如下图所示。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PAGE | | | | | | | | |
| Infimum |  |  |  | Split\_rec |  |  |  | supremum |
|  | 记录1 | 记录2 | 记录3 | 记录4 | 记录5 | 记录6 | 记录7 |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Left page | | | | | Right page | | | | | |
| Infimum |  |  |  | supremum | Infimum | Split\_rec |  |  |  | supremum |
|  | 记录1 | 记录2 | 记录3 |  |  | 记录4 | 记录5 | 记录6 | 记录7 |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Left page | | | | | Right page | | | | | |
| Infimum |  |  |  | supremum | Infimum | Split\_rec |  |  |  | supremum |
|  | 记录1 | 记录2 | 记录3 | 此处从split\_rec上继承锁 |  | 记录4 | 记录5 | 记录6 | 记录7 | 此处也要从原先的锁 |

往左分裂和往右分裂差不多，但是不需要执行第3个步骤，因为往左的分裂，记录supremum的锁是稳定的，往左分裂通过函数lock\_update\_split\_left实现

**页的合并**

与页的分裂操作一样，innodb存储引擎页的合并操作依然可分为往右的合并与往左的合并。当发生页的合并操作时，同样需要对行锁信息进行维护。首先来看往左的分裂，其步骤如下：

1. 记录左页合并前最大的用户记录orig\_pred
2. 调用函数page\_copy\_rec\_list\_start将右页中的用户记录复制到左页，同时更新对应行锁的信息
3. 调用函数lock\_rec\_inherit\_to\_gap将右页记录supremum上的锁移动给左页的supremum。
4. 调用函数lock\_rec\_move将右页记录supremum上的锁移动给左页的supremum

上述过程通过lock\_update\_merge\_left实现。

# B+树索引

## btr\_cur\_search\_to\_nth\_level(index, level, mode, latch\_mode, cursor,has\_search\_latch)

参数说明：

1.mode: 虽然查询的要求可能有各种情况，如 >, >=, <, <=，但是在非叶子节点进行查询时查询模式仅能为 < 或者 <=，对于插入操作，其在叶子节点上总是通过模式PAGE\_CUR\_LE查找记录，查找得到的记录为待插入记录的前一条记录。

若对主键和唯一索引进行查询，其查询模式的选择为PAGE\_CUR\_GE,而不能是PAGE\_CUR\_LE。这时因为首先需要考虑innodb存储引擎支持多版本并发控制，因此即使对健值进行了唯一约束，但实际在页的存储中，仍然可能包含多个健值相同的记录，只是仅一个记录的delete flag 为0,并且该记录总是在最后一个。例如下面的叶子节点的记录：

(1, ‘a’), \*(2, ‘b’), (2, ‘c’), \*(3, d), (3, e), (4, f), (5, g)

对上层用户来看，该记录的主键为1,2,3,4,5。而实际页中可能存在多个重复的主键记录，只是其delete flag =1(这里用\*标记了)。那么如果根据模式PAGE\_CUR\_LE查询主键3的记录，则会得到查询记录的前一个，很显然这会引起并发错误。因此当根据唯一的约束进行查询时，必须使用模式PAGE\_CUR\_GE。

虽然查询模式有4种，但是在非叶子节点进行查询时，只能有PAGE\_CUR\_LE和PAGE\_CUR\_L这两种模式，在源码中变量page\_mode表示的就是非叶子节点的查询模式

switch (mode) {

case PAGE\_CUR\_GE:

page\_mode = PAGE\_CUR\_L;

break;

case PAGE\_CUR\_G:

page\_mode = PAGE\_CUR\_LE;

break;

default:

#ifdef PAGE\_CUR\_LE\_OR\_EXTENDS

ut\_ad(mode == PAGE\_CUR\_L || mode == PAGE\_CUR\_LE

|| mode == PAGE\_CUR\_LE\_OR\_EXTENDS);

#else /\* PAGE\_CUR\_LE\_OR\_EXTENDS \*/

ut\_ad(mode == PAGE\_CUR\_L || mode == PAGE\_CUR\_LE);

#endif /\* PAGE\_CUR\_LE\_OR\_EXTENDS \*/

page\_mode = mode;

break;

}

显然这时因为B+树特点所致，然而这会发生得到的记录时伪记录supremum的情况，根据如下清单创建测试表t

Drop table if exists t;

Create table t (

a int not null,

b blob,

primary key(a)

) type=innodb;

Insert into t values (1 ,repeat(‘a’, 7000));

Insert into t values (3 ,repeat(‘b’, 7));

Insert into t values (5 ,repeat(‘b’, 7));

Insert into t values (7 ,repeat(‘b’, ‘7’));

Insert into t values (9 ,repeat(‘b’, ‘7’));

Insert into t values (11 ,repeat(‘c’, 7000));

Insert into t values (13 ,repeat(‘d’, 7000));

上述清单根据健值顺序的插入了7条记录，插入第7条记录会导致分裂。由于之前已经顺序插入了5条记录，因此有读者可能认为分裂点记录就是插入的最后主键为13的记录，然而实际并非如此，实际分裂后B+树索引的状态如下图所示。

1 | | 7 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | 1，repeat(‘a’, 7000) | | 3 , repeat(‘b’, 7) | | 5 , repeat(b,7) | | |  | | --- | | 7 , repeat(b,7) | | 9 , repeat(b,9) | | 11 , repeat(c,7000) | | 13 , repeat(d,7000) | |

从上图可以看出innodb引擎选择的中间记录作为分裂点记录，这时因为在分裂前首先提高了树的高度，innodb引擎的root页的位置是不变的，其通过page\_copy\_list\_end\_to\_create\_page将root页中的记录移动到新页中，在这个函数中会将PAGE\_N\_DIRECTION重置为0，因此当插入主键值为13的记录时，不会讲插入的记录作为分裂点的记录，而是根据无序的插入来进行判断的，那结果就如上图了。

若这时进行如下查询：

Select a, length(b) from t where a=7;

那么通过函数btr\_cur\_search\_to\_nth\_level查询得到的结果为记录supremum,而不是直接查询得到主键为7的记录。导致这个现象的原因是虽然主键查询的模式为PAGE\_CUR\_GE,但是在非叶子节点中(这个例子中就是root页)的查询模式为PAGE\_CUR\_L，也就是查询主键小于7的记录。则根据B+树的特性，其查询的是root页记录1指向的下层记录而该页的记录都小于7，因此函数最后返回了伪记录supremum。这不会对最终结果造成任何影响，因为在函数row\_search\_for\_mysql中，若查询得到的记录为supremum，innodb存储引擎会自动搜索下一个页以便得到最终正确的数据。同时也是这个原因，对于唯一约束的健值进行查询时，需要使用模式PAGE\_CUR\_GE而不是PAGE\_CUR\_LE.

这还会对加锁操作产生一定的影响，例如下面的SQL语句；

Select a, length(b) from t where a=7 lock in share mode;

由于首先定位到的是左页中的记录supremum，因此innodb存储引擎需要对这个记录进行加s-lock，然后再对右页中的主键为7的记录加s-lock，尚若这时又有新的事务在另外的会话中执行如下的语句；

Insert into t values (6, ‘a’);

那这个语句会被阻塞。而原因就是左页中的记录supremum已经被之前的事务加s-lock了。插入记录需要判断下一条记录是否没有锁，因此插入操作发生等待，而从理论上来说，这个插入是可以立刻执行的，并不会产生任何的并发问题。

并不会总是首先定位到记录supremum，然后再进行查询，这样会多一次额外的I/O开销。这种情况仅在查询页中的第一条用户记录时才可能会产生问题。

2.latch\_mode

参数latch\_mode表示需要对页和索引内存对象加何种latch进行保护，该参数的有效值说明如下

|  |  |
| --- | --- |
| 有效值 | 说明 |
| BTR\_SEARCH\_LEAF | 对索引内存对象加s-latch，对查询的页加s-latch。当查询到叶子节点时释放索引内存对象的s-latch |
| BTR\_MODIFY\_LEAF | 对索引内存对象加s-latch，对查询的页加x-latch。当查询到叶子节点时释放索引内存对象的s-latch |
| BTR\_NO\_LATCHES | 对索引内存对象加s-latch,对查询的页不加任何latch保护 |
| BTR\_MODIFY\_TREE | 对索引内存对象加x-latch，对查询的页不加任何latch保护 |
| BTR\_CONT\_MODIFY\_TREE | 函数开始前已经对索引内存对象加x-latch，对查询的页加x-latch |
| BTR\_SEARCH\_PREV | 对查询的页加s-latch |
| BTR\_MODIFY\_PREV | 对查询的页加x-latch |
|  |  |
| BTR\_INSERT | 表示此次查询是否用于insert操作 |
| BTR\_ESTIMATE |  |
| BTR\_DELETE |  |
| BTR\_DELETE\_MARK |  |
| BTR\_IGNORE\_SEC\_UNIQUE |  |

若函数btr\_cur\_search\_to\_nth\_level最终定位到的页不是叶子节点，即level大于0，那么对页加x-latch，同时不释放索引内存对象上的latch保护。否则根据latch\_mode对叶子节点加上latch保护，并根据latch\_mode选择是否释放索引对象上的latch保护。

从上表中可以发现innodb存储引擎总是首先对索引内存加上s-latch保护，然后进行页操作，若操作insert，update，delete不会导致非叶子节点发生变化，即不会发生分裂、合并、树的高度变化，则其立即释放索引内存对象的s-latch保护，称这种方式为乐观方式。否则，立即释放索引内存对象以及页上的latch，并对内存索引对象和页加x-latch保护，这种方式称为悲观保护方式，并发在此处受到了限制。

参数latch\_mode还包含以下的flag

BTR\_INSERT

BTR\_ESTIMATE。

BTR\_INSERT表示此次查询是否用于insert操作。Innodb存储引擎会对非唯一辅助索引插入进行优化，因此BTR\_INSERT表示是否可以进行插入的优化操作。

BTR\_ESTIMATE表示需要对返回的结果集数量进行预估，仅在RANGE查询开始前需要进行预估操作。

3.cursor

struct btr\_cur\_t {

dict\_index\_t\* index;

page\_cur\_t page\_cur;

buf\_block\_t\* left\_block;

que\_thr\_t\* thr;

enum btr\_cur\_method flag;

ulint tree\_heigh;

ulint up\_match;

ulint up\_bytes;

ulint low\_match;

ulint low\_bytes;

ulint n\_feids;

ulint n\_bytes;

ulint flod;

btr\_path\_t\* path\_arr;

}

数据结构btr\_cur\_t各个变量的说明如下

|  |  |
| --- | --- |
| 变量 | 说明 |
| index | 此次查询使用的索引 |
| page\_cur | 查询得到的结果 |
| left\_block | 查询得到记录所在页的左兄弟页 |
| Thr | 线程队列 |
| flag | 使用何种查询得到记录结果，有效值为：   * BTR\_CUR\_HASH使用自适应哈希查询得到记录 * BTR\_CUR\_HASH\_FAIL使用自适应哈希查询记录失败，之后使用B+树索引查询成功。 * BTR\_CUR\_BINARY直接使用B+树索引查询得到结果。 * BTR\_CUR\_INSERT\_TO\_IBUF使用插入缓存进行记录的插入操作 |
| Tree\_height | 树的高度 |
| up\_match | 函数page\_cur\_search\_with\_march返回的结果 |
| up\_bytes |
| low\_match |
| low\_bytes |
| Fold | Hash fold |
| Path | 查询得到记录的路径，并保存每个路径上的查询信息. |
|  |  |

struct btr\_path\_t {

ulint nth\_rec;

ulint n\_recs;

ulint page\_no;

ulint page\_level;

};

遍 nth\_rec表示查询得到的记录时页中的第几个记录，或者说记录前还包含有多少记录(包括伪记录).变量n\_recs表示页中一共有多少个记录。通过这些信息可以预估得到一个RANGE查询返回的记录数量，这通过btr\_estimate\_n\_rows\_in\_range完成。

估算方法：

1,统计range查询时两次定位之间的记录的数量，即n\_rows.

2,统计下层共有多少页：n\_rows = n\_rows \* 每个页的平均记录数.

3,当查询得到叶子节点后，预估RANGE的记录数量：若树的层数小于等于2层，则返回

预估值为n\_rows，则返回2\*n\_rows.

因为B+树索引一个页中至少有2个记录，因此最后的预估值的计算为2\*n\_rows，但显然这时最小的可能性，因此是underestimate，来看下图所示range查询的例子。

|40 | | 80 | | 120 | | 160 |

|120 | | 130 | 150 |

|80 | | 90 |

|40 | | 50 | | 60 | | 70 |

|160 | | 170 | | 180 |

在上述的例子中假设要估算大于40,小于120 (40, 120)的记录数量时，首先根据查询模式PAGE\_CUR\_GE调用函数btr\_cur\_search\_to\_nth\_level查询记录40，这时得到的查询路径path信息为：

Path1[0].nth\_rec =1 ,path1[0].n\_rec = 4;

Path1[1].nth\_rec =1 ,path1[1].n\_rec = 4;

同样的，根据查询模式PAGE\_CUR\_LE调用函数btr\_cur\_search\_to\_nth\_level查询记录40，

这时得到的查询路径path信息为：

Path2[0].nth\_rec =3 ,path2[0].n\_rec = 4;

Path2[1].nth\_rec =1 ,path2[1].n\_rec = 3;

那么根据之前介绍的预估流程，则

* 首先判断root页两次查询之间的距离， n\_rows = path2[0].nth – path2[0].nth\_rec=2。
* 接着查询到下层：n\_rows = n\_rows \* 每个页的平均数量 = 2 \*(path[1].n\_rec + path[2].n\_rec) /2 = 2\*3 =6;
* 由于树的高度为2，所以range的预估值就为n\_rows，即预估的range查询记录为6。

可以看到实际range之间的记录为7个记录，这里预估得到的结果为6,。此外这个例子树的高度为2，所以直接返回n\_rows的值就可以了。

需要注意的是，不同的查询模式下，cursor定位得到的记录可能完全不相同，例如有以下记录，其中a是主键：

(a, b), \*(2, 1), (2, 2), \*(4, 3), \*(4, 4), (4, 6), (6, 6), (8, 8)

若查询a=2的记录，则调用的查询模式为PAGE\_CUR\_GE，查询完成后cursor指向的记录为(2, 2)。Innodb存储引擎多版本的特性为最后一个主键值a为2的记录时真正需要查询的记录，之前主键值为a的记录是伪删除的记录，需要等待被purge线程进行清理。

若查询a>2的记录，则查询模式为PAGE\_CUR\_LE | BTR\_INSERT，查询完成后 cursor指向的记录为(2, 2).

若查询一个不存在的记录，如a=3的记录，则cursor指向的记录应该为(4, 3)。

## DML操作

Innodb会有乐观和悲观两个阶段，对于insert操作而言，首先调用的是乐观插入，如果发现此次操作会导致页的分裂，则调用悲观插入。乐观插入的函数为btr\_cur\_optimistic\_insert。在调用乐观插入时，若发现记录不能插入到页中，首先对页进行一次整理操作，然后再尝试是否可以进行插入。若还是不行，即发生分裂或者树的高度发生变化，再调用悲观插入。

## 插入

函数btr\_cur\_optimistic\_insert用来进行乐观的插入。函数btr\_cur\_optimistic\_insert的参数说明如下表所示。

|  |  |
| --- | --- |
| 参数 | 说明 |
| flag | 有效值为:  BTR\_NO\_LOCKING,表示插入后不需要对查询的记录加上一个锁。例如对于插入缓存，不会对其进行并发的读取操作，因此不需要对插入的记录进行保护。  BTR\_KEEP\_SYS\_FLAG，表示更新记录的隐藏列roll\_ptr。例如当插入的是辅助索引或者是非叶子节点时，记录不含有隐藏列，因此需要设置该标志位。  BTR\_NO\_UNDO\_LOG\_FLAG，表示不需要记录undo log。例如回滚时不需要再次产生undo log。 |
| cursor | Cursor指向插入前通过函数btr\_cur\_search\_to\_nth\_level定位到待插入记录的前一条记录，其查询mode为PAGE\_CUR\_LE，latch\_mode为BTR\_MODIFY\_LEAF |
| offsets |  |
| heap |  |
| entry | 待插入的记录，用逻辑记录表示 |
| rec | 当插入成功，返回插入后的记录 |
| big\_rec | 当插入成功，并且插入的记录转化为大记录，返回大记录格式 |
| n\_ext |  |
| thr | 查询线程 |
| mtr | 页的日志 |

插入的记录开始为逻辑记录，因此首先需要将其转化为物理记录，当转化为物理记录后可以计算记录页是否有足够的空间来容纳新插入的记录，若没有函数返回DB\_FAIL。特别需要注意的是，innodb存储引擎要求插入操作保留1/16页的大小作为预留空间(填充因子)。这样做的好处是当发生update操作时，不需要立刻对页进行分裂操作。

若插入的记录tuple需要转换为大记录格式，则首先要将行溢出页的部分保存在变量big\_rec中，当调用函数btr\_cur\_optimistic\_insert完成后，再插入行溢出页列的部分。

将逻辑记录tuple转换为物理记录后，还需要通过函数btr\_cur\_ins\_lock\_and\_undo检查锁的信息，并生成对应的undo日志。若锁检测到下一个记录已经被其他事务持有锁，则函数返回DB\_WAIT\_LOCK.

接着进行插入操作，通过之前的检查，这一步确保记录是一定能够插入页中的。首先调用page模块的函数page\_cur\_insert\_rec\_low进行记录的插入操作，若插入失败，则调用函数btr\_page\_reorganize对页进行整理，释放一些页中的空间，然后再次调用函数page\_cur\_insert\_rec\_low进行记录的插入操作。总之，这里肯定能确保记录能够插入到页中，否则innodb宕机。

若插入操作需要更新内存中该页的锁信息，如插入时发生了等待，或者flag不包含BTR\_NO\_LOCKING\_FLAG标志位，则调用函数lock\_update\_insert。

最后若插入的对象时insert buffer时(insert bufffer本身也是一颗B+树)，更新对应插入缓冲位图页中的信息。

函数btr\_cur\_optimistic\_insert完成之后，会释放持有页的x-latch。若变量big\_rec不为NULL，表示有行溢出数据。这时需要对索引内存对象加x-latch，再对页加上x-latch，之后调用函数btr\_store\_big\_rec\_extern\_fields将溢出列的数据存放到行溢出页中。

悲观插入的函数为btr\_cur\_pessimistic\_insert，其函数的参数部分与乐观插入完全相同。不同点如下：

* 索引内存对象持有x-latch，页持有x-latch。Latch的开销较大，并发性受到限制。
* 悲观插入前，依然会调用btr\_cur\_optimistic\_insert尝试乐观插入(不过这时依然持有内存索引对象的x-latch)。这时因为可能事务在乐观插入后和本次悲观插入前，有其他的事务插入导致页的分裂。这时可能仅需要乐观插入就能完成操作。
* 若确定进行悲观操作，则需要预留一些区的空间，以保证索引树的结构发生变化，但是有足够的磁盘空间可以保证完成这次操作。插入操作完成后会释放这部分预留空间。
* 调用函数btr\_root\_raise\_and\_insert或者btr\_page\_split\_and\_insert完成分裂和插入操作。

预留空间算法如下，可以发现一般至少预留3个extents大小的空间：

if (!(flags & BTR\_NO\_UNDO\_LOG\_FLAG)) {

/\* First reserve enough free space for the file segments

of the index tree, so that the insert will not fail because

of lack of space

进行悲观插入之前，预留空间，以保证索引树的结构发生变化，

有足够的磁盘空间可以保证这次插入操作完成，

插入操作完成后，会释放这部分预留空间。

\*/

ulint n\_extents = cursor->tree\_height / 16 + 3;

success = fsp\_reserve\_free\_extents(&n\_reserved, index->space,

n\_extents, FSP\_NORMAL, mtr);

if (!success) {

return(DB\_OUT\_OF\_FILE\_SPACE);

}

}

此外，需要注意的是，在innodb中，对于非叶子节点的插入都是悲观插入。非叶子节点的插入有函数btr\_insert\_on\_non\_leaf\_level完成

## 非主键更新(update)

同插入操作一样，对于非主键更新操作也可以分为乐观和悲观两种。乐观更新中又可以分为原地更新和普通乐观更新。原地更新是指更新记录中各个列大小在更新过程中没有发生改变。需要特别注意的是，更新的所有列的大小都发生变化，而不是记录的整体大小没有发生变化。此外，包含extern属性的列同样不能进行原地更新。更新操作发生变化列的新值保存在数据结构

struct upd\_field\_t{

unsigned field\_no:16; /\*!< field number in an index, usually

the clustered index, but in updating

a secondary index record in btr0cur.cc

this is the position in the secondary

index \*/

#ifndef UNIV\_HOTBACKUP

unsigned orig\_len:16; /\*!< original length of the locally

stored part of an externally stored

column, or 0 \*/

que\_node\_t\* exp; /\*!< expression for calculating a new

value: it refers to column values and

constants in the symbol table of the

query graph \*/

#endif /\* !UNIV\_HOTBACKUP \*/

dfield\_t new\_val; /\*!< new value for the column \*/

};

struct upd\_t{

ulint info\_bits; /\*!< new value of info bits to record;

default is 0 \*/

ulint n\_fields; /\*!< number of update fields \*/

upd\_field\_t\* fields; /\*!< array of update fields \*/

};

可以看到数据结构upd\_t保存着更新后所有列的新值。变量fields并通过数据结构upd\_field\_t保存着每个列的编号、更新的值以及extern属性。函数row\_upd\_changes\_field\_size\_or\_external通过该update对象与记录比较就能得知是否可以进行原地更新。

一般辅助索引更新都是不能原地更新，其是delete mark + insert操作，这是因为辅助索引更新会导致B+树索引列的值发生变化。但是有一种情况比较特殊，那就辅助索引列的字符集发生了改变，但是排序规则没有发生改变，这时就可以原地更新。

函数btr\_cur\_optimistic\_update进行乐观更新，但是更新的前提是主键列没有发生改变。即更新不涉及主键列。此外乐观更新过程中，仅持有记录所在页的x-latch, btr\_cur\_optimistic\_update参数如下

|  |  |
| --- | --- |
| 参数 | 说明 |
| flags | 与乐观插入相同 |
| cursor | 指向需要更新的记录 |
| offsets | 保存更新操作发生变化列的新值 |
| heap |  |
| update |  |
| cmpl\_info | 辅助索引的更新信息 |
| thr | 查询线程 |

由于函数btr\_cur\_optimistic\_update乐观更新不对主键值进行修改，因此对于更新记录是否可以加x-lock，仅需要在加锁前将更新记录上可能存在的隐式锁转为显示锁，然后对更新的记录尝试加x-lock，若可以加x-lock，那么更新操作可以立刻执行。否则需要等待其他事务释放更新记录所持有的lock对象。

若不能再函数btr\_cur\_optimistic\_update中进行原地更新，那么原记录会先删除，然后插入新的记录，但是记录的主键没有改变，因此记录还是在同一个页中，只是新记录的heap\_no发生了改变，故需要更新锁的信息。这时采用将原来的锁信息移动到伪记录infimum上，待记录更新完成后再将伪记录上的锁移动到新记录上。这样做的好处是，记录的锁通过移动完成，不需要先删除后新建的方式来完成，提高了锁管理的效率。

悲观更新有函数btr\_cur\_pessimistic\_update完成，函数开始前已经有了索引内存对象的x-latch以及更新记录所在页的x-latch。其首先还是会调用乐观更新，因为可能更新记录所在的页已经完成分裂操作，因此本次更新操作使用乐观更新可能会成功。此外，悲观更新与乐观更新不同的是：

* 悲观更新可以更新含有extern属性的列，或者记录含有extern属性的列
* 悲观更新可以处理更新导致页会产生分裂的情况

若悲观更新导致页发生分裂，其插入操作使用函数btr\_cur\_pessimistic\_insert，并且要将BTR\_NO\_UNDO\_LOG\_FLAG、BTR\_NO\_LOCKING\_FLAG,BTR\_KEEP\_SYS\_FLAG的标志位打开。因为上述这些都有update操作生成，插入操作不需要再产生上述信息。

对于锁信息的处理，悲观更新与乐观更新对于更新记录本身锁的处理方式相同，即首先将更新记录的锁信息移动到伪记录infimum上，待更新完成后将锁信息移动回原记录上。然而由于悲观更新会引起页的分裂操作，因此在某些情况下，需要对伪记录supremum锁进行修正。

发生这种修正的条件为：

* 更新的记录不是页中的第一个用户记录
* 更新操作会引起页向右进行分裂
* 分裂点记录就是更新记录

那么这是分裂后原记录所在页的伪记录supremum会继承分裂点记录后的锁信息，并设置为gap类型。但是由于更新时记录本身已经有锁，因此伪记录supremum需要继承的是更新的记录，也就是分裂点的记录锁信息，而在分裂时，其锁信息被移动到了为记录infimum上。这个过程在函数btr\_cur\_pess\_upd\_restore\_supremum中完成

## 主键更新

前面介绍的btr\_cur\_optimistic\_update和btr\_cur\_pessimistic\_update都不构成对主键列的更新，即主键没有更新。若更新操作包含需要更新主键值变更，那么innodb的处理步骤如下:

* 将原主键的delete flag设置为1
* 插入新主键记录的值
* Purge线程判断是否有其他事务引用原主键记录，若无，对记录进行彻底的删除。

当前不关心步骤1和3，只关心步骤2。这种处理方式将一个更新操作实际变为了插入操作。对于列属性为extern进行区别对待，列属性为extern时，将其大部分数据存放在溢出页中，若更新没有更新extern，那么新的记录需要继承这个溢出页的数据。同时，在purge线程清理原记录时，若被删除的记录列不拥有(ownership)这个溢出页，那么不能对溢出页进行删除。类似的，若回滚时被删除的记录列继承溢出页，那么同样不能对溢出页进行删除。继承和拥有通过extern列上的BTR\_EXTERN\_OWNER\_FLAG和BTR\_EXTERN\_INHERITED\_FLAG进行标记。简单来说，BTR\_EXTERN\_OWNER\_FLAG为0表示记录拥有该溢出页数据，purge是可以进行删除。回滚时，若BTR\_EXTERN\_INHERITED\_FLAG为1，则不能删除溢出页的数据。

需要注意的是：在生产环境应该做到基本不对主键进行修改。

## 删除

Innodb支持多版本并发控制，因此对于删除操作，分为两个步骤

1. 更新记录的delete flag
2. Purge线程彻底删除记录

步骤1在触发删除操作的事务中完成，步骤2在后台的purge线程中完成。Purge线程需要检查是否还有其他事务在使用记录，如无，则可以将记录删除。通常来说，将步骤1称为delete mark操作，因为只是将记录的delete flag设置为1，记录依然存在于页中。步骤2称为彻底的删除，这时会将记录占用的空间放入到页的PAGE\_FREE中去。

函数btr\_cur\_del\_mark\_set\_clust\_rec和btr\_cur\_del\_mark\_set\_sec\_rec用来对聚集索引和辅助索引中的记录进行delete mark操作，即更新记录的delete flag。上述操作都需要对记录加上一个显示锁，因此需要分别通过函数lock\_clust\_rec\_modify\_check\_and\_lock和lock\_sec\_rec\_modify\_check\_and\_lock对记录进行加锁。由于辅助索引上没有隐式的锁，因此不需要将可能存在的隐式锁转化为显示锁。对于聚集索引记录的伪删除操作，还需产生undo日志，这时与辅助索引记录伪操作的不同点。

Delete mark 会产生如下格式的重做日志。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 聚集索引记录的delete mark 的redo log | | | | | | |
| MLOG\_REC\_CLUST\_DELETE\_MARK | \*space | \*offset | Flags | Val | Roll\_ptr | Rec\_offset |
| 辅助索引记录的delete mark的redo log | | | | | | |
| MLOG\_REC\_SEC\_DELETE\_MARK | \*space | \*offset | Val | Rec\_offset | | |

带\*标记的表示压缩方式进行存储，可以看到聚集索引记录与辅助索引记录的delete mark操作产生的日志大致相同，不同之处仅在与聚集引擎记录的delete mark操作需要存储额外的flags和roll\_ptr信息，而这些信息是聚集索引记录本身所独有的。

## 持久游标(persistent cursor)

Btr0cur模块用于树对B+树索引进行增删改查的操作，产生对应的redo log和undo log，并且处理可能发生的B+树分裂或者合并的情况。然而，在大多数情况下，上层并不直接调用btr0cur中的查询函数，例如btr\_cur\_search\_to\_nth\_level。而是通过一个称为(persistent curosr)的对象来处理查询，并通过这个对象来调用btr0cur中的查询函数。此外，将查询得到的记录信息保存在persistent cursor中。

数据结构btr\_pcur\_t

|  |  |
| --- | --- |
| 字段 | 说明 |
| btr\_cur | B+树索引的游标 |
| latch\_mode | 同函数btr\_cur\_search\_to\_nth\_level |
| old\_stored | 表示是否有记录存放在持有游标中，有效值为：   * BTR\_PCUR\_OLD\_STORED * BTR\_PCUR\_OLD\_NOT\_STORED |
| old\_rec | 持久游标保存查询得到的记录 |
| old\_n\_fields |  |
| rel\_pos | 持久游标定位到的位置，有效值为：  持久游标定位到的位置，有效值为：   * BTR\_PCUR\_AFTER\_LAST\_IN\_TREE定位到的页没有记录，伪定位到伪记录infimum * BTR\_PCUR\_BEFORE\_FIRST\_IN\_TREE定位到的页没有记录，且定位到伪记录supremum * BTR\_PCUR\_ON定位到用户记录，old\_rec保存指向的用户记录 * BTR\_PCUR\_BEFORE定位到infimum记录，old\_rec保存第一个用户记录 * BTR\_PCUR\_AFTER定位到记录supremum，old\_rec保存最后一个用户记录 |
| block\_when\_stored |  |
| modify\_clock | 上次保存的记录信息是页的clock信息,用于判断是否在restore过程中页发生变化，若有则需要重新定位记录 |
| pos\_state | 标记位置信息，有效值为：   * BTR\_PCUR\_IS\_POSITIONED * BTR\_PCUR\_WAS\_POSITIONED * BTR\_PCUR\_NOT\_POSITIONED |
| search\_mode | 同函数btr\_cur\_search\_to\_nth\_level |
| trx\_if\_known |  |
| old\_rec\_buf |  |
| buf\_size |  |

在进行select，udpate,delete操作时，首先需要定位到第一条记录，然后开始扫描下一条记录(fetch next record)，知道不符合条件的记录位置。持久游标用于保存每次查询到的记录，待查询下一条记录时，首先恢复上一次查询的记录，然后再获取下一条记录。这样设计的原因是当用户扫描记录时，页中的记录可能发生了变化，这时若按照之前的记录进行扫描，可能得到错误的情况。例如页中的记录发生了变化，或者页发生了分裂或者合并操作。若根据之前查询得到的记录保存在变量old\_rec中，这个过程通过函数btr\_pcur\_store\_position完成，待要扫描一下记录，需要进行恢复，若页没有发生改变则可以直接使用old\_rec来定位下一条记录，否则需要根据old\_rec中的索引健值重新定位记录再进行查询。这个过程通过函数btr\_pcur\_store\_postion完成。

对于唯一健值的等值查询且查询时锁的模式不为LOCK\_X时，innodb存储引擎做了优化，不需要持久游标保存查询得到的记录。

## 哈希索引

# Insert buffer(change buffer)

Insert buffer 只能缓存非唯一的索引，如果是唯一索引，那么在插入到insert buffer时需要判断插入的记录是否唯一，这需要回去读磁盘上的数据了，而insert buffer设计的目的是为了减少读取磁盘，所以插入唯一索引，会导致失去insert buffer的意义。

## 存储结构

Innodb也是一棵B+树，这棵B+树的健值是待插入到辅助索引页的page\_no。B+树索引的大小一样为16KB,其内存同样适用缓冲池。宏IBUF\_POOL\_SIZE\_PER\_MAX\_SIZE定义了insert buffer最多占用缓冲池中页的数量。目前该值为2，因此表示insert buffer最多可能占用1/2的缓冲池的大小。

Insert buffer 会将插入的辅助索引记录缓存起来，然而其必须保证一个辅助索引页中缓存的记录不会引起页的分裂。当缓存的记录过多时，其会主动的读取辅助索引页，首先将记录合并回页，然后再进行悲观插入操作。基于上述索引，innodb存储引擎存在一个称为insert buffer bitmap的页，其用来追踪每个辅助索引页的剩余空间，当剩余空间小于某个阈值时，那么必须主动进行一次合并操作。

一个insert buffer bitmap页总共可以管理16384个页(256个extents)。然而并不是每个页都是辅助索引页，因此insert buffer bitmap 页中保存信息需要占用4位空间，其定义如下表

|  |  |  |
| --- | --- | --- |
| 字段 | 占用bit数量 | 说明 |
| IBUF\_BITMAP\_FREE | 2 | 追踪辅助索引页的剩余空间 |
| IBUF\_BITMAP\_BUFFERED | 1 | 辅助索引是否在右记录的insert buffer中，用来判断是否需要进行合并 |
| IBUF\_BITMAP\_IBUF | 1 | 改页是否为insert buffer树中的页 |

一个页在insert buffer bitmap中共占用4位，前2位用来追踪辅助索引的剩余空间，由于仅有2位，因此其可能的值为0,1,2,3，表示的含义为：

* 0表示辅助索引页的剩余空间小于512bytes
* 1表示辅助索引页的剩余空间小于1024bytes
* 2表示辅助索引页的剩余空间小于1K – 2K
* 3表示辅助索引页的剩余空间大于2K

由于insert bufferbitmap只能追踪一个辅助索引页2k的可用空间，因此insert buffer 一次最多缓存一个辅助索引页的总大小为2k。函数ibuf\_index\_page\_calc\_free\_bits将辅助索引页的剩余空间转为对应的bit位，函数ibuf\_index\_page\_calc\_free\_from\_bits将bit位转为辅助索引页的可用空间容量。

当insert buffer bitmap页能判断辅助索引页的可用空间小于512字节时，其将会主动读取辅助索引页，并将insert buffer 缓存的对应记录插入到辅助索引页中。

一个insert buffer bitmap 页能管理16384个页，因此一个表空间中可能存在多个insert buffer bitmap页，此外，insert buffer bitmap页的位置总是固定的，其总是每16384个页的第2个页。函数ibuf\_bitmap\_page用来判断是否为insert buffer bitmap页，函数ibuf\_bitmap\_page\_no\_calc用来判断辅助索引页所对应的insert buffer bitmap页的page\_no。

除了insert buffer bitmap页之外，insert buffer 还设计了两个固定的页，insert buffer header页 和insert buffer root页。这两个页的位置同样是固定的，对应的page\_no分别为FSP\_IBUF\_HEADER\_PAGE\_NO(3) 和 FSP\_IBUF\_TREE\_ROOT\_PAGE\_NO(4)。

在之前介绍的B+树索引中，root页中保存着非叶子节点的页节点段的段头(segment header)信息。Insert buffer 的B+树索引仅有一个数据段，用来存放所有的非叶子节点和叶子节点。此外，该段的段头信息并不保存在root页，而是存放在一个独立的页中。这样的设计目的是为了避免死锁。

另一个与之前介绍的B+树索引不同的是，insert buffer B+树索引页的空间分配并不是直接通过段，而是其自身实现了一个FSP，用来管理可用空闲页，这部分页通过链表的形式保存在root页的PAGE\_BTR\_IBUF\_FREE\_LIST处。这样的设计同样是为了避免死锁

在insert buffer中，B+树索引的健值为原先待插入到辅助索引页的page\_no，因此需要对插入的辅助索引记录进行一些重构后，再插入到insert buffer 索引中。重构通过函数ibuf\_entry\_build完成

|  |  |  |
| --- | --- | --- |
| Ibuf rec | | |
| Page\_no | Field\_info | Secondary index record |

可以看到：对比直接插入到辅助索引的记录，插入到insert buffer 叶子节点的记录多了两部分信息。一部分是记录原先需插入到辅助索引页的page\_no，这也是insert buffer索引的健值。另一部分的内容是辅助索引每个列的信息，每个列占用4个字节，其保存的内容如下表

|  |  |
| --- | --- |
| 占用字节 | 说明 |
| 1 | 列的 类型(mtype) |
| 1 | 列的精确类型(prtype) |
| 2 | 列的长度 |

Innodb保存这部分插入列的信息是有必要的，因为在插入时插入的记录是逻辑记录，在之后从insert buffer 中合并到实际页中时，若没有上诉信息，就不能构造与插入时一样的完整逻辑记录。

当进行合并操作时，即将insert buffer中的记录插入到辅助索引数据页，函数ibuf\_build\_entry\_from\_ibuf\_rec用于将保存在insert buffer中的记录转化为待插入到实际辅助索引页的逻辑记录。函数ibuf\_search\_tuple\_build用来构造在insert buffer索引树种查询指定辅助索引页的逻辑。

## Insert buffer逻辑控制

Innodb最多使用缓存大小的1/2，这意味着辅助索引记录不可能一直停留在insert buffer树中，insert buffer只是一个暂时的插入缓冲，最终插入的记录还是要存储到辅助索引页的，称这个操作为merger。

根据不同的处理方式，可将合并的情况分为主动和被动，主动合并是指通过innodb的master thread，定期的将缓存在insert buffer中的记录回刷到辅助索引页。辅助索引页的选择是随机的，这有函数ibuf\_contract\_for\_n\_pages(ibuf\_contract，ibuf\_merge\_in\_background)中完成，在master trhead中，若每秒有足够空闲的io处理能力，则可刷5个缓存的辅助索引页的记录。对于是否有空闲IO处理能力。。。

被动合并是指，由用户线程主动发起的辅助索引页读取操作，这时会被动的将记录合并到原辅助索引页中。由于辅助索引页已经读取到缓冲池，那么之后对于该辅助索引页的插入操作将不会再将记录缓存在insert buffer中了，因为此时已完全没有必要了。

可以发现insert buffe其实只是一个纯粹的写缓存对象，仅对插入操作有帮助。由于根据辅助索引页进行索引，所以不能进行辅助索引记录的查询，因此对于读取操作没有任何帮助。

注意：在关闭mysql的时候，并不要求insert buffer 被合并到相应的表上去，而是会在下次启动的时候再进行合并。

Insert buffer 结构

struct ibuf\_t{

ulint size; /\*!< current size of the ibuf index

tree, in pages \*/

ulint max\_size; /\*!< recommended maximum size of the ibuf index tree, in pages \*/

ulint seg\_size; /\*!< allocated pages of the file

segment containing ibuf header and

tree \*/

bool empty; /\*!< Protected by the page

latch of the root page of the

insert buffer tree

(FSP\_IBUF\_TREE\_ROOT\_PAGE\_NO). true

if and only if the insert

buffer tree is empty. \*/

ulint free\_list\_len; /\*!< length of the free list \*/

ulint height; /\*!< tree height \*/

dict\_index\_t\* index; /\*!< insert buffer index \*/

ulint n\_merges; /\*!< number of pages merged \*/

ulint n\_merged\_ops[IBUF\_OP\_COUNT];

/\*!< number of operations of each type

merged to index pages \*/

ulint n\_discarded\_ops[IBUF\_OP\_COUNT];

/\*!< number of operations of each type

discarded without merging due to the

tablespace being deleted or the

index being dropped \*/

};

|  |  |
| --- | --- |
| 字段 | 说明 |
| Max\_size | 建议ibuf最多可以使用的page数量 |
| Size | 当前ibuf tree使用的page数量 |
| Seg\_size | 已经从段中分配页的数量 |
| Empty | Insert buffer tree 为空 |
| Free\_list\_len | Root页中free列表中的页的长度 |
| Height | Ibuf tree的高度 |
| index | Ibuf tree索引对象 |
| n\_merges | 合并到辅助索引页的次数 |
| N\_merged\_ops | 每种合并操作的合并次数  IBUF\_OP\_INSERT = 0,  IBUF\_OP\_DELETE\_MARK = 1,  IBUF\_OP\_DELETE = 2, |
| n\_discarded\_ops | 每种操作被丢弃的次数，因为insert buffer 中的记录对应的table被删除而被discard了 |
|  |  |

# 缓存池(buffer pool)

# 事务处理

几个问题

Doublewrite ,redo log ,undo log 之间的关系

1. 设计doublewrite是为了避免出现partial write的问题，因为innodb的page是16K，内核的对磁盘的原子操作时512字节，所以一个page完整的写到磁盘，不能保证是原子操作，中间弄个doublewrite buffer，如果page(16)写到磁盘出现了partial write了，在启动的时候就从doublewrite buffer中恢复。

在doublewrite只用2个extent的page，但是还是申请了2个extent的page加上32个fragment page，是因为doublewrite中不使用segment的fragment page，不使用fragment page，是不能保证fragment page是物理连续的

1. Undo log ,这个设计是为了事务回滚的需要，一致性的非锁定读是指innodb通过行多版本控制的方式来读取当前执行时间时数据库中行的数据。如果读取的行正在执行行delete或者update操作，这时读取操作不会因此去等待行上的锁释放，相反，innodb存储引擎会去读取行的一个快照数据。

直观的非锁定一致性读的情况如下图所示。之所以称为非锁定读，是因为非锁定读不需要等待访问记录上的x锁的释放。快照数据是指该行之前的版本的数据，该实现是通过undo日志来完成的。而undo日志用来在事务中回滚数据，因此快照本身是没有额外开销的。此外，读取快照数据不需要上锁，因为没有事务需要对历史的数据进行修改。

x-locked

Snapshot data

Snapshot data

Checkpoint 在何处

## 事务系统段

在innodb的设计中，在一个特定物理位置保存了关于事务的相关信息，这个对象称为事务系统段。事务系统段保存一下几部分信息：

* 事务相关信息
* 回滚段的segment header 信息
* Mysql二进制日志位置信息
* DOUBLEWRITE段信息

函数trx\_sys\_create用来创建事务系统段对象，并且将事务系统段的segment header保存在创建的事务系统段中，这与之前创建B+树索引叶子节点和非叶子节点段不同。而实际来说，事务系统段仅有一个页，用来存储上述4部分的基本信息。这也说明为什么在innodb的段设计中，要首先分配碎片页给段对象，因为有的段对象非常小，可能只有一个页的需求。这时直接分配一个区的页给段显得有些浪费。

## trx\_sys\_t和trx\_t类

trx\_t整体可以分为以下几个部分

* 事务相关信息
* 锁
* Undo日志
* Read\_view