《MySQL 源码系列》分享梗概

2020 年 6 月 9 日星期二
分享嘉宾：郝国庆　热璞数据库数据库专家

领导说既然开分享啦，就分享一些比较干的东西，我想来想去哈，binlog 比较合适。为啥呢:

1. binlog 是 mysql 实现复制的基础

2.复制是高可用啊、灾备啊的解决方案的基础(原生的高可用，灾备解决方案)

所以，binlog 就变成了高可用和灾备解决方案的基础。

3. 很容易获取，同时也好上手。如果上来就整 trx，lock，redo undo log。估计把大家整蒙了，大家也没有兴趣继续研究源码了。

今天我们就来介绍下 binlog 相关的知识

1.binlog mysql 二进制日志，顾名思义，binlog 是 mysql 在进行说句或者控制操作时产生的日志文件。使用 MySQL 的同学都知道 binlog 可以用来做数据恢复，搭建复制。我们可以用 binlog 做指定时间点的回复。

但是如果有同学不小心操作错了一条数据，但是这个数据库实例又比较大，如果为了恢复这一条数据而使用整个实例的备份文件，耗时很久还不方便，这个时候呢，其实我们是可以使用 binlog 来生成闪回语句进行单条数据的回退操作

2..binlog 文件由两部分组成，包括 relaylog，此时我们看下 mysql 的源码:

mysqlbinlog.cc:

function　dump_local_log_entries, check_header()

```
  if (memcmp(header, BINLOG_MAGIC, sizeof(header)))
  {
    error("File is not a binary log file.");
    DBUG_RETURN(ERROR_STOP);
  }
```

binlog 的 header 和 events

/* 4 bytes which all binlogs should begin with */

#define BINLOG_MAGIC "\xfe\x62\x69\x6e"

所有的 binlog event 都会以这 4 个字节开头，这 4 个字节用于校验该文件是否是 binlog 文件。这个事件结束之后，才会是一个个的 binlog event。

我们打开 binlog 试一下，可以开到开头。我们改掉一个 bin 中的一个字符，在用 mysqlbinlog 打开一下这个文件试一下，哈哈，报错了，File is not a binlog file

3. binlog 是在什么时候产生的呢

a.首先我们必须在配置文件里打开 binlog 选项，mysql 才会产生 binlog

b. binlog 有三种 format：**row、statements、 mix**

Row： 记录所有数据变化，比如一条 delete 语句，binlog 记录的 row event 解析出来可能是 where @1=？ and @2 = ？

statements delete from t1 where id < 10;

mix：兼具 row 和 statements 的优点，也就是说也有可能有他们两个的缺点哦。比如对表的操作可能会退化成语句格式，说是 DDL 走 statement，DML 走 row，但是实际测下来，部分 DML 还是走了 statement

后续我们所介绍的 trx 的构成不明说的话，都是以 row 格式哈。

c. 说道 binlog 的产生时间，就不得不提一个参数，sync_binlog = 1，如果说道 sync_binlog 等于就得介绍 innodb_flush_log_at_trx_commit.

sync_binlog 是针对，binlog 落盘的，当 sync_binlog = 1 时，一个事务 mysql 或对 binlog 做一次落盘，innodb_flush_log_at_trx_commit 是针对 innodb 存储引擎的，它对应的是 redo log undo log 的落盘。

我们知道 binlog 是在 mysql 的 server 层，redo log 是存储引擎，因此也需要保证 binlog 和 redo log 的数据一致性。mysql 使用了内部 xa 来保证数据的一致。这个往细里讲的话，也差不多要一节课。

源文件在 binlog.cc 的 ordered_commit 函数。今天不是咱们的重点，我就简单讲一下流

程

准备阶段:

1>. binlog 准备：将上一次 commit 队列中最大的 seq number 写入到本次事务的 last_commit 中

2>. innodb 准备：redo 写入 os cache，写 xid 到 undo 以便回滚使用

3>. xid 写入到到 binlog cache 中

提交阶段:

1>. innodb_flush_log_at_trx_commit = 1，sync 刷 redo,从 os cache 刷 redo log 到磁盘，循环每个事务的 binlog cache 到 os cache（有一个小点哦，如果此时 sync_binlog != 1 会触发 dump 线程发送 event 给从库）

2>. 将 binlog 从 os cache 刷到 binlog 磁盘,此时，如果 sync_binlog = 1 则会触发 dump 现场发送 event

-------大家想想如果有复制，是不是 mysql 5.7 的 after sync 的?

3>. 做 innodb 层的提交(redo commit)，引擎层一旦 commit，这个事务就算彻底执行完成，其他人就可以看到这个事务修改的数据了

-------5.7 之前的版本是在这个时候发送 binlog 给从的哦，所以叫 after comit

根据上面的流程，我们能够看到，其实在提交阶段，第 2 步，binlog 已经在磁盘上产生了。同时，也在这个时候将 binlog 发送给从库。所以我们可以说，after sync 是无损的复制。

上面我们介绍了 binlog 的一些东西，下面咱们从更细的角度来看 binlog:

我们在介绍 binlog 组成的时候，带了一句，binlog 是有 binlog header(4 字节) 和 events 组成的，因此我们可以说，event 是 binlog 中记录数据操作或者一个事务的最小单位了（可能有点绕）。

mysql 是事务数据库，gtid 出现之后，每个事务拥有自己的 trx_no，在 slave 回放时也是按照事务回放的，因此可以说 binlog 回放时的最小单位事务。我们可以理解 trx 是一个逻辑的概念，这个逻辑的概念怎么落到 binlog 上的二进制数据上呢，这就需要我们后续介绍的 event 了。一个 trx 是一对 events 的集合，这些 events 也是有规律可循的。

events 总共有 38 中不同的分类(包括 unkown_event),其中部分类型已经废弃了，但是为了

兼容还是保留着。

mysql 每个 event 有三部分构成：

event header： 19 个字节

fixed data(posted header)：format_description_event 中 post_headers 中记录该 event 对应的长度

variable data： 变动的变量的长度，比如 query_event 中 query_sql 的长度(后续介绍)

event header 构成如下：

timestamp	0:4
type_code	4:1
server_id	5:4
event_length	9:4
next_position	13:4
flags	17:2

我们先介绍一下，每个 binlog 中的第一个事件：format_description_event

```
head = {
  when = {
    tv_sec = 1591684192
    tv_use =0
  }
  event_type = 15
  event_type_name = Format_desc
  unmasked_server_id = 3
  event_len = 119
  log_pos = 123
```

```
    flags = 0
  }

  binlog_version = 4
  server_version = "5.7.23-debug-log"
  created = 1591684192
  common_header_len = 19
  post_header_len = std::vector of length 39, capacity 39 =
{56,13,0,8,0,18,0,4,4,4,4,18,0,0,95,0,4,26,8,0,0,0,8,8,8,2,0,0,0,10,10,10,42,42,0,18,52,0,1}
  checksum_alg = CRC32
  number_of_event_types = 38
```

可以看到 format_description_event 的结构，其中 post_header_len 解析每个 event 都会用到

我们再看一个 query event

```
{
  head = {
    when = {
      tv_sec = 1591684196
      tv_use =0
    }
    event_type = 2
    event_type_name = Query
    unmasked_server_id = 3
    event_len = 83
    log_pos = 302
    flags = 8
  }
```

query_data_written = 17

data_len = 12

thread_id = 2

query_exec_time = 0

db_len = 6

error_code = 0

status_var_len = 281474976710655

flags2 = 0

sql_mode = 1436549152

catalog_len = 3

catalog = std

auto_increment_increment = 0

auto_incrment_offset = 0

charset = "!"

time_zone_len = 0

time_zone_str =""

lc_time_names_number = 0

charset_database_number = 0

table_map_for_update = 0

master_data_written = 0

user_len = 0

user = ""

host = ""

host_len = 0

mts_accessed_dbs = 1

mts_accessed_db_names = {

  "testdb

}

explicit_defaults_ts = TERNARY_UNSET

q_len = 5

query = "BEGIN"

db = "testdb"

}

可以看到 event_type = 2 ，翻一下 log_event_type 的枚举类，可以看到 2 正好是 query event，上面说每个 event 还有变长部分，比如 query， db 等等。这个工具，是把 binlog 解析过了，

如果没有解析，我们直接读 16 进制数据呢，就以 query event 为例，我们看看 event type 的地址，下标为 4 开始，占一个字节，是不是也是 02

这样大家对 event 有一个大致的了解了吧。下面我们介绍下，各种类型的 trx 是怎么由 event 组成的。咱们先简单事务，在复杂事务(row 格式)，我们先介绍 DDL 吧

0. DDL，咱们创建个测试库，再创建个表

我们可以看到，创建测试库使用了一个 gtid_event，query event，创建表时又使用了 gtid_event 和 query event. 我们用我们开发的 binlog 解析工具看一下明细。

通过明细，我们可以看到第一个 query event 里存储的确实是 create databases 语句，第二个 query event 里存储的是存储的 query 是 create table 的语句

DDL 的 trx 由两个 event 构成，gtid event，query event

1. normal trx(dml)

我们接着做一个插入操作：insert into test1(name) values('ggggg');

可以看到，event 一下多了很多哦，从 154 开始，一个插入操作使用了如下 events

Gtid event

Query event

Table_map

Write_rows

Xid event

（Rotate 咱们不用关心，他是因为我做了 flush log 导致的。他表示该 binlog 文件结束，已

切换新的 binlog 文件。这个事件结束之后，才会是一个个的 binlog）

在 DML 当中，Query event 存储的就不是实际执行的语句，而是存储了一个 BEGIN

那我分别做一个 update 和 delete 大家看下，event 类型和顺序，我们可以看到 event 顺序和 insert 一样，不同的时，update，rows event 由 write rows 变成了 update rows，delete 则由 write rows 变成了 delete rows

这样大家对最简单的 DML 和 DDL trx 的 event 构成大家有概念了吧。

DDL: Gtid_event, Query_event

DML: Gtid_event, Qeury_event, Table_map_event, X_rows_event, Xid_event

有了这个概念，我们就介绍后续 trx 的 events 构成了。顺序我调整一下，我们后面介绍下 big trx。

我们会发现，big trx 的 events 组成和单条的 DML 一样， 唯一不同的就是 delete_rows_event 多了很多。

后面接着呢，是 modify multi table trx ，原谅我的 Chinglish 哈，我想说的时 multi table dml event，我们看一下顺序，它现在变成了：

table_map_event,write_rows_event, table_map_event, write_rows_event

注意 table_map_event 的 table_id 和 write_rows_event 的 table_id，会有一一对应的关系。rows event 对应的就是其上面紧接着的 table_map_event

这一点和后面的 trx 有些不同。

接下来我们看下 join update event，他和多表 dml 已经不一样了。他的顺序是 table_map_event,table_map_event，rows_event, rows_event.注意 table_id 的对应关系

接下来我们看下 trigger 相同的情况出现了，trigger 和 join update 的 binlog 顺序一样。那怎么区分这两种类型的事务呢？

最后，我们来看下 xa trx

我们可以看到一个 xa 事务，它的 events 顺序是酱紫的

Gtid，Query, table_map, write_rows query, XA_prepare, Gtid, Qeury.

而且它占了两个 gtid 哦。

为啥我们要对 trigger event， join update event 以及 xa event 做特殊介绍呢，因为，根据他 event 的类型不同，我们是要做不一样的处理的。

比如 xa event，它涉及多个 mysql 实例，如果我想讲这个 xa 事务，放到一个实例上回放该怎么处理?

依赖 binlog 的 replication

a. 搭建方法，时间比较紧，这个 DBA 基本都会，我就不介绍了哈

b. 复制的启动过程

与复制相关的命令主要包括了如下几个: change master、show slave stat、show master stat、start slave、stop slave 等命令。

sql_parser.cc

mysql_execute_command

```
case SQLCOM_CHANGE_MASTER:
  {

    if (check_global_access(thd, SUPER_ACL))
      goto error;
    res= change_master_cmd(thd);
    break;
  }
  case SQLCOM_SHOW_SLAVE_STAT:
  {
    /* Accept one of two privileges */
    if (check_global_access(thd, SUPER_ACL | REPL_CLIENT_ACL))
      goto error;
```

```
    res= show_slave_status_cmd(thd);
    break;
  }
  case SQLCOM_SHOW_MASTER_STAT:
  {
    /* Accept one of two privileges */
    if (check_global_access(thd, SUPER_ACL | REPL_CLIENT_ACL))
      goto error;
    res = show_master_status(thd);
    break;
  }
```

当在从库上，执行 change master to ，但没有 start slave 的时候，实际只是注册了一个信息，主机没有任何操作，从机会生成 relay_log.000001，在这个 relay log 文件中里面的 format_description_event
实际都是 slave 本机的。

start slave 之后，slave 做的操作

```
mysql_execute_command()
 |-start_slave_cmd()
    |-start_slave()
       |-start_slave_threads()
          |-start_slave_thread()                ← 先启动 IO 线程，无误再启动 SQL 线程
          | |-handle_slave_io()                 ← IO 线程处理函数
          |
          |-start_slave_thread()
             |-handle_slave_sql()               ← SQL 线程处理函数
```

io thread 会做如下操作

handle_slave_io()

|-my_thread_init() ← 0) 线程初始化

|-init_slave_thread()

|

|-safe_connect() ← 1) 以标准的连接方式连上 master

|-get_master_version_and_clock() 并获取主库的所需信息

|-get_master_uuid()

|-io_thread_init_commands()

|

|-register_slave_on_master() ← 2) 把自己注册到 master 上去

||-net_store_data() ← 设置数据包

||-simple_command() ← S 把自己的 ID、IP、端口、用户名提交给 M, 用于注册

|| ← **上述会发送 COM_REGISTER_SLAVE 命令**

|

| ###1BEGIN while 循环中检测 io_slave_killed()

|

|-request_dump() ← 3) 开始请求数据，向 master 请求 binlog 数据

||-RUN_HOOK() ← 调用 relay_io->before_request_transmit()

||-int2store() ← 会根据是否为 GTID 作区分

||-simple_command() ← 发送 dump 数据请求

| | ← ** 执 行 COM_BINLOG_DUMP_GTID/COM_BINLOG_DUMP 命令**

|

| ###2BEGIN while 循环中检测 io_slave_killed()

|

|-read_event() ← 4) 读取 event 并存放到本地 relay log 中

| |-cli_safe_read() ← 等待主库将 binlog 数据发过来

| |-my_net_read()

|-RUN_HOOK() ← 调用 relay_io->after_read_event()

|

|-queue_event() ← 5) 将接收到的 event 保存在 relaylog 中

|-RUN_HOOK() ← 调用 relay_io->after_queue_event()

|-flush_master_info()

当主库收到从库的注册申请时，主库做如下操作

```
bool dispatch_command(THD *thd, const COM_DATA *com_data,
                      enum enum_server_command command)
{
  ... ...
  switch (command) {
      ... ...
#ifdef HAVE_REPLICATION
      case COM_REGISTER_SLAVE:       // 注册 slave
          if (!register_slave(thd, (uchar*)packet, packet_length))
              my_ok(thd);
          break;
#endif
#ifdef EMBEDDED_LIBRARY
      case COM_BINLOG_DUMP_GTID:
          error= com_binlog_dump_gtid(thd, packet, packet_length);
          break;
      case COM_BINLOG_DUMP:
          error= com_binlog_dump(thd, packet, packet_length);
          break;
```

#endif

... ...

}

... ...

}

接着因为 slave 发送了 request_dump 命令，主就会通过通过 dump 线程将 binlog 发送给从库

dispatch_command()

|-com_binlog_dump_gtid() ← COM_BINLOG_DUMP_GTID

|-com_binlog_dump() ← COM_BINLOG_DUMP

|-kill_zombie_dump_threads() ← 如果同一个备库注册，会移除跟该备库匹配的 binlog dump 线程

|-mysql_binlog_send() ← 上述两个命令都会执行到此处

| ← 会打开文件，在指定位置读取文件，将 event 按照顺序发给备库

|-Binlog_sender::run() ← 调用 rpl_binlog_sender.cc 中的发送

|-init()

| |-init_heartbeat_period() ← 启动心跳

| |-transmit_start() ← RUN_HOOK()，binlog_transmit_delegate

|

|-###BEGIN while()循环，只要没有错误，线程未被杀死，则一直执行

|-open_binlog_file()

|-send_binlog() ← 发送二进制日志

| |-send_events()

| |-after_send_hook()

| |-RUN_HOOK() ← 调用 binlog_transmit->after_send_event()钩子函数

|

|-set_last_file()

|-end_io_cache()

|-mysql_file_close()

|-###END

接着我们介绍从的另一个线程的工作

handle_slave_sql()　　←###作为协调线程

|-my_thread_init()

|-init_slave_thread()

|-slave_start_workers()　　MTS(Multi-Threaded Slave)

| |-init_hash_workers()

| |-slave_start_single_worker()

| |-Rpl_info_factory::create_worker()

| |-handle_slave_worker()　　←###对于复制的并行执行线程

| |-my_thread_init()

| |-init_slave_thread()

| |-slave_worker_exec_job_group()

| |-pop_jobs_item()　　← 获取具体的 event(ev)，会阻塞等待

==<<<==

| |　　← 在 while 循环中执行

| |-is_gtid_event()

| |-worker->slave_worker_exec_event(ev)

| |-ev->do_apply_event_worker()　　← 调用该函数应用 event

| |-do_apply_event()　　← 利用 C++多态性执行对应的 event

|

|-### 如下从 IO 线程中读取数据

|-sql_slave_killed()　　← 只要线程未 kill 则一直执行

|-exec_relay_log_event()

|-next_event() ← 从 cache 或者 relaylog 中读取 event

| |-sql_slave_killed() ← 只要线程未 kill 则一直执行

| |-Log_event::read_log_event() ← 读取记录，第一参数为 IO_CACHE

| |-my_b_read() ← 从磁盘读取头部，并检查头部信息是否合法

| |-Log_event::read_log_event() ← 处理读取到缓存中的数据，第一个参数为 char*

| | ... ...

| |-Write_rows_log_event() ← 根据不同的 event 类型，创建 ev 对象

| |-Update_rows_log_event()

| |-Delete_rows_log_event()

| | ... ...

|

|-apply_event_and_update_pos() ← 执行 event 并修改当前读的位置

|-append_item_to_jobs() ← 发送给 workers 线程==>>>==

主要利用了 event 的多态

我们以 insert 为例，其 rows event 为 write rows event

```
int Rows_log_event::do_apply_event(Relay_log_info const *rli)
{
  ... ...
  table=
    m_table= const_cast<Relay_log_info*>(rli)->m_table_map.get_table(m_table_id);
  ... ...
  if (table)
```

```
{
  ... ...
  if ((m_rows_lookup_algorithm != ROW_LOOKUP_NOT_NEEDED) &&
      !is_any_column_signaled_for_table(table, &m_cols))
  {
    error= HA_ERR_END_OF_FILE;
    goto AFTER_MAIN_EXEC_ROW_LOOP;
  }
  switch (m_rows_lookup_algorithm)
  {
    case ROW_LOOKUP_HASH_SCAN:
      do_apply_row_ptr= &Rows_log_event::do_hash_scan_and_update;
      break;

    case ROW_LOOKUP_INDEX_SCAN:
      do_apply_row_ptr= &Rows_log_event::do_index_scan_and_update;
      break;

    case ROW_LOOKUP_TABLE_SCAN:
      do_apply_row_ptr= &Rows_log_event::do_table_scan_and_update;
      break;

    case ROW_LOOKUP_NOT_NEEDED:
      DBUG_ASSERT(get_general_type_code() == binary_log::WRITE_ROWS_EVENT);

      /* No need to scan for rows, just apply it */
      do_apply_row_ptr= &Rows_log_event::do_apply_row;
      break;
```

```
            default:
                DBUG_ASSERT(0);
                error= 1;
                goto AFTER_MAIN_EXEC_ROW_LOOP;
                break;
        }

        do {

            error= (this->*do_apply_row_ptr)(rli);

            if (handle_idempotent_and_ignored_errors(rli, &error))
                break;

            /* this advances m_curr_row */
            do_post_row_operations(rli, error);

        } while (!error && (m_curr_row != m_rows_end));
        ... ...
    }
    ... ...
}
```

它最后直接调用了 do_apply_row

```
do_apply_row()
  |-do_exec_row()
     |-write_row()
        |-ha_start_bulk_insert()
```

最后直接调用引擎层的 ha_start_bulk_insert 将数据插进去。