数据库内核月报 － 2023 / 01

当期文章

MySQL Binlog 源码入门

Author: 徐明威

MySQL主从节点之间同步数据，备份点还原，会用到binary log，其负责记录数据更改的操作。因为Binlog在运用到数据页之前需要经过复杂的过程，没有redolog直接，所以性能比不上直接使用redo复制的方式（物理复制的优势），但是它也有不可或缺的作用。本文重点介绍MySQL Binlog的作用、记录的内容、组织结构、写入方式、主备复制等内容，基于MySQL 8.0的代码。因为网上对Binlog各个知识点的介绍都非常详细，但是知识点非常杂，所以给本人初学Binlog的时候带来很多困难，因此本文的目的是总结这些知识点。

> 本文内容基于 MySQL Community 8.0.13 Version

1、为什么要有Binlog

MySQL上下分为SQL层和引擎层，不同存储引擎中的日志格式是不同的，由于要对多引擎支持，必须在SQL层设计逻辑日志以透明化不同存储引擎，而这个逻辑日志就是Binlog。当有数据修改请求时，primary会产生包含该修改操作的Binlog，并发送给replica，replica通过回放该Binlog以执行和primary同样的修改。此外还可用于备份点还原。

在PolarDB中，虽然通过物理复制可以完成上面的功能，但是MySQL生态中用户需要Binlog导出数据库做审计、数据校验、数据清理等操作；以及用户混合使用多种数据库搭建业务平台也需要Binlog完成不同数据库之间数据传输；还有某些数据备份工具例如阿里云DTS、第三方的OGG（Oracle）、开源的canal/open-replicator、以及MySQL自带的mysqlbinlog等仍然依赖Binlog。所以PolarDB也支持Binlog，但是其通过Logic Redo的方式将Binlog写入redo中来提升性能。

2、Binlog的记录格式

显然，server记录Binlog要尽量少，因为对数据库的修改只有在其Binlog落盘后才算成功，同时还要保证在主从上执行的同一语句的结果相同。所以Mysql提供了三种记录Binlog的格式：基于语句的（statement-based logging），基于行的（row-based logging）和混合的（mixed logging）。可以通过binlog-format指定。

在介绍Binlog类型前先说下什么是非确定性的语句（ non-deterministic），即同一条语句在集群的不同server上执行的结果不同，举个例子：UUID()，如果在某个修改操作的SQL中使用了这个语句，那么在不同server上的效果是不同的。

基于语句的方式会直接记录SQL语句，这种方式产生的Binlog少，占用磁盘空间和I/O也最少，此外主从复制的数据也少，审计数据库更改也更加方便，缺点是无法用于非确定性的语句（ non-deterministic）；基于行的方式记录了对表中某个行的修改，这种方式因为是直接复制整个行，所以可以避免上面的问题，此外需要拿的行锁也更少，缺点就是日志本身占用空间更大，采用二进制记录格式不易审计；混合的方式会根据操作类型（切换原则）切换使用这两种方式。此外DDL操作即使在row格式下也是记录SQL语句的。

所以如果存储空间和I/O不是主要问题，最好使用基于行的记录格式，因为这样更加安全。

最后放一个测试的demo，让大家更好的从具体SQL去理解这三种记录格式。本文接下来只介绍row-based logging。

3、Binlog文件里面是什么

Binlog的版本为4，以未加密的Binlog为例，布局如图1所示，Binlog的开头分别由MAGIC HEADER、FORMAT_DESCRIPTION_EVENT和PREVIOUS_GTIDS_LOG_EVENT构成。后面就是一个个其他Binlog Event了。注意是否开启GTID都是这样的布局，区别是若未开启GTID（gtid_mode=OFF），则previous_gtids_log_event和gtid_log_event记录的gtid为空。

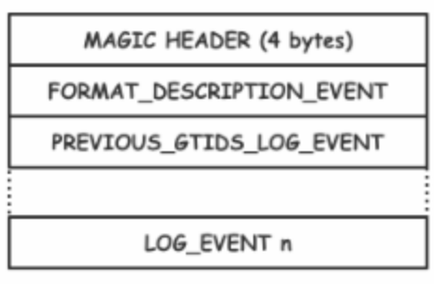

图1 开启GTID且未加密的Binlog文件

3.1、Binlog Event结构

前面说了Binlog的格式，下面说下Binlog文件的内容，首先介绍各种Binlog Event，每个Event分为Event Header、Post-Header、Payload、Event Footer四部分，如图2所示。

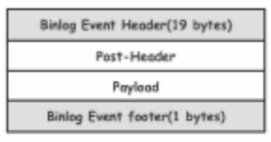

图2 单个Binlog Event结构

Event Header 内容类型一样，占用19bytes，对应类，内容如图3，图4所示：

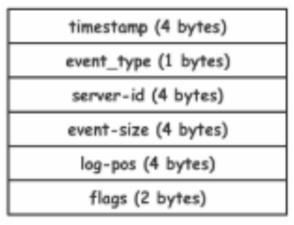

图3 Event Header 内存结构

Binlog::EventHeader:

Type	Name	Description
int<4>	timestamp	seconds since unix epoch
int<1>	event_type	See binary_log::Log_event_type
int<4>	server-id	server-id of the originating mysql-server. Used to filter out events in circular replication
int<4>	event-size	size of the event (header, post-header, body)
if binlog-version > 1 {		
int<4>	log-pos	position of the next event
int<2>	flags	See Binlog Event Header Flags

图4 Event Header 各个字段含义

Event Footer记录计算event checksum的算法信息，这个信息也记录在FDE：FORMAT_DESCRIPTION_EVENT中，同一Binlog文件中的该信息一样，对应Log_event_footer类。Post-Header、Payload分别是每个Event 类型的Header和内容实体，不同Event种类不同。下面介绍几个典型event_type。

3.2、Binlog Event类型

- FORMAT_DESCRIPTION_EVENT

该event写在Binlog文件开始4字节的位置，紧挨着magic。用于描述binlog的layout和解码Binlog Event。该类型中Post-Header、Payload指的是同样的内容。记录了Binlog Version、Mysql Server Version和Create Timestamp信息。

- PREVIOUS_GTIDS_LOG_EVENT

GTID在后面介绍，这里只用知道这个Event中涵盖了该Binlog之前所有Binlog文件中（包括已经被删除的）事务的GTID，也就是说记录了所有被执行的事务的GTID。为什么说是被执行了的？因为在Binlog rotate出新的文件前，旧文件的事务会被提交或者回滚，保留下来的一定被提交了。

- GTID_LOG_EVENT

GTID唯一的对应一个事务。因为Binlog是逻辑层日志，本身不幂等，所以为了防止一个事务被多次执行，每个事务都需要有一个全局的事务标识——Global Transaction IDentifier（GTID）。GTID由全局事务标识的UUID和递增的Group number组成。该Event中还记录了事务在不同server上的提交时间，更详细的可以查看MySQL GTID EVENT。

- QUERY_EVENT

基于statement格式的对数据修改操作都是以这种Binlog类型记录，此外，DDL也是以这种类型记录的。这里不详细展开了，详见官方文档。

- WRITE_ROWS_EVENT、UPDATE_ROWS_EVENT、DELETE_ROWS_EVENT

Row记录格式下，对表的修改会产生这些类型的Binlog。

- XID_EVENT

在XA事务commit时记录，标识事务的结尾，其中XID是事务号，由一个8位无符号整型表示。在recover时会根据它判断Binlog所记录的XA事务是否完整，注意XID和GTID所描述的不同，XID是对上层应用而言的事务号，关系到事务能否原子的执行；GTID为保证集群内的一致性，关系到事务能否在集群中的所有server上有且只有一次执行。

3.3、小结

介绍完Event类型后，很容易理解在基于Row格式记录的Binlog中，Event往往以图5这两种形式组合排布，图5（左）中的QUERY_EVENT记录的内容是执行的具体SQL，图5右中的QUERY_EVENT记录的是BEGIN，标识事务的开启。在下一节中，我会结合Binlog文件内容介绍Binlog是如何高效，安全的写入磁盘的，以及如何与物理层日志之间保持一致性。

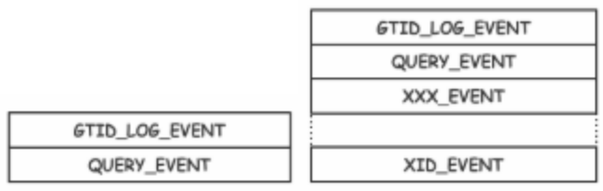

图5 DDL（左）/DML（右） Binlog事务

4、Binlog是如何写入的

由于MySQL的SQL和引擎层的双日志体系，Binlog写入需要解决多个引擎之间事务执行的一致性问题。此外，由于从日志产生到落盘是数据库写入的关键路径，所以写入的效率也是需要关注的。下面我就从这两个方面来介绍Binlog的写入过程。

4.1、分布式事务模型——XA

XA源于Distributed Transaction Processing: The XA Specification，这篇文章定义了分布式事务处理模型，其中定义了事务管理器（充当协调者），负责为事务分配标识符，监视它们在不同参与者上执行的进度，并负责事务完成和故障恢复。 还定义了资源管理器，充当参与者，受协调者管理。此外还有应用程序，充当事务的发起者。

4.1.1 MySQL中的XA类型以及协调者选择

在MySQL中，如果事务的参与者是各个实例节点，那么是外部XA，由上层程序担当协调者，上层程序可以通过XA start，XA prepre，XA end，和XA commit的命令管理事务的执行。如果事务的参与者只在单实例节点内部，那么称为内部XA，例如参与者是Binlog和innodb。对于内部XA的协调者，如果开启Binlog，则Binlog为协调者，显然选择Binlog作为协调者是最合适的，因为Binlog位于引擎层之上且还负责主备之间数据的同步。如果不开Binlog，且只有innodb一个成员，那就不需要XA了。但是如果没有Binlog且在引擎层有多个参与者，那么MySQL会使用TC_LOG_MMAP作为协调者。XA采用两阶段提交协议保证分布式事务的一致性。两阶段提交分为prepare和commit两个阶段，协议的内容参考分布式事务两阶段提交，在Prepare阶段前，进入函数ha_commit_trans。这里有个参数all。'all为false' 表示这是用户发出的显式提交，'all为true'表示是DDL 发出的隐式提交。某些DDL在执行完成后会隐式提交，也就是无需用户调用commit等结束语句而自发提交，这就意味着一条DDL是一个单独的事务，用户无法回滚它，详见Statements That Cause an Implicit Commit。如果打开了autocommit，DML也会自发提交，详见autocommit。所以XA事务有很多种情况（内部、外部、是否开Binlog、是否为DDL等），接下来主要介绍开启row_based格式的Binlog，开启GTID，存储引擎只有innodb的内部XA执行过程。以DDL和DML语句为例，整个过程如图6所示。

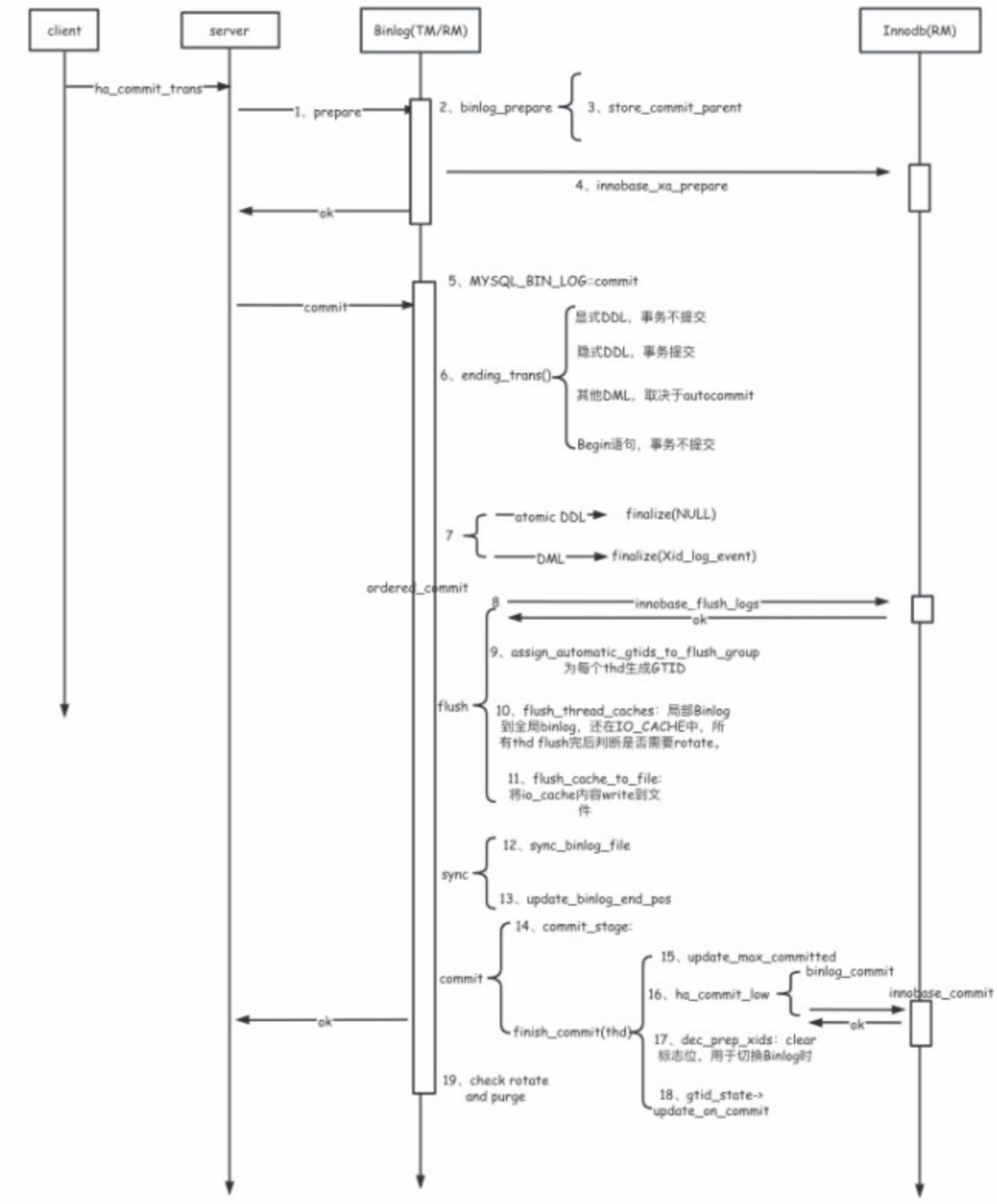

图6 由Binlog担任协调者的XA事务处理过程

4.1.2 Prepare阶段

prepare阶段分为binlog的prepare和innodb的prepare。进入binlog和innodb prepae前会设置durability_property = HA_IGNORE_DURABILITY, 表示在innodb prepare和finish_commit()时，不刷redo log到磁盘。

- Binlog Prepare

入口：binlog_prepare
对于all为false的事务，会更新该事务的last_commited为此时most recently commited事务的sequence_number，sequence_number是Binlog提交的逻辑时间戳，可用于在slave节点上并行执行Binlog事务，生成和自增策略参考Binlog事务依赖策略。

- Innodb Prepare

入口：innobase_xa_prepare
初始化Innodb事务，将事务的状态由TRX_STATE_ACTIVE设置为TRX_STATE_PREPARED，标志事务进入prepare阶段，在undo log page中写入TRX_UNDO_PREPARED状态，若有xid则会在undo中也记录xid信息。

4.1.3 Commit阶段

Commit成功意味着在当前事务中所做的更改是永久性的，并且对其他session可见。Commit阶段分为Binlog的Commit和Innodb的Commit。

- Binlog Commit

入口：binlog_commit
在Commit之前，Binlog已经写入到局部Binlog（见4.2.2），Commit时只需在结尾写入Binlog事务结尾的标识，例如XID_EVENT，在recover的时候据此判断事务的完整性。因为每条DDL都会implicit提交，所以一个DDL事务只会记录一条QUERY EVENT，所以结尾不需要记录XID_EVENT就可判断DDL事务是否完整。

随后开始提交Binlog，也就是将各个thd的Binlog事务写到Binlog文件中，Binlog文件中事务之间要彼此独立的顺序排列，不会交错，因为交错的事务难以被slave apply。然而一个一个写binlog并落盘显然效率极低，为了提高效率，MySQL采用Group Commit的方式。整体过程在网上有很多讲解，本文主要从代码层面讲解具体的几个关键函数，关于Group Commit的实现方法本文不介绍了。Group Commit分为三个阶段flush、sync、commit。在flush阶段将redo log持久化，将Binlog 写到文件系统的page cache中。在sync阶段将Binlog刷盘。下面具体介绍（序号对应图中的序号）：

（6）ending_trans()函数判断是否对本次事务进行提交，有四种情况，用户发起的一条DDL语句会分别执行显式（explicit）和隐式（implicit）的提交，若为显式，则事务不提交；若为隐式才真正提交，对DDL而言，其会在执行时将autocommit置为0，所以autocommit对DDL不生效，因为不论autocommit是否开启，DDL都会由server自发做提交（implicit commit）。若为DML，则若autocommit为1，自动提交，autocommit为0，则由用户手动提交。若为Begin语句，不论autocommit是否开启，都不提交。

（7）对于需要提交的事务，如果是DML，会在trx_cache结尾append一个Xid_log_event。随后进入ordered_commit。

（8）进入flush阶段，change_stage将线程入队。然后由leader执行process_flush_stage_queue，这里先刷innobase层的日志，也就是刷redo(innobase_flush_logs)，如果innodb_flush_log_at_trx_commit为1，则这里将redo落盘。

（9）assign_automatic_gtids_to_flush_group为每个thd生成GTID。

（10）flush_thread_caches，首先将上一步生成的GTID写到全局Binlog中，然后将局部binlog刷到全局Binlog，此时数据还在IO_CACHE结构中。thd的binlog_cache_mngr管理两种局部Binlog event缓存：stmt_cache和trx_cache，前者记录非事务性Binlog，后者记录事务型Binlog。XA事务中只有trx_cache有数据。所有局部Binlog flush完后判断是否需要rotate，若需要，将在ordered_commit最后完成。

（11）flush_cache_to_file将IO_CACHE中Binlog write到文件。

（12）进入sync阶段，sync_period用于控制sync的周期，比如经过几次flush后做一次sync。sync_binlog_file将Binlog落盘。由配置参数sync_binlog控制。

（13）如果sync_period为1，则sync_binlog_file完更新atomic_binlog_end_pos，这个参数标识binlog结尾。如果sync_period不为1，则flush完就更新atomic_binlog_end_pos。

（14）进入commit阶段，该阶段主要执行finish_commit，如果opt_binlog_order_commits==false，那么事务就不按照之前的顺序，各自进行提交(finish_commit)，这种情况下不能保证innodb commit顺序和binlog写入顺序一致，这不会影响到数据一致性，在高并发场景下还能提升一定的吞吐量。但可能影响到物理备份的数据一致性，例如xtrabackup（而不是基于其上的innobackup脚本）依赖于事务页上记录的binlog的end位点（flush_thread_caches会更新），如果位点发生乱序，就会导致备份的数据不一致。

（15）执行finish_commit， update_max_committed更新最大commit事务的序号。

（16）分别执行Binlog和innodb的commit。Binlog Commit在前面已经完成了，所以这里什么也不做。实际只有Innodb的Commit。

（17）dec_prep_xids: 清除 m_atomic_prep_xids，rotate Binlog时通过它判断当前Binlog是否有正在提交的事务。

（18）将commit事务的GTID加入executed_gtids。

（19）在第10步判断的，如果Binlog文件大小超过了max_binlog_size，则会rotate新的Binlog。

- Innodb Commit

入口：innobase_commit

将undo头的状态修改为TRX_UNDO_CACHED或TRX_UNDO_TO_FREE或TRX_UNDO_TO_PURGE (undo相关知识参阅之前的月报)；并释放事务锁，清理读写事务链表、readview等一系列操作。每个事务在commit阶段也会去更新事务页的binlog位点。然后根据该session已执行的GTID去更新全局GTID SET。在8.0.17版本会将GTID持久化到undo日志中（原因）。

4.2、写入效率

本节介绍Binlog写入之前，先介绍IO_CACHE结构，该结构贯穿了任何与Binlog相关文件（index文件，purge_index_file，crash_safe_index_file等）的读写过程，随后介绍XA过程中局部的Binlog和全局的Binlog。最后介绍仍然存在的性能瓶颈和解决方案。

4.2.1 IO_CACHE

文件系统虽然向上呈现一段连续的空间，但是其内部以页的形式管理，页的大小通常为4K，满足4K对齐的读写对文件系统的性能会有很大的提高。而IO_CACHE的作用就是充当一层缓存，将连续的数据写入进行4K对齐后写入文件系统。

知道了IO_CACHE的作用后，来看看其Binlog是如何利用它的。Binlog文件初始化的过程如下：

```
class IO_CACHE_ostream {
    bool IO_CACHE_ostream::open() {
        file = mysql_file_open(log_file_key, file_name, O_CREAT | O_WRONLY, MYF(MY_WME));
        init_io_cache(&m_io_cache, file, cache_size, WRITE_CACHE, 0, 0, flags);
    }
    IO_CACHE m_io_cache;
}

init_io_cache
|
---->init_io_cache_ext(){
    info->file = file;
    info->buffer = (uchar *)my_malloc(key_memory_IO_CACHE, buffer_block, flags);
    init_functions(info);
}
```

可以看出MySQL在打开Binlog文件后将文件描述符交给IO_CACHE结构管理，IO_CACHE初始化过程中，会申请一个缓冲，默认大小是8K，随后计算读写缓冲区的位点以便对齐写入，还定义了对IO_CACHE的读写函数。IO_CACHE详见IO_CACHE源码解析。如图7所示，IO_CACHE会对齐PageCache进行写入，满足对齐条件后就会刷到Page Cache中，之后sync到Binlog文件。

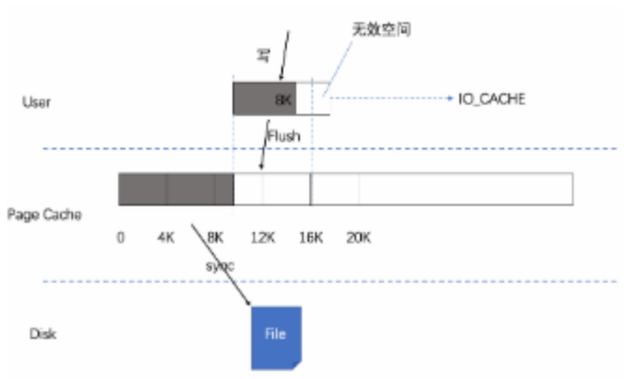

图7 数据写IO_CACHE的过程

4.2.2 局部Binlog和全局Binlog

Binlog中的事务是顺序独立的，不能交错，原因是交错的Binlog事务无法被slave重放。但是多个客户端连接MySQL，并对其并发写入的场景经常出现。为了解决高并发过程中顺序写入的问题，MySQL为每个连接都配置了一个局部的Binlog文件，各个连接产生的Binlog会事先写到各个局部Binlog中，等到group commit时再将各个局部Binlog合并到全局Binlog文件中。

- 局部Binlog

局部Binlog通过Binlog_cache_storage结构管理，实际上也是对IO_CACHE结构的包裹，可以通过binlog_cache_size来控制它的大小，如果事务的binlog日志大小超出了binlog_cache_size的定义的大小，多出来的部分会存在临时文件中，但是事务总大小不能超过max_binlog_cache_size。上面我们说到IO_CACHE在初始化的时候会关联一个磁盘文件，这里也不例外，但是这里特殊在是临时文件，通过下面这个函数创建。

```
bool real_open_cached_file(IO_CACHE *cache) {
  if ((cache->file = mysql_file_create_temp(
         cache->file_key, name_buff, cache->dir, cache->prefix,
         (O_RDWR | O_TRUNC), MYF(MY_WME))) >= 0) {
    error = 0;
    /*
      Remove an open tempfile so that it doesn't survive
      if we crash.
    */
    (void)my_delete(name_buff, MYF(MY_WME));
  }
}
```

该文件以"ML"为前缀，如果创建成功，则会被立刻删除，但是由于文件的描述符并没有被释放，所以该文件依然能被读写，当程序crash后，该文件会被真正的删除。由于其中保留的数据未提交，所以重启后无需恢复，其实删除就是最好的恢复。

- 全局Binlog

全局Binlog是当前打开的Binlog，在MySQL启动时构造在m_binlog_file变量中，管理Binlog写入流。其底层依然是通过IO_CACHE管理文件写入的。

- group commit

在SQL执行的过程中，Binlog会伴随着产生并写入到局部Binlog中，在xa事务提交时，局部Binlog中的事务会被顺序拷贝到全局Binlog中。关于Group Commit可以参考4.1.3和MySQL组提交，图8展示了Binlog Event写入局部Binlog，并在提交时由局部Binlog拷贝至全局Binlog的过程。

```
/*将局部Binlog拷贝到全局Binlog*/
bool MYSQL_BIN_LOG::do_write_cache(Binlog_cache_storage *cache,
                                   Binlog_event_writer *writer) {
    cache->copy_to(writer, &error)
}
```

图8 Binlog Event写入局部Binlog和全局Binlog

4.2.3 性能问题

开启Binlog后的性能一直被诟病，对于AWS Aurora在开启Binlog后，通常有50%到60%的性能损耗；对于PolarDB也差不多是这个数值；Oracle则不写入Binlog，而是通过物理redo日志去生成Binlog。即使Binlog会带来如此严重的性能问题，但它仍然在业务中不可或缺的。所以数据库厂商采取了一些方法去解决这些问题。本节将介绍PolarDB和Aurora是如何提升Binlog性能的。

- PolarDB

考虑到在Group Commit的过程中，在flush阶段redo会被sync到磁盘，在sync阶段Binlog会被sync到磁盘，这两个过程是串行的，两次对云盘的写入会造成很大的性能损耗，所以PolarDB采用logic redo的方法，将Binlog数据记录到redo日志中，在sync阶段，将redo和Binlog一起刷盘。关于Logic的详细介绍，可以移步Logic redo。

- Aurora

Aurora面临的问题一样，瓶颈依然在全局Binlog sync到磁盘的时候。但是它的切入点和PolarDB不同，它的做法是enhance Binlog：将全局Binlog的sync过程打散到SQL执行的过程中，将局部Binlog下推到存储节点，这样在SQL执行过程中，Binlog就向存储节点写入，等到最后提交时，只需要存储节点对这些局部Binlog进行合并即可。详见
AWS re:Invent2022 Aurora 发布了啥

5、Binlog Recover

说完了Binlog写入过程，很容易想到如果写入过程中程序崩溃了怎么办，所以下面将介绍Binlog的Recover过程。Recover是基于xa过程的，本质上是根据已经落盘的Binlog决定如何处理未提交的事务。因为rotate新的Binlog时会recover老的Binlog中所有事务，因此在Binlog启动时，只需对最新的Binlog文件执行Recover即可，对于某个事务而言，如果它记录的Binlog是完整的（关于完整的Binlog事务参考第三章小结部分），说明它可以提交，反之，如果缺失任意一条Event都是不完整的Binlog，不完整的Binlog会被删除，与之关联的事务（binlog事务，innodb的事务）都会回滚。

- XID

在前面介绍Binlog事务和XA过程的时候，可知每个Binlog事务都有个对应XID，对于非DDL Binlog事务，XID会以XID_EVENT的类型在事务提交时写到事务结尾；对于DDL事务，XID包含在DDL所在的QUERY_EVENT里。这个XID其实是xa事务的id，唯一标识每个xa事务。在xa过程的Innodb prepare时，会设置事务的状态为prepare，并记录在undo page中。这样Binlog recover时候根据xid能去Innodb层找哪些事务是prepare状态的，对这些事务提交或回滚。

- recover

入口：int MYSQL_BIN_LOG::open_binlog
打开index 文件中最后一个 binlog，若该文件没有正常关闭（LOG_EVENT_BINLOG_IN_USE_F 置位），则recover它。从头开始，挨个扫描每个Binlog Event，只要发现某个Binlog事务不完整，那么该Binlog和其后面的Binlog都会被truncate。前面完整的Binlog的事务依据它们的xid去innodb层提交，其他事务进行回滚。原因是事务的binlog已经完整落盘，所以redolog也落盘了，该事务是可提交的。至此Binlog的recover完成，但是为了体现Binlog是参与者，之后会调用空函数binlog_dummy_recover()，该函数为空，因而后续的也不会调用commit和rollback函数。实际只进行innodb的recover。
innodb的recover函数为innobase_xa_recover()，函数的主要目的是找到innobase层所有prepare状态的事务，这些事务的XID与前面Binlog找到的XID进行比对，从而决定哪些需要回滚innobase_rollback_by_xid，哪些需要提交innobase_commit_by_xid。提交和回滚可参考innodb事务系统。
最后，Binlog会truncate到保留最后一个完整的事务，清除LOG_EVENT_BINLOG_IN_USE_F，表示binlog文件正常关闭，并rotate出一个新的Binlog进行写入。
recover讲完，primary上的Binlog基本讲完了，下面将介绍Binlog是如何完成数据同步的——Binlog复制。

6、Binlog复制

首先需要建立连接，MySQL将对应的连接称为channel，slave节点通过change master指令可以与master建立一个channel并对其命名，change master指令可以指定复制开始文件和位点。随后由slave发起start slave开启复制，slave可以为所有channel都开启复制（start slave）,也可以只为特定的channel开启复制（start slave for channel 'channel_1'），开启复制是通过建立复制Binlog的IO线程和对其回放的SQL线程，本文不讨论SQL线程。下面来看看连接建立与复制过程，如图9所示。

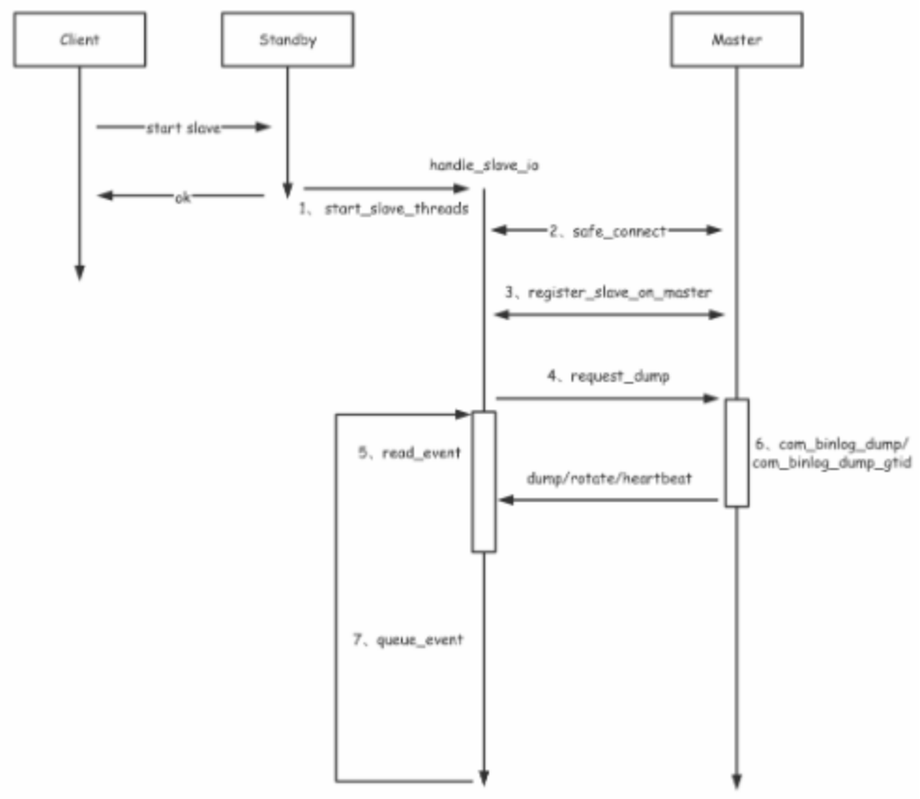

图9 连接建立与复制过程

（1）这里根据设置的thread_mask会启动相应的线程：handle_slave_io或handle_slave_sql线程。这里介绍handle_slave_io线程。handle_slave_sql是slave回放binlog的线程，执行完线程启动后，在handle_slave_io线程初始化完并被加到thd_manager后，客户端就能收到该指令执行的响应了。连接master的操作在后续执行。
（2）slave通过safe_connect->connect_to_master与master建立连接。
（3）slave向master发送COM_REGISTER_SLAVE指令，在master端register_slave，检查slave的权限，将其serverid、host、user、passwd等信息放在slave_list结构中。
（4）slave发起dump请求，command是COM_BINLOG_DUMP，若开启gtid和auto_position(GTID Auto-Positioning)，通过设置gtid_mode=ON和在change master时指定MASTER_AUTO_POSITION=1，则使用GTID复制，command是COM_BINLOG_DUMP_GTID，区别是后者会发送slave上的m_exclude_gtid（slave上已有的Binlog事务gtid集合），master只会复制不在该gtid集合中的Binlog事务。
（5）read_event调用mysql_binlog_fetch读取从master发来的packet，阻塞等待。
（6）master响应request_dump()发起的COM_BINLOG_DUMP(_GTID)请求，如图10所示。

- 首先初始化Binlog sender，如果slave未指定复制起始位点（change master指令可指定位点，还有是位点会保存在master.info文件，由slave启动时读取。），则在sender init时初始化位点为index文件中第一个binlog文件的第一个event位置（pos=4）。
- 然后打开Binlog文件send_binlog，在该函数中：1）函数get_binlog_end_pos会判断当前正在复制的Binlog文件是否和全局Binlog文件相同，如果不同说明该文件不是最后一个Binlog，复制完该文件后需要rotate到下一个继续复制。如果相同，则复制完后会等待该Binlog文件中新的写入（end_pos更新）。2）函数send_events()一次读一个完整的event并发送，如果开启GTID和auto_position，则不发送gtid包含在m_exclude_gtid中的事务。如果没有event发送，则会等待超时并发送heartbeat event，heatbeat还可用于告知slave：master复制位点；

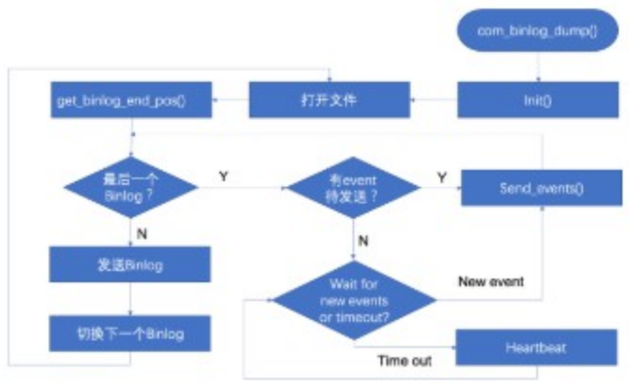

图10 master响应com_binlog_dump

（7）来自master的Binlog会被存储到slave的relay log中，通常slave的Binlog被称为relay log。后续SQL线程会解析并应用relay log。

杂记——Binlog相关文件

1、Binlog文件

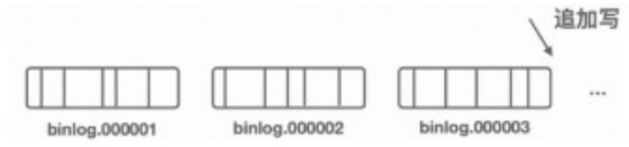

图11 Binlog文件

Binlog文件命名由log_bin_log或log-bin指定，这里假定为binlog，后面的例子中也一样，如图11所示，写入方式为顺序追加写。通过mysqlbinlog可以查看文件内容。

2、Index文件

每行都是Binlog文件名。如图12所示。

图12 Binlog Index文件内容

3、crash_safe_index_file

临时文件，内容为Binlog文件名。保证了修改index文件时，写入的Binlog文件名是原子的，图13是在index文件中写文件名的一个例子，可以很容易看出crash_safe_index_file的功能。该文件命名方式为：./binlog.index_crash_safe

图13 在index文件中写文件名的过程

4、purge_index_file

临时文件，内容为Binlog文件名。保证index内容和Binlog文件互相匹配。图14展示的是新建Binlog文件时需要在purge_index_file文件中写入一条新的文件名，在Binlog文件创建完成后将该Binlog文件名写入index文件，随后删除purge_index_file。由此可见该文件中记录的Binlog文件名都是在创建过程中并且还未来得及被记录在index文件的Binlog文件，所以每次打开index文件时，会检查并删除purge_index_file中记录的Binlog文件，避免index文件和Binlog文件不匹配。该文件的命名方式为：./binlog.~rec~

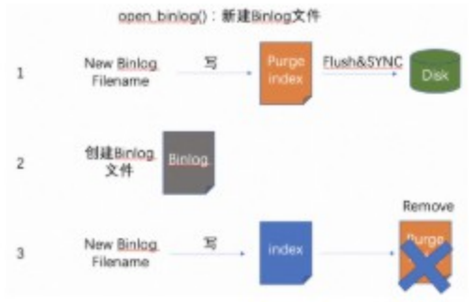

图14 新建Binlog文件过程

笔者也在学习过程，如有任何问题欢迎评论和私信进行指正和讨论。

其他参考资料

[1] 物理复制解读

[2] MySQL Replication Events – Statement versus Row-Based Formats

[3] Mysql Binlog Event

[4] InnoDB 事务子系统介绍