# 背景

二进制日志包含数据库的所有更改记录，包括数据和结构两方面。二进制日志不记录SELECT 或SHOW 等不修改数据的操作。运行带有二进制日志的服务器会带来轻微的性能影响。二进制日志能保证数据库出故障时数据是安全的。只有完整的事件或事务会被记录或回读。

为什么应该使用二进制日志？

● 复制：使用二进制日志，可以把对服务器所做的更改以流式方式传输到另一台服务器上。从（slave）服务器充当镜像副本，也可用于分配负载。接受写入的服务器称为主（master）服务器。

● 时间点恢复：假设你在星期日的00：00进行了备份，而数据库在星期日的08：00出现故障。使用备份可以恢复到周日00：00的状态；而使用二进制日志可以恢复到周日08：00的状态。

# 概述

## 简介

Binlog它记录了所有的**DDL和DML**(除了数据查询语句、SHOW)语句，以**事件（EVENT）**形式记录，还包含语句所执行的消耗的时间，MySQL的二进制日志是**事务安全型**的。

注意：**对于DELETE删除了0行记录的情况，STATEMENT方式会记录该条记录，但是ROW方式不会记录**。

可在MySQL客户端执行SHOW BINLOG EVENTS IN ‘mysql-bin.000010’\G命令查看该binlog记录了哪些事件，基本格式如下：

Log\_name:musql-bin.000010

Pos:2324

Event\_type:Query

Server\_id:285326927

End\_log\_pos:2450

Info:use `json`;ALTER TABLE feature ADD INDEX(feature\_street)

一般来说开启二进制日志大概会有1%的性能损耗（MySQL官方测试数据），虽然会降低性能，但是binlog可以用于主从复制（replication）和point-in-time的恢复，所以还是很有必要开启的。

参考：

<https://database.51cto.com/art/202009/626540.htm>

<https://www.jianshu.com/p/e051000e0cce?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation>

**查看：**

SHOW BINLOG EVENT IN ‘mysqld.00001’\G;

**配置：**

通过参数log-bin[=name]可以开启二进制日志，如果不指定name，默认二进制日志文件名为主机名，后缀名为二进制日志的序列号，所在路径为数据库所在目录（datadir）。

## 功能

一般用于：复制、增量恢复

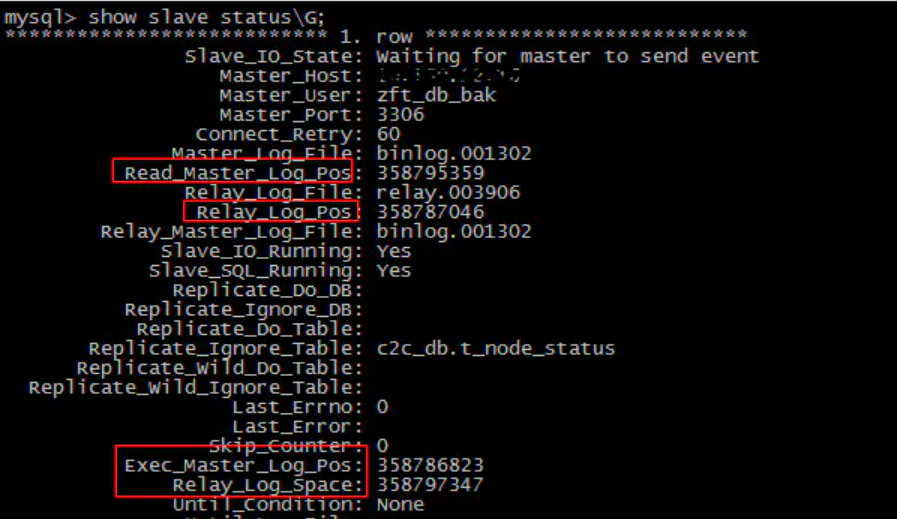
## at position

用mysqlbinlog解析出来的binlog很容易看到如下内容：



这里的at 259的259指的是binlog的第259字节,因而在默认max\_binlog\_size=1G的配置下，binlog几乎都是# at 4开头，以# at 1073744392左右结尾(1G大小)。

顺带说明下在备机show slave status\G;的\*Log\_Pos代表的也是那个binlog里的字节：

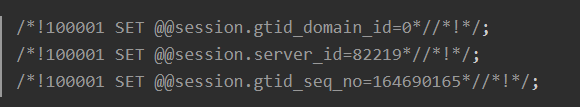


## gtid

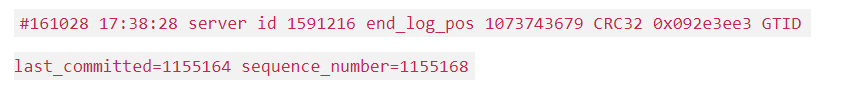
全局唯一事务号，没什么可说的，两个不同事务的gtid必定不相同，MySQL官方版本binlog中形式这样：



MariaDB的gtid形式就有些不一样了，binlog中会这样记录:



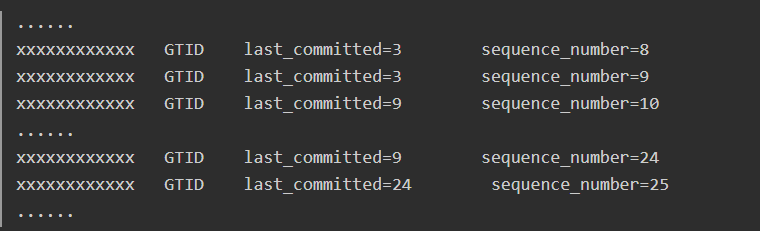
## sequence\_number



这个在每个binlog产生时从1开始然后递增，每增加一个事务则sequencenumber就加1，你可能好奇有了gtid何必多此一举再加个sequencenumber来标识事务呢，请看下面：

## lastcommitted

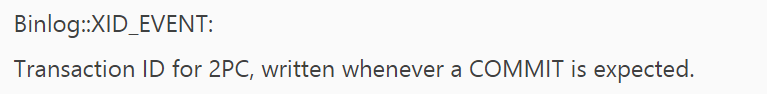
这个在binlog中用来标识组提交，同一个组提交里多个事务gtid不同，但lastcommitted确是一致的，MySQL正是依据各个事务的lastcommitted来判断它们在不在一个组里；一个组里的lastcommitted与上一个组提交事务的sequencenumber相同，这样sequencenumber就必须存在了：



这代表sequencenumber=10到sequencenumber=24的事务在同一个组里(因为lastcommitted都相同,是9)  
 注意：组提交只与lastcommitted有关，这也是MySQL基于组提交(logic clock)的并行复制方式即使在gtid关闭情形下也能生效的原因。

## xid

根据官方文档说明，这是用来标识xa事务的



# 分类

## 二进制日志索引文件

## 二进制日志文件

二进制日志包括两类文件：二进制日志索引文件（文件名后缀为.index）用于记录所有的二进制文件，二进制日志文件（文件名后缀为.00000\*）记录数据库所有的DDL和DML(除了数据查询语句)语句事件。

# 日志格式

binlog有三种格式：Statement, Row和Mixed.

基于SQL语句的复制（statement-based replication, SBR）

基于行的复制（row-based replication, RBR）

混合模式复制（mixed-based replication, MBR）

## Statement

记录的是**逻辑SQL**，每一条修改操作的sql都会记录在binlog中。

**优点：**不需要记录每一行的变化，减少了binlog日志量，节约了IO, 提高了性能。

**缺点：**无法完全保证slave节点与master节点数据完全一致。

由于记录的只是执行语句，为了这些语句能在slave上正确运行，因此还必须记录每条语句在执行的时候的一些相关信息，以保证所有语句能在slave得到和在master端执行的时候相同的结果。另外mysql的复制，像一些特定函数的功能，slave可与master上要保持一致会有很多相关问题。

相比row能节约多少性能与日志量，这个取决于应用的SQL情况，正常同一条记录修改或者插入row格式所产生的日志量还小于statement产生的日志量，但是考虑到如果带条件的update操作，以及整表删除，alter表等操作，row格式会产生大量日志，因此在考虑是否使用row格式日志时应该根据应用的实际情况，其所产生的日志量会增加多少，以及带来的IO性能问题。

## Row

5.1.5版本的MySQL才开始支持row level的复制，它不记录sql语句上下文相关信息，仅**保存哪条记录被修改**。

**优点：**binlog中可以不记录执行的sql语句的上下文相关的信息，仅需要记录那一条记录被修改成什么了。所以row的日志内容会非常清楚的记录下每一行数据修改的细节。而且不会出现某些特定情况下的存储过程，或function，以及trigger的调用和触发无法被正确复制的问题。

**缺点：**所有的执行的语句当记录到日志中的时候，都将以每行记录的修改来记录，这样可能会产生大量的日志内容。

新版本的MySQL中对row level模式也被做了优化，并不是所有的修改都会以row level来记录，像遇到表结构变更的时候就会以statement模式来记录，如果sql语句确实就是update或者delete等修改数据的语句，那么还是会记录所有行的变更。

## Mixed

从5.1.8版本开始，MySQL提供了Mixed格式，实际上就是Statement与Row的结合。

在Mixed模式下，（默认情况下）一般的语句修改使用statment格式保存binlog，如一些函数，statement无法完成主从复制的操作，则采用row格式保存binlog，MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种。

**会采用row格式的情况包括：**

1、表的存储引擎为NDB，这时对表的DML操作都会以ROW格式记录；

2、使用了UUID()、USER()、CURRENT\_USER()、FOUND\_ROWS()、ROW\_COUNT()等不确定函数；

3、使用了INSERT DELAY语句；

4、使用了用户定义函数（UDF）；

5、使用了临时表（temporary table）。

注：不需要死记硬背这些情况会使用row，之所以不使用statement模式，就是因为简单的逻辑SQL无法实现数据的回放，比如使用UUID()，这个是随机的，无法保证每次结果都一样，为了保证主从复制数据一致性，则必须是记录数据的变化信息。

**Mixed格式理论上无误，实际上是有问题的，会出现级联复制和大量数据增删改，且有不确定DML语句的时候，会出现binlog丢失。**

此外，binlog\_format参数还有对于存储引擎的限制：



在通常情况下，我们将参数binlog\_format设置为ROW，这可以为数据库的恢复和复制带来更好的可靠性。但是不能忽略的一点是，这会带来二进制文件大小的增加，这些语句的ROW格式可能需要更大的容量。而由于复制是采用传输二进制日志方式实现的，因此复制的网络开销也会增加。

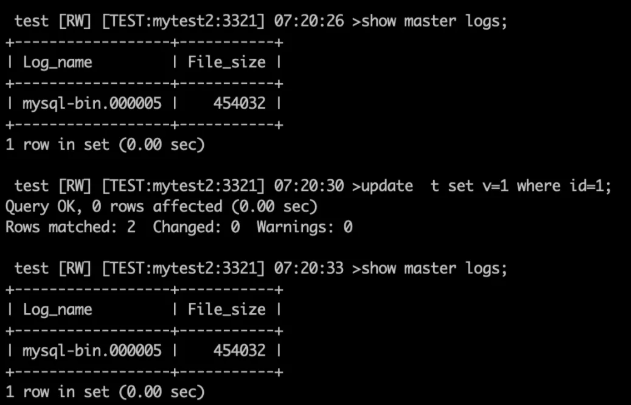
要查看二进制日志文件内容，必须通过MySQL提供的工具mysqlbinlog。对于STATEMENT格式的二进制日志文件，在使用mysqlbinlog后，看到的就是执行的逻辑SQL语句。但是，如果使用ROW格式记录，会发现mysqlbinlog的结果变得“不可读”，其实只要加上参数-v或-vv就能清楚地看到执行的具体信息了（-vv会比-v显示更新的类型）。

注：要区分行的格式和binlog的格式。

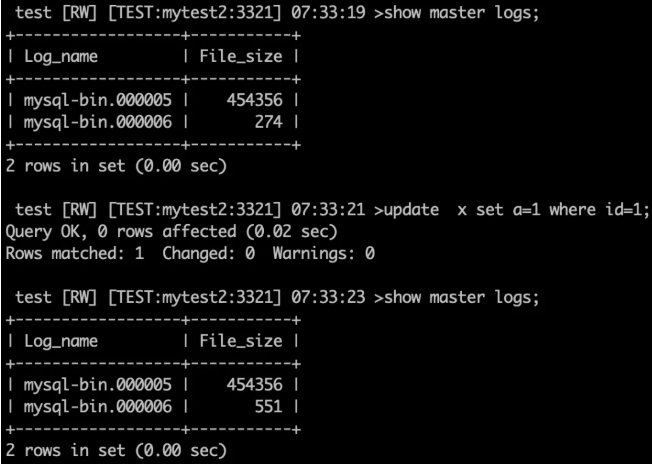
## 区别

**update字段为相同的值是否会记录binlog？**

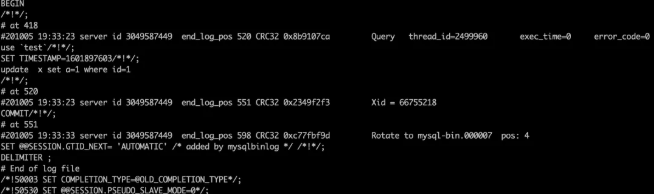
binlog\_format为ROW模式



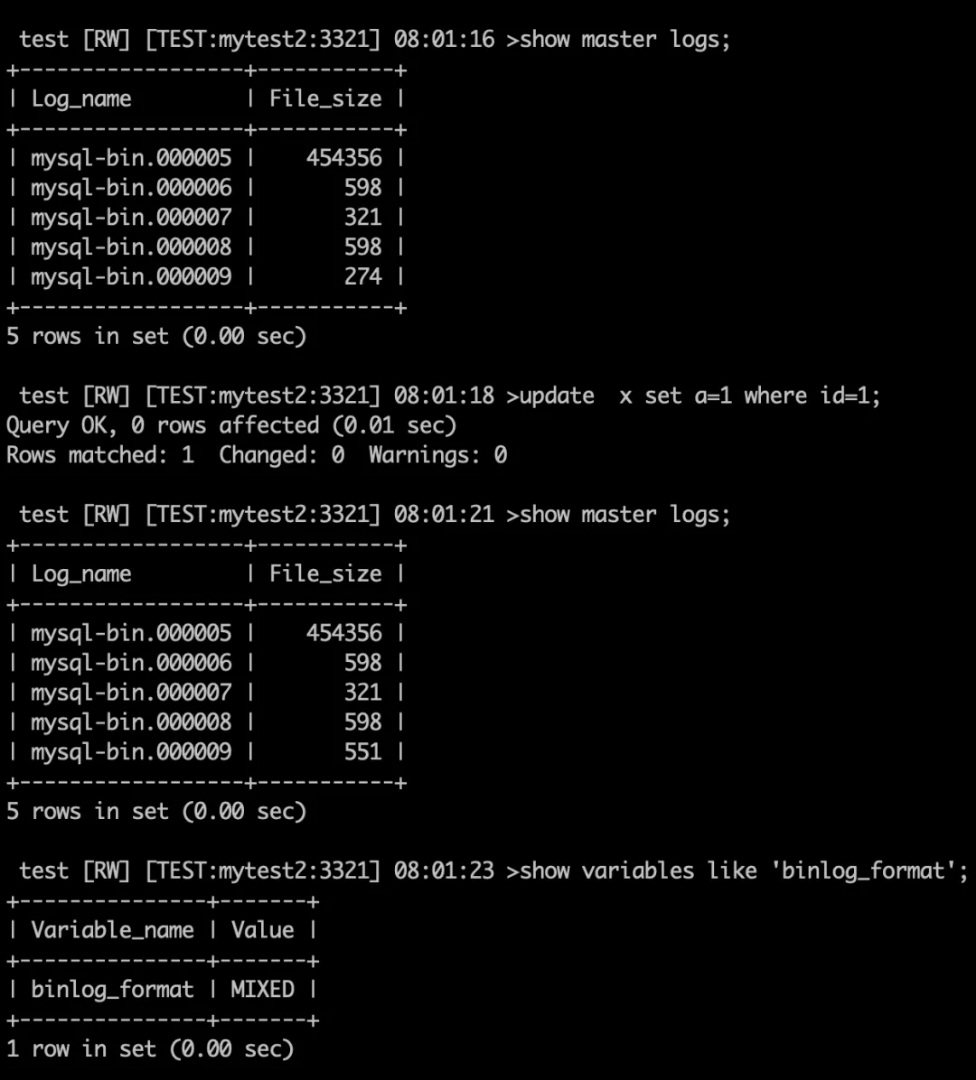
binlog\_format为STATEMENT模式



解析binlog内容，完整的记录了update语句。



binlog\_format为MIXED模式



当 row\_format 为mixed或者statement格式是，binlog 的大小发生改变，不管是否真的更新数据，MySQL都记录执行的sql 到binlog。

**总结：**

基于row模式时，server层匹配到要更新的记录，发现新值和旧值一致，不做更新，就直接返回，也不记录binlog。

基于 statement 或者 mixed格式，MySQL执行 update 语句，并把更新语句记录到binlog。

# 8.0新特性

## binlog压缩

### 概述

二进制日志（binlog）是MySQL日志结构中重要的部分；记录了数据的更改操作，用于数据恢复、数据复制以及审计。然而在众多实际场景中经常发生高并发引起binlog暴涨的问题将挂载点空间占满以及主从网络带宽成为瓶颈时主从延时过大。8.0.20版本推出binlog压缩功能，有效缓解甚至解决此类问题。

MySQL从8.0.20开始集成ZSTD算法，开启压缩功能后；以事务为单位进行压缩写入二进制日志文件，降低原文件占用的磁盘空间。压缩后的事务以压缩状态有效负载在复制流中发送到从库（MGR架构中为组member）或客户端（例如 mysqlbinlog）。

官网连接：<https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-20.html>

### 原理

1、开启压缩功能后，通过ZSTD算法对每个事务进行压缩，写入二进制日志。

2、新版本更改了libbinlogevents，新增Transaction\_payload\_event作为压缩后的事务表示形式。

class Transaction\_payload\_event : public Binary\_log\_event {

protected:

const char \*m\_payload;

uint64\_t m\_payload\_size;

transaction::compression::type m\_compression\_type;

uint64\_t m\_uncompressed\_size;

3、新增Transaction\_payload\_event编码器/解码器，用于实现对压缩事务的编码和解码。

namespace binary\_log {

namespace transaction {

namespace compression {

enum type {

/\* No compression. \*/

NONE = 0,

/\* ZSTD compression. \*/

ZSTD = 1,

};

4、在mysqlbinlog中设计和实现每个事务的解压缩和解码，读取出来的日志与未经压缩的原日志相同，并打印输出所用的压缩算法，事务形式，压缩大小和未压缩大小，作为注释。

#200505 16:24:24 server id 1166555110 end\_log\_pos 2123 CRC32 0x6add0216 Transaction\_Payload payload\_size=863 compression\_type=ZSTD uncompressed\_size=2184

# Start of compressed events!

5、从库（或MGR-member）在接收已压缩的binlog时识别Transaction\_payload\_event，不进行二次压缩或解码。以原本的压缩状态写入中继日志；保持压缩状态。回放日志的解码和解压缩过程由SQL线程负责。

总结日志压缩过程为：

1）单位事务需要提交并记录 binlog。

2）压缩编码器在缓存中通过ZSTD算法压缩以及编码该事务。

3）将缓存中压缩好的事务写入日志中，落盘。

日志读取过程为：

客户端工具（mysqlbinlog、sql 线程）对压缩日志进行解压缩、解码。解压出原本未压缩的日志进行读取或回放。

### 注意事项

1、压缩功能以事务为单位进行压缩，不支持非事务引擎。

2、仅支持对ROW 模式的binlog进行压缩。

3、目前仅支持ZSTD压缩算法，但是，底层设计是开放式的，因此后续官方可能会根据需要添加其他压缩算法（例如zlib或lz4）。

4、压缩动作是并行进行的，并且发生在binlog落盘之前的缓存步骤中。

5、压缩过程占用本机CPU及内存资源。在主从延迟的场景中，如果性能瓶颈时，网络带宽、压缩功能可以有效缓解主从延迟；但是如果性能瓶颈是本机自身处理能力，那么压缩功能反而可能加大主从延迟。

### 特性测试

MySQL版本：8.0.20

架构：一主一从半同步

测试方案：

1、搭建好 MySQL 8.0.20 的主从架构

2、主从上开启压缩功能、并设置压缩等级，默认为 3，随着压缩级别的增加，数据压缩率也会增加，但同时 CPU 及内存的资源消耗也将增加。

mysql> set binlog\_transaction\_compression=on;

mysql> set binlog\_transaction\_compression\_level\_zstd=10;

3、查看压缩前后相同SQL产生的binlog大小。

压缩前binlog大小约为300M

-rw-r----- 1 mysql mysql 251M May 6 09:31 mysql-bin.000001

-rw-r----- 1 mysql mysql 50M May 6 09:31 mysql-bin.000002

压缩后binlog大小约150M

-rw-r----- 1 mysql mysql 148M May 6 09:32 mysql-bin.000004

4、查看压缩前后相同SQL在低主从带宽的网络环境中tps的比较。

限制网络速率

tc qdisc add dev eth0 root handle 1:0 netem delay 100ms

tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 256kbit buffer 1600 limit 3000

压缩前压测结果：

SQL statistics:

queries performed:

read: 3976

write: 1136

other: 568

total: 5680

transactions: 284 (9.17 per sec.)

queries: 5680 (183.32 per sec.)

压缩后压测结果：

SQL statistics:

queries performed:

read: 4746

write: 1356

other: 678

total: 6780

transactions: 339 (10.15 per sec.)

queries: 6780 (202.92 per sec.)

**结论**

1、MySQL新推出的binlog压缩功能，当压缩级别设置为10时，压缩率约为50%左右，能够较大程度减少binlog所占用的空间。

2、压缩功能能够一定程度提升因网络带宽所带来的主从延迟，集群tps不降低，略微提升。

# 参数

## max\_binlog\_size

max\_binlog\_size：指定了单个二进制日志文件最大值，如果超过该值，则产生新的二进制日志文件后缀名+1，并记录到.index文件。

需要注意的是由于一个事务不会跨越两个binlog文件，所以存在超大事务时，binlog文件会超过max\_binlog\_size。

flush logs也会关闭当前的binlog，并重新生成一个编号+1的binlog文件。

每当mysql服务重启时，会自动执行此命令，刷新binlog日志；在mysqldump备份数据时加上-F选项也会刷新binlog日志。

## binlog\_cache\_size

binlog\_cache\_size：控制缓冲大小，默认大小32K，**基于会话**的（即每个session都会分配内存），因此每开启一个事务就分配一个binlog\_cache\_size大小的缓存，所以不能设置过大。当一个事务的记录大于binlog\_cache\_size时，MySQL会把缓冲中的日志写入一个临时文件中，因此该值又不能设置太小。

通过SHOW GLOBAL STATUS命令查看binlog\_cache\_use、binlog\_cache\_disk\_use的状态，可以判断当前binlog\_cache\_size的设置是否合理。binlog\_cache\_use记录了使用缓冲写二进制日志的次数，binlog\_cache\_disk\_use记录了使用临时文件写二进制日志的次数。

## max\_binlog\_cache\_size

max\_binlog\_cache\_size默认值是18446744073709547520，这个值很大，够我们使用的，表示的是所有binlog能够使用的最大cache内存大小。

当我们执行多语句事务的时候，所有session的使用内存binlog\_cache\_size超过max\_binlog\_cache\_size的值就会报错：“Multi-statement transaction required more than max\_binlog\_cache\_size bytes of storaged”。

## binlog\_stmt\_cache\_size

binlog\_stmt\_cache\_size：为每个session分配的内存，在事务过程中用来存储每一条非事务性语句的缓存。和binlog\_cache\_size分别分配。

## max\_binlog\_stmt\_cache\_size

max\_binlog\_stmt\_cache\_size：与max\_binlog\_cache\_size类似

## sync\_binlog

sync\_binlog：表示每写缓冲多少次就要同步到磁盘，这个参数**直接影响MySQL的性能和完整性**。

sync\_binlog=0：当事务提交后，MySQL仅仅是将binlog\_cache中的数据写入binlog文件，但是不执行fsync之类的磁盘同步指令通知文件系统将缓存刷新到磁盘，而让Filesystem自行决定或者cache满了才同步，这个是性能最好的，但是风险也是最大的。因为一旦系统crash，在binlog\_cache中的所有binlog信息都会被丢失。

sync\_binlog=n，在进行n此事务提交以后，MySQL将执行一次fsync之类的磁盘同步指令，同时文件系统将binlog文件缓存刷新到磁盘。配置为1是最安全的，但是也是最影响性能的、当设置为1的时候，即使系统crash，也最多丢失binlog\_cache中未提交的一个事务，对实际数据没有任何实质性影响。

涉及到复制的时候，把sync\_binlog设置为1，由于是先写binlog，然后commit，如果在两步之间系统crash了，仍然会出现binlog和实际数据不一致，原因在于binlog和redo之间的数据一致性。

如果设置为1，表示采用同步写磁盘的方式来写二进制日志，这时候写操作不使用操作系统的缓冲来写二进制日志。

sync\_binlog的默认值为0，如果使用InnoDB存储引擎进行复制，并且想得到最大的可用性，建议将该值设置为ON（对数据库IO系统带来一定影响）。

但是，即使将sync\_binlog设置为1，还是会有一种情况导致问题发生。当使用InnoDB存储引擎时，在一个事务发出COMMIT动作之前，由于sync\_binlog为1，因此会将二进制日志立即写入磁盘。如果这时写入了二进制日志，但是提交还没有发生，并且此时发生了宕机，那么在MySQL数据库下次启动时，由于COMMIT操作并没有发生，这个事务被回滚。但是二进制日志已经记录了该事物信息，不能被回滚。这个问题可以通过参数innodb\_support\_xa设置为1来解决，虽然innodb\_support\_xa与XA事务有关，但是它同时也确保了二进制日志和InnoDB存储引擎文件的同步。

## binlog-do-db

binlog-do-db：表示需要写入哪些库的日志，默认为空，表示需要同步所有库的日志到二进制日志。

## binlog-ignore-db

binlog-ignore-db：表示需要忽略写入哪些库的日志，默认为空，表示需要同步所有库的日志到二进制日志。

## replicate-do-db

## replicate-ignore-db

## log-slave-update

log-slave-update：如果当前数据库是复制中的slave节点，则它不会将从master取得并执行的二进制日志写入自己的二进制文件中。如果需要写入，要设置log-slave-update。如果需要搭建master->slave->slave这种架构的复制，则必须设置该参数。

## relay-log-recovery

relay-log-recovery：slave重启是丢弃所有的relay日志，重新从master开始同步。

## slave-skip-errors

slave-skip-errors：当slave在复制时发生错误就会停止，该配置项指定slave复制语句时忽略这些错误：

1007：数据库已经存在，创建数据库失败

1008：数据库不存在，删除数据库失败

1050：数据表已经存在，创建数据表失败

1060：数据表不存在，删除数据表失败

1061：字段重复，导致无法插入

1062：重复键名

1068：定义了多个主键

## binlog\_format

binlog\_format：记录二进制日志的格式。在MySQL5.1之前，没有这个参数，所有二进制文件的格式都是基于SQL语句（statement）级别的，因此基于这个格式的二进制日志文件的复制（Replication）和Oracle的逻辑Standby有点类似。

该值可以设置为STATEMENT、ROW和MIXED。

1、STATEMENT格式下，记录的是逻辑SQL语句。

2、ROW格式下，记录表的行更改情况。

3、MIXED格式下，MySQL默认采用STATEMENT格式进行二进制文件记录，但是在一些情况下会使用ROW格式。

## binlog\_annotate\_row\_events

binlog\_annotate\_row\_events：对于binlog\_format为row模式下是否添加注释，如果为on，每条dml语句都会有对应的注释显式原始语句。

## binlog\_cehcksum

binlog\_cehcksum：控制server写入每个事物到binlog是否进行校验，选择CRC32则进行CRC32校验。

# 原理

## binlog与redo一致性

### 必要性

我们首先要明白为什么需要保持binog与redo log之前数据的一致性，这里分两个方面来解释：

1. 保证binlog里面存在的事务一定在redo log里面存在，也就是binlog里不会比redo log多事务（可以少，因为redo log里面记录的事务可能有部分没有commit，这些事务最终可能会被rollback）。先来看这样一个场景（后面的场景都是假设binlog开启）：在一个AB复制环境下主库crash，然后crash recovery，此时如果binlog里面的事务信息与redo log里面的信息不一致，那么就会出现主库利用redo log进行恢复后，然后binlog部分的内容复制到从库去，然后出现主从数据不一致状态，所以需要保证binlog和redo log两者事务一致性。
2. 保证binlog里面事务顺序与redo log事务顺序一致性。这也是很重要的一点，假设两者记录的事务顺序不一致，那么会出现类似于主库事务执行的顺序为ta->tb->tc->td，但是binlog里面记录的是ta->tc->tb->td，binlog复制到从库后导致主从的数据不一致。

**必要性：**

保证binlog存在的事务一定在redo log里面存在。**主从复制架构中，主机崩溃恢复依赖redo log和binlog，从机数据来源是主机binlog**。

保证binlog里面事务顺序与redo log事务顺序一致。

### 解决方案

引入XA协议：

XA是由X/Open组织提出的分布式事务的规范（X代表transaction；A代表accordant）。XA规范主要定义了（全局）事务管理器（TM：Transaction Manager）和（局部）资源管理器（RM：Resource Manager）之间的接口。XA为了实现分布式事务，将事务的提交分成了两个阶段：也就是2PC（two phase commit），XA协议就是通过将事务的提交分为两个阶段来实现分布式事务。

prepare阶段：

持锁prepare\_commit\_mutex

write/sync redo log

undo设置为prepared状态

commit阶段：

write/sync binlog

innodb commit，写入commit标记，释放prepare\_commit\_mutex锁

说明：

1. 以binlog写入与否作为事务提交成功与否的标志
2. 由于prepare\_commit\_mutex锁存在，保证binlog和redo log之间顺序一致，但是却导致每个事物都需要一个fsync操作，导致性能急剧下降。

由于采用prepare\_commit\_mutex存在性能问题，MySQL5.6开始采用Binary Log Group Commit（BLGC）方案。

## XA与binlog

MySQL中的XA实现分为：外部XA和内部XA。

### 外部XA

前者是指我们通常意义上的分布式事务实现使用MySQL中的XA实现分布式事务时必须使用serializable隔离级别。

### 内部XA

内部XA将事务的提交分为两个阶段，而这种实现，解决了binlog和redo log的一致性问题。

第一阶段：InnoDB prepare，持有prepare\_commit\_mutex，并且write/sync reod log；将回滚段（undo段）设置为prepared状态，binlog不作任何操作。

第二阶段：包含两步：

1. write/sync binlog
2. InnoDB commit

以binlog的写入与否作为事务提交成功与否的标志，innodb commit标志并不是事务成功与否的标志。

详细过程如下：

#### preapre阶段

第一阶段，事务管理器向所有涉及到的数据库服务器发出prepare“准备提交”请求，数据库收到请求后执行数据修改和日志记录等处理，处理完成只是把事务的状态改为“可以提交”，然后把结果返回给事务管理器。

#### commit阶段

事务管理器收到回应后进入第二阶段，如果在第一阶段内有任何一个数据库的操作发生了错误，或者事务管理器收不到某个数据库的回应，则认为事务失败，回撤所有数据库的事务。数据库服务器收不到第二阶段的确认请求消息，也会把“可以提交”的事务回撤。如果第一阶段中所有数据库都提交成功，那么事务管理向数据库服务器发出“确认提交”请求，数据库服务器把事务的“可以提交”状态改为“提交完成”状态，然后返回应答。

## group commit

通过内部XA事务来保证binlog里面记录的事务不会比redo log多（也可以间接的理解为binlog一定只记录提交事务），这么做的原因是为了crash recovery后主从保持一致。

并发情况下是怎么来保证binlog与redo log之间顺序一致的？

MySQL5.6之前版本的解决方案是通过prepare\_commit\_mutex锁保证MySQL数据库上层binlog和InnoDB存储引擎层的事务提交顺序一致。

在每次进行XA事务时，在prepare阶段事务先拿到一个全局的prepare\_commit\_mutex，然后执行持久化（fsync）redo log与binlog，然后等fsync完了之后再释放prepare\_commit\_mutex，这样相当于串行化的效果。虽然保证了binlog与redo log之间的顺序一致性，但是却导致每个事物都需要一个fsync操作，而大家都知道在一次持久化的过程中代价最大的操作就是fsync了，而像write()这些不落地的操作代价相对而言就很小。并且还是持有全局大锁（prepare\_commit\_mutex：prepare和commit共用一把锁），这会导致性能急剧下降。

这也是MySQL5.6之前group commit失效的问题。

为了保证binlog和redo的提交顺序一致，并且能够进行group commit。MySQL5.6开始采用binary log group commit（BLGC）方案。

**说明：**

当引入group commit之后，sync\_binlog的含义就变了假定设为1000，表示的不再是1000个事务后做一次fsync，而是1000个事务组。

### MySQL5.6

BLGC实现方案如下：

在MySQL数据库上层进行提交时，首先按照顺序将其放入一个队列中，队列中的第一个事务成为leader，其他事务成为follower，leader控制着follower的行为。

BLGC的步骤分为以下三个阶段：

Flush阶段：将各个线程的binlog从cache写到文件中

Sync阶段：将内存中的binlog刷新到磁盘，若队列中有多个事务，那么仅一次fsync操作就完成了二进制日志的写入，这就是BLGC

Commit阶段：leader根据顺序调用存储引擎事务的提交（这里不用写redo log，在prepare阶段已写），InnoDB存储引擎本就支持group commit，因此修复了原先由于锁prepare\_commit\_mutex导致group commit失效的问题。

每个阶段同时只有一个线程在操作（分为三个阶段，每个阶段的任务分配给一个专门的线程，这就是经典的并发优化）。

这种实现的优势在于三个节点可以并发执行，从而提高效率。主义prepare阶段没有变，还是write/sync redo log。

当有一组事务在进行commit阶段时，其他事务可以进行flush阶段，从而使group commit不断生效。Group commit的效果由队列中事务的数量决定，若每次队列中仅有一个事务，那么效果和之前的差不多，甚至更差。当提交的事务越多时，group commit效果越明显，数据库性能的提升也就越大。

此时的崩溃恢复流程也是：

1. 扫描最后一个binlog文件，提取其中的xid
2. InnoDB位置了状态未prepare的事务链表，将这些事务的xid和binlog记录的xid做比较，如果在binlog中存在，则提交，否则回滚事务

通过这种方式，可以让InnoDB和binlog中的事务状态保持一致，显然只有事务在InnoDB层完成了prepare，并且写入了binlog，就可以从崩溃中恢复事务。

上面的优化解决了binlog的group commit问题，但是无法提升redo log的group commit写入以及减少log\_sys->mutex的竞争。（一个事务可能有多个redo log条目，redo log条目的写入顺序并不要求按照事务的提交顺序进行写入；由于reod log的条目最终会写入到redo log buffer，所以需要一个轻量级的锁log\_sys->mutex来控制不同事务向redo log buffer的写入）

为了解决上述问题，MySQL5.7引入了新的优化。

### MySQL5.7

从XA恢复逻辑可知，只要保证InnoDB prepare的redo日志在写binlog前完成write/sync即可。

因此我们对group commit的第一个stage的逻辑做了些许修改，大概描述如下：

1. InnoDB prepare，记录当前的LSN到thd中
2. 进入group commit的flush stage；leader搜索队列，同时算出队列中的最大的LSN
3. 将InnoDB的redo log write/fsync到指定的LSN（注：这一步就是reod log组写入，因为小于等于LSN的redo log被一次性写入到in\_logfile[0|1]）
4. 写binlog并进行随后的工作（sync binlog，innodb commit等）

也就是将redo log的write/sync延迟到了binlog group commit的flush阶段之后，sync binlog之前。

通过延迟写redo log的方式，显式的redo log做了一次组写入（redo log group write），并减少了（redo log）log\_sys->mutex的竞争。

也就是将binlog group commit对应的redo log也进行了group write，这里binlog和redo log都进行了优化。

## 事务崩溃恢复

事务崩溃恢复过程如下：

1. 崩溃恢复时，扫描最后一个binlog文件，提取其中的xid；
2. Innodb维持了状态为prepare的事务链表，将这些事务的xid和binlog中记录的xid做比较，如果在binlog中存在，则提交，否则回滚事务

通过这种方式，可以让InnoDB和binlog中的事务状态保持一致。

在prepare阶段崩溃，事务未写入binlog且存储引擎未提交，则会回滚。

在write/sync binlog阶段崩溃，也会回滚。

在写入InnoDB commit标志时崩溃，则没有关系，因为已经记录了此次事务的binlog，恢复时，会重新对commit标志进行写入。

**说明：MySQL为什么只需要扫描最后一个binlog文件呢？**

原因是每次在rotate到新的binlog文件时，总是保证没有正在提交的事务，然后fsync一次InnoDB的redo log。这样就可以保证老的binlog文件中的事务在InnoDB总是提交的。

## 总结

sync\_binlog=1且innodb\_flush\_log\_at\_trx\_commit=1时MySQL5.7的提交动作：

第一阶段：InnoDB prepare，将回滚段（undo段）设置为prepared状态，binlog不做任何操作，记录当前的LSN到thd中；

第二阶段：

Flush阶段：leader搜索队列，同时算出队列中最大的LSN，将各个线程的binlog从cache写到文件中；

将InnoDB的redo log write/fsync到指定的LSN；

Sync阶段：将内存中的binlog刷新到磁盘，若队列中有多个事务，那么仅一次fsync操作就完成了二进制日志的写入；

Commit阶段：leader根据顺序调用存储引擎层事务的提交，并将undo log从prepare状态设置为提交状态（可清理状态），提交事务锁等一系列操作。

说明：如果你关闭了binlog\_order\_commits选项，那么事务就各自进行提交，这种情况是不能保证innodb commit顺序和binlog写入顺序一致的，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。但可能会影响到物理备份的数据一致性，例如使用xtrabackup（而不是基于其上的innobackup脚本）依赖于事务页上记录的binlog位点，如果位点发生乱序，就会导致备份的数据不一致。

# 操作命令

## 开启日志

要启用二进制日志，必须设置 log\_bin 和 server\_id 并重新启动服务器。可以在log\_bin 内提及 path和 base 名称。

例如，log\_bin设置为/data/mysql/binlogs/server1，二进制日志存储在/data/mysql/binlogs文件夹中名为server1.000001、server1.000002等的日志文件中。每当服务器启动或刷新日志时，或者当前日志的大小达到max\_binlog\_size时，服务器都会在系列中创建一个新文件。

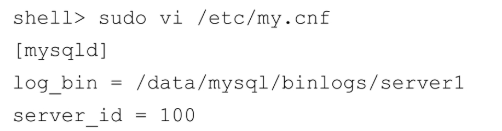
为了跟踪当前有哪些binlog文件，MySQL还会维护一个二进制日志索引文件，用来记录当前有哪些二进制日志文件有效，索引文件的文件默认为序号后缀改为.index后缀，方然用户可以通过log-bin-index配置项来修改。每个二进制日志的位置都在server1.index文件中被维护。

### 启用二进制日志

关于是否生成binlog可以通过sql\_log\_bin=0来关闭本链接的所有语句产生binlog，也可以通过binlog-db-db、binlog-ignore-db、replicate-db-db、replicate-ignode-db来控制。

1. 启用二进制日志并设置server\_id。在自己常用的编辑器中打开MySQL配置文件并添加以下代码。选择server\_id，使其在基础架构中对每个MySQL服务器都是唯一的。

也可以简单地把log\_bin变量放在my.cnf中，不赋予任何值。在这种情况下，二进制日志是在数据目录中创建的。可以使用主机名作为目录名称。



**扩展：**

开启binlog日志：

1）vi编辑打开mysql配置文件

# vi /usr/local/mysql/etc/my.cnf

在[mysqld] 区块

设置/添加 log-bin=mysql-bin 确认是打开状态(值 mysql-bin 是日志的基本名或前缀名)；

重启mysqld服务使配置生效

# pkill mysqld

# /usr/local/mysql/bin/mysqld\_safe --user=mysql &

2）也可登录mysql服务器，通过mysql的变量配置表，查看二进制日志是否已开启 单词：variable[ˈvɛriəbəl] 变量

登录服务器

# /usr/local/mysql/bin/mysql -uroot -p123456

mysql> show variables like 'log\_%';

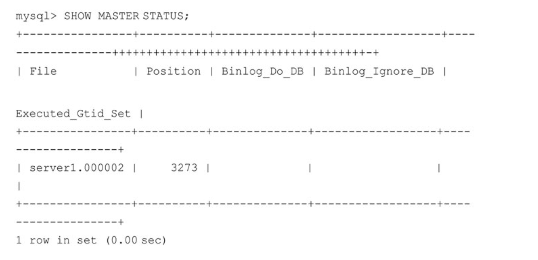
1. 重新启动MySQL服务器：



1. 验证是否创建了二进制日志：



1. 执行SHOW BINARY LOGS；或SHOW MASTER LOGS；，以显示服务器的所有二进制日志。
2. 执行命令SHOW MASTER STATUS；以获取当前的二进制日志位置：



一旦server1.000001达到max\_binlog\_size（默认为1 GB），一个新文件server1.000002就会被创建，并被添加到server1.index中。可以使用 SET@@global.max\_binlog\_size=536870912动态设置max\_binlog\_size。

### 禁用会话的二进制日志

有些情况下我们不希望将执行语句复制到其他服务器上。为此，可以使用以下命令来禁用该会话的二进制日志：



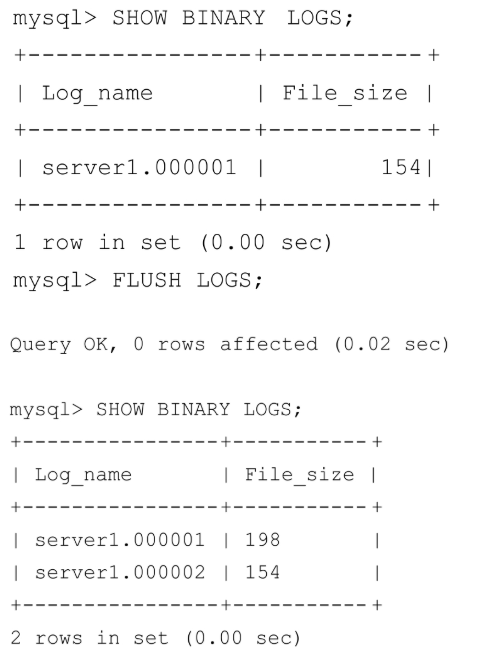
在这条语句后的所有SQL语句都不会被记录到二进制日志中，不过这仅仅是针对该会话的。

要重新启用二进制日志，可以执行以下操作：



### 移至下一个日志

可以使用FLUSH LOGS命令关闭当前的二进制日志并打开一个新的二进制日志：



### 清理二进制日志

随着写入次数的增多，二进制日志会消耗大量空间。如果放任不管，这些写入操作将很快占满磁盘空间，因此清理它们至关重要。

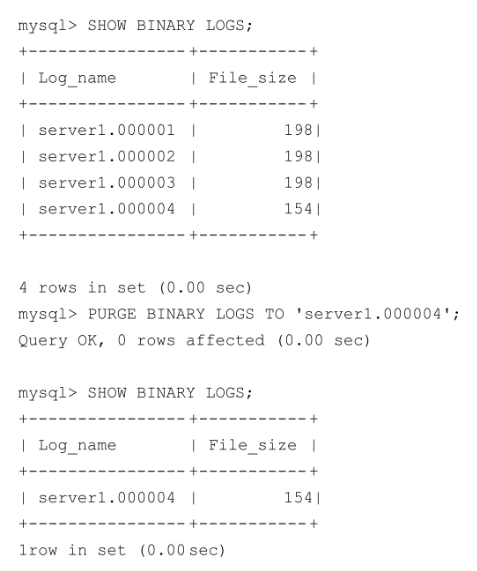
1. 使用binlog\_expire\_logs\_seconds 和expire\_logs\_days 设置日志的到期时间。

如果想以天为单位设置到期时间，请设置 expire\_logs\_days。例如，如果要删除两天之前的所有二进制日志，请SET@@global.expire\_logs\_days=2。如果将该值设置为0，则禁用设置会自动到期。

如果想以更细的粒度来设置到期时间，可以使用 binlog\_expire\_logs\_seconds变量，它能够以秒为单位来设置二进制日志过期时间。

这个变量的效果和 expire\_logs\_days 的效果是叠加的。例如，如果expire\_logs\_days是1并且binlog\_expire\_logs\_seconds是43200，那么二进制日志就会每 1.5 天清除一次。这与将binlog\_expire\_logs\_seconds设置为129600、将expire\_logs\_days设置为0的效果是相同的。在MySQL 8.0 中，binlog\_expire\_logs\_seconds 和expire\_logs\_days必须设置为0，以禁止自动清除二进制日志。

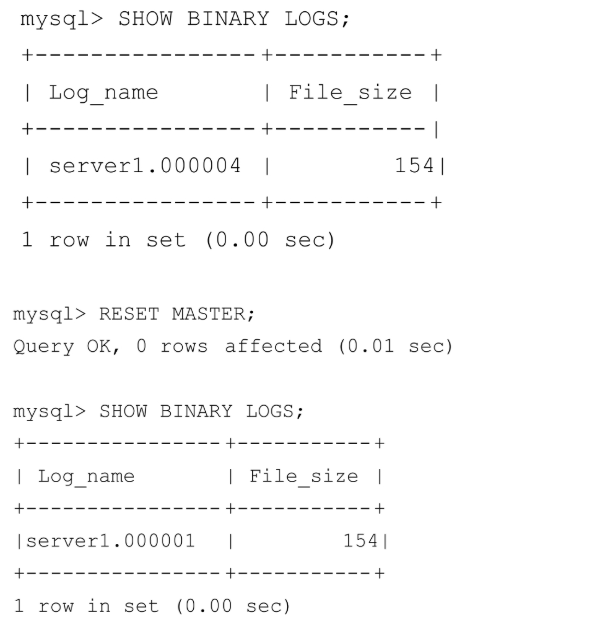
1. 要手动清除日志，请执行PURGE BINARY LOGS TO＇＜file\_name＞＇。例如，有server1.000001、server1.000002、server1.000003和server1.000004文件，如果执行 PURGEBINARY LOGS TO＇server1.000004＇，则从server1.000001 到 server1.000003 的所有文件都会被删除，但文件server1.000004不会被删除：



除了指定某个日志文件，还可以执行命令PURGE BINARY LOGS BEFORE＇2017-08-03 15：45：00＇。除了使用BINARY，还可以使用MASTER。

mysql＞ PURGE MASTER LOGS TO ＇server1.000004＇可以实现和之前语句一样的效果。

1. 要删除所有二进制日志并再次从头开始，请执行RESET MASTER：



## 查看binlog\_format

show variables like ‘binlog\_format’;

## 设置binlog过期时间

SET global expire\_logs\_days=10;

## 查看日志列表

### 查看主节点日志

mysql> show master logs;

### 查看备节点日志

mysql> show slave logs;

## 查看节点状态

即最后(最新)一个binlog日志的编号名称，及其最后一个操作事件pos结束点(Position)值

### 主节点状态

查看主节点状态信息：

mysql> show master status;

### 从节点状态

查看从节点状态信息：

mysql> show slave status;

## 刷新log日志

自此刻开始产生一个新编号的binlog日志文件

mysql> flush logs;

注：每当mysqld服务重启时，会自动执行此命令，刷新binlog日志；在mysqldump备份数据时加 -F 选项也会刷新binlog日志；

## 删除(清空)所有binlog日志

### 重置binlog

mysql> reset master; //删除master的binlog

mysql> reset master; //删除slave的中继日志

### 自动删除binlog

expire\_logs\_days：binlog日志保存的天数，如果设置为0，则表示一直都不删除binlog日志，设置成7，表示自动删除7天前的日志。

### 手动删除binlog(purge)

PURGE {MASTER | BINARY} LOGS TO ‘log\_name’;

PURGE {MASTER | BINARY} LOGS BEFORE ‘date’;

举例：

PURGE MASTER LOGS TO ‘mysql-bin.00010’;

PURGE MASTER LOGS BEFORE ‘2020-02-02 10:00:00’;

注意：最好在slave上面去看下当前同步到那个binlog文件了，用show slave status查看。否则，master上删多了的话，就造成slave缺失文件而导致数据不一致了。

## 启动复制

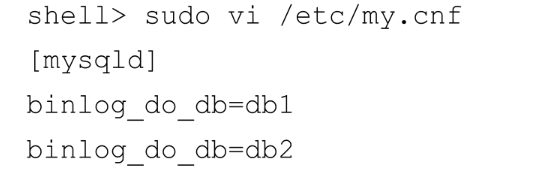
Start slave可以指定线程类型：IO\_THREAD，SQL\_THREAD，如果不指定，两个都启动。

注：具体工作流程参考《MySQL主从复制》。

## 忽略要写入二进制日志的数据库

可以通过在my.cnf中指定--binlog-do-db=db\_name选项，来选择将哪些数据库写入二进制日志。要指定多个数据库，就必须使用此选项的多个实例。由于数据库的名字可以包含逗号，因此如果提供逗号分隔列表，则该列表将被视为单个数据库的名字。需要重新启动MySQL服务器才能使更改生效。

打开my.cnf并添加以下行



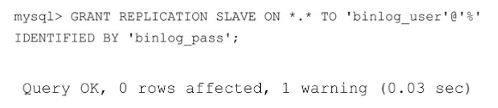
binlog-do-db 上的行为从基于语句的日志记录更改为基于行的日志记录，就像mysqlbinlog实用程序中的--database选项一样。在基于语句的日志记录中，只有那些默认数据库（即用USE选择的）的语句才会被写入二进制日志。使用 binlog-do-db 选项时应该非常小心，因为它的工作方式与你在使用基于语句的日志记录时的方式不同。

# 二进制日志备份

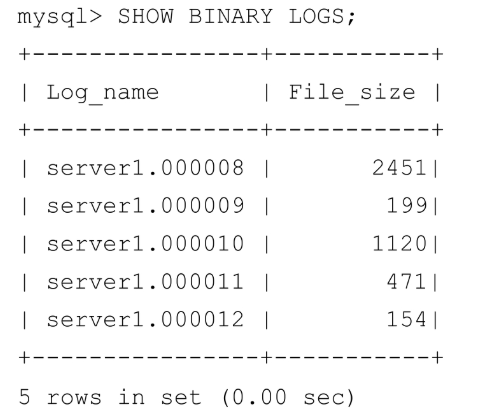
备份二进制日志：该进程将二进制日志从数据库服务器流式传输到远程备份服务器。既可以从从服务器也可以从主服务器进行二进制日志备份。如果你正在从主服务器进行二进制日志备份，并在从服务器进行实际备份，则应使用--dump-slave获取相应的主日志位置。如果你使用的是mydumper或XtraBackup，则主和从二进制日志位置会被同时提供。

**操作步骤：**

1. 在服务器上创建一个复制用户，并设置一个强密码：

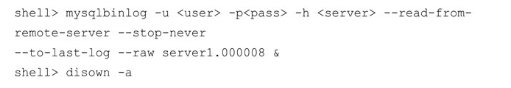


1. 检查服务器上的二进制日志：

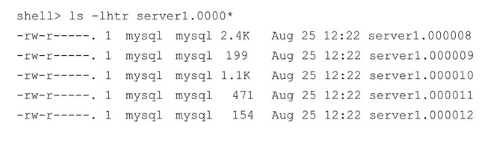


你可以在服务器上找到第一个可用的二进制日志，可以从这里开始备份。在这个例子中，它是server1.000008。

1. 登录到备份服务器并执行以下命令，会将二进制日志从MySQL服务器复制到备份服务器。你可以使用nohup或disown：



1. 验证是否正在备份二进制日志：



# 解析工具

## mysqlbinlog

可以使用mysqlbinlog实用程序（MySQL已包含）从二进制日志中提取内容，并将其应用到其他服务器。

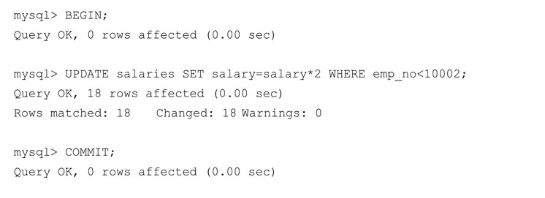
1、准备工作

使用各种二进制格式执行几条语句。如果把binlog\_format设置为GLOBAL级别（全局范围），必须断开并重新连接，以使更改生效。如果想保持连接，请把binlog\_format设置为SESSION级别（会话范围）。

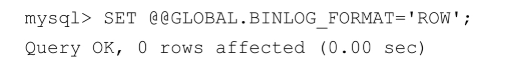
更改为基于语句的复制（SBR）：



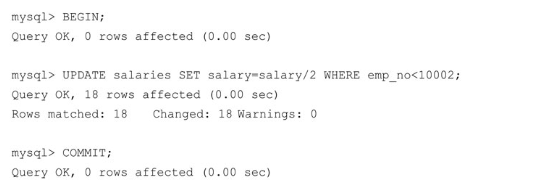
更新几行：



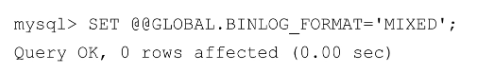
更改为基于行的复制（RBR）：



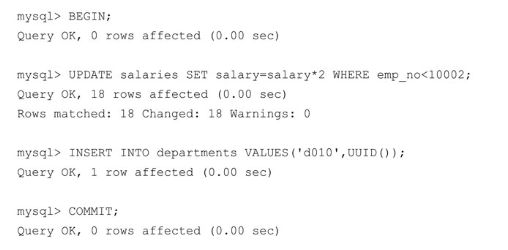
更新几行：



改为MIXED格式：



更新几行：

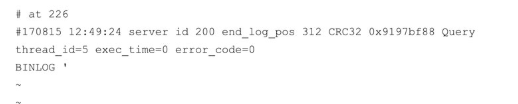


1. 操作步骤

要显示日志server1.000001的内容，请执行以下操作：



你会得到类似下面这样的输出结果：



在第一行中，＃at后面的数字表示二进制日志文件中事件的起始位置（文件偏移量）。第二行包含了语句在服务器上被启用的时间戳。时间戳后面跟着 server ID、end\_log\_pos、thread\_id、exec\_time和error\_code。

● server id：产生该事件的服务器的server\_id值（在这个例子中为200）。

● end\_log\_pos：下一个事件的开始位置。

● thread\_id：指示哪个线程执行了该事件。

● exec\_time：在主服务器上，它代表执行事件的时间；在从服务器上，它代表从服务器的最终执行时间与主服务器的开始执行时间之间的差值，这个差值可以作为备份相对于主服务器滞后多少的指标。

● error\_code：代表执行事件的结果。零意味着没有错误发生。

**回顾：**

1. 我们在基于语句的复制中执行了UPDATE语句，而且在二进制日志中记录了相同的语句。除了保存在服务器上，会话变量也被保存在二进制日志中，以在从库上复制相同的行为：



1. 当使用基于行的复制（RBR）时，会以二进制格式对整行（而不是语句）进行保存，而且二进制格式不能读取。此外，你可以观察长度，单个更新语句会生成很多数据。



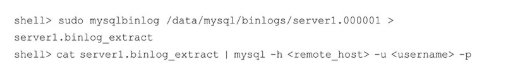
1. 当使用MIXED格式时，UPDATE语句被记录为SQL语句，而INSERT语句以基于行的格式被记录，因为INSERT有非确定性的UUID()函数：



提取的日志可以被传送给MySQL以回放事件。在重放二进制日志时最好使用force选项，这样即使force选项卡在某个点上，执行也不会停止。稍后，你可以查找错误并手动修复数据。



或者也可以先保存到文件中，稍后执行：



### 查看binlog日志内容

mysqlbinlog -v mysql-bin.000001 | more

-v -v comments on column data types

--hexdump augment output with hexadecimal and ASCII event dump

### 根据时间和位置进行抽取

我们可以通过指定位置从二进制日志中提取部分数据。假设你想做时间点恢复。假如在 2017-08-1912：18：00 执行了 DROP DATABASE 命令，并且最新的可用备份是在2017-08-19 12：00：00 做的，该备份已经恢复。现在，需要恢复从 12：00：01 至2017-08-19 12：17：00 的数据。请记住，如果提取完整的日志，它还将包含 DROP DATABASE命令，该命令将再次擦除数据。

可以通过--start-datetime和--stop-datatime选项来指定提取数据的时间窗口。



使用时间窗口的缺点是，你会失去灾难发生那一刻的事务。为避免这种情况，必须在二进制日志中使用事件的文件偏移量。

一个连续的备份会保存它已完成备份的所有binlog文件的偏移量。备份恢复后，必须从备份的偏移量中提取binlog。

假设备份的偏移量为471，执行DROP DATABASE命令的偏移量为1793。可以使用--start-position和--stop-position选项来提取偏移量之间的日志：



请确保DROP DATABASE命令在提取的binlog中不再出现。

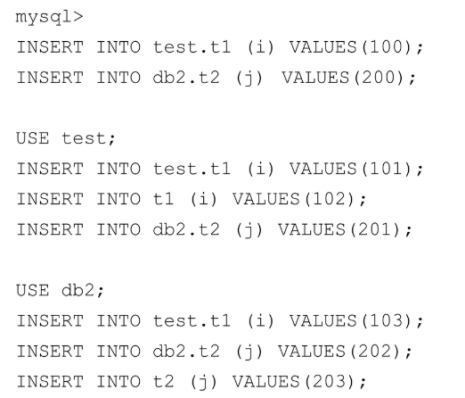
### 基于数据库进行提取

使用--database选项可以过滤特定数据库的事件。如果多次提交，则只有最后一个选项会被考虑。这对于基于行的复制非常有效。但对于基于语句的复制和MIXED，只有当选择默认数据库时才会提供输出。

以下命令从employees 数据库中提取事件：



正如MySQL 8文档中所解释的，假设二进制日志是通过使用基于语句的日志记录执行这些语句而创建的：



mysqlbinlog--database=test 不输出前两个 INSERT 语句，因为没有默认数据库。

mysqlbinlog--database=test 输出USE test后面的三条INSERT语句，但不是USE db2后面的三条INSERT语句。

因为没有默认数据库，mysqlbinlog--database=db2 不输出前两条INSERT语句。

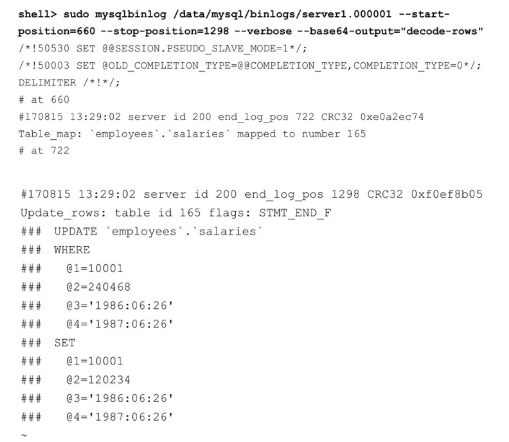
mysqlbinlog--database=db2不会输出USE test后的三条INSERT语句，但会输出在USE db2之后的三条INSERT语句。

### 提取行事件显示

默认情况下，基于行的复制日志显示为二进制格式。要查看行信息，必须将--verbose或-v选项传递给mysqlbinlog。行事件的二进制格式以注释的伪SQL语句的形式显示，其中的行以＃＃＃开始。可以看到，单个更新语句被改写为了每行的UPDATE语句。



如果你只想查看没有二进制行信息的伪 SQL 语句，请指定--base64-output=〝decode-rows〝以及--verbose：



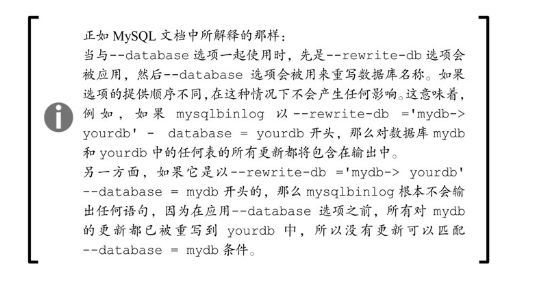
### 重写数据库名称

假设你想将生产服务器上的employees 数据库的二进制日志恢复为开发服务器上的employees\_dev，可以使用--rewrite-db=＇from\_name-＞ to\_name＇选项。这会将所有from\_name重写为to\_name。

要转换多个数据库，请多次指定该选项：



可以看到上面使用了 use`employees\_dev`/\*！\*/；语句。因此，在恢复时，所有更改将应用于employees\_dev数据库。



### 在恢复时禁用二进制日志

在恢复二进制日志的过程中，如果你不希望mysqlbinlog进程创建二进制日志，则可以使用--disable-log-bin选项：

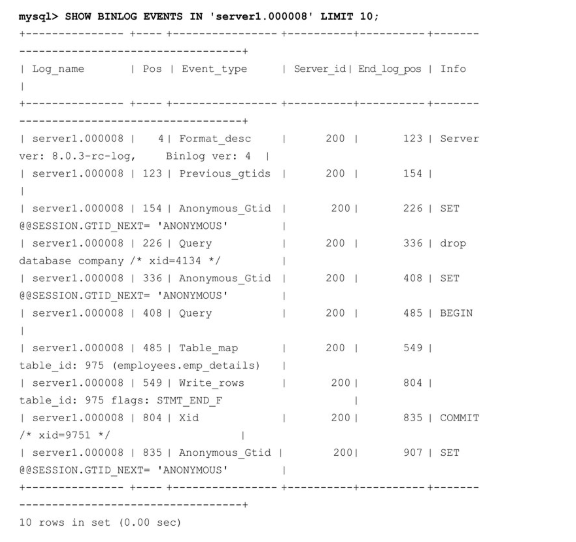


可以看到SQL\_LOG\_BIN=0被写入binlog恢复文件中，这将防止创建binlog。

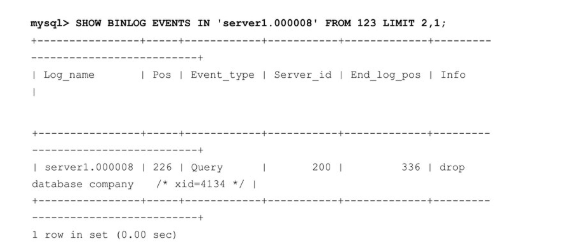


### 显示二进制日志文件中的事件

除了使用mysqlbinlog，还可以使用SHOW BINLOG EVENTS命令来显示事件。以下命令将显示server1.000008二进制日志中的事件。如果未指定LIMIT，则显示所有事件：



也可以指定位置和偏移量：



### 数据恢复

如果我们要跳过某个时间段或者位置段，需要指定起止时间或者位置信息。

#### 通过事件的时间恢复

我们可以通过参数--start-datetime和--stop-datetime指定恢复binlog日志的起止时间点，时间使用DATETIME格式。

比如在时间点2005-04-20 10:00:00我们删除一个库，我们要恢复该时间点前的所有日志：

mysqlbinlog --stop-datetime="2005-04-20 9:59:59 " /usr/local/mysql/data/binlog.123456 | mysql -u root -p

我们可能几个小时后才发现该错误，后面又有一系列的增删改等操作，我们还需要恢复后续的binlog，我们可以指定起始时间：

mysqlbinlog --start-datetime="2005-04-20 10:01:00" --stop-datetime="2005-04-20 9:59:59 " /usr/local/mysql/data/binlog.123456 | mysql -u root -p

通过这种方法恢复，我们需要通过查看binlog日志知道发生误操作的准确时间点，查看日志我们可以先将日志输出到文本里：

mysqlbinlog /usr/local/mysql/data/binlog.123456 > /tmp/mysql\_restore.sql

#### 通过事件的位置恢复

我们可以通过参数--start-position和--stop-position指定恢复binlog日志的起止位置点，通过位置的恢复需要我们有更加精细的操作，例如在某个时间点我们执行了错误的语句，且这个时间点前后都有大并发操作，要确定破坏性SQL的时间点，我们可以先导出大致的时间段的日志到文件以缩小查找范围，再去分析和确定。

mysqlbinlog --start-datetime="2005-04-20 10:01:00" --stop-datetime="2005-04-20 9:59:59 " /usr/local/mysql/data/binlog.123456 > /tmp/mysql\_restore.sql

确定好需要跳过的位置之后，我们就可以进行恢复了：

mysqlbinlog --stop-position=368312 binlog.123456 | mysql -u root -p

mysqlbinlog --start-position=368315 binlog.123456 | mysql -u root -p

注：mysqlbinlog工具的输出会在每条SQL语句前增加SET TIMESTAMP语句，恢复的数据及MySQL日志反映当前时间。

**如果在同一个时间点，binlog日志中有执行过多条SQL语句的话，那么我们在恢复数据库时，一定要根据pos点的位置来恢复数据切记切记！因为此时在binlog日志中，时间点是一样的，但是pos位置节点是唯一的**。

## show binlog events命令

show binlog events in ‘mysql-bin.000001’;

## 二进制查看/hexdump

hexdump -Cv mysql-bin.000001

# 迁移工具

## mysqlbinlogmove

由于二进制日志占用越来越多的空间，有时你可能希望更改二进制日志的位置，可以按照以下步骤操作。单独更改log\_bin是不够的，必须迁移所有二进制日志并在索引文件中更新位置。mysqlbinlogmove工具可以自动执行这些任务，简化你的工作。

操作步骤：

你需要先安装MySQL工具集，以使用mysqlbinlogmove脚本。

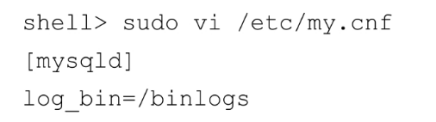
1. 停止MySQL服务器的运行：



1. 启动mysqlbinlogmove工具。如果要将二进制日志从/data/mysql/binlogs更改为/binlogs，则应使用以下命令。如果base name不是默认名称，则必须通过--bin-log-basename 选项设定basename：



1. 编辑my.cnf文件并更新log\_bin的新位置：



1. 启动MySQL服务器：



如果有大量的二进制日志，服务器的停机时间会很长。为了避免这种情况，可以使用--server选项来重新定位所有二进制日志——但是当前正在使用的二进制日志（具有较高序列号）除外。然后停止服务器的运行，使用上述方法，并重新定位最后一个二进制日志，这会快很多，因为只有一个文件存在。然后你可以更改my.cnf并启动服务器。例如：



# 应用

## 主从复制

MySQL Replication在Master端开启binlog，Master把它的二进制日志传递给slaves来达到master-slave数据一致的目的。

通过复制和执行二进制日志使得远程的MySQL数据库（一般称为slave或standby）与一台MySQL数据库（一般称为master或primary）进行实时同步。

## 数据恢复

数据恢复，某些数据的恢复需要二进制日志，例如，如果需要恢复数据库全量备份的文件，可以通过二进制日志进行point-in-time恢复。

通过使用mysqlbinlog工具来使恢复数据。

举例：

mysqlbinlog -vv mysql-bin.000001 | mysql -u -p

## 审计（audit）

审计（audit）：用户通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入的攻击。