# 主从复制

## 背景

为什么要做主从复制？

1、在业务复杂的系统中，有这么一个情景，有一句sql语句需要锁表，导致暂时不能使用读的服务，那么就很影响运行中的业务，使用主从复制，让主库负责写，从库负责读，这样，即使主库出现了锁表的情景，通过读从库也可以保证业务的正常运作。

2、从库可以做数据的热备

3、架构的扩展。业务量越来越大，I/O访问频率过高，单机无法满足，此时做多库的存储，降低磁盘I/O访问的频率，提高单个机器的I/O性能。

## 概述

复制是指将主数据库的DDL和DML操作通过**二进制日志**传到复制服务器（也叫从库）上，然后在从库上对这些日志**重新执行**（也叫**重做**），从而使得从库和主库的数据保持同步。

MySQL支持一台主库同时向若干台从库进行复制，从库同时也可以作为其他服务器的主库，实现链式的复制。

MySQL复制的优点：

1. 如果主库出现问题，可以快速切换到从库提供服务；
2. 可以在从库上执行查询操作，降低主库的访问压力；
3. 可以在从库上执行备份，以避免备份期间影响主库的服务。

**注意：**

由于MySQL实现的是**异步的复制**，所以主从库之间存在一定的差距，在从库上进行的查询操作需要考虑到这些数据的差异，**一般只有更新不频繁的数据或者对实时性要求不高的数据可以通过从库查询，实时性要求高的数据仍然需要从主数据库获得**。

## 主从复制与一致性

### 单机数据一致性

MySQL作为一个可插拔的数据库系统，支持插件式的存储引擎，在设计上分为Server层和Storage Engine层。

在Server层，MySQL以events的形式记录数据库各种操作的Binlog二进制日志，其基本核心作用有：复制和备份。

除此之外，我们结合多样化的业务场景需求，基于Binlog的特性构建了强大的MySQL生态，如：DTS、单元化、异构系统之间实时同步等等，Binlog早已成为MySQL生态中不可缺少的模块。

而在Storage Engine层，InnoDB作为比较通用的存储引擎，其在高可用和高性能两方面作了较好的平衡，早已经成为使用MySQL的首选。

和大多数关系型数据库一样，InnoDB采用WAL技术，即InnoDB Redo Log记录了对数据文件的物理更改，并保证总是日志先行，在持久化数据文件前，保证之前的redo日志已经写到磁盘。

Binlog和InnoDB Redo Log是否落盘将直接影响实例在异常宕机后数据能恢复到什么程度。InnoDB提供了相应的参数来控制事务提交时，写日志的方式和策略，例如：

innodb\_flush\_method:控制innodb数据文件、日志文件的打开和刷写的方式，建议取值：fsync、O\_DIRECT。

innodb\_flush\_log\_at\_trx\_commit:控制每次事务提交时，重做日志的写盘和落盘策略，可取值：0，1，2。

当innodb\_flush\_log\_at\_trx\_commit=1时，每次事务提交，日志写到InnoDB Log Buffer后，会等待Log Buffer中的日志写到Innodb日志文件并刷新到磁盘上才返回成功。

sync\_binlog：控制每次事务提交时，Binlog日志多久刷新到磁盘上，可取值：0或者n(N为正整数)。

不同取值会影响MySQL的性能和异常crash后数据能恢复的程度。当sync\_binlog=1时，MySQL每次事务提交都会将binlog\_cache中的数据强制写入磁盘。

innodb\_doublewrite：控制是否打开double writer功能，取值ON或者OFF。

当Innodb的page size默认16K，磁盘单次写的page大小通常为4K或者远小于Innodb的page大小时，发生了系统断电/os crash ，刚好只有一部分写是成功的，则会遇到partial page write问题，从而可能导致crash后由于部分写失败的page影响数据的恢复。InnoDB为此提供了Double Writer技术来避免partial page write的发生。

innodb\_support\_xa:控制是否开启InnoDB的两阶段事务提交.默认情况下，innodb\_support\_xa=true，支持xa两段式事务提交。

以上参数不同的取值分别影响着MySQL异常crash后数据能恢复的程度和写入性能，实际使用过程中，需要结合业务的特性和实际需求，来设置合理的配置。比如:

MySQL单实例，Binlog关闭场景：

innodb\_flush\_log\_at\_trx\_commit=1，innodb\_doublewrite=ON时，能够保证不论是MySQL Crash 还是OS Crash 或者是主机断电重启都不会丢失数据。

MySQL单实例，Binlog开启场景：

默认innodb\_support\_xa=ON，开启binlog后事务提交流程会变成两阶段提交，这里的两阶段提交并不涉及分布式事务，mysql把它称之为内部xa事务。

当innodb\_flush\_log\_at\_trx\_commit=1，sync\_binlog=1，innodb\_doublewrite=ON，,innodb\_support\_xa=ON时，同样能够保证不论是MySQL Crash 还是OS Crash 或者是主机断电重启都不会丢失数据。

但是，当由于主机硬件故障等原因导致主机完全无法启动时，则MySQL单实例面临着单点故障导致数据丢失的风险，故MySQL单实例通常不适用于生产环境。

### 集群数据一致性

MySQL集群通常指MySQL的主从复制架构。

通常使用MySQL主从复制来解决MySQL的单点故障问题，其通过逻辑复制的方式把主库的变更同步到从库，主备之间无法保证严格一致的模式，于是，MySQL的主从复制带来了主从“数据一致性”的问题。MySQL的复制分为：异步复制、半同步复制、全同步复制。

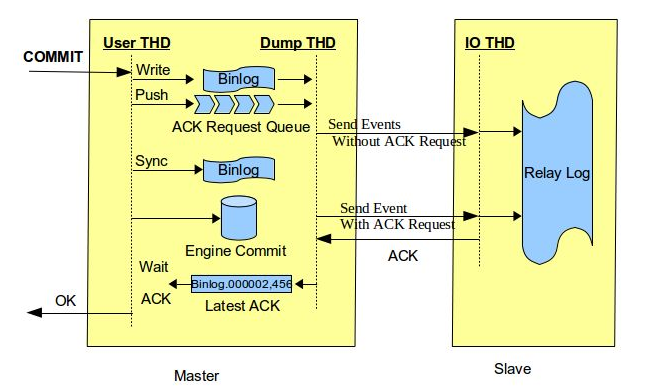
#### 异步复制

主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果crash掉了，此时主上已经提交的事务可能并没有传到从库上，如果此时，强行将从提升为主，可能导致“数据不一致”。早期MySQL仅仅支持异步复制。

#### 半同步复制

MySQL在5.5中引入了半同步复制，主库在应答客户端提交的事务前需要保证至少一个从库接收并写到relay log中，半同步复制通过rpl\_semi\_sync\_master\_wait\_point参数来控制master在哪个环节接收 slave ack，master 接收到 ack 后返回状态给客户端，此参数一共有两个选项 AFTER\_SYNC & AFTER\_COMMIT。

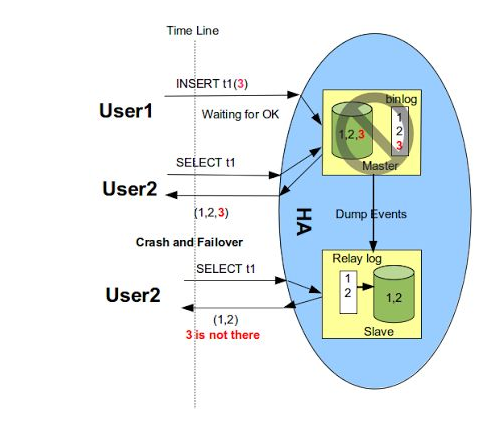
配置为WAIT\_AFTER\_COMMIT



rpl\_semi\_sync\_master\_wait\_point为WAIT\_AFTER\_COMMIT时，commitTrx的调用在engine层commit之后，如上图所示。

即在等待Slave ACK时候，虽然没有返回当前客户端，但事务已经提交，其他客户端会读取到已提交事务。如果Slave端还没有读到该事务的events，同时主库发生了crash，然后切换到备库。

那么之前读到的事务就不见了，出现了数据不一致的问题：

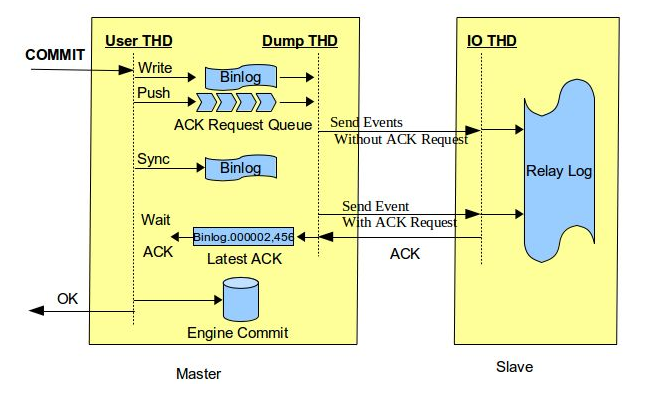


如果主库永远启动不了，那么实际上在主库已经成功提交的事务，在从库上是找不到的，也就是数据丢失了。

配置为WAIT\_AFTER\_SYNC

MySQL官方针对上述问题，在5.7.2引入了Loss-less Semi-Synchronous，在调用binlog sync之后，engine层commit之前等待Slave ACK。这样只有在确认Slave收到事务events后，事务才会提交。

如下图所示：



在after\_sync模式下解决了after\_commit模式带来的数据不一致的问题，因为主库没有提交事务。

但也会有个问题，当主库在binlog flush并且binlog同步到了备库之后，binlog sync之前发生了abort，那么很明显这个事务在主库上是未提交成功的（由于abort之前binlog未sync完成，主库恢复后事务会被回滚掉），但由于从库已经收到了这些Binlog，并且执行成功，相当于在从库上多出了数据，从而可能造成“数据不一致”。

此外，MySQL半同步复制架构中，主库在等待备库ack时候，如果超时会退化为异步后，也可能导致“数据不一致”。

## 原理

MySQL复制原理大致如下：

1. 首先，MySQL主库在事务提交时会把数据变更作为事件Events记录在二进制日志文件binlog中；MySQL主库上的sync\_binlog参数控制binlog日志刷新到磁盘；
2. **主库推送二进制日志文件binlog中的事件到从库的中继日志Relay Log**，之后从库根据中继日志Relay Log重做数据变更操作，通过逻辑复制以此来达到主库和从库的数据一致。

MySQL通过3个线程来完成主从库间的数据复制：其中Binlog Dump线程跑在主库上，I/O线程和SQL线程跑在从库上。当在从库上启动复制（START SLAVE）时，首先创建I/O线程连接主库，主库随后创建Binlog Dump线程读取数据库事件并发送给I/O线程，I/O线程获取到事件数据后更新到从库的中继日志Relay Log中去，之后从库上的SQL线程读取中继日志Relay Log中更新的数据库事件并应用。

**从库有两个线程IO线程和SQL线程**

1. 从库的IO线程向主库的主进程发送请求，主库验证从库，交给主库IO线程负责数据传输；
2. 主库IO线程对比从库发送过来的master.info里的信息，将binlog文件信息，偏移量和binlog文件名等发送给从库；
3. 从库接收到信息后，将binlog信息保存到relay-bin中，同时更新master.info的偏移量和binlog文件名；
4. 从库的SQL线程不断的读取relay-bin的信息，同时将读到的偏移量和文件名写到relay-log.info文件，binlog信息写进自己的数据库，一次同步操作完成；
5. 完成上次同步后，从库IO线程不断的向主库IO线程要binlog信息；
6. 从库如果也要做主库，也要打开log\_bin 和log-slave-update参数。

### 配置文件

配置读写mysql主从复制的步骤：

1. 在主库与从库都安装mysql数据库
2. 在主库的配置文件(/etc/my.cnf)中配置server-id 和log-bin
3. 在登陆主库后创建认证用户并做授权。
4. 在从库的配置文件(/etc/my.cnf)中配置server-id
5. 登陆从库后，指定master并开启同步开关。

需要注意的是server-id主从库的配置是不一样的。

**server-id存在作用：**

mysql同步的数据中是包含server-id的，而server-id用于标识该语句最初是从哪个server写入的。因此server-id一定要有的。

Server-id不能相同的原因：每一个同步中的slave在master上都对应一个master线程，该线程就是通过slave的server-id来标识的；每个slave在master端最多有一个master线程，如果两个slave的server-id 相同，则后一个连接成功时，slave主动连接master之后，如果slave上面执行了slave stop；则连接断开，但是master上对应的线程并没有退出；当slave start之后，master不能再创建一个线程而保留原来的线程，那样同步就可能有问题；

在mysql做主主同步时，多个主需要构成一个环状，但是同步的时候有要保证一条数据不会陷入死循环，这里就是靠server-id来实现的。

### 日志文件

#### 二进制文件Binlog

二进制文件（Binlog）会把MySQL中的所有数据修改操作以二进制的形式记录到日志文件中，包括Create、Drop、Insert、Update、Delete等操作，但二进制日志文件（binlog）不会记录select操作，因为select操作并不修改数据。

可以通过show variables查看binlog格式，binlog支持statement、row、mixed三种格式，也对应了MySQL的三种复制技术。

二进制日志文件binlog的格式有以下3种：

Statement：基于SQL语句级别的binlog，每条修改数据的SQL都会保存在binlog中，存储日志量是最小的。

Row：基于行级别，记录每一行数据的变化，也就是将每行数据的变化都记录到binlog中，记录得非常仔细，但并不记录原始SQL，存储event数据，存储日志量大，但是不能很直接的进行读取；在复制的时候，并不会因为存储过程或触发器造成主从库数据不一致的问题，但是记录的日志量较statement格式要大得多。

Mixed：混合statement和row模式，默认情况下采用statement模式记录，某些情况下会切换到row模式。如果每天数据操作量很大，产生的日志比较多，可以考虑选择使用mixed格式。

注：Row格式比Statement格式更能保证从库数据的一致性（复制的是记录，而不是单纯操作SQL）。当然，Row格式下的Binlog的日志量很可能会增大非常多，在设置时需要考虑磁盘空间问题。

同时对应MySQL复制的3种技术：

binlog\_format=Statement：基于SQL语句的复制，也叫Statement-Based Replication（SBR），MySQL5.1.4之前仅提供基于SQL语句的复制。

binlog\_format=Row：基于行的复制，也叫Row-Based Replication（RBR）。

binlog\_format=Mixed：混合复制模式，混合了基于SQL语句的复制和基于行的复制。

#### 中继日志文件Relay Log

中继日志文件Relay Log的文件格式、内容和二进制日志文件binlog一样，唯一的区别在于从库上的SQL线程在执行完当前中继日志文件Relay Log中的事件之后，SQL线程会自动删除当前中继日志Relay Log，避免从库上的中继日志文件占用过多的磁盘空间。

为了保证从库Crash重启之后，从库的I/O线程和SQL线程仍然能够知道从哪里开始复制，从库上默认还会创建两个日志文件master.info和relay-log.info用来保存复制的进度。这两个文件在磁盘上以文件形式分别记录了从库的I/O线程当前读取二进制日志binlog的进度和SQL线程应用中继日志Relay Log的进度。

### 过程

#### 步骤

整个主从复制的流程可以通过以下图示理解：





步骤一：主库db的更新事件(update、insert、delete)被写到binlog

步骤二：从库发起连接，连接到主库

步骤三：此时主库创建一个binlog dump thread，把binlog的内容发送到从库

步骤四：从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log

步骤五：还会创建一个SQL线程，从relay log里面读取内容，从`Exec\_Master\_Log\_Pos`位置开始执行读取到的更新事件，将更新内容写入到slave的db

注：上面的解释是解释每一步做了什么，整个mysql主从复制是异步的，不是按照上面的步骤执行的。





主从复制的基础是主库记录数据库的所有变更记录到binlog。binlog是数据库服务器启动的那一刻起，保存所有修改数据库结构或内容的一个文件。

mysql主从复制是一个**异步**的复制过程，主库发送更新事件到从库，从库读取更新记录，并执行更新记录，使得从库的内容与主库保持一致。

在主库里，只要有更新事件出现，就会被依次地写入到binlog里面，之后会推到从库中作为从库进行复制的数据源。

#### binlog输出线程

**binlog输出线程**。每当有从库连接到主库的时候，主库都会创建一个线程然后发送binlog内容到从库。

对于每一个即将发送给从库的sql事件，binlog输出线程会将其锁住。一旦该事件被线程读取完之后，该锁会被释放，即使在该事件完全发送到从库的时候，该锁也会被释放。

在从库里，当复制开始的时候，从库就会创建两个线程进行处理：

#### 从库I/O线程

**从库I/O线程**。当START SLAVE语句在从库开始执行之后，从库创建一个I/O线程，该线程连接到主库并请求主库发送binlog里面的更新记录到从库上。

从库I/O线程读取主库的binlog输出线程发送的更新并拷贝这些更新到本地文件，其中包括relay log文件。

#### 从库的SQL线程

**从库的SQL线程**。从库创建一个SQL线程，这个线程读取从库I/O线程写到relay log的更新事件并执行。

可以知道，对于每一个主从复制的连接，都有三个线程。拥有多个从库的主库为每一个连接到主库的从库创建一个binlog输出线程，每一个从库都有它自己的I/O线程和SQL线程。

从库通过创建两个独立的线程，使得在进行复制时，从库的读和写进行了分离。因此，即使负责执行的线程运行较慢，负责读取更新语句的线程并不会因此变得缓慢。比如说，如果从库有一段时间没运行了，当它在此启动的时候，尽管它的SQL线程执行比较慢，它的I/O线程可以快速地从主库里读取所有的binlog内容。这样一来，即使从库在SQL线程执行完所有读取到的语句前停止运行了，I/O线程也至少完全读取了所有的内容，并将其安全地备份在从库本地的relay log，随时准备在从库下一次启动的时候执行语句。

#### 状态查看

查看主从复制的状态

当主从复制正在进行中时，如果想查看从库两个线程运行状态，可以通过执行在从库里执行”show slave statusG”语句，以下的字段可以给你想要的信息：

Master\_Log\_File — 上一个从主库拷贝过来的binlog文件

Read\_Master\_Log\_Pos — 主库的binlog文件被拷贝到从库的relay log中的位置

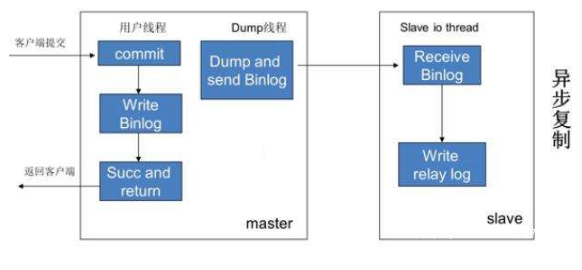
Relay\_Master\_Log\_File — SQL线程当前处理中的relay log文件

Exec\_Master\_Log\_Pos — 当前binlog文件正在被执行的语句的位置

### 复制方式

#### 异步复制

MySQL的复制是异步复制。



MySQL默认的复制即是异步的，主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果crash掉了，此时主上已经提交的事务可能并没有传到从上，如果此时，强行将从提升为主，可能导致新主上的数据不完整。

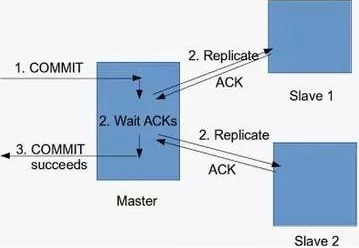


#### 整体/部分复制

MySQL复制可是对整个实例进行复制，也可以对实例中的某个库或者是某个表进行复制。



#### 全同步复制



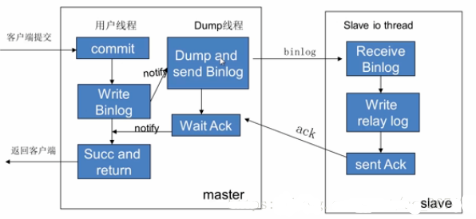
指当主库执行完一个事务，所有的从库都执行了该事务才返回给客户端。因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。

#### 半同步复制

MySQL支持半同步复制。



在MySQL5.5之前，MySQL的复制是异步操作，主库和从库的数据之间存在一定的延迟，这样存在一个隐患：当在主库上写入一个事务并提交成功，而从库尚未得到主库推送的Binlog日志时，主库宕机了，例如主库磁盘损坏、内存故障等造成主库上该事务binlog丢失，此时从库可能损失这个事务，从而造成数据不一致。



为了解决这个问题，MySQL5.5引入了半同步复制机制。在MySQL5.5之前的异步复制时，主库执行完commit提交操作后，在主库写入binlog日志后即可成功返回客户端，无需等待Binlog日志传送给从库。而半同步复制时，为了保证主库上的每一个binlog事务都能够被可靠的复制到从库上，**主库在每次事务成功提交时，并不及时反馈给前端应用用户，而是等待其中一个从库也接收到binlog事务并成功写入到中继日志后，主库才返回commit操作成功给客户端**。半同步复制保证了事务成功提交后，至少有两份日志记录，一份在主库的binlog日志上，一份在至少一个从库的中继日志Relay Log上，从而更进一步保证了数据的完整性。

半同步复制很大程度上取决于主从库之间的网络情况，往返时延RTT（Round-Trip Time）越小决定了从库的实时性越好。通俗地说，主从库之间网络越快，从库越实时。

从半同步复制的流程会发现，半同步复制的“半”就体现在：虽然主库和从库的Binlog日志是同步的，但是主库并不等待从库应用这部分日志就返回提交结果，这部分操作是异步的，从库的数据并不是和主库实时同步的，所以只能称为半同步，而不是完全的实时同步。

mysql主从复制的原理是什么？

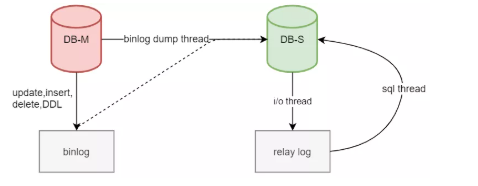
binlog: binary log，主库中保存所有更新事件日志的二进制文件。

## 类型

存在两种复制类型：

### 基于binlog二进制日志的复制

master用户写入数据，生成event记到binary log中。slave接收master上传来的binlog，然后按顺序应用，重现master上的操作。



传统的复制基于(filepos)，当主从发生宕机，切换的时候有问题。

slave保存的是原master上的(filepos)，无法直接指向新master上的(filepos)。

### 使用GTID完成基于事务的复制

主从切换后，在传统方式里，需要找到binlog和POS点，然后执行change master to指向新的主库。对于不是很有经验的运维人员来说，往往会找错，造成主从同步复制报错，在MySQL5.6版本里，无须再找binlog和POS点，只需要知道master的IP、端口、账号和密码即可，因为同步复制是自动的，MySQL会通过内部机制GTID（Global Transaction ID）自动找点同步。

**GTID的工作原理：**

1、master更新数据时，会**在事务前产生GTID，一同记录到binlog日志中**。

2、slave端的i/o线程将变更的binlog，写入到本地的relay log中。

3、**sql线程从relay log中获取GTID，然后对比slave端的binlog是否有记录**。

4、如果有记录，说明该GTID的事务已经执行，slave会忽略。

5、如果没有记录，slave就会从relay log中执行该GTID的事务，并记录到binlog。

6、在解析过程中会判断是否有主键，如果没有就用二级索引，如果没有就用全部扫描。

**优点：**

1、一个事务对应一个唯一ID，**一个GTID在一个服务器上只会执行一次**

2、GTID是用来代替传统复制的方法，**GTID复制与普通复制模式的最大不同就是不需要指定二进制文件名和位置**

3、减少手工干预和降低服务故障时间，当主机挂了之后通过软件从众多的备机中提升一台备机为主机

**限制：**

1、不支持非事务引擎

2、不支持create table ... select 语句复制(主库直接报错)

原理：会生成两个sql，一个是DDL创建表SQL，一个是insert into插入数据的sql。由于DDL会导致自动提交，所以这个sql至少需要两个GTID，但是GTID模式下，只能给这个sql生成一个GTID。

3、不允许一个SQL同时更新一个事务引擎表和非事务引擎表

4、在一个复制组中，必须要求统一开启GTID或者是关闭GTID

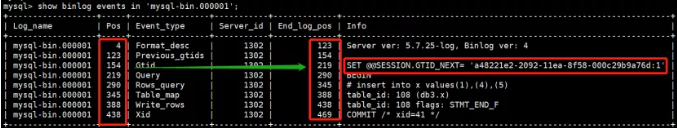
5、开启GTID需要重启（5.7除外）

6、开启GTID后，就不再使用原来的传统复制方式

7、对于create temporary table 和 drop temporary table语句不支持

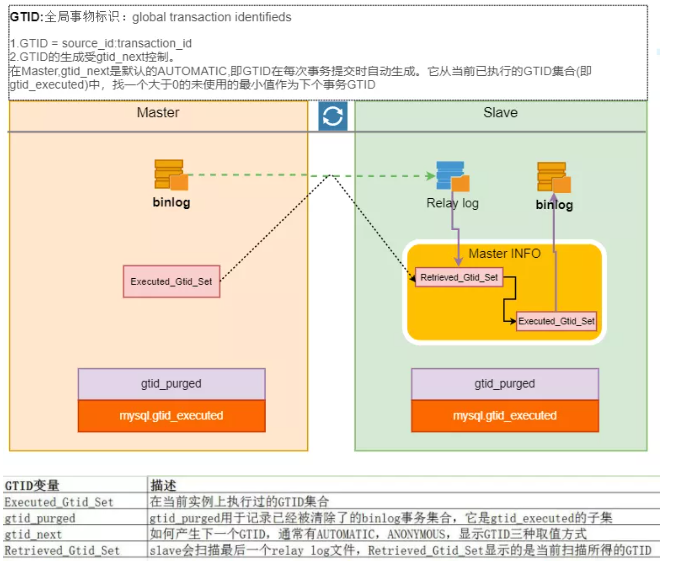
8、不支持sql\_slave\_skip\_counter

### 区别



主从复制，默认是通过pos复制(postion)方式，将用户进行的每一项操作都进行编号(pos)，每一个event都有一个起始编号，一个终止编号。GTID就是类似于pos的一个作用，全局通用并且日志文件里事件的GTID值是一致的。

pos与GTID在日志里是一个标识符，在slave里已不同的方式展现。



GTID的生成受gtid\_next控制。

在Master上，gtid\_next是默认的AUTOMATIC，即GTID在每次事务提交时自动生成。它从当前已执行的GTID集合(即gtid\_executed)中，找一个大于0的未使用的最小值作为下个事务GTID。同时将GTID写入到binlog(set gtid\_next记录)，在实际的更新事务记录之前。

在Slave上，从binlog先读取到主库的GTID(即set gtid\_next记录)，而后执行的事务采用该GTID。

## 特点

### 好处

MySQL复制的优点：

1、如果主库出现问题，可以快速切换到从库提供服务；

2、可以在从库上执行查询操作，降低主库的访问压力；

3、可以在从库上执行备份，以避免备份期间影响主库的服务。

做数据的热备，作为后备数据库，主数据库服务器故障后，可切换到从数据库继续工作，避免数据丢失。

架构的扩展。业务量越来越大，I/O访问频率过高，单机无法满足，此时做多库的存储，降低磁盘I/O访问的评率，提高单个机器的I/O性能。

读写分离，使数据库能支持更大的并发。在报表中尤其重要。由于部分报表sql语句非常的慢，导致锁表，影响前台服务。如果前台使用master，报表使用slave，那么报表sql将不会造成前台锁，保证了前台速度。

**好处一:实现服务器负载均衡**

通过服务器复制功能，可以在主服务器和从服务器之间实现负载均衡。即可以通过在主服务器和从服务器之间切分处理客户查询的负荷，从而得到更好地客户相应时间。通常情况下，数据库管理员会有两种思路。

一是在主服务器上只实现数据的更新操作。包括数据记录的更新、删除、新建等等作业。而不关心数据的查询作业。数据库管理员将数据的查询请求全部 转发到从服务器中。这在某些应用中会比较有用。如某些应用，像基金净值预测的网站。其数据的更新都是有管理员更新的，即更新的用户比较少。而查询的用户数 量会非常的多。此时就可以设置一台主服务器，专门用来数据的更新。同时设置多台从服务器，用来负责用户信息的查询。将数据更新与查询分别放在不同的服务器 上进行，即可以提高数据的安全性，同时也缩短应用程序的响应时间、提高系统的性能。

二是在主服务器上与从服务器切分查询的作业。在这种思路下，主服务器不单单要完成数据的更新、删除、插入等作业，同时也需要负担一部分查询作 业。而从服务器的话，只负责数据的查询。当主服务器比较忙时，部分查询请求会自动发送到从服务器重，以降低主服务器的工作负荷。当然，像修改数据、插入数 据、删除数据等语句仍然会发送到主服务器中，以便主服务器和从服务器数据的同步。

**好处二：通过复制实现数据的异地备份**

可以定期的将数据从主服务器上复制到从服务器上，这无疑是先了数据的异地备份。在传统的备份体制下，是将数据备份在本地。此时备份 作业与数据库服务器运行在同一台设备上，当备份作业运行时就会影响到服务器的正常运行。有时候会明显的降低服务器的性能。同时，将备份数据存放在本地，也 不是很安全。如硬盘因为电压等原因被损坏或者服务器被失窃，此时由于备份文件仍然存放在硬盘上，数据库管理员无法使用备份文件来恢复数据。这显然会给企业 带来比较大的损失。

而如果使用复制来实现对数据的备份，就可以在从服务器上对数据进行备份。此时不仅不会干扰主服务气的正常运行，而且在备份过程中主服务器可以继 续处理相关的更新作业。同时在数据复制的同时，也实现了对数据的异地备份。除非主服务器和从服务器的两块硬盘同时损坏了，否则的话数据库管理员就可以在最 短时间内恢复数据，减少企业的由此带来的损失。

**好处三：提高数据库系统的可用性**

数据库复制功能实现了主服务器与从服务器之间数据的同步，增加了数据库系统的可用性。当主服务器出现问题时，数据库管理员可以马上让从服务器作为主服务器，用来数据的更新与查询服务。然后回过头来再仔细的检查主服务器的问题。此时一般数据库管理员也会采用两种手段。

一是主服务器故障之后，虽然从服务器取代了主服务器的位置，但是对于主服务器可以采取的操作仍然做了一些限制。如仍然只能够进行数据的查询，而 不能够进行数据的更新、删除等操作。这主要是从数据的安全性考虑。如现在一些银行系统的升级，在升级的过程中，只能够查询余额而不能够取钱。这是同样的道理。

二是从服务器真正变成了主服务器。当从服务器切换为主服务器之后，其地位完全与原先的主服务器相同。此时可以实现对数据的查询、更新、删除等操 作。为此就需要做好数据的安全性工作。即数据的安全策略，要与原先的主服务器完全相同。否则的话，就可能会留下一定的安全隐患。

### 问题

主从复制也带来其他一系列性能瓶颈问题：

1、写入无法扩展

2、写入无法缓存

3、复制延时

4、锁表率上升

5、表变大，缓存率下降

针对上述问题，引入数据库垂直分区和水平分区解决。

针对复制延时问题，数据库引入最大可用、最大性能等策略，但是无法权衡安全和高效，即如果实现RPO=0则必然效率会低。阿里OceanBase能够做到RPO=0。

## 查看状态

查看主从复制的状态

当主从复制正在进行中时，如果想查看从库两个线程运行状态，可以通过执行在从库里执行”show slave statusG”语句，以下的字段可以给你想要的信息：

Master\_Log\_File—上一个从主库拷贝过来的binlog文件

Read\_Master\_Log\_Pos—主库binlog文件被拷贝到从库的relay log中的位置

Relay\_Master\_Log\_File—SQL线程当前处理中的relay log文件

Exec\_Master\_Log\_Pos—当前binlog文件正在被执行的语句的位置

## 复制拓扑

复制存在多种拓扑结构：一主库多备库，主-主复制，环形复制，树或金字塔形等，3种常见的架构包括：一主多从复制架构、多级复制架构和双主复制/Dual Master架构。

### 一主多从复制架构

在主库读取请求压力非常大的场景下，可以通过配置一主多从复制架构实现读写分离，把大量对实时性要求不是特别高的读请求通过负载均衡分布到多个从库上，降低主库的读取压力。

在主库出现异常宕机的情况下，可以把一个从库切换为主库继续提供服务。

### 多级复制架构

一主多从的架构能够解决大部分读请求压力特别大的场景的需求，考虑到MySQL的复制是主库“推送”Binlog日志到从库，主库的I/O压力和网络压力会随着从库的增加而增长（每个从库都会在主库上有一个独立的Binlog Dump线程来发送事件），而多级复制架构解决了一主多从场景下，主库额外的I/O和网络压力。

对比一主多从的架构，多级复制架构仅仅是在主库master1复制到从库slave1、slave2、slave3的中间增加一个二级主库master2，主库只需要给一个从库master2“推送”binlog日志即可，减轻主库master1的压力。二级主库master2再“推送”binlog日志给从库slave1、slave2、slave3。

多级复制架构解决了一主多从场景下，主库的I/O负载和网络压力，其缺点：MySQL的复制是异步复制，多级复制场景下主库的数据是经历两次复制才到达从库的，期间的延时比一主多从复制场景下只经历一次复制的要大。

### 双主复制/Dual Master复制

双主/Dual Master架构特别适用于DBA做维护等需要主从切换的场景，通过双主/Dual Master架构避免了重复搭建从库的麻烦。

## 容量规划

## 复制管理

## 复制启动选项

## 日常运维

### 查看主从复制的状态

当主从复制正在进行中时，如果想查看从库两个线程运行状态，可以通过执行在从库里执行”show slave statusG”语句，以下的字段可以给你想要的信息：

Master\_Log\_File — 上一个从主库拷贝过来的binlog文件

Read\_Master\_Log\_Pos — 主库的binlog文件被拷贝到从库的relay log中的位置

Relay\_Master\_Log\_File — SQL线程当前处理中的relay log文件

Exec\_Master\_Log\_Pos — 当前binlog文件正在被执行的语句的位置

### 主从复制延迟

在异步或半同步的复制结构中，从库出现延迟是一件十分正常的事。

虽出现延迟正常，但是否需要关注，则一般是由业务来评估。

如：从库上有需要较高一致性的读业务，并且要求延迟小于某个值，那么则需要关注。

**复制逻辑：**

1、主库将对数据库实例的变更记录到binlog中。

2、主库会有binlog dump线程实时监测binlog的变更并将这些新的events推给从库（Master has sent all binlog to slave; waiting for more updates）

3、从库的IO Thread接收这些events，并将其记录入relaylog。

4、从库的SQL Thread读取relaylog的events，并将这些events应用（或称为重放）到从库实例。

上述为默认的异步复制逻辑，半同步复制又有些许不同，此处不再赘述。

#### 判断从库延迟

此外，判断从库有延迟是十分简单的一件事：

在从库上通过SHOW SLAVE STATUS

检查Seconds\_Behind\_Master值即可。

#### 原因及处理思路

产生延迟的原因及处理思路

##### 主库DML请求频繁（tps较大）

即主库写请求较多，有大量insert、delete、update并发操作，短时间产生了大量的binlog。

**原因分析：**

主库并发写入数据，而从库SQL Thread为单线程应用日志，很容易造成relaylog堆积，产生延迟。

**解决思路：**

做sharding，通过scale out打散写请求。或考虑升级到MySQL 5.7+，开启基于逻辑时钟的并行复制。

##### 主库执行大事务

比如大量导入数据，INSERT INTO $tb1 SELECT \* FROM $tb2、LOAD DATA INFILE等

比如UPDATE、DELETE了全表等

Exec\_Master\_Log\_Pos一直未变，Slave\_SQL\_Running\_State为Reading event from the relay log

分析主库binlog，看主库当前执行的事务也可知晓。

**原因分析：**

假如主库花费200s更新了一张大表，在主从库配置相近的情况下，从库也需要花几乎同样的时间更新这张大表，此时从库延迟开始堆积，后续的events无法更新。

**解决思路：**

拆分大事务，及时提交。

##### 主库对大表执行DDL语句

现象和主库执行大事务相近。

检查Exec\_Master\_Log\_Pos一直未动，也有可能是在执行DDL。

分析主库binlog，看主库当前执行的事务也可知晓。

**原因分析：**

1、DDL未开始，被阻塞，SHOW SLAVE STATUS检查到Slave\_SQL\_Running\_State为waiting for table metadata lock，且Exec\_Master\_Log\_Pos不变。

2、DDL正在执行，SQL Thread单线程应用导致延迟增加。Slave\_SQL\_Running\_State为altering table，Exec\_Master\_Log\_Pos不变

**解决思路：**

通过processlist或information\_schema.innodb\_trx来找到阻塞DDL语句的查询，干掉该查询，让DDL正常在从库执行。

DDL本身造成的延迟难以避免，建议考虑：

① 业务低峰期执行

② set sql\_log\_bin=0后，分别在主从库上手动执行DDL（此操作对于某些DDL操作会造成数据不一致，请务必严格测试）

##### 主库与从库配置不一致

**原因分析：**

硬件上：主库实例服务器使用SSD，而从库实例服务器使用普通SAS盘、cpu主频不一致等

配置上：如RAID卡写策略不一致，OS内核参数设置不一致，MySQL落盘策略不一致等

**解决思路：**

尽量统一DB机器的配置（包括硬件及选项参数）

甚至对于某些OLAP业务，从库实例硬件配置高于主库等

##### 表缺乏主键或唯一索引

binlog\_format=row的情况下，如果表缺乏主键或唯一索引，在UPDATE、DELETE的时候可能会造成从库延迟骤增。

此时Slave\_SQL\_Running\_State为Reading event from the relay log。

并且SHOW OPEN TABLES WHERE in\_use=1的表一直存在。

Exec\_Master\_Log\_Pos不变。

mysqld进程的cpu几近100%（无读业务时），io压力不大

**原因分析：**

做个极端情况下的假设，主库更新一张500w表中的20w行数据，该update语句需要全表扫描

而row格式下，记录到binlog的为20w次update操作，此时SQL Thread重放将特别慢，每一次update可能需要进行一次全表扫描

**解决思路：**

检查表结构，保证每个表都有显式自增主键，并建立合适索引。

##### 从库自身压力过大

**原因分析：**

从库执行大量select请求，或业务大部分select请求被路由到从库实例上，甚至大量OLAP业务，或者从库正在备份等。

此时可能造成cpu负载过高，io利用率过高等，导致SQL Thread应用过慢。

**解决思路：**

建立更多从库，打散读请求，降低现有从库实例的压力。

##### MyISAM存储引擎

此时从库Slave\_SQL\_Running\_State为Waiting for table level lock

**原因分析：**

MyISAM只支持表级锁，并且读写不可并发操作。

主库在设置@@concurrent\_insert对应值的情况下，能并发在select时执行insert，但从库SQL Thread重放时并不可并发，有兴趣可以再去看看myisam这块的实现。

**解决思路：**

当然是选择原谅它了，既然选择了MyISAM，那么也应该要有心理准备。（还存在其他场景，也不推荐MyISAM在复制结构中使用）

改成InnoDB吧。

##### 总结

通过SHOW SLAVE STATUS与SHOW PROCESSLIST查看现在从库的情况。（顺便也可排除在从库备份时这种原因）

若Exec\_Master\_Log\_Pos不变，考虑大事务、DDL、无主键，检查主库对应的binlog及position即可。

若Exec\_Master\_Log\_Pos变化，延迟逐步增加，考虑从库机器负载，如io、cpu等，并考虑主库写操作与从库自身压力是否过大。

当然，Seconds\_Behind\_Master也不一定准确，存在在少部分场景下，虽Seconds\_Behind\_Master为0，但主从数据不一致的情况。

# 基于日志点的复制

在Master端建立复制用户

备份Master端的数据，并在Slave端恢复

使用Change master命名配置复制

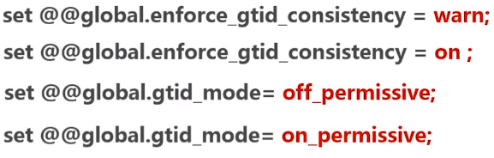
# 在线变更复制类型

在线将基于日志的复制变更为基于事务的复制

先决条件：集群中所有服务器的版本均高于5.7.6

集群中所有服务器gtid\_mode都设为off

处理步骤：





在线将基于事务的复制变更为基于日志的复制

先决条件：集群中所有服务器的版本均高于5.7.6

集群中所有服务器gtid\_mode都设为on

处理步骤：





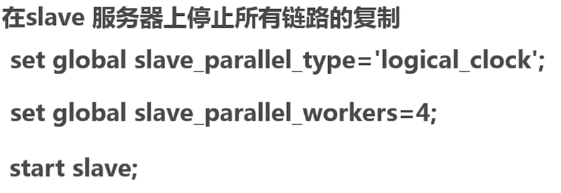
# 多源复制

多源复制也就是多Master复制，允许一个Slave对应多个Master

如何使用多源复制：



# 多线程复制



# mysqldump

<https://dev.mysql.com/doc/refman/5.7/en/mysqldump.html>