大家好，我是马听，个人公众号：MySQL数据库联盟。

DBA需要掌握多种技能，其中MySQL主从复制是DBA必须掌握的关键技术之一。

这一份资料，就来全面介绍一下MySQL主从复制。

包括原理、安装以及维护。

主从复制会涉及大量实验，建议大家可以用这篇文稿，再结合讲解视频来学习。

慕课网视频学习地址如下（免费）：

https://www.imooc.com/learn/1397。

这份资料目录如下：

1-1 深入剖析主从复制的底层原理

1 什么是MySQL主从复制?

是指将一个MySQL数据库A的数据变更复制到另外一个MySQL实例B

这个A就可以称为主库，B称为从库

2 主从复制的作用

数据备份和恢复

可以将主库上的数据复制到一个或多个从库上。这个时候，从库可以充当一个灾备的副本。当主库宕机时，可以在从库恢复数据。

高可用

当主库发生故障时，提升一个从库为新的主，保证业务可用。

目前很多高可用架构，就是基于主从来实现的，比如双主+keepalived、MHA、Orchestrator。

负载均衡

可以将读操作在从库执行，从而减轻主库的压力。

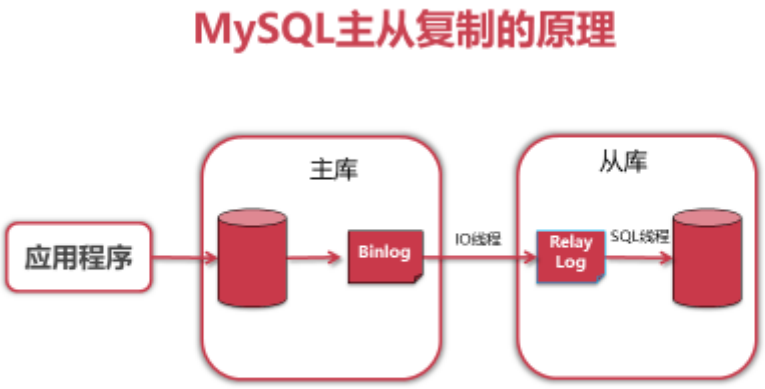
3 MySQL主从复制的原理

主库必须开启二进制日志

当主库有写操作时（比如insert、update，delete），会记录到主库的Binlog中

从库通过IO线程读取主库的Binlog里面的内容，传给从库的Relay Log（中继日志）

从库的sql线程负责读取它的relay log里的信息并应用到数据库中



4 两种复制方法

基于位点的复制

从库根据主库的位点信息进行执行回放。

这些位点信息包括二进制日志文件名（File Name）和事件位置（Position）。

比如我们执行

```
show master status\G
```

File就是当前实例最后一个事务写的Binlog文件

Position 就是最后一个事务在Binlog中的位点

Binlog_Do_DB 表示需要记录Binlog的数据库，可以通过参数配置，我们会在这一章后面的内容详细介绍

Binlog_Ignore_DB 表示不会记录Binlog的数据库

Executed_Gtid_Set 表示执行过的GTID（全局事务标识符）集合

基于GTID的复制

GTID是MySQL 5.6 引入的一种全局事务标识

在基于 GTID 的复制中，主库上，每个事务都会被分配一个唯一的GTID，从库使用这些 GTID 来同步主库的事务。

1-2 如何快速搭建主从复制?

1 部署过程

确定主库Binlog是否开启

修改server_id

创建复制用户

主库导出数据

可以考虑使用mysqldump、xtrabackup、Clone Plugin

从库导入数据

确定备份时的位点

在从库配置主库信息

查看复制状态

测试数据是否同步

详细步骤如下:

1 准备

确定主库Binlog是否开启

```
show global variables like "log_bin";
```

并且查看配置文件中，Binlog是否开启，防止重启后，Binlog未开启的情况

```
vim /data/mysql/conf/my.cnf
```

搜索

```
log-bin
```

修改server_id

server_id在主从或者其他的高可用架构中，用来区分不同的实例。同一个架构中，不同的实例，server_id一定不能相同，一般建议把server_id改成ip的后两段，这样，同机房，重复的概率就很低了。

查询server_id可以执行

```
select @@global.server_id;
```

可以设置的范围是1到2的32次方减1

可以先动态修改，

```
set global server_id=15270;
```

然后再修改配置文件

```
vim /data/mysql/conf/my.cnf
```

搜索

server-id

关闭GTID

vim /data/mysql/conf/my.cnf

```
gtid_mode=off

#enforce_gtid_consistency=on
```

创建复制用户

在主库创建复制用户

```
CREATE USER 'repl'@'%' identified  WITH mysql_native_password BY 'Uid_dQc63';
GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
```

主库导出数据

```
cd /data/backup/
mysqldump -uroot -p --single-transaction --all-databases --master_data=2 --set-
gtid-purged=OFF >alldb_bak.sql
```

其中:

--single-transaction，开启一个事务进行备份，并且把会话隔离级别设置成REPEATABLE READ。在备份时不阻塞应用的情况下，也可以保证数据一致性

--all-databases，备份所有的库

--master-data，记录备份时的位点，会记录位点，只是以注释的方式记录

传输导从库

```
scp alldb_bak.sql 192.168.152.30:/data/backup
```

从库导入数据

```
cd /data/backup
mysql -uroot -p <alldb_bak.sql
```

确定备份时的位点

```
head -n 30 alldb_bak.sql
```

在从库配置主库信息

```
CHANGE MASTER TO
MASTER_HOST='192.168.152.70',
MASTER_USER='repl',
MASTER_PASSWORD='Uid_dQc63',
MASTER_LOG_FILE='mysql-bin.000020',
MASTER_LOG_POS=490037;

start slave;
```

查看复制状态

```
SHOW SLAVE STATUS\G
```

再来解释各个字段的含义：

```
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event    //从库IO线程的状
态，这个状态表示正在等待主库发送事件
                  Master_Host: 192.168.152.70          //主库的IP地址或主机名
                  Master_User: repl                    //用于从库连接到主库的用户名
                  Master_Port: 3306                    //主库的端口
                Connect_Retry: 60                        //如果从库与主库之间的连接中
断，从库尝试重新连接的时间间隔
              Master_Log_File: mysql-bin.000025        //主库当前正在写入的二进制日志
文件的名称
          Read_Master_Log_Pos: 534                    //从库复制的主库二进制日志文件的
位置
               Relay_Log_File: mysql-relay-bin.000002 //从库正在写入的中继日志文件的名
称
                Relay_Log_Pos: 662                    //从库正在写入中继日志文件的位置
        Relay_Master_Log_File: mysql-bin.000025        //从库复制的主库二进制日志文件
的名称
             Slave_IO_Running: Yes                    //从库的I/O线程是否正在运行
            Slave_SQL_Running: Yes                    //从库的SQL线程是否正在运行
              Replicate_Do_DB:                         //需要复制的数据库
          Replicate_Ignore_DB:                         //不需要复制的数据库
           Replicate_Do_Table:                         //需要复制的表
       Replicate_Ignore_Table:                         //不需要复制的表
      Replicate_Wild_Do_Table:                         //需要复制的通配符表名称
  Replicate_Wild_Ignore_Table:                         //需要复制的通配符表名称
                   Last_Errno: 0                        //上一条错误的错误码
                   Last_Error:                         //上一条错误信息
                 Skip_Counter: 0                        //跳过的事件数量
          Exec_Master_Log_Pos: 534                        //SQL线程当前正在执行的主库
二进制日志文件的位置
              Relay_Log_Space: 871                        //中继日志文件的总大小
              Until_Condition: None       // 执行 start slave until 的条件，我们有时
候，创建主从之后，启动同步，
```

```
                                          //是不需要完全同步的，只要同步到某个事务，比如误操作恢复，只要同步到误
                                          //操作前一个事务就行

               Until_Log_File:             //执行until所到的binlog文件

                Until_Log_Pos: 0
             //执行until所到的位点
           Master_SSL_Allowed: No           //是否允许使用SSL链接到主库
           Master_SSL_CA_File:               // SSL连接所需的CA证书文件
           Master_SSL_CA_Path:              //SSL连接所需的CA证书文件夹路径
              Master_SSL_Cert:               //用于SSL连接的证书文件路径
            Master_SSL_Cipher:               //用于SSL连接的加密算法
               Master_SSL_Key:               //用于SSL连接的私钥文件路径
        Seconds_Behind_Master: 0              //从库落后于主库的时间
Master_SSL_Verify_Server_Cert: No              //是否验证主服务器证书的SSL连接。值为"No"，表示不验证
                Last_IO_Errno: 0              //上一次I/O错误的错误代码
                Last_IO_Error:               //上一次I/O错误的错误信息
               Last_SQL_Errno: 0              //上一次SQL错误的错误代码
               Last_SQL_Error:               //上一次SQL错误的错误信息
  Replicate_Ignore_Server_Ids:               //忽略某台机器的server_id
             Master_Server_Id: 15270          //主库的server_id
                  Master_UUID: ec992f67-a08b-11ed-9a9f-024255bd70b6    //主服务器的UUID
             Master_Info_File: mysql.slave_master_info    //保存主服务器信息的文件
                    SQL_Delay: 0        //如果有设置复制延迟，则显示延迟的时间量
          SQL_Remaining_Delay: NULL         //剩余的复制延迟时间
      Slave_SQL_Running_State: Slave has read all relay log; waiting for more updates
      //从服务器当前的SQL线程状态，Slave has read all relay log; waiting for more updates"，表示从服务器已读取所有中继日志，并正在等待更多更新
           Master_Retry_Count: 86400        //主服务器重试连接的次数
                  Master_Bind:              //副本绑定到的网络接口
      Last_IO_Error_Timestamp:              //上一次I/O错误发生的时间戳
     Last_SQL_Error_Timestamp:              //上一次SQL错误发生的时间戳
               Master_SSL_Crl:              //SSL相关
           Master_SSL_Crlpath:              //SSL相关
           Retrieved_Gtid_Set:              //检索到的GTID集
            Executed_Gtid_Set: ec992f67-a08b-11ed-9a9f-024255bd70b6:1-2613611
//已经执行的GTID集合
                Auto_Position: 0            //是否启动自动位置试别
         Replicate_Rewrite_DB:              //重写复制操作中的数据库名

                 Channel_Name:              // 复制通道的名称
           Master_TLS_Version:              //主服务器使用的TLS版本
       Master_public_key_path:              //服务器公钥文件的路径
        Get_master_public_key: 0            //是否获取主服务器的公钥
            Network_Namespace:              // 网络命名空间
```

如果Slave_IO_Running 和Slave_SQL_Running 都是yes，就表示复制已经建立

测试数据是否同步

```
use martin
-- 主库执行
create table repl_test(id int);

-- 从库确定
show tables

select * from repl_test;

-- 主库写入数据
insert into repl_test select 1;

-- 从库查看
select * from repl_test;
```

主库使用Xtrabackup进行全备

```
xtrabackup --defaults-file=/data/mysql/conf/my.cnf -uroot -p --backup --
stream=xbstream --target-dir=./ >/data/backup/xtrabackup.xbstream
```

输入密码

将全备传到从库

备库创建恢复目录

```
mkdir /data/backup/recover
```

主库把备库传到从库

```
scp xtrabackup.xbstream 192.168.152.30:/data/backup/recover
```

清空新实例

关闭新建的实例

```
/etc/init.d/mysql.server stop
```

清空新实例数据目录和事务日志目录：

```
rm /data/mysql/data/* -rf
rm /data/mysql/binlog/* -rf
```

把全备导入新实例

并把全备恢复到新的 MySQL 中

```
cd /data/backup/recover
xbstream -x < xtrabackup.xbstream
xtrabackup --prepare --target-dir=./

xtrabackup --defaults-file=/data/mysql/conf/my.cnf --copy-back --target-dir=./
```

启动 MySQL

```
chown -R mysql.mysql /data/mysql
/etc/init.d/mysql.server start
```

查看备份时的位点信息

```
cat xtrabackup_binlog_info
```

创建复制关系

```
CHANGE MASTER TO
MASTER_HOST='192.168.152.70',
MASTER_USER='repl',
MASTER_PASSWORD='Uid_dQc63',
MASTER_LOG_FILE='mysql-bin.000023',
MASTER_LOG_POS=196;

start slave;
```

查看主从状态

```
show slave status\G
```

测试同步

主库写入

```
insert into repl_test select 2;
```

从库查询

```
select * from repl_test;
```

1-3 基于GTID的复制如何搭建?

1 GTID是什么?

GTID的组成

如果我们开启GTID，每个事务，都会有一个全局唯一的编号。

GTID实际是由UUID+TID组成的

```
show global variables like "%gtid_purged%";
```

再来看MySQL的UUID

```
show global variables like "server_uuid";
```

是一个MySQL实例的唯一标识，

比如我们在另外一个实例执行，就可以看到一个不一样的uuid

MySQL在第一次启动时，会自动持久化到data目录下的auto.cnf这个文件里

我们可以来查看一下

```
cat /data/mysql/data/auto.cnf
```

而gtid后面的tid代表了该实例已经提交的事务数量

再来看下控制gtid开启和关闭的参数gtid_mode

有四个值:

off：新的事物和复制的事物，都是没有GTID值的，没有GTID的事务，可以称为匿名事务

off_permissive：新的事物是匿名事务，复制中的事务可以是匿名事务，也可以是GTID事务

on_permissive：新的事务是GITD事务，复制中的事务可以是匿名事务，也可以是GTID事务

on：新事务和复制事务都是GTID事务。

并且只能是相邻的两个参数才能修改，比如要把gtid_mode从off改为on需要先改为off_permissive，再改为on_permissive，最后再改成on。

gtid_executed

用来存储当前实例上已经执行的事务GTID集合

除了执行下面的语句查看gtid_executed

```
show global variables like "gtid_executed";
```

还可以执行

```
select * from mysql.gtid_executed;
```

来查看已经执行事务的gtid集合

enforce_gtid_consistency

有三个值:

OFF:所有事务都允许违反GTID一致性

ON:不允许事务违反GTID一致性

WARN:所有事务都允许违反GTID一致性，但是会给出警告，并且会打印在错误日志里。

这里，我们可以模拟一条不符合GTID规范的SQL，

首先把ENFORCE_GTID_CONSISTENCY设置为ON

```
create table gtid_test1(a int);
create table gtid_test2(a int)engine=myisam;
-- 打印error log
-- 再执行事务
 begin;
 insert into gtid_test1 select 1;
 insert into gtid_test2 select 1;
 commit;
```

来看一下报错是怎样的。

把ENFORCE_GTID_CONSISTENCY设置为WARN

再来执行不符合GTID规范的SQL

```
 begin;
 insert into gtid_test1 select 1;
 insert into gtid_test2 select 1;
 commit;
```

这里就直接报错，不让写入了

2 基于GTID复制的优势

可以知道事务在哪个实例上提交的

比如主从，或者其他高可用架构，比如MGR，集群通过GTID来标记事务，用来跟踪每个实例上提交的事务。

比较方便进行复制结构上的故障转移

在基于位点的复制中，比如一主两从的架构中，当主库出现故障，提升一个新的主之后，其他的从默认并不知道新的主的位点信息。需要借助其他工具，比如MHA或者Orchestrator。

而有了GTID，搭建主从，或者故障转移，就不需要再关注BInlog的位点信息。

很方便判断主从一致性

因为复制完全基于事务的

3 使用GTID的注意事项

事务和非事务引擎不能在同一个事物里

因为混合到一起，可能会导致一个事务分配多个GTID

create table ... select

MySQL 8.0.21之前，如果Binlog是row格式，这个语句实际上被记录为两个独立事件，一个用于创建表，另一个用于从源表中的行插入到刚刚创建的新表，所以不能使用。

从MySQL 8.0.21开始，这条SQL在Binlog会被记录成一个原子DDL。

建议设置--enforce-gtid-consistency

为了防止会导致GTID复制失败的语句执行，所有的服务器都必须在开启gtid的时候，开启

--enforce-gtid-consistency

4 位点复制改成GTID复制（一）

查看GTID状态

```
show  global variables like 'gtid_mode';
show  global variables like 'enforce_gtid_consistency';
```

修改enforce_gtid_consistency=warn

```
SET global ENFORCE_GTID_CONSISTENCY = WARN;
```

主和从都要执行

做这一步骤的目的是让服务器正常运行一段时间，如果有不符合GITD一致性的语句，提醒我们，就可以考虑调整代码。

防止后面改成GTID模式，导致业务SQL执行失败。

修改enforce_gtid_consistency=on

```
SET global ENFORCE_GTID_CONSISTENCY = ON;
```

如果违反了gtid一致性，直接报错

5 位点复制改成GTID复制（二）

修改gtid_mode=off_permissive

```
SET GLOBAL GTID_MODE = OFF_PERMISSIVE;
```

修改gtid_mode=on_permissive

每台机器执行

```
SET GLOBAL GTID_MODE = ON_PERMISSIVE;
```

查看正在执行的匿名事务数量

等待每台机器的ONGOING_ANONYMOUS_TRANSACTION_COUNT为0

```
SHOW STATUS LIKE 'ONGOING_ANONYMOUS_TRANSACTION_COUNT';
```

6 位点复制改成GTID复制（三）

刷新日志

```
flush logs
```

开启GTID

主从都开启GTID

```
SET GLOBAL GTID_MODE = ON;
```

参数持久化

将gtid_mode=ON和 添加enforce_gtid_consistency=ON到 my.cnf。

7 位点复制改成GTID复制（四）

修改复制参数

从库执行

```
stop slave;
CHANGE MASTER TO MASTER_AUTO_POSITION = 1;
START SLAVE;
```

查看复制状态

```
show slave status\G
```

测试同步

```
create table gtid_test3(id int);
```

show slave status\G可以看到，Retrieved_Gtid_Set新增了内容，也就是收到的GTID集合

再执行一个事务

```
create table gtid_test4(id int);
```

Retrieved_Gtid_Set就变成了一个范围

8 从0部署GTID复制（一）

确定主库Binlog是否开启

```
show global variables like "log_bin";
```

并且查看配置文件中，Binlog是否开启，防止重启后，Binlog未开启的情况

```
vim /data/mysql/conf/my.cnf
```

搜索

```
log-bin
```

修改server_id

确定主从的server_id是否相同

```
select @@global.server_id;
```

动态修改

```
set global server_id=15270;
```

再修改配置文件

```
vim /data/mysql/conf/my.cnf
```

从库做同样的操作

开启GTID

先动态修改

```
SET GLOBAL GTID_MODE = OFF_PERMISSIVE;
SET GLOBAL GTID_MODE = ON_PERMISSIVE;
SET GLOBAL GTID_MODE = ON;
set global enforce_gtid_consistency=ON;
```

再修改配置文件，让配置持久化

```
gtid_mode=ON
enforce_gtid_consistency=ON
```

9 从0部署GTID复制（二）

创建复制用户

```
CREATE USER 'repl'@'%' identified WITH mysql_native_password BY 'Uid_dQc63';
GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
```

当然，这个我们在之前讲基于位点的主从复制部署的时候，创建了这个复制用户的，所以可以不执行。

主库导出数据

可以考虑使用mysqldump、xtrabackup、Clone Plugin，我们使用mysqldump

```
cd /data/backup/
mysqldump -uroot -p --single-transaction --all-databases --master-data=2 --set-
gtid-purged=on >alldb_bak_for_gtid.sql
```

把备份传到从库

```
scp alldb_bak_for_gtid.sql 192.168.152.30:/data/backup
```

从库导入数据

清空从库的gtid_executed，执行

```
reset master;
```

再导入数据

```
cd /data/backup
mysql -uroot -p <alldb_bak_for_gtid.sql
```

10 从0部署GTID复制（三）

在从库配置主库信息

```
stop slave;
reset slave;

change master to
master_host='192.168.152.70',
master_user='repl',
master_password='Uid_dQc63',
MASTER_AUTO_POSITION=1;

start slave;
```

查看复制状态

```
show slave status\G
```

测试数据是否同步

主库执行

```
create table t1_gtid(id int);
insert into t1_gtid select 1;
```

从库查询数据

```
select * from t1_gtid;
show slave status\G
```

1-4 如何进行多线程复制的配置?

在传统的复制模式下，我们也许经常会遇到主从延迟的场景。

这种情况，就非常不利于读写分离的业务场景，也就是写在主库，读在从库。

并且假如通过主从实现的高可用架构，如果长时间有延迟，也不太利于主从切换。因为即使切换了，从库也是少了一部分数据的。

1 主从延迟的主要原因

那究竟是什么原因导致的主从延迟呢?

在主库，可能有多个客户端并发往MySQL中写入数据，但是把Relay Log中的数据写入到从库MySQL，却只有单个SQL线程进行操作，这也是导致主从延迟的主要原因。

当然，在后面，我们会花一节介绍导致主从延迟的其他原因。

这时候，我们可能就会考虑，是不是可以通过多个SQL线程来消化主库写入的事务呢?

2 关于多线程的思考

这里我们就来一起思考一下，假如让我们设计多个SQL线程并发执行，需要考虑哪些细节。

能不能把事务依次发给不同的线程?

首先来问大家，事务能不能依次发给不同的SQL线程执行。

比如第一个事务发送给线程1，第二个事务发送个线程2

其实不行的，因为，事务被分开执行后，可能在从库会出现第二个事务先执行完的情况，如果两个事务正好是操作同一行，因为主库和从库的执行顺序不一样，就可能导致主从数据不一致。

同一个事务的多个更新，能不能分给不同的线程执行

也不行，比如这个事务更新了同一张表的不同行，如果两条更新语句分到不同的线程，假如某一行更新完成，另外一行还没更新，这个时候，从库查询这张表的数据，就会发现和主库不一致，这样也是不行的。

多线程要满足的条件

根据上面两个问题，我们来总结一下，多线程要满足的两个基本条件。

更新同一行的两个事务，必须被分到同一个线程里

同一个事务不能被拆开，必须放到同一个线程里

3 MySQL 5.6的并行复制

支持库级别的并行复制

从MySQL 5.6 开始，支持库级别的并行复制

配置方式

```
show global variables like "slave_parallel_workers";
```

参数slave_parallel_workers，表示并发的从库SQL线程数量

跟CPU核数对应

```
set global slave_parallel_workers=4;
stop slave;
start slave;
```

再来查看线程数

```
show processlist;
```

可以看到，就有4个复制线程了（这里可以结合自己的实验，或者可以到课程视频里看下效果）。

5.6并行复制的不足

如果表都集中在一个 DB 里，或者热点表集中在一个库中，那就没有什么效果了。

4 MySQL组提交

在学完日志那一章有关Redo Log和Binlog的内容之后，我们知道Redo Log和Binlog的刷盘速度，影响着MySQL的 TPS，也就是每秒事务数。

所以，MySQL就增加了一种组提交机制，多个事务放在一起进行落盘，这样就能大大提升MySQL的TPS。

我们在MySQL日志那一章，介绍了MySQL 两阶段提交分别是准备阶段和提交阶段。

在执行更新语句时，会将更新操作记录到 Redo Log中，此时Redo Log处于 prepare状态。会通知执行器执行完成了，随时可以提交事务。

执行器生成这个操作的Binlog，并把Binlog写入磁盘

执行器调用引擎的提交事务接口，引擎把刚刚写入的Redo Log改成提交状态。更新完成。

我们可以把两阶段提交理解下方图片右边的这种形式：

组提交中：

在第三步的redo log落盘，带上多个事务。

在第四步的binlog落盘，也带上多个事务。

同时处于redo log prepare 状态的事务，可以一起落盘，因为已经通过了冲突检测；

处于 prepare 状态的事务，与处于 commit 状态的事务之间，也是可以一起落盘。

这样可以减少IOPS的消耗。

5 MySQL 5.7的并行复制

5.7并行复制原理

基于组提交的并行复制，同时处于redo log prepare 状态的事务，在备库执行时是可以并行的，因为已经通过了冲突检测；处于 prepare 状态的事务，与处于 commit 状态的事务之间，在备库执行时也是可以并行的。

配置方式

5.7 增加参数：slave_parallel_type。用来控制并行复制的类型。

配置为 DATABASE，表示使用 MySQL 5.6 版本的按库并行策略；

配置为 LOGICAL_CLOCK，表示采用5.7新的并行复制逻辑

```
set global slave_parallel_workers=8;
stop slave sql_thread;
set global slave_parallel_type=logical_clock;
start slave sql_thread;
flush logs;
```

配置后对比

```
show processlist
```

6 MySQL 5.7.22及之后版本的并行复制

MySQL 5.7.22 版本里，MySQL 增加了一个新的并行复制策略，基于 WRITESET 的并行复制。

MySQL 5.7.22的并行复制介绍

新增了一个参数 binlog_transaction_dependency_tracking，用来控制是否启用这个新策略。这个是在主库配置的，这个参数的可选值有以下三种。

COMMIT_ORDER，表示的就是前面介绍的，根据同时进入 prepare 和 commit 来判断是否可以并行的策略。

WRITESET，表示的是对于事务涉及更新的每一行，计算出这一行的 hash 值，组成集合 writeset。如果两个事务没有操作相同的行，也就是说它们的 writeset 没有交集，就可以并行。

哈希值使用 唯一索引+库名+表名等信息，调用一个哈希函数得到的。

如果表中有多个唯一索引，那每个唯一索引都有一个哈希值。

WRITESET_SESSION，是在 WRITESET 的基础上多了一个约束，即在主库上同一个线程先后执行的两个事务，在备库执行的时候，要保证相同的先后顺序。

也就是，即使主库是一个线程执行的，在从库也可以通过多个线程回放。

MySQL 5.7.22的并行复制配置

主库

```
set global binlog_transaction_dependency_tracking = 'writeset';
```

从库

```
set global slave_parallel_workers=8;
stop slave sql_thread;
set global slave_parallel_type=logical_clock;
set global binlog_group_commit_sync_delay = 3000;
start slave sql_thread;
flush logs;
```

7 测试多线程复制效果前的准备

我们会通过压测工具sysbench多个线程往主库写入数据，然后看不开多线程复制和开启多线程复制，从库的延迟情况。

sysbench安装

```
curl -s
https://packagecloud.io/install/repositories/akopytov/sysbench/script.rpm.sh |
sudo bash
sudo yum -y install sysbench
```

主库创建用户和压测库

创建压测库

```
create database sysbench_db;
```

创建用户

```
CREATE USER 'repl_rw'@'%' identified WITH mysql_native_password BY 'Uda_dQc63';
GRANT REPLICATION SLAVE,REPLICATION
CLIENT,create,drop,insert,delete,update,select,index ON *.* TO 'repl_rw'@'%';
```

从库创建延迟统计脚本

提示词：

> 编写一个Shell脚本，每秒获取从库的Seconds_Behind_Master的值，
>
> 从库IP为192.168.152.30
>
> 并把结果写到主库martin库的seconds_behind_master表中，表还包括时间字段，
>
> 并在脚本之外加上建表语句
>
> 主库IP为192.168.152.70
>
> MySQL的用户名是repl_rw，密码是Uda_dQc63

生成的脚本：

```
#!/bin/bash
```

```
# 主库连接参数
master_host="192.168.152.70"
master_user="repl_rw"
master_password="Uda_dQc63"
master_database="martin"

# 从库连接参数
slave_host="192.168.152.30"
slave_user="repl_rw"
slave_password="Uda_dQc63"

# 建表语句
create_table_statement="CREATE TABLE IF NOT EXISTS
$master_database.seconds_behind_master (id INT AUTO_INCREMENT PRIMARY KEY, time
TIMESTAMP DEFAULT CURRENT_TIMESTAMP, seconds_behind_master INT);"

# 创建表
mysql -h $master_host -u $master_user -p$master_password -D $master_database -e
"$create_table_statement"

while true; do
    current_time=$(date +"%Y-%m-%d %H:%M:%S")
    seconds_behind_master=$(mysql -h $slave_host -u $slave_user -p$slave_password
-e "SHOW SLAVE STATUS\G" | awk '/Seconds_Behind_Master/ {print $2}')

    insert_statement="INSERT INTO $master_database.seconds_behind_master (time,
seconds_behind_master) VALUES ('$current_time', '$seconds_behind_master');"
    mysql -h $master_host -u $master_user -p$master_password -D $master_database
-e "$insert_statement"

    sleep 1
done
```

我们先用这种方式来查看主从延迟，当然，DBA体系课（https://class.imooc.com/sale/dba）后面实战部分，会部署监控系统，通过监控来查看延迟，那更合适。

8 配置Grafana展示从库延迟

Grafana安装

```
wget https://dl.grafana.com/enterprise/release/grafana-enterprise-10.0.2-
1.x86_64.rpm yum install -y grafana-enterprise-10.0.2-1.x86_64.rpm
```

启动grafana

```
systemctl start grafana-server
```

运行获取复制延迟的脚本

```
sh seconds_behind_master.sh
```

这样，有初始数据，可以方便我们配置Grafana的dashboard。

编辑Grafana页面

```
select time,seconds_behind_master from seconds_behind_master;
```

9 关闭并行复制进行压测

关闭并行复制

在主库执行

```
set global binlog_transaction_dependency_tracking=COMMIT_ORDER;

set global binlog_group_commit_sync_delay = 0;
```

从库临时关闭并行复制

```
stop slave sql_thread;
set global binlog_transaction_dependency_tracking=COMMIT_ORDER;
set global binlog_group_commit_sync_delay = 0;
set global slave_parallel_workers=0;
set global slave_parallel_type='database';
start slave sql_thread;
```

运行压测命令

从库运行脚本

```
sh seconds_behind_master.sh
```

主库执行压测命令（写入数据）

```
sysbench --db-driver=mysql  --mysql-host=192.168.152.70 --mysql-port=3306 --
mysql-user='repl_rw' --mysql-password='Uda_dQc63' --mysql-db=sysbench_db --
threads=10 --table_size=500000 --tables=10  --time=100  oltp_write_only prepare
```

10 开启并行复制进行压测

开启并行复制

主库

```
set global binlog_transaction_dependency_tracking = 'writeset';
set global binlog_group_commit_sync_delay = 3000;
```

从库

```
set global slave_parallel_workers=8;
stop slave sql_thread;
set global slave_parallel_type=logical_clock;
set global binlog_group_commit_sync_delay = 3000;
start slave sql_thread;
flush logs;
```

运行压测命令

主库重建压测库

```
drop database sysbench_db;
create database sysbench_db;
```

主库执行压测命令（写入数据）

```
sysbench --db-driver=mysql  --mysql-host=192.168.152.70 --mysql-port=3306 --
mysql-user='repl_rw' --mysql-password='Uda_dQc63' --mysql-db=sysbench_db --
threads=4 --table_size=500000 --tables=10  --time=100  oltp_write_only prepare
```

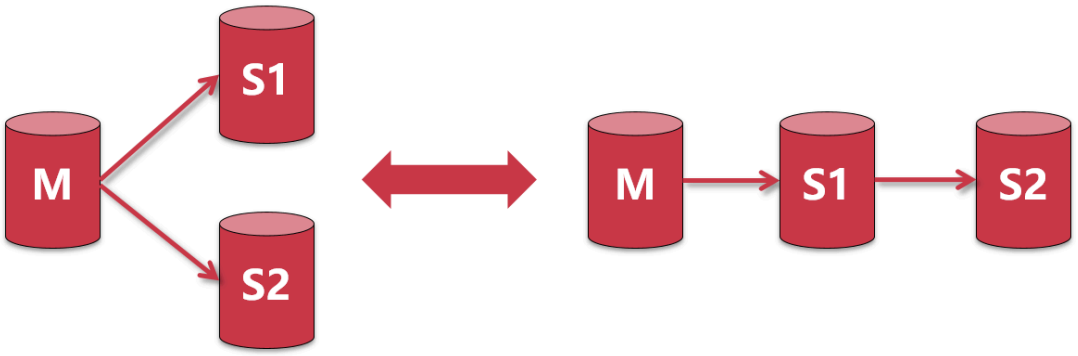
延迟对比如下：

1-5 一主两从和级联架构之间要怎么切换?

在生产环境，我们可能会有一主多从（如下图左边的这种场景）的情况，多个从库提供查询。

有时候，可能会调整架构，比如把一主两从改成主从从

主从从这种架构我们称为级联（如下图右边的这种场景）



先来简单过一下原理，这如果看不懂没事，在下方还有实验，在结合视频（https://www.imooc.com/learn/1397），应该就好理解了。

如果我们要把左边的一主两从架构切换成右边的级联架构

可以先停掉S2的复制

再停S1的复制，这时候记录S1的位点以及对应主库M的位点，

说明S1这个时候的数据，跟对应主库的那个位点的数据，是一样的。

这个时候就可以启动S1的复制了

再开启S2的同步，通过start slave UNTIL，同步到S1刚才记录的主库位点

在停掉S2的同步，

把复制直接挂到S1上，位点就是刚才S1停掉复制的时候，本身记录的位点信息。

启动复制。

如果要把右边的级联架构改成左边的一主两从架构

S1先停掉复制

再查看当前位点以及对应主库的位点信息，此时因为复制已经停掉的，所以位点信息也一直是静态的

在S2中

执行show slave status\G

查看此时对应S1的位点

如果这里看到的S1位点信息和刚才S1自己看到的位点信息一致

就说明S2已经追到S1停止复制的那个位点了

就可以停止S2的复制

再启动S1的复制

再把S2的复制接到M上

接的位点，就是刚才S1看到主库M上对应的位点信息

如果觉得上面的描述有点绕，我们来实际操作一次，应该就好理解了。

2 部署一套基于位点的一主两从架构

部署三套MySQL

略

修改配置文件

关闭GTID

```
gtid_mode=off
#enforce_gtid_consistency=on
```

同时需要开启一个参数

```
log-slave-updates = 1
```

表示从库从主库同步的数据也记录Binlog。比如级联复制

A -> B -> C

B是必须开启这个参数的，否则C获取不到数据

修改server-id

重启MySQL

```
/etc/init.d/mysql.server restart
```

创建一主两从架构

在主库创建复制用户

```
CREATE USER 'repl'@'%' identified WITH mysql_native_password BY 'Uid_dQc63';
GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
```

主库导出数据

```
cd /data/backup
xtrabackup --defaults-file=/data/mysql/conf/my.cnf -uroot -p --backup --
stream=xbstream --target-dir=./ >/data/backup/xtrabackup.xbstream
```

输入密码

将全备传到从库

备库创建恢复目录

```
mkdir /data/backup/recover
```

主库把备库传到两个从库

```
scp xtrabackup.xbstream 192.168.152.30:/data/backup/recover
```

停掉从实例

```
/etc/init.d/mysql.server stop
```

清空新实例数据目录和Binlog目录

```
rm /data/mysql/data/* -rf

rm /data/mysql/binlog/* -rf
```

把全备恢复到新的 MySQL 中

```
cd /data/backup/recover
xbstream -x < xtrabackup.xbstream
xtrabackup --prepare --target-dir=./

xtrabackup --defaults-file=/data/mysql/conf/my.cnf --copy-back --target-dir=./
```

启动 MySQL

```
chown -R mysql.mysql /data/mysql
/etc/init.d/mysql.server start
```

查看备份时的位点信息

```
cat xtrabackup_binlog_info
```

在两台从库创建复制关系

```

show slave status\G

CHANGE MASTER TO
MASTER_HOST='192.168.152.70',
MASTER_USER='repl',
MASTER_PASSWORD='Uid_dQc63',
MASTER_LOG_FILE='mysql-bin.000048',
MASTER_LOG_POS=196;

start slave;
```

查看主从状态

```
show slave status\G
```

测试同步
主库写入

```
use martin

create table repl_test(id int);
insert into repl_test select 1;
```

从库查询

```
use martin
select * from repl_test;
```

3 基于位点的复制架构切换

一主两从改成级联

断开S2的复制

```
stop slave
```

等一会

断开S1的复制

```
stop slave
```

s1执行show slave status\G

得到master_log_file　　mysql-bin.000049

和exec_master_log_pos　　1277

Exec_Master_Log_Pos，从库执行到的主库的位点

可以得到S1停止复制的时候，主库Binlog位点

s1查看当前位点

```
show master status
```

比如记录的Binlog文件和位点如下：

mysql-bin.000050

1281

s1启动复制

```
start slave;
```

S2 启动复制，同步到s1相同的位置（这个根据你自己的实际位点进行更改）

```
start slave UNTIL MASTER_LOG_FILE ="mysql-bin.000049",  MASTER_LOG_POS=1277;
```

查看复制状态

```
show slave status\G
```

只要同步到对应位置，SQL线程就会停止

S2接到S1（binlog文件和位点记得改成你自己环境的）

```
stop slave;


change master to master_host='192.168.152.30',
MASTER_USER='repl',
MASTER_PASSWORD='Uid_dQc63',
master_log_file='mysql-bin.000050',
master_log_pos=1281;

start slave;
```

查看复制状态

```
show slave status\G
```

数据同步测试

主库写入测试数据

```
use martin

insert into repl_test select 222;
```

从库查询数据

```
use martin

select * from repl_test;

show slave status\G
```

级联改成一主两从

在S1上操作

```
stop slave;
show master status\G
```

查看当前的位点信息

再执行

```
show slave status\G;
```

记录此时对应M的Binlog和位点

在S2中操作

show slave status\G;

如果Relay_Master_Log_File和Exec_Master_Log_Pos等于S1的位点

停止复制

```
stop slave;
```

再启动S1的复制

```
start slave;
```

重新配置S2的复制

```
change master to master_host='192.168.152.70',
MASTER_USER='repl',
MASTER_PASSWORD='Uid_dQc63',
master_log_file='mysql-bin.000049',
master_log_pos=1277;

start slave;
```

基于位点的复制改成基于GTID的复制

修改enforce_gtid_consistency=warn

```
SET global ENFORCE_GTID_CONSISTENCY = WARN;
```

主和从都要执行

做这一步骤的目的是让服务器正常运行一段时间，如果有不符合GITD一致性的语句，提醒我们，就可以考虑调整代码。

防止后面改成GTID模式，导致业务SQL执行失败。

修改enforce_gtid_consistency=on

```
SET global ENFORCE_GTID_CONSISTENCY = ON;
```

如果违反了gtid一致性，直接报错

修改gtid_mode=off_permissive

```
SET GLOBAL GTID_MODE = OFF_PERMISSIVE;
```

每台机器执行

```
SET GLOBAL GTID_MODE = ON_PERMISSIVE;
```

查看正在执行的匿名事务数量

```
SHOW STATUS LIKE 'ONGOING_ANONYMOUS_TRANSACTION_COUNT';
```

刷新日志

```
flush logs;
```

主从都开启GTID

```
SET GLOBAL GTID_MODE = ON;
```

参数持久化

将gtid_mode=ON和 添加enforce_gtid_consistency=ON到 my.cnf。

修改复制参数
从库执行

```
stop slave;
CHANGE MASTER TO MASTER_AUTO_POSITION = 1;
START SLAVE;
```

查看复制状态

```
show slave status\G
```

数据同步测试

主库写入测试数据

```
use martin

insert into repl_test select 666;
```

从库查询数据

```
use martin

select * from repl_test;

show slave status\G
```

GTID复制一主两从改级联

```
stop slave;

change master to
master_host='192.168.152.30',
master_user='repl',
master_password='Uid_dQc63',
MASTER_AUTO_POSITION=1;

start slave;
```

数据同步测试

主库写入测试数据

```
use martin

insert into repl_test select 666;
```

从库查询数据

```
use martin

select * from repl_test;

show slave status\G
```

这样看来，对于架构调整，还是GTID复制方便很多，所以，也建议各位同学后面在生产环境使用GTID复制。

GTID级联改一主两从

```
stop slave;

change master to
master_host='192.168.152.70',
master_user='repl',
master_password='Uid_dQc63',
MASTER_AUTO_POSITION=1;

start slave;
```

1-6 想忽略某张表的复制或者只复制某张表，应该怎么配置?

有时候，比如部分从库，并不需要查询所有的表，那我们在同步时，可以只同步需要的表或者忽略部分的表。

当然，这里要注意，如果从库是用作备用的，那就不建议忽略部分表的同步了。

1 实验环境准备

创建测试库和测试表

创建两个测试库，并创建一些表

```
create database db1;
create database db2;

use db1
create table table_01(
    id int not null auto_increment primary key,
    name varchar(10)
);

create table table_02(
    id int not null auto_increment primary key,
    name varchar(10)
);

create table log_01(
    id int not null auto_increment primary key,
    name varchar(10)
);


use db2

create table test_01(
    id int not null auto_increment primary key,
    name varchar(10)
);
```

写入测试数据

往所有的表写入数据：

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
```

查询数据

```
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

只复制某一个库

配置方法

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_DO_DB=(db1);
start slave sql_thread;
```

如果需要持久化，多个库需要配置多行

--replicate-do-db=db1

查看复制状态，

```
show slave status\G
```

可以看到Replicate_Do_DB是db1

也就是只复制db1

测试数据同步

再往所有的表写入数据：

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));

--查询4张表的数据
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

可以看到，db2的数据并没有同步

取消复制过滤

执行

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_DO_DB=();
start slave sql_thread;
-- 确定
show slave status\G
```

3 忽略某个库的复制

配置方法

可以执行

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_IGNORE_DB=(db1);
start slave sql_thread;
```

如果想持久化配置，在配置文件中加上

```
--replicate-ignore-db=db1
```

查看复制状态

```
show slave status\G
```

可以看到Replicate_Ignore_DB 后面的值是db1

测试数据同步

再到主库写入数据

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
```

从库查询数据

```
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

取消复制过滤

```
stop slave sql_thread;
CHANGE REPLICATION FILTER REPLICATE_IGNORE_DB=();
start slave sql_thread;
--
show slave status\G
```

4 只复制指定的表

配置方法

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_DO_TABLE=(db1.table_01);
start slave sql_thread;

--确定
show slave status\G
```

如果需要持久化

```
--replicate-do-table=db1.table_01
```

测试数据同步

再到主库写入数据

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
```

从库查询数据

```
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

取消复制过滤

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_DO_TABLE=();
start slave sql_thread;
--确定
show slave status\G
```

5 忽略复制指定的表

配置方法

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE=(db1.log_01);
start slave sql_thread;
--确定
show slave status\G
```

如果要持久化，在配置文件加入

```
--replicate-ignore-table=db1.log_01
```

测试数据同步

再到主库写入数据

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
```

从库查询数据

```
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

取消复制过滤

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE=();
start slave sql_thread;

--确定
show slave status\G
```

6 指定同步某些表

配置方法

支持通配符的

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db1.table%');
start slave sql_thread;

-- 确定
show slave status\G
```

如果需要持久化，配置

```
--replicate-wild-do-table=db1.table%
```

测试数据同步

再到主库写入数据

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
```

从库查询数据

```
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

可以看到，只有db1中以表名以table开头的表，才能同步

取消复制过滤

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ();
start slave sql_thread;
--确定
show slave status\G
```

7 忽略同步某些表

配置方法

执行

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_WILD_IGNORE_TABLE = ('db1.table%');
start slave sql_thread;
--确定
show slave status\G
```

如果要持久化

```
--replicate-wild-ignore-table=db1.table%
```

测试数据同步

再到主库写入数据

```
insert into db1.table_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.table_02(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db1.log_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
insert into db2.test_01(name) values (SUBSTRING(MD5(RAND()), 1, 10));
```

从库查询数据

```
select * from db1.table_01;
select * from db1.table_02;
select * from db1.log_01;
select * from db2.test_01;
```

其他的表都同步了最新的数据，就db1库中，以table开头的表，从库没有同步数据。

取消复制过滤

```
STOP SLAVE SQL_THREAD;
CHANGE REPLICATION FILTER REPLICATE_WILD_IGNORE_TABLE = ();
start slave sql_thread;
--确定
show slave status\G
```

8 总结

我们再对这一节讲的内容做一个总结，后面如果会用到复制过滤，可以参考这张图。

类型	动态配置方法	配置文件配置方法	配置取消方法
只复制某一个库	CHANGE REPLICATION FILTER REPLICATE_DO_DB=(db1);	--replicate-do-db=db1	CHANGE REPLICATION FILTER REPLICATE_DO_DB=();
忽略某个库的复制	CHANGE REPLICATION FILTER REPLICATE_IGNORE_DB=(db1);	--replicate-ignore-db=db1	CHANGE REPLICATION FILTER REPLICATE_IGNORE_DB=();
只复制指定的表	CHANGE REPLICATION FILTER REPLICATE_DO_TABLE=(db1.table_01);	--replicate-do-table=db1.table_01	CHANGE REPLICATION FILTER REPLICATE_DO_TABLE=();

类型	动态配置方法	配置文件配置方法	配置取消方法
忽略复制指定的表	CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE= (db1.log_01);	--replicate-ignore-table=db1.log_01	CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE=();
指定同步某些表	CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db1.table%');	--replicate-wild-do-table=db1.table%	CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ();
忽略同步某些表	CHANGE REPLICATION FILTER REPLICATE_WILD_IGNORE_TABLE = ('db1.table%');	--replicate-wild-ignore-table=db1.table%	CHANGE REPLICATION FILTER REPLICATE_WILD_IGNORE_TABLE = ();

1-6 如果复制延迟了该如何处理?

我们在工作中，很可能会遇到从库延迟的情况

那么，如果出现延迟，我们应该怎样处理，才能缓解延迟呢?

这一节内容我们就来聊聊主从延迟的解决办法。

1 复制延迟的原因（一）

主库增删改并发大

我们在多线程复制那一节也讲过，如果从库没开启多线程复制，而主库多个并发执行增删改操作，从库单个SQL线程解析Relay Log的内容，把数据写入从库，也就是主库并发，从库单线程，显然，也很可能导致延迟。

比如用压测工具执行并发写入：

```
create database sysbench_db;
sysbench --db-driver=mysql  --mysql-host=192.168.152.70 --mysql-port=3306 --
mysql-user='repl_rw' --mysql-password='Uda_dQc63' --mysql-db=sysbench_db --
threads=4 --table_size=500000 --tables=4  --time=100  oltp_write_only prepare
```

可以在从库看到延迟。

```
show slave status\G
```

大表在做DDL

一方面 DDL 会产生元数据锁，可能产生阻塞，另外一方面，DDL 是在主库执行完，才写入 Binlog ，因此从库是在主库执行完才开始执行的。如果是单线程复制，这期间从库其他事务需要等待。所以可能产生比较久的延迟。一般建议使用 Online DDL 工具或者在业务低峰做大表 DDL 操作

我们来构造一条会产生延迟的DDL语句

```
use sysbench_db
show tables;
alter table sbtest1 add column d char(10) after c;
```

执行的过程，从库并不会延迟，因为主库没执行完，就还没写到Binlog中

当主库执行完后，在从库就会看到延迟

这里之所以使用after c，是因为从MySQL 8.0.12开始，InnoDB存储引擎已经原生支持快速加列的功能，不过只能支持在表最后加列，我们的版本是8.0.25，在中间加列，还是不能实现快速加列，所以需要执行很久，也就导致了延迟

也可以来试试直接加在表的最后

```
alter table sbtest1 add column e char(10);
```

瞬间执行完，这也是8.0快速加列的优势。再大的表，在最后增加一个字段，根本不会导致延迟

从库备份导致延迟

我们很多时候都会在从库执行备份任务。

备份工具都需要执行 FLUSH TABLES WITH READ LOCK，这期间拷贝非事务引擎的表，如果非事务引擎的数据比较多，执行FTWRL的时间也会很长，这期间写入会被阻塞，就可能导致延迟。

大事务

当有大事务时，主库可能在这个事务执行的时候，其他事务也能并发执行。但是在从库，单线程复制的情况下，其他事务只能等这个事务执行完，才能在从库执行；

我们来模拟一个大事务

```
insert into sbtest1(k,c,pad) select k,c,pad from sbtest1;
```

执行完之后，在从库，就看到了延迟。

从库配置差

有时候，为了节约成本，会考虑使用低配的机器做为从库，

就会出现从库处理能力比主库差很多，从库回放的速度追不上主库的写入速度

也可能会导致延迟

2 怎样判断延迟

Seconds_Behind_Master

一种常规的方法就是 show slave status 查看 Seconds_Behind_Master，这个参数表示从库延迟的秒数。

如果是0，表示可能没有延迟。这里为什么是可能呢?

当从库正在主动处理更新时，此字段显示从库上的当前时间戳与从库上当前正在处理的事件的主库上记录的原始时间戳之间的差异。

当副本上当前没有处理任何事件时，该值为 0

在某些情况下，Seconds_Behind_Master 并不一定准确。比如网络中断时，Seconds_Behind_Master = 0 ，并不能代表主从无延迟。

因此，有比这个更准确的一种方法：对比位点或 GTID。

对比位点

如果是基于位点的复制，则判断 Master_Log_File 跟 Relay_Master_Log_File 是否相等，如果 Relay_Master_Log_File 落后 Master_Log_File，则表示主从存在延迟。

其中

Master_Log_File 表示 IO 线程正在读取的主库 binlog 文件名

Relay_Master_Log_File 表示SQL 线程最近执行的事务对应的主库 binlog 文件名

或者判断 Read_Master_Log_Pos 跟 Exec_Master_Log_Pos 是否相等，如果后者落后前者很多，则表示延迟比较高。

其中

Read_Master_Log_Pos 表示IO 线程正在读取的主库 binlog 文件中的位点

Exec_Master_Log_Pos 表示 SQL 线程最近读取和执行的事务对应的主库 binlog 文件中的位点

对比GTID

如果开启了 GTID 复制，则可以对比 Retrieved_Gtid_Set 和 Executed_Gtid_Set 是否相等，如果 Executed_Gtid_Set 落后很多，则表示存在延迟。

其中

Retrieved_Gtid_Set：从库收到的所有日志的 GTID 集合；

Executed_Gtid_Set：从库已经执行完的 GTID 集合。

我们可以让ChatGPT帮我们写一个延迟判断的脚本，提示词如下：

> 帮忙写一个MySQL复制延迟的判断的shell脚本
>
> 开启GTID的，可以判断show slave status结果中的 Retrieved_Gtid_Set 减去 Executed_Gtid_Set 的值来计算从库落后的事务数

> 并每秒输出一次上面计算出来的落后事务数，不要通过查询表的方式计算落后事务数
>
> 比如
>
> Retrieved_Gtid_Set: ec992f67-a08b-11ed-9a9f-024255bd70b6:4429853-4431720
>
> Executed_Gtid_Set: ec992f67-a08b-11ed-9a9f-024255bd70b6:1-4431655
>
> 就需要计算4431720-4431655
>
> 并提供脚本里MySQL用户的创建命令，用户名用delay_check，密码用Yd_asdfa15

生成的脚本如下：

```
#!/bin/bash

# MySQL连接信息
MYSQL_HOST="localhost"
MYSQL_PORT="3306"
MYSQL_USER="delay_check"
MYSQL_PASSWORD="Yd_asdfa15"

# 获取落后的事务数
get_delayed_transaction_count() {
  RETRIEVED_GTID_SET=$(mysql -h $MYSQL_HOST -P $MYSQL_PORT -u $MYSQL_USER -
p$MYSQL_PASSWORD -e "SHOW SLAVE STATUS\G" | awk '/Retrieved_Gtid_Set/{print $2}')
  EXECUTED_GTID_SET=$(mysql -h $MYSQL_HOST -P $MYSQL_PORT -u $MYSQL_USER -
p$MYSQL_PASSWORD -e "SHOW SLAVE STATUS\G" | awk '/Executed_Gtid_Set/{print $2}')

  RETRIEVED_GTID_SET_COUNT=$(echo $RETRIEVED_GTID_SET | cut -d':' -f2 | cut -d'-'
-f2)
  EXECUTED_GTID_SET_COUNT=$(echo $EXECUTED_GTID_SET | cut -d':' -f2 | cut -d'-' -
f2)

  let "DELAYED_TRANSACTION_COUNT = RETRIEVED_GTID_SET_COUNT -
EXECUTED_GTID_SET_COUNT"
}

# 输出落后的事务数
output_delayed_transaction_count() {
  echo "Delayed Transaction Count: $DELAYED_TRANSACTION_COUNT"
}

while true; do
  # 获取延迟的事务数
  get_delayed_transaction_count

  # 输出落后的事务数
  output_delayed_transaction_count

  sleep 1
done
```

从库运行脚本

主库用压测工具执行并发写入：

```
drop database sysbench_db;
create database sysbench_db;
```

压测

```
sysbench --db-driver=mysql  --mysql-host=192.168.152.70 --mysql-port=3306 --
mysql-user='repl_rw' --mysql-password='Uda_dQc63' --mysql-db=sysbench_db --
threads=4 --table_size=500000 --tables=4  --time=100  oltp_write_only prepare
```

可以在从库看到延迟。

3 主从延迟处理方法（一）

开启多线程

在多线程那一节，我们也进行了实验，开启多线程复制对比不开启多线程复制，延迟是底很多的。

所以最推荐的解决延迟的方法就是开启多线程复制。

调整一些参数

比如innodb_flush_log_at_trx_commit 和 sync_binlog

这里我们来讲一下两个参数的作用。

innodb_flush_log_at_trx_commit：控制重做日志刷新到磁盘的策略，有 0 、1 和 2 三种值。

0：master线程每秒把 redo log buffer 写到操作系统缓存，再刷到磁盘；

1：每次提交事务都将 redo log buffer 写到操作系统缓存，再刷到磁盘；

2：每次事务提交都将 redo log buffer 写到操作系统缓存，由操作系统来管理刷盘。

sync_binlog：控制 binlog 的刷盘时机，可配置 0、1 或者大于 1 的数字。

0：二进制日志从不同步到磁盘，依赖OS刷盘机制；

1：二进制日志每组事务提交都会刷盘；

n(n>1) : 每n组事务提交落盘一次。

所以当从库有延迟的时候，可以把从库的 innodb_flush_log_at_trx_commit 设置为 0，也就是 master 线程每秒把 redo log buffer 写到操作系统缓存，再刷到磁盘，而不是每个事务都刷一次，这可以大大缓解从库的压力。

另外就是可以把 sync_binlog 调整为大于 100 的数字，表示 100 多次事务，才会做一次二进制日志的落盘。

调整从库机器配置

刚才我们也讲到了，从库配置差，可能也会导致延迟，如果经常出现延迟情况，并且从库需要提供查询的，就可以考虑提高从库的配置。

3 主从延迟处理方法（二）

避免大事务

如果延迟是大事务导致的，可以在源头上做规范，避免大事务。

使用PT工具执行耗时长的DDL

pt-online-schema-change，可以实现在线修改表结构，不锁表，比如mysql 5.6，DDL，就强烈建议使用，工具的具体用法，我们会在后面PT工具一章详细介绍。

调整架构

另外，我们有时也会遇到这种场景，比如某张表比较大，延迟经常是这张表导致的，从库上读取数据时又用不上这张表，就可以考虑把大表单独创建一个从库进行复制。然后在原来的从库忽略这张表的复制，业务查询原来的从库就基本没延迟了。

1-7 详解复制常见的问题及其处理办法

我们在工作中，多多少少会遇到复制报错的情况，这一节内容，就来跟大家分享一下，复制常见的报错以及处理方法。

1 复制常见问题及处理办法（一）

server_id重复

我们在配置主从的时候，假如忘记修改server_id，导致主从都一样，就会报错

我们来模拟这个复制问题

进到从库，执行

```
show slave status\G
```

然后把这个从库的server_id改成跟主库一样，来模拟server_id重复的情况

注意，这个实验只能在自己的测试环境操作，千万别在生产环境尝试。

```
set global server_id=15270;
```

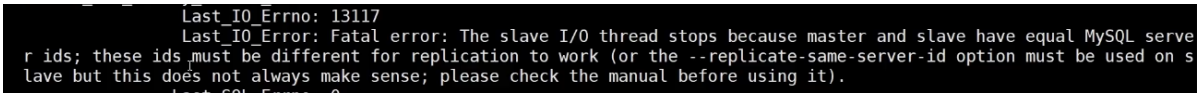
再来重启复制

```
stop slave;
start slave;
```

再来查看复制状态

```
show slave status\G
```

可以看到复制报错了：



解决办法

把从库的server-id改成和主库不一样，一般建议是IP后两位组合成的纯数字。

```
set global server_id=15230;
```

再重启复制

```
stop slave;

start slave;
```

再来查复制状态

```
show slave status\G
```

可以看到，复制就恢复了

端口不通

先来模拟端口不通的场景

首先在主库机器上，增加防火墙策略，不允许从库访问主库的3306端口：

```
iptables -A INPUT -s 192.168.152.30 -p tcp --dport 3306 -j DROP
```

从库再telnet主库端口，

```
telnet  192.168.152.70 3306
```

发现不通了

重启复制

```
stop slave;start slave;
```

查看复制状态

```
show slave status\G
```

可以看到Slave_IO_Running的值已经变成了Connecting，并且Seconds_Behind_Master变成NULL了。

解决办法：

将主从端口调通，保证从库能 telnet 通对方的 3306 端口。

先把刚才加的防火墙策略给删掉，

执行

```
iptables -L -n --line-numbers
```

再删除刚才加的规则

```
iptables -D INPUT xx
```

再给从库开放3306端口

从库再次telnet主库的3306端口

```
telnet  192.168.152.70 3306
```

如果能通，就再重启复制

```
stop slave;start slave;
```

从库磁盘空间满了

这种情况，在自己测试环境出现过几次，生产环境，因为有磁盘使用率的监控，所以基本不会存在这个问题。

我们也来模拟一下

在从库创建一个10G的临时大文件

```
dd if=/dev/zero of=fill bs=1M count=10000
```

主库再写入数据

通过询问ChatGPT，

> 创建一个MySQL存储过程，往一张测试表写入1万行数据，并提供建表语句

再到主库MySQL中执行

从库执行

```
show slave status\G
```

可以看到，延迟一直在增加

这时候，执行stop slave ，可能会一直等待

解决办法，

释放一些磁盘空间

比如我们这个例子，把fill文件删除就行

再新窗口登录MySQL，执行

```
show slave status\G
```

就可以看到复制恢复了。

2 复制常见问题及处理办法（二）

主库要新增的内容在从库已经有了

来模拟这种报错

在从库新建一个库

```
create database martin_test;
```

再到主库执行同样的建库语句：

```
create database martin_test;
```

再去从库执行

```
show slave status\G
```

Last_Error 中就会提示，不能再创建 martin_test 库，因为已经存在了。

解决办法：

从库跳过这条记录。

如果是 GTID 模式：

假如 Executed_Gtid_set 为：

```
65c07f5e-1e37-11ed-b657-fa163e0d61f4:1-10139
```

则恢复操作为：

```
stop slave;
set @@session.gtid_next='65c07f5e-1e37-11ed-b657-fa163e0d61f4:10140';
```

session.gtid_next 后面接的是 Executed_Gtid_set 下一个值。

使用空事务代替报错的事务

```
begin;commit;
```

再设置使用自动生成的GTID

```
set session gtid_next='AUTOMATIC';
```

启动复制

```
start slave;
```

查看复制状态

```
show  slave status\G
```

如果是位点模式

```
stop slave;
set GLOBAL SQL_SLAVE_SKIP_COUNTER=1;
start slave;
show slave status\G
```

主库要更新的记录从库没有

可以先模拟出这种报错

创建测试表并写入数据：

```
use martin_test
CREATE TABLE test_repl (
id int NOT NULL AUTO_INCREMENT,
a int NOT NULL,
PRIMARY KEY (id)
) ENGINE=InnoDB CHARSET=utf8mb4;
insert into test_repl values (1,1),(2,2);
```

在从库删除一条记录

```
use martin_test
delete from test_repl where id=1;
```

再到主库更新 id =1 的记录

```
update test_repl set a=2 where id=1;
```

从库执行 show slave status\G
提示要更新的记录不存在。

解决办法：
我们可以根据上面的 Relay_Log_File 和 Relay_Log_Pos来解析出报错的 SQL 语句。

```
mysqlbinlog  mysql-relay-bin.000014 --start-position=1112 --base64-output=decode-
rows -v >/tmp/1110.sql
```

可以找到原记录，原记录就是 WHERE 后面对应的值，然后在MySQL中执行

```
insert into test_repl select 1,1;
stop slave;start slave;
```

这样主从同步就恢复了。

找不到主库的Binlog位点

从库执行

```
stop slave;
```

主库刷新日志

```
flush logs;
```

在主库随便做一个变更

```
create database a1;
```

再刷新日志

```
flush logs;
```

查看所有的Binlog文件

```
show binary logs;
```

再清空之前的日志文件

```
purge binary logs to 'mysql-bin.000062';
```

从库启动复制

```
start slave;
```

查看复制状态，会有如下报错：

```
               Last_IO_Errno: 13114
               Last_IO_Error: Got fatal error 1236 from master when reading data from binary log: 'Cannot replicate b
ecause the master purged required binary logs. Replicate the missing transactions from elsewhere, or provision a new s
lave from backup. Consider increasing the master's binary log expiration period. The GTID set sent by the slave is '08
64fad8-2716-11ee-8416-000c299e5214:1-6,
3e58c925-b396-11ed-9d79-000c2965ac6b:1-14524531', and the missing transactions are '3e58c925-b396-11ed-9d79-000c2965ac
6b:14524532''
```

大概意思就是：

从二进制日志读取数据时，从主服务器得到致命错误1236无法复制，因为主服务器清除了所需的二进制日志。可以从其他地方复制丢失的事务，或者从备份中提供一个新的从服务器。考虑增加主机二进制日志的过期时间
还告诉了我们缺失事务的gtid范围。

解决办法

1 如果有其他正常的从库，可以把报错的从库接到这个从库，复制缺失的事务

```
stop slave;

change master to
master_host='192.168.152.30',
master_user='repl',
master_password='Uid_dQc63',
MASTER_AUTO_POSITION=1;

start slave;
```

2 如果没有其他从库，那建议是重建复制。

1-8 一步一步带你了解复制的演进历程

在很多场景下，MySQL 的高可用都是借助主从复制实现的，而 MySQL 复制不断的演进，也使得她越来越受欢迎。这一节内容就来聊聊 MySQL 复制的演进历程。

1 三种日志格式对复制的影响

1.1 开始支持复制时的statement格式

MySQL 从 3.23 版本开始支持复制，但是在 5.1.5 之前只支持 statement 格式的复制，尽管这种模式下，binlog 日志量相对比较少，但是涉及到跨库更新、或者使用结果不确定的函数时，比如 UUID()，容易出现主从数据不一致的情况。

我们可以来做个实验

主从设置日志格式为statement

```
stop slave;
set global binlog_format=statement;
start slave;
```

创建一张测试表

```
use martin
create table statement_t1(
    id int not null auto_increment primary key,
    a varchar(100)
);
insert into statement_t1(a) values (uuid());
```

主从库对比数据

```
select * from statement_t1;
```

就出现主从数据不一样的情况了。

实验之后，我们需要把Binlog格式改回成Row格式

```
stop slave;
set global binlog_format=row;
start slave;
```

1.2 开始支持Row格式的复制

从 MySQL 5.1.5 开始，新增了 Row 格式，日志中会记录每一行数据被修改的形式，因此 Row 格式下的复制，主从之间的数据一致性得到了保障，但是缺点是 binlog 日志量相对 statement 较多。

1.3 新增mixed格式

从 MySQL 5.1.8 开始新增 mixed 格式，当可能造成主从数据不一致的 SQL 时，binlog 使用 row 格式记录，否则使用 statement 格式记录。显然这种格式下，日志量是介于 statement 和 row 格式之间的。

2 半同步复制的演进

2.1 异步复制

传统的 MySQL 复制为异步复制，其原理如下：

在主库开启 binlog 的情况下；

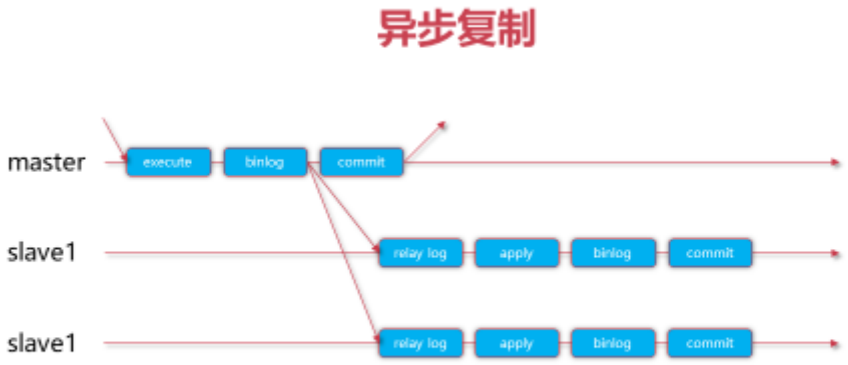
如果主库有变更操作，会记录到 binlog 中；

主库通过 IO 线程把 binlog 里面的内容传给从库的中继日志（relay log）中；

主库给客户端返回 commit 成功（这里不管从库是否已经收到了事务的 binlog）；

从库的 SQL 线程负责读取它的 relay log 里的信息并应用到从库数据库中。

其过程如下图：



在异步复制下，假如配置了自动切换的前提下，主库突然宕机，然后从提升为主时，原来主库上可能有一部分已经完成提交的数据还没来得及发送到从库，就可能产生数据丢失。为了解决这个问题，在 MySQL 5.5 版本中引入了半同步复制。下面来看下半同步复制的原理。

2.2 半同步复制

半同步复制原理如下：

在主库开启 binlog 的情况下；

如果主库有增删改的语句，会记录到 binlog 中；

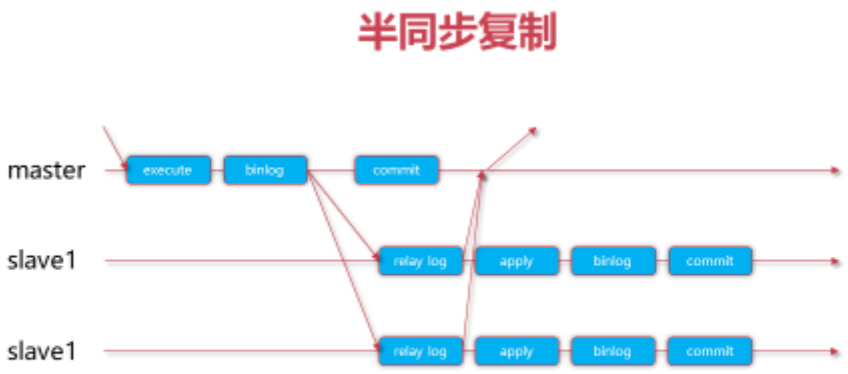
主库通过 IO 线程把 binlog 里面的内容传给从库的中继日志（relay log）中；

从库收到 binlog 后，发送给主库一个 ACK，表示收到了；

主库收到这个 ACK 以后，才能给客户端返回 commit 成功；

从库的 SQL 线程负责读取它的 relay log 里的信息并应用到从库数据库中。

其过程如下图：



跟传统异步复制相比，半同步复制保证了所有客户端发送过确认提交的事务，从库都已经收到这个日志了。

我们来看下配置方法

在主库

执行命令安装：

```
install plugin rpl_semi_sync_master soname 'semisync_master.so';
```

安装完成之后，再来查看系统库MySQL的plugin表

```
select * from mysql.plugin;
```

再来开启半同步复制

```
set global rpl_semi_sync_master_enabled=1;
```

再到从库
执行命令安装：

```
install plugin rpl_semi_sync_slave soname 'semisync_slave.so';
```

再来开启半同步复制

```
set global rpl_semi_sync_slave_enabled=1;
```

为了保证重启后继续生效。需要修改配置文件

修改主库的配置文件，增加

```
plugin_load="rpl_semi_sync_master=semisync_master.so;rpl_semi_sync_slave=semisync
_slave.so"
rpl_semi_sync_master_enabled = 1
rpl_semi_sync_slave_enabled = 1
```

如果这套MySQL还没上线，建议修改完配置文件之后重启MySQL

防止配置错误，导致下一次重启MySQL时启动异常。

```
/etc/init.d/mysql.server restart
```

如果已经有业务再使用了，那可以不重启MySQL。

再修改从库的配置文件，增加

```
plugin_load="rpl_semi_sync_master=semisync_master.so;rpl_semi_sync_slave=semisync
_slave.so"
rpl_semi_sync_master_enabled = 1
rpl_semi_sync_slave_enabled = 1
```

然后需要在从库重启IO线程

```
stop slave io_thread;
start slave io_thread;
```

查看半同步复制状态

```
show global status like '%semi%';
```

Rpl_semi_sync_master_clients	当前处于半同步状态的Slave节点的个数
Rpl_semi_sync_master_net_avg_wait_time	主库等待从库回复的平均时间，单位是微妙
Rpl_semi_sync_master_net_wait_time	主库等待从库回复的总时间
Rpl_semi_sync_master_net_waits	主库等待从库回复的总次数
Rpl_semi_sync_master_no_times	主库关闭半同步复制的次数
Rpl_semi_sync_master_no_tx	主库没有收到从库的回复而提交的次数，也就是半同步复制没有成功提交的次数

Rpl_semi_sync_master_status 表示主节点是半同步复制还是异步复制，如果是on，表示半同步复制，如果是off，表示异步复制，可能是插件没启用或者半同步复制因此提交确认超时而退回到异步复制。

这里补充一点，就是开启半同步复制的情况，主库等待从库返回确认信息的过程中，如果一直等待，直到超过rpl_semi_sync_master_timeout配置的时间，半同步就会退步成异步复制。

如果slave节点追上了Master节点，那么Master节点又会重新转成半同步复制。

Rpl_semi_sync_master_timefunc_failures 主库调用时间函数失败的次数

Rpl_semi_sync_master_tx_avg_wait_time 主库花费在每个事务上的平均等待时间

Rpl_semi_sync_master_tx_wait_time 主库等待事务的总时间

Rpl_semi_sync_master_tx_waits 主库等待事务的次数。

Rpl_semi_sync_master_wait_pos_backtraverse 当事务开始等待答复的顺序与二进制日志事件的写入顺序不同的次数

Rpl_semi_sync_master_wait_sessions 等待从库回复的会话数

Rpl_semi_sync_master_yes_tx 从库成功确认的提交数

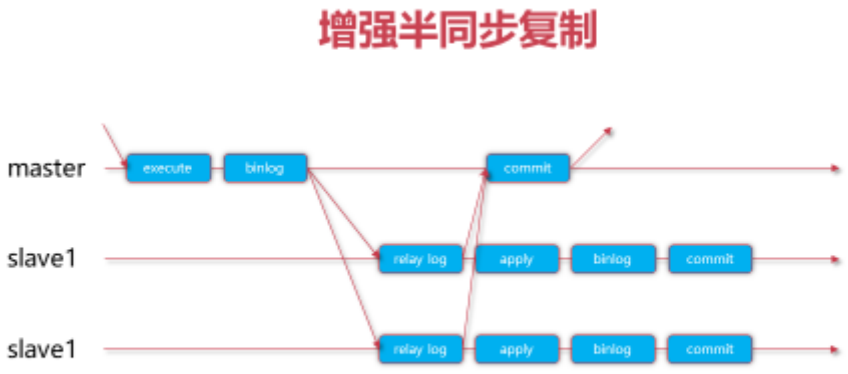
Rpl_semi_sync_slave_status 从库是否运行半同步复制

但是这种模式下，实际上主库已经将该事务 commit 到事务引擎层，只是在等待返回而已，而此时其他 session 已经可以看到数据发生了变化，如果此时主库宕机，有可能从库还没写 Relay log，就会发生其他 session 切换前后查询的数据不一致的情况。

因此，从 MySQL 5.7 开始，引入了增强半同步复制（无损复制）。

2.3 增强半同步复制

增强半同步复制（也叫无损复制）是在半同步复制基础上做了微调，即主库写数据到 binlog 后，就开始等待从库的应答 ACK，直到至少一个从库写入 Relay log 并进行数据落盘后，才返回给主库消息，通知主库可以执行 commit 操作了，然后主库开始提交到事务引擎层。



由参数rpl_semi_sync_master_wait_point控制

默认值是AFTER_SYNC，表示采用前面所讲的半同步复制策略

主库将每个事务写入Binlog和从库，并将Binlog同步到磁盘，同步后，主库等待事务接收的副本确认，收到确认后，主库将事务提交到存储引擎并将结果返回给客户端，然后客户端可以继续。

如果设置成 AFTER_COMMIT

表示

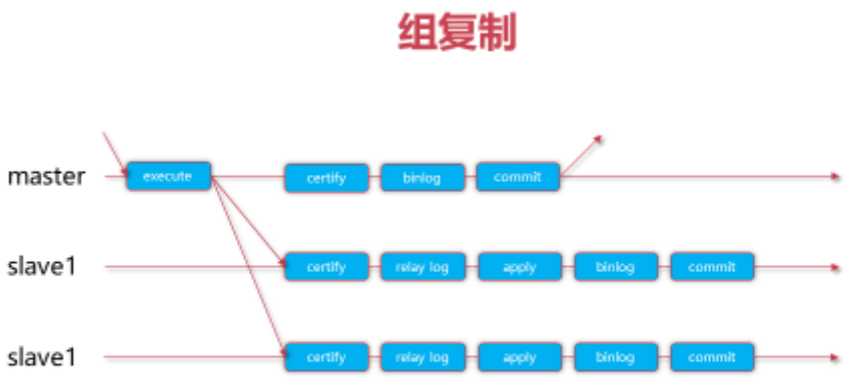
主库将每个事务写入其二进制日志和从库，同步二进制日志，并将事务提交到存储引擎。提交后，主库等待事务接收的从库确认。收到确认后，主库将结果返回给客户端，然后客户端可以继续。

但是，增强半同步复制其实也是存在问题的，假设有一个事务写 binlog 后 crash 了，事务还没有发送给从库，这时从库提升为主库，对客户端来说，数据没问题的，因为主库还没给客户端返回 commit；但是当老的主库恢复后，由于这个事务的 binlog 已经写入磁盘了，因此没办法回滚，在 crash recover 的机制下，会把这些事务重新提交，这就导致老的主库比新的主库多事务的情况。

因此又出现了一种复制形式--组复制。

3 组复制

MySQL 5.7 推出了组复制（MySQL Group Replication，简称：MGR），是基于内置的主从复制的架构实现的，主要在事务提交的过程中，嵌入单独的 binlog 封装逻辑，并通过专门的复制通道进行数据传输。



Group Replication 复制插件使用 Paxos 协议的原子广播特性，来保证在集群内的大多数节点都能接收到数据包，当数据节点接收到 write set 之后，每个节点上按照相同的规则对事务进行排序，并进行冲突检测。

对于 master，当冲突检测通过之后，数据变更写入自身的 binlog 中，然后进行存储引擎层的提交（如果发现事务冲突，则进行事务回滚）；

对于 slave，当冲突检测通过之后，就把主库发来的 binlog 写入自身的 relay log 中，然后 sql 线程读取 relay log 进行重放，并把重放的 binlog 日志写入自身的 binlog 中，然后存储引擎内部进行提交（如果发现事务冲突，则丢弃主库发送过来的 binlog 日志）。

4 并行复制

并行复制在第4节有详细讲解，这里就过一遍。

4.1 MySQL 5.6的并行复制

在传统的复制模式下，我们也许经常会遇到主从延迟的场景。这是因为在 MySQL 5.6 之前，MySQL 只支持单线程复制。

从 MySQL 5.6 版本开始，支持并行复制策略，但是只支持库级别的。如果表都集中在一个 DB 里，或者热点表集中在一个库中，那就没有什么效果了。

4.2 MySQL 5.7的并行复制

由参数：slave-parallel-type 控制并行复制策略。

配置为 DATABASE，表示使用 MySQL 5.6 版本的按库并行策略；

配置为 LOGICAL_CLOCK，同时处于 prepare 状态的事务，在备库执行时是可以并行的；处于 prepare 状态的事务，与处于 commit 状态的事务之间，在备库执行时也是可以并行的。

4.3 MySQL 5.7.22的并行复制

MySQL 5.7.22 版本里，MySQL 增加了一个新的并行复制策略，基于 WRITESET 的并行复制。

相应地，新增了一个参数 binlog-transaction-dependency-tracking，用来控制是否启用这个新策略。这个参数的可选值有以下三种。

COMMIT_ORDER，表示的就是前面介绍的，根据同时进入 prepare 和 commit 来判断是否可以并行的策略。

WRITESET，表示的是对于事务涉及更新的每一行，计算出这一行的 hash 值，组成集合 writeset。如果两个事务没有操作相同的行，也就是说它们的 writeset 没有交集，就可以并行。

WRITESET_SESSION，是在 WRITESET 的基础上多了一个约束，即在主库上同一个线程先后执行的两个事务，在备库执行的时候，要保证相同的先后顺序。

5 基于GTID的复制

GTID的出现

通过 sql_slave_skip_counter 跳过事务和通过 slave_skip_errors 忽略错误的方法，虽然都最终可以建立从库 B 和新主库 A'的主备关系，但这两种操作都很复杂，而且容易出错。MySQL 5.6 版本引入了 GTID，解决了这个问题。并且 GTID 复制模式，让我们配置主从更加简单。基于 GTID 的复制配置及维护我们在前面也详细讲过了，这里就不再讲了。

GTID的优势

可以知道事务在哪个实例上提交的

比如主从，或者其他高可用架构，比如MGR，集群通过GTID来标记事务，用来跟踪每个实例上提交的事务。

比较方便进行复制结构上的故障转移，不需要再关注位点，我们在一主两从和级联切换的那一节也讲了，GTID复制直接接到新的主上就行。

很方便判断主从一致性，因为复制完全基于事务的

1-9 复制常用参数到底有哪些?

这一节，来总结一下MySQL主从复制需要重点关注哪些参数，以及这些参数的作用。

1 复制相关的日志参数

log_bin

主从复制，主库必须开启Binlog，因为是通过把主库Binlog传到从库的relay log，再通过SQL线程把relay log中的内容回放到从库，来完成数据同步的。

binlog_format

这个参数是设置Binlog日志格式的。

这个参数对复制影响还是比较大的：

如果设置成statement，使用一些不确定的函数，可能导致主从数据不一致。

对数据一致性要求较高的场景，建议把binlog_format设置成row。

log_slave_updates

表示从库从主库同步的数据是否在从库也记录Binlog。

比如级联复制：

A -> B -> C

B是必须开启这个参数的，否则C获取不到数据。

2 GTID相关参数

gtid_mode

有四个值：

off：新的事物和复制的事物，都是没有GTID值的，没有GTID的事务，可以称为匿名事务

off_permissive：新的事物是匿名事务，复制中的事务可以是匿名事务，也可以是GTID事务

on_permissive：新的事务是GITD事务，复制中的事务可以是匿名事务，也可以是GTID事务

on：新事务和复制事务都是GTID事务。

并且只能是相邻的两个参数才能修改，比如要把gtid_mode从off改为on需要先改为off_permissive，再改为on_permissive，最后再改成on。

enforce_gtid_consistency

有三个值：

OFF:所有事务都允许违反GTID一致性

ON:不允许事务违反GTID一致性

WARN:所有事务都允许违反GTID一致性，但是会给出警告，并且会打印在错误日志里。

当我们需要在线开启GTID时，建议先把这个参数调整为WARN，观察一段时间，看是否有业务执行了违反GTID一致性的SQL，如果有，就调整代码，不然直接设置为ON，可能会导致业务中断。

3 主从差异配置

server_id

在同一个复制拓扑中，server_id一定不能重复。

否则就会报server_id重复的错。

read_only

通常主从架构中，从库建议设置成只读，防止在从库做了变更，没同步到主库，导致主从数据不一致的场景。

4 双主相关配置

auto_increment_increment

这个参数可以使用在双主场景，控制自增主键的间隔。

auto_increment_offset

控制自增列的初始值。

来测试一下双主的这两个配置。

我们先来构造一个双主的环境

在主库，直接配置成从库的副本

```
stop slave;

change master to
master_host='192.168.152.30',
master_user='repl',
master_password='Uid_dQc63',
MASTER_AUTO_POSITION=1;

start slave;
```

再来查看复制状态，可以看到，现在是双向复制的状态了。

首先申明，不到万不得已，不建议生产环境配置成双主，其实问题很多的。

比如自增主键冲突

所以如果使用了双主双写，就需要设置

auto_increment_increment和auto_increment_offset

比如主库从1开始自增，自增间隔是2，后面新增的主键值类似1、3、5、7

而从库从2开始自增，自增间隔也是2，后面新增的主键值类似2、4、6、8

通过这种方法，可以避免双写的时候，主键冲突的情况。

在主库

```
set global auto_increment_increment=2;
set global auto_increment_offset=1;
```

在从库，其实现在算另外一个主库了

```
set global auto_increment_increment=2;
set global auto_increment_offset=2;
```

在第一台主库上创建一张测试表

```
use martin
create table auto_increment_t1(
    id int not null auto_increment primary key,
    a varchar(10)
);
```

退出MySQL，重新登录，写入两行数据

```
insert into auto_increment_t1(a) values ('aaa');
insert into auto_increment_t1(a) values ('bbb');
```

再到从库，退出MySQL客户端，重新登录MySQL

写入两行数据

```
insert into auto_increment_t1(a) values ('ccc');
insert into auto_increment_t1(a) values ('ddd');
```

查询数据，

select * from auto_increment_t1;

可以看到，从库写入的两行数据，自增ID分别是4和6

5 复制过滤

replicate_do_db

只同步某个库。

replicate_ignore_db

忽略同步某个库。

replicate_do_table

只同步某些表。

replicate_ignore_table

忽略同步某些表。

replicate_wild_do_table

指定同步某些表，支持正则。

replicate_wild_ignore_table

忽略同步某一些表，支持正则。

binlog_do_db

只记录指定数据库的Binlog，不过不建议使用，因为Binlog不单单用于复制，还能用于数据恢复。

即使只想复制指定库，也建议使用replicate_do_db。

binlog_ignore_db

不记录指定数据库的Binlog，不建议使用。

6 多线程复制相关配置

slave_parallel_workers

控制复制的线程数。

slave_parallel_type

控制并行复制的策略。

如果是database，表示使用的是MySQL 5.6的按库并行策略；

如果是lagical_clock，表示采用的是5.7新的并行复制逻辑。

binlog_transaction_dependency_tracking

5.7.22新增的参数，可配置成3个值。

COMMIT_ORDER，根据同时进入 prepare 和 commit 来判断是否可以并行的策略。

WRITESET，表示的是对于事务涉及更新的每一行，计算出这一行的哈希值，组成集合 writeset。如果两个事务没有操作相同的行，也就是说它们的 writeset 没有交集，就可以并行。

> **关于哈希值**
>
> 哈希值使用唯一索引+库名+表名等信息，调用一个哈希函数得到的。

> 如果表中有多个唯一索引，那每个唯一索引都有一个哈希值。

WRITESET_SESSION，是在 WRITESET 的基础上多了一个约束，即在主库上同一个线程先后执行的两个事务，在备库执行的时候，要保证相同的先后顺序。

7 半同步复制相关配置

rpl_semi_sync_master_enabled

主库开启半同步复制

rpl_semi_sync_slave_enabled

从库开启半同步复制

rpl_semi_sync_master_wait_point

增强半同步复制

增强半同步复制（也叫无损复制）是在半同步复制基础上做了微调，即主库写数据到 binlog 后，就开始等待从库的应答 ACK，直到至少一个从库写入 Relay log 并进行数据落盘后，才返回给主库消息，通知主库可以执行 commit 操作了，然后主库开始提交到事务引擎层。

由参数rpl_semi_sync_master_wait_point控制。

默认值是AFTER_SYNC。

表示采用前面所讲的半同步复制策略。主库将每个事务写入Binlog和从库，并将Binlog同步到磁盘，同步后，主库等待事务接收的副本确认，收到确认后，主库将事务提交到存储引擎并将结果返回给客户端，然后客户端可以继续。

如果设置成AFTER_COMMIT

表示主库将每个事务写入其二进制日志和从库，同步二进制日志，并将事务提交到存储引擎。提交后，主库等待事务接收的从库确认。收到确认后，主库将结果返回给客户端，然后客户端可以继续。

好的，和复制相关的一些参数就聊到这里。

1-10 通过复制来恢复误删的库

我DBA体系课（https://class.imooc.com/sale/dba）备份和恢复那一章，讲到了误删除数据的两种恢复方式，一种是全备加Binlog，另外一种是使用my2sql工具解析Binlog中误操作的回滚语句，在写入到主库。

这一节内容，我们来聊一下，基于复制的数据恢复方法，过程是误操作后，把上一次全备导入到新的MySQL，在把这个新的MySQL配置成误操作数据库的从库。

然后同步到误操作的前一个事务，这样，从库的数据就是误操作前一刻的数据了，比如我们误删除了某个库，就把这个库的数据导回到原来的数据库，就能完成恢复。

1 准备阶段

新建测试库表并写入数据

```
te database recover;
use recover;

CREATE TABLE test_recover (
id int NOT NULL AUTO_INCREMENT,
a int NOT NULL,
PRIMARY KEY (id)
) ENGINE=InnoDB CHARSET=utf8mb4;

insert into test_recover values (1,1),(2,2);
```

创建备份用户

```
CREATE USER `u_xtrabackup`@`localhost` IDENTIFIED WITH MYSQL_NATIVE_PASSWORD BY
'Ijnbgt@123';
GRANT SELECT, RELOAD, PROCESS, SUPER, LOCK TABLES,BACKUP_ADMIN ON *.* TO
`u_xtrabackup`@`localhost`;
```

在源实例进行全量备份

```
cd /data/backup
xtrabackup --defaults-file=/data/mysql/conf/my.cnf -uu_xtrabackup -p'Ijnbgt@123'
--backup --stream=xbstream --target-dir=./ >/data/backup/xtrabackup.xbstream
```

2 写入增量并模拟误操作

模拟增量数据写入

```
use recover
insert into test_recover values (3,3);
```

模拟误操作删库

```
drop database recover;
```

3 为误操作的MySQL配置一套从库

在另外的机器R上准备一个新的MySQL实例，跟误操作的MySQL版本一致（平时建议是在每个机房为每一个版本准备一个临时用于恢复的MySQL实例）。

把全备传到机器R上。

```
scp xtrabackup.xbstream 192.168.12.162:/data/backup/recover
```

关闭R上的MySQL实例

清空R上MySQL实例的数据目录和Binlog目录

```
rm /data/mysql/data/* -rf
rm /data/mysql/binlog/* -rf
```

并把全备恢复到R上的MySQL中

```
cd /data/backup/recover
xbstream -x < xtrabackup.xbstream
xtrabackup --prepare --target-dir=./
xtrabackup --defaults-file=/data/mysql/conf/my.cnf --copy-back --target-dir=./
chown -R mysql.mysql /data/mysql
```

启动R上的MySQL。

将R上的MySQL配置成原实例的从库，但不要开启复制

```
stop slave;

reset slave;

change master to
master_host='192.168.12.161',
master_user='repl',
master_password='Uid_dQc63',
master_auto_position = 1;
```

这里一定要注意：不要启动复制！

因为启动复制，会同步主库的所有操作，也包括误操作，这样，这个从库的数据，还是没有误删除库的数据。

4 确定误操作事务的GTID

找到回档时间点对应的 Binlog 文件

通过下面命令确定误操作事务的GTID

```
cd /data/mysql/binlog
cp mysql-bin.000065 /data/backup/
cd /data/backup/
mysqlbinlog mysql-bin.000065 --start-datetime='2023-07-31 22:00:00'  --stop-
datetime='2023-07-31 22:50:00'  --base64-output=decode-rows -v
>/data/backup/1.sql
```

再来查看/data/backup/1.sql里的内容：

```
......
SET @@SESSION.GTID_NEXT= '3e58c925-b396-11ed-9d79-000c2965ac6b:14524559'/*!*/;
......
use `martin`/*!*/;
......
DROP TABLE `recover` /* generated by server */
......
```

说明误操作事务的GTID为：3e58c925-b396-11ed-9d79-000c2965ac6b:14524559。

5 R上的MySQL同步到误操作前一个事务

先启动IO线程

```
start slave io_thread;
```

再启动SQL线程到误操作前一个事务

```
start slave sql_thread  until sql_before_gtids='3e58c925-b396-11ed-9d79-
000c2965ac6b:14524559';
```

再来查看复制状态

```
show slave status\G
```

如果IO线程是Yes，SQL线程是No。就表示复制已经同步到误操作前一个事务了。

当然，需要我们确定一下，当前数据是否为误操作之前那个时间点的数据

```
select * from recover.test_recover;
```

这个也可以找业务一起来确定一下。

再清空复制关系

```
stop slave;
reset slave;
```

6 数据恢复

备份R机器MySQL的数据，再导入原实例。操作如下：

```
mysqldump -u'root' -p --set-gtid-purged=off -B recover >recover.sql
scp recover.sql 192.168.12.161:/data/backup/
```

再到原来的实例，确定recover库是没有的，导入误删除的库

```
mysql -uroot -p <recover.sql
```

确定数据是否恢复，可以找某张表进行确定，比如：

```
select * from recover.test_recover;
```

这个例子中，如果查询的数据有3行，说明恢复成功。

到这里，整个恢复过程就完成了。

1-11 通过延迟从库来恢复数据

在前面，我们讲了，通过创建一个临时从库，再把数据同步到误操作的前一个事务，来恢复误删除的数据。

但是临时准备一套从库，会多花费很多时间，那有没有更快的办法呢?

这一篇文章，就讲一下通过延迟从库，来恢复误删除的数据。

比如平时这个从库都是延迟主库1小时，当主库出现误操作，从库直接同步到误操作前一个事务，这样从库的数据就是误操作前一刻的数据。

这样，再把数据导入到之前误操作的主库，完成恢复。

我们来开始实验。

1 配置延迟从库

在原实例进行全量备份

```
cd /data/backup
xtrabackup --defaults-file=/data/mysql/conf/my.cnf -uu_xtrabackup -p'Ijnbgt@123'
--backup --stream=xbstream --target-dir=./ >/data/backup/xtrabackup.xbstream
```

传到从库

```
scp xtrabackup.xbstream 192.168.12.162:/data/backup/recover
```

关闭从库的MySQL实例

```
/etc/init.d/mysql.server stop
```

清空目标实例数据目录和事务日志目录：

```
rm /data/mysql/data/* -rf
rm /data/mysql/binlog/* -rf
```

并把全备恢复到新的 MySQL 中

```
cd /data/backup/recover
xbstream -x < xtrabackup.xbstream
xtrabackup --prepare --target-dir=./
xtrabackup --defaults-file=/data/mysql/conf/my.cnf --copy-back --target-dir=./

chown -R mysql.mysql /data/mysql
/etc/init.d/mysql.server start
```

登录从库MySQL

```
stop slave;
reset slave;

change master to
master_host='192.168.12.161',
master_user='repl',
master_password='Uid_dQc63',
master_delay=3600,
master_auto_position = 1;
start slave;
```

其中，master_delay表示从库延迟的秒数。

2 主库写入测试数据

```
create database recover1;
use recover1;

CREATE TABLE test_recover (
id int NOT NULL AUTO_INCREMENT,
a int NOT NULL,
PRIMARY KEY (id)
) ENGINE=InnoDB CHARSET=utf8mb4;

insert into test_recover values (1,1),(2,2);
```

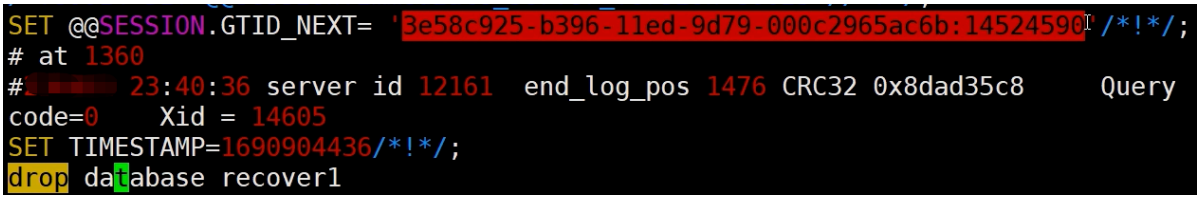
3 模拟主库误操作

```
drop database recover1;
```

4 确定误操作的GTID

```
cd /data/mysql/binlog
cp mysql-bin.000065 /data/backup/
cd /data/backup/
mysqlbinlog mysql-bin.000065 --start-datetime='2023-07-31 22:00:00'  --stop-
datetime='2023-07-31 22:50:00'  --base64-output=decode-rows -v
>/data/backup/1.sql
```

再来查看/data/backup/1.sql这个文件，搜索drop关键字，内容如下：



那误操作事务的GTID就是：3e58c925-b396-11ed-9d79-000c2965ac6b:14524559。

5 配置同步到误操作前一个事务

从库执行

```
stop slave
CHANGE MASTER TO MASTER_DELAY = 0;

start slave sql_thread  until sql_before_gtids='3e58c925-b396-11ed-9d79-
000c2965ac6b:14524559';
start slave io_thread;
```

也就是把从库同步到主库误操作前一个事务。

这样，数据就能恢复到误操作前一刻了。

6 导出误操作的数据在主库恢复

在从库备份主库误删除的库，操作如下：

```
mysqldump -u'root' -p --set-gtid-purged=off -B recover1 >recover1.sql
scp recover1.sql 192.168.12.161:/data/backup/
```

再到原来的实例，确定recover库是没有的，如果没有，则导入从库传过来的备份：

```
mysql -uroot -p <recover1.sql
```

7 确定数据是否恢复

再来查询误删除库recover1里表的数据：

```
select * from recover1.test_recover;
```

如果能查到(1,1),(2,2)这两行数据，说明通过延迟从库恢复数据成功。

1-12 通过ChatGPT编写复制创建脚本

在主库创建备份用户

```
create user u_bak@'%' identified by 'Ijd71Gcd_a';
grant select,reload,process,lock tables,replication
client,replication_slave_admin,show view,trigger on *.* to 'u_bak'@'%';
```

再来创建复制用户

```
CREATE USER 'repl'@'%' identified  WITH mysql_native_password BY 'Uid_dQc63';

GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%';
```

通过ChatGPT编写复制创建脚本

提示词

> 编写一个自动创建主从复制的Shell脚本
>
> 复制是基于GTID的
>
> 脚本是在从库执行
>
> 主库的IP是192.168.152.70
>
> 主库的备份用户是u_bak，密码是Ijd71Gcd_a
>
> 需要从从库通过mysqldump远程备份主库的数据，再导入到从库
>
> 主库的复制用户是repl，密码是Uid_dQc63
>
> 从库的用户是root，密码是Auca@502846303

生成的脚本如下：

```
#!/bin/bash
```

```
# 主库信息
master_host="192.168.12.161"
backup_user="u_bak"
backup_password="Ijd71Gcd_a"

# 从库信息
replication_user="repl"
replication_password="Uid_dQc63"
slave_host="localhost"
root_user="root"
root_password="Auca@502846303"

# 从库备份数据并导入
echo "Backup and import data on slave..."
mysqldump --single-transaction --master-data=2 -h $master_host -u $backup_user -
p$backup_password --all-databases > backup.sql
mysql -h $slave_host -u $root_user -p$root_password < backup.sql

# 设置从库复制参数
echo "Setting up replication on slave..."
mysql -h $slave_host -u $root_user -p$root_password -e "CHANGE MASTER TO
MASTER_HOST='$master_host', MASTER_USER='$replication_user',
MASTER_PASSWORD='$replication_password', MASTER_AUTO_POSITION=1;"
mysql -h $slave_host -u $root_user -p$root_password -e "START SLAVE;"

# 检查从库复制状态
echo "Checking slave replication status..."
mysql -h $slave_host -u $root_user -p$root_password -e "SHOW SLAVE STATUS\G"

echo "Done."
```

验证数据同步

主库写入数据

```
create database two;
```

从库确认

```
show databases;
```

写在最后

如果大家想系统的学习DBA，欢迎购买我的DBA体系课：

https://class.imooc.com/sale/dba

更多MySQL干货，欢迎关注公众号：


微信搜一搜
MySQL数据库联盟