

MySQL高可用集群篇

1. 集群搭建之主从复制

1.1 主从复制作用

主从架构有什么用？

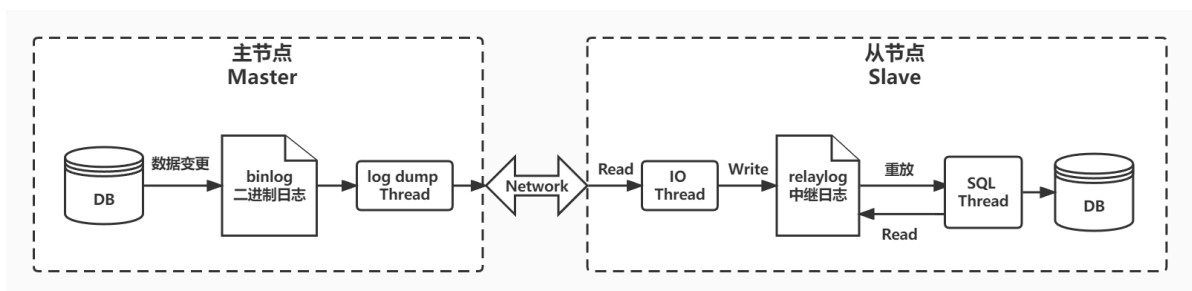
通过搭建MySQL主从集群，可以缓解MySQL的数据存储以及访问的压力。

1. **数据安全（主备）**：给主服务增加一个数据备份。基于这个目的，可以搭建主从架构，或者也可以基于主从架构搭建互主的架构。
2. **读写分离（主从）**：对于大部分的Java业务系统来说，都是读多写少的，读请求远远高于写请求。这时，当主服务的访问压力过大时，可以将数据读请求转为由从服务来分担，主服务只负责数据写入的请求，这样大大缓解数据库的访问压力。
3. **故障转移-高可用**：当MySQL主服务宕机后，可以由一台从服务切换成为主服务，继续提供数据读写功能。

对于高可用架构，主从数据的同步也只是实现故障转移的一个前提条件，要实现MySQL主从切换，还需要依靠一些其他的中间件来实现。比如MMM、MHA、MGR。

在一般项目中，如果数据库的访问压力没有那么大，那读写分离不一定是必须要做的，但是，主从架构和高可用架构则是必须要搭建的。

1.2 主从复制原理



MySQL服务的主从架构都是通过 binlog 日志文件来进行的。

具体流程如下：

1. 在主服务上打开binlog记录每一步的数据库操作
2. 然后，从服务上会有一个IO线程，负责跟主服务建立一个TCP连接，请求主服务将binlog传输过来
3. 这时，主库上会有一个IO dump线程，负责通过这个TCP连接把binlog日志传输给从库的IO线程
4. 主服务器MySQL服务将所有的写操作记录在 binlog 日志中，并生成 log dump 线程，将 binlog 日志传给从服务器MySQL服务的 I/O 线程。
5. 接着从服务的IO线程会把读取到的binlog日志数据写入自己的relay日志文件中。
6. 然后从服务上另外一个SQL线程会读取relay日志里的内容，进行操作重演，达到还原数据的目的。

注意：

1. 主从复制是异步的逻辑的 SQL 语句级的复制
2. 复制时，主库有一个 I/O 线程，从库有两个线程，即 I/O 和 SQL 线程
3. 实现主从复制的必要条件是主库要开启记录 binlog 的功能
4. 作为复制的所有 MySQL 节点的 server-id 都不能相同
5. binlog 文件只记录对数据内容有更改的 SQL 语句，不记录任何查询语句
6. 双方MySQL必须版本一致，至少需要主服务的版本低于从服务
7. 两节点间的时间需要同步

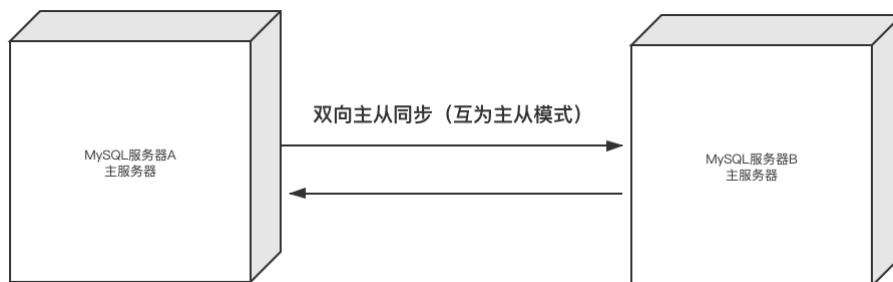
主从复制形式：

1. 一主一从
2. 主主复制
3. 一主多从
4. 多主一从
5. 级联复制

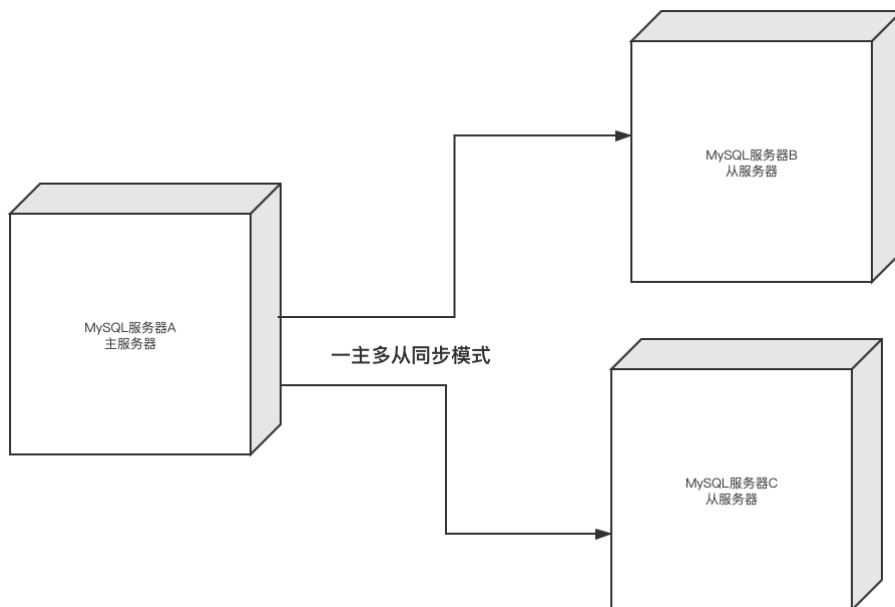
1) 一主一从（备份）



2) 主主复制

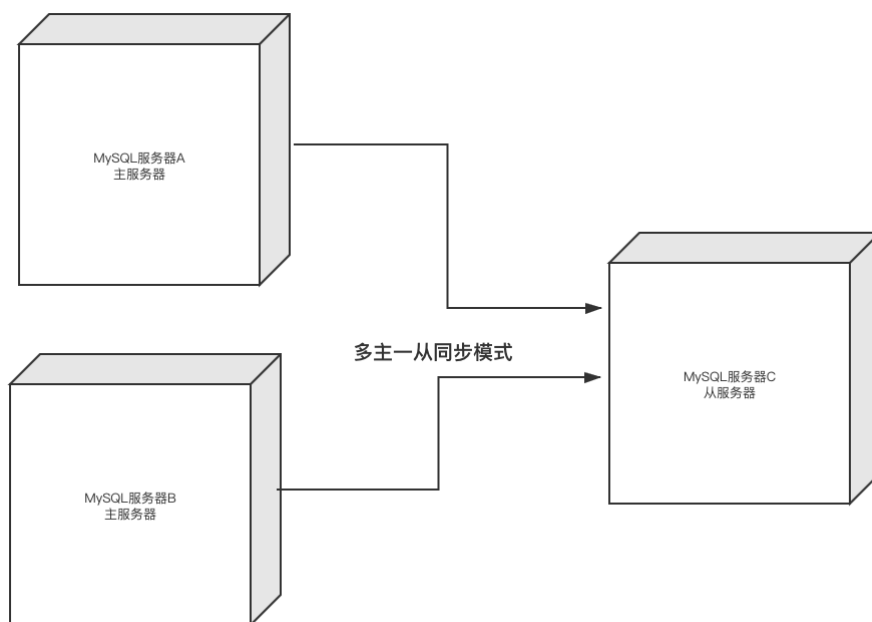


3) 一主多从

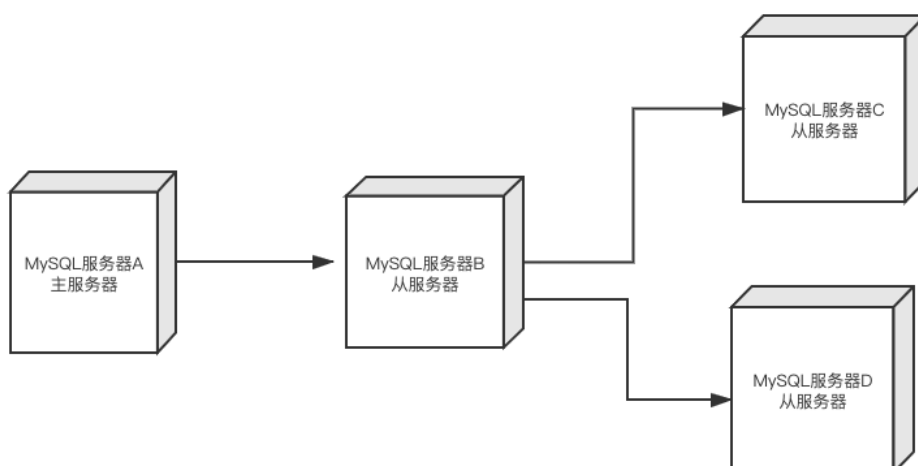


4) 多主一从

5.7后开始支持



5) 联级复制



1.3 binlog和relay日志

- **binlog**：二进制日志，将数据改变记录到二进制（binary）日志中，可用于本机数据恢复和主从同步。
- **relaylog**：中继日志，Slave节点会把中继日志中的事件信息一条一条的在本地执行一次，实现主从同步这个过程也叫数据重放。

1.3.1 binlog的三种模式

1) ROW模式

日志中会记录成每一行数据被修改的形式，然后在slave端再对相同的数据进行修改

- **优点**：binlog中可以**不记录执行的sql语句的上下文相关的信息**，仅仅只需要记录那一条记录被修改了，修改成什么样了。所以ROW模式的日志的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**存储过程或函数**，以及**触发器**的调用和触发无法被正确复制的问题。
- **缺点**：ROW模式下，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记录，这样会产生大量的日志内容。
 - 比如：有这样一条update语句：update product set owner_member_id='d' where owner_member_id='a'，执行之后，日志中记录的**不是**这条update语句所对应的事件（MySQL是以事件的形式来记录binlog日志），而是这条语句所更新的每一条记录的变化情况，这样就记录成很多条记录被更新的很多事件，自然binlog日志的量会很大。

2) Statement模式

Statement模式：每一条修改数据的SQL都会记录到Master的binlog中。Slave在复制的时候SQL进程会解析成和原来Master端执行过的相同的SQL来再次执行。

- **优点**：Statement模式下的优点，首先就是**解决了ROW模式下的缺点**，不需要记录每一行数据的变化，减少binlog日志量，节约io，提高性能。因为他**只需要记录在master上所执行的语句的细节，以及执行语句时候的上下文的信息**。
- **缺点**：由于它是记录的执行语句，所以为了让这些语句在Slave端也能正确执行，那么他还必须记录每条语句在执行的时候的一些相关信息，也就是上下文信息，以保证所有语句在Slave端被执行的时候能够得到和在Master端执行时候相同的结果。
 - 另外，由于MySQL现在发展比较快，很多新功能加入，使MySQL的复制遇到了不小的挑战，自然复制的时候涉及到越复杂的内容，BUG也就越容易出现。在Statement模式下，目前已经发现的就有不少情况会造成MySQL的复制出BUG，主要是修改数据的时候使用了某些特定的函数或者功能的时候会出现。
 - 比如：sleep()在有些版本就不能正确复制。

3) Mixed模式

- 实际上就是**前两种模式的结合**，在Mixed模式下，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**之间选一种。
- 新版本中的Statement模式还是和以前一样，仅仅记录执行的语句。而新版本的MySQL中对ROW模式被做了优化，并不是所有的修改都会以ROW模式来记录，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。

1.3.2 开启binlog

修改my.cnf文件

在[mysqld]段下添加：

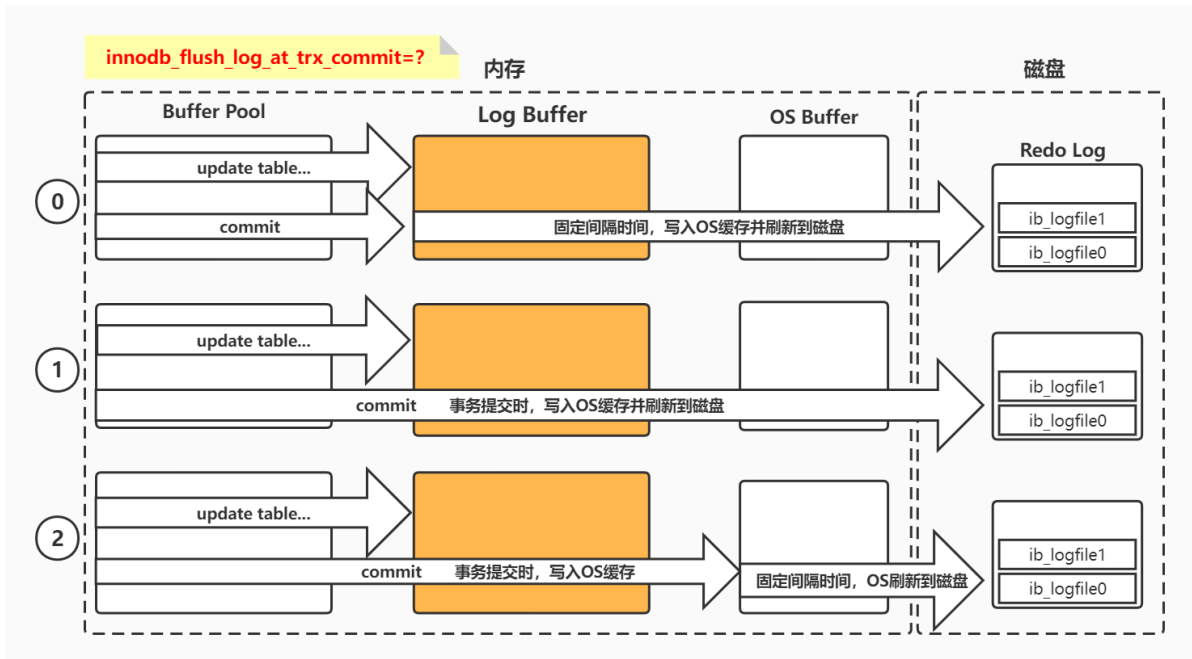
```
1  # binlog刷盘策略
2  sync_binlog=1
3  # 需要备份的数据库
4  binlog-do-db=hello
5  # 不需要备份的数据库
6  binlog-ignore-db=mysql
7  # 启动二进制文件
8  log-bin=mysql-bin
9  # 服务器ID
10 server-id=132
11 #只保留7天的二进制日志，以防磁盘被日志占满(可选)
12 expire-logs=7
```

sync_binlog参数：

- **0**：存储引擎不进行binlog的刷新到磁盘，而由操作系统的文件系统控制缓存刷新。
- **1**：每提交一次事务，存储引擎调用文件系统的sync操作进行一次缓存的刷新，这种方式最安全，但性能较低。
- **n**：当提交的日志组=n时，存储引擎调用文件系统的sync操作进行一次缓存的刷新。

sync_binlog=0或sync_binlog大于1，事务被提交，而尚未同步到磁盘。因此，在电源故障或操作系统崩溃时有可能服务器已承诺尚未同步一些事务到二进制日志。因此它是不可能执行例行程序恢复这些事务，他们将会丢失二进制日志。

类似redo日志的刷盘机制，如下图：



1.3.3 调整binlog日志模式

查看binlog的日志模式：

```
1 mysql> show variables like 'binlog_format';
2 +-----+-----+
3 | Variable_name | value |
4 +-----+-----+
5 | binlog_format | ROW   |
6 +-----+-----+
7 1 row in set (0.00 sec)
```

调整binlog的日志模式：binlog的三种格式： `STATEMENT`、`ROW`、`MIXED`。

```
1 mysql> set binlog_format=STATEMENT;
2 Query OK, 0 rows affected (0.00 sec)
3
4 mysql> show variables like 'binlog_format';
5 +-----+-----+
6 | Variable_name | value   |
7 +-----+-----+
8 | binlog_format | STATEMENT |
9 +-----+-----+
10 1 row in set (0.00 sec)
```

1.3.4 如何查看binlog和relaylog日志？

方式一：使用mysqlbinlog查看binlog日志文件

因为binlog日志文件：mysql-bin.000005是二进制文件，没法用vi等打开，这时就需要mysql的自带的mysqlbinlog工具进行解码，执行：`mysqlbinlog mysql-bin.000005` 可以将二进制文件转为可阅读的sql语句。

```
1 | mysqlbinlog --base64-output=decode-rows -v -v mysql-bin.000001 > binlog.txt
```

方式二：在MySQL终端查看binlog

`show master logs`，查看所有二进制日志列表，和 `show binary logs` 同义。

```
1 | mysql> show master logs;
2 | +-----+-----+
3 | | Log_name          | File_size |
4 | +-----+-----+
5 | | mysql-bin.000001 |         385 |
6 | +-----+-----+
7 | 1 row in set (0.00 sec)
```

使用 `show binlog events` 命令可以以列表的形式显示日志中的事件信息。

`show binlog events`命令的格式：

```
1 | show binlog events [IN 'log_name'] [FROM pos] [LIMIT [offset,] row_count];
```

说明：

- IN 'log_name': 指定要查询的binlog文件名（如果省略此参数，则默认指定第一个binlog文件）；
- FROM pos: 指定从哪个pos起始点开始查起（如果省略此参数，则从整个文件的第一个pos点开始算）；
- LIMIT [offset]：偏移量（默认为0）；
- row_count: 查询总条数（如果省略，则显示所有行）。

```
1 | mysql> show binlog events in 'mysql-bin.000001';
2 | +-----+-----+-----+-----+-----+-----+
3 | | Log_name          | Pos | Event_type      | Server_id | End_log_pos | Info
4 | +-----+-----+-----+-----+-----+-----+
5 | | mybin.000001      | 4   | Format_desc     | 132       | 123         | Server ver: 5.7.30-log, Binlog ver: 4
6 | | mybin.000001      | 123 | Previous_gtid   | 132       | 154         |
7 | +-----+-----+-----+-----+-----+-----+
8 | 2 rows in set (0.00 sec)
```

切换binlog文件：

```
1 | mysql> flush logs;
2 | query OK, 0 rows affected (0.00 sec)
```

注意：刷新日志会生成一个新的日志文件

1.4 案例：基于Pos主从复制

注意：在做案例之前，最好是卸载已有的MySQL，并安装全新的MySQL(5.7版本)。

1.4.1 开放端口

需要将3306端口放行，如果是内网也可关闭防火墙

1.4.2 主服务器配置

查看binlog是否开启可以使用命令：

```
1 | mysql> show variables like 'log_bin%';
2 | +-----+-----+
3 | | variable_name | value |
4 | +-----+-----+
5 | | log_bin      | OFF  |
6 | | log_bin_basename |      |
7 | | log_bin_index  |      |
8 | | log_bin_trust_function_creators | OFF  |
9 | | log_bin_use_v1_row_events | OFF  |
10 | +-----+-----+
11 | 5 rows in set (0.12 sec)
```

log_bin如果是OFF代表是未开启状态

```
mysql> show variables like 'log_bin%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| log_bin       | ON    |
| log_bin_basename | /var/lib/mysql/mysql-bin |
| log_bin_index  | /var/lib/mysql/mysql-bin.index |
| log_bin_trust_function_creators | OFF |
| log_bin_use_v1_row_events | OFF |
+-----+-----+
5 rows in set (0.00 sec)
```

第一步：修改my.cnf文件

在[mysqld]段下添加：


```
1 # binlog刷盘策略
2 sync_binlog=1
3 # 需要备份的数据库
4 binlog-do-db=hello
5 # 不需要备份的数据库
6 binlog-ignore-db=mysql
7 # 启动二进制文件
8 log-bin=mysql-bin
9 # 服务器ID
10 server-id=132
```

第二步：重启mysql服务

```
1 systemctl restart mysqld
```

第三步：主机给从机授备份权限

注意：先要登录到MySQL命令客户端

```
1 mysql> GRANT REPLICATION SLAVE ON *.* TO '从机MySQL用户名'@'从机IP' identified
  by '从机MySQL密码';
```

示例：

```
1 GRANT REPLICATION SLAVE ON *.* TO 'root'@'%' identified by 'root';
```

注意：一般不用root帐号，“%”表示所有客户端都可能连，只要帐号，密码正确，此处可用具体客户端IP代替，如39.99.131.178，加强安全。

mysql5.7对密码的强度是有要求的，必须是字母+数字+符号组成的，可以使用如下方法调整密码强度

设置密码长度最低位数

```
mysql> set global validate_password_length=4;
```

设置密码强度级别

```
mysql> set global validate_password_policy=0;
```

validate_password_policy有以下取值：

Policy	Tests Performe
0 or LOW	Length
1 or MEDIUM	numeric, lowercase/uppercase, and special characters
2 or STRONG	Length; numeric, lowercase/uppercase, and special characters

默认是1，即MEDIUM，所以刚开始设置的密码必须符合长度，且必须含有数字，小写或大写字母，特殊字符。

第四步：刷新权限

```
1 | mysql> FLUSH PRIVILEGES;
```

第五步：查询master的状态

```
1 | mysql> show master status;
2 | +-----+-----+-----+-----+
3 | | File          | Position | Binlog_Do_DB | Binlog_Ignore_DB |
   | Executed_Gtid_Set |
4 | +-----+-----+-----+-----+
5 | | mysql-bin.000002 |      593 | hello        | mysql              |
   | |
6 | +-----+-----+-----+-----+
7 | 1 row in set
```

1.4.3 从服务器配置

第一步：修改my.conf文件

```
1 | [mysqld]
2 | server-id=133
```

第二步：重启mysqld服务

```
1 | systemctl restart mysqld
```

第三步：重启并登录到MySQL进行配置Slave

```
1 | mysql>change master to
2 | master_host='172.17.187.78',
3 | master_port=3306,
4 | master_user='root',
5 | master_password='root',
6 | master_log_file='mysql-bin.000001',
7 | master_log_pos=1109,
8 | MASTER_AUTO_POSITION=0;
```

注意：语句中间不要断开，`master_port`为MySQL服务器端口号（无引号），`master_user`为执行同步操作的数据库账户，`593`无单引号（此处的`1109`就是`show master status`中看到的`position`的值，这里的`mysql-bin.000001`就是`file`对应的值）。

第四步：启动从服务器复制功能

```
1 mysql>start slave;
```

第五步：检查从服务器复制功能状态

```
1 mysql> show slave status \G;
2 .....(省略部分)
3 Slave_IO_Running: Yes //此状态必须YES
4 Slave_SQL_Running: Yes //此状态必须YES
5 .....(省略部分)
```

注：Slave_IO及Slave_SQL进程必须正常运行，即YES状态，否则都是错误的状态(如：其中一个NO均属错误)。

保存 查询创建工具 美化 SQL 代码段 文本 导出结果 创建图表

Slave【从节点】 hello 运行 停止 解释

1 show slave status;

信息	摘要	结果 1	剖析	状态
	Relay_Log_Pos	Relay_Master_Log_File	Slave_IO_Running	Slave_SQL_Running
	320	mysql-bin.000002	Yes	Yes

以上操作过程，从服务器配置完成。

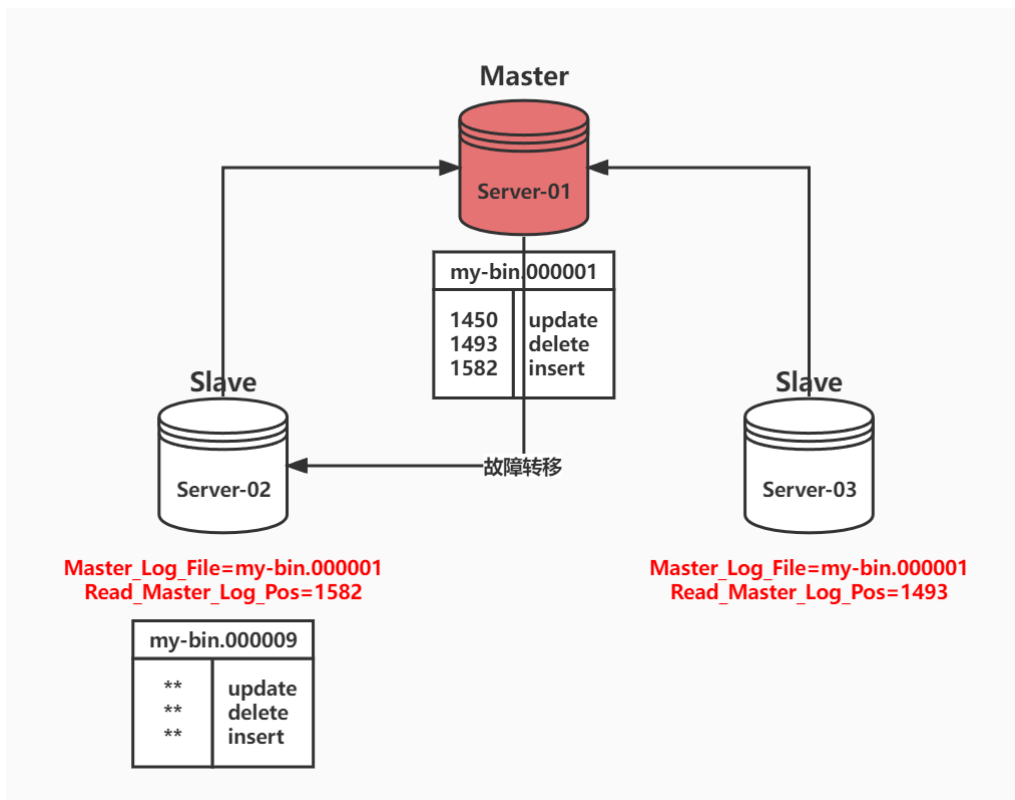
1.4.4 测试

搭建成功之后，往主机中插入数据，看看从机中是否有数据

1.4.5 思考这么做会有什么问题？

请至少说出三个问题：

1. 故障转移
2. 数据一致性：同步丢失
 - 半同步：数据同步中间件Canal，MQ
3. 同步延迟
4. 只同步一部分，全库同步
5. ...



1.5 案例：基于GTID的主从复制

1.5.1 什么是GTID?

从MySQL 5.6.5开始新增了一种基于GTID的复制方式。GTID即全局事务ID（Global Transaction Identifier），其保证每个主节点上提交的事务，在从节点可以一致性的复制。

这种方式强化了数据库的主备一致性，故障恢复以及容错能力。GTID在一主一从情况下没有优势，对于**两主以上**的结构优势异常明显，可以在数据不丢失的情况下切换新主。

GTID实际上是由UUID+TID (即transactionId)组成的，其中UUID(即server_uuid)产生于auto.cnf文件，是一个MySQL实例的唯一标识。TID代表了该实例上已经提交的事务数量，并且随着事务提交单调递增，所以GTID能够保证每个MySQL实例事务的执行。**GTID在一组复制中，全局唯一。**通过GTID的UUID可以知道这个事务在哪个实例上提交的。

```
1 | cat /var/lib/mysql/auto.cnf
```

```
[root@hero02 ~]# cat /var/lib/mysql/auto.cnf
[auto]
server-uuid=52010ef6-550b-11ed-8295-00163e35c94b
```

GTID的具体形式：

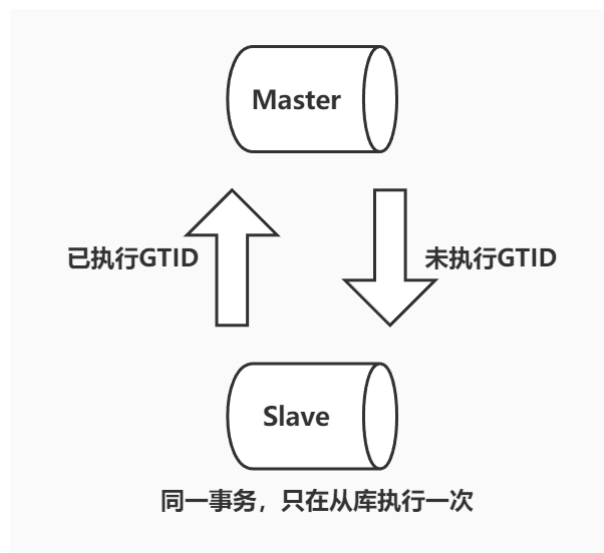
```

1  mysql> show master status;
2  +-----+-----+-----+-----+-----+
3  | File          | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set
4  +-----+-----+-----+-----+-----+
5  | bin.000004    | 1147     | hello        | mysql             | 52010ef6-550b-11ed-8295-00163e35c94b:1-57
6  +-----+-----+-----+-----+-----+
7  1 row in set (0.00 sec)
8
9  GTID:52010ef6-550b-11ed-8295-00163e35c94b:1-57
10 UUID:52010ef6-550b-11ed-8295-00163e35c94b
11 transactionId:1-57
12 transactionId 是在该主库上生成的事务序列号，从1开始，1-2代表第二个事务；第1-n代表n个事务。

```

官网: <https://dev.mysql.com/doc/refman/5.7/en/replication-gtids-lifecycle.html>

GTID可以保证不会重复执行同一个事务，并且会补全没有执行的事务；



优势：

- GTID相对于binlog+pos 复制数据安全性更高、failover更简单、搭建主从复制更简单
 - 实现 failover不用像binlog+pos 复制那样需要找 log_file 和 log_pos
 - 一个 GTID 在一个node上只执行一次，避免重复执行导致数据混乱或者主从不一致
- 根据 GTID 可以快速的确定事务最初是在哪个实例上提交
- 使得 DBA 在运维中做集群变迁时更加方便

注意事项（不足）：

- 在一个复制组中必须要求统一开启GTID或是关闭GTID
- 主从库的表存储引擎必须是一致，不允许一个SQL同时更新一个事务引擎和非事务引擎的表
- MySQL在主从复制时如果要跳过报错，可以采取以下方式跳过SQL（event）组成的事务，但GTID不支持以下方式

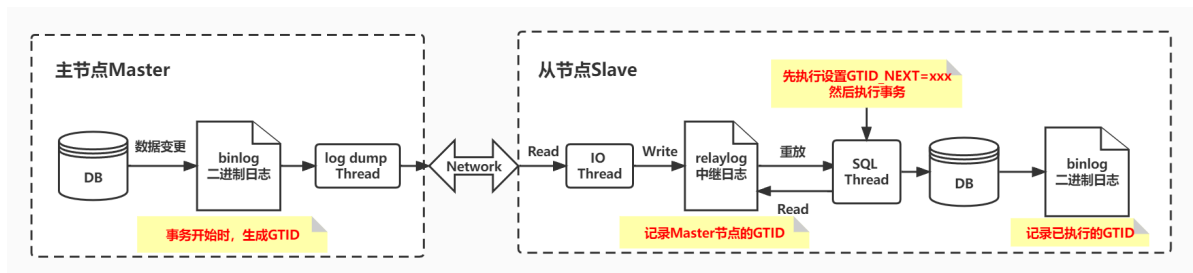
```

1 set global SQL_SLAVE_SKIP_COUNTER=1;
2 start slave sql_thread;

```

- 不支持如下语句复制（主库直接报错）
 - create table...select
 - create temporary table
 - drop temporary table
 - ...

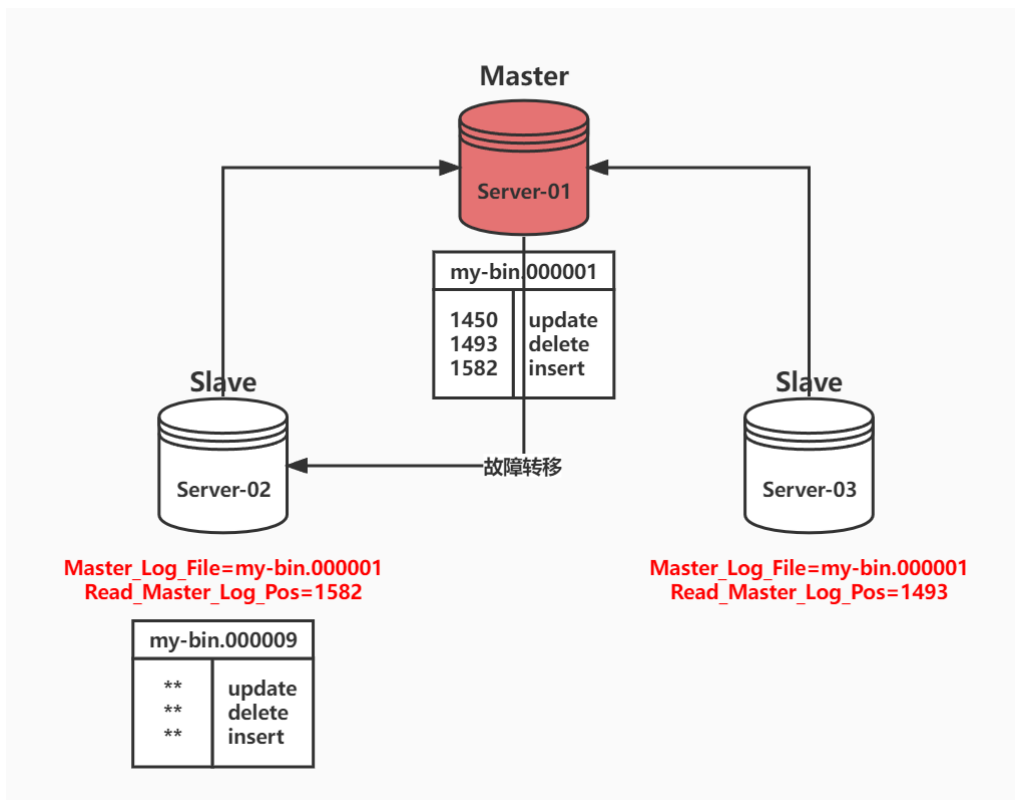
1.5.2 GTID 主从复制原理



- Master更新数据时，会在事务前产生GTID一同记录到binlog日志中
- Slave的IO Thread将变更后的binlog写入到本地的relaylog中，这其中含有Master的GTID
- SQL Thread读取这个 GTID 的值并设置 GTID_NEXT变量，告诉 Slave下一个要执行的 GTID 值，然后对比 Slave 端的 binlog 是否有该 GTID
 - 如果有，说明该 GTID 的事务已经执行 Slave 会忽略
 - 如果没有，Slave 就会执行该 GTID 事务，并记录该 GTID 到自身的 binlog
- 在解析过程中会判断是否有主键，如果没有就用二级索引，如果没有二级索引就用全表扫描

1.5.3 GTID解决了什么问题？

通过GTID可以很方便的进行复制结构上的**故障转移**，新主设置，这就很好地解决了下面这个图所展现出来的问题。



如果没有GTID:

- Server01(Master)崩溃: 根据从上show slave status获得Master_log_File/Read_Master_Log_Pos的值来看
 - Server01(Master)最新pos是1582
 - Server02(Slave)是1582, 已经跟上了主
 - Server03(Slave)是1493, 没有跟上主
- 这时要是把Server02提升为主, Server03变成Server02的从。
- 如果没有全局事务ID, 在Server03上执行change的时候就需要做一些计算, 保证之前的事务都执行完毕了!
 - 进行主节点故障转移的时候就比较繁琐

如果使用GTID:

- 这个问题在GTID出现后, 就显得非常的简单。
- 由于同一事务的GTID在所有节点上的值一致, 那么根据Server03当前停止点的GTID就能定位到Server02上的GTID。
- 甚至由于MASTER_AUTO_POSITION功能的出现, 我们都不需要知道GTID的具体值, 直接使用CHANGE MASTER TO MASTER_HOST='xxx', MASTER_AUTO_POSITION命令就可以直接完成failover的工作。

注意: 在构建主从复制之前, 在一台将成为主的实例上进行一些操作(如: 数据清理等), 通过GTID复制, 这些在主从成立之前的操作也会被复制到从服务器上, 引起复制失败。也就是说通过GTID复制都是从最先开始的事务日志开始, 即使这些操作在复制之前执行。比如在server1上执行一些drop、delete的清理操作, 接着在server2上执行change的操作, 会使得server2也进行server1的清理操作。

1.5.4 搭建GTID同步集群

他的搭建方式跟我们上面的主从架构整体搭建方式差不多。只是需要在my.cnf中修改一些配置。

1) 主节点

```
1  gtid_mode=on
2  enforce_gtid_consistency=on
3
4  # 强烈建议，其他格式可能造成数据不一致
5  binlog_format=row
```

2) 从节点

```
1  gtid_mode=on
2  enforce_gtid_consistency=on
3
4  # 做级联复制的时候，再开启。允许下端接入slave
5  log_slave_updates=1
```

3) 使用GTID的方式，salve重新挂载master端：

启动以后最好不要立即执行事务，先change master上，然后在执行事务。

使用下面的sql切换slave到新的master。

```
1  # 停止从节点
2  stop slave;
3  # 切换主节点配置，比基于pos简单不少
4  change master to
5  master_host='172.17.187.78',
6  master_port=3306,
7  master_user='root',
8  master_password='root',
9  master_auto_position=1;
10 # 启动从节点
11 start slave;
```

4) 启动GTID的两种情况

分别重启主服务和从服务，就可以开启GTID同步复制，启动方法有两种情况

情况一：如果是新搭建的服务器，直接启动就行了

情况二：如果是在已经跑的服务器，需要重启mysqld

- 启动之前要先关闭master的写入，保证所有slave端都已经和master端数据保持同步
- 所有slave需要加上skip_slave_start=1的配置参数，避免启动后还是使用老的复制协议

```
1  # 避免启动后还是使用老的复制协议
2  skip_slave_start=1
```

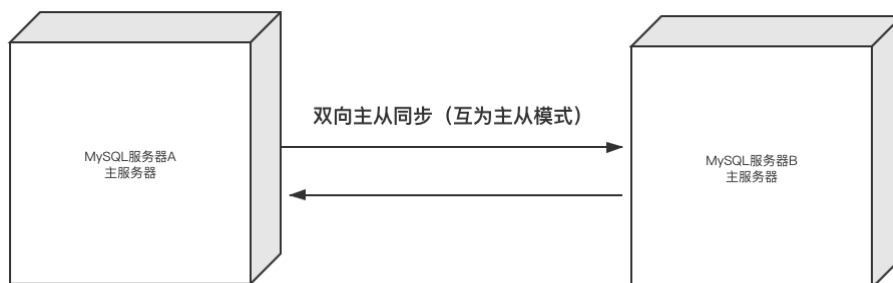

5) 测试

```
1 | show master status;
```

```
mysql> show master status;
+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| mysql-bin.000003 | 439      | hello        | mysql              | 52010ef6-550b-11ed-8295-00163e35c94b:1 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

1.6 案例：其他主从集群：一主多从、互为主从

1.6.1 互为主从



前面咱们搭建一主一从的MySQL主从同步集群，具有了数据同步的基础功能。而在生产环境中，通常会以此为基础，根据业务情况以及负载情况，搭建更大更复杂的集群。

例如：一主多从集群、多级从的主从集群、多主集群

- 一主多从：进一步提高整个集群的读能力
- 一主多级从：减轻主节点进行数据同步的压力
- 多主集群：提高整个集群的高可用能力
- 互主集群：我们也可以扩展出互为主从的互主集群甚至是环形的主从集群，实现多活部署

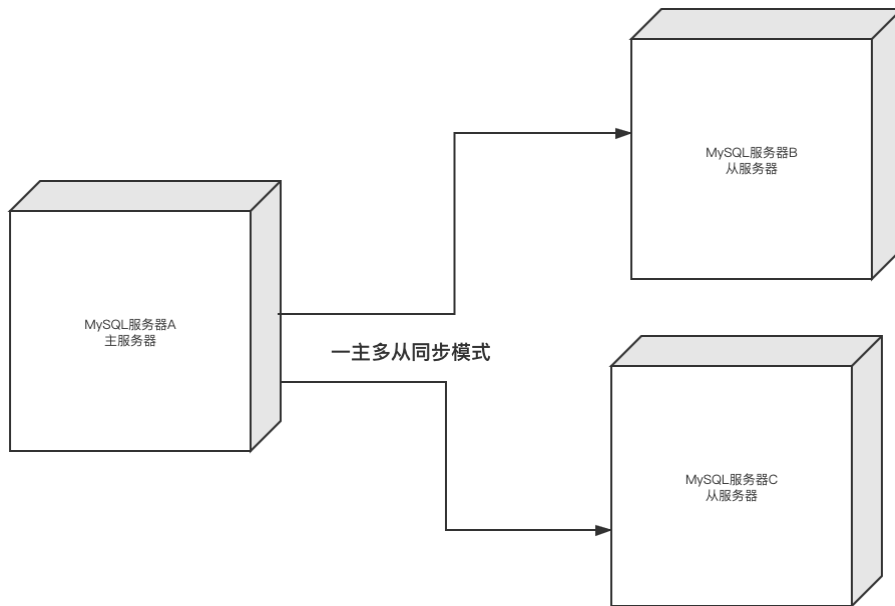
搭建互主集群只需要按照上面的方式，在主服务上打开一个slave进程，并且指向slave节点的binlog当前文件地址和位置。

```
mysql> show slave status \G;
***** 1. row *****
Slave_IO_State:
Master_Host: 127.0.0.1
Master_User: root
Master_Port: 3307
Connect_Retry: 60
Master_Log_File: master-bin.000002
Read_Master_Log_Pos: 2942
Relay_Log_File: DESKTOP-4ML7SEJ-relay-bin.000003
Relay_Log_Pos: 4
Relay_Master_Log_File: master-bin.000002
Slave_IO_Running: No
Slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:

mysql> show master status;
+-----+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+
| binlog.000002  | 375      | masterdemo   |                   |                    |
+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

搭建互主集群，只需要在主服务上打开slave服务，然后将binlog的文件和位置指向从服务的binlog文件和地址

1.6.2 一主多从



如果要扩展成一主多从的集群架构，其实更简单，只需要增加一个binlog复制就行了。但是，如果集群已经运行过一段时间，要扩展新的从节点就有一个问题，历史数据没办法通过binlog来恢复。所以扩展新的slave节点时，就需要增加数据复制的操作。

操作如下：操作的本质是数据备份恢复

- MySQL的数据备份恢复操作相对比较简单，可直接通过SQL语句直接来完成。

```
1 | mysqldump -u root -p --all-databases > backup.sql
```

- 通过这个指令，就可以将整个数据库的所有数据导出成backup.sql，然后把这个backup.sql分发到新的MySQL服务器上

```
1 | scp -P 22 /root/backup.sql root@172.17.187.81:/root
```

- 执行下面的指令将数据全部导入到新的MySQL从服务中

```
1 | mysql -uroot -p < backup.sql
```

- 这样新的MySQL服务就已经有了所有的历史数据。然后，再按照上面的步骤，配置Slave数据同步

1.7 案例：半同步复制机制

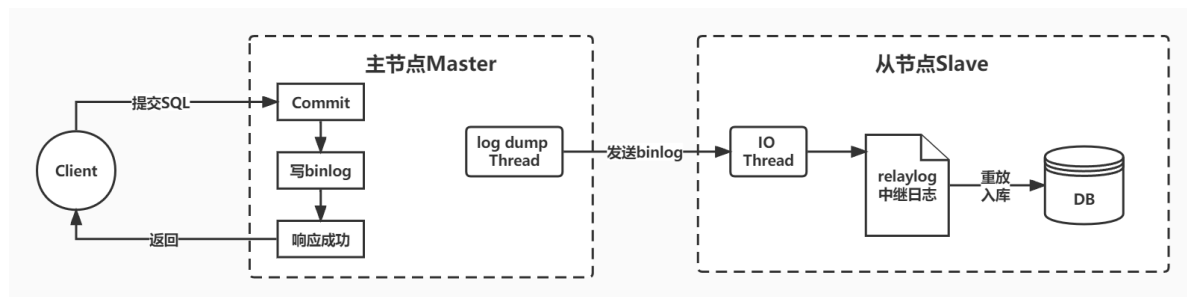
1.7.1 半同步复制介绍

到现在为止咱们已经可以搭建MySQL的主从集群，互主集群。但是，这里的集群都有一个隐患“会丢数据”，为什么呢？

1) 异步复制

MySQL主从集群默认采用的是一种**异步复制**的机制。

主服务在执行用户提交的事务后，写入binlog日志，然后就给客户端返回一个成功的响应了。而binlog会由一个dump线程异步发送给Slave从服务。由于这个发送binlog的过程是异步的。主服务在向客户端反馈执行结果时，是不知道binlog是否同步成功了。这时候如果主服务宕机了，而从服务还没有备份到新执行的binlog，那就有可能丢数据。



怎么解决？

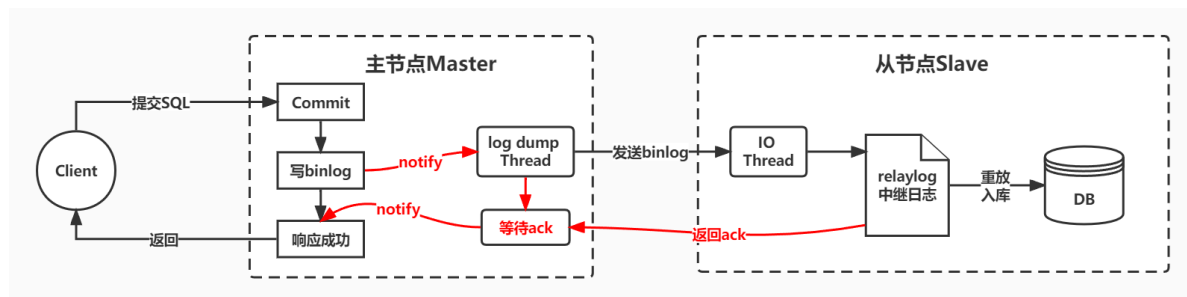
- 这就要靠MySQL的**半同步复制机制**来保证数据安全

2) 半同步复制

半同步复制机制是一种介于异步复制和全同步复制之前的机制。

主库在执行完客户端提交的事务后，并不是立即返回客户端响应，而是等待至少一个从库接收并写到relaylog中，才会返回给客户端。

MySQL在等待确认时，默认会等 10 秒，如果超过10秒没有收到ack，就会降级成为 异步复制。



这种半同步复制相比异步复制，能够有效的提高数据的安全性。但是这种安全性也不是绝对的，他只保证事务提交后的binlog至少传输到了一个从库，且并不保证从库应用这个事务的binlog是成功的。另一方面，半同步复制机制也会造成一定程度的延迟，这个延迟时间最少是一个TCP/IP请求往返的时间。整个服务的性能是会有所下降的。而当从服务出现问题时，主服务需要等待的时间就会更长，要等到从服务的服务恢复或者请求超时才能给用户响应。

1.7.2 搭建半同步复制集群

半同步复制需要基于特定的扩展模块来实现。而mysql从 5.5版本 开始，往上的版本都默认自带了这个模块。这个模块包含在mysql安装目录下的lib/plugin目录下的 `semisync_master.so` 和 `semisync_slave.so` 两个文件中。

需要在主服务上安装 `semisync_master` 模块，在从服务上安装 `semisync_slave` 模块。

```
[root@hero02 plugin]# pwd
/usr/lib64/mysql/plugin
[root@hero02 plugin]# ll
总用量 60088
-rwxr-xr-x 1 root root 103656 8月 30 12:42 adt_null.so
-rwxr-xr-x 1 root root 357752 8月 30 12:42 authentication_ldap_sasl_client.so
-rwxr-xr-x 1 root root 43856 8月 30 12:43 auth_socket.so
-rwxr-xr-x 1 root root 943448 8月 30 12:43 connection_control.so
drwxr-xr-x 2 root root 4096 10月 26 16:49 debug
-rwxr-xr-x 1 root root 22264328 8月 30 12:43 group_replication.so
-rwxr-xr-x 1 root root 485696 8月 30 12:42 ha_example.so
-rwxr-xr-x 1 root root 947336 8月 30 12:43 innodb_engine.so
-rwxr-xr-x 1 root root 953096 8月 30 12:43 keyring_file.so
-rwxr-xr-x 1 root root 462208 8月 30 12:43 keyring_def.so
-rwxr-xr-x 1 root root 1265096 8月 30 12:43 libmemcached.so
-rwxr-xr-x 1 root root 8974152 8月 30 12:43 libpluginmecab.so
-rwxr-xr-x 1 root root 21616 8月 30 12:43 locking_service.so
-rwxr-xr-x 1 root root 54192 8月 30 12:43 mypluglib.so
-rwxr-xr-x 1 root root 41488 8月 30 12:43 mysql_no_log.so
-rwxr-xr-x 1 root root 22110152 8月 30 12:44 mysql_native_password.so
-rwxr-xr-x 1 root root 49616 8月 30 12:43 rewrite_example.so
-rwxr-xr-x 1 root root 592616 8月 30 12:43 rewriter.so
-rwxr-xr-x 1 root root 937936 8月 30 12:43 semisync_master.so
-rwxr-xr-x 1 root root 160888 8月 30 12:43 semisync_slave.so
-rwxr-xr-x 1 root root 210040 8月 30 12:43 validate_password.so
-rwxr-xr-x 1 root root 507200 8月 30 12:43 version_token.so
```

检查是否支持动态插件

```
1 mysql> select @@have_dynamic_loading;
2 +-----+
3 | @@have_dynamic_loading |
4 +-----+
5 | YES                      |
6 +-----+
7 1 row in set (0.00 sec)
```

查看半同步插件位置:

```
1 mysql> show variables like 'plugin_dir';
2 +-----+-----+
3 | variable_name | value |
4 +-----+-----+
5 | plugin_dir    | /usr/lib64/mysql/plugin/ |
6 +-----+-----+
7 1 row in set (0.00 sec)
8
9 mysql> system ls -l /usr/lib64/mysql/plugin/ |grep semi
10 -rwxr-xr-x 1 root root 937936 8月 30 12:43 semisync_master.so
11 -rwxr-xr-x 1 root root 160888 8月 30 12:43 semisync_slave.so
```

1) 登陆Master, 安装semisync_master模块:

```
1 # 通过扩展库来安装半同步复制模块, 需要指定扩展库的文件名。
2 mysql> install plugin rpl_semi_sync_master soname 'semisync_master.so';
3 Query OK, 0 rows affected (0.01 sec)
4
5 # 查看系统全局参数
6 # rpl_semi_sync_master_timeout就是半同步复制时等待应答的最长等待时间, 默认是10秒, 可以根据情况自行调整。
7 mysql> show global variables like 'rpl_semi%';
8 +-----+-----+
9 | variable_name | value |
10 +-----+-----+
```

```

11 | rpl_semi_sync_master_enabled | OFF |
12 | rpl_semi_sync_master_timeout | 10000 |
13 | rpl_semi_sync_master_trace_level | 32 |
14 | rpl_semi_sync_master_wait_for_slave_count | 1 |
15 | rpl_semi_sync_master_wait_no_slave | ON |
16 | rpl_semi_sync_master_wait_point | AFTER_SYNC |
17 +-----+
18 6 rows in set, 1 warning (0.02 sec)
19
20 # 打开半同步复制的开关
21 mysql> set global rpl_semi_sync_master_enabled=ON;
22 query OK, 0 rows affected (0.00 sec)
23
24 # 单位是毫秒，可动态调整，表示主库事务等待从库返回commit成功信息超过30秒就降为异步模式，
25 set global rpl_semi_sync_master_timeout=30000;
26
27 # 检查是否有成功加载
28 mysql> select * from mysql.plugin;
29 +-----+
30 | name | dl |
31 +-----+
32 | rpl_semi_sync_master | semisync_master.so |
33 | validate_password | validate_password.so |
34 +-----+
35 2 rows in set (0.00 sec)

```

在第二行查看系统参数时，最后的一个参数 `rpl_semi_sync_master_wait_point` 其实表示一种半同步复制的方式。

半同步复制有两种方式：

- **AFTER_SYNC 方式**：默认，主库把日志写入binlog并且复制给从库，然后开始等待从库的响应。**从库返回成功后，主库再提交事务**，接着给客户端返回一个成功响应
 - 性能更差，安全性更好
- **AFTER_COMMIT 方式**：主库把日志写入binlog并且复制给从库，**主库就提交自己的本地事务，再等待从库返回给自己一个成功响应**，等到响应后主库再给客户端返回响应。
 - 性能更好，安全性较差
- **区别：提交事务的时机不同**

2) 登陆Slave，安装semisync_slave模块

```

1  mysql> install plugin rpl_semi_sync_slave soname 'semisync_slave.so';
2  query OK, 0 rows affected (0.01 sec)
3
4  mysql> show global variables like 'rpl_semi%';
5  +-----+
6  | variable_name | value |
7  +-----+
8  | rpl_semi_sync_slave_enabled | OFF |
9  | rpl_semi_sync_slave_trace_level | 32 |
10 +-----+
11 2 rows in set, 1 warning (0.01 sec)
12

```

```

13 mysql> set global rpl_semi_sync_slave_enabled = on;
14 query OK, 0 rows affected (0.00 sec)
15
16 mysql> show global variables like 'rpl_semi%';
17 +-----+-----+
18 | variable_name | value |
19 +-----+-----+
20 | rpl_semi_sync_slave_enabled | ON |
21 | rpl_semi_sync_slave_trace_level | 32 |
22 +-----+-----+
23 2 rows in set, 1 warning (0.00 sec)
24
25 mysql> stop slave;
26 query OK, 0 rows affected (0.01 sec)
27
28 mysql> start slave;
29 query OK, 0 rows affected (0.01 sec)

```

slave端的安装过程基本差不多，不过要注意下安装完slave端的半同步插件后，需要重启下slave服务。

3) 监控半同步复制环境

主服务

```

1  mysql> show status like 'rpl_semi_sync%';
2  +-----+-----+
3  | variable_name | value |
4  +-----+-----+
5  | Rpl_semi_sync_master_clients | 1 | --显示处于半同步的slave节点个数
6  | Rpl_semi_sync_master_net_avg_wait_time | 0 |
7  | Rpl_semi_sync_master_net_wait_time | 0 |
8  | Rpl_semi_sync_master_net_waits | 1 |
9  | Rpl_semi_sync_master_no_times | 0 |
10 | Rpl_semi_sync_master_no_tx | 0 | --未成功发送到slave的事务数或不成功提交的数量
11 | Rpl_semi_sync_master_status | ON | --表示主库半同步模式处于打开状态
12 | Rpl_semi_sync_master_timefunc_failures | 0 |
13 | Rpl_semi_sync_master_tx_avg_wait_time | 2535 |
14 | Rpl_semi_sync_master_tx_wait_time | 5071 |
15 | Rpl_semi_sync_master_tx_waits | 1 |
16 | Rpl_semi_sync_master_wait_pos_backtraverse | 0 |
17 | Rpl_semi_sync_master_wait_sessions | 0 |
18 | Rpl_semi_sync_master_yes_tx | 1 | --已成功发送到slave的事务数或确认成功提交的数量
19 +-----+-----+
20 14 rows in set (0.00 sec)
21 mysql> use hello;
22 Database changed
23 mysql> create table t1(id int,name varchar(10));
24 query OK, 0 rows affected (0.05 sec)
25
26 mysql> insert into t1 values (1,'hero');

```

```
27 | query OK, 1 row affected (0.01 sec)
```

从服务

```
1 | mysql> show status like 'rpl_semi_sync%';
2 | +-----+-----+
3 | | variable_name | value |
4 | +-----+-----+
5 | | Rpl_semi_sync_slave_status | ON | --是否处于半同步状态
6 | +-----+-----+
7 | 1 row in set (0.00 sec)
```

4) 半同步复制测试

场景：验证slave挂了后，同步模式的变化：

1、将Slave主机停止slave的io线程

```
1 | mysql> stop slave io_thread;
2 | query OK, 0 rows affected (0.01 sec)
```

2、Master中执行插入语句，等待了30S

```
1 | mysql> insert into t1 values (2,'benson');
2 | query OK, 1 row affected (10.01 sec)
```

3、检查Master上的同步模式变化：

```
1 | mysql> show status like 'rpl_semi_sync%';
2 | +-----+-----+
3 | | variable_name | value |
4 | +-----+-----+
5 | | Rpl_semi_sync_master_clients | 1 |
6 | | Rpl_semi_sync_master_net_avg_wait_time | 0 |
7 | | Rpl_semi_sync_master_net_wait_time | 0 |
8 | | Rpl_semi_sync_master_net_waits | 3 |
9 | | Rpl_semi_sync_master_no_times | 1 |
10 | | Rpl_semi_sync_master_no_tx | 1 | 有一个事务没有同步到slave
11 | | Rpl_semi_sync_master_status | OFF | 变成了off，表示为异步模式
12 | | Rpl_semi_sync_master_timefunc_failures | 0 |
13 | | Rpl_semi_sync_master_tx_avg_wait_time | 2535 |
14 | | Rpl_semi_sync_master_tx_wait_time | 5071 |
15 | | Rpl_semi_sync_master_tx_waits | 2 |
16 | | Rpl_semi_sync_master_wait_pos_backtraverse | 0 |
17 | | Rpl_semi_sync_master_wait_sessions | 0 |
18 | | Rpl_semi_sync_master_yes_tx | 2 |
19 | +-----+-----+
20 | 14 rows in set (0.00 sec)
```

4、启动Slave

```
1 mysql> start slave;
2 Query OK, 0 rows affected (0.00 sec)
```

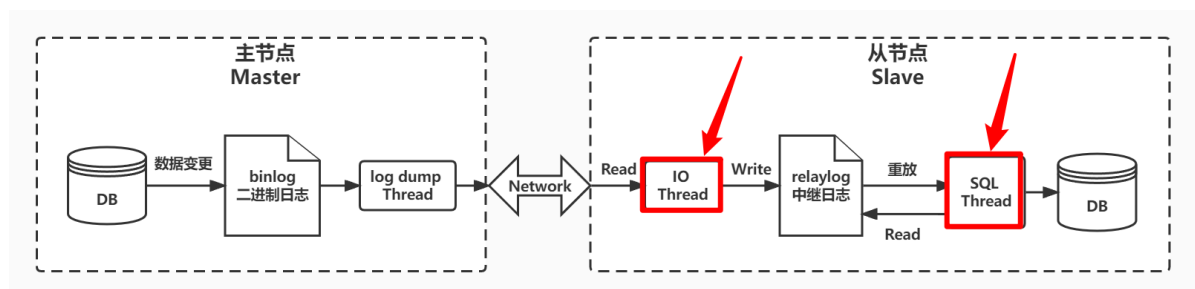
5、再次检查Master上的同步模式变化:

```
1 mysql> show status like 'rpl_semi_sync%';
2 +-----+-----+
3 | Variable_name | Value |
4 +-----+-----+
5 | Rpl_semi_sync_master_clients | 1 |
6 | Rpl_semi_sync_master_net_avg_wait_time | 0 |
7 | Rpl_semi_sync_master_net_wait_time | 0 |
8 | Rpl_semi_sync_master_net_waits | 4 |
9 | Rpl_semi_sync_master_no_times | 1 |
10 | Rpl_semi_sync_master_no_tx | 1 |
11 | Rpl_semi_sync_master_status | ON |
12 | Rpl_semi_sync_master_timefunc_failures | 0 |
13 | Rpl_semi_sync_master_tx_avg_wait_time | 2535 |
14 | Rpl_semi_sync_master_tx_wait_time | 5071 |
15 | Rpl_semi_sync_master_tx_waits | 2 |
16 | Rpl_semi_sync_master_wait_pos_backtraverse | 0 |
17 | Rpl_semi_sync_master_wait_sessions | 0 |
18 | Rpl_semi_sync_master_yes_tx | 2 |
19 +-----+-----+
20 14 rows in set (0.00 sec)
```

1.8 分析主从同步延迟的原因及解决办法

1.8.1 原因

在我们搭建的这个主从集群中，有一个比较隐藏的问题，就是这样的主从复制之间会有延迟。这在做了读写分离后，会更容易体现出来。即数据往主服务写，而读数据在从服务读。这时候这个主从复制延迟就有可能造成刚插入了数据但是查不到。当然，这在我们目前的这个集群中是很难出现的，但是在大型集群中会很容易出现。



出现这个问题的根本在于:

- Slave服务器IO Thread拉取binlog日志时，存在延迟
 - 面向业务的Master数据都是多线程并发写入的，而Slave是单个线程慢慢拉取binlog的，中间会有效率差!
- Slave服务器SQL Thread执行relaylog里面的SQL语句时，存在延迟
 - 同步到Slave后所有SQL必须都要在Slave里执行一遍，如果Master源源不断的写入，一旦有延迟产生，那么延迟加重的可能性就会越来越大

注意：5.6.3 之前的IO Thread仅有一个，5.6.3之后有多线程去读，速度会大幅提升，所以主从延迟也会相对少一些

1.8.2 解决办法

实际上主从同步延迟根本没有什么根治的办法，只不过可以做一些缓解的措施：

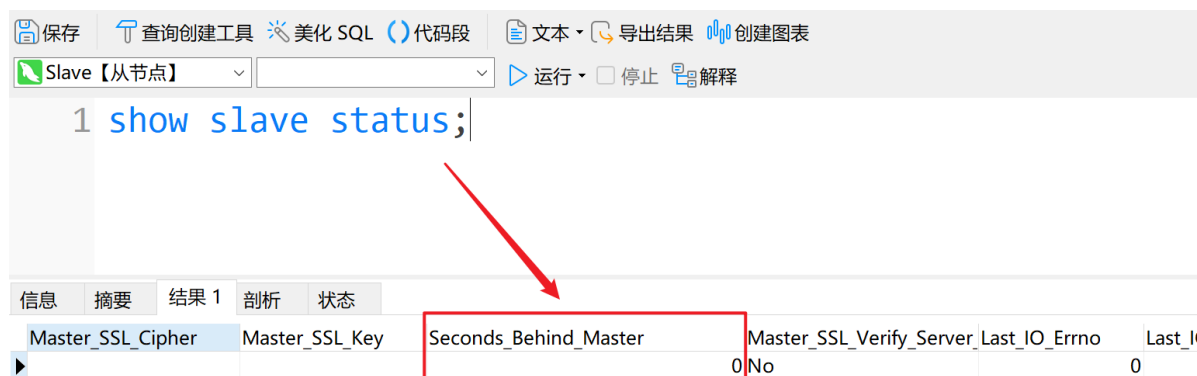
1. Master负责更新对安全性的要求比从服务器高，Slave负责分散读则不需要这么高的数据安全性，所以可以这么配置
 - Master: `sync_binlog=1`, `innodb_flush_log_at_trx_commit=1`,
 - Slave: `sync_binlog=0`, `innodb_flush_log_at_trx_commit=0`，提升查询的效率，无需保证安全性
2. 增加Slave数量分散读的压力降低Slave的负载
3. 重写代码，插入一条数据以后，尽量不要马上去查数据，插入数据直接去更新，不要查询。如果确实存在这样的业务，必须插入数据后马上查询到，对这个查询设置直连主库
4. 开启并行复制：让Slave用多线程并行复制 `binlog` 数据
5. 等等

1.8.3 如何判断主从延迟？

MySQL提供了从服务器状态命令，可以通过 `show slave status` 进行查看，比如可以看看 **Seconds_Behind_Master** 参数的值来判断，是否有发生主从延时。

其值有这么几种：

- NULL：表示IO Thread或是SQL Thread有任何一个发生故障，也就是该线程的Running状态是No，而非Yes
- 0：该值为零，表示主从复制状态正常



The screenshot shows a MySQL client window with the command `show slave status;` entered. The output is displayed in a table with the following columns: Master_SSL_Cipher, Master_SSL_Key, Seconds_Behind_Master, Master_SSL_Verify_Server, Last_IO_Errno, and Last_I. The value for Seconds_Behind_Master is 0, indicating that the slave is not behind the master.

Master_SSL_Cipher	Master_SSL_Key	Seconds_Behind_Master	Master_SSL_Verify_Server	Last_IO_Errno	Last_I
		0	No		0

1.9 全库同步与部分同步

之前提到，我们目前配置的主从同步是针对全库配置的，而实际环境中，一般并不需要针对全库做备份，而只需要对一些特别重要的库或者表来进行同步。

那如何针对库和表做同步配置呢？

- 首先在Master端：在my.cnf中，可以通过以下这些属性指定需要针对哪些库或者哪些表记录binlog

```
1 #需要同步的二进制数据库名
2 binlog-do-db=hello
3 binlog-do-db=mycat
4 #只保留7天的二进制日志，以防磁盘被日志占满(可选)
5 expire-logs-days=7
6 #不备份的数据库
7 binlog-ignore-db=information_schema
8 binlog-ignore-db=performance_schema
9 binlog-ignore-db=sys
```

- 然后在Slave端：在my.cnf中，需要配置备份库与主服务的库的对应关系。

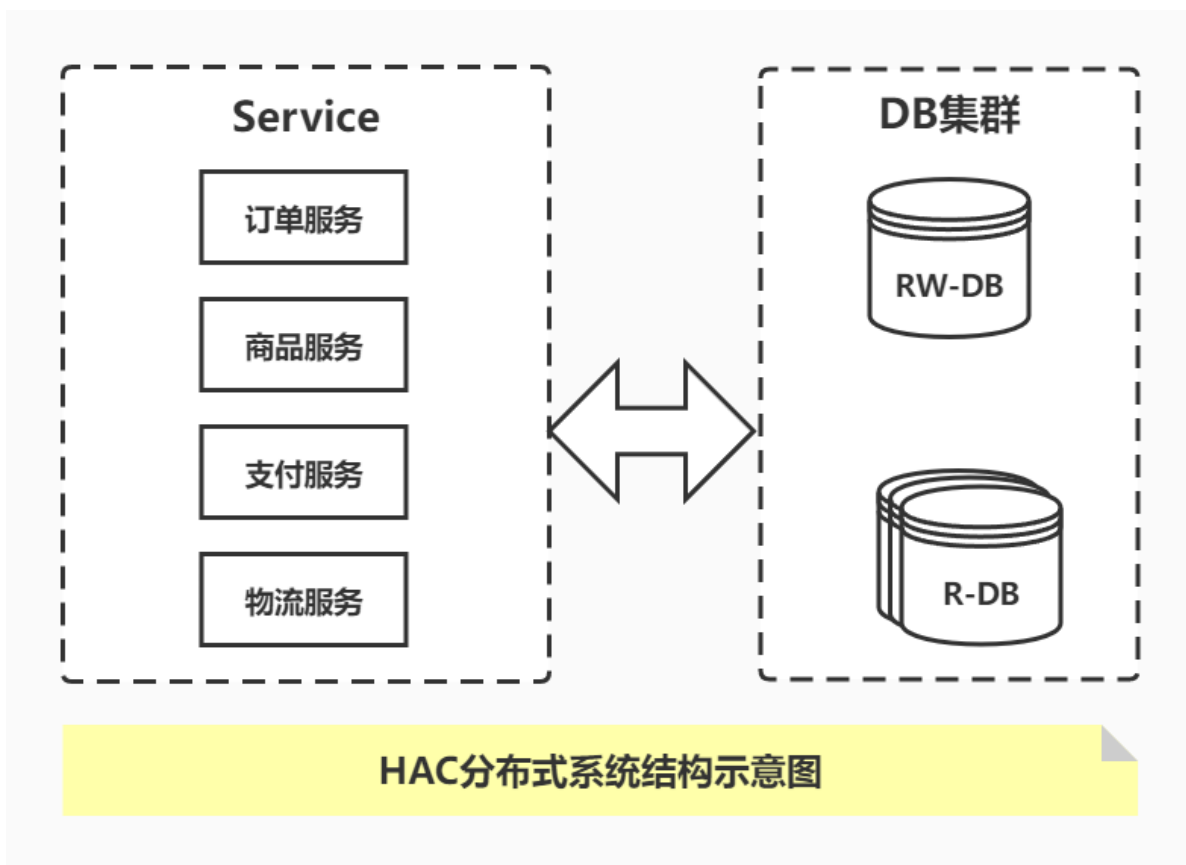
```
1 #如果salve库名称与master库名相同，使用本配置
2 replicate-do-db = hello
3 #如果master库名[hello]与salve库名[hello01]不同，使用以下配置[需要做映射]
4 replicate-rewrite-db = hello -> hello01
5 #如果不是要全部同步[默认全部同步]，则指定需要同步的表
6 replicate-wild-do-table=hello01.t1
7 replicate-wild-do-table=hello01.tab_user
```

配置完成了之后，在show master status指令中，就可以看到Binlog_Do_DB和Binlog_Ignore_DB两个参数的作用。

注意：MySQL主从复制，只会保证主机对外提供服务，而从机是不对外提供服务的，只是在后台为主机进行备份。如果需要从库对外提供服务则需要进行读写分离。

2. 集群搭建之读写分离

2.1 读写分离的理解



HAC: High Availability Cluster高可用集群

3. 高可用集群解决方案：基于主从复制