MySQL 高可用篇 分享

分享大纲

什么是高可用？支撑高可用的能力有哪些？

MySQL高可用的前提，复制

复制是指复制什么？什么是binlog？binlog的作用？

复制的原理

复制的类型：异步、半同步、并行

公司采用哪种复制方式？

MySQL高可用框架的种类及优缺点：自身高可用、旁路高可用

MySQL旁路高可用框架有哪些？MHA、MGR、RDB

不同高可用框架实现原理是什么？有什么优缺点？

公司的高可用架构是怎么样的（Orchestrator+HaControl）？

能够应对哪些实际场景？处理哪些问题？（异常监控报警、异常自动切换、多机房容灾切换）

（可选）MGR的分布式算法原理：paxos

（可选）由multi paxos到raft协议

（可选）异地多活实现原理

什么是高可用

高可用性-维基百科

高可用性(high availability)，指系统无中断地执行其功能的能力，高可用性通常通过**提高系统的容错能力**来实现。

其度量方式，是根据系统正常运转的时间，与系统总运作时间的比较。计算公式为:

$$A = \frac{\mathrm{MTBF}}{\mathrm{MTBF} + \mathrm{MTTR}}$$

MTTR，平均修复时间（Mean Time To Repair）。 即出现故障后修复故障的平均时间。 **MTTR**越小，表示故障时间越短，**可用性**也就越**高**。

MTBF，平均无故障时间（Mean Time Between Failures）。 即两次故障之间正常运行的平均时间。 **MTBF**越大，表明越不容易出故障，**可用性**自然**高。**

在线系统和执行关键任务的系统通常要求其可用性要达到5个9标准(99.999%)。

可用性	年故障时间
99.9999%	32秒
99.999%	5分15秒
99.99%	52分34秒
99.9%	8小时46分
99%	3天15小时36分

高可用的能力

- 集群内高可用（当集群内的master节点挂了该怎么办？）
- 多机房容灾（当一个机房由于断电，全挂了，怎么办？）
- 自动处理故障（公司内MySQL有5w+的实例，平均每天有40个实例出现异常，难道需要DBA逐一手动处理吗？）
- 处理过载保护（避免实例出现满负荷，导致雪崩）

MySQL Binlog

binlog（binary log，二进制日志)，记录了对MySQL数据库执行的**更新操作(Insert、Update、Delete)**，但是不包括select、show这类操作。

Binlog事件（Binlog的存储内容）

总的来说，binlog中存储的内容称之为二进制事件(binary log event)，简称事件。我们的每一个数据库更新操作(Insert、Update、Delete等)，都会对应的一个事件。

从大的方面来说，binlog主要分为2种格式：

- **Statement模式：**binlog中记录的就是我们执行的SQL；
- **Row模式：**binlog记录的是每一行记录的每个字段变化前后得到值。

Statement模式

mysql5.0及之前的版本只支持基于语句的复制，也称之为逻辑复制，也就是binary log文件中，直接记录的就是数据更新对应的sql。

假设有名为test库中有一张user表，如下：

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

现在，我们往user表中插入一条数据

```
insert into user(name) values("tianbowen");
```

之后，可以使用"**show binlog events**" 语法查看binary log中的内容，如下：

```
mysql> show binlog events in 'mysql-bin.000004';
+------------------+-----+----------------+-----------+-------------+-----------------------------------------------------------+
| Log_name         | Pos | Event_type     | Server_id | End_log_pos | Info                                                      |
+------------------+-----+----------------+-----------+-------------+-----------------------------------------------------------+
| mysql-bin.000004 |   4 | Format_desc    |         1 |         123 | Server ver: 5.7.10-3-log, Binlog ver: 4                   |
| mysql-bin.000004 | 123 | Previous_gtids |         1 |         154 |                                                           |
| mysql-bin.000004 | 154 | Anonymous_Gtid |         1 |         219 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                      |
| mysql-bin.000004 | 219 | Query          |         1 |         298 | BEGIN                                                     |
| mysql-bin.000004 | 298 | Intvar         |         1 |         330 | INSERT_ID=1                                               |
| mysql-bin.000004 | 330 | Query          |         1 |         446 | use `test`; insert into user(name) values("tianbowen")    |
| mysql-bin.000004 | 446 | Xid            |         1 |         477 | COMMIT /* xid=91 */                                       |
+------------------+-----+----------------+-----------+-------------+-----------------------------------------------------------+
```

红框中的Event，是我们执行上面Insert语句产生的4个Event。下面进行详细的说明：

(划重点)首先，需要说明的是，每个事务都是以Query Event作为开始，其INFO列内容为"BEGIN"，以Xid Event表示结束，其INFO列内容为COMMIT。即使对于单条更新SQL我们没有开启事务，Mysql也会默认的帮我们开启事务。因此在上面的红色框中，尽管我们只是执行了一个INSERT语句，没有开启事务，但是Mysql默认帮我们开启了事务，所以第一个Event是Query Event，最后一个是Xid Event。

接着，是一个**Intvar Event**，因为我们的Insert语句插入的表中，主键是自增的(AUTO_INCREMENT)列，Mysql首先会自增一个值，这就是Intvar Event的作用，这里我们看到INFO列的值为INSERT_ID=1，也就是说，这次的自增主键id为1。需要注意的是，这个事件，只会在Statement模式下出现。

然后，还是一个Query Event，这里记录的就是我们插入的SQL。这也体现了Statement模式的作用，就是记录我们执行的SQL。

Row模式

mysql5.1开始支持基于行的复制，这种方式记录的某条sql影响的所有行记录**变更前**和**变更后**的值。Row模式下主要有以下10个事件：

```
TABLE_MAP_EVENT

DELETE_ROWS_EVENTv0     UPDATE_ROWS_EVENTv0     WRITE_ROWS_EVENTv0

DELETE_ROWS_EVENTv1     UPDATE_ROWS_EVENTv1     WRITE_ROWS_EVENTv1

DELETE_ROWS_EVENTv2     UPDATE_ROWS_EVENTv2     WRITE_ROWS_EVENTv2
```

很直观的，我们看到了INSERT、DELETE、UPDATE操作都有3个版本(v0、v1、v2)，v0和v1已经过时，我们只需要关注V2版本。

此外，还有一个**TABLE_MAP_EVENT**，这个event我们需要特别关注，可以理解其作用就是记录了INSERT、DELETE、UPDATE操作的表结构。

下面，我们通过案例演示，ROW模式是如何记录变更前后记录的值，而不是记录SQL。这里只演示UPDATE，INSERT和DELETE也是类似。

在前面的操作步骤中，我们已经插入了2条记录，如下：

```
mysql> select * from user;
+----+-------------+
| id | name        |
+----+-------------+
|  1 | tianbowen   |
|  2 | tianshouzhi |
+----+-------------+
```

现在需要从Statement模式切换到Row模式，重启Mysql之后，执行以下SQL更新这两条记录：

```
update user set name='wangxiaoxiao';
```

在binary log中，会把这2条记录变更前后的值都记录下来，以下是一个逻辑示意图：

```
          记录1变更前后的值
+------------+----+--------------+
|            | id | name         |
+------------+----+--------------+
|    before  |  1 | tianbowen    |
+------------+----+--------------+
|    after   |  1 | wangxiaoxiao |
+------------+----+--------------+

          记录2变更前后的值
+------------+----+--------------+
|            | id | name         |
+------------+----+--------------+
|    before  |  2 | tianshouzhi  |
+------------+----+--------------+
|    after   |  2 | wangxiaoxiao |
+------------+----+--------------+
```

该逻辑示意图显示了，在默认情况下，受到影响的记录行，每个字段变更前的和变更后的值，都会被记录下来，即使这个字段的值没有发生变化。

接着，我们还是通过"show binlog events"语法来验证：

```
mysql> show binlog events in 'mysql-bin.000005';
+------------------+-----+----------------+-----------+-------------+---------------------------------------+
| Log_name         | Pos | Event_type     | Server_id | End_log_pos | Info                                  |
+------------------+-----+----------------+-----------+-------------+---------------------------------------+
| mysql-bin.000005 |   4 | Format_desc    |         1 |         123 | Server ver: 5.7.10-3-log, Binlog ver: 4 |
| mysql-bin.000005 | 123 | Previous_gtids |         1 |         154 |                                       |
| mysql-bin.000005 | 154 | Anonymous_Gtid |         1 |         219 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'  |
| mysql-bin.000005 | 219 | Query          |         1 |         291 | BEGIN                                 |
| mysql-bin.000005 | 291 | Table_map      |         1 |         341 | table_id: 108 (test.user)             |
| mysql-bin.000005 | 341 | Update_rows    |         1 |         445 | table_id: 108 flags: STMT_END_F       |
| mysql-bin.000005 | 445 | Xid            |         1 |         476 | COMMIT /* xid=15 */                   |
+------------------+-----+----------------+-----------+-------------+---------------------------------------+
```

首先我们可以看到的是，在Row模式下，单条SQL依然会默认开启事务，通过Query Event(值为BEGIN)开始，以Xid Event结束。

接着，我们看到了一个Table_map 事件，就是前面提到的TABLE_MAP_EVENT，在INFO列，我们可以看到其记录table_id为108，操作的是test库中user表。

最后，是一个Update_rows事件，然而其INFO，并没有像Statement模式那样，显示一条SQL，我们无法直接看到其变更前后的值是什么。此时，由于存储的都是二进制内容，直接vim无法查看，我们需要借助另外一个工具**mysqlbinlog**来查看其内容。如下：

```
[root@39 mysql]# mysqlbinlog --start-position=291 --stop-position=445 --base64-output="decode-rows" -v  mysql-bin.000005
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=1*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 291
#190605 23:56:17 server id 1  end_log_pos 341 CRC32 0x5a58791a  Table_map: `test`.`user` mapped to number 108
# at 341
#190605 23:56:17 server id 1  end_log_pos 445 CRC32 0x63436578  Update_rows: table id 108 flags: STMT_END_F
### UPDATE `test`.`user`
### WHERE
###   @1=1
###   @2='tianbowen'
### SET
###   @1=1
###   @2='wangxiaoxiao'
### UPDATE `test`.`user`
### WHERE
###   @1=2
###   @2='tianshouzhi'
### SET
###   @1=2
###   @2='wangxiaoxiao'
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
DELIMITER ;
# End of log file
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;
```

截图中显示了2个event，第一个红色框就是Table_map事件，第二个是Update_rows事件。

在第二个红色框架中，显示了两个Update sql，这是只是mysqlbinlog工具为了方便我们查看，反解成SQL而已。我们看到了WHERE以及SET子句中，并没有直接列出字段名，而是以**@1**、**@2**这样的表示字段位于数据库表中的顺序。**事实上，这里显示的内容，WHERE部分就是每个字段修改前的值，而SET部分，则是每个字段修改后的值，**也就是变更前后的值都会记录。

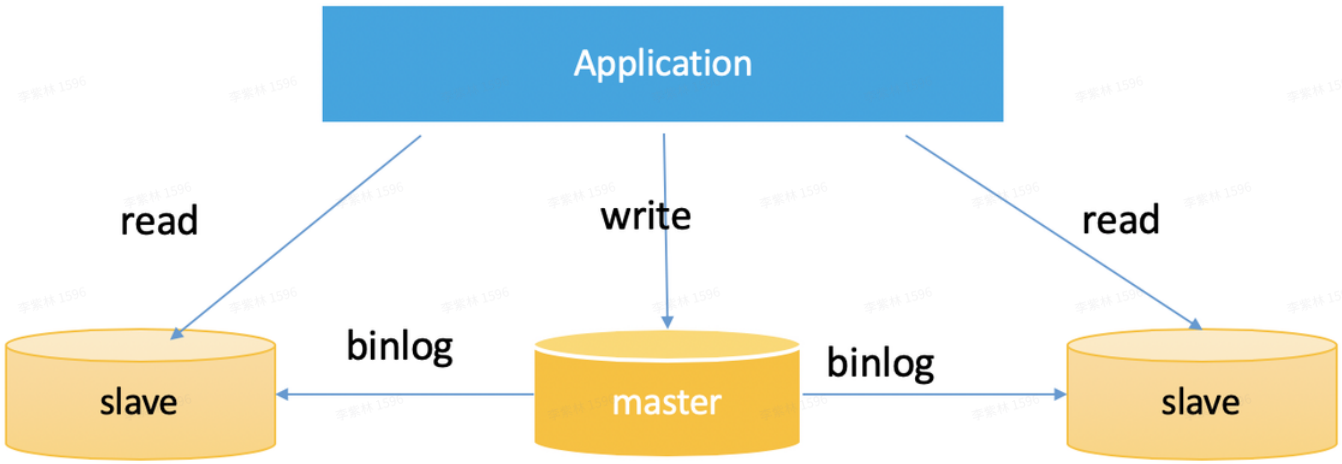
这里我们思考一下mysqlbinlog工具的工作原理，其可以将二进制数据反解成SQL进行展示。那么，**如果我们可以自己解析binlog，就可以做数据恢复，这并非是什么难事**。例如用户误删除的数据，执行的是DETELE语句，由于Row模式下会记录变更之前的字段的值，我们可以将其反解成一个INSERT语句，重新插入，从而实现数据恢复。

Binlog的应用场景

- 读写分离
- 数据恢复
- 最终一致性
- 异地多活

读写分离

最典型的场景就是Mysql主从之间通过binlog复制来实现横向扩展，来实现读写分离。如下图所示：



在这种场景下：

- 有一个主库Master，所有的更新操作都在master上进行
- 同时会有多个Slave，每个Slave都连接到Master上，获取binlog在本地回放，实现**数据复制**。

数据恢复

一些同学可能有误删除数据库记录的经历，或者因为误操作导致数据库存在大量脏数据的情况。

这些都可以通过反解binlog来完成数据恢复

最终一致性

在实际开发中，我们经常会遇到一些需求，在数据库操作成功后，需要进行一些其他操作，如：发送一条消息到MQ中、更新缓存或者更新搜索引擎中的索引等。

如何保证数据库操作与这些行为的一致性，就成为一个难题。以数据库与redis缓存的一致性为例：操作数据库成功了，可能会更新redis失败；反之亦然。很难保证二者的完全一致。

遇到这种看似无解的问题，最好的办法是换一种思路去解决它：不要同时去更新数据库和其他组件，只是简单的更新数据库即可。

如果数据库操作成功，必然会产生binlog。之后，我们通过一个组件，来模拟的mysql的slave，拉取并解析binlog中的信息。通过解析binlog的信息，去异步的更新缓存、索引或者发送MQ消息，**保证数据库与其他组件中数据的最终一致**。

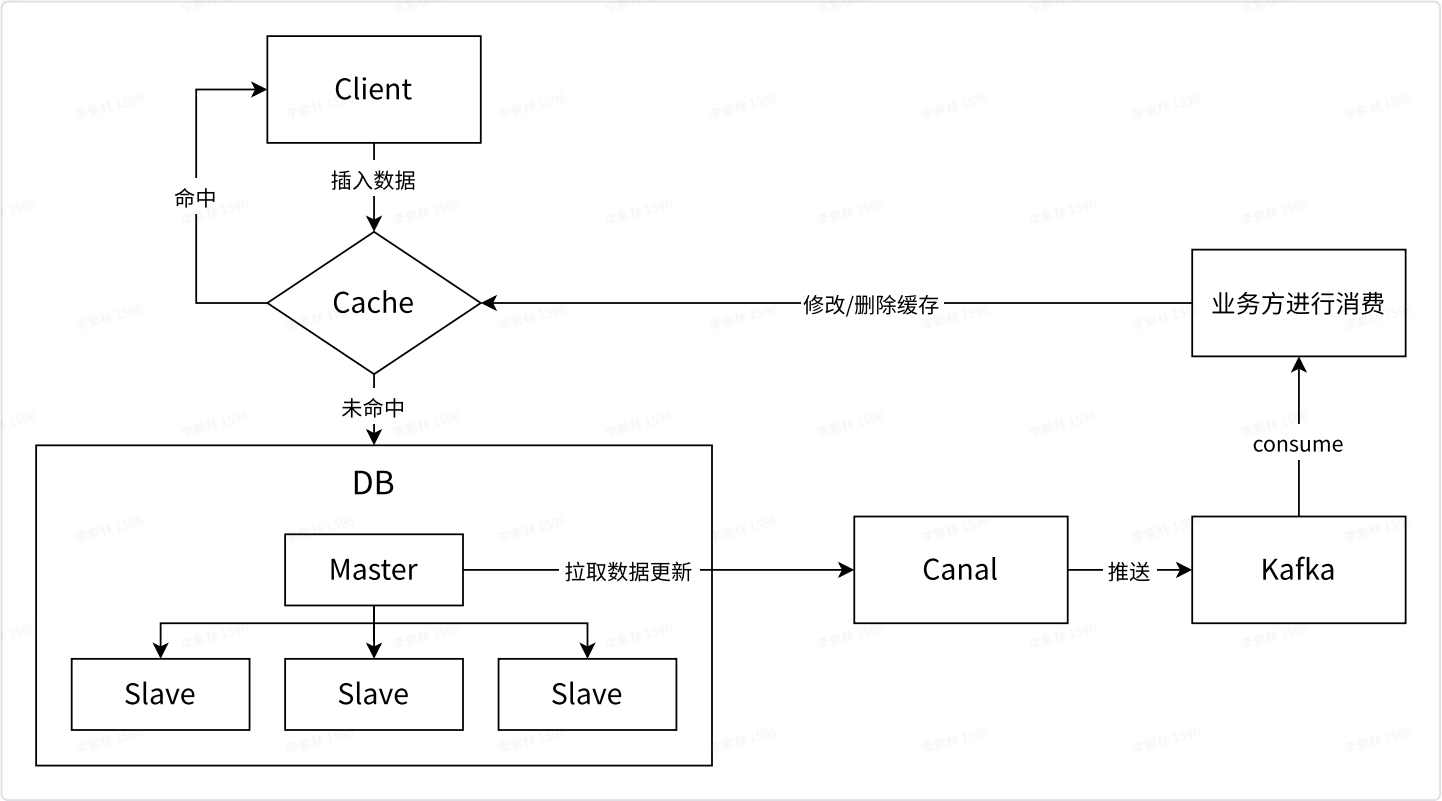
在这里，我们将模拟slave的组件，统一称之为**binlog同步组件**。你并不需要自己编写这样的一个组件，已经有很多开源的实现，例如linkedin的databus，阿里巴巴的canal，美团点评的puma等。

用canal来实现缓存更新

> canal，译意为水道/管道/沟渠，主要用途是基于 MySQL 数据库增量日志解析，提供增量数据订阅和消费。

canal的工作原理：就是把自己伪装成MySQL slave，模拟MySQL slave的交互协议向MySQL Mater发送dump协议，MySQL mater收到canal发送过来的dump请求，开始推送binary log给canal，然后canal解析binary log，再发送到存储目的地，比如MySQL，Kafka，Elastic Search等等。

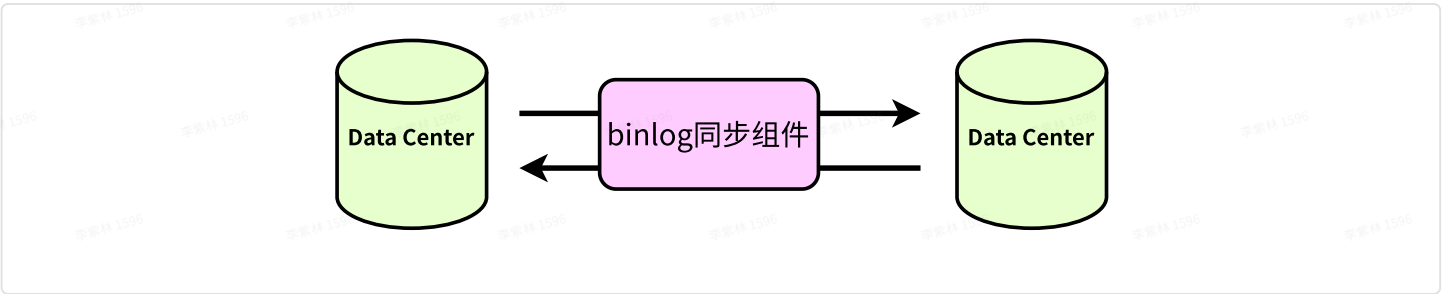
用canal来实现缓存更新的流程：



说明：大体流程就是 canal 充当一个 mysql 的从服务器，从 master 拉取 binlog 变化，将更新内容推送至 kafka 中，然后客户端启动消费者订阅主题，根据数据变化执行对应的业务逻辑。

异地多活

一个更大的应用场景，异地多活场景下，跨数据中心之间的数据同步。这种场景的下，多个数据中心都需要写入数据，并且往对方同步。以下是一个简化的示意图：

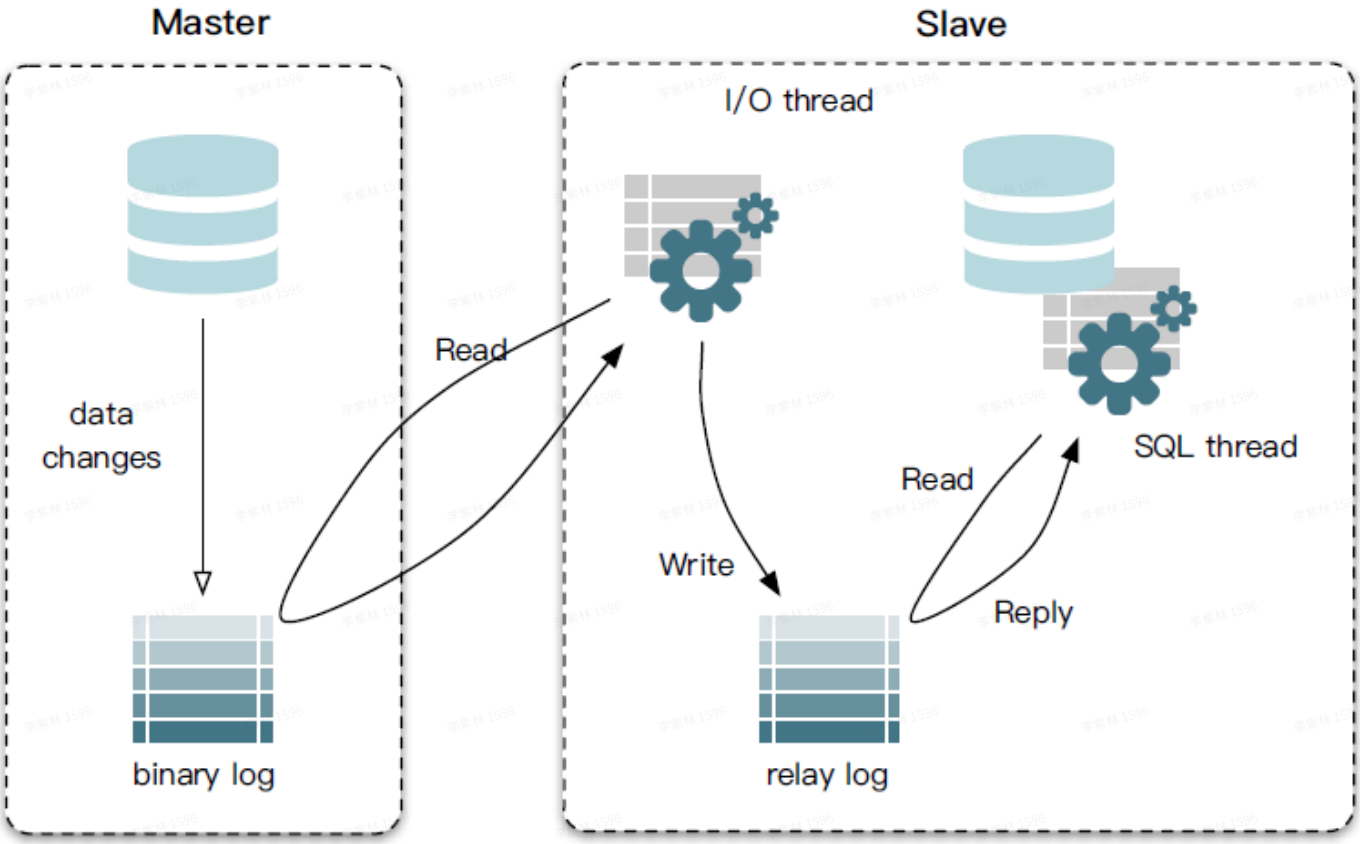


其中，binlog同步组件可以是上文提到的canal

MySQL复制

mysql主从复制

mysql主从复制的流程如下图所示：



主要分为3个步骤：

- **第一步：**master在每次准备提交事务完成数据更新前，将更改记录写入到二进制日志(binary log)中（这些记录叫做二进制日志事件，binary log event，简称event)
- **第二步：**slave启动一个I/O线程来读取主库上binary log中的事件，并记录到slave自己的中继日志(relay log)中。
- **第三步：**slave还会启动一个SQL线程，该线程从relay log中读取事件并在备库执行，从而实现备库数据的更新。

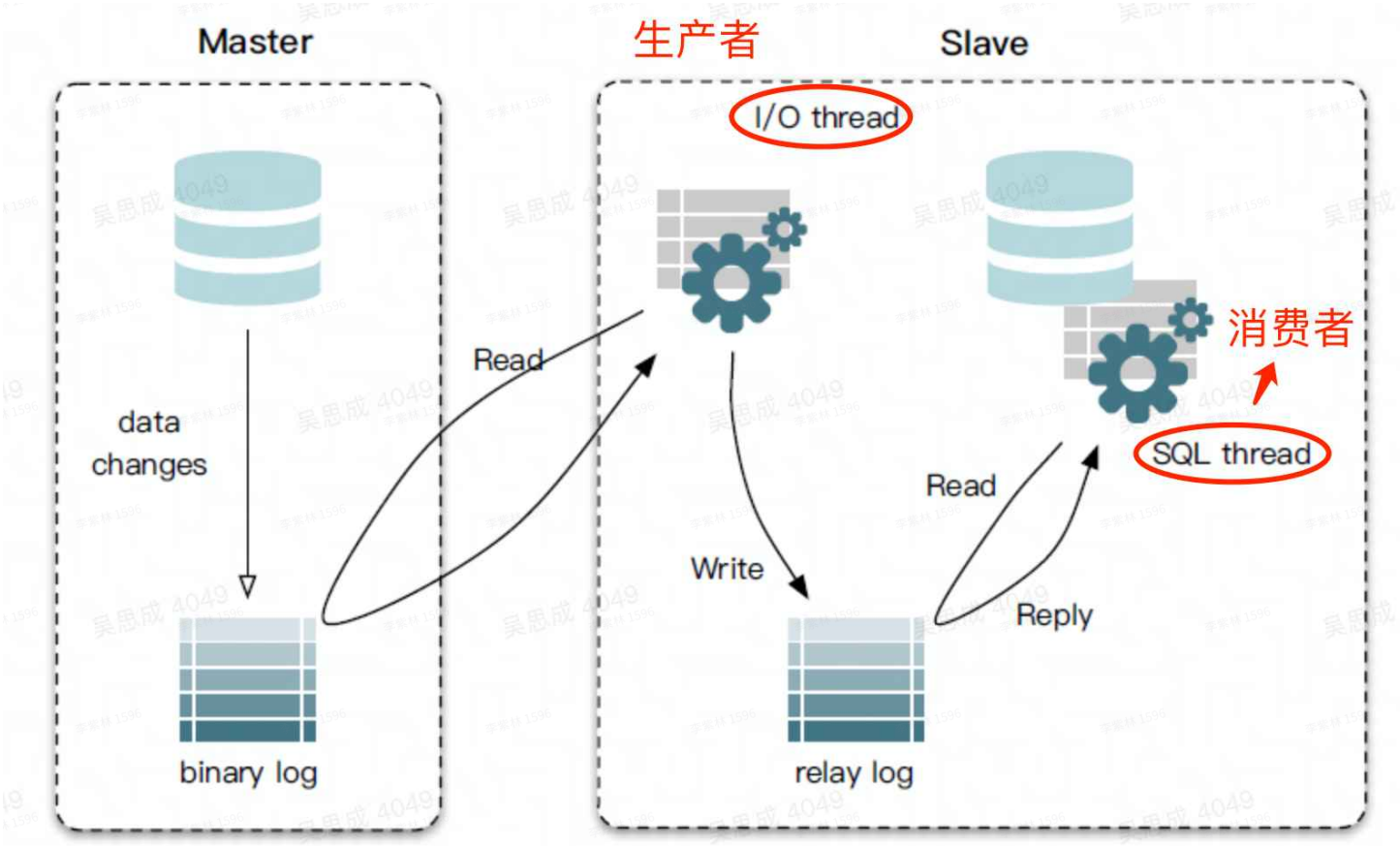
复制的两大难题：

- 读写分离-主从数据延迟。Slave回放relay日志需要时间，存在延迟
- 数据不一致。异步复制时，若主机宕机，切主时会造成数据丢失

并行复制

并行复制的原理：加大消费者并发。如下图，如果我们在重放relay日志时，能够并发执行，则重放速度能够提升。

举例，假设relay log中有1000条insert语句，那么此时开10个SQL线程并发执行insert命令，远比只开1个SQL线程执行insert命令快。



那么我们可以随意并发吗?

不行，因为事务之间可能存在冲突，因此在并发复制时，需要考虑如何避免事务冲突。目前比较常见的做法是:

- 基于schema的并行复制
- 基于logical_clock的并行复制
- 基于writeset的并行复制

基于schema并行复制

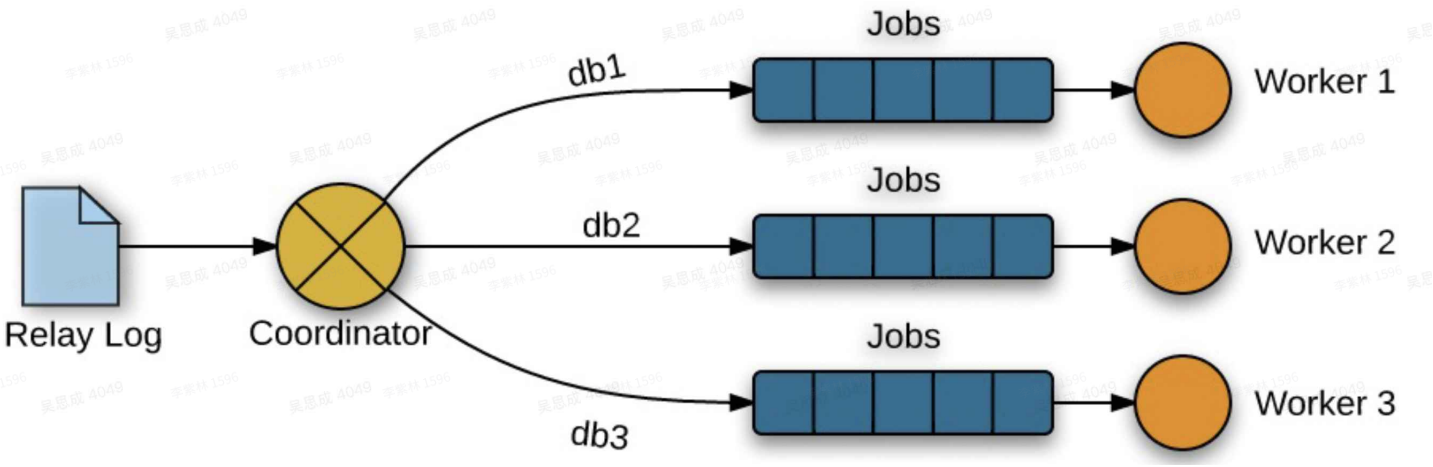
MySQL中的schema是啥？简单理解，就是database，一个schema下会存放多张表。

```
mysql>
mysql>
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| bits               |
| byteflow           |
| byteflow_data      |
| iamdb              |
| mars_base          |
| mars_devops        |
| mars_gecko         |
| mars_gray_release  |
| mars_h5            |
| mars_mpaas         |
| mars_plugin_read   |
| monitor            |
| mysql              |
| performance_schema |
| saveu              |
| sys                |
| test               |
| tob_component      |
| tt_rds_meta        |
+--------------------+
20 rows in set (0.00 sec)
```

原理：

如果binlog events操作的是不同schema的对象，不是DDL，且操作的对象没有对其他schema的foreign key关联，则这些binlog events在slave上做重放的时候可以并行。（这些events不会冲突）

流程：



1. Coordinator将event**按db插入**各Work线程的任务队列，Work从队列里取出event执行；
2. 同一个事务内的event都发给同一个worker，保证事务的一致性；
3. 分发关系由包含db信息的event(如Table_map)决定，其它event按决定好的关系进行分发；

优点：

- 实现逻辑比较简单
- schema较多且写入分散的情况下，并行复制效率高

缺点：

- 按照DBName进行分发的，分发粒度太大，如果只有一个DB时，就退化成单线程模式。

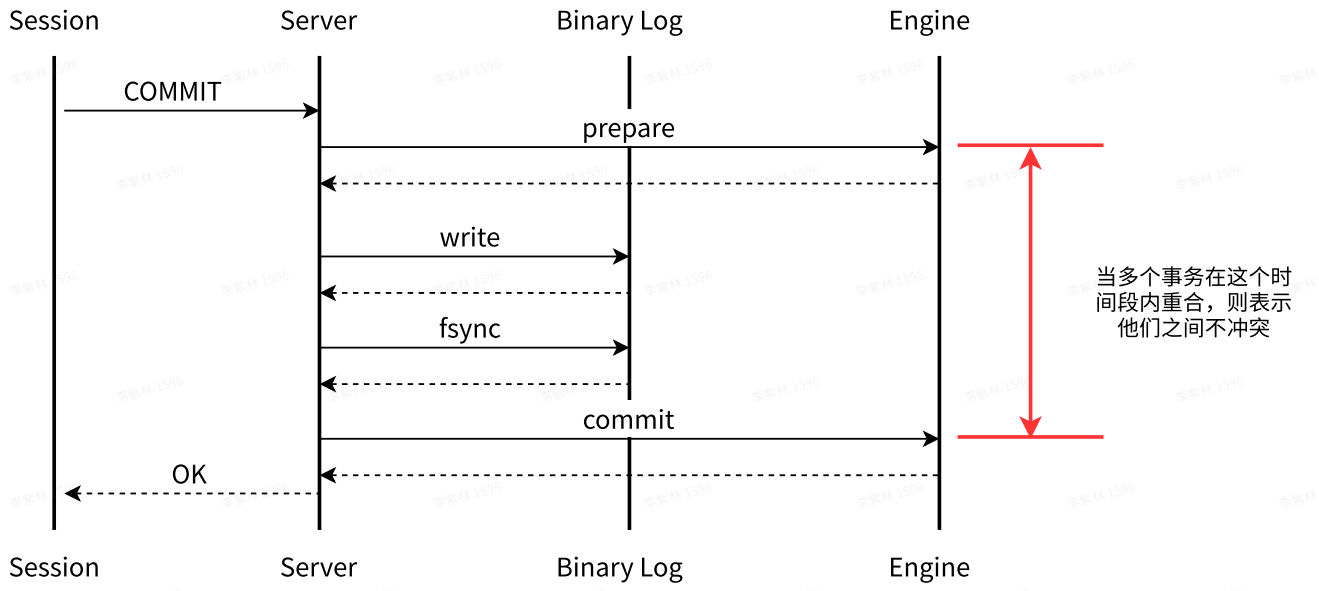
- DDL语句或跨库（Schema）的语句不能并行执行。e.g. 操作的对象对其他schema有foreign key关联

思考：是否可以支持表级别的并行复制?

基于Logical_clock并行复制

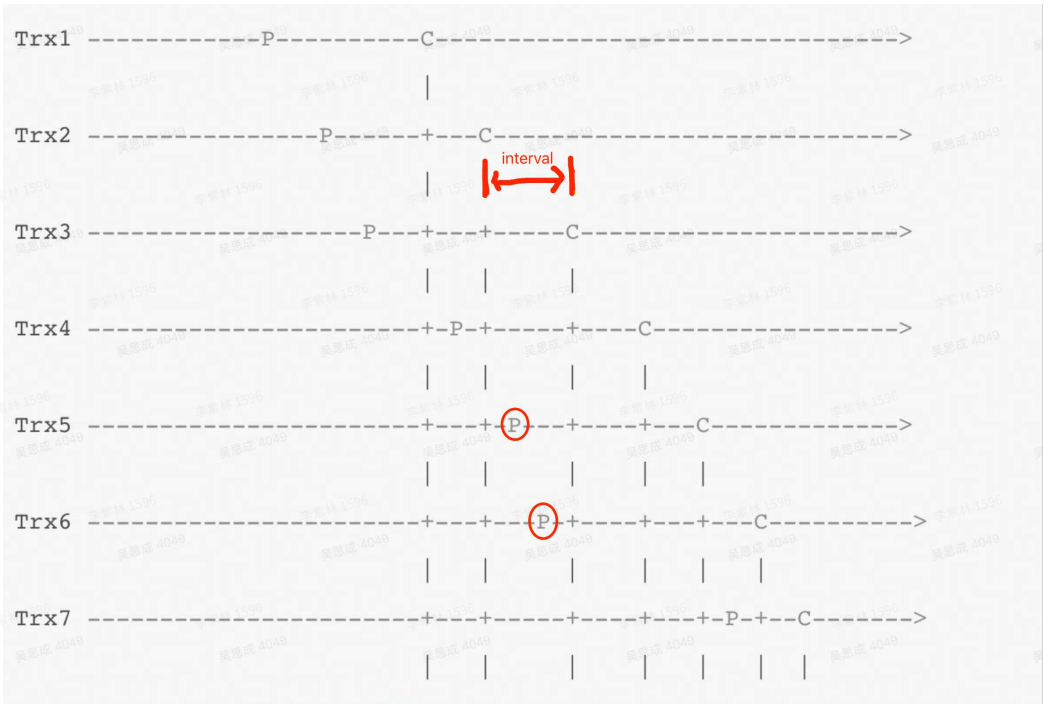
原理：

因为MySQL中写入是基于锁的并发控制，Master端同时处于prepare阶段且未提交的事务就不会存在锁冲突。简单理解就是，有冲突的事务都会通过锁来控制串行执行，因此同时处于prepare阶段的事务必然是不会冲突的事务（反证法），可以并行执行。



单个事务commit流程

commit-parent-base模式



算法简介：p是prepare阶段，c是commit，每行代表一个事务，commit和commit之间构成一个interval，那么所有**在同一个interval下进入prepare阶段的事务，都是不冲突的**。例如Trx5、Trx6都是在Trx2的commit和Trx3的commit之间的interval中进入的prepare阶段，因此Trx5、Trx6可以并行执行。

但是存在一种情况，即两个事务虽然不在同个interval下，但是他们俩持有的是不同的锁，也不会冲突。例如，Trx4和Trx5不在同个interval下，但是他们在一段时间下各自持有自己的锁，说明他们互不冲突，因此他们也可以并行执行，但是在commit-parent-based模式下却不能并行。所有有了改进版，lock-based模式。

lock-based模式

对于master节点：

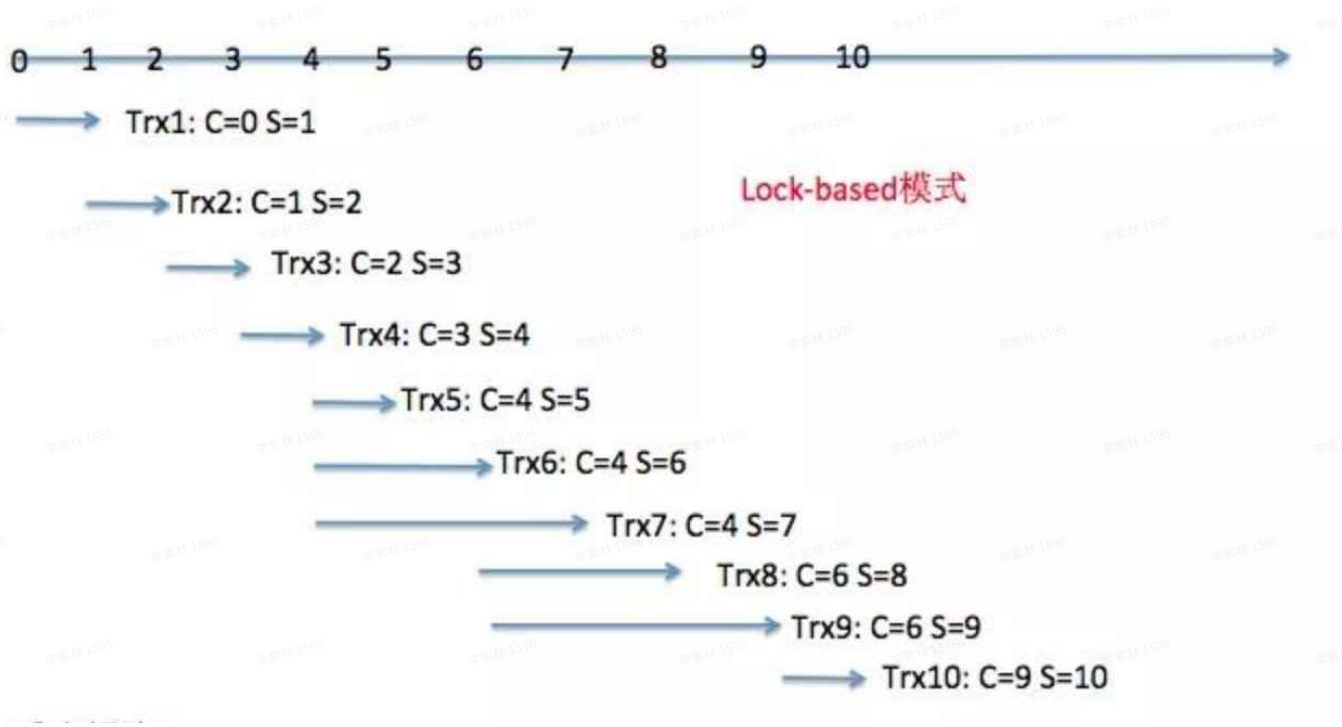
事务开始时获取当前最大的last_commits_timestamp（最近一次commit的时间点）；再维护一个全局计数器sequence_number作为时钟，每来一个新的事务，sequence_number加一。

每个事务进来时，都会去获取当前的last_commits_timestamp，以及获取一个sequence_number，那么（last_commits_timestamp，sequence_number）就是该事务的时间间隔(interval)。

对于slave节点：

从master节点读取binlog至slave的relay log之后，已经知道了每个事务的interval，也就是[last_commits_timestamp，sequence_number]，那么就可以通过下面的算法来判读事务间能否并行复制

算法中心思想：只要两个事务的interval有重叠，就说明这俩事务同时持有不同的锁，表示这两个事务没有冲突，意味着两个事务可以并行执行。



理解上需要注意的事:

- 这里的时间并不是我们日常生活中的时间，而是将每个sequence_number视为“时间点”；
- 每次事务发生commit的时候，就会获取查看当前的sequence_number是多少，从而作为commit的“时间点”；
- 而每次有新的事务进来的时候，都会去获取last_commited_timestamp（最近commit的时间点）；

算法流程：

从图中可以看出，[5,6,7]和[8,9]可以并行执行。

而实际上在slave节点回放relay log时，是逐个事务按顺序进行分发至worker去执行的，因此实际上并不是[5,6,7]并发执行完后才能到[8,9]并发执行的。而是每一个事务在分发前，判断该事务的last_commited_timestamp是否小于当前正在执行的commit时间点。

例如：

在trx5执行时，它的commit时间点是5

此时判断trx6能否执行：trx6的last_commit为4，4<[5]，可以执行，且trx6的commit时间点是6

然后判断trx7能否执行：trx7的last_commit为4，4<[5, 6]，可以执行，且trx7的commit时间点是7

然后判断trx8能否执行：trx8的last_commit为6，6>=[5, 6]，不可以执行，也就是说trx8需要等到trx5、trx6执行完成后才可以执行。

（trx5、trx6执行完成）

判断trx8能否执行：trx8的last_commit为6，6<[7]，可以执行，此时是[trx7,trx8]并行执行

基于writeset并行复制

事务并行执行的条件：事务之间没有冲突，则可以并行执行

算法中心思想：

1. 通过map去维护一段时间内的事务语句
2. 如果有一个新的事务过来，则会遍历map中的事务，判断新的事务是否和Map中的事务冲突，如果不冲突，则能够减少当前的last_commit_timestampt，【last_commit_timestampt，sequence_number】的间隔越大，则并发的可能性越大，从而提高并发效率。

PS：如何判断事务是否冲突，这里用到了基于主键的冲突检测(binlog_transaction_depandency_tracking = COMMIT_ORDERE|WRITESET|WRITESET_SESSION, 修改的row的主键或非空唯一键没有冲突，即可并行)

并行复制总结

模式	并行效率	适用范围	说明
schema			仅适用多database
logical_clock	中	大	
writeset	高	大	加强版logical_clock

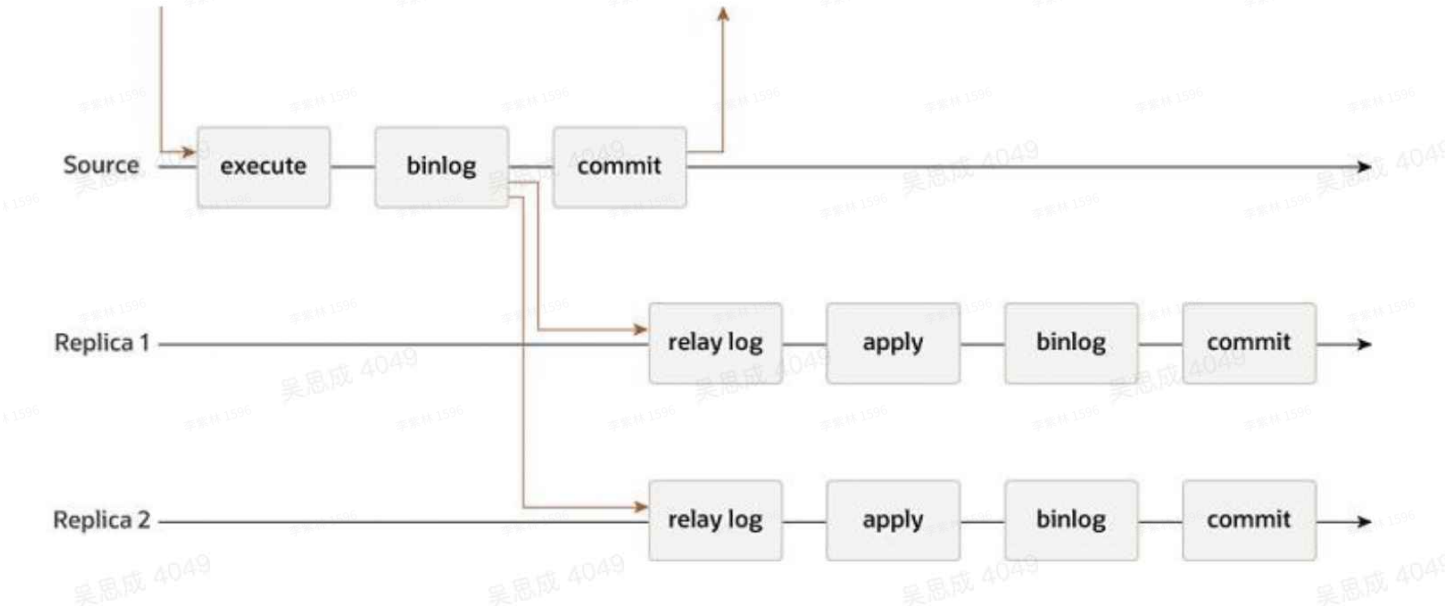
- 并行复制为提高从库回放效率，并没有改变复制机制
  - 天生存在一定的延迟
- 并行复制并不根本解决主从数据延迟问题
  - 从库本身查询负载高
  - 大事务写入，单个事务的回放需要比较长的时间

半同步复制

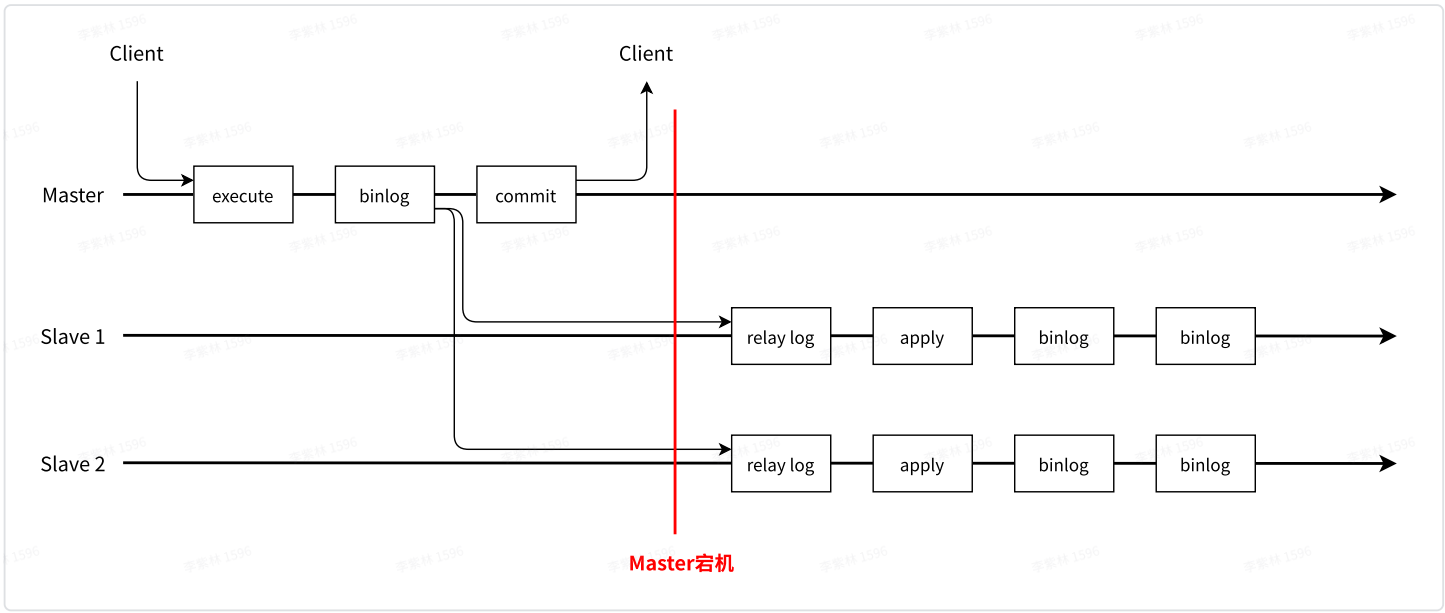
解决的问题：数据一致性，主机宕机切换时不会造成数据丢失。

异步复制

master写入binlog后，不去关注slave是否将binlog写入relay log，而是直接commit事务，与此同时slave异步的将master的binlog写入到relay log，并回放



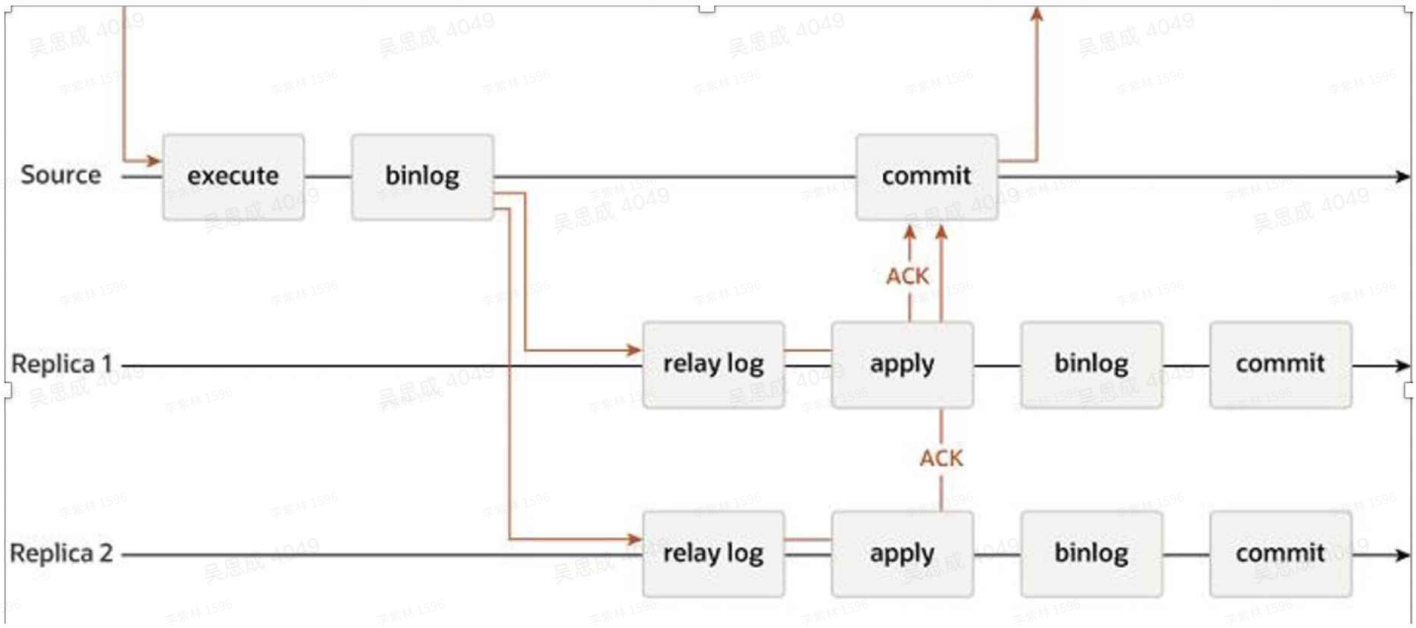
数据不同步的例子：



在红线位置，若Master节点宕机，则会自动触发切换Master操作，在备选的Slave节点中进行选主。但是如图所示，Slave节点没有将宕机前的数据同步，因此一旦被选取为Master节点，则会导致数据丢失，和宕机前的master节点数据不一致。

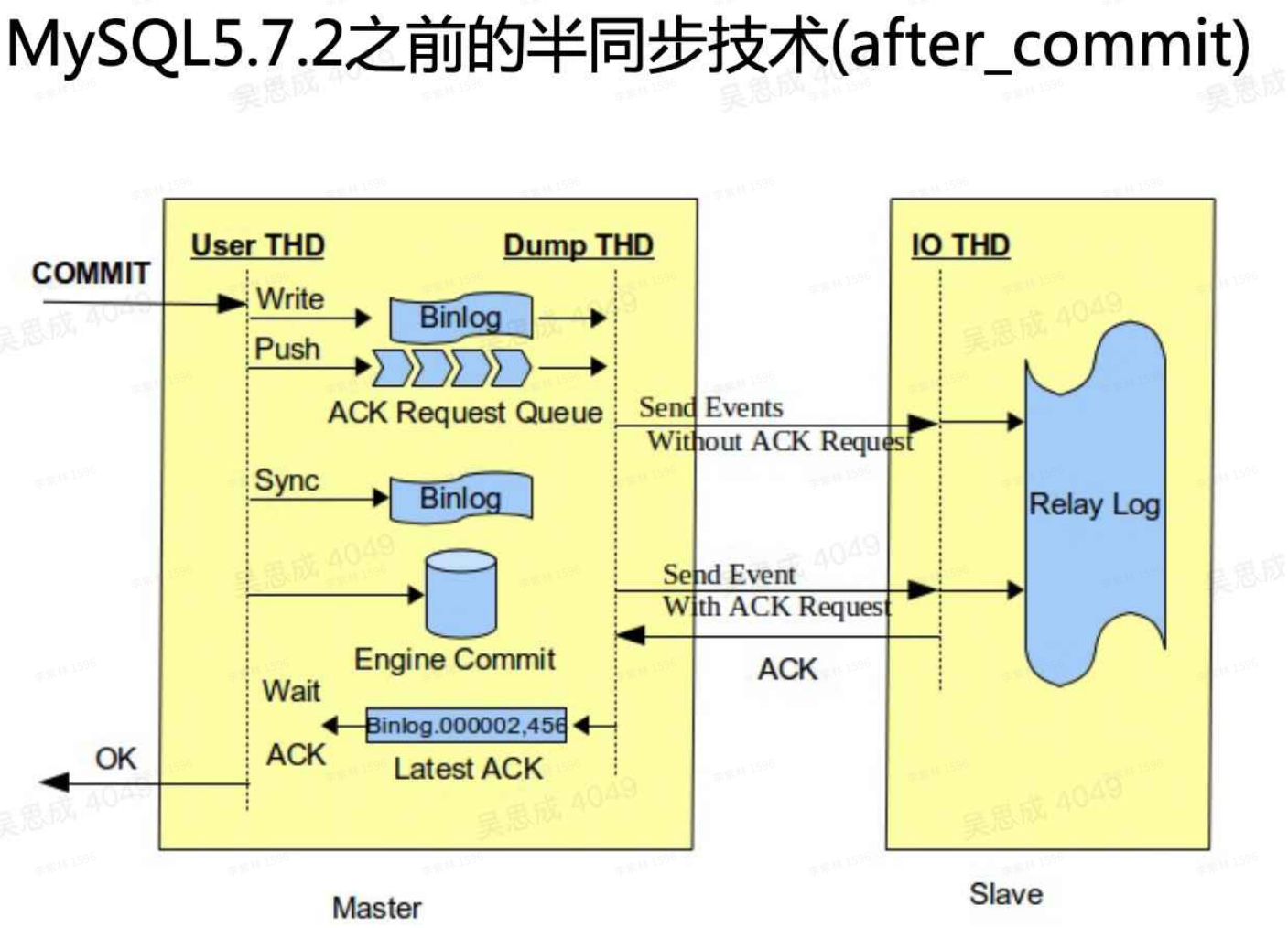
半同步复制

master写入binlog后，等待slave将binlog数据写入到relay log，接收到slave返回的ack之后，再完成事务的写入流程



after-commit模式

在master执行完engine commit之后才开始等待ack。

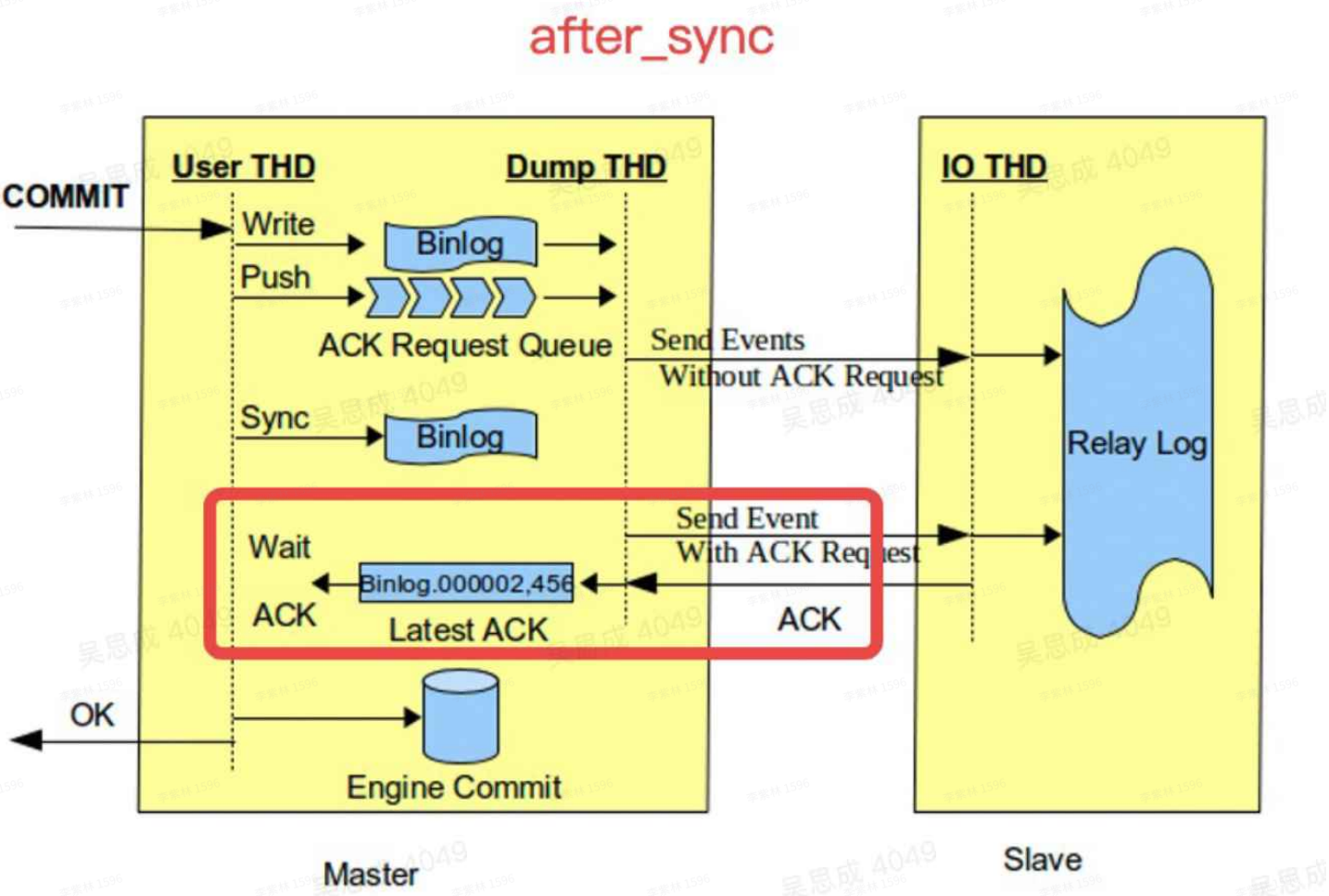


（engine commit意味着数据已写入磁盘了）

存在的问题：会出现幻读的现象。假设此时数据库有数据（1，2），如果在master执行完engine commit之后，此时master的数据是（1，2，3），而slave的数据还是（1，2），这时user1去读数据，读到的是（1，2，3），如果master宕机了，刚好来了个user2去读数据，那么只能去slave节点读数据，读到的是（1，2），则出现了幻读现象。

after-sync模式

master在收到ack之后再执行engine commit（被称为**半同步增强版**）

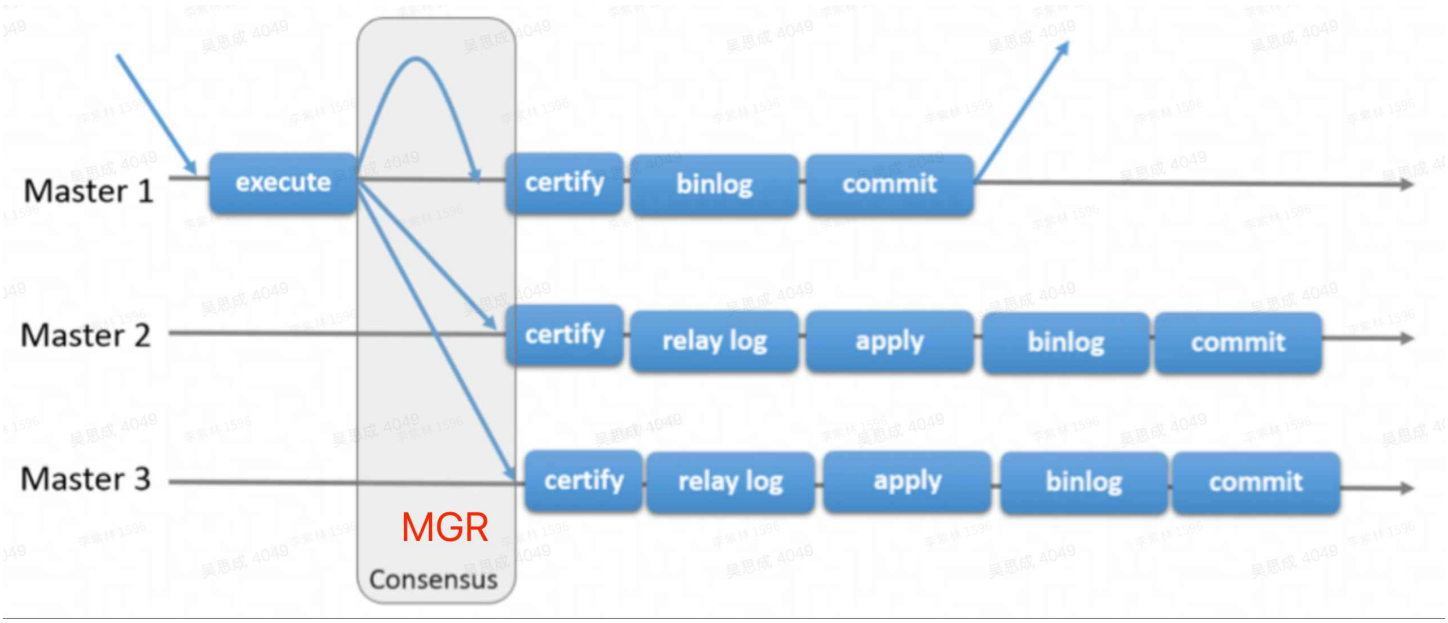


优点：能够保持数据一致性，如果在接收到ack之前master宕机了，那么master和slave的数据都是（1，2），则不会出现幻读

缺点：可能会导致重复写入。由于master需要等待slave的ack信号，这个等待的时机我们称为“卡主”，如果卡主的时间过长，导致master的数据并没有更新，有些业务场景下会去读取数据，发现数据并没更新后认为是事务提交失败，会重复写入新的数据，从而导致数据的重复写入。

Group Replication

使用MGR组件来保证数据一致性



- MGR组件使用分布式数据一致性协议Paxos来保证数据一致性
- 事务提交时用MGR插件将事务信息广播给各个节点，过半数节点确认可提交后写入节点才可提交成功
- 其他节点还是通过回放binlog同步数据

优点：

- 数据一致性更高
- 自动故障切换
- 多主模式，多节点写入

缺点：

- 写入性能损失20-30%
- 受网络影响大，网络抖动对集群稳定性影响大

半同步复制总结

数据一致性强度：MGR>增强半同步复制>半同步复制>异步复制

写入性能：MGR<增强半同步复制<半同步复制<异步复制

MySQL高可用

高可用组件的分类

旁路高可用组件

实例宕机后组件会去检测集群状态，进行拓扑调整和流量调整，包括vip漂移或者代理刷新等操作，对业务影响相对小一些。

集群自身高可用

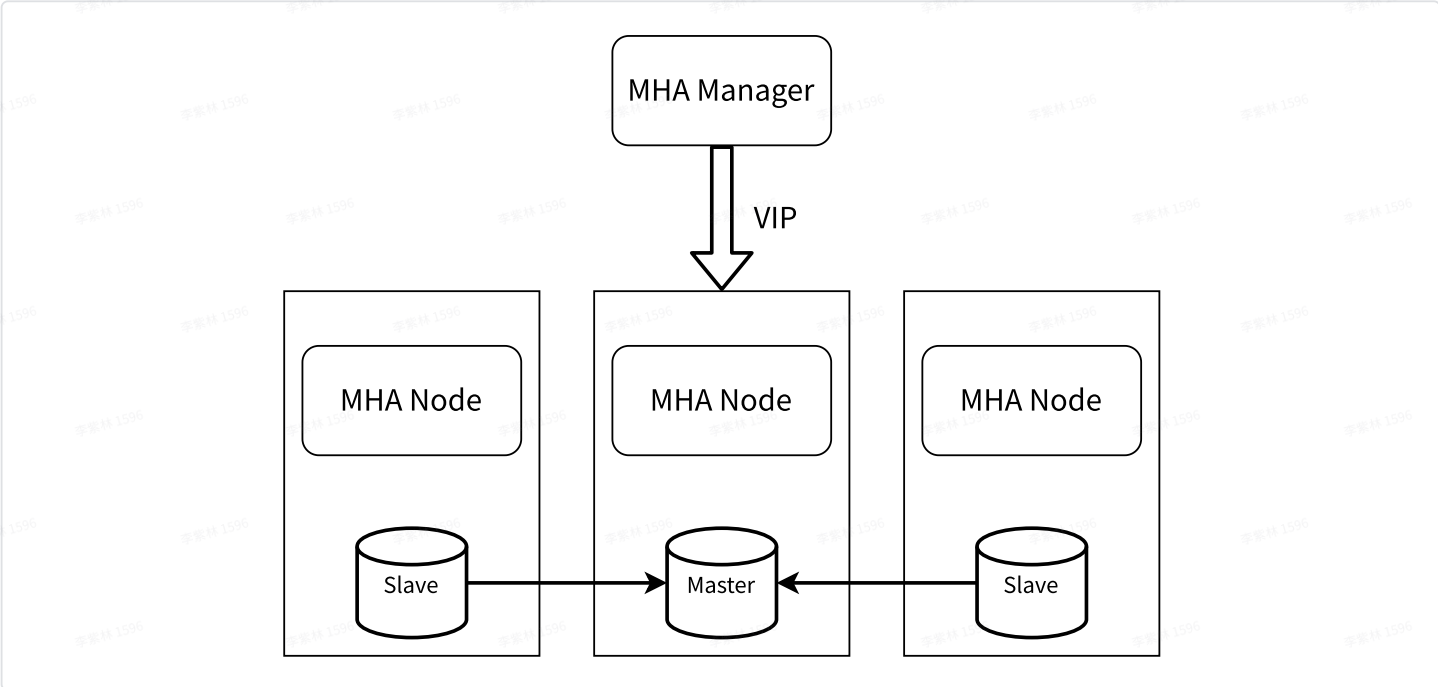
实例宕机后集群自动做选主和节点剔除的操作，但是需要业务去感知底层实例ip的变动，对业务影响较大。

常见高可用组件演进

MHA

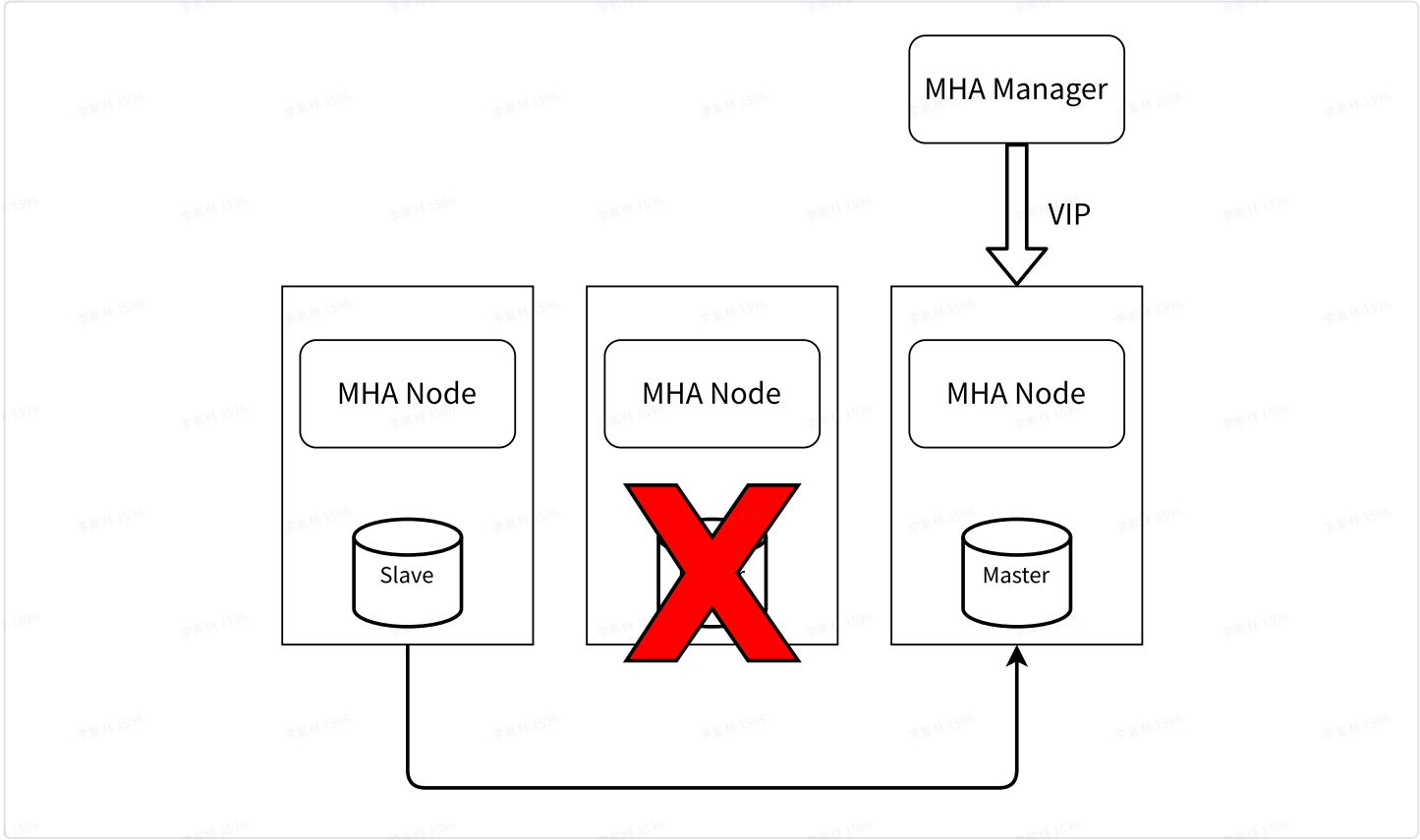
> 半同步复制确保了master的事务同步到了slave，但是依旧可能存在部分slave缺失binlog event，主库Failover的困难点就是无法识别slave是不是缺失了主库binlog event。

Master High Availability，它为 MySQL 主从复制架构提供了 automating master failover (自动化故障转移) 功能。MHA 在监控到 master 节点故障时，会提升其中**拥有最新数据的 slave 节点**成为新的master 节点，在此期间，MHA 会通过于从节点获取额外信息来避免一致性方面的问题。



MHA Manager：定时探测集群中的master结点，当master出现故障时，它可以自动将最新数据的slave提升为新的master，然后将所有其他的slave重新指向新的master，将VIP漂移到新master结点，整个故障转移过程对应用程序完全透明。

MHA Node：运行在每台MySQL服务器上，主要作用是收集binlog信息，当发生故障时，能够选取数据较完备的Slave节点切换为主节点



优点：

- 对现有架构侵入较小，只需要在机器上安装对应管理节点。
- 机器无硬件故障切主时会自动补齐binlog ，保证数据完整性。

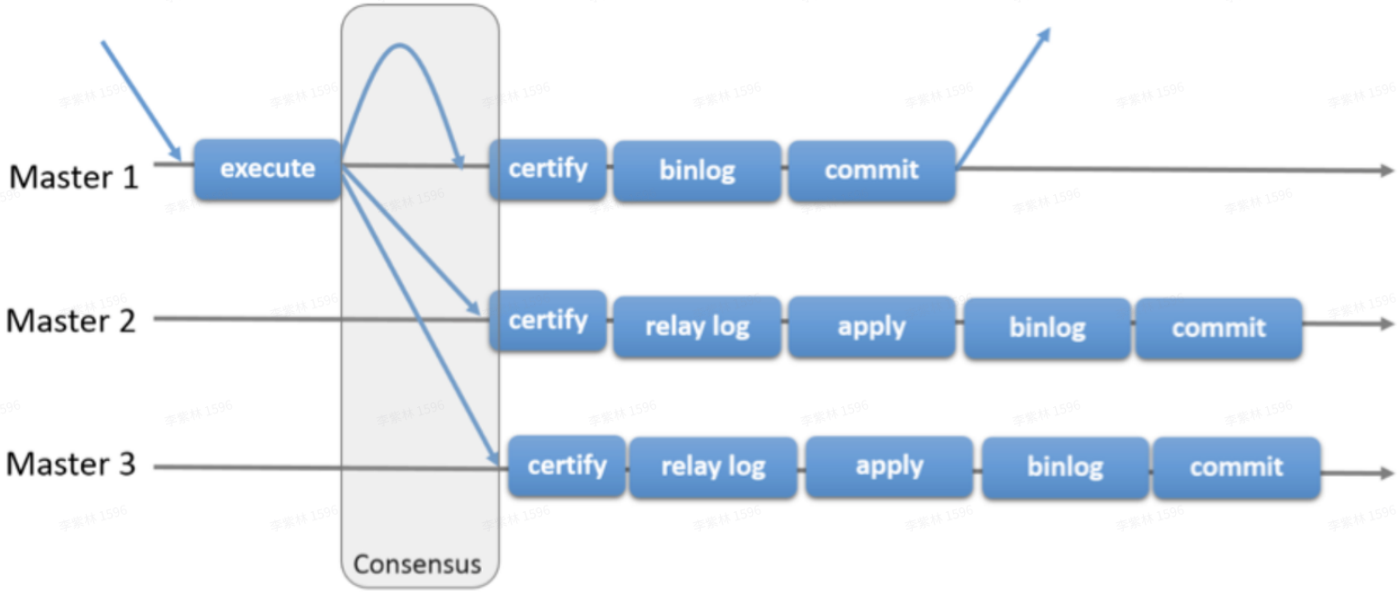
缺点：

- 管理节点在每个节点都需要安装，同时会引入vip, 管理维护会相对麻烦。
- MHA Manager和MHA Node的通信依赖ssh ，不符合公司安全要求。
- 依赖ssh，满足不了多机房的需求

MGR(MySQL Group Replication)

Group Replication由至少3个或更多个节点共同组成一个数据库集群，基于分布式一致性算法Paxos实现，事务的提交必须经过半数以上节点同意方可提交，在集群中每个节点上都维护一个数据库状态机，保证节点间事务的一致性。

MySQL Group Replication是建立在已有MySQL复制框架的基础之上，通过新增Group Replication Protocol协议及Paxos协议的实现，形成的整体高可用解决方案。与原有复制方式相比，主要增加了certify的概念，如下图所示：



certify模块主要负责检查事务是否允许提交，是否与其它事务存在冲突，如两个事务可能修改同一行数据。在单机系统中，两个事务的冲突可以通过封锁来避免。

但在多主模式下，不同节点间没有分布式锁，所以无法使用封锁来避免。为提高性能，Group Replication乐观地来对待不同事务间的冲突，乐观的认为多数事务在执行时是没有并发冲突的。事务分别在不同节点上执行，直到准备提交时才去判断事务之间是否存在冲突。

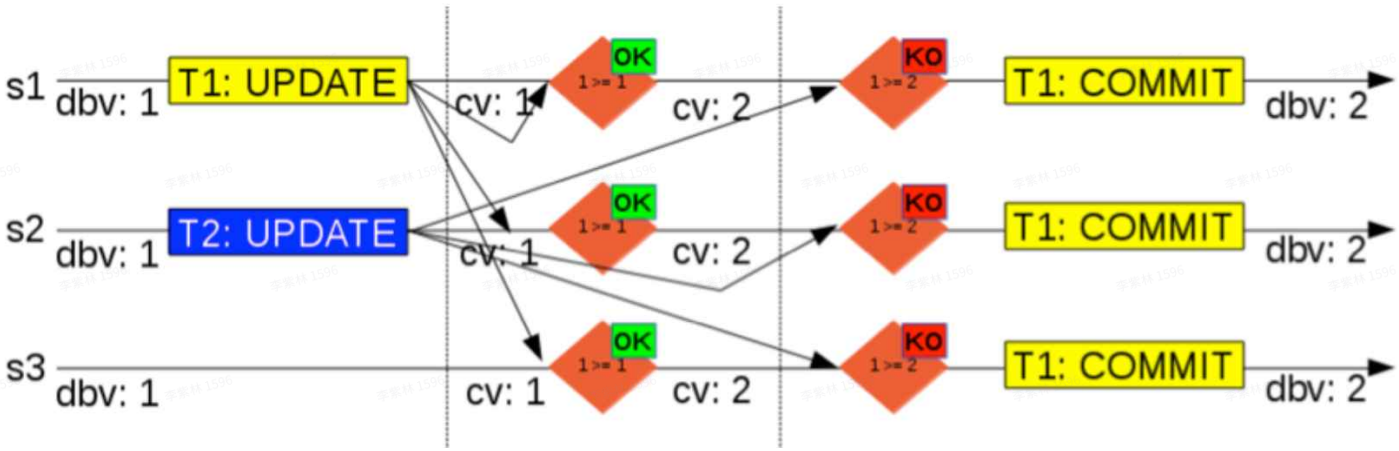
优点：

- 支持多主写入，无延迟复制，能保证数据强一致性
- 自动检测故障和集群容错

缺点：

- 对现有架构侵入较大
- 只支持innodb储存引擎，同时每张表都必须有主键
- 集群至少3个节点，至多接入9个节点
- 多主模式下，不支持在不同实例对同一行数据进行操作
- 对网络要求比较高，官方建议同机房部署

多主模式下事务冲突的判断



在节点s1上发起一个更新事务T1，几乎同时，在节点s2上也发起一个更新事务T2，当T1在s1本地完成更新后，准备提交之前，将其writeset及更新时的版本dbv=1(db version)发送给group；

同时T2在s2本地完成更新后，准备提交之前，将其writeset及更新时的版本dbv=1也发送给group。

此时需要注意的是，group组内的通讯是采用基于paxos协议的xcom来实现的，**它的一个特性就是消息是有序传送，每个节点接收到的消息顺序都是相同的**，并且至少保证半数以上节点收到才会认为消息发送成功。xcom的这些特性对于数据库状态机来说非常重要，是保证数据库状态机一致性的关键因素。

本例中我们假设先顺序发送T1事务的certification请求，各节点收到T1事务的certification请求，则发现当前版本cv=1(commit version)，而数据更新时的版本dbv=1，所以没有冲突，T1事务可以提交，并将当前版本cv修改为2；

之后马上又收到T2事务的certification请求，此时当前版本cv=2，而数据更新时的版本dbv=1，表示数据更新时更新的是一个旧版本，此事务与其它事务存在冲突，因此事务T2必须回滚。

xcom如何确保消息是有序传送的?

在多主情况下，每个主节点被视为paxos leader，和其余的节点构成一个paxos实例（instance），假设有四个写节点，则全局有四个paxos。在每一个paxos实例下，由于Leader只有一个（写节点），因此实例内的数据顺序是一致的。那么如何确保实例之间的顺序一致呢？这里xcom预设了一个实例间消息顺序的排序规则，全局的数据顺序由两个参数决定：

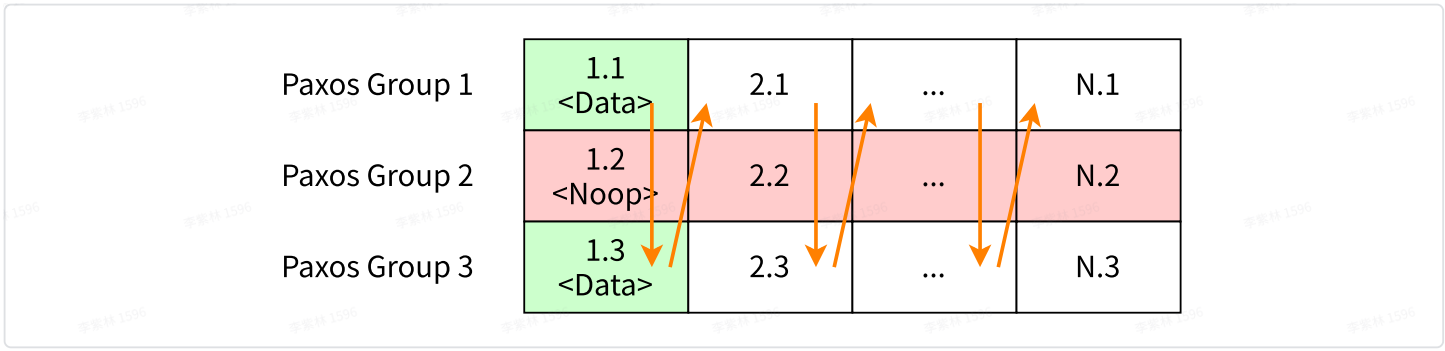
- 组内顺序号
- 组号(即Leader节点在集群中的序号)

所以组间的排序是事先确定的。<1,1>在<1,2>前，<1,2>在<1,3>,<1,3>在<2,1>前...。

Paxos Group 1	1.1	2.1	...	N.1
Paxos Group 2	1.2	2.2	...	N.2
Paxos Group 3	1.3	2.3	...	N.3

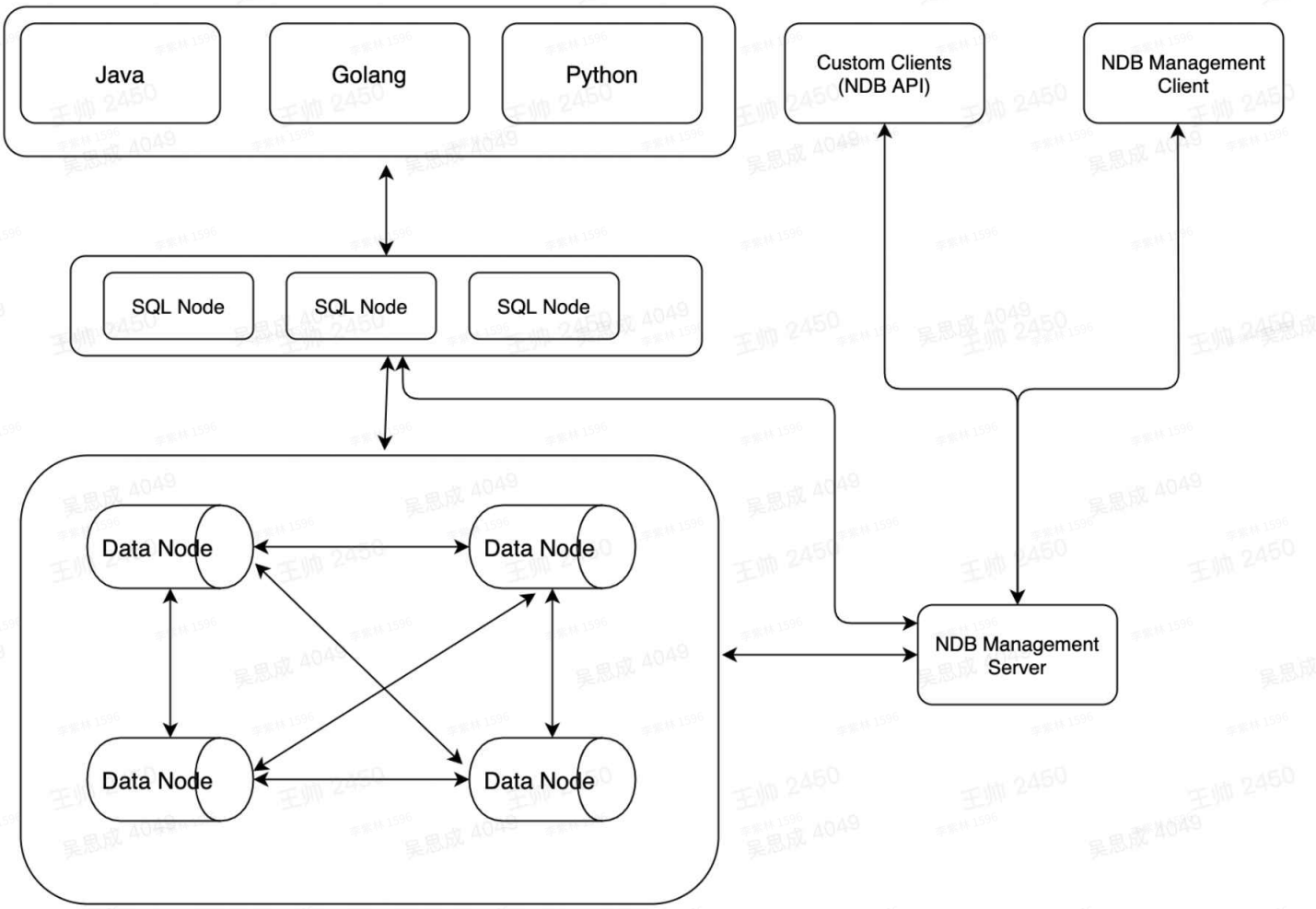
Xcom必须要按照顺序将的完成paxos过程的数据发送给应用。由于顺序是事先确定的，即顺序靠后的数据先完成paxos过程，也不能提前发送给应用。比如<1,3>可能比<1,2>先完成paxos过程，但是必须要等到<1,2>完成，并发送给应用后，才能将<1,3>的数据发送给应用。

那如果其中的一个节点很久没有数据，岂不是会导致后面其他Paxos组已经完成paxos过程的数据也一直没有办法发送给应用吗？是的，为了避免这个问题Mencius中加入了Noop操作。当一个节点发现比自己的顺序号靠后的数据已经完成了Paxos过程时(并且自己的这个位置上没有paxos操作)，就会广播一个Noop，告诉其他节点自己的这个顺序号可以跳过。



注意: Noop操作是直接通过一个消息发送出去的，不需要Paxos的过程。因为除了Leader其他节点不能在这个Paxos组内发送任何数据。

NDB



NDB cluster: 官方集群的部署方案，通过使用NDB存储引擎实时备份冗余数据，实现数据库的高可用性和数据一致性。

Manage 节点：负责整个Cluster 集群中各个节点的管理工作，包括集群的配置，启动，关闭各节点等。管理节点会获取整个Cluster 环境中各节点的状态和错误信息，并且将各Cluster 集群中各个节点的信息反馈给整个集群中其他的所有节点。

SQL 结点：主要负责实现一个数据库在存储层之上的所有事情，比如连接管理，query 优化和响应，cache 管理等等，只有存储层的工作交给了NDB 数据节点去处理了。

Data 结点：用于保存数据、索引，控制事务。插入的数据按照主键的哈希值分散到不同的节点组里面保存（每个节点组保存部分数据），另外每个节点组内，数据会复制到不同的数据节点上以实现冗余。

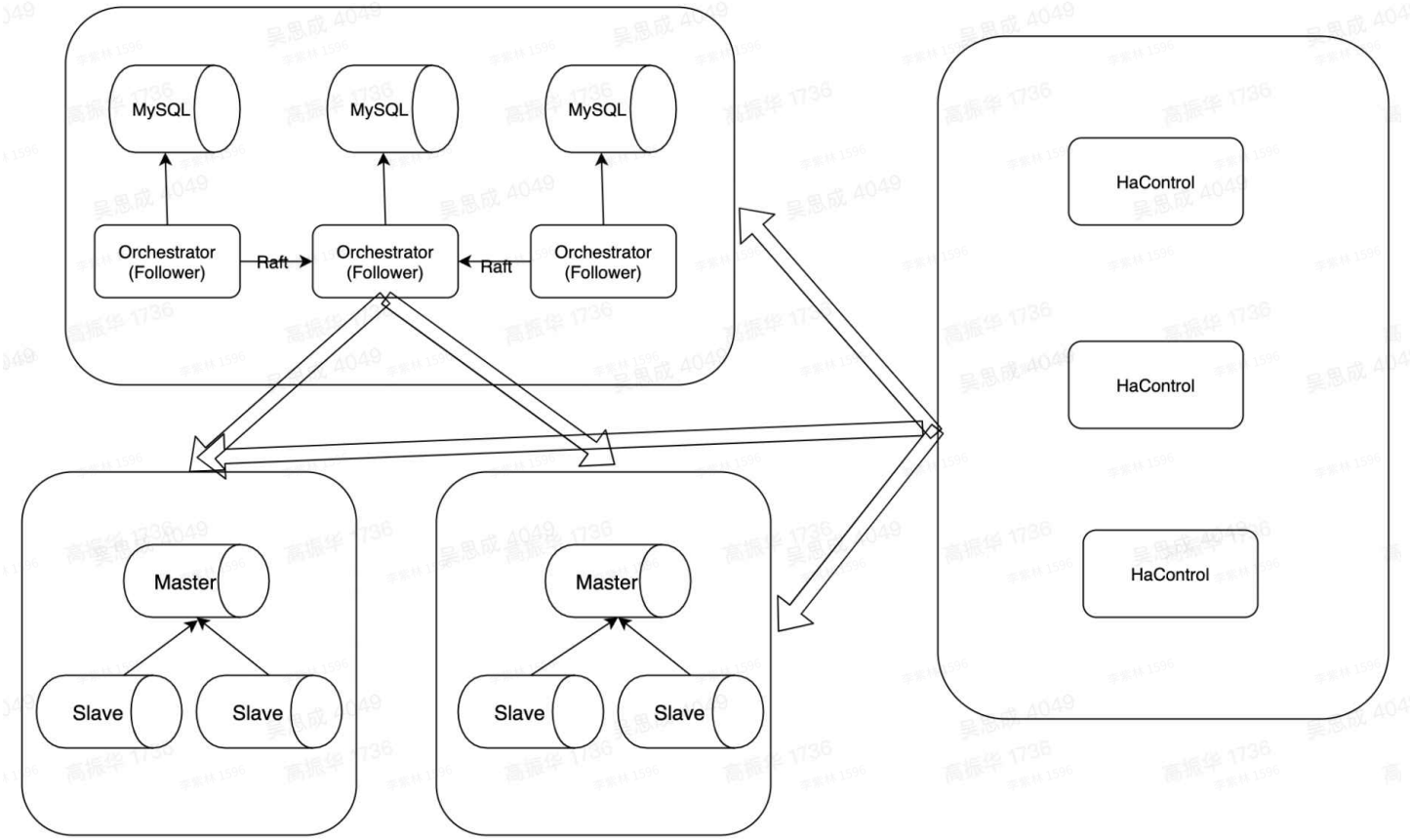
优点：

- 可以实现数据的强一致性
- 扩展性很好，增加节点即可实现数据库集群的扩展

缺点：

- 配置较复杂，需要使用NDB储存引擎，与MySQL常规引擎存在一定差异
- 多个节点通过网络实现通讯和数据同步、查询等操作，因此整体性受网络速度影响比较大

公司的高可用架构：Orchestrator+HaControl



Orchestrator 是一个开源的HA组件,负责 mysql 实例信息的实时进行采集。每3个 Orchestrator 结点组成一个 Orchestrator 集群，每个集群的每个 Orchestrator 结点通过 raft 协议进行数据通信和同步，Orchestrator 集群的高可用也是通过raft实现。

HaControl 负责整个 HA 的切换逻辑，它周期性的检查 Orchestrator 的实例采集结果，如果发现实例超时没有采集则进行相应的报警，DBA 接到报警后判断是否需要切换，然后发送对应的切换命令给 HAControl 进行切换处理。

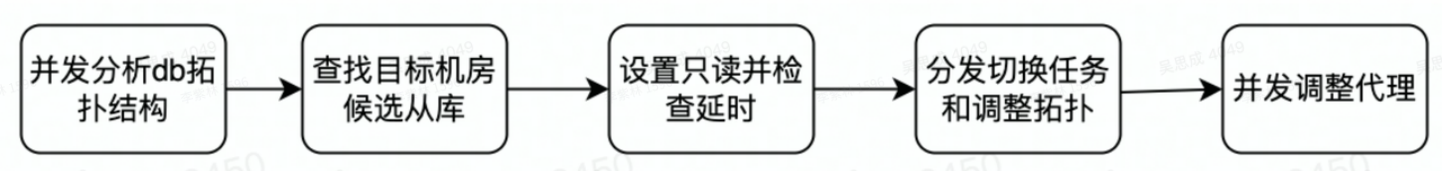
核心功能：

1. 收集mysql实例的实时状态
2. 当mysql实例异常时，发送异常监控报警
3. 实例的切换（异常切换、主动切换）
4. 多机房的容灾切换等功能
5. 业务场景定制化切换（半同步场景）

自动切换逻辑

1. 会去设置阈值，因为网络波动会导致实例批量失败，如果频繁切换，导致一些配置上的异常。 一般设置在40多，即40个实例失败
2. 尝试重连实例，避免由于网络抖动导致HaControl连接实例失败，从而误切换
3. 尝试和其他实例建连，是避免由于HaControl孤岛而导致误判
4. 检查最近是否切换过，是避免因为业务方请求量异常导致实例批量失败，这种情况即使切换实例也没用，因为切换后还是会失败，需要找业务方了解情况

1.1 多机房容灾切换



公司MySQL现状和规模

部署现状

- 传统主从复制（异步复制，只有业务需要强一致的情况下才考虑半同步）
- 多机房部署

部署规模

- 实例5w+，集群5k+
- 日常宕机数量较多（主库10+，从库40+）

2. Paxos和Raft

高可用需要确保分布式一致性，而分布式一致性比较经典的算法则是Paxos和Raft

Paxos 和 Raft

参考：

Mysql Binlog应用场景与原理深入剖析

超详细的canal入门，看这篇就够了

基于Canal+Kafka实现缓存实时更新

《MySQL 复制的前世今生》final .pdf

http://mysql.taobao.org/monthly/2017/12/03/

https://zhuanlan.zhihu.com/p/87963038

Mysql高可用架构演进 （学员版）.pdf

Mysql通过MHA实现高可用

MySQL MGR架构原理简介

MySQL Group Replication的Paxos实现

mysql常见高可用方案

RDS高可用架构概述