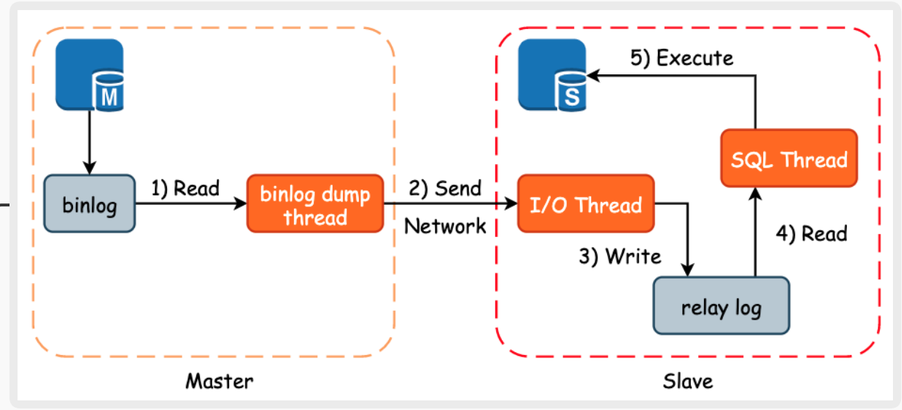
MySQL Replication

异步复制

基本流程



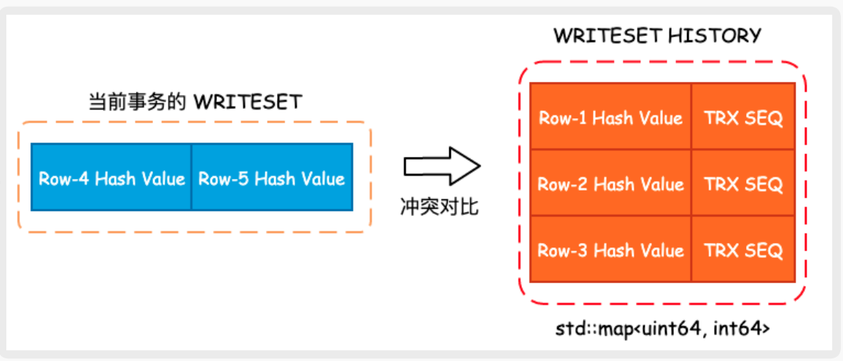
1. Master 开启一个 binlog dump 线程，用于读取 Master 节点的 binlog 内容，并通过 TCP 长连接发送给 Slave 节点
2. Slave 通过 I/O 线程与 Master 建立连接，并将接收到的 binlog 写入到 relay log（中继日志）中。relay log 相当于一个缓冲区，防止后续的 SQL 线程执行过慢，大量数据堆积在内存中，同时也可以作为问题排查的手段之一
3. 最后，SQL Thread 从中继日志中读取数据，解析并执行之。在高版本的 MySQL 中， 这个过程将使用多线程的方式执行，以减少主从之间的复制延迟

MySQL 异步复制模型本身是比较简单和直观的，从模型上来看，比较复杂的地方就在如何使用并行执行的方式来执行 relay log，并且保证对同一行进行修改的多个事务按照原有顺序执行

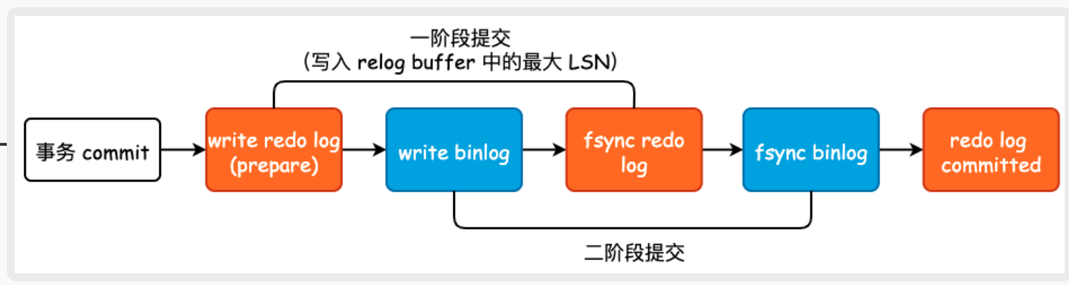
并行复制策略

binlog 并行执行的问题就在于如何对 binlog 进行分发，假设我们现在有 4 个 Worker Thread 并行执行 binlog，那么需要使用什么手段保证数据的一致性呢?

1. 按表分发
   - 如果两个事务是对不同的表进行操作，那么这两个事务即可并行执行。处理方式也非常简单，对表名称进行哈希，并对结果进行 worker 数量的取模，将该事务分发至对应的 worker 即可
   - 但是，如果事务同时对多张表进行了修改的话，上述简单模型就会出现问题。因此，我们还需要记录使用 HashMap 记录下当前线程有哪些事务正在执行或者是正在排队，其中 key 为表名，value 为等待的事务数量
   - 在大多数情况下基于表的并行复制策略能够快速地执行，但是，如果遇到热点表的话，该热点表仍然是串行复制，同样会出现效率问题
2. 按行分发
   - **MySQL 针对基于行的复制进行了优化，将判断两个事务是否存在"冲突"（即是否更新了同一行）由从节点转移至主节点**
   - MySQL 会记录下更新的每一行的哈希值，组成一个集合。为了能够唯一标识同一行，哈希值通常由"库名+表名+索引名+索引值"计算得到，同时，如果存在除主键的其它唯一索引的话，这些唯一索引的名称和值也会被写入至集合中
   - 这个由哈希值所组成的集合被称之为 WRITESET，MySQL 会遍历 WRITESET，于 WRITESET HISTORY 这一 Map 中进行冲突判断，如果没有任何冲突的话，将会直接插入至该 Map 中



3. 基于 Group Commit



在 Group Commit 中，如果两个事务能够同时处于 Prepare 阶段，或者是处于 Prepare 到 Committed 之间，那么这两个事务事务一定通过了锁校验，换而言之，它们一定没有修改同一行数据，也就是说，它们在从库上可以并行执行

MySQL 主要提供了两种并行复制策略，一种是按行分发，称之为 WRITESET；另一种则是基于 Group Commit 的并行复制策略，称之为 COMMIT ORDER

其实还有一种策略，叫做 WRITE_SESSION。WRITE_SESSION 是建立在 WRITESET 之上的，多了一个约束: 在同一个线程先后执行的两个事务，在 Slave 上执行时，要保证原有的顺序。对一致性要求非常高的系统可以选用该策略，平常使用相对较少

对比

- COMMIT ORDER 的策略完全按照 Master 的并行度进行回放，如果 Master 并发写入并不是很多的话，那么 COMMIT ORDER 将退化成单线程回放，也会导致主从之间出现延迟。因此，COMMIT ORDER 适合在 Master 并发程度较高时使用
- WRITESET 模式是基于每个事务并行，如果事务间更新的记录不冲突，就可以并行执行。即使 Master 以单线程的方式插入多条数据，如果它们并不存在冲突，Slave 就可以使用多线程的方式进行回放。不过，相较于 COMMIT ORDER 来说，WIRTESET 需要消耗更多的 CPU 和内存资源

更多细节

我们可以使用 mysqlbinlog binlog.N | grep last_ | sed -e 's/server id.*last/[...] last/' -e 's/.rbr_only.*/ [...]/' 来查看 binlog 文件中每一个事务的 last_committed 和 sequence_number 序列号

binlog 中记录了 sequence_number 和 last_committed，其中 sequence_number 是自增 ID，在单个 binlog 中自增；而 last_committed 则表示上一个提交的事务 ID。如果两个事务的 last_committed 相同，说明这两个事务是在同一个 Group 内提交的，或者说，它们可以并发执行

WRITESET

```
mysql> show variables like "binlog_transaction_dependency_tracking";
+-------------------------------------------+---------------+
| Variable_name                             | Value         |
+-------------------------------------------+---------------+
| binlog_transaction_dependency_tracking | WRITESET |
+-------------------------------------------+---------------+

mysql> insert into user values(1,  "10001", "Auz");
mysql> insert into user values(2, "10002", "Buz");
mysql> insert into user values(3, "10003", "Cuz");
mysql> insert into user values(4, "10004", "Duz");
```

```
#210720 19:29:16 [...] last_committed=1 sequence_number=2 [...]
#210720 19:29:23 [...] last_committed=1 sequence_number=3 [...]
#210720 19:29:32 [...] last_committed=1 sequence_number=4 [...]
#210720 19:29:40 [...] last_committed=1 sequence_number=5 [...]
```

可以看到，在 Master 的 binlog 中，原本顺序执行的 4 个事务拥有着同样的 last_committed，也就是说，Salve 可以直接并行地执行它们

COMMIT ORDER

```
mysql> show variables like "binlog_transaction_dependency_tracking";
+-------------------------------------------+--------------------+
| Variable_name                             | Value              |
+-------------------------------------------+--------------------+
| binlog_transaction_dependency_tracking | COMMIT_ORDER |
+-------------------------------------------+--------------------+

mysql> insert into user values(1,  "10001", "Auz");
mysql> insert into user values(2, "10002", "Buz");
mysql> insert into user values(3, "10003", "Cuz");
mysql> insert into user values(4, "10004", "Duz");
```

```
#210720 19:46:14 [...] last_committed=1 sequence_number=2 [...]
#210720 19:46:17 [...] last_committed=2 sequence_number=3 [...]
#210720 19:46:20 [...] last_committed=3 sequence_number=4 [...]
#210720 19:46:23 [...] last_committed=4 sequence_number=5 [...]
```

可以看到，假如并行复制策略为 COMMIT ORDER 的话，那么串行执行的多个事务即使不存在冲突，binlog 中的 last_committed 也是完全不同的。换而言之，它们在从库无法并行执行

综上，我们需要根据实际的业务场景来选择合适的并行复制策略，不过在大多数的时候，WRITESET 将会有更低的延迟

监测复制延迟

Second Behind Master

- MySQL 提供了 Second Behind Master 字段来大致地给出主-从之间的复制延迟，但是这个值并不一定是准确的
- 对于 Master 的每一个事务而言，在提交以后都会记录一个时间，假设这个时间为 t1。Slave 在执行此条事务时，会获取当前时间 t2，那么使用 t2-t1 就会得到一个时间差，也就是 Second Behind Master
- **之所以说 Second Behind Master 这个值不准确的原因在于，假如 I/O Thread 有延迟的话，Second Behind Master 的值也可能极小，但是此时从库已经落后很多了**
- 这里要备注的是，如果 Slave 机器的时间和 Master 不一致的话，是不会影响 Second Behind Master 的计算的。因为在建立长连接时，Slave 会对比自己和 Master 的系统时间，如果存在差异，那么在计算 Second Behind Master 的时候会补偿掉这个差值

因此，我们可以引入一张心跳表（Heartbeat Table）来更准确的检测主从复制之间的延迟

```
CREATE TABLE heartbeat (
  server_uuid VARCHAR(36) PRIMARY KEY,
  ts TIMESTAMP(6) NOT NULL
);
```

同时可以使用 MySQL 中自带的 Event Scheduler 每隔 2/3/5 秒执行一次时间的更新即可。当然，在计算主库和从库之间的 ts 差值时，需要补偿掉原有机器的时间差值

```
replace into heartbeat values(@@server_uuid, NOW());
```

这其实就是 pt-heartbeat 的基本原理，只不过对其进行了简化而已