28 读写分离有哪些坑？

**读写分离、以及怎么处理主备延迟导致的读写分离问题**

读写分离基本结构

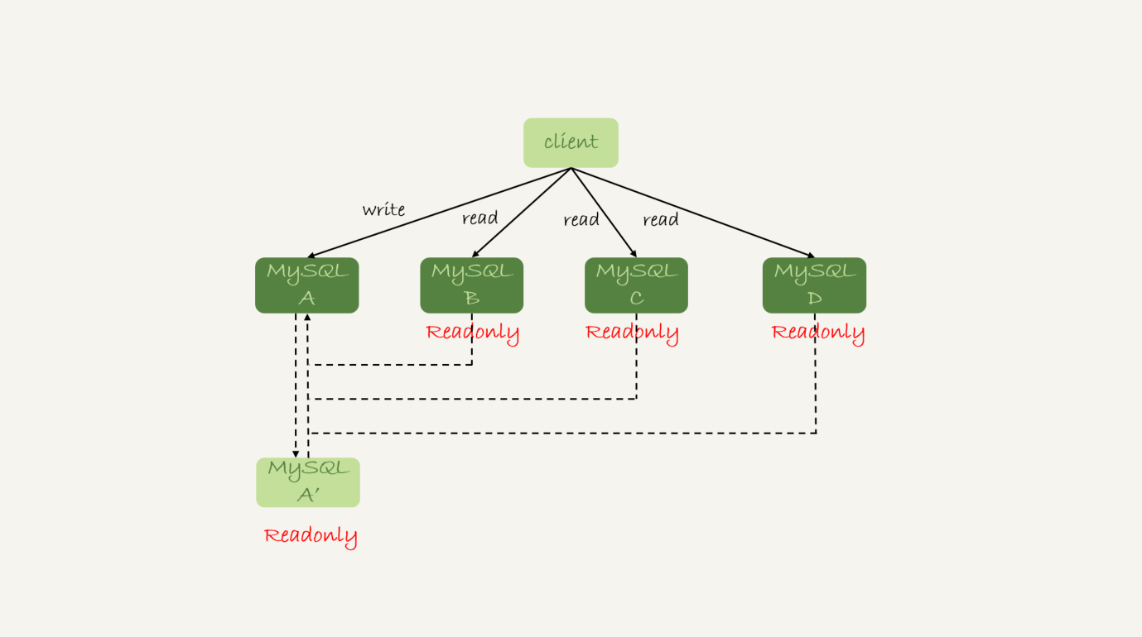


图1 读写分离基本结构

读写分离主要目标是分摊主库压力，图1中的结构是客户端（client）主动做负载均衡，这种模式下一般会把数据库的连接信息放在客户端的连接层，有客户端选择后端数据库进行查询

还有一种架构，在MySQL和客户端之间有一个中间代理层proxy,客户端只连接proxy,有proxy根据请求类型和上下文决定请求的分发路由

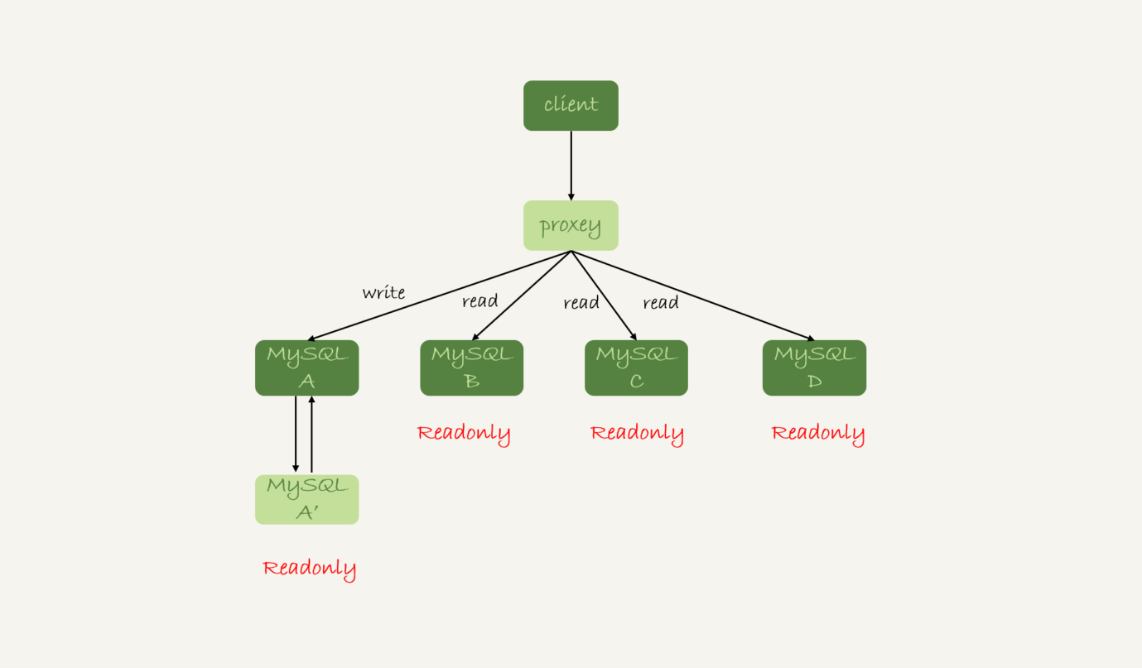


图2 带proxy的读写分离架构

客户端直连和带proxy的读写分离架构，各有那些特点

1.客户端直连方案，因为少了一层proxy转发，所以查询性能稍微好一点，并且整体架构简单，排查问题更方便，但这种方案，由于要了解后端部署细节，所以在出现主备切换、库迁移等操作的时候，客户端都户感知到，并且需要调整数据库连接信息

感觉客户端太麻烦了，信息大量冗余，结构很丑，其实未必，一般采用这样的架构，一户会伴随一个负责管理后端的组件，比如zookeeper,尽量让业务端只专注于业务逻辑开发

1. 带proxy的架构，对客户端比较友好，客户端不需要关注后端细节，连接维护、后端信息维护等工作，都由proxy完成的，但这样的话，对后端维护团队的要求户更高，而且，proxy也需要有高可用架构。因此，带proxy架构的整体相对比较复杂。

具体选择哪个方案取决于数据库团队提供的能力，目前，趋势是往带proxy的架构方向发展

**主题：由于主从可能存在的延迟，客户端执行完一个更新事物马上发起查询，如果查询选择的是从库的话，就有可能读到刚刚的事务更新之前的状态，称为“过期读”。**

怎样处理过期读

方案包括

·强制走主库方案

·sleep方案

·判断主备无延迟方案

·配合semi-sync方案

·等主库位点方案

·等GTID方案

**强制走出库方案**

强制走主库方案是：将查询请求做分类。通常情况下，可以将查询请求分为两类

1. 对于必须拿到最新结果的请求，强制将其发到主库上。比如一个交易平台，卖家发布商品以后，马上返回主页面，看商品是否发布成功。那么，这个请求需要拿到最新的结果，就必须走主库
2. 对于可以读到就数据的请求，才将其发到从库上，在这个交易平台上，买家来逛商铺页面，就算晚几秒看到最新商品，也可以接受的，。这类请求就可以在走从库。

这个方案是不是有点畏难和取巧的意思，但其实这个方案**是用得最多的**。

这个方案的问题在于，有时，会碰到“所有查询都不能是过期读”的需求，比如一些金融类的业务。要放弃读写分离，所有读写压力都在主库，等于放弃了扩展性。

**可以支持读写分离的情况下，有哪些解决过期读的方案**，并分析各个方案的优缺点

**Sleep方案**

主库更新后，读从库之前先sleep一下，具体的方案就是，类似于执行一条select sleep(1)命令

这个方案的假设是，大多数情况下主备延迟1秒之内，做一个sleep可以有很大概率拿到最新的数据。

以卖家发布商品为例，商品发布后，用Ajax直接把客户端输入的内容作为“新商品”显示在页面上，而不是真正的去数据库查询。

卖家可以通过这个显示，来确认产品已经发布成功了，等到卖家在刷新页面，去查看商品的时候，其实已经过了一段时间了，也就达到了sleep的目的，进而解决了过期读的问题

这个方案解决了类似的过期读问题，但存在不精确，包含两层意思：

1. 如果这个查询请求本来0.5秒就可以在从库上拿到正确结果，也会等1秒
2. 如果延迟1秒，还是会出现过期读

这个方案可以解决类似Ajax场景下的过期读问题，但是不够精确

**判断主备无延迟方案**

确保主备无延迟，通常有三种做法

通过25篇，知道show slave status结果里seconds\_behind\_master参数的值，可以来衡量主备延迟时间的长短

所以，**第一种确保主备无延迟的方法是，**每次从库执行查询请求前，先判断seconds\_behind\_master是否已经等于0，如果还不等于0，那就必须等到这个参数变成0才能执行查询请求

Seconds\_behind\_master的单位是秒，如果觉得精度不够的话，还可以采用对比位点和GTID的方法来确保主备无延迟

图3所以，是一个show slave status结果部分截图

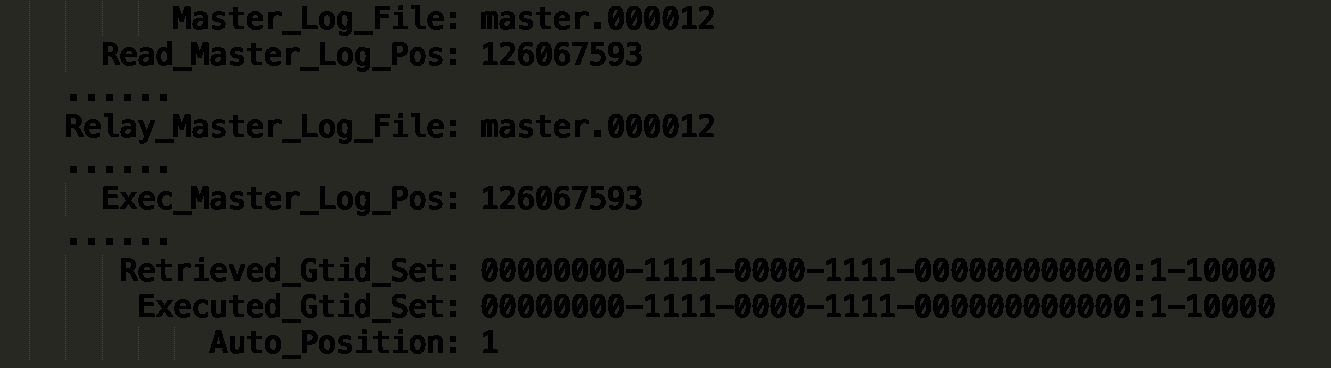


图 3，show slave status 结果

**第二种，**对比位点确保主备无延迟

·master\_Log\_file和Read\_Master\_Log\_Pos，表示读到主库的最新位点

·Relay\_Master\_Log\_FIle和Exec\_Master\_Log\_Pos，表示备库执行的最新位点

如果Master\_Log\_file 和Relay\_Master\_Log\_File、Read\_Master\_Log\_Pos和Exec\_Master\_Log\_Pos这两组值完全相同，就表示接收到的日志已经同步完成

**第三种方法，**对比GTID集合确保主备无延迟

·Auto\_Position=1表示这对主备关系使用了GTID协议

·Retrieved\_Gtid\_Set,是备库收到所有日志的GTID集合

·Executed\_Gtid\_Set,是备库所有已经执行完的GTID集合

如果这连个集合相同，也表示备库接收到日志都已经同步完成

**对比位点和对比GTID这两中方法，都要比判断seconds\_behind\_master是否为0更准确**

在执行查询之前，先判断从库是否同步完成的方法，相比sleep方案，准确度确实提升了不少，但是还没有达到“精确”的程度，为什么这么说？

回顾一个事务binlog在主备库之间的状态

1. 主库执行完成，写入binlog，并反馈给客户端
2. binlog被从主库发送给备库，备库收到
3. 在备库执行binlog完成

判断主备无延迟的逻辑，是“备库收到日志都执行完成了”，但是，从binlog在主备之间状态的分析中，还有一部分日志，处于客户端已经接收到提交确认，而备库还没有收到日志的状态

图4就是这样一个状态

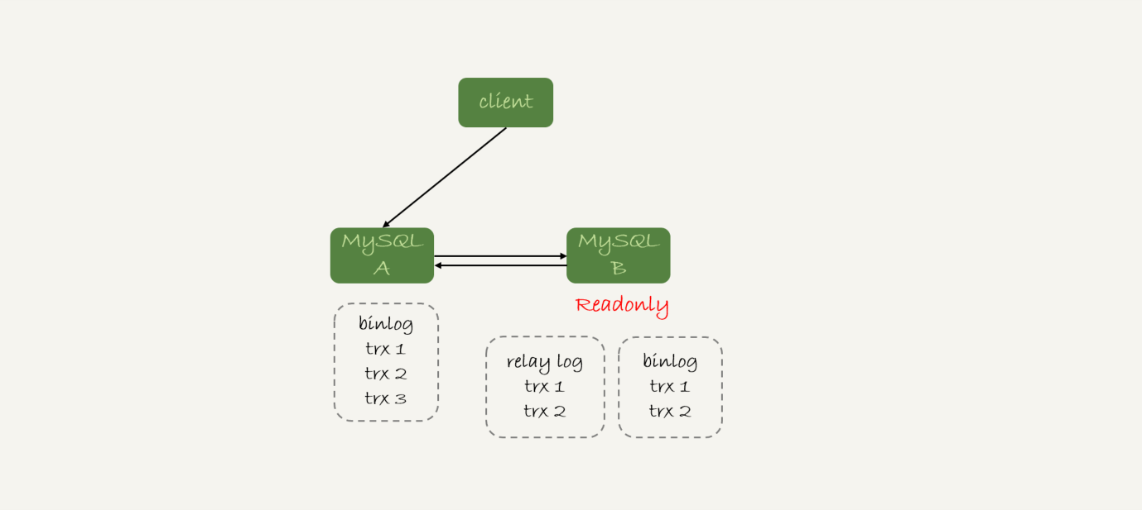


图4 备库还没收到trx 3

主库上执行完三个事务trx1,trx2和trx3

1. trx1和trx2已经传到从库，并且已经执行完了
2. Trx3在主库执行完成，并且已经回顾给客户端，但是还没有传到从库中

如果此时在从库上还行查询请求，按上面的逻辑，从库认为已经没有同步延迟， 但是还是查不到trx3的，就是出现了**过去读**

**上面问题的解决办法**

**配合semi-sync**

解决这个问题，引入半同步复制，也就是semi-sync replication

1. 事务提交的时候，主库把binlog发给从库
2. 从库收到binlog以后，发回给主库一个ack,表示收到
3. 主库收到这个ack后，才能给客户端返回“事务完成”的确认

如果启用了semi-sync，就表示所有给客户端发送过确认的事务，都确保了备库已经收到了这个日志

在25篇评论区，问题：如果主库掉电的时候，有些binlog还来不及发给从库，会不会导致系统数据丢失？

答案是，如果使用的是普通的异步模式复制，就可能会丢失，但semi-sync就可以解决这个问题

这样，semi-sync+ 位点判断的方案，只对一主一备的场景是成立的，在一主多从场景中，主库只要等到一个ack,就开始给客户端返回确认，此时，在从库上执行查询请求，有两种情况

1. 如果查询是落在这个响应了ack的从库上，是能够保证读到最新数据
2. 但是如果查询落到了其他从库上，他们可能还没有收到最新的日志，就会产生过期读