分库分表

1. 为什么要分库分表

分库分表一定是为了支撑高并发、数据量大两个问题的。

分表是啥意思？就是把一个表的数据放到多个表中，然后查询的时候你就查一个表。比如按照用户id来分表，将一个用户的数据就放在一个表中。然后操作的时候你对一个用户就操作那个表就好了。这样可以控制每个表的数据量在可控的范围内，比如每个表就固定在200万以内。

分库是啥意思？就是你一个库一般我们经验而言，最多支撑到并发2000，就需要扩容了，而且一个健康的单库并发值你最好保持在每秒1000左右，不要太大。那么你可以将一个库的数据拆分到多个库中，访问的时候就访问一个库好了。

这就是所谓的分库分表。

2. 用过哪些分库分表中间件？优缺点？

数据库中间库，一般分为两种类型：

- client层，就是一个jar包，引用即可。
- proxy层，需要独立部署数据库中间件的集群，运维。

比较常见的包括：cobar、TDDL、atlas、**sharding-jdbc**、**mycat**

cobar：阿里b2b团队开发和开源的，属于proxy层方案。早些年还可以用，但是最近几年都没更新了，基本没啥人用，差不多算是被抛弃的状态吧。而且不支持读写分离、存储过程、跨库join和分页等操作。

TDDL：淘宝团队开发的，属于client层方案。不支持join、多表查询等语法，就是基本的crud语法是ok，但是支持读写分离。目前使用的也不多，因为还依赖淘宝的diamond配置管理系统。

atlas：360开源的，属于proxy层方案，以前是有一些公司在用的，但是确实有一个很大的问题就是社区最新的维护都在5年前了。所以，现在用的公司基本也很少了。

sharding-jdbc：当当开源的，属于client层方案。确实之前用的还比较多一些，因为SQL语法支持也比较多，没有太多限制，而且目前推出到了2.0版本，**支持分库分表、读写分离、分布式id生成、柔性事务（最大努力送达型事务、TCC事务）**。而且确实之前使用的公司会比较多一些（这个在官网有登记使用的公司，可以看到从2017年一直到现在，是不少公司在用的），目前社区也还一直在开发和维护，还算是比较活跃，个人认为算是一个现在也可以选择的方案。

mycat：基于cobar改造的，属于proxy层方案，支持的功能非常完善，而且目前应该是非常火的而且不断流行的数据库中间件，社区很活跃，也有一些公司开始在用了。但是确实相比于sharding jdbc来说，年轻一些，经历的锤炼少一些。

所以综上所述，现在其实建议考量的，就是sharding-jdbc和mycat，这两个都可以去考虑使用。

sharding-jdbc这种client层方案的优点在于不用部署，运维成本低，不需要代理层的二次转发请求，性能很高，但是如果遇到升级啥的需要各个系统都重新升级版本再发布，各个系统都需要耦合sharding-jdbc的依赖；

mycat这种proxy层方案的缺点在于需要部署，自己及运维一套中间件，运维成本高，但是好处在于对于各个项目是透明的，如果遇到升级之类的都是自己中间件那里搞就行了。

通常来说，这两个方案其实都可以选用，但是我个人建议中小型公司选用sharding-jdbc，client层方案轻便，而且维护成本低，不需要额外增派人手，而且中小型公司系统复杂度会低一些，项目也没那么多；

但是中大型公司最好还是选用mycat这类proxy层方案，因为可能大公司系统和项目非常多，团队很大，人员充足，那么最好是专门弄个人来研究和维护mycat，然后大量项目直接透明使用即可。

3. 如何对数据库进行垂直拆分或水平拆分的?

水平拆分：就是把一个表的数据给弄到多个库的多个表里去，但是每个库的表结构都一样，只不过每个库表放的数据是不同的，所有库表的数据加起来就是全部数据。水平拆分的意义，就是将数据均匀放更多的库里，然后用多个库来抗更高的并发，还有就是用多个库的存储容量来进行扩容。

垂直拆分：就是把一个有很多字段的表给拆分成多个表，或者是多个库上去。每个库表的结构都不一样，每个库表都包含部分字段。一般来说，会将较少的访问频率很高的字段放到一个表里去，然后将较多的访问频率很低的字段放到另外一个表里去。因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。

这个其实挺常见的，不一定我说，大家很多同学可能自己都做过，把一个大表拆开，订单表、订单支付表、订单商品表。

还有表层面的拆分，就是分表，将一个表变成N个表，就是让每个表的数据量控制在一定范围内，保证SQL的性能。否则单表数据量越大，SQL性能就越差。一般是200万行左右，不要太多，但是也得看具体你怎么操作，也可能是500万，或者是100万。你的SQL越复杂，就最好让单表行数越少。

无论是分库了还是分表了，上面说的那些数据库中间件都是可以支持的。就是基本上那些中间件可以做到你分库分表之后，中间件可以根据你指定的某个字段值，比如说userid，自动路由到对应的库上去，然后再自动路由到对应的表里去。

你得考虑一下，你的项目里该如何分库分表？一般来说，垂直拆分，你可以在表层面来做，对一些字段特别多的表做一下拆分；水平拆分，你可以说是并发承载不了，或者是数据量太大，容量承载不了，你给拆了，按什么字段来拆，你自己想好；分表，你考虑一下，你如果哪怕是拆到每个库里去，并发和容量都ok了，但是每个库的表还是太大了，那么你就分表，将这个表分开，保证每个表的数据量并不是很大。

而且这儿还有两种分库分表的方式，一种是按照**range**来分，就是每个库一段连续的数据，这个一般是按比如时间范围来的，但是这种一般较少用，因为很容易产生热点问题，大量的流量都打在最新的数据上了；或者是按照某个字段**hash**一下均匀分散，这个较为常用。

range分法，好处在于说，后面扩容的时候，就很容易，因为你只要预备好，给每个月都准备一个库就可以了，到了一个新的月份的时候，自然而然，就会写新的库了；缺点，但是大部分的请求，都是访问最新的数据。实际生产用range，要看场景，你的用户不是仅仅访问最新的数据，而是均匀的访问现在的数据以及历史的数据。

hash分法，好处在于说，可以平均分配每个库的数据量和请求压力；坏处在于说扩容起来比较麻烦，会有一个数据迁移的这么一个过程。

4. 分库分表需要解决的问题

1）事务问题

使用分布式事务。

2）跨节点Join的问题

普遍做法是分两次查询实现。在第一次查询的结果集中找出关联数据的id,根据这些id发起第二次请求得到关联数据。

3）跨节点的count,order by,group by以及聚合函数问题

分别在各个节点上得到结果后在应用程序端进行合并。和join不同的是每个结点的查询可以并行执行，因此很多时候它的速度要比单一大表快很多。但如果结果集很大，对应用程序内存的消耗是一个问题。

4）数据迁移，容量规划，扩容等问题

来自淘宝综合业务平台团队，它利用对2的倍数取余具有向前兼容的特性（如对4取余得1的数对2取余也是1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。

5. 如何动态分库分表?

1）停机迁移方案

先来一个最low的方案，比如大家伙儿凌晨12点开始运维，网站或者app挂个公告，说0点到早上6点进行运维，系统届时无法访问。接下来到0点之后，停机，系统停掉，此时没有流量写入了，数据库静止了。然后提前开发好导数据的工具，此时直接跑起来，然后将单库单表的数据哗哗哗读出来，写到分库分表里面去。导数完了之后，就ok了，修改系统的数据库连接配置啥的，包括可能代码和SQL也许有修改，那你就用最新的代码，然后直接启动连到新的分库分表上去。

但是这个方案比较low，谁都能干，我们来看看高大上一点的方案。

2）双写迁移方案

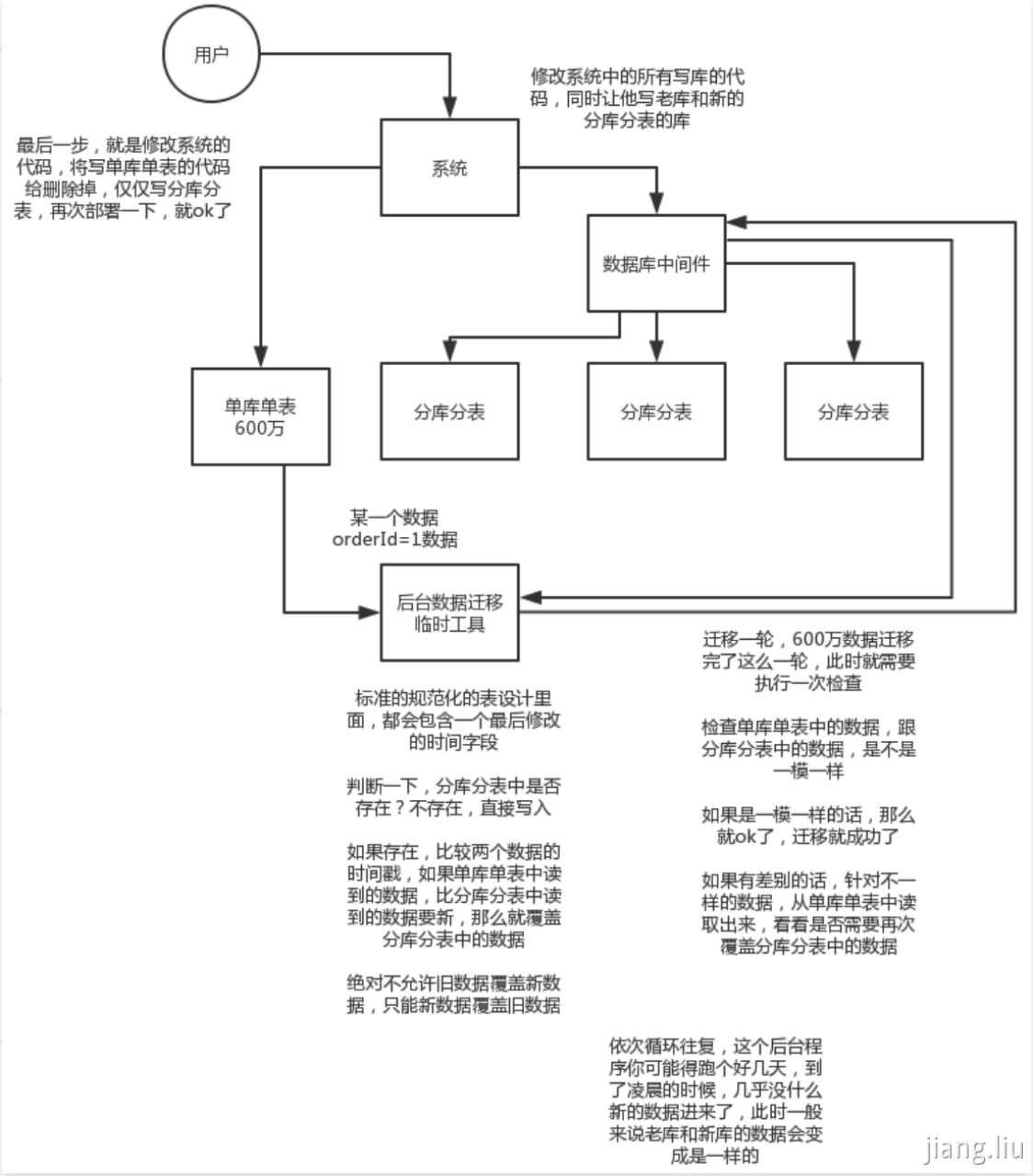
这个迁移方案比较靠谱一些，不用停机，不用看北京凌晨4点的风景。

简单来说，就是在线上系统里面，之前所有写库的地方（增删改操作），除了对老库增删改，都加上对新库的增删改，这就是所谓双写，老库和新库同时写。

然后系统部署之后，新库数据差太远，开发一个导数程序，读老库数据写入新库，写的时候要根据gmt_modified这类字段判断这条数据的最后修改的时间，除非是读出来的数据在新库里没有，或者是比新库的数据新才会写。

接着导完一轮之后，有可能数据还是存在不一致，那么就程序自动做一轮校验，比对新老库每个表的每条数据，接着如果有不一样的，就针对那些不一样的，从老库读数据再次写。反复循环，直到两个库每个表的数据都完全一致为止。

接着当数据完全一致了，就ok了，基于仅仅使用分库分表的最新代码，重新部署一次，不就仅仅基于分库分表在操作了么，还没有几个小时的停机时间，很稳。所以现在基本玩儿数据迁移之类的，都是这么干了。



6. 如何设计可以动态扩容缩容的分库分表方案?

比如说分库分表方案是这样：

（1）选择一个数据库中间件，调研、学习、测试

（2）设计你的分库分表的一个方案，你要分成多少个库，每个库分成多少个表，3个库每个库4个表

（3）基于选择好的数据库中间件，以及在测试环境建立好的分库分表的环境，然后测试一下能否正常进行分库分表的读写

（4）完成单库单表到分库分表的迁移，双写方案

（5）线上系统开始基于分库分表对外提供服务

（6）扩容了，扩容成6个库，每个库需要12个表，你怎么来增加更多库和表呢?

现在问题来了，你现在这些库和表又支撑不住了，要继续扩容咋办？这个可能就是说你的每个库的容量又快满了，或者是你的表数据量又太大了，也可能是你每个库的写并发太高了，你得继续扩容。

==> 扩容方案：

1）停机扩容

这个方案就跟停机迁移一样，步骤几乎一致，唯一的一点就是那个导数的工具，是把现有库表的数据抽出来慢慢倒入到新的库和表里去。一般不建议这么做。从单库单表迁移到分库分表的时候，数据量并不是很大，但是需要在分库分表的基础上扩容，肯定说明数据量实在是太大了，可能多达几亿条，甚至几十亿，你这么玩儿，可能会出问题。面试的时候千万别这么说。

2）优化后的方案

一开始上来就是32个库，每个库32个表，1024张表。

这个分法，第一，基本上国内的互联网肯定都是够用了，第二，无论是并发支撑还是数据量支撑都没问题。

每个库正常承载的写入并发量是1000，那么32个库就可以承载32 * 1000 = 32000的写并发，如果每个库承载1500的写并发，32 * 1500 = 48000的写并发，接近5万/s的写入并发，前面再加一个MQ，削峰，每秒写入MQ 8万条数据，每秒消费5万条数据。

1024张表，假设每个表放500万数据，在MySQL里可以放50亿条数据。每秒的5万写并发，总共50亿条数据，对于国内大部分的互联网公司来说，其实一般来说都够了。

针对分库分表的扩容，第一次分库分表，就一次性给他分个够，32个库，1024张表，可能对大部分的中小型互联网公司来说，已经可以支撑好几年了。

利用32 * 32来分库分表，即分为32个库，每个库里一个表分为32张表。一共就是1024张表。根据数据id先对32取模路由到库，再对32取模路由到表。

> 刚开始的时候，这个库可能就是逻辑库，建在一个数据库上的，就是一个mysql服务器可能建了n个库，比如16个库。后面如果要拆分，就是不断在库和mysql服务器之间做迁移就可以了。然后系统配合改一下配置即可。
>
> 比如说最多可以扩展到32个数据库服务器，每个数据库服务器是一个库。如果还是不够？最多可以扩展到1024个数据库服务器，每个数据库服务器上面一个库一个表。因为最多是1024个表么。

哪怕是要减少库的数量，也很简单，其实说白了就是按倍数缩容就可以了，然后修改一下路由规则。

```Java
对2 ^ n取模
orderId 模 32 = 库
orderId / 32 模 32 = 表
```

7. 分库分表之后，id主键如何处理?

1）数据库自增id

系统里每次得到一个id，都是往一个库的一个表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个id。拿到这个id之后再往对应的分库分表里去写入。这个方案的好处就是方便简单，谁都会用；缺点就是单库生成自增id，要是高并发的话，就会有瓶颈的。

适用场景：数据量太大导致的分库分表扩容，你可以用这个方案，因为可能每秒最高并发最多就几百，那么就走单独的一个库和表生成自增主键即可。

2）uuid

好处就是本地生成，不要基于数据库来了；不好之处就是，uuid太长了，作为主键性能太差了，不适合用于主键。

适合的场景：如果你是要随机生成个什么文件名了，编号之类的，你可以用uuid，但是作为主键是不能用uuid的。

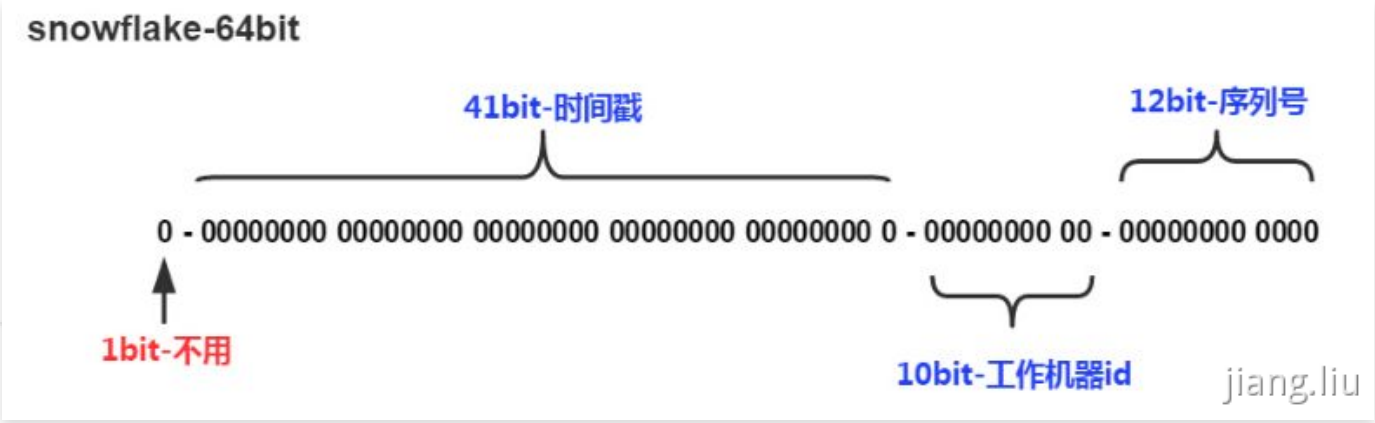
3）获取系统当前时间

获取当前时间即可，但是问题是，并发很高的时候，比如一秒并发几千，会有重复的情况，这个是肯定不合适的。基本就不用考虑了。

适用场景：一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个id，如果业务上你觉得可以接受，那么也是可以的。你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号，订单编号，时间戳 + 用户id + 业务含义编码。

4）snowflake算法（雪花算法）

twitter开源的分布式id生成算法，就是把一个64位的long型的id，1个bit是不用的，用其中的41 bit作为毫秒数，用10 bit作为工作机器id，12 bit作为序列号。



64位的long型的id，64位的long -> 二进制

```
0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 11001 | 000000000000
```

2018-01-01 10:00:00 -> 做了一些计算，再换算成一个二进制，41bit来放 -> `0001100 10100010 10111110 10001001 01011100 00`。

机房id，17 -> 换算成一个二进制 -> 10001。

机器id，25 -> 换算成一个二进制 -> 11001。

snowflake算法服务，会判断当前这个请求是否是，机房17的机器25，在2175/11/7 12:12:14时间点发送过来的第一个请求，如果是第一个请求，假设在2175/11/7 12:12:14时间里，机房17的机器25，发送了第二条消息，snowflake算法服务，会发现说机房17的机器25，在2175/11/7 12:12:14时间里，在这一毫秒，之前已经生成过一个id了，此时如果你同一个机房，同一个机器，在同一个毫秒内，再次要求生成一个id，此时会只能把`0000 00000000`加1。

雪花算法有一个工具类，可以基于该类进行封装。

源码中的规则：

- 机房 id（5bit） 最大只能是 32 以内，
- 机器 id（5bit） 最大只能是 32 以内，
- 序列号（12 bit），最多在 4096 以内（一毫秒内可以最多生成4096个ID，性能相当牛逼）。

8. Mysql读写分类的原理?

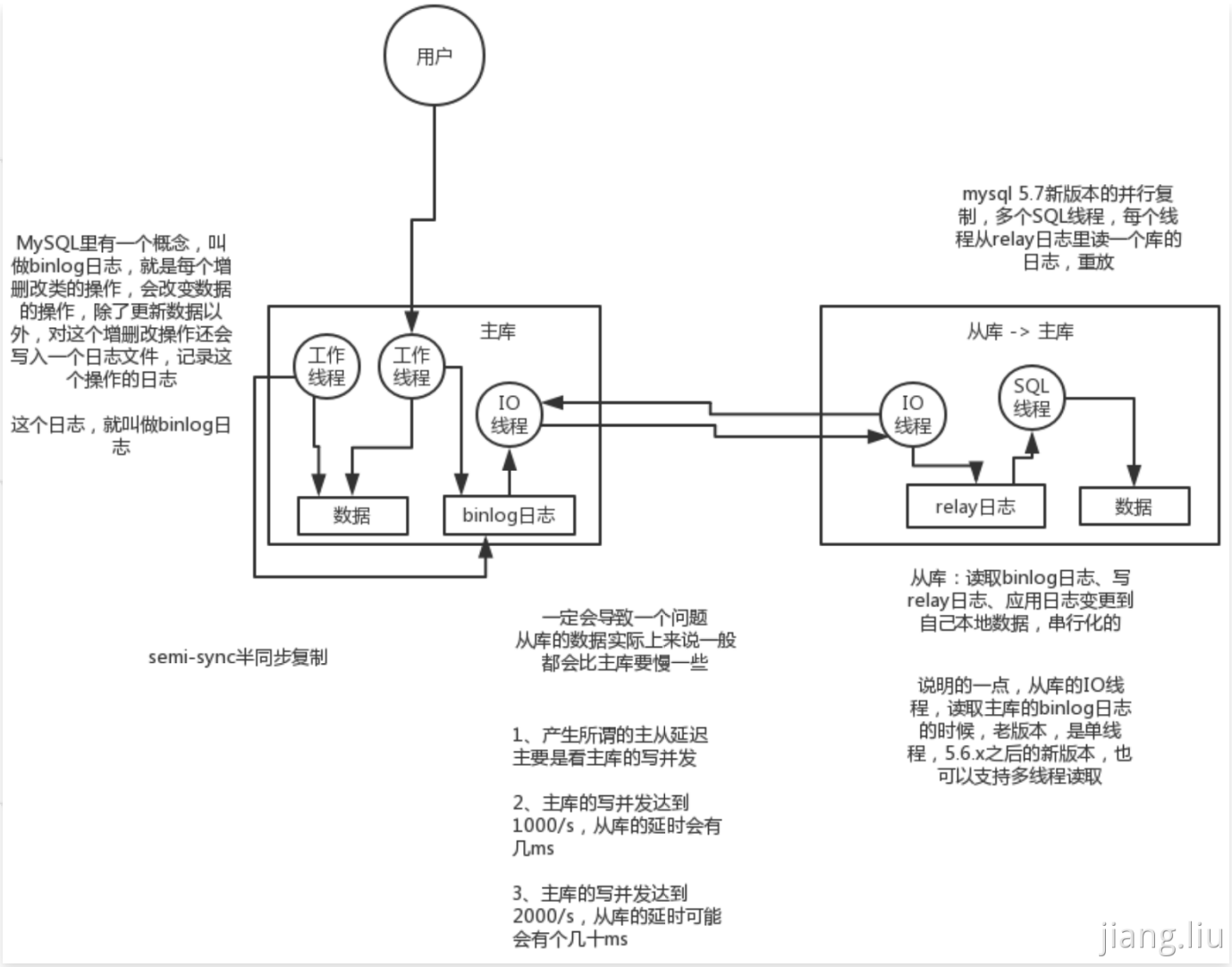
如何实现mysql的读写分离?

其实很简单，就是基于主从复制架构，简单来说，就搞一个主库，挂多个从库，然后我们就单单只是写主库，然后主库会自动把数据给同步到从库上去。

MySQL主从复制原理?

主库将变更写**binlog**日志，然后从库连接到主库之后，从库有一个IO线程，将主库的binlog日志拷贝到自己本地，写入一个**中继日志**中。接着从库中有一个SQL线程会从中继日志读取binlog，然后执行binlog日志中的内容，也就是在自己本地再次执行一遍SQL，这样就可以保证自己跟主库的数据是一样的。

看下图流程！！！



这里有一个非常重要的一点，就是从库同步主库数据的过程是**串行化**的，也就是说主库上并行的操作，在从库上会串行执行。所以这就是一个非常重要的点了，由于从库从主库拷贝日志以及串行执行SQL的特点，在高并发场景下，从库的数据一定会比主库慢一些，是有延时的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。

而且这里还有另外一个问题，就是如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

所以mysql实际上在这一块有两个机制，一个是**半同步复制**，用来解决主库数据丢失问题；一个是**并行复制**，用来解决主从同步延时问题。

半同步复制：semi-sync复制，指的就是主库写入binlog日志之后，就会强制立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库的ack之后才会认为写操作完成了。（因为从库将数据拉取到relay log之后，还并未写到表里面，所以叫做半同步）

并行复制：指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这是库级别的并行

跟人家聊主从复制一定要讲清楚下面几个问题：

1）主从复制的原理

2）主从延迟问题产生的原因

3）主从复制的数据丢失问题，以及半同步复制的原理

4）并行复制的原理，多库并发重放relay日志，缓解主从延迟问题。

mysql主从同步延时问题（精华）

show status，Seconds_Behind_Master，你可以看到从库复制主库的数据落后了几ms

比如用了mysql主从架构之后，可能会发现刚写入库的数据结果没查到，结果就完蛋了。。。。

所以实际上你要考虑好应该在什么场景下来用这个mysql主从同步，建议是一般在读远远多于写，而且读的时候一般对数据时效性要求没那么高的时候，用mysql主从同步。

所以这个时候，我们需要考虑，可以用mysql的并行复制，但是问题是那是库级别的并行，所以有时候作用不是很大。

所以通常来说，我们会对于那种写了之后立马就要保证可以查到的场景，**采用强制读主库的方式**，这样就可以保证你肯定的可以读到数据了吧。其实用一些数据库中间件是没问题的。

一般来说，如果主从延迟较为严重：

- 分库，将一个主库拆分为4个主库，每个主库的写并发就500/s，此时主从延迟可以忽略不计。
- 打开mysql支持的并行复制，多个库并行复制，如果说某个库的写入并发就是特别高，单库写并发达到了2000/s，并行复制还是没意义。28法则，很多时候比如说，就是少数的几个订单表，写入了2000/s，其他几十个表10/s。
- 重写代码，写代码的同学，要慎重，当时我们其实短期是让那个同学重写了一下代码，插入数据之后，直接就更新，不要查询。
- 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库。不推荐这种方法，你这么搞导致读写分离的意义就丧失了。

> 插入一条数据--> 查询这条数据 --> 更新这条数据

> 上面的步骤，在主从复制下出现同步延迟的时候，会查询到null，导致第三步无法执行。这种一般直接取消第二步查询操作，直接更新。更新也在主库。不会出现查不到更新不了的情况。

Slave的IO线程连上Master，并请求日志文件指定位置(或从开始的日志)之后的日志的内容。

Master接收到来自Slave的IO线程请求后，负责复制IO线程根据请求的信息读取指定日志之后的日志信息，返回给Slave端的IO线程。返回信息中除了日志所包含的信息，还包含了包括本次返回的信息在Master端的Binary Log文件的名称和位置。

Slave的IO线程接受到信息后，将日志内容一次写入Slave端的Relay Log文件(mysql-relay-bin.xxxx)的末端，并将读取到的Master端的bin-log的文件和位置记录到master-info文件中，以便在下一次读取时能够清楚地告诉Master，下次从bin-log哪个位置开始往后的日志内容。

Slave的SQL线程检测检测到Relay Log中更新内容后，会马上解析该Log文件中的内容，还原成在Master端真实执行时的可执行的SQL语句，并执行这些SQL语句。实际上Master和Slave执行同样的语句。

主从延迟的主要原因是：主库多线程并发更新，从库单线程串行更新。**解决方法**：将从库变为多线程更新，可以使用mysql-transfer：是一个基于mysql的补丁，用来加速主从同步速度。