1. MySQL数据库应该使用“utf8mb4”编码，永远都不要再使用“utf8”，MySQL的“utf8”实际上不是真正的UTF-8，“utf8”编码最多支持每个字符三个字节，而UTF-8是每个字符最多四个字节。
2. MySQL数据库的默认最大连接数是100，sleep连接最大存活时间默认是28800秒，容易出现Too Many Connections 问题。解决办法：调大最大连接数，减少sleep连接存活时间。
3. MySql使用limit分页，当页数过大，当limit分页的偏移量越大的时候扫描的行数就越多，这就是为什么我们查询越靠后的数据越慢。
4. 在以后的项目中，主推Jackson和gson，逐渐淘汰FastJson
5. Lombok对于第一个字母小写，第二个字母大写的属性生成get-set方法有问题，1.不要这样命名属性，2.生成get-set方法覆盖。
6. Lombok中Accessor（chain=true）会生成返回值不为void的set方法。
7. 单利模式不一定是绝对安全，他可以被反射破坏，获取到私有构造函数，把构造函数设置为共有，然后创建对象。解决办法：在构造函数中添加判断，如果对象存在抛出异常。
8. Spring单例Bean注入多例，因为单利Bean只实例化一次，所以注入多例
9. 布隆过滤

# MySQL连接数太多应该怎么解决

Mysql 数据库的默认最大连接数是：100，

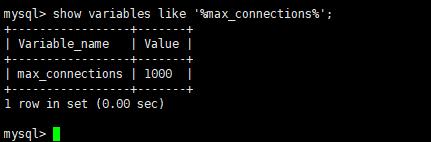
对于多人开发的单体项目来说，虽然我们同时在用的连接不会超过10个，理论上100 绰绰有余，但是除了我们正在使用的连接以外，还有很大一部分 Sleep 的连接，这个才是真正的罪魁祸首。

分析到了问题的根源，我们就需要对症下药，依次解决：

**修改 Mysql 最大连接数量**

首先查看当前 Mysql 最大连接数量是多少：

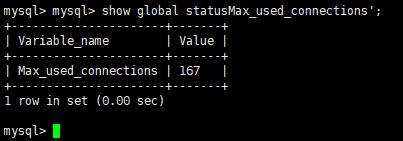
show variables like '%max\_connections%';



这里我已经修改过了，所以是 1000，没有改过的童鞋应该还是 100，

然后查看从这次 mysql 服务启动到现在，同一时刻并行连接数的最大值：

show status like 'Max\_used\_connections';



对于 Mysql 的最大连接数设置，在首次配置的时候设置一个较大的数值，以后在使用的过程中，周期的查询 Max\_used\_connections 然后根据他的值和服务器的性能确定一个最适合当前项目的最大连接数

最大连接数的修改有两种方式

1. 使用 sql 语句(立即生效，但服务器重启后失效)：

set global max\_connections = 1000;

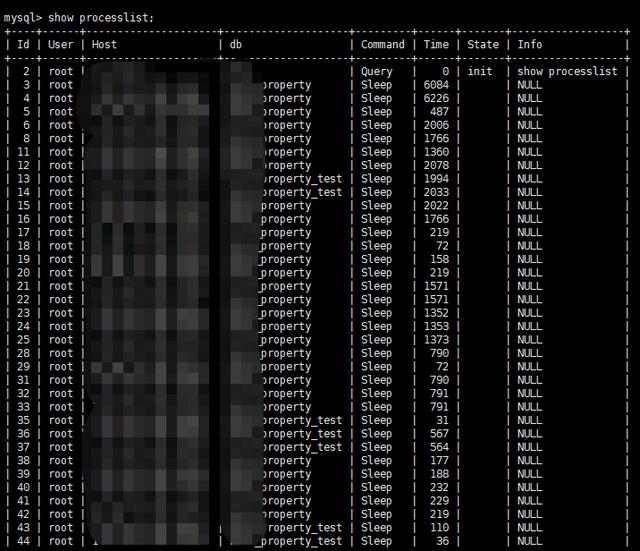


1修改 /etc/my.cnf.添加 max\_connections = 1000 永久有效。重启后生效

但更改最大连接数只能从表面上解决问题，随着我们开发人员的增多，Sleep 连接也会更多，到时候万一又达到了 1000 的上限，难道我们又得改成 10000 吗？这显然是非常不可取的。所以我们不仅要治标，还要治本。杀掉多余的 Sleep 连接就是治本

## ****杀掉Sleep连接****

我们可以通过 show\_processlist 命令来查看当前的所有连接状态

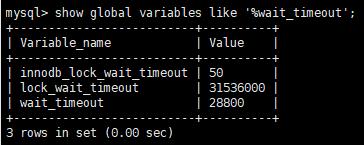


可以发现， Sleep 的连接占了绝大多数。

Mysql 数据库有一个属性 wait\_timeout 就是 sleep 连接最大存活时间，默认是 28800 s，换算成小时就是 8 小时，我的天呐！这也太长了！严重影响性能。相当于今天上班以来所有建立过而未关闭的连接都不会被清理。

执行命令：

show global variables like '%wait\_timeout';



我们将他修改成一个合适的值，这里我改成了 250s。当然也可以在配置文件中修改，添加 wait\_timeout = 250。这个值可以根据项目的需要进行修改，以 s 为单位。我在这里结合 navicat 的超时请求机制配置了 240s。

执行命令：

set global wait\_timeout=250;

这样，就能从根本上解决 Too Many Connections 的问题了

# 如何优化mysql分页查询

## 为什么想到了优化分页查询

握草，我一听就觉得事情不对，肯定没好事，走过去一看，同事对我说，他维护的这张日志表数据已经超过500w了，可能是测试的同事在做压力测试，导致了数据库的用户操作日志记录一下子就突破了几百万，现在，同事写的分页查询速度已经很慢很慢了，当查询的记录越靠后的时候，查询时间越久，果不其然，没过多久，我们就在禅道上看到了测试大哥提交的bug：查看1000页以后的日志返回速度极慢，这是被迫优化啊，好尴尬。

## 如何优化

问题是找到了，那是因为当数据库存放的记录过大的时候，查询越靠后的记录速度越慢，为什么查询越靠后的记录就越慢呢？

我这里主要介绍mysql的分页优化，sqlserver、Oracle可以参考思想，还是回归之前的那个问题，我们来看看问什么查询会慢？我们需要实现准备好数据表以及记录。

## 创建数据表

DROP TABLE IF EXISTS `user`;  
  
CREATE TABLE `user` (  
 `id` bigint(20) NOT NULL AUTO\_INCREMENT,  
 `u\_name` varchar(50) COLLATE utf8mb4\_unicode\_ci NOT NULL COMMENT '用户名',  
 `u\_password` varchar(100) COLLATE utf8mb4\_unicode\_ci NOT NULL COMMENT '密码',  
 `u\_mail` varchar(100) COLLATE utf8mb4\_unicode\_ci DEFAULT NULL COMMENT '邮箱',  
 `u\_phone` varchar(11) COLLATE utf8mb4\_unicode\_ci DEFAULT NULL COMMENT '手机',  
 `u\_sex` tinyint(1) DEFAULT NULL COMMENT '性别',  
 `u\_headImg` varchar(500) COLLATE utf8mb4\_unicode\_ci DEFAULT NULL COMMENT '头像',  
 PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO\_INCREMENT=4762599 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_unicode\_ci;

## 添加数据

我在数据表中添加了100w张三、100w李四、100w王五、200w赵六的用户，所以表中一共500w数据，添加数据的方式采用的存储过程。

BEGIN  
 declare i int; # declare语句是在复合语句中声明变量的指令  
 set i=1;  
 while i<=2000000 do  
 insert into `user`(u\_name,u\_password,u\_mail,u\_phone,u\_sex,u\_headImg) values('赵六','000000','zhaoliu@163.com','18800000000',0,'oss.file.com/images/zhaoliu.png');  
 set i=i+1;  
 end while;  
END

## 查询过慢的原因

数据表和记录都已经准备好了，现在我们就需要来排查为什么分页查询页码越靠后查询速度越慢，我们先来看一个普通的分页查询：

select \* from user order by id desc limit 100,10;

这是查询第十页的数据，我相信大部分的人在写分页查询的时候都是这么写的，其中100：偏移量，意思就是说从哪里之后就是我需要的数据，10：表示需要查询多少条记录，这个就是mysql的分页查询语法，你能看出这条sql存在什么问题吗？乍一看好像没啥问题，真的是这样吗？我们来看几个例子。

**1：偏移量=0**

select \* from user order by id desc limit 0,10;

**查询时间**



**2.偏移量=1000**

select \* from user order by id desc limit 1000,10;

**查询时间**



**3.偏移量=10000**

select \* from user order by id desc limit 10000,10;

**查询结果**



就算偏移量达到了10000，我们的查询速度还是很快的，这说明这条sql就没有任何问题了吗？既然这样，我就让你们看一下当偏移量达到200w的时候，会发生什么事情?

**4.偏移量= 400w**

select \* from user order by id desc limit 4000000,10;

**查询结果**



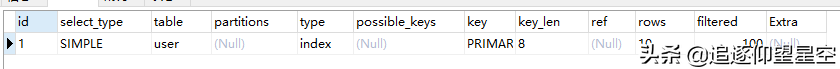
也就是说偏移量越大，查询的时候就越久，这是问什么呢？明明都是从查询10条记录，为什么偏移量越大，查询时间越久呢？

## limit分页的原理

为什么会慢？我们不妨先猜测一下，像函数、扫描记录过多等等都会影响查询的速度，很显然这里我们并没有使用到函数，所以这会不会是扫描的记录过多呢？

这个就和limit有关了，你们知道limit是如何实现分页的吗？我们使用wxplain关键字来分别打印一下偏移量=0、1000、10000、400w的查询详情。

**1：偏移量=0**



**2.偏移量=1000**



**3.偏移量=10000**



**4.偏移量= 400w**



我们先来解释一下这些字段分别是什么意思

id:标识符

select\_type:查询的类型。

table:结果表

partitions:匹配的分区

type:表的连接类型

possible\_keys:也许会使用的索引

key:实际使用的索引

key\_len:索引字段的长度

ref:列与索引的比较

rows:扫描出的行数

filtered:按表条件过滤的行百分比

Extra:执行情况的描述和说明

我们对比一下上面的信息，会发现只有一个字段的值有着很大的区别，那就是rows：扫描的行数，当limit分页的偏移量越大的时候扫描的行数就越多，这就是为什么我们查询越靠后的数据越慢。

假如你现在要查询的偏移量为100w，那么limit会扫描1000010行，然后丢弃前100w行数据，留下最后10行，返回给我们，所以说我们只需要控制扫描的行数，查询的速度自然就快了，那如何控制扫描的行数呢？

## 1.最大id查询法

扫描意思呢？举个例子，我查询第一页的时候是limit 0,10 查询到的最后一条id是10，那么下一页的查询只需要查询id大于10的19条数据即可。

explain select \* from user where id > 4000000 limit 10;



我们看到rows=1949780，这个表示可能扫描了这么多行，这个行数是因为扫描id>400w的记录，后面还有两百万，而这里表示可能扫描了1949780行，但是由于limit 10的存在，所以扫描了10行之后就停止扫描了，我们也可以对比一下使用这种方式和直接使用limit 4000000，10的效率，千差万别。**但是这种比较局限，只能适用于自增组件**， uuid生成的主键这种方式不适用。

## 2.BETWEEN … AND

select \* from user where id BETWEEN 4000000 and 4000010

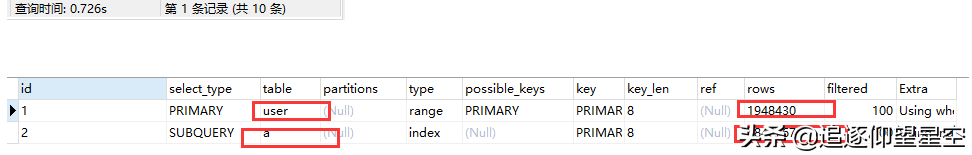




**这种方式也只能适用于自增主键**，**并且id没有断裂，否者不推荐这种方式**，我们发现使用BETWEEN AND的时候查询出来11条记录，也就是说BETWEEN AND包含了两边的边间条件。使用的时候需要特别注意一下。

## 3.limit id

select \* from user where id > (select id from user limit 4000000, 1) limit 10;



这种查询方式就是先扫描4000010行，但是只取出id，然后再查询id大于这个值的前10条，这样虽然也是扫描了400多w行记录，由于id是主键，拥有者主键索引，所以查询 查询一个id的limit速度会快很多，我们可以对比一下一开始的limit 4000000，10，效率相差了3倍多。

## 4.延迟关联（个人推荐）

什么叫延迟关联，他让mysql扫描尽可能少的记录，获取到需要访问的记录后再根据关联列回到

原表查询需要的所有列，这样听起来是不是很拗口，我们用sql来实现一下。

select \* from user INNER JOIN( select id from user limit 4000000,10 ) as a USING(id)



我们可以看到这种查看的方式和第三种的效率差不多，但是当字段比较多，类型的长度比较长的是候，这种还是比较有优势的。

## 5.分表查询

mysql推荐一张表的存储不要超过500w数据，查询400w不到1秒对于一般的查询来说已经可以了，如果还要更快的话，我建议使用分表存储，分表又分两种情况，水平分表于垂直分表。

## 水平分表

假如一张表的原始数据有1000w条数据，我可分三张表存储，一张表300的万，这样查询的时候压力就会小很多，并且效率也很高很多，那问题来了，如何这个水平水表如何实现呢？像可以借助mycat之类的中间件，阿里云也提供了数据库的分表技术，当然，你也可以自己手写分表，但是自己手写分表的时候需要注意id重复以及如何定义搭配当前id在那张表中，算法推荐使用hash值。

## 垂直分表

假如张彪的记录有100w，按正常来说查询速度应该不会太慢，但是由于这张表的字段超多，而且还有很多text类型的字段，这个时候我们可以将占用空间比较小的字段分在一张表，占用空间比较大的字段分在另一张表，两张表一一关联，这样，查询的时候就会快很多了。

## 冷热表

这里我还有一种分表思想，可以借鉴一下，那就是冷热表。

什么时冷热表？大家都用银行的app吧，你们查询账单的时候会发现只能查询近几个月的数据，之前的数据需要去柜台获取在查询历史账单中查看，他这里就是冷热表的设计思想。

我们新建两张一模一样的表，一张表存放近三个月的记录（时间随情况而定，不一定时三个月） a表，另一张表存放三个月之前的数据：b表，用户产生的新记录可以存放在a表中，可以在每天凌晨的时候定时扫描a表，只要记录已经在三个月之前了，我们就可以将记录迁移到b表中，对于用户来说，查询近三个月的数据时他们比较敏感的，三个月之前的数据他们查询的可能并不多，所以这样的设计完全是合理的。

## 索引

这一点相信大家都知道，添加索引可以提高查询效率，如何我们的分页查询牵扯到条件的话，我们可以给条件添加索引，数据库会维护一张对应的索引表，查询的时候会先查询索引表，根据索引表返回的记录直接查询记录表，这样也减少了扫描的行数，但是需要注意，只要发生一下几点，索引都有可能不会被触发，一定要注意。

1.查询条件使用is not null。

2.like语句，比如 keyword like ‘%笔记本’，索引失效，%不能再最前面。

3.OR 前后条件中只要有一个没有添加索引，那么将扫描全表，索引失效。

4.组合索引：使用组合索引的时候必要要带上第一个索引的字段，否则组合索引不生效。

5. > 、< 、 <>。

6. 没有单引号的字符串。

## 总结

总而言之，查询优化的重点就在于如何能扫描最少的记录，返回查询的结果，看上去容易，但是真正做起来的时候会发现是那么的不容易，对于写后端的程序猿来说，sql是家常菜，也是必不可少的一道菜，因为sql写的好不好直接决定着你程序的抗压能力强不强，这个时候你可能会说我可以使用缓存来降低数据库的访问，这只是治标不治本，只有写出漂亮的sql才能让程序立于不败之地。至于文章开头说的8秒是因为同事的那种表比我的这张表复杂得多，并且还加上了查询总记录的时间，我这里并没有给出count()查询时间，500w的数据，count()差不多都要两秒，所以sount(\*)，查询总数也是需要优化的，这个优化就比limit的优化简单多了，这里就不多做说明了

# 分库分表方案

**一、数据库瓶颈**

不管是IO瓶颈，还是CPU瓶颈，最终都会导致数据库的活跃连接数增加，进而逼近甚至达到数据库可承载活跃连接数的阈值。在业务Service来看就是，可用数据库连接少甚至无连接可用。接下来就可以想象了吧（并发量、吞吐量、崩溃）。

**1、IO瓶颈**

第一种：磁盘读IO瓶颈，热点数据太多，数据库缓存放不下，每次查询时会产生大量的IO，降低查询速度 ->**分库和垂直分表。**

第二种：网络IO瓶颈，请求的数据太多，网络带宽不够 ->**分库。**

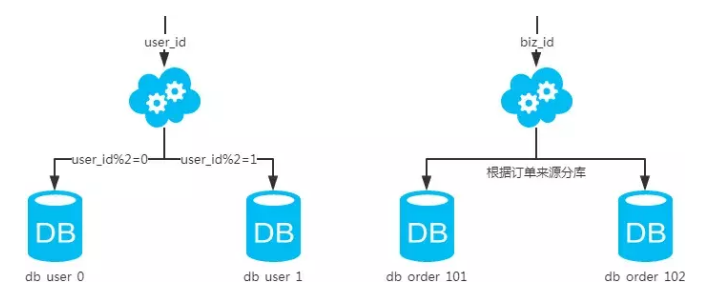
**2、CPU瓶颈**

第一种：SQL问题，如SQL中包含join，group by，order by，非索引字段条件查询等，增加CPU运算的操作 -> SQL优化，建立合适的索引，在业务Service层进行业务计算。

第二种：单表数据量太大，查询时扫描的行太多，SQL效率低，CPU率先出现瓶颈 -> **水平分表。**

**二、分库分表**

**1、水平分库**



1、概念：以字段为依据，按照一定策略（hash、range等），将一个库中的数据拆分到多个库中。

2、结果：

每个库的结构都一样；

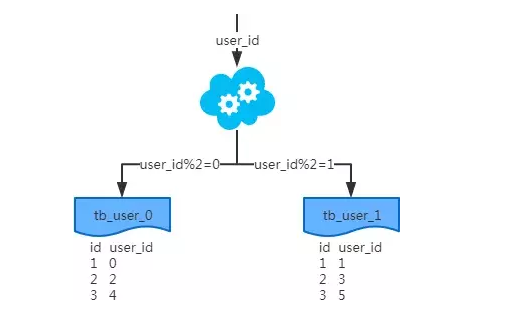
每个库的数据都不一样，没有交集；

所有库的并集是全量数据；

3、场景：系统绝对并发量上来了，分表难以根本上解决问题，并且还没有明显的业务归属来垂直分库。

4、分析：库多了，io和cpu的压力自然可以成倍缓解。

**2、水平分表**



1、概念：以字段为依据，按照一定策略（hash、range等），将一个表中的数据拆分到多个表中。

2、结果：

每个表的结构都一样；

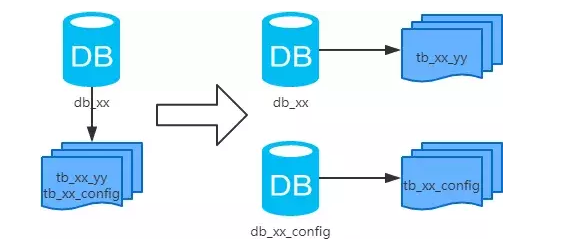
每个表的数据都不一样，没有交集；

所有表的并集是全量数据；

3、场景：系统绝对并发量并没有上来，只是单表的数据量太多，影响了SQL效率，加重了CPU负担，以至于成为瓶颈。

4、分析：表的数据量少了，单次SQL执行效率高，自然减轻了CPU的负担。

**3、垂直分库**



1、概念：以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。

2、结果：

每个库的结构都不一样；

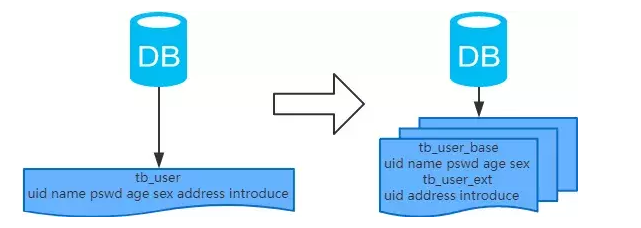
每个库的数据也不一样，没有交集；

所有库的并集是全量数据；

3、场景：系统绝对并发量上来了，并且可以抽象出单独的业务模块。

4、分析：到这一步，基本上就可以服务化了。例如，随着业务的发展一些公用的配置表、字典表等越来越多，这时可以将这些表拆到单独的库中，甚至可以服务化。再有，随着业务的发展孵化出了一套业务模式，这时可以将相关的表拆到单独的库中，甚至可以服务化。

**4、垂直分表**



1、概念：以字段为依据，按照字段的活跃性，将表中字段拆到不同的表（主表和扩展表）中。

2、结果：

每个表的结构都不一样；

每个表的数据也不一样，一般来说，每个表的字段至少有一列交集，一般是主键，用于关联数据；

所有表的并集是全量数据；

3、场景：系统绝对并发量并没有上来，表的记录并不多，但是字段多，并且热点数据和非热点数据在一起，单行数据所需的存储空间较大。以至于数据库缓存的数据行减少，查询时会去读磁盘数据产生大量的随机读IO，产生IO瓶颈。

4、分析：可以用列表页和详情页来帮助理解。垂直分表的拆分原则是将热点数据（可能会冗余经常一起查询的数据）放在一起作为主表，非热点数据放在一起作为扩展表。这样更多的热点数据就能被缓存下来，进而减少了随机读IO。拆了之后，要想获得全部数据就需要关联两个表来取数据。但记住，千万别用join，因为join不仅会增加CPU负担并且会讲两个表耦合在一起（必须在一个数据库实例上）。关联数据，应该在业务Service层做文章，分别获取主表和扩展表数据然后用关联字段关联得到全部数据。

**三、分库分表工具**

1、sharding-sphere：jar，前身是sharding-jdbc；

2、TDDL：jar，Taobao Distribute Data Layer；

3、Mycat：中间件。

注：工具的利弊，请自行调研，官网和社区优先。

**四、分库分表步骤**

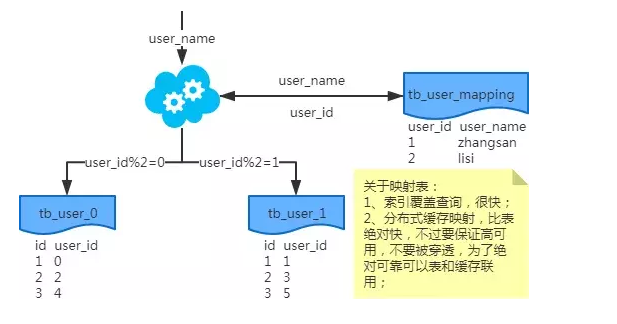
根据容量（当前容量和增长量）评估分库或分表个数 -> 选key（均匀）-> 分表规则（hash或range等）-> 执行（一般双写）-> 扩容问题（尽量减少数据的移动）。

**五、分库分表问题**

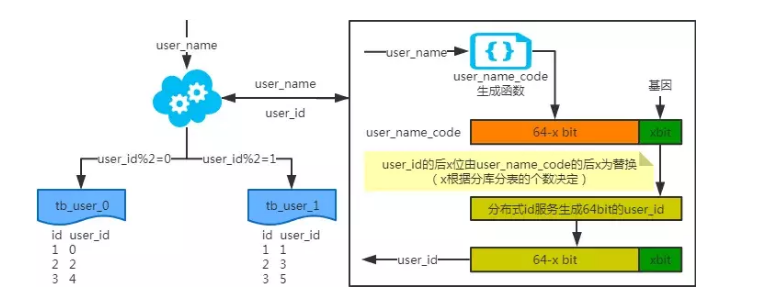
**1、非partition key的查询问题（水平分库分表，拆分策略为常用的hash法）**

1、端上除了partition key只有一个非partition key作为条件查询

* 映射法



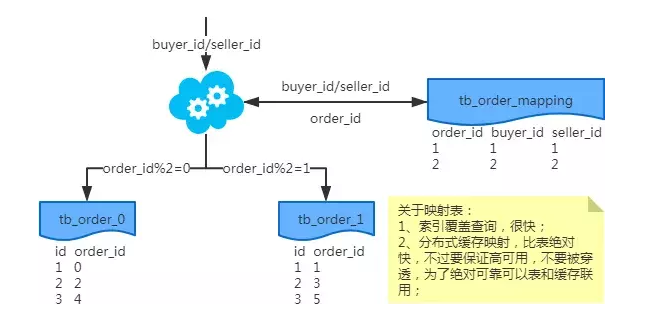
* 基因法



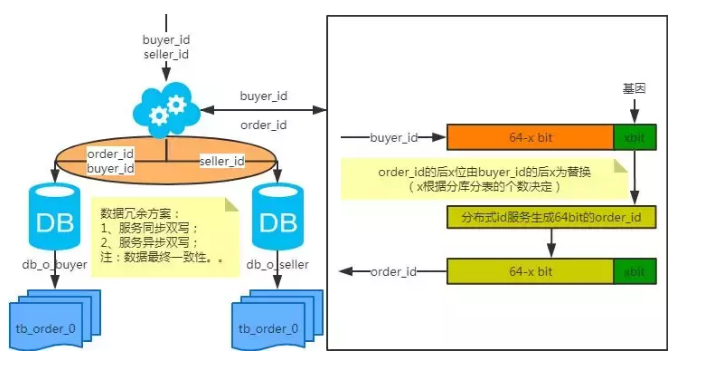
注：写入时，基因法生成user\_id，如图。关于xbit基因，例如要分8张表，23=8，故x取3，即3bit基因。根据user\_id查询时可直接取模路由到对应的分库或分表。根据user\_name查询时，先通过user\_name\_code生成函数生成user\_name\_code再对其取模路由到对应的分库或分表。id生成常用**snowflake算法。**

2、端上除了partition key不止一个非partition key作为条件查询

* 映射法



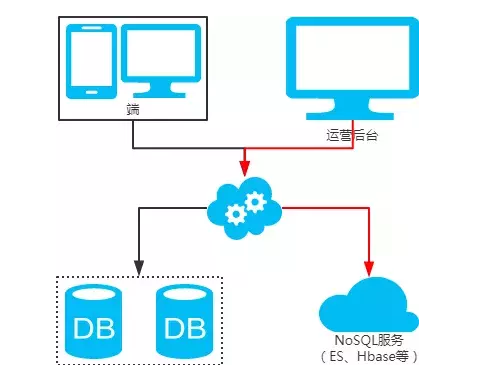
* 冗余法



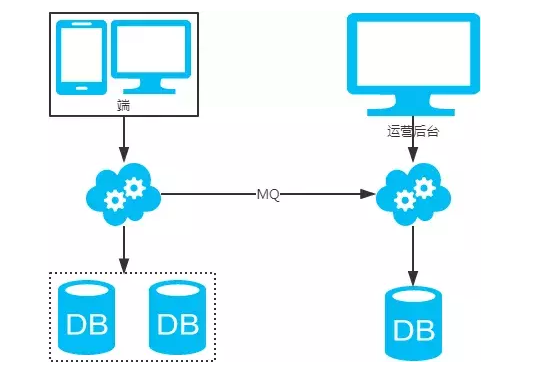
注：按照order\_id或buyer\_id查询时路由到db\_o\_buyer库中，按照seller\_id查询时路由到db\_o\_seller库中。感觉有点本末倒置！有其他好的办法吗？改变技术栈呢？

3、后台除了partition key还有各种非partition key组合条件查询

* NoSQL法



* 冗余法

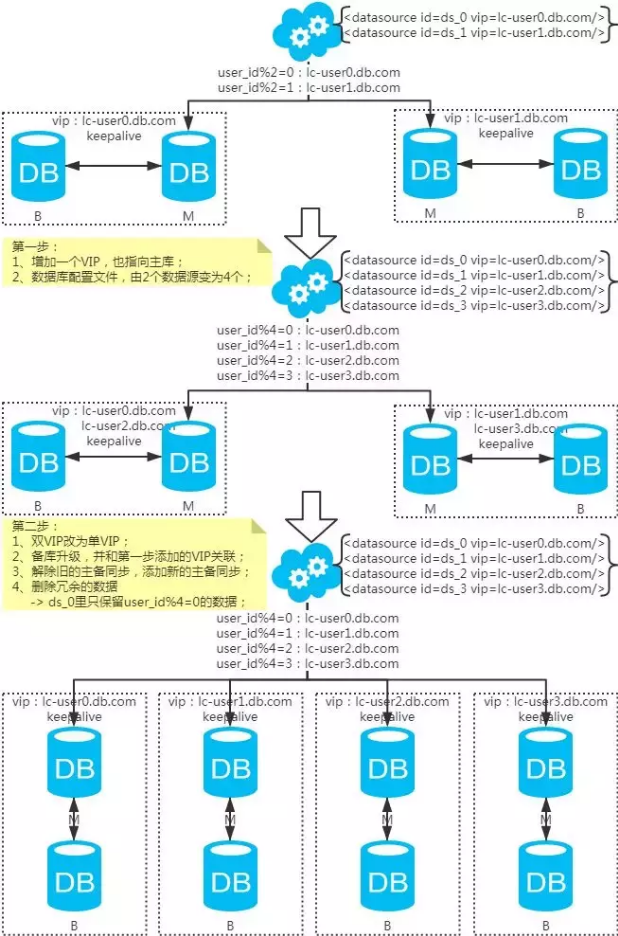


**2、非partition key跨库跨表分页查询问题（水平分库分表，拆分策略为常用的hash法）**

注：用NoSQL法解决（ES等）。

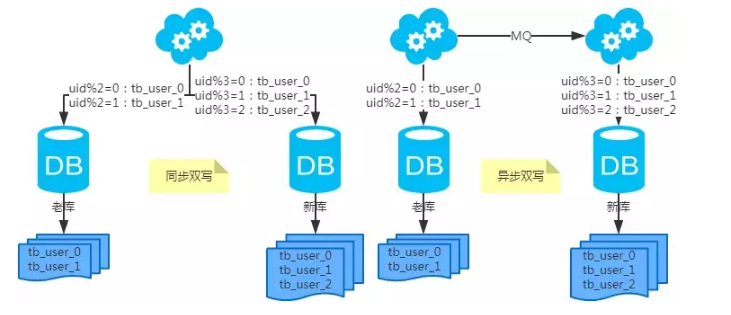
**3、扩容问题（水平分库分表，拆分策略为常用的hash法）**

1、水平扩容库（升级从库法



注：扩容是成倍的。

2、水平扩容表（双写迁移法）



第一步：（同步双写）应用配置双写，部署；

第二步：（同步双写）将老库中的老数据复制到新库中；

第三步：（同步双写）以老库为准校对新库中的老数据；

第四步：（同步双写）应用去掉双写，部署；

注：双写是通用方案。

**六、分库分表总结**

分库分表，首先得知道瓶颈在哪里，然后才能合理地拆分（分库还是分表？水平还是垂直？分几个？）。且不可为了分库分表而拆分。

1、选key很重要，既要考虑到拆分均匀，也要考虑到非partition key的查询。

2、只要能满足需求，拆分规则越简单越好。

**七、分库分表示例**

示例GitHub地址：https://github.com/littlecharacter4s/study-sharding