### 走一遍

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动

firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

# 体验一下面试官对于分库分表这个事儿的一个连环炮

1. 为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？

用过哪些分库分表中间件？

不同的分库分表中间件都有什么优点和缺点？

你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

（2）现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？

（3）如何设计可以动态扩容缩容的分库分表方案？

（4）分库分表之后，id主键如何处理？

# 来来来！咱们聊一下你们公司是怎么玩儿分库分表的？

为什么要分库分表（设计高并发系统的时候，数据库层面该如何设计）？

用过哪些分库分表中间件？

不同的分库分表中间件都有什么优点和缺点？

你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

其实这块肯定是扯到高并发了，因为分库分表一定是为了支撑高并发、数据量大两个问题的。而且现在说实话，尤其是互联网类的公司面试，基本上都会来这么一下，分库分表如此普遍的技术问题，不问实在是不行，而如果你不知道那也实在是说不过去！

### （1）为什么要分库分表？（设计高并发系统的时候，数据库层面该如何设计？）

说白了，分库分表是两回事儿，大家可别搞混了，可能是光分库不分表，也可能是光分表不分库，都有可能。我先给大家抛出来一个场景。

假如我们现在是一个小创业公司（或者是一个BAT公司刚兴起的一个新部门），现在注册用户就20万，每天活跃用户就1万，每天单表数据量就1000，然后高峰期每秒钟并发请求最多就10。。。天，就这种系统，随便找一个有几年工作经验的，然后带几个刚培训出来的，随便干干都可以。

结果没想到我们运气居然这么好，碰上个CEO带着我们走上了康庄大道，业务发展迅猛，过了几个月，注册用户数达到了2000万！每天活跃用户数100万！每天单表数据量10万条！高峰期每秒最大请求达到1000！同时公司还顺带着融资了两轮，紧张了几个亿人民币啊！公司估值达到了惊人的几亿美金！这是小独角兽的节奏！

好吧，没事，现在大家感觉压力已经有点大了，为啥呢？因为每天多10万条数据，一个月就多300万条数据，现在咱们单表已经几百万数据了，马上就破千万了。但是勉强还能撑着。高峰期请求现在是1000，咱们线上部署了几台机器，负载均衡搞了一下，数据库撑1000 QPS也还凑合。但是大家现在开始感觉有点担心了，接下来咋整呢。。。。。。

再接下来几个月，我的天，CEO太牛逼了，公司用户数已经达到1亿，公司继续融资几十亿人民币啊！公司估值达到了惊人的几十亿美金，成为了国内今年最牛逼的明星创业公司！天，我们太幸运了。

但是我们同时也是不幸的，因为此时每天活跃用户数上千万，每天单表新增数据多达50万，目前一个表总数据量都已经达到了两三千万了！扛不住啊！数据库磁盘容量不断消耗掉！高峰期并发达到惊人的5000~8000！别开玩笑了，哥。我跟你保证，你的系统支撑不到现在，已经挂掉了！

好吧，所以看到你这里你差不多就理解分库分表是怎么回事儿了，实际上这是跟着你的公司业务发展走的，你公司业务发展越好，用户就越多，数据量越大，请求量越大，那你单个数据库一定扛不住。

比如你单表都几千万数据了，你确定你能抗住么？绝对不行，单表数据量太大，会极大影响你的sql执行的性能，到了后面你的sql可能就跑的很慢了。一般来说，就以我的经验来看，单表到几百万的时候，性能就会相对差一些了，你就得分表了。

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2000，一定要扩容了，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，为啥要分库分表？你明白了吧

### （2）用过哪些分库分表中间件？不同的分库分表中间件都有什么优点和缺点？

这个其实就是看看你了解哪些分库分表的中间件，各个中间件的优缺点是啥？然后你用过哪些分库分表的中间件。

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。

我们，数据库中间件都是自研的，也用过proxy层，后来也用过client层

### （3）你们具体是如何对数据库如何进行垂直拆分或水平拆分的？

水平拆分的意思，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。

垂直拆分的意思，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。

好了，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。

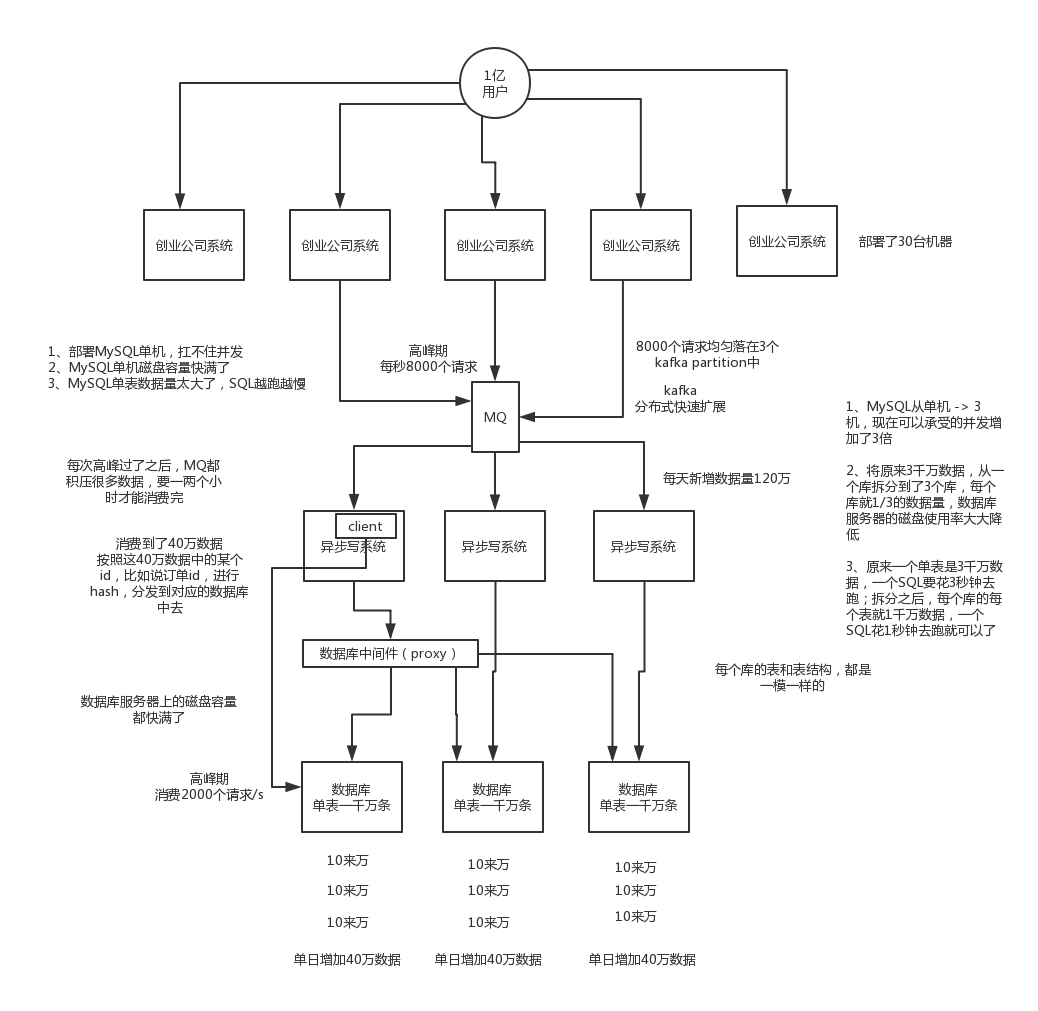
你就得考虑一下，你的项目里该如何分库分表？一般来说，垂直拆分，你可以在表层面来做，对一些字段特别多的表做一下拆分；水平拆分，你可以说是并发承载不了，或者是数据量太大，容量承载不了，你给拆了，按什么字段来拆，你自己想好；分表，你考虑一下，你如果哪怕是拆到每个库里去，并发和容量都ok了，但是每个库的表还是太大了，那么你就分表，将这个表分开，保证每个表的数据量并不是很大。

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。

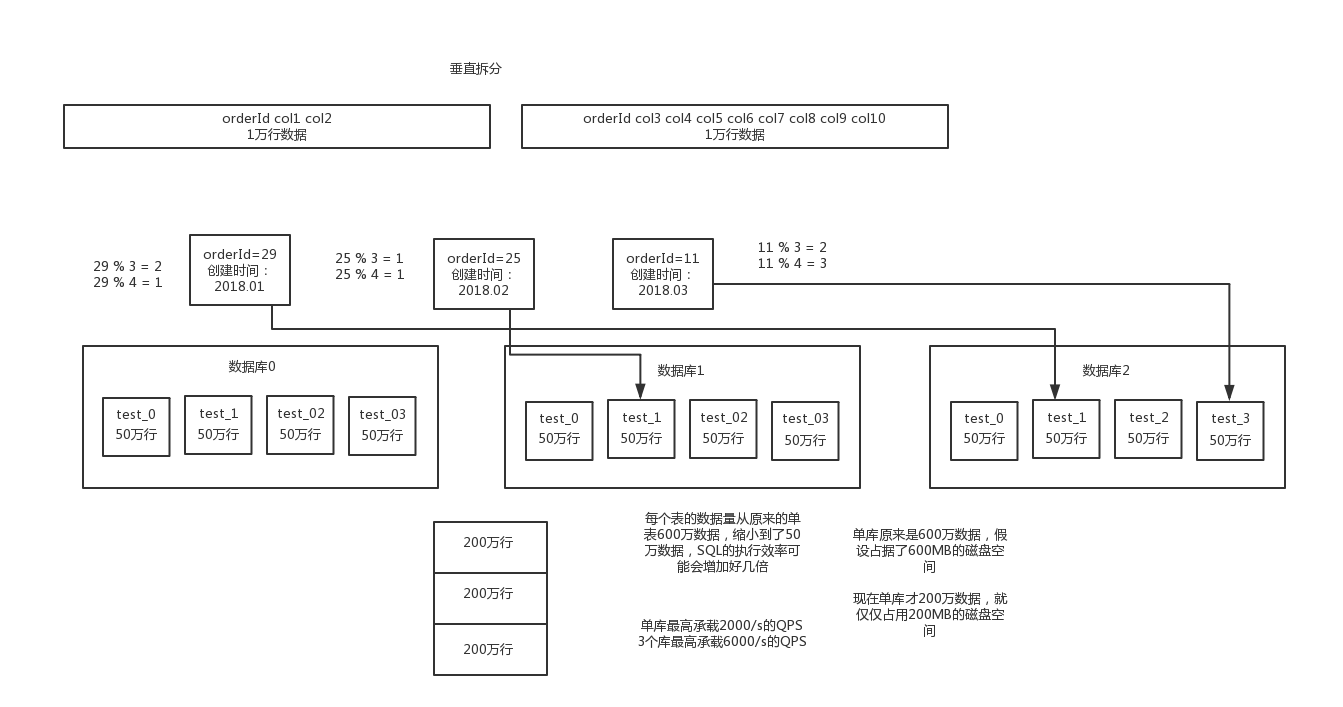
range来分，好处在于说，后面扩容的时候，就很容易，因为你只要预备好，给每个月都准备一个库就可以了，到了一个新的月份的时候，自然而然，就会写新的库了；缺点，但是大部分的请求，都是访问最新的数据。实际生产用range，要看场景，你的用户不是仅仅访问最新的数据，而是均匀的访问现在的数据以及历史的数据

hash分法，好处在于说，可以平均分配没给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

01\_分库分表的由来



02\_数据库如何拆分



# 你们当时是如何把系统不停机迁移到分库分表的？

现在有一个未分库分表的系统，未来要分库分表，如何设计才可以让系统从未分库分表动态切换到分库分表上？

你看看，你现在已经明白为啥要分库分表了，你也知道常用的分库分表中间件了，你也设计好你们如何分库分表的方案了（水平拆分、垂直拆分、分表），那问题来了，你接下来该怎么把你那个单库单表的系统给迁移到分库分表上去？

所以这都是一环扣一环的，就是看你有没有全流程经历过这个过程

假设，你现有有一个单库单表的系统，在线上在跑，假设单表有600万数据

3个库，每个库里分了4个表，每个表要放50万的数据量

假设你已经选择了一个分库分表的数据库中间件，sharding-jdbc，mycat，都可以

你怎么把线上系统平滑地迁移到分库分表上面去

sharding-jdbc：自己上官网，找一个官网最基本的例子，自己写一下，试一下，跑跑看，是非常简单的

mycat：自己上官网，找一个官网最基本的例子，自己写一下，试一下看看

1个小时以内就可以搞定了

# 这个其实从low到高大上有好几种方案，我们都玩儿过，我都给你说一下

### （1）停机迁移方案

我先给你说一个最low的方案，就是很简单，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，无法访问。。。。。。

接着到0点，停机，系统挺掉，没有流量写入了，此时老的单库单表数据库静止了。然后你之前得写好一个导数的一次性工具，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。

验证一下，ok了，完美，大家伸个懒腰，看看看凌晨4点钟的北京夜景，打个滴滴回家吧

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

这个是我们常用的一种迁移方案，比较靠谱一些，不用停机，不用看北京凌晨4点的风景

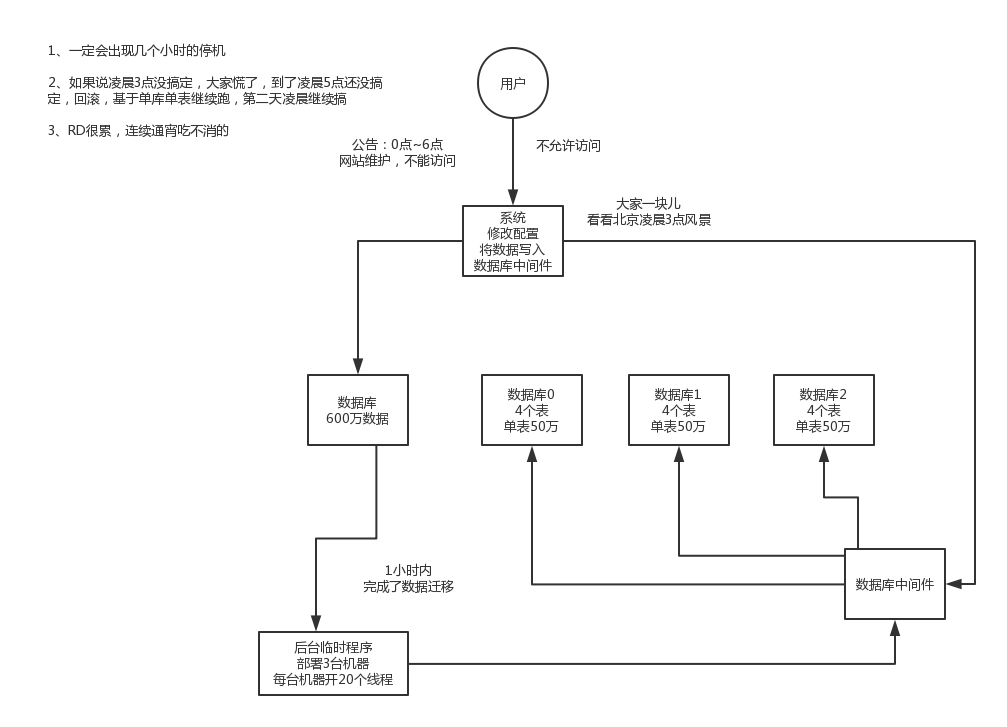
简单来说，就是在线上系统里面，之前所有写库的地方，增删改操作，都除了对老库增删改，都加上对新库的增删改，这就是所谓双写，同时写俩库，老库和新库。

然后系统部署之后，新库数据差太远，用之前说的导数工具，跑起来读老库数据写新库，写的时候要根据gmt\_modified这类字段判断这条数据最后修改的时间，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。

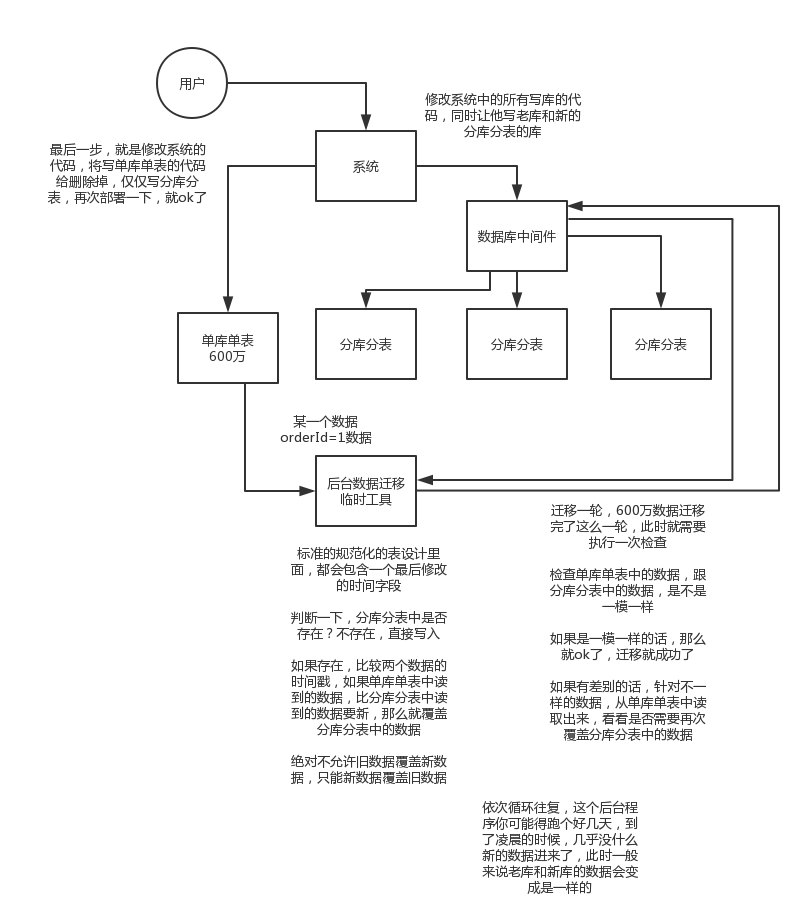
接着导完一轮之后，有可能数据还是存在不一致，那么就程序自动做一轮校验，比对新老库每个表的每条数据，接着如果有不一样的，就针对那些不一样的，从老库读数据再次写。反复循环，直到两个库每个表的数据都完全一致为止。

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。

01\_长时间停机分库分表



02\_不停机双写方案



# 好啊！那如何设计可以动态扩容缩容的分库分表方案？

如何设计可以动态扩容缩容的分库分表方案？

2、面试官心里分析

（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？

这个是你必须面对的一个事儿，就是你已经弄好分库分表方案了，然后一堆库和表都建好了，基于分库分表中间件的代码开发啥的都好了，测试都ok了，数据能均匀分布到各个库和各个表里去，而且接着你还通过双写的方案咔嚓一下上了系统，已经直接基于分库分表方案在搞了。

那么现在问题来了，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。

这都是玩儿分库分表线上必须经历的事儿

### （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。但是最好别这么玩儿，有点不太靠谱，因为既然分库分表就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。

从单库单表迁移到分库分表的时候，数据量并不是很大，单表最大也就两三千万

写个工具，多弄几台机器并行跑，1小时数据就导完了

3个库+12个表，跑了一段时间了，数据量都1亿~2亿了。光是导2亿数据，都要导个几个小时，6点，刚刚导完数据，还要搞后续的修改配置，重启系统，测试验证，10点才可以搞完

### （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

我可以告诉各位同学说，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。

有些除非是国内排名非常靠前的这些公司，他们的最核心的系统的数据库，可能会出现几百台数据库的这么一个规模，128个库，256个库，512个库

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

谈分库分表的扩容，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

一个实践是利用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。根据某个id先根据32取模路由到库，再根据32取模路由到库里的表。

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。

这么搞，是不用自己写代码做数据迁移的，都交给dba来搞好了，但是dba确实是需要做一些库表迁移的工作，但是总比你自己写代码，抽数据导数据来的效率高得多了。

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。

对2 ^ n取模

orderId 模 32 = 库

orderId / 32 模 32 = 表

259 3 8

1189 5 5

352 0 11

4593 17 15

1、设定好几台数据库服务器，每台服务器上几个库，每个库多少个表，推荐是32库 \* 32表，对于大部分公司来说，可能几年都够了

2、路由的规则，orderId 模 32 = 库，orderId / 32 模 32 = 表

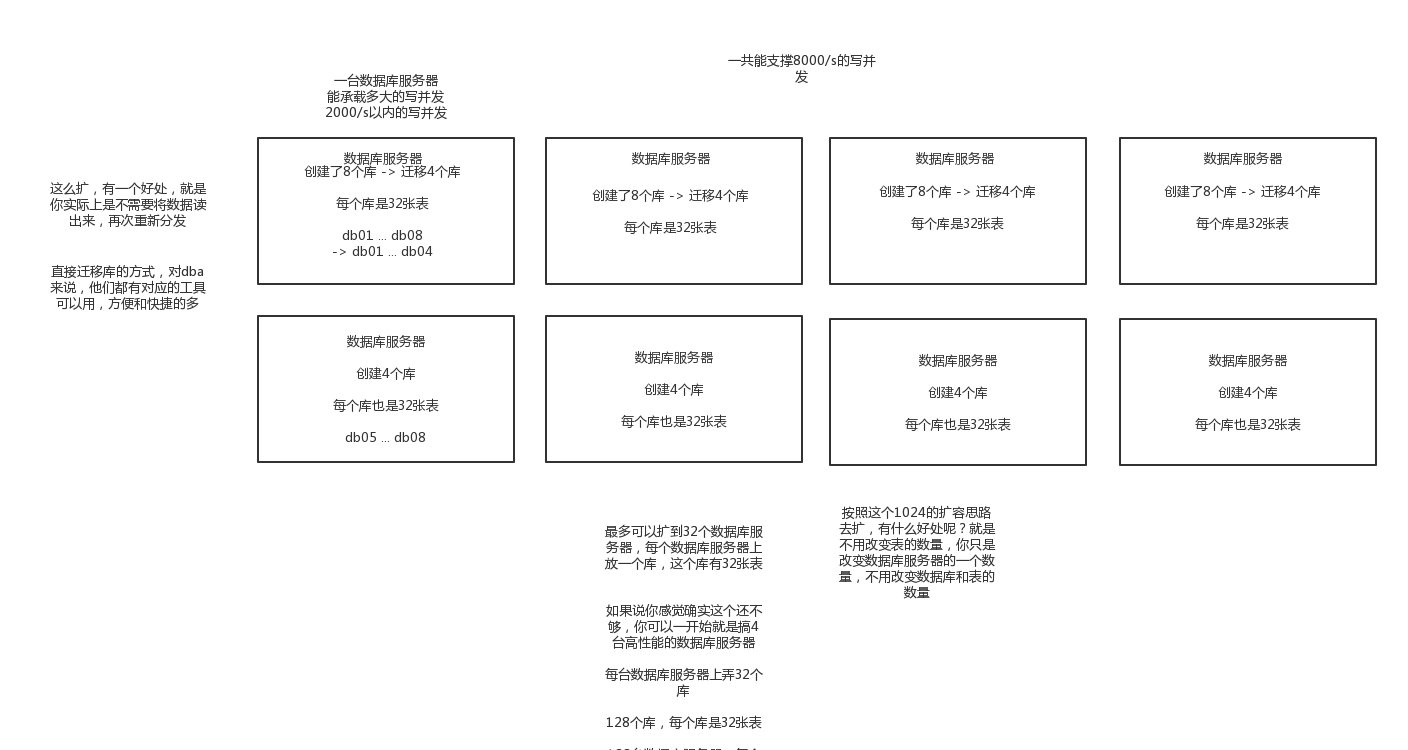
3、扩容的时候，申请增加更多的数据库服务器，装好mysql，倍数扩容，4台服务器，扩到8台服务器，16台服务器

4、由dba负责将原先数据库服务器的库，迁移到新的数据库服务器上去，很多工具，库迁移，比较便捷

5、我们这边就是修改一下配置，调整迁移的库所在数据库服务器的地址

6、重新发布系统，上线，原先的路由规则变都不用变，直接可以基于2倍的数据库服务器的资源，继续进行线上系统的提供服务

01\_分库分表扩容方案



# 一个关键的问题！分库分表之后全局id咋生成？

分库分表之后，id主键如何处理？

其实这是分库分表之后你必然要面对的一个问题，就是id咋生成？因为要是分成多个表之后，每个表都是从1开始累加，那肯定不对啊，需要一个全局唯一的id来支持。所以这都是你实际生产环境中必须考虑的问题。

### （1）数据库自增id

这个就是说你的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；如果你硬是要改进一下，那么就专门开一个服务出来，这个服务每次就拿到当前id最大值，然后自己递增几个id，一次性返回一批id，然后再把当前最大id值修改成递增几个id之后的一个值；但是无论怎么说都是基于单个数据库。

适合的场景：你分库分表就俩原因，要不就是单库并发太高，要不就是单库数据量太大；除非是你并发不高，但是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。

并发很低，几百/s，但是数据量大，几十亿的数据，所以需要靠分库分表来存放海量的数据

### （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。

UUID.randomUUID().toString().replace(“-”, “”) -> sfsdf23423rr234sfdaf

### （3）获取系统当前时间

这个就是获取当前时间即可，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。

适合的场景：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

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

1 bit：不用，为啥呢？因为二进制里第一个bit为如果是1，那么都是负数，但是我们生成的id都是正数，所以第一个bit统一都是0

41 bit：表示的是时间戳，单位是毫秒。41 bit可以表示的数字多达2^41 - 1，也就是可以标识2 ^ 41 - 1个毫秒值，换算成年就是表示69年的时间。

10 bit：记录工作机器id，代表的是这个服务最多可以部署在2^10台机器上哪，也就是1024台机器。但是10 bit里5个bit代表机房id，5个bit代表机器id。意思就是最多代表2 ^ 5个机房（32个机房），每个机房里可以代表2 ^ 5个机器（32台机器）。

12 bit：这个是用来记录同一个毫秒内产生的不同id，12 bit可以代表的最大正整数是2 ^ 12 - 1 = 4096，也就是说可以用这个12bit代表的数字来区分同一个毫秒内的4096个不同的id

64位的long型的id，64位的long -> 二进制

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

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，会判断一下，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，此时我只能把加1

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000001

比如我们来观察上面的那个，就是一个典型的二进制的64位的id，换算成10进制就是910499571847892992。

public class IdWorker{

private long workerId;

private long datacenterId;

private long sequence;

public IdWorker(long workerId, long datacenterId, long sequence){

// sanity check for workerId

// 这儿不就检查了一下，要求就是你传递进来的机房id和机器id不能超过32，不能小于0

if (workerId > maxWorkerId || workerId < 0) {

throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0",maxWorkerId));

}

if (datacenterId > maxDatacenterId || datacenterId < 0) {

throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0",maxDatacenterId));

}

System.out.printf("worker starting. timestamp left shift %d, datacenter id bits %d, worker id bits %d, sequence bits %d, workerid %d",

timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId);

this.workerId = workerId;

this.datacenterId = datacenterId;

this.sequence = sequence;

}

private long twepoch = 1288834974657L;

private long workerIdBits = 5L;

private long datacenterIdBits = 5L;

private long maxWorkerId = -1L ^ (-1L << workerIdBits); // 这个是二进制运算，就是5 bit最多只能有31个数字，也就是说机器id最多只能是32以内

private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits); // 这个是一个意思，就是5 bit最多只能有31个数字，机房id最多只能是32以内

private long sequenceBits = 12L;

private long workerIdShift = sequenceBits;

private long datacenterIdShift = sequenceBits + workerIdBits;

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

private long sequenceMask = -1L ^ (-1L << sequenceBits);

private long lastTimestamp = -1L;

public long getWorkerId(){

return workerId;

}

public long getDatacenterId(){

return datacenterId;

}

public long getTimestamp(){

return System.currentTimeMillis();

}

public synchronized long nextId() {

// 这儿就是获取当前时间戳，单位是毫秒

long timestamp = timeGen();

if (timestamp < lastTimestamp) {

System.err.printf("clock is moving backwards. Rejecting requests until %d.", lastTimestamp);

throw new RuntimeException(String.format("Clock moved backwards. Refusing to generate id for %d milliseconds",

lastTimestamp - timestamp));

}

// 0

// 在同一个毫秒内，又发送了一个请求生成一个id，0 -> 1

if (lastTimestamp == timestamp) {

sequence = (sequence + 1) & sequenceMask; // 这个意思是说一个毫秒内最多只能有4096个数字，无论你传递多少进来，这个位运算保证始终就是在4096这个范围内，避免你自己传递个sequence超过了4096这个范围

if (sequence == 0) {

timestamp = tilNextMillis(lastTimestamp);

}

} else {

sequence = 0;

}

// 这儿记录一下最近一次生成id的时间戳，单位是毫秒

lastTimestamp = timestamp;

// 这儿就是将时间戳左移，放到41 bit那儿；将机房id左移放到5 bit那儿；将机器id左移放到5 bit那儿；将序号放最后10 bit；最后拼接起来成一个64 bit的二进制数字，转换成10进制就是个long型

return ((timestamp - twepoch) << timestampLeftShift) |

(datacenterId << datacenterIdShift) |

(workerId << workerIdShift) |

sequence;

}

0 | 0001100 10100010 10111110 10001001 01011100 00 | 10001 | 1 1001 | 0000 00000000

private long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {

timestamp = timeGen();

}

return timestamp;

}

private long timeGen(){

return System.currentTimeMillis();

}

//---------------测试---------------

public static void main(String[] args) {

IdWorker worker = new IdWorker(1,1,1);

for (int i = 0; i < 30; i++) {

System.out.println(worker.nextId());

}

}

}

怎么说呢，大概这个意思吧，就是说41 bit，就是当前毫秒单位的一个时间戳，就这意思；然后5 bit是你传递进来的一个机房id（但是最大只能是32以内），5 bit是你传递进来的机器id（但是最大只能是32以内），剩下的那个10 bit序列号，就是如果跟你上次生成id的时间还在一个毫秒内，那么会把顺序给你累加，最多在4096个序号以内。

所以你自己利用这个工具类，自己搞一个服务，然后对每个机房的每个机器都初始化这么一个东西，刚开始这个机房的这个机器的序号就是0。然后每次接收到一个请求，说这个机房的这个机器要生成一个id，你就找到对应的Worker，生成。

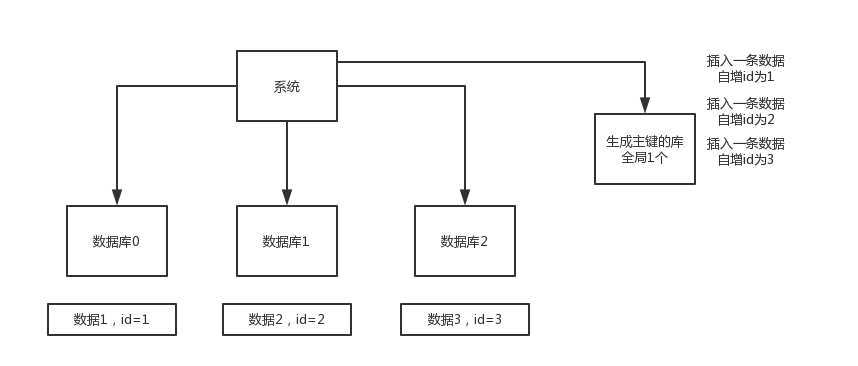
他这个算法生成的时候，会把当前毫秒放到41 bit中，然后5 bit是机房id，5 bit是机器id，接着就是判断上一次生成id的时间如果跟这次不一样，序号就自动从0开始；要是上次的时间跟现在还是在一个毫秒内，他就把seq累加1，就是自动生成一个毫秒的不同的序号。

这个算法那，可以确保说每个机房每个机器每一毫秒，最多生成4096个不重复的id。

利用这个snowflake算法，你可以开发自己公司的服务，甚至对于机房id和机器id，反正给你预留了5 bit + 5 bit，你换成别的有业务含义的东西也可以的。

这个snowflake算法相对来说还是比较靠谱的，所以你要真是搞分布式id生成，如果是高并发啥的，那么用这个应该性能比较好，一般每秒几万并发的场景，也足够你用了。

01\_分库分表的id主键问题



02\_snowflake算法

