总的目录：

https://www.processon.com/view/link/60fb9421637689719d246739

百亿级库表的容量架构与流量架构

数据库服务器的参考能力

一台数据库服务器能够承受多大的并发量、数据量 ，受内外两方面因素影响。

内在因素

搞清楚需要估算的数据库服务器是什么配置：

1. 确定数据库是MySQL还是Oracle亦或是DB2、PostgreSQL等；
2. CPU是几核？现代数据库应用都充分的运用了多核CPU的并行处理能力；
3. 内存多大？数据库的索引数据、缓存数据都会进入内存中；
4. 磁盘IO能力：数据库文件都存储在磁盘中，所以磁盘的IO能力将是影响数据库性能的最直接因素；
5. 网络带宽：网络的上行和下行带宽，数据库服务器可支持的最大连接数是多少。

外在因素

无论是应用程序开发还是SQL查询、操作都遵循“天下武功，唯快不破”的道理。更快意味着服务器资源的快速释放，以便CPU能继续处理其他的任务请求。

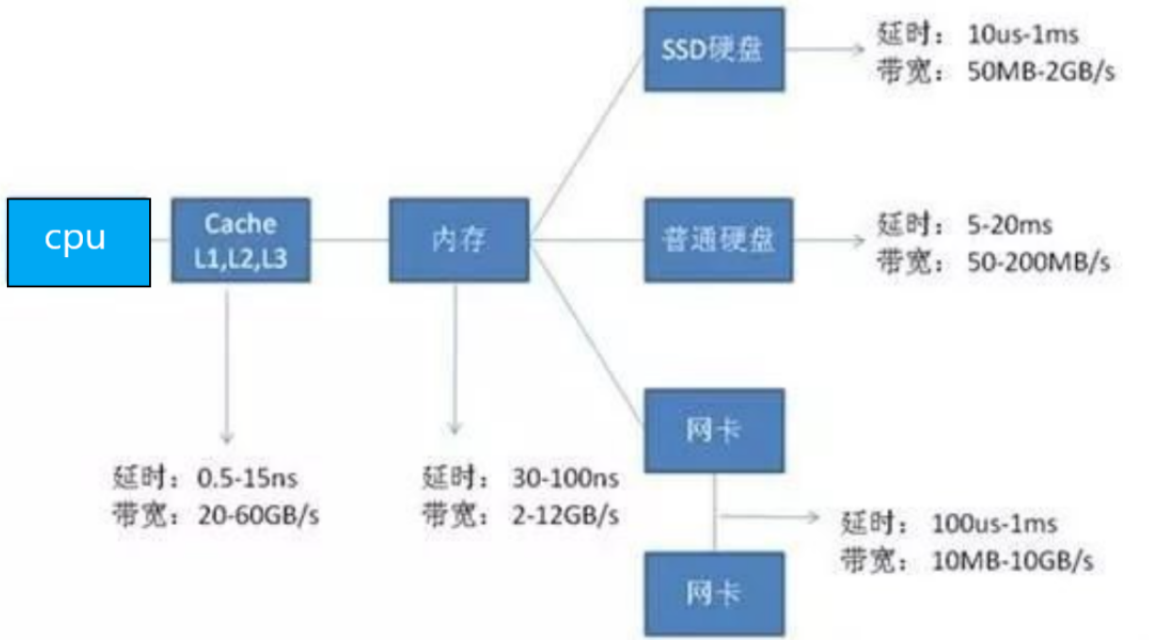
做评估数据库的并发量的时候，需要结合使用数据库的程序，否则数据库服务器性能再好，也是属于纸上谈兵。

做数据库并发量评估最好的办法是做数据库压力测试，压力测试的时候，我们可以一点一点的加压力，逐步的得出数据库的：

• QPS：Queries Per Second 每秒处理的查询数（如果是数据库，就相当于读取）

• TPS：Transactions Per Second 每秒处理的事务数(如果是数据库，就相当于写入、修改)

• IOPS：每秒磁盘进行的I/O操作次数



单表的参考数据量

一般来说电商的日订单都是百千万级甚至是亿万级别的了，小小的数据库肯定是撑不住的，这时候就要提前考虑分库分表了。

国内一般大厂规则参考：

- 单表500万条记录，正常水平
- 800万条警戒线
- 1000万条必须要分库分表

> 单表太大，怎么办？ 方法之一：减少请求数量； 方法之二： 分库分表。

单库的流量峰值并发能力

1500qps

为什么要分库分表?

当数据库产生性能瓶颈：IO瓶颈或CPU瓶颈。两种瓶颈最终都会导致数据库的活跃连接数增加，进而达到数据库可承受的最大活跃连接数阈值。终会导致应用服务无连接可用，造成灾难性后果。可以先从代码，sql，索引几方面进行优化。如果这几方面已经没有太多优化的余地，就该考虑分库分表了。

IO瓶颈

- 磁盘读IO瓶颈。由于热点数据太多，数据库缓存完全放不下，查询时会产生大量的磁盘IO，查询速度会比较慢，这样会导致产生大量活跃连接，最终可能会发展成无连接可用的后果。可以采用一主多从，读写分离的方案，用多个从库分摊查询流量。或者采用分库+水平分表（把一张表的数据拆成多张表来存放，比如订单表可以按user_id来拆分）的方案。
- 第二种：磁盘写IO瓶颈。由于数据库写入频繁，会产生频繁的磁盘写入IO操作，频繁的磁盘IO操作导致产生大量活跃连接，最终同样会发展成无连接可用的后果。这时只能采用分库方案，用多个库来分摊写入压力。再加上水平分表的策略，分表后，单表存储的数据量会更小，插入数据时索引查找和更新的成本会更低，插入速度自然会更快。

CPU瓶颈

- SQL问题。如果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，会对CPU产生明显的压力。 这时可以考虑SQL优化，创建适当的索引，也可以把一些计算量大的SQL逻辑放到应用中处理。
- 单表数据量太大。由于单张表数据量过大，比如超过1000W，查询时遍历树的层次太深或者扫描的行太多，SQL效率会很低，也会非常消耗CPU。这时可以根据业务场景水平分表。

分表

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。

分库

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，较为合理是1500。

这就是所谓的**分库分表**，为啥要分库分表?

场景	分库分表前	分库分表后
并发支撑情况	MySQL 单机部署，扛不住高并发	MySQL从单机到多机，能承受的并发增加了多倍
磁盘使用情况	MySQL 单机磁盘容量几乎撑满	拆分为多个库，数据库服务器磁盘使用率大大降低
SQL 执行性能	单表数据量太大，SQL 越跑越慢	单表数据量减少，SQL 执行效率明显提升

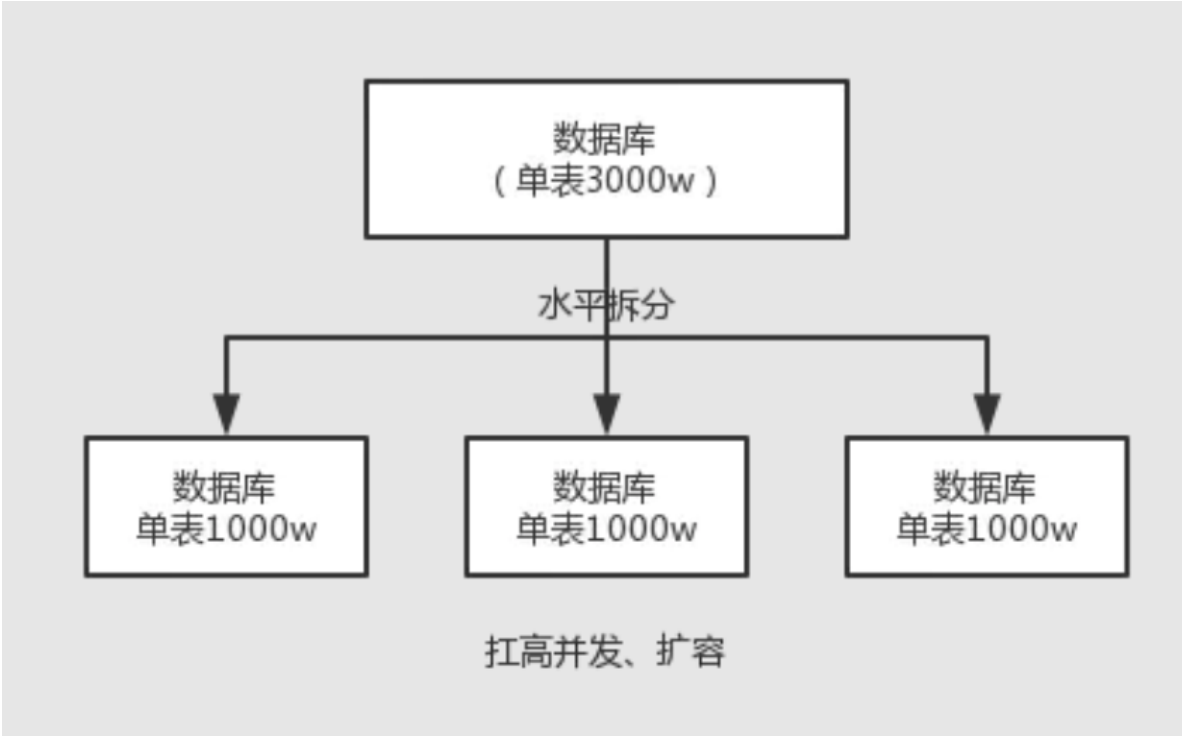
分表：水平拆分与垂直拆分

水平拆分

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。

> 目标：从扩展的角度，提高并发能力。



水平拆分一般是分库和分表的结合。水平拆分，就是**让每个表的row 数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500 万行左右， SQL越复杂，就最好让单表行数越少。

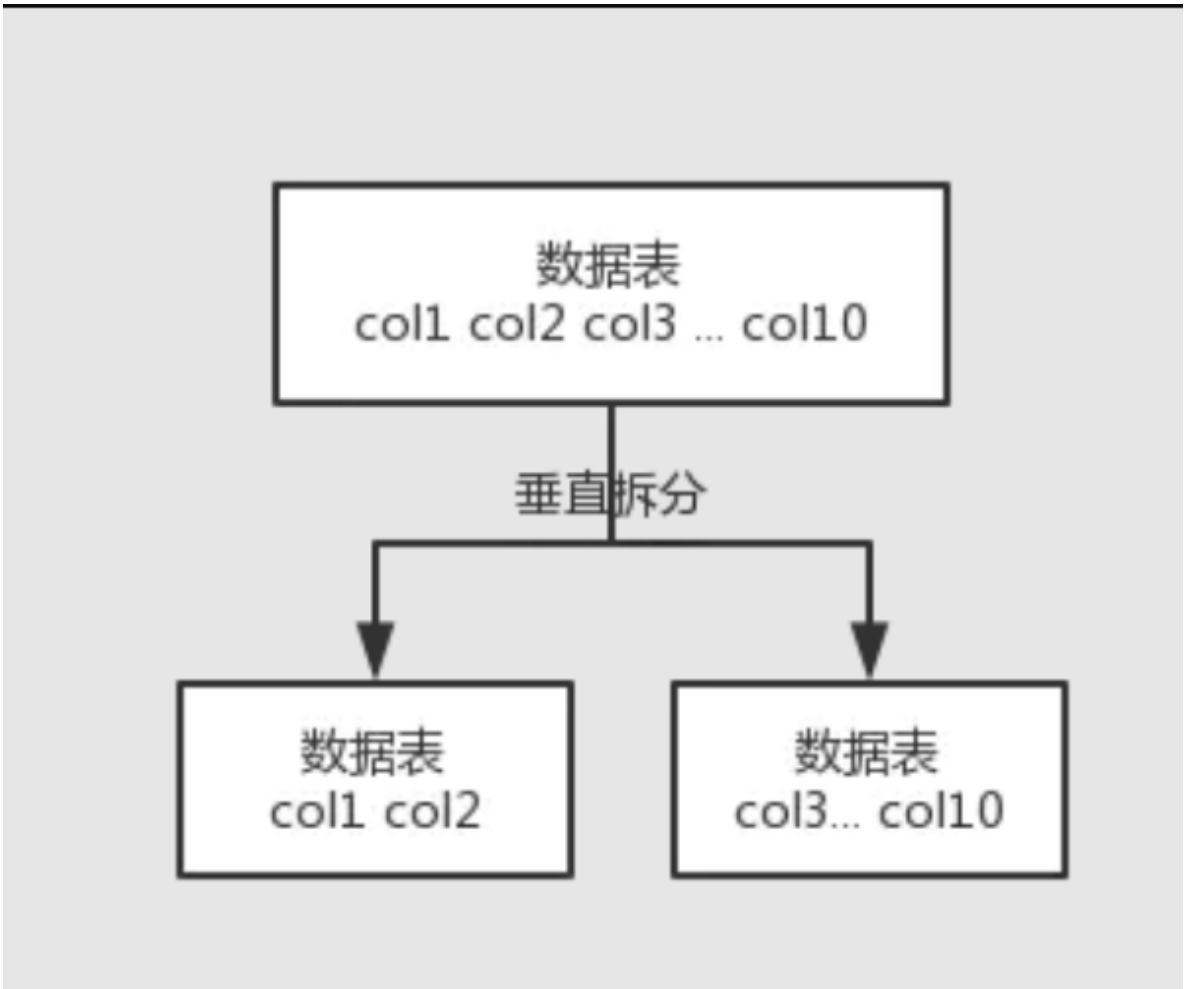
垂直拆分

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。

> 目标：从充分利用数据库缓存的角度，提升性能。



例如，把一个大的订单表拆开，订单表、订单支付表、订单商品表。

水平拆分时数据分片的方式

数据分片的**主要的方式**：

range 分片

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，因为很容易发生**数据倾斜**，大量的流量都打在**最新的数据**上了。

hash 分片

一是按照某个字段hash一下，分到不同的表， 均匀分散。

hash 分发，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过程，之前的数据需要重新计算 hash 值重新分配到不同的库或表。

ID取模分片

此种分片规则将数据分成n份（通常dn节点也为n），从而将数据均匀的分布于各个表中，或者各节点上。

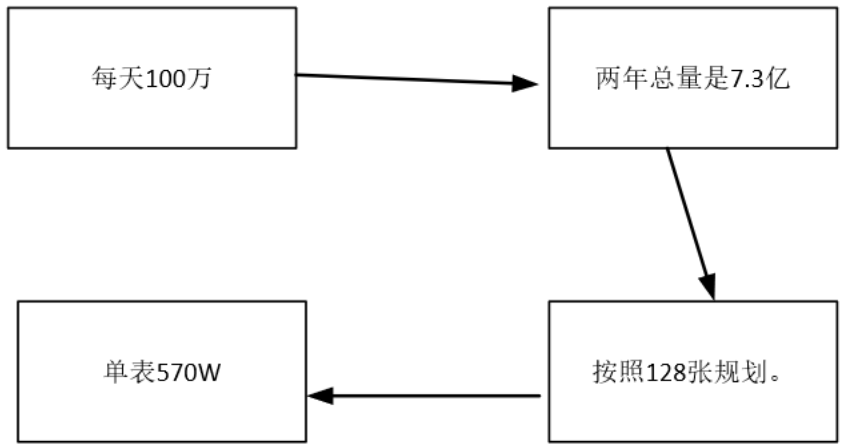
扩容方便。

亿级库表架构设计

表的数量规划

即便按照**每天100万，2年内保持稳定的要求**，进行表的数据量规划：

> 两年数据记录总量是7.3亿（每天100万*730天）
>
> , 假设每张表的标准值为500W，表的数据量平均是146张，换成2的幂，比较接近128张，或者说，单表570W（570=7.3亿/128），也是可以接受的，所以就按照128张规划。



库的数量规划

即便按照QPS峰值**1W的要求**，进行表的库的规划：

> 每个库正常承载的写入并发量是 1500，那么8个库就可以承载8 * 1500 = 12000 的写并发

悲观一点，如果每秒写入超过 5WQPS，可以通过**MQ削峰+批写入**的降级策略，MQ的写入吞吐量，可以轻松到达10W级别。

总的规划

利用 16 * 8 来分库分表，即分为8 个库，每个库里一个表分为 16 张表。一共就是 128 张表。根据某个 id 先根据 8 取模路由到库，再根据16取模路由到库里的表。

Id	id % 8 (库)	id /8 % 16 (表)
1095	7	8
1096	0	9
1097	1	9
1098	2	9

做一个测试

```
    @Test
    public void testmod() {
        for (int i = 0; i < 128; i++) {
            int id=1000+i;
            System.out.print("id = " + id);
            System.out.print("  id%8 = " + id%8 );
            System.out.print("   id/8%16 = " + id/8%16 );
            System.out.println(" " );
        }
    }
```

结果如下:

```
id = 1088  id%8 = 0   id/8%16 = 8
id = 1089  id%8 = 1   id/8%16 = 8
id = 1090  id%8 = 2   id/8%16 = 8
id = 1091  id%8 = 3   id/8%16 = 8
id = 1092  id%8 = 4   id/8%16 = 8
id = 1093  id%8 = 5   id/8%16 = 8
id = 1094  id%8 = 6   id/8%16 = 8
id = 1095  id%8 = 7   id/8%16 = 8
id = 1096  id%8 = 0   id/8%16 = 9
id = 1097  id%8 = 1   id/8%16 = 9
id = 1098  id%8 = 2   id/8%16 = 9
id = 1099  id%8 = 3   id/8%16 = 9
id = 1100  id%8 = 4   id/8%16 = 9
id = 1101  id%8 = 5   id/8%16 = 9
id = 1102  id%8 = 6   id/8%16 = 9
id = 1103  id%8 = 7   id/8%16 = 9
id = 1104  id%8 = 0   id/8%16 = 10
id = 1105  id%8 = 1   id/8%16 = 10
id = 1106  id%8 = 2   id/8%16 = 10
id = 1107  id%8 = 3   id/8%16 = 10
id = 1108  id%8 = 4   id/8%16 = 10
id = 1109  id%8 = 5   id/8%16 = 10
id = 1110  id%8 = 6   id/8%16 = 10
id = 1111  id%8 = 7   id/8%16 = 10
id = 1112  id%8 = 0   id/8%16 = 11
```

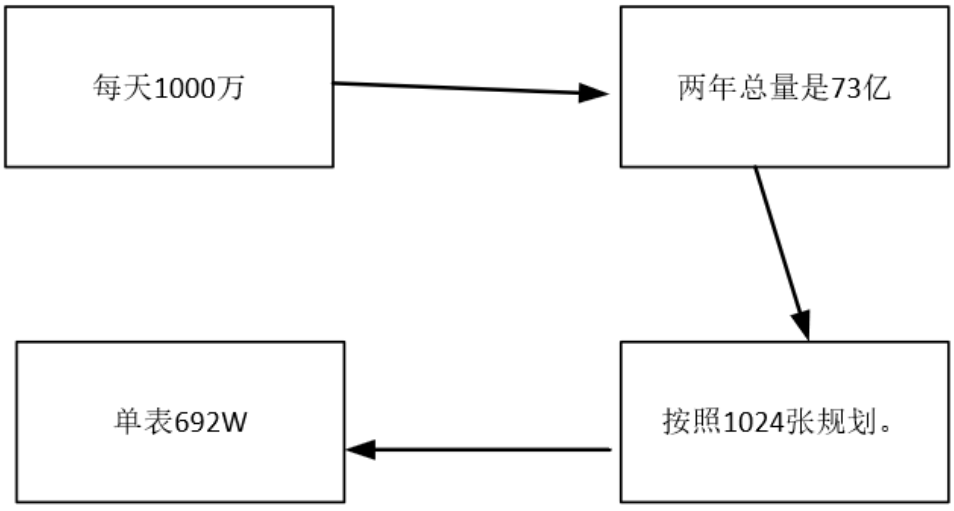
```
id = 1113  id%8 = 1   id/8%16 = 11
id = 1114  id%8 = 2   id/8%16 = 11
id = 1115  id%8 = 3   id/8%16 = 11
id = 1116  id%8 = 4   id/8%16 = 11
id = 1117  id%8 = 5   id/8%16 = 11
id = 1118  id%8 = 6   id/8%16 = 11
id = 1119  id%8 = 7   id/8%16 = 11
id = 1120  id%8 = 0   id/8%16 = 12
id = 1121  id%8 = 1   id/8%16 = 12
id = 1122  id%8 = 2   id/8%16 = 12
id = 1123  id%8 = 3   id/8%16 = 12
id = 1124  id%8 = 4   id/8%16 = 12
id = 1125  id%8 = 5   id/8%16 = 12
id = 1126  id%8 = 6   id/8%16 = 12
id = 1127  id%8 = 7   id/8%16 = 12
```

百亿级库表架构设计

表的数量规划

即便按照**每天1000万，2年内保持稳定的要求**，进行表的数据量规划：

> 两年总量是73亿（每天1000万*730天），假设每张表的标准值为500W，表的数据量平均是1460张，换成2的幂，比较接近1024张，或者说，单表692W（692=73亿/1024），也是可以接受的，所以就按照1024张规划。



库的数量规划

即便按照QPS峰值**5W的要求**，进行表的库的规划：

> 每个库正常承载的写入并发量是 1500，那么32个库就可以承载32 * 1500 = 48000 的写并发

悲观一点，如果每秒写入超过 5WQPS，可以通过**MQ削峰+批写入**的降级策略，MQ的写入吞吐量，可以轻松到达10W级别。

总的规划

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。

Id	id % 32 (库)	id /32 % 32 (表)
1095	7	2
1096	8	2
1097	9	2
1098	10	2

做一个测试

```
    @Test
public void testmod() {
    for (int i = 0; i < 128; i++) {
        int id=1000+i;
        System.out.print("id = " + id);
        System.out.print("  id%32 = " + id%32 );
        System.out.print("   id/32%32 = " + id/32%32 );
        System.out.println(" " );
    }
}
```

结果如下：

```
id = 1088  id%32 = 0   id/32%32 = 2
id = 1089  id%32 = 1   id/32%32 = 2
id = 1090  id%32 = 2   id/32%32 = 2
id = 1091  id%32 = 3   id/32%32 = 2
id = 1092  id%32 = 4   id/32%32 = 2
id = 1093  id%32 = 5   id/32%32 = 2
id = 1094  id%32 = 6   id/32%32 = 2
id = 1095  id%32 = 7   id/32%32 = 2
id = 1096  id%32 = 8   id/32%32 = 2
id = 1097  id%32 = 9   id/32%32 = 2
id = 1098  id%32 = 10   id/32%32 = 2
id = 1099  id%32 = 11   id/32%32 = 2
id = 1100  id%32 = 12   id/32%32 = 2
id = 1101  id%32 = 13   id/32%32 = 2
id = 1102  id%32 = 14   id/32%32 = 2
id = 1103  id%32 = 15   id/32%32 = 2
```

```
id = 1104  id%32 = 16   id/32%32 = 2
id = 1105  id%32 = 17   id/32%32 = 2
id = 1106  id%32 = 18   id/32%32 = 2
id = 1107  id%32 = 19   id/32%32 = 2
id = 1108  id%32 = 20   id/32%32 = 2
id = 1109  id%32 = 21   id/32%32 = 2
id = 1110  id%32 = 22   id/32%32 = 2
id = 1111  id%32 = 23   id/32%32 = 2
id = 1112  id%32 = 24   id/32%32 = 2
id = 1113  id%32 = 25   id/32%32 = 2
id = 1114  id%32 = 26   id/32%32 = 2
id = 1115  id%32 = 27   id/32%32 = 2
id = 1116  id%32 = 28   id/32%32 = 2
id = 1117  id%32 = 29   id/32%32 = 2
id = 1118  id%32 = 30   id/32%32 = 2
id = 1119  id%32 = 31   id/32%32 = 2
id = 1120  id%32 = 0   id/32%32 = 3
id = 1121  id%32 = 1   id/32%32 = 3
id = 1122  id%32 = 2   id/32%32 = 3
id = 1123  id%32 = 3   id/32%32 = 3
id = 1124  id%32 = 4   id/32%32 = 3
id = 1125  id%32 = 5   id/32%32 = 3
id = 1126  id%32 = 6   id/32%32 = 3
id = 1127  id%32 = 7   id/32%32 = 3
```

一开始上来就是 32 个库，每个库 32 个表，那么总共是 1024 张表。

这个规划，基本够用，如果还不够用，就可以按照上面的规则，继续扩充。

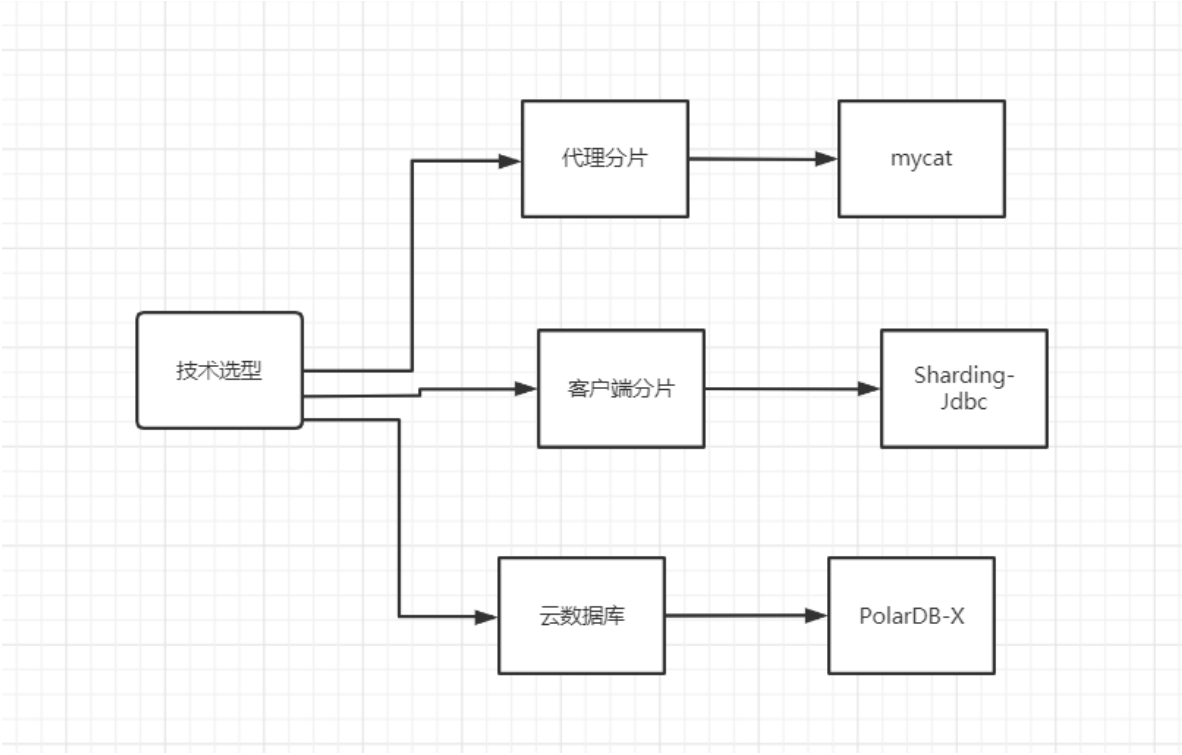
后面如果要拆分，就是不断在库和 mysql 服务器之间做迁移就可以了。然后系统配合改一下配置即可。

这里对步骤做一个总结：

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，id 模 32 = 库，orderId / 32 模 32 = 表

分库分表技术选型

分库分表主要有三种方案：



代理中间件分库分表

利用MyCat，cobar这种代理中间件分库分表。

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。

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。

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。

好处是:

和业务代码耦合度很低，只需做一些配置即可，接入成本低。

缺点是:

1 性能问题，这种代理中间件需要单独部署，所以从调用链路上又多了一层。

2 分库分表逻辑完全由代理中间件管理，对于程序员完全是黑盒，一旦代理本身出问题（比如出错或宕机），会导致无法查询和存储相关业务数据，引发灾难性的后果。

如果不熟悉代理中间件源码，排查问题会非常困难。

> 曾经有公司使用MyCat，线上发生故障后，被迫修改方案，三天三夜才恢复系统。CTO也废了！

客户端分片

利用Sharding-Jdbc，TDDL等以Jar包形式呈现的轻量级组件分库分表。

TDDL

淘宝团队开发的，属于 client 层方案。支持基本的 crud 语法和读写分离，但不支持 join、多表查询等语法。目前使用的也不多，因为还依赖淘宝的 diamond 配置管理系统。

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，是有不少公司在用的），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**。

> 缺点是，会有一定的代码开发工作量，对业务有一些侵入性。好处是对程序员透明，程序员对分库分表逻辑的把控会更强，一旦发生故障，排查问题会比较容易。

云数据库

直接使用云平台的分布式数据库服务

> 例如，阿里云的分布式数据库服务PolarDB-X。

选型建议：

综上，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。

理论上。这两个方案其实都可以选用

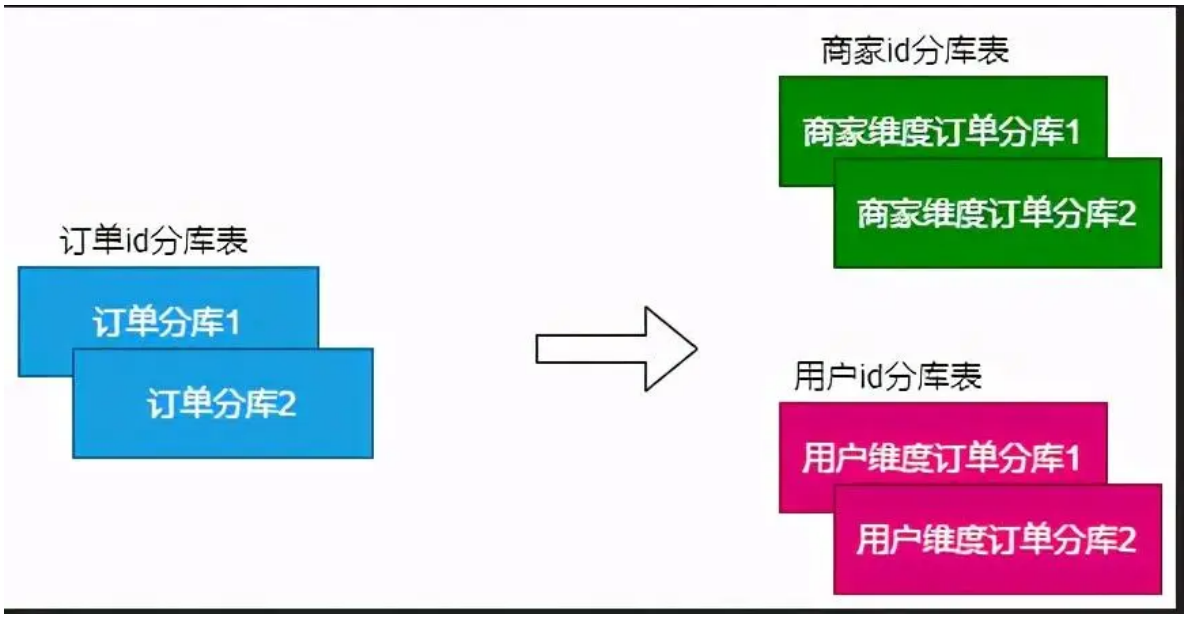
个人建议中小型公司选用 sharding-jdbc，client 层方案轻便，而且维护成本低，不需要额外增派人手，而且中小型公司系统复杂度会低一些，项目也没那么多；

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。

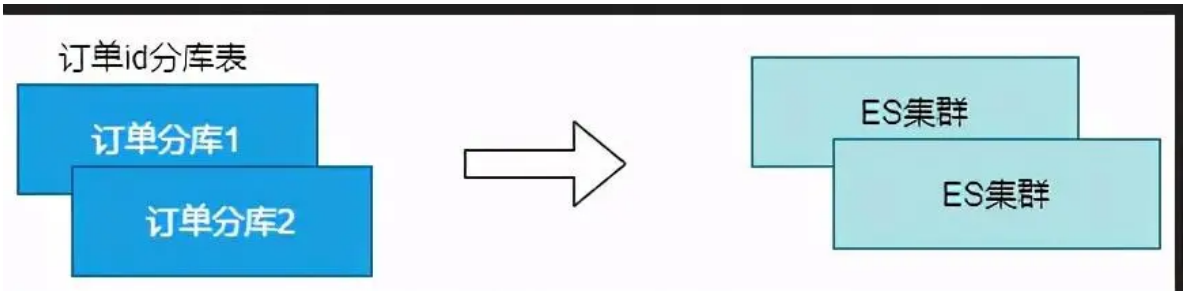
百亿级数据的异构查询

比如对于订单库，当对其分库分表后，如果想按照商家维度或者按照用户维度进行查询，那么是非常困难的，因此可以通过异构数据库来解决这个问题。

按照不同的维度，聚合异构数据

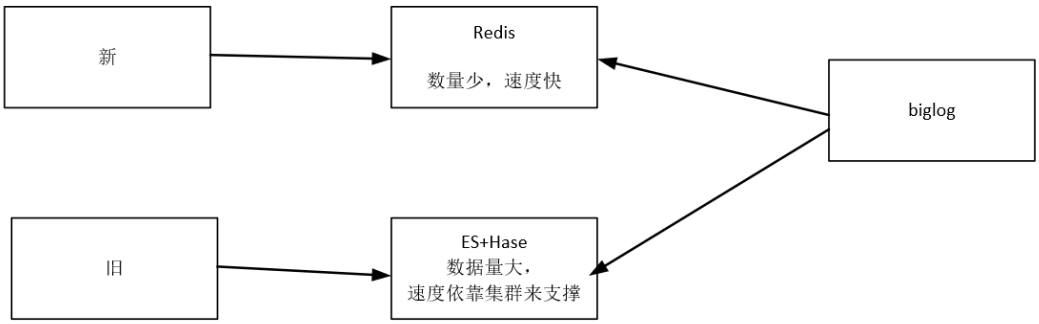


采用ES文档索引异构：



异构数据主要存储数据之间的关系，然后通过查询源库查询实际数据。不过，有时可以通过数据冗余存储来减少源库查询量或者提升查询性能。

或者采用下图的ES+Hbase的异构：



一致性hash分片方案

通过前面的描述，不难看出我们的分库分表方案有一些缺陷，比如采用hash取模的方式会产生数据分布不均匀的情况，扩容缩容也非常麻烦。

这些问题可以用一致性hash方案解决。基于虚拟节点设计原理的一致性hash可以让数据分布更均匀。

而且一致性hash采用环形设计思路，在增减节点时，使得数据迁移的成本会更低，只需要迁移临近节点的数据。

不过需要扩容时基本上要成倍扩容，在hash环上每个节点间隙都增加新的节点，这样才能分摊所有原有节点的访问和存储压力。

关于数据倾斜问题

分库分表，采用Hash取模的方式路由库表，经常会出现数据倾斜问题，比如可能会导致某些表的数据特别多，远超过其他表。

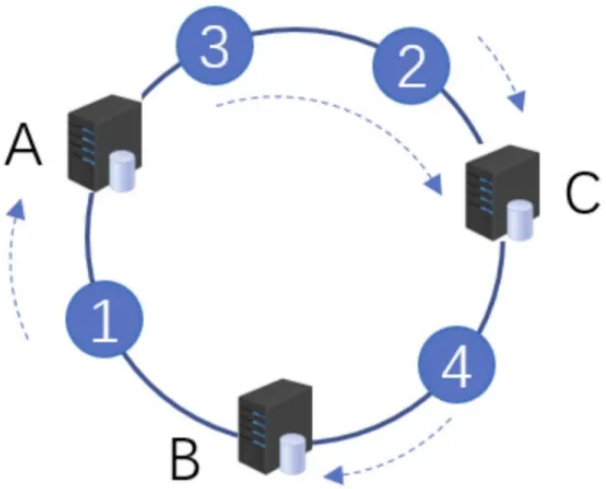
严重时，甚至可能影响系统性能和稳定性。

数据倾斜特别严重时，就要考虑二次分表了，也就是添加一定数量的表，然后按照相应规则将对应的数据迁移到新表中。

上面我们提到了一致性哈希算法，这里我们就可以利用一致性哈希算法对倾斜的数据表进行二次分表。

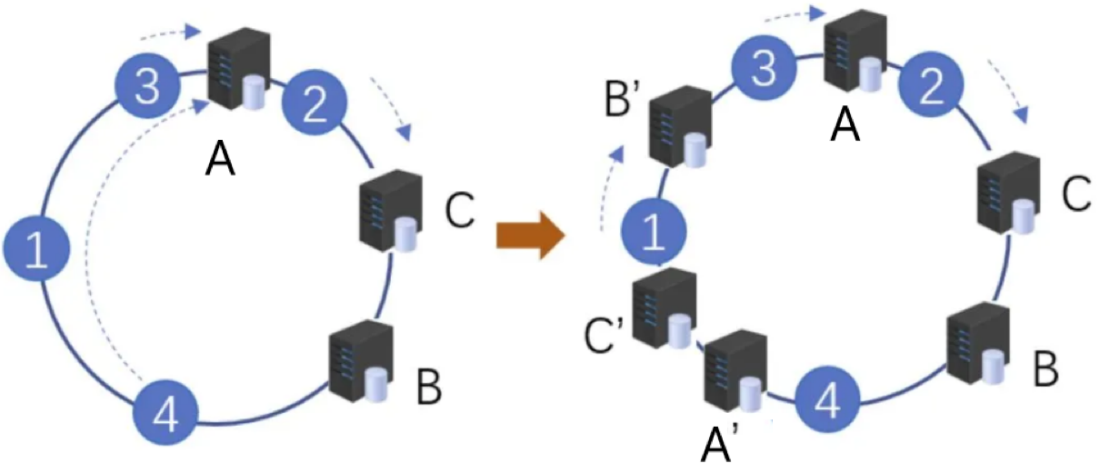
首先，一致性哈希算法可以动态添加虚拟节点，在不增加表的情况下，减少数据倾斜。其次，一致性Hash算法在应对增加一张表这种场景时，只会引发部分数据迁移，不会像普通的Hash算法一样导致需要迁移几乎所有数据。如果没明白上面两句话的意思，不要紧，继续往下看。

一致性hash采用环形设计思路，根据节点（库，表）的Hash值将所有节点映射到Hash环上，在我们查询和写入数据时，会根据该数据的Hash值计算出该数据在Hash环上的位置，然后从该位置开始按顺时针方向找到Hash环上最近一个节点，这个节点就是该数据对应的节点（库，表）。



实际使用中，由于服务器节点数量有限，很有可能出现数据分布不均匀的情况。

这个时候会出现大量数据都被映射到某一个节点的情况，如下图左侧所示。



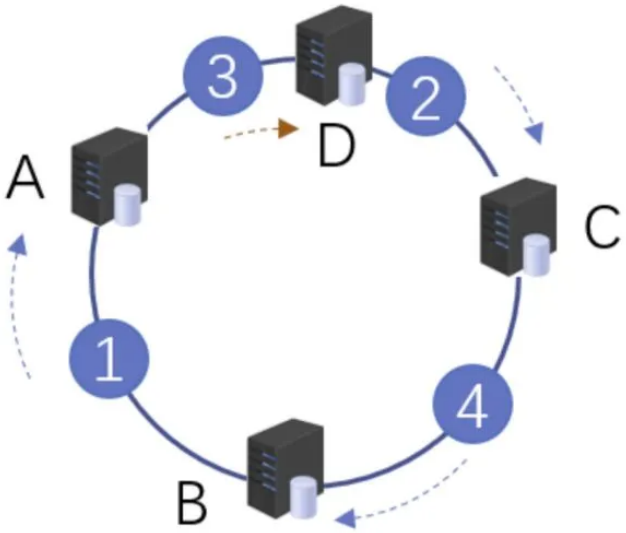
为了解决这个问题，一致性Hash算法巧妙利用了虚拟节点。

虚拟节点是实际节点在Hash环上的复制品，一个实际节点可以对应很多的虚拟节点。虚拟节点越多，Hash环上的节点也就越多，数据分布也会越均匀。 如上图右侧所示，B’、C’、A’ 是原始节点复制出来的虚拟节点，

原本映射到节点A上的数据1和4，分别被映射到了B’和A’，最终会落到对应的真实节点B和A上。

> 我们可以看到，虚拟节点让数据分布更均匀了。并且虚拟节点越多， 数据分布越均匀。

如果需要增加新的节点（库，表）。例如新增了节点D，位置落在数据2和3之间。按照顺时针原则，数据2依然在节点C上，而数据3就会路由到节点D。



所以利用一致性Hash算法，在增加节点后我们只需要将很少的数据迁移到新节点上，在这里只需要将C节点的部分数据迁移到D上。

数据迁移方案

停机迁移方案

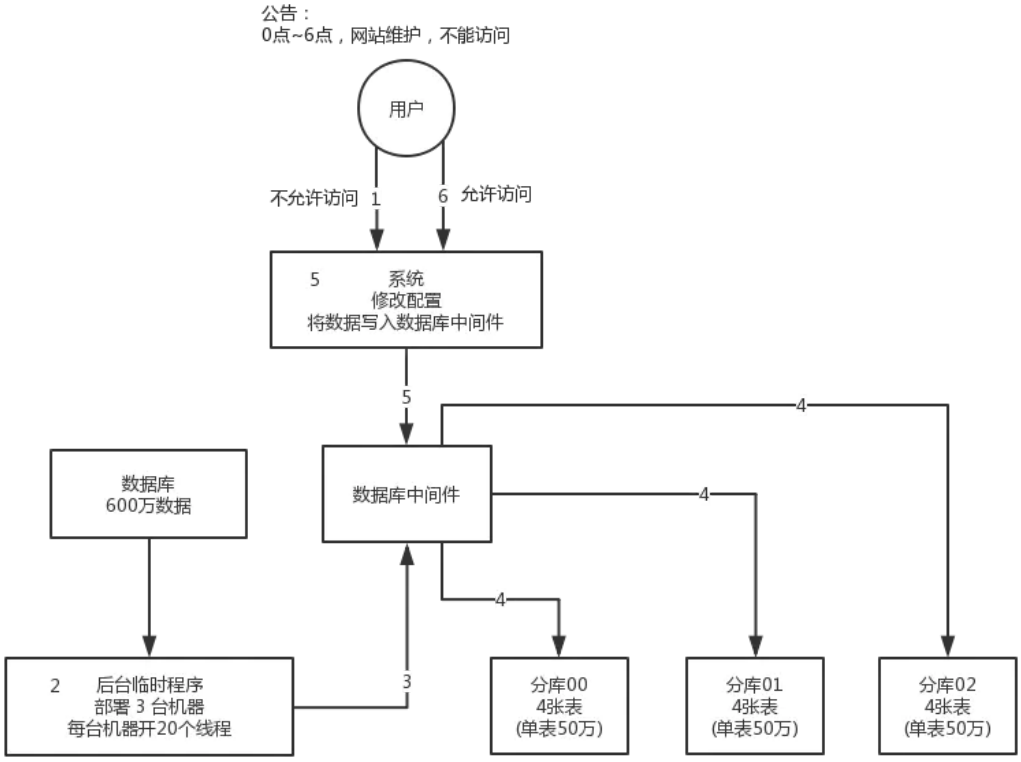
先来一个最 low 的方案：

准备工作：迁移之前，装好新库，建好新表，已经使用**导数的工具**，然后将老表的数据出来，写到分库分表里面去。并且做完检查，通过脚本程序检验数据，看新库数据是否准确以及有没有漏掉的数据

> 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，系统停掉，没有流量写入了，此时老的单库单表数据库静止了。

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，迁移完成。



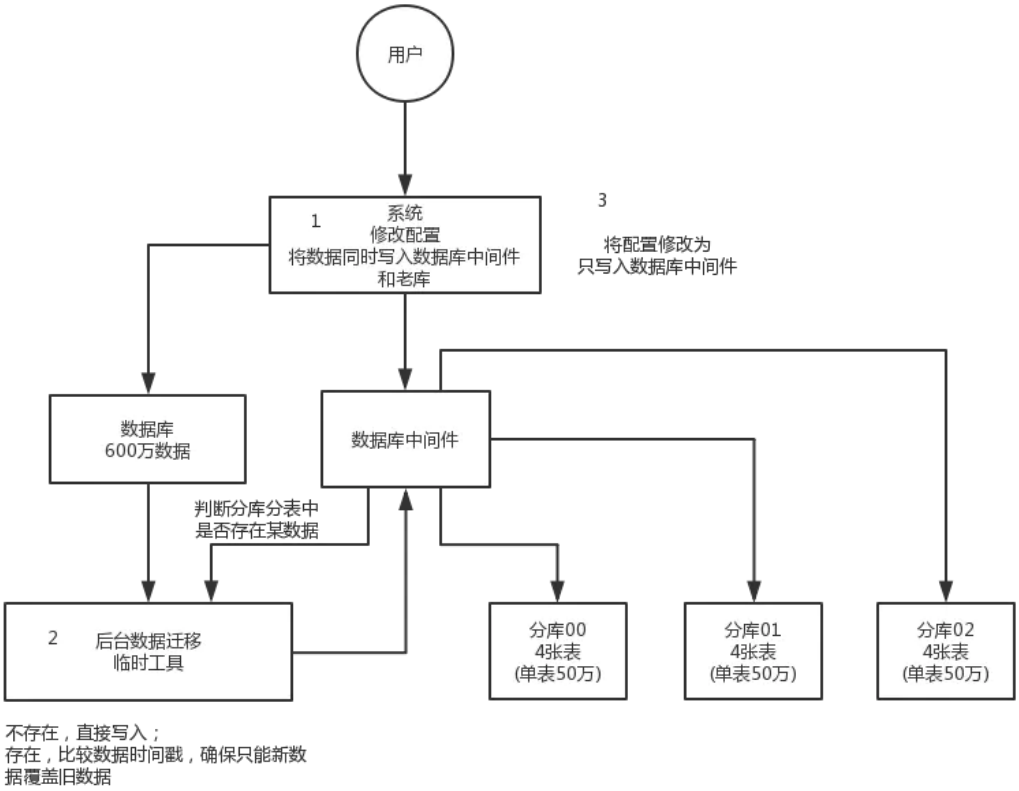
双写迁移方案

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**，这就是所谓的**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。简单来说，就是不允许用老数据覆盖新数据。

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，都是这么干的。



数据同步方案

我们可以看到上面双写的方案比较麻烦，很多数据库写入的地方都需要修改代码。有没有更好的方案呢?

> 我们还可以利用Canal，DataBus等工具做数据同步。以阿里开源的Canal为例。

利用同步工具，就不需要开启双写了，服务层也不需要编写双写的代码，直接用Canal做增量数据同步即可。相应的步骤就变成了：

1. 代码准备。

   准备Canal代码，解析binary log字节流对象，并把解析好的订单数据写入新库。

   准备迁移程序脚本，用于做老数据迁移。

   准备校验程序脚本，用于校验新库和老库的数据是否一致。

2. 运行Canal代码，开始增量数据（线上产生的新数据）从老库到新库的同步。

3. 利用脚本程序，将某一时间戳之前的老数据迁移到新库。

   注意：

   1，时间戳一定要选择开始运行Canal程序后的时间点（比如运行Canal代码后10分钟的时间点），避免部分老数据被漏掉；

   3，迁移过程一定要记录日志，尤其是错误日志，如果有些记录写入失败，我们可以通过日志恢复数据，以此来保证新老库的数据一致。

4. 第3步完成后，我们还需要通过脚本程序检验数据，看新库数据是否准确以及有没有漏掉的数据

5. 数据校验没问题后，开启双读，起初给新库放少部分流量，新库和老库同时读取。由于延时问题，新库和老库可能会有少量数据记录不一致的情况，所以新库读不到时需要再读一遍老库。逐步将读流量切到新库，相当于灰度上线的过程。遇到问题可以及时把流量切回老库

6. 读流量全部切到新库后，将写入流量切到新库（可以在代码里加上热配置开关。注：由于切换过程Canal程序还在运行，仍然能够获取老库的数据变化并同步到新库，所以切换过程不会导致部分老库数据无法同步新库的情况）

7. 关闭Canal程序

8. 迁移完成。

分库分表之后，id 主键如何处理?

考点分析

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。

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、占用空间大，**作为主键性能太差**了；更重要的是，UUID 不具有有序性，会导致 B+ 树索引在写的时候有过多的随机写操作（连续的 ID 可以产生部分顺序写），还有，由于在写的时候不能产生有顺序的 append 操作，而需要进行 insert 操作，将会读取整个 B+ 树节点到内存，在插入这条记录后会将整个节点写回磁盘，这种操作在记录占用空间比较大的情况下，性能下降明显。

适合的场景：如果你是要随机生成个什么文件名、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
```

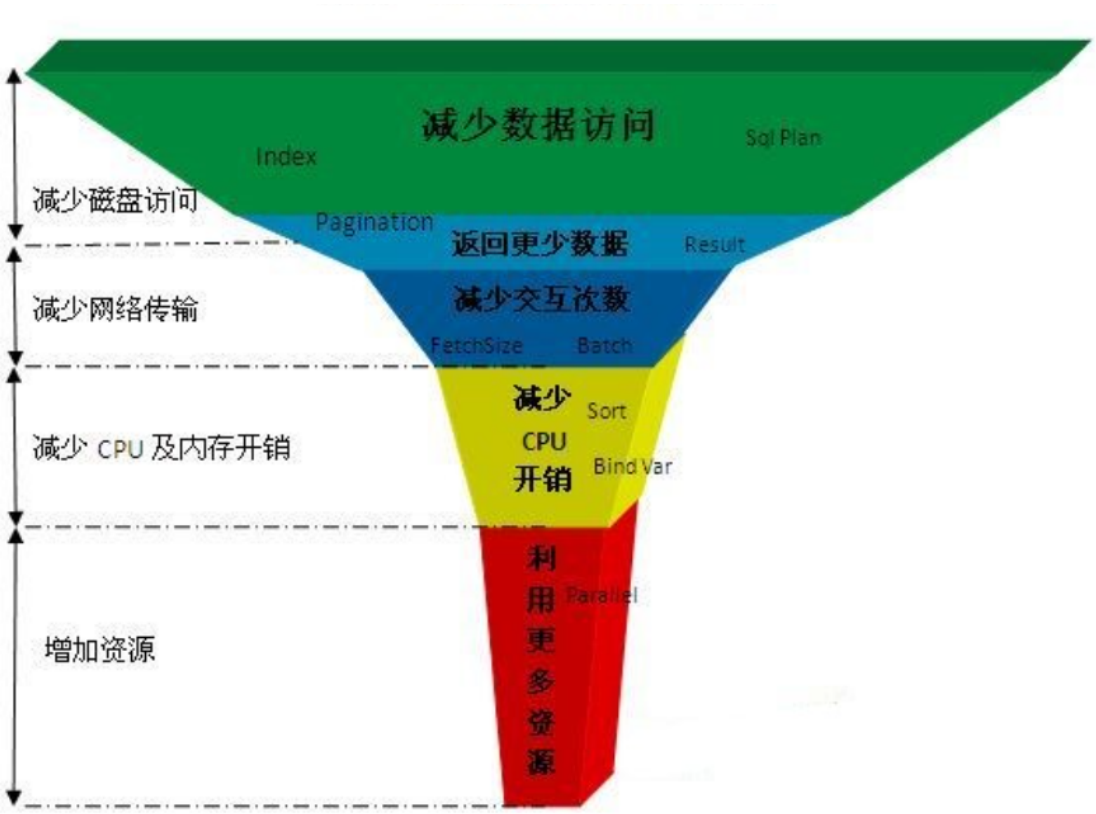
snowflake 算法

snowflake 算法是 twitter 开源的分布式 id 生成算法，采用 Scala 语言实现，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 12 bit 代表的数字来区分**同一个毫秒内**的 4096 个不同的 id。

方法之一减少请求数量

避免重复的请求消耗不必要的资源；通过客户端自身的处理能力来响应请求，而不必到达服务端，这两方面可以有效避免由于请求量过于庞大而造成的超出服务器承受范围的问题。



参考文献

https://zhuanlan.zhihu.com/p/197964246

https://www.cnblogs.com/wuer888/p/14524303.html#autoid-0-0-0

https://www.cnblogs.com/xuzhengzong/articles/11346937.html