### 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？）

#### 分表

比如你单表都几千万数据了，你确定你能扛住么？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

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。



### 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**

**Sharding-jdbc**

当当开源的，属于 client 层方案，目前已经更名为 [ShardingSphere](https://github.com/apache/incubator-shardingsphere)（后文所提到的 Sharding-jdbc，等同于 ShardingSphere）。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，而且截至 2019.4，已经推出到了 4.0.0-RC1 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。

**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。

总结

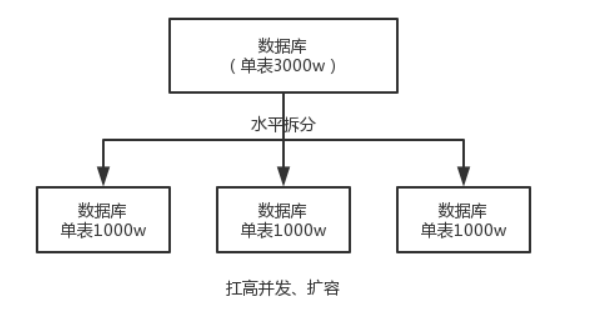
综上，现在其实建议考量的，就是 Sharding-jdbc 和 Mycat，这两个都可以去考虑使用。

1，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；

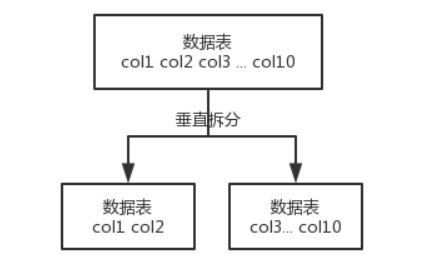
2，Mycat 这种 proxy 层方案的缺点在于需要部署，自己运维一套中间件，运维成本高，但是**好处在于对于各个项目是透明的**，如果遇到升级之类的都是自己中间件那里搞就行了。

### 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，然后用多个库来扛更高的并发，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，无论分库还是分表，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，扩容的时候很简单，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。

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。

#### 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

#### 2.1、水平分库

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。  
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；

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。  
分析：库多了，io和cpu的压力自然可以成倍缓解。

#### 2.2、水平分表

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。  
结果：

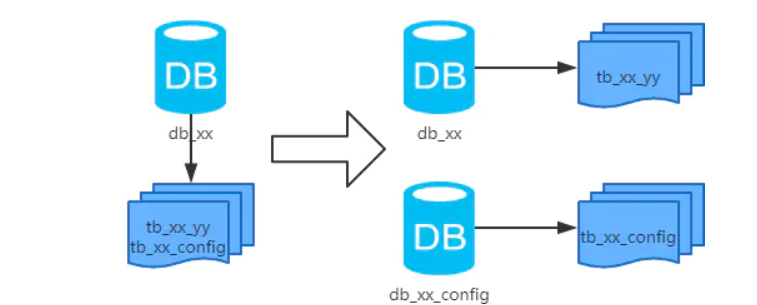
每个表的结构都一样；

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；

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。  
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。

#### 2.3、垂直分库



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。  
结果：

库的结构都不一样；

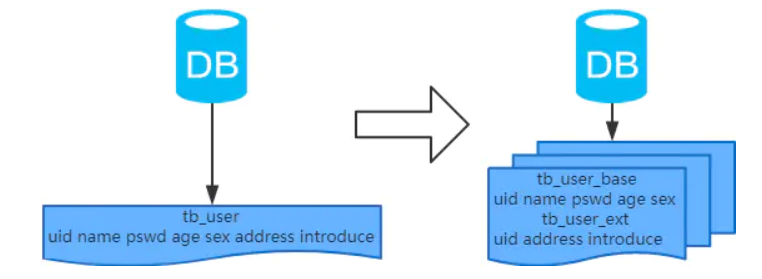
每个库的数据也不一样，没有交集；

所有库的并集是全量数据；

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。

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。

#### 2.4、垂直分表



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。  
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；

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。

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。

**垂直切分的优点**：

* 解决业务系统层面的耦合，业务清晰
* 与微服务的治理类似，也能对不同业务的数据进行分级管理、维护、监控、扩展等
* 高并发场景下，垂直切分一定程度的提升IO、数据库连接数、单机硬件资源的瓶颈

缺点：

* 部分表无法join，只能通过接口聚合方式解决，提升了开发的复杂度
* 分布式事务处理复杂
* 依然存在单表数据量过大的问题（需要水平切分）

### 2. 分库分表策略

https://my.oschina.net/boonya/blog/3098522

#### 1. 取余

#### 2. 按照范围分片

 按照范围分片，顾名思义，就是首先对整体数据进行范围划分，然后将各个范围区间分配到对应的数据库节点上，当用户插入数据时，根据指定字段的值，判断其属于哪个范围，然后将数据插入到该范围对应的数据库节点上。需要注意的是，这里会配置一个默认的范围，当用户插入的数据不再任何指定的范围内时，该数据将会被插入到默认节点上。如下是按范围分片的配置：

#### 3. 按照日期进行分片

        按照日期分片，这种方式相对来说理解稍微复杂一点，我们这里直接展示一个配置示例：

#### 4. 按照月份进行分片

        按照月份进行分片，顾名思义，就是以月为单位，判断目标时间在哪个月内，然后就将数据分配到这个月对应的数据节点上。如下是按照月份进行分片的配置示例：

#### 5. 按照枚举值分片

        按照枚举值分片比较适合于某个字段只有固定的几个值的情况，比如省份。通过配置文件将每个枚举值对应的数据库节点进行映射，这样对于指定类型的数据，就会被分配到同一个数据库实例中。如下是按照枚举值分片的一个示例：

#### 6. 范围取模

        范围取模分片的优点在于，既拥有范围分片的固定范围数据不做迁移的优点，也拥有了取模分片对于热点数据均匀分布的优点。首先我们还是以一个示例进行讲解：

#### 7. 二进制取模范围分片

        二进制取模分片的方式与范围取模非常相似，但也有不同，其分片方式主要是根据目标分片字段的低10位的值来判断其属于哪个分片。我们首先还是以一个示例进行讲解：

#### 8. 一致性hash分片

        一致性hash分片方式上面的二进制取模方式非常相似，不过一致性hash的虚拟槽的概念更强，并且一致性hash分片的虚拟槽的数量是可配置的。如下是一个典型的一致性hash分片的配置方式：

#### 9. 按照目标字段前缀指定的进行分区

        按照目标字段前缀进行分片，这种方式就比较好理解，其会获取到指定分区字段的前缀值，然后将其转换为十进制数字，将其作为分区值，如果该数字超过了分区数量，则会将当前数据放在默认分区。如下是按照字符串前缀方式分区的配置示例：

#### 10. 按照前缀ASCII码和值进行取模范围分片

        按照前缀ASCII码和值进行取模，顾名思义，就是取了前缀之后，将其转换为ASCII码值，然后对取模基数进行取模，最后将求得的余数按照配置文件中的范围分配到对应的数据库节点上。如下是该分区方式的配置示例：

## 面试题

**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。

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。

### 停机迁移方案双写迁移方案

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，**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。

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，系统停掉，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。

# 分库分表之后，id 主键如何处理

# 分库分表之后分布式如何保证ID全局唯一性

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:**通过唯一识别码16个字节128位的长数字。**  
组成部分：当前日期和时间序列+全局的唯一性网卡mac地址  
优点：代码实现简单、不占用宽带、数据迁移不受影响  
缺点：无序、无法保证趋势递增（要求3）字符存储、传输、查询慢、不可读

### 基于数据库的实现方案

#### 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**无论如何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。

### 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

**Snowflake雪花算法**

 国外的twitter分布式下ID生成算法

1bit+41bit+10bit+10+bit=62bit

高位随机+毫秒数+机器码（数据中心+机器id）+10的流水好

国内：

保证数据的唯一性就行了IDC机房

优点：代码实现简单、不占用宽带、数据迁移不受影响、低位趋势递增

缺点：强以来时钟（多台服务器时间一定要一样）、无序无法保证趋势递增（要求3）

**你们有没有做 MySQL 读写分离？如何实现 MySQL 的读写分离？MySQL 主从复制原理的是啥？如何解决 MySQL 主从同步的延时问题？**

## 面试官心理分析

高并发这个阶段，肯定是需要做读写分离的，啥意思？因为实际上大部分的互联网公司，一些网站，或者是 app，其实都是读多写少。所以针对这个情况，就是写一个主库，但是主库挂多个从库，然后从多个从库来读，那不就可以支撑更高的读并发压力了吗？

## 面试题剖析

### 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 主从复制原理的是啥？

**主库将变更写入 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 relay 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。**

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**这个所谓**半同步复制**，也叫 semi-sync 复制，指的就是主库写入 binlog 日志之后，就会将**强制**此时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。

所谓**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。

**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。