

18 | 可伸缩架构案例：数据太多，如何无限扩展你的数据库？

2020-04-01 王庆友

架构实战案例解析

[进入课程 >](#)




讲述：王庆友

时长 20:01 大小 18.34M



你好，我是王庆友。在 [第 16 讲](#) 中，我和你介绍了很多可伸缩的架构策略和原则。那么今天，我会通过 1 号店订单水平分库的实际案例，和你具体介绍如何实现系统的可伸缩。

问题和解决思路

2013 年，随着 1 号店业务的发展，每日的订单量接近 100 万。这个时候，订单库已有上亿条记录，订单表有上百个字段，这些数据存储在一个 Oracle 数据库里。当时，我们已经实现了订单的服务化改造，只有订单服务才能访问这个订单数据库，但随着单量的增长， 在线促销的常态化，单一数据库的存储容量和访问性能都已经不能满足业务需求了，订单数据库已成为系统的瓶颈。所以，对这个数据库的拆分势在必行。

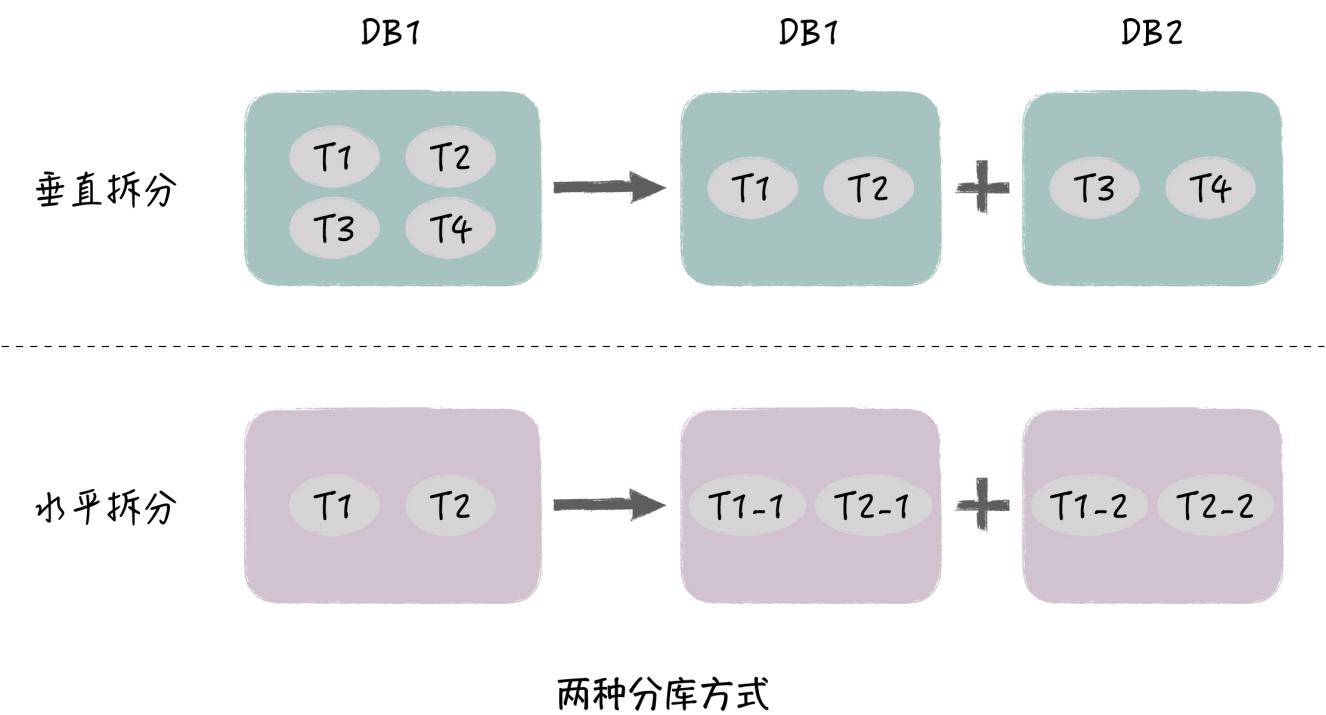
数据库拆分一般有两种做法，一个是垂直分库，还有一个是水平分库。

垂直分库

简单来说，垂直分库就是数据库里的表太多，我们把它们分散到多个数据库，一般是根据业务进行划分，把关系密切的表放在同一个数据库里，这个改造相对比较简单。

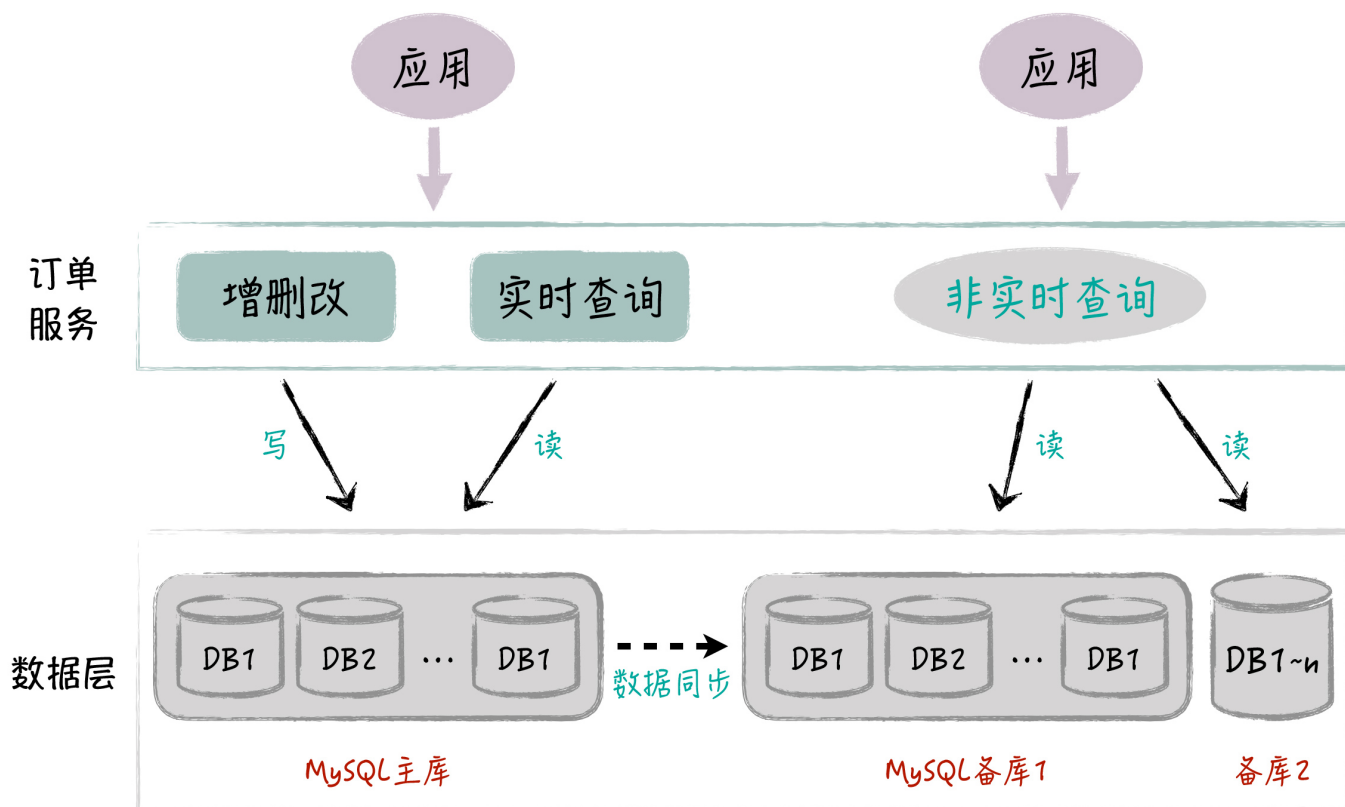
水平分库

某些表太大，单个数据库存储不下，或者数据库的读写性能有压力。通过水平分库，我们把一张表拆成多张表，每张表存放部分记录，分别保存在不同的数据库里，水平分库需要对应用做比较大的改造。



当时，1 号店已经通过服务化，实现了订单库的**垂直拆分**，它的订单库主要包括订单基本信息表、订单商品明细表、订单扩展表。这里的问题不是表的数量太多，而是单表的数据量太大，读写性能差。所以，1 号店通过**水平分库**，把这 3 张表的记录分到多个数据库当中，从而分散了数据库的存储和性能压力。

水平分库后，应用通过订单服务来访问多个订单数据库，具体的方式如下图所示：



原来的一个 Oracle 库被现在的多个 MySQL 库给取代了，每个 MySQL 数据库包括了 1 主 1 备 2 从，都支持读写分离，主备之间通过自带的同步机制来实现数据同步。所以，你可以发现，**这个项目实际包含了水平分库和去 Oracle 两大改造目标。**

分库策略

我们先来讨论一下水平分库的具体策略，包括要选择哪个分库维度，数据记录如何划分，以及要分为几个数据库。

分库维度怎么定？

首先，我们需要考虑根据哪个字段来作为分库的维度。

这个字段选择的标准是，尽量避免应用代码和 SQL 性能受到影响。具体地说，就是现有的 SQL 在分库后，它的访问尽量落在单个数据库里，否则原来的单库访问就变成了多库扫描，不但 SQL 的性能会受到影响，而且相应的代码也需要进行改造。

具体到订单数据库的拆分，你可能首先会想到按照**用户 ID**来进行拆分。这个结论是没错，但我们最好还是要有量化的数据支持，不能拍脑袋。

这里，最好的做法是，先收集所有 SQL，挑选出 WHERE 语句中最常出现的过滤字段，比如说这里有三个候选对象，分别是用户 ID、订单 ID 和商家 ID，每个字段在 SQL 中都会出现三种情况：

- 1. 单 ID 过滤，比如说 “用户 ID=? ” ；
- 2. 多 ID 过滤，比如 “用户 ID IN(?,?,?)” ；
- 3. 该 ID 不出现。

最后，我们分别统计这三个字段的使用情况，假设共有 500 个 SQL 访问订单库，3 个候选字段出现的情况如下：

过滤字段	单ID过滤	多ID过滤	不出现
用户ID	120	40	330
订单ID	60	80	360
商家ID	15	0	485

从这张表来看，结论非常明显，我们应该选择用户 ID 来进行分库。

不过，等一等，这**只是静态分析**。我们知道，每个 SQL 访问的频率是不一样的，所以，我们还要分析每个 SQL 的实际访问量。

在项目中，我们分析了 Top15 执行次数最多的 SQL（它们占总执行次数 85%，具有足够代表性），按照执行的次数，如果使用用户 ID 进行分库，这些 SQL 85% 会落到单个数据库，13% 落到多个数据库，只有 2% 需要遍历所有的数据库。所以说，**从 SQL 动态执行次数的角度来看**，用户 ID 分库也明显优于使用其他两个 ID 进行分库。

这样，通过前面的量化分析，我们知道按照用户 ID 分库是最优的选择，同时也大致知道了分库对现有系统会造成多大影响。比如在这个例子中，85% 的 SQL 会落到单个数据库，那么这部分的数据访问相对于不分库来说，执行性能会得到一定的优化，这样也解决了我们之前对分库是否有效果的疑问，坚定了分库的信心。

数据怎么分？

好，分库维度确定了以后，我们如何把记录分到各个库里呢？

一般有两种数据分法：

1. **根据 ID 范围进行分库**，比如把用户 ID 为 1 ~ 999 的记录分到第一个库，1000 ~ 1999 的分到第二个库，以此类推。
2. **根据 ID 取模进行分库**，比如把用户 ID mod 10，余数为 0 的记录放到第一个库，余数为 1 的放到第二个库，以此类推。

这两种分法，各自存在优缺点，如下表所示：

评级指标	按照范围分库	按照取模分库
库数量	前期数目比较小，可以随用户/业务按需增长	前期即根据mod因子确定库数量，数目一般比较大
访问性能	前期库数量小，全库查询消耗资源少，单库查询性能略差	前期库数量大，全库查询消耗资源多，单库查询性能略好
调整库数量	比较容易，一般只需为新用户增加库，老库拆分也只影响单个库	困难，改变mod因子导致数据在所有库之间迁移
数据热点	新旧用户购物频率有差异，有数据热点问题	新旧用户均匀分布到各个库，无热点

在实践中，为了运维方便，选择 ID 取模进行分库的做法比较多。同时为了数据迁移方便，一般分库的数量是按照倍数增加的，比如说，一开始是 4 个库，二次分裂为 8 个，再分成 16 个。这样对于某个库的数据，在分裂的时候，一半数据会移到新库，剩余的可以不用动。与此相反，如果我们每次只增加一个库，所有记录都要按照新的模数做调整。

在这个项目中，我们结合订单数据的实际情况，最后采用的是**取模**的方式来拆分记录。

补充说明：按照取模进行分库，每个库记录数一般比较均匀，但也有些数据库，存在超级 ID，这些 ID 的记录远远超过其他 ID。比如在广告场景下，某个大广告主的广告数可能占很大比例。如果按照广告主 ID 取模进行分库，某些库的记录数会特别多，对于这些超级 ID，需要提供单独库来存储记录。

分几个库？

现在，我们确定了记录要怎么分，但具体要分成几个数据库呢？

分库数量，首先和**单库能处理的记录数**有关。一般来说，MySQL 单库超过了 5000 万条记录，Oracle 单库超过了 1 亿条记录，DB 的压力就很大（当然这也和字段数量、字段长度和查询模式有关系）。

在满足前面记录数量限制的前提下，如果分库的数量太少，我们达不到分散存储和减轻 DB 性能压力的目的；如果分库的数量太多，好处是单库访问性能好，但对于跨多个库的访问，应用程序需要同时访问多个库，如果我们并发地访问所有数据库，就意味着要消耗更多的线程资源；如果是串行的访问模式，执行的时间会大大地增加。

另外，分库数量还直接影响了**硬件的投入**，多一个库，就意味着要多投入硬件设备。所以，具体分多少个库，需要做一个综合评估，一般初次分库，我建议你分成 4~8 个库。在项目中，我们拆分为了 6 个数据库，这样可以满足较长一段时间的订单业务需求。

分库带来的问题

不过水平分库解决了单个数据库容量和性能瓶颈的同时，也给我们带来了一系列新的问题，包括数据库路由、分页以及字段映射的问题。

分库路由

分库从某种意义上来说，意味着 DB Schema 改变了，必然会影响应用，但这种改变和业务无关，所以我们要尽量保证分库相关的逻辑都在数据访问层进行处理，对上层的订单服务透明，服务代码无需改造。

当然，要完全做到这一点会很困难。那么具体哪些改动应该由 DAL（数据访问层）负责，哪些由订单服务负责，这里我给你一些可行的建议：

对于**单库访问**，比如查询条件已经指定了用户 ID，那么该 SQL 只需访问特定库即可。此时应该由 DAL 层自动路由到特定库，当库二次分裂时，我们也只需要修改取模因子就可以了，应用代码不会受到影响。

对于**简单的多库查询**，DAL 层负责汇总各个分库返回的记录，此时它仍对上层应用透明。

对于**带聚合运算的多库查询**，比如说带 groupby、orderby、min、max、avg 等关键字，建议可以让 DAL 层汇总单个库返回的结果，然后由上层应用做进一步的处理。这样做有两方面的原因，一方面是因为让 DAL 层支持所有可能的聚合场景，实现逻辑会很复

杂；另一方面，从 1 号店的实践来看，这样的聚合场景并不多，在上层应用做针对性处理，会更加灵活。

DAL 层还可以进一步细分为**底层 JDBC 驱动层**和**偏上面的数据访问层**。如果我们基于 JDBC 层面实现分库路由，系统开发难度大，灵活性低，目前也没有很好的成功案例。

在实践中，我们一般是基于持久层框架，把它进一步封装成 **DDAL** (Distributed Data Access Layer, 分布式数据访问层)，实现分库路由。1 号店的 DDAL 就是基于 iBatis 进一步封装而来的。

分页处理

水平分库后，分页查询的问题比较突出，因为有些分页查询需要遍历所有库。

举个例子，假设我们要按时间顺序展示某个商家的订单，每页有 100 条记录，由于是按商家查询，我们需要遍历所有数据库。假设库数量是 8，我们来看下水平分库后的分页逻辑：

如果是取第 1 页数据，我们需要从每个库里按时间顺序取前 100 条记录，8 个库汇总后共有 800 条，然后我们对这 800 条记录在应用里进行二次排序，最后取前 100 条；

如果取第 10 页数据，则需要从每个库里取前 1000 (100×10) 条记录，汇总后共有 8000 条记录，然后我们对这 8000 条记录进行二次排序后，取第 900 到 1000 之间的记录。

你可以看到，在分库情况下，对于每个数据库，我们要取更多的记录，并且汇总后，还要在应用里做二次排序，越是靠后的分页，系统要耗费更多的内存和执行时间。而在不分库的情况下，无论取哪一页，我们只要从单个 DB 里取 100 条记录即可，也无需在应用内部做二次排序，非常简单。

那么，我们如何解决分库情况下的分页问题呢？这需要具体情况具体分析：

如果是为前台应用提供分页，我们可以限定用户只能看到前面 n 页（这个限制在业务上也是合理的，一般看后面的分页意义不大，如果一定要看，可以要求用户缩小范围重新查询）；

如果是后台批处理任务要求分批获取数据，我们可以加大分页的大小，比如设定每次获取 5000 条记录，这样可以有效减少分页的访问次数；

分库设计时，一般还有配套的大数据平台负责汇总所有分库的记录，所以有些分页查询，我们可以考虑走大数据平台。

分库字段映射

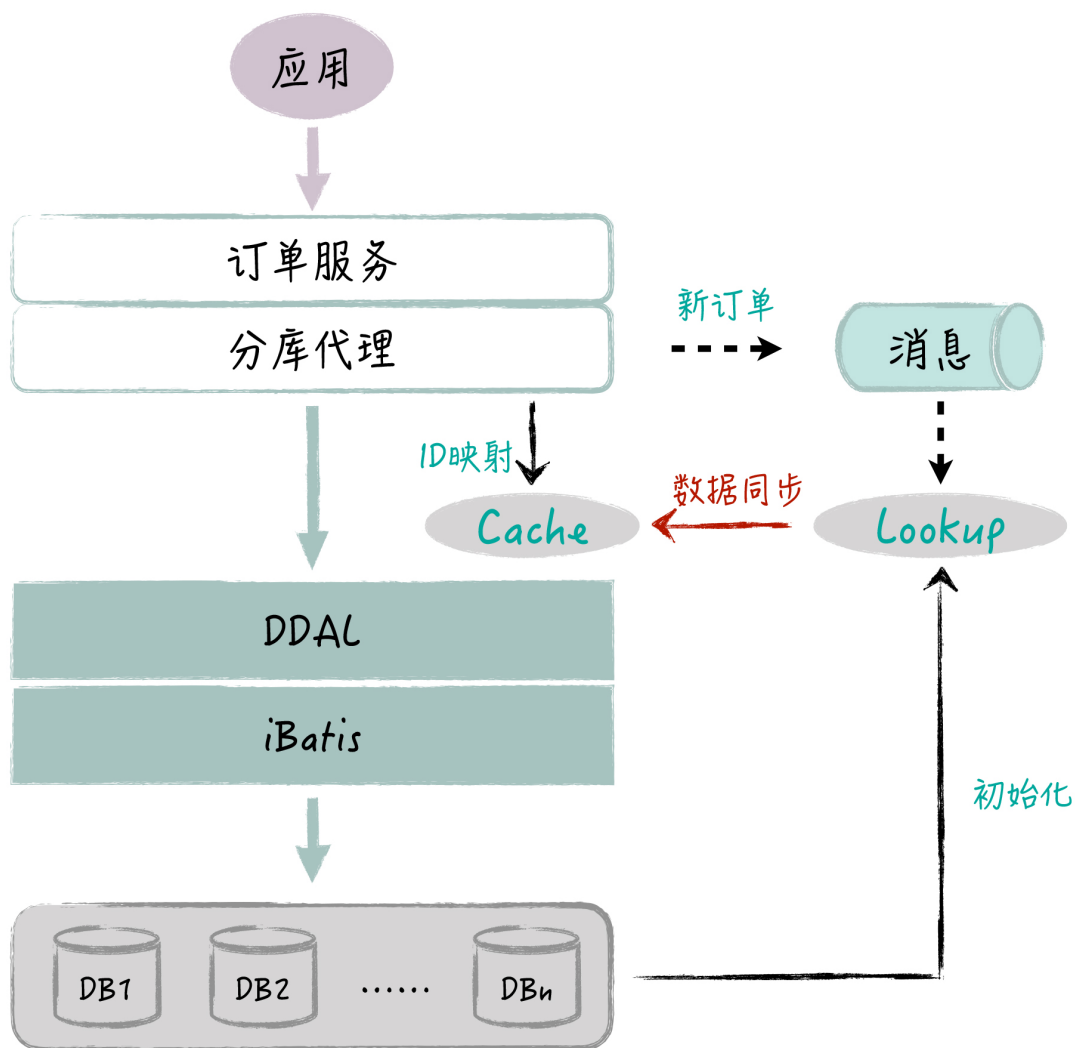
分库字段只有一个，比如这里，我们用的是用户 ID，如果给定用户 ID，这个查询会落到具体的某个库。但我们知道，在订单服务里，根据**订单 ID** 查询的场景也很多见，不过由于订单 ID 不是分库字段，如果不对它做特殊处理，系统会盲目查询所有分库，从而带来不必要的资源开销。

所以，这里我们为**订单 ID 和用户 ID 创建映射，保存在 Lookup 表里**，我们就可以根据订单 ID，找到相应的用户 ID，从而实现单库定位。

Lookup 表的记录数和订单库记录总数相等，但它只有 2 个字段，所以存储和查询性能都不是问题，这个表在单独的数据库里存放。在实际使用时，我们可以通过**分布式缓存**，来优化 Lookup 表的查询性能。此外，对于新增的订单，除了写订单表，我们同时还要写 Lookup 表。

整体架构

通过以上分析，最终的 1 号店订单水平分库的总体技术架构如下图所示：



上层应用通过订单服务访问数据库；

分库代理实现了分库相关的功能，包括聚合运算、订单 ID 到用户 ID 的映射，做到分库逻辑对订单服务透明；

Lookup 表用于订单 ID 和用户 ID 的映射，保证订单服务按订单 ID 访问时，可以直接落到单个库，Cache 是 Lookup 表数据的缓存；

DDAL 提供库的路由，可以根据用户 ID 定位到某个库，对于多库访问，DDAL 支持可选的多线程并发访问模式，并支持简单的记录汇总；

Lookup 表初始化数据来自于现有的分库数据，当新增订单记录时，由分库代理异步写入。

如何安全落地？

订单表是系统的核心业务表，它的水平拆分会影响到很多业务，订单服务本身的代码改造也很大，很容易导致依赖订单服务的应用出现问题。我们在上线时，必须谨慎考虑。

所以，为了保证订单水平分库的总体改造可以安全落地，整个方案的实施过程如下：

首先，实现 Oracle 和 MySQL 两套库并行，所有数据读写指向 Oracle 库，我们通过数据同步程序，把数据从 Oracle 拆分到多个 MySQL 库，比如说 3 分钟增量同步一次。

其次，我们选择几个对数据实时性要求不高的访问场景（比如访问历史订单），把订单服务转向访问 MySQL 数据库，以检验整套方案的可行性。

最后，经过大量测试，如果性能和功能都没有问题，我们再一次性把所有实时读写访问转向 MySQL，废弃 Oracle。

这里，我们把上线分成了两个阶段：**第一阶段**，把部分非实时的功能切换到 MySQL，这个阶段主要是为了**验证技术**，它包括了分库代理、DDAL、Lookup 表等基础设施的改造；**第二阶段**，主要是**验证业务功能**，我们把所有订单场景全面接入 MySQL。1 号店两个阶段的上线都是一次性成功的，特别是第二阶段的上线，100 多个依赖订单服务的应用，通过简单的重启就完成了系统的升级，中间没有出现一例较大的问题。

项目总结

1 号店在完成订单水平分库的同时，也实现了去 Oracle，设备从小型机换成了 X86 服务器，我们通过水平分库和去 Oracle，不但支持了订单量的未来增长，并且总体成本也大幅下降。

不过由于去 Oracle 和订单分库一起实施，带来了双重的性能影响，我们花了很大精力做**性能测试**。为了模拟真实的线上场景，我们通过 **TCPCopy**，把线上实际的查询流量引到测试环境，先后经过 13 轮的性能测试，最终 6 个 MySQL 库相对一个 Oracle，在当时的数据量下，SQL 执行时间基本持平。这样，我们**在性能不降低的情况下，通过水平分库优化了架构，实现了订单处理能力的水平扩展。**

1 号店最终是根据用户 ID 后三位取模进行分库，初始分成了 6 个库，理论上可以支持多达 768 个库。同时我们还改造了订单 ID 的生成规则，使其包括用户 ID 后三位，这样新订单 ID 本身就包含了库定位所需信息，无需走 Lookup 映射机制。随着老订单归档到历史库，在前面给出的架构中，Lookup 表相关的部分就可以逐渐废弃了。

如果要扩充数据库的数量，从 6 个升到 12 个，我们可以分三步走：

1. 增加 6 个 MySQL 实例，把现有 6 个库的数据同步到新的库，比如说，0 号库同步到 6 号库，1 号库同步到 7 号库等等；
2. 在配置文件里把分库的取模从 6 变成 12；
3. 通过数据库脚本，每个库删掉一半数据，比如对于 0 号库，删掉用户 $ID \% 12 = 6$ 的记录，对于 6 号库，删掉用户 $ID \% 12 = 0$ 的记录。

你可以看到，通过这样的分库方式，整个数据库扩展是非常容易的，不涉及复杂的数据跨库迁移工作。

订单的水平分库是一项系统性工作，需要大胆设计，谨慎实施。**你需要把握住这几个要点：**

首先，你需要在分库策略的指导下，结合实际情况，在每个方面做出最合适的选择；

其次，对于特殊场景，如分页查询，你需要具体问题具体解决；

最后，你要总体规划，控制好落地步骤，包括对系统改造、性能测试、数据迁移、上线实施等各个环节做好衔接，保证业务不出问题。

总结

今天我和你分享了 1 号店订单水平分库的实际案例，并给出了具体的做法和原因，相信你已经掌握了如何通过对数据库的水平拆分，来保证系统的高性能和可伸缩。

水平分库是针对有状态的存储节点进行水平扩展，相对于无状态的节点，系统改造的复杂性比较高，要考虑的点也比较多。通过今天的分享，希望你以后在设计一个复杂方案时，能够更全面地思考相关的细节，提升架构设计能力。

最后，给你留一道思考题：你公司的数据库有什么瓶颈吗，你计划对它做什么样的改造呢？

欢迎在留言区和我互动，我会第一时间给你反馈。如果觉得有收获，也欢迎你把这篇文章分享给你的朋友。感谢阅读，我们下期再见。

点击参与 

20年架构老兵邀你一起 打卡，带你进阶资深架构师



扫一扫参与小程序话题



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 高性能架构案例：如何设计一个秒杀系统？


精选留言 (5)

 写留言



Jxin

2020-04-01

- 1.分页为何不走es？ 这样聚合多库数据觉得比较复杂，而复杂本身就是需要警惕的事。
- 2.取模那里，0库转移 $\%12=7$ 的数据。没有6号库，只有5号库，转移 $\%12=11$ 的数据。
- 3.用户id可以截取写入订单号中，以此减少一次中间表映射的成本。担心安全性，则做加...
展开 

作者回复: 1. 分页怎么处理具体要看场景，实时性要求不高的可以走大数据或ES，要求高的还是要并行直接查数据库。

2. 关于数据迁移逻辑，原来6个库（编号为0-5），分别放 $\text{Id} \% 6$ 后0-5的数据，现在新增加编号是6-11号库，原来0号库一半数据($\text{id} \% 12 = 6$)要迁到6号库,文中说的6号库，指的是迁移后新加的6号库。

3. 文章最后提到，最后落的方案是按照用户id后三位取模，对订单编号进行改造，加入了用户ID后三位。



👍 1



AlfredLover

2020-04-01

精彩，分页这里，代码层面是循环去查询每个库的记录，还是每个库一个线程并行的方式去查询？

作者回复: 文章后面有提到，dal层提供开关，让上层调用时选择是否并行执行



👍 1



台风骆骆

2020-04-01

老师的课特别接地气，实用，赞

展开 ▼



tt

2020-04-01

都是具体的，有步骤的，可以落地的经验，收藏！

展开 ▼



正在减肥的胖籽。

2020-04-01

老师中午好，请教你几个问题

1.水平拆分库，代码层面也就是修改路由。你们线上怎么实现平滑迁移？如果我现在4个表，需要拆分成8个表。上线的时候用户还需要正常访问。我现在没想到好的方案。

展开 ▼

作者回复: 凌晨2-3点短暂停系统，提前全网会发公告



