大型电商分布式系统实践 第5周

DATAGURU专业数据分析社区

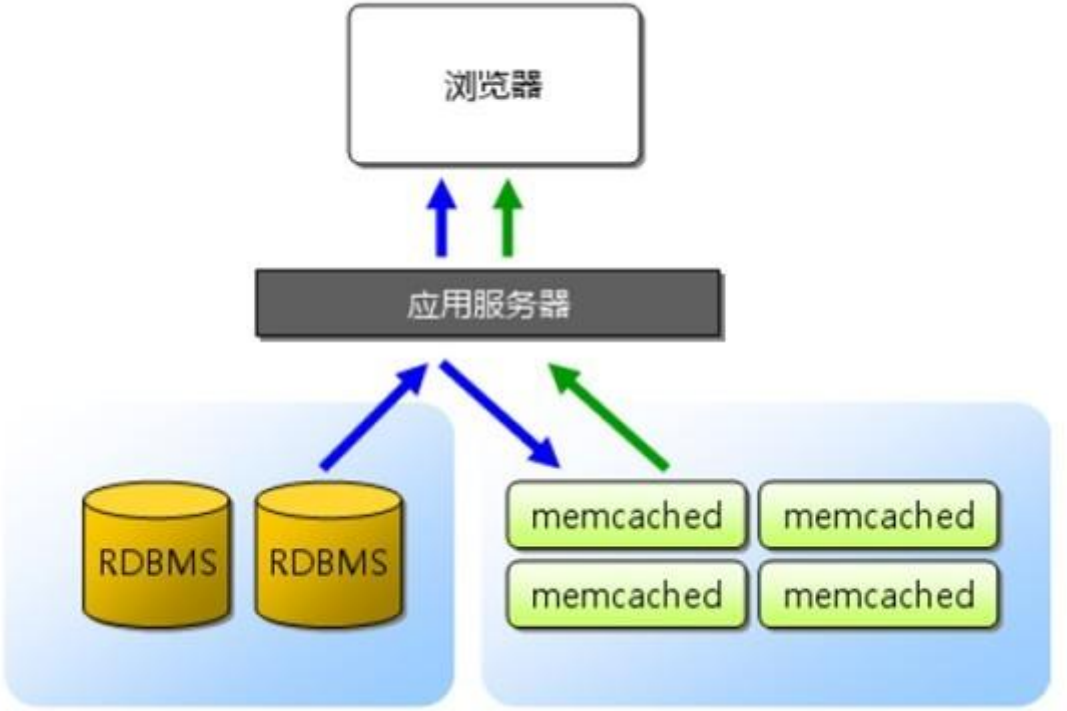
上节内容回顾

分布式系统的基础设施

一个大型的稳健成熟的分布式系统的背后，往往会涉及众多的支撑系统，我们将这些支撑系统称为分布式系统的基础设施。除了前面所介绍的分布式协作及配置管理系统zookeeper之外，我们进行系统架构设计所依赖的基础设施，还包括分布式缓存系统、持久化存储、分布式消息系统、搜索引擎，以及CDN系统、负载均衡系统、运维自动化系统等等，还有后面章节所要介绍的实时计算系统、离线计算系统、分布式文件系统、日志收集系统、监控系统、数据仓库等等。

分布式缓存

高并发环境下，大量的读写请求涌向数据库，磁盘的处理速度与内存显然不在一个量级，从减轻数据库的压力和提高系统响应速度两个角度来考虑，一般都会在数据库之前加一层缓存。由于单台机器的内存资源以及承载能力有限，并且，如果大量使用本地缓存，也会使相同的数据被不同的节点存储多份，对内存资源造成较大的浪费，因此，才催生出了分布式缓存。



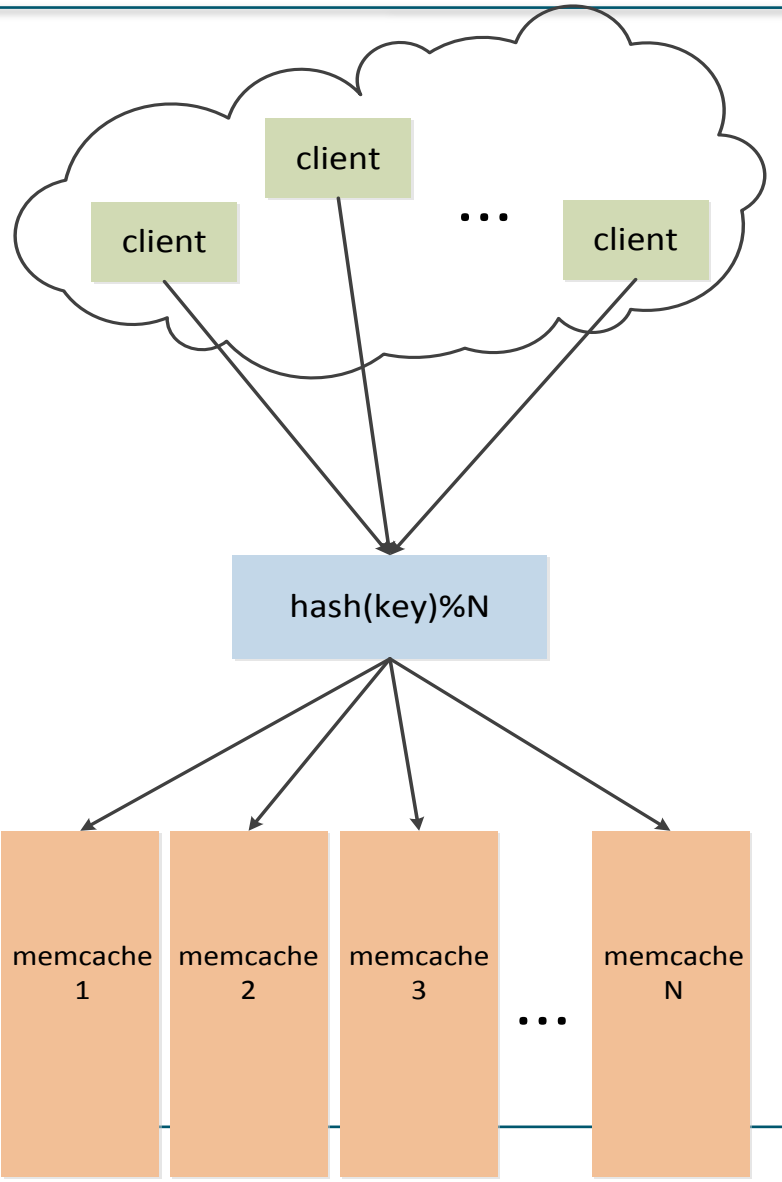
memcache

memcache是一款开源的高性能的分布式内存对象缓存系统，用于在应用中减少对数据库的访问，提高应用的访问速度，并降低数据库的负载。

为了在内存中提供数据的高速查找能力，memcache使用 key-value的形式存储和访问数据，在内存中维护一张巨大的HashTable，使得对数据查询的时间复杂度降低到O(1)，保证了对数据的高性能访问。内存的空间总是有限的，当内存没有更多的空间来存储新的数据时，memcache就会使用LRU(Least Recently Used)算法，将最近不常访问的数据淘汰掉，以腾出空间来存放新的数据。memcache存储支持的数据格式也是灵活多样的，通过对象的序列化机制，可以将更高层抽象的对象转换成为二进制数据，存储在缓存服务器中，当前端应用需要时，又可以通过二进制内容反序列化，将数据还原成原有对象。

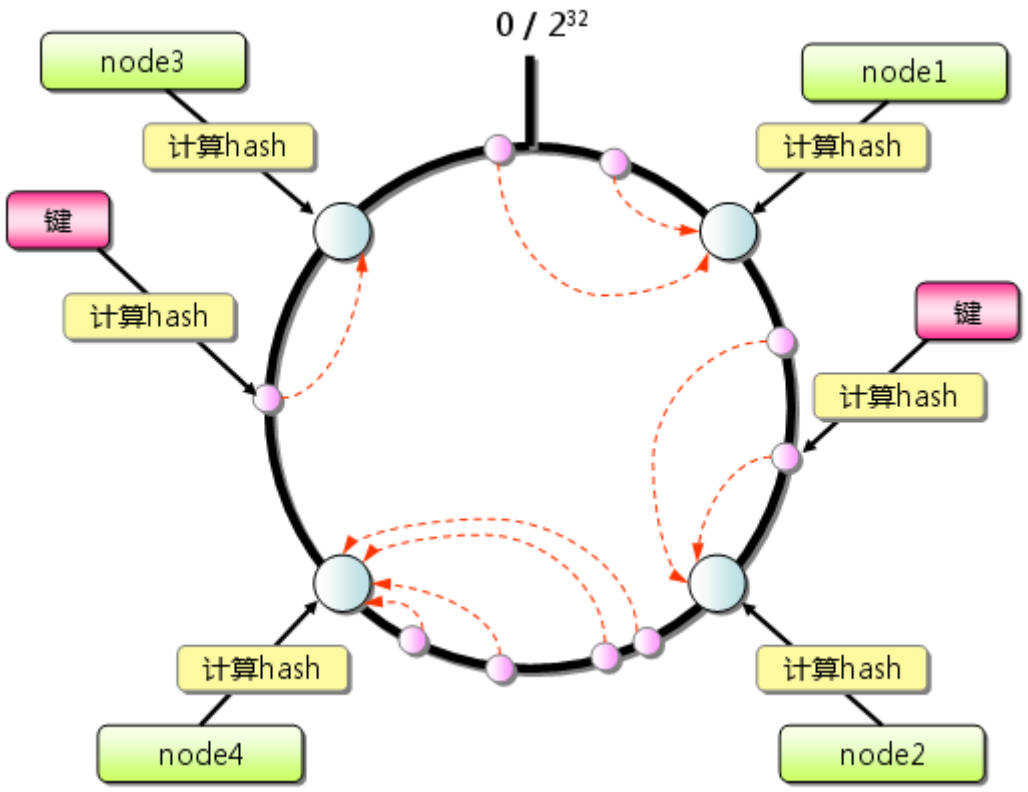
缓存的分布式架构

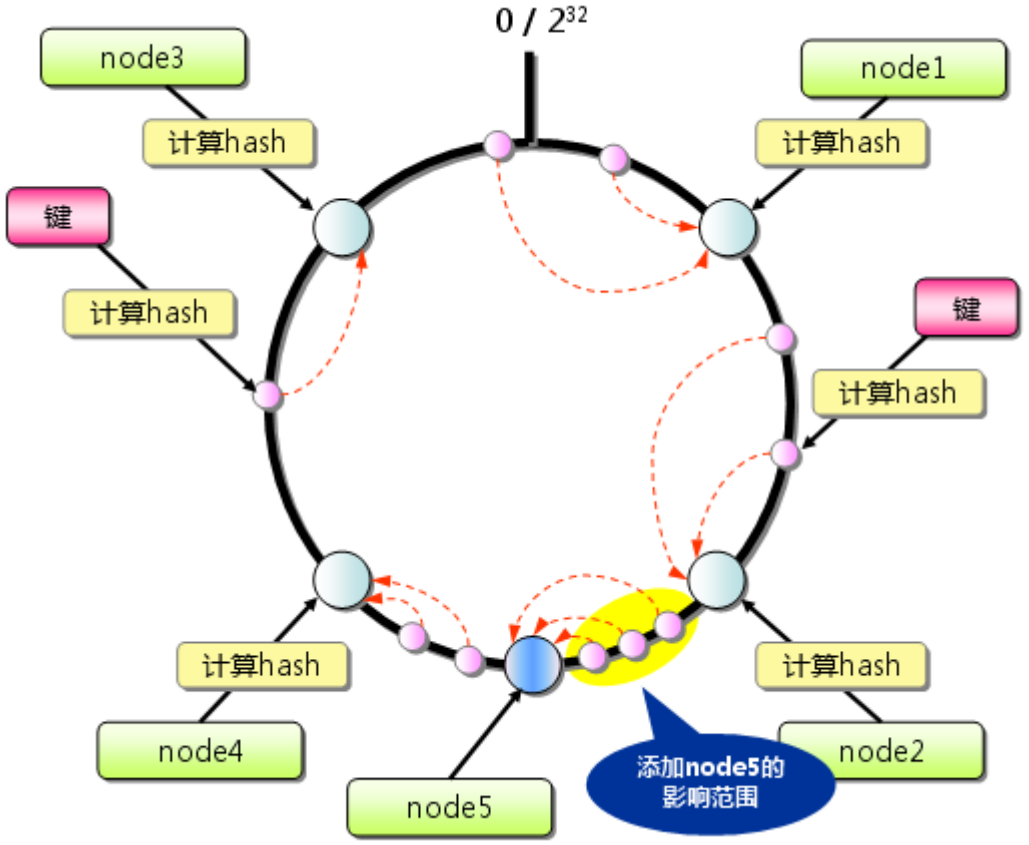
memcache本身并不是一种分布式的缓存系统，它的分布式，是由访问它的客户端来实现的。一种比较简单的实现方式是根据缓存的key来进行hash，当后端有N台缓存服务器时，访问的服务器为hash(key)%N，这样可以将前端的请求均衡的映射到后端的缓存服务器，如图所示，但是，这样也会导致一个问题，一旦后端某台缓存服务器宕机，或者是由于集群压力过大，需要新增新的缓存服务器，大部分的key将会重新分布，对于高并发系统来说，这可能会演变成一场灾难，所有的请求将如洪水般疯狂的涌向后端的数据库服务器，而数据库服务器的不可用，将会导致整个应用的不可用，形成所谓的“雪崩效应”



一致性hash算法

consistent hash算法能够在一定程度上改善缓存的雪崩问题，它能够在移除/添加一台缓存服务器时，尽可能小的改变已存在的key映射关系，避免大量key的重新映射。



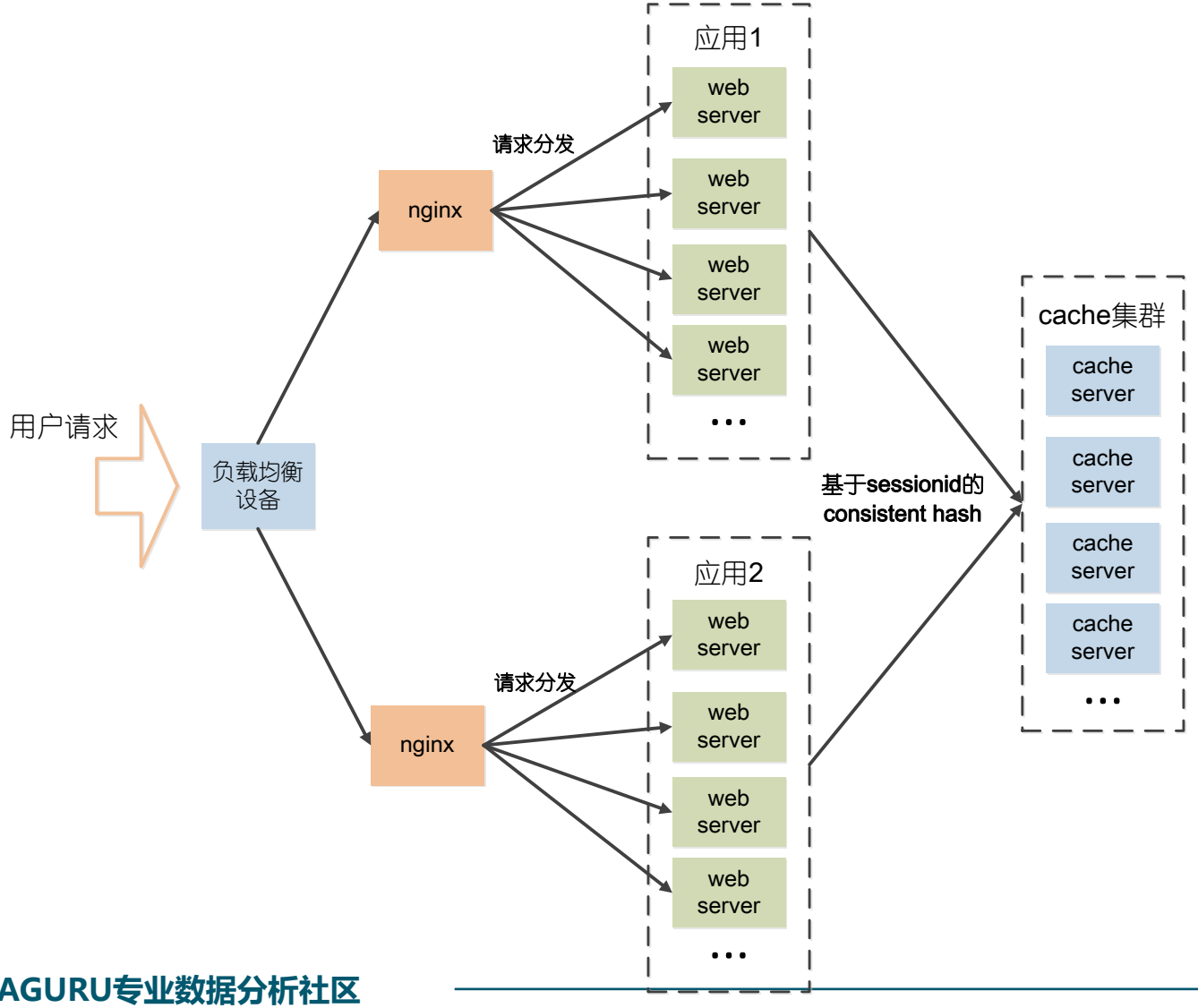


分布式session

传统的应用服务器，如tomcat、jboss等等，其自身所实现的session管理大部分都是基于单机的，对于大型分布式网站来说，支撑其业务的远远不止是一台服务器，而是一个分布式集群，请求在不同服务器之间跳转，需要保持服务器之间的session同步。传统网站一般通过将一部分数据存储在cookie中，来规避分布式环境下session的操作，这样做弊端很多，一方面cookie的安全性一直广为诟病，并且，cookie存储数据的大小是有限制的，随着移动互联网的发展，很多情况下还得兼顾移动端的session需求，使得采用cookie来进行session同步的方式弊端更为凸显。分布式session正是在这种情况下应运而生的。

一种分布式session解决方案

前端用户请求经过随机分发之后，可能会命中后端任意的web server，并且，web server也可能会因为各种不确定的原因宕机，这种情况下，session是很难在集群间同步的，而通过将session以sessionid作为key，保存到后端的缓存集群中，使得不管请求如何分配，即便是web server宕机，也不会影响其他的web server通过sessionid从cache server中获得session，这样，即实现了集群间的session同步，又提高了web server的容错性。



缓存的容灾

业务强依赖缓存，缓存需做到容灾：

1. 双机房互相备份

2. 数据复制多份，单台缓存失效，集群间能够自动复制和备份

3. 数据库留有余量

4. 万兆网卡

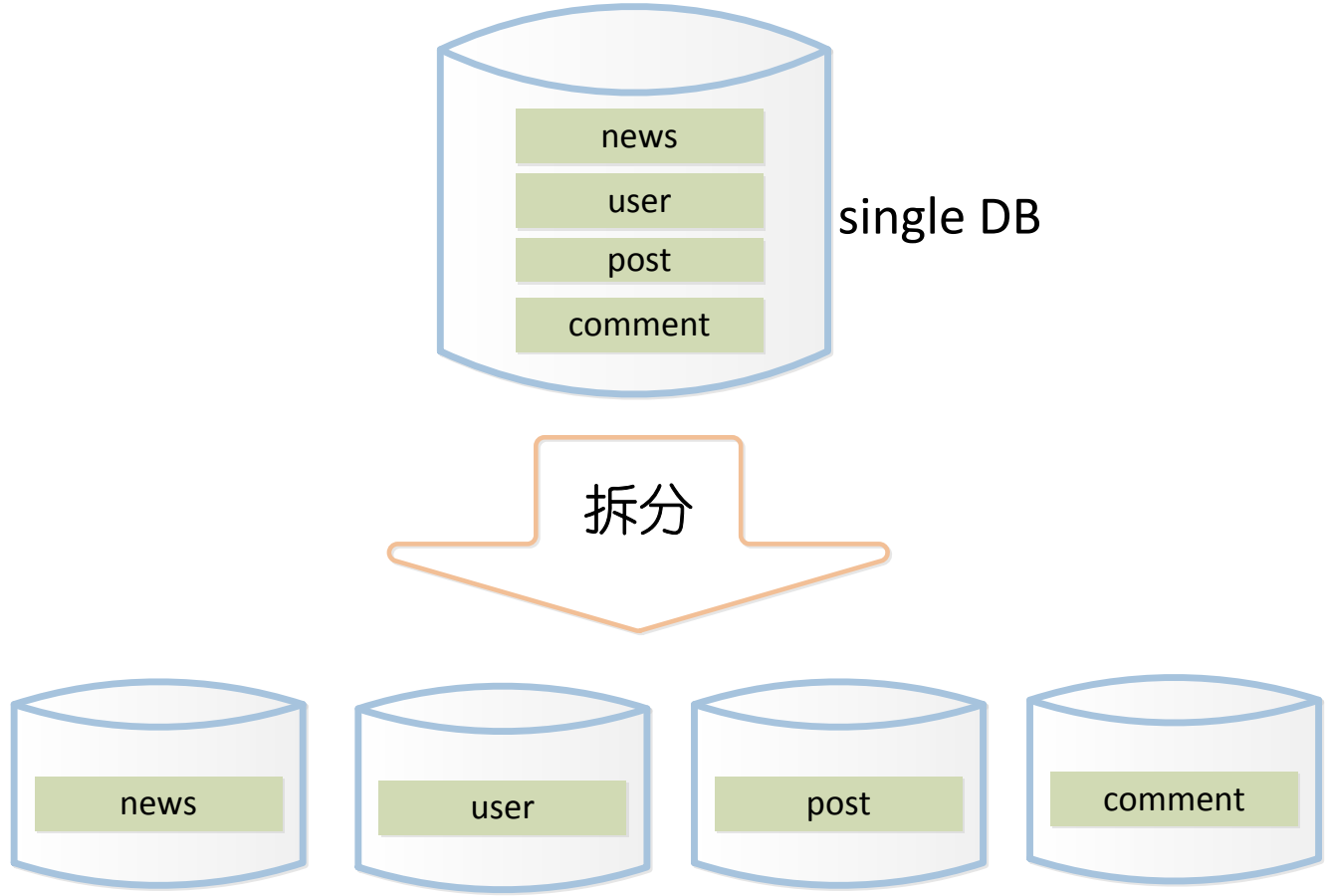
持久化存储

传统的IOE解决方案，使用和扩展的成本越来越高，使得互联网企业不得不思考新的解决方案，开源软件加廉价PC server的分布式架构，得益于社区的支持，在节约成本的同时，也给系统带来了良好的扩展能力，并且，由于开源软件的代码透明，使得企业能够以更低的代价定制更符合自身使用场景的功能，以提高系统的整体性能。互联网企业常用的三种数据存储方案，传统关系型数据库mysql，用来存储结构化数据，google率先提出的bigtable概念及其开源实现HBase，则用来存储海量的非结构化数据，还有诸如包含丰富数据类型的key-value存储redis，文档型存储mongodb等等..

关系型数据库mysql—业务拆分

业务发展初期为了便于快速迭代，很多应用都采用集中式的架构，随着业务规模的扩展，系统变得越来越复杂，访问量越来越大，不得不进一步扩展系统的吞吐能力。

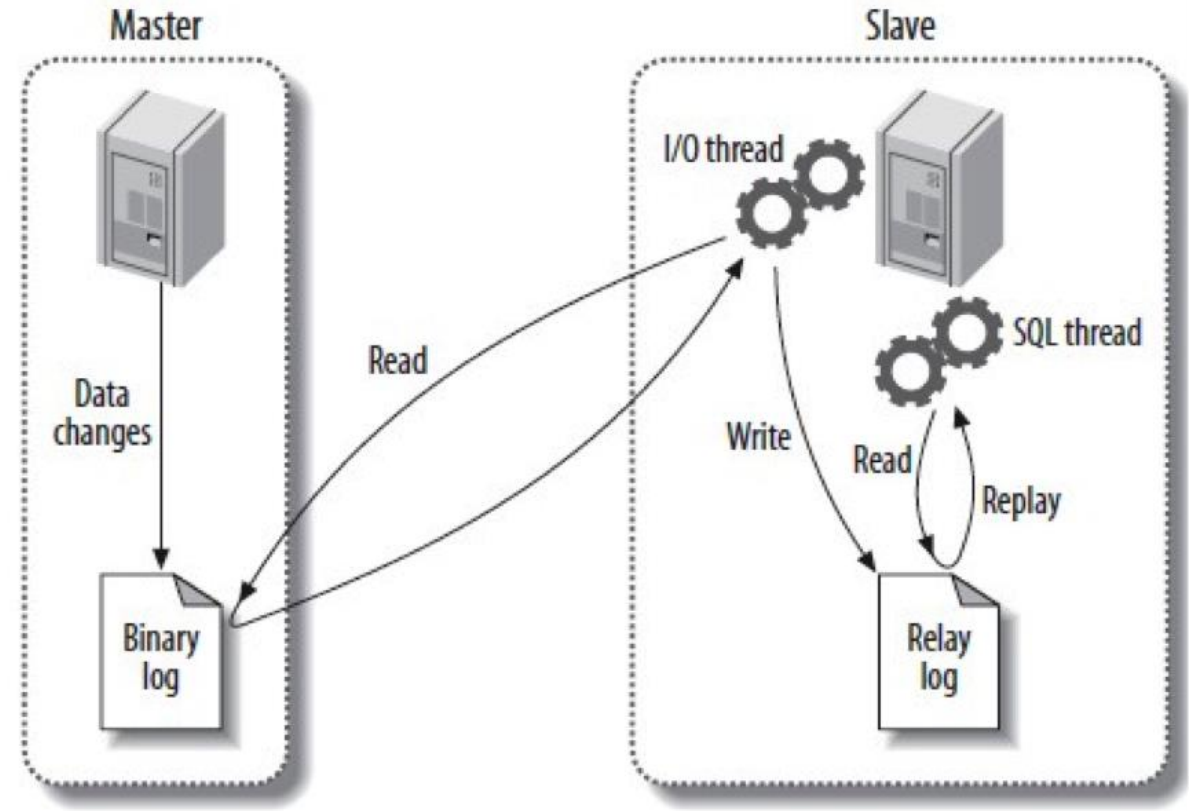
举例来说，假设某门户网站，它包含了新闻、用户、帖子、评论等等几大块内容，对于数据库来说，它可能包含这样几张表，news、users、post、comment，随着数据量的增加，可以根据业务逻辑进行拆分，分成多个库，以提高系统吞吐能力。



关系型数据库mysql—数据复制

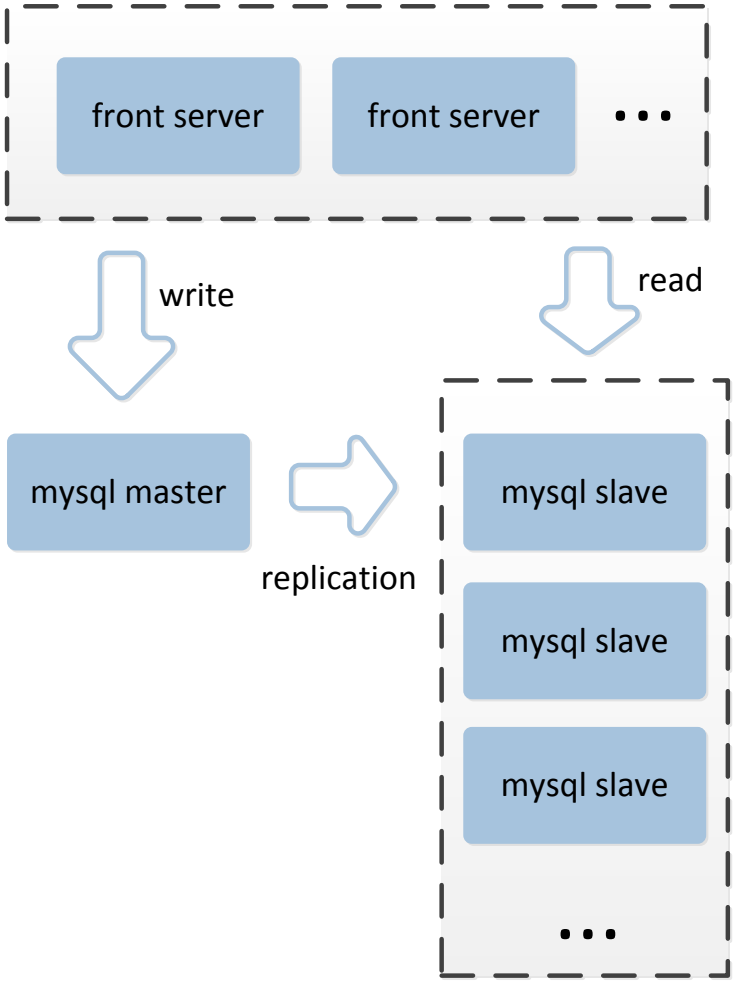
通过数据库的复制策略，可以将一台mysql数据库服务器中的数据复制到其他的mysql数据库服务器之上，当各台数据库服务器上都包含相同数据的时候，前端应用通过访问mysql集群中任意一台服务器，都能够读取到相同的数据，这样，每台mysql服务器所需要承担的负载就会大大降低，从而提高整个系统的承载能力，达到系统扩展的目的。

要实现数据库的复制，需要开启master服务器端的binary log，数据复制的过程实际上就是slave从master获取binary log，然后再在本地镜像的执行日志中所记录的操作，由于复制过程是异步的，因此，master和slave之间的数据有可能存在延迟的现象，此时只能够保证数据最终的一致性。



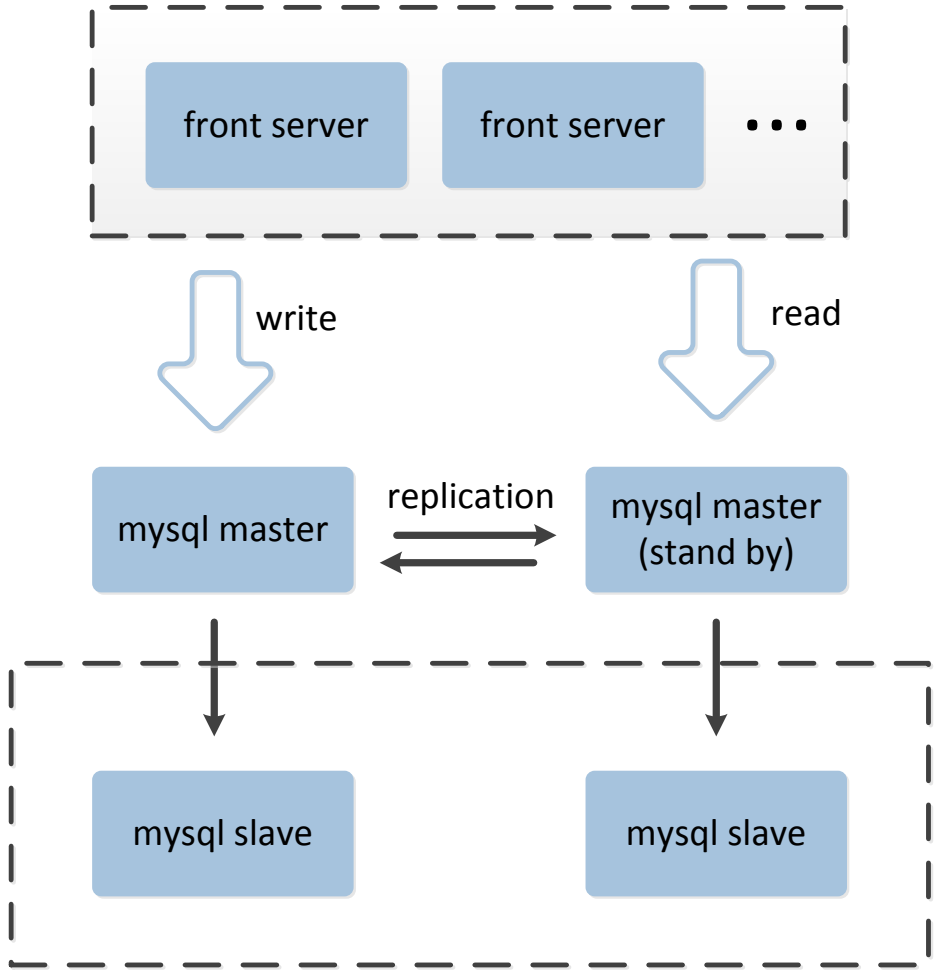
关系型数据库mysql—读写分离

前端服务器通过master来执行数据写入的操作，数据的更新通过binary log同步到slave集群，而对于数据读取的请求，则交由slave来处理，这样，slave集群可以分担数据库读的压力，并且，读写分离还保障了数据能够达到最终一致性。一般而言，大多数站点的读数据库操作要比写数据库操作更为密集，如果读的压力较大，还可以通过新增slave来进行系统的扩展，因此，master-slave的架构能够显著的减轻前面所提到的单库读的压力，毕竟在大多数应用当中，读的压力要比写的压力大的多。



关系型数据库mysql—dual-master架构

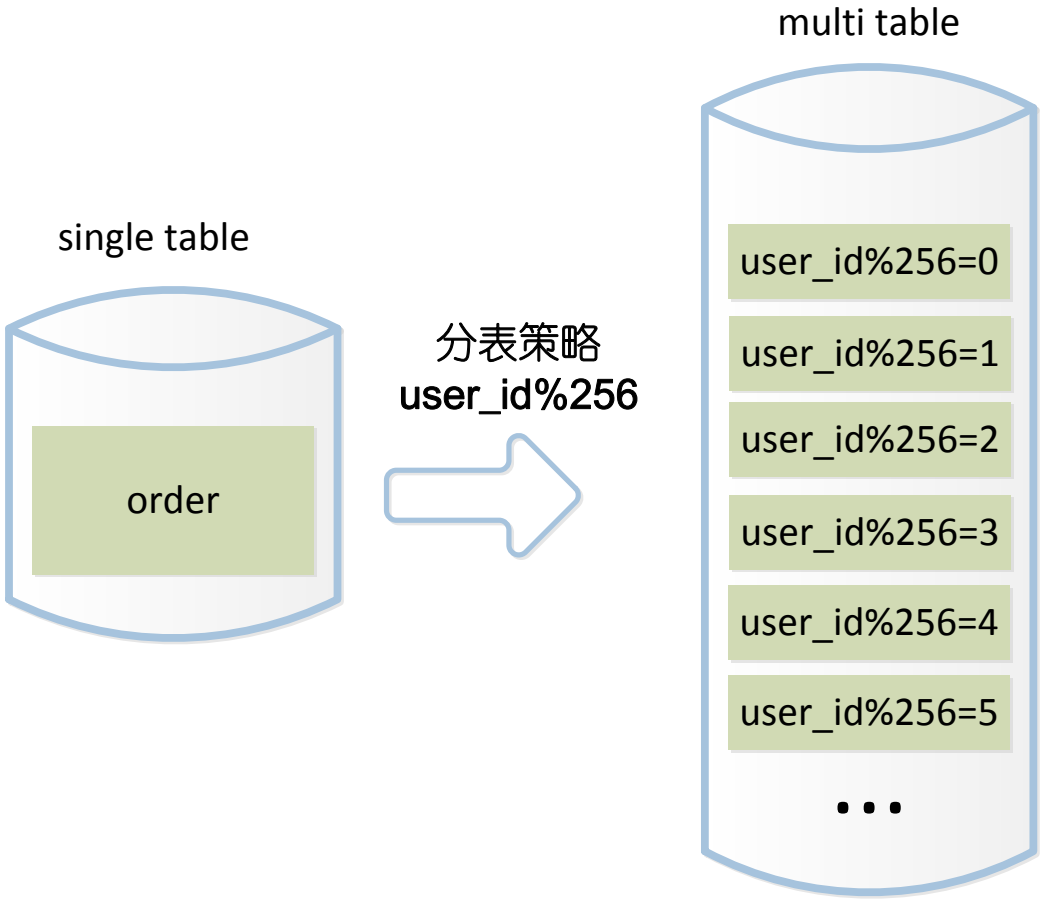
master-slaves复制架构存在一个问题，即所谓的单点故障，当master宕机，系统将无法写入，而在某些特定的场景下，可能需要master停机，以便进行系统维护、优化或者升级，同样的道理，master停机将导致整个系统都无法写入，直到master恢复，大部分情况下这显然是难以接受的。为了尽可能的降低系统停止写入的时间，最佳的方式就是采用dual master架构，即master-master架构



关系型数据库mysql—分表

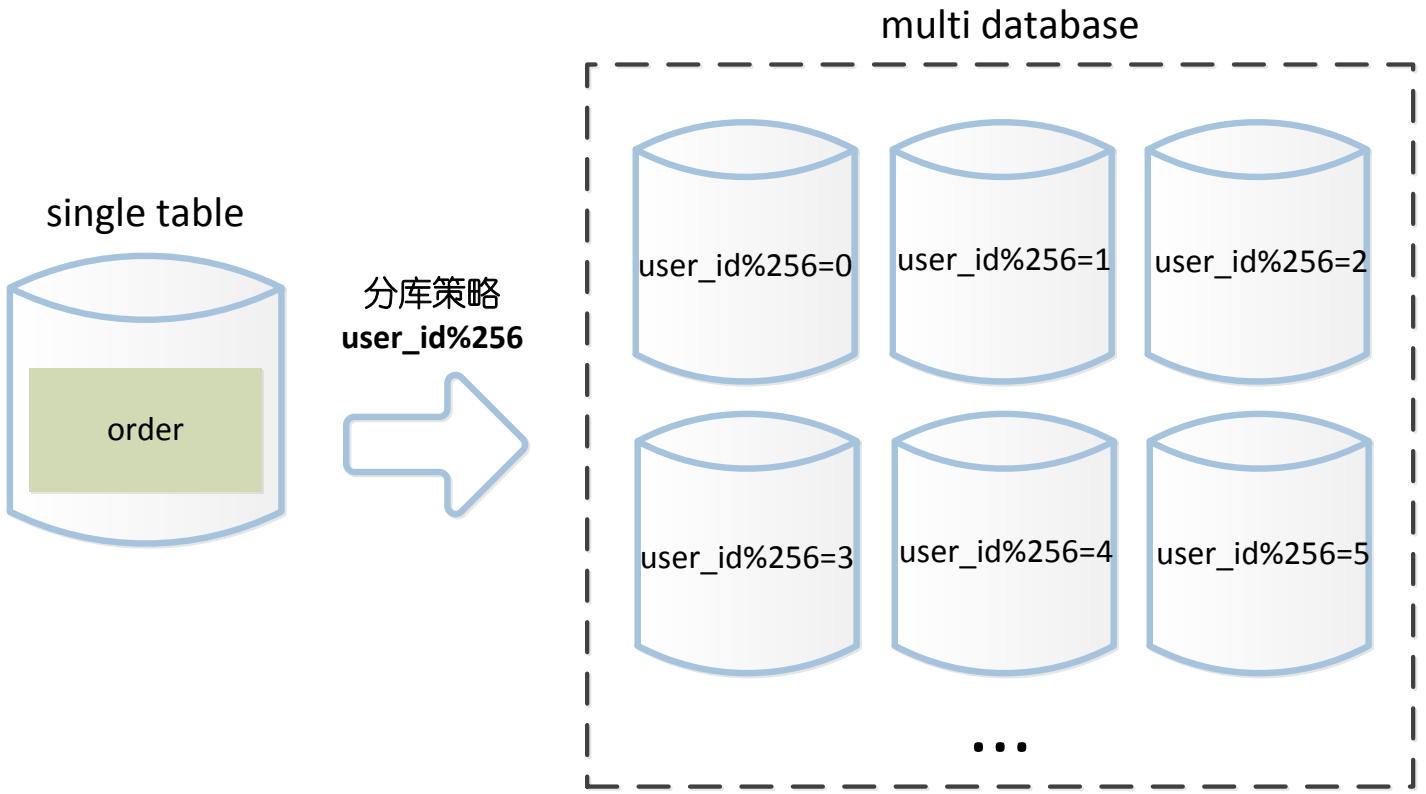
对于访问极为频繁且数据量巨大的单表来说，我们首先要做的，就是减少单表的记录条数，以便减少数据查询所需要的时间，提高数据库的吞吐，这就是所谓的**分表**。在分表之前，首先需要选择适当的分表策略，使得数据能够较为均衡的分布到多张表中，并且，不影响正常的查询。

对于互联网企业来说，大部分数据都是与用户进行关联的，因此，用户id是最常用的分表字段，因为大部分查询都需要带上用户id，这样既不影响查询，又能够使数据较为均衡的分布到各个表中。



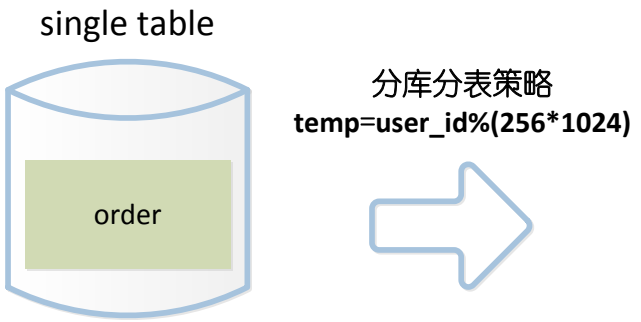
关系型数据库mysql—分库

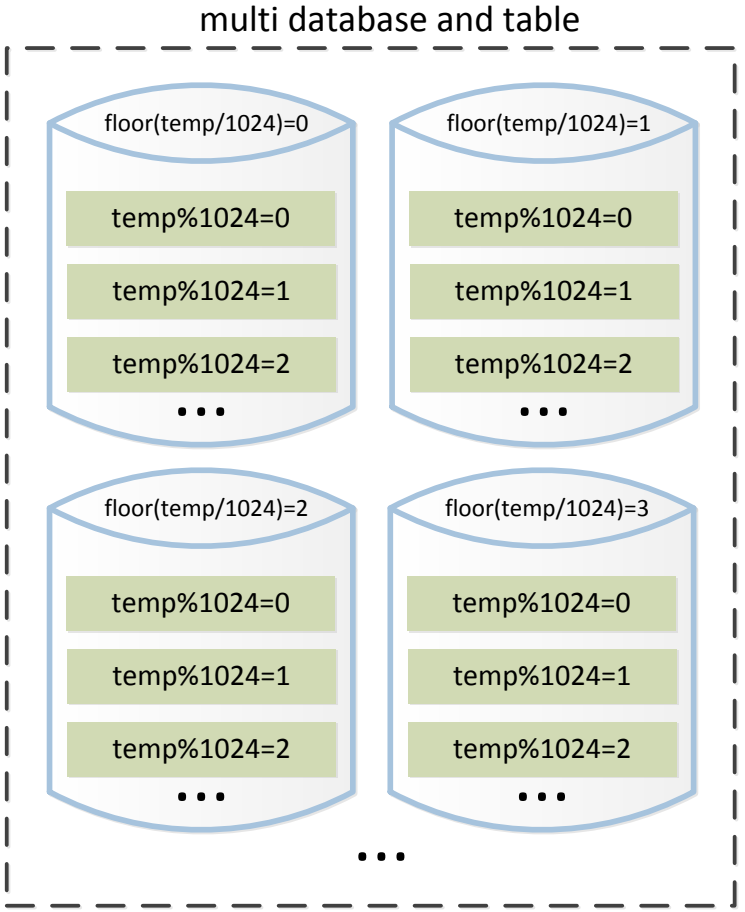
分表能够解决单表数据量过大带来的查询效率下降的问题，但是，却无法给数据库的并发处理能力带来质的提升，面对高并发的读写访问，当数据库master服务器无法承载写操作压力时，不管如何扩展slave服务器，此时都没有意义了。因此，我们必须换一种思路，对数据库进行拆分，从而提高数据库写入能力，这就是所谓的**分库**。



关系型数据库mysql—分库分表

有的时候，数据库可能即面临着高并发访问的压力，又需要面对海量数据的存储问题，这时候，需要对数据库即采用分库策略，又采用分表策略，以便同时扩展系统的并发处理能力，以及提升单表的查询性能，这就是所谓的**分库分表**。





中间变量=user_id%(库数量*每个库的表数量)
库=取整(中间变量/每个库的表数量)
表=中间变量%每个库的表数量

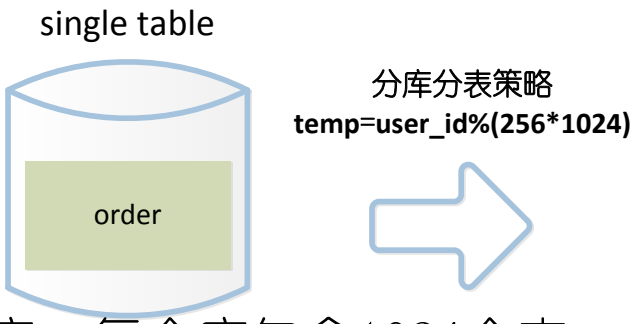
关系型数据库mysql—分库分表策略

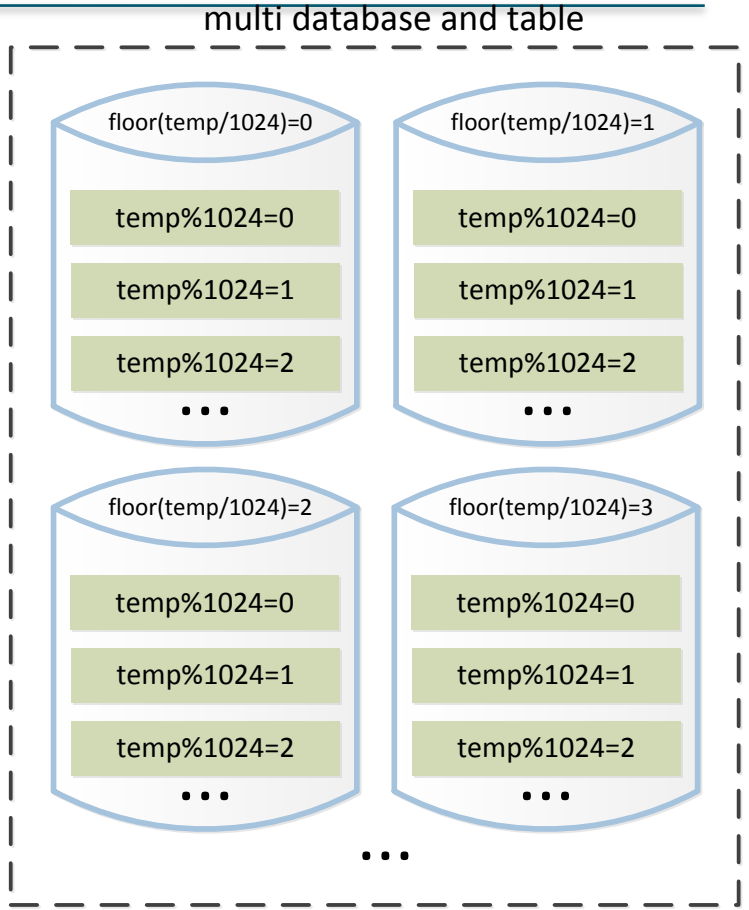
分库分表的策略比前面的仅分库或者仅分表的策略要更为复杂，一种分库分表的路由策略如下：

中间变量=user_id%(库数量*每个库的表数量)

库=取整(中间变量/每个库的表数量)

表=中间变量%每个库的表数量





假设将原来的单库单表order拆分成256个库，每个库包含1024个表，那么，按照前面所提到的路由策略，对于userid=262145的访问，路由的计算过程如下：

中间变量=262145%(256*1024)=1

库=取整(1/1024)=0

表=1%1024=1

这意味着，对于userid=262145的订单记录的查询和修改，将被路由到第0个库的第1个表中执行。

分库分表带来的限制

1. 条件查询、分页查询受到限制，查询必须带上分库分表所带上的id

2. 事务可能跨多个库，数据一致性无法通过本地事务实现，无法使用外键

3. 分库分表规则确定以后，扩展变更规则需要迁移数据

。。。。。。

Thanks

FAQ时间