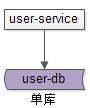
**无限容量数据库架构设计**

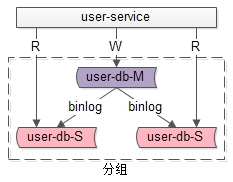
1. **数据库架构(以用户中心为例)**
2. ***单库体系架构***



User-service：用户中心服务，对调用者提供友好的RPC接口(Remote Procedure Calls 远程过程调用 (RPC) 是一种协议，程序可使用这种协议向网络中的另一台计算机上的程序请求服务)

User-db：一个库进行数据存储

1. ***数据库分组架构***



含义：一主多从，主从同步，读写分离的数据库架构

* User-service：用户中心服务
* User-db-M(master)：主库，提供数据库写服务
* User-db-S(slave)：从库，提供数据库度读服务

其中主库和从库构成的数据库集群称为“组”

特点：同一个组里的数据库集群：

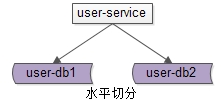
* 主从之间通过binlog进行数据同步
* 多个实例数据库结构完全相同
* 多个实例存储的数据也完全相同，本质是将数据进行复制

应用场景：大部分互联网业务读多写少，数据库的读往往最先成为性能瓶颈，则如果希望

* 线性提升数据库读性能
* 通过消除读写锁冲入提升数据库写性能
* 通过冗余从库实现数据的读高可用

分组架构中，数据库的主库依然是写单点；该架构解决的是“数据库读写高并发量”问题。

1. ***数据库分片架构***



含义：分片结构也就是常说的水平切分数据库架构：

* User-service：用户中心服务
* User-db1：水平切分成两份中的第一份
* User-db2：水平切分成两份中的第二份

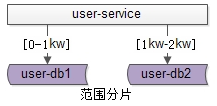
分片后，多个数据库实例也可以构成一个数据库集群

分表 or 分库：当然是分库

* 分表依然共用一个数据库文件，仍然有磁盘IO的竞争
* 分库能够很容易的将数据迁移到不同的数据库实例，乃至数据库机器上，扩展性更好

怎么实现水平切分：

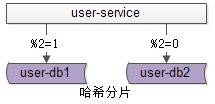
* 范围法：以用户中心的业务主键uid为划分依据，将数据水平切分到两个数据库实例上



User-db1：储存0到1千万的uid数据

User-db2：储存1千万到2千万的uid数据

* 哈希法(更广发)：以用户中心业务主键uid为划分依据，将数据水平切分到两个数据库实例上



User-db1：储存uid取模得1的uid数据

User-db2：储存uid取模得2的uid数据

分片的特点：同一个分片的数据库集群

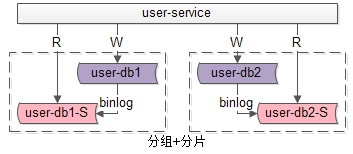
* 多个实例之间本身不直接产生联系，不像主从间有binlog同步
* 多个实例数据库结构，也完全相同
* 多个实例储存的数据之间没有交集，所有实例间数据并集构成全局数据

分片的应用场景：大部分互联网业务数据量很大，单库容量容易成为瓶颈，通过分片可以：

* 线性提升数据库写性能
* 降低单库数据容量

也就是解决的问题是“数据库数据量大”的问题

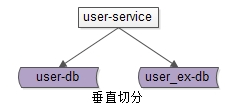
1. ***数据分组+分片架构***



如果业务读写并发都很高，数据量也大，通常采用该种架构：

* 通过分片来降低单库的数据量，线性提升数据库写性能
* 通过分组来线性提升数据的读性能，保证库的高可用

1. ***数据库垂直切分***



含义：以用户中心为例，可以这样进行垂直切分：

User(uid, uname, passwd, sex, age, …)

User\_EX(uid, intro, sign, …)

* 垂直切分开的表，主键都是uid
* 登录名，密码，性别，年龄等属性放在一个垂直表（库）里
* 自我介绍，个人签名等属性放在另一个垂直表（库）里

如何切分：根据业务对数据进行垂直切分时，一般要考虑属性的“长度”和“访问频度”：

* 长度较短，访问频率较高的发在一起
* 长度较长，访问频度较低的放在一起

这是由于数据库会以行为单位，将数load到内存中，在内存容量有限的情况下，长度短且访问频度高的属性，内存能够load更多的数据，命中率会更高，磁盘IO会减少，数据库性能会提升。

特点

* 多个实例之间不产生直接联系，即没有binlog同步
* 多个实例数据库结构都不一样
* 多个实例存储的数据之间至少一列交集，一般说都是业务主键，所有实例间数据并集构成全局数据

应用场景：

既可以降低单库数据量，还可以降低磁盘IO从而提升吞吐量，但是它与业务结合比较紧密，不是所有业务都能够进行垂直切分。

1. ***总结***

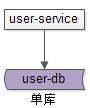
* 业务初期用单库
* 读压力大，读高可用，用分组
* 数据量大，写线性扩容，用分片
* 属性短，访问频度高的属性，垂直拆分到一起。

1. **实践一：单key业务，如何做到数据库无限容量(用户中心为例)**
2. ***用户中心***

这是一个非常常见的业务，主要提供用户注册、登陆、信息查询与修改的服务，其核心元数据为User(uid, login\_name, passwd, sex, age, nickname, …),其中：

* Uid为用户ID，主键
* login\_name, passwd, sex, age, nickname, …等用户属性

数据库设计上，一般来说在业务出去器，单裤单表就能搞定这样的需求，典型的架构设计为:



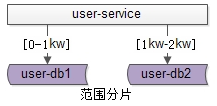
User-service：用户中心服务，对调用者提供友好的RPC接口(Remote Procedure Calls 远程过程调用 (RPC) 是一种协议，程序可使用这种协议向网络中的另一台计算机上的程序请求服务)

User-db：一个库进行数据存储

1. ***用户中心水平切分方法***

当数据量越来越大时，需要对数据库进行水平切分，常见的方法有“范围法”和“哈希法”

* 范围法：以用户中心的业务主键uid为划分依据，将数据水平切分到两个数据库实例上：



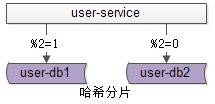
User-db1：储存0到1千万的uid数据

User-db2：储存1千万到2千万的uid数据

优点在于：

* 切分策略简单，根据uis，按照范围，user-center很快能够定位到数据在哪个库上
* 扩容简单，如果容量不够，主要增加user-db3即可

缺点在于：

* Uid必须满足递增的特性
* 数据量不均，新增的user-db3，在初期的数据会比较少
* 请求量不均，一般来说，新注册的用户活跃度会比较高，故user-db2往往负载更高，导致服务器利用率不平衡。
* 哈希法：以主键uid为划分依据，将数据水平切分到两个数据库实例上：
* 

User-db1：储存uid取模得1的uid数据

User-db2：储存uid取模得2的uid数据

优点在于：

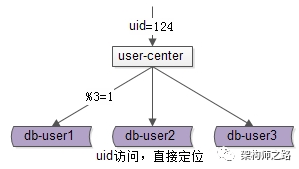
* 切分策略简单，根据uid，按照hash，user-center很快能够定位到数据在哪个库
* 数据量均衡，只要uid均匀，数据在各个库上的分布一定是均衡的
* 请求量均衡，只要uid均匀，负载在各个库上的分布一定是均衡的

缺点在于：

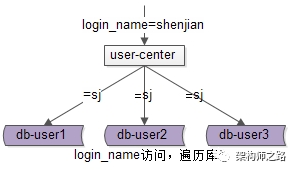
* 扩容麻烦，如果容量不够，要增加一个库，重新还是可能会导致数据迁移，如何平滑的进行数据迁移，是一个需要解决的问题

1. ***水平切分后带来的问题***

* 对于uid属性上的查询，可以直接路由到库



* 对于非uid属性上的查询：



由于不知道login\_name落在哪个库上，往往需要遍历所有库，当分库数量多起来，性能显著降低

解决方案：

- 首先对业务进行分析，看非uid属性上有哪些查询需求

A．用户侧，前台访问：基本上是单挑记录的查询，访问量较大，服务需要高可用，并发对一致性要求较高，主要有两类业务需求：1)用户登录，通过login\_name/phone/email查询用户的尸体；2)用户信息查询，登录之后，通过uid来查询用户的实例。

B．运营侧，后台访问：根据产品、运营需求，访问模式各异，按照不同字段查询。这类查询基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性要求不高，一致性也没那么严格

- 用户中心在数据量较大的情况下使用uid进行水平切分，对于非uid属性上的查询需求，架构设计的核心思路：

A．用户侧：采用“建立非uid属性到uid的映射关系的架构方案

B．运营侧：采用“前台于后台分离”的架构方案

-用户中心-用户侧实践

A．索引表法

1) 思路：

Uid能够直接定位到库，login\_name不能，如果通过login\_name查询到uid，问题解决

2) 解决方案：

~ 建立一个索引表来记录login\_name->uid的映射关系

~ 用login\_name来访问时，先通过索引表查询到uid再定位到相应的库

~ 索引表属性较少，可以容纳非常多的数据，一般不需要分库

~ 如果数据量过大，可以通过login\_name来分库

3) 潜在不足：

多一次数据库查询，性能下降一倍

B．缓存映射法

1) 思路：

访问索引表性能较低，把映射关系放在缓存里性能更佳

2)解决方案：

~Login\_name查询先到cache中查询uid，再根据uid定位数据库

~假设cache miss，采取扫描全库的方法获取login\_name对应的uid，放入cache

~login\_name到uid的映射关系不会变化，映射关系一旦放入缓存，不会更改，无需淘汰，缓存命中率超高

~如果数据量过大，可以通过login\_name进行cache水兵切分

3) 潜在不足：

多一次cache查询

C．login\_name生成uid

1) 思路：

不进行远程查询，有login\_name直接得到uid

2) 解决方案：

~用户注册时，设计函数login\_name生成uid，uid=f(login\_name)，按uid分库插入数据

~用login\_name来访问时，先通过函数计算出uid，由uid路由到对应库

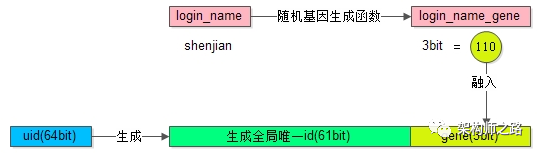
3) 潜在不足：

该函数设计技巧性高，有uid生成冲突的风险

D．login\_name基因融入uid

1）思路：

不能用login\_name生成uid，可以从login\_name抽取“基因”，融入uid中



假设分8库，采用uid%8,路由，潜台词是，uid的最后三个bit决定这条数据落在哪个库上，这三个bit就是所谓的“基因”

2）解决方案：

~在用户注册时，设计函数login\_name生成3bit基因，login\_name\_gene=f(login\_name）

~同时，生成61bit的全局唯一id，作为用户的标识

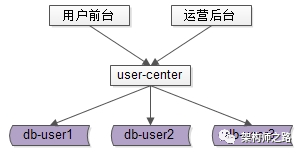
~接着把3bit的login\_name\_gene也作为uid的一部分

~生成64bit的全局唯一id，由id和login\_name\_gene拼装而成，并按照uid分库插入数据

~用login\_name来访问时，先通过login\_name再次复原3bit基因，通过login\_name\_gene%8直接定位到库

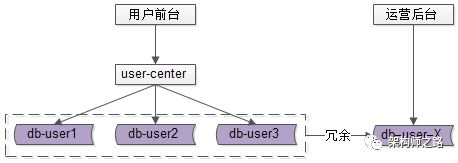
-用户中心-运营侧实践

A．后台业务需求各异，基本上是批量分页的访问，这类访问计算量较大，返回数据量较大，比较消耗数据库性能。如果前后台共用一批服务和一个数据库，有可能导致，由于后台的“少数几个请求”的“批量查询”的“抵消”访问，导致数据库cpu偶尔瞬时100%，影响前台正常用户的访问



B．为了满足后台业务各类“奇形怪状”的需求，往往会在数据库上建立各种索引，这些索引占用大量内存，会是的用户侧前台业务uid/login\_name上的读写性能大幅度降低，处理时间增长

C．由此剔除前后台分离的架构方案：



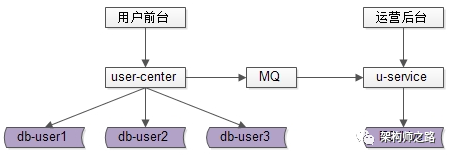
用户侧前台业务需求架构不变，产品运营侧后台业务则抽取独立的web/service/db来支持，解除系统之间的耦合，对于业务复杂、并发低、无需高可用、能接受一定延时的后台业务：

~去掉service层，在运营后台web层通过dao直接访问db

~不需要反向代理，不需要集群冗余

~不需要访问实时库，可以通过MQ或者线下异步同步数据

~在数据库非常大的时候，可以使用更切合大量数据允许接受高延时的“索引外置”(ES搜索系统)或者“Hive”的设计方案



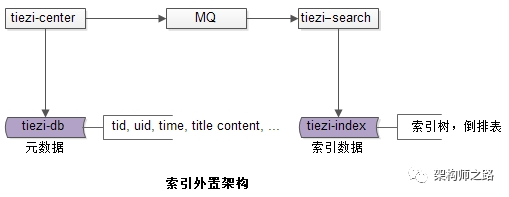
1. **实践二：一对多业务，如何做到数据库无限容量（以帖子中心为例）**
2. ***帖子中心业务分析***

* 写操作：发布帖子、修改帖子、删除帖子
* 读操作
* 通过tid查询帖子实体—单行查询
* 通过uid查询用户发布过的帖子—列表查询
* 帖子检索，通过时间、标题等搜索符合条件的帖子

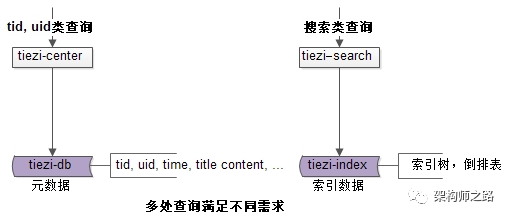
1. ***架构设计***

在数据量和并发量都较大的时候，通常通过元数据与索引数据分离的架构来满

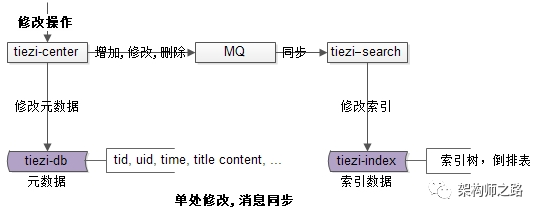
足不同的需求



* 关键点
* Tiezi-center：帖子服务
* Tiezi-db：提供元数据存储
* Tiezi-search：帖子搜索服务
* Tiezi-index：提供索引数据存储
* MQ：tiezi-center与tiezi-search通讯媒介，一般不直接使用RPC调用，而是通过对两个子系统解耦
* 对于读需求：



* Tid和uid上的查询需求，可以由tiezi-center从元数据读取并返回
* 其他类检索，由tiezi-search从检索数据检索并返回
* 对于写需求：



* 增加，修改，删除的操作都会从tiezi-center发起
* Tiezi-center修改元数据
* Tiezi-center将信息修改通知发送给MQ
* Tiezi-search从MQ接受修改信息
* Tiezi-search修改索引数据

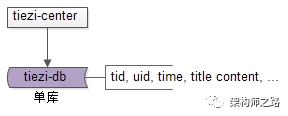
1. ***元数据设计***

通过对帖子中心的业务进行分析，其核心元数据：

Tiezi(tid,uid,title,content,…)

* 其中：
* tid为帖子id，主键
* uid为用户id，发帖人
* time，title，content…等为帖子属性
* 数据库设计

业务初期，单库就能满足元数据存储要求，典型架构为：



- tiezi-center：帖子中心服务，对调用者提供友好的RPC接口

- tiezi-db：对帖子数据进行存储

· 查询操作

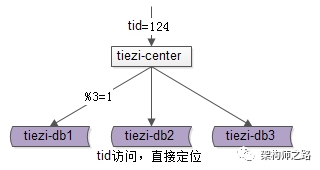
在相关字段上建立索引，就能满足相关业务需求：

- 帖子记录查询，通过tid查询，约占读请求量90%

- 帖子列表查询，通过uid查询，约占读请求量10%

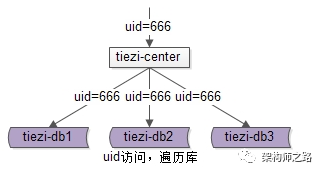
1. ***帖子中心水平切分-tid切分法***

当数据越来越大，需要对帖子数据的存储进行线性扩展。又因为帖子记录查询量占到了总请求量的90%，我们可以通过tid字段取模来进行水平切分：



* 优点：
* 100%的写请求直接定位到库
* 90%的渡请求可以直接定位到库
* 缺点：

一个用户发布的所有帖子可能会落到不同的库上，10%的请求通过uid来查询会比较麻烦，如图，一个uid访问需要遍历所有的库：



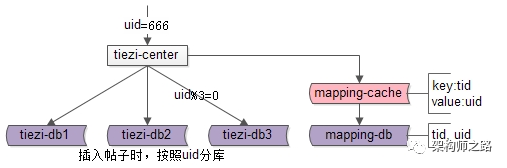
1. ***帖子中心水平切分-uid切分法***

通过uid分库来确保同一用户发布的是所有帖子都落在同一个库上，这时新出现一个问题，通过tid来查询，就不知道这个帖子落在哪个库上了，导致需要遍历全库，提出一个解决方案：

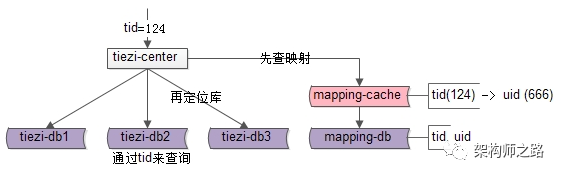
新增一个索引库：t\_mapping(tid,uid):

* 这个库只有两列，容纳很多数据
* 即使数据量过大，索引库可以利用tid水平切分
* 这类kv形式的索引结构，可以很好的利用cache优化查询性能
* 一旦帖子发布，tid和uid的映射关系就不会发生改变，cache命中率会非常高

使用uid分库，并增加索引库记录tid到uid的映射关系之后，每当有uid上的查询：



每当有tid上的查询：



* 先查询索引表，通过uid查询到对应的uid
* 在通过uid定位到库

这样做的优点在于：

* 一个用户发布的所有帖子落在同一个库上
* 10%的请求通过uid来查询列表，可以直接定位到库
* 索引表cache命中率非常高，因为tid与uid的映射关系不会变

缺点在于：

* 90%的tid请求以及100%的修改请求不能直接定位到库，需要先进行一次索引表的查询(虽然很快)
* 数据插入时需要操作数据与索引表，可能引发潜在的一致性问题

1. ***帖子中心水平切分-基因法(一对多业务分库最佳实践)***

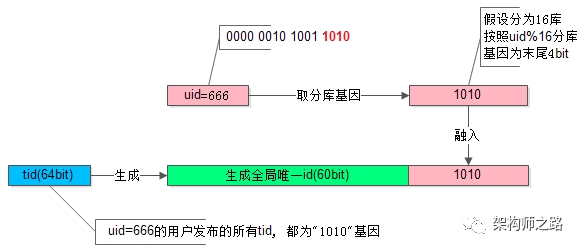
它既能够通过uid定位到库，又不需要建立索引表来进行二次查询。

* 分库基因

通过uid分库，假设分为16个库，采用uid%16的方式来进行数据库路由，这里的uid%16，其本质是uid的最后4个bit决定这行数据落在哪个库上，这4个bit就是分库基因

* 基因法分库

在“1对多”的业务场景，使用“1”分库，在“多”的数据id生成时，id末尾加入分库基因，就能同时满足“1”和“多”的分库查询需求。



如图，uid=666的用户发布了一条帖子（666二进制表示为1010011010）：

* 使用uid%16分库，决定这行数据要插入到哪个库中
* 分库基因是uid的最后4个bit，即1010
* 生成tid时，先使用一种分布式id生成算法生成前60bit
* 将分库基因加入到tid的最后4个bit
* 拼装成最终64bit帖子tid

这样保证了同一用户发布的所有帖子的tid都落在同一个库上，tid的最后4个bit相同，于是：

* 通过uid%16能够定位到库
* 通过tid%16也能够定位到库

这样的潜在问题有两个：

* 同一个uid发布的tid 落在同一个库，会不会造成数据不均衡？

只要uid均衡，每个用户平均发布帖子是均衡的，每个库的数据就是均衡的

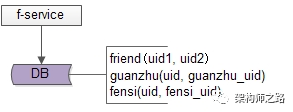
* 最开始分16库，分库基因是4bit，未来要扩充成32库，分库基因变成了5bit，怎么破？
* 提前做好容量评估，例如事先规划好未来5年内数据增长256库足够，就提前预留8bit基因

1. **实践三——多对多业务，如何做到数据库无限容量（以好友中心为例）**
2. ***好友中心业务分析***

好友关系分为两类，强好友关系(如qq)和弱好友关系（如weibo）：



好友中心是一个典型的“多对多”关系，一个用户可以添加多个好友，也可以被多个好友添加，典型架构如下：



* Friend-service：好友中心服务，对调用者提供友好的RPC接口
* Db：对好友数据进行储存

1. ***弱好友关系-元数据简版实现***

通过弱好友关系业务分析，其核心元数据为：

* guanzhu(uid,guanzhu\_uid):记录uid所关注的用户的uid
* fensi(uid,fensi\_uid):记录uid所有粉丝用户的uid

需要注意的是，一条弱关系的产生会产生两条记录，一条关注记录，一条粉丝记录

1. ***强好友关系-元数据实现(一)***

通过强好友关系的分析，其核心元数据：friend(uid，uid2)

其中，uid1和uid2是强好友关系的两方，在插入数据库的时候可以规定uid>uid2

[查询时候尽量避免在where子句中使用or来连接条件，否则将导致引擎放弃使用索引而进行全表扫描]

1. ***强好友关系-元数据实现(二)***

强好友关系其实算是弱好友关系的特例，根据这一特性，可以使用弱好友关系的元数据进行实现，即数据冗余的guanzhu表和fensi表(正表T1和反表T2)。

1. ***数据冗余是实现多对多关系水平切分的常用实践***

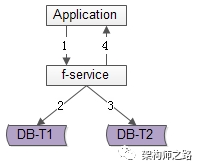
在数据量小的时候，强好友关系的两类实现其实差距不大，但当数据量大时，数据冗余的优势就会体现。

* friend表，数据量大时，如果使用uid1来分库，那么uid2上的查询就需要遍历多库
* 正表T1和反表T2通过数据冗余来实现好友关系，{1,2}{2,1}分别存在于两表中，故两个表都可以使用uid分库，且均需要进行一次查询，就能找到对应的关注与粉丝，而不需要全库扫描。

数据冗余，是多对多关系在数据量大时，数据水平切分的常用实践。

1. ***如何进行数据冗余***

* 方法一：服务同步冗余



由好友中信服务同步写冗余数据，

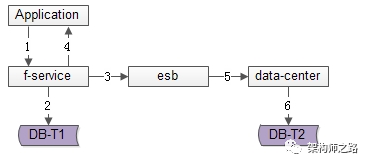
* 业务方调用服务，新增数据
* 服务先插入T1数据，再插入T2数据
* 服务返回业务方新增数据成功

这样做的优点在于：

* 不复杂，服务层又单次写变为两次写
* 数据一致性相对较高(因为双写成功才返回)

也存在这缺点：

* 请求的处理时间增加
* 数据仍可能不一致(T1完成后服务重启，T2中也没有数据写入)
* 方法二：服务异步冗余



数据的双写不再由好友中心服务来完成，服务层异步发出一个消息，通过消息总线发送给一个专门的数据复制服务来写入冗余是数据，

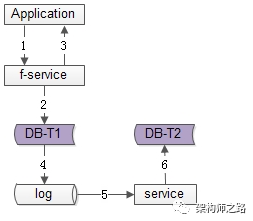
* 业务方调用服务新增数据
* 服务先插入T1数据
* 服务想消息总线发送一个异步消息(发出即可，不用等返回，通常很快完成)
* 服务返回业务方新增数据成功
* 消息总线将消息传递给数据同步中心
* 数据同步中心插入T2数据

这样做的优点在于：

* 请求处理时间短

缺点在于：

* 系统复杂性增加，多引入了一个消息总线组件和一个专用的数据复制服务
* 因为返回业务线数据插入成功，数据还不一定插入到T2中，因此数据有一个不一致时间窗口(虽然很短并且最终一致)
* 在消息总线丢失消息时，冗余表会不一致
* 方法三：线下异步冗余(解除了“数据”冗余对系统的耦合)



数据的双写不再由好友中心服务来完成，二十由线下的一个服务或者任务来完成：

* 业务方调用服务，新增数据
* 服务先插入T1数据
* 服务返回业务方新增数据成功
* 数据会被写入到数据库的log中
* 线下服务或者任务读取数据库的log
* 线下服务或者任务插入T2数据

这样的优点在于：

* 数据双写与业务完全解耦
* 请求处理时间短

这样的缺点在于：

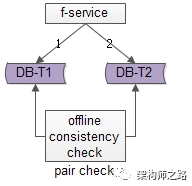
* 返回业务线数据插入成功时，数据还不一定插入到T2中，因此数据有一个不一致的时间窗口(虽然很短而且最终会一致)
* 数据的一致性依赖于线上服务或者任务的可靠性

数据冗余解决多对多关系的数据库水平切分问题，但是新的问题又出现了，如何保证正表T1与反表T2的数据一致性。

1. ***如何保证数据的一致性***

第六段关于数据冗余的问题，两部操作总是无法保证原子性，总有出现数据不一致的可能，高吞吐分布式事务是业内尚未解决的难题，此时的架构优化方向，并不是完全保证数据的一致，而是尽早地发现不一致，并修复不一致。最终修复不一致。

* 线下扫描正反冗余表全部数据



如上，线下启动一个离线的扫描工具，不停的比对正表T1和T2，如果发现数据不一致，就进行补偿修复

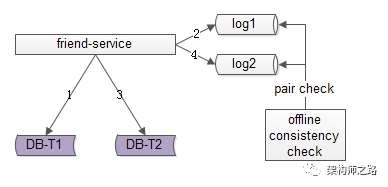
这样的优点在于：

* 比较简单，开发代价小
* 线上服务无需修改，修复工具与线上服务解耦

缺点在于：

* 扫描效率低，会扫描大量“已经能够保证一致”的数据
* 由于扫描的数据量大，扫描一轮的时间比较长，即数据如果不一致，则不一致的时间窗口比较长
* 扫描增量式数据

每次只扫描增量的日志数据，就能够极大提高效率，缩短数据不一致的时间窗口：



* 写入正表T1
* 第一步成功后，写入log1
* 写入反表T2
* 第二步成功后，写入日志log2

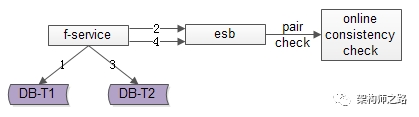
当然还需要一个离线扫描工具，不停的比对日志log1和log2，如果发现数据不一致，就进行补偿修复。

这样的优点在于：

* 虽比方法一复杂，但仍然比较简单
* 数据扫描效率高，只扫描增量数据

这样的缺点在于：

* 线上服务略有修改(代价不高，多写了两条日志)
* 虽然比方法一更实时，但时效还是不高，不一致窗口取决于扫描的周期
* 实时线上“消息对”检测



这次不写日志，而是向消息总线发送消息，

* 写入正表T1
* 第一步成功后，发送消息msg1
* 写入反表T2
* 第二步成功后，发送消息msg2

这次需要一个实时订阅消息的服务不停的收消息。假设正常情况下msg1和msg2的接收时间应该在3s以内，如果检测服务在收到msg1后没有收到msg2，就尝试检测数据的一致性，不一致时进行补偿修复

这样做的优点在于：

* 效率高
* 实时性告

这样作业有缺点：

* 方案比较复杂，线上引入了消息总线这个组件
* 线下多了一个订阅总线的检测服务

技术方案本身就是一个投入产出比的折中，可以根据业务对一致性的需求程度来决定使用哪一种方法。

1. **实践四——多key业务，如何做到数据库无限容量（以订单中心为例）**
2. ***业务分析***

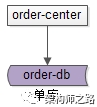
* 多key类业务

指一条元数据有多个属性存在前台在线查询需求

* 订单中心业务

订单中心是一个非常常见的“多key业务”，主要提供订单的查询与修改的服务，核心元数据为Order(oid,buyer\_uid,seller\_uid,time,money,detail…)。

数据库设计上，在业务初期，单库单表就能够搞定这个需求，典型架构如下：



* order-center：订单中心服务，对调用者提供友好的RPC接口
* order-db：对订单进行数据存储

随着订单量的增加，数据库需要进行水平切分，由于存在多个key上的查询需求，用哪个字段进行切分，成了需要解决的关键技术问题：

* 如果用oid切分，buyer\_uid和seller\_uid上的查询则需要遍历多库
* 如果用buyer\_uid或者seller\_uid来切分，其他属性上的查询则需要遍历多库

1. ***订单中心属性查询需求分析***

前台访问，最典型的有三类需求：

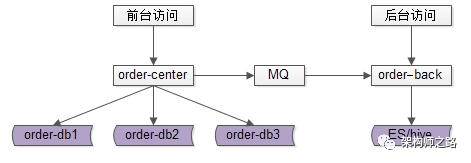
* 订单实体查询：通过oid查询订单实体，90%流量属于这类需求
* 用户订单列表查询：通过buyer\_uid分页查询用户历史订单列表，9%的流量属于这类需求
* 商家订单列表查询，通过seller\_uid分页查询商家历史订单列表，1%流量属于这类需求

前台访问的特点：吞吐量大，服务要求高可用，用户对订单的访问一致性要求高，商家对订单的访问一致性要求较低，可以接受一定时间的延时

后台访问，根据产品，运营需求，访问模式各异：按照时间，架构，商品，详情来进行查询。特点有：运营侧的查询基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性要求不高，对一致性要求也不严格，允许秒级升值十秒级别的查询延时

1. ***前台和后台分离的架构设计***

如果前台业务和后台业务共用一批服务和一个数据库，有可能导致，由于后台“少数几个请求”的批量查询和“低效”访问，导致数据库的菜谱偶尔瞬间100%，影响前台用户的访问，前台和后台查询需求不同，对系统的要求也不一样，故应该两者解耦，实施“前台与后台分离”的架构设计：

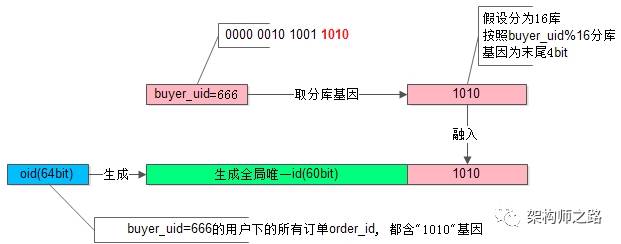


前台业务不变，站点访问，服务分层，数据库水平切分；后台业务需求则抽取独立的web/service/db来支持，解除系统之间的耦合，对于“业务复杂”“并发量低”“无需高可用”“能接受一定的延时”的后台业务：

* 去掉service层，在运营后台web层通过dao直接访问数据层
* 可以不需要反向代理，不需要集群冗余
* 可以通过MQ或者线下异步同步数据，牺牲一些数据的实时性
* 可以使用更契合大量数据允许接受高延时的索引外置或者HIVE设计方案

1. ***前台数据库水平切分***

* 假设没有seller\_uid上的查询需求，而只有oid和buyer\_uid的查询需求，就退化为一个“一对多”的业务场景，对于“一对多”的业务，水平切分应该使用“基因法”

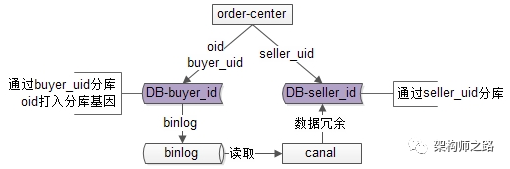


通过buyer\_uid分库，假设分为十六个库，采用buyer\_uid%16的方式来进行数据库路由，所谓的模16，其本质是buyer\_uid的最后4个bit决定这行落在哪个库，这四个bit就是分库基因。在订单数据oid生成时oid末端加入分库基因，让同一个buyer\_uid下的所有订单都含有相同基因落在同一个分库上，如上图。

通过这种方法保证，同一个用户下的所有订单oid都落在用一个库上，oid的最后4个bit都相同，于是：

1. 通过buyer\_uid%16能够定位到库
2. 通过oid%16也能定位到库

* 假设没有oid上的查询，而只有buyer\_uid和seller\_uid上的查询需求，就退化成一个多对多的业务场景，对于多对多的业务，水平切分应该使用“数据冗余法”：



当有订单生成，通过buyer\_uid分库，oid中融入分库基因，写入DB-buyer库，通过线下异步的方式，通过binlog+canal将数据冗余到DB-seller库中，buyer库通过buyer\_uid分库，seller库通过seller\_uid分库，前者满足oid和buyer\_uid的查询需求，后者，满足seller\_uid的查询需求。

* Oid/buyer\_uid/seller\_uid同时存在，采取上两种方案的综合，通过buyer\_uid分库，oid加入分库基因，seller\_uid数据冗余。