从用户中心开始，聊“单KEY”类业务数据库水平切分架构实践

本文将以“用户中心”为例，介绍**“单KEY”类业务**，随着数据量的逐步增大，数据库性能显著降低，数据库水平切分相关的架构实践：

- 如何来实施水平切分。
- 水平切分后常见的问题。
- 典型问题的优化思路及实践。

一、用户中心

用户中心是一个非常常见的业务，主要提供用户注册、登录、信息查询与修改的服务，其核心元数据为：

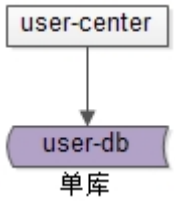
GitChat

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，一般来说在业务初期，单库单表就能够搞定这个需求，典型的架构设计为：



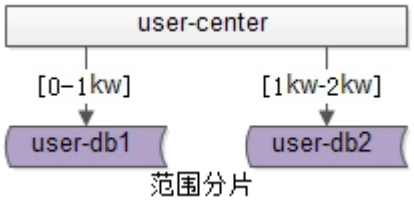
- user-center：用户中心服务，对调用者提供友好的RPC接口。
- user-db：对用户进行数据存储。

二、用户中心水平切分方法

当数据量越来越大时，需要多用户中心进行水平切分，常见的水平切分算法有“**范围法**”和“**哈希法**”。

范围法

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去：



- user-db1：存储0到1千万的uid数据。
- user-db2：存储0到2千万的uid数据。

范围法的**优点**是：

- 切分策略简单，根据uid，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，只要增加user-db3即可。

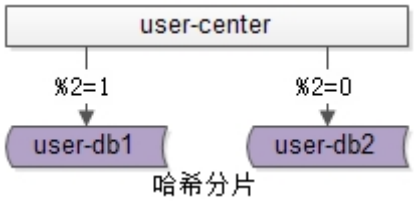
GitChat

范围法的**不足**是：

- uid必须要满足递增的特性。
- 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往往会比user-db1负载要高，导致服务器利用率不平衡。

哈希法

哈希法：也是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去：



- user-db1：存储uid取模得1的uid数据。
- user-db2：存储uid取模得0的uid数据。

哈希法的**优点**是：

- 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上。
- 数据量均衡，只要uid是均匀的，数据在各个库上的分布一定是均衡的。
- 请求量均衡，只要uid是均匀的，负载在各个库上的分布一定是均衡的。

哈希法的**不足**是：

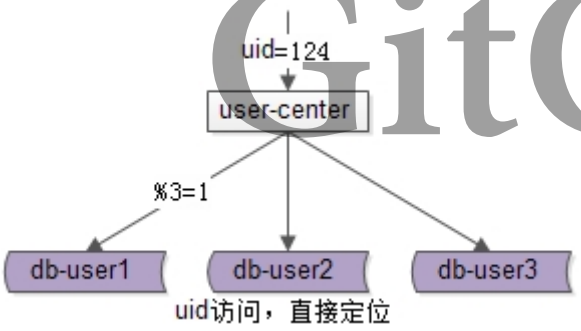
- 扩容麻烦，如果容量不够，要增加一个库，重新hash可能会导致数据迁移，如何平滑的进行数据迁移，是一个需要解决的问题

三、用户中心水平切分后带来的问题

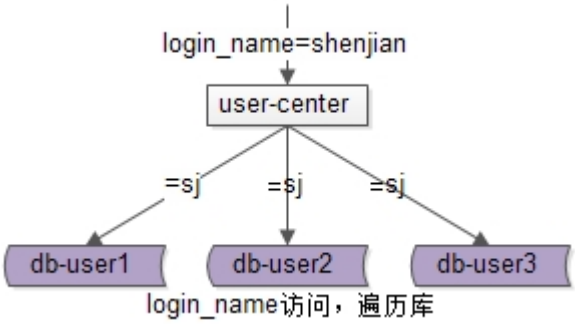
使用uid来进行水平切分之后，整个用户中心的业务访问会遇到什么问题呢？

对于uid属性上的查询可以直接路由到库，假设访问uid=124的数据，取模后能够直接定位db-user1：

GitChat



对于非uid属性上的查询，例如login_name属性上的查询，就悲剧了：



假设访问login_name=shenjian的数据，由于不知道数据落在哪个库上，往往需要遍历所有库，当分库数量很多时，性能会显著降低。

如何解决分库后，非uid属性上得查询问题，是后文要重点讨论的内容。

四、用户中心非uid属性查询需求分析

任何脱离业务的架构设计都是耍流氓，在进行架构讨论之前，先来对业务进行简要分析，看非uid属性上有哪些查询需求。

根据楼主这些年的架构经验，用户中心非uid属性上经常有两类业务需求：

- 用户侧，前台访问，最典型的有两类需求。
    - **用户登录**：通过login_name/phone/email查询用户的实体，1%请求属于这种类型
    - **用户信息查询**：登录之后，通过uid来查询用户的实例，99%请求属这种类型

    用户侧的查询基本上是单条记录的查询，访问量较大，服务需要高可用，并且对一致性的要求较高。
- 运营侧，后台访问，根据产品、运营需求，访问模式各异。

    按照年龄、性别、头像、登陆时间、注册时间来进行查询。

    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，对可用性的要求不高，对一致性的要求也没这么严格。

GitChat

这两类不同的业务需求，应该使用什么样的架构方案来解决呢？

五、用户中心水平切分架构思路

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为：

- 针对用户侧，应该采用“**建立非uid属性到uid的映射关系**”的架构方案。
- 针对运营侧，应该采用“**前台与后台分离**”的架构方案。

六、用户中心-用户侧最佳实践

索引表法

思路：uid能直接定位到库，login_name不能直接定位到库，如果通过login_name能查询到uid，问题解决。

解决方案：

- 建立一个索引表记录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，再定位相应的库。
- 索引表属性较少，可以容纳非常多数据，一般不需要分库。
- 如果数据量过大，可以通过login_name来分库。

潜在不足：多一次数据库查询，性能下降一倍。

缓存映射法

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。

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，扫描全库获取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水平切分。

GitChat

潜在不足：多一次cache查询。

login_name生成uid

思路：不进行远程查询，由login_name直接得到uid.

解决方案：

- 在用户注册时，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，即uid=f(login_name)再来一遍，由uid路由到对应库。

潜在不足：该函数设计需要非常讲究技巧，有uid生成冲突风险。

login_name基因融入uid

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3个bit决定这条数据落在哪个库上，这3个bit就是所谓的“基因”。

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)，如上图粉色部分。
- 同时，生成61bit的全局唯一id，作为用户的标识，如上图绿色部分。
- 接着把3bit的login_name_gene也作为uid的一部分，如上图屎黄色部分。
- 生成64bit的uid，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，先通过函数由login_name再次复原3bit基因。
- login_name_gene=f(login_name)，通过login_name_gene%8直接定位到库。

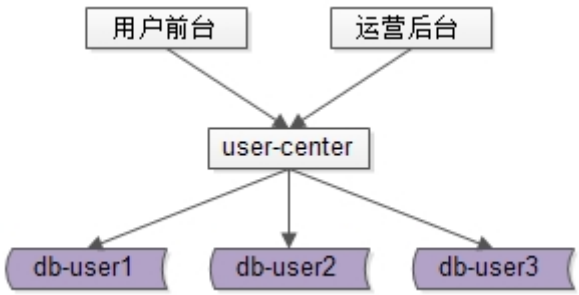
GitChat

七、用户中心-运营侧最佳实践

前台用户侧，业务需求基本都是单行记录的访问，只要建立非uid属性login_name/phone/email到uid的映射关系，就能解决问题。

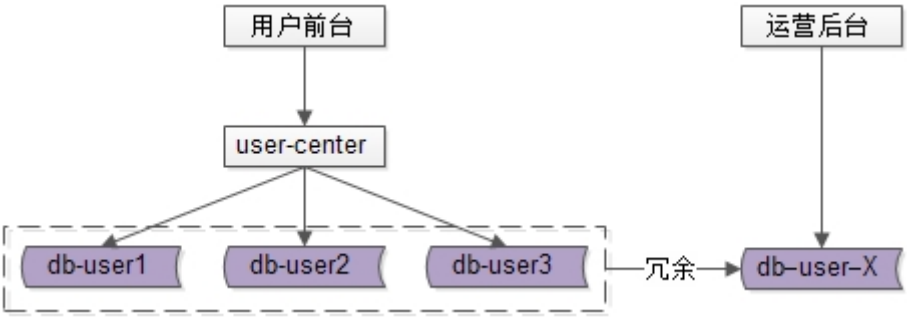
后台运营侧，业务需求各异，基本是批量分页的访问，这类访问计算量较大，返回数据量较大，比较消耗数据库性能。

如果此时前台业务和后台业务公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“低效”访问，导致数据库的cpu偶尔瞬时100%，影响前台正常用户的访问（例如，登录超时）。



而且，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，会使得用户侧前台业务uid/login_name上的查询性能与写入性能大幅度降低，处理时间增长。

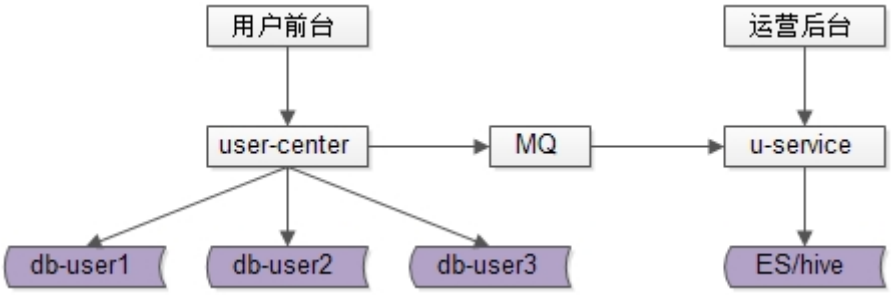
对于这一类业务，应该采用“前台与后台分离”的架构方案：



用户侧前台业务需求架构依然不变，产品运营侧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”“能接受一定延时”的后台业务：

- 可以去掉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。
- 在数据库非常大的情况下，可以使用更契合大量数据允许接受更高延时的“索引外置”或者“HIVE”的设计方案。

GitChat



八、总结

本文以“用户中心”为典型的 “单KEY”类业务，水平切分的架构点，做了这样一些介绍。

水平切分方式：

- 范围法
- 哈希法

水平切分后碰到的问题：

- 通过uid属性查询能直接定位到库，通过非uid属性查询不能定位到库。

非uid属性查询的典型业务：

- 用户侧，前台访问，单条记录的查询，访问量较大，服务需要高可用，并且对一致性的要求较高。
- 运营侧，后台访问，根据产品、运营需求，访问模式各异，基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性的要求不高，对一致性的要求也没这么严格。

这两类业务的架构设计思路：

- 针对用户侧，应该采用“建立非uid属性到uid的映射关系”的架构方案。
- 针对运营侧，应该采用“前台与后台分离”的架构方案。

用户前台侧，“建立非uid属性到uid的映射关系”最佳实践：

- 索引表法：数据库中记录login_name->uid的映射关系。
- 缓存映射法：缓存中记录login_name->uid的映射关系。
- login_name生成uid.
- login_name基因融入uid.

GitChat

运营后台侧，“前台与后台分离”最佳实践：

- 前台、后台系统web/service/db分离解耦，避免后台低效查询引发前台查询抖动。
- 可以采用数据冗余的设计方式。
- 可以采用“外置索引”（例如ES搜索系统）或者“大数据处理”（例如HIVE）来满足后台变态的查询需求。

九、还有哪些未尽事宜

- 以帖子中心为典型的 “1对多”类业务。
- 以好友关系为典型的“多对多”类业务。
- 以订单中心为典型的“多KEY”类业务。

他们的水平拆分架构又应该怎么处理，敬请期待下期。

希望大家有收获。