从订单中心开始，聊“多KEY”类业务数据库水平切分架构实践

不知不觉，水平切分系列文章已经和大家相伴走过半年，介绍了“单key”，“一对多”，“多对多”等不同业务场景下，水平切分的方式方法与最佳实践。

本篇讲义将以“订单中心”为例，介绍“多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。

一、什么是“多key”类业务

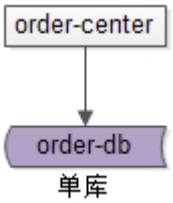
所谓的“多key”，是指一条元数据中，有多个属性上存在前台在线查询需求。

订单中心业务分析

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，其核心元数据为：
Order(oid, buyer_uid, seller_uid, time, money, detail...);其中：

1. oid为订单ID，主键
2. buyer_uid为买家uid
3. seller_uid为卖家uid
4. time, money, detail, …等为订单属性

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：



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

随着订单量的越来越大，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：

1. 如果用oid来切分， `buyer_uid` 和 `seller_uid` 上的查询则需要遍历多库。
2. 如果用 `buyer_uid` 或 `seller_uid` 来切分，其他属性上的查询则需要遍历多库。

总之，很难有一个完全之策，在展开技术方案之前，先一起梳理梳理查询需求。

二、订单中心属性查询需求分析

在进行架构讨论之前，先来对业务进行简要分析，看哪些属性上有查询需求。

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%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相对较低，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。

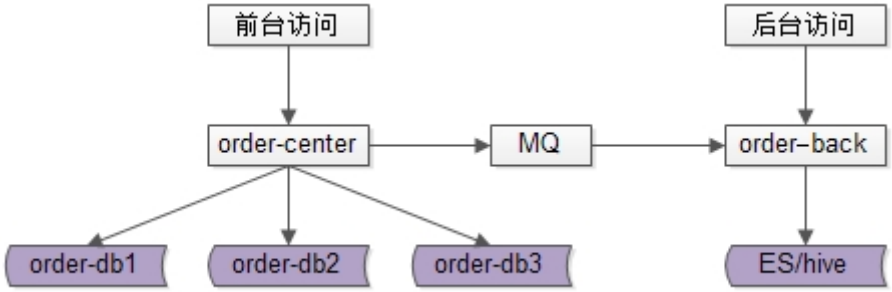
运营侧的查询基本上是批量分页的查询，由于是内部系统，访问量很低，对可用性的要求不高，对一致性的要求也没这么严格，允许秒级甚至十秒级别的查询延时。

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？

三、前台与后台分离的架构设计

如果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，订单查询超时）。

前台与后台访问的查询需求不同，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。



前台业务架构不变，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更高延时的“索引外置”或者“HIVE”的设计方案。

解决了后台业务的访问需求，问题转化为，前台的oid，buyer_uid，seller_uid如何来进行数据库水平切分呢？

多个维度的查询较为复杂，对于复杂系统设计，可以逐步简化。

四、假设没有seller_uid

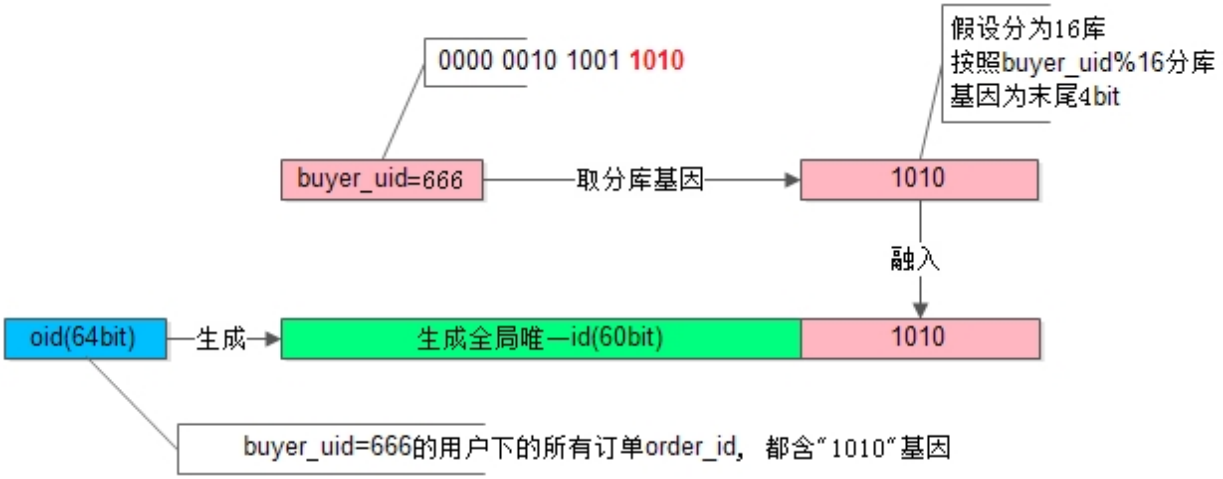
订单中心，假设没有 `seller_uid` 上的查询需求，而只有oid和 `buyer_uid` 上的查询需求，就蜕化为一个“1对多”的业务场景，对于“1对多”的业务，水平切分应该使用“基因法”。

再次回顾一下，什么是分库基因？

通过 `buyer_uid` 分库，假设分为16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决定这行数据落在哪个库上，这4个bit，就是分库基因。

也再次回顾一下，什么是基因法分库？

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。



如上图所示，buyer_uid=666的用户下了一个订单：

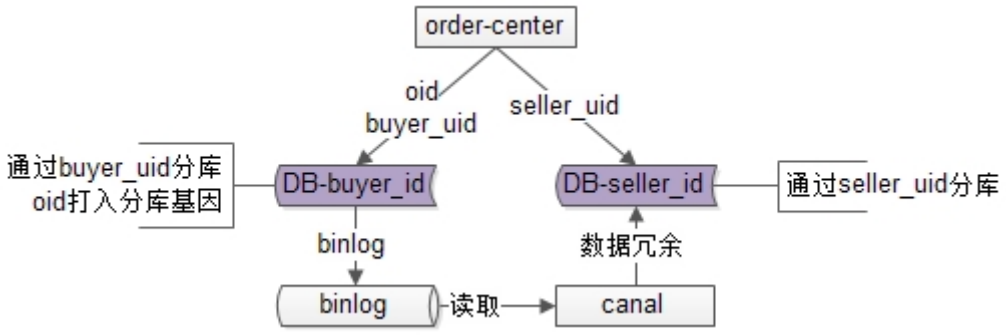
- 使用buyer_uid%16分库，决定这行数据要插入到哪个库中。
- 分库基因是buyer_uid的最后4个bit，即1010。
- 在生成订单标识oid时，先使用一种分布式ID生成算法生成前60bit（上图中绿色部分）。
- 将分库基因加入到oid的最后4个bit（上图中粉色部分），拼装成最终64bit的订单oid（上图中蓝色部分）。

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

- 通过buyer_uid%16能够定位到库。
- 通过oid%16也能定位到库。

五、假设没有oid

订单中心，假设没有oid上的查询需求，而只有buyer_uid和seller_uid上的查询需求，就蜕化为一个“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”。



如上图所示：

1. 当有订单生成时，通过buyer_uid分库，oid中融入分库基因，写入DB-buyer库。
2. 通过线下异步的方式，通过binlog+canal，将数据冗余到DB-seller库中。
3. 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。

数据冗余的方法有很多种：

1. 服务同步双写。
2. 服务异步双写。
3. 线下异步双写（上图所示，是线下异步双写）。

不管哪种方案，因为两步操作不能保证原子性，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，而是尽早的发现不一致，并修复不一致。

最终一致性，是高吞吐互联网业务一致性的常用实践。保证数据最终一致性的方案有三种：

1. 冗余数据全量定时扫描。
2. 冗余数据增量日志扫描。
3. 冗余数据线上消息实时检测。

这些方案细节在“多对多”业务水平拆分的文章里详细展开分析过，便不再赘述。

六、oid/buyer_uid/seller_uid同时存在

通过上述分析：

- 如果没有 `seller_uid` ，“多key”业务会蜕化为“1对多”业务，此时应该使用“基因法”分库：使用 `buyer_uid` 分库，在oid中加入分库基因
- 如果没有oid，“多key”业务会蜕化为“多对多”业务，此时应该使用“数据冗余法”分库：使用 `buyer_uid` 和 `seller_uid` 来分别分库，冗余数据，满足不同属性上的查询需求
- 如果oid/buyer_uid/seller_uid同时存在，可以使用上述两种方案的综合方案，来解决“多key”业务的数据库水平切分难题。

七、总结

复杂难题的解决，都是一个化繁为简，逐步击破的过程。

对于像订单中心一样复杂的“多key”类业务，在数据量较大，需要对数据库进行水平切分时，对于后台需求，采用**“前台与后台分离”的架构设计方法**：

- 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。
- 采用前台与后台数据冗余的设计方式，分别满足两侧的需求。
- 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。

对于前台需求，**化繁为简的设计思路**，将“多key”类业务，分解为“1对多”类业务和“多对多”类业务分别解决：

- 使用**“基因法”，解决“1对多”分库需求**：使用 buyer_uid 分库，在oid中加入分库基因，同时满足oid和 buyer_uid 上的查询需求
- 使用**“数据冗余法”，解决“多对多”分库需求**：使用 buyer_uid 和 seller_uid 来分别分库，冗余数据，满足 buyer_uid 和 seller_uid 上的查询需求
- 如果oid/buyer_uid/seller_uid同时存在，可以使用上述**两种方案的综合方案，来解决“多key”业务的数据库水平切分难题。**

数据冗余会带来一致性问题，高吞吐互联网业务，要想完全保证事务一致性很难，常见的实践是最终一致性。

任何脱离业务的架构设计都是耍流氓，共勉。

水平切分是一个很有意思的话题，感谢坚持半年订阅的小伙伴们，下个月最后一章，为答谢大伙的支持，可免费订阅（gitchat不允许免费，设定为1元），欢迎大家订阅。