某电商平台分库分表的背景

下面的例子和数据，来自O2O电商平台，**每日优鲜。**

一家电商公司，随着业务增长每天的订单量很快从30万单增长到了100万单，订单总量也突破了一亿。当时用的Mysql数据库。根据监控，我们的每秒最高订单量已经达到了2000笔（不包括秒杀，秒杀TPS已经上万了。

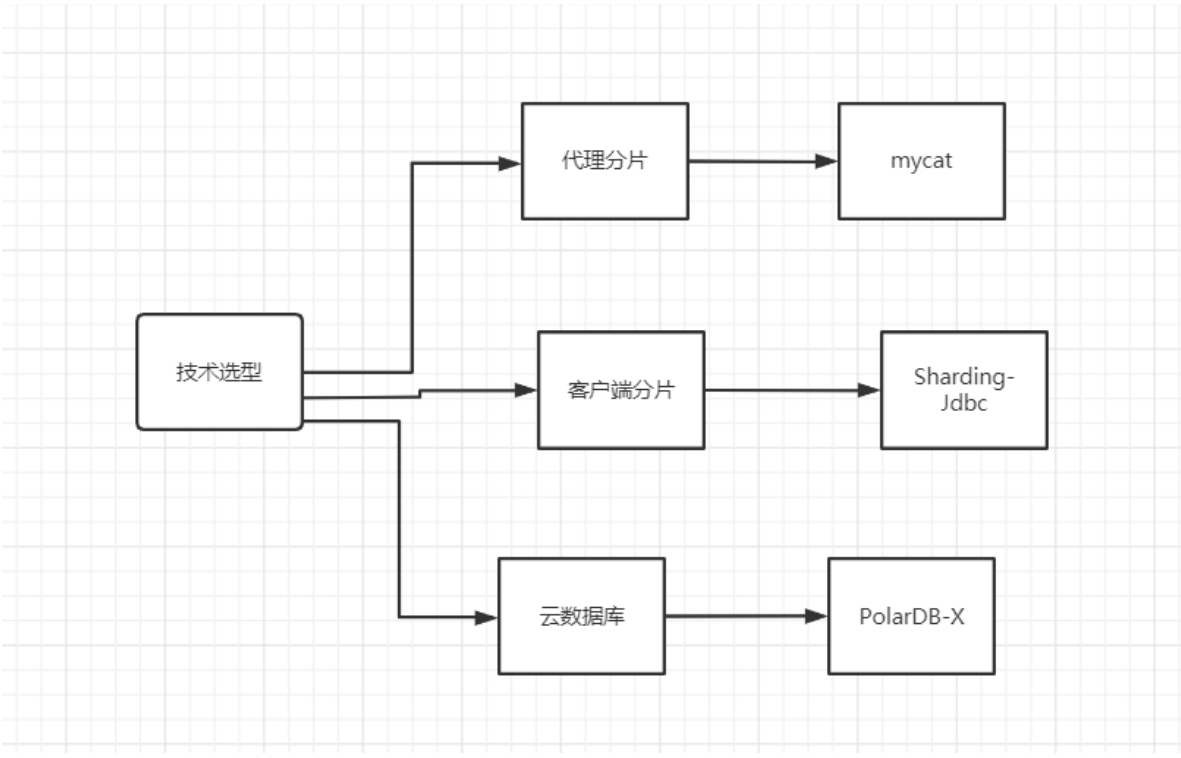
重构？说这么高大上，不就是分库分表吗？的确，就是**分库分表。**

不过除了分库分表，**还包括管理端的解决方案**，**比如**运营，客服和商务需要从**多维度查询订单数据**

分库分表后，怎么满足大家的需求？

分库分表后，上线方案和数据不停机迁移方案都需要慎重考虑。为了保证系统稳定，还需要考虑相应的降级方案。

技术选型



稳妥起见，该电商平台选用了第二种方案，使用更轻量级的Sharding-Jdbc。

分库分表架构规划：

目标： 我们希望经过本次重构，系统能支撑两年，两年内不再大改。

业务方预期： 两年内日单量达到1000万。相当于两年后日订单量要翻10倍。

悲观的预估

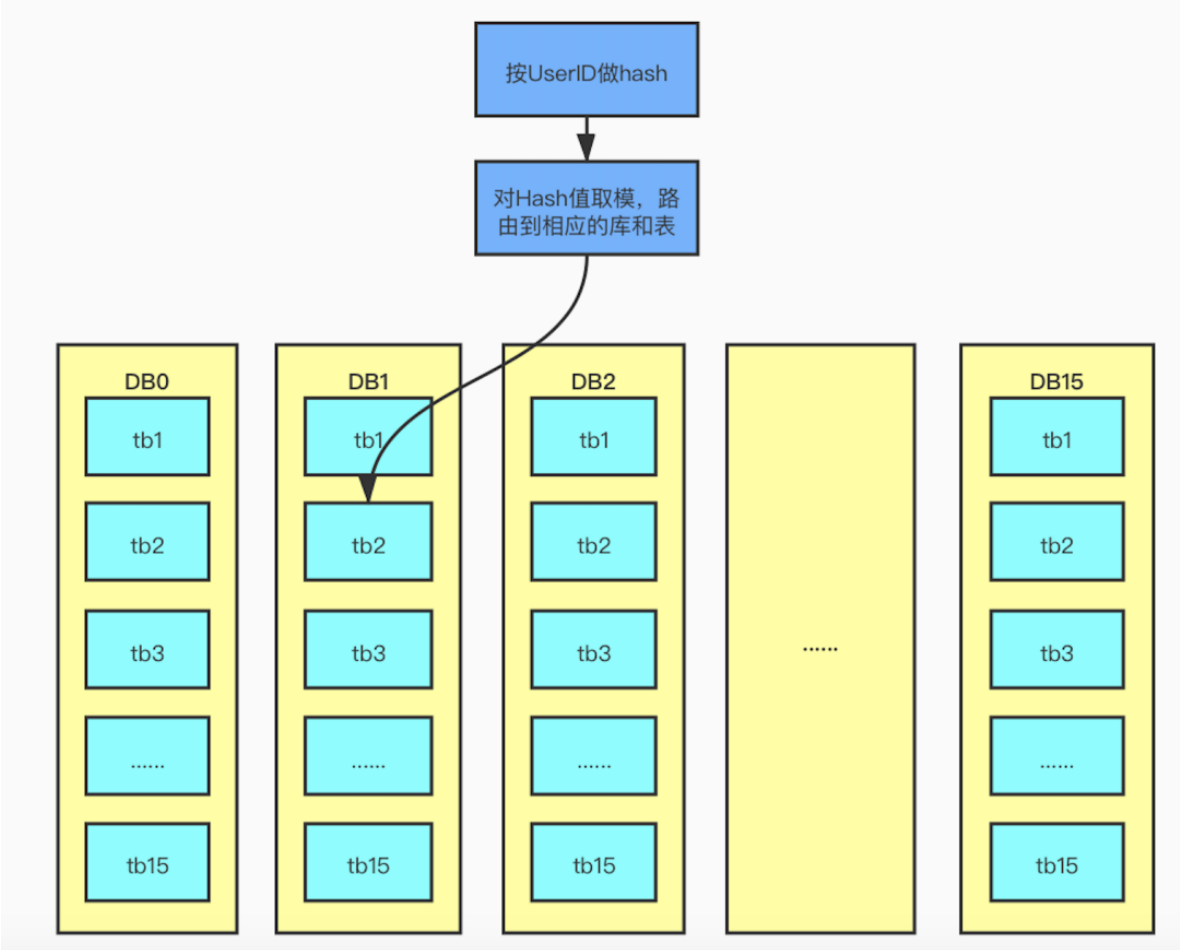
根据上面的数据，我们分成了16个数据库,每个库分了16张表，按user_id做hash。

即便按照每天1000万的订单量规划，两年总单量是73亿，每个库的数据量平均是4.56亿（4.56亿=73亿/16）,，每张表的数据量平均是2850万（2850万=4.56亿/16）,虽然有点超出了1000W的建议值，但是这是按照两年之后理想的值做的预估。实际没有那么多。

乐观的预估

即便按照每天100万的订单量规划，两年总单量是7.3亿，每个库的数据量平均是0.456亿（0.456亿=7.3亿/16），每张表的数据量平均是285万（285万=0.456亿/16）。

分库分表主要是为了APP 用户端下单和查询使用，按user_id的查询频率最高，其次是order_id。所以我们选择user_id做为sharding column，按user_id做hash，将相同用户的订单数据存储到同一个数据库的同一张表中。这样用户在网页或者App上查询订单时只需要路由到一张表就可以获取用户的所有订单了，这样就保证了查询性能。



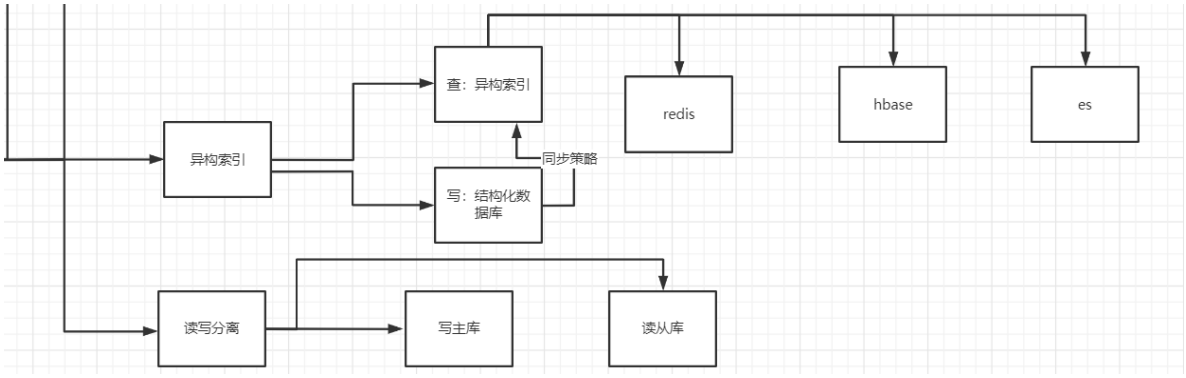
对于提升查询性能?

有读者可能会问，查询直接查数据库，会不会有性能问题？是的。

方案主要有：

1 异构索引

2 读写分离



异构索引

查询的时候，走**异构索引。**

在上层加了Redis，Redis做了分片集群，用于存储活跃用户最近50条订单。

这样一来，只有少部分在Redis查不到订单的用户请求才会到数据库查询订单，这样就减小了数据库查询压力，

读写分离

而且每个分库还有两个从库，查询操作只走从库，进一步分摊了每个分库的压力。

管理端技术方案

分库分表后，不同用户的订单数据散落在不同的库和表中，如果需要根据用户ID之外的其他条件查询订单。

例如，运营同学想从后台查出某天iphone7的订单量，就需要从所有数据库的表中查出数据然后在聚合到一起。

> 这就回到一个问题，很多小伙伴问：分库分表后，怎么关联?
>
> 插一句：数据都不在一个库中，实现关联就复杂了。

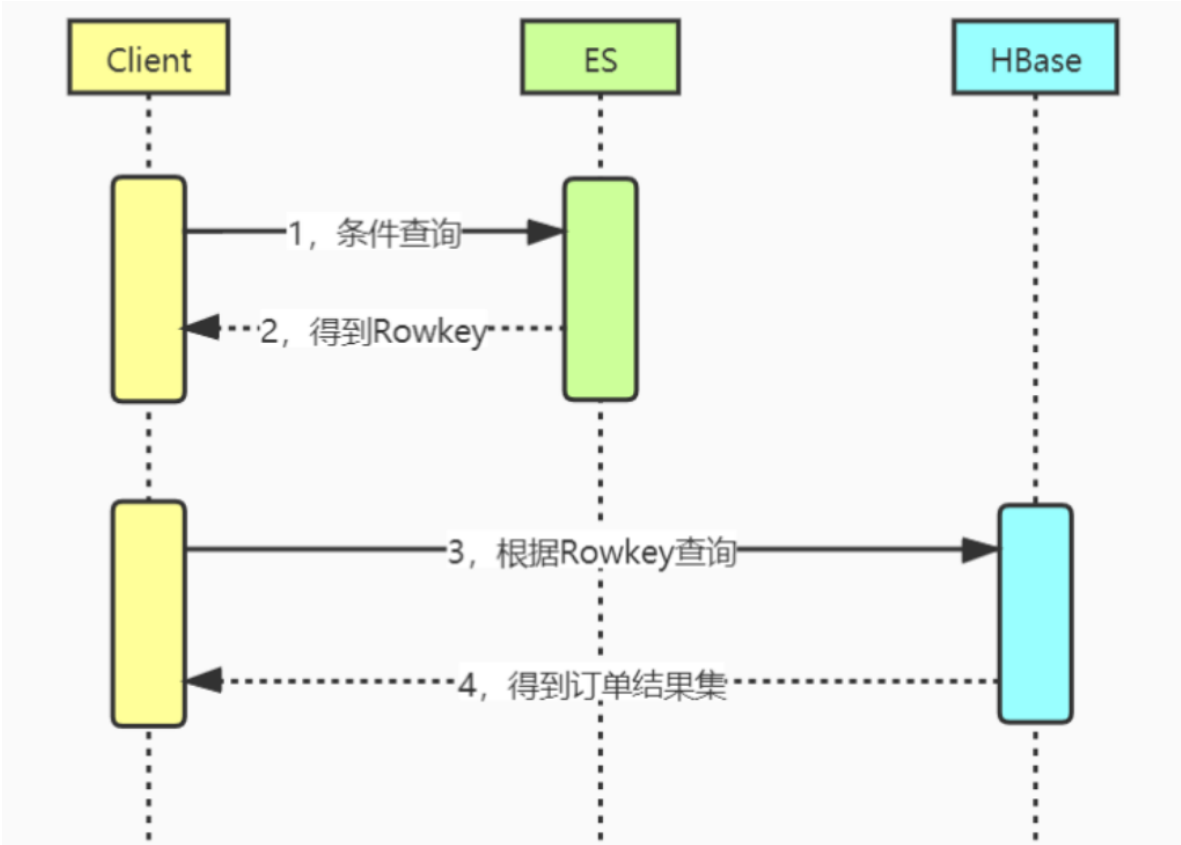
这样代码实现非常复杂，而且查询性能也会很差。所以我们需要一种更好的方案来解决这个问题。

我们采用了ES（ElasticSearch）+HBase组合的方案，将索引与数据存储隔离。

可能参与条件检索的字段都会在ES中建一份索引，例如商家，商品名称，订单日期等。所有订单数据全量保存到HBase中。我们知道HBase支持海量存储，而且根据rowkey查询速度超快。而ES的多条件检索能力非常强大。可以说，这个方案把ES和HBase的优点发挥地淋漓尽致。

ES+HBase组合的方案的查询过程：

先根据输入条件去ES相应的索引上查询符合条件的rowkey值，然后用rowkey值去HBase查询，后面这一步查询速度极快，查询时间几乎可以忽略不计。如下图：



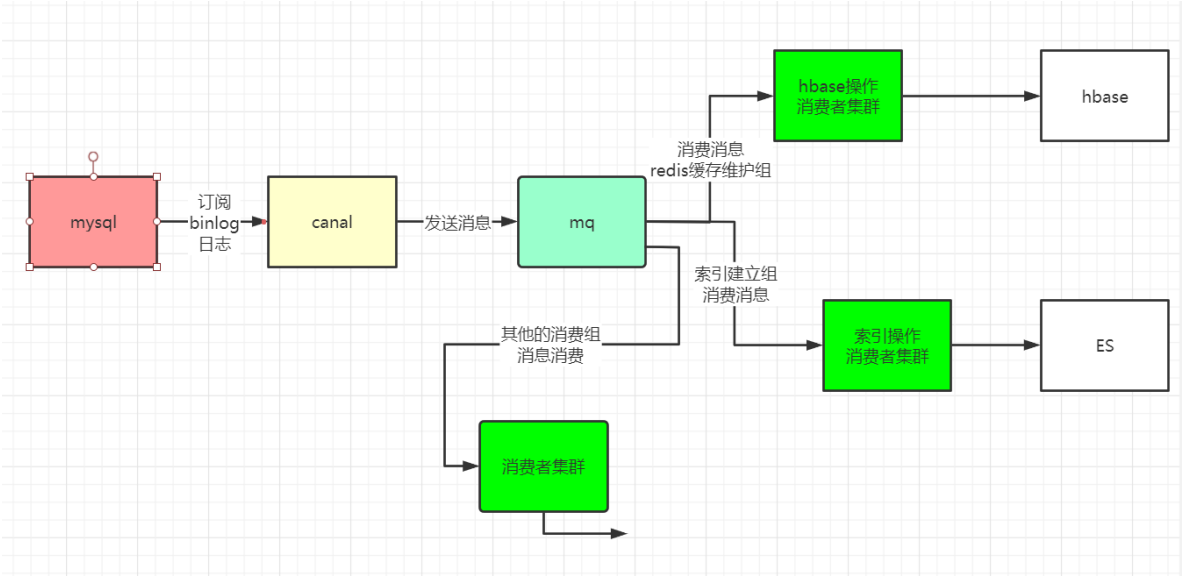
该方案，解决了管理端通过各种字段条件查询订单的业务需求，同时也解决了商家端按商家ID和其他条件查询订单的需求。

如果用户希望查询最近50条订单之前的历史订单，也同样可以用这个方案。

Mysql实时同步到Hbase和ES中

Mysql中的订单数据需要实时同步到Hbase和ES中。 同步方案是什么？

我们利用Canal实时获取Mysql库表中的增量订单数据，然后把订单数据推到消息队列RocketMQ中，消费端获取消息后把数据写到Hbase，并在ES更新索引。



上面是Canal的原理图，

- Canal模拟mysql slave的交互协议，把自己伪装成mysql的从库
- 向mysql master发送dump协议
- mysql master收到dump协议，发送binary log给slave（Canal)
- Canal解析binary log字节流对象，根据应用场景对binary log字节流做相应的处理

为了保证数据一致性，不丢失数据。我们使用了RocketMQ的事务型消息，保证消息一定能成功发送。另外，另外，在Hbase和ES都操作成功后才做ack操作，保证消息最终被消费。

不停机数据迁移

在互联网行业，很多系统的访问量很高，即便在凌晨两三点也有一定的访问量。由于数据迁移导致服务暂停，是很难被业务方接受的！下面就聊一下在用户无感知的前提下，我们的不停机数据迁移方案！

数据迁移过程我们要注意哪些关键点呢?

第一，保证迁移后数据准确不丢失，即每条记录准确而且不丢失记录；

第二，不影响用户体验，尤其是访问量高的C端业务需要不停机平滑迁移；

第三，保证迁移后的系统性能和稳定性。

常用的数据迁移方案主要包括：

方案一，挂从库，

方案二，双写

方案三，利用数据同步工具。

下面分别做一下介绍。

挂从库

在主库上建一个从库。从库数据同步完成后，将从库升级成主库（新库），再将流量切到新库。

这种方式适合表结构不变，而且空闲时间段流量很低，允许停机迁移的场景。

一般发生在平台迁移的场景，如从机房迁移到云平台，从一个云平台迁移到另一个云平台。

大部分中小型互联网系统，空闲时段访问量很低。在空闲时段，几分钟的停机时间，对用户影响很小，业务方是可以接受的。所以我们可以采用停机迁移的方案。步骤如下：

1，新建从库（新数据库），数据开始从主库向从库同步。

2，数据同步完成后，找一个空闲时间段。为了保证主从数据库数据一致，需要先停掉服务，然后再把从库升级为主库。如果访问数据库用的是域名，直接解析域名到新数据库（从库升级成的主库），如果访问数据库用的是IP，将IP改成新数据库IP。

3，最后启动服务，整个迁移过程完成。

这种迁移方案的优势:

> 迁移成本低，迁移周期短。

缺点是:

> 切换数据库过程需要停止服务。

我们的并发量比较高，而且又做了分库分表，表结构也变了，所以不能采取这种方案！

双写

老库和新库同时写入，然后将老数据批量迁移到新库，最后流量切换到新库并关闭老库读写。

这种方式**适合数据结构发生变化**，**不允许停机迁移**的场景。

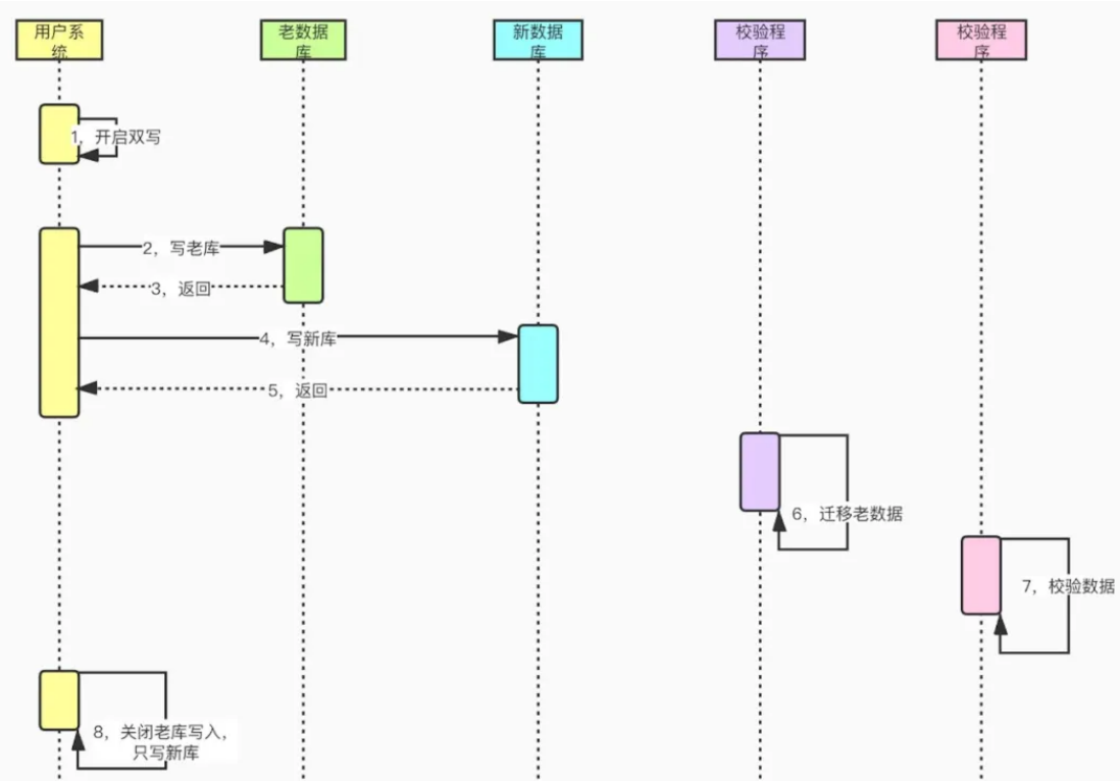
一般发生在**系统重构时**，表结构发生变化，如表结构改变或者**分库分表**等场景。

有些大型互联网系统，平常并发量很高，即便是空闲时段也有相当的访问量。几分钟的停机时间，对用户也会有明显的影响，甚至导致一定的用户流失，这对业务方来说是无法接受的。所以我们需要考虑一种用户无感知的不停机迁移方案。

聊一下我们的具体迁移方案，步骤如下：

1. 代码准备。在服务层对订单表进行增删改的地方，要同时操作新库（分库分表后的数据库表）和老库，需要修改相应的代码（同时写新库和老库）。准备迁移程序脚本，用于做老数据迁移。准备校验程序脚本，用于校验新库和老库的数据是否一致。
2. 开启双写，老库和新库同时写入。注意：任何对数据库的增删改都要双写；对于更新操作，如果新库没有相关记录，需要先从老库查出记录，将更新后的记录写入新库；为了保证写入性能，老库写完后，可以采用消息队列异步写入新库。
3. 利用脚本程序，将某一时间戳之前的老数据迁移到新库。注意：1，时间戳一定要选择开启双写后的时间点，比如开启双写后10分钟的时间点，避免部分老数据被漏掉；2，迁移过程遇到记录冲突直接忽略，因为第2步的更新操作，已经把记录拉到了新库；3，迁移过程一定要记录日志，尤其是错误日志，如果有双写失败的情况，我们可以通过日志恢复数据，以此来保证新老库的数据一致。
4. 第3步完成后，我们还需要通过脚本程序检验数据，看新库数据是否准确以及有没有漏掉的数据
5. 数据校验没问题后，开启双读，起初给新库放少部分流量，新库和老库同时读取。由于延时问题，新库和老库可能会有少量数据记录不一致的情况，所以新库读不到时需要再读一遍老库。然后再逐步将读流量切到新库，相当于灰度上线的过程。遇到问题可以及时把流量切回老库
6. 读流量全部切到新库后，关闭老库写入（可以在代码里加上热配置开关），只写新库

7. 迁移完成，后续可以去掉双写双读相关无用代码。



利用数据同步工具

我们可以看到上面双写的方案比较麻烦，很多数据库写入的地方都需要修改代码。有没有更好的方案呢?

> 我们还可以利用Canal，DataBus等工具做数据同步。以阿里开源的Canal为例。

利用同步工具，就不需要开启双写了，服务层也不需要编写双写的代码，直接用Canal做增量数据同步即可。相应的步骤就变成了：

1. 代码准备。

   准备Canal代码，解析binary log字节流对象，并把解析好的订单数据写入新库。

   准备迁移程序脚本，用于做老数据迁移。

   准备校验程序脚本，用于校验新库和老库的数据是否一致。

2. 运行Canal代码，开始增量数据（线上产生的新数据）从老库到新库的同步。

3. 利用脚本程序，将某一时间戳之前的老数据迁移到新库。

   注意：

   1，时间戳一定要选择开始运行Canal程序后的时间点（比如运行Canal代码后10分钟的时间点），避免部分老数据被漏掉；

   3，迁移过程一定要记录日志，尤其是错误日志，如果有些记录写入失败，我们可以通过日志恢复数据，以此来保证新老库的数据一致。

4. 第3步完成后，我们还需要通过脚本程序检验数据，看新库数据是否准确以及有没有漏掉的数据

5. 数据校验没问题后，开启双读，起初给新库放少部分流量，新库和老库同时读取。由于延时问题，新库和老库可能会有少量数据记录不一致的情况，所以新库读不到时需要再读一遍老库。逐步将读流量切到新库，相当于灰度上线的过程。遇到问题可以及时把流量切回老库

6. 读流量全部切到新库后，将写入流量切到新库（可以在代码里加上热配置开关。注：由于切换过程Canal程序还在运行，仍然能够获取老库的数据变化并同步到新库，所以切换过程不会导致部分老库数据无法同步新库的情况）

7. 关闭Canal程序

8. 迁移完成。

扩容缩容方案

需要对数据重新hash取模，再将原来多个库表的数据写入扩容后的库表中。整体扩容方案和上面的不停机迁移方案基本一致。

采用双写或者Canal等数据同步方案都可以。

异步降级方案

在大促期间订单服务压力过大时，可以将同步调用改为异步消息队列方式，来减小订单服务压力并提高吞吐量。

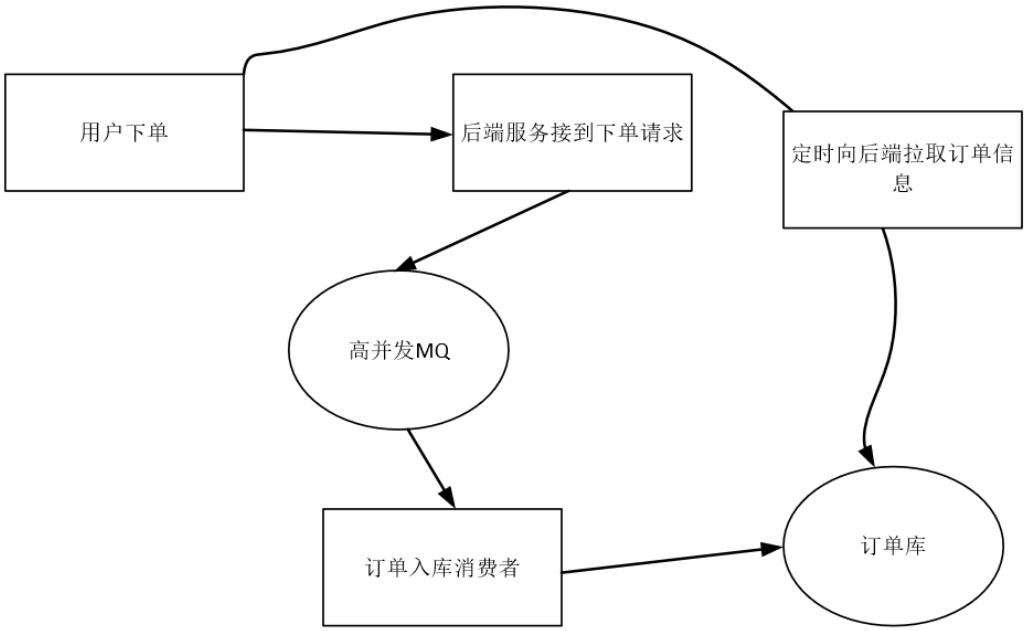
大促时某些时间点瞬间生成订单量很高。我们采取异步批量写数据库的方式，来减少数据库访问频次，进而降低数据库的写入压力。

详细步骤：后端服务接到下单请求，直接放进消息队列，订单服务取出消息后，先将订单信息写入Redis，每隔100ms或者积攒10条订单，批量写入数据库一次。

前端页面下单后定时向后端拉取订单信息，获取到订单信息后跳转到支付页面。

用这种异步批量写入数据库的方式大幅减少了数据库写入频次，从而明显降低了订单数据库写入压力。

流程如下图：



不过，因为订单是异步写入数据库的，就会存在数据库订单和相应库存数据暂时不一致的情况，以及用户下单后不能及时查到订单的情况。

因为毕竟是降级方案，可以适当降低用户体验，我们保证数据最终一致即可。

根据系统压力情况，可以在大促开始时开启异步批量写的降级开关，大促结束后再关闭降级开关。