# Shardingsphere jdbc：

## 背景

我们平时做项目开发。一开始，通常都先用一张数据表，而一般来说数据表写到两千万条数据之后，底层 B+ 树的层级结构就可能会变高，不同层级的数据页一般都放在磁盘里不同的地方，换言之，磁盘 IO 就会增多，带来的便是查询性能变差。

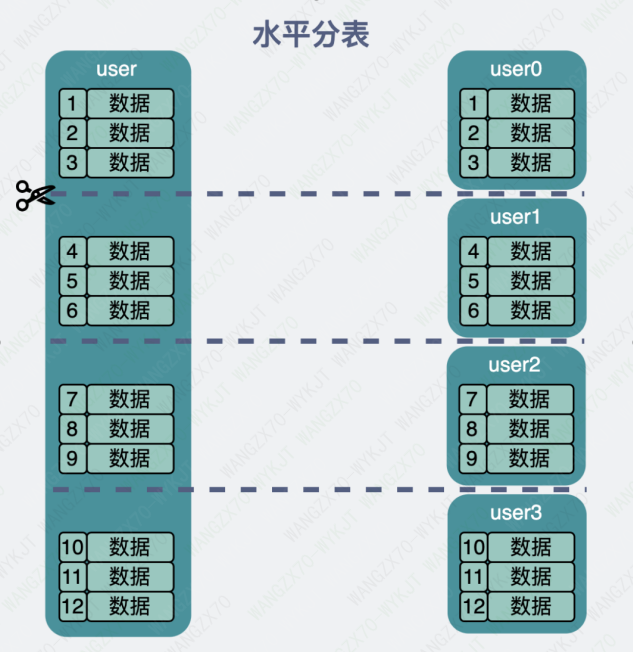
于是，当我们单表需要管理的数据变得越来越多，就不得不考虑数据库分表。而这里的分表，分为水平分表和垂直分表。

垂直分表的原理比较简单，一般就是把某几列拆成一个新表，这样单行数据就会变小， B+ 树里的单个数据页（固定 16KB）内能放入的行数就会变多，从而使单表能放入更多的数据。

垂直分表没有太多可以说的点。下面，我们重点说说最常见的水平分表。

水平分表有好几种做法，但不管是哪种，本质上都是将原来的 user 表，变成 user\_0, user1, user2 .... userN 这样的 N 多张小表。

从读写一张 user 大表，变成读写 user\_1 … userN 这样的 N 张小表。



每一张小表里，只保存一部分数据，但具体保存多少，这个自己定，一般就定 500 万~ 两千万。

1. 如果想要支持更多维度的查询，可以监听 MySQL 的 binlog，将数据写入到 es，提供近实时的查询能力；

2.当然，用 TiDB 替换 MySQL 也是个思路。

3.不要做过早的优化，没事别上来就分 100 个表，很多时候真用不上。

## 2.唯⼀主键

⼀般我们数据库的主键都是⾃增的，那么分表之后主键冲突的问题就是⼀个⽆法避免的问题，最简单的

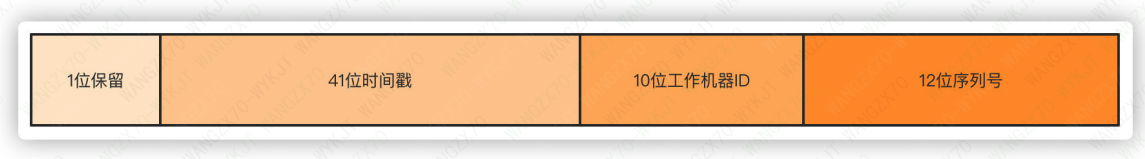
办法就是以⼀个唯⼀的业务字段作为唯⼀的主键，⽐如订单表的订单号肯定是全局唯⼀的。

常⻅的分布式⽣成唯⼀ID的⽅式很多，最常⻅的雪花算法Snowflake、滴滴Tinyid、美团Leaf。以雪花算

法举例来说，⼀毫秒可以⽣成4194304多个ID。

第⼀位不使⽤，默认都是0，41位时间戳精确到毫秒，可以容纳69年的时间，10位⼯作机器ID⾼5位是数

据中⼼ID，低5位是节点ID，12位序列号每个节点每毫秒累加，累计可以达到2^12 4096个ID



## 3.查询问题

### **C端查询**

⾸先说带shardingkey的查询，⽐如就通过订单号查询，不管你分⻚还是怎么样都是能直接定位到具体的

表来查询的，显然查询是不会有什么问题的。

如果不是shardingkey的话，上⾯举例说的以订单号作为shardingkey的话，像APP、⼩程序这种⼀般都

是通过⽤户ID查询，那这时候我们通过订单号做的sharding怎么办？很多公司订单表直接⽤⽤户ID做

shardingkey，那么很简单，直接查就完了。那么订单号怎么办，⼀个很简单的办法就是在订单号上带上

⽤户ID的属性。举个很简单的例⼦，原本41位的时间戳你觉得⽤不完，⽤户ID是10位的，订单号的⽣成

规则带上⽤户ID，落具体表的时候根据订单号中10位⽤户ID hash取模，这样⽆论根据订单号还是⽤户ID

查询效果都是⼀样的。

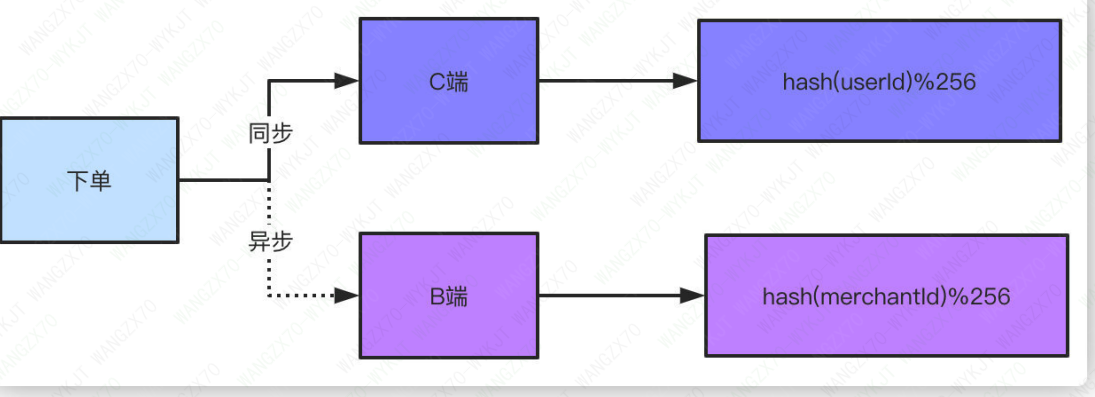


现实中真正的流量⼤头都是来⾃于⽤户端C端，所以本质上解决了⽤户端的问题，这个问题就解了⼤半，

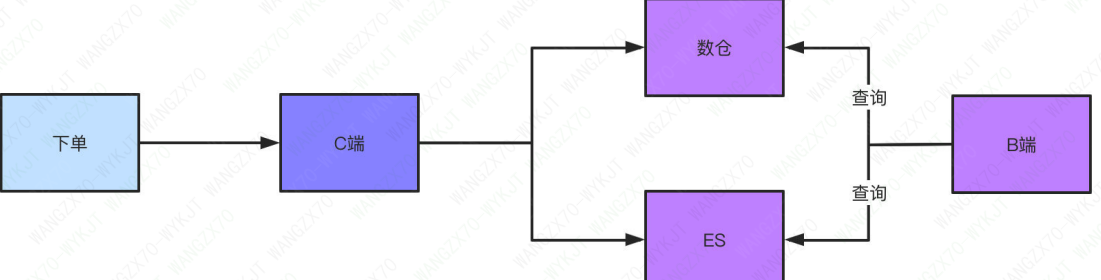
剩下来⾃商户卖家端B端、后台⽀持运营业务的查询流量并不会很⼤，这个问题就好解。

### **B端查询**

1.双写， 商家id作为shardingkey



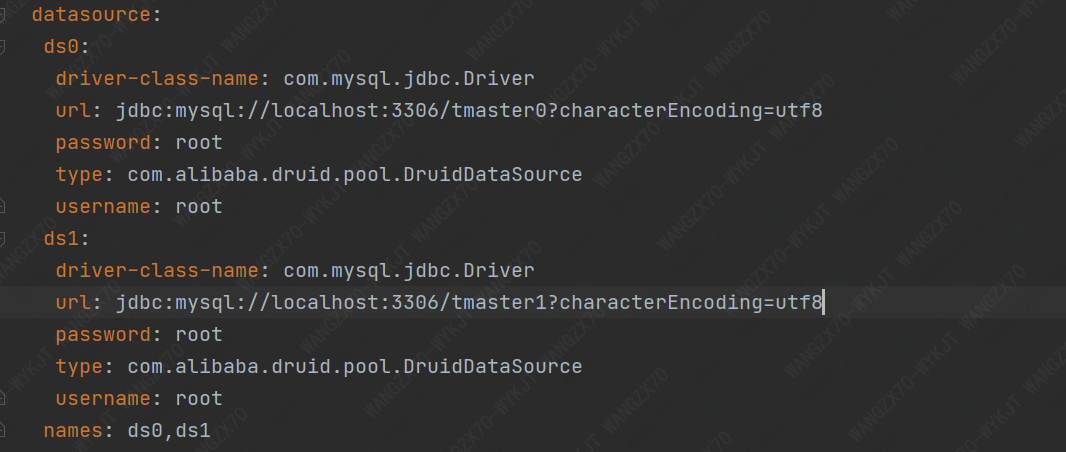
2.通过nosql或者大数据查询



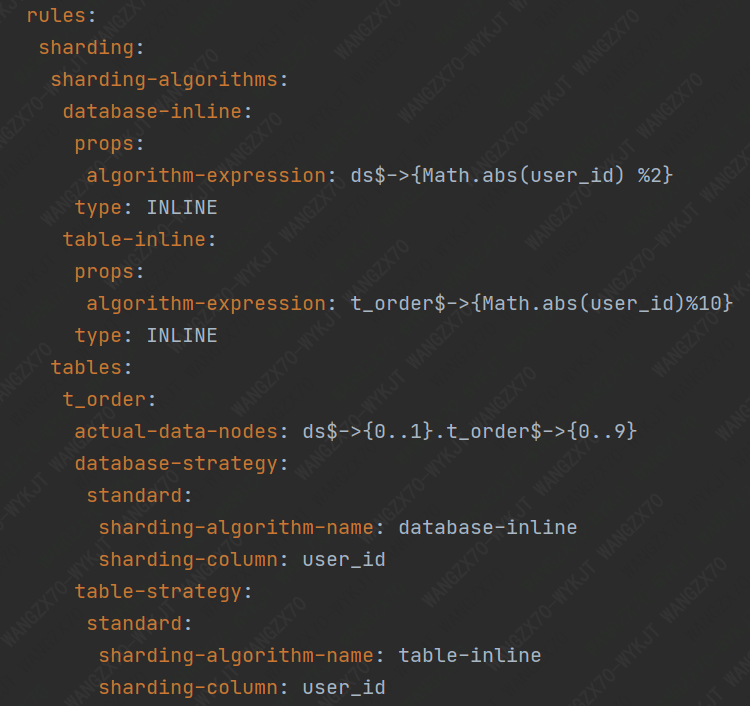
## 4.配置规则(two21)

springboot starter方式引入5.2.1版本，官方提供的maven包

配置两个数据库

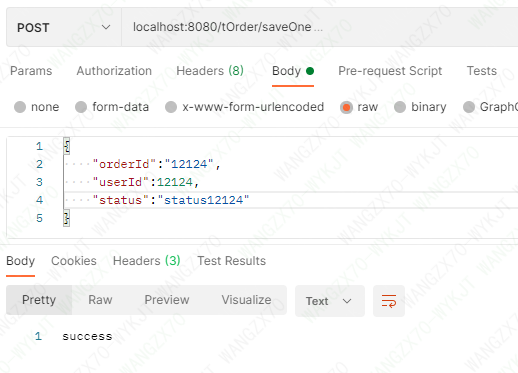


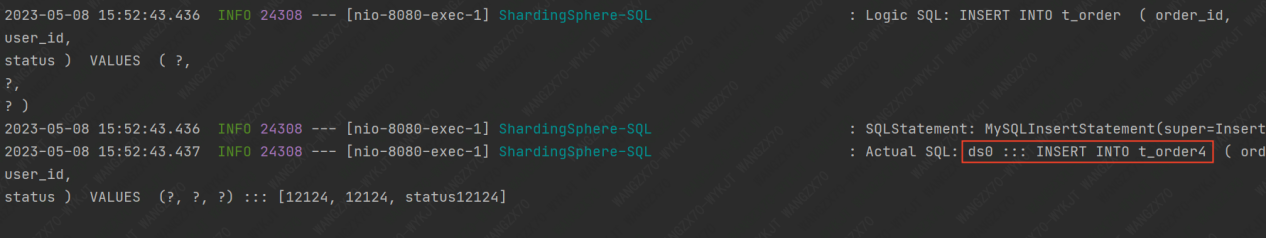
按照user\_id取模2进行分库,按照user\_id取模10进行分表



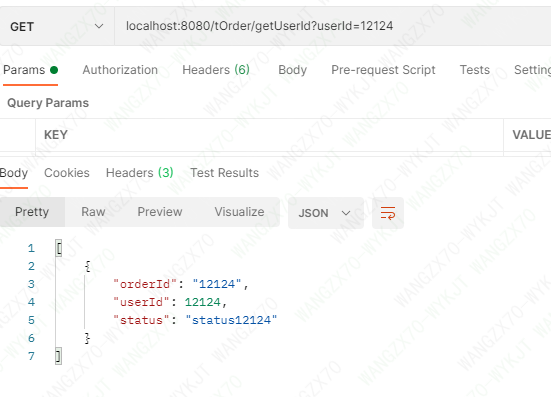
## 5.基本的增删改查

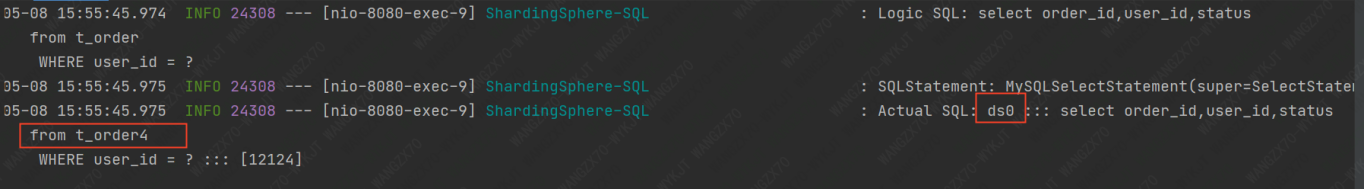
新增:走分片键,逻辑的sql语句转换成对应的实际sql语句



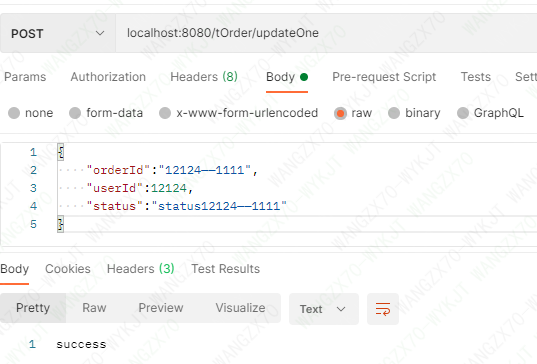


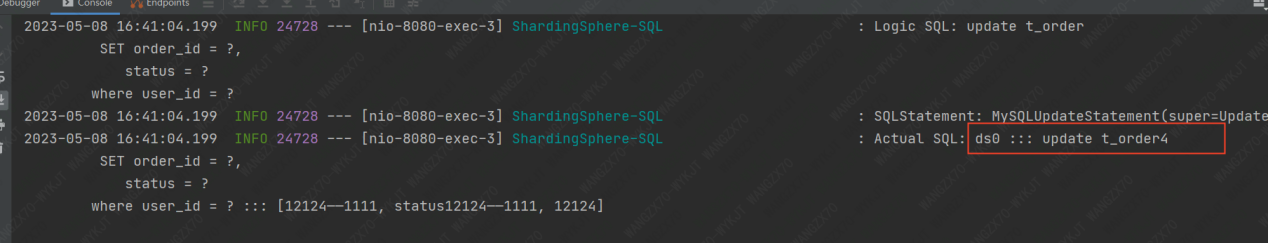
查询sql:走分片键



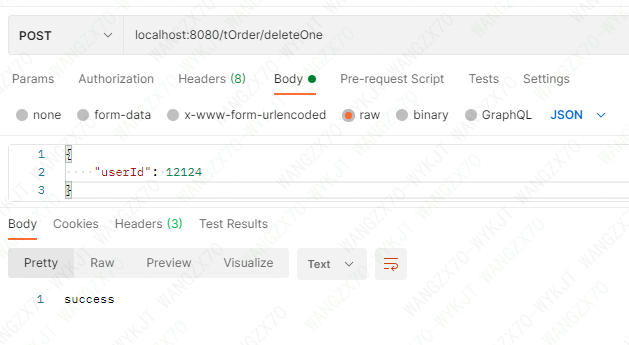


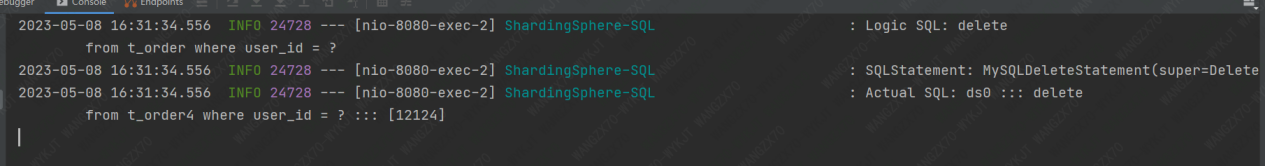
更新sql:走分片键



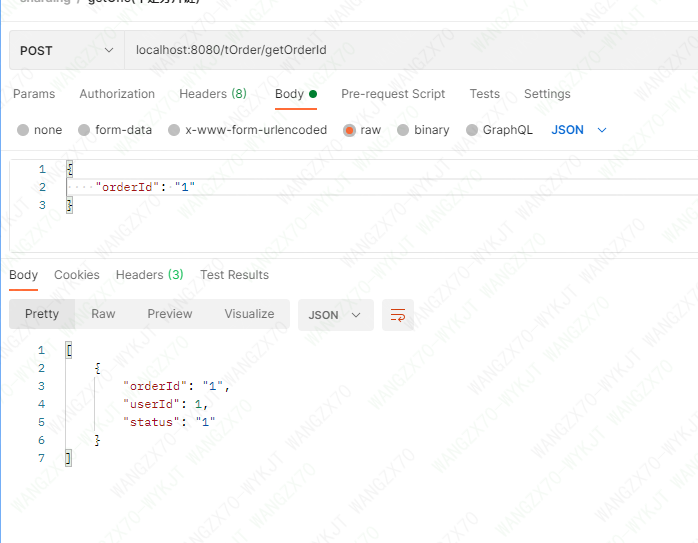


删除sql:走分片键

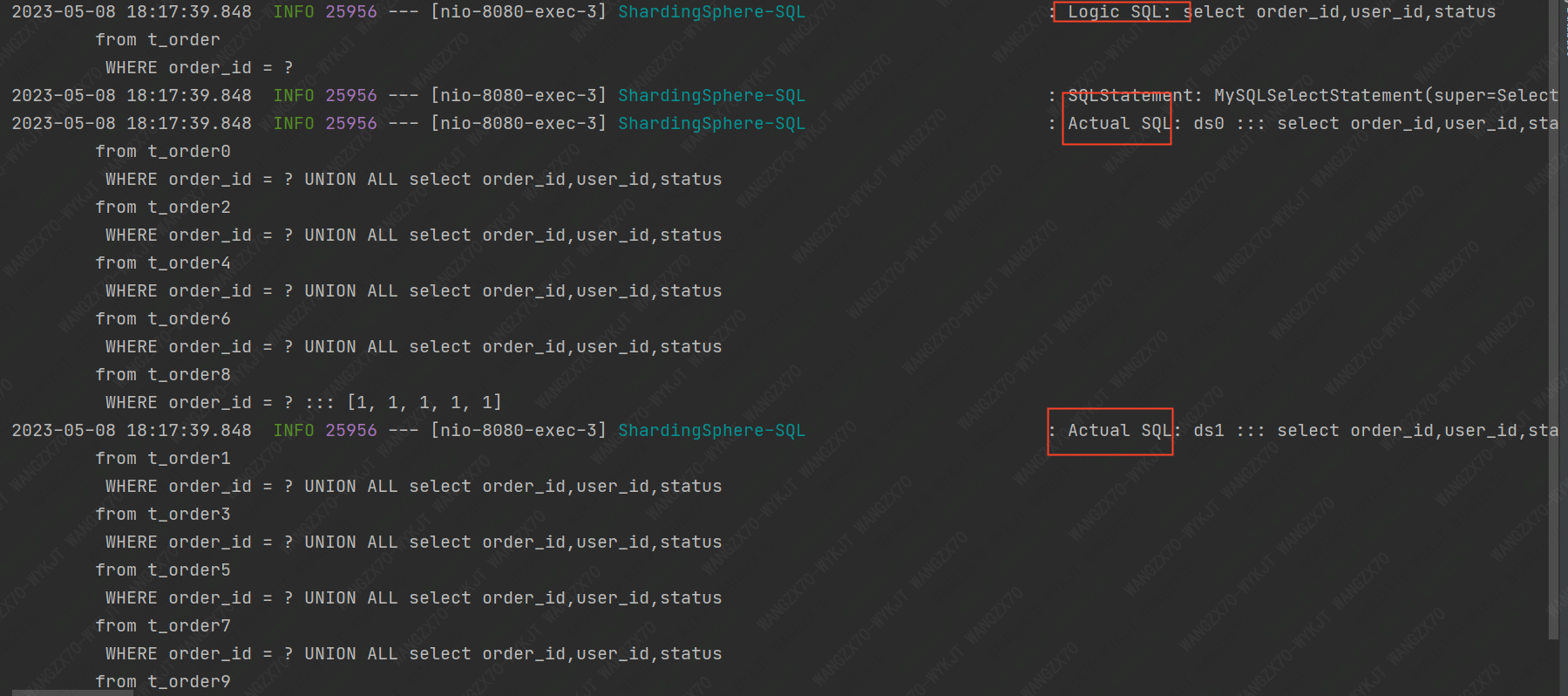




没走分片键的查询



全库全表扫描



## 6.分页查询 todo