

分库分表的深入实战剖析

内容大纲

- 分库分表概念
- 电商系统下订单性能瓶颈问题
- 分库分表原则剖析&产生的问题剖析
- 电商系统亿级订单数据分库分表实战指导

一、分库分表概念

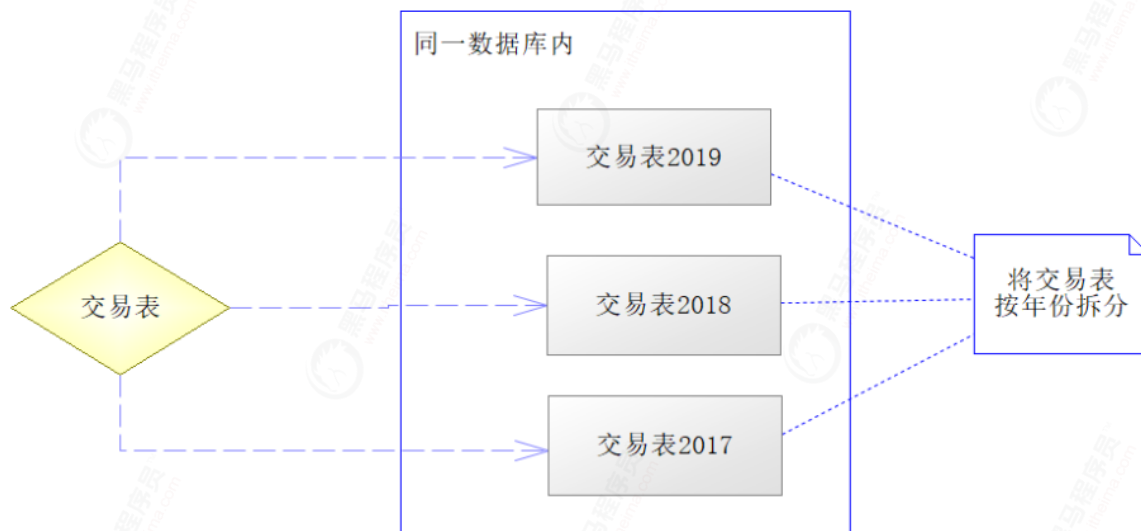
概念：

在数据爆炸的年代，单表数据达到千万级别，甚至过亿的量，都是很常见的情景。这时候再对数据库进行操作就是非常吃力的事情了，select个半天都出不来数据，这时候业务已经难以维系。不得已，分库分表提上日程，我们的目的很简单，减小数据库的压力，缩短表的操作时间。



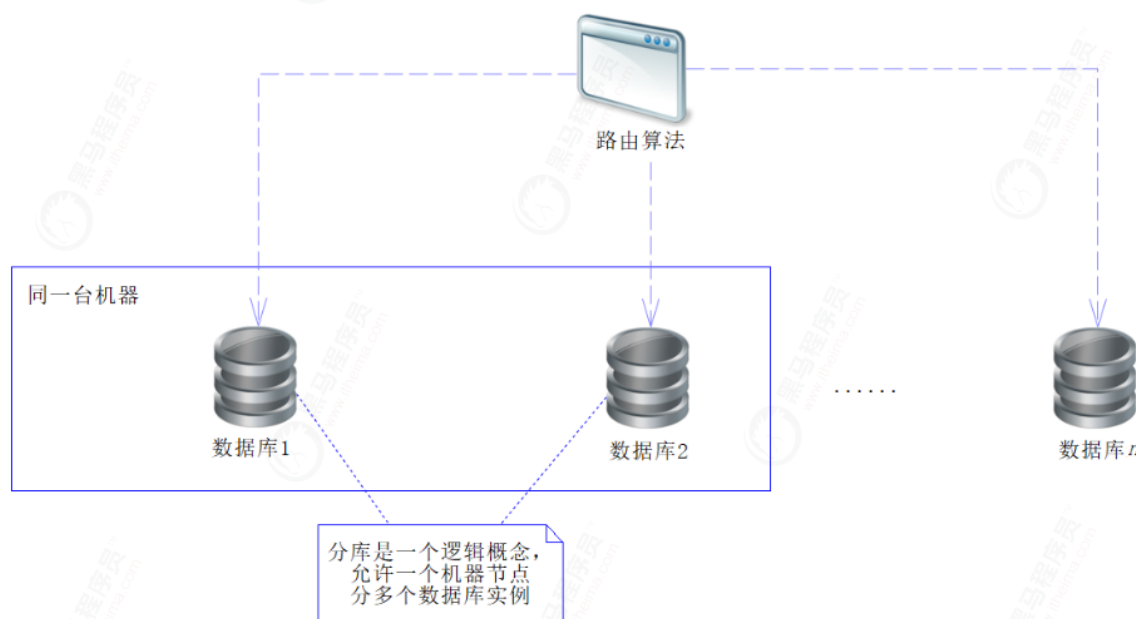
1、分表：

分表是指在一个或者多个数据库实例内，将一张表拆分为多张表存储。一般来说，分表是因为该表需要存储很庞大的记录数，如果将其堆积到一起，就会导致数据量过于庞大（一般 MySQL 的表是 5000 万条记录左右）引发性能瓶颈。一般分表会按照某种算法进行拆分，如交易记录，可能按年份拆分：



2、分库:

分库是指将一套数据库的设计结构，部署到多个数据库实例的节点中去，在应用的时候，按照一定的方法通过多个数据库实例节点访问数据。请注意，这里的数据库实例节点是一个逻辑概念，不是一个物理概念，什么意思呢？简单地说，一个机器节点可以部署多个数据库实例节点，也可以一个机器节点只部署一个数据库实例节点，所以机器节点不一定等于数据库节点。而机器节点是物理概念，是看得到的真实的机器；数据库节点是逻辑概念，是看不到的东西。为了更好地说明分库的概念,如下图



Why分库分表:

1、前提:

移动互联网时代，海量的用户每天产生海量的数量，比如：

- 用户表
- 订单表
- 交易流水表

以支付宝用户为例，8亿；微信用户更是多大10亿。订单表更夸张，比如美团外卖，每天都是几千万的订单。淘宝的历史订单总量应该是百亿，甚至是千亿级别，这些海量数据远不是一张表能Hold住的。事实上MySQL单表可以存储10亿级数据，只是这时候性能比较差，业界公认MySQL单表容量在1KW以下是最佳状态，因为这是它的BTREE索引树高在3~5之间。

既然一张表无法搞定，那么就想办法将数据放到多个地方，目前比较普遍的方案有3个：

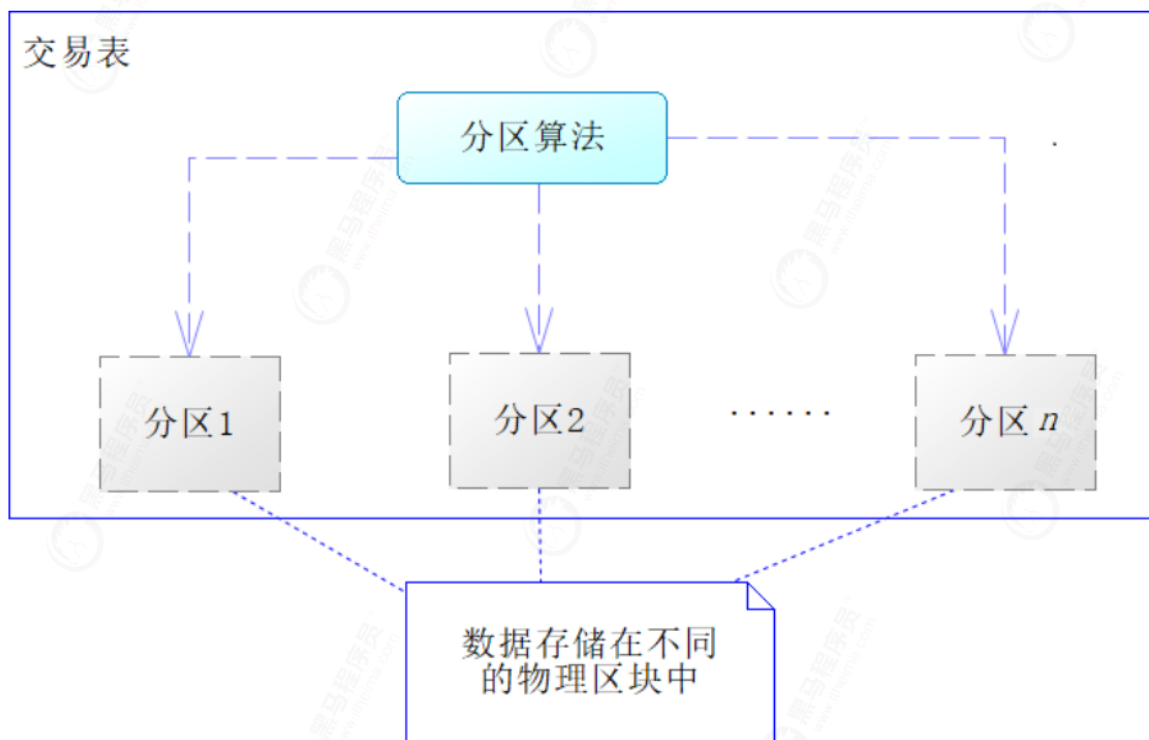
- 分区
- 分库分表
- NoSQL/NewSQL

说明：只分库，或者只分表，或者分库分表融合方案都统一认为是分库分表方案，因为分库，或者分表只是一种特殊的分库分表而已。NoSQL比较具有代表性的是MongoDB，es。NewSQL比较具有代表性的是TiDB。

2、Why Not 分区：

分区原理：

分区技术与分表技术很类似，只是分区技术属于数据库内部的技术，对于开发者来说，它逻辑上仍旧是一张表，开发时不需要改变 SQL 表名。将一张表切分为多个物理区块



它的缺点很明显：很多的资源都受到单机的限制，例如连接数，网络吞吐等！虽然每个分区可以独立存储，但是分区表的总入口还是一个MySQL示例。从而导致它的并发能力非常一般，远远达不到互联网高并发的要求！

使用分区，你的业务必须具备两个特点：

- 数据不是海量(分区数有限，存储能力有限)
- 并发能力要求不高

3、Why Not NoSQL/NewSQL：

目前绝大部分公司的核心数据都是：**以RDBMS存储为主，NoSQL/NewSQL存储为辅**！互联网公司又以MySQL为主，国企&银行等不差钱的企业以Oracle/DB2为主！NoSQL/NewSQL宣传的无论多牛逼，就现在各大公司对它的定位，都是RDBMS的补充，而不是取而代之！

4、Why 分库分表：

首先目前互联网行业处理海量数据的通用方法：**分库分表**。

虽然大家都是采用分库分表方案来处理海量核心数据，但是还没有一个一统江湖的中间件，这里列举一些有一定知名度的分库分表中间件：

- 阿里的TDDL,DRDS和cobar
- 开源社区的Sharding-jdbc (3.x已经更名为sharding-sphere)
- 民间组织的MyCAT
- 360的Atlas

备注：sharding-jdbc的作者张亮大神原来在当当，现在在京东金融。但是sharding-jdbc的版权属于开源社区，不是公司的，也不是张亮个人的！

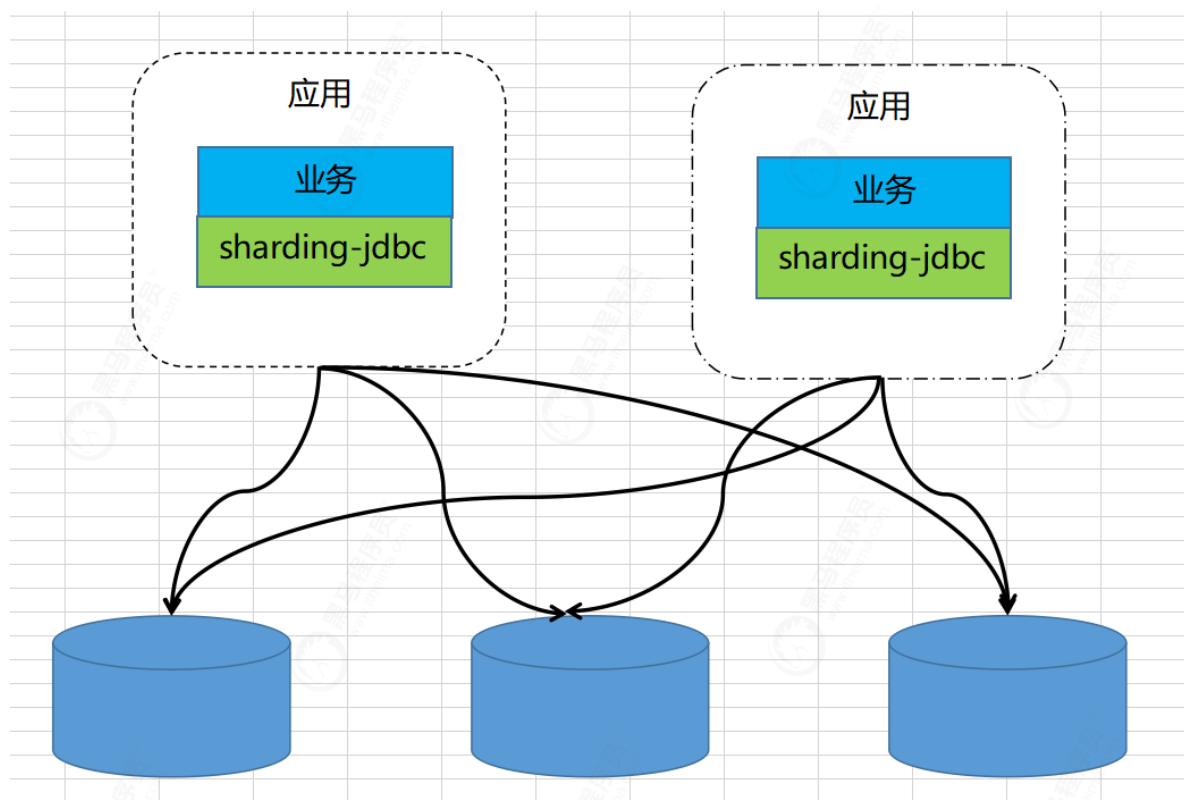
分库分表中间件归结类型

分库分表的中间件很多，但是可以归结为两大类型：

- CLIENT模式
- PROXY模式

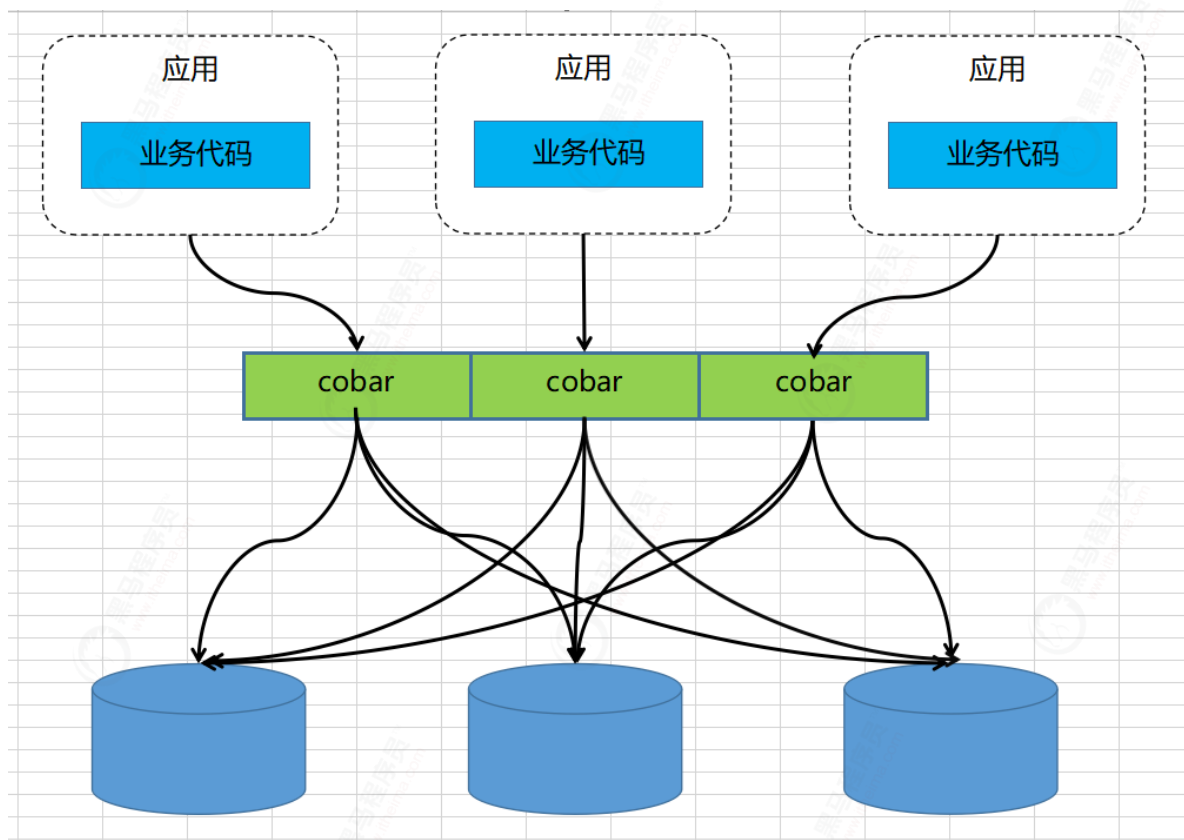
1、CLIENT模式：

代表有阿里的TDDL，开源社区的sharding-jdbc（sharding-jdbc的3.x版本即sharding-sphere已经支持了proxy模式）。架构如下：



2、PROXY模式：

代表有阿里的cobar，民间组织的MyCAT。架构如下：



3、小结

从以上两种模式可以看出sharding-jdbc作为一个组件集成在应用内，而mycat则作为一个独立的应用需要单独部署。

维度	Sharding-JDBC	MyCAT
性能	高	中
应用场景限制	java应用	无
是否支持自定义sharding路由	是	是
最大支持sharding路由维度	2	1
分布式事务	标准事务，XA强一致事务，柔性事务	支持弱xa、支持XA分布式事务(1.6.5)
限制	不支持子语句，不支持union和union all,不支持批量插入，不支持distinct聚合	详见《MYCAT权威指南》-5.6 MyCat目前存在的限制
是否开源	是	是

但是，无论是CLIENT模式，还是PROXY模式。几个核心的步骤是一样的：**SQL解析，重写，路由，执行，结果归并。**

client模式，架构简单，性能损耗比较小，运维成本低。

mycat的单机模式无法保证可靠性，一旦宕机则服务就变得不可用，你又不得不引入HAProxy来实现他的高可用集群部署方案，为了解决HAProxy的高可用问题，有需要采用keepalived来实现。

二、电商系统下订单性能瓶颈问题

1、性能瓶颈的产生

1)、原大众点评的订单单表早就已经突破200G,由于查询维度较多，即使加了两个从库，优化索引，仍然存在很多查询不理想的情况。去年大量抢购活动的开展，使数据库达到瓶颈，应用智能通过限速、异步队列等对其进行保护；业务需求层出不穷，原有的订单模型很难满足业务需求，但是基于原订单表的DDL又非常吃力，无法达到业务要求。

2)、订单表在电商项目中存储的数量是最多的，但是电商项目中使用mysql数据库较多，那么mysql的存储量最佳大概在1000w条，那么当存储量大于1000w条的时候，查询订单就会性能很差。

3)、IO瓶颈

- 磁盘读IO瓶颈，热点数据太多，数据库缓存放不下，每次查询是会产生大量的IO,降低查询速度 -> 分库操作和垂直分表操作
- 网络IO瓶颈，请求的数据太多，网络带宽不够 -> 分库操作

4)、CPU瓶颈

- SQL问题，如SQL中包含join,group by ,order by ,非索引字段条件查询等，增加CPU运算的操作 ->SQL优化，建立一个合适的索引，在业务中进行运算优化。
- 单表数据量太大，查询时扫描的行太多，SQL效率低，CPU率先出现瓶颈 -->水平分表操作

2、性能瓶颈的破除

随着以上问题越来越突出，订单数据库的切分就愈发急迫了。本次切分，我们的目标是未来几年内不需要担心订单容量的问题。

使用分库分表和SQL优化来解决这个订单性能瓶颈问题。

三、分库分表原则剖析&产生的问题剖析

1、分库分表原则

1)、能不分就不分

MySQL是关系型数据库，数据库表之间的关系从一定的角度上映射了业务逻辑。任何分库分表的行为都会在某程度上提升业务逻辑的复杂度，数据库除了承载数据的存储和访问外，协助业务更好的实现需求和逻辑也是其重要工作之一。分库分表会带来数据的合并，查询或者更新条件的分离，事务的分离等等多种后果，业务实现的复杂程度往往会翻倍或者指数级上升。所以，在分库分表之前，不要为分而分，去做其他力所能及的事情，比如升级硬件，升级网络，升级数据库库版本，读写分离，负载均衡等。所有分库分表的前提是，这些都已经尽力了。

2)、数据量太大，正常的运维不满足正常业务访问

对数据库的备份。如果单表或者单个实例太大，在做备份的时候需要大量的磁盘IO或者网络IO资源。例如1T的数据，网络传输占用50MB的时候，需要20000秒才能传输完毕，在此整个过程中的维护风险都是高于平时的。

对数据表的修改。如果某个表过大，对此表做DDL的时候，MySQL会锁住全表，这个时间可能很长，在这段时间业务不能访问此表，影响甚大。

整个表热点，数据访问和更新频繁，经常有锁等待，你又没有能力去修改源码，降低锁的粒度，那么只会把其中的数据物理拆开，用空间换时间，变相降低访问压力。

3)、表设计不合理，需要对某些字段垂直拆分

举一个例子，如果一个用户表，在最初设计表的时候是这样的：

users表

字段名称	字段类型	备注
id	bigint	用户的ID
name	varchar	用户名字
last_login_time	datetime	最近登录时间
personal_info	text	私人信息
xxxx	xxxx	其他信息字段

这种设计是很常见的，但是：

情况1：

你的业务中如果用户数从100W增长到10亿。你为了统计活跃用户，在每个人登录的时候都会记录一下他的最近登录时间。并且用户活跃的很，不断的去update这个login_time字段值，那么这个表就会压力很大。站在业务的角度上，最好的办法就是先把last_login_time字段拆分出去，我们暂且叫他user_time。这样做的好处是业务的代码只有在用到这个字段的时候修改一下就行了。如果不这么做，直接把users表水平切分了，那么所有访问users表的地方都要进行修改。

情况2：

personal_info这个字段本来没啥用，就是让用户注册的时候填一些个人爱好而已，基本不查询。一开始的时候有没有无所谓，但是后来发现两个问题：一，这个字段占用了大量的空间，因为是text类型，有很多人喜欢长篇大论地介绍自己。更糟糕的是二，不知道哪天哪个产品经理心血来潮，说允许个人信息公开吧，以方便让大家更好的相互了解。那么在所有人猎奇窥私心理的影响下，对此字段的访问大幅度增加。数据库压力瞬间扛不住了，这个时候，只好考虑对这个表进行垂直拆分。

4)、某些数据表出现了无穷增长

在目前项目中，各种的评论，消息，日志记录。这个增长不是跟人口成比例的，而是不可控的，例如微博的feed的广播，我发一条消息，会扩散给很多很多人。虽然主体可能只存一份，但不排除一些索引或者路由有这种存储需求。这个时候，增加存储，提升机器配置已经苍白无力了，水平切分是最佳实践。拆分的标准很多，按用户的，按时间的，按用途的等等方式进行拆分。

5)、安全性和可用性的考虑

这个很容易理解，鸡蛋不要放在一个篮子里，我不希望我的数据库出问题，但我希望在出问题的时候不要影响到100%的用户，这个影响的比例越少越好，那么，水平切分可以解决这个问题，把用户，库存，订单等等本来同统一的资源切分掉，每个小的数据库实例承担一小部分业务，这样整体的可用性就会提升。这对Qunar(去哪儿)这样的业务还是比较合适的，人与人之间，某些库存与库存之间，关联不太大，可以做一些这样的切分。

6)、业务耦合性考虑

这个跟上面有点类似，主要是站在业务的层面上，我们的火车票业务和烤羊腿业务是完全无关的业务，虽然每个业务的数据量可能不太大，放在一个MySQL实例中完全没问题，但是很可能烤羊腿业务的DBA或者开发人员水平很差，动不动给你出一些幺蛾子，直接把数据库搞挂。这个时候，火车票业务的人员虽然技术很优秀，工作也很努力，照样被老板打屁股。解决的办法很简单:惹不起，躲得起。

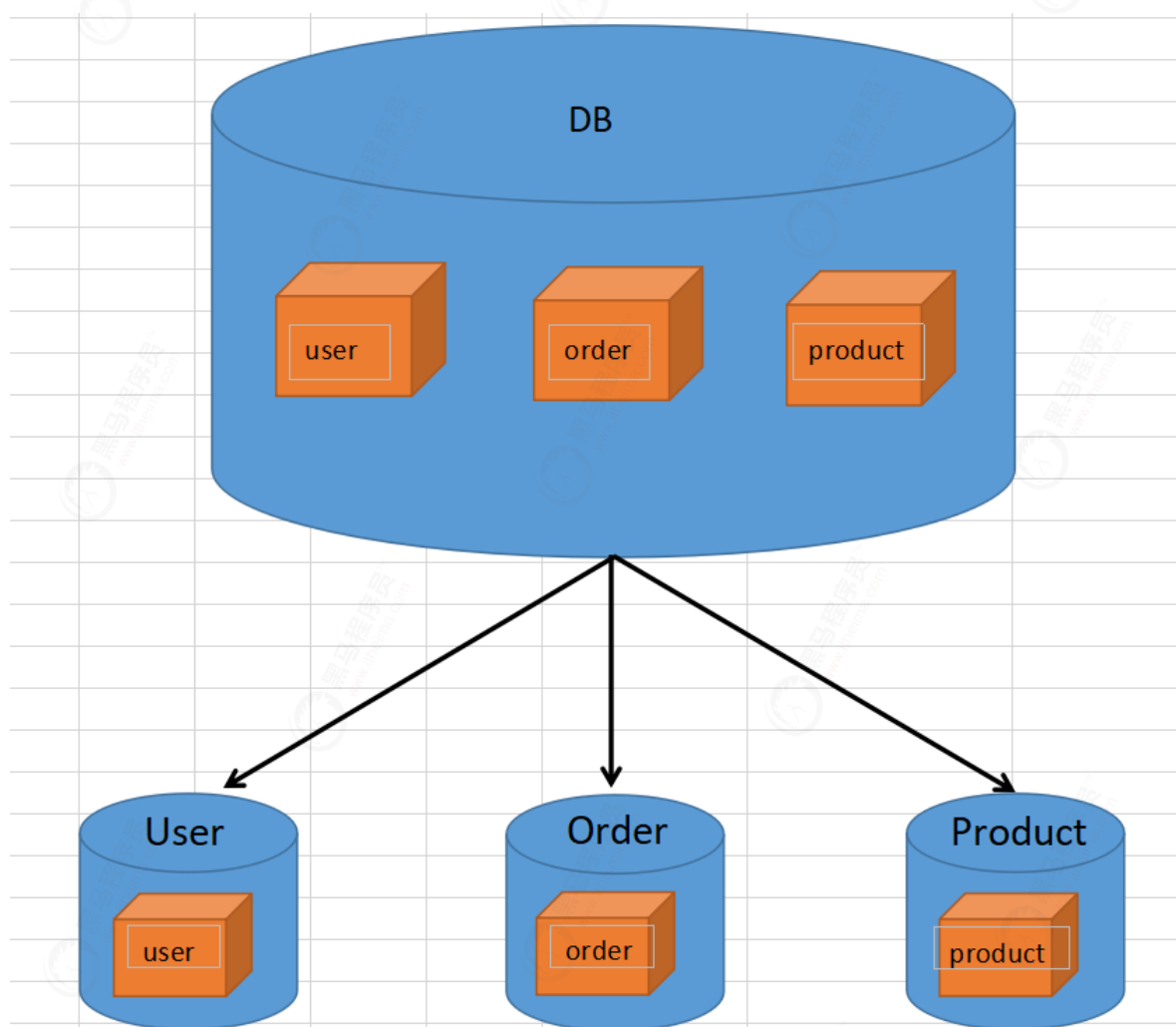
2、分库分表架构方案

1)、垂直切分

垂直切分常见有垂直分库和垂直分表两种。

A.垂直分库:

是根据数据库里面的数据库表的相关性进行拆分，比如：一个数据库里面既存在用户数据，又存在订单数据，那么垂直拆分可以吧用户数据放在用户库，把订单数据放到订单库。另外在“微服务”盛行的今天已经非常普及，按照业务模块来划分不同的数据库，也是一种垂直拆分，而不是像早期一样将所有的数据表都放在同一个数据库中。如下图：



以表为依据，按照业务归属不同，将不同的表拆分到不同的库中

注意事项：

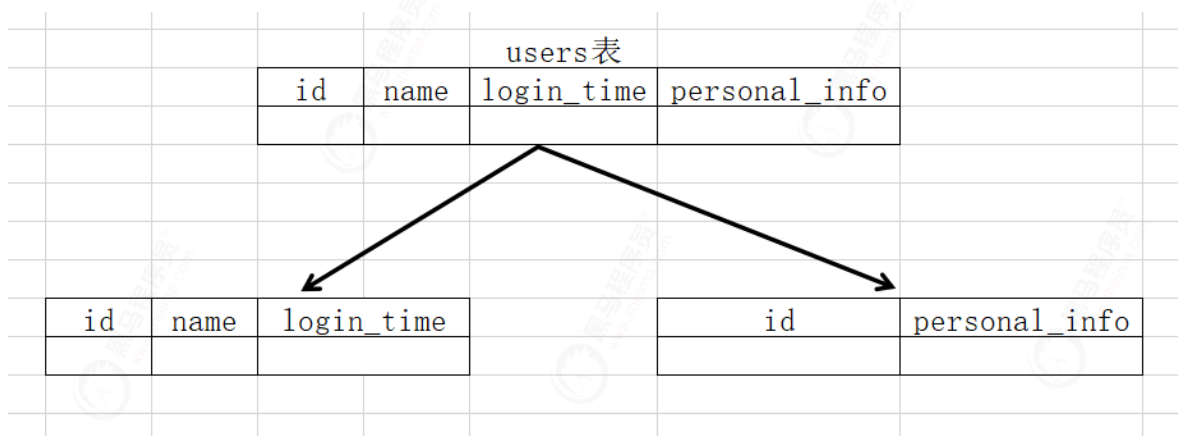
- 1、每个库的结构都一样
- 2、每个库的数据也不一样，没有交集
- 3、所有库的并集是全部数据

使用垂直分库的场景：

当系统绝对并发量上来了，并且可以抽象出单独的业务模块的情况下使用垂直分库方案。

B.垂直分表

在日常开发和设计中比较常见，通俗的说法叫做“大表拆小表”。拆分是基于关系型数据库中的“列”（字段）进行的。通常情况，某个表中的字段比较多，可以新建一张“扩展表”，将不经常使用或者长度较大的字段拆分出去放到“扩展表”中，如下图所示：



以字段为依据，按照字段的活跃性，将表中字段拆到不同的表中（主表和扩展表）。

说明：拆分字段的操作建议在数据库设计阶段就做好。如果是在发展过程中拆分，则需要改写以前的查询语句，会额外带来一定的成本和风险，建议谨慎。

注意事项：

- 1、每个表的结构不一样
- 2、每个表的数据也不一样，一般来说，每个表的字段只有一列交集，一般是主键，用于关联数据
- 3、所有表的并集是全部数据

使用垂直分表的场景：

当系统绝对并发量没有上来，表的记录并不多，但是字段多，并且热点数据和非热点数据在一起，单行数据所需要的存储空间较大，以致于数据库缓存的数据行减少，查询时回去读磁盘数据产生大量随机读IO,产生IO瓶颈的时候使用垂直分表。

C.垂直拆分的优点和缺点：

优点：

- 拆分后业务清晰，拆分规则明确
- 系统之间进行整合或扩展很容易
- 按照成本，应用的等级，应用的类型等将表放在不同的机器上，便于管理
- 便于实现动静分离，冷热分离的数据库表的设计模式
- 数据维护简单。

缺点：

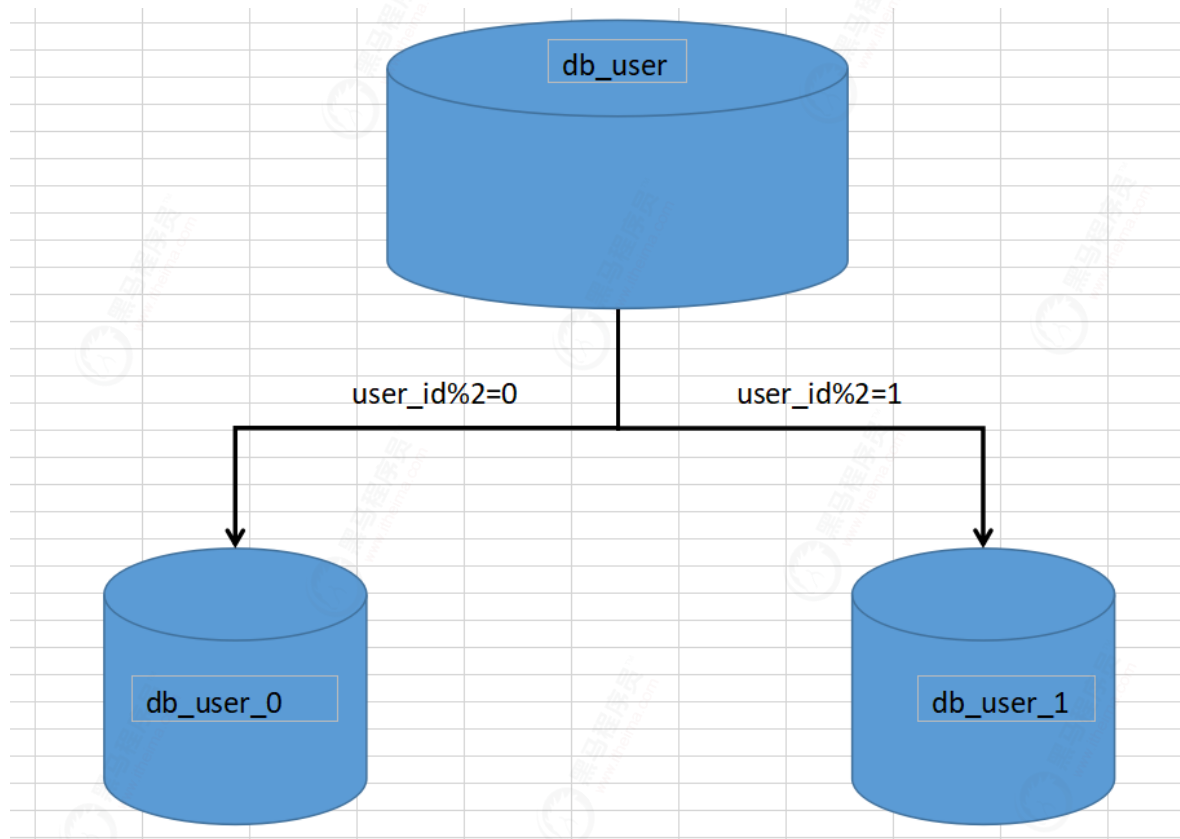
- 主键出现冗余，需要管理冗余列。
- 会引起表连接JOIN操作(增加CPU开销)可以通过在业务服务器上进行join来减少数据库压力。
- 依然存在单表数据量过大的问题(需要水平拆分)。
- 事务处理复杂。

2)、水平切分

水平切分是通过某种策略将数据分片来存储，分为水平分表和水平分库两部分，每片数据会分散到不同的MySQL表或者库中，达到分布式的效果，能够支持非常大的数据量。

A.水平分库

以字段为依据，按照一定策略(hash、range等)，将一个库中的数据拆分到多个库中。



注意事项：

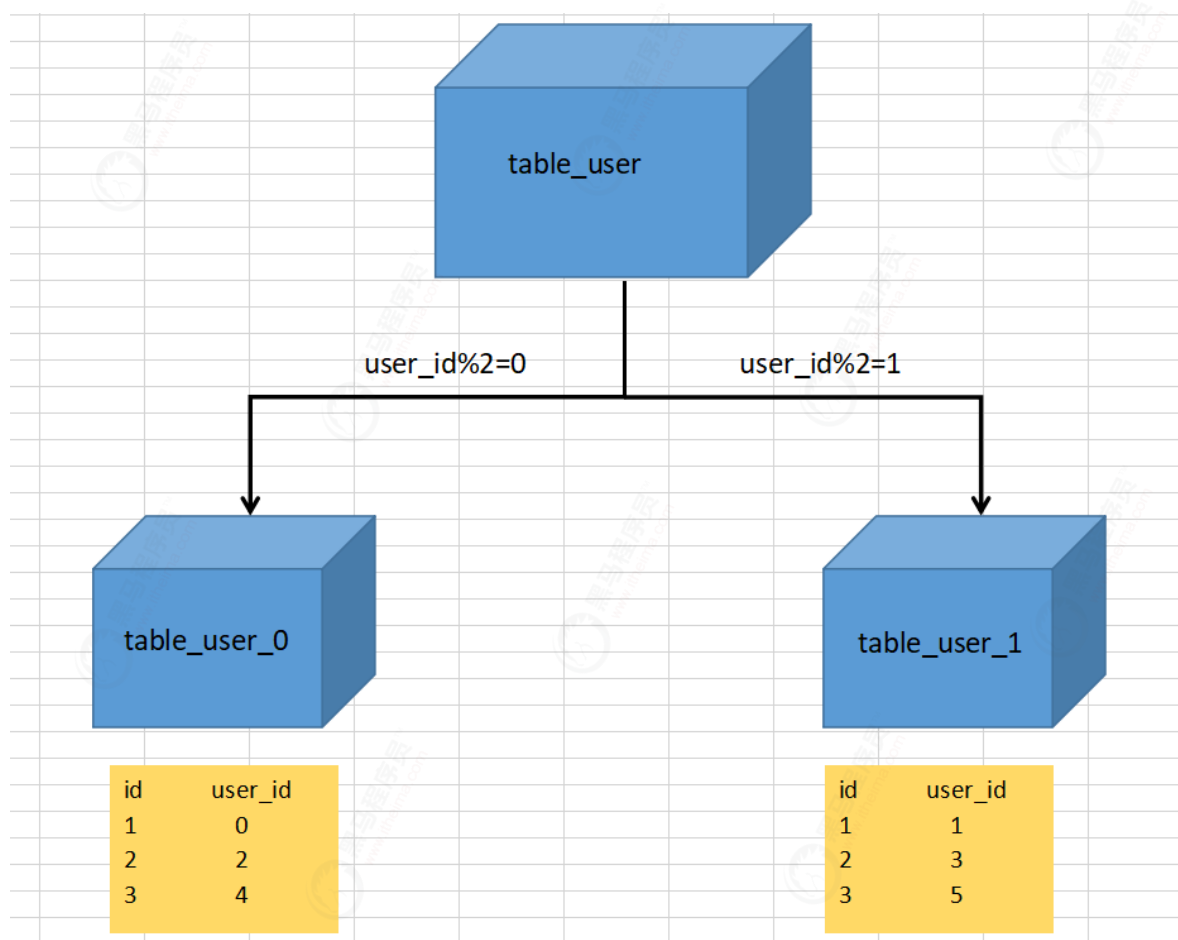
- 1、每个库的结构都一样
- 2、每个库中的数据不一样，没有交集
- 3、所有库的数据并集是全部数据

使用水平分库的场景：

当系统绝对并发量上来了，分表难以从根本上解决问题，并且还没有明显的业务归属进行抽取业务的情况下使用水平分库方案。

B.水平分表

以字段为依据，按照一定策略(hash,range等)，将一个表中的数据拆分到多个表中。



注意事项:

- 1、每个表的结构都一样
- 2、每个表的数据不一样，没有交集，所有表的并集是全部数据

使用水平分表的场景：

当系统绝对并发量没有上来，只是单表的数据量太多，影响SQL效率，加重了CPU负担，以致于成为瓶颈的情况下使用水平分表方案。

C.水平拆分的优点和缺点：

优点：

- 单库单表的数据保持在一定的量级，有助于性能的提高
- 切分的表结构相同，应用层改造较少，只需要增加路由规则即可
- 提高了系统的稳定性和负载能力

缺点：

- 切分后，数据是分散的，很难利用数据库的Join操作，跨库Join性能较差
- 拆分规则难以抽象
- 分片事务的一致性难以解决
- 数据多次拓展难度跟维护量极大

3)、小结

- 垂直分库和水平分库的区别
 - 垂直分库是按照业务模块的不同进行拆分
 - 水平分库是按照一定的策略进行拆分，达到把一个库中的数据进行拆分到不同的数据库中。
- 垂直分表和水平分表的区别

- 垂直分表是按照表中的冷热数据(活跃数据)进行拆分的, 并且拆分完了之后, 一般是分主表和拓展表, 那么两张表的结构不一样, 两张表的关联靠id来支持。
- 水平分表是按照一定的策略进行拆分, 目的是把单表的庞大的数量按照一定的策略分摊到拆分之后的小表中, 拆分之后, 每张表的结构是一样的, 但是数据不一样。

3、分库分表产生的问题剖析

分库分表能有效的缓解单机单库和单表带来的性能瓶颈和压力, 突破网络IO、硬件资源、连接数的瓶颈, 同时也带来一些问题, 问题如下:

1)、事务一致性问题

分布式事务:

当更新内容同时存在于不同库找那个, 不可避免会带来跨库事务问题。跨分片事务也是分布式事务, 没有简单的方案, 一般可使用“XA 协议”和“两阶段提交”处理。

分布式事务能最大限度保证了数据库操作的原子性。但在提交事务时需要协调多个节点, 推后了提交事务的时间点, 延长了事务的执行时间, 导致事务在访问共享资源时发生冲突或死锁的概率增高。

随着数据库节点的增多, 这种趋势会越来越严重, 从而成为系统在数据库层面上水平扩展的枷锁。

最终一致性:

对于那些性能要求很高, 但对一致性要求不高的系统, 往往不苛求系统的实时一致性, 只要在允许的时间段内达到最终一致性即可, 可采用事务补偿的方式。

与事务在执行中发生错误立刻回滚的方式不同, 事务补偿是一种事后检查补救的措施, 一些常见的实现方法有: 对数据进行对账检查, 基于日志进行对比, 定期同标准数据来源进行同步等。

解决办法:

使用分布式事务, 比如阿里的Seata分布式事务, XA两阶段事务, saga柔性事务。

2)、跨节点关联查询Join问题

切分之前, 系统中很多列表和详情表的数据可以通过 Join 来完成, 但是切分之后, 数据可能分布在不同的节点上, 此时 Join 带来的问题就比较麻烦了, 考虑到性能, 尽量避免使用 Join 查询。

解决的办法:

全局表:

全局表, 也可看做“数据字典表”, 就是系统中所有模块都可能依赖的一些表, 为了避免库 Join 查询, 可以将这类表在每个数据库中都保存一份。这些数据通常很少修改, 所以不必担心一致性的问题。

字段冗余:

一种典型的反范式设计, 利用空间换时间, 为了性能而避免 Join 查询。

例如, 订单表在保存 userId 的时候, 也将 userName 也冗余的保存一份, 这样查询订单详情表就可以查到用户名 userName, 就不用查询买家 user 表了。

但这种方法适用场景也有限, 比较适用依赖字段比较少少的情况, 而冗余字段的一致性也较难保证。

数据组装(常用):

在系统 Service 业务层面, 分两次查询, 第一次查询的结果集找出关联的数据 id, 然后根据 id 发起器第二次请求得到关联数据, 最后将获得的结果进行字段组装。这是比较常用的方法。

ER分片:

关系型数据库中，如果已经确定了表之间的关联关系（如订单表和订单详情表），并且将那些存在关联关系的表记录存放在同一个分片上，那么就能较好地避免跨分片 Join 的问题。

可以在一个分片内进行 Join，在 1:1 或 1:n 的情况下，通常按照主表的 ID 进行主键切分。

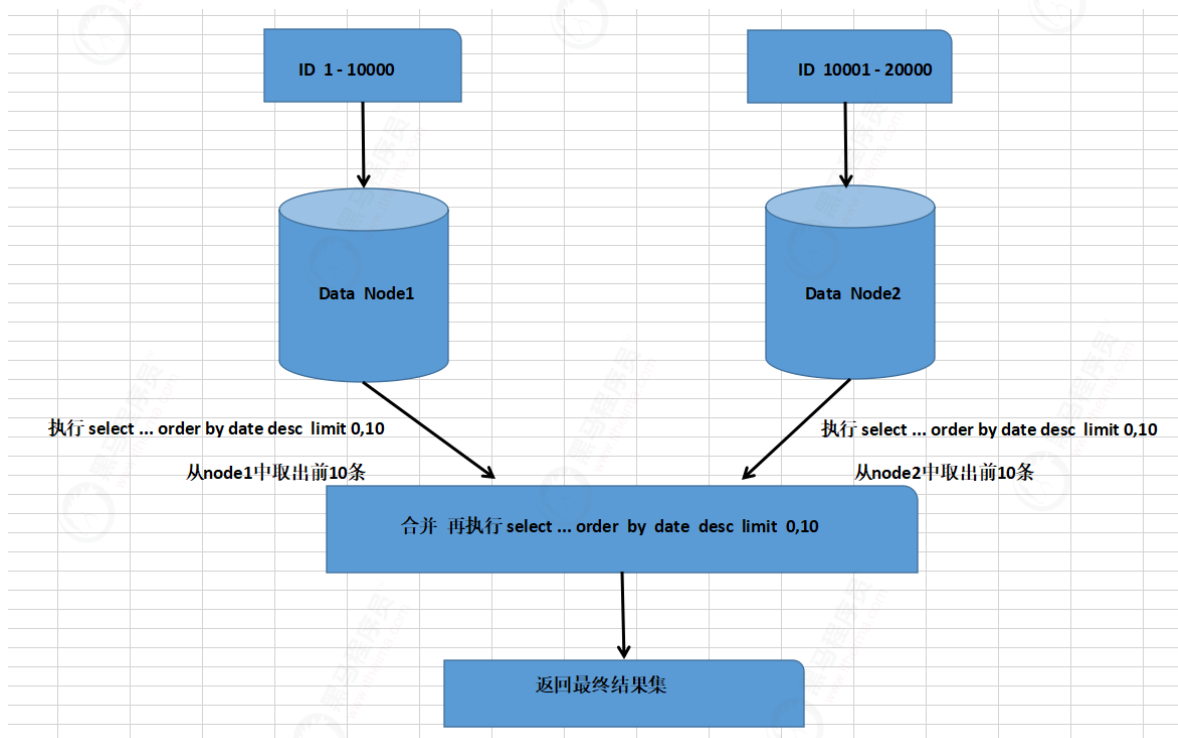
3)、跨节点分页、排序、函数问题

跨节点多库进行查询时，会出现 limit 分页、order by 排序等问题。

分页需要按照指定字段进行排序，当排序字段就是分片字段时，通过分片规则就比较容易定位到指定的分片；当排序字段非分片字段时，就变得比较复杂。

需要先在不同的分片节点中将数据进行排序并返回，然后将不同分片返回的结果集进行汇总和再次排序。

最终返回给用户如下图：



上图只是取第一页的数据，对性能影响还不是很大。但是如果取得页数很大，情况就变得复杂的多。

因为各分片节点中的数据可能是随机的，为了排序的准确性，需要将所有节点的前N页数据都排序好做合并，最后再进行整体排序，这样的操作很耗费 CPU 和内存资源，所以页数越大，系统性能就会越差。

在使用 Max、Min、Sum、Count 之类的函数进行计算的时候，也需要先在每个分片上执行相应的函数，然后将各个分片的结果集进行汇总再次计算。

4)、全局主键避重问题

在分库分表环境中，由于表中数据同时存在不同数据库中，主键值平时使用的自增长将无用武之地，某个分区数据库自生成 ID 无法保证全局唯一。

因此需要单独设计全局主键，避免跨库主键重复问题。这里有一些策略：

1. Twitter的Snowflake（又名“雪花算法”）
2. UUID/GUID（一般应用程序和数据库均支持）
3. MongoDB ObjectID（类似UUID的方式）
4. Ticket Server（数据库生存方式，Flickr采用的就是这种方式）

5)、数据迁移、扩容问题

当业务高速发展、面临性能和存储瓶颈时，才会考虑分片设计，此时就不可避免的需要考虑历史数据的迁移问题。

一般做法是先读出历史数据，然后按照指定的分片规则再将数据写入到各分片节点中。

此外还需要根据当前的数据量个 QPS，以及业务发展速度，进行容量规划，推算出大概需要多少分片（一般建议单个分片的单表数据量不超过 1000W）。

四、电商系统亿级订单数据分库分表实战指导

1、Sharding-JDBC中间件：

1)、sharding-jdbc是什么？

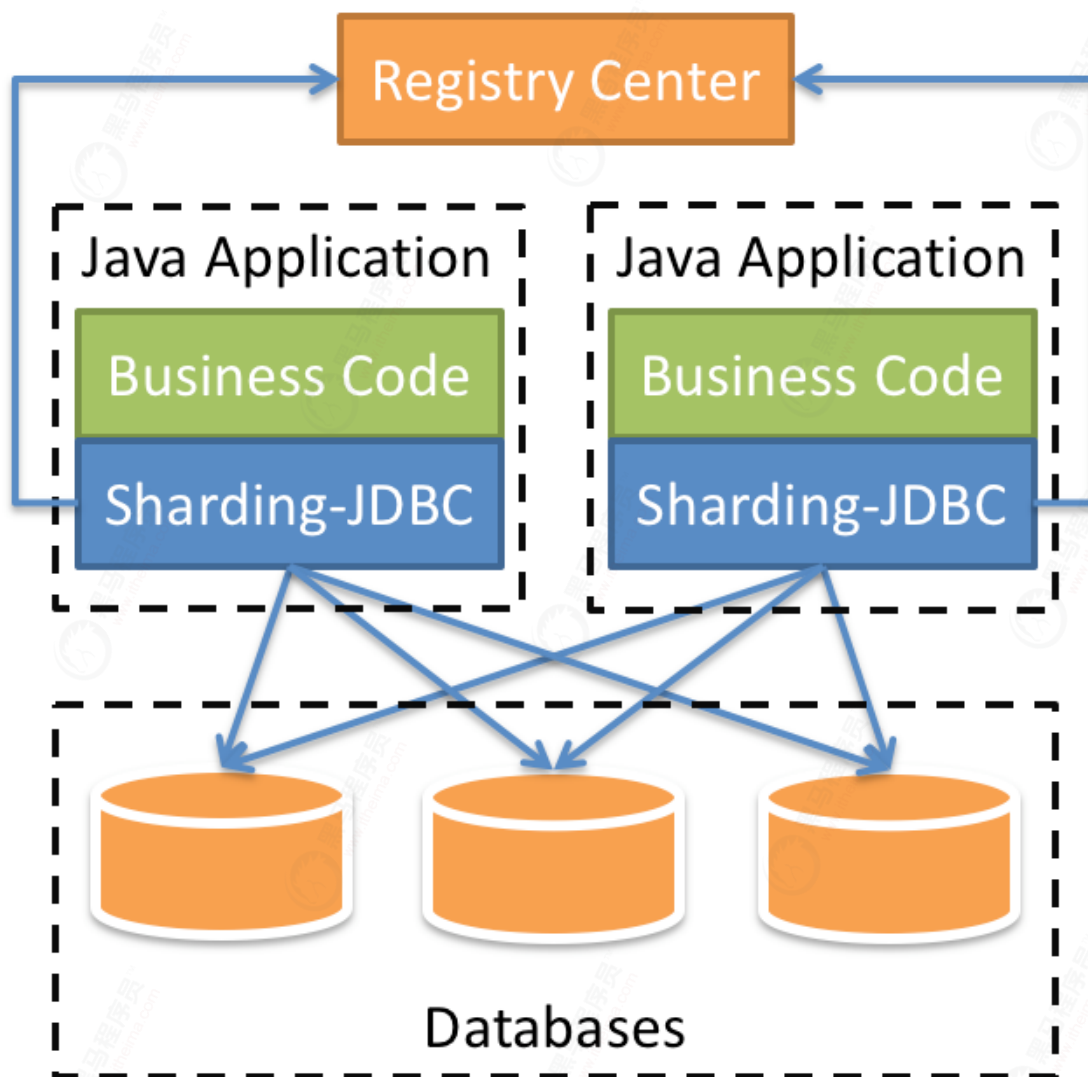
Sharding-JDBC是分布式数据中间件Sharding-Sphere中的重要组成部分，官方的介绍如下：

官网地址：<https://shardingsphere.apache.org/>

Sharding-Sphere是一套开源的分布式数据库中间件解决方案组成的生态圈，它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（计划中）这3款相互独立的产品组成。他们均提供标准化的数据分片、分布式事务和数据库治理功能，可适用于如Java同构、异构语言、容器、云原生等各种多样化的应用场景。

定位为轻量级Java框架，在Java的JDBC层提供的额外服务，它使用客户端直连数据库，以jar包形式提供服务，无需额外部署和依赖，可理解为增强版的JDBC驱动，完全兼容JDBC和各种ORM框架。

- 适用于任何基于JDBC的ORM框架，如JPA,Hibernate,Mybatis,Spring JDBC Template或直接使用JDBC。
- 支持任何第三方的数据库连接池，如DBCP,C3P0,Druid,HikariCP等。
- 支持任意实现JDBC规范的数据库。目前支持MySQL,Oracle,SQLServer以及任何遵循SQL92标准的数据库。



2)、sharding-jdbc能做什么？

Sharding-JDBC的核心功能为数据分片和读写分离，通过sharding-JDBC，应用可以透明的使用jdbc访问已经分库，读写分离的多个数据源，而不用担心数据源的数量以及数据如何分布。

A.sharding-jdbc核心概念：

- 逻辑表 (LogicTable)：具有相同逻辑和数据结构的水平分片数据库(表)。例如订单数据根据主键最后一个数字划分为10个表，他们是从tb_order_0到tb_order_9,其逻辑名称为tb_order。
- 实际表 (ActualTable)：真正存在于分片数据库的物理表，即tb_order_0到tb_order_9。
- 数据节点 (DataNode)：数据分片的最小单元。由数据源名称和数据表组成，例：ds_0.tb_order_0。
- 动态表 (DynamicTable)：逻辑表和物理表不一定需要在配置规则中静态配置。如，按照日期分片的场景，物理表的名称随时间的推移会产生变化。比如：tb_order_0一定是静态表，tb_order_\$有可能是tb_order_1,也有可能是tb_order_2。
- 绑定表 (BindingTable)：指分片规则一致的主表和子表。例如：tb_order表和tb_order_item表，均按照order_id分片，则此两张表互为绑定表关系。绑定表之间的多表关联查询不会出现笛卡尔积关联，关联查询效率将大大提升。

举例说明，如果SQL为：

```
1 SELECT i.* FROM t_order o JOIN t_order_item i ON o.order_id=i.order_id WHERE  
   o.order_id in (10, 11);
```

在不配置绑定表关系时，假设分片键order_id 将数值10路由至第0片，将数值11路由至第1片，那么路由后的SQL应该为4条，他们会出现笛卡尔积现象：

```
1 SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id
  WHERE o.order_id in (10, 11);
2
3 SELECT i.* FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id
  WHERE o.order_id in (10, 11);
4
5 SELECT i.* FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id
  WHERE o.order_id in (10, 11);
6
7 SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id
  WHERE o.order_id in (10, 11);
```

在配置绑定表配置后，路由的SQL应该为2条：

```
1 SELECT i.* FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id
  WHERE o.order_id in (10, 11);
2
3 SELECT i.* FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id
  WHERE o.order_id in (10, 11);
```

其中t_order在FROM的最左侧，ShardingSphere将会以它作为整个绑定表的主表。所有路由计算将会只使用主表的策略，那么t_order_item表的分片计算将会使用t_order的条件。故绑定表之间的分区键要完全相同。

B.sharding-jdbc分片键，算法，策略：

- 分片键：用于分片的数据库字段，是将数据库(表)水平拆分的关键字段。例：将订单表中的订单主键的尾数取模分片，则订单主键为分片字段。
- 分片算法：通过分片算法将数据分片，支持通过 =、>=、<=、>、<、BETWEEN 和 IN 分片。分片算法需要应用方开发者自行实现，可实现的灵活度非常高。

目前提供4种分片算法。由于分片算法和业务实现紧密相关，因此并未提供内置分片算法，而是通过分片策略将各种场景提炼出来，提供更高层级的抽象，并提供接口让应用开发者自行实现分片算法。

◦ 精确分片算法

对应PreciseShardingAlgorithm，用于处理使用单一键作为分片键的=与IN进行分片的场景。需要配合StandardShardingStrategy使用。

◦ 范围分片算法

对应RangeShardingAlgorithm，用于处理使用单一键作为分片键的BETWEEN AND、>、<、>=、<=进行分片的场景。需要配合StandardShardingStrategy使用。

◦ 复合分片算法

对应ComplexKeysShardingAlgorithm，用于处理使用多键作为分片键进行分片的场景，包含多个分片键的逻辑较复杂，需要应用开发者自行处理其中的复杂度。需要配合ComplexShardingStrategy使用。

◦ Hint分片算法

对应HintShardingAlgorithm，用于处理使用Hint行分片的场景。需要配合HintShardingStrategy使用。

- 分片策略：包含分片键和分片算法，由于分片算法的独立性，将其独立抽离。真正可用于分片操作的是分片键 + 分片算法，也就是分片策略。目前提供5种分片策略。

- 标准分片策略

对应StandardShardingStrategy。提供对SQL语句中的=, >, <, >=, <=, IN和BETWEEN AND的分片操作支持。StandardShardingStrategy只支持单分片键，提供PreciseShardingAlgorithm和RangeShardingAlgorithm两个分片算法。PreciseShardingAlgorithm是必选的，用于处理=和IN的分片。RangeShardingAlgorithm是可选的，用于处理BETWEEN AND, >, <, >=, <=分片，如果不配置RangeShardingAlgorithm，SQL中的BETWEEN AND将按照全库路由处理。

- 复合分片策略

对应ComplexShardingStrategy。复合分片策略。提供对SQL语句中的=, >, <, >=, <=, IN和BETWEEN AND的分片操作支持。ComplexShardingStrategy支持多分片键，由于多分片键之间的关系复杂，因此并未进行过多的封装，而是直接将分片键值组合以及分片操作符透传至分片算法，完全由应用开发者实现，提供最大的灵活性。

- 行表达式分片策略

对应InlineShardingStrategy。使用Groovy的表达式，提供对SQL语句中的=和IN的分片操作支持，只支持单分片键。对于简单的分片算法，可以通过简单的配置使用，从而避免繁琐的Java代码开发，如：t_user_\$->{u_id % 8} 表示t_user表根据u_id模8，而分成8张表，表名称为

t_user_0到t_user_7。

- Hint分片策略

对应HintShardingStrategy。通过Hint指定分片值而非从SQL中提取分片值的方式进行分片的策略。

- 不分片策略

对应NoneShardingStrategy。不分片的策略。

C.sharding-jdbc快速入门:

1、需求说明:

使用sharding-jdbc完成对订单表的**水平分表**，通过快速入门的开发，了解sharding-jdbc使用方法

人工创建两张表，t_order_1和t_order_2，这两张表是订单表拆分后的表，通过sharding-jdbc向订单表插入数据，按照一定的分片规则，主键为偶数的进入t_order_1，另一部分数据进入t_order_2，通过sharding-jdbc查询数据，根据SQL语句的内容从t_order_1或t_order_2查询数据。

2、环境搭建:

1、创建订单数据库order_db

```
1 CREATE DATABASE order_db CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

2、在order_db中创建t_order_1、t_order_2表

```
1 DROP TABLE IF EXISTS t_order_1;
2 CREATE TABLE t_order_1(
3     `order_id` BIGINT(20) NOT NULL COMMENT '订单id',
4     `price` DECIMAL(10,2) NOT NULL COMMENT '订单价格',
5     `user_id` BIGINT(20) NOT NULL COMMENT '下单用户id',
6     `status` VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
7     NULL COMMENT '订单状态',
8     PRIMARY KEY (`order_id`) USING BTREE
```

```

8 ) ENGINE = INNODB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT
  = Dynamic;
9
10 DROP TABLE IF EXISTS t_order_2;
11 CREATE TABLE t_order_2(
12     `order_id` BIGINT(20) NOT NULL COMMENT '订单id',
13     `price` DECIMAL(10,2) NOT NULL COMMENT '订单价格',
14     `user_id` BIGINT(20) NOT NULL COMMENT '下单用户id',
15     `status` VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
  NULL COMMENT '订单状态',
16     PRIMARY KEY (`order_id`) USING BTREE
17 ) ENGINE = INNODB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT
  = Dynamic;

```

3、引入maven依赖坐标pom.xml

```

1 <parent>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-parent</artifactId>
4     <version>2.1.0.RELEASE</version>
5     <relativePath/> <!-- lookup parent from repository -->
6 </parent>
7 <dependencies>
8     <dependency>
9         <groupId>org.projectlombok</groupId>
10        <artifactId>lombok</artifactId>
11        <version>1.18.0</version>
12    </dependency>
13    <dependency>
14        <groupId>mysql</groupId>
15        <artifactId>mysql-connector-java</artifactId>
16        <version>5.1.47</version>
17    </dependency>
18    <dependency>
19        <groupId>org.mybatis.spring.boot</groupId>
20        <artifactId>mybatis-spring-boot-starter</artifactId>
21        <version>2.1.0</version>
22    </dependency>
23    <dependency>
24        <groupId>com.alibaba</groupId>
25        <artifactId>druid-spring-boot-starter</artifactId>
26        <version>1.1.16</version>
27    </dependency>
28    <dependency>
29        <groupId>org.apache.shardingsphere</groupId>
30        <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
31        <version>4.0.0-RC1</version>
32    </dependency>
33    <dependency>
34        <groupId>org.springframework.boot</groupId>
35        <artifactId>spring-boot-starter-test</artifactId>
36        <version>2.1.0.RELEASE</version>
37    </dependency>
38 </dependencies>

```

3、配置分片规则:

分片规则配置是sharding-jdbc进行对分库分表操作的重要依据，配置内容包括：数据源、主键生成策略、分片策略等。在application.properties中配置

```
1 #端口号
2 server.port=56000
3 #实例名称
4 spring.application.name= sharding_quick
5 #表示后发现的bean会覆盖之前相同名称的bean
6 spring.main.allow-bean-definition-overriding=true
7 #该配置项就是指将带有下划线的表字段映射为驼峰格式的实体类属性
8 mybatis.configuration.map-underscore-to-camel-case=true
9 #以下是分片规则配置
10
11 ##定义数据源
12 spring.shardingsphere.datasource.names=m1
13
14 spring.shardingsphere.datasource.m1.type=com.alibaba.druid.pool.DruidDataSource
15 spring.shardingsphere.datasource.m1.driver-class-name=com.mysql.jdbc.Driver
16 spring.shardingsphere.datasource.m1.url=jdbc:mysql://118.31.18.203:3306/order_db?useUnicode=true
17 spring.shardingsphere.datasource.m1.username=root
18 spring.shardingsphere.datasource.m1.password=root
19
20 #指定t_order表的数据分布情况，配置数据节点
21 spring.shardingsphere.sharding.tables.t_order.actualDataNodes=m1.t_order_${1..2}
22
23 #指定t_order表的主键生成策略为SNOWFLAKE
24 spring.shardingsphere.sharding.tables.t_order.keyGenerator.column=order_id
25 spring.shardingsphere.sharding.tables.t_order.keyGenerator.type=SNOWFLAKE
26
27 #指定t_order表的分片策略，分片策略包括分片键和分片算法
28 spring.shardingsphere.sharding.tables.t_order.tableStrategy.inline.shardingColumn=order_id
29 spring.shardingsphere.sharding.tables.t_order.tableStrategy.inline.algorithmExpression=t_order_${order_id % 2+1}
30
31 #打开sql输出日志
32 spring.shardingsphere.props.sql.show=true
33
34 #日志级别
35 logging.level.root=info
36 logging.level.org.springframework.web=info
37 logging.level.com.itheima=debug
38 logging.level.druid.sql=debug
```

4、测试新增订单:

1、创建订单实体

```

1  /**
2   * 订单实体
3   */
4  @Data
5  public class Order {
6
7      private Long orderId;
8      private BigDecimal price;
9      private Long userId;
10     private String status;
11 }

```

2、创建mapper接口

```

1  /**
2   * 订单mapper接口
3   */
4  @Mapper
5  public interface OrderMapper {
6      @Insert("insert into t_order(price,user_id,status) values(#{price},#{
7          {userId},{status}})")
8      public int insertOrder(Order order);
9  }

```

3、测试新增订单

```

1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class ShardingQuickApplicationTests {
4
5      @Autowired
6      private OrderMapper orderMapper;
7
8      //测试新增
9      @Test
10     public void testInsertOrder(){
11
12         for (int i = 0; i < 10; i++) {
13             Order order=new Order();
14             order.setPrice(new BigDecimal((i+1)*5));
15             order.setStatus("success"+"|"+i);
16             order.setUserId(1L);
17             orderMapper.insertOrder(order);
18         }
19     }
20 }

```

4、控制台打印sql

```

main] ShardingSphere-SQL          : Rule Type: sharding
main] ShardingSphere-SQL          : Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
main] ShardingSphere-SQL          : SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent()))], :
main] ShardingSphere-SQL          : Actual SQL: m1 ::: insert into t_order_1 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [4999975, 1, success|999994, 491486231994040320]
main] c.itheima.mapper.OrderMapper.insertOrder : <== Updates: 1
main] c.itheima.mapper.OrderMapper.insertOrder : ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
main] c.itheima.mapper.OrderMapper.insertOrder : ==> Parameters: 4999980(BigDecimal), 1(Long), success|999995(String)
main] ShardingSphere-SQL          : Rule Type: sharding
main] ShardingSphere-SQL          : Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
main] ShardingSphere-SQL          : SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent()))], :
main] ShardingSphere-SQL          : Actual SQL: m1 ::: insert into t_order_2 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [4999980, 1, success|999995, 491486232178589697]
main] c.itheima.mapper.OrderMapper.insertOrder : <== Updates: 1
main] c.itheima.mapper.OrderMapper.insertOrder : ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
main] c.itheima.mapper.OrderMapper.insertOrder : ==> Parameters: 4999985(BigDecimal), 1(Long), success|999996(String)

```

通过日志可以发现order_id为奇数的被插入到t_order_2表，为偶数的被插入到t_order_1表，达到预期目标。

5、测试查询订单：

1、创建mapper接口

```
1 //根据id查询多个订单
2 @Select({"<script>"+
3         "select "+
4         " * "+
5         " from t_order t "+
6         " where t.order_id in "+
7         "<foreach collection='orderIds' item='id' open='(' "
8         "separator=', ' close=')>" +
9         "#{id}" +
10        "</foreach>" +
11        "</script>"})
12 public List<Map> selectListByOrderIds(@Param("orderIds") List<Long>
13        orderIds);
```

2、测试类方法

```
1 //根据id查询多个订单
2 @Test
3 public void testSelect(){
4     List<Long> orderIds=new ArrayList<>();
5     orderIds.add(490947906191228928L);
6     orderIds.add(490947906191228929L);
7     orderMapper.selectListByOrderIds(orderIds);
8 }
```

3、打印sql日志

```
ShardingSphere-SQL          : Rule Type: sharding
ShardingSphere-SQL          : Logic SQL: select * from t_order t where t.order_id in ( ?, ? )
ShardingSphere-SQL          : SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables{tables=[Table{name=t_order, alias=Optional.of(t)}])
ShardingSphere-SQL          : Actual SQL: ml ::: select * from t_order_1 t where t.order_id in ( ?, ? ) ::: [490947906191228928, 490947906191228929]
ShardingSphere-SQL          : Actual SQL: ml ::: select * from t_order_2 t where t.order_id in ( ?, ? ) ::: [490947906191228928, 490947906191228929]
c.i.m.OrderMapper.selectListByOrderIds : <= Total: 1
com.alibaba.druid.pool.DruidDataSource : (dataSource-1) closing ...
com.alibaba.druid.pool.DruidDataSource : (dataSource-1) closed
```

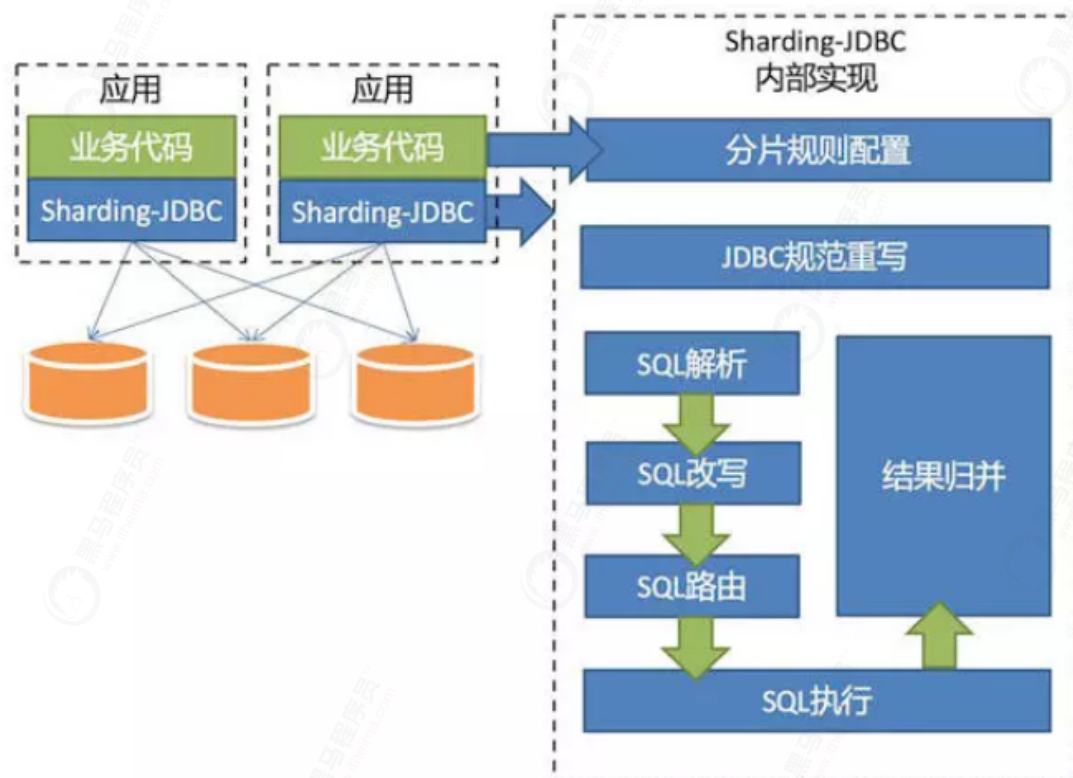
通过日志可以发现，根据传入order_id的奇偶不同，sharding-jdbc分别去不同的表检索数据，达到预期目标。

6、小结：

通过日志分析，Sharding-JDBC在拿到用户要执行的sql之后干了哪些事儿：

- 解析sql，获取分片键值，在本例中是order_id。
- Sharding-JDBC通过规则配置 t_order_\${order_id % 2 + 1}，知道了当order_id为偶数时，应该往t_order_1表插数据，为奇数时，往t_order_2插数据。
- 于是Sharding-JDBC根据order_id的值改写sql语句，改写后的SQL语句是真实所要执行的SQL语句。
- 执行改写后的真实sql语句。
- 将所有真正执行sql的结果进行汇总合并，返回。

D.sharding-jdbc执行原理：



由图可知，在sql执行过程中需要经过几个过程：

例如现在有一条查询语句：

```
1 | select * from tb_user where id=10;
```

进行了分库分表操作，2个库ds0,ds1,采用的分片键为id,逻辑表为tb_user,真实表为tb_user_0,tb_user_1两张表，分库、分表算法均为取余(%2)。

- sql解析：通过解析sql语句提取分片键列与值进行分片，例如比较符=、in、between and,及查询的表等
- sql改写：根据解析结果，及采用的分片逻辑改写sql，上例经过sql改写之后，真实的语句为：

```
1 | select * from tb_user_0 where id=10;
```

- sql路由：找到sql需要去哪个库，哪个表执行语句，上例sql根据采用的策略可以得到将在ds0库tb_user_0表执行语句。
- sql执行：执行改写后的sql。
- 结果归并：当我们执行某些复杂语句时，sql可能会在多个库、多个表中执行，sql分别对应执行后会对结果集进行归并操作，得到最终的结果。

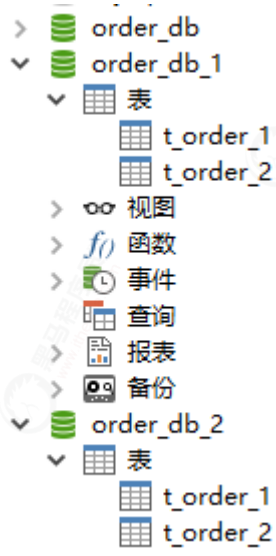
3)、水平分库

A.需求说明：

水平分库是把同一个表的数据按一定规则拆到不同的数据库中，每个库可以放在不同的服务器上。

接下来咱们继续对快速入门中的例子进行完善。

B.将原有order_db库拆分为order_db_1、order_db_2



```
1 CREATE DATABASE order_db_1 CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
2 CREATE DATABASE order_db_2 CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
3
4 DROP TABLE IF EXISTS t_order_1;
5 CREATE TABLE t_order_1(
6     `order_id` BIGINT(20) NOT NULL COMMENT '订单id',
7     `price` DECIMAL(10,2) NOT NULL COMMENT '订单价格',
8     `user_id` BIGINT(20) NOT NULL COMMENT '下单用户id',
9     `status` VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
10 NULL COMMENT '订单状态',
11     PRIMARY KEY (`order_id`) USING BTREE
12 ) ENGINE = INNODB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT
13 = Dynamic;
14
15 DROP TABLE IF EXISTS t_order_2;
16 CREATE TABLE t_order_2(
17     `order_id` BIGINT(20) NOT NULL COMMENT '订单id',
18     `price` DECIMAL(10,2) NOT NULL COMMENT '订单价格',
19     `user_id` BIGINT(20) NOT NULL COMMENT '下单用户id',
20     `status` VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT
21 NULL COMMENT '订单状态',
22     PRIMARY KEY (`order_id`) USING BTREE
23 ) ENGINE = INNODB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT
24 = Dynamic;
```

C.分片规则修改

由于数据库拆分了两个，这里需要配置两个数据源。分库需要配置分库的策略，和分表策略的意义类似，通过分库策略实现数据操作针对分库的数据库进行操作

```
1 ##定义数据源
2 spring.shardingsphere.datasource.names=m1,m2
3 #数据源1
4 spring.shardingsphere.datasource.m1.type=com.alibaba.druid.pool.DruidDataSou
5 rce
6 spring.shardingsphere.datasource.m1.driver-class-name =com.mysql.jdbc.Driver
7 spring.shardingsphere.datasource.m1.url=jdbc:mysql://118.31.18.203:3306/orde
8 r_db_1?useUnicode=true
9 spring.shardingsphere.datasource.m1.username=root
10 spring.shardingsphere.datasource.m1.password=root
```



```

9  #数据源2
10 spring.shardingsphere.datasource.m2.type=com.alibaba.druid.pool.DruidDataSource
11 spring.shardingsphere.datasource.m2.driver-class-name=com.mysql.jdbc.Driver
12 spring.shardingsphere.datasource.m2.url=jdbc:mysql://118.31.18.203:3306/order_db_2?useUnicode=true
13 spring.shardingsphere.datasource.m2.username=root
14 spring.shardingsphere.datasource.m2.password=root
15 .....
16 #分库策略，以user_id为分片键，分片策略为user_id%2+1，user_id为偶数操作m1数据源，否则操作m2。
17 spring.shardingsphere.sharding.tables.t_order.databaseStrategy.inline.shardingColumn=user_id
18 spring.shardingsphere.sharding.tables.t_order.databaseStrategy.inline.algorithmExpression=m$->{user_id % 2+1 }
19

```

分库策略，如何将一个逻辑表映射到多个数据源

spring.shardingsphere.sharding.tables.<逻辑表名称>.database-strategy.<分片策略>.<分片策略属性名>= #分片策略属性值

分表策略，如何将一个逻辑表映射为多个实际表

spring.shardingsphere.sharding.tables.<逻辑表名称>.table-strategy.<分片策略>.<分片策略属性名>= #分片策略属性值

目前例子中都使用inline分片策略，若对其他分片策略细节若感兴趣，请查阅官方文档：

<https://shardingsphere.apache.org/document/legacy/4.x/document/cn/features/sharding/concept/sharding/>

D.测试插入订单

1、测试代码

```

1  //-----水平分库
2  @Test
3  public void testInsertTable(){
4
5      for (int i = 0; i < 10; i++) {
6          Order order=new Order();
7          order.setPrice(new BigDecimal((i+1)*5));
8          order.setStatus("WAIT_PAY"+"|"+i);
9          order.setUserId(1L);
10         orderMapper.insertOrder(order);
11     }
12     for (int i = 0; i < 10; i++) {
13         Order order=new Order();
14         order.setPrice(new BigDecimal((i+1)*10));
15         order.setStatus("WAIT_PAY"+"|"+i);
16         order.setUserId(2L);
17         orderMapper.insertOrder(order);
18     }
19 }

```

2、打印sql日志

```

: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent()))]), rc
: Actual SQL: m2 :: insert into t_order_2 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [45, 1, WAIT_PAY|8, 491708668568403969]
der : <= Updates: 1
der : ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
der : ==> Parameters: 50(BigDecimal), 1(Long), WAIT_PAY|9(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent()))]), rc
: Actual SQL: m2 :: insert into t_order_1 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [50, 1, WAIT_PAY|9, 491708668799090688]
der : <= Updates: 1
der : ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
der : ==> Parameters: 10(BigDecimal), 2(Long), WAIT_PAY|0(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent()))]), rc
: Actual SQL: m1 :: insert into t_order_2 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [10, 2, WAIT_PAY|0, 491708669004611585]
der : <= Updates: 1
der : ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
der : ==> Parameters: 20(BigDecimal), 2(Long), WAIT_PAY|1(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent()))]), rc
: Actual SQL: m1 :: insert into t_order_1 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [20, 2, WAIT_PAY|1, 491708669222715392]
der : <= Updates: 1
der : ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
der : ==> Parameters: 30(BigDecimal), 2(Long), WAIT_PAY|2(String)

```

3、查看数据库和表

order_db_1			
对象 t_order_1 @order_db_1 (阿...		对象 t_order_2 @order_db_1 (阿...	
order_id	price	user_id	status
491708669222715392	20	2	WAIT_PAY 1
491708669663117312	40	2	WAIT_PAY 3
491708670116102144	60	2	WAIT_PAY 5
491708670556504064	80	2	WAIT_PAY 7
491708670980128768	100	2	WAIT_PAY 9

order_db_2			
对象 t_order_1 @order_db_2 (阿...		对象 t_order_2 @order_db_2 (阿...	
order_id	price	user_id	status
491708667008122880	10	1	WAIT_PAY 1
491708667448524800	20	1	WAIT_PAY 3
491708667901509632	30	1	WAIT_PAY 5
491708668362883072	40	1	WAIT_PAY 7
491708668799090688	50	1	WAIT_PAY 9

通过日志和数据库可以看出，根据分片键user_id的奇偶不同，数据分别落在了不同数据源，达到目标。

E.测试查询订单

1、测试代码

原快速入门代码

```

1 //根据id查询多个订单
2 @Test
3 public void testSelect(){
4     List<Long> orderIds=new ArrayList<>();
5     orderIds.add(491708667008122880L); //在order_db_2库中t_order_1表
6     orderIds.add(491708666341228545L); //在order_db_2库中t_order_2表
7     orderMapper.selectListByOrderIds(orderIds);
8 }

```

2、打印sql日志

```

: Rule Type: sharding
: Logic SQL: select * from t_order t where t.order_id in ( ?, ? )
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_order, alias=0
: Actual SQL: m1 ::: select * from t_order_1 t where t.order_id in ( ?, ? ) :: [491708667008122880, 491708666341228545]
: Actual SQL: m1 ::: select * from t_order_2 t where t.order_id in ( ?, ? ) :: [491708667008122880, 491708666341228545]
: <== Total: 0

```

1)、大家发现，所有查的都是m1数据源上的，这是不对的，因为测试代码中两个订单id都是在order_db_2数据库m2中的

问题的原因：是我们的分片规则中已经是分库又分表了，我们发现我们对于表的分片规则中是这样的，我们设定的是固定了m1数据源。

```

#指定t_order表的数据分布情况，配置数据节点
spring.shardingsphere.sharding.tables.t_order.actualDataNodes=m1.t_order_-$->{1..2}

#指定t_order表的主键生成策略为SNOWFLAKE
spring.shardingsphere.sharding.tables.t_order.keyGenerator.column=order_id
spring.shardingsphere.sharding.tables.t_order.keyGenerator.type=SNOWFLAKE

```

2)、更改配置数据节点数据源

```

1 #指定t_order表的数据分布情况，配置数据节点
2 spring.shardingsphere.sharding.tables.t_order.actualDataNodes=m$->
   {1..2}.t_order_-$->{1..2}

```

3)、然后在进行测试

```

: Rule Type: sharding
: Logic SQL: select * from t_order t where t.order_id in ( ?, ? )
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_order, alias=Optic
: Actual SQL: m1 ::: select * from t_order_1 t where t.order_id in ( ?, ? ) :: [491708667008122880, 491708666341228545]
: Actual SQL: m1 ::: select * from t_order_2 t where t.order_id in ( ?, ? ) :: [491708667008122880, 491708666341228545]
: Actual SQL: m2 ::: select * from t_order_1 t where t.order_id in ( ?, ? ) :: [491708667008122880, 491708666341228545]
: Actual SQL: m2 ::: select * from t_order_2 t where t.order_id in ( ?, ? ) :: [491708667008122880, 491708666341228545]
: <== Total: 2

```

4)、我们发现出现了四条sql语句，这是什么原因呢？

大家一定要先确定一件事，就是我们目前的配置规则中是即分库又分表的规则，所以我们的sql语句中只涉及到order_id，而这个order_id是分表的分片键，并没有分库的分片键，所以shardingjdbc将广播路由到所有的库中，然后在每个库中去路由对应的数据节点表。

5)、此时我们去修改sql语句，加入user_id分片键

```

1 //根据订单id和userId查询多个订单
2 @Select({"<script>" +
3     "select " +
4     " * " +
5     " from t_order t" +
6     " where t.order_id in " +
7     "<foreach collection='orderIds' item='id' open='(' separator=',',
close=')'>" +
8     "#{id}" +
9     "</foreach>" +
10    "AND t.user_id=#{userId}" +
11    "</script>"}
12 public List<Map> selectListByUserIdAndOrderIds(@Param("userId") Long
userId,@Param("orderIds") List<Long> orderIds);

```

6)、修正测试代码

```

1 //根据orderId和userId查询多个订单
2 @Test
3 public void testSelectByUserIdAndOrderId(){
4     List<Long> orderIds=new ArrayList<>();
5     orderIds.add(491708667008122880L);//在order_db_2库中t_order_1表
6     orderIds.add(491708666341228545L);//在order_db_2库中t_order_2表
7     Long userId=1L;
8     List<Map> maps = orderMapper.selectListByUserIdAndOrderIds(userId,
orderIds);
9     System.out.println(maps);
10 }

```

7)、sql日志打印

```

: Rule Type: sharding
: Logic SQL: select * from t_order t where t.order_id in ( ?, ? ) AND t.user_id=?
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_order, alias=Optional.of(t))]), r
: Actual SQL: m2 ::: select * from t_order_1 t where t.order_id in ( ?, ? ) AND t.user_id=? ::: [491708667008122880, 491708666341228545, 1]
: Actual SQL: m2 ::: select * from t_order_2 t where t.order_id in ( ?, ? ) AND t.user_id=? ::: [491708667008122880, 491708666341228545, 1]
: <== Total: 2
order_id=491708666341228545, status=WAIT_PAY[0]]

```

查询条件user_id为1，根据分片策略 $m \rightarrow \{user_id \% 2 + 1\}$ 计算得出m2，此sharding-jdbc将sql路由到m2

4)、垂直分库

A.需求说明:

是指按照业务将表进行分类，分布到不同的数据库上面，每个库可以放在不同的服务器上，它的核心理念是专库专用。接下来看一下在单库中实现垂直分库操作。

B.创建数据库:

```

1  #创建数据库
2  CREATE DATABASE user_db CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
3
4  #创建表
5  DROP TABLE IF EXISTS t_user;
6  CREATE TABLE t_user(
7      user_id BIGINT(20) NOT NULL COMMENT '用户id',
8      fullname VARCHAR(255) CHARACTER SET 'utf8' COLLATE 'utf8_general_ci' NOT
9      NULL COMMENT '用户姓名',
10     user_type CHAR(1) DEFAULT NULL COMMENT '用户类型',
11     PRIMARY KEY (user_id) USING BTREE
12 ) ENGINE = INNODB CHARACTER SET = 'utf8' COLLATE = 'utf8_general_ci'
13 ROW_FORMAT= Dynamic;

```

C.分片规则修改:

在原有的基础之上加入

```

1  ##定义数据源
2  spring.shardingsphere.datasource.names=m0,m1,m2
3
4  #数据源0
5  spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSou
6  rce
7  spring.shardingsphere.datasource.m0.driver-class-name =com.mysql.jdbc.Driver
8  spring.shardingsphere.datasource.m0.url=jdbc:mysql://118.31.18.203:3306/user
9  _db?useUnicode=true
10
11 #t_user分表策略
12 spring.shardingsphere.sharding.tables.t_user.actualDataNodes=m0.t_user
13 spring.shardingsphere.sharding.tables.t_user.tableStrategy.inline.shardingCo
14 lumn=user_id
15 spring.shardingsphere.sharding.tables.t_user.tableStrategy.inline.algorithmE
16 xpression=t_user
17 #指定t_user表的主键生成策略
18 spring.shardingsphere.sharding.tables.t_user.keyGenerator.column=user_id
19 spring.shardingsphere.sharding.tables.t_user.keyGenerator.type=SNOWFLAKE

```

D. 数据操作:

userMapper接口

```

1  /**
2   * user表的mapper接口
3   */
4  @Mapper
5  public interface UserMapper {
6      //插入用户
7      @Insert("insert into t_user(fullname,user_type)values(#{fullname},#{
8          userType})")
9      public int insertUser(@Param("fullname") String
10         fullname,@Param("userType") String userType);
11
12      //根据id查询多个用户

```



```

11     @Select("<script>" +
12             " select " +
13             " * " +
14             " from t_user t " +
15             " where t.user_id in " +
16             "<foreach collection='userIds' item='id' open='('
17             separator=',' close=')'>" +
18             "#{id}" +
19             "</foreach>" +
20             "</script>")
21     public List<Map> selectListByUserIds(@Param("userIds") List<Long>
22     userIds);

```

E.测试代码

```

1 //-----垂直分库
2
3 @Autowired
4 private UserMapper userMapper;
5 //测试新增
6 @Test
7 public void testInsertUser(){
8     for (int i = 0; i < 10; i++) {
9         userMapper.insertUser("张三"+i,i+"");
10    }
11 }
12
13 //根据orderId和userId查询多个订单
14 @Test
15 public void testSelectUsers(){
16     List<Long> userIds=new ArrayList<>();
17     userIds.add(491758273087668224L);
18     userIds.add(491758273284800513L);
19     List<Map> maps = userMapper.selectListByUserIds(userIds);
20     System.out.println(maps);
21 }

```

F.sql日志打印:

插入用户:

```

: Rule Type: sharding
: Logic SQL: insert into t_user(fullname,user_type)values(?,?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_user, ali
: Actual SQL: m0 ::: insert into t_user (fullname, user_type, user_id) VALUES (?, ?, ?) ::: [张三9, 9, 491758274736029696]
: <== Updates: 1

```

查询用户:

```

: Rule Type: sharding
: Logic SQL: select * from t_user t where t.user_id in ( ?, ? )
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_user, alias=
: Actual SQL: m0 ::: select * from t_user t where t.user_id in ( ?, ? ) ::: [491758273087668224, 491758273284800513]
: <== Total: 2
, fullname=???2]

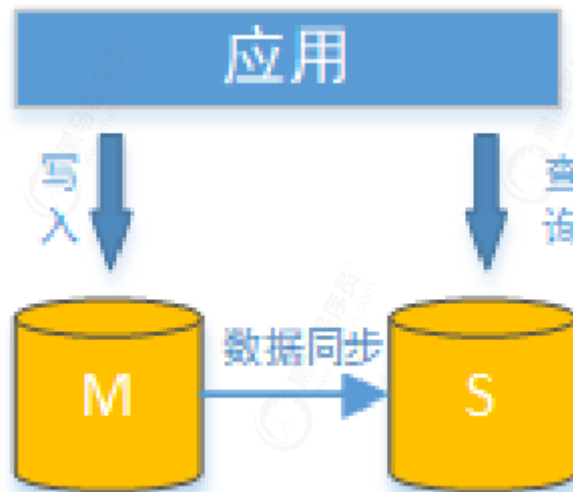
```

通过日志可以看出t_user表的插入和查询数据均被落在了m0数据源, 达到目标。

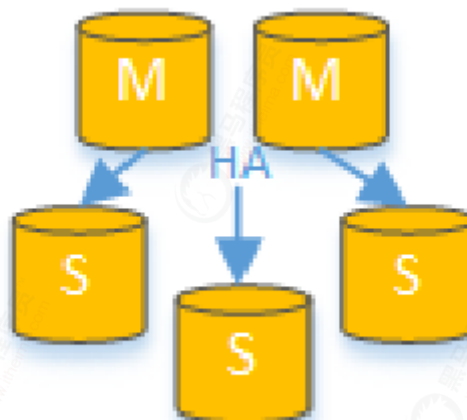
5)、读写分离

A.理解读写分离:

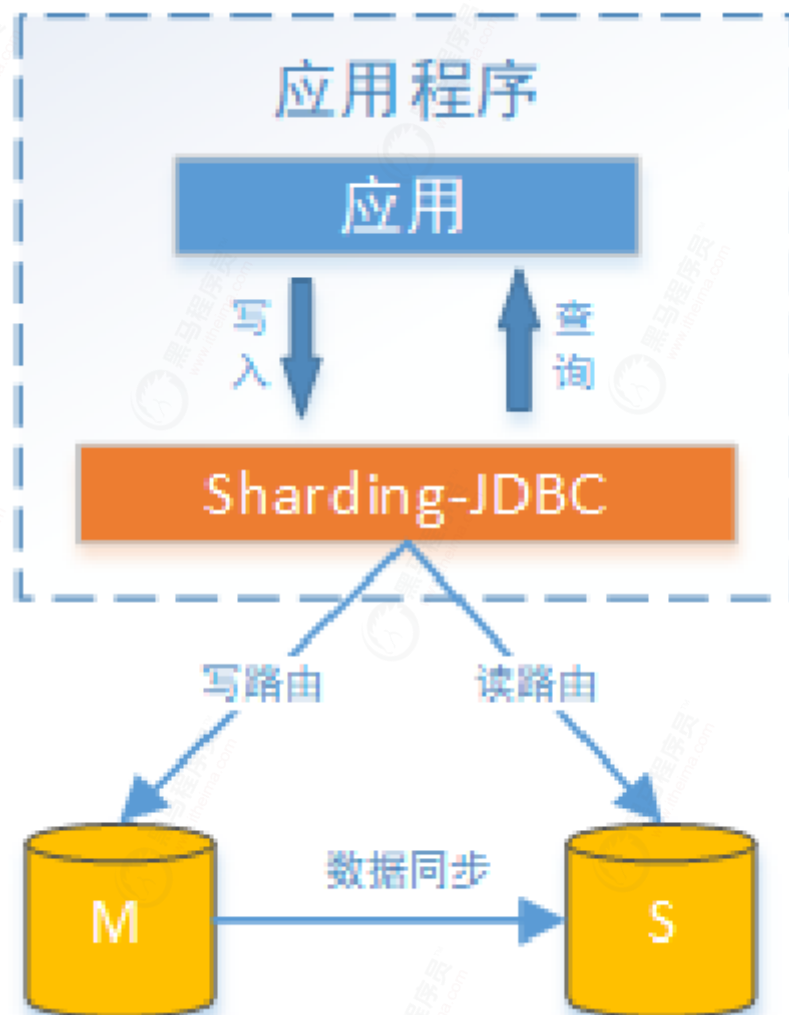
面对日益增加的系统访问量，数据库的吞吐量面临着巨大瓶颈。对于同一时刻有大量并发读操作和较少写操作类型的应用系统来说，将数据库拆分为主库和从库，主库负责处理事务性的增删改操作，从库负责处理查询操作，能够有效的避免由数据更新导致的行锁，使得整个系统的查询性能得到极大的改善。



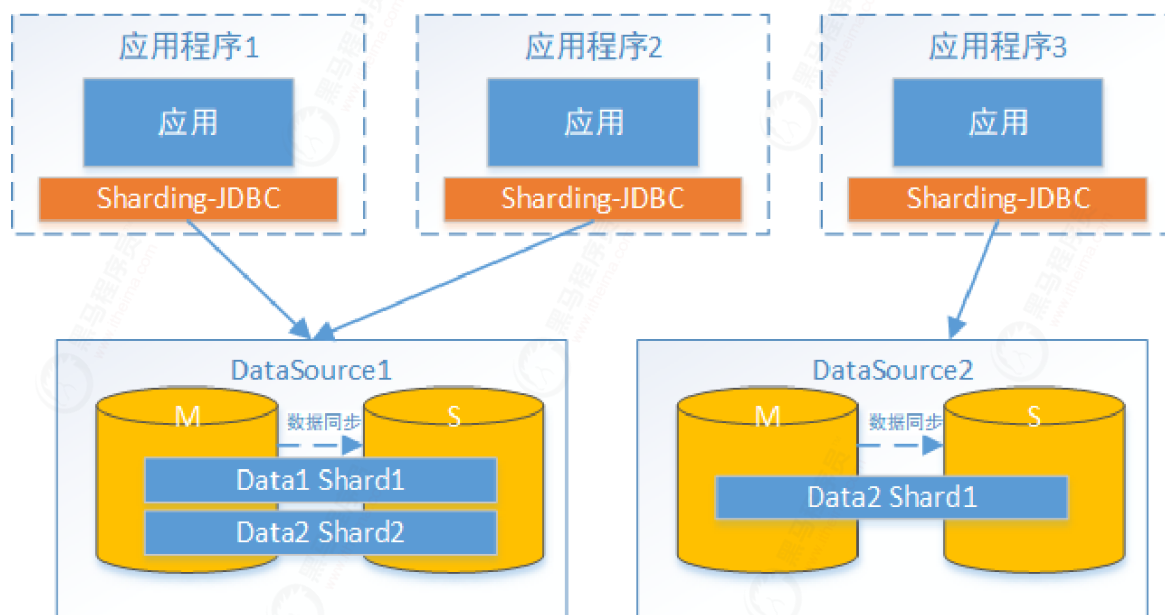
通过一主多从的配置方式，可以将查询请求均匀分散到多个数据副本，能够进一步的提升系统的处理能力。使用多主多从的方式，不但能够提升系统的吞吐量，还能够提升系统的可用性，可以达到在任何一台数据库宕机，甚至磁盘物理损坏的情况下仍然不影响系统的正常运行。



读写分离的数据节点中的数据内容是一致的，而水平分片的每个数据节点的数据内容却并不相同。将水平分片和读写分离联合使用，能够更加有效的提升系统的性能。Sharding-JDBC读写分离则是根据SQL语义的分析，将读操作和写操作分别路由至主库与从库。它提供透明化读写分离，让使用方尽量像使用一个数据库一样使用主从数据库集群。



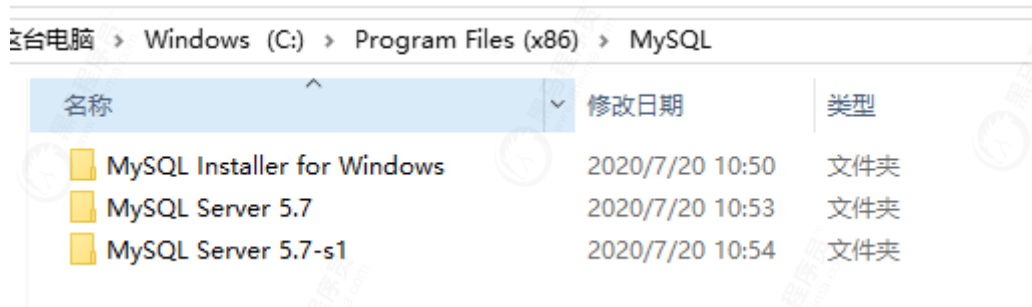
Sharding-JDBC提供一主多从的读写分离配置，可独立使用，也可配合分库分表使用，同一线程且同一数据库连接内，如有写入操作，以后的读操作均从主库读取，用于保证数据一致性。Sharding-JDBC不提供主从数据库的数据同步功能，需要采用其他机制支持。



接下来，咱们对上面例子中user_db进行读写分离实现。为了实现Sharding-JDBC的读写分离，首先，要进行mysql的主从同步配置。

B.mysql主从同步设置(window版):

1、新增mysql实例



复制原有mysql如: C:\Program Files (x86)\MySQL\MySQL Server 5.7(作为主库) -> C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1(作为从库), 并修改以下从库的my.ini:

```
1 [mysqld]
2 # The TCP/IP Port the MySQL Server will listen on
3 port=3307
4 # Path to installation directory. All paths are usually resolved relative to
  this.
5 basedir="C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1/"
6 # Path to the database root
7 datadir=C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\Data
```

```
# The TCP/IP Port the MySQL Server will listen on
port=3307

# Path to installation directory. All paths are usually resolved relative to
basedir="C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1/"

# Path to the database root
datadir=C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\Data
```

然后将从库安装为windows服务, 注意配置文件位置:

```
1 C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\bin>mysqld install mysql-s1
  --defaults-file="C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\my.ini"
```

```
C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\bin>mysqld install mysql-s1 --defaults-file="C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\my.ini"
Service successfully installed.
C:\Program Files (x86)\MySQL\MySQL Server 5.7-s1\bin>m
```

注意事项:

使用管理员身份进行cmd, 否则会报"Install/Remove of the Service Denied"错误

由于从库是从主库复制过来的, 因此里面的数据完全一致, 可使用**原来的账号、密码**登录。

打开计算机服务列表可以查看到服务中有两个mysql服务



2.修改主、从库的配置文件(my.ini), 新增内容如下:

主库中添加

```

1 #开启日志
2 log-bin=mysql-bin
3 #设置服务id,主从服务不能一致
4 server-id=1
5
6 #设置需要同步的数据库
7 binlog-do-db=user_db
8 #屏蔽系统库同步
9 binlog-ignore-db=mysql
10 binlog-ignore-db=information_schema
11 binlog-ignore-db=performance_schema

```

从库中添加

```

1 #开启日志
2 log-bin=mysql-bin
3 #设置服务id,主从不能一致
4 server-id=2
5
6 #设置需要同步的数据库
7 replicate_wild_do_table=user_db.%
8 #屏蔽系统库同步
9 replicate_wild_ignore_table=mysql.%
10 replicate_wild_ignore_table=information_schema.%
11 replicate_wild_ignore_table=performance_schema.%

```

重启主库和从库:

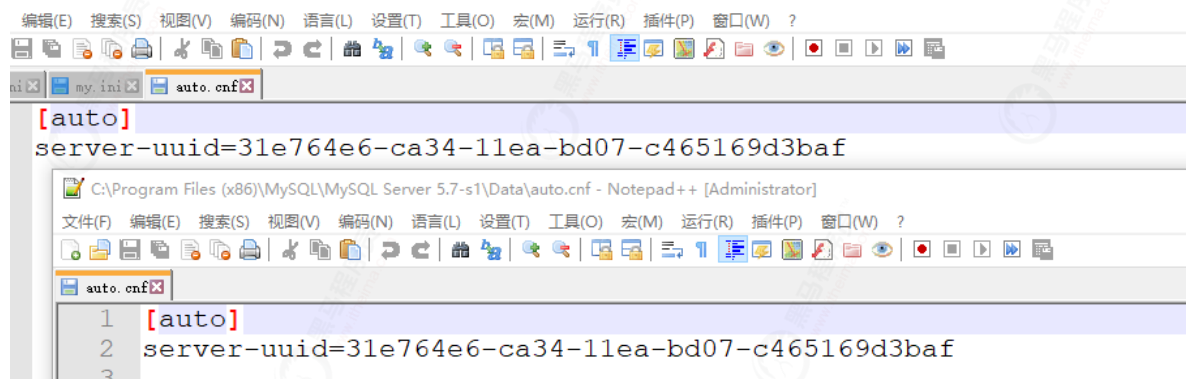
```

1 net start [主库服务名称]
2 net start [从库服务名称]

```

MongoDB	Mon...	正在...	自动	本地系统
MySQL57		正在...	自动	网络服务
mysql-s1		正在...	自动	本地系统

请注意,主从 MySQL下的数据(data)目录下有个文件auto.cnf,文件中定义了uuid,要保证主从数据库实例的 uuid不一样,建议直接删除掉,重启服务后将会重新生成。



3.授权主从复制专用账号

切换到主库中


```

1 #授权主备复制专用账号
2 GRANT replication SLAVE ON *.* TO 'db_sync'@'%' IDENTIFIED by 'db_sync';
3 #刷新权限
4 FLUSH PRIVILEGES;
5 #确认位点 记录下文件名以及位点
6 SHOW MASTER STATUS;

```

信息	结果1	概况	状态	
File	Position	Binlog_Do_DB	Binlog_Ignore_DB	Executed_Gtid_Set
mysql-bin.000001	592	user_db	mysql,information_schema,performance_schema	

4.设置从库向主库同步数据、并检查链路

```

1 #先停止同步
2 STOP SLAVE;
3 #修改从库指向主库，使用上一步记录的文件名以及位点
4 CHANGE MASTER TO
5     master_host='localhost',
6     master_user='db_sync',
7     master_password='db_sync',
8     master_log_file='mysql-bin.000001',
9     master_log_pos=592;
10 #启动同步
11 START SLAVE;
12 #查看从库状态Slave_IO_Running和Slave_SQL_Running都为yes说明同步成功，如果不为yes,请
    检查error_Log，然后排查相关异常
13 SHOW SLAVE STATUS

```

最后测试在主库修改数据库，看从库是否能够同步成功。

在主库中修改user_db数据库表中的字段值，看从库中表中的相应字段值是否改变，如果改变则同步成功，否则失败。

C.Sharding-jdbc读写分离:

写入主库，读取从库，主库和从库会进行主从复制操作

1、分片规则修改

```

1 ##定义数据源
2 spring.shardingsphere.datasource.names=m0,m1,m2,s0
3
4 #数据源s0
5 spring.shardingsphere.datasource.s0.type=com.alibaba.druid.pool.DruidDataSource
6 spring.shardingsphere.datasource.s0.driver-class-name=com.mysql.jdbc.Driver
7 spring.shardingsphere.datasource.s0.url=jdbc:mysql://localhost:3307/user_db?
    useUnicode=true
8 spring.shardingsphere.datasource.s0.username=root
9 spring.shardingsphere.datasource.s0.password=root
10
11 #数据源0
12 spring.shardingsphere.datasource.m0.type=com.alibaba.druid.pool.DruidDataSource
13 spring.shardingsphere.datasource.m0.driver-class-name=com.mysql.jdbc.Driver

```

```

14 spring.shardingsphere.datasource.m0.url=jdbc:mysql://localhost:3306/user_db?
   useUnicode=true
15 spring.shardingsphere.datasource.m0.username=root
16 spring.shardingsphere.datasource.m0.password=root
17 .....
18 #主库从库逻辑数据源定义ds0为user_db, 指定谁是主库, 谁是从库
19 spring.shardingsphere.sharding.master-slave-
   rules.ds0.masterDataSourceName=m0
20 spring.shardingsphere.sharding.master-slave-
   rules.ds0.slaveDataSourceNames=s0
21 .....
22 #t_user分表策略
23 #spring.shardingsphere.sharding.tables.t_user.actualDataNodes=m0.t_user
24 #配置指定主库还是从库中的t_user表
25 spring.shardingsphere.sharding.tables.t_user.actualDataNodes=ds0.t_user
26 spring.shardingsphere.sharding.tables.t_user.tableStrategy.inline.shardingCo
   lumn=user_id
27 spring.shardingsphere.sharding.tables.t_user.tableStrategy.inline.algorithmE
   xpression=t_user
28 spring.shardingsphere.sharding.tables.t_user.keyGenerator.column=user_id
29 spring.shardingsphere.sharding.tables.t_user.keyGenerator.type=SNOWFLAKE

```

2、测试代码

```

1 //-----垂直分库&读写分离
2 @Autowired
3 private UserMapper userMapper;
4 //测试新增
5 @Test
6 public void testInsertUser(){
7     for (int i = 10; i < 14; i++) {
8         userMapper.insertUser("张三"+i,"");
9     }
10 }
11 //根据userId查询多个订单
12 @Test
13 public void testSelectUsers(){
14     List<Long> userIds=new ArrayList<>();
15     userIds.add(491956648743534593L);
16     userIds.add(491956649020358656L);
17     List<Map> maps = userMapper.selectListByOrderIds(userIds);
18     System.out.println(maps);
19 }

```

3、sql日志打印:

插入数据, 到主库中

```

: ==> Preparing: insert into t_user(fullname,user_type)values(?,?)
: ==> Parameters: 张三13(String), (String)
: Rule Type: sharding
: Logic SQL: insert into t_user(fullname,user_type)values(?,?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_user, alias=Optional
: Actual SQL: m0 ::: insert into t_user (fullname, user_type, user_id) VALUES (?, ?, ?) ::: [张三13, , 491956649020358656]
: <== Updates: 1

```

通过日志可以看出, 所有写操作落入m0数据源, 主库中

当插入主库数据之后, mysql配置的主从复制功能就会马上把主库中的新数据同步到从库中表中

读取数据，从库中

```
.cationTests : Started ShardingQuickApplicationTests in 3.998 seconds (JVM running for 4.726)
lyOrderIds : ==> Preparing: select * from t_user t where t.user_id in ( ?, ? )
lyOrderIds : ==> Parameters: 491956648743534593(Long), 491956649020358656(Long)
: Rule Type: sharding
: Logic SQL: select * from t_user t where t.user_id in ( ?, ? )
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_user, alias=0pti
: Actual SQL: s0 :: select * from t_user t where t.user_id in ( ?, ? ) :: [491956648743534593, 491956649020358656]
lyOrderIds : <== Total: 2
56649020358656, fullname=张三13] ]
DataSource : {dataSource-1} closing ...
```

通过日志可以看出，所有写操作落入s0数据源，从库中，达到目标。

6)、sharding-jdbc实战应用

A.需求说明

在电商项目中，每个订单项中都有订单id,订单价格，订单状态，而且该订单还必须绑定用户，不同的用户所包含的订单不一样。

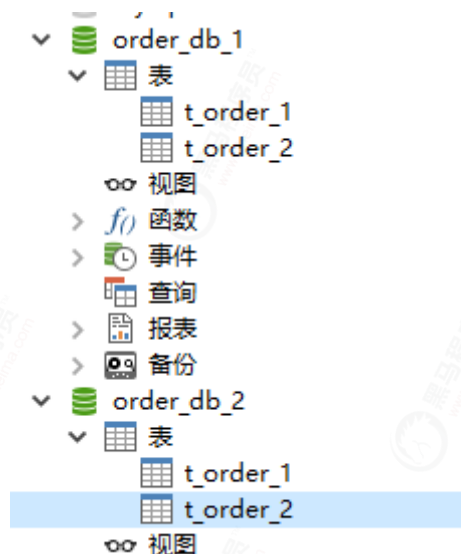
2020-06-14 订单号: 1061060770116435075		可孚医疗器...	和我联系	📄 📄 📄			
	可孚医用酒精喷雾医疗皮肤便携式75度消毒免洗消毒液 随身装洗手液 [交易快照] 颜色分类: 【限时买1送1】 100ml圆瓶装 共2瓶	¥49.99 ¥9.90	1	申请售后 运费险已出单	¥83.90 (含运费: ¥0.00) 手机订	交易成功 订单详情 查看物流	再次购买 申请开票 追加评论
		可孚医用口罩一次性器械旗舰店医护防护口罩50 只装现货口罩 [交易快照] 颜色分类: 医用口罩100只 (10只/袋)	¥199.00 ¥96.00	1			

本案例实现的功能如下：

- 添加订单
- 订单分页查询

B.数据库设计

在本地数据库中创建order_db_1和order_db_2,并且导入百万级表数据



db_order_1				db_order_2			
t_order_1		t_order_2		t_order_1		t_order_2	
order_id	price	user_id	status	order_id	price	user_id	status
490857325872021504	10	2	success1	49085732535150593	5	2	success0
490857326262091776	20	2	success3	490857326064959489	15	2	success2
490857326639579136	30	2	success5	490857326446641153	25	2	success4
490857327092503968	40	2	success7	490857326836711425	35	2	success6
49085732746828544	50	2	success9	490857327302279169	45	2	success8
490857327897870336	60	2	success11	490857327675572225	55	2	success10
490857328275357696	70	2	success13	490857328086614017	65	2	success12
490857328657039360	80	2	success15	490857328459907073	75	2	success14
490857329055498240	90	2	success17	490857328670948865	85	2	success16
490857329432985600	100	2	success19	490857329244241921	95	2	success18
490857329814667264	110	2	success21	490857329621729281	105	2	success20
490857330221514752	120	2	success23	490857330028576769	115	2	success22
490857330607390720	130	2	success25	490857330418647041	125	2	success24
49085733097460992	140	2	success27	490857330791940097	135	2	success26
490857331416891392	150	2	success29	490857331215564801	145	2	success28
490857331802767360	160	2	success31	490857331601440769	155	2	success30
490857332230586368	170	2	success33	490857332016676865	165	2	success32
490857332624850944	180	2	success35	490857332440301569	175	2	success34
49085733298144000	190	2	success37	490857332862850944	185	2	success36
490857333371437056	200	2	success39	4908573332998144000	195	2	success38
490857333769895936	210	2	success41	49085733371437056	200	2	success40
490857334143188992	220	2	success43	490857333769895936	210	2	success42
490857334520676352	230	2	success45	490857334143188992	220	2	success44
490857334898163712	240	2	success47	490857334520676352	230	2	success46
490857335279845376	250	2	success49	490857334898163712	240	2	success48
49085733569744256	260	2	success51	490857335279845376	250	2	success50
49085733614980352	270	2	success53	49085733569744256	260	2	success52
49085733657301978260	280	2	success55	49085733614980352	270	2	success54
				49085733657301978260	280	2	success56

说明，本案例中操作的是订单库，订单表

- 对db_order数据库进行了水平分库操作，分片键是user_id，分片策略是{user_id % 2 + 1};
- 对t_order表进行水平分表，分片键为order_id,分片策略是{t_order % 2 + 1 }。
- 为避免主键冲突，ID生成策略采用雪花算法来生成全局唯一ID
- 使用读取分离来提高性能，可用性。

C.主从同步

参考读写分离章节，对以下库进行主从同步配置：重点

主库：

```

1 # 设置需要同步的数据库
2 binlog-do-db=order_db_1
3 binlog-do-db=order_db_2

```

从库：

```

1 #设置需要同步的数据库
2 replicate_wild_do_table=order_db_1.%
3 replicate_wild_do_table=order_db_2.%

```

D.实现步骤

1、搭建maven工程

pom.xml

```

1 <parent>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-parent</artifactId>
4     <version>2.1.0.RELEASE</version>
5     <relativePath/> <!-- lookup parent from repository -->
6 </parent>
7 <modelVersion>4.0.0</modelVersion>
8
9 <artifactId>sharding_order</artifactId>
10 <properties>
11     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

```

```
12     <project.reporting.outputEncoding>UTF-  
13 8</project.reporting.outputEncoding>  
14     <java.version>1.8</java.version>  
15 </properties>  
16 <dependencies>  
17     <dependency>  
18         <groupId>org.springframework.boot</groupId>  
19         <artifactId>spring-boot-starter-web</artifactId>  
20         <version>2.1.0.RELEASE</version>  
21     </dependency>  
22  
23     <dependency>  
24         <groupId>org.projectlombok</groupId>  
25         <artifactId>lombok</artifactId>  
26         <version>1.18.0</version>  
27     </dependency>  
28  
29  
30     <dependency>  
31         <groupId>javax.interceptor</groupId>  
32         <artifactId>javax.interceptor-api</artifactId>  
33         <version>1.2</version>  
34     </dependency>  
35  
36     <dependency>  
37         <groupId>mysql</groupId>  
38         <artifactId>mysql-connector-java</artifactId>  
39         <version>5.1.47</version>  
40     </dependency>  
41  
42     <dependency>  
43         <groupId>org.mybatis.spring.boot</groupId>  
44         <artifactId>mybatis-spring-boot-starter</artifactId>  
45         <version>2.1.0</version>  
46     </dependency>  
47  
48     <dependency>  
49         <groupId>com.alibaba</groupId>  
50         <artifactId>druid-spring-boot-starter</artifactId>  
51         <version>1.1.16</version>  
52     </dependency>  
53     <dependency>  
54         <groupId>org.apache.shardingsphere</groupId>  
55         <artifactId>sharding-jdbc-spring-boot-starter</artifactId>  
56         <version>4.0.0-RC1</version>  
57     </dependency>  
58  
59     <dependency>  
60         <groupId>com.baomidou</groupId>  
61         <artifactId>mybatis-plus-boot-starter</artifactId>  
62         <version>3.1.0</version>  
63     </dependency>  
64     <dependency>  
65         <groupId>com.baomidou</groupId>  
66         <artifactId>mybatis-plus-generator</artifactId>  
67         <version>3.1.0</version>  
68     </dependency>
```



```

69     <dependency>
70         <groupId>org.mybatis</groupId>
71         <artifactId>mybatis-typehandlers-jsr310</artifactId>
72         <version>1.0.2</version>
73     </dependency>
74     <dependency>
75         <groupId>org.springframework.boot</groupId>
76         <artifactId>spring-boot-starter-test</artifactId>
77         <scope>test</scope>
78         <version>2.1.0.RELEASE</version>
79     </dependency>
80
81 </dependencies>

```

2、分片规则配置

application.properties

```

1  #端口号
2  server.port=561000
3  #实例名称
4  spring.application.name= sharding_order
5  #表示后发现的bean会覆盖之前相同名称的bean
6  spring.main.allow-bean-definition-overriding=true
7  #该配置项就是指将带有下划线的表字段映射为驼峰格式的实体类属性
8  mybatis.configuration.map-underscore-to-camel-case=true
9  #以下是分片规则配置
10
11  ##定义数据源
12  spring.shardingsphere.datasource.names=m1,m2,s1,s2
13  #-----
14  #定义db_order_1数据源主库
15  spring.shardingsphere.datasource.m1.type=com.alibaba.druid.pool.DruidDataSou
16  rce
17  spring.shardingsphere.datasource.m1.driver-class-name =com.mysql.jdbc.Driver
18  spring.shardingsphere.datasource.m1.url=jdbc:mysql://localhost:3306/order_db
19  _1?useUnicode=true
20  spring.shardingsphere.datasource.m1.username=root
21  spring.shardingsphere.datasource.m1.password=root
22
23  #定义db_order_1数据源从库
24  spring.shardingsphere.datasource.s1.type=com.alibaba.druid.pool.DruidDataSou
25  rce
26  spring.shardingsphere.datasource.s1.driver-class-name =com.mysql.jdbc.Driver
27  spring.shardingsphere.datasource.s1.url=jdbc:mysql://localhost:3307/order_db
28  _1?useUnicode=true
29  spring.shardingsphere.datasource.s1.username=root
30  spring.shardingsphere.datasource.s1.password=root
31
32  #读写分离，主库和从库绑定
33  spring.shardingsphere.sharding.master-slave-
34  rules.ds1.masterDataSourceName=m1
35  spring.shardingsphere.sharding.master-slave-
36  rules.ds1.slaveDataSourceNames=s1
37
38  #-----
39  -----

```

```
33
34 #定义db_order_2数据源主库
35 spring.shardingsphere.datasource.m2.type=com.alibaba.druid.pool.DruidDataSource
36 spring.shardingsphere.datasource.m2.driver-class-name=com.mysql.jdbc.Driver
37 spring.shardingsphere.datasource.m2.url=jdbc:mysql://localhost:3306/order_db
38 spring.shardingsphere.datasource.m2.username=root
39 spring.shardingsphere.datasource.m2.password=root
40
41 #定义db_order_2数据源从库
42 spring.shardingsphere.datasource.s2.type=com.alibaba.druid.pool.DruidDataSource
43 spring.shardingsphere.datasource.s2.driver-class-name=com.mysql.jdbc.Driver
44 spring.shardingsphere.datasource.s2.url=jdbc:mysql://localhost:3307/order_db
45 spring.shardingsphere.datasource.s2.username=root
46 spring.shardingsphere.datasource.s2.password=root
47
48 #读写分离，主库和从库绑定
49 spring.shardingsphere.sharding.master-slave-
50 rules.ds2.masterDataSourceName=m2
51 spring.shardingsphere.sharding.master-slave-
52 rules.ds2.slaveDataSourceNames=s2
53
54 #指定t_order表的数据分布情况，配置数据节点
55 #spring.shardingsphere.sharding.tables.t_order.actualDataNodes=m$->
56 {1..2}.t_order_$->{1..2}
57 spring.shardingsphere.sharding.tables.t_order.actualDataNodes=ds$->
58 {1..2}.t_order_$->{1..2}
59
60 #指定t_order表的主键生成策略为SNOWFLAKE
61 spring.shardingsphere.sharding.tables.t_order.keyGenerator.column=order_id
62 spring.shardingsphere.sharding.tables.t_order.keyGenerator.type=SNOWFLAKE
63
64 #t_order分表策略，指定t_order表的分片策略，分片策略包括分片键和分片算法
65 spring.shardingsphere.sharding.tables.t_order.tableStrategy.inline.shardingC
66 olumn=order_id
67 spring.shardingsphere.sharding.tables.t_order.tableStrategy.inline.algorithm
68 Expression=t_order_$->{order_id % 2+1}
69
70 #分库策略，以user_id为分片键，分片策略为user_id%2+1，user_id为偶数操作m1数据源，否则
71 操作m2。
72 spring.shardingsphere.sharding.tables.t_order.databaseStrategy.inline.shardi
73 ngColumn=user_id
74 spring.shardingsphere.sharding.tables.t_order.databaseStrategy.inline.algori
75 thmExpression=ds$->{user_id % 2+1 }
76
77 #打开sql输出日志
78 spring.shardingsphere.props.sql.show=true
79
80 #日志级别
81 logging.level.root=info
82 logging.level.org.springframework.web=info
83 logging.level.com.itheima=debug
84 logging.level.druid.sql=debug
```

3、创建order实体

```
1
2  /**
3   * 订单实体
4   */
5  @Data
6  public class Order {
7
8      private Long orderId;
9      private BigDecimal price;
10     private Long userId;
11     private String status;
12
13 }
```

4、插入订单

新建orderMapper接口

```
1  /**
2   * 订单mapper接口
3   */
4  @Mapper
5  public interface OrderMapper {
6
7      //插入订单
8      @Insert("insert into t_order(price,user_id,status) values(#{price},#{
9          {userId},#{status})")
10     public int insertOrder(Order order);
11 }
```

测试代码

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  public class ShardingOrderApplicationTest {
4
5      @Autowired
6      private OrderMapper orderMapper;
7      //新增订单
8      @Test
9      public void insertTest(){
10         for (int i = 5000001; i < 5000010; i++) {
11             Order order=new Order();
12             order.setPrice(new BigDecimal((i+1)*5));
13             order.setStatus("wait pay"+"|"+i);
14             order.setUserId(1L);
15             orderMapper.insertOrder(order);
16         }
17     }
18 }
```

sql日志打印

```

: ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
: ==> Parameters: 25000045(BigDecimal), 1(Long), wait pay|5000008(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent())]))
: Actual SQL: m2::: insert into t_order_1 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [25000045, 1, wait pay|5000008, 492043360161234944]
: <== Updates: 1
: ==> Preparing: insert into t_order(price,user_id,status) values(?, ?, ?)
: ==> Parameters: 25000050(BigDecimal), 1(Long), wait pay|5000009(String)
: Rule Type: sharding
: Logic SQL: insert into t_order(price,user_id,status) values(?, ?, ?)
: SQLStatement: InsertStatement(super=DMLStatement(super=AbstractSQLStatement(type=DML, tables=Tables(tables=[Table(name=t_order, alias=Optional.absent())]))
: Actual SQL: m2::: insert into t_order_2 (price, user_id, status, order_id) VALUES (?, ?, ?, ?) ::: [25000050, 1, wait pay|5000009, 492043360190595073]
: <== Updates: 1

```

根据日志来看，数据插入的是m2数据源，这是因为user_id为分片键进行分库操作，然后根据order_id作为分片键进行分表操作的。

在分片规则中已经配置了主从库的绑定，从日志上看所有的插入写的操作均插入到主库中了，体现了读写分离的特性。

查看主库和从库数据表截图

查看主库中的order_db_2数据库中的表数据和查看从库中的order_db_2数据库中的表数据是否同步，如果同步表示主从同步ok，如果有问题没有同步，则参考读写分离篇章操作。

5、分页查询订单

新建orderMapper接口

```

1 //分页查询订单
2 @Select("select * from t_order where user_id=#{userId} order by order_id
   desc limit #{start},#{size}")
3 public List<Order> selectByPage(@Param("userId") Long
   userId,@Param("start") Integer start,@Param("size") Integer size);

```

测试代码

```

1 //分页查询订单
2 @Test
3 public void testSelectByPage(){
4     Long userId=1L;
5     Integer page=2;
6     Integer size=10;
7     Integer start=(page-1)*size;
8     List<Order> orders = orderMapper.selectByPage(userId, start, size);
9     System.out.println(orders);
10 }

```

sql日志打印

```

: Started ShardingOrderApplicationTest in 3.517 seconds (JVM running for 4.314)
: ==> Preparing: select * from t_order where user_id=? order by order_id desc limit ?,?
: ==> Parameters: 1(Long), 10(Integer), 10(Integer)
: Rule Type: sharding
: Logic SQL: select * from t_order where user_id=? order by order_id desc limit ?,?
: SQLStatement: SelectStatement(super=DQLStatement(super=AbstractSQLStatement(type=DQL, tables=Tables(tables=[Table(name=t_order, alias=
: Actual SQL: s2 ::: select * from t_order_1 where user_id=? order by order_id desc limit ?,? ::: [1, 0, 20]
: Actual SQL: s2 ::: select * from t_order_2 where user_id=? order by order_id desc limit ?,? ::: [1, 0, 20]
: <== Total: 10
rId=491486232572854273, price=4999990.00, userId=1, status=success|999997), Order(orderId=491486232375721984, price=4999985.00, userId=1,

```

通过日志来看，查询读操作，路由到了从库中，体现了读写分离的特性，达到了目标。但是page=2的时候，sharding-jdbc解析sql还是limit 0,20? 但是最终sql归并结果是正确的。

说明:

分页查询是业务中最常见的场景，Sharding-jdbc支持常用关系数据库的分页查询，不过Sharding-jdbc的分页功能比较容易让使用者误解，用户通常认为分页归并会占用大量内存。在分布式的场景中，将LIMIT 10000000, 10改写为LIMIT 0, 10000010，才能保证其数据的正确性。用户非常容易产生ShardingSphere会将大量无意义的数据加载至内存中，造成内存溢出风险的错觉。其实大部分情况都通过流式归并获取数据结果集，因此ShardingSphere会通过结果集的next方法将无需取出的数据全部跳过，并不会将其存入内存。

但同时需要注意的是，由于排序的需要，大量的数据仍然需要传输到Sharding-jdbc的内存空间。因此，采用LIMIT这种方式分页，并非最佳实践。由于LIMIT并不能通过索引查询数据，因此如果可以保证ID的连续性，通过ID进行分页是比较好的解决方案，例如：

```
1 | SELECT * FROM t_order WHERE id > 100000 AND id <= 100010 ORDER BY id;
```

排序功能是由Sharding-jdbc的排序归并来完成，由于在SQL中存在ORDER BY语句，因此每个数据结果集自身是有序的，因此只需要将数据结果集当前游标指向的数据值进行排序即可。这相当于对多个有序的数组进行排序，归并排序是最适合此场景的排序算法。

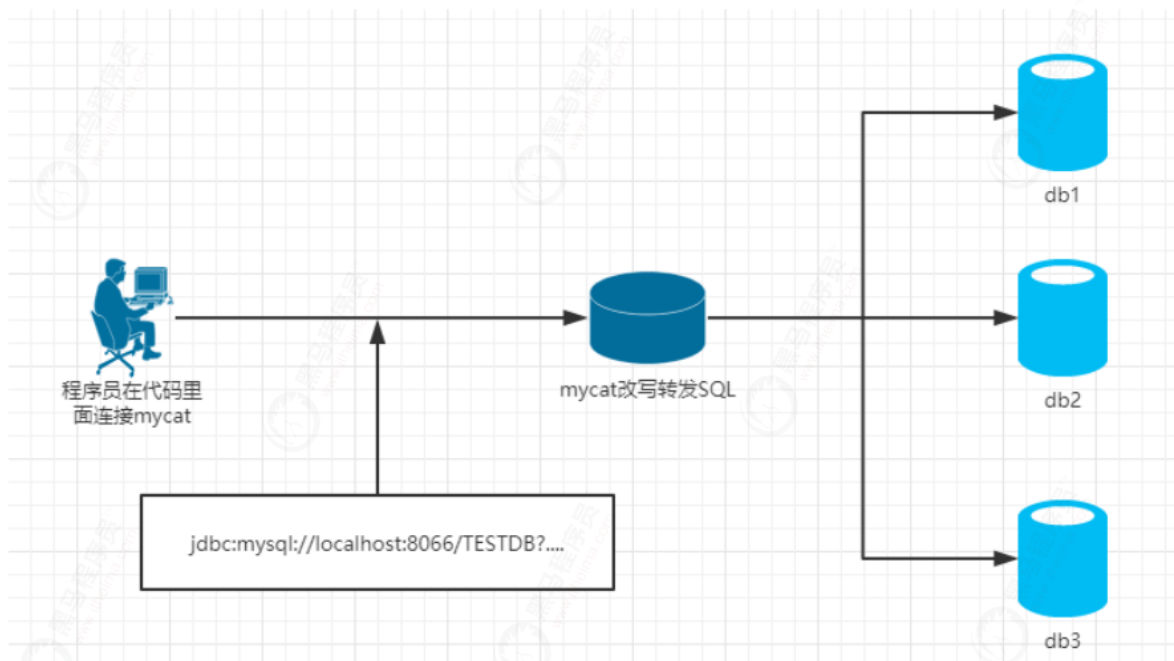
2、MyCAT中间件:

说明，MyCAT中间件，本次不作为重点讲解，如果有兴趣的话可以查看《Mycat权威指南-全部-完整版.pdf》

3、Sharding-JDBC与MyCAT的区别:

- mycat 是一个基于第三方应用中间件的数据库代理框架，客户端所有的jdbc请求都必须要先交给mycat，再由mycat转发到具体的真实服务器中。sharding-jdbc是一个jar形式，在本地应用层重写的jdbc原生的方法，实现数据库分片形式。
- mycat 属于服务器端的数据库中间件，而sharding-jdbc是一个本地数据库中间件框架。
- 从设计理念上看确实有一定的相似性。主要流程都是SQL解析->SQL路由->SQL改写->SQL执行->结果归并。但架构设计上是不同的。mycat是基于Proxy，它复写了Mycat协议，将Mycat server伪装成一个mycat数据库；而sharding-jdbc是基于jdbc的扩展是以jar包的形式提供轻量级服务的。
- 总结：mycat 与 sharding-jdbc 类似于 nginx 与springcloud ribbon

Mycat(proxy中间件层):



Sharding-jdbc(TDDL为代表的上层应用):

