分库分表

1 概述

1.1 为什么分库分表

在P2P平台中，标的信息和投标信息做为平台基础业务数据存在。随着平台的发展，这些数据可能会越来越多，甚至达到亿级。以MySQL为例，单库数据量在5000万以内性能比较好，超过阈值后性能会随着数据量的增大而明显降低。单表的数据量超过1000w，性能也会下降严重。这就会导致查询一次所花的时间变长，并发操作达到一定量时可能会卡死，甚至把系统给拖垮，因此我们的P2P平台需要解决这个性能瓶颈问题。

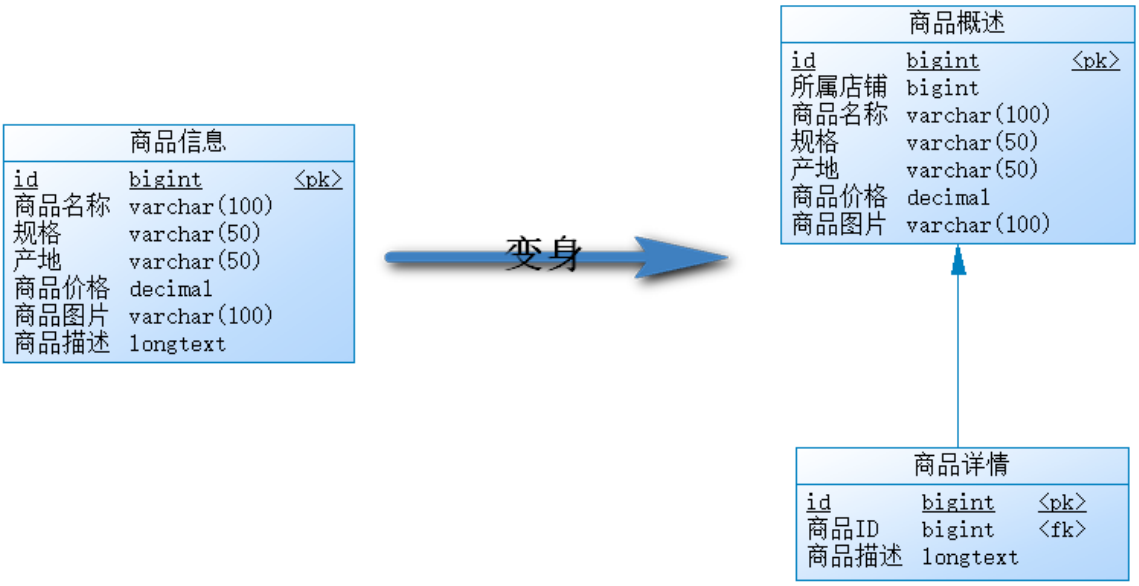
我们是否可以通过提升服务器硬件能力来提高数据处理能力？能，但是这种方案很贵，并且提高硬件是有上限的。**那我们能不能把数据分散在不同的数据库中，使得单一数据库和表的数据量变小，从而达到提升数据库操作性能的目的？** 可以，这就是数据库分库分表。

分库分表就是把较大的数据库和数据表按照某种策略进行拆分。目的在于：降低每个库、每张表的数据量，减小数据库的负担，提高数据库的效率，缩短查询时间。另外，因为分库分表这种改造是可控的，底层还是基于RDBMS，因此整个数据库的运维体系以及相关基础设施都是可重用的。

1.2 分库分表的方式

1.2.1 垂直分表

用户在电商平台流览商品时，首先看到的是商品的基本信息，如果对该商品感兴趣时才会继续查看该商品的详细描述。因此，商品基本信息的访问频次要高于商品详细描述信息，商品基本信息的访问效率要高于商品详细描述信息(大字段)。 由于这两种数据的特性不一样，因此考虑将商品信息表拆分如下：



这种拆分就叫**垂直分表。垂直分表定义：将一个表的字段分散到多个表中，每个表存储其中一部分字段。**垂直分表带来的提升是：

1. 减少IO争抢，减少锁表的几率，查看商品详情的与商品概述互不影响
2. 充分发挥高频数据的操作效率，对商品概述数据操作的高效率不会被操作商品详情数据的低效率所拖累。

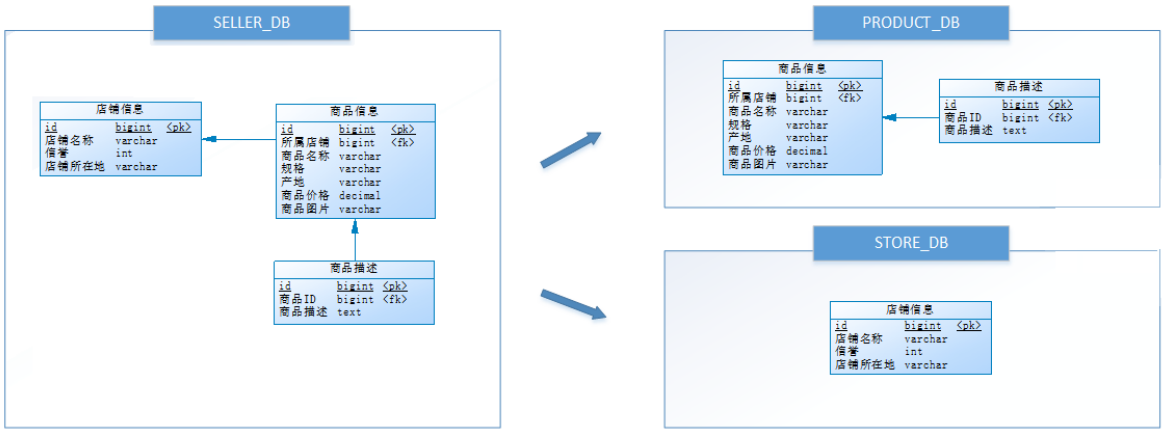
一般来说，某业务实体中的各个数据项的访问频次是不一样的，部分数据项可能是占用存储空间比较大的BLOB或是TEXT，例如上例中的**商品描述字段**。所以，当数据量很大时，可以**将表按字段拆分，将热门字段、冷门字段分开放置在不同表中**。垂直切分带来的性能提升，主要集中在热门数据的操作效率上，而且磁盘争用情况减少。通常我们按以下原则进行垂直拆分:

- 把不常用的字段单独放在一张表
- 把text，blob等大字段拆分出来单独放在一张表
- 经常组合查询的字段单独放在一张表中

1.2.2 垂直分库

通过垂直分表，数据库性能得到了一定程度的提升，但是还没有达到要求，并且磁盘空间也快不够了，因为数据还是始终存放在一台服务器。库内垂直分表只解决了单一表数据量过大的问题，但没有将表分布到不同机器的库上，因此对于减轻数据库的压力来说，作用有限，大家还是竞争同一个物理机的CPU、内存、网络IO、磁盘。

以电商平台为例，可以把原有的SELLER_DB(卖家库)，拆分为PRODUCT_DB(商品库)和STORE_DB(店铺库)，并把这两个库分散到不同服务器上，如下图所示:



由于**商品信息**与**商品描述**业务耦合度较高，因此一起被存放在PRODUCT_DB(商品库)；而**店铺信息**相对独立，因此单独被存放在STORE_DB(店铺库)，这就叫**垂直分库**。

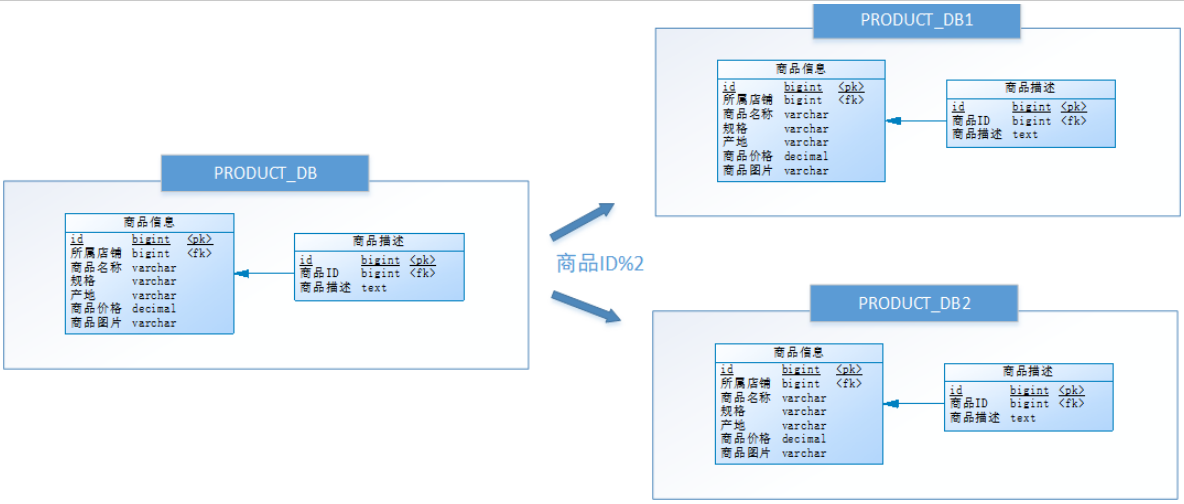
垂直分库是指按照业务将表进行分类，分布到不同的数据库上面，每个库可以放在不同的服务器上，从而达到多个服务器共同分摊压力的效果。垂直分库带来的提升是:

- 解决业务层面的耦合，业务清晰
- 能对不同业务的数据进行分级管理、维护、监控、扩展等
- 高并发场景下，垂直分库在一定程度上可以提升IO、数据库连接数、单机硬件资源的性能

1.2.3 水平分库

经过**垂直分表和垂直分库**后，数据库性能问题就完全解决了？假设某电商平台发展迅猛，PRODUCT_DB(商品库)单库存储数据已经超出预估。假设目前该平台有8w店铺，每个店铺平均有150个不同规格的商品，再算上增长，那商品数量就会达到1500w+级别，并且PRODUCT_DB(商品库)属于访问非常频繁的资源，性能瓶颈再次出现。

能再次垂直分库吗？从业务角度分析，目前已经无法再次垂直拆分。于是我们又想了一个办法，判断商品ID是奇数还是偶数，然后把商品信息分别存放到两个数据库中。也就是说，要操作某条数据，先分析这条数据的商品ID，如果商品ID为奇数，将此操作映射至RRODUCT_DB1(商品库1)；如果商品ID为偶数，将操作映射至RRODUCT_DB2(商品库2)，这就叫**水平分库**。



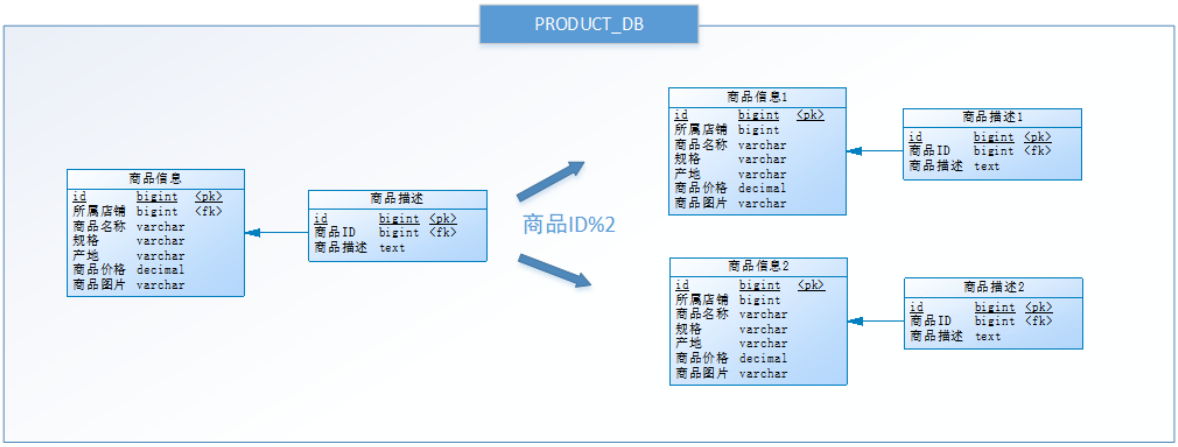
水平分库是把同一个表的数据按一定规则拆分到不同的数据库中，每个库可以放在不同的服务器上。它带来的提升是：

- 解决了单库大数据，高并发的性能瓶颈。
- 按照合理拆分规则拆分，join操作基本避免跨库。
- 提高了系统的稳定性及可用性。

当一个应用难以再细粒度的垂直切分，或切分后数据量行数仍然巨大，存在单库读写、存储性能瓶颈，这时候就需要进行**水平分库**了，经过水平切分的优化，往往能解决单库存储量及性能瓶颈。但由于同一个表被分配在不同的数据库，需要额外进行数据操作的路由工作，因此大大增加了系统复杂度。

1.2.4 水平分表

数据库能水平拆分，那数据表是不是也可以呢？我们尝试把某PRODUCT_DB(商品库)内的表，进行了一次水平拆分：



与水平分库的思路类似，不过这次拆分的目标是表，商品信息及商品描述被分成了两套表。如果商品ID为奇数，将此操作映射至商品信息1表；如果商品ID为偶数，将操作映射至商品信息2表，这就叫**水平分表。水平分表**是在同一个数据库内，把同一个表的数据按一定规则拆分到多个表中。它带来的提升是：

- 优化单一表数据量过大而产生的性能问题
- 避免IO争抢并减少锁表的几率

库内的水平分表，解决了单一表数据量过大的问题，分出来的小表中只包含一部分数据，从而使得单个表的数据量变小，提高检索性能。但由于同一个表的数据被拆分为多张表，也需要额外进行数据操作的路由工作，因此增加了系统复杂度。

1.2.5 小结

- 垂直分表：可以把一个宽表的字段按访问频次、业务耦合松紧、是否是大字段的原则拆分为多个表，这样既能使业务清晰，还能提升部分性能。拆分后，尽量从业务角度避免联查，否则性能方面将得不偿失。
- 垂直分库：可以把多个表按业务耦合松紧归类，分别存放在不同的库，这些库可以分布在不同服务器，从而使访问压力被多服务器负载，大大提升性能，同时能提高整体架构的业务清晰度，不同的业务库可根据自身情况定制优化方案。但是它需要解决跨库带来的所有复杂问题。
- 水平分库：可以把一个表的数据(按数据行)分到多个不同的库，每个库只有这个表的部分数据，这些库可以分布在不同服务器，从而使访问压力被多服务器负载，大大提升性能。它不仅需要解决跨库带来的所有复杂问题，还要解决数据路由的问题。
- 水平分表：可以把一个表的数据(按数据行)分到多个同一个数据库的多张表中，每个表只有这个表的部分数据，这样做能小幅提升性能，它仅仅作为水平分库的一个补充优化。

一般来说，在系统设计阶段就应该根据业务耦合松紧来确定垂直分库，垂直分表方案，在数据量及访问压力不是特别大的情况，首先考虑缓存、读写分离、索引技术等方案。若数据量极大，且持续增长，再考虑水平分库分表方案。

1.3 分库分表带来的问题

分库分表有效的缓解了大数据、高并发带来的性能和压力，也能突破网络IO、硬件资源、连接数的瓶颈，但同时也带来了一些问题。

1.3.1 事务一致性问题

由于分库分表把数据分布在不同库甚至不同服务器，不可避免会带来**分布式事务**问题，我们需要额外编程解决该问题。

1.3.2 跨节点join

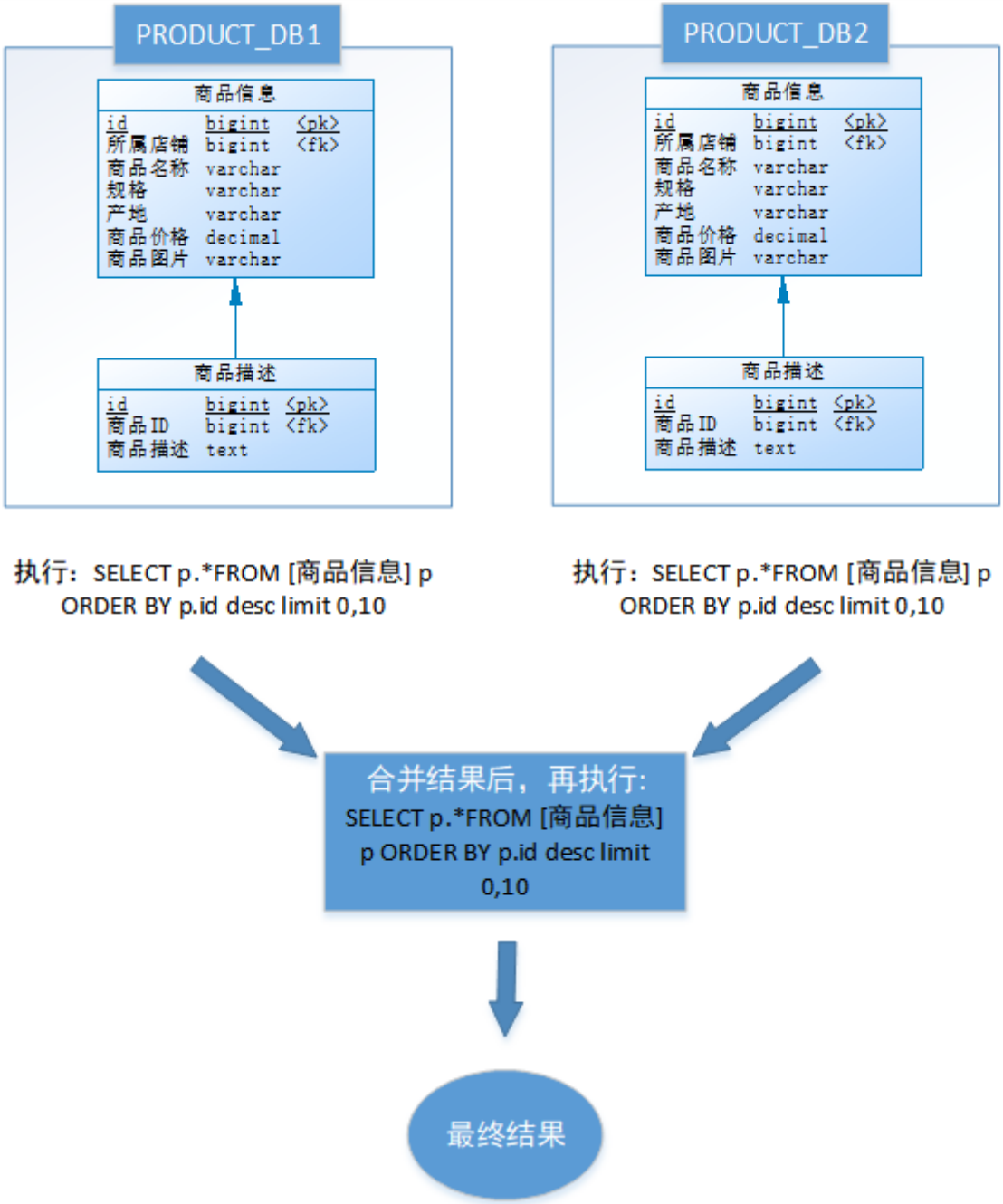
在没有进行分库分表前，我们检索商品时可以通过以下SQL对店铺信息进行关联查询：

```
SELECT p.*,s.[店铺名称],s.[信誉]
FROM [商品信息] p
LEFT JOIN [店铺信息] s ON p.id = s.[所属店铺]
WHERE...ORDER BY...LIMIT...
```

但经过分库分表后，**[商品信息]**和**[店铺信息]**不在一个数据库或一个表中，甚至不在一台服务器上，无法通过sql语句进行关联查询，我们需要额外编程解决该问题。

1.3.3 跨节点分页、排序和聚合函数

跨节点多库进行查询时，limit分页、order by排序以及聚合函数等问题，就变得比较复杂了。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。例如，进行水平分库后的商品库，按ID倒序排序分页，取第一页：

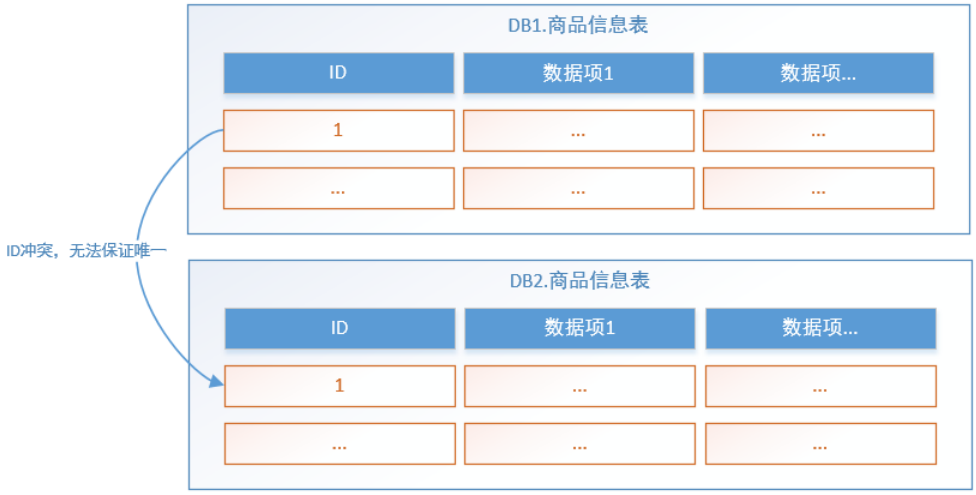


以上流程是取第一页的数据，性能影响不大，但由于商品信息的分布在各数据库的数据可能是随机的，如果是取第N页，需要将所有节点前N页数据都取出来合并，再进行整体的排序，操作效率可想而知，所以请求页数越大，系统的性能也会越差。

在使用Max、Min、Sum、Count之类的函数进行计算的时候，与排序分页同理，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和再次计算，最终将结果返回。

1.3.4 主键避重

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，某个分区数据库生成的ID无法保证全局唯一。因此需要单独设计全局主键，以避免跨库主键重复问题。



由于分库分表之后，数据被分散在不同的服务器、数据库和表中。因此，对数据的操作也就无法通过常规方式完成，并且它还带来了一系列的问题。我们在开发过程中需要通过一些中间件解决这些问题，市面上有很多中间件可供我们选择，其中Sharding-JDBC较为流行。

2 Sharding-JDBC

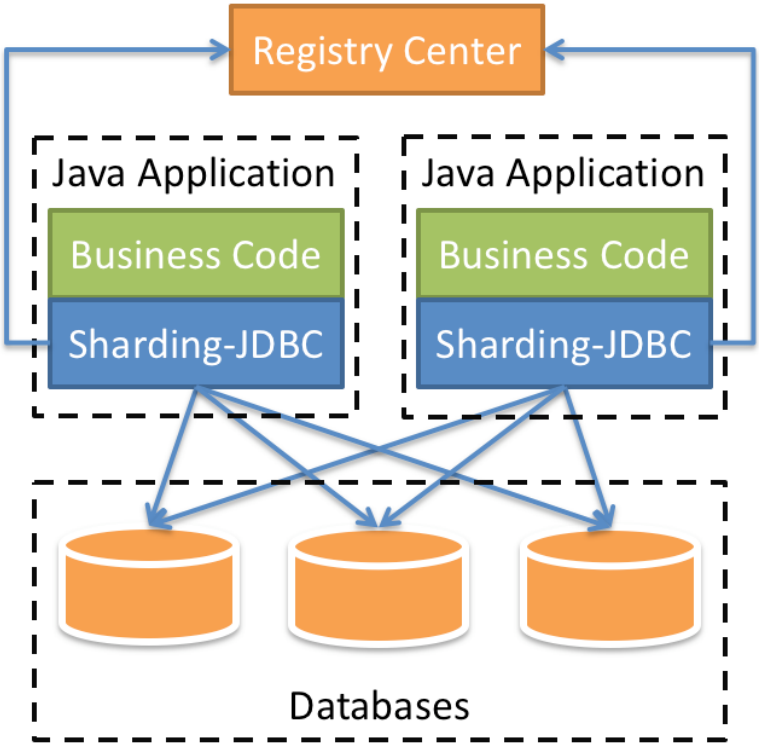
2.1 概览

Sharding-JDBC是当当网研发的开源分布式数据库中间件。从 3.0 开始，Sharding-JDBC更名为Sharding-Sphere，之后该项目进入Apache孵化器，4.0之后的版本为Apache版本。

ShardingSphere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。 它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Java同构、异构语言、容器、云原生等各种多样化的应用场景。

咱们目前只需关注Sharding-JDBC，它定位为轻量级Java框架，在Java的JDBC层提供额外服务。 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Java的ORM框架，如：JPA, Hibernate, Mybatis, Spring JDBC Template或直接使用JDBC。
- 适用于任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, Druid, HikariCP等。
- 适用于任意支持JDBC规范的数据库，如：MySQL，Oracle，SQLServer和PostgreSQL。



2.2 功能介绍

Sharding-JDBC可以进行分库分表，同时又可以解决分库分表带来的问题，它的核心功能是：数据分片和读写分离。

2.2.1 数据分片

数据分片是Sharding-JDBC核心功能，它是指按照某个维度将存放在单一数据库中的数据分散存放至多个数据库或表中，以达到提升性能瓶颈以及可用性的效果。 数据分片的有效手段是对关系型数据库进行分库和分表。在使用Sharding-JDBC进行数据分片前，需要了解以下概念：

- **逻辑表**

水平拆分的数据库（表）的相同逻辑和数据结构表的总称。例：订单数据根据主键尾数拆分为10张表，分别是`t_order_0`到`t_order_9`，他们的逻辑表名为`t_order`。

- **真实表**

在分片的数据库中真实存在的物理表。即上个示例中的`t_order_0`到`t_order_9`。

- **数据节点**

数据分片的最小单元。由数据源名称和数据表组成，例：`ds_0.t_order_0`。

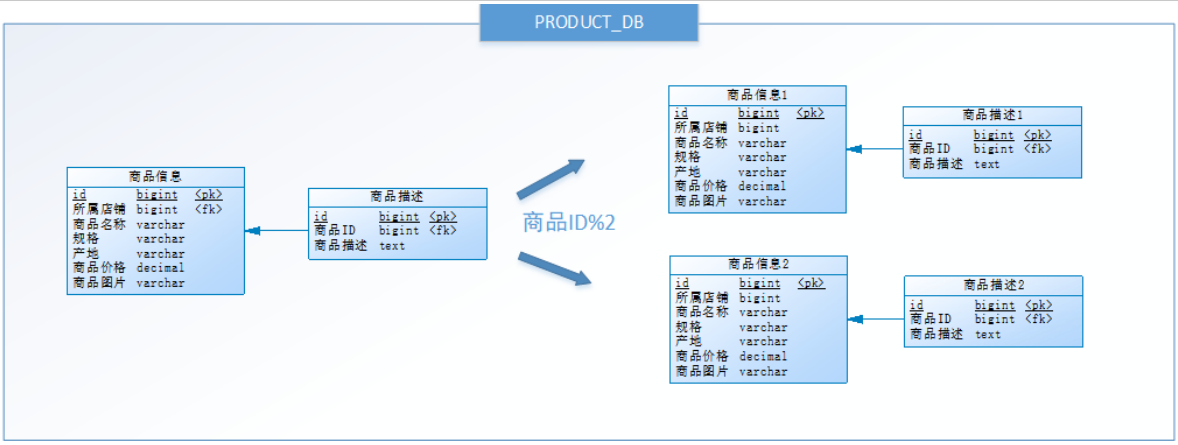
- **分片键**

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。 SQL中如果无分片字段，将执行全路由，性能较差。 除了对单分片字段的支持，ShardingSphere也支持根据多个字段进行分片。

- **自增主键生成策略**

通过在客户端生成自增主键替换以数据库原生自增主键的方式，做到分布式全局主键无重复。

- **绑定表**



指分片规则一致的主表和子表。例如：`商品信息表`表和`商品描述`表，均按照`商品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，关联查询效率将大大提升。以上图为例，如果SQL为：

```
select p1.*,p2.商品描述 from 商品信息 p1 inner join 商品描述  p2 on  p1.id=p2.商品id;
```

在不配置绑定表关系时，那么最终执行的SQL应该为4条，它们呈现为笛卡尔积：

```
select p1.*,p2.商品描述 from 商品信息1 p1 inner join 商品描述1  p2 on  p1.id=p2.商品id

select p1.*,p2.商品描述 from 商品信息2 p1 inner join 商品描述2  p2 on  p1.id=p2.商品id

select p1.*,p2.商品描述 from 商品信息1 p1 inner join 商品描述2  p2 on  p1.id=p2.商品id

select p1.*,p2.商品描述 from 商品信息2 p1 inner join 商品描述1  p2 on  p1.id=p2.商品id
```

在配置绑定表关系后，最终执行的SQL应该为2条：

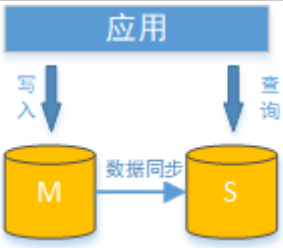
```
select p1.*,p2.商品描述 from 商品信息1 p1 inner join 商品描述1  p2 on  p1.id=p2.商品id

select p1.*,p2.商品描述 from 商品信息2 p1 inner join 商品描述2  p2 on  p1.id=p2.商品id
```

注意：绑定表之间的分片键要完全相同。

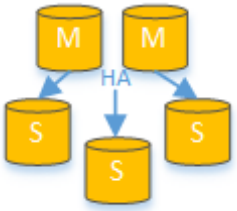
2.2.2 读写分离

面对日益增加的系统访问量以及高并发的情况，数据库的性能面临着巨大瓶颈。 数据库的“写”操作是比较耗时的(例如：写10000条数据到oracle可能要3分钟)，而数据库的“读”操作相对较快(例如：从oracle读10000条数据可能只要5秒钟)。在高并发的情况下，写操作会严重拖累读操作，这是单纯分库分表无法解决的。

我们可以将数据库拆分为主库和从库，主库只负责处理增删改操作，从库只负责处理查询操作，这就是读写分离。它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。

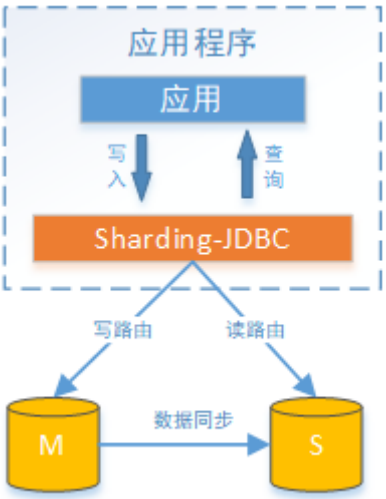


我们还可以搞一主多从，这样就可以将查询请求均匀的分散到多个从库，能够进一步的提升系统的处理能力。 使用多主多从的方式，不但能够提升系统的吞吐量，还能够提升系统的可用性，可以达到在任何一个数据库宕机，甚至磁盘物理损坏的情况下仍然不影响系统的正常运行。



读写分离的数据节点中的数据内容是一致的，所以在采用读写分离时，要注意解决主从数据同步的问题。

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进行读写分离操作。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。



2.3 入门案例

2.3.1 需求描述

使用Sharding-JDBC实现电商平台的商品列表展示，每个列表项中除了包含商品基本信息、商品描述信息之外，还包括了商品所属的店铺信息，如下所示：

Vita Coco唯他可可椰子水饮料进口nfc青椰果汁500ml*24瓶原味正品

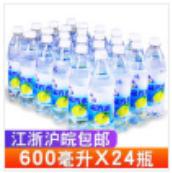
￥298.00 包邮

vitacoco唯他可可旗舰店 北京

绿力冬瓜茶 绿力冬瓜汁饮料310ml *24罐 整箱装冬瓜茶饮料

￥88.80
运费: 18.00

绿力食品旗舰店 上海



上海风味盐汽水柠檬味汽水夏季防暑降温碳酸饮料600ml*24瓶整箱

￥26.80
运费: 8.00

萌大叔食品 上海

沙棘汁野山坡吕梁山西特产饮料整箱生榨果汁纯沙棘汁原浆16瓶装

￥47.88 包邮

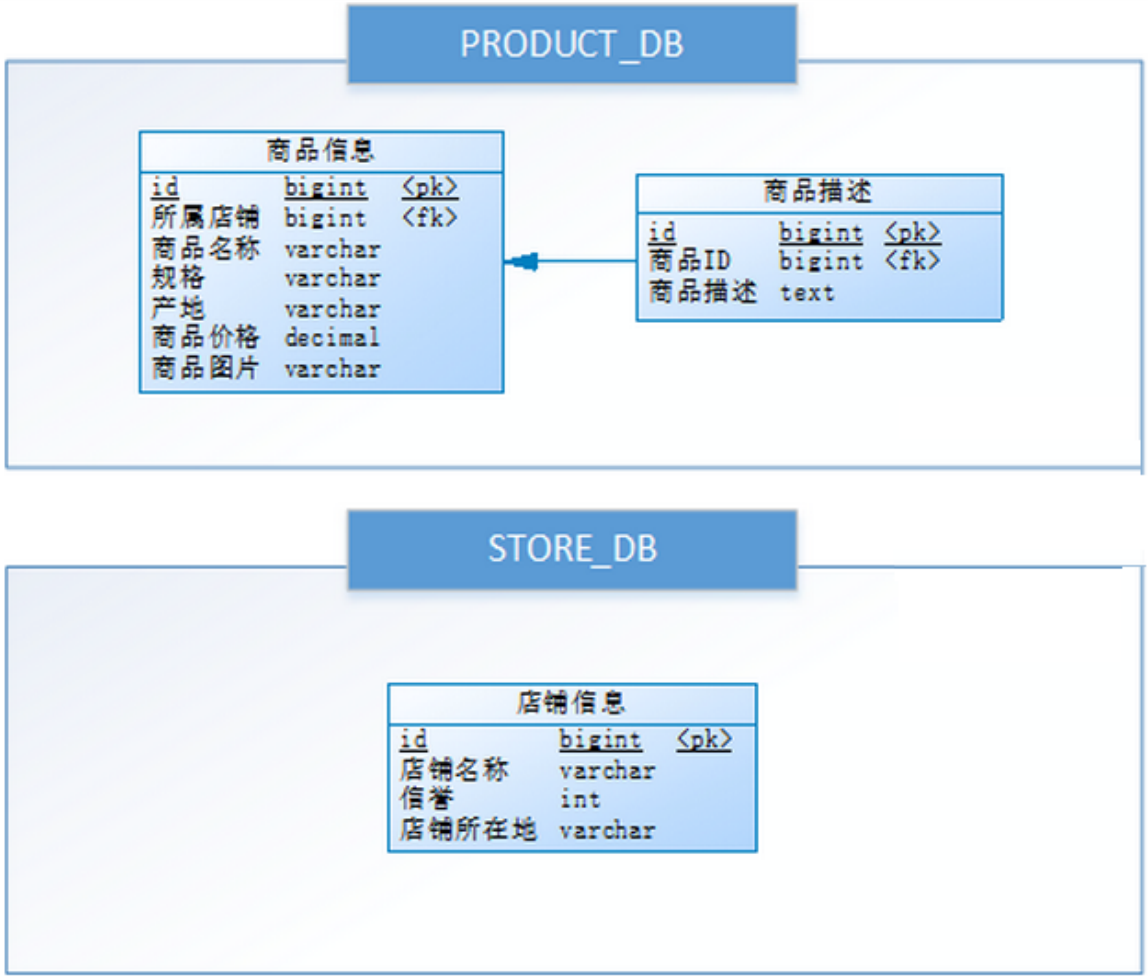
tianjiaoshiye 山西 吕梁

2.3.2 开发环境

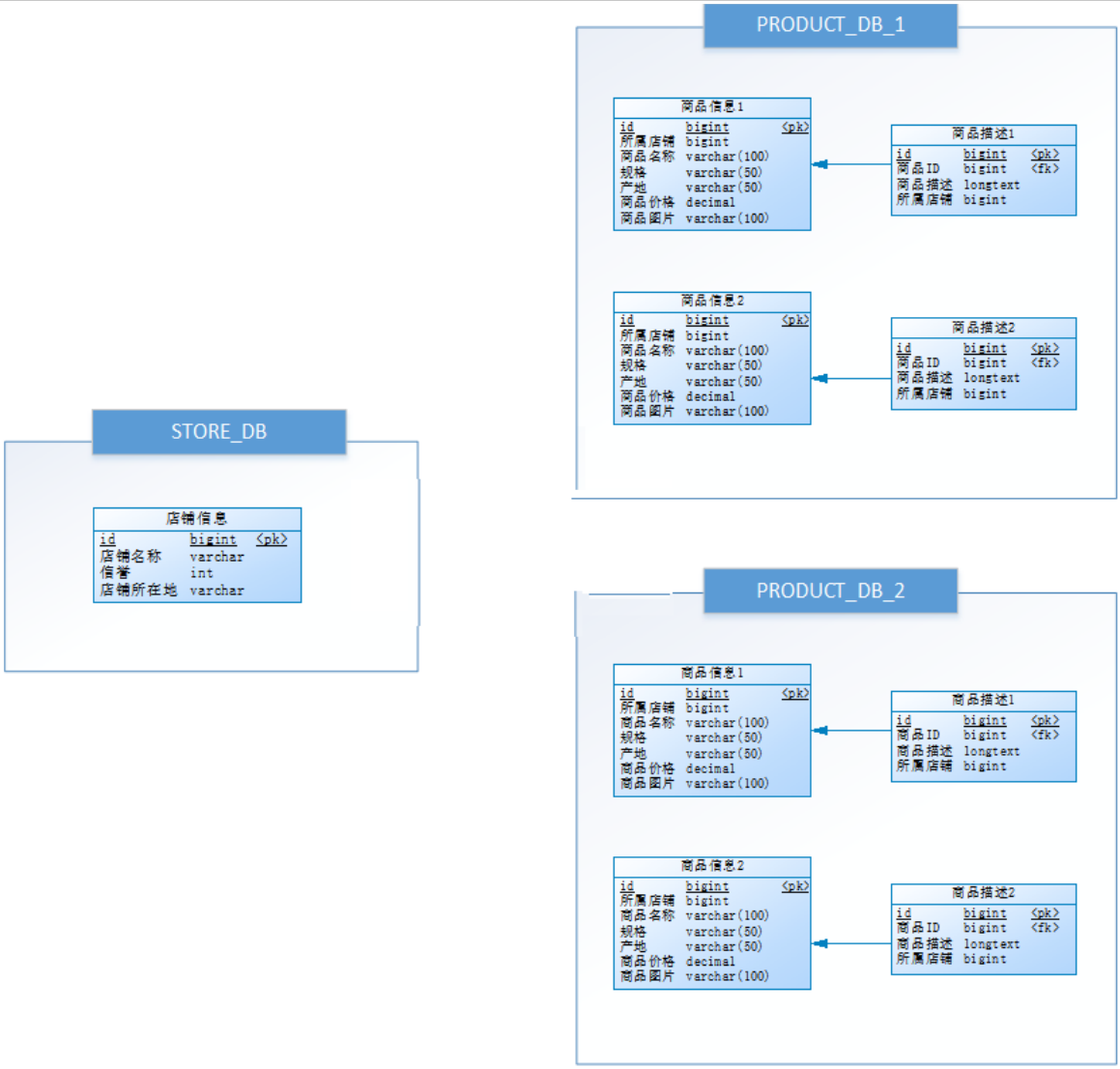
- 数据库：MySQL-5.7.25
- JDK：1.8.0_201
- 应用框架：spring-boot-2.1.3.RELEASE，Mybatis 3.5.0
- Sharding-JDBC：sharding-jdbc-spring-boot-starter-4.0.0-RC1

2.3.3 数据库设计

商品与店铺信息之间进行了**垂直分库**，拆分为了PRODUCT_DB(商品库)和STORE_DB(店铺库)；商品信息还进行了**垂直分表**，拆分为了商品基本信息(store_info)和商品描述信息(product_info)：



考虑到商品信息的数据增长性，对PRODUCT_DB(商品库)进行了**水平分库**，**分片键**使用店铺id，**分片策略**为店铺ID%2 + 1，对商品基本信息(product_info)和商品描述信息(product_descript)进行**水平分表**，**分片键**使用商品id，**分片策略**为商品ID%2 + 1,并将这两个表设置为**绑定表。**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，雪花算法类似于UUID，但是它能生成有序的ID，有利于提高数据库性能。最终数据库设计如下图所示：



2.3.4 搭建数据库环境

2.3.4.1 MySQL主从同步

1. 为了能在一台电脑上(本机)演示出主从架构，复制本机原有mysql一份，例如：复制D:\mysql-5.7.25(作为主库) 到 D:\mysql-5.7.25-s1(作为从库), 修改主、从库的配置文件(my.ini)。

   主库配置：

```
[mysqld]
#开启日志
log-bin = mysql-bin
#设置服务id，主从不能一致
server-id = 1
#设置需要同步的数据库
binlog-do-db=store_db
binlog-do-db=product_db_1
binlog-do-db=product_db_2
#屏蔽系统库同步
binlog-ignore-db=mysql
binlog-ignore-db=information_schema
binlog-ignore-db=performance_schema
```

从库配置：

```
[mysqld]
#设置3307端口
port = 3307
# 设置mysql数据库的数据的存放目录(该目录不一定在mysql安装目录下)
datadir=D:\mysql-5.7.25-s1\data
#开启日志
log-bin = mysql-bin
#设置服务id，主从不能一样
server-id = 2
#设置需要同步的数据库
replicate_wild_do_table=store_db.%
replicate_wild_do_table=product_db_1.%
replicate_wild_do_table=product_db_2.%
#屏蔽系统库同步
replicate_wild_ignore_table=mysql.%
replicate_wild_ignore_table=information_schema.%
replicate_wild_ignore_table=performance_schema.%
```

然后在命令行窗口(以管理员身份运行)中将从库安装为windows服务，注意配置文件位置：

```
D:\mysql-5.7.25-s1\bin> mysqld install mysqls1 --defaults-file="D:\mysql-5.7.25-s1\my.ini"
```

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, 现在重启主库和从库。

请注意，从库数据(data)目录下有个文件auto.cnf，也要与主库不一样，建议直接删除掉，重启服务后将会重新生成。

2. 授权主从复制专用账号

```
#切换至主库bin目录，登录主库
mysql -h localhost -uroot -p123
#授权主从复制专用账号
GRANT REPLICATION SLAVE ON *.* TO 'db_sync'@'%' IDENTIFIED BY 'db_sync';
#刷新权限
FLUSH PRIVILEGES;
#确认位点 记录下文件名以及位点
show master status;
```

```
mysql> show master status;
+------------------+----------+
| File             | Position |
+------------------+----------+
| mysql-bin.000001 |      592 |
+------------------+----------+
1 row in set (0.00 sec)
```

3. 设置从库向主库同步数据、并检查链路

```
#切换至从库bin目录，登录从库
mysql -h localhost -P3307 -uroot -p123
#修改从库指向到主库，使用上一步记录的文件名以及位点
```

```
CHANGE MASTER TO
 master_host = 'localhost',
 master_user = 'db_sync',
 master_password = 'db_sync',
 master_log_file = 'mysql-bin.000001',
 master_log_pos = 592;

#执行该命令前，一定要重启主库和从库服务
show slave status\G
#执行该命令后，确认Slave_IO_Runing以及Slave_SQL_Runing两个状态位是否为“Yes”，如果不为Yes，请检查error_log，然后排查相关异常。

#注意：如果之前此从库已有主库指向，需要先执行以下命令清空
STOP SLAVE IO_THREAD FOR CHANNEL '';
reset slave all;
```

2.3.4.2 初始化数据库

登录并连接主库，然后执行如下脚本：

1. 执行store_db.sql创建store数据库和store_info表
2. 执行product_db_1.sql创建product_db_1数据库和其中的四张表
3. 执行product_db_2.sql创建product_db_2数据库和其中的四张表

此时观察从库，我们会发现从库中已经存在上述数据库和表，说明主从数据同步已经发挥了作用。

2.3.5 功能实现

（1）基础环境搭建

用IDEA打开dbsharding工程，以dbsharding为总体父工程，用来管理依赖，sharding-jdbc-demo为入门案例工程，用来实现功能。在pom文件中引入maven依赖：

```
    <dependencies>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
```

```
            <artifactId>mybatis-spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
        </dependency>

        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>

        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
        </dependency>

    </dependencies>
```

(2) 基本配置

```
server.port=56081

spring.application.name = sharding-jdbc-demo
server.servlet.context-path = /sharding-jdbc-demo
spring.http.encoding.enabled = true
spring.http.encoding.charset = UTF-8
spring.http.encoding.force = true

# 开启swagger
swagger.enable = true
```

```
# 同名bean允许覆盖
spring.main.allow-bean-definition-overriding=true

# 将带有下划线的表字段映射为驼峰格式的实体类属性
mybatis.configuration.map-underscore-to-camel-case = true
```

（3）配置sharding-jdbc

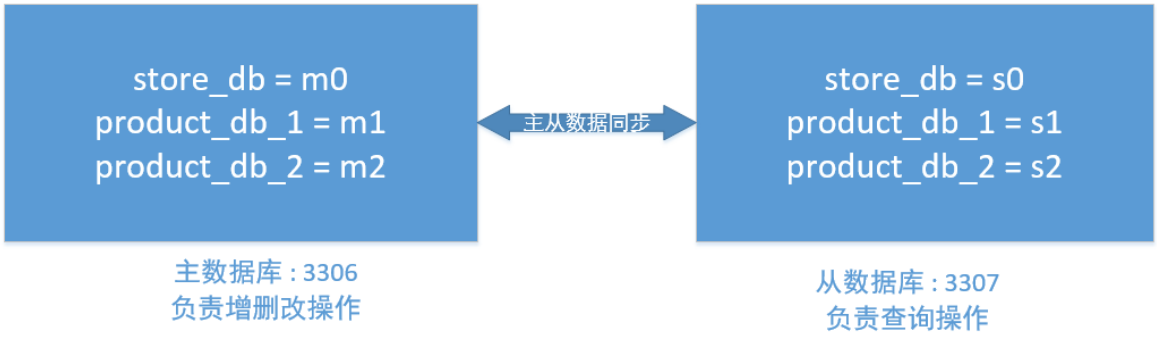
怎么使用sharding-jdbc？在编程中主要就是玩儿配置：

```
# 真实数据源定义
spring.shardingsphere.datasource.names = m0,m1,m2,s0,s1,s2

spring.shardingsphere.datasource.m0.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m0.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m0.url = jdbc:mysql://localhost:3306/store_db?
useUnicode=true
spring.shardingsphere.datasource.m0.username = root
spring.shardingsphere.datasource.m0.password = 123

spring.shardingsphere.datasource.m1.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m1.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m1.url =
jdbc:mysql://localhost:3306/product_db_1?useUnicode=true
spring.shardingsphere.datasource.m1.username = root
spring.shardingsphere.datasource.m1.password = 123

spring.shardingsphere.datasource.m2.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.m2.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.m2.url =
jdbc:mysql://localhost:3306/product_db_2?useUnicode=true
spring.shardingsphere.datasource.m2.username = root
spring.shardingsphere.datasource.m2.password = 123

spring.shardingsphere.datasource.s0.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.s0.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.s0.url = jdbc:mysql://localhost:3307/store_db?
useUnicode=true
spring.shardingsphere.datasource.s0.username = root
spring.shardingsphere.datasource.s0.password = 123

spring.shardingsphere.datasource.s1.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.s1.driver-class-name = com.mysql.jdbc.Driver
spring.shardingsphere.datasource.s1.url =
jdbc:mysql://localhost:3307/product_db_1?useUnicode=true
spring.shardingsphere.datasource.s1.username = root
spring.shardingsphere.datasource.s1.password = 123

spring.shardingsphere.datasource.s2.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.s2.driver-class-name = com.mysql.jdbc.Driver
```

```
spring.shardingsphere.datasource.s2.url =
jdbc:mysql://localhost:3307/product_db_2?useUnicode=true
spring.shardingsphere.datasource.s2.username = root
spring.shardingsphere.datasource.s2.password = 123

# 主库从库逻辑数据源定义 ds0为store_db ds1为product_db_1 ds2为product_db_2
spring.shardingsphere.sharding.master-slave-rules.ds0.master-data-source-name=m0
spring.shardingsphere.sharding.master-slave-rules.ds0.slave-data-source-names=s0

spring.shardingsphere.sharding.master-slave-rules.ds1.master-data-source-name=m1
spring.shardingsphere.sharding.master-slave-rules.ds1.slave-data-source-names=s1

spring.shardingsphere.sharding.master-slave-rules.ds2.master-data-source-name=m2
spring.shardingsphere.sharding.master-slave-rules.ds2.slave-data-source-names=s2


# 分库策略，以store_info_id为分片键，分片策略为store_info_id % 2+1
spring.shardingsphere.sharding.default-database-strategy.inline.sharding-column
= store_info_id
spring.shardingsphere.sharding.default-database-strategy.inline.algorithm-
expression = ds$->{store_info_id % 2+1}

# store_info分表策略，固定分配至ds0的store_info真实表
spring.shardingsphere.sharding.tables.store_info.actual-data-nodes = ds$->
{0}.store_info
spring.shardingsphere.sharding.tables.store_info.table-strategy.inline.sharding-
column = id
spring.shardingsphere.sharding.tables.store_info.table-
strategy.inline.algorithm-expression = store_info


# product_info分表策略，分布在ds1,ds2的product_info_1和product_info_2表 ，分片策略为
product_info_id % 2+1，product_info_id采用雪花算法
spring.shardingsphere.sharding.tables.product_info.actual-data-nodes = ds$->
{1..2}.product_info_$->{1..2}
spring.shardingsphere.sharding.tables.product_info.table-
strategy.inline.sharding-column = product_info_id
spring.shardingsphere.sharding.tables.product_info.table-
strategy.inline.algorithm-expression = product_info_$->{product_info_id % 2+1}
spring.shardingsphere.sharding.tables.product_info.key-
generator.column=product_info_id
spring.shardingsphere.sharding.tables.product_info.key-generator.type=SNOWFLAKE

# product_descript分表策略，分布在ds1,ds2的product_descript_1和product_descript_2表
，分片策略为product_info_id % 2+1，id采用雪花算法
spring.shardingsphere.sharding.tables.product_descript.actual-data-nodes = ds$->
{1..2}.product_descript_$->{1..2}
spring.shardingsphere.sharding.tables.product_descript.table-
strategy.inline.sharding-column = product_info_id
spring.shardingsphere.sharding.tables.product_descript.table-
strategy.inline.algorithm-expression = product_descript_$->{product_info_id %
2+1}
spring.shardingsphere.sharding.tables.product_descript.key-generator.column=id
spring.shardingsphere.sharding.tables.product_descript.key-
generator.type=SNOWFLAKE

# 设置product_info,product_descript为绑定表，这两张表中的分片键名字必须一致
spring.shardingsphere.sharding.binding-tables = product_info,product_descript
```

```
# 打开sql输出日志
spring.shardingsphere.props.sql.show = true
```

由于配置代码太多，为了便于理解，可以参考下图进行梳理：



1. store_db数据库只有一张表: store_info
2. product_db_1和product_db_2数据库各有4张表:
   product_info_1、product_info_2、product_descript_1、product_descript_2

(4) Dao层代码

```
@Mapper
@Component
public interface ProductDao {

    @Insert("insert into
product_info(store_info_id,product_name,spec,region_code,price,image_url)
value(#{storeInfoId},#{productName},#{spec},#{regionCode},#{price},#
{imageUrl})")
    @Options(useGeneratedKeys = true,keyProperty = "id",keyColumn =
"product_info_id")
    int insertProductInfo(ProductInfo productInfo);

    @Insert("insert into
product_descript(product_info_id,descript,store_info_id) value(#
{productInfoId},#{descript},#{storeInfoId})")
    @Options(useGeneratedKeys = true,keyProperty = "id",keyColumn = "id")
    int insertProductDescript(ProductDescript productDescript);


    @Select("select i.*, d.descript from product_info i inner join
product_descript d on i.product_info_id = d.product_info_id ")
    List<ProductInfo> selectProductList();

}
```

其他代码不在这里展示，比较简单，和之前的开发没有什么变化。

2.3.6 测试

- 通过**创建商品接口**新增商品进行分库验证，store_info_id为奇数的数据在product_db_1,为偶数的数据在product_db_2。

- 通过**创建商品接口**新增商品进行分表验证，product_id为奇数的数据在product_info_1、product_descript_1，为偶数的数据在product_info_2、product_descript_2。
- 主从数据同步验证
- 通过**商品查询接口**进行商品数据查询，验证读写分离

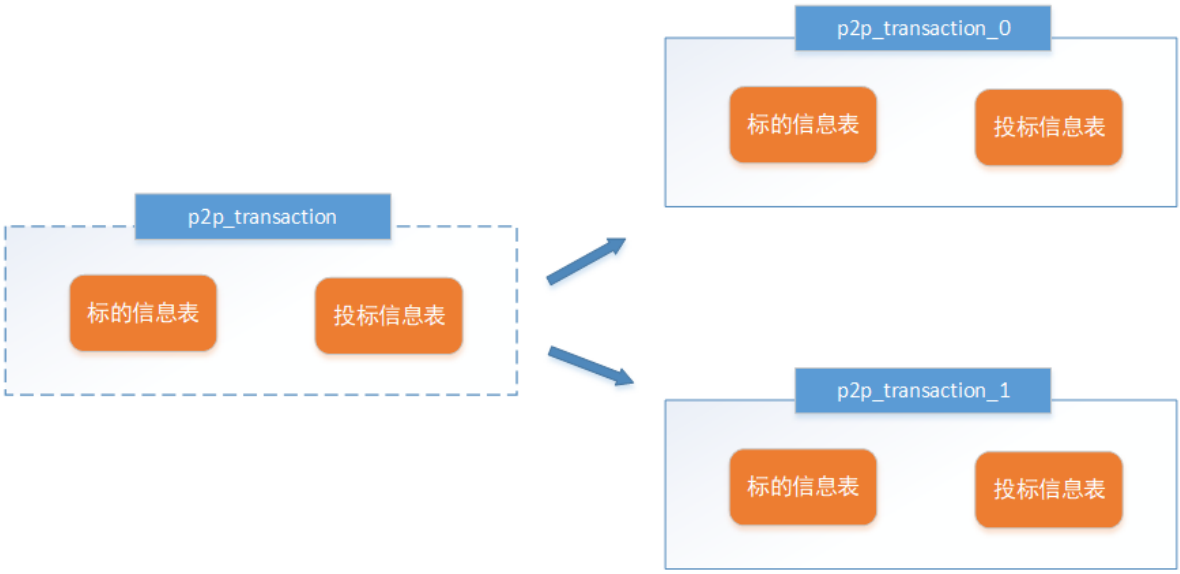
3 P2P项目分库分表

3.1 问题分析

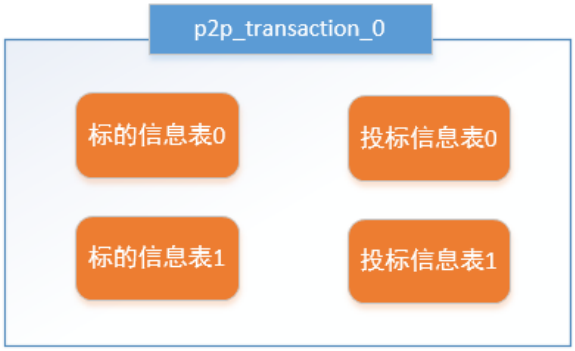
在P2P平台中，标的信息和投标信息做为平台基础业务数据存在。随着平台的发展，这些数据可能会越来越多，甚至达到亿级。以MySQL为例，单库数据量在5000万以内性能比较好，超过阈值后性能会随着数据量的增大而明显降低。单表的数据量超过1000w，性能也会下降严重。这就会导致查询一次所花的时间变长，并发操作达到一定量时可能会卡死，甚至把系统给拖垮，因此我们需要对P2P平台的数据库进行分库分表提升性能，并使用Sharding-JDBC进行数据库操作。

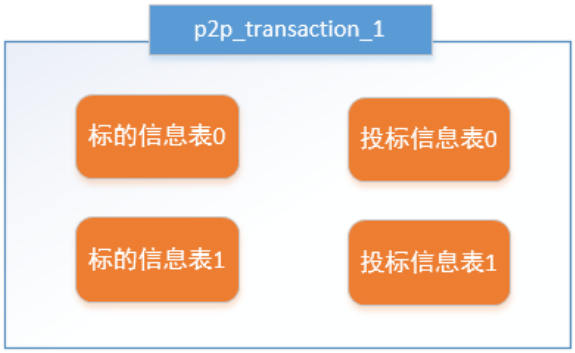
3.2 数据库设计

1. 我们会单独创建p2p_transaction数据库(交易中心)存储和标的相关的数据，例如：标的信息、投标信息等。首先对该数据库进行分库，相同发标人的数据最好不要分散，否则查询相关信息要跨库，因此以**发标人**作为分片键，分片策略采取**发标人ID % 2**。



2. 然后对库内的标的信息和投标信息进行分表，根据需求此两表会以**标的ID**作为关联键联合查询，因此以**标的ID**作为分片键，分片策略采取**标的ID % 2**，并将标的信息和投标信息设置为**绑定表**，最终形成如下数据库设计：





3. 在学习第一章的时候，已经把上述数据库和表造好了，可以查看一下

注意：P2P项目中不再演示主从架构和数据同步，由大家自行实现

3.3 开发环境搭建

1. 把资料文件夹中的wanxinp2p-transaction-service工程导入到当前P2P项目中
2. 检查maven依赖和工程中的基本代码
3. 在Apollo中创建transaction-service项目，在application名称空间中进行如下配置：

```
swagger.enable = true
spring.mvc.throw-exception-if-no-handler-found = true

spring.shardingsphere.datasource.names = ds0,ds1

spring.shardingsphere.datasource.ds0.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds0.driver-class-name =
com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds0.url =
jdbc:mysql://localhost:3306/p2p_transaction_0?useUnicode=true
spring.shardingsphere.datasource.ds0.username = root
spring.shardingsphere.datasource.ds0.password = 123

spring.shardingsphere.datasource.ds1.type =
com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds1.driver-class-name =
com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds1.url =
jdbc:mysql://localhost:3306/p2p_transaction_1?useUnicode=true
```

```
spring.shardingsphere.datasource.ds1.username = root
spring.shardingsphere.datasource.ds1.password = 123

spring.shardingsphere.sharding.default-database-strategy.inline.sharding-column
= CONSUMER_ID
spring.shardingsphere.sharding.default-database-strategy.inline.algorithm-
expression = ds$->{CONSUMER_ID % 2}

spring.shardingsphere.sharding.tables.project.actual-data-nodes = ds$Missing
superscript or subscript argument->{0..1}.project_$->{0..1}
spring.shardingsphere.sharding.tables.project.table-strategy.inline.sharding-
column = ID
spring.shardingsphere.sharding.tables.project.table-strategy.inline.algorithm-
expression = project_$->{ID % 2}

spring.shardingsphere.sharding.tables.tender.actual-data-nodes = ds$Missing
superscript or subscript argument->{0..1}.tender_$->{0..1}
spring.shardingsphere.sharding.tables.tender.table-strategy.inline.sharding-
column = PROJECT_ID
spring.shardingsphere.sharding.tables.tender.table-strategy.inline.algorithm-
expression = tender_$->{PROJECT_ID % 2}

spring.shardingsphere.sharding.binding-tables = project,tender

spring.shardingsphere.props.sql.show = true
```

其他配置请自行完成。

4. 检查application.yml的配置
5. 设置启动参数：-Denv=dev -Dapollo.cluster=DEFAULT -Dserver.port=53060