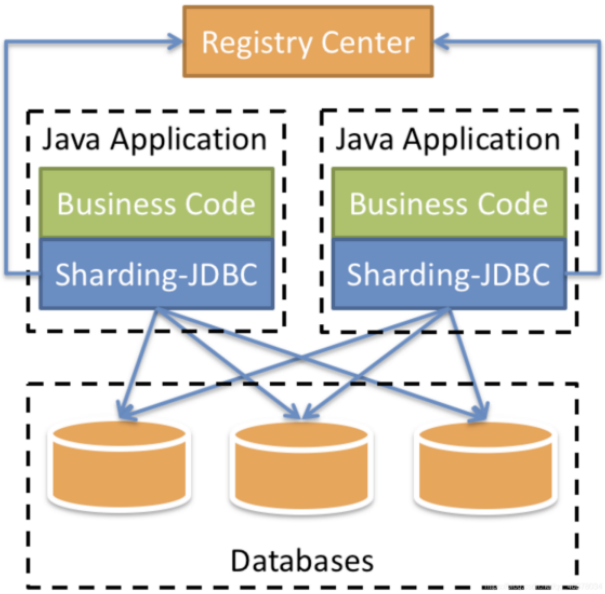
SpringBoot + Sharding JDBC，一文搞定分库分表、读写分离

java1234 2022-10-06 09:06 发表于江苏

大家好 我是锋哥！

Sharding-JDBC 最早是当当网内部使用的一款分库分表框架，到2017年的时候才开始对外开源，这几年在大量社区贡献者的不断迭代下，功能也逐渐完善，现已更名为 ShardingSphere ，2020年4月16日正式成为 Apache 软件基金会的顶级项目。

ShardingSphere-Jdbc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，可理解为增强版的Jdbc驱动，完全兼容Jdbc和各种ORM框架。



随着版本的不断更迭 ShardingSphere 的核心功能也变得多元化起来。

从最开始 Sharding-JDBC 1.0 版本只有数据分片，到 Sharding-JDBC 2.0 版本开始支持数据库治理（注册中心、配置中心等等），再到 Sharding-JDBC 3.0版本又加分布式事务 （支持 Atomikos、Narayana、Bitronix、Seata），如今已经迭代到了 Sharding-JDBC 4.0 版本。

现在的 ShardingSphere 不单单是指某个框架而是一个生态圈，这个生态圈 Sharding-JDBC 、Sharding-Proxy 和 Sharding-Sidecar 这三款开源的分布式数据库中间件解决方案所构成。

ShardingSphere 的前身就是 Sharding-JDBC，所以它是整个框架中最为经典、成熟的组件，先从 Sharding-JDBC 框架入手学习分库分表。

1 核心概念

分库分表

分库，显而易见，就是一个数据库分成多个数据库，部署到不同机器。

分表，就是一个数据库表分成多个表。

分片

一般在提到分库分表的时候，大多是以 水平切分模式 （水平分库、分表）为基础来说的，数据分片将原本一张数据量较大的表例如 t_order 拆分生成数个表结构完全一致的小数据量表 t_order_0、t_order_1、…、t_order_n，每张表只存储原大表中的一部分数据，当执行一条SQL时会通过 分库策略 、 分片策略 将数据分散到不同的数据库、表内。

数据节点

数据节点是分库分表中一个不可再分的最小数据单元（表），它由数据源名称和数据表组成，例如上图中 order_db_1.t_order_0、order_db_2.t_order_1 就表示一个数据节点。

逻辑表

逻辑表是指一组具有相同逻辑和数据结构表的总称。

比如将订单表 t_order 拆分成 t_order_0 … t_order_9 等 10张表。

此时会发现分库分表以后数据库中已不在有 t_order 这张表，取而代之的是 t_order_n，但在代码中写 SQL 依然按 t_order 来写。此时 t_order 就是这些拆分表的逻辑表。

真实表

真实表也就是上边提到的 t_order_n 数据库中真实存在的物理表。

分片键

用于分片的数据库字段。将 t_order 表分片以后，当执行一条SQL时，通过对字段 order_id 取模的方式来决定，这条数据该在哪个数据库中的哪个表中执行，此时 order_id 字段就是 t_order 表的分片健。

这样以来同一个订单的相关数据就会存在同一个数据库表中，大幅提升数据检索的性能，不仅如此 sharding-jdbc 还支持根据多个字段作为分片健进行分片。

分片算法

上边提到可以用分片健取模的规则分片，但这只是比较简单的一种，在实际开发中还希望用 >=、<=、>、<、BETWEEN 和 IN 等条件作为分片规则，自定义分片逻辑，这时就需要用到分片策略与分片算法。

从执行 SQL 的角度来看，分库分表可以看作是一种路由机制，把 SQL 语句路由到期望的数据库或数据表中并获取数据，分片算法可以理解成一种路由规则。

咱们先捋一下它们之间的关系，分片策略只是抽象出的概念，它是由分片算法和分片健组合而成，分片算法做具体的数据分片逻辑。

分库、分表的分片策略配置是相对独立的，可以各自使用不同的策略与算法，每种策略中可以是多个分片算法的组合，每个分片算法可以对多个分片健做逻辑判断。

分片算法和分片策略的关系

注意：sharding-jdbc 并没有直接提供分片算法的实现，需要开发者根据业务自行实现。

sharding-jdbc 提供了4种分片算法。

1、精确分片算法

精确分片算法（PreciseShardingAlgorithm）用于单个字段作为分片键，SQL中有 = 与 IN 等条件的分片，需要在标准分片策略（StandardShardingStrategy ）下使用。

2、范围分片算法

范围分片算法（RangeShardingAlgorithm）用于单个字段作为分片键，SQL中有 BETWEEN AND、>、<、>=、<= 等条件的分片，需要在标准分片策略（StandardShardingStrategy ）下使用。

3、复合分片算法

复合分片算法（ComplexKeysShardingAlgorithm）用于多个字段作为分片键的分片操作，同时获取到多个分片健的值，根据多个字段处理业务逻辑。需要在复合分片策略（ComplexShardingStrategy ）下使用。

4、Hint分片算法

Hint分片算法（HintShardingAlgorithm）稍有不同，上边的算法中都是解析SQL 语句提取分片键，并设置分片策略进行分片。但有些时候并没有使用任何的分片键和分片策略，可还想将SQL 路由到目标数据库和表，就需要通过手动干预指定SQL的目标数据库和表信息，这也叫强制路由。

分片策略

上边讲分片算法的时候已经说过，分片策略是一种抽象的概念，实际分片操作的是由分片算法和分片健来完成的。

1、标准分片策略

标准分片策略适用于单分片键，此策略支持 PreciseShardingAlgorithm 和 RangeShardingAlgorithm 两个分片算法。

其中 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， >， <，>=，<= 条件分片，如果不配置RangeShardingAlgorithm，SQL中的条件等将按照全库路由处理。

2、复合分片策略

复合分片策略，同样支持对 SQL语句中的 =，>， <， >=， <=，IN和 BETWEEN AND 的分片操作。不同的是它支持多分片键，具体分配片细节完全由应用开发者实现。

3、行表达式分片策略

行表达式分片策略，支持对 SQL语句中的 = 和 IN 的分片操作，但只支持单分片键。这种策略通常用于简单的分片，不需要自定义分片算法，可以直接在配置文件中接着写规则。

t_order_$->{t_order_id % 4} 代表 t_order 对其字段 t_order_id取模，拆分成4张表，而表名分别是t_order_0 到 t_order_3。

4、Hint分片策略

Hint分片策略，对应上边的Hint分片算法，通过指定分片健而非从 SQL中提取分片健的方式进行分片的策略。

分布式主键

数据分片后，不同数据节点生成全局唯一主键是非常棘手的问题，同一个逻辑表（t_order）内的不同真实表（t_order_n）之间的自增键由于无法互相感知而产生重复主键。

尽管可通过设置自增主键 初始值 和 步长 的方式避免ID碰撞，但这样会使维护成本加大，乏完整性和可扩展性。如果后去需要增加分片表的数量，要逐一修改分片表的步长，运维成本非常高，所以不建议这种方式。

为了让上手更加简单，ApacheShardingSphere 内置了UUID、SNOWFLAKE 两种分布式主键生成器，默认使用雪花算法（snowflake）生成64bit的长整型数据。不仅如此它还抽离出分布式主键生成器的接口，方便实现自定义的自增主键生成算法。

广播表

广播表：存在于所有的分片数据源中的表，表结构和表中的数据在每个数据库中均完全一致。一般是为字典表或者配置表 t_config，某个表一旦被配置为广播表，只要修改某个数据库的广播表，所有数据源中广播表的数据都会跟着同步。

绑定表

绑定表：那些分片规则一致的主表和子表。比如：t_order 订单表和 t_order_item 订单服务项目表，都是按 order_id 字段分片，因此两张表互为绑定表关系。

那绑定表存在的意义是啥呢？

通常在业务中都会使用 t_order 和 t_order_item 等表进行多表联合查询，但由于分库分表以后这些表被拆分成N多个子表。如果不配置绑定表关系，会出现笛卡尔积关联查询，将产生如下四条SQL。

```
SELECT * FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id
SELECT * FROM t_order_0 o JOIN t_order_item_1 i ON o.order_id=i.order_id
SELECT * FROM t_order_1 o JOIN t_order_item_0 i ON o.order_id=i.order_id
SELECT * FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id
```

而配置绑定表关系后再进行关联查询时，只要对应表分片规则一致产生的数据就会落到同一个库中，那么只需 t_order_0 和 t_order_item_0 表关联即可。

```
SELECT * FROM t_order_0 o JOIN t_order_item_0 i ON o.order_id=i.order_id
SELECT * FROM t_order_1 o JOIN t_order_item_1 i ON o.order_id=i.order_id
```

注意：在关联查询时 t_order 它作为整个联合查询的主表。所有相关的路由计算都只使用主表的策略，t_order_item 表的分片相关的计算也会使用 t_order 的条件，所以要保证绑定表之间的分片键要完全相同。

2 MySQL主从复制

docker配置mysql主从复制

创建主服务器所需目录

```
mkdir -p /usr/local/mysqlData/master/cnf
mkdir -p /usr/local/mysqlData/master/data
```

定义主服务器配置文件

```
vim /usr/local/mysqlData/master/cnf/mysql.cnf
```

```
[mysqld]
## 设置server_id,注意要唯一
server-id=1
## 开启binlog
log-bin=mysql-bin
## binlog缓存
binlog_cache_size=1M
## binlog格式(mixed、statement、row,默认格式是statement)
binlog_format=mixed
```

创建并启动**mysql**主服务

```
docker run -itd -p 3306:3306 --name master -v /usr/local/mysqlData/master/cnf:/etc/mysql/conf.d -
```

添加复制**master**数据的用户**reader**，供从服务器使用

```
[root@aliyun /]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
6af1df686fff        mysql:5.7           "docker-entrypoint..."   5 seconds ago       Up 4 seconds
[root@aliyun /]# docker exec -it master /bin/bash
root@41d795785db1:/# mysql -u root -p123456

mysql> GRANT REPLICATION SLAVE ON *.* to 'reader'@'%' identified by 'reader';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

创建从服务器所需目录，编辑配置文件

```
mkdir /usr/local/mysqlData/slave/cnf -p
mkdir /usr/local/mysqlData/slave/cnf -p
vim /usr/local/mysqlData/slave/cnf/mysql.cnf

[mysqld]
## 设置server_id,注意要唯一
server-id=2
## 开启binlog,以备Slave作为其它Slave的Master时使用
```

```
log-bin=mysql-slave-bin
## relay_log配置中继日志
relay_log=edu-mysql-relay-bin
## 如果需要同步函数或者存储过程
log_bin_trust_function_creators=true
## binlog缓存
binlog_cache_size=1M
## binlog格式(mixed、statement、row,默认格式是statement)
binlog_format=mixed
## 跳过主从复制中遇到的所有错误或指定类型的错误,避免slave端复制中断
## 如:1062错误是指一些主键重复,1032错误是因为主从数据库数据不一致
slave_skip_errors=1062
```

创建并运行**mysql**从服务器

```
docker run -itd -p 3307:3306 --name slaver -v /usr/local/mysqlData/slave/cnf:/etc/mysql/conf.d -v
```

在从服务器上配置连接主服务器的信息

首先主服务器上查看master_log_file、master_log_pos两个参数，然后切换到从服务器上进行主服务器的连接信息的设置

主服务上执行：

```
root@6af1df686fff:/# mysql -u root -p123456

mysql> show master status;
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-bin.000003 |      591 |              |                  |                   |
+------------------+----------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

docker查看主服务器容器的ip地址

```
[root@aliyun /]# docker inspect --format='{{.NetworkSettings.IPAddress}}' master
172.17.0.2
```

从服务器上执行：

```
[root@aliyun /]# docker exec -it slaver /bin/bash
root@fe8b6fc2f1ca:/# mysql -u root -p123456

mysql> change master to master_host='172.17.0.2',master_user='reader',master_password='reader',ma
```

从服务器启动I/O 线程和SQL线程

```
mysql> start slave;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> show slave status\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 172.17.0.2
                  Master_User: reader
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000003
          Read_Master_Log_Pos: 591
               Relay_Log_File: edu-mysql-relay-bin.000002
                Relay_Log_Pos: 320
        Relay_Master_Log_File: mysql-bin.000003
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

Slave_IO_Running: Yes，Slave_SQL_Running: Yes 即表示启动成功

binlog和redo log回顾

redo log（重做日志）

InnoDB首先将redo log放入到redo log buffer，然后按一定频率将其刷新到redo log file

下列三种情况下会将redo log buffer刷新到redo log file：

- Master Thread每一秒将redo log buffer刷新到redo log file
- 每个事务提交时会将redo log buffer刷新到redo log file

- 当redo log缓冲池剩余空间小于1/2时，会将redo log buffer刷新到redo log file

MySQL里常说的WAL技术，全称是Write Ahead Log，即当事务提交时，先写redo log，再修改页。也就是说，当有一条记录需要更新的时候，InnoDB会先把记录写到redo log里面，并更新Buffer Pool的page，这个时候更新操作就算完成了

Buffer Pool是物理页的缓存，对InnoDB的任何修改操作都会首先在Buffer Pool的page上进行，然后这样的页将被标记为脏页并被放到专门的Flush List上，后续将由专门的刷脏线程阶段性的将这些页面写入磁盘

InnoDB的redo log是固定大小的，比如可以配置为一组4个文件，每个文件的大小是1GB，循环使用，从头开始写，写到末尾就又回到开头循环写（顺序写，节省了随机写磁盘的IO消耗）

Write Pos是当前记录的位置，一边写一边后移，写到第3号文件末尾后就回到0号文件开头。Check Point是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件

Write Pos和Check Point之间空着的部分，可以用来记录新的操作。如果Write Pos追上Check Point，这时候不能再执行新的更新，需要停下来擦掉一些记录，把Check Point推进一下

当数据库发生宕机时，数据库不需要重做所有的日志，因为Check Point之前的页都已经刷新回磁盘，只需对Check Point后的redo log进行恢复，从而缩短了恢复的时间

当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Check Point，将脏页刷新回磁盘

binlog（归档日志）

MySQL整体来看就有两块：一块是Server层，主要做的是MySQL功能层面的事情；还有一块是引擎层，负责存储相关的具体事宜。redo log是InnoDB引擎特有的日志，而Server层也有自己

的日志，称为binlog

binlog记录了对MySQL数据库执行更改的所有操作，不包括SELECT和SHOW这类操作，主要作用是用于数据库的主从复制及数据的增量恢复

使用mysqldump备份时，只是对一段时间的数据进行全备，但是如果备份后突然发现数据库服务器故障，这个时候就要用到binlog的日志了

binlog格式有三种：STATEMENT，ROW，MIXED

- STATEMENT模式：binlog里面记录的就是SQL语句的原文。优点是并不需要记录每一行的数据变化，减少了binlog日志量，节约IO，提高性能。缺点是在某些情况下会导致master-slave中的数据不一致
- ROW模式：不记录每条SQL语句的上下文信息，仅需记录哪条数据被修改了，修改成什么样了，解决了STATEMENT模式下出现master-slave中的数据不一致。缺点是会产生大量的日志，尤其是alter table的时候会让日志暴涨
- MIXED模式：以上两种模式的混合使用，一般的复制使用STATEMENT模式保存binlog，对于STATEMENT模式无法复制的操作使用ROW模式保存binlog，MySQL会根据执行的SQL语句选择日志保存方式

redo log和**binlog**日志的不同

- redo log是InnoDB引擎特有的；binlog是MySQL的Server层实现的，所有引擎都可以使用
- redo log是物理日志，记录的是在某个数据也上做了什么修改；binlog是逻辑日志，记录的是这个语句的原始逻辑，比如给ID=2这一行的c字段加1
- redo log是循环写的，空间固定会用完；binlog是可以追加写入的，binlog文件写到一定大小后会切换到下一个，并不会覆盖以前的日志

两阶段提交

```
create table T(ID int primary key, c int);

update T set c=c+1 where ID=2;
```

执行器和InnoDB引擎在执行这个update语句时的内部流程：

- 执行器先找到引擎取ID=2这一行。ID是主键，引擎直接用树搜索找到这一行。如果ID=2这一行所在的数据也本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回

- 执行器拿到引擎给的行数据，把这个值加上1，得到新的一行数据，再调用引擎接口写入这行新数据
- 引擎将这行新数据更新到内存中，同时将这个更新操作记录到redo log里面，此时redo log处于prepare状态。然后告知执行器执行完成了，随时可以提交事务
- 执行器生成这个操作的binlog，并把binlog写入磁盘
- 执行器调用引擎的提交事务接口，引擎把刚刚写入的redo log改成提交状态，更新完成

update语句的执行流程图如下，图中浅色框表示在InnoDB内部执行的，深色框表示是在执行器中执行的

将redo log的写入拆成了两个步骤：prepare和commit，这就是两阶段提交

MySQL主从复制原理

从库B和主库A之间维持了一个长连接。主库A内部有一个线程，专门用于服务从库B的这个长连接。一个事务日志同步的完整过程如下：

- 在从库B上通过change master命令，设置主库A的IP、端口、用户名、密码，以及要从哪个位置开始请求binlog，这个位置包含文件名和日志偏移量
- 在从库B上执行start slave命令，这时从库会启动两个线程，就是图中的I/O线程和SQL线程。其中I/O线程负责与主库建立连接
- 主库A校验完用户名、密码后，开始按照从库B传过来的位置，从本地读取binlog，发给B
- 从库B拿到binlog后，写到本地文件，称为中继日志
- SQL线程读取中继日志，解析出日志里的命令，并执行

由于多线程复制方案的引入，SQL线程演化成了多个线程。

主从复制不是完全实时地进行同步，而是异步实时。这中间存在主从服务之间的执行延时，如果主服务器的压力很大，则可能导致主从服务器延时较大。

3 为什么需要分库分表？

首先要明确一个问题，单一的数据库是否能够满足公司目前的线上业务需求，比如用户表，可能有几千万，甚至上亿的数据，只是说可能，如果有这么多用户，那必然是大公司了，那么这个时候，如果不分表也不分库的话，那么数据了上来的时候，稍微一个不注意，MySQL单机磁盘容量会撑爆，但是如果拆成多个数据库，磁盘使用率大大降低。

这样就把磁盘使用率降低，这是通过硬件的形式解决问题，如果数据量是巨大的，这时候，SQL 如果没有命中索引，那么就会导致一个情况，查这个表的SQL语句直接把数据库给干崩了。

即使SQL命中了索引，如果表的数据量 超过一千万的话， 查询也是会明显变慢的。这是因为索引一般是B+树结构，数据千万级别的话，B+树的高度会增高，查询自然就变慢了，当然，这是题外话了。

4 MySQL分库分表原理

分库分表

水平拆分： 同一个表的数据拆到不同的库不同的表中。可以根据时间、地区或某个业务键维度，也可以通过hash进行拆分，最后通过路由访问到具体的数据。拆分后的每个表结构保持一致

垂直拆分： 就是把一个有很多字段的表给拆分成多个表，或者是多个库上去。每个库表的结构都不一样，每个库表都包含部分字段。一般来说，可以根据业务维度进行拆分，如订单表可以拆分为订单、订单支持、订单地址、订单商品、订单扩展等表；也可以，根据数据冷热程度拆分，20%的热点字段拆到一个表，80%的冷字段拆到另外一个表

不停机分库分表数据迁移

一般数据库的拆分也是有一个过程的，一开始是单表，后面慢慢拆成多表。那么就看下如何平滑的从MySQL单表过度到MySQL的分库分表架构

- 利用MySQL+Canal做增量数据同步，利用分库分表中间件，将数据路由到对应的新表中
- 利用分库分表中间件，全量数据导入到对应的新表中
- 通过单表数据和分库分表数据两两比较，更新不匹配的数据到新表中
- 数据稳定后，将单表的配置切换到分库分表配置上

5 分库分表方案

分库分表方案，不外乎就两种，一种是垂直切分，一种是水平切分。

但是总有做开发的小伙伴不知道这垂直切分和水平切分到底是什么样的，为什么垂直切分，为什么水平切分，什么时候应该选择垂直切分，什么时候应该选择水平切分。

有人是这么说的，垂直切分是根据业务来拆分数据库，同一类业务的数据表拆分到一个独立的数据库，另一类的数据表拆分到其他数据库。

有些人不理解这个，实际上垂直切分也是有划分的，上面描述的是垂直切分数据库，可能容易让很多人不太理解，但是如果是垂直切分表，那么肯定百分之90的人都能理解。

垂直切分

有一张Order表，表中有诸多记录，比如设计这么一张简单的表。

id	order_id	order_date	order_type	order_state
1	cd96cff0356e483caae6b2ff4e878fd6	2022-06-18 13:57:11	支付宝	1
2	e2496f9e22ce4391806b18480440526a	2022-06-18 14:22:33	微信	2
3	9e7ab5a1915c4570a9eaaaa3c01f79c1	2022-06-18 15:21:44	现金	2

以上是简化版Order表，如果想要垂直切分，那么应该怎么处理？

直接拆分成2个表，这时候就直接就一分为2 ，咔的一下拆分成两个表

Order1

id	order_id	order_date
1	cd96cff0356e483caae6b2ff4e878fd6	2022-06-18 13:57:11
2	e2496f9e22ce4391806b18480440526a	2022-06-18 14:22:33
3	9e7ab5a1915c4570a9eaaaa3c01f79c1	2022-06-18 15:21:44

Order2

id	order_type	order_state

id	order_type	order_state
1	支付宝	1
2	微信	2
3	现金	2

这时候主键ID保持的时一致的，而这个操作，就是垂直拆分，分表的操作。

既然说了垂直拆分，那么必然就有水平拆分，

什么是水平拆分呢？

实际上水平拆分的话，那真的是只有一句话。

水平切分

按照数据来拆分

水平拆分数据库： 将一张表的数据 (按照数据行) 分到多个不同的数据库。每个库的表结构相同，每个库都只有这张表的部分数据，当单表的数据量过大，如果继续使用水平分库，那么数据库的实例 就会不断增加，不利于系统的运维。这时候就要采用水平分表。

水平拆分表： 将一张表的数据 (按照数据行) ，分配到同一个数据库的多张表中，每个表都只有一部分数据。

来看看Order表进行水平拆分的话，是什么样子的。

Order1

id	order_id	order_date	order_type	order_state
1	cd96cff0356e483caae6b2ff4e878fd6	2022-06-18 13:57:11	支付宝	1
2	e2496f9e22ce4391806b18480440526a	2022-06-18 14:22:33	微信	2

Order2

id	order_id	order_date	order_type	order_state

id	order_id	order_date	order_type	order_state
3	9e7ab5a1915c4570a9eaaaa3c01f79c1	2022-06-18 15:21:44	现金	2

实际上就是水平的把表数据给分成了2份，这么看起来是不是就很好理解了。

6 分库分表带来的问题

事务问题

首先，分库分表最大的隐患就是，事务的一致性， 当需要更新的内容同时分布在不同的库时，不可避免的会产生跨库的事务问题。

原来在一个数据库操 作，本地事务就可以进行控制，分库之后 一个请求可能要访问多个数据库，如何保证事务的一致性，目前还没有简单的解决方案。

无法联表的问题

还有一个就是，没有办法进行联表查询了，因为，原来在一个库中的一些表，被分散到多个库，并且这些数据库可能还不在一台服务器，无法关联查询，所以相对应的业务代码可能就比较多了。

分页问题

分库并行查询时，如果用到了分页，每个库返回的结果集本身是无序的，只有将多个库中的数据先查出来，然后再根据排序字段在内存中进行排序，如果查询结果过大也是十分消耗资源的。

分库分表的技术

目前比较流行的就两种，一种是MyCat，另外一种则是Sharding-Jdbc，都是可以进行分库的，

MyCat是一个数据库中间件，Sharding-Jdbc是以 jar 包提供服务的jdbc框架。

Mycat和Sharding-jdbc 实现原理也是不同：

Mycat的原理中最重要的一个动词是“拦截”，它拦截了用户发送过来的SQL语句，首先对SQL语句做了一些特定的分析：如分库分表分析、路由分析、读写分离分析、缓存分析等，然后将此SQL发往后端的真实数据库，并将返回的结果做适当的处理，最终再返回给用户。

而Sharding-JDBC的原理是接受到一条SQL语句时，会陆续执行SQL解析 => 查询优化 => SQL路由 => SQL改写 => SQL执行 => 结果归并 ，最终返回执行结果。

小结

垂直分表：将一张宽表(字段很多的表)，按照字段的访问频次进行拆分，就是按照表单结构进行 拆。

垂直分库：根据不同的业务，将表进行分类，拆分到不同的数据库。这些库可以部署在不同的服 务器，分摊访问压力。

水平分库：将一张表的数据 (按照数据行) 分到多个不同的数据库。每个库的表结构相同

水平分表：将一张表的数据 (按照数据行) ，分配到同一个数据库的多张表中，每个表都只有一部 分数据。

7 Sharding-Jdbc实现读写分离

搭建mysql主从服务

根据上面 docker配置mysql主从复制 部分搭建。

主服务创建库表

```
CREATE DATABASE sharding-jdbc-db;

CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
```

```
  `nickname` varchar(100) DEFAULT NULL,
  `password` varchar(100) DEFAULT NULL,
  `sex` int(11) DEFAULT NULL,
  `birthday` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

创建SpringBoot工程，引入依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-in
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.8</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.itjing</groupId>
    <artifactId>springboot-sharding-jdbc</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-sharding-jdbc</name>
    <description>springboot-sharding-jdbc</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <!-- web -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- lombok -->
        <dependency>
            <groupId>org.projectlombok</groupId>
```

```
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <!-- mybatis -->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>1.3.2</version>
        </dependency>

        <!-- mysql -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>

        <!-- druid -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
            <version>1.1.17</version>
        </dependency>

        <!-- sharding-jdbc -->
        <dependency>
            <groupId>org.apache.shardingsphere</groupId>
            <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
            <version>4.0.0-RC1</version>
        </dependency>

        <!-- test -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
```

```
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

配置文件

```
spring:
  main:
    allow-bean-definition-overriding: true
  shardingsphere:
    datasource:
      ds:
        maxPoolSize: 100
      # master-ds1数据库连接信息
      ds1:
        driver-class-name: com.mysql.jdbc.Driver
        maxPoolSize: 100
        minPoolSize: 5
        password: 123456
        type: com.alibaba.druid.pool.DruidDataSource
        url: jdbc:mysql://192.168.56.111:3306/sharding-jdbc-db?useUnicode=true&useSSL=false&serve
        username: root
      # slave-ds2数据库连接信息
```

```
      ds2:
        driver-class-name: com.mysql.jdbc.Driver
        maxPoolSize: 100
        minPoolSize: 5
        password: 123456
        type: com.alibaba.druid.pool.DruidDataSource
        url: jdbc:mysql://192.168.56.111:3307/sharding-jdbc-db?useUnicode=true&useSSL=false&serve
        username: root
      # slave-ds3数据库连接信息
      ds3:
        driver-class-name: com.mysql.jdbc.Driver
        minPoolSize: 5
        password: 123456
        type: com.alibaba.druid.pool.DruidDataSource
        url: jdbc:mysql://192.168.56.111:3307/sharding-jdbc-db?useUnicode=true&useSSL=false&serve
        username: root
      # 配置数据源
      names: ds1,ds2,ds3
    masterslave:
      # 配置slave节点的负载均衡均衡策略,采用轮询机制
      load-balance-algorithm-type: round_robin
      # 配置主库master,负责数据的写入
      master-data-source-name: ds1
      # 配置主从名称
      name: ms
      # 配置从库slave节点
      slave-data-source-names: ds2,ds3
    # 显示sql
    props:
      sql:
        show: true
    # 配置默认数据源ds1 默认数据源,主要用于写
    sharding:
      default-data-source-name: ds1

# 整合mybatis的配置
mybatis:
  type-aliases-package: com.itjing.sharding.entity
```

定义Controller、Mapper、Entity

```
package com.itjing.sharding.entity;

import lombok.Data;

/**
 * @author lijing
 * @date 2022年06月19日 10:45
 * @description
 */
@Data
public class User {
    private Integer id;

    private String nickname;

    private String password;

    private Integer sex;

    private String birthday;
}

package com.itjing.sharding.controller;

import com.itjing.sharding.entity.User;
import com.itjing.sharding.mapper.UserMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Random;

/**
 * @author lijing
 * @date 2022年06月19日 10:45
 * @description
```

```
 */
@RestController
@RequestMapping("/api/user")
public class UserController {

    @Autowired
    private UserMapper userMapper;

    @PostMapping("/save")
    public String addUser() {
        User user = new User();
        user.setNickname("zhangsan" + new Random().nextInt());
        user.setPassword("123456");
        user.setSex(1);
        user.setBirthday("1997-12-03");
        userMapper.addUser(user);
        return "success";
    }

    @GetMapping("/findUsers")
    public List<User> findUsers() {
        return userMapper.findUsers();
    }
}


package com.itjing.sharding.mapper;

import com.itjing.sharding.entity.User;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

import java.util.List;

/**
 * @author lijing
 * @date 2022年06月19日 10:46
 * @description
 */
@Mapper
public interface UserMapper {
```

```
    @Insert("insert into t_user(nickname,password,sex,birthday) values(#{nickname},#{password},#{
    void addUser(User user);

    @Select("select * from t_user")
    List<User> findUsers();
}
```

启动项目验证

启动日志中三个数据源初始化成功：

调用 `http://localhost:8080/api/user/save` 一直进入到ds1主节点

调用 `http://localhost:8080/api/user/findUsers` 一直进入到ds2、ds3节点，并且轮询进入

8 Sharding-Jdbc实现分库分表

分表

创建库表

创建数据库及其对应的相同的两张表结构的表

先在MySQL上创建数据库，直接起名叫做 `order`

然后分别创建两个表，分别是 `order_1` 和 `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，主键为偶数的落入order_1表 ，为奇数的落入order_2表，再通过Sharding-Jdbc 进行查询。

```
DROP TABLE IF EXISTS order_1;
CREATE TABLE order_1 (
 order_id BIGINT(20) PRIMARY KEY AUTO_INCREMENT ,
 user_id INT(11) ,
 product_name VARCHAR(128),
 COUNT INT(11)
);

DROP TABLE IF EXISTS order_2;
CREATE TABLE order_2 (
 order_id BIGINT(20) PRIMARY KEY AUTO_INCREMENT ,
 user_id INT(11) ,
 product_name VARCHAR(128),
 COUNT INT(11)
);
```

创建**SpringBoot**的项目

略

依赖

参照读写分离的依赖

配置文件

比较重要的一步，那就是配置分片规则，因为这里的分表是直接把数据进行水平拆分成到2个表中，所以属于水平切分数据表的操作，配置如下：

```
# 读写分离
server:
  servlet:
    encoding:
      enabled: true
```

```
      charset: UTF-8
      force: true
spring:
  application:
    name: sharding-jdbc-simple
  main:
    allow-bean-definition-overriding: true
  shardingsphere:
    datasource:
      names: db1
      db1:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.jdbc.Driver
        url: jdbc:mysql://127.0.0.1:3306/order?characterEncoding=UTF-8&useSSL=false
        username: root
        password: root
    sharding:
      tables:
        order:
          actual-data-nodes: db1.order_$->{1..2}
          key-generator:
            column: order_id
            type: SNOWFLAKE
          # 分表策略
          table-strategy:
            inline:
              sharding-column: order_id
              algorithm-expression: order_$->{order_id % 2 + 1}
    props:
      sql:
        show: true

mybatis:
  configuration:
    map-underscore-to-camel-case: true
```

测试

```
package com.itjing.sharding.mapper;

import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Param;

/**
 * @author lijing
 * @date 2022年06月19日 11:18
 * @description
 */
@Mapper
public interface OrderMapper {

    /**
     * 新增订单
     */
    @Insert("INSERT INTO order(user_id,product_name,COUNT) VALUES(#{user_id},#{product_name},#{co
    int insertOrder(@Param("user_id") int user_id, @Param("product_name") String product_name, @P

}


package com.itjing.sharding.controller;

import com.itjing.sharding.mapper.OrderMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author lijing
 * @date 2022年06月19日 11:20
 * @description
 */
@RestController
@RequestMapping("/api/order")
public class OrderController {

    @Autowired
    private OrderMapper orderMapper;
```

```
    @PostMapping("/save")
    public String testInsertOrder() {
        for (int i = 0; i < 10; i++) {
            orderMapper.insertOrder(100 + i, "空调" + i, 10);
        }
        return "success";
    }
}
```

当执行完毕的时候，可以看下日志：

再看下数据库：

偶数订单在表1中，奇数订单在表2中。

接下来就是直接执行查询，然后去查询对应表中的数据。

给定1表和2表中的一个order_id 来进行 In 查询，看是否能正确返回想要的数据：

```
/**
 * 查询订单
 */
```

```
@Select({"<script>" +
    "select * from order p where p.order_id in " +
    "<foreach collection='orderIds' item='id' open='(' separator = ',' close=')'>#{id}</foreach>"
    + "</script>"})
List<Map> findOrderByIds(@Param("orderIds") List<Long> orderIds);



@GetMapping("find")
public void testFindOrderByIds() {
    List<Long> ids = new ArrayList<>();
    ids.add(745241267423674369L);
    ids.add(745241268338032640L);

    List<Map> list = orderMapper.findOrderByIds(ids);
    System.out.println(list);
}
```

很成功，使用Sharding-JDBC 进行单库水平切分表的操作已经完成了。

分库

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，在上面装好数据库之后，就可以开始进行操作了。

建立库表

建立数据库 order1 和 order2，然后创建相同表结构的表。

```
DROP TABLE IF EXISTS order_info;
CREATE TABLE order_info (
 order_id BIGINT(20) PRIMARY KEY AUTO_INCREMENT ,
 user_id INT(11) ,
 product_name VARCHAR(128),
 COUNT INT(11)
);
```

配置

```
# 读写分离
server:
  servlet:
    encoding:
      enabled: true
      charset: UTF-8
      force: true
spring:
  application:
    name: sharding-jdbc-simple
  main:
    allow-bean-definition-overriding: true
  shardingsphere:
    datasource:
      names: db1,db2
      db1:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/order1?characterEncoding=UTF-8&useSSL=false
        username: root
        password: root
      db2:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost:3306/order2?characterEncoding=UTF-8&useSSL=false
        username: root
        password: root
        ## 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db1数据源，否则操作
    sharding:
      tables:
        order_info:
          actual-data-nodes: db$->{1..2}.order_info
          key-generator:
            column: order_id
            type: SNOWFLAKE
```

```
        # 分库策略
        database-strategy:
          inline:
            sharding-column: user_id
            algorithm-expression: db$->{user_id % 2 + 1}
    props:
      sql:
        show: true

mybatis:
  configuration:
    map-underscore-to-camel-case: true
```

配置文件，在这里是通过配置对数据库的分片策略，来指定数据库进行操作。

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db1数据源，否则操作db2。

这样的分库策略，直接通过 user_id 的奇偶性，来判断到底是用哪个数据源，用哪个数据库和表数据的。

测试

```
@Insert("INSERT INTO order_info(user_id,product_name,COUNT) VALUES(#{user_id},#{product_name},#{c
    int insertOrderFk(@Param("user_id") int user_id, @Param("product_name") String product_name, (


@PostMapping("/saveFk")
public String saveFk() {
    for (int i = 0; i < 10; i++) {
        orderMapper.insertOrderFk(i, "空调" + i, 1);
    }
    return "success";
}
```

看日志的话，看样子是成功了，看一下数据库：

这么看下来，保存的数据是没问题的，从水平切分来看，把数据分别保存了order1和order2库中的 order_info 里面，也就是说数据算是水平切分到了不同的数据库对应的表中。

分库分表后的查询

```
@Select({"<script>"+
    "select * from order_info p where p.order_id in " +
    "<foreach collection='orderIds' item='id' open='(' separator = ',' close=')'>#{id}</foreach>"
    +"</script>"})
List<Map> findOrderByIdsFk(@Param("orderIds") List<Long> orderIds);


@GetMapping("findFk")
public void testFindOrderByIdsFk() {
    List<Long> ids = new ArrayList<>();
    ids.add(7452520930019901 45L);
    ids.add(7452520940967034 88L);

    List<Map> list = orderMapper.findOrderByIdsFk(ids);
    System.out.println(list);
}
```

9 相关配置

在说配置之前，得先了解一下关于Sharding-JDBC的执行流程，不然也不知道这些配置都是干嘛用的。

当把SQL发送给 Sharding 之后，Sharding 会经过五个步骤，然后返回接口，这五个步骤分别是：

- SQL解析
- SQL路由
- SQL改写
- SQL执行
- 结果归并

SQL解析：编写SQL查询的是逻辑表，执行时 ShardingJDBC 要解析SQL，解析的目的是为了找到需要改写的位置。

SQL路由：SQL的路由是指将对逻辑表的操作，映射到对应的数据节点的过程. ShardingJDBC会获取分片键判断是否正确，正确 就执行分片策略(算法) 来找到真实的表。

SQL改写：程序员面向的是逻辑表编写SQL，并不能直接在真实的数据库中执行，SQL改写用于将逻辑 SQL改为在真实的数据库中可以正确执行的SQL。

SQL执行：通过配置规则 `order_$->{order_id % 2 + 1}` ，可以知道当 order_id 为偶数时，应该向 order_1表中插入数据，为奇数时向 order_2表插入数据。

结果归并：将所有真正执行sql的结果进行汇总合并，然后返回。

都知道，要是用Sharding分库分表，那么自然就会有相对应的配置，而这些配置才是比较重要的地方，而其中比较经典的就是分片策略了。

分片策略

分片策略分为分表策略和分库策略，它们实现分片算法的方式基本相同，没有太大的区别，无非一个是针对库，一个是针对表。

而一般分片策略主要是分为如下的几种：

- **standard**：标准分片策略
- **complex**：复合分片策略
- **inline**：行表达式分片策略，使用**Groovy**的表达式.

- **hint：Hint分片策略，对应 `HintShardingStrategy` 。**
- **none：不分片策略，对应 `NoneShardingStrategy` 。**

标准分片策略StandardShardingStrategy

使用场景：SQL 语句中有>，>=, <=，<，=，IN 和 `BETWEEN AND` 操作符，都可以应用此分片策略。

也就是说，SQL 语句中频繁的出现这些符号的时候，而且这个时候还想要进行分库分表的时候，就可以采用这个策略了。

但是这个时候要谨记一些内容，那就是标准分片策略（ `StandardShardingStrategy` ），它只支持对单个分片键（字段）为依据的分库分表，并提供了两种分片算法 `PreciseShardingAlgorithm` （精准分片）和 `RangeShardingAlgorithm` （范围分片）。

在使用标准分片策略时，精准分片算法是必须实现的算法，用于 SQL 含有 = 和 IN 的分片处理；范围分片算法是非必选的，用于处理含有 BETWEEN AND 的分片处理。

复合分片策略

使用场景：SQL 语句中有>，>=，<=，<，=，IN 和 `BETWEEN AND` 等操作符，不同的是复合分片策略支持对多个分片键操作。

这里要注意的就是多个分片键，也就是说，如果分片的话需要使用两个字段作为分片键，自定义复合分片策略。

行表达式分片策略

它的配置相当简洁，这种分片策略利用 `inline.algorithm-expression` 书写表达式。

这里就是使用的这个，来完成的分片，而且行表达式分片策略适用于做简单的分片算法，无需自定义分片算法，省去了繁琐的代码开发，是几种分片策略中最为简单的。

但是要注意，行表达式分片策略，它只支持单分片键。

Hint分片策略

Hint分片策略（ `HintShardingStrategy` ）和其他的分片策略都不一样了，这种分片策略无需配置分片键，分片键值也不再从 SQL中解析，而是由外部指定分片信息，让 SQL在指定的分库、分表中执行。

不分片策略

不分片策略这个没啥可说的，不分片的话，用Sharing-JDBC的话，可能就没啥意思了。毕竟玩的就是分片。

官方文档：https://shardingsphere.apache.org/document/current/cn/overview/

来源：blog.csdn.net/qq_40378034/article/details/115264837

End

阅读原文

喜欢此内容的人还喜欢

Mybatis中SQL注入攻击的3种方式，真是防不胜防！

江南一点雨

node应用故障定位顶级技巧—动态追踪技术[Dynamic Trace]

元语言

基于 Flask、Echarts、Pandas 等实现的图书分析大屏展示系统（附源码）

Python联盟