# 概述

## 1.1单库单表遇到的问题

在传统单机单库单表的数据库服务中，随着流量的增大，我们往往遇到如下问题：

1. 磁盘IO瓶颈：热点数据太多，数据库缓存放不下，每次查询都会产生大量磁盘IO
2. 网络IO瓶颈：请求数据太多，网络带宽不够用
3. CPU瓶颈：单库数据量太大，扫描数据太多
4. 连接瓶颈：单库QPS过高，连接数过多，拆库可以降低QPS压力

mysql单表在超过2千万数据以后，性能会大幅下降

## 1.2分库分表

分库分表分为：

1. 垂直分表：按字段活跃性拆分，将热点数据和非热点数据分开。可以提高数据库缓存可缓存记录数
2. 垂直分库：按业务分为多个库，使得各系统职责更加明确，业务更加清晰。可以解决IO问题，但不能解决单表数据量过大的问题
3. 水平分表：将一个表中数据按某个规则拆分到多张表中。可以解决单表数据量过大问题，但无法解决单库IO瓶颈
4. 水平分库：水平分表难以解决问题时，可以拆分到多个库。可以解决单表数据量过大问题，也可以提高磁盘IO和可承受连接数，但是会大大提高系统复杂度

更多详细分库分表内容可以查看mysql笔记

## 1.3分库分表带来的问题

### 1.3.1 join问题

分库分表以后，数据被分散到不同的表中，因此难以利用Mysql的Join特性，原本一次查询能够完成的业务，现在需要多次查询才能完成。最难的还是水平切分的跨库join

最好的解决方案个人认为还是将join拆分为多个sql执行，多个sql将结果查回来后在内存中组装。如：

SELECT \* FROM t\_class LEFT JOIN r\_class\_student ON t\_class.class\_id = r\_class\_student.class\_id WHERE class\_name =”三年级一班”;

可以改成：

SELECT \* FROM t\_class WHERE class\_name =”三年级一班”;

SELECT \* FROM r\_class\_student WHERE class\_id = “1”;

这样做有很多好处：

1. sql粒度更细，sql可以灵活组装，复用度更高，有利于需求变动后对sql进行修改
2. sql复杂度降低，设计索引更加容易，优化sql更加方便
3. 缓存利用率提高，无论是数据库缓存还是数据库外部缓存，缓存的命中率提高了，如果是join的sql，只要任一条件有一点点改动，都是一条全新的缓存，利用效率太低
4. 可以解决分库分表下跨库join的问题

### 1.3.2分布式事务问题

详见mysql笔记分布式事务处理一节

### 1.3.3 统计、排序、分页、分组问题

业务中常常需要数据统计或者排序等功能，分库分表后统计、排序、分页、分组，聚合函数等操作难以处理。分库分表的中间件会将数据捞取到内存中，在内存中排序和统计，但是当数据量较大时，这种方式难以实现。可以将数据导入到ES或者Hive中去完成数据统计和计算工作

### 1.3.4 数据库连接池问题

存在多数据源管理，假如存在10个数据库实例，和10个应用层，每个应用层连接每个数据库实例的连接池最小16，最大64，那么每个应用层需要管理的数据库连接最小是： 10 \* 16 ，最大是 10 \* 64，每个数据库的连接个数最小是：10 \* 16，最大是：10 \* 64，那么多数据源管理是一个很大的问题。

### 1.3.5 全局ID问题

分库分表以后，数据库的自增主键不再能满足我们的要求。业务对于ID有如下要求：

1. 全局唯一：主键id必须在各分库、分表中保持唯一性
2. 趋势递增：由于mysql存储引擎多使用的是聚簇索引，而聚簇索引对于顺序插入较为友好，因此需要保证插入的主键是有序的
3. 单调递增：保证生成的下一个ID必须大于上一个ID
4. 高可用：ID生成系统必须高可用，否则一旦瘫痪，会引发所有系统瘫痪，并且要保证其高性能

解决方案如下：

1. **UUID方案**

**通用唯一识别码**是一组32位数的16进制数字，其总数极大。若每纳秒产生1百万个 UUID，要花100亿年才会将所有 UUID 用完，若每秒产生10亿笔 UUID ，100年后只产生一次重复的机率是50%

**JDK中内置了生成UUID的方法**

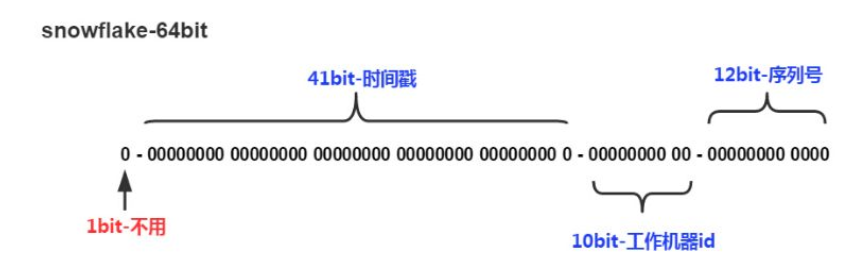
**优点：**

* + 1. 性能非常高：本地生成，没有网络消耗。

**缺点：**

1. 不易于存储：UUID太长，16字节128位，通常以36长度的字符串表示，很多场景不适用。
2. ID作为主键时性能太差，顺序插入会产生大量页分裂，降低性能
3. **类snowflake算法**

Java的long类型是64位的，其范围在-2^63 ~ 2^63 -1之间，即共有2^64位，我们将其划分区段，第1bit不用，其后的41bit用于表示时间，这样共可以表示2^41/(1000L\*3600\*24\*365)，大约69.7年；中间10位表示可以表示2^10=1024台机器，这1024台机器是代表ID在哪台机器上生成的，最多可以有1024台；最后12位将在同一毫秒内的同一节点上从0开始不断累加，最多可累加到4095，一旦这一毫秒过去，所有机器的累加全部清零



这么一来，同一毫秒内，最多可以产生1024\*4096=2^22=4194304个ID，这个数目已经很大了，对于绝大部分应用都是够用的

**优点：**

* + 1. 毫秒数在高位，自增序列在低位，整个ID都是趋势递增的
    2. 不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的
    3. 可以根据自身业务特性分配bit位，非常灵活

**缺点：**

1. 强依赖机器时钟，如果机器上时钟回拨(多数是由服务器校正时间引起)，会导致发号重复或者服务会处于不可用状态
2. 如果毫秒内并发度不高，最后4位为0的概率很高，因此在并发度不高的应用中生成偶数主键的概率更大，容易造成数据分布不均匀的情况发生。在shardingsphere 3.1.0版本中，这个问题已解决

其他方案请看<https://tech.meituan.com/2017/04/21/mt-leaf.html>

### 1.3.6 非partition key的查询问题

水平切分中，对于以分区键为条件的查询可以很好的满足，但实际业务中，查询条件是多种多样的。

### 1.3.7 扩容问题

分库分表扩容问题，在分库分表以后，如果涉及的分片已经达到承受的最大数据量，就需要对集群进行扩容，而且一般是成倍的扩容，通用的扩容方法包括的步骤：

* 按新旧规则，对新旧数据库进行双写
* 将双写前的历史数据按照新的分片规则迁移到新的数据库
* 将旧的分片规则查询改为按新的分片规则查询
* 将双写逻辑下线，按照新的分片规则写入数据
* 删除历史数据

详见<https://zhuanlan.zhihu.com/p/83674503>

<https://www.cnblogs.com/barrywxx/p/11532122.html>

来自淘宝综合业务平台团队，它利用对2的倍数取余具有向前兼容的特性（如对4取余得1的数对2取余也是1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。总得来说，这些方案都不是十分的理想，多多少少都存在一些缺点，这也从一个侧面反映出了Sharding扩容的难度。

### 1.3.8分库分表维度的问题

假如用户购买了商品,需要将交易记录保存取来，如果按照用户的纬度分表，则每个用户的交易记录都保存在同一表中，所以很快很方便的查找到某用户的 购买情况，但是某商品被购买的情况则很有可能分布在多张表中，查找起来比较麻烦。反之，按照商品维度分表，可以很方便的查找到此商品的购买情况，但要查找购买人的交易记录比较麻烦。   
所以常见的解决方式有：

* 1. 通过扫表的方式解决，此方法基本不可能，效率太低了。
  2. 记录两份数据，一份按照用户纬度分表，一份按照商品维度分表。数据冗余问题很难处理，一旦修改数据，则要同时修改冗余数据
  3. 通过搜索引擎解决，但如果实时性要求很高，又得关系到实时搜索。
  4. 多个分区键。我们来看以下解决方案，在线图书系统允许用户对书籍进行评论，一条评论关联了书籍ID和用户ID。我们既需要看到某个用户的所有评论，也需要看到某本书的所有评论。假设我们为用户表和书籍表都进行了分片，分区键分别为userId和bookId，然而评论表则跨越了多个维度。我们可以将评论ID分别存入user表和book表，这种反范式的设计可以帮助我们处理多维度分片的问题，从而避免存储冗余数据。

### 1.3.9 公共表问题

在实际场景中，地理区域表、参数表、字典表等表都属于公共表，他们的特点是数据量有限，变动少，高频查询，一旦出现分库以后，公共表的处理问题也需要注意

可以考虑将这些表在每个分库中都保存一份，对公共表的事务性操作都同时发送到所有分库执行

## 1.4 sharding JDBC和sharding proxy

sharding JDBC是帮助我们做客户端的分库分表，是一个jar包

优点：灵活，支持jdbc的所有数据库

缺点：仅支持java语言,对业务侵入性较大

sharding proxy是做服务端分库分表，它是一个服务节点

优点：对编程语言没有要求，对业务侵入性较小

缺点：仅支持mysql、pgsql等少数sql。增加了节点，复杂度和不稳定性增加

# 数据分片sharding jdbc

sharding jdbc的核心功能是数据分片和读写分离，通过sharding jdbc，应用可以透明地使用JDBC访问已经分库分表、读写分离地多个数据源，而不用关心数据源之间的部署细节

## 2.1 核心概念

**逻辑表：**指一组具有相同逻辑和数据结构表的总称，如：t\_order

**真实表：**在分片数据库中真实存在的数据表，如：t\_order\_0

**数据节点：**指分库分表中不可再分的一个最小数据单元(表)，由数据源名称和数据表组成，如：order\_db\_1.t\_order\_0

**绑定表：**指分片规则一致的两张表，即两张表的分区规则和分区键完全相同，对这两张表进行join查询不会出现笛卡尔积关系。比如：

t\_order和t\_order\_item都通过order\_id分片，且规则一致

如果不设置绑定表，则联表查询会出现：

t\_order\_0 left join t\_order\_item\_0

t\_order\_0 left join t\_order\_item\_1

t\_order\_1 left join t\_order\_item\_0

t\_order\_1 left join t\_order\_item\_1

如果设置了绑定表，则联表查询变为：

t\_order\_0 left join t\_order\_item\_0

t\_order\_1 left join t\_order\_item\_1

**广播表：**也叫公共表。指在所有数据源中都存在的表，且各数据源中表结构和表中数据完全一致，如：字典表

**分片键：**用于分库分表的数据字段

**分片算法：**用于数据分片的算法。目前主要有三种分片算法，支持通过=、>=、<=、<、>、Between、和In分片，具体的分片算法由业务开发者自行根据业务实现，但sharding-jdbc提供三种分片模式：

1. 标准分片算法(StandardShardingAlgorithm)：使用单一分片键进行分片。需配合标准分片策略(StandardShardingStrategy)使用
2. 复合分片算法(ComplexKeysShardingAlgorithm):用于处理使用多个分片键的分片算法，具体实现由开发者实现。需配合复合分片策略(ComplexShardingStrategy)使用
3. Hint分片算法(HintShardingAlgorithm)：用于处理使用Hint行分片的场景。需配合Hint分片策略(HintShardingStrategy)使用

**分片策略：**分片键+分片算法构成完整的分片策略，sharding jdbc通过一个包含分片键的表达式来指定分片策略，如：t\_user\_$->{u\_id%8}标识t\_user表根据u\_id按8取模，分成8张表，表名称从t\_user\_0到t\_user\_7

主要有4种分片策略：

1. 标准分片策略(StandardShardingStrategy)
2. 复合分片策略(ComplexShardingStrategy)
3. Hint分片策略(HintShardingStrategy)
4. 不分片策略(NoneShardingStrategy)

## 2.2 行表达式

在学习如何配置sharding-jdbc之前，首先需要学习行表达式语法。

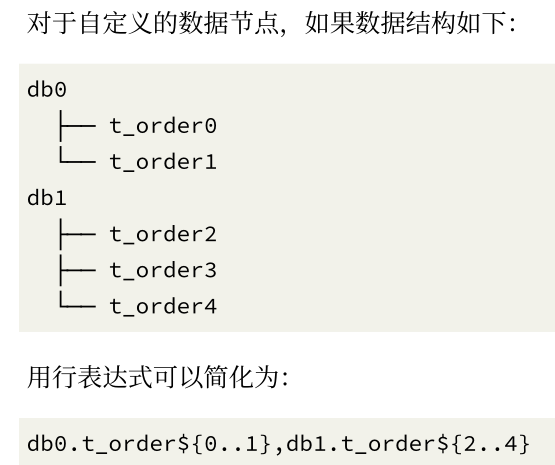
行表达式是为了简化数据节点分片规则的配置而使用的，目前只支持在数据节点配置和分片算法配置中使用，其语法使用Groovy语法：

1. 使用${expression}或$->{expression}来标识这是一个行表达式，但由于前者和spring本身的属性占位符冲突，更建议使用$->{expression}表达式
2. ${begin..end} 表示区间范围，如db$->{0..3}表示：db0 db1 db2 db3
3. ${[enum1,enum2,enum3,enum4]} 表示枚举值，如：db$->{['0','1','2']}表示db0 db1 db2
4. ${expression} 计算表达式，如：db${id % 3}表示db后面的数据按sql中id对3取模进行运算
5. 多个表达式按表达式间的笛卡尔积进行组合，如：${['online','offline']}\_table\_${1..3}最终解析为：

online\_table\_1,online\_table\_2,online\_table\_3和

offline\_table\_1,offline\_table\_2,offline\_table\_3

1. 如图：



1. 对db0.t\_order01 , db0.t\_order02 , db1.t\_order01 , db1.t\_order02

可以使用db${['0','1']}.t\_order0${1..2}

**需要注意的是/在行表达式里面并不是整除，而是除以，如果乱用/可能导致出现db0.25之类的路由出现，如果需要使用整除，需要将A/B改为A.intdiv(B)**

## 2.3 分布式唯一主键

在单库中，我们可以使用数据库为我们生成的唯一自增主键功能，但在多个数据库中，如何生成唯一自增主键成了一个棘手问题。sharding-sphere提供了内置的分布式主键生成器，例如：UUID,SNOWFLAKE雪花算法等，也支持用于自定义实现自增主键生成器

### 2.3.1 UUID方案

采用UUID.randomUUID的方式产生分布式主键

优点：

(1)性能非常高：通过JDK本地生成，没有网络消耗。

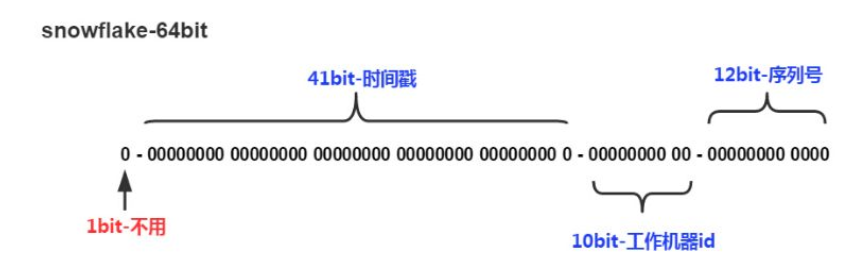
缺点：

(1)不易于存储：UUID太长，16字节128位，通常以36长度的字符串表示，很多场景不适用。

(2)ID作为主键时性能太差，顺序插入会产生大量页分裂，降低性能

### 2.3.2 雪花算法方案

Java的long类型是64位的，其范围在-2^63 ~ 2^63 -1之间，即共有2^64位，我们将其划分区段，第1bit不用，其后的41bit用于表示时间，这样共可以表示2^41/(1000L\*3600\*24\*365)，大约69.7年；中间10位表示可以表示2^10=1024台机器，这1024台机器是代表ID在哪台机器上生成的，最多可以有1024台；最后12位将在同一毫秒内的同一节点上从0开始不断累加，最多可累加到4095，一旦这一毫秒过去，所有机器的累加全部清零



这么一来，同一毫秒内，最多可以产生1024\*4096=2^22=4194304个ID，这个数目已经很大了，对于绝大部分应用都是够用的

优点：

(1)毫秒数在高位，自增序列在低位，整个ID都是趋势递增的

(2)不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的

(3)可以根据自身业务特性分配bit位，非常灵活

缺点：

(1)强依赖机器时钟，如果机器上时钟回拨(多数是由服务器校正时间引起)，会导致发号重复或者服务会处于不可用状态

//采用ip地址生成二进制hostNumber，即0000000000（二进制）~1111111111（二进制）,即范围从0~1023  
private String hostNumber;  
//上次生成id的时间戳，防止服务器时间回退  
private long lastTimeStamp = -1L;  
private final long maxNum = 4095;//4095

public long generateId() {  
 long currentTime;  
 synchronized (this) {  
 currentTime = System.*currentTimeMillis*();  
 if (currentTime < lastTimeStamp) {  
 //发生服务器时钟回拨了，抛出异常  
 throw new RuntimeException("服务器时间发生回退，id生成异常请重试");  
 }  
  
 if (currentTime == lastTimeStamp) {  
 if (num == maxNum) {  
 //达到最大值了，阻塞到下一毫秒  
 while (currentTime <= lastTimeStamp) {  
 currentTime = System.*currentTimeMillis*();  
 }  
 //到下一秒之后，num置为0  
 num = 0;  
 }  
 num++;  
 } else {  
 //当前时间已经不是上一次生成id的毫秒时间了，需要将num置0  
 num = 0;  
 }  
 // log.info(currentTime + " | " + num);  
 //生成完毕，修改上次生成id的时间戳  
 lastTimeStamp = currentTime;  
 }  
 String idBinaryStr = "0" + Long.*toBinaryString*(currentTime) + hostNumber + addZero(Integer.*toBinaryString*(num), 12);  
 *log*.info("" + Long.*parseLong*(idBinaryStr, 2));  
 return Long.*parseLong*(idBinaryStr, 2);  
}

## 2.4 内核解剖

### 2.4.1 结果归并

数据从各分库(表)查询回来以后，需要对其进行归并，将多个结果集合并成为一个结果集，sharding sphere提供了如下几种归并方式：

#### 2.4.1.1 流式归并

流式归并是一种利用数据库游标技术，避免一次将结果集全部加载到应用服务器内存中的优化技术。它可以通过游标下移的方式逐条归并数据，节省应用服务器内存资源。由于需要持有连接等待游标移动，因此这种模式更耗费数据库连接资源，但可以节约客户端服务器的内存

#### 2.4.1.2 内存归并

指一次性将所有分库分表的结果集全部传输加载到客户端服务器的内存中，这样更节约数据库连接，但对于应用服务器内存消耗更加严重

#### 2.4.1.3 装饰者归并

装饰者归并是对流式归并或者内存归并的统一功能增强，比如归并时、需要聚合sum前，在进行聚合计算前，都需要通过流式归并或内存归并查出结果集，查出结果集后还需要进行聚合计算的归并，这就是一种装饰者归并

### 2.4.2 分页性能

#### 2.4.2.1 全局视野法的性能瓶颈

对于查询偏移量offset过大的分页查询会导致数据库性能低下，比如：

SELECT \* FROM t\_order order by createTime LIMIT 1000000,10

这条sql在假设有两张分表的情况下被改写为：

SELECT \* FROM t\_order\_0 order by createTime LIMIT 1000010,10

SELECT \* FROM t\_order\_1 order by createTime LIMIT 1000010,10

这意味着将有2 \* 1000010条数据记录被传输到应用服务器，加重了应用服务器的内存压力和网络消耗压力

#### 2.4.2.2 sharding sphere的优化

sharding sphere对此进行了优化

1. 流式处理+归并排序

流式归并(参见内核解剖)会逐条从各结果集中取出数据，因此不会对内存造成压力，由于各个结果集都是局部有序的，因此最适合采用归并排序，由于只需要合并即可，并不需要拆分成子数据再排序，因此归并排序的时间复杂度进化为O(n)

1. 对于仅落到单分片的查询不改写sql

**优化后消除了内存压力，但网络消耗压力依然无法解除**

## 2.5 快速开始

### 2.5.1 快速开始

1.引入依赖包：

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

2.编写配置文件：

**spring:  
 shardingsphere:  
 datasource:  
 names:** db0,db1 *# 指定数据源名字，多数据源以逗号分隔* **db0:  
 type:** com.alibaba.druid.pool.DruidDataSource  
 **url:** jdbc:mysql://127.0.0.1:3306/kjyq\_shard\_1?useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=Asia/Shanghai  
 **driver-class-name:** com.mysql.jdbc.Driver  
 **username:** root  
 **password:** 123456  
  
 **db1:  
 type:** com.alibaba.druid.pool.DruidDataSource  
 **url:** jdbc:mysql://127.0.0.1:3306/kjyq\_shard\_2?useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=Asia/Shanghai  
 **driver-class-name:** com.mysql.jdbc.Driver  
 **username:** root  
 **password:** 123456  
  
 **sharding:  
  
 tables:** *#　具体分片规则配置* **t\_chat\_record:** *#逻辑表名* **key-generator-column-name:** id *# 主键* **key-generator:** *# 使用雪花算法id* **column:** id  
 **type:** SNOWFLAKE  
 **actual-data-nodes:** db$->{0..1}.t\_chat\_record *# 实际的数据节点(数据源名.逻辑表名)  
 # 分库规则* **database-strategy:  
 inline:** *# 行表达式* **sharding-column:** id  
 **algorithm-expression:** db$->{id % 2}  
 *# 分表规则  
# table-strategy:  
 # inline: # 行表达式  
# sharding-column: id  
# algorithm-expression: t\_chat\_record->{id % 2} # 按模运算分配  
# sys\_user: # 非设置为默认数据库的，数据库中不需要分表的可以这么配，表多时候会很麻烦，后面会讲一个配置多数据源的方法，更为简便  
# actual-data-nodes: db1.sys\_user  
# table-strategy:  
# none:  
# any: none  
 # 绑定表规则，即逻辑表名。(例如t\_user代表了t\_user\_0, t\_user\_1, t\_user\_2 ...)* **binding-tables:** t\_chat\_record  
 **props:  
 sql.show:** true *# 是否开启SQL显示*

这里仅进行分库，将数据库分为kjyq\_shard\_1和kjyq\_shard\_2，两个库中各有一张t\_chat\_record表

CREATE TABLE `t\_chat\_record` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT,

`case\_record\_id` bigint(20) DEFAULT NULL,

`case\_id` bigint(20) DEFAULT NULL,

`type` tinyint(4) DEFAULT NULL,

`content` varchar(500) DEFAULT NULL,

`account` varchar(50) DEFAULT NULL,

`time` timestamp(6) NOT NULL DEFAULT current\_timestamp(6) ON UPDATE current\_timestamp(6),

`name` varchar(50) DEFAULT NULL,

`create\_time` timestamp(6) NOT NULL DEFAULT '0000-00-00 00:00:00.000000',

`update\_time` timestamp(6) NOT NULL DEFAULT '0000-00-00 00:00:00.000000',

`create\_by` varchar(50) DEFAULT NULL,

`update\_by` varchar(50) DEFAULT NULL,

`deleted` tinyint(4) DEFAULT 0,

PRIMARY KEY (`id`)

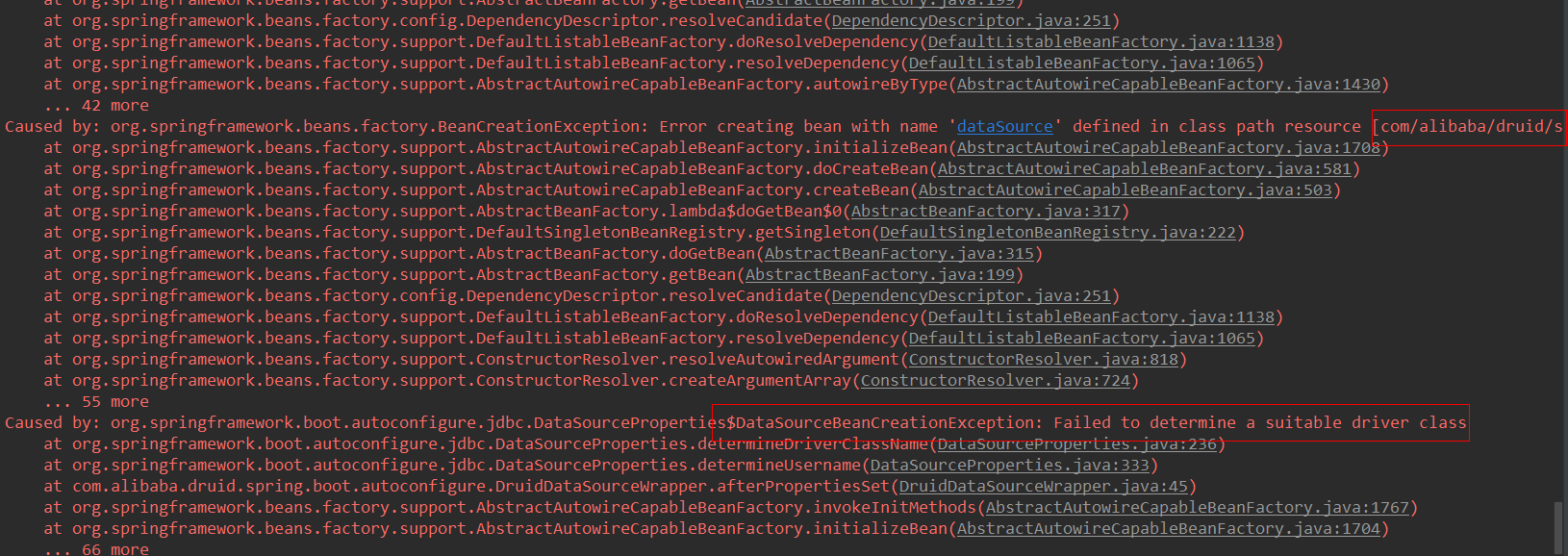
) ENGINE=InnoDB AUTO\_INCREMENT=0 DEFAULT CHARSET=utf8mb4;

然后正常编写相应的dao操作方法即可

### 2.5.2 druid的坑

这里踩到一个大坑，报错信息为：

DataSourceProperties$DataSourceBeanCreationException: Failed to determine a suitable driver class

 这里是说druid找不到合适的mysql驱动，但是我们的驱动包和配置都是正常的。跟进源码debug发现dataSource的各项属性都为null，明显是druid找不到我们的yml配置引起的。

在正常项目中，我们配置datasource的yml格式为：

**spring:  
 datasource:  
 url:**

而使用了sharding-jdbc后变为：

**spring:  
 shardingsphere:  
 datasource:  
 names:** db0,db1 *# 指定数据源名字，多数据源以逗号分隔* **db0:  
 type:**

所以无法获取对应的yml属性，那么根源是在于我们用了druid和springboot的整合包，因为sharding-jdbc已经和springboot整合生成datasosurce了，我们在配置文件中就不会为druid-spring-boot-starter再配置datasource，读取不到yml后，druid就会报错：只需要将maven依赖从：

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid-spring-boot-starter</artifactId>  
 <version>${druid-spring-boot-starter.version}</version>  
 </dependency>

改为：

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.1.16</version>  
 </dependency>

即可

## 2.6 配置文件

sharding-jdbc支持java，YAML，spring命名空间和spring boot starter四种方式配置，这里仅介绍通过YAML方式来配置

sharding-jdbc的配置主要分为三大块：

1. 数据源集合配置(datasources)
2. 规则集合(sharding)
3. 属性配置(props)

配置时需要注意以下几点：

1. 如果数据库中只有部分数据分库分表，仍需要将不分库分表的表也配置在分片规则中，因为sharding-jdbc仍需要为部分库分表的数据寻找合适的数据源datasource。但是sharding-jdbc提供了两种配置方式来简化配置：
2. 配置default-data-source：凡是在默认数据源中的表可以无需配置在分片规则中，sharding-jdbc将会在找不到分片数据源的情况下将表路由至默认数据源
3. 将不参与分库分表的数据独立于sharding-jdbc之外，在应用中使用多个数据源分别处理分片和不分片的情况

**2.6.1 YAML配置**

spring:  
 shardingsphere:  
 datasource:  
 names: db0,db1 *# 指定数据源名字，多数据源以逗号分隔* db0:  
 type: com.alibaba.druid.pool.DruidDataSource *# 连接池类型* url: jdbc:mysql://127.0.0.1:3306/kjyq\_shard\_1?characterEncoding=utf-8&useUnicode=true&useSSL=false&serverTimezone=Asia/Shanghai  
 driver-class-name: com.mysql.jdbc.Driver  
 username: root  
 password: 123456  
  
 db1:  
 type: com.alibaba.druid.pool.DruidDataSource  
 url: jdbc:mysql://127.0.0.1:3306/kjyq\_shard\_2?characterEncoding=utf-8&useUnicode=true&useSSL=false&serverTimezone=Asia/Shanghai  
 driver-class-name: com.mysql.jdbc.Driver  
 username: root  
 password: 123456  
  
 sharding:  
 tables:  
 t\_chat\_record:  
 *#sharidng 表对应的数据源以及物理名称,需要用表达式处理,表示表实际上在哪些数据源存在* actual-data-nodes: db${0..1}.t\_chat\_record\_${1..4}  
 *# 数据源的分片策略（分库策略）* databaseStrategy:  
 *#standard: #用于单分片键，支持单个分片键的精确分片 或 范围分片  
 #sharding-column: #仅支持单个分片键*

*#指定 精确分片算法实现类(=和in)，该类需要实现PreciseShardingAlgorithm接口  
 #precise-algorithm-class-name: 如com.cetc52.ycsx.sharding.LogStandardPreciseShardingAlgorithm*

*#指定 范围分片算法实现类(between)，该类需要实现RangeShardingAlgorithm接口  
 #range-algorithm-class-name:   
 #complex: #用于多分片键  
 #sharding-column: #支持多个分片键，多个键用逗号分隔  
 #algorithm-class-name: #指定 复合分片算法实现类，该类需要实现ComplexKeyShardingAlgorithm接口  
  
 #hint: #用于hint分片，不需要指定分片键，适用于分片字段非sql字段决定的情况  
 #algorithm-class-name: #指定 hint分片算法实现类，该类需要实现HintShardingAlgorithm接口  
  
 #inline 用于行表达式，使用groovy表达式简化分片，提供对sql中=和in的分片操作支持，仅支持单分片键，相比于standard和complex不用写实现类了  
 #sharding-column: #仅支持单个分片键  
 #algorithm-expression: # 数据源分片表达式,根据指定的表达式计算得到需要路由到的数据源名称,这里想用整除不能用/，否则出现小数，得用intdiv方法* inline:  
 *# 数据源分片列名称* sharding-column: case\_record\_id  
 *# 数据源分片表达式,根据指定的表达式计算得到需要路由到的数据源名称,这里想用整除不能用/，否则出现小数，得用intdiv方法* algorithm-expression: db$->{(case\_record\_id % 8).intdiv(4)}  
 *# 数据表的分片策略（分表策略）* tableStrategy:  
 inline:  
 *# 数据表分片列名称* shardingColumn: case\_record\_id  
 *# 数据表分片表达式,根据指定的表达式计算得到需要路由到的数据源名称* algorithmExpression: t\_chat\_record\_$->{case\_record\_id % 4 + 1}  
 *# 自动生成分布式主键，使用自动生成主键的话，xml的insert语句中就不能出现主键列，否则不会生成主键* key-generator:  
 *#主键列名* column: id  
 *#主键生成策略* type: SNOWFLAKE  
  
 t\_chat\_info:  
 actualDataNodes: db${0..1}.t\_chat\_info\_${1..4}  
 databaseStrategy:  
 inline:  
 shardingColumn: case\_record\_id  
 algorithm-expression: db$->{(case\_record\_id % 8).intdiv(4)}  
 tableStrategy:  
 inline:  
 shardingColumn: case\_record\_id  
 algorithmExpression: t\_chat\_info\_$->{case\_record\_id % 4 + 1}  
 key-generator:  
 column: id  
 type: SNOWFLAKE  
  
 bindingTables: *# 绑定表，多个表之间用逗号相隔,和tables标签同一级* - t\_chat\_record,t\_chat\_info  
  
 *# 广播表 对于广播表发生的所有数据变更,都会不经sharidng处理,而是直接发送到所有数据节点* broadcastTables:  
 - t\_config  
  
 *#不需要分库分表的表配置在这里，项目中常常只是部分大表需要分片，而不是所有的表都需要拆分  
 #这里指定需要分表的sql都去该数据源即可，sharding-jdbc将会在找不到分片数据源的情况下将表路由至默认数据源* default-data-source-name: db0  
  
 props:  
 *# 调试时使用，打印sql改写和路由至数据源的详细信息打印出来，5.x版本后sql.show参数调整为sql-show* sql.show: true

## 2.7 sql支持度

由于sql语法灵活复杂，虽然sharding-jdbc尽力提供相应的sql支持，但由于分布式数据库和单库场景的差异，难免会出现不支持的sql

### 2.7.1 支持项

#### 2.7.1.1 路由至单数据节点的支持项

100%兼容

#### 2.7.1.2 路由至多数据节点的支持项

全面支持DML(增删改查)、DDL(数据库定义语言，如create/alter/drop table/index等)、DCL(数据库控制语句，如grant、revoke等)、TCL(事务控制语句)、和部分DAL(数据访问层，如读写分离)，支持分页、去重、排序、分组、聚合、关联查询

### 2.7.2 不支持项

#### 2.7.2.1 路由至多数据节点的不支持项

1. 部分支持case when

case when中包含子查询则不支持

case when中使用逻辑表名则不支持(请使用表别名)

1. 不支持union 以及union all
2. 部分支持子查询。子查询和外层查询同时指定分片键值时，分片键值必须保持一致，否则难以路由。eg：

支持SELECT COUNT(1) FROM (select \* from t\_order where order\_id=1) AS o WHERE o.order\_id =1

不支持SELECT COUNT(1) FROM (select \* from t\_order where order\_id=1) AS o WHERE o.order\_id =2

1. 不支持包含schema的sql

### 2.7.3 无法精确路由的情形

将分片键放在函数或表达式中会导致全路由，即发送到每一个分库(表)中执行

如以create\_time进行分片

SELECT \* FROM t\_order WHERE to\_date(create\_time,`yyyy-mm-dd`)=`2020-01-01`;

这条sql将无法确定数据在哪个分片中

# 分布式事务

# 读写分离sharding-jdbc

和数据分片不同，读写分离的配置是通过sql语义分析，将读操作和写操作分别路由至主库和从库。将数据分片和读写分离联合使用，可以大幅提高系统性能

## 4.1 核心概念

**主库：**事务性操作使用的数据库，**目前只支持单主库**

**从库：**查询数据所使用的库，可支持多从库

**主从同步：**将主库的数据异步地同步到从库的操作，由于是异步操作，所以只能保证最终一致性，主库复制到从库可以使用mysql的bin log和relay log来实现

**负载均衡策略**：采取一定策略将查询请求均匀疏导至不同从库

## 4.2 配置文件

*# 切记： 首先要开启 数据库数据同步功能 mysql： 开启mysql-binlog日志 mysql开启主从同步配置：https://www.cnblogs.com/songwenjie/p/9371422.html  
# 配置过程中可能会出现一些问题 ，细心查看 其中一个问题的解决方案 ： https://blog.csdn.net/weixin\_37998647/article/details/79950133  
#sharding-jdbcReadWrite 实现读写分离的配置 master/slave master 主要写数据 slave实现读数据 这是sharding-jdbc 通过执行的sql 来判断的*spring:  
 main:  
 allow-bean-definition-overriding: true *# 当遇到同一个实体类同时映射多个相同的表名字的时候，是否允许覆盖注册* shardingsphere:  
 *# 配置多个数据源* datasource:  
 names: ds-master0,ds-slave0,ds-master1,ds-slave1  
 ds-master0:  
 type: com.alibaba.druid.pool.DruidDataSource  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://127.0.0.1:3307/jdbc\_master?useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=Asia/Shanghai  
 username: root  
 password: 123456  
 initial-size: 4 *# 初始化时建立物理连接的个数* min-idle: 1 *# 最小连接池数量* max-active: 40 *# 最大连接数量* query-timeout: 60000 *#查询超时* filters: stat,config *# stat表示sql合并* testOnBorrow: false *# 申请连接时执行validationQuery检测连接是否有效* ds-slave0:  
 type: com.alibaba.druid.pool.DruidDataSource  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://127.0.0.1:3308/jdbc\_master?useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=Asia/Shanghai  
 username: root  
 password: 123456  
 initial-size: 4 *# 初始化时建立物理连接的个数* min-idle: 1 *# 最小连接池数量* max-active: 40 *# 最大连接数量* query-timeout: 60000 *#查询超时* filters: stat,config *# stat表示sql合并* testOnBorrow: false *# 申请连接时执行validationQuery检测连接是否有效  
# 此处省略ds-master1,ds-slave1两个数据源的配置  
  
 # 配置读写分离规则* masterslave:  
 ds-master0:

*#主库 多个逗号隔开* master-data-source-name: ds-master0

*# 从库 多个逗号隔开 ds\_slave0,ds\_slave1 多个从库 读时采用 下面的配置方式读数据* slave-data-source-names: ds-slave0

*#从库负载均衡算法类型，可选值：ROUND\_ROBIN，RANDOM。(循环/随机) 若`loadBalanceAlgorithmClassName`存在则忽略该配置*

loadBalanceAlgorithmType: ROUND\_ROBIN

ds-master1: master-data-source-name: ds-master1 *#主库 多个逗号隔开* slave-data-source-names: ds-slave1 *# 从库 多个逗号隔开 ds\_slave0,ds\_slave1 多个从库 读时采用 下面的配置方式读数据* loadBalanceAlgorithmType: ROUND\_ROBIN *#从库负载均衡算法类型，可选值：ROUND\_ROBIN，RANDOM。(循环/随机) 若`loadBalanceAlgorithmClassName`存在则忽略该配置* props:  
 sql.show: true *# 显示sql*

## 4.3 支持项

1. **提供一主多从读写分离配置，可单独使用，也可以配合分库分表一同使用**
2. **同一线程且同一数据库连接内，如果有事务操作，以后读操作均从主库读取，防止因复制延迟导致读取不到自己的修改**

## 4.4 不支持项

1. **主库和从库的数据同步（由数据库实现）**
2. **主库和从库数据同步导致数据延迟不一致**
3. **多主库双写或多写**

# 弹性伸缩组件sharding-scaling

对于原本单数据库的系统，如何将数据迁移到分片后的数据库中，一直依赖是一个比较难处理的问题，对于已经分片的数据库，随着业务量的快速增加，如何对现有的分片集群进行弹性扩容或缩容也是一个难题。从4.1.0版本开始，sharding sphere提供了shardingSphere-Scaling作为用户数据迁移及弹性伸缩的解决方案

## 5.1 挑战和目标

sharding sphere在分片策略和算法上完全由用户自己定义，给与了用户最大的灵活性，如此高的灵活度给弹性伸缩带来了很大的困难。如何找到一种方式能够支持各类不同的分片策略和算法，又能高效地伸缩数据节点成为一个难题

如何在伸缩过程中，做到不影响业务，尽量减少数据不可用的时间，甚至做到用户无感知，是弹性伸缩的另一个挑战

最后弹性伸缩如何保证数据的可用性和正确性，是第三个挑战

## 5.2 原理

sharding sphere使用的弹性伸缩解决方案为：临时地使用两组数据库集群，在伸缩完成后切换



采用这种方式，有一定优缺点：

优点：

1. 伸缩过程不会对老库造成影响
2. 伸缩失败无风险
3. 不受分片策略限制

缺点：

1. 在一定时间内需要额外的服务器
2. 所有数据都需要移动

弹性伸缩模块会通过解析旧分片规则，提取配置中的数据源、数据节点等信息，创建伸缩工作流，一次弹性伸缩主要分为以下四个步骤：

1. **准备阶段**
2. **存量数据迁移阶段**
3. **增量数据同步阶段**
4. **规则切换阶段**

## 5.3 伸缩流程

### 5.3.1 准备阶段

准备阶段，弹性伸缩模块会进行数据源连通性及权限校验，同时进行存量数据统计、日志位点记录，最后根据数据量和用户设置的并行度进行任务拆分



### 5.3.2 存量数据迁移阶段

执行在准备阶段拆分好的存量数据迁移作业，存量迁移采用JDBC查询方式，直接从数据节点中读取数据，并使用新规则写入新集群



### 5.3.3 增量数据同步阶段

由于存量数据迁移需要耗费相当长的时间，所以需要对在此阶段内新转增的数据进行同步。不同数据库使用的技术不同，单总体上都是基于数据库的复制功能来实现对变更数据的捕获

MYSQL：订阅并解析binlog

PostgreSql：采用官方逻辑复制test\_decoding

捕获这些增量修改后，由弹性伸缩模块根据新规则写入到新数据节点。当增量数据基本同步完成时，进入规则切换阶段（由于业务系统未停止，此时仍不断有新的数据变动，但没关系，我们会在下个阶段继续处理）



### 5.3.4 规则切换阶段

在此阶段，可能存在一定时间的业务数据只读窗口期，通过设置数据库只读或者sharding sphere的熔断机制，让旧数据节点的数据暂时不可修改，确保增量同步完全完成

这个窗口期短则数秒，长则数分钟，取决于用户量。确认完成后，sharding sphere可通过配置中心修改配置，将业务重新导向新规则集群，弹性伸缩完成



# 影子库压测

基于微服务分布式架构下，由于整体服务是需要通过一些列微服务调用，中间件调用来完成业务的，因此对单个微服务的压测已经不能代表真实场景，而完全再搭建一套压测环境需要完全复制生产环境的服务，中间件等环境，还要保证硬件等各项因素与生成环境一致，成本太高。在这种场景下，业内通常选择全链路压测的方式，即在生产环境压测，但这样一来，测试数据就会被打入生产数据库，对真实数据造成污染，sharding sphere提供了影子库压测功能，将数据分流至生产库和测试影子库，确保压测真实可靠

## 分页优化

跨库分页（写到mysql笔记中）

1. **全局视野法：**
2. 将order by time desc offset (x-1)\*y limit y改写为order by time desc offset 0 limit x\*y
3. 将order by time desc offset 0 limit x\*y发送到各分库(表)执行，即每个分库(表)都获取前x页的数据
4. 将数据在服务端内存中汇总，共得到n\*(x\*y)条数据
5. 服务端对得到的数据进行内存排序，取到第x页数据返回给用户

优点：

实现简单，不需要在业务上做限制

缺点：

随着用户选择的页数(x)越大，n\*(x\*y)越大，排序越耗时耗CPU，给服务端内存造成的压力也越大，甚至可能出现oom，而且加大了网络传输的带宽损耗

1. **禁止跳页(sharding-jdbc不能用)**
2. 前端做限制，不允许任意跳页，只允许用户点击下一页或者上一页
3. 每次点击要求前端传入本页中order by字段的最大(小)值

**上一页：**各分库(表)执行where time > #{maxTime} order by time **asc** limit y，得到比第x+1页最大值maxTime更大的最后y条数据，共得到n\*y条数据。在内存中对n\*y条数据进行内存排序，将第x页数据返回给用户。上一页比较难做，可以考虑只做下一页，然后将走过的每一页都缓存到redis中去

**下一页：**各分库(表)执行where time < #{minTime} order by time **desc** limit y，得到比第x-1页最小值minTime更小的前y条数据，共得到n\*y条数据。在内存中对n\*y条数据进行内存排序，将第x页数据返回给用户

优点：

1. 不会随着x增大降低查询效率
2. 降低了内存压力和带宽压力

缺点：

1. 业务端需要限制，不允许随意设置x值跳页，只能一次次点击下一页进行浏览
2. 实现起来复杂度较高