数据库分库分表之sharding-jdbc

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 作者 | 备注 |
| v1.0 | 2018/08/22 | christian | create |
|  |  |  |  |
|  |  |  |  |

目录

[参考资料 1](#_Toc522719781)

[为什么分库 1](#_Toc522719782)

[解决思路 2](#_Toc522719783)

[解决方案1：硬件+商用数据库 2](#_Toc522719784)

[解决方案2：开源数据库集群版本 2](#_Toc522719785)

[解决方案3：非关系型数据库代替 关系型数据库 2](#_Toc522719786)

[解决方案4：数据分片 2](#_Toc522719787)

[sharding-jdbc 3](#_Toc522719788)

[sharding-jdbc里边的核心概念 3](#_Toc522719789)

[shardingjdbc里边的核心流程（详见sharding-jdbc官方文档） 5](#_Toc522719790)

[数据治理 6](#_Toc522719791)

[使用方式 6](#_Toc522719792)

[shardingjdbc的不支持项 12](#_Toc522719793)

[分布式事务 12](#_Toc522719794)

[总结 14](#_Toc522719795)

## 参考资料

sharding-jdbc官方文档 ： <http://shardingsphere.io/>

测试源码地址 ：https://github.com/likemytea/demoshardingjdbc

rabbitmq 官网 ：http://www.rabbitmq.com/

## 为什么分库

一般普遍的数据存储方案为：核心关联数据集中存储到单一物理节点上（另外：不相关联的表可以拆分到不同的物理节点，即数据垂直拆分，比如将日志数据和核心数据拆分到不同的库存储到不同的物理节点上），灾备使用数据异步备份的方式如mysql的binary log方案，高并发访问请求的情况下，可以为数据库配置一主多从的部署策略解决并发查询引起的数据库性能瓶颈。

如上的数据方案可以应对大部分的业务需求，但当单表数据规模达到一定阀值，大多的开源数据库会随着索引高度的增加使得磁盘访问的IO次数增加，进而导致查询性能的大幅下降，更新的效率下降较查询更加明显。一言概括之就是：**关系型数据库数据量过大导致的性能瓶颈**

解决思路大概有以下几方面：

## 解决思路

### 解决方案1：硬件+商用数据库

硬件从普通的PC server用unix的小型机代替，数据库从mysql，postgresql用oracle，DB2代替，但我们大家都知道当今互联网时代这种方案是逆潮流而动的，先不论成本，单单是小型机服务器的运维人才和商用数据库的运维人员我们就很难找到，更何况要说服各位的boss掏腰包把钱投在服务器和软件提供商身上，所以这个方案不是本节讨论点，而且本人对小型机也不是很熟

### 解决方案2：开源数据库集群版本

使用如：mysql cluster 的社区版

缺点是没有发现有企业在生产环境使用，我想根本原因是里边的源码和技术特性对我们来说太复杂，大多数人无法掌控。所以这个方案不是本节讨论重点

### 解决方案3：非关系型数据库代替 关系型数据库

对于nosql类数据库，实际业务场景中只能在部分业务中取代关系型数据库。典型的开源组件有hadoop旗下的hbase和mongodb

### 解决方案4：数据分片

将集中于单一节点的数据拆分并分别存储到多个数据库或表，称为分库分表。 分库可以有效分散高并发量，分表虽然无法缓解并发量，但仅跨表仍然可以使用数据库原生的ACID事务（原子事务）。而一旦跨库，涉及到事务的问题就会变得无比复杂。

分库分表一般有两种拆分方式。垂直拆分和水平拆分

#### 垂直拆分：

按照业务拆分的方式称为垂直拆分。例如，将用户库和订单库拆分到不同的数据库中。垂直拆分可以缓解数据量和访问量带来的问题，但无法根治，如果垂直拆分之后的用户和订单数量依然超过单节点所能承载的阈值，则需要水平拆分来进一步处理。

#### 水平拆分：

将一个表中的数据按照一定的业务规则拆分至不同表和数据库中，称之为水平拆分。

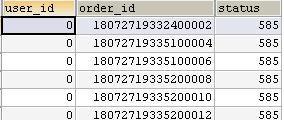
这也是接下来本文的中心。本文档介绍的开源sharding-jdbc

## sharding-jdbc

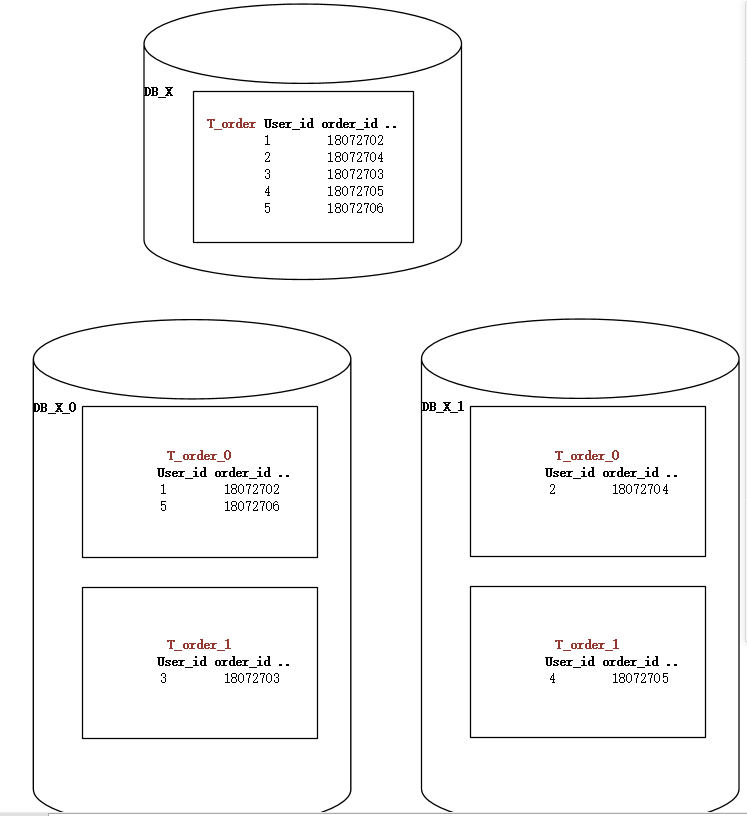
### sharding-jdbc里边的核心概念

介绍下面概念前我们先模拟一个业务场景，假设一个电商系统数据库，里边有一亿+数据的订单表（t\_order），我们对其分片:

**t\_order**



想要实现的分片:按照user\_id取模分库。另外按照order\_id取模分表，则效果如下图：



逻辑表: 拆分前的表的名字，上例中的t\_order

真实表: 在分片的数据库中真实存在的物理表 ，上例中的 t\_order\_0 t\_order\_1

数据节点: 上例中的 db\_x\_0.t\_order\_0 db\_x\_1.t\_order\_1 等等

绑定表：指在任何场景下分片规则均一致的主表和子表。如上例中的t\_order,如果再有另外一个表，它的分库分表策略完全和t\_order一致，则它们为绑定表关系，我们在实际使用过程中必须设置它，**否则sharding-jdbc的源码会做笛卡儿积查询处理，导致降低查询性能我们还不知到哪里出了问题**。

逻辑索引：某些数据库（如：**PostgreSQL**）不允许同一个库存在名称相同索引，某些数据库（如：MySQL）则允许只要同一个表中不存在名称相同的索引即可。逻辑索引用于同一个库不允许出现相同索引名称的分表场景，需要将同库不同表的索引名称改写为索引名 + 表名，改写之前的索引名称成为逻辑索引。

### shardingjdbc里边的核心流程（详见sharding-jdbc官方文档）

**sql解析**：分为词法解析和语法解析。先通过词法解析将SQL拆分为一个个不可再分的单词。再使用语法解析器对SQL进行理解，并最终提炼出解析上下文。解析上下文包括表、选择项、排序项、分组项、聚合函数、分页信息、查询条件以及可能需要修改的占位符的标记

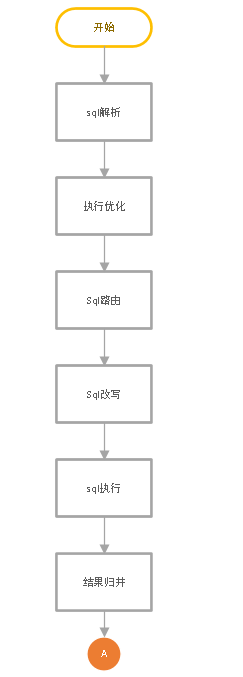
执行器优化：合并和优化分片条件，如OR等

sql路由: 目前支持分片路由、Hint路由、广播路由、单播路由以及阻断路由等方式。

sql改写: 将SQL改写为在真实数据库中可以正确执行的语句。

sql执行: 通过多线程执行器异步执行，但同一个物理数据源的不同分表的SQL会采用同一连接的同一线程，以保证其事务的完整性。<--**性能的提升就在于此**

结果归并: 将多个执行结果集归并以便于通过统一的JDBC接口输出。



### 数据治理

shardingjdbc可以使用zookeeper做服务治理（服务即指的数据库实例服务）

我们这里使用zookeeper治理从库的熔断及可用性上

原理大致是：从库在实例在可用时将其自身数据源并服务器里的PID注册到zookeeper临实节点上，这里zookeeper的作用主要是用来维护和监控你存储的数据的状态变化。通过监控这些数据状态的变化，从而可以达到对这些从库的集群管理。

#### 数据治理使用方式

// 配置基于Zookeeper的注册中心

ZookeeperConfiguration zkConfig = new ZookeeperConfiguration();

zkConfig.setServerLists("localhost:2181");

zkConfig.setNamespace("sharding-sphere-orchestration");

// 配置数据治理

OrchestrationConfiguration orchConfig = new OrchestrationConfiguration("orchestration-sharding-data-source", zkConfig, false, OrchestrationConfiguration.SHARDING);

// 获取数据源对象

DataSource dataSource = OrchestrationShardingDataSourceFactory.createDataSource(dataSourceMap, shardingRuleConfig, new ConcurrentHashMap(), new Properties(), orchConfig);

### 使用方式

#### 引入依赖包

<!-- sharding jdbc start -->

<dependency>

<groupId>io.shardingsphere</groupId>

<artifactId>sharding-jdbc</artifactId>

<version>3.0.0.M1</version>

</dependency>

<!-- sharding jdbc end -->

#### 配置系统初始化类(详见源码)

package com.chenxing.demoshardingjdbc.config;

import java.sql.SQLException;

import java.util.Arrays;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import com.alibaba.druid.pool.DruidDataSource;

import com.chenxing.demoshardingjdbc.dao.base.MyJdbcTemplate;

import com.google.common.collect.Lists;

import io.shardingsphere.core.api.ShardingDataSourceFactory;

import io.shardingsphere.core.api.config.MasterSlaveRuleConfiguration;

import io.shardingsphere.core.api.config.ShardingRuleConfiguration;

import io.shardingsphere.core.api.config.TableRuleConfiguration;

import io.shardingsphere.core.api.config.strategy.InlineShardingStrategyConfiguration;

/\*\*

\* Description:

\*

\* @author liuxing

\* @date 2018年7月23日aaa

\* @version 1.0

\*/

@Configuration

public class DBShardingConfig {

public DruidDataSource createDefaultDruidDataSource() {

DruidDataSource druidDataSource = new DruidDataSource();

druidDataSource.setMaxWait(60000l);

druidDataSource.setMaxActive(5);

druidDataSource.setInitialSize(1);

druidDataSource.setMinIdle(1);

druidDataSource.setTimeBetweenEvictionRunsMillis(3000l);

druidDataSource.setMinEvictableIdleTimeMillis(300000l);

druidDataSource.setConnectionProperties("druid.stat.slowSqlMillis=3000");

druidDataSource.setValidationQuery("SELECT 'x'");

druidDataSource.setTestWhileIdle(true);

druidDataSource.setTestOnBorrow(false);

druidDataSource.setTestOnReturn(false);

return druidDataSource;

}

@Bean(name = "shardingDataSource", destroyMethod = "close")

@Qualifier("shardingDataSource")

public DataSource getShardingDataSource() throws Exception {

// 配置真实数据源

Map<String, DataSource> dataSourceMap = createDataSourceMap();

// 配置库分片和表分片规则

ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration();

shardingRuleConfig.getTableRuleConfigs().add(getTableRuleConfig());

// 配置默认的数据源名称（如果不配置此项，没有分库的表在插入数据库的时候，就不知路由到哪个数据库）

shardingRuleConfig.setDefaultDataSourceName("ds");

// 配置读写分离

shardingRuleConfig.setMasterSlaveRuleConfigs(getMasterSlaveRuleConfiguration());

DataSource dataSource = null;

try {

dataSource = ShardingDataSourceFactory.createDataSource(dataSourceMap, shardingRuleConfig,

new HashMap<String, Object>(), new Properties());

} catch (SQLException e) {

e.printStackTrace();

}

return dataSource;

}

// 配置分片规则

private TableRuleConfiguration getTableRuleConfig() {

TableRuleConfiguration orderTableRuleConfig = new TableRuleConfiguration();

orderTableRuleConfig.setLogicTable("t\_order");

orderTableRuleConfig.setActualDataNodes("ds${0..1}.t\_order\_${0..1}");

// 配置分库策略（Groovy表达式配置db规则）

orderTableRuleConfig.setDatabaseShardingStrategyConfig(

new InlineShardingStrategyConfiguration("user\_id", "ds${user\_id % 2}"));

// 配置分表策略（Groovy表达式配置表路由规则）

orderTableRuleConfig.setTableShardingStrategyConfig(

new InlineShardingStrategyConfiguration("order\_id", "t\_order\_${order\_id % 2}"));

return orderTableRuleConfig;

}

// 配置默认的分片规则

private TableRuleConfiguration configDefaultTableRule() {

TableRuleConfiguration orderTableRuleConfig = new TableRuleConfiguration();

return orderTableRuleConfig;

}

/\*\* 读写分离规则配置 \*\*/

public List<MasterSlaveRuleConfiguration> getMasterSlaveRuleConfiguration() throws SQLException {

MasterSlaveRuleConfiguration masterSlaveRuleConfig1 = new MasterSlaveRuleConfiguration("ds0",

"ds0", Arrays.asList("ds0\_slave0", "ds0\_slave1"));

MasterSlaveRuleConfiguration masterSlaveRuleConfig2 = new MasterSlaveRuleConfiguration("ds1",

"ds1", Arrays.asList("ds1\_slave0", "ds1\_slave1"));

return Lists.newArrayList(masterSlaveRuleConfig1, masterSlaveRuleConfig2);

}

private DruidDataSource getDataSource(String dsname) throws Exception {

DruidDataSource ds = createDefaultDruidDataSource();

if ("ds".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else if ("ds0".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds0?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else if ("ds0\_slave0".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds0\_slave0?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else if ("ds0\_slave1".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds0\_slave1?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else if ("ds1".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds1?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else if ("ds1\_slave0".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds1\_slave0?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else if ("ds1\_slave1".equals(dsname)) {

ds.setDriverClassName("com.mysql.jdbc.Driver");

ds.setUrl("jdbc:mysql://172.16.31.43:3306/ds1\_slave1?useUnicode=true&characterEncoding=UTF-8&useSSL=false");

ds.setUsername("liuxing");

ds.setPassword("Liuxing009!");

return ds;

} else {

throw new Exception("did not discover this datasource name!");

}

}

Map<String, DataSource> createDataSourceMap() throws Exception {

Map<String, DataSource> result = new HashMap<>();

result.put("ds", getDataSource("ds"));

result.put("ds0", getDataSource("ds0"));

result.put("ds0\_slave0", getDataSource("ds0\_slave0"));

result.put("ds0\_slave1", getDataSource("ds0\_slave1"));

result.put("ds1", getDataSource("ds1"));

result.put("ds1\_slave0", getDataSource("ds1\_slave0"));

result.put("ds1\_slave1", getDataSource("ds1\_slave1"));

return result;

}

@Bean(name = "myJdbcTemplatep1")

public MyJdbcTemplate getJdbcTemplatePrimary1(@Qualifier("shardingDataSource") DataSource dataSource) {

return new MyJdbcTemplate(dataSource);

}

// @Bean(name = "myBEDSoftTransaction")

// public BEDSoftTransaction

// initializeSoftTransaction(@Qualifier("shardingDataSource") DataSource

// dataSource,

// @Qualifier("commonDataSource") DataSource commonDataSource)

// throws SQLException {

// // 1. 配置SoftTransactionConfiguration

// SoftTransactionConfiguration transactionConfig =

// configSoftTransaction(dataSource, commonDataSource);

//

// // 2. 初始化SoftTransactionManager

// SoftTransactionManager transactionManager = new

// SoftTransactionManager(transactionConfig);

// transactionManager.init();

// // 3. 获取BEDSoftTransaction

// BEDSoftTransaction transaction = (BEDSoftTransaction) transactionManager

// .getTransaction(SoftTransactionType.BestEffortsDelivery);

// return transaction;

// }

//

// // 配置柔性事务配置

// private SoftTransactionConfiguration

// configSoftTransaction(@Qualifier("shardingDataSource") DataSource dataSource,

// @Qualifier("commonDataSource") DataSource commonDataSource) {

// // 事务管理器管理的数据源

// SoftTransactionConfiguration transactionConfig = new

// SoftTransactionConfiguration(dataSource);

// // 事务日志存储类型。可选值: RDB,MEMORY。使用RDB类型将自动建表

// transactionConfig.setStorageType(TransactionLogDataSourceType.RDB);

// // 存储事务日志的数据源，如果storageType为RDB则必填

// transactionConfig.setTransactionLogDataSource(commonDataSource);

// // 同步的事务送达的最大尝试次数

// transactionConfig.setSyncMaxDeliveryTryTimes(3);

// return transactionConfig;

// }

/\*\* common 数据源 start \*/

@Primary

@Bean(name = "commonDataSource", destroyMethod = "close", initMethod = "init")

@ConfigurationProperties("spring.datasource.commondb")

public com.alibaba.druid.pool.DruidDataSource getDataSourcep3() {

DruidDataSource druidDataSource = new DruidDataSource();

return druidDataSource;

}

}

### shardingjdbc的不支持项

不支持存储过程，函数，游标的操作

不支持nativesql

不支持save point

还有一些其它项目，参考

http://shardingsphere.io/document/current/cn/manual/sharding-jdbc/unsupported-items/

## 分布式事务

sharding-jdbc完全支持非跨库事务，例如：仅分表，或分库但是路由的结果在单库中。

对于跨库事务：不支持硬件异常或网络异常引起的跨库事务

#### 使用非事务消息（rabbitMQ）的分布式事务解决方案

例：购买商品过程中A系统更新订单和B系统更新库存（库存信息和订单信息在不同的数据库内）

处理方案分为两阶段：MQ生产者阶段和MQ消费者阶段

MQ生产者阶段：

1. 更新订单数据库成功，向MQ投递消息（消息指令：消费库存）也成功,ok，程序进入到后续消费消息环节
2. 更新订单数据库失败，不向MQ发消息，业务逻辑结束
3. 更新订单数据库成功，向MQ投递消息失败，抛出异常，刚刚执行的更新数据库操作回滚，业务逻辑结束。

MQ消费者阶段：

1. 消息出列后，执行库存信息入库操作，如果入库成功，mq内的消息丢弃，如果入库失败，消息仍旧保存在mqserver。 rabbitmq的ACK消息确认机制保证consumer端消费消息失败后，消息仍旧保存在 mqserver队列内

2，关于消息的重复消费。执行库存信息入库操作的服务尽量保持幂等性，尽管rabbitmq已经规避了消息重复消费的问题。

#### 使用事务消息（rocketmq）的分布式事务解决方案（未实现）

处理思路：

1. A系统向消息中间件发送一条预备消息
2. 消息中间件保存预备消息并返回成功
3. A执行本地事务
4. A发送提交消息给消息中间件

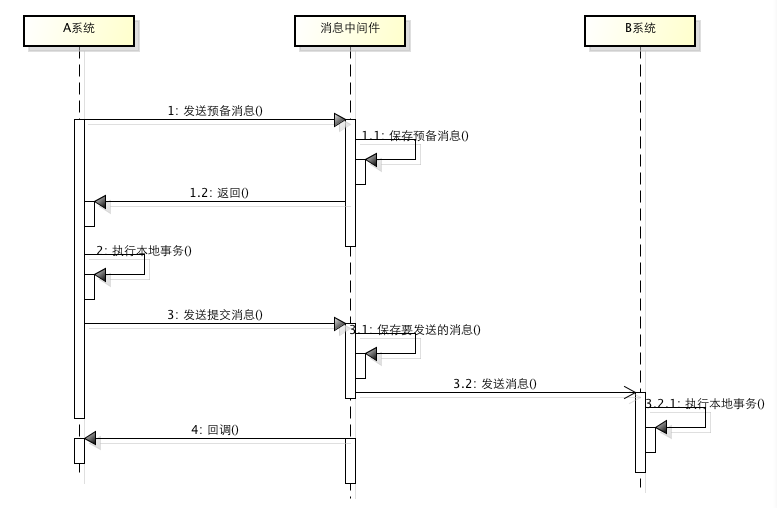
通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

步骤一出错，则整个事务失败，不会执行A的本地操作

步骤二出错，则整个事务失败，不会执行A的本地操作

步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息

步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务



总体来看，和非事务消息相比较而言：使用事务消息不必再实现A系统本地事务的回滚业务，但明显增加了系统的复杂度，因为A系统要实现一个回调接口，且MQ需要支持事务消息机制。

## 总结

我个人理解：

※如果能垂直分库，尽可能垂直分库

※对于水平拆分，能靠同库分表解决单表数据量的问题，就无需按表分库

※尽量避免分布式事务