

Table of Contents

Introduction	1.1
0.概述	1.2
0.1 dble 简介与整体架构	1.2.1
0.2 dble对MyCat做的增强	1.2.2
0.3 快速开始	1.2.3
0.3.1 docker镜像快速开始	1.2.3.1
0.3.2 docker-compose快速开始	1.2.3.2
0.4 数据拆分简介	1.2.4
1.配置文件	1.3
1.1 cluster.cnf	1.3.1
1.2 bootstrap.cnf	1.3.2
1.3 user.xml	1.3.3
1.4 db.xml	1.3.4
1.5 sharding.xml	1.3.5
1.6 log4j2.xml	1.3.6
1.7 全局序列配置	1.3.7
1.7.1 MySQL offset-step方式	1.3.7.1
1.7.2 时间戳方式(类Snowflake)	1.3.7.2
1.7.3 分布式时间戳方式(类Snowflake)	1.3.7.3
1.7.4 分布式offset-step方式	1.3.7.4
1.8 cache配置	1.3.8
1.8.1 cache配置	1.3.8.1
1.8.2 ehcache配置	1.3.8.2
1.9 自定义拆分算法	1.3.9
1.10 版本变更	1.3.10
1.11 自定义告警	1.3.11
1.12 自定义全局表一致性检查	1.3.12
1.13 Schema下默认拆分表	1.3.13
2.功能描述	1.4
2.0 管理端元数据库	1.4.1
2.0.1 dble_config表	1.4.1.1
2.1 管理端命令	1.4.2
2.1.1 select命令	1.4.2.1
2.1.2 set命令	1.4.2.2
2.1.3 show命令	1.4.2.3
2.1.4 switch命令	1.4.2.4
2.1.5 kill命令	1.4.2.5
2.1.6 stop命令	1.4.2.6
2.1.7 reload命令	1.4.2.7
2.1.8 rollback命令 (已废弃)	1.4.2.8
2.1.9 offline命令	1.4.2.9
2.1.10 online命令	1.4.2.10
2.1.11 file命令 (已废弃)	1.4.2.11
2.1.12 log命令 (已废弃)	1.4.2.12
2.1.13 配置检查命令	1.4.2.13
2.1.14 pause & resume 命令	1.4.2.14
2.1.15 慢查询日志相关命令	1.4.2.15
2.1.16 创建/删除物理库命令	1.4.2.16
2.1.17 check @@metadata命令	1.4.2.17
2.1.18 release @@metadata命令	1.4.2.18
2.1.19 split命令	1.4.2.19
2.1.20 flow_control 命令	1.4.2.20

2.1.21 刷新连接池命令	1.4.2.21
2.1.22 脱离集群命令	1.4.2.22
2.2 全局序列	1.4.3
2.2.1 MySQL offset-step方式	1.4.3.1
2.2.2 时间戳方式	1.4.3.2
2.2.3 分布式时间戳方式	1.4.3.3
2.2.4 分布式offset-step方式	1.4.3.4
2.3 读写分离	1.4.4
2.4 注解	1.4.5
2.5 分布式事务	1.4.6
2.5.1 XA事务概述	1.4.6.1
2.5.2 XA事务的提交以及回滚	1.4.6.2
2.5.3 XA事务的后续补偿以及日志清理	1.4.6.3
2.5.4 XA事务的记录	1.4.6.4
2.5.5 一般分布式事务概述	1.4.6.5
2.5.6 检测疑似残留XA事务	1.4.6.6
2.6 连接池管理	1.4.7
2.7 内存管理	1.4.8
2.8 集群同步协调&状态管理	1.4.9
2.9 grpc 告警	1.4.10
2.10 表meta数据管理	1.4.11
2.10.1 Meta信息初始化	1.4.11.1
2.10.2 Meta信息维护	1.4.11.2
2.10.3 一致性检测	1.4.11.3
2.10.4 View Meta	1.4.11.4
2.11 统计管理	1.4.12
2.11.1 查询条件统计	1.4.12.1
2.11.2 表状态统计	1.4.12.2
2.11.3 用户状态统计	1.4.12.3
2.11.4 命令统计	1.4.12.4
2.11.5 heartbeat统计	1.4.12.5
2.11.6 网络读写统计	1.4.12.6
2.11.7 快速sql统计命令	1.4.12.7
2.12 故障切换	1.4.13
2.13 前后端连接检查	1.4.14
2.14 ER表	1.4.15
2.15 global表	1.4.16
2.16 缓存的使用	1.4.17
2.17 执行计划	1.4.18
2.18 性能观测和调整	1.4.19
2.19 智能计算reload	1.4.20
2.20 慢查询日志	1.4.21
2.21 单条SQL性能trace	1.4.22
2.22 KILL @@DDL_LOCK	1.4.23
2.23 外部高可用联动	1.4.24
2.23.1 外部后端MYSQL-HA连接	1.4.24.1
2.23.2 命令的使用说明	1.4.24.2
2.23.3 命令的实现细节	1.4.24.3
2.23.4 简单的HA交互使用案例	1.4.24.4
2.24 超时控制	1.4.25
2.25 流量控制	1.4.26
2.26 client_found_rows权能标志	1.4.27
2.27 general日志	1.4.28
2.28 sql统计	1.4.29
2.29 load data批处理模式	1.4.30

2.30 in子查询是否转join的说明	1.4.31
2.31 DDL日志解读	1.4.32
2.32 分析用户	1.4.33
2.33 hint指定执行计划	1.4.34
2.34 安全加密	1.4.35
2.34.1 SSL自签名证书生成	1.4.35.1
2.34.2 DBLE启用SSL	1.4.35.2
2.35 堆外内存泄露监控	1.4.36
2.36 延迟检测	1.4.37
2.37 审计日志	1.4.38
2.38 tcp相关参数	1.4.39
2.39 HTAP 快速上手指南	1.4.40
2.40 dble中线程处理(print、kill、recover)	1.4.41
3.语法兼容	1.5
3.1 DDL	1.5.1
3.1.1 DDL&Table Syntax	1.5.1.1
3.1.2 DDL&View Syntax	1.5.1.2
3.1.3 DDL&Index Syntax	1.5.1.3
3.1.4 DDL透传	1.5.1.4
3.1.5 DDL&Database_Syntax	1.5.1.5
3.1.6 ONLINE DDL	1.5.1.6
3.2 DML	1.5.2
3.2.1 INSERT	1.5.2.1
3.2.2 REPLACE	1.5.2.2
3.2.3 DELETE	1.5.2.3
3.2.4 UPDATE	1.5.2.4
3.2.5 SELECT	1.5.2.5
3.2.6 SELECT JOIN syntax	1.5.2.6
3.2.7 SELECT UNION Syntax	1.5.2.7
3.2.8 SELECT Subquery Syntax	1.5.2.8
3.2.9 LOAD DATA	1.5.2.9
3.2.10 不支持的DML语句	1.5.2.10
3.3 Prepared SQL Syntax	1.5.3
3.4 Transactional, Savepoint and Locking Statements	1.5.4
3.4.1 一般事务语法	1.5.4.1
3.4.2 含隐式提交语句	1.5.4.2
3.4.3 SAVEPOINT 语法	1.5.4.3
3.4.4 Lock&unlock语法	1.5.4.4
3.4.5 SET TRANSACTION Syntax	1.5.4.5
3.4.6 XA 事务语法	1.5.4.6
3.4.7 其它不支持的事务语法	1.5.4.7
3.5 DAL	1.5.5
3.5.1 SET	1.5.5.1
3.5.2 SHOW	1.5.5.2
3.5.3 KILL	1.5.5.3
3.5.4 不支持的DAL语句	1.5.5.4
3.6 存储过程支持方式	1.5.6
3.7 Utility Statements	1.5.7
3.8 Hint	1.5.8
3.9 其他不支持语句	1.5.9
3.10 函数与操作符支持列表(alpha版本)	1.5.10
3.11 导入导出方式	1.5.11
4.协议兼容	1.6
4.1 基本包	1.6.1
4.2 连接建立	1.6.2

4.3 文本协议	1.6.3
4.4 二进制协议 (Prepared Statements)	1.6.4
4.5 服务响应包	1.6.5
5. 已知限制	1.7
5.1 druid引发的限制	1.7.1
5.2 其他已知限制	1.7.2
6. 与MySQL Server的差异化描述	1.8
6.1 事务中遇到主键冲突需要显式回滚	1.8.1
6.2 INSERT不能显式指定自增序列	1.8.2
6.3 增加"show all tables"	1.8.3
6.4 去除了增删改的message信息	1.8.4
6.5 information_schema等库的支持	1.8.5
7. 开发者须知	1.9
7.1 SQL开发编写原则	1.9.1
7.2 dble连接Demo	1.9.2
7.3 其他注意事项	1.9.3
8. 配置示例	1.10
8.1 时间戳方式全局序列的配置	1.10.1
8.2 MySQL-offset-step 方式全局序列的配置	1.10.2
9. sysbench压测dble示例	1.11
9.1 测试环境及架构	1.11.1
9.2 修改dble配置	1.11.2
9.3 使用sysbench进行压测	1.11.3
A. Faq	1.12
A.1 ErrorCode	1.12.1
max Connections	1.12.1.1
Out Of Memory Error	1.12.1.2
The Problem Of Hint	1.12.1.3
NestLoop Parameters Lead To Temptable Exception	1.12.1.4
Can't Get Variables From ShardingNode	1.12.1.5
Port already in use:1984	1.12.1.6
Sharding Column Cannot Be Null	1.12.1.7
A.2 原理解释	1.12.2
How To Use Explain To Resolve The Distribution Rules Of Group Gy	1.12.2.1
Hash And ConsistentHashing And Jumpstringhash	1.12.2.2
A.3 使用说明	1.12.3
ToBeContinued2	1.12.3.1

db1e 中文技术参考手册

注意

本分支上的手册适用于3.23.08.x版db1e，其他版本的文档请参考对应tag分支或者release版文档。

目录

参考 [gitbook](#) 左侧目录区 或 [SUMMARY.md](#)

软件包下载

[下载](#) 或者[github镜像站下载](#)

文档PDF下载

[文档PDF下载](#) 或者[github镜像站下载](#)

注意：如版本尚未发布，则不能下载

中文公开课

[db1e中文公开课](#)

官方技术支持

- 代码库 [github: github.com/actiontech/db1e](https://github.com/actiontech/db1e)
- 自动化测试库 [github: github.com/actiontech/db1e-test-suite](https://github.com/actiontech/db1e-test-suite)
- 文档库 [github: github.com/actiontech/db1e-docs-cn](https://github.com/actiontech/db1e-docs-cn)
- 文档库 [github pages: actiontech.github.io/db1e-docs-cn](https://actiontech.github.io/db1e-docs-cn)
- 网站: [DBLE官方网站](#)
- QQ group: 669663113
- 开源社区微信公众号



提示

如果您使用了db1e，请告诉我们。

联系我们

如果想获得db1e 的商业支持, 您可以联系我们:

- 全国支持: 400-820-6580
- 华北地区: 86-13910506562, 汪先生
- 华南地区: 86-18503063188, 曹先生
- 华东地区: 86-18930110869, 梁先生
- 西南地区: 86-13540040119, 洪先生

0 概览

- 0.1 dble 简介与整体架构
- 0.2 dble对MyCat做的增强
- 0.3 快速开始
 - 0.3.1 docker镜像快速开始
 - 0.3.2 docker-compose快速开始
- 0.4 数据拆分简介

0.1 dble 简介与整体架构

0.1.1 dble简介

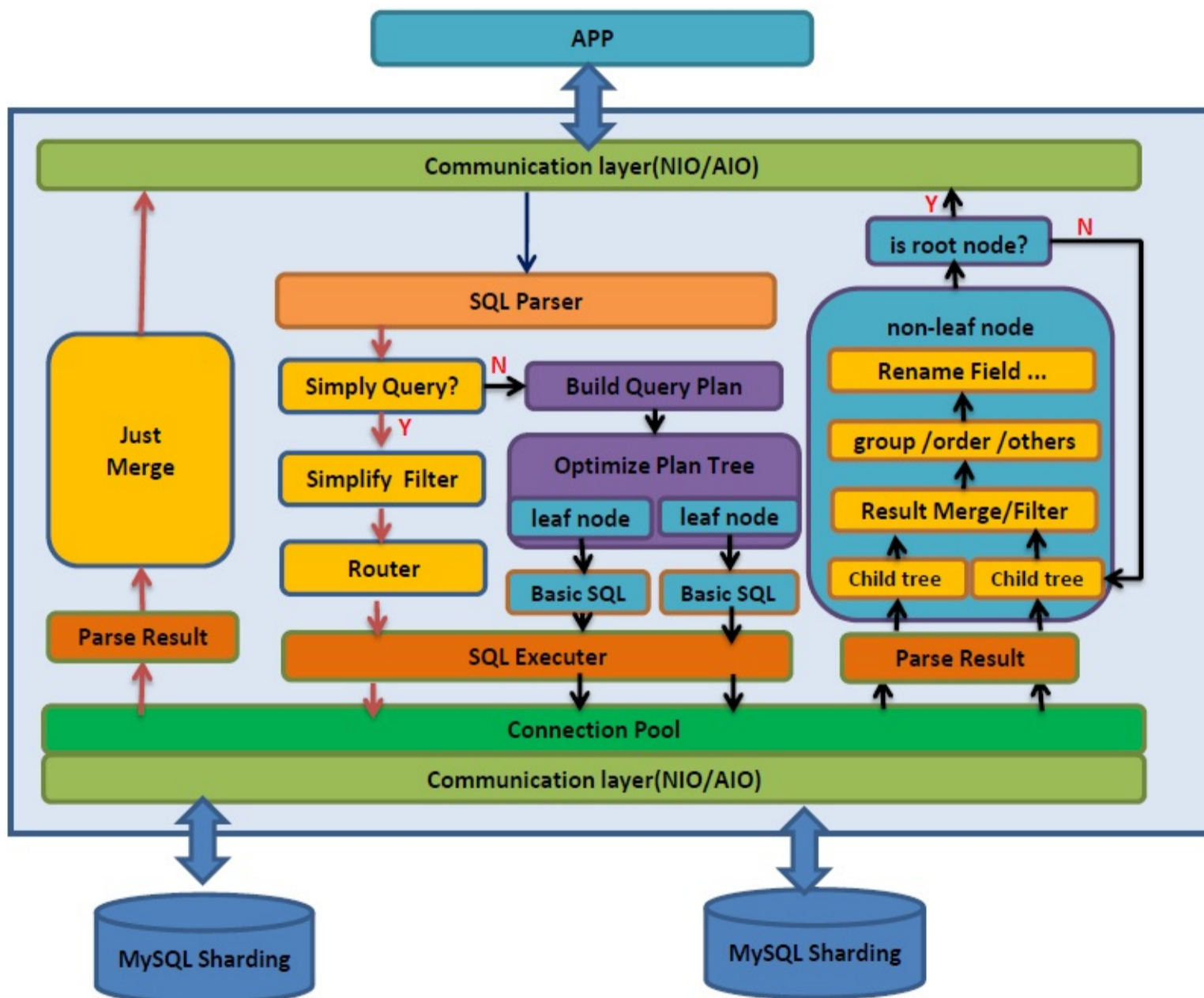
dble是上海爱可生信息技术有限公司基于MySQL的高可扩展性的分布式中间件，存在以下几个优势特性：

- **数据水平拆分** 随着业务的发展，您可以使用dble来替换原始的单个MySQL实例。
- **兼容MySQL** 与MySQL协议兼容，在大多数情况下，您可以用它替换MySQL来为您的应用程序提供新的存储，而无需更改任何代码。
- **高可用性** dble服务器可以用作集群，业务不会受到单节点故障的影响。
- **SQL支持** 支持SQL 92标准和MySQL方言。我们支持复杂的SQL查询，如group by, order by, distinct, join, union, sub-query等等。
- **复杂查询优化** 优化复杂查询，包括但不限于全局表连接分片表，ER关系表，子查询，简化选择项等。
- **分布式事务支持** 使用两阶段提交的分布式事务。您可以为了性能选择普通模式或者为了数据安全采用XA模式。当然，XA模式依赖于MySQL-5.7的XA Transaction，MySQL节点的高可用性和数据的可靠性。

0.1.2 dble由来

- dble 是基于开源项目MyCat的，在此对于MyCat的贡献者们致以由衷的感谢。
- 对我们来说，专注于MySQL是一个更好的选择。所以我们取消了对其他数据库的支持，对兼容性，复杂查询和分布式事务的行为进行了深入的改进/优化。当然，还修复了一些bugs。详情可见[dble对MyCat做的改进](#)

0.1.3 dble内部架构



0.2 dble对MyCat做的增强

建议大家收看[免费课程](#)来初步认识dble的改进项

0.2.1 缺陷修复（目前mycat社区不活跃，不再追踪mycat的bug）

- 由于对堆外内存的使用不当，导致高并发操作时对同一内存可能发生“double free”，从而造成JVM异常，服务崩溃。 #4
- XA事务漏洞：包乱序导致客户端崩溃 #21
- where关键字写错时，会忽视后面的where条件，会得到错误的结果，比如select * from customer wher id=1;#126
- 对于一些隐式分布式事务，例如insert into table values(节点1)，(节点2)；原生mycat直接下发，这样当某个节点错误时，会造成该SQL执行了一部分
- 权限黑名单针对同一条sql只在第一次生效。#92
- 聚合/排序的支持度非常有限，而且在很多场景下还存在结果不正确、执行异常等问题 #43,#31,#44
- 针对between A and B语法，hash拆分算法计算出来的范围有误#23
- 开启全局表一致性检查时，对全局表的处理存在诸多问题，例如不能alter table、insert...on duplicate...时不更新时间戳、update...in ()报错等#24, #25,#26, #5
- 多值插入时，全局序列生成重复值 #1
- ER表在一个事务内被隔离,不能正确插入子表数据#13
- sharding-join结果集不正确#17
详情及其他修正请见 [修复列表](#)

0.2.2 实现改进

- 对某些标准SQL语法支持不够好的方面作了改进，例如对create table if not exists...、alter table add/drop [primary] key...等语法的支持
- 对整体内部IO结构进行了大幅的改造和调优:参见[dble的IO结构](#) 或者见本节结尾附录2
- 禁止普通用户连接管理端口进行管理操作，增强安全性 #56
- 对全局序列做了如下改进
 - 删去无工程意义的本地文件方式
 - 改进数据库方式、ZK方式，使获取的序列号更加准确
 - 改进时间戳方式和ZK ID生成器方式，消除并发低时的数据分布倾斜问题
 - 修复了数据库方式全局序列中线程安全的问题#489
 - 移除自定义语法 限制：全局序列值不能显式指定
 - 原来：insert into table1(id,name) values(next value for MYCATSEQ_GLOBAL,'test');
 - 现在1：insert into table1(name) values('test');
 - 现在2：insert into table1 values('test');
 - 注意时间戳方式需要该字段是bigint
- 改进对ER表的支持，智能处理连接隔离，解决同一事务内不可以同时写入父子表的问题，并优化ER表的执行计划
- 系统通过智能判断，对于一些没有显式配置但实际符合ER条件的表视作ER表同样处理
- 在中间件内进行智能解析与判断，使用正确的schema，替换有缺陷的checkSQLschema 参数
- conf/index_to_charset.properties的内容固化到代码。
- 对于前端按照用户限制连接数，限制总连接数
- 改进原本的SQL统计，增加UPDATE/DELETE/INSERT

0.2.3 功能增强

- 提供了更强大的查询解析树，取代ShareJoin，使跨节点的语法支持度更广(join,union,subquery)，执行效率更高，同时聚合/排序也有了较大改进
- 提供科学的元数据管理机制，更好的支持show、desc等管理命令，支持不指定columns的insert语句 #7
- 元数据自动检查
 - 启动时对元数据进行一致性检查
 - 配置定时任务，对元数据进行一致性检查
- 提供更详实的执行计划，更准确的反映SQL语句的执行过程
 - 举例：mysql> explain select * from sharding_two_node a inner join sharding_four_node b on a.id =b.id;

```

+-----+-----+-----+
| SHARDING_NODE | TYPE      | SQL/REF                                     |
+-----+-----+-----+
| dn1.0         | BASE SQL | select `a`.`id`,`a`.`c_char`,`a`.`ts`,`a`.`si` from `sharding_two_node` `a` ORDER BY `a`.`id` ASC |
| dn2.0         | BASE SQL | select `a`.`id`,`a`.`c_char`,`a`.`ts`,`a`.`si` from `sharding_two_node` `a` ORDER BY `a`.`id` ASC |
| dn1.1         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| dn2.1         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| dn3.0         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| dn4.0         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| merge.1       | MERGE    | dn1.0, dn2.0                               |
| merge.2       | MERGE    | dn1.1, dn2.1, dn3.0, dn4.0                 |
| join.1        | JOIN     | merge.1, merge.2                           |
+-----+-----+-----+
9 rows in set (0.00 sec)

```


- `set` 系统变量语句的改进
- `set charset/names` 语句的支持
- [分布式事务:XA](#)实现方式的异常处理的改进
- 大小写敏感支持
- 支持DUAL
- 支持单次请求[多语句](#)(部分客户端有使用,C和C++常见)
- 不断丰富路由规则优化和条件优化
- 升级Druid,跟进最新的解析器
- 升级fastjson,避免安全问题
- [智能判断reload](#)连接变更,热更新连接池
- 对MySQL协议及GUI工具/Driver更友好的支持
- 增加更多的[管理端命令](#),满足更多运维需要
- 缓存支持使用RocksDB
- 增加[慢查询日志功能](#),兼容mysqldumpslow 和 pt-query-digest
- [Trace功能](#),用于分析单条查询的性能瓶颈
- 大小写敏感依赖于后端MySQL
- 支持[Prepared SQL Statement Syntax](#)
- 支持以下子查询
 - The Subquery as Scalar Operand
 - Comparisons Using Subqueries
 - Subqueries with ANY, IN, or SOME
 - Subqueries with ALL
 - Subqueries with EXISTS or NOT EXISTS
 - Derived Tables (Subqueries in the FROM Clause)
- 支持dble层面的 [View](#)
- 支持MySQL8.0 默认登陆验证插件
- 提供[自定义告警接口](#)
- 支持[自定义拆分算法](#)
- 支持自增列可以设置为非主键列
- 可以观察执行中的DDL
- 提供配置[预检查功能](#)
- [流量暂停和恢复功能](#)

0.2.4 功能裁减

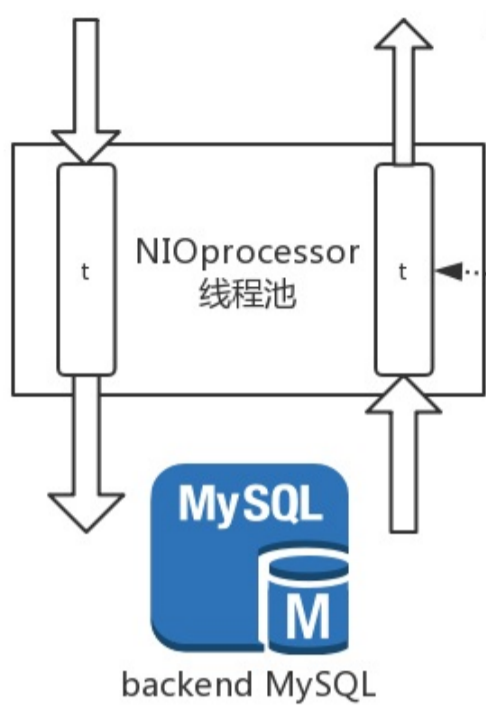
- 仅保留枚举、范围、HASH、日期等分片算法,对这几个算法进行了可用性的改进,使之更加贴合实际应用,项目需要时可以按需提供
- 移除异构数据库支持
- 禁止某些不支持的功能,这些功能客户端调用时不会报错,但结果并非用户想要的结果,例如无效的`set`语句
- 移除目前实现有问题的第一结点库内分表模式
- 移除`writeType`参数,等效于原来`writeType = 1`
- 移除`handleDistributedTransactions` 选项,直接支持分布式事务

0.2.5 附录

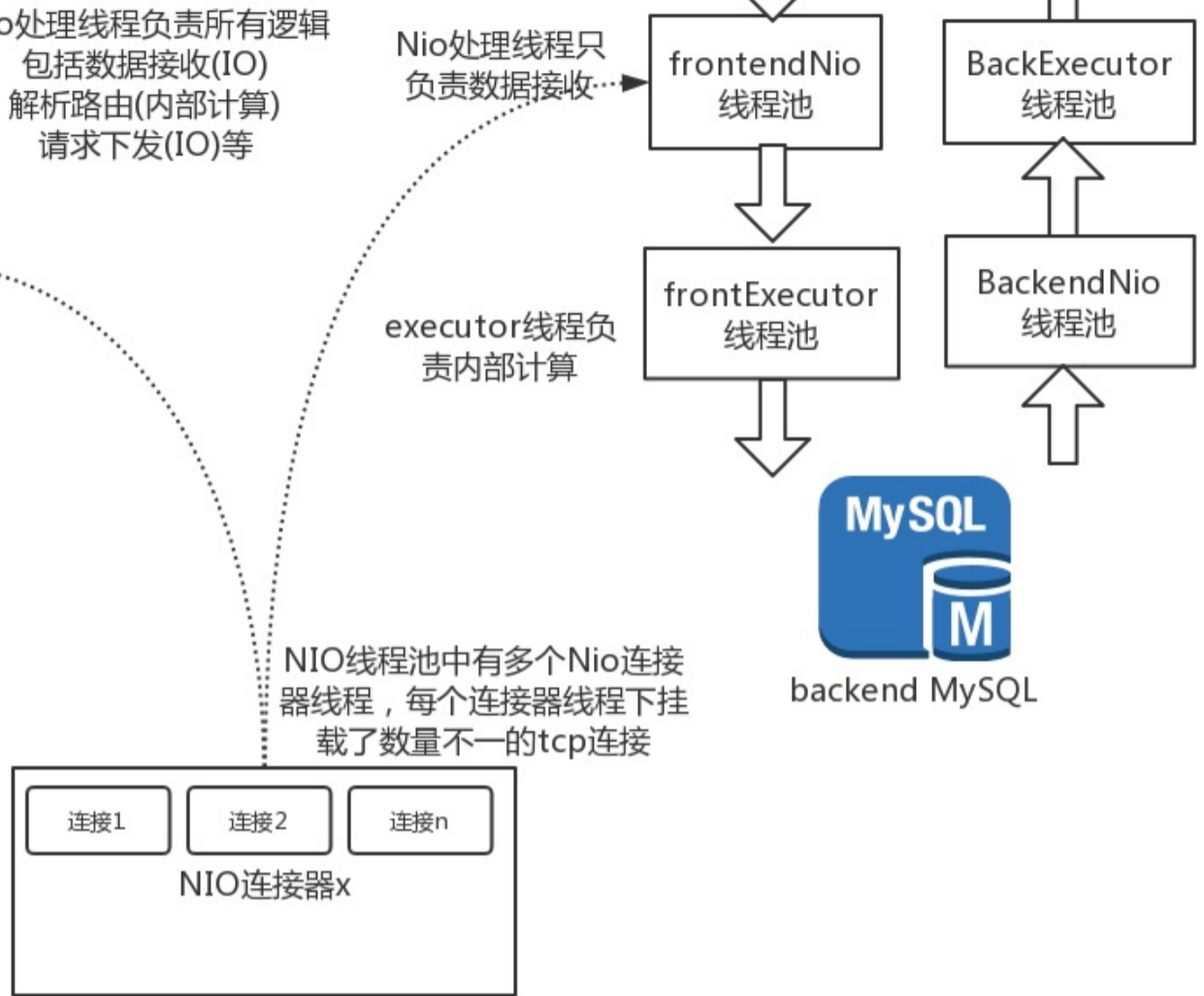
一些功能改进的详细描述,见[开源数据库中间件DBLE对MyCAT的增强与改进](#)

0.2.6 附录2

Mycat以及dble 2.18.02.0之前版本



dble 2.18.02以及之后版本



Nio处理器（处理线程）内部为挂载不定数量个连接，并且循环响应每个连接请求

在数据处理和数据接收进行线程分割之后(dble 2.18.02)，使得dble可以并发响应挂载在同一个NIO连接器(同一个processor线程)上的请求

e.g.

恰好我们存在场景连接1, 2同时有请求过来，旧版本需要循环处理连接1, 2的任务，在连接1的任务处理完成之前，连接2的任务无法进行处理

新的IO结构中，连接1的数据被接收完毕之后，NIO线程就可以接收连接2的数据，并且此时连接1的数据已经在executor线程池进行处理中，连接1, 2之间的任务执行变成并行

0.3 快速开始

0.3.0.1 关于本节

- 本节内容为您介绍如何使用dble安装包快速部署并启动一个dble服务，并简单了解dble的使用和管理

0.3.0.2 安装准备

以下部分将被需要作为dble启动的基础支撑

- 两个启动的MySQL实例
dble是通过连接mysql数据库实例来进行数据的存储，所以请至少准备两个正在运行的mysql实例，假设您的机器上存在两个MySQL实例：
A:\$url=ip1:3306,\$user=test,\$password=testPsw
B:\$url=ip2:3306,\$user=test,\$password=testPsw
请正确配置/etc/hosts，保证此MySQL实例可以正确访问，否则之后可能会报错 "NO ROUTE TO HOST"。
- JVM环境
dble是使用java开发的，所以需要启动dble您先需要在机器上安装java版本1.8或以上，并且确保JAVA_HOME参数被正确的设置

0.3.0.3 下载并安装

- 通过此链接(<https://github.com/actiontech/dble/releases>)下载最新版本的安装包
- 解压并安装dble到指定文件夹中

```
mkdir -p $working_dir
cd $working_dir
tar -xvf actiontech-dble-$version.tar.gz
cd $working_dir/dble/conf
mv cluster_template.cnf cluster.cnf
mv bootstrap_template.cnf bootstrap.cnf
mv db_template.xml db.xml
mv user_template.xml user.xml
mv sharding_template.xml sharding.xml
```

0.3.0.4 dble的初始化配置

- 修改db.xml，找到其中的 instanceM1 和 instanceM2，将数据库信息替换成已经安装启动的 MySQL 实例：

```
<dbInstance name="instanceM1" url="ip1:3306" user="your_user" password="your_psw" maxCon="1000" minCon="10"
    primary="true">

<dbInstance name="instanceM2" url="ip2:3306" user="your_user" password="your_psw" maxCon="1000" minCon="10"
    primary="true"/>
```

0.3.0.5 启动并连接

- 启动命令

```
cd $working_dir/dble
bin/dble start
```

- 如果启动失败请使用此命令查看失败的详细原因 `tail -f logs/wrapper.log`
- 使用mysql客户端直接连接dble管理端口，默认密码654321 `mysql -p -P9066 -h 127.0.0.1 -u man1`
- 您可以使用mysql一样的方式执行以下语句用于在实例上建立虚拟结点对应的schema

```
create database @@shardingnode='dn$1-6';
```

另外此端口还可以执行一些其他命令

- 使用mysql客户端直接连接dble服务，默认密码123456 `mysql -p -P8066 -h 127.0.0.1 -u root`
- 您可以使用mysql一样的方式执行以下语句

```
use testdb;
drop table if exists tb_enum_sharding;
create table if not exists tb_enum_sharding (
  id int not null,
  code int not null,
  content varchar(250) not null,
  primary key(id)
)engine=innodb charset=utf8;
insert into tb_enum_sharding values(1,10000,'1'),(2,10010,'2'),(3,10000,'3'),(4,10010,'4');
```

0.3.1 快速开始(docker)

0.3.1.1 关于本节

- 本节内容为您介绍如何通过dockerhub上的dble镜像快速启动一个dble

0.3.1.2 安装准备

- 安装docker
- 若需要使用docker-compose快速启动，请安装docker-compose
- 安装mysql连接工具，用于进行连接测试观察结果

0.3.1.3 安装过程

按照顺序依次执行以下docker命令：

```
docker network create -o "com.docker.network.bridge.name="dble-net" --subnet 172.18.0.0/16 dble-net

docker run --name backend-mysql1 --ip 172.18.0.2 -e MYSQL_ROOT_PASSWORD=123456 -p 33061:3306 --network=dble-net -d mysql:5.7 --server-id=1

docker run --name backend-mysql2 --ip 172.18.0.3 -e MYSQL_ROOT_PASSWORD=123456 -p 33062:3306 --network=dble-net -d mysql:5.7 --server-id=2

sleep 30

docker run -d -i -t --name dble-server --ip 172.18.0.5 -p 8066:8066 -p 9066:9066 --network=dble-net actiontech/dble:latest
```

通过以上命令依次创建一个docker网络，两个分别映射到主机 33061 和 33062 端口的mysql服务，一个将服务端和管理端映射到主机 8066 和 9066 端口的服务。服务将在约一分钟之后被启动，这是由于为了进行快速的启动需要对于mysql和dble的配置进行一些初始化。

0.3.1.4 连接并使用

使用准备好的mysql连接工具连接主机的8066或者9066端口，在docker的默认配置中

8066 端口(服务端口能够执行SQL语句)的用户和密码为 root/123456

9066 端口(管理端口能够执行管理语句)的用户和密码为 man1/654321

此例子中准备了一些表格并提前进行了表格创建，若需要连接更多的表格配置详情和使用方法

请在dble-server容器中查阅 /opt/dble/conf/sharding.xml 文件

在虚拟机已经有安装mysql客户端的状态下，可以使用以下默认命令进行连接

```
#连接 dble 业务端口
mysql -P8066 -u root -p123456 -h 127.0.0.1
#连接 dble 管理端口
mysql -P9066 -u man1 -p654321 -h 127.0.0.1
#连接后端mysql1
mysql -P33061 -u root -p123456 -h 127.0.0.1
#连接后端mysql2
mysql -P33062 -u root -p123456 -h 127.0.0.1
```

0.3.1.5 环境清理

使用完成或者进行环境重建的时候可以使用以下命令进行环境的清空

```
docker stop backend-mysql1
docker stop backend-mysql2
docker stop dble-server
docker rm backend-mysql1
docker rm backend-mysql2
docker rm dble-server
docker network rm dble-net
```

0.3.1.6 docker-compose 版本快速启动

docker-compose启动需要先从github项目下载对应的配置文件

```
wget https://raw.githubusercontent.com/actiontech/dble/master/docker-images/quick-start/docker-compose.yml
```

通过使用docker-compose的配置脚本直接启动两个mysql镜像以及一个dble镜像，在配置文件存放目录执行

```
docker-compose up
```

同样的，默认状态下启动 dble 的状态和使用 quick-start 启动的 dble 一致，dble-server容器会启用服务端口和管理端口 8066/9066 对外提供服务 可以使用默认用户和密码 root/123456 连接8066端口进行测试，同时两个后端mysql的本机端口分别是 33061/33062，默认的用户和密码也是root/123456

在使用或者测试完毕之后，在配置文件存放目录下使用以下指令回收对应资源

```
docker-compose stop
docker-compose rm
```

0.3.1.7 尝试使用本地配置启动docker-compose

注意

本小结的内容需要用户在充分了解并掌握dble配置和结构的状态下进行，作为一种快速启动特定配置dble以供测试或调试使用 首次了解并使用dble的用户可以跳过此节

本地配置启动dble的原理是通过volumes配置的映射将本地的配置目录映射到docker容器中，之后初始化的时候在初始化脚本中将本地配置目录中的文件复制到对应的dble/conf目录中，之后再行初始化和dble启动

首先需要在docker-compose.yml的最后一段dble-server容器配置中添加以下内容

```
volumes:
  - /opt/test/conf:/opt/self_conf
```

此命令将本地的/opt/test/conf目录映射到目标容器/opt/self_conf目录中去 之后调整dble-server容器的启动命令

```
command: ["/opt/dble/bin/wait.sh", "backend-mysql11:3306", "--", "/opt/self_conf/docker_init_start.sh"]
```

将原本调用的/opt/dble/bin/docker_init_start.sh修改为本次初始化准备使用的self_conf目录下的初始化脚本

这里对于本地启动的初始化脚本只给出如下一些建议

- 首先将需要修改的配置文件放置到/opt/dble/conf目录下
- 调用/opt/dble/bin/dble start启动dble服务
- 调用脚本/opt/dble/bin/wait-for-it.sh 脚本监听dble 8066 服务，等待dble服务启动完成
- 调用mysql命令对于dble的后端数据库以及初始数据进行初始化

0.3.2 dble 镜像本地build

0.3.2.1 关于本节

- 本节内容为您介绍如何打包生成 dble 的 docker 镜像
- 如何使用 docker-compose 快速搭建dble不同运行容器环境

0.3.2.2 准备

- 安装docker
- 安装docker-compose

0.3.2.3 打包过程

若使用dockerhub中的镜像，则可跳过下面的步骤。

1. 可以通过下面链接：<http://blog.luckly-mjw.cn/tool-show/github-directory-downloader/index.html> 下载dble项目下的 <https://github.com/actiontech/dble/tree/master/docker-images> 目录
2. 将下载的文件 dble_master_docker-images.zip 解压

```
mkdir -p $working_dir
cd $working_dir
unzip dble_master_docker-images.zip
cd docker-images/dble-image
```

3. 执行命令生成 dble 镜像，其中 -t 指定 dble 镜像的 tag 名称，可以自己调整

```
docker build --build-arg MODE=quick-start --build-arg DBLE_VERSION=latest -t="actiontech/dble:latest" .
```

有两个编译参数：

MODE：指定 docker 镜像配置文件的模板样式，有三种，分别是：mgr, rwSplit, quick-start

DBLE_VERSION：指定 docker 镜像所使用的 dble 的版本

注意点：

1. 指定 MODE 后所使用的配置文件对应 \$working_dir/docker-images/dble-image 下面各个文件夹里面的配置文件，如果需要自定义，请自行调整
2. 3.20.10.0版本的配置文件有重大变动，因此和 3.20.10.0 之前版本的配置文件是不兼容的，因此若使用 3.20.10.0 之前的版本做镜像，需要调整配置文件

0.3.2.4 docker-compose 运行镜像

在 dble 的源码目录结构中，\$working_dir/docker-images 下分别有 mgr, quick-start, rwSplit 三个文件夹，每个文件夹中包含了各自运行的 docker-compose 文件，其中：

- mgr：启动两组mgr(六个mysql实例)，一个 dble
- quick-start：生成两个 mysql 实例，一个 dble
- rwSplit：生成一主一从两个 mysql 实例，一个 dble

用户根据自己的需求选择一个文件夹，执行以下命令启动：

```
docker-compose up -d
```

在使用或者测试完毕之后，在使用以下指令回收对应资源

```
docker-compose stop
docker-compose rm
```

0.3.2.5 验证效果

0.3.2.5.1 验证读写分离效果

1. 查看主从复制关系：

```

mysql> show slave status\G;
***** 1. row *****
      Slave_IO_State: Waiting for master to send event
        Master_Host: 10.186.61.151
          Master_User: user
          Master_Port: 33306
        Connect_Retry: 60
      Master_Log_File: mysql-bin.000004
  Read_Master_Log_Pos: 154
        Relay_Log_File: 4bad16278f02-relay-bin.000006
          Relay_Log_Pos: 367
  Relay_Master_Log_File: mysql-bin.000004
      Slave_IO_Running: Yes
      Slave_SQL_Running: Yes
      Replicate_Do_DB:
      Replicate_Ignore_DB:
      Replicate_Do_Table:
      Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
      Replicate_Wild_Ignore_Table:
          Last_Errno: 0
          Last_Error:
          Skip_Counter: 0
      Exec_Master_Log_Pos: 154
      Relay_Log_Space: 747
      Until_Condition: None
      Until_Log_File:
      Until_Log_Pos: 0
      Master_SSL_Allowed: No
      Master_SSL_CA_File:
      Master_SSL_CA_Path:
      Master_SSL_Cert:
      Master_SSL_Cipher:
      Master_SSL_Key:
      Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
          Last_IO_Errno: 0
          Last_IO_Error:
          Last_SQL_Errno: 0
          Last_SQL_Error:
      Replicate_Ignore_Server_Ids:
      Master_Server_Id: 1
          Master_UUID: 46bb9692-e5f3-11ea-8340-0242ac110002
      Master_Info_File: /var/lib/mysql/master.info
          SQL_Delay: 0
      SQL_Remaining_Delay: NULL
      Slave_SQL_Running_State: Slave has read all relay log; waiting for more updates
      Master_Retry_Count: 86400
      Master_Bind:
      Last_IO_Error_Timestamp:
      Last_SQL_Error_Timestamp:
      Master_SSL_Crl:
      Master_SSL_Crlpath:
      Retrieved_Gtid_Set:
      Executed_Gtid_Set:
          Auto_Position: 0
      Replicate_Rewrite_DB:
      Channel_Name:
      Master_TLS_Version:
1 row in set (0.00 sec)

```

0.3.2.5.2 验证mgr效果

1. 查看两组mgr关系:


```
[root@localhost]docker-images# docker exec mgr-a-1 mysql -h127.0.0.1 -p3306 -uroot -p123456 \
-e "SHOW STATUS LIKE 'group_replication_primary_member';" \
-e "SELECT * FROM performance_schema.replication_group_members;"

+-----+-----+
| Variable_name          | Value                               |
+-----+-----+
| group_replication_primary_member | 72da84d7-0c4b-11eb-9f0e-0242ac120002 |
+-----+-----+

+-----+-----+-----+-----+-----+
| CHANNEL_NAME          | MEMBER_ID                           | MEMBER_HOST | MEMBER_PORT | MEMBER_STATE |
+-----+-----+-----+-----+-----+
| group_replication_applier | 72da84d7-0c4b-11eb-9f0e-0242ac120002 | mgr-a-1     | 3306        | ONLINE       |
| group_replication_applier | 7314efdd-0c4b-11eb-ba28-0242ac120004 | mgr-a-3     | 3306        | ONLINE       |
| group_replication_applier | 733b00fe-0c4b-11eb-bbea-0242ac120003 | mgr-a-2     | 3306        | ONLINE       |
+-----+-----+-----+-----+-----+

[root@localhost]docker-images# docker exec mgr-b-1 mysql -h127.0.0.1 -p3306 -uroot -p123456 \
-e "SHOW STATUS LIKE 'group_replication_primary_member';" \
-e "SELECT * FROM performance_schema.replication_group_members;"

+-----+-----+
| Variable_name          | Value                               |
+-----+-----+
| group_replication_primary_member | 728c327d-0c4b-11eb-9300-0242ac120005 |
+-----+-----+

+-----+-----+-----+-----+-----+
| CHANNEL_NAME          | MEMBER_ID                           | MEMBER_HOST | MEMBER_PORT | MEMBER_STATE |
+-----+-----+-----+-----+-----+
| group_replication_applier | 728c327d-0c4b-11eb-9300-0242ac120005 | mgr-b-1     | 3306        | ONLINE       |
| group_replication_applier | 732c5b3b-0c4b-11eb-9eb1-0242ac120007 | mgr-b-3     | 3306        | ONLINE       |
| group_replication_applier | 733c6350-0c4b-11eb-b0fb-0242ac120006 | mgr-b-2     | 3306        | ONLINE       |
+-----+-----+-----+-----+-----+
```

2. 观察dblc日志, 进入dblc-server容器查看/opt/dblc/logs/下的相关日志

```
[root@localhost]docker-images# docker exec -it dblc-server bash
[root@dble-server /]# less /opt/dblc/logs/wrapper.log
[root@dble-server /]# less /opt/dblc/logs/dblc.log

#bootstrap.cnf配置中useOuterHa参数配置为false时, 才会有该日志生成
[root@dble-server /]# less /opt/dblc/logs/custom_mysql_ha.log
```

3. 进入dblc-server容器, 查看/opt/dblc/conf/db.xml配置如下:

```
[root@localhost]# docker exec -it dblc-server bash
[root@dble-server /]# cat /opt/dblc/conf/db.xml

<dbGroup name="dbGroup1" rwSplitMode="2" delayThreshold="10000">
  <heartbeat>show slave status</heartbeat>
  <dbInstance name="instanceM1" url="172.18.0.2:3306" user="root" password="123456" maxCon="300" minCon="10"
    primary="true" readWeight="1" id="xx1">
  </dbInstance>
  <dbInstance name="instanceS1" url="172.18.0.3:3306" user="root" password="123456" maxCon="1000" minCon="10" readWeight="2">
    <property name="testOnCreate">false</property>
  </dbInstance>
  <dbInstance name="instanceS2" url="172.18.0.4:3306" user="root" password="123456" maxCon="1000" minCon="10" readWeight="2">
    <property name="testOnCreate">false</property>
  </dbInstance>
</dbGroup>
```

4. 停用mgr-a-1的主实例, 进入dblc-server容器中查看变化: custom_mysql_ha.log中出现172.18.0.2:3066...is not alive、db.xml中dbGroup1组的主实例为instanceS1:

```
[root@localhost]docker-images# docker-compose stop mgr-a-1
[root@localhost]docker-images# docker exec -it dble-server bash
[root@dble-server /]# less /opt/dble/logs/custom_mysql_ha.log
...
2020-10-12 07:05:08 [DBLEDbGroupsCheck] [INFO] DbInstance 172.18.0.2:3306 in dbGroup1 is not alive!
2020-10-12 07:05:08 [DBLEDbGroupsCheck] [INFO] DbInstance 172.18.0.3:3306 in dbGroup1 is normal!
2020-10-12 07:05:08 [DBLEDbGroupsCheck] [INFO] DbInstance 172.18.0.4:3306 in dbGroup1 is normal!
2020-10-12 07:05:08 [DBLEDbGroupsCheck] [INFO] DbInstance 172.18.0.5:3306 in dbGroup2 is normal!
2020-10-12 07:05:08 [DBLEDbGroupsCheck] [INFO] DbInstance 172.18.0.6:3306 in dbGroup2 is normal!
2020-10-12 07:05:08 [DBLEDbGroupsCheck] [INFO] DbInstance 172.18.0.7:3306 in dbGroup2 is normal!
...

[root@dble-server /]# cat /opt/dble/conf/db.xml

<dbGroup name="dbGroup1" rwSplitMode="2" delayThreshold="10000">
  <heartbeat>show slave status</heartbeat>
  <dbInstance name="instanceM1" url="172.18.0.2:3306" user="root" password="123456" maxCon="300" minCon="10"
    readWeight="1" id="xx1">
  </dbInstance>
  <dbInstance name="instanceS1" url="172.18.0.3:3306" user="root" password="123456" maxCon="1000" minCon="10" readWeight="2" primary=
"true">
    <property name="testOnCreate">false</property>
  </dbInstance>
  <dbInstance name="instanceS2" url="172.18.0.4:3306" user="root" password="123456" maxCon="1000" minCon="10" readWeight="2">
    <property name="testOnCreate">false</property>
  </dbInstance>
</dbGroup>
```

0.4 数据拆分简介

0.4.1 数据拆分简介

- 在dble中将表格按照数据分片的大致方式将表格的归为三个种类
 - 全局表：设置为全局表的表格将会在每个mysql节点上存在一个实体，并且在每个表格实体里面存放的都是全量的数据，一般用作字典表之类的数据量小、和多表关联或者是作为数据字典的表格。
 - 拆分表：设置为拆分表的表格会根据具体选择的拆分算法类型将其中的数据按照一定的规则分别存放不同的mysql节点中去，每个节点存放一部分的数据。一般用以存放超大数据量的业务类表格，通过对于业务类表格的水平数据拆分实现性能的优化。
 - 非拆分表：设置为非拆分表的表格会在指定的mysql节点上单独存在一个表格实体，在此表格中存放全量数据。一般用于存放数据压力不大的业务类表格，类似冷门功能的业务数据表。

0.4.2 规划拆分方案

- 系统开发之前，应该对业务特点进行深度分析，对数据规模进行较准确的评估，根据表的数据量、数据特点和表与表之间的关系，决定哪些表需要分片。数据节点的数量应该根据数据量和访问性能要求合理配置，过多的节点数量不仅浪费资源，而且增加运维复杂度，有时不仅不能提升性能,还会降低性能。
- 对于小表（数据量不大，如千万以下），尽量不要配置成拆分表。如果表比较独立，与其他的表基本上不进行join运算，可以作为非拆分表处理，性能不能满足要求的时候可以通过配置读写分离机制来提高性能。如果需要与拆分表进行join运算，可以配置成全局表。
- 关于每个实例的规格，选择的根据有两个因素：
 - 需要存储的最大磁盘空间，需要通过拆分算法计算，根据存储数据量最大的节点来计算，并预估未来2到3年的数据增长；
 - 估算一个实例需要的最大 QPS和TPS，要根据最慢的节点估算。
- QPS、TPS与实际的SQL语句、数据量、数据结构和数据节点的规格有关，根据经验来估计的话，很可能偏差较大。系统建设之前，应该配置同等或者接近的环境，进行针对性的性能测试，从而做出准确的判断。

0.4.3 数据拆分表格的配置方法

- 数据拆分配置包括节点配置和拆分规则配置。节点配置决定了数据的物理存储方式，由主机和节点组成，主机代表具体的数据库实例，节点代表实例中的数据库。拆分规则决定了数据在不同的分区上读写的规则，由拆分算法和应用拆分算法的逻辑库、表、字段组成。
- 数据写入时，系统对指定的字段值应用拆分算法得到目标节点，然后将数据写入目标节点。数据读取时，系统对查询条件应用拆分算法得到数据源节点，从源节点获取数据，在中间层进行结果合并之后返回请求方，对于不同的查询条件，源节点可能有一个或多个。

1.dble配置文件基础

- 配置文档列表以及相关对应功能
 - [cluster.cnf](#):集群参数配置
 - [bootstrap.cnf](#):实例参数配置，包括JVM启动参数，dble性能，定时任务，端口等
 - [user.xml](#):dble 用户配置
 - [db.xml](#): 数据库相关配置
 - [sharding.xml](#): 数据拆分相关配置
 - [log4j.xml](#): log4j2.xml，配置日志参数
 - 全局序列: 全局序列相关配置
 - [cache配置](#): 配置缓存参数
 - [自定义拆分算法](#)
 - [自定义告警](#)
 - [自定义全局表一致性检查](#)
 - [Schema下默认拆分表](#)
- 重要日志及文件:
 - [/logs/wrapper.log](#): 启动日志，如果dble启动失败，将会有日志出现在这个文件中
 - [/logs/dble.log](#): dble日志，日志记录并反馈dble执行过程中的重要信息
- 配置文件变更记录:
 - [配置文件变更记录](#)

dble 在3.20.07.0 版本做了配置的重构。历史变更请参考[2.20.04.0的变更](#)

2. 配置升级

可以通过升级工具dble_update_config将配置从2.20.04.0 升级到3.20.07.0，如果是更早版本，建议先升级到2.20.04.0

升级工具下载地址:

- AMD架构——[dble_update_config](#)
- ARM架构——[dble_update_config_arm64](#)

升级工具用法:

```
dble_update_config/dble_update_config_arm64 [-i=read_dir] [-o=write_dir] [-p=rootPath]
```

read_dir/write_dir: 如果不指定，缺省值为当前目录，建议指定或者提前备份配置 rootPath:如果集群模式是zk, 那么缺省值为 `/dble` , 如果集群模式是ucore, 缺省值为 `universe/dble`

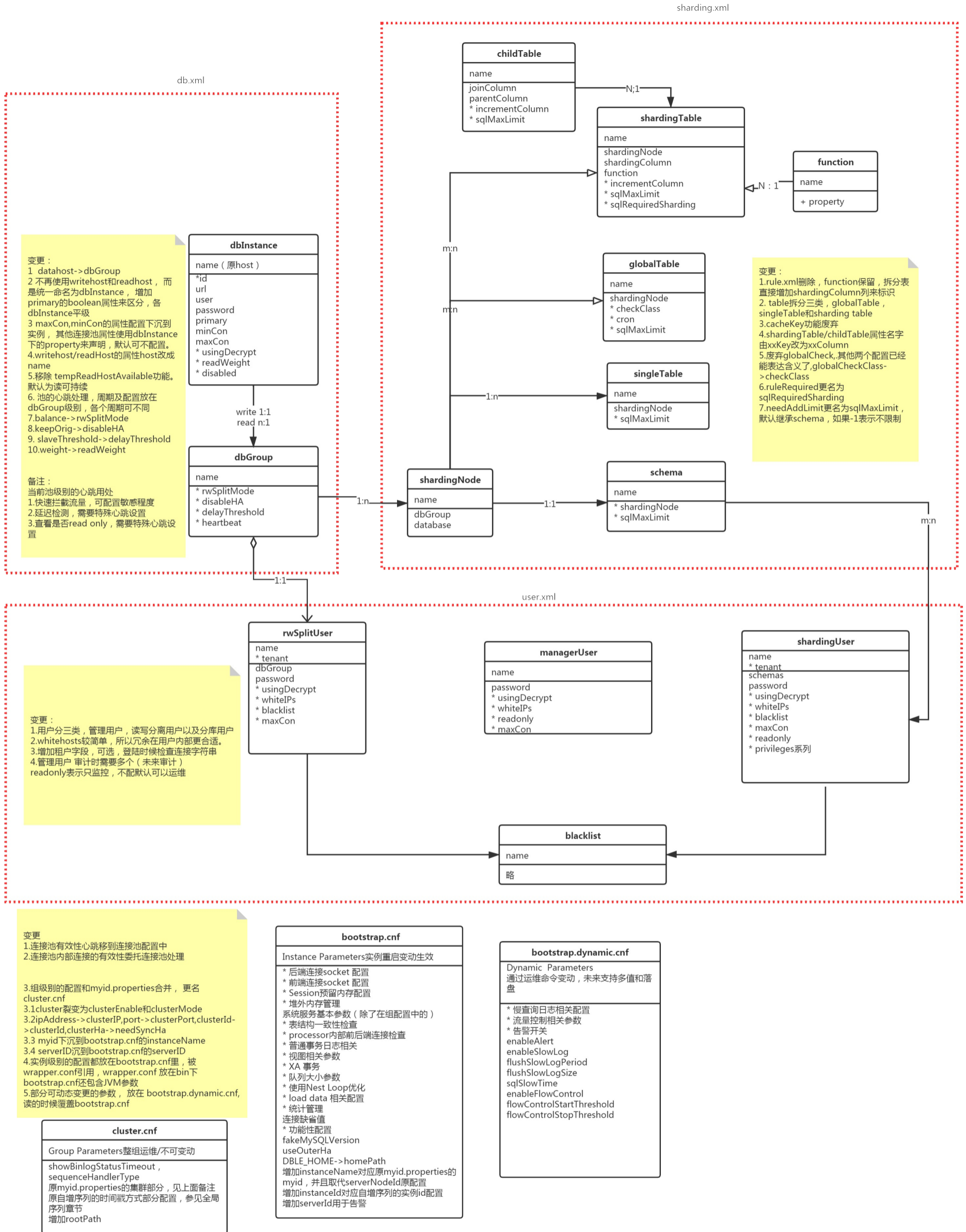
工具将会读取文件:

```
myid.properties
wrapper.conf
server.xml
schema.xml
rule.xml
log4j2.xml
cacheservice.properties(option)
sequence_distributed_conf.properties for type3 (option)
sequence_time_conf.properties for type2 (option)
```

然后写出文件:

```
cluster.cnf
bootstrap.cnf
user.xml
db.xml
sharding.xml
log4j2.xml
cacheservice.properties(option)
```

3. 重构后的配置概览图:



1.1 cluster.cnf

本配置文件为key=value的形式，用于配置在一个dble集群中各个dble实例必须一致的一些参数，key/value值大小写敏感。如果没有此文件启动dble，将不具有集群功能（配置同步，DDL同步等），并且部分参数使用默认值。

注意：本配置文件如果需要修改,需要多个dble节点停下来，修改配置，重新初始化集群元数据(zk可删除原有元数据),然后挨个启动dble
某些属性在生产环境不建议修改，例如sequenceHandlerType，它的各个值之间生成的序列是不兼容的，修改可能会破坏序列的唯一性。

配置名称	配置内容&示例	可选项/默认值	详细描述
clusterEnable	是否开启集群功能	非必需项，可配置true/false。默认值为false	标记是否开启集群功能，如果开启，需要配置clusterIP，clusterPort，rootPath，clusterID
clusterMode	集群调度中心的模式	可配置zk/ucore。无默认值，clusterEnable开启后不配此项，会导致启动报错	zk表示集群调度中心使用zookeeper，ucore表示使用爱可生商业集群调度中心
clusterIP	集群调度中心的IP	无默认值，clusterEnable开启后不配此项，会导致启动报错	clusterMode为zk时，配置zk完整地址，例如10.186.19.aa:2281,10.186.60.bb:2281；clusterMode为core时，配置ucore的ip地址，可为逗号隔开的集群IP地址
clusterPort	集群调度中心的端口	无默认值	clusterMode为zk时，此项可以空缺；clusterMode为ucore时，配置ucore的端口号。
rootPath	集群调度中心的根目录	无默认值，clusterEnable开启后不配此项，会导致启动报错	集群调度中心的根目录，按需配置
clusterId	本dble集群的名称	无默认值，clusterEnable开启后不配此项，会导致启动报错	本dble集群的名称，多个使用同一实例名称的dble会被视为在同一集群下
needSyncHa	ha接口是否需要集群同步	可配置true/false。默认值为false	当使用高可用接口时，是否需要使用集群同步，当配置为true时，实例级别的参数useOuterHa会被置为true
showBinlogStatusTimeout	拉取一致性binlog线的超时时间	可配置正整数，默认60000,单位毫秒	拉取一致性binlog线的超时时间
sequenceHandlerType	全局序列处理器的方式	可配置1~4，默认值2	<p>在初始化的时候根据这个配置选择不同的序列生成器进行加载</p> <p>1, MySQL offset-step序列方式, sequence信息存储在数据库中</p> <p>2, 时间戳方式(类Snowflake)</p> <p>3, 分布式time序列(类Snowflake)</p> <p>4, 分布式offset-step序列</p>
sequenceStartTime	时间相关的全局序列的起始时间	非必需项，默认2010-11-04 09:42:54，修改需要保持这个格式	仅当sequenceHandlerType为2或3时候这个值有意义
sequenceInstanceByZk	分布式time序列是否使用zk来生成唯一实例id	非必需项，可以配置true/false,默认true	仅当sequenceHandlerType为3，并且clusterMode为zk时，这个值有意义
grpcTimeout	使用ucore时，grpc调用的超时时间	非必需项，单位为秒，默认10s	仅当clusterMode为ucore时，这个值有意义

1.1.1 不使用的例子

不配置 或者

```
clusterEnable=false
#showBinlogStatusTimeout=60000
sequenceHandlerType=2
#sequenceStartTime=2010-11-04 09:42:54
#sequenceInstanceByZk=true
```

1.1.2 使用ZK的例子

```
clusterEnable=true
clusterMode=zk
clusterIP=10.186.19.aa:2281,10.186.60.bb:2281
rootPath=/db1e
clusterId=cluster-1
#needSyncHa=false
#showBinlogStatusTimeout=60000
sequenceHandlerType=2
#sequenceStartTime=2010-11-04 09:42:54
#sequenceInstanceByZk=true
```

1.1.3 使用ucore的例子

```
clusterEnable=true
clusterMode=ucore
clusterIP=10.186.19.aa,10.186.60.bb
clusterPort=5700
rootPath=universe/db1e
clusterId=cluster-1
#needSyncHa=false
#showBinlogStatusTimeout=60000
sequenceHandlerType=2
#sequenceStartTime=2010-11-04 09:42:54
#sequenceInstanceByZk=true
#grpcTimeout=10
```

1.2 bootstrap.cnf

本配置文件为dble实例启动时候加载的配置文件。默认使用wrapper.cnf启动的外置参数文件的格式，即使不使用wrapper启动时，也会加载此文件。

1.2.1 jvm启动参数

本文件第一部分为JVM启动参数,可以根据需要修改。

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8088,suspend=n
-server
-XX:+AggressiveOpts
-Dfile.encoding=UTF-8
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=1984
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.host=127.0.0.1
-Xmx4G
-Xms1G
-XX:MaxDirectMemorySize=2G
```

大部分以上的配置都没有特殊的意义，仅仅是一般的JVM配置，关于JVM调优的部分需要以现实情况进行操作，在此仅介绍以个特殊情况

MaxDirectMemorySize上限值为81917M，约等于79G。需要根据机器的情况进行提前适配，不然会导致服务无法正常启动 具体的细节为需要大于bufferPoolPageNumber*bufferPoolPageSize，这两个选项在第二部分中配置

bufferPoolPageNumber 的默认值= (MaxDirectMemorySize * 0.8 /bufferPoolPageSize), 向下取整，上限值为32767

bufferPoolPageSize 的默认值= 2M

以下为建议值：

dble总内存=0.6 可用物理内存(刨除操作系统,驱动等的占用)

Xmx = 0.4 dble总内存

MaxDirectMemorySize = 0.6 * dble总内存

另外，在启动参数中的bufferPoolPageNumber 和bufferPoolPageSize 受MaxDirectMemorySize影响。

1.2.2 dble系统参数

本文件第二部分为dble系统参数,配置格式遵照启动参数的格式：-Dkey=value，注意如果是通过JSW启动的dble服务，需要复合JSW对配置文件的要求，也就是不能有空格，具体参见 [wrapper.app.parameter_file](#)

具体系统参数含义参见以下表格。

是否开启sql统计

模块	配置名称	配置内容	默认值/单位	详细作用原理或应用	配置范围
	homePath	基本目录 慢查询日志 (slowLogs)、视图记录日志 (viewConf)、xa的tm日志 (xalogs)、load data临时文件 (temp) 等存放路径的父目录	无默认值,不配置会报错	其他一些类似于事务或者视图存储的的根路径	有效路径,推荐配置为当前目录
	instanceName	实例名称	无默认值，不配置会报错	集群配置时的唯一标识，xa事务时的实例标识	集群内唯一值
	instanceId	实例id	无默认值	集群配置时的唯一标识，全局序列时的唯一标识	仅当sequenceHandlerType=2 或者 3时候有意义。 当sequenceHandlerType=2，合法值范围为0~1023

系统服务基本参数					当 sequenceHandlerType=3, 合法值范围为 0~511
	serverId	服务器名称	默认值为服务器IP	dble所在机器的名称	用于告警时候报告服务器名称
	bindIp	服务IP	默认 "0.0.0.0"	在服务初始化的时候作为侦听的IP	有效IP地址, 推荐默认
	serverPort	服务端口	默认8066	在服务初始化的时候作为服务侦听的端口	机器空闲端口
	managerPort	控制端口	默认9066	在服务初始化的时候作为控制侦听的	机器空闲端口
	maxCon	控制最大连接数	默认0	默认不做限制。若 maxCon大于0, 建立的连接数大于maxCon之后, 建立连接会失败. 注意当各个用户的 maxcon总和值大于此值时, 以当前值为准。全局maxCon不作用于manager用户	正整数
	NIOFrontRW	NIO前端处理器的数量, 兼容旧参数 processors, 同时配置NIOFrontRW和 processors, 以NIOFrontRW为准	默认java虚拟机核数, 单位个	进行前端网络IO吞吐的线程数	正整数
	NIOBackendRW	NIO后端处理器的数量, 兼容旧参数 backendProcessors, 同时配置NIOBackendRW和 backendProcessors, 以NIOBackendRW为准	默认java虚拟机核数, 单位个	进行后端网络IO吞吐的线程数	正整数
	frontWorker	前端业务处理线程池数量, 兼容旧参数 processorExecutor, 同时配置frontWorker和 processorExecutor, 以frontWorker为准	默认 (单核为2, 否则等于宿主主机核数)	进行前端具体业务处理的线程池大小, 负责解析路由下发	正整数
	managerFrontWorker	前侧管理端线程池数量	默认 (单核为2, 否则等于宿主主机核数/2)	负责前侧管理端执行&处理命令的线程池大小	正整数
backendWorker	后端业务处理线程池数量, 兼容旧参数 backendProcessorExecutor, 同时配置backendWorker和 backendProcessorExecutor, 以backendWorker为准	默认 (单核为2, 否则等于宿主主机核数)	进行后端具体业务处理的线程池大小, 负责回收结果集合并	正整数	

	complexQueryWorker	复杂查询后端业务线程池数量，兼容旧参数complexExecutor，同时配置complexQueryWorker和complexExecutor，以complexQueryWorker为准	默认 (单核为2,否则等于宿主机核数，宿主机核数大于8时，数量为8)	负责复杂查询或者子命令结果集的回收	正整数
	writeToBackendWorker	广播下发SQL时候批量处理的线程池，兼容旧参数writeToBackendExecutor，同时配置writeToBackendWorker和writeToBackendExecutor，以writeToBackendWorker为准	默认 (单核为2,否则等于宿主机核数)	负责广播下发SQL时候批量处理	正整数
	fakeMySQLVersion	Dble模拟Mysql版本号	默认NULL	模拟成正常的MySQL版本,用于与客户端协议交互 + 分布式使用规则: 填写的版本号不能高于后端Mysql节点的最低版本号,否则启动失败 + 读写分离使用规则: 填写的版本号和后端MySQL节点的版本号需要在范围上保持一致,否则启动失败。 MySQL官方认为5.7.20<=版本<8.0.0和="" 版本="">=8.0.3是一致的,除此之外其它的版本则认为是同一种情况。例如以下的配置情况: 1.fakeMySQLVersion : 5.7.20 mysql-version: 5.7.25 2.fakeMySQLVersion : 8.0.3 mysql-version: 8.0.23 3.fakeMySQLVersion : 5.7.15 mysql-version: 8.0.1 建议: 填写的大版本号与后端Mysql节点的大版本号保持一致	MYSQL版本号
	serverBacklog	前端tcp连接 backlog	默认2048	前端tcp连接 backlog	正整数
	usePerformanceMode	是否启用性能模式	默认0/单位无	开启之后Dble会大量占用CPU资源,并提供更高的性能体现,慎用	1-是0-否
	useOuterHa	是否启用外部高可用联动	默认为true,若此时不设置外部高可用,将不做切换	如关闭此功能并且dble部署方式为单机,将使用默认的切换方式,详情请见切换相关章节	true/false
groupConcatMaxLen	GROUP CONCAT()函数允许的最大结果长度	默认为1024	GROUP CONCAT()函数允许的最大结果长度,以字节为单位	正整数	
	charset	字符集	utf8mb4	服务启动后的默认字符集于所有字符集相关的部分,包括前端连接和后端连接	有效字符集

连接缺省值	maxPacketSize	包大小限制	默认 4×1024×1024	限制请求的包大小，启动时候dble会拉取并尝试同步(此值+1024)到每个dbInstance，如果同步失败，就取配置值与各个dbInstance中最小的那个值-1024. 留出1024的冗余用于对SQL改写或者上下文同步的支持	正整数
	txIsolation	隔离级别	默认 3	执行具体SQL的时候会比较前后端连接， 启动时候dble会拉取并尝试同步此值到每个dbInstance，如果同步失败或者session级别重新设置该值，session在SQL下发之前，会执行session级别的隔离级别set	1- READ_UNCOMMITTED 2- READ_COMMITTED 3- REPEATABLE_READ 4-SERIALIZABLE
	autocommit	是否自动提交	默认 1，自动提交	启动时候dble会拉取并尝试同步此值到每个dbInstance 如果同步失败或者session级别重新设置该值，执行具体SQL的时候会比较，如果不一致将会执行session级别的set	0/1
功能性配置	useCompression	是否启用数据压缩	默认 0否	使用mysql压缩协议	1 - 是 0 - 否
	capClientFoundRows	是否开启 Client_Found_Rows 权能标识	默认 false，关闭	dble开启 Client_Found_Rows 权能标识	true - 开启 false - 关闭
	usingAIO	是否启用AIO	默认0	在初始化服务的时候将会作为判断启用AIO或是NIO的依据	1 - 是 0 - 否
线程使用率统计	useThreadUsageStat	开启线程使用率统计	默认0/单位无	开启之后能在管理端通过管理命令show @@thread_used查看各个部分的线程使用情况	1-是0-否
	useCostTimeStat	是否启用查询耗时统计	默认0/单位无	开启之后以一定的比例统计查询过程中的各个步骤的耗时情况，可以使用BTraceCostTime.java进行观测,也可在管理端使用show @@cost_time观察	1-是0-否
	maxCostStatSize		默认100	show @@cost_time结果最近保留的行数	
	costSamplePercent	查询采样百分比	默认1/单位%	在耗时采样统计中实际采样百分比为costSamplePercent	
				如果值为1，那么在服务初始化的时候会启动一个定时任务，在	

一致性检查	checkTableConsistency	表格一致性检查	默认0	定时任务会检查DB是不是存在，表格是不是存在，表结构是否一致	1-是,0-否
	checkTableConsistencyPeriod	表格一致性检查周期	默认30×60×1000,单位毫秒	表格一致性检查周期	正整数
processor内部前后端连接检查	sqlExecuteTimeout	后端连接执行超时时间	默认 300,单位秒	如果超过这个时间没有完毕，就直接关闭连接	正整数
	idleTimeout	前端连接无响应超时时间	默认 10 × 60 × 1000,单位毫秒	在processor定时连接检查时，发现前端连接上一次的读写距今超过阈值，会直接关闭连接	正整数
	processorCheckPeriod	processor定时任务检查周期	1000,单位毫秒	根据此配置定时的检查在processor中的前后端连接的状态	正整数
后端连接socket 配置	backSocketSoRcvbuf	后端套接字接收缓冲区大小	1024×1024×4,单位字节	在创建后端管道的时候作为buffer大小使用	正整数
	backSocketSoSndbuf	后端套接字发送缓冲区大小	1024×1024,单位字节	在创建后端管道的时候作为buffer大小使用	正整数
	backSocketNoDelay	后端Nagle算法是否禁用	默认1/单位无	在创建后端管道的时候禁用延迟加载，会影响网络包的情况 详见 相关资料	1-是,0-否
前端连接socket 配置	frontSocketSoRcvbuf	前端套接字接受缓冲区大小	1024 × 1024,单位字节	在读取网络传输信息的时候作为每次缓冲的大小使用	正整数
	frontSocketSoSndbuf	前端套接字发送缓冲区大小	1024×1024×4,单位字节	在创建前端管道的时候作为buffer大小使用	正整数
	frontSocketNoDelay	前端Nagle算法是否禁用	默认1	在创建前端管道的时候禁用延迟加载 相关资料	1-是,0-否
Session预留内存配置	orderMemSize	session中的复杂查询order预留内存	默认4,单位M	在session初始化的时候创建内存分配对象，在复杂查询order by的时候使用到	正整数
	otherMemSize	session中的复杂查询其他预留内存	默认4,单位M	在session初始化的时候创建内存分配对象，在复杂查询subQuery以及distinctd的时候使用	正整数
	joinMemSize	session中的复杂查询join预留内存	默认4, 单位M	在session初始化的时候创建内存分配对象，在复杂查询join使用到	正整数

堆外内存管理	bufferPoolChunkSize	内存池中分配的最小粒度	默认4096,单位字节	内存池中分配的最小粒度,需要的大小除以此粒度,向上取整	
	bufferPoolPageNumber	预分配内存池页数量	默认 $0.8 \times \text{MaxDirectMemorySize} / \text{bufferPoolPageSize}$ (default 2M), 向下取整	在初始化的时候通过和bufferPoolPageSize的相乘确定缓冲池最后的大小,内存配置建议见 启动参数	
	bufferPoolPageSize	预分配内存池页大小	默认 $1024 \times 1024 \times 2$, 单位字节	在初始化的时候通过和bufferPoolPageNumber的相乘确定缓冲池最后的大小, 注意: 虚拟机参数 <i>MaxDirectMemorySize</i>(见启动参数)需要大于 <i>bufferPoolPageNumber * bufferPoolPageSize</i>, 否则会触发OOM	
	mappedFileSize	文件映射区单个文件最大体积	默认 $1024 \times 1024 \times 64$, 单位字节	在初始化的时候此参数确定文件映射区最大容量,参见内存管理章节	
maxResultSet	大结果集阈值	默认 512×1024 , 单位字节	当查询的结果集超过这个阈值时, 查询的SQL和查询结果集的大小才会被记录到结果集统计里面	正整数	
enableSessionActiveRatioStat	统计前端连接繁忙率开关	默认为1, 开启	统计前端连接繁忙率开关; 当开启性能模式(-DusePerformanceMode=1)时, 此配置不生效	0或者1	
enableConnectionAssociateThread	记录当前时间, 前端/后端连接分别使用线程情况的开关	默认为1, 开启	记录当前时间, 前端/后端连接分别使用线程情况的开关; 当开启性能模式(-DusePerformanceMode=1)时, 此配置不生效	0或者1	
普通事务日志相关	recordTxn	事务log记录	默认0	在初始化服务的时候会注册一个类, 其作用就是将事务的log写到一个指定的文件中	1-是, 0-否
	transactionLogBaseDir	事务log目录	默认当前路径/txlogs	当开启日志log记录时, 记录文件会被存放在对应目录下	绝对路径
	transactionLogBaseName	事务log文件名称	默认server-tx	事务记录存储文件的文件名	符合运行系统文件的命名规范
	transactionRotateSize	事务日志单个文件大小。	默认16, 单位M		正整数
	xaRecoveryLogBaseDir	xa的tm日志路径	dble目录/xalogs/	此日志涉及到XA事务状态的记录, 并且在Dble意外重启之后需要从里面获取重启之前的xa事务状态, 切	绝对路径

				勿自行修改	
XA 事务	xaRecoveryLogBaseName	xa的tm日志名称	xalog		符合运行系统文件的命名规范
	xaSessionCheckPeriod	XA定时任务执行周期	默认1000, 单位ms	在server开始的时候会注册一个定时任务以此参数为执行周期 (注: 定时任务必定会被注册) 如果有尝试多次没有成功提交的session在之后的定时任务会被重复提交	正整数
	xaLogCleanPeriod	定时XAlog清除周期	默认1000, 单位ms	在server开始的时候会根据这个周期注册一个定时任务 (注: 定时任务必定会被注册) 定时清XA log, 主要是将已经回滚和提交成功的一部分从记录中删除	正整数
	xaRetryCount	后台重试XA次数	默认0	后台定时任务重试XA次数, 0为无限重试, 达到设定次数后, 停止重试	正整数
	xaldCheckPeriod	检测疑似残留XA任务的周期	默认300, 单位s	后台定时检测疑似残留Xid任务; 如果设置小于等于0, 则表示不开启此检测定时任务	正整数
视图相关参数	viewPersistenceConfBaseDir	视图记录本地文件路径	dble目录/viewConf	用于存放视图本地记录文件的文件路径, 集群配置时无意义	绝对路径
	viewPersistenceConfBaseName	视图记录本地文件名	viewJson	视图记录的文件文件名, 集群配置时无意义	符合运行系统文件的命名规范
队列大小参数	joinQueueSize	join时, 左右结点的暂存数据行数的队列大小	1024	当行数大于此值而又没有及时被消费者消费掉, 将会阻塞, 目的是防止接收数据量太大, 堆积在内存中	正整数
	mergeQueueSize	merge时, 左右结点的暂存数据行数的队列大小	1024	当行数大于此值而又没有及时被消费者消费掉, 将会阻塞, 目的是防止接收数据量太大, 堆积在内存中	正整数
	orderByQueueSize	排序时, 左右结点的暂存数据行数的队列大小	1024	当行数大于此值而又没有及时被消费者消费掉, 将会阻塞, 目的是防止接收数据量太大, 堆积在内存中	正整数
	useJoinStrategy	是否使用nest loop 优化	默认不使用	开启之后会尝试判断join两边的where来重新调整查询SQL下发的顺序	true 开启 false 不开启
				值为-1时, 如果useJoinStrategy=true, 那么进入useJoinStrategy的逻辑, 否则不进行	

join相关的策略	joinStrategyType	nest loop 优化策略	默认值为-1，根据不同数值使用不同的nestloop策略	nestloop处理，值为0时，不进行nestloop处理(无视useJoinStrategy是否开启)，值为1时进入useJoinStrategy的逻辑处理(无视useJoinStrategy是否开启)，值为2时，进入alwaysTryNestLoop的逻辑处理(无视useJoinStrategy是否开启)	允许范围在-1到2之间
	nestLoopConnSize	临时表阈值	默认4	若临时表行数大于这两个值乘积，则报告错误	正整数
	nestLoopRowsSize	临时表阈值	默认2000		
	inSubQueryTransformToJoin	in子查询转成join进行查询	默认false（默认不使用）	in子查询在dblc内部可以尝试转成join处理，也可以选择不开启。可以通过查询计划比较两种方式的执行（执行过程有所区别，性能也会因为sql的不同而有不同的表现），开启与否不影响最终结果	true 开启 false 不开启
慢查询日志相关配置	enableSlowLog	慢查询日志开关	默认为0，关闭	慢查询日志开关	0或者1
	slowLogBaseDir	慢查询日志存储文件夹	dblc根目录/slowlogs	慢查询日志存储文件夹	文件夹路径
	slowLogBaseName	慢查询日志存储文件名前缀	slow-query	慢查询日志存储文件名前缀(后缀名是.log)	合法文件名
	flushSlowLogPeriod	日志刷盘周期，单位秒	1	日志刷盘周期，每隔这个周期，会强制将内存数据刷入磁盘	正整数
	flushSlowLogSize	日志刷盘条数阈值	1000	日志刷盘条数阈值，内存中每次写出这么多条日志，会强制刷盘1次	正整数
	sqlSlowTime	慢日志时间阈值，单位毫秒	100	慢日志时间阈值，大于此时间的查询会记录下来	正整数或者0
	slowQueueOverflowPolicy	慢日志队列无空间时，后续日志的处理策略	2	慢日志队列存储的是等待落盘的慢日志，目前队列固定长度为2000，当队列满了，后续慢日志会根据不同的策略进行不同的处理 策略1：保障持续服务业务：当遇到慢查询超过等待写入阈值时，丢弃当前产生的慢SQL，并发出告警； 策略2：尽量保证慢日志不丢失：当遇到慢查询超过等待写入阈值时，触发阻塞队列机制，新的慢SQL等待加入落盘队列，并触发告警；最大阻塞时间为3s，超时会丢弃当前慢日志的记录	1或2

load data 相关配置	maxCharsPerColumn	每列所允许最大字符数	默认为65535	每列所允许最大字符数	正整数
	maxRowSizeToFile	需要持久化的最大行数，在开启load data批处理模式下是拆分文件的阈值	默认为100000	当load data的数据行数超过阈值后，会将数据保存在文件中以防OOM。在开启批处理模式后load data的数据行数超过阈值后，会将该文件按照阈值拆分成多个文件进行保存，拆分过程中最后的数据行不到阈值放入在最后一个拆分的文件中，而不是再单独创建新文件存放	正整数
	enableBatchLoadData	是否启用load data的批处理模式	默认为0	load data会将导入文件按照maxRowSizeToFile的值拆分成多个文件分批导入	正整数，0为不开启，1为开启。其他数字无效
流量控制相关参数	enableFlowControl	是否启用流量控制，true/false	默认为false	具体流量控制请参见相关功能描述章节	true/false
	flowControlHighLevel	触发流量控制的前端连接水位	默认为4194304 (4096K)	当部分前端连接的写出队列的字节数超出水位时触发流量控制，单位为字节数	正整数
	flowControlLowLevel	流量控制取消的前端连接水位	默认为262144 (256K)	当流量控制中的前端连接写出队列的字节数小于水位，则取消流量控制，单位为字节数	正整数
游标相关参数	enableCursor	是否开启 server-side-cursor.	默认false	注意游标功能必须客户端和服务端同时开启才有效。且目前只支持分库分表场景使用。另外，如果客户端没开，服务端开启了，运行 prepare statement依旧会损失一小部分性能。具体见 4.4	true or false
	maxHeapTableSize	临时表在内存中存储的最大大小,单位byte	默认为4096	临时表在内存中存储的最大大小，超过这个大小会被落盘	整数，大于等于 0
	heapTableBufferChunkSize	读 buffer 的 cache 的大小	默认等于bufferPoolChunkSize, 单位byte	读取临时表临时文件时，读 buffer 的大小	正整数，必须是bufferPoolChunkSize的整数倍
	enableGeneralLog	是否开启general日志	默认为0，关闭	开启后会将所有接收的sql记录在general日志文件中	0：关闭/1：开启
	generalLogFile	general日志文件地址；	默认为general/general.log	若配置以 '/' 开头则作为绝对路径生效，反之，则在homepath后拼接得到最终绝对路径且生效	符合运行系统 文件的命名规范

general日志相关参数	generalLogFileSize	general日志文件大小;	默认16M	当general.log超过其大小则将会生成yyyy-MM/general-MM-dd-%d.log (默认格式)文件; 类似与log4j	正整数
	generalLogQueueSize	处理general日志的队列大小	默认4096	涉及内部实现机制; (类似与log4j的AsyncLogger)	正整数且必须为2的次方
sql statistic相关参数	enableStatistic	是否开启全量统计	默认0	是否开启全量统计	1: 开启, 0: 不开启
	enableStatisticAnalysis	是否开启sql对user、table、condition聚合统计	默认为0, 即不开启	作为show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition聚合统计的开关	1: 开启, 0: 不开启
	associateTablesByEntryByUserTableSize	sql_statistic_by_associate_tables_by_entry_by_user表的大小	默认1024	超过其大小值, 则淘汰溢出的历史数据	整数且大于1
	frontendByBackendByEntryByUserTableSize	sql_statistic_by_frontend_by_backend_by_entry_by_user表的大小	默认1024	超过其大小值, 则淘汰溢出的历史数据	整数且大于1
	tableByUserByEntryTableSize	sql_statistic_by_table_by_user_by_entry表的大小	默认1024	超过其大小值, 则淘汰溢出的历史数据	整数且大于1
	statisticQueueSize	处理sql statistic的队列大小	默认4096	涉及内部实现机制; (类似与log4j的AsyncLogger)	正整数且必须为2的次方
	samplingRate	sql抽样统计的采样率	默认为100, 即开启全量统计	samplingRate是个百分数。假如有100条事务进入dblc, 采样率设置为4, 此时从100条事务中随机采样4条, 且单个事务中的所有语句都会记录。	[0,100]之间的正整数
	sqlLogTableSize	sql log 表格大小	默认1024	超过其大小值, 则淘汰最旧的一条事务(在开启事务的情况下, 一个事务中可能包含多个sql)	正整数
读写分离相关配置	rwStickyTime	读写分离场景下, 主(写)从(读)实例的粘滞时间段	默认1000(ms), 若设置为0, 则表示不开启主(写)从(读)实例的粘滞	执行当前读SQL的时间, 距离上一次写SQL执行的时间段, 没有超过rwStickyTime时间段时, 则当前读SQL将会下发至后端主(写)实例。	正整数
	district	dblc配置所属区域, 读写分离本地读场景下使用	默认为null	读写分离本地读场景下读流量转发依据, 该参数会和db.xml中dbInstance下的dbDistrict参数匹配	有效字符集

	dataCenter	dble配置所属数据中心,读写分离本地读场景下使用	默认为null	读写分离本地读场景下读流量转发依据,该参数会和db.xml中dbInstance下的dbDataCenter参数匹配	有效字符集
堆外内存泄露监控相关	enableMemoryBufferMonitor	是否开启堆外内存泄露监控	0	是否开启堆外内存泄露监控	0表示关闭, 1 表示开启
	enableMemoryBufferMonitorRecordPool	是否记录连接池的常驻内存	1	是否记录连接池的常驻内存,一般不用修改。该类内存通常被连接池一直持有且不释放,属于正常现象也不属于泄露。如果对观测产生了困扰可关闭	0表示关闭, 1 表示开启
审计日志相关参数	enableSqlDumpLog	是否开启审计日志	默认为0, 关闭	开启后会记录读写分离(或者分析用户), 执行sql下发后端实例的系列信息	0: 关闭/1: 开启
	sqlDumpLogBasePath	审计日志的base路径	默认为sqldump	审计日志的base路径	合法路径
	sqlDumpLogFileName	审计日志文件名	默认为sqldump.log	日志文件名, 生成日志的相对路径: sqldump/sqldump.log	合法文件名
	sqlDumpLogCompressFilePattern	压缩日志文件命名格式	默认为\${date:yyyy-MM}/sqldump-%d{MM-dd}-%i.log.gz	如: sqldump/2022-10/sqldump-10-11-1.log.gz (精度为: 天)	合法文件名格式
	sqlDumpLogOnStartupRotate	重启触发日志翻转	默认为1	每次重启, 是否触发翻转; 1-是, 0-否	1-是, 0-否
	sqlDumpLogSizeBasedRotate	文件大小上限触发日志翻转	默认为50MB	当sqldump.log文件大小达到50MB, 触发日志翻转; 单位可以为:KB、MB、GB	合法文件大小
	sqlDumpLogTimeBasedRotate	间隔天数, 触发日志翻转	默认为1	当设置为1时, 则每天会进行日志反转	正整数
	sqlDumpLogDeleteFileAge	对过期时间段内的压缩文件进行删除	默认为90d	对过期90天内的压缩文件进行删除, 单位可以为:d(天)、h(时)、m(分)、s(秒); (注意, 精度单位需要与sqlDumpLogCompressFilePattern保持一致, 否者可能不生效; 原因: 基于log4j2的实现, 存在这个现象)	合法时间段
				匹配sqlDumpLogCompressFilePath 路径下 且满	

				足 sqlDumpLogDeleteFileAge 的文件进行删除	
tcp相关参数	tcpKeepIdle	启用tcp-keepalive连接检测的连接空闲时间	默认为30s,最大值跟使用的操作系统相关	dblc连接如果没有数据发送,多久后会发送Keepalive探测分组,生效条件需要指定版本的jdk,具体可查看2.38章节内容	正整数
	tcpKeepInterval	启用tcp-keepalive连接检测的时间间隔	默认为10s,最大值跟使用的操作系统相关	dblc连接开启tcp-keepalive探活机制后,前后两次探测之间的时间间隔,生效条件需要指定版本的jdk,具体可查看2.38章节内容	正整数
	tcpKeepCount	启用tcp-keepalive检测启用后的最大重试次数	默认3,最大值跟使用的操作系统相关	开启tcp-keepalive探活机制后,关闭一个非正常连接之前的最大重试次数,生效条件需要指定版本的jdk,具体可查看2.38章节内容	正整数

1.2.3 bootstrap.dynamic.cnf

有些参数是可以通过管理命令热生效的,这部分参数会写到bootstrap.dynamic.cnf中,当dble重启时候,会用内部的值替换bootstrap.cnf中对应的值,这些参数是:

```
enableAlert
enableSlowLog
flushSlowLogPeriod
flushSlowLogSize
sqlSlowTime
enableFlowControl
flowControlLowLevel
flowControlHighLevel
enableGeneralLog
generalLogFile
enableStatistic
associateTablesByEntryByUserTableSize
frontendByBackendByEntryByUserTableSize
tableByUserByEntryTableSize
enableBatchLoadData
maxRowSizeToFile
enableMemoryBufferMonitor
xaIdCheckPeriod
enableSqlDumpLog
```

1.2.4 配置实例

```
#encoding=UTF-8
-agentlib:jdwp=transport=dt_socket,server=y,address=8088,suspend=n
-server
-XX:+AggressiveOpts
-Dfile.encoding=UTF-8
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=1984
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.host=127.0.0.1
-Xmx4G
-Xms1G
-XX:MaxDirectMemorySize=2G
# base config
-DhomePath=.
-DinstanceName=1
# valid for sequenceHandlerType=2 or 3
-DinstanceId=1
```

```
-DinstanceId=1
-DserverId=xxx1
#-DbindIp=0.0.0.0
#-DserverPort=8066
#-DmanagerPort=9066
#-DmaxCon=1024
#-DNIOFrontRW=4
#-DNIOBackendRW=12
#-DfrontWorker=4
#-DmanagerFrontWorker=2
#-DbackendWorker=12
#-DcomplexQueryWorker=8
#-DwriteToBackendWorker=4

-DfakeMySQLVersion=5.7.11

# serverBacklog size,default 2048
-DserverBacklog=2048

#-DusePerformanceMode=0
# if need out HA
-DuseOuterHa=true

# connection
#-Dcharset=utf8mb4
-DmaxPacketSize=167772160
-DtxIsolation=2
#-Dautocommit=1

#parameter for mysql
#-DgroupConcatMaxLen=1024

# option
#-DuseCompression=1
-DusingAIO=0

-DuseThreadUsageStat=1
# query time cost statistics
#-DuseCostTimeStat=0
#-DmaxCostStatSize=100
#-DcostSamplePercent=1

# consistency
# check the consistency of table structure between nodes,default not
-DcheckTableConsistency=0
# check period, the default period is 60000 milliseconds
-DcheckTableConsistencyPeriod=60000

# processor check conn
-DprocessorCheckPeriod=1000
-DsqlExecuteTimeout=3000
-DidleTimeout=1800000

#-DbackSocket unit:bytes
#-DbackSocketSoRcvbuf=4194304
#-DbackSocketSoSndbuf=1048576
#-DbackSocketNoDelay=1

# frontSocket
#-DfrontSocketSoRcvbuf=1048576
#-DfrontSocketSoSndbuf=4194304
#-DfrontSocketNoDelay=1

# query memory used for per session,unit is M
-DotherMemSize=4
-DorderMemSize=4
-DjoinMemSize=4

# off Heap unit:bytes
-DbufferPoolChunkSize=32767
-DbufferPoolPageNumber=512
-DbufferPoolPageSize=2097152
```

```

# transaction log
# 1 enable record the transaction log, 0 disable ,the unit of transactionRotateSize is M
-DrecordTxn=0
#-DtransactionLogBaseDir=/txlogs
#-DtransactionLogBaseName=server-tx
#-DtransactionRotateSize=16
# XA transaction
# use XA transaction ,if the mysql service crash,the unfinished XA commit/rollback will retry for several times , it is the check period
for ,default is 1000 milliseconds
-DxaSessionCheckPeriod=1000
# use XA transaction ,the finished XA log will removed. the default period is 1000 milliseconds
-DxaLogCleanPeriod=1000
# XA Recovery Log path
# -DxaRecoveryLogBaseDir=/xalogs/
# XA Recovery Log name
#-DxaRecoveryLogBaseName=xalog
# XA Retry count, retry times in backend, 0 means always retry until success
#-DxaRetryCount=0

#-DviewPersistenceConfBaseDir=/viewPath
#-viewPersistenceConfBaseName=viewJson

# for join tmp results
#-DmergeQueueSize=1024
#-DorderByQueueSize=1024
#-DjoinQueueSize=1024

# true is use JoinStrategy, default false
#-DuseJoinStrategy=true
#-DjoinStrategyType=-1
-DnestLoopConnSize=4
-DnestLoopRowsSize=2000

# if enable the slow query log
-DenableSlowLog=1
# the slow query log location
#-DslowLogBaseDir=./slowlogs
#-DslowLogBaseName=slow-query
# the max period for flushing the slow query log from memory to disk after last time , unit is second
-DflushSlowLogPeriod=1
# the max records for flushing the slow query log from memory to disk after last time
-DflushSlowLogSize=1000
# the threshold for judging if the query is slow , unit is millisecond
-DsqlSlowTime=100

# used for load data,maxCharsPerColumn means max chars length for per column when load data
#-DmaxCharsPerColumn=65535
# used for load data, because dble need save to disk if loading file contains large size
#-DmaxRowSizeToFile=100000
if enable the batch load data
#-DenableBatchLoadData=1
#enableFlowControl=false
#-DflowControlHighLevel=4194304
#-DflowControlLowLevel=262144

# if enable the general log
#-DenableGeneralLog=1
# general log file path
#-DgeneralLogFile=general/general.log
# maximum value of file, unit is mb
#-DgeneralLogFileSize=16
# the queue size must not be less than 1 and must be a power of 2
#-DgeneralLogQueueSize=4096

# if enable statistic sql
#-DenableStatistic=1
#-DenableStatisticAnalysis=0
#-DassociateTablesByEntryByUserTableSize=1024
#-DfrontendByBackendByEntryByUserTableSize=1024
#-DtableByUserByEntryTableSize=1024
# processing queue size must not be less than 1 and must be a power of 2
#-DstatisticQueueSize=4096
# samplingRate
#-DsamplingRate=100
# size of sql log table

```

```
# size of sql log table
#-DsqlLogTableSize=1024
#-DmaxResultSet=524288

#-DinSubQueryTransformToJoin=false
#For rwSplitUser, Implement stickiness for read and write instances, the default value is 1000ms
#-DrwStickyTime=1000

# if enable frontend connection activity ratio statistics
#-DenableSessionActiveRatioStat=1
# if enable frontend connection and backend connection are associated with threads
#-DenableConnectionAssociateThread=1
#-Ddistrict=
#-DdataCenter=
#-DxaIdCheckPeriod=300

# whether enable the memory buffer monitor
#-DenableMemoryBufferMonitor=0
#-DenableMemoryBufferMonitorRecordPool=1

#-DenableSqlDumpLog=0
#-DsqlDumpLogBasePath=sqldump
#-DsqlDumpLogFileName=sqldump.log
#-DsqlDumpLogCompressFilePattern=${date:yyyy-MM}/sqldump-%d{MM-dd}-%i.log.gz
#-DsqlDumpLogOnStartupRotate=1
#-DsqlDumpLogSizeBasedRotate=50MB
#-DsqlDumpLogTimeBasedRotate=1
#-DsqlDumpLogDeleteFileAge=90d
#-DsqlDumpLogCompressFilePath=*/sqldump-*.log.gz

#-DtcpKeepIdle=30
#-DtcpKeepInterval=10
#-DtcpKeepCount=3
```

1.3 user.xml 配置

1.3.1 整体XML结构

- managerUser (可多值,至少一个)
- shardingUser (可多值)
- rwSplitUser (可多值)
- analysisUser (可多值)
- hybridTAUser (可多值)
- blacklist(可多值)

配置注意事项:

1. 当user.xml文件中不配置shardingUser，dble不再加载sharding.xml配置文件(即dble不具备分表分库)，包括集群情况下出现sharding.xml不一致，均属于已知现象。

1.3.2 managerUser(管理用户配置)

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	用户名	符合mysql用户名规范的字符串	用户唯一标识，用于登录校验
password	密码		用户密码校验
usingDecrypt	是否启用加密	可配置true/false，默认false	<p>启用加密password项配置通过执行脚本encrypt.sh 0:{user}:{password}的结果进行配置</p> <p>举例： encrypt.sh 0:xxx:123456 fP/nl3XPXrSfWjpQzit5IIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWruFvkJPUZpbQ2qX9u Q==</p> <p>配置项： password fP/nl3XPXrSfWjpQzit5IIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWruFvkJPUZpbQ2qX9u Q==</p> <p>user xxx 登录项： -u xxx -p123456</p>
whiteIPs	可登录的ip的白名单	可选项	可以参考本配置文件中的whiteIPs内容
readOnly	是否是只读管理用户	可配置true/false，默认false	只读用户不能进行运维管理操作，只能进行show或者select
maxCon	负载限制，默认不做限制	正整数	用户的连接数限制,会在用户验证登录的时候进行校验，默认0，表示不做限制。特别的，管理用户不受系统级别的maxCon的限制

1.3.3 shardingUser(分库用户配置)

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	用户名	符合mysql用户名规范的字符串	用户唯一标识，用于登录校验
password	密码		用户密码校验
usingDecrypt	是否启用加密	可配置true/false，默认false	启用加密password项配置通过执行脚本encrypt.sh 0:{user}:{password}的结果进行配置 举例： encrypt.sh 0:xxx:123456 fP/nl3XPXrSfWjpQzit5lIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWruFvkJPUZpbQ2qX9u Q== 配置项： password fP/nl3XPXrSfWjpQzit5lIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWruFvkJPUZpbQ2qX9u Q== user xxx 登录项： -u xxx -p123456
whiteIPs	可登录的ip的白名单	可选项	可以参考本配置文件中的whiteIPs内容
readOnly	是否是只读分库用户	可配置true/false，默认false	只读用户不能进行DML操作，只能进行show或者select
tenant	租户名	可选配置。和用户名相当于整个用户列表的联合主键	可以参考本配置文件中的tenant内容
schemas	该用户可以访问的schema列表	可配置多值，用逗号隔开	该用户可以访问的schema列表，schema参见sharding.xml中的schma名称
maxCon	负载限制，默认不做限制	正整数	用户的连接数限制,会在用户验证登录的时候进行校验，默认0，表示不做限制。特别的，当系统级别的maxCon已经到达上限之后，本用户的maxCon会失效，不能新建连接
blacklist	blacklist的名称	可选配置	可以参考本配置文件中的blacklist内容
privileges	子元素，具体table的增删改查权限	可选配置	可以参考本配置文件中的privileges内容

1.3.3.1 user.privileges.schema

user.privileges 下的schema的dml权限，可配置多值

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	schema名称		用以标识对应schema
dml	dml权限	0000	权限判断，每一位分别表示INSERT UPDATE SELECT DELETE四种权限 1- 拥有权限 0-没有权限 例如拥有所有权限为1111
table	子元素	可配置多个	如果没有配置，则table继承schema的权限

1.3.3.2 user.privileges.schema.table

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	表格名称		在权限判断的时候作为key值
dml	dml权限	0000	权限判断，每一位分别表示INSERT UPDATE SELECT DELETE四种权限 1- 拥有权限 0-没有权限 例如拥有所有权限为1111

1.3.4 rwSplitUser(读写用户配置)

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	用户名	符合mysql用户名规范的字符串	用户唯一标识，用于登录校验
password	密码		用户密码校验
usingDecrypt	是否启用加密	可配置true/false，默认false	启用加密password项配置通过执行脚本encrypt.sh 0:{user}:{password}的结果进行配置 举例： encrypt.sh 0:xxx:123456 fP/nl3XPXrSfWjpQzit5IIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWrufVkJPUZpbQ2qX9u Q== 配置项： password fP/nl3XPXrSfWjpQzit5IIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWrufVkJPUZpbQ2qX9u Q== user xxx 登录项： -u xxx -p123456
whiteIPs	可登录的ip的白名单	可选项	可以参考本配置文件中的whiteIPs内容
tenant	租户名	可选配置。和用户名相当于整个用户列表的联合主键	可以参考本配置文件中的tenant内容
dbGroup	该用户对应的数据库组dbGroup	单值配置	对应db.xml中的dbGroup名称
maxCon	负载限制，默认不做限制	正整数	用户的连接数限制,会在用户验证登录的时候进行校验，默认0，表示不做限制。特别的，当系统级别的maxCon已经到达上限之后，本用户的maxCon会失效，不能新建连接
blacklist	blacklist的名称	可选配置	可以参考本配置文件中的blacklist内容

1.3.5 analysisUser(分析用户配置)

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	用户名	符合分析数据库用户名规范的字符串（目前支持clickhouse）	用户唯一标识，用于登录校验
password	密码		用户密码校验
usingDecrypt	是否启用加密	可配置true/false，默认false	<p>启用加密password项配置通过执行脚本encrypt.sh 0:{user}:{password}的结果进行配置</p> <p>举例： encrypt.sh 0:xxx:123456 fP/nI3XPXrSfWjpQzit5IIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWruFvkJPUZpbQ2qX9u Q==</p> <p>配置项： password fP/nI3XPXrSfWjpQzit5IIOrRU1QRXu LTYtATUG0fGW2k5kdXUhKL5zf02 hE6nGjdnSWruFvkJPUZpbQ2qX9u Q==</p> <p>user xxx 登录项： -u xxx -p123456</p>
whiteIPs	可登录的ip的白名单	可选项	可以参考本配置文件中的whiteIPs内容
tenant	租户名	可选配置。和用户名相当于整个用户列表的联合主键	可以参考本配置文件中的tenant内容
dbGroup	该用户对应的数据库组dbGroup	单值配置	对应db.xml中的dbGroup名称；注意：这里的dbGroup可以与hybridTAUser的apNode共用同一个
maxCon	负载限制，默认不做限制	正整数	用户的连接数限制,会在用户验证登录的时候进行校验，默认0，表示不做限制。特别的，当系统级别的maxCon已经到达上限之后，本用户的maxCon会失效，不能新建连接
blacklist	blacklist的名称	可选配置	可以参考本配置文件中的blacklist内容

1.3.6 hybridTAUser(HTAP用户配置)

- 可参考 [HTAP 快速上手指南](#)

配置名称	配置内容	可选项/默认值	详细作用原理或应用
name	用户名	符合mysql用户名规范的字符串	用户唯一标识，用于登录校验
password	密码		用户密码校验
usingDecrypt	是否启用加密	可配置true/false，默认false	启用加密password项配置通过执行脚本encrypt.sh 0:{user}:{password}的结果进行配置 举例： encrypt.sh 0:xxx:123456 fP/nI3XPXrSfWjpQzit5lIOrRU1QRX uLTYtATUG0fGW2k5kdXUhKL5zf0 2hE6nGjdnSWrufVkJPUZpbQ2qX 9uQ== 配置项： password fP/nI3XPXrSfWjpQzit5lIOrRU1QRX uLTYtATUG0fGW2k5kdXUhKL5zf0 2hE6nGjdnSWrufVkJPUZpbQ2qX 9uQ== user xxx 登录项： -u xxx -p123456
whiteIPs	可登录的ip的白名单	可选项	可以参考本配置文件中的whiteIPs内容
readOnly	是否是只读分库用户	可配置true/false，默认false	只读用户不能进行DML操作，只能进行show或者select
tenant	租户名	可选配置。和用户名相当于整个用户列表的联合主键	可以参考本配置文件中的tenant内容
schemas	该用户可以访问的schema列表	可配置多值，用逗号隔开	该用户可以访问的schema列表，schema参见sharding.xml中的schma名称 注意：对应的dbGroup的databaseType必须是clickhouse
maxCon	负载限制，默认不做限制	正整数	用户的连接数限制,会在用户验证登录的时候进行校验，默认0，表示不做限制。特别的，当系统级别的maxCon已经到达上限之后，本用户的maxCon会失效，不能新建连接
blacklist	blacklist的名称	可选配置	可以参考本配置文件中的blacklist内容
privileges	子元素，具体table的增删改查权限	可选配置	可以参考本配置文件中的privileges内容

1.3.7 blacklist(黑名单配置)

配置名称	配置内容	配置范围/可选项	详细作用原理或应用
name	blacklist的名称		用于被用户引用，标记黑名单
property	子元素，可以有多个	详细的黑名单校验规则	如果开启黑名单校验具体的校验规则将有所有property来确定

1.3.7.1 blacklist.property(详细的黑名单配置)

形式为

```
<property name="selectHavingAlwayTrueCheck">true</property>
```

下面表格来描述key和value的含义。

- 解析判断

useAllow

配置名称	描述	默认值	可选项	详细作用原理或应用	分库分表支持程度	读写分离支持程度
multiStatementAllow	是否允许一次执行多条语句	false	true false	sql数 >1, 值为true时, 允许执行多条语句。值为false时, 不允许执行多条语句。由于dblc是个中间件, 在协议解析层面已经处理过多语句了, 所以这个黑名单的配置在Dble下不生效。多语句开关根据应用需要驱动自行设置	不支持	不支持

- sql类型判断

配置名称	描述	默认值	可选项	详细作用原理或应用	命中黑名单sql样例	分库分表支持程度	读写分离支持程度
insertAllow	是否允许执行INSERT语句	true	true false	值为true时, 允许执行INSERT语句。值为false时, 不允许执行INSERT语句	insert into t1 valu1 values(4,5);	支持	支持
deleteAllow	是否允许执行DELETE语句	true	true false	值为true时, 允许执行DELETE语句。值为false时, 不允许执行DELETE语句	delete from t1;	支持	支持
updateAllow	是否允许执行UPDATE语句	true	true false	值为true时, 允许执行UPDATE语句。值为false时, 不允许执行UPDATE语句	update t1 set id = 1 where id =10;	支持	支持
mergeAllow	是否允许执行merge语句	true	true false	值为true时, 允许允许执行merge语句。值为false时, 不允许允许执行merge语句	insert into t1 valu1 values(4,5);	mysql不支持该语法,无意义	mysql不支持该语法,无意义
callAllow	是否允许执行call语句	true	true false	值为true时, 允许执行call语句。值为false时, 不允许执行call语句	call proc_arc(1);	支持, 分库分表的存储过程默认需要带hint, 详见 https://actiontech.github.io/dble-docs-cn/3.SQL_Syntax/3.6_procedure_support.html	支持
truncateAllow	是否允许执行Truncate语句	true	true false	值为true时, 允许执行Truncate语句。值为false时, 不允许执行Truncate语句	truncate table t1;	支持	支持
createTableAllow	是否允许创建表	true	true false	值为true时, 允许创建表。值为false时, 不允许创建表	create table t1(id int, age int);	支持	支持
renameTableAllow	是否允许执行Rename语句	true	true false	值为true时, 允许执行Rename语句。值为false时, 不允许执行Rename语句	rename table t1 to t4;	不支持	支持
	是否允许执行Alter Table语句	true	true false	注意: 类似ALTER TABLE t1 RENAME t6的sql会被替换成renam t1 to t6,属于RenameTableAllow控制 值为	alter table t1 add d	支持	支持

				llow控制 值为true时, 允许执行Alter Table语句。值为false时, 不允许执行Alter Table语句	timestamp;		
dropTableAllow	是否允许执行DropTable语句	true	true false	值为true时, 允许执行DropTable语句。值为false时, 不允许执行DropTable语句	drop table t1;	支持	支持
setAllow	是否允许执行Set语句	true	true false	值为true时, 允许执行Set语句。值为false时, 不允许执行Set语句	set @name = 1;	支持	支持
replaceAllow	是否允许执行Replace语句	true	true false	值为true时, 允许执行Replace语句。值为false时, 不允许执行Replace语句	replace into t1 values (1, 1);	支持	支持
describeAllow	是否允许执行describe语句	true	true false	值为true时, 允许执行describe语句。值为false时, 不允许执行describe语句	describe t1 id;	支持	支持
showAllow	是否允许执行show语句	true	true false	值为true时, 允许执行show语句。值为false时, 不允许执行show语句	show columns from t1;	支持	支持
commitAllow	是否允许执行Commit语句	true	true false	值为true时, 允许执行Commit语句。值为false时, 不允许执行Commit语句	commit;	支持	支持
rollbackAllow	是否允许执行Rollback语句	true	true false	值为true时, 允许执行Rollback语句。值为false时, 不允许执行Rollback语句	rollback;	支持	支持
useAllow	是否允许执行Use语句	true	true false	值为true时, 允许执行Use语句。值为false时, 不允许执行Use语句	use db1;	支持	支持
hintAllow	是否允许执行Hint语句	true	true false	值为true时, 允许执行Hint语句。值为false时, 不允许执行Hint语句	select * from t1/!TEMPORARY *;	支持	支持
lockTableAllow	是否允许执行LockTable语句	true	true false	值为true时, 允许执行LockTable语句。值为false时, 不允许执行LockTable语句	lock table t1 write;	支持	支持
startTransactionAllow	是否允许执行StartTransaction语句	true	true false	值为true时, 允许执行StartTransaction语句。值为false时, 不允许执行StartTransaction语句	start transaction;	支持	支持
blockAllow	是否允许语句块	true	true false	值为true时, 允许执行LockTable语句。值为false时, 不允许执行LockTable语句	begin select * from t1 where id=1end;	支持	支持

LockTable语句							
noneBaseStatementAllow	是否允许非以上基本语句的其他语句, 通过这个选项就能够屏蔽DDL	false	true false	值为true时, 不允许以上基本语句的其他语句。 值为false时, 允许执行以上基本语句的其他语句		支持	支持

- sql组成元素判断

说明: 组成元素用expr代替, 另外未做特殊声明如果判断条件中有多个条件默认为并列都需要满足 除非对以下内容有足够深的了解, 否则不建议使用, 保持默认值即可

1.常规设置

配置名称	描述	默认值	可选项	详细作用原理或应用	命中黑名单sql样例	分库分表支持程度	读写分离支持程度
mustParameterized	含有where条件时, 是否必须参数化	false	true false	<p>如果为True, 则不允许类似 WHERE ID = 1 这种不参数化的 SQL</p> <p>例如</p> <p>1.select * from t1 inner join t3 on t1.id = 1; 没有where条件, 不受规则控制</p> <p>值为true时, 不允许不参数化的 SQL。值为false时, 允许不参数化的SQL</p>	select * from t1 where t1.id =1;	支持	支持
constArithmeticAllow	拦截常量运算的条件	true	true false	<p>值为true时, 允许执行常量运算的条件。值为false时, 不允许执行常量运算的条件</p>	<p>select * from t1 where 1>1;</p> <p>select * from t1 where id = 3-1;</p> <p>select * from t1 where true & false;</p>	支持	支持
limitZeroAllow	是否允许limit 0 这样的语句	false	true false	<p>值为true时, 允许执行limit 0 这样的语句。值为false时, 不允许执行limit 0 这样的语句</p>	select * from t1 limit 0;	支持	支持
selectAllow	是否允许执行SELECT语句	true	true false	<p>值为true时, 允许执行SELECT语句。值为false时, 不允许执行SELECT语句 (与使用方面无关的tips: 这个判断正常应该放在sql类型判断的代码逻辑中, druid把它放在了sql组成元素判断)</p>	select id from t1;	支持	支持
			true	<p>需要满足以下条件</p> <ol style="list-style-type: none"> 1.expr 属于 SQLAllColumn Expr类型 2.expr所在的sql是select类型 3.所在sql的from中的表为普通单表 4.不带有别名(x.*) <p>举例:</p> <p>1.select t.* from t1 t; 不属于 SQLAllColumn</p>			

selectAllColumnAllow	是否允许查询所有列	true	true false	<p>举例:</p> <p>1.select t.* from t1 t; 不属于SQLAllColumnExpr类型, 所以不符合该规则, 不受黑名单控制</p> <p>2.select * from t1, t3; from后面的表是一个结果集, 所以不符合该规则, 不受黑名单控制</p> <p>3.select * from t1 inner join t3; from后面的表是一个结果集, 所以不符合该规则, 不受黑名单控制</p> <p>值为true时, 允许查询所有列。 值为false时, 不允许查询所有列</p>	select * from t1;	支持	支持
commentAllow	是否允许语句中存在注释	false	true false	值为true时, 允许语句中存在注释。值为false时, 不允许语句中存在注释	select * from t1 where id =1 or 1=1 /*dblesql=select 1 from account */;	支持	支持
conditionOpXorAllow	查询条件(where或having)中是否允许有XOR条件	false	true false	<p>特例:</p> <p>select * from t1 inner join t3 on t1.id =(1 xor 1); 没有where 或者having, 不受规则控制</p> <p>值为true时, 允许查询条件(where或having)中有XOR条件。值为false时, 不允许查询条件(where或having)中有XOR条件</p>	select * from t1 where id = (1 xor 1); select * from t1 having id = (1 xor 1);	支持	支持
conditionOpBitwiseAllow	查询条件中是否允许有"&"、"~"、" "、"^"运算符	true	true false	值为true时, 允许查询条件中是否允许有"&"、"~"、" "、"^"运算符。值为false时, 不允许查询条件中是否允许有"&"、"~"、" "、"^"运算符	<p>select * from t1 where id = (1 & 1);</p> <p>select * from t1 where id = (1 & select id from t1 limit1);</p> <p>select * from t1 where id = (1 ^ 1);</p> <p>select * from t1 where id = (1 ~ 1);</p> <p>select * from t1 where id = (1 1);</p>	支持	支持
	查询条件中是否允许连续两个常量运算表	false	true false	<p>必须是and两侧表达式为恒真或者恒假的表达式</p> <p>举例:</p> <p>select *from t1 where 3=1 or 3=3; 不是在and两侧, 不受规则控制</p>	<p>select *from t1 where 3=1 and 3=3;</p> <p>select *from t1 where 3=1 or (1=1 and 3=3);</p> <p>select *from t1 where 3=1 and (1=1 and 3=3);</p> <p>select * from t3 where 1=1 and k like '%'; (需要开启conditionLikeTrueAllow为false)</p>	支持	支持

conditionDoubleConstantAllow	达式			值为true时，允许连续两个常量运算表达式。值为false时，不允许连续两个常量运算表达式	<pre>select * from t3 where 1=1 and 1= (select count(*) from t1 limit1); select * from t3 where 2=1 and true = true; select * from t3 where id =1 and true = true or id =1 or(1=1 and id =2);</pre>		
deleteWhereNoneCheck	检查DELETE语句是否无where条件	false	true false	<p>需满足以下条件</p> <ol style="list-style-type: none"> 1.sql没有where条件 2.没有using 3.from的表为普通单表 <p>举例：</p> <pre>delete from t1 using t1 inner join t2; 使用 using，导致条件 不满足，所以 不符合该规则， 不受黑名单控制</pre> <pre>delete t1,t2 from t1 left join t2 on t1.id=t2.id; from 的表是一个结果 集，所以不符合 该规则，不受黑 名单控制</pre> <p>值为true时，不允许语句中无where条件。值为false时，允许语句中无where条件</p>	<pre>delete from t1; delete t1 from t1 left join t2 on t1.id=t2.id;</pre>	支持	支持
updateWhereNoneCheck	检查UPDATE语句是否无where条件	false	true false	<p>需满足以下条件</p> <ol style="list-style-type: none"> 1.没有limit <p>举例：</p> <pre>1.update t1 set idd =1 limit 1;有 limit条件，所以 不符合该规则， 不受黑名单控制</pre> <p>值为true时，不允许语句无where条件。值为false时，允许语句中无where条件</p>	<pre>update t1 set idd =1 ;</pre>	支持	支持
conditionAlwaysFalseAllow	检查查询条件(WHERE/HAVING子句)中是否包含AND永假条件	false	true false	<p>值为true时，允许语句中查询条件(WHERE/HAVING子句)中包含AND永假条件。值为false时，不允许语句中查询条件(WHERE/HAVING子句)中包含AND永假条件</p>	<pre>select * from t1 where id = 567 and 2 = 1; select * from t1 having id =1 and 2=1;</pre>	支持	支持
				实际条件如果稍微复杂，可能就判断不出来了，例如：			

				(WHERE/HAVING子句)中包含AND永真条件。值为false时,不允许语句中查询条件 (WHERE/HAVING子句)中包含AND永真条件	select * from t1 having id =1 and 1=1;		
conditionLikeTrueAllow	检查查询条件(WHERE/HAVING子句)中是否包含LIKE永真条件	true	true false	单独使用并不会拦截。 conditionLikeTrueAllow为false的时候才会把like '%当永真条件配合其他参数(比如conditionAlwaysTrueAllow)使用 值为true时,语句中查询条件(WHERE/HAVING子句)中包含LIKE永真条件,不判断为永真条件。值为false时,语句中查询条件(WHERE/HAVING子句)中包含LIKE永真条件,判断为永真条件	select * from t3 where id = 5 and k like '%'; select * from t1 having id =1 and k like '%';	支持	支持
selectLimit	配置最大返回行数	-1	-1 不设置	配置最大返回行数,如果select语句没有指定最大返回行数,会自动修改select添加返回限制		不支持, dble中不可用	不支持, dble中不可用

2.select into 参数

配置名称	描述	默认值	可选项	详细作用原理或应用	命中黑名单sql样例	分库分表支持程度	读写分离支持程度
selectIntoAllow	SELECT查询中是否允许INTO语句	true	true false	值为true时,允许select into语句。值为false时,不允许select into语句	select * into @myvar from t1; select * from t1 into @myvar for update; select id, data into @x, @y from test.t1 limit 1;	不支持	支持
selectIntoOutfileAllow	当outfile子句不是最外层的sql时, SELECT ... INTO OUTFILE 是否允许	false	true false	满足outfile子句不是最外层的sql时, 值为true时, 允许SELECT ... INTO OUTFILE语句。值为false时, 不允许SELECT ... INTO OUTFILE语句	select * from t1 where id in(select id into outfile '/exportdata/customers.txt' fields terminated by ';' optionally enclosed by "" lines terminated by '\n' from t1); 这个sql并不符合sql语法, 是基于druid逻辑构造出来的sql, 所以无实际意义	不支持, 无意义	不支持, 无意义

3.AlwaysTrue条件 (规则复杂, 理论上都可以使用, 但规则有点奇葩, 一般来说, 无论开关都无影响, 建议保持默认值)

配置名称	描述	默认值	可选项	详细作用原理或应用	命中黑名单sql样例	分库分表支持程度	读写分离支持程度
selectWhereAlwaysTrueCheck	检查SELECT语句的WHERE子句是否为一个AlwaysTrue条件	true	true false	<p>AlwaysTrue条件规则复杂：</p> <p>1.where条件存在恒真</p> <p>2.sql以注释结尾</p> <p>3.条件部分不是简单SQL（单个条件、含有简单数值对等或大小比较、直接是真假值的表达式等）</p> <p>比如update t1 set idd =1 where id = id 是恒真，但不命中条件2和3</p> <p>值为true时，不允许满足以上条件的sql执行。 值为false时，允许满足以上条件的sql执行</p>	<pre>select id from t1 where id =1 union select 1 /*!db!sql=select 1 from account */;</pre>	支持	支持
selectHavingAlwaysTrueCheck	检查SELECT语句的HAVING子句是否为一个AlwaysTrue条件	true	true false	<p>AlwaysTrue条件含义详见selectWhereAlwaysTrueCheck</p> <p>值为true时，不允许满足以上条件的sql执行。 值为false时，允许满足以上条件的sql执行</p>	<pre>select * from t1 having id = 1 or 1=1 /*!db!sql=select 1 from account */;</pre>	支持	支持
deleteWhereAlwaysTrueCheck	检查DELETE语句的WHERE子句是否为一个AlwaysTrue条件	true	true false	<p>AlwaysTrue条件含义详见selectWhereAlwaysTrueCheck</p> <p>值为true时，不允许满足以上条件的sql执行。 值为false时，允许满足以上条件的sql执行</p>	<pre>delete from t1 where id = 1 or 1=1 /*!db!sql=select 1 from account */;</pre>	支持	支持
updateWhereAlwaysTrueCheck	检查UPDATE语句的WHERE子句是否为一个AlwaysTrue条件	true	true false	<p>AlwaysTrue条件含义详见selectWhereAlwaysTrueCheck</p> <p>值为true时，不允许满足以上条件的sql执行。 值为false时，允许满足以上条件的sql执行</p>	<pre>update t1 set idd =1 where id = id or 1=1 /*!db!sql=select 1 from account */;</pre>	支持	支持

4.复杂规则设置（规则复杂，理论上都可以使用，但规则有点奇葩，一般来说，无论开关都无影响，建议保持默认值）

配置名称	描述	默认值	可选项	详细作用原理或应用	命中黑名单sql样例	分库分表支持程度	读写分离支持程度
caseConditionConstAllow	是否允许复杂查询中外部是一个常量	false	true false	<p>具体条件如下</p> <ol style="list-style-type: none"> 子查询是简单case类型select语句 子查询外部是常量 <p>值为true时，允许子查询外部对应的是常量。值为false时，子查询外部对应的是常量那么就在SQL检查的时候抛出异常</p>	<pre>select id from t1 where id =1 union select 1 /*!db!sql=select 1 from account */;</pre>	支持	支持
selectUnionCheck	是否进行union check	true	true false	<p>检测SELECT UNION，具体条件如下</p> <ol style="list-style-type: none"> left sql需包含from条件 left sql有where条件 right sql 没有from条件 操作符为UNION 或者 UNION ALL 或者 UNION DISTINCT sql结尾有注释 <p>值为true时，不允许UNION语句。值为false时，允许UNION语句</p>	<pre>select id from t1 where id =1 union select 1 /*!db!sql=select 1 from account */;</pre>	支持	支持

5.禁用对象检测配置 functionCheck

配置名称	描述	默认值	可选项	详细作用原理或应用	备注
tableCheck	检测是否使用了禁用的表	true	true false	这个需要配合druid的配置模式使用，在dble此功能无法被使用	druid使用，dble不具有使用意义
functionCheck	检测是否使用了禁用的函数	true	true false	这个需要配合druid的配置模式使用，在dble此功能无法被使用	druid使用，dble不具有使用意义
objectCheck	检测是否使用了“禁用对象”	true	true false	这个需要配合druid的配置模式使用，在dble此功能无法被使用	druid使用，dble不具有使用意义
variantCheck	检测是否使用了“禁用的变量”	true	true false	这个需要配合druid的配置模式使用，在dble此功能无法被使用	druid使用，dble不具有使用意义
readOnlyTables	指定的表只读，不能够在SELECT INTO、DELETE、UPDATE、INSERT、MERGE中作为“被修改表”出现	空	需要指定表	指定的表只读后，在SELECT INTO、DELETE、UPDATE、INSERT、MERGE语句中出现会抛出异常返回错误信息	druid使用，dble不具有使用意义，不能配置该参数
schemaCheck	检测是否使用了禁用的Schema	true	true false	这个需要配合druid的配置模式使用，在dble此功能无法被使用	druid使用，dble不具有使用意义，不能配置该参数

6.其他规则设置

配置名称	描述	默认值	可选项	详细作用原理或应用	备注
selectMinusCheck	检测SELECT MINUS	true	true false	值为true时，允许SELECT MINUS语句。值为false时，不允许SELECT MINUS语句	mysql不支持该语法，不具备使用意义
selectExceptCheck	检测SELECT EXCEPT	true	true false	值为true时，允许except语句。值为false时，不允许except语句	mysql不支持该语法，不具备使用意义
selectIntersectCheck	检测SELECT INTERSECT	true	true false	值为true是，不允许INTERSECT语句，值为false时，允许INTERSECT语句	mysql不支持该语法，不具备使用意义
strictSyntaxCheck	是否进行严格的语法检测	true	true false	Druid SQL Parser在某些场景不能覆盖所有的SQL语法，出现解析SQL出错。属于调试级别的参数，在正常的使用中不建议更改	druid开发者功能，不具备使用意义，保持默认值即可
minusAllow	是否允许SELECT * FROM A MINUS SELECT * FROM B这样的语句	true	true false	值为true时，允许MINUS语句。值为false时，不允许MINUS语句	mysql不支持该语法，不具备使用意义
intersectAllow	是否允许SELECT * FROM A INTERSECT SELECT * FROM B这样的语句	true	true false	值为true时，允许intersect语句。值为false时，不允许intersect语句	mysql不支持该语法，不具备使用意义
completeInsertValuesCheck	在dble依赖的1.0.31、1.2.6版本中没有效果	false	true false	druid内部函数调用值，不建议修改	druid使用，dble不具有使用意义
doPrivilegedAllow	druid内部权限控制使用	false	true false	druid内部函数调用flag,不建议修改	druid使用，dble不具有使用意义
wrapAllow	是否允许调用Connection/Statement/ResultSet的isWrapFor和unwrap方法	true	true false	druid内部函数调用flag	druid连接池功能，dble不具有使用意义
metadataAllow	是否允许调用Connection.getMetadata方法	true	true false	druid内部函数调用flag	druid连接池功能，dble不具有使用意义

1.3.8 tenant（租户配置）

dble支持两种方式的设置

1. 用户:租户 这种方式以:分隔开用户和租户，一起作为登录的用户 如：

```
mysql -u用户:租户 -p -h
```

```
DriverManager.getConnection("jdbc:mysql://127.0.0.1:8066", "root2:tenant1", "123456");
```

2. JDBC-connectionAttributes 在 connectionAttributes 中添加tenant指定租户名称 如：

```
DriverManager.getConnection("jdbc:mysql://127.0.0.1:8066?connectionAttributes=tenant:tenant1", "root2", "123456");
```

以上两种方式中，若同时设置了1/2两种方式，则采用方式1的配置

1.3.9 whitIPs（IP白名单）

默认不限制，值为IP，多个用逗号隔开

格式：

支持用户输入多ip, 如192.168.1.2,192.168.2.22
 支持用户输入IP段, 如192.168.1.10-192.168.1.100
 支持用户输入通配符, 如192.168.1.%
 支持用户输入IP/CIDR格式, 如192.168.1.1/20

以上格式同样适合IPV4/IPV6

注:

- 管理员用户一旦配置该项, 默认允许本机 (127.0.0.1、0:0:0:0:0:0:1) 登陆
- IPV6格式中不支持IPv4映射

1.3.10 完整例子

```
<?xml version="1.0" encoding="UTF-8"?>
<dbtle:user xmlns:dbtle="http://dbtle.cloud/">
  <managerUser name="man1" password="654321" whiteIPs="127.0.0.1,0:0:0:0:0:0:1" readOnly="false"/>
  <managerUser name="user" usingDecrypt="true" readOnly="true" password="AqEkFEuIFAX6g2TJQnp4cJ2r7Yc0Z4/KBsZqKhT8qSz18Aj91e8lx049BKQE1C6
0FfW4c38pCYa8QGFTub7pnw==" />

  <shardingUser name="root" password="123456" schemas="testdb" readOnly="false" blacklist="blacklist1" maxCon="20"/>
  <shardingUser name="root2" password="123456" schemas="testdb,testdb2" maxCon="20" tenant="tenant1">
    <privileges check="true">
      <schema name="testdb" dml="0110">
        <table name="tb01" dml="0000"/>
        <table name="tb02" dml="1111"/>
      </schema>
    </privileges>
  </shardingUser>
  <!--rwSplitUser not work for now-->
  <rwSplitUser name="rwsu1" password="123456" dbGroup="dbGroup1" blacklist="blacklist1"
    maxCon="20"/>

  <analysisUser name="analysisUser" password="123456" dbGroup="dbGroup3" maxCon="20"/>

  <hybridTAUser name="hysu1" password="111111" schemas="testdb3" maxCon="20"/>
  <hybridTAUser name="hysu2" password="111111" schemas="testdb3" maxCon="20" blacklist="blacklist1" tenant="tenant2">
    <privileges check="true">
      <schema name="testdb3" dml="0110">
        <table name="tb_global1" dml="0000"/>
        <table name="tb_global2" dml="1111"/>
      </schema>
    </privileges>
  </hybridTAUser>

  <blacklist name="blacklist1">
    <property name="selectAllow">true</property>
  </blacklist>
</dbtle:user>
```

1.4 db.xml

db.xml包含具体的数据库组和实例配置,可以配置多组, 每组可以配置多个实例

1.4.1 dbGroup配置

- dbGroup

配置名称	配置内容&示例	可选项/默认值	详细描述
name	节点名称	必需项, 无默认值	dbGroup的唯一标识, 不允许重复
rwSplitMode	读操作的负载均衡模式	必需项, 无默认值, 候选值0/1/2/3	<p>在进行读负载均衡的时候会根据这个配置进行</p> <p>0: 不做均衡, 直接分发到主实例, 从实例将被忽略, 不会尝试建立连接池, 但会有心跳连接</p> <p>1: 读操作在所有从实例中均衡, 当所有从实例都不可用时, 下发语句会报错。</p> <p>2: 读操作在所有实例中均衡。</p> <p>3: 读操作在所有从实例中均衡, 当所有从实例都不可用时, 将语句发往主实例。</p> <p>具体拓扑结构见负载均衡相关章节</p>
delayThreshold	指定主从延迟阈值, 单位毫秒	默认-1, 表示无延迟	<p>1:在进行读取负载均衡的时候会根据最近一次的心跳状态以及读库和主库的延迟进行判断, 如果主从复制不工作或者复制延迟超过delayThreshold配置, 则认为此节点不适合进行读取, 依赖于心跳为show slave status</p> <p>2:不依赖于心跳语句, 通过和delayPeriodMillis参数, delayDatabase参数一起使用, 实现延迟检测, 具体可参考delay_detection章节</p> <p>3:此配置会影响到进行读负载均衡的时候延迟检测的开启, 如果delayThreshold=-1那么读负载均衡选取的时候不会进行延迟检测</p> <p>4:如果配置了delayPeriodMillis参数和delayDatabase参数, 那么1中的延迟检测失效, 2中的延迟检测生效</p>
delayPeriodMillis	指定主从延迟检测周期, 单位毫秒	默认-1, 表示不启用	延迟检测周期, 需要和delayThreshold和delayDatabase一起使用生效
delayDatabase	指定延迟检测时mysql中的database	默认null,表示不配置	延迟检测需要向mysql写入sql, 所以需要向mysql的database, 来确定往哪里写入, 需要和delayThreshold和delayPeriodMillis一起使用生效
disableHA	是否禁用该组的高可用切换	可配置true/false, 默认false	禁用该组的高可用切换, 可能在某些场景下需要
heartbeat	子元素,心跳语句	必选项	<p>该配置会在服务启动时设置的心跳任务里面被使用到, 用于进行mysql实例状态的判断.</p> <p>该配置有以下几种建议值:</p> <ol style="list-style-type: none"> 普通心跳只是用于探活, 建议使用select 1 使用select @@read_only 探测节点可用性以及可写性 使用show slave status,可以探活, 检查复制是否正常, 以及延迟检测。如果Seconds_Behind_Master返回的状态有延迟, 那么会被记入mysql实例的主从延迟中, 影响读请求的路由分发, 延迟超过指定限制读写分离会变为只读主库。
dbInstance	子元素, 表示组下的实例, 可配置多个	空	具体的物理节点配置

- heartbeat

配置名称	配置内容&示例	可选项/默认值	详细描述
timeout	heartbeat子元素,心跳超时阈值,单位: 秒	默认0	心跳超时阈值。前置知识: dble会按照 heartbeatPeriodMillis 的间隔向 dbInstance 发送心跳 心跳发起时候会查上次心跳是否为不正常的心跳,如果上次心跳尚未返回,并且距离最近的正常心跳的时间大于timeout,则标记该结点不可达。 例如:心跳周期 (heartbeatPeriodMillis)2秒,第一次心跳正常,2s后的第二次心跳未返回,4s后第三次心跳发起时候发现上次不正常,不会真的再次下发,而是会根据4s和timeout的大小来确定该节点是否真正超时(该节点使用时候才会真正用到) 如果未超时,则什么也不做,继续下一个周期。如果超时了,则尝试杀掉超时的连接,无论是否杀成功,都会在下一个周期换一个连接继续做心跳,极端情况下会消耗很多连接 2. 心跳连续返回失败后, dble使用该结点时,会检查距离第一次失败的时间差,如果大于timeout,则报该结点不可达。
errorRetryCount	heartbeat子元素,心跳失败后的尝试次数	默认1, 0表示不重试	心跳失败/心跳连接被关闭后,开始重试errorRetryCount次。 1. 心跳失败重试期间,连接池状态为error状态,一旦重试成功,则标记回OK。(目的:防止网络抖动或者连接异常断开场景) 2. 心跳连接被关闭重试期间,连接池状态为ok状态,只有都重试失败后才会置为error状态 3. 重试期间超时,按照超时逻辑处理。
keepAlive	heartbeat子元素,心跳发送后的等待响应时间	默认60秒	dble会按照heartbeatPeriodMillis的间隔向dbInstance发送心跳,发起心跳检查的时候会检查心跳连接是否可用,如果当前检查心跳连接的时间大于(上次心跳返回的正常心跳的时间+heartbeatPeriodMillis+keepAlive)的值,则杀掉该连接,在下一个周期换一个连接继续做心跳。 例如:心跳周期 (heartbeatPeriodMillis)10秒,第一次心跳正常,记录心跳返回的时间为13: 10: 00,接下来的心跳一直没有正常返回,当第九次心跳检测时,当前时间为13: 11: 20,大于 (13: 10: 00 + 10s + 60s),则认为该连接不可用,则杀掉该连接,在下一个周期换一个连接继续做心跳。

- dbInstance

配置名称	配置内容&示例	可选项/默认值	详细描述
name	写节点名称	空	节点名称作为标识
id	id	默认值为name对应的值	id标识
url	写节点地址ip:port	空	被分成IP和PORT用于连接数据库
user	写节点用户	空	用于连接数据库
password	写节点用户密码	空	用于连接数据库
usingDecrypt	是否启用加密password	候选值false/true,默认值false	如果设置为true, password属性值应该为用工具encrypt.sh加密串 1: {name}:{user}:{password} 得到的串
minCon	空闲时, 保有的最小后端连接数	必需项	后端连接池中空闲时维持的最小连接数量, 即常驻连接数。实际工作中, minCon不得小于当前dbGroup实际用到的shardingNode的个数(别名 numOfShardingNodes), 如果启动时发现minCon小于 numOfShardingNodes 会被调整为 numOfShardingNodes。
maxCon	最大后端连接数	必需项	后端连接池最大允许的连接数量。实际工作中, maxCon不得小于当前dbGroup实际用到的shardingNode的个数(别名 numOfShardingNodes), 也不得小于 minCon, 如果启动时发现 maxCon 小于 这两个值 会被调整为这两个值的最大值(即 numOfShardingNodes 和 minCon 当中的最大值)。如果连接耗尽,将无法下发请求并返回报错。
readWeight	节点权重(负载均衡时候使用)	默认不配置表示所有节点等量负载	负载均衡过程中会查看所有节点的权重是否相等, 如果不相等, 那么就会根据权重来配置压力。该值需是大于等于 0 的整数。如果配为0表示该节点不参与读。需注意, 总权重(所有节点权重之和)必须大于 0。
primary	主实例需要配置成true	false	用来标识主节点
disabled	标记改实例不可用	false	高可用切换时候可能会用到
databaseType	数据库实例类型	默认不配置,表示后端数据库为mysql	后端数据库实例为mysql时, 可填写mysql, 后端数据库实例为clickhouse时, 可填写clickhouse,其余值不支持。 + 值为mysql时, 仅支持shardingUser和rwSplitUser使用, + 值为clickhouse时, 支持analysisUser用户以及涉及apNode的hybridTAUser用户使用, 同个dbGroup下的dbInstance需要保证databaseType的值相同 + 知识补充: clickHouse默认大小写敏感且不可更改; 因此当db.xml中有效的clickHouse配置时, 意味着db.xml中所有mysql实例的lower_case_table_names值必须均为0, 否则启动失败
property	连接池属性	空	具体的后端连接池属性
dbDistrict	后端mysql实例所在区域	空	读写分离本地读场景下读流量转发依据, 该参数会和bootstrap.cnf中的district参数匹配
dbDataCenter	后端mysql实例所在数据中心	空	读写分离本地读场景下读流量转发依据, 该参数会和bootstrap.cnf中的dataCenter参数匹配

- property

[后端连接池属性](#)

举例如下:


```

<?xml version="1.0"?>
<db!e:db xmlns:db!e="http://db!e.cloud/">

  <dbGroup name="dbGroup1" rwSplitMode="1" delayThreshold="10000">
    <heartbeat errorRetryCount="1" timeout="10" keepAlive="60">show slave status</heartbeat>
    <dbInstance name="instanceM1" url="ip4:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="true">
      <property name="testOnCreate">false</property>
      <property name="testOnBorrow">false</property>
      <property name="testOnReturn">false</property>
      <property name="testWhileIdle">true</property>
      <property name="connectionTimeout">30000</property>
      <property name="connectionHeartbeatTimeout">20</property>
      <property name="timeBetweenEvictionRunsMillis">30000</property>
      <property name="idleTimeout">600000</property>
      <property name="heartbeatPeriodMillis">10000</property>
      <property name="evictorShutdownTimeoutMillis">10000</property>
    </dbInstance>

    <!-- can have multi read instances -->
    <dbInstance name="instanceS1" url="ip5:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="false">
      <property name="heartbeatPeriodMillis">60000</property>
    </dbInstance>
  </dbGroup>

  <dbGroup name="dbGroup2" rwSplitMode="1" delayThreshold="1000" delayPeriodMillis="2000" delayDatabase="test">
    <heartbeat errorRetryCount="1" timeout="10" keepAlive="60">show slave status</heartbeat>
    <dbInstance name="instanceM2" url="ip5:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="true">
    </dbInstance>

    <!-- can have multi read instances -->
    <dbInstance name="instanceS2" url="ip6:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="false">
      <property name="heartbeatPeriodMillis">60000</property>
    </dbInstance>
  </dbGroup>
</db!e:db>

```

1.4.2 MySQL 用户权限声明

为了让 dble 能正常工作，需根据需要，给后端 MySQL 用户开启以下权限。

权限项目	作用
SELECT	数据查询权限
INSERT	新增数据权限
UPDATE	更新数据权限
DELETE	删除数据权限
FILE	load data等数据导出导入的权限
CREATE	建库(管理端)，建表，建索引权限
DROP	删除表的权限
ALTER	变更表结构的权限
LOCK TABLES	lock tables的权限
ALTER ROUTINE	(hint)变更/删除存储过程的权限
CREATE ROUTINE	(hint)创建存储过程的权限
EXECUTE	(hint)执行存储过程的权限
INDEX	建立/删除索引权限
SUPER	用于KILL功能
SHOW DATABASES	部分GUI工具会使用INFORMATION_SCHEMA SCHEMATA表
PROCESS	用于管理端show processlist 功能
REPLICATION CLIENT	可能场景： 1.用于配置了主从关系的数据结点,使用了读写分离 2.使用show slave status作为心跳 3.需要使用show @@binlog_status功能
REFERENCES	外键约束(纯语法支持，无使用意义)
XA_RECOVER_ADMIN	当后端mysql版本大于8.0时，后端mysql用户会需要XA_RECOVER_ADMIN 权限

1.5 sharding.xml 配置

1.5.1 整体XML结构

- schema (虚拟schema, 可配置多个)
- shardingNode (虚拟分片, 可配置多个)
- apNode (虚拟ap节点分片, 可配置多个)
- function (拆分算法, 可配置多个)

配置注意事项:

1. 当user.xml文件中不含shardingUser或者hybridTAUser任意一个, dble不再加载sharding.xml配置文件(即dble不具备分表分库), 包括集群情况下出现sharding.xml不一致, 均属于已知现象。

1.5.2 schema配置

- schema

配置名称	配置内容	可选项/默认值	详细描述
name	schema名称		schema的唯一标识, 不允许重复
shardingNode	涉及的数据点	缺省无, 可配置多个	未配置时, 只加载xml中的table子节点 有配置时, 1、table子节点的配置会覆盖schema中的配置; 2、配置单个, 对应物理schema下存在且不在配置内的table被视为default single table 3、配置多个(与function搭配), 所有对应物理schema下均存在且不在配置内的table被视为default sharding table; 不推荐生产环境使用, 见1.13 Schema下默认拆分表
function	默认sharding table使用的拆分规则	缺省无	引用function节中的拆分规则名称那个(在shardingNode配置多个时, 此function生效); 不推荐生产环境使用, 见1.13 Schema下默认拆分表
apNode	默认OLAP下发的单节点(对应clickHouse分片节点)	缺省无	引用apNode节点, 作为符合OLAP路由条件时所下发的节点; 注意: 0、只允许配置一个apNode 1、shardingUser对应的schema中不能配置apNode 2、hybridTAUser对应的schema中必须配置apNode
sqlMaxLimit	最大返回结果集限制	默认-1	当且仅当查询的SQL符合下列条件时, 产生效果 1 整个查询属于单表查询, 包括简单查询以及(order/group/聚合函数) 2 表格对应的shema需要有对应的sqlMaxLimit配置
logicalCreateADrop	是否允许逻辑创建和删除	默认值: true	为true的话, 可以执行创建和删除的语句, 但实际上不会创建或删除schema; 反之, 执行创建和删除语句会报错
shardingTable	子属性, 拆分表, 详见shardingTable配置选项	可配置多个	每个表格的详细配置信息
globalTable	子属性, 全局表, 详见globalTable配置选项	可配置多个	每个表格的详细配置信息
singleTable	子属性, 单节点表, 详见singleTable配置选项	可配置多个	每个表格的详细配置信息

- shardingTable

配置名称	配置内容	可选项/默认值	详细描述
name	表格名称	必须项	表名，可以配置多个使用','分割
shardingNode	表格涉及的数据节点	必须项	可使用通配符： xxx\$n0-n1 此种格式指定xxxn0, ..., xxxnm, ..., xxxn1作为该表的数据节点。 dn\$1-6 等价于 dn1,dn2, dn3,dn4,dn5,dn6
shardingColumn	拆分列名	必须项	标记表格的拆分列名
function	使用的拆分规则	必须项	引用function节中的拆分规则名称那个
incrementColumn	表格自增列	可选项，默认无	指定表格自增列，指定自增列的表格会使用全局序列
sqlMaxLimit	最大返回结果集限制	可选项，继承父级schema的sqlMaxLimit	与schemas 中的对应配置效果相同，但是覆盖schema中配置
sqlRequiredSharding	是否要求sql中包含拆分列条件	默认false	如果是true, sql中未包含拆分列条件，返回报错
specifyCharset	dble是否使用ISO-8859-1下发语句	默认false	如果是true,dble使用ISO-8859-1编码下发语句
childTable	关联子表信息，详见childTable选项	可配置多个	路由是通过父子关系进行ER关联

- childTable

配置名称	配置内容	可选项/默认值	详细描述
name	表格名称	必需项	表名，可以配置多个使用','分割
joinColumn	指定同父表进行join操作时的join键	必需项	子表和父表关联的字段
parentColumn	指定进行join操作时父表中的join键	必需项	如果父表为非子表，在父表中该字段必须与其拆分规则/拆分键有对等关系。
incrementColumn	表格自增列	默认空	显式指定表格自增列
sqlMaxLimit	最大返回结果集限制	可选项，继承父级schema的sqlMaxLimit	与schemas 中的对应配置效果相同，但是覆盖schema中配置
specifyCharset	dble是否使用ISO-8859-1下发语句	默认false	如果是true,dble使用ISO-8859-1编码下发语句
childTable	关联子表信息，详见childTable选项	可配置多个	路由是通过父子关系进行ER关联

- globalTable

配置名称	配置内容	可选项/默认值	详细描述
name	表格名称	必须项	表名，可以配置多个使用','分割
shardingNode	表格涉及的数据节点	必须项	可使用通配符： xxx\$n0-n1 此种格式指定xxxn0, ..., xxxnm, ..., xxxn1作为该表的数据节点。 dn\$1-6 等价于 dn1,dn2, dn3,dn4,dn5,dn6
sqlMaxLimit	最大返回结果集限制	可选项，继承父级schema的sqlMaxLimit	与schemas 中的对应配置效果相同，但是覆盖schema中配置
specifyCharset	dble是否使用ISO-8859-1下发语句	默认false	如果是true,dble使用ISO-8859-1编码下发语句
checkClass	全局表检查类	可选项	全局表检查自定义类名或者是缩写dble自带CHECKSUM和COUNT两种默认实现
cron	全局表一致性检查周期	0 0 0 * * ?	quartz定时任务时间设置 详见： http://www.quartz-scheduler.org/api/2.4.0-SNAPSHOT/org/quartz/CronScheduleBuilder.html

- singleTable

配置名称	配置内容	可选项/默认值	详细描述
name	表格名称	必须项	表名，可以配置多个使用','分割
shardingNode	表格涉及的数据节点	必需项	唯一的shardingNode节点，配置多个会报错
sqlMaxLimit	最大返回结果集限制	可选项，继承父级schema的sqlMaxLimit	与schemas 中的对应配置效果相同，但是覆盖schema中配置
specifyCharset	dble是否使用ISO-8859-1下发语句	默认false	如果是true,dble使用ISO-8859-1编码下发语句

1.5.3 shardingNode配置

- shardingNode

配置名称	配置内容&示例	详细描述
name	数据节点名称, 唯一, 例如"dn,dn\$0-5"	作为数据节点的标识以及键, 节点个数的计算方法为: 从值出发, 以',' (逗号) 分隔字符串, 如果其中有连续几项拥有相同的字符串前缀X(不能为空)并且后续其他几位为连续的数字时 (比如0到5), 可以以"X\$0-5"来省略表示, 个数为: 以逗号分隔的字符串个数加上包含\$的连续个数。name的个数必须等于database与dbGroup的个数之积。 注意: shadingNode之间以及与apNode之间, name不能重复
database	shardingNode对应的存在于mysql物理实例中的schema, 可以配置单个或多个使用, 例如"db,db\$0-5"	所使用的详细数据库节点, 节点个数的计算方法为: 从值出发, 以',' (逗号) 分隔字符串, 如果其中有连续几项拥有相同的字符串前缀X(不能为空)并且后续其他几位为连续的数字时 (比如0到5), 可以以"X\$0-5"来省略表示, 个数为: 以逗号分隔的字符串个数加上包含\$的连续个数。
dbGroup	shardingNode对应的数据库组, 参考db.xml中的dbGroup名称, 可以配置单个或多个使用, 例如"dh,dh\$0-5"	用于关联对应的Host节点, 节点个数的计算方法为: 从值出发, 以',' (逗号) 分隔字符串, 如果其中有连续几项拥有相同的字符串前缀X(不能为空)并且后续其他几位为连续的数字时 (比如0到5), 可以以"X\$0-5"来省略表示, 个数为: 以逗号分隔的字符串个数加上包含\$的连续个数。

例如:

```
<shardingNode name="dn1" dbGroup="localhost1" database="db1" />
```

name, dbGroup, database均可用如下格式在单个配置中配置多个节点: xxx\$n0-n1, xxx, 这种格式的意义为: xxxn0, ..., xxxnm, ..., xxxn1, xxx, 其中

```
n0 < nm < n1。
```

例如: 配置

```
<shardingNode name="dn1$0-19" dbGroup="localhost1$0-9" database="db1$0-1" />
```

等同于:

```
<shardingNode name="dn10" dbGroup="localhost10" database="db10" />
<shardingNode name="dn11" dbGroup="localhost10" database="db11" />
<shardingNode name="dn12" dbGroup="localhost11" database="db10" />
<shardingNode name="dn13" dbGroup="localhost11" database="db11" />
...
<shardingNode name="dn119" dbGroup="localhost19" database="db11" />
```

注意: 如果是使用通配符的配置, 那么shardingNode(name)的通配符展开个数必须等于dbGroup通配符展开个数与database通配符展开个数之积, 上例中, name的个数为20, dbGroup的个数为10, database的个数为2; 又例如

```
<shardingNode name="dn,dn1$0-19,dnx" dbGroup="localhost,localhost1$0-9" database="db1$0-1" />
```

中, name的个数为22, dbGroup的个数为11, database的个数为2。

注意: 若出现两个不同的shardingNode拥有同样的database以及dbGroup, 在配置检查的时候会报错(包括从通配符批量生成的shardingNode); 另外, shardingNode的name不能与apNode重名。

1.5.4 apNode配置

- apNode

注意: apNode的位置应在shardingNode的后面, 不然如下配置会引发报错

配置名称	配置内容&示例	详细描述
name	数据节点名称, 唯一, 例如"dn,dn\$0-5"	作为数据节点的标识以及键, 节点个数的计算方法为: 从值出发, 以',' (逗号) 分隔字符串, 如果有连续几项拥有相同的字符串前缀X(不能为空)并且后续其他几位为连续的数字时 (比如0到5), 可以以"X\$0-5"来省略表示, 个数为: 以逗号分隔的字符串个数加上包含\$的连续个数。name的个数必须等于database与dbGroup的个数之积。 注意: apNode之间以及与shardingNode之间, name不能重复
database	apNode对应的存在于clickHouse物理实例中的schema, 可以配置单个或多个使用, 例如"db,db\$0-5"	所使用的详细数据库节点, 节点个数的计算方法为: 从值出发, 以',' (逗号) 分隔字符串, 如果有连续几项拥有相同的字符串前缀X(不能为空)并且后续其他几位为连续的数字时 (比如0到5), 可以以"X\$0-5"来省略表示, 个数为: 以逗号分隔的字符串个数加上包含\$的连续个数。
dbGroup	apNode对应的数据库组, 参考db.xml中的dbGroup名称, 可以配置单个或多个使用, 例如"dh,dh\$0-5"	用于关联对应的Host节点, 节点个数的计算方法为: 从值出发, 以',' (逗号) 分隔字符串, 如果有连续几项拥有相同的字符串前缀X(不能为空)并且后续其他几位为连续的数字时 (比如0到5), 可以以"X\$0-5"来省略表示, 个数为: 以逗号分隔的字符串个数加上包含\$的连续个数。 注意: 对应的dbGroup的databaseType必须是clickhouse

例如:

```
<apNode name="apNode1" dbGroup="localhost1" database="ap_db1" />
```

name, dbGroup, database均可用如下格式在单个配置中配置多个节点: xxx\$n0-n1, xxx, 这种格式的意义为: xxxn0, ..., xxxnm, ..., xxxn1, xxx, 其中

```
n0 < nm < n1。
```

例如: 配置

```
<apNode name="apNode1$0-19" dbGroup="localhost1$0-9" database="ap_db1$0-1" />
```

等同于:

```
<apNode name="apNode10" dbGroup="localhost10" database="ap_db10" />
<apNode name="apNode11" dbGroup="localhost10" database="ap_db11" />
<apNode name="apNode12" dbGroup="localhost11" database="ap_db10" />
<apNode name="apNode13" dbGroup="localhost11" database="ap_db11" />
...
<apNode name="apNode119" dbGroup="localhost19" database="ap_db11" />
```

注意: 如果是使用通配符的配置, 那么apNode(name)的通配符展开个数必须等于dbGroup通配符展开个数与database通配符展开个数之积, 上例中, name的个数为20, dbGroup的个数为10, database的个数为2; 又例如

```
<apNode name="dn,dn1$0-19,dnx" dbGroup="localhost,localhost1$0-9" database="db1$0-1" />
```

中, name的个数为22, dbGroup的个数为11, database的个数为2。

注意: 若出现两个不同的apNode拥有同样的database以及dbGroup, 在配置检查的时候会报错(包括从通配符批量生成的apNode); 另外, apNode的name不能与shardingNode重名。

1.5.5 function配置

拆分算法定义有如下形式:

name: 定义分区算法名, 在分区规则定义中被引用。class: 指定分区算法实现类。每一种分区算法要求的参数个数, 类型各不相同, property name部分用于指定相应分区算法的参数。这部分请参考各分区算法描述。

- function

配置名称	配置内容	说明
name	函数的名称	在分区规则定义中被引用
class	拆分算法	只能是Enum, NumberRange, Hash, StringHash, Date, PatternRange, jumpStringHash之一
property	根据具体的function代码示例的属性进行配置参数	可以配置多个属性

举例:

```
<function name="rang-long" class="com.actiontech.dble.route.function.AutoPartitionByLong">
  <property name="mapFile">auto-sharding-long.txt</property>
  ...
</function>
```

支持的分区算法：目前，已支持的分区算法有：hash, stringhash, enum, numberrange, patternrange, date, jumpstringhash.

1.5.5.1.hash分区算法

function的class属性设置为“hash”或者“com.actiontech.dble.route.function.PartitionByLong”的分区规则应用该算法。具体配置如下：

```
<function name="hashLong" class="hash">
  <property name="partitionCount">C1[,C2, ...Cn]</property>
  <property name="partitionLength">L1[,L2, ...Ln]</property>
</function>
```

partitionCount:指定分区的区间数，具体为 $C1 [+C2 + \dots + Cn]$.

partitionLength:指定各区间长度，具体区间划分为 $[0, L1), [L1, 2L1), \dots, [(C1-1)L1, C1L1), [C1L1, C1L1+L2), [C1L1+L2, C1L1+2L2), \dots$ 其中，每一个区间对应一个数据节点。

例如，配置F1：

```
<property name="partitionCount">2,3</property>
<property name="partitionLength">100,50</property>
```

将划分如下的分区：

$[0, 100) [100, 200) [200, 250) [250, 300) [300, 350)$

再如,配置F2:

```
<property name="partitionCount">2</property>
<property name="partitionLength">1000</property>
```

将划分如下的分区： $[0, 1000) [1000, 2000)$

根据具体配置，模的基数M有如下计算公式： $C1L1 + \dots + Cn Ln$. 上面的例子中F1的M值为350，F2的M值为2000。在进行分片查找时，将分区字段key和M值进行求模运算： $value = key \bmod M$ 得到的value值再从区间分布中找到自己数据节点的序号。例如，当配置为F1, key =805 时, value = 105,那么从5个区间内发现对应的数据节点的序号为1(从0开始)。

结点的个数N记为 $C1 [+C2 + \dots + Cn]$. 上面的例子中F1的N值为5，F2的N值为2。

注意事项：

1. M不能大于2880。2880的原因是这样的:2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 30, 32, 36, 40, 45, 48, 60, 64, 72, 80, 90, 96, 120, 144, 160, 180, 192, 240, 288, 320, 360, 480, 576, 720, 960, 1440是2880的约数,这样预分片扩容方便。
2. N必须要等于逻辑表的shardingNode属性指定的shardingNode数量之和，如shardingNode="dn1,dn2,dn3,dn4"中，N必须等于4。
3. Cn和Ln的个数必须相等。
4. 分区字段必须为整型字段，如果是其他类型，要求值可转化为数字。
5. 当partitionLength为1时，hash分区算法退化为求模算法，M及N均为partitionCount的值。
6. NULL作为分片列的值的时候数据的结果恒落在0号节点（第一个节点上）

1.5.5.2.stringhash分区算法

class属性设置为“stringhash”或者“com.actiontech.dble.route.function.PartitionByString”的分区规则应用该算法。具体配置如下：

```
<function name="hashSting" class="stringhash">
  <property name="partitionCount">C1[,C2, ...Cn]</property>
  <property name="partitionLength">L1[,L2, ...Ln]</property>
  <property name="hashSlice">l:r</property>
</function>
```

partitionCount，**partitionLength**的具体意义参看hash分区算法。**hashSlice**：指定参与hash值计算的key的子串。字符串从0开始索引计数。

hashSlice常见用法：

1. "0:" 或者 "0:0" 代表使用整个字符串
2. "0:50" 代表截取字符串前50个字符（长度不足50就使用整个字符串）

3. "0:-10" 代表截取除了最后10个字符以外的所有其他字符（长度不足10就使用空字符串）

hashSlice子串截取有如下计算步骤和格式：

步骤1. 子串索引区间格式

格式	条件	区间
n	$n \geq 0$	(0,n)
n	$n < 0$	(n,0)
:r		(0,r)
l:		(l,0)
:		(0:0)
l:r		(l, r)

步骤2. 子串索引区间修订

a. 左边界l修订

第一次修订			第二次修订	
修订前	条件	修订后	条件	修订后
l	$l \geq 0$	l		l
l	$l < 0$	$l = l + \text{length}$	$l < 0$	$l = 0$

b. 右边界r修订

第一次修订			第二次修订	
修订前	条件	修订后	条件	修订后
r	$r > 0$	r	$r > \text{length}$	$r = \text{length}$
r	$r \leq 0$	$r = r + \text{length}$		r

注：*length*是分区字段实际串的长度。

步骤3. 结果确定

条件	结果子串	hash值
$l < r$	索引在[l, r)范围的子串	通过子串计算的hash值
$l \geq r$	空字符串	0

这个子串截取算法只是看起来比较复杂。用简洁但不太准确的语言可描述为：区间边界为负值的，从分区字段串尾部开始计数；区间边界为正值的，从分区字段串首部开始计数；然后做一些纠错处理。

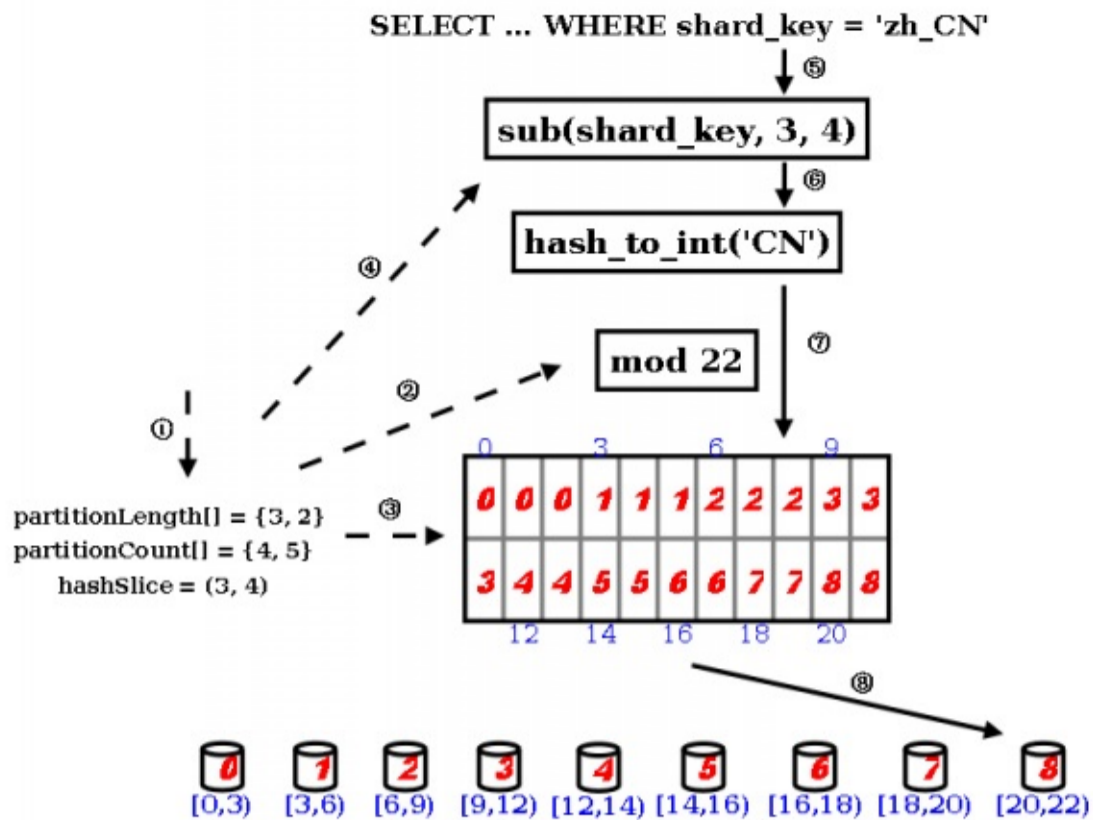
该算法与hash分区算法类似，但针对的分区字段为字符串类型。在进行分片查找时，要先对分区字段求hash值，然后将得到的hash值做hash分区算法的分区查找运算。

注意事项：

1. 该分区算法和hash分区算法有同样的限制(注意事项3除外)
2. 分区字段为字符串类型。

算法解析：

stringhash 按照用户定义的起点和终点去截取分片索引字段中的部分字符，根据当中每个字符的二进制 unicode 值换算出一个长整型数值，然后就直接调用内置 hash 算法求解分片路由：先求模得到逻辑分片号，再根据逻辑分片号直接映射到物理分片。



- 用户需要定义 `partitionLength[]` 和 `partitionCount[]` 两个数组和 `hashSlice` 二元组。
- 在 DBLE 的启动阶段，点乘两个数组得到模数，也是逻辑分片数量。
- 并且根据两个数组的叉乘，得到各个逻辑分片到物理分片的映射表（物理分片数量由 `partitionCount[]` 数组的元素值之和）。
- 此外根据 `hashSlice` 二元组，约定把分片索引值中的第 4 字符到第 5 字符（字符串以 0 开始编号，编号 3 到编号 4 等于第 4 字符到第 5 字符）字符串用于“字符串->整型”的转换。
- 在 DBLE 的运行过程中，用户访问使用这个算法的表时，WHERE 子句中的分片索引值会被提取出来，取当中的第 4 个字符到第 5 字符，送入下一步。
- 设置一个初始值为 0 的累计值，逐个取字符，把累计值乘以 31，再把这个字符的 unicode 值当成整型加入到累计值中，如此类推直至处理完截取出来的所有字符，此时的累计值就能够代表用户的分片索引值，完成了“字符串->整型”的转换。
- 对上一步的累计值进行求模，得到逻辑分片号。
- 再根据逻辑分片号，查映射表，直接得到物理分片号。

1.5.5.3.enum分区算法

class属性设置为“enum”或者“com.actiontech.dble.route.function.PartitionByFileMap”的分区规则应用该算法。具体配置如下：

```
<function name="enum" class="enum">
  <property name="mapFile">partition.txt</property>
  <property name="defaultNode">0</property>
  <property name="type">0</property>
</function>
```

mapFile:指定配置文件名。其格式将在下面做详细说明。**defaultNode**: 指定默认节点号。默认值为-1，不指定默认节点。**type**: type值必须为整数，用来指定配置文件中key的类型。0: 整型；其它非0整数: 字符串。

配置文件格式如下：a. type值为0时，

```
#comment //comment this line will be skipped
int1=node0
int2=node1
...
```

b. type值为非0时，

```
#comment //comment this line will be skipped
string1=node0
string2=node1
...
```

在进行分片查找时，分片字段的枚举值对应的数据节点既是目的节点。如果不能在配置映射中找到枚举值对应的数据节点：如果配置了 `defaultNode`，则 `defaultNode` 既是目的数据节点，否则出错。

注意事项：

1. 不包含“=”的行将被跳过。
2. nodex为数据节点索引号。
3. 重复的枚举值的分区数据节点以最后一个配置为准。

4. 分片字段为该枚举类型。
5. 分片字段为NULL时，数据落在defaultNode节点上，若此时defaultNode没有配置，则会报错;当真实存在于mysql的字段值为not null的时候,报错 "Sharding column can't be null when the table in MySQL column is not null"

1.5.5.4.numberrange分区算法

class属性设置为“numberrange”或者“com.actiontech.dble.route.function.AutoPartitionByLong”的分区规则应用该算法。具体配置如下：

```
<function name="rangeLong" class="numberrange">
  <property name="mapFile">partition.txt</property>
  <property name="defaultNode">0</property>
</function>
```

mapFile:指定配置文件名。其格式将在下面做详细说明。**defaultNode**:指定默认节点号。默认值为-1，不指定默认节点。

配置文件格式如下：`#comment //comment this line will be skipped start1-end1=node1 start2-end2=node2 ...`

该分区算法相当于定义了区间序列 [start1, end1], [start2, end2], ... 每一个区间对应一个数据节点。在进行分片查找时，分片字段的值落在的区间对应的数据节点即是目的节点。如果不能在配置映射中找到分片字段的值所落的区间时，分两种情况：1.配置了defaultNode，则defaultNode是目的数据节点；2.没配defaultNode，出错。注意事项：

1. 不包含“=”的行将被跳过。
2. nodex为数据节点索引号。
3. 如果区间存在重合，在对重合部分的分片字段值进行分片查找时在配置文件中最先定义的区间对应的数据节点为目的节点。
4. 分片字段为整型。
5. 分片字段为NULL时，数据落在defaultNode节点上，若此时defaultNode没有配置，则会报错;当真实存在于mysql的字段值为not null的时候,报错 "Sharding column can't be null when the table in MySQL column is not null"

1.5.5.5.patternrange分区算法

class属性设置为“patternrange”或者“com.actiontech.dble.route.function.PartitionByPattern”的分区规则应用该算法。具体配置如下：

```
<function name="pattern" class="patternrange">
  <property name="mapFile">partition.txt</property>
  <property name="patternValue">1024</property>
  <property name="defaultNode">0</property>
</function>
```

mapFile:指定配置文件名。其格式将在下面做详细说明。**patternValue**:指定模值，默认为1024。**defaultNode**:指定默认节点号。默认值为-1，不指定默认节点。

配置文件格式如下：`#comment //comment this line will be skipped start1-end1=node1 start1-end2=node2 ...`

该分区算法类似于numberrange分区算法。但在进行分片查找时先对分片字段值进行模patternValue运算，然后将得到的模数进行等同于numberrange分区算法的分区查找。注意事项：

1. 不包含“=”的行将被跳过。
2. nodex为数据节点索引号。
3. 如果区间存在重合，在对重合部分的分片字段值进行分片查找时在配置文件中最先定义的区间对应的数据节点为目的节点。
4. 分片字段的内容必须可以转化为整数。如果不能转化为整数：如果配置了defaultNode，目的数据节点为defaultNode；否则，出错。
5. 分片字段为NULL时，数据落在defaultNode节点上，若此时defaultNode没有配置，则会报错;当真实存在于mysql的字段值为not null的时候,报错 "Sharding column can't be null when the table in MySQL column is not null"

1.5.5.6.date分区算法

class属性设置为“date”或者“com.actiontech.dble.route.function.PartitionByDate”的分区规则应用该算法。具体配置如下：

```
<function name="partbydate" class="date">
  <property name="dateFormat">yyyy-MM-dd</property>
  <property name="sBeginDate">2015-01-01</property>
  [<property name="sEndDate">2015-01-31</property>]
  <property name="sPartitionDay">10</property>
  <property name="defaultNode">0</property>
</function>
```

dateFormat:指定日期的格式。**sBeginDate**:指定日期的开始时间。**sEndDate**:指定日期的结束时间。该性质可以不配置或配置为空("")。**sPartitionDay**:指定分区的间隔，单位是天。**defaultNode**:指定默认节点号。默认值为-1，不指定默认节点。

该算法有两种工作模式：**模式1**：不配置sEndDate或者将sEndDate配置为"" 在这种模式下，该算法将时间以sBeginDate为开始，以sPartitionDay为间隔，进行区间划分，每个区间对应一个数据节点。在进行区间查找时，分区字段值落在的区间对应的数据节点即是目的数据节点。如果分区字段值小于sBeginDate：如果配置了defaultNode，则数据会被路由至defaultNode；如果没配置defaultNode,则会报错。**模式2**：配置sEndDate且不为"" 在这种模式下，该算法将时间以sBeginDate为开始，以sPartitionDay为间隔，以sEndDate为终点，进行区间划分，划分为N个区间，每个区间对应一个数据节点。在进行区间查找时：如果分区字段值小于等于sEndDate，则计算过程和结果等同于模式1。如果分区字段值大sEndDate，则分片查找公式为： $index = ((key - sBeginDate) / sPartitionDay) \% N$ ，其中key为分片字段值，index为映射到的数据节点索引。这等同于一个环形映射。如果分区字段值小于sBeginDate，则会检查是否设置了defaultNode。如果设置了，数据会路由至defaultNode;否则报错。

注意事项：

1. 分片字段必须是符合dateFormat的日期字符串。
2. 区间划分不以日历时间为准，无法对应到日历时间。内部进行区间划分时，会将sPartitionDay转化为以86400000毫秒为一天进行运算。
3. 在模式2的情况下，如果(sEndDate - sBeginDate)不是sPartitionDay的整数倍，则索引号为0的数据节点承载更多的数据。
4. 分片字段为NULL时，数据落在defaultNode节点上，若此时defaultNode没有配置，则会报错;当真实存在于mysql的字段值为not null的时候,报错 "Sharding column can't be null when the table in MySQL column is not null"

1.5.5.7.跳增字符串算法

class属性设置为"jumpstringhash"或者"com.actiontech.dble.route.function.PartitionByJumpConsistentHash"的分区规则应用该算法，具体配置如下

```
<function name="jumphash"
  class="jumpStringHash">
  <property name="partitionCount">2</property>
  <property name="hashSlice">0:2</property>
</function>
```

partitionCount:分片数量 **hashSlice**:分片截取长度，具体见1.5.5.2。该算法来自于Google的一篇文章[A Fast, Minimal Memory, Consistent Hash Algorithm](#)其核心思想是通过概率分布的方法将一个hash值在每个节点分布的概率变成1/n，并且可以通过更简便的方法可以计算得出，而且分布也更加均匀

注意事项：

1. 分片字段值为NULL时，数据恒落在0号节点之上;当真实存在于mysql的字段值为not null的时候,报错 "Sharding column can't be null when the table in MySQL column is not null"
2. 如果不设置hashSlice, 3.21.02以及之前版本默认值是 (0:-1) ,不是一个很好的默认值。故在3.21.06改为了默认值 (0:0)

1.5.6 完整配置举例

举例：

```
<?xml version="1.0"?>
<!--
  ~ Copyright (C) 2016-2020 ActionTech.
  ~ License: http://www.gnu.org/licenses/gpl.html GPL version 2 or higher.
-->

<dble:sharding xmlns:dble="http://dble.cloud/" version="4.0">

  <schema name="testdb" sqlMaxLimit="100">
    <shardingTable name="tb_enum_sharding" shardingNode="dn1,dn2" sqlMaxLimit="200" function="func_enum" shardingColumn="code"/>
    <shardingTable name="tb_range_sharding" shardingNode="dn1,dn2,dn3" function="func_range" shardingColumn="id" specifyCharset="false"/>
  </schema>

  <!--er tables-->
  <shardingTable name="tb_hash_sharding" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="id"/>
  <shardingTable name="tb_hash_sharding_er1" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="id"/>
  <shardingTable name="tb_hash_sharding_er2" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="id2"/>
  <shardingTable name="tb_hash_sharding_er3" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="id" incrementColumn="id2"/>

  <shardingTable name="tb_uneven_hash" shardingNode="dn1,dn2,dn3" function="func_uneven_hash" shardingColumn="id"/>

  <shardingTable name="tb_mod" shardingNode="dn1,dn2,dn3,dn4" function="func_mod" shardingColumn="id" sqlRequiredSharding="true"/>

  <shardingTable name="tb_jump_hash" shardingNode="dn1,dn2" function="func_jumpHash" shardingColumn="code"/>

  <shardingTable name="tb_hash_string" shardingNode="dn1,dn2,dn3,dn4" function="func_hashString" shardingColumn="code"/>

  <shardingTable name="tb_date" shardingNode="dn1,dn2,dn3,dn4" function="func_date" shardingColumn="create_date"/>

  <shardingTable name="tb_pattern" shardingNode="dn1,dn2" function="func_pattern" shardingColumn="id"/>
  <!--global tables-->
  <globalTable name="tb_global1" shardingNode="dn1,dn2" sqlMaxLimit="103" specifyCharset="false"/>
  <globalTable name="tb_global2" shardingNode="dn1,dn2,dn3,dn4" cron="0 0 0 * * ?" checkClass="CHECKSUM"/>
  <!--single node table-->
  <singleTable name="tb_single" shardingNode="dn6" sqlMaxLimit="105"specifyCharset="false"/>
  <!--er tables-->
```

```

<shardingTable name="tb_parent" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="id">
  <childTable name="tb_child1" joinColumn="child1_id" parentColumn="id" sqlMaxLimit="201">
    <childTable name="tb_grandson1" joinColumn="grandson1_id" parentColumn="child1_id" specifyCharset="false"/>
    <childTable name="tb_grandson2" joinColumn="grandson2_id" parentColumn="child1_id2" specifyCharset="false"/>
  </childTable>
  <childTable name="tb_child2" joinColumn="child2_id" parentColumn="id"/>
  <childTable name="tb_child3" joinColumn="child3_id" parentColumn="id2"/>
</shardingTable>
</schema>
<!-- sharding testdb2 route to database named dn5 in localhost2 -->
<schema name="testdb2" shardingNode="dn5"/>
<shardingNode name="dn1" dbGroup="dbGroup1" database="db_1"/>
<shardingNode name="dn2" dbGroup="dbGroup2" database="db_2"/>
<shardingNode name="dn3" dbGroup="dbGroup1" database="db_3"/>
<shardingNode name="dn4" dbGroup="dbGroup2" database="db_4"/>
<shardingNode name="dn5" dbGroup="dbGroup1" database="db_5"/>
<shardingNode name="dn6" dbGroup="dbGroup2" database="db_6"/>
<!-- enum partition -->
<function name="func_enum" class="Enum">
  <property name="mapFile">partition-enum.txt</property>
  <property name="defaultNode">0</property><!--the default is -1,means unexpected value will report error-->
  <property name="type">0</property><!--0 means key is a number, 1 means key is a string-->
</function>
<!-- number range partition -->
<function name="func_range" class="NumberRange">
  <property name="mapFile">partition-number-range.txt</property>
  <property name="defaultNode">0</property><!--he default is -1,means unexpected value will report error-->
</function>
<!-- Hash partition,when partitionLength=1, it is a mod partition, MAX(sum(count*length[i]) must not more then 2880-->
<function name="func_common_hash" class="Hash">
  <property name="partitionCount">2</property>
  <property name="partitionLength">512</property>
</function>
<!-- Hash partition,when partitionLength=1, it is a mod partition, MAX(sum(count*length[i]) must not more then 2880-->
<function name="func_uneven_hash" class="Hash">
  <property name="partitionCount">2,1</property>
  <property name="partitionLength">256,512</property>
</function>
<!-- eg: mod 4 -->
<function name="func_mod" class="Hash">
  <property name="partitionCount">4</property>
  <property name="partitionLength">1</property>
</function>
<!-- jumpStringHash partition for string-->
<function name="func_jumpHash" class="jumpStringHash">
  <property name="partitionCount">2</property>
  <property name="hashSlice">0:2</property>
</function>
<!-- Hash partition for string-->
<function name="func_hashString" class="StringHash">
  <property name="partitionCount">4</property>
  <property name="partitionLength">256</property>
  <property name="hashSlice">0:2</property>
  <!--<property name="hashSlice">-4:0</property> -->
</function>
<!-- date partition 4 case:
1.set sEndDate and defaultNode: input <sBeginDate ,router to defaultNode; input>sEndDate ,mod the period
2.set sEndDate, but no defaultNode:input <sBeginDate report error; input>sEndDate ,mod the period
3.set defaultNode without sEndDate: input <sBeginDate router to defaultNode;input>sBeginDate + (node size)*sPartionDay-1 will report error(expected is defaultNode,but can't control now)
4.sEndDate and defaultNode are all not set: input <sBeginDate report error;input>sBeginDate + (node size)*sPartionDay-1 will report error
-->
<function name="func_date" class="Date">
  <property name="dateFormat">yyyy-MM-dd</property>
  <property name="sBeginDate">2015-01-01</property>
  <property name="sEndDate">2015-01-31</property> <!--if not set sEndDate,then in fact ,the sEndDate = sBeginDate+ (node size)*sPartionDay-1 -->
  <property name="sPartionDay">10</property>
  <property name="defaultNode">0</property><!--the default is -1-->
</function>
<!-- pattern partition : mapFile must contains all value of 0-patternValue-1,key and value must be Continuous increase-->
<function name="func_pattern" class="PatternRange">
  <property name="mapFile">partition-pattern.txt</property>
  <property name="patternValue">1024</property>
  <property name="defaultNode">0</property><!--contains string which is not number,router to default node-->
</function>
</db:sharding>

```


1.6 log4j2.xml

1.6.1 配置详述

Dble中的整体配置和一般java项目的log4j2.xml没有什么区别

1.6.1.1 日志滚动删除配置

日志滚动删除是通过DefaultRolloverStrategy的配置对于RollingRandomAccessFile的RolloverStrategy内容进行重载

```
<DefaultRolloverStrategy max="100">
  <Delete basePath="logs" maxDepth="2">
    <IfFileName glob="*/dble-*.log.gz">
      <IfLastModified age="2d">
        <IfAny>
          <IfAccumulatedFileSize exceeds="1 GB" />
          <IfAccumulatedFileCount exceeds="10" />
        </IfAny>
      </IfLastModified>
    </IfFileName>
  </Delete>
</DefaultRolloverStrategy>
```

上例中参数说明如下:

basePath: 基准路径, 日志的统计归类和删除会在此目录下进行

maxDepth: 最大路径深度, 在basePath路径下maxDepth深度的日志文件都会被扫描, 譬如.../logs/2018-01-01 .../logs/2018-01-02 此类路径也都会被扫描和统计

glob: 日志格式, 在所有扫描到的文件中符合此命名规范的文件都被认为是日志文件

age: 举例最后一次修改文件的时间, 只有修改时间超过限值的文件才会被考虑删除

IfAccumulatedFileSize: 触发文件大小, 符合上述条件的文件大小总量达到触发值则触发清理

IfAccumulatedFileCount: 触发文件数量, 符合上述条件的文件数量达到触发值则触发清理

例子详述: 在此例中代表会监控logs目录下2层目录深度内所有命名符合"/dble-.log.gz"并且最后修改时间已经超出2天的文件, 当有符合上述条件的文件大小到达1GB或者文件数量到达10的时候会触发清理

1.6.2 配置实例

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN" packages="com.actiontech.dble.log">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d [%-5p][%t] %m %throwable{full} (%C:%F:%L) %n"/>
    </Console>

    <RollingRandomAccessFile name="RollingFile" fileName="logs/dble.log"
      filePattern="logs/${date:yyyy-MM}/dble-%d{MM-dd}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %5p [%t] (%l) - %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <OnStartupTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="250 MB"/>
        <TimeBasedTriggeringPolicy/>
      </Policies>
      <DefaultRolloverStrategy max="100">
        <Delete basePath="logs" maxDepth="2">
          <IfFileName glob="*/dble-*.log.gz">
            <IfLastModified age="2d">
              <IfAny>
                <IfAccumulatedFileSize exceeds="1 GB" />
                <IfAccumulatedFileCount exceeds="10" />
              </IfAny>
            </IfLastModified>
          </IfFileName>
        </Delete>
      </DefaultRolloverStrategy>
    </RollingRandomAccessFile>

    <RollingFile name="ThreadChecker" fileName="logs/thread.log"
      filePattern="logs/${date:yyyy-MM}/thread-%d{MM-dd}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %5p [%t] (%l) - %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <OnStartupTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="250 MB"/>
        <TimeBasedTriggeringPolicy/>
      </Policies>
      <DefaultRolloverStrategy max="10"/>
    </RollingFile>
  </Appenders>
  <Loggers>
    <Logger name="ThreadChecker" additivity="false" includeLocation="false">
      <AppenderRef ref="ThreadChecker"/>
    </Logger>
    <asyncRoot level="debug" includeLocation="true">
      <AppenderRef ref="Console"/>
      <AppenderRef ref="RollingFile"/>
    </asyncRoot>
  </Loggers>
</Configuration>
```

1.7 全局序列

在分库分表的情况下，数据库自增主键无法保证自增主键的全局唯一。为此，dble提供了全局序列，并且针对不同应用场景提供了多种实现方式。虽然提供了多种方式，但在生产环境不建议修改，因为它的各个值之间生成的序列是不兼容的，修改可能会破坏序列的唯一性。要应用全局序列，首先需要在cluster.cnf 中配置：

```
sequenceHandlerType=n
```

根据sequenceHandlerType的类型配置具体的实现：

- 1:MySQL offset-step方式
- 2:时间戳方式(类Snowflake)
- 3:分布式时间戳方式(类Snowflake)
- 4:分布式offset-step方式

其次，使用全局序列的表要在sharding.xml中配置表格并指定其自增列。

自增列的指定逻辑：以显式声明的以incrementColumn中的指定为准

```
//有显式指定incrementColumn=pid, 则pid作为自增列
<shardingTable name="table1" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="id" incrementColumn="pid"/>
```

配置之后的使用示例：

```
insert into table1(name) values('test');
insert into table1 set name = 'test';
```

每一个全局序列的具体实现配置将在各个小节进行详细说明；其功能将在[2.2 全局序列](#)进行详细描述。

与MySQL差异

在MySQL中要求自增列必须含有唯一键，在dble中对于表格的自增列的唯一属性不做要求，但是在dble进行插入数据操作的时候不允许用户手动插入自增列的数值，自增列只能由dble自己生成的ID进行填充

并且存在以下情形和mysql的行为不一致

```
table1拥有列aid,bid,cid,did 其中bid为在dble中的自增列
insert into table1 values(1,2,3)
和以下sql的效果相等
insert into table1 set aid = 1,cid = 2,did = 3
```

特别提醒 在dble中全局序列只在生成的时候确保其唯一性，在之后的过程中用户被允许使用update或者replace语句进行更新自增字段（特例：自增字段同时也是分片字段是除外）。这给予用户提供了充足的修改空间但同时要求用户在更新自增字段的时候有足够的谨慎和了解

1.7.1 MySQL-offset-step 方式

1.7.1.1 MySQL-offset-step序列配置

mysql序列由文件sequence_db_conf.properties进行配置。具体配置有如下格式:

```
#this is comment

`schema1`.`table1`=node1

`schema1`.`table2`=node1

`schema2`.`table1`=node2

...
```

schemaX: 使用全局序列的dble表所属的dble库名。

tableX: 使用全局序列dble的dble表名。

nodeX: 实现序列功能的数据节点名。此节点具体配置见下节。

1.7.1.2 数据节点配置

mysql序列的实现依赖一些存储函数。因此在序列可用之前必须在数据节点**nodeX**上创建这些存储函数，并针对使用此节点的表（**schemaX`.`tableX`**）初始化序列初始值。

数据节点**nodeX**上创建必要的表和存储函数

具体步骤:

a. 用mysql客户端登录到**nodeX**上

```
mysql ...
```

b. 切换到**nodeX**上的后端存储数据库db(参见1.5 sharding.xml)

```
use db;
```

c. 执行创建脚本dbseq.sql (dbseq.sql的内容详见1.7.1.3节)

```
source ../dbseq.sql;
```

d. 初始化相应序列初始值

```
INSERT INTO DBLE_SEQUENCE VALUES (`schemaX`.`tableX`, 1, 1);
```

```
...
```

1.7.1.3 dbseq.sql内容

dbseq.sql的内容如下:

```
DROP TABLE IF EXISTS DBLE_SEQUENCE;
CREATE TABLE DBLE_SEQUENCE ( name VARCHAR(64) NOT NULL, current_value BIGINT(20) NOT NULL, increment INT NOT NULL DEFAULT 1, PRIMARY KEY (
name) ) ENGINE=InnoDB;

-- -----
-- Function structure for `dble_seq_currval`
-- -----
DROP FUNCTION IF EXISTS `dble_seq_currval`;
DELIMITER ;;
CREATE FUNCTION `dble_seq_currval`(seq_name VARCHAR(64)) RETURNS varchar(64) CHARSET latin1
DETERMINISTIC
BEGIN
DECLARE retval VARCHAR(64);
SET retval="-1,0";
SELECT concat(CAST(current_value AS CHAR),",",CAST(increment AS CHAR) ) INTO retval FROM DBLE_SEQUENCE WHERE name = seq_name;
RETURN retval ;
END
;;
DELIMITER ;

-- -----
-- Function structure for `dble_seq_nextval`
-- -----
DROP FUNCTION IF EXISTS `dble_seq_nextval`;
DELIMITER ;;
CREATE FUNCTION `dble_seq_nextval`(seq_name VARCHAR(64)) RETURNS varchar(64) CHARSET latin1
DETERMINISTIC
BEGIN
```



```

DECLARE retval VARCHAR(64);
DECLARE val BIGINT;
DECLARE inc INT;
DECLARE seq_lock INT;
set val = -1;
set inc = 0;
SET seq_lock = -1;
SELECT GET_LOCK(seq_name, 15) into seq_lock;
if seq_lock = 1 then
SELECT current_value + increment, increment INTO val, inc FROM DBLE_SEQUENCE WHERE name = seq_name for update;
if val != -1 then
UPDATE DBLE_SEQUENCE SET current_value = val WHERE name = seq_name;
end if;
SELECT RELEASE_LOCK(seq_name) into seq_lock;
end if;
SELECT concat(CAST((val - inc + 1) as CHAR), ",", CAST(inc as CHAR)) INTO retval;
RETURN retval;
END
;;
DELIMITER ;

-- -----
-- Function structure for `dbtle_seq_setvals`
-- -----
DROP FUNCTION IF EXISTS `dbtle_seq_nextvals`;
DELIMITER ;;
CREATE FUNCTION `dbtle_seq_nextvals`(seq_name VARCHAR(64), count INT) RETURNS VARCHAR(64) CHARSET latin1
DETERMINISTIC
BEGIN
DECLARE retval VARCHAR(64);
DECLARE val BIGINT;
DECLARE seq_lock INT;
SET val = -1;
SET seq_lock = -1;
SELECT GET_LOCK(seq_name, 15) into seq_lock;
if seq_lock = 1 then
SELECT current_value + count INTO val FROM DBLE_SEQUENCE WHERE name = seq_name for update;
IF val != -1 THEN
UPDATE DBLE_SEQUENCE SET current_value = val WHERE name = seq_name;
END IF;
SELECT RELEASE_LOCK(seq_name) into seq_lock;
end if;
SELECT CONCAT(CAST((val - count + 1) as CHAR), ",", CAST(count as CHAR)) INTO retval;
RETURN retval;
END
;;
DELIMITER ;

-- -----
-- Function structure for `dbtle_seq_setval`
-- -----
DROP FUNCTION IF EXISTS `dbtle_seq_setval`;
DELIMITER ;;
CREATE FUNCTION `dbtle_seq_setval`(seq_name VARCHAR(64), value BIGINT) RETURNS varchar(64) CHARSET latin1
DETERMINISTIC
BEGIN
DECLARE retval VARCHAR(64);
DECLARE inc INT;
SET inc = 0;
SELECT increment INTO inc FROM DBLE_SEQUENCE WHERE name = seq_name;
UPDATE DBLE_SEQUENCE SET current_value = value WHERE name = seq_name;
SELECT concat(CAST(value as CHAR), ",", CAST(inc as CHAR)) INTO retval;
RETURN retval;
END
;;
DELIMITER ;

```

1.7.2 时间戳方式

时间戳方式由bootstrap.cnf 和cluster.cnf文件进行配置。具体配置如下：

bootstrap.cnf的instanceId：指定instance id值， 必须为[0,1023]之间的整数。

cluster.cnf的sequenceStartTime：指定开始时间戳，格式必须为 YYYY-MM-dd HH:mm:ss，默认开始时间 2010-10-04 09:42:54。

注意事项：**bootstrap.cnf的instanceId**的配置必须使该dble实例在dble集群中唯一。

另外，使用这种方式需要对应字段为bigint来保证63位

配置示例

见[时间戳方式全局序列的配置](#)

1.7.3 分布式时间戳方式

分布式时间戳方式由bootstrap.cnf 和cluster.cnf文件进行配置。具体配置格式如下：

bootstrap.cnf的instanceId：指定instance id值，必须为[0,511]之间的整数，当**cluster.cnf的sequenceInstanceByZk**为true时id由zk生产。

cluster.cnf的sequenceStartTime：指定开始时间戳，格式必须为 YYYY-MM-dd HH:mm:ss，默认开始时间 2010-10-04 09:42:54。。

注意事项：

1. 当cluster.cnf的sequenceInstanceByZk的值配置为true时，必须配置zookeeper服务器(参见1.1 cluster.cnf)。
2. **bootstrap.cnf的instanceId**，保证dble在集群中唯一。
3. 使用这种方式需要对应字段为bigint来保证63位。

1.7.4 分布式offset-step方式

分布式offset-step方式由文件sequence_conf.properties进行配置。具体格式如下：

```
# this is comment
`schema1`.`table1`.MINID=1001
`schema1`.`table1`.MAXID=2000
`schema1`.`table1`.CURID=1000

`schema2`.`table2`.MINID=1001
`schema2`.`table2`.MAXID=20000
`schema2`.`table2`.CURID=1000
```

schemaX：使用mysql序列的dble表所属的dble库名。

tableX：使用mysql序列的dble表名。

注意事项：

1. 每一个zk序列均需要指定当前区间内最小值MINID，当前区间内最大值MAXID，当前区间内当前值CURID。这些值仅在初始配置时有效。
2. 初始配置时MINID要比CURID大1，否则序列值从MINID+1开始。
3. 值 (MAXID - MINID + 1) 为每次从zookeeper服务器获取的序列值数量。
4. 配置dble使用zookeeper服务器，具体见相关配置 [1.1 cluster.cnf](#)。

1.8 cache配置

- [1.8.1 cache配置](#)
- [1.8.2 ehcache配置](#)

1.8.1 cache 配置

1.8.1.1 dble的cache使用

dble的cache使用有如下两类：

- SQLRouteCache：从前端连接收到的SQL以及对应的路由结果 内容： schema_user_SQL -> 具体路由结果RouteResult
- ER_SQL2PARENTID：父子表辅助查询SQL以及对应的路由结果。在插入ER子表的时候需要根据子表joinColumn(父表parentColumn)计算它应该插入的结点，所以需要辅助路由来查询，然后将辅助路由及对应的结果缓存下来以备下次查询。内容为： schema:select * from 父表 where parentKey = (value of joinColumn) -> 对应数据shardingNode

1.8.1.2 dble的cache实现

dble的cache实现有如下几种：

- ehcache, 用ehcache缓存作为cache实现。
- leveldb, 用leveldb数据库作为cache实现。
- mapdb, 用MapDB数据库引擎作为cache实现。
- rocksdb, 用RocksDB数据库引擎作为cache实现。

1.8.1.3 dble的cache配置

dble的cache配置分为总配置和实现配置。总配置由文件cacheservice.properties进行设定。实现配置由各个实现具体指定，具体详见各个实现的分章节说明。

总配置有如下格式：

设置缓存类型：

factory.cache_type=cache_type

设置分类缓存的具体值，key为缓存池名字，value是类型，最大容量，以及失效时间

A.SQL路由缓存

pool.SQLRouteCache=type,max_size,expire_seconds

B.ER表子表路由缓存

pool.ER_SQL2PARENTID=type,max_size,expire_seconds

1.8.1.4 cache配置说明

总配置文件中各配置项说明：

- 以#开头的行为注释，被忽略。空行被忽略。
- factory.cache_type=cache_type是cache的总开关。cache_type指定cache类型，具体可以为：ehcache、leveldb、mapdb 或者rocksdb。如果要用cache功能，必须配置该配置项。这个配置项可以指定多个，每行仅能指定一个。每一个指定一个cache实现。

例如：

配置，

```
factory.encache=ehcache
pool.SQLRouteCache=encache,10000,1800
pool.ER_SQL2PARENTID=encache,1000,1800
```

中的type就必须是ehcache。而配置：

```
factory.encache=ehcache
factory.leveldb=leveldb
pool.SQLRouteCache=encache,10000,1800
pool.ER_SQL2PARENTID=leveldb,1000,1800
```

中的type可以为encache或者leveldb。

- pool.SQLRouteCache=type,max_size,expire_seconds和pool.ER_SQL2PARENTID=type,max_size,expire_seconds分别配置SQLRouteCache和ER_SQL2PARENTID的缓存功能。这两个配置项可以配置也可以不配置，不配值则不使用相应的缓存功能。type指定缓存类型，必须是已配置的缓存实现类型；max_size指定缓存的最大大小，单位是字节；expire_seconds指定缓存项的生命周期，单位是秒。

- default缓存用于缓存没有为其指定特定缓存的表的分区键值到数据所在节点的映射。

1.8.1.5 注意事项

- 使用 RocksDB作为 cache 实现时，需要在dble目录下手工创建 rocksdb 目录，否则dble启动失败。

1.8.2 ehcache缓存配置

要用ehcache实现缓存，必须在cacheservice.properties中配置使用ehcache，具体请参看上一节内容。

1.8.2.1 ehcache版本

目前，dble使用的是2.6.11.

1.8.2.2 ehcache配置

ehcache的配置通过文件ehcache.xml进行。

具体的缓存存储策略和配置请参看<http://www.ehcache.org/documentation/ehcache-2.6.x-documentation.pdf>。

例如：

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="ehcache.xsd" maxEntriesLocalHeap="100000000"
maxBytesLocalDisk="50G" updateCheck="false">
    <defaultCache maxElementsInMemory="1000000" eternal="false" overflowToDisk="false" diskSpoolBufferSizeMB="30" maxElementsOn
Disk="10000000" diskPersistent="false" diskExpiryThreadIntervalSeconds="120" memoryStoreEvictionPolicy="LRU"/>
</ehcache>
```

需要特殊说明是：

1.dble仅用ehcache配置的defaultCache级别创建cache。

2.maxEntriesLocalHeap

该属性指定允许存储元素的最大条数。如果设定了该值且不为0，则缓存大小限制为缓存的**条数限制**，具体限制由cacheservice.properties中**max_size**指定，参见上一节内容。如果没有设定该参数，则缓存大小限制仍为大小限制，具体限制由cacheservice.properties中**max_size**指定，参见上一节内容。

3.timeToldleSeconds

该属性指定一个元素在未被请求的情况下允许在缓存中存在的最长时间。它将被cacheservice.properties中**expire_seconds**取代。

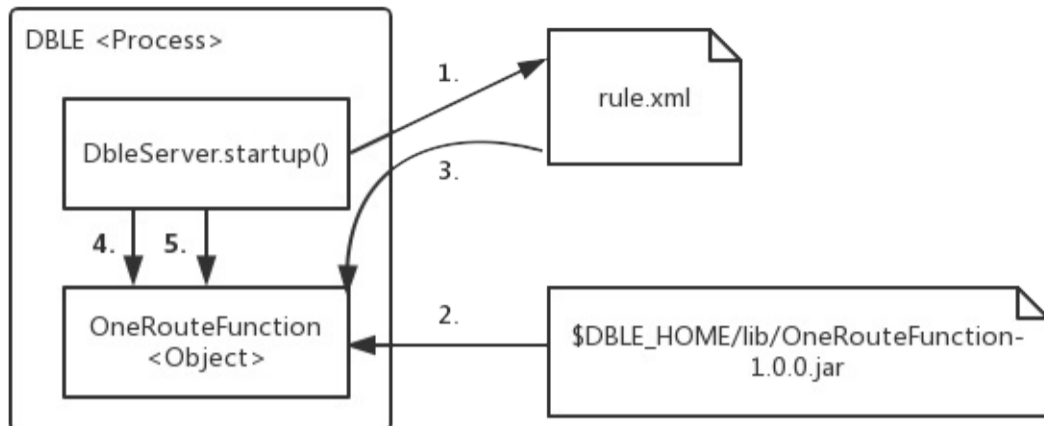
4.dble将defaultCache配置应用到每一个创建的cache。

1.9 自定义拆分算法

1.9.1 工作原理

1.9.1.1 函数的加载

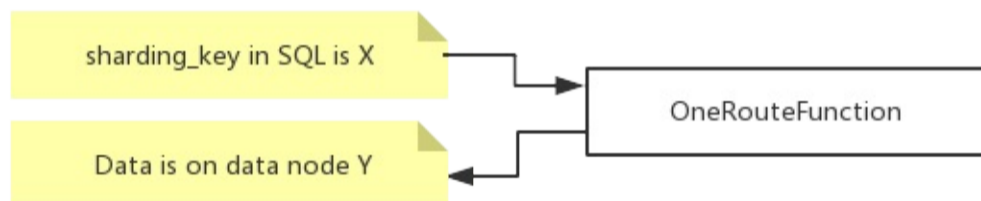
路由函数的加载发生在dble启动或重载时。



1. dble读取sharding.xml时，根据用户配置的标签的class属性
2. dble通过Java的反射机制，从\$DBLE_HOME/lib的jar包中，找到对应的jar（里的class文件），加载同名的类并创建对象
3. dble会逐个扫描中的标签，并根据name属性来调用路由函数的对应setter，以此完成赋值过程——例如，如果用户配置了2，那么dble就会尝试找路由函数中叫做setPartitionCount()的方法，并将字符串“2”传给它
4. dble调用路由函数的selfCheck()方法，执行函数编写者制定的检查动作，例如检查赋值得到的变量值是否有问题
5. dble调用路由函数的init()方法，执行函数编写者制定的准备动作，例如创建后面要用到的一些中间变量

1.9.1.2 路由计算

路由函数接受用户SQL中的分片字段的值，计算出这个值对应的数据记录应该在哪个编号的数据分片（逻辑分片）上，DBLE从而知道把这个SQL准确发到这些分片上。



1.9.1.3 参数查询

用户通过管理端口（默认9066），通过SHOW @@ALGORITHM WHERE SCHEMA=? AND TABLE=?来查询表上的路由算法时，dble调用路由算法的getAllProperties()方法，直接从内存中获取路由信息的配置。

```
mysql> show @@algorithm where schema=testdb and table=seqtest;
+-----+-----+
| KEY          | VALUE          |
+-----+-----+
| TYPE         | SHARDING TABLE |
| COLUMN      | ID             |
| CLASS       | com.actiontech.dble.route.function.PartitionByLong |
| partitionCount | 2             |
| partitionLength | 1             |
+-----+-----+
5 rows in set (0.05 sec)
```

1.9.2 开发和部署

1.9.2.1 开发

开发时，理论上只需要引入AbstractPartitionAlgorithm抽象类和RuleAlgorithm接口及它们的依赖类就可以了。但实际上AbstractPartitionAlgorithm抽象类依赖了TableConfig类，由此开启了环游世界的依赖之旅。因此，现实的操作还是引用整个DBLE项目的源代码会比较直接方便。

开发一个新的路由函数时，必须给这个路由函数的开发新建项目，然后再引用DBLE项目（项目引用项目的方式）。而**不应该**直接打开DBLE的项目，然后在DBLE的项目里面直接新建源代码来直接开发（内嵌开发方式）。通过遵循这个做法，会有以下好处：

1. 路由函数可以独立打包，直接去看路由函数的jar包版本就能够确认函数版本；而把路由函数嵌到DBLE里的话，就容易出现DBLE版本一样，但不清楚里面的

函数是什么版本的窘况

2. 路由函数的递进可以更加自由，如果DBLE的AbstractPartitionAlgorithm抽象类和RuleAlgorithm接口没有变动，同一版本的路由函数可以延续使用好几个版本的DBLE，而不需要每次DBLE释放新版就得去重编译
3. 可以让路由函数中的受保护代码免受DBLE自身的开源协议影响

1.9.2.2 部署

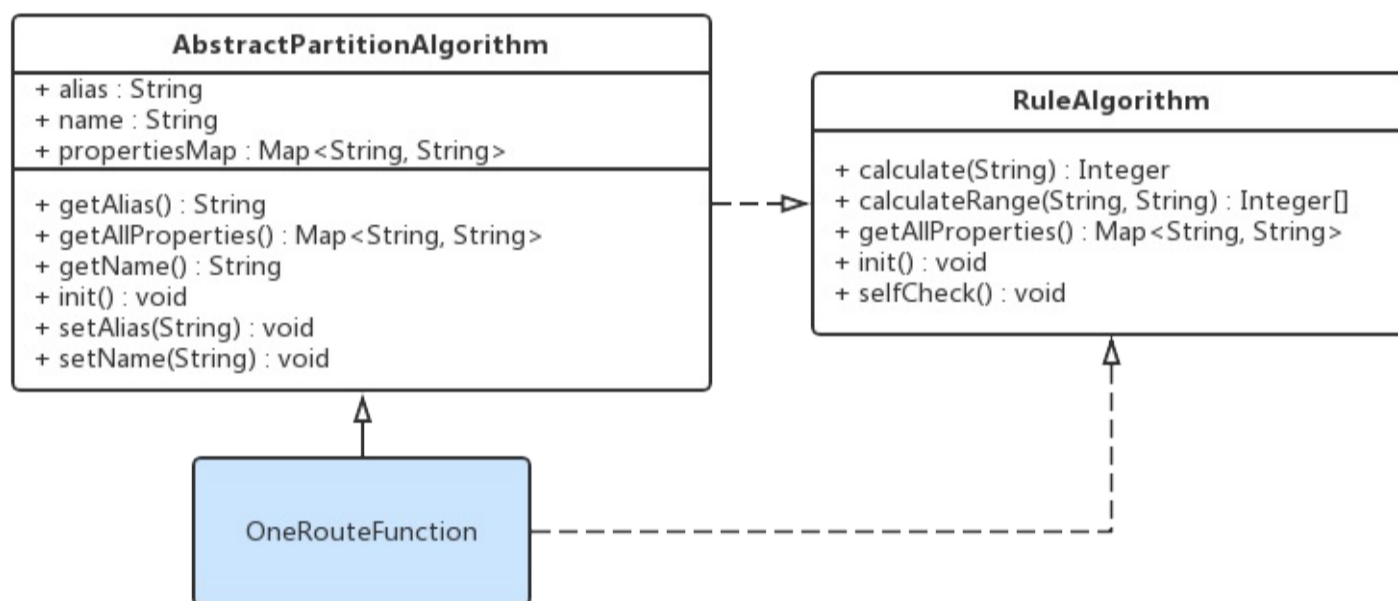
完成开发之后，成品打包成jar包进行发布，而**不要**直接发布class和依赖的library（其他项目的jar包或class文件）。

让DBLE使用上新的路由函数的过程：

1. 将成品jar包放入\$DBLE_HOME/lib目录中
2. 调整jar包的所有者权限（chown）和文件权限（chmod），使之与其他\$DBLE_HOME/lib目录里的jar包一样
3. 按照原来的思路配置sharding.xml，但需要注意标签的class属性必须要填写新的路由函数类的完全限定名（Fully Qualified Name），例如net.john.dble.route.functions.NewFunction
4. 配置逻辑表之类的必要信息，**重启**DBLE后，自动生效。

1.9.3 接口规范

每个路由函数本质上就是一个继承了AbstractPartitionAlgorithm抽象类，并且实现了RuleAlgorithm接口的一个类。下面以内置的com.actiontech.dble.route.function.PartitionByLong为例，介绍实现一个路由函数类所需要做的最小工作（必要工作）。



1.9.3.1 配置项setters

在sharding.xml中，我们需要配置partitionCount和partitionLength两个配置项。

```

<function name="hashmod" class="com">
  <property name="partitionCount">4</property>
  <property name="partitionLength">1</property>
</function>
  
```

为了让dble在函数加载过程中，能够认出这里的partitionCount（值为4）和partitionLength（值为1），因此PartitionByLong类中，就必须有属性设置方法（setter）setPartitionCount()和setPartitionLength()。而因为sharding.xml是个文本型的XML文件，所以这些函数的传入参数就只能是一个String，数据类型转换和预处理的动作就由这些setter来处理了。

```

public void setPartitionCount(String partitionCount) {
    this.count = toIntArray(partitionCount);
    /* 参考本文的getAllProperties()的说明 */
    propertiesMap.put("partitionCount", partitionCount);
}

public void setPartitionLength(String partitionLength) {
    this.length = toIntArray(partitionLength);
    /* 参考本文的getAllProperties()的说明 */
    propertiesMap.put("partitionLength", partitionLength);
}
  
```

1.9.3.2 selfCheck()

在函数加载过程中，完成了配置项赋值之后，dble会调用这个路由函数对象的selfCheck()方法，让这个对象自我检查刚才读进来的配置项的值，放在一起是不是有问题。如果有问题的话，路由函数编写者在这时候，可以通过抛出RuntimeException来进行报错，并终止dble使用这个函数，当然，由于RuntimeException的霸道，dble自己也会因此而报错退出。

由于selfCheck()是RuleAlgorithm接口的要求，而且AbstractPartitionAlgorithm抽象类没又实现它，对于想偷懒或者没有必要进行这个检查的人来说，还是需要自行定义一个空的同名方法来实现它。

```
@Override
public void selfCheck() {
}
```

1.9.3.3 init()

在函数加载过程的最后，dble调用这个路由函数对象的init()方法，让这个对象完成一些内部的初始化工作。

在我们的例子PartitionByLong里，通过init()方法准备了PartitionUtil对象，其中有一个哈希值的范围与逻辑分片号对应的数组，这样在后面的路由计算时就能通过查数组来加速得到结果。

```
@Override
public void init() {
    partitionUtil = new PartitionUtil(count, length);

    initHashCode();
}
```

1.9.3.4 calculate()和calculateRange()

dble执行用户SQL时，根据用户SQL的不同，调用calculate()或calculateRange()来确定用户的SQL应该发到哪个数据分片上去。

从IPO (Input-Process-Output) 来分析，calculate()和calculateRange()的工作原理是一样的：

- Input: 用户SQL中的分片字段值
- Output: 用户SQL应该要发往的数据分片的编号
- Process: Input与Output转换的计算过程，由函数开发者编写

calculate()和calculateRange()的使用场景不同，导致它们存在着一些微小的差异。

函数名	调用场景	Input	Output
calculate()	用户SQL里分片字段的值是单值的情况，例如 ... WHERE sharding_key = 1	1个String	1个Integer
calculateRange()	用户SQL里分片字段的值是连续范围，例如 ... WHERE sharding_key BETWEEN 1 AND 5	2个String	Integer数组

```

@Override
public Integer calculate(String columnValue) {
    try {
        if (columnValue == null || columnValue.equalsIgnoreCase("NULL")) {
            return 0;
        }
        long key = Long.parseLong(columnValue);
        return calculate(key);
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException("columnValue:" + columnValue + " Please eliminate any quote and non number within it.", e);
    }
}

@Override
public Integer[] calculateRange(String beginValue, String endValue) {
    long begin = 0;
    long end = 0;
    try {
        begin = Long.parseLong(beginValue);
        end = Long.parseLong(endValue);
    } catch (NumberFormatException e) {
        return new Integer[0];
    }
    int partitionLength = partitionUtil.getPartitionLength();
    if (end - begin >= partitionLength || begin > end) { //TODO: optimize begin > end
        return new Integer[0];
    }
    Integer beginNode = calculate(begin);
    Integer endNode = calculate(end);

    if (endNode > beginNode || (endNode.equals(beginNode) && partitionUtil.isSingleNode(begin, end))) {
        int len = endNode - beginNode + 1;
        Integer[] re = new Integer[len];

        for (int i = 0; i < len; i++) {
            re[i] = beginNode + i;
        }
        return re;
    } else {
        int split = partitionUtil.getSegmentLength() - beginNode;
        int len = split + endNode + 1;
        if (endNode.equals(beginNode)) {
            //remove duplicate
            len--;
        }
        Integer[] re = new Integer[len];
        for (int i = 0; i < split; i++) {
            re[i] = beginNode + i;
        }
        for (int i = split; i < len; i++) {
            re[i] = i - split;
        }
        return re;
    }
}
}

```

1.9.3.5 getAllProperties()

当用户找dble要路由函数的配置信息时，dble通过访问路由函数的getAllProperties()来获得一个<配置项, 配置值>的哈希表，然后将里面的内容逐项返回给用户。

getAllProperties()是RuleAlgorithm接口所规定要实现的，但为了简化编写新的路由函数的工作，在AbstractPartitionAlgorithm抽象类里，定义了propertiesMap这个私有变量，并且把“将propertiesMap交出去”作为实现了getAllProperties()方法的默认实现。一般来说，这个默认的实现能满足需求，而新路由函数编写者只需要在配置项setters处理用户配置时，将<配置项, 配置值>给put()进propertiesMap里就好了。

```

@Override
public Map<String, String> getAllProperties() {
    return propertiesMap;
}

```

1.9.4 内置路由函数的缩写与类名对照表

DBLE内置的路由函数都位于com.actiontech.dble.route.function命名空间。但实际配置sharding.xml的时候，却不用写那么长的完全限定名，这其实都是XMLRuleLoader类做了转换，因此实现了简写。下面就是7个内置函数的类名和它们的简写。

简写名	完整类名
date	PartitionByDate
enum	PartitionByFileMap
hash	PartitionByLong
jumpstringhash	PartitionByJumpConsistentHash
numberrange	AutoPartitionByLong
patternrange	PartitionByPattern
stringhash	PartitionByString

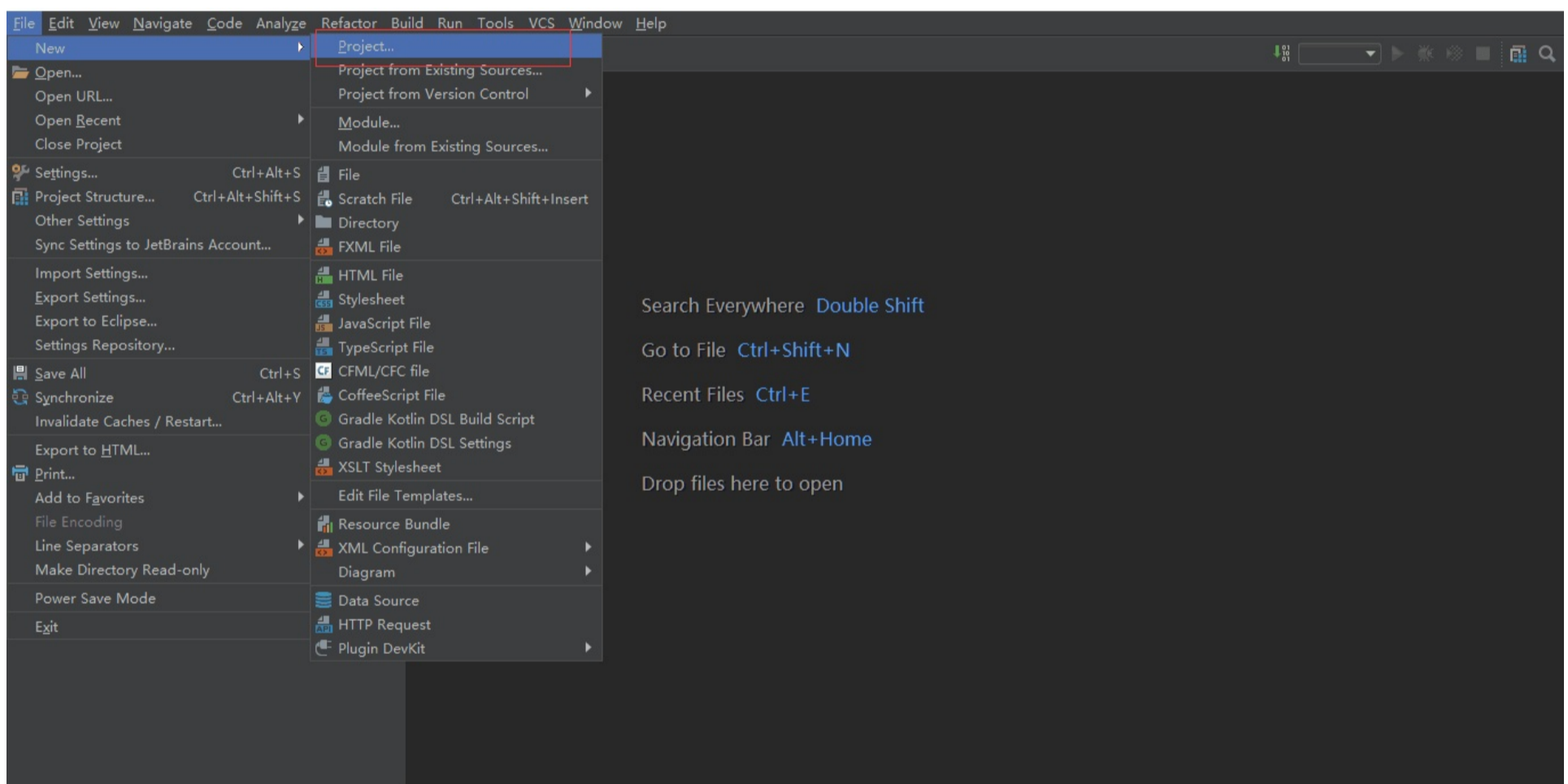
1.9.5 IntelliJ IDEA中的实践

1.9.5.0 前提

1. 安装java开发环境
2. 准备好dble的最新的release版本jar包，以下是2.19.05.0 安装包下载链接：
<https://github.com/actiontech/dble/releases/download/2.19.05.0%2Ftag/actiontech-dble-2.19.05.0.tar.gz>，解压后在lib目录中可以找到dble的对应版本的jar包。

1.9.5.1 创建 java 项目

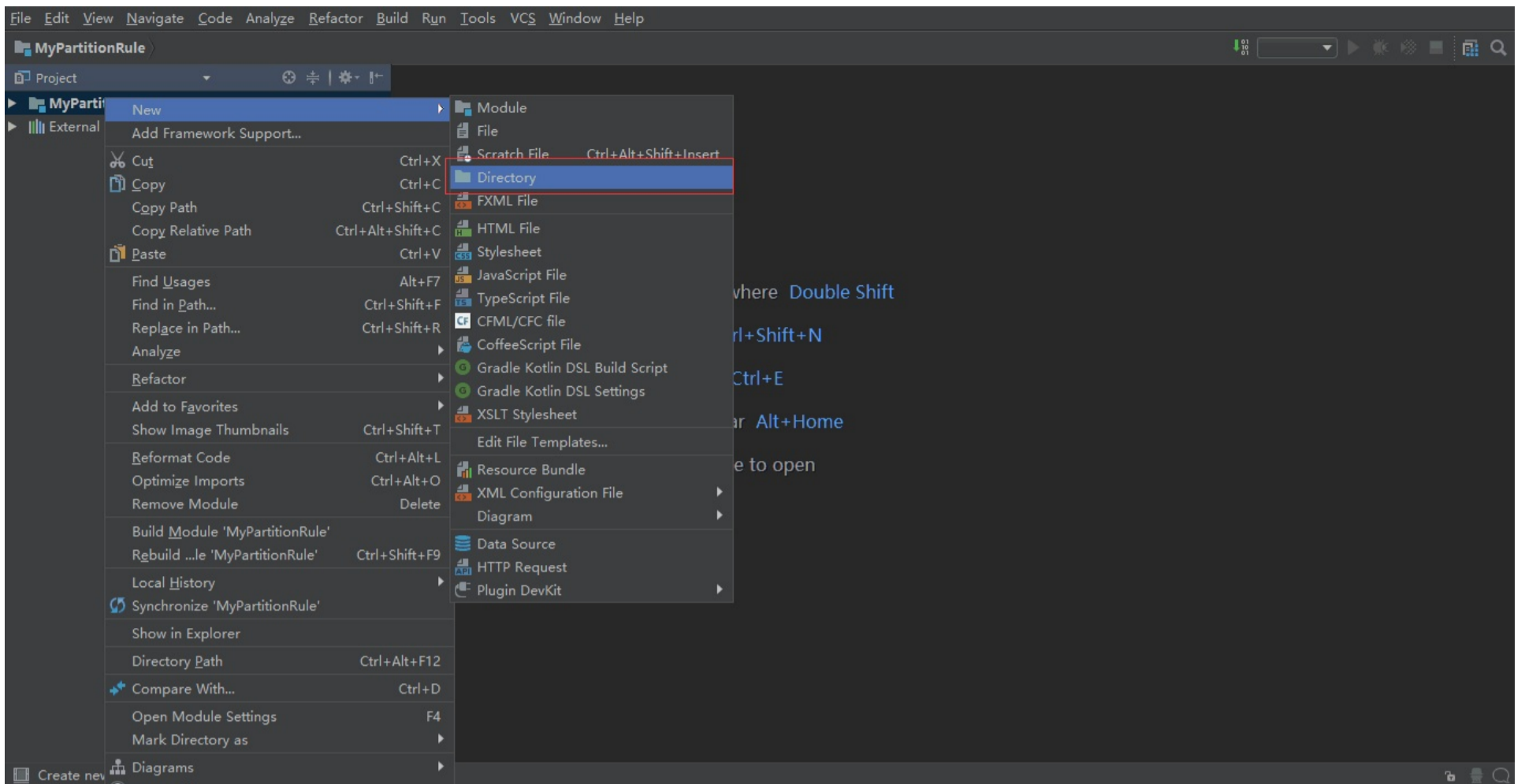
创建简单的java项目，点击红框中Project



点击Project后弹出框中默认选中java，若没有选中，手动选中后一路点击next，填写恰当的项目名即可。

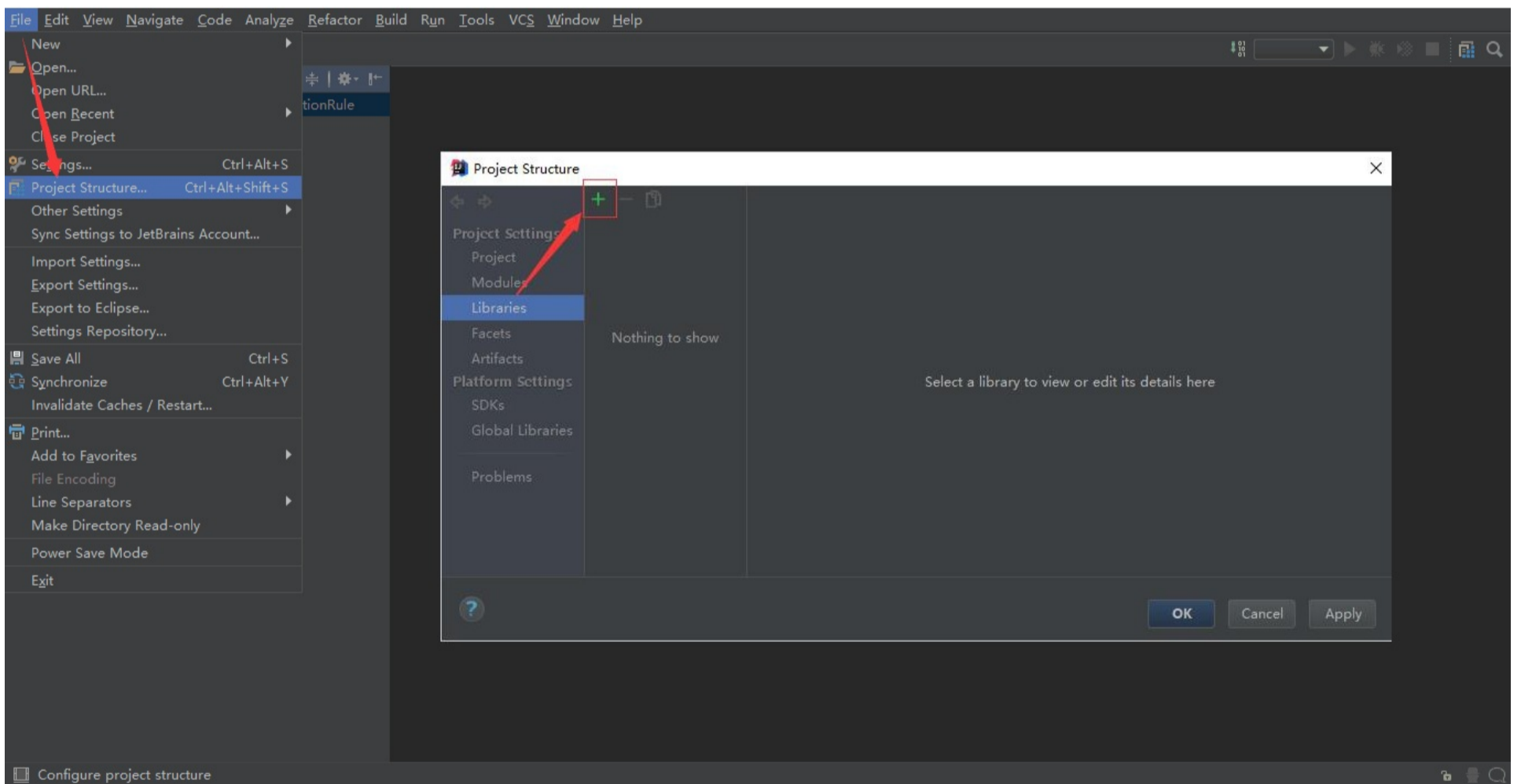
1.9.5.2 引入 dble jar 包

右击新建的java项目，新建lib目录



将 dble 的jar包复制到lib目录下。

将lib中的dble jar 包添加为项目的依赖。



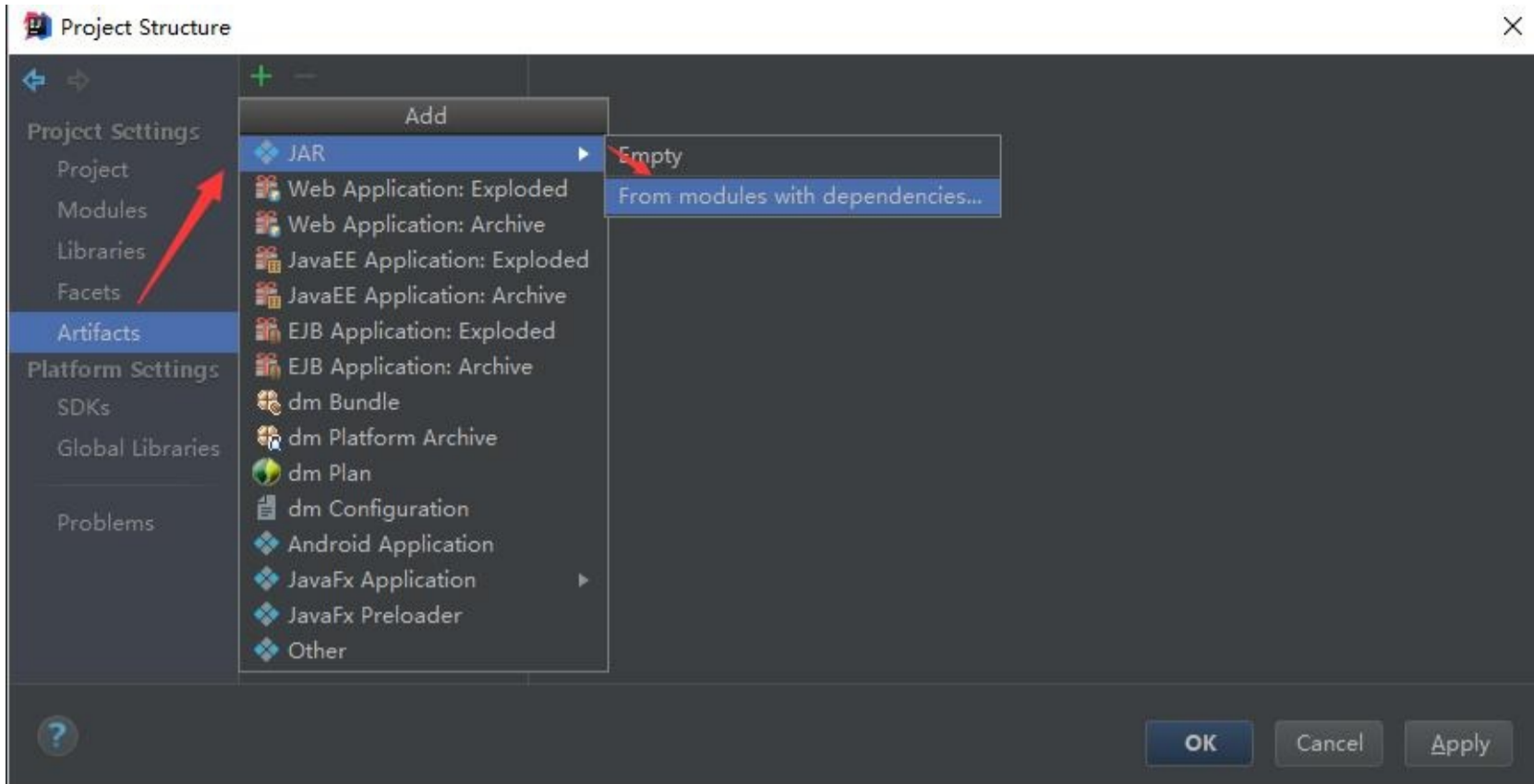
点击加号之后选中文件系统中的lib目录后加入即可

1.9.5.3 自定义分片算法类

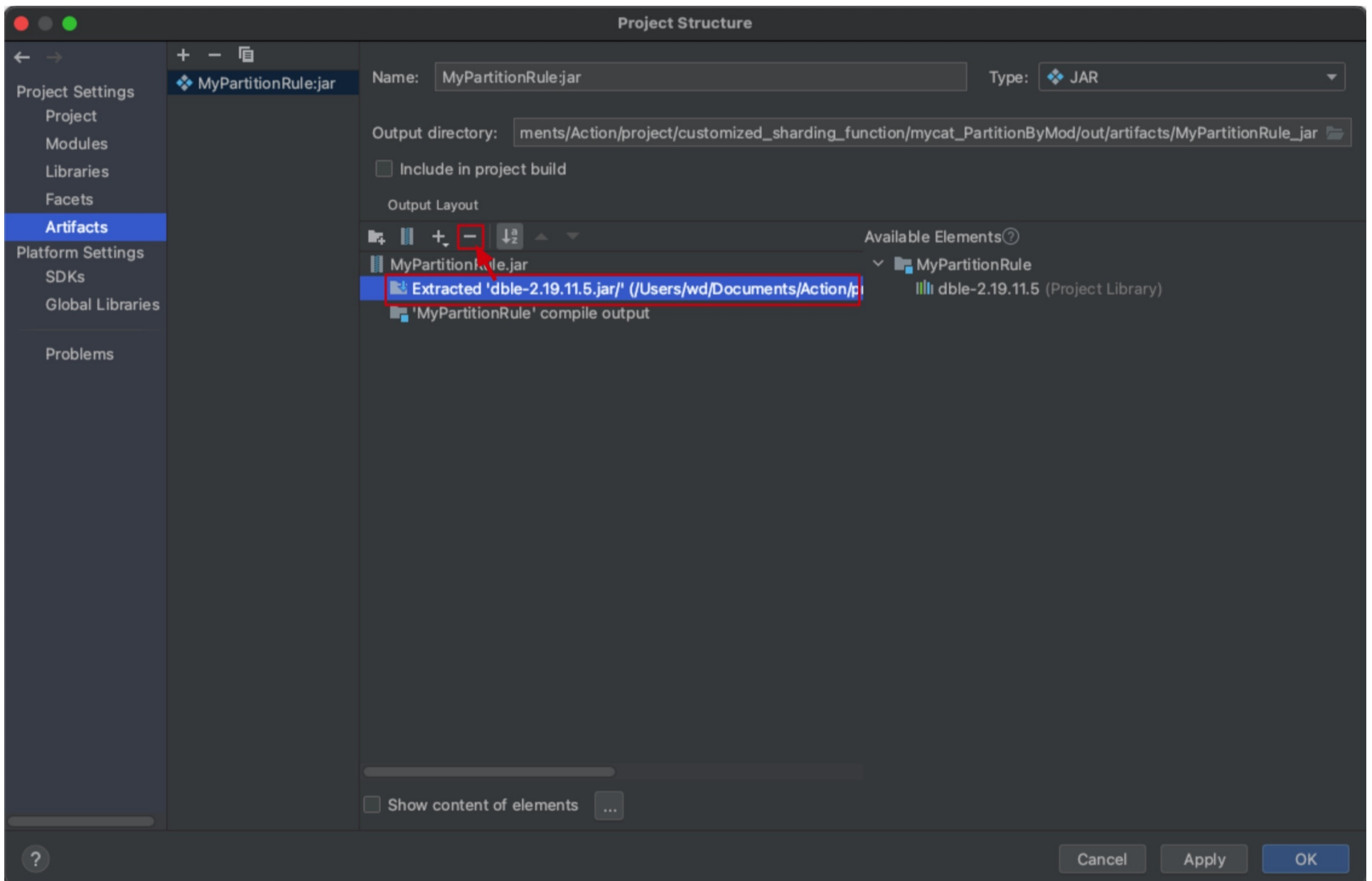
在新建项目中自定义算法类，具体方式请参照上文。

1.9.5.4 新建Artifacts

选中Project Structure后，点击Artifacts，在弹出的对话框中直接点击OK。

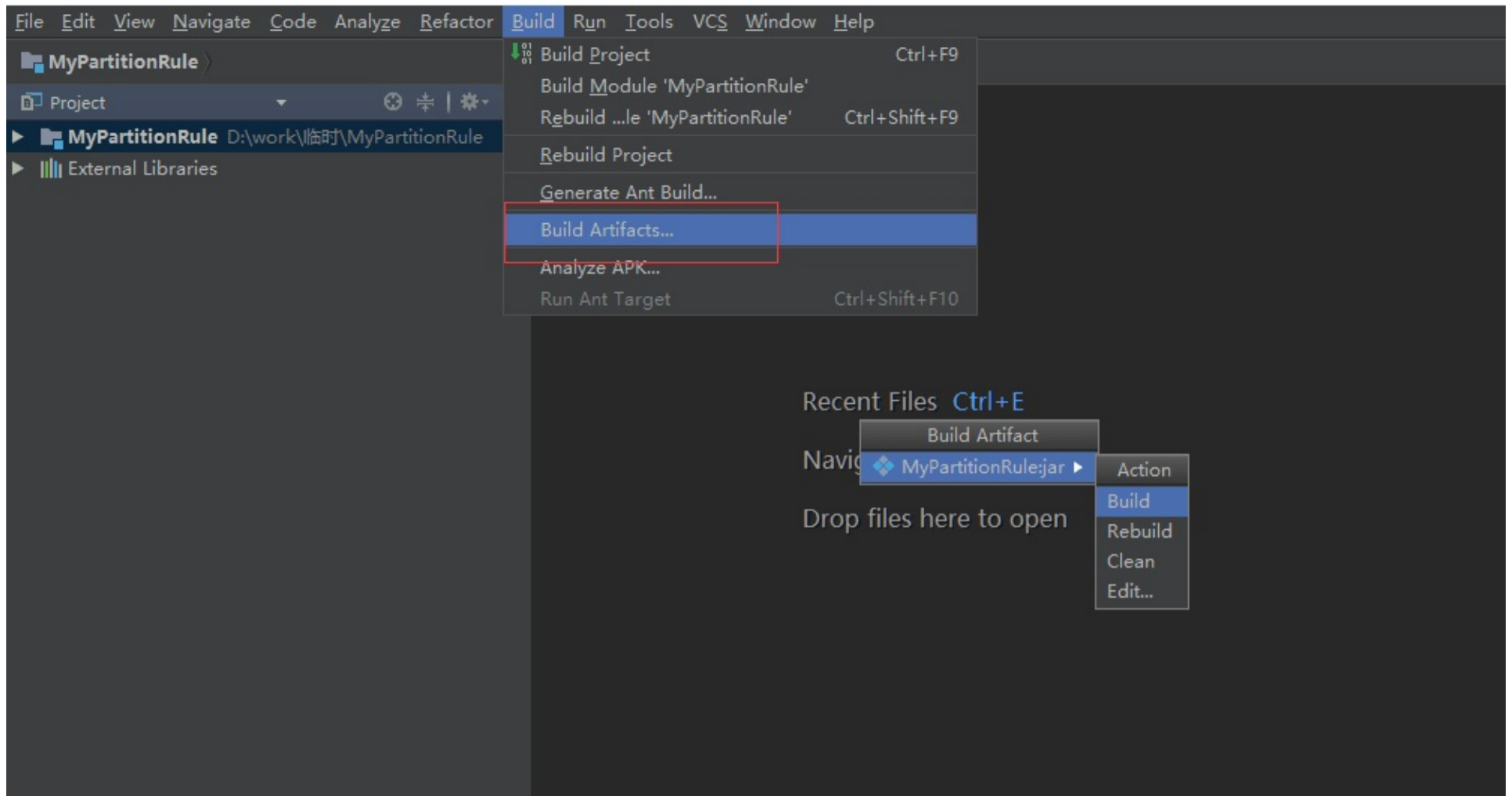


1.9.5.5 注意，请移除Artifacts中dble jar包，目的：防止dble jar参与打包（若没有，则忽略），然后点击Apply/ok



1.9.5.6 编译jar包

选择菜单栏中Build下的Build Artifacts，在弹出的对话框中选中自定义的Artifact后build。



编译完成之后，在项目路径下会生成一个out文件夹，在artifact子文件夹下可以找到生成的jar包。

1.10 版本变更

- 3.23.08.0
 - 配置文件调整
 - 管理端命令调整
 - sql_log表结构调整
- 3.23.04.0
 - 配置文件调整
- 3.22.11.0
 - 配置文件调整
- 3.22.01.0
 - 管理端命令调整
 - dble_db_instance表结构调整
- 3.21.06.0
 - 配置文件版本变更
 - 集群中存储格式调整
 - 日志类管理端命令删除
 - dble_thread_pool表结构调整
- 3.21.02.0
 - 集群中sequence结构调整
 - 配置文件验证方式调整
 - 配置文件版本变更
- 3.20.07.0
 - 配置文件版本变更

3.23.08.0 变更内容

1 配置调整

1.1 bootstrap.cnf

1.1.1 sql统计相关的变动

变更项	变更内容	变更版本	向后兼容性	变更说明
移除	useSqlStat	3.23.08	不兼容	原useSqlStat=0，现在由samplingRate=0代替；原useSqlStat=1，现在由samplingRate=100代替；
移除	bufferUsagePercent	3.23.08	不兼容	原来默认达到内存的80%，则将产生sql统计数据进行处理，达到释放部分内存的目的（原配合useSqlStat=1使用）
移除	clearBigSQLResultSetMapMs	3.23.08	不兼容	原作为定时检测sql统计数据占用内存情况的周期（原配合useSqlStat=1、bufferUsagePercent使用）
移除	sqlRecordCount	3.23.08	不兼容	原作为慢查询show @@sql.slow记录阈值（原配合useSqlStat=1使用）
变更	samplingRate	3.23.08	兼容	修改默认值；原先为0，现在为100；表示开启采样率为100%的sql统计，对应管理端sql_log表
新增	enableStatisticAnalysis	3.23.08	兼容	作为show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition聚合统计的开关，默认不开启

2 管理端命令调整

2.1 sql统计相关show命令

涉及变动的命令：show @@sql、show @@sql.high、show @@sql.slow、show @@sql.large、show @@sql.resultSet、show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition;

移除的命令有：reload @@sqlslow=?、reload @@user_stat;

- 调整之前：

- 以上8个show命令均由 useSqlStat 开关参数控制。（与 samplingRate 参数、管理端sql_log表无关）
- show @@sql、show @@sql.high、show @@sql.slow、show @@sql.large、show @@sql.resultSet 区分用户存储，比如show @@sql，每个用户存储最新的1024条sql；show @@sql.high，每个用户存储近期执行1024条高频的sql。（原保存5N1024条sql统计的数据量，其中N表示用户个数）
- show @@sql、show @@sql.high、show @@sql.slow、show @@sql.large、show @@sql.resultSet 通过在后面追加true的方式清理数据；比如：show @@sql true；或者使用reload @@user_stat命令统一清理这5个命令的全部数据
- show @@sql.slow 由reload @@sqlslow=? 设置符合慢日志的阈值

- 调整之后：

- show @@sql、show @@sql.high、show @@sql.slow、show @@sql.large、show @@sql.resultSet；
 - 这5个命令查询出来的数据来源于管理端sql_log表；sql_log表由 samplingRate 开关参数和 sqlLogTableSize 表大小限制参数控制，因此show @@sql.xx相关命令从此版本开始由这两个参数控制。
 - sqlLogTableSize 参数默认控制管理端sql_log表中只保存最新的1024个事务数据；由于show @@sql.xx相关命令的数据都基于sql_log，因此对比之前版本，show @@sql.xx相关命令的数据不再区分用户存储。（现在仅保存1024个事务数据量）
 - 不支持show @@sql.xx true；统一清理数据的方式为：truncate sql_log
 - show @@sql.slow与慢日志 统一配置和开关；调整为由 sqlSlowTime 参数控制和reload @@slow_query.time=?;
- show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition；
 - 由 enableStatisticAnalysis 开关参数控制，默认关闭，此外可使用动态开关命令enable/disable @@statisticAnalysis；这3个show命令维护各自数据（表示有3份数据，与sql_log无关）。

3 sql_log表结构调整

增加字段result_size：表示结果集的字节大小

3.23.04.0 变更内容

1 配置调整

dble在3.23.04.0版本对读写分离模式的fakeMySQLVersion(in bootstrap.cnf)设置规则进行了调整。

以往规则和分布式一致：填写的版本号不能高于后端Mysql节点的最低版本号，否则启动失败。

调整后（和分布式的规则区分开）：填写的版本号和后端MySQL节点版本号需要在范围上保持一致，否则启动失败。

MySQL官方认为**5.7.20<=版本<8.0.0**和**版本>=8.0.3**是一致的，除此之外其它的版本则认为是同一种情况。例如以下的配置情况：

- fakeMySQLVersion：5.7.20 mysql-version：5.7.25
- fakeMySQLVersion：8.0.3 mysql-version：8.0.23
- fakeMySQLVersion：5.7.15 mysql-version：8.0.1

3.22.11.0 变更内容

1 配置文件调整

1.1 db.xml

变更项	变更内容	变更版本	向后兼容性	变更说明
修改	delayThreshold	3.22.11	兼容	修改参数单位。原先为秒，现在为毫秒

3.22.01.0 变更内容

1 管理端命令调整

管理端的线程池名称做了调整，BusinessExecutor调整为frontWorker，backendBusinessExecutor调整为backendWorker，complexQueryExecutor调整为complexQueryWorker，writeToBackendExecutor调整为writeToBackendWorker

管理端的线程名称BusinessExecutor0替换为0-frontWorker，backendBusinessExecutor0替换为0-backendWorker，writeToBackendExecutor0替换为0-writeToBackendWorker，\$_NIO_REACTOR_FRONT-0替换为0-NIOFrontRW，\$_NIO_REACTOR_BACKEND-0替换为0-NIOBackendRW

调整以下管理端命令：

- show @@threadpool;
- show @@threadpool.task;
- show @@thread_used;

2 dble_db_instance表结构调整

- 增加字段database_type：表示dbInstance的数据库类型

3.21.06.0 变更内容

1 dble的配置文件版本变更

1.1 bootstrap.cnf

变更项	变更内容	变更版本	向后兼容性
新增	<code>inSubQueryTransformToJoin</code> , in子查询转成join进行查询	3.21.06	不兼容
新增	<code>enableCursor</code> , 默认为 <code>false</code>	3.21.06	不兼容

`inSubQueryTransformToJoin`

原来：如果sql中存在in子查询并且满足可以转join的条件，在dble中会默认将in子查询转成join处理

更改后：在默认情况下，不会将in子查询转成join处理，而是正常按照子查询处理。如果需要将in子查询转成join处理，请在bootstrap.cnf增加该参数（`-DinSubQueryTransformToJoin=true`）

具体说明请参考：[in子查询是否转join的说明](#)

`enableCursor`:

原来：老版本是只需要 client 显式开启即可使用游标。

现在：新版本游标功能变成需要client 和 server 同时显式开启才能使用，server 默认不开启。

升级模式：通常不需要这个功能，如果确实需要可以开启。需注意开启会导致所有的prepared statement牺牲一部分性能，用于查询列的数量，详情见【4.4 prepared statement】。

1.2 sharding.xml

变更项	变更内容	变更版本	向后兼容性
修改	<code>jumpStringHash</code> 中的 <code>hashSlice</code> 默认值。	3.21.06	不兼容

原来：如果不设置`hashSlice`，之前版本默认值是（0:-1），不是一个很好的默认值，会丢失一个字符的计算。具体见文档【1.5】中关于 `stringhash` 的介绍

现在：3.21.06改为了默认值（0:0）

升级模式：需检查`jumpStringHash`的`hashSlice`是否有配置,如果原本没有配置需要配置为（0:-1）。

2 集群中存储格式调整

dble在3.21.06.0版本对集群的存储格式进行了调整

原有：配置中心（zk）存储value的格式不尽相同，且不具备向后兼容性

更改后：配置中心（zk）存储value的格式均改成了json 格式。并且在在外层包裹了一个统一的格式。具体格式如下：

```
{
  "instanceName": "1", //bootstrap.cnf 里面的instanceName
  "apiVersion": 1, //为了兼容性增加的版本字段
  "createdAt": 1628669627058, //创建这个节点的时间戳（毫秒）
  "data": { ... } //具体数据以 json 格式存储在这里
}
```

升级方式：由于两者的元数据互不兼容，升级前需停止旧版本 dble，然后删除配置中心中当前集群的所有元数据，即删除目录 `/{rootPath}/{clusterId}`，再启动新版本。如需缩短不可用的时间，新版本可以使用不同的`clusterId`，即启动一个和旧版本隔离的全新集群。

注意：

`rootPath`和`clusterId` 的对应值见 `cluster.cnf`

如果不删除，升级会导致dble 无法启动，并报错 "you may use old incompatible metadata."

降级方式：同升级。

3 日志类管理端命令删除

删除以下管理端命令：

1. `log @@[file=logname limit=numberOfRow key=keyword regex=regex]`
2. `show @@syslog limit=?`
3. `file @@list`
4. `file @@show filename`
5. `file @@upload filename content`

4 dble_thread_pool表结构调整

- 原有字段size修改为pool_size：表示当前实际线程池的大小
- 增加字段core_pool_size：表示设置的核心线程池大小

3.21.02.0 变更内容

1 集群中sequence结构调整

db1e在3.21.02.0版本对序列的存储结构进行了调整

原有：配置中心（zk）存储sequence的结构为key-value形式。key：文件名；value：文件原始内容

更改后：配置中心（zk）存储sequence的结构为key-value形式。key：文件名；value：文件内容的json形式

升级方式：升级前需删除配置中心的sequence配置(rootPath/clusterId/conf/sequences)，存储结构详情见：[ZK整体目录结构](#)

2 dble的配置文件验证方式调整

在3.21.02.0版本对dble的xml配置文件验证方式调整

原有：使用dtd文件验证xml的格式，dble中使用DocumentBuilder的方式解析xml文件

更改后：使用xsd文件验证xml的格式，dble中使用jaxb2.0方式解析xml文件

升级方式：升级前需删除原有xml配置文件中的标签：db.xml中的

```
<!DOCTYPE dble:db SYSTEM "db.dtd">、sharding.xml中的<!DOCTYPE dble:sharding SYSTEM "sharding.dtd">、user.xml中的<!DOCTYPE dble:user SYSTEM "user.dtd">
```

3 dble的配置文件版本变更

3.1 bootstrap.cnf

变更项	变更内容	变更版本	向后兼容性	升级方式
变更	homePath必须显示声明	3.21.02	兼容	升级前需要将原来在bootstrap.cnf中未声明的homePath，加上-DhomePath=。（路径为当前目录，已声明的无需更改）

3.20.07.0 变更内容

1 配置文件版本变更

db1e 在3.20.07.0 版本做了配置的重构。历史变更请参考[2.20.04.0的变更](#)

可以通过升级工具dble_update_config将配置从2.20.04.0 升级到3.20.07.0，如果是更早版本，建议先升级到2.20.04.0

升级工具用法：

```
dble_update_config [-i=read_dir] [-o=write_dir] [-p=rootPath]
```

read_dir/write_dir: 如果不指定，缺省值为当前目录，建议指定或者提前备份配置 rootPath:如果集群模式是zk, 那么缺省值为 /dble , 如果集群模式是ucore, 缺省值为 universe/dble

工具将会读取文件:

myid.properties

wrapper.conf

server.xml

schema.xml

rule.xml

log4j2.xml

cacheservice.properties(option)

sequence_distributed_conf.properties for type3 (option)

sequence_time_conf.properties for type2 (option)

然后写出文件:

cluster.cnf

bootstrap.cnf

user.xml

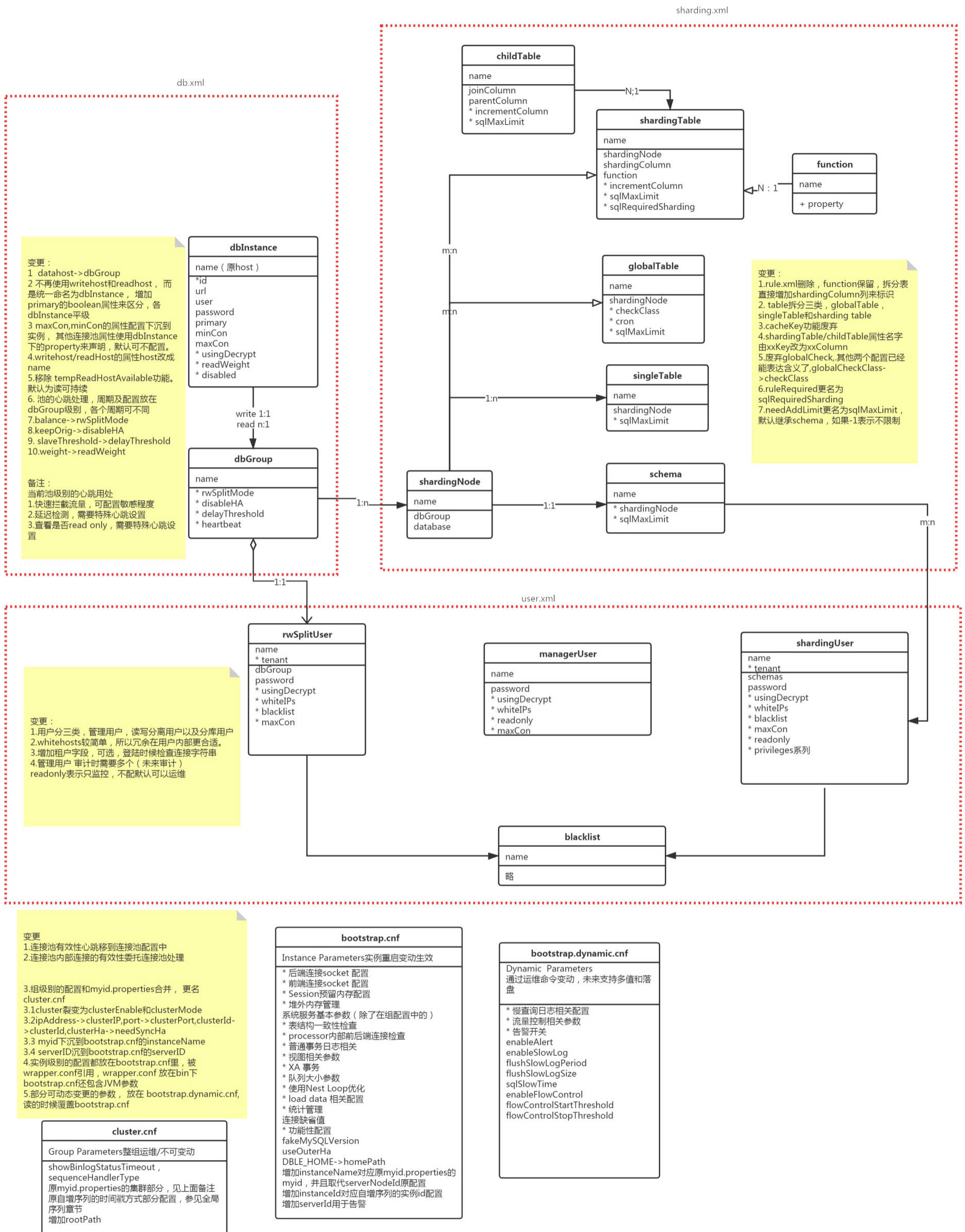
db.xml

sharding.xml

log4j2.xml

cacheservice.properties(option)

重构后的配置概览图：

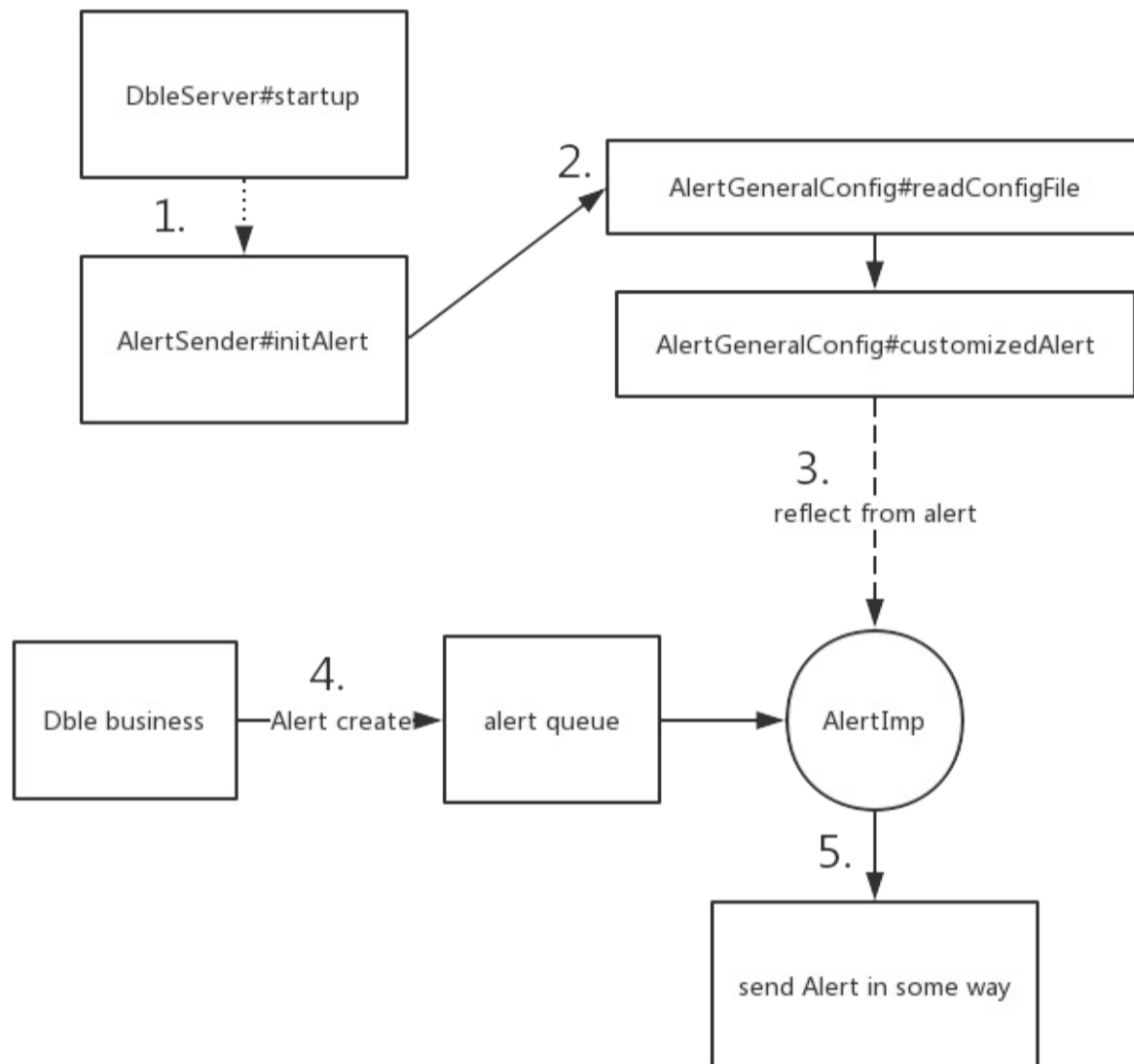


1.11 自定义告警模块

- 告警工作原理
 - 告警模块的加载
 - 告警发送和解除
 - 告警Interface Alert详解
 - 告警发送对象详解
- 自定义告警的开发和部署
 - 自定义告警的开发
 - 自定义告警的使用
 - 自定义告警示例以及示例文件
- 第三方自定义告警项目示例

1.11.1 工作原理

1.11.1.1 告警模块的加载



告警模块加载仅在dble启动的时候进行

- dble启动，通过一系列加载之后，开始加载告警模块
- 由对象AlertGeneralConfig读取告警配置文件dble_alert.properties
- 根据读取到配置文件中的alert配置项进行对象反射，获取到用户自定义对象的实例
- 当dble运行过程中发生告警或者告警解除事件，将告警任务放入任务队列中
- 由一个dble线程循环消费队列中的内容，调用用户自定义对象对于告警任务进行发送

1.11.1.2 告警发送和解除

在dble的告警模块中，将系统中的告警分成以下两种类型

- 可自动解决告警
- 不可自动解决告警

其中“可自动解决告警”为dble在运行过程中遭遇的偶发错误导致的告警，当错误被修正或者重试成功时告警被解除。例如：后端连接数到达最大值，XA事务提交单次失败等

而“不可自动解决告警”则为dble不会重试也不能解决的错误，譬如：启动无法读取xa日志记录文件等

同时在dble中还将告警分成以下两种类型

- dble内部发生的错误告警

- dble和外部交互发生的错误告警

其中“dble内部发生的告警”指的是在dble对于内部任务在执行过程中发生的异常或者其他现象，比如写文件失败，kill后端连接失败等等 而“dble和外部交互发生的错误告警”则指的是dble和外部具体节点交互时候的错误，譬如无法连接某后端节点，或者某后端节点心跳失败等等

根据以上两种分类，在dble的告警系统中存在着以下的四种告警信息

- 可自动解决-dble内部告警
- 不可自动解决-dble内部告警
- 可自动解决-dble与外部交互告警
- 不可自动解决-dble与外部交互告警

针对以上的四种告警的类别，我们可以发现，在dble的告警周期中，需要有以下四个逻辑上的方法：

- dble与外部交互告警发送方法
- dble内部告警发送方法
- dble与外部交互告警解除发送方法
- dble内部告警解除发送方法

1.11.1.3 告警Interface Alert详解

```
public interface Alert {
    void alertSelf(ClusterAlertBean bean);
    void alert(ClusterAlertBean bean);
    boolean alertResolve(ClusterAlertBean bean);
    boolean alertSelfResolve(ClusterAlertBean bean);
    void alertConfigCheck() throws Exception;
}
```

以上是整个告警接口的定义，可以看到除了一个比较特殊的方法alertConfigCheck（检查告警配置）其他的几个方法基本可以说是成对出现

- alert告警发送 --- dble与外部交互告警
- alertSelf告警发送 --- dble内部告警发送
- alertResolve告警解决发送 --- dble与外部告警解除发送
- alertSelfResolve内部告警解除发送 --- dble内部告警解除发送

四个信息发送的方法分别对应到上文中对于几个dble告警系统中的方法需求

并在此解释下alert和alertSelf方法的区别：alert方法的调用输入中为ClusterAlertBean设定了alertComponentType和alertComponentId两个字段，而alertSelf方法则没有，设定适当的DBLEserver标识的动作交给Alert对象来完成 当然在alertResolve和alertSelfResolve之间的区别也是同样的

1.11.1.4 告警发送对象详解

```
public class ClusterAlertBean {
    String code;           //告警的具体代码
    String level;         //告警发生的具体级别
    String desc;          //告警发送的详细描述
    String sourceComponentType; //告警发生的问题源头组件类型
    String sourceComponentId; //告警发送的问题源头组件ID
    String alertComponentType; //具体发送告警的组件类型
    String alertComponentId; //具体发送告警的组件ID
    String serverId;      //告警发生的服务器ID
    long timestampUnix;   //告警发送的时候戳
    long resolveTimestampUnix; //告警解除的时间，仅有解除的告警拥有此字段的值
    Map<String, String> labels; //告警的额外附加信息，补充信息
}
```

以上是在dble中对于告警对象的定义，每个字段的含义都在注释中进行了说明，在自定义告警发送的时候只需要对于告警对象中用户所关心的信息进行发送和处理即可

1.11.2 自定义告警的开发和部署

1.11.2.1 自定义告警的开发

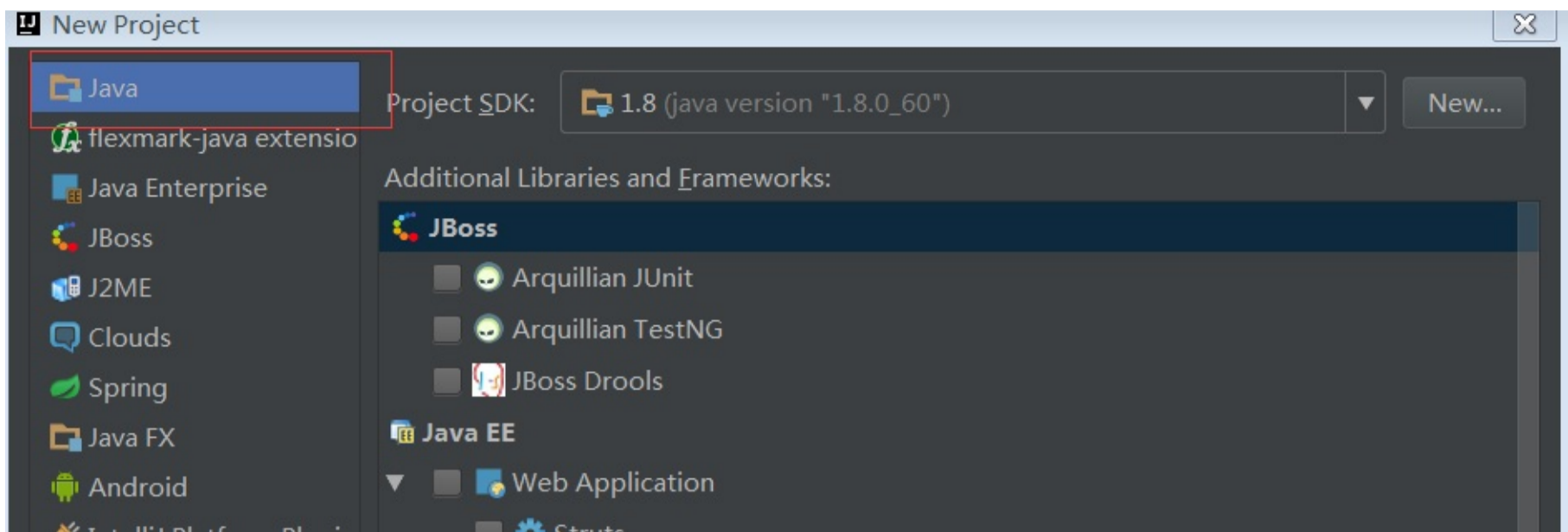
从上文中对于整个dble中告警系统的介绍，可以大致的将dble中的告警进行如下的总结：

- dble会在启动的时候根据配置文件中的信息加载一个Alert的对象
- dble的告警由dble内部的代码进行固定的触发，并将告警的任务发送到一个指定的告警队列中
- dble内部会有一个线程循环调用Alert的实现对象对于告警进行发送
- dble中的告警发送分为四种情况“内部告警”“外部告警”“内部告警解决”“外部告警解决”

按照上文中对于dble内部告警机制和原理解释，可以整理出对于自定义告警的基本开发过程 -----创建一个Alert接口的实现，并且按照需要的方式将告警的信息发送到指定的地方

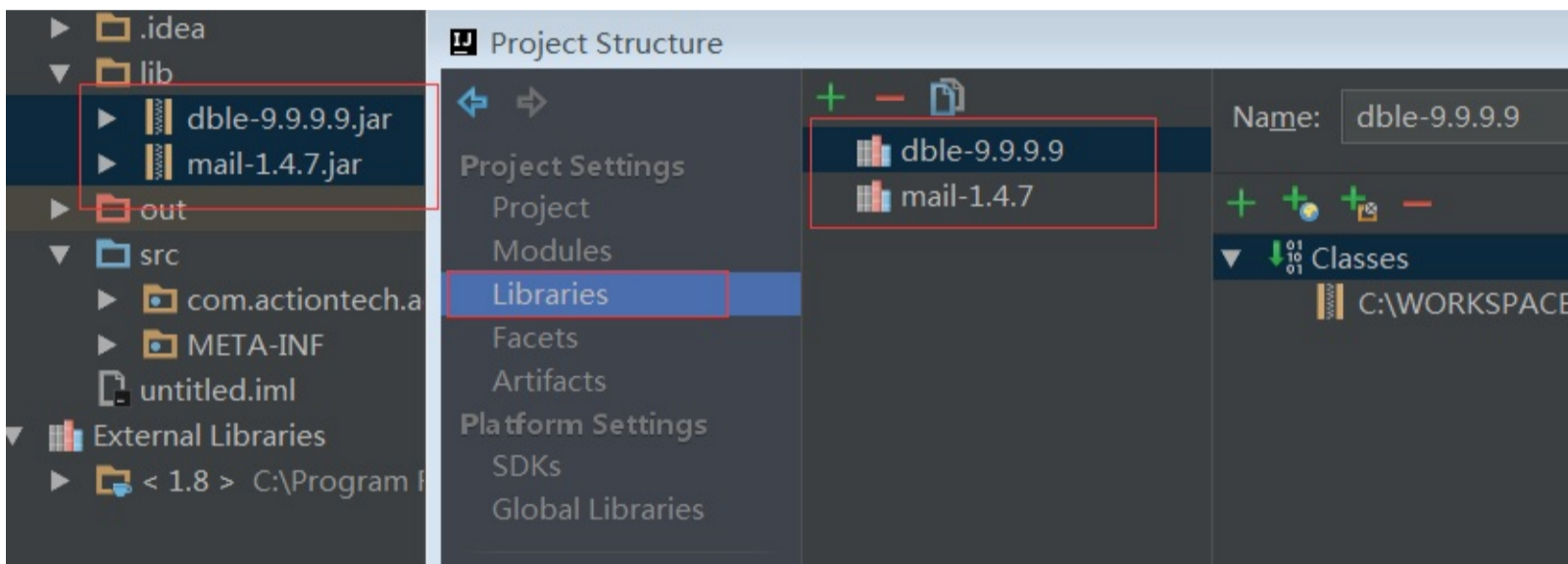
为了达成这个目的，可以按照以下的步骤进行逐一的开发，以下提供一个基于IDEA逐步操作的图文步骤，以一个通过邮件发送告警信息为例

1 创建一个java项目



最简单的java项目就能完成这个告警自定义的开发

2 将需要的依赖包copy到项目中，并且添加到lib



在此例中由于仅需额外依赖一个java的邮件发送包mail，所以这里仅导入了两个jar包作为依赖

3.1 编写Alert实现类初始化方法

```
public MailAlert() {
    //init the mail data and read config file
    properties = AlertGeneralConfig.getInstance().getProperties();
}
```

在Alert实现类MailAlert中，可以通过调用AlertGeneralConfig.getInstance().getProperties();获取到dble启动时候加载到的告警配置文件(dble_alert.properties)中的内容 所以，在这个地方如果自定义的告警需要使用到各种参数，都可以直接写在文件dble_alert.properties的配置中，而在Alert实现类里面按照key值进行获取即可

3.2 编写Alert的配置检查

```
@Override
public void alertConfigCheck() throws ConfigException {
    //check if the config is correct
    if (properties.getProperty(MAIL_SENDER) == null
        || properties.getProperty(SENDER_PASSWORD) == null
        || properties.getProperty(MAIL_SERVER) == null
        || properties.getProperty(MAIL_RECEIVE) == null) {
        throw new ConfigException("alert check error, for some config is missing");
    }
}
```

在此例中由于是基于邮件进行告警发送，所以启动的时候需要在dble_alert.properties文件中配置MAIL_SENDER，SENDER_PASSWORD，MAIL_SERVER，MAIL_RECEIVE四个选项 当配置不能满足自定义的这些选项的时候则抛出异常，dble启动加载的时候会打印对应的异常日志，并且切换为内置默认告警启动上文中

```
properties.getProperty(MAIL_RECEIVE)
```

的这种方式则是从配置中获取自定义配置项的方法

3.3 编写发送告警主体方法

由于此例中发送告警统一为邮件发送，包括告警和告警解决仅存在邮件内容中的差异，所以在这里仅用一个邮件发送方法send进行实现

```

private boolean sendMail(boolean isResolve, ClusterAlertBean clusterAlertBean) {
    try {
        Properties props = new Properties();
        props.setProperty("mail.debug", "true");
        props.setProperty("mail.smtp.auth", "true");
        props.setProperty("mail.host", properties.getProperty(MAIL_SERVER));
        props.setProperty("mail.transport.protocol", "smtp");

        MailSSLSocketFactory sf = new MailSSLSocketFactory();
        sf.setTrustAllHosts(true);
        props.put("mail.smtp.ssl.enable", "true");
        props.put("mail.smtp.ssl.socketFactory", sf);

        Session session = Session.getInstance(props);

        Message msg = new MimeMessage(session);
        msg.setSubject("DBLE告警 " + (isResolve ? "RESOLVE\n" : "ALERT\n"));
        StringBuilder builder = new StringBuilder();
        builder.append(groupMailMsg(clusterAlertBean, isResolve));
        msg.setText(builder.toString());
        msg.setFrom(new InternetAddress(properties.getProperty(MAIL_SENDER)));

        Transport transport = session.getTransport();
        transport.connect(properties.getProperty(MAIL_SERVER), properties.getProperty(MAIL_SENDER), properties.getProperty(SENDER_PASS
WORD));

        transport.sendMessage(msg, new Address[]{new InternetAddress(properties.getProperty(MAIL_RECEIVE))});
        transport.close();
        //send EMAIL SUCCESS return TRUE
        return true;
    } catch (Exception e) {
        e.printStackTrace();
    }
    //send fail reutrnrn false
    return false;
}

private String groupMailMsg(ClusterAlertBean clusterAlertBean, boolean isResolve) {
    StringBuffer sb = new StringBuffer("Alert mail:\n");
    sb.append("    Alert type:" + clusterAlertBean.getCode() + " " + (isResolve ? "RESOLVE\n" : "ALERT\n"));
    sb.append("    Alert message:" + clusterAlertBean.getDesc() + "\n");
    sb.append("    Alert component:" + clusterAlertBean.getAlertComponentType() + "\n");
    sb.append("    Alert componentID:" + clusterAlertBean.getAlertComponentId() + "\n");
    sb.append("    Alert source:" + clusterAlertBean.getAlertComponentId() + "\n");
    sb.append("    Alert server:" + clusterAlertBean.getServerId() + "\n");
    sb.append("    Alert time:" + TimeStamp2Date(clusterAlertBean.getTimestampUnix()) + "\n");
    String detail = "|";
    if (clusterAlertBean.getLabels() != null) {
        for (Map.Entry<String, String> entry : clusterAlertBean.getLabels().entrySet()) {
            detail += entry.getKey() + ":" + entry.getValue();
        }
    }
    sb.append("    Other detail:" + detail + "\n");
    return sb.toString();
}

```

具体如何使用java发送邮件我在此不进行赘述了，大概上面这段代码的原理就是从配置properties获取邮件配置的项，然后发送邮件到指定邮箱即可 而具体的groupMailMsg就是将告警发送过来的对象clusterAlertBean中的数据组织成邮件的文本内容，完全可以根据实际需要进行组织

3.4 实现具体的几个告警发送的方法


```

@Override
public void alertSelf(ClusterAlertBean clusterAlertBean) {
    alert(clusterAlertBean.setAlertComponentType(COMPONNENT_TYPE).setAlertComponentId(properties.getProperty(COMPONNENT_ID)));
}

@Override
public void alert(ClusterAlertBean clusterAlertBean) {
    clusterAlertBean.setSourceComponentType(COMPONNENT_TYPE).
        setSourceComponentId(properties.getProperty(COMPONNENT_ID)).
        setServerId(properties.getProperty(SERVER_ID)).
        setTimestampUnix(System.currentTimeMillis() * 1000000);
    sendMail(false, clusterAlertBean);
}

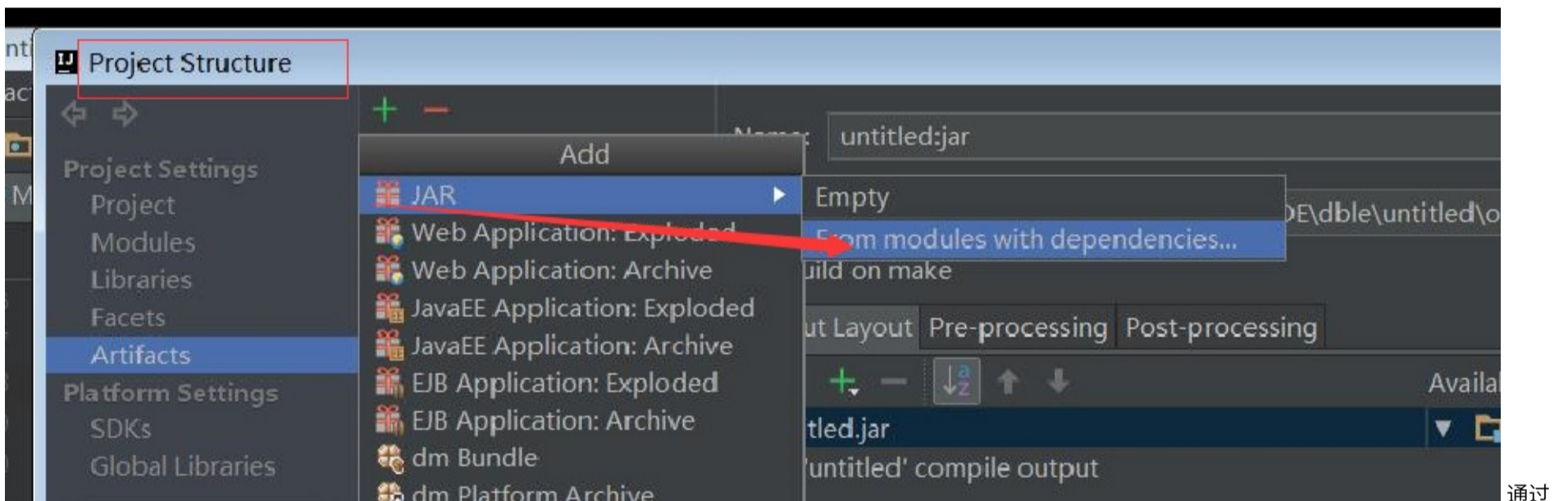
@Override
public boolean alertResolve(ClusterAlertBean clusterAlertBean) {
    clusterAlertBean.setSourceComponentType(COMPONNENT_TYPE).
        setSourceComponentId(properties.getProperty(COMPONNENT_ID)).
        setServerId(properties.getProperty(SERVER_ID)).
        setTimestampUnix(System.currentTimeMillis() * 1000000);
    return sendMail(true, clusterAlertBean);
}

@Override
public boolean alertSelfResolve(ClusterAlertBean clusterAlertBean) {
    return alertResolve(clusterAlertBean.setAlertComponentType(COMPONNENT_TYPE).setAlertComponentId(properties.getProperty(COMPONNENT_ID)));
}

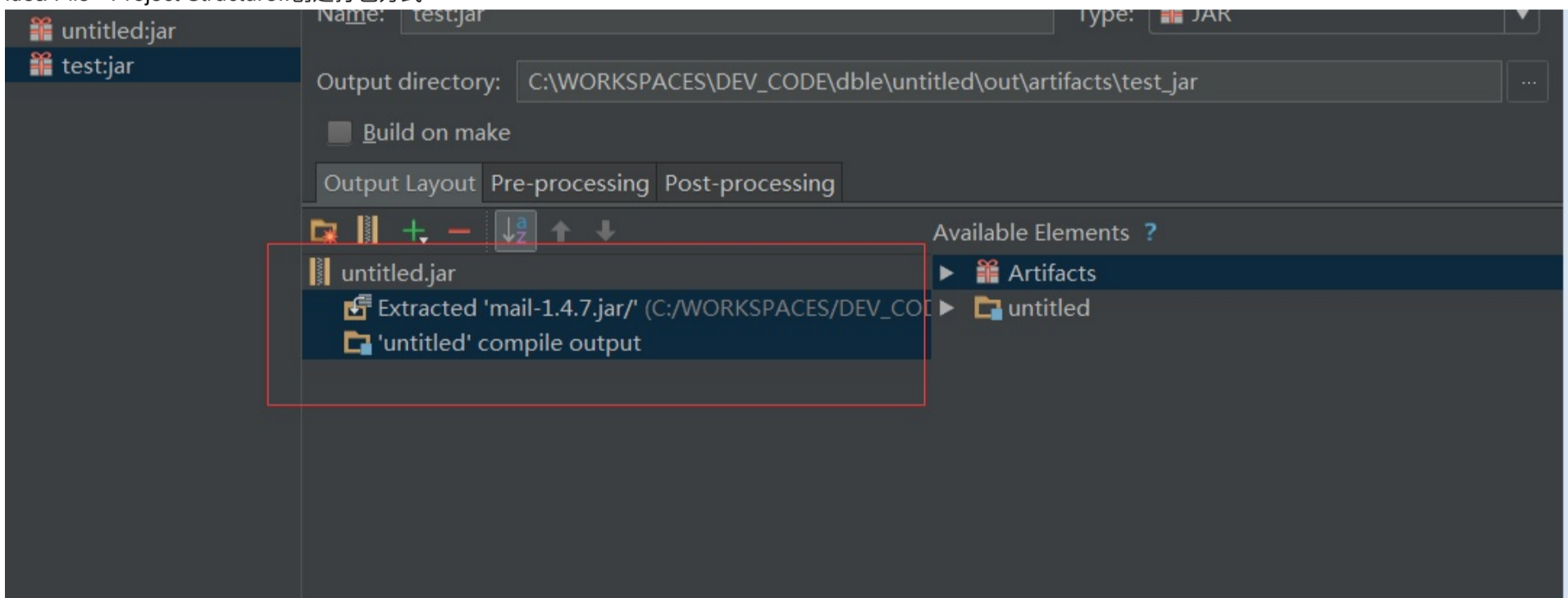
```

以上方法实现就特别简单了，一般情况下都可以按照以上的示例进行实现，基本上就是补全部分缺失的信息，调用发送的方法即可

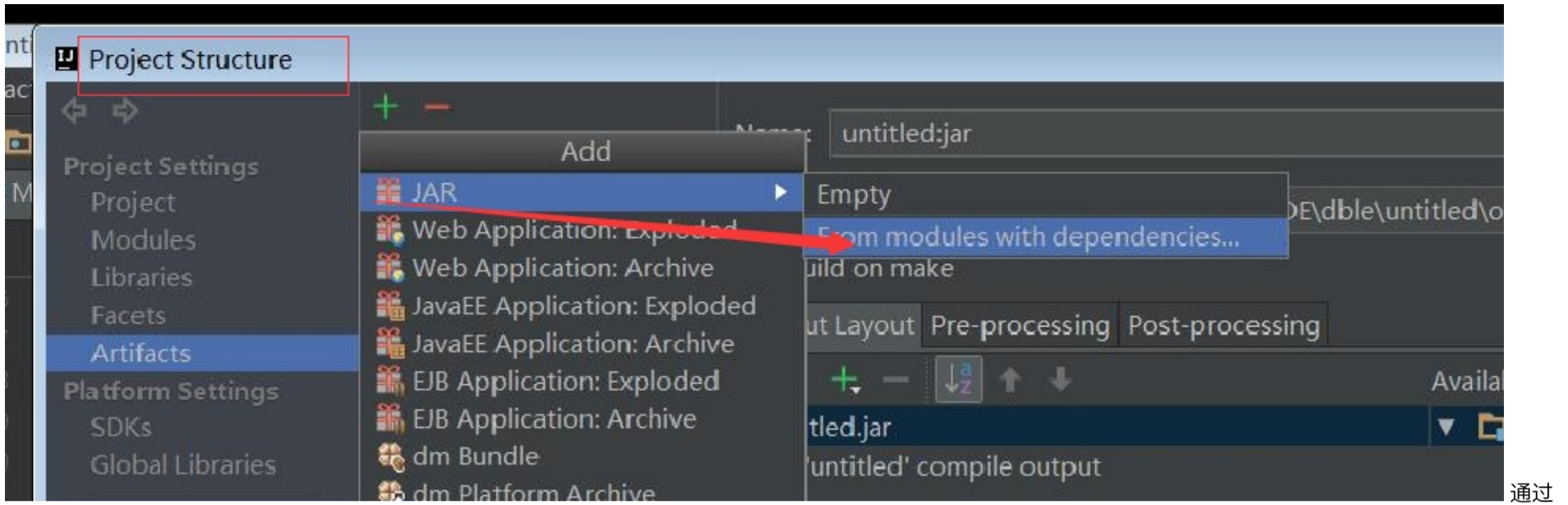
3.5 打包成jar进行使用



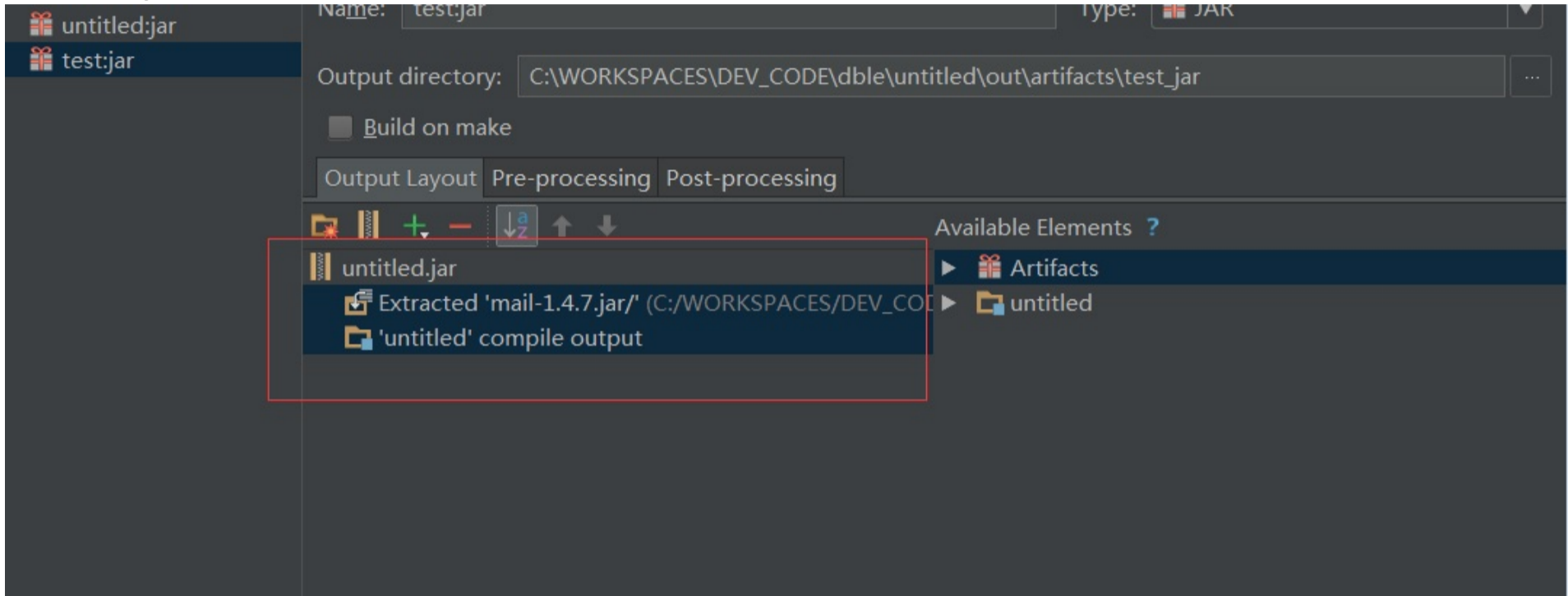
idea File->Project Structure..创建打包方式



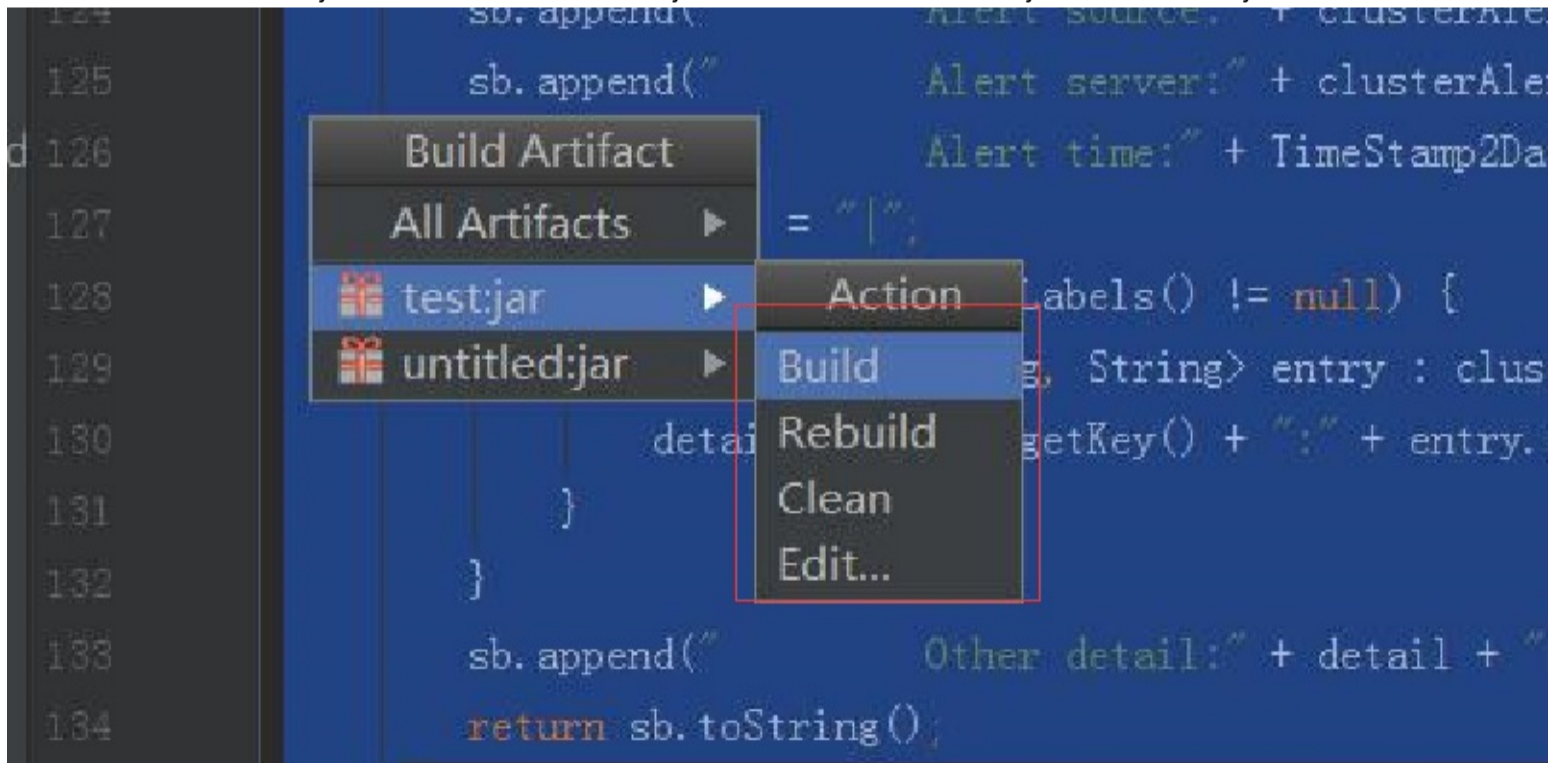
确认打包的内容，注意的点是不要将double的jar包也当作依赖打包到项目的jar包中去，当自定义中使用的jar包和double的依赖jar包发生重复的时候，也请不要把重复



idea File->Project Structure..创建打包方式



确认打包的内容，注意的点是不要将db的jar包也当作依赖打包到项目的jar包中去，当自定义中使用的jar包和db的依赖jar包发生重复的时候，也请不要把重复



的依赖打包进去

通过IDEA的

Build->Build Artifacts..对于项目进行打包，能在项目新生成的out文件夹下找到对应的jar包

1.11.2.2 自定义告警的使用

有了上面打包出来的jar包之后就是该如何进行使用了，大致的步骤只有三个

- 将打包完成的jar包放入db安装目录的lib目录下
- 配置文件db_alert.properties使得自定义的文件能被加载到,将类名配置到alert配置项中去
- 重启db使得加载生效 在这里同样是衔接上文的案例，文件复制到lib下在此进行跳过 之后修改db_alert.properties配置文件,大致如下图所示

```

alert=com.actiontech.addtionAlert.MailAlert
mail_sender=123456798@qq.com
sender_pass=qwertyuiop
mail_server=smtp.qq.com
mail_receive=yyyyyyyyyyyy@actionsky.com
server_id=dble-server-001
component_id=DBLE-FOR-XXX-01

```

告警代码	解释	告警分类
DBLE_WRITE_TEMP_RESULT_FAIL	写出中间结果集到文件失败	内部/非自解决
DBLE_XA_RECOVER_FAIL	XA事务恢复失败	内部/非自解决
XA_READ_XA_STREAM_FAIL	读取XA事务记录文件失败	内部/非自解决
DBLE_XA_READ_DECODE_FAIL	解析XA事务记录文件失败	内部/非自解决
DBLE_XA_READ_IO_FAIL	读取XA事务记录文件失败	内部/非自解决
DBLE_XA_WRITE_IO_FAIL	写XA事务记录文件失败	内部/非自解决
DBLE_XA_WRITE_CHECK_POINT_FAIL	XA写出检查点信息失败	内部/自解决
DBLE_XA_BACKGROUND_RETRY_FAIL	XA后台重试提交失败	内部/自解决
DBLE_REACH_MAX_CON	节点连接数到达配置最大值，获取连接失败	内部/自解决
DBLE_TABLE_NOT_CONSISTENT_IN_SHARDINGS	表结构在不同数据节点中不一致	内部/自解决
DBLE_TABLE_NOT_CONSISTENT_IN_MEMORY	表结构在数据节点中和dbLe内存中不一致	内部/自解决
DBLE_GLOBAL_TABLE_NOT_CONSISTENT	全局表结构在不同数据节点中不一致	内部/自解决
DBLE_CREATE_CONN_FAIL	创建连接到后端mysql的连接失败	外部/自解决
DBLE_DB_INSTANCE_CAN_NOT_REACH	后端连接创建不可达	外部/非自解决
DBLE_KILL_BACKEND_CONN_FAIL	Kill后端连接执行失败	内部/非自解决
DBLE_NIOREACTOR_UNKNOWN_EXCEPTION	NIO意外报错	内部/非自解决
DBLE_NIOREACTOR_UNKNOWN_THROWABLE	NIO意外严重错误	内部/非自解决
DBLE_NIOCONNECTOR_UNKNOWN_EXCEPTION	NIO后端连接创建器意外错误	内部/非自解决
DBLE_TABLE_LACK	配置中的表格未被创建	内部/自解决
DBLE_GET_TABLE_META_FAIL	获取表格的创建语句失败	内部/自解决
DBLE_TEST_CONN_FAIL	测试后端连接可达性失败	内部/非自解决
DBLE_HEARTBEAT_FAIL	心跳后端节点失败	外部/自解决
DBLE_SHARDING_NODE_LACK	缺少可用的shardingNode节点	外部/自解决
DBLE_AP_NODE_LACK	缺少可用的apNode节点	外部/自解决
DBLE_XA_SUSPECTED_RESIDUE	疑似Xaid残留	内部/非自解决
DBLE_DB_SLAVE_INSTANCE_DELAY	主从延迟超高delayThreshold的值	内部/自解决
DBLE_XA_BACKGROUND_RETRY_STOP	XA重试机制中如果配置了xaRetryCount，重试次数到达该值时，重试停止	内部/非自解决
SLOW_QUERY_QUEUE_POLICY_ABORT	slowQueueOverflowPolicy为1时，因队列满了而丢弃慢日志记录	内部/非自解决
SLOW_QUERY_QUEUE_POLICY_WAIT	slowQueueOverflowPolicy为2时，虽然队列满了，但触发了阻塞机制，最终慢日志没有丢失，落盘成功	内部/非自解决
DBLE_THREAD_SUSPECTED_HANG	检测到Timer或者TimerScheduler线程出现疑似hang	内部/自解决

自定义全局表检查

- [全局表一致性检查逻辑](#)
- [全局表自定义方法详述](#)
 - [getCountSQL](#)
 - [getFetchCols](#)
 - [resultEquals](#)
 - [failResponse](#)
 - [resultResponse](#)
- [自定义全局表检查操作步骤](#)
- [自定义全局表检查配置](#)

背景

全局表是dble中一种特殊类型的表格，一般来说认为在一个全局表table_a所有分布的节点上，table_a因同时满足以下两个条件：

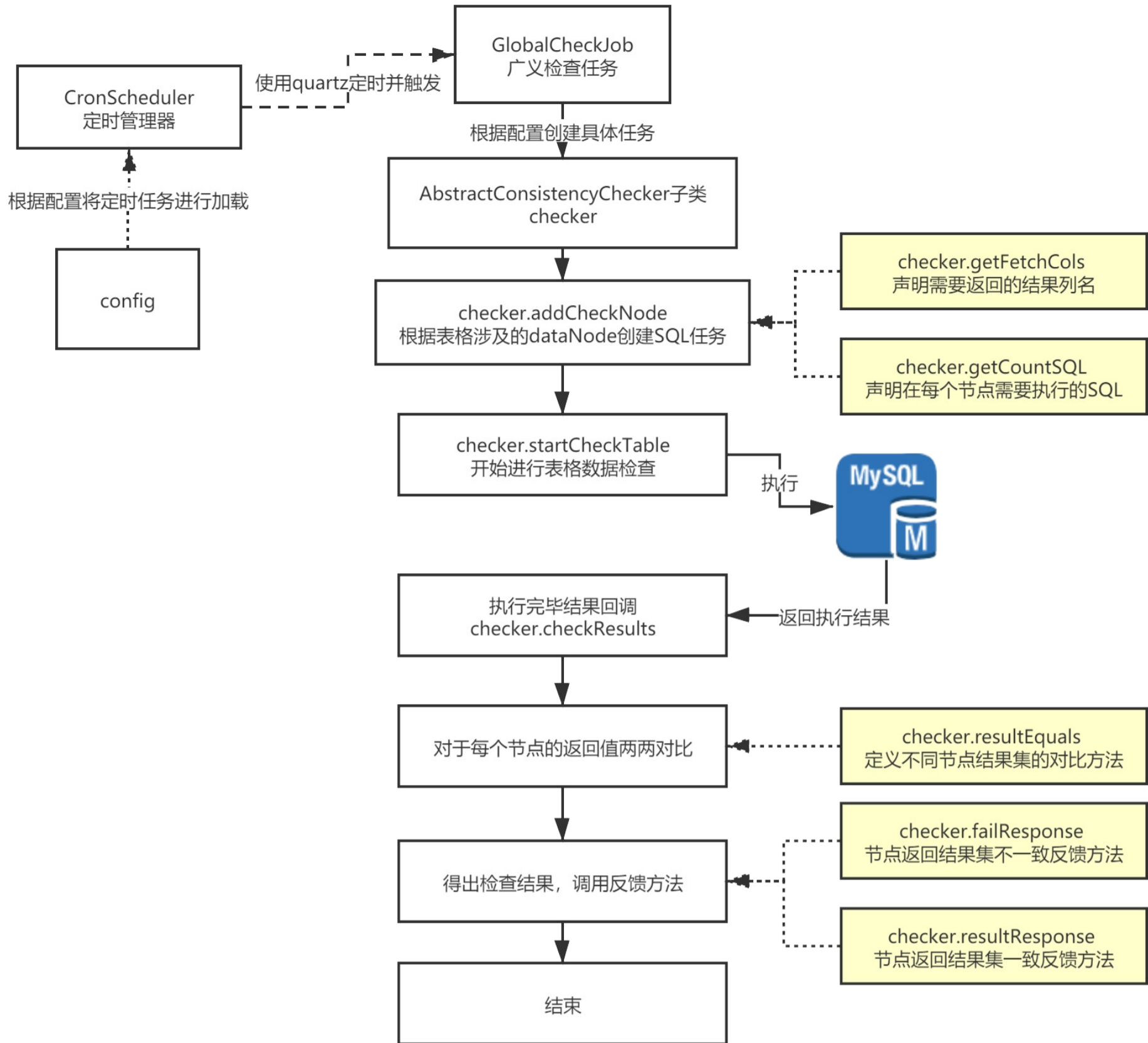
- 拥有相同的表格结构
- 拥有相同的表格数据

但事实上在系统和dble的运行过程中，可能由于一些不可避免分布式事务方面的误差，导致在长时间运行之后，不同节点上面的table_a上面的数据不一致。为了及时的发现问题并不再造成更进一步的错误，dble中采用定时进行表格数据检查的方式对于table_a中的数据一致性进行检查，并及时把检查的结果通知到运维人员。

工作原理大致逻辑

全局表检查工作逻辑

全局表检查的大致逻辑如下图所示：



上图中着色的部分允许接收用户的自定义，在下一小节中会对于每个步骤进行详细的说明

整体上来说全局表检查的工作原理分为以下几个步骤：

- 加载表格检查配置，在启动或者是reload阶段将配置加载到定时任务管理器CronScheduler中
- 当根据配置的触发条件正常触发时执行GlobalCheckJob开始任务
- 在GlobalCheckJob中根据表格配置计算表格配置并创建具体的SQL检查任务
 - 此时会调用checker中的方法返回执行的具体SQL语句以及需要取得的结果列名
 - 按照shardingNode的结构将SQL任务提前构造完毕
- 触发SQL执行，逐个执行构造完成的SQL任务，下发SQL到MySQL进行执行
- 等待所有的SQL执行结果都返回(成功或者失败)
- SQL执行结果返回，回调方法checkResults进行结果集检查
- 根据checker中的结果集比较方法对于SQL执行的结果进行比较
- 调用回调方法进行结果反馈，当SQL执行结果有超过一个版本(存在不一致)时调用失败接口failResponse，当SQL执行结果只有一个版本(所有正常返回的结果一致)时调用resultResponse方法

全局表检查方法详解

1. 执行SQL定义String getCountSQL(String dbName, String tName)

功能： 返回全局表检查需要对于表格执行的SQL内容

输入： SQL执行的MySQL中database的名称，所检查的表格的名称

输出： 检查具体需要执行的SQL

举例：

```
public String getCountSQL(String dbName, String tName) {
    //假如需要对于对应的table名字求checksum
    return "checksum table " + tName;
}
```

2.结果集定义 getFetchCols()

功能： 返回SQL执行完毕需要使用的结果集中的列名

输入： 无

输出： 需要收集的列名list

举例：

```
public String[] getFetchCols() {
    //checksum返回结果,我们只关心Checksum字段的返回值
    //mysql> checksum table suntest;
    //+-----+-----+
    //| Table      | Checksum |
    //+-----+-----+
    //| db1.suntest |1290812451|
    //+-----+-----+
    //所以return的内容只需要一个Checksum的列名即可
    return new String[]{"Checksum"};
}
```

3.SQL结果比较方法 boolean resultEquals(result1,result2)

功能： 用于判断两个不同节点的返回结果是否一致

输入： 不同节点的两个节点result, result1,result2

```
SQLQueryResult<List<Map<String, String>>> result
```

```
result
|
----- row(List)
|
-----Key-Value(Field-Value)
例如checksum
result
|
----- row(List<1> checksum table suntest只有一行返回结果)
|
-----Key-Value(checksum - 1290812451 getFetchCols只取了一列)
```

输出： 需要收集的列名list

举例：

```
public boolean resultEquals(SQLQueryResult<List<Map<String, String>>> or, SQLQueryResult<List<Map<String, String>>> cr) {
    //因为checksum只有一行,并且即使表不存在也会有一行结果集
    //所以直接取结果集的第一行即可
    Map<String, String> oresult = or.getResult().get(0);
    Map<String, String> cresult = cr.getResult().get(0);
    //直接对比Map中checksum的值是不是一致即可
    return (oresult.get("Checksum") == null && cresult.get("Checksum") == null) ||
        (oresult.get("Checksum") != null && cresult.get("Checksum") != null &&
            oresult.get("Checksum").equals(cresult.get("Checksum")));
}
```

4.失败行为接口 failResponse(resultList)

功能： 检查失败的通知/响应/其他自定义行为

输入： 检查结果列表

输出： 无

举例：

```
public void failResponse(List<SQLQueryResult<List<Map<String, String>>>> res) {
    //简单的情况下直接在日志中打印出对应信息
    //如果有需要可以自行实现发邮件/发短信/发接口给告警系统等等
    String errorMsg = "Global Consistency Check fail for table :" + schema + "-" + tableName;
    System.out.println(errorMsg);
    for (SQLQueryResult<List<Map<String, String>>> r : res) {
        System.out.println("Checksum is : " + r.getResult().get(0).get("Checksum"));
    }
}
```

5.其他结果通知 void resultResponse(errorList)

功能： 检查结果的通知/响应/其他自定义行为

输入： 检查(报错)结果列表

输出： 无

举例：

```

public void resultResponse(List<SQLQueryResult<List<Map<String, String>>> eList) {
    //输入参数是检查过程中SQL执行报错的list, 因为SQL自定义
    //不同的检查SQL对于SQL报错的处理不同, 具体报错应该别忽视
    //或者应该视作不一致, 由用户自己进行定义
    String tableId = schema + "." + tableName;

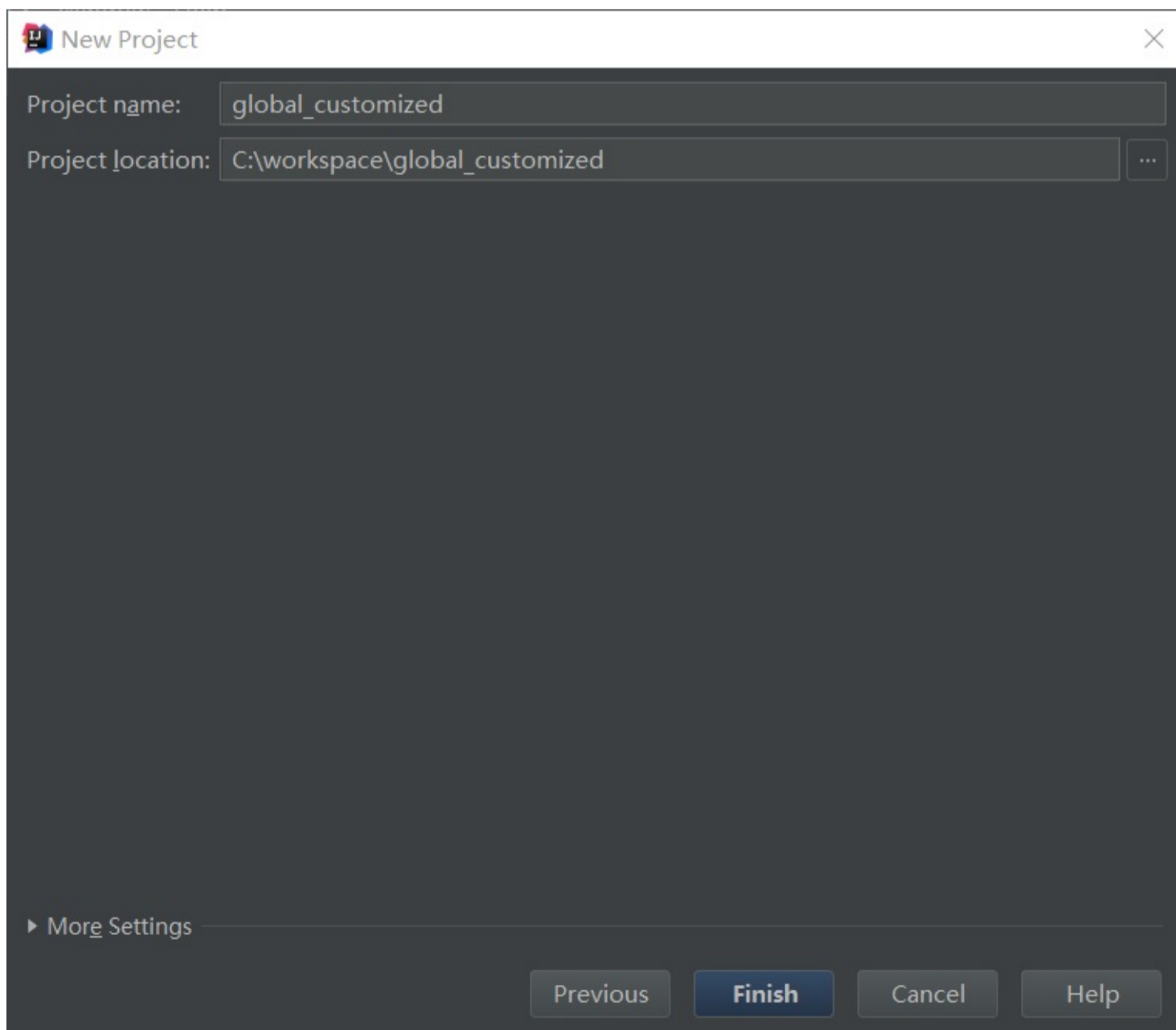
    if (eList.size() == 0) {
        System.out.println("Global Consistency Check success for table : " + schema + "-" + tableName);
    } else {
        System.out.println("Global Consistency Check fail for table : " + schema + "-" + tableName);
        StringBuilder sb = new StringBuilder("Error when check Global Consistency, Table ");
        sb.append(tableName).append(" shardingNode ");
        for (SQLQueryResult<List<Map<String, String>>> r : eList) {
            System.out.println("error node is : " + r.getTableId() + "-" + r.getShardingNode());
            sb.append(r.getShardingNode()).append(",");
        }
        sb.setLength(sb.length() - 1);
    }
}

```

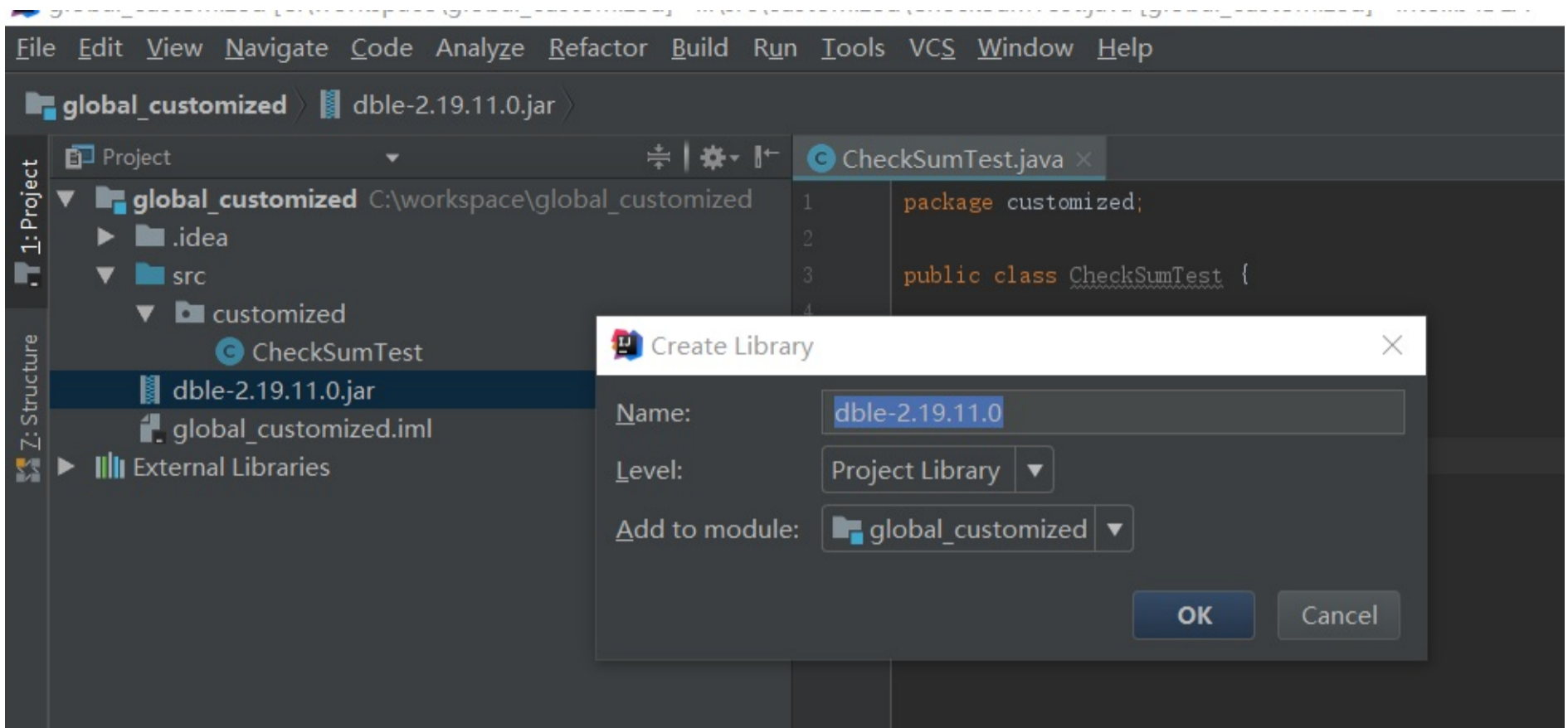
自定义全局表检查的开发及使用

检查的自定义步骤

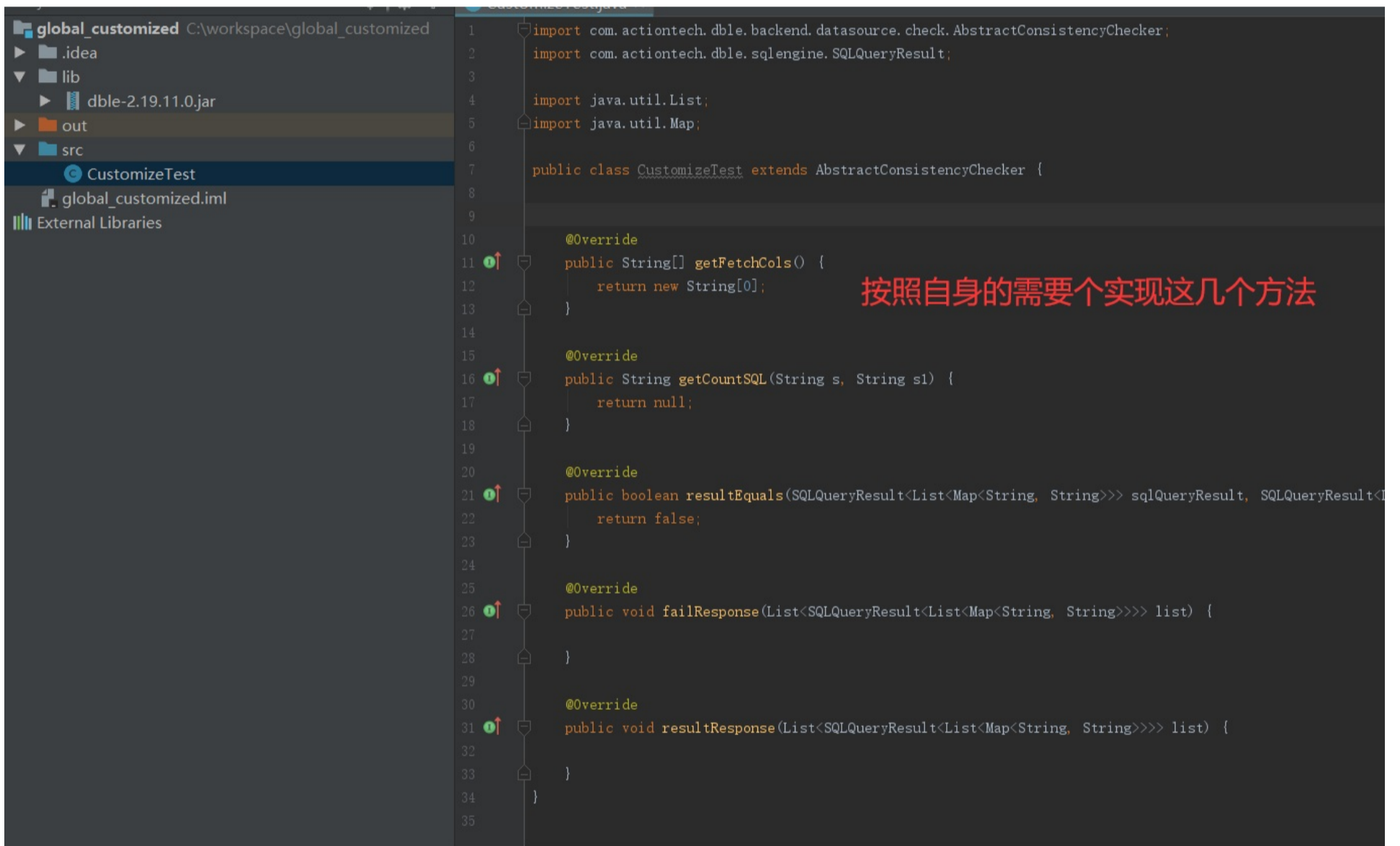
1 创建一个java项目



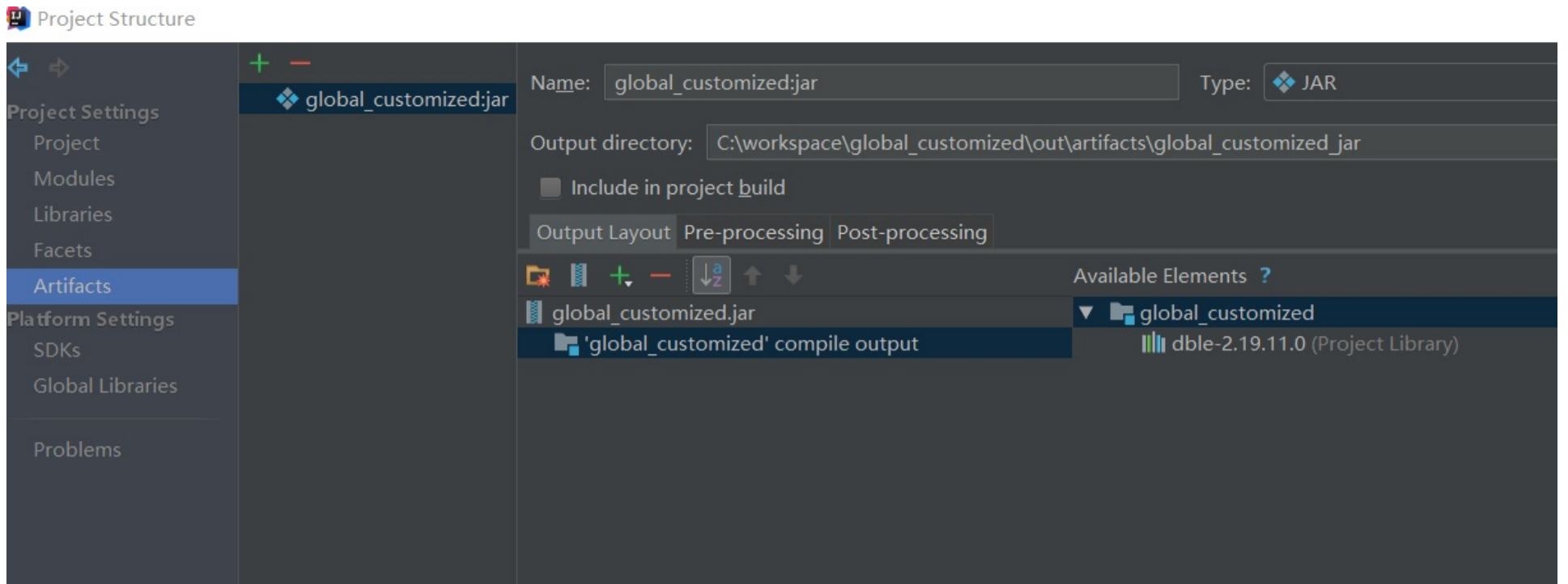
2 将需要的依赖包copy到项目中, 并且添加到lib



3 按照上一节介绍逐个实现5个自定义方法



4 打包成jar进行使用



5 示例文件及jar包

[示例代码下载](#)

自定义检查的配置

当前的全局表检查定义为schema.table级别，需要对于每个需要进行全局表一致性检查的表格进行配置，配置的下放带来一些繁琐的工作，但是却提供了一个重要的特性，用户可以根据不同全局表格的需要，或者是业务上面的特性，给与不同的全局表格不同的校验方式

注意:检查方式的修改仅在reload或者重启之后生效

举例:

```
<!-- dble内置CHECKSUM检查方式 -->
<globalTable name="tb_global1" shardingNode="dn1,dn2" cron = "0 * * * * ?" globalCheckClass="CHECKSUM"/>

<!-- dble内置COUNT检查方式 -->
<globalTable name="tb_global2" shardingNode="dn1,dn2" cron = "0 * * * * ?" globalCheckClass="COUNT"/>

<!-- 上文中我们自定义的CustomizeTest类的检查方式 -->
<globalTable name="tb_global3" shardingNode="dn1,dn2" cron = "0 * * * * ?" globalCheckClass="CustomizeTest"/>
```

自定义的jar包和其他dble内的自定义功能一样，将jar包放置于algorithm或者lib目录下就会在启动的时候被dble加载到，但是由于java中的类加载方式，如果由更新jar包内容和新增jar包的情况下，请先重启dble进程

注意: 当修改自定义jar包的时候请重启dble，此时reload可能无法得到预期的结果

1.13 Schema下默认拆分表

1.13.1 背景

需要3000+张拆分规则相同的表参与poc测试，手动逐个配置表较繁琐，希望在对应schema节点配置默认拆分规则，凡是在此schema下建立的表默认采用其拆分规则进行路由。

1.13.2 配置模版

```
<!-- schema default multi shardingNode[dn1,dn2] and split algorithm[func_common_hash];
In multi shardingNode, loaded tables are called 'default sharding tables'; In fact, equivalent to shardingTable;
But, it is not recommended to configure the Sharding table in the production environment -->
<schema name="testdb3" shardingNode="dn1,dn2" function="func_common_hash"/>
```

【注意】：不建议在生产环境中使用此配置方式创建拆分表

1.13.3 实际演练

1.13.3.1 配置

```
<schema name="TESTDB0" shardingNode="dn9,dn10" function="func_common_hash" sqlMaxLimit="100">
  <shardingTable name="tableA" shardingNode="dn1,dn2" function="func_common_hash" shardingColumn="c1"/>
</schema>
```

1.13.3.2 创建表

```
CREATE TABLE `tableA` (
  `c1` int(11) ,
  `c2` varchar(200) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `tableB` (
  `c1` int(11) auto_increment,
  `c2` varchar(200) DEFAULT NULL,
  `c3` int(11) ,
  `c4` int(11) ,
  `c5` int(11) ,
  `c6` int(11) ,
  INDEX indexs (c5,c6),
  unique KEY (`c4`),
  KEY `index1` (`c3`),
  primary KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

1.13.3.3 详解

- **Dble启动时**：加载所有分片中均存在的表；若各个节点表不一致会有相应的日志或者告警
- **shardingNode和function匹配**：与shardingTable中的shardingNode和function匹配规则一致
- **拆分列选举规则**：对schema下默认拆分表，拆分列由Dble内部选举规则决定
 - 选举时机：
 - 根据执行DDL语句中仅对 create table 语句进行选举拆分列；其他修改表结构的DDL不会重新选举拆分列
 - 加载元数据(如启动/reload)，会根据 show creat table 语句进行选举拆分列
 - 选举规则：先避开自增列(如auto_increment列)，根据优先级高低选举为拆分列：主键->唯一键->索引列->id列->第一列；(不支持与function的数据类型智能选举列)；如tableB选举的拆分列为'c4'；
 - **注意事项**：若中途执行修改表结构的DDL后，重新加载元数据(执行 reload @@metadata)，拆分列会因重新选举而可能发生变化(导致后续路由结果与之前的不一致)
- **View支持度**：schema有配置默认拆分算法时，该schema仅支持Dble层面的View
- **DML&DDL支持度**：与shardingTable支持度一致
- **告警**：开启告警功能，在加载元数据时(如reload)或者开启表一致性检查时(bootstrap.cnf中 -DcheckTableConsistency=1)
 - 部分shardingNode对应的物理库中表存在丢失场景，Dble会有对应告警提示
 - 所有shardingNode对应的物理库中表均丢失场景，Dble会从内存中移除该表；对应日志关键字检索: has been lost, will remove his metadata
- **reload**
 - reload @@metadata [where schema=? [and table=?]]：从默认拆分片中加载元数据
 - reload @@config_all [-s] [-f] [-r]：是否从默认拆分片中加载元数据需具体场景而定(理论上与默认单分片加载元数据的逻辑一致)
- **管理端表**：
 - dble_schema中，支持查看function列
 - dble_table、dble_table_sharding_node、dble_sharding_table表中，id字段增加以'FC'前缀的表，表示该表是从schema默认拆分片加载出来的；('FC'前

缀id实际上为全局id, Dble运行过程中途可能会存在表丢失或者多次reload操作等行为, 会出现id不连续)

```
mysql> select * from dble_schema;
+-----+-----+-----+-----+
| name   | sharding_node | function           | sql_max_limit |
+-----+-----+-----+-----+
| TESTDB0 | dn9, dn10     | func_common_hash   | 100           |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from dble_table;
+-----+-----+-----+-----+
| id   | name  | schema | max_limit | type   |
+-----+-----+-----+-----+
| C1   | tableA | TESTDB0 | 100       | SHARDING |
| FC2  | tableB | TESTDB0 | 100       | SHARDING |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from dble_table_sharding_node;
+-----+-----+-----+
| id   | sharding_node | order |
+-----+-----+-----+
| C1   | dn1           | 0     |
| C1   | dn2           | 1     |
| FC2  | dn9           | 0     |
| FC2  | dn10          | 1     |
+-----+-----+-----+
4 rows in set (0.01 sec)

mysql> select * from dble_sharding_table;
+-----+-----+-----+-----+
| id   | increment_column | sharding_column | sql_required_sharding | algorithm_name |
+-----+-----+-----+-----+
| C1   | NULL             | C1              | false                 | func_common_hash |
| FC2  | NULL             | C4              | false                 | func_common_hash |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

2.功能描述

- 2.0 管理端元数据库
- 2.1 管理端命令
 - 2.1.1 select命令
 - 2.1.2 set命令
 - 2.1.3 show命令
 - 2.1.4 switch命令
 - 2.1.5 kill命令
 - 2.1.6 stop命令
 - 2.1.7 reload命令
 - 2.1.8 rollback命令 (已废弃)
 - 2.1.9 offline命令
 - 2.1.10 online命令
 - 2.1.11 file命令 (已废弃)
 - 2.1.12 log命令 (已废弃)
 - 2.1.13 配置检查命令
 - 2.1.14 pause & resume 命令
 - 2.1.15 慢查询日志相关命令
 - 2.1.16 创建/删除物理库命令
 - 2.1.17 check @@metadata命令
 - 2.1.18 release @@reload_metadata命令
 - 2.1.19 split命令
 - 2.1.20 flow_control 命令
 - 2.1.21 刷新连接池命令
 - 2.1.22 脱离集群命令
- 2.2 全局序列
 - 2.2.1 MySQL offset-step方式
 - 2.2.2 时间戳方式
 - 2.2.3 分布式时间戳方式
 - 2.2.4 分布式offset-step方式
- 2.3 读写分离
- 2.4 注解
- 2.5 分布式事务
 - 2.5.1 XA事务概述
 - 2.5.2 XA事务的提交以及回滚
 - 2.5.3 XA事务的后续补偿以及日志清理
 - 2.5.4 XA事务的记录
 - 2.5.5 一般分布式事务概述
 - 2.5.6 检测疑似残留XA事务
- 2.6 连接池管理
- 2.7 内存管理
- 2.8 集群同步协调&状态管理
- 2.9 grpc 告警
- 2.10 表meta数据管理
 - 2.10.1 Meta信息初始化
 - 2.10.2 Meta信息维护
 - 2.10.3 一致性检测
 - 2.10.4 View Meta
- 2.11 统计管理
 - 2.11.1 查询条件统计
 - 2.11.2 表状态统计
 - 2.11.3 用户状态统计
 - 2.11.4 命令统计
 - 2.11.5 heartbeat统计
 - 2.11.6 网络读写统计
 - 2.11.7 快速sql统计命令
- 2.12 故障切换
- 2.13 前后端连接检查
- 2.14 ER表
- 2.15 global表
- 2.16 缓存的使用
- 2.17 执行计划
- 2.18 性能观测和调整
- 2.19 智能计算reload
- 2.20 慢查询日志

- 2.21 单条SQL性能trace
- 2.22 KILL @@DDL_LOCK
- 2.23 外部高可用联动
- 2.24 超时控制
- 2.25 流量控制
- 2.26 client_found_rows权能标志
- 2.27 general日志
- 2.28 sql统计
- 2.29 load data批处理模式
- 2.30 in子查询是否转join的说明
- 2.31 DDL日志解读
- 2.32 分析用户
- 2.33 hint指定执行计划
- 2.34 安全加密
- 2.35 堆外内存泄露监控
- 2.36 延迟检测
- 2.37 审计日志
- 2.38 tcp相关参数
- 2.39 HTAP 快速上手指南
- 2.40 dble中线程处理(print、kill、recover)

2.0 管理端元数据库 dble_information

2.0.0 简介

dble_information 提供了一系列表格来描述 dble 内部一些元数据，可以通过管理端口连接 dble 之后，use dble_information 之后查询内部的元数据信息。

以下是这些表格的详述：

支持表格的投影(select)

支持表格的选择(where)

支持表格的连接(join)

支持表格的非关联where子查询

支持聚合运算

支持排序

支持标量函数

支持 use dble_information

支持 show tables [like]

支持 desc|describe table xxx

支持 show databases; 注意和 show @@database 不同

部分表格支持 INSERT/UPDATE/DELETE

2.0.1 dble_information 下的表

2.0.1.0 version

- 表名：version
- 含义：dble 版本号
- 字段：

列名	主键	注释
version	true	版本号

- 数据行：

```
- dble 的版本号
```

2.0.1.1 dble_variables

- 表名：dble_variables
- 含义：全局设置
- 字段：

列名	主键	注释
variable_name	true	变量名
variable_value		变量值
comment		说明
read_only		是否只读

- 数据行：

```
- version_comment: 版本信息
- isOnline: 在线状态
- heap_memory_max: 堆内存的最大限制(mb)
- direct_memory_max: 通过-XX:MaxDirectMemorySize设置的值
```

附加项：show @@sysparam 中的所有配置

2.0.1.2 dble_status

- 表名：dble_status
- 含义：全局状态
- 字段：

列名	主键	注释
variable_name	true	变量名
variable_value		变量值
comment		说明

- 数据行:

- uptime: dble启动的时间长度(秒)
- current_timestamp: dble系统的当前时间
- startup_timestamp: dble系统的启动时间
- heap_memory_max: 堆内存的最大限制
- heap_memory_used: 堆内存的使用量
- heap_memory_total: 堆内存的总量
- config_reload_timestamp: 上次config加载时间
- direct_memory_max: 通过-XX:MaxDirectMemorySize设置的值
- direct_memory_pool_size: 内存池的大小, 等于bufferpoolpagesize和bufferpoolpagenumber的乘积
- direct_memory_pool_used: 已经使用的内存池中的directmemory内存
- questions: 请求数
- transactions: 事务数

2.0.1.3 thread pool系列

2.0.1.3.1 dble_thread_pool

- 表名: dble_thread_pool
- 含义: 线程池使用情况
- 字段:

列名	主键	注释
name	true	线程池名称
pool_size		实际线程池大小
core_pool_size		理论线程池大小 (和所设置值保持一致)
active_count		活跃的线程数
waiting_task_count		等待消费的任务数(如果值较大, 需增大处理线程数)

- 数据行: 略
- **注意:** 这里的 active_count/waiting_task_count 都是只统计 线程数 的变化。但是, dble 针对某些线程池 (frontWorker和writeToBackendWorker) 采用了常驻线程的实现方式, 故线程数不变, 使得这些字段不能反映 "是否消费了任务" 以及 "消费了多少个任务", 如需获得线程池执行任务的统计情况, 可以考虑使用dble_thread_pool_task表。

2.0.1.3.2 dble_thread_pool_task

- 表名: dble_thread_pool_task
- 含义: 线程池的任务消费情况
- 字段:

列名	主键	注释
name	true	线程池名称
pool_size		实际线程池大小
active_task_count		正在处理中的任务数
task_queue_size		等待消费的任务数(如果值较大, 需增大处理线程数)
completed_task		已完成的任务数量
total_task		总共任务数量

- 数据行: 略

2.0.1.4 dble_processor

- 表名: dble_processor
- 含义: processor信息
- 字段:

列名	主键	注释
name	true	名称
type		类型(session/backend)
conn_count		其负责处理的连接数
conn_net_in		网络接收流量 (线程不安全)
conn_net_out		网络发送流量 (线程不安全)

- 数据行：略

2.0.1.5 dble_sharding_node

- 表名：dble_sharding_node
- 含义：sharding_node配置及状态
- 字段：

列名	主键	注释
name	true	名称
db_group		db_group名
db_schema		db_schema名
pause		是否暂停流量

- 数据行：略

2.0.1.6 dble_db_group

- 表名：dble_db_group
- 含义：db_group配置及状态
- 字段：

列名	主键	注释
name	true	名称
heartbeat_stmt		心跳用的sql语句
heartbeat_timeout		心跳超时时间(秒)
heartbeat_retry		心跳重试次数
rw_split_mode		读写分离模式
delay_threshold		延迟时间
disable_ha		是否停用高可用
active		是否落盘

- 数据行：略

2.0.1.7 dble_db_instance

- 表名：dble_db_instance
- 含义：db_instance配置及状态
- 字段：

列名	主键	注释
name	true	名称
db_group	true	db_group的主键
addr		地址
port		端口
primary		是否是主节点
user		账号
password_encrypt		加密后的密码
encrypt_configured		是否启用加密
active_conn_count		当前活动的后端连接数量
idle_conn_count		当前空闲的后端连接数量
read_conn_request		获取读连接的次数
write_conn_request		获取写连接的次数
disabled		是否被标记为disabled
last_heartbeat_ack_timestamp		上次收到心跳回复时间戳
last_heartbeat_ack		init/ok/error/timeout
heartbeat_status		idle/checking
heartbeat_failure_in_last_5min		过去5分钟, 心跳失败的次数
min_conn_count		最小后端连接数量
max_conn_count		最大后端连接数量
read_weight		读负载
db_district		后端mysql实例所在区域
db_data_center		后端mysql实例所在数据中心
id		id
connection_timeout		获取连接的超时时间
connection_heartbeat_timeout		空闲连接检测后的超时时间
test_on_create		连接创建后是否检测有效性
test_on_borrow		连接被借出后是否检测有效性
test_on_return		连接被返回时是否检测有效性
test_while_idle		连接空闲时是否检测有效性
time_between_eviction_runs_millis		扩缩容线程的检测周期
evictor_shutdown_timeout_millis		扩缩容线程停止的超时时间
idle_timeout		连接空闲多久之后被回收
heartbeat_period_millis		连接池的心跳周期
flow_high_level		流量控制的连接高水位
flow_low_level		流量控制的连接低水位

- 数据行: 略

2.0.1.8 dble_schema

- 表名: dble_schema
- 含义: schema配置
- 字段:

列名	主键	注释
name	true	版本号
sharding_node		dble_sharding_node表的主键
ap_node		dble_ap_node表的主键
function		拆分规则
sql_max_limit		最大返回结果集限制
logical_create_and_drop		是否允许逻辑创建和删除 (为true的话, 可以执行创建和删除的语句, 但实际上不会创建或删除schema; 反之, 执行创建和删除语句会报错)

- 数据行: 略

2.0.1.9 session_variables

- 表名: session_variables
- 含义: 前端连接变量
- 字段:

列名	主键	注释
session_conn_id	true	前端连接id
variable_name	true	变量名
variable_value		变量名值
variable_type		变量类型(sys/user)

- 数据行:

已知变量：

- tx_read_only: 只读事务(mysql8.0)
- transaction_read_only: 只读事务(mysql5.7)
- character_set_client: 字符集
- collation_connection: 字符集
- character_set_results: 字符集
- tx_isolation_level: 隔离级别(mysql8.0)
- transaction_isolation: 隔离级别 (mysql5.7)
- autocommit: 自动提交
- 其他被特别设置的变量

2.0.1.10 session_connections

- 表名: session_connections
- 含义: 前端连接
- 字段:

列名	主键	注释
session_conn_id	true	前端连接id
remote_addr		远端地址
remote_port		远端端口
local_port		本地端口
processor_id		负责处理连接的处理器id
user		登录用户名
tenant		租户
schema		当前schema? (分库分表/读写分离 功能不同)
sql		最后运行的sql(如果长度大于1024个字符, 将会被截断为1024)
sql_execute_time		已完成sql的响应时间, 或未完成的sql的持续时间(单位为ms) (由于实现方式的原因, 可能出现正负20ms的误差)
sql_start_timestamp		sql的开始时间戳
sql_stage		运行的当前阶段, 结束时会变成finished
conn_net_in		网络接收流量
conn_net_out		网络发送流量
conn_estab_time		连接建立时长(毫秒)
conn_recv_buffer		接收缓冲区大小(字节) (若值较大, 说明正在接收较大的数据包, 或者接收的数据包没有及时被消费)
conn_send_task_queue		网络发送任务队列(个) (若值较大, 说明没有线程对数据包进行及时发送)
conn_recv_task_queue		网络接收任务队列(个) (若值较大, 说明没有线程对数据包进行及时消费)
in_transaction		该连接是否在事务中
xa_id		该连接中的xid, 如果存在具体值, 说明处于XA事务中
entry_id		入口id

- 数据行: 略

2.0.1.11 backend_variables

- 表名: backend_variables
- 含义: 后端连接变量
- 字段:

列名	主键	注释
backend_conn_id	true	后端连接id
variable_name	true	变量名
variable_value		变量值
variable_type		变量类型(sys/user)

- 数据行:

已知行:

- tx_read_only, 只读事务(mysql8.0)
- transaction_read_only, 只读事务(mysql5.7)
- character_set_client, 字符集
- collation_connection, 字符集
- character_set_results, 字符集
- tx_isolation_level, 隔离级别(mysql8.0)
- transaction_isolation, 隔离级别(mysql5.7)
- autocommit, 自动提交
- 其他被特别设置的变量

2.0.1.12 backend_connections

- 表名: backend_connections
- 含义: 后端连接
- 字段:

列名	主键	注释
backend_conn_id	true	后端连接id
db_group_name		db组
db_instance_name		db实例
remote_addr		远端地址
remote_port		远端端口
remote_processlist_id		远端的mysql线程id
local_port		本地端口
processor_id		负责处理连接的处理器id
user		登录用户名
schema		当前schema? (分库分表/读写分离 功能不同)
session_conn_id		与之对应的前端连接id, 未使用时可为空
sql		最后运行的sql(如果长度大于1024个字符, 将会被截断为1024)
sql_execute_time		响应时间或者未完成sql持续时间 (由于实现方式的原因, 可能出现正负20ms的误差)
mark_as_expired_timestamp		该连接被标记为退休的时间, 退休连接将不返还连接池, 当前任务结束后即关闭
conn_net_in		网络接收流量
conn_net_out		网络发送流量
conn_estab_time		连接建立时长(秒)
borrowed_from_pool		是否从连接池中取出使用(这列可以按照当前连接池属性扩展为多列)
state		后端连接在连接池中的状态, 比如是否空闲等
conn_recv_buffer		接收缓冲区大小(字节) (若值较大, 说明正在接收较大的数据包, 或者接收的数据包没有及时被消费)
conn_send_task_queue		网络发送任务队列(个) (若值较大, 说明没有线程对数据包进行及时发送)
used_for_heartbeat		该连接是否被用于心跳检测
conn_closing		该连接是否正在被关闭
xa_status		该连接的xa状态
in_transaction		该连接是否在事务中

- 数据行: 略

2.0.1.13 dble_table系列

2.0.1.13.0 dble_table

- 表名: dble_table
- 含义: table基本信息
- 字段:

列名	主键	注释
id	true	自增序列 (配置中的table前缀为C、schema配置单个mysql节点中的table的前缀为M、schema配置多个mysql节点中的table的前缀为FC)
name		名称
schema		schema名称
max_limit		最大返回结果集限制
type		global/single/sharding/child/no sharding

- 逻辑主键
name、schema
- 数据行: 略

2.0.1.13.1 dble_global_table

- 表名: dble_global_table
- 含义: 全局表信息

- 字段:

列名	主键	注释
id	true	dbtable的id字段
check		是否开启一致性检查
checkClass		开启一致性检查的class
cron		开启一致性检查的定时任务

- 数据行: 略

2.0.1.13.2 dble_sharding_table

- 表名: dble_sharding_table
- 含义: 分片表信息
- 字段:

列名	主键	注释
id	true	dbtable的id字段
increment_column		全局序列
sharding_column		拆分列
sql_required_sharding		sqlRequiredSharding配置
algorithm_name		拆分算法的名称

- 数据行: 略

2.0.1.13.3 dble_child_table

- 表名: dble_child_table
- 含义: 分片子表信息
- 字段:

列名	主键	注释
id	true	dbtable的id字段
parent_id		父表的id
increment_column		全局序列
join_column		关联列
paren_column		父表的列

- 数据行: 略

2.0.1.13.4 dble_table_sharding_node

- 表名: dble_table_sharding_node
- 含义: 表和sharding_node关联信息
- 字段:

列名	主键	注释
id	true	dbtable的id字段
sharding_node	true	分片节点
order		sharding_node的顺序(对拆分表意义重大, 从0开始计数)

- 数据行: 略

2.0.1.14 dble_algorithm

- 表名: dble_algorithm
- 含义: 拆分算法的配置
- 字段:

列名	主键	注释
name	true	名称
key	true	属性名称
value		属性值
is_file		mapfile的显示格式 (true: file, false: content) 有些算法由于使用了mapfile, 可能导致属性过多, 这种情况最多展示1024字节, 超过则显示文件名称

- 数据行: 略

2.0.1.15 dble_entry系列

2.0.1.15.0 dble_entry

- 表名: dble_entry
- 含义: 登录入口表(目前为用户或者用户+租户的模式)
- 字段:

列名	主键	注释
id	true	自增序列
type		入口类型(username/conn_attr), 通过用户名识别或通过连接属性识别
user_type		是否为管理用户/读写分离用户/sharding用户
username		用户名
password_encrypt		密码
encrypt_configured		原本密码是否加密
conn_attr_key		连接属性键, 目前支持tenant或者空
conn_attr_value		连接属性值
white_ips		白名单
readonly		是否只读 (读写分离用户不支持, 填写-)
max_conn_count		最大连接数限制
blacklist		黑名单

- 数据行: 略

2.0.1.15.1 dble_entry_schema

- 表名: dble_entry_schema
- 含义: 分库用户对应的schema的关系表
- 字段:

列名	主键	注释
id	true	dble_entry表的id
schema	true	虚拟schema的名字

- 数据行: 略

2.0.1.15.2 dble_rw_split_entry

- 表名: dble_rw_split_entry
- 含义: 分库用户对应的schema的关系表
- 字段:

列名	主键	注释
id	true	自增序列（需要实现）
type		入口类型(username/conn_attr), 通过用户名识别或通过连接属性识别
username		用户名
password_encrypt		密码
encrypt_configured		是否启用加密
conn_attr_key		连接属性键，目前支持tenant或者空
conn_attr_value		连接属性值
white_ips		白名单
max_conn_count		最大连接数限制
blacklist		黑名单
db_group		对应的db_group的名字

- 逻辑主键
username、conn_attr_key、conn_attr_value
- 数据行：略

2.0.1.15.3 dble_entry_table_privilege

- 表名：dble_entry_table_privilege
- 含义：分库用户的对于表的privilege权限
- 字段：

列名	主键	注释
id	true	dble_entry表的id
schema	true	对应的schema的名字
table	true	对应的table的名字
exist metas		对应table的元数据在dble中是否存在
insert		是否允许insert
update		是否允许update
select		是否允许select
delete		是否允许delete
is_effective		配置是否生效

- 数据行：略

2.0.1.16 dble_blacklist

- 表名：dble_blacklist
- 含义：黑名单信息
- 字段：

列名	主键	注释
name	true	名称
property_key	true	黑名单属性key
property_value		黑名单属性value
user_configured		是否是用户配置的

- 数据行：略

2.0.1.17 processlist

- 表名：processlist
- 含义：查看前端连接和后端连接对应关系(若前端连接没有对应的后端连接，显示NULL)
- 字段：

列名	主键	注释
front_id	true	前端连接id
db_instance		对应后端的实例name
mysql_id	true	后端连接对应的 mysql 线程id
user		用户名
front_host		客户端主机名
mysql_db		后端连接默认数据库, 来自于 mysql 'show processlist' 字段 db
command		mysql线程正在执行的指令类型, 来自于 mysql 'show processlist' 字段 command
time		mysql线程处于当前state的时间, 来自于 mysql 'show processlist' 字段 time
state		mysql线程执行状态, 来自于 mysql 'show processlist' 字段 state
info		mysql线程执行语句, 来自于 mysql 'show processlist' 字段 info

- 数据行: 略(类似show @@processlist)

2.0.1.18 dble_thread_usage

- 表名: dble_thread_usage
- 含义: 线程使用率
- 字段:

列名	主键	注释
thread_name	true	线程名称
last_quarter_min		最近15s平均使用率
last_minute		最近1min平均使用率
last_five_minute		最近5min平均使用率

- 数据行: 略(类似show @@thread_used;)

2.0.1.19 dble_reload_status

- 表名: dble_reload_status
- 含义: 最近的reload信息
- 字段:

列名	主键	注释
index	true	reload对应的编号, 能与日志中的[RL]日志编号相对应
cluster		当前dble使用的集群方式
reload_type		最近的reload的类型 reload_matadata/reload_all/manager_insert/manager_update/manager_delete
reload_status		最近一次reload的执行状态 not_reloading/self_reload/meta_reload/waiting_others
last_reload_start		起始时间
last_reload_end		结束时间
trigger_type		触发类型local_command/cluster_notify
end_type		结束原因

- 数据行: 略(类似show @@reload_status)

2.0.1.20 dble_xa_session

- 表名: dble_xa_session
- 含义: 后端重试xa事务信息
- 字段:

列名	主键	注释
front_id	true	前端连接id
xa_id		xa事务id
xa_state		xa事务状态
sharding_node	true	xa提交失败的sharding_node名称,需要展开成多行

- 数据行: 略(show @@session.xa)

2.0.1.21 dble_ddl_lock

- 表名: dble_ddl_lock
- 含义: 当前dble内部未释放的ddl
- 字段:

列名	主键	注释
schema	true	schema名称
table	true	table名称
sql		ddl sql语句

- 数据行: 略(类似show @@ddl)

2.0.1.22 sql_statistic_by_frontend_by_backend_by_entry_by_user

- 表名: sql_statistic_by_frontend_by_backend_by_entry_by_user
- 含义: 统计前端业务用户下发sql至分片(后端节点)的执行情况
- 字段:

列名	主键	注释
entry	true	dble_entry表的id
user	true	业务用户(不包含管理端用户)
frontend_host	true	登录的ip地址
backend_host	true	后端服务的ip地址
backend_port	true	后端服务的端口
sharding_node	true	分片节点
tx_count	false	事务次数
tx_rows	false	事务中影响或者检索的行数
tx_time	false	事务耗时
sql_insert_count	false	insert命令执行的次数
sql_insert_rows	false	insert返回的影响行数
sql_insert_time	false	insert的耗时
sql_update_count	false	update命令执行的次数
sql_update_rows	false	update返回的影响行数
sql_update_time	false	update的耗时
sql_delete_count	false	delete命令执行的次数
sql_delete_rows	false	delete返回的影响行数
sql_delete_time	false	delete的耗时
sql_select_count	false	select命令执行的次数
sql_select_rows	false	dble获取后端节点返回的行数
sql_select_time	false	select的耗时
last_update_time	false	此条记录的更新时间

- 数据如:

```
mysql> select * from sql_statistic_by_frontend_by_backend_by_entry_by_user;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| entry | user | frontend_host | backend_host | backend_port | sharding_node | db_instance | tx_count | tx_rows | tx_time | sql_insert |
|_count | sql_insert_rows | sql_insert_time | sql_update_count | sql_update_rows | sql_update_time | sql_delete_count | sql_delete_rows |
| sql_delete_time | sql_select_count | sql_select_rows | sql_select_time | last_update_time |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 3 | test | 127.0.0.1 | 10.186.63.8 | 24801 | dn1 | instanceM1 | 1 | 1 | 15293 | |
|_0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 15293 | 2021-07-09 11:18:10.525 |
| 3 | test | 127.0.0.1 | 10.186.63.7 | 24801 | dn2 | instanceM2 | 1 | 3 | 13819 |
|_0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 3 | 13819 | 2021-07-09 11:18:10.525 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

2.0.1.23 sql_statistic_by_table_by_user_by_entry

- 表名：sql_statistic_by_table_by_user_by_entry
- 含义：统计前端业务用户下发sql各个表的情况
- 字段：

列名	主键	注释
entry	true	dble_entry表的id
user	true	业务用户（不包含管理端用户）
table	true	表
sql_insert_count	false	insert命令执行的次数
sql_insert_rows	false	insert返回的影响行数
sql_insert_time	false	insert的耗时
sql_update_count	false	update命令执行的次数
sql_update_rows	false	update返回的影响行数
sql_update_time	false	update的耗时
sql_delete_count	false	delete命令执行的次数
sql_delete_rows	false	delete返回的影响行数
sql_delete_time	false	delete的耗时
sql_select_count	false	select命令执行的次数
sql_select_examined_rows	false	dble获取后端节点返回的行数
sql_select_rows	false	发送前端的行数
sql_select_time	false	select的耗时
last_update_time	false	此条记录的更新时间

- 数据如：

```
mysql> select * from sql_statistic_by_table_by_user_by_entry;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| entry | user | table                | sql_insert_count | sql_insert_rows | sql_insert_time | sql_update_count | sql_update_rows | sql_update_time | sql_delete_count | sql_delete_rows | sql_delete_time | sql_select_count | sql_select_rows | sql_select_examined_rows | sql_select_time | last_update_time |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      3 | test | testdb.tb_jump_hash |                0 |                0 |                0 |                0 |                0 |                0 |                0 |                0 |                1 |                4 |                4 |                0 | 2021-07-09 11:18:10.536 |
|      3 | test | null                 |                0 |                0 |                0 |                0 |                0 |                0 |                1 |                1 |                1 |                1 |                1 |                0 | 2021-07-09 11:09:30.755 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

2.0.1.24 sql_statistic_by_associate_tables_by_entry_by_user

- 表名：sql_statistic_by_associate_tables_by_entry_by_user
- 含义：统计前端业务用户下发sql关联表的情况
- 字段：

列名	主键	注释
entry	true	dble_entry表的id
user	true	业务用户（不包含管理端用户）
tables	true	关联的表
sql_select_count	false	select命令执行的次数
sql_select_examined_rows	false	dble获取后端节点返回的行数
sql_select_rows	false	发送前端的行数
sql_select_time	false	select的耗时
last_update_time	false	此条记录的更新时间

- 数据如：

```
mysql> select * from sql_statistic_by_associate_tables_by_entry_by_user;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| entry | user | associate_tables      | sql_select_count | sql_select_rows | sql_select_examined_rows | sql_select_time | last_update_time |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      3 | test | testdb.tabler,testdb.tb_jump_hash |                1 |                168 |                46 |                92004 | 2021-07-09 11:20:16.392 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

2.0.1.25 sql_log

- 表名：sql_log
- 含义：采样统计前端业务用户下发sql
- 字段：

列名	主键	注释
sql_id	true	sql id
sql_stmt	false	SQL语句(最长保留1024个字节)
sql_digest	false	SQL语句digest(最长保留1024个字节)
sql_type	false	SQL类型
tx_id	true	dble中事务的全局变量ID
entry	false	dble_entry表的id
user	false	用户名
source_host	false	来源IP
source_port	false	来源port
rows	false	返回前端的行数
examined_rows	false	从后端抽取的行数
result_size	false	结果集的字节大小
start_time	false	开始时间, 单位纳秒
duration	false	持续时间, 单位毫秒

- 数据如:

```
mysql> select * from sql_log;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| sql_id | sql_stmt | sql_digest | sql_type | tx_id | entry | user | source_host | source_port | rows | examined_rows | start_time | duration |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | show databases | show databases | Show | 1 | 3 | test | 127.0.0.1 | 8066 | 7 | 0 | 2021-07-09 11:09:30 | 7919460 |
| 2 | show tables | show tables | Show | 2 | 3 | test | 127.0.0.1 | 8066 | 32 | 0 | 2021-07-09 11:09:30 | 14972767 |
| 3 | select @@version_comment limit 1 | SELECT @@version_comment LIMIT ? | Select | 3 | 3 | test | 127.0.0.1 | 8066 | 1 | 0 | 2021-07-09 11:09:30 | 2131628 |
| 4 | show tables | show tables | Show | 4 | 3 | test | 127.0.0.1 | 8066 | 32 | 0 | 2021-07-09 11:09:34 | 1428683 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.04 sec)
```

2.0.1.26 sql_log_by_tx_by_entry_by_user

- 表名: sql_log_by_tx_by_entry_by_user
- 含义: sql log汇总统计
- 字段:

列名	主键	注释
tx_id	false	dble中事务的全局变量ID
entry	false	dble_entry表的id
user	false	用户名
source_host	false	来源IP
source_port	false	来源port
sql_ids	false	事务包含的sql_id的聚合(最长保留1024个字节)
sql_exec	false	事务包含的SQL的个数
tx_duration	false	事务经历的时间
busy_time	false	事务内SQL的总运行时间
examined_rows	false	从后端抽取的行数

- 数据如:

```
mysql> select * from sql_log_by_tx_by_entry_by_user;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| tx_id | entry | user | source_host | source_port | sql_ids | sql_exec | tx_duration | busy_time | examined_rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | 3 | test | 127.0.0.1 | 8066 | 1 | 1 | 7919460 | 7919460 | 0 |
| 2 | 3 | test | 127.0.0.1 | 8066 | 2 | 1 | 14972767 | 14972767 | 0 |
| 3 | 3 | test | 127.0.0.1 | 8066 | 3 | 1 | 2131628 | 2131628 | 0 |
| 4 | 3 | test | 127.0.0.1 | 8066 | 4 | 1 | 1428683 | 1428683 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.05 sec)
```

2.0.1.27 sql_log_by_digest_by_entry_by_user

- sql_log_by_digest_by_entry_by_user
- 含义：sql log汇总统计
- 字段：

列名	主键	注释
sql_digest	false	sql语句digest
entry	false	dble_entry表的id
user	false	用户名
exec	false	相同sql digest的sql语句执行次数
duration	false	sql语句的执行总时间
rows	false	返回前端的行数
examined_rows	false	从后端抽取的行数
avg_duration	false	平均sql语句执行时间

- 数据如：

```
mysql> select * from sql_log_by_digest_by_entry_by_user;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| sql_digest | entry | user | exec | duration | rows | examined_rows | avg_duration |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| SELECT @@version_comment LIMIT ? | 3 | test | 1 | 2131628 | 1 | 0 | 2131628.0000 |
| show databases | 3 | test | 1 | 7919460 | 7 | 0 | 7919460.0000 |
| show tables | 3 | test | 2 | 16401450 | 64 | 0 | 714341.5000 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.02 sec)
```

2.0.1.28 sql_log_by_tx_digest_by_entry_by_user

- sql_log_by_tx_digest_by_entry_by_user
- 含义：sql log汇总统计
- 字段：

列名	主键	注释
tx_digest	false	sql_digest的聚合
exec	false	相同tx_digest事务的执行次数
entry	false	dble_entry表的id
user	false	用户名
sql_exec	false	事务包含的SQL的执行次数
source_host	false	来源IP
source_port	false	来源port
sql_ids	false	事务包含的sql_id的聚合(最长保留1024个字节)
tx_duration	false	事务经历的时间
busy_time	false	事务内SQL的总运行时间
examined_rows	false	从后端抽取的行数

- 数据如：

```
mysql> select * from sql_log_by_tx_digest_by_entry_by_user;
```

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| tx_digest          | exec | user | entry | sql_exec | source_host | source_port | sql_ids | tx_duration | busy_time |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| SELECT @@version_comment LIMIT ? | 1 | test | 3 | 1 | 127.0.0.1 | 8066 | 3 | 2131628 | 2131628 |
| show databases    | 1 | test | 3 | 1 | 127.0.0.1 | 8066 | 1 | 7919460 | 7919460 |
| show tables       | 2 | test | 3 | 2 | 127.0.0.1 | 8066 | 2,4 | 16401450 | 16401450 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.03 sec)

```

2.0.1.29 dble_config

- 表名: dble_config
- 含义: 当前dble内存中的配置信息 (db、sharding、user、sequence)
- 字段:

列名	主键	注释
content	false	db、sharding、user、sequence的配置信息 (json字符串)

- 数据行: 略
- 使用方式见: [dble_config表](#)

2.0.1.30 dble_xa_recover

- 表名: dble_xa_recover
- 含义: 查看所有存活的主节点下XA情况
- 字段:

列名	主键	注释
dbgroup	false	节点组
instance	false	节点名
ip	false	节点ip
port	false	节点端口
formatid	false	mysql中xa_recover表中的formatid字段
gtrid_length	false	mysql中xa_recover表中的gtrid_length字段
bqual_length	false	mysql中xa_recover表中的bqual_length字段
data	false	mysql中xa_recover表中的data字段

- 数据行: 略

2.0.1.31 dble_flow_control

- 表名: dble_flow_control
- 含义: 查看所有连接的流量控制状态
- 字段:

列名	主键	注释
connection_type	true	连接的类型, 固定为 MySQLConnection/ServerConnection其中之一
connection_id	true	连接在dble中的ID信息, 可以通过ID查找日志
connection_info	false	接详细信息, 使用端口, IP地址, 用户, MySQL中的连接ID等
writing_queue_bytes	false	当前连接的准备写出的队列里积压的字节数
reading_queue_bytes	false	当前连接的已经读取的队列里积压的字节数, 前端连接不支持此功能, 恒为null
flow_controlled	false	当前连接是否处于被流控的状态

- 数据行: 略

2.0.1.32 session_connections_active_ratio

- 表名：session_connections_active_ratio
- 含义：查看前端连接分别在 近30s/1min/5min 期间的繁忙率
实际以毫秒计算，比如：近30s繁忙时间为15000ms，则繁忙率： $(15000\text{ms}/30000\text{ms}) * 100\% = 50\%$
- 字段：

列名	主键	注释
session_conn_id	true	前端连接id
last_half_minute	false	近30s 的繁忙率（实际以毫秒计算）
last_minute	false	近1min 的繁忙率
last_five_minute	false	近5min的繁忙率

- 数据行：略

2.0.1.33 session_connections_associate_thread

- 表名：session_connections_associate_thread
- 含义：查看当前时间前端连接使用的线程情况
- 字段：

列名	主键	注释
session_conn_id	true	前端连接id
thread_name	true	线程名

- 数据行：略

2.0.1.34 backend_connections_associate_thread

- 表名：backend_connections_associate_thread
- 含义：查看当前时间后端连接使用的线程情况
- 字段：

列名	主键	注释
backend_conn_id	true	后端连接id
thread_name	true	线程名

- 数据行：略

2.0.1.35 dble_cluster_renew_thread

- 表名：dble_cluster_renew_thread
- 含义：集群下的renew线程
- 字段：

列名	主键	注释
renew_thread	true	renew线程名

- 数据行：略

2.0.1.32 recycling_resource

- 表名：recycling_resource
- 含义：查看延迟关闭的资源
- 字段：

列名	主键	注释
type	false	资源类型，取值有： dbGroup/dbInstance/backendConnection
info	false	资源的详情

- 数据行：略

2.0.1.33 dble_memory_resident

- 表名：dble_memory_resident

- 含义：显示当前未回收的堆外内存，仅在开启enableMemoryBufferMonitor后有效
- 字段：

列名	主键	注释
id	true	唯一 id，也为 buffer 的地址
stacktrace	false	buffer 分配的堆栈
buffer_type	false	buffer 的用途（可能值：NORMAL（常规）/HEARTBEAT（心跳）/POOL（连接池））
allocate_size	false	buffer 期望分配的大小（非实际大小，实际大小还会向上取整取bufferPoolChunkSize的整数倍）
allocate_time	false	buffer 分配的时间点
alive_second	false	buffer 存活的时间
sql	false	与该 buffer 相关的 sql，可能为空，可能是 <<FRONT>>表示前端登录握手，<<BACK>>表示后端登录握手

- 数据行：略

2.0.1.34 dble_ap_node

- 表名：dble_ap_node
- 含义：ap_node配置及状态
- 字段：

列名	主键	注释
name	true	名称
db_group		db_group名
db_schema		db_schema名

- 数据行：略

2.0.2 支持INSERT/UPDATE/DELETE的语法&表格

为满足读写分离的场景，对部分配置表格支持增删改功能

2.0.2.0 INSERT Syntax

```
INSERT
  [INTO] tbl_name
  [(col_name [, col_name] ...)]
  {VALUES | VALUE} (value_list) [, (value_list)] ...
```

```
INSERT
  [INTO] tbl_name
  SET assignment_list
```

注意：

- 不支持 INSERT ...SELECT ,LOW_PRIORITY,DELAYED,HIGH_PRIORITY,IGNORE,ON DUPLICATE KEY UPDATE,PARTITION

2.0.2.1 UPDATE Syntax

```
UPDATE table_reference
  SET assignment_list
  WHERE where_condition
value:
  {expr | DEFAULT}
assignment:
  col_name = value
assignment_list:
  assignment [, assignment]
```

注意：

- 不支持多表更新
- 不支持子查询
- 不支持修改主键（物理主键、逻辑主键）
- 不支持LOW_PRIORITY, IGNORE, ORDER BY, LIMIT ,PARTITION
- 不支持不带条件（where）

- 不支持别名

2.0.2.2 DELETE Syntax

```
DELETE FROM tbl_name WHERE where_condition
```

注意：

- 不支持多表删除
- 不支持子查询
- 不支持LOW_PRIORITY, IGNORE, ORDER BY, LIMIT ,PARTITION
- 不支持不带条件 (where)
- 不支持别名

2.0.2.3 TRUNCATE Syntax

```
TRUNCATE [TABLE] tbl_name
```

注意：

- 暂无

2.0.2.4 支持INSERT/UPDATE/DELETE的表格

2.0.2.4.0 dble_db_group

- 不可写列：active

2.0.2.4.1 dble_db_instance

- 不可写列：active_conn_count、idle_conn_count、read_conn_request、write_conn_request、last_heartbeat_ack_timestamp、last_heartbeat_ack、heartbeat_status、heartbeat_failure_in_last_5min

注：由于dble_db_group、dble_db_instance的结构特殊性（整体存储在配置文件db.xml文件中），所以插入一个新的Mysql实例组时需先插入dble_db_group，紧接着插入相对应的dble_db_instance，才能保证实例组准确落盘；如果不遵循该插入顺序会导致数据不一致等未知问题

2.0.2.4.2 dble_rw_split_entry

- 不可写列：id、blacklist、type

2.0.2.4.3 dble_thread_pool

- 只支持UPDATE，不支持INSERT/DELETE
- 只能更新列：core_pool_size，其余列不可更新

注：

- 1、由于JDK原生线程池（ThreadPoolExecutor）扩缩容机制问题，新建的线程和即将被回收的线程需要一定的时机才会被处理，所以设置core_pool_size后可能并不是立即生效，有一定的延迟性
- 2、不支持调整AIO场景下的NIOFrontRW、NIOBackendRW参数，只支持NIO场景下的NIOFrontRW、NIOBackendRW参数
- 3、避免在高并发的时候调整线程池数目

2.0.2.5 支持TRUNCATE的表格

sql_statistic_by_frontend_by_backend_by_entry_by_user、sql_statistic_by_table_by_user_by_entry、sql_statistic_by_associate_tables_by_entry_by_user、sql_log

2.0.1.27 dble_config表

含义:

当前dble内存中的配置信息 (db、sharding、user、sequence)

使用方式:

语句: `select * from dble_config\G`

结果: Json格式, 需将结果拷贝出进行格式化查看。如: [Json格式化工具](#)

```
***** 1. row *****
content: {"version":"4.0","dbGroup":[{"rwSplitMode":0,"name":"ha_group1","delayThreshold":100,"disableHA":"true","heartbeat":{"value":"select user()"},"dbInstance":[{"name":"hostM1","url":"***","password":"123456","user":"root","maxCon":200,"minCon":10,"usingDecrypt":"false","disabled":"false","id":"hostM1Id","readWeight":"10","primary":true},{"name":"hostM5","url":"***","password":"123456","user":"root","maxCon":15,"minCon":15,"disabled":"false","primary":false}]},"rwSplitMode":0,"name":"ha_group2","heartbeat":{"value":"select user()"},"dbInstance":[{"name":"hostM2","url":"***","password":"123456","user":"root","maxCon":200,"minCon":10,"primary":true}]},"schema":[{"name":"testdb","sqlMaxLimit":100,"shardingNode":"dn1","table":[{"type":"ShardingTable","properties":{"function":"func_enum","shardingColumn":"code","name":"tb_enum_sharding","shardingNode":"dn1,dn2","sqlMaxLimit":200}},{"type":"GlobalTable","properties":{"name":"test1","shardingNode":"dn1,dn2,dn3,dn4"}}]},"name":"testdb2","shardingNode":"dn1"},"shardingNode":[{"name":"dn1","dbGroup":"ha_group1","database":"db_1"}, {"name":"dn2","dbGroup":"ha_group1","database":"db_2"}, {"name":"dn3","dbGroup":"ha_group2","database":"db_3"}, {"name":"dn4","dbGroup":"ha_group2","database":"db_4"}],"function":[{"name":"func_enum","clazz":"Enum","property":[{"value":"partition-enum.txt","name":"mapFile"}, {"value":"0","name":"defaultNode"}, {"value":"1","name":"type"}]}],"user":[{"type":"ManagerUser","properties":{"readOnly":false,"name":"man1","password":"654321","usingDecrypt":"false","maxCon":10}},{"type":"ShardingUser","properties":{"schemas":"testdb","readOnly":false,"blacklist":"blacklist1","name":"root","password":"123456","maxCon":20}},{"type":"RwSplitUser","properties":{"dbGroup":"ha_group3","blacklist":"blacklist1","name":"rwsu1","password":"123456","maxCon":20}]},"blacklist":[{"name":"blacklist1"}],"sequence_db_conf.properties":{"`TESTDB`.`GLOBAL`":"dn1","`TESTDB`.`COMPANY`":"dn1","`TESTDB`.`CUSTOMER`":"dn1","`TESTDB`.`ORDERS`":"dn1","`TESTDB`.`myauto_test`":"dn1"}}
1 row in set (0.33 sec)
```

格式化后:

```
{
  "version":"4.0",
  "dbGroup":[
    {
      "rwSplitMode":0,
      "name":"ha_group1",
      "delayThreshold":100,
      "disableHA":"true",
      "heartbeat":{
        "value":"select user()"
      },
      "dbInstance":[
        {
          "name":"hostM1",
          "url":"***",
          "password":"123456",
          "user":"root",
          "maxCon":200,
          "minCon":10,
          "usingDecrypt":"false",
          "disabled":"false",
          "id":"hostM1Id",
          "readWeight":"10",
          "primary":true
        },
        {
          "name":"hostM5",
          "url":"***",
          "password":"123456",
          "user":"root",
          "maxCon":15,
          "minCon":15,
          "disabled":"false",
          "primary":false
        }
      ]
    },
    {
      "rwSplitMode":0,
      "name":"ha_group2",
      "heartbeat":{
        "value":"select user()"
      },
```

```

    "dbInstance":[
      {
        "name":"hostM2",
        "url":"**",
        "password":"123456",
        "user":"root",
        "maxCon":200,
        "minCon":10,
        "primary":true
      }
    ]
  },
  {
    "rwSplitMode":0,
    "name":"ha_group3",
    "delayThreshold":-1,
    "heartbeat":{
      "value":"select user()"
    },
    "dbInstance":[
      {
        "name":"hostM3",
        "url":"**",
        "password":"123456",
        "user":"root",
        "maxCon":15,
        "minCon":15,
        "disabled":"false",
        "primary":true
      }
    ]
  }
],
"schema":[
  {
    "name":"testdb",
    "sqlMaxLimit":100,
    "shardingNode":"dn1",
    "table":[
      {
        "type":"ShardingTable",
        "properties":{
          "function":"func_enum",
          "shardingColumn":"code",
          "name":"tb_enum_sharding",
          "shardingNode":"dn1, dn2",
          "sqlMaxLimit":200
        }
      },
      {
        "type":"GlobalTable",
        "properties":{
          "name":"test1",
          "shardingNode":"dn1, dn2, dn3, dn4"
        }
      }
    ]
  },
  {
    "name":"testdb2",
    "shardingNode":"dn1"
  }
],
"shardingNode":[
  {
    "name":"dn1",
    "dbGroup":"ha_group1",
    "database":"db_1"
  },
  {
    "name":"dn2",
    "dbGroup":"ha_group1",
    "database":"db_2"
  },
  {
    "name":"dn3",
    "dbGroup":"ha_group2",
    "database":"db_3"
  }
]

```

```

    },
    {
      "name": "dn4",
      "dbGroup": "ha_group2",
      "database": "db_4"
    }
  ],
  "function": [
    {
      "name": "func_enum",
      "clazz": "Enum",
      "property": [
        {
          "value": "partition-enum.txt",
          "name": "mapFile"
        },
        {
          "value": "0",
          "name": "defaultNode"
        },
        {
          "value": "1",
          "name": "type"
        }
      ]
    }
  ],
  "user": [
    {
      "type": "ManagerUser",
      "properties": {
        "readOnly": false,
        "name": "man1",
        "password": "654321",
        "usingDecrypt": "false",
        "maxCon": 10
      }
    },
    {
      "type": "ShardingUser",
      "properties": {
        "schemas": "testdb",
        "readOnly": false,
        "blacklist": "blacklist1",
        "name": "root",
        "password": "123456",
        "maxCon": 20
      }
    },
    {
      "type": "RwSplitUser",
      "properties": {
        "dbGroup": "ha_group3",
        "blacklist": "blacklist1",
        "name": "rwsu1",
        "password": "123456",
        "maxCon": 20
      }
    }
  ],
  "blacklist": [
    {
      "name": "blacklist1"
    }
  ],
  "sequence_db_conf.properties": {
    "`TESTDB`.`GLOBAL`": "dn1",
    "`TESTDB`.`COMPANY`": "dn1",
    "`TESTDB`.`CUSTOMER`": "dn1",
    "`TESTDB`.`ORDERS`": "dn1",
    "`TESTDB`.`myauto_test`": "dn1"
  }
}

```

适用场景：

在此之前查看dble中的配置只能通过配置文件一一查看，由于文件具有可修改的属性，因此这种方式得到的配置可能和真实的配置不一致。

表dble_config中的数据来自于dble内存中的实际配置信息，通过查询该表得到的配置更具备真实性、权威性

2.1 管理端命令集

- 2.1.1 select命令
- 2.1.2 set命令
- 2.1.3 show命令
- 2.1.4 switch命令
- 2.1.5 kill命令
- 2.1.6 stop命令
- 2.1.7 reload命令
- 2.1.8 rollback命令 (已废弃)
- 2.1.9 offline命令
- 2.1.10 online命令
- 2.1.11 file命令 (已废弃)
- 2.1.12 log命令 (已废弃)
- 2.1.13 配置检查命令
- 2.1.14 pause & resume 命令
- 2.1.15 慢查询日志相关命令
- 2.1.16 创建/删除物理库命令
- 2.1.17 check @@metadata命令
- 2.1.18 release @@reload_metadata 命令
- 2.1.19 split 命令
- 2.1.20 flow_control 命令
- 2.1.21 刷新连接池命令
- 2.1.22 脱离集群命令

2.1.1 select 命令

2.1.1.1 select @@VERSION_COMMENT

select @@VERSION_COMMENT;

描述：查询dble的版本信息；

例：

```
MySQL [(none)]> select @@VERSION_COMMENT;
+-----+
| @@VERSION_COMMENT |
+-----+
| dble Server (ActionTech) |
+-----+
1 row in set (0.02 sec)
```

列描述：

略

2.1.1.2 select @@SESSION.TX_READ_ONLY / select @@SESSION.Transaction_READ_ONLY

select @@SESSION.TX_READ_ONLY;

select @@SESSION.Transaction_READ_ONLY

描述：为了支持驱动连接管理端时下发的上下文

结果：返回管理用户是否readonly

2.1.1.3 select @@max_allowed_packet

select @@max_allowed_packet;

描述：限制请求的包大小

特殊说明：

- 1.如果mysql中限制请求的包大小超过该值+1024，那么dble就不会同步该值到mysql中。所以会出现mysql max_allowed_packet和 dble中不一致的情况。
- 2.在客户端使用该语句依然返回的是dble在配置文件中定义的值。
- 3.类似其他相同作用语句比如 show variables like 'max_allowed_packet' 会下发到其中一个节点的mysql中查询，返回结果就可能会和dble在配置文件中定义的值不一致。
- 4.dble中的值一定小于等于mysql中的值。

例：

```
mysql> select @@max_allowed_packet;
+-----+
| @@max_allowed_packet |
+-----+
|          16776640 |
+-----+
1 row in set (0.01 sec)
```

2.1.1.4 select TIMEDIFF(NOW(), UTC_TIMESTAMP())

select TIMEDIFF(NOW(), UTC_TIMESTAMP())

描述：无实际意义，仅为了支持驱动连接管理端时下发的上下文

结果：定值，永远返回00:00:00

2.1.2 set xxx

set xxx;

其中，xxx为要设置的变量。

描述：无实际意义，仅为了支持驱动连接管理端时下发的上下文

结果：永远返回OK

2.1.3 show命令

2.1.3.1 show @@time.current

show @@time.current;

描述：展示系统当前时间

结果：略

2.1.3.2 show @@time.startup

show @@time.startup;

描述：展示系统启动时间

结果：略

2.1.3.3 show @@version

show @@version;

描述：展示dble版本

结果：略

2.1.3.4 show @@server

show @@server;

描述：dble的当前信息

例：

```
mysql> show @@server;
+-----+-----+-----+-----+-----+-----+-----+
| UPTIME      | USED_MEMORY | TOTAL_MEMORY | MAX_MEMORY | RELOAD_TIME      | CHARSET | STATUS |
+-----+-----+-----+-----+-----+-----+-----+
| 1h 4m 47s  | 17414592 | 87031808 | 1840250880 | 2017/10/17 16:42:09 | utf8    | ON     |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.05 sec)
```

列描述：

```
UPTIME: 服务已经启动时间
USED_MEMORY: 已使用堆内存
TOTAL_MEMORY: 总共的堆内存
MAX_MEMORY: 最大可用堆内存
RELOAD_TIME: 上次config加载时间
CHARSET: 当前管理端登录用户指定的COLLATE字符集
STATUS: 在线状态
```

2.1.3.5 show @@threadpool / show @@threadpool.task

show @@threadpool;

描述：展示当前线程池的线程信息

例：

```
mysql> show @@threadpool;
+-----+-----+-----+-----+-----+-----+
| NAME          | POOL_SIZE | ACTIVE_COUNT | TASK_QUEUE_SIZE | COMPLETED_TASK | TOTAL_TASK |
+-----+-----+-----+-----+-----+-----+
| Timer         | 1 | 0 | 0 | 22596 | 22596 |
| frontWorker  | 8 | 8 | 0 | 216 | 217 |
| managerFrontWorker | 2 | 2 | 0 | 216 | 217 |
| backendWorker | 8 | 8 | 0 | 216 | 217 |
| complexQueryWorker | 8 | 0 | 0 | 0 | 0 |
| writeToBackendWorker | 8 | 8 | 0 | 0 | 0 |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.03 sec)
```

列描述：

NAME: 线程池名称
 POOL_SIZE: 线程池大小
 ACTIVE_COUNT: 处理中的线程数量
 TASK_QUEUE_SIZE: 队列中的线程数量
 COMPLETED_TASK: 已完成的线程数量
 TOTAL_TASK: 总共线程数量

注意:

- 这里的 ACTIVE_COUNT/TASK_QUEUE_SIZE/COMPLETED_TASK/TOTAL_TASK 都是只统计 线程数 的变化。但是, dble 针对某些线程池 (frontWorker和 writeToBackendWorker) 采用了常驻线程的实现方式, 故线程数不变, 使得这些字段不能反映 "是否消费了任务" 以及 "消费了多少了个任务"。为此, 如需获得线程池执行任务的统计情况, 可以考虑使用show @@threadpool.task。

show @@threadpool.task;

描述: 展示当前线程池的执行任务的情况

例:

```
mysql> show @@threadpool.task;
+-----+-----+-----+-----+-----+-----+
| NAME          | POOL_SIZE | ACTIVE_TASK_COUNT | TASK_QUEUE_SIZE | COMPLETED_TASK | TOTAL_TASK |
+-----+-----+-----+-----+-----+-----+
| Timer         | 1         | 0                 | 0               | 100             | 100        |
| frontWorker   | 8         | 8                 | 0               | 45              | 46         |
| managerFrontWorker | 2         | 2                 | 0               | 45              | 46         |
| backendWorker | 8         | 8                 | 0               | 1631            | 1631       |
| complexQueryWorker | 8         | 0                 | 0               | 98              | 98         |
| writeToBackendWorker | 8         | 8                 | 0               | 0               | 0          |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

列描述:

NAME: 线程池名称
 POOL_SIZE: 线程池大小
 ACTIVE_TASK_COUNT: 处理中的任务数量
 TASK_QUEUE_SIZE: 队列中的任务数量
 COMPLETED_TASK: 已完成的任务数量
 TOTAL_TASK: 总共任务数量

2.1.3.6 show @@database

show @@database;

描述: 展示配置的schema名字

结果: 略

2.1.3.7 show @@shardingnode

show @@shardingnode;

描述: 展示配置中所有已使用的shardingnode信息

例:

```
mysql> show @@shardingnode;
+-----+-----+-----+-----+-----+-----+-----+-----+
| NAME | DB_GROUP          | SCHEMA_EXISTS | ACTIVE | IDLE | SIZE | EXECUTE | RECOVERY_TIME |
+-----+-----+-----+-----+-----+-----+-----+-----+
| dn1  | dh1/dble_test    | true          | 0     | 0   | 1000 | 34      | -1             |
| dn2  | dh2/dble_test    | true          | 0     | 0   | 1000 | 34      | -1             |
| dn3  | dh1/dble2_test   | false         | 0     | 0   | 1000 | 26      | -1             |
| dn4  | dh2/dble2_test   | true          | 0     | 0   | 1000 | 26      | -1             |
| dn5  | dh1/nosharding   | true          | 0     | 0   | 1000 | 9       | -1             |
+-----+-----+-----+-----+-----+-----+-----+-----+
5 rows in set (0.09 sec)
```

列描述:

NAME: 名称
 DB_GROUP: dbGroupName/实际schema
 SCHEMA_EXISTS: 对应后端物理库是否存在, true为存在, false为不存在。
 ACTIVE: 当前活动的后端连接数量
 IDLE: 当前空闲的后端连接数量(空闲容量维护疑似bug)
 SIZE: maxCon容量
 EXECUTE: 有过活动的后端连接数量统计
 RECOVERY_TIME: 恢复心跳还需要秒数(stop @@heartbeat 中设置)

如果要查看某个schema相关的shardingnode信息, 执行:

```
show @@shardingnode where schema=xxx;
```

其中, xxx为要查看的schema的名字。

2.1.3.8 show @@dbinstance

show @@dbinstance

描述: 展示配置的所有dbinstance信息

例:

```
mysql> show @@dbinstance;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| DB_GROUP | NAME | HOST | PORT | W/R | ACTIVE | IDLE | SIZE | EXECUTE | READ_LOAD | WRITE_LOAD | DISABLED |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| localhost2 | hostS1 | 10.18x.2x.63 | 3307 | W | 1 | 9 | 100 | 11 | 0 | 0 | true |
| localhost1 | hostM1 | 10.18x.2x.64 | 3306 | W | 1 | 9 | 100 | 17 | 0 | 0 | false |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.09 sec)
```

列描述

DB_GROUP: dbinstance所属DB_GROUP
 NAME: dbinstance名称
 HOST: host名
 PORT: 端口
 W/R: 读写结点标识
 ACTIVE: 当前活动的后端连接数量, 按照dbinstance统计
 IDLE: 当前空闲的后端连接数量, 按照dbinstance统计(空闲容量维护疑似bug)
 SIZE: maxCon容量
 EXECUTE: 有过活动的后端连接数量统计, 按照dbinstance统计
 READ_LOAD: 对select或者show语句的统计 (每个事务统计一次)
 WRITE_LOAD: 非select或者show语句的统计, 当开启显示事务后sql都被统计为WRITE_LOAD(每个事务统计一次)
 DISABLED: db.xml中dbinstance中的配置 (2.19.09.0以前的版本没有此列, disabled为true的结点不显示)

如果要查看某个shardingnode对应的dbinstance信息, 执行:

```
show @@dbinstance where shardingnode=xxx;
```

其中, xxx为要查看的shardingnode的名字。

2.1.3.9 show @@dbinstance.synstatus

show @@dbinstance.synstatus;

描述: 展示当前各dbinstance的同步信息

前提条件: heartbeat 配置了 show slave status (参见db.xml)

例:

```
mysql> show @@dbinstance.synstatus \G
***** 1. row *****
      DB_GROUP: dbGroup2
        NAME: instanceM3
        HOST: 111.231.25.141
        PORT: 30309
    MASTER_HOST: mysql3
    MASTER_PORT: 3306
    MASTER_USER: replicator
SECONDS_BEHIND_MASTER: 0
  SLAVE_IO_RUNNING: Yes
  SLAVE_SQL_RUNNING: Yes
    SLAVE_IO_STATE: Waiting for master to send event
    CONNECT_RETRY: 10
    LAST_IO_ERROR:
1 row in set (0.00 sec)
```

列描述:

```
DB_GROUP: dbinstance所属DB_GROUP
NAME: dbinstance名称
HOST: 主机名/ip
PORT: 端口
```

其余列含义参见mysql中show slave status的命令。

2.1.3.10 show @@dbinstance.syndetail where name=?

show @@dbinstance.syndetail where name=xxx;

其中, xxx为要查看的dbinstance的名字。

描述: 展示24小时内各dbinstance的历次同步信息

例:

```
mysql> show @@dbinstance.syndetail WHERE name =hostM2;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| DB_GROUP | NAME | HOST | PORT | MASTER_HOST | MASTER_PORT | MASTER_USER | TIME | SECONDS_BEHIND_MASTER |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| localhost2 | hostM2 | 10.18x.2x.64 | 3320 | 10.18x.2x.62 | 3320 | qrep | 2017-10-17 18:31:27 | -1 |
| localhost2 | hostM2 | 10.18x.2x.64 | 3320 | 10.18x.2x.62 | 3320 | qrep | 2017-10-17 18:31:57 | -1 |
| localhost2 | hostM2 | 10.18x.2x.64 | 3320 | 10.18x.2x.62 | 3320 | qrep | 2017-10-17 18:32:27 | -1 |
| localhost2 | hostM2 | 10.18x.2x.64 | 3320 | 10.18x.2x.62 | 3320 | qrep | 2017-10-17 18:32:57 | -1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
4 row in set (0.05 sec)
```

列描述:

```
DB_GROUP: dbinstance所属DB_GROUP
NAME: dbinstance名称
HOST: 主机名/ip
PORT: 端口
```

其余列含义参见mysql中show slave status的命令。

2.1.3.11 show @@datasource.cluster

show @@datasource.cluster;

描述: 此功能在2.20.04.0版本已经废除。

2.1.3.12 show @@processor

show @@processor;

描述: 展示dble实例的processor信息

例:

```

mysql> show @@processor\G
***** 1. row *****
      NAME: frontProcessor0
      NET_IN: 0
      NET_OUT: 0
      REACT_COUNT: 0
      R_QUEUE: 0
      W_QUEUE: 0
      FREE_BUFFER: 1072169008
      TOTAL_BUFFER: 1073741824
      BU_PERCENT: 0
      BU_WARN: 0
      FC_COUNT: 0
      BC_COUNT: 0
***** 2. row *****
      NAME: frontProcessor1
      NET_IN: 0
      NET_OUT: 267
      REACT_COUNT: 0
      R_QUEUE: 0
      W_QUEUE: 0
      FREE_BUFFER: 1072169008
      TOTAL_BUFFER: 1073741824
      BU_PERCENT: 0
      BU_WARN: 0
      FC_COUNT: 0
      BC_COUNT: 0
***** 3. row *****
      NAME: frontProcessor2
      NET_IN: 0
      NET_OUT: 150
      REACT_COUNT: 0
      R_QUEUE: 0
      W_QUEUE: 0
      FREE_BUFFER: 1072169008
      TOTAL_BUFFER: 1073741824
      BU_PERCENT: 0
      BU_WARN: 0
      FC_COUNT: 0
      BC_COUNT: 0
***** 4. row *****
      NAME: frontProcessor3
      NET_IN: 0
      NET_OUT: 1548
      REACT_COUNT: 0
      R_QUEUE: 0
      W_QUEUE: 0
      FREE_BUFFER: 1072169008
      TOTAL_BUFFER: 1073741824
      BU_PERCENT: 0
      BU_WARN: 0
      FC_COUNT: 0
      BC_COUNT: 0
...

```

列描述:

```

NAME: 名称
NET_IN: 接收流量
NET_OUT: 发送流量
REACT_COUNT: 固定值0
R_QUEUE: 固定值0
W_QUEUE: 写队列大小
FREE_BUFFER: BufferPool free大小
TOTAL_BUFFER: BufferPool 总大小
BU_PERCENT: BufferPool使用率百分比
BU_WARN: 固定值0
FC_COUNT: 前端连接数量
BC_COUNT: 后端连接数量

```

2.1.3.13 show @@command

show @@command;

描述: processor对各个类型的数据包的分类统计信息

例:

```
mysql> show @@command;
```

PROCESSOR	INIT_DB	QUERY	STMT_PREPARE	STMT_EXECUTE	STMT_CLOSE	PING	KILL	QUIT	OTHER
Processor0	0	0	0	0	0	0	0	0	0
Processor1	0	0	0	0	0	0	0	0	0
Processor2	0	6	0	0	0	0	0	0	0
Processor3	0	4	0	0	0	0	0	0	0

4 rows in set (0.00 sec)

列描述：

```
PROCESSOR: processor名称
INIT_DB: COM_INIT_DB
QUERY: COM_QUERY
STMT_PREPARE: COM_STMT_PREPARE
STMT_EXECUTE: COM_STMT_EXECUTE
STMT_CLOSE: COM_STMT_CLOSE
PING: COM_PING
KILL: COM_PROCESS_KILL
QUIT: COM_QUIT
OTHER: 其余
```

2.1.3.14 show @@connection where processor=? and front_id=? and host=? and user=?

```
show @@connection where processor=? and front_id=? and host=? and user=?;
```

描述：查询前端连接信息，可通过processor, front_id, host和user进行过滤筛选，条件可以任意组合搭配。

例：

```
mysql> show @@connection where processor='frontProcessor4' \G
***** 1. row *****
      PROCESSOR: frontProcessor4
      FRONT_ID: 4
      HOST: 192.168.2.190
      PORT: 9066
      LOCAL_PORT: 52082
      USER: man1
      SCHEMA:
      CHARACTER_SET_CLIENT: utf8mb4
      COLLATION_CONNECTION: utf8mb4_general_ci
      CHARACTER_SET_RESULTS: utf8mb4
      NET_IN: 1438
      NET_OUT: 10925
      ALIVE_TIME(S): 526
      RECV_BUFFER: 32767
      SEND_QUEUE: 0
      RECV_QUEUE: 0
      TX_ISOLATION_LEVEL:
      AUTOCOMMIT:
      SYS_VARIABLES:
      USER_VARIABLES:
      XA_ID: -
1 row in set (0.01 sec)
```

结果列描述：

PROCESSOR: PROCESSOR名称
 FRONT_ID: 前端连接ID
 HOST: 客户端host
 PORT: 本地端口(流量或者管理)
 LOCAL_PORT: 客户端端口
 USER: 用户
 SCHEMA: 所在的schema
 CHARACTER_SET_CLIENT: 字符集信息
 COLLATION_CONNECTION: 字符集信息
 CHARACTER_SET_RESULTS : 字符集信息
 NET_IN: 接收流量
 NET_OUT: 发送流量
 ALIVE_TIME(S): 连接建立时长
 RECV_BUFFER: 接收缓冲区大小(字节)
 SEND_QUEUE: 发送缓冲区队列里的任务数量
 RECV_QUEUE: 接收缓冲区队列里的任务数量
 TX_ISOLATION_LEVEL: 隔离级别
 AUTOCOMMIT: 略
 SYS_VARIABLES: 系统变量
 USER_VARIABLES: 用户变量

2.1.3.15 show @@cache

show @@cache;

描述: 展示cache信息

例:

```
mysql> show @@cache;
+-----+-----+-----+-----+-----+-----+-----+-----+
| CACHE          | MAX  | CUR  | ACCESS | HIT  | PUT  | LAST_ACCESS | LAST_PUT |
+-----+-----+-----+-----+-----+-----+-----+-----+
| ER_SQL2PARENTID | 1000 | 0    | 0      | 0    | 0    |              |          |
| SQLRouteCache  | 10000| 0    | 0      | 0    | 0    |              |          |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.09 sec)
```

列描述:

CACHE: cache名
 MAX: 最大容量
 CUR: 当前容量
 ACCESS: 缓存查询次数
 HIT: 命中次数
 PUT: 加入缓存计数器
 LAST_ACCESS: 上一次查询时间戳(格式为yyyy/mm/dd hh:mm:ss)
 LAST_INPUT: 上一次加入缓存时间戳(格式为yyyy/mm/dd hh:mm:ss)

2.1.3.16 show @@backend where processor=? and backend_id=? and mysqlid=? and host=? and port=?

show @@backend where processor=? and backend_id=? and mysqlid=? and host=? and port=?;

描述: 查询活动的后端连接信息, 可与show @@session结合使用。该命令可通过processor, backend_id, mysqlid, host和port进行过滤筛选, 条件可以任意组合搭配。

例:

```
mysql> show @@backend where processor='backendProcessor9' and host='172.18.0.3' \G
***** 1. row *****
      processor: backendProcessor9
      BACKEND_ID: 29
      MYSQLID: 26
      HOST: 172.18.0.3
      PORT: 3306
      LOCAL_TCP_PORT: 34848
      NET_IN: 93
      NET_OUT: 85
      ACTIVE_TIME(S): 699
      CLOSED: false
      STATE: IDLE
      SEND_QUEUE: 0
      SCHEMA: NULL
      CHARACTER_SET_CLIENT: utf8mb4
      COLLATION_CONNECTION: utf8mb4_general_ci
      CHARACTER_SET_RESULTS: utf8mb4
      TX_ISOLATION_LEVEL: 2
      AUTOCOMMIT: true
      SYS_VARIABLES:
      USER_VARIABLES:
      XA_STATUS: 0
      DEAD_TIME:
      USED_FOR_HEARTBEAT: false
1 row in set (0.01 sec)
```

列描述:

```
processor: processor名称
BACKEND_ID: 后端连接ID
MYSQLID: mysql线程id(对应节点上的show processlist里的MYSQLID)
HOST: 主机名
PORT: 端口
LOCAL_TCP_PORT: tcp连接的本地端口
NET_IN: 接收流量大小
NET_OUT: 发送流量大小
ACTIVE_TIME(S): 连接建立时间(单位秒)
CLOSED: 是否被关闭
STATE: 使用状态,有以下几种状态: IN USE(使用中)、IDLE(空闲)、HEARTBEAT CHECK(心跳检测中)、EVICT、IN CREATION OR OUT OF POOL(新建或在池外)、UNK
NOWN STATE(未知状态)
SEND_QUEUE: 发送缓冲队列大小
SCHEMA: schema上下文
CHARACTER_SET_CLIENT: 字符集信息
COLLATION_CONNECTION: 字符集信息
CHARACTER_SET_RESULTS: 字符集信息
TX_ISOLATION_LEVEL: 隔离级别(新建未使用过的连接为-1,表示未初始化)
AUTOCOMMIT: 是否自动提交
SYS_VARIABLES: 系统变量
USER_VARIABLES: 用户变量
XA_STATUS: xa状态
DEAD_TIME: 连接池被回收的时间,连接在完成任务后也会关闭回收
USED_FOR_HEARTBEAT: 是否被用于心跳
```

2.1.3.17 show @@session

show @@session;

描述: 展示当前活动前端session的后端连接信息

例:

```
mysql> show @@session ;
+-----+-----+-----+
| FRONT_ID | DN_COUNT | DN_LIST |
+-----+-----+-----+
| 2 | 2 | BackendConnection[backendId=59, host=172.100.9.5 [... ] |
+-----+-----+-----+
1 row in set (0.00 sec)
```

列描述:

```
FRONT_ID: 前端连接ID
DN_COUNT: 后端连接个数:
DN_LIST: 后端连接的详情,
```


DN_LIST例如:

```
BackendConnection[id = 15 host = **** port = 3306 localPort = 56355 mysqlId = 53690 db config = dbInstance[name=hostM1,disabled=false,maxCon=6000,minCon=17]
```

含义:

```
id: 后端连接id
host: host ip
port: 端口号
localPort: dble对应后端连接的本地端口
mysqlId: 对应后端连接在数据库内的线程id (show processlist)
db config: dble中的配置信息
  name: 实例名
  disabled: 实例是否可用
  maxCon: 实例最大连接数
  minCon: 实例最小连接数
```

2.1.3.18 show @@connection.sql

show @@connection.sql;

描述: 当前活动session的前端的SQL信息

例:

```
mysql> show @@connection.sql;
+-----+-----+-----+-----+-----+-----+-----+-----+
| FRONT_ID | HOST          | USER | SCHEMA | START_TIME          | EXECUTE_TIME | SQL                               | STAGE   |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1        | 0:0:0:0:0:0:1 | man  | NULL   | 2017/10/17 17:00:58 | 139          | show @@connection.sql           | Read SQL |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.13 sec)
```

列描述:

```
FRONT_ID: 前端连接ID
HOST: 客户端host
USER: 用户
SCHEMA: 所在的schema
START_TIME: 上次接收请求时间戳
EXECUTE_TIME: 响应时间或者未完成SQL持续时间 (由于实现方式的原因, 可能出现正负20ms的误差)
SQL: 如果长度大于1024个字符, 将会被截断为1024
STAGE: 运行的当前阶段, 结束时会变成finished
```

2.1.3.19 show @@sql

show @@sql;

描述: 直接取sql_log表的数据

例:

```
mysql> show @@sql;
+-----+-----+-----+-----+-----+-----+
| ID | USER | START_TIME          | EXECUTE_TIME | SQL                               |
+-----+-----+-----+-----+-----+-----+
| 1  | root | 2017/10/17 17:37:22 | 381          | select * from sharding_two_node LIMIT 100 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.02 sec)
```

列描述:

```
ID: 行号
USER: 用户
START_TIME: 上次接收请求时间戳
EXECUTE_TIME: 响应时间
SQL: 略
```

类似查询SQL:

```
select sql_id as ID, user as USER, start_time as START_TIME, duration as EXECUTE_TIME, sql_stmt as SQL from dble_information.sql_log order by start_time desc
```

2.1.3.20 show @@sql.high

show @@sql.high;

描述：取sql_log表中数据按照执行频率降序查询

例：

```
mysql> show @@sql.high;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+
+ | ID   | USER | FREQUENCY | AVG_TIME | MAX_TIME | MIN_TIME | EXECUTE_TIME | LAST_TIME           | SQL
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+
+ | 1   | root |          1 |      381 |      381 |      381 |          381 | 2017/10/17 17:37:23 | SELECT * FROM sharding_two_node LIMIT ?
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+
1 row in set (0.06 sec)
```

列描述：

ID: 行号
 USER: 用户
 FREQUENCY: sql曾被执行次数
 AVG_TIME: 平均执行耗时
 MAX_TIME: 最大执行耗时
 MIN_TIME: 最小执行耗时
 EXECUTE_TIME: 最近一次执行耗时
 LAST_TIME: 最近一次执行时间戳
 SQL: 略

类似查询SQL：

```
select sql_id as ID, user as USER, count(0) as FREQUENCY, avg(duration) AVG_TIME, max(duration) as MAX_TIME, min(duration) as MIN_TIME, duration as EXECUTE_TIME, start_time as LAST_TIME, sql_digest as SQL from dble_information.sql_log group by sql_digest,user order by start_time
```

2.1.3.21 show @@sql.slow

show @@sql.slow;

描述：取sql_log表中执行时间超过 sqlSlowTime (默认100ms, 可动态修改)的sql

例：

```
mysql> show @@sql.slow;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+ | USER | START_TIME           | EXECUTE_TIME | SQL
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+ | root | 2017/10/17 17:37:22 |          381 | select * from sharding_two_node LIMIT 100 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+
1 row in set (0.07 sec)
```

列描述：

USER: 用户
 START_TIME: 上次接收请求时间戳
 EXECUTE_TIME: 响应时间
 SQL: 略

类似查询SQL：

```
select user as USER, start_time as START_TIME, duration as EXECUTE_TIME, sql_stmt as SQL from dble_information.sql_log where duration >= ${slowTime}*1000000 order by start_time
```

2.1.3.22 show @@sql.resultset

show @@sql.resultset;

描述：取sql_log表中查询sql执行的频率和结果集大小超过 maxResultSet (默认512K)的sql情况

例：

```
mysql> show @@sql.resultset;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID   | USER | FREQUENCY | SQL                               | RESULTSET_SIZE |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | root | 1         | SELECT * FROM sharding_two_node | 1048576        |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.05 sec)
```

列描述:

ID:行号
 USER: 用户
 FREQUENCY:sql曾被执行次数
 SQL: 略
 RESULTSET_SIZE:最近一次的结果集大小

类似查询SQL:

```
select sql_id as ID, user as USER, t2.FREQUENCY, sql_stmt as SQL, result_size as RESULT_SIZE from dble_information.sql_log t1
inner join (select max(sql_id) as maxId, count(0) as FREQUENCY from dble_information.sql_log group by sql_digest having result_size >= ${m
axResultSet} order by maxId) t2 on t1.sql_id = t2.maxId;
```

2.1.3.23 show @@sql.sum

show @@sql.sum;

描述: 展示用户的sql执行情况, 是否带.user结果是一样的.带参数true, 表示查询结束后清空已经缓存的结果

例:

```
mysql> show @@sql.sum;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID   | USER | R   | W   | R%   | MAX | NET_IN | NET_OUT | TIME_COUNT | TTL_COUNT | LAST_TIME          |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | root | 1   | 0   | 1.00 | 1   | 41    | 840    | [0, 0, 1, 0] | [0, 0, 1, 0] | 2017/10/17 17:37:23 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.26 sec)
```

列描述:

ID:行号
 USER:用户
 R:读的次数(select、show、desc、explain)
 W:写的次数(insert、update、delete、DDL等sql)
 R%:R/(R+W)*100%
 MAX:最大并发数
 NET_IN:网络流入量
 NET_OUT:网络流出量
 TIME_COUNT:query在四个时间区间的个数分布, 四个区间分别是前一天22-06 夜间,06-13 上午,13-18下午,18-22 晚间
 TTL_COUNT:query耗时在四个时间级别内的个数分布, 四个区间分别是10毫秒内,10 - 200毫秒内,1秒内,超过 1秒
 LAST_TIME:上次SQL执行时间戳

如果需要在查询后重置统计, 执行:

```
show @@sql.sum true;
```

2.1.3.24 show @@sql.sum.user

等同于:

```
show @@sql.sum;
```

如果需要在查询后重置统计, 执行:

```
show @@sql.sum.user true;
```

2.1.3.25 show @@sql.sum.table

show @@sql.sum.table;

描述: 展示各个表的读写情况

例:

```
mysql> show @@sql.sum.table;
+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | TABLE | R | W | R% | RELATABLE | RELACOUNT | LAST_TIME |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | sharding_two_node | 1 | 0 | 1.00 | NULL | NULL | 2017/10/17 17:37:23 |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.06 sec)
```

列描述:

ID:行号
 TABLE:表名
 R:读的次数(select、show、desc、explain)
 W:写的次数(insert、update、delete、DDL等sql)
 R%:R/(R+W)*100%
 RELATABLE:关联表的名称(目前拆分表关联查询都使用查询计划树,此值为NULL)
 RELACOUNT:关联表的个数(目前拆分表关联查询都使用查询计划树,此值为NULL)
 LAST_TIME:上次SQL执行时间戳

如果需要在查询后重置统计,执行:

```
show @@sql.sum.table true;
```

2.1.3.26 show @@heartbeat

show @@heartbeat;

描述:展示dbinstance的heartbeat信息

例:

```
mysql> show @@heartbeat;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| NAME | HOST | PORT | RS_CODE | RETRY | STATUS | TIMEOUT | EXECUTE_TIME | LAST_ACTIVE_TIME | STOP | RS_MESSAGE |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| hostM1 | 10.18x.2x.63 | 3320 | OK | 0 | idle | 0 | 8,8,8 | NULL | false | NULL |
| hostM2 | 10.18x.2x.64 | 3320 | OK | 0 | idle | 0 | 9,9,9 | NULL | false | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.07 sec)
```

列描述:

NAME:dbGroup名称
 HOST:主机名/IP
 PORT:端口
 RS_CODE:状态码,有以下四种状态:INIT, OK, ERROR, TIMEOUT
 RETRY:重试错误次数
 STATUS:checking/idle
 TIMEOUT:心跳超时阈值(来源于db.xml中heartbeat子元素timeout的值,默认为0)
 EXECUTE_TIME:最近3个时段的平均响应时间,默认1, 10, 30分钟
 LAST_ACTIVE_TIME:上次收到心跳回复时间戳
 STOP:是否stop,和stop命令相关
 RS_MESSAGE:心跳失败信息,当RS_CODE为INIT, OK, TIMEOUT时,message为null,只有当RS_CODE为ERROR时,message才会显示最近一次心跳失败的信息

2.1.3.27 show @@heartbeat.detail where name=?

show @@heartbeat.detail where name=xxx;

其中,xxx为要查询的dbinstance的名字。

描述:展示指定dbinstance的heartbeat的详细信息

前提条件:至少发生过一次心跳语句(与shardingNodeHeartbeatPeriod相关)

例:

```
mysql> show @@heartbeat.detail where name='hostM1';
+-----+-----+-----+-----+-----+
| NAME | HOST | PORT | TIME | EXECUTE_TIME |
+-----+-----+-----+-----+-----+
| hostM1 | 10.18x.2x.63 | 3320 | 2017-10-17 17:31:58 | 7 |
| hostM1 | 10.18x.2x.63 | 3320 | 2017-10-17 17:32:59 | 9 |
+-----+-----+-----+-----+-----+
2 row in set (0.00 sec)
```

列描述:

```

NAME:dbGroup名称
HOST:主机名/IP
PORT:端口
TIME:收到心跳时间戳
EXECUTE_TIME:心跳执行耗时(毫秒)

```

2.1.3.28 show @@sysparam

show @@sysparam;

描述：展示sysconfig参数配置

结果：略

2.1.3.29 show @@white

show @@white;

描述：展示配置的白名单信息

例：

```

mysql> show @@white;
+-----+-----+
| IP          | USER |
+-----+-----+
| 0:0:0:0:0:0:1 | root |
| 127.0.0.1    | root |
| 0:0:0:0:0:0:1 | test |
| 127.0.0.1    | test |
+-----+-----+
4 rows in set (0.00 sec)

```

列描述：

略

2.1.3.30 show @@directmemory

show @@directmemory;

描述：堆外内存使用总览

结果集举例：

```

+-----+-----+-----+
| DIRECT_MEMORY_MAXED | DIRECT_MEMORY_POOL_SIZE | DIRECT_MEMORY_POOL_USED |
+-----+-----+-----+
| 3GB                 | 1024MB                   | 44KB                     |
+-----+-----+-----+
1 row in set (0.16 sec)

```

结果列描述：

```

DIRECT_MEMORY_MAXED:通过-XX:MaxDirectMemorySize设置的值
DIRECT_MEMORY_POOL_SIZE: 内存池的大小, 等于bufferPoolPageSize和bufferPoolPageNumber的乘积
DIRECT_MEMORY_POOL_USED: 已经使用的内存池中的DirectMemory内存

```

2.1.3.31 show @@command.count

show @@command.count;

描述：查询当前系统的查询数；

结果：略

2.1.3.32 show @@connection.count

show @@connection.count;

描述：查询当前的前端链接数；

结果：略

2.1.3.33 show @@backend.statistics

show @@backend.statistics;

描述：查询系统中进程的后端数据源信息；

例:

```
MySQL [(none)]> show @@backend.statistics;
+-----+-----+-----+-----+
| HOST          | PORT  | ACTIVE | TOTAL |
+-----+-----+-----+-----+
| 192.168.2.177 | 3307  | 0      | 10    |
| 192.168.2.177 | 3308  | 0      | 10    |
+-----+-----+-----+-----+
2 rows in set (0.02 sec)
```

列描述:

HOST: 数据源的ip
 PORT: 数据源的端口
 ACTIVE: 数据源正在被使用的链接数
 TOTAL : 活的后端链接数。

2.1.3.34 show @@backend.old

show @@backend.old;

描述: reload @@config_all之后待回收的活动的后端链接信息

结果格式: 同show @@backend

2.1.3.35 show @@binlog.status

show @@binlog.status;

描述: 对被分库分表 (sharding.xml) 使用的mysql节点拉一条一致性的binlog线。

例:

```
mysql> show @@binlog.status;
+-----+-----+-----+-----+-----+-----+-----+
| Url          | File          | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+-----+-----+-----+
| 10.18x.2x.63:3320 | mysql-bin.000024 | 14128    |              |                  | 7ad71aab-de94-11e5-9488-3a935460da28:1-67646 |
| 10.18x.2x.64:3320 | mysql-bin.000049 | 604440   |              |                  | ba8f8b5c-debf-11e5-a87b-26b8a61f9012:1-91 |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.11 sec)
```

列描述:

Url: 后端节点的连接Url值
 其余列: 等同于在对应节点上执行show master status的结果。

2.1.3.36 show @@help

show @@help;

描述: 展示帮助信息

结果: 略

2.1.3.37 show @@sql.large

show @@sql.large;

描述: 取sql_log表中查询select sql的结果集行数超过10000条的

例:

```
mysql> show @@sql.large;
+-----+-----+-----+-----+-----+
| USER | ROWS | START_TIME          | EXECUTE_TIME | SQL |
+-----+-----+-----+-----+-----+
| root | 20000 | 2017/10/17 17:37:23 | 381          | SELECT * FROM sharding_two_node LIMIT ? |
+-----+-----+-----+-----+-----+
1 row in set (0.06 sec)
```

列描述:

USER: 用户
 ROWS: 该查询的行数
 START_TIME: 上次接收请求时间戳 EXECUTE_TIME: 响应时间
 SQL: 略

类似查询SQL:

```
select user as USER, rows as ROWS, start_time as START_TIME, duration as EXECUTE_TIME, sql_stmt as SQL from dble_information.sql_log where sql_type='Select' and rows > 10000;
```

2.1.3.38 show @@sql.condition

show @@sql.condition;

描述: 查询条件统计, 需要配合reload @@query_cf 使用, 前者设置了table&column后, 运行此语句后展示sql查询条件统计信息。(最多100000条, 超出后不再统计)

比如select from sharding_two_node where id =0; 和select from sharding_two_node where id =1;

例:

```
mysql> show @@sql.condition;
+-----+-----+-----+-----+
| ID | KEY | VALUE | COUNT |
+-----+-----+-----+-----+
| 2 | sharding_two_node.id | 0 | 1 |
| 3 | sharding_two_node.id | 1 | 2 |
| 2 | sharding_two_node.id.valuekey | size | 2 |
| 3 | sharding_two_node.id.valuecount | total | 3 |
+-----+-----+-----+-----+
4 rows in set (0.05 sec)
```

列描述:

ID: 行号
 KEY: schema.table 最后两行为schema.table.valuekey 和 schema.table.valuecount
 VALUE: 对应key的value值
 COUNT: 查询的次数

2.1.3.39 show @@cost_time;

show @@cost_time;

描述: 查询query耗时统计的结果, 需要在bootstrap.cnf中开启useCostTimeStat选项之后才会有统计结果

例:

```
mysql> show @@cost_time;
+-----+-----+-----+
| OVER_ALL(us) | FRONT_PREPARE | BACKEND_EXECUTE |
+-----+-----+-----+
| 71496 | Id:9, Time:53135; Id:12, Time:54056 | Id:9, Time:16924; Id:12, Time:16006 |
| 15316 | Id:17, Time:2301; Id:11, Time:3196 | Id:17, Time:10691; Id:11, Time:11397 |
+-----+-----+-----+
2 rows in set (0.05 sec)
```

列描述:

OVER_ALL: 总耗时
 FRONT_PREPARE: 前端连接以及dble中的耗时
 BACKEND_EXECUTE: 后端连接执行耗时

2.1.3.40 show @@shardingNodes where schema=? and table=?;

show @@shardingNodes

描述: 查询某具体表格的节点信息

例:

```
mysql> show @@shardingNodes where schema=testdb and table=seqtest;
+-----+-----+-----+-----+-----+-----+-----+
| NAME | SEQUENCE | HOST          | PORT | PHYSICAL_SCHEMA | USER | PASSWORD |
+-----+-----+-----+-----+-----+-----+-----+
| dn1  | 0        | 10.186.24.113 | 3309 | db1              | root | 123456   |
| dn2  | 1        | 10.186.24.113 | 3309 | db2              | root | 123456   |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.05 sec)
```

列描述:

NAME: 节点名称
 SEQUENCE: 节点编号
 HOST: 节点所在的IP
 PORT: 节点对应的服务端口
 PHYSICAL_SCHEMA: 节点所对应的物理库
 USER: 节点连接的用户
 PASSWORD: 节点连接的密码

2.1.3.41 show @@algorithm where schema=? and table=?;

show @@algorithm

描述: 查询某具体表格的分片算法信息, 由于不同算法会有不同的分片参数以及辅助文件及数据, 所以不同算法表格的输出分片事项都不相同

例:

```
mysql> show @@algorithm where schema=testdb and table=seqtest;
+-----+-----+-----+-----+-----+
| KEY          | VALUE          |
+-----+-----+-----+-----+
| TYPE        | SHARDING TABLE |
| COLUMN      | ID              |
| CLASS       | com.actiontech.dble.route.function.PartitionByLong |
| partitionCount | 2              |
| partitionLength | 1              |
+-----+-----+-----+-----+
5 rows in set (0.05 sec)
```

行描述:

KEY: 分片事项
 VALUE: 详细信息

2.1.3.42 show @@thread_used;

show @@thread_used;

描述: 查看各个主要业务处理线程的使用状况

例:

```
mysql> show @@thread_used;
+-----+-----+-----+-----+
| THREAD_NAME          | LAST_QUARTER_MIN | LAST_MINUTE | LAST_FIVE_MINUTE |
+-----+-----+-----+-----+
| 0-NIOBackendRW      | 0%                | 0%          | 0%                |
| 0-NIOFrontRW        | 0%                | 0%          | 0%                |
| 0-backendWorker     | 0%                | 0%          | 0%                |
| 0-frontWorker       | 0%                | 0%          | 0%                |
| 0-managerFrontWorker | 0%                | 0%          | 0%                |
| 0-writeToBackendWorker | 0%                | 0%          | 0%                |
| 1-backendWorker     | 0%                | 0%          | 0%                |
| 1-frontWorker       | 0%                | 0%          | 0%                |
| 1-managerFrontWorker | 0%                | 0%          | 0%                |
+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

行描述:

THREAD_NAME: 线程名称
 LAST_QUARTER_MIN: 最近15秒使用率
 LAST_MINUTE: 最近一分钟使用率
 LAST_FIVE_MINUTE: 最近五分钟使用率

2.1.3.43 show @@ddl;

show @@ddl;

描述：查看正在执行，没有在dble内部释放锁的DDL

例：

```
mysql> show @@ddl;
+-----+-----+-----+-----+
| Schema | Table           | Sql                                     |
+-----+-----+-----+-----+
| testdb | sharding_two_node | alter table sharding_two_node add column id2 int |
| mytest | sharding_four_node | drop table sharding_four_node                |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

行描述：

Schema: Schema名称
Table: Table名称
Sql: ddl sql语句

2.1.3.44 show @@processlist;

show @@processlist;

描述：查看前端连接和后端连接对应关系，若前端连接没有对应的后端连接，显示NULL。

例：

```
mysql> show @@processlist;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Front_Id | db_instance | MysqlId | User | Front_Host | db | Command | Time | State | Info |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | instanceM2 | 2303 | root | 127.0.0.1:33222 | db2 | Sleep | 17 | | NULL |
| 2 | instanceM2 | NULL | man1 | 127.0.0.1:34882 | NULL | NULL | 0 | | NULL |
| 3 | instances2 | 2259 | root | 127.0.0.1:33226 | db1 | Sleep | 4 | | NULL |
| 3 | instances2 | 2308 | root | 127.0.0.1:33226 | db2 | Sleep | 4 | | NULL |
| 3 | instances2 | 2304 | root | 127.0.0.1:33226 | db1 | Sleep | 4 | | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
5 rows in set (0.05 sec)
```

行描述：

Front_Id: 前端连接ID
db_instance: 前端连接下发操作的实例名
MysqlId: 后端连接对应的 mysql 线程ID
User: 用户名
Front_Host: 客户端主机名
db: 后端连接默认数据库，来自于 mysql 'show processlist' 字段 db
Command: mysql线程正在执行的指令类型，来自于 mysql 'show processlist' 字段 Command
Time: mysql线程处于当前state的时间，来自于 mysql 'show processlist' 字段 Time
State: mysql线程执行状态，来自于 mysql 'show processlist' 字段 State
Info: mysql线程执行语句，来自于 mysql 'show processlist' 字段 Info

2.1.3.45 show @@session.xa;

show @@session.xa;

描述：查看后台重试的xa事务信息。

例：

```
mysql> show @@session.xa;
+-----+-----+-----+-----+
| FRONT_ID | XA_ID           | XA_STATE           | SHARDING_NODES |
+-----+-----+-----+-----+
| 1 | 'Dble_Server.1.1' | TX_COMMIT_FAILED_STATE | dn1,dn3         |
+-----+-----+-----+-----+
1 rows in set (0.00 sec)
```

行描述：

FRONT_ID: 前端连接ID
 XA_ID: xa事务id
 XA_STATE: xa事务状态
 SHARDING_NODES: xa提交失败的shardingNode名称

2.1.3.46 show @@reload_status

show @@reload_status

描述: 查看dble中最近的reload信息

举例

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| INDEX | CLUSTER | RELOAD_TYPE | RELOAD_STATUS | LAST_RELOAD_START | LAST_RELOAD_END | TRIGGER_TYPE | END_TYPE |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 | No Cluster | RELOAD_ALL | NOT_RELOADING | 2020/06/19 14:28:04 | 2020/06/19 14:28:05 | LOCAL_COMMAND | RELOAD_END |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

行描述:

INDEX: reload对应的编号, 能与日志中的[RL]日志编号相对应
 CLUSTER: 当前dble使用的集群方式
 RELOAD_TYPE: 最近的reload的类型 RELOAD_ALL/RELOAD_META/MANAGER_INSERT/MANAGER_UPDATE/MANAGER_DELETE
 RELOAD_STATUS: 最近一次reload的执行状态not_reloading/self_reload/meta_reload/waiting_others
 LAST_RELOAD_START: 起始时间
 LAST_RELOAD_END: 结束时间
 TRIGGER_TYPE: 触发类型 LOCAL_COMMAND/CLUSTER_NOTIFY
 END_TYPE: 结束原因 RELOAD_END/INTERRUPTED

此命令被用于配合命令[release @@reload_metadata](#)

2.1.3.47 show @@user

show @@user

描述: 查看dble 所有用户

举例

```
mysql> show @@user;
+-----+-----+-----+-----+
| Username | Manager | Readonly | Max_con |
+-----+-----+-----+-----+
| man1     | Y       | N        | no limit |
| root     | N       | N        | no limit |
| user     | N       | N        | no limit |
+-----+-----+-----+-----+
3 rows in set (0.03 sec)
```

行描述:

Username: 用户名
 Manager: 是否是管理用户
 Readonly: 是否是只读用户
 Max_con: 最大连接数

2.1.3.48 show @@user.privilege

show @@user.privilege

描述: 查看dble 用户的权限信息, 不包含管理用户

举例

```
mysql> show @@user.privilege;
+-----+-----+-----+-----+-----+-----+-----+
| Username | Schema | Table | INSERT | UPDATE | SELECT | DELETE |
+-----+-----+-----+-----+-----+-----+-----+
| root     | testdb1 | *     | Y      | Y      | Y      | Y      |
| root     | testdb  | *     | Y      | Y      | Y      | Y      |
| user     | testdb  | *     | N      | Y      | Y      | N      |
+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

行描述:

```
Username: 用户名
Schema: 用户授权逻辑库
Table: 用户显式指定dml权限的表名, 未指定的其他表使用*表示
INSERT: 插入权限位
UPDATE: 更新权限位
SELECT: 查询权限位
DELETE: 删除权限位
```

2.1.3.49 show @@data_distribution where table ='schema.table'

show @@data_distribution where table ='schema.table'

描述: 查看某个表在各个节点上的数据分布情况

举例

```
+-----+-----+
| SHARDING_NODE | COUNT |
+-----+-----+
| dn1           | 100   |
| dn2           | 101   |
| dn3           | 98    |
| dn4           | 104   |
+-----+-----+
4 rows in set (0.09 sec)
```

行描述:

```
SHARDING_NODE: 数据结点名字
COUNT: 数据量
```

2.1.3.50 show @@Questions

show @@Questions

描述: 查看自启动之后SQL服务端口执行的QUERY和Transaction数量

举例:

```
mysql> show @@Questions;
+-----+-----+
| Questions | Transactions |
+-----+-----+
|          0 |             0 |
+-----+-----+
```

行描述:

```
Questions: 收到的查询的数量
Transactions: 执行事务的数量(非事务查询算单语句事务)
```

Transactions统计规则:

1. 非事务中执行sql, 只要有响应结果都参与统计; 事务中, 执行含有隐式提交语句报错1064、1046错误码, 不参与统计
2. 事务中, 执行exit(隐式rollback)表示事务结束, 参与统计
3. set多个变量时, 如'set autocommit=0,autocommit=n,xxxx, xxxx;' 一律取最右边的autocommit=n, 根据实际场景实际处理 (若无autocommit=n则视为普通sql处理)
4. sharding中执行'set xa = on/off/1/0'语句和rwsplit中执行的'XA start/end/prepare/commit/rollback XXX'都被视为普通sql
5. rwsplit中执行multi-query(指一次执行多个sql,mysql client可使用delimiter关键字实现), multi-query将会直接透传至后端节点, 这里会被视作为事务级sql(如commit), 参与统计(实际上Transactions只+1, Questions只+1)

2.1.3.51 show @@connection_pool

show @@connection_pool

描述: 查看后端连接池的各种属性

举例:

```
mysql> show @@connection_pool;
+-----+-----+-----+-----+
| DB_GROUP | DB_INSTANCE | PROPERTY | VALUE |
+-----+-----+-----+-----+
| dbGroup1 | instanceM1 | minCon   | 2     |
| dbGroup1 | instanceM1 | maxCon   | 4     |
| dbGroup1 | instanceM1 | testOnCreate | false |
| dbGroup1 | instanceM1 | testOnBorrow | false |
| dbGroup1 | instanceM1 | testOnReturn | false |
| dbGroup1 | instanceM1 | testWhileIdle | false |
| dbGroup1 | instanceM1 | connectionHeartbeatTimeout | 20 |
| dbGroup1 | instanceM1 | connectionTimeout | 10000 |
| dbGroup1 | instanceM1 | heartbeatPeriodMillis | 10000 |
| dbGroup1 | instanceM1 | idleTimeout | 600000 |
| dbGroup1 | instanceM1 | evictorShutdownTimeoutMillis | 10000 |
| dbGroup1 | instanceM1 | timeBetweenEvictionRunsMillis | 30000 |
+-----+-----+-----+-----+
12 rows in set (0.01 sec)
```

行描述:

```
DB_GROUP: dbinstance所属DB_GROUP
DB_INSTANCE: dbinstance名
PROPERTY: 属性名
VALUE: 属性值
```

2.1.3.52 show @@cap_client_found_rows

show @@cap_client_found_rows

描述: 查看cap_client_found_rows权能标志

举例:

```
mysql> show @@cap_client_found_rows;
+-----+
| @@cap_client_found_rows |
+-----+
| 0 |
+-----+
1 row in set (0.02 sec)
```

值描述:

```
0-关闭, 1-开启
```

2.1.3.53 show @@general_log

show @@general_log

描述: 查看general相关信息

举例:

```
mysql> show @@general_log;
+-----+-----+
| NAME | VALUE |
+-----+-----+
| general_log | ON |
| general_log_file | /tmp/dble-general/general/general.log |
+-----+-----+
2 rows in set (0.02 sec)
```

值描述:

```
general_log: 关闭/启用
general_log_file: 显示general日志的绝对路径
```

2.1.3.54 show @@statistic;

show @@statistic;

描述: 查看sql statistic相关信息

举例:

```
mysql> show @@statistic;
+-----+-----+
| NAME                | VALUE |
+-----+-----+
| statistic            | OFF   |
| statisticAnalysis    | OFF   |
| associateTablesByEntryByUserTableSize | 1024  |
| frontendByBackendByEntryByUserTableSize | 1024  |
| tableByUserByEntryTableSize | 1024  |
| samplingRate         | 0     |
| sqlLogTableSize      | 1024  |
| queueMonitor         | monitoring |
+-----+-----+
6 rows in set (0.01 sec)
```

值描述:

```
statistic: 是否开启sql全量统计
statisticAnalysis: 是否开启sql对用户、table、condition聚合统计
associateTablesByEntryByUserTableSize: sql_statistic_by_associate_tables_by_entry_by_user表格大小
frontendByBackendByEntryByUserTableSize: sql_statistic_by_frontend_by_backend_by_entry_by_user表格大小
tableByUserByEntryTableSize: sql_statistic_by_table_by_user_by_entry表格大小
samplingRate: 采样统计的采样率, 采样率为0的话表示关闭采样统计。采样率是[0,100]之间的整数, 单位是 %。
sqlLogTableSize: sql_log 表格存储事务个数大小
queueMonitor: 观测队列状态
```

2.1.3.55 show @@load_data.fail;

show @@load_data.fail;

描述: load data批处理模式下查询本次失败的文件

举例:

```
show @@load_data.fail;
Empty set (0.01 sec)

if have error file may like
show @@load_data.fail;
+-----+-----+
| error_load_data_file |
+-----+-----+
| ./temp/error/1-data-table-dn1.txt |
| ./temp/error/1-data-table-dn2.txt |
+-----+-----+
2 rows in set (0.01 sec)
```

值描述:

```
error_load_data_file: 错误文件地址
```

2.1.3.56 show @@statistic_queue.usage;

show @@statistic_queue.usage;

描述: 查看队列的使用率情况列表(观测期间, 每次查询结果递增)

举例:

```
show @@statistic_queue.usage;
+-----+-----+
| TIME                | USAGE |
+-----+-----+
| 2021-05-31 16:33:30 | 0.00% |
| 2021-05-31 16:33:35 | 0.00% |
| 2021-05-31 16:33:40 | 0.00% |
+-----+-----+
3 rows in set (0.01 sec)
```

值描述:

```
TIME: 采样时间点
USAGE: 使用率
```

2.1.4 switch命令

本小节部分自2.20.04.0 版本起废弃

2.1.5 kill命令

2.1.5.1 kill @@connection;

kill @@connection id1,id2,...;

其中, idx为 前端连接id值, 可以通过show @@connection 获取。

描述: 关闭存在的前端链接, 对不存在的前端链接, 不会报错

结果: 返回OK, 关闭的前端链接数。

2.1.5.2 kill @@xa_session;

kill @@xa_session id1,id2,...;

其中, idx为 session id值, 可以通过show @@session.xa 获取。

描述: 取消指定session后台重试xa事务, 对不存在的session, 不会报错

结果: 返回OK, 取消的session数量。

2.1.5.3 KILL @@DDL_LOCK where schema=? and table=?

KILL @@DDL_LOCK where schema=? and table=?;

描述: 释放指定schema下table的ddl锁, 详细描述可参考 [2.22 KILL @@DDL_LOCK](#)

结果: 返回OK。

2.1.5.4 kill @@load_data

kill @@load_data;

描述: 导入文件如果发生回滚, 再次导入该文件时不再跳过已经成功导入的数据会从头重新导入数据

结果: 返回OK。

2.1.5.6 kill @@cluster_renew_thread '?'

kill @@cluster_renew_thread '?'

描述: 当集群模式为ucore(clusterMode=ucore)时, 执行分布式操作, Dble内部会起新的renewThread线程, 目的以续约分布式锁, 保证锁不超时

结果: 返回OK。

2.1.6 stop命令

2.1.6.1 stop @@heartbeat

stop @@heartbeat keys:dbGroup;

其中：keys:dbGroup名字列表，可以是多个，用逗号隔开的key；该命令停止的是每个dbGroup的写实例。**key**:可以直接是dbGroup，也可以是dbGroup\$0-n（如dbGroup\$0-2实际会输出dbGroup[0],dbGroup[1],dbGroup[2]，疑似BUG）；

value: 应当是个整数，单位毫秒

描述：设置dbGroup名为key的host停止heartbeat n秒

结果：返回OK

注意：未作异常处理，慎用

2.1.7 reload 命令

2.1.7.1 reload @@config

reload @@config;

2.19.09.0(不含)版本以前请参考对应章节文档，2.19.09.0版本之后功能完全等同于reload @@config_all

2.1.7.2 reload @@config_all

reload @@config_all [-s] [-f] [-r];

描述：重新加载所有配置，涉及user.xml，db.xml，sharding.xml 内容，

-s 在测试链接阶段，后端链接不可用时不会终止reload的执行，只会在日志中输出相关错误信息。默认不加之参数时遇到后端不可用的情况，会终止reload的执行并返回报错。

-f 关闭所有变更的dbGroup（加-r参数所有dbGroup会被视为变更）相关的处于事务中的前端链接,如果无此参数默认仅将相应后端链接放入旧链接池。

-r 不做智能判断,将所有后端连接池全部重新加载一遍。不加之参数时，将对新配置进行智能判断，只会对增删改的连接池做变更，不影响未作变更的连接池
更多细节 参考 [2.19 智能计算reload_all](#)

结果：OK 或者ERROR

相关影响：当执行此命令时，当有以下情况发生时，涉及到的表的meta信息会被重载，否则保持原有表meta信息。

- 有新增的表
- 有删除的表
- 表的shardingNode或者类别发生变更
- 表的shardingNode对应的物理节点或者对应的dbGroup/dbInstance发生变更
- 有新增的schema
- 有删除的schema
- schema 的默认shardingNode属性发生变更
- schema的shardingNode对应的物理节点或者对应的dbGroup/dbInstance发生变更。

另外，如果包含-r参数则不做上述判断，全部重新加载meta数据。

如果不包含-r但是包含-s参数,则对metadata是否需要重新加载的计算时，忽略所有dbGroup/dbInstance的变更

注意,不能在配置变更中体现的某些变化是无法重新加载metadata的，举例[#1002](#)

一个带有默认shardingNode的schema尝试通过删除配置将拆分表或者global表变成非拆分表是不符合规范的。应当避免这种操作。

注意：如果使用默认的切换方式(即单实例部署并且system的outerHA属性为false), 需要做配置的重载时,需要人工保证流程是标准的,否则可能导致切换功能故障,具体请参看相关章节[2.12 故障切换](#)。

2.1.7.3 reload @@metadata

reload @@metadata;

描述：重新加载所有元数据信息。

结果：返回OK。

支持过滤表达式

reload @@metadata where schema=? [and table=?]

描述：重新加载指定schema中所有表或指定表的元数据信息。

结果：返回OK。

reload @@metadata where table in ('schema1.table1','schema2.table2','schema1.table3',...)

描述：重新加载schema1中table1,table3和schema2中table2的元数据信息。

结果：返回OK。

2.1.7.4 reload @@sqlslow=N;

reload @@sqlslow=N;

描述：设定用户分析统计的slow sql时间阈值到N毫秒； 结果：OK

2.1.7.5 reload @@user_stat (已废弃)

reload @@user_stat;

自3.23.08.0 版本起废弃此命令

2.1.7.6 reload @@query_cf

reload @@query_cf[=table&column];

其中，table为要统计的目标表的表名，column为目标表中目标列的列名。

描述：重设show @@sql.condition要统计的条件。

结果：返回OK

如果要清除查询条件统计表列的设置执行命令：

```
reload @@query_cf;
```

或者

```
reload @@query_cf=NULL;
```

2.1.7.7 reload @@general_log_file=?

```
reload @@general_log_file = 'general/general.log';
```

描述：重设general日志路径；若设置以/开头的值则作为绝对路径生效，反之，则在homepath后拼接值得到最终绝对路径且生效。

结果：返回OK

2.1.7.8 reload @@statistic_table_size = ? [where table='?' | where table in (dble_information.tableA,...)]

```
reload @@statistic_table_size = 90;
```

描述：将统计表格（sql_statistic_by_frontend_by_backend_by_entry_by_user、sql_statistic_by_table_by_user_by_entry、sql_statistic_by_associate_tables_by_entry_by_user）的大小设置为90。

结果：返回OK

```
reload @@statistic_table_size = 90 where table = 'sql_statistic_by_table_by_user_by_entry'
```

描述：将统计表格sql_statistic_by_table_by_user_by_entry的大小设置为90。

结果：返回OK

```
reload @@statistic_table_size = 90 where table
in(sql_statistic_by_table_by_user_by_entry,sql_statistic_by_associate_tables_by_entry_by_user)
```

描述：将统计表格sql_statistic_by_table_by_user_by_entry、sql_statistic_by_associate_tables_by_entry_by_user的大小设置为90。

结果：返回OK

```
reload @@statistic_table_size = 90 where table='sql_log'
```

描述：将采样统计统计表格sql_log的大小设置为90。

结果：返回OK

2.1.7.9 reload @@samplingRate = ?

描述：设置采样统计的采样率，采样率为0的话表示关闭采样统计。采样率是[0,100]之间的整数，单位是 %。

2.1.7.10 reload @@load_data.num=N

```
reload @@load_data.num=N
```

描述：N代表设置maxRowSizeToFile的值，不开启批处理模式代表文件中的sql语句条数达到该阈值就会把该文件进行存储，开启批处理模式会将该文件按照阈值拆分成多个文件。

结果：返回OK

2.1.7.11 reload @@xaldCheck.period=N

```
reload @@xaldCheck.period=N
```

描述：N>0时，表示开启疑似残留XA检测，且作为定时检测任务的周期(以s为单位)；N<=0时,表示停止或者不开启定时检测；如：N=60，表示开启定时检测，每隔60s进行检测疑似残留XA。

结果：返回OK

2.1.9 offline命令

offline;

描述：设置dble处于离线状态, 之后外部对dble的ping命令, select user, 心跳都会返回错误。

问题：其他查询未处理, 合理性需要考量

结果：OK

2.1.10 online命令

online;

描述：设置dble处于在线状态，与offline相反

结果：OK

2.1.13 配置检查命令dryrun

背景；当我们修改配置之后并且reload之前，可以通过dryrun来检查配置的正确性。

举例，当这样一个sharding.xml 准备reload时，先观察dryrun结果： sharding.xml:

```
<?xml version="1.0"?>
<db!e:sharding xmlns:db!e="http://db!e.cloud/">
  <schema name="testdb" sqlMaxLimit="100" >
    <shardingTable name="sharding_two_node" shardingNode="dn1, dn2" shardingColumn="id" function="two-long" />
    <shardingTable name="sharding_two_node2" shardingNode="dn1, dn2" shardingColumn="id" function="two-long" />
    <shardingTable name="sharding_two_node3" shardingNode="dn1, dn2" shardingColumn="id" function="two-long" />
    <shardingTable name="sharding_four_node" shardingNode="dn1, dn2, dn3, dn4" shardingColumn="id" function="rule_simple" />
    <globalTable name="test_table" shardingNode="dn$1-2"/>
    <shardingTable name="a_test" shardingNode="dn1, dn2, dn3, dn4" shardingColumn="id" function="rule_simple" />
    <shardingTable name="a_order" shardingNode="dn1, dn2, dn3, dn4" shardingColumn="id" function="rule_simple" />
    <shardingTable name="test_shard" shardingNode="dn1, dn2, dn3, dn4" shardingColumn="id" function="rule_simple" />
    <globalTable name="test_global" shardingNode="dn1, dn2, dn3, dn4"/>
    <shardingTable name="sbtest1" shardingNode="dn1, dn2, dn3, dn4" shardingColumn="id" function="rule_simple" />
  </schema>
  <schema name="nosharding_test" sqlMaxLimit="100" shardingNode="dn5">
  </schema>
  <shardingNode name="dn1" dbGroup="dh1" database="ares_test" />
  <shardingNode name="dn2" dbGroup="dh2" database="db!e_test" />
  <shardingNode name="dn3" dbGroup="dh1" database="db!e_test" />
  <shardingNode name="dn4" dbGroup="dh2" database="db!e_test" />
  <shardingNode name="dn5" dbGroup="dh1" database="nosharding" />
  <shardingNode name="dn8" dbGroup="dh1" database="xxxxxxx" />

  <function name="rule_simple" class="Hash">
    <property name="partitionCount">4</property>
    <property name="partitionLength">1</property>
  </function>
</db!e:sharding>
<function class="Hash" name="two-long">
  <property name="partitionCount">2</property>
  <property name="partitionLength">1</property>
</function>
```

dryrun结果如下：

```
mysql> dryrun;
+-----+-----+-----+
| TYPE | LEVEL | DETAIL |
+-----+-----+-----+
| Xml | WARNING | shardingNode dn9 is useless |
| Xml | WARNING | shardingNode dn8 is useless |
+-----+-----+-----+
2 rows in set (0.58 sec)
```

列名含义：

TYPE: 错误类型，比如XML表示xml配置错误，BACKEND表示后端连接错误

LEVEL: 错误级别：分为WARNING 和ERROR表，一般来说WARNING错误不影响启动和使用，但需要注意。

DETAIL : 错误详情

2.1.14 shardingNode级别的流量暂停和恢复功能

背景：当我们做部分shardingNode的拆分或者是数据重分片时，有时希望不影响到其他无关的业务所涉及到的dbGroup和shardingNode，所以希望只停下部分shardingNode的流量。

2.1.14.0 功能描述

pause功能会在dble不停止的状态下停止对于指定后端节点的流量，在暂停期间的所有涉及到节点的路由结果会被挂起（为防止挂起的连接过多，queue和wait_limit参数会控制挂起连接的数量和时间），直到恢复命令resume被执行之后,之前被挂起的查询才会继续进行

pause的执行不一定成功，pause命令将会先获取分布式锁，然后在指定的timeout的时间内等待需要暂停的流量上的所有事务或者正在执行的sql结束，当目标节点上来自dble的sql或者事务一直无法在指定时间内结束的时候，本次暂停会返回失败。

pause（暂停）是一种dble server级别的全局状态，同一时间段只能执行一个 pause 操作，暂停节点不可追加或者逐步恢复，只能一起暂停指定的一个或几个shardingNode并一起恢复流量。暂停期间执行 reload命令、dble重启、新增dble节点 均可以维持暂停状态。这里推荐谨慎的使用暂停并选择影响的范围，推荐逐个变动暂停，reload，恢复，再进行下一个变动

2.1.14.1 暂停流量：

```
pause @@shardingNode = 'dn1,dn2' and timeout = 10 ,queue = 10,wait_limit = 10;
```

参数描述:

timeout:这个参数是等待涉及到的事务完成的时间，如果在到达timeout之后，仍然有事务未完成，本次pause失败，单位秒。

queue:这个参数表示暂停期间的阻塞前端连接的数量，超过此数量时，前端连接建立将会发生错误。

wait_limit:是针对被阻塞的每个单个的前端的时间限制，如果被阻塞了超过wait_limit的时间，将会返回错误，单位秒。

2.1.14.2 恢复流量：

```
RESUME; 返回正常的OK 或者错误“No shardingNode paused”
```

2.1.14.3 查看当前暂停状态：

```
show @@pause;
```

```
mysql> show @@pause;
+-----+
| PAUSE_SHARDING_NODE |
+-----+
| dn1                  |
| dn2                  |
+-----+
2 rows in set (0.15 sec)
```

2.1.15 慢日志相关命令

慢日志相关的命令:

2.1.15.1 查询慢查询日志的开启状态

```
mysql> show @@slow_query_log;
+-----+
| @@slow_query_log |
+-----+
| 0                |
+-----+
1 row in set (0.00 sec)
```

2.1.15.2 开启慢查询日志

```
mysql> enable @@slow_query_log;
Query OK, 1 row affected (0.09 sec)
enable slow_query_log success
```

2.1.15.3 关闭慢查询日志

```
mysql> disable @@slow_query_log;
Query OK, 1 row affected (0.03 sec)
disable slow_query_log success
```

2.1.15.4 查看慢查询日志统计阈值

```
mysql> show @@slow_query.time;
+-----+
| @@slow_query.time |
+-----+
| 100                |
+-----+
1 row in set (0.00 sec)
```

2.1.15.5 修改慢查询日志统计阈值

```
mysql> reload @@slow_query.time=200;
Query OK, 1 row affected (0.10 sec)
reload @@slow_query.time success

mysql> show @@slow_query.time;
+-----+
| @@slow_query.time |
+-----+
| 200                |
+-----+
1 row in set (0.00 sec)
```

2.1.15.6 查看慢查询日志刷盘周期

```
mysql> show @@slow_query.flushperiod;
+-----+
| @@slow_query.flushperiod |
+-----+
| 1                          |
+-----+
1 row in set (0.00 sec)
```

2.1.15.7 修改慢查询日志刷盘周期

```
mysql> reload @@slow_query.flushperiod=2;
Query OK, 1 row affected (0.05 sec)
reload @@slow_query.flushPeriod success

mysql> show @@slow_query.flushperiod;
+-----+
| @@slow_query.flushperiod |
+-----+
| 2 |
+-----+
1 row in set (0.00 sec)
```

2.1.15.8 查看慢查询日志刷盘条数阈值

```
mysql> show @@slow_query.flushsize;
+-----+
| @@slow_query.flushsize |
+-----+
| 1000 |
+-----+
1 row in set (0.01 sec)
```

2.1.15.9 修改慢查询日志刷盘条数阈值

```
mysql> reload @@slow_query.flushsize=1100;
Query OK, 1 row affected (0.03 sec)
reload @@slow_query.flushSize success

mysql> show @@slow_query.flushsize;
+-----+
| @@slow_query.flushsize |
+-----+
| 1100 |
+-----+
1 row in set (0.00 sec)
```

2.1.15.10 修改并查看慢日志队列无空间时后续日志的处理策略

slowQueueOverflowPolicy配置体现在[bootstrap.cnf](#)文件中

```
mysql> reload @@slow_query.queue_policy=1;
Query OK, 1 row affected (0.02 sec)
reload @@slow_query.queue_policy success

mysql> select * from db_variables where variable_name = 'slowQueueOverflowPolicy';
+-----+-----+-----+-----+
| variable_name | variable_value | comment | read_only |
+-----+-----+-----+-----+
| slowQueueOverflowPolicy | 1 | Slow log queue overflow policy, the default is 2 | false |
+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

2.1.15.11 查看某个连接的当前执行状态

show @@connection.sql.status where FRONT_ID= ?; 此功能需要开启慢日志才有效，当对应的连接当前query已经执行完毕时，执行此命令的结果与 `trace` 功能相同。如果query正在执行，本结果将试图展示query执行到哪一个步骤了。例如，广播查询


```
mysql> show @@connection.sql.status where FRONT_ID= 1;
```

OPERATION	START(ms)	END(ms)	DURATION(ms)	SHARDING_NODE	SQL/REF
Read_SQL	0.0	0.082598	0.082598	-	-
Parse_SQL	0.082598	0.676424	0.593826	-	-
Route_Calculation	0.676424	0.895382	0.218958	-	-
Prepare_to_Push/Optimize	0.895382	6743.838628	6742.943246	-	-
Execute_SQL	6743.838628	6753.488422	9.649794	dn1	select * from sharding_4_t1
Execute_SQL	6743.838628	6751.472835	7.634207	dn3	select * from sharding_4_t1
Execute_SQL	6743.838628	6750.981646	7.143018	dn4	select * from sharding_4_t1
Execute_SQL	6743.838628	6753.31394	9.475312	dn2	select * from sharding_4_t1
Fetch_result	6753.488422	6754.383316	0.894894	dn1	select * from sharding_4_t1
Fetch_result	6751.472835	6751.656604	0.183769	dn3	select * from sharding_4_t1
Fetch_result	6750.981646	6751.188385	0.206739	dn4	select * from sharding_4_t1
Fetch_result	6753.31394	6754.286055	0.972115	dn2	select * from sharding_4_t1
Write_to_Client	6750.981646	unfinished	unknown	-	-

13 rows in set (0.04 sec)

再比如join

```
mysql> show @@connection.sql.status where FRONT_ID= 1;
```

OPERATION	START(ms)	END(ms)	DURATION(ms)	SHARDING_NODE	SQL/REF
Read_SQL	0.0	0.039588	0.039588	-	-
Parse_SQL	0.039588	0.756578	0.71699	-	-
Route_Calculation	0.756578	1.5547	0.798122	-	-
Prepare_to_Push/Optimize	1.5547	3.428551	1.873851	-	-
Execute_SQL	3.428551	2362.10579	2358.677239	dn1_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Fetch_result	2362.10579	unfinished	unknown	dn1_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Execute_SQL	3.428551	2362.122407	2358.693856	dn2_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Fetch_result	2362.122407	unfinished	unknown	dn2_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Execute_SQL	3.428551	2362.307153	2358.878602	dn3_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Fetch_result	2362.307153	unfinished	unknown	dn3_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Execute_SQL	3.428551	2364.523615	2361.095064	dn4_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
Fetch_result	2364.523615	unfinished	unknown	dn4_0	select `a`.`age`,`a`.`id`,`a`.`name` from `sharding_4_t1` `a` ORDER BY `a`.`id` ASC
MERGE_AND_ORDER	2362.639012	unfinished	unknown	merge_and_order_1	dn1_0; dn2_0; dn3_0; dn4_0
SHUFFLE_FIELD	4178.383366	unfinished	unknown	shuffle_field_1	merge_and_order_1
Execute_SQL	3.428551	2365.71371	2362.285159	dn1_1	select `b`.`id` from `sharding_2_t1` `b` ORDER BY `b`.`id` ASC
Fetch_result	2365.71371	unfinished	unknown	dn1_1	select `b`.`id` from `sharding_2_t1` `b` ORDER BY `b`.`id` ASC
Execute_SQL	3.428551	2365.952707	2362.524156	dn2_1	select `b`.`id` from `sharding_2_t1` `b` ORDER BY `b`.`id` ASC
Fetch_result	2365.952707	unfinished	unknown	dn2_1	select `b`.`id` from `sharding_2_t1` `b` ORDER BY `b`.`id` ASC
MERGE_AND_ORDER	2366.164823	unfinished	unknown	merge_and_order_2	dn1_1; dn2_1
SHUFFLE_FIELD	not started	unfinished	unknown	-	-
JOIN	not started	unfinished	unknown	-	-
SHUFFLE_FIELD	not started	unfinished	unknown	-	-
Write_to_Client	not started	unfinished	unknown	-	-

```
23 rows in set (0.04 sec)
```

2.1.16 创建/删除物理数据库命令

2.1.16.1 创建物理数据库

用于dble启动后发现有些shardingNode对应的物理库还未建立，可以使用后端命令一次性建立。

命令格式：

```
create database @@shardingNode ='dn.....'
```

shardingNode值支持 dn\$1-4 这种形式。

当所包含的shardingNode至少有一个不在配置文件当中时，或者配置的shardingNode未被引用时，均会返回错误：shardingNode \$Name does not exists.

否则会对所有 shardingNode 执行 create database if not exists \$databaseName，执行完成之后返回OK。

2.1.16.2 删除物理数据库

用于删除某些shardingNode对应的物理库，可以使用后端命令一次性删除。

命令格式：

```
drop database @@shardingNode ='dn.....'
```

shardingNode值支持 dn\$1-4 这种形式。

当所包含的shardingNode至少有一个不在配置文件当中时，返回错误：shardingNode \$Name does not exists.

否则会对所有 shardingNode 执行 drop database if exists \$databaseName，执行完成之后返回OK。若在执行过程中发生错误，会将show @@shardingNode里面的SCHEMA_EXISTS置为false，需要用户人工确认下是否物理库已删除成功。

check系列命令

2.1.17.0 check @@metadata 命令

用于检查meta信息是否存在以及加载的时间。

命令格式：

- 第一种形式 `check @@metadata`，返回结果可能是：
 1. 上一次 reload @@metadata 的datetime；
 2. 上一次 reload @@config_all的datetime（注意：在配置没有变更的情况下，不会同步元数据，因此datetime是不会变化的）；
 3. 启动时加载meta的datetime
- 第二种形式 `check full @@metadata`，并且支持以下过滤条件：
 - `where schema=? and table=?`
 - `where schema=?`
 - `where reload_time='yyyy-MM-dd HH:mm:ss'`，`where reload_time>='yyyy-MM-dd HH:mm:ss'`，`where reload_time<='yyyy-MM-dd HH:mm:ss'`
 - `where reload_time is null`
 - `where consistent_in_sharding_nodes=0`
 - `where consistent_in_sharding_nodes = 1`
 - `where consistent_in_memory=0`
 - `where consistent_in_memory = 1`
 - If no where, retrun all results.
- `check full @@metadata` 结果集如下：

schema	table	reload_time	table_structure	consistent_in_sharding_nodes	consistent_in_memory
schema	table	2018-09-18 11:01:04	CREATE TABLE table`(...	1	1

column `table_structure` 和 `show create table` 命令结果的形式一样

column `consistent_in_sharding_nodes` 表示不同分片之间的一致性，0为不一致，1为一致

column `consistent_in_memory` 表示内存中meta与实际后端结点的一致性，0为不一致，1为一致

当table_structure列为null时，consistent_in_sharding_nodes列和consistent_in_memory列没有意义。

当consistent_in_sharding_nodes为0时，consistent_in_memory没有意义。

2.1.17.1 check @@global schema = " [and table = "]"

用于进行手动全局表检查的命令，当即触发一次特定范围的全局表检查，并且将检查结果作为返回值进行展示。

结果如下所示：

```
mysql> check @@global schema = 'testdb';
+-----+-----+-----+-----+
| SCHEMA | TABLE      | DISTINCT_CONSISTENCY_NUMBER | ERROR_NODE_NUMBER |
+-----+-----+-----+-----+
| testdb | tb_global1  | 0                             | 0                 |
+-----+-----+-----+-----+
```

SCHEMA: 所检查的SCHEMA名字

TABLE: 所检查的TABLE名字

DISTINCT_CONSISTENCY_NUMBER: 返回有几个不同的检查结果的版本

ERROR_NODE_NUMBER: 在检查过程中有几个节点执行SQL报错

具体最终结果是否符合用户的预期，还是要根据表格的检查SQL和返回信息来进行判断，但一般情况下认为当DISTINCT_CONSISTENCY_NUMBER的值大于1的情况下，表格中的内容一定不一致

2.1.18 release 命令

2.1.18.1 release @@reload_metadata

描述：打断正在进行的reload_metadata步骤。当reload/reload config_all/reload metadata进行到最后一个步骤时，都会进行reload metadata。就经验而言这个步骤存在hang死的风险，所以当出现reload metadata步骤迟迟未返回的情况，可以使用这个命令进行终止，使得dble能够继续提供服务。结果：OK 或者ERROR

注意：

- 此命令会导致config可能和table meta不匹配的情况，若执行此命令，请务必在后面追加命令reload @@metadata更新最新的meta信息
- 当一个reload过程被打断时，执行reload的进程会返回编号为5999的SQL错误，请慎重对待此错误CODE，这个错误代表着“新的配置被应用，meta信息在更新的过程中终止”
- 在执行运维操作的过程中可能会需要查看dble当前的reload状态，可通过具体命令show @@reload_status进行查看

2.1.19 split 命令

背景

在进行POC时，现场人员进行数据导入时经常遇到各种问题，比较典型的是dble在导入文件时，对部分sql语句的不支持。另外，未分片的历史数据通过dble导入，旧数据会路由到分片，在数据量较大时，耗时会比较长，在此过程中出现错误的话也很难排查。基于以上原因，2.19.09.0版本提供工具对mysqldump导出的源数据文件按照后端分片节点进行分割。分割后的数据文件可以在每个后端分片执行导入，适配数据按分片导入的需求。

dump文件语句处理

1. create database：会将逻辑数据库转换为物理库。
2. ddl语句：根据表的分片节点写入到对应后端节点的dump文件中，对于自增列，会将自增列的数据类型修改为bigint。
3. insert：全局序列值会被dble替换为全局序列，再按照拆分列根据拆分算法路由到对应后端节点的dump文件中。
4. 一些属性设置的语句会根据最近一次解析的ddl来决定下发到具体的后端节点的dump文件中。
5. 会跳过对视图的处理
6. 会跳过对子表的处理

使用方法

命令

在管理端口执行以下命令：

```
mysql > split src dest [-sschema] [-r500] [-w512] [-l10000] [--ignore] [-t2]
```

- src：表示原始dump文件名
- dest：表示生成的dump文件存放的目录
- -s：表示默认逻辑数据库名，当dump文件中不包含schema的相关语句时，会默认导出到该schema。如：当dump文件中包含schema时，dump文件中的优先级高于-s指定的；若文件中的schema不在配置中，则使用-s指定的schema，若-s指定的schema也不在配置中，则返回报错
- -r：表示设置读文件队列大小，默认500
- -w：表示设置写文件队列大小，默认512，且必须为2的次幂
- -l：表示split后一条insert中最多包含的values，只针对分片表，默认4000
- --ignore：insert时，忽略已存在的数据
- -t：表示多线程处理文件中insert语句时线程池的大小

生成的分片文件以格式：源文件名-shardingNode名-时间戳.dump，最新的文件时间戳最大。

例如：我的原始dump文件是 /tmp/mysql_dump.sql，我想切分以后输出到/tmp/dump/目录下：命令就是：

```
split /tmp/mysql_dump.sql /tmp/dump/
```

日志

默认情况下，split过程中生成的日志打印到在dble.log中，提供配置让split命令产生的日志单独存放，若需要开启，则需修改log4j.xml文件。

```
<Configuration status="WARN">
  <Appenders>
    <!-- 将下面的此段配置追加至已安装dble的log4j.xml中的Appenders下 -->
    <RollingFile name="DumpFileLog" fileName="logs/dump.log"
      filePattern="logs/${date:yyyy-MM}/dump-%d{MM-dd}-%i.log.gz">
      <PatternLayout>
        <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %5p [%t] (%l) - %m%n</Pattern>
      </PatternLayout>
      <Policies>
        <OnStartupTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="250 MB"/>
        <TimeBasedTriggeringPolicy/>
      </Policies>
      <DefaultRolloverStrategy max="10"/>
    </RollingFile>
  </Appenders>
  <Loggers>
    <!-- 将下面的此段配置追加至已安装dble的log4j.xml中的Loggers下，可通过调整level为debug来调整性能 -->
    <Logger name="dumpFileLog" level="info" additivity="false" includeLocation="false" >
      <AppenderRef ref="DumpFileLog" />
      <AppenderRef ref="RollingFile"/>
    </Logger>
  </Loggers>
</Configuration>
```

可通过日志中的“dump file has been read d%”关键字来查看解析进度。

可开启日志的debug级别来调整性能

任务停止

执行dump file任务的管理连接不受 idletimeout 参数的限制。用户可以通过kill @@connection id 方式杀掉管理连接从而停止dump file的任务的执行。

使用限制

1. 数据导入之后需要运维检查下数据完整性。
2. 对于使用全局序列的表，表原先全局序列中的值会被擦除，替换成全局序列，需要注意。
3. 暂时不支持子表的dump操作。

2.1.20 flow_control 命令

2.1.20.1 查询流量控制当前配置状态

```
mysql> flow_control @@show;
+-----+-----+-----+
| FLOW_CONTROL_TYPE | FLOW_CONTROL_HIGH_LEVEL | FLOW_CONTROL_LOW_LEVEL |
+-----+-----+-----+
| FRONT_END        | 4194304 | 262144 |
| dbGroup1-hostM1  | 4194304 | 262144 |
| dbGroup2-hostM2  | 4194304 | 262144 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

列描述:

- FLOW_CONTROL_TYPE 流量控制的类型，前端连接为固定值“FRONT_END”;后段连接为所在的实例名，格式为：组名-实例名
- FLOW_CONTROL_HIGH_LEVEL 流量控制的高水位线，积压队列的字节数到达此位置开始控制流量
- FLOW_CONTROL_LOW_LEVEL 流量控制低水位线，积压队列的字节数低于此位置不再控制流量

2.1.20.2 修改流量控制当前配置状态

```
flow_control @@set [enableFlowControl = true/false] [flowControlHighLevel = ?] [flowControlLowLevel = ?]
```

```
MySQL [(none)]> flow_control @@set enableFlowControl = true flowControlHighLevel= 100000 flowControlLowLevel = 30000;
Query OK, 0 rows affected (0.02 sec)
```

通过此命令可以修改对应内存中生效的值，同时会变更文件 bootstrap.dynamic.cnf 将其固化，对应关系如下:

- enableFlowControl: bootstrap.cnf中enableFlowControl参数
- flowControlHighLevel: bootstrap.cnf中flowControlHighLevel参数
- flowControlLowLevel: bootstrap.cnf中flowControlLowLevel参数

注: 以上命令三个参数顺序不可变更

2.1.20.3 查看当前连接的流量控制状态

```
MySQL [(none)]> flow_control @@list;
+-----+-----+-----+-----+-----+-----+
| CONNECTION_TYPE | CONNECTION_ID | CONNECTION_INFO | WRITING_QUEUE_BYTES | READING_QUEUE_BYTES | FLOW_CONTROLLED |
+-----+-----+-----+-----+-----+-----+
| ServerConnection | 1 | 127.0.0.1:50817/schema1 user = root | 464594 | NULL | false |
| MySQLConnection | 8 | 10.186.65.86:3307/db2 mysqlId = 1287 | 0 | 0 | false |
| MySQLConnection | 12 | 10.186.65.86:3308/db1 mysqlId = 1557 | 0 | 0 | false |
| MySQLConnection | 6 | 10.186.65.86:3307/db1 mysqlId = 1285 | 0 | 86172 | false |
| MySQLConnection | 15 | 10.186.65.86:3308/db2 mysqlId = 1559 | 0 | 0 | false |
+-----+-----+-----+-----+-----+-----+

```

列描述:

- CONNECTION_TYPE 连接的类型，固定为MySQLConnection/ServerConnection其中之一
- CONNECTION_ID 连接在dble中的ID信息，可以通过ID查找日志
- CONNECTION_INFO 连接详细信息，使用端口，IP地址，用户，MySQL中的连接ID等
- WRITING_QUEUE_BYTES 当前连接的准备写出的队列里积压的字节数
- READING_QUEUE_BYTES 当前连接的已经读取的队列里积压的字节数，前端连接不支持此功能，恒为null
- FLOW_CONTROLLED 当前连接是否处于被流控的状态

如果需要过滤结果集，可以使用dble_information.dble_flow_control 来过滤

2.1.21 刷新后端连接池

2.1.21.1 介绍&背景

当在后端mysql做全局变量/权限等的变更时，需要dbInstance的连接池进行一次重新建立，新建的连接才会有新的属性。

2.1.21.2 命令

fresh conn [forced] where dbGroup ='groupName' [and dbInstance='instanceName'];

- 不包含forced时，则空闲的后端连接直接丢弃，而正在使用的后端连接等归还后丢弃
- 包含forced时，则无论空闲还是正在使用的后端连接都直接丢弃；另外正在使用的后端连接对应的前端连接也会全部断开
- 不指定dbInstance时，则dbGroup下所有的dbInstance都将刷新
- 指定dbInstance时，多个dbInstance以逗号隔开，则刷新dbGroup下指定的dbInstance

2.1.22 脱离集群 命令

- 背景
- 介绍
- 命令
 - `cluster @@detach`
 - `cluster @@attach`
- 伪集群模式下的限制
 - 阻塞等待
 - 无法执行
 - 不一致问题
- 术语解释

以下内容只针对集群状态下的 `dbie` 有效。

2.1.22.1 背景

在部分场合下，集群状态下的 `dbie` 可能需要临时断开和集群中心（指`zookeeper/ucore`）的连接。

典型的应用场景是：集群中心需要进行升级/维护时。

在版本<3.21.10，唯一的方式是配置修改为单机版并重新启动 `dbie`；需要加回来时，改会原来的配置并重启 `dbie`。但这对业务影响很大，通常在生产环境下难以接受。

2.1.22.2 介绍

在版本≥3.21.10，`dbie`提供了一个脱离集群的指令，可以在不重启的情况下，临时断开和集群中心的连接，让 `dbie` 以一个 伪集群模式 的单实例方式运行。

之所以叫做 伪集群，因为他的状态和 普通的单机版`dbie` 不同，`dbie` 会认为自己还是处于集群状态，操作方式还是和集群方式相同，只是没法和集群通讯。他执行一些集群相关指令(含义见2.1.22.4)会报错，比如 `reload @@config`。

2.1.22.3 命令

`cluster @@detach [timeout=10]`

- 脱离集群，断开和集群中心的连接。
- 在执行完 脱离集群指令后，在加入集群之前，集群相关指令(含义见2.1.22.4)将不可用且报错：“cluster is detached”
- 可以指定 `timeout` 单位是秒，默认是 10s，不建议设置过大，因为会阻塞其他指令。

`cluster @@attach [timeout=10]`

- 唯一用途：脱离集群的逆操作。用于重新加入集群。恢复和集群中心的连接
- 可以指定 `timeout` 单位是秒，默认是 10s，不建议设置过大，因为会阻塞其他指令。

2.1.22.4 伪集群模式下的限制

2.1.22.4.1 阻塞等待

脱离集群 指令分为三个阶段。

1. 预备阶段
2. 执行阶段
3. 执行完成

在第一步预备阶段，如果有先来的 集群相关指令或者管理端(9066执行的)指令 正在执行，则 脱离集群指令 会阻塞等待这些指令完成，或者阻塞超时了直接返回报错。因为脱离集群和这些指令并发可能有风险。

在第一步预备阶段和第二步执行阶段，如果有后来的 集群相关指令或者管理端(9066执行的)指令 即将执行，则这些指令会阻塞等待，直到 脱离集群 指令完成。因为脱离集群和这些指令并发可能有风险。

简单来说，脱离集群指令会被 集群相关指令或者管理端(9066执行的)指令 阻塞，集群相关指令或者管理端(9066执行的)指令 会被 脱离集群 阻塞。为了防止 集群相关指令或者管理端(9066执行的)指令被阻塞太久影响实际使用，建议 `timeout`超时时间不宜配置太长。

加入集群 指令 同上。

2.1.22.4.2 无法执行

在执行完 脱离集群指令后，在加入集群之前，集群相关指令将不可用且报错：“cluster is detached”。这是合理的，因为此时和集群不通讯，所以无法完成进一步操作。

2.1.22.4.3 不一致问题

在执行完 脱离集群指令后，如果集群内的其他节点执行 集群相关指令，将正常执行，但是，将不会对该节点产生任何影响，因为收不到消息。此时，集群外和集群内的 `dbie` 很有可能状态不一致，即使后续重新加入了集群，但是`dbie` 仍不能保证一致性。

比如，集群中有 A,B,C 三个节点。节点 A 脱离了集群，节点 B 加入了一张表t1并进行reload，此时节点C和节点B的信息一致，均包含 表t1，但是节点A收不到消息，不会 reload 也不知道 表t1 的存在。在这个时间点，A 和 B 不一致。在这之后，节点 A 重新加入了集群，但是A 已经错过了 reload 事件，不会进行 reload ，依旧是和 B 不一致。

因此不建议在 脱离集群后加入集群前 这个时间点执行集群相关指令。 如果已经做了，请手动确认一致性。

2.1.22.5 术语解释

集群相关指令 指的是 使用了集群元数据的指令，主要是 https://actiontech.github.io/dble-docs-cn/2.Function/2.08_cluster.html 提到的指令。包括以下这些（可能信息更新不及时，具体以链接里提到的为主）

- DDL
- reload @@config
- show @@binlog.status
- 创建view
- 高可用命令同步
- 暂停流量
- xa 日志的commit/rollback
- 全局序列中采用的”分布式offset-step方式“实现的 table 的插入语句

2.2 全局序列

在分库分表的情况下，数据库自增主键无法保证自增主键的全局唯一。

为此，dble提供了全局序列，并且针对不同应用场景提供了多种实现方式。要应用全局序列，需要在cluster.cnf中配置：

```
sequenceHandlerType=n
```

其中sequenceHandlerType包括如下种类：

- 1: [MySQL offset-step方式](#)
- 2: [时间戳方式](#)
- 3: [分布式时间戳方式](#)
- 4: [分布式offset-step方式](#)

全局序列用于给自增列赋值，写入数据时字段值由系统自动生成，不可以再指定值，使用示例：

```
/*id为主键，配置为自增长*/  
insert into table1(name) values('test');
```

2.2.1 MySQL offset-step方式

MySQL offset-step方式利用了数据库的并发和并发更新互斥的特性。

在数据库中建立一张表，保存序列的当前值和步长，使用该序列时，系统从表中读取序列的当前值，加上步长，返回给客户端的同时更新表中的当前值。线程获取序列的执行步骤如下：

1. 检查是否有请求序列未用的缓存的序列值，如果没有，执行步骤2；否则，执行步骤4。
2. 到相应的数据节点上执行 `SELECT dble_seq_nextval('\seqName\')`；其中，**seqName**为请求序列的序列名。此步骤获取的序列值数依赖于该序列的步长**increment**，为区间`[current_value, current_value+increment)`中的值。
3. 缓存步骤2得到的序列值。
4. 返回缓存的序列值中第一个未用的序列值。

以上提到的部分配置内容参见[1.7.1 offset-step方式](#)

2.2.2 时间戳方式

这种方式下，全局序列在dble服务实例本地产生，只能生成全局唯一的ID，不能实现连续自增。

使用这种方式需要对应字段为bigint来保证63位(63位的原因是Java没有无符号整数类型，所以最高位恒为0，保证全局序列是个正数)序列值是63bits的整数。整数的位模式如下：

a.29bits	b.10bits	c.12bits	d.12bits
----------	----------	----------	----------

其中，

- a - e为从高位到低位。
- a为系统当前时间戳的低41位中的高29位。
- b为10位instance id，使用bootstrap.cnf的instanceId
- c为12位自增长值
- d为系统当前时间戳的低41位中的低12位。

注意事项：

1. bootstrap.cnf的instanceId的最大值均为1023。
2. 每毫秒时间内允许的最多序列值为4095。为了保证序列值的唯一性，在毫秒时间内请求超过4095个序列值时系统会进行等待到下一毫秒开始。
3. 因为java没有无符号整数,实际使用41位时间戳相对于1288834974657L(2010年左右)的偏移量。
4. 相对于偏移量的处理过够后的41位时间戳可供使用69年。

2.2.3 分布式时间戳方式

本方式提供一个基于Zookeeper(也可以本地配置)的分布式ID生成器,可以生成全局唯一的63位二进制ID。
PS:63位的原因是Java没有无符号整数类型,所以最高位恒为0,保证全局序列是个正数

2.2.3.1 位模式

序列值是63bits的整数。整数的位模式如下:

a.9bits	b.9bits	c.6bits	d.39bits
---------	---------	---------	----------

其中:

- a - e为从高位到低位。
- a为线程id的低9位值。
- b为9位实例 id值(根据配置, 此值为bootstrap.cnf的instanceId值或者从zookeeper服务器获取的值, 参见[1.7.3 分布式time序列](#))。
- c为6位自增长值
- d为系统当前时间戳的低39位值(可以使用17年)。

2.2.3.2 退化的分布式时间序列

如果cluster.cnf中的sequenceInstanceByZk值不为true, 序列的维护仅依赖于单实例 (bootstrap.cnf的instanceId值的维护), 此时序列类似于时间戳方式(参见[2.2.2 时间戳方式](#))。

2.2.3.3 分布式时间序列

如果cluster.cnf中的sequenceInstanceByZk值为true, 序列的维护用zookeeper的临时自增节点来维持。每次生成全局序列时,向zk申请一个临时自增节点,通过计算自增节点数 % 32 获取INSTANCEID.

2.2.4 分布式offset-step方式

分布式offset-step方式利用zookeeper的分布式锁功能进行实现。

线程在获取序列值时执行如下步骤：

1. 根据序列名(`**schemaX**`.`**tableX**`)获取相应的分布式锁。
2. 从zookeeper服务器上获取未用序列值的最小值min。
3. 根据配置(参见[1.7.4 分布式offset-step序列](#)),计算能获取的最大未用序列值max。
4. 用max+1更新zookeeper服务器上的未用序列值的最小值min。
5. 释放相应的分布式锁。

2.3 读写分离

3.20.10.0版本dble支持单纯的读写分离，可以和分库分表功能分开单独使用。3.20.10.0之前的版本，分库分表也支持读写分离，兼容该功能。

2.3.1 读写分离配置

2.3.1.1 单纯使用读写分离功能的配置

若想启用dble的读写分离，仅需在 user.xml 文件中配置 rwSplitUser并指定对应的dbGroup即可。dbGroup的配置参考db.xml的章节。这里需要注意的是三种用户配置的顺序是固定的。user.xml的配置请参考user.xml章节。

```
<dble:user xmlns:dble="http://dble.cloud/" version="4.0">
  <managerUser name="man1" password="654321" maxCon="100"/>
  <shardingUser name="root" password="123456" schemas="testdb" readOnly="false" maxCon="20"/>
  <rwSplitUser name="rwsu1" password="123456" dbGroup="rwGroup" maxCon="20"/>
</dble:user>
```

配置注意事项：

1. 当user.xml文件中不配置shardingUser，dble不再加载sharding.xml配置文件(即dble不具备分表分库)，包括集群情况下出现sharding.xml不一致，均属于已知现象。
2. 当同时开启dble读写分离和分库分表的功能，分库分表引用的dbGroup和读写分离引用的dbGroup必须相互独立。rwSplitUser引用的dbGroup，仅需在db.xml中定义即可。shardingUser引用的dbGroup，需要被配置的schemas对应的sharding.xml中的shardingNode所引用。
3. 多个rwSplitUser可以引用同一个dbGroup。
4. 被读写分离或者分库分表使用的dbGroup内的instance才会有心跳和连接池；未被有效使用的dbGroup内的instance只有心跳，不会初始化连接池。

2.3.1.2 分库分表中读写分离的配置

分库分表中的读写分离，配置好db.xml和sharding.xml即可，具体参考db.xml和sharding.xml的章节

2.3.2 负载均衡

dble通过配置多个dbInstance为读操作提供负载均衡，注意的是rwSplitMode配置不为0，详细请参见db.xml章节。负载均衡规则如下：

1. 确定参与读写分离的dbInstance集合
2. 负载均衡算法

2.3.2.1 确定参与读写分离的dbInstance集合

该算法在每次连接获取时提供可用的dbInstances实例集。可用的dbInstance是指心跳正常的dbInstance，这里需要注意的是show slave status和其他心跳语句是有区别的，以该语句作为心跳语句，心跳正常只是基本前提。dble会根据最近一次的心跳返回结果判断读库和主库的延迟，如果延迟超过delayThreshold配置，则不会将此节点加入到dbInstances实例集中，如果delayThreshold=-1那么不会进行延迟检测。

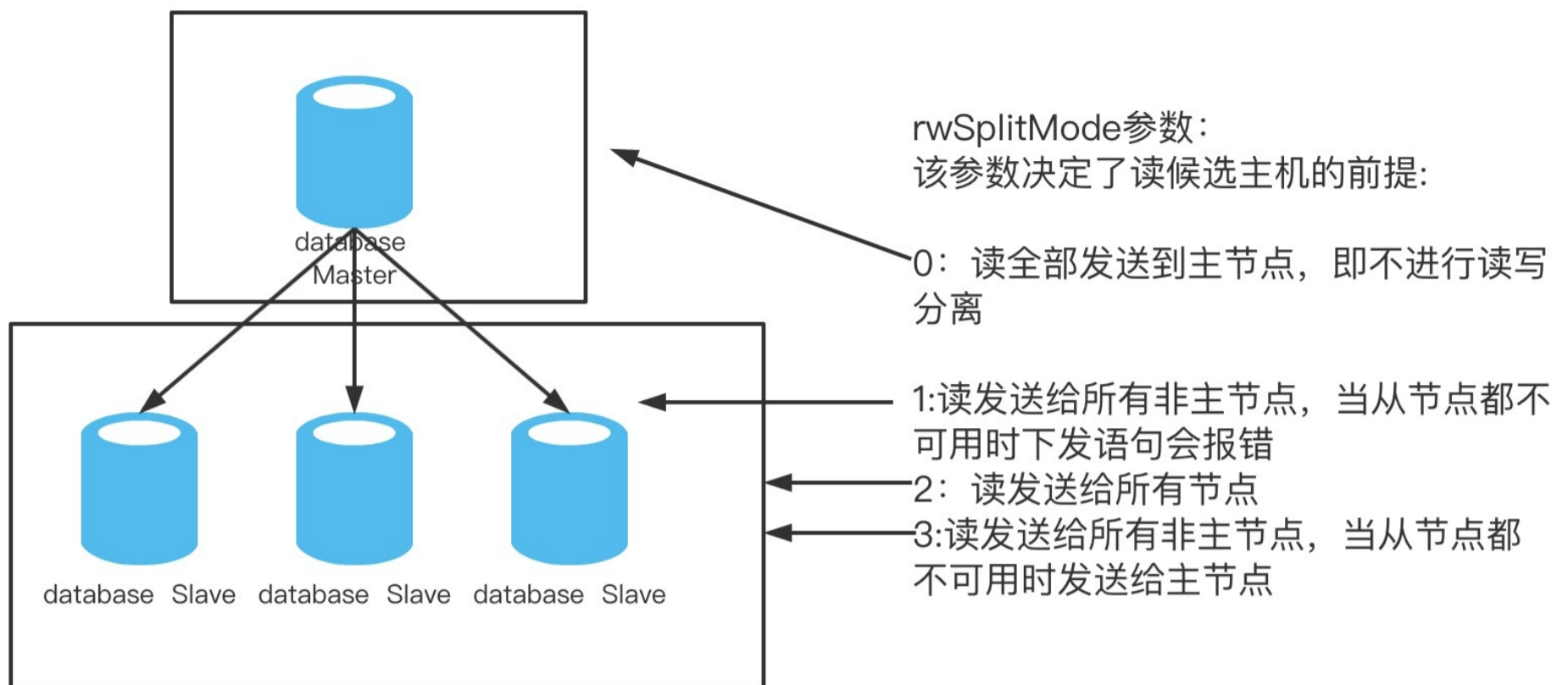
- 写节点(primary="true")可用
 - rwSplitMod配置为2，则写节点有资格参与到读写分离，将写节点加入到dbInstances实例集
 - 读节点(primary没配置或者 primary="false")
 - 节点可用且需要进行延迟检测，检查延迟是否在阈值内再决定是否加入到dbInstances实例集
 - 节点可用且不需要进行延迟检测，直接加入到dbInstances实例集
- 写节点异常
 - 检查读节点是否可用，与上面读节点的检测机制一致

2.3.2.2 负载均衡算法

该算法在dbInstance集合中选择一个dbInstance实例来获取连接。

- dbInstance集合为空，前端报错。
- dbInstance集合非空
 - 每个dbInstance有权重设置(readWeight参数)，但不都是等值权重，依权重随机选择。
 - 每个dbInstance无权重设置或所有权重等值，则等权随机选择。此种情况只是上面情况的特例。

2.3.2.3 写节点是否参与均衡与dbGroup的rwSplitMode属性有关，具体见下图



2.3.3 读写分离支持语句类型

在事务中所有语句都会发主，在非事务中则根据语句类型进行负载均衡。

2.3.3.1 纯读写分离支持的语句类型

1. ddl
2. dml
3. prepared statement协议
4. 函数，存储过程

2.3.3.2 分库分表支持的语句类型

1. 可进行负载均衡的SQL语句为 select 或者 show。

2.3.4 读写分离功能限制

1. druid 解析器限制 - 不支持set语句中存在特殊字符；
2. druid 解析器限制 - set session transaction read write, isolation level repeatable read中，逗号后的语句不生效；
3. 只读事务（在 >= dble 3.21.06.x版本中支持，之前的版本不支持此功能）
4. 不支持set transaction read write；
5. select 语句现在的逻辑是都进行负载，还没有进行细节的区分，比如有些语句需要强制发主，如系统函数，系统表，系统变量；
6. select ... into 或者 load data中存在用户变量，通过dble再次查询该变量，变量值不对；
7. 在会话中，删除正在使用的库，mysql会将当前库置为null，dble会依然保留正在使用的库；
8. set 语句目前只支持会话级别系统变量和用户变量的设置，若需要设置密码等可以使用hint的方式设置或去后端节点去设置；
9. 读写分离会打破原先的隔离级别的原有语义，对此有严格要求的需要酌情考虑；
10. 创建临时表后，之后所有的语句都发往主，因为临时表不支持主从复制。直到你删除了所有临时表后，原先的负载均衡策略恢复；
11. 部分客户端，比如在设置了 allowMultiQueries=true (默认为 false) 的 jdbc，此时客户端可以一次性发送 multi-queries，dble 对此情况不做拆分，全部发往主。MySQL Command-Line client 会在客户端拆分语句，一次只发送一条语句，故不会有该条限制；

2.3.5 读写分离后端实例的粘滞性

见rwStickyTime参数

功能背景

读写分离非事务场景下，写完立刻读，读会发到从机上可能存在主从延迟从而导致读不到数据。

粘滞性

执行当前读SQL的时间，距离上一次写SQL执行的时间段，没有超过rwStickyTime时间段时，则当前读SQL将会下发至后端主(写)实例。

特例：Hint SQL不参与实例的粘滞性

特性

此粘滞性功能，优先于db.xml中的rwSplitMode配置

举例说明

假设，rwStickyTime=1000，表示粘滞时间段为1000ms；操作如下：

Step	Time Line	SQL	InstanceDB of backend	说明
0	50ms	Hint_SQL_1(如: /*master*/sql)	master	下发到主实例；不参与粘滞，不更新时间点timeA
1	100ms	写SQL_1	master	下发到主实例；更新时间点timeA=100ms
2	500ms	读SQL_2	master	原本应该下发到从实例，但满足rwStickyTime>0且(500ms-timeA) <=rwStickyTime，因此SQL最终下发到主实例
3	600ms	Hint_SQL_2(如: /*slave*/sql)	slave	依旧下发到从实例；不参与粘滞
4	900ms	读SQL_3	master	原本应该下发到从实例，但满足rwStickyTime>0且(900ms-timeA) <=rwStickyTime，因此SQL最终下发到主实例
5	2000ms	读SQL_4	slave	下发到从实例；(并不满足rwStickyTime>0&&(2000ms-timeA) <=rwStickyTime)

补充：在读写分离中，读SQL和写SQL的定义较为简明：`select...`、`show ...` 语句为读SQL，其他语句即为写SQL。

2.3.6 读写分离本地读

功能背景

读写分离本地读场景下，dble实例的配置与instance实例的配置一致，就可以认定该instance为本地实例，读流量会优先下发到本地实例，如果本地实例存在异常或者不可用，按照策略下发到其他实例

2.3.6.1 使用读写分离本地读的的配置

若想启用dble的读写分离本地读功能，需要在bootstrap.cnf和db.xml文件中配置

bootstrap.cnf

```
-Ddistrict=district1
-DdataCenter=dataCenterA
```

db.xml

```
<?xml version="1.0"?>
<dble:db xmlns:dble="http://dble.cloud/">

  <dbGroup name="dbGroup1" rwSplitMode="1" delayThreshold="10000">
    <heartbeat errorRetryCount="1" timeout="10" keepAlive="60" >show slave status</heartbeat>
    <dbInstance name="instanceM1" url="ip4:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" dbDistrict="district1"
dbDataCenter="dataCenterA" primary="true">
    </dbInstance>
    <!-- can have multi read instances -->
    <dbInstance name="instanceS1" url="ip5:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" dbDistrict="district1"
dbDataCenter="dataCenterA" primary="false">
    </dbInstance>
    <dbInstance name="instanceS2" url="ip6:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" dbDistrict="district1"
dbDataCenter="dataCenterA" primary="false">
    </dbInstance>
    <dbInstance name="instanceS3" url="ip7:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" dbDistrict="district
1" dbDataCenter="dataCenterB" primary="false">
    </dbInstance>
    <dbInstance name="instanceS4" url="ip8:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" dbDistrict="district2"
dbDataCenter="dataCenterC" primary="false">
    </dbInstance>
  </dbGroup>
</dble:db>
```

配置注意事项：

1. 只有select语句生效
2. rwSplitMode具有更高优先级
3. 使用该功能至少需要bootstrap.cnf中配置district参数，对应的dbinstance配置dbDistrict参数
4. 本地读匹配优先级

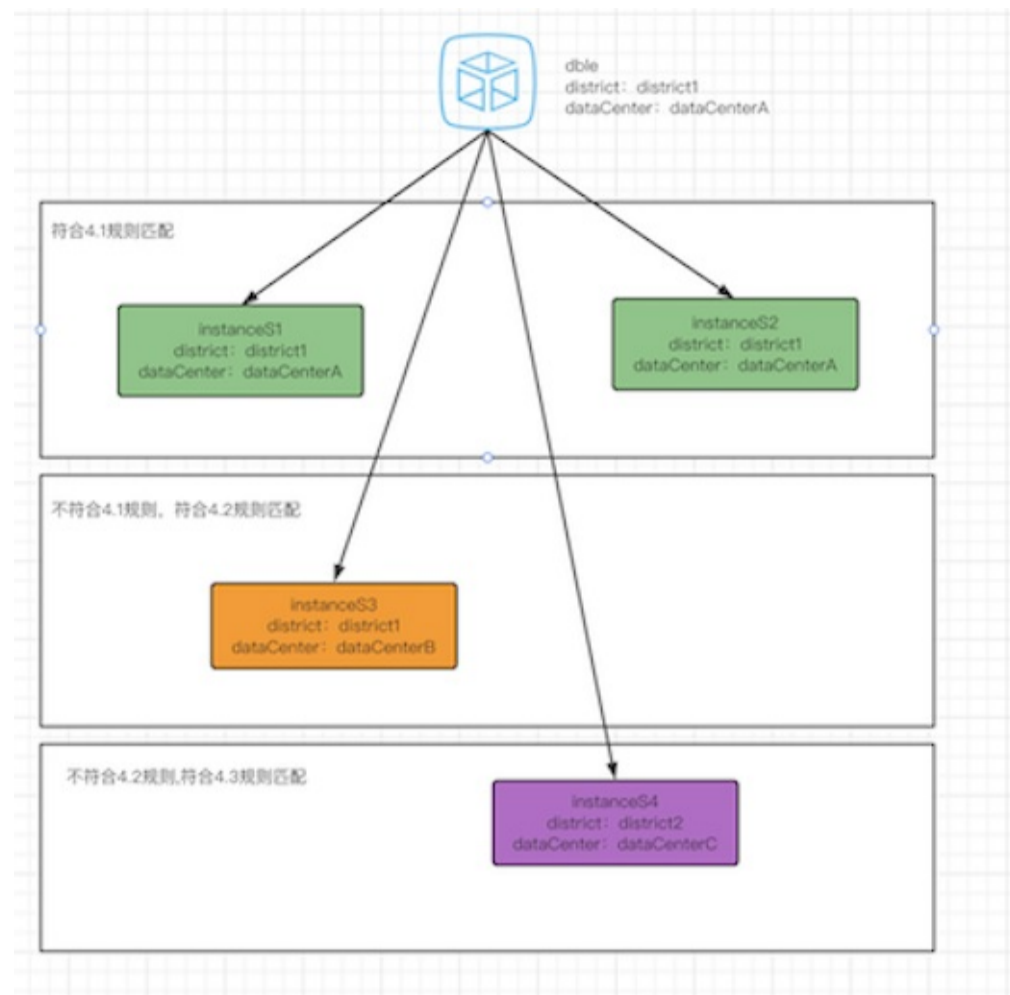
4.1 从district标签和dataCenter标签都符合的instance中挑选下发语句

4.2 找不到符合4.1中的条件或者符合4.1中条件的instance都处于异常状态，从只符合district标签匹配的instance中挑选下发语句

4.3 找不到符合4.2中的条件或者符合4.2中条件的instance都处于异常状态，从剩余的instance中挑选下发语句

5. hint语句按照hint的写法下发

举例说明： dble使用如上配置，使用dbGroup1的读写分离用户下发一条select语句



2.4 注解/Hint

Hint, 即注解。我们定义为:在要执行的SQL语句前添加额外的一段由注解组织的代码, 这样SQL就能按照编写者的意图执行, 这段代码称之为“注解”。注解的作用如下:

- 指定路由, 比如强制读写分离。
- 帮助dble支持一些不能实现的语句, 如单节点内存储过程的创建和调用, 如insert...select...;

原理解释: 分布式环境下执行SQL语句的流程是先进行SQL解析处理, 计算出路由信息后, 然后到对应的物理库上去执行; 若传入的SQL语句无法解析, 则不会去执行。注解则可以告诉解析器按照注解内的SQL (称之为注解SQL) 去进行解析处理, 解析出路由信息后, 将真正要执行的SQL语句 (称之为原始SQL) 发送到对应的物理库上去执行。

2.4.1 Hint语法

Hint语法有三种形式:

1. /*!dble:type=...*/
2. /*#dble:type=...*/
3. /* */(只适用于读写分离功能)

```
/*#dble: */" for mybatis and "/*!dble: */" for mysql
```

具体语法介绍请见: [Hint](#)

其中, type有4种值可选: shardingnode, db_type, sql, db_instance_url。每一种值的功能和形式详见各个部分的具体说明。

2.4.2 类型shardingnode

1. 形式
 - shardingnode=node
 - 其中, node为单个数据节点名,不能为多值(node定义参见配置1.5 sharding.xml)。
2. 功能
 - 为不方便路由或者不能路由的的语句指定具体的目的数据节点。

2.4.3 类型db_type

1. 形式
 - db_type=master或者db_type=slave
2. 功能
 - 帮助实现正确的业务逻辑, 强制读写分离。
3. 注意事项
 - delete, insert, replace, update, ddl语句不能使用db_type=slave进行注解。

2.4.4 类型sql

1. 形式
 - sql=sql_statement
2. 功能
 - 用sql_statement的路由结果集作为实际sql语句的执行数据节点。支持存储过程。

2.4.5 类型db_instance_url

1. 形式
 - 类型db_instance_url=ip:port
2. 功能
 - 在读写分离场景下, 可直接下发到相应的mysql实例
3. 注意事项
 - 运行delete, insert, replace, update, ddl语句时需考虑mysql节点的read_only属性

2.4.6 注意事项

写注解需要注意如下事项:

- dble的注解和MySQL原生注解含义不同, 想通过MySQL原生注解来设置变量或者指定索引是无法得到预期结果的。如[#1169](#)
- 使用select语句作为注解SQL, 不要使用delete/update/insert 等语句。 delete/update/insert 等语句虽然也能用在注解中, 但这些语句在SQL处理中有一些额外的逻辑判断, 会降低性能, 不建议使用;
- 注解SQL 本身禁用表关联语句, 注解目的是路由计算, 如果本身写得过于复杂, 会影响路由计算;
- 使用hint做DDL需要额外执行reload @@metadata
- 使用hint做session级别的系统变量和环境变量可能不会生效, 请慎用

- 使用注解并不额外增加的执行时间；从解析复杂度以及性能考虑，注解SQL应尽量用最简单的SQL 语句，如select id from tab_a where id='10000'；
- 能不用注解也能够解决的场景，尽量不用注解。
- 在读写分离场景下，语句为/ xxx /的注释(纯注释，不包含sql)，uproxy会将该语句发往slave，dble会将该语句发往master。

2.5 分布式事务

- [2.5.1 XA事务概述](#)
- [2.5.2 XA事务的提交以及回滚](#)
- [2.5.3 XA事务的后续补偿以及日志清理](#)
- [2.5.4 XA事务的记录](#)
- [2.5.5 一般分布式事务概述](#)
- [2.5.6 检测疑似残留XA事务](#)

2.5.1 XA事务概述

2.5.1.1 XA事务概述

普通事务只能在单节点内部保证事务的完整性，如果事务要在不同的节点上执行，无法保证数据一致性，具有跨节点的数据一致性要求的场景需要使用2阶段协议实现分布式事务。由于以上普通事务的缺陷所以Dble引入了Mysql的XA事务来解决这个问题，MySQL5.7之前版本的XA事务存在一些问题，启用需要MySQL 5.7版本，否则无法保证数据不丢失。dble提供的分布式事务采用两阶段提交协议，目前还是弱XA事务，不能绝对保证跨节点数据的强一致性。

1. 事务开始前需要设置手动提交：set autocommit=0;
2. 使用命令开启XA事务：set xa=on;
3. 执行相应的SQL语句。
4. 对事务提交或者回滚，事务结束：commit/rollback;

分布式事务的性能开销比较大，所以只推荐在全局表的事务以及其他一些对一致性要求比较高的场景中使用。一般情况下尽量不要在同一个事务中运行跨节点的SQL语句。dble支持大事务，但一方面大事务会造成事务执行时间上升，事务信息规模扩大，导致系统性能下降；另一方面，大事务也容易造成潜在的事务数据不一致问题。因此，大事务一定要谨慎使用，一般建议单个事务的SQL语句不要超过100条。

以下提供一个JDBC使用XA事务的实例，注意，此例子为最简单的demo，切勿直接在生产环境上使用。

生产环境需要一些更健壮的代码，比如使用连接池(连接重用)时，在放回连接池之前需要将系统变量设置回初始值。

```
public class XaDemo {

    public static final String URL = "jdbc:mysql://localhost:8066/testdb";
        //在这里也可以使用jdbc:mysql://127.0.0.1:8066?sessionVariables=xa=1
        //进行替代，在这种情况下不需要执行set xa = 1
    public static final String USER = "root";
    public static final String PASSWORD = "123456";

    public static void main(String[] args){
        try {
            //1.加载驱动程序
            Class.forName("com.mysql.jdbc.Driver");
            //2. 获得数据库连接
            Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
            //3.操作数据库，实现增删改查
            Statement stmt = conn.createStatement();
            stmt.execute("set xa = 1");
            //开始一个xa事务
            stmt.execute("begin");
            try {
                //执行相关数据操作的时候需要对于可能出现的错误进行catch
                //并在错误出现的时候对于整个事务进行rollback
                stmt.execute("insert into xa_test set id = 11,name = '3333'");
                stmt.execute("insert into xa_test set id = 22,name = '333'");
                stmt.execute("insert into xa_test set id = 3,name = '33'");
                //数据执行完成提交
                stmt.execute("commit");
            }catch (Exception e){
                System.out.println(" error "+e);
                //如果在数据操作的时候出现错误，将整个事务的操作回滚
                stmt.execute("rollback");
            }finally {
                stmt.close();
                conn.close();
            }
        }

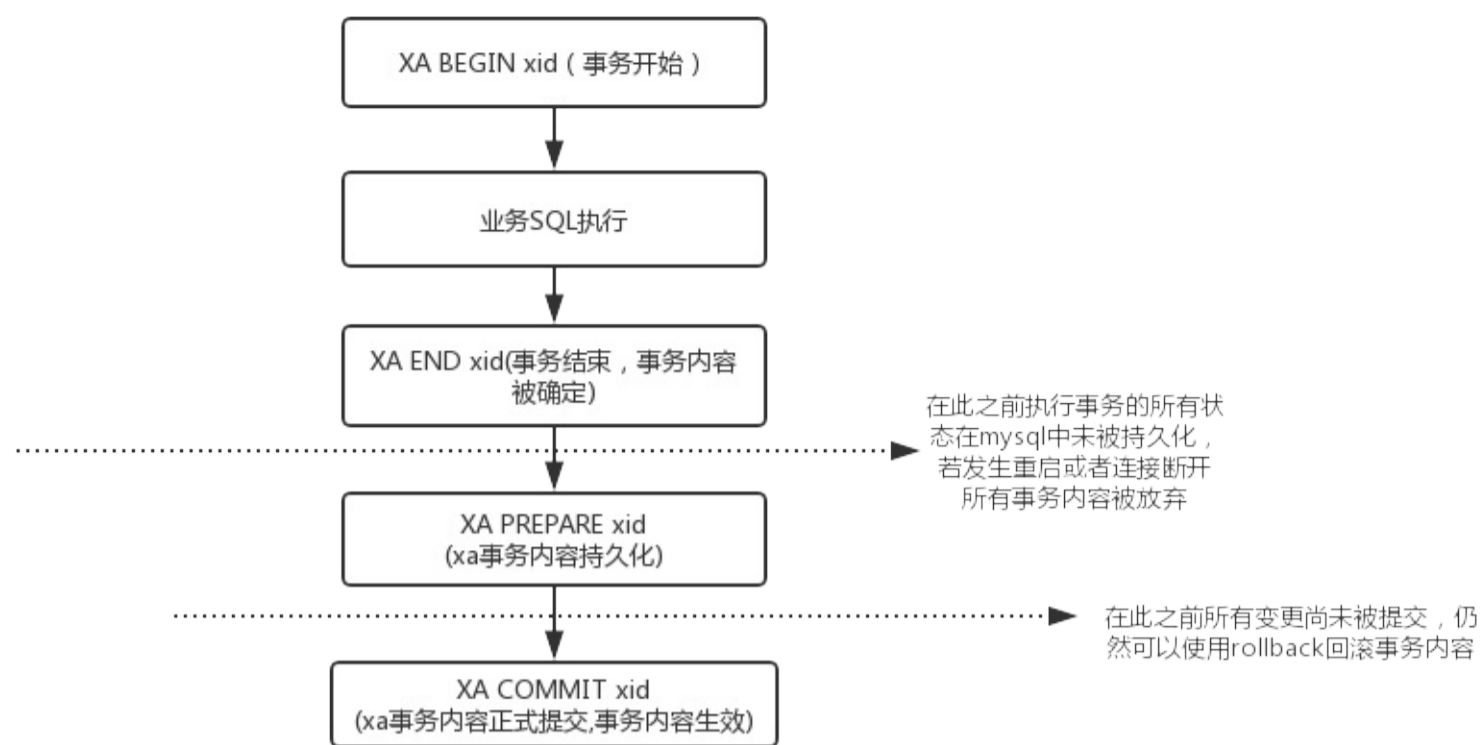
        }catch(Exception e){

        }

    }
}
```

2.5.1.2 XA事务的基础

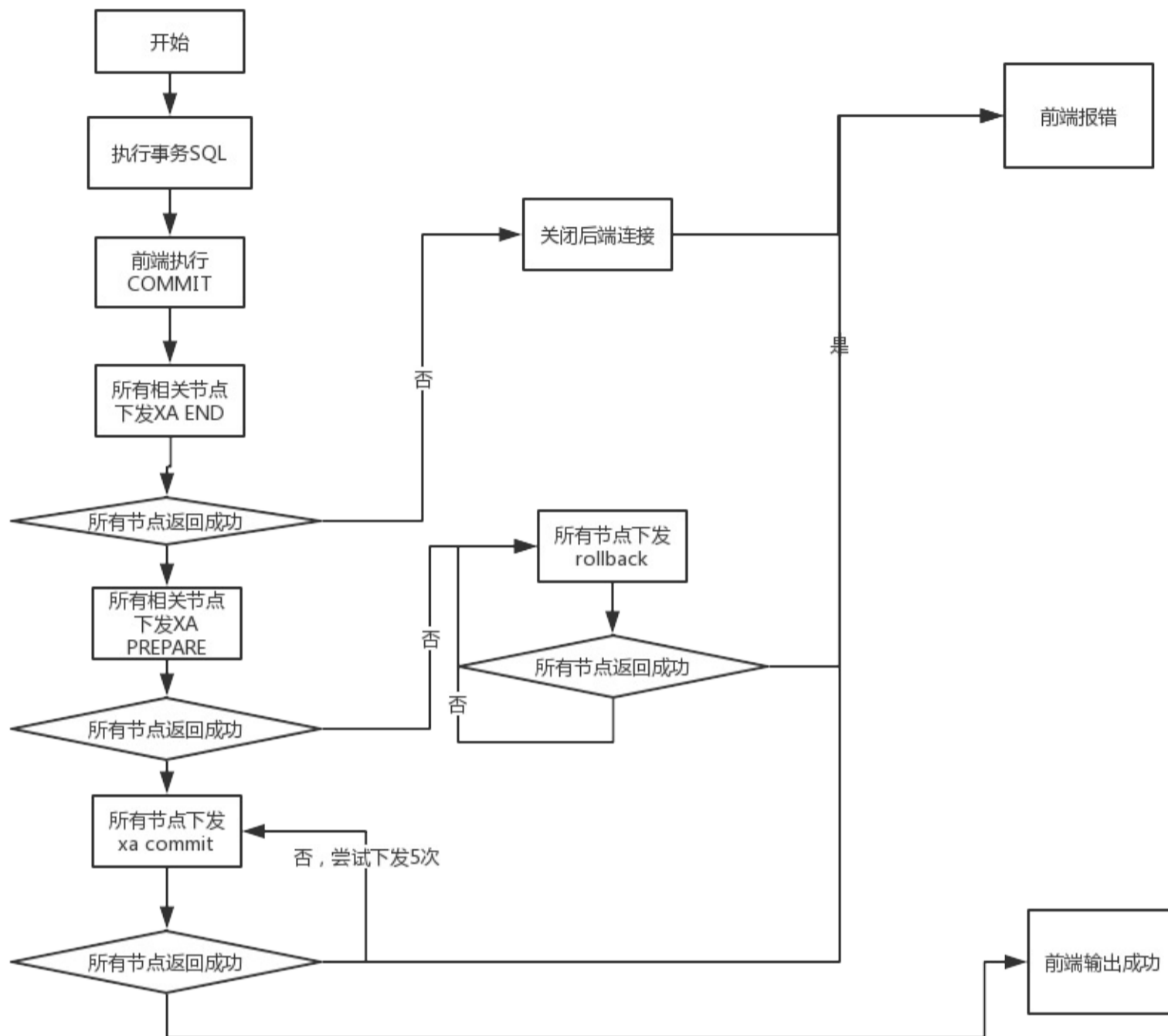
xa事务基于mysql5.7的xa事务的特性，其流程和特性由以下图所示：



2.5.2 XA事务的提交以及回滚

2.5.2.1 XA事务的提交

在Dble中采用二段提交的方式对于XA事务进行提交 具体的逻辑可见下图

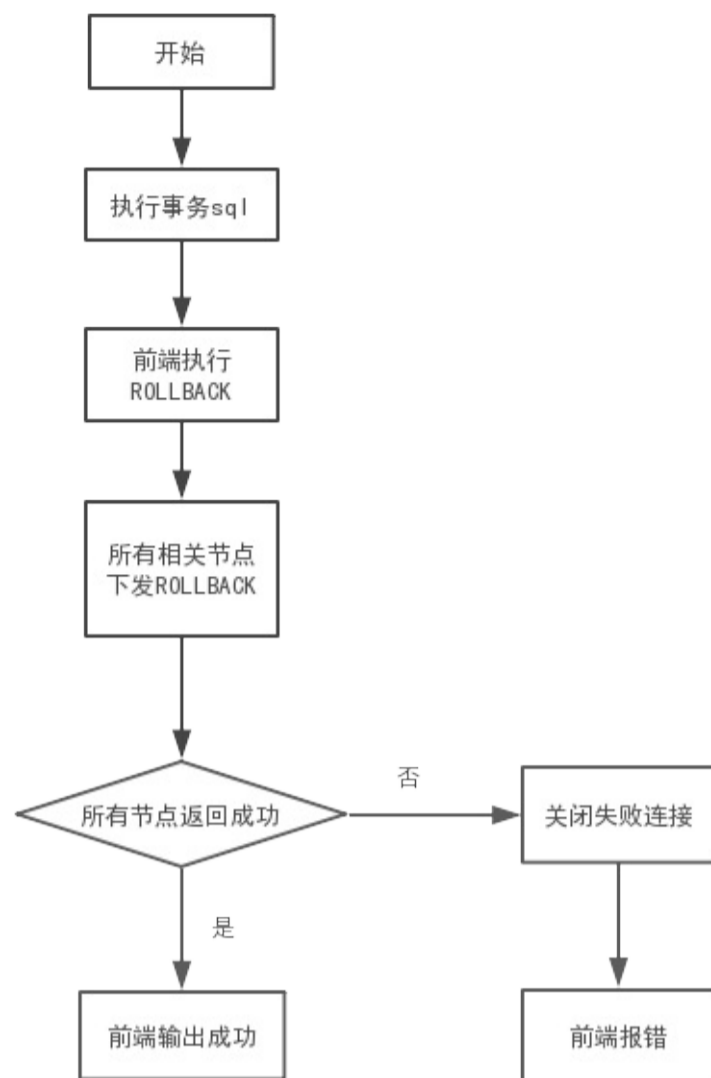


整体来说的处理原则如下：

1. 将XA事务的提交分为END PREPARE COMMIT三个部分
2. 如果在PREPARE下发之前有节点失败或报错，关闭所有后端连接放弃事务数据
3. 如果在PREPARE下发过程中发生失败，则回滚事务，所有节点下发ROLLBACK
4. 如果在COMMIT节点发生失败，则尝试重新下发，几次尝试未果将事务交给定时任务来继续重试

2.5.2.2 XA事务的回滚

相对来说回滚的逻辑就容易的多，直接在所有节点下发rollback。如果失败，直接关闭失败的连接。



2.5.2.3 XA事务重试机制

2.19.03.0之前的dble版本对于失败的事务，策略是将事务交给定时任务后台进行无限重试。2.19.03.0版本对这一过程进行了可配置化，并提供管理命令控制这一过程。

2.5.2.3.1 配置

2.19.03.0版本可以通过bootstrap.cnf文件中的xaRetryCount属性配置xa后台重试策略：

1. 当 xaRetryCount 等于0时，后台无限重试
2. 当 xaRetryCount 大于0时，后台尝试次数达到xaRetryCount后，重试停止且发送告警

若重试失败，会发出一条告警，重试成功后自动解决相应告警。

2.5.2.3.2 相关命令

2.19.03.0版本新增两个管理命令

1. 通过 `show @@session.xa` 查看后台重试xa事务信息
2. 通过 `kill @@xa_session id1,id2...` 取消指定session后台尝试xa事务

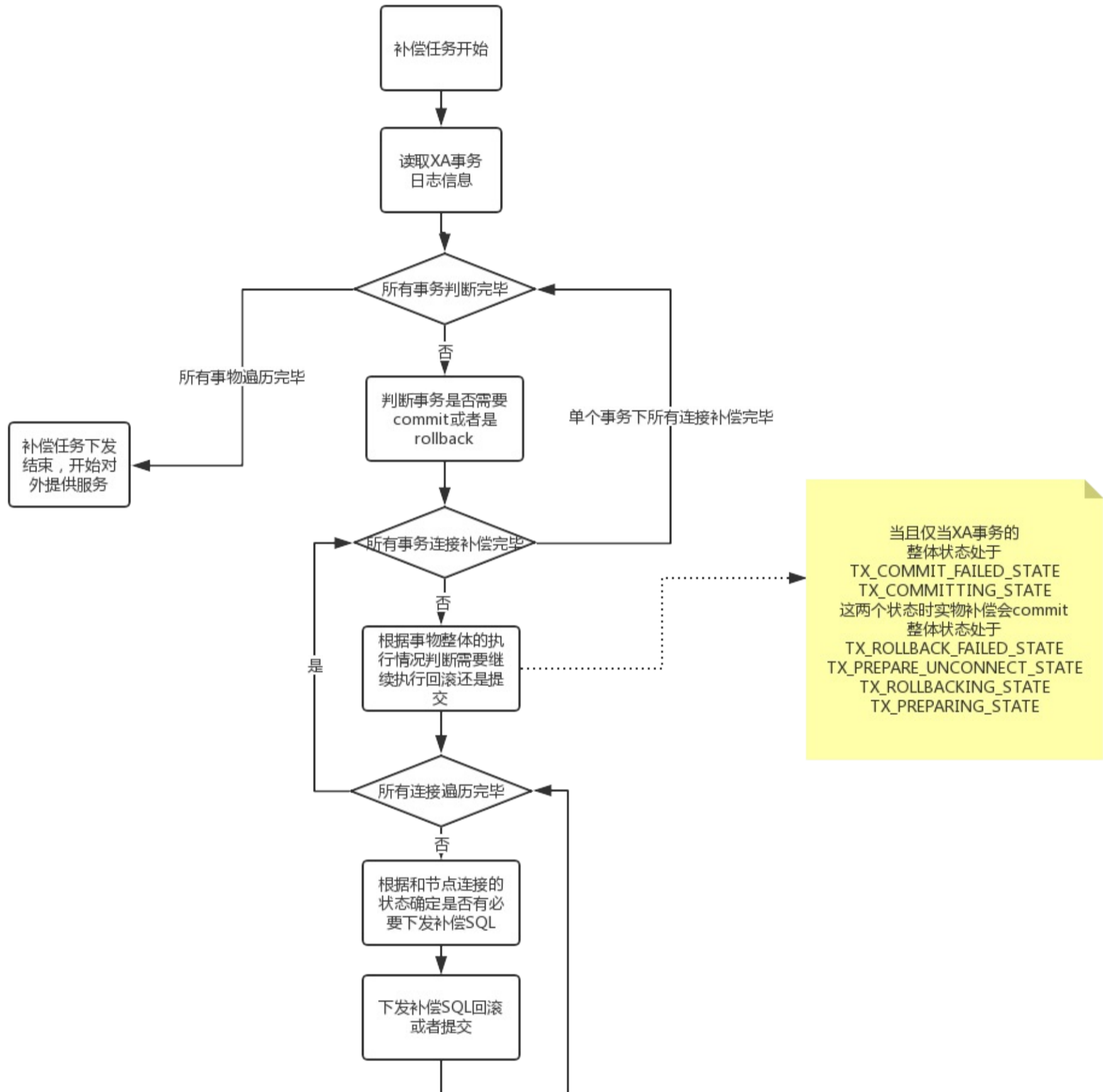
2.5.2.3.3 实现细节

重试时会先从后端连接池中获取连接，如果重试失败，相对应的连接会被关闭回收，更糟糕是重试多次都失败的话，有可能池内的连接都会被回收，此时就会新建后端连接来进行重试

2.5.3 XA事务的后续补偿以及日志清理

2.5.3.1 XA事务的补偿逻辑

由于XA事务是通过多次提交来达成最终的提交和回滚的，所以就会出现由于服务下线或者其他原因导致的提交或是回滚任务被执行了一半 在这种情况下发生的时候 dble通过XA事务记录的日志进行日志的处理以及补偿 详细通过在dble重启的时候在重启之前对于XA日志进行读取，并根据里面的内容对于XA事务进行回滚或者补偿 具体的补偿流程如下图所示：



注：XA事务的补偿不保证所有事务都能在补偿期间正确提交，若补偿期间依然因为SQL执行造成失败，只会在dble普通日志中体现或者告警（如果有配置告警功能）

2.5.3.2 XA事务的日志清理逻辑（xaLogClean）

xa事务的日志只保存最近一段时间的，所以需要定期的清理已经正确提交或者是正确回滚的日志数据 会根据bootstrap.cnf中的配置信息xaLogCleanPeriod定时将没有后续作用的数据做清除

2.5.3.3 XA事务定期连接检查（xaSessionCheck）

在dble提供服务期间如果发生commit(狭义commit,特指XA事务中prepare之后的commit动作)失败或者是rollback失败，那么失败的事务将被存储到内存队列中 并且进行定时的提交或回滚，直到事务被正确的提交或回滚 这是由于在xa事务的逻辑中prepare全部完成之后的事务已经都被成功持久化，仅需要提交即可

2.5.4 XA事务的记录

2.5.4.1 XA 事务过程中记录的内容

由于在Dble中采用两段提交的分布式事务，所以使用XA事务的时候对于DBLE本身就拥有了状态。状态就需要有文件或者其他方式的记录，其中关于XA事务细节的记录主要是记录以下几个部分

1. 事务ID
2. 事务状态
3. 事务中每个节点的连接host
4. 事务中每个节点的连接端口
5. 事务中每个节点连接最后的事务状态
6. 事务中每个节点连接的过期状态(没有实际作用)
7. 事务中每个节点连接对应的后端数据库

这里举例一个记录的实例

```
{
  "id": "'Dble_Server.1.15'",
  "state": "8",
  "participants": [
    {
      "host": "10.186.24.37",
      "port": "3308",
      "p_state": "8",
      "expires": 0,
      "schema": "db3",
      "tableName": "testdb.test1",
      "repeatTableIndex": 0
    },
    {
      "host": "10.186.24.37",
      "port": "3306",
      "p_state": "8",
      "expires": 0,
      "schema": "db2",
      "tableName": "testdb.test2",
      "repeatTableIndex": 0
    },
    {
      "host": "10.186.24.37",
      "port": "3308",
      "p_state": "8",
      "expires": 0,
      "schema": "db2",
      "tableName": "testdb.test3",
      "repeatTableIndex": 0
    },
    {
      "host": "10.186.24.37",
      "port": "3306",
      "p_state": "8",
      "expires": 0,
      "schema": "db1",
      "tableName": "testdb.test4",
      "repeatTableIndex": 0
    }
  ]
}
```

2.5.4.2 XA事务中status的标识字典

status	状态	解释
0	TX_INITIALIZE_STATE	XA事务处于初始化状态
1	TX_STARTED_STATE	XA事务处于开始状态，在事务开始直到提交或者回滚之前 XA事务的状态一直会保持此状态
2	TX_ENDED_STATE	XA END下发成功状态
3	TX_PREPARED_STATE	XA PREPARED成功状态
4	TX_PREPARE_UNCONNECT_STATE	XA PREPARED下发过程中连接被断开
5	TX_COMMIT_FAILED_STATE	XA COMMIT 下发失败
6	TX_ROLLBACK_FAILED_STATE	XA ROLLBACK 失败
7	TX_CONN_QUIT	后端mysql连接失败
8	TX_COMMITTED_STATE	XA 事务提交成功
9	TX_ROLLBACKED_STATE	XA 事务回滚成功
10	TX_COMMITTING_STATE	XA 事务正在提交
11	TX_ROLLBACKING_STATE	XA 事务正在回滚
12	TX_PREPARING_STATE	XA 事务正在下发prepare

2.5.4.3 XA事务记录的存储方式

一、本地文件方式

顾名思义在这种方式下，xa事务的状态将以本地记录文件的方式被存放到对应的文件中，具体的路径和文件名配置是

配置与bootstrap.cnf中的{xaRecoveryLogBaseDir}/{XaRecoveryLogBaseName}.log默认条件下文件会被储存在./xalogs/xalog-1.log

一般只在Dble单机状态下使用本地文件方式，使用集群时本地文件的方式将在集群状态下造成不可预知的错误

二、ZK存储方式

ZK存储方式不需要额外的配置，当Dble使用ZK配置时，自动默认XA事务记录的存储方式也会是ZK存储

具体的XA事务记录的内容保持不变，记录在dble/{clusterId}/XALOG/{myid} 的Key值中

2.5.4.4 其他补充

参与记录规则

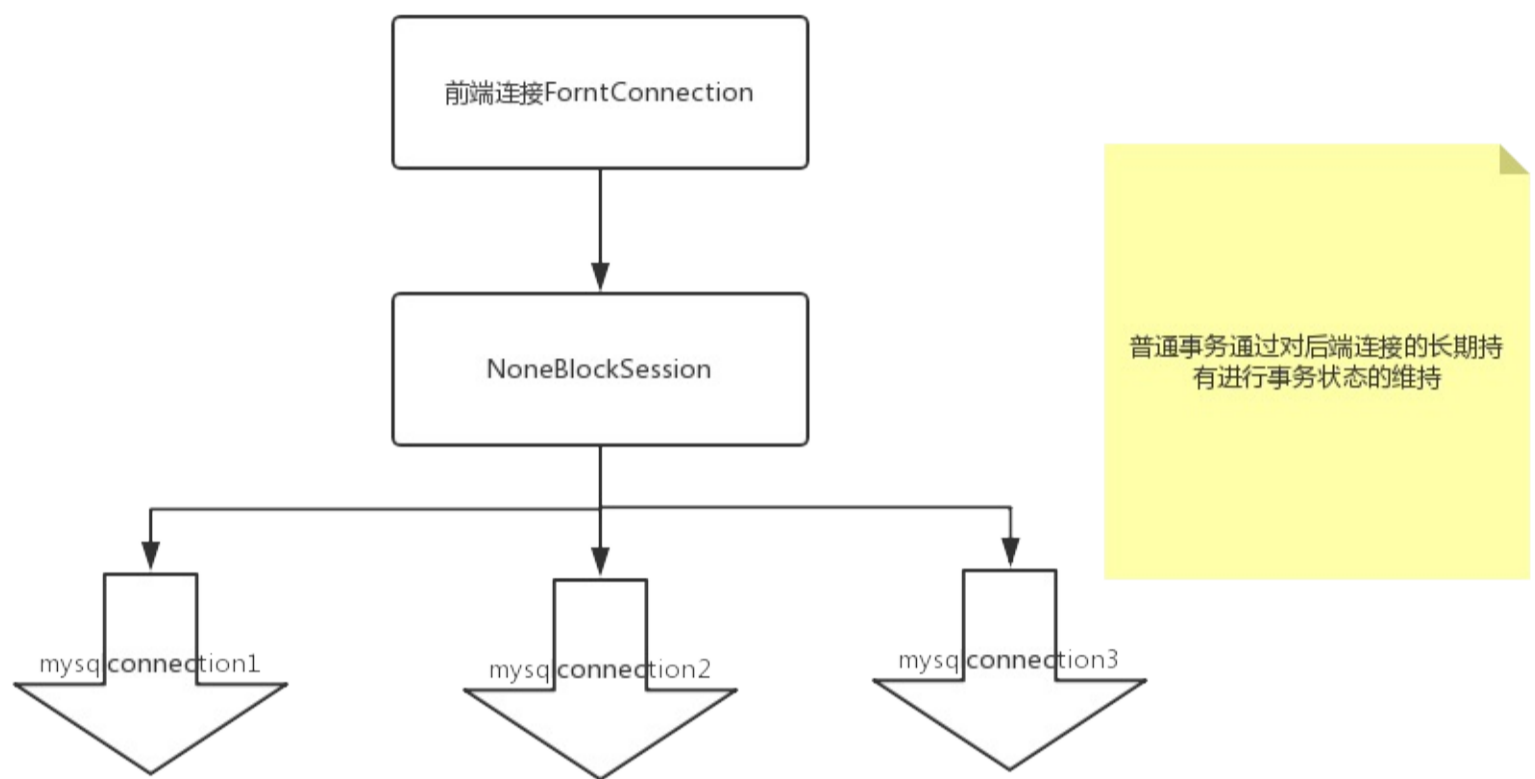
- 1、仅Sharding用户执行的相关事务，参与记录（RwSplit用户不参与统计）；
- 2、有效(真正意义上的开启或结束显式事务)的事务控制语句，如begin等，参与记录；
- 3、事务中，Modify类型sql，都参与记录；
- 4、非事务中，Modify类型且实际下发多个节点的sql，才参与记录；
(Modify定义：具有修改数据或结构的sql)

2.5.5 一般分布式事务概述

在分布式事务中整体的逻辑和mysql的事务逻辑类似，通过长期持有的连接来进行，每个前端连接frontconnection对应一个session，在dble的每个session中有对应的事务状态以及session所持有的后端连接集合target，在非事务状态下或者是autocommit状态下每次后端连接被使用完毕之后就会被移除target并释放回空闲连接池，但是在事务开启的状态下，在SQL执行完毕的时候connection会在target中长期储存，直到session发起commit或者是rollback。

综上所述我们可以看到，Dble中的普通分布式事务其实就是后端mysql事务的集合，并且这个事务是没有文件记录的，由于mysql的事务特性在事务发生的过程中若断开连接等同于放弃事务，所以可能出现在commit的过程中由于各种意外情况导致事务的部分提交，例如连接后端四个节点dn1,dn2,dn3,dn4在提交commit进行依次下发的時候dn1,dn2,dn3都提交成功，但是dn4由于网络意外提交失败，导致了预期执行的部分内容丢失，并且由于dn1,dn2,dn3已经提交成功无法进行数据回滚，只能进行人工的数据补偿。

其逻辑图如下：



:

2.5.6 检测疑似残留XA事务

2.5.6.1 介绍&背景

一、介绍Xid

Xid:一个下发至各个节点中的xa事务名称.

Sharding用户登陆的前端连接在开启XA事务(set xa = on)下会分配到一个Xid_Session(简称);

Xid_Session格式: `Dble_Server.{instanceName}.{xaIDInc}` 其中 `instanceName` 为dble的实例命名, `xaIDInc` 全局自增id.

在路由准备下发至节点阶段中,会在Xid_Session之后拼接具体 `db` (物理库名)获取最终的Xid,用来下发至各个节点;

Xid格式: `Dble_Server.{instanceName}.{xaIDInc}.{db}`

`xaIDInc` 起始值的定义:在dble启动时, `xaIDInc`起始值为1(默认情况);但如果存在`xaRecovery`日志且其中有存留上次Xid记录场景下, `xaIDInc`初始值则为上次Xid中`xaIDInc`值+1.

注意: `xaRecovery`日志: `{xaRecoveryLogBaseDir}/{XaRecoveryLogBaseName}.log`,默认为`xalogs/xalog-1.log`

二、背景

节点层面存留dble生成的Xid;在`xaRecovery`日志清理的情况下启动dble,意味着`xaIDInc`从1开始增长;当增长到残留Xid中`xaIDInc`时,会出现'The XID already exists'报错;此功能无需额外配置,可以在dble启动阶段、运行阶段将疑似残留XA事务问题很好的暴露出来.

2.5.6.2 检测机制

检测对象:被Sharding表关联dbGroup中的主节点 疑似残留Xid的格式:符合Xid_Session或者Xid的格式

疑似残留Xid的正则表达式: `Dble_Server.{instanceName}.(\d)(.[^\s]+)?`

疑似残留Xid的日志关键字: Suspected residual xa transaction.....

一、启动阶段

只要满足疑似残留Xid的正则表达式,则被视为疑似残留Xid,直接启动失败;

二、运行阶段

满足疑似残留Xid的正则表达式且节点中的`xaIDInc`小于dble内部(正在使用)`xaIDInc`,则被视为疑似残留Xid.

设置检测周期

默认每隔300s的定时检测任务,见bootstrap.cnf中 `-DxaIdCheckPeriod=300`

管理端命令:

```
reload @@xaIdCheck.period=60; -- 表示开启(或者调整)以60s为周期的定时检测任务
reload @@xaIdCheck.period=0; -- 值小于等于0时,表示关闭定时检测任务
```

日志中关键字: Start XaldCheckPeriod、Stop XaldCheckPeriod

2.5.6.3 其他检测

一、查看所有节点下的xa情况

管理端,查询dble_xa_recover表查看所有存活的主节点下XA情况.

二、查看dble中正在使用的Xid

管理端,查询session_connections表(或show @@connection)的结果中xa_id字段,xa_id值为Xid_Session格式表示处于XA事务中.

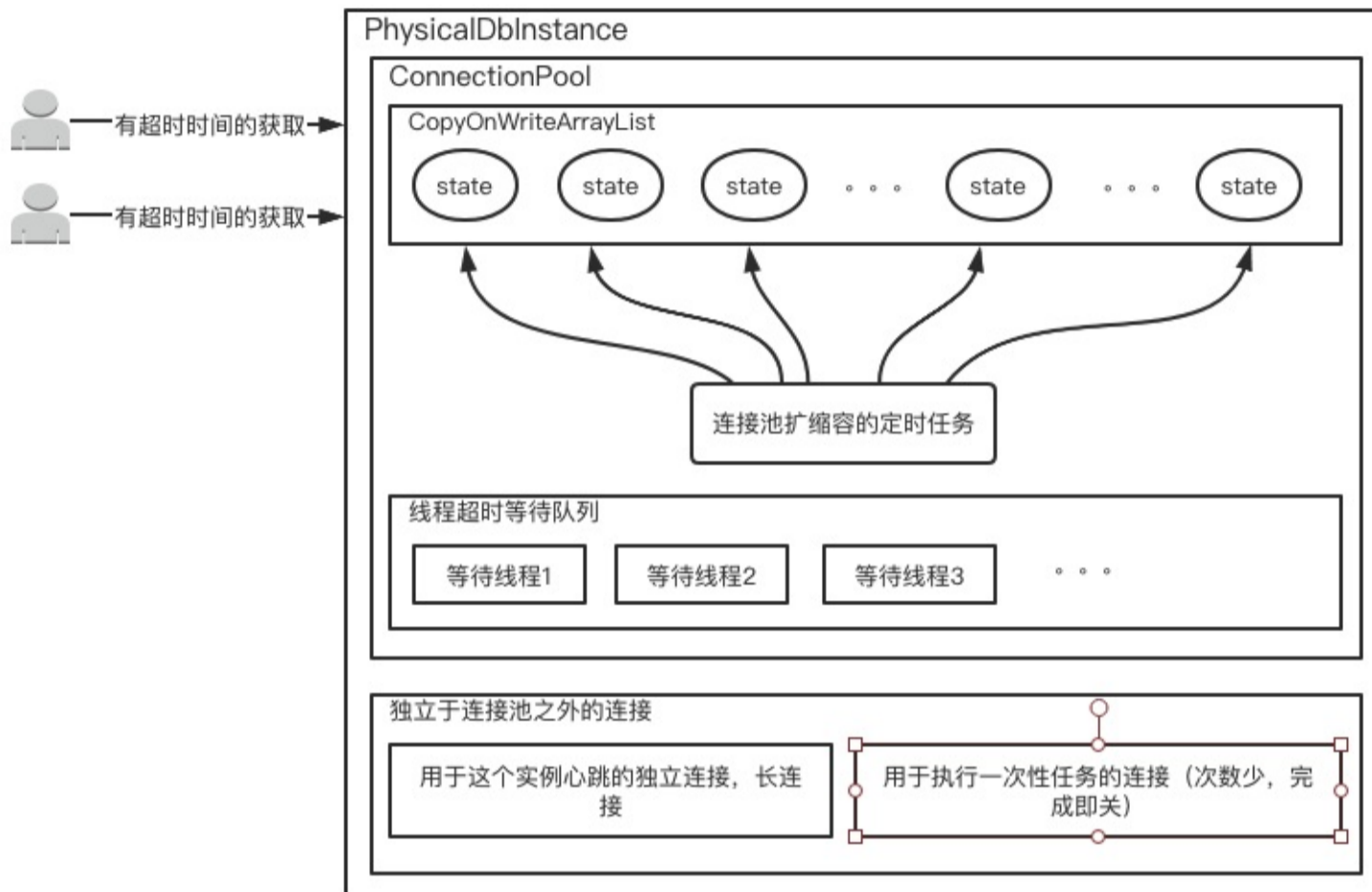
2.6 连接池管理

在dble中每个后端MySQL节点由 PhysicalDbInstance 表示，PhysicalDbInstance中维护了两类连接：

1. 大部分业务使用的连接由连接池管理
2. 独立于连接池之外的连接，这类连接主要有两类：MySQL实例的心跳连接和用于OneTimeJob的一次性任务的连接，这种连接是一次性的，用完即关，次数也比较少。

2.6.1 dble后端连接池管理

后端连接池使用 CopyOnWriteArrayList 存储该MySQL实例的全量连接，通过连接的state状态来维护连接的初始化，借出，空闲，心跳，移除状态。连接池初始化之后会维护一个evictor线程来维持连接池的扩缩容以及空闲连接的有效性。结构如下图所示：



2.6.1.1 连接获取

业务向连接池请求获取后端连接，会遍历连接池中的全量连接直到找到第一个空闲连接。若连接池中当时没有空闲连接，则线程会进入超时等待队列，在超时时间内未获取到连接，前端报错。

2.6.1.2 连接释放

后端连接处理完业务之后，状态会被置为空闲，并且处理线程会唤醒在超时等待队列中的线程来重新获取连接。

2.6.1.3 连接池扩缩容

连接池初始化之后会维护一个evictor线程来维持连接池的扩缩容以及空闲连接的有效性，evictor线程是一个定时任务。

扩容：当空闲连接数小于minCon时，维持连接池中的空闲连接在minCon的数量上，每次扩容的连接数量通过以下公式计算： $\min(\text{配置的最小空闲连接数} - \text{当前连接池中空闲连接数}, \text{配置的最大连接数} - \text{连接池中的总连接数}) - \text{正在创建的连接数}$ ，若数量大于0，则创建该数量的连接。

缩容：当空闲连接数大于minCon时，维持连接池中的空闲连接在minCon的数量上，每次关闭的连接数量通过以下公式计算： $(\text{连接池中的最小连接数} - \text{配置的最小连接数}) > 0 \ \&\& \ \text{连接达到 idleTimeout}$ 。

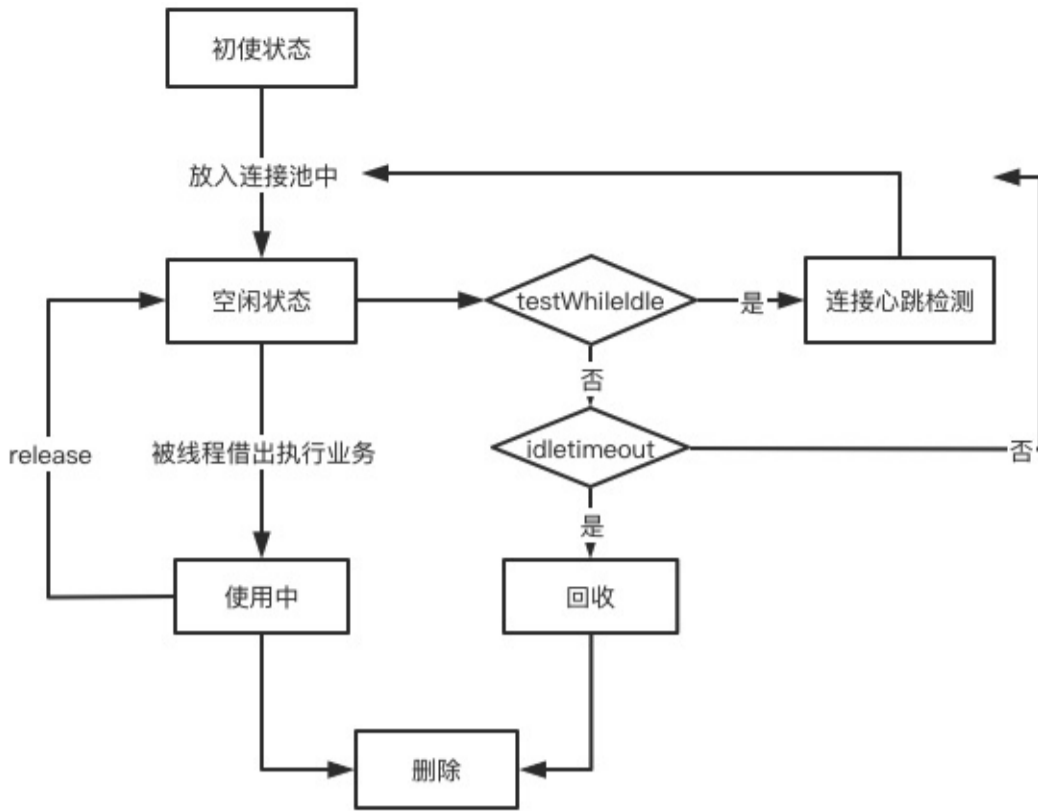
2.6.1.4 连接有效性检测

在连接的不同阶段，提供对连接有效性的检测手段。

- testOnCreate为true，在连接被创建后，会发送ping命令探测连接有效性，若在connectionHeartbeatTimeout没有收到结果，会关闭连接。
- testOnBorrow为true，在连接被借出后，会发送ping命令探测连接有效性，若在connectionHeartbeatTimeout没有收到结果，会关闭连接。
- testOnReturn为true，在连接被返回后，会发送ping命令探测连接有效性，若在connectionHeartbeatTimeout没有收到结果，会关闭连接。
- testWhileIdle为true，对所有空闲连接，发送ping命令探测连接有效性，若在connectionHeartbeatTimeout没有收到结果，会关闭连接。

2.6.2 连接状态管理

由上文我们知道，后端连接池使用 CopyOnWriteArrayList 存储该MySQL实例的全量连接，通过连接的state状态来维护连接的初始化，借出，空闲，心跳，移除状态。下面是连接状态的跃迁图：



2.6.3 连接流量控制阈值

dble对分库分表的流量有控制功能，详见[2.25 dble流量控制](#)。

后端连接的高水位(flowHighLevel)和低水位(flowLowLevel)的配置也在这里体现，单位字节

2.6.4 连接池属性

属性名	默认值	单位	含义
testOnCreate	false	无	连接创建后是否检测有效性
testOnBorrow	false	无	连接被借出后是否检测有效性
testOnReturn	false	无	连接被返回时是否检测有效性
testWhileIdle	false	无	连接空闲时是否检测有效性
connectionTimeout	30000 (30s)	毫秒	获取连接的超时时间
connectionHeartbeatTimeout	20	毫秒	空闲连接检测后的超时时间
timeBetweenEvictionRunsMillis	30000 (30s)	毫秒	扩缩容线程的检测周期
idleTimeout	600000 (10 minute)	毫秒	连接空闲多久之后被回收
heartbeatPeriodMillis	10000 (10s)	毫秒	连接池的心跳周期
evictorShutdownTimeoutMillis	10000 (10s)	毫秒	扩缩容线程停止的超时时间
flowHighLevel	4194304	字节	后端连接流量控制的高水位
flowLowLevel	262144	字节	后端连接流量控制的低水位

示例：

```

<?xml version="1.0"?>
<!DOCTYPE dbInstance SYSTEM "db.dtd">
<dbInstance xmlns:dble="http://dble.cloud/">

  <dbGroup name="dbGroup1" rwSplitMode="1" delayThreshold="10000">
    <heartbeat errorRetryCount="1" timeout="10" keepAlive="60" >show slave status</heartbeat>
    <dbInstance name="instanceM1" url="ip4:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="true">
      <property name="testOnCreate">false</property>
      <property name="testOnBorrow">false</property>
      <property name="testOnReturn">false</property>
      <property name="testWhileIdle">true</property>
      <property name="connectionTimeout">30000</property>
      <property name="connectionHeartbeatTimeout">20</property>
      <property name="timeBetweenEvictionRunsMillis">30000</property>
      <property name="idleTimeout">600000</property>
      <property name="heartbeatPeriodMillis">10000</property>
      <property name="evictorShutdownTimeoutMillis">10000</property>
      <property name="flowHighLevel">4194304 </property>
      <property name="flowLowLevel">262144 </property>
    </dbInstance>

    <!-- can have multi read instances -->
    <dbInstance name="instanceS1" url="ip5:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="false">
      <property name="heartbeatPeriodMillis">60000</property>
    </dbInstance>
  </dbGroup>
</dbInstance>

```

2.6.5 dble后端连接池的心跳管理

dble后端MySQL节点的心跳管理是通过定时任务来完成的，检测周期由heartbeatPeriodMillis来控制。dble会对每一个后端MySQL节点持有一个长连接，定期发送心跳语句，根据返回结果的不同将连接池标记为不同的状态。若心跳异常，会影响evictor线程的扩缩容。

2.6.5.1 心跳周期内的不同阶段

我们可以将每个心跳周期简单划分为两个阶段：

- 检测阶段：心跳检测发起到收到回复的阶段
- 空闲阶段：心跳返回后到下一个心跳检测发起的阶段

2.6.5.2 心跳状态

dble的心跳状态有四种：

- init状态：初始状态，具体指收到第一个心跳响应报文前的状态
- ok状态：收到一次正常的心跳返回后的状态
- timeout状态：最近的一次心跳在HeartbeatTimeout时间段内没有收到响应
- error状态：心跳语句返回错误或者心跳连接异常都可能导致此种状态，dble里面会有重试机制来预防网络抖动等网络方面的异常。

2.6.5.3 心跳重试

此处的心跳重试分为三种情况：第一种是心跳语句返回错误导致的重试，第二种是心跳连接关闭导致的重试，第三种是心跳超时后才收到响应导致的重试。

- 对于心跳语句返回失败，dble会立即将连接池状态置为error状态，随即会发送errorRetryCount次心跳，若有一次心跳正常，心跳恢复成ok状态。
- 对于心跳连接关闭引起的失败，dble会在接下来的时间立即发送errorRetryCount次心跳，若有一次心跳正常，则停止重试，但如果都失败，则将连接池状态置为error状态
- 对于标记为timeout状态后收到姗姗来迟的OK响应，则将会被重置为init状态并立即发送一次心跳。

2.6.6 连接池补充说明

在dble中，dble会与配置的数据库建立连接，db.xml中的dbInstance标签中配置了具体的数据库实例，dble与数据库的连接通过连接池来管理。

- 每个dbInstance的连接池都是独立的，连接池的连接数通过dbInstance中的maxCon参数和minCon参数来控制
- 每一个dbInstance中配置的mysql实例，dble都会建立一个连接池来管理。只有在rwSplitMode=0时，主实例会建立连接池，从实例不会建立连接池

举例：

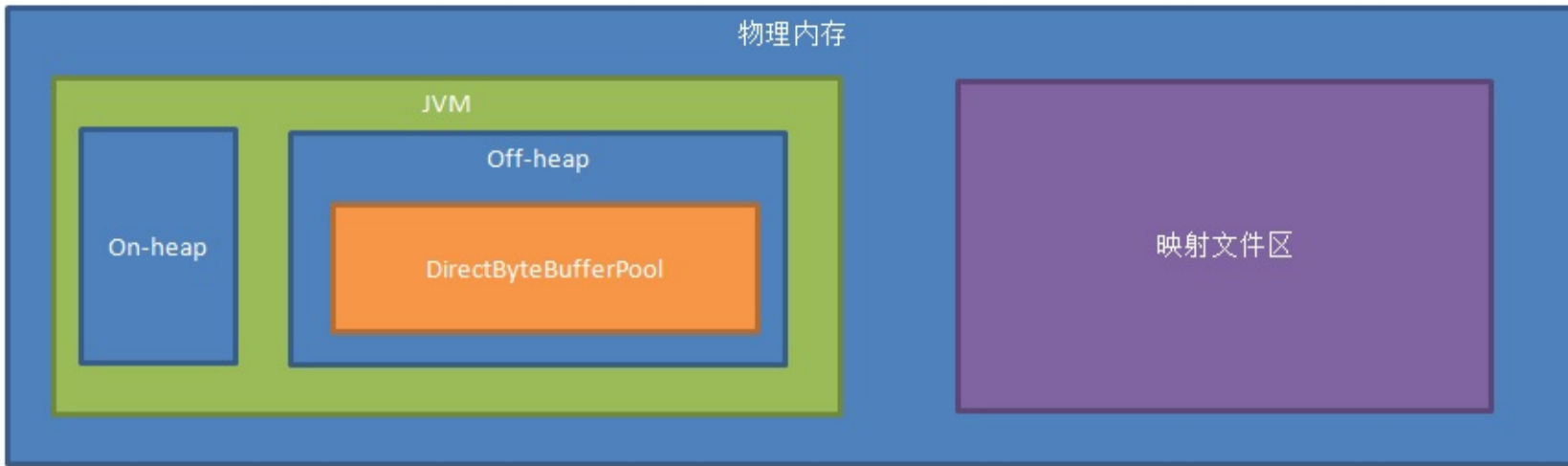
```
<dbGroup name="dbGroup1" rwSplitMode="1" delayThreshold="10000">
  <heartbeat errorRetryCount="1" timeout="10" keepAlive="60">show slave status</heartbeat>
  <dbInstance name="instanceM1" url="ip4:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="true">
</dbInstance>
</dbGroup>
<dbGroup name="dbGroup2" rwSplitMode="1" delayThreshold="10000">
  <heartbeat errorRetryCount="1" timeout="10" keepAlive="60">show slave status</heartbeat>
  <dbInstance name="instanceM2" url="ip4:3306" user="your_user" password="your_psw" maxCon="100" minCon="10" primary="true">
</dbInstance>
</dbGroup>
<dbGroup name="dbGroup3" rwSplitMode="1" delayThreshold="10000">
  <heartbeat errorRetryCount="1" timeout="10" keepAlive="60">show slave status</heartbeat>
  <dbInstance name="instanceM3" url="ip5:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="true">
</dbInstance>
</dbGroup>
```

配置注意事项:

- 1.instanceM1和instanceM2虽然配置了相同mysql，dble是根据dbInstance来建立连接池，上述例子中总共会建立三个连接池
- 2.instanceM1和instanceM2虽然配置了相同mysql，连接池中都是相互独立的，instanceM1的连接池中连接数量最大为200，最小为50，instanceM2的连接池中连接数量最大为100，最小为10。二者相互独立

2.7 内存管理

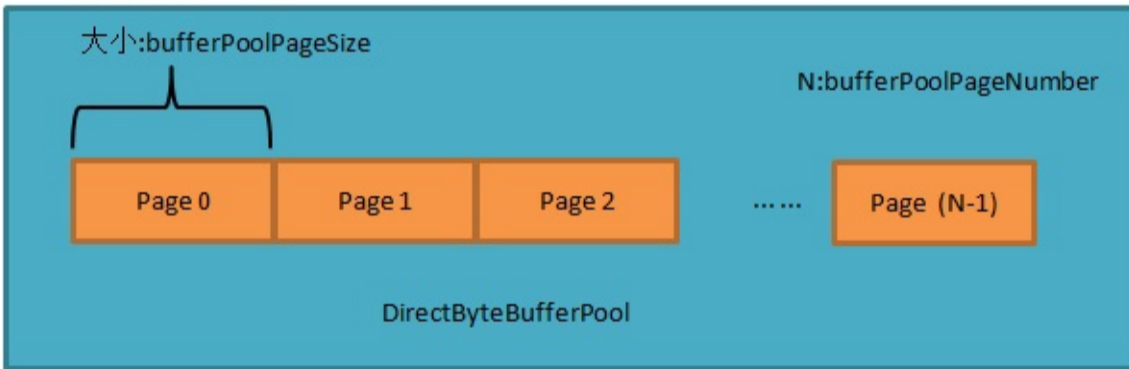
2.7.1 内存结构概览：



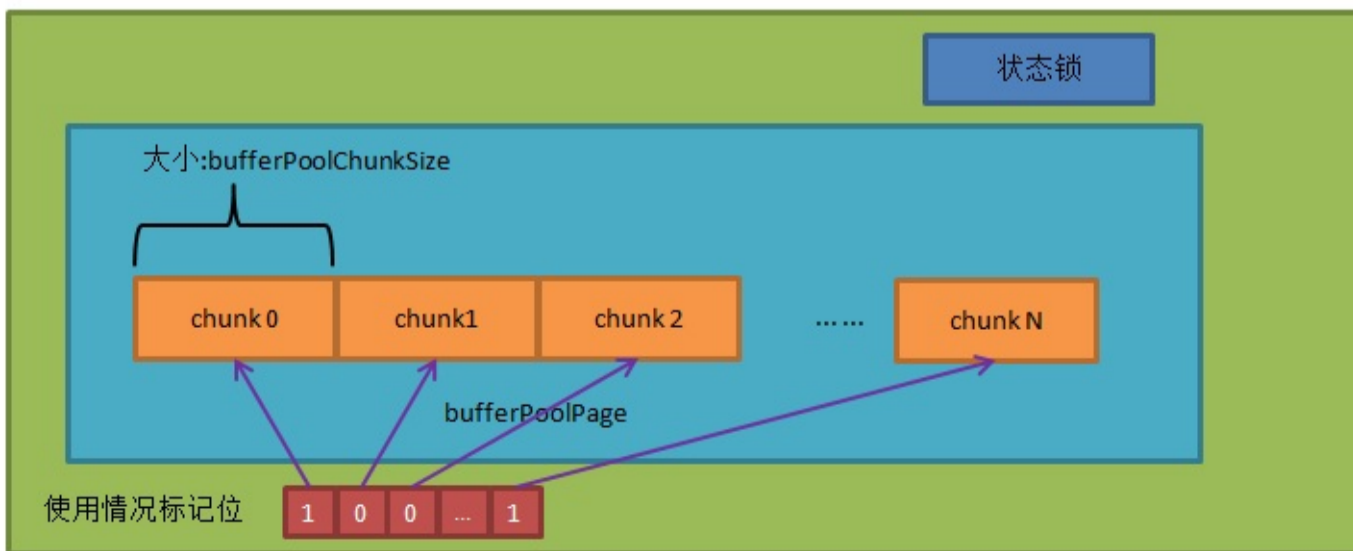
- On-Heap 大小由JVM 参数Xms ,Xmx 决定，就是正常服务需要的内存，由jvm自动分配和回收。
- Off-Heap大小由JVM 参数 XX:MaxDirectMemorySize 大小确定。
- DirectByteBufferPool 大小 = $bufferPoolPageNumber * bufferPoolPageSize$
 $bufferPoolPageNumber$ 和 $bufferPoolPageSize$ 可在bootstrap.cnf 配置， $bufferPoolPageSize$ 默认为2M, $bufferPoolPageNumber$ 默认为 $(MaxDirectMemorySize * 0.8 / bufferPoolPageSize)$, 向下取整。
- 映射文件区不在JVM之内
 大小计算方法求得值 $tmpMin = \text{Min}(\text{物理内存的一半}, \text{free内存})$
 可在映射区保存的文件个数= $(\text{向下取整}(tmpMin/mappedFileSize))$
 真实使用值= 可在映射区保存的文件个数* $mappedFileSize$ ($mappedFileSize$ 默认64M)
 其中 $mappedFileSize$ 可由bootstrap.cnf 指定
 注：这里是通过单例模式，一次性确定映射文件区的大小，如果运行过程中，内存被其他进程占用，这里可能存在风险

2.7.2 DirectByteBufferPool

BufferPool结构概览：



单个bufferPoolPage的内部结构:



1.用途:

1.1 网络读写时使用

1.2 中间结果集暂存时用于写数据前的缓存buffer

2.初始化:

按照参数,初始化为bufferPoolPageNumber个页,每个页大小为bufferPoolPageSize

3.内存分配

3.1 分配大小

如果不指定分配大小,则默认分配一个最小单元(最小单元由bufferPoolChunkSize决定,默认大小为4k,请最好设为bufferPoolPageSize的约数,否则最后一个会造成浪费)。

如果指定分配大小,则分配放得下分配大小的最小单元的整数倍(向上取整)。

总之,大小为 $M * \text{bufferPoolChunkSize}$

3.2 分配方式

遍历缓冲池从N+1页到bufferPoolPageNumber-1页(上次分配过的记为第N页)

对单页加锁在每个页中从头寻找未被使用的连续M个最小单元

如果没找到,再从第0页找到第N页

以上成功后更新上次分配页,标记分配的单元

如果找不到可存放的单页(比如大于bufferPoolPageSize),直接分配On-Heap内存

4.内存回收

4.1 如果是On-Heap内存

直接clear,等GC回收

4.2 如果是Off-Heap内存

遍历所有页,找到对应页

对单页加锁,到对应页的对应块的位置,标记为未使用

2.7.3 处理中间结果集过大时内存使用

目前dble对每个session的内存管理如下:

- Join内存管理,对应上限参数joinMemSize(默认4M),用于管理join操作暂存的数据
- Order内存管理,对应上限参数orderMemSize(默认4M),用于管理排序操作暂存的数据
- Other内存管理,对应上限参数otherMemSize(默认4M),用于管理distinct、group、nestloop等操作总暂存的数据

每一个复杂查询中,当子查询单元需要在中间件当中暂存数据的时候,数据会存在Heap内存当中,但如果当前存储使用的内存大于4M,则需要写内存映射文件。

如果内存映射文件个数达到上限(参见概览中的可在映射区保存的文件的个数),则会去写硬盘。

写文件的时候,当单个文件大于mappedFileSize时,会将文件拆分。

注1:内存映射文件总大小在第一次使用时就确定,有风险

注2:写硬盘时候的缓冲区是从DirectByteBufferPool申请的chunk大小的

2.8 集群同步协调&状态管理

2.8.1 概述

大多数时候，db1e节点是无状态的，所以可以用常用的高可用/负载均衡软件来接入各个节点。

这里不讨论各个负载均衡软件的使用。

主要讨论一下某些情况下需要同步状态的操作和细节。

注：本部分内容需要额外部署zookeeper用于管理集群的状态和同步。

2.8.2 相关配置

2.8.2.1 cluster.conf

以ZK为例

```
# 开启集群模式
clusterEnable=true
# 集群模式元数据中心为zk
clusterMode=zk
# zk地址
clusterIP=10.186.19.aa:2281,10.186.60.bb:2281
#zk为db1e提供的根目录
rootPath=/db1e
#本组db1e的组名
clusterId=cluster-1
# 是否需要同步Ha状态
#needSyncHa=false
# 拉取一致性binlog线的超时时间
#showBinlogStatusTimeout=60000
#自增序列类型
sequenceHandlerType=2
#自增序列默认开始时间
#sequenceStartTime=2010-11-04 09:42:54
#自增序列类型为3时，instanceId是否由ZK生成
#sequenceInstanceByZk=true
```

2.8.2.2 bootstrap.conf

instanceName 实例名称，用于发起集群任务或者汇报自己完成集群任务的标记 instanceId （自增序列使用，根据类型范围为0到1023 或者0到511）

2.8.3 初始化状态

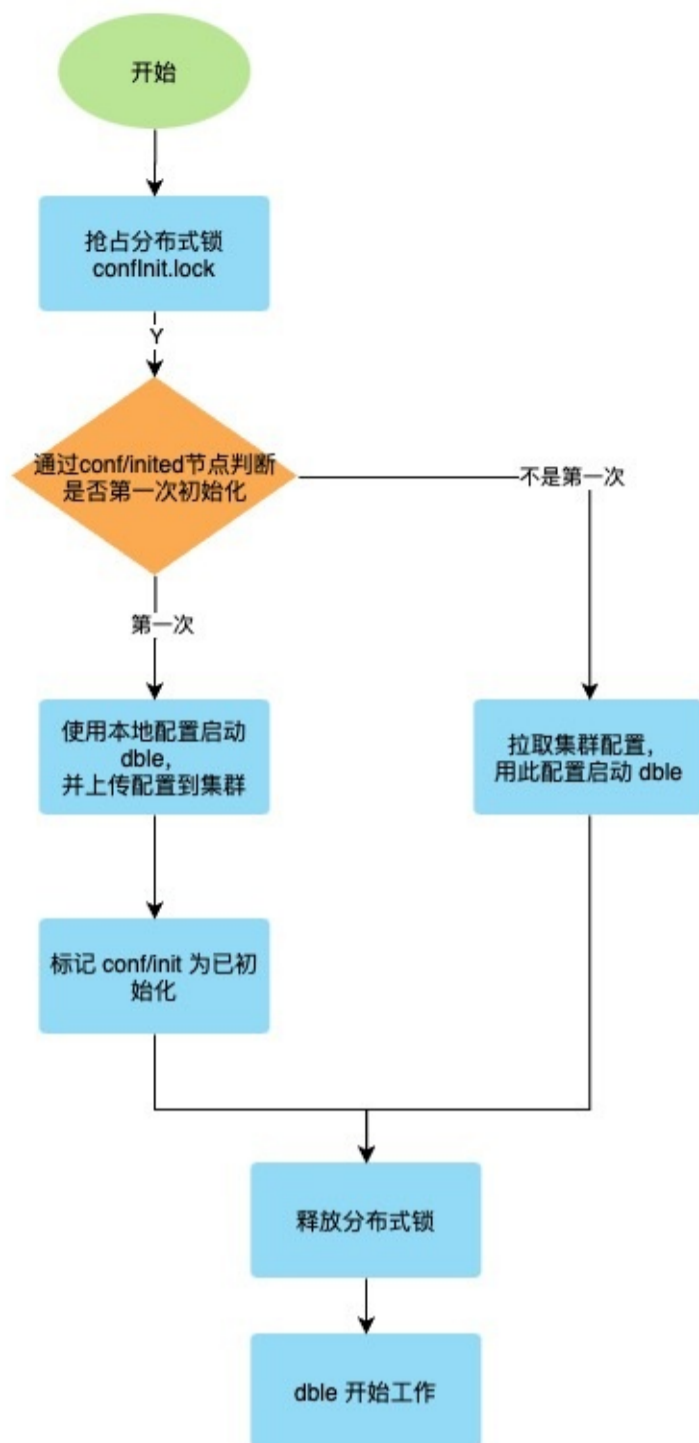
方式1.

通过执行脚本init_zk_data.sh方式将某个节点的配置文件等数据写入ZK,所有节点启动时都从ZK拉取配置数据。

方式2.

在第一个节点第一次启动时自动将自己的配置文件写入ZK,其他节点启动时从ZK拉取。

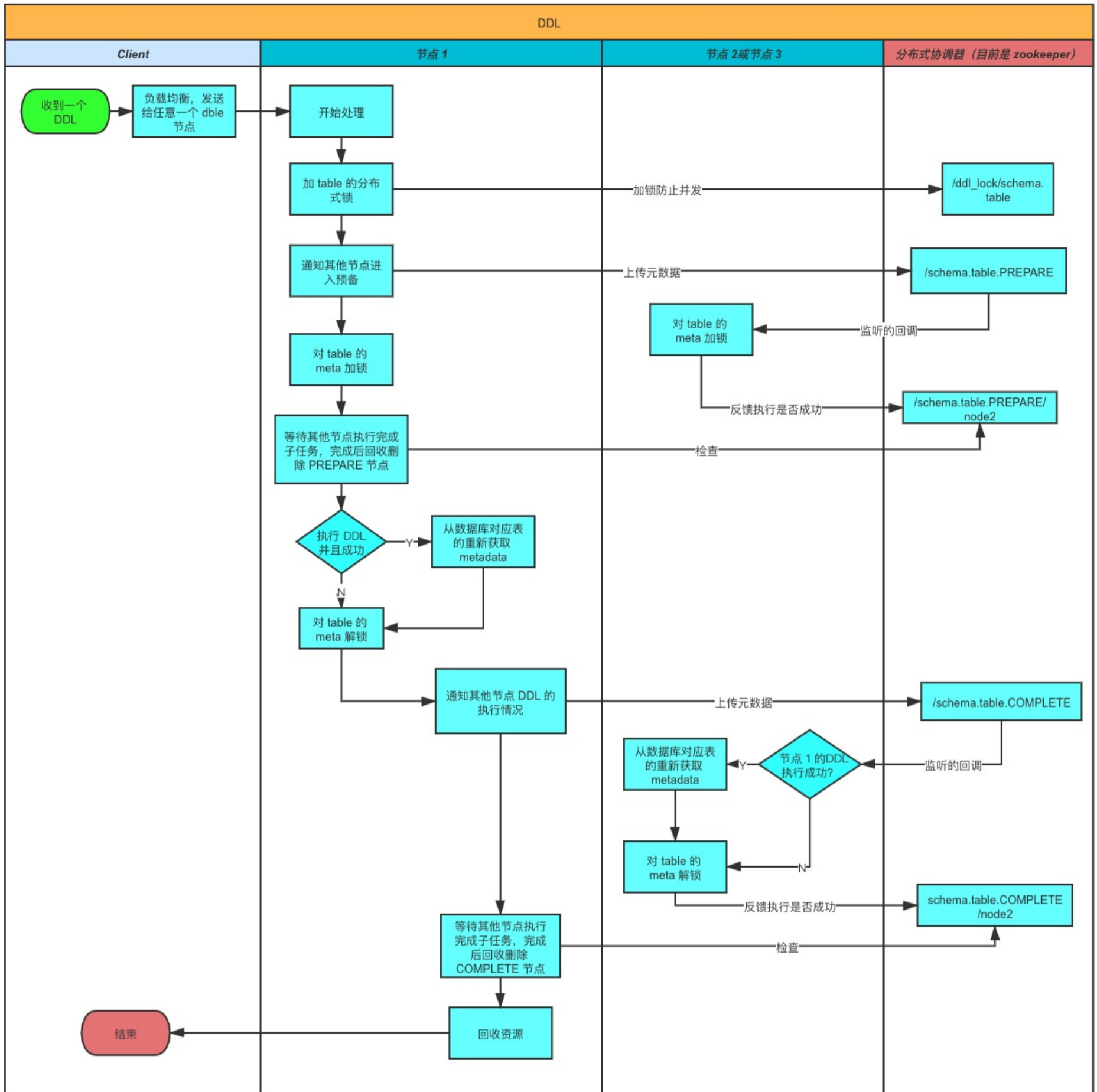
第一个节点的判定：用分布式锁抢占的方式，未抢占到节点会阻塞等待直到获取到分布式锁，如果此时初始化标记已经被设置，则从ZK拉取配置，否则将自己本地配置写入。



2.8.4 状态同步

A.DDL

做DDL时候会在执行的某个节点成功后，将消息推给ZK，ZK负责通知其他结点做变更。流程如下图：

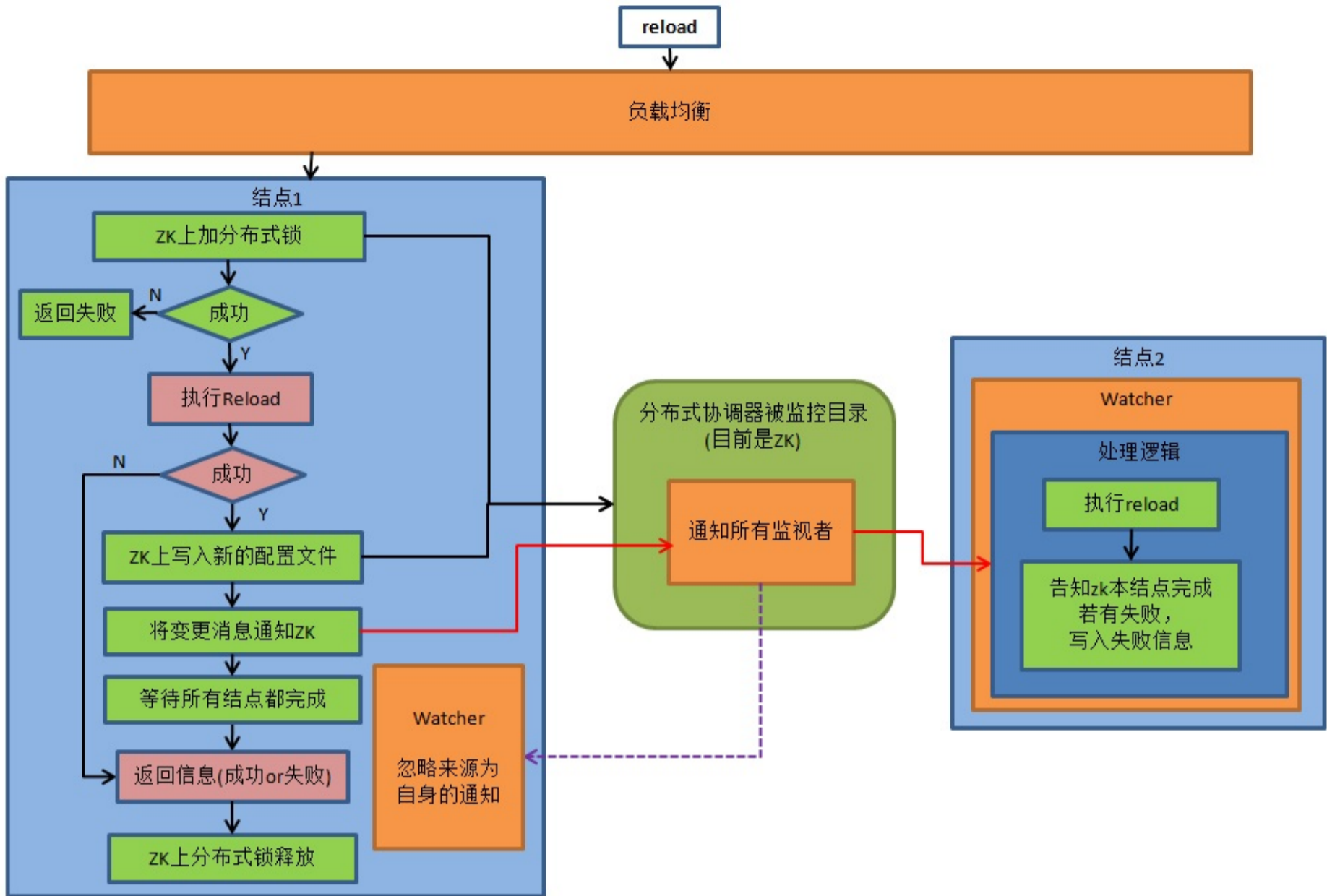


类似于二阶段提交的方式

- 注：
1. 新节点启动时，加载元数据前会检测是否有其他节点在做DDL变更，如果有，则等待。
 否则加分布式锁防止加载元数据期间其他节点做DDL变更，直到元数据加载完释放分布式锁。
 1. "等待其他节点执行完成子任务" 指的是：每个节点完成任务后，都会创建子节点用来告知分布式协调器自己已经完成，执行主任务的节点负责检查所有节点是否都创建了子节点（即检查是否都完成了任务），如果完成则删除父节点（回收资源）。此操作为原子操作。
 1. 发起节点故障：如果执行DDL的节点故障下线，其他节点会侦听到此消息，保证解开对应节点的tablemeta锁，并记录故障告警（如果配置了告警通道），需要运维人工介入修改zk对应ddl节点的状态，检查各个节点meta数据状态，可能需要reload metadata。
 1. 逻辑上不应该有某个监听节点上加载meta失败的情况，如果发生了，告警处理
 (人工介入对应节点的meta是不正确的,需要reload meta)
 1. 注:view目前是异步模式,可能存在某个间隙view修改成功，查询仍旧拿到旧版的view结构。

B.reload @@config/ reload @@config_all

执行流程如下图：



注：如果在部分结点失败，则会返回错误及错误原因以及结点名。设计影响面：db.xml, sharding.xml, userxml, sequence_conf.properties 和 sequence_db_conf.properties

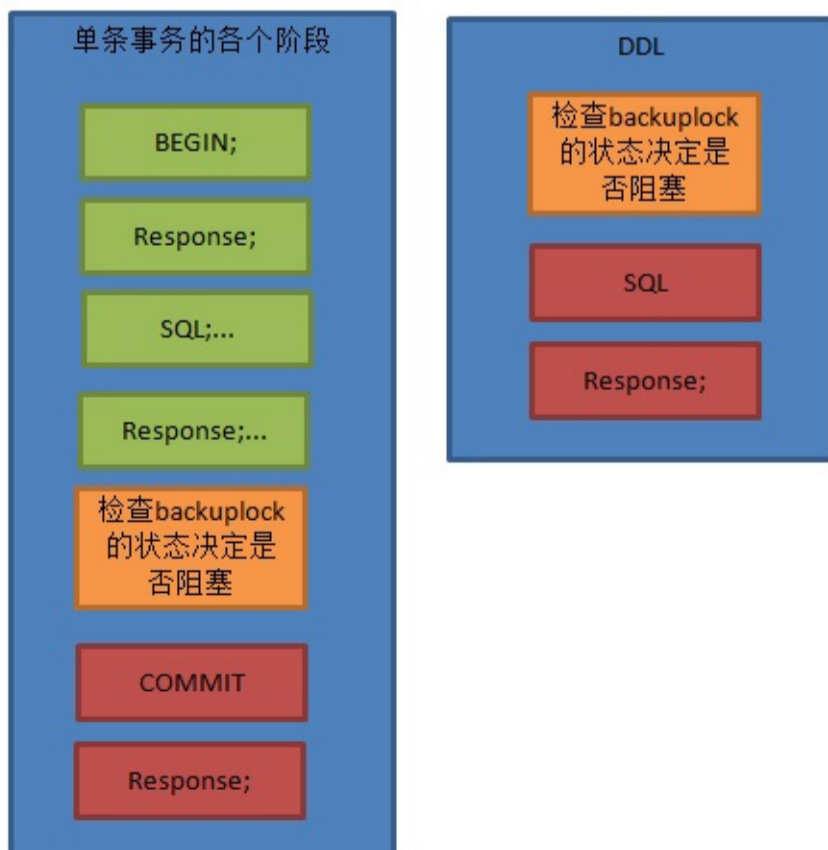
C. 拉取一致性的binlog线

目的：

获得后端数据库实例的一致性binlog位置。由于两阶段提交的第二阶段执行在各结点无法保证时序性和同步性，所以直接下发show master status获取binlog可能会造成不一致。

实现方式：

如下图，当前端收到show @@binlog.status 语句时，遍历当前所有活动session查看状态。若session处于绿色区域，则在进入红色区域前等待知道show @@binlog.status结果返回。若存在session处于红色区域，则需要等待所有红色区域的session返回结果走出红色区域后下发show master status。



超时处理：

此处有可能有死锁发生。

场景:session1 正在更新tableA, 处于绿色区域, session2下发有关于tableA 的DDL, 等待 metaLock解锁, 处于红色区域.session3 下发show @@binlog.status. 此时session1 等待session3, session2等待session1,session3等待session2.

因此引入超时机制。如果session3 等待超过showBinlogStatusTimeout(默认60s, 可配置), 自动放弃等待, 环状锁解除。

集群协调:

- 1.收到请求后同步通知ZK, 先等待本身结点准备工作结束, 之后zk通知其他结点处理。
- 2.所有结点遍历各自的活动的session, 进入红色区域的等待处理完成, 绿色区域的暂停进入红色区域。
- 3.结点将准备好/超时将状态上报给ZK
- 4.主节点等待所有结点状态上报完成之后, 判断是否可以执行任务, 若是,则执行show @@binlog.status并返回结果, 否则报告本次执行失败。
- 5.主节点通过ZK通知各结点继续之前的任务

集群超时处理:

若有结点超时未准备好, 主节点会报超时错误, 并通过ZK通知各结点继续之前的任务。

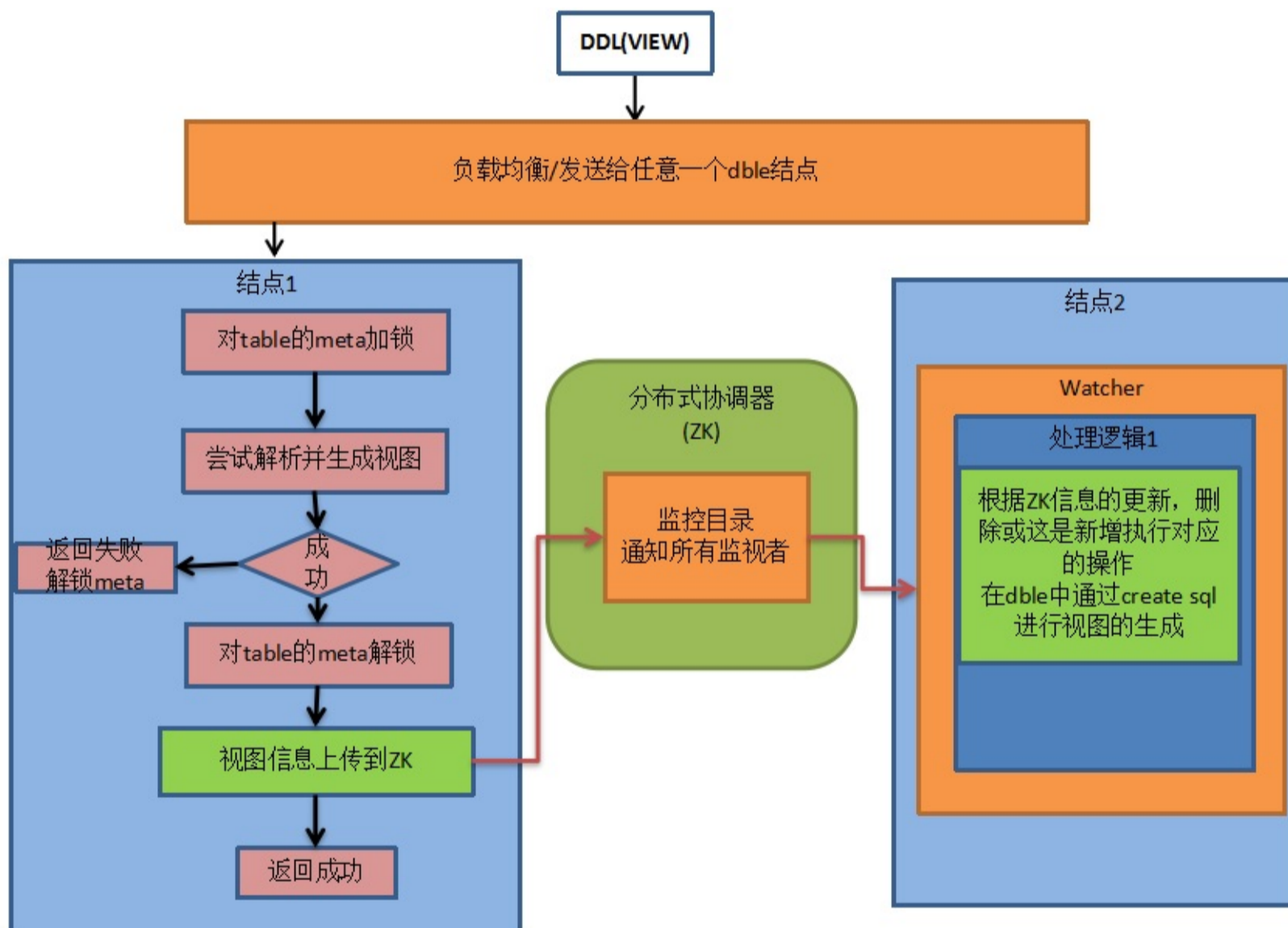
故障处理:

主节点执行过程中故障下线, 其他结点会感知, 保证自己结点一定时间后自动解锁继续原有任务。

ZK状态需要人工干预。人工等待所有结点超时之后, 手动删除/修改zk上的状态信息以便下次执行时不出问题。

D.View管理

在使用集群模式是, 使用zk进行view视图信息的管理, 使得整个dble集群能够进行视图信息的同步 由于视图只用于数据查询且不会造成数据异常的属性, 在视图同步时采用异步同步的方法



注 在zk上的view数据信息如下形式 key schema.table value {"serverId":"create_Server_id", "createSql":"view_create_sql"}

E.online状态

dble启动时候会在online的目录下注册自己的信息, 如果此时正在做DDL, 会阻塞一段时间, 防止表结构元数据不一致。

功能:

- 1.作为集群协调时候检查哪些dble完成了对应任务的标准
- 2.dble故障后同集群的其他实例会发现状态并处理完收尾的状态, 主要是DDL状态残留, 拉取一致性binlog线或者暂停流量的残余。

F.高可用命令同步

当启用needSyncHa时候, 此选项才生效

F.1 disable工作流程:

- 1.申请分布式锁
- 2.本地执行disable
- 3.将信息写入集群
- 4.等待所有结点完成disable
- 5.完成并清理
- 6.释放分布式锁

F.2 enable工作流程：

- 1.申请分布式锁
- 2.本地执行enable
- 3.将信息写入集群
- 4.释放分布式锁
- 5.其他订阅结点异步完成enable

F.3 switch工作流程：

- 1.申请分布式锁
- 2.本地执行switch
- 3.将信息写入集群
- 4.释放分布式锁
- 5.其他订阅结点异步完成switch

G.暂停流量（一般用于扩容迁移等）

G.1 pause流程

1. 申请分布式锁 `pause_node.lock`
2. 通知其他结点
3. 本结点停流量
4. 等待其他结点完成停流量 或者超时
5. 返回暂停成功或者失败
6. 释放分布式锁

G.2 resume流程

1. 申请分布式锁 `pause_node.lock`
2. 本结点恢复流量
3. 通知其他结点
4. 等待其他结点完成恢复流量 或者超时
5. 返回暂停成功或者失败
6. 释放分布式锁

2.8.5 XA日志管理

在使用集群模式时，未完成的XA日志会存放在zookeeper上，更加安全，防止某台机器硬盘物理损坏导致日志丢失（此处可能会有并发高吞吐引发的性能及其他问题，待测试）。

2.8.6 ZK整体目录结构

```

rootPath(配置在cluster.cnf中的)
  clusterId(配置在cluster.cnf中的)
  conf
    initied
    status
    operator
      instanceName(临时的, key为bootstrap.cnf配置内容。功能:reload响应id)
  sharding(sharding.xml的json信息)
  db(db.xml的json信息)
  user(user.xml的json信息)
  migration(用于暂停流量)
    pause
      instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)
    resume
      instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)
  sequences
    instanceid //zk 分布式方式
    incr_sequence//批量步长方式
    table_name
    common(sequence_conf.properties 或者sequence_db_conf.properties)
  binlog_pause
    status (发生时建立, 结束后回收)
    instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)
  lock
    syncMeta.lock(临时的, 启动时防止元数据变更)
    confInit.lock
    confChange.lock
    binlogStatus.lock
    ddl_lock/schema.table
    view_lock/`schema`.`table`
    dbGroup_locks/groupName
    pause_node.lock
  online
    instanceName(临时的, key为bootstrap.cnf配置内容。)
    ddl
      schema.table.PREPARE (ddl操作的第一阶段, 用于加锁)
        instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)
      schema.table2.COMPLETE (ddl操作的第二阶段, 此时 ddl 已执行完成, 开始同步元数据)
        instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)
      schema.table3.PREPARE
  xalog
    node1
    node2
  view
    schema:view
    operator (订阅目录)
      schema.view:(update/delete,注意:create当作update处理)
      instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)

  dbGroups
    dbGroup_status
    groupName
    dbGroup_response
    instanceName(临时的, key为bootstrap.cnf配置内容。功能:响应结点)

```

2.8.7 全局序列

类twitter snowflake 方式, ZK完成的工作是生成每个节点的instanceID。

类offset-step 方式, ZK完成的工作是存储当前的Step值。

2.8.9 移除集群中分布式锁的方式

前置知识

当clusterMode=ucore (使用爱可生商业集群调度中心), 使用ucore集群调度中心的分布式锁是有超时机制的; dble内部为了防止使用期间超时, 内部会给分布式锁分别创建一个线程(renewThread), 专门用来续约对应分布式锁, 防止锁超时。

clusterMode=zk, zk集群调度中心的分布式锁没有超时机制, 则不用创建renew线程。

可检索 续约日志: `renew lock of session success`

背景

在某些不可预测的情况下, 可能会导致分布式锁残留, 导致dble内部的renew线程一直工作, 比如conf reload的场景; 希望能通过不重启的方式实现renew线程自杀, 最终达到分布式锁超时而释放的效果。

查看renew线程列表

在管理端侧，查看dble_cluster_renew_thread表，此表展示当前时间点正在做分布式操作时获取分布式锁对应的renew线程列表，如：

```
mysql> select * from dble_cluster_renew_thread;
+-----+
| renew_thread |
+-----+
| UCORE_RENEW_universe/dble-v3/ushard-1/lock/ddl_lock/testdb.tablea |
+-----+
2 rows in set (0.00 sec)
```

注意：非集群模式下 或者 集群模式不为ucore时，查询的结果为空；

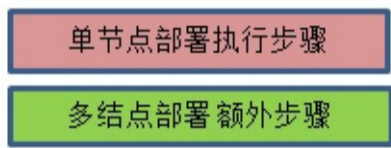
采用kill renew线程

```
mysql> kill @@cluster_renew_thread 'UCORE_RENEW_universe/dble-v3/ushard-1/lock/confChange.lock';
Query OK, 0 rows affected (0.00 sec)
kill cluster renew thread successfully!
```

可检索 kill日志： manual kill cluster renew thread

注意：在实际实现中，并不是在集群调度中心里直接把对应的分布式锁删除，而是通过中断dble内部renew线程，实现不再对分布式锁续约，达到分布式锁最终因超时而自行释放。

2.8.10 附录



图例

2.9 Grpc告警功能

2.9.1 告警功能概述

Dble拥有和商业项目ucore进行告警对接的功能，当dble触发某些重要的报错信息时，会通过ucore提供的grpc接口将对应的告警信息发送到ucore告警中，免去了运维人员在日志文件中的大量搜索，能够直观展示在页面上。

2.9.2 告警配置依赖

依赖cluster.cnf和bootstrap.cnf 的告警的基础信息

名称	内容	默认值	详细作用原理或应用	实例/全局属性
url	grpc告警的url	cluster.cnf 里的clusterIP	在发送grpc的时候作为IP地址使用	实例
port	告警端口	cluster.cnf 里的clusterPort	grpc发送的目的端口	实例
serverId	服务器ID	\$ushard-id(ip1,ip2) ,其中\$ushard-id 是bootstrap.cnf 里的instanceName	接口参数	实例
componentId	组件ID	\$ushard-id 即bootstrap.cnf 里instanceName	接口参数	实例
componentType	组件类型	ushard	接口参数	实例

2.10 meta数据管理

Meta数据的管理包含以下部分：

- [2.10.1 Meta信息初始化](#)
- [2.10.2 Meta信息维护](#)
- [2.10.3 一致性检测](#)
- [2.10.4 View Meta](#)

2.10.1 Meta信息初始化

dble在启动时会逐个同步抓取相关表，视图的信息，并对分区表做一致性检测。

2.10.1.1 表信息的初始化

表信息是从后端数据节点抓取的。dble在启动时会根据schema，table，数据节点的配置（参见1.5 sharding.xml）对逐个数据节点执行：

```
show tables;
```

```
show create table ...
```

获取后端表的实际创建信息并做解析以获取需要的信息。

2.10.1.2 视图信息的初始化

视图的信息是属于Dble中间件的状态信息，是为数不多的Dble自有的状态之一，存在两种形式的初始化

1 非集群模式下通过本地文件进行view的存储和初始化

2 集群状态下通过ZK进行view的存储和初始化

关于其他view meta的实现以及使用细节详见2.10.4 view meta

2.10.1.3 分区表的一致性规则

分区表分布在多个实例上的后端表的表结构不必完全相同，但最好完全形同。多个后端表被认为一致的规则为：

- 有相同的列；
- 有相同的主键；
- 有相同的唯一键；
- 有同样的索引；
- 有相同的键。

只要这五个方面相同就认为后端的表具有一致性，一致性检测通过。

2.10.2 Meta信息维护

dbie在每次执行某些类型ddl语句之后都会更新相关表，视图的元信息。目前，更新元信息的ddl语句类型如下：

- create table语句
- drop table语句
- alter table语句
- truncate table语句
- create index语句
- drop index语句

根据是否配置zookeeper服务器服务(参见1.1 cluster.cnf)，Meta信息的维护逻辑分如下两种情况：

1. 不用zookeeper服务

在此种情况下，由于仅有一个dbie运行实例，本地更新已是全部信息，不必做进一步的维护逻辑。

2. 利用zookeeper服务

此种情况下的更新逻辑为：

- a. 在启动时每一个dbie实例都向zookeeper服务器注册监听事件，监听系统中表元信息的改变。
- b. 当某一个dbie实例更新了某些表的元信息之后，它负责向zookeepr服务器广播更新事件。
- c. 其他dbie实例在监听到更新事件后更新自己维护的元信息。

2.10.3 一致性检测

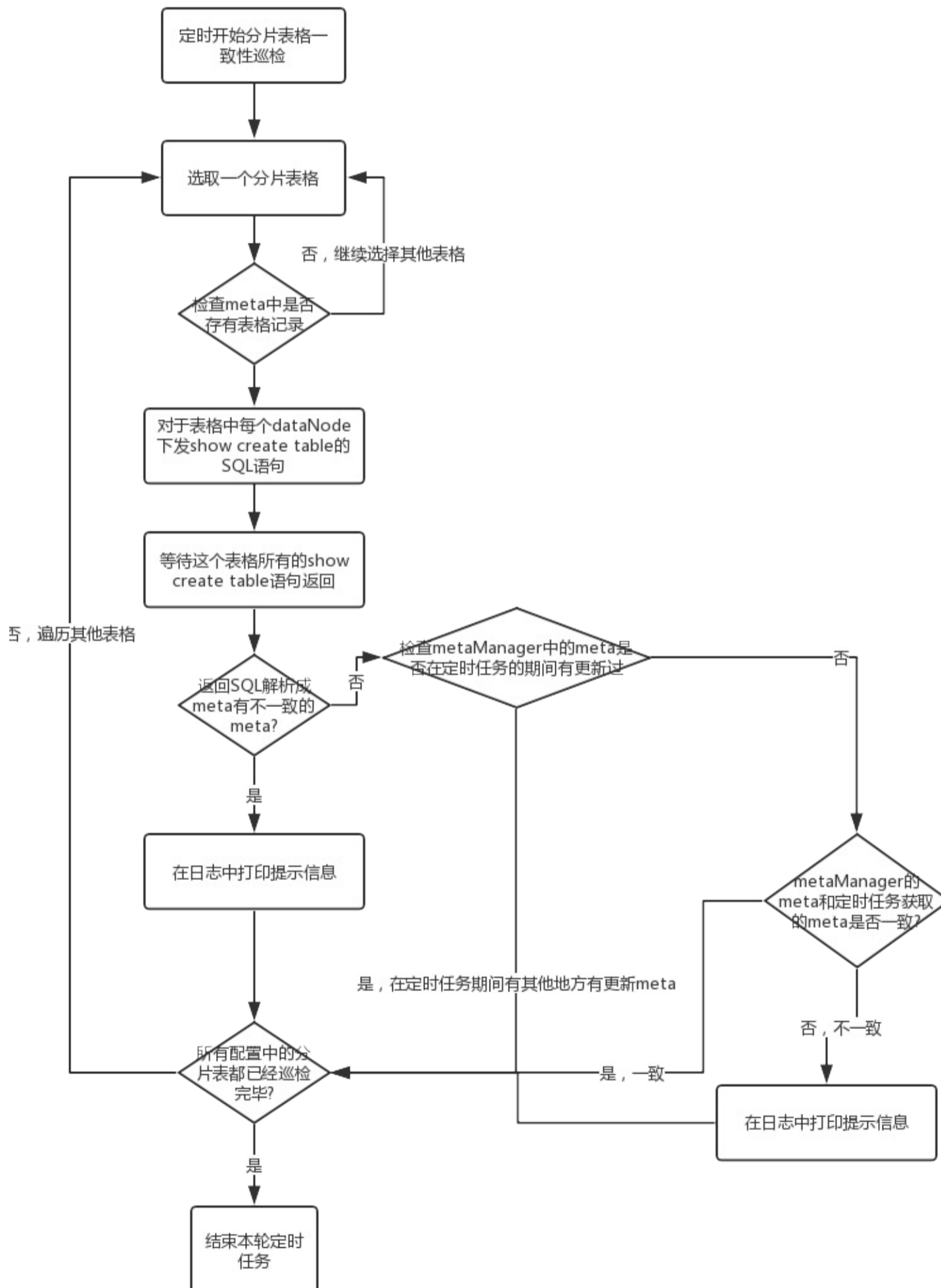
由于Dble对于分片的应用和处理可以看到，对于某张具体的分片表，dble默认在具体数据库节点上的所有表格的实体拥有同样的结构，由于可能在Dble运行过程中的种种问题导致了数据库节点的表格结构不一致的情况，在Dble中有对于各个分片表的表格结构进行定期检查确认的机制。

分片表结构一致性检查的定时任务为tableStructureCheckTask，其周期为checkTableConsistencyPeriod默认值30×60×1000（30分钟），任务开关为checkTableConsistency默认为0（关闭），以上配置均配置在bootstrap.cnf中

大致的表格一致性检查任务逻辑如下：

- 循环配置信息中的schema以及table
- 检查对应数据库和表格的meta信息是否存在
- 对于每个有效的节点下发SQL “show create table”
- 根据收到的建表语句创建新的meta，并通过Set进行去重
- 判断有几个不同的meta，若meta数量超过一个则进行在日志中打印告警信息
- 将新老的meta进行对比，如果发现新老meta之间的meta也不一致则在日志中打印告警信息

整体的流程图如下：



全局表的一致性检查

全局表在Dble的使用中被认为每个节点都存在全量的数据，所以在全局表的检查中不光检查了全局表的表格结构，还在需要另外检查表格的数据一致性,功能开启由全局表具体配置来决定。

2.10.4 view meta

2.10.4.1 view 种类

- mysql中的view
- dble中的view

2.10.4.2 view meta概述

在Dble 2.18.11.0 版本中新增了对view的支持，作为一个中间件支持view采用的方法是和复杂查询类似的逻辑，将VIEW创建的语句中的select部分进行解析，并将解析所得到的解析树进行存储，当有query调用到视图的时候使用解析树进行局部的替代还原成完整的查询SQL，从而达到view实现的效果。

在MySQL中的View meta信息是一种持久化的存储。Dble中的view实现方式也类似，这样Dble中view的实现会给Dble本身带来状态，这个状态需要Dble自身进行保存和维护，所以在view中采取了和XA事务日志相似的方式，并且以视图创建SQL的形式进行以下两种方式的存储：

- 本地文件存储
- ZK存储

在每次发生重启的时候Dble会在初始化meta的时候，通过读取文件或者是ZK中的信息将创建视图的SQL进行重新的解析，最终以解析树的形式保存到meta中去。

在Dble 2.19.10.0版本中新增了对mysql view创建的支持，mysql view会被dble直接下发到后端mysql中执行。不过，创建mysql view有着各种限制，首先view所属的schema必须采取以下配置，即垂直分片的方式：

```
<schema name="schema2" sqlMaxLimit="100" shardingNode="dn5">
</schema>
```

view中涉及的表必须是shardingNode dn5 中的表。对于mysql view，dble则没有持久化，但是依然会有相对应的view meta。mysql view 支持通过dble修改，也支持在后端节点手动修改并通过reload方式加载到dble。

2.10.4.2 view meta保存

本地文件存储

本地文件的存储中，对应的信息以JSON的格式存放在本地文件中，文件的存储路径以及文件名称通过bootstrap.cnf中的viewPersistenceConfBaseDir和viewPersistenceConfBaseName两个参数进行配置（默认存储于./viewConf/viewJson中），具体的文件内容举例如下

```
[{
  "schema": "testdb",
  "list": [{
    "name": "view_test",
    "sql": "create view view_test as select * from a_test"
  }, {
    "name": "vt2",
    "sql": "create or replace view vt2 as select * from suntest"
  }, {
    "name": "suntest",
    "sql": "create view suntest as select * from sbtest"
  }
  ]
}]
```

ZK K/V存储

在集群状态下Dble的各个内部状态需要同步，包括view的创建删除和修改，具体的key值会存储在ZK BASE_PATH/view下，并使用key值schema_name:view_name，在view中使用json的格式进行create sql和修改的serverID的存储，以下给出一个举例：

```
{
  "serverId": "10010",
  "createSql": "create view view_test as select * from a_test"
}
```

此JSON字符串当作value存储在 /.../view/testdb:view_test

2.11 统计管理

统计管理包含以下几个部分

- 2.11.1 查询条件统计
- 2.11.2 表状态统计
- 2.11.3 用户状态统计
- 2.11.4 命令统计
- 2.11.5 heartbeat统计
- 2.11.6 网络读写统计
- 2.11.7 快速sql统计命令

2.11.1 查询条件统计

查询条件统计是在处理查询结果时进行的。统计的是关于某个表中关于某列的条件。
dble系统每次只能进行关于一个表中的一个列的条件统计。
要进行此类统计首先必须开启enableStatisticAnalysis，其次要设置要统计的表和列，详见下节。

2.11.1.1 查询条件统计表列设置

查询条件统计表列的设置命令：

```
reload @@query_cf=table&column;
```

其中table为要统计的目标表的表名，column为目标表中目标列的列名。

如果要清除查询条件统计表列的设置执行命令：

```
reload @@query_cf;
```

2.11.1.2 查看查询条件统计结果

要查看查询条件统计结果执行如下命令：

```
show @@sql.condition;
```

2.11.2 表状态统计

表状态统计是在处理查询结果时进行的。要进行此类统计必须开启enableStatisticAnalysis。

2.11.2.1 统计内容

表状态主要统计如下内容：

- 读操作的次数
- 写操作的次数
- 表关系，即在同一个语句中出现多个表，以第一个表作为操作的主表，其他表是和主表有关系的表。
- 对主表最后一次操作的时刻。

2.11.2.2 统计结果查看

表状态的统计结果可以通过如下命令查看：

```
show @@sql.sum.table;
```

如果要重置表状态统计，执行如下命令：

```
show @@sql.sum.table true;
```


2.11.3 用户状态统计

用户状态统计是在处理查询结果时进行的。要进行此类统计必须开启enableStatisticAnalysis。

2.11.3.1 统计内容

用户状态主要统计如下内容：

- 读操作的次数
- 写操作的次数
- 执行的时间分布情况

2.11.3.2 用户状态统计服务的命令

```
+ show @@sql.sum;  
+ show @@sql.sum.user;
```

2.11.4 命令统计

命令统计是分类统计dble执行的命令计数，在执行各个命令类的命令时进行统计。统计的命令类如下：

1. initDB
2. query
3. stmtPrepare
4. stmtSendLongData
5. stmtReset
6. stmtExecute
7. stmtClose
8. ping
9. kill
10. quit
11. heartbeat
12. other, 除以上命令类命令之外的所有命令。

2.11.4.1 命令统计结果查看

命令统计结果查看执行如下命令：

- show @@command;
- show @@command.count; 具体命令的使用请参看[2.1 管理端命令集](#)

2.11.5 heartbeat统计

heartbeat统计后端mysql实例的heartbeat状态信息，在对后端mysql实例进行heartbeat检测时进行统计。

2.11.5.1 统计内容

heartbeat统计每一次从heartbeat查询发送到查询结果接收之间的时间差，同步状态。

2.11.5.2 统计结果查看

heartbeat统计的查看执行如下命令：

- show @@heartbeat;
- show @@heartbeat.detail where name=xxx; 其中，xxx为dbinstance名字。
- show @@dbinstance.synstatus;
- show @@dbinstance.syndetail where name=xxx;其中，xxx为dbinstance名字。

以上命令的使用请参看看参看[2.1 管理端命令集](#)

2.11.6 网络读写统计

dble的每一个前后端在进行网络通信时对读写的数据量进行统计。

2.11.6.1 统计内容

- 网络读字节数
- 网络写字节数
- 最后一次进行读/写的时刻

2.11.6.2 服务的命令

- show @@connection;
- show @@backend;
- show @@connection.sql;

以上命令的使用请参看[2.1 管理端命令集](#)

2.11.7 快速sql统计命令

sql统计在执行过程中进行的, 要进行此类统计必须开启samplingRate=100。以下命令的数据来源于管理端sql_log表。相关配置见[sql统计](#)

2.11.3.1 统计内容

主要统计如下内容:

1. 执行sql的用户、开始时间、耗时
2. 近期执行频率较高的sql
3. 符合慢sql的查询
4. sql响应的结果集行数超过10000条的 5、结果集响应包的大小超过maxResultSet的

2.11.3.2 sql快速统计服务的命令

```
+ show @@sql;  
+ show @@sql.high;  
+ show @@sql.slow;  
+ show @@sql.large;  
+ show @@sql.resultset;
```

2.11.3.3 统一清理方式

```
truncate sql_log
```

2.12 故障切换

版本2.20.04.0起对切换功能进行了重定义,废除原有的切换功能, 请知悉。

新的MySQL高可用切换分为两类:

一个是单实例部署的dble会内置一个自带的高可用切换的python3脚本, 跟随dble启动和停止,需要设置bootstrap.cnf中system的useOuterHa 参数为false。

另一个是支持第三方的高可用切换接口功能, 支持集群部署。当然, 单实例dble也可以采用这种方式, 需要设置system的useOuterHa 参数为true。

这种方式情况下, 如果没有真的配置第三方高可用切换组件, 则什么也不会发生。 具体请参考[高可用切换接口](#)

2.12.1 前提条件

- 单实例部署的dble, 设置bootstrap.cnf中system的useOuterHa 参数为false。
- 安装好相应的python3环境, 详情参考本章"自定义python脚本"部分

2.12.2 工作方式

dble启动时跟随dble启动python3脚本进程, 脚本会读取db.xml内部的配置, 检查后端数据库结点的状态, 发生异常后会执行切换命令来通知dble

dble提供以下运维命令:

```
show @@custom_mysql_ha
```

用于查看或者监控当前进程是否存活

当python脚本意外终止时, 可以手工运行命令来启动:

```
enable @@custom_mysql_ha
```

当然, 也可以手动关闭该脚本

```
disable @@custom_mysql_ha
```

2.12.2 注意事项

2.12.2.1 此功能目前只支持在linux环境下使用。

2.12.2.2 reload注意事项

做reload @@config 之前需要修改配置文件, 这时候如果不停止python脚本, 可能会读到中间状态的文件, 所以我们建议的标准流程如下:

1. disable @@custom_mysql_ha 关闭切换功能
2. 修改配置文件
3. reload @@config 重新加载配置
4. enable @@custom_mysql_ha 开启切换功能

2.12.2.3 自定义python脚本

custom_mysql_ha.py 脚本在dble安装目录的bin目录下,使用python3编写,您可以根据实际情况自行修改。

为了使它能正常工作, 需要做以下的准备工作:

1、安装Python3,确认python3是否已安装可通过如下命令

```
/usr/local/bin/python3 --version
/usr/local/bin/pip3 --version
```

2、安装mysqlclient及依赖

CentOS 依赖

```
yum install mysql-devel
```

or Ubuntu 依赖

```
apt-get install libmysqlclient-dev
```

然后

```
pip3 install mysqlclient
```

3、six

```
pip3 install six
```

or

```
pip3 install six -i http://pypi.douban.com/simple --trusted-host pypi.douban.com
```

4、coloredlogs

```
pip3 install coloredlogs
```

or

```
pip3 install coloredlogs -i http://pypi.douban.com/simple --trusted-host pypi.douban.com
```

5、rsa

```
pip3 install rsa
```

or

```
pip3 install rsa -i http://pypi.douban.com/simple --trusted-host pypi.douban.com
```

2.13 前后端连接检查

2.13.1 前端连接无响应超时检查

- 根据在bootstrap.cnf中配置的processorCheckPeriod进行定时的前后端连接检查
- 根据配置在bootstrap.cnf中的idleTimeout判断前端连接是否存在无响应时间超限的现象
- 如果发现无响应时间超过限度则关闭连接

2.13.2 后端连接SQL超时检查

- 根据在bootstrap.cnf中配置的processorCheckPeriod进行定时的前后端连接检查
- 根据配置sqlExecuteTimeout检查所有正在执行的后端连接，是否有执行时间超限的情况
- 关闭所有执行时间超过限度的非DDL后端连接

2.14 ER 拆分

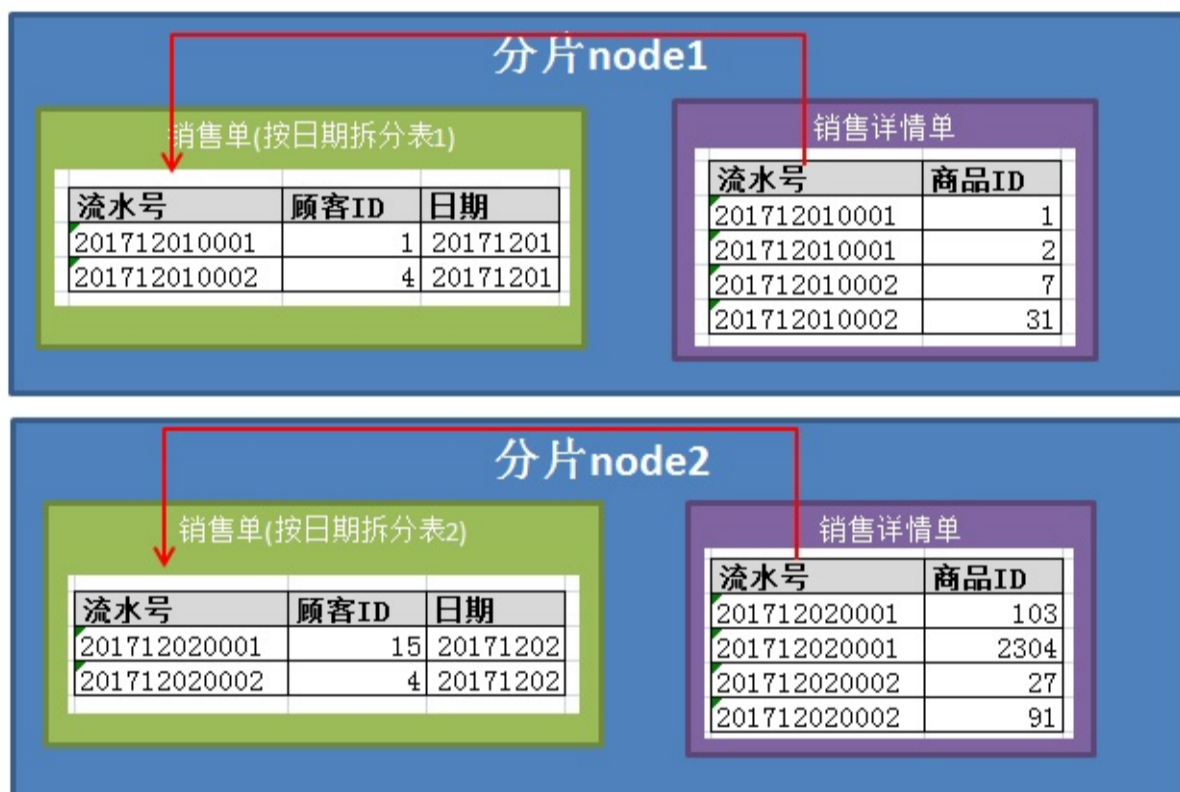
2.14.1 普通ER拆分

当我们需要两个表要进行join的时候,由于数据被分配到不同的节点上,普通的nest loop 方式可能效率会很低。如果我们能把需要join的表按照统一规则划分到相同的区上,就能大概率的解决这一问题。

举个简单的例子如下:



这样两张表具有外键关系,我们可以仿照聚簇的方式按照对应列来进行拆分,这样就可以不跨库实现join了。如下图:



要实现这样的ER功能,需要如下配置。

```
<shardingTable name="sales" shardingNode="dn1,dn2" function="sharding" shardingColumn="id">
  <childTable name="sales_detail" joinColumn="sales_detail_pos_num" parentColumn="sales_pos_num"/>
</table>
```

2.14.2 智能的ER关系

当我们有多个不同的表时,上面的配置方式有点难以使用了。

这种情况下,如果2张或者多张表在dble上的分片规则相同并且具体分片也相同,即使没有配置ER关系,也会当作ER关系来处理。

举例如下配置(片段):

```

<!-- schema片段-->
<shardingTable name="tableA" shardingNode="dn1,dn2" function="hash_function" shardingColumn="id_a" />
<shardingTable name="tableB" shardingNode="dn1,dn2" function="hash_function" shardingColumn="id_b" />
<shardingTable name="tableC" shardingNode="dn2,dn1" function="hash_function" shardingColumn="id_c" />
<shardingTable name="tableD" shardingNode="dn3,dn4" function="hash_function" shardingColumn="id_a" />
<shardingTable name="tableE" shardingNode="dn1,dn2" function="hash_function" shardingColumn="id_a" />
<shardingTable name="tableF" shardingNode="dn1,dn2" function="enum_par" shardingColumn="id_a" />

<!-- rfunction片段-->
<function name="enum_par"
  class="com.actiontech.dble.route.function.PartitionByFileMap">
  <property name="mapFile">partition-hash-int.txt</property>
</function>
<function name="hash_function" class="com.actiontech.dble.route.function.PartitionByLong">
  <property name="partitionCount">2</property>
  <property name="partitionLength">512</property>
</function>

```

最终会得出这样一个映射关系，识别分组会根据数据分布结点和function的唯一性将表分为几组，同一组的才会有ER关系。

table名	拆分列	数据分布结点	function	识别分组
tableA	id_a	dn1,dn2	hash_function	1
tableB	id_b	dn1,dn2	hash_function	1
tableC	id_c	dn2,dn1	hash_function	2
tableD	id_a	dn3,dn4	hash_function	3
tableE	id_a	dn1,dn2	hash_function	1
tableF	id_a	dn1,dn2	enum_par	4

即，ER关系集合为：

```

<tableA.id_a , tableB.id_b, tableE.id_a >

<tableC.id_c>

<tableD.id_a>

<tableF.id_a>

```

PS: 此处略去了schema，实际实现需要标识schema防止重复，ER关系是到列的，如果关联关系不是上述表对应的列，也不会视为ER。

2.15 global 表

在一些业务系统中，存在着类似字典表的表格，它们与业务表之间可能有关系，这种关系，可以理解为“标签”，而不应理解为通常的“主从关系”。这些表具有以下几个特性：

- 变动不频繁
- 数据量总体变化不大
- 数据规模不大，很少有超过数十万条记录。

鉴于此，dble 定义了一种特殊的表，称之为“全局表”，全局表具有以下特性：

- 全局表的插入、更新操作会实时在所有节点上执行，保持各个分片的数据一致性
- 全局表的查询操作，只从一个节点获取
- 全局表可以跟任何一个表进行JOIN 操作

将字典表或者符合字典表特性的一些表定义为全局表，某种程度上部分解决了数据JOIN的难题。

举例如下：



对于数据量不大的字典表（例:超市商品），在多个分片上都有一份同样的副本

相关JOIN语句可以直接下发给各个结点，直接合并结果集就行。

JOIN 例子(伪SQL):

```
SELECT 日期, 商品名, COUNT(*) AS 订单量
FROM 商品表
JOIN 销售详单 USING(商品ID)
WHERE 日期范围(跨结点)
GROUP BY 日期, 商品名。
```

2.16 cache 的使用

cache 的配置请参见1.6节的内容。

2.16.1 路由缓存

路由缓存里键值对是[sql, 路由节点]

配置的值里KEY是pool.SQLRouteCache

VALUE 是逗号隔开的三个值，分别为cachefactory的name,容量，超时时间。

2.16.2 ER子表计算缓存

路由缓存里键值对是[子表对应的joinColumn, 路由节点]

KEY是pool.ER_SQL2PARENTID

VALUE是逗号隔开的三个值，分别为cachefactory的name,容量，超时时间

2.17 执行计划

2.17.1 执行计划的意义

对执行计划进行分析，可以了解中间件和节点是否对SQL语句生成了最优的执行计划，是否有优化的空间，从而为SQL优化提供重要的参考信息。

2.17.2 执行计划的分类

dble的执行计划分为两个层次：dble层的执行计划与节点层的执行计划。

dble层的执行计划: 在SQL语句执行前，dble会根据SQL语句的基本信息，判断该SQL语句应该在哪些节点上执行，将SQL改写成在节点上执行的具体形式，并决定采用何种策略进行数据合并与计算等。

节点层的执行计划: 就是原生的MySQL执行计划。

2.17.3 dble层的执行计划

dble用EXPLAIN指令来查看dble层的执行计划。如例1:

```
explain select * from test;
+-----+-----+-----+
| SHARDING_NODE | TYPE   | SQL/REF          |
+-----+-----+-----+
| dn1           | BASE SQL | SELECT * FROM test LIMIT 100 |
| dn2           | BASE SQL | SELECT * FROM test LIMIT 100 |
+-----+-----+-----+
2 rows in set (0.01 sec)
```

或者例2:

```
mysql> explain select * from test where id =1;
+-----+-----+-----+
| SHARDING_NODE | TYPE   | SQL/REF          |
+-----+-----+-----+
| dn1           | BASE SQL | select * from test where id =1 |
+-----+-----+-----+
1 row in set (0.04 sec)
```

EXPLAIN指令的执行结果包括语句下发的节点，实际下发的SQL语句和数据的合并操作的信息。这些信息是系统静态分析产生的，并没有真正的执行语句。

另外，复杂查询的查询计划也会有所反映，可以通过计划来优化查询语句

如例3:

```
mysql> explain select * from sharding_two_node a inner join sharding_four_node b on a.id =b.id;
+-----+-----+-----+
| SHARDING_NODE | TYPE   | SQL/REF          |
+-----+-----+-----+
| dn1.0         | BASE SQL | select `a`.`id`,`a`.`c_char`,`a`.`ts`,`a`.`si` from `sharding_two_node` `a` ORDER BY `a`.`id` ASC |
| dn2.0         | BASE SQL | select `a`.`id`,`a`.`c_char`,`a`.`ts`,`a`.`si` from `sharding_two_node` `a` ORDER BY `a`.`id` ASC |
| dn1.1         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| dn2.1         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| dn3.0         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| dn4.0         | BASE SQL | select `b`.`id`,`b`.`c_flag`,`b`.`c_decimal` from `sharding_four_node` `b` ORDER BY `b`.`id` ASC |
| merge.1      | MERGE  | dn1.0, dn2.0    |
| merge.2      | MERGE  | dn1.1, dn2.1, dn3.0, dn4.0 |
| join.1       | JOIN   | merge.1, merge.2 |
+-----+-----+-----+
9 rows in set (0.00 sec)
```

再举例4:

```
mysql> explain select id from single union all select b.si from sharding_four_node a inner join sharding_two_node b on a.id =b.id
+-----+-----+-----+
| SHARDING_NODE | TYPE      | SQL/REF                                     |
+-----+-----+-----+
| dn1.0         | BASE SQL  | select `single`.`id` from `single`         |
| dn1.1         | BASE SQL  | select `a`.`id` from `sharding_four_node` `a` ORDER BY `a`.`id` ASC |
| dn2.0         | BASE SQL  | select `a`.`id` from `sharding_four_node` `a` ORDER BY `a`.`id` ASC |
| dn3.0         | BASE SQL  | select `a`.`id` from `sharding_four_node` `a` ORDER BY `a`.`id` ASC |
| dn4.0         | BASE SQL  | select `a`.`id` from `sharding_four_node` `a` ORDER BY `a`.`id` ASC |
| dn1.2         | BASE SQL  | select `b`.`si`,`b`.`id` from `sharding_two_node` `b` ORDER BY `b`.`id` ASC |
| dn2.1         | BASE SQL  | select `b`.`si`,`b`.`id` from `sharding_two_node` `b` ORDER BY `b`.`id` ASC |
| merge.2       | MERGE     | dn1.1, dn2.0, dn3.0, dn4.0               |
| merge.3       | MERGE     | dn1.2, dn2.1                             |
| join.1        | JOIN      | merge.2, merge.3                         |
| merge.1       | MERGE     | dn1.0                                     |
| union_all.1   | UNION_ALL | join.1, merge.1                          |
+-----+-----+-----+
12 rows in set (0.01 sec)
```

我们看到，查询计划分为3列，分别是SHARDING_NODE,TYPE和SQL/REF。

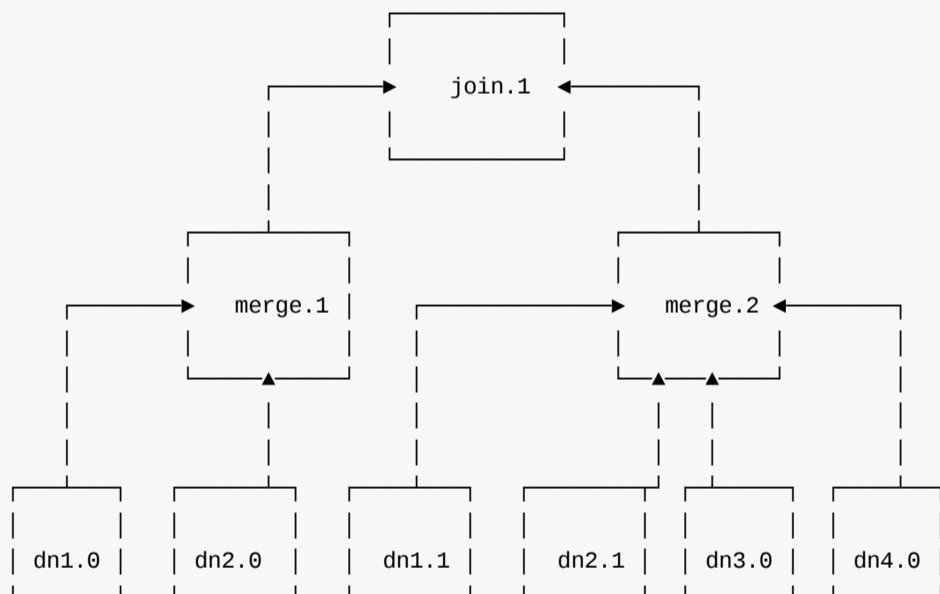
2.17.3.1 简单查询计划的解读

首先，我们要建立一个概念，查询计划结果分两类，一类是简单查询，如例1例2，中间件只做路由计算，所以查询计划比较简单。这种场景下，

- **SHARDING_NODE**列的内容对应于sharding.xml中的shardingNode的name属性，表明sql将被发往哪里。
- **TYPE**列恒为 `BASE SQL`，表示这是个基本查询
- **SQL/REF**列实际退化为SQL，标识了实际下发的SQL内容，看例1的无条件广播实际就被改写，增加了limit 100（当然这个功能也可以设置或者关闭）。

2.17.3.2 复杂查询计划的解读

另一类是复杂查询，这类查询其实需要dble进行计算，比如例3中的跨库join，实际是构建一个类似这样的查询树。



查询树更多细节请参考[公开课](#)相关章节。所以，理解了查询树，就能理解查询计划了。

- **SHARDING_NODE**列的内容分两类，一类对应于sharding.xml中的shardingNode的name属性，表明sql将被发往哪里，对应查询树的叶子结点，和简单查询类似，另一类是查询树的非叶子结点，代表了在dble中需要计算的算子的名称。由于同一个结点可能有多次查询下发。同名算子也可能有多个，所以用点号 and 自增序列区别。
- **TYPE**列：叶子结点为 `BASE SQL`，表示这是下发的基本查询，非叶子结点表示是算子的类型：算子包括：
 - MERGE：合并
 - MERGE_AND_ORDER：带排序的合并（归并排序）
 - AGGREGATE：聚合
 - DISTINCT：去重
 - LIMIT：取前n行
 - WHERE_FILTER：where过滤
 - HAVING_FILTER：having过滤
 - SHUFFLE_FIELD：列名&数据类型整理
 - UNION_ALL：union all操作，如果原sql是union，会裂为UNION_ALL和DISTINCT两个算子
 - ORDER：排序
 - NOT_IN：not in 处理。
 - JOIN：join操作
 - DIRECT_GROUP：直接group by。
 - NEST_LOOP：NEST_LOOP 类型的join，结果返回处理
 - IN_SUB_QUERY：in子查询结果返回处理。

- ALL_ANY_SUB_QUERY: all/any子查询结果返回处理。
- SCALAR_SUB_QUERY: 标量子查询结果返回处理
- RENAME_DERIVED_SUB_QUERY: DERIVED子查询结果处理。
- INNER_FUNC_ADD: sql包含一些dble特殊处理的函数时的处理逻辑,例如LAST_INSERT_ID
- INNER_FUNC_MERGE: sql的select仅包含dble特殊处理的函数时的处理逻辑,例如LAST_INSERT_ID。
- for CHILD in UPDATE_SUB_QUERY.RESULTS: update多表场景中,遍历先下发的select语句的结果集,填充到update单表语句后下发。
- MERGE UPDATE: update多表场景中,回收整理循环下发的结果
- SQL/REF列: 叶子结点退化为SQL,标识了实际下发的SQL内容;非叶子结点表达了查询树中子结点的名称,对应SHARDING_NODE列的名称。通过这样,可以通过查询计划画出一颗完整的查询树来,也就能了解分布式查询计划的细节了。

2.17.4 节点层的执行计划

通过EXPLAIN2命令可查看指定节点上的执行计划。如:

```
mysql> explain2 shardingnode=dn1 sql=select * from test where id =1;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | test | NULL | ALL | NULL | NULL | NULL | NULL | 1 | 100.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

explain2会将sql语句加上explain下发到指定的shardingnode执行,并把节点上explain的结果返回客户端,可以用于观察单个节点的sql执行计划。

2.18 性能观测以及调试概览

- Btrace脚本性能观察(观察查询过程中每个不同阶段的耗时)
- manager命令show @@thread_used观察(观察不同线程的负载情况)

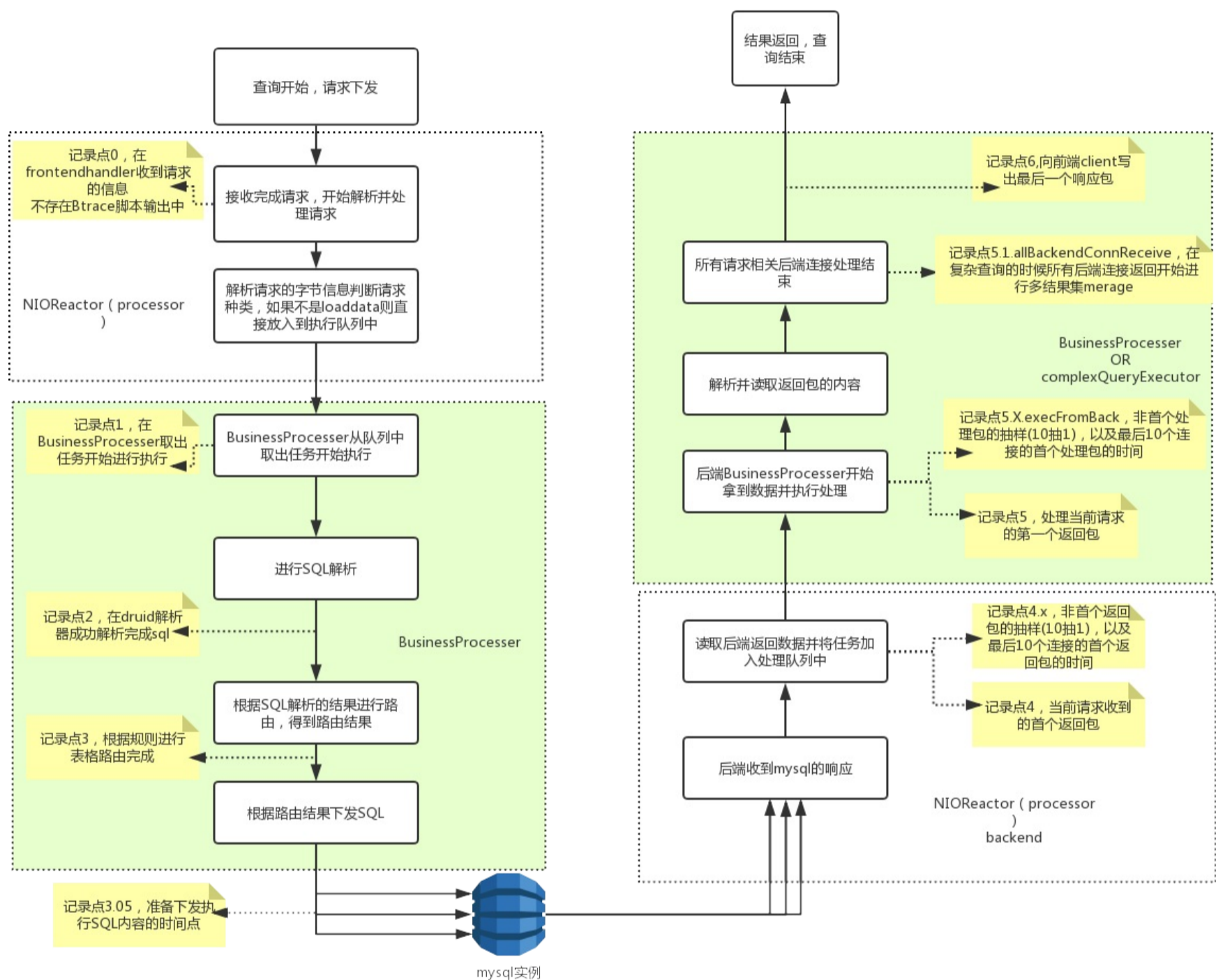
2.18.1 Btrace脚本观察

2.18.1.1 观察

Dble源码中自带观测脚本,BTraceCostTime.java文件是专用的Btrace观测脚本,当前脚本适用Btrace v.1.3

Btrace 相关资料<https://github.com/btraceio/btrace>

开启性能观察的统计需要配置bootstrap.cnf中的两个参数useCostTimeStat和costSamplePercent,分别是启用耗时监控的标志位以及耗时监控的采样百分比,useCostTimeStat = 1的状态下耗时监控被开启,并且以costSamplePercent的概率进行统计,默认的采样率是1%,采样率过低有可能导致btrace脚本输出没有统计结果的情况,采样率过高则会影响到性能本身。脚本中包含几个统计点如下图所示:



通过执行btrace监控正在运行的dble,可以获得类似以下的结果图:

```
profiling:
Block                Invocations  SelfTime.Total  SelfTime.Avg  SelfTime.Min  SelfTime.Max  WallTime.Total  WallTime.Avg  WallTime.Min  WallTime.Max
request->1.startProcess  9073        -638142734     -70334        -1051058      493260        202952071       22368         10565         493260
request->2.endParse      9073        234134936     25805         13206         523393       437087007       48174         23771         1016653
request->3.endRoute     9073        404389553     44570         20123         121474       841476560       92745         43894         1075553
request->4.resFromBack  9073         592398         65            -649019       1602901      4805691043      529669        261612        1602901
request->5.startExecuteBackend 9073       -56808823     -6261         -1749483     2020297      5047581273     556329        350530        2020297
request->6.response     9073        59150286      6519         3045         366620       5167315454     562913        353575        2386917
```

以上输出值有效信息为Block,Invocations,WallTime.,而SelfTime.由于btrace的原因 不准确,可以忽略。

其中通过记录一个查询在每个记录点的时间点来观察是否存在某个中间步骤耗时过长,并进行针对性的优化和调整

2.18.1.2 节点描述

- **0**: 逻辑时间0,来自客户端的请求首次被dble接收到的时间点
- **1.startProcess**: 开始处理前端请求的时间节点
- **2.endParse**: SQL被解析完成的时间节点
- **3.endRoute**: SQL被路由完成的时间节点
- **3.05 readyToDeliver**: SQL准备开始下发的时间点

- **4.resFromBack** : 前端请求首次由任意个后端连接返回信息
- **4.X.resFromBack** : 前端请求后端连接的首个返回包的采样时间点,例: 4.3.resFromBack指的是第三个后端连接首次返回包给dble的时间点, 具体的后端连接的编号为首次返回的顺序, 即首个有网络包返回的后端连接编号为1
- **5.startExecuteBackend** : 前端请求后端连接首个包进入处理阶段的时间点
- **5.X.startExecuteBackend** : 前端请求后端连接的首个包被处理的采样时间点, 例: 5.3.startExecuteBackend指的是第三个后端连接首次进入到后端数据处理的阶段, 具体的编号为首次进入处理阶段的顺序
- **5.1.allBackendConnReceive** : 在复杂查询的情况下开始merage的时间点
- **6.response** : 给前端开始最终返回包的时间点

注意:Btrace关于性能跟踪的脚本设计专为单一SQL模式下使用, 在多个SQL混合查询的情况下, 由于每个SQL涉及的采样点可能不同, 会出现数据上的异常, 甚至是节点顺序和时间点倒挂的现象

2.18.1.3 调整策略

- 1-0的耗时增长: 需增大bootstrap.cnf中frontWorker的值 (其中0比表示dble开始读到某条语句时的时间, 即初始值: 0, 所以1-0的时间就是request->1.startProcess的输出值, 不需要进行计算)
- 4-3的耗时增长: 需增大bootstrap.cnf中NIOBackendRW的值
- 5-4的耗时增长: 需增大bootstrap.cnf中backendWorker的值

2.18.2 Manager命令观察

2.18.2.1 观察

Dble 在18.02.0版本中新添加了manager端口的性能观测命令, 可以通过命令查看各个线程的负载情况, 需要配合在配置文件bootstrap.cnf中的新增参数useThreadUsageStat进行使用

使用命令show @@thread_used会返回各个dble中关键线程最近时间的负载情况, 如下示例

```
mysql> show @@thread_used;
+-----+-----+-----+-----+
| THREAD_NAME | LAST_QUARTER_MIN | LAST_MINUTE | LAST_FIVE_MINUTE |
+-----+-----+-----+-----+
| backendBusinessExecutor2 | 0% | 0% | 0% |
| backendBusinessExecutor1 | 0% | 0% | 0% |
| backendBusinessExecutor0 | 0% | 0% | 0% |
| BusinessExecutor3 | 0% | 0% | 0% |
| $_NIO_REACTOR_BACKEND-2 | 0% | 0% | 0% |
| BusinessExecutor1 | 0% | 0% | 0% |
| $_NIO_REACTOR_BACKEND-3 | 0% | 0% | 0% |
| BusinessExecutor2 | 12% | 3% | 3% |
| $_NIO_REACTOR_BACKEND-0 | 0% | 0% | 0% |
| $_NIO_REACTOR_FRONT-0 | 0% | 0% | 0% |
| $_NIO_REACTOR_BACKEND-1 | 0% | 0% | 0% |
| BusinessExecutor0 | 0% | 0% | 0% |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)
```

- **BussinessExecutorX**
面向前端的业务处理线程, 主要处理前端请求的解析, sql解析路由, 下发查询到mysql实例等
- **backendWorkerX**
面向后端的业务处理线程, 主要处理后端mysql查询结果的返回解析, 结果聚合, 并发回结果到client
- **\$_NIO_REACTOR_FRONT_X**
前端请求接受线程, 负责请求的接收和读取, 之后把数据处理交给BussinessExecutor进行
- **\$_NIO_REACTOR_BACKEND_X**
后端请求接收线程, 负责mysql返回信息的接收和读取, 之后把数据交给backendWorker处理

2.18.2.2 调整策略

当输出的统计结果中, 有一个或者多个类型的线程使用率过高 (经验值为超过80%), 可以适当调整对应的处理线程的数量

- **NIOFrontRW** 参数控制前端\$_NIO_REACTOR_FRONT_X的数量
- **NIOBackendRW** 参数控制后端\$_NIO_REACTOR_BACKEND_X的数量
- **backendWorker** 参数控制backendWorkerX数量
- **frontWorker** 参数控制BussinessExecutorX数量

2.19 智能计算reload

我们对reload @@config_all 做了重构，增加了对dbGroup/dbInstance的变化计算，使得reload行为对整个系统的影响变到最小。

变更名称的dbGroup/dbInstance，等效于删除旧的dbGroup/dbInstance，新增新的dbGroup/dbInstance

2.19.1 默认reload @@config_all

连接池行为描述:

2.19.1.1 不变的dbGroup

此种情况下,dbGroup连接不发生变化,如果关联的schmea发生变更,在需要使用时候进行连接新建或者偷取并同步上下文的方式进行更新。

2.19.1.2 新增的dbGroup

建立新的连接池供使用

2.19.1.3 删除的dbGroup

遍历当前连接池,如果没有事务正在使用连接,则回收,否则放回后端待回收连接池(在show @@backend中可以看到放入回收池的时间),等事务结束时候连接被关闭

待回收的连接可以通过 [2.0.1.32 recycling_resource](#) 查询到

2.19.1.4 新增的dbInstance

不影响正在使用的连接,新增之后建立新的连接池供使用

2.19.1.5 变更的dbInstance

以往空闲的连接会直接关闭,正在使用中的连接会延迟关闭(可以通过 [2.0.1.32 recycling_resource](#) 查询到),也会根据变更后的配置建立新的连接池供使用

2.19.1.6 删除的dbInstance

以往空闲的连接会直接关闭,正在使用中的连接会延迟关闭(可以通过 [2.0.1.32 recycling_resource](#) 查询到)

2.19.2 reload @@config_all -f

强制回收所有正在使用的链接

连接池行为描述:

2.19.2.1 不变的dbGroup/dbInstance

此种情况下,dbGroup/dbInstance连接不发生变化,正在使用的连接会被回收

2.19.2.2 新增的dbGroup/dbInstance

建立新的连接池供使用

2.19.2.3 删除的dbGroup/dbInstance

遍历如果当前连接池,如果没有事务正在使用连接,则回收,否则关闭对应的前端连接以及相关的后端连接

2.19.3 reload @@config_all -r

不计算dbGroup/dbInstance的变化,相当于原本所有的连接池会被删除,然后新建所有的连接池

遍历旧连接池,如果没有事务正在使用连接,则回收,否则放回后端待回收连接池(在show @@backend中可以看到放入回收池的时间),等事务结束时候连接被关闭

待回收的连接可以通过 [2.0.1.32 recycling_resource](#) 查询到

2.19.4 reload @@config_all -s

跳过dbGroup连接检查(与节点建立连接,连接成功,检测通过)

2.20 慢查询日志

类似于MySQL的慢查询日志，可以全局开启并设置，记录dble server运行过程当中的慢查询日志，日志格式兼容MySQL慢查询分析工具(已测试过MySQL官方工具mysqldumpslow和Percona的pt-query-digest)

此外，开启了慢查询日志工具之后，也可以查询某个连接的当前SQL的执行状态，相关命令为：[show @@connection.sql.status where FRONT_ID= ?;](#)

2.20.1 在bootstrap.cnf里增加了6个参数，用于启动时候控制慢查询日志的行为

```
<!-- 是否开启慢查询日志 -->
-DenableSlowLog=1
<!-- 慢查询日志保存文件目录 -->
-DslowLogBaseDir=./slowlogs
<!-- 慢查询日志保存文件前缀名称 -->
-DslowLogBaseName=slow-query
<!-- 日志两次刷盘之间的最大周期，单位是秒 -->
-DflushSlowLogPeriod=1
<!-- 日志两次刷盘之间内存中的最大条数阈值 -->
-DflushSlowLogSize=1000
<!-- 慢查询统计阈值，大于此值会被认为是慢查询，单位是毫秒 -->
-DsqlSlowTime=100
```

2.20.2 管理端口增加命令，用于运行过程中动态修改慢查询日志统计行为

```
enable @@slow_query_log; -- 开启慢查询日志
show @@slow_query_log; -- 查询慢查询日志的开启状态
disable @@slow_query_log; -- 关闭慢查询日志
show @@slow_query_log; -- 再次查询慢查询日志的开启状态

show @@slow_query.time; -- 查看慢查询日志统计阈值
reload @@slow_query.time=200; -- 修改慢查询日志统计阈值

show @@slow_query.flushperiod; -- 查看慢查询日志刷盘周期
reload @@slow_query.flushperiod=2; -- 修改慢查询日志刷盘周期

show @@slow_query.flushsize; -- 查看慢查询日志刷盘条数阈值
reload @@slow_query.flushsize=1100; -- 修改慢查询日志刷盘条数阈值
```

2.20.3 支持慢查询日志分析工具：MySQL的mysqldumpslow工具和Percona的pt-query-digest工具

慢查询日志大概是这样的：

```
/FAKE_PATH/mysqlld, Version: FAKE_VERSION. started with:
Tcp port: 3320 Unix socket: FAKE_SOCKET
Time          Id Command      Argument
# Time: 2018-08-23T17:40:10.149000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.132709 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000350 Prepare_Push: 0.116678 dn1_First_Result_F
etch: 0.013686 dn1_Last_Result_Fetch: 0.001422 Write_Client: 0.001995
SET timestamp=1535017210149;
select * from sharding_two_node where id =1;
# Time: 2018-08-23T17:40:10.200000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.035600 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000062 Prepare_Push: 0.006733 dn2_First_Result_F
etch: 0.012524 dn1_First_Result_Fetch: 0.010971 dn2_Last_Result_Fetch: 0.015368 dn1_Last_Result_Fetch: 0.005119 Write_Client: 0.017834
SET timestamp=1535017210200;
select * from sharding_two_node;
# Time: 2018-08-23T17:40:10.282000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.045337 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000166 Prepare_Push: 0.003941 dn1_First_Result_F
etch: 0.039652 dn1_Last_Result_Fetch: 0.000300 Write_Client: 0.001578
SET timestamp=1535017210282;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:10.315000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.031232 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.005467 Prepare_Push: 0.001989 dn2_First_Result_F
etch: 0.020240 dn2_Last_Result_Fetch: 0.001900 Write_Client: 0.003536
SET timestamp=1535017210315;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:10.432000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.116672 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.013625 Prepare_Push: 0.024767 dn2_First_Result_F
etch: 0.056395 dn1_First_Result_Fetch: 0.026420 dn2_Last_Result_Fetch: 0.000743 dn1_Last_Result_Fetch: 0.001700 Write_Client: 0.051861
```

```
SET timestamp=1535017210432;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:10.772000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.338569 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000082 Prepare_Push: 0.258365 dn1_0_First_Result_Fetch: 0.047494 dn1_0_Last_Result_Fetch: 0.029018 dn2_0_First_Result_Fetch: 0.042964 dn2_0_Last_Result_Fetch: 0.033525 Write_Client: 0.009385
SET timestamp=1535017210772;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:10.821000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.046745 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000059 Prepare_Push: 0.025401 dn1_0_First_Result_Fetch: 0.011755 dn1_0_Last_Result_Fetch: 0.001180 Generate_New_Query: 0.001706 dn1_1_First_Result_Fetch: 0.004224 dn1_1_Last_Result_Fetch: 0.001213 Write_Client: 0.001384
SET timestamp=1535017210821;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:12.061000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.036952 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001111 Prepare_Push: 0.001132 dn1_First_Result_Fetch: 0.034266 dn1_Last_Result_Fetch: 0.000084 Write_Client: 0.000443
SET timestamp=1535017212061;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:12.091000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.028213 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000666 Prepare_Push: 0.001206 dn2_First_Result_Fetch: 0.025991 dn2_Last_Result_Fetch: 0.000101 Write_Client: 0.000349
SET timestamp=1535017212091;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:12.132000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.040365 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000223 Prepare_Push: 0.001172 dn2_First_Result_Fetch: 0.019852 dn1_First_Result_Fetch: 0.019810 dn2_Last_Result_Fetch: 0.000901 dn1_Last_Result_Fetch: 0.000780 Write_Client: 0.019160
SET timestamp=1535017212132;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:12.145000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.012196 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000115 Prepare_Push: 0.001403 dn1_0_First_Result_Fetch: 0.006714 dn1_0_Last_Result_Fetch: 0.002561 dn2_0_First_Result_Fetch: 0.006787 dn2_0_Last_Result_Fetch: 0.001806 Write_Client: 0.002280
SET timestamp=1535017212145;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:12.164000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.016979 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000224 Prepare_Push: 0.002236 dn1_0_First_Result_Fetch: 0.006678 dn1_0_Last_Result_Fetch: 0.000703 Generate_New_Query: 0.000866 dn1_1_First_Result_Fetch: 0.004532 dn1_1_Last_Result_Fetch: 0.000879 Write_Client: 0.001002
SET timestamp=1535017212164;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:13.134000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.010213 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000145 Prepare_Push: 0.001520 dn1_First_Result_Fetch: 0.007996 dn1_Last_Result_Fetch: 0.000201 Write_Client: 0.000551
SET timestamp=1535017213134;
select * from sharding_two_node where id =1;
# Time: 2018-08-23T17:40:13.153000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.014257 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000080 Prepare_Push: 0.002394 dn2_First_Result_Fetch: 0.008839 dn1_First_Result_Fetch: 0.008837 dn2_Last_Result_Fetch: 0.001424 dn1_Last_Result_Fetch: 0.002407 Write_Client: 0.002945
SET timestamp=1535017213153;
select * from sharding_two_node;
# Time: 2018-08-23T17:40:13.212000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.029822 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000063 Prepare_Push: 0.001128 dn1_First_Result_Fetch: 0.028277 dn1_Last_Result_Fetch: 0.000109 Write_Client: 0.000355
SET timestamp=1535017213212;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:13.240000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.027695 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000067 Prepare_Push: 0.000682 dn2_First_Result_Fetch: 0.026582 dn2_Last_Result_Fetch: 0.000078 Write_Client: 0.000364
SET timestamp=1535017213240;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:13.321000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.076093 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000231 Prepare_Push: 0.001334 dn2_First_Result_Fetch: 0.035072 dn1_First_Result_Fetch: 0.035074 dn2_Last_Result_Fetch: 0.018756 dn1_Last_Result_Fetch: 0.001263 Write_Client: 0.039457
SET timestamp=1535017213321;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
```

```
# Time: 2018-08-23T17:40:13.348000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.026278 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000335 Prepare_Push: 0.001249 dn1_0_First_Result_Fetch: 0.011028 dn1_0_Last_Result_Fetch: 0.009279 dn2_0_First_Result_Fetch: 0.019200 dn2_0_Last_Result_Fetch: 0.003441 Write_Client: 0.004600
SET timestamp=1535017213348;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:13.381000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.029152 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000662 Prepare_Push: 0.003189 dn1_0_First_Result_Fetch: 0.014453 dn1_0_Last_Result_Fetch: 0.001013 Generate_New_Query: 0.000911 dn1_1_First_Result_Fetch: 0.005703 dn1_1_Last_Result_Fetch: 0.001483 Write_Client: 0.002114
SET timestamp=1535017213381;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:14.163000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.012540 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000197 Prepare_Push: 0.001303 dn2_First_Result_Fetch: 0.006452 dn1_First_Result_Fetch: 0.007858 dn2_Last_Result_Fetch: 0.004065 dn1_Last_Result_Fetch: 0.002960 Write_Client: 0.004588
SET timestamp=1535017214163;
select * from sharding_two_node;
# Time: 2018-08-23T17:40:14.220000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.027587 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000051 Prepare_Push: 0.000744 dn1_First_Result_Fetch: 0.026441 dn1_Last_Result_Fetch: 0.000104 Write_Client: 0.000350
SET timestamp=1535017214220;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:14.253000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.031984 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000070 Prepare_Push: 0.001144 dn2_First_Result_Fetch: 0.030202 dn2_Last_Result_Fetch: 0.000182 Write_Client: 0.000568
SET timestamp=1535017214253;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:14.292000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.037327 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000272 Prepare_Push: 0.001316 dn2_First_Result_Fetch: 0.014299 dn1_First_Result_Fetch: 0.014331 dn2_Last_Result_Fetch: 0.001148 dn1_Last_Result_Fetch: 0.000753 Write_Client: 0.021440
SET timestamp=1535017214292;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:14.303000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.010244 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000050 Prepare_Push: 0.001101 dn1_0_First_Result_Fetch: 0.004540 dn1_0_Last_Result_Fetch: 0.002781 dn2_0_First_Result_Fetch: 0.004708 dn2_0_Last_Result_Fetch: 0.002592 Write_Client: 0.002092
SET timestamp=1535017214303;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:14.327000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.021078 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000109 Prepare_Push: 0.002098 dn1_0_First_Result_Fetch: 0.006720 dn1_0_Last_Result_Fetch: 0.000748 Generate_New_Query: 0.001158 dn1_1_First_Result_Fetch: 0.008043 dn1_1_Last_Result_Fetch: 0.001147 Write_Client: 0.001269
SET timestamp=1535017214327;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:15.254000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.010569 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000076 Prepare_Push: 0.001050 dn1_First_Result_Fetch: 0.008330 dn1_Last_Result_Fetch: 0.000146 Write_Client: 0.001113
SET timestamp=1535017215254;
select * from sharding_two_node where id =1;
# Time: 2018-08-23T17:40:15.321000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.024216 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000081 Prepare_Push: 0.001295 dn1_First_Result_Fetch: 0.021938 dn1_Last_Result_Fetch: 0.000422 Write_Client: 0.000902
SET timestamp=1535017215321;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:15.351000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.027796 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000666 Prepare_Push: 0.000760 dn2_First_Result_Fetch: 0.025984 dn2_Last_Result_Fetch: 0.000094 Write_Client: 0.000386
SET timestamp=1535017215351;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:15.392000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.039805 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000217 Prepare_Push: 0.000804 dn2_First_Result_Fetch: 0.017410 dn1_First_Result_Fetch: 0.017468 dn2_Last_Result_Fetch: 0.001490 dn1_Last_Result_Fetch: 0.001223 Write_Client: 0.021374
SET timestamp=1535017215392;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:15.410000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
```

```
# Query_time: 0.017384 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000152 Prepare_Push: 0.001183 dn1_0_First_Result
_Fetch: 0.005037 dn1_0_Last_Result_Fetch: 0.007164 dn2_0_First_Result_Fetch: 0.008156 dn2_0_Last_Result_Fetch: 0.004962 Write_Client:
0.004043
SET timestamp=1535017215410;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:15.434000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.021341 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000318 Prepare_Push: 0.002764 dn1_0_First_Result
_Fetch: 0.010897 dn1_0_Last_Result_Fetch: 0.000544 Generate_New_Query: 0.000798 dn1_1_First_Result_Fetch: 0.004506 dn1_1_Last_Result_F
etch: 0.000790 Write_Client: 0.000845
SET timestamp=1535017215434;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:16.322000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.030106 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000217 Prepare_Push: 0.001253 dn1_First_Result_F
etch: 0.028330 dn1_Last_Result_Fetch: 0.000086 Write_Client: 0.000306
SET timestamp=1535017216322;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:16.353000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.030005 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001306 Prepare_Push: 0.001004 dn2_First_Result_F
etch: 0.027242 dn2_Last_Result_Fetch: 0.000140 Write_Client: 0.000453
SET timestamp=1535017216353;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:16.403000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.049615 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001459 Prepare_Push: 0.000830 dn2_First_Result_F
etch: 0.024286 dn1_First_Result_Fetch: 0.025469 dn2_Last_Result_Fetch: 0.001726 dn1_Last_Result_Fetch: 0.000853 Write_Client: 0.023039
SET timestamp=1535017216403;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:16.526000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.121702 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000337 Prepare_Push: 0.000889 dn1_0_First_Result
_Fetch: 0.009370 dn1_0_Last_Result_Fetch: 0.002010 dn2_0_First_Result_Fetch: 0.009160 dn2_0_Last_Result_Fetch: 0.001779 Write_Client:
0.109753
SET timestamp=1535017216526;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:16.560000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.030306 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001534 Prepare_Push: 0.001759 dn1_0_First_Result
_Fetch: 0.011846 dn1_0_Last_Result_Fetch: 0.001663 Generate_New_Query: 0.003223 dn1_1_First_Result_Fetch: 0.006428 dn1_1_Last_Result_F
etch: 0.002601 Write_Client: 0.002291
SET timestamp=1535017216560;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:17.325000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.017545 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.006231 Prepare_Push: 0.002335 dn1_First_Result_F
etch: 0.008121 dn1_Last_Result_Fetch: 0.000277 Write_Client: 0.000857
SET timestamp=1535017217325;
select * from sharding_two_node where id =1;
# Time: 2018-08-23T17:40:17.390000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.026216 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000854 Prepare_Push: 0.000904 dn1_First_Result_F
etch: 0.024157 dn1_Last_Result_Fetch: 0.000081 Write_Client: 0.000301
SET timestamp=1535017217390;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:17.411000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.020095 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000699 Prepare_Push: 0.000711 dn2_First_Result_F
etch: 0.017634 dn2_Last_Result_Fetch: 0.000132 Write_Client: 0.001051
SET timestamp=1535017217411;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:17.491000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.078505 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001702 Prepare_Push: 0.000763 dn2_First_Result_F
etch: 0.018547 dn1_First_Result_Fetch: 0.018482 dn2_Last_Result_Fetch: 0.036637 dn1_Last_Result_Fetch: 0.000566 Write_Client: 0.057558
SET timestamp=1535017217491;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:17.518000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.026112 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000686 Prepare_Push: 0.000872 dn1_0_First_Result
_Fetch: 0.007054 dn1_0_Last_Result_Fetch: 0.001072 dn2_0_First_Result_Fetch: 0.005839 dn2_0_Last_Result_Fetch: 0.017248 Write_Client:
0.016586
SET timestamp=1535017217518;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:17.558000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.038199 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000966 Prepare_Push: 0.005189 dn1_0_First_Result
```

```
_Fetch: 0.013137 dn1_0_Last_Result_Fetch: 0.001134 Generate_New_Query: 0.003973 dn1_1_First_Result_Fetch: 0.010228 dn1_1_Last_Result_F
etch: 0.003564 Write_Client: 0.002115
SET timestamp=1535017217558;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:18.353000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.019048 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.003008 Prepare_Push: 0.000844 dn2_First_Result_F
etch: 0.006415 dn1_First_Result_Fetch: 0.009082 dn2_Last_Result_Fetch: 0.000323 dn1_Last_Result_Fetch: 0.005902 Write_Client: 0.008781
SET timestamp=1535017218353;
select * from sharding_two_node;
# Time: 2018-08-23T17:40:18.410000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.025498 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000060 Prepare_Push: 0.000696 dn1_First_Result_F
etch: 0.024394 dn1_Last_Result_Fetch: 0.000084 Write_Client: 0.000348
SET timestamp=1535017218410;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:18.430000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.018794 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000047 Prepare_Push: 0.001301 dn2_First_Result_F
etch: 0.017073 dn2_Last_Result_Fetch: 0.000099 Write_Client: 0.000373
SET timestamp=1535017218430;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:18.471000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.039810 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000052 Prepare_Push: 0.000661 dn2_First_Result_F
etch: 0.019799 dn1_First_Result_Fetch: 0.019923 dn2_Last_Result_Fetch: 0.000698 dn1_Last_Result_Fetch: 0.000814 Write_Client: 0.019298
SET timestamp=1535017218471;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:18.484000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.012214 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000047 Prepare_Push: 0.001782 dn1_0_First_Result
_Fetch: 0.007109 dn1_0_Last_Result_Fetch: 0.001544 dn2_0_First_Result_Fetch: 0.005518 dn2_0_Last_Result_Fetch: 0.001470 Write_Client:
0.003568
SET timestamp=1535017218484;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:18.507000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.019695 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000046 Prepare_Push: 0.001448 dn1_0_First_Result
_Fetch: 0.006244 dn1_0_Last_Result_Fetch: 0.000988 Generate_New_Query: 0.001564 dn1_1_First_Result_Fetch: 0.007080 dn1_1_Last_Result_F
etch: 0.001306 Write_Client: 0.001137
SET timestamp=1535017218507;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:19.351000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.020937 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000059 Prepare_Push: 0.000800 dn1_First_Result_F
etch: 0.019607 dn1_Last_Result_Fetch: 0.000169 Write_Client: 0.000472
SET timestamp=1535017219351;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:19.370000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.018011 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001184 Prepare_Push: 0.000583 dn2_First_Result_F
etch: 0.015894 dn2_Last_Result_Fetch: 0.000129 Write_Client: 0.000351
SET timestamp=1535017219370;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:19.412000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.041319 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000689 Prepare_Push: 0.000573 dn2_First_Result_F
etch: 0.017735 dn1_First_Result_Fetch: 0.017876 dn2_Last_Result_Fetch: 0.000601 dn1_Last_Result_Fetch: 0.000806 Write_Client: 0.022322
SET timestamp=1535017219412;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:19.423000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.010063 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000200 Prepare_Push: 0.001136 dn1_0_First_Result
_Fetch: 0.006820 dn1_0_Last_Result_Fetch: 0.000694 dn2_0_First_Result_Fetch: 0.003519 dn2_0_Last_Result_Fetch: 0.003944 Write_Client:
0.001443
SET timestamp=1535017219423;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:19.454000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.027592 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000182 Prepare_Push: 0.012798 dn1_0_First_Result
_Fetch: 0.005960 dn1_0_Last_Result_Fetch: 0.000530 Generate_New_Query: 0.000811 dn1_1_First_Result_Fetch: 0.005659 dn1_1_Last_Result_F
etch: 0.000926 Write_Client: 0.001101
SET timestamp=1535017219454;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:20.312000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.025903 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001470 Prepare_Push: 0.000887 dn1_First_Result_F
etch: 0.022114 dn1_Last_Result_Fetch: 0.000197 Write_Client: 0.001433
```

```

SET timestamp=1535017220312;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:20.342000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.028503 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.005643 Prepare_Push: 0.001172 dn2_First_Result_F
etch: 0.021342 dn2_Last_Result_Fetch: 0.000074 Write_Client: 0.000346
SET timestamp=1535017220342;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:20.381000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.037424 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000641 Prepare_Push: 0.000959 dn2_First_Result_F
etch: 0.015139 dn1_First_Result_Fetch: 0.015238 dn2_Last_Result_Fetch: 0.000795 dn1_Last_Result_Fetch: 0.000956 Write_Client: 0.020685
SET timestamp=1535017220381;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:20.408000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.016143 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000122 Prepare_Push: 0.001979 dn1_0_First_Result
_Fetch: 0.004408 dn1_0_Last_Result_Fetch: 0.000484 Generate_New_Query: 0.000965 dn1_1_First_Result_Fetch: 0.006059 dn1_1_Last_Result_F
etch: 0.001553 Write_Client: 0.000755
SET timestamp=1535017220408;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:21.214000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.023376 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000073 Prepare_Push: 0.001306 dn1_First_Result_F
etch: 0.021694 dn1_Last_Result_Fetch: 0.000081 Write_Client: 0.000302
SET timestamp=1535017221214;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:21.241000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.025408 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000083 Prepare_Push: 0.001029 dn2_First_Result_F
etch: 0.023856 dn2_Last_Result_Fetch: 0.000122 Write_Client: 0.000440
SET timestamp=1535017221241;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:21.281000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.038482 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000087 Prepare_Push: 0.000871 dn2_First_Result_F
etch: 0.016690 dn1_First_Result_Fetch: 0.016708 dn2_Last_Result_Fetch: 0.000579 dn1_Last_Result_Fetch: 0.000891 Write_Client: 0.020835
SET timestamp=1535017221281;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:21.293000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.011657 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000615 Prepare_Push: 0.001320 dn1_0_First_Result
_Fetch: 0.006906 dn1_0_Last_Result_Fetch: 0.001589 dn2_0_First_Result_Fetch: 0.005105 dn2_0_Last_Result_Fetch: 0.001548 Write_Client:
0.003341
SET timestamp=1535017221293;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:21.312000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.017169 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000635 Prepare_Push: 0.001609 dn1_0_First_Result
_Fetch: 0.006997 dn1_0_Last_Result_Fetch: 0.000728 Generate_New_Query: 0.001037 dn1_1_First_Result_Fetch: 0.004816 dn1_1_Last_Result_F
etch: 0.000709 Write_Client: 0.000703
SET timestamp=1535017221312;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:22.150000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.026153 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000180 Prepare_Push: 0.000771 dn1_First_Result_F
etch: 0.024940 dn1_Last_Result_Fetch: 0.000061 Write_Client: 0.000261
SET timestamp=1535017222150;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:22.170000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.019181 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000180 Prepare_Push: 0.000642 dn2_First_Result_F
etch: 0.018060 dn2_Last_Result_Fetch: 0.000088 Write_Client: 0.000299
SET timestamp=1535017222170;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:22.220000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.049834 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000261 Prepare_Push: 0.000735 dn2_First_Result_F
etch: 0.019862 dn1_First_Result_Fetch: 0.019807 dn2_Last_Result_Fetch: 0.000418 dn1_Last_Result_Fetch: 0.000655 Write_Client: 0.029031
SET timestamp=1535017222220;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:22.240000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.019128 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000934 Prepare_Push: 0.002731 dn1_0_First_Result
_Fetch: 0.013430 dn1_0_Last_Result_Fetch: 0.000521 dn2_0_First_Result_Fetch: 0.003296 dn2_0_Last_Result_Fetch: 0.001243 Write_Client:
0.011072
SET timestamp=1535017222240;
select count(*) from sharding_two_node;

```



```
# Time: 2018-08-23T17:40:22.270000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.028479 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.003233 Prepare_Push: 0.004986 dn1_0_First_Result_Fetch: 0.009870 dn1_0_Last_Result_Fetch: 0.001172 Generate_New_Query: 0.001590 dn1_1_First_Result_Fetch: 0.006060 dn1_1_Last_Result_Fetch: 0.000771 Write_Client: 0.000700
SET timestamp=1535017222270;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:23.097000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.053956 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000056 Prepare_Push: 0.001034 dn1_First_Result_Fetch: 0.052343 dn1_Last_Result_Fetch: 0.000174 Write_Client: 0.000523
SET timestamp=1535017223097;
select * from sharding_two_node where id =1;
# Time: 2018-08-23T17:40:23.110000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.010839 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000499 Prepare_Push: 0.000680 dn2_First_Result_Fetch: 0.006349 dn1_First_Result_Fetch: 0.009082 dn2_Last_Result_Fetch: 0.000270 dn1_Last_Result_Fetch: 0.000333 Write_Client: 0.003311
SET timestamp=1535017223110;
select * from sharding_two_node;
# Time: 2018-08-23T17:40:23.181000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.027573 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000109 Prepare_Push: 0.000980 dn2_First_Result_Fetch: 0.026156 dn2_Last_Result_Fetch: 0.000086 Write_Client: 0.000328
SET timestamp=1535017223181;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:23.231000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.049380 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.002435 Prepare_Push: 0.000670 dn2_First_Result_Fetch: 0.025278 dn1_First_Result_Fetch: 0.025242 dn2_Last_Result_Fetch: 0.000392 dn1_Last_Result_Fetch: 0.000629 Write_Client: 0.021032
SET timestamp=1535017223231;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:23.268000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.025207 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000060 Prepare_Push: 0.001492 dn1_0_First_Result_Fetch: 0.007693 dn1_0_Last_Result_Fetch: 0.000752 Generate_New_Query: 0.001946 dn1_1_First_Result_Fetch: 0.008776 dn1_1_Last_Result_Fetch: 0.005040 Write_Client: 0.001884
SET timestamp=1535017223268;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:24.121000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.027104 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001558 Prepare_Push: 0.001107 dn1_First_Result_Fetch: 0.024084 dn1_Last_Result_Fetch: 0.000085 Write_Client: 0.000356
SET timestamp=1535017224121;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:24.141000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.019191 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000072 Prepare_Push: 0.000673 dn2_First_Result_Fetch: 0.017923 dn2_Last_Result_Fetch: 0.000092 Write_Client: 0.000522
SET timestamp=1535017224141;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:24.182000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.039883 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000483 Prepare_Push: 0.000584 dn2_First_Result_Fetch: 0.017241 dn1_First_Result_Fetch: 0.017320 dn2_Last_Result_Fetch: 0.000603 dn1_Last_Result_Fetch: 0.000767 Write_Client: 0.021575
SET timestamp=1535017224182;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:24.196000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.012406 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000073 Prepare_Push: 0.000958 dn1_0_First_Result_Fetch: 0.008102 dn1_0_Last_Result_Fetch: 0.001255 dn2_0_First_Result_Fetch: 0.007566 dn2_0_Last_Result_Fetch: 0.001772 Write_Client: 0.002300
SET timestamp=1535017224196;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:24.218000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.021238 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000864 Prepare_Push: 0.001143 dn1_0_First_Result_Fetch: 0.010305 dn1_0_Last_Result_Fetch: 0.000532 Generate_New_Query: 0.001852 dn1_1_First_Result_Fetch: 0.005359 dn1_1_Last_Result_Fetch: 0.000618 Write_Client: 0.000661
SET timestamp=1535017224218;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:25.093000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.029579 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000106 Prepare_Push: 0.000882 dn1_First_Result_Fetch: 0.028241 dn1_Last_Result_Fetch: 0.000069 Write_Client: 0.000351
SET timestamp=1535017225093;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:25.121000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
```

```
# Query_time: 0.027422 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001867 Prepare_Push: 0.001330 dn2_First_Result_F
etch: 0.023887 dn2_Last_Result_Fetch: 0.000102 Write_Client: 0.000338
SET timestamp=1535017225121;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:25.161000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.038859 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000050 Prepare_Push: 0.000753 dn2_First_Result_F
etch: 0.019091 dn1_First_Result_Fetch: 0.019189 dn2_Last_Result_Fetch: 0.000582 dn1_Last_Result_Fetch: 0.000560 Write_Client: 0.018965
SET timestamp=1535017225161;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:25.191000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.016379 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000043 Prepare_Push: 0.001276 dn1_0_First_Result
_Fetch: 0.007469 dn1_0_Last_Result_Fetch: 0.000678 Generate_New_Query: 0.001327 dn1_1_First_Result_Fetch: 0.003927 dn1_1_Last_Result_F
etch: 0.000787 Write_Client: 0.000893
SET timestamp=1535017225191;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:26.026000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.029878 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000162 Prepare_Push: 0.000916 dn1_First_Result_F
etch: 0.028497 dn1_Last_Result_Fetch: 0.000084 Write_Client: 0.000303
SET timestamp=1535017226026;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:26.051000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.024231 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001105 Prepare_Push: 0.000469 dn2_First_Result_F
etch: 0.022188 dn2_Last_Result_Fetch: 0.000100 Write_Client: 0.000470
SET timestamp=1535017226051;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:26.091000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.039762 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.001669 Prepare_Push: 0.001915 dn2_First_Result_F
etch: 0.018107 dn1_First_Result_Fetch: 0.018187 dn2_Last_Result_Fetch: 0.000633 dn1_Last_Result_Fetch: 0.000832 Write_Client: 0.018071
SET timestamp=1535017226091;
insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:26.105000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.012664 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000261 Prepare_Push: 0.000935 dn1_0_First_Result
_Fetch: 0.007328 dn1_0_Last_Result_Fetch: 0.000733 dn2_0_First_Result_Fetch: 0.006229 dn2_0_Last_Result_Fetch: 0.002592 Write_Client:
0.003554
SET timestamp=1535017226105;
select count(*) from sharding_two_node;
# Time: 2018-08-23T17:40:26.134000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.028335 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000046 Prepare_Push: 0.003442 dn1_0_First_Result
_Fetch: 0.009563 dn1_0_Last_Result_Fetch: 0.001069 Generate_New_Query: 0.001856 dn1_1_First_Result_Fetch: 0.010875 dn1_1_Last_Result_F
etch: 0.000798 Write_Client: 0.000712
SET timestamp=1535017226134;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
# Time: 2018-08-23T17:40:26.859000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.014882 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000067 Prepare_Push: 0.001351 dn1_First_Result_F
etch: 0.013084 dn1_Last_Result_Fetch: 0.000137 Write_Client: 0.000381
SET timestamp=1535017226859;
select * from sharding_two_node where id =1;
# Time: 2018-08-23T17:40:26.874000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.010509 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000066 Prepare_Push: 0.001761 dn2_First_Result_F
etch: 0.006921 dn1_First_Result_Fetch: 0.008256 dn2_Last_Result_Fetch: 0.000279 dn1_Last_Result_Fetch: 0.000211 Write_Client: 0.001761
SET timestamp=1535017226874;
select * from sharding_two_node;
# Time: 2018-08-23T17:40:26.931000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.028690 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000237 Prepare_Push: 0.001126 dn1_First_Result_F
etch: 0.026194 dn1_Last_Result_Fetch: 0.000640 Write_Client: 0.001133
SET timestamp=1535017226931;
delete from sharding_two_node where id =15;
# Time: 2018-08-23T17:40:26.951000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.018818 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000342 Prepare_Push: 0.001671 dn2_First_Result_F
etch: 0.016482 dn2_Last_Result_Fetch: 0.000063 Write_Client: 0.000323
SET timestamp=1535017226951;
delete from sharding_two_node where id =519;
# Time: 2018-08-23T17:40:26.991000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.039399 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000082 Prepare_Push: 0.000706 dn2_First_Result_F
etch: 0.019233 dn1_First_Result_Fetch: 0.019167 dn2_Last_Result_Fetch: 0.000426 dn1_Last_Result_Fetch: 0.000739 Write_Client: 0.019444
SET timestamp=1535017226991;
```

```

insert into sharding_two_node values(15,'15',15),(519,'519',519);
# Time: 2018-08-23T17:40:27.032000Z
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
# Query_time: 0.029495 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.000064 Prepare_Push: 0.001349 dn1_0_First_Result_Fetch: 0.005745 dn1_0_Last_Result_Fetch: 0.000632 Generate_New_Query: 0.001056 dn1_1_First_Result_Fetch: 0.018101 dn1_1_Last_Result_Fetch: 0.002282 Write_Client: 0.000863
SET timestamp=1535017227032;
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);

```

2.20.3.1 mysqldumpslow 结果:

```

Reading mysql slow query log from /tmp/slow3.log
Count: 17 Time=0.05s (0s) Lock=0.00s (0s) Rows=0.0 (0), root[root]@[0:0:0:0:0:0:1]
insert into sharding_two_node values(N,'S',N),(N,'S',N)

Count: 13 Time=0.05s (0s) Lock=0.00s (0s) Rows=0.0 (0), root[root]@[0:0:0:0:0:0:1]
select count(*) from sharding_two_node

Count: 6 Time=0.04s (0s) Lock=0.00s (0s) Rows=0.0 (0), root[root]@[0:0:0:0:0:0:1]
select * from sharding_two_node where id =N

Count: 33 Time=0.03s (0s) Lock=0.00s (0s) Rows=0.0 (0), root[root]@[0:0:0:0:0:0:1]
delete from sharding_two_node where id =N

Count: 17 Time=0.03s (0s) Lock=0.00s (0s) Rows=0.0 (0), root[root]@[0:0:0:0:0:0:1]
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=N)

Count: 6 Time=0.02s (0s) Lock=0.00s (0s) Rows=0.0 (0), root[root]@[0:0:0:0:0:0:1]
select * from sharding_two_node

```

2.20.3.2 pt-query-digest 结果:

```

# 710ms user time, 70ms system time, 23.35M rss, 68.36M vsz
# Current date: Thu Aug 23 17:48:25 2018
# Hostname: 10-186-24-63
# Files: /tmp/slow_query4.log
# Overall: 92 total, 6 unique, 5.41 QPS, 0.18x concurrency _____
# Time range: 2018-08-23T17:40:10 to 2018-08-23T17:40:27
# Attribute      total      min       max       avg       95%      stddev  median
# =====
# Exec time      3s        10ms     339ms    34ms     75ms    37ms    27ms
# Lock time      0          0         0         0         0         0         0
# Rows sent      0          0         0         0         0         0         0
# Rows examine   0          0         0         0         0         0         0
# Query size     4.91k     31        94       54.64    92.72   20.87   42.48
# Generate New   0.03      0.00     0.00     0.00     0.00     0.00     0.00
# Prepare Push   0.56      0.00     0.26     0.01     0.01     0.03     0.00
# Read SQL      0.07      0.00     0.01     0.00     0.00     0.00     0.00
# Write Client   0.70      0.00     0.11     0.01     0.02     0.01     0.00
# dn1 0 First    0.29      0.00     0.05     0.01     0.01     0.01     0.01
# dn1 0 Last R   0.07      0.00     0.03     0.00     0.01     0.01     0.00
# dn1 1 First    0.12      0.00     0.02     0.01     0.01     0.00     0.01
# dn1 1 Last R   0.03      0.00     0.01     0.00     0.00     0.00     0.00
# dn1 First Re   0.93      0.01     0.05     0.02     0.03     0.01     0.02
# dn1 Last Res   0.04      0.00     0.01     0.00     0.00     0.00     0.00
# dn2 0 First    0.13      0.00     0.04     0.01     0.02     0.01     0.01
# dn2 0 Last R   0.08      0.00     0.03     0.01     0.02     0.01     0.00
# dn2 First Re   0.80      0.01     0.06     0.02     0.03     0.01     0.02
# dn2 Last Res   0.09      0.00     0.04     0.00     0.01     0.01     0.00

# Profile
# Rank Query ID      Response time Calls R/Call V/M  I
# =====
# 1 0x13233F8ADA41C6E2D889AEE0C2B... 0.8815 28.1% 33 0.0267 0.00 DELETE sharding_two_node
# 2 0xF68D16B46E487184E8FD3BB3912... 0.8525 27.1% 17 0.0501 0.01 INSERT sharding_two_node
# 3 0xB46D813C53609C853F7CBA6D2DB... 0.6306 20.1% 13 0.0485 0.15 SELECT sharding_two_node
# 4 0x3FB41587E746A475282C1ED2606... 0.4335 13.8% 17 0.0255 0.00 SELECT sharding_two_node
# 5 0x04CDF91DDFC4E1DD7A22E312C72... 0.2399 7.6% 6 0.0400 0.05 SELECT sharding_two_node
# MISC 0xMISC 0.1028 3.3% 6 0.0171 0.0 <1 ITEMS>

# Query 1: 2.06 QPS, 0.06x concurrency, ID 0x13233F8ADA41C6E2D889AEE0C2BC6CB5 at byte 943
# Scores: V/M = 0.00
# Time range: 2018-08-23T17:40:10 to 2018-08-23T17:40:26
# Attribute      pct  total      min       max       avg       95%      stddev  median
# =====

```

```

# Count          35      33
# Exec time      28    882ms   18ms   45ms   27ms   31ms   5ms   27ms
# Lock time      0      0      0      0      0      0      0      0
# Rows sent      0      0      0      0      0      0      0      0
# Rows examine   0      0      0      0      0      0      0      0
# Query size     27    1.37k    42     43    42.52  42.48  0.50  42.48
# Prepare Push   6      0.04    0.00   0.00   0.00   0.00   0.00   0.00
# Read SQL       35    0.03    0.00   0.01   0.00   0.00   0.00   0.00
# Write Client   2      0.02    0.00   0.00   0.00   0.00   0.00   0.00
# dn1 First Re   45    0.42    0.02   0.04   0.03   0.03   0.01   0.03
# dn1 Last Res   7      0.00    0.00   0.00   0.00   0.00   0.00   0.00
# dn2 First Re   47    0.38    0.02   0.03   0.02   0.03   0.00   0.02
# dn2 Last Res   3      0.00    0.00   0.00   0.00   0.00   0.00   0.00
# String:
# Hosts          0:0:0:0:0:0:1
# Users          root
# Query_time distribution
# 1us
# 10us
# 100us
# 1ms
# 10ms #####
# 100ms
# 1s
# 10s+
# Tables
# SHOW TABLE STATUS LIKE 'sharding_two_node'\G
# SHOW CREATE TABLE `sharding_two_node`\G
delete from sharding_two_node where id =15\G
# Converted for EXPLAIN
# EXPLAIN /*!50100 PARTITIONS*/
select * from sharding_two_node where id =15\G

# Query 2: 1.06 QPS, 0.05x concurrency, ID 0xF68D16B46E487184E8FD3BB3912A3658 at byte 1690
# Scores: V/M = 0.01
# Time range: 2018-08-23T17:40:10 to 2018-08-23T17:40:26
# Attribute      pct  total   min    max    avg    95%  stddev  median
# =====
# Count          18    17
# Exec time      27    853ms   37ms   117ms  50ms   78ms   21ms   40ms
# Lock time      0      0      0      0      0      0      0      0
# Rows sent      0      0      0      0      0      0      0      0
# Rows examine   0      0      0      0      0      0      0      0
# Query size     21    1.06k    64     64     64     64     0      64
# Prepare Push   7      0.04    0.00   0.02   0.00   0.00   0.01   0.00
# Read SQL       32    0.02    0.00   0.01   0.00   0.00   0.00   0.00
# Write Client   63    0.45    0.02   0.06   0.03   0.05   0.01   0.02
# dn1 First Re   37    0.35    0.01   0.04   0.02   0.03   0.00   0.02
# dn1 Last Res   40    0.01    0.00   0.00   0.00   0.00   0.00   0.00
# dn2 First Re   46    0.37    0.01   0.06   0.02   0.03   0.01   0.02
# dn2 Last Res   72    0.07    0.00   0.04   0.00   0.02   0.01   0.00
# String:
# Hosts          0:0:0:0:0:0:1
# Users          root
# Query_time distribution
# 1us
# 10us
# 100us
# 1ms
# 10ms #####
# 100ms #####
# 1s
# 10s+
# Tables
# SHOW TABLE STATUS LIKE 'sharding_two_node'\G
# SHOW CREATE TABLE `sharding_two_node`\G
insert into sharding_two_node values(15,'15',15) /*... omitted ...*/\G

# Query 3: 0.81 QPS, 0.04x concurrency, ID 0xB46D813C53609C853F7CBA6D2DB4047C at byte 2152
# Scores: V/M = 0.15
# Time range: 2018-08-23T17:40:10 to 2018-08-23T17:40:26
# Attribute      pct  total   min    max    avg    95%  stddev  median
# =====
# Count          14    13
# Exec time      20    631ms   10ms   339ms  49ms   116ms  84ms   12ms
# Lock time      0      0      0      0      0      0      0      0
# Rows sent      0      0      0      0      0      0      0      0
# Rows examine   0      0      0      0      0      0      0      0

```

```

# Query size      9      494      38      38      38      38      0      38
# Prepare Push   48     0.27     0.00     0.26     0.02     0.00     0.07     0.00
# Read SQL       5     0.00     0.00     0.00     0.00     0.00     0.00     0.00
# Write Client   24     0.17     0.00     0.11     0.01     0.02     0.03     0.00
# dn1 0 First    48     0.14     0.00     0.05     0.01     0.01     0.01     0.01
# dn1 0 Last R   80     0.06     0.00     0.03     0.00     0.01     0.01     0.00
# dn2 0 First   100     0.13     0.00     0.04     0.01     0.02     0.01     0.01
# dn2 0 Last R  100     0.08     0.00     0.03     0.01     0.02     0.01     0.00
# String:
# Hosts          0:0:0:0:0:0:1
# Users          root
# Query_time distribution
# 1us
# 10us
# 100us
# 1ms
# 10ms #####
# 100ms #####
# 1s
# 10s+
# Tables
# SHOW TABLE STATUS LIKE 'sharding_two_node'\G
# SHOW CREATE TABLE `sharding_two_node`\G
# EXPLAIN /*!50100 PARTITIONS*/
select count(*) from sharding_two_node\G

# Query 4: 1 QPS, 0.03x concurrency, ID 0x3FB41587E746A475282C1ED2606795FB at byte 2596
# Scores: V/M = 0.00
# Time range: 2018-08-23T17:40:10 to 2018-08-23T17:40:27
# Attribute      pct      total      min      max      avg      95%      stddev      median
# =====      ==      =====      =====      =====      =====      =====      =====
# Count          18       17
# Exec time      13      434ms     16ms     47ms     26ms     38ms      8ms       26ms
# Lock time      0        0         0         0         0         0         0         0
# Rows sent      0        0         0         0         0         0         0         0
# Rows examine   0        0         0         0         0         0         0         0
# Query size     31      1.56k     94        94        94        94         0         94
# Generate New  100     0.03     0.00     0.00     0.00     0.00     0.00     0.00
# Prepare Push   13     0.07     0.00     0.03     0.00     0.01     0.01     0.00
# Read SQL       12     0.01     0.00     0.00     0.00     0.00     0.00     0.00
# Write Client   2      0.02     0.00     0.00     0.00     0.00     0.00     0.00
# dn1 0 First    51     0.15     0.00     0.01     0.01     0.01     0.00     0.01
# dn1 0 Last R   19     0.01     0.00     0.00     0.00     0.00     0.00     0.00
# dn1 1 First   100     0.12     0.00     0.02     0.01     0.01     0.00     0.01
# dn1 1 Last R  100     0.03     0.00     0.01     0.00     0.00     0.00     0.00
# String:
# Hosts          0:0:0:0:0:0:1
# Users          root
# Query_time distribution
# 1us
# 10us
# 100us
# 1ms
# 10ms #####
# 100ms
# 1s
# 10s+
# Tables
# SHOW TABLE STATUS LIKE 'sharding_two_node'\G
# SHOW CREATE TABLE `sharding_two_node`\G
# EXPLAIN /*!50100 PARTITIONS*/
select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1)\G

# Query 5: 0.38 QPS, 0.01x concurrency, ID 0x04CDF91DDFC4E1DD7A22E312C72C268D at byte 0
# Scores: V/M = 0.05
# Time range: 2018-08-23T17:40:10 to 2018-08-23T17:40:26
# Attribute      pct      total      min      max      avg      95%      stddev      median
# =====      ==      =====      =====      =====      =====      =====      =====
# Count          6        6
# Exec time      7      240ms     10ms     133ms     40ms     128ms     43ms     35ms
# Lock time      0        0         0         0         0         0         0         0
# Rows sent      0        0         0         0         0         0         0         0
# Rows examine   0        0         0         0         0         0         0         0
# Query size     5      258       43        43        43        43         0         43
# Prepare Push   22     0.12     0.00     0.12     0.02     0.12     0.04     0.00
# Read SQL       9     0.01     0.00     0.01     0.00     0.01     0.00     0.00
# Write Client   0     0.01     0.00     0.00     0.00     0.00     0.00     0.00
# dn1 First Re  11     0.10     0.01     0.05     0.02     0.05     0.02     0.01

```

```
# dn1 Last Res 6 0.00 0.00 0.00 0.00 0.00 0.00 0.00
# String:
# Hosts 0:0:0:0:0:0:1
# Users root
# Query_time distribution
# 1us
# 10us
# 100us
# 1ms
# 10ms #####
# 100ms #####
# 1s
# 10s+
# Tables
# SHOW TABLE STATUS LIKE 'sharding_two_node'\G
# SHOW CREATE TABLE `sharding_two_node`\G
# EXPLAIN /*!50100 PARTITIONS*/
select * from sharding_two_node where id =1\G
```

2.20.3.3 限制

当mysqldumpslow版本为5.6或更早的版本时，在解析慢日志的过程中，由于mysqldumpslow只能识别'190428 10:28:16'格式的Time字段，而dble却使用了'2019-04-28T10:28:16.515000Z'字段格式，因此解析生成的慢日志会出错。详情参见issue:<https://github.com/actiontech/dble/issues/908>

2.20.4 附：慢查询日志格式解析

2.20.4.1 MySQL 慢查询日志格式

先放一段正常记录的MySQL慢日志

```

/usr/local/mysql5.7.11/bin/mysqld-debug, Version: 5.7.11-debug-log (MySQL Community Server - Debug (GPL)). started with:

Tcp port: 3320 Unix socket: /tmp/mysql_3320.sock

Time          Id Command  Argument

# Time: 2018-05-15T10:53:23.798040Z

# User@Host: action[action] @ [192.168.2.206] Id: 436

# Query_time: 296.145816 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0

use test;

SET timestamp=1526381603;

drop table sharding_two_node;

# Time: 2018-05-15T11:32:25.549290Z

# User@Host: action[action] @ [192.168.2.206] Id: 451

# Query_time: 129.555883 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0

use nosharding;

SET timestamp=1526383945;

drop table test4;

# Time: 2018-05-15T11:32:25.550190Z

# User@Host: action[action] @ [192.168.2.206] Id: 454

# Query_time: 84.316518 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0

SET timestamp=1526383945;

insert into test4 values(1, '1');

# Time: 2018-05-15T11:37:01.079214Z

# User@Host: action[action] @ [192.168.2.206] Id: 483

# Query_time: 49.571983 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0

SET timestamp=1526384221;

drop table test3;

# Time: 2018-07-11T05:28:34.161405Z

# User@Host: action[action] @ [192.168.2.206] Id: 16421

# Query_time: 10.035706 Lock_time: 0.000000 Rows_sent: 1 Rows_examined: 0

use test;

SET timestamp=1531286914;

insert into test4 values(1, '1');

```

我们从MySQL源代码上来分析慢日志的格式:

1.1 文件头: (包含版本信息等)

```
/usr/local/mysql5.7.11/bin/mysqld-debug, Version: 5.7.11-debug-log (MySQL Community Server - Debug (GPL)). started with:
```

```
Tcp port: 3320 Unix socket: /tmp/mysql_3320.sock
```

```
Time Id Command Argument
```

如果MySQL实例没有专门设置参数log-short-format, 则会有time行和session信息行

1.2 time行

例如:

```
# Time: 2018-05-15T10:53:23.798040Z
```

1.3 session信息行

例如:

```
# User@Host: action[action] @ [192.168.2.206] Id: 436
```

1.4 执行时间行

注意用两个空格分隔不同键值对, 用单个空格区分key和value

```
# Query_time: 296.145816 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0
```

1.5 如果有Database changed, 则有转换schema行:

```
use nosharding;
```

1.6 set行(注: mysqldumpslow官方工具只处理了SET timestamp=)

举例

```
SET timestamp=1526383945;
```

1.6.1 last_insert_id

如果设置了

```
stmt_depends_on_first_successful_insert_id_in_prev_stmt (含义待调查)
```

会有last_insert_id=某个值

1.6.2 insert_id

如果没有设置log-short-format, 并且

```
auto_inc_intervals_in_cur_stmt_for_binlog.nb_elements() (含义待调查)
```

会有last_id=某个值

1.6.3 timestamp=

设置时间戳

1.7 命令行

例如:

```
# administrator command
```

如果是is_command (含义待调查), 则有此行

1.8 SQL语句行

例如

```
insert into test4 values(1, '1');
```

2.20.4.2 dble 慢日志格式

为了兼容mysqldumpslow和pt-query-digest 工具, dble的慢日志格式如下:

2.1 文件头

```
/FAKE_PATH/mysqld, Version: FAKE_VERSION. started with:
```

```
Tcp port: 3320 Unix socket: FAKE_SOCKET
```

```
Time          Id Command  Argument
```

2.2 time行

因为java8目前只能获取到毫秒级别的绝对时间戳, 所以时间戳后三位为0, 与mysql不同, 例如:

```
# Time: 2018-08-23T17:40:10.149000Z
```

2.3 session信息行

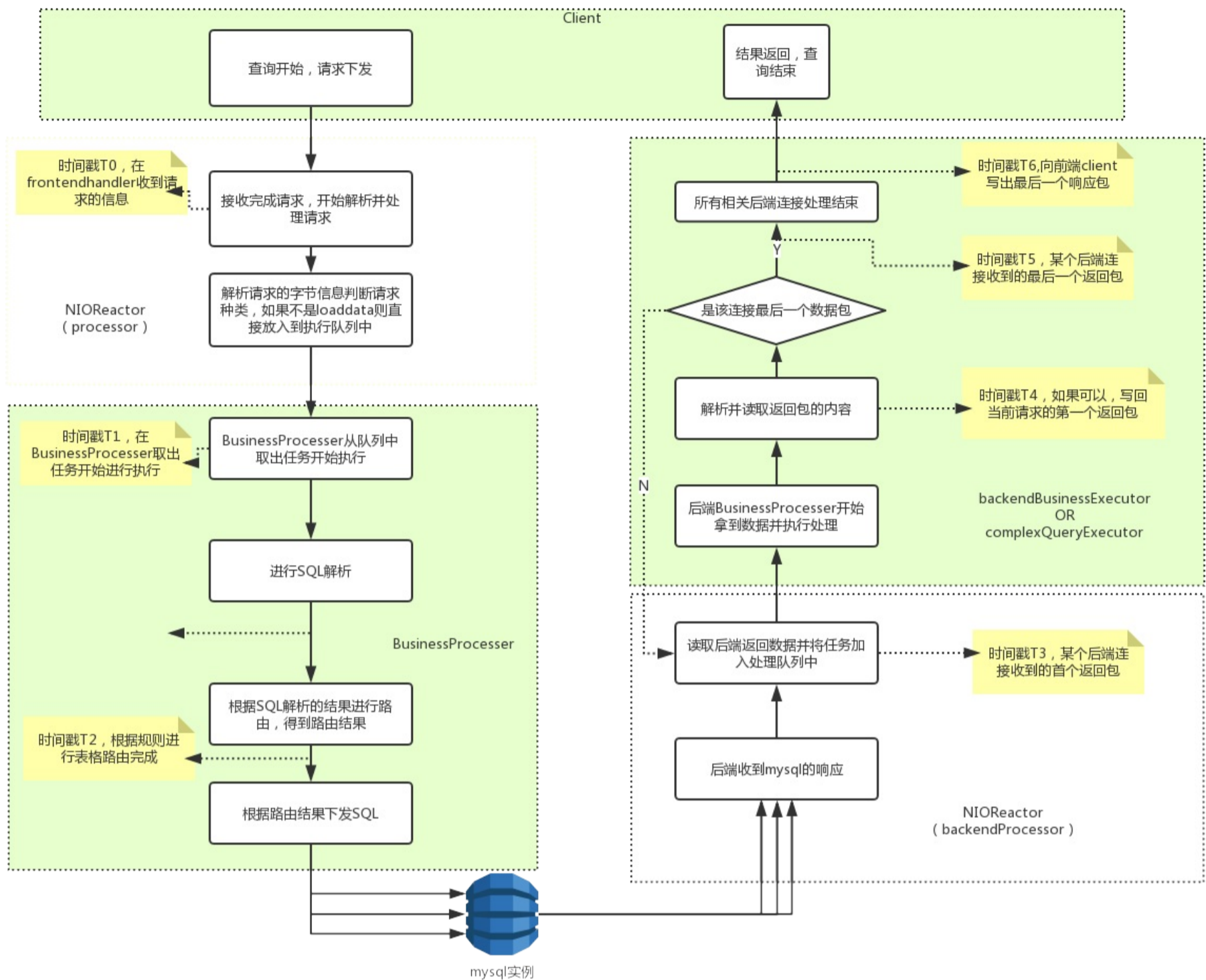
例如:

```
# User@Host: root[root] @ [0:0:0:0:0:0:1] Id: 2
```

2.4 执行时间行

注意用两个空格分隔不同键值对，用单个空格区分key和value.

如图，这是dble的内部流程，根据此图：



我们在兼容MySQL的情况下增加了如下字段：

Read_SQL: 接受到SQL字节数据开始到将字节数据转为可供解析的SQL字符串 的耗时，即T1-T0

Prepare_Push: 解析/路由/尝试优化及其他准备工作耗时，与Read_SQL是线性顺序的,即T2-T1

{shardingnodeName}_First_Result_Fetch: 某个shardingnodeName收到的第一个数据包，相对于Prepare_Push结束时的耗时,即T3-T2。另外：不同的shardingnodeName之间是并行的

{shardingnodeName}_Last_Result_Fetch: 某个shardingnodeName收到的最后一个数据包，相对于自己的{shardingnodeName}_First_Result_Fetch的耗时,即T5-T3

Inner_Execute: 执行 show /set ... 、 commit等非业务型sql语句的执行时间

Write_Client: 可以开始返回客户端（收到一个后端返回数据包）到最后一个数据包准备写出的时间，如果是复杂查询，可能包含合并等操作的结果,即T6-T4

另外，MySQL原有的Query_time为dble开始读取SQL到准备写回最后一个数据包的时间间隔,即T6-T0

Lock_time, Rows_sent, Rows_examined用0填充

例如：

```
# Query_time: 0.116672 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0 Read_SQL: 0.013625 Prepare_Push: 0.024767 dn2_First_Result_Fetch: 0.056395 dn1_First_Result_Fetch: 0.026420 dn2_Last_Result_Fetch: 0.000743 dn1_Last_Result_Fetch: 0.001700 Write_Client: 0.051861
```

2.5 set行(只处理了SET timestamp=)

例如：

```
SET timestamp=1535017210432;
```

2.6 SQL语句行

例如

```
insert into sharding_two_node values(15, '15', 15), (519, '519', 519);
```

2.21 单条SQL性能trace

这个功能与MySQL的profile功能类似，用于统计一条查询在dble内部的各个阶段的耗时情况，用来发现性能瓶颈，改进SQL或者dble配置，达到提高性能的目的。此功能为session级别。

1. 可以通过此命令检查trace功能是否开启

```
mysql> select @@trace;
+-----+
| @@trace |
+-----+
| 0       |
+-----+
1 row in set (0.02 sec)
```

2. 开启 trace 功能

```
mysql> set trace =1;
Query OK, 0 rows affected (0.09 sec)

mysql> select @@trace;
+-----+
| @@trace |
+-----+
| 1       |
+-----+
1 row in set (0.00 sec)
```

3. 单节点查询的trace结果举例

```
mysql> select * from sharding_two_node where id =1;
+-----+-----+-----+
| id | c_flag | c_decimal |
+-----+-----+-----+
| 1 | 1      | 1.0000    |
+-----+-----+-----+
1 row in set (0.02 sec)

mysql> show trace;
+-----+-----+-----+-----+-----+-----+
| OPERATION          | START(ms) | END(ms)   | DURATION(ms) | SHARDING_NODE | SQL/REF          |
+-----+-----+-----+-----+-----+-----+
| Read SQL           | 0.0       | 0.1085    | 0.1085        | -              | -               |
| Parse SQL          | 0.1085    | 0.49607   | 0.38757       | -              | -               |
| Route Calculation  | 0.49607   | 1.274142  | 0.778072      | -              | -               |
| Prepare to Push    | 1.274142  | 1.560543  | 0.286401      | -              | -               |
| Execute SQL        | 1.560543  | 18.711851 | 17.151308     | dn1            | select * from sharding_two_node where id =1 |
| Fetch result       | 18.711851 | 18.978213 | 0.266362      | dn1            | select * from sharding_two_node where id =1 |
| Write to Client    | 18.711851 | 19.276344 | 0.564493      | -              | -               |
| Over All           | 0.0       | 19.276344 | 19.276344     | -              | -               |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

4. 多节点查询的trace结果举例

```
mysql> select * from sharding_two_node ;
```

```
+-----+-----+-----+
| id | c_flag | c_decimal |
+-----+-----+-----+
| 513 | 513 | 513.0000 |
| 514 | 514 | 514.0000 |
| 515 | 515 | 515.0000 |
| 516 | 516 | 516.0000 |
| 1 | 1 | 1.0000 |
| 2 | 2 | 2.0000 |
| 3 | 3 | 3.0000 |
| 4 | 4 | 4.0000 |
| 5 | 5 | 5.0000 |
| 7 | 7 | 7.0000 |
| 8 | 8 | 8.0000 |
| 9 | 9 | 9.0000 |
| 10 | 10 | 10.0000 |
| 11 | 11 | 11.0000 |
| 12 | 12 | 12.0000 |
+-----+-----+-----+
```

```
15 rows in set (0.01 sec)
```

```
mysql> show trace;
```

```
+-----+-----+-----+-----+-----+-----+
| OPERATION | START(ms) | END(ms) | DURATION(ms) | SHARDING_NODE | SQL/REF |
+-----+-----+-----+-----+-----+-----+
| Read SQL | 0.0 | 0.079175 | 0.079175 | - | - |
| Parse SQL | 0.079175 | 0.637315 | 0.55814 | - | - |
| Route Calculation | 0.637315 | 1.046389 | 0.409074 | - | - |
| Prepare to Push | 1.046389 | 1.465238 | 0.418849 | - | - |
| Execute SQL | 1.465238 | 8.141409 | 6.676171 | dn1 | SELECT * FROM sharding_two_node LIMIT 100 |
| Execute SQL | 1.465238 | 7.59109 | 6.125852 | dn2 | SELECT * FROM sharding_two_node LIMIT 100 |
| Fetch result | 8.141409 | 8.817824 | 0.676415 | dn1 | SELECT * FROM sharding_two_node LIMIT 100 |
| Fetch result | 7.59109 | 8.366718 | 0.775628 | dn2 | SELECT * FROM sharding_two_node LIMIT 100 |
| Write to Client | 7.59109 | 9.324157 | 1.733067 | - | - |
| Over All | 0.0 | 9.324157 | 9.324157 | - | - |
+-----+-----+-----+-----+-----+-----+
```

```
10 rows in set (0.00 sec)
```

5. 多节点写入的SQL的trace结果举例,事实上,这是一个隐式分布式事务

```
mysql> insert into sharding_two_node values(15, '15',15),(519, '519',519);
Query OK, 2 rows affected (0.06 sec)
```

```
mysql> show trace;
```

```

+-----+-----+-----+-----+-----+-----+
-----+
| OPERATION          | START(ms) | END(ms)   | DURATION(ms) | SHARDING_NODE | SQL/REF
|
+-----+-----+-----+-----+-----+-----+
-----+
| Read SQL           | 0.0        | 0.131959 | 0.131959     | -              | -
|
| Parse SQL          | 0.131959   | 0.601637 | 0.469678     | -              | -
|
| Route Calculation  | 0.601637   | 0.825479 | 0.223842     | -              | -
|
| Prepare to Push    | 0.825479   | 1.025374 | 0.199895     | -              | -
|
| Execute SQL        | 1.025374   | 27.095675 | 26.070301    | dn1            | INSERT INTO sharding_two_node VALUES (15, '15',
15)
|
| Execute SQL        | 1.025374   | 25.023911 | 23.998537    | dn2            | INSERT INTO sharding_two_node VALUES (519, '519
', 519)
|
| Fetch result       | 27.095675   | 27.405046 | 0.309371     | dn1            | INSERT INTO sharding_two_node VALUES (15, '15',
15)
|
| Fetch result       | 25.023911   | 26.478398 | 1.454487     | dn2            | INSERT INTO sharding_two_node VALUES (519, '519
', 519)
|
| Distributed Transaction Prepare | 27.405046 | 27.736411 | 0.331365     | -              | -
|
| Distributed Transaction Commit | 27.736411 | 57.426311 | 29.6899      | -              | -
|
| Write to Client    | 25.023911   | 57.428266 | 32.404355    | -              | -
|
| Over All           | 0.0         | 57.428266 | 57.428266    | -              | -
|
+-----+-----+-----+-----+-----+-----+
-----+
12 rows in set (0.00 sec)
```

6. 复杂查询的trace结果举例

```

mysql> select count(*) from sharding_two_node;
+-----+
| COUNT(*) |
+-----+
|      20 |
+-----+
1 row in set (0.01 sec)

mysql> show trace;
+-----+-----+-----+-----+-----+-----+
| OPERATION          | START(ms) | END(ms)   | DURATION(ms) | SHARDING_NODE | SQL/REF
+-----+-----+-----+-----+-----+-----+
| Read SQL          | 0.0       | 0.08553  | 0.08553      | -              | -
| Parse SQL         | 0.08553  | 0.56987  | 0.48434      | -              | -
| Try Route Calculation | 0.56987  | 0.71698  | 0.14711      | -              | -
| Try to Optimize   | 0.71698  | 1.237487 | 0.520507     | -              | -
| Execute SQL       | 1.237487 | 9.091029 | 7.853542     | dn1.0          | select COUNT(*) as `_$COUNT$_rpda_0` from `sharding_tw
o_node` LIMIT 100
| Fetch result      | 9.091029 | 10.186782 | 1.095753     | dn1.0          | select COUNT(*) as `_$COUNT$_rpda_0` from `sharding_tw
o_node` LIMIT 100
| Execute SQL       | 1.237487 | 8.348635 | 7.111148     | dn2.0          | select COUNT(*) as `_$COUNT$_rpda_0` from `sharding_tw
o_node` LIMIT 100
| Fetch result      | 8.348635 | 9.342241 | 0.993606     | dn2.0          | select COUNT(*) as `_$COUNT$_rpda_0` from `sharding_tw
o_node` LIMIT 100
| MERGE             | 8.721543 | 10.289905 | 1.568362     | merge.1        | dn1.0; dn2.0
| ORDERED_GROUP     | 8.726919 | 10.424309 | 1.69739      | ordered_group.1 | merge.1
| LIMIT             | 9.020162 | 10.499574 | 1.479412     | limit.1         | ordered_group.1
| SHUFFLE_FIELD     | 9.023584 | 10.501529 | 1.477945     | shuffle_field.1 | limit.1
| Write to Client   | 9.072457 | 11.52055 | 2.448093     | -              | -
| Over All          | 0.0       | 11.52055 | 11.52055     | -              | -
+-----+-----+-----+-----+-----+-----+
14 rows in set (0.03 sec)

```

7. 子查询的trace结果举例

```
mysql> select count(*) from sharding_two_node where id =(select id from sharding_two_node where id=1);
```

```
+-----+
| COUNT(*) |
+-----+
|      1 |
+-----+
1 row in set (0.03 sec)
```

```
mysql> show trace;
```

```
+-----+-----+-----+-----+-----+-----+
| OPERATION          | START(ms) | END(ms)   | DURATION(ms) | SHARDING_NODE | SQL/REF |
+-----+-----+-----+-----+-----+-----+
| Read SQL           | 0.0        | 0.063047 | 0.063047     | -              | -      |
| Parse SQL          | 0.063047  | 0.491182 | 0.428135     | -              | -      |
| Try Route Calculation | 0.491182  | 0.799576 | 0.308394     | -              | -      |
| Try to Optimize    | 0.799576  | 2.347412 | 1.547836     | -              | -      |
| Execute SQL        | 2.347412  | 11.183808 | 8.836396     | dn1.0          | select `sharding_two_node`.`id` as `autoalias_scalar`
` from `sharding_two_node` where id = 1 LIMIT 2 |
| Fetch result       | 11.183808 | 12.360691 | 1.176883     | dn1.0          | select `sharding_two_node`.`id` as `autoalias_scalar`
` from `sharding_two_node` where id = 1 LIMIT 2 |
| MERGE              | 11.889546 | 12.436445 | 0.546899     | merge.1        | dn1.0   |
| LIMIT              | 11.894923 | 12.483364 | 0.588441     | limit.1        | merge.1 |
| SHUFFLE_FIELD      | 11.896389 | 12.48483  | 0.588441     | shuffle_field.1 | limit.1 |
| SCALAR_SUB_QUERY   | 12.038123 | 12.485808 | 0.447685     | scalar_sub_query.1 | shuffle_field.1 |
| Generate New Query | 12.485808 | 13.824463 | 1.338655     | -              | -      |
| Execute SQL        | 13.824463 | 26.749647 | 12.925184    | dn1.1          | scalar_sub_query.1; select COUNT(*) as `$_COUNT$_rpd`
a_0` from `sharding_two_node` where sharding_two_node.id = 1 |
| Fetch result       | 26.685134 | 28.753476 | 2.068342     | dn1.1          | scalar_sub_query.1; select COUNT(*) as `$_COUNT$_rpd`
a_0` from `sharding_two_node` where sharding_two_node.id = 1 |
| MERGE              | 26.954918 | 29.091683 | 2.136765     | merge.2        | dn1.1   |
| ORDERED_GROUP      | 26.977889 | 29.563316 | 2.585427     | ordered_group.1 | merge.2 |
| SHUFFLE_FIELD      | 27.568285 | 29.567226 | 1.998941     | shuffle_field.2 | ordered_group.1 |
| Write to Client    | 27.72517  | 30.014911 | 2.289741     | -              | -      |
| Over All           | 0.0        | 30.014911 | 30.014911    | -              | -      |
+-----+-----+-----+-----+-----+-----+
18 rows in set (0.01 sec)
```

2.22 KILL @@DDL_LOCK

背景

在运维dble集群的过程中，有时会遇到 " There is other session is doing DDL " 或者 "xxx is doing DDL " 等问题，这些问题会导致某些表无法被操作，无法进行 reload 等。此时，需要人工干预保证集群运行正常。

逻辑判断

在 dble 集群中，可以通过 ZK 中某些键值对推断正在执行 DDL 的节点状态，比如通过键 `universe/dble/{cluster-id}/ddl/{schema.table}` 的值内容推断。该值理论上只会被发起者节点所修改，其他节点会订阅该键值，该值的内容为json格式，其中有一个 `status` 字段：

1. `status` 为 INIT 时，发起动作的节点会锁住本节点对应 `table` 的 `table meta`并获取当前key的分布式锁。其他的dble节点订阅到该值后会锁住本节点上对应 `table` 的 `table meta`；
2. 待发起节点执行DDL之后，会根据 DDL 执行结果更新当前键的值，此时根据 DDL 执行结果 `status` 分为两种情形：SUCCESS 和 FAILED。其他节点订阅到 `status` 为 SUCCESS 时，才会真正执行对应的DDL操作，否则会取消执行DDL并释放本节点对应`table` 的 `table meta` 锁；
3. 每个节点都会插入一条 `universe/dble/{cluster-id}/ddl/{schema.table}/{dble-id}:SUCCESS` 数据。

到此，除发起者的节点执行已经结束，但是发起者节点会等待其他节点响应，只有全部节点都汇报执行完成之后，当前ddl才算完成。

发起者节点通过什么来判断其他节点都响应完成？根据以下两个键的内容：

- `universe/dble/{cluster-id}/ddl/{schema.table}/{dble-id}`
- `universe/dble/{cluster-id}/online/下属结点`

所有执行当前DDL操作的节点都会在`universe/dble/{cluster-id}/ddl/{schema.table}/` 下面插入记录，发起节点通过判断 `online` 节点是否在 `universe/dble/{cluster-id}/ddl/{schema.table}/` 下面留下记录判断 DDL 是否被该节点所应用。只有两者相符合时，ddl才算真正意义上的完成。

此时，发起者会释放 `table meta` 锁 和 分布式锁，并且删除 `universe/dble/{cluster-id}/ddl/{schema.table}` kv 树。

kill ddl_lock

当前指令只会释放对应ddl在当前节点中所持有的元数据锁，但是不会影响执行该ddl的线程的状态。若出现上述问题，可以参考 上面如何判断其他节点都响应完成的方式找出哪些没有响应，在这些节点和发起节点上执行当前命令，并手动删除`universe/dble/{cluster-id}/ddl/{schema.table}` kv 树。

注意事项

1. 需要注意kill指令的执行顺序：先在各个从节点上执行该指令，最后在主节点上执行。
2. 如果不是特殊因素，此命令请不要随意使用。

2.23 外部高可用联动

- [2.23.1 外部后端MYSQL-HA连接](#)
- [2.23.2 命令的使用说明](#)
- [2.23.3 命令的实现细节](#)
- [2.23.4 简单的HA交互使用案例](#)

2.23.1 外部后端MYSQL-HA连接

dble在版本2.19.09.0开始提供对应的外部接口，基于以下几个基础要点进行设计：

- 外部HA组件能够通过一定的方式和dble进行信息的主动沟通（包括脚本，网络通信，命令等方式）
- 外部HA组件能够做出对与MYSQL可用性的明确判断（包括哪些节点停流量，哪些节点主从切换，哪些节点启动流量）

在基于以上的两个条件的情况下，基于提供外部接口通用性以及易用性的考虑，决定通过管理端用户命令的方式添加对应的交互接口，具体的提供以下的三个交互接口：

- 停止流量命令(dbGroup @@disable)
- 启动流量命令(dbGroup @@enable)
- 主从切换命令(dbGroup @@switch)

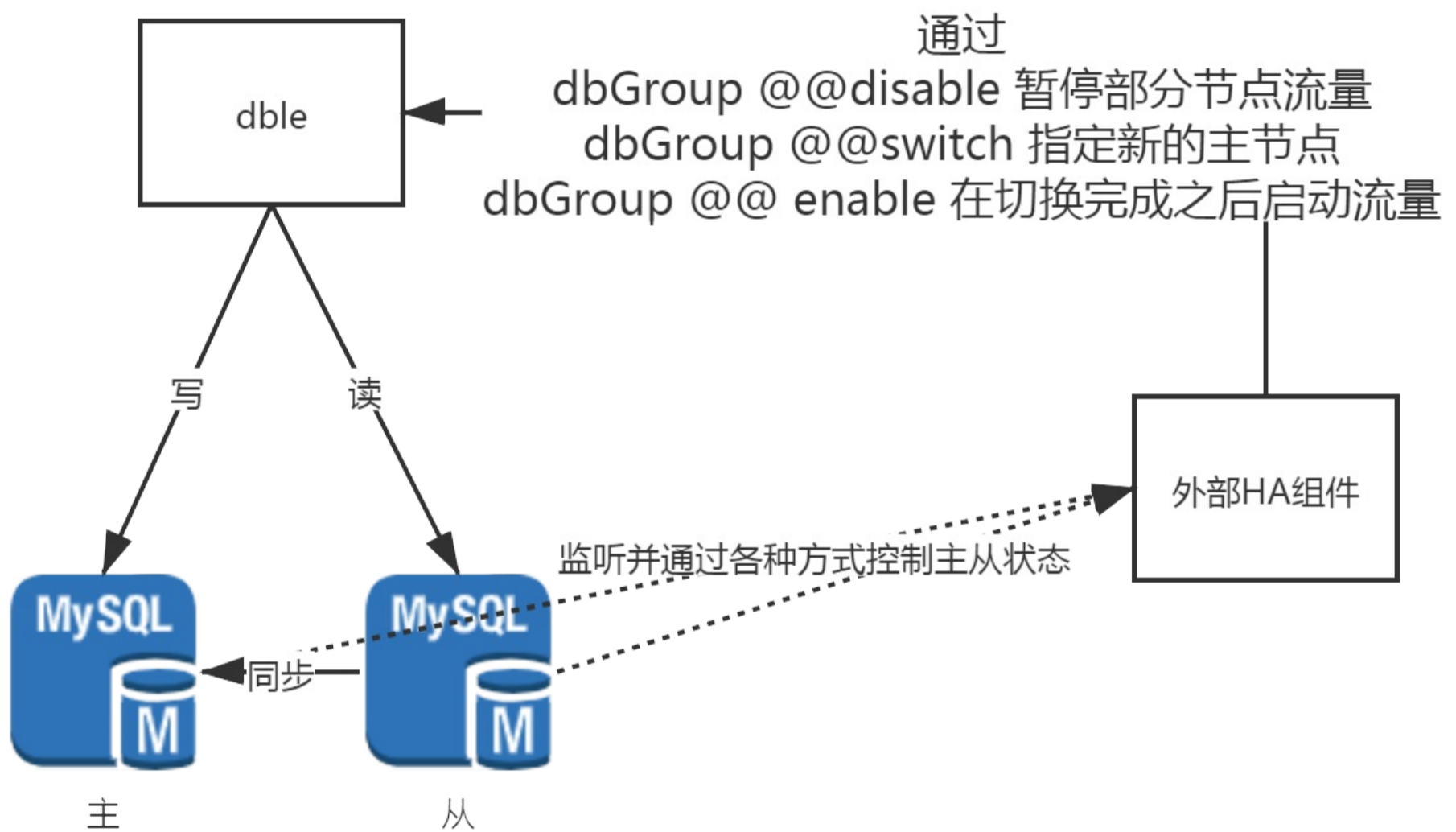
其次为了方便进行管理和观测，提供了一个当前切换状态的查询命令：

- 切换状态查询命令(dbGroup @@events)

外部HA组件可以通过 mysql命令、脚本调用mysql命令、或者程序的DB连接执行mysql命令的方式进行交互。

同时此功能设计考虑到zk集群用户的需求，当用户使用zk集群状态下的dble进行高可用切换时，会有集群的联动机制，外部HA组件只需要通知其中一个节点即可。

以下提供一个外部HA和dble交互设计图：



更加详细的情况请查询以下文档：

[命令的使用说明](#)

[命令的实现细节](#)

[简单的HA交互使用案例](#)

2.23.2 dbGroup命令的具体使用方法和解释

外部ha启用参数

当项目准备使用dble的外部ha联动方式来完成高可用切换或者同步的时候，需要提前进行如下配置

bootstrap.cnf 中的对应参数

```
-DuseOuterHa=true
```

注意:本参数的调整需要重启dble服务

cluster.cnf 的对应参数

```
needSyncHa = true
```

- 当此参数启用时，集群状态的dble将会在集群中同步自身的dbGroup的状态
- 此配置会强制将bootstrap.cnf 中的useOuterHa设置为true
- 当bootstrap.cnf 中的useOuterHa但cluster.cnf的needSyncHa不为true时，dble可以执行高可用切换的所有指令，但是其行为退化为单机dble，需要人工进行集群中多个dble的状态同步

注意：此参数的调整需要重启集群内的所有dble服务

关于dble中对于后端MySQL状态的解释

dbInstance的状态“disabled/enable”仅表示dble层面对于具体每个MySQL后端节点是否允许有流量的标识，和具体的MySQL存活状态无关

dbGroup @@disable

命令细节：

```
dbGroup @@disable name = 'dbGroup_name' [instance = 'instance_name']
```

- 其中的dbGroup_name指的是在db.xml中配置的dbGroup的名称，而instance_name指的是在一个dbInstance的name
- 当此命令不指定instance = '..'的内容时，默认将此dbGroup下所有dbInstance的状态置为disabled
- 被标记为disabled的节点无法提供正常的查询，即使对应的mysql真实的存活着
- 具体的dbInstance的disable状态可以通过管理端命令show @@dbInstance进行查询
- 若当前dble服务尚存连接被disable的连接，在命令执行过程中会被全部关闭，包括正在新建过程中的连接，可能会导致少量的查询报错
- 关闭连接的策略为对于使用中的连接先从池中删除，后续由定时任务（时间周期固定为5s）进行关闭；而空闲的连接则会直接关闭
- 由于连接关闭策略具有延迟性，可能会导致disable命令返回结果后，后端连接实际上还没有关闭

dbGroup @@enable

命令细节：

```
dbGroup @@enable name = 'dbGroup_name' [instance = 'instance_name']
```

- 其中的dbGroup_name指的是在db.xml中配置的dbGroup的名称，而instance_name指的是在一个dbInstance的name
- 当此命令不指定instance = '..'的内容时，默认将此dbGroup下所有dbInstance的状态置为enable

dbGroup @@switch

命令细节： dbGroup @@switch name = 'dbGroup_name' master = 'instance_name'

- 其中的dbGroup_name指的是在db.xml中配置的dbGroup的名称，而instance_name指的是在一个dbInstance的name
- 此命令name和master内容都为必填，在缺少任意元素的状态下会报错
- 此命令的作用会导致dbGroup下的标记为primary的dbInstance发生重置，也就是写节点会发生变化
- 此命令不会导致所有dbInstance节点的disable状态变化，但如果命令使得一个dbInstance从primary退化成为非primary，此dbInstance上的所有既有连接都会被关闭，以确保新的写请求不会被写入到错误的dbInstance上面去，这可能导致一些前端连接的查询报错和事务失败

2.23.3 高可用联动命令的逻辑细节

简述

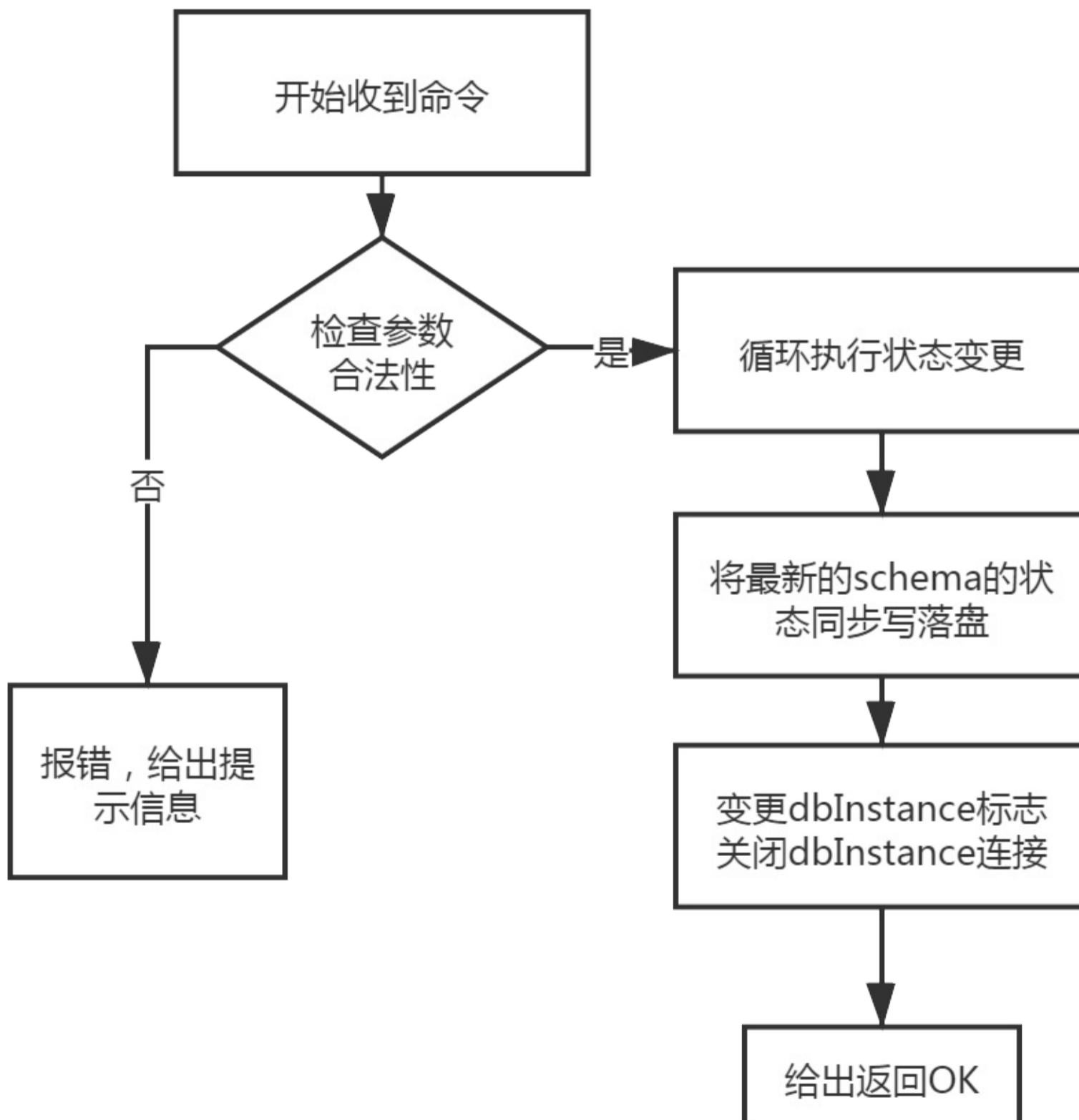
在dble高可用协同的几个接口中基本逻辑都是相似的，只是去更新dbGroup的属性，只不过在细节上每个命令都有一些自身的特殊行为

- disable命令在执行过程中会断开当前所有连接，并且在zk集群状态下会要求其他节点同步响应
- enable命令没有任何附加行为
- switch命令会在切换过程中断开旧primary dbInstance的所有连接

dbGroup @@disable

dble在单机情况下大致逻辑如下：

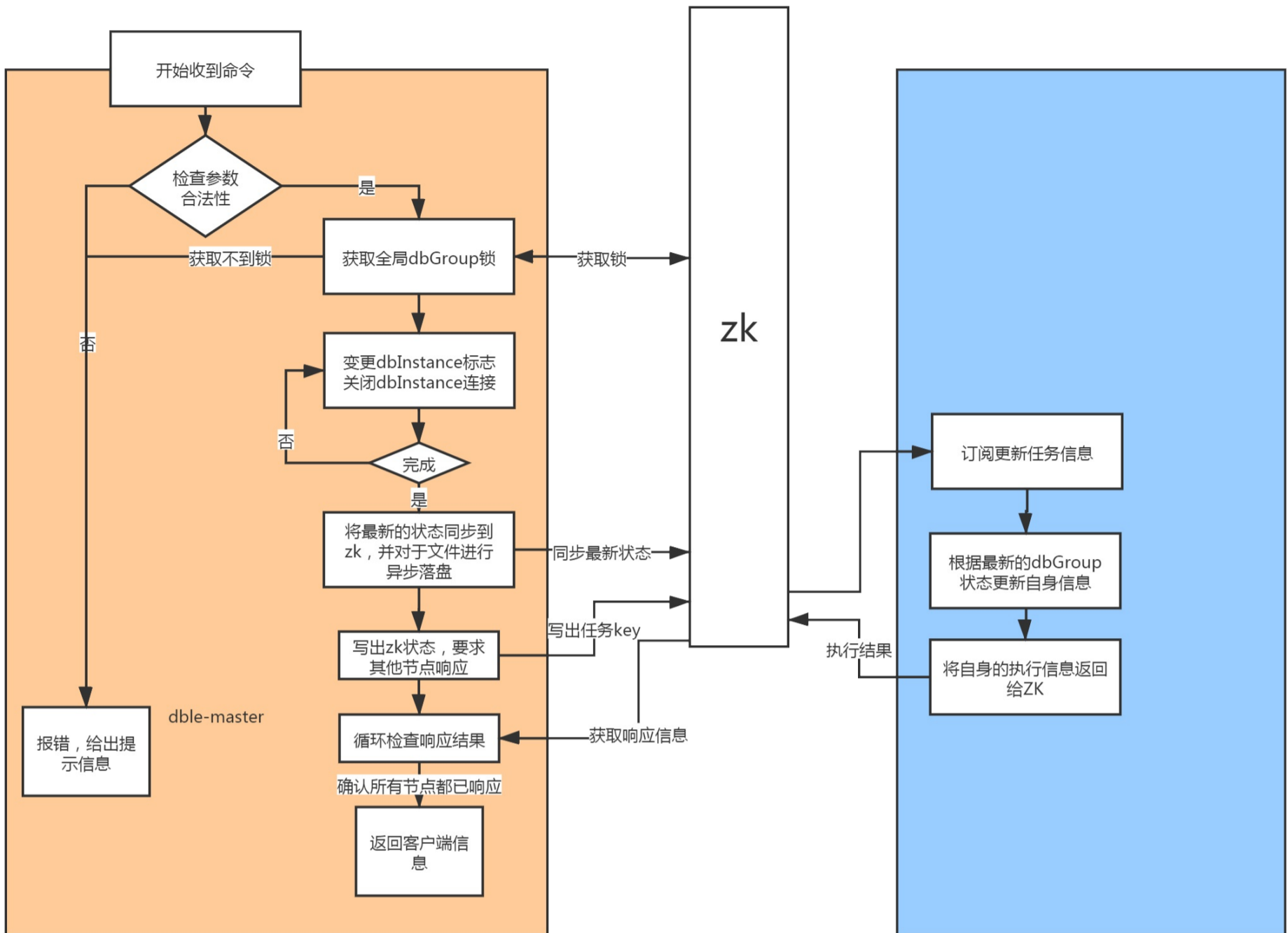
- 检查命令参数是否符合条件
- 更新dbInstance的状态
- 关闭所有已经存在的连接
- 同步更新配置文件，将dbGroup的最新状态落盘
- 返回OK信息



在集群状态下的逻辑有所不同：

- 检查命令参数是否符合条件
- 更新dbInstance的状态
- 关闭所有已经存在的连接
- 异步更新配置文件，将dbGroup的最新状态落盘

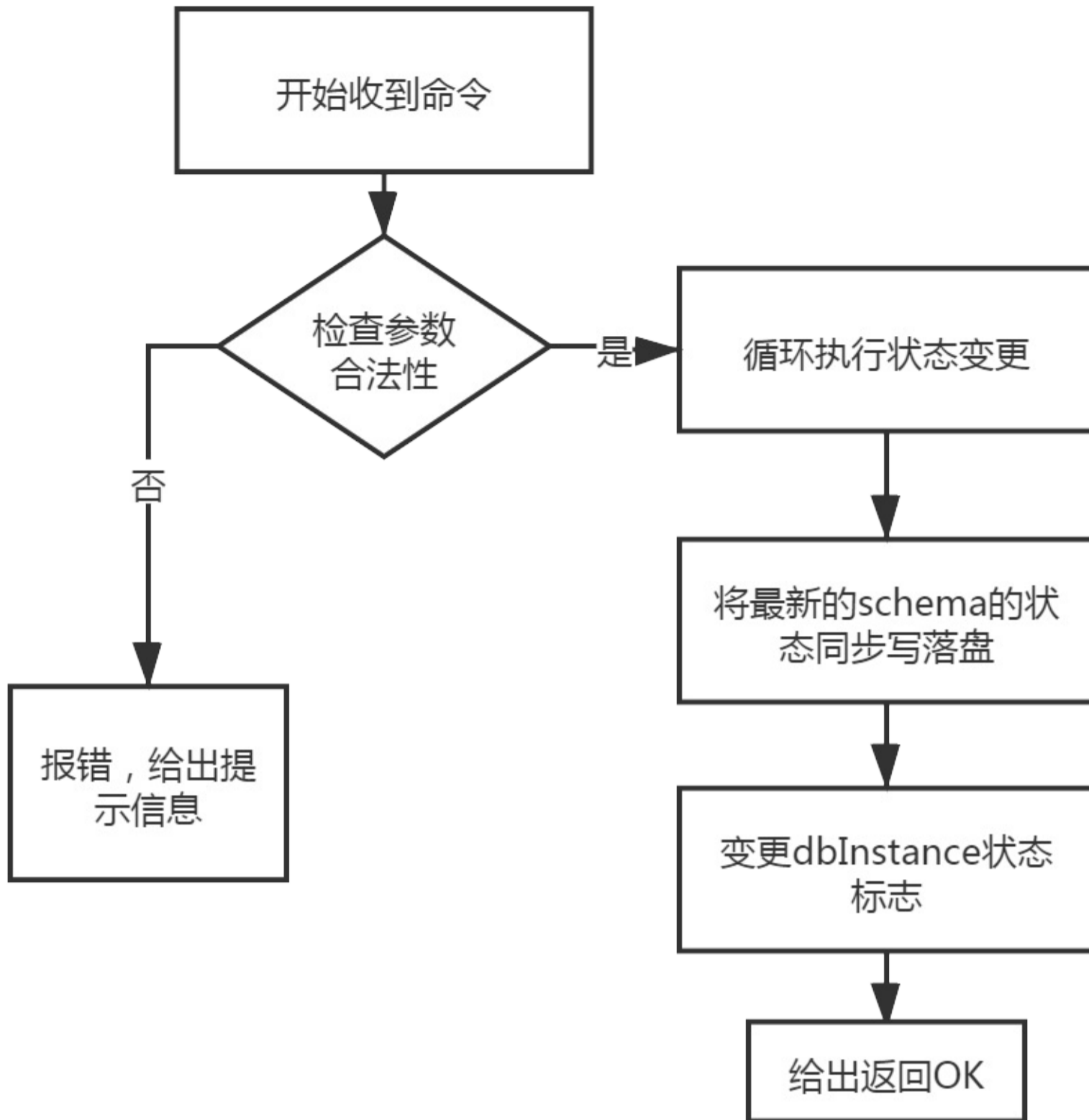
- 将最新的dbGroup的状态信息同步到zk上
- 写出一个zk上面的key任务，等待其他节点响应
- 其他节点响应任务，更新自身写出响应结果
- 发起dble检查所有响应结果，等待所有节点响应完成
- 返回最终的执行结果



dbGroup @@enable

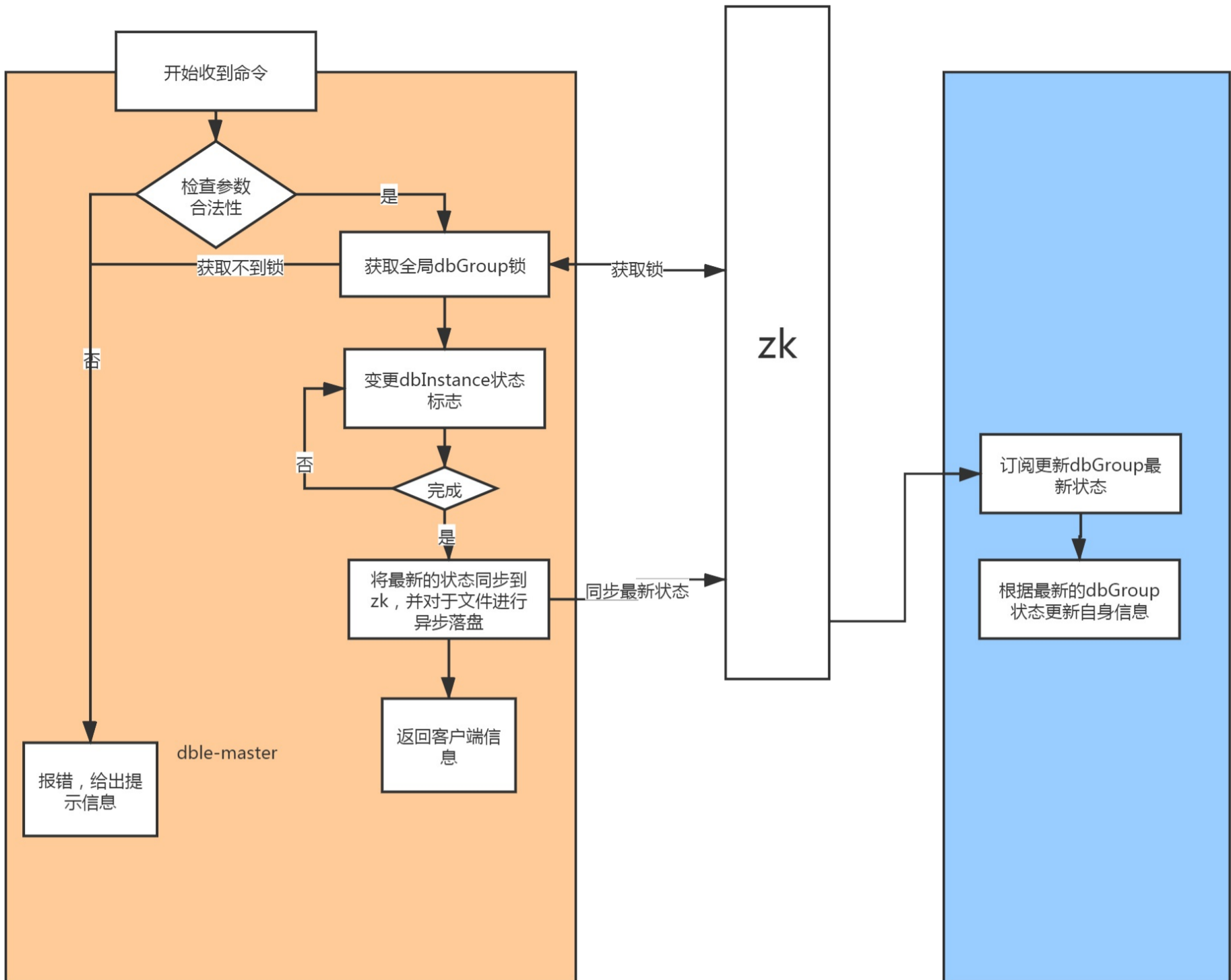
dble在单机情况下的逻辑大致如下:

- 检查命令参数是否符合条件
- 更新dbInstance的状态
- 同步更新配置文件, 将dbGroup的最新状态落盘
- 返回OK信息



在ZK集群的状况下逻辑大致如下：

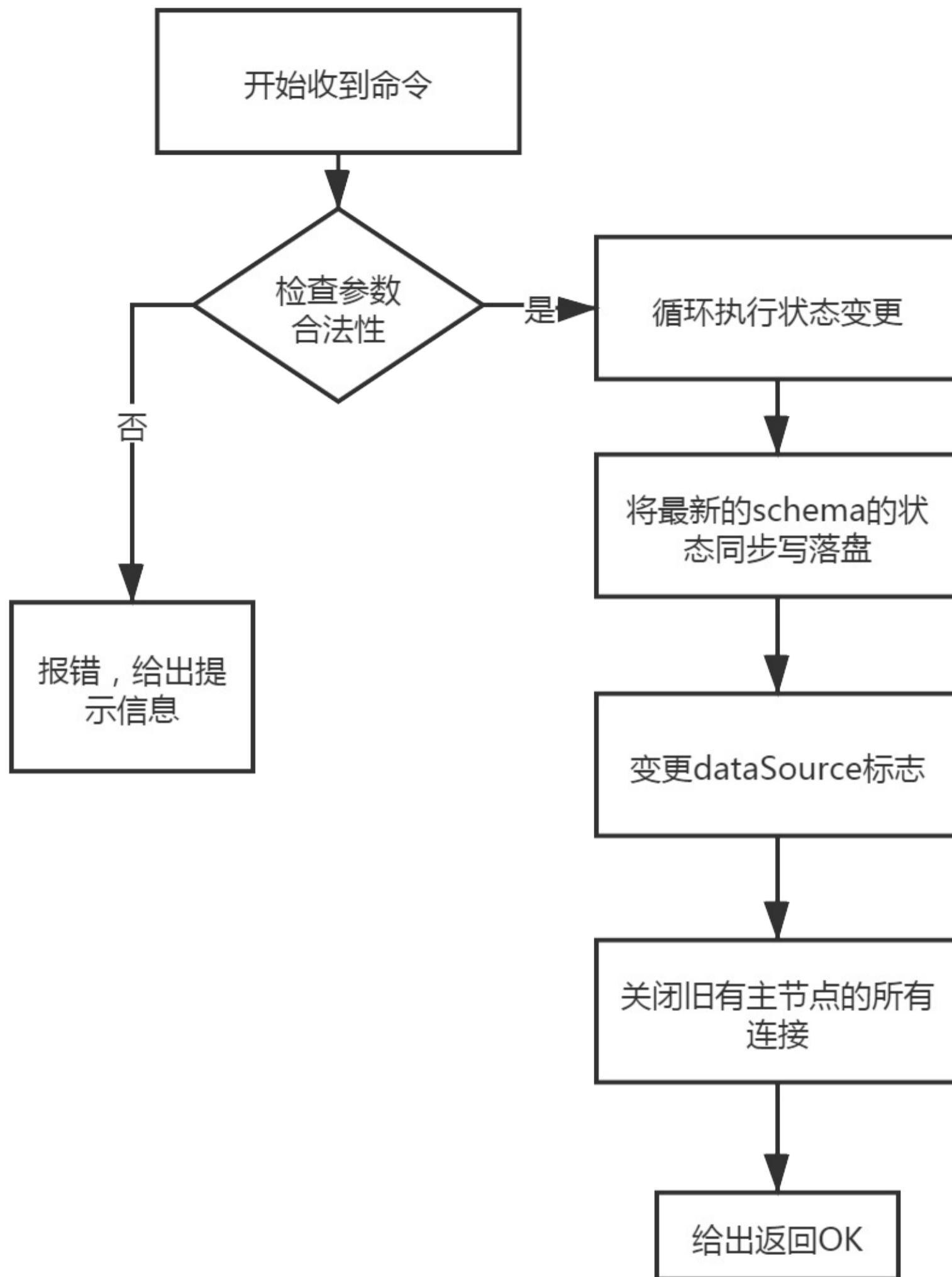
- 检查命令参数是否符合条件
- 更新dbInstance的状态
- 关闭所有已经存在的连接
- 异步更新配置文件，将dbGroup的最新状态落盘
- 将最新的dbGroup的状态信息同步到zk上
- 写出一个zk上面的key任务，等待其他节点响应
- 其他节点响应任务，更新自身写出响应结果
- 发起dble检查所有响应结果，等待所有节点响应完成
- 返回最终的执行结果



dbGroup @@switch

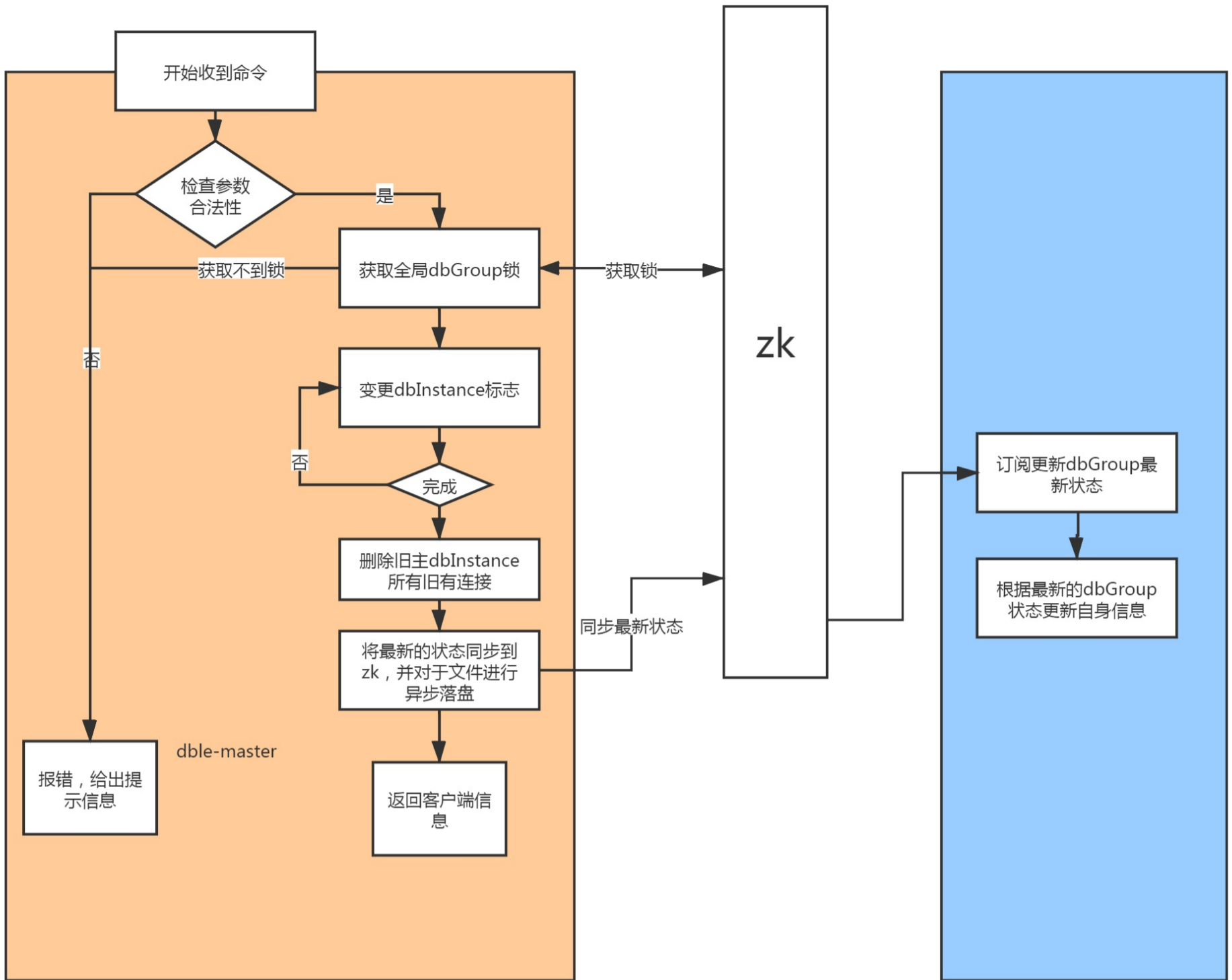
dble在单机情况下的其逻辑大致如下:

- 检查命令参数是否符合条件
- 更新dbInstance的状态
- 关闭旧primary dbInstance的所有连接
- 同步更新配置文件, 将dbGroup的最新状态落盘
- 返回OK信息



在ZK集群的状况下逻辑大致如下:

- 检查命令参数是否符合条件
- 更新dbInstance的状态
- 关闭所有已经存在的连接
- 异步更新配置文件, 将dbGroup的最新状态落盘
- 将最新的dbGroup的状态信息同步到zk上
- 写出一个zk上面的key任务, 等待其他节点响应
- 其他节点响应任务, 更新自身写出响应结果
- 发起dble检查所有响应结果, 等待所有节点响应完成
- 返回最终的执行结果

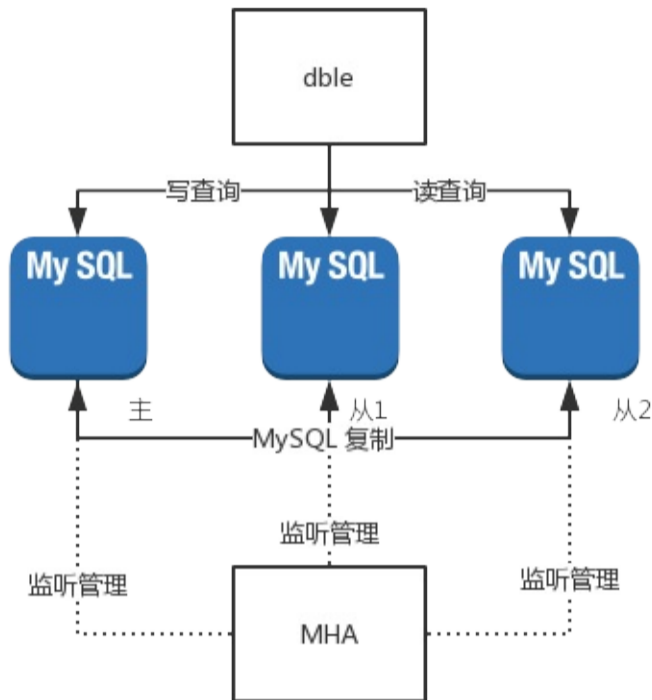


2.23.4 mha-dble高可用联动实例

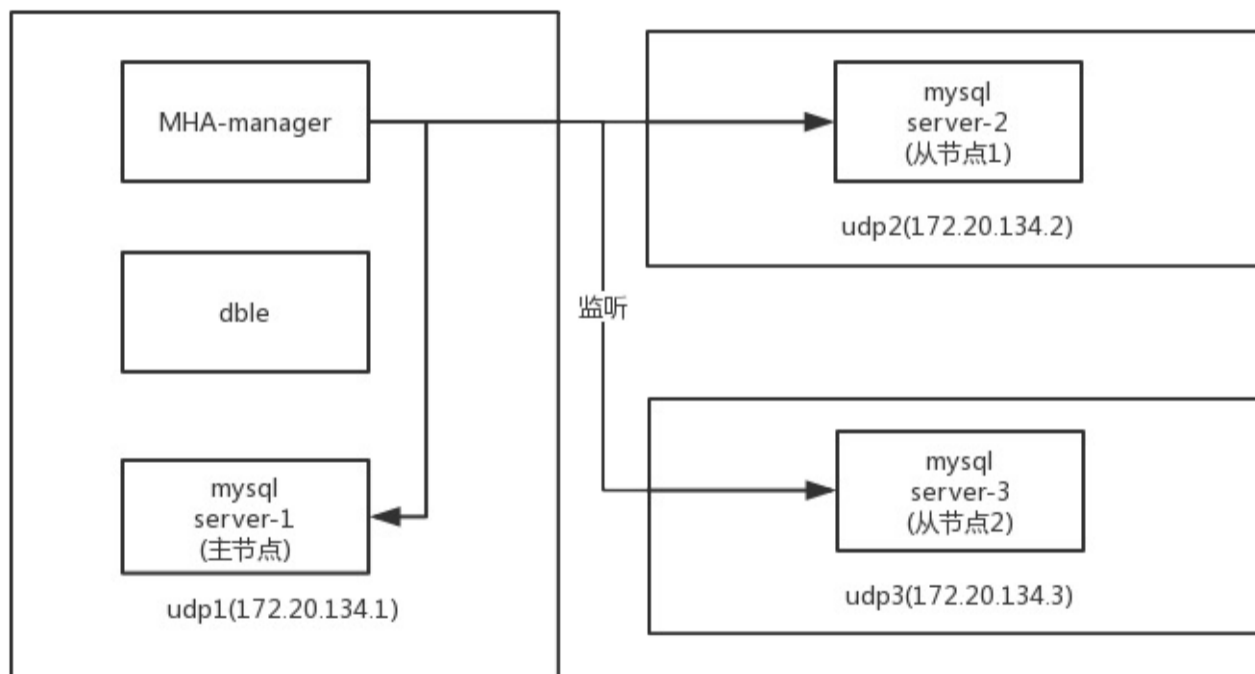
基础介绍

服务结构图

整体的服务结构如下图所示，dble将一个一组两从的MySQL视作一个读写分离的dbGroup，然后通过MHA对于此MySQL进行复制的管理：



下图是对于各个服务在每个docker容器上面的分布图：



实验过程

- 搭建一个MySQL主从(一主两从)的基础环境
- 搭建MHA监管MySQL复制组的环境
- 搭建dble环境使用这MySQL组进行启动
- 手动kill MySQL主实例让MHA托管的高可用组发生切换
- 验证dble和MySQL的高可用切换过程

实验目的

本实验原则上存在两个目的：

- 验证dble在2.19.09.0版本提供的高可用接口功能
- 搭建简单的MHA和dble高可用的环境范例

前期环境准备

前期准备环境：

三个docker容器，安装并部署mysql实例，并让三个mysql之间形成一主两从的复制关系(操作略)

操作过程

mha环境搭建

- 在每个docker容器中开启ssh服务

```
/usr/sbin/sshd -D
```

- 给每个容器之间创建免密登录,去到所有容器中执行，包括需要给自身容器创建

```
ssh-keygen -t rsa
ssh-copy-id -i ~/.ssh/id_rsa.pub root@other1
ssh-copy-id -i ~/.ssh/id_rsa.pub root@other2
ssh-copy-id -i ~/.ssh/id_rsa.pub root@self
```

- 确保/usr/bin/mysqlbinlog和/usr/bin/mysql文件存在，若不存在，请使用yum安装mysql即可
- 创建所有节点上的MHA工作目录

```
mkdir /etc/masterha/app1 -p
mkdir /var/log/masterha/app2 -p
mkdir /var/log/masterha/app1 -p
```

- 给所有mysql节点统一复制权限

```
grant all on *.* to root@%' identified by '123456' with grant option;
grant replication slave on *.* to repl@%' identified by 'repl';
```

- 下载并安装MHA的rpm包

```
从项目的文档页面能够直接找到对应的下载地址
https://github.com/yoshinorim/mha4mysql-manager/wiki/Downloads
下载一个MHA Manager 0.56 rpm RHEL6
以及一个MHA Node 0.56 rpm RHEL6
并将下载完成的rpm包上传到对应的容器中
注意使用yum localinstall 命令对于rpm包进行安装，这样在安装过程中可以由yum来进行依赖的处理
Node的包需要先于Manager的包安装，不然会有依赖方面的报错
在所有容器上安装完成这两个rpm
当yum无法正确安装rpm依赖，请更新yum源到阿里云的yum
```

切换脚本准备

mha这个项目是使用perl写的，同时mha这个项目给用户提供了大量的自定义插入接口，可以允许用户在一些关键步骤进行自定义的操作响应以及通知，在自动高可用切换过程中，mha提供了一个切换脚本的入口master_ip_failover_script脚本配置

详细的参数介绍可以参考官网的解释https://github.com/yoshinorim/mha4mysql-manager/wiki/Parameters#master_ip_failover_script

大体上来说这个脚本的三种命令会在以下几个时间节点得到调用

- 启动检查HA的状态，调用脚本master_ip_failover的status命令
- 发现MySQL master节点失效，调用master_ip_failover的stop命令，并输入失效master的信息
- 在新选择的master节点补偿数据完成，并进行写恢复设置read_only=0，调用master_ip_failover脚本的start命令

在本次实验中，通过自定义的脚本行为来进行mha和dblc之间的事件交互，所以我们对于原有的样例脚本进行以下的修改

- 原有MySQL master节点失效时，将dbGroup上面对应写节点的状态修改成disable，这要求在stop命令中对于dblc发送dbGroup @@disable命令
- 当新的MySQL master被选出来并且上线时，在脚本start阶段通过调用dbGroup @@switch命令将新的master节点切换成为dbGroup中的主节点

按照以上的逻辑对于修改完成的master_ip_failover脚本如下(注意给创建的脚步提供执行权限)

```
#!/usr/bin/env perl

# Copyright (C) 2011 DeNA Co.,Ltd.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
```

```

#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc.,
# 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

## Note: This is a sample script and is not complete. Modify the script based on your environment.

use strict;
use warnings FATAL => 'all';

use Getopt::Long;
use MHA::DBHelper;

my (
    $command,      $ssh_user,      $orig_master_host,
    $orig_master_ip, $orig_master_port, $new_master_host,
    $new_master_ip, $new_master_port, $new_master_user,
    $new_master_password
);

GetOptions(
    'command=s'          => \$command,
    'ssh_user=s'         => \$ssh_user,
    'orig_master_host=s' => \$orig_master_host,
    'orig_master_ip=s'  => \$orig_master_ip,
    'orig_master_port=i' => \$orig_master_port,
    'new_master_host=s' => \$new_master_host,
    'new_master_ip=s'   => \$new_master_ip,
    'new_master_port=i' => \$new_master_port,
    'new_master_user=s' => \$new_master_user,
    'new_master_password=s' => \$new_master_password,
);

exit &main();

sub main {
    if ( $command eq "stop" || $command eq "stopssh" ) {

        # $orig_master_host, $orig_master_ip, $orig_master_port are passed.
        # If you manage master ip address at global catalog database,
        # invalidate orig_master_ip here.
        my $exit_code = 1;
        eval {

            # dbGroup @@disable name = "dbGroup1" instance='$orig_master_host'
            # 调用对应的disable命令,使得部分节点不可写
            $orig_master_host =~ tr/./_/;
            system "mysql -P9066 -u man1 -p654321 -h 172.20.134.1 -e \"dbGroup \@@@disable name = 'dbGroup1' instance='\".\$orig_master_host.\"'\";
        };

        $exit_code = 0;
    };
    if ($?) {
        warn "Got Error: $@\n";
        exit $exit_code;
    }
    exit $exit_code;
}
elseif ( $command eq "start" ) {

    # all arguments are passed.
    # If you manage master ip address at global catalog database,
    # activate new_master_ip here.
    # You can also grant write access (create user, set read_only=0, etc) here.
    my $exit_code = 10;
    eval {
        my $new_master_handler = new MHA::DBHelper();

        # args: hostname, port, user, password, raise_error_or_not
        $new_master_handler->connect( $new_master_ip, $new_master_port,
            $new_master_user, $new_master_password, 1 );

        ## Set read_only=0 on the new master
        $new_master_handler->disable_log_bin_local();
        print "Set read_only=0 on the new master.\n";
        $new_master_handler->disable_read_only();

        ## Creating an app user on the new master
        print "Creating app user on the new master..\n";
    };
}

```

```
$new_master_handler->enable_log_bin_local();
$new_master_handler->disconnect();

## try to switch the dbGroup master into new master
## 调用dbGroup switch的命令, 将新的new_master_host节点提升
$new_master_host =-tr/./_/;
system "mysql -P9066 -u man1 -p654321 -h 172.20.134.1 -e \"dbGroup \@switch name = 'dbGroup1' master='\".$new_master_host.\"'\";

$exit_code = 0;
};
if ($?) {
    warn $?;

    # If you want to continue failover, exit 10.
    exit $exit_code;
}
exit $exit_code;
}
elseif ( $command eq "status" ) {
    # test for start command
    exit 0;
}
else {
    &usage();
    exit 1;
}
}

sub usage {
    print
    "Usage: master_ip_failover --command=start|stop|stopssh|status --orig_master_host=host --orig_master_ip=ip --orig_master_port=port --new_m
aster_host=host --new_master_ip=ip --new_master_port=port\n";
}
}
```

将脚本存放到指定目录/etc/masterha/app1目录下

并且创建MHA最终使用的 app1.conf配置文件在/etc/masterha/app1目录, 具体创建的配置文件内容如下所示:

```

#mha manager工作目录
manager_workdir = /var/log/masterha/app1
manager_log = /var/log/masterha/app1/app1.log
remote_workdir = /var/log/masterha/app2
master_ip_failover_script=/etc/masterha/app1/master_ip_failover
# master_ip_online_change_script=/etc/masterha/app1/master_ip_online_change
# MySQL管理帐号和密码
user=root
password=123456
# 系统ssh用户
ssh_user=root

# 复制帐号和密码
repl_user=repl
repl_password= repl

# 监控间隔(秒)
ping_interval=1
manager_log=/var/log/masterha/app1/manager.log

[server1]
hostname=172.20.134.1
master_binlog_dir = /opt/3306/
port=3306

[server2]
# 每个机器上面的mysql实例的信息
hostname=172.20.134.2
master_binlog_dir = /opt/3306/
candidate_master=1
check_repl_delay=0
port=3306

[server3]
# 每个机器上面的mysql实例的信息
hostname=172.20.134.3
master_binlog_dir = /opt/3306/
candidate_master=1
check_repl_delay=0
port=3306

```

最终通过以下命令对于MHA的监听线程进行启动

```
nohup masterha_manager --conf=/etc/masterha/app1/app1.conf >> /var/log/masterha/app1/manager.log 2>&1 &
```

dble配置

配置dble在这里是比较简单的内容，我们仅使用默认的配置即可，从项目的release页面下载最新的2.19.09.0安装包，解压并将conf文件下的文件配置好

```

mv cluster_template.cnf cluster.cnf
mv bootstrap_template.cnf bootstrap.cnf
mv db_template.xml db.xml
mv user_template.xml user.xml
mv sharding_template.xml sharding.xml

```

并按照当前的配置需求对于其中的db.xml以进行如下的配置调整

db.xml

```

<dbGroup name="dbGroup1" delayThreshold="10000">
  <heartbeat>show slave status</heartbeat>
  <dbInstance name="172_20_134_1" url="172.20.134.1:3306" password="123456" user="root" disabled="false" id="udp-1" primary="true" /
>
  <dbInstance name="172_20_134_2" url="172.20.134.2:3306" password="123456" user="root" disabled="false" id="udp-3" />
  <dbInstance name="172_20_134_3" url="172.20.134.3:3306" password="123456" user="root" disabled="false" id="udp-2" />
</dbGroup>

```

启动dble，并通过管理用户man1查看基础状态下的dble后端节点状态

```
MySQL [(none)]> show @@dbinstance;
```

DB_GROUP	NAME	HOST	PORT	W/R	ACTIVE	IDLE	SIZE	EXECUTE	READ_LOAD	WRITE_LOAD	DISABLED
dbGroup1	172_20_134_1	172.20.134.1	3306	W	1	0	1000	1	0	0	false
dbGroup1	172_20_134_3	172.20.134.3	3306	R	1	0	1000	0	0	0	false
dbGroup1	172_20_134_2	172.20.134.2	3306	R	1	0	1000	0	0	0	false

可见在初始状态下，节点的使用属性和配置文件一致，172.20.134.1节点作为写节点，并且所有节点的使用状态都是“可用”

最终效果

- 通过ps命令在172.20.134.1容器中找到对应的mysqld进程
- 通过kill -9 命令对于172.20.134.1容器中的mysqld进程进行关闭
- 重新通过命令检查dble中此时的后端节点状态，发现如下的执行结果

```
MySQL [(none)]> show @@dbinstance;
```

DB_GROUP	NAME	HOST	PORT	W/R	ACTIVE	IDLE	SIZE	EXECUTE	READ_LOAD	WRITE_LOAD	DISABLED
dbGroup1	172_20_134_2	172.20.134.2	3306	W	1	0	1000	0	0	0	false
dbGroup1	172_20_134_3	172.20.134.3	3306	R	1	0	1000	0	0	0	false
dbGroup1	172_20_134_1	172.20.134.1	3306	R	0	0	1000	0	0	0	true

- 检查当前dble的配置文件，发现配置文件中的后端节点位置发生扭转,172_20_134_2变成新的写节点

```
<dbGroup name="dbGroup1" delayThreshold="10000">
  <heartbeat>show slave status</heartbeat>
  <dbInstance name="172_20_134_1" url="172.20.134.1:3306" password="123456" user="root" disabled="false" id="udp-1" />
  <dbInstance name="172_20_134_2" url="172.20.134.2:3306" password="123456" user="root" disabled="false" id="udp-3" />
  <dbInstance name="172_20_134_3" url="172.20.134.3:3306" password="123456" user="root" disabled="false" id="udp-2" primary="true"
/>
</dbGroup>
```

- 检查存在于172.20.134.3上mysql从的状态，发现其复制指向已经发生切换，切换为新主172.20.134.2，与dble中最新的配置文件相符

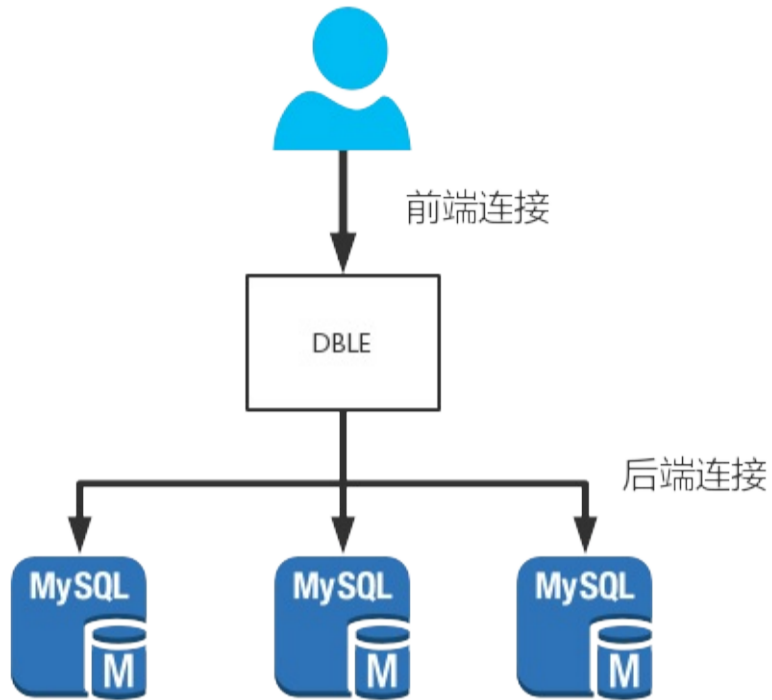
```
MySQL [(none)]> show slave status\G
```

```
***** 1. row *****
Slave_IO_State: Waiting for master to send event
Master_Host: 172.20.134.2
Master_User: repl
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: mysql-bin.000001
Read_Master_Log_Pos: 2113
Relay_Log_File: udp3-relay-bin.000002
Relay_Log_Pos: 320
Relay_Master_Log_File: mysql-bin.000001
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
.....
```

2.24 超时（连接/执行）控制

概述

在dble中连接分为两种，前后端连接，分别对应客户端连接dble的网络连接和dble中连接mysql的网络连接大致的情況如下图所示：



由于网络TCP连接的特性，dble中需要对于这两种连接都有超时控制，具体在dble中将连接的超时控制分为：**前端连接无响应超时**、**后端连接空闲超时**和**后端连接执行超时**；当超时发生的时候，dble会通过切断超时连接的方法进行控制。

超时的检查和实现

在配置文件bootstrap.cnf中存在三个配置项

- sqlExecuteTimeout(后端连接执行超时)
- idleTimeout（前端连接无响应超时）
- processorCheckPeriod（超时检查周期）

以下描述具体超时检查的实现逻辑

- dble按照配置的processorCheckPeriod作为周期去周期性检查所有的连接
- 检查所有后端连接，若满足以下条件就关闭后端连接
 - 此后端连接已经被某个前端连接借走（正在执行或持有）
 - 此后端连接距离上一次收发包超过sqlExecuteTimeout
 - 此后端连接没有在执行DDL或者执行xa事务
- 检查所有前端连接，若满足以下条件就关闭前端连接
 - 此前端连接正在执行xa事务，并不处于commit失败补偿/rollback失败补偿
 - 此前端连接距离上一次收发包超过idleTimeout

综上所述为超时关闭连接的逻辑可以描述为：

- 后端连接非DDL,XA事务的情况下被前端连接借走，并且超过sqlExecuteTimeout没有收发包（包括普通事务执行间隔，后端连接被长期持有的时间超时）
- 前端连接非XA事务特殊阶段，超过idleTimeout没有收发包（包括在loaddata大文件过程中发生的长时间空档）

SQL执行超时注意事项

- 部分语言或者框架的连接池会长期持有连接，并且没有设定或者定时发送网络包，可能会导致空闲的连接自动断开，然后应用在取用到对应断开连接的时候可能会报错
- 当开启一个事务但是长期不执行SQL会导致后端连接被判断为执行超时，当应用再次在连接上执行内容的时候，会给出后端连接已关闭的报错信息
- 当前端连接执行一个大文件内容load data的时候，由于后端可能执行较慢，前端连接存在超过idleTimeout导致连接关闭的情况，当有类似数据导入计划的时候，考虑暂时放宽idleTimeout的限制
- 连接的超时检查和关闭是通过一个循环来实现的，所以并不是一个实时检查的值，最坏的情况对于超时的检查会迟到一个processorCheckPeriod周，当应用对于SQL超时需要一个严格的时间设定时，请谨慎使用

2.25 dble流量控制

背景

在之前的dble版本中，当进行大文件load以及大结果集查询的过程中，都有可能由于数据的发送不及时造成数据在dble内存中堆积，当条件足够的时候甚至有可能造成dble服务的OOM，进而影响服务的稳定运行。

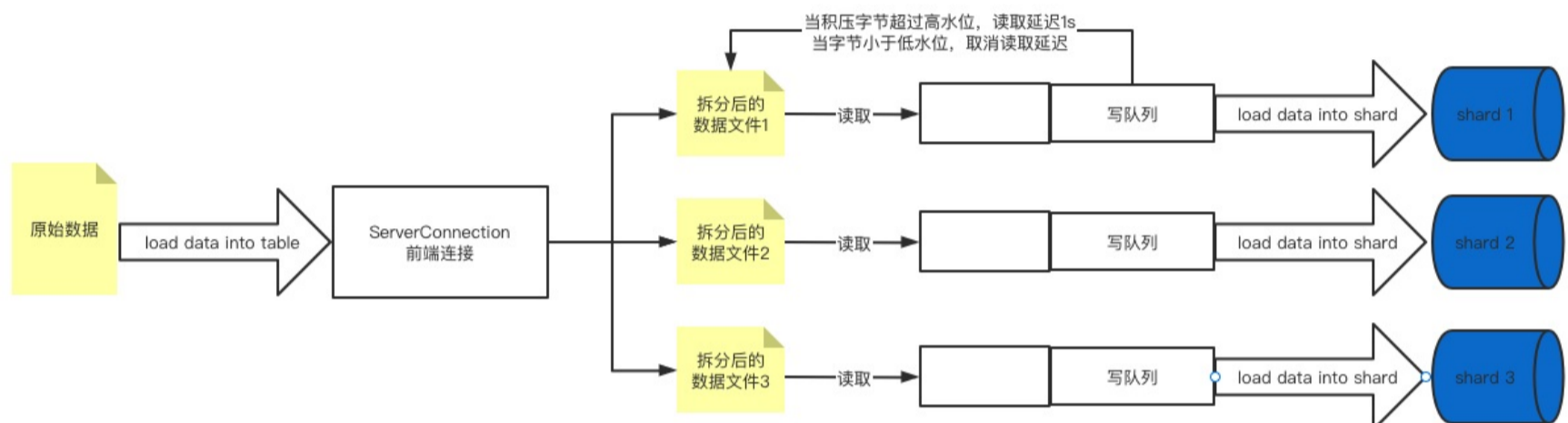
在最初2.20.04中由社区开发者@ssxlulu提供了对于这部分流量控制的实现，通过连接级别的写队列长度，进行数据加载/获取的负反馈调节，从而实现在数据load和大结果集查询过程中的内存使用情况稳定，在此基础上，我们又做了一些改进。

原理

dble中的流量控制通过连接级别的写出队列进行负反馈调节，具体在生效的时候分成两种具体的形式：

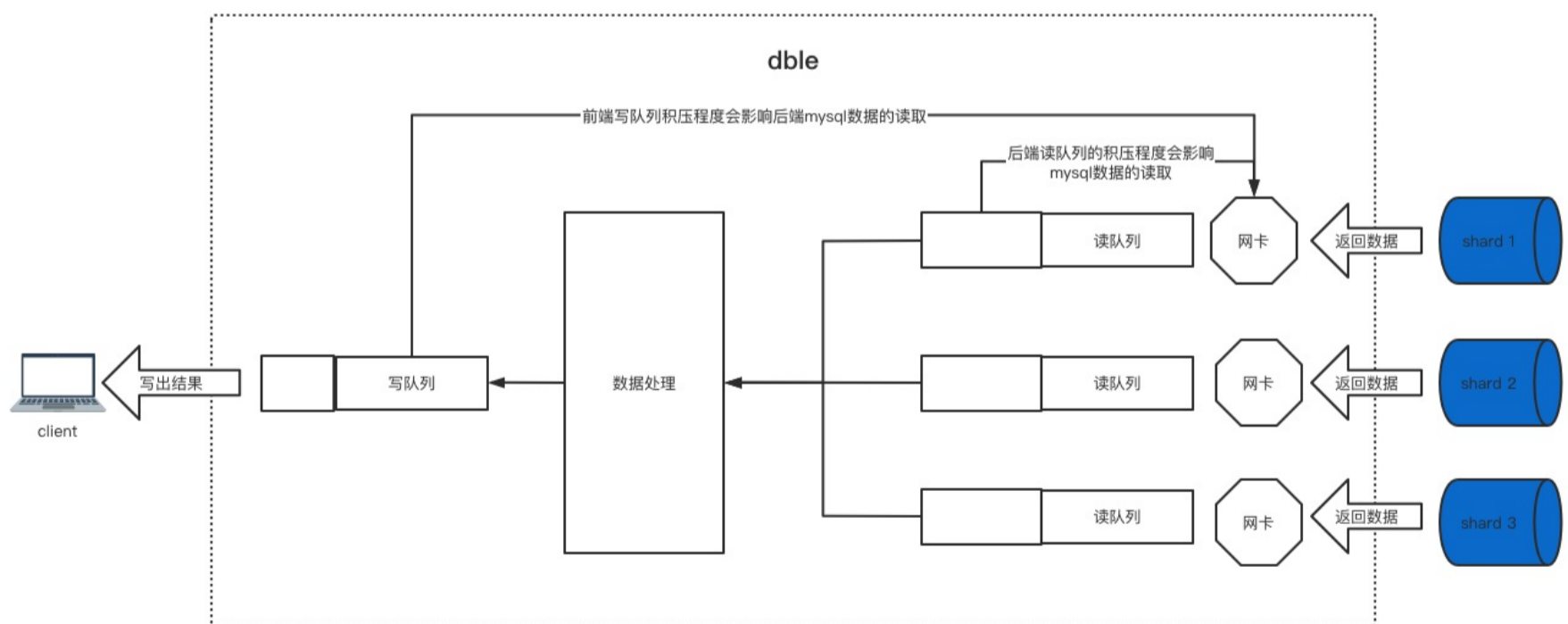
• Load data过程：

- 后端连接写出队列积压字节数过大，大于后端链接的高水位(flowHighLevel)时，写出线程暂停，等积压字节被写出，降低到低水位(flowLowLevel)后恢复。逻辑示意图如下：



• Select过程：

- 前端连接写出队列积压字节数过大(比如中间件和客户端之间网络不好可能会造成此现象), 大于高水位(flowControlHighLevel), 后端MySQL连接开始流量控制, 停止读数据
- 当MySQL已读取的数据队列积压, 积压的字节数过大(比如中间件和mysql在同一机器上), 大于后端连接的高水位(flowHighLevel), 后端mysql连接开始流量控制, 停止读数据
- 当前端连接写出队列积压字节数小于低水位(flowControlLowLevel)并且后端连接读取积压字节数小于低水位(flowLowLevel), 后端mysql连接停止流量控制, 开始读数据
- 逻辑示意图如下：



注：流量控制功能的生效级别为连接级别，不同连接之间的队列长度不会互相影响,另外只有在nio下此功能才有效

使用举例

本功能为默认关闭功能，需要在bootstrap.cnf中进行相关配置进行显式开启，或者通过管理端的辅助命令进行实时的调整。在bootstrap.cnf中使用下列参数使得功能开启并生效：


```
enableFlowControl(是否开启流量控制开关)
flowControlHighLevel(写队列上限阈值, 写队列超限时开启流量控制)
flowControlLowLevel(写队列下限阈值, 写队列低于阈值时取消流量控制)
```

- 修改参数的管理命令为:
`flow_control @@set [enableFlowControl = true/false] [flowControlHighLevel = ?] [flowControlLowLevel = ?]`
- 后端属性flowLowLevel和flowHighLevel在连接池属性处配置, 可以通过重载命令等来配置或者修改, 开关也由bootstrap.cnf的enableFlowControl控制。
- 展示当前流控配置参数信息:
 - `flow_control @@show .`
- 展示连接是否正在被流量控制:
 - `flow_control @@list`
 - 如果需要过滤结果集, 可以使用 `dbls_information.dble_flow_control` (与 `flow_control @@list` 等价) 来过滤

2.26 开启/关闭client_found_rows权能标志

2.26.1 介绍&背景

在客户端与服务器在初次连接的时候，服务端发送初始化握手包时会带上自己所支持的**权能标志**；客户端接收后会对服务器发送的权能标志进行筛选，保留自身所支持权能标志且返回给服务器，从而保证服务器与客户端通讯的兼容性。因此权能标志是在初次连接确定，不能动态修改。

在DBLE中后端连接都是使用连接池预先建立好的，导致与前端请求不同导致行为不一样；因此在dble管理端中新增了对client_found_rows权能标志更改和后端连接池的刷新。

2.26.1.1 client_found_rows的作用

若初次连接handshake协议中启用client_found_rows权能标志，表示在DML等操作时结果集里返回发现行(found rows)，而不是影响行(affect rows)。

2.26.1.2 client_found_rows值设定：

MYSQL客户端：默认关闭client_found_row；（则返回结果集里为affect rows）

JDBC：默认开启client_found_rows；（则返回结果集里为found rows）

2.26.1.3 JDBC中useAffectedRows与client_found_row的关系

useAffectedRows=true 即关闭client_found_rows

useAffectedRows=false(默认) 即开启client_found_rows

2.26.2 具体

2.26.2.1 在bootstrap.cnf里增加参数

```
#是否开启client_found_rows权能标志，默认关闭
-DcapClientFoundRows=false
```

2.26.2.2 管理端口增加命令

```
show @@cap_client_found_rows; -- 查询client_found_row权能标志开启状态 0-关闭 1-开启
disable @@cap_client_found_rows; -- 关闭client_found_row权能标志
enable @@cap_client_found_rows; -- 开启client_found_row权能标志
```

注意：如果在不停dble服务的情况下，更改该权能标志，为了保证与后端连接的mysql的该权能标志一致，(强调)需要[刷新连接池](#)；否则insert的结果集不正确

2.26.3 验证

step1. 启动dble(默认关闭client_found_rows)

step2. 管理用户身份开启client_found_rows权能标志

```
mysql -uman1 -h192.xx.xx.xx -P9066 -p654321
enable @@cap_client_found_rows;
```

step3. 用户身份尝试登录

- 在3.20.10.0版本中，登录失败，提示客户端与dble内存中的client_found_rows权能标志不一致

```
mysql -uroot -h192.xx.xx.xx -P8066 -p123456
ERROR 1045 (HY000): The client requested CLIENT_FOUND_ROWS capabilities does not match, in the manager use show @@cap_client_found_rows check latest status.
```

- 在3.20.10.1及后续版本，登录成功，但会在dble.log中提示客户端与dble内存中的client_found_rows权能标志不一致，此时返回结果以dble的client_found_rows值为准

```
the client requested CLIENT_FOUND_ROWS capabilities is 'found rows', dble is configured as 'affect rows',pls set the same.
or
the client requested CLIENT_FOUND_ROWS capabilities is 'affect rows', dble is configured as 'found rows',pls set the same.
```

2.27 general日志

2.27.1 介绍

开启general日志会将所有到达dble的sql语句以(仅支持)file方式记录；开启后性能损耗在3%~5%，不需要观察下发sql情况时建议关闭该功能

已知问题：

- 1.执行Execute包时，dble接收的long data数据将以16进制形式打印在general日志中
- 2.读写分离用户，执行Execute包，不将sql打印出来（原因；读写分离实际相当于sql的中转站，中途不做任何数据的保留）

2.27.2 bootstrap.cnf中general log相关配置

```
# dble存放各种文件的父目录，假设当前路径为/tmp/
-DhomePath=.

# 开启general log，默认关闭；0-关闭，1-开启
#-DenableGeneralLog=1

# general log文件的路径，默认general/general.log；若设置以 '/' 开头的值则作为绝对路径生效，反之，则在homepath后拼接值得到最终绝对路径；比如：
# 若设值为general/general.log，则最终文件绝对路径为：/tmp/general/general.log
# 若设值为/general/general.log，最终文件绝对路径为：/general/general.log
#-DgeneralLogFile=general/general.log

# 触发翻转的文件大小，默认16，以mb为单位；当超过16MB则将general.log翻转为yyy-MM/general-MM-dd-%d.log的格式文件
#-DgeneralLogFileSize=16

# 内部实现机制用到的队列大小，值必须为2的次方，默认4096
#-DgeneralLogQueueSize=4096
```

2.27.3 管理端命令

2.27.3.1 show @@general_log

查询general的开关和路径信息；另外也可以使用use dble_information; select * from dble_variables where variable_name like '%general%';

```
show @@general_log;
+-----+-----+
| NAME          | VALUE          |
+-----+-----+
| general_log   | ON             |
| general_log_file | /tmp/./general/general.log |
+-----+-----+
2 rows in set (0.03 sec)
```

2.27.3.2 disable @@general_log

关闭general log

```
disable @@general_log;
Query OK, 1 row affected (0.02 sec)
disable general_log success
```

2.27.3.3 enable @@general_log

开启general log

```
enable @@general_log;
Query OK, 1 row affected (0.02 sec)
enable general_log success
```

2.27.3.4 reload @@general_log_file=?

重置general log的文件路径

```
reload @@general_log_file='/tmp/dble-general/general/general.log';
Query OK, 1 row affected (0.00 sec)
reload general_log path success
```

2.28 sql统计

2.28.1 介绍

- 指标：统计dble中的事务、后端节点执行sql的(CRUD)次数、耗时、以及返回的行数(或影响行数)
- 维度：业务端下发的sql、dble内部下发至后端节点的sql
- 性能：
 - 单独开启 `-DsamplingRate=100` (默认), 性能损耗: 0%~15%(根据1000w的数据量进行单点select压测得出)。
 - 开启所有(`-DenableStatisticAnalysis=1` 、 `-DenableStatisticAnalysis=1` 、 `-DsamplingRate=100`), 性能损耗: 0%~30%(根据1000w的数据量进行单点select压测得出)。
 - 在不同场景下, 比如表的分片数越多、返回结果集行数越多等场景, 都会加大性能损耗。
 - 注意, 在开启 `-DenableStatisticAnalysis=1` 使用读写分离用户比分表分库用户压测结果看, 损耗更大; 因为读写分离直接将sql透穿下发后侧, 而show @@sql.sum.table中是对table聚合统计, 需要提取sql中table, 而提取过程中是个比较耗时的操作, 因此性能损耗会增加。
- 可视化建议: 将数据吐给类似prometheus的第三方监控工具, 这样比直接使用dble统计表格更加直观

2.28.2 bootstrap.cnf中sql统计的相关配置

```
# 开启statistic的开关, 默认关闭; 0-关闭, 1-开启
#-DenableStatistic=0

# 开启show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition聚合统计, 默认开启; 0-关闭, 1-开启
#-DenableStatisticAnalysis=0

# 统计表的大小, 默认1024
#-DassociateTablesByEntryByUserTableSize=1024
#-DfrontendByBackendByEntryByUserTableSize=1024
#-DtableByUserByEntryTableSize=1024

# 内部实现机制用到的队列大小, 值必须为2的次方, 默认4096
#-DstatisticQueueSize=4096

# 采样率, 默认为100即全采样, 采样率是[0, 100]之间的整数, 单位是 %。
#-DsamplingRate=100

# sql_log 表格大小
#-DsqlLogTableSize=1024
```

2.28.3 管理端命令

2.28.3.1 show @@statistic

查询statistic的开关、表格大小

```
show @@statistic;
+-----+-----+
| NAME                                | VALUE |
+-----+-----+
| statistic                           | OFF   |
| statisticAnalysis                    | OFF   |
| associateTablesByEntryByUserTableSize | 1024  |
| frontendByBackendByEntryByUserTableSize | 1024  |
| tableByUserByEntryTableSize          | 1024  |
| samplingRate                          | 0     |
| sqlLogTableSize                       | 1024  |
| queueMonitor                          | monitoring |
+-----+-----+
6 rows in set (0.01 sec)
```

2.28.3.2 disable @@statistic

关闭sql全量统计

```
disable @@statistic;
Query OK, 1 row affected (0.01 sec)
```

2.28.3.3 enable @@statistic

开启sql全量统计

```
enable @@statistic;
Query OK, 1 row affected (4.26 sec)
```

2.28.3.4 reload @@statistic_table_size = ? [where table='?' | where table in (dble_information.tableA,...)]

重置统计表的大小

```
reload @@statistic_table_size = 90;
Query OK, 1 row affected (0.02 sec)

reload @@statistic_table_size = 90 where table = 'sql_statistic_by_table_by_user_by_entry';
Query OK, 1 row affected (0.02 sec)

reload @@statistic_table_size = 90 where table in(sql_statistic_by_table_by_user_by_entry,sql_statistic_by_associate_tables_by_entry_by_user);
Query OK, 1 row affected (0.02 sec)

reload @@statistic_table_size = 90 where table = 'sql_log';
Query OK, 1 row affected (0.02 sec)
```

2.28.3.5 reload @@samplingRate=?

设置采样统计率(等于0表示关闭采样统计)

```
reload @@samplingRate=90;
Query OK, 1 row affected (0.01 sec)
```

2.28.3.2 disable @@statisticAnalysis

关闭show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition聚合统计

```
disable @@statisticAnalysis;
Query OK, 1 row affected (0.01 sec)
```

2.28.3.3 enable @@statisticAnalysis

开启show @@sql.sum.user、show @@sql.sum.table、show @@sql.condition聚合统计

```
enable @@statisticAnalysis;
Query OK, 1 row affected (4.26 sec)
```

2.28.4 管理端统计表

采样统计:

sql_log sql_log_by_digest_by_entry_by_user (sql_log表的视图)

sql_log_by_tx_by_entry_by_user (sql_log表的视图)

sql_log_by_tx_digest_by_entry_by_user (sql_log表的视图)

全量统计:

sql_statistic_by_frontend_by_backend_by_entry_by_user

sql_statistic_by_table_by_user_by_entry

sql_statistic_by_associate_tables_by_entry_by_user

以上表(非视图)都支持truncate命令

2.28.5 统计的规则

以业务端执行的事务(非事务查询算单语句事务)为单位同步将收集的数据流入统计表中.

sharding:

- 由dble层面解析表或数据库不存在等报错sql, 一律不参与统计
- explain、explain2语句不参与统计
- exit不参与统计

rwsplit:

- sql报1064错误码, 不参与统计
- 执行multi-query(指一次执行多个sql,mysql client可使用delimiter关键字实现), multi-query将会直接透传至后端节点, 这里会被视作为事务级sql(如commit), 参与统计

其他: 在 enableStatistic=0、statisticAnalysis=0、samplingRate=0的情况下, 动态开启任意其中一个, 会清理所有缓存的sql统计数据(sql_log、sql_statistic_by_frontend_by_xxxx表中的数据)

2.28.6 统计队列使用率的观测手段**2.28.6.1 start @@statistic_queue_monitor [observeTime = ? [and intervalTime = ?]]**

开始观测, 同时可设置观测总时长observeTime和采样间隔intervalTime(单位:s,m/min,h)

```
start @@statistic_queue_monitor; -- 使用默认值observeTime为1min, intervalTime为5s
start @@statistic_queue_monitor observeTime = 2min; -- observeTime为2min, intervalTime使用默认值5s
start @@statistic_queue_monitor observeTime = 2min and intervalTime = 10s; -- observeTime为2min, intervalTime为10s
```

2.28.6.2 stop @@statistic_queue_monitor"

停止观测

```
stop @@statistic_queue_monitor";
```

2.28.6.3 show @@statistic_queue.usage

查看队列的使用率情况列表(观测期间, 每次查询结果递增)

```
show @@statistic_queue.usage;
+-----+-----+
| TIME           | USAGE |
+-----+-----+
| 2021-05-31 16:33:30 | 0.00% |
| 2021-05-31 16:33:35 | 0.00% |
| 2021-05-31 16:33:40 | 0.00% |
+-----+-----+
3 rows in set (0.01 sec)
```

TIME: 采样时间点
USAGE: 使用率

2.28.6.4 drop @@statistic_queue.usage

清空使用率情况列表

```
drop @@statistic_queue.usage;
```

2.28.6.5 其他补充

- 1、统计队列在被观测情况下(show @@statistic中的queueMonitor对应值为monitoring), 执行关闭所有统计功能后(statistic为OFF且samplingRate为0)时, 则观测会被中断.
- 2、在未开启任意统计时, 执行start @@statistic_queue_monitor报错. 3、每次执行start @@statistic_queue_monitor, 都会先清空使用率情况列表.
- 4、使用率情况列表中的数据以软引用(SoftReference)方式作为缓存方式; 意味着: 当jvm内存不足时, 列表中的数据会被回收(现象: 列表的数据量变少).
- 5、查看统计队列大小(statisticQueueSize), 管理端中执行select * from dble_variables where variable_name='statisticQueueSize'.
- 6、统计队列大小(statisticQueueSize)值不支持动态改动; 在bootstrap.cnf中调整其值后, 需要重启dble才能生效.

2.29 load data批处理模式

2.29.1 介绍

在使用load data导出数据时，如果期间发生网络超时等异常状况就会导致load data产生回滚。所以引入了“分批导入”的处理方式：将需要导入的文件按照阈值（见bootstrap.cnf）拆分成多个文件进行分批导入，这样在load data途中发生异常状况时，已经成功已导入的文件不会被回滚，并会停止导入发生异常后的文件。再次基于源文件的load data，DBLE则会跳过已成功导入的文件继续load data。此外，在开启批处理模式后，如果待导入的文件按照既定load data语法规则（或不符合表结构等）存在错误，DBLE会停止load data并会展示该文件错误的sql语句的内容，待错误修复后，再次load data仍会跳过已导入的数据，从而节约时间成本。

注意：

- 1.如果文件发生错误，只有文件修改完成并且正确导入后或者使用kill @@load_data,/temp/error 下的文件才会被删除。
- 2.如果文件发生错误，在修改文件内容时修改了文件名称会被视为新文件重新导入。
- 3.每次导入文件会删除上次导入遗留的内容（如上次生成的错误文件等）。
- 4.错误文件命名方式为：数字-文件名-表名-下发节点名.txt，比如1-data-table-dn1.txt。1代表文件分割后的第一个文件，data为导入的文件名称，table代表插入的表名，dn1代表该文件需要下发的节点。

2.29.2 bootstrap.cnf中load data批处理模式的相关配置

```
# 开启BatchLoadData的开关，默认关闭；0-关闭，1-开启
#-DenableBatchLoadData=1
# 拆分文件的阈值，默认为100000
#-DmaxRowSizeToFile=100000
```

2.29.3 管理端命令

2.29.3.1 show @@load_data.fail

查询本次load data失败的文件

```
show @@load_data.fail;
Empty set (0.01 sec)

if have error file may like
show @@load_data.fail;
+-----+
| error_load_data_file          |
+-----+
| ./temp/error/1-data-table-dn1.txt |
| ./temp/error/1-data-table-dn2.txt |
+-----+
2 rows in set (0.01 sec)
```

2.29.3.2 disable @@load_data_batch

关闭load data批处理模式

```
disable @@load_data_batch;
Query OK, 1 row affected (0.00 sec)
disable load_data_batch success
```

2.29.3.3 enable @@load_data_batch

开启load data批处理模式

```
enable @@load_data_batch;
Query OK, 1 row affected (0.01 sec)
enable load_data_batch success
```

2.29.3.4 reload @@load_data.num=?

修改需要持久化的最大行数，在开启load data批处理模式下是拆分文件的阈值

```
reload @@load_data.num=200000;
Query OK, 1 row affected (0.00 sec)
reload @@load_data.num success
```

2.29.5 kill @@load_data

导入文件如果发生回滚，再次导入该文件时不再跳过已经成功导入的数据会从头重新导入数据

```
kill @@load_data;  
Query OK, 1 row affected (0.00 sec)  
kill @@load_data success
```


2.30 in子查询是否转join的说明

Issue

[问题链接](#)

example

- 执行sql
explain select a.* from gtest a where 1=1 and a.id in (select b.id from test b) order by a.id;
- inSubQueryTransformToJoin = true 时in子查询执行计划

```

+-----+-----+-----+
| SHARDING_NODE | TYPE | SQL/REF |
+-----+-----+-----+
| dn1_0 | BASE SQL | select `a`.`id`,`a`.`name` from `gtest` `a` ORDER BY `a`.`id` ASC |
| dn2_0 | BASE SQL | select `a`.`id`,`a`.`name` from `gtest` `a` ORDER BY `a`.`id` ASC |
| merge_and_order_1 | MERGE_AND_ORDER | dn1_0; dn2_0 |
| shuffle_field_1 | SHUFFLE_FIELD | merge_and_order_1 |
| dn1_1 | BASE SQL | select DISTINCT `b`.`id` as `autoalias_scalar` from `test` `b` ORDER BY autoali |
as_scalar ASC |
| dn2_1 | BASE SQL | select DISTINCT `b`.`id` as `autoalias_scalar` from `test` `b` ORDER BY autoali |
as_scalar ASC |
| merge_and_order_2 | MERGE_AND_ORDER | dn1_1; dn2_1 |
| distinct_1 | DISTINCT | merge_and_order_2 |
| shuffle_field_3 | SHUFFLE_FIELD | distinct_1 |
| rename_derived_sub_query_1 | RENAME_DERIVED_SUB_QUERY | shuffle_field_3 |
| shuffle_field_4 | SHUFFLE_FIELD | rename_derived_sub_query_1 |
| join_1 | JOIN | shuffle_field_1; shuffle_field_4 |
| shuffle_field_2 | SHUFFLE_FIELD | join_1 |
+-----+-----+-----+

```

- inSubQueryTransformToJoin = false 时in子查询执行计划

SHARDING_NODE	TYPE	SQL/REF
dn1_0	BASE SQL	select DISTINCT `b`.`id` as `autoalias_scalar` from `test` `b`
dn2_0	BASE SQL	select DISTINCT `b`.`id` as `autoalias_scalar` from `test` `b`
merge_1	MERGE	dn1_0; dn2_0
distinct_1	DISTINCT	merge_1
shuffle_field_1	SHUFFLE_FIELD	distinct_1
in_sub_query_1	IN_SUB_QUERY	shuffle_field_1
dn1_1	BASE SQL(May No Need)	in_sub_query_1; select `a`.`id`,`a`.`name` from `gtest` `a` where `a`.`id` in ('{NEED_TO_REPLACE}') ORDER BY `a`.`id` ASC
dn2_1	BASE SQL(May No Need)	in_sub_query_1; select `a`.`id`,`a`.`name` from `gtest` `a` where `a`.`id` in ('{NEED_TO_REPLACE}') ORDER BY `a`.`id` ASC
merge_and_order_1	MERGE_AND_ORDER	dn1_1; dn2_1
shuffle_field_2	SHUFFLE_FIELD	merge_and_order_1

Resolution

前置条件：test表数据量少称为小表，gtest表数据量多称为大表。子查询select b.id from test b简称为 subQuery。整条sql select a.* from gtest a where 1=1 and a.id in (select b.id from test b) order by a.id; 简称为sql。

正常使用in子查询中应该有意识的会把小表的查询结果当成条件放在大表中查询。在inSubQueryTransformToJoin = true的执行中会把多张表的数据都查出来然后再做join处理，这样处理方式可能并不符合预期使用子查询写法的目的（多查询了mysql中的数据）。在inSubQueryTransformToJoin = false的执行中会先处理subQuery，然后sql会带上subQuery的结果去mysql中查询，而非之前的对两张表做join处理。如果有嵌套子查询的情况，会先处理最里层的subQuery然后递归处理外面一层直至最外面一层。

conditions

- Column中包含子查询
- join时候包含子查询
- having中包含子查询
- order by 包含等值子查询
- where 后面包含子查询
- 子查询中嵌套子查询
- in子查询必须出现在where中才会被dble进行处理

explain comparison

- 子查询有三种形式，scalar_sub_query, in_sub_query, all_any_sub_query，出现于 SQL/REF中。只有处理 in_sub_query形式会出现执行计划不一致的情况。

example

sql

```
SELECT a.id, select max(b.id) from test b where b.id in (select distinct d.id from sing1 d) as name FROM sharding_4_t1 a ORDER BY a.id;
```

step

- 这条sql有子查询，并且子查询出现在Column处，那么这条子查询需要处理。
- in子查询出现在where条件中那么执行计划不一致。

special

any, some ,all函数可能会在dble中当成in子查询处理。

- any , some 函数，并且函数的前置操作符是 =, 会当成in子查询处理。
- all函数，并且函数的前置操作符是!=, <>, 会被当成in子查询处理。

example

等价于用in子查询写法。

- `select * from sharding_4_t1 where id=any(select id from test where age=1) order by name desc;`
- `select * from sharding_4_t1 where id=all(select id from test where age=1) order by name desc;`

不等价于用in子查询写法。

- `select * from sharding_4_t1 where id!=any(select id from test where age=1) order by name desc;`
- `select * from sharding_4_t1 where id=all(select id from test where age=1) order by name desc;`

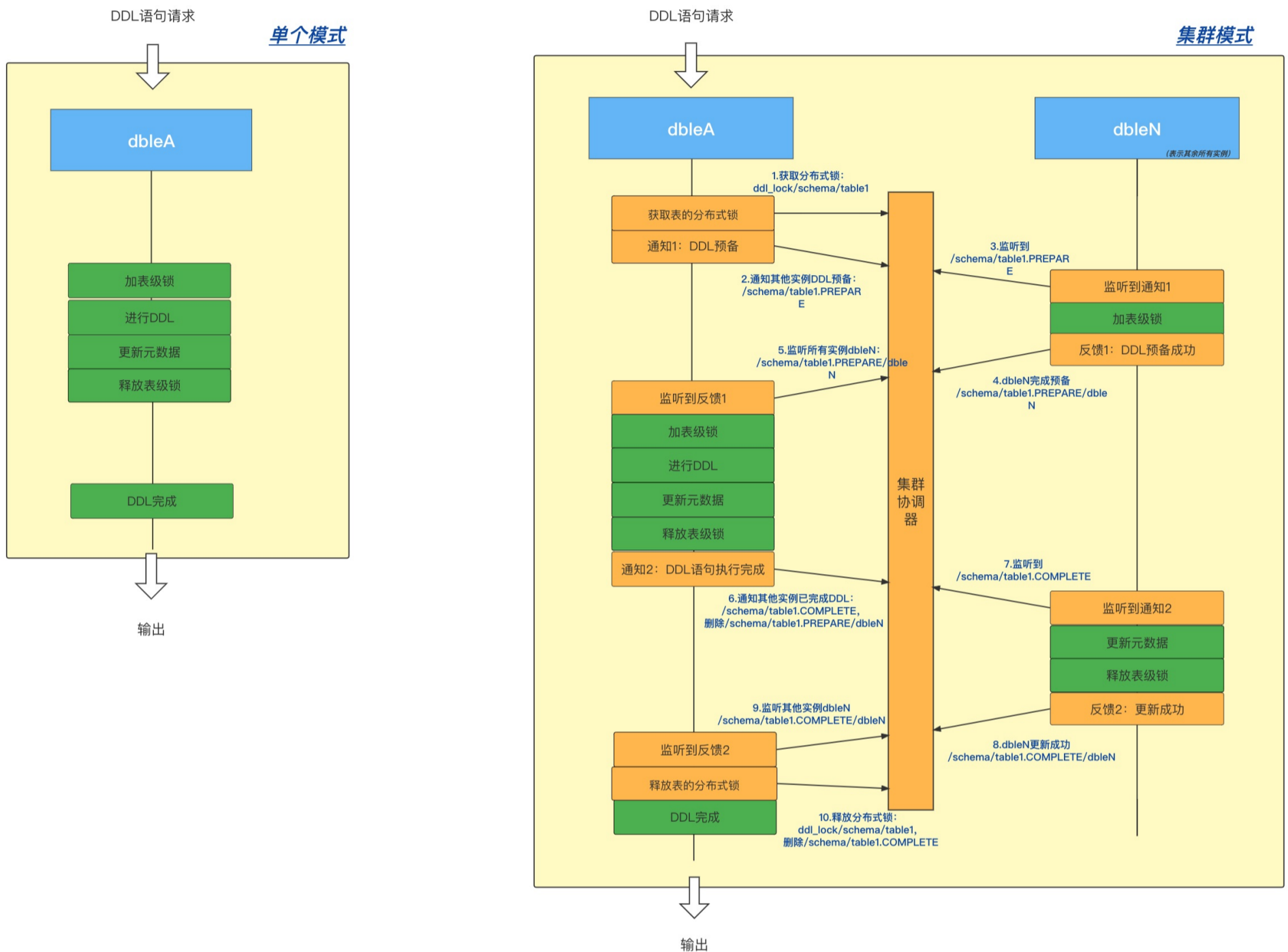
所以在以上特殊函数的特殊情况会当成in子查询进行处理。

2.31 DDL日志解读

适宜版本

>=3.22.01

DDL流程



日志格式

[DDL_{序号id}{.分片名}] <{阶段{.状态}}> {描述}

SQL1

```
`CREATE TABLE tableB (id int(11) DEFAULT NULL,id2 int(11) DEFAULT NULL,name varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin 1;`
```

阶段详解

大致阶段	时间线	dbleA实例	dbleN实例
init_ddl_trace 进入DDL日志追踪		[DDL_2] init_ddl_trace	
		[DDL_2] notice_cluster_ddl_prepare.start /*通知所有实例进入ddl.PREPARE阶段*/	
			[DDL_NOTIFIED] receive_ddl_prepare /*接收ddl.PREPARE通知*/

notice_cluster_ddl_prepare 通知集群ddl预备 (架构为非集群时, 跳过此阶段)			[DDL_NOTIFIED] add_table_lock.start /*开始对本实例加表级锁*/
			[DDL_NOTIFIED] add_table_lock.succ /*加锁成功*/
		[DDL_2] notice_cluster_ddl_prepare.succ /*所有实例都成功进入ddl.PREPARE 阶段*/	
add_table_lock 加表级锁		[DDL_2] add_table_lock.start /*开始对本实例加表级锁*/	
		[DDL_2] add_table_lock.succ /*加锁成功*/	
test_ddl_conn 测试后端连接, 执行'select 1'; (如果tableB是单节点表时, 跳过此 阶段)		[DDL_2] test_ddl_conn.start /*开始进入后端连接测试阶段 */	
		[DDL_2.dn1] test_ddl_conn.start /*开始进入dn1对应后端连接的测试 阶段 */	
		[DDL_2.dn1] test_ddl_conn.get_conn /*成功获取dn1对应后端连接, dn2~4同样 */	
		[DDL_2.dn1] test_ddl_conn.succ /*dn1对应后端连接执行select 1成 功, dn2~4同样 */	
		[DDL_2] test_ddl_conn.succ /*所有分片对应的后端连接执行 select 1成功, dn2~4同样 */	
exec_ddl_sql 执行ddl语句		[DDL_2] exec_ddl_sql.start /*开始执行sql */	
		[DDL_2.dn1] exec_ddl_sql.start /*开始进入dn1对应后端连接的执行 ddl阶段, dn2~4同样 */	
		[DDL_2.dn1] exec_ddl_sql.get_conn /*成功获取dn1对应后端连接, dn2~4同样*/	
		[DDL_2.dn1] exec_ddl_sql.succ /*dn1对应后端连接执行ddl成功, dn2~4同样*/	
		[DDL_2] exec_ddl_sql.succ /*执行ddl成功 */	
update_table_metadata 更新元数据		[DDL_2] update_table_metadata.start /*开始更新本实例中表的元数据*/	
		[DDL_2] update_table_metadata.succ /*更新成功*/	
notice_cluster_ddl_complete 通知集群DDL已完成 (架构为非集群时, 跳过此阶段)		[DDL_2] notice_cluster_ddl_complete.start /*通知所有实例进入ddl.COMPLETE 阶段*/	
			[DDL_NOTIFIED] receive_ddl_complete /*接收ddl.COMPLETE通知*/
			[DDL_NOTIFIED] update_table_metadata.start /*开始更新本实例中表的元数据*/
			[DDL_NOTIFIED] update_table_metadata.succ /*更新成功*/
			[DDL_NOTIFIED] release_table_lock.succ /*本实例释放表级锁*/
	[DDL_2] notice_cluster_ddl_complete.succ /*所有实例都成功进入 ddl.COMPLETE阶段*/		
release_table_lock 释放表级锁		[DDL_2] release_table_lock.succ /*本实例释放表级锁*/	
finish_ddl_trace 结束DDL日志追踪		[DDL_2] finish_ddl_trace	

状态

状态	描述
succ	成功
fail	失败
get_conn	成功获取后端连接

集群模式

架构：分别有dbleA、dbleN(表示其余实例)；在dbleA中执行SQL1

dbleA实例日志

```

2021-12-23 10:42:05,425 [INFO ][BusinessExecutor1] ===== init_ddl_trace [DDL_2] ===== (:)
2021-12-23 10:42:05,425 [INFO ][BusinessExecutor1] [DDL_2] <init_ddl_trace> Routes end and Start ddl{CREATE TABLE `tableB` (`id` int(11) D
EFAULT NULL, `id2` int(11) DEFAULT NULL, `name` varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1} execution stage. In Fronten
dConnection[id = 1 port = 8066 host = 127.0.0.1 local_port = 52436 isManager = false startupTime = 1640227316027 skipCheck = false isFlowC
ontrol = false] (:)
2021-12-23 10:42:05,425 [INFO ][BusinessExecutor1] [DDL_2] <notice_cluster_ddl_prepare.start> Notify and wait for all instances to enter p
hase PREPARE (:)
2021-12-23 10:42:05,547 [INFO ][BusinessExecutor1] [DDL_2] <notice_cluster_ddl_prepare.succ> All instances have entered phase PREPARE (:)
2021-12-23 10:42:05,547 [INFO ][BusinessExecutor1] [DDL_2] <add_table_lock.start> (:)
2021-12-23 10:42:05,547 [INFO ][BusinessExecutor1] [DDL_2] <add_table_lock.succ> (:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2] <test_ddl_conn.start> Start execute 'select 1' to detect a valid connection for
shardingNodes[dn1,dn3,dn2,dn4] (:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn1] <test_ddl_conn.start> In shardingNode[dn1],about to execute sql{select 1}
(:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn1] <test_ddl_conn.get_conn> Get BackendConnection[id = 9 host = 10.186.63.8 po
rt = 24801 localPort = 52423 mysqlId = 5924 db config = dbInstance[name=instanceM1,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn2] <test_ddl_conn.start> In shardingNode[dn2],about to execute sql{select 1}
(:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn2] <test_ddl_conn.get_conn> Get BackendConnection[id = 11 host = 10.186.63.7 p
ort = 24801 localPort = 52426 mysqlId = 3282 db config = dbInstance[name=instanceM2,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn3] <test_ddl_conn.start> In shardingNode[dn3],about to execute sql{select 1}
(:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn3] <test_ddl_conn.get_conn> Get BackendConnection[id = 8 host = 10.186.63.8 po
rt = 24801 localPort = 52424 mysqlId = 5925 db config = dbInstance[name=instanceM1,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn4] <test_ddl_conn.start> In shardingNode[dn4],about to execute sql{select 1}
(:)
2021-12-23 10:42:05,548 [INFO ][BusinessExecutor1] [DDL_2.dn4] <test_ddl_conn.get_conn> Get BackendConnection[id = 10 host = 10.186.63.7 p
ort = 24801 localPort = 52427 mysqlId = 3281 db config = dbInstance[name=instanceM2,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,550 [INFO ][complexQueryExecutor4] [DDL_2.dn1] <test_ddl_conn.succ> (:)
2021-12-23 10:42:05,550 [INFO ][complexQueryExecutor4] [DDL_2.dn3] <test_ddl_conn.succ> (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor4] [DDL_2.dn4] <test_ddl_conn.succ> (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2.dn2] <test_ddl_conn.succ> (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2] <test_ddl_conn.succ> (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2] <exec_ddl_sql.start> This ddl will be executed separately in the shardingNo
des[dn1,dn3,dn2,dn4] (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2.dn1] <exec_ddl_sql.start> In shardingNode[dn1],about to execute sql{CREATE T
ABLE `tableB` (`id` int(11) DEFAULT NULL, `id2` int(11) DEFAULT NULL, `name` varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1
} (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2.dn1] <exec_ddl_sql.get_conn> Get BackendConnection[id = 9 host = 10.186.63.8
port = 24801 localPort = 52423 mysqlId = 5924 db config = dbInstance[name=instanceM1,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2.dn2] <exec_ddl_sql.start> In shardingNode[dn2],about to execute sql{CREATE T
ABLE `tableB` (`id` int(11) DEFAULT NULL, `id2` int(11) DEFAULT NULL, `name` varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1
} (:)
2021-12-23 10:42:05,553 [INFO ][complexQueryExecutor2] [DDL_2.dn2] <exec_ddl_sql.get_conn> Get BackendConnection[id = 11 host = 10.186.63.
7 port = 24801 localPort = 52426 mysqlId = 3282 db config = dbInstance[name=instanceM2,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,554 [INFO ][complexQueryExecutor2] [DDL_2.dn3] <exec_ddl_sql.start> In shardingNode[dn3],about to execute sql{CREATE T
ABLE `tableB` (`id` int(11) DEFAULT NULL, `id2` int(11) DEFAULT NULL, `name` varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1
} (:)
2021-12-23 10:42:05,554 [INFO ][complexQueryExecutor2] [DDL_2.dn3] <exec_ddl_sql.get_conn> Get BackendConnection[id = 8 host = 10.186.63.8
port = 24801 localPort = 52424 mysqlId = 5925 db config = dbInstance[name=instanceM1,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,554 [INFO ][complexQueryExecutor2] [DDL_2.dn4] <exec_ddl_sql.start> In shardingNode[dn4],about to execute sql{CREATE T
ABLE `tableB` (`id` int(11) DEFAULT NULL, `id2` int(11) DEFAULT NULL, `name` varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1
} (:)
2021-12-23 10:42:05,554 [INFO ][complexQueryExecutor2] [DDL_2.dn4] <exec_ddl_sql.get_conn> Get BackendConnection[id = 10 host = 10.186.63.
7 port = 24801 localPort = 52427 mysqlId = 3281 db config = dbInstance[name=instanceM2,disabled=false,maxCon=10,minCon=3] (:)
2021-12-23 10:42:05,581 [INFO ][complexQueryExecutor4] [DDL_2.dn3] <exec_ddl_sql.succ> (:)
2021-12-23 10:42:05,581 [INFO ][complexQueryExecutor2] [DDL_2.dn1] <exec_ddl_sql.succ> (:)
2021-12-23 10:42:05,583 [INFO ][complexQueryExecutor2] [DDL_2.dn4] <exec_ddl_sql.succ> (:)
2021-12-23 10:42:05,604 [INFO ][complexQueryExecutor2] [DDL_2.dn2] <exec_ddl_sql.succ> (:)
2021-12-23 10:42:05,605 [INFO ][complexQueryExecutor2] [DDL_2] <exec_ddl_sql.succ> (:)
2021-12-23 10:42:05,606 [INFO ][complexQueryExecutor2] [DDL_2] <update_table_metadata.start> (:)
2021-12-23 10:42:05,608 [INFO ][complexQueryExecutor2] [DDL_2] <update_table_metadata> Start execute sql{show create table} in the shardin
gNodes[dn4] to get table[tableB]'s information (:)
2021-12-23 10:42:05,615 [INFO ][complexQueryExecutor4] [DDL_2] <update_table_metadata> In shardingNode[dn4], fetching success. (:)
2021-12-23 10:42:05,616 [INFO ][complexQueryExecutor4] [DDL_2] <update_table_metadata.succ> Successful to update table[testdb.tableB]metad
ata (:)
2021-12-23 10:42:05,616 [INFO ][complexQueryExecutor2] [DDL_2] <notice_cluster_ddl_complete.start> Notify and wait for all instances to en
ter phase COMPLETE (:)
2021-12-23 10:42:05,735 [INFO ][complexQueryExecutor2] [DDL_2] <notice_cluster_ddl_complete.succ> All instances have entered phase COMPLET
E (:)
2021-12-23 10:42:05,735 [INFO ][complexQueryExecutor2] [DDL_2] <release_table_lock.succ> (:)
2021-12-23 10:42:05,817 [INFO ][complexQueryExecutor2] [DDL_2] <finish_ddl_trace> Execute success (:)
2021-12-23 10:42:05,817 [INFO ][complexQueryExecutor2] ===== finish_ddl_trace [DDL_2] ===== (:)

```

```
2021-12-23 10:47:21,358 [INFO ][Curator-PathChildrenCache-4] [DDL_NOTIFIED] <receive_ddl_prepare> Received: initialize ddl{CREATE TABLE `tableB` (`id` int(11) DEFAULT NULL, `id2` int(11) DEFAULT NULL, `name` varchar(100) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1} of table[testdb.tableB] (:)
2021-12-23 10:47:21,358 [INFO ][Curator-PathChildrenCache-4] [DDL_NOTIFIED] <add_table_lock.start> (:)
2021-12-23 10:47:21,358 [INFO ][Curator-PathChildrenCache-4] [DDL_NOTIFIED] <add_table_lock.succ> (:)
2021-12-23 10:47:21,461 [INFO ][Curator-PathChildrenCache-4] [DDL_NOTIFIED] <receive_ddl_complete> Received: ddl execute success notice for table[testdb.tableB] (:)
2021-12-23 10:47:21,461 [INFO ][Curator-PathChildrenCache-4] [DDL_NOTIFIED] <update_table_metadata.start> (:)
2021-12-23 10:47:21,465 [INFO ][Curator-PathChildrenCache-4] [DDL_NOTIFIED] <update_table_metadata> Start execute sql{show create table} in the shardingNodes[dn1,dn2,dn3,dn4] to get table[tableB]'s information (:)
2021-12-23 10:47:21,469 [INFO ][complexQueryExecutor4] [DDL_NOTIFIED] <update_table_metadata> In shardingNode[dn1], fetching success. (:)
2021-12-23 10:47:21,469 [INFO ][complexQueryExecutor7] [DDL_NOTIFIED] <update_table_metadata> In shardingNode[dn4], fetching success. (:)
2021-12-23 10:47:21,469 [INFO ][complexQueryExecutor5] [DDL_NOTIFIED] <update_table_metadata> In shardingNode[dn3], fetching success. (:)
2021-12-23 10:47:21,470 [INFO ][complexQueryExecutor5] [DDL_NOTIFIED] <update_table_metadata> In shardingNode[dn2], fetching success. (:)
2021-12-23 10:47:21,471 [INFO ][complexQueryExecutor5] [DDL_NOTIFIED] <update_table_metadata.succ> Successful to update table[testdb.tableB]metadata (:)
2021-12-23 10:47:21,471 [INFO ][complexQueryExecutor5] [DDL_NOTIFIED] <release_table_lock.succ> (:)
```

单个模式

仅看上面dbleA实例日志，忽略集群相关日志即可。

日志检索方式

```
cat dble.log | grep '[DDL_2]' | grep '[DDL_NOTIFIED]'
```


2.32 分析用户

3.22.01.0版本dble支持单纯使用分析用户

2.32.1 分析用户配置

2.32.1.1 单纯使用分析用户功能的配置

仅需在 user.xml 文件中配置 analysisUser并指定对应的dbGroup即可。dbGroup的配置参考db.xml的章节。这里需要注意的是四种用户配置的顺序是固定的。user.xml的配置请参考user.xml章节。

```
<dble:user xmlns:dble="http://dble.cloud/" version="4.0">
  <managerUser name="man1" password="654321" maxCon="100"/>
  <shardingUser name="root" password="123456" schemas="testdb" readOnly="false" maxCon="20"/>
  <rwSplitUser name="rwsu1" password="123456" dbGroup="rwGroup" maxCon="20"/>
  <analysisUser name="analysisUser" password="123456" dbGroup="dbGroup3" blacklist="blacklist1" maxCon="20"/>
</dble:user>
```

配置注意事项：

1. 当user.xml文件中不配置shardingUser，dble不再加载sharding.xml配置文件(即dble不具备分表分库)，包括集群情况下出现sharding.xml不一致，均属于已知现象。
2. 多个analysisUser可以引用同一个dbGroup。
3. 被用户使用的dbGroup内的instance才会有心跳和连接池；未被有效使用的dbGroup内的instance只有心跳，不会初始化连接池。

2.32.2 负载均衡

dble通过配置多个dbInstance为读操作提供负载均衡，注意的是rwSplitMode配置不为0，详细请参见db.xml章节。负载均衡规则如下：

1. 确定参与读写分离的dbInstance集合
2. 负载均衡算法

2.3.2.1 确定参与读写分离的dbInstance集合

该算法在每次连接获取时提供可用的dbInstances实例集

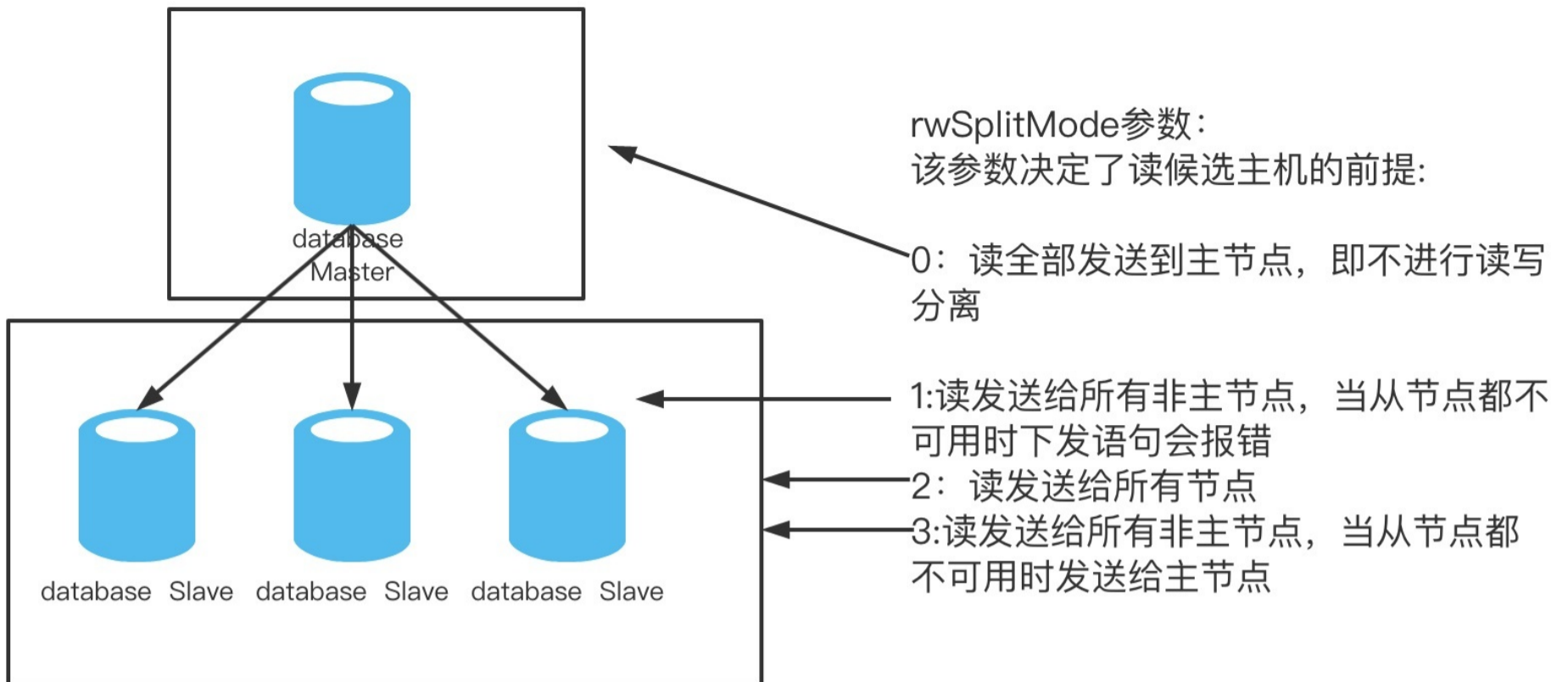
- 写节点(primary="true")可用
 - rwSplitMod配置为2，则写节点有资格参到读写分离，将写节点加入到dbInstances实例集
 - 读节点(primary没配置或者 primary="false")
 - 节点可用且需要进行延迟检测，检查延迟是否在阈值内再决定是否加入到dbInstances实例集
 - 节点可用且不需要进行延迟检测，直接加入到dbInstances实例集
- 写节点异常
 - 检查读节点是否可用，与上面读节点的检测机制一致

2.32.2.2 负载均衡算法

该算法在dbInstance集合中选择一个dbInstance实例来获取连接。

- dbInstance集合为空，前端报错。
- dbInstance集合非空
 - 每个dbInstance有权重设置(readWeight参数)，但不都是等值权重，依权重随机选择。
 - 每个dbInstance无权重设置或所有权重等值，则等权随机选择。此种情况只是上面情况的特例。

2.32.2.3 写节点是否参与均衡与dbGroup的rwSplitMode属性有关，具体见下图



2.32.3 分析用户支持语句类型

按照clickhouse对于mysql语法的兼容，对于select语句，dble目前都能支持

2.32.4 分析用户功能限制

1. 目前仅支持clickhouse
2. 目前支持select语句，其他类型语句dble不保证正确性
3. 分析用户不支持带库名直接登录

2.23 通过hint指定复杂查询执行计划

需求背景

有如下场景1:

```
table_a a left join table_b b on a.col_1 = b.col_1 left join table_c c on a.col_2 =c.col_2 where a.col =xxx
```

在3.22.01.0之前版本的查询计划是:

1. a 表带条件 a.col=xxx 下发, 结果集比较小, 大约数百
2. b 表大表, 全数据拉取
3. c 表小表, 全数据拉取

三表并发下发, 在dble内存中进行join, 其中 b 表比较大, 占用内存比较大, 这样造成这条sql的执行效率不高, 并且dble容易内存溢出。因此, 期望如下的查询计划:

1. a 表带条件下发, 结果集大约数百
2. b 表带着 a 表的结果下发
3. c 表带着 a 表的结果下发

这样, a 表先下发, 之后 b 表带上 a 表查询回来的 col_1 的结果下发, c 表带着 a 表查询回来的 col_2 的结果下发, 这里, b 表和 c 表的是可以并发下发的。最终将结果在dble内部进行join, 这样dble处理的结果集就小很多。

有如下场景2:

```
table_a a left join table_b b on a.col_1 = b.col_1 left join table_c c on a.sharding_col = c.sharding_col where a.col =xxx
```

同场景1的处理方式。因此, 期望如下的查询计划:

1. a, c 表优先进行联表查询处理, 带条件下发, 结果集大约数百
2. b 表带着 a 表查询返回的 col_1 的结果下发

有如下场景3:

```
table_a a left join table_b b on a.col_1 = b.col_1 left join table_c c on b.col_2 = c.col_2 where a.col =xxx
```

同场景1的查询计划。

因此, 期望如下的查询计划:

1. a 表带条件下发, 结果集大约数百
2. b 表带着 a 表的结果下发
3. c 表带着 b 表的结果下发

这样, a 表先处理, 然后 b 表带着 a 表 col_1 结果下发, 最后 c 表带着 b 表 col_2的值下发。

另外, 还可以有如下的查询计划:

1. a 表带条件下发, 结果集大约数百
2. b 表带着a表的结果下发
3. c 表数据量不大的情形下全量下发

这样, a 表先处理, 然后 b 表带着a 表 col_1 的结果下发, 同时 c 表并发

hint使用场景举例

数据插入 (jdbc方式)

```
import java.sql.*;
import java.util.*;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public abstract class jdbcTest {
    static AtomicInteger index = new AtomicInteger(0);
    static volatile Connection conn = null;
    private static List<Connection> list = new ArrayList<>();
    private static void createConn(String username, String password) {
        String JDBC_DRIVER = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://127.0.0.1:8066/test1?useSSL=false";
        try {
```

```

// 注册 JDBC 驱动
Class.forName(JDBC_DRIVER);
conn = DriverManager.getConnection(url, username, password);
list.add(conn);
} catch (Exception e) {
    e.printStackTrace();
}
}

private static void createTable() {
    Statement stmt;
    try {
        // 注册 JDBC 驱动
        Connection conn = list.get(index.incrementAndGet());
        stmt = conn.createStatement();
        stmt.addBatch("drop table if EXISTS t_spec_group;");
        stmt.addBatch("drop table if EXISTS t_spu;");
        stmt.addBatch("drop table if EXISTS t_sku;");
        stmt.addBatch("drop table if EXISTS t_warehouse_sku;");
        stmt.executeBatch();
        stmt.addBatch("create table t_spec_group(" +
            "    id      int unsigned primary key comment '主键'," +
            "    spg_id int unsigned not null comment '品类ID'," +
            "    `type` varchar(200) not null comment '品类类型'," +
            "    `name` varchar(200) not null comment '品类名称'" +
            ") comment ='详细品类表';");
        stmt.addBatch("create table t_spu(" +
            "    id          int unsigned primary key comment '主键'," +
            "    title        varchar(200) not null comment '标题'," +
            "    category_id  int unsigned not null comment '产品ID'," +
            "    saleable     int unsigned not null comment '是否上架'," +
            "    spg_id       int unsigned comment '品类ID'" +
            ") comment ='产品表';");
        stmt.addBatch("create table t_sku(" +
            "    id          int unsigned primary key comment '主键'," +
            "    spu_id       int unsigned not null comment '商品ID'," +
            "    spg_id       int unsigned not null comment '品类ID'," +
            "    title        varchar(200) not null comment '标题'," +
            "    price        int unsigned not null comment '价格'" +
            ") comment ='商品表';");
        stmt.addBatch("create table t_warehouse_sku(" +
            "    warehouse_id int unsigned comment '主键'," +
            "    sku_id        int unsigned comment '商品ID'," +
            "    spg_id       int unsigned not null comment '品类ID'," +
            "    title        varchar(200) not null comment '标题'," +
            "    type         varchar(200) comment '品类类型'," +
            "    num          int unsigned not null comment '库存数量'" +
            ") comment ='仓库商品库存表';");
        stmt.executeBatch();
        stmt.clearBatch();
        System.out.println("-----end-----");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private static void insertSpec_group() {
    PreparedStatement ps = null;
    try {
        Connection conn = list.get(index.incrementAndGet());
        String sql = "INSERT INTO t_spec_group (id, spg_id,type, name) VALUES (?, ?, ?, ?)";
        ps = conn.prepareStatement(sql);
        int size = 300;
        for (int i = 0; i < size; i++) {

            if (i < 200) {
                ps.setInt(1, i);
                ps.setInt(2, i + 2000000);
                ps.setString(3, "phone");
                ps.setString(4, "iphone" + i);
            } else {
                ps.setInt(1, i);
                ps.setInt(2, i);
                ps.setString(3, "desk" + i);
                ps.setString(4, "idesk" + i);
            }
            ps.addBatch();
        }
    }
}

```

```

        if (i % 500 == 0) {
            // 执行批量更新
            ps.executeBatch();
            // 清空执行过的sql
            ps.clearBatch();
        }
    }
    ps.executeBatch();
    // 清空执行过的sql
    ps.clearBatch();
    System.out.println("-----insertSpec_group---end-----");
} catch (Exception e) {
    e.printStackTrace();
}
}

private static void insertT_spu() {
    PreparedStatement ps = null;
    try {
        Connection conn = list.get(index.incrementAndGet());
        String sql = "INSERT INTO t_spu (id, title, category_id,saleable,spg_id) VALUES (?, ?, ?, ?, ?)";
        ps = conn.prepareStatement(sql);
        int size = 1000000;
        for (int i = 0; i < size; i++) {
            ps.setInt(1, i);
            if (i < 200) {
                ps.setString(2, "this is phone");
                ps.setInt(5, i + 2000000);
            } else {
                ps.setString(2, "this is desk" + i);
                ps.setInt(5, i);
            }

            ps.setInt(3, i);
            ps.setInt(4, 1);

            ps.addBatch();
            if (i % 500 == 0) {
                // 执行批量更新
                ps.executeBatch();
                // 清空执行过的sql
                ps.clearBatch();
            }
        }
        ps.executeBatch();
        // 清空执行过的sql
        ps.clearBatch();
        System.out.println("-----insertT_spu---end-----");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private static void insertT_sku() {
    PreparedStatement ps = null;
    try {
        Connection conn = list.get(index.incrementAndGet());
        String sql = "INSERT INTO t_sku (id, spu_id,spg_id,title,price) VALUES (?, ?, ?,?, ?)";
        ps = conn.prepareStatement(sql);
        int size = 1000000;
        for (int i = 0; i < size; i++) {
            ps.setInt(1, i);
            ps.setInt(2, i);

            if (i < 200) {
                ps.setInt(3, i + 2000000);
                ps.setString(4, "iphone" + i);
            } else {
                ps.setInt(3, i);
                ps.setString(4, "idesk" + i);
            }
        }
        ps.setInt(5, new Random().nextInt(2000));
        ps.addBatch();
        if (i % 500 == 0) {
            // 执行批量更新
            ps.executeBatch();
            // 清空执行过的sql
            ps.clearBatch();
        }
    }
}

```

```

    }
    }
    ps.executeBatch();
    // 清空执行过的sql
    ps.clearBatch();
    System.out.println("----- insertT_sku---end-----");
} catch (Exception e) {
    e.printStackTrace();
}
}

private static void insertT_warehouse_sku() {
    PreparedStatement ps = null;
    try {
        Connection conn = list.get(index.incrementAndGet());
        String sql = "INSERT INTO t_warehouse_sku (warehouse_id, sku_id, spg_id, title, type, num) VALUES (?, ?, ?, ?, ?, ?)";
        ps = conn.prepareStatement(sql);
        int size = 1000000;
        for (int i = 0; i < size; i++) {
            ps.setInt(1, i);
            ps.setInt(2, i);
            if(i < 200){

                ps.setInt(3, i + 2000000);
                ps.setString(4, "iphone" + i);
                ps.setString(5, "phone");
            }else {
                ps.setInt(3, i);
                ps.setString(4, "idesk" + i);
                ps.setString(5, "desk");
            }
            ps.setInt(6, new Random().nextInt(200));
            ps.addBatch();
            if (i % 500 == 0) {
                // 执行批量更新
                ps.executeBatch();
                // 清空执行过的sql
                ps.clearBatch();
            }
        }
        ps.executeBatch();
        // 清空执行过的sql
        ps.clearBatch();
        System.out.println("-----insertT_warehouse_sku---end-----");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws InterruptedException {
    int size = 6;
    //需要改成user.xml中配置的用户名和密码
    String username = "aa";
    String password = "123456";
    ThreadPoolExecutor executor = new ThreadPoolExecutor(size, size, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<>());
    for (int i = 0; i < size; i++) {
        createConn(username, password);
    }
    createTable();
    executor.execute(() -> insertSpec_group());
    executor.execute(() -> insertT_warehouse_sku());
    executor.execute(() -> insertT_sku());
    executor.execute(() -> insertT_spu());
}
}

```

sharding.xml

```

<?xml version="1.0"?>
<!DOCTYPE dble:sharding SYSTEM "sharding.dtd">
<dble:sharding xmlns:dble="http://dble.cloud/">
  <schema name="test1" >
    <shardingTable name="t_spec_group" shardingNode="dn1,dn2" function="sql-mod" shardingColumn="id"></shardingTable>
    <shardingTable name="t_spu" shardingNode="dn1,dn2" function="sql-mod" shardingColumn="id"></shardingTable>
    <shardingTable name="t_sku" shardingNode="dn1,dn2" function="hash-string-into-two" shardingColumn="title"></shardingTable>
    <shardingTable name="t_warehouse_sku" shardingNode="dn1,dn2" function="hash-string-into-two" shardingColumn="title"></shardingTable>
  </schema>

  <shardingNode dbGroup="dbGroup1" database="db1" name="dn1"/>
  <shardingNode dbGroup="dbGroup2" database="db1" name="dn2"/>
  <shardingNode dbGroup="dbGroup3" database="db1" name="dn3"/>
  <shardingNode dbGroup="dbGroup4" database="db1" name="dn4"/>

  <function name="hash-string-into-two" class="StringHash">
    <property name="partitionCount">2</property>
    <property name="partitionLength">1</property>
  </function>

  <function name="sql-mod" class="Hash">
    <property name="partitionCount">2</property>
    <property name="partitionLength">1</property>
  </function>

</dble:sharding>

```

db.xml

```

<?xml version="1.0"?>
<!--
  ~ Copyright (C) 2016-2020 ActionTech.
  ~ License: http://www.gnu.org/licenses/gpl.html GPL version 2 or higher.
-->
<!DOCTYPE dble:db SYSTEM "db.dtd">
<dble:db xmlns:dble="http://dble.cloud/" version="4.0">
  <dbGroup name="dbGroup1" rwSplitMode="0" delayThreshold="10000" >
    <heartbeat timeout="30" >show slave status</heartbeat>
    <dbInstance name="M1" url="ip1:3306" user="root" password="123456" maxCon="300" minCon="10" id="100"
      primary="true" >
    </dbInstance>
  </dbGroup>

  <dbGroup name="dbGroup2" rwSplitMode="0" delayThreshold="10000">
    <heartbeat>show slave status</heartbeat>
    <dbInstance name="M2" url="ip2:3306" user="root" password="123456" id="1" maxCon="2000" minCon="10"
      primary="true">
    </dbInstance>
  </dbGroup>

  <dbGroup name="dbGroup3" rwSplitMode="0" delayThreshold="10000">
    <heartbeat errorRetryCount="1" timeout="10">show slave status</heartbeat>
    <dbInstance name="M3" url="ip3:3306" user="root" password="123456" id="1" maxCon="2000" minCon="10"
      primary="true">
    </dbInstance>
  </dbGroup>

  <dbGroup name="dbGroup4" rwSplitMode="2" delayThreshold="10000">
    <heartbeat errorRetryCount="1" timeout="10">show slave status</heartbeat>
    <dbInstance name="M4" user="root" password="123456" url="ip4:3306" maxCon="20" minCon="10"
      primary="true">
    </dbInstance>
  </dbGroup>
</dble:db>

```

带有where条件

场景一

```
select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on b.spg_id=c.spg_id where a.type = 'phone';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dbtle:plan=a & b & c */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on b.spg_id=c.spg_id where a.type = 'phone';
```

hint语法 a & b & c

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint写法

场景二

```
select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dbtle:plan=a & b & c */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone';
```

hint语法 a & b & c

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint写法

使用hint后

```
/*!dbtle:plan=a & (b | c) */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone';
```

hint语法 a & (b | c)

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百，c表带着a表的结果下发，结果集约为数百

结论：当a表的结果集较小且b表和c表的结果集较大，更推荐该hint写法

场景三

```
select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on b.spg_id=c.spg_id where a.type = 'phone';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集约为数百

使用hint后

```
/*!dbtle:plan=a & b & c */ select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on b.spg_id=c.spg_id where a.type = 'phone';
```

hint语法 a & b & c

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint写法

使用hint后

```
/*!dbtle:plan=a & b | c */ select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on b.spg_id=c.spg_id where a.type = 'phone';
```

hint语法 a & b | c

1. a表带有条件下发，结果集约为数百，c表直接下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百

结论：当a表的结果集和c表的结果集较小且b表的结果集较大，更推荐使用该hint写法

场景四

```
select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on a.spg_id=c.spg_id where a.type = 'phone';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集约为数百

使用hint后

```
/*!dbtle:plan=a & b & c */ select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on a.spg_id=c.spg_id where a.type = 'phone';
```

hint语法 a & b & c

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint'写法

使用hint后

```
/*!dbtle:plan=a & (b | c) */ select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on a.spg_id=c.spg_id where a.type = 'phone';
```

hint语法 a & (b | c)

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百,c表带着a表的结果下发，结果集约为数百

结论：当a表的结果集较小且b表和c表的结果集较大时，更推荐使用该hint写法

使用hint后

```
/*!dbtle:plan=a & b | c */ select * from t_warehouse_sku a inner join t_sku b on a.sku_id = b.id inner join t_spec_group c on a.spg_id=c.spg_id where a.type = 'phone';
```

hint语法 a & b | c

1. a表带有条件下发，结果集约为数百，c表直接下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百

结论：当a表和c表的结果集较小且b表的结果集较大，更推荐使用该hint写法

场景五

```
select * from t_warehouse_sku a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on a.title=c.title where a.type = 'phone';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dbtle:plan=(a,c) & b*/ select * from t_warehouse_sku a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on a.title=c.title where a.type = 'phone';
```

hint语法 (a,c) & b

1. a表和c表带有条件整体下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百

结论：推荐使用hint写法

场景六

```
select * from t_warehouse_sku a inner join t_spec_group b on a.spg_id = b.spg_id inner join t_sku c on a.title=c.title where a.type = 'phone';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dble:plan=(a,c) & b*/ select * from t_warehouse_sku a inner join t_spec_group b on a.spg_id = b.spg_id inner join t_sku c on a.title=c.title where a.type = 'phone';
```

hint语法 (a,c) & b

1. a表和c表带有条件整体下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百

结论：推荐使用hint写法

使用hint后

```
/*!dble:plan=(a,c) | b*/ select * from t_warehouse_sku a inner join t_spec_group b on a.spg_id = b.spg_id inner join t_sku c on a.title=c.title where a.type = 'phone';
```

hint语法 (a,c) | b

1. a表和c表带有条件整体下发，结果集约为数百,b表直接结果下发，结果集约为数百

结论：当a表和c表整体下发且b表的结果集较小时，更推荐使用该hint写法

场景七

```
select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and b.category_id < 200;
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表带有条件下发，结果集约为数百
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dble:plan=a & b & c */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and b.category_id < 200;
```

hint语法 a & b & c

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发和where条件下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint写法

使用hint后

```
/*!dble:plan=a & (b | c) */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and b.category_id < 200;
```

hint语法 a & (b | c)

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发和where条件下发，结果集约为数百
3. c表带着a表的结果下发，结果集约为数百

结论：当a表的结果集较小并且b表和c表的结果集较大时,更推荐使用该hint写法

场景八

```
select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and c.title like 'iphone%';
```

使用hint前（并发下发）

1. a表带有条件下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表带有条件下发，结果集约为数百

使用hint后

```
/*!dble:plan=a & b & c */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and c.title like 'iphone%';
```

hint语法 a & b & c

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果和where条件下发，结果集约为数百

结论：推荐使用hint写法

使用hint后

```
/*!dble:plan=a & (b | c) */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and c.title like 'iphone%';
```

hint语法 a & (b | c)

1. a表带有条件下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表的结果和where条件下发，结果集约为数百

结论：当a表的结果集较小且b表和c表的结果集较大,推荐使用hint写法

使用hint后

```
/*!dble:plan=a & b | c */ select * from t_warehouse_sku a left join t_spu b on a.spg_id = b.spg_id left join t_sku c on a.sku_id=c.id where a.type = 'phone' and c.title like 'iphone%';
```

hint语法 a & b | c

1. a表带有条件下发，结果集约为数百,c表带有条件下发，结果集约为数百，
2. b表带着a表的结果下发，结果集约为数百

结论：当a表和c表的结果集较小且b表的结果集较大,更推荐该hint写法

不带有where条件

场景场景一

```
select * from t_spec_group a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on b.spg_id=c.spg_id;
```

使用hint前（并发下发）

1. a表直接下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dble:plan=a & b & c */ select * from t_spec_group a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on b.spg_id=c.spg_id ;
```

hint语法 a & b & c

1. a表直接下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint写法

场景二

```
select * from t_spec_group a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on a.spg_id=c.spg_id;
```

使用hint前（并发下发）

1. a表直接下发，结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dble:plan=a & b & c */ select * from t_spec_group a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on a.spg_id=c.spg_id ;
```

hint语法 a & b & c

1. a表直接下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百
3. c表带着a表和b表的结果下发，结果集约为数百

结论：推荐使用hint写法

使用hint后

```
/*!dble:plan=a & ( b | c) */ select * from t_spec_group a inner join t_spu b on a.spg_id = b.spg_id inner join t_sku c on a.spg_id=c.spg_id;
```

hint语法 a & (b | c)

1. a表直接下发，结果集约为数百
2. b表带着a表的结果下发，结果集约为数百，c表带着a表的结果下发，结果集约为数百

结论：当a表的结果集较小，更推荐使用该hint写法

场景三

```
select * from t_spec_group a inner join t_warehouse_sku b on a.spg_id = b.spg_id left join t_sku c on b.title=c.title;
```

使用hint前（并发下发）

1. a表直接下发，全数据拉取结果集约为数百
2. b表直接下发，全数据拉取结果集为百万
3. c表直接下发，全数据拉取结果集为百万

使用hint后

```
/*!dble:plan=(b,c) & a */ select * from t_spec_group a inner join t_warehouse_sku b on a.spg_id = b.spg_id left join t_sku c on b.title=c.title;
```

hint语法 (b,c) & a

1. b表和c表整体下发，结果集约为数百
2. a表带着b表的结果下发，结果集约为数百

结论：b表和c表存在er关系,推荐使用hint写法

使用hint后

```
/*!dble:plan=(b,c) | a */ select * from t_spec_group a inner join t_warehouse_sku b on a.spg_id = b.spg_id inner join t_sku c on b.title=c.title;
```

hint语法 (b,c) | a

1. b表和c表整体下发，结果集约为数百
2. a表带着b表的结果下发，结果集约为数百

结论：b表和c表存在er关系且a表的结果集较小，更推荐使用该hint写法

hint语法

针对上面三种场景，dble不能估算数据量的大小，按照表达式运算来尽量优化下发顺序。在dble 3.22.01.0版本中，dble提供通过hint的方式让用户可以自定义合理的执行顺序。

hint 的语法沿用 [dble hint](#) 比如：

```
/*!dble:plan=a & ( b | c )$left2inner$right2inner$in2join$use_table_index*/ sql
```

其中关键点在于 `a & (b | c)` 表达式，其中 `a`, `b`, `c` 表示 sql 中的 **表的别名**

我们使用 `&`, `|` 表示两表操作的先后顺序。针对上面的不同场景可以使用如下表达式指定复杂查询的执行顺序：

- 对于场景1: `a & (b | c)`
- 对于场景2: `(a,c) & b`
- 对于场景3: 第一种小场景可以是: `a&b&c` , 第二种小场景可以是`(a & b) | c`

其中：

1. `(a,c)` 表示`a`和`c`表之间存在ER关系，可以整体下推
2. `&` 表示后面的内容依赖前面的内容，需要等待前面的结果返回之后带入到后面之中作为条件下发，相当于`nestloop`的方式
3. `|` 表示两者可以并发，数据处理方式取决于`join`的方式
4. `left2inner` 参数表示是将`left join`转成`inner join`
5. `right2inner` 参数表示是将`right join`转成`inner join`
6. `in2join` 参数表示将`in`子查询转为`join`查询；（此参数优先于`bootstrap.cnf`中的`inSubQueryTransformToJoin`策略）

在实际使用中，sql中的表别名通常是由框架生成，不易获取。dble提供 `use_table_index` 参数，使用该参数可以通过sql中表的序列号来表示表的别名。比如：

```
/*!dble:plan=1 & 2 & 3 $use_table_index*/ select * from t1 a left join t2 b on a.id = b.id left join t3 c on a.id=c.id
```

这样的话，1 就表示表 `a`，2 表示表 `b`，3 表示表 `c`。1, 2, 3表示 sql 中的 **表的别名序列号**

等价于：

```
/*!dble:plan=a & b & c*/ select * from t1 a left join t2 b on a.id = b.id left join t3 c on a.id=c.id
```

hint使用nestLoop的原则

- hint期望的下发结果，如果违背优化的初衷那么就会报错
举例： `a join b` ,如果`a,b`具有`er`关系，hint希望执行为 `(a & b)` ,那么就会报错
- hint期望的下发方式被判定为不合理就会报错
举例： `a join b on a.col1 = b.col1 join c on c.col2 = a.col2` , hint希望执行为 `(a & b & c)` , 那么就会报错

限制

1. 对于像 `Hibernate` 这样自动生成表别名的框架，当前还不支持。后续会优化。
2. 当 sql 存在笛卡尔积（`join` 不指定关联key）时，暂不支持，hint会报错。举例：`select * from table_a a, table_b b`
3. 当 sql 存在 多个 `right join` 时，暂不支持，hint会报错
4. 当 sql 存在 子查询 时，暂不支持，hint会报错
5. `left join` 和 `inner join`指向同一个节点的执行顺序不被允许，会报错。举例：`/*!dble:plan=a & c & b */ SELECT * FROM Employee a LEFT JOIN Dept b on a.name=b.manager inner JOIN Info c on a.name=c.name and b.manager=c.name ORDER BY a.name;` 其中，`a` 和 `c` 可以正常 `inner join` ,但其结果和 `b` 发生 `join` 时，需要同时完成 `a` 和 `b` 的 `left join`以及 `c` 和 `b` 的`inner join`，这在sql语法上不受支持，故不支持。
6. sql具有`er`关系，但是hint依旧下发成功。
原因：我们尽可能的按照hint期望的方式下发语句，所以dble可能尝试在内部改写sql以便满足hint的需求，举例 `/*!dble:plan=a | c | b */ SELECT a.Name,a.DeptName,b.Manager,c.salary FROM Employee a LEFT JOIN Dept b on a.DeptName=b.DeptName LEFT JOIN Level c on a.Level=c.levelname and c.salary=10000 order by a.Name ;` 会被调整为 `SELECT a.Name,a.DeptName,b.Manager,c.salary FROM Employee a LEFT JOIN Level c on a.Level=c.levelname and c.salary=10000 LEFT JOIN Dept b on a.DeptName=b.DeptName order by a.Name` ,此时`a`表和`c`表不具有`er`关系，且`er`关系的检测不能跨节点，所以没有违背hint使用`nestLoop`的原则的第一条,可以正常下发

2.34 dble安全加密

默认情况下，前侧应用与dble之间采用非加密连接，因此第三方可以在网络层面抓取协议包并解析，从而查看交互的数据，包含查询的语句、结果等，那么就会造成数据库中数据的泄露。

为了解决以上提到的数据可能泄露的问题，参考MySQL加密方式，dble同样支持启用基于TLS协议的加密连接，前侧使用MySQL Client、MySQL驱动都可以连接加密后的DBLE。

关于加密方式如何配置请参考以下：

- [2.34.1 SSL自签名证书生成](#)
- [2.34.2 DBLE启用SSL](#)

2.34.1 SSL自签名证书生成

MySQL中使用的是自签名证书，自签名证书是由不受信的CA机构颁发的数字证书，也就是自己签发的证书。与受信任的CA签发的传统数字证书不同，自签名证书是由一些公司或软件开发商创建、颁发和签名的。因此这里DBLE也将采用自签名证书方式制作SSL证书

证书介绍

证书名称	说明
ca.pem	自签名CA证书；用于验证数字证书的可信度
server-cert.pem、server-key.pem	服务端数字证书和私钥；作为服务端身份，适用于除java以外的语言
client-cert.pem、client-key.pem	客户端数字证书和私钥；作为客户端身份，适用于除java以外的语言
truststore.jks	包含自签名CA证书的JKS密钥库；适用于java语言
serverkeystore.jks	包含服务端数字证书和私钥的JKS密钥库；适用于java语言
clientkeystore.jks	包含客户端数字证书和私钥的JKS密钥库；适用于java语言

证书生成

MySQL中[ca.pem]就是自签CA证书，服务端证书[server-cert.pem]和客户端证书[client-cert.pem]都是由[ca.pem]签发的。

以下生成方式需要借助openssl，需要提前安装

yum安装方式：yum install openssl -y

具体生成步骤如下：

```

1、制作CA自签名证书(包含公钥)和私钥
# 创建CA私钥 [ca-key.pem]:
openssl genrsa 2048 > ca-key.pem
# 使用私钥生成对应的证书[ca.pem]
openssl req -new -x509 -nodes -days 3600 -key ca-key.pem -out ca.pem

2、创建私钥和签发服务端的数字证书
# 创建服务端私钥[server-key.pem]和服务端的签发请求[server-req.pem]
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout server-key.pem -out server-req.pem
# 将服务端私钥转成RSA私钥文件格式
openssl rsa -in server-key.pem -out server-key.pem
# 使用CA私钥根据签发请求签发生成服务端证书[server-cert.pem]，其证书包含公钥、所有者、有效期等明文信息，也有经过CA私钥对公钥、所有者、有效期等加密后的签名
openssl x509 -req -in server-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out server-cert.pem

3、创建私钥和签发客户端的数字证书
# 创建客户端私钥[client-key.pem]和客户端的签发请求[client-req.pem]
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout client-key.pem -out client-req.pem
# 将客户端私钥转成RSA私钥文件格式
openssl rsa -in client-key.pem -out client-key.pem
# 使用CA私钥根据签发请求签发生成客户端证书[client-cert.pem]，其证书包含公钥、所有者、有效期等明文信息，也有经过CA私钥对公钥、所有者、有效期等加密后的签名
openssl x509 -req -in client-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out client-cert.pem

4、验证服务端和客户端数字证书是否可信，当输出的结果为OK，表示通过
openssl verify -CAfile ca.pem server-cert.pem client-cert.pem

5、证书内容查看
openssl x509 -text -in ca.pem
openssl x509 -text -in server-cert.pem
openssl x509 -text -in client-cert.pem

```

证书转换

openssl生成的证书格式pem、crt等格式，在Java语言并不能识别，需要额外使用keytool工具转换成p12、jks格式

keytool是一个JAVA数据证书的管理工具，keytool会将密钥及证书，其中可包括私钥、信任证书存储在一个keystore的文件中，用于（通过数字签名）自我认证（用户向别的用户/服务认证自己）或数据完整性以及认证服务

具体操作步骤如下：

1、自签CA证书的JKS密钥库

#将[ca.pem]导入Java平台的密钥库中，java支持密钥库类型有：JKS，JCEKS，PKCS12，PKCS11和DKS。这里生成JKS扩展名的truststore.jks密钥库，此密钥库的密码设置为123456（可自定义）

```
keytool -import -noprompt -file ca.pem -keystore truststore.jks -storepass 123456
```

2、服务端的数字证书的JKS密钥库

#将[server-cert.pem]和[server-key.pem](证书的密钥文件)转成p12类型的密钥库，然后在转成JKS类型的密钥库，此密钥库的密码设置为123456（可自定义）

```
openssl pkcs12 -export -in server-cert.pem -inkey server-key.pem -out serverkeystore.p12 -passout pass:123456
```

```
keytool -importkeystore -srckeystore serverkeystore.p12 -srcstoretype PKCS12 -destkeystore serverkeystore.jks -srcstorepass 123456 -deststorepass 123456
```

3、客户端的数字证书的JKS密钥库

#将[client-cert.pem]和[client-key.pem](证书的密钥文件)转成p12类型的密钥库，然后在转成JKS类型的密钥库，此密钥库的密码设置为123456

```
openssl pkcs12 -export -in client-cert.pem -inkey client-key.pem -out clientkeystore.p12 -passout pass:123456
```

```
keytool -importkeystore -srckeystore clientkeystore.p12 -srcstoretype PKCS12 -destkeystore clientkeystore.jks -srcstorepass 123456 -deststorepass 123456
```


2.34.2 DBLE启用SSL

服务端（DBLE）

配置bootstrap.cnf（需填写绝对路径）

```
-DsupportSSL=true
-DserverCertificateKeyStoreUrl=${服务端数字证书和私钥的JKS密钥库}
-DserverCertificateKeyStorePwd=${对应密钥库的密码}
-DtrustCertificateKeyStoreUrl=${自签名CA证书的JKS密钥库}
-DtrustCertificateKeyStorePwd=${对应密钥库的密码}
```

检查是否配置成功（管理端9066中查看）

```
mysql> select * from dble_variables where comment like '%SSL%';
+-----+-----+-----+-----+-----+-----+
| variable_name          | variable_value                                     | comment                                     |
|-----+-----+-----+-----+-----+-----+
| isSupportSSL          | true                                               | Whether support for SSL to establish from |
| serverCertificateKeyStoreUrl | ${服务端数字证书和私钥的JKS密钥库}             | Service certificate required for SSL      |
| trustCertificateKeyStoreUrl | ${自签名CA证书的JKS密钥库}                     | Trust certificate required for SSL       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.07 sec)
```

注意: 如果isSupportSSL为false, 表示不支持ssl协议; 根据dble.log启动日志中找到ssl初始失败的原因, 比如, 可能是密钥库的路径配置不对等。

客户端

建立连接的SSL模式

参照MySQL中的SSL配置, DBLE也为客户端提供了以下几种模式:

- ssl-mode=DISABLED

描述: Client端使用未加密的连接

```
client: mysql -u*** -p*** --ssl-mode=DISABLED
```

```
jdbc: jdbc:mysql://localhost:8066/testdb?useSSL=false
```

- ssl-mode=PREFFERED

描述: 默认行为, client端尝试使用加密进行连接, 如果无法构建加密连接, 则会退回到未加密的连接

```
client: mysql -u*** -p*** --ssl-mode=PREFFERED
```

```
jdbc: jdbc:mysql://localhost:8066/testdb?requireSSL=false&useSSL=true&verifyServerCertificate=false
```

- ssl-mode=REQUIRED

描述: Client端需要加密连接, 如果无法构建连接, 则Client端将报错

```
client: mysql -u*** -p*** --ssl-mode=REQUIRED
```

```
jdbc: jdbc:mysql://localhost:8066/testdb?requireSSL=true&useSSL=true&verifyServerCertificate=false
```

- ssl-mode=VERIFY_CA

- 单向认证

描述: Client端需要加密连接, 并且客户端会根据配置的ca证书对服务端证书进行验证

```
client: mysql -u*** -p*** --ssl-mode=VERIFY_CA --ssl-ca='${自签名CA证书}'
```

```
jdbc:
```

```

jdbc:mysql://localhost:8066/testdb?
requireSSL=true
&useSSL=true
&verifyServerCertificate=true
&trustCertificateKeyStoreUrl=file:${自签名CA证书的JKS密钥库}
&trustCertificateKeyStorePassword=${自签名CA证书的JKS密钥库的密码}

```

- 双向认证

描述: Client端需要加密连接, 客户端会根据配置的ca证书对服务端证书进行验证, 同时服务端也会验证客户端证书的有效性

```
client: mysql -u*** -p*** --ssl-mode=VERIFY_CA --ssl-ca='${自签名CA证书}' --ssl-cert='${客户端数字证书}' --ssl-key='${客户端私钥}'
```

jdbc:

```

jdbc:mysql://localhost:8066/testdb?
requireSSL=true
&useSSL=true
&verifyServerCertificate=true
&trustCertificateKeyStoreUrl=file:${自签名CA证书的JKS密钥库}
&trustCertificateKeyStorePassword=${自签名CA证书的JKS密钥库的密码}
&clientCertificateKeyStoreUrl=file:${客户端数字证书和私钥的JKS密钥库}
&clientCertificateKeyStorePassword=file:${客户端数字证书和私钥的JKS密钥库}

```

- ssl-mode=VERIFY_IDENTITY(不适用)

描述: 基于VERIFY_CA模式, 追加了证书中服务器的主机验证; 上面自签名证书不适用此模式

验证连接是否加密

- MYSQL CLIENT中, 查看当前连接的状态 (管理端连接暂时不支持此命令)

```

mysql> \s
...
SSL:          Cipher in use is DHE-RSA-AES256-SHA # 表示当前连接采用SSL方式连接
...

```

- DBLE日志

- 以下包含ssl=OpenSSL, 说明采用的OpenSSL

```

2022-05-26 11:27:55,557 [INFO ][BusinessExecutor4] FrontendConnection[id = 3 port = 8066 host = 127.0.0.1 local_port = 57752 isManager = false startupTime = 1653535675511 skipCheck = false isFlowControl = false onlyTcpConnect = false ssl = OpenSSL] SSL handshake complete (SSLHandler.java:248)

```

- 以下包含ssl=no, 说明没采用加密传输

```

2022-05-26 11:32:37,908 [INFO ][BusinessExecutor2] connection id close for reason [quit cmd] with connection FrontendConnection[id = 4 port = 8066 host = 192.168.0.109 local_port = 58114 isManager = false startupTime = 1653535957751 skipCheck = false isFlowControl = false onlyTcpConnect = false ssl = no] (AbstractConnection.java:154)

```

2.35 堆外内存泄露监控

2.35.1 介绍

支持版本：

```
>=3.22.11.0
```

简介：

dbler 使用了direct memory(堆外内存)用于加速io读写，代价是 dbler 需要额外处理堆外内存的分配和释放。如果因为程序异常可能导致堆外内存不被释放，这些堆外内存就被泄露了。

当越来越多的堆外内存被泄露，后果是堆内内存池满了，接着 dbler 启用堆内内存代替堆外内存，从而影响 dbler 的性能。

堆外内存泄露监控 功能可以用于监控堆外内存的释放和监控，便于发现泄漏点，从而可以进行相应的修复。

何时应开启：

当发现已使用的 direct memory 较大（几百MB）或者持续增长时，且在业务低峰也没有降下来。此时应该开启该监控调查增长的原因。

已使用的 direct memory 可以通过 `show @@directmemory` 命令的 `DIRECT_MEMORY_POOL_USED` 字段查看。

性能：

该功能主要用于定位故障点，不应该长期开启，当开启此功能后会存在约20%的性能下降（不同机器不一样），需要为内存、cpu预留20%的空间。

术语定义：

每一块分配的堆外内存简称为buffer。

2.35.2 原理

开启开关后，每次分配的堆外内存都会被记录在表`dbler_memory_resident`中。当该内存被回收后，表会删除相同 id 的记录。

2.35.3 bootstrap.cnf中的相关配置

```
# whether enable the memory buffer monitor
#-DenableMemoryBufferMonitor=0
#-DenableMemoryBufferMonitorRecordPool=1
```

2.35.4 管理端命令

2.35.4.1 enable @@memory_buffer_monitor

开启监控

```
mysql> enable @@memory_buffer_monitor;
Query OK, 1 row affected (4.26 sec)
```

2.35.4.2 disable @@memory_buffer_monitor

关闭监控并回收数据。

```
mysql> disable @@memory_buffer_monitor;
Query OK, 1 row affected (0.01 sec)
disable MemoryBufferMonitor success
```

2.35.4.3 select * from dbler_memory_resident \G

显示当前未回收的 buffer。该表的字段含义可查看【管理端元数据库】章节。

开启了监控该表才有效。且开关开启后的内存泄露才有可能记录，开启前的已经出现的内存泄露将永远无法被记录到。

为避免误解，buffer 存活时间不到 1s 的 不被记录在表内，这些 buffer 通常很快就被释放。

```
mysql> select * from dble_memory_resident \G
***** 1. row *****
      id: 140185807364096
  alive_second: 29.892
    stacktrace:
com.actiontech.dble.buffer.MemoryBufferMonitor.addRecord(MemoryBufferMonitor.java:80)
com.actiontech.dble.buffer.DirectByteBufferPool.allocate(DirectByteBufferPool.java:58)
com.actiontech.dble.net.connection.AbstractConnection.allocate(AbstractConnection.java:431)
com.actiontech.dble.net.connection.AbstractConnection.findReadBuffer(AbstractConnection.java:529)
com.actiontech.dble.net.connection.FrontendConnection.findReadBuffer(FrontendConnection.java:358)
com.actiontech.dble.net.impl.nio.NIOSocketWR.asyncRead(NIOSocketWR.java:358)
com.actiontech.dble.services.mysqlauthenticate.MySQLFrontAuthService.register(MySQLFrontAuthService.java:61)
com.actiontech.dble.net.connection.AbstractConnection.register(AbstractConnection.java:601)

      buffer_type: POOL
  allocate_size: 4096
  allocate_time: 2022-12-07 17:21:39.901
        sql: <<FRONT>>
1 row in set (0.00 sec)
```

2.35.5 使用方法

开启开关，这样之后的泄露将被记录。开启后维持一段时间后，观察 `buffer_type=NORMAL` 的记录（即非常驻内存），可以通过以下 sql 观察：

```
mysql> select * from dble_memory_resident where buffer_type="NORMAL"\G
Empty set (0.02 sec)
```

- 如果以上 sql 产生了结果且 `alive_second` 较小，可能是正常的业务分配，可以先等等看看记录会不会回收
- 如果以上 sql 产生了结果且 `alive_second` 较大（即 `buffer` 长时间没回收），可联系 `dble` 支持进行处理。
- 如果以上 sql 没有结果，同时这段时间内 `DIRECT_MEMORY_POOL_USED` 也没有上升，这说明这段时间泄漏点没有触发或者不存在泄露。需要多尝试不同的业务和场景来找到泄漏点。
- 如果以上 sql 没有结果，同时这段时间内 `DIRECT_MEMORY_POOL_USED` 也上升了，此时可能存在常驻内存泄露。可以发送 `dble_memory_resident` 表的结果并联系 `dble` 支持进行处理。

2.36 延迟检测

主从复制延迟检查功能，可以有效的对主从复制延迟做到监控，感知，自动化处理，来应对部分对主从数据一致性敏感的业务，防止超过容忍度的主从数据延迟导致的数据不一致问题。

主从复制延迟时间超过定义延迟时间时，dble会将该slave从dbInstance的负载均衡中剔除，直到主从复制延迟时间重新恢复到定义的延迟时间内才会再次将slave加入到dbInstance的负载均衡中。

2.36.1 延迟检测配置

启用延迟检测需要在db.xml中配置三个参数，分别为delayThreshold，delayPeriodMillis，delayDatabase，具体参数介绍可参考db.xml的章节。

```
<dbGroup name="dbGroup1" rwSplitMode="1" delayThreshold="1000" delayPeriodMillis="2000" delayDatabase="test">
  <heartbeat errorRetryCount="1" timeout="10" keepAlive="60">show slave status</heartbeat>
  <dbInstance name="instanceM1" url="ip5:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="true">
</dbInstance>

  <!-- can have multi read instances -->
  <dbInstance name="instanceS1" url="ip6:3306" user="your_user" password="your_psw" maxCon="200" minCon="50" primary="false">
    <property name="heartbeatPeriodMillis">60000</property>
  </dbInstance>
</dbGroup>
```

配置注意事项

- 1.在delayPeriodMillis，delayDatabase和delayThreshold同时配置下启用该延迟检测，如果仅配置delayThreshold，那么就使用之前的延迟检测逻辑
- 2.开启该延迟检测会导致rwStickyTime参数失效
- 3.如果主节点下发语句结果失败，那么从节点默认没有延迟

2.36.2 原理

1.dble在primary="true"的mysql中新增监控表u_delay

```
create table if not exists delaydatabase.u_delay(
source VARCHAR(256) NOT NULL,
real_timestamp varchar(26) NOT NULL,
logic_timestamp BIGINT default 0);
```

列名解释

Source :哪个dble发起的请求，命名方式为dble_dbGroupName_instanceName，instanceName为-DinstanceName参数

real_timestamp:物理时间戳，dble中的物理时间，dble的检测机制不依赖该字段，为保留字段

logic_timestamp:逻辑时间戳，dble的检测机制依赖该字段

2.以delayPeriodMillis周期向master-mysql发送replace into语句

3.以delayPeriodMillis周期向slave-mysql发送select语句

4.slave-mysql读取logic_timestamp字段的信息，并与master-mysql更新的logic_timestamp字段作比较，两者差值 $delayPeriodMillis > delayThreshold$ ，那么该slave-mysql将从dbInstance的负载均衡中剔除 两者差值 $delayPeriodMillis < delayThreshold$ ，如果该slave-mysql已经从dbInstance的负载均衡中剔除，那么重新将该slave-mysql加入到dbInstance的负载均衡中，如果该slave-mysql在dbInstance的负载均衡中，不做处理

2.36.2.1 如何检测延迟

master会以自增的方式更新logic_timestamp字段，slave会读取logic_timestamp字段的信息，如果结果有差值，那么差值*delayPeriodMillis就是延迟时间。

2.36.3 管理端介绍

delay_detection表

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
---+
| db_group_name | name | host                | delay | status | message                | last_active_time | backend_conn_id | logic_upd
ate |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
---+
| dbGroup1     | M1  | 10.186.62.41:3312 | 0     | ok    | NULL                   | 2022-12-09 15:09:45 | 293 |
0 |
| dbGroup1     | S1  | 10.186.62.41:3309 | 0     | ok    | NULL                   | 2022-12-09 15:09:45 | 295 |
0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
---+
2 rows in set (0.00 sec)
```

列描述 db_group_name: dbGroup名称

name:dbInstanceName名称

host: mysql ip和端口

delay:从节点与主节点的延迟时间

status:连接状态: ok (连接正常), error (连接异常), init(连接建立中), timeout(超时)

message: 当连接不是ok时的信息

last_active_time:最后一次延迟检测响应时间

backend_conn_id: 后端连接id

logic_update:逻辑更新次数, 仅用于重置后端连接使用

说明: 由于延迟检测连接可能在实际使用中遇到不可知的问题, 会导致延迟检测一直处于无法正常工作状态, 所以提供重置连接方式通过更新表中logic_update字段, 来重置该mysql的连接

举例:

```
update delay_detection set logic_update = 1 where backend_conn_id = 26;
```

2.37 审计日志/sql_dump log

2.37.1 介绍

在读写分离(或者分析)用户中，方便观察SQL在DBLE中具体下发到某个后端实例，耗时等信息；
局限性：分表分库用户不参与审计日志的记录

2.37.2 日志格式

[时间][SQL digest hash][SQL类型][事务ID][影响行数][用户][前端IP:端口][后端IP:端口][执行时间] SQL文本内容

- [时间]: 执行完后，日志记录的时间；精确到毫秒
- [SQL digest hash]: SQL摘要信息的hash值
- [SQL类型]: SQL类型；一般有: Insert、Update、Delete、Select、Show、DDL、Begin、Commit、Rollback等，其余为Other
- [事务ID]: 是一根前侧连接级别的事务累加的计数器；同一根前侧连接中，事务ID相同的语句视为同一事务内的语句
- [影响行数]: 返回的行数
- [用户]: 用户名称(:租户名称)
- [前端IP:端口]: 客户端的IP和PORT
- [后端IP:端口]: 用到的后端连接对应的物理库所在的IP和PORT
- [执行时间]: SQL执行耗时；以毫秒为单位
- SQL文本内容: 截取SQL的1024长度用于展示

2.37.3 相关配置

```
-DenableSqlDumpLog=0          # 开关, 0-关闭(默认), 1-开启
-DsqlDumpLogBasePath=sqldump # base路径
-DsqlDumpLogFileName=sqldump.log # 日志文件名, 生成日志的相对路径: sqldump/sqldump.log
-DsqlDumpLogCompressFilePattern=${date:yyyy-MM}/sqldump-%d{MM-dd}-%i.log.gz ## 压缩日志文件命名格式, 如: sqldump/2022-10/sqldump-10-11-1.log.gz (精度为: 天)
-DsqlDumpLogOnStartupRotate=1 # 每次重启, 是否触发日志翻转; 1-是, 0-否
-DsqlDumpLogSizeBasedRotate=50MB # 当sqldump.log文件大小达到50MB, 触发日志翻转; 单位可以为:KB、MB、GB
-DsqlDumpLogTimeBasedRotate=1 # 间隔天数, 触发日志翻转; 如: 1, 则每天会进行反转
-DsqlDumpLogDeleteFileAge=90d # 对过期90天内的压缩文件进行删除, 单位可以为:d(天)、h(时)、m(分)、s(秒); (注意, 精度单位需要与sqlDumpLogCompressFilePattern保持一致, 否则可能不生效; 原因: 基于log4j2的实现, 存在这个现象)
-DsqlDumpLogCompressFilePath=*/sqldump-*.log.gz # 过期的文件压缩匹配
```

2.37.4 相关命令

```
enable @@sqldump_sql; -- 开启
disable @@sqldump_sql; -- 关闭
select * from dble_variables where variable_name like '%sqldump%'; -- 查询sqldump相关的变量
```

2.37.5 其他

- 0、SQL在DBLE中执行，但没有实际下发到后端实例，则不参与记录
- 1、Exit、Quit 语句不参与记录
- 3、COM_STMT_PREPARE 不参与统计，COM_STMT_EXECUTE参与统计
- 4、执行多语句，也参与记录
- 5、DBLE内部语法依赖于Druid，若Druid不支持的语法且需要脱敏处理，因此暂时无法展示sql模版，“SQL文本内容”部分均由“Other”代替展示
- 6、审计日志基于log4j2的实现

2.38 tcp相关参数

2.38.1 介绍

db1e在启动时会创建与mysql通信的连接池，但是这些连接可能在tcp三次握手时由于某些情况丢失了第三次握手的信息。此时这些连接被mysql直接丢弃，而在db1e侧并无感知，导致db1e的连接池的连接的数量达不到预期设置的值，导致db1e出现异常。为了解决这个问题，linux本身就有keep-alive机制，并提供对应参数控制，但是这是系统级别参数，一旦直接修改可能影响其他应用影响面较大。db1e在bootstrap.cnf中提供相关参数配置并且只影响db1e内的连接。

2.38.2 jdk版本要求

oracle jdk 1.8-261及其以上版本

openjdk 1.8-272及其以上版本

2.38.3 相关配置

```
-DtcpKeepIdle=30 #启用tcp-keepalive连接检测的连接空闲时间
-DtcpKeepInterval=10 #启用tcp-keepalive连接检测的时间间隔
-DtcpKeepCount=3 #启用tcp-keepalive检测启用后的最大重试次数
```

2.38.4 使用场景

- 1.检测并关闭tcp三次握手中丢失了第三次握手信息导致的半连接
- 2.检测挂掉的连接，如导致连接挂掉的原因很多，如服务停止、网络波动、宕机、应用重启等
- 3.防止因为网络不活动而断连，如使用NAT代理或者防火墙等

2.38.5 注意事项

- 1.db1e会在启动的时候校验jdk版本，如果版本不符合要求db1e会给出相应warn日志，并且配置文件的这三个参数不会生效，db1e可以正常启动
- 2.该功能属于试用功能，可能导致后端连接异常断开，可通过调大tcpKeepIdle，tcpKeepInterval，tcpKeepCount的值来规避问题

2.39 HTAP 快速上手指南

2.39.1 介绍

随着MySQL发展为世界上最为流行的开源数据库系统，用户在其中存储了大量的数据，并且运行着关键的业务逻辑，对这些数据进行实时分析成为一个日益增长的需求。当MySQL不能满足需求时，用户寻求一个更好的解决方案。

DBLE采用的方案是在只有以OLTP为主的MySQL数据库的基础上，引入处理OLAP型需求擅长的Clickhouse数据库，两套数据库系统可实现TP/AP负载的完全隔离，并通过数据同步工具等进行数据的实时同步；另外DBLE会将前端发送过来的业务SQL根据SQL类型等规则自动将TP型负载路由到MySQL上，而将AP型负载路由到Clickhouse上，实现TP/AP负载自动识别下发

2.39.2 配置

1. user.xml：配置HTAP型需求的用户hybridTAUser

```
<hybridTAUser name="htap_user1" password="111111" schemas="testdb"/>
```

2. sharding.xml：配置hybridTAUser对应的schema

```
<!--这里apNode指向的Clickhouse的节点，shardingNode指向的MySQL的节点-->
<schema name="testdb" apNode="apNode1" shardingNode="dn1">
  ....
</schema>
<shardingNode name="dn1" dbGroup="dbGroup1" database="db_1"/>
<apNode name="apNode1" dbGroup="dbGroup2" database="ap_db1"/>
```

3. db.xml：配置HTAP对应的MySQL和Clickhouse节点

```
<!--MySQL-->
<dbGroup name="dbGroup1" rwSplitMode="0" delayThreshold="100">
  <heartbeat>select 1</heartbeat>
  <dbInstance name="hostM1" url="ip:port" user="root" password="123456" maxCon="100" minCon="10" primary="true"/>
</dbGroup>
<!--Clickhouse-->
<dbGroup name="dbGroup2" rwSplitMode="0" delayThreshold="100">
  <heartbeat>select 1</heartbeat>
  <dbInstance name="hostM2" url="ip:port" user="default" password="123456" maxCon="100" minCon="10" primary="true" databaseType="clickhouse"/>
</dbGroup>
```

4. 数据同步：自行配置，这里推荐Clickhouse官方提供的以binlog方式实现的[MaterializedMySQL方案](#)

配置限制

1. shardingUser对应的schema中不能配置apNode
2. hybridTAUser对应的schema中必须配置apNode
3. hybridTAUser涉及apNode对应的dbGroup的databaseType必须是clickhouse
4. analysisUser的dbGroup可以与hybridTAUser的apNode共用同一个
5. shardingNode和apNode的name不能重复

2.39.3 使用案例

前提：MySQL和Clickhouse数据库中数据是同步的，数据都为：

```
mysql> select * from table_1;
+-----+
| id | column1 |
+-----+
| 1 | abc     |
| 2 | def     |
| 3 | ghi     |
+-----+
3 rows in set (0.01 sec)
```

case 1: OLTP

#由于SQL未包含聚合函数等AP的规则，所以被判定为OLTP负载，也就拆分下发给了dn节点对应的MySQL数据库

```
mysql> explain select id from table_1;
+-----+-----+-----+
| SHARDING_NODE | TYPE      | SQL/REF          |
+-----+-----+-----+
| dn1           | BASE SQL  | select id from table_1 |
| dn2           | BASE SQL  | select id from table_1 |
| dn3           | BASE SQL  | select id from table_1 |
| dn4           | BASE SQL  | select id from table_1 |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> select id from table_1;
+----+
| id |
+----+
| 3  |
| 1  |
| 2  |
+----+
3 rows in set (0.01 sec)
```

case 2: OLAP

#由于SQL包含group by语法，所以被判定为OLAP负载，也就下发给了ap1节点对应的Clickhouse数据库

```
mysql> explain select id from table_1 group by id;
+-----+-----+-----+
| SHARDING_NODE | TYPE      | SQL/REF          |
+-----+-----+-----+
| ap1           | BASE SQL  | select id from table_1 group by id |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select id from table_1 group by id;
+-----+
| id  |
+-----+
| 3  |
| 2  |
| 1  |
+-----+
3 rows in set (0.01 sec)
```

2.39.4 具体说明

SQL路由

支持智能路由和手动路由

智能路由

根据SQL类型、事务等情况判定SQL下发给AP或TP:

1. 包含聚合函数的SELECT需要下发给AP
具体指的是: min、sum、count、avg、max、STDDEV_POP、STDDEV_SAMP、VAR_POP、VAR_SAMP、group by
2. 事务
ClickHouse不支持事务，所以针对事务的情况都需要发给TP
3. 增删改
根据数据同步机制，Clickhouse使用binlog同步mysql数据，可以认为ck作为从，mysql作为主，这样的结构也就限制DML只能下发给TP

手动路由

前侧业务已经区分是AP/TP型，TP业务和AP业务区分不同的应用程序，不同的应用程序通过不同的用户连接到DBLE来实现业务隔离:

纯OLTP型应用: 可使用shardingUser (见[1.3 user.xml 配置](#))

纯OLAP型应用: 可使用analysisUser (见[1.3 user.xml 配置](#))

拆分粒度

应用发送给DBLE的SQL (称之为完整SQL)，经过DBLE解析后可能会被拆分成多个SQL (称之为子SQL)，后续会按照路由规则把这些SQL下发给具体的后端节点

这里DBLE在选择SQL进行智能路由时，会针对子SQL进行智能判断并下发给对应的TP/AP型的数据库

查看配置

- 查看hybridTAUser的配置

```
mysql> select * from dble_entry a join dble_entry_schema b on a.id = b.id where user_type = 'hybridTAUser'\G
***** 1. row *****
      id: 4
      type: username
      user_type: hybridTAUser
      username: apuser1
password_encrypt: hwj/raQ08POPSUzykAbUnQVvzw11IdbQw4fbZxmocw71BW6Y0He0Z0nIZfRkXsbQ4KMPegG4D2KkQnwZpbYmpA==
encrypt_configured: false
      conn_attr_key: NULL
      conn_attr_value: NULL
      white_ips: NULL
      readonly: -
      max_conn_count: no limit
      blacklist: NULL
      id: 4
      schema: aptest
1 row in set (0.01 sec)
```

- 查看apNode的配置

```
mysql> select * from dble_ap_node;
+-----+-----+-----+
| name | db_group | db_schema |
+-----+-----+-----+
| ap1  | dbGroup4 | mysql_db_test2 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

2.39.5 不支持场景

- 跨Clickhouse库的SQL目前未做支持，只能路由到MySQL数据库
- 由于Clickhouse在支持隐式UNION语法上（[Clickhouse UNION Clause](#)）和MySQL不兼容，所以使用UNION时应该显式的写明是UNION ALL还是UNION DISTINCT，不然下发给Clickhouse会报错

2.40 dble中线程处理(print、kill、recover)

目前dble内部维护的主要以下几个线程：

```
mysql> select * from dble_thread_pool;
+-----+-----+-----+-----+-----+
| name          | pool_size | core_pool_size | active_count | waiting_task_count |
+-----+-----+-----+-----+-----+
| Timer         | 1         | 1              | 0           | 0                 |
| TimerScheduler | 2         | 2              | 0           | 15                |
| frontWorker   | 8         | 8              | 1           | 0                 |
| managerFrontWorker | 4         | 4              | 4           | 0                 |
| backendWorker | 8         | 8              | 0           | 0                 |
| complexQueryWorker | 8         | 8              | 1           | 0                 |
| writeToBackendWorker | 8         | 8              | 8           | 0                 |
| NIOFrontRW    | 8         | 8              | 8           | 0                 |
| NIOBackendRW  | 8         | 8              | 8           | 0                 |
+-----+-----+-----+-----+-----+

mysql> select * from dble_thread_pool_task;
+-----+-----+-----+-----+-----+-----+
| name          | pool_size | active_task_count | task_queue_size | completed_task | total_task |
+-----+-----+-----+-----+-----+-----+
| Timer         | 1         | 0               | 0             | 89208         | 89208     |
| TimerScheduler | 2         | 0               | 15            | 1158830       | 1158845   |
| frontWorker   | 8         | 0               | 0             | 0             | 0         |
| managerFrontWorker | 4         | 0               | 0             | 60            | 60        |
| backendWorker | 8         | 0               | 0             | 11339         | 11339     |
| complexQueryWorker | 8         | 1               | 0             | 2189          | 2190     |
| writeToBackendWorker | 8         | 0               | 0             | 0             | 0         |
+-----+-----+-----+-----+-----+-----+
7 rows in set (0.42 sec)
```

补充（在非性能模式下）：

- TimerScheduler，作为dble内部定时任务的调度器；
 - 线程数固定(core_pool_size)2个，搭配无界队列；活动时由定时任务触发周期而定，active_count有时为0；
 - 定时任务个数见task_queue_size；有时task_queue_size会减少，说明部分任务正在执行中；
 - 负责几个重要的定时任务包括心跳检测、主从延迟检测、疑似残留xid检测。
 - 如果此连接池中2个线程都hang了，completed_task不变，task_queue_size表示任务个数会持续增长
- Timer，异步参与执行TimerScheduler线程中做一些较为复杂任务；
 - 核心线程为1个，预备线程数1个，搭配有界队列(固定大小65535);pool_size初始时为1，当队列满时，pool_size会增大为2；
 - 如果此连接池中2个线程都hang了，则pool_size、active_count均为2，completed_task、total_task不再变，task_queue_size值为65535。
- frontWorker，负责8066前端连接处理请求的线程池；
 - 线程数固定，且为常驻线程（active_count等于pool_size），线程长期均处于RUNNING状态
 - 如果active_count小于pool_size，可以尝试使用thread @@recover name='0-frontWorker'；
- managerFrontWorker，负责9066前端连接处理请求的线程池；
 - 同frontWorker
- writeToBackendWorker，广播下发SQL时候批量处理的线程池
 - 同 frontWorker
- NIOFrontRW，进行前端网络IO吞吐的线程数
 - 同frontWorker
- NIOBackendRW，进行后端网络IO吞吐的线程数
 - 同frontWorker
- backendWorker，后端业务处理线程
 - 线程数固定，但非常驻线程；业务需要处理时，active_count会+1，但不超过pool_size
- complexQueryWorker，负责处理一些复杂逻辑或者；
 - 线程数不固定，且非常驻线程；业务需要处理时，active_count会+1，pool_size会出现超过core_pool_size
- 其他线程
 - 只能通过 thread @@print 或者jstack等方式观测

2.40.1 dble中观察疑似hang的线程

2.40.1.1 内部检测

局限：仅对Timer、TimerScheduler线程&线程池的检测

dble单独起一个内部观测的"ThreadChecker"线程，固定间隔2min检测一次。

检测机制：

1. 若某个任务执行的时间长达10s -- thread.log中会有提示 Thread[{}] suspected hang, execute time:[{}ms] more than 10s, currentState:[{}]
2. 若本次线程池距离上一次检测周期所保留的快照对比active_task_count、completed_task没有发生变化，则thread.log中会有提示 The thread pool where the thread[{}] is located is in the hang state and cannot work. Trigger alarm ,且会有DBLE_THREAD_SUSPECTED_HANG告警

3. 若不满足第2点了，则DBLE_THREAD_SUSPECTED_HANG告警会自动解决

2.40.1.2 通过print线程堆栈的方式，人为判断

`thread @@print` 打印所有堆栈信息

`thread @@print name=?` 指定单个线程打印堆栈信息

强调注意：这两个命令类似于jstack工具，都会有STW的风险；因此建议不要轻易执行print命令

2.40.2 当确定某个线程执行任务的过长或者hang了，dble还可以哪些手段可以补救？

`thread @@kill name=?`，用于中断某个线程正在执行任务

`thread @@kill poolname=?`，仅支持TimerScheduler/Timer线程池中中断

`thread @@recover name=?`，一般对常驻线程的恢复操作

`thread @@recover poolname=?`，仅支持TimerScheduler/Timer线程池恢复

强调注意：不要随意执行以上命令

具体场景&尝试补救

- TimerScheduler
 - 场景1：当发现存在单个线程hang了（多个时间点这个线程的堆栈没有变化）
 - 尝试补救：
 1. 执行 `thread @@kill name=?` 尝试将执行中的任务中断；
 2. 继续观察这个线程后续的堆栈是否有发生变化，若发生了变化了，则视为解决hang问题；若没有发生变化，则表示此线程大概率陷入死循环，则只能暂时接受剩余正常线程继续工作。
 - 场景2：当发现池中所有线程都hang了（多个时间点的所有堆栈不再变化）且completed_task不再继续增长
 - 尝试补救：
 1. 执行 `thread @@kill poolname=?` 将整个线程池内部进行shutdown操作（一些重要的心跳检测、主从延迟检测等任务也会停掉，在恢复之前可以通过查看心跳信息发现所有的实例中的STOP均为true，所有实例均不可用）；
 2. 若pool_size为0，表示连接池shutdown成功；继续执行 `thread @@recover poolname=?` 命令，以及观察pool_size恢复原来的初始值，completed_task、total_task均在重新增长（重要的心跳检测、主从延迟检测等任务也会恢复）。
 3. 若pool_size为2保持不变，则表示此线程池可能陷入死循环，只能通过重启解决此现象。
 - 注意：`thread @@kill name='0-TimerScheduler'` + `thread @@kill name='1-TimerScheduler'` 不等于 `thread @@kill poolname='TimerScheduler'`
- Timer
 - 与TimerScheduler类似处理
- frontWorker
 - 场景1：当发现存在单个线程hang了（多个时间点这个线程的堆栈没有变化）
 - 尝试补救：
 1. 执行 `thread @@kill name=?` 尝试将执行中的任务中断；
 2. 若active_count值不变，后续单独观察此线程后续堆栈是否有变化，若堆栈有变化表示已解决此线程hang问题，若堆栈无变化，则表示此线程可能陷入死循环了，在影响面较大的情况下只能通过重启解决，影响面较小的情况，可暂时接受。
 3. 若active_count值减少，则继续执行 `thread @@recover name=?`，观察active_count值+1（注意，这里如果name为'0-frontWorker'，实际上恢复的可能并不是'0-frontWorker'，具体原因与实现机制有关，这里不做阐述）。
 - **强烈建议：**不要将所有的常驻线程，都被kill，否则会引起dble不可正常使用。
- managerFrontWorker
 - 与frontWorker类似
- writeToBackendWorker
 - 与frontWorker类似
- NIOFrontRW，进行前端网络IO吞吐的线程数
 - 不支持kill/recover命令（因为kill掉NIOFrontRW会影响已经建立的前侧连接的业务执行）
- NIOBackendRW，进行后端网络IO吞吐的线程数
 - 1.与frontWorker类似
 - 2.在做recover之后，dble中的旧连接池中的连接不再由此RW维护了；因此需要做些对后端连接的刷新操作，比如：`fresh conn forced where dbGroup='groupName'`用来刷新连接池 或者执行 `reload @@config_all -r` 重置连接池。
 - 注意，当kill掉这个线程时，dble中内部与后端连接会受到影响：比如心跳不正常、执行sql有时hang有时can't reach等 凡事与后端实例交互的都会收到牵连，因此，不要随意使用kill命令。
- backendWorker
 - 场景1：当发现存在单个线程hang了（多个时间点这个线程的堆栈没有变化），
 - 尝试补救：
 1. 执行 `thread @@kill name=?` 尝试将执行中的任务中断；
 2. 继续单独观察此线程后续堆栈是否有变化，若堆栈有变化表示已解决此线程hang问题，若堆栈无变化，则表示此线程可能陷入死循环了，在影响面较大的情况下只能通过重启解决，影响面较小的情况，可暂时接受。
 3. 此线程无需执行 `thread @@recover` 命令。
- complexQueryWorker
 - 与backendWorker类似
- 其他线程，可通过jstack等工具观测
 - 与backendWorker类似，但建议不要轻易 `thread @@kill name=?` 命令

以上线程检测、print、kill、recover操作均会记录到logs/therad.log中，可自行演练观察日志。

3.语法兼容

- 3.1 DDL
 - 3.1.1 DDL&Table Syntax
 - 3.1.2 DDL&View Syntax
 - 3.1.3 DDL&Index Syntax
 - 3.1.4 DDL透传
 - 3.1.5 DDL&Database Syntax
 - 3.1.6 ONLINE DDL
- 3.2 DML
 - 3.2.1 INSERT
 - 3.2.2 REPLACE
 - 3.2.3 DELETE
 - 3.2.4 UPDATE
 - 3.2.5 SELECT
 - 3.2.6 SELECT JOIN syntax
 - 3.2.7 SELECT UNION Syntax
 - 3.2.8 SELECT Subquery Syntax
 - 3.2.9 LOAD DATA
 - 3.2.10 不支持的DML语句
- 3.3 Prepared SQL Syntax
- 3.4 Transactional and Locking Statements
 - 3.4.1 一般事务语法
 - 3.4.2 含隐式提交语句
 - 3.4.3 SAVEPOINT语法
 - 3.4.4 Lock&unlock语法
 - 3.4.5 SET TRANSACTION Syntax
 - 3.4.6 XA 事务语法
 - 3.4.7 其它不支持的事务语法
- 3.5 DAL
 - 3.5.1 SET
 - 3.5.2 SHOW
 - 3.5.3 KILL
 - 3.5.4 不支持的DAL语句
- 3.6 存储过程支持方式
- 3.7 Utility Statements
- 3.8 Hint
- 3.9 其他不支持语句
- 3.10 函数与操作符支持列表(alpha版本)
- 3.11 导入导出方式

3.1 DDL

DDL 包含以下几部分内容

注意，当DDL执行时，涉及到的相同的表上的DML和DDL操作将会报错。

- [3.1.1 DDL&Table Syntax](#)
- [3.1.2 DDL&View Syntax](#)
- [3.1.3 DDL&Index Syntax](#)
- [3.1.4 DDL透传](#)

3.1.1 TABLE DDL

3.1.1.1 CREATE TABLE Syntax

```

CREATE TABLE [IF NOT EXISTS] tbl_name
  (create_definition,...)
  [table_options]
  [partition_options]

create_definition:
  col_name column_definition

column_definition:
  data_type [NOT NULL | NULL] [DEFAULT default_value]
  [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
  [COMMENT 'string']

data_type:
  BIT[(length)]
| TINYINT[(length)] [UNSIGNED] [ZEROFILL]
| SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
| MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
| INT[(length)] [UNSIGNED] [ZEROFILL]
| INTEGER[(length)] [UNSIGNED] [ZEROFILL]
| BIGINT[(length)] [UNSIGNED] [ZEROFILL]
| REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
| DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
| FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
| DECIMAL[(length[,decimals])] [UNSIGNED] [ZEROFILL]
| NUMERIC[(length[,decimals])] [UNSIGNED] [ZEROFILL]
| DATE
| TIME[(fsp)]
| TIMESTAMP[(fsp)]
| DATETIME[(fsp)]
| YEAR
| CHAR[(length)]
| VARCHAR(length)
| BINARY[(length)]
| VARBINARY(length)
| TINYBLOB
| BLOB
| MEDIUMBLOB
| LONGBLOB
| TINYTEXT
| TEXT
| MEDIUMTEXT
| LONGTEXT
| ENUM(value1,value2,value3,...)

table_options:
  table_option [[,] table_option] ...

table_option:
  ENGINE [=] engine_name
| [DEFAULT] CHARACTER SET [=] charset_name
| CHECKSUM [=] {0 | 1}
| [DEFAULT] COLLATE [=] collation_name
| COMMENT [=] 'string'
| CONNECTION [=] 'connect_string'
| KEY_BLOCK_SIZE [=] value
| MAX_ROWS [=] value
| MIN_ROWS [=] value
| PASSWORD [=] 'string'
| ROW_FORMAT [=] {DEFAULT|DYNAMIC|FIXED|COMPRESSED|REDUNDANT|COMPACT}
| STATS_AUTO_RECALC [=] {DEFAULT|0|1}
| STATS_PERSISTENT [=] {DEFAULT|0|1}

partition_options:
  {[LINEAR] HASH(expr)
  | PARTITION BY [linear] KEY (column_list)
  | RANGE{(expr) | COLUMNS(column_list)}
  | LIST{(expr) | COLUMNS(column_list)}
  }
  [(partition_definition [, partition_definition] ...)]

```

注意：

- `engine_name`仅能为忽略大小写的“InnoDB”
- 不建议含有枚举类型的表作为分片表，比如表结构：`CREATE TABLE `test` (`id` enum('1','2','3') DEFAULT '1')`。因为此种表在插入id列时，既可以使用枚举值插入，也可以使用枚举值的下标，'1'的下标是1，以此类推。若用户以枚举值进行分片，但是插入时确使用枚举值下标，因为dble不会将下标转换为枚举值，所以分片会出现问题，详细可参考issue：<https://github.com/actiontech/dble/issues/816>。

例：

```
create table if not exists test(
id bigint primary key AUTO_INCREMENT,
col1 int not null default 5,
col2 int null COMMENT 'info for col1',
col3 varchar(20) not null,
col4 varchar(20) unique key
);

create table test(
id int primary key,
col_bit BIT(1),
col_tinyint TINYINT(2) UNSIGNED ZEROFILL,
col_smallint SMALLINT(3) UNSIGNED ZEROFILL,
col_mediumint MEDIUMINT(4) UNSIGNED ZEROFILL,
col_int INT(5) UNSIGNED ZEROFILL,
col_integer INTEGER(6) UNSIGNED ZEROFILL,
col_bigint BIGINT(7) UNSIGNED ZEROFILL,
col_real REAL(8,1) UNSIGNED ZEROFILL,
col_double DOUBLE(9,2) UNSIGNED ZEROFILL,
col_float FLOAT(10,3) UNSIGNED ZEROFILL,
col_decimal DECIMAL(11,4) UNSIGNED ZEROFILL,
col_numeric NUMERIC(12,5) UNSIGNED ZEROFILL,
col_date DATE,
col_time TIME(3),
col_timestamp TIMESTAMP(4),
col_datetime DATETIME(5),
col_year YEAR,
col_char CHAR(10) ,
col_varcgar VARCHAR(20) ,
col_binary BINARY(30),
col_varbinary VARBINARY(40),
col_tinyblob TINYBLOB,
col_blob BLOB,
col_mediumblob MEDIUMBLOB,
col_longblob LONGBLOB,
col_tinytext TINYTEXT ,
col_text TEXT ,
col_mediumtext MEDIUMTEXT ,
col_longtext LONGTEXT ,
col_enum ENUM('a','b','c')
);
```

或者

```
create table test(
id int primary key,
col1 varchar(20)
)ENGINE = innodb
AVG_ROW_LENGTH = 20
DEFAULT CHARACTER SET = utf8
CHECKSUM = 1
DEFAULT COLLATE = utf8_general_ci
COMMENT = 'info of table test'
CONNECTION = '111111'
DELAY_KEY_WRITE = 1
INSERT_METHOD = LAST
KEY_BLOCK_SIZE = 65536
MAX_ROWS = 3
MIN_ROWS = 2
PACK_KEYS = 1
ROW_FORMAT = DEFAULT;
```

3.1.1.2 ALTER TABLE Syntax

```
ALTER [IGNORE] TABLE tbl_name
    [alter_specification [, alter_specification] ...]

alter_specification:
| ADD [COLUMN] col_name column_definition
    [FIRST | AFTER col_name ]
| ADD [COLUMN] (col_name column_definition,...)
| ADD {INDEX | KEY} [index_name]
| CHANGE [COLUMN] old_col_name new_col_name column_definition
    [FIRST|AFTER col_name]
| MODIFY [COLUMN] col_name column_definition
    [FIRST | AFTER col_name]
| DROP [COLUMN] col_name
| DROP {INDEX | KEY} index_name
| ADD [INDEX|KEY] [index_name] (index_col_name,...)
| DROP {INDEX|KEY} index_name
| ADD PRIMARY KEY (index_col_name,...)
| DROP PRIMARY KEY
| ALTER [COLUMN] col_name
    {SET DEFAULT {literal | (expr)} | DROP DEFAULT}
| COMMENT [=] 'string'
```

例:

```
alter table test add column col5 int not null default 1 first,add column col6 int after col4;
alter table test change column col1 col1_new int after col3;
alter table test modify column col1_new varchar(20) after id;
alter table test drop column col6;
alter table test add key idx_col4(col4);
alter table test add index idx_col4(col4);
alter table test drop key idx_col4;
alter table test drop index idx_col4;
alter table test drop primary key;
alter table test add primary key (id);
alter table test alter column col set default 0;
alter table test alter column col drop default;
alter table test comment = 'string';
```

3.1.1.3 DROP TABLE Syntax

```
DROP TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

例:

```
drop table if exists test cascade;
drop table test restrict;
```

3.1.1.4 TRUNCATE TABLE Syntax

```
TRUNCATE [TABLE] tbl_name
```

例:

```
truncate table test;
```

3.1.2 VIEW DDL

Syntax

create view :

```
CREATE [OR REPLACE] VIEW  
    view_name [(column_list)]  
    AS select_statement
```

alter view :

```
alter VIEW  
    view_name [(column_list)]  
    AS select_statement
```

drop view:

```
DROP VIEW [IF EXISTS] view_name [, view_name]
```

show create view :

```
SHOW CREATE VIEW view_name;
```

3.1.3 INDEX DDL

3.1.3.1 CREATE INDEX Syntax

```
CREATE [UNIQUE|FULLTEXT] INDEX index_name
  [index_type]
  ON tbl_name (index_col_name,...)

index_col_name:
  col_name [(length)] [ASC | DESC]

index_type:
  USING {BTREE | HASH}
```

例:

```
create unique index idx1 using btree on test(col1);
create index idx2 using hash on test(col2);
create fulltext index idx3 on test(col4);
create fulltext index idx4 on test(col4(10));
```

3.1.3.2 DROP INDEX Syntax

```
DROP INDEX index_name ON tbl_name
```

例:

```
drop index idx1 on test;
```

3.1.4 DDL透传

除了前面章节中的DDL语句，dble在非注解方式方式下不支持mysql中的其它DDL语句(eg. ALTER EVENT)。但通过如下的注解方式，dble支持所有的mysql中的DDL语句：

```
/*!dble:sql=select ... from tbx where id=M*/ ddl statement;
```

其中，tbx为分区表，id为分区列，M为分区列的某个值。

例如：

```
MySQL [TESTDB]> /*!dble:sql=select * from a_test where id=2*/CREATE PROCEDURE account_count()  
  
    BEGIN    SELECT 'Number of accounts:', COUNT(*) FROM mysql.user;  
  
END//
```

3.1.5 DATABASE DDL

3.1.5.1 CREATE DATABASE Syntax

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name  
    [create_specification] ...
```

create_specification:

```
    [DEFAULT] CHARACTER SET [=] charset_name  
    | [DEFAULT] COLLATE [=] collation_name  
    | DEFAULT ENCRYPTION [=] {'Y' | 'N'}
```

注意:

- schema一定要在shema.xml中配置
- create_specification 中的选项不生效
- 只具有语法意义，没有实际语义的作用，需要建库请用管理端口对应的命令

3.1.6 ONLINE DDL

3.1.6.1 背景

- 在3.20.04.0或之前版本的dble中执行ddl，需要加表锁，若是在集群中，也会通知其他dble加上表锁，加锁期间执行对应表的SQL会报错。在ddl执行结束后dble会下发show create table 得到建表语句，得到建表语句后会解析重新生成元数据，集群中的其他dble也会执行此操作。
- 但一部分SQL其实对dble不造成影响，对dble造不造成影响的判断依据在于是否需要更改dble中表的元数据，目前dble中表的元数据只关心表的列名称，列类型，类是否为null。若这样的sql在mysql侧也是onlineDDL的，可以无需加锁，直接下发执行，例如增加索引的操作。

3.1.6.2 ONLINE DDL种类

以下穷举了MySQL 8.0中所有的online ddl以及是否在dble中支持online ddl模式。

- [Index Operations](#)
- [Primary Key Operations](#)
- [Column Operations](#)
- [Generated Column Operations](#)
- [Foreign Key Operations](#)
- [Table Operations](#)
- [Tablespace Operations](#)
- [Partitioning Operations](#)

序号	类别	操作	语句	直接下发?	备注
1	索引操作	创建或者新增二级索引	CREATE INDEX <i>name</i> ON <i>table</i> (<i>col_list</i>);ALTER TABLE <i>tbl_name</i> ADD INDEX <i>name</i> (<i>col_list</i>);	是	涉及到表的事务结束后才会执行该操作。操作期间，支持表的并发读写。
2		删除索引	DROP INDEX <i>name</i> ON <i>table</i> ;ALTER TABLE <i>tbl_name</i> DROP INDEX <i>name</i> ;	是	涉及到表的事务结束后才会执行该操作。操作期间，支持表的并发读写。
3		索引重命名	ALTER TABLE <i>tbl_name</i> RENAME INDEX <i>old_index_name</i> TO <i>new_index_name</i> , ALGORITHM=INPLACE, LOCK=NONE;	否	mysql 5.6 该操作不是online ddl
4		添加 FULLTEXT 索引	CREATE FULLTEXT INDEX <i>name</i> ON <i>table</i> (<i>column</i>);	否	有限制，具体参见mysql官方文档
5		添加 SPATIAL 索引	CREATE TABLE <i>geom</i> (<i>g</i> GEOMETRY NOT NULL); ALTER TABLE <i>geom</i> ADD SPATIAL INDEX(<i>g</i>), ALGORITHM=INPLACE, LOCK=SHARED;	否	有限制，具体参见mysql官方文档
6		修改索引类型	ALTER TABLE <i>tbl_name</i> DROP INDEX <i>i1</i> , ADD INDEX <i>i1</i> (<i>key_part</i> ,...) USING BTREE, ALGORITHM=INSTANT;	是	
7	主键操作	新增主键	ALTER TABLE <i>tbl_name</i> ADD PRIMARY KEY (<i>column</i>), ALGORITHM=INPLACE, LOCK=NONE;	否	主键操作过程中都需要重建表数据，代价较昂贵
8		删除主键	ALTER TABLE <i>tbl_name</i> DROP PRIMARY KEY, ALGORITHM=COPY;	否	该操作只允许COPY操作，过程中不支持并发ddl
9		修改主键	ALTER TABLE <i>tbl_name</i> DROP PRIMARY KEY, ADD PRIMARY KEY (<i>column</i>), ALGORITHM=INPLACE, LOCK=NONE;	否	主键的确定在初期就应该确认好，避免在后期修改
10	列操作	新增列	ALTER TABLE <i>tbl_name</i> ADD COLUMN <i>column_name</i> <i>column_definition</i> ,	否	dble需要重新生成表元数据

			ALGORITHM=INSTANT;		
11		删除列	ALTER TABLE <i>tbl_name</i> DROP COLUMN <i>column_name</i> , ALGORITHM=INPLACE, LOCK=NONE;	否	dbtle需要重新生成表元数据
12		重命名列	ALTER TABLE <i>tbl</i> CHANGE <i>old_col_name</i> <i>new_col_name</i> <i>data_type</i> , ALGORITHM=INPLACE, LOCK=NONE;	否	dbtle需要重新生成表元数据
13		列重排序	ALTER TABLE <i>tbl_name</i> MODIFY COLUMN <i>col_name</i> <i>column_definition</i> FIRST, ALGORITHM=INPLACE, LOCK=NONE;	否	
14		修改列数据类型	ALTER TABLE <i>tbl_name</i> CHANGE <i>c1</i> <i>c1</i> BIGINT, ALGORITHM=COPY;	否	执行期间，不允许dml
15		加长 VARCHAR 长度	ALTER TABLE <i>tbl_name</i> CHANGE COLUMN <i>c1</i> <i>c1</i> VARCHAR(255), ALGORITHM=INPLACE, LOCK=NONE;	否	涉及到dbtle的表元数据
16		给列设置默认值	ALTER TABLE <i>tbl_name</i> ALTER COLUMN <i>col</i> SET DEFAULT <i>literal</i> , ALGORITHM=INSTANT;	是	
17		删除列的默认值	ALTER TABLE <i>tbl</i> ALTER COLUMN <i>col</i> DROP DEFAULT, ALGORITHM=INSTANT;	是	
18		修改自增值	ALTER TABLE <i>table</i> AUTO_INCREMENT= <i>next_value</i> , ALGORITHM=INPLACE, LOCK=NONE;	否	若是分片节点，给每个节点设置成相同的自增值没有意义
19		列可以为NULL	ALTER TABLE <i>tbl_name</i> MODIFY COLUMN <i>column_name</i> <i>data_type</i> NULL, ALGORITHM=INPLACE, LOCK=NONE;	否	涉及到dbtle的表元数据
20		列不可以为NULL	ALTER TABLE <i>tbl_name</i> MODIFY COLUMN <i>column_name</i> <i>data_type</i> NOT NULL, ALGORITHM=INPLACE, LOCK=NONE;	否	涉及到dbtle的表元数据
21		修改 ENUM 或 SET列的定义	CREATE TABLE <i>t1</i> (<i>c1</i> ENUM('a', 'b', 'c')); ALTER TABLE <i>t1</i> MODIFY COLUMN <i>c1</i> ENUM('a', 'b', 'c', 'd'), ALGORITHM=INSTANT;	否	不常用
22	外键操作	新增外键	ALTER TABLE <i>tbl1</i> ADD CONSTRAINT <i>fk_name</i> FOREIGN KEY <i>index(col1)</i> REFERENCES <i>tbl2(col2)</i> <i>referential_actions</i> ;	否	dbtle中的表会跨多实例，表与表之间建立外键无实际意义
23		删除外键	ALTER TABLE <i>tbl</i> DROP FOREIGN KEY <i>fk_name</i> ;		
24	Generated列操作	新增 Generated 列	ALTER TABLE <i>t1</i> ADD COLUMN (<i>c2</i> INT GENERATED	否	相当于新增一列

24	Generated列操作	新增 Generated 列	ALWAYS AS (c1 + 1) STORED), ALGORITHM=COPY;	否	相当于新增一列
25		修改 Generated 列顺序	ALTER TABLE t1 MODIFY COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) STORED FIRST, ALGORITHM=COPY;		
26		删除 Generated 列	ALTER TABLE t1 DROP COLUMN c2, ALGORITHM=INPLACE, LOCK=NONE;		
27		新增 VIRTUAL 列	ALTER TABLE t1 ADD COLUMN (c2 INT GENERATED ALWAYS AS (c1 + 1) VIRTUAL), ALGORITHM=INSTANT;		
28		修改 VIRTUAL 列顺序	ALTER TABLE t1 MODIFY COLUMN c2 INT GENERATED ALWAYS AS (c1 + 1) VIRTUAL FIRST, ALGORITHM=COPY;		
29		删除 VIRTUAL 列	ALTER TABLE t1 DROP COLUMN c2, ALGORITHM=INSTANT;		
30	表空间操作			否	更偏向MySQL运维测
31	表分区操作			否	dble 分库分表的功能不涉及表分区，不需要支持

3.1.6.3 限制

1. dble 使用阿里的druid作为解析器，因此有些online ddl语句的“ALGORITHM=INPLACE, LOCK=NONE”不被支持，详情参考：<https://github.com/alibaba/druid/issues/3750>

3.2 DML

DML 包含以下内容

- [3.2.1 INSERT](#)
- [3.2.2 REPLACE](#)
- [3.2.3 DELETE](#)
- [3.2.4 UPDATE](#)
- [3.2.5 SELECT](#)
- [3.2.6 SELECT JOIN syntax](#)
- [3.2.7 SELECT UNION Syntax](#)
- [3.2.8 SELECT Subquery Syntax](#)
- [3.2.9 LOAD DATA](#)
- [3.2.10 不支持的DML语句](#)

3.2.1 INSERT

3.2.1.1 Syntax

正常情况下，INSERT只支持下面2种写法。

```

INSERT
  [INTO] tbl_name
  [(col_name [, col_name] ...)]
  { {VALUES | VALUE} (value_list) [, (value_list)] ... }
  [ON DUPLICATE KEY UPDATE assignment_list]

INSERT
  [INTO] tbl_name
  SET assignment_list
  [ON DUPLICATE KEY UPDATE assignment_list]

value:
  {expr | DEFAULT}

value_list:
  value [, value] ...

assignment:
  col_name =
    value

assignment_list:
  assignment [, assignment] ...

```

特殊情况下，支持下面这种写法。特殊场景参看下文细节

```

INSERT
  [INTO] tbl_name
  [(col_name [, col_name] ...)]
  { SELECT ...
    | TABLE table_name
  }
  [ON DUPLICATE KEY UPDATE assignment_list]

assignment:
  col_name =
    value

assignment_list:
  assignment [, assignment] ...

```

3.2.1.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```

INSERT
- [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
  [(col_name [, col_name] ...)]
  { {VALUES | VALUE} (value_list) [, (value_list)] ... }
- [AS row_alias[(col_alias [, col_alias] ...)]]
  [ON DUPLICATE KEY UPDATE assignment_list]

```

```

INSERT
- [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
  SET assignment_list
- [AS row_alias[(col_alias [, col_alias] ...)]]
  [ON DUPLICATE KEY UPDATE assignment_list]

```

```

INSERT
- [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
  [(col_name [, col_name] ...)]
  { SELECT ...
    | TABLE table_name
  - | VALUES row_constructor_list
  }
  [ON DUPLICATE KEY UPDATE assignment_list]

```

```

value:
  {expr | DEFAULT}

```

```

value_list:
  value [, value] ...

```

```

-row_constructor_list:
- ROW(value_list)[, ROW(value_list)][, ...]

```

```

assignment:
  col_name =
    value
- | [row_alias.]col_name
- | [tbl_name.]col_name
- | [row_alias.]col_alias

```

```

assignment_list:
  assignment [, assignment] ...

```

3.2.1.3 举例

```

insert into test (col1,col3) values(1,'cust1'),(2,'cust2');
insert into test (col1,col3) values(default,'cust3');
insert into test set col1=4,col3='cust4';
insert into test set col1=default,col3='cust5';
insert into test (col1,col3) values(default,cast(now() as char));

```

3.2.1.4 限制

- 在插入ER关系的子表时，每个语句只允许插入一个ROW
- 全局序列在插入时不允许指定值，全部由dble序列生成
- 对于含有枚举类型的分片表，比如表结构：CREATE TABLE `test` (`id` enum('1','2','3') DEFAULT '1')，在插入id列时，既可以使用枚举值插入，也可以使用枚举值的下标，'1'的下标是1，以此类推。若用户以枚举值进行分片，但是插入时确使用枚举值下标，因为dble不会将下标转换为枚举值，所以分片会出现问题，详细可参考issue：<https://github.com/actiontech/dble/issues/816>。
- 存在特例，当 insert/replace... select 语句满足以下条件时，dble会在确保数据安全性的情况下对于SQL进行下发执行
 - 当插入目标是单节点表时，要求所有数据来源的表格都有明确的路由信息路由到同一节点
 - 当插入目标是全局表时，要求所有的数据来源表格都是全局表，并且路由范围能够覆盖插入目标
 - 当插入目标是分片表时，要求分片列的数据直接来自拥有同样分片逻辑的分片表，并且对于select子查询中间的其他表格，要求能够子查询部分路由结果能整体下发，并且逻辑上无错误

3.2.2 REPLACE

3.2.2.1 Syntax

正常情况下，REPLACE只支持下面2种写法。

```
REPLACE
  [INTO] tbl_name
  [(col_name [, col_name] ...)]
  {VALUES | VALUE} (value_list) [, (value_list)] ...
```

```
REPLACE
  [INTO] tbl_name
  SET assignment_list
```

```
value:
  {expr | DEFAULT}
```

```
value_list:
  value [, value] ...
```

```
assignment:
  col_name = value
```

```
assignment_list:
  assignment [, assignment] ...
```

特殊情况下，支持下面这种写法。特殊场景参看下文细节

```
REPLACE
  [INTO] tbl_name
  [(col_name [, col_name] ...)]
  {SELECT ... | TABLE table_name}
```

3.2.1.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```

REPLACE
- [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
  [(col_name [, col_name] ...)]
- {
  {VALUES | VALUE} (value_list) [, (value_list)] ...
- |VALUES row_constructor_list
- }

REPLACE
- [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
  SET assignment_list

REPLACE
- [LOW_PRIORITY | DELAYED]
  [INTO] tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
  [(col_name [, col_name] ...)]
  {SELECT ... | TABLE table_name}

value:
  {expr | DEFAULT}

value_list:
  value [, value] ...

-row_constructor_list:
- ROW(value_list)[, ROW(value_list)][, ...]

assignment:
  col_name = value

assignment_list:
  assignment [, assignment] ...

```

3.2.2.3 举例

```

REPLACE INTO test VALUES (1, 'Old', '2014-08-20 18:47:00');
REPLACE INTO test set id = 1, type= 'Old',create_date = '2014-08-20 18:47:00';

```

3.2.2.4 限制

- 由于replace的语义为如果存在则替换，如果不存在则新增，所以在使用表格自增主键的时候，如果对于自增表格使用replace且ID不存在，那么就会插入一条指定ID的数据，并不会自动生成ID
- 存在特例，当 insert/replace... select 语句满足以下条件时，dble会在确保数据安全性的情况下对于SQL进行下发执行
 - 当插入目标是单节点表时，要求所有数据来源的表格都有明确的路由信息路由到同一节点
 - 当插入目标是全局表时，要求所有的数据来源表格都是全局表，并且路由范围能够覆盖插入目标
 - 当插入目标是分片表时，要求分片列的数据直接来自拥有同样分片逻辑的分片表，并且对于select子查询中间的其他表格，要求能够子查询部分路由结果能整体下发，并且逻辑上无错误

3.2.3 DELETE

3.2.3.1 Single-Table Syntax

3.2.3.1.1 Syntax

```
DELETE
  FROM tbl_name [[AS] tbl_alias]
  [WHERE where_condition]
```

3.2.3.1.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
DELETE
- [LOW_PRIORITY] [QUICK] [IGNORE]
  FROM tbl_name [[AS] tbl_alias]
- [PARTITION (partition_name [, partition_name] ...)]
  [WHERE where_condition]
- [ORDER BY ...]
- [LIMIT row_count]
```

3.2.3.1.3 举例

```
delete from test where id>5;
```

3.2.3.1.4 限制

- 原则上Delete语句中的where_condition部分只允许出现简单的条件，不能支持计算表达式以及子查询

3.2.3.2 Multiple-Table Syntax

- 原则上不支持多表Join 的DELETE
- 存在特例，当DELETE满足以下判断条件时，部分复杂DELETE语句会在确保数据正确的情况下被下发执行
 - 包括update/delete语句操作多表的时候，【操作的全是全局表，并拥有同样的分片范围，并且where条件中不含有子查询】或【所有操作的表都有条件显式路由到同一个节点，并且where条件不含有子查询】
 - 包括update/delete语句操作单表，但是where条件中包含子查询时，【被操作的表格是单节点表，where条件中的所有表格都有条件能路由到同一个节点】或【被操作的表格是全局表或者分片表，其余所有表格都是全局表，并且分片范围都能覆盖被操作的表格（需注意“分片范围”指的是 配置文件里面配的表分片范围，而不是经过 where 二次筛选后的范围。）】

3.2.3.2.1 Syntax

```
DELETE
  tbl_name[*] [, tbl_name[*]] ...
  FROM table_references
  [WHERE where_condition]

DELETE
  FROM tbl_name[*] [, tbl_name[*]] ...
  USING table_references
  [WHERE where_condition]
```

3.2.3.2.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
DELETE
- [LOW_PRIORITY] [QUICK] [IGNORE]
  tbl_name[*] [, tbl_name[*]] ...
  FROM table_references
  [WHERE where_condition]

DELETE
- [LOW_PRIORITY] [QUICK] [IGNORE]
  FROM tbl_name[*] [, tbl_name[*]] ...
  USING table_references
  [WHERE where_condition]
```


3.2.4 UPDATE

3.2.4.1 Single-Table Syntax

3.2.4.1.1 Syntax

```
UPDATE
  table_reference
  SET assignment_list
  [WHERE where_condition]

value:
  {expr | DEFAULT}

assignment:
  col_name = value

assignment_list:
  assignment [, assignment] ...
```

3.2.4.1.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
UPDATE
- [LOW_PRIORITY] [IGNORE]
  table_reference
  SET assignment_list
  [WHERE where_condition]
- [ORDER BY ...]
- [LIMIT row_count]

value:
  {expr | DEFAULT}

assignment:
  col_name = value

assignment_list:
  assignment [, assignment] ...
```

3.2.4.1.3 举例

```
UPDATE test SET VALUE =1 where id=5;
```

3.2.4.1.4 限制

- 原则上UPDATE语句中的where_condition部分只允许出现简单的条件，不能支持计算表达式以及子查询

3.2.4.2 Multiple-Table Syntax

- 原则上不支持多表Join 的UPDATE
- 存在特例，当update满足以下判断条件时，部分复杂UPDATE语句会在确保数据正确的情况下被下发执行
 - 包括update语句操作多表的时候，【操作的全是全局表，并拥有同样的分片范围，并且where条件中不含有子查询】或【所有操作的表都有条件显式路由到同一个节点，并且where条件不含有子查询】
 - 包括update语句操作单表，但是where条件中包含子查询时，【被操作的表格是单节点表，where条件中的所有表格都有条件能路由到同一个节点】或【被操作的表格是全局表或者分片表,其余所有表格都是全局表，并且分片范围都能覆盖被操作的表格（需注意“分片范围”指的是 配置文件里面配的表分片范围，而不是经过 where 二次筛选后的范围。）】
- 支持多表情况下的整体下发：where条件包含所有表的分片字段，并且存在ER关系，此时SQL是可以整体下发到多个节点
- 支持部分场景下的update多表：
 - 目前只支持两张表的update，一张表为更新的表，另一张表为查询的表，查询的表支持子查询的形式
 - 支持更改同一张表的多个字段，不支持更改多个表的字段
 - set、where不包含子查询、表达式

3.2.4.2.1 Syntax

```
UPDATE
  table_references
  SET assignment_list
  [WHERE where_condition]
```

3.2.4.2.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

UPDATE

```
- [LOW_PRIORITY] [IGNORE]
  table_references
  SET assignment_list
  [WHERE where_condition]
```

3.2.5 SELECT

3.2.5.1 Syntax

```

SELECT
  [ALL | DISTINCT | DISTINCTROW ]
  select_expr [, select_expr] ...
  [FROM table_references]
  [WHERE where_condition]
  [GROUP BY {col_name | expr }, ... ]
  [HAVING where_condition]
  [ORDER BY {col_name | expr }
   [ASC | DESC], ... ]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  [FOR {UPDATE | SHARE}
   [NOWAIT | SKIP LOCKED]
   | LOCK IN SHARE MODE]

```

3.2.5.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```

SELECT
  [ALL | DISTINCT | DISTINCTROW ]
  - [HIGH_PRIORITY]
  - [STRAIGHT_JOIN]
  - [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
  - [SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
  select_expr [, select_expr] ...
  - [into_option]
  [FROM table_references
  - [PARTITION partition_list]
  ]
  [WHERE where_condition]
  [GROUP BY {col_name | expr
  - | position
  }, ...
  - [WITH ROLLUP]
  ]
  [HAVING where_condition]
  - [WINDOW window_name AS (window_spec)
  - [, window_name AS (window_spec)] ...]
  [ORDER BY {col_name | expr
  - | position
  } [ASC | DESC], ...
  - [WITH ROLLUP]
  ]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  - [into_option]
  [FOR {UPDATE | SHARE}
  - [OF tbl_name [, tbl_name] ...]
  [NOWAIT | SKIP LOCKED]
  | LOCK IN SHARE MODE]
  - [into_option]

-into_option: {
- INTO OUTFILE 'file_name'
- [CHARACTER SET charset_name]
- export_options
- | INTO DUMPFILE 'file_name'
- | INTO var_name [, var_name] ...
-}

```

3.2.5.3 举例

```
select id,col1,col3 from test where id=3;
select distinct col1,col3 from test where id>=3;
select count(*),max(id),col1 from test group by col1 desc having(count(*)>1) order by col1 desc;
select id,col1,col3 from test order by id limit 2 offset 2;
select id,col1,col3 from test order by id limit 2,2;
select 1+1,'test',id,col1*1.1,now() from test limit 3;
select current_date,current_timestamp;
select * from test where id=3 for update skip locked;
select * from test where id=3 for share;
select * from test where id=3 LOCK IN SHARE MODE;
```

3.2.6 JOIN Syntax:

本节主要描述 SELECT 语句和多表的 DELETE 以及 UPDATE 语句中的 table_references 部分里面的 JOIN 语法

3.2.6.1 Syntax

```
table_references:
    table_reference [, table_reference] ...

table_reference: {
    table_factor
    | joined_table
}

table_factor: {
    tbl_name [[AS] alias]
    | table_subquery [AS] alias
    | ( table_references )
}

joined_table: {
    table_reference {[INNER | CROSS] JOIN | STRAIGHT_JOIN} table_factor [join_specification]
    | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification
    | table_reference NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN table_factor
}

join_specification: {
    ON search_condition
    | USING (join_column_list)
}

join_column_list:
    column_name [, column_name] ...
```

注意：这里table_subquery的问题，会在后续章节单独展开讨论

3.2.6.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```

table_references:
    escaped_table_reference [, escaped_table_reference] ...

escaped_table_reference: {
    table_reference
- | { OJ table_reference }
}

table_reference: {
    table_factor
| joined_table
}

table_factor: {
    tbl_name
- [PARTITION (partition_names)]
    [[AS] alias]
- [index_hint_list]
|
- [LATERAL]
    table_subquery [AS] alias
- [(col_list)]
| ( table_references )
}

joined_table: {
    table_reference {[INNER | CROSS] JOIN | STRAIGHT_JOIN} table_factor [join_specification]
| table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification
| table_reference NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN table_factor
}

join_specification: {
    ON search_condition
| USING (join_column_list)
}

join_column_list:
    column_name [, column_name] ...

-index_hint_list:
- index_hint [, index_hint] ...

-index_hint: {
- USE {INDEX|KEY}
- [FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
- | {IGNORE|FORCE} {INDEX|KEY}
- [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
-}

-index_list:
- index_name [, index_name] ...

```

3.2.7 UNION Syntax:

与MYSQL语法完全兼容

```
query_block UNION [ALL | DISTINCT] query_block  
  [UNION [ALL | DISTINCT] query_block]  
  [...]
```

3.2.8 Subquery

3.2.8.1 The Subquery as Scalar Operand

支持标量子查询

For example :

```
SELECT (SELECT s2 FROM t1);
SELECT (SELECT s1 FROM t2) FROM t1;
SELECT UPPER((SELECT s1 FROM t1)) FROM t2;
```

3.2.8.2 Comparisons Using Subqueries

支持子查询用于比较

子查询最常用的形式是:

```
non_subquery_operand comparison_operator (subquery)
```

其中 comparison_operator 是这些运算符之一:

```
= > < >= <= <> != <=>
```

MySQL 也允许这种结构:

```
non_subquery_operand LIKE (subquery)
```

3.2.8.3 Subqueries with ANY, IN, or SOME

支持此类子查询 Syntax:

```
operand comparison_operator ANY (subquery)
operand IN (subquery)
operand comparison_operator SOME (subquery)
```

其中 comparison_operator 是这些运算符之一:

```
= > < >= <= <> !=
```

3.2.8.4 Subqueries with ALL

支持此类子查询 Syntax:

```
operand comparison_operator ALL (subquery)
```

3.2.8.5 Subqueries with EXISTS or NOT EXISTS

支持此类子查询

For example:

```
SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

3.2.8.6 Derived Tables (Subqueries in the FROM Clause)

部分支持此类子查询

For example:

```
SELECT ... FROM (subquery) [AS] tbl_name ...
```

不支持下面的形式:

```
SELECT * FROM JSON_TABLE(arg_list) [AS] tbl_name ...
SELECT ... FROM (subquery) [AS] tbl_name (col_list) ...
```

3.2.8.7 Row Subqueries

不支持行子查询

3.2.8.8 Correlated Subqueries

不支持相关子查询和相关子查询与其他子查询混合的方式

3.2.9 LOAD DATA

3.2.9.1 Syntax

```
LOAD DATA
  [LOCAL]
  INFILE 'file_name'
  [REPLACE | IGNORE]
  INTO TABLE tbl_name
  CHARACTER SET 'charset_name'
  [{FIELDS | COLUMNS}
   [TERMINATED BY 'string']
   [[OPTIONALLY] ENCLOSED BY 'char']
   [ESCAPED BY 'char']
  ]
  [LINES
   [STARTING BY 'string']
   [TERMINATED BY 'string']
  ]
  [IGNORE number {LINES }]
  [(col_name_or_user_var
   [, col_name_or_user_var] ...)]
  [SET col_name={expr | DEFAULT}
   [, col_name={expr | DEFAULT}] ...]
```

3.2.9.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字，红绿两色表示稍微不相同的地方

```
LOAD DATA
- [LOW_PRIORITY | CONCURRENT]
  [LOCAL]
  INFILE 'file_name'
  [REPLACE | IGNORE]
  INTO TABLE tbl_name
- [PARTITION (partition_name [, partition_name] ...)]
- [CHARACTER SET charset_name]
+ CHARACTER SET 'charset_name'
  [{FIELDS | COLUMNS}
   [TERMINATED BY 'string']
   [[OPTIONALLY] ENCLOSED BY 'char']
   [ESCAPED BY 'char']
  ]
  [LINES
   [STARTING BY 'string']
   [TERMINATED BY 'string']
  ]
  [IGNORE number {
    LINES
- | ROWS
  }]
  [(col_name_or_user_var
   [, col_name_or_user_var] ...)]
  [SET col_name={expr | DEFAULT}
   [, col_name={expr | DEFAULT}] ...]
```

3.2.9.3 举例

```
load data infile 'data.txt' into table test_table CHARACTER SET 'utf8mb4' FIELDS TERMINATED by ',';
```

3.2.9.4 原理

db1e解析MySQL协议之后，会根据数据路由拆分文件，每满足maxRowSizeToFile(可通过bootstrap.cnf配置)就写到文件中，再通过 load data local infile 的方式导入到后端结点。

所以，这里local_infile这个参数会影响到load data的正确性。

可参考[#1085](#)

3.2.9.5 限制

- 存在BUG导致在db1e中CHARACTER SET charset_name必填
- 由于 db1e 依赖的 druid 解析器的限制，charset_name必须单引号包括。比如 CHARACTER SET 'utf8mb4'，不可使用CHARACTER SET "utf8mb4" 或者 CHARACTER SET utf8mb4
- 在mysql中如果插入的数据不符合规范会插入部分数据，相对来说db1e的load data对正确性有更高的要求，一个错误的发生会导致整体操作的回滚

- 在ENCLOSED BY的时候存在BUG，使用的的时候会导致转义的数据无法被正确转义存储到数据库中
- 由于当前解析设定的限制，loaddata默认每列最大字节数65535，可通过bootstrap.cnf中的maxCharsPerColumn配置修改
- 使用load data导入数据时,若导入表是分片表,应保证导入文件中分片键数据符合分片规则的要求,否则dble将会报错.详细参考issue:<https://github.com/actiontech/dble/issues/770>
- load data语句需要严格按照语法书写,由于druid解析器的缘故,若语法或关键字错误,druid会解析出错误的语句,从而导致结果错误.详细请参照issue:<https://github.com/actiontech/dble/issues/1248>
- load data语句读字段的时候，如果遇到行结束符，会认为是本行结束了，需要注意:<https://github.com/actiontech/dble/issues/1507>
- load data中使用用户变量后，再次查询用户变量的值是不正确的，可参考issue：<https://github.com/actiontech/dble/issues/1761>

3.2.10 不支持的DML语句

3.2.10.1 以下语句不支持，点击链接会跳转到MySQL官方文档

[DO Statement](#)

[EXCEPT Clause](#)

[HANDLER Statement](#)

[IMPORT TABLE Statement](#)

[INTERSECT Clause](#)

[LOAD XML Statement](#)

[Parenthesized Query Expressions](#)

[TABLE Statement](#)

[VALUES Statement](#)

[WITH \(Common Table Expressions\)](#)

3.2.10.2 部分支持的语句

1. [Set Operations with UNION, INTERSECT, and EXCEPT](#)

目前dble只支持UNION

1. [CALL Statement](#)

存储过程支持方式参见[3.6_procedure_support](#)

3.3 Prepared Statements

3.3.1 PREPARE Statement

完全兼容MySQL

```
PREPARE stmt_name FROM preparable_stmt
```

例:

```
prepare stmt1 from "select * from a_test where id=?";
```

3.3.2 EXECUTE Statement

完全兼容MySQL

```
EXECUTE stmt_name  
[USING @var_name [, @var_name] ...]
```

例:

```
SET @a = 1;  
EXECUTE stmt1 USING @a;
```

3.3.3 DEALLOCATE PREPARE Statement

完全兼容MySQL

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

例:

```
DROP PREPARE stmt1;
```

3.4 Transactional and Locking Statements

Transactional and Locking Statements 包含以下内容

- [3.4.1 一般事务语法](#)
- [3.4.2 含隐式提交语句](#)
- [3.4.3 SAVEPOINT 语法](#)
- [3.4.4 Lock&unlock语法](#)
- [3.4.5 SET TRANSACTION Syntax](#)
- [3.4.6 XA 事务语法](#)
- [3.4.7 其它不支持的事务语法](#)

3.4.1 START TRANSACTION, COMMIT, and ROLLBACK Statements

3.4.1.1 Syntax

开启事务

```
START TRANSACTION

BEGIN

SET autocommit = {0 | 1}
```

提交事务

```
COMMIT
```

回滚事务

```
ROLLBACK
```

3.4.1.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
START TRANSACTION
- [transaction_characteristic [, transaction_characteristic] ...]

-transaction_characteristic: {
-   WITH CONSISTENT SNAPSHOT
-   | READ WRITE
-   | READ ONLY
-}

BEGIN
-[WORK]

COMMIT
-[WORK] [AND [NO] CHAIN] [[NO] RELEASE]

ROLLBACK
-[WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET autocommit = {0 | 1}
```

3.4.1.3 限制

- 2PC实现的分布式事务(非xa方式)可能会出现commit时部分提交的情况,如需保障最终一致性,需要开启XA

3.4.2 Implicit commit SQL

3.4.2.1 含隐式提交语句

注意：执行含隐式提交语句出现1064,1046错误码，则隐式提交并不会实际生效。

3.4.2.2 sharding用户中，隐式提交支持度比较有限，具体如下：

- [3.1.1 DDL&Table Syntax](#)
- [3.1.2 DDL&View Syntax](#)
- [3.1.3 DDL&Index Syntax](#)
- Lock tables...

3.4.2.3 rwsplit用户中，隐式提交支持度几乎同mysql: [见详请](#)

3.4.3 SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Syntax

3.4.3.1 Syntax

语法与MySQL完全相同

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

3.4.2.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

3.4.3.2 举例

```
# start transaction
set autocommit = 0;

# savepoint
savepoint s0;
insert into test value(1);
savepoint s1;
insert into test value(2);
savepoint s2;
insert into test value(3);

# rollback to
rollback to s0

# release
release savepoint s0
```

3.4.3.3 限制

1. 在mysql中,在事务外也可以定义savepoint,但是这些savepoint是没有意义的.因此在dble中savepoint强制在事务中使用,否则报错.
2. ROLLBACK TO [SAVEPOINT] *identifier* 语句暂不支持 work 可选项.

3.4.4 Lock&unlock

3.4.4.1 Syntax

```
LOCK TABLES
  tbl_name [[AS] alias] lock_type

lock_type: {
  READ | WRITE
}

UNLOCK TABLES
```

3.4.4.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
LOCK TABLES
  tbl_name [[AS] alias] lock_type
-  [, tbl_name [[AS] alias] lock_type] ...

lock_type: {
  READ
-  [LOCAL]
  |
-  [LOW_PRIORITY]
  WRITE
}

UNLOCK TABLES
```

3.4.4.3 举例

```
lock tables test_table read;
unlock tables;
```

3.4.4.4 限制

1. 当前session加锁后访问其他表可能不会被阻止或者报错。
2. 加写锁后，复杂查询可能不会返回正确结果。

3.4.5 SET TRANSACTION Syntax

3.4.5.1 Syntax

```
SET SESSION TRANSACTION ISOLATION LEVEL level
```

```
level: {
  REPEATABLE READ
  | READ COMMITTED
  | READ UNCOMMITTED
  | SERIALIZABLE
}
```

也支持

```
SET @@SESSION.TX_ISOLATION = 'level_str'
```

```
level_str:
  REPEATABLE-READ
  | READ-COMMITTED
  | READ-UNCOMMITTED
  | SERIALIZABLE
```

3.4.5.2 与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
SET
- [GLOBAL |
  SESSION
- ]
  TRANSACTION
  transaction_characteristic
- [, transaction_characteristic] ...

transaction_characteristic: {
  ISOLATION LEVEL level
- | access_mode
}

level: {
  REPEATABLE READ
  | READ COMMITTED
  | READ UNCOMMITTED
  | SERIALIZABLE
}

-access_mode: {
-   READ WRITE
-   | READ ONLY
-}
```

注：因为隔离级别不加session关键字语义不同，作用域为下一个语句，目前暂不支持不加session

3.4.6 XA 事务语法

db1e不支持外部XA事务，以下讨论内容是对于后端结点之间使用xa事务的问题

3.4.6.1 Syntax

开启XA

```
set xa = {0|1}
```

开启事务

```
START TRANSACTION;  
  
BEGIN  
  
SET autocommit = {0 | 1}
```

提交事务

```
COMMIT
```

回滚事务

```
ROLLBACK
```

3.4.6.2 限制

- xa事务中不支持含有隐式提交的sql

3.4.7 其他不支持的事务语法

[LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE Statements](#)

3.5 DAL

DAL主要包含以下内容

- [3.5.1 SET](#)
- [3.5.2 SHOW](#)
- [3.5.3 KILL](#)
- [3.5.4 不支持的DAL 语句](#)

3.5.1 SET语句

3.5.1.1 XA

```
set xa=value

value:
  0
  | off
  | false
  | 1
  | on
  | true
```

例:

```
set xa=1;
```

注意事项: XA设置不能在多变量设置语句中使用。

3.5.1.2 AUTOCOMMIT

```
set autocommit=value

value:
  0
  | off
  | false
  | 1
  | on
  | true
```

例:

```
set autocommit=1;
```

注意事项: AUTOCOMMIT设置不能在多变量设置语句中使用。

3.5.1.3 NAMES

```
SET NAMES {'charset_name' [COLLATE 'collation_name'] | DEFAULT}
```

例:

```
set names utf8;

set names utf8 collate utf8_general_ci;

set names default;
```

3.5.1.4 CHARSET

```
SET {CHARACTER SET | CHARSET}
{'charset_name' | DEFAULT}
```

例:

```
set CHARACTER SET utf8;
```

3.5.1.5 COLLATION_CONNECTION/CHARSET_SET_X

```
SET COLLATION_CONNECTION='collation_name'

SET CHARSET_SET_CLIENT='charset_name',

SET CHARSET_SET_RESULTS='charset_name' 其中, 'charset_name' 可以为NULL。

SET CHARSET_SET_CONNECTION='charset_name'
```

例:

```
set collation_connection=utf8_general_ci;

set CHARSET_SET_CLIENT=utf8;

set CHARSET_SET_RESULTS=utf8;

set CHARSET_SET_CONNECTION=utf8;
```

3.5.1.6 TRANSACTION ACCESS MODE

```
SET SESSION { TX_READ_ONLY | TRANSACTION_READ_ONLY}=value
```

```
value:
  0
  | off
  | false
  | 1
  | on
  | true
```

例:

```
set session @@tx_read_only=1;
```

注意:

在MySQL中, 支持在事务设置只读, 但是不影响当前事务, 由于dble的事务模型不同, 因此可能会影响当前事务。

3.5.1.7 TRANSACTION ISOLATION LEVEL

```
SET SESSION {TRANSACTION_ISOLATION | TX_ISOLATION}=level
```

```
level:
```

```
READ-UNCOMMITTED | READ-COMMITTED | REPEATABLE-READ | SERIALIZABLE
```

例:

```
SET SESSION TX_ISOLATION=READ-COMMITTED;
```

3.5.1.8 USER/SYSTEM VARIABLE

```
SET variable_assignment[, variable_assignment ] ...
```

```
variable_assignment:
```

```
  @user_var_name = expr

  | SESSION system_var_name = expr

  | system_var_name = expr

  | @@system_var_name = expr

  | @@session.system_var_name = expr
```

注意事项:

1. 不能设置全局系统变量。
2. 支持的系统变量为:

```
audit_log_current_session
audit_log_filter_id
auto_increment_increment
auto_increment_offset
autocommit
big_tables
binlog_direct_non_transactional_updates
binlog_error_action
binlog_format
binlog_row_image
```



```
binlog_rows_query_log_events
binlogging_impossible_mode
block_encryption_mode
bulk_insert_buffer_size
character_set_client
character_set_connection
character_set_database
character_set_filesystem
character_set_results
character_set_server
collation_connection
collation_database
collation_server
completion_type
debug
debug_sync
default_storage_engine
default_tmp_storage_engine
default_week_format
disconnect_on_expired_password
div_precision_increment
end_markers_in_json
eq_range_index_dive_limit
error_count
explicit_defaults_for_timestamp
external_user
foreign_key_checks
group_concat_max_len
gtid_next
gtid_owned
identity
innodb_create_intrinsic
innodb_ft_user_stopword_table
innodb_lock_wait_timeout
innodb_optimize_point_storage
innodb_strict_mode
innodb_support_xa
innodb_table_locks
innodb_tmpdir
insert_id
interactive_timeout
join_buffer_size
keep_files_on_create
last_insert_id
lc_messages
lc_time_names
lock_wait_timeout
long_query_time
low_priority_updates
max_allowed_packet
max_delayed_threads
max_error_count
max_execution_time
max_heap_table_size
max_insert_delayed_threads
max_join_size
max_length_for_sort_data
max_seeks_for_key
max_sort_length
max_sp_recursion_depth
max_statement_time
max_user_connections
min_examined_row_limit
mysam_repair_threads
mysam_sort_buffer_size
mysam_stats_method
ndb-allow-copying-alter-table
ndb_autoincrement_prefetch_sz
ndb-blob-read-batch-bytes
ndb-blob-write-batch-bytes
ndb_deferred_constraints
ndb_force_send
ndb_fully_replicated
ndb_index_stat_enable
ndb_index_stat_option
ndb_join_pushdown
ndb_log_bin
ndb_log_bin
```

```
ndb_table_no_logging
ndb_table_temporary
ndb_use_copying_alter_table
ndb_use_exact_count
ndb_use_transactions
ndbinfo_max_bytes
ndbinfo_max_rows
ndbinfo_show_hidden
ndbinfo_table_prefix
net_buffer_length
net_read_timeout
net_retry_count
net_write_timeout
new
old_alter_table
old_passwords
optimizer_prune_level
optimizer_search_depth
optimizer_switch
optimizer_trace
optimizer_trace_features
optimizer_trace_limit
optimizer_trace_max_mem_size
optimizer_trace_offset
parser_max_mem_size
preload_buffer_size
profiling
profiling_history_size
proxy_user
pseudo_slave_mode
pseudo_thread_id
query_alloc_block_size
query_cache_type
query_cache_wlock_invalidate
query_prealloc_size
rand_seed1
rand_seed2
range_alloc_block_size
range_optimizer_max_mem_size
rbr_exec_mode
read_buffer_size
read_rnd_buffer_size
session_track_gtids
session_track_schema
session_track_state_change
session_track_system_variables
show_old_temporals
sort_buffer_size
sql_auto_is_null
sql_big_selects
sql_buffer_result
sql_log_bin
sql_log_off
sql_mode
sql_notes
sql_quote_show_create
sql_safe_updates
sql_select_limit
sql_warnings
storage_engine
thread_pool_high_priority_connection
thread_pool_prio_kickup_timer
time_zone
timestamp
tmp_table_size
transaction_alloc_block_size
transaction_allow_batching
transaction_prealloc_size
transaction_write_set_extraction
tx_isolation
tx_read_only
unique_checks
updatable_views_with_limit
version_tokens_session
version_tokens_session_number
wait_timeout
warning_count
```

例:

```
set @a=20;

SET SESSION sql_mode = 'TRADITIONAL';

SET sql_mode = 'TRADITIONAL';
```

1. insert_id 在使用过程中可能会在另一个前端连接中被重置而导致主键冲突的问题。sql_auto_is_null 和 insert_id 是联合使用的,使用时也有此限制。详情参见 issue: <https://github.com/actiontech/dble/issues/1252>.

3.5.1.9 TRACE

用于观察单条SQL的性能,打开此开关后,可以执行需要观察性能查询语句,然后执行 `show trace` 来观察最后结果。

可以用 `select @@trace` 观察当前的开启状态。详情请见 [单条SQL性能trace](#)

```
set trace=value

value:
  0
  | off
  | false
  | 1
  | on
  | true
```

例:

```
set trace=1;
```

3.5.2 SHOW语句

3.5.2.1 dble劫持的SHOW

- SHOW DATABASES
将sharding.xml 中的所有schema展示出来。
- SHOW CREATE DATABASE [IF NOT EXISTS] schema
将sharding.xml 中的指定schema的创建语句展示，创建语句为dble伪造，无实际意义。
- SHOW [FULL|ALL] TABLES [FROM db_name] [LIKE 'pattern'| WHERE expr]
当schema没有配置默认节点时，将schema下配置的tables直接展示出来。
当schema有默认节点时，将语句转发至默认节点，然后将结果集与schema下配置的tables做一个去重合并，再返回给客户端。
- SHOW ALL TABLES [FROM db_name] [LIKE 'pattern'| WHERE expr]
dble自有命令，与SHOW FULL TABLES 返回结果集类似，不同之处是Table_type这列分为了 SHARDING TABLE, sharding table, GLOBAL TABLE。参见 [6.Differernce_from_MySQL_Server.md](#)。
- SHOW [FULL] {COLUMNS | FIELDS} FROM tbl_name [{FROM|IN} db_name] [LIKE 'pattern' | WHERE expr]
将逻辑schema转为物理schema之后下发到表所在的任意节点。
- SHOW { INDEX | INDEXES | KEYS} {FROM | IN} tbl_name [{FROM | IN} db_name] [WHERE expr]
将逻辑schema转为物理schema之后下发到表所在的任意节点。
- SHOW CREATE TABLE tbl_name
将逻辑schema转为物理schema之后下发到表所在的任意节点。
- SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern' | WHERE expr]
随机转发到任意节点，收到结果集后，用本地变量进行覆盖（global 不正确？）
- SHOW CREATE VIEW view_name
将dble层面的view展示出来
- SHOW CHARSET
将之转为show character set 之后透传转发
- SHOW TABLE STATUS [{FROM | IN} db_name] [LIKE 'pattern' | WHERE expr]
只是为了支持SQLyog，其中name列逻辑和show tables一致，其他列均为伪造。
- SHOW TRACE
观察trace结果，详情请见 [单条SQL性能trace](#)

注意事项：

所有以上命令的explain结果可能不准确。

例：

```
show databases;
show full tables;
show columns from a_test;
show index from a_test;
show create table a_test;
show variables;
show charset;
```

3.5.2.2 dble透传的SHOW

除了dble劫持的特定SHOW语句外，其它SHOW语句都透传，这些语句与MySQL语法相同。

例：

```
SHOW CHARACTER SET;
SHOW CHARACTER SET like 'utf8';
SHOW CHARACTER SET where maxlen=2;
```

3.5.3 KILL

完全支持MySQL语法：

```
KILL [CONNECTION | QUERY] processlist_id
```

3.5.3.1 KILL [CONNECTION] conn_id

其中，conn_id为 前端连接id值，可以通过运维命令show @@connection 获取。

3.5.3.1.1 举例

```
kill 1;
```

3.5.3.1.2 限制

- 在Kill自身连接的时候只会向自身写入OK包，不会有其他操作
- 如果Kill的连接在XA事务的提交或者回滚状态，不会直接关闭后端连接，会仅关闭前端连接
- 后端连接的关闭通过向MYSQL节点发送 KILL processlist_id 来完成

3.5.3.2 KILL query conn_id

其中，conn_id为 前端连接id值，可以通过运维命令show @@connection 获取。

3.5.3.2.1 举例

```
kill query 1;
```

3.5.3.2.3 说明

- dble 中kill query的实现是将正在执行语句的后端连接与前端连接相剥离的方式来实现。
- 后端未执行完成的语句，取决与mysql自身的机制。

3.5.3.2.2 限制

- 对于ddl语句，不保证一致性
- 对于未开启事务的dml操作不保证一致性

3.5.4 不支持的DAL语句

3.5.4 .1 以下语句不支持，[点击链接会跳转到MySQL官方文档](#)

[Account Management Statements](#)

[Resource Group Management Statements](#)

[Table Maintenance Statements](#)

[Component, Plugin, and Loadable Function Statements](#)

[CLONE Statement](#)

[BINLOG Statement](#)

[CACHE INDEX Statement](#)

[FLUSH Statement](#)

[LOAD INDEX INTO CACHE Statement](#)

[RESET Statement](#)

[RESET PERSIST Statement](#)

[RESTART Statement](#)

[SHUTDOWN Statement](#)

3.6 存储过程支持方式

存储过程通过注解的方式支持

3.6.1 Syntax

Create procedure

```

/Hint/
CREATE
  [DEFINER = user]
  PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
  [characteristic ...] routine_body

/Hint/
CREATE
  [DEFINER = user]
  FUNCTION [IF NOT EXISTS] sp_name ([func_parameter[,...]])
  RETURNS type
  [characteristic ...] routine_body

proc_parameter:
  [ IN | OUT | INOUT ] param_name type

func_parameter:
  param_name type

type:
  Any valid MySQL data type

characteristic: {
  COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
  Valid SQL routine statement

```

和MySQL语法的区别如下，绿色代表增加的部分,也就是hint

```

+ /Hint/
CREATE
  [DEFINER = user]
  PROCEDURE [IF NOT EXISTS] sp_name ([proc_parameter[,...]])
  [characteristic ...] routine_body

+ /Hint/
CREATE
  [DEFINER = user]
  FUNCTION [IF NOT EXISTS] sp_name ([func_parameter[,...]])
  RETURNS type
  [characteristic ...] routine_body

proc_parameter:
  [ IN | OUT | INOUT ] param_name type

func_parameter:
  param_name type

type:
  Any valid MySQL data type

characteristic: {
  COMMENT 'string'
| LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
  Valid SQL routine statement

```

drop procedure

```
/Hint/ DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

和MySQL语法的区别如下，绿色代表增加的部分,也就是hint

```

+ /Hint/
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name

```

call procedure

```
/Hint/ CALL sp_name([parameter[,...]])
```

```
/Hint/ CALL sp_name[()]
```

和MySQL语法的区别如下，绿色代表增加的部分,也就是hint

```

+ [/Hint/]
CALL sp_name([parameter[,...]])

+ [/Hint/]
CALL sp_name[()]

```

3.6.2 举例

```

删除存储过程:
/*!dble:sql=select 1 from account */drop procedure if exists proc_arc;
创建存储过程:
/*!dble:sql=select 1 from account */create procedure proc_arc(userid1 int)
begin
  insert into account_arc select * from account where userid=userid1;
  update account set arc_flag=true,arc_time=now() where userid=userid1;
end;
调用存储过程:
/*!dble:sql=select 1 from account */call proc_arc(1);

```

3.6.3 限制

- dble支持存储过程和自定义函数的透传，存储过程的开发完全使用MySQL的语法，开发、调试与部署的方法同单机MySQL相同。存储过程和自定义函数需要在所有节点上创建，节点扩容的时候也需要考虑存储过程和自定义函数的迁移。
- 存储过程和自定义函数是直接发送到节点上执行，中间件不参与运算，因此要慎重使用，需要保证过程的内部不出现跨节点运算。
- 存储过程调用时，要在调用语句之前增加注解，系统根据注解透传到节点运行，存储过程的执行路径以及执行结果的正确性由开发者保证。对于只是写入数据，不返回结果的存储过程，需要注意避免重复写入数据。对于返回结果的存储过程，需要特别注意返回结果的正确性。dble不会对存储过程的结果进行汇聚运算，只能由应用端自行完成。

3.7 Utility Statements

3.7.1 USE Statement

完全兼容MySQL语法

```
USE db_name
```

例:

```
use TESTDB;
```

3.7.2 EXPLAIN Statement

EXPLAIN 在dble中只负责查询计划，查询计划用法见[2.17_explain](#)

```
EXPLAIN explainable_stmt
```

```
explainable_stmt: {
  SELECT statement
| DELETE statement
| INSERT statement
| REPLACE statement
| UPDATE statement
}
```

与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
-{EXPLAIN | DESCRIBE | DESC}
- tbl_name [col_name | wild]

- {
EXPLAIN
- | DESCRIBE | DESC}
- [explain_type]
- {
  explainable_stmt
- | FOR CONNECTION connection_id}

-{EXPLAIN | DESCRIBE | DESC} ANALYZE [FORMAT = TREE] select_statement

-explain_type: {
- FORMAT = format_name
-}

-format_name: {
- TRADITIONAL
- | JSON
- | TREE
-}

explainable_stmt: {
  SELECT statement
- | TABLE statement
| DELETE statement
| INSERT statement
| REPLACE statement
| UPDATE statement
}
```

例:

```
explain SELECT select * from a_test where id=1;
```

注意事项:

1. INSERT中表不能为自增序列表
2. 在dble中，EXPLAIN 不等价于DESC

3.7.3 DESC

DESC在dble中只负责查看数据表结构

```
{DESCRIBE | DESC} tbl_name [col_name | wild]
```

与MySQL语法区别

下文中红色表示不支持的语法或关键字

```
{EXPLAIN | DESCRIBE | DESC}
  tbl_name [col_name | wild]

- {
- EXPLAIN
- | DESCRIBE | DESC}
- [explain_type]
- {
- explainable_stmt
- | FOR CONNECTION connection_id}

- {EXPLAIN | DESCRIBE | DESC} ANALYZE [FORMAT = TREE] select_statement

-explain_type: {
- FORMAT = format_name
-}

-format_name: {
- TRADITIONAL
- | JSON
- | TREE
-}

-explainable_stmt: {
- SELECT statement
- | TABLE statement
- | DELETE statement
- | INSERT statement
- | REPLACE statement
- | UPDATE statement
-}
```

例:

```
DESC a_test id;
```

注意: 在dble中, EXPLAIN 不等价于DESC

3.7.4 EXPLAIN2

查询计划2用法见[2.17_explain](#)

```
EXPLAIN2 shardingNode=node_name sql=sql_stmt
```

例:

```
explain2 shardingNode=dn2 sql=select * from a_test where id=1;
```

3.7.5 HELP Statement 不支持

3.8 Hint

3.8.1 分库分表-Syntax

```
/* { ! | #}dbtle: {sql=SELECT select_expr FROM table_references WHERE where_condition

        |shardingNode=shardingNode_name

        |db_type={slave|master}}

*/ ordinary_sql
```

3.8.2 读写分离-Syntax

```
/* { ! | #}dbtle: {db_type={slave|master}}

        |db_instance_url={ip:port}}

*/ ordinary_sql
/* master */ ordinary_sql
/* uproxy_dest: ip:port */ ordinary_sql
```

注意:

- 在不影响sql语句完整性的前提下, /* master */ 和 /* uproxy_dest: ip:port */ 可放在sql语句的首尾和中间

3.8.3 举例

```
/*!dbtle:sql=select 1 from sbtest */ call p_show_time();
/*!dbtle:shardingNode=dn1*/ update sbtest set name = 'test';
/*!dbtle:db_type=master*/ select count(*) from sbtest;
/*!dbtle:db_instance_url=127.0.0.1:3307*/ select count(*) from sbtest;
/*#dbtle:sql=select 1 from sbtest */ call p_show_time();
/*#dbtle:shardingNode=dn1*/ update sbtest set name = 'test';
/*#dbtle:db_type=master*/ select count(*) from sbtest;
/*#dbtle:db_instance_url=127.0.0.1:3307*/ select count(*) from sbtest;
select /* master */ * from sbtest;
show processlist /* uproxy_dest: 127.0.0.1:3307 */
```

3.9 其他不支持语句

- Compound-Statement Syntax
- Replication Statements
- DDL:
 - 不支持客户端针对database的操作语句，如alter database、drop database。create database 在客户端遇到会判断schema已经配置后返回ok，否则报错。
 - 不支持管理端针对database的操作语句，如alter database。
 - 不支持create table时的一些table option，如DATA DIRECTORY、ALGORITHM等，table option在alter table时也不能修改
 - 不支持ALTER TABLE ... LOCK ...
 - 不支持ALTER TABLE ... ORDER BY ...
 - 不支持create table ... select ...
 - 库名、表名不可修改，拆分字段的名称和类型都不可以变更
 - 不支持外键关联
 - 不支持临时表
 - 不支持分布式级别的存储过程和自定义函数
 - 不支持触发器

- DML:
 - 对于INSERT... VALUES(expr)，不支持expr中含有子查询
 - 支持部分INSERT... SELECT... 举例

```
<shardingTable name="test10" shardingNode="dn2,dn3,dn4" function="hash-three" shardingColumn="id"/>
<shardingTable name="test11" shardingNode="dn2,dn3,dn4" function="hash-three" shardingColumn="id"/>
<shardingTable name="test12" shardingNode="dn3,dn4" function="hash-two" shardingColumn="id"/>
<singleTable name="test20" shardingNode="dn2" />
<singleTable name="test22" shardingNode="dn1" />
<globalTable name="test30" shardingNode="dn1,dn2,dn3,dn4" />
<globalTable name="test31" shardingNode="dn1,dn2,dn3,dn4" />
```

- 同一类型的有相同节点的表都支持，如insert into test10(id,name) select id,name from test11; insert into test30(id,name) select id,name from test31;
- 不同类型的表不支持，如 insert into test30(id,score) select id,score from test10;
- 同一类型的但是节点不同的表不支持，如insert into test20(id,score) select id,score from test22;insert into test10(id,score) select id,score from test12;
- 不支持不包含拆分字段的INSERT语句
- 不支持HANDLER语句
- 不支持修改拆分字段的值
- 不支持DELETE ... ORDER BY ... LIMIT ...
- 不支持DELETE/UPDATE ...LIMIT路由到一个分片表的多个节点
- 不支持DO语句
- 查询:
 - 不支持select ... use/ignore index ...
 - 不支持select ... group by ... with rollup
 - 不支持select ... for update | lock in share mode 正确语义
 - 不支持select ... into outfile ...
 - 不支持Row Subqueries
 - 不支持select ... union [all] select ... order by ..., 可写成(select ...) union [all] (select ...) order by ...
 - 不支持session变量赋值与查询，如set @rowid=0;select @rowid:=@rowid+1,id from user;
- 管理语句:
 - 不支持用户管理及权限管理语句
 - 不支持表维护语句，包括ANALYZE/CHECK/CHECKSUM/OPTIMIZE/REPAIR TABLE
 - 不支持INSTALL/UNINSTALL PLUGIN语句
 - 不支持BINLOG语句
 - 不支持CACHE INDEX/ LOAD INDEX INTO CACHE语句
 - 不支持除FLUSH TABLES [WITH READ LOCK]以外的其他FLUSH语句，FLUSH TABLE也仅语法支持无实际意义
 - 不支持RESET语句
 - 不支持大部分的运维SHOW语句，如SHOW PROFILES、SHOW ERRORS等

3.10 函数与操作符支持列表(alpha版本)

3.10.0 注意：

1. 如果能保证SQL会将函数整体下发给某个后端结点，函数支持度没有意义，支持度将托管于MySQL.
2. 在2.18.09.0 版本之前（含），涉及到中文字符集的字符串相关函数有bug。
3. 除了聚合函数经过充分测试之外，其余函数只进行了简单的测试，并没有全覆盖测试，请使用前充分测试。
4. 未在列表中的函数，一概不支持。

3.10.1 Operators

Name	Description	Support
AND, &&	Logical AND	Y
=	Assign a value (as part of a SET statement, or as part of the SET clause in an UPDATE statement)	Y
:=	Assign a value	N
BETWEEN ... AND ...	Check whether a value is within a range of values	Y
BINARY	Cast a string to a binary string	N
&	Bitwise AND	Y
~	Bitwise inversion	Y
	Bitwise OR	Y
^	Bitwise XOR	Y
CASE	Case operator	Y
DIV	Integer division	Y
/	Division operator	Y
=	Equal operator	Y
<=>	NULL-safe equal to operator	Y
>	Greater than operator	Y
>=	Greater than or equal operator	Y
IS	Test a value against a boolean	Y
IS NOT	Test a value against a boolean	Y
IS NOT NULL	NOT NULL value test	Y
IS NULL	NULL value test	Y
->	Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT().	N
->>	Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()).	N
<<	Left shift	Y
<	Less than operator	Y
<=	Less than or equal operator	Y
LIKE	Simple pattern matching	Y
-	Minus operator	Y
%, MOD	Modulo operator	Y
NOT, !	Negates value	Y
NOT BETWEEN ... AND ...	Check whether a value is not within a range of values	Y
!=, <>	Not equal operator	Y
NOT LIKE	Negation of simple pattern matching	Y
NOT REGEXP	Negation of REGEXP	Y
, OR	Logical OR	Y
+	Addition operator	Y
REGEXP	Whether string matches regular expression	Y
>>	Right shift	Y

RLIKE	Whether string matches regular expression	N
SOUNDS LIKE	Compare sounds	N
*	Multiplication operator	N
-	Change the sign of the argument	Y
XOR	Logical XOR	Y
COALESCE()	Return the first non-NULL argument	Y
GREATEST()	Return the largest argument	Y
IN()	Check whether a value is within a set of values	Y
INTERVAL()	Return the index of the argument that is less than the first argument	Y
ISNULL()	Test whether the argument is NULL	Y
LEAST()	Return the smallest argument	Y
STRCMP()	Compare two strings	Y

3.10.2 Control Flow Functions

Name	Description	Support
CASE	Case operator	Y
IF()	If/else construct	Y
IFNULL()	Null if/else construct	Y
NULLIF()	Return NULL if expr1 = expr2	Y

3.10.3 String Functions

Name	Description	Support
ASCII()	Return numeric value of left-most character	Y
BIN()	Return a string containing binary representation of a number	N
BIT_LENGTH()	Return length of argument in bits	Y
CHAR()	Return the character for each integer passed	Y
CHAR_LENGTH()	Return number of characters in argument	Y
CHARACTER_LENGTH()	Synonym for CHAR_LENGTH()	Y
CONCAT()	Return concatenated string	Y
CONCAT_WS()	Return concatenate with separator	Y
ELT()	Return string at index number	Y
EXPORT_SET()	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string	N
FIELD()	Return the index (position) of the first argument in the subsequent arguments	Y
FIND_IN_SET()	Return the index position of the first argument within the second argument	Y
FORMAT()	Return a number formatted to specified number of decimal places	Y
FROM_BASE64()	Decode base64 encoded string and return result	N
HEX()	Return a hexadecimal representation of a decimal or string value	Y
INSERT()	Insert a substring at the specified position up to the specified number of characters	Y
INSTR()	Return the index of the first occurrence of substring	Y
LCASE()	Synonym for LOWER()	Y
LEFT()	Return the leftmost number of characters as specified	Y
LENGTH()	Return the length of a string in bytes	Y
LIKE	Simple pattern matching	Y
LOAD_FILE()	Load the named file	N
	Return the position of the first occurrence of	

	substring	
LOWER()	Return the argument in lowercase	Y
LPAD()	Return the string argument, left-padded with the specified string	Y
LTRIM()	Remove leading spaces	Y
MAKE_SET()	Return a set of comma-separated strings that have the corresponding bit in bits set	Y
MATCH	Perform full-text search	N
MID()	Return a substring starting from the specified position	N
NOT LIKE	Negation of simple pattern matching	Y
NOT REGEXP	Negation of REGEXP	Y
OCT()	Return a string containing octal representation of a number	N
OCTET_LENGTH()	Synonym for LENGTH()	N
ORD()	Return character code for leftmost character of the argument	Y
POSITION()	Synonym for LOCATE()	N
QUOTE()	Escape the argument for use in an SQL statement	Y
REGEXP	Whether string matches regular expression	Y
REPEAT()	Repeat a string the specified number of times	Y
REPLACE()	Replace occurrences of a specified string	Y
REVERSE()	Reverse the characters in a string	Y
RIGHT()	Return the specified rightmost number of characters	Y
RLIKE	Whether string matches regular expression	N
RPAD()	Append string the specified number of times	Y
RTRIM()	Remove trailing spaces	Y
SOUNDEX()	Return a soundex string	Y
SOUNDS LIKE	Compare sounds	Y
SPACE()	Return a string of the specified number of spaces	Y
STRCMP()	Compare two strings	Y
SUBSTR()	Return the substring as specified	Y
SUBSTRING()	Return the substring as specified	Y
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter	Y
TO_BASE64()	Return the argument converted to a base-64 string	N
TRIM()	Remove leading and trailing spaces	Y
UCASE()	Synonym for UPPER()	Y
UNHEX()	Return a string containing hex representation of a number	Y
UPPER()	Convert to uppercase	Y
WEIGHT_STRING()	Return the weight string for a string	N

3.10.3.1 HEX 注意事项

由于 MySQL 文档中，只说明了 HEX 作用于小整数 (≤ 64 位) 和 string 的情况。没有具体说明大整数 (> 64 位) 和浮点数这类比较极端的情况。在以上这些极端情况下，MySQL 8.0 和 MySQL 5.7 的结果不一定一致。dble 的 HEX 结果和 MySQL 8 实现保持一致，但建议用户不要过度依赖这个结果。

3.10.4 Numeric Functions and Operators

Name	Description	Support
ABS()	Return the absolute value	Y
ACOS()	Return the arc cosine	Y
ASIN()	Return the arc sine	Y
ATAN()	Return the arc tangent	Y
ATAN2(), ATAN()	Return the arc tangent of the two arguments	Y
CEIL()	Return the smallest integer value not less than the argument	Y
CEILING()	Return the smallest integer value not less than the argument	Y
CONV()	Convert numbers between different number bases	Y
COS()	Return the cosine	Y
COT()	Return the cotangent	Y
CRC32()	Compute a cyclic redundancy check value	Y
DEGREES()	Convert radians to degrees	Y
DIV	Integer division	Y
/	Division operator	Y
EXP()	Raise to the power of	Y
FLOOR()	Return the largest integer value not greater than the argument	Y
LN()	Return the natural logarithm of the argument	Y
LOG()	Return the natural logarithm of the first argument	Y
LOG10()	Return the base-10 logarithm of the argument	Y
LOG2()	Return the base-2 logarithm of the argument	Y
-	Minus operator	Y
MOD()	Return the remainder	N (If the SQL pass through, still supported)
%, MOD	Modulo operator	Y
PI()	Return the value of pi	Y
+	Addition operator	Y
POW()	Return the argument raised to the specified power	Y
POWER()	Return the argument raised to the specified power	Y
RADIANS()	Return argument converted to radians	Y
RAND()	Return a random floating-point value	Y
ROUND()	Round the argument	Y
SIGN()	Return the sign of the argument	Y
SIN()	Return the sine of the argument	Y
SQRT()	Return the square root of the argument	Y
TAN()	Return the tangent of the argument	Y
*	Multiplication operator	Y
TRUNCATE()	Truncate to specified number of decimal places	Y
-	Change the sign of the argument	Y

3.10.5 Date and Time Functions

Name	Description	Support
ADDDATE()	Add time values (intervals) to a date value	Y
ADDTIME()	Add time	Y
CONVERT_TZ()	Convert from one time zone to another	N
CURDATE()	Return the current date	Y
CURRENT_DATE()	Synonyms for CURDATE()	Y
CURRENT_TIME()	Synonyms for CURTIME()	Y

CURRENT_TIMESTAMP()	Synonyms for NOW()	Y
CURTIME()	Return the current time	Y
DATE()	Extract the date part of a date or datetime expression	Y
DATE_ADD()	Add time values (intervals) to a date value	Y
DATE_FORMAT()	Format date as specified	Y
DATE_SUB()	Subtract a time value (interval) from a date	Y
DATEDIFF()	Subtract two dates	Y
DAY()	Synonym for DAYOFMONTH()	N
DAYNAME()	Return the name of the weekday	Y
DAYOFMONTH()	Return the day of the month (0-31)	Y
DAYOFWEEK()	Return the weekday index of the argument	Y
DAYOFYEAR()	Return the day of the year (1-366)	Y
EXTRACT()	Extract part of a date	Y
FROM_DAYS()	Convert a day number to a date	Y
FROM_UNIXTIME()	Format Unix timestamp as a date	Y
GET_FORMAT()	Return a date format string	Y
HOUR()	Extract the hour	Y
LAST_DAY	Return the last day of the month for the argument	N
LOCALTIME()	Synonym for NOW()	Y
LOCALTIMESTAMP()	Synonym for NOW()	Y
MAKEDATE()	Create a date from the year and day of year	Y
MAKETIME()	Create time from hour, minute, second	Y
MICROSECOND()	Return the microseconds from argument	Y
MINUTE()	Return the minute from the argument	Y
MONTH()	Return the month from the date passed	Y
MONTHNAME()	Return the name of the month	Y
NOW()	Return the current date and time	Y
PERIOD_ADD()	Add a period to a year-month	Y
PERIOD_DIFF()	Return the number of months between periods	Y
QUARTER()	Return the quarter from a date argument	Y
SEC_TO_TIME()	Converts seconds to 'HH:MM:SS' format	Y
SECOND()	Return the second (0-59)	Y
STR_TO_DATE()	Convert a string to a date	Y
SUBDATE()	Synonym for DATE_SUB() when invoked with three arguments	Y
SUBTIME()	Subtract times	Y
SYSDATE()	Return the time at which the function executes	Y
TIME()	Extract the time portion of the expression passed	Y
TIME_FORMAT()	Format as time	Y
TIME_TO_SEC()	Return the argument converted to seconds	Y
TIMEDIFF()	Subtract time	Y
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments	N
TIMESTAMPADD()	Add an interval to a datetime expression	Y
TIMESTAMPDIFF()	Subtract an interval from a datetime expression	Y
TO_DAYS()	Return the date argument converted to days	Y
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0	Y
UNIX_TIMESTAMP()	Return a Unix timestamp	Y
UTC_DATE()	Return the current UTC date	Y
UTC_TIME()	Return the current UTC time	Y

UTC_TIMESTAMP()	Return the current UTC date and time	Y
WEEK()	Return the week number	Y
WEEKDAY()	Return the weekday index	Y
WEEKOFYEAR()	Return the calendar week of the date (1-53)	Y
YEAR()	Return the year	Y
YEARWEEK()	Return the year and week	Y
CURRENT_DATE	Synonyms for CURDATE()	N
CURRENT_TIME	Synonyms for CURTIME()	N
CURRENT_TIMESTAMP	Synonyms for NOW()	N
LOCALTIME	Synonym for NOW()	N
LOCALTIMESTAMP()	Synonym for NOW()	N

3.10.6 Cast Functions and Operators

Name	Description	Support
BINARY	Cast a string to a binary string	N
CAST()	Cast a value as a certain type	Y
CONVERT()	Cast a value as a certain type	Y

3.10.6.1 CAST 不支持以下类型或语法

BINARY

CHAR[(N)] [charset_info] 包含charset_info时

JSON

SIGNED [INTEGER] 包含INTEGER时(druid解析问题)

UNSIGNED [INTEGER] 包含INTEGER时 (druid解析问题)

3.10.6.2 CONVERT 不支持以下类型或语法

BINARY

CHAR[(N)] [charset_info] 包含charset_info时

JSON

SIGNED [INTEGER] 包含INTEGER时(druid解析问题)

UNSIGNED [INTEGER] 包含INTEGER时 (druid解析问题)

3.10.7 Bit Functions and Operators

Name	Description	Support
BIT_COUNT()	Return the number of bits that are set	Y
&	Bitwise AND	Y
~	Bitwise inversion	Y
	Bitwise OR	Y
^	Bitwise XOR	Y
<<	Left shift	Y
>>	Right shift	Y

3.10.8 Aggregate (GROUP BY) Functions

Name	Description	Support
AVG()	Return the average value of the argument	Y
BIT_AND()	Return bitwise AND	Y
BIT_OR()	Return bitwise OR	Y
BIT_XOR()	Return bitwise XOR	Y
COUNT()	Return a count of the number of rows returned	Y
COUNT(DISTINCT)	Return the count of a number of different values	Y
GROUP_CONCAT()	Return a concatenated string	Y
JSON_ARRAYAGG()	Return result set as a single JSON array	N
JSON_OBJECTAGG()	Return result set as a single JSON object	N
MAX()	Return the maximum value	Y
MIN()	Return the minimum value	Y
STD()	Return the population standard deviation	Y
STDDEV()	Return the population standard deviation	Y
STDDEV_POP()	Return the population standard deviation	Y
STDDEV_SAMP()	Return the sample standard deviation	Y
SUM()	Return the sum	Y
VAR_POP()	Return the population standard variance	Y
VAR_SAMP()	Return the sample variance	Y
VARIANCE()	Return the population standard variance	Y

备注: STD 和 VARIANCE 相关函数, 因为分布式计算的局限性, 精度会有一些问题, 见

[STD\(\) / STDDEV\(\) / STDDEV_POP\(\) / STDDEV_SAMP\(\) / VAR_POP\(\) / VAR_SAMP\(\) / VARIANCE\(\)](#) result precision is not correct

在 AVG\SUM 等相关计算函数的时候, 由于 dble 的数据来源于 MySQL 的查询返回,

当 MySQL 默认返回的数据精度不够时, 可能出现最终查询的结果和 MySQL 的计算结果有少许差异的情况, 见

[for data type float, dble and mysql may get different results](#)

3.10.9 JSON Functions

Name	Description	Support
JSON_EXTRACT()	selected from the parts of the json document matched by the path arguments	Y
JSON_UNQUOTE()	Unquotes JSON value and returns the result as a utf8mb4 string.	Y

3.10.10 Full-Text Search Functions

not supported

3.10.11 XML Functions

not supported

3.10.12 Encryption and Compression Functions

not supported

3.10.13 Information Functions

not supported

3.10.14 Spatial Analysis Functions

not supported

3.10.15 Functions Used with Global Transaction IDs

not supported

3.10.16 MySQL Enterprise Encryption Functions

not supported

3.10.17 Miscellaneous Functions

not supported

参考资料: [MySQL5.7 函数文档](#)

3.11 导入导出方式的支持

3.11.1 支持工具

1. workbench
2. dbeaver
3. mysqldump
4. navicat
5. 导入数据也可以使用mysql中的source和load data

3.11.2 注意点

1. 若使用mysqldump导出时，请按照以下格式进行导出，否则可能出现错误，因为有些 mysqldump 参数dble不支持。

```
./mysqldump -h127.0.0.1 -utest -P3306 -p111111 --default-character-set=utf8mb4 --master-data=2 --single-transaction --set-gtid-purged=off --hex-lob --databases schema1 > export.sql
```

1. 导入时，脚本中若存在非注释性的视图相关语句，需要注释掉或删除。
2. 导出时，因为dble对视图相关的一些语句不支持，因此尽量确保导出的dble中不存在视图。

4 协议兼容

- [4.1 基本包](#)
- [4.2 连接建立](#)
- [4.3 文本协议](#)
- [4.4 二进制协议 \(Prepared Statements\)](#)
- [4.5 服务响应包](#)

4.1 基本包

- 标准包：支持
- 大包（16M以上）：支持
- 压缩包：支持, 需要全局配置, 参见bootstrap.cnf
- 压缩后的大包：不支持

4.2 连接建立

4.2.1 Authentication Plugin

a.空(即默认,等同于mysql_native_password 或者8.0以后的caching_sha2_password)

b.mysql_native_password

c.caching_sha2_password

4.2.2 Capabilities

大部分权能标志位在建立连接之后是不能修改的。因为dble的连接模型是预先建立后端连接池，所以在建立时候已经初始化了权能标志位。之后如果新的前端连接建立连接时候设置了不同的权能标志位，就无法生效传递给后端连接。

比如jdbc连接dble设置了CLIENT_FOUND_ROWS 相关的属性，是无法实际生效的。

又比如，dble设置了IGNORE_SPACE，所以不做任何操作的情况下 `select @@sql_mode` 会永远包含IGNORE_SPACE

这样带来的限制是，某些和函数同名的词会当成保留字来处理。参见[MySQL文档](#) 以及相关[issue-972](#)

当然，IGNORE_SPACE又是个特例，它可以通过session级别的sql_mode进行设置。

这里列举一下dble建立连接权能标志位的使用情况。

如：

名称	标记值	描述	后端连接设置值	模拟服务端权能位
CLIENT_LONG_PASSWORD	1	Use the improved version of Old Password Authentication. Assumed to be set since 4.1.1.	Y	Y
CLIENT_FOUND_ROWS	2	Send found rows instead of affected rows in EOF_Packet	Y	Y
CLIENT_LONG_FLAG	4	Get all column flags.	Y	Y
CONNECT_WITH_DB	8	Database (schema) name can be specified on connect in Handshake Response Packet.	Y	Y
CLIENT_NO_SCHEMA	16	Don't allow database.table.column.	N	N
CLIENT_COMPRESS	32	Compression protocol supported	bootstrap.cnf的useCompression选项控制	
CLIENT_ODBC	64	Special handling of ODBC behavior.No special behavior since 3.22.	Y	Y
CLIENT_LOCAL_FILES	128	Can use LOAD DATA LOCAL.	Y	Y
CLIENT_IGNORE_SPACE	256	Ignore spaces before '('.	Y	Y
CLIENT_PROTOCOL_41	512	New 4.1 protocol	Y	Y
CLIENT_INTERACTIVE	1024	This is an interactive client.	Y	Y
CLIENT_SSL	2048	Use SSL encryption for the session	N	N
CLIENT_IGNORE_SIGPIPE	4096	Client only flag.Not used.	Y	Y
CLIENT_TRANSACTIONS	8192	Client knows about transactions	Y	Y
CLIENT_RESERVED	16384	DEPRECATED:Old flag for 4.1 protocol.	N	N
CLIENT_RESERVED2	32768	DEPRECATED:Old flag for 4.1 authentication.	Y	Y
CLIENT_MULTI_STATEMENTS	65536	Enable/disable multi-stmt support	Y	Y
CLIENT_MULTI_RESULTS	131072	Enable/disable multi-results	Y	Y
CLIENT_PS_MULTI_RESULTS	262144	Multi-results and OUT parameters in PS-protocol	N	N
CLIENT_PLUGIN_AUTH	524288	Client supports plugin authentication.	N	Y
CLIENT_CONNECT_ATTRS	1048576	Client supports connection attributes.	N	N
CLIENT_PLUGIN_AUTH_LENENC_CLIENT_DATA	2097152	Enable authentication response packet to be larger than 255 bytes.	N	N
CLIENT_CAN_HANDLE_EXPIRED_PASSWORDS	4194304	Don't close the connection for a user account with expired password.	N	N
CLIENT_SESSION_TRACK	8388608	Capable of handling server state change information.	N	N
CLIENT_DEPRECATED_EOF	16777216	Client no longer needs EOF_Packet and will use OK_Packet instead.	N	N
CLIENT_SSL_VERIFY_SERVER_CERT	1UL << 30	Verify server certificate	N	N
CLIENT_REMEMBER_OPTIONS	1UL << 31	Don't reset the options after an unsuccessful connect.	N	N

权能标志位定义：

参考 https://dev.mysql.com/doc/dev/mysql-server/8.0.13/group__group__cs__capabilities__flags.html

4.3 文本协议

4.3.1 Supported

- COM_INIT_DB
Specifies the default schema for the connection.
- COM_PING
Sends a packet containing one byte to check that the connection is active.
- COM_QUERY
Sends the server an SQL statement to be executed immediately.Support Multi-Statement.
- COM_QUIT
Client tells the server that the connection should be terminated.
- COM_SET_OPTION
Enables or disables server option.
- COM_CHANGE_USER
Resets the connection and re-authenticates with the given credentials.
- COM_RESET_CONNECTION
Resets a connection without re-authentication.
 - 关闭后端连接(rollback & unlock)
 - 事务状态情况
 - 用户变量清空
 - 系统变量恢复成系统默认值
 - prepare清空
 - 上下文(字符集, 隔离级别)恢复成为默认值
 - LAST_INSERT_ID 置零
- COM_FIELD_LIST
MySQL Doc said that it is deprecated from 5.7.11 . But some tools are still use it, like OGG or MariaDB client.

4.3.1.1 Multi-Statement

- Supported
 - DML:select/insert/update/replace/delete
 - DDL
 - OTHER
 - BEGIN;
 - COMMIT;
 - LOCK TABLE
 - UNLOCK TABLES
 - START
 - KILL
 - USE
 - ROLLBACK
 - MYSQL_CMD_COMMENT
 - MYSQL_COMMENT
 - SELECT VERSION_COMMENT (SELECT @@VERSION_COMMENT)
 - SELECT DATABASE (select database())
 - SELECT USER (select user())
 - SELECT VERSION (select version())
 - SELECT SESSION_INCREMENT(select @@session.auto_increment_increment)
 - SELECT SESSION_ISOLATION(select @@session.tx_isolation)
 - SELECT LAST_INSERT_ID(select last_insert_id(#) as id)
 - SELECT IDENTITY(select @@identity)
 - SELECT SESSION_TX_READ_ONLY (select @@session.tx_read_only)
- Not Supported
 - EXPLAIN
 - EXPLAIN2
 - DESCRIBE
 - SET
 - SHOW DATABASES/TABLES/TABLE_STATUS/COLUMNS/INDEX/CREATE_TABLE/VARIABLES/CREATE_VIEW/CHARSET
 - HELP
 - LOAD_DATA_INFILE_SQL
 - CREATE_VIEW
 - REPLACE_VIEW
 - ALTER_VIEW
 - DROP_VIEW

4.3.2 Not Supported

- COM_DEBUG
Forces the server to dump debug information to stdout
- COM_STATISTICS
Get internal server statistics.
- COM_CREATE_DB
- COM_DROP_DB

4.3.3 Internal

- COM_SLEEP
Used inside the server only.
- COM_CONNECT an internal command in the server.
- COM_TIME an internal command in the server.
- COM_DAEMON an internal command in the server.
- COM_DELAYED_INSERT an internal command in the server.

4.3.4 Deprecated

- COM_PROCESS_INFO
Deprecated from 5.7.11.
- COM_PROCESS_KILL
Deprecated from 5.7.11.
- COM_SHUTDOWN
Deprecated from 5.7.9.
- COM_REFRESH
Deprecated from 5.7.11.

4.4 预编译语句 (Prepared Statements)

4.4.1 开启方式

客户端

- 如果是jdbc需开启useServerPrepStmts, 此时才会使用 server-side prepare, 否则属于 client-side prepare。
- 其他语言的 driver通常不用配置选项。

验证是否开启了

只提供 java 版本的

```
PreparedStatement preparedStatement = con.prepareStatement("select t1.id from no_sharding_t1 t1 where t1.id=?"); //可用于验证是否使用了db1e
侧 prepare assert preparedStatement instanceof ServerPreparedStatement;
```

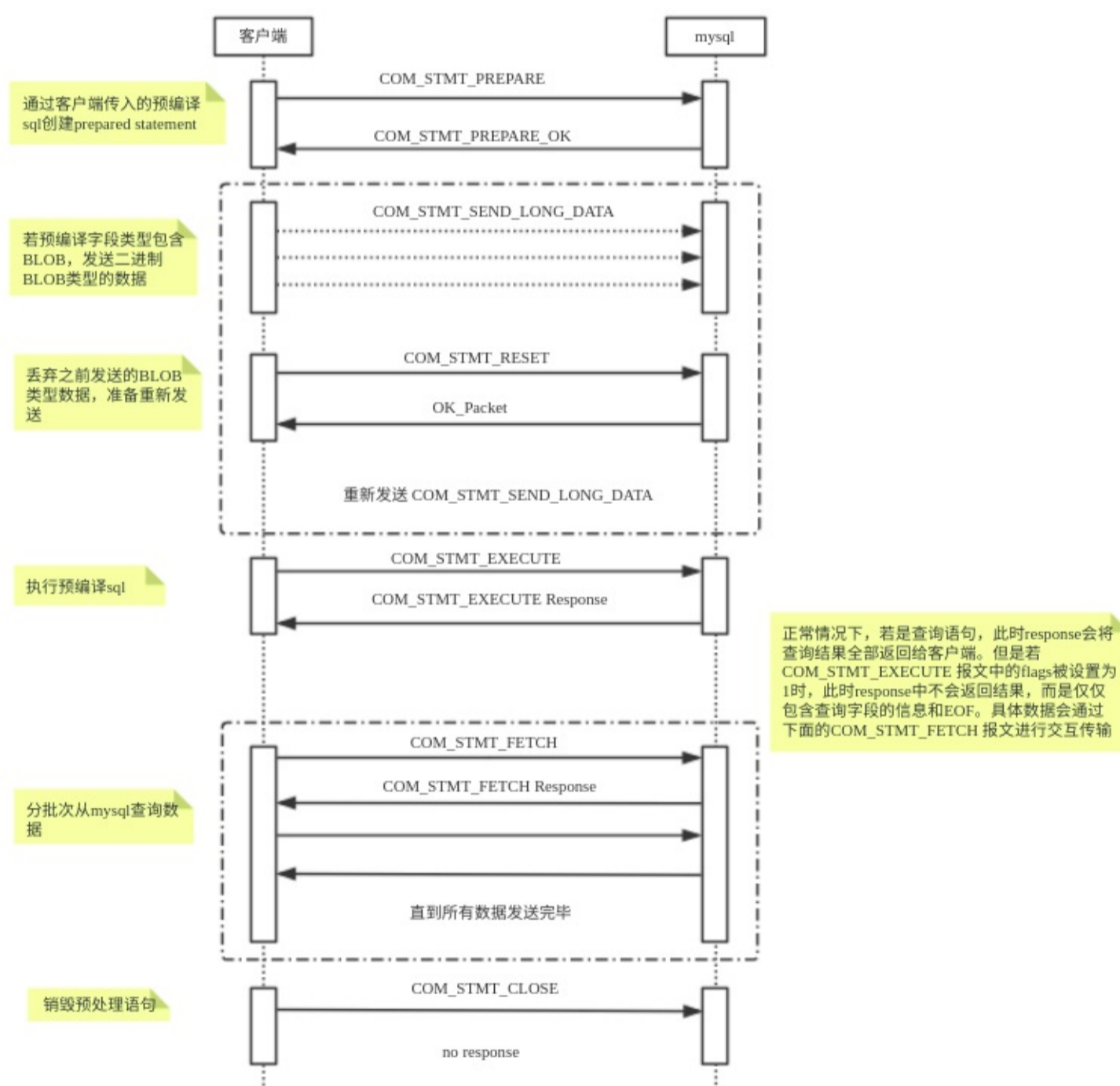
4.4.2 使用方式

和直连 mysql 一样

4.4.4 分类

- server-side prepare: 通过 client发送 PS 协议的报文给server, 由 server来完成拼装参数、优化、执行。
- client-side prepare : 由 client 来实现 PS 接口, prepare 阶段完成拼装参数, 拼装完后, 一次性发送 即时 SQL给 server, 由 server 来完成优化、执行。这本质上是一个伪预编译, 上述的 "省去了每次都要解析优化的过程" 这个优点无法实现。

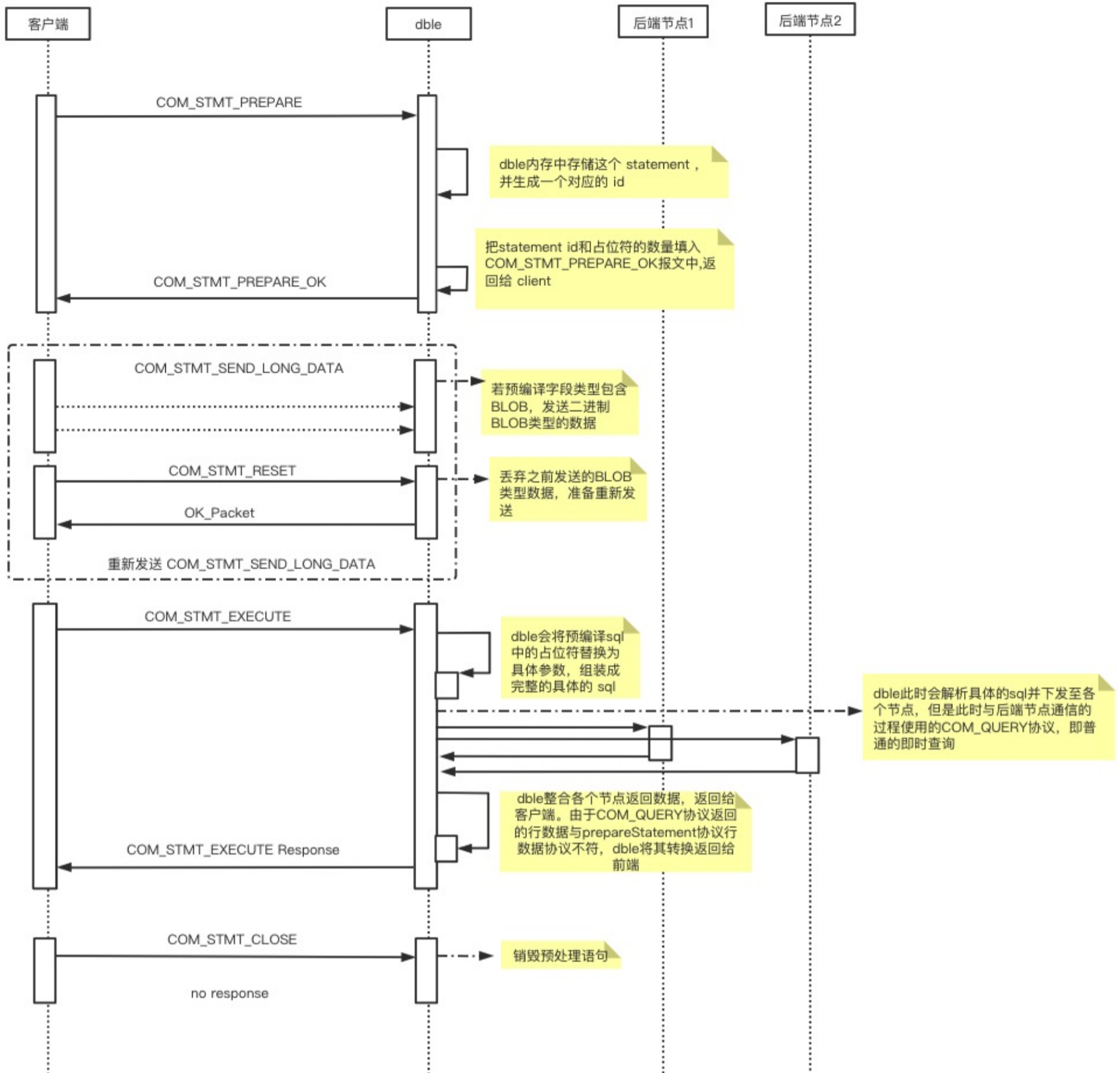
4.4.5 MySQL流程



注意点:

- 可通过url方式指定useCursorFetch=true, 开启分批从server查询数据。
- 但是jdbc中默认fetchSize为0, jdbc中必须fetchSize > 0才会发送fetch包分批查询数据, 否则和普通prepareStatement没有区别。

4.4.6 Dble流程



可以看到 client <-> dble 通讯使用了 server-side prepare, dble <-> mysql 通讯使用了 client-side prepare, 也就是说后端通讯和普通的即时查询无异, 只是需要做一些协议上的包上的转换。

4.4.7 通讯协议介绍

- **COM_STMT_CLOSE**
Closes a previously prepared statement.
- **COM_STMT_EXECUTE**
Executes a previously prepared statement.
- **COM_STMT_RESET**
Resets a prepared statement on client and server to state after preparing.
- **COM_STMT_SEND_LONG_DATA**
When data for a specific column is big, it can be sent separately.
- **COM_STMT_PREPARE**
Prepares a statement on the server
NOTICE: Although COM_STMT_PREPARE works, but dble will not do pre-compile.
- **COM_STMT_FETCH**
Fetches rows from a prepared statement

4.4.8 Dble 游标

不支持读写分离场景

4.4.8.1 游标分类

- server-side cursor: server把结果集暂存起来, 维护一个游标, client 根据需要读取指定的行数
- client-side cursor: client 从 tcp 层面 控制报文的读取, 当报文较大时暂停读取 socket。(不推荐, 因为server 需要等待所有数据发送给 client 后, 才能释放资源。)
- 另一种 client-side cursor: client 把所有结果集读取到本地缓存, client每次从缓存读取指定行数 (不推荐, 本质上是伪 cursor, 只实现了 cursor API。并且在数据量较大时很容易撑爆 client 的内存)

4.4.8.2 游标开启必要条件

DBLE 端

- 如果版本<3.21.02, 不支持。
- 如果版本=3.21.02,无需设置
- 如果版本>3.21.02,需在 bootstrap.cnf开启-DenableCursor=true

客户端

1. 使用支持游标的driver (mysql官方的jdbc driver就支持)
2. 如果是jdbc需开启useServerPrepStmts和useCursorFetch选项
3. 执行 preparedStatement 后设置 fetchSize, 必须大于 0.
4. 执行 execute

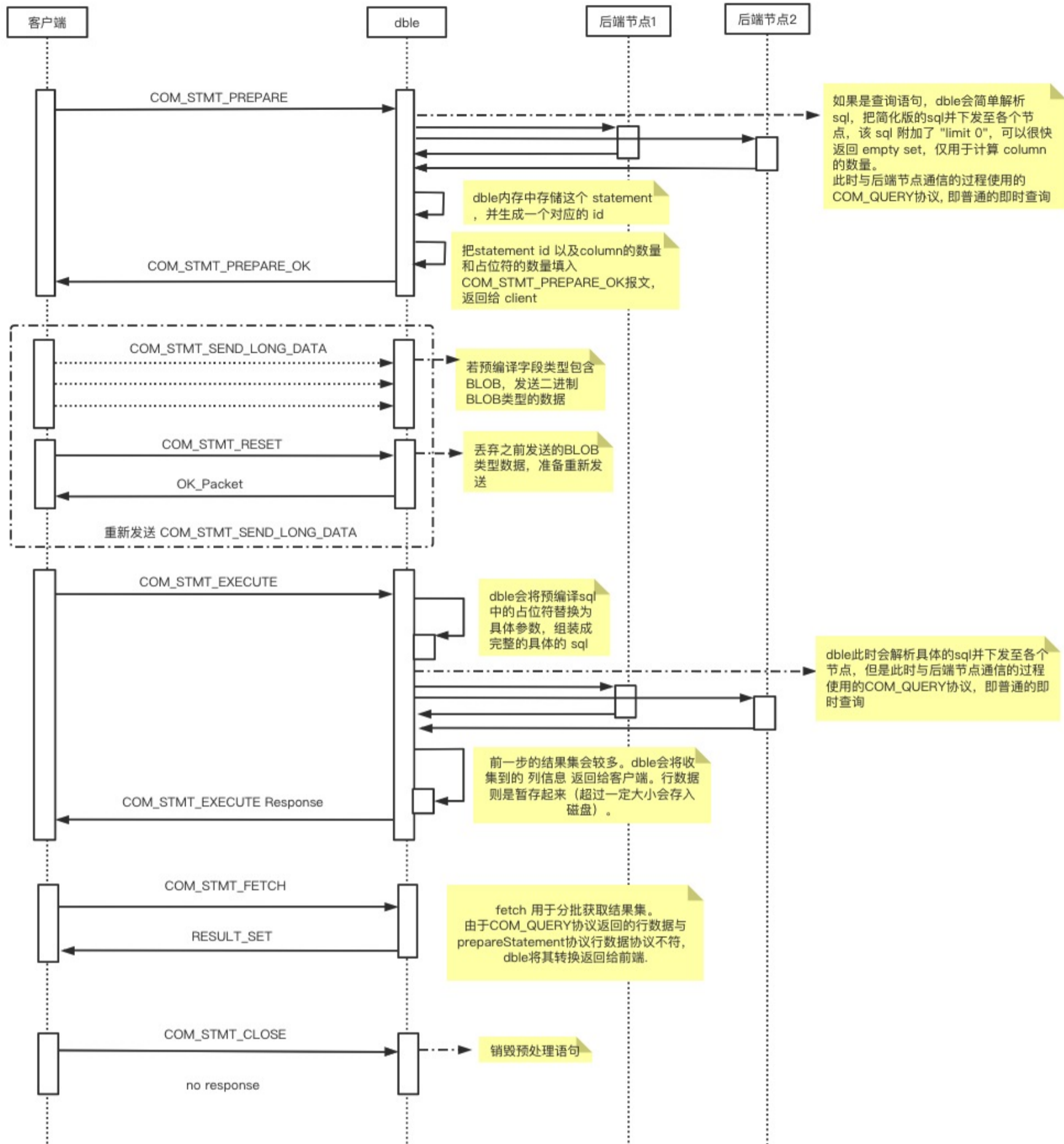
此时是开启游标的, 如果对结果集 resultSet进行遍历, 会按 fetchSize 的大小一次次地从 dble 取回数据。

验证是否开启了游标

客户端执行第4步后, 调用私有方法 useServerFetch 可验证。

```
final ResultSet resultSet = preparedStatement.executeQuery();
//可用于验证是否使用了server-side 游标
Method method = com.mysql.cj.jdbc.StatementImpl.class.getDeclaredMethod("useServerFetch");
method.setAccessible(true);
Boolean useServerFetch = (Boolean) method.invoke(preparedStatement);
assert useServerFetch==true;
```

4.4.8.3 Dble Server-side Cursor Flow



原理:

1. prepare阶段下发特殊语句。用于计算sql中的列数, 这是client所需的开启游标的必要条件。
2. execute阶段把结果集存储到临时文件
3. fetch阶段把结果集分批次一次次取出来

4.4.8.4 相关参数

- maxHeapTableSize
- heapTableBufferChunkSize

见文档 https://actiontech.github.io/dble-docs-cn/1.config_file/1.02_bootstrap.cnf.html

4.5 服务响应包

兼容支持：

- EOF_Packet
- ERR_Packet
- OK_Packet
- LOCAL_INFILE Packet
- PACKET_LOCAL_INFILE
- PACKET_RESULTSET

5. 已知限制

- [5.1 druid引发的限制](#)
- [5.2 其他已知限制](#)

5.1 druid引发的限制

1. INSERT ... VALUE ... 和 INSERT ... VALUES ...这个两个语句中，VALUE[S]关键字必须写正确，否则会出现解析正常，但结果错误。
这个是由于druid的bug引起的。当VALUES[S]关键字错误时，druid会把这个错误的关键字当作别名处理。
已向druid提[issue2218](#)。
dble相关issue [dble_issue_379](#).
2. 当开启黑名单时，`SHOW ALL TABLES` 等dble的自定义语句可能不支持
3. 其他
参见 [dble_issue_788](#)

5.2 其他已知限制

1. 显式配置父子表，在子表插入数据时，不能多值插入
原因：子表插入数据时,如有parentkey不是父表的拆分列。需要去对应父表的拆分结点中反向查询路由规则，如果此时是多值插入，就会变成多值反查，查询功能较难完成，即使完成，性能也会很差。
相关issue：<https://github.com/actiontech/dble/issues/12>
2. 不建议使用JDBC的rewriteBatchedStatements=true
原因：
insert：多条简单insert拼接成一条长的insert.. values(),(),一个com_query包，对于dble来说，可能引发分布式事务，降低性能和数据一致性。
3. 使用JDBC的useServerPrepStmts=true会降低性能
原因：dble是将前端的Binary Protocol 转为Text Protocol，收到结果集之后再反向转回协议，所以需要额外的工作，降低了性能。
4. lock/unlock 实现不完整
相关issue：<https://github.com/actiontech/dble/issues/38>
5. 不支持在schema.xml里配置复合主键（3.20.07.0移除此配置）
原因：schema.xml里的主键作用为主键路由和全局序列，这两者暂不支持多主键。
相关issue：<https://github.com/actiontech/dble/issues/70>
6. 不保证主键唯一性 由于dble不要求配置在table中的主键在创建表格时一定为主键，并且由于dble后端分布式存储的方式, dble不对于主键做唯一性检查，并且允许用户对于非分片列的主键字段进行任意更新
7. 并发更新多行数据/全局表数据可能导致死锁超时
原因：并发情况下，分布式下发sql可能乱序。
相关issue：<https://github.com/actiontech/dble/issues/85>
8. 防火墙导致无响应
原因：防火墙可能drop包，java层面的tcp_keepalive无法指定时间。
相关issue：<https://github.com/actiontech/dble/issues/87>
9. 方差/标准差精度问题
原因：方差计算方式导致
相关issue：<https://github.com/actiontech/dble/issues/100>
10. order by lock in share mode/for update , lock clause is ignored
原因：无法支持。
相关issue：<https://github.com/actiontech/dble/issues/127>
11. 不支持_charset_name 'string' _charset_name+b'val'
相关issue：<https://github.com/actiontech/dble/issues/262>
相关issue：<https://github.com/actiontech/dble/issues/267>
12. 未能正确支持 set sql_select_limit
相关issue：<https://github.com/actiontech/dble/issues/331>
13. 日期拆分算法中，sEndDate如果不配置，default node就无用
原因：算法设计问题。
相关issue：<https://github.com/actiontech/dble/issues/357>
14. selece @@sql_mode 始终包含IGNORE_SPACE
原因：后端权能标志位设置导致，参见4.2节内容。
相关issue：<https://github.com/actiontech/dble/issues/364>
15. replace ... into
由于replace的语义为如果存在则替换，如果不存在则新增，所以在使用表格自增主键的时候 如果对于自增表格使用replace且ID不存在，那么就会插入一条指定ID的数据，并不会自动生成ID
16. kill 语句杀自身session，直接返回ok，不会有任何实质性操作
17. 由于2.19.01版本在rule/schema/server.xml中加入了version字段造成的BUG，导致在2.19.01使用zk集群进行同步化操作的时候要求version字段不能为空或者不填，此问题在后续版本会进行修复
18. 若mysql节点上设置了 set global local_infile = 0 ,dble load data指令执行报错
原因：dble会将load data指令转换为 load data local infile ... 指令下发至各个后端mysql，所以各个节点 local_infile 参数需要开启。
相关issue：<https://github.com/actiontech/dble/issues/1111>
19. 不能正确支持 set @@sql_auto_is_null=on; 原因：set @@sql_auto_is_null=on 的语义是用新生成的自增序列取代null,在dble层不做实现。
相关issue：<https://github.com/actiontech/dble/issues/978>
20. 复杂查询会透传难以理解的报错，建议结合explain语句分析 原因：需要枚举所有错误来做替换.成本高，收益低 相关issue：<https://github.com/actiontech/dble/issues/1449>
21. 开启 enableCursor 选项后，在分库分表场景下，所有的 prepareStatement 的 prepare 阶段均需要向 mysql 执行特殊语句，用来获取结果集的列数并返回给前端，这个执行会损失一部分性能（即使客户端没有开启游标）。

22. 开启XA功能后，不支持执行含有隐式提交的sql

23. select 中的 group by 条件为true或false时，结果集不固定

原因：group by 条件为true或false时，mysql的结果集排序规则受存储引擎的影响。InnoDB情况下，优先按照PRIMARY KEY进行排序，没有PRIMARY KEY的前提下，按照INSERT顺序排序。在dble层面目前无法感知INSERT的顺序问题，所以无法做到和mysql行为一致

相关issue : <https://github.com/actiontech/dble/issues/3177>

6. 与MySQL Server的差异化描述

这里主要描述一些与MySQL Server不同的行为，这些行为不是bug，是分布式场景下的一些正常的行为表现，但与已知的MySQL行为不一致。

- [6.1 事务中遇到主键冲突需要显式回滚](#)
- [6.2 INSERT不能显式指定自增序列](#)
- [6.3 增加"show all tables"](#)
- [6.4 去除了增删改的message信息](#)
- [6.5 information_schema等库的支持](#)

6.1 事务中遇到主键冲突需要显式回滚，MySQL不需要

具体表现如下:

MYSQL行为:

```
[test_yhq]>select * from char_columns_4;
+-----+
| id | c_char |
+-----+
| 1 | xx |
| 4 | z |
+-----+
2 rows in set (0.02 sec)
[test_yhq]>begin;
Query OK, 0 rows affected (0.01 sec)

[test_yhq]>insert into char_columns_4 values(1,'yy');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
[test_yhq]>insert into char_columns_4 values(2,'yy');
Query OK, 1 row affected (0.00 sec)

[test_yhq]>commit;
Query OK, 0 rows affected (0.02 sec)
```

dble行为:

```
[testdb]>select * from sharding_four_node order by id;
+-----+-----+-----+
| id | c_flag | c_decimal |
+-----+-----+-----+
| 1 | 1_1 | 1.0000 |
| 2 | 2 | 2.0000 |
| 3 | 3 | 3.0000 |
+-----+-----+-----+
3 rows in set (0.28 sec)

begin;
Query OK, 0 rows affected (0.01 sec)

[testdb]>insert into sharding_four_node values(1,'1',1.0);
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
[testdb]>insert into sharding_four_node values(13,'13',13.0);
ERROR 1003 (HY000): Transaction error, need to rollback.Reason:[ errNo:1062 Duplicate entry '1' for key 'PRIMARY']
[testdb]>commit;
ERROR 1003 (HY000): Transaction error, need to rollback.Reason:[ errNo:1062 Duplicate entry '1' for key 'PRIMARY']
```

6.2 INSERT自增序列由dble生成，不能显式指定自增列，MySQL可以。

具体表现如下:

MySQL行为:

```
desc mysql_autoinc;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra      |
+-----+-----+-----+-----+-----+-----+
| c_char | char(255) | YES  |     | NULL    |            |
| id     | bigint(20)| NO   | PRI | NULL    | auto_increment |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.02 sec)

[test_yhq]>insert into mysql_autoinc values('1',1);
Query OK, 1 row affected (0.01 sec)
```

dble行为

```
desc sharding_four_node_autoinc;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra      |
+-----+-----+-----+-----+-----+-----+
| c_char | char(255) | YES  |     | NULL    |            |
| id     | bigint(20)| NO   | PRI | NULL    | auto_increment |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)
[testdb]>insert into sharding_four_node_autoinc values('2',2);
ERROR 1064 (HY000): In insert Syntax, you can't set value for Autoincrement column!
```


6.3 ADD "show all tables"

The optional ALL modifier causes SHOW TABLES to display a second output column with values of BASE TABLE for a table ,VIEW for a view, SHARDING TABLE for a sharding table and GLOBAL TABLE for a global table.

具体表现如下:

```
[testdb]>show all tables;
+-----+-----+
| Tables in testdb      | Table_type  |
+-----+-----+
| global_four_node     | GLOBAL TABLE |
| global_four_node_autoinc | GLOBAL TABLE |
| global_two_node      | GLOBAL TABLE |
| sbtest1              | SHARDING TABLE |
| sharding_four_node   | SHARDING TABLE |
| sharding_four_node2  | SHARDING TABLE |
| sharding_four_node_autoinc | SHARDING TABLE |
| sharding_two_node    | SHARDING TABLE |
| single               | SHARDING TABLE |
| customer             | BASE TABLE   |
| district             | BASE TABLE   |
+-----+-----+
11 rows in set (0.02 sec)
```

6.4 去除了增删改的message 信息

具体表现如下:

MySQL行为:

```
mysql> insert into sharding_two_node values(9, '9', 9.0), (10, '10', 10.0);  
Query OK, 2 rows affected (0.24 sec)  
Records: 2 Duplicates: 0 Warnings: 0
```

dble行为:

```
mysql> insert into sharding_two_node values(11, '11', 11.0), (12, '12', 12.0);  
Query OK, 2 rows affected (0.49 sec)
```

6.5 information_schema等库的支持

背景

用户使用 Navicat Premium 12 连接dble时，Navicat Premium 12会查询 information_schema，mysql 等数据库中系统表的数据。dble 需要支持查询这些系统表的语句，保证这些 driver 的正常使用。

Navicat Premium12

1. SELECT SCHEMA_NAME, DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME FROM information_schema.SCHEMATA;

该语句查询当前 mysql 实例中的所有 scheme的名称，character set 以及 collation。该语句会影响Navicat Premium 12的使用。

dble中的scheme是逻辑上的，可以通过 SchemeConfig 获取所有 scheme 的名称，scheme 的 character set 以及 collation 使用默认返回。

问题1：dble中的scheme是逻辑上的，会对应多个shardingNode，会出现shardingNode的 character set 以及 collation 与 默认character set 和 collation 不同,需要运维安装MySQL时候保证。

问题2：当前dble中处理 SCHEMATA 系统表时，只是对表名做了判断，并未对查询字段做检验。

以下语句不影响driver的使用

对此种语句的处理是根据请求字段，伪造一个行数为0的报文返回。

```

1. SELECT TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
   FROM information_schema.TABLES WHERE TABLE_SCHEMA = 'testdb'
   ORDER BY TABLE_SCHEMA, TABLE_TYPE
2. SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, COLUMN_TYPE
   FROM information_schema.COLUMNS
   WHERE TABLE_SCHEMA = 'testdb'
   ORDER BY TABLE_SCHEMA, TABLE_NAME
3. SELECT DISTINCT ROUTINE_SCHEMA, ROUTINE_NAME, PARAMS.PARAMETER
   FROM information_schema.ROUTINES LEFT JOIN
   ( SELECT SPECIFIC_SCHEMA, SPECIFIC_NAME,
     GROUP_CONCAT(CONCAT(DATA_TYPE, ' ', PARAMETER_NAME) ORDER BY ORDINAL_POSITION SEPARATOR ', ') PARAMETER, ROUTINE_TYPE
   FROM information_schema.PARAMETERS GROUP BY SPECIFIC_SCHEMA, SPECIFIC_NAME, ROUTINE_TYPE
   )PARAMS
   ON ROUTINES.ROUTINE_SCHEMA = PARAMS.SPECIFIC_SCHEMA AND
   ROUTINES.ROUTINE_NAME = PARAMS.SPECIFIC_NAME AND
   ROUTINES.ROUTINE_TYPE = PARAMS.ROUTINE_TYPE
   WHERE ROUTINE_SCHEMA = 'testdb' ORDER BY ROUTINE_SCHEMA
4. SELECT TABLE_NAME, CHECK_OPTION, IS_UPDATABLE, SECURITY_TYPE, DEFINER
   FROM information_schema.VIEWS
   WHERE TABLE_SCHEMA = 'testdb' ORDER BY TABLE_NAME ASC
5. SELECT * FROM information_schema.ROUTINES
   WHERE ROUTINE_SCHEMA = 'testdb' ORDER BY ROUTINE_NAME
6. SELECT EVENT_CATALOG, EVENT_SCHEMA, EVENT_NAME, DEFINER, TIME_ZONE,
   EVENT_DEFINITION, EVENT_BODY, EVENT_TYPE, SQL_MODE, STATUS, EXECUTE_AT,
   INTERVAL_VALUE, INTERVAL_FIELD, STARTS, ENDS, ON_COMPLETION, CREATED,
   LAST_ALTERED, LAST_EXECUTED, ORIGINATOR, CHARACTER_SET_CLIENT,
   COLLATION_CONNECTION, DATABASE_COLLATION, EVENT_COMMENT
   FROM information_schema.EVENTS WHERE EVENT_SCHEMA = 'testdb'
   ORDER BY EVENT_NAME ASC
7. SELECT COUNT(*) FROM information_schema.TABLES
   WHERE TABLE_SCHEMA = 'testdb' UNION
   SELECT COUNT(*)
   FROM information_schema.COLUMNS
   WHERE TABLE_SCHEMA = 'testdb' UNION
   SELECT COUNT(*) FROM information_schema.ROUTINES WHERE ROUTINE_SCHEMA = 'testdb'

```

7 开发者须知

- [7.1 SQL开发编写原则](#)
- [7.2 dble连接Demo](#)
- [7.3 其他注意事项](#)

7.1 SQL开发编写原则

- 分布式数据库处理的是分布式关系运算，其SQL的优化方法与单机关系数据库有所不同，侧重考虑的分布式环境中的网络开销。分布式执行计划中，应该尽量将SQL中的运算下推到底层各个节点执行，避免跨节点运算，从而减少网络开销、提升SQL执行效率。

1) 执行计划

访问数据时的一组有序的操作步骤集合，称为执行计划。dble的执行计划分为两个层次：dble层的执行计划与节点层的执行计划。对执行计划进行分析，可以了解中间件和节点是否对SQL语句生成了最优的执行计划，是否有优化的空间，从而为SQL优化提供重要的参考信息。在SQL语句执行前，dble会根据SQL语句的基本信息，判断该SQL语句应该在哪些节点上执行，将SQL改写成在节点上执行的具体形式，并决定采用何种策略进行数据合并与计算等，这就是dble层的执行计划。节点层的执行计划就是原生的MySQL执行计划。dble用EXPLAIN指令来查看dble层的执行计划。如：

```
explain select id,accountno from account where userid=2;
```

EXPLAIN指令的执行结果包括语句下发的节点，实际下发的SQL语句和数据的合并操作的信息。这些信息是系统静态分析产生的，并没有真正的执行语句。通过EXPLAIN2命令可查看指定节点上的执行计划。如：

```
explain2 shardingNode=dn1 sql=select id,accountno from account where userid=2;
```

explain2会将sql语句加上explain下发到指定的shardingNode执行，并把节点上explain的结果返回调用者。

2) SQL优化

对于SQL优化有以下原则：

- SQL语句中尽可能带有拆分字段，并且拆分字段过滤条件的取值范围越小，越有助于提高查询速度。在数据写入时，必须给拆分字段赋值。
- 拆分字段的查询条件尽可能是等值条件。
- 如果拆分字段的条件是IN子句，则IN后面的值的数目应尽可能少。特别注意，随着业务增长，某些IN子句的条件会随之增长。
- 如果SQL语句不带有拆分字段，那么DISTINCT、GROUP BY和ORDER BY在同一个SQL语句中尽量只出现一种。
- 数据查询时，应该尽量减少节点返回的结果数量，这样能够使消耗的网络带宽最小，使查询性能能够达到最优状态。

3) 跨节点查询

跨节点查询通常发生在下列语句中：

- 涉及多个分片的分布式Join；
- 涉及多个分片的聚合函数；
- 复杂子查询。

这些语句可以通过以下策略来优化：

- 合理调整查询语句，使查询条件中包含拆分字段的等值条件，这样就能够将语句下推到单一节点执行。
- 参与Join的表配置相同的拆分规则，查询时将拆分字段作为Join的关联条件，这样Join操作就可以在节点内完成。
- 将参与Join的表中数据量较小的表配置成全局表，通过数据冗余避免跨节点。
- 如果一定需要跨节点Join，尽量对驱动表添加更多的过滤条件，从而使参与跨节点Join的数据量尽可能的少。
- 如果查询涉及到了多个节点，则尽量不要使用limit a,b这样的子句
- GROUP子句尽量包含拆分字段。
- 对语句进行改写，将复杂子查询分解为多条语句来执行。
- 子查询尽可能改写成Join的形式。

7.2 dble连接Demo

开发框架连接

ibatis

利用ibatis连接dble时，连接方式与MySQL相同。下面是一个简单示例。 JDBC配置：

```
jdbc.driverClass=com.mysql.jdbc.Driver
jdbc.jdbcUrl=jdbc:mysql://127.0.0.1:8066/TESTDB?useUnicode=true&characterEncoding=utf-8
jdbc.user=root
jdbc.password=123456
```

映射文件配置：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.mapper.UserMapper">
  <insert id="saveUser" parameterType="com.bean.User">
    insert into user(id,name,phone,birthday)
    values (0,#{name},#{phone},#{birthday})
    <selectKey keyProperty="id" order="after" resultType="int">
      select last_insert_id() as id
    </selectKey>
  </insert>
  <delete id="deleteUserById" parameterType="java.lang.String">
    delete from user where id=#{id}
  </delete>
  <update id="updateUser" parameterType="com.bean.User">
    update user set name=#{name},phone=#{phone},birthday=#{birthday} where id=#{id}
  </update>
  <update id="updateUsers">
    /*!dble:sql=select * from user;*/update users set usercount=(select count(*) from user),ts=now()
  </update>
  <select id="getUserById" parameterType="java.lang.String" resultType="com.bean.User">
    select * from user where id=#{id}
  </select>
  <select id="getUsers" resultType="com.bean.User">
    select * from user
  </select>
```

语句select last_insert_id() as id可用来获取新写入记录的ID。 updateUsers方法用到了 dble的注解，由于ibatis中的符号#具有特殊含义，因此注解中不能含有#。

hibernate

利用hibernate连接dble时，连接方式与MySQL相同。下面是一个简单示例。 hibernate.cfg.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://192.168.58.51:8066/testdb?
useUnicode=true&characterEncoding=utf-8</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">123456</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
    <property name="hibernate.format_sql">>true</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <mapping resource="com/actiontech/test/News.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

News.hbm.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.actiontech.test.News" table="news_table">
    <id name="id" type="java.lang.Integer">
      <column name="id" />
    </id>
    <property name="title" type="java.lang.String">
      <column name="title" />
    </property>
    <property name="content" type="java.lang.String">
      <column name="content" />
    </property>
  </class>
</hibernate-mapping>
```

News.java:

```
package com.actiontech.test;
public class News {
  private Integer id;
  private String title;
  private String content;
  public Integer getId() {
    return id;
  }
  public void setId(Integer id) {
    this.id = id;
  }
  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
}
```

public String getContent() { return content; } public void setContent(String content) { this.content = content; } }</pre> NewsManager.java:

```
package com.actiontech.test;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class NewsManager {
  public static void main(String[] args)
    throws Exception {
    Configuration config = new Configuration().configure();
    SessionFactory factory = config.buildSessionFactory();
    Session session = factory.openSession();
    Transaction transaction = session.beginTransaction();
    News news = new News();
    news.setId(10);
    news.setTitle("dble示例");
    news.setContent("Hibernate 连接dble的第一个例子");
    session.save(news);
    transaction.commit();
    session.close();
    factory.close();
  }
}
```

dble虽然支持Hibernate但不建议使用Hibernate，因为Hibernate无法控制SQL的生成，无法做到对查询SQL的优化，大数量下可能会出现性能问题。

JDBC

利用JDBC连接dble时，连接方式与MySQL相同。下面是一个简单示例：

```

package com.actiontech.test;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicLong;
public class SingleMixEngine {
    public static void main(String[] args) throws Exception {
        Class.forName("com.mysql.jdbc.Driver");
        Properties props = new Properties();
        props.setProperty("user", "root");
        props.setProperty("password", "123456");
        SingleMixEngine engine = new SingleMixEngine();
        engine.execute(props,"jdbc:mysql://192.168.58.51:8066/testdb");
    }
    final AtomicLong tmAl = new AtomicLong();
    final String tableName="news_table";
    public void execute(Properties props,String url) {
        CountDownLatch cdl = new CountDownLatch(1);
        long start = System.currentTimeMillis();
        for (int i = 0; i < 1; i++) {
            TestThread insertThread = new TestThread(props,cdl, url);
            Thread t = new Thread(insertThread);
            t.start();
            System.out.println("Test start");
        }
        try {
            cdl.await();
            long end = System.currentTimeMillis();
            System.out.println("Test end,total cost:" + (end-start) + "ms");
        } catch (Exception e) {
        }
    }
}

class TestThread implements Runnable {
    Properties props;
    private CountDownLatch countDownLatch;
    String url;
    public TestThread(Properties props,CountDownLatch cdl,String url) {
        this.props = props;
        this.countDownLatch = cdl;
        this.url = url;
    }
    public void run() {
        Connection connection = null;
        PreparedStatement ps = null;
        Statement st = null;
        try {
            connection = DriverManager.getConnection(url,props);
            connection.setAutoCommit(true);
            st = connection.createStatement();
            String dropSql = "drop table if exists " + tableName;
            System.out.println("Execute SQL:\n\t"+dropSql);
            st.execute(dropSql);

            String createSql = "create table " + tableName + "(id int,title varchar(20),content varchar(50))";
            System.out.println("Execute SQL:\n\t"+createSql);
            st.execute(createSql);

            String insertSql = "insert into " + tableName + " (id,title,content) values(?,?,?)";
            System.out.println("Prepared SQL:\n\t"+insertSql);
            ps = connection.prepareStatement(insertSql);
            for (int i = 1; i <= 3; i++) {
                ps.setInt(1,i);
                ps.setString(2, "测试"+i);
                ps.setString(3, "这是第"+i+"条测试数据");
                ps.execute();
                System.out.println("Insert data:\t"+i+", "+"测试"+i+", "+"这是第"+i+"条测试数据");
            }
        }
    }
}

```



```

String querySQL = "select * from " + tableName + " order by id";
System.out.println("Execute SQL:\n\t"+querySQL);
ResultSet rs = st.executeQuery(querySQL);
int colcount = rs.getMetaData().getColumnCount();
System.out.println("Current Data:");
while(rs.next()){
    for(int i=1;i<=colcount;i++){
        System.out.print("\t"+rs.getString(i));
    }
    System.out.println();
}

String updateSql = "update " + tableName + " set title='test1' where id=1";
System.out.println("Execute SQL:\n\t"+updateSql);
st.execute(updateSql);
rs = st.executeQuery(querySQL);
System.out.println("Current Data:");
while(rs.next()){
    for(int i=1;i<=colcount;i++){
        System.out.print("\t"+rs.getString(i));
    }
    System.out.println();
}

String deleteSql = "delete from " + tableName + " where id=2";
System.out.println("Execute SQL:\n\t"+deleteSql);
st.execute(deleteSql);
rs = st.executeQuery(querySQL);
System.out.println("Current Data:");
while(rs.next()){
    for(int i=1;i<=colcount;i++){
        System.out.print("\t"+rs.getString(i));
    }
    System.out.println();
}

String createIndexSql = "create index idx_1 on " + tableName + "(title)";
System.out.println("Execute SQL:\n\t"+createIndexSql);
st.execute(createIndexSql);

String dropIndexSql = "drop index idx_1 on " + tableName;
System.out.println("Execute SQL:\n\t"+dropIndexSql);
st.execute(dropIndexSql);
} catch (Exception e) {
    System.out.println(new java.util.Date().toString());
    e.printStackTrace();
} finally {
    if (ps != null)
        try {
            ps.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    if (connection != null)
        try {
            connection.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    this.countDownLatch.countDown();
}
}
}
}
}

```

7.3 其他注意事项

- 在应用分布式数据库时，首先要考虑清楚，是否真的需要分片，在所有环境下分片都是不得已的选择，必然会增加研发、运维、管理的复杂度，现在有很多技术例如分区、复制、缓存等都能提高系统的处理速度和吞吐量，也许我们应用这些技术能够实现技术目标而无需分片
- 数据分片之前，请确保已经对单机数据库做了优化，包括系统架构、硬件、MySQL版本、数据读写模式等，数据分片之后这些优化依然非常有意义
- 配置拆分规则时，还应该同时考虑运维的复杂性，以及未来系统的扩容。
- 数据量较大时，DDL语句会耗时较多，分布式环境下这个问题会更加突出，因此应尽量在业务空闲时进行DDL操作。

8 配置示例

- [8.1 时间戳方式全局序列的配置](#)
- [8.2 MySQL-offset-step 方式全局序列的配置](#)

8.1 时间戳方式全局序列的配置

配置表tb的id列为时间戳方式全局序列，并按id列分片

1) cluster.conf

```
sequenceHandlerType=2
sequenceStartTime=2010-10-01 09:42:54
...
```

2) bootstrap.conf

```
instanceId=1
...
```

3) user.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dbtle:user xmlns:dbtle="http://dbtle.cloud/">
  <managerUser name="test" password="test"/>
  <shardingUser name="abc" password="abc" schemas="myschema" maxCon="1000000">
  </shardingUser>
</dbtle:user>
```

4) db.xml

```
<?xml version="1.0"?>
<dbtle:db xmlns:dbtle="http://dbtle.cloud/" >
  <dbGroup name="host_1" rwSplitMode="0" delayThreshold="10000">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM1" url="172.100.10.101:3306" user="test1" password="test1" maxCon="1000" minCon="1000" primary="true" />
  </dbGroup>
  <dbGroup name="host_2" rwSplitMode="0" delayThreshold="10000">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM2" url="172.100.10.102:3306" user="test1" password="test1" maxCon="1000" minCon="1000" primary="true" />
  </dbGroup>
</dbtle:db>
```

5) sharding.xml

```
<?xml version="1.0"?>
<dbtle:sharding xmlns:dbtle="http://dbtle.cloud/">
  <schema name="myschema" shardingNode="dn1">
    <shardingTable name="sbtest1" shardingNode="dn1,dn2" function="mod" shardingColumn="id" incrementColumn="id" />
  </schema>
  <shardingNode name="dn1" dbGroup="host_1" database="dbtle"/>
  <shardingNode name="dn2" dbGroup="host_2" database="dbtle"/>
  <function name="mod" class="Hash">
    <property name="partitionCount">2</property>
    <property name="partitionLength">1</property>
  </function>
</dbtle:sharding>
```

6) 实验

```
``mysql mysql -utest -p111111 -h127.0.0.1 -P8066 -Dmyschema mysql> drop table if exists sbtest1; Query OK, 0 rows affected (0.05 sec) mysql> create table sbtest1(id bigint(20), k int unsigned not null default '0', primary key(id)); Query OK, 0 rows affected (0.05 sec)
```

```
mysql> insert into sbtest1 values(2); Query OK, 1 row affected (0.11 sec)
```

```
mysql> select * from sbtest1; 查看序列
```

注意事项，自增列的数据类型要是bigint

8.2 MySQL-offset-step 方式全局序列的配置

配置表sbtest1的id列为MySQL-offset-step方式全局序列，并按id列分片

1) cluster.conf

```
sequenceHandlerType=1
...
```

2) user.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dbtle:user xmlns:dbtle="http://dbtle.cloud/" >
  <managerUser name="test" password="test"/>

  <shardingUser name="abc" password="abc" schemas="myschema" maxCon="1000000">
    </shardingUser>

</dbtle:user>
```

3) db.xml

```
<?xml version="1.0"?>
<dbtle:db xmlns:dbtle="http://dbtle.cloud/" >
  <dbGroup name="host_1" rwSplitMode="0" delayThreshold="10000">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM1" url="172.100.10.101:3306" user="test1" password="test1" maxCon="1000" minCon="1000" primary="true" />
  </dbGroup>
  <dbGroup name="host_2" rwSplitMode="0" delayThreshold="10000">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM2" url="172.100.10.102:3306" user="test1" password="test1" maxCon="1000" minCon="1000" primary="true" />
  </dbGroup>
</dbtle:db>
```

4) sharding.xml

```
<?xml version="1.0"?>
<dbtle:sharding xmlns:dbtle="http://dbtle.cloud/" >
  <schema name="myschema" shardingNode="dn1">
    <shardingTable name="sbtest1" shardingNode="dn1,dn2" function="mod" shardingColumn="id" incrementColumn="id" />
  </schema>
  <shardingNode name="dn1" dbGroup="host_1" database="dbtle"/>
  <shardingNode name="dn2" dbGroup="host_2" database="dbtle"/>

  <function name="mod" class="Hash">
    <property name="partitionCount">2</property>
    <property name="partitionLength">1</property>
  </function>
</dbtle:sharding>
```

5) sequence_db_conf.properties

```
#sequence stored in shardingNode
`myschema`.`sbtest1`=dn1
```

myschema, sbtest1, dn1均为在sharding.xml配置的值

在dn1分片对应的后端host_1/dbtle上执行dbtle安装目录下的conf/dbseq.sql(路径根据实际情况自行修改)。

```
mysql -h172.100.10.101 -utest1 -ptest1 -Ddbtle
mysql>source conf/dbseq.sql
```

在上述sql文件执行成功后向创建的表DBLE_SEQUENCE插入自增相关的配置数据：

```
mysql -h172.100.10.101 -utest1 -ptest1 -Ddbtle
mysql>INSERT INTO DBLE_SEQUENCE VALUES ('`myschema`.`sbtest1`', 16, 1);
```

DBLE_SEQUENCE列说明:

- name: 在sequence_db_conf.properties中配置的逻辑数据库和表名
- current_value: 全局序列的当前值
- increment: 每次取出多少值用于全局序列, 注意全局序列递增的步长固定是1

6) 实验

登录db1业务端口创建设置了全局序列并以其分片的表:

```
mysql -utest -p111111 -h127.0.0.1 -P8066 -Dmyschema
mysql> drop table if exists sbtest1;
Query OK, 0 rows affected (0.05 sec)
mysql> create table sbtest1(id int, k int unsigned not null default '0', primary key(id));
Query OK, 0 rows affected (0.05 sec)

mysql> insert into sbtest1 values(2);
Query OK, 1 row affected (0.11 sec)

mysql> select * from sbtest1;
+-----+
| id | k |
+-----+
| 17 | 2 |
+-----+
1 row in set (0.01 sec)
```

从上面的sql可以看到, 在设置DBLE_SEQUENCE表时, current_value设置的是16, 在insert后变为了17。

配置要点:

- sequence_db_conf.properties:

```
`myschema`.`sbtest1`=dn1
```

- 在sequence_db_conf.properties配置的后端分片dn1对应的后端数据库上执行dbseq.sql, 并插入全局序列表对应的记录

9 sysbench压测dble示例

- [9.1 测试环境及架构](#)
- [9.2 修改dble配置](#)
- [9.3 使用sysbench进行压测](#)

9.1 测试环境及架构

版本信息

- Sysbench version: 1.0
- Dble version: 5.6.23-dble-2.19.11.0-2d7c4911b7a4fecaa9eb0299f49c32ec11e97c42-20200228124218
- MySQL version: 5.7.25

测试架构

- sysbench独立服务器 (172.20.134.1)
- dble独立服务器 (172.20.134.2)
- 3台mysql独立服务器 (172.20.134.3, 172.20.134.4, 172.20.134.5)

9.2 修改dble配置

说明：此配置仅为示例配置，并非调优配置，请根据运行环境自行调优，调优步骤参考：[2.18 性能观测以及调试概览](#)

1. bootstrap.cnf

调整如下参数 -DNIOFrontRW=10 -DNIOBackendRW=10 -DfrontWorker=8 -DbackendWorker=6 -DsqlExecuteTimeout=3000000

2.user.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dble:user xmlns:dble="http://dble.cloud/" version="4.0">
  <managerUser name="root" password="111111" />
  <shardingUser name="test" password="111111" schemas="sbtest" maxCon="1000000">
  </shardingUser>
</dble:user>
```

3.db.xml

```
<?xml version="1.0"?>
<dble:db xmlns:dble="http://dble.cloud/" version="4.0">
  <dbGroup name="host_1" rwSplitMode="0" delayThreshold="-1">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM1" url="172.20.134.3:3306" user="test1" password="test1" maxCon="1000" minCon="100" primary="true"/>
  </dbGroup>
  <dbGroup name="host_2" rwSplitMode="0" delayThreshold="-1">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM1" url="172.20.134.4:3306" user="test1" password="test1" maxCon="1000" minCon="100" primary="true"/>
  </dbGroup>
  <dbGroup name="host_3" rwSplitMode="0" delayThreshold="-1">
    <heartbeat >select USER()</heartbeat>
    <dbInstance name="hostM1" url="172.20.134.5:3306" user="test1" password="test1" maxCon="1000" minCon="100" primary="true"/>
  </dbGroup>
</dble:db>
```

4.sharding.xml

```
<?xml version="1.0"?>
<dble:sharding xmlns:dble="http://dble.cloud/" version="4.0">
<schema name="sbtest">
  <shardingTable name="sbtest1" shardingNode="dn$1-9" function="hash-sysbench" shardingColumn="id" />
</schema>
<shardingNode name="dn$1-3" dbGroup="host_1" database="dbledb$1-3" />
<shardingNode name="dn$4-6" dbGroup="host_2" database="dbledb$4-6" />
<shardingNode name="dn$7-9" dbGroup="host_3" database="dbledb$7-9" />
<function name="hash-sysbench" class="Hash">
  <property name="partitionCount">9</property>
  <property name="partitionLength">1</property>
</function>
</dble:sharding>
```

5.在后端mysql节点创建相应物理库

- 172.20.134.3:3306 创建库：dbledb1, dbledb2, dbledb3
- 172.20.134.4:3306 创建库：dbledb4, dbledb5, dbledb6
- 172.20.134.5:33066 创建库：dbledb7, dbledb8, dbledb9

9.3 使用sysbench进行压测

说明：

如果在压测中遇到问题请参考：[how about a sysbench-testing-quick-start](#)；如果在此参考中未包括您遇到的问题请在issue下添加评论，将问题及配置做出说明。

注意：

使用sysbench压测语句，在高并发下会出现主键冲突的报错。

```
FATAL: mysql_drv_query() returned error 1062 (Duplicate entry '49823' for key 'PRIMARY') for query 'INSERT INTO sbtest1 (id, k, c, pad) VALUES (49823, 58210, '27111667985-11552069038-79242882109-05602914209-02374993639-32242662584-65155028223-08319627673-44873060047-22215118936', '07405724915-32799061660-96650146042-59717172693-66753749407')'
```

解决方式见如上链接。

数据清理：

```
/usr/share/sysbench/oltp_read_write.lua --mysql-db=sbtest --mysql-host=172.20.134.2 --mysql-port=8066 --mysql-user=test --mysql-password=111111 --auto_inc=off --tables=1 --table-size=100000 --threads=4 --time=30 --report-interval=1 --max-requests=0 --percentile=95 --db-ps-mode=disable --skip-trx=on cleanup
```

数据准备：

```
/usr/share/sysbench/oltp_read_write.lua --mysql-db=sbtest --mysql-host=172.20.134.2 --mysql-port=8066 --mysql-user=test --mysql-password=111111 --auto_inc=off --tables=1 --table-size=100000 --threads=4 --time=30 --report-interval=1 --max-requests=0 --percentile=95 --db-ps-mode=disable --skip-trx=on prepare
```

执行压测：

```
/usr/share/sysbench/oltp_read_write.lua --mysql-db=sbtest --mysql-host=172.20.134.2 --mysql-port=8066 --mysql-user=test --mysql-password=111111 --auto_inc=off --tables=1 --table-size=100000 --threads=4 --time=30 --report-interval=1 --max-requests=0 --percentile=95 --db-ps-mode=disable --skip-trx=on run
```

- [A.1 ErrorCode](#)
- [A.2 原理解释](#)
- [A.3 使用说明](#)

- [Max Connections](#)
- [Out Of Memory Error](#)
- [The Problem Of Hint](#)
- [NestLoop Parameters Lead To Temptable Exception](#)
- [Can't Get Variables From ShardingNode](#)
- [Port Already In Use 1984](#)
- [Sharding Column Cannot Be Null](#)

dbLe-MaxConnections

Issue

- [Err] 3009 - java.io.IOException: the max activeConnections size can not be max than maxconnections.

Resolution

1. 检查是否使用了过多的短链，导致频繁建立连接，链接池不够用

注意：不要过多的使用短链，容易消耗dbLe资源

2. maxCon配置过小，调大maxCon

配置名称	配置内容	默认值/单位	作用原理或应用
maxCon	控制最大连接数	默认1024	大于此连接数之后,建立连接会失败. 注意当各个用户的maxcon总和值大于此值时,以当前值为准 全局 maxCon不作用于manager用户

Root Cause

配置的链接池数量不够用，导致报错。

Relevant Content

数据库连接池

1. **maxWait**

从连接池获取连接的超时等待时间，单位毫秒。

注意：maxActive不要配置过大，虽然业务量飙升后还能处理更多的请求，但其实连接数的增多在很多场景下反而会减低吞吐量

2. **connectionProperties**

可以配置 connectTimeout 和 socketTimeout，单位都是毫秒。connectTimeout 配置建立 TCP 连接的超时时间；socketTimeout 配置发送请求后等待响应的超时时间。

注意：不设置这两项超时时间，服务会有高的风险。

例如网络异常下 socket 没有办法检测到网络错误，如果没有设置 socket 网络超时，连接就会一直等待 DB 返回结果，造成新的请求都无法获取到连接。

3. **maxActive**

最大连接池数量，允许的最大同时使用中的连接数。

注意：当业务出现大流量涌入时，连接池耗尽，maxWait未配置或者配置为 0 时将无限等待，导致等待队列越来越长，表现为业务接口大量超时，实际吞吐越低。

dbld-OutOfMemoryError

Setting

load data语句，一共有17G的数据，每条语句4K

load data相关参数值均为默认

Issue

- INFO | jvm 1 | 2019/06/28 14:55:37 | Exception in thread "backendBusinessExecutor17" java.lang.OutOfMemoryError: GC overhead limit exceeded.

Special instructions: backendBusinessExecutor17 被替换成了17-backendWorker

Resolution

- 调小maxRowSizeToFile参数值，减少占用内存量

配置名称	配置内容	默认值	详细作用原理或应用
maxCharsPerColumn	每列所允许最大字符数	默认为65535	每行所允许最大字符数
maxRowSizeToFile	需要持久化的最大行数	默认为10000	当load data的数据行数超过阈值后，会将数据保存在文件中以防OOM

- 调大wrapper.conf中的Xmx值

注意：发生OOM，可以关注一下这个值的大小是否配置合适

Root Cause

- maxRowSizeToFile参数设置过大，对于机器来说超出了承载的数据量，导致内存溢出
- 配置文件中Xmx设置过小，内存不够

注意：On-Heap 大小由JVM 参数Xms ,Xmx 决定，就是正常服务需要的内存，由jvm自动分配和回收。

Relevant Content

load data相关配置

设置load data相关参数时，不要过度的调大，防止OOM

JVM配置

以下为建议值：

- dbld总内存=0.6 可用物理内存(刨除操作系统,驱动等的占用)
- Xmx = 0.4 dbld总内存
- MaxDirectMemorySize = 0.6 * dbld总内存

堆内存分配

- JVM初始分配的内存由-Xms指定，默认是物理内存的1/64；
- JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4。
- 默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制。

dble-TheProblemOfHint

Setting

- mysql> `!#dble:sql=select 1 from rp_cre_data_mobile_track_cmcc / call update_track();`
- 预期：表为分片表，sql语句应该下发到所有节点
- 结果：只有一个节点在执行

Issue

- mysql> `show @@processlist;`

Front_Id	shardingNode	BconnID	user	Front_Host	db	Command	Time	State	Info
33	NULL	NULL	root	略	NULL	NULL	0	updating	NULL
34	NULL	NULL	root	略	NULL	NULL	0	updating	NULL
35	NULL	NULL	root	略	NULL	NULL	0	updating	NULL
41	dn9	9372	root	略	db9	Query	0	updating	NULL
42	NULL	NULL	root	略	NULL	NULL	0	updating	NULL
43	NULL	NULL	root	略	NULL	NULL	0	updating	NULL
30	NULL	NULL	admin	略	NULL	NULL	0	updating	NULL

Resolution

- 将注解方式：`!#dble:type=.../`
改为：`!dble:type=.../`
- 或者在mysql client端，加上 `-c` 选项

注意：mysql --help

`-c, --comments`

Preserve comments. Send comments to the server. The default is --skip-comments (discard comments), enable with --comments.

Root Cause

- `!#dble:type=.../` 这种注释方式是mysql的标准注释
- 如果不加 `-c` 选项，默认注释会被skip

Relevant Content

hint作用

- 指定路由，比如强制读写分离。
- 帮助dble支持一些不能实现的语句，如单节点内存储过程的创建和调用，如`insert...select...`；

Hint语法

Hint语法有三种形式：

- `!dble:type=...*`
- `!#dble:type=...*`
- `!*`(只适用于读写分离功能)

type有4种值可选：shardingNode, db_type, sql, db_instance_url。

type详情请见：https://actiontech.github.io/dble-docs-cn/2.Function/2.04_hint.html

Hint注意事项

- 使用select语句作为注解SQL，不要使用delete/update/insert等语句。delete/update/insert等语句虽然也能用在注解中，但这些语句在SQL处理中有一些额外的逻辑判断，会降低性能，不建议使用
- 注解SQL 禁用表关联语句
- 使用hint做DDL需要额外执行`reload @@metadata`
- 使用hint做session级别的系统变量和环境变量可能不会生效，请慎用
- dble的注解和MySQL原生注解含义不同，想通过MySQL原生注解来设置变量或者指定索引是无法得到预期结果的。如#1169
- 使用注解并不额外增加的执行时间；从解析复杂度以及性能考虑，注解SQL应尽量用最简单的SQL语句，如`select id from tab_a where id='10000'`；
- 能不用注解也能够解决的场景，尽量不用注解

dbLe-NestLoop Parameters Lead To Temptable Exception

Setting

- 开启了NestLoop优化，设置NestLoop值的范围为4
- 两个表join关联：student表和class表
- student表结构：id列、name列、class_name列，主键id
- class表结构：id列，class_name列、teacher_name列
- ```
SELECT class.teacher_name FROM student LEFT JOIN class on student.class_name=class.class_name WHERE student.name="张三";
```

### Issue

- com.actiontech.dbLe.plan.common.exception.TempTableException: temptable too much rows,[rows size is 5].

### Resolution

- 调大NestLoop中的默认参数值，如下：

| 配置名称             | 配置内容           | 默认值/单位 | 详细作用原理或应用                            |
|------------------|----------------|--------|--------------------------------------|
| useJoinStrategy  | 是否使用nestLoop优化 | 默认不使用  | 开启之后会尝试判断join两边的where来重新调整查询SQL下发的顺序 |
| nestLoopConnSize | 临时表阈值          | 默认4    | 若临时表行数大于这两个值乘积，则报告错误                 |
| nestLoopRowsSize | 临时表阈值          | 默认2000 | 若临时表行数大于这两个值乘积，则报告错误                 |

- 或者关闭NestLoop，不使用其优化，如下：
  - ```
<property name="useJoinStrategy">false</property>
```
- 为了更直观，本例提前调小参数值，引发场景复现

Root Cause

- 使用了NestLoop优化，但是依据NestLoop规则选择出的表数据量太大，超出了NestLoop默认范围
- NestLoop优化规范

注意：

- 对于两表join时，NestLoop选择小表作为驱动表（同样适用于多join）
- 判断依据：是否有where条件，有where条件的作为小表
- 当两个表都有where条件或者都没有where条件，NestLoop无法判断，不起作用
- 在不确定哪个表为小表时，不建议开启NestLoop
- 针对本例SQL做详细说明：如下
 - ```
SELECT class.teacher_name FROM student LEFT JOIN class on student.class_name=class.class_name WHERE student.name="张三";
```

根据dbLe中NestLoop优化规则可知：

1. SQL中使用了student表和class表
2. where条件限制指定于student表，所以NestLoop可以根据规则选择出student表作为驱动表（小表）
3. 实际SQL中student表的数据量较大，超出了NestLoop值的范围，引发报错

### Relevant Content

#### MySQL的多表连接之NestLoop介绍

1. NestLoop:
  - 对于被连接的数据子集较小的情况，Nested Loop是个较好的选择。
  - Nested Loop就是扫描一个表（外表），每读到一条记录，就根据Join字段上的索引去另一张表（内表）里面查找，若Join字段上没有索引查询优化器一般就不会选择Nested Loop。
  - 在Nested Loop中，内表（一般是带索引的大表）被外表（也叫“驱动表”，一般为小表——不紧相对其它表为小表，而且记录数的绝对值也较小，不要求有索引）驱动，外表返回的每一行都要在内表中检索找到与它匹配的行，因此整个查询返回的结果集不能太大。
2. NestLoop优缺点

| 类别       | 使用条件 | 相关资源      | 特点                                     | 缺点                                 |
|----------|------|-----------|----------------------------------------|------------------------------------|
| NestLoop | 任何条件 | CPU、磁盘I/O | 当有高选择性索引或进行限制性搜索时效率比较高，能够快速返回第一次的搜索结果。 | 当索引丢失或者查询条件限制不够时，效率很低；当表记录数多时，效率低。 |



## dble-Can't get variables from shardingNode

### Setting

- db.xml 片段

```
<dbGroup name="localhost1" rwSplitMode="0" delayThreshold="10000">
<heartbeat>show slave status</heartbeat>
<dbInstance host="hostM1" url="localhost:3306" user="root" password="nE7jA%5m" maxCon="1000" minCon="10" primary="true" > </dbInstance>
</dbGroup>
<dbGroup name="localhost2" rwSplitMode="0" delayThreshold="10000">
<heartbeat>show slave status</heartbeat>
<dbInstance host="hostM2" url="localhost:3306" user="root" password="nE7jA%5m" maxCon="1000" minCon="10" primary="true"> </dbInstance>
</dbGroup>
```

### Issue

- 查看dble启动日志:

```
Running dble-server...
wrapper | --> Wrapper Started as Console
wrapper | Launching a JVM...
jvm 1 | Wrapper (Version 3.2.3)
http://wrapper.tanukisoftware.org
jvm 1 | Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.
jvm 1 |
jvm 1 | java.io.IOException:Can't get variables from shardingNode ...
wrapper | <-- Wrapper Stopped
```

### Resolution

1. 检查mysql版本及是否正常启动, 不支持mysql5.1, 请升级mysql, 如果启动正常见下一步;
2. db.xml中的root用户能否通过配置文件的信息连接mysql, 连接成功见下一步;
3. 检查root用户权限;
4. 连接mysql, 并执行show variables命令, 未执行成功见下一步;
5. 修改配置文件中dbGroup指定的后端数据库密码, 更新配置文件, dble正常启动

### Root Cause

- 通过db.xml 配置信息成功连接mysql后, 并不能执行show variables
- 由于mysql 5.7 初始化之后, 首次使用随机密码登陆, 没有修改密码, 无法对数据库进行操作

### Relevant Content

1. 安装好mysql5.7后, 第一次初始化数据库
2. 随机密码登录mysql, 首次登录后, mysql要求必须修改默认密码, 否则不能执行任何其他数据库操作, 这样体现了不断增强的Mysql安全性。
3. 第一次登陆后必须更改密码:
  - mysql> show databases;
  - ERROR 1820 (HY000): You must reset your password using ALTER USER statement before executing this statement.
  - mysql > set password = password('xxxxx');

## dbld-Port already in use:1984

### Issue

- wrapper.log-Error1

```
STATUS | wrapper | 2019/07/23 16:37:06 | --> Wrapper Started as Daemon
STATUS | wrapper | 2019/07/23 16:37:06 | Launching a JVM...
INFO | jvm 1 | 2019/07/23 16:37:06 | OpenJDK 64-Bit Server VM warning: ignoring option MaxPermSize=64M; support was removed in 8.0
INFO | jvm 1 | 2019/07/23 16:37:07 | 错误: 代理抛出异常错误: java.rmi.server.ExportException: Port already in use: 1984; nested exception
is:
INFO | jvm 1 | 2019/07/23 16:37:07 | java.net.BindException: Address already in use (Bind failed)
INFO | jvm 1 | 2019/07/23 16:37:07 | sun.management.AgentConfigurationException: java.rmi.server.ExportException: Port already in use: 198
4;
```

- wrapper.log-Error2

```
STATUS | wrapper | 2019/07/26 16:12:48 | --> Wrapper Started as Daemon
STATUS | wrapper | 2019/07/26 16:12:49 | Launching a JVM...
INFO | jvm 1 | 2019/07/26 16:12:49 | Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
INFO | jvm 1 | 2019/07/26 16:12:49 | Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.
INFO | jvm 1 | 2019/07/26 16:12:49 |
INFO | jvm 1 | 2019/07/26 16:12:51 | java.net.BindException: Address already in use
INFO | jvm 1 | 2019/07/26 16:12:51 | at sun.nio.ch.Net.bind0(Native Method)
INFO | jvm 1 | 2019/07/26 16:12:51 | at sun.nio.ch.Net.bind(Net.java:433)
INFO | jvm 1 | 2019/07/26 16:12:51 | at sun.nio.ch.Net.bind(Net.java:425)
INFO | jvm 1 | 2019/07/26 16:12:51 | at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:223)
INFO | jvm 1 | 2019/07/26 16:12:51 | at com.actiontech.dble.net.NIOAcceptor.<init>(NIOAcceptor.java:46)
```

### Resolution

- 根据Error1:  
修改配置文件wrapper.conf: 修改被占用端口1984  
-Dcom.sun.management.jmxremote.port=1984
- 根据Error2:  
netstat -nap 查看程序运行的pid (8066和9066依然存在)  
kill -9 pid 杀掉进程
- 启动dbld成功

### Root Cause

- 已经启动过一个开启jmx服务的java程序后, 再启动dbld会报这个错;
- dbld启动过程中会占用三个端口: 业务端口, 管理端口, jvm对外提供jmx服务端口;
- jmx可通过jconsole连接上jvm, 观测jvm的运行状态。

### Relevant Content

#### 1. JVM

JVM是一种使用软件模拟出来的计算机, 它用于执行Java程序, 有一套非常严格的技术规范, 是Java跨平台的依赖基础。Java虚拟机有自己想象中的硬件, 如处理器, 堆栈, 寄存器等, 还有相应的指令系统它允许Java程序就好像一台计算机允许c或c++程序一样。

#### 2. JMX

所谓JMX, 是Java Management Extensions的缩写, 是Java管理系统的一个标准、一个规范, 也是一个接口, 一个框架。它和JPA、JMS是一样的, 就是通过将监控和管理涉及到的各个方面的问题和解决办法放到一起, 统一设计, 以便向外提供服务, 以供使用者调用。

#### 3. Jconsole

Jconsole是JDK自带的监控工具, 它用于连接正在运行的本地或者远程的JVM, 对运行在java应用程序的资源消耗和性能进行监控, 并画出大量的图表, 提供强大的可视化界面。

自身占用的服务器内存很小, 甚至可以说几乎不消耗。

## dbLe-Sharding column can't be null

### Setting

- sharding.xml部分配置如下：

```
<shardingTable shardingColumn="number" ... >
...
<function name="rangeLong" class="NumberRange">
 <property name="mapFile">partition.txt</property>
 <property name="defaultNode">0</property>
</function>
```

- create table account (id int(10),number int(10) not null,name varchar(20) not null);
- insert into account (id,number,name) values (1,NULL,'aaa');

### Issue

ERROR 1064 (HY000): Sharding column can't be null when the table in MySQL column is not null

### Resolution

- number列和name列的插入值不为NULL；
- 或者修改number列为允许插入NULL值；  
ALTER TABLE account MODIFY number VARCHAR (20);
- 注意：**上一步的前提是：  
在blacklist中开启参数alterTableAllow;

```
<blacklist name="bk1">
 <property name="alterTableAllow">>true</property>
</blacklist>
```

并修改sharding-by-range中的拆分列，dbLe不允许对分片键或ER键进行alter，会造成无法分片；

```
<shardingTable shardingColumn="id" ... >
```

- alter列值为允许插入空值后，再将拆分列修改为原值。

### Root Cause

- 在MySQL中执行相同insert：  
报错：ERROR 1048 (23000): Column 'number' cannot be null
- desc查看表结构：number列和name列均定义为非空列，不允许插入空值。

Field	Type	Null	Key	Default	Extra
id	int(10)	YES		NULL	
number	int(10)	NO		NULL	
name	varchar(20)	NO		NULL	

- [How To Use Explain To Resolve The Distribution Rules Of Group Gy](#)
- [Hash And ConsistentHashing And Jumpstringhash](#)

# dbpedia-How To Use Explain To Resolve The Distribution Rules Of Group Gy

## Questions

一张数据表做了分表。如果查询里有group by分组统计，运行原理是按范围去各分表查询出数据后，再到中间件里进行分组统计的吗？

## Conclusions

- 会在中间件中做数据重聚合
  1. 利用explain工具查看sql的执行过程
  2. dbpedia 在explain上做了大量改善，相比mycat能提供更详实的执行计划，更准确的反映SQL语句的执行过程

## For Example

1. 配置好配置文件：

sharding.xml：

```
<shardingTable name="eee" shardingNode="dn1,dn2" function="hashLong" shardingColumn="id"/>
...
<function name="hashLong" class="Hash">
 <property name="partitionCount">2</property>
 <property name="partitionLength">128</property>
</function>
```

1. 在dbpedia client创建表 eee 并插入数据：mysql> select \* from eee;

```
| id | name | -- | -- | 1 | 上海 | 2 | 广州 | 3 | 杭州 | 4 | 北京 | 5 | 北京 | 130 | 北京 | 131 | 北京 | 132 | 上海 | 133 | 上海 | 134 | 上海 |
```

mysql> select name,count(name) from eee group by name;

name	COUNT(name)
上海	4
北京	4
广州	1
杭州	1

1. 利用explain工具查看sql的执行过程

SHARDING_NODE	TYPE	SQL/REF
dn1_0	BASE SQL	select eee . name ,COUNT(name) as _\$COUNT\$_rpda_0 from eee GROUP BY eee . name ASC
dn2_0	BASE SQL	select eee . name ,COUNT(name) as _\$COUNT\$_rpda_0 from eee GROUP BY eee . name ASC
merge_1	MERGE	dn1_0; dn2_0
aggregate_1	AGGREGATE	merge_1
shuffle_field_1	SHUFFLE_FIELD	aggregate_1

## Instructions

由explain的结果可知：

1. dbpedia将sql语句下发到对应shardingnode执行
2. 将对应shardingnode数据结果进行merge
3. 对merge后的数据进行group by聚合
4. SHUFFLE\_FIELD进行整理，达到用户预期的结果
  - 注意：普通用户可以不关注SHUFFLE\_FIELD

## Relevant Content

### dbpedia的内部功能层

1. 在dbpedia内部，包括了三个部分：面向app的连接层，内部功能层，面向mysql的连接池。
2. 内部功能层实现：前端请求接收，处理后由后端协议层发出，将数据返回给用户。

内部功能涉及到了简单查询和复杂查询：

- 简单查询：直接下发单个/多个节点
- 复杂查询：dble内部需要进行排序、聚合、join、group by，结果集计算
- 详细介绍：<https://opensource.actionsky.com/dble-lesson-one/>

#### **dble内部各线程池简介**

1. 后端IO接收线程 处理来自MySQL连接的网络包（sql的执行结果、查询的结果集）。
2. 后端业务处理线程 简单查询的后端MySQL返回处理，并将结果转发反馈给客户端。
3. 复杂查询处理线程 处理复杂查询MySQL返回结果，包括结果集的聚合，对比，排序，去重，子查询语句下发等。

PS: dble中仅此线程池不限线程数量

## dbLe-Hash And ConsistentHashing And Jumpstringhash

### Questions

- dble中的hash和一致性hash是否相同
- 为什么没有一致性hash

### Conclusions

- dble中的hash并非一致性hash
- 从一致性hash的性能和均衡性来看，已被跳增一致性hash取代

### Instructions

#### dbLe-hash

- 配置如下：
- sharding.xml

```
<function name="hashLong" class="hash">
<property name="partitionCount">1,2</property>
<property name="partitionLength">10,20</property>
</function>
```

- 或者

```
<function name="hashLong" class="hash">
<property name="partitionCount">4</property>
<property name="partitionLength">10</property>
</function>
```

**注意：** partitionCount：指定分区的区间数  
partitionLength：指定各区间的长度  
模值 (M) 为最后一个区间的末尾值 (C1L1 + ... + CnLn)

对于dbLe-hash来说，该算法可以均匀地将数据分布到各个节点。

例如：

Count=2,Length=2 -> [0,2)[2,4) -> 模值=4

key分别取值：1,2,3,4,5,6,7,8

数据分布情况如下：

node1	node2
1,4,5,8	2,3,6,7

当增加一个节点，使Count=3,Length=2 -> [0,2)[2,4)[4,6) -> 模值=6 key分别取值：1,2,3,4,5,6,7,8

数据分布情况如下：

node1	node2	node3
1,6,7	2,3,8	4,5

从这个例子中我们可以看出：当节点数增加时，大多数旧的数据都需要重新分布，而重新分布的成本就是需要在count数发生变化的时候，进行数据迁移，大多数的数据都需要重新移动。

### summary

因此，对于这种算法，当node数发生变化（增加、移除）后，大多数旧的数据都需要重新分布，并进行数据迁移。

### Consistent Hashing

一致性哈希，将整个哈希值空间组织成一个虚拟的圆环，整个空间按顺时针方向组织。例如我们有NodeA、Node B、Node C、Node D四个节点，有数据A、数据B、数据C、数据D四个数据对象，根据一致性哈希算法，数据A会被分布到Node A上，B被分布到Node B上，C被分布到Node C上，D被分布到Node D上：

```
数据A ——> NodeA
数据B ——> NodeB
数据C ——> NodeC
数据D ——> NodeD
```

现假设Node C不幸宕机，此时数据A、B、D不会受到影响，只有数据C被重分布到Node D。如果在系统中增加一台服务器Node X：

```

数据A ——> NodeA
数据B ——> NodeB
新增 ——> NodeX
数据C ——> NodeC
数据D ——> NodeD

```

数据A、B、D不受影响，只有数据C需要重分布到新的Node X。也就是说旧数据之间不会发生数据的变动。

虽然一致性Hash算法解决了节点变化导致的数据迁移问题，但是数据项分布的均匀性不够好。一致性哈希算法分布不均匀的原因是因为：将node进行哈希后，这些值并没有均匀地落在环上，因此，这些节点所管辖的范围（每个节点实际占据环上的区间大小不）并不均匀，最终导致了数据分布的不均匀。

因此，为使得每个节点在环上所“管辖”更加均匀，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个服务节点，称为虚拟节点。

例如：

我们有NodeA、Node B；可以为每台服务器计算三个虚拟节点，于是可以分别计算“Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，形成六个虚拟节点。

同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上，“Node B#1”、“Node B#2”、“Node B#3”三个虚拟节点的数据均定位到Node B上，用来解决分布不均的问题。

```

Node A#1 ——> NodeA
Node A#2 ——> NodeA
Node A#3 ——> NodeA
Node B#1 ——> NodeB
Node B#2 ——> NodeB
Node B#3 ——> NodeB

```

因此，通过增加虚节点的方法，使得每个节点在环上所“管辖”更加均匀。这样就既保证了在节点变化时，尽可能小的影响数据分布的变化，而同时又保证了数据分布的均匀。也就是靠增加“节点数量”加强管辖区间的均匀。

## summary

对于一致性hash算法来说，当node数发生变化（增加、移除）后，旧的数据之间没有发生变动，只是需要将部分旧数据重新分布到新的节点。

## Jumpstringhash

- 配置如下：
- sharding.xml

```

<function name="jumphash" class="jumpStringHash">
 <property name="partitionCount">2</property>
 <property name="hashSlice">0:2</property>
</function>

```

**注意：** partitionCount：分片数量 hashSlice：分片截取长度

该算法来自于Google的一篇文章A Fast, Minimal Memory, Consistent Hash Algorithm，核心思想是通过概率分布的方法将一个hash值在每个节点分布的概率变成 $1/n$ ，并且可以通过更简便的方法可以计算得出，并且分布也更加均匀。

设计目标是把对象均匀地分布在所有节点中（平衡性）；当节点数量变化时，只需要把一些对象从旧节点移动到新节点，不需要做其它移动（单调性）。

根据论文原理，可以这样说明：

- 假设我们有四个节点，比作为四个桶，分别是：0,1,2,3。
- 数据落入第一个桶(0)的概率是 $1/n$ ；落入第二个桶(1)的概率是 $1/(n+1)$ ，也就是 $1/2$ ；落入第三个桶(2)的概率是 $1/(n+1)$ ，也就是 $1/3$ ；落入第四个桶(3)的概率是 $1/(n+1)$ ，也就是 $1/4$ 。
- 由此，一般规律是：数据落入每个桶中的概率，有占比  $n/(n+1)$  的结果保持不变，而有  $1/(n+1)$  跳变为  $n+1$ 。
- 而每个数据都是落入至index（max）中，比如数据a有概率分布至0,1,2,3四个桶中，那么最终会落入至3内。

当增加一个桶时：由0,1,2,3变为0,1,2,3,4，需要重新分布的数据仅是每个桶内有概率分布到新桶内的数据，旧数据之间不会发生变化。

## summary

由于Jumpstringhash采用的是概率分布的结果，因此计算相对于一致性hash较简单。

- Jumpstringhash相比于一致性hash算法来说，占用内存更小，计算更快，数据分布更均匀；
- Jumpstringhash和一致性hash算法，对于节点变动的情况下，都是将部分旧数据重新分布到新节点上，旧数据之间不会发生变动；
- Jumpstringhash和一致性hash算法相比于dble-hash来说，节点变动的情况下，旧数据之间不会发生变动；
- dble-hash相比于Jumpstringhash和一致性hash算法来说，最大的优势在于计算方便，可以人为的快速计算出结果。



- [ToBeContinued2](#)

待更新