

中国第一开源社区出品

第一版



Mycat

权威指南

Mycat

开源项目组 著



目录

入门篇	12
MYCAT 开源宣言	12
第 1 章 概述	14
1.1 数据库切分概述	14
1.2 垂直切分	17
1.3 水平切分	18
第 2 章 MYCAT 前世今生	21
2.1 序章	21
2.2 Mycat 闪耀登场	28
2.3 Mycat 概述	31
第 3 章 MYCAT 中的概念	35
3.1 数据库中间件	35
3.2 逻辑库(schema)	35
3.3 逻辑表 (table)	36
3.4 分片节点(dataNode)	37
3.5 节点主机(dataHost)	37
3.6 分片规则(rule)	37
3.7 全局序列号(sequence)	38
3.8 多租户	38
第 4 章 快速入门	39
4.1 10 分钟入门	39
4.2 快速镜像方式体验 MyCAT	41
4.3 服务安装与配置	41
4.4 服务启动与启动设置	44
4.5 基于 zk 的启动	45
4.6 demo 使用	52
第 5 章 日志分析	53

5.1 warpper 日志:	54
5.2 mycat 日志.....	55
5.3 debug 模式下分析 sql 执行。	58
5.4 异常日志.....	62
第 6 章 MYCAT 防火墙配置	64
第 7 章 MYCAT 的配置	69
7.1 搞定 schema.xml.....	69
7.2 schema 标签.....	69
7.3 table 标签.....	72
7.4 childTable 标签.....	74
7.5 dataNode 标签.....	75
7.6 dataHost 标签.....	76
7.7 heartbeat 标签.....	78
7.8 server.xml.....	80
7.9 system 标签.....	82
7.10 rule.xml.....	94
7.11 tableRule 标签.....	94
7.12 function 标签.....	94
第 8 章 MYCAT 的分片 JOIN.....	96
8.1 join 概述.....	96
8.2 全局表.....	98
8.3 ER Join.....	99
8.4 Share join.....	100
8.5 catlet (人工智能)	103
8.6 Spark/Storm 对 join 扩展.....	105
第 9 章 全局序列号	106
9.1 全局序列号介绍.....	106
9.2 本地文件方式.....	106
9.3 数据库方式.....	106

9.4 本地时间戳方式.....	109
9.5 分布式 ZK ID 生成器.....	110
9.6 Zk 递增方式.....	110
9.7 其他方式.....	111
9.8 自增长主键.....	111
第 10 章 MYCAT 分片规则.....	114
10.1 分片规则概述.....	114
10.2 Mycat 全局表.....	114
10.3 ER 分片表.....	114
10.4 多对多关联.....	115
10.5 Mycat 常用的分片规则.....	116
10.6 权限控制.....	130
10.7 多租户支持.....	130
第 11 章 常见问题与解决方案.....	131
11.1 Mycat 目前有哪些功能与特性?	131
11.2 Mycat 除了 Mysql 还支持哪些数据库?	132
11.3 Mycat 目前有生产案例了么?	132
11.4 Mycat 稳定性与 Cobar 如何?	132
11.5 Mycat 支持集群么?	132
11.6 Mycat 多主切换需要人工处理么?	132
11.7 Mycat 目前有多少人开发?	132
11.8 Mycat 目前有哪些项目?	132
11.9 Mycat 最新的稳定版本是哪个到哪里下载?	133
11.10 Mycat 如何配置字符集?.....	133
11.11 Mycat 后台管理监控如何使用?	133
11.12 Mycat 主键插入后应用如何获取?	133
11.13 Mycat 如何启动与加入服务?	133
11.14 Mycat 运行 sql 时经常阻塞或卡死是什么原因?	134
11.15 Mycat 中, 旧系统数据如何迁移到 Mycat 中?	134

11.16 Mycat 如何对旧分片数据迁移或扩容, 支持自动扩容么?	134
11.17 Mycat 支持批量插入吗?	134
11.18 Mycat 支持多表 Join 吗?	134
11.19 Mycat 启动报主机不存在的问题?	134
11.20 Mycat 连接会报无效数据源 (Invalid datasource) ?	134
11.21 Mycat 使用中如何提需求或 bug?	135
11.22 Mycat 如何建表与创建存储过程?	135
11.23 Mycat 目前有多少人维护?	135
11.24 Mycat 支持的或者不支持的语句有哪些?	135
11.25 Mycat/JDBC 连接报 PacketTooBigException 异常.....	135
11.26 Mycat 中文乱码的问题.....	136
11.27 Mycat 无法登陆 Access denied.....	136
11.28 Mycat 的分片数据插入报异常 IndexOutOfBoundsException.....	136
11.29 Mycat ER 分片子表数据插入报错.....	137
11.30 Mycat 最大内存无法调整至 4G 以上.....	137
11.31 Mycat 使用过程中报错怎么办	137
第 12 章 MYCAT 性能测试指南	137
高级进阶篇	143
第 1 章读写分离	143
1.1 MySQL 主从复制的几种方案.....	143
1.2 MySQL 主从复制的几个问题.....	146
1.3 Mycat 支持的读写分离.....	148
第 2 章 高可用与集群	151
2.1 MySQL 高可用的几种方案.....	151
首先我们看看 MySQL 高可用的几种方案:	151
2.2 Mycat 高可用方案.....	155
2.3 Galary Cluster 配置.....	158
第 3 章 事务支持	159
3.1 Mycat 里的数据库事务.....	159

3.2 XA 事务原理.....	159
3.3 XA 事务的问题和 MySQL 的局限.....	162
3.4 XA 事务使用指南.....	163
3.5 保证 repeatable read.....	164
第 4 章 MYCAT SQL 拦截机制.....	165
第 5 章 MYCAT 注解	167
5.1 注解原理.....	167
5.2 注解使用示例.....	169
第 6 章 MYCAT 支持的 CATLET 实现.....	172
第 7 章 JDBC 多数据库支持.....	172
7.1 JDBC 概述.....	172
7.2 JDBC 体系结构.....	173
7.3 JDBC API.....	174
7.4 JDBC 4.0.....	175
7.5 Mycat 对 JDBC 的支持.....	176
7.6 NoSQL 支持(MongoDB).....	176
7.7 MongoDB.....	177
7.7.1 配置支持 Mongoddb.....	178
7.8 Oracle.....	185
7.8.1 配置支持 Oracle.....	185
7.8.2 三层嵌套分页.....	186
7.8.3 rownum 控制最大条数.....	187
7.9 SQL Server.....	187
7.9.1 配置支持 SQL Server.....	187
7.9.2 row_number 分页.....	188
7.9.3 row_number 与 top 结合分页.....	188
7.9.4 top 限制最大条数.....	188
7.10 DB2.....	188
7.10.1 row_number 分页.....	189

7.10.2 fetch first rows only 控制最大条数.....	189
7.11 Spark SQL/Hive.....	189
7.11.1 配置 Mycat.....	189
7.11.2 配置 Hive 安装模式.....	190
7.11.3 配置 Spark SQL	192
7.12 PostgreSQL.....	192
7.12.1 limit 分页自动转换.....	192
第 8 章 管理命令与监控	193
第 9 章 压缩协议支持	211
9.1 压缩协议支持.....	211
9.2 配置说明.....	211
9.3 压缩性能测试.....	211
9.4 mysql 压缩协议.....	211
第 10 章 MYCAT-WEB	213
第 11 章 MYCAT 对存储过程的支持	235
第 12 章 MYCAT 对 ZOOKEEPER 的支持	235
生产实践篇.....	237
第 1 章 生产实践案例-MYCAT 读写分离案例.....	237
第 2 章分表分库案例	240
2.1 SAAS 多租户案例.....	240
2.2 每天 2 亿数据的实时查询案例.....	241
2.3 物联网 26 亿数据的案例.....	242
2.4 大型分布式零售系统案例.....	243
第 3 章 生产环境部署	246
3.1 单节点 mycat 部署.....	246
3.2 mycat 的高可用与负载均衡.....	246
第 4 章 MYCAT 最佳实践	262
第 5 章 MYCAT 实施指南	263

5.1 Mycat 项目实施步骤.....	263
5.2 分表分库原则.....	264
5.3 后端存储的选择.....	267
5.4 数据拆分原则.....	268
5.5 DataNode 的分布问题.....	269
5.6 Mycat 目前存在的限制.....	269
第 6 章 数据迁移与扩容实践	271
6.1 离线扩容缩容.....	271
6.2 案例一：使用一致性 Hash 进行分片.....	274
6.3 案例二：使用范围分片.....	282
6.4 数据迁移的注意点.....	284
6.5 load data 批量导入.....	285
6.6 使用 mysqldump 进行数据迁移.....	287
6.7 迁移一个表中的部分数据.....	288
6.8 数据自动迁移方案设计.....	288
6.9 数据自动迁移使用指南.....	290
6.9.1 约束条件	290
6.9.2 准备	290
rule.xml 配置	290
在 myid.properties 中配置集群相关信息.....	291
schema.xml 配置	292
table 节点配置.....	292
datahost 节点的 slaveIDs 属性配置	292
配置新的 dataNode	292
使用 zk 来管理 mycat 集群.....	293
数据库表的要求	294
6.9.3 执行迁移命令.....	294
6.9.4 查看任务进度和结果.....	295
6.9.5 异常处理	298

6.9.6 数据迁移测试.....	303
第 7 章 版本选择与升级指南	305
7.1 版本选择.....	305
7.2 mycat1.2 中的功能	306
7.3 mycat1.3 中的功能	306
7.4 mycat1.4 中的功能	307
7.5 mycat1.5 中的功能	308
7.6 mycat1.6 中的功能	311
7.7 小结.....	313
7.8 1.6 升级指南.....	313
第 8 章 性能调优	314
8.1 主机调优.....	314
8.2 JVM 调优.....	315
8.3 MyCAT 调优.....	319
8.4 MySQL 通用调优.....	320
开发篇.....	324
第 1 章 加入 MYCAT	324
1.1 如何加入 Mycat.....	324
1.2 如何获取源码.....	324
第 2 章 MYCAT 开发基础	325
2.1 代码调试入口.....	325
2.2 中间件开发技能.....	325
第 3 章 MYCAT 架构分析	326
3.1 MyCAT 和 TDDL、Amoeba、Cobar 的架构比较.....	326
3.2 框架比较.....	326
3.3 点评.....	328
3.4 其它资料.....	329
第 4 章 MYCAT 线程模型分析	330

4.1 MyCAT 线程模型.....	330
4.2 Mycat 线程介绍.....	330
4.3 Cobar 线程介绍.....	334
4.4 Cobar 为什么那么多个线程池?	336
4.5 MyCAT 与 Cobar 的比较.....	339
第 5 章 MYCAT 的连接池模型.....	340
第 6 章 MYCAT 的网络通信框架	345
6.1 先从一个测试说起.....	345
6.2 MyCAT 网络框架.....	347
6.3 与 Cobar 原有 NIO 细节比较.....	366
6.4 MyCAT 的 AIO 实现.....	371
第 7 章 MYCAT 的路由与分发流程.....	381
7.1 路由的作用.....	381
7.2 路由解析器.....	381
7.3 druid 路由解析的两种方式.....	384
7.4 路由计算.....	388
7.5 路由计算的核心要素.....	393
7.6 单个表的路由计算.....	393
7.7 多个表的路由计算.....	394
7.8 全局表的路由计算.....	395
7.9 or 语句的路由计算.....	396
7.10 系统语句的路由计算	402
7.11 相关类图和序列图.....	402
7.12 路由解析过程中的一些控制变量.....	405
第 8 章 MYCAT 的 JDBC 后端框架.....	406
8.1 JDBC 方式访问后端数据库.....	406
8.2 JDBC 相关类图.....	406
8.3 JDBCDataSource	407
8.4 JDBCConnection.....	410

8.5 JDBCHeartbeat.....	416
第 9 章 MYCAT 的事务管理机制	418
9.1 Mycat 事务源码分析.....	419
第 10 章 MYCAT 的分页和跨库 JOIN	422
10.1 多数据库支持的分页机制.....	422
10.2 ShareJoin 代码分析.....	437
第 11 章 MYCAT 缓存.....	448
11.1 缓存介绍及代码分析	448
11.2 SQLRouteCache.....	451
11.3 TableID2DataNodeCache	453
11.4 ER_SQL2PARENTID.....	458
第 12 章 MYCAT 的分片规则设计	462
12.1 分片规则设计架构.....	463
12.2 分片规则自定义实现	466
第 13 章 MYCAT LOAD DATA 源码.....	471
13.1 load data 代码分析	471
13.2 mysql 压缩协议代码分析.....	479
第 14 章 MYCAT 外传-群英会	486
14.1 我不做大哥很多年.....	486
14.2 冰风影.....	486
14.3 从零开始.....	487
14.4 黑白咖啡.....	488
14.5 石头狮子.....	488
14.6 Rainbow.....	489
14.7 Mycat 铁杆粉丝.....	490
14.8 兵临城下.....	490
14.9 我是谁.....	491
14.10 当太极遇到 AK47.....	491
14.11 传说中的 Mycat 大美女	493

14.12 Mycat 至尊酱油师.....	493
14.13 白衣公子.....	494
14.14 他入错了行.....	495
14.15 烟花易冷-奎.....	496
14.16 海王星.....	496
14.17 太极鸟人.....	497
14.18 成都-顽石神.....	498
14.19 杭州-白.....	498
14.20 allnet-深海.....	498
14.21 明明 Ben.....	498
14.22 上海-袁文华.....	499
14.23 杭州-yuanfang.....	499
14.24 胡雅辉.....	499
14.25 KK.....	500
14.26 CrazyPig.....	500
14.27 传说的学霸.....	500
14.28 毛茸茸的逻辑.....	501
14.30 深圳-Java-HelloWorld.....	501

入门篇

MYCAT 开源宣言



2013 年阿里的 Cobar 在社区使用过程中发现存在一些比较严重的问题，及其使用限制，经过 Mycat 发人第一次改良，第一代改良版——Mycat 诞生。Mycat 开源以后，一些 Cobar 的用户参与了 Mycat 的开发，最终 Mycat 发展成为一个由众多软件公司的实力派架构师和资深开发人员维护的社区型开源软件。

2014 年 Mycat 首次在上海的《中华架构师》大会上对外宣讲，更多的人参与进来，随后越来越多的项目采用了 Mycat。

2015 年 5 月，由核心参与者们一起编写的第一本官方权威指南《Mycat 权威指南》电子版发布，累计超过 500 本，成为开源项目中的首创。

2015 年 10 月为止，Mycat 项目总共有 16 个 Committer。

截至 2015 年 11 月，超过 300 个项目采用 Mycat，涵盖银行、电信、电子商务、物流、移动应用、O2O 的众多领域和公司。

截至 2015 年 12 月，超过 4000 名用户加群或研究讨论或测试或使用 Mycat。

Mycat 是基于开源 cobar 演变而来，我们对 cobar 的代码进行了彻底的重构，使用 NIO 重构了网络模块，并且优化了 Buffer 内核，增强了聚合，Join 等基本特性，同时兼容绝大多数数据库成为通用的数据库中间件。1.4 版本以后完全的脱离基本 cobar 内核，结合 Mycat 集群管理、自动扩容、智能优化，成为高性能的中间件。我们致力于开发高性能数据库中间而努力。永不收费，永不闭源，持续推动开源社区的发展。

Mycat 吸引和聚集了一大批业内大数据和云计算方面的资深工程师，Mycat 的发展壮大基于开源社区志愿者的持续努力，感谢社区志愿者的努力让 Mycat 更加强大，同时我们也欢迎社区更多的志愿者，特别是公司能够参与进来，参与 Mycat 的开发，一起推动社区的发展，为社区提供更好的开源中间件。

Mycat 还不够强大，Mycat 还有很多不足，欢迎社区志愿者的持续优化改进。

Mycat 捐赠地址: <http://www.mycat.io/donate.html>

Mycat 官方网站: <http://www.mycat.io/>

Mycat 源码: <https://github.com/MyCATapache/Mycat-Server>

Mycat 下载地址: <https://github.com/MyCATapache/Mycat-download>

Mycat 开源社区

2018.07.25

第 1 章 概述

1.1 数据库切分概述

1.1.1 OLTP 和 OLAP

在互联网时代，海量数据的存储与访问成为系统设计与使用的瓶颈问题，对于海量数据处理，按照使用场景，主要分为两种类型：联机事务处理（OLTP）和联机分析处理（OLAP）。

联机事务处理（OLTP）也称为面向交易的处理系统，其基本特征是原始数据可以立即传送到计算中心进行处理，并在很短的时间内给出处理结果。

联机分析处理（OLAP）是指通过多维的方式对数据进行分析、查询和报表，可以同数据挖掘工具、统计分析工具配合使用，增强决策分析功能。

对于两者的主要区别可以用下表来说明：

	OLTP	OLAP
系统功能	日常交易处理	统计、分析、报表
DB 设计	面向实时交易类应用	面向统计分析类应用
数据处理	当前的, 最新的细节的, 二维的分立的	历史的, 聚集的, 多维的集成的, 统一的
实时性	实时读写要求高	实时读写要求低
事务	强一致性	弱事务
分析要求	低、简单	高、复杂

1.1.2 关系型数据库和 NoSQL 数据库

针对上面两类系统有多种技术实现方案，存储部分的数据库主要分为两大类：关系型数据库与 NoSQL 数据库。

关系型数据库，是建立在关系模型基础上的数据库，其借助于集合代数等数学概念和方法来处理数据库中的数据。主流的 oracle、DB2、MS SQL Server 和 mysql 都属于这类传统数据库。

NoSQL 数据库，全称为 Not Only SQL，意思就是适用关系型数据库的时候就使用关系型数据库，不适用的时候也没有必要非使用关系型数据库不可，可以考虑使用更加合适的数据存储。主要分为临时性键值存储（memcached、Redis）、永久性键值存储（ROMA、Redis）、面向文档的数据库（MongoDB、CouchDB）、面向列的数据库（Cassandra、HBase），每种 NoSQL 都有其特有的使用场景及优点。

Oracle, mysql 等传统的关系数据库非常成熟并且已大规模商用，为什么还要用 NoSQL 数据库呢？主要是由于随着互联网发展，数据量越来越大，对性能要求越来越高，传统数据库存在着先天性的缺陷，即单机（单库）性能瓶颈，并且扩展困难。这样既有单机单库瓶颈，却又扩展困难，自然无法满足日益增长的海量数据存储及其性能要求，所以才会出现了各种不同的 NoSQL 产品，NoSQL 根本性的优势在于在云计算时代，简单、易于大规模分布式扩展，并且读写性能非常高。

下面分析下两者的特点，及优缺点：

	关系型数据库	NoSQL 数据库
特点	- 数据关系模型基于关系模型，结构化存储，完整性约束 - 基于二维表及其之间的联系，需要连接、并、交、差、除等数据操作 - 采用结构化的查询语言（SQL）做数据读写 - 操作需要数据的一致性，需要事务甚至是强一致性	- 非结构化的存储 - 基于多维关系模型 - 具有特有的使用场景
优点	- 保持数据的一致性（事务处理） - 可以进行 join 等复杂查询 - 通用化，技术成熟	- 高并发，大数据下读写能力较强 - 基本支持分布式，易于扩展，可伸缩 - 简单，弱结构化存储
缺点	- 数据读写必须经过 sql 解析，大量数据、高并发下读写性能不足 - 对数据做读写，或修改数据结构时需要加锁，影响并发操作 - 无法适应非结构化存储 - 扩展困难 - 昂贵、复杂	- join 等复杂操作能力较弱 - 事务支持较弱 - 通用性差 - 无完整约束复杂业务场景支持较差

虽然在云计算时代，传统数据库存在着先天性的弊端，但是 NoSQL 数据库又无法将其替代，NoSQL 只能作为传统数据的补充而不能将其替代，所以规避传统数据库的缺点是当前大数据时代必须要解决的问题。如果传统数据易于扩展，可切分，就可以避免单机（单库）的性能缺陷，但是由于目前开源或者商用的传统数据库基本不支持大规模自动扩展，所以需要借助第三方来做处理，那就是本书要讲的数据切分，下面就来分析一下如何进行数据切分。

1.1.3 何为数据切分？

简单来说，就是指通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。

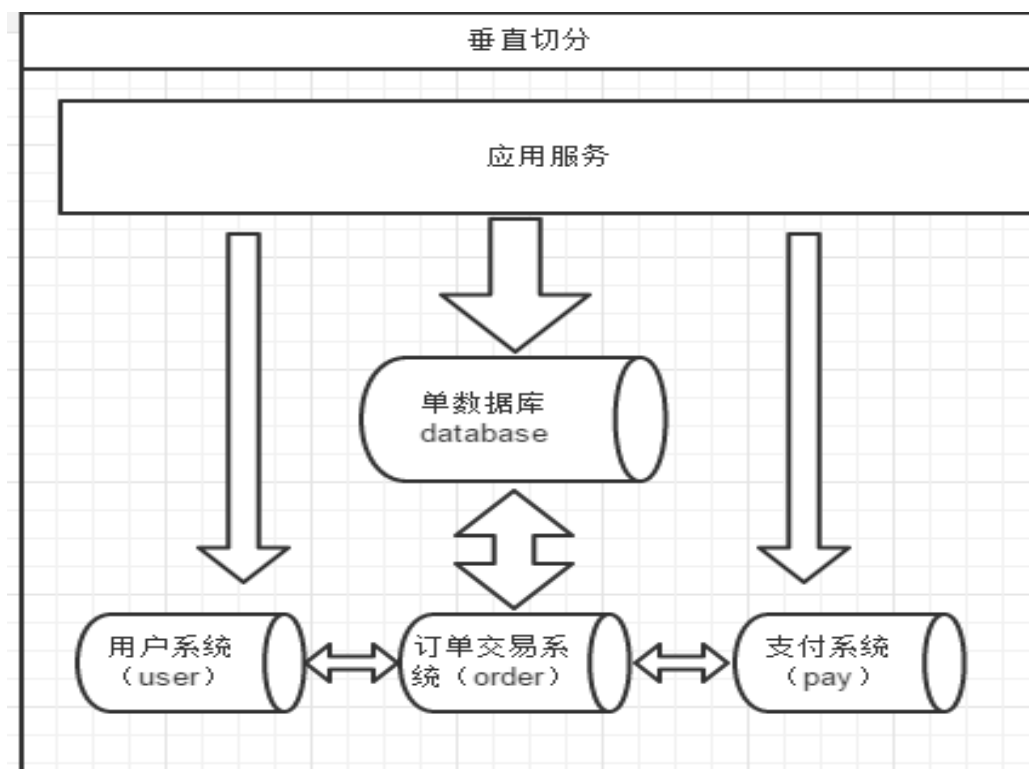
数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。一种是按照不同的表（或者 Schema）来切分到不同的数据库（主机）之上，这种切可以称之为数据的垂直（纵向）切分；另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上面，这种切分称之为数据的水平（横向）切分。

垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表拆分到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平切分于垂直切分相比，相对来说稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就较根据表名来拆分更为复杂，后期的数据维护也会更为复杂一些。

1.2 垂直切分

一个数据库由很多表的构成，每个表对应着不同的业务，垂直切分是指按照业务将表进行分类，分布到不同的数据库上面，这样也就将数据或者说压力分担到不同的库上面，如下图：



系统被切分成了，用户，订单交易，支付几个模块。

一个架构设计较好的应用系统，其总体功能肯定是由很多个功能模块所组成的，而每一个功能模块所需要的数据对应到数据库中就是一个或者多个表。而在架构设计中，各个功能模块相互之间的交互点越统一越少，系统的耦合度就越低，系统各个模块的维护性以及扩展性也就越好。这样的系统，实现数据的垂直切分也就越容易。

但是往往系统之有些表难以做到完全的独立，存在这扩库 join 的情况，对于这类的表，就需要去做平衡，是数据库让步业务，共用一个数据源，还是分成多个库，业务之间通过接口来做调用。在系统初期，数据量比较少，或者资源有限的情况下，会选择共用数据源，但是当数据发展到了一定的规模，负载很大的情况，就需要必须去做分割。

一般来讲业务存在着复杂 join 的场景是难以切分的，往往业务独立的易于切分。如何切分，切分到何种程度是考验技术架构的一个难题。

下面来分析下垂直切分的优缺点：

优点：

- 拆分后业务清晰，拆分规则明确；

- 系统之间整合或扩展容易；
- 数据维护简单。

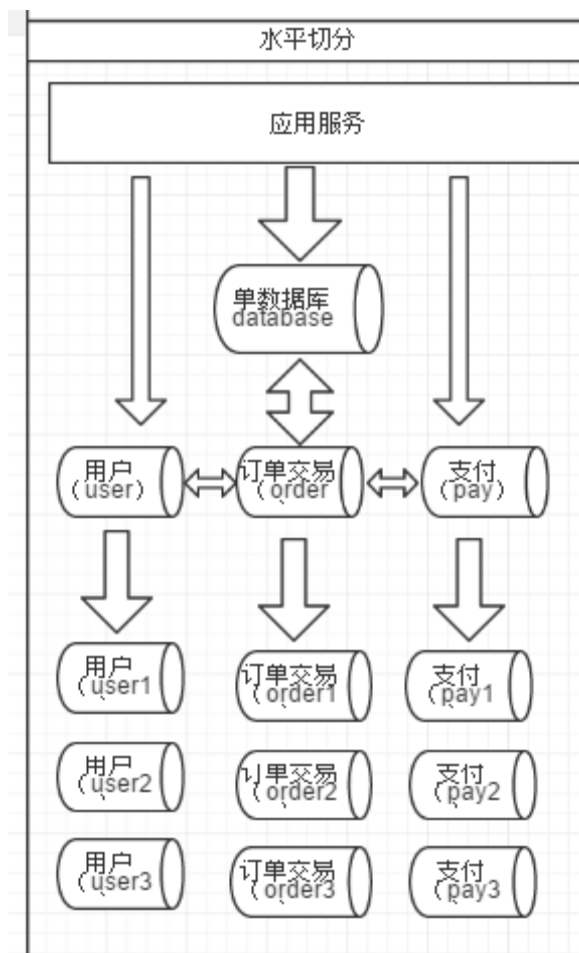
缺点：

- 部分业务表无法 join，只能通过接口方式解决，提高了系统复杂度；
- 受每种业务不同的限制存在单库性能瓶颈，不易数据扩展跟性能提高；
- 事务处理复杂。

由于垂直切分是按照业务的分类将表分散到不同的库，所以有些业务表会过于庞大，存在单库读写与存储瓶颈，所以需要水平拆分来做解决。

1.3 水平切分

相对于垂直拆分，水平拆分不是将表做分类，而是按照某个字段的某种规则来分散到多个库之中，每个表中包含一部分数据。简单来说，我们可以将数据的水平切分理解为是按照数据行的切分，就是将表中的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中，如图：

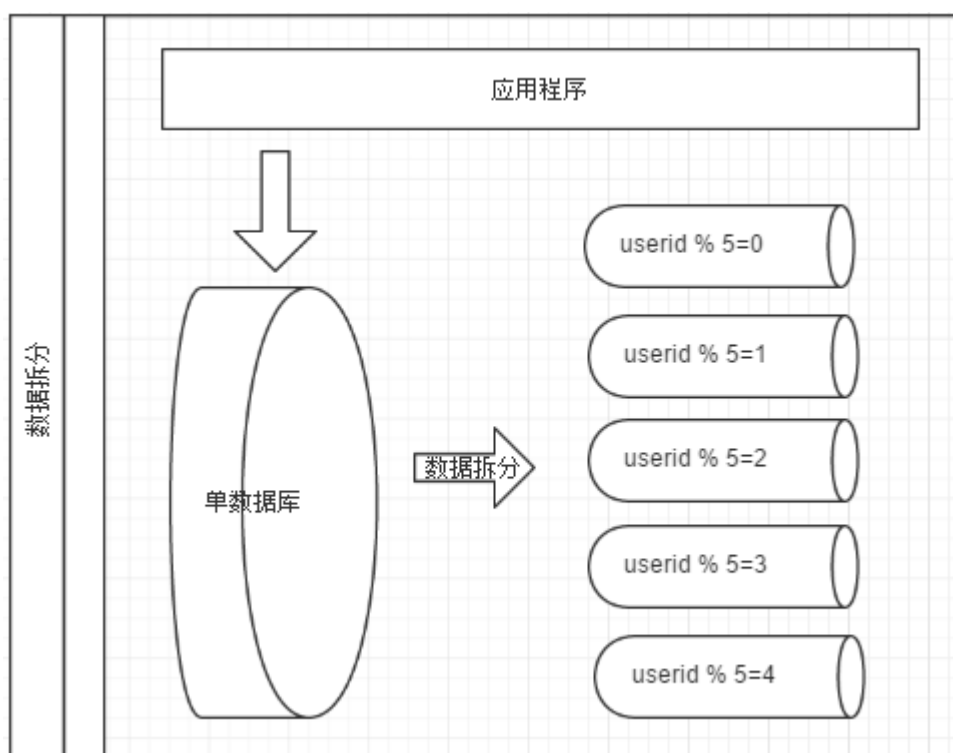


拆分数据就需要定义分片规则。关系型数据库是行列的二维模型，拆分的第一原则是找到拆分维度。比如：从会员的角度来分析，商户订单交易系统查询会员某天某月某个订单，那么就需要按照会员结合日期来拆分，不同的数据按照会员 ID 做分组，这样所有的数据查询 join 都会在单库内解决；如果从商户的角度来讲，要查询某个商家某天所有的订单数，就需要按照商户 ID 做拆分；但是如果系统既想按会员拆分，又想按商家数据，则会有-定的困难。如何找到合适的分片规则需要综合考虑衡量。

几种典型的分片规则包括：

- 按照用户 ID 求模，将数据分散到不同的数据库，具有相同数据用户的数据都被分散到一个库中；
- 按照日期，将不同月甚至日的数据分散到不同的库中；
- 按照某个特定的字段求模，或者根据特定范围段分散到不同的库中。

如图，切分原则都是根据业务找到适合的切分规则分散到不同的库，下面用用户 ID 求模举例：



既然数据做了拆分有优点也就优缺点。

优点：

- 拆分规则抽象好，join 操作基本可以数据库做；
- 不存在单库大数据，高并发的性能瓶颈；
- 应用端改造较少；
- 提高了系统的稳定性跟负载能力。

缺点：

- 拆分规则难以抽象；
- 分片事务一致性难以解决；
- 数据多次扩展难度跟维护量极大；
- 跨库 join 性能较差。

前面讲了垂直切分跟水平切分的不同跟优缺点，会发现每种切分方式都有缺点，但共同的特点缺点有：

- 引入分布式事务的问题；
- 跨节点 Join 的问题；
- 跨节点合并排序分页问题；
- 多数据源管理问题。

针对数据源管理，目前主要有两种思路：

A. 客户端模式，在每个应用程序模块中配置管理自己需要的一个（或者多个）数据源，直接访问各个数据库，在模块内完成数据的整合；

B. 通过中间代理层来统一管理所有的数据源，后端数据库集群对前端应用程序透明；

可能 90%以上的人在面对上面这两种解决思路的时候都会倾向于选择第二种，尤其是系统不断变得庞大复杂的时候。确实，这是一个非常正确的选择，虽然短期内需要付出的成本可能会相对更大一些，但是对整个系统的扩展性来说，是非常有帮助的。

Mycat 通过数据切分解决传统数据库的缺陷，又有了 NoSQL 易于扩展的优点。通过中间代理层规避了多数数据源的处理问题，对应用完全透明，同时对数据切分后存在的问题，也做了解决方案。下面章节就分析，mycat 的由来及如何进行数据切分问题。

由于数据切分后数据 Join 的难度在此也分享一下数据切分的经验：

第一原则：能不切分尽量不要切分。

第二原则：如果要切分一定要选择合适的切分规则，提前规划好。

第三原则：数据切分尽量通过数据冗余或表分组（Table Group）来降低跨库 Join 的可能。

第四原则：由于数据库中间件对数据 Join 实现的优劣难以把握，而且实现高性能难度极大，业务读取尽量少使用多表 Join。

什么是 mycat，mycat 从哪里来，又是如何解决这些问题的，下一章让我们来作分析。

第 2 章 Mycat 前世今生

2.1 序章

如果我有一个 32 核心的服务器，我就可以实现 1 个亿的数据分片，我有 32 核心的服务器么？没有，所以我至今无法实现 1 个亿的数据分片。——Mycat' s Plan

上面这句话是 Mycat 1.0 快要完成时候的一段感言，而当发展到 Mycat 1.3 的时候，我们又有了一个新的 Plan：

如果我们有 10 台物理机，我们就可以实现 1000 亿的数据分片，我们有 10 台物理机么？没有，所以，Mycat 至今没有机会验证 1000 亿大数据的支撑能力——Mycat' s Plan 2.0

“每一个成功的男人背后都有一个女人”。自然 Mycat 也逃脱不了这个法则。Mycat 背后是阿里曾经开源的知名产品——Cobar。Cobar 的核心功能和优势是 MySQL 数据库分片，此产品曾经广为流传，据说最早的发起者对 Mysql 很精通，后来从阿里跳槽了，阿里随后开源的 Cobar，并维持到 2013 年年初，然后，就没有然后了。

Cobar 的思路和实现路径的确不错。基于 Java 开发的，实现了 MySQL 公开的二进制传输协议，巧妙地将自己伪装成一个 MySQL Server，目前市面上绝大多数 MySQL 客户端工具和应用都能兼容。比自己实现一个新的数据库协议要明智的多，因为生态环境在哪里摆着。

Cobar 使用起来也非常方便。由于是基于 Java 语言开发的，下载下来解压，安装 JDK，然后配置几个不是很复杂的配置文件，猛击鼠标，就能启动 Cobar。因此这个开源产品赢得了很多 Java 粉丝以及 PHP 用户的追捧。当然，笨人(Leader us)也跟着进入，并且在某个大型云项目中——“苦海无边”的煎着熬，良久。

爱情就像是见鬼。只有撞见了，你才会明白爱情是怎么回事。TA 是如此神秘，欲语还羞。情窦初开的你又玩命将 TA 的优点放大，使自己成为一只迷途的羔羊。每个用过 Cobar 的人就像谈过一段一波三折、荡气回肠的爱情，令你肝肠寸断。就像围城：里面的人已经出不来了，还有更多的人拼命想挤进去。

仅以此文，献给哪些努力在 IT 界寻求未来的精英和小白们，还有更多被无视的，正准备转行的同仁，同在江湖混，不容易啊，面试时候就装装糊涂，放人家一马，说不定，以后又是一个 Made in China 的乔布斯啊。

如果我有一个 32 核心的服务器，我就可以实现 1 个亿的数据分片，我有 32 核心的服务器么？没有，所以我至今无法实现 1 个亿的数据分片。——Mycat 's Plan

2.1.1 曾经的 TA

曾经的 TA，长发飘飘，肤若凝脂，国色天香，长袖善舞，所以，一笑倾城。

那已成传说，一如您年少时的坚持：“书中自有黄金屋...”

Cobar 曾是多少 IT 骚年心中的那个 TA，有关 Cobar 的这段美好的描述(不能说是广告)俘虏了众多程序猿躁动纯真的心：

Cobar 是阿里巴巴研发的关系型数据的分布式处理系统，该产品成功替代了原先基于 Oracle 的数据存储方案，目前已经接管了 3000+个 MySQL 数据库的 schema，平均每天处理近 50 亿次的 SQL 执行请求。

50 亿有多大？99%的普通人类看到这个数字，已经不能呼吸。当然，我指的是**RMB**。99%的程序猿除了对工资比较敏感，其实对数字通常并不感冒。上面这个简单的数字描述，已立刻让我们程序型的大脑短路。恨不得立刻百度 Cobar，立刻 Download，立刻熬夜研究。做个简单的推算，50 亿次请求转换为每个 schema 每秒的数据访问请求即 TPS，于是我们得到一个让自己不能相信的数字：20TPS，每秒不到 20 个访问。

Cobar 最重要的特性是分库分表。Cobar 可以让你把一个 MySQL 的 Table 放到 10 个甚至 100 个位于不同物理机上的 MySQL 服务器上去存储，而在用户看来是一张表（逻辑表）。这样功能很有价值。比如：我们有 1 亿的订单，则可以划分为 10 个分片，存储到 2-10 个物理机上。每个 MySQL 服务器的压力减少，而系统的响应时间则不会增加。看上去很完美的功能，而且潜意识里，执行这句 SQL：

```
select count(*) from order
```

100%的人都会认为：会返回 1 条数据，但事实上，Cobar 会返回 N 条数据，N=分片个数。

接下来我们继续执行 SQL：

```
select count(*) from order order by order_date
```

你会发现奇怪的乱序现象，而且结果还随机，这是因为，Cobar 只是简单的把上述 SQL 发给了后端 N 个分片对应的 MySQL 服务器去执行，然后把结果集直接输出....

再继续看看，我们常用的 Limit 分页的结果...可以么？答案是：**不可以**。

这个问题可以在客户端程序里做些工作来解决。所以随后出现了 Cobar Client。据我所知，很多 Cobar 的使用者也都是自行开发了类似 Cobar Client 的工具来解决此类问题。从实际应用效果来说，一方面，客户端编程方式解决，困难度很高，Bug 率也居高不下；另一方面，对于 DBA 和运维来说，增加了困难度。

当你发现这个问题的严重性，再回头看看 Cobar 的官方文档，你怅然若失，四顾茫然。

接下来，本文将隐藏在 Cobar 代码中那些不为人知的秘密逐一披露，你洞悉了这些秘密，就会明白 Mycat 为什么会横空出世。

2.1.2 Cobar 的十一个秘密

2.1.1.1 第一个秘密：Cobra 会假死？

是的，很多人遇到这个问题。如何来验证这点呢？可以做个简单的小实验，假如你的分片表中配置有表 company,则打开 mysql 终端，执行下面的 SQL：

```
select sleep(500) from company;
```

此 SQL 会执行等待 500 秒，你再努力以最快的速度打开 N 个 mysql 终端，都执行相同的 SQL，确保 $N >$ 当前 Cobra 的执行线程数：

```
show @@threadpool
```

的所有 Processor1-E 的线程池的线程数量总和，然后你再执行任何简单的 SQL，或者试图新建立连接，都会无法响应，此时

```
show @@threadpool
```

里面看到 TASK_QUEUE_SIZE 已经在积压中。

不可能吧，据说 Cobra 是 NIO 的非阻塞的，怎么可能阻塞！别激动，去看看代码，Cobra 前端是 NIO 的，而后端跟 Mysql 的交互，是阻塞模式，其 NIO 代码只给出了框架，还未来得及实现。真相永远在代码里，所以，为了发现真相，还是转行去做码农吧！貌似码农也像之前的技术工人，越来越稀罕了。

2.1.1.2 第二个秘密：高可用的陷阱？

每一个秘密的背后，总是隐藏着更大的秘密。Cobra 假死的秘密背后，还隐藏着一个更为“强大”的秘密，那就是假死以后，Cobra 的频繁主从切换问题。我们看看 Cobra 的一个很好的优点——“高可用性”的实现机制，下图解释了 Cobra 如何实现高可用性：

分片节点 dn2_M1 配置了两个 dataSource，并且配置了心跳检测(heartbeat)语句，在这种配置下，每个 dataNode 会定期对当前正在使用的 dataSource 执行心跳检测，默认是第一个，频率是 10 秒钟一次，当心跳检测失败以后，会自动切换到第二个 dataSource 上进行读写，假如 Cobra 发生了假死，则在假死的 1 分钟内，Cobra 会自动切换到第二个节点上，因为假死的缘故，第二个节点的心跳检测也超时。于是，1 分钟内 Cobra 频繁来回切换，懂得 MySQL 主从复制机制的人都知道，在两个节点上都执行写操作意味着什么？——可能数据一致性被破坏，谁也不知道那个机器上的数据是最新的。

还有什么情况下，会导致心跳检测失败呢？这是一个不得不说的秘密：当后端数据库达到最大连接后，会对新建连接全部拒绝，此时，Cobar 的心跳检测所建立的新连接也会被拒绝，于是，心跳检测失败，于是，一切都悄悄的发生。

幸好，大多数同学都没有配置高可用性，或者还不了解此特性，因此，这个秘密，一直在安全的沉睡。

2.1.1.3 第三个秘密：看上去很美的自动切换

Cobar 很诱人的一个特性是高可用性，高可用性的原理是数据节点 DataNode 配置引用两个 DataSource，并做心跳检测，当第一个 DataSource 心跳检测失败后，Cobar 自动切换到第二个节点，当第二个节点失败以后，又自动切换回第一个节点，一切看起来很美，无人值守，几乎没有宕机时间。

在真实的生产环境中，我们通常会用至少两个 Cobar 实例组成负载均衡，前端用硬件或者 HAProxy 这样的负载均衡组件，防止单点故障，这样一来，即使某个 Cobar 实例死了，还有另外一台接手，某个 Mysql 节点死了，切换到备节点继续，至此，一切看起来依然很美，喝着咖啡，听着音乐，领导视察，你微笑着点头——No problem, Everything is OK!直到有一天，某个 Cobar 实例果然如你所愿的死了，不管是假死还是真死，你按照早已做好的应急方案，优雅的做了一个不是很艰难的决定——重启那个故障节点，然后继续喝着咖啡，听着音乐，轻松写好故障处理报告发给领导，然后又度过了美好的一天。

你忽然被深夜一个电话给惊醒，你来不及发火，因为你的直觉告诉你，这个问题很严重，大量的订单数据发生错误很可能是昨天重启 cobar 导致的数据库发生奇怪的问题。你努力排查了几个小时，终于发现，主备两个库都在同时写数据，主备同步失败，你根本不知道那个库是最新数据，紧急情况下，你做了一个很英明的决定，停止昨天故障的那个 cobar 实例，然后你花了 3 个通宵，解决了数据问题。

这个陷阱的代价太高，不知道有多少同学中枪过，反正我也是躺着中枪过了。若你还不清楚为何会产生这个陷阱，现在我来告诉你：

- Cobar 启动的时候，会用默认第一个 Datasource 进行数据读写操作；
- 当第一个 Datasource 心跳检测失败，会切换到第二个 Datasource；
- 若有两个以上的 Cobar 实例做集群，当发生节点切换以后，你若重启其中任何一台 Cobar，就完美调入陷阱；

那么，怎么避免这个陷阱？目前只有一个办法，节点切换以后，尽快找个合适的时间，全部集群都同时重启，避免隐患。为何是重启而不是用节点切换的命令去切换？想象一下 32 个分片的数据库，要多少次切换？

MyCAT 怎么解决这个问题的？很简单，节点切换以后，记录一个 properties 文件（conf 目录下），重启的时候，读取里面的节点 index，真正实现了无故障无隐患的高可用性。

2.1.1.4 第四个秘密：只实现了一半的 NIO

NIO 技术用作 JAVA 服务器编程的技术标准，已经是不容置疑的业界常规做法，若一个 Java 程序员，没听说过 NIO，都不好意思说自己是 Java 人。所以 Cobar 采用 NIO 技术并不意外，但意外的是，只用了一半。

Cobar 本质上是一个“数据库路由器”，客户端连接到 Cobar，发生 SQL 语句，Cobar 再将 SQL 语句通过后端与 MySQL 的通讯接口 Socket 发出去，然后将结果返回给客户端的 Socket 中。下面给出了 SQL 执行过程简要逻辑：

```
SQL->FrontConnection->Cobar->MySQLChanel->MySQL
```

FrontConnection 实现了 NIO 通讯，但 MySQLChanel 则是同步的 IO 通讯，原因很简单，指令比较复杂，NIO 实现有难度，容易有 BUG。后来最新版本 Cobar 尝试了将后端也 NIO 化，大概实现了 80% 的样子，但没有完成，也存在缺陷。

由于前端 NIO，后端 BIO，于是另一个有趣的设计产生了——两个线程池，前端 NIO 部分一个线程池，后端 BIO 部分一个线程池。各自相互不干扰，但这个设计的结果，导致了线程的浪费，也对性能调优带来很大的困难。

由于后端是 BIO，所以，也是 Cobar 吞吐量无法太高、另外也是其假死的根源。

MyCAT 在 Cobar 的基础上，完成了彻底的 NIO 通讯，并且合并了两个线程池，这是很大一个提升。从 1.1 版本开始，MyCAT 则彻底用了 JDK7 的 AIO，有一个重要提升。

2.1.1.5 第五个秘密：阻塞、又见阻塞

Cobar 本质上类似一个交换机，将后端 Mysql 的返回结果数据经过加工后再写入前端连接并返回，于是前后端连接都存在一个“写队列”用作缓冲，后端返回的数据发到前端连接 FrontConnection 的写队列中排队等待被发送，而通常情况下，后端写入的速度要大于前端消费的速度，在跨分片查询的情况下，这个现象更为明显，于是写线程就在这里又一次被阻塞。

解决办法有两个，增大每个前端连接的“写队列”长度，减少阻塞出现的情况，但此办法只是将问题抛给了使用者，要是使用者能够知道这个写队列的默认值小了，然后根据情况进行手动尝试调整也行，但 Cobar 的代码中并没有把这个问题暴露出来，比如写一个告警日志，队列满了，建议增大队列数。于是绝大多数情况下，大家就默默的排队阻塞，无人知晓。

MyCAT 解决此问题的方式则更加人性化，首先将原先数组模式的固定长度的队列改为链表模式，无限制，并且并发性更好，此外，为了让用户知道是否队列过长了（一般是因为 SQL 结果集返回太多，比如 1 万条记录），当超过指定阈值（可配）后，会产生一个告警日志。

```
<system> <property name="frontWriteQueueSize">1024</property> </system>
```

2.1.1.6 第六个秘密：又爱又恨的 SQL 批处理模式

正如一枚硬币的正反面无法分离，一块磁石怎样切割都有南北极，爱情中也一样，爱与恨总是纠缠着，无法理顺，而 Cobar 的 SQL 批处理模式，也恰好是这样一个令人又爱又恨的个性。

通常的 SQL 批处理，是将一批 SQL 作为一个处理单元，一次性提交给数据库，数据库顺序处理完以后，再返回处理结果，这个特性对于数据批量插入来说，性能提升很大，因此也被普遍应用。JDBC 的代码通常如下：

```
String sql = "insert into travelrecord (id,user_id,traveldate,fee,days) values(?,?,?,?);";
ps = con.prepareStatement(sql);
for (Map<String, String> map : list) {
    ps.setLong(1, Long.parseLong(map.get("id")));
    ps.setString(2, (String) map.get("user_id"));
    ps.setString(3, (String) map.get("traveldate"));
    ps.setString(4, (String) map.get("fee"));
    ps.setString(5, (String) map.get("days"));
    ps.addBatch();
}
ps.executeBatch();
con.commit();
ps.clearBatch();
```

但 Cobar 的批处理模式的实现，则有几个地方是与传统不同的：

- 提交到 cobar 的批处理中的每一条 SQL 都是单独的数据库连接来执行的；
- 批处理中的 SQL 并发执行。

并发多连接同时执行，则意味着 Batch 执行速度的提升，这是让人惊喜的一个特性，但单独的数据库连接并发执行，则又带来一个意外的副作用，即事务跨连接了，若一部分事务提交成功，而另一部分失败，则导致脏数据问题。看到这里，你是该“爱”呢还是该“恨”？

先不用急着下结论，我们继续看看 Cobar 的逻辑，SQL 并发执行，其实也是依次获取独立连接并执行，因此还是有稍微的时间差，若某一条失败了，则 cobar 会在会话中标记“事务失败，需要回滚”，下一个没执行的 SQL 就抛出异常并跳过执行，客户端就捕获到异常，并执行 rollback，回滚事务。绝大多数情况下，数据库正常

运行，此刻没有宕机，因此事务还是完整保证了，但万一恰好在某个 SQL commit 指令的时候宕机，于是杯具了，部分事务没有完成，数据没写入。但这个概率有多大呢？一条 insert insert 语句执行 commit 指令的时间假如是 50 毫秒，100 条同时提交，最长跨越时间是 5000 毫秒，即 5 秒中，而这个 C 指令的时间占据程序整个插入逻辑的时间的最多 20%，假如程序批量插入的执行时间占整个时间的 20%（已经很大比例了），那就是 $20\% \times 20\% = 4\%$ 的概率，假如机器的可靠性是 99.9%，则遇到失败的概率是 $0.1\% \times 4\% = \text{十万分之四}$ 。十万分之四，意味着 99.996% 的可靠性，亲，可以放心了么？

另外一个问题，即批量执行的 SQL，通常都是 insert 的，插入成功就 OK，失败的怎么办？通常会记录日志，重新找机会再插入，因此建议主键是能日志记录的，用于判断数据是否已经插入。

最后，假如真要多个 SQL 使用同一个后端 MYSQL 连接并保持事务怎么办？就采用通常的事务模式，单条执行 SQL，这个过程中，Cobar 会采用 Session 中上次用过的物理连接执行下一个 SQL 语句，因此，整个过程是与通常的事务模式完全一致。

2.1.1.7 第七个秘密：庭院深深锁清秋

说起死锁，貌似我们大家都只停留在很久远的回忆中，只在教科书里看到过，也看到过关于死锁产生的原因以及破解方法，只有 DBA 可能会偶尔碰到数据库死锁的问题。但很多用了 Cobar 的同学后来经常发现一个奇怪的问题，SQL 很久没有应答，百思不得其解，无奈之下找 DBA 排查后发现竟然有数据库死锁现象，而且比较频繁发生。要搞明白为什么 Cobar 增加了数据库死锁的概率，只能从源码分析，当一个 SQL 需要拆分为多条 SQL 去到多个分片上执行的时候，这个执行过程是并发执行的，即 N 个 SQL 同时在 N 个分片上执行，这个过程抽象为教科书里的事务模型，就变成一个线程需要锁定 N 个资源并执行操作以后，才结束事务。当这 N 个资源的锁定顺序是随机的情况下，那么就很容易产生死锁现象，而恰好 Cobar 并没有保证 N 个资源的锁定顺序，于是我们再次荣幸“中奖”。

2.1.1.8 第八个秘密：出乎意料连接池

数据库连接池，可能是仅次于线程池的我们所最依赖的“资源池”，其重要性不言而喻，业界也因此而诞生了多个知名的开源数据库连接池。我们知道，对于一个 MySQL Server 来说，最大连接通常是 1000-3000 之间，这些连接对于通常的应用足够了，通常每个应用一个 Database 独占连接，因此足够用了，而到了 Cobar 的分表分库这里，就出现了问题，因为 Cobar 对后端 MySQL 的连接池管理是基于分片——Database 来实现的，而不是整个 MySQL 的连接池共享，以一个分片数为 100 的表为例，假如 50 个分片在 Server1 上，就意味着 Server1

上的数据库连接被切分为 50 个连接池，每个池是 20 个左右的连接，这些连接池并不能互通，于是，在分片表的情况下，我们的并发能力被严重削弱。明明其他水池的水都是满的，你却只能守着空池子等待。。。

2.1.1.9 第九个秘密：无奈的热装载

Cobar 有一个优点，配置文件热装载，不用重启系统而热装载配置文件，但这里存在几个问题，其中一个问题是很多人不满的，即每次重载都把后端数据库重新断连一次，导致业务中断，而很多时候，大家改配置仅仅是为了修改分片表的定义，规则，增加分片表或者分片定义，而不会改变数据库的配置信息，这个问题由来已久，但却不太好修复。

2.1.1.10 第十个秘密：不支持读写分离

不支持读写分离，可能熟悉相关中间件的同学第一反应就是惊讶，因为一个 MySQL Proxy 最基本的功能就是提供读写分离能力，以提升系统的查询吞吐量和查询性能。但的确 Cobar 不支持读写分离，而且根据 Cobar 的配置文件，要实现读写分离，还很麻烦。可能有些人认为，因为无法保证读写分离的时延，因此无法确定是否能查到之前写入的数据，因此读写分离并不重要，但实际上，Mycat 的用户里，几乎没有不使用读写分离功能的，后来还有志愿者增加了强制查询语句走主库（写库）的功能，以解决刚才那个问题。

2.1.1.11 第十一个秘密：不可控的主从切换

Cobar 提供了 MySQL 主从切换能力，这个功能很实用也很方便，但你无法控制它的切换开启或关闭，有时候我们不想它自动切换，因为到目前为止，还没有什么好的方法来确认 MySQL 写节点宕机的时候，备节点是否已经 100%完成数据同步，因此存在数据不一致的风险，如何更可靠的确定是否能安全切换，这个问题比较复杂，Mycat 也一直在努力完善这个特性。

2.2 Mycat 闪耀登场

*当大批软件工程师开始觉醒，用互联网思维思考和规划自己的人生，第四次工业革命才拉开序幕——
《Mycat 宣言》*

Mycat 最早的版本完成于 2013 年年底，实现于雾霾中的北京城。

Mycat 要解决的第一个问题就是要将 Cobar 后端实现为非阻塞模式。将 Cobar 从“个人版”提升到真正的“企业版”。据未经证实的渠道了解，非开源的 Cobar 内部版本已经实现后端 NIO，但是并没有开源出来。于是 Mycat 注定要诞生了，尽管可能不会是 Leader-us 发起的。

但软件界里，总会有那么一些桀骜不驯的人，用一个电脑，在某一个不经意的晚上，写了一段代码，惊艳了这个世界。

Mycat 的前身是 OpencloudDB，而现在的 Mycat QQ 群则用来开发一个叫做 MycloudOA 的云平台的 SAAS 企业办公软件的，半年的时间里，这个群聚集了一大帮 IT 人，拥有超过 10 个“顾问”头衔的、超过十个“架构师”头衔的、超过 20 个“研发”头衔的庞大志愿者团队，然后，仅有不到 3 个人提交过文档和少量代码，其他的人都很专业的谈论着需求、谈论着框架、谈论着市场，最后的最后，大家都变成了资深酱油瓶，于是 MycloudOA 出师未捷身先死。

OpencloudDB 改名为 Mycat，一个原因是简单好记，另外一个原因，是打算未来入驻 Apache。因为 Apache Tomcat 也是一只猫，从年龄来看，Tomcat 算是 Mycat 表姐吧，从相貌身材来看，Tomcat 她表妹，绝对是东方第一萌妹子，虽然目前 Rainbow 大侠设计的 Mycat Logo，看起来是个 100% 的女汉子。

Mycat 1.0 的发布，立即引起不少人的关注，曾经参与 MycloudOA 开发的一些小伙伴陆续加入进来，资深酱油师 Michael 还注册了一个 openclouddb 的网站，随后又实现了 Mycat 全局序列号（基于文件方式）；一些了解或使用过 Cobar 的同学也陆续加入，网名为无影的大侠，提供了最早的 Mycat 分页排序的源码，最早在生产系统上部署了 Mycat 并且采用 HA Proxy 方式做高可用方案；随后，一个叫做小鱼的 PHP 高手，在不到 3 个月时间内，用 Mycat 改造了原先的电商系统。后来又有一些美容美发的 SAAS 创业项目采用了 Mycat；再后来，一些比较大的电信软件领域的公司和项目开始使用 Mycat，他们中的大多数都对 Mycat 做过不少的贡献，比如测试，Bug 修复等。发展到今天，Mycat 核心研发团队里的大多数人，都是来自上述这些公司。

Mycat 1.3 的诞生，是 Mycat 历史上最重大的一个里程碑。在这个版本里，需求、测试和功能开发各项工作，首次从个人为主变为开源团队为主的模式，更多的人参与到需求、开发、测试以及 Bug 修复活动中，基本上确定的 Bug 都在 2 4 小时内修复并有志愿者或用户确认修复。Mycat 1.3 版本的性能与 1.2 比提升巨大，功能更完备，这是因为包括武、成都-研发、冰峰影、Leader-us 等实力派编程高手各自负责一部分重要模块并一起协同研发，后来又加入聆听、从零开始、南哥、Mclaren、兵临城下等新的一批实力派编程达人，以及正在排队等待收编的 PCY 实力派干将，其他关于参与 Mycat 官网建设、文档编写和翻译的就更多了（当然也失联很多）。截至目前，Mycat 志愿者团队有以 Marshy 大美女为首的负责官网和广告的团队，以 Leader-us 为首的负责 Mycat-Server 研发的团队、以 Rainbow 为首的 Mycat-Web 的研发团队、以海王星为首的 QA 团队，以及群龙无首的测试团队和 DBA 团队。

此外，Mycat 开源社区正在进一步强化数据库监控、智能调优等方面的功能，未来将实现一键优化的能力，根据拦截到的 SQL 的执行统计数据，自动分析热点数据、给出建议的索引和优化措施以及读写分离的建议，DBA 一键完成优化，数据迁移也将可以在节目上点击鼠标完成。

Mycat 截至到 2015 年 4 月，保守估计已经有超过 60 个项目在使用，主要应用在电信领域、互联网项目，大部分是交易和管理系统，少量是信息系统。比较大的系统中，数据规模单表单月 30 亿。以后 Mycat 和 Mycat 社区成为 IT 和互联网创业的最佳伴侣。

下面信息是使用者在 Mycat github 上公布的使用案例：

soforth commented 29 days ago



不错，点个赞，目前我们部署了2台机器，处理近亿条数据。

jakbb commented 27 days ago



运行在安智账户系统中，数据量单表总量6KW，20多张表，上亿条数据。运行良好，高并发下偶尔出现sql操作有缓存延迟的现象

Hisen158 commented 27 days ago



公安某项目已上线，主要使用mycat分库分表服务于web系统做代理统计查询，数据总计目前20个表，30亿数据，选取适合的业务使用mycat，而非所有业务都依托于mycat.

kdalan commented 19 days ago



某电影票务行业系统，支撑线下1200家影院POS设备的刷卡及券类验证，使用Mycat-server-1.2.2做为数据库访问中间件，一期已上线并稳定运行2月余

dicome commented 14 days ago



联通某系统已上线Mycat10个月左右，从1.1版本开始到现在的1.3，采用分库+按日分区的方式，大概15个表左右，只保留一个月数据量，超期迁走，其中几个大表单表数据量约保持1.5亿左右。总数据量没统计，估算在7亿左右，运行稳定。

suxl commented 13 days ago



移动医疗产品，使用技术架构(spring+spring mvc+mybatis+mycat(mysql集群)+redis+nginx+Haproxy)，使用mycat主要应用于采集数据的分布式存储，实现分库分表和读写分离，目前数据量在1.5亿左右，并且在不断增长中，预计今年将突破5亿。

abirdman commented 13 days ago



大型零售系统，支持全国2万家以上门店使用，预计全面上线后每月新增订单1千万左右。最大的表会1年2亿左右。产品目前开发阶段，2015年4月8日上线初始版本，全面上线支持2w+门店的阶段还需要一定的时间。

huifeng168 commented 2 days ago



征信辅助系统，数据量单表总量10KW，目前系统运行良好，使用mycat之后，不像以前要经常时不时的去看数据库是否正常了。

更多案例请点击：

<https://github.com/MyCAT/apache/Mycat-Server/issues/112>

2.3 Mycat 概述

2.3.1 功能介绍

Mycat 是什么？从定义和分类来看，它是一个开源的分布式数据库系统，是一个实现了 MySQL 协议的 Server，前端用户可以把它看作是一个数据库代理，用 MySQL 客户端工具和命令行访问，而其后端可以用 MySQL 原生 (Native) 协议与多个 MySQL 服务器通信，也可以用 JDBC 协议与大多数主流数据库服务器通信，其核心功能是分表分库，即将一个大表水平分割为 N 个小表，存储在后端 MySQL 服务器里或者其他数据库里。

Mycat 发展到目前的版本，已经不是一个单纯的 MySQL 代理了，它的后端可以支持 MySQL、SQL Server、Oracle、DB2、PostgreSQL 等主流数据库，也支持 MongoDB 这种新型 NoSQL 方式的存储，未来还会支持更多类型的存储。而在最终用户看来，无论是那种存储方式，在 Mycat 里，都是一个传统的数据库表，支持标准的 SQL 语句进行数据的操作，这样一来，对前端业务系统来说，可以大幅降低开发难度，提升开发速度，在测试阶段，可以将一个表定义为任何一种 Mycat 支持的存储方式，比如 MySQL 的 MyASIM 表、内存表、或者 MongoDB、LevelDB 以及号称是世界上最快的内存数据库 MemSQL 上。试想一下，用户表存放在 MemSQL 上，

大量读频率远超过写频率的数据如订单的快照数据存放于 InnoDB 中，一些日志数据存放于 MongoDB 中，而且还能把 Oracle 的表跟 MySQL 的表做关联查询，你是否有一种不能呼吸的感觉？而未来，还能通过 Mycat 自动将一些计算分析后的数据灌入到 Hadoop 中，并能用 Mycat+Storm/Spark Stream 引擎做大规模数据分析，看到这里，你大概明白了，Mycat 是什么？Mycat 就是 BigSQL，Big Data On SQL Database。

对于 DBA 来说，可以这么理解 Mycat：

Mycat 就是 MySQL Server，而 Mycat 后面连接的 MySQL Server，就好象是 MySQL 的存储引擎，如 InnoDB，MyISAM 等，因此，Mycat 本身并不存储数据，数据是在后端的 MySQL 上存储的，因此数据可靠性以及事务等都是 MySQL 保证的，简单的说，Mycat 就是 MySQL 最佳伴侣，它在一定程度上让 MySQL 拥有了能跟 Oracle PK 的能力。

对于软件工程师来说，可以这么理解 Mycat：

Mycat 就是一个近似等于 MySQL 的数据库服务器，你可以用连接 MySQL 的方式去连接 Mycat（除了端口不同，默认的 Mycat 端口是 8066 而非 MySQL 的 3306，因此需要在连接字符串上增加端口信息），大多数情况下，可以用你熟悉的对象映射框架使用 Mycat，但建议对于分片表，尽量使用基础的 SQL 语句，因为这样能达到最佳性能，特别是几千万甚至几百亿条记录的情况下。

对于架构师来说，可以这么理解 Mycat：

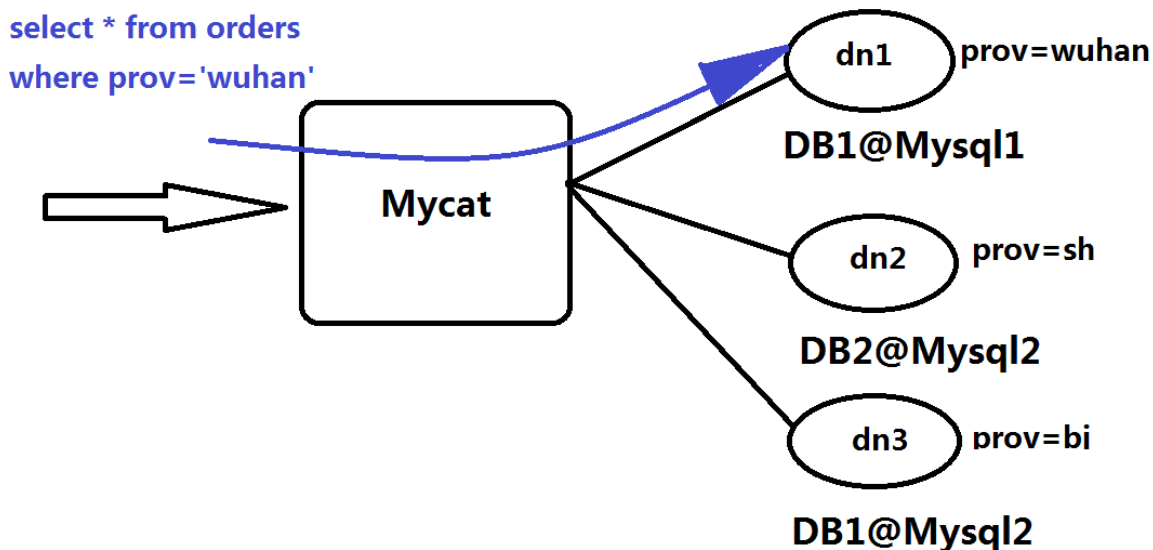
Mycat 是一个强大的数据库中间件，不仅仅可以用作读写分离、以及分表分库、容灾备份，而且可以用于多租户应用开发、云平台基础设施、让你的架构具备很强的适应性和灵活性，借助于即将发布的 Mycat 智能优化模块，系统的数据访问瓶颈和热点一目了然，根据这些统计分析数据，你可以自动或手工调整后端存储，将不同的表映射到不同存储引擎上，而整个应用的代码一行也不用改变。

当前是个大数据的时代，但究竟怎样规模的数据适合数据库系统呢？对此，国外有一个数据库领域的权威人士说了一个结论：千亿以下的数据规模仍然是数据库领域的专长，而 Hadoop 等这种系统，更适合的是千亿以上的规模。所以，Mycat 适合 1000 亿条以下的单表规模，如果你的数据超过了这个规模，请投靠 Mycat Plus 吧！

2.3.2 Mycat 原理

Mycat 的原理并不复杂，复杂的是代码，如果代码也不复杂，那么早就成为一个传说了。

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



上述图片里，Orders 表被分为三个分片 datanode（简称 dn），这三个分片是分布在两台 MySQL Server 上 (DataHost)，即 datanode=database@datahost 方式，因此你可以用一台到 N 台服务器来分片，分片规则为 (sharding rule)典型的字符串枚举分片规则，一个规则的定义是分片字段 (sharding column)+分片函数(rule function)，这里的分片字段为 prov 而分片函数为字符串枚举方式。

当 Mycat 收到一个 SQL 时，会先解析这个 SQL，查找涉及到的表，然后看此表的定义，如果有分片规则，则获取到 SQL 里分片字段的值，并匹配分片函数，得到该 SQL 对应的分片列表，然后将 SQL 发往这些分片去执行，最后收集和处理所有分片返回的结果数据，并输出到客户端。以 select * from Orders where prov=?语句为例，查到 prov=wuhan，按照分片函数，wuhan 返回 dn1，于是 SQL 就发给了 MySQL1，去取 DB1 上的查询结果，并返回给用户。

如果上述 SQL 改为 select * from Orders where prov in ('wuhan' , 'beijing')，那么，SQL 就会发给 MySQL1 与 MySQL2 去执行，然后结果集合并后输出给用户。但通常业务中我们的 SQL 会有 Order By 以及 Limit 翻页语法，此时就涉及到结果集在 Mycat 端的二次处理，这部分的代码也比较复杂，而最复杂的则属两个表的 Join 问题，为此，Mycat 提出了创新性的 ER 分片、全局表、HBT (Human Brain Tech)人工智能的 Catlet、以及结合 Storm/Spark 引擎等十八般武艺的解决办法，从而成为目前业界最强大的方案，这就是开源的力量！

2.3.3 应用场景

Mycat 发展到现在，适用的场景已经很丰富，而且不断有新用户给出新的创新性的方案，以下是几个典型的应用场景：

- 单纯的读写分离，此时配置最为简单，支持读写分离，主从切换；
- 分表分库，对于超过 1000 万的表进行分片，最大支持 1000 亿的单表分片；
- 多租户应用，每个应用一个库，但应用程序只连接 Mycat，从而不改造程序本身，实现多租户化；
- 报表系统，借助于 Mycat 的分表能力，处理大规模报表的统计；
- 替代 Hbase，分析大数据；
- 作为海量数据实时查询的一种简单有效方案，比如 100 亿条频繁查询的记录需要在 3 秒内查询出来结果，

除了基于主键的查询，还可能存在范围查询或其他属性查询，此时 Mycat 可能是最简单有效的选择。

2.3.4 Mycat 长期路线图

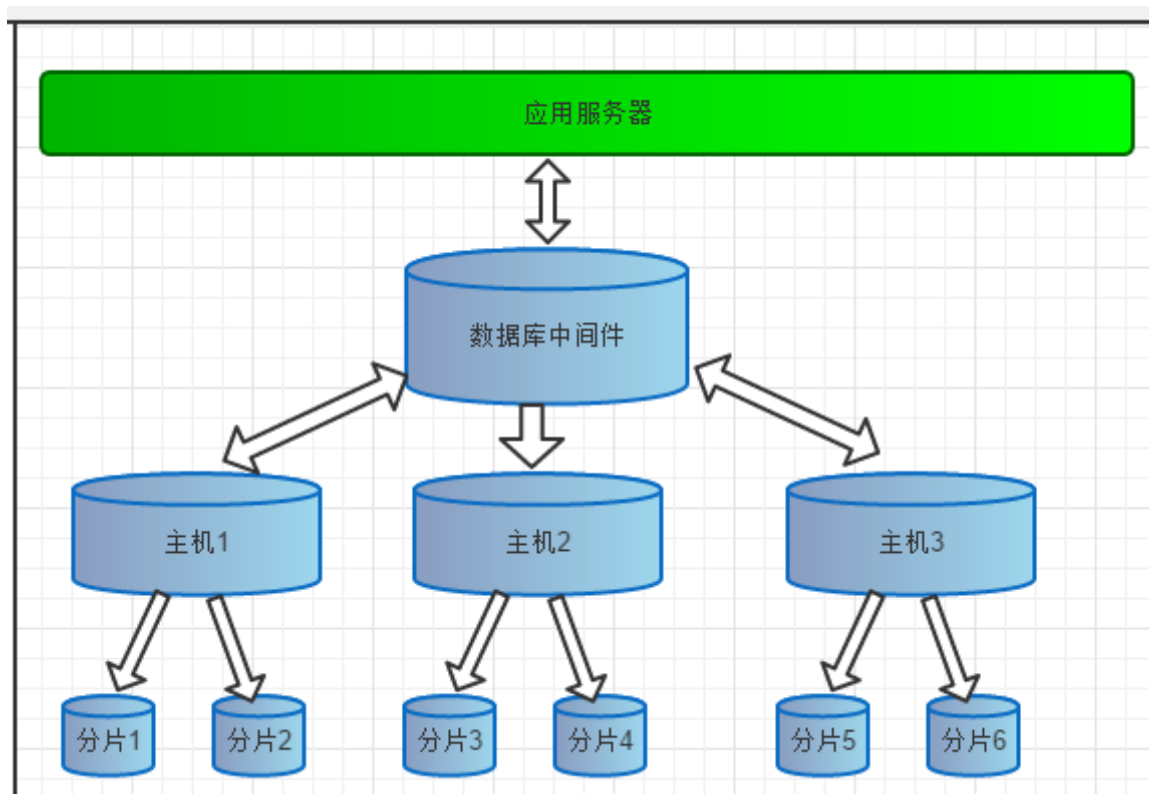
- 强化分布式数据库中间件的方面的功能，使之具备丰富的插件、强大的数据库智能优化功能、全面的系统监控能力、以及方便的数据运维工具，实现在线数据扩容、迁移等高级功能。
- 进一步挺进大数据计算领域，深度结合 Spark Stream 和 Storm 等分布式实时流引擎，能够完成快速的巨表关联、排序、分组聚合等 OLAP 方向的能力，并集成一些热门常用的实时分析算法，让工程师以及 DBA 们更容易用 Mycat 实现一些高级数据分析处理功能。
- 不断强化 Mycat 开源社区的技术水平，吸引更多的 IT 技术专家，使得 Mycat 社区成为中国的 Apache，并将 Mycat 推到 Apache 基金会，成为国内顶尖开源项目，最终能够让一部分志愿者成为专职的 Mycat 开发者，荣耀跟实力一起提升。
- 依托 Mycat 社区，聚集 100 个 CXO 级别的精英，众筹建设亲亲山庄，Mycat 社区+亲亲山庄=中国最大 IT O2O 社区。

第 3 章 Mycat 中的概念

3.1 数据库中间件

前面讲了 Mycat 是一个开源的分布式数据库系统，但是由于真正的数据库需要存储引擎，而 Mycat 并没有存储引擎，所以并不是完全意义的分布式数据库系统。

那么 Mycat 是什么？Mycat 是数据库中间件，就是介于数据库与应用之间，进行数据处理与交互的中间服务。由于前面讲的对数据进行分片处理之后，从原有的一个库，被切分为多个分片数据库，所有的分片数据库集群构成了整个完整的数据库存储。



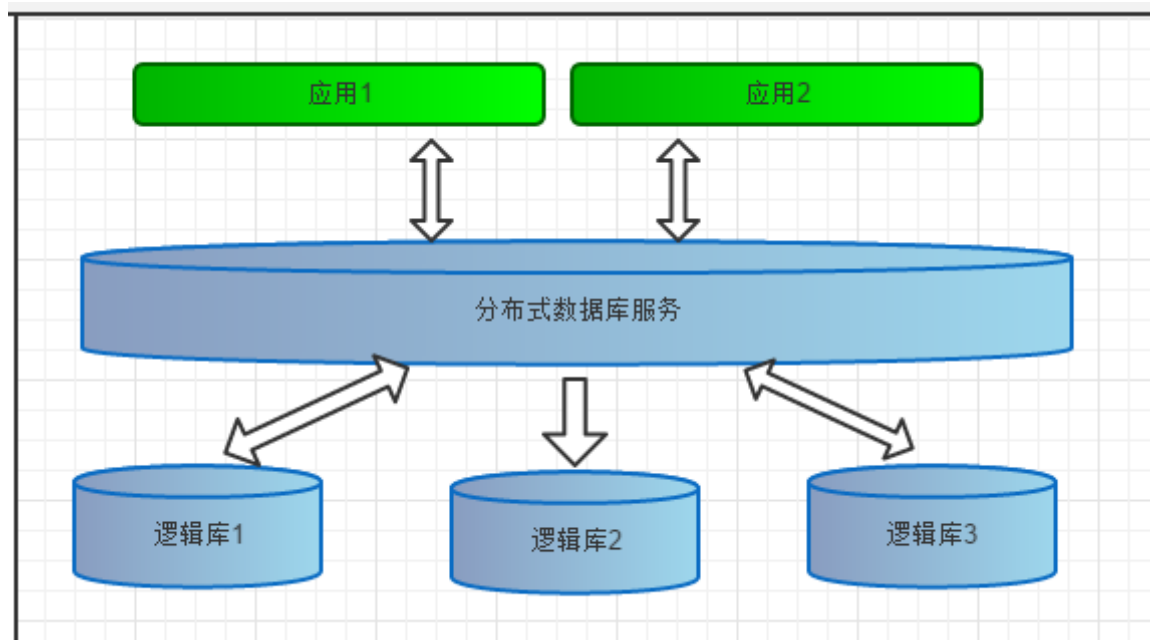
如上图所表示，数据被分到多个分片数据库后，应用如果需要读取数据，就要需要处理多个数据源的数据。如果没有数据库中间件，那么应用将直接面对分片集群，数据源切换、事务处理、数据聚合都需要应用直接处理，原本该是专注于业务的应用，将会花大量的工作来处理分片后的问题，最重要的是每个应用处理将是完全的重复造轮子。

所以有了数据库中间件，应用只需要集中与业务处理，大量的通用的数据聚合，事务，数据源切换都由中间件来处理，中间件的性能与处理能力将直接决定应用的读写性能，所以一款好的数据库中间件至关重要。

3.2 逻辑库(schema)

前面一节讲了数据库中间件，通常对实际应用来说，并不需要知道中间件的存在，业务开发人员只需要知道数据库的概念，所以数据库中间件可以被看做是一个或多个数据库集群构成的逻辑库。

在云计算时代，数据库中间件可以以多租户的形式给一个或多个应用提供服务，每个应用访问的可能是一个独立或者是共享的物理库，常见的如阿里云数据库服务器 RDS。



3.3 逻辑表 (table)

3.3.1 逻辑表

既然有逻辑库，那么就会有逻辑表，分布式数据库中，对应用来说，读写数据的表就是逻辑表。逻辑表，可以是数据切分后，分布在一个或多个分片库中，也可以不做数据切分，不分片，只有一个表构成。

3.3.2 分片表

分片表，是指那些原有的很大数据的表，需要切分到多个数据库的表，这样，每个分片都有一部分数据，所有分片构成了完整的数据。

例如在 mycat 配置中的 `t_node` 就属于分片表，数据按照规则被分到 `dn1, dn2` 两个分片节点(dataNode)上。

```
<table name="t_node" primaryKey="vid" autoIncrement="true" dataNode="dn1,dn2" rule="rule1" />
```

3.3.3 非分片表

一个数据库中并不是所有的表都很大，某些表是可以不用进行切分的，非分片是相对分片表来说的，就是那些不需要进行数据切分的表。

如下配置中 `t_node`，只存在于分片节点 (dataNode) `dn1` 上。

```
<table name="t_node" primaryKey="vid" autoIncrement="true" dataNode="dn1" />
```

3.3.4 ER 表

关系型数据库是基于实体关系模型 (Entity-Relationship Model) 之上，通过其描述了真实世界中事物与关系，Mycat 中的 ER 表即是来源于此。根据这一思路，提出了基于 E-R 关系的数据分片策略，子表的记录与所关联的父表记录存放在同一个数据分片上，即子表依赖于父表，通过表分组 (Table Group) 保证数据 Join 不会跨库操作。

表分组 (Table Group) 是解决跨分片数据 join 的一种很好的思路，也是数据切分规划的重要一条规则。

3.3.5 全局表

一个真实的业务系统中，往往存在大量的类似字典表的表，这些表基本上很少变动，字典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。

对于这类的表，在分片的情况下，当业务表因为规模而进行分片以后，业务表与这些附属的字典表之间的关联，就成了比较棘手的问题，所以 Mycat 中通过数据冗余来解决这类表的 join，即所有的分片都有一份数据的拷贝，所有将字典表或者符合字典表特性的一些表定义为全局表。

数据冗余是解决跨分片数据 join 的一种很好的思路，也是数据切分规划的另外一条重要规则。

3.4 分片节点(dataNode)

数据切分后，一个大表被分到不同的分片数据库上面，每个表分片所在的数据库就是分片节点 (dataNode)。

3.5 节点主机(dataHost)

数据切分后，每个分片节点 (dataNode) 不一定会独占一台机器，同一机器上面可以有多个分片数据库，这样一个或多个分片节点 (dataNode) 所在的机器就是节点主机 (dataHost)，为了规避单节点主机并发数限制，尽量将读写压力高的分片节点 (dataNode) 均衡的放在不同的节点主机 (dataHost)。

3.6 分片规则(rule)

前面讲了数据切分，一个大表被分成若干个分片表，就需要一定的规则，这样按照某种业务规则把数据分到某个分片的规则就是分片规则，数据切分选择合适的分片规则非常重要，将极大的避免后续数据处理的难度。

3.7 全局序列号(sequence)

数据切分后，原有的关系数据库中的主键约束在分布式条件下将无法使用，因此需要引入外部机制保证数据唯一性标识，这种保证全局性的数据唯一标识的机制就是全局序列号（sequence）。

3.8 多租户

多租户技术或称多重租赁技术，是一种软件架构技术，它是在探讨与实现如何于多用户的环境下共用相同的系统或程序组件，并且仍可确保各用户间数据的隔离性。在云计算时代，多租户技术在共用的数据中心以单一系统架构与服务提供多数客户端相同甚至可定制化的服务，并且仍然可以保障客户的数据隔离。目前各种各样的云计算服务就是这类技术范畴，例如阿里云数据库服务（RDS）、阿里云服务器等等。

多租户在数据存储上存在三种主要的方案，分别是：

3.8.1 独立数据库

这是第一种方案，即一个租户一个数据库，这种方案的用户数据隔离级别最高，安全性最好，但成本也高。

优点：

为不同的租户提供独立的数据库，有助于简化数据模型的扩展设计，满足不同租户的独特需求；

如果出现故障，恢复数据比较简单。

缺点：

增大了数据库的安装数量，随之带来维护成本和购置成本的增加。

这种方案与传统的一个客户、一套数据、一套部署类似，差别只在于软件统一部署在运营商那里。如果面对的是银行、医院等需要非常高数据隔离级别的租户，可以选择这种模式，提高租用的定价。如果定价较低，产品走低价路线，这种方案一般对运营商来说是无法承受的。

3.8.2 共享数据库，隔离数据架构

这是第二种方案，即多个或所有租户共享 Database，但是每个租户一个 Schema。

优点：

为安全性要求较高的租户提供了一定程度的逻辑数据隔离，并不是完全隔离；每个数据库可以支持更多的租户数量。

缺点：

如果出现故障，数据恢复比较困难，因为恢复数据库将牵扯到其他租户的数据；

如果需要跨租户统计数据，存在一定困难。

3.8.3 共享数据库，共享数据架构

这是第三种方案，即租户共享同一个 Database、同一个 Schema，但在表中通过 TenantID 区分租户的数据。这是共享程度最高、隔离级别最低的模式。

优点：

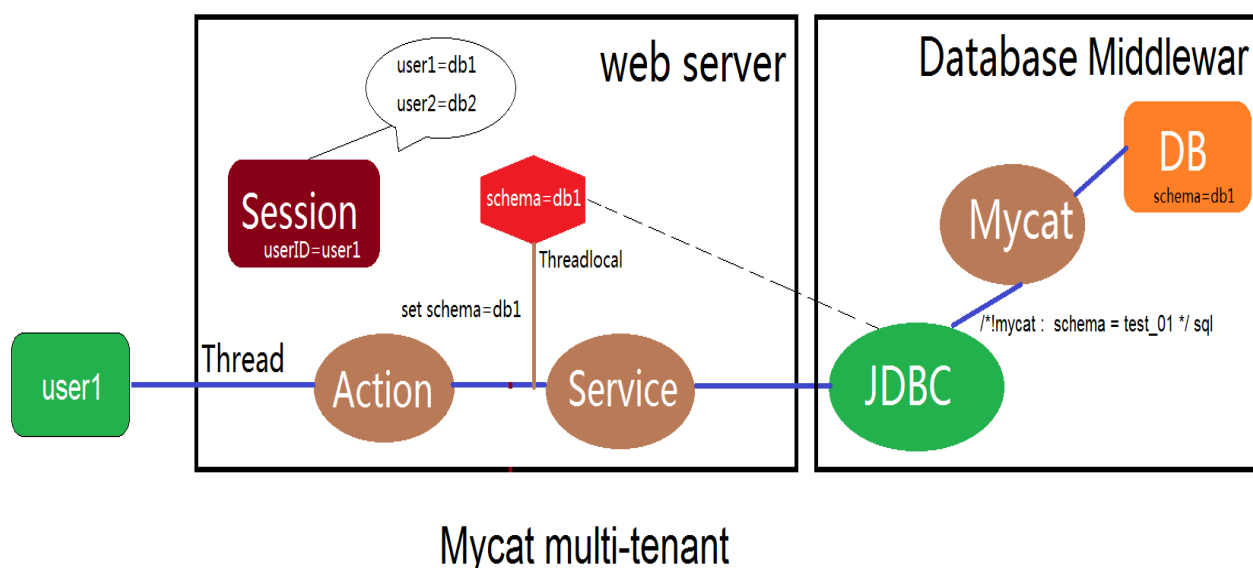
三种方案比较，第三种方案的维护和购置成本最低，允许每个数据库支持的租户数量最多。

缺点：

隔离级别最低，安全性最低，需要在设计开发时加大对安全的开发量；

数据备份和恢复最困难，需要逐表逐条备份和还原；

如果希望以最少的服务器为最多的租户提供服务，并且租户接受以牺牲隔离级别换取降低成本，这种方案最适合。



第 4 章快速入门

4.1 10 分钟入门

MyCAT 是使用 JAVA 语言进行编写开发，使用前需要先安装 JAVA 运行环境(JRE),由于 MyCAT 中使用了 JDK7 中的一些特性，所以要求必须在 JDK7 以上的版本上运行。

4.1.1 环境准备

- 1) JDK 下载

<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

注：必须 JDK7 或更高版本。

2) MySQL 下载

<http://dev.mysql.com/downloads/mysql/5.5.html#downloads>

注：MyCAT 支持多种数据库接入，如：MySQL、SQLServer、Oracle、MongoDB 等，推荐使用 MySQL 做集群。

3) MyCAT 项目主页

<https://github.com/MyCATApache/>

注：MyCAT 相关源码、文档都可以在此地址下进行下载。

4.1.2 环境安装与配置

如果是第一次刚接触 MyCAT，建议先下载 MyCAT-Server 源码到本地，通过 Eclipse 等工具进行配置和运行，便于深入了解和调试程序运行逻辑。

1) MyCAT-Server 源码下载

由于 MyCAT 源码目前主要托管在 github 上，需要先在本地安装和配置好相关环境，具体参考群共享中“github-eclipse 开发指南.docx”，这说明有很详细的配置说明，按照文档中的步骤把 MyCAT-Server 源码下载到本地即可。

MyCAT-Server 仓库地址：<https://github.com/MyCATApache/Mycat-Server.git>

2) 源码调试与配置

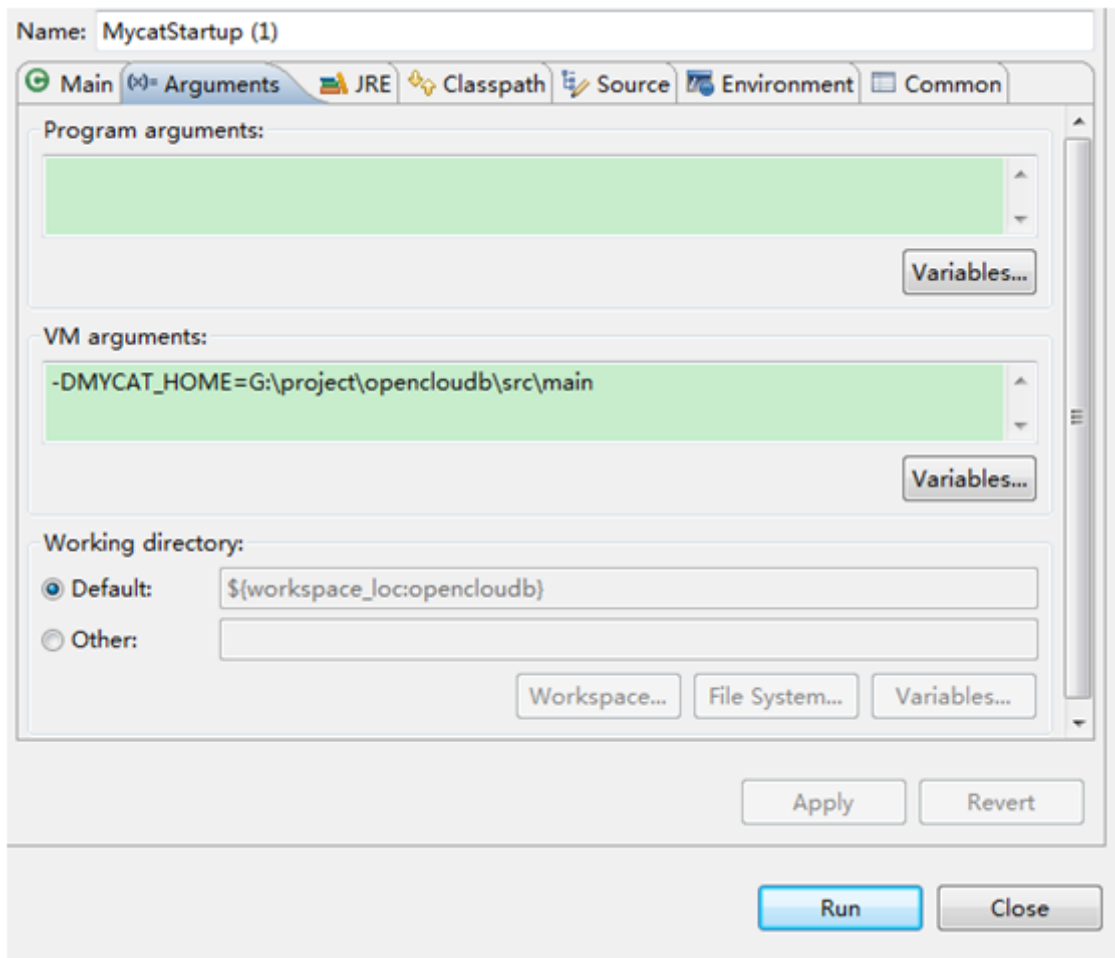
MyCAT 目前主要通过配置文件的方式来定义逻辑库和相关配置：

- MYCAT_HOME/conf/schema.xml 中定义逻辑库，表、分片节点等内容；
- MYCAT_HOME/conf/rule.xml 中定义分片规则；
- MYCAT_HOME/conf/server.xml 中定义用户以及系统相关变量，如端口等。

注：以上几个文件的具体配置请参考前面章节中的具体说明。

3) 源码运行

MyCAT 入口程序是 io.mycat.MycatStartup.java，右键 run as 出现下面的界面，需要设置 MYCAT_HOME 目录，为你工程当前所在目录(src/main)：



设置完 MYCAT 主目录后即可正常运行 MyCAT 服务。

注：若启动报错，DirectBuffer 内存不够，则可以再加 JVM 系统参数：

XX:MaxDirectMemorySize=128M

4.2 快速镜像方式体验 MyCAT

此方式通过将已经安装和配置好的 MySQL+MyCAT 做成镜像，可实现快速运行和体验 MyCAT 服务。

镜像文件及快速运行体验文档下载地址：

暂无

4.3 服务安装与配置

4.3.1 linux

MyCAT 有提供编译好的安装包，支持 windows、Linux、Mac、Solaris 等系统上安装与运行。

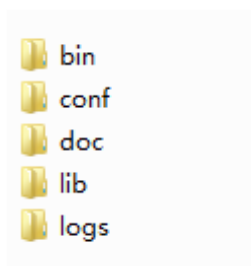
linux 下可以下载 Mycat-server-xxxxx.linux.tar.gz 解压在某个目录下, 注意目录不能有空格, 在

```
/usr/local/Mycat/  
    /bin  
    /conf  
    /doc  
    /lib  
    logs  
useradd Mycat  
chown -R Mycat.Mycat /usr/local/Mycat
```

Linux(Unix)下, 建议放在 usr/local/Mycat 目录下, 如下:

下面是修改 MyCAT 用户密码的方式(仅供参考):

```
[root@db1 ~]# passwd Mycat  
Changing password for user Mycat.  
New password:  
Retype new password:  
passwd: all authentication tokens updated successfully.  
[root@db1 ~]#
```



目录解释如下:

bin 程序目录, 存放了 window 版本和 linux 版本, 除了提供封装成服务的版本之外, 也提供了 nowrap 的 shell 脚本命令, 方便大家选择和修改, 进入到 bin 目录:

Linux 下运行: ./mycat console, 首先要 chmod +x *

注: mycat 支持的命令{ console | start | stop | restart | status | dump }

conf 目录下存放配置文件, server.xml 是 Mycat 服务器参数调整和用户授权的配置文件, schema.xml 是逻辑库定义和表以及分片定义的配置文件, rule.xml 是分片规则的配置文件, 分片规则的具体一些参数信息单独存放为文件, 也在这个目录下, 配置文件修改, 需要重启 Mycat 或者通过 9066 端口 reload.

lib 目录下主要存放 mycat 依赖的一些 jar 文件.

日志存放在 logs/mycat.log 中, 每天一个文件, 日志的配置是在 conf/log4j.xml 中, 根据自己的需要, 可以调整输出级别为 debug, debug 级别下, 会输出更多的信息, 方便排查问题.

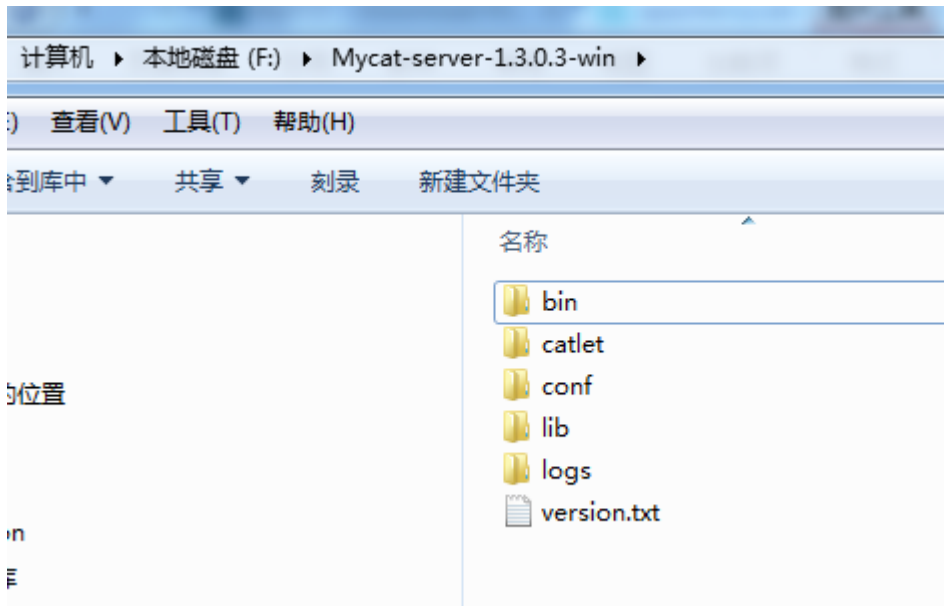
注意：Linux 下部署安装 MySQL，默认不忽略表名大小写，需要手动到/etc/my.cnf 下配置

lower_case_table_names=1 使 Linux 环境下 MySQL 忽略表名大小写，否则使用 MyCAT 的时候会提示找不到表的错误！

4.3.2 windows

MyCAT 有提供编译好的安装包，支持 windows、Linux、Mac、Solaris 等系统上安装与运行。

windows 下可以下载 Mycat-server-xxxxx-win.tar.gz 解压在某个目录下，建议解压到本地某个盘符根目录下，如下：



目录解释如下：

bin 程序目录，存放了 window 版本和 linux 版本，除了提供封装成服务的版本之外，也提供了 nowrap 的 shell 脚本命令，方便大家选择和修改，进入到 bin 目录：

Windows 下运行：运行: mycat.bat 在控制台启动程序，也可以装载成服务，若此程序运行有问题，也可以运行 startup_nowrap.bat，确保 java 命令可以在命令执行。

Windows 下将 MyCAT 做成系统服务：MyCAT 提供 warp 方式的命令，可以将 MyCAT 安装成系统服务并可启动和停止。

1) 进入 bin 目录下，输入 ./mycat start 启动 mycat 服务。

conf 目录下存放配置文件，server.xml 是 Mycat 服务器参数调整和用户授权的配置文件，schema.xml 是逻辑库定义和表以及分片定义的配置文件，rule.xml 是分片规则的配置文件，分片规则的具体一些参数信息单独存放为文件，也在这个目录下，配置文件修改，需要重启 Mycat 或者通过 9066 端口 reload。

lib 目录下主要存放 mycat 依赖的一些 jar 文件。

日志存放在 logs/mycat.log 中，每天一个文件，日志的配置是在 conf/log4j.xml 中，根据自己的需要，可以调整输出级别为 debug，debug 级别下，会输出更多的信息，方便排查问题。

4.4 服务启动与启动设置

4.4.1 linux

MyCAT 在 Linux 中部署启动时，首先需要在 Linux 系统的环境变量中配置 MYCAT_HOME,操作方式如下：

- 1) vi /etc/profile,在系统环境变量文件中增加 MYCAT_HOME=/usr/local/Mycat。
- 2) 执行 source /etc/profile 命令，使环境变量生效。

如果是在多台 Linux 系统中组建的 MyCAT 集群，那需要在 MyCAT Server 所在的服务器上配置对其他 ip 和主机名的映射，配置方式如下：

vi /etc/hosts

例如：我有 4 台机器，配置如下：

IP 主机名：

192.168.100.2 sam_server_1

192.168.100.3 sam_server_2

192.168.100.4 sam_server_3

192.168.100.5 sam_server_4

编辑完后，保存文件。

经过以上两个步骤的配置，就可以到/usr/local/Mycat/bin 目录下执行：

./mycat start

即可启动 mycat 服务！

4.4.2 windows

MyCAT 在 windows 中部署时，建议放在某个盘符的根目录下，如果不是在根目录下，请尽量不要放在包含中文的目录下

如：D:\Mycat-server-1.4-win\

命令行方式启动：

从 cmd 中执行命令到达 D:\Mycat-server-1.4-win\bin 目录下，执行 startup_nowrap.bat 即可启动 MyCAT 服务。

注：执行此命令时，需要确保 windows 系统中已经配置好了 JAVA 的环境变量，并可执行 java 命令。jdk 版本必须是 1.7 及以上版本。

服务方式启动：

未封装

4.5 基于 zk 的启动

1.5 开始会支持本地 xml 启动，以及从 zk 加载配置转为本地 xml 的两种方式，conf 下的 zk.conf 文件里设置 loadfromzk 参数默认为 false

如果没有这个文件，或者没有 loadfromzk 为 true 的参数，即从本地加载。下面介绍从 ZK 启动相关配置。

Zk-create.yaml 说明

1.5 正式引入 zookeeper（以下简称 zk）管理 Mycat-Server,启动 server 第一步是初始化 zk 数据，下面介绍初始化 zk 数据步骤，信息在 zk-create.yaml。Mycat ZK 配置文件详解：

[https://github.com/MyCATapache/Mycat-](https://github.com/MyCATapache/Mycat-doc/blob/master/%E8%AE%BE%E8%AE%A1%E6%96%87%E6%A1%A3/2.0/MyCat%20ZK%E9%85%8D%E7%BD%AE%E6%96%87%E4%BB%B6%E8%AF%A6%E8%A7%A3.docx)

[doc/blob/master/%E8%AE%BE%E8%AE%A1%E6%96%87%E6%A1%A3/2.0/MyCat%20ZK%E9%85%8D%E7%BD%AE%E6%96%87%E4%BB%B6%E8%AF%A6%E8%A7%A3.docx](https://github.com/MyCATapache/Mycat-doc/blob/master/%E8%AE%BE%E8%AE%A1%E6%96%87%E6%A1%A3/2.0/MyCat%20ZK%E9%85%8D%E7%BD%AE%E6%96%87%E4%BB%B6%E8%AF%A6%E8%A7%A3.docx)

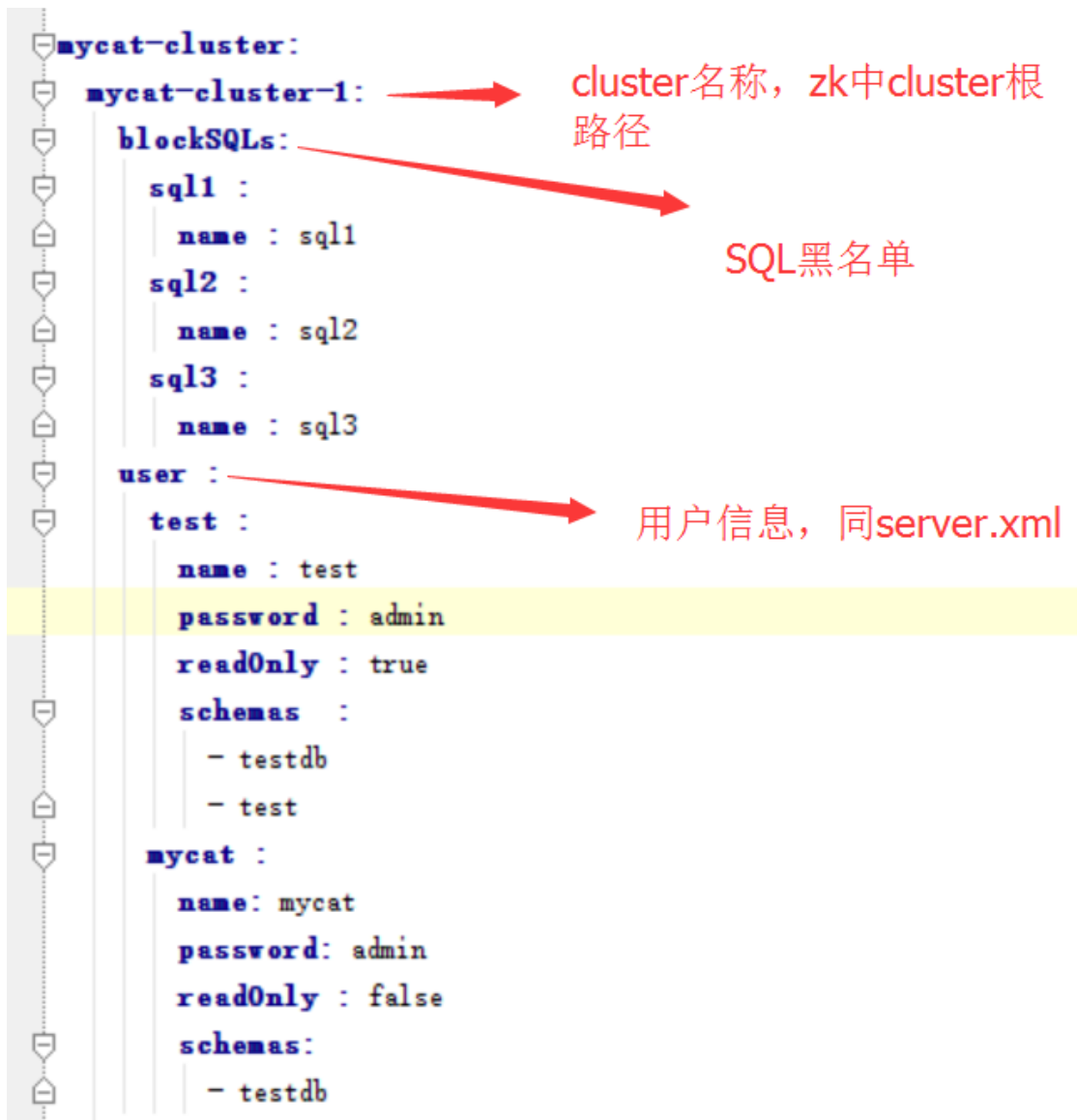
1、zk-create 总体结构



2、参数说明

2.1、zkURL,zk 连接地址

2.2、mycat-cluster



```

rule :
  sharding-by-enum :
    name : sharding-by-enum
    functionName : io.mycat.route.function.PartitionByFileMap
    column : create_time
    defaultnode : 0
    type : 0
    config :
      10000 : 0
      10010 : 1
  sharding-by-hour :
    name : sharding-by-hour
    functionName : io.mycat.route.function.LatestMonthPartion
    column : createTime
    splitOneDay : 24

  auto-sharding-long :
    name : auto-sharding-long
    column : id
    functionName : io.mycat.route.function.AutoPartitionByLong
    defaultNode : 0
    config :
      0-2000000 : 0
      2000001-4000000 : 1
      4000001-8000000 : 2

```

分片规则，同rule.xml

```

rule :
  sequence:
    sequence-3 :
      current_value : 100000
      increament : 100
    sequence-2 :
      workid: 1
      centerid : 2
    sequence-0 :
      type : file
    sequence-1 :
      type : 1
      config :
        current_value : 100000
        increament : 100
      sequence-mapping :
        T_NODE : 0

```

序列号配置



```
schema :
  TESTDB :
    name : TESTDB
    checkSQLSchema : false
    defaultMaxLimit : 100

    travelrecord :
      name : travelrecord
      datanode : dn1, dn2, dn3
      ruleName : auto-sharding-long

    company :
      name : company
      datanode : dn1, dn2, dn3
      primaryKey : ID
      type : 1 //全局表为 1

    goods :
      name : goods
      datanode : dn1, dn2
      primaryKey : ID
      type : 1 //全局表为 1
```

schema配置，同 schema.xml

```
datanode :
  dn1:
    name : dn1
    dataHost : localhost1
    database : db1
  dn2:
    name : dn2
    dataHost : localhost1
    database : db2
  dn3:
    name : dn3
    dataHost : localhost1
    database : db3
  offer_dn$0-127:
    name : offer_dn$0-127
    dataHost : localhost1
    database : db1$0-127
```

datanode,同schema.xml

```
datahost :
  localhost1 :
    name : localhost1
    maxcon : 1000
    mincon : 10
    balance : 0
    writetype : 0
    dbtype : mysql
    dbDriver : native
    switchType : 1
    slaveThreshold : 100
    heartbeatSQL : select user()
    mysqlGroup : mysql_rep_1
```

datahost,同schema.xml

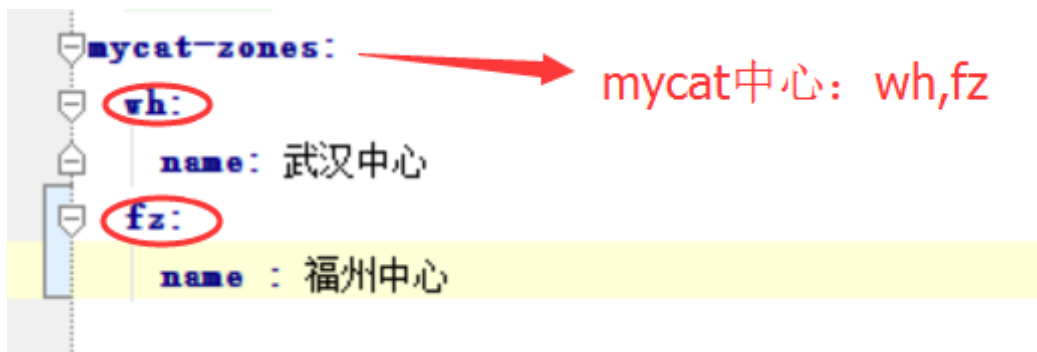
mysql复制组，原
schema.xml中
writehost/readhost,
具体配置在之后说明

2.3、mycat-hosts

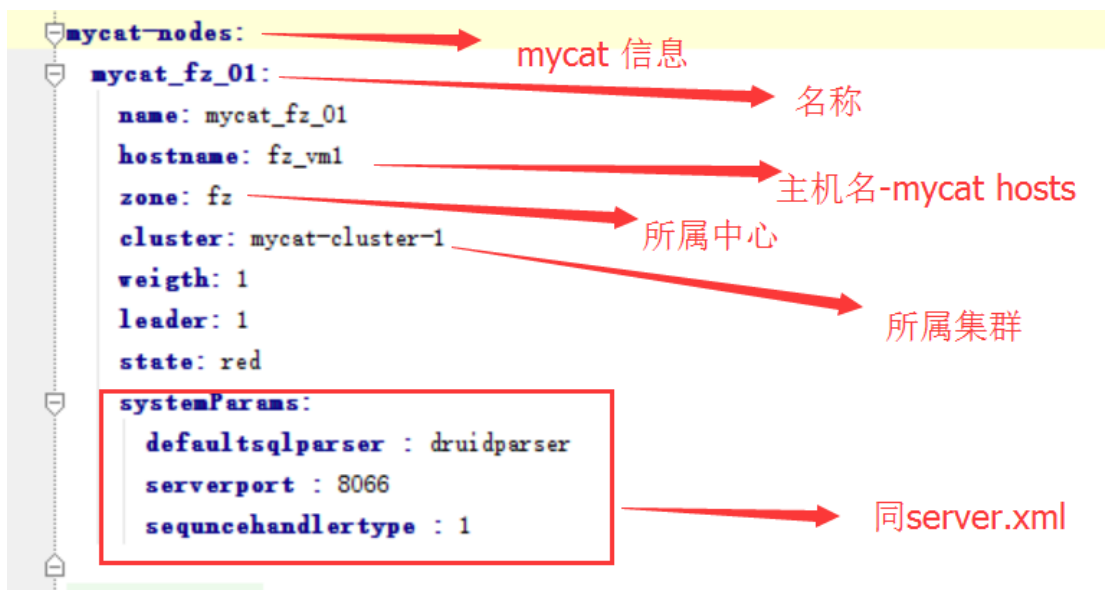
```
mycat-hosts:
  fz_vm1:
    hostname: fz_vm1
    ip: 192.168.10.2
    root: root
    password: admin
```

mycat 主机信息

2.4、mycat-zones



2.5、mycat-nodes



2.6、mycat-mysqls



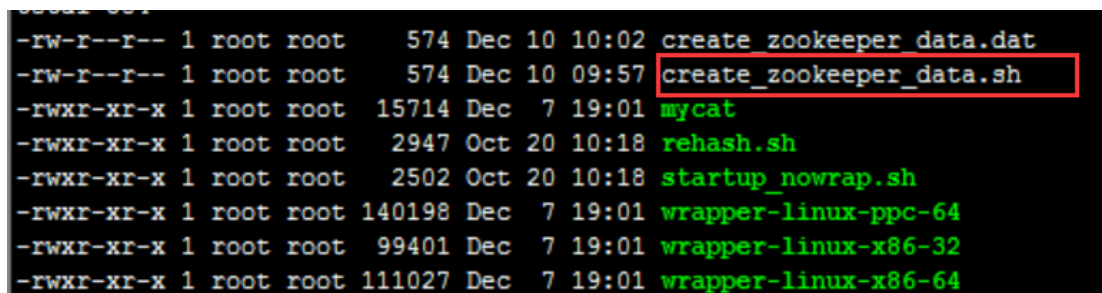
2.7、mysql-reps



Zk 初始化

1、进入 MYCAT/bin 目录

cd /data/test1/mycat/bin

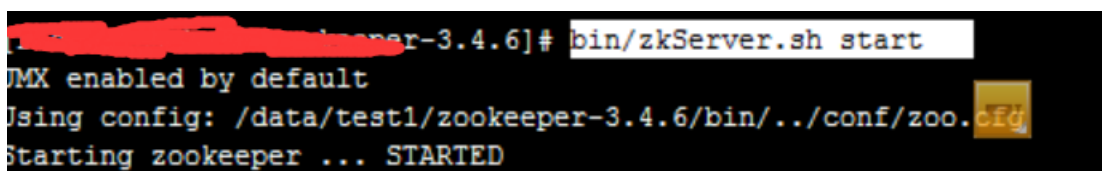


2、修改 MYCAT/conf/zk-create.yaml 内容

修改方法见 “Zk-create.yaml 说明”。

3、启动 ZK

启动 ZK: bin/zkServer.sh start



登陆 ZK: bin/zkCli.sh

```
[root@localhost ~]# cd /opt/zookeeper-3.4.6/
[root@localhost zookeeper-3.4.6]# bin/zkCli.sh
Connecting to localhost:2181
2015-12-10 10:26:46,938 [myid:] - INFO [main:Environment@100] - Client environment:zooke
2015-12-10 10:26:46,942 [myid:] - INFO [main:Environment@100] - Client environment:host.
2015-12-10 10:26:46,942 [myid:] - INFO [main:Environment@100] - Client environment:java.
2015-12-10 10:26:46,944 [myid:] - INFO [main:Environment@100] - Client environment:java.
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:java.
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:java.
n/./build/lib/*.jar:/data/test1/zookeeper-3.4.6/bin/./lib/slf4j-log4j12-1.6.1.jar:/data
ib/netty-3.7.0.Final.jar:/data/test1/zookeeper-3.4.6/bin/./lib/log4j-1.2.16.jar:/data/te
er-3.4.6.jar:/data/test1/zookeeper-3.4.6/bin/./src/java/lib/*.jar:/data/test1/zookeeper-
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:java.
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:java.
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:java.
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:os.na
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:os.ar
2015-12-10 10:26:46,945 [myid:] - INFO [main:Environment@100] - Client environment:os.ve
2015-12-10 10:26:46,946 [myid:] - INFO [main:Environment@100] - Client environment:user.
2015-12-10 10:26:46,946 [myid:] - INFO [main:Environment@100] - Client environment:user.
2015-12-10 10:26:46,946 [myid:] - INFO [main:Environment@100] - Client environment:user.
2015-12-10 10:26:46,947 [myid:] - INFO [main:ZooKeeper@438] - Initiating client connecti
erMain$MyWatcher@ee01430
Welcome to ZooKeeper!
2015-12-10 10:26:46,984 [myid:] - INFO [main-SendThread(localhost:2181):ClientCnxn$SendT
t to authenticate using SASL (unknown error)
JLine support is enabled
2015-12-10 10:26:46,989 [myid:] - INFO [main-SendThread(localhost:2181):ClientCnxn$SendT
[zk: localhost:2181(CONNECTING) 0] 2015-12-10 10:26:47,035 [myid:] - INFO [main-SendThre
localhost/127.0.0.1:2181, sessionId = 0x15189b57a8d0000, negotiated timeout = 30000
```

4、初始化 ZK 数据

sh create_zookeeper_data.sh

等待执行结束后，检查 ZK 数据

5、检查 ZK 数据

```
[zk: localhost:2181(CONNECTED) 1] ls /mycat
[mycat-mysqls, mycat-zones, mycat-nodes, mycat-hosts, mycat-cluster, mycat-mysqlgroup]
[zk: localhost:2181(CONNECTED) 2]
```

OK,数据初始化成功。

4.6 demo 使用

springMVC+ibatis+FreeMarker 连接 mycat 示例:

<http://pan.baidu.com/s/1qWr4AF6>

第 5 章日志分析

mycat 的日志文件配置为 MYCAT_HOME/conf/log4j.xml,结构为:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="ConsoleAppender" class="org.apache.log4j.ConsoleAppender">

    <layout class="org.apache.log4j.PatternLayout">

      <param name="ConversionPattern" value="%d{MM-dd HH:mm:ss.SSS} %5p [%t] (%F:%L) -%m%n" />

    </layout>

  </appender>

  <appender name="FILE" class="org.apache.log4j.RollingFileAppender">

    <param name="file" value="${MYCAT_HOME}/logs/mycat.log" />

    <param name="Append" value="false"/>

    <param name="MaxFileSize" value="10000KB"/>

    <param name="MaxBackupIndex" value="10"/>

    <param name="encoding" value="UTF-8" />

    <layout class="org.apache.log4j.PatternLayout">

      <param name="ConversionPattern" value="%d{MM/dd HH:mm:ss.SSS} %5p [%t] (%F:%L) -%m%n" />

    </layout>

  </appender>

  <root>

    <level value="debug" />

    <appender-ref ref="ConsoleAppender" />

  </root>

</log4j:configuration>
```

日志配置是标准的 log4j 配置, 其中:

```
<param name="file" value="${MYCAT_HOME}/logs/mycat.log" />
```

是日志文件的存放目录。

```
<root>
```

```
<level value="debug" />
```

```
<appender-ref ref="ConsoleAppender" />
```

```
</root>
```

是日志的级别，生成环境下建议将级别调整为 info/ware,如果是研究测试，特别是碰到异常可以通过开启 debug 模式观察日志的信息查找异常原因。

5.1 warpper 日志:

目前 Mycat 的启动是经过 warapper 封装成启动脚本，所以日志也会有其相关的日志文件：

`${MYCAT_HOME}/logs/warapper.log`,再启动时候如果系统环境配置错误或缺少配置时，导致 Mycat 无法启动，可以通过查看 `warrpper.log` 查看具体错误原因。

正常启动状态的 warpper 日志为：

```
STATUS | wrapper | 2015/04/12 15:05:00 | --> Wrapper Started as Daemon
```

```
STATUS | wrapper | 2015/04/12 15:05:00 | Launching a JVM...
```

```
INFO | jvm 1 | 2015/04/12 15:05:01 | Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
```

```
INFO | jvm 1 | 2015/04/12 15:05:01 | Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.
```

```
INFO | jvm 1 | 2015/04/12 15:05:01 |
```

```
INFO | jvm 1 | 2015/04/12 15:05:01 | log4j 2015-04-12 15:05:01 [./conf/log4j.xml] load completed.
```

```
INFO | jvm 1 | 2015/04/12 15:05:02 | MyCAT Server startup successfully. see logs in logs/mycat.log
```

如果启动异常会有对应的异常信息，比如：

```
STATUS | wrapper | 2015/02/14 01:43:44 | --> Wrapper Started as Daemon
```

```
STATUS | wrapper | 2015/02/14 01:43:44 | Launching a JVM...
```

```
INFO | jvm 1 | 2015/02/14 01:43:45 | Error: Exception thrown by the agent : java.rmi.server.ExportException:  
Port already in use: 1984; nested exception is:
```

```
INFO | jvm 1 | 2015/02/14 01:43:45 | java.net.BindException: Address already in use
```

```
ERROR | wrapper | 2015/02/14 01:43:45 | JVM exited while loading the application.
```

日志显示异常原因为 java.net.BindException: Address already in use,也就是端口占用,很有可能是原有服务未停止,或者 Mycat 默认端口被其他程序占用,正常启动成功后会有 mycat.log 日志,如果服务未启动成功不会有对应的日志。也可以去修改 conf 文件夹里的 wrapper.conf 里的 wrapper.java.additional.7=-Dcom.sun.management.jmxremote.port=1984, server.xml 的<property name="serverPort">8066</property>和<property name="managerPort">9066</property>, 这方法适合一台机器上两个 mycat 或者 1984,8066,9066 端口被其它应用占用的情况。

5.2 mycat 日志

下面看一下 info 级别小成功启动的日志。

```
04-29 21:46:59.121 INFO [main] (PhysicalDBPool.java:81) -total resouces of dataHost jdbcHost is :4
04-29 21:46:59.126 INFO [main] (PhysicalDBPool.java:81) -total resouces of dataHost jdbcHost2 is :4
04-29 21:46:59.143 INFO [main] (CacheService.java:125) -create layer cache pool TableID2DataNodeCache
of type encache ,default cache size 10000 ,default expire seconds18000
04-29 21:46:59.145 INFO [main] (DefaultLayedCachePool.java:80) -create child Cache: TESTDB_ORDERS for
layered cache TableID2DataNodeCache, size 50000, expire seconds 18000
04-29 21:46:59.472 INFO [main] (DynaClassLoader.java:35) -dyna class load from E:\MyProject\Mycat-
Server\main\catlet,and auto check for class file modified every 60 seconds
04-29 21:46:59.477 INFO [main] (MycatServer.java:192) -
=====
04-29 21:46:59.478 INFO [main] (MycatServer.java:193) -MyCat is ready to startup ...
04-29 21:46:59.478 INFO [main] (MycatServer.java:203) -Startup processors ...,total processors:4,aio thread
pool size:8
each process allocated socket buffer pool bytes ,buffer chunk size:4096 buffer pool's
capacity(buferPool/bufferChunk) is:4000
04-29 21:46:59.479 INFO [main] (MycatServer.java:204) -sysconfig params:SystemConfig
[processorBufferLocalPercent=100, frontSocketSoRcvbuf=1048576, frontSocketSoSndbuf=4194304,
backSocketSoRcvbuf=4194304, backSocketSoSndbuf=1048576, frontSocketNoDelay=1,
backSocketNoDelay=1, maxStringLiteralLength=65535, frontWriteQueueSize=2048, bindIp=0.0.0.0,
serverPort=8066, managerPort=9066, charset=utf8, processors=4, processorExecutor=8, timerExecutor=2,
```

```
managerExecutor=2, idleTimeout=1800000, catletClassCheckSeconds=60, sqlExecuteTimeout=300,
processorCheckPeriod=1000, dataNodeIdleCheckPeriod=300000, dataNodeHeartbeatPeriod=10000,
clusterHeartbeatUser=_HEARTBEAT_USER_, clusterHeartbeatPass=_HEARTBEAT_PASS_,
clusterHeartbeatPeriod=5000, clusterHeartbeatTimeout=10000, clusterHeartbeatRetry=10, txIsolation=3,
parserCommentVersion=50148, sqlRecordCount=10, processorBufferPool=16384000,
processorBufferChunk=4096, defaultMaxLimit=100, sequenceHandlerType=1, 04-29 21:47:01.343 INFO
[main] (PhysicalDBPool.java:296) -init result :finished 10 success 10 target count:10
04-29 21:47:01.343 INFO [main] (PhysicalDBPool.java:238) -jdbchost2 index:0 init success
MyCAT Server startup successfully. see logs in logs/mycat.log
04-29 21:51:21.846 INFO [main] (PhysicalDBPool.java:81) -total resouces of dataHost jdbchost is :4
04-29 21:51:21.848 INFO [main] (PhysicalDBPool.java:81) -total resouces of dataHost jdbchost2 is :4
```

该部分日志可以看到配置的数据源相关信息，上面是两个数据源连接 datahost

```
04-29 21:51:21.856 INFO [main] (CacheService.java:125) -create layer cache pool
TableID2DataNodeCache of type encache ,default cache size 10000 ,default expire seconds18000
04-29 21:51:21.857 INFO [main] (DefaultLayedCachePool.java:80) -create child Cache: TESTDB_ORDERS for
layered cache TableID2DataNodeCache, size 50000, expire seconds 18000
04-29 21:51:22.104 INFO [main] (DynaClassLoader.java:35) -dyna class load from E:\MyProject\Mycat-
Server\main\catlet,and auto check for class file modified every 60 seconds
```

该部分描述了 Mycat 的缓存信息及动态类加载信息。

```
04-29 21:51:22.107 INFO [main] (MycatServer.java:203) -Startup processors ...,total processors:4,aio
thread pool size:8
each process allocated socket buffer pool bytes ,buffer chunk size:4096 buffer pool's
capacity(buferPool/bufferChunk) is:4000
04-29 21:51:22.108 INFO [main] (MycatServer.java:204) -sysconfig params:SystemConfig
[processorBufferLocalPercent=100, frontSocketSoRcvbuf=1048576, frontSocketSoSndbuf=4194304,
backSocketSoRcvbuf=4194304, backSocketSoSndbuf=1048576, frontSocketNoDelay=1,
backSocketNoDelay=1, maxStringLiteralLength=65535, frontWriteQueueSize=2048, bindIp=0.0.0.0,
serverPort=8066, managerPort=9066, charset=utf8, processors=4, processorExecutor=8, timerExecutor=2,
managerExecutor=2, idleTimeout=1800000, catletClassCheckSeconds=60, sqlExecuteTimeout=300,
```

```
processorCheckPeriod=1000, dataNodeIdleCheckPeriod=300000, dataNodeHeartbeatPeriod=10000,
clusterHeartbeatUser=_HEARTBEAT_USER_, clusterHeartbeatPass=_HEARTBEAT_PASS_,
clusterHeartbeatPeriod=5000, clusterHeartbeatTimeout=10000, clusterHeartbeatRetry=10, txIsolation=3,
parserCommentVersion=50148, sqlRecordCount=10, processorBufferPool=16384000,
processorBufferChunk=4096, defaultMaxLimit=100, sequenceHandlerType=1,
sqlInterceptor=io.mycat.interceptor.impl.DefaultSqlInterceptor, sqlInterceptorType=select,
sqlInterceptorFile=E:\MyProject\Mycat-Server/logs/sql.txt, mutiNodeLimitType=0, mutiNodePatchSize=100,
defaultSqlParser=druidparser, usingAIO=0, packetHeaderSize=4, maxPacketSize=16777216, mycatNodeid=1]
04-29 21:51:22.131 INFO [main] (MycatServer.java:262) -using nio network handler
```

该部分描述了 Mycat 线程池、buffer、连接池等等所有的配置信息，通过该启动项可以得知当前运行的 Mycat 个参数调整情况，生产环境下需要做部分参数调整，可以根据该日志分析参数情况。

```
04-29 21:58:35.407 INFO [main] (MycatServer.java:280) -$_MyCatManager is started and listening on
9066
04-29 21:58:35.408 INFO [main] (MycatServer.java:284) -$_MyCatServer is started and listening on 8066
```

该部分描述了 Mycat 启动端口。

```
04-29 21:58:35.408 INFO [main] (MycatServer.java:289) -Initialize dataHost ...
04-29 21:58:35.408 INFO [main] (PhysicalDBPool.java:267) -init backend myqsl source ,create connections
total 10 for master index :0
04-29 21:58:35.410 INFO [main] (PhysicalDatasource.java:356) -not ilde connection in pool,create new
connection for masterConnectionMeta [schema=mycat_node1, charset=utf8, txIsolation=-1,
autocommit=true]
04-29 21:58:35.412 INFO [main] (PhysicalDatasource.java:356) -not ilde connection in pool,create new
connection for masterConnectionMeta connected successfully MySqlConnection [id=8,
lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89020, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-29 21:58:35.471 INFO [$ _NIOREACTOR-1-RW] (GetConnectionHandler.java:66) -connected successfully
MySqlConnection [id=9, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1,
```

```

borrowed=true, fromSlaveDB=false, threadId=89021, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-29 21:58:35.472 INFO [$_NIOREACTOR-2-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=10, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=89022, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-29 21:58:35.615 INFO [main] (PhysicalDBPool.java:296) -init result :finished 10 success 10 target count:10
04-29 21:58:35.615 INFO [main] (PhysicalDBPool.java:238) -jdbchost index:0 init success
04-29 21:58:35.615 INFO [main] (PhysicalDBPool.java:267) -init backend myqsl source ,create connections
total 10 for master index :0

```

该部分描述了 Mycat 时后端连接池的初始化过程。

如果某个连接断掉或异常心跳检测会有对应的日志如：

```

04-29 22:01:07.274 INFO [$_NIOConnector] (AbstractConnection.java:398) -close
connection,reason:heartbeat
connecterr ,[thread=$_NIOConnector,class=MySQLDetector,host=192.168.0.2,port=33061,localPort=0,schem
a=null]

```

该日志是心跳检测到连接异常关闭后端连接的日志，可以通过该日志查看后端数据连接状态。

5.3 debug 模式下分析 sql 执行。

下面分析 sql: select * from t_user t; 的执行

```

04-29 22:06:10.187 INFO [$_NIOREACTOR-3-RW] (FrontendAuthenticator.java:161) -ServerConnection [id=1,
schema=null, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=null]'mycat' login
success
04-29 22:06:10.188 DEBUG [$_NIOREACTOR-3-RW] (ServerQueryHandler.java:64) -ServerConnection [id=1,
schema=null, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=null]SET NAMES utf8
04-29 22:06:10.192 DEBUG [$_NIOREACTOR-3-RW] (ServerQueryHandler.java:64) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]SHOW STATUS

```

```

04-29 22:06:10.227 DEBUG [$_NIOREACTOR-3-RW] (NonBlockingSession.java:118) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]SHOW STATUS,
route={
    1 -> dn2{SHOW STATUS}
} rrs
04-29 22:06:10.228 DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost2
04-29 22:06:10.228 DEBUG [$_NIOREACTOR-3-RW] (MySQLConnection.java:437) -con need syn ,total syn
cmd 1 commands SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;schema change:false
con:MySQLConnection [id=13, lastTime=1430316370226, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=17188, charset=utf8, txIsolation=0, autocommit=true,
attachment=dn2{SHOW STATUS}, respHandler=SingleNodeHandler [node=dn2{SHOW STATUS},
packetId=0], host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.313 DEBUG [$_NIOREACTOR-3-RW] (ServerQueryHandler.java:64) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]select * from
t_user t
04-29 22:06:10.315 DEBUG [$_NIOREACTOR-3-RW] (EnchachePool.java:76) -SQLRouteCache miss
cache ,key:mycatselect * from t_user t
04-29 22:06:10.419 DEBUG [$_NIOREACTOR-3-RW] (EnchachePool.java:59) -SQLRouteCache add
cache ,key:mycatselect * from t_user t value:select * from t_user t, route={
    1 -> dn1{SELECT *
FROM t_user t
LIMIT 100}
    2 -> dn2{SELECT *
FROM t_user t
LIMIT 100}
}
04-29 22:06:10.420 DEBUG [$_NIOREACTOR-3-RW] (NonBlockingSession.java:118) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]select * from

```



```

t_user t, route={
    1 -> dn1{SELECT *
FROM t_user t
LIMIT 100}
    2 -> dn2{SELECT *
FROM t_user t
LIMIT 100}
} rrs

04-29 22:06:10.420  DEBUG [$_NIOREACTOR-3-RW] (MultiNodeQueryHandler.java:78) -execute mutinode
query select * from t_user t

04-29 22:06:10.422  DEBUG [$_NIOREACTOR-3-RW] (MultiNodeQueryHandler.java:93) -has data merge logic
04-29 22:06:10.422  DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost

04-29 22:06:10.422  DEBUG [$_NIOREACTOR-3-RW] (MySQLConnection.java:437) -con need syn ,total syn
cmd 1 commands SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;schema change:false
con:MySQLConnection [id=1, lastTime=1430316370409, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=89067, charset=utf8, txIsolation=0, autocommit=true,
attachment=dn1{SELECT *
FROM t_user t
LIMIT 100}, respHandler=io.mycat.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]

04-29 22:06:10.423  DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost2

04-29 22:06:10.423  DEBUG [$_NIOREACTOR-3-RW] (MySQLConnection.java:437) -con need syn ,total syn
cmd 1 commands SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;schema change:false
con:MySQLConnection [id=11, lastTime=1430316370409, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=17189, charset=utf8, txIsolation=0, autocommit=true,
attachment=dn2{SELECT *
FROM t_user t

```

```

LIMIT 100}, respHandler=io.mycat.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.432  DEBUG [$_NIOREACTOR-1-RW] (MultiNodeQueryHandler.java:165) -received ok
response ,executeResponse:false from MySQLConnection [id=1, lastTime=1430316370409,
schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false, threadId=89067,
charset=utf8, txIsolation=3, autocommit=true, attachment=dn1{SELECT *
FROM t_user t
LIMIT 100}, respHandler=io.mycat.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=121.40.121.133, port=3306, statusSync=io.mycat.mysql.nio.MySQLConnection$StatusSync@7485fef2,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.434  DEBUG [$_NIOREACTOR-1-RW] (DataMergeService.java:138) -field metadata
inf:[RECEIVE_ADDRESS=ColMeta [colIndex=1, colType=253],

```

通过该日志可以看到 Mycat 整个执行的计划。

其中最重要的是 sql 路由的计划，可以看到 sql 具体被分配到那个分片执行：

```

04-29 22:06:10.420  DEBUG [$_NIOREACTOR-3-RW] (NonBlockingSession.java:118) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]select * from
t_user t, route={
    1 -> dn1{SELECT *
FROM t_user t
LIMIT 100}
    2 -> dn2{SELECT *
FROM t_user t
LIMIT 100}
} rrs
04-29 22:06:10.420  DEBUG [$_NIOREACTOR-3-RW] (MultiNodeQueryHandler.java:78) -execute mutinode
query select * from t_user t

```

该部分描述了该条 sql 被分配到到了分片 dn1、dn2 上同时执行，如果某个某个 sql 通过缓存、分片规则或者注解指定只会在某个分片执行，则 sql 只会被分配到到某个分片，例如：

sql=select * from t_user t where t.user_id=121;该条数据只在分片 1 上。

```
04-29 22:13:40.960  DEBUG [$_NIOREACTOR-3-RW] (NonBlockingSession.java:118) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]select * from
t_user t where t.user_id=121, route={
    1 -> dn1{SELECT *
FROM t_user t
WHERE t.user_id = 121
LIMIT 100}
} rrs
```

从日志可以看出 sql 只被路由到 dn1 节点执行。

5.4 异常日志

```
java.sql.SQLException: com.alibaba.druid.sql.parser.ParserException: syntax error, error
in :'elect * from t_user t where t.',expect IDENTIFIER, actual IDENTIFIER elect
    at
io.mycat.route.impl.DruidMycatRouteStrategy.routeNormalSqlWithAST(DruidMycatRouteStrategy.java:44)
    at io.mycat.route.impl.AbstractRouteStrategy.route(AbstractRouteStrategy.java:52)
    at io.mycat.route.RouteService.route(RouteService.java:118)
    at io.mycat.server.ServerConnection.routeEndExecuteSQL(ServerConnection.java:165)
    at io.mycat.server.ServerConnection.execute(ServerConnection.java:154)
    at io.mycat.server.ServerQueryHandler.query(ServerQueryHandler.java:125)
    at io.mycat.net.FrontendConnection.query(FrontendConnection.java:250)
    at io.mycat.net.handler.FrontendCommandHandler.handle(FrontendCommandHandler.java:56)
    at io.mycat.net.FrontendConnection.handle(FrontendConnection.java:357)
    at io.mycat.net.AbstractConnection.onReadData(AbstractConnection.java:276)
    at io.mycat.net.NIOSocketWR.asyncRead(NIOSocketWR.java:186)
    at io.mycat.net.AbstractConnection.asyncRead(AbstractConnection.java:238)
    at io.mycat.net.NIOReactor$RW.run(NIOReactor.java:97)
    at java.lang.Thread.run(Thread.java:745)
```

```
Caused by: com.alibaba.druid.sql.parser.ParserException: syntax error, error in : 'select * from t_user t where  
t.',expect IDENTIFIER, actual IDENTIFIER select  
    at com.alibaba.druid.sql.parser.SQLParser.printError(SQLParser.java:229)  
    at  
com.alibaba.druid.sql.parser.SQLStatementParser.parseStatementList(SQLStatementParser.java:325)  
    at com.alibaba.druid.sql.parser.SQLStatementParser.parseStatement(SQLStatementParser.java:1655)  
    at  
io.mycat.route.impl.DruidMycatRouteStrategy.routeNormalSqlWithAST(DruidMycatRouteStrategy.java:41)  
    ... 13 more
```

如上面日志异常原因为 sql 错误导致 sql 解析器无法解析 sql，通过分析错误日志可以找到具体的出错原因。

Mycat 日志很重要，当发现 SQL 执行有异常的时候，大多数情况下，都可以通过分析 Mycat 日志来定位错误，当发现 Bug 存在的时候，也建议把相关日志信息附上，一并提交 github issue。

第 6 章 Mycat 防火墙配置

白名单和 SQL 黑名单说明：

在 server.xml 中配置：

```
<firewall>

    <whitehost>

        <host user="mycat" host="127.0.0.1"></host>  ip 白名单 用户对应的可以访问的 ip 地址

    </whitehost>

    <blacklist check="true">

        <property name="selectAllow">false</property> 黑名单允许的 权限 后面为默认

    </blacklist>

</firewall>
```

黑名单拦截明细配置

配置项	缺省值	描述
selectAllow	true	是否允许执行 SELECT 语句
selectAllColumnAllow	true	是否允许执行 SELECT * FROM T 这样的语句。如果设置为 false，不允许执行 select * from t，但 select * from (select id, name from t) a。这个选项是防御程序通过调用 select * 获得数据表的结构信息。
selectIntoAllow	true	SELECT 查询中是否允许 INTO 字句
deleteAllow	true	是否允许执行 DELETE 语句
updateAllow	true	是否允许执行 UPDATE 语句
insertAllow	true	是否允许执行 INSERT 语句
replaceAllow	true	是否允许执行 REPLACE 语句
mergeAllow	true	是否允许执行 MERGE 语句，这个只在 Oracle 中有用
callAllow	true	是否允许通过 jdbc 的 call 语法调用存储过程
setAllow	true	是否允许使用 SET 语法

truncateAllow	true	truncate 语句是危险，缺省打开，若需要自行关闭
createTableAllow	true	是否允许创建表
alterTableAllow	true	是否允许执行 Alter Table 语句
dropTableAllow	true	是否允许修改表
commentAllow	false	是否允许语句中存在注释，Oracle 的用户不用担心，Wall 能够识别 hints 和注释的区别
noneBaseStatementAllow	false	是否允许非以上基本语句的其他语句，缺省关闭，通过这个选项就能够屏蔽 DDL。
multiStatementAllow	false	是否允许一次执行多条语句，缺省关闭
useAllow	true	是否允许执行 mysql 的 use 语句，缺省打开
describeAllow	true	是否允许执行 mysql 的 describe 语句，缺省打开
showAllow	true	是否允许执行 mysql 的 show 语句，缺省打开
commitAllow	true	是否允许执行 commit 操作
rollbackAllow	true	是否允许执行 roll back 操作
如果把 selectIntoAllow、deleteAllow、updateAllow、insertAllow、mergeAllow 都设置为 false，这就是一个只读数据源了。		

拦截配置 - 永真条件

配置项	缺省值	描述
selectWhereAlwayTrueCheck	true	检查 SELECT 语句的 WHERE 子句是否是一个永真条件
selectHavingAlwayTrueCheck	true	检查 SELECT 语句的 HAVING 子句是否是一个永真条件
deleteWhereAlwayTrueCheck	true	检查 DELETE 语句的 WHERE 子句是否是一个永真条件
deleteWhereNoneCheck	false	检查 DELETE 语句是否无 where 条件，这是有风险的，但不是 SQL 注入类型的风险
updateWhereAlayTrueCheck	true	检查 UPDATE 语句的 WHERE 子句是否是一个永真条件

updateWhereNoneCheck	false	检查 UPDATE 语句是否无 where 条件，这是有风险的，但不是 SQL 注入类型的风险
conditionAndAlwayTrueAllow	false	检查查询条件(WHERE/HAVING 子句)中是否包含 AND 永真条件
conditionAndAlwayFalseAllow	false	检查查询条件(WHERE/HAVING 子句)中是否包含 AND 永假条件
conditionLikeTrueAllow	true	检查查询条件(WHERE/HAVING 子句)中是否包含 LIKE 永真条件
其他拦截配置		
配置项	缺省值	描述
selectIntoOutfileAllow	false	SELECT ... INTO OUTFILE 是否允许，这个是 mysql 注入攻击的常见手段，缺省是禁止的
selectUnionCheck	true	检测 SELECT UNION
selectMinusCheck	true	检测 SELECT MINUS
selectExceptCheck	true	检测 SELECT EXCEPT
selectIntersectCheck	true	检测 SELECT INTERSECT
mustParameterized	false	是否必须参数化，如果为 True，则不允许类似 WHERE ID = 1 这种不参数化的 SQL
strictSyntaxCheck	true	是否进行严格的语法检测，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，可以临时把这个选项设置为 false，同时把 SQL 反馈给 Druid 的开发者。
conditionOpXorAllow	false	查询条件中是否允许有 XOR 条件。XOR 不常用，很难判断永真或者永假，缺省不允许。
conditionOpBitwiseAllow	true	查询条件中是否允许有"&"、"~"、" "、"^"运算符。
conditionDoubleConstAllow	false	查询条件中是否允许连续两个常量运算表达式
minusAllow	true	是否允许 SELECT * FROM A MINUS SELECT * FROM B 这样的语句
intersectAllow	true	是否允许 SELECT * FROM A INTERSECT SELECT * FROM B 这样的语句
constArithmeticAllow	true	拦截常量运算的条件，比如说 WHERE FID = 3 - 1，其中"3 - 1"是常量运算表达式。

limitZeroAllow false 是否允许 limit 0 这样的语句

禁用对象检测配置

配置项	缺省值	描述
tableCheck	true	检测是否使用了禁用的表
schemaCheck	true	检测是否使用了禁用的 Schema
functionCheck	true	检测是否使用了禁用的函数
objectCheck	true	检测是否使用了“禁用对对象”
variantCheck	true	检测是否使用了“禁用的变量”

readOnlyTables	空	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

配置 demo

新增 IP 限制登陆功能，与 mysql 账号授权指定 IP 段类似，下面介绍用法。在 server.xml 添加信任 IP /SQL 黑名单

示例：

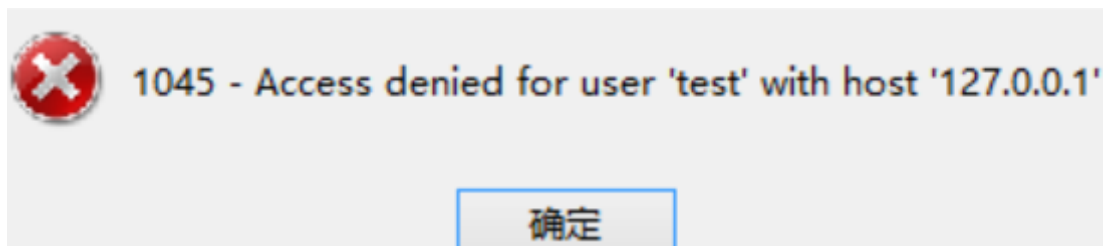
只允许 xx IP 的主机登陆，并且不允许执行 SELECT *操作

```
<firewall>
  <whitehost>
    <host host="自己指定 IP" user="test"></host>
  </whitehost>
  <blacklist check="true">
    <property name="selectAllColumnAllow">false</property>
  </blacklist>
</firewall>
```

使用指定主机登陆，执行 SELECT *，被限制执行


```
mysql> SELECT * FROM tparbyday1;  
ERROR 3012 (HY000): 'SELECT *' not allow  
mysql> █
```

使用主机列表以外 IP 登陆，被限制登陆



第 7 章 Mycat 的配置

7.1 搞定 schema.xml

Schema.xml 作为 MyCat 中重要的配置文件之一，管理着 MyCat 的逻辑库、表、分片规则、DataNode 以及 DataSource。弄懂这些配置，是正确使用 MyCat 的前提。这里就一层层对该文件进行解析。

7.2 schema 标签

```
<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100"></schema>
```

schema 标签用于定义 MyCat 实例中的逻辑库，MyCat 可以有多个逻辑库，每个逻辑库都有自己的相关配置。可以使用 schema 标签来划分这些不同的逻辑库。

如果不配置 schema 标签，所有的表配置，会属于同一个默认的逻辑库。

```
<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">
<table name="travelrecord" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" ></table>
</schema>

<schema name="USERDB" checkSQLschema="false" sqlMaxLimit="100">
<table name="company" dataNode="dn10,dn11,dn12" rule="auto-sharding-long" ></table>
</schema>
```

如上所示的配置就配置了两个不同的逻辑库，逻辑库的概念和 MYSQL 数据库中 Database 的概念相同，我们在查询这两个不同的逻辑库中表的时候需要切换到该逻辑库下才可以查询到所需要的表。

```
mysql> use USERDB;
Database changed
mysql> select * from company;
Empty set (0.09 sec)

mysql> select * from travelrecord;
ERROR 1064 (HY000): can't find table define in schema ,table:TRAUDELRECORD schema
:USERDB
mysql>
```

如果你发现显示该错误信息，需要到 server.xml 添加该用户可以访问到的 schema 就可以了。具体的内容待后续章节阐述。

```
mysql> use USERDB;
No connection. Trying to reconnect...
Connection id: 3
Current database: *** NONE ***

ERROR 1044 (HY000): Access denied for user 'test' to database 'USERDB'
mysql>
```

```

17      <!--默认是65535 64K 用于sql解析时最大文本长度-->
18      <!--<property name="maxStringLength">65535</property>-->
19      <!--<property name="sequenceHandlerType">0</property>-->
20      <!--<property name="backSocketNoDelay">1</property>-->
21      <!--<property name="frontSocketNoDelay">1</property>-->
22      <!--<property name="processorExecutor">16</property>-->
23      <!--
24          <property name="mutiNodeLimitType">1</property>
25          <property name="mutiNodePatchSize">100</property>
26          <property name="processors">32</property> <property name="serverPort">8066</property> <property name="idleTimeout">300000</property>
27          <property name="serverPort">8066</property> <property name="idleTimeout">300000</property>
28          <property name="idleTimeout">300000</property>
29          <property name="frontWriteQueueSize">4096</property>
30      </system>
31      <user name="test">
32          <property name="password">test</property>
33          <property name="schemas">TESTDB,USERDB</property>
34      </user>
35
36      <user name="user">
37          <property name="password">user</property>
38          <property name="schemas">TESTDB</property>
39          <property name="readOnly">true</property>
40      </user>
41      <!-- <cluster> <node name="cobari1"> <property name="host">127.0.0.1</property> <property name="weight">1</property> </node> </cluster> -->
42      <!-- <quarantine> <host name="1.2.3.4"> <property name="weight">1</property> </host> </quarantine> -->
43      <!-- <quarantine> <host name="1.2.3.4"> <property name="weight">1</property> </host> </quarantine> -->
44      <!-- <quarantine> <host name="1.2.3.4"> <property name="weight">1</property> </host> </quarantine> -->
45
46  </mycat:server>
47

```

schema 标签的相关属性:

属性名	值	数量限制
dataNode	任意 String	(0..1)
checkSQLschema	Boolean	(1)
sqlMaxLimit	Integer	(1)

7.2.1 dataNode

该属性用于绑定逻辑库到某个具体的 database 上, 1.3 版本如果配置了 dataNode, 则不可以配置分片表, 1.4 可以配置默认分片, 只需要配置需要分片的表即可, 具体如下配置:

1.3 配置:

```
<schema name="USERDB" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn1">  
    <!--里面不能配置任何表-->  
  
</schema>
```

1.4 配置:

```
<schema name="USERDB" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn2">  
    <!--配置需要分片的表-->  
  
    <table name="tuser" dataNode="dn1" />  
  
</schema>
```

那么现在 tuser 就绑定到 dn1 所配置的具体 database 上, 可以直接访问这个 database, 没有配置的表则会走默认节点 dn2, 这里注意没有配置在分片里面的表工具查看无法显示, 但是可以正常使用。

7.2.2 checkSQLschema

当该值设置为 true 时, 如果我们执行语句 `**select * from TESTDB.travelrecord;**` 则 MyCat 会把语句修改为 `**select * from travelrecord;**`。即把表示 schema 的字符去掉, 避免发送到后端数据库执行时报 `** (ERROR 1146 (42S02): Table 'testdb.travelrecord' doesn't exist) . **`

不过, 即使设置该值为 true, 如果语句所带的是并非是 schema 指定的名字, 例如: `**select * from db1.travelrecord;**` 那么 MyCat 并不会删除 db1 这个字段, 如果没有定义该库的话则会报错, 所以在提供 SQL 语句的最好是不带这个字段。

7.2.3 sqlMaxLimit

当该值设置为某个数值时。每条执行的 SQL 语句, 如果没有加上 limit 语句, MyCat 也会自动的加上所对应的值。例如设置值为 100, 执行 `**select * from TESTDB.travelrecord;**` 的效果为和执行 `**select * from TESTDB.travelrecord limit 100;**` 相同。

设置该值的话, MyCat 默认会把查询到的信息全部都展示出来, 造成过多的输出。所以, 在正常使用中, 还是建议加上一个值, 用于减少过多的数据返回。

当然 SQL 语句中也显式的指定 limit 的大小，不受该属性的约束。

需要注意的是，如果运行的 schema 为非拆分库的，那么该属性不会生效。需要手动添加 limit 语句。

7.3 table 标签

```
<table name="travelrecord" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" ></table>
```

Table 标签定义了 MyCat 中的逻辑表，所有需要拆分的表都需要在这个标签中定义。

table 标签的相关属性：

属性名	值	数量限制
name	String	(1)
dataNode	String	(1..*)
rule	String	(0..1)
ruleRequired	boolean	(0..1)
primaryKey	String	(1)
type	String	(0..1)
autoIncrement	boolean	(0..1)
subTables	String	(1)
needAddLimit	boolean	(0..1)

7.3.1 name 属性

定义逻辑表的表名，这个名字就如同我在数据库中执行 create table 命令指定的名字一样，同个 schema 标签中定义的名字必须唯一。

7.3.2 dataNode 属性

定义这个逻辑表所属的 dataNode, 该属性的值需要和 dataNode 标签中 name 属性的值相互对应。如果需要定义的 dn 过多 可以使用如下的方法减少配置：

```
<table name="travelrecord" dataNode="multipleDn$0-99,multipleDn2$100-199" rule="auto-sharding-long" ></table>
```

```
<dataNode name="multipleDn$0-99" dataHost="localhost1" database="db$0-99" ></dataNode>
<dataNode name="multipleDn2$100-199" dataHost="localhost1" database=" db$100-199" ></dataNode>
```

这里需要注意的是 database 属性所指定的真实 database name 需要在后面添加一个，例如上面的例子中，我需要在真实的 mysql 上建立名称为 dbs0 到 dbs99 的 database。

7.3.3 rule 属性

该属性用于指定逻辑表要使用的规则名字，规则名字在 rule.xml 中定义，必须与 tableRule 标签中 name 属性属性值一一对应。

7.3.4 ruleRequired 属性

该属性用于指定表是否绑定分片规则，如果配置为 true，但没有配置具体 rule 的话，程序会报错。

7.3.5 primaryKey 属性

该逻辑表对应真实表的主键，例如：分片的规则是使用非主键进行分片的，那么在使用主键查询的时候，就会发送查询语句到所有配置的 DN 上，如果使用该属性配置真实表的主键。那么 MyCat 会缓存主键与具体 DN 的信息，那么再次使用非主键进行查询的时候就不会进行广播式的查询，就会直接发送语句给具体的 DN，但是尽管配置该属性，如果缓存并没有命中的话，还是会发送语句给具体的 DN，来获得数据。

7.3.6 type 属性

该属性定义了逻辑表的类型，目前逻辑表只有“全局表”和“普通表”两种类型。对应的配置：

- 全局表：global。
- 普通表：不指定该值为 global 的所有表。

7.3.7 autoIncrement 属性

mysql 对非自增长主键，使用 last_insert_id() 是不会返回结果的，只会返回 0。所以，只有定义了自增长主键的表才可以用 last_insert_id() 返回主键值。

mycat 目前提供了自增长主键功能，但是如果对应的 mysql 节点上数据表，没有定义 auto_increment，那么在 mycat 层调用 last_insert_id() 也是不会返回结果的。

由于 insert 操作的时候没有带入分片键，mycat 会先取下这个表对应的全局序列，然后赋值给分片键。这样才能正常的插入到数据库中，最后使用 last_insert_id() 才会返回插入的分片键值。

如果要使用这个功能最好配合使用数据库模式的全局序列。

使用 autoIncrement=“true” 指定这个表有使用自增长主键，这样 mycat 才会不抛出分片键找不到的异常。

使用 `autoIncrement="false"` 来禁用这个功能，当然你也可以直接删除掉这个属性。默认就是禁用的。

7.3.8 subTables

使用方式添加 `subTables="t_order$1-2,t_order3"`。

目前分表 1.6 以后开始支持 并且 `dataNode` 在分表条件下只能配置一个，分表条件下不支持各种条件的 `join` 语句。

7.3.9 needAddLimit 属性

指定表是否需要自动的在每个语句后面加上 `limit` 限制。由于使用了分库分表，数据量有时会特别巨大。这时候执行查询语句，如果恰巧又忘记了加上数量限制的话。那么查询所有的数据出来，也够等上一小会儿的。

所以，mycat 就自动的为我们加上 `LIMIT 100`。当然，如果语句中有 `limit`，就不会在次添加了。

这个属性默认为 `true`，你也可以设置成 `false` 禁用掉默认行为。

7.4 childTable 标签

`childTable` 标签用于定义 E-R 分片的子表。通过标签上的属性与父表进行关联。

childTable 标签的相关属性：

属性名	值	数量限制
name	String	(1)
joinKey	String	(1)
parentKey	String	(1)
primaryKey	String	(0..1)
needAddLimit	boolean	(0..1)

7.4.1 name 属性

定义子表的表名。

7.4.2 joinKey 属性

插入子表的时候会使用这个列的值查找父表存储的数据节点。

7.4.3 parentKey 属性

属性指定的值一般为与父表建立关联关系的列名。程序首先获取 joinkey 的值，再通过 **parentKey** 属性指定的列名产生查询语句，通过执行该语句得到父表存储在哪个分片上。从而确定子表存储的位置。

7.4.4 primaryKey 属性

同 table 标签所描述的。

7.4.5 needAddLimit 属性

同 table 标签所描述的。

7.5 dataNode 标签

```
<dataNode name="dn1" dataHost="lch3307" database="db1" ></dataNode>
```

dataNode 标签定义了 MyCat 中的数据节点，也就是我们通常所说的数据分片。一个 **dataNode** 标签就是一个独立的数据分片。

例子中所表述的意思为：使用名字为 lch3307 数据库实例上的 db1 物理数据库，这就组成一个数据分片，最后，我们使用名字 dn1 标识这个分片。

dataNode 标签的相关属性：

属性名	值	数量限制
name	String	(1)
dataHost	String	(1)
database	String	(1)

7.5.1 name 属性

定义数据节点的名字，这个名字需要是唯一的，我们需要在 table 标签上应用这个名字，来建立表与分片对应的关系。

7.5.2 dataHost 属性

该属性用于定义该分片属于哪个数据库实例的，属性值是引用 dataHost 标签上定义的名称属性。

7.5.3 database 属性

该属性用于定义该分片属于哪个具体数据库实例上的具体库，因为这里使用两个维度来定义分片，就是：实例+具体的库。因为每个库上建立的表和表结构是一样的。所以这样做就可以轻松的对表进行水平拆分。

7.6 dataHost 标签

作为 Schema.xml 中最后的一个标签，该标签在 mycat 逻辑库中也是作为最底层的标签存在，直接定义了具体的数据库实例、读写分离配置和心跳语句。现在我们就解析下这个标签。

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="0"
writeType="0" dbType="mysql" dbDriver="native">
<heartbeat>select user()</heartbeat>
<!-- can have multi write hosts -->
<writeHost host="hostM1" url="localhost:3306" user="root"
password="123456">
<!-- can have multi read hosts -->
<!-- <readHost host="hostS1" url="localhost:3306" user="root" password="123456"
/> -->
</writeHost>
<!-- <writeHost host="hostM2" url="localhost:3316" user="root" password="123456"/> -->
</dataHost>
```

dataHost 标签的相关属性：

属性名	值	数量限制
name	String	(1)
maxCon	Integer	(1)
minCon	Integer	(1)
balance	Integer	(1)
writeType	Integer	(1)
dbType	String	(1)
dbDriver	String	(1)

7.6.1 name 属性

唯一标识 dataHost 标签，供上层的标签使用。

7.6.2 maxCon 属性

指定每个读写实例连接池的最大连接。也就是说，标签内嵌套的 writeHost、readHost 标签都会使用这个属性的值来实例化出连接池的最大连接数。

7.6.3 minCon 属性

指定每个读写实例连接池的最小连接，初始化连接池的大小。

7.6.4 balance 属性

负载均衡类型，目前的取值有 3 种：

1. balance="0", 不开启读写分离机制，所有读操作都发送到当前可用的 writeHost 上。
2. balance="1", 全部的 readHost 与 stand by writeHost 参与 select 语句的负载均衡，简单的说，当双主双从模式(M1->S1, M2->S2, 并且 M1 与 M2 互为备), 正常情况下, M2,S1,S2 都参与 select 语句的负载均衡。
3. balance="2", 所有读操作都随机的在 writeHost、readhost 上分发。
4. balance="3", 所有读请求随机的分发到 writerHost 对应的 readhost 执行, writerHost 不负担读压力, 注意 balance=3 只在 1.4 及其以后版本有, 1.3 没有。

7.6.5 writeType 属性

负载均衡类型，目前的取值有 3 种：

1. writeType="0", 所有写操作发送到配置的第一个 writeHost, 第一个挂了切到还生存的第二个 writeHost, 重新启动后已切换后的为准, 切换记录在配置文件中:dnindex.properties .

2. writeType="1", 所有写操作都随机的发送到配置的 writeHost, 1.5 以后废弃不推荐。**switchType 属性**

- -1 表示不自动切换。
- 1 默认值, 自动切换。
- 2 基于 MySQL 主从同步的状态决定是否切换。

7.6.6 dbType 属性

指定后端连接的数据库类型，目前支持二进制的 mysql 协议，还有其他使用 JDBC 连接的数据库。例如：mongodb、oracle、spark 等。

7.6.7 dbDriver 属性

指定连接后端数据库使用的 Driver，目前可选的值有 native 和 JDBC。使用 native 的话，因为这个值执行的是二进制的 mysql 协议，所以可以使用 mysql 和 maridb。其他类型的数据库则需要使用 JDBC 驱动来支持。

从 1.6 版本开始支持 postgresql 的 native 原始协议。

如果使用 JDBC 的话需要将符合 JDBC 4 标准的驱动 JAR 包放到 MYCAT\lib 目录下，并检查驱动 JAR 包中包括如下目录结构的文件：META-INF\services\java.sql.Driver。在这个文件内写上具体的 Driver 类名，例如：
com.mysql.jdbc.Driver。

7.6.8 switchType 属性

-1 表示不自动切换

1 默认值，自动切换

2 基于 MySQL 主从同步的状态决定是否切换

心跳语句为 show slave status

3 基于 MySQL galary cluster 的切换机制（适合集群）（1.4.1）

心跳语句为 show status like 'wsrep%'

7.6.9 tempReadHostAvailable 属性

如果配置了这个属性 writeHost 下面的 readHost 仍旧可用，默认 0 可配置（0、1）。

7.7 heartbeat 标签

这个标签内指明用于和后端数据库进行心跳检查的语句。例如,MYSQL 可以使用 select user(), Oracle 可以使用 select 1 from dual 等。

这个标签还有一个 connectionInitSql 属性，主要是当使用 Oracla 数据库时，需要执行的初始化 SQL 语句就这个放到这里面来。例如：alter session set nls_date_format='yyyy-mm-dd hh24:mi:ss'

1.4 主从切换的语句必须是：show slave status

7.7.1 writeHost 标签、readHost 标签

这两个标签都指定后端数据库的相关配置给 mycat，用于实例化后端连接池。唯一不同的是，writeHost 指定写实例、readHost 指定读实例，组着这些读写实例来满足系统的要求。

在一个 dataHost 内可以定义多个 writeHost 和 readHost。但是，如果 writeHost 指定的后端数据库宕机，那么这个 writeHost 绑定的所有 readHost 都将不可用。另一方面，由于这个 writeHost 宕机系统会自动检测到，并切换到备用的 writeHost 上去。

这两个标签的属性相同，这里就一起介绍。

属性名	值	数量限制
host	String	(1)
url	String	(1)
password	String	(1)
user	String	(1)
weight	String	(1)
usingDecrypt	String	(1)

7.7.2 host 属性

用于标识不同实例，一般 writeHost 我们使用*M1，readHost 我们用*S1。

7.7.3 url 属性

后端实例连接地址，如果是使用 native 的 dbDriver，则一般为 address:port 这种形式。用 JDBC 或其他 dbDriver，则需要特殊指定。当使用 JDBC 时则可以这么写：jdbc:mysql://localhost:3306/。

7.7.4 user 属性

后端存储实例需要的用户名字。

7.7.5 password 属性

后端存储实例需要的密码。

7.7.6 weight 属性

权重 配置在 readhost 中作为读节点的权重（1.4 以后）。

7.7.7 usingDecrypt 属性

是否对密码加密默认 0 否 如需要开启配置 1，同时使用加密程序对密码加密，加密命令为：

执行 mycat jar 程序（1.4.1 以后）：

```
java -cp Mycat-server-1.4.1-dev.jar io.mycat.util.DecryptUtil 1:host:user:password
```

Mycat-server-1.4.1-dev.jar 为 mycat download 下载目录的 jar

1:host:user:password 中 1 为 db 端加密标志，host 为 dataHost 的 host 名称

7.8 server.xml

7.8.1 配置

server.xml 几乎保存了所有 mycat 需要的系统配置信息。其在代码内直接的映射类为 SystemConfig 类。

7.8.2 user 标签

```
<user name="test">
  <property name="password">test</property>
  <property name="schemas">TESTDB</property>
  <property name="readOnly">true</property>
  <property name="benchmark">11111</property>
  <property name="usingDecrypt">1</property>

  <privileges check="false">

    <schema name="TESTDB" dml="0010" showTables="custome/mysql">

      <table name="tbl_user" dml="0110"></table>

      <table name="tbl_dynamic" dml="1111"></table>

    </schema>

  </privileges>
</user>
```

server.xml 中的标签本就不多，这个标签主要用于定义登录 mycat 的用户和权限。例如上面的例子中，我定义了一个用户，用户名为 test、密码也为 test，可访问的 schema 也只有 TESTDB 一个。

如果我在 schema.xml 中定义了多个 schema，那么这个用户是无法访问其他的 schema。在 mysql 客户端看来则是无法使用 use 切换到这个其他的数据库。

如果使用了 use 命令，则 mycat 会报出这样的错误提示：

```
ERROR 1044 (HY000): Access denied for user 'test' to database 'xxx'
```

这个标签嵌套的 property 标签则是具体声明的属性值，正如上面的例子。我们可以修改 user 标签的 name 属性来指定用户名；修改 password 内的文本来修改密码；修改 readOnly 为 true 或 false 来限制用户是否只是可读的；修改 schemas 内的文本来控制用户可访问的 schema；修改 schemas 内的文本来控制用户可访问的 schema，同时访问多个 schema 的话使用，隔开，例如：

```
<property name="schemas">TESTDB,db1,db2</property>
```

Benchmark 属性

Benchmark:mycat 连接服务降级处理：

benchmark 基准，当前端的整体 connection 数达到基准值是，对来自该账户的请求开始拒绝连接，0 或不设表示不限制

例如 <property name="benchmark">1000</property>

usingDecrypt 属性

是否对密码加密默认 0 否 如需要开启配置 1，同时使用加密程序对密码加密，加密命令为：

执行 mycat jar 程序：

```
java -cp Mycat-server-1.4.1-dev.jar io.mycat.util.DecryptUtil 0:user:password
```

Mycat-server-1.4.1-dev.jar 为 mycat download 下载目录的 jar

1:host:user:password 中 0 为前端加密标志

privileges 子节点

对用户的 schema 及 下级的 table 进行精细化的 DML 权限控制，privileges 节点中的 check 属性是用于标识是否开启 DML 权限检查，默认 false 标识不检查，当然 privileges 节点不配置，等同 check=false，由于 Mycat 一个用户的 schemas 属性可配置多个 schema，所以 privileges 的下级节点 schema 节点同样可配置多个，对多库多表进行细粒度的 DML 权限控制

Schema/Table 上的 dml 属性描述

参数	说明	事例（禁止增删改查）
dml	insert,update,select,delete	0000

注：设置了 schema，但只设置了个别 table 或 未设置 table 的 DML，自动继承 schema 的 DML 属性

privileges 配置事例如下：

```
<user name="zhuam">

    <property name="password">111111</property>

    <property name="schemas">TESTDB,TESTDB1</property>

    <!-- 表级权限: Table 级的 dml(curd)控制, 未设置的 Table 继承 schema 的 dml -->

    <!-- TODO: 非 CURD SQL 语句, 透明传递至后端 -->

    <privileges check="true">

        <schema name="TESTDB" dml="0110" >

            <table name="table01" dml="0111"></table>

            <table name="table02" dml="1111"></table>

        </schema>

        <schema name="TESTDB1" dml="0110">

            <table name="table03" dml="1110"></table>

            <table name="table04" dml="1010"></table>

        </schema>

    </privileges>

</user>
```

7.9 system 标签

这个标签内嵌套的所有 property 标签都与系统配置有关，请注意，下面我会省去标签 property 直接使用这个标签的 name 属性内的值来介绍这个属性的作用。

7.9.1 charset 属性

字符集设置。

配置属性 charset

```
<system> <property name="charset">utf8</property> </system>
```

如果需要配置 utf8mb2 等特殊字符集可以在

index_to_charset.properties 配置中

配置数据库端的字符集 ID=字符集

例如：

224=utf8mb4

配置字符集的时候一定要坚持 mycat 的字符集与数据库端的字符集是一致的，可以通过变量来查询：

```
show variables like 'collation_%';
```

```
show variables like 'character_set_%';
```

7.9.2 defaultSqlParser 属性

由于 mycat 最初是模仿 Foundation DB 的 sql 解析器，而后才添加的 Druid 的解析器。所以这个属性用来指定默认的解析器。目前的可用的取值有：druidparser 和 fdbparser。使用的时候可以选择其中的一种，目前一般都使用 druidparser。

1.3 解析器默认为 fdbparser，1.4 默认为 druidparser，1.4 以后 fdbparser 作废。

7.9.3 processors 属性

这个属性主要用于指定系统可用的线程数，默认值为机器 CPU 核心线程数。

主要影响 processorBufferPool、processorBufferLocalPercent、processorExecutor 属性。

NIOProcessor 的个数也是由这个属性定义的，所以调优的时候可以适当的调高这个属性。

7.9.4 processorBufferChunk 属性

这个属性指定每次分配 Socket Direct Buffer 的大小，默认是 4096 个字节。这个属性也影响 buffer pool 的长度。如果一次性获取的数过大 buffer 不够用 经常出现警告，则可以适当调大。

7.9.5 processorBufferPool 属性

这个属性指定 bufferPool 计算 **比例值**。由于每次执行 NIO 读、写操作都需要使用到 buffer，系统初始化的时候会建立一定长度的 buffer 池来加快读、写的效率，减少建立 buffer 的时间。

Mycat 中有两个主要的 buffer 池:

- BufferPool
- ThreadLocalPool

BufferPool 由 ThreadLocalPool 组合而成，每次从 BufferPool 中获取 buffer 都会优先获取 ThreadLocalPool 中的 buffer，未命中之后才会去获取 BufferPool 中的 buffer。也就是说 ThreadLocalPool 是作为 BufferPool 的二级缓存，每个线程内部自己使用的。当然，这其中还有一些限制条件需要线程的名字是由 \$_ 开头。然而，BufferPool 上的 buffer 则是每个 NIOProcessor 都共享的。

默认这个属性的值为：**默认 $\text{bufferChunkSize}(4096) * \text{processors 属性} * 1000$**

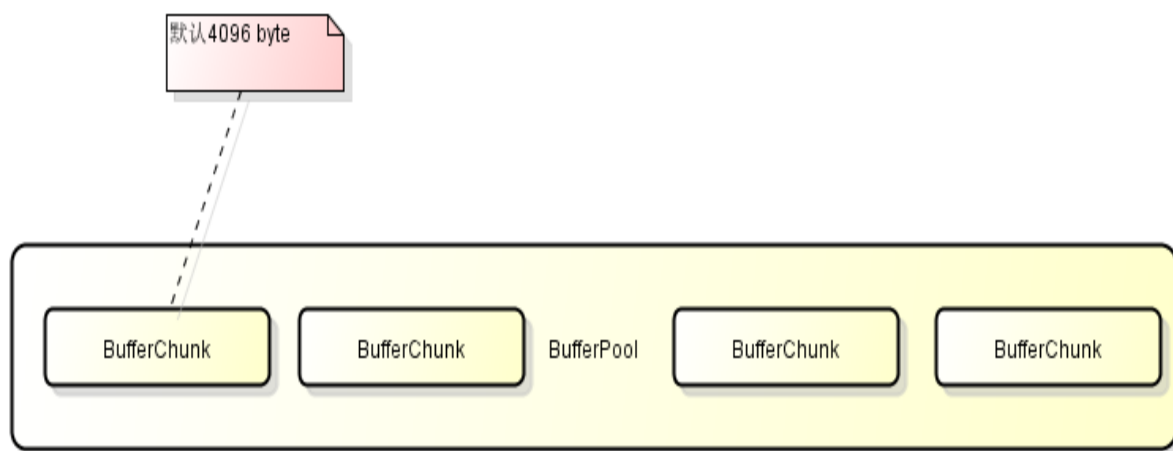
BufferPool 的总长度 = $\text{bufferPool} / \text{bufferChunk}$ 。

若 bufferPool 不是 bufferChunk 的整数倍，则总长度为前面计算得出的商 + 1

假设系统线程数为 4，其他都为属性的默认值，则：

$$\text{bufferPool} = 4096 * 4 * 1000$$

$$\text{BufferPool 的总长度} : 4000 = 16384000 / 4096$$



7.9.6 processorBufferLocalPercent 属性

前面提到了 ThreadLocalPool。这个属性就是用来控制分配这个 pool 的大小用的，但其也并不是一个准确的值，也是一个比例值。这个属性默认值为 100。

线程缓存百分比 = bufferLocalPercent / processors 属性。

例如，系统可以同时运行 4 个线程，使用默认值，则根据公式每个线程的百分比为 25。最后根据这个百分比来计算出具体的 ThreadLocalPool 的长度公式如下：

ThreadLocalPool 的长度 = 线程缓存百分比 * BufferPool 长度 / 100

假设 BufferPool 的长度为 4000，其他保持默认值。

那么最后每个线程建立上的 ThreadLocalPool 的长度为：1000 = 25 * 4000 / 100

7.9.7 processorExecutor 属性

这个属性主要用于指定 NIOProcessor 上共享的 businessExecutor 固定线程池大小。mycat 在处理一些异步逻辑的时候会把任务提交到这个线程池中。新版本中这个连接池的使用频率不是很大了，可以设置一个较小的值。

7.9.8 sequenceHandlerType 属性

指定使用 Mycat 全局序列的类型。0 为本地文件方式，1 为数据库方式，2 为时间戳序列方式，3 为分布式 ZK ID 生成器，4 为 zk 递增 id 生成。

从 1.6 增加两种 ZK 的全局 ID 生成算法。

7.9.9 TCP 连接相关属性

- StandardSocketOptions.SO_RCVBUF
- StandardSocketOptions.SO_SNDBUF
- StandardSocketOptions.TCP_NODELAY

以上这三个属性，分别由：

frontSocketSoRcvbuf 默认值：1024 * 1024

frontSocketSoSndbuf 默认值：4 * 1024 * 1024

frontSocketNoDelay 默认值：1

backSocketSoRcvbuf 默认值：4 * 1024 * 1024

backSocketSoSndbuf 默认值：1024 * 1024

backSocketNoDelay 默认值：1

各自设置前后端 TCP 连接参数。Mycat 在每次建立前、后端连接的时候都会使用这些参数初始化连接。可以按系统要求适当的调整这些 buffer 的大小。TCP 连接参数的定义，可以查看 Javadoc。

7.9.10 Mysql 连接相关属性

初始化 mysql 前后端连接所涉及的一些属性：

packetHeaderSize : 指定 Mysql 协议中的报文头长度。默认 4。

maxPacketSize : 指定 Mysql 协议可以携带的数据最大长度。默认 16M。

idleTimeout : 指定连接的空闲超时时间。某连接在发起空闲检查下，发现距离上次使用超过了空闲时间，那么这个连接会被回收，就是被直接的关闭掉。默认 30 分钟，单位毫秒。

charset : 连接的初始化字符集。默认为 utf8。

txIsolation : 前端连接的初始化事务隔离级别，只在初始化的时候使用，后续会根据客户端传递过来的属性对后端数据库连接进行同步。默认为 REPEATED_READ，设置值为数字默认 3。

READ_UNCOMMITTED = 1;

READ_COMMITTED = 2;

REPEATED_READ = 3;

SERIALIZABLE = 4;

sqlExecuteTimeout:SQL 执行超时的时间，Mycat 会检查连接上最后一次执行 SQL 的时间，若超过这个时间则会直接关闭这连接。默认时间为 300 秒，单位秒。

7.9.11 心跳属性

mycat 中有几个周期性的任务来异步的处理一些我需要的工作。这些属性就在系统调优的过程中也是比不可少的。

processorCheckPeriod : 清理 NIOProcessor 上前后端空闲、超时和关闭连接的间隔时间。默认是 1 秒，单位毫秒。。

dataNodeIdleCheckPeriod : 对后端连接进行空闲、超时检查的时间间隔，默认是 300 秒，单位毫秒。

dataNodeHeartbeatPeriod : 对后端所有读、写库发起心跳的间隔时间，默认是 10 秒，单位毫秒。

7.9.12 服务相关属性

这里介绍一个与服务相关的属性，主要会影响外部系统对 mycat 的感知。

bindIp : mycat 服务监听的 IP 地址，默认值为 0.0.0.0。

serverPort : 定义 mycat 的使用端口，默认值为 8066。

managerPort : 定义 mycat 的管理端口，默认值为 9066。

7.9.13 fakeMySQLVersion

mycat 模拟的 mysql 版本号，默认值为 5.6 版本，如非特需，不要修改这个值，目前支持设置 5.5,5.6 版本，其他版本可能会有问题。

此特性从 1.6 版本开始支持。

7.9.14 全局表一致性检测

```
<property name="useGlobleTableCheck">0</property> <!-- 1 为开启全加班一致性检测、0 为关闭 -->
```

原理通过在全局表增加_MYCAT_OP_TIME 字段来进行一致性检测，类型为 bigint，create 语句通过 mycat 执行会自动加上这个字段，其他情况请自己手工添加。

此特性从 1.6 版本开始支持。

“增加 mycat 新任务，全局表定义中，需要有一个时间戳字段，每次记录的 update, insert，确保时间字段赋值，并且 mycat 增加定时检测逻辑，检测记录总量，以及最新时间戳的匹配，简单有效的发现全局表不一致的问题。/ 测试修复类 / 1.5&2.0 /12.9 /leader-us”

全局表一致性定时检测主要分为两个部分：

1. SQL 拦截部分

主要实现对所有全局表中记录进行修改的语句进行拦截，比如：

ServerParse. INSERT，

ServerParse. UPDATE，

ServerParse. REPLACE (mycat-server 不支持)

对所有对全局表的 insert, update 操作进行拦截，首先判断该全局表是否存在一个记录时间戳的内部列_mycat_op_time：

```
public class GlobalTableUtil{
```

```
/** 全局表 保存修改时间戳的字段名，用于全局表一致性检查 */
```

```
public static final String GLOBAL_TABLE_MYCAT_COLUMN = "_mycat_op_time";
```

如果不存在，输出警告，哪个 db 的哪个全局表没有内部列：

```
if(innerColumnNotExist.size() > 0){
    for(SQLQueryResult<Map<String, String>> map : innerColumnNotExist){
        if(tableName.equalsIgnoreCase(map.getTableName())){
            StringBuilder warnStr = new StringBuilder();
            if(map != null)
                warnStr.append(map.getDataNode()).append(".");
            warnStr.append(tableName).append(" inner column: ")
                .append(GlobalTableUtil.GLOBAL_TABLE_MYCAT_COLUMN)
                .append(" is not exist.");
            LOGGER.warn(warnStr.toString());
            return sql;
        }
    }
}
```

然后返回原始 sql。 不需要进行拦截。

如果存在一个记录时间戳的内部列，那么对该 insert 或者 update 语句进行 SQL 拦截修改：

```
if(sqlType == ServerParse.INSERT){
    sql = convertInsertSQL(sql, tableName);
}
if(sqlType == ServerParse.UPDATE){
    sql = convertUpdateSQL(sql, tableName);
}
```

1.1 insert语句的拦截逻辑

对所有对全局表进行insert的sql语句，进行改写，比如下面的user是全局表：

```
insert into user(id,name)
valueS(1111,'dig'),
(1111, 'dig'),
```

```
(1111, 'dig') ,
```

```
(1111, 'dig');
```

会被改写成:

```
insert into user(id,name, _mycat_op_time)
```

```
valueS(1111, 'dig', 1450423751170),
```

```
(1111, 'dig', 1450423751170),
```

```
(1111, 'dig', 1450423751170) ,
```

```
(1111, 'dig', 1450423751170);
```

其中 `_mycat_op_time` 是内部列的名称:

```
public static final String GLOBAL_TABLE_MYCAT_COLUMN = "_mycat_op_time";
```

而 `1450423751170` 是在插入时在 `mycat-server` 上生成的一个时间戳对应的 `long` 整数(对应到数据库是 `bigint`)。然后该语句发送给所有 `db` 在其全局表中进行插入。

如果 `insert` 语句自带了内部列 `_mycat_op_time`, 比如:

```
insert into user(id,name, _mycat_op_time)
```

```
valueS(1111, 'dig', 13545);
```

那么会输出警告, 并且也进行拦截改写成如下形式:

```
insert into user(id,name, _mycat_op_time)
```

```
valueS(1111, 'dig', 1450423751170);
```

然后发送给所有 `db` 在其全局表中进行插入。

对 `mycat-server` 不支持的 `sql` 语句, 本拦截器, 不进行任何操作, 直接返回原始 `sql`。如果在拦截过程中发生任何异常, 也返回原始 `sql` 语句, 不进行任何修改操作。保证该拦截不会影响系统原有的健壮性。

1.2 update语句的拦截逻辑

`Update` 语句的拦截逻辑和 `insert` 语句原理是相似的。也是判断是否有内部列。

如果没有输出警告信息, 如果有则进行拦截。

对全局表 `user` 的如下 `update`:

```
update user set name='dddd',pwd='aaa'
```

```
where id=2
```

会被改写成:

```
update user set name='dddd',pwd='aaa', _mycat_op_time=1450423751170
```

```
where id=2
```

如果原始 `sql` 带有 `_mycat_op_time` 那么进行警告, 然后替换它的值, 比如:

```
update user set name='dddd',pwd='aaa', _mycat_op_time=1111
```

```
where id=2;
```

会被改写成:

```
update user set name='dddd',pwd='aaa', _mycat_op_time=1450423751170
```

```
where id=2;
```

然后将语句发送给所有的全局表进行执行。

这样的话,如果有哪个表上的insert,update执行失败,那么内部列 `_mycat_op_time` 的最大值,以及全局表的记录总数就会不一致。Delete语句也一样,只是无需拦截。下面的检查机制就是根据这个原理来操作的。

2. 一致性的定时检测

在MycatServer的startup中引入一个定时检查任务:

```
timer.schedule(glableTableConsistencyCheck(), 0L, 1000 * 1000L);
```

```
// 全局表一致性检查任务
```

```
private TimerTask glableTableConsistencyCheck() {  
    return new TimerTask() {  
        @Override  
        public void run() {  
            timerExecutor.execute(new Runnable() {  
                @Override  
                public void run() {  
                    GlobalTableUtil.consistencyCheck();  
                }  
            });  
        }  
    };  
};
```

其实现在GlobalTableUtil 类中:

该类首先获得所有的全局表:

```
static {  
    getGlobalTable(); // 初始化 globalTableMap
```

```
}
```

其实现，参见代码。

GlobalTableUtil.consistencyCheck() 的实现，主要思路是，首先根据所有的全局表，找到对应的PhysicalDBNode，然后找到对应的PhysicalDatasource，然后对PhysicalDatasource中的所有db进行三项检测：

2.1 检测全局表的内部列是否存在

```
checker.checkInnerColumnExist();
```

检测的实现是通过一个SQLJob来异步操作的，对应的SQL语句为：

```
select count(*) as inner_col_exist from information_schema.columns where column_name='  
_mycat_op_time' and table_name='user' and table_schema='db1';
```

如果返回的inner_col_exist 大于0,那么就表示存在内部列，如果等于0，那么就表示不存在内部列。

如果PhysicalDatasource上某个db的全局表没有内部列，那么将这些db记录在一个list中，然后在SQL 拦截过程中进行判断，如果是全局表，但是没有内部列，那么就输出警告，不对SQL进行拦截改写，因为该全局表没有内部列，无需改写SQL。在第一项检测完成之后，才能进行第二项检测。

2.2 检测全局表的记录总数

```
checker.checkRecordCout();
```

检查过程是类似的，都是通过SQLjob来完成的，只是对应的语句不一样：

```
select count(*) as record_count from user; (假设user表为全局表)
```

2.3 检测全局表的时间戳的最大值

```
checker.checkMaxTimeStamp();
```

检查过程是类似的，都是通过SQLjob来完成的，只是对应的语句不一样：

```
select max(_mycat_op_time) as max_timestamp from user (假设user表为全局表)
```

三项检查完成之后，就获得了如下所示的结果：

全局表的记录总数(user表为全局表，并且系统有三个db)：

db1. user.record_count: 43546565

db2. user.record_count: 43546565

db3. user.record_count: 43546565

全局表的最大时间戳：


```
db1. user.max_timestamp: 1450578802241
db2. user.max_timestamp: 1450578802241
db3. user.max_timestamp: 1450578802241
```

然后前端，比如 mycat-eye 就可以将该结果显示出来。目前直接在log中输出，也可以考虑引入像 H2这样的Java实现的嵌入式数据库来记录该结果。**H2**实现为仅仅一个jar包，十分适合作为 mycat-server层面的一个非文件存储方式。有一些信息如果存在在文件中，查询起来不太方便，比如上面的检测结果就是如此。

实际的SQLJob的执行，主要参照了原有的heartbeat的实现，主要在下面两个类中：

MySQLConsistencyChecker

MySQLConsistencyHelper

具体可以参考代码，和heartbeat的实现基本是一样的。

每一次定时检查，会对所有全局表进行上述三项检测。

总结成一句：

SQL的拦截实现记录全局表被修改时的时间戳；定时任务实现对全局表记录总数和时间戳最大值的获取。

3. 如何使用全局表一致性检测

- 1> 在所有全局表中增加一个 bigint 的内部列，列名为 **_mycat_op_time**，(alter table t add column _mycat_op_time bigint [not null default 0]); 同时建议在该列上建立索引(alter table t add index _op_idx(_mycat_op_time))
- 2> 在对全局表进行crud时，最好将内部列当作不存在一样，也就是最好不要对内部列 update, insert等操作，不然会在Log中进行警告：不用操作内部列；
- 3> 因为全局表多了一个内部列，所以在对全局表进行insert时，必须携带列名，也就是insert into t(id,name) values(xx,xx)，不能使用insert into t values(xx,xx)；因为会报错：列数不对。这是唯一的一个小问题。未来可能会fix掉。

7.9.15 分布式事务开关

```
<!--分布式事务开关, 0为不过滤分布式事务, 1为过滤分布式事务(如果分布式事务内只涉及全局表, 则不过滤), 2为不过滤分布式事务,但是记录分布式事务日志-->
```

```
<property name="handleDistributedTransactions">0</property>
```

主要应用场景，主要为了控制是否允许跨库事务。

此特性从 1.6 版本开始支持。

7.9.16 Off Heap for Mycat

此特性从 1.6 版本开始支持。

```
<!--  
    off heap for merge/order/group/limit    1 开启 0 关闭  
-->  
<property name="useOffHeapForMerge">1</property>
```

- 1.使用非堆内存(Direct Memory)处理跨分片结果集的 Merge/order by/group by/limit。
- 2.通过 server.xml 中的 useOffHeapForMerge 参数配置是否启用非堆内存处理跨分片结果集
- 3.Mycat 内存分层管理:

- a.结果集处理内存;
- b.系统预留内存;
- c.网络处理内存共三块。

其中网络处理内存部分全部为 Direct Memory，结果集内存分为 Direct Memory 和 HeapMemory。

但目前仅使用 Direct Memory。系统预留内存为 On Heap Memory。JVM 参数，必须设置-

XX:MaxDirectMemorySize 和 -Xmx

例如:-Xmx1024m -Xmn512m -XX:MaxDirectMemorySize=2048m -Xss256K -XX:+UseParallelGC

上述分层可以避免 OOM 问题，以及减少 Full GC 回收时间，提高 mycat 响应速度。

- 4.使用 TimeSort 和 RadixSort，跨分片结果集合并排序使用 PriorityQueue，其中经测试 RadixSort 适合 LONG，INT，SHORT,Float，Double，String 数据类型排序，性能优越。

- 5.Java obj 连续内存存取，二进制序列化和反序列化，使用缓存友好的数据结构 Map 和 Row。

- 6.支持内存和外存并存的排序方式，结果集排序可以达上亿规模。此时应注意:

- a.此时前端和后端空闲连接超时检测时间应该设置大些，避免空闲检测关闭 front 或者 backend connection,造成 Mysqlclient 连接丢失时结果集无法正确。

- b.设置-Xmn 值尽可能大些，新生代使用 UseParallelGC 垃圾回收器，-Xss 设置 512K 比较合适，物理内存足够时，MaxDirectMemorySize 尽可能设置大些，可以加快结果集处理时间，

例如:-Xmx1024m -Xmn512m -XX:MaxDirectMemorySize=2048m -Xss256k -XX:+UseParallelGC。

7.10 rule.xml

rule.xml 里面就定义了我们表进行拆分所涉及到的规则定义。我们可以灵活的对表使用不同的分片算法，或者对表使用相同的算法但具体的参数不同。这个文件里面主要有 tableRule 和 function 这两个标签。在具体使用过程中可以按照需求添加 tableRule 和 function。

7.11 tableRule 标签

这个标签定义表规则。

定义的表规则，在 schema.xml：

```
<tableRule name="rule1">
<rule>
    <columns>id</columns>
    <algorithm>func1</algorithm>
</rule>
</tableRule>
```

name 属性指定唯一的名字，用于标识不同的表规则。

内嵌的 rule 标签则指定对物理表中的哪一列进行拆分和使用什么路由算法。

columns 内指定要拆分的列名字。

algorithm 使用 function 标签中的 name 属性。连接表规则 and 具体路由算法。当然，多个表规则可以连接到同一个路由算法上。table 标签内使用。让逻辑表使用这个规则进行分片。

7.12 function 标签

```
<function name="hash-int"
    class="io.mycat.route.function.PartitionByFileMap">
    <property name="mapFile">partition-hash-int.txt</property>
</function>
```

name 指定算法的名字。

class 制定路由算法具体的类名字。

property 为具体算法需要用到的一些属性。

路由算法的配置可以查看算法章节。

第 8 章 Mycat 的分片 join

8.1 join 概述

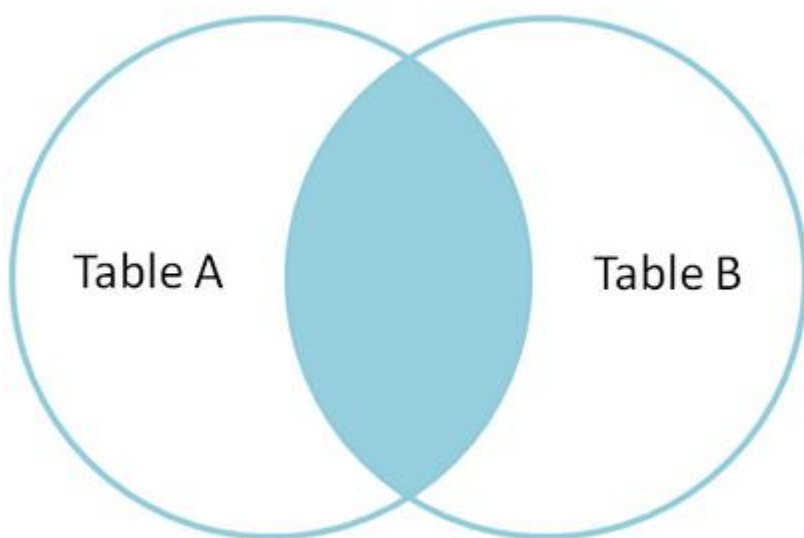
Join 绝对是关系型数据库中最常用一个特性，然而在分布式环境中,跨分片的 join 确是最复杂的，最难解决一个问题。

下面我们简单介绍下各种 Join 操作。

INNER JOIN

内连接，也叫等值连接，inner join 产生同时符合 A 表和 B 表的一组数据。

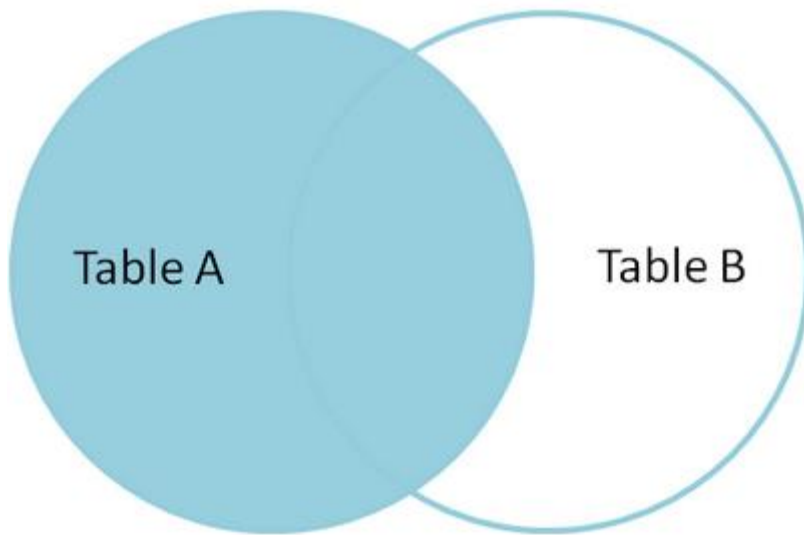
如图：



LEFT JOIN

左连接从 A 表(左)产生一套完整的记录,与匹配的 B 表记录(右表) .如果没有匹配,右侧将包含 null,在 Mysql 中等同于 left outer join。

如图：



RIGHT JOIN

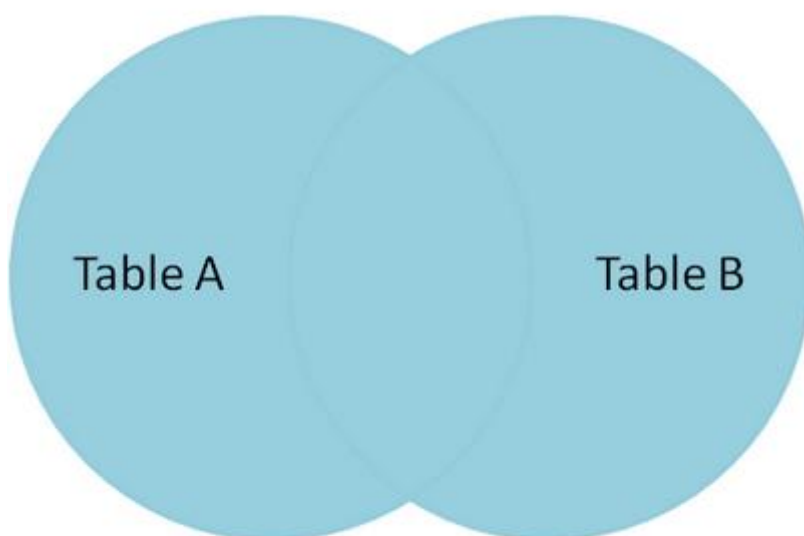
同 Left join, AB 表互换即可。

Cross join

交叉连接，得到的结果是两个表的乘积，即笛卡尔积。笛卡尔（Descartes）乘积又叫直积。假设集合 $A=\{a,b\}$ ，集合 $B=\{0,1,2\}$ ，则两个集合的笛卡尔积为 $\{(a,0),(a,1),(a,2),(b,0),(b,1),(b,2)\}$ 。可以扩展到多个集合的情况。类似的例子有，如果 A 表示某学校学生的集合，B 表示该学校所有课程的集合，则 A 与 B 的笛卡尔积表示所有可能的选课情况。

Full join

全连接产生的所有记录（双方匹配记录）在表 A 和表 B。如果没有匹配,则对面将包含 null。



性能建议

尽量避免使用 Left join 或 Right join,而用 Inner join

在使用 Left join 或 Right join 时, ON 会优先执行, where 条件在最后执行, 所以在使用过程中, 条件尽可能的在 ON 语句中判断, 减少 where 的执行

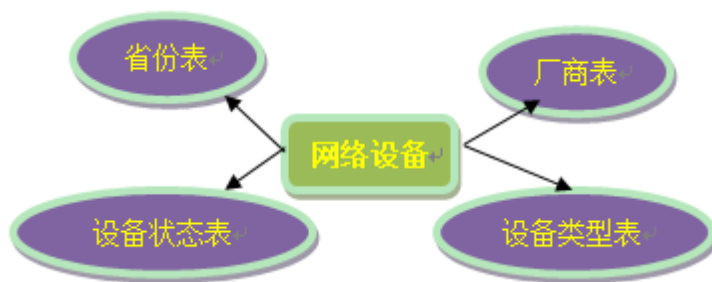
少用子查询, 而用 join。

Mycat 目前版本支持跨分片的 join,主要实现的方式有四种。

全局表, ER 分片, catletT(人工智能)和 ShareJoin, ShareJoin 在开发版中支持, 前面三种方式 1.3.0.1 支持。

8.2 全局表

一个真实的业务系统中, 往往存在大量的类似字典表的表格, 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”, 这些表基本上很少变动, 可以根据主键 ID 进行缓存, 下面这张图说明了一个典型的“标签关系”图:



在分片的情况下, 当业务表因为规模而进行分片以后, 业务表与这些附属的字典表之间的关联, 就成了比较棘手的问题, 考虑到字典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, MyCAT 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, 则从另外一个方面, 很好的解决了数据 JOIN 的难题。

通过全局表+基于 E-R 关系的分片策略, MyCAT 可以满足 80%以上的企业应用开发。

配置

全局表配置比较简单，不用写 Rule 规则，如下配置即可：

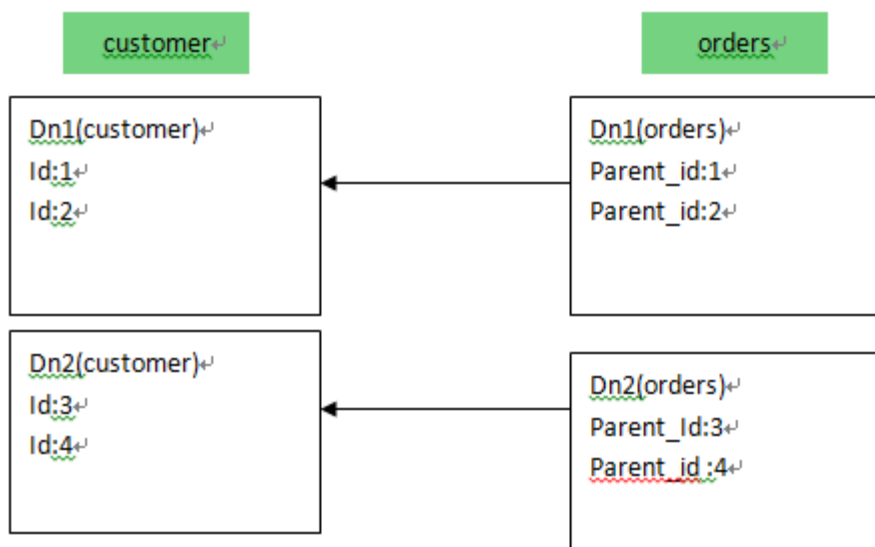
```
<table name="company" primaryKey="ID" type="global" dataNode="dn1,dn2,dn3" />
```

需要注意的是，全局表每个分片节点上都要有运行创建表的 DDL 语句。

8.3 ER Join

MyCAT 借鉴了 NewSQL 领域的新秀 Foundation DB 的设计思路，Foundation DB 创新性的提出了 Table Group 的概念，其将子表的存储位置依赖于主表，并且物理上紧邻存放，因此彻底解决了 JOIN 的效率和性能问题，根据这一思路，提出了基于 E-R 关系的数据分片策略，子表的记录与所关联的父表记录存放在同一个数据分片上。

customer 采用 sharding-by-intfile 这个分片策略，分片在 dn1,dn2 上，orders 依赖父表进行分片，两个表的关联关系为 orders.customer_id=customer.id。于是数据分片和存储的示意图如下：



这样一来，分片 Dn1 上的 customer 与 Dn1 上的 orders 就可以进行局部的 JOIN 联合，Dn2 上也如此，再合并两个节点的数据即可完成整体的 JOIN，试想一下，每个分片上 orders 表有 100 万条，则 10 个分片就有 1 个亿，基于 E-R 映射的数据分片模式，基本上解决了 80% 以上的企业应用所面临的问题。

配置

以上述例子为例，schema.xml 中定义如下的分片配置：

```
<table name="customer" dataNode="dn1,dn2" rule="sharding-by-intfile">
  <childTable name="orders" joinKey="customer_id" parentKey="id"/>
</table>
```


8.4 Share join

ShareJoin 是一个简单的跨分片 Join,基于 HBT 的方式实现。

目前支持 2 个表的 join,原理就是解析 SQL 语句,拆分成单表的 SQL 语句执行,然后把各个节点的数据汇集。

配置

支持任意配置的 A,B 表如:

A,B 的 dataNode 相同

```
<table name="A" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />
<table name="B" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />
```

A,B 的 dataNode 不同

```
<table name="A" dataNode="dn1,dn2 " rule="auto-sharding-long" />
<table name="B" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />
```

或

```
<table name="A" dataNode="dn1 " rule="auto-sharding-long" />
<table name="B" dataNode=" dn2,dn3" rule="auto-sharding-long" />
```

代码测试

先把表 company 从全局表修改下配置

```
<table name="company" primaryKey="ID" dataNode="dn1,dn2,dn3" rule="mod-long" />
```

重新插入数据

```
mysql> delete from company;
```

```
Query OK, 9 rows affected (0.19 sec)
```

```
mysql> insert company (id,name) values(1,'mycat');
```

```
Query OK, 1 row affected (0.08 sec)
```

```
mysql> insert company (id,name) values(2,'ibm');
```

```
Query OK, 1 row affected (0.03 sec)
```

```
mysql> insert company (id,name) values(3,'hp');
```

Query OK, 1 row affected (0.03 sec)

下面可以看下普通的 join 和 sharejoin 的区别

```
mysql> select a.*,b.id, b.name as tit from customer a,company b where a.company_id=b.id;
```

```
+----+-----+-----+-----+----+-----+
| id | name | company_id | sharding_id | id | tit |
+----+-----+-----+-----+----+-----+
| 3 | feng | 3 | 10000 | 3 | hp |
+----+-----+-----+-----+----+-----+
```

1 row in set (0.03 sec)

```
mysql> /*!mycat:catlet=demo.catlets.ShareJoin */ select a.*,b.id, b.name as tit from customer a,company b
on a.company_id=b.id;
```

```
+----+-----+-----+-----+----+-----+
| id | name | company_id | sharding_id | id | tit |
+----+-----+-----+-----+----+-----+
| 3 | feng | 3 | 10000 | 3 | hp |
| 1 | wang | 1 | 10000 | 1 | mycat |
| 2 | xue | 2 | 10010 | 2 | ibm |
+----+-----+-----+-----+----+-----+
```

3 rows in set (0.05 sec)

其他两种写法

```
/*!mycat:catlet=demo.catlets.ShareJoin */ select a.*,b.id, b.name as tit from customer a join company b on
a.company_id=b.id;
```

```
+----+-----+-----+-----+----+-----+
| id | name | company_id | sharding_id | id | tit |
+----+-----+-----+-----+----+-----+
| 3 | feng | 3 | 10000 | 3 | hp |
| 1 | wang | 1 | 10000 | 1 | mycat |
```

```
| 2 | xue |      2 |    10010 | 2 | ibm |
```

```
+----+-----+-----+-----+----+-----+
```

3 rows in set (0.01 sec)

```
/*!mycat:catlet=demo.catlets.ShareJoin */ select a.*,b.id, b.name as tit from customer a join company b where  
a.company_id=b.id;
```

```
+----+-----+-----+-----+----+-----+
```

```
| id | name | company_id | sharding_id | id | tit |
```

```
+----+-----+-----+-----+----+-----+
```

```
| 3 | feng |      3 |    10000 | 3 | hp |
```

```
| 1 | wang |      1 |    10000 | 1 | mycat |
```

```
| 2 | xue |      2 |    10010 | 2 | ibm |
```

```
+----+-----+-----+-----+----+-----+
```

3 rows in set (0.01 sec)

对*的支持, 还可以这样写 SQL

```
mysql> /*!mycat:catlet=demo.catlets.ShareJoin */ select a.*,b.* from customer a join company b on  
a.company_id=b.id;
```

```
+----+-----+-----+-----+----+-----+
```

```
| id | name | company_id | sharding_id | name |
```

```
+----+-----+-----+-----+----+-----+
```

```
| 1 | wang |      1 |    10000 | mycat |
```

```
| 2 | xue |      2 |    10010 | ibm |
```

```
| 3 | feng |      3 |    10000 | hp |
```

```
+----+-----+-----+-----+----+-----+
```

3 rows in set (0.02 sec)

```
mysql> /*!mycat:catlet=demo.catlets.ShareJoin */ select * from customer a join company b on  
a.company_id=b.id;
```

```
+----+-----+-----+-----+----+-----+
```

```
| id | name | company_id | sharding_id | name |
+----+-----+-----+-----+-----+
| 1 | wang | 1 | 10000 | mycat |
| 2 | xue | 2 | 10010 | ibm |
| 3 | feng | 3 | 10000 | hp |
+----+-----+-----+-----+-----+
3 rows in set (0.02 sec)
```

```
/*!mycat:catlet=demo.catlets.ShareJoin */ select a.id,a.user_id,a.traveldate,a.fee,a.days,b.id as nnid, b.title as
tit from travelrecord a join hotnews b on b.id=a.days order by a.id ;
```

8.5 catlet (人工智能)

解决跨分片的 SQL JOIN 的问题，远比想象的复杂，而且往往无法实现高效的处理，既然如此，就依靠人工的智力，去编程解决业务系统中特定几个必须跨分片的 SQL 的 JOIN 逻辑，MyCAT 提供特定的 API 供程序员调用，这就是 MyCAT 创新性的思路——人工智能。

以一个跨节点的 SQL 为例。

```
Select a.id,a.name,b.title from a,b where a.id=b.id
```

其中 a 在分片 1, 2, 3 上, b 在 4, 5, 6 上, 需要把数据全部拉到本地 (MyCAT 服务器) ,执行 JOIN 逻辑, 具体过程如下 (只是一种可能的执行逻辑) :

```
EngineCtx ctx=new EngineCtx();//包含 MyCat.SQLEngine
```

```
String sql=, "select a.id ,a.name from a " ;
```

```
//在 a 表所在的所有分片上顺序执行下面的本地 SQL
```

```
ctx.executeNativeSQLSequceJob(allAnodes,new DirectDBJoinHandler());
```

DirectDBJoinHandler 类是一个回调类, 负责处理 SQL 执行过程中返回的数据包, 这里的这个类, 主要目的是用 a 表返回的 ID 信息, 去 b 表上查询对于的记录, 做实时的关联:

```
DirectDBJoinHandler{
```

```
Private HashMap<byte[],byte[]> rows;//Key 为 id,value 为一行记录的 Column 原始 Byte 数组, 这里是
a.id,a.name,b.title 这三个要输出的字段
```

```

    Public Boolean onHeader(byte[] header)
    {
        //保存 Header 信息，用于从 Row 中获取 Field 字段值
    }

    Public Boolean onRowData(byte[] rowData)
    {
        String id=getColumnAsString( "id" );
        //放入结果集,b.title 字段未知，所以先空着
        rows.put(getColumnRawBytes( "id" ),rowData);

        //满 1000 条，发送一个查询请求
        String sql=" select b.id, b.name  from b where id in (.....)" ;

        //此 SQL 在 B 的所有节点上并发执行，返回的结果直接输出到客户端
        ctx.executeNativeSQLParallJob(allBNodes,sql ,new MyRowOutPutDataHandler(rows));

    }

    Public Boolean onRowFinished()
    {
    }

    Public void onJobFinished()
    {
        If(ctx.allJobFinished())
        {
            {///used total time ....

        }
    }
}

```

最后，增加一个 Job 事件监听器，这里是所有 Job 完成后，往客户端发送 RowEnd 包，结束整个流程。

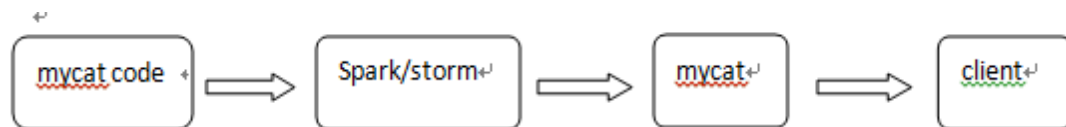
```
ctx.setJobEventListener(new JobEventHandler(){public void onJobFinished(){ client.writeRowEndPackage()}});
```

以上提供一个 SQL 执行框架，完全是异步的模式执行，并且以后会提供更多高质量的 API，简化分布式数据处理，比如内存结合文件的数据 JOIN 算法，分组算法，排序算法等等，期待更多的牛人一起来完善。

8.6 Spark/Storm 对 join 扩展

看到这个标题，可能会感到很奇怪，Spark 和 Storm 和 Join 有关系吗？有必要用 Spark，storm 吗？

mycat 后续的功能会引入 spark 和 storm 来做跨分片的 join,大致流程是这样的在 mycat 调用 spark,storm 的 api,把数据传送到 spark,storm，在 spark,storm 进行 join,在把数据传回 mycat,mycat 在返回给客户端。



第 9 章 全局序列号

9.1 全局序列号介绍

在实现分库分表的情况下，数据库自增主键已无法保证自增主键的全局唯一。为此，MyCat 提供了全局 sequence，并且提供了包含本地配置和数据库配置等多种实现方式。

9.2 本地文件方式

原理：此方式 MyCAT 将 sequence 配置到文件中，当使用到 sequence 中的配置后，MyCAT 会更新 classpath 中的 sequence_conf.properties 文件中 sequence 当前的值。

配置方式：

在 sequence_conf.properties 文件中做如下配置：

GLOBAL_SEQ.HISIDS=

GLOBAL_SEQ.MINID=1001

GLOBAL_SEQ.MAXID=1000000000

GLOBAL_SEQ.CURID=1000

其中 HISIDS 表示使用过的历史分段(一般无特殊需要可不配置)，MINID 表示最小 ID 值，MAXID 表示最大 ID 值，CURID 表示当前 ID 值。

server.xml 中配置：

```
<system><property name="sequenceHandlerType">0</property></system>
```

注：sequenceHandlerType 需要配置为 0，表示使用本地文件方式。

使用示例：

```
insert into table1(id,name) values(next value for MYCATSEQ_GLOBAL, 'test' );
```

缺点：当 MyCAT 重新发布后，配置文件中的 sequence 会恢复到初始值。

优点：本地加载，读取速度较快。

9.3 数据库方式

原理

在数据库中建立一张表, 存放 sequence 名称(name), sequence 当前值(current_value), 步长(increment int 类型每次读取多少个 sequence, 假设为 K)等信息;

Sequence 获取步骤:

1).当初次使用该 sequence 时, 根据传入的 sequence 名称, 从数据库这张表中读取 current_value, 和 increment 到 MyCat 中, 并将数据库中的 current_value 设置为原 current_value 值+increment 值。

.MyCat 将读取到 current_value+increment 作为本次要使用的 sequence 值, 下次使用时, 自动加 1, 当使用 increment 次后, 执行步骤 1)相同的操作。

MyCat 负责维护这张表, 用到哪些 sequence, 只需要在这张表中插入一条记录即可。若某次读取的 sequence 没有用完, 系统就停掉了, 则这次读取的 sequence 剩余值不会再使用。

配置方式:

server.xml 配置:

```
<system> <property name="sequenceHandlerType">1 </property> </system>
```

注: sequenceHandlerType 需要配置为 1, 表示使用数据库方式生成 sequence。

数据库配置:

1) 创建 MYCAT_SEQUENCE 表

– 创建存放 sequence 的表

```
DROP TABLE IF EXISTS MYCAT_SEQUENCE;
```

– name sequence 名称

– current_value 当前 value

– increment 增长步长! 可理解为 mycat 在数据库中一次读取多少个 sequence. 当这些用完后, 下次再从数据库中读取。

```
CREATE TABLE MYCAT_SEQUENCE (name VARCHAR(50) NOT NULL,current_value INT NOT NULL,increment INT NOT NULL DEFAULT 100, PRIMARY KEY(name)) ENGINE=InnoDB;
```

– 插入一条 sequence

```
INSERT INTO MYCAT_SEQUENCE(name,current_value,increment) VALUES ( 'GLOBAL' , 100000, 100);
```

2) 创建相关 function

– 获取当前 sequence 的值 (返回当前值,增量)

```
DROP FUNCTION IF EXISTS mycat_seq_currval;
```

```
DELIMITER
```

```
CREATE FUNCTION mycat_seq_currval(seq_name VARCHAR(50)) RETURNS varchar(64) CHARSET  
utf-8
```

```
DETERMINISTIC
```

```
BEGIN
```

```
DECLARE retval VARCHAR(64);
```

```
SET retval= "-999999999,null" ;
```

```
SELECT concat(CAST(current_value AS CHAR), ",",CAST(increment AS CHAR)) INTO retval FROM  
MYCAT_SEQUENCE WHERE name = seq_name;
```

```
RETURN retval;
```

```
END
```

```
DELIMITER;
```

– 设置 sequence 值

```
DROP FUNCTION IF EXISTS mycat_seq_setval;
```

```
DELIMITER
```

```
CREATE FUNCTION mycat_seq_setval(seq_name VARCHAR(50),value INTEGER) RETURNS varchar(64)  
CHARSET utf-8
```

```
DETERMINISTIC
```

```
BEGIN
```

```
UPDATE MYCAT_SEQUENCE
```

```
SET current_value = value
```

```
WHERE name = seq_name;
```

```
RETURN mycat_seq_currval(seq_name);
```

```
END
```

```
DELIMITER;
```

– 获取下一个 sequence 值

```
DROP FUNCTION IF EXISTS mycat_seq_nextval;
```

```
DELIMITER
```

```
CREATE FUNCTION mycat_seq_nextval(seq_name VARCHAR(50)) RETURNS varchar(64) CHARSET
```

utf-8

```
DETERMINISTIC
```

```
BEGIN
```

```
UPDATE MYCAT_SEQUENCE
```

```
SET current_value = current_value + increment WHERE name = seq_name;
```

```
RETURN mycat_seq_currval(seq_name);
```

```
END
```

```
DELIMITER;
```

4) **sequence_db_conf.properties** 相关配置,指定 **sequence** 相关配置在哪个节点上:

例如:

```
USER_SEQ=test_dn1
```

注意: MYCAT_SEQUENCE 表和以上的 3 个 function, 需要放在同一个节点上。function 请直接在具体节点的数据库上执行, 如果执行的时候报:

you might want to use the less safe log_bin_trust_function_creators variable

需要对数据库做如下设置:

windows 下 my.ini[mysqld]加上 log_bin_trust_function_creators=1

linux 下/etc/my.cnf 下 my.ini[mysqld]加上 log_bin_trust_function_creators=1

修改完后, 即可在 mysql 数据库中执行上面的函数。

使用示例:

```
insert into table1(id,name) values(next value for MYCATSEQ_GLOBAL, 'test' );
```

9.4 本地时间戳方式

ID= 64 位二进制 (42(毫秒)+5(机器 ID)+5(业务编码)+12(重复累加)

换算成十进制为 18 位数的 long 类型, 每毫秒可以并发 12 位二进制的累加。

使用方式:

a. 配置 server.xml

```
<property name="sequenceHandlerType">2</property>
```

b. 在 mycat 下配置: sequence_time_conf.properties

WORKID=0-31 任意整数

DATAACENTERID=0-31 任意整数

多个 mycat 节点下每个 mycat 配置的 WORKID, DATAACENTERID 不同, 组成唯一标识, 总共支持 $32 \times 32 = 1024$ 种组合。

ID 示例: 56763083475511

9.5 分布式 ZK ID 生成器

```
<property name="sequenceHandlerType">3</property>
```

Zk 的连接信息统一在 myid.properties 的 zkURL 属性中配置。

基于 ZK 与本地配置的分布式 ID 生成器(可以通过 ZK 获取集群 (机房) 唯一 InstanceID, 也可以通过配置文件配置 InstanceID)

ID 结构: long 64 位, ID 最大可占 63 位

* |current time millis(微秒时间戳 38 位, 可以使用 17 年)|clusterId (机房或者 ZKId, 通过配置文件配置 5 位) |instanceId (实例 ID, 可以通过 ZK 或者配置文件获取, 5 位) |threadId (线程 ID, 9 位) |increment(自增, 6 位)

* 一共 63 位, 可以承受单机房单机器单线程 $1000 \times (2^6) = 640000$ 的并发。

~~* 一共 63 位, 可以承受单机房单机器单线程 $1000 \times (2^7) = 1280000$ 的并发。~~

* 无悲观锁, 无强竞争, 吞吐量更高

配置文件: sequence_distributed_conf.properties, 只要配置里面: INSTANCEID=ZK 就是从 ZK 上获取 InstanceID。

9.6 Zk 递增方式

```
<property name="sequenceHandlerType">4</property>
```

Zk 的连接信息统一在 myid.properties 的 zkURL 属性中配置。

4 是 zookeeper 实现递增序列号

- * 配置文件: sequence_conf.properties
- * 只要配置好 ZK 地址和表名的如下属性
- * TABLE.MINID 某线程当前区间内最小值
- * TABLE.MAXID 某线程当前区间内最大值
- * TABLE.CURID 某线程当前区间内当前值
- * 文件配置的 MAXID 以及 MINID 决定每次取得区间, 这个对于每个线程或者进程都有效
- * 文件中的这三个属性配置只对第一个进程的第一个线程有效, 其他线程和进程会动态读取 ZK

9.7 其他方式

1) 使用 catelet 注解方式

```
/*!mycat:catlet=demo.catlets.BatchGetSequence */SELECT mycat_get_seq( 'GLOBAL' ,100);
```

注: 此方法表示获取 GLOBAL 的 100 个 sequence 值, 例如当前 GLOBAL 的最大 sequence 值为 5000, 则通过此方式返回的是 5001, 同时更新数据库中的 GLOBAL 的最大 sequence 值为 5100。

2) 利用 zookeeper 方式实现

.....

9.8 自增长主键

9.8.1 MyCAT 自增长主键和返回生成主键 ID 的实现

说明:

- 1) mysql 本身对非自增长主键, 使用 last_insert_id()是不会返回结果的, 只会返回 0;
- 2) mysql 只会对定义自增长主键, 可以用 last_insert_id()返回主键值;

MyCAT 目前提供了自增长主键功能, 但是如果对应的 mysql 节点上数据表, 没有定义 auto_increment, 那么在 MyCAT 层调用 last_insert_id()也是不会返回结果的。

正确配置方式如下:

1) mysql 定义自增主键

```
CREATE TABLE table1(
    'id_' INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    'name_' INT(10) UNSIGNED NOT NULL,
```

PRIMARY KEY ('id_')

) ENGINE=MYISAM AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;

2) mycat 定义主键自增

```
[root@test conf]# vim schema.xml
<?xml version="1.0"?>
<!DOCTYPE mycat:schema SYSTEM "schema.dtd">
<mycat:schema xmlns:mycat="http://org.opencloudb/">
  <schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">
    <!-- random sharding using mod sharind rule -->
    <!-- autoIncrement="true" 属性表示该表使用主键自增长策略-->
    <table name="table1" primaryKey="id_" autoIncrement="true" dataNode="dn1,dn2" rule="mod-long" />
    <table name="mycat_sequence" primaryKey="name" dataNode="dn1"/>
  </schema>

  <dataNode name="dn1" dataHost="localhost1" database="db1" />
  <dataNode name="dn2" dataHost="localhost1" database="db2" />

  <dataHost name="localhost1" maxCon="1000" minCon="20" balance="0" writeType="0" dbType="mysql" dbDriver="native">
    <heartbeat>select user()</heartbeat>
    <writeHost host="hostM1" url="127.0.0.1:3366" user="root" password="123456">
    </writeHost>
  </dataHost>
</mycat:schema>
```

3) mycat 对应 sequence_db_conf.properties 增加相应设置

TABLE1=dn1

4) 在数据库中 mycat_sequence 表中增加 TABLE1 表的 sequence 记录

测试使用:

127.0.0.1/root:[TESTDB> insert into tt2(name_) values('t1');

Query OK, 1 row affected (0.14 sec)

127.0.0.1/root:[TESTDB> select last_insert_id();

+-----+

| LAST_INSERT_ID() |

+-----+

| 100 |

+-----+

1 row in set (0.01 sec)

```
127.0.0.1/root:[TESTDB> insert into tt2(name_) values( 't2' );
```

Query OK, 1 row affected (0.00 sec)

```
127.0.0.1/root:[TESTDB> select last_insert_id();
```

```
+-----+
```

```
| LAST_INSERT_ID() |
```

```
+-----+
```

```
| 101 |
```

```
+-----+
```

1 row in set (0.00 sec)

```
127.0.0.1/root:[TESTDB> insert into tt2(name_) values( 't3' );
```

Query OK, 1 row affected (0.00 sec)

```
127.0.0.1/root:[TESTDB> select last_insert_id();
```

```
+-----+
```

```
| LAST_INSERT_ID() |
```

```
+-----+
```

```
| 102 |
```

```
+-----+
```

1 row in set (0.00 sec)

Myibatis 中新增记录后获取 last_insert_id 的示例:

```
<insert id="insert" parameterType="com.sam.user.model.User">
    insert into base_user
    (user_name,login_name,login_pwd,role_id)
    values
    ({userName},{loginName},{loginPwd},{roleId})
    <selectKey resultType="java.lang.Long" order="AFTER" keyProperty="id">
        select last_insert_id() as id
    </selectKey>
</insert>
```

第 10 章 Mycat 分片规则

10.1 分片规则概述

在数据切分处理中，特别是水平切分中，中间件最终要的两个处理过程就是数据的切分、数据的聚合。选择合适的切分规则，至关重要，因为它决定了后续数据聚合的难易程度，甚至可以避免跨库的数据聚合处理。

前面讲了数据切分中重要的几条原则，其中有几条是数据冗余，表分组（Table Group），这都是业务上规避跨库 join 的很好的方式，但不是所有的业务场景都适合这样的规则，因此本章将讲述如何选择合适的切分规则。

10.2 Mycat 全局表

如果你的业务中有些数据类似于数据字典，比如配置文件的配置，常用业务的配置或者数据量不大很少变动的表，这些表往往不是特别大，而且大部分的业务场景都会用到，那么这种表适合于 Mycat 全局表，无须对数据进行切分，只要在所有的分片上保存一份数据即可，Mycat 在 Join 操作中，业务表与全局表进行 Join 聚合会优先选择相同分片内的全局表 join，避免跨库 Join，在进行数据插入操作时，mycat 将把数据分发到全局表对应的所有分片执行，在进行数据读取时候将会随机获取一个节点读取数据。

目前 Mycat 没有做全局表的数据一致性检查，后续版本 1.4 之后可能会提供全局表一致性检查，检查每个分片的数据一致性。

全局表的配置如下

```
<table name="t_area" primaryKey="id" type="global" dataNode="dn1,dn2" />
```

10.3 ER 分片表

有一类业务，例如订单（order）跟订单明细（order_detail），明细表会依赖于订单，也就是说会存在表的主从关系，这类似业务的切分可以抽象出合适的切分规则，比如根据用户 ID 切分，其他相关的表都依赖于用户 ID，再或者根据订单 ID 切分，总之部分业务总会可以抽象出父子关系的表。这类表适用于 ER 分片表，子表的记录与所关联的父表记录存放在同一个数据分片上，避免数据 Join 跨库操作。

以 order 与 order_detail 例子为例，schema.xml 中定义如下的分片配置，order、order_detail 根据 order_id 进行数据切分，保证相同 order_id 的数据分到同一个分片上，在进行数据插入操作时，Mycat 会获取 order 所在的分片，然后将 order_detail 也插入到 order 所在的分片。

```

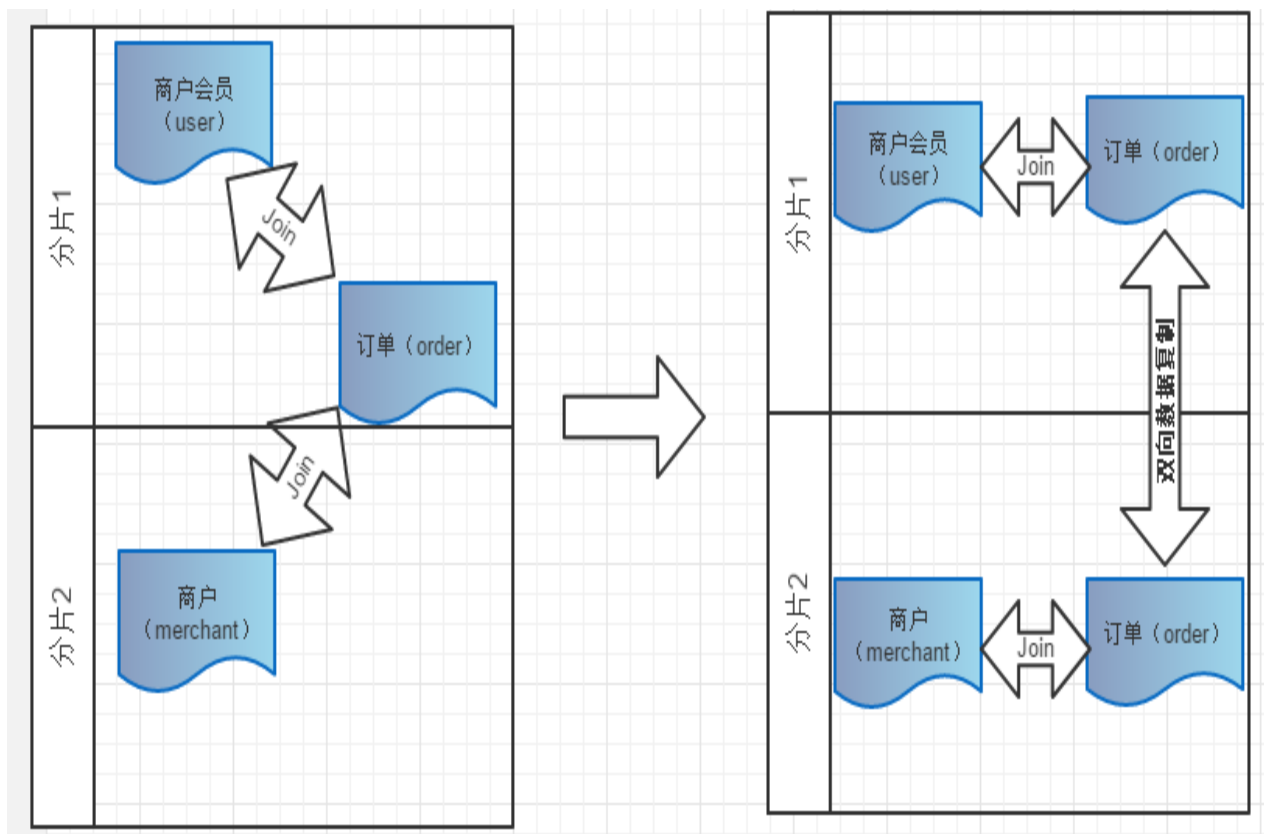
<table name="order" dataNode="dn$1-32" rule="mod-long">
<childTable name="order_detail" primaryKey="id" joinKey="order_id" parentKey="order_id" />
</table>

```

10.4 多对多关联

有一类业务场景是“主表 A+关系表+主表 B”，举例来说就是商户会员+订单+商户，对应这类业务，如何切分？

从会员的角度，如果需要查询会员购买的订单，那按照会员进行切分即可，但是如果查询商户当天售出的订单，那又需要按照商户做切分，可是如果既要按照会员又要按照商户切分，几乎是无法实现，这类业务如何选择切分规则非常难。目前还暂时无法很好支持这种模式下的 3 个表之间的关联。目前总的原则是需要从业务角度来看，关系表更偏向哪个表，即“A 的关系”还是“B 的关系”，来决定关系表跟从那个方向存储，未来 Mycat 版本中将考虑将中间表进行双向复制，以实现从 A-关系表 以及 B-关系表的双向关联查询如下图所示：



10.4.1 主键分片 vs 非主键分片

当你没人任何字段可以作为分片字段的时候，主键分片就是唯一选择，其优点是按照主键的查询最快，当采用自动增长的序号作为主键时，还能比较均匀的将数据分片在不同的节点上。

若有某个合适的业务字段比较合适作为分片字段，则建议采用此业务字段分片，选择分片字段的条件如下：

- 尽可能的比较均匀分布数据到各个节点上；
- 该业务字段是最频繁的或者最重要的查询条件。

常见的除了主键之外的其他可能分片字段有“订单创建时间”、“店铺类别”或“所在省”等。当你找到某个合适的业务字段作为分片字段以后，不必纠结于“牺牲了按主键查询记录的性能”，因为在这种情况下，MyCAT 提供了“主键到分片”的内存缓存机制，热点数据按照主键查询，丝毫不损失性能。

```
<table name="t_user" primaryKey="user_id" dataNode="dn$1-32" rule="mod-long">
<childTable name="t_user_detail" primaryKey="id" joinKey="user_id" parentKey="user_id" />
</table>
```

对于非主键分片的 table，填写属性 primaryKey，此时 MyCAT 会将你根据主键查询的 SQL 语句的第一次执行结果进行分析，确定该 Table 的某个主键在什么分片上，并进行主键到分片 ID 的缓存。第二次或后续查询 mycat 会优先从缓存中查询是否有 id->node 即主键到分片的映射，如果有直接查询，通过此种方法提高了非主键分片的查询性能。

本节主要讲了如何去分片，如何选择合适的分片的规则，总之尽量规避跨库 Join 是一条最重要的原则，下一节将介绍 Mycat 目前已有的分片规则，每种规则都有特定的场景，分析每种规则去选择合适的应用到项目中。

10.5 Mycat 常用的分片规则

10.5.1 分片枚举

通过在配置文件中配置可能的枚举 id，自己配置分片，本规则适用于特定的场景，比如有些业务需要按照省份或区县来做保存，而全国省份区县固定的，这类业务使用本条规则，配置如下：

```
<tableRule name="sharding-by-intfile">
<rule>
<columns>user_id</columns>
<algorithm>hash-int</algorithm>
</rule>
</tableRule>
<function name="hash-int" class="io.mycat.route.function.PartitionByFileMap">
<property name="mapFile">partition-hash-int.txt</property>
<property name="type">0</property>
<property name="defaultNode">0</property>
</function>

partition-hash-int.txt 配置：
10000=0
```

```
10010=1
DEFAULT_NODE=1
```

上面 columns 标识将要分片的表字段, algorithm 分片函数,
其中分片函数配置中, mapFile 标识配置文件名称, type 默认值为 0, 0 表示 Integer, 非零表示 String,
所有的节点配置都是从 0 开始, 及 0 代表节点 1

```
/**
 * defaultNode 默认节点:小于 0 表示不设置默认节点, 大于等于 0 表示设置默认节点
 * 默认节点的作用: 枚举分片时, 如果碰到不识别的枚举值, 就让它路由到默认节点
 * 如果不配置默认节点 (defaultNode 值小于 0 表示不配置默认节点), 碰到
 * 不识别的枚举值就会报错,
 * like this: can't find datanode for sharding column:column_name val:ffffff
 */
```

10.5.2 固定分片 hash 算法

本条规则类似于十进制的求模运算, 区别在于是二进制的操作, 是取 id 的二进制低 10 位, 即 id 二进制 &1111111111。

此算法的优点在于如果按照 10 进制取模运算, 在连续插入 1-10 时候 1-10 会被分到 1-10 个分片, 增大了插入的事务控制难度, 而此算法根据二进制则可能会分到连续的分片, 减少插入事务事务控制难度。

```
<tableRule name="rule1">
<rule>
<columns>user_id</columns>
<algorithm>func1</algorithm>
</rule>
</tableRule>

<function name="func1" class="io.mycat.route.function.PartitionByLong">
<property name="partitionCount">2,1</property>
<property name="partitionLength">256,512</property>
</function>
```

配置说明:

上面 columns 标识将要分片的表字段, algorithm 分片函数,
partitionCount 分片个数列表, partitionLength 分片范围列表
分区长度:默认为最大 $2^n=1024$, 即最大支持 1024 分区

约束:

count,length 两个数组的长度必须是一致的。

$1024 = \text{sum}((\text{count}[i] * \text{length}[i]))$. count 和 length 两个向量的点积恒等于 1024

用法例子:

本例的分区策略: 希望将数据水平分成 3 份, 前两份各占 25%, 第三份占 50%。(故本例非均匀分区)

```
// |<-----1024----->|
```

```
// |<---256-->|<---256-->|<-----512----->|
// | partition0 | partition1 | partition2 |
// | 共 2 份,故 count[0]=2 | 共 1 份, 故 count[1]=1 |
int[] count = new int[] { 2, 1 };
int[] length = new int[] { 256, 512 };
PartitionUtil pu = new PartitionUtil(count, length);

// 下面代码演示分别以 offerId 字段或 memberId 字段根据上述分区策略拆分的分配结果

int DEFAULT_STR_HEAD_LEN = 8; // cobar 默认会配置为此值
long offerId = 12345;
String memberId = "qiushuo";

// 若根据 offerId 分配, partNo1 将等于 0, 即按照上述分区策略, offerId 为 12345 时将会被分配到 partition0 中
int partNo1 = pu.partition(offerId);

// 若根据 memberId 分配, partNo2 将等于 2, 即按照上述分区策略, memberId 为 qiushuo 时将会被分到 partition2 中
int partNo2 = pu.partition(memberId, 0, DEFAULT_STR_HEAD_LEN);
```

如果需要平均分配设置: 平均分为 4 分片, $\text{partitionCount} \times \text{partitionLength} = 1024$

```
<function name="func1" class="io.mycat.route.function.PartitionByLong">
<property name="partitionCount">4</property>
<property name="partitionLength">256</property>
</function>
```

10.5.3 范围约定

此分片适用于, 提前规划好分片字段某个范围属于哪个分片,

$\text{start} \leq \text{range} \leq \text{end}$.

range start-end ,data node index

K=1000,M=10000.

```
<tableRule name="auto-sharding-long">
<rule>
<columns>user_id</columns>
<algorithm>rang-long</algorithm>
</rule>
</tableRule>
<function name="rang-long" class="io.mycat.route.function.AutoPartitionByLong">
<property name="mapFile">autopartition-long.txt</property>
<property name="defaultNode">0</property>
</function>
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数，
rang-long 函数中 mapFile 代表配置文件路径
defaultNode 超过范围后的默认节点。

所有的节点配置都是从 0 开始，及 0 代表节点 1，此配置非常简单，即预先制定可能的 id 范围到某个分片

0-500M=0

500M-1000M=1

1000M-1500M=2

或

0-100000000=0

100000001-200000000=1

10.5.4 取模

此规则为对分片字段求模运算。

```
<tableRule name="mod-long">
<rule>
<columns>user_id</columns>
<algorithm>mod-long</algorithm>
</rule>
</tableRule>
<function name="mod-long" class="io.mycat.route.function.PartitionByMod">
<!-- how many data nodes -->
<property name="count">3</property>
</function>
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数，

此种配置非常明确即根据 id 进行十进制求模预算，相比固定分片 hash，此种在批量插入时可能存在批量插入单事务插入多数据分片，增大事务一致性难度。

10.5.5 按日期（天）分片

此规则为按天分片。

```
<tableRule name="sharding-by-date">
<rule>
<columns>create_time</columns>
<algorithm>sharding-by-date</algorithm>
</rule>
</tableRule>
<function name="sharding-by-date" class="io.mycat.route.function.PartitionByDate">
<property name="dateFormat">yyyy-MM-dd</property>
<property name="sBeginDate">2014-01-01</property>
```

```

    <property name="sEndDate">2014-01-02</property>
    <property name="sPartitionDay">10</property>
</function>

```

配置说明：

columns：标识将要分片的表字段

algorithm：分片函数

dateFormat：日期格式

sBeginDate：开始日期

sEndDate：结束日期

sPartitionDay：分区天数，即默认从开始日期算起，分隔 10 天一个分区

如果配置了 sEndDate 则代表数据达到了这个日期的分片后后循环从开始分片插入。

```

Assert.assertEquals(true, 0 == partition.calculate( "2014-01-01" ));
Assert.assertEquals(true, 0 == partition.calculate( "2014-01-10" ));
Assert.assertEquals(true, 1 == partition.calculate( "2014-01-11" ));
Assert.assertEquals(true, 12 == partition.calculate( "2014-05-01" ));

```

10.5.6 取模范围约束

此种规则是取模运算与范围约束的结合，主要为了后续数据迁移做准备，即可以自主决定取模后数据的节点分布。

```

    <tableRule name="sharding-by-pattern">
<rule>
<columns>user_id</columns>
<algorithm>sharding-by-pattern</algorithm>
</rule>
</tableRule>
<function name="sharding-by-pattern" class="io.mycat.route.function.PartitionByPattern">
<property name="patternValue">256</property>
<property name="defaultNode">2</property>
<property name="mapFile">partition-pattern.txt</property>

</function>

```

partition-pattern.txt

```

partition-pattern.txt

# id partition range start-end ,data node index
##### first host configuration
1-32=0
33-64=1
65-96=2
97-128=3

```

```
##### second host configuration
129-160=4
161-192=5
193-224=6
225-256=7
0-0=7
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数，patternValue 即求模基数，defaultNode 默认节点，如果配置了默认，则不会按照求模运算

mapFile 配置文件路径

配置文件中，1-32 即代表 id%256 后分布的范围，如果在 1-32 则在分区 1，其他类推，如果 id 非数据，则会分配在 defaultNode 默认节点

```
String idVal = "0" ;
Assert.assertEquals(true, 7 == autoPartition.calculate(idVal));
idVal = "45a" ;
Assert.assertEquals(true, 2 == autoPartition.calculate(idVal));
```

10.5.7 截取数字做 hash 求模范围约束

此种规则类似于取模范围约束，此规则支持数据符号字母取模。

```
<tableRule name="sharding-by-prefixpattern">
<rule>
<columns>user_id</columns>
<algorithm>sharding-by-prefixpattern</algorithm>
</rule>
</tableRule>
<function name="sharding-by-pattern"
class="io.mycat.route.function.PartitionByPrefixPattern">
<property name="patternValue">256</property>
<property name="prefixLength">5</property>
<property name="mapFile">partition-pattern.txt</property>

</function>
```

partition-pattern.txt

```
partition-pattern.txt

# range start-end ,data node index
# ASCII
# 8-57=0-9 阿拉伯数字
# 64、65-90=@、A-Z
# 97-122=a-z
```

```
##### first host configuration
1-4=0
5-8=1
9-12=2
13-16=3
##### second host configuration
17-20=4
21-24=5
25-28=6
29-32=7
0-0=7
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数，patternValue 即求模基数，prefixLength ASCII 截取的位数

mapFile 配置文件路径

配置文件中，1-32 即代表 id%256 后分布的范围，如果在 1-32 则在分区 1，其他类推

此种方式类似方式 6 只不过采取的是将列种获取前 prefixLength 位列所有 ASCII 码的和进行求模 $\text{sum} \% \text{patternValue}$,获取的值，在范围内的分片数，

```
String idVal= "gf89f9a" ;
```

```
Assert.assertEquals(true, 0==autoPartition.calculate(idVal));
```

```
idVal= "8df99a" ;
```

```
Assert.assertEquals(true, 4==autoPartition.calculate(idVal));
```

```
idVal= "8dhdf99a" ;
```

```
Assert.assertEquals(true, 3==autoPartition.calculate(idVal));
```

10.5.8 应用指定

此规则是在运行阶段有应用自主决定路由到那个分片。

```
<tableRule name="sharding-by-substring">
<rule>
<columns>user_id</columns>
<algorithm>sharding-by-substring</algorithm>
</rule>
</tableRule>
<function name="sharding-by-substring"
class="io.mycat.route.function.PartitionDirectBySubString">
<property name="startIndex">0</property><!-- zero-based -->
<property name="size">2</property>
<property name="partitionCount">8</property>
```

```
<property name="defaultPartition">0</property>
</function>
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数

此方法为直接根据字符串（必须是数字）计算分区号（由应用传递参数，显式指定分区号）。

例如 id=05-100000002

在此配置中代表根据 id 中从 startIndex=0，开始，截取 siz=2 位数字即 05，05 就是获取的分区，如果没传默认分配到 defaultPartition

10.5.9 截取数字 hash 解析

此规则是截取字符串中的 int 数值 hash 分片。

```
<tableRule name="sharding-by-stringhash">
<rule>
<columns>user_id</columns>
<algorithm>sharding-by-stringhash</algorithm>
</rule>
</tableRule>
<function name="sharding-by-stringhash"
class="io.mycat.route.function.PartitionByString">
<property name="partitionLength">512</property><!-- zero-based -->
<property name="partitionCount">2</property>
<property name="hashSlice">0:2</property>
</function>
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数

函数中 partitionLength 代表字符串 hash 求模基数，

partitionCount 分区数，

hashSlice hash 预算位，即根据子字符串中 int 值 hash 运算

hashSlice : 0 means str.length(), -1 means str.length()-1

/**

* "2" -> (0,2)

* "1:2" -> (1,2)

* "1:" -> (1,0)

* "-1:" -> (-1,0)


```

* ":-1" -> (0,-1)

* ":@" -> (0,0)

*/

```

例子:

```

String idVal=null;
rule.setPartitionLength("512");
rule.setPartitionCount("2");
rule.init();
rule.setHashSlice("0:2");
//    idVal = "0";
//    Assert.assertEquals(true, 0 == rule.calculate(idVal));
//    idVal = "45a";
//    Assert.assertEquals(true, 1 == rule.calculate(idVal));


//last 4
rule = new PartitionByString();
rule.setPartitionLength("512");
rule.setPartitionCount("2");
rule.init();
//last 4 characters
rule.setHashSlice("-4:0");
idVal = "aaaabbbb0000";
Assert.assertEquals(true, 0 == rule.calculate(idVal));
idVal = "aaaabbbb2359";
Assert.assertEquals(true, 0 == rule.calculate(idVal));

```

10.5.10 一致性 hash

一致性 hash 预算有效解决了分布式数据的扩容问题。

```

<tableRule name="sharding-by-murmur">
<rule>
<columns>user_id</columns>
<algorithm>murmur</algorithm>
</rule>
</tableRule>

<function name="murmur" class="io.mycat.route.function.PartitionByMurmurHash">
<property name="seed">0</property><!-- 默认是 0-->
<property name="count">2</property><!-- 要分片的数据库节点数量，必须指定，否则没法分片-->
<property name="virtualBucketTimes">160</property><!-- 一个实际的数据库节点被映射为这么多虚拟

```

节点，默认是 160 倍，也就是虚拟节点数是物理节点数的 160 倍-->

```
<!--
```

```
<property name="weightMapFile">weightMapFile</property>
```

节点的权重，没有指定权重的节点默认是 1。以 properties 文件的格式填写，以从 0 开始到 count-1 的整数值也就是节点索引为 key，以节点权重值为值。所有权重值必须是正整数，否则以 1 代替 -->

```
<!--
```

```
<property name="bucketMapPath">/etc/mycat/bucketMapPath</property>
```

用于测试时观察各物理节点与虚拟节点的分布情况，如果指定了这个属性，会把虚拟节点的 murmur hash 值与物理节点的映射按行输出到这个文件，没有默认值，如果不指定，就不会输出任何东西 -->

```
</function>
```

10.5.11 按单月小时拆分

此规则是单月内按照小时拆分，最小粒度是小时，可以一天最多 24 个分片，最少 1 个分片，一个月完后下月从头开始循环。

每个月月尾，需要手工清理数据。

```
<tableRule name="sharding-by-hour">
<rule>
<columns>create_time</columns>
<algorithm>sharding-by-hour</algorithm>
</rule>
</tableRule>

<function name="sharding-by-hour" class="io.mycat.route.function.LatestMonthPartion">
<property name="splitOneDay">24</property>
</function>
```

配置说明：

columns：拆分字段，字符串类型 (yyyymmddHH)

splitOneDay：一天切分的分片数

```
LatestMonthPartion partion = new LatestMonthPartion();
partion.setSplitOneDay(24);
Integer val = partion.calculate("2015020100");
assertTrue(val == 0);
val = partion.calculate("2015020216");
assertTrue(val == 40);
val = partion.calculate("2015022823");
assertTrue(val == 27 * 24 + 23);

Integer[] span = partion.calculateRange("2015020100", "2015022823");
assertTrue(span.length == 27 * 24 + 23 + 1);
assertTrue(span[0] == 0 && span[span.length - 1] == 27 * 24 + 23);
```

```
span = partition.calculateRange("2015020100", "2015020123");
assertTrue(span.length == 24);
assertTrue(span[0] == 0 && span[span.length - 1] == 23);
```

10.5.12 范围求模分片

先进行范围分片计算出分片组，组内再求模

优点可以避免扩容时的数据迁移，又可以一定程度上避免范围分片的热点问题

综合了范围分片和求模分片的优点，分片组内使用求模可以保证组内数据比较均匀，分片组之间是范围分片可以兼顾范围查询。

最好事先规划好分片的数量，数据扩容时按分片组扩容，则原有分片组的数据不需要迁移。由于分片组内数据比较均匀，所以分片组内可以避免热点数据问题。

```
<tableRule name="auto-sharding-rang-mod">
<rule>
<columns>id</columns>
<algorithm>rang-mod</algorithm>
</rule>
</tableRule>
<function name="rang-mod"
    class="io.mycat.route.function.PartitionByRangeMod">
<property name="mapFile">partition-range-mod.txt</property>
<property name="defaultNode">21</property>
</function>
```

配置说明：

上面 columns 标识将要分片的表字段，algorithm 分片函数，

rang-mod 函数中 mapFile 代表配置文件路径

defaultNode 超过范围后的默认节点顺序号，节点从 0 开始。

partition-range-mod.txt

range start-end ,data node group size

以下配置一个范围代表一个分片组，=号后面的数字代表该分片组所拥有的分片的数量。

0-200M=5 //代表有 5 个分片节点

200M1-400M=1

400M1-600M=4

600M1-800M=4

800M1-1000M=6

10.5.13 日期范围 hash 分片

思想与范围求模一致，当由于日期在取模会有数据集中问题，所以改成 hash 方法。

先根据日期分组，再根据时间 hash 使得短期内数据分布的更均匀

优点可以避免扩容时的数据迁移，又可以一定程度上避免范围分片的热点问题

要求日期格式尽量精确些，不然达不到局部均匀的目的

```
<tableRule name="rangeDateHash">
  <rule>
    <columns>col_date</columns>
    <algorithm>range-date-hash</algorithm>
  </rule>
</tableRule>
<function name="range-date-hash"
class="io.mycat.route.function.PartitionByRangeDateHash">
  <property name="sBeginDate">2014-01-01 00:00:00</property>
  <property name="sPartionDay">3</property>
  <property name="dateFormat">yyyy-MM-dd HH:mm:ss</property>
  <property name="groupPartionSize">6</property>
</function>
```

sPartionDay 代表多少天分一个分片

groupPartionSize 代表分片组的大小

10.5.14 冷热数据分片

根据日期查询日志数据 冷热数据分布，最近 n 个月的到实时交易库查询，超过 n 个月的按照 m 天分片。

```
<tableRule name="sharding-by-date">
  <rule>
    <columns>create_time</columns>
    <algorithm>sharding-by-hotdate</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-hotdate" class="io.mycat.route.function.PartitionByHotDate">
  <property name="dateFormat">yyyy-MM-dd</property>
  <property name="sLastDay">10</property>
  <property name="sPartionDay">30</property>
```

</function>

10.5.15 自然月分片

按月份列分区，每个自然月一个分片，格式 `between` 操作解析的范例。

```
<tableRule name="sharding-by-month">
<rule>
<columns>create_time</columns>
<algorithm>sharding-by-month</algorithm>
</rule>
</tableRule>
<function name="sharding-by-month" class="io.mycat.route.function.PartitionByMonth">
<property name="dateFormat">yyyy-MM-dd</property>
<property name="sBeginDate">2014-01-01</property>
</function>
```

配置说明：

`columns`：分片字段，字符串类型

`dateFormat`：日期字符串格式

`sBeginDate`：开始日期

```
PartitionByMonth partition = new PartitionByMonth();
partition.setDateFormat("yyyy-MM-dd");
partition.setsBeginDate("2014-01-01");
partition.init();
Assert.assertEquals(true, 0 == partition.calculate("2014-01-01"));
Assert.assertEquals(true, 0 == partition.calculate("2014-01-10"));
Assert.assertEquals(true, 0 == partition.calculate("2014-01-31"));
Assert.assertEquals(true, 1 == partition.calculate("2014-02-01"));
Assert.assertEquals(true, 1 == partition.calculate("2014-02-28"));
Assert.assertEquals(true, 2 == partition.calculate("2014-03-1"));
Assert.assertEquals(true, 11 == partition.calculate("2014-12-31"));
Assert.assertEquals(true, 12 == partition.calculate("2015-01-31"));
Assert.assertEquals(true, 23 == partition.calculate("2015-12-31"));
```

10.5.16 有状态分片算法

有状态分片算法与之前的分片算法不同,它是为数据自动迁移而设计的.

直至 2018 年 7 月 24 日为止,现支持有状态算法的分片策略只有 `crc32slot` 欢迎大家提供更多有状态分片算法.

一个有状态分片算法在使用过程中暂时存在两个操作

一种是初始化,使用 mycat 创建配置带有有状态分片算法的 table 时(推介)或者第一次配置有状态分片算法的 table 并启动 mycat 时,有状态分片算法会根据表的 dataNode 的数量划分分片范围并生成 ruledata 下的文件,这个分片范围规则就是' 状态' ,一个表对应一个状态,对应一个有状态分片算法实例,以及对应一个满足以下命名规则的文件:

算法名字_schema 名字_table 名字.properties

文件里内容一般具有以下特征

```
8=91016-102399
7=79639-91015
6=68262-79638
5=56885-68261
4=45508-56884
3=34131-45507
2=22754-34130
1=11377-22753
0=0-11376
```

行数就是 table 的分片节点数量,每行的' 数字-数字' 就是分片算法生成的范围,这个范围与具体算法实现有关,一个分片节点可能存在多个范围,这些范围以逗号,分隔.一般来说,不要手动更改这个文件,应该使用算法生成范围,而且需要注意的是,物理库上的数据的分片字段的值一定要落在对应范围里.

一种是添加操作,即数据扩容,具体参考第六章的 6.8 与 6.9

添加节点,有状态分片算法根据节点的变化,重新分配范围规则,之后执行数据自动迁移任务.

10.5.17 crc32slot 分片算法

crc32slot 是有状态分片算法的实现之一,具体参考第六章 数据自动迁移方案设计

$\text{crc32}(\text{key}) \% 102400 = \text{slot}$

slot 按照范围均匀分布在 dataNode 上,针对每张表进行实例化,通过一个文件记录 slot 和节点映射关系,迁移过程中通过 zk 协调

其中需要在分片表中增加 slot 字段,用以避免迁移时重新计算,只需要迁移对应 slot 数据即可
分片最大个数为 102400 个,短期内应该够用,每分片一千万,总共可以支持一万亿数据

配置说明:

```
<table name="travelrecord" dataNode="dn1,dn2" rule="crc32slot" />
```

使用 mycat 配置完表后使用 mycat 创建表

```
USE TESTDB;  
CREATE TABLE `travelrecord` (  
    id xxxx  
    xxxxxxxx  
) ENGINE=INNODB DEFAULT CHARSET=utf8;
```

10.6 权限控制

10.6.1 远程连接配置(读、写权限)

目前 Mycat 对于中间件的连接控制并没有做太复杂的控制，目前只做了中间件逻辑库级别的读写权限控制。

```
<user name="mycat">  
    <property name="password">mycat</property>  
    <property name="schemas">order</property>  
<property name="readOnly">true</property>  
</user>  
<user name="mycat2">  
    <property name="password">mycat</property>  
    <property name="schemas">order</property>  
  
</user>
```

配置说明：

配置中 name 是应用连接中间件逻辑库的用户名。

mycat 中 password 是应用连接中间件逻辑库的密码。

order 中是应用当前连接的逻辑库中所对应的逻辑表。schemas 中可以配置一个或多个。

true 中 readOnly 是应用连接中间件逻辑库所具有的权限。true 为只读，false 为读写都有，默认为 false。

10.7 多租户支持

单租户就是传统的给每个租户独立部署一套 web + db。由于租户越来越多，整个 web 部分的机器和运维成本都非常高，因此需要改进到所有租户共享一套 web 的模式（db 部分暂不改变）。

基于此需求，我们对单租户的程序做了简单的改造实现 web 多租户共享。具体改造如下：

1.web 部分修改：

a.在用户登录时，在线程变量（ThreadLocal）中记录租户的 id

b.修改 jdbc 的实现：在提交 sql 时，从 ThreadLocal 中获取租户 id, 添加 sql 注释，把租户的 schema 放到注释中。例如： `/*!mycat : schema = test_01 */ sql ;`

2.在 db 前面建立 proxy 层，代理所有 web 过来的数据库请求。proxy 层是用 mycat 实现的，web 提交的 sql 过来时在注释中指定 schema, proxy 层根据指定的 schema 转发 sql 请求。

3.Mycat 配置：

```
<user name="mycat">
    <property name="password">mycat</property>
    <property name="schemas">order</property>
    <property name="readOnly">true</property>

</user>

<user name="mycat2">
    <property name="password">mycat</property>
    <property name="schemas">order</property>

</user>
```

第 11 章 常见问题与解决方案

11.1 Mycat 目前有哪些功能与特性？

答：

- 支持 SQL 92 标准；
- 支持 Mysql 集群，可以作为 Proxy 使用；
- 支持 JDBC 连接多数据库；
- 支持 NoSQL 数据库；
- 支持 galera for mysql 集群，percona-cluster 或者 mariadb cluster，提供高可用性数据分片集群；
- 自动故障切换，高可用性；
- 支持读写分离，支持 Mysql 双主多从，以及一主多从的模式；
- 支持全局表，数据自动分片到多个节点，用于高效表关联查询；
- 支持独有的基于 E-R 关系的分片策略，实现了高效的表关联查询；
- 支持一致性 Hash 分片，有效解决分片扩容难题；

- 多平台支持，部署和实施简单；
- 支持 Catelet 开发，类似数据库存储过程，用于跨分片复杂 SQL 的人工智能编码实现，143 行 Demo 完成跨分片的两个表的 JOIN 查询；
- 支持 NIO 与 AIO 两种网络通信机制，Windows 下建议 AIO，Linux 下目前建议 NIO；
- 支持 Mysql 存储过程调用；
- 以插件方式支持 SQL 拦截和改写；
- 支持自增长主键、支持 Oracle 的 Sequence 机制。

11.2 Mycat 除了 Mysql 还支持哪些数据库？

答：mongodb、oracle、sqlserver、hive、db2、postgresql。

11.3 Mycat 目前有生产案例了么？

答：目前 Mycat 初步统计大概 600 家公司使用。

11.4 Mycat 稳定性与 Cobar 如何？

答：目前 Mycat 稳定性优于 Cobar，而且一直在更新，Cobar 已经停止维护，可以放心使用。

11.5 Mycat 支持集群么？

答：目前 Mycat 没有实现对多 Mycat 集群的支持，可以暂时使用 haproxy 来做负载，或者统计硬件负载。

11.6 Mycat 多主切换需要人工处理么？

答：Mycat 通过心跳检测，自主切换数据库，保证高可用性，无须手动切换。

11.7 Mycat 目前有多少人开发？

答：Mycat 目前开发全部是志愿者无偿支持，主要有以 leaderus 为首的 Mycat-Server 开始、以 rainbow 为首的 Mycat-web 开发、以海王星为首的产品发布及代码管理，还有以 Marshy 为首的推广。

11.8 Mycat 目前有哪些项目？

答：Mycat-Server：Mycat 核心服务；

Mycat-spider：Mycat 爬虫技术；

Mycat-ConfigCenter：Mycat 配置中心；

Mycat-BigSQL：Mycat 大数据处理（暂未更细）；

Mycat-Web：Mycat 监控及 web(新版开发中)；

Mycat-Balance：Mycat 集群负载（暂未更细）。

11.9 Mycat 最新的稳定版本是哪个到哪里下载？

答：打包代码：Mycat 最新稳定版是 1.5.1，1.6 为 alpha，下载地址是：

<https://github.com/MyCATapache/Mycat-download>。

文档：<https://github.com/MyCATapache/Mycat-doc>。

源码：<https://github.com/MyCATapache/Mycat-Server>。

11.10 Mycat 如何配置字符集？

答：在配置文件 server.xml 配置,默认配置为 utf8。

```
<system>
```

```
<property name="charset">utf8</property>
```

```
</system>
```

11.11 Mycat 后台管理监控如何使用？

答：9066 端口可以用 JDBC 方式执行命令，在界面上进行管理维护，也可以通过命令行查看命令行操作。

命令行操作是：`mysql -h127.0.0.1 -utest -ptest -P9066` 登陆，然后执行相应命令。

11.12 Mycat 主键插入后应用如何获取？

答：获得自增主键，插入记录后执行 `select last_insert_id()` 获取。

11.13 Mycat 如何启动与加入服务？

答：目前 Mycat 暂未封装加入服务，需要自己封装。

linux 环境为：

```
./mycat start 启动
```

```
./mycat stop 停止
```

```
./mycat console 前台运行
```

```
./mycat restart 重启服务
```

`./mycat pause` 暂停

`./mycat status` 查看启动状态

window 启动为:

直接双击运行 `startup_nowrap.bat` , 如果闪退用 `cmd` 模式运行查看日志。

11.14 Mycat 运行 sql 时经常阻塞或卡死是什么原因?

答: 如果出现执行 sql 语句长时间未返回, 或卡死, 请检查是否是虚拟机下运行或 cpu 为单核, 具体解决方式

请参考: <https://github.com/MyCATapache/MyCat-Server/issues/73>, 如果仍旧无法解决, 可以

暂时跳过, 目前有些环境阻塞卡死原因未知。

11.15 Mycat 中, 旧系统数据如何迁移到 Mycat 中?

答: 旧数据迁移目前可以手工导入, 在 mycat 中提取配置好分配规则及后端分片数据库, 然后通过 `dump`

或 `loaddata` 方式导入, 后续 Mycat 就做旧数据自动数据迁移工具。

11.16 Mycat 如何对旧分片数据迁移或扩容, 支持自动扩容么?

答: 目前除了一致性 hash 规则分片外其他数据迁移比较困难, 目前暂时可以手工迁移, 未提供自动迁移方

案, 具体迁移方案情况 Mycat 权威指南对应章节。

11.17 Mycat 支持批量插入吗?

答: 目前 Mycat1.3.0.3 以后支持多 values 的批量插入, 如 `insert into(xxx) values(xxx),(xxx)` 。

11.18 Mycat 支持多表 Join 吗?

答: Mycat 目前支持 2 个表 Join, 后续会支持多表 Join, 具体 Join 请看 Mycat 权威指南对应章节。

11.19 Mycat 启动报主机不存在的问题?

答: 需要添加 ip 跟主机的映射。

11.20 Mycat 连接会报无效数据源 (Invalid datasource) ?

答: 例如报错: `mysql> select * from company;`

`ERROR 3009 (HY000): java.lang.IllegalArgumentException: Invalid DataSource:0`

这类错误最常见是一些配置问题例如 schema.xml 中的 dataNode 的配置和实际不符合，请先仔细检查配置项，确保配置没有问题。如果不是配置问题，分析具体日志看出错原因，常见的有：

如果是应用连：在某些版本的 Mysql 驱动下连接 Mycat 会报错，可升级最新的驱动包试下。

如果是服务端控制台连，确认 mysql 是否开启远程连接权限，或防火墙是否设置正确，或者数据库 database 是否配置，或用户名密码是否正确。

11.21 Mycat 使用中如何提需求或 bug?

答：bug 或新需求可以到群里提问，同时最好到 github 发起以 issues：

<https://github.com/MyCATapache/Mycat-Server/issues>

11.22 Mycat 如何建表与创建存储过程?

答：注意注解中语句是节点的表请替换成自己表如 select 1 from 表，查出来的数据在那个节点往哪个节点建

存储过程

```
/*!mycat: sql=select 1 from 表 */ CREATE DEFINER=`root`@`%` PROCEDURE `proc_test`() BEGIN  
END ;
```

表：

```
/*!mycat: sql=select 1 from 表 */create table ttt(id int);
```

11.23 Mycat 目前有多少人维护?

答：目前初步统计有 10 人以上核心人员维护。

11.24 Mycat 支持的或者不支持的语句有哪些?

答：insert into，复杂子查询，3 表及其以上跨库 join 等不支持。

11.25 MycatJDBC 连接报 PacketTooBigException 异常

答：检查 mysqljdbc 驱动的版本，在使用 mycat1.3 和 mycat1.4 版本情况下，不要使用 jdbc5.1.37 和 38 版本的驱动，会出现如下异常报错：com.mysql.jdbc.PacketTooBigException: Packet for query is too large (60 > -1). You can change this value on the server by setting the max_allowed_packet' variable。建议使用 jdbc5.1.35 或者 36 的版本。

11.26 Mycat 中文乱码的问题

答：如果在使用 mycat 出现中文插入或者查询出现乱码，请检查三个环节的字符集设置：**1) 客户端环节**（应用程序、mysql 命令或图形终端工具）连接 mycat 字符集 **2) mycat 连接数据库的字符集** **3) 数据库**（mysql, oracle）字符集。这三个环节的字符集如果配置一致，则不会出现中文乱码，其中尤其需要注意的是客户端连接 mycat 时使用的连接字符集，通常的中文乱码问题一般都由此处设置不当引出。其中 mycat 内部默认使用 utf8 字符集，在最初启动连接数据库时，mycat 会默认使用 utf8 去连接数据库，当客户端真正连接 mycat 访问数据库时，mycat 会使用客户端连接使用的字符集修改它连接数据库的字符集，在 mycat 环境的管理 9066 端口，可以通过 show @@backend 命令查看后端数据库的连接字符集，通过 show @@connection 命令查看前端客户端的连接字符集。客户端的连接可以通过指定字符集编码或者发送 SET 命令指定连接 mycat 时 connection 使用的字符集，常见客户端连接指定字符集写法如下：

1) jdbcUrl=jdbc:mysql://localhost:8066/databaseName? characterEncoding=iso_1

2) SET character_set_client = utf8;用来指定解析客户端传递数据的编码

SET character_set_results = utf8;用来指定数据库内部处理时使用的编码

SET character_set_connection = utf8;用来指定数据返回给客户端的编码方式

3) mysql -utest -ptest -P8066 --default-character-set=gbk

11.27 Mycat 无法登陆 Access denied

答：Mycat 正常安装配置完成，登陆 mycat 出现以下错误：

```
[mysql@master ~]$ mysql -utest -ptest -P8066
```

```
ERROR 1045 (28000): Access denied for user 'test'@'localhost' (using password: YES)
```

请检查在 schema.xml 中的相关 dataHost 的 mysql 主机的登陆权限，一般都是因为配置的 mysql 的用户登陆权限不符合，mysql 用户权限管理不熟悉的请自己度娘。只有一种情况例外，mycat 和 mysql 主机都部署在同一台设备，其中主机 localhost 的权限配置正确，使用 -hlocalhost 能正确登陆 mysql 但是无法登陆 mycat 的情况，请使用 -h127.0.0.1 登陆，或者本地网络实际地址，不要使用 -hlocalhost，很多使用者反馈此问题，原因未明。

11.28 Mycat 的分片数据插入报异常 IndexOutOfBoundsException

答：在一些配置了分片策略的表进行数据插入时报错，常见的报错信息如下：

`java.lang.IndexOutOfBoundsException:Index:4,size:3` 这类报错通常由于分片策略配置不对引起，请仔细检查并理解分片策略的配置，例如：使用固定分片 hash 算法，`PartitionByLong` 策略，如果 `schema.xml` 里面设置的分片数量 `dataNode` 和 `rule.xml` 配置的 `partitionCount` 分片个数不一致，尤其是出现分片数量 `dataNode` 小于 `partitionCount` 数量的情况，插入数据就可能会报错。很多使用者都没有仔细理解文档中对分片策略的说明，用默认 `rule.xml` 配置的值，没有和自己实际使用环境进行参数核实就进行分片策略使用造成这类问题居多。

11.29 Mycat ER 分片子表数据插入报错

答：一般都是插入子表时出现不能找到父节点的报错。报错信息如：`[Err] 1064 - can't find (root) parent sharding node for sql:`。此类 ER 表的插入操作不能做为一个事务进行数据提交，如果父子表在一个事务中进行提交，显然在事务没有提交前子表是无法查到父表的数据的，因此就无法确定 sharding node。如果是 ER 关系的表在插入数据时不能在同一个事务中提交数据，只能分开提交。

11.30 Mycat 最大内存无法调整至 4G 以上

答：mycat1.4 的 JVM 使用最大内存调整如果超过 4G 大小，不能使用 `wrapper.java.maxmemory` 参数，需要使用 `wrapper.java.additional` 的写法，注意将 `wrapper.java.maxmemory` 参数注释，例如增加最大内存至 8G：`wrapper.java.additional.10=-Xmx8G`。

11.31 Mycat 使用过程中报错怎么办

答：记住无论什么时候遇到报错，如果不能第一时间理解报错的原因，首先就去看日志，无论是启动 (`wrapper.log`) 还是运行过程中 (`mycat.log`)，请相信良好的日志是编程查错的终极必杀技。日志如果记录信息不够，可以调整 `conf/log4j.xml` 中的 `level` 级别至 `debug`，所有的详细信息均会记录。另外如果在群里面提问，尽量将环境配置信息和报错日志提供清楚，这样别人才能快速帮你定位问题。

第 12 章 Mycat 性能测试指南

Mycat 自身提供了一套基准性能测试工具，这套工具可以用于性能测试、疲劳测试等，包括分片表插入性能测试、分片表查询性能测试、更新性能测试、全局表插入性能测试等基准测试工具。

这里需要说明的一点是，分片表的性能测试不同于普通单表，因为它的数据是分布在几个 Datahost 上的，因此插入和查询，都必需要特定的工具，才能做到多个节点同时负载请求，通过观察每个主机的负载，能够确定是否你的测试是合理和正确的。

大量测试表明，当带宽不是问题而且带宽没有占满，比如千兆网网络连接的 Mycat 和 MySQL 服务器，以及测试客户端，（通常个人电脑到服务器的连接为 100M），分片表的性能取决于后端部署 MySQL 的物理机的个数，比如每个 MySQL 的性能是 5 万 Tps，则 3 台理论上是 15 万，而 Mycat 能达到 80-95%之间，即 12 万以上。

关于带宽问题，是一个比较棘手的问题，通常需要监控交换机、MySQL 服务器、Mycat 服务器、以获取测试过程中的端口流量信息，才能确定是否带宽存在问题，另外，很多企业里，千兆交换机采用了百兆的普通网线的情况时有发生，防不胜防，所以，在不能控制的网络环境里，测试最大性能的目标通常无法实现。

另外，很多人测试的时候，并不知道 MySQL 直连的性能，因此无法正确比较 Mycat 的性能，所以，建议性能测试过程里，首先直连 MySQL 进行性能测试，可以同时直连多个 MYSQL 服务器，然后把测试结果累计，作为直连的性能指标，然后改为连接 Mycat 进行测试，这样的对比才是有价值的，当插件过大的时候，需要先排除是否存在 MySQL 冷热不均的现象，然后考虑 Mycat 性能调优。

测试工具在单独的包中，解压到任意机器中执行使用，跟 MyCAT Server 没有关联关系，此测试工具很强大，可以测试任意表，和任意数据库，测试工具下载：

<https://github.com/MyCATapache/Mycat-download> 目录下的 testtool.tar.gz 中。

解压后，在 bin 目录里运行文中的测试脚本。

标准插入性能测试脚本 test_stand_insert_perf.sh 支持任意表的定制化业务数据的随机生成功能了，在 sql 模板文件中用 \${int(1-100)} 这种变量，测试程序会随机生成符合要求的值并插入数据库。

```
./test_stand_insert_perf.sh jdbc:mysql://localhost:8066/TESTDB test test 10 file=mydata-create.sql
```

其中 mydata-create.sql 的内容如下：

```
total=10000000
```

```
sql=insert into my_table1 (....) values ('${date(yyyyMMddHHmmssSSS-[2014-2015]y)}-${int(0-9999)}ok${int(1111-9999)}xxx ','${char([0-9]2:2)} OPP_${enum(BJ,SH,WU,GZ)}_1',10,${int(10-999)},${int(10-99)},100,3,15,'${date(yyyyMMddHHmmssSSS-[2014-2015]y)}${char([a-f,0-9]8:8)} ','${phone(139-189)},2,${date(yyyyMMddHH-[2014-2015]y)},${date(HH:mm:ssSSS)},${int(100-1000)},${enum(0000,0001,0002)}')
```

目前支持的有以下类型变量：

Int:\${int(..)} 可以是,\${int(10-999)}或者,\${int(10,999)}前者表示从 10 到 999 的值，后者表示 10 或者 999

Date:日期如`{date(yyyyMMddHHmmssSSS-[2014-2015]y)}`表示从 2014 到 2015 年的时间，前面是输出格式，符合 Java 标准

Char:字符串`{char([0-9]2:2)}`表示从 0 到 9 的字符，长度为 2 位 (2:2)，`{char([a-f,0-9]8:8)}`表示从 a 到 f 以及 0 到 9 的字符串随机组成，定长为 8 位。

Enmu:枚举，表示从指定范围内获取一个值，`{enum(0000,0001,0002)}`，里面可以是任意字符串或数字等内容。

标准查询性能测试脚本 `test_stand_select_perf` 也支持 `sqlTemplate` 的变量方式，查询任意指定的 sql

`./test_stand_select_perf.sh jdbc:mysql://localhost:8066/TESTDB test test 10 100000 file=mysql-select.sql`

其中 `oppcall-select.sql` 的内容类似下面:

```
sql=select * from mytravelrecord where id = ${int(1-1000000)}
```

表明查询 id 为 1 到 1000000 之间的随机 SQL。

注意: Windows 下 `file=xxx.slq` 需要加引号:

```
test_stand_insert_perf.bat jdbc:mysql://localhost:8066/TESTDB test test 50 "file=oppcall.sql"
```

首先参考 MyCAT 性能调优，确保整个系统达到最优配置。

性能测试，建议先小规模压力预热 10-20 分钟，这是众所周知的 Java 的特性，越跑越快。

测试的硬件和网络条件:

- 建议至少 3 台服务器;
- MyCAT Server 一台;
- Mysql 一台;
- 带宽应该是至少 100M，建议千兆;
- 压力程序在另一台，压力程序的机器也可以由性能差的机器来代替。

有条件的话，分片库在不同的 MYSQL 实例上，如 20 个分片，每个 MYSQL 实例 7 个分片，而且最好有多台 MYSQL 物理机。

分片表的录入性能测试-T01

测试案例: 分片表的并发录入性能测试，测试 DEMO 中的 `travelrecord` 表，此表的基准 DDL 语句: `create travelrecord: create table travelrecord (id bigint not null primary key,user_id varchar(100),traveldate DATE, fee decimal,days int);`

此表的标准分片方式为基于 ID 范围的自动分片策略。Schema.xml 中配置如下:

```
<table name="travelrecord" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />
```


默认是 3 个分片，分片 ID 范围定义在 autopartition-long.txt 中，建议修改为以下或更大的数值范围分片，每个分片 500 万数据

```
# range start-end ,data node index  
0-2000000=0  
2000001-4000000=1  
4000001-6000000=2
```

根据自己的情况，可以每个分片放更多的数据，进行对比性能测试，当分片 index 增加时，注意 dataNode 也增加（dataNode= “dn1,dn2,dn3” ）。

测试的输入参数如下[jdbcurl] [user] [password] [threadpoolsize] [recordrange]:

Jdbcurl:连接 mycat 的地址，格式为 jdbc:mysql://localhost:8066/TESTDB

User 连接 Mycat 的用户名

Password:密码

Threadpoolsize:并发线程请求，可以在 50-2000 左右调整，看看哪种情况下的性能最好

Recordrang:插入的分片系列以及对应的 ID 范围，minId-maxId 然后逗号分开，对应多组分片的 ID 范围，如 0-200000,200001-400000,400001-600000，跟分片配置保持一致。

测试过程：

每次测试，建议先执行重建表的操作，以保证测试环境的一致性：

连接 mycat 8066 端口，在命令行执行下面的操作：

```
drop table travelrecord;
```

```
create table travelrecord (id bigint not null primary key,user_id varchar(100),traveldate DATE, fee decimal,days int);
```

先预测试：

执行命令：

```
test_stand_insert_perf jdbc:mysql://localhost:8066/TESTDB test test 100 "0-100M,100M1-200M,200M1-400"
```

MyCAT 温馨提示：并发线程数表明同时至少有多少个 Mysql 连接会被打开，当 SQL 不跨分片的时候，并发线程数 =MYSQL 连接数，在 Mycat conf/schema.xml 中，将 minCon 设置为 >= 并发连接数，这种情况下重启 MYCAT，会初始建立 minCon 个连接，并发测试结果更好，另外，也可以验证是否当前内存设置，以及 MYSQL 是否支持开启这么多连接，若无法支持，则 logs/mycat.log 日志中会有告警错误信息，建议测试过程中 tail -f logs/mycat.log 观察有无错误信息。另外，开启单独的 Mycat 管理窗口，mysql -utest -ptest -P9066 然后运行 show @@datasource 可以看到后端连接的使用情况。Show @@threadpool 可以看线程和 SQL 任务积压的情况。

也可以同时启动多个测试程序，在不同的机器上，并发进行测试，每个测试程序写入一个分片的数据范围，对于 1 个亿的数据插入测试来说，可能效果更好，毕竟单机并发线程 50 个左右已经差不多极限：

```
test_stand_insert_perf jdbc:mysql://localhost:8066/TESTDB test test 100 "0-100M"
est_stand_insert_perf jdbc:mysql://localhost:8066/TESTDB test test 100 100M1-200M"
```

全局表的查询性能测试 T02：

全局表自动在多个节点上同步插入，因此其插入性能有所降低，这里的插入表为 goods 表，执行的命令类似 T01 的测试。温馨提示：全局表是同时往多个分片上写数据，因此所需并发 MYSQL 数连接为普通表的 3 倍，最好的模式是全局表分别在多个 mysql 实例上。

建表语句：

```
drop table goods;
```

```
create table goods(id int not null primary key,name varchar(200),good_type tinyint,good_img_url
varchar(200),good_created date,good_desc varchar(500), price double);
```

```
test_globaltable_insert_perf.bat jdbc:mysql://localhost:8066/TESTDB test test 100 1000000
```

本机笔记本，4G 内存，数据库与 Mycat 以及测试程序都在一起，跑出来每秒 1000 多的插入速度：

分片表的查询性能测试 T03：

此测试可以在 T01 的集成上运行，先生成大量 travelrecord 记录，然后进行并发随机查询，此测试是在分片库上，基于分片的主键 ID 进行随机查询，返回单条记录，多线程并发随机执行 N 此记录查询，每次查询的记录主键 ID 是随机选择，在 maxID(参数)范围之内。

测试工具 test_stand_select_perf 的参数如下

```
[jdbcurl] [user] [password] [threadpoolsize] [executetimes] [maxId]
```

Executetimes：每个线程总共执行多少次随机查询，建议 1000 次以上

maxId：travelrecord 表的最大 ID，可以执行 select max(id) from travelrecord 来获取。

Example:

```
test_stand_select_perf.bat jdbc:mysql://localhost:8066/TESTDB test test 100 10000 50000
```

分片表的汇聚性能测试 T04:

此测试可以在 T01 的集成上运行，先生成大量 travelrecord 记录，然后进行并发随机查询，此测试执行分片库上的聚合、排序、分页的性能，SQL 如下：

```
select sum(fee) total_fee, days, count(id), max(fee), min(fee) from travelrecord group by days order by days desc limit ?
```

测试工具 test_stand_merge_sel_perf 的参数如下

[

```
jdbcurl] [user] [password] [threadpoolsize] [executetimes] [limit]
```

Executetimes: 每个线程总共执行多少次随机查询，建议 1000 次以上

limit: 分页返回的记录个数，必须大于 30

Example:

```
test_stand_merge_sel_perf.bat jdbc:mysql://localhost:8066/TESTDB test test 10 100 100
```

分片表的更新性能测试 T05:

此测试可以在 T01 的集成上运行，先生成大量 travelrecord 记录，然后进行并发更新操作，

```
update travelrecord set user =? ,traveldate=?,fee=?,days=? where id=?
```

测试工具 test_stand_update_perf 的参数如下

```
[jdbcurl] [user] [password] [threadpoolsize] [record]
```

record: 总共修改多少条记录，>5000

Example:

```
test_stand_update_perf.bat jdbc:mysql://localhost:8066/TESTDB test test 10 10000
```

高级进阶篇

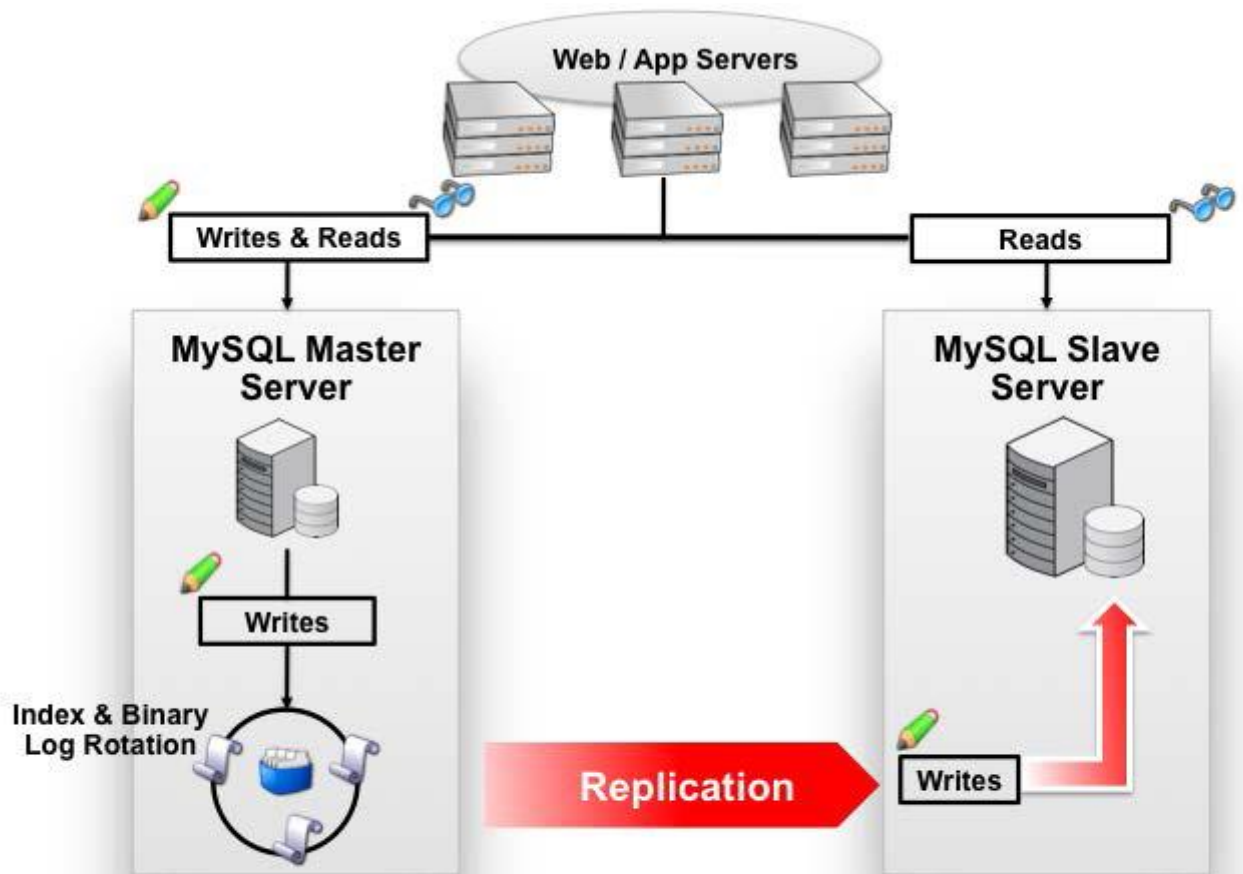
第 1 章读写分离

1.1 MySQL 主从复制的几种方案

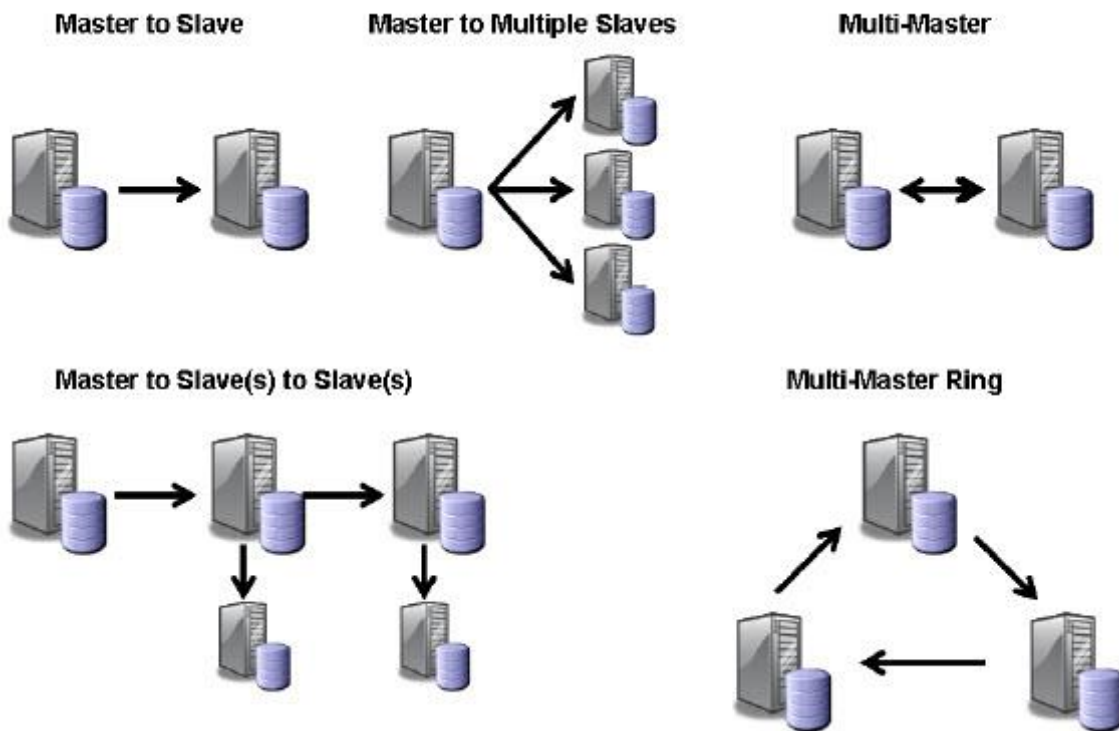
数据库读写分离对于大型系统或者访问量很高的互联网应用来说，是必不可少的一个重要功能。

从数据库的角度来说，对于大多数应用来说，从集中到分布，最基本的一个需求不是数据存储的瓶颈，而是在于计算的瓶颈，即 SQL 查询的瓶颈，我们知道，正常情况下，Insert SQL 就是几十个毫秒的时间内写入完成，而系统中的大多数 Select SQL 则要几秒到几分钟才能有结果，很多复杂的 SQL，其消耗服务器 CPU 的能力超强，不亚于死循环的威力。在没有读写分离的系统上，很可能高峰时段的一些复杂 SQL 查询就导致数据库服务器 CPU 爆表，系统陷入瘫痪，严重情况下可能导致数据库崩溃。因此，从保护数据库的角度来说，我们应该尽量避免没有主从复制机制的单节点数据库。

对于 MySQL 来说，标准的读写分离是主从模式，一个写节点 Master 后面跟着多个读节点，读节点的数量取决于系统的压力，通常是 1-3 个读节点的配置，如下图所示：



MySQL 支持更多的主从复制的拓扑关系，如下图所示，但通常我们不会采用双向主从同步以及环状的拓扑：



MySQL 主从复制的原理如下：

第一步是在主库上记录二进制日志（稍后介绍如何设置）。在每次准备提交事务完成数据更新前，主库将数据更新的事件记录到二进制日志中。MySQL 会按事务提交的顺序而非每条语句的执行顺序来记录二进制日志。在记录二进制日志后，主库会告诉存储引擎可以提交事务了。下一步，备库将主库的二进制日志复制到其本地的中继日志中。首先，备库会启动一个工作线程，称为 I/O 线程，I/O 线程跟主库建立一个普通的客户端连接，然后在主库上启动一个特殊的二进制转储(binlog dump、线程（该线程没有对应的 SQL 命令），这个二进制转储线程会读取主库上二进制日志中的事件。它不会对事件进行轮询。如果该线程追赶上了主库，它将进入睡眠状态，直到主库发送信号量通知其有新的事件产生时才会被唤醒，备库 I/O 线程会将接收到的事件记录到中继日志中。

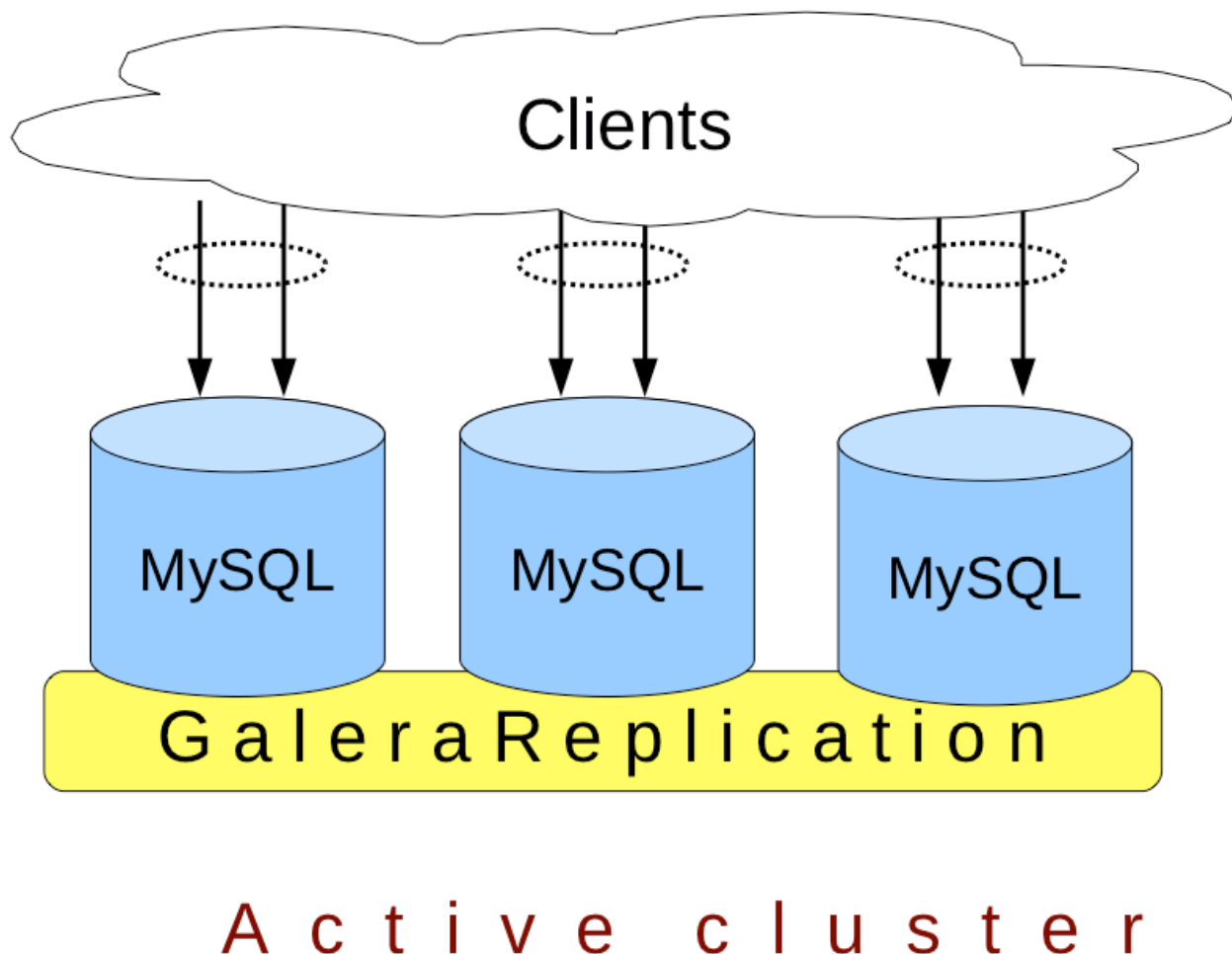
备库的 SQL 线程执行最后一步，该线程从中继日志中读取事件并在备库执行，从而实现备库数据的更新。当 SQL 线程追赶上 I/O 线程时，中继日志通常已经在系统缓存中，所以中继日志的开销很低。SQL 线程执行的事件也可以通过配置选项来决定是否写入其自己的二进制日志中，它对于我们稍后提到的场景非常有用。这种复制架构实现了获取事件和重放事件的解耦，允许这两个过程异步进行。也就是说 I/O 线程能够独立于 SQL 线程之外工作。但这种架构也限制了复制的过程，其中最重要的一点是在主库上并发运行的查询在备库只能串行化执行，因为只有一个 SQL 线程来重放中继日志中的事件。后面我们将会看到，这是很多工作负载的性能瓶颈所在。虽然有一些针对该问题的解决方案，但大多数用户仍然受制于单线程。MySQL5.6 以后，提供了基于 GTID 多开启多线程同步复制的方案，即每个库有一个单独的(sql thread)

进行同步复制，这将大大改善 MySQL 主从同步的数据延迟问题，配合 Mycat 分片，可以更好的将一个超级大表的数据同步的时延降低到最低。此外，用 GTID 避免了在传送 binlog 逻辑上依赖文件名和物理偏移量，能够更好的支持自动容灾切换，对运维人员来说应该是一件令人高兴的事情，因为传统的方式里，你需要找到 binlog 和 POS 点，然后 change master to 指向，而不是很有经验的运维，往往会将其找错，造成主从同步复制报错，在 mysql5.6 里，无须再知道 binlog 和 POS 点，需要知道 master 的 IP、端口，账号密码即可，因为同步复制是自动的，mysql 通过内部机制 GTID 自动找点同步。

即使是并发复制机制、仍然无法避免主从数据库的数据瞬间不同步的问题，因此又有了一种增强的方案，即 galera for mysql、percona-cluster 或者 mariadb cluster 等集群机制，他们是一种多主同步复制的模式，可以在任意节点上进行读写、自动控制成员，自动删除故障节点、自动加入节点、真正给予行级别的并发复制等强大能力！

下图是其原理图，通常是采用 3 个 MySQL 节点作为一个 Cluster，即提供了 3 倍的数据库读的并发能力。galera for mysql 集群这种方式，是牺牲了数据的写入速度，以换取最大程度的数据并发访问能力，类似

Mycat 里的全局表，并且保证了数据同时存在几个有效的副本，从而具有非常高的可靠性，因此在某种程度上，可以替代 Oracle 的一些关键场景，**目前开源中间件中，只有 Mycat 很完美的支持了 galera for mysql 集群模式。



1.2 MySQL 主从复制的几个问题

MySQL 主从复制并不完美，存在着几个由来已久的问题，首先一个问题是复制方式：

- 基于 SQL 语句的复制(statement-based replication, SBR);
- 基于行的复制(row-based replication, RBR);
- 混合模式复制(mixed-based replication, MBR);
- 基于 SQL 语句的方式最古老的方式，也是目前默认的复制方式，后来的两种是 MySQL 5 以后才出现的复制方式。

RBR 的优点：

- 任何情况都可以被复制，这对复制来说是最安全可靠的；

- 和其他大多数数据库系统的复制技术一样；
- 多数情况下，从服务器上的表如果有主键的话，复制就会快了很多。

RBR 的缺点：

- binlog 大了很多；
- 复杂的回滚时 binlog 中会包含大量的数据；
- 主服务器上执行 UPDATE 语句时，所有发生变化的记录都会写到 binlog 中，而 SBR 只会写一次，这会导致频繁发生 binlog 的并发写问题；
- 无法从 binlog 中看到都复制了写什么语句。

SBR 的优点：

- 历史悠久，技术成熟；
- binlog 文件较小；
- binlog 中包含了所有数据库更改信息，可以据此来审核数据库的安全等情况；
- binlog 可以用于实时的还原，而不仅仅用于复制；
- 主从版本可以不一样，从服务器版本可以比主服务器版本高。

SBR 的缺点：

- 不是所有的 UPDATE 语句都能被复制，尤其是包含不确定操作的时候；
- 复制需要进行全表扫描(WHERE 语句中没有使用到索引)的 UPDATE 时，需要比 RBR 请求更多的行级锁；
- 对于一些复杂的语句，在从服务器上的耗资源情况会更严重，而 RBR 模式下，只会对那个发生变化的记录产生影响；
- 数据表必须几乎和主服务器保持一致才行，否则可能会导致复制出错；
- 执行复杂语句如果出错的话，会消耗更多资源。

选择哪种方式复制，会影响到复制的效率以及服务器的损耗，甚以及数据一致性问题，目前其实没有很好的客观手段去评估一个系统更适合哪种方式的复制，Mycat 未来希望能通过智能调优模块给出更科学的建议。

第二个问题是关于主从同步的监控问题，Mysql 有主从同步的状态信息，可以通过命令 `show slave status` 获取，除了获知当前是否主从同步正常工作，另外一个重要指标就是 `Seconds_Behind_Master`，从字面理解，它表示当前 MySQL 主从数据的同步延迟，单位是秒，但这个指标从 DBA 的角度并不能简单的理解为延迟多少秒，感兴趣的同学可以自己去看，但对于应用来说，简单的认为是主从同步的时间差就可以了，另外，当主从同步

停止以后，重新启动同步，这个数值可能会是几万秒，取决于主从同步停止的时间长短，我们可以认为数据此时有很多天没有同步了，而这个数值越接近零，则说明主从同步延迟最小，我们可以采集这个指标并汇聚曲线图，来分析我们的数据库的同步延迟曲线，然后根据此曲线，给出一个合理的阈值，主从同步的时延小于阈值时，我们认为从库是同步的，此时可以安全的从从库读取数据。Mycat 未来将支持这种优化，让应用更加可靠的读取到预期的从库数据。

1.3 Mycat 支持的读写分离

1. 配置 mysql 端主从的数据自动同步，mycat 不负任何的数据同步问题。
2. Mycat 配置读写分离，具体参数参加前面章节。

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="1"
  writeType="0" dbType="mysql" dbDriver="native">
  <heartbeat>select user()</heartbeat>
  <!-- can have multi write hosts -->
  <writeHost host="hostM1" url="localhost:3306" user="root" password="123456">
    <!-- can have multi read hosts -->
    <readHost host="hostS1" url="localhost2:3306" user="root" password="123456"
      weight="1" />
  </writeHost>
</dataHost>
```

或者

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="1"
  writeType="0" dbType="mysql" dbDriver="native">
  <heartbeat>select user()</heartbeat>
  <!-- can have multi write hosts -->
  <writeHost host="hostM1" url="localhost:3306" user="root" password="123456">
  </writeHost>

  <writeHost host="hostS1" url="localhost:3307" user="root" password="123456">
  </writeHost>
</dataHost>
```

以上两种取模第一种当写挂了读不可用，第二种可以继续使用，事务内部的一切操作都会走写节点，所以读操作不要加事务，如果读延时较大，使用根据主从延时的读写分离，或者强制走写节点：

1.3.1 应用强制走写：

一个查询 SQL 语句以/*balance*/注解来确定其是走读节点还是写节点。

1.6 以后添加了强制走读走写处理：

强制走从：

```
/*!mycat:db_type=slave*/ select * from travelrecord
```

```
/*#mycat:db_type=slave*/ select * from travelrecord
```

强制走写：

```
/*#mycat:db_type=master*/ select * from travelrecord
```

```
/*!mycat:db_type=master*/ select * from travelrecord
```

1.3.2 根据主从延时切换：

1.4 开始支持 MySQL 主从复制状态绑定的读写分离机制，让读更加安全可靠，配置如下：

MyCAT 心跳检查语句配置为 show slave status，dataHost 上定义两个新属性：switchType="2" 与 slaveThreshold="100"，此时意味着开启 MySQL 主从复制状态绑定的读写分离与切换机制，Mycat 心跳机制通过检测 show slave status 中的 "Seconds_Behind_Master", "Slave_IO_Running", "Slave_SQL_Running" 三个字段来确定当前主从同步的状态以及 Seconds_Behind_Master 主从复制时延，当 Seconds_Behind_Master>slaveThreshold 时，读写分离筛选器会过滤掉此 Slave 机器，防止读到很久之前的旧数据，而当主节点宕机后，切换逻辑会检查 Slave 上的 Seconds_Behind_Master 是否为 0，为 0 时则表示主从同步，可以安全切换，否则不会切换。

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="0"
    writeType="0" dbType="mysql" dbDriver="native" switchType="2"
slaveThreshold="100">
    <heartbeat>show slave status </heartbeat>
    <!-- can have multi write hosts -->
    <writeHost host="hostM1" url="localhost:3306" user="root"
        password="123456">
    </writeHost>
    <writeHost host="hostS1" url="localhost:3316" user="root"
        password="123456" />
</dataHost>
```

1.4.1 开始支持 MySQL 集群模式，让读更加安全可靠，配置如下：

MyCAT 心跳检查语句配置为 show status like 'wsrep%' ，

dataHost 上定义两个新属性： switchType="3"

此时意味着开启 MySQL 集群复制状态绑定的读写分离与切换机制，Mycat 心跳机制通过检测集群复制时延时，如果延时过大或者集群出现节点问题不会负载改节点。

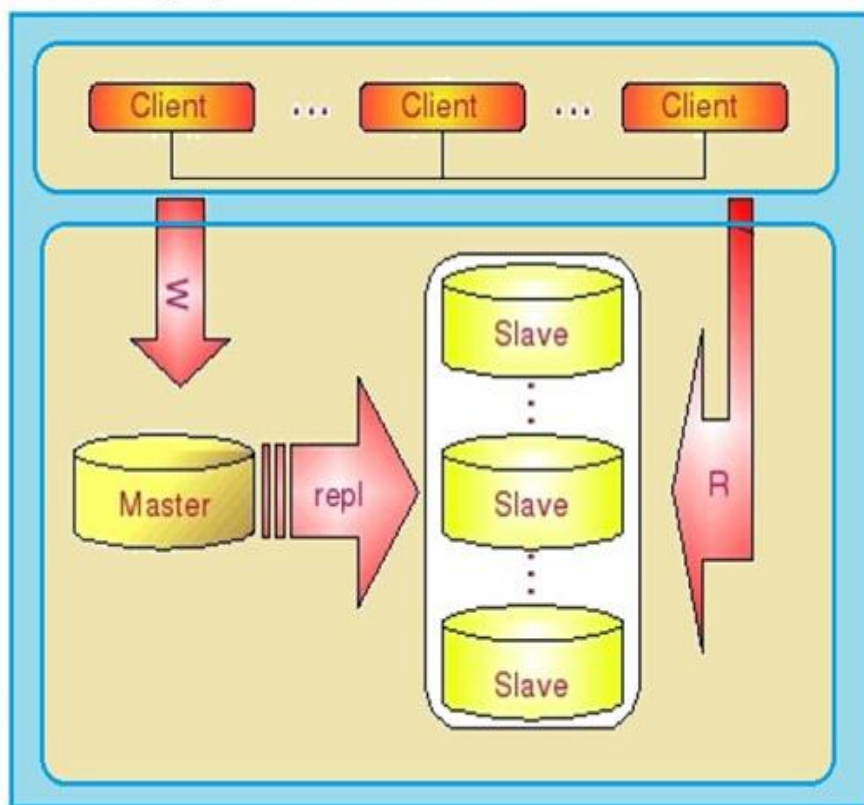
```
dataHost name="localhost1" maxCon="1000" minCon="10" balance="0"
  writeType="0" dbType="mysql" dbDriver="native" switchType="3" >
    <heartbeat> show status like 'wsrep%' </heartbeat>
    <writeHost host="hostM1" url="localhost:3306"
user="root"password="123456">
      </writeHost>
    <writeHost
host="hostS1"url="localhost:3316"user="root"password="123456" ></writeHost>
</dataHost>
```

第 2 章 高可用与集群

2.1 MySQL 高可用的几种方案

首先我们看看 MySQL 高可用的几种方案：

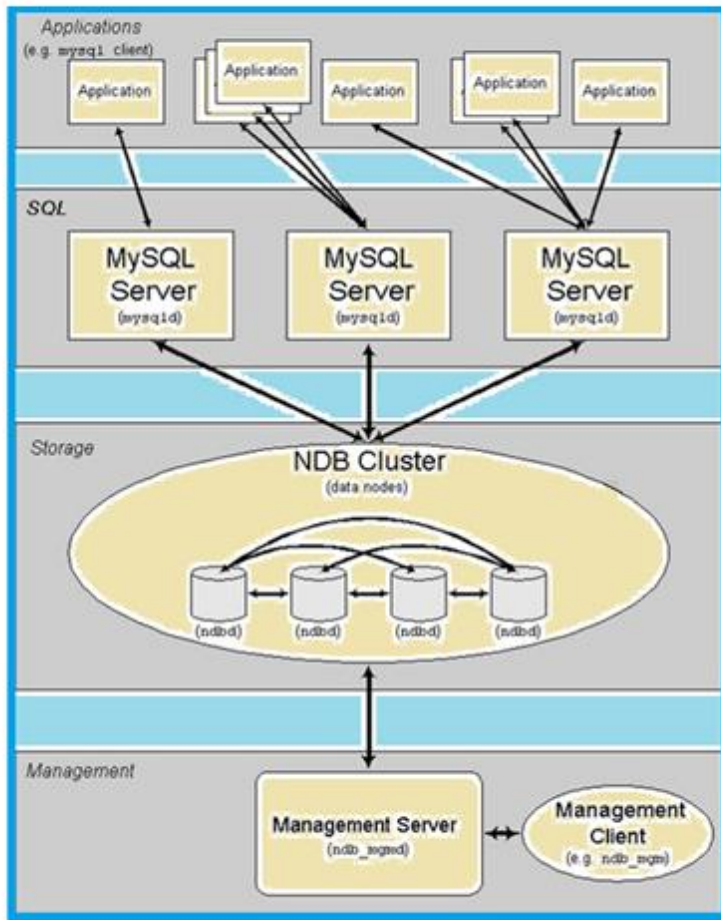
主从复制 + 读写分离



客户端通过 Master 对数据库进行写操作，slave 端进行读操作，并可进行备份。Master 出现问题后，可以手动将应用切换到 slave 端

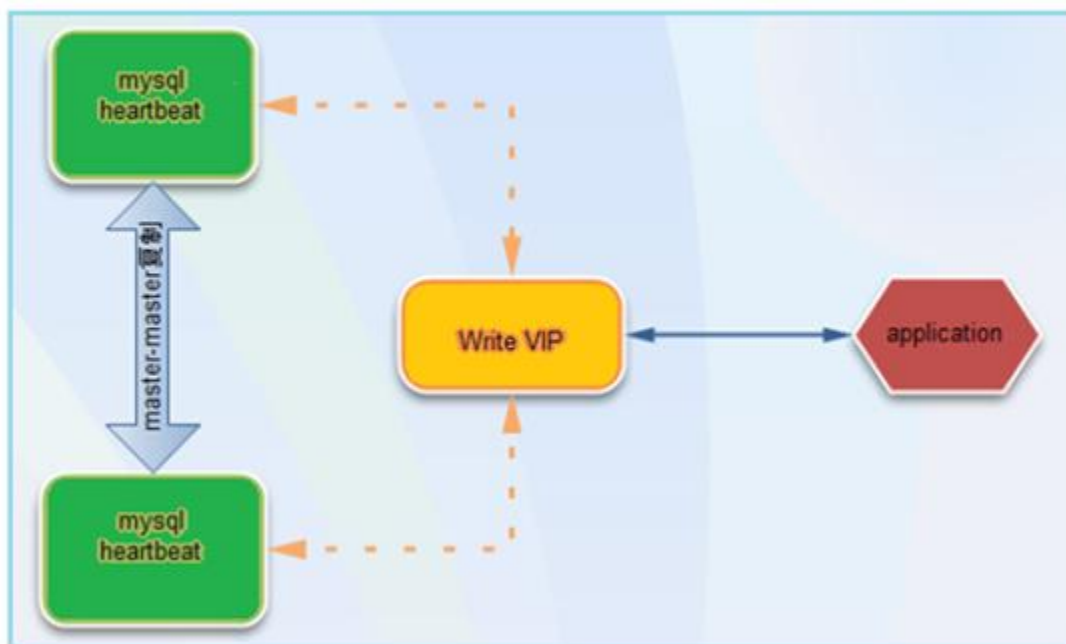
对于数据实时性要求不是特别严格的应用，只需要通过廉价的 pc server 来扩展 Slave 的数量，将读压力分散到多台 Slave 的机器上面，即可通过分散单台数据库服务器的读压力来解决数据库端的读性能瓶颈，毕竟在大多数数据库应用系统中的读压力还是要比写压力大很多。这在很大程度上解决了目前很多中小型网站的数据库压力瓶颈问题，甚至有些大型网站也在使用类似方案解决数据库瓶颈。

MySQL Cluster



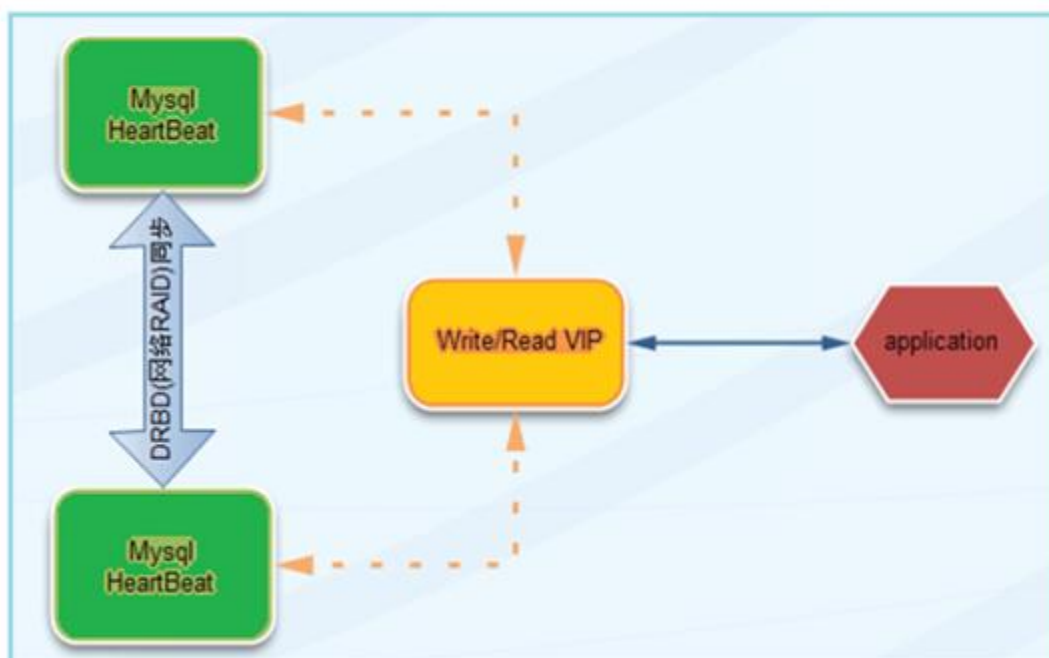
MySQL Cluster 由一组成计算机构成，每台计算机上均运行着多种进程，包括 MySQL 服务器，NDB Cluster 的数据节点，管理服务器，以及（可能）专门的数据访问程序。NDB” 是一种“内存中”的存储引擎，它具有可用性高和数据一致性好的特点。MySQL Cluster 要实现完全冗余和容错，至少需要 4 台物理主机，其中两个为管理节点。MySQL Cluster 使用不那么广泛，除了自身构架因素、适用的业务有限之外，另一个重要的原因是其安装配置管理相对复杂繁琐，总共有几十个操作步骤，需要 DBA 花费几个小时才能搭建或完成。重启 MySQL Cluster 数据库的管理操作之前需要执行 46 个手动命令，需要耗费 DBA 2.5 小时的时间，而依靠 MySQL Cluster Manager 只需一个命令即可完成，但 MySQL Cluster Manager 仅作为商用 MySQL Cluster 运营商级版本 (CGE) 数据库的一部分提供，需要购买。其官方的说明，若应用中的 SQL 操作为主键数据库访问，包含一些 JOIN 操作而非对整个表执行常规扫描和 JOIN 而返回数万行数据，则适合 Cluster，否则不合适，从这一条限制来看，表明大多数业务场景并不合适 MySQL Cluster，业内有资深人士也凭评价：NDB 不适合大多数业务场景，而且有安全问题。

HeartBeat+双主复制



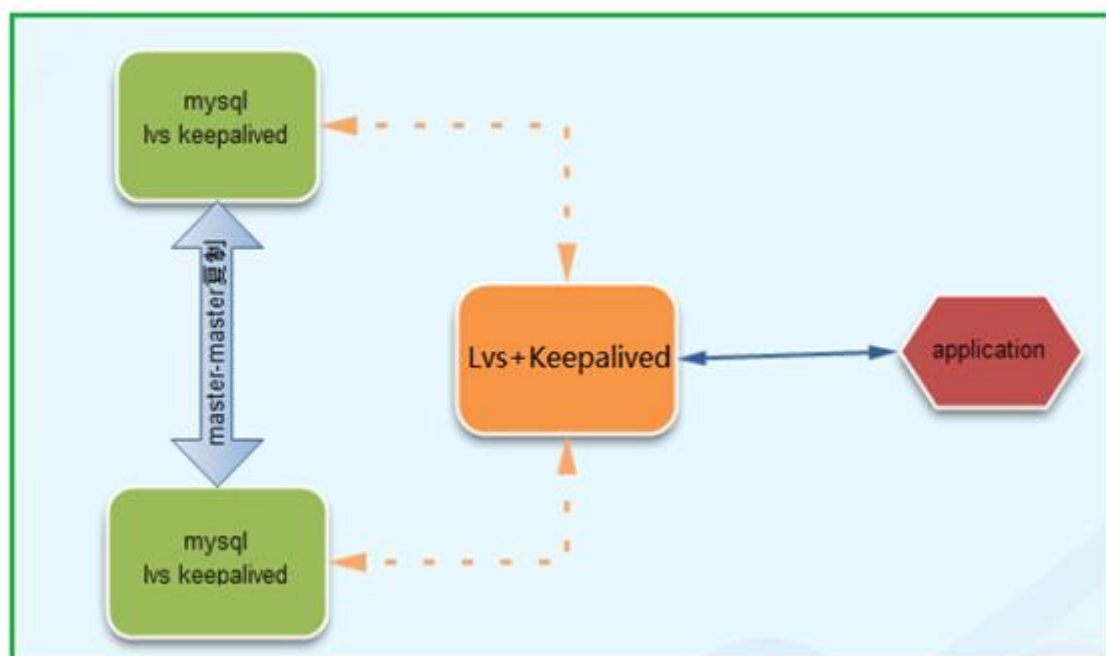
heartbeat 是 Linux-HA 工程的一个组件,heartbeat 最核心的包括两个部分：心跳监测和资源接管。在指定的时间内未收到对方发送的报文，那么就认为对方失效，这时需启动资源接管模块来接管运行在对方主机上的资源或者服务。

HeartBeat+DRBD+MySQL



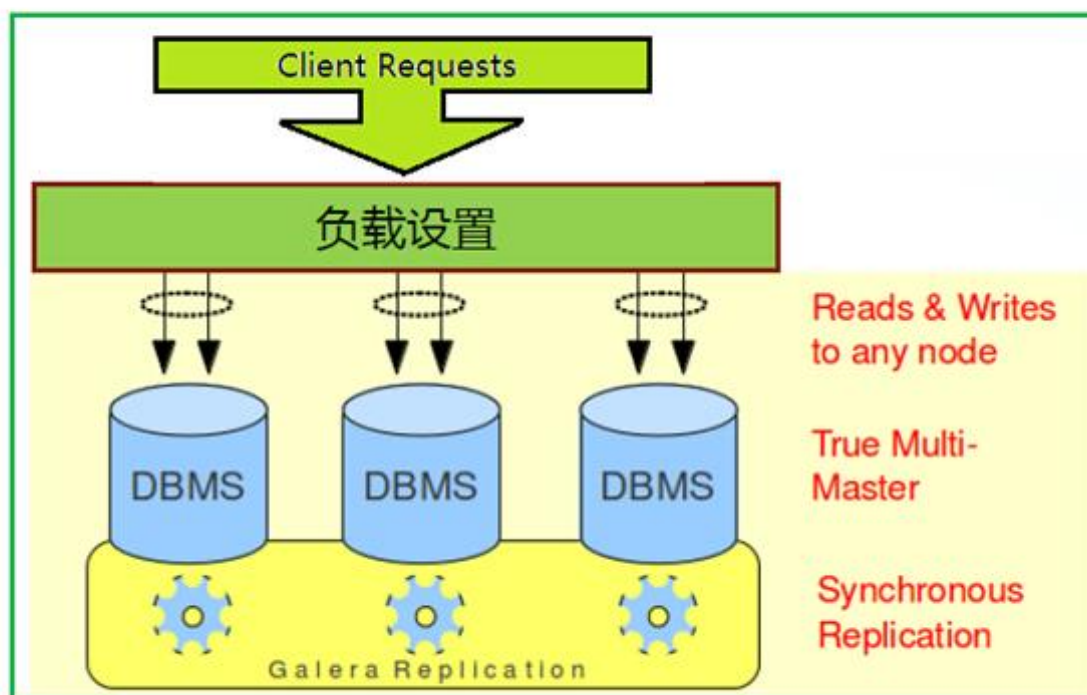
DRBD 是通过网络来实现块设备的数据镜像同步的一款开源 Cluster 软件，它自动完成网络中两个不同服务器上的磁盘同步，相对于 binlog 日志同步，它是更底层的磁盘同步，理论上 DRDB 适合很多文件型系统的高可用。

Lvs+Keepalived+双主复制



Lvs 是一个虚拟的服务器集群系统，可以实现 LINUX 平台下的简单负载均衡。keepalived 是一个类似于 layer3, 4 & 5 交换机制的软件，主要用于主机与备机的故障转移，这是一种适用面很广的负载均衡和高可用方案，最常用于 Web 系统。

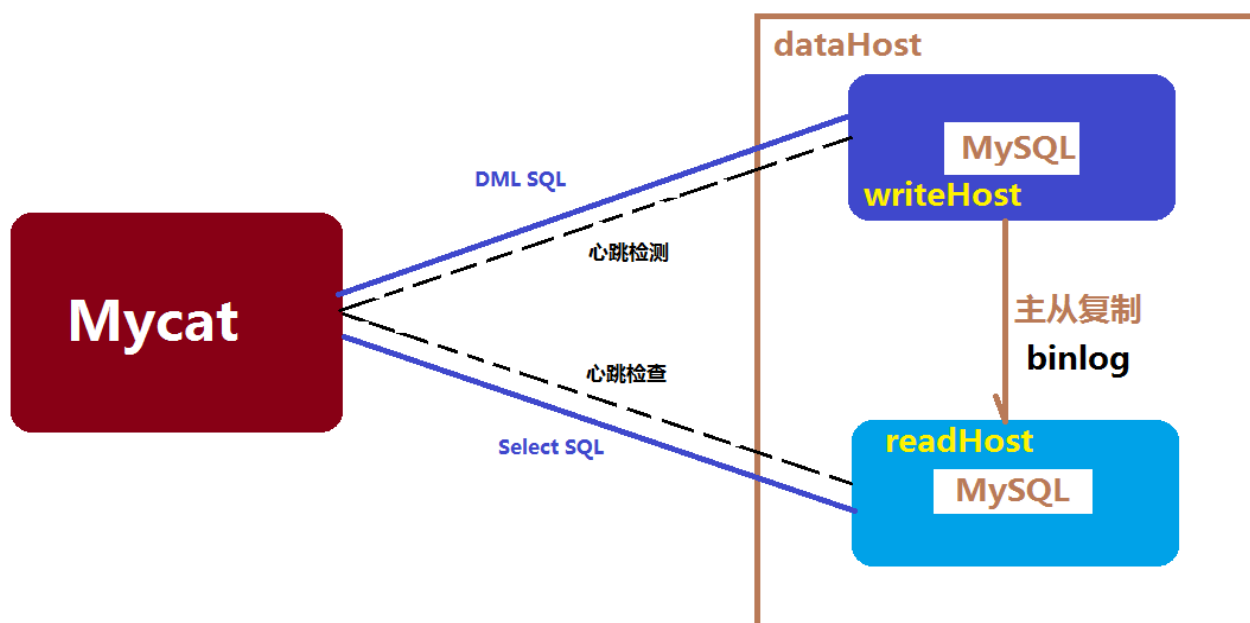
MariaDB Galera



这种 gluster 模式可以说是全新的一种高可用方案，前面也提到其优点，它的缺点不多，不支持 XA，不支持 Lock Table，只能用 InnoDB 引擎。

2.2 Mycat 高可用方案

Mycat 作为一个代理层中间件，Mycat 系统的高可用涉及到 Mycat 本身的高可用以及后端 MySQL 的高可用，前面章节所讲的 MySQL 高可用方案都可以在此用来确保 Mycat 所连接的后端 MySQL 服务的高可用性。在大多数情况下，建议采用标准的 MySQL 主从复制高可用性配置并交付给 Mycat 来完成后端 MySQL 节点的主从自动切换。

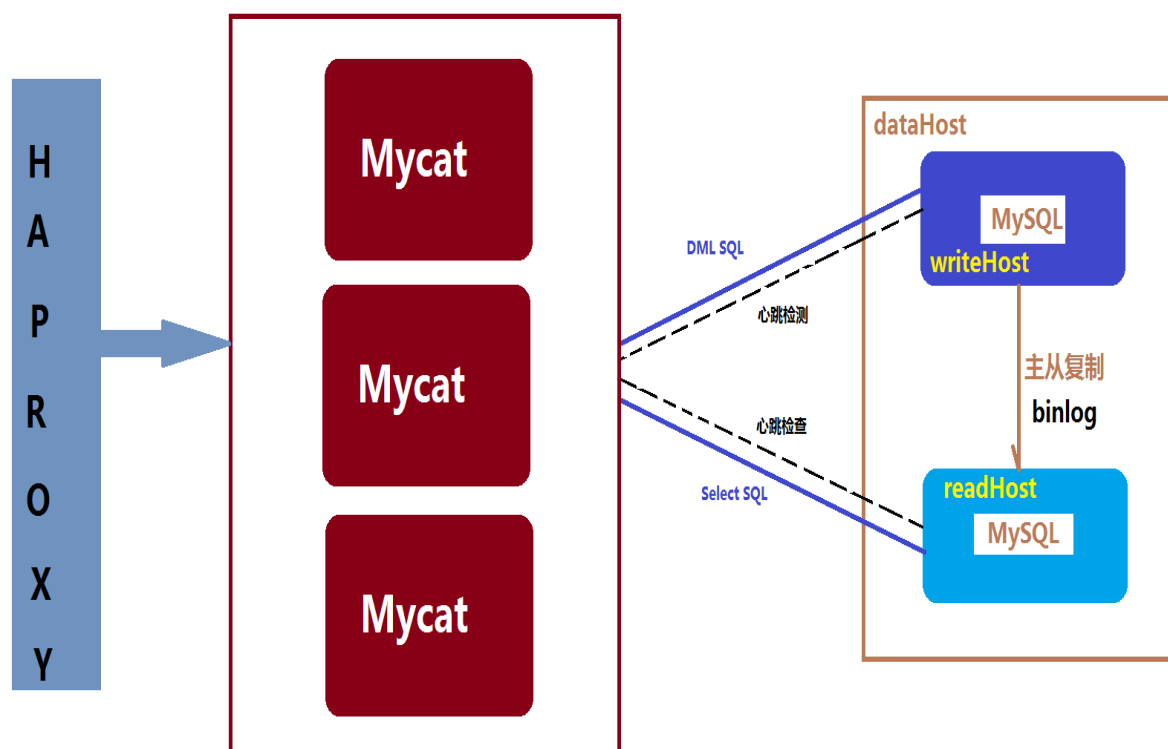


如图所示，MySQL 节点开启主从复制的配置方案，并将主节点配置为 Mycat 的 dataHost 里的 writeNode，从节点配置为 readNode，同时 Mycat 内部定期对一个 dataHost 里的所有 writeHost 与 readHost 节点发起心跳检测，正常情况下，Mycat 会将第一个 writeHost 作为写节点，所有的 DML SQL 会发送给此节点，若 Mycat 开启了读写分离，则查询节点会根据读写分离的策略发往 readHost(+writeHost)执行，当一个 dataHost 里面配置了两个或多个 writeHost 的情况下，如果第一个 writeHost 宕机，则 Mycat 会在默认的 3 次心跳检查失败后，自动切换到下一个可用的 writeHost 执行 DML SQL 语句，并在 conf/dnindex.properties 文件里记录当前所用的 writeHost 的 index（第一个为 0，第二个为 1，依次类推），注意，此文件不能删除和擅自改变，除非你深刻理解了它的作用以及你的目的。

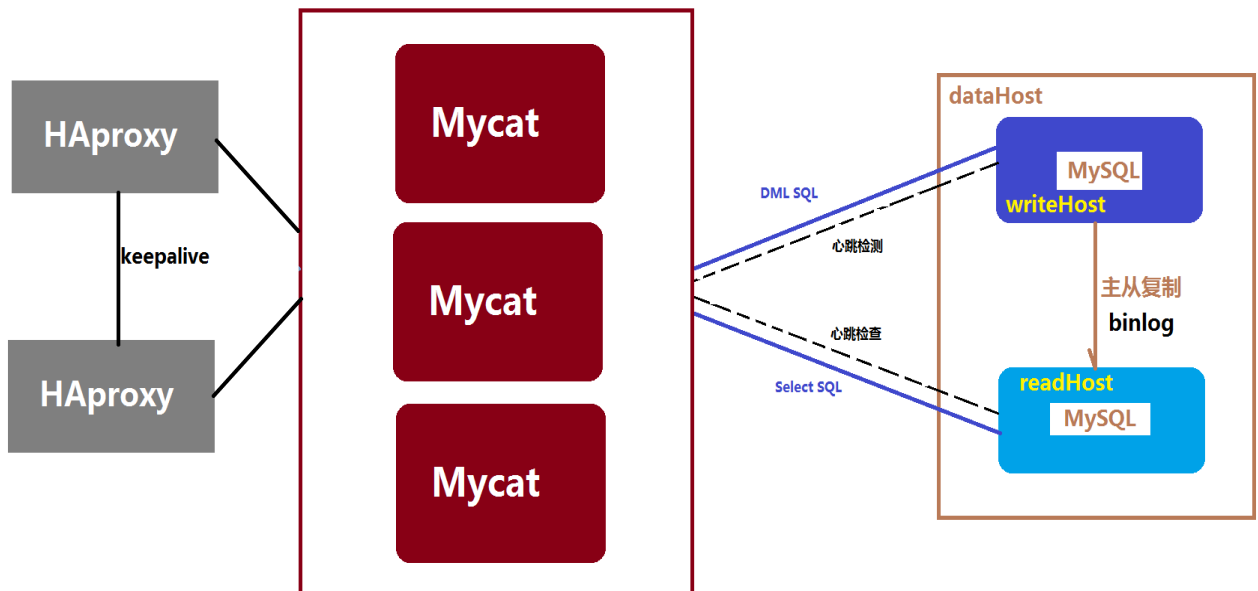
那么问题来了，当原来配置的 MySQL 写节点宕机恢复以后，怎么重新加入 Mycat，要不要恢复为原来的写节点？关于这个问题，我们也曾与 DBA 讨论很久，最终的建议方案是，保持现有状态不变，改旗易帜，恢复后的 MySQL 节点作为从节点，跟随新的主节点，重新配置主从同步，原先跟随该节点做同步的其他节点也同样换帅，重新配置同步源，这些节点的数据手工完成同步以后，再加入 Mycat 里。目前 1.3 版本的 Mycat 还没有实现监控 MySQL 主从同步状态的功能，因此这个过程里，DBA 可以先修改 MySQL 的密码，让 Mycat 无法链接故障服务器，等同步完成以后，恢复密码，这样 Mycat 就自动重新将修复好的 MySQL 纳管进来了。

说完了 MySQL 部分，接下来我们看看 Mycat 自身的高可用性，由于 Mycat 自身是属于无状态的中间件（除了主从切换过程中记录的 dnindex.properties 文件），因此 Mycat 很容易部署为集群方式，提供高可用方案。原先有规划 Mycat-balance 组件，专门用于 Mycat 负载均衡，但由于缺乏志愿者，也没有经过生产实践验证，

因此暂时不建议使用，官方建议是采用基于硬件的负载均衡器或者软件方式的 HAProxy，HAProxy 相比 LVS 的使用要简单很多，功能方面也很丰富，免费开源，稳定性也是非常好，可以与 LVS 相媲美，根据官方文档，HAProxy 可以跑满 10Gbps-New benchmark of HAProxy at 10 Gbps using Myricom' s 10GbE NICs (Myri-10G PCI-Express)，这个作为软件级负载均衡，也是比较惊人的，下图是 HAProxy+Mycat 集群+MySQL 主从所组成的高可用性方案：



如果还担心 HAProxy 的稳定性和单点问题，则可以用 keepalived 的 VIP 的浮动功能，加以强化：



2.3 Galary Cluster 配置

Mycat1.4.1 开始支持 galary cluster 集群的配置，提高心跳可用。

配置如下：

1.4.1 开始支持 MySQL 集群模式，让读更加安全可靠，配置如下：

MyCAT 心跳检查语句配置为 `show status like 'wsrep%'` ，

dataHost 上定义两个新属性： `switchType="3"`

此时意味着开启 MySQL 集群复制状态绑定的读写分离与切换机制，Mycat 心跳机制通过检测集群复制时延时，如果延时过大或者集群出现节点问题不会负载改节点。

```
dataHost name="localhost1" maxCon="1000" minCon="10" balance="0"
  writeType="0" dbType="mysql" dbDriver="native" switchType="3" >
    <heartbeat> show status like 'wsrep%' </heartbeat>
    <writeHost host="hostM1" url="localhost:3306"
user="root"password="123456">
      </writeHost>
    <writeHost
host="hostS1"url="localhost:3316"user="root"password="123456" ></writeHost>
</dataHost>
```

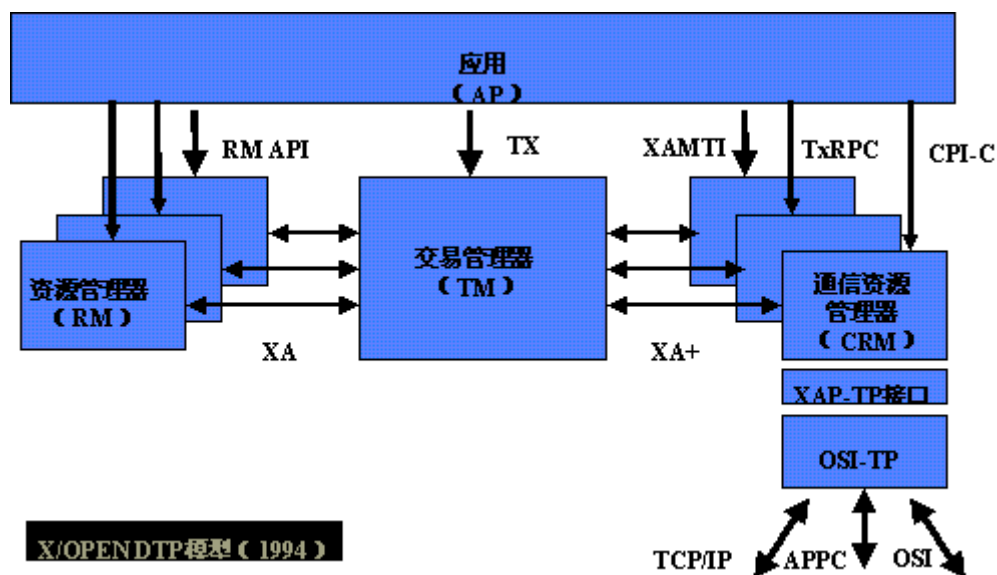
第 3 章 事务支持

3.1 Mycat 里的数据库事务

Mycat 目前没有出来跨分片的事务强一致性支持，目前单库内部可以保证事务的完整性，如果跨库事务，在执行的时候任何分片出错，可以保证所有分片回滚，但是一旦应用发起 commit 指令，无法保证所有分片都成功，考虑到某个分片挂的可能性不大所以称为弱 xa。

3.2 XA 事务原理

分布式事务处理（Distributed Transaction Processing，DTP）指一个程序或程序段，在一个或多个资源如数据库或文件上为完成某些功能的执行过程的集合，分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）。X/Open 组织（即现在的 Open Group）定义了分布式事务处理模型。X/Open DTP 模型（1994）包括应用程序（AP）、事务管理器（TM）、资源管理器（RM）、通信资源管理器（CRM）四部分。一般，常见的事务管理器（TM）是交易中间件，常见的资源管理器（RM）是数据库，常见的通信资源管理器（CRM）是消息中间件，下图是 X/Open DTP 模型：



一般的编程方式是这样的：

- 配置 TM，通过 TM 或者 RM 提供的方式，把 RM 注册到 TM。可以理解为给 TM 注册 RM 作为数据源。

一个 TM 可以注册多个 RM。

- AP 从 TM 获取资源管理器的代理（例如：使用 JTA 接口，从 TM 管理的上下文中，获取出这个 TM 所管理的 RM 的 JDBC 连接或 JMS 连接）
- AP 向 TM 发起一个全局事务。这时，TM 会通知各个 RM。XID（全局事务 ID）会通知到各个 RM。
- AP 通过 1 中获取的连接，直接操作 RM 进行业务操作。这时，AP 在每次操作时把 XID(包括所属分支的信息)传递给 RM，RM 正是通过这个 XID 与 2 步中的 XID 关联来知道操作和事务的关系的。
- AP 结束全局事务。此时 TM 会通知 RM 全局事务结束。
- 开始二段提交，也就是 prepare - commit 的过程。
- XA 协议(XA Specification), 指的是 TM 和 RM 之间的接口，其实这个协议只是定义了 xa_ 和 ax_ 系列的函数原型以及功能描述、约束和实施规范等。至于 RM 和 TM 之间通过什么协议通信，则没有提及，目前知名的数据库，如 Oracle, DB2 等，都是实现了 XA 接口的，都可以作为 RM。Tuxedo、TXseries 等事务中间件可以通过 XA 协议跟这些数据源进行对接。JTA(Java Transaction API)是符合 X/Open DTP 的一个编程模型，事务管理和资源管理器支架也是用了 XA 协议。

下面两个图片分别给出了 XA 成功与失败的两种情况，首先是 XA 事务成功的流程图：



然后，是 XA 事务失败的流程图：



XA 事务的关键在于 TM 组件，其中的难点技术点如下：

****第二段提交时，当 RM1 commit 完成了，而 RM2 commit 还没有完成，这时 TM 需要进行协调，当 RM2 恢复以后，重新提交之前没有 Commit 的事务，或者自动回滚之前 Rollback 的事务。**

****因此 TM 需要记录 XA 事务的状态，以及在各个 RM 上的执行情况，这个日志文件需要存储在可靠的地方，用来进行 XA 事务异常之后的补救工作。**

在 The XA Specification 里的 2.3 小节：Transaction Completion and Recovery 明确提到 TM 是要记录日志的：

In Phase 2, the TM issues all RMs an actual request to commit or roll back the transaction branch, as the case may be. (Before issuing requests to commit, the TM stably records the fact that it decided to commit, as well as a list of all involved RMs.) All RMs commit or roll back changes to shared resources and then return status to the TM. The TM can then discard its knowledge of the global transaction.

TM 是一定要把事务的信息，比如 XID，哪个 RM 已经完成了等保存起来的。只有当全部的 RM 提交或者回滚完后，才能丢弃这些事务的信息。

于是我们明白 TM 是一个单点，要非常可靠才行。

以 Java 分布式事务的开源 TM 组件 atomikos 为例，它是通过在应用的目录下生成日志文件来保证，如果失败，在重启后可以通过日志来完成未完成的事务。

Mycat 未来计划以 Zookeeper 作为 XA 事务的日志存储手段，实现 TM 角色以支持 XA 事务。

3.3 XA 事务的问题和 MySQL 的局限

XA 事务的明显问题是 timeout 问题，比如当一个 RM 出问题了，那么整个事务只能处于等待状态。这样可以会连锁反应，导致整个系统都很慢，最终不可用，另外 2 阶段提交也大大增加了 XA 事务的时间，使得 XA 事务无法支持高并发请求。

避免使用 XA 事务的方法通常是最终一致性。

举个例子，比如一个业务逻辑中，最后一步是用户账号增加 300 元，为了减少 DB 的压力，先把这个放到消息队列里，然后后端再从消息队列里取出消息，更新 DB。那么如何保证，这条消息不会被重复消费？或者重复消费后，仍能保证结果是正确的？在消息里带上用户帐号在数据库里的版本，在更新时比较数据的版本，如果相同则加上 300；比如用户本来有 500 元，那么消息是更新用户的钱数为 800，而不是加上 300；

另外一个方式是，建一个消息是否被消费的表，记录消息 ID，在事务里，先判断消息是否已经消息过，如果没有，则更新数据库，加上 300,否则说明已经消费过了，丢弃。

前面两种方法都必须从流程上保证是单方向的。

其实严格意义上，用消息队列来实现最终一致性仍然有漏洞，因为消息队列跟当前操作的数据库是两个不同的资源，仍然存在消息队列失败导致这个账号增加 300 元的消息没有被存储起来（当然复杂的高级的消息队列产品可以避免这种现象，但仍然存在风险），而第二种方式则由于新的表跟之前的事务操作的表示在一个 Database 中，因此不存在上述的可能性。

MySQL 的 XA 事务，长期以来都存在一个缺陷：

MySQL 数据库的主备数据库的同步，通过 Binlog 的复制完成。而 Binlog 是 MySQL 数据库内部 XA 事务的协调者，并且 MySQL 数据库为 binlog 做了优化——binlog 不写 prepare 日志，只写 commit 日志。所有的参与节点 prepare 完成，在进行 xa commit 前 crash。crash recover 如果选择 commit 此事务。由于 binlog 在 prepare 阶段未写，因此主库中看来，此分布式事务最终提交了，但是此事务的操作并未写到 binlog 中，因此也就未能成功复制到备库，从而导致主备库数据不一致的情况出现。

Prior to MySQL 5.7.7, XA transactions were not compatible with replication. This was because an XA transaction that was in PREPARED state would be rolled back on clean server shutdown or client disconnect. Similarly, an XA transaction that was in PREPARED state would still exist in PREPARED state in case the server was shutdown abnormally and then started again, but the contents of the transaction

could not be written to the binary log. In both of these situations the XA transaction could not be replicated correctly.

In MySQL 5.7.7 and later, there is a change in behavior and an XA transaction is written to the binary log in two parts. When XA PREPARE is issued, the first part of the transaction up to XA PREPARE is written using an initial GTID. A `XA_prepare_log_event` is used to identify such transactions in the binary log. When XA COMMIT or XA ROLLBACK is issued, a second part of the transaction containing only the XA COMMIT or XA ROLLBACK statement is written using a second GTID. Note that the initial part of the transaction, identified by `XA_prepare_log_event`, is not necessarily followed by its XA COMMIT or XA ROLLBACK, which can cause interleaved binary logging of any two XA transactions. The two parts of the XA transaction can even appear in different binary log files. This means that an XA transaction in PREPARED state is now persistent until an explicit XA COMMIT or XA ROLLBACK statement is issued, ensuring that XA transactions are compatible with replication.

3.4 XA 事务使用指南

Mycat 从 1.6.5 版本开始支持标准 XA 分布式事务，考虑到 mysql 5.7 之前版本 xa 的 2 个 bug，所以推荐最佳搭配 XA 功能使用 mysql 5.7 版本。

Mycat 实现 XA 标准分布式事务，mycat 作为 xa 事务协调者角色，即使事务过程中 mycat 宕机挂掉，由于 mycat 会记录事务日志，所以 mycat 恢复后会进行事务的恢复善后处理工作。

考虑到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。

使用示例：

XA 操作说明

1. `set autocommit=0;`

XA 事务 需要设置手动提交

2. `set xa=on;`

使用该命令开启 XA 事务


```
3. insert into travelrecord(id,name)
values(1,'N'),(6000000,'A'),(321,'D'),(13400000,'C'),(59,'E');
```

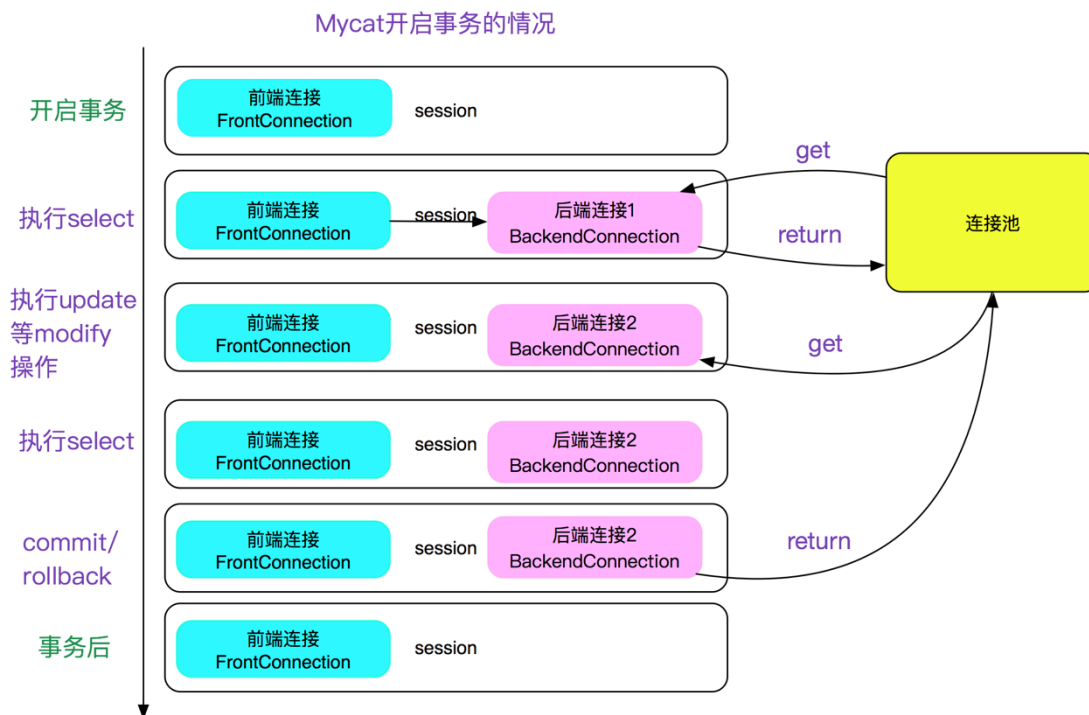
执行相应的 SQL 语句部分

```
4. commit;
```

对事务进行提交，事务结束

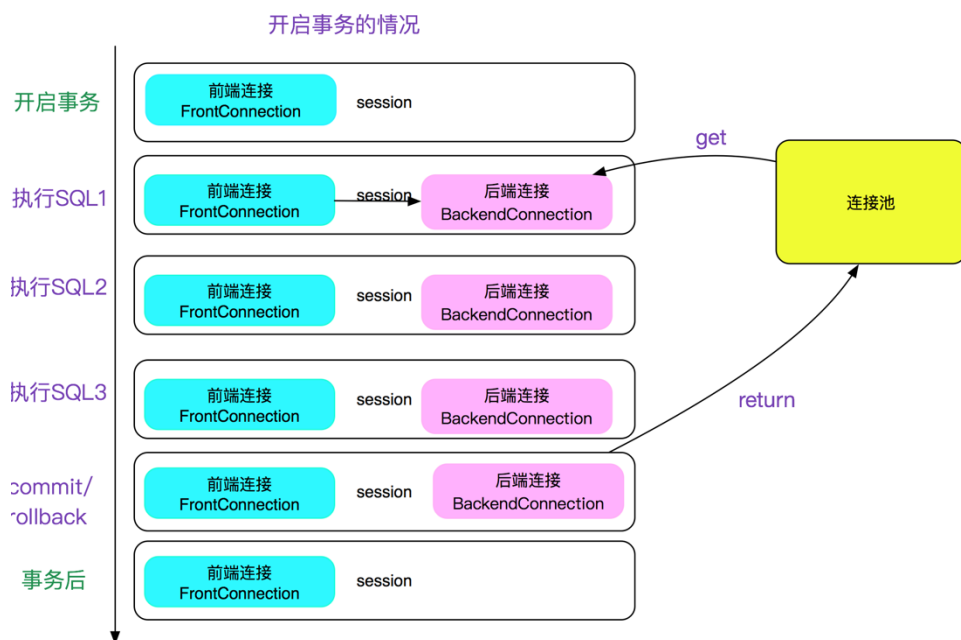
3.5 保证 repeatable read

mycat 有一个特性，就是开事务之后，如果不运行 update/delete/select for update 等更新类语句 SQL 的话，不会将当前连接与当前 session 绑定。如下图所示：



这样做的好处是可以保证连接可以最大限度的复用，提升性能。

但是,这就会导致两次 select 中如果有其它的在提交的话,会出现两次同样的 select 不一致的现象,即不能 repeatable read,这会让人直连 mysql 的人很困惑,可能会在依赖 repeatable read 的场景出现问题。所以做了一个开关,当 server.xml 的 system 配置了 strictTxIsolation=true 的时候(true),会关掉这个特性,以保证 repeatable read,加了开关后如下图所示:

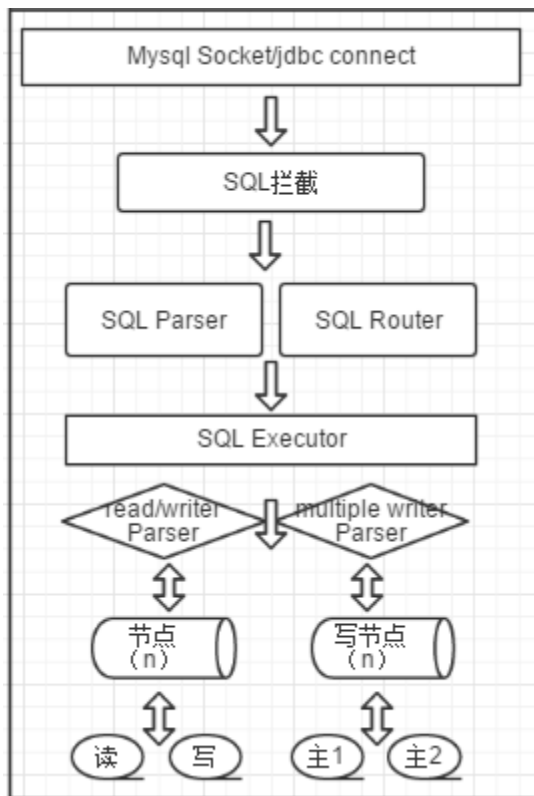


第 4 章 Mycat SQL 拦截机制

SQL 拦截是一个比较有用的高级技巧,用户可以写一个 java 类,将传入 MyCAT 的 SQL 进行改写然后交给 Mycat 去执行,此技巧可以完成如下一些特殊功能:

- 捕获和记录某些特殊的 SQL;
- 记录 sql 查找异常;
- 出于性能优化的考虑,改写 SQL,比如改变查询条件的顺序或增加分页限制;
- 将某些 Select SQL 强制设置为 Read 模式,走读写分离(很多事务框架很难剥离事务中的 Select SQL;
- 后期 Mycat 智能优化,拦截所有 sql 做智能分析,自动监控节点负载,自动优化路由,提供数据库优化建议。

SQL 拦截的原理是在路由之前拦截 SQL,然后做其他处理,完了之后再做路由,执行如下图所示:



默认的拦截器实现了 Mysql 转义字符的过滤转换，非默认拦截器只有一个拦截记录 sql 的拦截器。

a. 默认 SQL 拦截器：

配置：

```

<system>
<property name="sqlInterceptor">io.mycat.interceptor.impl.DefaultSqlInterceptor</property>
</system>

```

源码：

```

/**
 * escape mysql escape letter
 */
@Override
public String interceptSQL(String sql, int sqlType) {
    if (sqlType == ServerParse.UPDATE || sqlType == ServerParse.INSERT || sqlType ==
ServerParse.SELECT || sqlType == ServerParse.DELETE) {
        return sql.replace("\\'", "'");
    } else {

```

```
return sql;

}

}
```

b. 捕获记录 sql 拦截器配置：

```
<system>

<property name="sqlInterceptor">io.mycat.interceptor.impl.StatisticsSqlInterceptor</property>

<property name="sqlInterceptorType">select, update, insert, delete</property>

<property name="sqlInterceptorFile">E:/mycat/sql.txt</property>

</system>
```

sqlInterceptorType：拦截 sql 类型

sqlInterceptorFile：sql 保存文件路径

注意：捕获记录 sql 拦截器的配置只有 1.4 及其以后可用，1.3 无本拦截。

如果需要实现自己的 sql 拦截，只需要将配置类改为自己配置即可：

- 1.定义自定义类 implements SQLInterceptor，然后改写 sql 后返回。
- 2.将自己实现的类放入 catlet 目录，可以为 class 或 jar。
- 3.配置配置文件：

```
<system>

<property name="sqlInterceptor">io.mycat.interceptor.impl.自定义 class</property>

<!--其他配置-->

</system>
```

第 5 章 Mycat 注解

5.1 注解原理

概念：

MyCat 对自身不支持的 Sql 语句提供了一种解决方案——在要执行的 SQL 语句前添加额外的一段由注解 SQL 组织的代码，这样 Sql 就能正确执行，这段代码称之为“注解”。注解的使用相当于对 mycat 不支持的 sql 语句做了一层透明代理转发，直接交给目标的数据节点进行 sql 语句执行，其中注解 SQL 用于确定最终执行 SQL 的数据节点。注解的形式是：

```
/*!mycat: sql=注解 Sql 语句*/
```

注解的使用方式是：

```
/*!mycat: sql=注解 Sql 语句*/真正执行 Sql
```

使用时将=号后的“注解 Sql 语句”替换为需要的 Sql 语句即可，后面会提到具体的用法。

原理：

MyCat 执行 SQL 语句的流程是先进行 SQL 解析处理，解析出分片信息(路由信息)后，然后到该分片对应的物理库上去执行；若传入的 SQL 语句 MyCat 无法解析，则 MyCat 不会去执行；而注解则是告诉 MyCat 按照注解内的 SQL（称之为注解 SQL）去进行解析处理，解析出分片信息后，将注解后真正要执行的 SQL 语句（称之为原始 SQL）发送到该分片对应的物理库上去执行。

从上面的原理可以看到，注解只是告诉 MyCat 到何处去执行原始 SQL；因而使用注解前，要清楚的知道该原始 SQL 去哪个分片执行，然后在注解 SQL 中也指向该分片，这样才能使用！例子中的 sharding_id=10010 即是指明分片信息的。

需要说明的是，若注解 SQL 没有能明确到具体某个分片，譬如例子中的注解 SQL 没有添加 sharding_id=10010 这个条件，则 MyCat 会将原始 SQL 发送到 persons 表所在的所有分片上去执行去，这样造成的后果若是插入语句，则在多个分片上都存在重复记录，同样查询、更新、删除操作也会得到错误的结果！

• 解决问题：

1. MySql 不支持的语法结构，如 insert ...select...;
2. 同一个实例内的跨库关联查询，如用户库和平台库内的表关联；
3. 存储过程调用；
4. 表，存储过程创建。

• 注解规范

1. 注解 SQL 使用 select 语句，不允许使用 delete/update/insert 等语句；虽然 delete/update/insert 等语句也能用在注解中，但这些语句在 Sql 处理中有额外的逻辑判断，从性能考虑，请使用 select 语句
2. 注解 SQL 禁用表关联语句；
3. 注解 SQL 尽量用最简单的 SQL 语句，如 select id from tab_a where id=' 10000' ；
4. 无论是原始 SQL 还是注解 SQL，禁止 DDL 语句；
5. 能不用注解的尽量不用；

6. 详细要求见下表。

原始 Sql	注解 Sql	备注
Select	1. 选择能唯一确定分片的主表，如与用户表关联的时候可以选择用户表	
	2. 若是业务需要在主表所在的各个分片上都执行可以不加能确定分片的条件	
Insert	对于分片表	
	1. 使用 insert 的表做注解 SQL	
	2. 注解 SQL 必须能确认具体到某个分片	
	3. 原始 SQL 插入的字段必须包含分片字段	
	4. 原始 SQL 中包含的分片字段和注解 SQL 中的分片字段确定的分片务必要一致	
	5. 对于 insert ... select 这种语句，请务必确认插入的记录都在当前查找到的分片上	
	非分片表	
	1. 注解 SQL 必须能具体确认到某个分片	
Delete	2. 注解 SQL 包含的分片字段其分片上必须包含这个非分片表	
	1. 对于分片表使用要删除记录的表做注解 SQL	
Update	1. 对于分片表用所要更新的表做注解 SQL	
	1. 禁止更新分片表的分片列	
	3. 根据业务需要添加注解 Sql 的分片字段值	
Call	1. 若是要在所有的分片上都执行存储过程，则使用一个在所有分片上都包含的表，不添加任何分片条件 调用存储过程	
	2. 若是单个分片执行，使用能确认到这个分片的表以及分片条件	

补充说明:

使用注解并不额外增加 MyCat 的执行时间；从解析复杂度以及性能考虑，注解 SQL 应尽量简单。至于一个 SQL 使用注解和不使用注解的性能对比，不存在参考意义，因为前提是 MyCat 不支持的 SQL 才使用注解。

5.2 注解使用示例

注解支持的'!' 不被 mysql 单库兼容，

注解支持的'#' 不被 mybatis 兼容

新增加 mycat 字符前缀标志 Hintsql:"/** mycat: */"

从 1.6 开始支持三种注解方式：

```
/*#mycat:db_type=master*/ select * from travelrecord
```

```
/*!mycat:db_type=slave*/ select * from travelrecord
```

```
/**mycat:db_type=master*/ select * from travelrecord
```

1. Mycat 端执行存储创建表或存储过程为：

存储过程：

```
/*!mycat: sql=select 1 from test */ CREATE PROCEDURE `test_proc`() BEGIN END ;
```

表：

```
/*!mycat: sql=select 1 from test */create table test2(id int);
```

注意注解中语句是节点的表请替换成自己表如 select 1 from 表，注解内语句查出来的数据在哪个分片，数据在那个节点往哪个节点建。

2. 特殊语句自定义分片：

```
/*!mycat: sql=select 1 from test */insert into t_user(id,name) select id,name from t_user2;
```

3. 读写分离

配置了 Mycat 读写分离后，默认查询都会从读节点获取数据，但是有些场景需要获取实时数据，如果从读节点获取数据可能因延时而无法实现实时，Mycat 支持通过注解/*balance*/来强制从写节点查询数据：

a. 事务内的 SQL，默认走写节点，以注解/*balance*/开头，则会根据 schema.xml 的 dataHost 标签属性的

balance= "1" 或 "2" 去获取节点

b. 非事务内的 SQL，开启读写分离默认根据 balance= "1" 或 "2" 去获取，以注解/*balance*/开头则会走写节点解决部分已经开启读写分离，但是需要强一致性数据实时获取的场景走写节点

```
/*balance*/ select a.* from customer a where a.company_id=1;
```

4. 多表 ShareJoin

```
/*!mycat:catlet=demo.catlets.ShareJoin */ select a.*,b.id, b.name as tit from customer a,company b on a.company_id=b.id;
```

5.读写分离数据源选择

```
/*!mycat:db_type=master*/ select * from travelrecord
```

```
/*!mycat:db_type=slave*/ select * from travelrecord
```

```
/*mycat:db_type=master*/ select * from travelrecord
```

```
/*mycat:db_type=slave*/ select * from travelrecord
```

6. 多租户支持

通过注解方式在配置多个 schema 情况下, 指定走哪个配置的 schema。

- web 部分修改:
 - a. 在用户登录时, 在线程变量 (ThreadLocal) 中记录租户的 id
 - b. 修改 jdbc 的实现: 在提交 sql 时, 从 ThreadLocal 中获取租户 id, 添加 sql 注释, 把租户的 schema 放到注释中。例如: `/*!mycat : schema = test_01 */ sql ;`
- 在 db 前面建立 proxy 层, 代理所有 web 过来的数据库请求。proxy 层是用 mycat 实现的, web 提交的 sql 过来时在注释中指定 schema, proxy 层根据指定的 schema 转发 sql 请求。

```
/*!mycat : schema = test_01 */ sql ;
```


第 6 章 MyCAT 支持的 Catlet 实现

通过 catlet 支持跨分片复杂 SQL 实现以及存储过程支持等。使用方式主要通过 mycat 注释的方式来执行，如下：

1. 跨分片联合查询注解支持：

```
/*!mycat:catlet=demo.catlets.ShareJoin /select bu,sg.* from base_user bu,sam_glucose sg where  
bu.id_=sg.user_id;
```

注：sam_glucose 是跨分片表；

2. 存储过程注解支持：

```
/*!mycat: sql=select * from base_user where id_=1;*/ CALL proc_test();
```

注：目前执行存储过程通过 mycat 注解的方式执行，注意需要把存储过程中的 sql 写到注解中；

3. 批量插入与 ID 自增长结合的支持：

```
/*!mycat:catlet=demo.catlets.BatchInsertSequence */ insert into sam_test(name_)  
values( 't1' ),( 't2' );
```

注：此方式不需要在 sql 语句中显示的设置主键字段，程序在后台根据 primaryKey 配置的主键列，自动生成主键的 sequence 值并替换原 sql 中相关的列和值；

4. 获取批量 sequence 值的支持：

```
/*!mycat:catlet=demo.catlets.BatchGetSequence */SELECT mycat_get_seq( 'MYCAT_TEST' ,100);
```

注：此方法表示获取 MYCAT_TEST 表的 100 个 sequence 值，例如当前 MYCAT_TEST 表的最大 sequence 值为 5000，则通过此方式返回的是 5001，同时更新数据库中的 MYCAT_TEST 表的最大 sequence 值为 5100。

第 7 章 jdbc 多数据库支持

7.1 JDBC 概述

JDBC 是一套数据库访问协议，是由 Sun 定义一组接口，由数据库厂商来实现。是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问，它由一组用 Java 语言编写的类和接口组成。JDBC 为工具/数据库开发人员提供了一个标准的 API，据此可以构建更高级的工具和接口，使数据库开发人员能够用纯 Java API 编

写数据库应用程序。

会员逍遥子建议开设此课程，建议内容如下，大家看看如何，欢迎跟帖提出建议：

申请原因：MYCAT 背后有一支强大的技术团队，其参与者都是 5 年以上资深软件工程师、架构师、DBA 等，优秀的技术团队保证了 MYCAT 的产品质量。MYCAT 并不依托于任何一个商业公司，因此不像某些开源项目，将一些重要的特性封闭在其商业产品中，使得开源项目成了一个摆设。Mycat 技术越来越受到更多的关注和喜爱，期望全面学习 mycat 技术。同时也希望更多的介绍与数据库相关的技术。

课程大纲：Mycat 实战

1 Mycat 前世今生

Mycat 的历史、背后的团队、发展现状、RoadMap 等

2 Mycat 原理与入门

Mycat 的原理

主要功能

配置和使用入门

3 Mycat 故障排查指南

常见问题

日志排查

命令行工具指南

4 Mycat 分片规则详解

Mycat 几种分片规则的使用说明以及例子

5 Mycat 跨分片问题

Mycat 跨分片聚合处理

Mycat 跨分片 JOIN

Mycat 分布式事务

7.2 JDBC 体系结构

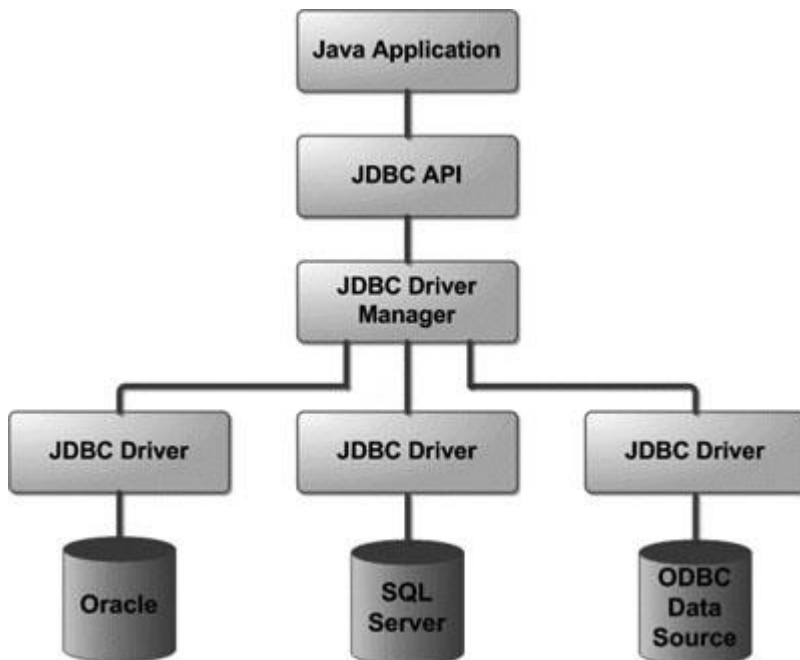
支持两层和三层的处理模式对数据库的访问，但一般 JDBC 体系结构由两层组成：

1: JDBC API:提供应用程序到 JDBC 管理器连接。

2: JDBC Driver API: 支持 JDBC 管理器-驱动器连接。

JDBC API 使用一个驱动程序管理器和数据库特定的驱动程序提供透明的异构数据库的连接，驱动程序管理器能够支持多个并发连接到多个异构数据库的驱动程序。

以下是架构图，它显示的 JDBC 驱动程序和 Java 应用程序与驱动程序管理器的位置：



7.3 JDBC API

DriverManager: 这个类管理数据库驱动程序的列表。从 Java 应用程序的连接请求匹配的合适的数据库驱动程序，使用通讯子协议。第一个 JDBC 驱动程序识别某个子协议将被用来建立一个数据库连接。

Driver: 此接口处理与数据库

服务器的通信。将直接与驱动程序对象很少。相反，您可以使用 DriverManager 隔离对象，这种类型的管理对象。它也抽象与驱动程序对象与工作相关的细节

Connection : 此接口与用于接触一个数据库的所有方法。连接对象通信的情况下，即，所有的通信是只通过与数据库连接对象。

Statement : 使用接口提交到数据库的 SQL 语句创建的对象。一些派生的接口接受，除了执行存储过程的参数。

ResultSet: 这些对象保存后，使用 Statement 对象执行 SQL 查询从数据库中检索数据。它作为一个迭代器，让您可以通过它的数据移动。

SQLException: 这个类处理的数据库应用程序中发生的任何错误。

7.4 JDBC 4.0

自从核心 Java 语言的第一个公开发行人版本起, JDBC 已经经历了十年的发展历程。它的当前版本 4.0 (Java 6.0 及之后的版本提供) 提供了一组更为丰富的 API, 主要目的在于改进软件开发的设计和性能。

新功能包括以下几个方面的变化:

- **数据库自动加载驱动程序:**

在此版 JDBC 中做到了, 您不必再显式地加载 `Class.forName` 了, 当您的程序首次试图连接数据库时, `DriverManager` 自动加载驱动到当前应用的 `CLASSPATH`。这是 JDBC 的一个比较大的改动。

- **异常处理的改进**

在 JDBC API 4.0 以前的版本中, 异常处理功能极其有限。对于所有类型的错误都会笼统地抛出一个 `SQLException` 异常-根本不存在异常的详细分类, 且没有相应的层次定义。所以这时, 你唯一能够得到一些有意义的信息的办法是检索和分析 `SQLState` 值。另一方面, `SQLState` 值及其相应的含义会因不同的数据源而有所改变; 因此, 要想追踪到问题的“根部”并且有效地处理异常是一件非常乏味的任务。

- **Connection 和 Statement 接口的增强功能**

有时数据库连接是不可用的, 尽管可能不必关闭这些连接并对之进行垃圾回收。处于这样的情况下, 数据库常常表现出速度缓慢且不具有响应性。此时, 在大多数情况下, 重新初始化该连接也许是解决这种问题的唯一方法。在 JDBC 4.0 以前版本时, 没有办法来区分一个旧连接和一个已经关闭的连接; 而新式 API 则在 `Connection` 接口中添加了一个 `isValid()` 方法用来查询是否连接仍然有效。

- **SQL2003 XML 数据类型的支持**

JDBC 4.0 把 `SQLXML` 定义为映射数据库 `SQLXML` 类型的 Java 数据类型。这种 API 支持把一个 XML 类型作为一个字符串或作为一个 `StAX` 流进行处理。`Streaming API for XML` (在 JSR 173 规范中确立) 基于 `Iterator` 模式, 它与基于 `Observer` 模式的 `Simple API for XML Processing (SAX)` 形成对照。

- **SQL ROWID 访问**

在许多数据库中, `RowId` 都被用作唯一标识一个表中行的方法。在查询条件中使用 `RowId` 往往是检索数据的最快方法, 特别是在 Oracle 和 DB2 数据库情况下。现在, 既然 `java.sql.RowId` 是一种内嵌的 Java 类型; 那么, 你就可以充分利用与其用法相关的性能优点。当表中存在重复的数据并且一些行数据相同时, `RowId` 是标识唯一

行的最有效的方法。然而，还要注意，RowId 在一个表中是唯一的，而对于整个数据库来说并非如此；它们可能发生变化并且不为所有数据库所支持。典型情况下，RowId 不是跨数据源可移植的；因此，当使用多种数据源时应该慎重。在数据源定义的生命周期内，只要一行未被删除，那么该行相应的 RowId 就一直保持有效。我们可以调用 DatabaseMetadata.getRowIdLifetime() 方法来决定 RowId 的生命周期。这个方法的返回类型是一个枚举类型。现在，把所有这些枚举类型总结到如下的表格中。

RowIdLifetime 枚举类型	定义
ROWID_UNSUPPORTED	数据源不支持 RowId 类型
ROWID_VALID_OTHER	实现依赖的生命周期
ROWID_VALID_TRANSACTION	生命周期至少包含事务
ROWID_VALID_SESSION	生命周期至少包含会话
ROWID_VALID_FOREVER	无限制生命周期

7.5 Mycat 对 JDBC 的支持

Mycat 在 1.3 版本开始正式实现对 JDBC 的支持，这一特性实现了对其它数据库的支持，如 Oracle、DB2、SQL Server，将其模拟为 MySQL Server 使用，也就是说 Mycat 从 mysql 的数据库中间件升级为数据库中间件，而且后端同时支持多数据库混合使用，成为一个数据平台。

Mycat 对 jdbc 的支持原理是通过将 Mycat 模拟为一个统一的 Mysql 数据库，应用以 jdbc 方式访问数据库时候，使用统一的 Mysql jdbc 方式连接，连接后各数据库使用不变。

例如：oracle 连接则是使用 mysql 驱动连接，然后 oracle 特有的分页 rownum 仍旧使用 oracle 语法，其他数据库类似。

Mycat 在 1.4 版本针对 JDBC 的执行引擎放入线程池中执行，据测试，比不用线程方式执行 SQL 语句效率高 20%-30%。

7.6 NoSQL 支持(MongoDB)

NoSQL=Not Only SQL,目前已经存在很多的 NoSQL 数据库，比如 MongoDB、Redis、Riak、HBase、Cassandra 等等。每一个都拥有以下几个特性中的一个：

不再使用 SQL 语言，比如 MongoDB、Cassandra 就有自己的查询语言

通常是开源项目

为集群运行而生

弱结构化——不会严格的限制数据结构类型

NoSQL 可以大体上分为 4 个种类：Key-value、Document-Oriented、Column-Family Databases 以及 Graph-Oriented Databases。

1. 键值 (Key-Value) 数据库

键值数据库就像在传统语言中使用的哈希表。你可以通过 key 来添加、查询或者删除数据，鉴于使用主键访问，所以会获得不错的性能及扩展性。

产品：Riak、Redis、Memcached、Amazon' s Dynamo

2. 面向文档 (Document-Oriented) 数据库

面向文档数据库会将数据以文档的形式储存。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名称与对应的值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用 XML、JSON 或者 JSONB 等多种形式存储。

产品：MongoDB、CouchDB、RavenDB

3. 列存储 (Wide Column Store/Column-Family) 数据库

列存储数据库将数据储存在列族 (column family) 中，一个列族存储经常被一起查询的相关数据。举个例子，如果我们有一个 Person 类，我们通常会一起查询他们的姓名和年龄而不是薪资。这种情况下，姓名和年龄就会被放入一个列族中，而薪资则在另一个列族中。

产品：Cassandra、HBase

8.图 (Graph-Oriented) 数据库

图数据库允许我们将数据以图的方式储存。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple 和 Next，则会有两个 “Founded by” 的边将 Apple 和 Next 连接到 Steve Jobs。

产品：Neo4J、Infinite Graph、OrientDB

7.7 MongoDB

Mycat 支持 JDBC 连接后端数据库，理论上支持任何数据库，如 ORACLE、DB2、SQL Server 等，是将其模拟为 MySQL，所以对其他数据库只支持标准的 SQL 语句，而对 NoSQL MongoDB 的支持，是封装 MongoDB API 基于 JDBC 的实现,目前 Mycat1.3 实现了对 mongodb 的支持。

7.7.1 配置支持 Mongodb

修改 conf 下的配置 schema.xml 文件中的以下内容：

配置 dataHost

在节点下新增一个 mongodb 的连接

```
<dataHost name="jdbchost" maxCon="1000" minCon="1" balance="0" writeType="0" dbType="mongodb"
dbDriver="jdbc">

  <heartbeat>select user()</heartbeat>

  <writeHost host="hostM" url="mongodb://192.168.0.99/" user="admin"
password="123456" ></writeHost>

</dataHost>
```

1. dbDriver 一定为 jdbc,dbType 代表数据库类型，可以为 mongodb,oracle,通过配置这个可以支持其他数据库
2. url 地址是 jdbc 连接的地址,和一般开发 java web 的 jdbc.url 地址一致
3. user,password 是用户名和密码,可以是任意值,目前不支持 mongodb 配置用户名和密码
4. 是心跳包的查询语句,可为空
5. 如果支持多个 mongodb 数据库，可以不用指定数据库名，在 dataNode 中指定

配置表：

```
<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">
```

之后加上表的配置：

```
<table name="people" primaryKey="_ID" dataNode="dn4" />
```

新增 dataNode 配置：

```
<dataNode name="dn4" dataHost="jdbchost" database="test1" />
```

```
<dataNode name="dn5" dataHost="jdbchost" database="test2" />
```

需要的 jar

mongo-java-driver-2.11.4.jar

这是 mongodb 官方提供的支持 java 的驱动包。

实现原理

通过实现标准的 JDBC 接口，调用 mongodb api 实现对 mongodb 的操作：

- (1) 解析 SQL 语句(druid sql parser 为 SQL 解析器)
- (2) 转化为 mongodb api (3) 发送到 mongodb 服务端实现

支持的 SQL 语法

Create table

create table people (name varchar(30),age int,sex int,diqu varchar(20),lev int);

```
mysql> create table people (name varchar(30),age int,sex int,diqu varchar(20),lev int);
Query OK, 1 row affected (0.00 sec)
OK!
mysql>
```

mongodb 中不用创建表，也可以使用。

Insert into 插入语句

insert into people (name,age,sex,diqu,lev) values('cs' ,22,1, 'sz' ,1);

```
mysql> insert into people (name,age,sex,diqu,lev) values('mongo',22,1,'sz',1);
Query OK, 1 row affected (0.02 sec)
OK!
```

注意在插入数据的时候，必须有字段名，否则会提示错误：

```
mysql> insert into people values('mongo',22,1,'sz',1);
ERROR 3009 (HY000): java.lang.RuntimeException: number of values and columns have to match
mysql>
```

查询下插入的数据：

```
mysql> select * from people where name='mongo';
+-----+-----+-----+-----+-----+-----+
| _id   | name  | age  | sex  | diqu  | lev  |
+-----+-----+-----+-----+-----+-----+
| 54a21dbd4001d690588ffe32 | mongo | 22   | 1    | sz    | 1    |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.10 sec)
```

Update table 更新语句

update people set age =23 where name= 'mongo' ;


```
mysql> update people set age =23 where name='mongo';
Query OK, 1 row affected (0.05 sec)
OK!

mysql> select * from people where name='mongo';
+-----+-----+-----+-----+-----+-----+
| _id          | name  | age  | sex  | diqu | lev  |
+-----+-----+-----+-----+-----+-----+
| 54a21dbd4001d690588ffe32 | mongo | 23   | 1    | sz   | 1    |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.06 sec)
```

Select 查询语句

支持*的查询

select * from people where name= 'mongo' ;

```
mysql> select * from people where name='mongo';
+-----+-----+-----+-----+-----+-----+
| _id          | name  | age  | sex  | diqu | lev  |
+-----+-----+-----+-----+-----+-----+
| 54a21dbd4001d690588ffe32 | mongo | 23   | 1    | sz   | 1    |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.06 sec)
```

支持指定字段名的查询

select name,age from people where name= 'mongo' ;

```
mysql> select name,age from people where name='mongo';
+-----+-----+-----+
| _id          | name  | age  |
+-----+-----+-----+
| 54a21dbd4001d690588ffe32 | mongo | 23   |
+-----+-----+-----+
1 row in set (0.00 sec)
```

where 条件

支持等于：

select name,age from people where name= 'mongo' ;

支持大于：

```
mysql> select name,age from people where age>23;
+-----+-----+-----+
| _id          | name  | age  |
+-----+-----+-----+
| 549eb4dc40018c3cf9748d65 | feng  | 30   |
| 549eb4f840018c3cf9748d66 | jifeng | 30   |
| 549eb53540018c3cf9748d67 | zhou  | 32   |
| 549eb54b40018c3cf9748d68 | sohudo | 32   |
+-----+-----+-----+
4 rows in set (0.01 sec)
```

支持小于：

```
mysql> select * from people where age<23;
```

_id	name	age	sex	diqu	lev
54a0c2374001a4714677799c	zz	19	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1

```
2 rows in set (0.00 sec)
```

支持小于等于:

```
mysql> select * from people where age<=23;
```

_id	name	age	sex	diqu	lev
54a0252740012420e1872a07	cai	23	2	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

```
4 rows in set (0.01 sec)
```

支持大于等于

```
mysql> select * from people where age>=23;
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

```
6 rows in set (0.01 sec)
```

支持不等于

```
mysql> select * from people where age<>23;
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1

```
6 rows in set (0.01 sec)
```

支持 AND

```
mysql> select * from people where age>=30 and lev=2;
```

_id	name	age	sex	diqu	lev
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb53540018c3cf9748d67	zhou	32	1	gz	2

2 rows in set (0.01 sec)

支持 and 表示范围

```
mysql> select * from people where age>18 and age<30;
```

_id	name	age	sex	diqu	lev
54a0252740012420e1872a07	cai	23	2	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

4 rows in set (0.00 sec)

支持多个 and

```
mysql> select * from people where age>18 and age<30 and sex=1;
```

_id	name	age	sex	diqu	lev
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

2 rows in set (0.00 sec)

支持 OR

```
mysql> select * from people where age>30 or diqu<>'gz';
```

_id	name	age	sex	diqu	lev
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

4 rows in set (0.01 sec)

```
mysql> select * from people where age>30 or diqu='gz';
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1

6 rows in set (0.01 sec)

支持多个 or

```
mysql> select * from people where age>30 or diqu='gz' or diqu='sz';
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

```
8 rows in set (0.01 sec)
```

支持 AND 和 OR 混合条件

```
mysql> select * from people where age>30 or (diqu='gz' and lev=1);
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1

```
5 rows in set (0.01 sec)
```

```
mysql> select * from people where (age>30 or diqu='gz') and lev=1;
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a0c2374001a4714677799c	zz	19	2	gz	1

```
4 rows in set (0.01 sec)
```

排序

支持升降序

```
mysql> select * from people order by age;
```

_id	name	age	sex	diqu	lev
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1
54a0252740012420e1872a07	cai	23	2	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
549eb53540018c3cf9748d67	zhou	32	1	gz	2

```
7 rows in set (0.01 sec)
```

```
mysql> select * from people order by age desc;
```

_id	name	age	sex	diqu	lev
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1

```
7 rows in set (0.01 sec)
```

多字段排序

```
mysql> select * from people order by age desc,lev desc;
```

_id	name	age	sex	diqu	lev
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1
54a0252740012420e1872a07	cai	23	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1

```
7 rows in set (0.01 sec)
```

支持 Limit

```
mysql> select * from people limit 5;
```

_id	name	age	sex	diqu	lev
549eb4dc40018c3cf9748d65	feng	30	1	gz	1
549eb4f840018c3cf9748d66	jifeng	30	1	gz	2
549eb53540018c3cf9748d67	zhou	32	1	gz	2
549eb54b40018c3cf9748d68	sohudo	32	1	gz	1
54a0252740012420e1872a07	cai	23	2	gz	1

```
5 rows in set (0.00 sec)
```

```
mysql> select * from people limit 5, 5;
```

_id	name	age	sex	diqu	lev
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

```
2 rows in set (0.00 sec)
```

```
mysql> select * from people limit 4, 5;
```

_id	name	age	sex	diqu	lev
54a0252740012420e1872a07	cai	23	2	gz	1
54a15c5d40017abf800821bb	cs	22	1	sz	1
54a21dbd4001d690588ffe32	mongo	23	1	sz	1

```
3 rows in set (0.00 sec)
```

Delete 删除语句

delete from people where name= 'zz' ;

```
mysql> delete from people where name='zz';
Query OK, 1 row affected (0.00 sec)
OK!

mysql> select * from people where name='zz';
Empty set (0.00 sec)
```

Drop 语句

drop table people;

删除表

7.8 Oracle

7.8.1 配置支持 Oracle

修改 conf 下的配置 schema.xml 文件中的以下内容：

配置 dataHost

在节点下在新增一个 oracle 的连接

```
<dataHost name="oracle1" maxCon="1000" minCon="1" balance="0" writeType="0" dbType="oracle"
dbDriver="jdbc">

<heartbeat>select 1 from dual</heartbeat>

<connectionInitSql>alter session set nls_date_format='yyyy-mm-dd hh24:mi:ss'</connectionInitSql>

<writeHost host="hostM1" url="jdbc:oracle:thin:@192.168.0.95:1521:orcl" user="test" password="test" >
```

```
</writeHost>
```

```
</dataHost>
```

- 1.dbDriver 一定为 jdbc,dbType 代表数据库类型, 可以为 mysql,oracle,mongodb
- 2.url 地址是 jdbc 连接的地址,和一般开发 java web 的 jdbc.url 地址一致, user,password 是用户名和密码
- 3.是心跳包的查询语句
- 4.是连接 oracle 的初始化语句,初始化本次会话的日期显示格式
- 5.需要 ojdbc14-x.jar 包(其它版本也支持)

配置表:

```
<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">
```

之后加上表的配置:

```
<table name="people" primaryKey="_ID" dataNode="dn4" needAddLimit="false"/>
```

needAddLimit 不自动在 sql 语句中使用 limit

新增 dataNode 配置:

```
dataNode name="dn4" dataHost="oracle1" database="test" />
```

7.8.2 三层嵌套分页

支持 oracle 的三层嵌套和 row_number2 种分页语法以及 rownum 控制最大条数的语法。

支持 limit 语法自动翻译原生分页, 详见 5.9 limit 分页自动转换。

以下分页等价 limit 5,10

```
select * from ( select row_.*, rownum rownum_ from ( select sid
from test where sts <> 'N' order by sid desc ) row_ where rownum
<= 15) where rownum_ > 5;
```

row_number 分页

以下分页等价 limit 5,10

```
SELECT *
FROM (SELECT sid, ROW_NUMBER() OVER (ORDER BY sid ) AS ROWNUM1
FROM test t
WHERE sts <> 'N'
) XX
```

```
WHERE ROWNUM1 > 5  
  
AND ROWNUM1 <= 15;
```

7.8.3 rownum 控制最大条数

以下语法控制查询结果最多 5 条

```
ELECT * FROM (SELECT * FROM test t) XX WHERE ROWNUM <= 5;
```

7.9 SQL Server

7.9.1 配置支持 SQL Server

修改 conf 下的配置 schema.xml 文件中的以下内容:

配置 dataHost

在节点下在新增一个 sqlserver 的连接

```
<dataHost name="sqlserver1" maxCon="1000" minCon="1" balance="0" writeType="0" dbType="sqlserver"  
dbDriver="jdbc">  
  
<heartbeat></heartbeat>  
  
<connectionInitSql></connectionInitSql>  
  
<writeHost host="hostM1" url="jdbc:sqlserver://localhost:1433" user="sa" password="sa" >  
  
</writeHost>  
  
</dataHost>
```

- 1.dbDriver 一定为 jdbc,dbType 代表数据库类型,可以为 sqlserver,oracle,mongodb
- 2.url 地址是 jdbc 连接的地址,和一般开发 java web 的 jdbc.url 地址一致, user,password 是用户名和密码
- 3.是心跳包的查询语句,可以为空
- 4.是连接 sqlserver 的初始化语句
- 5.需要 mssqljdbc*.jar 包(其它版本也支持)
- 6.如果支持多个数据库,可以不用指定数据库名,在 dataNode 中指定**

配置表:

```
<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">
```

之后加上表的配置:


```
<table name="people" primaryKey="_ID" dataNode="dn4" needAddLimit="false"/>
```

needAddLimit 不自动在 sql 语句中使用 limit

新增 dataNode 配置:

```
<dataNode name="dn4" dataHost="sqlserver1" database="test1" />
```

```
<dataNode name="dn5" dataHost="sqlserver1" database="test2" />
```

7.9.2 row_number 分页

支持 row_number 和 row_number 与 top 结合 2 种分页, 另外支持 top 限制最大条数。

支持 limit 语法自动翻译原生分页, 详见 5.9 limit 分页自动转换。

以下分页等价 limit 5,10

```
SELECT *  
FROM (SELECT sid, ROW_NUMBER() OVER (ORDER BY sid DESC) AS ROWNUM  
FROM test  
WHERE sts <> 'N'  
 ) XX  
WHERE ROWNUM > 5  
AND ROWNUM <= 15
```

7.9.3 row_number 与 top 结合分页

以下分页等价 limit 5,10

```
select * from ( select row_number()over(order by tempColumn)tempRowNumber,* from ( select top 15  
tempColumn=0, sid from test where sts<>'N' order by sid )t )tt where tempRowNumber>5;
```

7.9.4 top 限制最大条数

以下语法控制查询结果最多 5 条

```
select top 5 * from test where sts<>'N' order by sid
```

7.10 DB2

支持 row_number 分页和 fetch first rows only 语法

支持 limit 语法自动翻译原生分页, 详见 5.9 limit 分页自动转换。

7.10.1 row_number 分页

以下分页等价 limit 5,10

```
SELECT *  
FROM (SELECT sid, ROW_NUMBER() OVER (ORDER BY sid DESC) AS ROWNUM  
FROM test  
WHERE sts <> 'N'  
 ) XX  
WHERE ROWNUM > 5  
AND ROWNUM <= 15
```

7.10.2 fetch first rows only 控制最大条数

以下语法控制查询结果最多 5 条

```
SELECT sid  
FROM test  
ORDER BY sid desc  
FETCH FIRST 5 ROWS ONLY;
```

7.11 Spark SQL/Hive

Mycat 对 Spark SQL/Hive 的支持是通过 JDBC 来完成的，使用 Hive 官方提供的 jdbc 包，必须开启 hiveserver2 的服务和 Hive 安装模式为远程模式（元数据放置在远程的 Mysql 数据库）。

7.11.1 配置 Mycat

修改 conf 下的配置 schema.xml 文件中的以下内容：

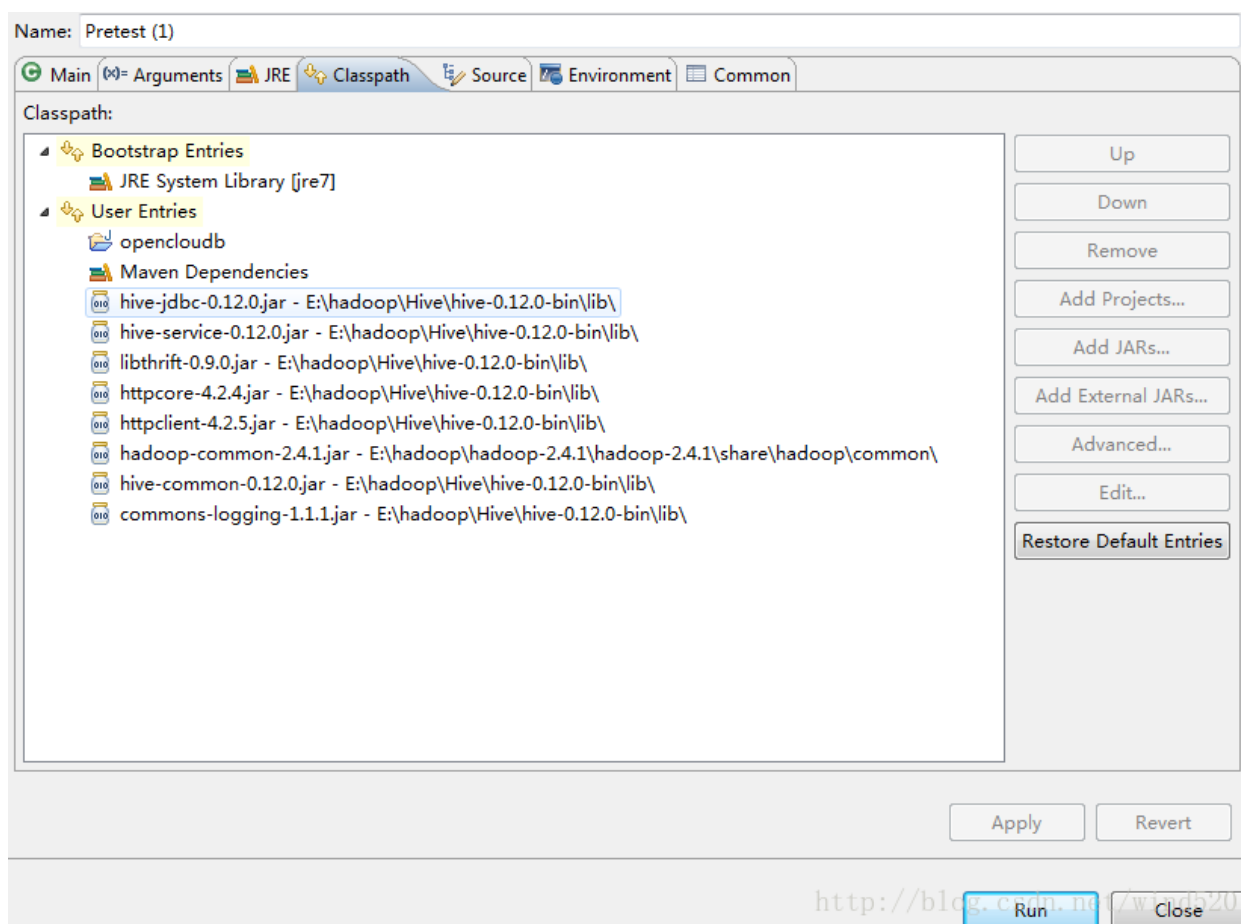
配置 dataHost

在节点下新增一个 spark 的连接

```
<dataHost name="sparksql" maxCon="1000" minCon="1" balance="0" dbType="spark" dbDriver="jdbc">  
<heartbeat></heartbeat>
```

```
<writeHost host="hostM1" url="jdbc:hive2://feng02:10000" user="jifeng" password="jifeng"></writeHost>
</dataHost>
```

- 1.dbDriver 一定为 jdbc,dbType 代表数据库类型, 可以为 spark,mysql,oracle,mongodb。
- 2.url 地址是 jdbc 连接的地址,和一般开发 java web 的 jdbc.url 地址一致, user,password 是用户名和密码。
- 3.是心跳包的查询语句,可以为空。
- 4.Spark SQL/Hive 和都是需要相同的 jar 包。



7.11.2 配置 Hive 安装模式

修改\$HIVE_HOME/conf/hive-site.xml

```
<property>
<name>javax.jdo.option.ConnectionURL</name>
<value>jdbc:mysql://jifengsql:3306/hive?createDatabaseIfNotExist=true</value>
<description>JDBC connect string for a JDBC metastore</description>
</property>

<property>
```

```

<name>javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
<description>Driver class name for a JDBC metastore</description>
</property>

<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>root</value>
<description>username to use against metastore database</description>
</property>

<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>jifeng</value>
<description>password to use against metastore database</description>
</property>

```

- javax.jdo.option.ConnectionURL mysql 数据库的 url 地址
- javax.jdo.option.ConnectionDriverName mysql jdbc 驱动
- javax.jdo.option.ConnectionUserName mysql 用户名
- javax.jdo.option.ConnectionPassword mysql 用户密码

启动 hiveserver2

- 命令行模式:

```
hive --service hiveserver2 --hiveconf hive.server2.thrift.port=10000
```

- 服务模式:

```
hiveserver2 start
```

```
[jifeng@feng01 conf]$ hive --service hiveserver2 --hiveconf hive.server2.thrift.port=10000
```

```
Starting HiveServer2
```

15/03/05 16:59:33 WARN conf.HiveConf: DEPRECATED: hive.metastore.ds.retry.* no longer has any effect.

Use hive.hmsHandler.retry.* instead

7.11.3 配置 Spark SQL

1. 需要先把 hive-site.xml 负责到 spark 的 conf 目录下

2. Running the Thrift JDBC/ODBC server

```
./sbin/start-thriftserver.sh --hiveconf hive.server2.thrift.port=10000 --hiveconf
hive.server2.thrift.bind.host=feng02 --master spark://feng02:7077 --driver-class-path
/home/jifeng/hadoop/spark-1.2.0-bin-2.4.1/lib/mysql-connector-java-5.1.32-bin.jar --executor-memory 1g
```

3. 端口:10000 服务器:feng02

spark master:spark://feng02:7077 driver-class-path:mysql 驱动包(hive 配置的)

7.12 PostgreSQL

支持 limit offset 分页语法以及 limit 控制最大条数的语法。

持 limit 语法自动翻译原生分页，详见 limit 分页自动转换。

```
select sid from test order by sid desc limit 10 offset 5;
```

等价于 mysql 的

```
select sid from test order by sid desc limit 5,10;
```

其实 mysql 也兼容 limit offset 写法

7.12.1 limit 分页自动转换

支持通过将标准的 limit 分页语法自动翻译转换为各数据库的原生分页，目前支持 limit 自动转换的数据库包括 oracle、sqlserver、db2、postgresql。

支持标准 limit 语法同时跨不同的数据库类型的分片。

例如表 test 的 dataNode 节点配置 oracle、sqlserver 等多个数据库类型的数据 node。

执行 limit 标准分页会针对每个数据库类型自动翻译分页语法，最后合并分页结果返回。

如果想查看自动翻译之后的原生分页语句，可以通过 explain 命令查看。

```
MySQL [base]> explain select * from test1 limit 5,10;
+-----+-----+
| DATA_NODE | SQL |
+-----+-----+
| dn6        | SELECT *
FROM <SELECT XX.*, ROWNUM AS RN
      FROM <SELECT *
            FROM test1
            > XX
      WHERE ROWNUM <= 15
            > XXX
WHERE RN > 5 |
+-----+-----+
1 row in set <0.00 sec>
```

第 8 章 管理命令与监控

MyCAT 自身有类似其他数据库的管理监控方式，可以通过 Mysql 命令行，登录管理端口（9066）执行相应的 SQL 进行管理，也可以通过 jdbc 的方式进行远程连接管理，本小节主要讲解命令行的管理操作。

登录：目前 mycat 有两个端口，8066 数据端口，9066 管理端口，命令行的登陆是通过 9066 管理端口来操作，登录方式类似于 mysql 的服务端登陆。

```
mysql -h127.0.0.1 -utest -ptest -P9066 [-dmycat]
```

-h 后面是主机，即当前 mycat 按照的主机地址，本地可用 127.0.0.1 远程需要远程 ip

-u Mycat server.xml 中配置的逻辑库用户

-p Mycat server.xml 中配置的逻辑库密码

-P 后面是端口 默认 9066，注意 P 是大写

-d Mycat server.xml 中配置的逻辑库

数据端口与管理端口的配置端口修改：

数据端口默认 8066，管理端口默认 9066，如果需要修改需要配置 serve.xml

```
<system>
<property name="serverPort">8067</property>
<property name="managerPort">9066</property>
</system>
```

命令总览：

通过 show @@help; 可以查看所有的命令, 如下:

```
mysql> show @@help;
```

+-----+-----+		
STATEMENT	DESCRIPTION	
+-----+-----+		
clear @@slow where datanode = ?	Clear slow sql by datanode	
clear @@slow where schema = ?	Clear slow sql by schema	
kill @@connection id1,id2,...	Kill the specified connections	
offline	Change MyCat status to OFF	
online	Change MyCat status to ON	
reload @@config	Reload all config from file	
reload @@route	Reload route config from file	
reload @@user	Reload user config from file	
rollback @@config	Rollback all config from memory	
rollback @@route	Rollback route config from memory	
rollback @@user	Rollback user config from memory	
show @@backend	Report backend connection status	
show @@cache	Report system cache usage	
show @@command	Report commands status	
show @@connection	Report connection status	
show @@connection.sql	Report connection sql	
show @@database	Report databases	
show @@datanode	Report dataNodes	
show @@datanode where schema = ?	Report dataNodes	
show @@datasource	Report dataSources	
show @@datasource where dataNode = ?	Report dataSources	
show @@heartbeat	Report heartbeat status	
show @@parser	Report parser status	
show @@processor	Report processor status	

show @@router	Report router status	
show @@server	Report server status	
show @@session	Report front session details	
show @@slow where datanode = ?	Report datanode slow sql	
show @@slow where schema = ?	Report schema slow sql	
show @@sql where id = ?	Report specify SQL	
show @@sql.detail where id = ?	Report execute detail status	
show @@sql.execute	Report execute status	
show @@sql.slow	Report slow SQL	
show @@threadpool	Report threadPool status	
show @@time.current	Report current timestamp	
show @@time.startup	Report startup timestamp	
show @@version	Report Mycat Server version	
stop @@heartbeat name:time	Pause dataNode heartbeat	
switch @@datasource name:index	Switch dataSource	
+-----+		

39 rows in set (0.00 sec)

reload @@config

在 MyCAT 的命令行监控窗口运行：

reload @@config;

该命令用于更新配置文件，例如更新 schema.xml 文件后在命令行窗口输入该命令，可不用重启即进行配置文件更新。运行结果参考如下：

```
mysql> reload @@config;
Query OK, 1 row affected (0.29 sec)
Reload config success
```

对应的 reload 配置有：

reload @@config	Reload all config from file
reload @@route	Reload route config from file （未实现）
reload @@user	Reload user config from file （未实现）

rollback @@config	Rollback all config from memory
rollback @@route	Rollback route config from memory （未实现）
rollback @@user	Rollback user config from memory （未实现）

show @@database

在 MyCAT 的命令行监控窗口运行：

show @@database;

该命令用于显示 MyCAT 的数据库的列表，对应 schema.xml 配置文件的 schema 子节点，参考运行结果如下：

```
mysql> show @@database;
+-----+
| DATABASE |
+-----+
| mycat   |
+-----+
1 row in set (0.00 sec)
```

show @@datanode

在 MyCAT 的命令行监控窗口运行：

show @@datanode;

该命令用于显示 MyCAT 的数据节点的列表，对应 schema.xml 配置文件的数据Node 节点，参考运行结果如下：

```
mysql> show @@datanode;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| NAME | DATHOST | INDEX | TYPE | ACTIVE | IDLE | SIZE | EXECUTE | TOTAL_TIME | MAX_TIME | MAX_SQL |
RECOVERY_TIME |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| blog | blog/blog | 0 | mysql | 0 | 13 | 100 | 329521 | 0 | 0 | 0 | -1 |
```

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set (0.00 sec)

```

其中，“NAME”表示 dataNode 的名称；“dataHost”表示对应 dataHost 属性的值，即数据主机；“ACTIVE”表示活跃连接数；“IDLE”表示闲置连接数；“SIZE”对应总连接数量。

运行如下命令，可查找对应的 schema 下面的 dataNode 列表：

show @@datanode where schema = ?

```

mysql> show @@datanode where schema = mycat;

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| NAME | DATHOST | INDEX | TYPE | ACTIVE | IDLE | SIZE | EXECUTE | TOTAL_TIME | MAX_TIME | MAX_SQL |
RECOVERY_TIME |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
| blog | blog/blog | 0 | mysql | 0 | 13 | 100 | 329541 | 0 | 0 | 0 | -1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set (0.00 sec)

```

show @@heartbeat

该命令用于报告心跳状态

RS_CODE 状态: OK_STATUS = 1;正常状态

ERROR_STATUS = -1; 连接出错

TIMEOUT_STATUS = -2;连接超时

INIT_STATUS = 0; 初始化状态

若节点故障，会连续默认 5 个周期检测，心跳连续失败，就会变成-1，节点故障确认，然后可能发生切换

参考运行结果如下所示：

```

mysql> show @@heartbeat;

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

```

-----+-----+
| NAME | TYPE | HOST | PORT | RS_CODE | RETRY | STATUS | TIMEOUT | EXECUTE_TIME |
LAST_ACTIVE_TIME | STOP |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+
| master | mysql | 121.40.121.133 | 3306 | 1 | 0 | idle | 30000 | 8334,7833,5722 | 2015-04-08 21:34:33 |
false |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+
1 row in set (0.00 sec)

```

show @@version

该命令用于获取 MyCAT 的版本，参考运行结果如下所示：

```

mysql> show @@version ;
+-----+
| VERSION |
+-----+
| 5.5.8-mycat-1.3 |
+-----+
1 row in set (0.00 sec)

```

show @@connection

该命令用于获取 Mycat 的前端连接状态，即应用与 mycat 的连接

kill @@connection id,id,id

用于杀掉连接。

参考运行结果如下所示：

```

mysql> show @@connection;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+
| PROCESSOR | ID | HOST | PORT | LOCAL_PORT | SCHEMA | CHARSET | NET_IN | NET_OUT |

```

ALIVE_TIME(S) RECV_BUFFER SEND_QUEUE txlevel autocommit											
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+											
-----+-----+-----+-----+											
Processor0	7	101.44.170.64	8066	13694	mycat	utf8	233	968	105	4096	
0	3	true									
Processor0	2	127.0.0.1	9066	34774	NULL	utf8	2014	33646	720	4096	
0	NULL	NULL									
Processor0	1	127.0.0.1	8066	44751	mycat	utf8	2502	85432	727	4096	
0	3	true									
Processor0	4	101.44.170.64	8066	13626	mycat	utf8	1244	3462	209	4096	
0	3	true									
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+											
-----+-----+-----+-----+											

4 rows in set (0.00 sec)

mysql> kill @@connection 7;

Query OK, 1 row affected (0.01 sec)

```
mysql> show @@connection;
```

PROCESSOR	ID	HOST	PORT	LOCAL_PORT	SCHEMA	CHARSET	NET_IN	NET_OUT	ALIVE_TIME(S)	RECV_BUFFER	SEND_QUEUE	txlevel	autocommit
Processor0	2	127.0.0.1	9066	34774	NULL	utf8	2060	34456	774	4096	0	NULL	NULL
Processor0	1	127.0.0.1	8066	44751	mycat	utf8	2502	85432	781	4096	0	3	true

```
| Processor0 | 4 | 101.44.170.64 | 8066 | 13626 | mycat | utf8 | 1259 | 3495 | 263 | 4096 |
0 | 3 | true |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

show @@backend

查看后端连接状态。

```
mysql> show @@backend;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+-----+-----+-----+-----+
| processor | id | mysqlId | host | port | l_port | net_in | net_out | life | closed | borrowed |
SEND_QUEUE | schema | txlevel | autocommit |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+-----+-----+-----+-----+
| Processor0 | 12 | 4768 | 121.40.121.133 | 3306 | 37141 | 236533254 | 2816448 | 1049325 | false | false |
0 | blog | 3 | true |
| Processor0 | 6 | 4632 | 121.40.121.133 | 3306 | 59890 | 299391847 | 3605804 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 13 | 4769 | 121.40.121.133 | 3306 | 37142 | 237221376 | 2850994 | 1049325 | false | false |
0 | blog | 3 | true |
| Processor0 | 5 | 4633 | 121.40.121.133 | 3306 | 59891 | 301727002 | 3551038 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 7 | 4628 | 121.40.121.133 | 3306 | 59886 | 300878413 | 3553483 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 8 | 4634 | 121.40.121.133 | 3306 | 59892 | 302614943 | 3647689 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 2 | 4630 | 121.40.121.133 | 3306 | 59888 | 308539162 | 3564896 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 9 | 4636 | 121.40.121.133 | 3306 | 59894 | 304212739 | 3686683 | 1296826 | false | false |
```

```

0 | blog | 3 | true |
| Processor0 | 10 | 4637 | 121.40.121.133 | 3306 | 59895 | 300780896 | 3573212 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 1 | 4631 | 121.40.121.133 | 3306 | 59889 | 301653846 | 3708506 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 14 | 4770 | 121.40.121.133 | 3306 | 37143 | 235054876 | 2784392 | 1049325 | false | false |
0 | blog | 3 | true |
| Processor0 | 3 | 4635 | 121.40.121.133 | 3306 | 59893 | 305185063 | 3618816 | 1296826 | false | false |
0 | blog | 3 | true |
| Processor0 | 11 | 0 | 121.40.121.133 | 3306 | 59896 | 7261962 | 1685851 | 1296825 | false | false |
0 | NULL | NULL | NULL |
| Processor0 | 4 | 4629 | 121.40.121.133 | 3306 | 59887 | 296327067 | 3631921 | 1296826 | false | false |
0 | blog | 3 | true |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+-----+-----+-----+-----+
14 rows in set (0.00 sec)

```

show @@cache;

查看 mycat 缓存。

SQLRouteCache: sql 路由缓存。

TableID2DataNodeCache : 缓存表主键与分片对应关系。

ER_SQL2PARENTID : 缓存 ER 分片中子表与父表关系。

```

mysql> show @@cache;
+-----+-----+-----+-----+-----+-----+-----+-----+
| CACHE | MAX | CUR | ACCESS | HIT | PUT | LAST_ACCESS | LAST_PUT |
+-----+-----+-----+-----+-----+-----+-----+
| SQLRouteCache | 10000 | 0 | 298175 | 0 | 0 | 1428815230596 | 0 |
| TableID2DataNodeCache.TESTDB_ORDERS | 50000 | 0 | 0 | 0 | 0 | 0 | 0 |
| ER_SQL2PARENTID | 1000 | 0 | 0 | 0 | 0 | 0 | 0 |

```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
```

3 rows in set (0.00 sec)

```
show @@datasource;
```

查看数据源状态，如果配置了主从，或者多主可以切换。

```
switch @@datasource name:index
```

切换数据源，name：schema 中配置的 dataHost 中 name。

index：schema 中配置的 dataHost 的 writeHost index 位标，即按照配置顺序从上到下的一次顺序，从 0 开始。

切换数据源时，会将原数据源所有的连接池中连接关闭，并且从新数据源创建新连接，此时 mycat 服务不可用。

dnindex.properties 文件在记录了当前的活跃 writer。

```
<dataHost name="blog" maxCon="100" minCon="10" balance="0"
```

```
writeType="0" dbType="mysql" dbDriver="native">
```

```
<heartbeat>select 1</heartbeat>
```

```
<writeHost host="master" url="127.0.0.1:3306" user="root" password="root"></writeHost>
```

```
<writeHost host="master2" url="127.0.0.1:3306" user="root1" password="root"></writeHost>
```

```
</dataHost>
```

```
mysql> show @@datasource;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
| DATANODE | NAME | TYPE | HOST | PORT | W/R | ACTIVE | IDLE | SIZE | EXECUTE |
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
| blog | master | mysql | 121.40.121.133 | 3306 | W | 0 | 10 | 100 | 16 |
```

```
| blog | master2 | mysql | 127.0.0.1 | 3306 | W | 0 | 0 | 100 | 0 |
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
```

2 rows in set (0.00 sec)

```
mysql> switch @@datasource blog:1;
```

Query OK, 1 row affected (1 min 0.05 sec)

04-12 15:21:06.617 INFO [\$_NIOREACTOR-2-RW] (PhysicalDBPool.java:296) -init result :finished 8
success 8 target count:10

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
MySQLConnection [id=38, lastTime=1428823206590, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=7085, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
MySQLConnection [id=39, lastTime=1428823206590, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=7084, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
MySQLConnection [id=41, lastTime=1428823206590, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=7087, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
MySQLConnection [id=42, lastTime=1428823206590, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=7090, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
MySQLConnection [id=43, lastTime=1428823206590, schema=mycat_node1, old shema=mycat_node1,
borrowed=true, fromSlaveDB=false, threadId=7088, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
MySQLConnection [id=45, lastTime=1428823206610, schema=mycat_node1, old shema=mycat_node1,

borrowed=true, fromSlaveDB=false, threadId=7091, charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]

04-12 15:21:06.617 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
 MySQLConnection [id=46, lastTime=1428823206610, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false, threadId=7092, charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]

04-12 15:21:06.618 DEBUG [\$_NIOREACTOR-2-RW] (PhysicalDatasource.java:386) -release channel
 MySQLConnection [id=47, lastTime=1428823206610, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false, threadId=7093, charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]

04-12 15:21:06.618 INFO [\$_NIOREACTOR-2-RW] (PhysicalDBPool.java:238) -jdbchost index:0 init success

04-12 15:21:06.618 INFO [\$_NIOREACTOR-2-RW] (MycatServer.java:366) -save DataHost index jdbchost cur index 0

04-12 15:21:06.620 INFO [\$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
 connection,reason:switch datasource ,MySQLConnection [id=34, lastTime=1428823025923, schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7068, charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]

04-12 15:21:06.620 INFO [\$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
 connection,reason:switch datasource ,MySQLConnection [id=26, lastTime=1428823025902, schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7061, charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]

04-12 15:21:06.620 INFO [\$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
 connection,reason:switch datasource ,MySQLConnection [id=30, lastTime=1428823025902, schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7063,

```

charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133,
port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-12 15:21:06.621 INFO [$ _NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
connection,reason:switch datasource ,MySQLConnection [id=31, lastTime=1428823025902,
schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7066,
charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133,
port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-12 15:21:06.621 INFO [$ _NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
connection,reason:switch datasource ,MySQLConnection [id=27, lastTime=1428823025923,
schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7064,
charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133,
port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-12 15:21:06.621 INFO [$ _NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
connection,reason:switch datasource ,MySQLConnection [id=33, lastTime=1428823025923,
schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7069,
charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133,
port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-12 15:21:06.622 INFO [$ _NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
connection,reason:switch datasource ,MySQLConnection [id=25, lastTime=1428823025902,
schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7060,
charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133,
port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-12 15:21:06.622 INFO [$ _NIOREACTOR-2-RW] (AbstractConnection.java:398) -close
connection,reason:switch datasource ,MySQLConnection [id=29, lastTime=1428823025902,
schema=mycat_node1, old shema=mycat_node1, borrowed=false, fromSlaveDB=false, threadId=7062,
charset=utf8, txIsolation=0, autocommit=true, attachment=null, respHandler=null, host=121.40.121.133,
port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-12 15:21:06.622 WARN [$ _NIOREACTOR-2-RW] (PhysicalDBPool.java:202) -[Host=jdbchost,result=[1-
>0],reason=MANAGER]

```

特别说明：

1. 本命令监控中好多命令暂未实现，具体实现以最新发布版本为准。

2. reload @@config, switch @@datasource name:index , 这两个命令再进行处理时, mycat 服务不可用, 谨慎处理, 防止正在提交的事务出错。

show 系统日志

++--

命令: show @@syslog limit

端口号: 该命令工作在 9066 端口, 用来在客户端命令窗口显示系统日志信息,

通常用于远程查看 Mycat-Server 的日志信息

参数: limit= 后接正整数, 该数值用来限定每次最多显示的日志条数

示例:

启动 Mycat-Server ,打开命令端口远程连接 Mycat-Server

mysql -utest -p -P9066 -hlocalhost

输入命令 show @@syslog limit=10 ; 即可显示最多 10 行 Mycat-Server 日志记录信息

使用 show @@help ; 命令可查看该命令的功能描述信息

Sql/SlowSql/SqlNum 统计命令

1、清除缓存

命令: reload@@user_stat

端口号: 该命令工作在 9066 端口, 用来将客户端执行 show @@sql ; show @@sql.sum ; show

@@slow.success ;

命令之后所缓存的信息清空;

参数: 无参数

2、Sql 统计示例

示例:

启动 Mycat-Server ,

打开命令端口远程连接 Mycat-Server 的管理端口

mysql -utest -p -P9066 -hlocalhost

再打开一个新的命令端口, 远程连接 Mycat-Server 的 SQL 操作端口

```
mysql -utest -p -P8066 -hlocalhost
```

在 8066 端口命令窗口中执行 SQL 语句操作，然后在 9066 端口命令窗口分别执行如下命令

```
show @@sql;
```

```
show @@sql.slow;
```

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

会显示出各自的提示信息，信息分别记录了 Mycat-Server 8066 端口上刚刚执行的 SQL 操作信息

在 9066 端口命令窗口中执行命令

```
reload @@user_stat;
```

该命令会将刚刚缓存的 {show @@sql, show @@sql.slow, show @@sql.sum} 记录信息全部清空

在 9066 端口，再次执行 show @@sql; 该命令，则会显示 "Empty set <0.00 sec>" 的提示信息

使用 show @@help; 该命令可以查看该功能的描述信息

++--

命令: show @@sql;

端口号: 该命令工作在 9066 端口，用来记录用户通过本地 8066 端口向 Mycat-Server 发送的 SQL 请求执行信息

信息包括有 ID 值，执行 SQL 语句的用户名称，执行的 SQL 语句，命令执行的起始时间，命令执行消耗时间

参数: 无参数

示例:

启动 Mycat-Server ,

打开命令端口远程连接 Mycat-Server 的管理端口

```
mysql -utest -p -P9066 -hlocalhost
```

再打开一个新的命令端口，远程连接 Mycat-Server 的 SQL 操作端口

```
mysql -utest -p -P8066 -hlocalhost
```

在 8066 连接的命令窗口中根据本地数据库表，输入相关的 SQL 语句

```
select * from TESTDB.company;
```

在 9066 连接的命令窗口输入命令

show @@sql;

```
mysql> show @@sql ;
+-----+-----+-----+-----+-----+
| ID      | USER | SQL                                     | START_TIME | EXECUTE_TIME |
+-----+-----+-----+-----+-----+
| 0       | test | select * from TESTDB.company          | 1449642386599 | 2            |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

将会显示出刚刚执行 SQL 语句详细的相关信息，ID 指的是？ USER 是通过 8066 远程连接到 Mycat-Server 的用户名称；

START_TIME, EXECUTE_TIME 分别是 SQL 语句的起始时间和从开始到结束命令执行消耗总时间。

++--

命令: show @@sql.slow;

端口号: 该命令工作在 9066 端口，是用来将用户通过 8066 端口向 Mycat-Server 发送的请求执行 SQL 语句中超过慢 SQL 时间阈值的

SQL 语句信息；

在这里首先应该明确的是何为'慢 SQL'，所谓的慢 SQL 是执行时间相对时间阈值耗时较长的 SQL 语句，在 mycat-1.4.1 版本中

增设该功能是为了方便用户从本地执行的 SQL 命令中筛选出耗时较长的 SQL 语句出来，针对耗时长久的 SQL 语句做出优化处理。

那么如何设定'慢 SQL'的时间阈值呢？这个命令稍后立即会介绍给大家

参数: 无参数

示例:

启动 Mycat-Server，

打开命令端口远程连接 Mycat-Server 的管理端口

mysql -utest -p -P9066 -hlocalhost

再打开一个新的命令端口，远程连接 Mycat-Server 的 SQL 操作端口

mysql -utest -p -P8066 -hlocalhost

为了方便演示，我们在 9066 端口中将'慢 SQL' 的时间阈值设定为 0 (ms)

(这样无论执行那种 SQL 语句，相关信息均会被当做慢 SQL 信息记录下来)

reload @@sqlslow=0 ; # 是的这条命令就是刚才提到的'如何设定慢 SQL' 时间阈值的命令

```
mysql> reload @@sqlslow=0 ;
Query OK, 1 row affected (0.00 sec)
Reset show @@sql.slow time success
```

然后切换到 8066 端口，根据本地数据库的情况执行一条 SQL 语句

select * from TESTDB.company ;

再将窗口切换到 9066 端口中，输入如下命令

show @@sql.slow ;

```
mysql> show @@sql.slow ;
+-----+-----+-----+-----+-----+
| USER | DATASOURCE | START_TIME | EXECUTE_TIME | SQL |
+-----+-----+-----+-----+-----+
| test | NULL | 1449643296200 | 2 | select * from TESTDB.compan |
y |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

值得注意的地方就是，现在 9066 窗口中显示出来的信息，是从设定阈值之后的时间点开始，本地 8066 端口上执行的所有时间超过慢 SQL 阈值的所有 SQL 语句信息，而在设定阈值之前 8066 端口上执行的所有符合新阈值的语句是不会被显示的

++--

命令: show @@sql.sum ;

端口号: 该命令工作在 9066 端口，是用来向用户展示本地 8066 号端口上执行的 SQL 命令的统计信息数据

参数: 无参数

示例:

启动 Mycat-Server ,

打开命令端口远程连接 Mycat-Server 的管理端口

mysql -utest -p -P9066 -hlocalhost

再打开一个新的命令端口，远程连接 Mycat-Server 的 SQL 操作端口

```
mysql -utest -p -P8066 -hlocalhost
```

在 8066 端口中执行 SQL 语句

```
select * from TESTDB.company ;
```

在 9066 端口命令窗口中输入命令

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

```
mysql> show @@sql.sum ;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | USER | R | W | R% | MAX | TIME_COUNT | TTL_COUNT | LAST_TIME |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | test | 2 | 0 | 1.00 | 1 | [0, 0, 2, 0] | [2, 0, 0, 0] | 1449643296202 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

其中 R,W 分别记录的是当前用户 (USER:test) 在 8066 端口的命令窗口中执行的 SQL 语句中, 有多少条读取数据库信息的语句, 有多少条执行写操作的语句, R% 是读写操作中读操作所占百分比;

其中, TIME_COUNT , 记录的是总共执行 SQL 操作的次数, TTL_COUNT , 记录的是?

LAST_TIME, 记录的是最后一次执行的时间戳, 该事件是相对于当前系统时间的;

而通过 'show @@time.current ;' 命令即可显示出当前时间, 二者的差值就是最后执行 SQL 命令据当前的时间;

新增 explain2 命令

Explain2 datanode=? Sql=?

```
mysql> explain2 datanode=dn1133 sql=SELECT tuserid FROM tparbyday1 where tuserid = '9999';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tparbyday1 | ALL | NULL | NULL | NULL | NULL | 44863 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

第 9 章 压缩协议支持

9.1 压缩协议支持

Mycat 从 1.4 开始支持 mysql 的压缩协议，在查询返回大的结果集和 load data 大量数据的性能提升比较明显。可以大大节省网络流量，但会消耗少量 cpu 资源。如果要启用压缩协议，则客户端、mycat、mysql 三者都启用才行。

9.2 配置说明

Mycat 可以在 server.xml 中配置 1 启用。

客户端如果是 mysql 命令行，则加参数 -C 启用压缩协议。

客户端如果是 jdbc 则在 jdbc 的 url 上加上参数 useCompression=true，例如：

```
jdbc:mysql://127.0.0.1:8066/base?useCompression=true
```

Mysql 服务端一般默认开启压缩协议支持，具体参考对应版本的官方文档。

9.3 压缩性能测试

一般网路条件越差，性能提升越明显。

测试环境客户端在电信网路，通过 vpn 连接到教育网内 mycat 服务器。

测试 load data local 一百万数据到 5 个分片，未开启压缩耗时 179 秒，开启压缩后耗时 70 秒，性能提升 2 倍多。

9.4 mysql 压缩协议

压缩协议属于 mysql 通讯协议的一部分，要启用压缩协议传输功能，前提条件客户端和服务端都必须支持 zlib 算法。

mysql 起始握手，先由 server 发起，client 分析并回应自己同意的特性，然后双方依照这些特性处理数据包。

通信时是否采用压缩会改变数据包的字节变化。

客户端的特性在首个回应（既握手包）服务器中体现，如：是否开启压缩、字符集、用户登录信息等。

1.未采用压缩时，客户端向服务器发送的包格式：

格式：3*byte,1*byte,1*byte,n*byte

表示：消息长度,包序号,请求类型,请求内容

2.采用压缩后，客户端向服务器发送的包格式：

格式：3*byte,1*byte,3*byte,n*byte

表示：消息长度，包序号，压缩包大小，压缩包内容

当压缩包大小为 0x00 时，表示当前包未采用压缩，则 n*byte 内容为原协议包内容

当压缩包大小大于 0x00 时，表示当前包已采用 zlib 压缩，则 n*byte 内容，先解压缩，解压后内容为原协议包内容

mysql 内部有一个约定，如果原协议包小于 50 字节时，对内容不压缩而保持原貌的方式，而 mysql 此举是为了减少 CPU 性能开销。mysql 的压缩协议对原协议是透明的，也就是说一个压缩包里可能包括一个或多个原协议包，甚至可能包括一些不完整的原协议包在内。也就是一个原协议包可能会被拆分到 2 个压缩包中。

第 10 章 Mycat-Web

Mycat,mycat eye 依赖 jdk1.7+环境, jdk 下载地址为:

<http://www.oracle.com/technetwork/java/javase/downloads/>

10.1.1 Zookeeper 环境要求

Mycat eye 需要 Zookeeper 作为配置中心

zookeeper-3.4.6.tar.gz

10.1.2 Zookeeper 配置

1. 解压 zookeeper-3.4.6.tar.gz;
2. zookeeper-3.4.6\conf 目录下把 zoo_sample.cfg 修改为 zoo.cfg;
3. 启动 zookeeper

Windows 操作系统启动命令: zookeeper-3.4.6\bin\zkServer.bat

Linux 操作系统启动命令: zookeeper-3.4.6\bin\zkServer.sh start

10.2 Mycat eye 环境部署

10.2.1 软件清单

Windows 版本:

Mycat-web-1.0-SNAPSHOT-20151208180035-win.zip

Linux 版本:

Mycat-web-1.0-SNAPSHOT-20151208180035-linux.tar.gz

下载地址: <https://github.com/MyCATapache/Mycat-download/tree/master/mycat-web-1.0>

10.2.2 运行

- 1: 解压

解压后生成 mycat-web 目录, 目录结构如下:

名称	修改日期	类型
 etc	2015/10/20 10:44	文件夹
 lib	2015/10/20 10:44	文件夹
 mycat-web	2015/12/8 18:00	文件夹
 rainbow.log	2015/12/8 18:33	文本文档
 readme.txt	2015/10/20 10:44	文本文档
 start.bat	2015/10/20 10:44	Windows 批处理...
 start.jar	2015/10/20 10:44	Executable Jar File

2: 先启动 zookeeper;

3: 然后启动 mycat eye

Windows 版本:

```
start.bat
```

Linux 版本:

```
start.sh
```

10.2.3 访问 mycat eye

访问地址: <http://localhost:8082/mycat/>

10.2.4 mycat.properties 配置

方法 1: 如果 zookeeper 连接不上

可以修改 mycat-web\mycat-web\WEB-INF\classes\ mycat.properties 配置文件

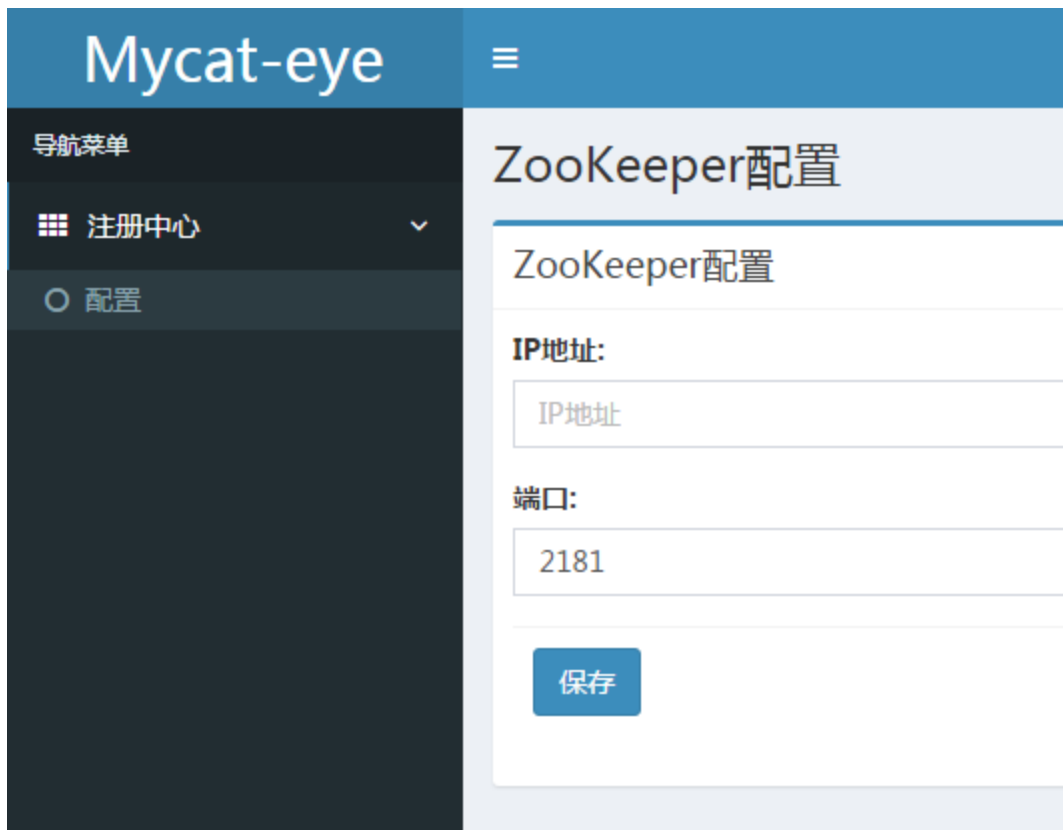
```
zookeeper=127.0.0.1:2181
```

配置下 zookeeper 的服务器和端口

方法 2: 访问 <http://localhost:8082/mycat/>

点击注册中心-配置

输入 Zookeeper 的 IP 地址和端口即可。



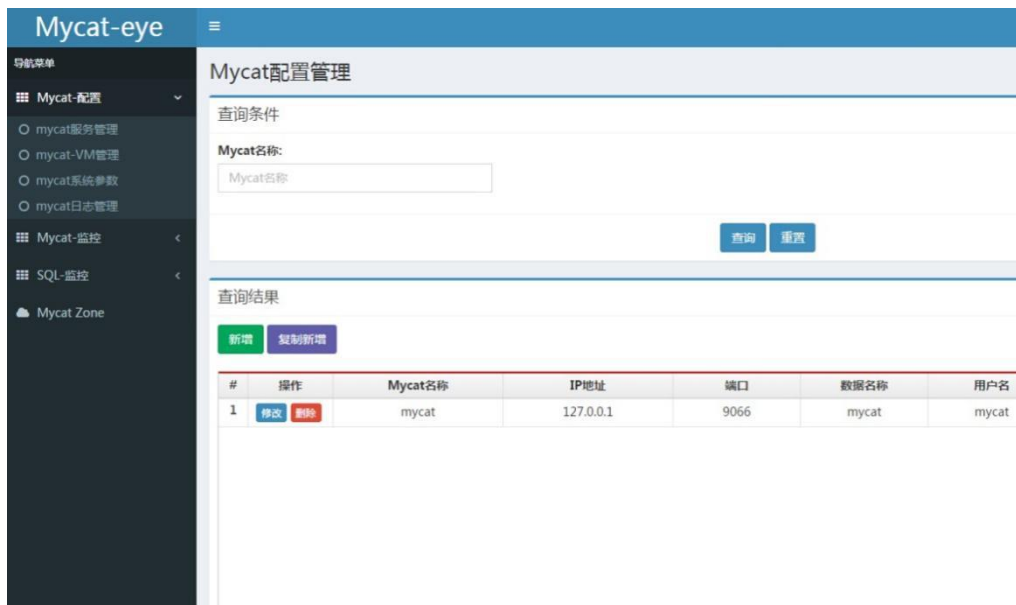
10.3 Mycat 配置说明

Mycat 主要是维护 mycat 节点和 mycat jmx 信息，有四个菜单

10.3.1 Mycat 服务管理

列表

显示所有管理的 mycat 服务，如下图所示：



新增

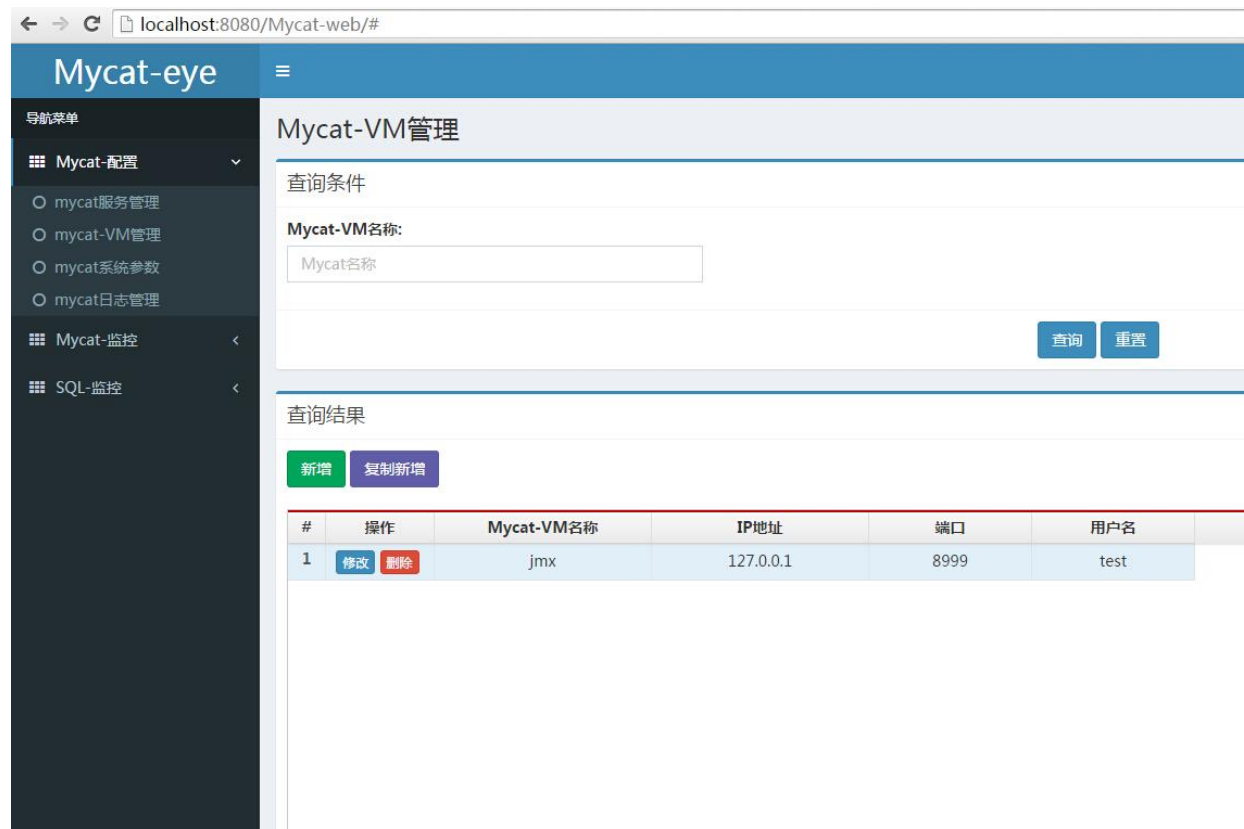
新增一个 mycat 服务，包括名称，IP，端口，数据库名称，用户名和密码



10.3.2 Mycat VM 管理

列表

显示所有管理的 mycatVM 服务



新增

新增一个 mycatVM 服务，包括名称，IP，端口，用户名和密码

Mycat-eye

≡

导航菜单

MyCat-配置

mycat服务管理

mycat-VM管理

mycat系统参数

mycat日志管理

MyCat-监控

SQL-监控

MyCat Zone

MyCat-VM管理

Jmx配置管理

MyCat-VM名称(必须为英文哦):

Jmx名称(必须为英文哦)

IP地址:

IP地址

端口:

端口

用户名:

admin

密码:

.....

保存

返回列表

10.3.3 Mycat 系统参数

查询 mycat 服务的系统参数

条件：选择 3.1 中的 mycat 服务

Mycat server 命令行输入：

```
show @@sysparam
```

Mycat 系统参数总共有 18 项，如下图：

mycat系统参数

mycat

#	参数名称	值	描述
1	processors	2	主要用于指定系统可用的线程数，默认值为Runtime.getRuntime().availableProcessors()方法返回的值。主要影响processorBufferPool、processorBufferLocalPercent、processorExecutor属性。NIOProcessor的个数也是由这个属性定义的，所以调优的时候可以适当的调高这个属性。
2	processorBufferChunk	4096B	指定每次分配Socket Direct Buffer的大小，默认是4096个字节。这个属性也会影响buffer pool的长度。
3	processorBufferPool	8192000B	指定bufferPool计算 比例值。由于每次执行NIO读、写操作都需要使用到buffer，系统初始化的时候会建立一定长度的buffer池来加快读、写的效率，减少建立buffer的时间
4	processorBufferLocalPercent	100	就是用来控制分配这个pool的大小用的，但其也并不是一个准确的值，也是一个比例值。这个属性默认值为100。线程缓存百分比 = bufferLocalPercent / processors属性。

#	参数名称	值	描述
5	processorExecutor	4	主要用于指定NIOProcessor上共多少个线程，mycat在需要处理一些异步逻辑的时候，这个连接池的使用频率不是很大了
6	sequenceHandlerType	本地文件方式	指定使用Mycat全局序列的类型。
7	Mysql_packetHeaderSize	4B	指定Mysql协议中的报文头长度。
8	Mysql_maxPacketSize	16M	指定Mysql协议可以携带的数据最大长度。
9	Mysql_idleTimeout	30分钟	指定连接的空闲超时时间。某连接超过了空闲时间，那么这个连接会被回收。
10	Mysql_charset	utf8	连接的初始化字符集。默认为utf8。
11	Mysql_txIsolation	REPEATED_READ	前端连接的初始化事务隔离级别，默认为REPEATABLE_READ，传递过来的属性对后端数据库连接生效。
12	Mysql_sqlExecuteTimeout	300秒	SQL执行超时的时间，Mycat会检查是否超时。

12	Mysql_sqlExecuteTimeout	300秒	SQL执行超时的时间，Mycat会个时间则会直接关闭这连接。默
13	Mycat_processorCheckPeriod	1秒	清理NIOProcessor上前后端空闲
14	Mycat_dataNodeIdleCheckPeriod	300秒	对后端连接进行空闲、超时检查
15	Mycat_dataNodeHeartbeatPeriod	10秒	对后端所有读、写库发起心跳的
16	Mycat_bindIp	0.0.0.0	mycat服务监听的IP地址，默认
17	Mycat_serverPort	8066	mycat的使用端口，默认值为80
18	Mycat_managerPort	9066	mycat的管理端口，默认值为90

10.3.4 Mycat 日志管理

查询 Mycat 最新的日志，默认显示 50 条

Mycat server 命令行输入：

```
show @@syslog limit=50
```

Mycat监控		
mycat		
#	日期	日志
1	12/09 09:36:42.236	INFO [\$ _NIOREACTOR-3-RW] (FrontendAuthenticator.java:193) -[thread=\$_NIOREACTOR-3-RW,class=ManagerConnection,id=3117,host=10.88.7.4,port=9066,schema=TESTDB]'test' login success
2	12/09 09:36:42.231	INFO [\$ _NIOREACTOR-2-RW] (FrontendAuthenticator.java:193) -[thread=\$_NIOREACTOR-2-RW,class=ManagerConnection,id=3116,host=10.88.7.4,port=9066,schema=TESTDB]'test' login success
3	12/09 09:36:42.226	INFO [\$ _NIOREACTOR-1-RW] (FrontendAuthenticator.java:193) -[thread=\$_NIOREACTOR-1-RW,class=ManagerConnection,id=3115,host=10.88.7.4,port=9066,schema=TESTDB]'test' login success
4	12/09 09:36:42.220	INFO [\$ _NIOREACTOR-0-RW] (FrontendAuthenticator.java:193) -[thread=\$_NIOREACTOR-0-RW,class=ManagerConnection,id=3114,host=10.88.7.4,port=9066,schema=TESTDB]'test' login success
5	12/09 09:36:42.216	INFO [\$ _NIOREACTOR-3-RW] (FrontendAuthenticator.java:193) -[thread=\$_NIOREACTOR-3-RW,class=ManagerConnection,id=3113,host=10.88.7.4,port=9066,schema=TESTDB]'test' login success
6	12/09 09:36:42.214	INFO [\$ _NIOREACTOR-2-RW] (AbstractConnection.java:458) -close connection,reason:quit cmd , [thread=\$_NIOREACTOR-2-RW,class=ManagerConnection,id=3112,host=10.88.7.4,port=9066,schema=TESTDB]

10.4 Mycat 监控

Mycat-eye 中的 mycat 性能监控、JVM 监控中的监控图是基于 jrds 实现;

1. 实现步骤:

1.1 通过 mycat 服务管理创建一个 mycat 监控服务。同时会基于 freemark 模板引擎生成 Jrds 配置信息。目前默认指定路径为: WEB-INF\jrdsconf\hosts 录下。

1.2 在通过点击【mycat 性能监控】菜单时, 页面会调用 jrds 提供的/reload。加载 WEB-INF\jrdsconf\hosts 下的*.xml 如: D_127.0.0.1_9066.xml 文件。

通过调用/graph/hostMycatList?hostprefix=D 加载监控列表 hostprefix 为 hosts 下的文件名称前缀。

10.4.1 Mycat 性能监控

条件: 选择 3.1 中的 mycat 服务, 包括:

Mycat 流量分析

Mycat 连接分析

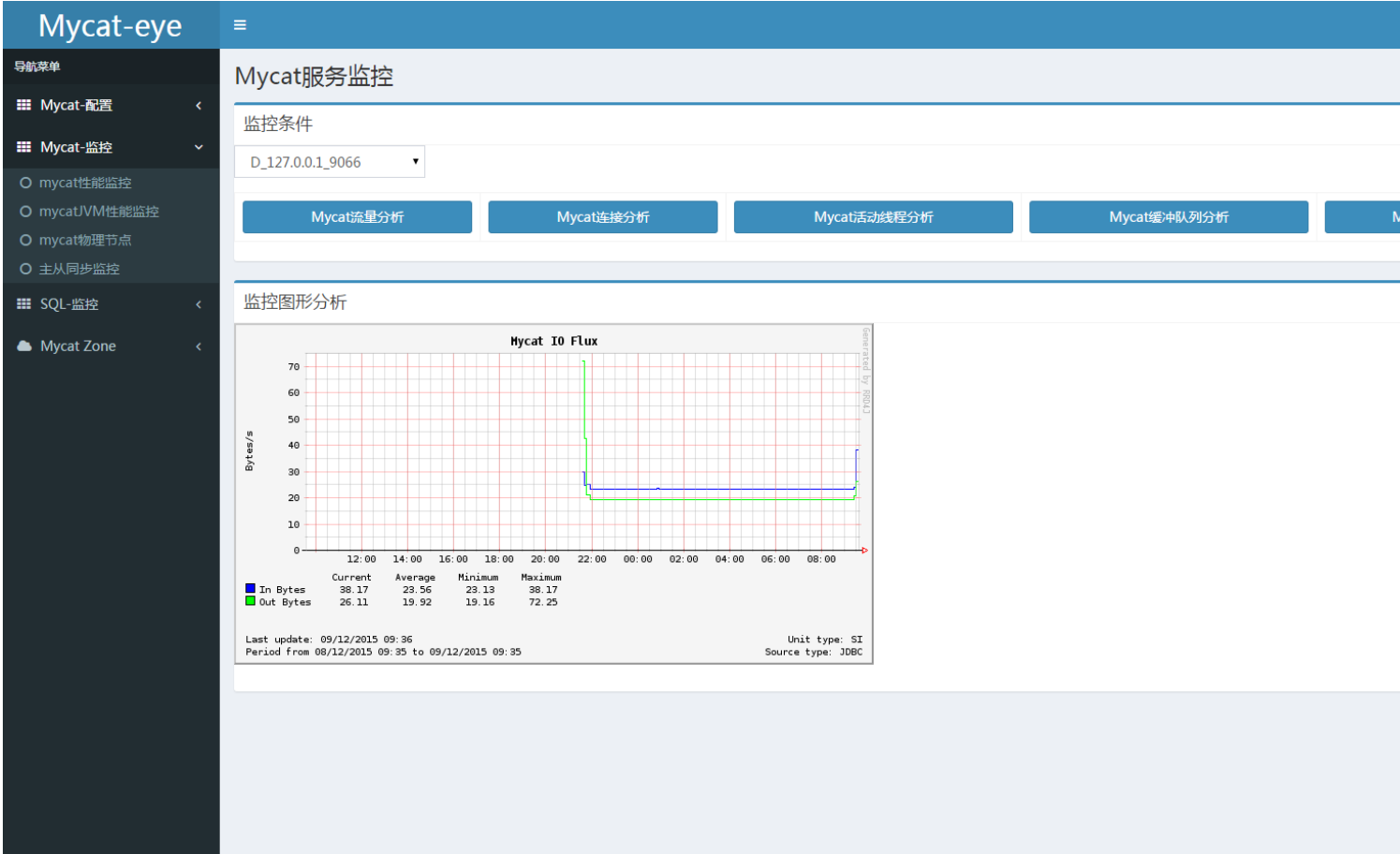
Mycat 活动线程分析

Mycat 缓冲队列分析

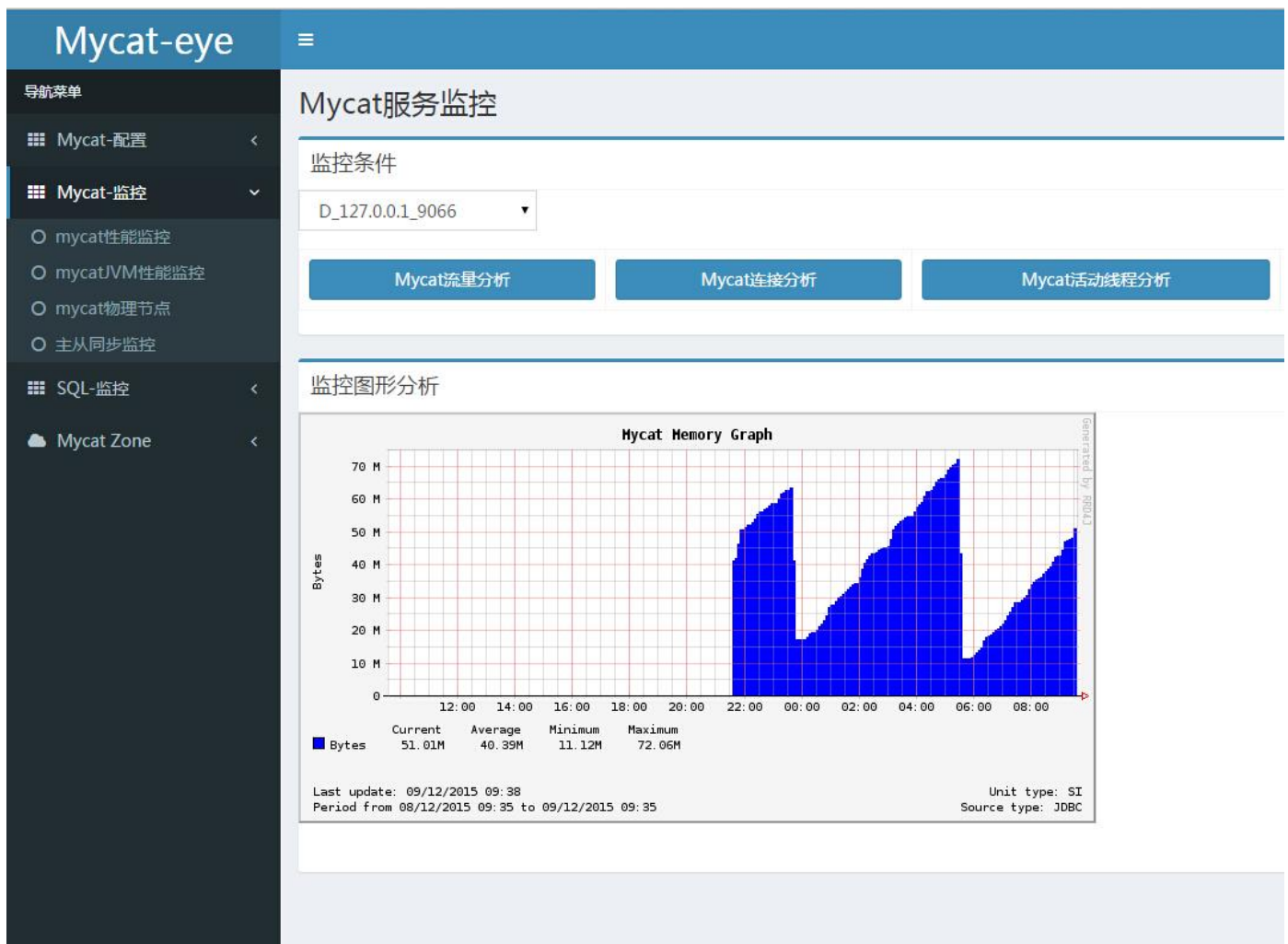
MycatTPS 分析

Mycat 内存分析

如图所示:







10.4.2 Mycat JVM 性能监控

1) JVM 监控需要在 Mycat Server 中配置启动参数:

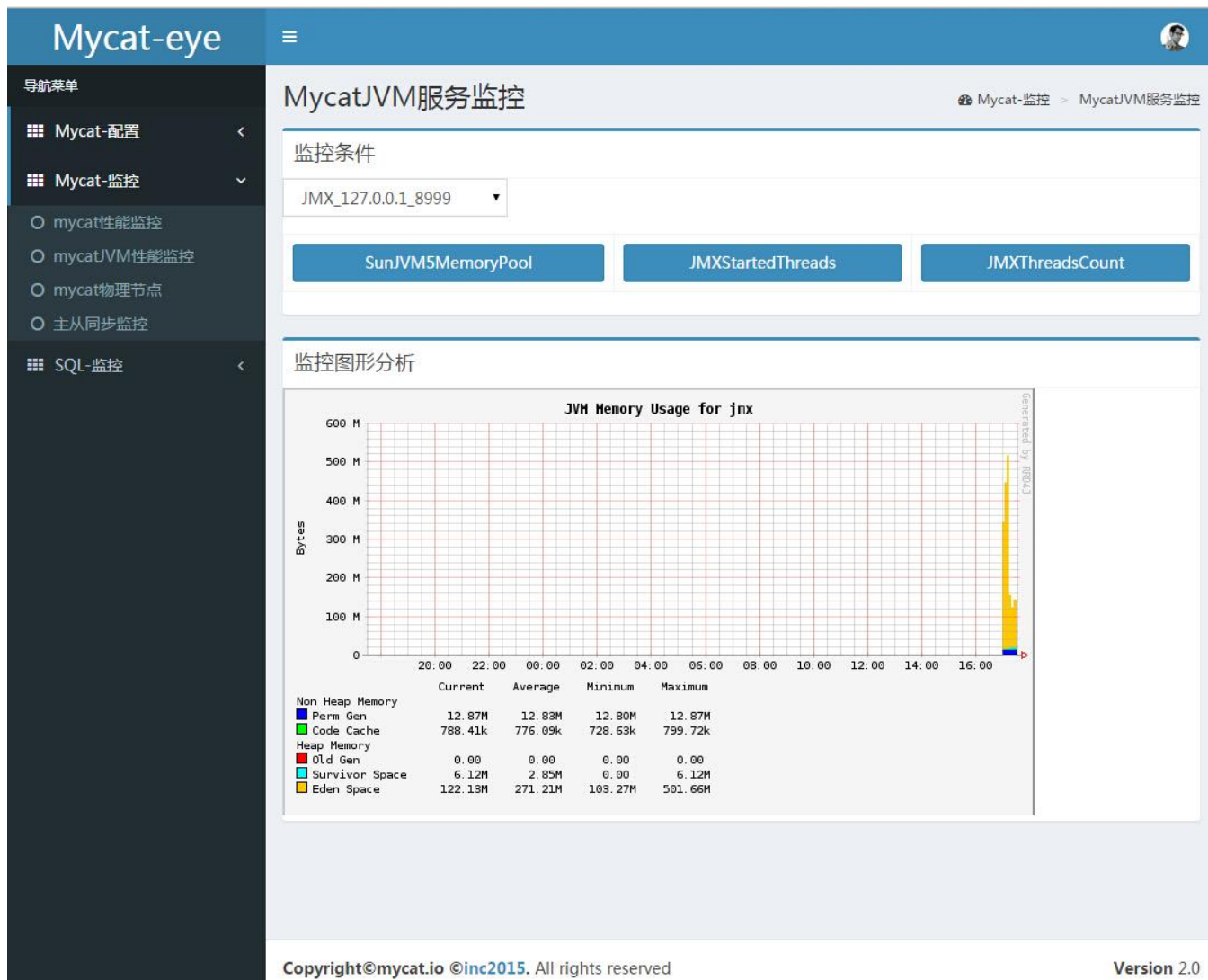
```
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -  
XX:CMSFullGCsBeforeCompaction=0 -XX:CMSInitiatingOccupancyFraction=70 -  
Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=8999 -  
Dcom.sun.management.jmxremote.authenticate=false -  
Dcom.sun.management.jmxremote.ssl=false
```

或者直接用配置好的 startup_nowrap.bat

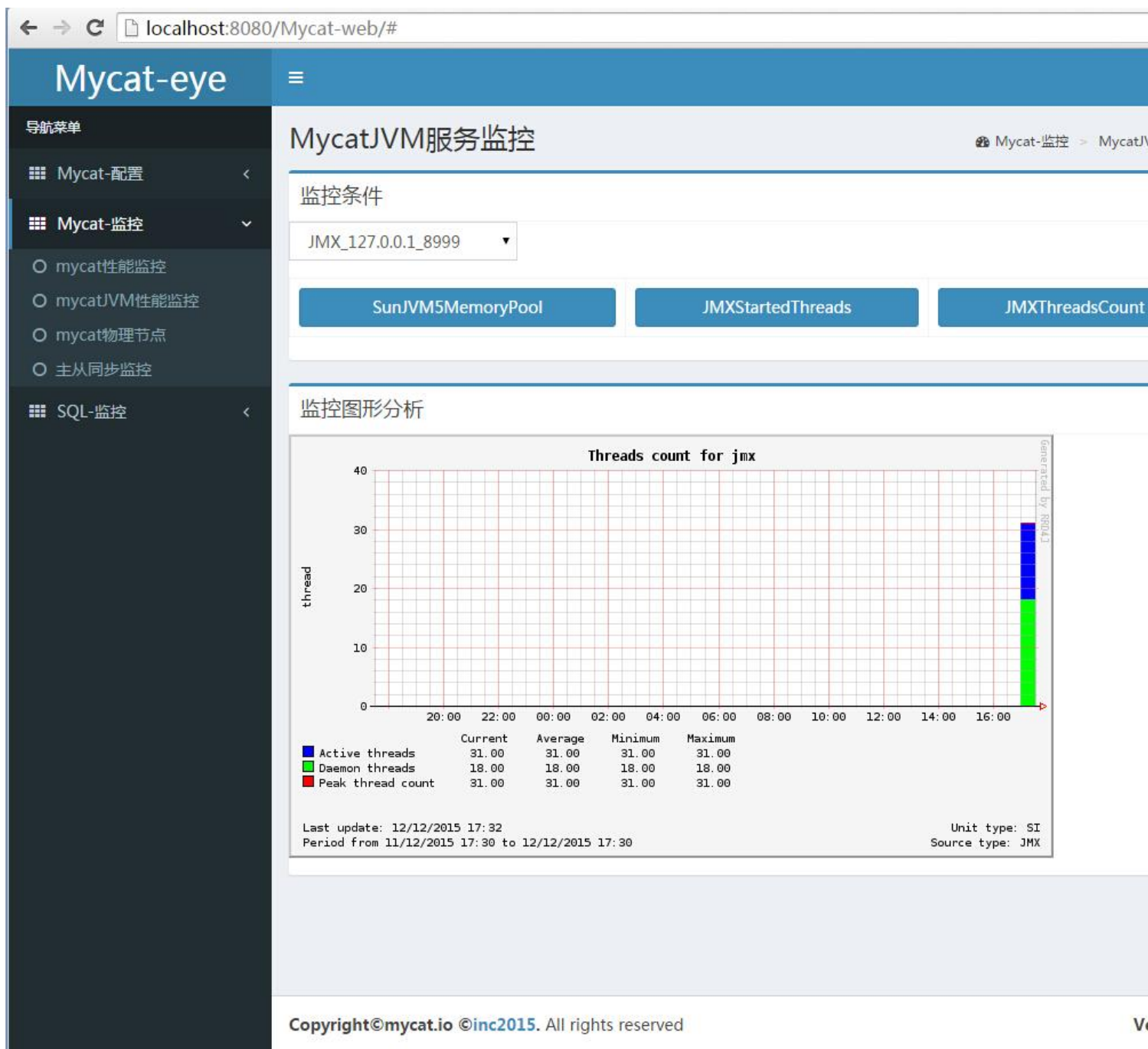


startup_nowrap.bat

2) 选择 3.2 中管理的 JVM

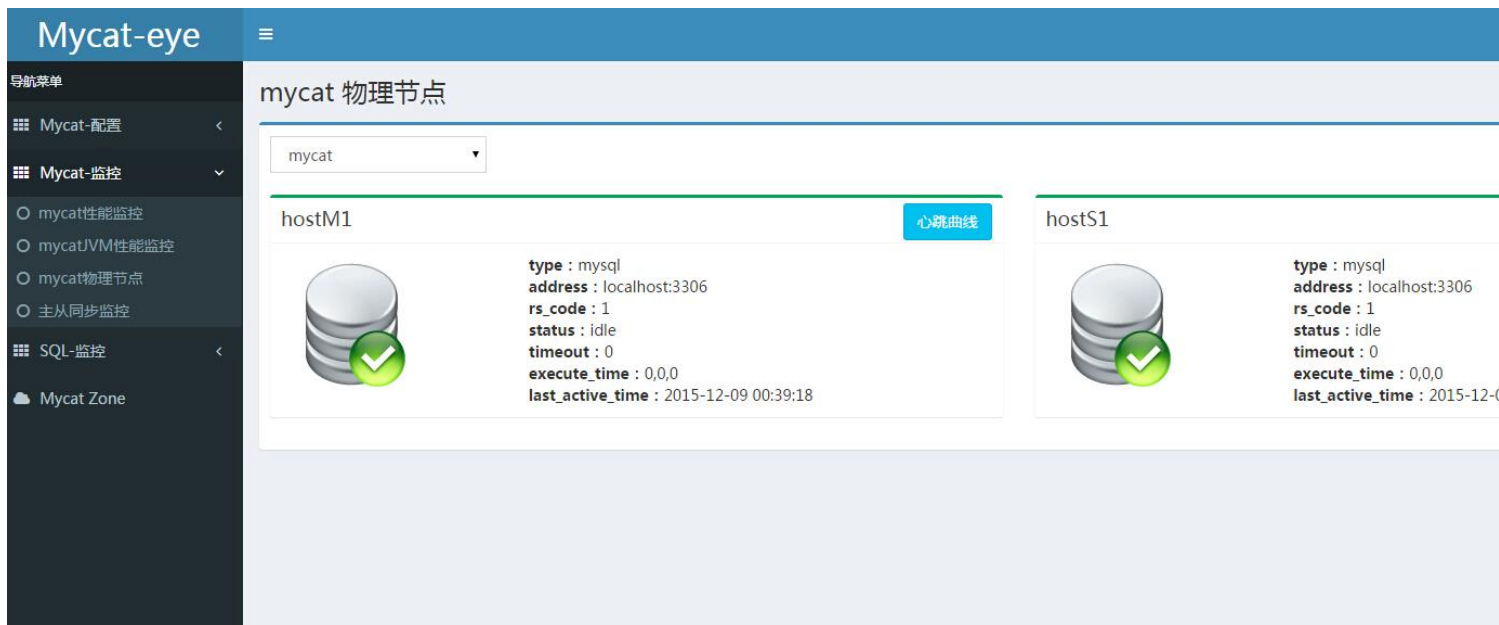






10.4.3 Mycat 物理节点

1) 选择 3.1 中管理的 Mycat，自动查询出 mysql 节点信息



2) 心跳曲线



10.4.4 Mycat 主从同步监控

1) 选择 3.1 中管理的 Mycat，自动查询出 mysql 主从节点信息

监控 mysql 的主从同步曲线

10.5 SQL 监控

SQL 监控，主要是监控和分析 SQL 语句

10.5.1 SQL 统计

按用户统计 SQL 读写比例，时间分布耗时。

Mycat server 命令行输入：

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

如图所示：



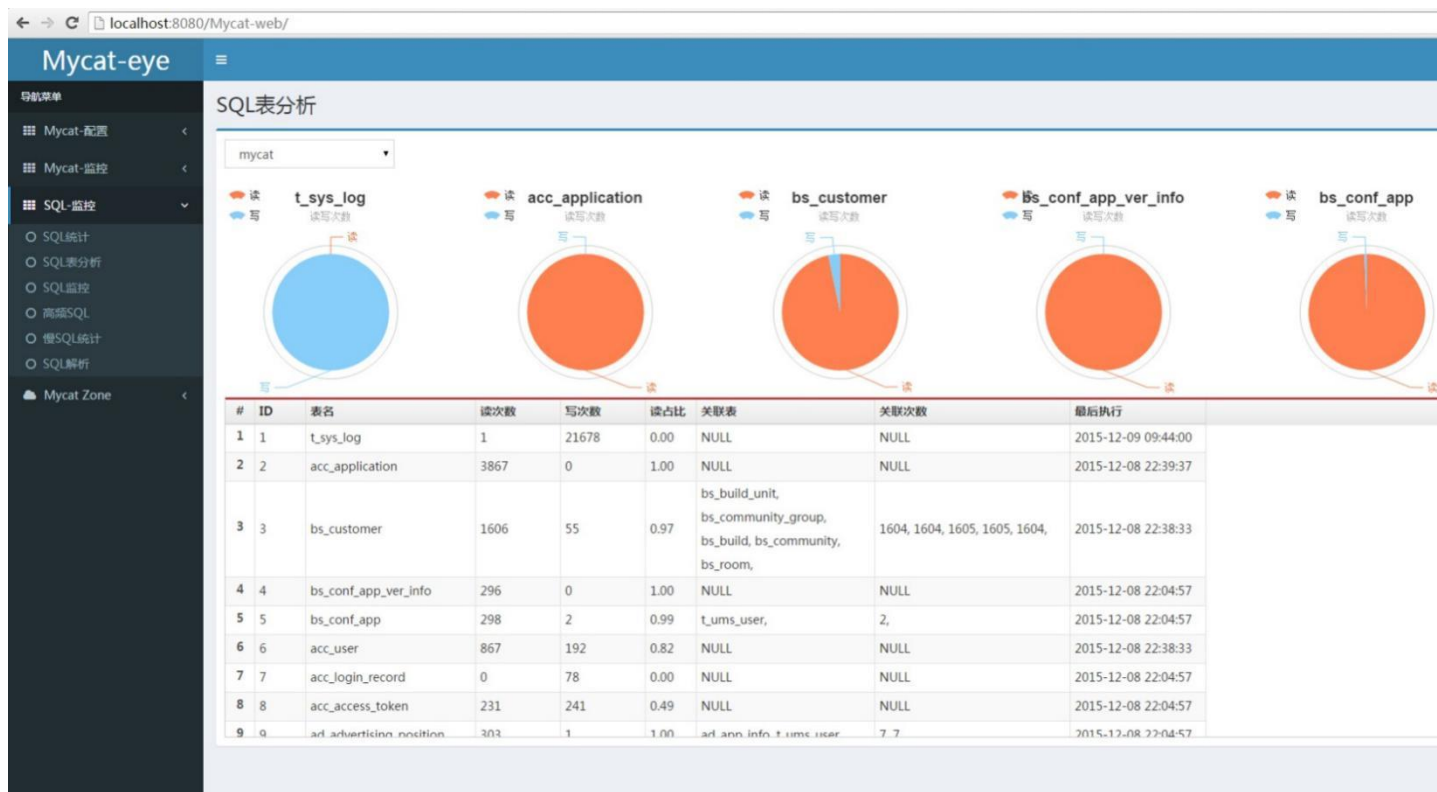
10.5.2 SQL 表统计

统计表的读写比例，表之间的关系；

Mycat server 命令行输入：

```
show @@sql.sum.table
```

如图所示：



10.5.3 SQL 监控

监控业务系统执行的 SQL 语句，默认 50 条；

Mycat server 命令行输入：

```
show @@sql
```

如图所示：

Mycat-eye						
SQL监控						
mycat 导出CSV						
#	ID	用户	SQL	耗时(ms)	执行时间	
6	4	test	(110006894, null, 1, 0, 'log', ' ', ' ', 'com.agile.api.action.DispatcherAction', 'DispatcherAction-callback call:[callBackService.invokeCallBackto spend:8ms]', '2015-12-09 09:43:18','2015-12-09 09:43:18','10.2.35.21',null,null)	1	2015-12-09 09:44:00	
7	3	test	select a.id, a.order_no as orderNo, a.pay_order_no as serialNumber, a.user_id as userId, a.mobile, a.pay_way as payWay, a.merchant_id as merchantId, a.pay_total_count as payTotalCount, a.balance_deduction as balanceDeduction, a.point_deduction as pointDeduction, a.use_point as usePoint, a.ratio, a.other_pay as otherPay, a.pay_time as payTime, a.pay_status as payStatus, a.quantity, a.body, a.trade_type, a.create_time as createTime, a.update_time as updateTime, a.app_id as appId, a.is_refund_time_out as isRefundTimeOut, a.third_party_no as thirdPartyNo, a.subject, a.real_pay as realPay, a.pay_type as payType, a.request_number as requestNumber, a.request_order_no as requestOrderNo, a.failure_time as failureTime, a.status as status, a.send_time as sendTime, a.beneficiary_uid as beneficiaryUid, a.type as type from pay_ment a WHERE	0	2015-12-09 09:44:00	

10. 5. 4 高频 SQL

SQL 进过归并，统计执行频率高的 SQL 语句；

Mycat server 命令行输入：

show @@sql.high

如图所示：

#	频率	SQL	耗时(ms)	最大耗时(ms)	最小耗时(ms)	最后执行
2	17714	SELECT COUNT(*) FROM t_ums_permission p WHERE 1 = 1 AND p.parent_id = ? AND p.type = ?	0	56	0	2015-12-09 09:33:23
3	3802	SELECT id, app_name AS appName, app_id AS appId, app_secret AS appSecret, description, is_disabled AS isDisabled, create_time AS createTime, app_type AS appType FROM acc_application WHERE app_id = ?	0	34	0	2015-12-08 22:39:37
4	2888	SELECT a.content FROM sa_seller_dis_content a WHERE a.seller_id = ? AND a.deleted_flag = ? AND a.status = ? AND a.start_time <= now() AND a.end_time > now()	1	23	1	2015-12-08 22:39:37
5	1642	SELECT a.id, a.serial_number_key AS serialNumberKey, a.serial_number_tmpl AS serialNumberTmpl, a.last_generate_number AS lastGenerateNumber, a.last_generate_time AS lastGenerateTime, a.comments FROM serial_number_helper a WHERE a.serial_number_key = ? LIMIT ?	0	13	0	2015-12-08 22:38:33
6	1638	UPDATE serial_number_helper a SET a.serial_number_key = ?, a.serial_number_tmpl = ?,	0	85688	0	2015-12-08 22:38:33

10.5.5 慢 SQL 统计

默认查询耗时 1000ms 的 SQL 语句，

Mycat server 命令行输入：

```
show @@sql.slow
```

如图所示：

#	用户	节点	开始时间	执行时间	SQL
1	test		2015-12-08 14:38:59	85688	update serial_number_helper a SET a.serial_number_key = 'base_payment_orderNumber', a.serial_number_tmpl = 'string:202,date:yyyyMMdd,sequence:6 month,rnd:4', a.last_generate_number = 357, a.last_generate_time = '2015-12-08 14:39:39', a.comments = '支付流水号生产规则说明' where a.id = 27
2	test		2015-12-08 14:39:23	56434	update serial_number_helper a SET a.serial_number_key = 'base_balance_orderId', a.serial_number_tmpl = 'string:202,date:yyyyMMdd,sequence:6 month,rnd:4', a.last_generate_number = 333, a.last_generate_time = '2015-12-08 14:40:03', a.comments = '余额流水号生产规则说明' where a.id = 12
					update serial_number_helper a SET a.serial_number_key =

10.5.6 SQL 解析

解析 SQL 语句，分 2 步执行；

第一步: **explain**

解析 SQL 语句的路由信息

第二步: **explain2 datanode = dn1 sql =**

制定节点，然后到 **mysql** 服务器执行

Explain 命令

Mycat server 命令行输入：

第一步: **explain select * from t_sys_log;**

第二步: **explain2 datanode = dn1 sql = SELECT * FROM t_sys_log LIMIT 10000**

如图所示：

Mycat-eye

≡

导航菜单

MyCat-配置

MyCat-监控

SQL-监控

SQL统计

SQL表分析

SQL监控

高频SQL

慢SQL统计

SQL解析

MyCat Zone

SQL解析

数据源:

mycat

sql:

select * from t_sys_log

解析

重置

解析结果

#	操作	datanode	SQL
1	解析	dn1	SELECT * FROM t_sys_log LIMIT 10000

SQL解析

数据源:

mycat

解析结果

#	操作	datanode
1	解析	dn1

SQL解析信息

#	id	select_type	table	type	possible_keys	key	key_len	ref	rows
1	1	SIMPLE	t_sys_log	ALL					102025

关闭

第 11 章 MyCAT 对存储过程的支持

目前从 1.6 版本开始完整支持 mysql 和 oracle 的存储过程，调用原理需要使用注解，把存储过程的调用当做普通 sql 来调用，注意在程序中别用以前存储过程的调用方式，要按照普通查询 sql 的调用方式。返回结果从 resultset 里取。

完美支持以下三种情况：

1. 无返回值

```
/*#mycat: sql=SELECT * FROM test */call p_test(1,@pout)
```

2. 返回普通 out 参数

```
/*#mycat: sql=SELECT * FROM test */ set @pin=111;call p_test(@pin,@pout);select @pout
```

3. 返回结果中有结果集时，则必须加注解，且注解中必须在 list_fields 中包括所有结果集参数名称，以逗号隔开 结果集参数必须在最后

```
/*#mycat: sql=SELECT * FROM test where id=1 ,list_fields='@p_CURSOR,@p_CURSOR1' */
```

第 12 章 MyCAT 对 zookeeper 的支持

12.1 mycat 配置 zk 化

1.6 对 zk 模块进行了重构，同时支持 zk 的 watch 机制，会将所有 zk 上的变动同步到本地配置。

1. 配置 myid.properties

loadZk=true	是否启用 zk
zkURL=127.0.0.1:2181	zk 地址支持多个
clusterId=mycat-cluster-1	mycat 集群名称，即一组相同的 mycat 为一个集群，一个集群名称配置唯一
myid=mycat_fz_01	集群内部 mycat 节点的 id，请保持集群内唯一

2. 在 conf 下的 zkconf 下配置常用的 schema1 rule server 等 xml 文件

执行 bin 目录下的 init_zk_data.bat 或者 init_zk_data.sh, 会自动将 zkconf 下的所有配置文件上传到 zk。

3. mycat 启动时如果 loadzk=true 会自动从 zk 下载配置文件覆盖本地配置。

12.2 zk 协调后端 mysql 切换

server.xml 中配置 `<property name="useZKSwitch">true</property>`, 则 mycat 在进行切换时会自动通过 zk 协调, 保证同一个集群下的 mycat 都切换到一致的状态

生产实践篇

第 1 章 生产实践案例-Mycat 读写分离案例

目前有大量 Mycat 的生产实践案例是属于简单的读写分离类型的，此案例主要用到 Mycat 的以下特性：

- 读写分离支持；
- 高可用。

大多数读写分离的案例是同时支持高可用性的，即 Mycat+MySQL 主从复制的集群，并开启 Mycat 的读写分离功能，这种场景需求下，Mycat 是最为简单并且功能最为丰富的一类 Proxy，正常情况下，配置文件也最为简单，不用每个表配置，只需要在 schema.xml 中的元素上增加 dataNode=“defaultDN”属性，并配置此 dataNode 对应的真实物理数据库的 database，然后 dataHost 开启读写分离功能即可。

若不要自动切换功能，即 MySQL 写节点宕机后不自动切换到备用节点，则如下配置：

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="1"
    writeType="0" dbType="mysql" dbDriver="native">
    <heartbeat>select user()</heartbeat>
    <!-- can have multi write hosts -->
    <writeHost host="hostM1" url="localhost:3306" user="root"
        password="123456">
        <!-- can have multi read hosts -->
        <readHost host="hostS1" url="localhost2:3306" user="root"
password="123456" />
    </writeHost>
</dataHost>
```

如果要实现自动切换到备用节点，则如下配置：

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="1"
    writeType="0" dbType="mysql" dbDriver="native">
    <heartbeat>select user()</heartbeat>
```

```

        <!-- can have multi write hosts -->

        <writeHost host="hostM1" url="localhost:3306" user="root"

            password="123456" />

<writeHost host="hostS1" url="localhost2:3306" user="root" password="123456" />

</dataHost>

```

此时，第一个 writeHost 故障后，会自动切换到第二个，第二个故障后自动切换到第三个，当你是 1 主 3 从的模式的时候，可以把第一个从节点配置为 writeHost 2，第 2 个和第三个从节点则配置为 writeHost 1 的 readHost，如下所示：

```

<dataHost name="localhost1" maxCon="1000" minCon="10" balance="1"

    writeType="0" dbType="mysql" dbDriver="native">

        <heartbeat>select user()</heartbeat>

        <writeHost host="hostM1" url="localhost:3306" user="root"

            password="123456" >

                <readHost host="hostS2" url="localhost3:3306" user="root"

password="123456" />

                <readHost host="hostS3" url="localhost4:3306" user="root"

password="123456" />

            </writeHost>

        <writeHost host="hostS1" url="localhost2:3306" user="root" password="123456" />

    </dataHost>

```

为了提升查询的性能，有人创新的设计了一种 MySQL 主从复制的模式，主节点为 InnoDB 引擎，读节点为 MyISAM 引擎，经过实践，发现查询性能提升不少。

此外，为了减少主从复制的时延，也建议采用 MySQL 5.6+ 的版本，用 GTID 同步复制方式减少复制的时延，可以将一个 Database 中的表，根据写频率的不同，分割成几个 Database，用 Mycat 虚拟为一个 Database，这样就满足了多库并发复制的优势，需要注意的是，要将有 Join 关系的表放在同一个库中。

最后，对于某些表，要求不能有复制时延，则可以考虑这些表 放到 Gluster 集群里，消除同步复制的时延问题，前提是这些表的修改操作并不很频繁，需要做性能测试，以确保能满足业务高峰。

总结一下，Mycat 做读写分离和高可用，可能的方案很灵活，只有你没想到的，没有做不到的。

第 2 章分表分库案例

2.1 SAAS 多租户案例

SAAS 多租户的案例是 Mycat 粉丝的创新性应用案例之一，思路巧妙并且实现方式简单。

SAAS 应用中，不同租户的数据是需要进行相互隔离的，比较常用的一种方式是不同的租户采用不同的 Database 存放业务数据，常规的做法是应用程序中根据租户 ID 连接到相应的 Database，通常是需要启动多个应用实例，每个租户一个，但这种模式消耗的资源比较多，而且不容易管理，还需要开发额外的功能，以对应租户和部署的应用实例。

在 Mycat 出现以后，有人利用 Mycat 的 SQL 拦截功能，巧妙的实现了 SAAS 多租户特性，传统应用仅做少量的改动，就直接进化为多租户的 SAAS 应用，下面的内容是 Mycat 用户提供的具体细节：

单租户就是传统的给每个租户独立部署一套 web + db。由于租户越来越多，整个 web 部分的机器和运维成本都非常高，因此需要改进到所有租户共享一套 web 的模式（db 部分暂不改变）。

基于此需求，我们对单租户的程序做了简单的改造实现 web 多租户共享。具体改造如下：

1.web 部分修改：

a.在用户登录时，在线程变量（ThreadLocal）中记录租户的 id

b.修改 jdbc 的实现：在提交 sql 时，从 ThreadLocal 中获取租户 id，添加 sql 注释，把租户的 schema 放到注释中。例如：`/*!mycat: schema = test_01 */ sql;`

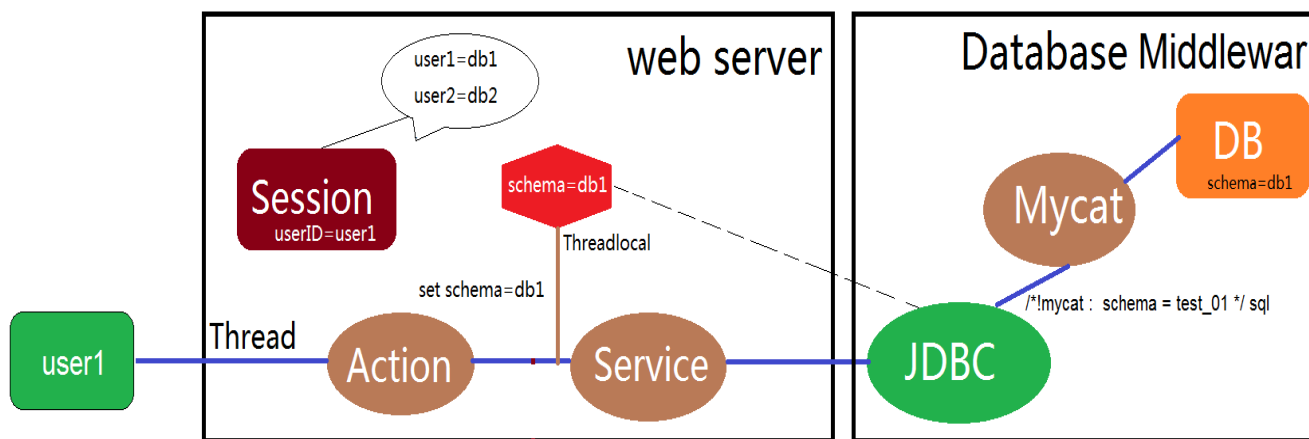
2.在 db 前面建立 proxy 层，代理所有 web 过来的数据库请求。proxy 层是用 mycat 实现的，web 提交的 sql 过来时在注释中指定 schema，proxy 层根据指定的 schema 转发 sql 请求。

此方案有几个关键点：

- ThreadLocal 变量的巧妙使用，与 Hibernate 的事务管理器一样的机制，线程的一个 ThreadLocal 变量中保留当前线程涉及到的数据库连接、事务状态等信息，当 Service 的某个事务托管的业务方法被调用时，Hibernate 自动完成数据库连接的建立或重用过程，当此方法结束时，自动回收数据库连接以及提交事务。在这里，操作数据库的线程中以 ThreadLocal 变量方式放入当前用户的 Id 以及对应的数据库 Schema (Database)，则此线程随后的整个调用方法堆栈中的任何一个点都能获取到用户对应的 Schema，包括在 JDBC 的驱动程序中。

- Mycat 的 SQL 拦截机制，Mycat 提供了强大的 SQL 注解机制，可以用来影响 SQL 的路由，用户可以灵活扩展。在此方案中，`/*!mycat : schema = test_01 */` 这个注解就表明此 SQL 将在 test_01 这个 Schema (Database)中执行

- 改造 MySQL JDBC 驱动，MySQL JDBC 驱动是开源的项目，在这里实现对 SQL 的拦截改造，比在程序里实现，要更加安全和可靠



Mycat multi-tenant

2.2 每天 2 亿数据的实时查询案例

某移动项目中，每天的账单结算业务数据估计高峰期每天 2 亿，需要能够响应快速查询，查询性能要求控制在 3 秒内，80% 的查询是根据用户手机号来查询当天或者最近几天的交易流水，此外还有供内部运维人员的查询条件，根据交易的某个内部流水号查询，由于并非单纯的主键查询，所以普通的 Key-Value 系统就难以应付，因此首先想到用分布式内存数据库系统，后来知道了 Mycat，于是开始评估测试 Mycat+MySQL 内存表的可能性，经过详细的分析测试对比，发现 MySQL 内存表方式与 InnoDB 的查询性能差异并不大，因为有索引的情况下，单条或少量结果集的查询，所耗费的磁盘 IO 并不大，而内存表的全表锁定问题会导致到数据录入和查询线程之间的竞争，其结果很不确定，可能导致查询的响应时间达到几十秒，另外，2 个亿的数据要全部装入内存，则计算需要 16G 以上内存，要保持 1 个月的数据，则需要差不多 500G 内存，而现网的机器也还没有那么大内存，最终经过详细的对比测试，采用了 InnoDB 表的方式，测试署环境：Mycat 一个+MySQL 一个，测试客户端也在本机，硬件为笔记本工作站：CPU 酷睿 4800 核心数量：四核心，8 线程，内存 16G，硬盘 SSD 混合硬盘。

MySQL 5.6 参数设置如下：

[mysqld]

```
tmp_table_size=0M
max_connections =2100
innodb_buffer_pool_size=4G
innodb_file_per_table=1
innodb_use_sys_malloc =0
innodb_undo_tablespace=64
innodb_open_files=1024
table_open_cache=1024
innodb_autoextend_increment=128
innodb_max_dirty_pages_pct=90
innodb_log_file_size =128M
innodb_log_buffer_size=16M
innodb_log_files_in_group=8
innodb_flush_log_at_trx_commit=2
enforce-gtid-consistency=true
```

上述设置，没有开启 bin-log（只对主从同步有效），innodb_buffer_pool_size 设置的比较大，日志相关的缓存也优化，每个表一个独立表空间（innodb_file_per_table=1），只有操作系统崩溃的时候才可能丢失 1 秒的数据（innodb_flush_log_at_trx_commit=2），这些配置对于非交易型数据是最佳配置。

查询 2 小时内的某个电话号码的交易信息（排序），限制为 20 条

```
select * from opp_call where calldate in (2014020100,2014020101) and phone = ${phone(139-189)}
order by callminutes desc limit 20;
```

finishend:200000 failed:0 qps:8338.87,query time min:0ms,max:941ms,avg:11.99

finishend:200000 failed:0 qps:8338.87,query time min:0ms,max:941ms,avg:11.99

上述 100 个并发随机查询 20 万次，平均响应时间是 12ms，最大<1S

总结：采用 calldate 的时间分片算法，每个分片保留 1 小时的记录，最多保留 31 天的数据，总共 774 个分片，均匀分布到后端 4-10 台物理机上，数据库建立合适的索引并做优化,满足查询响应时延<2S 的实时查询。

2.3 物联网 26 亿数据的案例

此案例由某研究所提供，场景是采集分布于不同点的探头数据并且保存到数据库中，提供实时查询，最终测试并通过了 10000 个网关并行插入采集数据的同时，进行界面查询的验收测试标准。

数据库分表：通过 Mycat 分库，3 台物理机，共 100 个数据库，每个库一张表。

	场景	接入设备	数据库存量 (亿)	吞吐量 (条/s)	查询用户	查询记录数	响应时间(s)
目标	小规模	2500	6.48	250	1	2500	10
	中规模	5000	12.96	500	1	2500	12
	大规模	10000	25.92	1000	1	2500	20
实测	优化前	500	0.12	500	1	720	186
	测试 1	10000	6.5	1500	10	10000+	1.4
	测试 2	10000	10.3	1500	10	10000+	1.1
	测试 3	10000	16.5	1000	20	10000+	1.3
	测试 4	10000	22.5	1000	20	10000+	1.6
	测试 5	10000	26.1	1000	20	10000+	1.4

从测试结果可以看出，通过性能优化后，一万个网关同时插入数据，当数据库存量在 10 亿以内时，吞吐量为 1500 条/秒，10 个用户并发查询 1 万条记录的时间为 1.1s 左右；当数据库存量扩展到 26.1 亿时，吞吐量降为 1000 条/秒，20 个用户并发查询 1 万条记录的时间为 1.4s 左右，完全符合预计的目标。

2.4 大型分布式零售系统案例

此案例为大型分布式零售系统，支持全国 2 万多家门店的使用，系统部署在北京、深圳多个机房，备用和容灾用，每月订单量千万级，最大的表 10 亿以上。该系统中五个子系统用到 mycat。

系统拆分步骤

1. 寻找大表。对某个子系统中所有表做数据量评估，这个可以找这个业务领域有经验的同事，或者有现有数据的可以根据现有数据量做评估，如评估一个表一年的记录条数、磁盘占用量，3 年的、五年的。

用户	表含义	表名	1年记录数(万条)	1年空间占用大小(Gb)	5年总量估计(万条)	5年总量估计(Gb)
			200	0.55	1,300	3.575
			90	0.44	585	2.86
			25	0.12	163	0.78
			200	0.25	1,300	1.625
			200	0.55	1,300	3.575
			200	0.57	1,300	3.705
			75000	153.67	487,500	998.855
			3600	205.19	23,400	1333.74
			500	16.12	3,250	104.78
			3500	7.5	22,750	48.75
			1500	2.45	9,750	15.925

这个步骤是为了找出系统中的大表，根据自己定的单表最大量来确定是否要拆分，如超过 800 万的表都要拆分。

2. 扩大拆分表范围。扩大拆分表指的是有些表虽然量级没达到 800 万，但是它与第一步选出的大表有关联查询，这些表也一起找出来，然后统筹一起定分片算法和拆分策略。

扩大拆分范围时常用全局表、相同拆分策略等方式。具体见后文第三章 Mycat 实施指南中的数据拆分原则。

3. 定分片策略。这个根据业务不同可能差异很大，需要对 mycat 支持的分片算法都了解清楚，同时对业务系统的业务要非常清楚（即这个工作是需要 2 个人来一起完成的，一个懂 mycat 的，一个懂业务的，如果这两个都懂的就更好了）。

我们的数据拆分方式使用

系统拆分按照后文 mycat 实施指南中的数据拆分原则进行，单表的数据量控制在 800 万以内。

针对零售的业务特点，我们的系统中可用的拆分维度有：经营区域（华东、华北、西北、华中等）、订货单位、管理城市、经营城市、店铺、时间范围等。

联合冗余字段的分片使用

在拆分过程中碰到一个场景，无法满足拆分原则，通过引入联合冗余字段，达到了拆分目的，场景如下：

某几个表业务上都与经营区域相关，但是所有经营区域只有 10 多个，按照数据量预估这个表会有 10 亿的量，按照经营区域拆分，单表能达到 1 亿，如果考虑高峰区域和冷门区域问题，这个峰值会更大，可能 2 亿都有可能。但是又没有其他好的拆分维度可以用，后来想到这个表中还有一个日期字段，查询时都可以加上时间区域的限制，

但是如果按照自然月拆分会如何呢？单表也会超过 800 万，最后确定如果联合这两个字段，多大的数据量都能拆开了，弄出了一个联合字段 zone_yyyymm，表示区域+自然月，1 年 12 个月，10 多个区域，能够拆分成 100 多个分片，这下来再大的数据量也能拆分开来了。

第 3 章 生产环境部署

3.1 单节点 mycat 部署

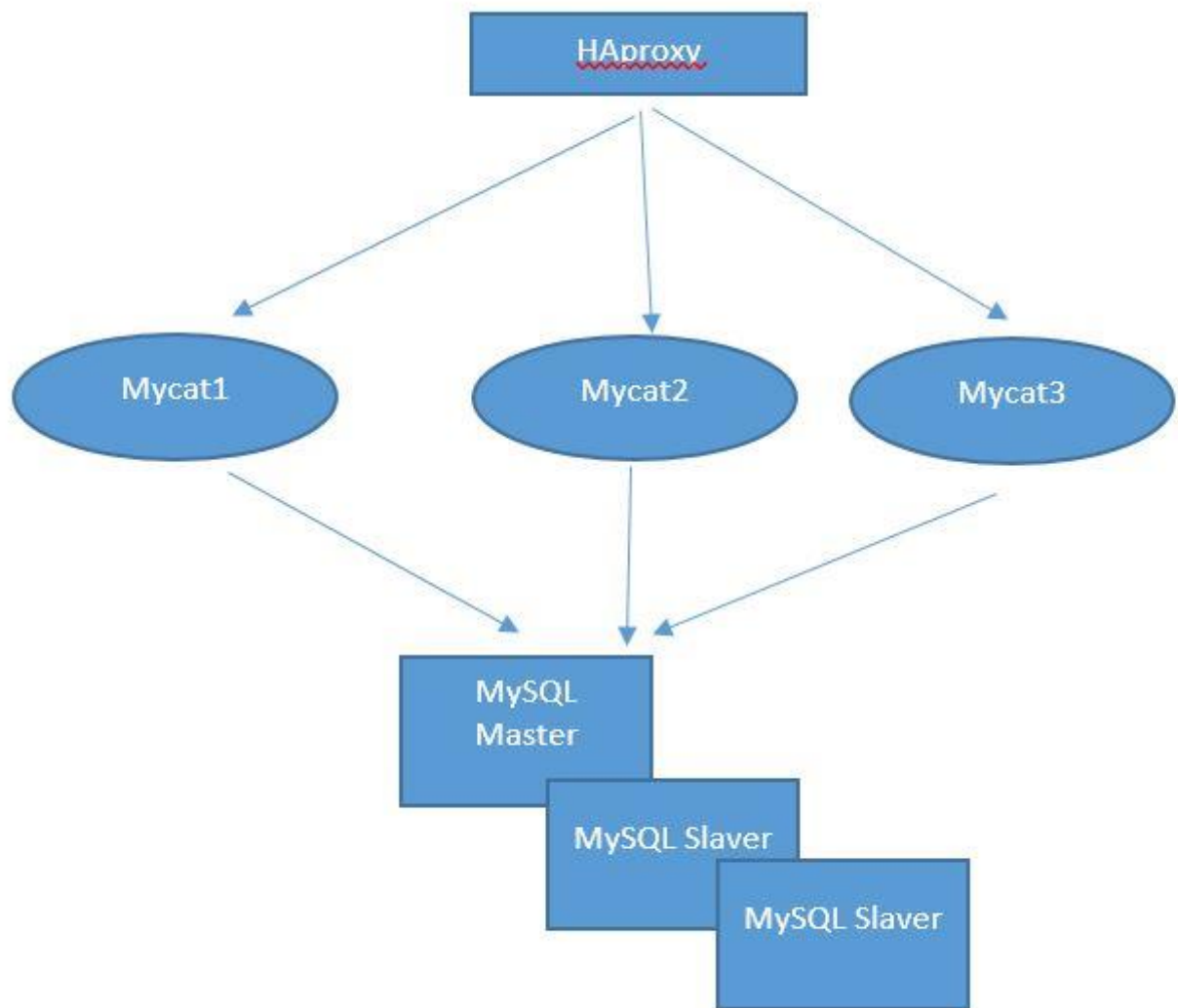
单节点 mycat 的部署指的是只部署一台 mycat 服务器，它与 mycat 集群部署是相对的，如果这台 mycat 服务器宕机了，mycat 就不可用了。

3.2 mycat 的高可用与负载均衡

3.2.1 什么是高可用？

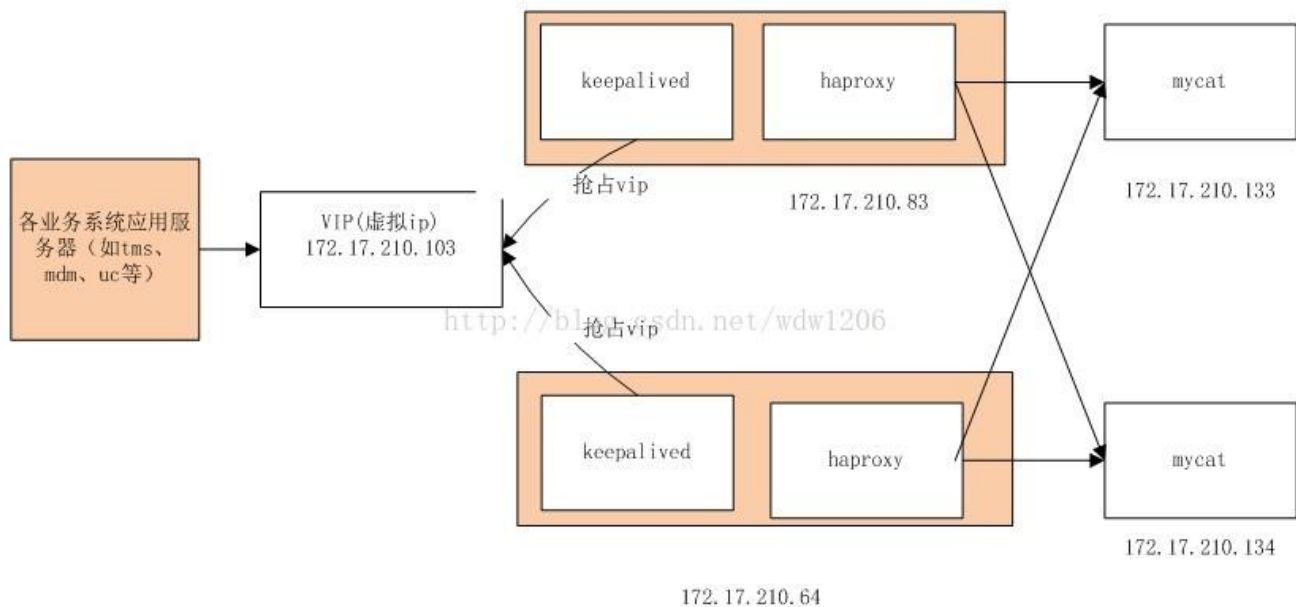
高可用通常也叫 HA (High Available)。指的是，一台服务器宕机了，照样能对外提供服务。常用的高可用软件方案有：LVS、keepalived、Heartbeat、roseHA (roseHA 为收费软件) 等。

Mycat 本身是无状态的，可以用 HAProxy 或四层交换机等设备组成 Mycat 的高可用集群，后端 MySQL 则配置为主从同步，此时整个系统就是高可用的，下图是一个典型的 Mycat 系统高可用的方案：



3.2.2 haproxy + keepalived + mycat 高可用与负载均衡集群配置

部署图:



集群部署图的理解：

- 1、keepalived 和 haproxy 必须装在同一台机器上（如 172.17.210.210.83 机器上，keepalived 和 haproxy 都要安装），keepalived 负责为该服务器抢占 vip（虚拟 ip），抢占到 vip 后，对该主机的访问可以通过原来的 ip（172.17.210.210.83）访问，也可以直接通过 vip（172.17.210.210.103）访问。
- 2、172.17.210.64 上的 keepalived 也会去抢占 vip，抢占 vip 时有优先级，配置 keepalived.conf 中的（priority 150 #数值愈大，优先级越高,172.17.210.64 上改为 120，master 和 slave 上该值配置不同）决定。但是一般哪台主机上的 keepalived 服务先启动就会抢占到 vip，即使是 slave，只要先启动也能抢到。
- 3、haproxy 负责将对 vip 的请求分发到 mycat 上。起到负载均衡的作用，同时 haproxy 也能检测到 mycat 是否存活，haproxy 只会将请求转发到存活的 mycat 上。
- 4、如果一台服务器（keepalived+haproxy 服务器）宕机，另外一台上的 keepalived 会立刻抢占 vip 并接管服务。

如果一台 mycat 服务器宕机，haproxy 转发时不会转发到宕机的 mycat 上，所以 mycat 依然可用。

3.2.3 haproxy 安装

```
useradd haproxy

#wget http://haproxy.1wt.eu/download/1.4/src/haproxy-1.4.25.tar.gz

# tar zxvf haproxy-1.4.25.tar.gz

# cd haproxy-1.4.25
```

```
# make TARGET=linux26 PREFIX=/usr/local/haproxy ARCH=x86_64
```

```
# make install PREFIX=/usr/local/haproxy
```

```
#cd /usr/local/haproxy
```

```
#chown -R haproxy.haproxy *
```

```
haproxy.cfg
```

```
#cd /usr/local/haproxy
```

```
#touch haproxy.cfg
```

```
#vi/usr/local/haproxy/haproxy.cfg
```

```
global
```

```
log 127.0.0.1 local0 ##记日志的功能
```

```
maxconn 4096
```

```
chroot/usr/local/haproxy
```

```
user haproxy
```

```
group haproxy
```

```
daemon
```

```
defaults
```

```
log global
```

```
option dontlognull
```

```
retries 3
```

```
option redispatch
```

```
maxconn 2000
```

```
timeout 5000
```

```
clitimeout 50000
```

```
srvtimeout 50000
```

```
listen admin_status 172.17.210.103:48800 ##VIP
```

```
stats uri/admin-status ##统计页面
```

```
stats auth admin:admin
```

```

mode http

option httplog

listen allmycat_service 172.17.210.103:8096 ##转发到 mycat 的 8066 端口，即 mycat 的服务端口

mode tcp

option tcplog

option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www

balance roundrobin

server mycat_64 172.17.210.64:8066 check port 48700 inter 5s rise 2 fall 3

server mycat_83 172.17.210.83:8066 check port 48700 inter 5s rise 2 fall 3

srvtimeout 20000

listen allmycat_admin 172.17.210.103:8097 ##转发到 mycat 的 9066 端口，及 mycat 的管理控制台端口

mode tcp

option tcplog

option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www

balance roundrobin

server mycat_64 172.17.210.64:9066 check port 48700 inter 5s rise 2 fall 3

server mycat_83 172.17.210.83:9066 check port 48700 inter 5s rise 2 fall 3

srvtimeout 20000

```

haproxy 记录日志

默认 haproxy 是不记录日志的，为了记录日志还需要配置 syslog 模块，在 linux 下是 rsyslogd 服务，先安装 rsyslog

```
yum -y install rsyslog
```

然后

记录 haproxy 日志的配置

```
cd /etc/rsyslog.d/
```

如果没有这个目录，新建

```
cd /etc
```

```
mkdir rsyslog.d
```

```
cd /etc/rsyslog.d/  
touch haproxy.conf
```

```
vi /etc/rsyslog.d/haproxy.conf
```

内容如下：

```
$ModLoad imudp  
$UDPServerRun 514  
  
local0.* /var/log/haproxy.log
```

```
vi /etc/rsyslog.conf
```

1、在##### RULES #####上面一行的地方加入以下内容：

```
# Include all config files in /etc/rsyslog.d/  
$IncludeConfig /etc/rsyslog.d/*.conf  
  
##### RULES #####
```

2、在 local7.* /var/log/boot.log 的下面加入以下内容（增加后的效果如下）：

```
# Save boot messages also to boot.log  
  
local7.* /var/log/boot.log  
  
local0.* /var/log/haproxy.log
```

保存，重启 rsyslog 服务

```
service rsyslog restart
```

现在你就可以看到日志（/var/log/haproxy.log）了

3.2.4 配置监听 mycat 是否存活

在 Mycat server1 Mycat server2 上都需要添加检测端口 48700 的脚本，为此需要用到 xinetd，xinetd 为 linux 系统的基础服务。

首先在 xinetd 目录下面增加脚本与端口的映射配置文件

1、如果 xinetd 没有安装，使用如下命令安装：

```
yum install xinetd -y
```

2、检查/etc/xinetd.conf 的末尾是否有这一句：includedir /etc/xinetd.d

没有就加上

3、检查 /etc/xinetd.d 文件夹是否存在，不存在也加上

```
cd /etc  
mkdir xinetd.d
```

4、增加 /etc/xinetd.d/mycat_status

监听 mycat 是否存活的配置,执行以下命令:

```
cd /etc  
mkdir xinetd.d  
cd /etc/xinetd.d/  
touch mycat_status
```

```
vim /etc/xinetd.d/mycat_status
```

内容如下:

```
service mycat_status  
{  
    flags      = REUSE  
    socket_type = stream  
    port       = 48700  
    wait       = no  
    user       = root  
    server     = /usr/local/bin/mycat_status  
    log_on_failure += USERID  
    disable    = no  
}
```

5、/usr/local/bin/mycat_status 脚本

内容如下:

```
#!/bin/bash  
  
#/usr/local/bin/mycat_status.sh  
  
# This script checks if a mycat server is healthy running on localhost. It will  
  
# return:
```

```
#
# "HTTP/1.x 200 OK\r" (if mycat is running smoothly)
#
# "HTTP/1.x 503 Internal Server Error\r" (else)
mycat=`/usr/local/mycat/bin/mycatstatus |grep'not running'| wc -l`
if [ "$mycat" = "0" ];
then
/bin/echo-e"HTTP/1.1 200 OK\r\n"
else
/bin/echo-e"HTTP/1.1 503 Service Unavailable\r\n"
fi
```

6、/etc/services 中加入 mycat_status 服务

加入 mycat_status 服务，

```
cd /etc
vi services
```

在末尾加入以下内容：

```
mycat_status  48700/tcp      # mycat_status
```

保存

重启 xinetd 服务

```
service xinetd restart
```

7、验证 mycat_status 服务是否启动成功

```
netstat -antup|grep 48700
```

如果成功会现实如下内容：

```
[root@localhost log]# netstat -antup|grep 48700
```

```
tcp 0 0 :::48700 :::* LISTEN 12609/xinetd
```

3.2.5 启动 haproxy

启动 haproxy 前必须先启动 keepalived，否则启动不了。

启动命令：

```
/usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg
```

3.2.6 启动 haproxy 异常情况

如果报以下错误：

```
[root@localhost bin]# /usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg
```

```
[ALERT] 183/115915 (12890) :Starting proxy admin_status: cannot bind socket
```

```
[ALERT] 183/115915 (12890) :Starting proxy allmycat_service: cannot bind socket
```

```
[ALERT] 183/115915 (12890) :Starting proxy allmycat_admin: cannot bind socket
```

原因为：该机器没有抢占到 vip，如果另一台服务启动正常，这个错误可以忽略不管，如果另一台也一样，使用 ping vip 命令看看 vip 是否生效，如果没有生效，说明 keepalived 没有启动成功，回去检查 keepalived 的异常再说。

为了使用方便可以增加一个启动，停止 haproxy 的脚本

```
touch /usr/local/haproxy/sbin/starthaproxy
chmod +x /usr/local/haproxy/sbin/starthaproxy
touch /usr/local/haproxy/sbin/stophaproxy
chmod +x /usr/local/haproxy/sbin/stophaproxy
```

启动脚本 starthap 内容如下：

```
#!/bin/sh
/usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg &
```

停止脚本 stophap 内容如下

```
#!/bin/sh
ps -ef | grep sbin/haproxy | grep -v grep | awk '{print $2}' | xargs kill -s 9
```

启动后可以通过 <http://172.17.210.103:48800/admin-status> (用户名密码都是 admin，haproxy.cfg 中配置的)

3.2.7 openssl 安装

openssl 必须安装，否则安装 keepalived 时无法编译，keepalived 依赖 openssl。

```
tar zxvf openssl-1.0.1g.tar.gz
./config--prefix=/usr/local/openssl
./config-t
make depend
make
make test
make install
ln -s /usr/local/openssl /usr/local/ssl
```

3.2.8 openssl 配置

```
vi /etc/ld.so.conf
```

在/etc/ld.so.conf 文件的最后面，添加如下内容：

```
/usr/local/openssl/lib
```

```
vi /etc/profile
```

内容如下：

```
export OPENSSL=/usr/local/openssl/bin
export PATH=$PATH:$OPENSSL
```

执行以下语句是环境变量生效：

```
source /etc/profile
```

安装 openssl-devel

```
yum install openssl-devel -y #如无法 yum 下载安装，请修改 yum 配置文件
```

测试：

```
ldd /usr/local/openssl/bin/openssl
linux-vdso.so.1 => (0x00007fff996b9000)
libdl.so.2 => /lib64/libdl.so.2 (0x00000030efc00000)
libc.so.6 => /lib64/libc.so.6 (0x00000030f0000000)
/lib64/ld-linux-x86-64.so.2 (0x00000030ef800000)
which openssl
/usr/bin/openssl
```

```
openssl version
```

```
OpenSSL 1.0.0-fips 29 Mar 2010
```

3.2.9 keepalived 安装

本文在 172.17.30.64、172.17.30.83 两台机器进行 keepalived 安装

安装

```
tar zxvf keepalived-1.2.13.tar.gz
```

```
cd keepalived-1.2.13
```

```
./configure--prefix=/usr/local/keepalived
```

```
make
```

```
make install
```

```
cp /usr/local/keepalived/sbin/keepalived /usr/sbin/
```

```
cp /usr/local/keepalived/etc/sysconfig/keepalived /etc/sysconfig/
```

```
cp /usr/local/keepalived/etc/rc.d/init.d/keepalived/etc/init.d/
```

```
mkdir /etc/keepalived
```

```
cd /etc/keepalived/
```

```
cp /usr/local/keepalived/etc/keepalived/keepalived.conf/etc/keepalived
```

```
mkdir-p/usr/local/keepalived/var/log
```

3.2.10 keepalived 配置

建检查 haproxy 是否存活的脚本

```
mkdir /etc/keepalived/scripts
```

```
cd /etc/keepalived/scripts
```

keepalived.conf:

```
vi /etc/keepalived/keepalived.conf
```

Master:

```
! Configuration Fileforkeepalived
```

```
vrp_script chk_http_port {
```

```
script"/etc/keepalived/scripts/check_haproxy.sh"
```

```
interval 2
```

```

weight 2
}

vrrp_instance VI_1 {
    state MASTER          #172.17.210.64 上改为 BACKUP
    interface eth0         #对外提供服务的网络接口

    virtual_router_id 51   #VRRP 组名，两个节点的设置必须一样，以指明各个节点属于同一 VRRP 组
    priority 150           #数值愈大，优先级越高,172.17.210.64 上改为 120
    advert_int 1           #同步通知间隔

    authentication {      #包含验证类型和验证密码。类型主要有 PASS、AH 两种，通常使用的类型为 PASS，据说
AH 使用时有问题
        auth_type PASS
        auth_pass 1111
    }

    track_script {
        chk_http_port      #调用脚本 check_haproxy.sh 检查 haproxy 是否存活
    }

    virtual_ipaddress {    #vip 地址，这个 ip 必须与我们在 lvs 客户端设定的 vip 相一致
        172.17.210.103 dev eth0 scope global
    }

    notify_master /etc/keepalived/scripts/haproxy_master.sh
    notify_backup /etc/keepalived/scripts/haproxy_backup.sh
    notify_fault /etc/keepalived/scripts/haproxy_fault.sh
    notify_stop /etc/keepalived/scripts/haproxy_stop.sh
}

slave:

```

! Configuration File for keepalived

```
vrrp_script chk_http_port {
```

```

script"/etc/keepalived/scripts/check_haproxy.sh"

interval 2

weight 2
}

vrrp_instance VI_1 {

    state BACKUP          #172.17.210.83 上改为 MASTER

    interface eth0        #对外提供服务的网络接口

    virtual_router_id 51   #VRRP 组名，两个节点的设置必须一样，以指明各个节点属于同一 VRRP 组

    priority 120           #数值愈大，优先级越高,172.17.210.83 上改为 150

    advert_int 1          #同步通知间隔

    authentication {      #包含验证类型和验证密码。类型主要有 PASS、AH 两种，通常使用的类型为 PASS，据说
AH 使用时有问题

        auth_type PASS

        auth_pass 1111
    }

    track_script {

        chk_http_port      #调用脚本 check_haproxy.sh 检查 haproxy 是否存活
    }

    virtual_ipaddress {    #vip 地址，这个 ip 必须与我们在 lvs 客户端设定的 vip 相一致

        172.17.210.103 dev eth0 scope global
    }

    notify_master /etc/keepalived/scripts/haproxy_master.sh

    notify_backup /etc/keepalived/scripts/haproxy_backup.sh

    notify_fault /etc/keepalived/scripts/haproxy_fault.sh

    notify_stop /etc/keepalived/scripts/haproxy_stop.sh
}

```

注意：

1. virtual_router_id 51 这个代表一个集群组，如果同一个网段还有另一组集群，请使用不同的组编号区分。

如换成 52、53 等。

2. interface eth1 和 172.17.210.103 dev eth1 scope global 中的 eth1 指的是网卡，如果是多网卡，可能会有 eth0, eth1, eth2..., 可以使用 ifconfig 命令查看，确保 eth0 是本机存在的网卡地址。有些服务器如果只有一个网卡，但被人为把 eth0 改成 eth1 了，你再写 eth0 就找不到了的。

check_haproxy.sh

```
vi /etc/keepalived/scripts/check_haproxy.sh
```

脚本含义：如果没有 haproxy 进程存在，就启动 haproxy，停止 keepalived

check_haproxy.sh

```
#!/bin/bash

STARTHAPROXY="/usr/local/haproxy/sbin/haproxy -f /usr/local/haproxy/haproxy.cfg"

STOPKEEPALIVED="/etc/init.d/keepalived stop"

LOGFILE="/usr/local/keepalived/var/log/keepalived-haproxy-state.log"

echo "[check_haproxy status]" >> $LOGFILE

A=`ps -C haproxy --no-header |wc -l`

echo "[check_haproxy status]" >> $LOGFILE

date >> $LOGFILE

if [ $A -eq 0 ];then

echo $STARTHAPROXY >> $LOGFILE

$STARTHAPROXY >> $LOGFILE 2>&1

sleep 5

fi

if [ `ps -C haproxy --no-header |wc -l` -eq 0 ];then

exit 0

else

exit 1

fi
```

haproxy_master.sh(master 和 slave 一样)

/etc/keepalived/scripts/haproxy_master.sh

```
#!/bin/bash

STARTHAPROXY="/usr/local/haproxy/sbin/haproxy-f/usr/local/haproxy/haproxy.cfg"
STOPHAPROXY="ps-ef |grep sbin/haproxy| grep -v grep|awk '{print $2}'|xargs kill -s 9"
LOGFILE="/usr/local/keepalived/var/log/keepalived-haproxy-state.log"

echo "[master]" >> $LOGFILE

date >> $LOGFILE

echo "Being master...." >> $LOGFILE 2>&1

echo "stop haproxy...." >> $LOGFILE 2>&1

$STOPHAPROXY >> $LOGFILE 2>&1

echo "start haproxy...." >> $LOGFILE 2>&1

$STARTHAPROXY >> $LOGFILE 2>&1

echo "haproxy started ..." >> $LOGFILE
```

haproxy_backup.sh(master 和 slave 一样)

/etc/keepalived/scripts/haproxy_backup.sh

```
#!/bin/bash

STARTHAPROXY="/usr/local/haproxy/sbin/haproxy-f/usr/local/haproxy/haproxy.cfg"
STOPHAPROXY="ps-ef |grep sbin/haproxy| grep -v grep|awk '{print $2}'|xargs kill -s 9"
LOGFILE="/usr/local/keepalived/var/log/keepalived-haproxy-state.log"

echo "[backup]" >> $LOGFILE

date >> $LOGFILE

echo "Being backup...." >> $LOGFILE 2>&1

echo "stop haproxy...." >> $LOGFILE 2>&1

$STOPHAPROXY >> $LOGFILE 2>&1

echo "start haproxy...." >> $LOGFILE 2>&1

$STARTHAPROXY >> $LOGFILE 2>&1

echo "haproxy started ..." >> $LOGFILE
```

haproxy_fault.sh(master 和 slave 一样)

/etc/keepalived/scripts/haproxy_fault.sh

```
#!/bin/bash
```

```
LOGFILE=/usr/local/keepalived/var/log/keepalived-haproxy-state.log
```

```
echo "[fault]" >> $LOGFILE
```

```
date >> $LOGFILE
```

```
haproxy_stop.sh(master 和 slave 一样)
```

```
/etc/keepalived/scripts/haproxy_stop.sh
```

```
#!/bin/bash
```

```
LOGFILE=/usr/local/keepalived/var/log/keepalived-haproxy-state.log
```

```
echo "[stop]" >> $LOGFILE
```

```
date >> $LOGFILE
```

启用服务

```
service keepalived start
```

第 4 章 Mycat 最佳实践



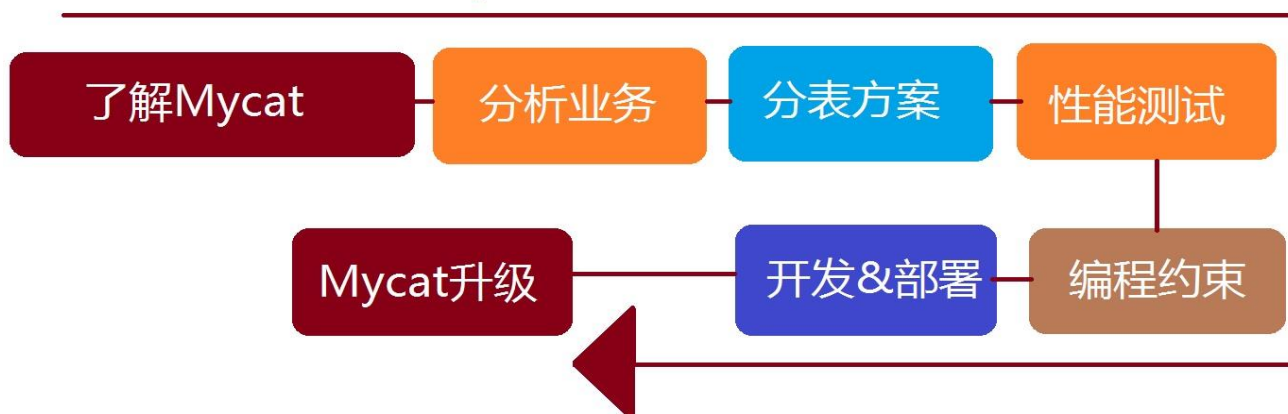
Mycat 如图所述通过后端接入不同的后端解决业务的完整需求。

第 5 章 Mycat 实施指南

5.1 Mycat 项目实施步骤

首先，全面了解 Mycat 的能力、目前的限制、以及可能的解决办法，然后，在此基础上，考虑是否用 Mycat 的分表分片功能，根据目前业务的数据模型和数据访问模式，确定几个可能的分表方案，然后对方案进行针对性的性能测试，在性能数据的基础上，最终决定采用怎样的分片策略。

Mycat实施流程



了解 Mycat 的能力，包括如下的方面：

- Mycat 的起源和解决的目标；
- Mycat 在数据库中间件方面的独特功能和定位；
- Mycat 的实际案例情况；
- Mycat 的优点和不足；
- Mycat 所提供的监控和测试工具；
- Mycat 社区的动态。

其中，关于分片规则的支持和扩展、多数据库支持、SQL 拦截和注解、跨库 Join、读写分离、缓存功能、高可用性等方面需要比较深入的学习和理解，有助于正确的使用 Mycat 来解决当前的业务问题。

接下来是分析当前业务，具体内容包括如下几个方面：

- 数据模型：重点关注数据的增长模式（实时大量增长还是缓慢增长）和规律、数据之间的关联关系；
- 数据访问模式：通过抓取系统中实际执行的 SQL，分析其频率、响应时间、对系统性能和功能的影响程度；
- 数据可靠性的要求：系统中不同数据表的可靠性要求，以及操作模式；

- 事务的要求：系统中哪些业务操作是严格事务的，哪些是普通事务或可以无事务的；
- 数据备份和恢复问题：目前的备份模式，对系统的压力等。

数据的模型和访问模式在很大程度上决定了未来数据分片的模式，包括哪些表用全局表、哪些用 ER 分片、哪些用范围分片规则、哪些用一致性 Hash 或自定义方式。而数据可靠性的要求，则影响到 Mycat 后端是采用普通的 MySQL 主从还是用 Gluster 多写模式，事务性要求需要相关的表或者 SQL 尽量不会垮分片执行，对于以后制定本项目的编程约束有重要意义。

分表方案则需要确定如下一些问题：

- 哪些表要分片、什么分片规则、依赖关联关系如何解决；
- 数据迁移和扩容的手段。

建议根据业务分析的结果，确定两套比较合适分表方案，然后进行性能测试，选出最佳的分表方案，性能测试可以采用 Mycat 自带的超级工具，此工具在前面提到过，可以模拟接近真实业务数据的数据，并随机制造大量的数据供测试，是目前开源的最佳数据库性能测试工具。

在最终进入开发之前，架构师还需要给出一个编程约束，需要明确列出不能执行的 SQL 语句，这些约束可能包括如下几种：

- 跨越太多节点的查询语句；
- 不能 Join 的表和相关的 Join SQL；
- 很影响性能的复杂 SQL；
- 对比较大的表的 SQL 操作提示。

最后在开发阶段，还应该做到如下几点

- 一开始就按照最初的分片设计和数据规模，制造大量的随机数据，进行开发和测试，尽早发现性能问题；
- 对所有的 SQL 进行统计分析，找出异常的 SQL，包括跨越太多分片的 SQL，以及执行缓慢的 SQL，对这些 SQL 进行分析和优化；
- 时刻关注性能问题。

当项目上线后，通过 Mycat Web 对系统进行监控，特别是服务的 IO 和网络指标，除此之外，对 Mycat 运行过程中的日志也要进行排查，告警信息可能是 SQL 错误，可能是 Mycat Bug，及时分析处理，并积极反馈给 Mycat 社区，寻求帮助。

5.2 分表分库原则

分表分库虽然能解决大表对数据库系统的压力，但它并不是万能的，也有一些不利之处，因此首要问题是，分不分库，分哪些库，什么规则分，分多少分片。

原则一：能不分就不分，1000 万以内的表，不建议分片，通过合适的索引，读写分离等方式，可以很好的解决性能问题。

原则二：分片数量尽量少，分片尽量均匀分布在多个 DataHost 上，因为一个查询 SQL 跨分片越多，则总体性能越差，虽然要好于所有数据在一个分片的结果，只在必要的时候进行扩容，增加分片数量。

原则三：分片规则需要慎重选择，分片规则的选择，需要考虑数据的增长模式，数据的访问模式，分片关联性问题，以及分片扩容问题，最近的分片策略为范围分片，枚举分片，一致性 Hash 分片，这几种分片都有利于扩容。

原则四：尽量不要在一个事务中的 SQL 跨越多个分片，分布式事务一直是个不好处理的问题。

原则五：查询条件尽量优化，尽量避免 Select * 的方式，大量数据结果集下，会消耗大量带宽和 CPU 资源，查询尽量避免返回大量结果集，并且尽量为频繁使用的查询语句建立索引。

这里特别强调一下分片规则的选择问题，如果某个表的数据有明显的时间特征，比如订单、交易记录等，则他们通常比较适合用时间范围分片，因为具有时效性的数据，我们往往关注其近期的数据，查询条件中往往带有时间字段进行过滤，比较好的方案是，当前活跃的数据，采用跨度比较短的时间段进行分片，而历史性的数据，则采用比较长的跨度存储。

总体上来说，分片的选择是取决于最频繁的查询 SQL 的条件，因为不带任何 Where 语句的查询 SQL，会便利所有的分片，性能相对最差，因此这种 SQL 越多，对系统的影响越大，所以我们要尽量避免这种 SQL 的产生。

如何准确统计和分析当前系统中最频繁的 SQL 呢？有几个简单做法：

- 采用特殊的 JDBC 驱动程序，拦截所有业务 SQL，并写程序进行分析
- 采用 Mycat 的 SQL 拦截器机制，写一个插件，拦截所欲 SQL，并进行统计分析
- 打开 MySQL 日志，分析统计所有 SQL

找出每个表最频繁的 SQL，分析其查询条件，以及相互的关系，并结合 ER 图，就能比较准确的选择每个表的分片策略。

对于大家经常提起的同库内分表的问题，这里做一些分析和说明，同库内分表，仅仅是单纯的解决了单一表数据过大的问题，由于没有把表的数据分布到不同的机器上，因此对于减轻 MySQL 服务器的压力来说，并没有太大的作用，大家还是竞争同一个物理机上的 IO、CPU、网络。此外，库内分表的时候，要修改用户程序发出的

SQL，可以想象一下 A、B 两个表各自分片 5 个分表情况下的 Join SQL 会有多么的反人类。这种复杂的 SQL 对于 DBA 调优来说，也是个很大的问题。因此，Mycat 和一些主流的数据库中间件，都不支持库内分表，但由于 MySQL 本身对此有解决方案，所以可以与 Mycat 的分库结合，做到最佳效果，下面是 MySQL 的分表方案：

- MySQL 分区；
- MERGE 表（MERGE 存储引擎）。

通俗地讲 MySQL 分区是将一大表，根据条件分割成若干个小表。mysql5.1 开始支持数据表分区了。如：某用户表的记录超过了 600 万条，那么就可以根据入库日期将表分区，也可以根据所在地将表分区。当然也可根据其其他的条件分区。

- RANGE 分区：基于属于一个给定连续区间的列值，把多行分配给分区，MySQL 分区支持的分区规则有以下几种：LIST 分区：类似于按 RANGE 分区，区别在于 LIST 分区是基于列值匹配一个离散值集合中的某个值来进行选择。

- HASH 分区：基于用户定义的表达式的返回值来进行选择的分区，该表达式使用将要插入到表中的这些行的列值进行计算。这个函数可以包含 MySQL 中有效的、产生非负整数值的任何表达式。

- KEY 分区：类似于按 HASH 分区，区别在于 KEY 分区只支持计算一列或多列，且 MySQL 服务器提供其自身的哈希函数。必须有一列或多列包含整数值。

在 Mysql 数据库中，Merge 表有点类似于视图，mysql 的 merge 引擎类型允许你把许多结构相同的表合并为一个表。之后，你可以执行查询，从多个表返回的结果就像从一个表返回的结果一样。每一个合并的表必须有完全相同表的定义和结构，但只支持只是支持 MyISAM 引擎。

- Mysql Merge 表的优点：
- 分离静态的和动态的数据；
- 利用结构接近的数据来优化查询；
- 查询时可以访问更少的数据；
- 更容易维护大数据集。

在数据量、查询量较大的情况下，不要试图使用 Merge 表来达到类似于 Oracle 的表分区的功能，会很影响性能。我的感觉是和 union 几乎等价。

Mycat 建议的方案是 Mycat 分库+MySQL 分区，此方案具有以下优势：

- 充分结合分布式的并行能力和 MySQL 分区表的优化；

- 可以灵活的控制表的数据规模;
- 可以两个维度对表进行分片, MyCAT 一个维度分库, MySQL 一个维度分区。

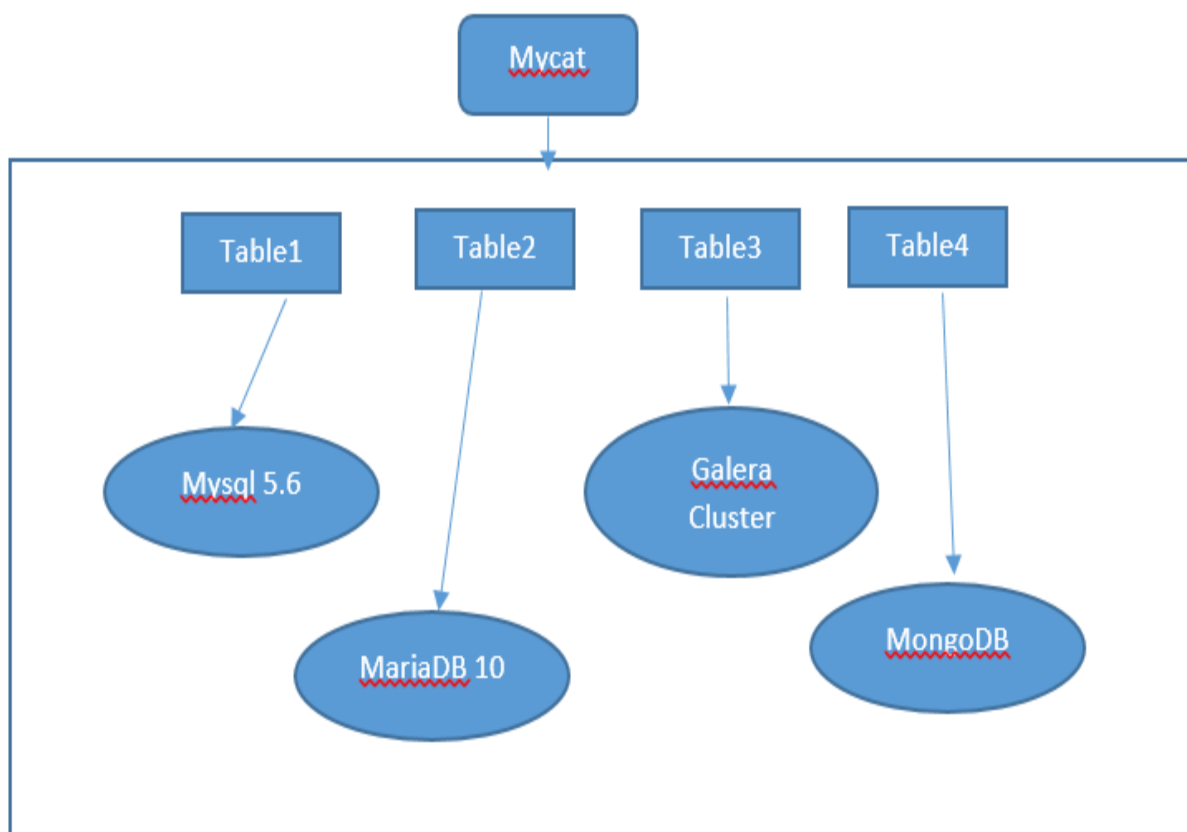
5.3 后端存储的选择

Mysql 尽量用比较新的稳定版, 当前来说 5.6 和 5.7 都是比较靠谱的一个选择, 因为 Mysql 这两个版本做了大量优化。另外 Mysql 的各种变种版本都可以考虑。以下是一些通用准则:

对于非严格苛刻交易型的数据表, 建议用 MariaDB, 这个版本目前在开源界很盛行, 评价很高, percona 版本也值得推荐, percona 有很多辅助的运维工具。

- 对于交易型的数据表, 可以考虑 Mysql 官方稳定版, 若交易型的数据表要求可靠性非常高, 比如是替代 Oracle, 也可以选择 Galera Cluster 这种高可用的方案, 他以一定的写入性能损失带来了数据的高可用和高并发访问。
- 根据数据的可靠性要求, 可以采用各种数据同步方案, 比如 1 主多从, 读写分离提升数据表的读的并发能力。
- 部分表可以用 NoSQL 方式存储, 而前端访问方式不变, Mycat 支持后端 MongoDB 和很多 NoSQL 系统, 以提升查询能力
- 部分表可以采用 MySQL 内存表, 来提升查询和写入速度, 替代部分复杂缓存方案。

下面是一个可能的 Mycat 部署方案, 不同的表用不同的存储方式, 让不同的表根据其访问模式, 都达到最佳状态。



5.4 数据拆分原则

1. 达到一定数量级才拆分（800 万）
2. 不到 800 万但跟大表（超 800 万的表）有关联查询的表也要拆分，在此称为大表关联表
3. 大表关联表如何拆：小于 100 万的使用全局表；大于 100 万小于 800 万跟大表使用同样的拆分策略；无法跟大表使用相同规则的，可以考虑从 java 代码上分步骤查询，不用关联查询，或者破例使用全局表。
4. 破例的全局表：如 item_sku 表 250 万，跟大表关联了，又无法跟大表使用相同拆分策略，也做成了全局表。破例的全局表必须满足的条件：没有太激烈的并发 update，如多线程同时 update 同一条 id=1 的记录。虽有多线程 update，但不是操作同一行记录的不在此列。多线程 update 全局表的同一行记录会死锁。批量 insert 没问题。
5. 拆分字段是不可修改的
6. 拆分字段只能是一个字段，如果想按照两个字段拆分，必须新建一个冗余字段，冗余字段的值使用两个字段的值拼接而成（如大区+年月拼成 zone_yyyymm 字段）。

7. 拆分算法的选择和合理性评判：按照选定的算法拆分后每个库中单表不得超过 800 万
8. 能不拆的就尽量不拆。如果某个表不跟其他表关联查询，数据量又少，直接不拆分，使用单库即可。

5.5 DataNode 的分布问题

DataNode 代表 MySQL 数据库上的一个 Database，因此一个分片表的 DataNode 的分布可能有以下几种：

- 都在一个 DataHost 上
- 在几个 DataHost 上，但有连续性，比如 dn1 到 dn5 在 Server1 上，dn6 到 dn10 在 Server2 上，依次类推
- 在几个 DataHost 上，但均匀分布，比如 dn1,dn2,d3 分别在 Server1,Server2,Server3 上，dn4 到 dn5 又重复如此

一般情况下，不建议第一种，二对于范围分片来说，在大多数情况下，最后一种情况最理想，因为当一个表的数据均匀分布在几个物理机上的时候，跨分片查询或者随机查询，都是到不同的机器上去执行，并行度最高，IO 竞争也最小，因此性能最好。

当 we 有几十个表都分片的情况下，怎样设计 DataNode 的分布问题，就成了一个难题，解决此难题的最好方式是试运行一段时间，统计观察每个 DataNode 上的 SQL 执行情况，看是否有严重不均匀的现象产生，然后根据统计结果，重新映射 DataNode 到 DataHost 的关系。

Mycat 1.4 增加了 distribute 函数，可以用于 Table 的 dataNode 属性上，表示将这些 dataNode 在该 Table 的分片规则里的引用顺序重新安排，使得他们能均匀分布到几个 DataHost 上：

```
<table name="oc_call" primaryKey="ID" dataNode="distribute(dn1$0-372,dn2$0-372)" rule="latest-month-calldate" />
```

其中 dn1xxx 与 dn2xxxx 是分别定义在 DataHost1 上与 DataHost2 上的 377 个分片。

5.6 Mycat 目前存在的限制

部分 SQL 还不能很好的支持

- 除了分片规则相同、ER 分片、全局表、以及 SharedJoin，其他表之间的 Join 问题目前还没有很好的解决，需要自己编写 Catlet 来处理。
- 不支持 Insert into 中不包括字段名的 SQ. L

- insert into x select from y 的 SQL，若 x 与 y 不是相同的分片规则，则不被支持，此时会涉及到跨分片转移。
- 跨分片的事务，目前只是弱 XA 模式，还没完全实现 XA 模式。
- 分片的 Table，目前不能执行 Lock Table 这样的语句，因为这种语句会随机发到某个节点，也不会全部分片锁定，经常导致死锁问题，此类问题常常出现在 sqldump 导入导出 SQL 数据的过程中。
- 目前 sql 解析器采用 Druid,再某些 sql 例如 order, group, sum, count 条件下，如果这类操作会出现兼容问题，比如：

```
select t.name as name1 from test order by t.name
```

- 这条语句 select 列的别名与 order by 不一致解析器会出现异常，所以在对列加别名时候要注意这类操作异常，特别是由 jpa 等类似的框架生成的语句会有兼容问题。

开发框架方面，虽然支持 Hibernate，但不建议使用 Hibernate，而是建议 Mybatis 以及直接 JDBC 操作，原因 Hibernate 无法控制 SQL 的生成，无法做到对查询 SQL 的优化，导致大数量下的性能问题。此外，事务方面，建议自己手动控制，查询语句尽量走自动提交事务模式，这样 Mycat 的读写分离会被用到，提升性能很明显。

第 6 章 数据迁移与扩容实践

6.1 离线扩容缩容

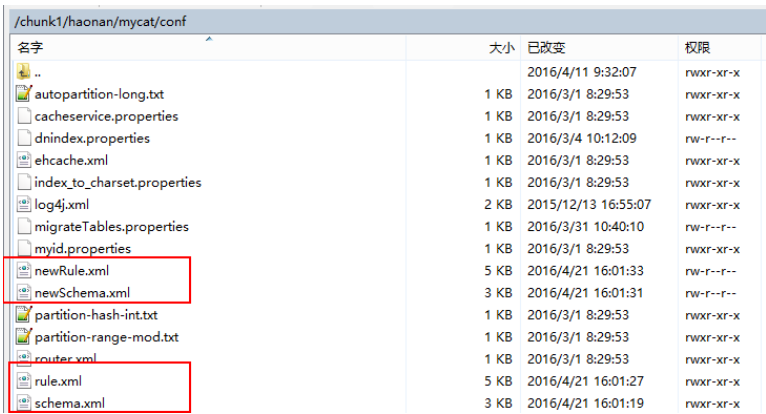
工具目前从 mycat1.6 开始支持。

一、准备工作

- 1、mycat 所在环境安装 mysql 客户端程序。
- 2、mycat 的 lib 目录下添加 mysql 的 jdbc 驱动包。
- 3、对扩容缩容的表所有节点数据进行备份，以便迁移失败后的数据恢复。

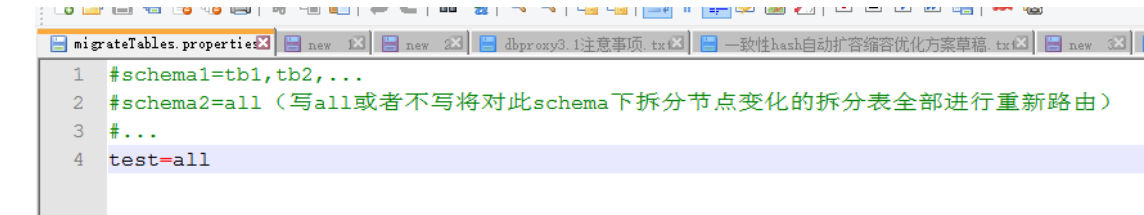
二、扩容缩容步骤

- 1、复制 schema.xml、rule.xml 并重命名为 newSchema.xml、newRule.xml 放于 conf 目录下。



名字	大小	已改变	权限
..		2016/4/11 9:32:07	rwXr-Xr-X
autopartition-long.txt	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
cacheservice.properties	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
dnindex.properties	1 KB	2016/3/4 10:12:09	rw-r--r--
ehcache.xml	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
index_to_charset.properties	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
log4j.xml	2 KB	2015/12/13 16:55:07	rwXr-Xr-X
migrateTables.properties	1 KB	2016/3/31 10:40:10	rw-r--r--
myid.properties	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
newRule.xml	5 KB	2016/4/21 16:01:33	rw-r--r--
newSchema.xml	3 KB	2016/4/21 16:01:31	rw-r--r--
partition-hash-int.txt	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
partition-range-mod.txt	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
router.xml	1 KB	2016/3/1 8:29:53	rwXr-Xr-X
rule.xml	5 KB	2016/4/21 16:01:27	rwXr-Xr-X
schema.xml	3 KB	2016/4/21 16:01:19	rwXr-Xr-X

- 2、修改 newSchema.xml 和 newRule.xml 配置文件为扩容缩容后的 mycat 配置参数（表的节点数、数据源、路由规则）。
- 3、修改 conf 目录下的 migrateTables.properties 配置文件，告诉工具哪些表需要进行扩容或缩容,没有出现在此配置文件的 schema 表不会进行数据迁移，格式：



```
#schema1=tb1,tb2,...
#schema2=all (写all或者不写将对此schema下拆分节点变化的拆分表全部进行重新路由)
#...
test=all
```

- 4、修改 bin 目录下的 dataMigrate.sh 脚本文件，参数如下：

tempFileDir 临时文件路径,目录不存在将自动创建

isAlwaysUseMaster 默认 true:不论是否发生主备切换,都使用主数据源数据, false: 使用当前数据源

mysqlBin: mysql bin 路径

cmdLength mysqldump 命令行长度限制 默认 110k 110*1024。在 LINUX 操作系统有限制单条命令行的长度是 128KB,也就是 131072 字节,这个值可能不同操作系统不同内核都不一样,如果执行迁移时报 Cannot run program "sh": error=7, Argument list too long 说明这个值设置大了,需要调小此值。

charset 导入导出数据所用字符集 默认 utf8

deleteTempFileDir 完成扩容缩容后是否删除临时文件 默认为 true

threadCount 并行线程数(涉及生成中间文件和导入导出数据)默认为迁移程序所在主机环境的 cpu 核数*2

delThreadCount 每个数据库主机上清理冗余数据的并发线程数,默认为当前脚本程序所在主机 cpu 核数/2

queryPageSize 读取迁移节点全部数据时一次加载的数据量 默认 10w 条

5、停止 mycat 服务(如果可以确保扩容缩容过程中不会有写操作,也可以不停止 mycat 服务)。

6、通过 crt 等工具进入 mycat 根目录,执行 bin/ dataMigrate.sh 脚本,开始扩容/缩容过程:

```
[iddbs@kf-app2 mycat]$ bin/dataMigrate.sh
/usr/java/jdk1.7.0_67/bin/java -DMYCAT_HOME="/chunk1/haonan/mycat" -classpath "/chunk1/haonan/mycat/
ator-client-2.9.0.jar:/chunk1/haonan/mycat/lib/curator-framework-2.9.0.jar:/chunk1/haonan/mycat/lib/ehcache-core-2.6.11.jar:/chunk1/haonan/mycat/lib/fastjson-1.2.7.jar:/chunk1/haonan/mycat/lib/json-20151123.jar:/chunk1/haonan/mycat/lib/leveldb-0.7.jar:/chunk1/haonan/mycat/lib/mapdb-1.0.7.jar:/chunk1/haonan/mycat/lib/mongo-java-driver-2.11.4.jar:/chunk1/haonan/mycat/lib/mysql-connector-java-5.1.15-bin.jar:/chunk1/haonan/mycat/lib/netty-3.7.0.Final.jar:/chunk1/haonan/mycat/lib/slf4j-api-1.7.12.jar:/chunk1/haonan/mycat/lib/slf4j-log4j12-1.7.12.jar:/chunk1/haonan/mycat/lib/stax2-api-1.5.4.jar:/chunk1/haonan/mycat/lib/wrapper.jar:/chunk1/haonan/mycat/lib/xml-api-1.0.2.jar:/chunk1/haonan/mycat/lib/xml-apis-1.0.2.jar" -XX:MaxPermSize=64M -XX:+AggressiveOpts -XX:MaxDirectMemorySize=2G org.opencldb.util.dat
ageSize=100000 -isAlwaysUseMaster=true -cmdLength=110*1024 -charset=utf8 -mysqlBin=/home/idd
2016-04-21 16:04:37:269 [1]-> creating migrator schedule and temp files for migrate...
+-----[test:WORKERS_INFO] migrate info-----+
|tableSize      = 1000000|
|migrate before = [dn1, dn2, dn3, dn4]|
|migrate after  = [dn1, dn2, dn3, dn4, dn5, dn6, dn7, dn8]|
|rule function  = PartitionByMod|
+-----+

+----[test:WORKERS_INFO] migrate schedule----+
|dn1[250000] -> [0, 0, 0, 0, 125000, 0, 0, 0]|
|dn2[250000] -> [0, 0, 0, 0, 0, 125000, 0, 0]|
|dn3[250000] -> [0, 0, 0, 0, 0, 0, 125000, 0]|
|dn4[250000] -> [0, 0, 0, 0, 0, 0, 0, 125000]|
+-----+

2016-04-21 16:04:39:111 [2]-> start migrate data...
[test:WORKERS_INFO] dn1->dn5 completed in 6221ms
[test:WORKERS_INFO] dn2->dn6 completed in 6532ms
[test:WORKERS_INFO] dn3->dn7 completed in 6532ms
[test:WORKERS_INFO] dn4->dn8 completed in 6605ms

2016-04-21 16:04:45:919 [3]-> cleaning redundant data...
[test:WORKERS_INFO] clean dataNode dn3 complete in 2850ms
[test:WORKERS_INFO] clean dataNode dn1 complete in 2857ms
[test:WORKERS_INFO] clean dataNode dn4 complete in 2858ms
[test:WORKERS_INFO] clean dataNode dn2 complete in 2867ms

2016-04-21 16:04:48:924 [4]-> validating tables migrate result...
+-----migrate result-----+
|[test:WORKERS_INFO] -> success|
+-----+
```

表迁移信息及迁移计划

执行数据迁移

清理旧节点上的冗余数据

校验迁移是否成功

7、扩容缩容成功后，将 newSchema.xml 和 newRule.xml 重命名为 schema.xml 和 rule.xml 并替换掉原文件，重启 mycat 服务，整个扩容缩容过程完成。

三、注意事项：

- 1) 保证拆分表迁移数据前后路由规则一致。
- 2) 保证拆分表迁移数据前后拆分字段一致。
- 3) 全局表将被忽略。
- 4) 不要将非拆分表配置到 migrateTables.properties 文件中。
- 5) 暂时只支持拆分表使用 mysql 作为数据源的扩容缩容。

四、优化

dataMigrate.sh 脚本中影响数据迁移速度的有 4 个参数，正式迁移数据前可以先进行一次测试，通过调整以下参数进行优化获得一个最快的参数组合。

threadCount 脚本执行所在主机的并行线程数（涉及生成中间文件和导入导出数据）默认为迁移程序所在主机环境的 cpu 核数*2

delThreadCount 每个数据库主机上清理冗余数据的并发线程数，默认为当前脚本程序所在主机 cpu 核数/2，同一主机上并发删除数据操作线程数过多可能会导致性能严重下降，可以逐步提高并发数，获取执行最快的线程个数。

queryPageSize 读取迁移节点全部数据时一次加载的数据量 默认 10w 条

cmdLength mysqldump 命令行长度限制 默认 110k 110*1024。尽量让这个值跟操作系统命令长度最大值一致，可以通过以下过程确定操作系统命令行最大长度限制：

逐步减少 100000，直到不再报错

```
/bin/sh -c "/bin/true $(seq 1 100000)"
```

获取不报错的值，通过 `wc -c` 统计字节数，结果即操作系统命令行最大长度限制（可能稍微小一些）

```

[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 100000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 90000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 80000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 70000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 60000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 50000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 40000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 30000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 20000)"
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 29000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 28000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 27000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 26000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 25000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 24000)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 23000)"
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 23900)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 23800)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 23700)"
-bash: /bin/true: Argument list too long
[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 23600)"
[iddbs@kf-app2 ~]$

```

```

[iddbs@kf-app2 ~]$ /bin/true "$(seq 1 23696)"
[iddbs@kf-app2 ~]$ seq 1 23696|wc -c
131070
[iddbs@kf-app2 ~]$

```

6.2 案例一：使用一致性 Hash 进行分片

当使用一致性 Hash 进行路由分片时，假设存在节点宕机/新增节点这种情况，那么相对于使用其他分片算法（如 mod），就能够尽可能小的改变已存在 key 映射关系，尽可能的减少数据迁移操作。当然一致性 hash 也有一个明显的不足，假设当前存在三个节点 A,B,C，且是使用一致性 hash 进行分片，如果你想对当前的 B 节点进行扩容，扩容后节点为 A,B,C,D，那么扩容完成后数据分布就会变得不均匀。A,C 节点的数据量是大于 B,D 节点的。

据测试，分布最均匀的是 mod，一致性哈希只是大致均匀。数据迁移也是，迁移量最小的做法是 mod，每次扩容后节点数都是 2 的 N 次方，这样的迁移量最小。但是 mod 需要对每个节点都进行迁移，这也是 mod 的不足之处。总之，还得酌情使用，根据业务选择最适合自己的方案。

6.2.1 配置使用

rule.xml: 定义分片规则

```
<tableRule name="sharding-by-murmur">
<rule>
<columns>SERIAL_NUMBER</columns>
<algorithm>murmur</algorithm>
</rule>
</tableRule>

<function name="murmur" class="io.mycat.route.function.PartitionByMurmurHash">
<property name="seed">0</property>
<property name="count">2</property>
<property name="virtualBucketTimes">160</property>
<!-- <property name="weightMapFile">weightMapFile</property>
<property name="bucketMapPath">/home/usr/mycat/bucketMapPath</property> -->
</function>
```

tableRule 定义分片规则

- name: 分片规则的名字。在 schema.xml 文件中调用。
- columns: 根据数据库中此字段进行分片。
- algorithm: 值是分片算法定义处的 name 属性。比如: murmur。

function 定义一致性 Hash 的参数。

- seed: 计算一致性哈希的对象使用的数值, 默认是 0。
- count: 待分片的数据库节点数量, 必须指定, 否则没法分片。
- virtualBucketTimes: 虚拟节点。默认是 160 倍, 也就是虚拟节点数是物理节点数的 160 倍。指定

virtualBucketTimes 可以使一致性 hash 分片更加均匀。

- bucketMapPath: 用于测试时观察各物理节点与虚拟节点的分布情况, 如果指定了这个属性, 会把虚拟节点的 murmur hash 值与物理节点的映射按行输出到这个文件, 没有默认值, 如果不指定, 就不会输出任何东西。必须是绝对路径, 且可读写。

schema.xml: 定义逻辑库, 表、分片节点等内容

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE mycat:schema SYSTEM "schema.dtd">
<mycat:schema xmlns:mycat="http://io.mycat/">
<schema name="mycat" checkSQLschema="false" sqlMaxLimit="100">
<table name="T_CMS_ORDER" primaryKey="ORDER_ID" dataNode="dn202_3316" rule="sharding-by-murmur" />
</schema>
<dataNode name="dn202_3316" dataHost="lh202_1" database="poc" />
<dataHost name="lh202_1" maxCon="2000" minCon="10" balance="0" writeType="0" dbType="mysql" dbDriver="native">
<heartbeat>select user()</heartbeat>
<writeHost host="master_host-m1" url="10.21.17.202:3316" user="usr" password="pwd"></writeHost>
<writeHost host="slave_host-m1" url="10.21.17.201:3317" user="usr" password="pwd"></writeHost>
</dataHost>
</mycat:schema>
```

server.xml: 定义用户以及系统相关变量, 如端口等。没有太高要求的可以只修改数据库部分。

```
<user name="mycat">
<property name="password">usr</property>
<property name="schemas">pwd</property>
</user>
```

经过以上配置就可以使用一致性 hash 了。

6.2.2 一致性 Hash 的数据迁移

开始迁移

进行一致性 hash 进行迁移的时候, 假设你新增加一个节点, 需要修改以下两个配置文件:

rule.xml

```
<function name="murmur" class="io.mycat.route.function.PartitionByMurmurHash">
<property name="seed">0</property>
```

```

<property name="count">3</property>

<property name="virtualBucketTimes">160</property>

<!-- <property name="weightMapFile">weightMapFile</property>

<property name="bucketMapPath">/home/usr/mycat/bucketMapPath</property> -->

</function>

```

需要把节点的数量从 2 个节点扩为 3 个节点。

schema.xml

```

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE mycat:schema SYSTEM "schema.dtd">

<mycat:schema xmlns:mycat="http://io.mycat/">

<schema name="mycat" checkSQLschema="false" sqlMaxLimit="100">

<table name="T_CMS_ORDER" primaryKey="ORDER_ID" dataNode="dn202_3316,dn201_3316"
rule="sharding-by-murmur" />

</schema>

<dataNode name="dn202_3316" dataHost="lh202_1" database="poc" />
<dataNode name="dn201_3316" dataHost="lh201_1" database="poc" />

<dataHost name="lh202_1" maxCon="2000" minCon="10" balance="0" writeType="0" dbType="mysql"
dbDriver="native">

<heartbeat>select user()</heartbeat>

<writeHost host="master_host-m1" url="10.21.17.202:3316" user="usr" password="pwd"></writeHost>
<writeHost host="savle_host-m1" url="10.21.17.201:3317" user="usr" password="pwd"></writeHost>

</dataHost>

<dataHost name="lh201_1" maxCon="2000" minCon="10" balance="0" writeType="0" dbType="mysql"
dbDriver="native">

<heartbeat>select user()</heartbeat>

<writeHost host="master_host-m1" url="10.21.17.201:3316" user="usr" password="pwd"></writeHost>

```

```
<writeHost host="savle_host-m1" url="10.21.17.202:3317" user="usr" password="pwd"> </writeHost>
</dataHost>
</mycat:schema>
```

需要添加新节点的 `dataNode` 和 `dataHost` 信息，以及在 `schema` 中的 `table` 标签下把新增节点的 `dataNode` 的 `name` 增加到 `dataNode` 的值中。

6.2.3 开始迁移

使用 `io.mycat.util.rehasher.RehashLauncher` 类进行数据迁移。参数以命令行的形式进行载入。如

```
-jdbcDriver=xxxxx -jdbcUrl=.... -host=192.168.1.1:3316 -user=xxxx -password=xxxx -database=xxxx
```

- `jdbcDriver`: 数据库驱动。如 `com.mysql.jdbc.Driver`。
- `jdbcUrl`: 连接数据库的 url，不同数据库不一样。如

`jdbc:mysql://10.21.17.201:3316/mycat?rewriteBatchedStatements=true`。

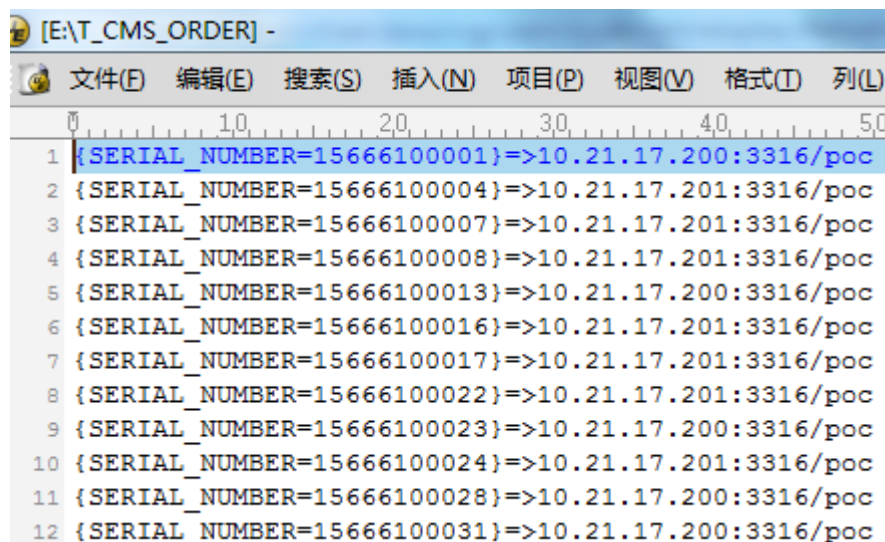
- `host`: 包括主机名和端口，形如 `ip:port`。如 `10.21.100.86:3316`。
- `user`: 连接数据库的用户名。如 `usr`。
- `database`: 数据库的名字。如 `mycat`。
- `password`: 连接数据库的密码。如 `pwd`。
- `tablesFile`: 记录数据表的文件，一个表一行。
- `shardingField`: 数据库中进行分片的字段。
- `rehashHostsFile`: 这个参数没有用到，按照当时的要求，这个类一次只处理一个节点，所以不需要配置
- `hashType`: 是 MURMUR hash 还是 mod hash。
- `seed`: 生成一致性 hash 对象的参数。默认为 0。
- `virtualBucketTimes`: 虚拟节点的倍数。默认为 160。
- `weightMapFile`: 节点的权重，没有指定权重的节点默认是 1。以 `properties` 文件的格式填写，以从 0 开

始到 `count-1` 的整数值也就是节点索引为 `key`，以节点权重值为值。所有权重值必须是正整数，否则以 1 代替。

- `rehashNodeDir`: 一个 linux 目录，这个程序执行完了，把计算结果输出到这个目录，一个表一个文件存在这个目录里，文件名是表名。

如果你觉得使用命令行的方式去读取配置不是那么方便，你也可以自己定义读取配置文件的算法，只要能保证 `io.mycat.util.rehasher.RehashLauncher` 这个类能够读到所有的配置就可以了。比如使用 `properties` 文件保存配置文件(每次修改配置文件后都需要重新编译)，本着怎么方便怎么写代码的原则，就是这么任性。

运行 `io.mycat.util.rehasher.RehashLauncher` 后生成的文件格式如下：



```
[E:\T_CMS_ORDER] -
文件(F) 编辑(E) 搜索(S) 插入(N) 项目(P) 视图(V) 格式(O) 列(L)
1 {SERIAL_NUMBER=15666100001}>=>10.21.17.200:3316/poc
2 {SERIAL_NUMBER=15666100004}>=>10.21.17.201:3316/poc
3 {SERIAL_NUMBER=15666100007}>=>10.21.17.201:3316/poc
4 {SERIAL_NUMBER=15666100008}>=>10.21.17.201:3316/poc
5 {SERIAL_NUMBER=15666100013}>=>10.21.17.200:3316/poc
6 {SERIAL_NUMBER=15666100016}>=>10.21.17.201:3316/poc
7 {SERIAL_NUMBER=15666100017}>=>10.21.17.201:3316/poc
8 {SERIAL_NUMBER=15666100022}>=>10.21.17.201:3316/poc
9 {SERIAL_NUMBER=15666100023}>=>10.21.17.200:3316/poc
10 {SERIAL_NUMBER=15666100024}>=>10.21.17.201:3316/poc
11 {SERIAL_NUMBER=15666100028}>=>10.21.17.200:3316/poc
12 {SERIAL_NUMBER=15666100031}>=>10.21.17.200:3316/poc
```

为了方便进行迁移，我们可以对代码进行适当的修改，如



```
RehashLauncher.java
String hostWithDatabase = args.getHostWithDatabase();
PrintStream ps = null;
StringBuffer rehashData = new StringBuffer();
try {
    ps = new PrintStream(output);
    while (!CollectionUtil.isEmpty(list = JdbcUtils
        .executeQuery(dataSource,
            "select " + args.getShardingField()
                + " from " + table
                + " limit ?,?", page++
                * pageSize, pageSize))) {
        for (int i = 0, l = list.size(); i < l; i++) {
            Object sf = list.get(i);
            Integer hash = alg.calculate(sf.toString());
            String host = rehashHosts[hash];
            if (host.equals(hostWithDatabase)) {
                rehashData.append(sf.toString().replaceAll(
                    "\\D", "")
                    + ",");
            }
        }
        // 组装输出信息
        ps.print("("
            + rehashData.toString().substring(0,
                rehashData.toString().length() - 1)
            + ")");
    }
} catch (Exception e) {
```



```

condOrder=$n

done

echo "***** 导出 *****"

date

# 1) 首先调用 mysqldump 进行数据导出
echo "开始导出主机:$ 表:T_CMS_ORDER."

mysqldump -h$rehashNode -P3316 -upoc -ppoc123 poc T_CMS_ORDER --default-character-set=utf8 --
extended-insert=false --no-create-info --add-locks=false --complete-insert --where=" SERIAL_NUMBER in
$condOrder " > ./T_CMS_ORDER_temp.sql

echo "导出结束."

echo " "

echo "***** 导入 *****"

date

# 2) 调用 mycat 接口进行数据导入
echo "开始导入 T_CMS_ORDER 表数据"

mysql -h$expanNode -P8066 -upoc -ppoc123 poc --default-character-set=utf8 < ./T_CMS_ORDER_temp.sql

echo "导入结束."

echo " "

echo "***** 删除数据 *****"

date

# 3) 当前两步都无误的情况下,删除最初的导出数据.

echo "开始删除已导出的数据表:."

mysql -h$rehashNode -P3316 -upoc -ppoc123 -e "use poc; DELETE FROM T_CMS_ORDER WHERE
SERIAL_NUMBER in $condOrder ; commit; "

echo "删除结束."

```

```

echo " "

echo "***** 清空临时文件 *****"

date

# 4) 清空临时文件

rm ./t_cms_order_temp.sql

echo "清空临时文件"

echo "#####主机:$rehashNode 处理完成#####"

date

echo " "

echo "ReHash 运行完毕."

```

假设文件名是：ReHashRouter.sh

1. 授权：chmod +x ReHashRouter.sh
2. 运行：./ReHashRouter.sh 10.21.17.200 10.21.17.201 /home/mycat/T_CMS_ORDER

6.3 案例二：使用范围分片

在使用范围分片算法进行路由分片时，配置非常简单。如下：

6.3.1 配置使用

rule.xml：定义分片规则

```

<tableRule name="auto-sharding-long">
<rule>
<columns>user_id</columns>
<algorithm>rang-long</algorithm>
</rule>
</tableRule>

<function name="rang-long" class="io.mycat.route.function.AutoPartitionByLong">

```

```
<property name="mapFile">autopartition-long.txt</property>
```

```
</function>
```

tableRule 定义分片规则

- name: 分片规则的名字。在 schema.xml 文件中调用。
- columns: 根据数据库中此字段进行分片。
- algorithm: 值是分片算法定义处的 name 属性。比如: rang-long。

function 定义范围分片的参数

可以看到根据范围自动分片的配置文件非常简单, 只有一个 mapFile(要赋予读的权限), 此 mapFile 文件定义了每个节点中 user_id 的范围, 如果 user_id 的值超过了这个范围, 那么则使用默认节点。当前版本代码中默认节点的值是-1, 表示不配置默认节点, 超过当前范围就会报错。当然你也可以在 property 中增加 defaultNode 的默认值, 如:

```
<property name="defaultNode">0</property>
```

mapFile 节点配置文件

当前版本提供了一个 mapFile 配置文件供大家参考和使用, 如下

```
# range start-end ,data node index  
  
# K=1000,M=10000.  
  
0-500M=0  
  
500M-1000M=1
```

所有的节点配置都是从 0 开始, 及 0 代表节点 1, 此配置非常简单, 即预先制定可能的 id 范围到某个分片。

(tips:K 和 M 的定义是在 io.mycat.route.function.NumberParseUtil 中定义的, 如果感兴趣的同学可以自己定义其他字母。)

扩容

如果业务需要或者数据超过当前定义的范围, 需要新增节点, 则可以在文件中追加 1000M-1500M=2 即可。当然新增的节点需要在 schema.xml 中进行定义。

```
# range start-end ,data node index  
  
# K=1000,M=10000.  
  
0-500M=0
```


500M-1000M=1

1000M-1500M=2

6.4 数据迁移的注意点

6.4.1 迁移时间的确定

在进行迁移之前，我们得先确定迁移操作发生的时间。停机操作需要尽可能的让用户感知不到，你可以观察每段时间系统的吞吐量，以此作为依据。一般来说，我们选择在凌晨进行升级操作。

6.4.2 数据迁移前的测试

需要做一些相关的性能测试，在条件允许的情况下在类似的环境中完全模拟，得到一些性能数据，然后不断的改进，看能够否有大的提升。

我们在做数据迁移的时候，就是在备份库中克隆的一套环境，然后在上面做的性能测试，在生产上的步骤方式都一样，之后在正式升级的时候就能够做到心中有数。什么时候需要注意什么，什么时候需要做哪些相关的检查。

6.4.3 数据备份

热备甚至冷备，在数据迁移之前进行完整的备份，一定要是全量的。甚至在允许的情况下做冷备都可以。数据的备份越充分，出现问题时就有了可靠的保证。

lob 数据类型的备份，做表级的备份 (create table nologging....)，对于 lob 的数据类型，在使用 imp,impdp 的过程中，瓶颈都在 lob 数据类型上了，哪怕表里的 lob 数据类型是空的，还是影响很大。自己在做测试的时候，使用 Imp 基本是一秒钟一千条的数据速度，impdp 速度有所提升，但是 parallle 没有起作用，速度大概是 1 秒钟 1 万条的样子。

如果在数据的导入过程中出了问题，如果有完整快速的备份，自己也有了一定的数据保证，要知道出问题之后再从备份库中导入导出，基本上都是很耗费时间的。

6.4.4 数据升级前的系统级检查

1. 内存检查。可以使用 top,free -m 来做一个检查，看内存的使用情况是否正常，是否有足够的内存空间。
2. 检查 cpu,io 情况。查看 iowait 是否稳定，保持在较低的一个幅度。
3. 检查进程的情况。检查是否有高 cpu 消耗的异常进程，检查是否有僵尸进程，排查后可以杀掉。

4. 是否有 crontab 的设置。如果在升级的时候有什么例行的 job 在运行，会有很大的影响，可以使用 `crontab -l` 来查看 crontab 的情况。
5. vxfs 下的 odm 是否已经启用。如果使用的 veritas 的文件系统，需要检查一下 odm 是否正常启用。
6. IO 简单测试。从系统角度来考虑，需要保证 io 的高效性。可以使用 `iostat,sar` 等来评估。
7. 网络带宽。数据迁移的时候肯定会从别的服务器中传输大量的文件,dump 等，如果网络太慢，无形中就是潜在的问题。可以使用 `scp` 来进行一个简单的测试。

6.4.5 异常情况

网络临时中断。网络的问题需要格外重视，可能在运行一些关键的脚本时，网络突然中断，那对于升级就是灾难，所以在准备脚本的时候，需要考虑到这些场景，保留完整的日志记录。

可以使用 `nohup` 来做外后台运行某些关键的脚本。这样网络断了以后，还有一线希望。在数据迁移，数据升级的时候，一定要保留完整的日志记录，这样如果稍候有问题，也可以及时查验，也可以避免很多不必要的纷争。如果有争议，可以找出日志来，一目了然。

当然，这样会有大量的日志产生，一定需要保证归档空间足够大，及时的转移归档文件。排除归档爆了以后数据的问题，使用 `sqlloader,impdp` 等数据迁移策略的时候，如果归档出了问题，是很头疼的问题。

6.5 load data 批量导入

`load data infile` 语句可以从一个文本文件中以很高的速度读入一个表中。性能大概是 `insert` 语句的几十倍。通常用来批量数据导入。目前只支持 `mysql` 数据库且 `dbDriver` 必须为 `native`。`Mycat` 支持 `load data` 自动路由到对应的分片。`Load data` 和压缩协议 `mycat` 从 1.4 开始支持。

6.5.1 语法和注意事项

标准示例：

```
load DATA local INFILE 'd:\88\qq.txt' IGNORE INTO TABLE test CHARACTER SET 'gbk' FIELDS
TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"' LINES TERMINATED BY '\n'(id,sid,asf) ;
```

注意：如果数据中可能包含一些特殊字符，比如分割符转义符等，建议用引号扩起来，通过 `OPTIONALLY ENCLOSED BY '"'` 指定。如果这样还不行，可以把字段值中的引号替换成 `\`。

如果指定 local 关键词，则表明从客户端主机读文件。如果 local 没指定，文件必须位于 mycat 所在的服务器上。

可以通过 fields terminated by 指定字符之间的分割符号，默认值为\t

通过 lines terminated by 可以指定行之间的换行符。默认为\n,这里注意有些 windows 上的文本文件的换行符可能为\r\n，由于是不可见字符，所以请小心检查。

character set 指定文件的编码，**建议跟 mysql 的编码一致**，否则可能乱码。其中字符集编码必须用引号扩起来，否则会解析出错。

还可以通过 replace | ignore 指定遇到重复记录是替换还是忽略。

目前列名必须指定，且必须包括分片字段，否则没办法确定路由。

其他参数参考 mysql 的 load data infile 官方文档说明。

注意其他参数的先后顺序不能乱，比如列名比较在最后的，顺序参考官方说明。

标准 load data 语句：

LOAD DATA 语句,同样被记录到 binlog,不过是内部的机制。

官方说明：<http://dev.mysql.com/doc/refman/5.7/en/insert-speed.html>

You can use the following methods to speed up inserts:

- If you are inserting many rows from the same client at the same time, use INSERT statements with multipleVALUES lists to insert several rows at a time. This is considerably faster (many times faster in some cases) than using separate single-row INSERT statements. If you are adding data to a nonempty table, you can tune thebulk_insert_buffer_size variable to make data insertion even faster. See Section 5.1.4, “Server System Variables” .
- When loading a table from a text file, use LOAD DATA INFILE. This is usually 20 times faster than using INSERTstatements. See Section 13.2.6, “LOAD DATA INFILE Syntax” .
- Take advantage of the fact that columns have default values. Insert values explicitly only when the value to be inserted differs from the default. This reduces the parsing that MySQL must do and improves the insert speed.
- See Section 8.5.4, “Bulk Data Loading for InnoDB Tables” for tips specific to InnoDB tables.
- See Section 8.6.2, “Bulk Data Loading for MyISAM Tables” for tips specific to MyISAM tables.

例子：

导出：

```
select * from tblog_article into outfile '/test.txt' FIELDS TERMINATED BY '\t' OPTIONALLY ENCLOSED BY
" ESCAPED BY '\\' LINES TERMINATED BY '\n';
```

导入：

```
load data local infile '/var/lib/mysql/blog/test.txt' INTO TABLE tblog_article FIELDS TERMINATED BY '\t'
OPTIONALLY ENCLOSED BY " ESCAPED BY '\\' LINES TERMINATED BY '\n'
```

```
(id,title,level,create_time,create_user,create_user,article_type_id,article_content,istop,status,read_count );
```

6.5.2 客户端配置

如果是 mysql 命令行连接的 mycat，则需要加上参数`-local-infile=1`。Jdbc 则无需设置。

Load data 测试性能

在一台 win8 下，jvm 1.7 参数默认，jdbc 连接 mycat。

处理器:	Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz 3.20 GHz
安装内存(RAM):	8.00 GB (7.83 GB 可用)
系统类型:	64 位操作系统，基于 x64 的处理器

测试结果 load data local 导入 1 百万数据到 5 个分片耗时 10 秒，1 千万数据到 5 个分片耗时 145 秒。

6.6 使用 mysqldump 进行数据迁移

mysqldump 是 mysql 自带的命令行工具。可以用它完成全库迁移（从一个 mysql 库完整迁移到 mycat），也可以迁移某几个表，还可以迁移某个表的部分数据。

6.6.1 全库迁移

- 迁移前准备

迁移前确保 mysql 库和 mycat 库中的表名一样（mycat 库中只需要有表名配置在 schema.xml 文件中即可）

- 从 mysql 导出

从 mysql 库上全库导出

```
mysqldump -c --skip-add-locks databaseName> /root/databaseName.sql
```

注意：（上面的语句没有`-uroot -ppassword` 参数，是因为 mysql 服务器设置了本机免密码等。

如果设置了密码:通过以下命名导出(用户名为 root, 密码为 123456):

```
mysqldump -uroot -p123456 -c --skip-add-locks databaseName> /root/databaseName.sql  
)
```

说明:两个参数不可少, 如下:

-c 参数不可少, -c, 全称为-complete-insert 表示使用完整的 insert 语句(用列名字)。

--skip-add-locks 表示导出数据时不加锁, 如果加锁涉及多分片时容易导致死锁。

- **导入到 mycat**

将 databaseName.sql 拷贝到 mycat 集群中的一台 mysql 服务器上/root 目录下。

- **连接 mycat:**

```
mysql -uusername -ppassword -h172.17.xxx.xxx -P8066
```

切换到指定的数据库:

```
use databaseName;
```

导入脚本:

```
source /root/databaseName.sql;
```

6.6.2 迁移一个库中的某几个表

只是导出命令不同, 其他与全库迁移一样

```
mysqldump -c --skip-add-locks databaseName table1 table2> /root/someTables.sql
```

6.7 迁移一个表中的部分数据

迁移一个表中的部分数据, 加参数-where 实现。

命令如下:

```
mysqldump -c --skip-add-locks databaseName tableName --where=" id > 900 " >
```

```
/root/onetableDataWithCondition.sql
```

6.8 数据自动迁移方案设计

mycat 数据自动迁移功能设计方案, 欢迎大家提意见和改进方案, **重点是步骤 6.切换路由**

总体思路:通过 dump 对应 slot 的全量数据, 增量数据通过 mysql 的 binlog 进行复制

优点：迁移过程不中断不影响现有业务，迁移支持数据一致性，迁移速度快

限制：需要在 mysql 机器上部署迁移节点，binlog 格式必须为 row 格式，表结构需要增加一个 slot 字段

6.8.1 分片规则设计

$\text{crc32}(\text{key}) \% 102400 = \text{slot}$

slot 按照范围均匀分布在 dataNode 上,针对每张表进行实例化,通过一个文件记录 slot 和节点映射关系,迁移过程中通过 zk 协调

其中需要在分片表中增加 slot 字段,用以避免迁移时重新计算,只需要迁移对应 slot 数据即可

分片最大个数为 102400 个,短期内应该够用,每分片一千万,总共可以支持一万亿数据

6.8.2 迁移步骤

1.任一 mycat 节点收到迁移指令,计算执行计划,划分出子任务,所有任务存入 zk。一个子任务为一个源节点中若干 slot 至多个目标节点

2.校验是否所有 mycat 节点存活,涉及的后端节点是否存活,mycat 辅助节点是否存活,都满足进入下一步,不满足给出错误提示。

3.从源节点 dump 全量数据

在 mysql 机器上部署 mycatNode,通过 mysqldump 导出指定 slot 的数据,mysqldumprt 可以导出数据同时指定 binlog 位置。

导出数据为 loaddata 格式的纯数据加表结构,而不是通常的 insert 脚本

dump 数据通过 loaddata 命令执行到对应新节点上

4.全量数据校验

对 dump 全量数据进行校验,校验成功后删除 dump 在硬盘上的数据文件

校验思路一种通过主键判断数据是否在对应节点,另外一种判断全部字段进行校验。

5.增量数据 binlog

通过解析 binlog 获取增量数据,将增量数据发送到新节点执行,执行完进行异步校验,校验结果存 zk

6.切换路由

为了保证数据一致性,方案有 2 个,1 是短时间拒绝服务,2 是短时间排队等待,只要不超时对业务无影响

切换触发条件 1: 迁移量占比例如超过比如 90% 或者 2: binlog 空闲 10 秒

两个思路:

1.开始切换---》mycat 路由计算出若是对应迁移数据则拒绝可写执行,读可以执行,返回错误----》binlog 追满且满足大于 mycat 最大事务时间 5 分钟--》停 binlog 和清理旧节点数据-----》切换 mycat 路由---》恢复正常对外服务

2.开始切换---》mycat 路由计算出若是对应迁移数据则通过排队或者锁等方式阻塞等待----》binlog 追满且满足大于 mycat 最大事务时间 5 分钟-----》停 binlog 和清理旧节点数据-----》切换 mycat 路由-----》原等待执行唤醒路由新节点执行--》恢复正常对外服务

问题：

1. 两个方案考虑到切换时当前正在执行的操作 都需要等待 mycat 最大事务时间默认 5 分钟，这个是否有更好思路去掉

2. 另外触发条件 binlog 空闲 10 秒是否合理，最后判断 binlog 追满怎么判断比较好，binlog 空闲 30 秒？

a.查询 binlog 位置来判断

b.通过遍历执行会话判断是否有事务在执行。

7.迁移中异常处理

迁移中步骤均记录与 zk，出现异常，人工介入处理，可以提供清理新节点数据命令(暂时还没提供此命令)

6.9 数据自动迁移使用指南

6.9.1 约束条件

扩容动态迁移需要满足如下条件：

1. 会使用 mysql,mycat,zk
2. 使用 crc32slot 分片算法，表中会加_slot 隐藏字段 int 类型
3. mysqldump 需要添加到环境变量的 path 可以直接调用
4. mysql 的 binlog 格式需要是 row,mysql 开启 binlog
5. 如果是 mariadb，需要将 mysql-binlog 的 jar 替换下，官方默认不支持，可以到
6. <https://github.com/magicdoom/mysql-binlog-connector-java> 下载编译替换
7. 下载最新版本的 1.6.5 版本 mycat
8. mycat 使用 zk 来管理集群

6.9.2 准备

rule.xml 配置

rule.xml 的 `<mycat:rule xmlns:mycat="http://io.mycat/">`根节点下必须存在以下节点

```

<tableRule name="crc32slot">
  <rule>
    <columns>id</columns>
    <algorithm>crc32slot</algorithm>
  </rule>
</tableRule>

<function name="crc32slot"
  class="io.mycat.route.function.PartitionByCRC32PreSlot">

</function>

```

注意的是,确保 crc32slot 节点不带有 count 子节点

在 myid.properties 中配置集群相关信息

```

loadZk=true
clusterId=mycat 集群唯一 ID
myid=集群内本实例 ID
clusterSize=2    集群大小
clusterNodes=mycat_fz_01,mycat_fz_02
# server booster ;    booster install on db same server,will reset all minCon to 2
type=server
boosterDataHosts=localhost1, localhost2

```

确保 loadZk=true 并开启 zookeeper,zookeeper 中有 mycat 配置

type 有两个选项,server 与 booster.其中 server 是普通的 mycat 节点, booster 节点负责全量导出,不做业务,只用于迁移。booster 节点是与 mysql 节点部署在同一台机器的 mycat 节点,一旦设置 booster, Schema.xml 中 dataHost 标签中的 minCon 属性会被覆盖设置为 2,minCon 属性是指定每个读写实例连接池的最小连接,初始化连接池的大小.与之相关的一个设置是 dataHost 的 maxCon 数量,这个数量留给用户限制最大连接数.为什么要把连接数最小值覆盖为 2?原因是 booster 节点可以平时就可以部署运行,但是它本身并不能占用过多连接导致影响 mysql 节点运行业务,但是考虑到一旦运行迁移任务又会占用很多连接,所以 maxCon 需要设置一个合理的值以减少对 mysql 节点的影响,一个可以约束的范围是一个 mysql 节点的对应的所有 dataHost 的 maxCon 加起来不超过这个 mysql 的最大连接数.另外,与减少 mysql 影响的设置还有 dataHost 的 slavelDs 的设置.

boosterDataHosts 代表本 mycat booster 节点负责哪些 dataHost 的迁移工作,支持多个 DataHost,以英文逗号,分隔, 迁移任务所需的 dataHost 需要在此写清楚,并且与 schema.xml(即 zookeeper 中的 schema)中的

dataNode 节点中的 DataHost 属性对应.如果迁移用到 dataHost 不在 boosterDataHosts 里,涉及的

dataHost 将不会运行迁移任务.

schema.xml 配置

table 节点配置

```
<table name="travelrecord" dataNode="xxxx" rule="crc32slot" />
```

要求所分片的表必须使用 `crc32slot` 算法分片

datahost 节点的 slaveIDs 属性配置

需要配置 slaveIDs 属性,格式为

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="0" slaveIDs="1,2,3"
  writeType="0" dbType="mysql" dbDriver="native" switchType="1"
  slaveThreshold="100" >
```

其中 slaveIDs 属性的值可以为

1

1,2,3(以逗号分隔)

0-2,3,4(等价于 0,1,2,3,4,以 - 分隔表示范围,-两边的值必须是 java 整型)

整个 slaveIDs 的范围必须大于等于 2

slaveIDs 的意义在同一个时刻,在一个使用 zk 管理的 mycat 集群中仅能进行对一个 schema 和 table 的组合进行迁移,这一次迁移会使用一个或者多个分别不同的 datahost,使用的每一个 datahost 会在 zk 中标记占用它的 slaveIDs 中的一个 slaveID.如果迁移需要用到的 datahost 的 slaveIDs 里面的 slaveID 被其他迁移任务占用完毕,则这次迁移构造失败.只要迁移任务没有正常完成,slaveID 是不会取消占用的.slaveID 将在迁移正常完成后在 zk 中标记取消占用.slaveID 对应一个 BinaryLogClient,一个 datahost 设置多个 slaveID 能起到类似令牌桶算法的流量控制的作用,slave 即为令牌,减少迁移操作对 datahost 的影响.

配置新的 dataNode

注意需要先将新的 dataNode 在配置中都加好,并通过 zk 刷新到集群里每个 mycat 里

使用 zk 来管理 mycat 集群

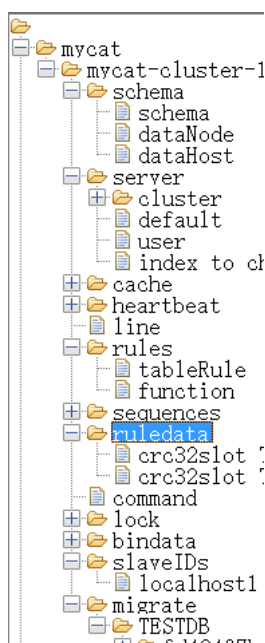
在 zkconf 下配置文件配置好, 然后通过 init_zk_data.sh 脚本初始化到 zk, 之后部署 mycatbooster 节点到每台 mysql 机器上

开发环境下可以运行

src\main\java\io\mycat\config\loader\zkprocess\xmltozk\XmltoZkMain.java 把 zkconf 下配置文件载入 zk.

查看 zk 里面的节点可以使用 ZoolInspector 工具查看,

zk 的节点与本地的 xml 配置文件具有以下映射关系



Mycat-cluster-x 的名字对应 myid.properties 文件里面的 clusterId,

节点内容是 myid.propertie 的内容

schema/schema 对应 schema 与 table 节点

schema/dataNode 有 dataNode 节点

schema/dataHost 有 dataHost 节点信息

slaveIDs 对应 schema.xml 的 dataHost 节点的 slavesIDs

ruledata 对应本地文件中的\conf\ruledata 里面的文件

rules/tableRule 有 rule.xml 的 tableRule 信息(在此处用不上)

rules/function 有 rule.xml 的 function 信息(在此处用不上)

...

数据库表的要求

要求：需要在分片表中增加_slot 类型为 int

基于 mycat 新建表，尽量不要擅自越过 mycat 建立表

如果是在 mycat 中的 schema 配置了逻辑表,实际上数据库中没有这个表,此时使用 mycat 创建这个表也可以自动设置上_slot 字段即输入

```
CREATE TABLE `travelrecord` (  
    id xxxxxx  
) ENGINE=INNODB DEFAULT CHARSET=utf8;
```

mycat 会自动生成

```
CREATE TABLE `travelrecord` (  
    id xxxxx  
    _slot int COMMENT '自动迁移算法 slot, 禁止修改'  
) ENGINE = InnoDB CHARSET = utf8
```

同样地 insert 语句也是自动修改为带有_slot 的 sql 语句

如果迁移时候所需的表没有_slot 字段,迁移任务是会忽略这个表的数据迁移并迁移成功

6.9.3 执行迁移命令

在连上 mycat 的 SQL 终端可以执行下面的命令

```
migrate -table=test -add=dn2,dn3,dn4
```

```
migrate -table=schema.test -add=dn2,dn3,dn4
```

test 是要迁移的表名 add 代表迁移到哪几个新的 dataNode 节点,请确保新节点上的库上的对应的表是空的

一个迁移命令只能写一个表名,支持 schema.table 语法,如果不指定 schema,则使用当前连接中的 schema.

也可以使用 use schema;命令指定 schema.其中 migrate table add 不区分大小写,test(表名,最后会转成大写),schema 区分大小写 add 代表迁移到哪几个新的 dataNode 节点,以逗号分隔,节点名区分大小写,如果对应节点没有对应的表,这个表将自动创建

如果想迁移多个表,请多次执行次命令。

如果迁移的表的数据的 slot 不在新的分片范围内,就不会产生迁移任务,出现异常提示

以下是可选参数：

-timeout = 120

120 代表 120 分钟，如果不写这个参数，则默认 120 分钟即两小时。

一次迁移可能涉及多个迁移任务，这个超时指的是正在执行 mysqldump 阶段的所有迁移任务的超时时间，如果超过了这个时间，迁移任务将直接判断完成的任务数量是否等于总的任务数量，如果都完成了，则进行下一个阶段的任务，否则不执行任何操作，之后可能需要人工参与处理 zk 的信息和数据库的数据。

-force=true

数据扩容迁移依赖 zookeeper。有时候可能会出现 mycat 启动，但是 myid.properties 中的 loadZk=false，此时 mycat 实际上读取配置是从本地 xml 文件读取的，但是 mycat 本身是连接上 zookeeper（一般来说，用户不能察觉 mycat 本身连接上了 zookeeper，但是 zookeeper 启动了，mycat 后启动，mycat 就有可能连接上），**此时亦可使用 -force 来执行数据迁移，此时数据迁移的配置使用的是本地的 xml 配置项。**

此时如果有其他 mycat 已经连接上了这个 zookeeper，并且 mycat 启动的时候是 loadZk=true 的，那么他们能监听到这个数据迁移，并能禁止重复的数据迁移命令。（正常情况）

其他情况，处理比较复杂，建议按照使用步骤来使用 mycat。

-force=false 是不存在的，写了这个这个参数忽略

-charset=UTF-8

此参数为 BinlogStream 时候根据传输的 byte[]生成字符串所用的编码,需要与数据库的表的编码一致,如果不设置这个属性,默认为 JDK 默认编码即 UTF-8

6.9.4 查看任务进度和结果

迁移进度可以通过 zk 上的 mycat/mycat-cluster-1/migrate 下查看

status 为 5 代表迁移全部成功

zk 集群任务路径:

```

\mycat
  \mycat-cluster-1
  \slaveIDs
    \1      ----->占用的 slaveID 的整数值
  \migrate
  \taskId(uuid)  {"add":"dn5","schema":"TESTDB","sql":"migrate -table=crctable -
add=dn5","status":5,"table":"crctable","exception":"" 运行时的部分异常信息
                  ,"backupFile":"xxxx" 迁移任务运行前的配置信息备份路径
                  } 任务详情

  \_clean
  \mycat_fz_01
  \_prepare
  \mycat_fz_01
  \_commit
  \mycat_fz_01
  \localhost1  [

{"fclass":"io.mycat.route.function.PartitionByCRC32PreSlot","from":"dn1","schema":"TE
STDB","size":8534,"slaveId":0,"slots":[{"end":34133,"size":8534,"start":25600}],
table":"crctable","to":"dn5"},

{"fclass":"io.mycat.route.function.PartitionByCRC32PreSlot","from":"dn2","schema":"TE
STDB","size":8534,"slaveId":0,"slots":[{"end":85333,"size":8534,"start":76800}],
table":"crctable","to":"dn5"},

{"fclass":"io.mycat.route.function.PartitionByCRC32PreSlot","from":"dn3","schema":"TE
STDB","size":2,"slaveId":0,"slots":[{"end":1,"size":2,"start":0}],table":"crctab
le","to":"dn5"},

{"fclass":"io.mycat.route.function.PartitionByCRC32PreSlot","from":"dn3","schema":"TE
STDB","size":8532,"slaveId":0,"slots":[{"end":8533,"size":8532,"start":2}],table
":"crctable","to":"dn6"},

{"fclass":"io.mycat.route.function.PartitionByCRC32PreSlot","from":"dn4","schema":"TE
STDB","size":8534,"slaveId":0,"slots":[{"end":59733,"size":8534,"start":51200}],
table":"crctable","to":"dn6"}]
\dn2-dn5 {"binlogFile":"mysql-
bin.000157","msg":"sucess","pos":10590010,"status":3}
\dn3-dn5 {"binlogFile":"mysql-

```

```
bin.000157", "msg": "sucess", "pos": 10590010, "status": 3}
\dn1-dn5 {"binlogFile": "mysql-
bin.000157", "msg": "sucess", "pos": 10590010, "status": 3}
\dn4-dn6 {"binlogFile": "mysql-
bin.000157", "msg": "sucess", "pos": 10590010, "status": 3}
\dn3-dn6 {"binlogFile": "mysql-
bin.000157", "msg": "sucess", "pos": 10590010, "status": 3}
```

taskNode 状态迁移流程

taskNode 具有三种状态分别对应 mycat 代码中实现的三种监听器

```
_prepare -> SwitchPrepareListener
_commit --> SwitchCommitListener
_clean --> SwitchCleanListener
```

\localhost1 为 taskNode 记录该 dataHost 所有的迁移任务.

首先状态为 taskNode 的 statue 为 0; taskNode.status = 1 时候 1.MigrateDumpRunner 全量下载需要迁移的 from 的 dataNode 下的对应区间的数据,

并且直接导入到 to 的 dataNode 的数据。同时 zk 的 \dn2-dn5 中写入 {"binlogFile": "mysql-bin.000157", "msg": "sucess", "pos": 10590010, "status": 1}(taskStatus) 2.BinlogStream 并且开始增量复制, 并且开启 BinlogIdleCheck 的定时器检查。 BinlogIdleCheck 检查,全量复制与增量复制都完成之后.(30 秒内没有增量数据了)认定增量完成, 将所有的 taskStatus 设置 status 为 3 如果, \taskId\localhost 下所有的迁移任务都(全部增量复制完成并且 30 秒内没有没有增量数据或者完成 90%) 则 向_prepare 写入当前的 datahost

SwitchPrepareListener 监听_prepare, _prepare\dataHost 的数量 == \taskId\dataHost 的数量 则将 taskNode.setStatus(2) taskNode.statue = 2 mycat 拦截所有的写和修改的操作.(通过路由拦截) 定时器 (SwitchPrepareChecker) 执行检查是否所有的事务执行完成等。 1. 所有操作中不在执行事务. 2. binlog 中没有事务 如果, 所有的迁移任务(\taskId\localhost 下的任务)的 taskStatus 设置 status 为 3; 并且断开 binlog 的增量复制. 向_commit 提交 SwitchCommitListener 监听 commit 的提交, 当所有的提交的数量等于集群配置的数量时候, taskNode.setStatus(3); 并且 开始切换 rule 和 schema 的配置, 完成之后 zk 中添加

clean/mycat_fz_01 准备进行数据的清理. SwitchCleanListener 监听 clean 的提交, 恢复路由, 关闭监听器

(clean, prepare, commit)的, 删除 slavelDs/dataHost/id 这个标识, taskNode.setStatus(5) 所有的切换操作完成.

6.9.5 异常处理

在 zookeeper 中的 taskId 节点中的数据有 exception 的字段,是部分的运行的异常信息

taskId 节点中的"exception": ""

taskId 节点中的"backupFile": "xxxx",文件夹的命名规则是 backup 年_月_日_时_分

另外 zookeeper 中以 dataHost 对应的节点,以 dnxxx-dnxxx 命名的节点是迁移的 mysqldump 的任务信息,里有迁移的状态

迁移任务运行前的配置信息备份路径,这个目录从 zookeeper 备份了

ruledata, schema/schema, rules/function 的配置信息

如果出现了异常,需要清除 zk 的状态和还原本地的文件和人工还原数据库。如果迁移过程出现了异常,导致 mycat 重启之后不能使用迁移命令,请查看 zk 中的 mycat/slaveIds 和 mycat/migrate 的迁移状态信息,把里面的数据删除即可(主要是删除\mycat\mycat-cluster-1\slaveIds 和\mycat\mycat-cluster-1\migrate\taskid)

需要根据情况同时删除 conf\ruledata\crc32xxxx.properties 和 zookeeper 中的 ruledata 中的 crc32xxxx.properties

需要按情况把本地的 schema.xml 和 zookeeper 中的 schema 中的对应的 table 的 dataNode 属性还原到迁移之前的值

如果遇上了下面没有指出的异常情况,请细节检查一些基本配置问题,比如 dataNode 的名字对上了,但是 dataNode 对应的 dataHost 没有对上或者迁移的表不是空的,导致分片范围计数错误,这些细节可能导致 commit 阶段.

异常信息:

```
table cannot be null
```

-table 命令没写表名

```
add cannot be null
```

-add 命令没添加的 dataNode

Mycat can temporarily execute the migration command.If other mycat does not connect to this zookeeper, they will not be able to perceive changes in the migration task.

You can command as follow:

```
migrate -table=schema.test -add=dn2,dn3 -force=true
```

to perform the migration.

此时 mycat 已经连接上了 zookeeper, 添加-force 参数可以执行迁移命令。如果其他已经连接上这个 zookeeper 的 mycat 是 loadZk=true 启动的, 那么它们可以接收到这个 mycat 的迁移命令的通知。如果不是 loadZk=true 启动的, 但是已经执行过-force 的迁移命令的, 那么他们也可以收到 mycat 迁移命令的通知。

zookeeper 的任务信息可以禁止多次创建相同 schema 和相同 table 的任务同时可以通知其他 mycat 节点在迁移完成后自动更新 schema 配置。

```
Mycat is not connected to zookeeper
```

mycat 没有连接上 zookeeper, 这个异常是执行迁移命令检查的。

```
Unknown database 'xxx'
```

当前默认的 schema 或者参数中指定的 schema 没有与 mycat 的 schema 匹配上, 不存在

```
Table 'xxx' doesn't define in schema 'xxx'
```

迁移命令中指定的 table 不在 schema 中

```
table: xxx rule is not be PartitionByCRC32PreSlot
```

table 的 rule 不是 PartitionByCRC32PreSlot 的, 即不是 rule="crc32slot"

```
table: xxx previous migrate task is still running,on the same time one table only one task
```


情况 1: mycat 整个集群没有异常, 只是迁移命令在这个集群中又被执行了一次, 被禁止执行重复命令。情况 2: zookeeper 里面有这个任务的信息, 但是并没有 mycat 在执行这个命令的任务, 可能是遇上了错误关闭 mycat 等情况。需要人工参与修复。

```
migrate error:xxx
```

构造迁移任务时候出错了, 具体情况具体分析。

```
Mycat is not connected to zookeeper!!
```

```
Please start zookeeper and restart mycat so that this mycat can temporarily  
execute the migration command.If other mycat does not connect to this zookeeper,  
they will not be able to perceive changes in the migration task.
```

```
After starting zookeeper,you can command tas follow:
```

```
migrate -table=schema.test -add=dn2,dn3 -force=true
```

```
to perform the migration.
```

mycat 使用 MycatStartup 启动的时候没有连接上 zookeeper 导致 MigrateHandler 类初始化时候连接 zookeeper 失败。只能此时迁移命令完全不可用, 只能查到具体原因分析了, 一般就是 zookeeper 没有启动导致的。

```
Not support charset XXX
```

使用-charset=UTF-8 选项所选的编码不被 Java 环境支持

```
in zookeeper is abnormal state,please repair manual!
```

迁移任务可能还没有开始,mycat 就发生了异常,之后又输入了相同的迁移命令,就有可能出现这个异常,主要原因是 zookeeper 的 taskNode 节点还没有建立,需要手动根据清理 zookeeper 状态以及检查 mycat 日志

以下是 mycat 输出的日志,这个日志不会在 MySQL 命令响应的信息中显示,这些可能对异常排错有帮助

```
-----check dataNode-----  
dataNode %s will be not participate in migration
```

这个日志的意思是迁移任务所需的 dataHost 在 myid.properties 文件中的

boosterDataHosts=localhost1,localhost2,...没有对上,这些 datahost 的迁移任务会被忽略,所以需要迁移用到的 dataHost 需要在此写清楚,并且与 schema.xml(即 zookeeper 中的 schema)中的 dataNode 节点中的 DataHost 属性对应.

```
-----task created success-----
```

迁移任务创建成功,此时 zookeeper 存有 taskId 的数据

```
----->migrate binlog event:xxxx
```

这条信息是 BinlogStream 增量复制的消息,可以根据这条信息的输出确定迁移任务正在运行

migrate 中 mysqldump 准备执行命令,如果超长时间没有响应则可能出错

这条信息是执行一个任务里的 mysqldump 命令前的提示信息

migrate 中 准备自动创建新的 table:

这个信息说明 mysqldump 已经执行成功,将要执行创建 table

migrate 中 进入 mysqldump 阶段

这条信息是开启多个 mysqldump 前的提示信息

migrate 中 binlog 连接 异常

binlog stream 增量复制异常,需要手动修复

migrate 中 clean 阶段异常

clean 阶段异常,迁移任务完成,临时数据清理异常,手动清理 zookeeper 的 taskId 任务和本地文件 temp/dump 就可以

migrate 中 commit 阶段异常

commit 阶段会进行 schema,rules 和 ruledata 的 zookeeper 和本地数据更新,此时失败需要具体分析,可能需要导入数据并进行迁移任务

migrate 中 强制更新本地文件失败

更新本地的 schema.xml 与 rule.xml 与 ruledata 的 crc32slot_表名.properties 失败,但是这个异常并不会影响后面的流程执行

```
"-----从 zookeeper 中拉取的新的 schema 的信息-----  
-----  
-----从 zookeeper 中拉取的新的 dataNode 的信息-----  
-----  
-----从 zookeeper 中拉取的新的 dataHost 的信息-----  
-----  
-----从 zookeeper 中拉取的新的 tablerule 的信息-----  
-----  
-----从 zookeeper 中拉取的新的 function 的信息-----  
-----  
-----从 zookeeper 中拉取的新的 ruleData 的信息-----  
-----
```

commit 阶段,把新的配置文件更新到 zookeeper 后重新拉取的信息

migrate 中 switch prepare 阶段异常

这条信息说明 prepare 阶段异常

```
crc32slot_xxxx.properties does not match schema table dataNode.
```

zookeeper 或本地中的 crc32slot_xxx.properties 与本地的 schema 的 table 的 dataNode 节点没有对应上,需要根据数据库实际判断,可以把 crc32slot%s.properties 删除,让 mycat 重启重新生成新的 crc32slot_xxxx.properties

```
some slot has not moved to
```

添加节点后,没有产生需要迁移任务,即不需要迁移

```
The dataNode {newDataNode} that needs to be added already exists"
```

逻辑表已经包含新添加的节点

```
The dataNode {newDataNode} does not exist
```

新添加的节点不存在

6.9.6 数据迁移测试

在 zkconf 下配置文件

现有数据库 db1,db2 里面都没有表

修改 myid.properties 中的

```
loadZk=true  
  
boosterDataHosts=localhost1
```

将 mycat 的 schema 配置成

```
<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">  
  <table name="travelrecord" dataNode="dn1,dn2" rule="crc32slot" />  
</schema>
```

dataNode="dn1,dn2" 两个节点

```
rule="crc32slot"
```

```
<dataNode name="dn1" dataHost="localhost1" database="db1" />
  <dataNode name="dn2" dataHost="localhost1" database="db2" />
  <dataNode name="dn3" dataHost="localhost1" database="db3" />
```

其中 dn3 是将要迁移到的节点

myid.properties 中的 localhost1 与这里的 localhost1 对应

```
<dataHost name="localhost1" maxCon="1000" minCon="10" balance="0" slaveIDs="1,2,3"
  writeType="0" dbType="mysql" dbDriver="native"
  switchType="1" slaveThreshold="100" >
```

slaveIDs="1,2,3"

在 zkconf 下配置文件配置好, 然后通过 init_zk_data.sh 脚本初始化到 zk

此时启动 MycatStartup,使用 zookeeper 配置启动

mysqld 输入 sql

```
USE TESTDB;
CREATE TABLE `travelrecord` (
  id xxxx
  xxxxxxxx
) ENGINE=INNODB DEFAULT CHARSET=utf8;
```

即可创建表

查看 ruledata\crc32slot_TRAVELRECORD.properties 文件,查看分片范围,

之后请使用 mycat 导入数据,导入的数据量超过一个分片范围即可

```
USE TESTDB;
MIGRATE -TABLE=travelrecord -ADD=dn3;

0925b9f891234a9e837027920b46fad0
```

返回 TASK_ID

打开 ZooInspector 连接上 zookeeper 查看 migrate\TESTDB\TASK_ID 看到 status 为 5 即数据迁移成功

6.9.7 数据自动迁移测试过程中可能有用的检查点

数据自动迁移必须以下基本功能:

支持单个表迁移, 添加一个或者多个节点

支持多个表同时迁移, 添加一个或者多个节点

一个 schema 上的 table 正在迁移, 禁止再次对其迁移

迁移失败的可能检查点:

zk 是否正常连接, 集群是否正常运行

数据落在 ruleData 范围内

新添节点上的库的表不为空, 有数据导致计数错误

不是 crc32 算法

dataHost 不存在

schema 不存在

DataNode 上对应的 DataHost 不存在

编码不对

boosterDataHosts 不与 dataHost 匹配

本地的 ruleData 导致 zk 的 ruleData 有误(需要把两处的 ruleData 删除)

zk 里面的 ruleData 有错误的数据(错误的数据需要通过删除来解决, 导入不一定能覆盖数据)

第 7 章 版本选择与升级指南

7.1 版本选择

目前 Mycat 已经开发到了 1.4 版本预计本书发布不久后就可以发布 1.4alpha 版本, 1.4 几乎完全兼容之前所有版本, 如果你是研究阶段可以用 1.4 作为研究, 目前 1.3 版本中 1.3.0.3 是最稳定的版本, 可以放心用于生产, 1.3 系列只做 bug 修复, 不再进行功能升级, 如果需要最新的功能可以用 1.4。

7.2 mycat1.2 中的功能

ER 分片

全局表

读写分离支持

1.3 中的读写分离模式为：默认事务内的 sql 都会走写节点，非事务内的节点会根据配置的 balance 做负载，不支持手动选择 select 走写节点，如果需要 select 走写节点需要添加事务。

全局序列号与自增主键支持，分为本地文件与数据库两种方式。

默认 sql 解析器为 founddb。

7.3 mycat1.3 中的功能

dump 批量导入，导入列必须指定。

insert 多 values 支持。

jdbc 多数据库支持，部分分页特性不支持。

Nosql 支持，引入 mongodb。

catlet 支持。

主键缓存只能路由优化。

支持的分片规则有，

AutoPartitionByLong

PartitionByDate

PartitionByFileMap

PartitionByLong

PartitionByMod

PartitionByMurmurHash

PartitionByPattern

PartitionByPrefixPattern

PartitionByString

PartitionDirectBySubString

增加 LockTable 和 UnlockTables 语句支持。

多租户实现。

默认 sql 解析器为 Druid，sql 的兼容性进一步提高。

节点通配方式为：

```
<table name="oc_call" primaryKey="ID" dataNode="distribute(dn1$0-371,dn11$0-371)" rule="latest-month-calldate" /> </schema>

<dataNode name="dn1" dataHost="localhost1" database="db$0-371" />

<dataNode name="dn11" dataHost="localhost2" database="db$0-371" />
```

表的节点配置中，有默认节点，如果全部的表不分片则配置默认节点，不支持部分不分片的表不配置，所有表必须配置。

7.4 mycat1.4 中的功能

loaddata 批量导入支持。

sql 拦截

读写分离 在 1.3 基础上扩展特性，支持手动选择 sql 走读还是走写。

jdbc 多数据库分页支持。

自主主键支持批量插入。

新增分片规则:LatestMonthPartion,PartitionByMonth

1.4 中的通配符为：

table 节点的 dataNode 属性,其中的 offer_dn\$0-3 等价于 offer_dn1, offer_dn2, offer_dn3 共 3 个节点

dataNode 节点的通配配置

分三种情况：

1. 同一个 dataHost 上有多个 database

```
<dataNodename= "dn$1-3" dataHost= "test1" database= "base$1-3" />
```

等价于 3 个 dataNode 节点，其中 name 和 database 中的通配数量必须相等。

```
<dataNode name= "dn1" dataHost= "test1" database= "base1" />
```

```
<dataNode name= "dn2" dataHost= "test1" database= "base2" />
```



```
<dataNode name= "dn3" dataHost= "test1" database= "base3" />
```

2. 多个 dataHost 上有相同的 database

```
<dataNode name= "dn$1-3" dataHost= "test$1-3" database= "base" />
```

等价于 3 个节点，其中 name 和 dataHost 中的通配数量必须相等。

```
<dataNode name= "dn1" dataHost= "test1" database= "base" />
```

```
<dataNode name= "dn2" dataHost= "test2" database= "base" />
```

```
<dataNode name= "dn3" dataHost= "test3" database= "base" />
```

3. 多个 dataHost 上有相同的多个 database

```
<dataNode name= "dn$1-6" dataHost= "test$1-3" database= "base$1-2" />
```

等价于 6 个节点，有 3 个 dataHost，每个 dataHost 上都有 2 个 database。、其中 name 的通配数量必须等于 dataHost 数量乘以 database 数量、<dataNode name= "dn1" dataHost= "test1" database= "base1" />

```
<dataNode name= "dn2" dataHost= "test1" database= "base2" />
```

```
<dataNode name= "dn3" dataHost= "test2" database= "base1" />
```

```
<dataNode name= "dn4" dataHost= "test2" database= "base2" />
```

```
<dataNode name= "dn5" dataHost= "test3" database= "base1" />
```

```
<dataNode name= "dn6" dataHost= "test3" database= "base2" />
```

支持 MySQL 主从复制状态绑定的读写分离机制

表的节点配置中，添加对不分片的表不配置，走默认节点支持。

7.5 mycat1.5 中的功能

1. 5-RELEASE

新功能

- 支持常见 mysql gui 不填写默认 dbname。
- 支持 navicat 的 showtable 语句。
- 支持 show full table from。

改进和修复

- 修复 mycat 版本导致应用驱动识别错误无法支持毫秒。
- 修复 分片节点 第一个节点不是默认节点时候 desc table 路由到默认节点的 bug 表名大写转换。
- 修复 去掉分号 bug 结尾-1 不是-2。
- 测试：堆栈实现解析表名。
- 修复 重新 xml dtd 验证失败问题。
- SQL 汇总统计 清理参数。

1.5-GA

新功能

- 高频 SQL 分析 加 user。
- xml 转换 yaml 命令行工具。
- 新增 SQL / HIGH / SLOW / TABLE 指令 clear 参数 true 表示清除 cache， 如: show @@sql true; show @@sql.high false;
- 配合 mycat eye SQL 监控持久化，获取数据后清理。
- SHOW @@White ip 白名单。

改进和修复

- 修复 wapper 日志无用输出。
- 修复 hint sql type 引擎的问题。
- 修复重写 xml dtd 丢失问题。
- 白名单 写回文件。
- ip 类型写错。
- 修复 注解 SQL 的 sqlType 与实际 SQL 的 sqlType 不一致问题。
- 根据用户的反馈，修复 QueryResult 在高并发的情况下 endTime 时间有延迟的问题, sql/high/slow/table 新增 clear 参数。
- fix bug for multi-tenancy using `#!/mycat:schema=DB1/ select * from table (oracle)`。
- 为 MyCat 的 SERVER VERSION 增加了注释，增加了一个 dump，可供调试时输出内容。
- Change version info。
- 增加 SET IGNORE UTIL。
- 实际使用中 PHP 用户经常会操作多个 SET 指令组成一个 Stmt，所以该指令检测功能独立出来
- 增加 reload @@sqlstat=open/close 指令到 help。
- fix 带物理库名路由到随机节点的 bug。
- 更换 License。
- fix sharejoin bug。

MyCat 1.5-ALPHA

新功能

- 新增支持 joinkey 为 varchar 类型的 sharejoin。
- 增加控制指令，可关闭或打开实时统计分析的功能。
- 忽略部分 SET 指令, 避免 WARN 不断的刷日志。
- 默认 mycat 统计分析模块为打开状态。
- 可通过如下指令设置关闭或打开 实时统计分析模块。
- reload @@sqlstat=close;
- reload @@sqlstat=open;
- 增加 SQL 条件的分析，用于 列值/访问次数 的实时统计。
- 支持设置规则 reload @@query_cf=表名&字段。
- 支持清除规则 reload @@query_cf=NULL。
- 支持 show @@sql.condition。
- 新增 setnodes 方法。
- xml to yaml tool。

改进和修复

- 修改 isInit 需要声明为 volatile。
- fix: HintHandlerFactory 线程安全问题和多次重复初始化的问题。
- -close connection,reason:program err:java.lang.IndexOutOfBoundsException。
- 问题 load 大文件出现临文件 dn1.txt 找不到，load 大文件出现空指针的异常，load 出现路由错误的问题 from berylgreen。
- fix gen zkurl bug。
- 修复 reload_all 的 bug。
- zk-create 文件有问题。
- 修复:客户端字符集同步不一致。
- 后端链接在同步完毕之后才回调修改当前后端链接的字符集,导致第一条发送出去的字符是使用后端链接的字符集进行编码的,而不是真正前端链接的字符集,导致编码出错。
- 添加 insert 误判的单元测试。
- 非彻底解决 insert 语句误判问题。。
- 修复 explain insert 执行的 bug。
- done load configuration from zookeeper。
- 修复 sql 统计列表 里面看到好多非业务 sql 问题，可能的慢 sql 里面 sql 执行时间不太对
- 添加默认不使用 zookeeper 进行加载。
- done load configuration from zookeeper。
- wrapper.ping.timeout。
- 修复在注解方式批量导入时，自增字段不能正确获取的问题（以本地时间算法的自增方式）。
- 解决 mycat 内部统计的 druid sql parse 类型转换错误。

7.6 mycat1.6 中的功能

1. 6-RELEASE

新功能

- 添加 show @@directmemory 监控命令。
- 新增 lock tables 功能。
- reload @@config_all 支持不影响当前事务。
- prepare 指令支持 blob。
- 分片表配置检查。
- zk 模块重构。

改进和修复

- 修复去库名 bug。
- 修复 group by 结果集错误。
- 处理关闭流问题，为日志输出增加堆栈打印。

1. 6-BETA

新功能

- 增加了用户 db/table 表级的 DML 语句权限控制。
- 重构原有隔离区，改为 firewall。
- 添加新路由规则，根据日期查询日志数据 冷热数据分布 ，最近 n 个月的到实时交易库查询，超过 n 个月的按照 m 天分片。

改进和修复

- change load data max column setting。
- 修复堆外排序的若干错误等的防火墙 BUG。
- 修复 prepare 指令多节点返回错误和单节点返回错误。
- 修复后端使用 pg 原生协议时当查询数据量大时原有读取方式 会出现 nio 的粘包问题。
- 解决数据类型 COL_TYPE_LONG 和 row 中列为 null 时，引起 Mycat 异常。
- 修复后端 pg 原生协议时类型错误、统计函数错误、bufferpool 使用等错误。
- 统一定时器时间单位为毫秒。
- 初步重构 zk 配置统一从 myid.properties 取。
- 修复 ShareJoin 关联右表没执行。
- 修复 mergeColsMap 空指针报错问题。

- 修复 schema.xml 中 配置 checkSQLschema="true"，sql 语句中含 schema 时，有 bug。
- 修复查询语句表名中存在【`】符号时无法路由至对应分片。
- 按天分片，跨头尾分片 BUG 修改。
- 修复 日志路由规则错误。
- 修改对于 update 语句中 set 子句包含分片字段更新语句的处理逻辑。

1.6-ALPHA

新功能

- 非堆内存(Direct Memory)处理跨分片结果集的 Merge/order by/group by/limit。
- 两种基于 zk 的全局序列。
- 停机扩容缩容工具，支持任意路由规则。
- 全局表一致性检测。
- server.xml 中添加配置项让 mycat 可以设置要模拟的 mysql 的版本号。
- 缓存池管理支持 DirectByteBufferPool 和 ByteBufferArena 切换。
- 新的注解方式 hint sql 支持的格式/** mycat: */。
- postgres 的 native 协议支持。
- 支持 mysql 和 oracle 存储过程，out 参数、多结果集返回。
- 预编译 prepare 的支持。
- master/slave 注解。
- 库内分表特性。
- 支持自生成 ID 的 batchInsert。
- 支持 rails 的 set names 语句。
- 新增 show @@sql.resultset 统计大结果集记录及其系统配置。
- TxReadOnly 支持。
- 兼容 PhpAdmin's 控制台管理,支持 mysql information_schema 元数据返回

改进和修复。

- navicat stat sql bug。
- PartitionByMod 算法未考虑引号的问题。
- 修复跨分片查询时空指针报错问题。
- Allow % in user name, which is used in some cloud MySQL DB。
- 心跳切换的判断不应该判断读写分离的状态。
- 当分库字段为 uuid 时，使用 sharding-by-murmur 规则配置主子表关系，导致主子表关联数据无法插入到同一个库中。
- 监测数据库同步状态，在 switchType=-1 或者 1 的情况下，也需要收集主从同步状态。
- 优化 SQLStat 导致性能下降。
- 后端连接切换或者挂掉修复。

- 对于 ShareJoin 的 bug 修改。
- receive rollback, but found backend connection is closed or quit。
- 按月分片设置起始月份范围从而循环使用, 对落此范围外的数据通过计算偏移得到目标分片。
- Optimization: handling oom error in NIOReactor and BufferPool class。
- Fix bug: a. Mycat hang problem b. SQL error and rollback blocked in。
- Fix bug, do not have having clause when route to single node。
- fix 遍历 map 的 bug, 事务隔离级别的优化, 完善 index_to_charset.properties。
- fix RouteStrategyFactory 线程安全问题 和 DefaultSqlInterceptor 导致的无法在末尾插入\字符的问题。
- fix DefaultSqlInterceptor 中为了支持 foundationdb parser 而进行的字符转换, 导致无法插入\。
- 修改 tryExistsCon 函数支持 master/slave 注解。
- int to long 防止发生越界。
- 主从同步切换 show slave status 的情况下, 修复 正常的 read host 不可用问题, 及 stat 处的 bug 修正。

1.65

新功能

- 心跳检测功能增强
- 两个表标准 JOIN 的支持
- 心跳切换同步
- 全局表一致性检测命令
- 数据自动迁移优化

7.7 小结

目前稳定版为 1.65

各版本的升级直接到 github 下载对应版本的最新更新日期版本进行升级。

<https://github.com/MyCATApache/Mycat-download>

7.8 1.6 升级指南

1.6 版本增加了大量新的功能, 同时对包结构进行了较大的重构。所以从 1.6 之前版本升级时请注意以下事项:

1. schema.xml 等 xml 文件的头部改为 `<mycat:schema xmlns:mycat="http://io.mycat/">`

2. rule.xml 中的<function name="murmur"

class="io.mycat.route.function.PartitionByMurmurHash">

Class 路径改成新的包路径

3. sharejoin 等注解包名 io.mycat.catlets.ShareJoin

第 8 章 性能调优

8.1 主机调优

Linux 主机的网络性能优化, mycat 所在服务器多网卡绑定, bond 技术, 增加网络吞吐量。

TCP 的性能取决于几方面因素, 最重要的是链接带宽(link bandwidth)(报文在网络上传输的速率)和往返时间(round-trip time)或 RTT(发送报文与接收到另一端的响应之间的延时)。这两个值确定称为 BDP(Bandwidth Delay Product)的内容。BDP 给出一种简单的方法计算理论上最优的 TCP Socket 缓冲区大小(其中保存排队等待传输和等待应用程序接收的数据)。缓冲区太小, TCP 窗口就不能完全打开, 这会限制性能; 缓冲区太大, 则会浪费宝贵的内存资源; 设置的缓冲区大小合适, 就可完全利用可用带宽。

BDP 计算公式:

$$BDP = \text{link bandwidth} \times \text{RTT}$$

若应用程序通过一个 100MB / s 的局域网通信, 其 RRT 为 500ms, 则 BDP 为: $100\text{MB} / \text{s} \times 0.0005 = 50\text{KB}$ 。

Linux 2.6 默认的 TCP 窗口大小是 110KB, 这将连接的带宽限制为 22M/S, 计算方法如下:

$$\text{throughput} = \text{window\_size} / \text{RTT}$$
$$110\text{KB} / 0.0005 = 2.2\text{MB} / \text{s}$$

使用上面计算的窗口大小, 得到带宽为 12.5 MB / s, 即:

$$625\text{KB} / 0.0005 = 12.5\text{MB} / \text{s}$$

应用可以根据自己的 Socket 计算最优的缓冲区大小。Socket 提供几个 Socket 选项，其中两个可以用于修改 Socket 的发送和接收缓冲区的大小。使用 SO_SNDBUF 和 SO_RCVBUF 选项来调整发送和接收缓冲区的大小。在 Linux 2.6 内核中，发送缓冲区的大小由调用用户定义，而接收缓冲区会自动加倍。通过计算合理设置缓冲区的大小，Socket 网络传输带宽的资源将得到充分利用，从而提高了传输性能。

8.2 JVM 调优

Mycat 的 jvm 相关配置是在 warrper 启动中配置例如：

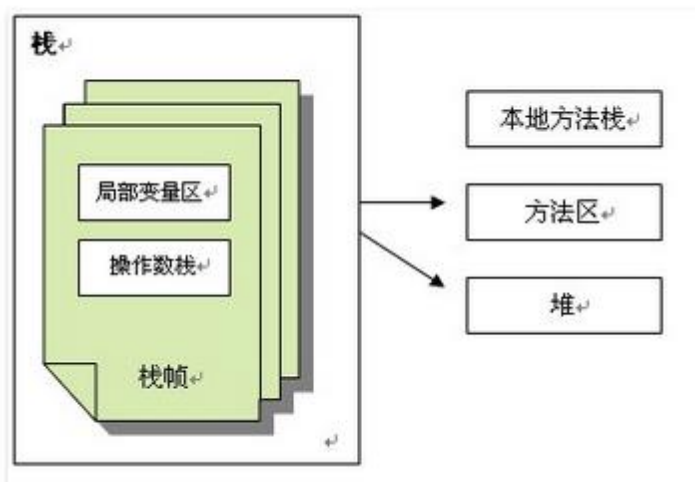
linux 下 startup_nowrap.sh

其他版本都会在对应的配置文件中配置。

8.2.1 JVM 结构

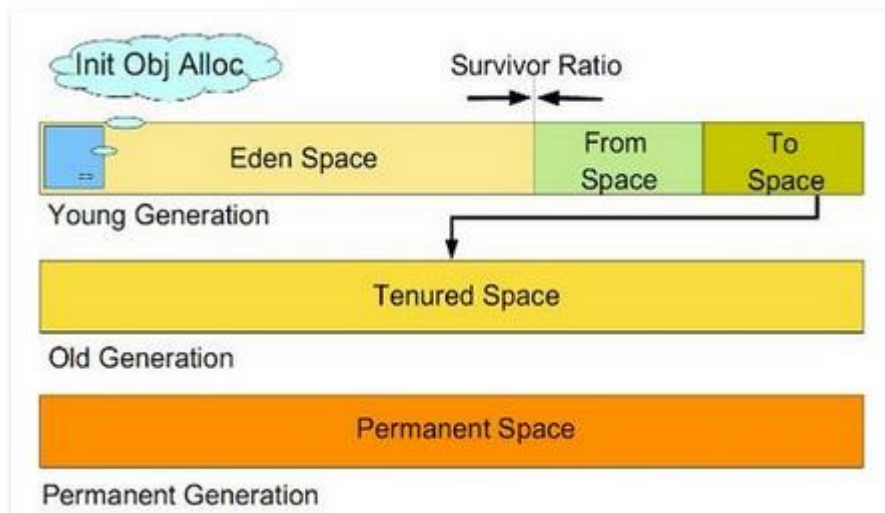
JVM 内存结构由堆、栈、本地方法栈、方法区等部分组成，另外 JVM 分别对新生代和旧世代采用不同的垃圾回收机制。

1. 首先来看一下 JVM 内存结构，它是由堆、栈、本地方法栈、方法区等部分组成，结构图如下所示。

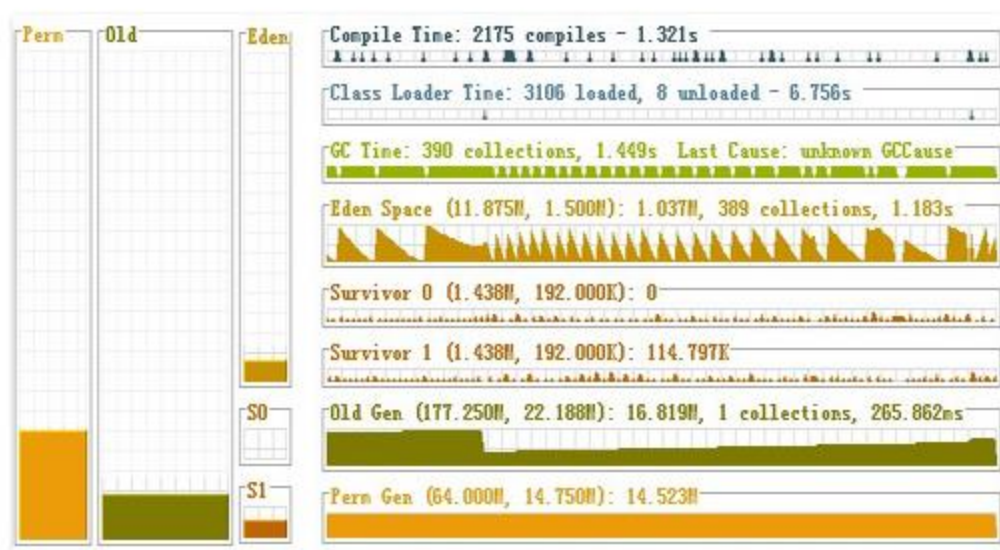


1) 堆

所有通过 new 创建的对象内存都在堆中分配，其大小可以通过 -Xmx 和 -Xms 来控制。堆被划分为新生代和旧世代，新生代又被进一步划分为 Eden 和 Survivor 区，最后 Survivor 由 FromSpace 和 ToSpace 组成，结构图如下所示：



新生代。新建的对象都是用新生代分配内存，Eden 空间不足的时候，会把存活的对象转移到 Survivor 中，新生代大小可以由-Xmn 来控制，也可以用-XX:SurvivorRatio 来控制 Eden 和 Survivor 的比例旧生代。用于存放新生代中经过多次垃圾回收仍然存活的对象 2)栈 每个线程执行每个方法的时候都会在栈中申请一个栈帧，每个栈帧包括局部变量区和操作数栈，用于存放此次方法调用过程中的临时变量、参数和中间结果 3)本地方法栈 用于支持 native 方法的执行，存储了每个 native 方法调用的状态 4)方法区 存放了要加载的类信息、静态变量、final 类型的常量、属性和方法信息。JVM 用持久代(PermanetGeneration)来存放方法区，可通过-XX:PermSize 和-XX:MaxPermSize 来指定最小值和最大值。



对 JVM 内存的系统级的调优主要的目的是减少 GC 的频率和 Full GC 的次数，过多的 GC 和 Full GC 是会占用很多的系统资源（主要是 CPU），影响系统的吞吐量。特别要关注 Full GC，因为它会对整个堆进行整理，导致 Full GC 一般由于以下几种情况：

旧生代空间不足。

调优时尽量让对象在新生代 GC 时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧世代创建对象。

Perm Genet Generation 空间不足。

增大 Perm Gen 空间，避免太多静态对象。

统计得到的 GC 后晋升到旧生代的平均大小大于旧世代剩余空间。

控制好新生代和旧生代的比例。

System.gc()被显示调用。

垃圾回收不要手动触发，尽量依靠 JVM 自身的机制。

调优手段主要是通过控制堆内存的各个部分的比例和 GC 策略来实现，下面来看看各部分比例不良设置会导致什么后果。

1) 新生代设置过小

一是新生代 GC 次数非常频繁，增大系统消耗；二是导致大对象直接进入旧世代，占据了旧世代剩余空间，诱发 Full GC。

2) 新生代设置过大

一是新生代设置过大会导致旧世代过小（堆总量一定），从而诱发 Full GC；二是新生代 GC 耗时大幅度增加一般说来新生代占整个堆 1/3 比较合适。

3) Survivor 设置过小

导致对象从 eden 直接到达旧世代，降低了在新生代的存活时间。

4) Survivor 设置过大

导致 eden 过小，增加了 GC 频率。

另外，通过-XX:MaxTenuringThreshold=n 来控制新生代存活时间，尽量让对象在新生代被回收

内存管理和垃圾回收 可知新生代和旧世代都有多种 GC 策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM 提供两种较为简单的 GC 策略的设置方式。

1) 吞吐量优先

JVM 以吞吐量为指标，自行选择相应的 GC 策略及控制新生代与旧世代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n 来设置。

2) 暂停时间优先

JVM 以暂停时间为指标，自行选择相应的 GC 策略及控制新生代与旧生代的大小比例，尽量保证每次 GC 造成的应用停止时间都在指定的数值范围内完成。这个值可由 `-XX:MaxGCPauseRatio=n` 来设置。

最后汇总一下 JVM 常见配置。

堆设置

`-Xms`:初始堆大小 `-Xmx`:最大堆大小。

`-XX:NewSize=n`:设置年轻代大小 `-XX:NewRatio=n`:设置年轻代和年老代的比值。如:为 3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4。

`-XX:SurvivorRatio=n`:年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个 Survivor 区占整个年轻代的 1/5。

`-XX:MaxPermSize=n`:设置持久代大小 收集器设置 `-XX:+UseSerialGC`:设置串行收集器。

`-XX:+UseParallelGC`:设置并行收集器 `-XX:+UseParalledlOldGC`:设置并行年老代收集器。

`-XX:+UseConcMarkSweepGC`:设置并发收集器 垃圾回收统计信息 `-XX:+PrintGC`。

`-XX:+PrintGCDetails` `-XX:+PrintGCTimeStamps`。

`-Xloggc:filename` 并行收集器设置 `-XX:ParallelGCThreads=n`:设置并行收集器收集时使用的 CPU 数。并行收集线程数。 `-XX:MaxGCPauseMillis=n`:设置并行收集最大暂停时间 `-XX:GCTimeRatio=n`:设置垃圾回收时间占程序运行时间的百分比。公式为 $1/(1+n)$ 。

并发收集器设置

`-XX:+CMSIncrementalMode`:设置为增量模式。适用于单 CPU 情况。

`-XX:ParallelGCThreads=n`:设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数。

系统相关 `-XX:+UseNUMA` numa 是一个 CPU 的特性。SMP 架构下，CPU 的核是对称，但是他们共享一条系统总线。所以 CPU 多了，总线就会成为瓶颈。在 NUMA 架构下，若干 CPU 组成一个组，组之间有点对点的通讯，相互独立。启动它可以提高性能。NUMA 需要硬件，操作系统，JVM 同时启用，才能启用。Linux 可以用 `numactl` 来配置 numa,JVM 通过 `-XX:+UseNUMA` 来启用。

`-XX:LargePageSizeInBytes=128m` 启用大内存页。

现在一个操作系统默认页是 4K。如果你的 heap 是 4GB，就意味着要执行 $1024*1024$ 次分配操作。所以最好能把页调大。这个配额设计操作系统，单改 Jvm 是不行的。Linux 上的配置有点复杂，不详述。在 Java1.6 中

UseLargePages 是默认开启的，LasrgePageSzieInBytes 被设置成了 4M。笔者看到一些情况下配置成了 128MB，在官方的性能测试中更是配置到 256MB。

以上说明绝大部分参考的网上资料，被文只是作为集中处理。

官方 JVM 参数说明：

<http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>

8.3 MyCAT 调优

MyCAT 所有的调优参数都可以才 server.xml 中找到。mycat 中几个关键的调优点已经 Mycat 性能调优指南.docx 中有所讨论，这里只做为该文档的补充。

本章主要讨论如下两个内容：

1. processors 数值的影响范围。
2. buffer 和 buffer 队列大小。

processors 数值定义了如下几个类的实例个数：

1. NIOProcessor。
2. NIOReactorPool。
3. AsynchronousChannelGroup。

NIOProcessor 类，持有所有的前后端连接，定期的空闲检查和写队列检查。要完成这个动作。Mycat 是通过遍历 NIOProcessor 持有的所有连接来完成的。

所以，可以适当的根据系统性能调整 NIOProcessor 的个数。使得前、后段连接可以均匀的分布在每个 NIOProcessor 上。这样，就可以加快每次的空闲检查和写队列检查。快速的将空闲的连接关闭，减轻服务器的内存使用量。

NIOReactor 是 NIO 中具体执行 selector 的类，当满足感兴趣的事件发生的时候，他就通知上次逻辑进行具体的处理。所以，NIOReactor 的个数据等于具体事件处理器的个数。如果系统的配置允许的话，应该尽可能的增大 NIOReactor 的数量。默认值是系统核心数。

AsynchronousChannelGroup 是 AIO 中必须提供的一个组成部分。AsynchronousChannelGroup 根据 processors 的数值，确定实例数和 channelGroup 组内的线程池大小。后端 AIO 连接循环取 AsynchronousChannelGroup 数组中的实例。所以。如果是在 AIO 模式下使用 Mycat 的话，调整这个参数也是有必要的。默认值是系统核心数。

最后，可以根据自己硬件的实际情况，配置 processors 的具体大小。例如，配置 processor 的个数为 16：

server.xml 文件中定义

```
<property name="processors">16</property>
```

还有一个要讨论的就是 buffer pool。因为，所有的 NIOProcessor 共享一个 buffer pool。

我们在 server.xml 中提到过：

BufferPool 的总长度 = bufferPool / bufferChunk

我们可以连接到 Mycat 管理端口上，使用 show @@processor 命令列出所有的 processor 状态。

查看列：FREE_BUFFER、TOTAL_BUFFER、BU_PERCENT。

如果 FREE_BUFFER 的数值过小，则说明配置的 buffer pool 大小可能不够。这时候就要手动配置根据公式这个属性了，pool 的大小最好是 bufferChunk 的整数倍。例如，配置 buffer pool 的大小为：5000

server.xml 文件中定义

```
<property name="processorBufferPool">20480000</property>
```

另一个 buffer pool 是线程内 buffer pool，这个值可以根据 processors 的数值计算出来。具体看 server.xml 配置详解。

8.4 MySQL 通用调优

首先 MySQL 要绝对避免使用 Swap 内存，网上有多种办法，可以参考。

这里是 MySQL5.6 及以上的调优参数，主要是提升多个 database/table 的写入和查询性能：

[mysqld]

当 Order By 或者 Group By 等需要用到结果集时，参数中设置的临时表的大小小于结果集的大小时，就会将该表放在磁盘上，这个时候在硬盘上的 IO 要比内存差很多。所耗费的时间也多很多，Mysql 会取 min(tmp_table_size, max_heap_table_size) 的值，因此两个设置为一样大小，除非是大量使用内存表的情况，此时 max_heap_table_size 要设置很大。

```
max_heap_table_size=200M
```

```
tmp_table_size=200M
```

下面这部分是 Select 查询结果集的缓存控制，query_cache_limit 表示缓存的 Select 结果集的最大字节数，这个可以限制哪些结果集缓存，query_cache_min_res_unit 表示结果集缓存的内存单元大小，若需要缓存的 SQL 结果集很小，比如返回几条记录的，则 query_cache_min_res_unit 越小，内存利用率越高，query_cache_size 表示总共用多少

内存缓存 Select 结果集，query_cache_type 则是控制是否开启结果集缓存，默认 0 不开启，1 开启，2 为程序控制方式缓存，比如 SELECT SQL_CACHE ...这个语句表明此查询 SQL 才会被缓存，对于执行频率比较高的一些查询 SQL，进行指定方式的缓存，效果会最好。

FLUSH QUERY CACH 命令则清理缓存，以更好的利用它的内存，但不会移除缓存，RESET QUERY CACHE 使命从查询缓存中移除所有的查询结果。

```
#query_cache_type =1
#query_cache_limit=102400
#query_cache_size = 2147483648
#query_cache_min_res_unit=1024
```

MySQL 最大连接数，这个通常在 1000-3000 之间比较合适，根据系统硬件能力，需要对 Linux 打开的最大文件数做修改

```
max_connections =2100
```

下面这个参数是 InnoDB 最重要的参数，是缓存 innodb 表的索引，数据，插入数据时的缓冲，尽可能的使用内存缓存，对于 MySQL 专用服务器，通常设置操作系统内存的 70%-80%最佳，但需要注意几个问题，不能导致 system 的 swap 空间被占用，要考虑你的系统使用多少内存，其它应用使用的内存，还有你的 DB 有没有 myisa 引擎，最后减去这些才是合理的值。

```
innodb_buffer_pool_size=4G
```

innodb_additional_mem_pool_size 除了缓存表数据和索引外,可以为操作所需的其他内部项分配缓存来提升 InnoDB 的性能。这些内存就可以通过此参数来分配。推荐此参数至少设置为 2MB，实际上，是需要根据项目的 InnoDB 表的数目相应地增加

```
innodb_additional_mem_pool_size=16M
```

innodb_max_dirty_pages_pct 值的争议，如果值过大，内存也很大或者服务器压力很大，那么效率很降低，如果设置的值过小，那么硬盘的压力会增加。

```
innodb_max_dirty_pages_pct=90
```

MyISAM 表引擎的数据库会分别创建三个文件：表结构、表索引、表数据空间。我们可以将某个数据库目录直接迁移到其他数据库也可以正常工作。然而当你使用 InnoDB 的时候，一切都变了。InnoDB 默认会将所有的数据库 InnoDB 引擎的表数据存储在一个共享空间中：ibdata1，这样就感觉不爽，增删数据库的时候，ibdata1 文件不会自动收缩，单个数据库的备份也将成为问题。通常只能将数据使用 mysqldump 导出，然后再导入解决这个问题。

innodb_file_per_table=1 可以修改 InnoDB 为独立表空间模式，每个数据库的每个表都会生成一个数据空间。

独立表空间

优点:

- 1.每个表都有自己独立的表空间。
- 2.每个表的数据和索引都会存在自己的表空间中。
- 3.可以实现单表在不同的数据库中移动。
- 4.空间可以回收 (drop/truncate table 方式操作表空间不能自动回收)
- 5.对于使用独立表空间的表, 不管怎么删除, 表空间的碎片不会太严重的影响性能, 而且还有机会处理。

缺点:

单表增加比共享空间方式更大。

结论:

共享表空间在 Insert 操作上有一些优势, 但在其它都没独立表空间表现好。

实际测试, 当一个 MySQL 服务器作为 Mycat 分片表存储服务器使用的情况下, 单独表空间的访问性能要大大好于共享表空间, 因此强烈建议使用独立表空间。

当启用独立表空间时, 由于打开文件数也随之增大, 需要合理调整一下 `innodb_open_files`、`table_open_cache` 等参数。

```
innodb_file_per_table=1
```

```
innodb_open_files=1024
```

```
table_open_cache=1024
```

Undo Log 是为了实现事务的原子性, 在 MySQL 数据库 InnoDB 存储引擎中, 还用 Undo Log 来实现多版本并发控制(简称: MVCC)。Undo Log 的原理很简单, 为了满足事务的原子性, 在操作任何数据之前, 首先将数据备份到 Undo Log, 然后进行数据的修改。如果出现了错误或者用户执行了 ROLLBACK 语句, 系统可以利用 Undo Log 中的备份将数据恢复到事务开始之前的状态。因此 Undo Log 的 IO 性能对于数据插入或更新也是很重要的一个因素。于是, 从 MySQL 5.6.3 开始, 这里出现了重大优化机会:

As of MySQL 5.6.3, you can store InnoDB undo logs in one or more separate undo tablespaces outside of the system tablespace. This layout is different from the default configuration where the undo log is part of the system tablespace. The I/O patterns for the undo log make these tablespaces good candidates to move to SSD storage, while keeping the system tablespace on hard disk storage. `innodb_rollback_segments` 参数在此被重命名为 `innodb_undo_logs`

因此总共有 3 个控制参数: innodb_undo_tablespaces 表明总共多少个 undo 表空间文件, innodb_undo_logs 定义在一个事务中 innodb 使用的系统表空间中回滚段的个数。如果观察到回滚日志有关的互斥争用, 可以调整这个参数以优化性能, 默认是 128 最大值, 官方建议先设小, 若发现竞争, 再调大

注意这里的参数是要安装 MySQL 时候初始化 InnoDB 引擎设置的, innodb_undo_tablespaces 参数无法后期设定。

```
innodb_undo_tablespaces=128
```

```
innodb_undo_directory= SSD 硬盘或者另外一块硬盘, 跟数据分开
```

```
innodb_undo_logs=64
```

下面是 InnoDB 的日志相关的优化选项

innodb_log_buffer_size 这是 InnoDB 存储引擎的事务日志所使用的缓冲区。类似于 Binlog Buffer, InnoDB 在写事务日志的时候, 为了提高性能, 也是先将信息写入 InnoDB Log Buffer 中, 当满足 innodb_flush_log_trx_commit 参数所设置的相应条件(或者日志缓冲区写满)之后, 才会将日志写到文件(或者同步到磁盘)中。innodb_log_buffer_size 不用太大, 因为很快就会写入磁盘。innodb_flush_log_trx_commit 的值有 0: log buffer 中的数据将以每秒一次的频率写入到 log file 中, 且同时会进行文件系统到磁盘的同步操作 1: 在每次事务提交的时候将 log buffer 中的数据都会写入到 log file, 同时也会触发文件系统到磁盘的同步; 2: 事务提交会触发 log buffer 到 log file 的刷新, 但并不会触发磁盘文件系统到磁盘的同步。此外, 每秒会有一次文件系统到磁盘同步操作。对于非关键交易型数据, 采用 2 即可以满足高性能的日志操作, 若要非常可靠的数据写入保证, 则需要设置为 1, 此时每个 commit 都导致一次磁盘同步, 性能下降。

innodb_log_file_size 此参数确定数据日志文件的大小, 以 M 为单位, 更大的设置可以提高性能, 但也会增加恢复故障数据库所需的时间。innodb_log_files_in_group 分割多个日志文件, 提升并行性。

innodb_autoextend_increment 对于大批量插入数据也是比较重要的优化参数 (单位是 M)

```
innodb_log_buffer_size=16M
```

```
innodb_log_file_size =256M
```

```
innodb_log_files_in_group=8
```

```
innodb_autoextend_increment=128
```

```
innodb_flush_log_at_trx_commit=2
```

#建议用 GTID 的并行复制, 以下是需要主从复制的情况下, 相关的设置参数。

```
#gtid_mode = ON
```

```
#binlog_format = mixed
```

```
#enforce-gtid-consistency=true
```


#log-bin=binlog

#log-slave-updates=true

开发篇

第 1 章 加入 Mycat

1.1 如何加入 Mycat

目前 Mycat 所用的语言为 Java，相关技术主要如下：

- Java Web 技术，参与 MyCAT Web 开发；
- JDBC 技术，可以完善 MyCAT Server 中的 JDBC 驱动部分；
- Java IO，多线程，算法，参与 MyCAT Server 与 MyCAT Balance 的代码优化和完善；
- SQL 优化与数据库技术，提供 MyCAT 智能优化的需求，实现和设计；
- NoSQL 技术，参与 MyCAT 支持 NoSQL 引擎的工作。

MyCAT Server 快速入门方式：

Eclipse 中启动 MyCAT 源码，进行调试，日志为 Debug 级别，学习了解 SQL 收取、解析、路由算法、SQL 执行逻辑、结果集处理等环节，了解工作机制。

关于通信部分，目前是 AIO 模型。

算法方面主要涉及到数据排序、分组等优化等。

建议熟悉 Java 文件映射内存的 API 和编程、高效多线程编程等技术。

欢迎针对任何需求的改进和完善，可以有缺陷，可以做的慢，只要不失联！！！！

MyCAT 官方交流 QQ 群：106088787

MyCAT 官网：<http://www.mycat.org.cn/>

MyCAT 源码及相关文档库：<https://github.com/MyCATApache/>

MyCAT 开发招募联系 QQ：294712221

1.2 如何获取源码

目前 MyCAT 最新程序的源码和文档都托管在 github 上，github 地址为：

第 2 章 Mycat 开发基础

2.1 代码调试入口

Mycat 运行的 main class 为 MycatStartup。在获取源代码之后，导入到 IDE 中。配置相关的启动参数就可以在 IDE 中调试 Mycat 了。

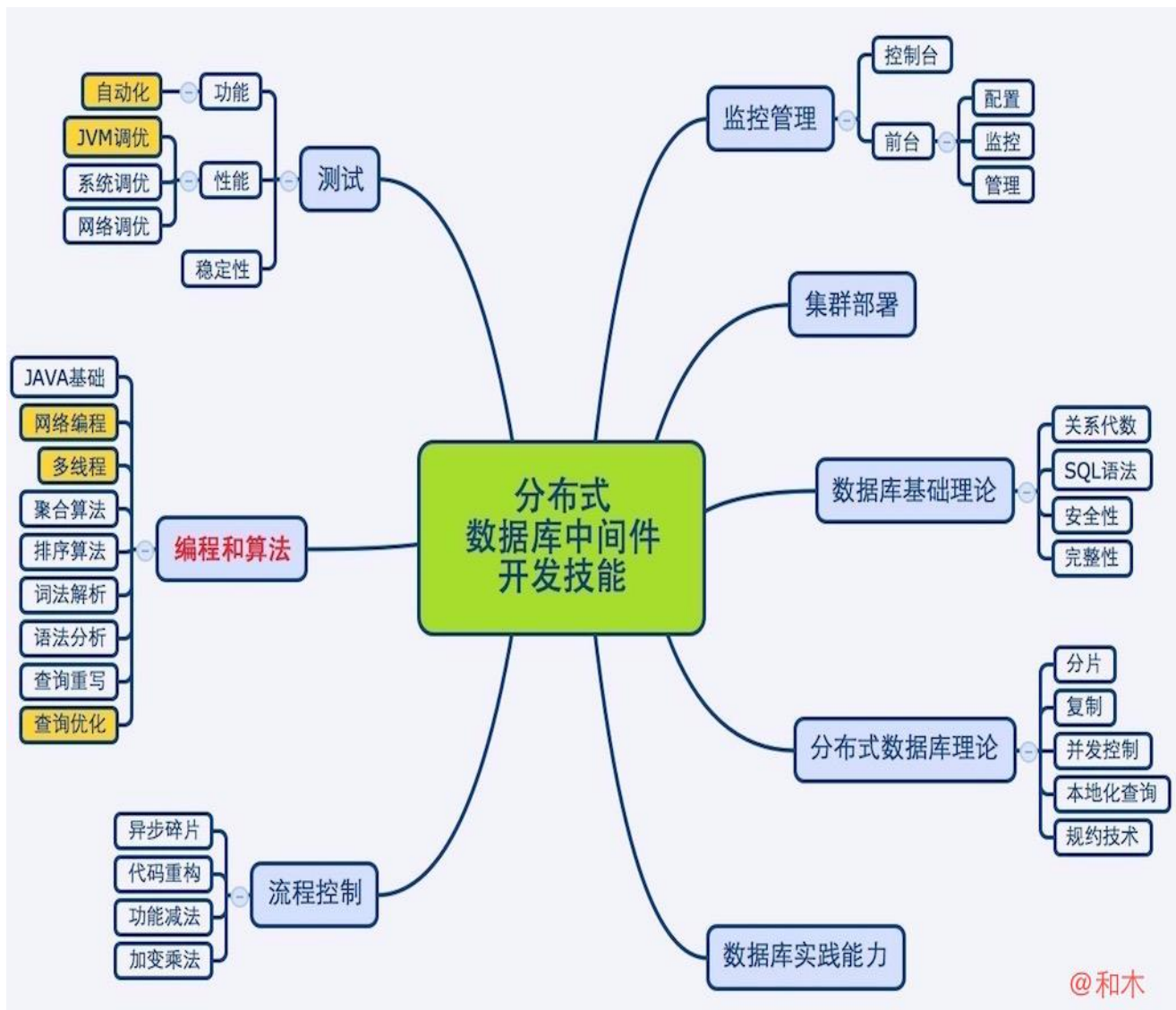
这里需要注意的是，需要指定 MYCAT_HOME 这个系统变量的值。这个值可以为任意的位置，不过一般是指定为与源代码同级的目录。可以在 IDE 运行选项内配置 VM OPTION。例如：

`-DMYCAT_HOME=D:\workspace\java\Mycat-Server。`

2.2 中间件开发技能

对中间件开发技能进行图形化展示，方便团队内各成员业余时间自学相关技能，其中：

- 多线程、网络编程、JVM 调优是无止境的，能多熟就多熟。
- 流程控制需要个人多思考，对于高性能框架，就是引入很多异步逻辑，进行碎片化编程。
- 不能一碰到需求就加一段代码而不管整体的融合性，不要只加不减，不时重构下结构删些代码多做些乘法。
- 各种理论知识要跟实践相结合，理论算法一个表现形式，真正落地时代码上则可能是另一种考虑，但总要略懂些。



第 3 章 Mycat 架构分析

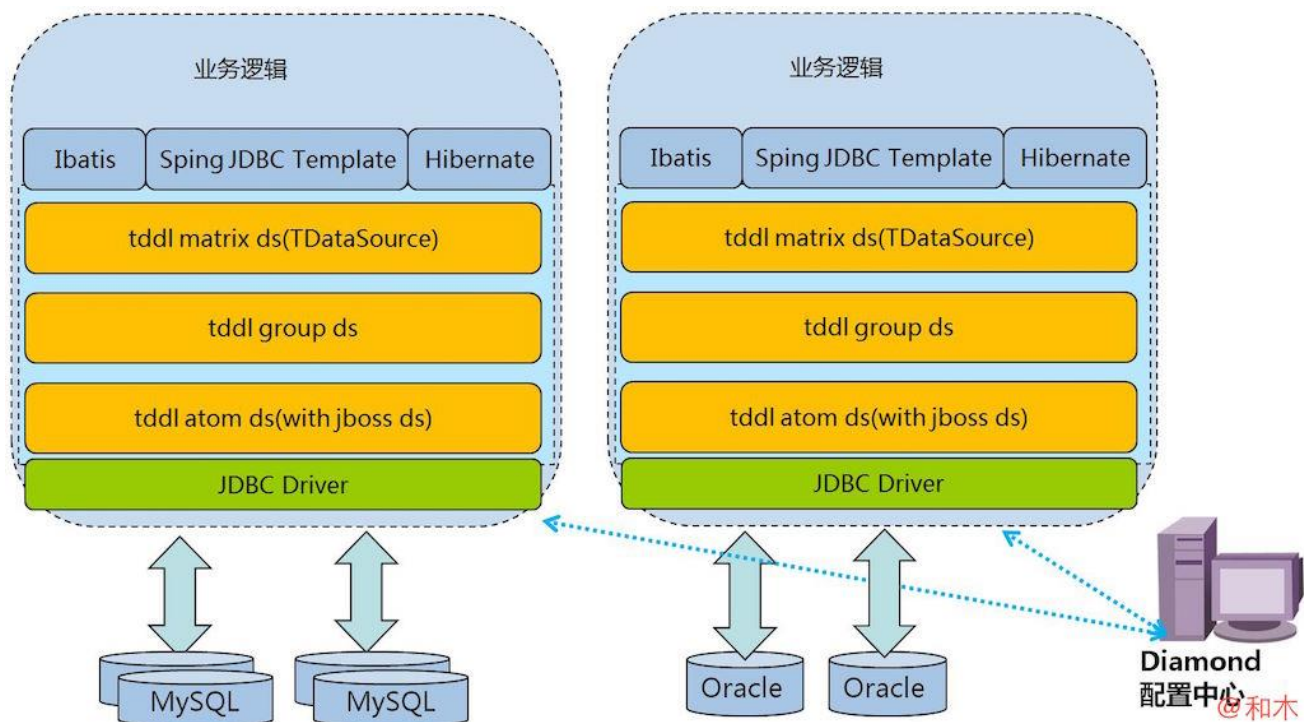
3.1 MyCAT 和 TDDL、Amoeba、Cobar 的架构比较

分布式数据库中间件 TDDL、Amoeba、Cobar、MyCAT 架构比较

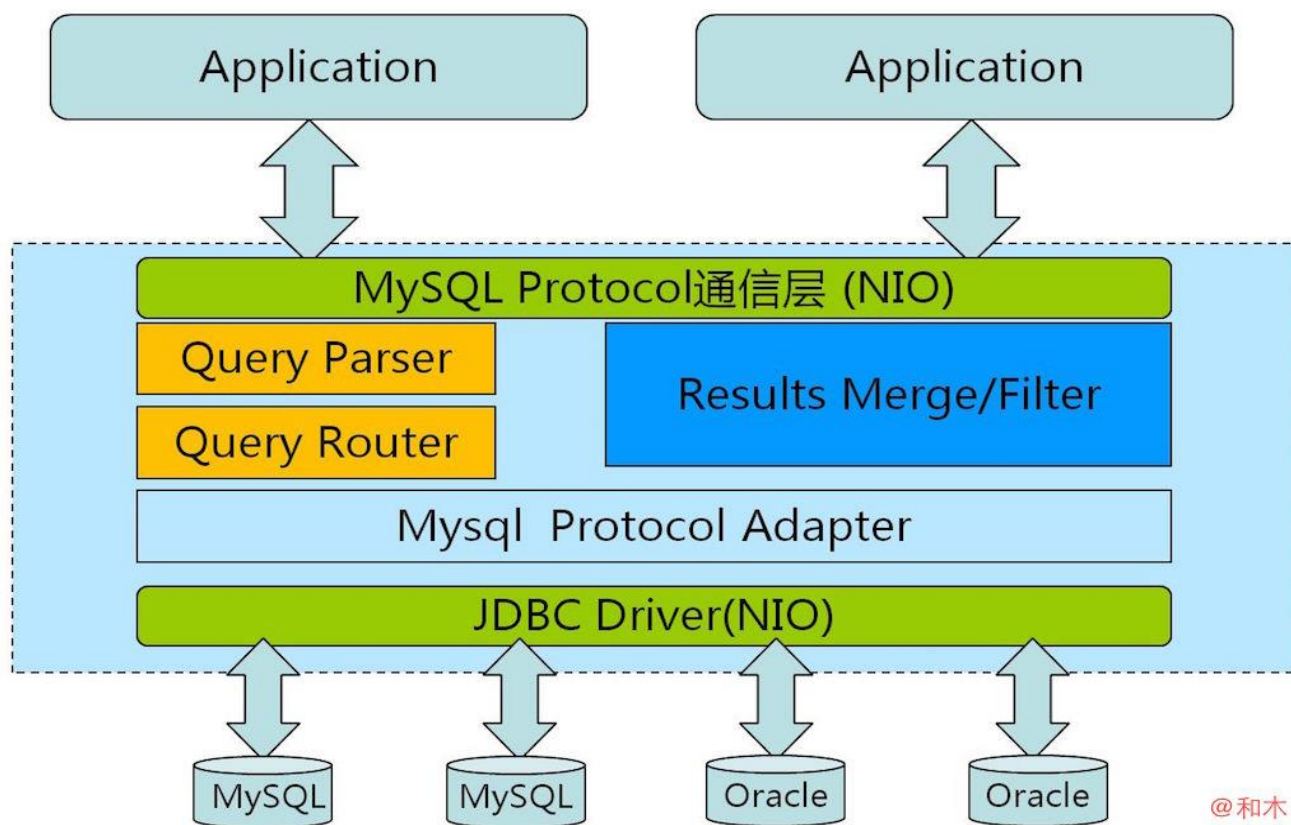
比较了业界流行的 MySQL 分布式数据库中间件，关于每个产品的介绍，网上的资料比较多，本文只是对几款产品的架构进行比较，从中可以看出中间件发展和演进路线。

3.2 框架比较

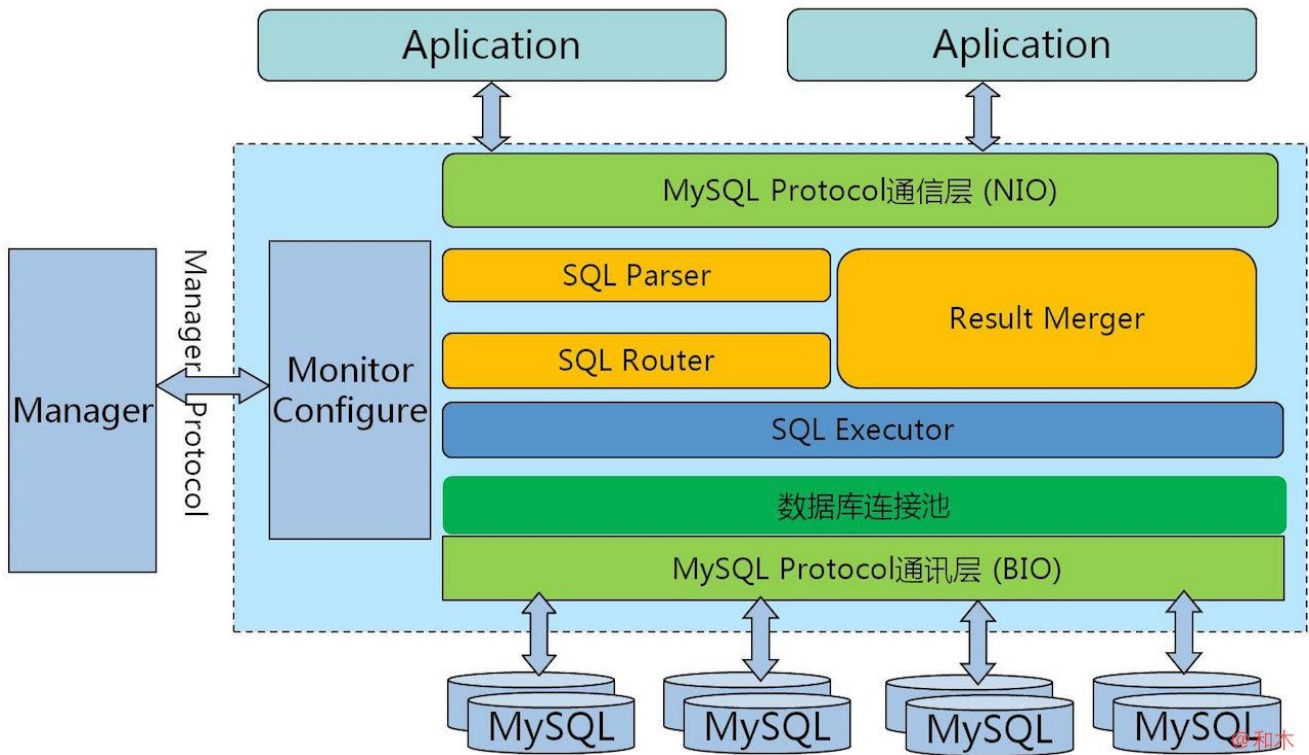
3.2.1 TDDL



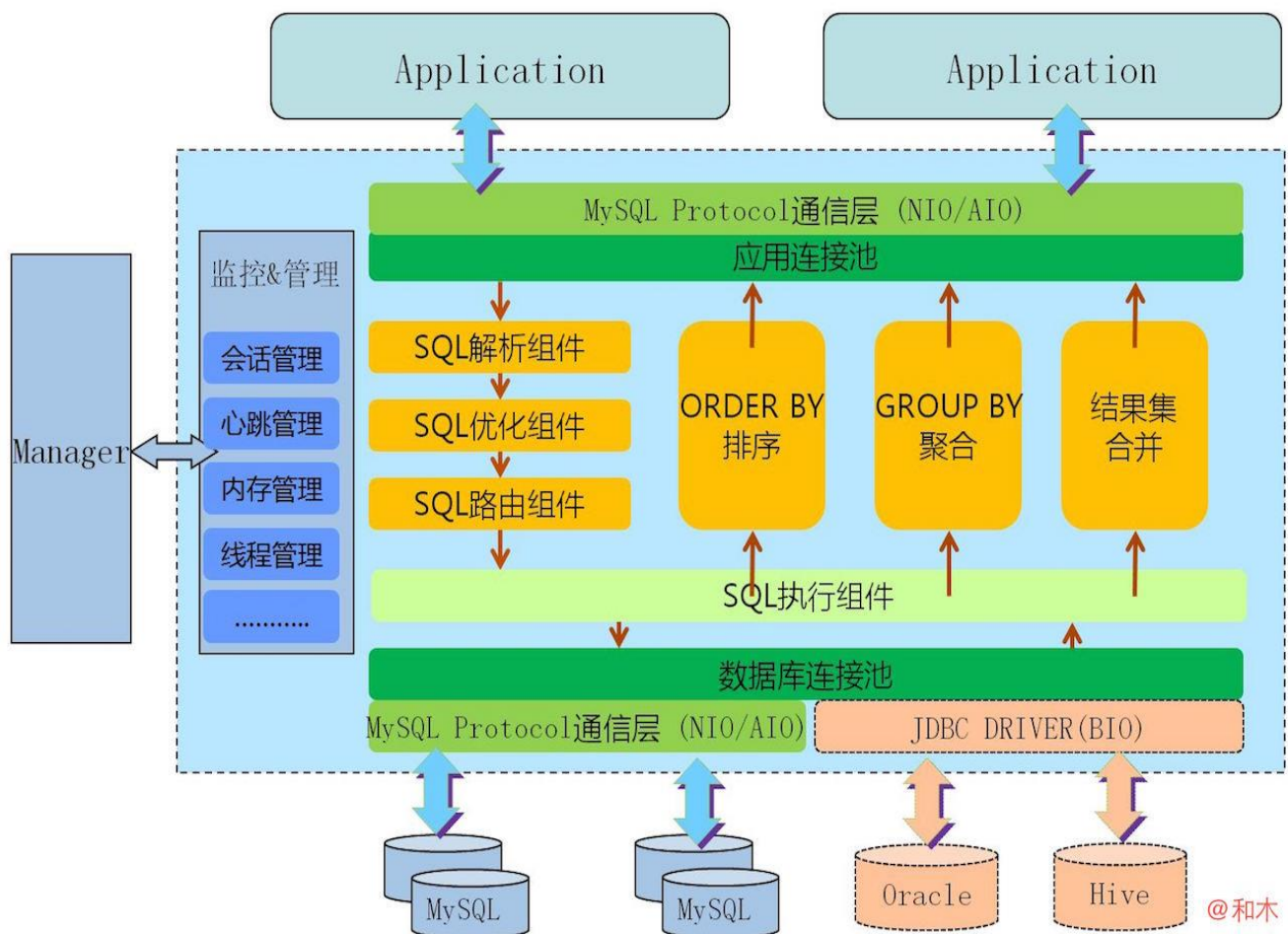
3.2.2 Amoeba



3.2.3 Cobar



3.2.4 MyCat



3.3 点评

1. TDDL 不同于其它几款产品，并非独立的中间件，只能算作中间层，是以 Jar 包方式提供给应用调用。属于 JDBC Shard 的思想，网上也有很多其它类似产品。

2. 另外，网上有关于 TDDL 的图，如 <http://www.tuicool.com/articles/nmeuu2> 中的图 1-2 TDDL 所处领域模型定位，**把 TDDL 画在 JDBC 下层了，这个是不对的，正确的位置是 TDDL 夹在业务层和 JDBC 中间**

3. Amoeba 是作为一个真正的独立中间件提供服务，即应用去连接 Amoeba 操作 MySQL 集群，就像操作单个 MySQL 一样。从架构中可以看来，Amoeba 算中间件中的早期产品，后端还在使用 JDBC Driver。

4. Cobar 是在 Amoeba 基础上进化的版本，一个显著变化是把后端 JDBC Driver 改为原生的 MySQL 通信协议层。后端去掉 JDBC Driver 后，意味着不再支持 JDBC 规范，不能支持 Oracle、PostgreSQL 等数据。但使用原生通信协议代替 JDBC Driver，后端的功能增加了很多想象力，比如主备切换、读写分离、异步操作等。

MyCat 又是在 Cobar 基础上发展的版本，两个显著点是：

- 后端由 BIO 改为 NIO，并发量有大幅提高。
- 增加了对 Order By、Group By、limit 等聚合功能的支持（虽然 Cobar 也可以支持 Order By、Group By、Limit 语法，但是结果没有进行聚合，只是简单返回给前端，聚合功能还是需要业务系统自己完成）。

3.3.1 目前社区情况：

TDDL 处于停滞状态 。

Amoeba 处于停滞状态 。

Cobar 处于停滞状态 。

MyCAT 社区非常活跃。

3.3.2 感想

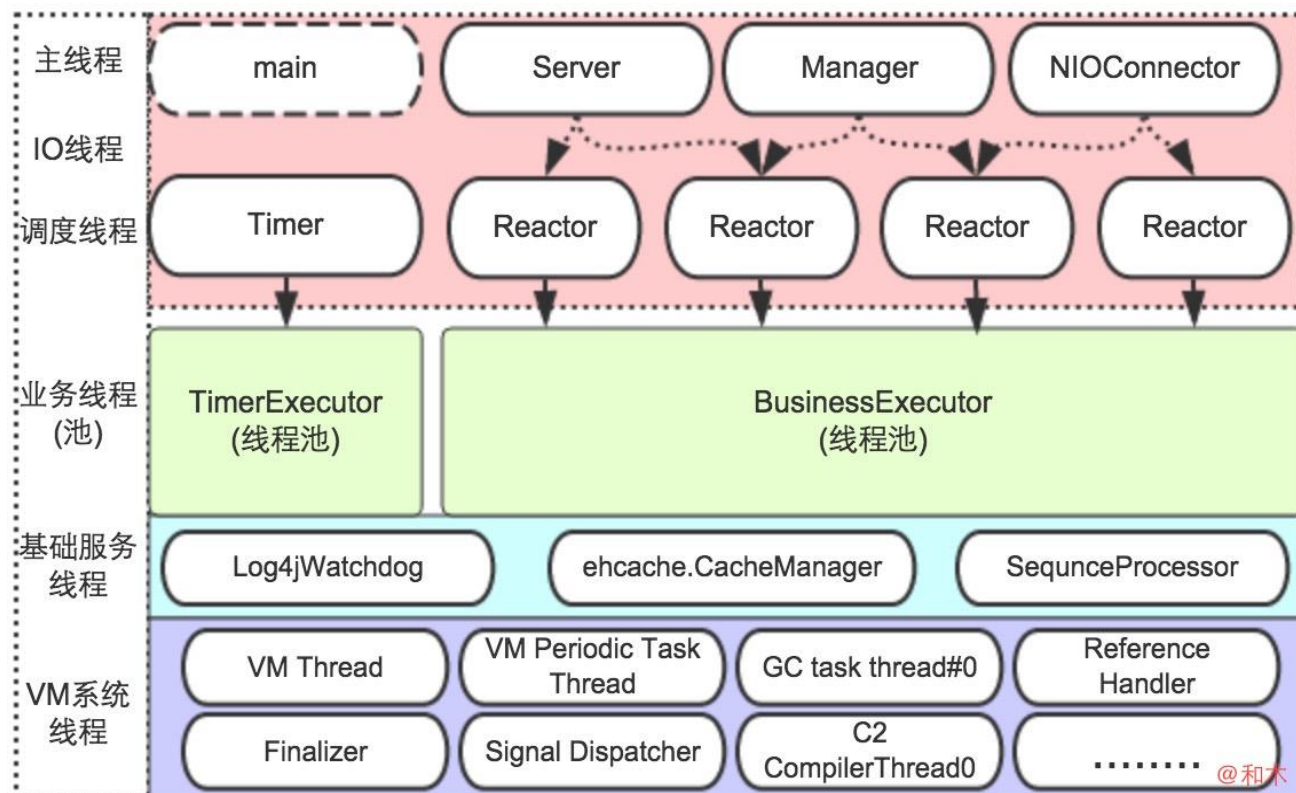
抛开 TDDL 不说，Amoeba、Cobar、MyCAT 这三者的渊源比较深，若 Amoeba 能继续下去，Cobar 就不会出来；若 Cobar 那批人不是都走光了的话，MyCAT 也不会再另起炉灶。所以说，在中国开源的项目很多，但是能坚持下去的非常难，MyCAT 社区现在非常活跃，也真是一件蛮难得的事。

3.4 其它资料

这个博客把几款产品的资料汇总在一起，倒也省得大家在网上到处搜了。mysql 中间件研究(Atlas, cobar, TDDL, mycat, heisenberg,Oceanus,vitess)

第 4 章 MyCAT 线程模型分析

4.1 MyCAT 线程模型



4.2 Mycat 线程介绍

4.2.1 Timer

Timer 单线程仅仅负责调度，任务的具体动作交给 timerExecutor。

4.2.2 TimerExecutor 线程池，

默认大小 N=2

任务通过 timer 单线程和 timerExecutor 线程池共同完成。这个 1+N 的设计方式比较巧妙！

但是 timerExecutor 跟 aioExecutor 大小默认一样，不太合理，定时任务没有这么大的运算量。

4.2.3 NIOConnect 主动连接事件分离器

一个线程，负责作为客户端连接 MySQL 的主动连接事件。

4.2.4 Server 被动连接事件分离器

一个线程，负责作为服务端接收来自业务系统的连接事件。

4.2.5 Manager 被动连接事件分离器

一个线程，负责作为服务端接收来自管理系统的连接事件。

4.2.6 NIOReactor 读写事件分离器

默认个数 $N = \text{processor size}$ ，通道建立连接后处理 NIO 读写事件。

由于写是采用通道空闲时其它线程直接写，只有写繁忙时才会注册写事件，再由 NIOReactor 分发。所以 NIOReactor 主要处理读操作。

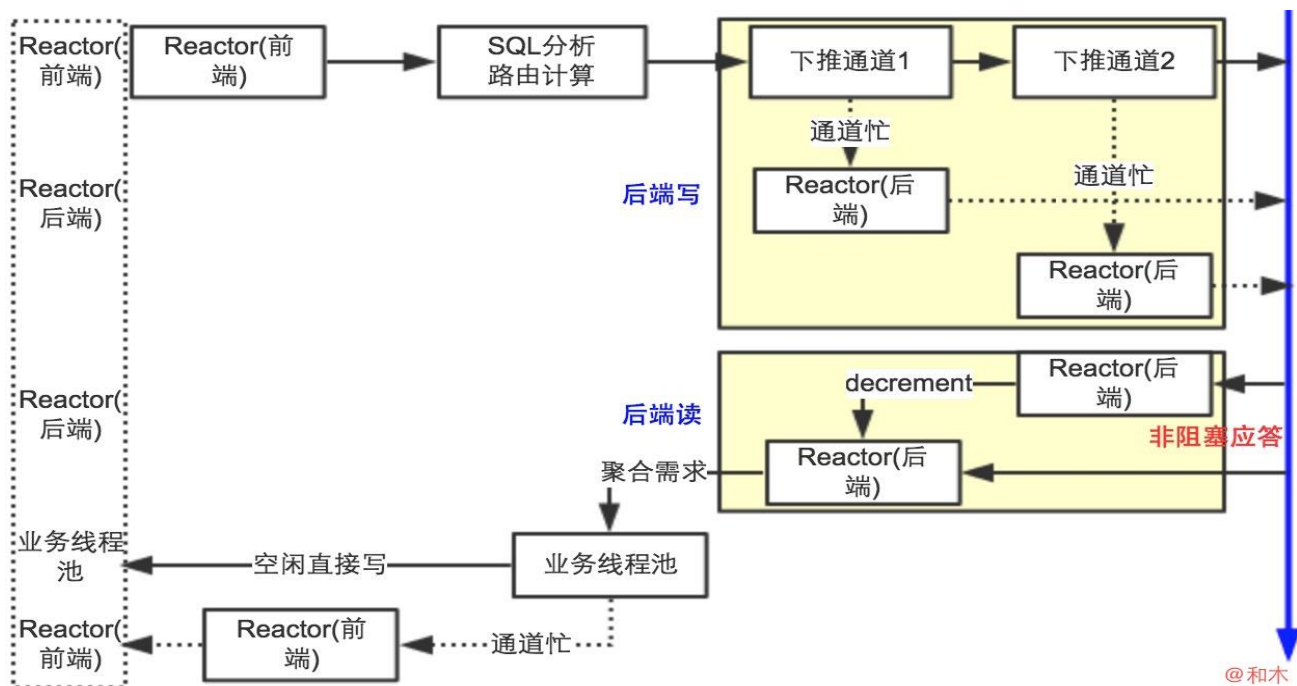
4.2.7 BusinessExecutor 线程池

默认大小 $N = \text{processor size}$ ，任务队列采用的 LinkedTransferQueue。

所有的 NIOReactor 把读出的数据交给 BusinessExecutor 做下一步的业务操作。

全局只有一个 BusinessExecutor 线程池，所有链接通道随机分成多个组，然后每组的多个通道共享一个 Reactor，所有的 Reactor 读取且解码后的数据下一步处理操作，又共享一个 BusinessExecutor 线程池。

4.2.8 一个 SQL 请求的线程切换



4.2.9 MyCAT 的线程快照

```
jstack 34179|grep prio
```

```
"Attach Listener" #32 daemon prio=9 os_prio=31 tid=0x00007f8f8ba15800 nid=0x2f07 waiting on condition
[0x0000000000000000]
```

```
"Timer1" #31 daemon prio=5 os_prio=31 tid=0x00007f8f8c0d1000 nid=0x7703 waiting on condition
[0x0000000126510000]
```

```
"Timer0" #30 daemon prio=5 os_prio=31 tid=0x00007f8f8c0d0000 nid=0x7607 waiting on condition
[0x000000012640d000]
```

```
"DestroyJavaVM" #29 prio=5 os_prio=31 tid=0x00007f8f8b01c000 nid=0x1303 waiting on condition
[0x0000000000000000]
```

```
"BusinessExecutor7" #28 daemon prio=5 os_prio=31 tid=0x00007f8f8b1e5800 nid=0x6f03 waiting on
condition [0x000000012630a000]
```

```
"BusinessExecutor6" #27 daemon prio=5 os_prio=31 tid=0x00007f8f8a3ab800 nid=0x6d03 waiting on
condition [0x0000000126207000]
```

```
"BusinessExecutor5" #26 daemon prio=5 os_prio=31 tid=0x00007f8f8a3b3000 nid=0x6b03 waiting on
condition [0x0000000126104000]
```

```
"BusinessExecutor4" #25 daemon prio=5 os_prio=31 tid=0x00007f8f89c04800 nid=0x6903 waiting on
condition [0x0000000126001000]
```

"BusinessExecutor3" #24 daemon prio=5 os_prio=31 tid=0x00007f8f89937800 nid=0x6703 waiting on condition [0x0000000125efe000]

"BusinessExecutor2" #23 daemon prio=5 os_prio=31 tid=0x00007f8f8a443800 nid=0x6503 waiting on condition [0x0000000125dfb000]

"BusinessExecutor1" #22 daemon prio=5 os_prio=31 tid=0x00007f8f8a43c000 nid=0x6303 waiting on condition [0x0000000125cf8000]

"BusinessExecutor0" #21 daemon prio=5 os_prio=31 tid=0x00007f8f8a3ae000 nid=0x6103 waiting on condition [0x0000000125bf5000]

"\$_MyCatServer" #20 prio=5 os_prio=31 tid=0x00007f8f8c098000 nid=0x5f03 runnable [0x0000000125af2000]

"\$_MyCatManager" #19 prio=5 os_prio=31 tid=0x00007f8f8a8ce800 nid=0x5d03 runnable [0x00000001259ef000]

"\$_NIOConnector" #18 prio=5 os_prio=31 tid=0x00007f8f89956800 nid=0x5b03 runnable [0x00000001256ec000]

"\$_NIOREACTOR-3-RW" #17 prio=5 os_prio=31 tid=0x00007f8f898b9000 nid=0x5903 runnable [0x00000001255e9000]

"\$_NIOREACTOR-2-RW" #16 prio=5 os_prio=31 tid=0x00007f8f8a914800 nid=0x5703 runnable [0x00000001254e6000]

"\$_NIOREACTOR-1-RW" #15 prio=5 os_prio=31 tid=0x00007f8f8a8d9800 nid=0x5503 runnable [0x00000001253e3000]

"\$_NIOREACTOR-0-RW" #14 prio=5 os_prio=31 tid=0x00007f8f8a8d9000 nid=0x5303 runnable [0x00000001252e0000]

"Log4jWatchdog" #13 daemon prio=5 os_prio=31 tid=0x00007f8f8a305000 nid=0x5107 waiting on condition [0x00000001251cd000]

"net.sf.ehcache.CacheManager@512ddf17" #11 daemon prio=5 os_prio=31 tid=0x00007f8f8a32d000 nid=0x4f03 in Object.wait() [0x00000001250ca000]

"MyCatTimer" #10 daemon prio=5 os_prio=31 tid=0x00007f8f8a162800 nid=0x4d03 in Object.wait() [0x0000000124fab000]

"Thread-0" #9 prio=5 os_prio=31 tid=0x00007f8f8b082000 nid=0x4b03 waiting on condition

[0x0000000124cf1000]

"Service Thread" #8 daemon prio=9 os_prio=31 tid=0x00007f8f8a801000 nid=0x4703 runnable

[0x0000000000000000]

"C1 CompilerThread2" #7 daemon prio=9 os_prio=31 tid=0x00007f8f8b025800 nid=0x4503 waiting on condition [0x0000000000000000]

"C2 CompilerThread1" #6 daemon prio=9 os_prio=31 tid=0x00007f8f8b025000 nid=0x4303 waiting on condition [0x0000000000000000]

"C2 CompilerThread0" #5 daemon prio=9 os_prio=31 tid=0x00007f8f8b023800 nid=0x4103 waiting on condition [0x0000000000000000]

"Signal Dispatcher" #4 daemon prio=9 os_prio=31 tid=0x00007f8f8b022000 nid=0x3017 runnable [0x0000000000000000]

"Finalizer" #3 daemon prio=8 os_prio=31 tid=0x00007f8f8a00e800 nid=0x2d03 in Object.wait() [0x0000000122b34000]

"Reference Handler" #2 daemon prio=10 os_prio=31 tid=0x00007f8f8a00d800 nid=0x2b03 in Object.wait() [0x0000000122a31000]

"VM Thread" os_prio=31 tid=0x00007f8f8b001000 nid=0x2903 runnable

"GC task thread#0 (ParallelGC)" os_prio=31 tid=0x00007f8f8980d800 nid=0x2103 runnable

"GC task thread#1 (ParallelGC)" os_prio=31 tid=0x00007f8f8980e000 nid=0x2303 runnable

"GC task thread#2 (ParallelGC)" os_prio=31 tid=0x00007f8f8980f000 nid=0x2503 runnable

"GC task thread#3 (ParallelGC)" os_prio=31 tid=0x00007f8f8980f800 nid=0x2703 runnable

"VM Periodic Task Thread" os_prio=31 tid=0x00007f8f8a840800 nid=0x4903 waiting on condition

4.3 Cobar 线程介绍

R 线程接收到读事件后解码出一个完整的 MySQL 协议包，下一步由 Handler 线程池进行 SQL 解析、路由计算。然后执行任务从 Handler 线程池转移到 Executor 线程池，以阻塞方式发送给后端 MySQL Server。Executor 收到 MySQL Server 应答后，会由最后一个 Executor 线程进行聚合，然后交给 W 线程。

4.3.7 W 线程

W 线程不停遍历 LinkedBlockingQueue 检查是否有写任务，若有则写入 Socket Channel。当 Channel 繁忙时，W 线程会注册 OP_WRITE 事件，通过 R 线程进行候补写操作。

4.3.8 ManageExecutor 线程池

Cobar 对来自 Manager 的请求和来自 Server 的请求做了分离，来自管理系统的请求，专门由 ManageExecutor 线程池处理。

4.3.9 InitExecutor 线程池

用来进行后端链路初始化。

4.4 Cobar 为什么那么多个线程池？

可以发现 Cobar 有下面这么多个线程池：

- TimerExecutor 线程池(一个)。
- InitExecutor 线程池(一个)。
- ManageExecutor 线程池(一个)。
- Handler 线程池(N 个)。
- Executor 线程池(N 个)。

注意上面的**个数单位是线程池，不是线程！**所以看起来有些眼花缭乱吧？

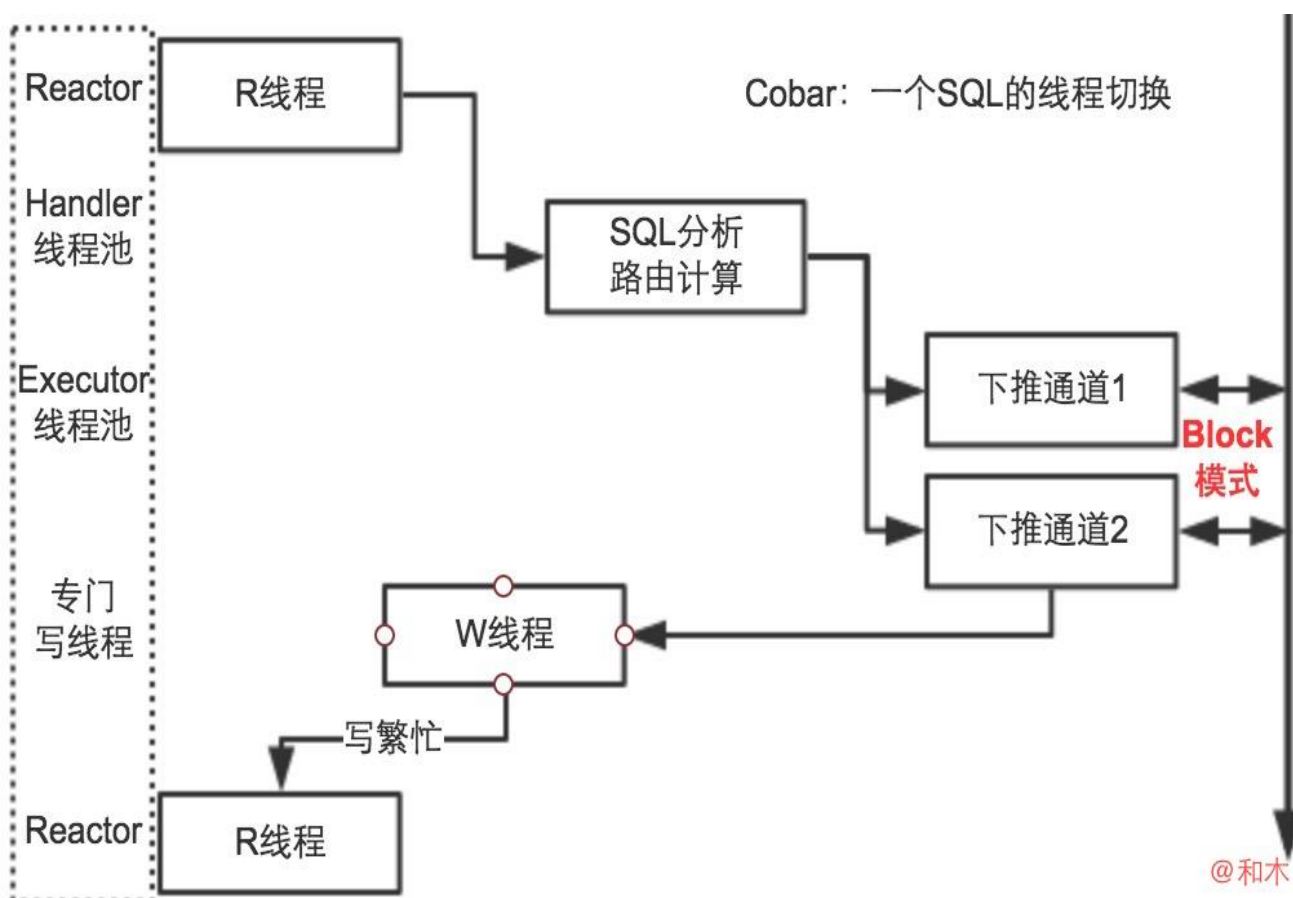
我不是 Cobar 的原作者，只能猜测最什么设计这么多线程池？那就是**因为后端采用了 BIO！**

- 因为后端 BIO，所以每一个请求到后端查询，都要阻塞一个线程，前端 NIO(Reactor-R 线程)必须要把执行任务交给 Executor 线程池。
- 由于存在聚合要求，前端 NIO 的一个 SQL 请求可能会对应多个后端请求，所以不只要阻塞一个 Executor 线程。为此增加了 Handler 做中间 SQL 解析、路由计算，路由计算完毕后再交给 Executor 执行。

- 由于后端是阻塞方式，在时，会导致 Executor 无空闲线程，为了避免管理端口输入命令无任何响应的现象，为此增加一个 ManageExecutor 线程池，专门处理 ManageExecutor 线程。
- 在后端 BIO 时，除了读写是阻塞方式外，链路建立过程也是阻塞方式，若同时链路建立请求多，也会阻塞大量线程。为避免业务、管理的相互干扰，为此增加了一个 InitExecutor 线程池专门做后端链路建立。
- 所以如果后端 BIO 改为 NIO，并优化逻辑执行过程，避免线程 sleep 或长时间阻塞，尽量通过 Reactor 直接计算，就可以大大降低线程上下文切换的损耗，上述各眼花缭乱的线程池就可以合并为一个业务线程池。

4.4.1 一个 SQL 请求的线程切换

下面是一个 SQL 请求执行过程的线程切换，可以看到 Cobar 的线程上下文切换还是比较多的。



4.4.2 Cobar 的线程快照

```
Cobar>jstack 10631|grep prio
```

```
"Processor0-E6" daemon prio=5 tid=7f931f057000 nid=0x11abcf000 waiting on condition [11abce000]
"Processor1-E6" daemon prio=5 tid=7f931f056000 nid=0x11aacc000 waiting on condition [11aacb000]
"TimerExecutor3" daemon prio=5 tid=7f931e206000 nid=0x119d22000 waiting on condition [119d21000]
"CobarServer" prio=5 tid=7f931d961000 nid=0x119c1f000 runnable [119c1e000]
```

"CobarManager" prio=5 tid=7f931f150800 nid=0x119b1c000 runnable [119b1b000]

"TimerExecutor2" daemon prio=5 tid=7f931d8c7800 nid=0x119a19000 waiting on condition [119a18000]

"TimerExecutor1" daemon prio=5 tid=7f931f14f800 nid=0x119916000 waiting on condition [119915000]

"InitExecutor1" daemon prio=5 tid=7f931f156800 nid=0x119813000 waiting on condition [119812000]

"InitExecutor0" daemon prio=5 tid=7f931f155800 nid=0x119710000 waiting on condition [11970f000]

"CobarConnector" prio=5 tid=7f931e203800 nid=0x11960d000 runnable [11960c000]

"TimerExecutor0" daemon prio=5 tid=7f931e201000 nid=0x11950a000 waiting on condition [119509000]

"Processor1-W" prio=5 tid=7f931d8c4800 nid=0x119407000 waiting on condition [119406000]

"Processor1-R" prio=5 tid=7f931d82c800 nid=0x119304000 runnable [119303000]

"Processor0-W" prio=5 tid=7f931d0ab800 nid=0x119201000 waiting on condition [119200000]

"Processor0-R" prio=5 tid=7f931d0aa800 nid=0x1190fe000 runnable [1190fd000]

"CobarTimer" daemon prio=5 tid=7f931e17f000 nid=0x118fde000 in Object.wait() [118fdd000]

"Low Memory Detector" daemon prio=5 tid=7f931e0ab800 nid=0x118b3b000 runnable [00000000]

"C2 CompilerThread1" daemon prio=9 tid=7f931e0aa800 nid=0x118a38000 waiting on condition [00000000]

"C2 CompilerThread0" daemon prio=9 tid=7f931e0aa000 nid=0x118935000 waiting on condition [00000000]

"Signal Dispatcher" daemon prio=9 tid=7f931e0a9000 nid=0x118832000 runnable [00000000]

"Surrogate Locker Thread (Concurrent GC)" daemon prio=5 tid=7f931e0a8800 nid=0x11872f000 waiting on condition [00000000]

"Finalizer" daemon prio=8 tid=7f931f037000 nid=0x116d52000 in Object.wait() [116d51000]

"Reference Handler" daemon prio=10 tid=7f931f036000 nid=0x116c4f000 in Object.wait() [116c4e000]

"VM Thread" prio=9 tid=7f931e094800 nid=0x116b4c000 runnable

"Gang worker#0 (Parallel GC Threads)" prio=9 tid=7f931f001800 nid=0x113005000 runnable

"Gang worker#1 (Parallel GC Threads)" prio=9 tid=7f931d001000 nid=0x113108000 runnable

"Gang worker#2 (Parallel GC Threads)" prio=9 tid=7f931d001800 nid=0x11320b000 runnable

"Gang worker#3 (Parallel GC Threads)" prio=9 tid=7f931d002000 nid=0x11330e000 runnable

"Concurrent Mark-Sweep GC Thread" prio=9 tid=7f931f002000 nid=0x1167c7000 runnable

"VM Periodic Task Thread" prio=10 tid=7f931d811800 nid=0x118c3e000 waiting on condition

"Exception Catcher Thread" prio=10 tid=7f931f001000 nid=0x10ff01000 runnable

4.5 MyCAT 与 Cobar 的比较

4.5.1 MyCAT 比 Cobar 减少了线程切换

Cobar 的后端采用 BIO 通信，后端读与后端写因为线程阻塞了，不存在线程切换，没有可比性，所以我们只比较 NIO 和业务逻辑部分。

Cobar 的线程模型中存在着大量的上下文切换,MyCAT 的线程调度尽量减少了线程间的切换，以写为例

Cobar 是业务线程先把写请求交给专门的 W 线程，W 线程再写过程中发现通道繁忙时再交给 R 线程；

MyCAT 对写的做法是业务线程发现通道空闲直接写，只有在通道繁忙时再交给 Reactor 线程。

4.5.2 减少线程切换与业务可能停顿的矛盾

MyCAT 几乎已经达到了线程简化的最高境界，有一个看似可行的方法：可以配置多个 NIOReactor，尽可能所有读、解码、业务处理都在 Reactor 线程中完成，而不必把任务交给 BusinessExecutor 线程池，从而减少线程的上下文切换，提高处理效率。

但是，不管配置几个 Reactor，还是要求多个通道共享一个 Reactor，（为什么？因为 Reactor 最多十几个、几十个，并发的链接通道可能上万个！）如果 Reactor 在读和解码请求后顺序处理业务逻辑，那么在处理业务逻辑过程中，Reactor 就无法响应其它通道的事件了，这个时候如果正好有共享同一个 Reactor 的其它通道的请求过来，就会出现停顿的现象。

那么如何做呢，就需要具体问题具体分析，要对业务逻辑进行归类：

- 对于业务较重的，比如大结果集排序，则送到 BusinessExecutor 线程池进行下一步处理；
- 于业务较轻的，比如单库直接转发的情况，则由 Reactor 直接完成，不再送线程池，减少上下文切换。

4.5.3 特别说明 ER 分片机制

如果涉及到 ER 分片，MyCAT 目前的机制：计算路由时以阻塞同步方式调用

FetchStoreNodeOfChildTableHandler，若由 Reactor 直接进行路由计算，会导致其它通道停顿现象。把 ER 分片同步改异步是个看似可行的方法，但这个改造工作量较大，会造成原来完整路由计算逻辑的碎片化。

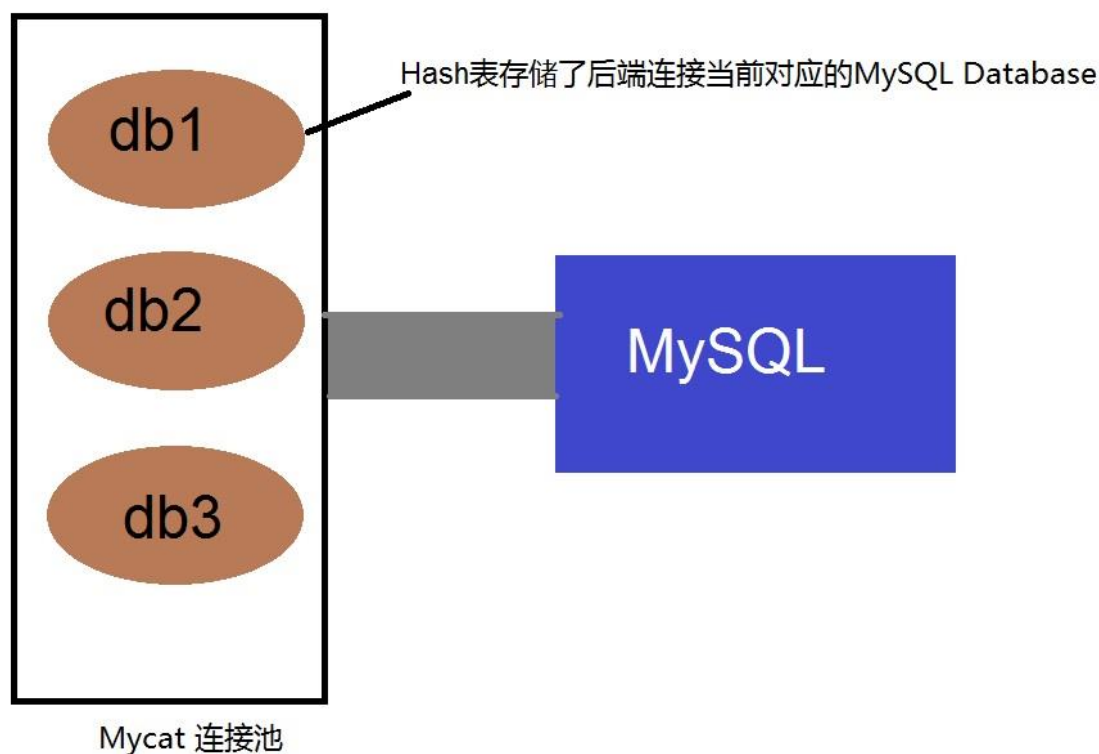
即使 ER 分片同步改异步了，每次子表操作都要遍历父表对性能损耗较大，即使采用缓存也不能最终解决问题。个人觉得，ER 分片这个功能比较鸡肋，建议生产部署时绕开这个功能，直接通过关联字段分片或表设计时增加冗余字段。

4.5.4 数据验证

1. 测试 sql 从收到请求到下推的总时长，如果时间可容忍，则不必切换到线程池。**忽略 ER 分片。**
2. 对于 manager 端口的命令，若存在执行时间比较的，也需要改为线程池来执行。
3. 对于收到的应答，大部分都不必切换到线程池。
4. 对于大量数据排序，只有在排序时，构造执行任务，切换到线程池完成。

第 5 章 mycat 的连接池模型

Mycat 为了最高效的利用后端的 MySQL 连接，采取了不同于 Cobar 也不同于传统 JDBC 连接池的做法，传统的做法是基于 Database 的连接池，即一个 MySQL 服务器上有 5 个 Database，则每个 Database 独占最大 200 个连接。这种模式的最大问题在于，将一个数据库所具备的最大 1000 个连接，隔离成了更新小的连接池，于是可能产生一个应用的连接不够，但其他应用的连接却很空闲的资源浪费情况，而对于分片这种场景，这个缺陷则几乎是致命的，因为每个分片所对应的 Database 的连接数量被限制在了一个很小的范围内，从而导致系统并发能力的大幅降低。而 Mycat 则采用了基于 MySQL 实例的连接池模式，每个 Database 都可以用现有的 1000 个连接中的空闲连接。



代码解读

在 Mycat 的连接池里，当前可用的 MySQL 连接是放到一个 HashMap 的数据结构里，Key 为当前连接对应的 Database，另外还有二级分类，即按照连接是自动提交还是手动提交模式进行区分，这个设计是为了高效的查询匹配的可用连接，具体逻辑如下：

当某个用户会话需要一个自动提交的，到分片 dn1(对应 db1) 的 SQL 连接的时候，连接池首先找是否有 db1 上的可用连接，如果有，看是否有自动提交模式的连接，找到就返回，否则返回 db1 上的手动提交模式的连接，若没有 db1 的可用连接，则随机返回一个其他 db 对应的可用连接，若没有可用连接，并且连接池还没达到上限，则创建一个新连接并返回，这个逻辑过程，我们会发现，用户会话得到的连接可能不是他原先想要的，比如 Database 不对应，或者事务模式不匹配，因此在执行具体的 SQL 之前，还有一个自动同步数据库连接的过程，包括事务隔离级别、事务模式、字符集、Database 等四个指标，同步完成以后，才会执行具体的 SQL 指令。

io.mycat.backend 目录下包括连接池相关的代码，其中：

PhysicalDBNode 是 Mycat 分片 (Datanode) 的对应，引用一个连接池对象 PhysicalDBPool，PhysicalDBPool 里面引用了真正的连接池对象 PhysicalDatasource，并且按照读节点和写节点分开引用，实现读写分类和节点切换的功能，其中 activedIndex 属性表明了当前是哪个写节点的数据源在生效。连接池对象连接池对象 PhysicalDatasource 里最重要的数据结构是 ConMap，它里面存储有当前的可用连接，它的关键代码如下：

```
public class ConMap {  
    // key -schema  
    private final ConcurrentHashMap<String, ConQueue> items = new ConcurrentHashMap<String,  
ConQueue>();  
  
    public ConQueue getSchemaConQueue(String schema) {  
        ConQueue queue = items.get(schema);  
        if (queue == null) {  
            ConQueue newQueue = new ConQueue();  
            queue = items.putIfAbsent(schema, newQueue);  
            return (queue == null) ? newQueue : queue;  
        }  
        return queue;  
    }  
}
```

```
}
```

```
public BackendConnection tryTakeCon(final String schema, boolean autoCommit) {  
    final ConQueue queue = items.get(schema);  
    BackendConnection con = tryTakeCon(queue, autoCommit);  
    if (con != null) {  
        return con;  
    } else {  
        for (ConQueue queue2 : items.values()) {  
            if (queue != queue2) {  
                con = tryTakeCon(queue2, autoCommit);  
                if (con != null) {  
                    return con;  
                }  
            }  
        }  
    }  
    return null;  
}
```

```
private BackendConnection tryTakeCon(ConQueue queue, boolean autoCommit) {  
  
    BackendConnection con = null;  
    if (queue != null && ((con = queue.takeIdleCon(autoCommit)) != null)) {  
        return con;  
    } else {  
        return null;  
    }  
}
```

```
}
```

tryTakeCon 是获取一个可用连接，代码的逻辑中，首先看对应的 Database 上是否有可用连接，如果有就立即返回，否则从其他的 Dabase 上找一个可用连接返回。

MySQLConnection 类为具体的 MySQL Native 连接对象，synAndDoExecute 方法则判断获取到的连接是否符合要求，若不符合要求，先同步状态，然后执行具体的 SQL。

```
...
```

```
private void synAndDoExecute(String xaTxID, RouteResultsetNode rrn,
int clientCharSetIndex, int clientTxIsolation,
boolean clientAutoCommit) {
String xaCmd = null;

    boolean conAutoComit = this.autocommit;
String conSchema = this.schema;
// never executed modify sql,so auto commit
boolean expectAutocommit = !modifiedSQLExecuted || isFromSlaveDB()
    || clientAutoCommit;
if (expectAutocommit == false && xaTxID != null && xaStatus == 0) {
    clientTxIsolation = Isolations.SERIALIZABLE;
    xaCmd = "XA START " + xaTxID + ';;'

}

int schemaSyn = conSchema.equals(oldSchema) ? 0 : 1;
int charsetSyn = (this.charsetIndex == clientCharSetIndex) ? 0 : 1;
int txIsolationSyn = (txIsolation == clientTxIsolation) ? 0 : 1;
int autoCommitSyn = (conAutoComit == expectAutocommit) ? 0 : 1;
int synCount = schemaSyn + charsetSyn + txIsolationSyn + autoCommitSyn;
if (synCount == 0) {
```

```

// not need syn connection

sendQueryCmd(rrn.getStatement());

return;
}

CommandPacket schemaCmd = null;

StringBuilder sb = new StringBuilder();

if (schemaSyn == 1) {

    schemaCmd = getChangeSchemaCommand(conSchema);

    // getChangeSchemaCommand(sb, conSchema);
}


if (charsetSyn == 1) {

    getCharsetCommand(sb, clientCharSetIndex);
}

if (txIsolationSyn == 1) {

    getTxIsolationCommand(sb, clientTxIsolation);
}

if (autoCommitSyn == 1) {

    getAutocommitCommand(sb, expectAutocommit);
}

if (xaCmd != null) {

    sb.append(xaCmd);
}

if (LOGGER.isDebugEnabled()) {

    LOGGER.debug("con need syn ,total syn cmd " + synCount

        + " commands " + sb.toString() + "schema change:"

        + (schemaCmd != null) + " con:" + this);
}

metaDataSyned = false;

```

```

statusSync = new StatusSync(xaCmd != null, conSchema,
    clientCharSetIndex, clientTxIsoLation, expectAutocommit,
    synCount);
// syn schema
if (schemaCmd != null) {
    schemaCmd.write(this);
}
// and our query sql to multi command at last
sb.append(rrn.getStatement());
// syn and execute others
this.sendQueryCmd(sb.toString());
// waiting syn result...
}

```

...

通过共享一个 MySQL 上的所有物理连接，并结合连接状态同步的特性，MyCAT 的连接池做到了最佳的吞吐量，也在一定程度上提升了整个系统的并发支撑能力。

第 6 章 Mycat 的网络通信框架

6.1 先从一个测试说起

某小组对 Cobar 和 MyCAT 做了一个简单的比较测试，过程如下

6.1.1 测试环境

利用 A、B、C 三大类服务器，在 A 台上面安装配置 MyCAT 及 Cobar，这样保证了硬件方面的一致性。B 类服务器上安装 Apache 这一 web 服务，使用 PHP 语言。C 类安装 MySQL 数据库，其中 B 类与 C 类均不止一台，主要目的是为了作压力的均分。C 类服务器安装了 4 台，存放了相同的数据库，对其中一个表进行分片存储。

测试软件使用的是 loadRunner。在对两个中间件分别进行测试的过程中，采用的 web 服务器执行页面及相关数据库，均未调整，仅在中间件上有分别。

	MyCAT	Cobar																																										
	一样都是模拟1500个用户。同时启动运行20分钟。																																											
全局计划	全局计划 总数: 1500 个 Vuser <table><tr><th>操作</th><th>属性</th></tr><tr><td>初始化</td><td>在每个 Vuser 运行之前将其初始化</td></tr><tr><td>启动 Vuser</td><td>同时启动 1500 个 Vuser</td></tr><tr><td>▶ 持续时间</td><td>运行00:20:00 (HH:MM:SS)</td></tr><tr><td>停止 Vuser</td><td>同时停止 全部 个 Vuser</td></tr></table>		操作	属性	初始化	在每个 Vuser 运行之前将其初始化	启动 Vuser	同时启动 1500 个 Vuser	▶ 持续时间	运行00:20:00 (HH:MM:SS)	停止 Vuser	同时停止 全部 个 Vuser																																
	操作	属性																																										
	初始化	在每个 Vuser 运行之前将其初始化																																										
	启动 Vuser	同时启动 1500 个 Vuser																																										
	▶ 持续时间	运行00:20:00 (HH:MM:SS)																																										
停止 Vuser	同时停止 全部 个 Vuser																																											
场景状态结果	<table><tr><th>场景状态</th><th>关闭</th><th></th></tr><tr><td>运行 Vuser</td><td>0</td><td></td></tr><tr><td>已用时间</td><td>00:21:59 (hh:mm:ss)</td><td></td></tr><tr><td>每秒点击次数</td><td>0.00 (最后 60 秒)</td><td></td></tr><tr><td>通过的事务</td><td>27544</td><td></td></tr><tr><td>失败的事务</td><td>45731</td><td></td></tr><tr><td>错误</td><td>45733</td><td></td></tr></table>	场景状态	关闭		运行 Vuser	0		已用时间	00:21:59 (hh:mm:ss)		每秒点击次数	0.00 (最后 60 秒)		通过的事务	27544		失败的事务	45731		错误	45733		<table><tr><th>场景状态</th><th>关闭</th><th></th></tr><tr><td>运行 Vuser</td><td>0</td><td></td></tr><tr><td>已用时间</td><td>00:29:38 (hh:mm:ss)</td><td></td></tr><tr><td>每秒点击次数</td><td>0.00 (最后 60 秒)</td><td></td></tr><tr><td>通过的事务</td><td>2998</td><td></td></tr><tr><td>失败的事务</td><td>613041</td><td></td></tr><tr><td>错误</td><td>613726</td><td></td></tr></table>	场景状态	关闭		运行 Vuser	0		已用时间	00:29:38 (hh:mm:ss)		每秒点击次数	0.00 (最后 60 秒)		通过的事务	2998		失败的事务	613041		错误	613726	
	场景状态	关闭																																										
运行 Vuser	0																																											
已用时间	00:21:59 (hh:mm:ss)																																											
每秒点击次数	0.00 (最后 60 秒)																																											
通过的事务	27544																																											
失败的事务	45731																																											
错误	45733																																											
场景状态	关闭																																											
运行 Vuser	0																																											
已用时间	00:29:38 (hh:mm:ss)																																											
每秒点击次数	0.00 (最后 60 秒)																																											
通过的事务	2998																																											
失败的事务	613041																																											
错误	613726																																											
命令响应情况	正常	<pre>mysql> use dbtest; Database changed mysql> show cobar_status; +-----+ ! STATUS ! +-----+ ! ON ! +-----+ 1 row in set (0.00 sec) mysql> show cobar_cluster; Empty set (0.00 sec) mysql></pre>																																										

表格中场景状态下，明显 MyCAT 通过事务达到 27544 个，而 Cobar 只有 2998，原因应该是 Cobar 假死之后对相关请求处理，均不再响应。

另外 Cobar 的内存直接上到 300, 000KB 以上，手动使用页面对测试实例连接单独访问访问不了，涉及到测试表的所有操作均不能再操作。Cobar 内部使用 show cobar_status;命令回馈正常。但是使用 show cobar_cluster;命令，cobar 反馈不了 cobar 的节点信息，而是返回 empty set。

测试过程中 MyCAT 行为正常。

Cobar 存在上述致命问题的原因是后端采用了 BIO，每个请求在等待应答时都会占用一个线程，当前端并发量大时，就产生了假死的现象。

MyCAT 对 Cobar 的网络框架进行了重构，后端 BIO 改为 AIO 和 NIO，同时还做了其它方面的优化，下面就慢慢道来。

6.2 MyCAT 网络框架

6.2.1 三种 IO 类型

系统 I/O 可分为阻塞型、非阻塞同步型以及非阻塞异步型。

阻塞型 I/O 意味着控制权直到调用操作结束了才会回到调用者手里。结果调用者被阻塞了，这段时间了做不了任何其它事情。更郁闷的是，在等待 IO 结果的时间里，调用者所在线程此时无法腾出手来去响应其它的请求，这真是太浪费资源了。拿 `read()` 操作来说吧，调用此函数的代码会一直僵在此处直至它所读的 socket 缓存中有数据到来。

相比之下，非阻塞同步是会立即返回控制权给调用者的。调用者不需要等等，它从调用的函数获取两种结果：要么此次调用成功进行了；要么系统返回错误标识告诉调用者当前资源不可用，你再等等或者再试度看吧。比如 `read()` 操作，如果当前 socket 无数据可读，则立即返回 `EWOULDBLOCK/EAGAIN`，告诉调用 `read()` 者“数据还没准备好，你稍后再试”。

在非阻塞异步调用中，稍有不同。调用函数在立即返回时，还告诉调用者，这次请求已经开始了。系统会使用另外的资源或者线程来完成这次调用操作，并在完成的时候知会调用者（比如通过回调函数）。拿 Windows 的 `ReadFile()` 或者 POSIX 的 `aio_read()` 来说，调用它之后，函数立即返回，操作系统在后台同时开始读操作。

在以上三种 IO 形式中，理论上，非阻塞异步是性能最高、伸缩性最好的。

同步和异步是相对于应用和内核的交互方式而言的，同步需要主动去询问，而异步的时候内核在 IO 事件发生的时候通知应用程序，而阻塞和非阻塞仅仅是系统在调用系统调用的时候函数的实现方式而已。

对于 JAVA 的 API 来说：

- `java.net.Socket` 就是典型的阻塞型 IO。
- `java NIO` 非阻塞同步。
- `java AIO` 非阻塞异步。

MyCAT 起源于 Cobar，Cobar 前端为 NIO 后端为 BIO，后端就是通过 `java.net.Socket` 进行读写，所以 Cobar 后端每次进行读写都会造成线程阻塞，后端能支持的连接总数就成为瓶颈所在。

MyCAT 在基于 Cobar 改版时，直接采用了 Java 7 的 AIO，前后端都实现了非阻塞异步。由于 Linux 并没有真正实现 AIO，实际测试下来，AIO 并不比 NIO 快，反而性能上比 NIO 还要慢。所以 MyCAT 在 2014 年下半年，做了一次网络通信框架的大调整，改为同时支持 AIO 和 NIO，通过启动参数让用户来选择哪种方式。虽然现在 AIO 比 NIO 慢，但是 MyCAT 仍然保留了 AIO 实现，就是为了等 Linux 真正实现 AIO 后，可以直接支持。

6.2.2 Reactor 和 Proactor

MyCAT 同时实现了 NIO 和 AIO，为了便于读者更清楚理解代码实现，先介绍 NIO 和 AIO 分布对应的两种设计模式：Reactor 和 Proactor。

一般情况下，I/O 复用机制需要事件分享器(event demultiplexor)。事件分享器的作用，即将那些读写事件源分发给各读写事件的处理者，就像送快递的在楼下喊：谁的什么东西送了，快来拿吧。开发人员在开始的时候需要在分享器那里注册感兴趣的事件，并提供相应的处理者(event handlers)，或者是回调函数；事件分享器在适当的时候会将请求的事件分发给这些 handler 或者回调函数。

涉及到事件分享器的两种模式称为：Reactor 和 Proactor。Reactor 模式是基于同步 I/O 的，而 Proactor 模式是和异步 I/O 相关的。在 Reactor 模式中，事件分离器等待某个事件或者应用或操作的状态发生（比如文件描述符可读写，或者是 socket 可读写），事件分离器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。

而在 Proactor 模式中，事件处理者(或者代由事件分离器发起)直接发起一个异步读写操作(相当于请求)，而实际的工作是由操作系统来完成的。发起时，需要提供的参数包括用于存放读到数据的缓存区，读的数据大小，或者用于存放外发数据的缓存区，以及这个请求完后的回调函数等信息。事件分离器得知了这个请求，它默默等待这个请求的完成，然后转发完成事件给相应的事件处理者或者回调。举例来说，在 Windows 上事件处理者投递了一个异步 IO 操作(称有 overlapped 的技术)，事件分离器等 IOCompletion 事件完成。这种异步模式的典型实现是基于操作系统底层异步 API 的，所以我们可称之为“系统级别”的或者“真正意义上”的异步，因为具体的读写是由操作系统代劳的。

Reactor 与 Proactor 两种模式的场景区别：

下面是 Reactor 的做法：

1. 等待事件响应 (Reactor job)。
2. 分发 “Ready-to-Read” 事件给用户句柄 (Reactor job)。
3. 读数据 (user handler job)。

4. 处理数据(user handler job)。

下面再来看看真正意义的异步模式 Proactor 是如何做的：

1. 等待事件响应 (Proactor job)。

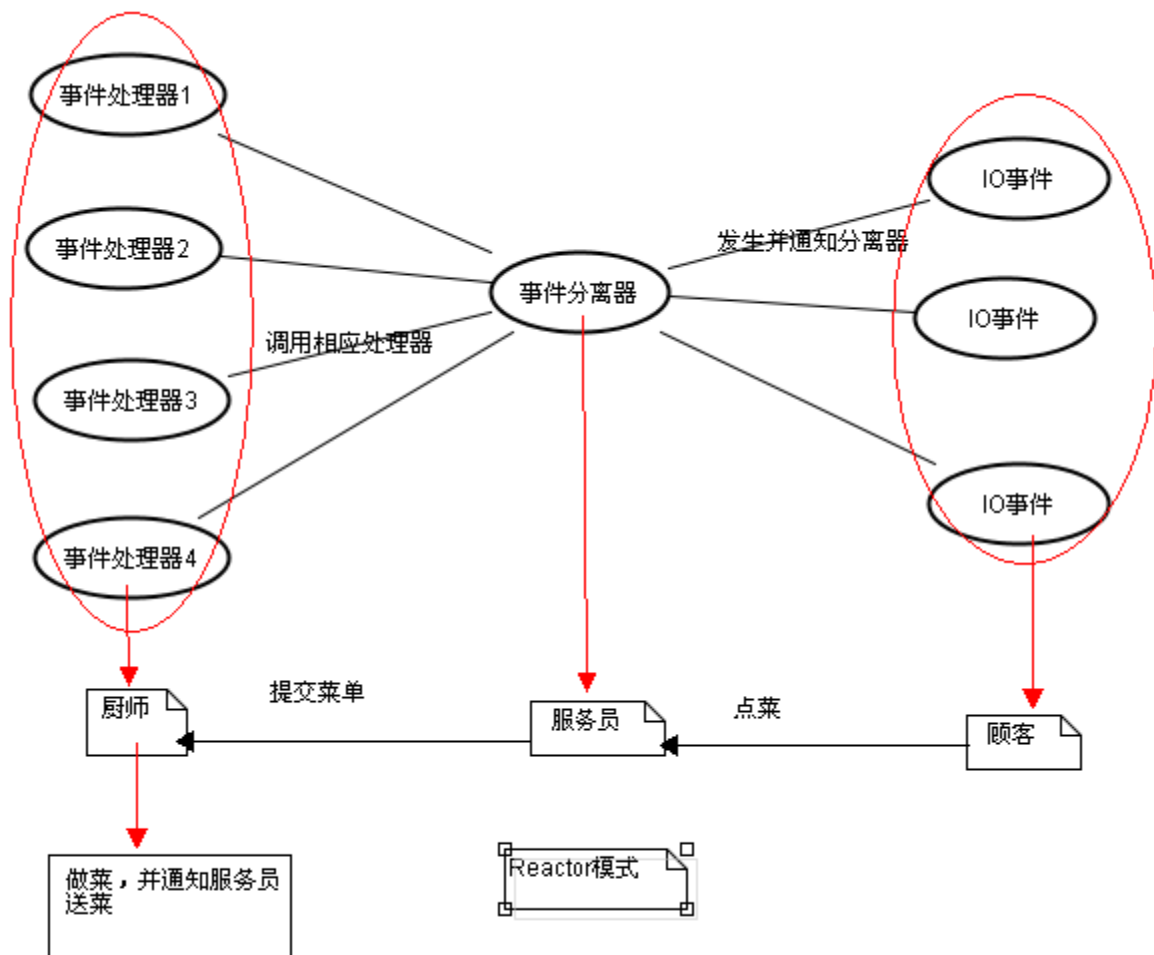
2. 读数据 (Proactor job)。

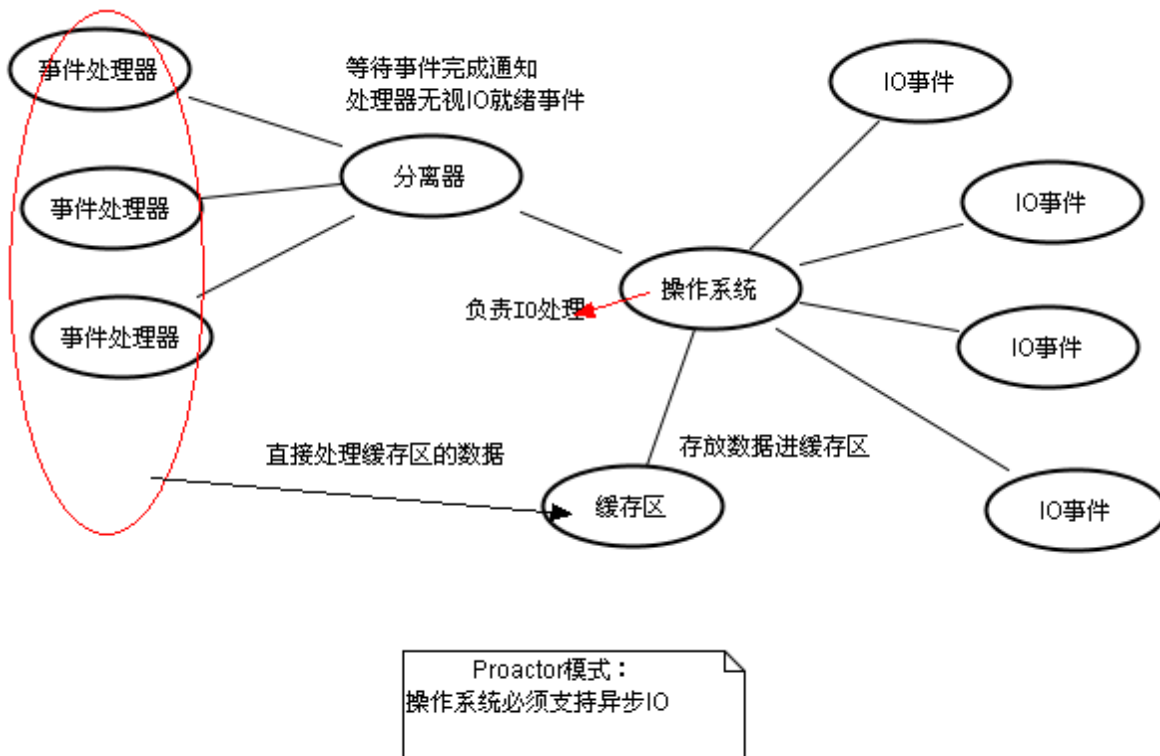
3. 分发 “Read-Completed” 事件给用户句柄 (Proactor job)。

4. 处理数据(user handler job)。

从上面可以看出，Reactor 和 Proactor 模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor 中需要应用程序自己读取或者写入数据，而 Proactor 模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的 IO 设备。

最后结合下面的两张图更容易理解（这是别人的图，非原创）：



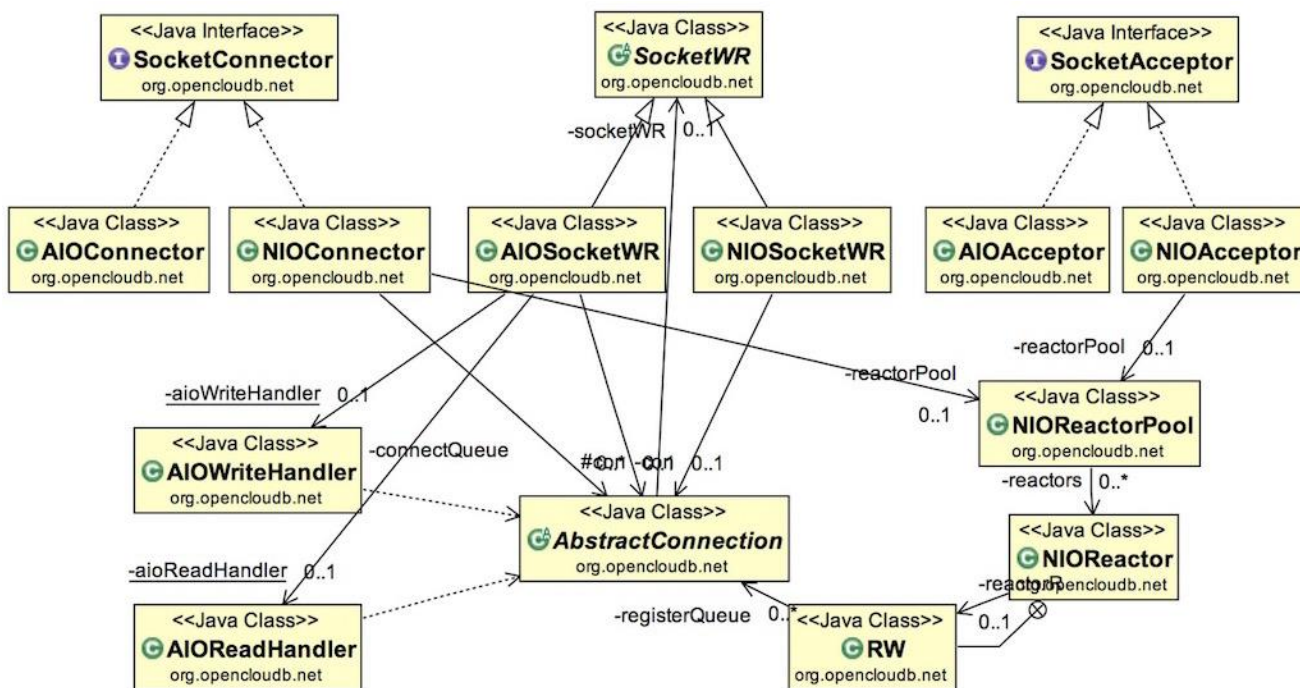


可以看到两者图中都有分离器，在 JAVA NIO 框架中分离器的逻辑需要用户通过 selector 自己完成

在 JAVA AIO 框架中，分离器有系统 API 自动完成，AsynchronousChannelGroup 就代替了分离的作用。

6.2.3 支持 AIO 和 NIO 的框架

前面已经讲了，MyCAT 可以通过系统参数选择是使用 AIO 还是 NIO，那么在代码里面是如何做到同时支持两种架构的呢。可以看下面的类图：



- SocketConnector 发起连接请求类，如 MyCAT 与 MySQL 数据库的连接，都是由 MyCAT 主动发起连接请求。
- SocketAcceptor 接收连接请求类，如 MyCAT 启动 9066 和 8066 分别侦听管理员和应用程序的连接请求。
- SocketWR 读写操作类，SocketConnector 和 SocketAcceptor 只负责 socket 建立，当 socket 连接建立后进行字节的读写操作则由 SocketWR 来完成。

这几个接口分别处理网络通道的四种不同类型的事件：

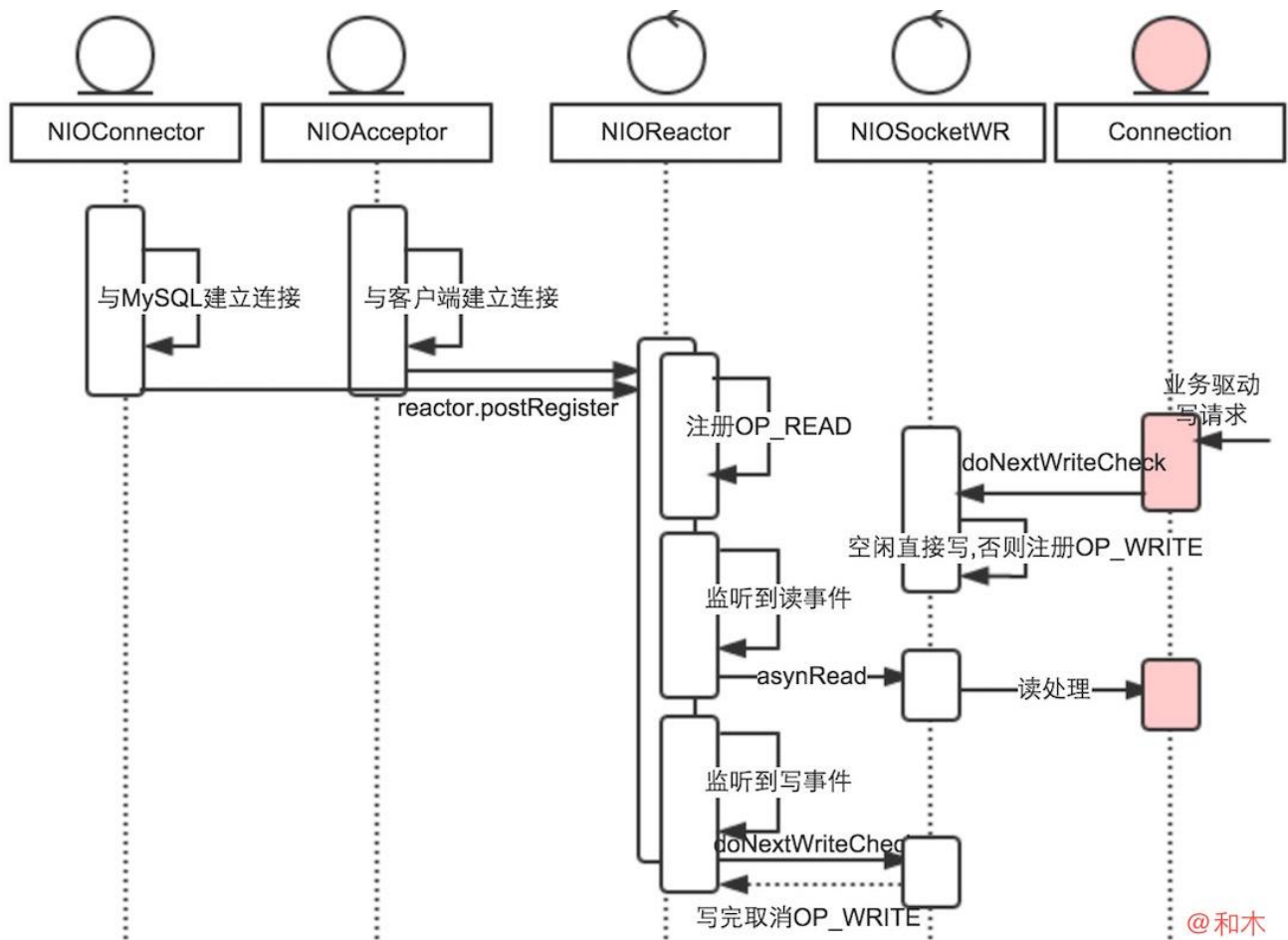
- Connect 客户端连接服务端事件。
- Accept 服务端接收客户端连接事件。
- Read 读事件。
- Write 写事件。

这四种事件在 AIO 和 NIO 的实现差别如下：

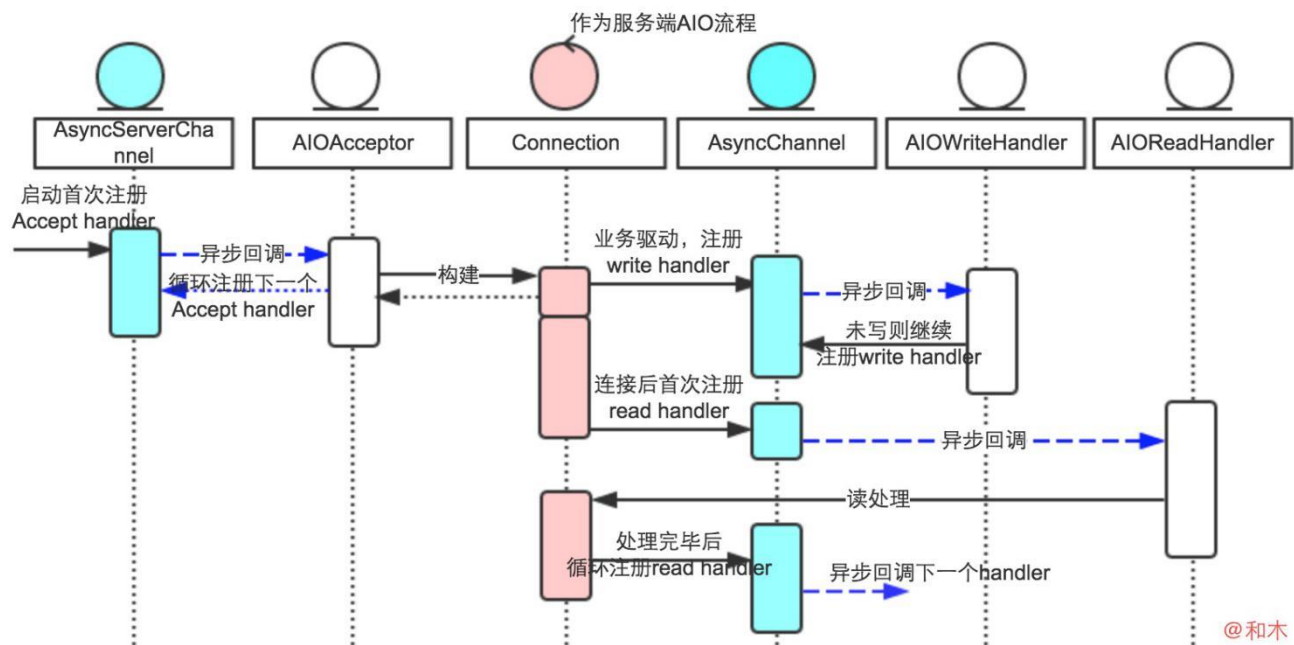
操作	NIO	AIO
Connect	注册 OP_CONNECT 事件，通过 selector 线程循环检查事件是否就绪	通过 AIO 的 connect 函数进行连接调用并注册 CompletionHandler 句柄，事件发生后回调
Accept	注册 OP_ACCEPT 事件，通过 selector 线程循环检查事件是否就绪	通过 AIO 的 accept 函数进行连接准备调用并注册 CompletionHandler 句柄，事件发生后回调
read	注册 OP_READ 事件，通过 selector 线程循环检查事件是否就绪	通过 AIO 的 read 函数传递缓存读内容的 buffer，并注册 CompletionHandler 句柄，事件发生后回调，回调时读入的内容已经写入 buffer
write	1.若通道空闲当前线程直接写，否则缓存队列，注册 OP_Write 事件； 2.通过 selector 线程循环检查写事件是否就绪	通过 AIO 的 write 函数传递要写的 buffer，并注册 CompletionHandler 句柄，事件发生后回调，回调时 buffer 内容已经写入到通道了

上面的类图看起来有些复杂，因为把 NIO 和 AIO 放在一起了，那么我们分开来讲。

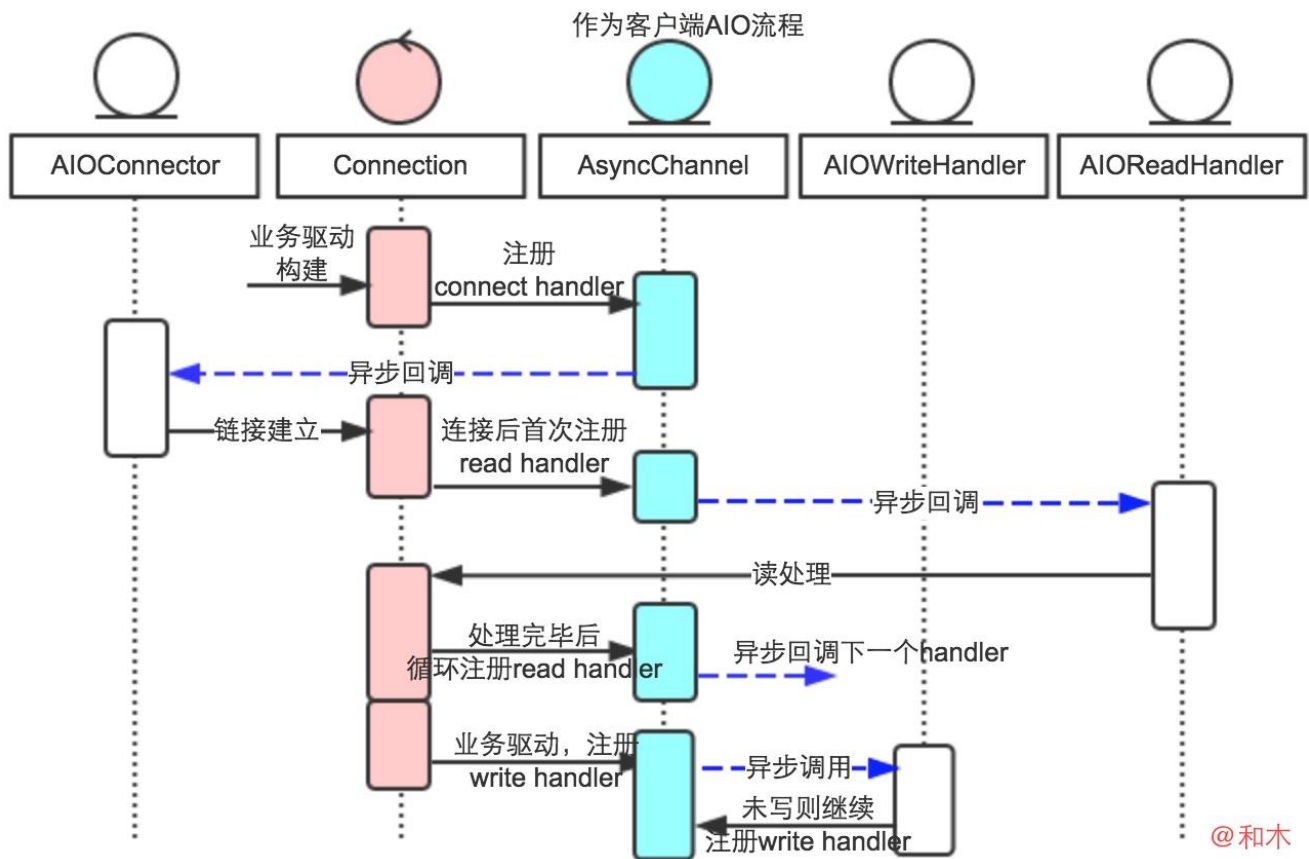
NIO 主要类调用



AIO 主要类调用-服务端



AIO 主要类调用-客户端



看起来好像是 AIO 的调用比 NIO 多吧，其实 NIO 比 AIO 要略麻烦些，因为 AIO 的调用关系全画了，NIO 对链接建立过程进行简化，否则一个图上画不开了。

6.2.4 MyCAT 的 NIO 实现

Selector（选择器）是 Java NIO 中能够检测一到多个 NIO 通道，并能够知晓通道是否为诸如读写事件做好准备的组件。这样，一个单独的线程可以管理多个 channel，从而管理多个网络连接。

Selector 可以监听四种不同类型的事件：

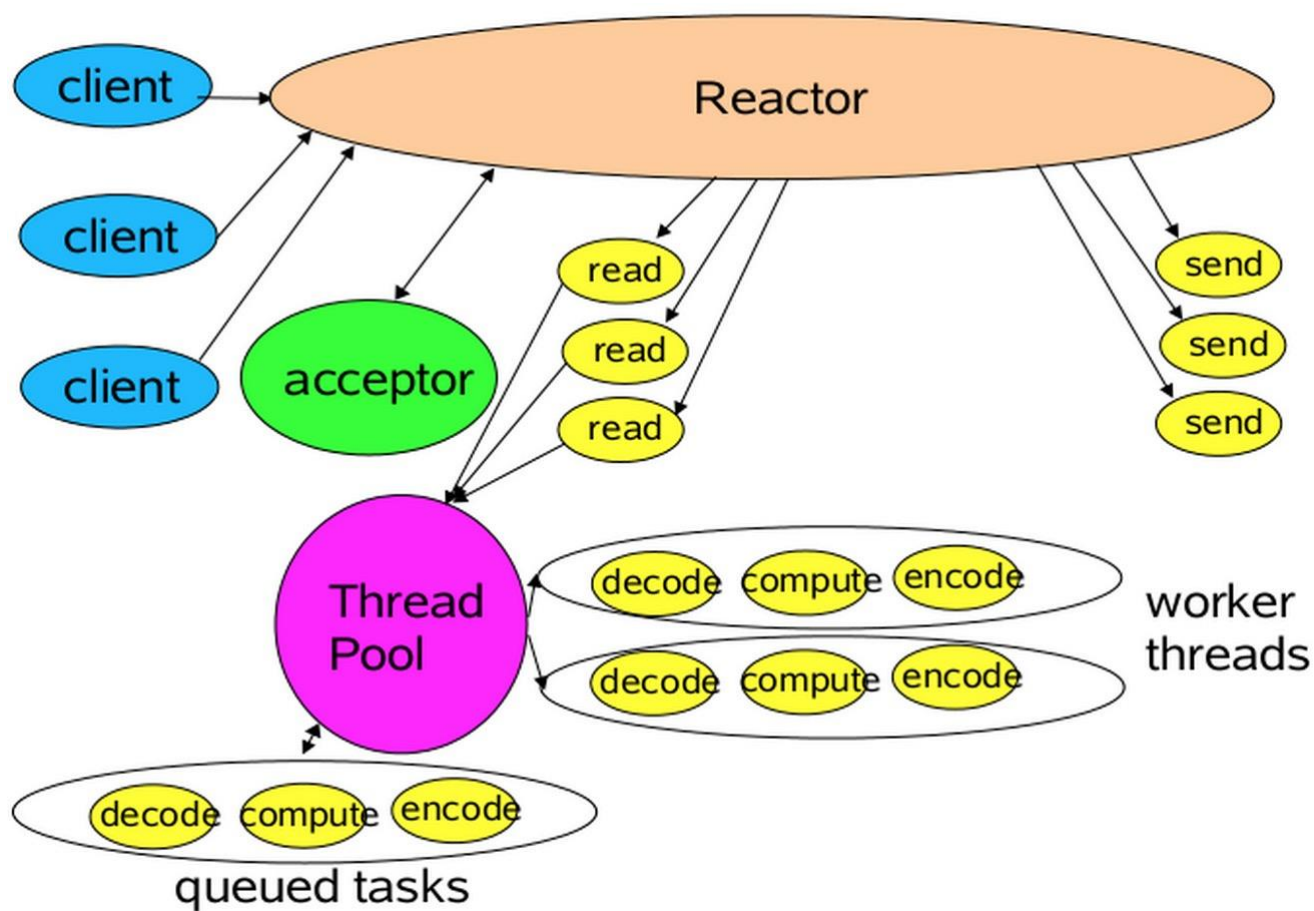
- Connect
- Accept
- Read
- Write

这四种事件用 SelectionKey 的四个常量来表示：

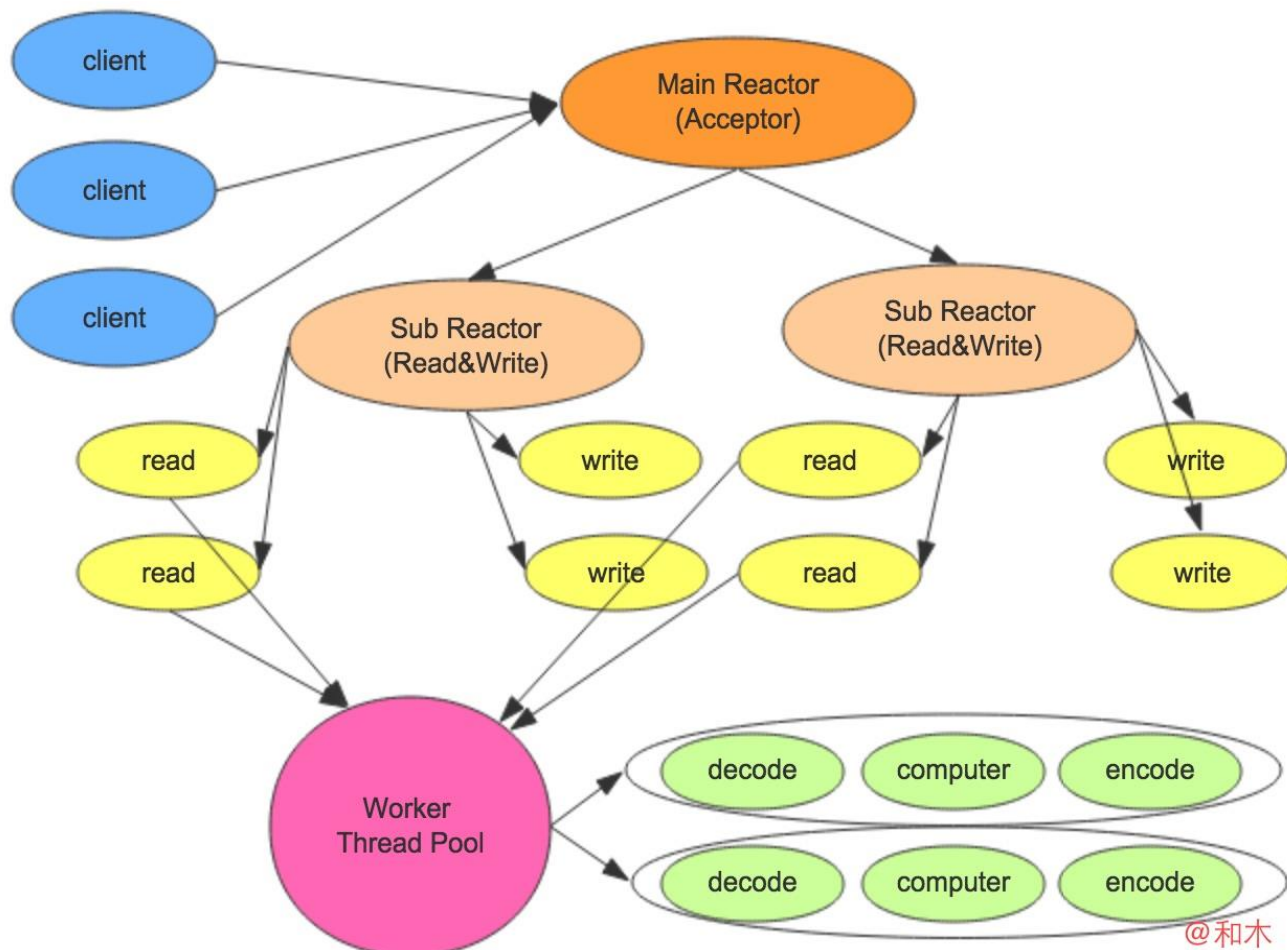
- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

前面已经说了，NIO 采用的 Reactor 模式：例如汽车是乘客访问的主体（Reactor），乘客上车后，到售票员（acceptor）处登记，之后乘客便可以休息睡觉去了，当到达乘客所要到达的目的地后，售票员将其唤醒即可。

典型的 Reactor 场景

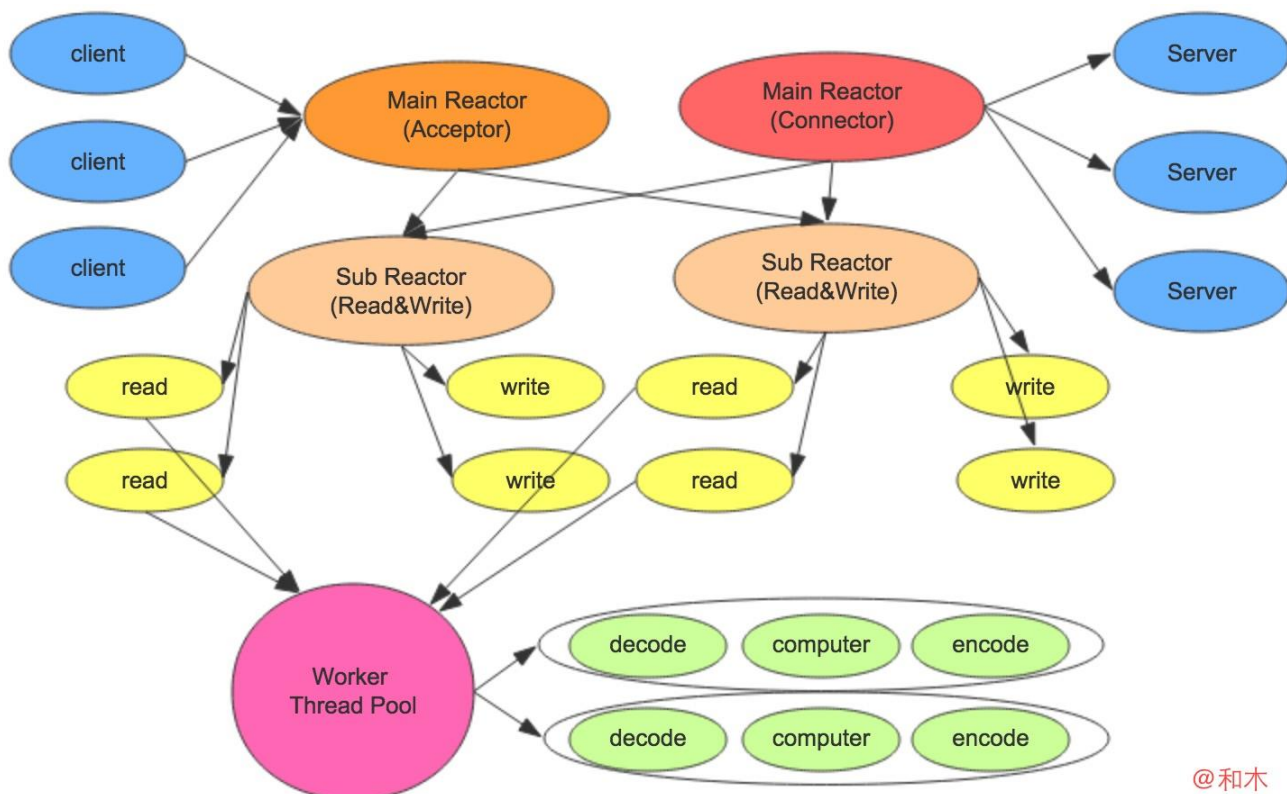


在高性能 IO 框架中，大都是采用多 Reactor 模式,即多个 dispatcher, 如下图所示:



@和木

上图是服务端采用多 Reactor 模式的典型场景，MyCAT 也采用多 Reactor 模式，另外 MyCAT 不仅做服务端，也要作为客户端去连接后端 MySQL Server，所以实际场景如下图所示：



@和木

多 Reactor 区分说明：

通常 Reactor 实现为一个线程,内部维护一个 Selector。

```
while(true){  
    int sel=selector.select(timeout);  
    processRegister();  
    if(sel>0)  
        processSelected();  
}
```

6.2.4.1 NIOConnector 类分析

NIOConnector 处理的是 Connect 事件，是客户端连接服务端事件，就是 MyCAT 作为客户端去主动连接 MySQL Server 的操作。

NIOConnector 类声明和关键成员变量

```
public final class NIOConnector extends Thread implements SocketConnector {  
  
    private final Selector selector;  
    private final BlockingQueue<AbstractConnection> connectQueue;  
    private final NIOReactorPool reactorPool;  
}
```

可以看到 NIOConnector 是一个线程，三个主要的成员变量

- selector 事件选择器
- connectQueue 需要建立连接的对象，临时放在这个队列里
- reactorPool 当连接建立后，从 reactorPool 中分配一个 NIOReactor，处理 Read 和 Write 事件

postConnect 函数

```
public void postConnect(AbstractConnection c) {  
    connectQueue.offer(c);  
}
```

```
selector.wakeup();
}
```

postConnect 函数的作用，是把需要建立的连接放到 connectQueue 队列中，然后再唤醒 selector。

postConnect 是在新建连接或者心跳时被 XXXXConnectionFactory 触发的。

```
▼ postConnect(AbstractConnection) : void - org.opencldb.net.NIOConnector
  ▼ make(MySQLDataSource, ResponseHandler, String) : MySQLConnection - org.opencldb.mysql.nio.MySQLConnectionFactory
    ► createNewConnection(ResponseHandler, String) : void - org.opencldb.mysql.nio.MySQLDataSource
  ▼ make(MySQLHeartbeat) : MySQLDetector - org.opencldb.heartbeat.MySQLDetectorFactory
    ► heartbeat() : void - org.opencldb.heartbeat.MySQLHeartbeat
```

connect 函数

```
private void connect(Selector selector) {
AbstractConnection c = null;
while ((c = connectQueue.poll()) != null) {
try {
SocketChannel channel = (SocketChannel) c.getChannel();
channel.register(selector, SelectionKey.OP_CONNECT, c);
channel.connect(new InetSocketAddress(c.host, c.port));
} catch (Throwable e) {
c.close(e.toString());
}
}
}}
```

connect 函数的目的就是处理 postConnect 函数操作的 connectQueue 队列：

1. 判断 connectQueue 中是否有新的连接请求。
2. 建立一个 SocketChannel。
3. 在 selector 中进行注册 OP_CONNECT。
4. 发起 SocketChannel.connect()操作。

run 函数

```
public void run() {
for (;;) {
```

```

        .....

selector.select(1000L);

connect(selector);

Set<SelectionKey> keys = selector.selectedKeys();

try {
    for (SelectionKey key : keys) {
        Object att = key.attachment();

        if (att != null && key.isValid() && key.isConnectable()) {

            finishConnect(key, att);

        } else {

            key.cancel();

        }

    }

} finally {

    keys.clear();

}

        .....

}

}

```

NIOConnector 继承 Thread 实现 run()函数，这是一个无限循环体，包含了两个主要循环操作

- 调用 connect 函数中，判断 connectQueue 中是否有新的连接请求，如有则在 selector 中进行注册，然后发起连接

- selector 监听事件，然后在 finishConnect 函数中对事件进行处理。在 NIOConnector 类中，只注册了 OP_CONNECT 事件，所以只对 OP_CONNECT 事件进行处理。

finishConnect 函数

在 NIOConnector 类中，只处理 OP_CONNECT 事件，当连接建立完毕后，Read 和 Write 事件如何处理呢？可以在 finishConnect 函数看到，当连接建立完毕后，从 reactorPool 中获得一个 NIOReactor，然后把连接传递到 NIOReactor，然后后续的 Read 和 Write 事件就交给 NIOReactor 处理了。

```
private void finishConnect(SelectionKey key, Object att) {
    BackendAIOConnection c = (BackendAIOConnection) att;

    .....

    NIOReactor reactor = reactorPool.getNextReactor();
    reactor.postRegister(c);

    .....
}
```

6.2.4.2.NIOAcceptor 类分析

NIOAcceptor 处理的是 Accept 事件，是服务端接收客户端连接事件，就是 MyCAT 作为服务端去处理前端业务程序发过来的连接请求。

NIOAcceptor 类声明和关键成员变量

```
public final class NIOAcceptor extends Thread implements SocketAcceptor{

private final Selector selector;

private final ServerSocketChannel serverChannel;

private final NIOReactorPool reactorPool;

}
```

可以看到 NIOAcceptor 的主体结构，与 NIOConnector 比较像，也是一个线程，也有三个主要的成员变量（其它非主要变量就不在这儿——列出了）。

- selector 事件选择器。
- serverChannel 监听新进来的 TCP 连接的通道。
- reactorPool 当连接建立后，从 reactorPool 中分配一个 NIOReactor，处理 Read 和 Write 事件。

NIOAcceptor 的构造函数

监听通道在 NIOAcceptor 构造函数里启动,然后注册到实际进行任务处理的 Dispatcher 线程的 Selector 中。

```
public NIOAcceptor(String name, String bindIp,int port,
    FrontendConnectionFactory factory, NIOReactorPool reactorPool)
    throws IOException {
```

```

this.selector = Selector.open();

this.serverChannel = ServerSocketChannel.open();

this.serverChannel.configureBlocking(false);

/** 设置 TCP 属性 */

serverChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);

serverChannel.setOption(StandardSocketOptions.SO_RCVBUF, 1024 * 16 * 2);

// backlog=100

serverChannel.bind(new InetSocketAddress(bindIp, port), 100);

this.serverChannel.register(selector, SelectionKey.OP_ACCEPT);

}

```

run 函数

```

public void run() {
for (;;) {
    try {
        selector.select(1000L);

        Set<SelectionKey> keys = selector.selectedKeys();

        try {
            for (SelectionKey key : keys) {
                if (key.isValid() && key.isAcceptable()) {
                    accept();
                } else {
                    key.cancel();
                }
            }
        } finally {
            keys.clear();
        }
    } catch (Throwable e) {

```

```

    LOGGER.warn(getName(), e);
}
}
}

```

NIOAcceptor 继承 Thread 实现 run()函数，与 NIOConnector 的 run()类似,也是一个无限循环体：

selector 不断监听连接事件，然后在 accept()函数中对事件进行处理。

在 NIOAcceptor 类中，只注册了 OP_ACCEPT 事件，所以只对 OP_ACCEPT 事件进行处理。

accept 函数

```

private void accept() {
channel = serverChannel.accept();
channel.configureBlocking(false);
FrontendConnection c = factory.make(channel);

.....

NIOReactor reactor = reactorPool.getNextReactor();
reactor.postRegister(c);

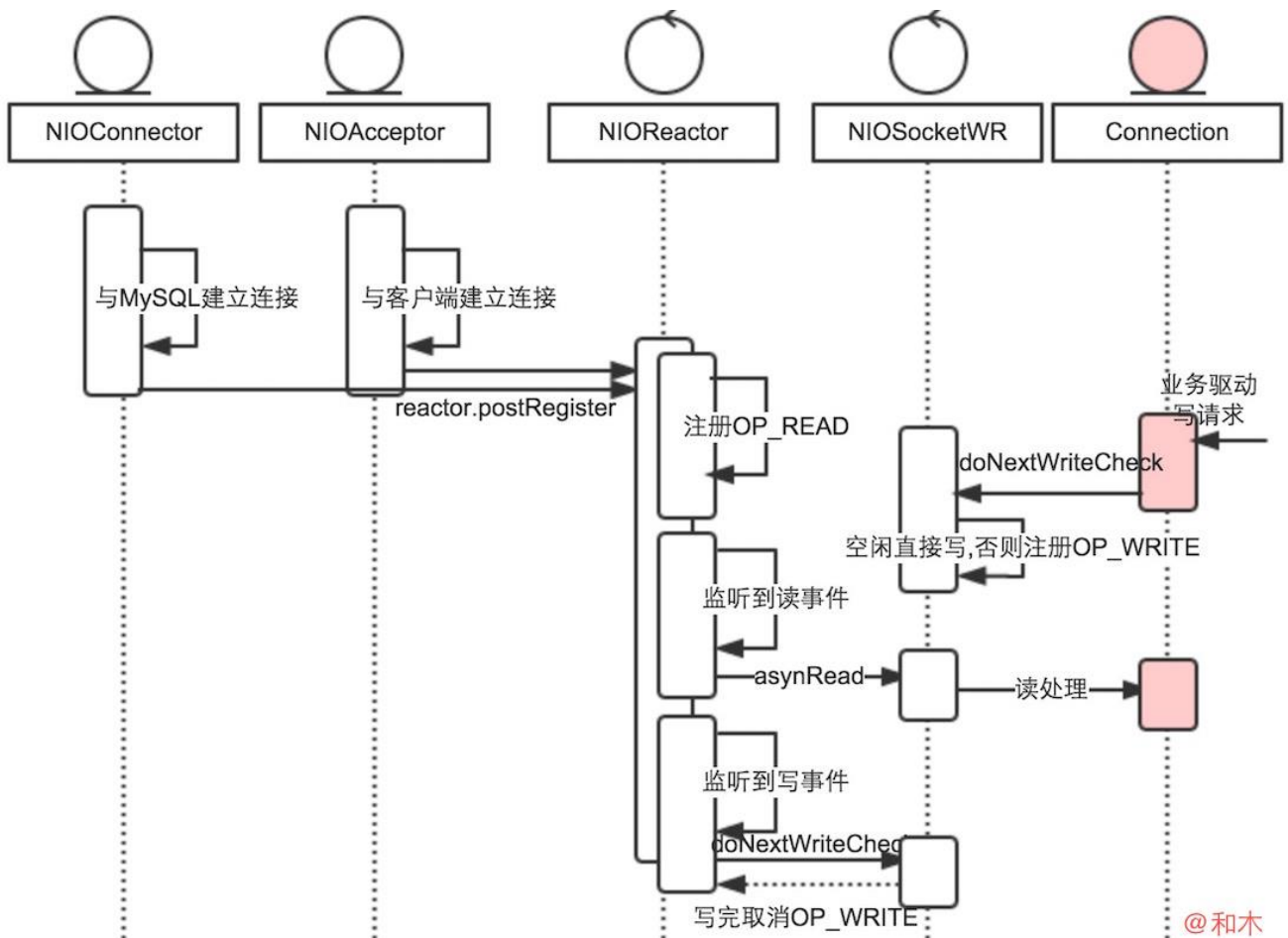
.....
}

```

NIOAcceptor 的 accept () 与 NIOConnector 的 finishConnect()类似，当连接建立完毕后，从 reactorPool 中获得一个 NIOReactor，然后把连接传递到 NIOReactor，然后后续的 Read 和 Write 事件就交给 NIOReactor 处理了。

6.2.4.3.NIOSocketWR 和 NIOReactor 分析

NIOConnector 和 NIOAcceptor 分别完成连接的建立，真正的内容的读写是由 NIOSocketWR 和 NIOReactor 共同完成的。可以参见下图。



先说一下 NIOSocketWR 和 NIOReactor 的关系。

下面是 NIOSocketWR 的类声明和主要成员变量，可以看到 NIOSocketWR 针对的某一条链路。

```
public class NIOSocketWR extends SocketWR {
    private SelectionKey processKey;
    private final AbstractConnection con;
    private final SocketChannel channel;
}
```

在来看一下 NIOReactor 的内部类 RW 的类声明和主要成员变量，可以看到 NIOReactor 包含一个 selector，是一个 dispatcher，用来负责多个链路事件的事件分发。

```
private final class RW implements Runnable {
    private final Selector selector;
    private final ConcurrentLinkedQueue<AbstractConnection> registerQueue;
}
```

NIOReactor.postRegister()

NIOConnector 和 NIOAcceptor 建立连接后，调用 NIOReactor.postRegister 进行注册。

```
final void postRegister(AbstractConnection c) {  
    reactorR.registerQueue.offer(c);  
    reactorR.selector.wakeup();  
}
```

NIOReactor.postRegister 并没有直接注册，而是把 AbstractConnection 对象加入缓冲队列,然后 wakeup selector 等待注册。

直接注册不可吗? 不是不可以,是效率问题, 至少加两次锁,锁竞争激烈。

- Channel 本身的 regLock,竞争几乎没有。
- Selector 内部的 key 集合,竞争激烈。

更好的方式就是采用上面这种方式，先放入缓冲队列，等待 selector 单线程进行注册。

NIOReactor.RW.run()

```
public void run() {  
    Set<SelectionKey> keys = null;  
    for (;;) {  
        try {  
            selector.select(500L);  
            register(selector);  
            keys = selector.selectedKeys();  
            for (SelectionKey key : keys) {  
                AbstractConnection con = null;  
                try {  
                    Object att = key.attachment();  
                    if (att != null && key.isValid()) {  
                        con = (AbstractConnection) att;  
                        if (key.isReadable()) {  
                            con.asynRead();  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```



```

    }

    if (key.isWritable()) {
        con.doNextWriteCheck();
    }

    } else {
        key.cancel();
    }

    } catch (Throwable e) {

    }

    }

    } catch (Throwable e) {
        LOGGER.warn(name, e);
    } finally {
        if (keys != null) {
            keys.clear();
        }
    }
}
}

```

NIOReactor 在内部类 RW 中继承 Thread 实现 run()函数，这是一个无限循环体，包含了三个主要循环操作。

- 注册事件，这儿只是注册 OP_READ 事件。

OP_WRITE 事件的注册放在 NIOSocketWR.doNextWriteCheck()函数中，doNextWriteCheck 既被 selector 线程调用，也会被其它的业务线程调用，此时就会存在 lock 竞争的问题，所以对于 OP_WRITE 事件也建议用队列缓存的方式，不过对于 MyCAT 的流量场景，大部分写操作是由业务线程直接写入，只有在网络繁忙时，业务线程不能一次全部写完，才会通过 OP_WRITE 注册方式进行候补写。所以此处可以考虑优化，但是性能上到底有多大提升，是否值得，优化前倒需要斟酌下。

- selector 监听事件，如果是读事件，就调用 `con.asyncRead()` 函数，进行字节的读取。对于 `asyncRead` 中如何提取 MySQL 协议包，就属于网络框架讨论的内容，可以参考其它章节。

- selector 监听到写事件，调用 `AbstractConnection.doNextWriteCheck()` 进行写事件的处理，在 `AbstractConnection.doNextWriteCheck()` 中，又调用 `NIOSocketWR.doNextWriteCheck()` 进行处理的。

NIOSocketWR.doNextWriteCheck()

`NIOSocketWR.doNextWriteCheck()` 的调用关系如下：

A method call graph showing the sequence of calls starting from `doNextWriteCheck()` in `org.opencldb.net.NIOSocketWR`. The graph includes `doNextWriteCheck()` in `AbstractConnection`, `run()` in `NIOReactor.RW`, `write(ByteBuffer)` in `AbstractConnection`, and several `execute` and `connectionError` methods in various handlers and response objects. The graph is presented in a tree-like structure with expandable nodes.

调用

者有两个

1. selector 循环写事件侦听。
2. 其它业务线程触发的写操作。

```
public void doNextWriteCheck() {  
    if (!writing.compareAndSet(false, true)) {  
        return;  
    }  
    try {  
        boolean noMoreData = write0();  
        writing.set(false);  
        if (noMoreData && con.writeQueue.isEmpty()) {  
            if ((processKey.isValid() && (processKey.interestOps() & SelectionKey.OP_WRITE) != 0)) {  
                disableWrite();  
            }  
        } else {  
            if ((processKey.isValid() && (processKey.interestOps() & SelectionKey.OP_WRITE) == 0)) {
```

```

        enableWrite(false);
    }
}
} catch (IOException e) {
    .....
}
}
}

```

1. 先判断是否正在写，如果正在写，退出（之前已经把写内容放到缓冲队列，那么此处是否可以优化呢，即当发送缓冲队列为空的时候,可以直接往 channel 写数据，不能写再放缓冲队列，理论上可以优化，但是写代码时要注意，因为必需要保证协议包的顺序，还要考虑到前一次写时，是否有 buffer 没有写完，若前一次写入时，最后一个 buffer 没有写完，记得退回缓冲队列；MyCAT 当前的实现方式是增加了一个变量专门存放上次未写完的 buffer) 。

2. write0()方法是只要 buffer 中还有，就不停写入；直到写完所有 buffer，或者写入时，返回写入字节为零，表示网络繁忙，就回临时退出写操作。

3. 没有完全写入并且缓冲队列为空,取消注册写事件。

4. 没有完全写入或者缓冲队列有代写对象,继续注册写时间。

5. 特别说明，writing.set(false)必须要在 boolean noMoreData = write0()之后和 if (noMoreData && con.writeQueue.isEmpty())之前，否则会导致当网络流量较低时，消息包缓存在内存中迟迟发不出去的现象。

6.3 与 Cobar 原有 NIO 细节比较

6.3.1 Cobar 的 NIO

Cobar 后端是采用 BIO，前端采用 NIO；Cobar 的 BIO 这儿就不必提了，对于原有 NIO 实现，跟 MyCAT 相比，读方式差不多，写的差别比较大。

NIOReactor.postWrite()

这儿传入的参数，不是要写的 buffer，而是一个连接对象，只是注册这个对象有内容需要写。要写的 buffer，在连接对象自己的缓存队列中。

这种方式与 MyCAT 差不多，连接对象自己维护写队列。

```
final void postWrite(NIOConnection c) {
    reactorW.writeQueue.offer(c);
}
```

NIOReactor.W 内部类

专门负责缓冲队列写，不停循环遍历，等待其它业务线程放入写数据。

```
private final class W implements Runnable {
    private final BlockingQueue<NIOConnection> writeQueue;
    private W() {
        this.writeQueue = new LinkedBlockingQueue<NIOConnection>();
    }
    public void run() {
        NIOConnection c = null;
        for (;;) {
            try {
                if ((c = writeQueue.take()) != null) {
                    c.writeByQueue();
                }
            } catch (Throwable e) {}
        }
    }
}
```

NIOReactor.R 内部类,为一个 selector

同时处理读事件和写事件。但是主要负责的是读，只有在网络非常繁忙等极少数情况下，小概率走到读分支。

```

private final class R implements Runnable {

private final Selector selector;

@Override

public void run() {

    final Selector selector = this.selector;

    for (;;) {

        try {

            selector.select(1000L);

            register(selector);

            Set<SelectionKey> keys = selector.selectedKeys();

            for (SelectionKey key : keys) {

                Object att = key.attachment();

                if (att != null && key.isValid()) {

                    int readyOps = key.readyOps();

                    if ((readyOps & SelectionKey.OP_READ) != 0) {

                        read((NIOConnection) att);

                    } else if ((readyOps & SelectionKey.OP_WRITE) != 0) {

                        c.writeByEvent();

                    } else {

                        key.cancel();

                    }

                } else {

                    key.cancel();

                }

            }

        } catch (Throwable e) {

        }

    }

}

```

```
}  
  
}
```

基于队列的写和基于事件的写

- 队列写：所有的写请求，放到缓存队列，由独立 W 线程进行写。如果未写完（比如网络繁忙），则注册写事件，然后会再 selector 发现写事件。
- 事件写：R 线程中，selector 探测到写事件后，进行写操作。如果写完了，则立即取消注册写事件，避免继续触发导致循环。
- 总结：主要是 W 线程进行写，只有在网络繁忙时，才会注册写事件，等待网络写就绪后，R 线程就会立即发现写事件，然后 R 线程再写一部分。

```
@Override  
  
public void writeByQueue() throws IOException {  
  
    if (isClosed.get()) {  
  
        return;  
  
    }  
  
    final ReentrantLock lock = this.writeLock;  
  
    lock.lock();  
  
    try {  
  
        // 满足以下两个条件时，切换到基于事件的写操作。  
  
        // 1.当前 key 对写事件不该兴趣。  
  
        // 2.write0()返回 false。  
  
        if ((processKey.interestOps() & SelectionKey.OP_WRITE) == 0  
&& !write0()) {  
  
            enableWrite();  
  
        }  
  
    } finally {  
  
        lock.unlock();  
  
    }  
  
}
```

```

@Override

public void writeByEvent() throws IOException {

    if (isClosed.get()) {

        return;

    }

    final ReentrantLock lock = this.writeLock;

    lock.lock();

    try {

        // 满足以下两个条件时，切换到基于队列的写操作。

        // 1.write0()返回 true。

        // 2.发送队列的 buffer 为空。

        if (write0() && writeQueue.size() == 0) {

            disableWrite();

        }

    } finally {

        lock.unlock();

    }

}

/**
 * 打开写事件
 */

private void enableWrite() {

    final Lock lock = this.keyLock;

    lock.lock();

    try {

        SelectionKey key = this.processKey;

        key.interestOps(key.interestOps() | SelectionKey.OP_WRITE);

    } finally {

```

```

    lock.unlock();
}

processKey.selector().wakeup();
}

/**
 * 关闭写事件
 */
private void disableWrite() {
    final Lock lock = this.keyLock;
    lock.lock();
    try {
        SelectionKey key = this.processKey;
        key.interestOps(key.interestOps() & OP_NOT_WRITE);
    } finally {
        lock.unlock();
    }
}
}

```

6.3.2 比较 MyCAT 和 Cobar 两种写方式

- Cobar 的写：业务线程把写请求放到缓冲队列，然后由独立写线程 W 负责，当 W 在写的时候，网络慢等原因导致未写完，然后注册写事件，由 R 线程(selector)进行候补写。
- MyCAT 的写：业务线程先通过加锁或者 AtomicBoolean 判断当前 channel 是否正在写数据，如空闲则由当前线程直接写，否则入缓冲队列交给其他线程写；在写的时候，网络慢等原因导致未写完，然后注册写事件，由 NIOReactor 线程(selector)进行候补写。
- MyCAT 采用这种方式的显著优点：尽可能减少系统调用和线程切换。

6.4 MyCAT 的 AIO 实现

6.4.1 JAVA AIO 体系

从代码风格上比较，NIO 和 AIO 的差别，就是 Reactor 和 Proactor 两种模式差别，对于典型的读场景，来回顾下他们的区分：

Reactor 的做法：

1. 等待事件响应 (Reactor job)。
2. 分发 “Ready-to-Read” 事件给用户句柄 (Reactor job)。
3. 读数据 (user handler job)。
4. 处理数据 (user handler job)。

Proactor 的做法：

1. 等待事件响应 (Proactor job)。
2. 读数据 (Proactor job)。
3. 分发 “Read-Completed” 事件给用户句柄 (Proactor job)。
4. 处理数据 (user handler job)。

可以看到两者最大的区别，就是到了 AIO，用户只管专心负责对读到的数据进行处理，如何读的过程过程就全交给系统层面去完成。

同样对于写操作，在 AIO 方式中，应用层只管把要写的 buffer 传递出去，等到系统写完，再回调应用层做其它动作。

而在 NIO 方式中，应用层要自己控制 buffer 写入 channel 的过程。

首先看下 AIO 引入的新的类和接口：

```
java.nio.channels.AsynchronousChannel
```

- 标记一个 channel 支持异步 IO 操作。

```
java.nio.channels.AsynchronousServerSocketChannel
```

- ServerSocket 的 aio 版本，创建 TCP 服务端，绑定地址，监听端口等。

```
java.nio.channels.AsynchronousSocketChannel
```

- 面向流的异步 socket channel，表示一个连接。

```
java.nio.channels.AsynchronousChannelGroup
```

- 异步 channel 的分组管理，目的是为了资源共享。一个 `AsynchronousChannelGroup` 绑定一个线程池，这个线程池执行两个任务：处理 IO 事件和派发 `CompletionHandler`。`AsynchronousServerSocketChannel` 创建的时候可以传入一个 `AsynchronousChannelGroup`，那么通过 `AsynchronousServerSocketChannel` 创建的 `AsynchronousSocketChannel` 将同属于一个组，共享资源。

```
java.nio.channels.CompletionHandler
```

- 异步 IO 操作结果的回调接口，用于定义在 IO 操作完成后所作的回调工作。
- AIO 的 API 允许两种方式来处理异步操作的结果：返回的 `Future` 模式或者注册 `CompletionHandler`，MyCAT 采用的是 `CompletionHandler` 的方式，这些 handler 的调用是由 `AsynchronousChannelGroup` 的线程池派发的。

`AsynchronousChannelGroup` 实际上扮演 `Proactor` 的角色，业务逻辑通过 `CompletionHandler` 接口实现。在整个 JAVA AIO 体系中，主要由四个地方需要注册 `CompletionHandler`，分别对应 `Accept`、`Connect`、`Read`、`Write` 四个不同的事件。

`AsynchronousServerSocketChannel` 类的 `accept`。

```
public abstract <A> void accept(A attachment,  
                                CompletionHandler<AsynchronousSocketChannel,? super A> handler)
```

`AsynchronousSocketChannel` 类的

```
public abstract <A> void connect(SocketAddress remote,  
                                A attachment,  
                                CompletionHandler<Void,? super A> handler)  
  
public final <A> void read(ByteBuffer dst,  
                           A attachment,  
                           CompletionHandler<Integer,? super A> handler)  
  
public final <A> void write(ByteBuffer dst,  
                             A attachment,  
                             CompletionHandler<Integer,? super A> handler)
```

在 Mycat 工程中，有四个类实现 `CompletionHandler` 接口，分别满足上面四个事件的注册。

6.4.2 AIOAcceptor

NIOAcceptor 负责作为服务端接受客户端的请求，通过 `AsynchronousServerSocketChannel.accept()` 进行写 accept 事件的注册。

类声明

虽然 `CompletionHandler` 定义为 `CompletionHandler<V,A>`，根据 `AsynchronousServerSocketChannel.accept()` 的参数定义，对 `AIOAcceptor` 而言，`V` 已经固定为 `AsynchronousSocketChannel`，`A` 可以自定义。

```
public final class AIOAcceptor implements SocketAcceptor,
    CompletionHandler<AsynchronousSocketChannel, Long> {
    private final AsynchronousServerSocketChannel serverChannel;
    private final FrontendConnectionFactory factory;

    public AIOAcceptor(String name, String ip, int port,
        FrontendConnectionFactory factory, AsynchronousChannelGroup group)
        throws IOException {
        ...
        this.factory = factory;
        serverChannel = AsynchronousServerSocketChannel.open(group);
        // backlog=100
        serverChannel.bind(new InetSocketAddress(ip, port), 100);
    }
}
```

跟 `NIOAcceptor` 一样，`AIO` 也要启动一个监听通道 `serverChannel`，绑定一个侦听端口。

启动方法 start

```
public void start() {
    this.pendingAccept();
};

private void pendingAccept() {
    if (serverChannel.isOpen()) {
```

```

serverChannel.accept(ID_GENERATOR.getId(), this);
}
}

```

AIO 的启动方法非常简单，就是调用 `AsynchronousServerSocketChannel` 的 `accept` 方法，把用户定义的 `CompletionHandler` 即 `AIOAcceptor` 传递就可以了。由 `AsynchronousChannelGroup` 担任 `proactor` 角色，当连接建立时，回调 `AIOAcceptor` 的 `completed` 或者 `failed` 方法。

completed 方法

```

@Override
public void completed(AsynchronousSocketChannel result, Long id) {
    accept(result, id);
    // next pending waiting
    pendingAccept();
}

private void accept(NetworkChannel channel, Long id) {
    try {
        ....
        FrontendConnection c = factory.make(channel);
        NIOProcessor processor = MycatServer.getInstance().nextProcessor();
        c.setProcessor(processor);
        c.register();
    } catch (Throwable e) {
        closeChannel(channel);
    }
}
}

```

`completed` 方法的内容跟 `NIOAcceptor` 的 `accept()` 函数的作用差不多，就对建立连接后的 `socket` 做下一步操作，而 AIO 比 NIO 还要略微简单些（NIO 还要做一次 `sub reactor` 的再分配。），AIO 只要调用 `FrontendConnection.register()` 向就可以了。

另外,AsynchronousServerSocketChannel 的 accept 方法注册的 completionHandler 只能被一次连接接入事件调用, 并且不能同时注册多个 pending 的 completionHandler, 否则会抛出 AcceptPendingException。所以当 completionHandler 被回调时, 为了服务器能继续接入新的连接, 要继续调用 AsynchronousServerSocketChannel 的 accept 方法注册一个新的 completionHandler, 用于下一个新连接的接入准备, 所以 completed 方法还要继续调用 pendingAccept()方法

6.4.3 AIOConnector

类声明

AIOConnector 实现 CompletionHandler<V,A>,用作在 connect 事件的用户句柄。根据 AsynchronousSocketChannel.connect()的参数定义, 对 AIOAcceptor 而言, V 已经固定为 Void, A 可以自定义。

```
public final class AIOConnector implements SocketConnector,  
    CompletionHandler<Void, AbstractConnection>{
```

被谁调用

在启动时初始化数据源、HeartBeat 和前端执行 Query 需要新建连接时, 通过 BackendConnectionFactory 的 make 方法中, 调用 connect 进行 handler 设置:

```
((AsynchronousSocketChannel) channel).connect(  
    new InetSocketAddress(dsc.getIp(), dsc.getPort()),  
    detector, (CompletionHandler) MycatServer.getInstance()  
        .getConnector());
```

completed 方法

```
@Override  
public void completed(Void result, AbstractConnection attachment) {  
    finishConnect(attachment);  
}  
  
private void finishConnect(AbstractConnection c) {  
    try {  
        if (c.finishConnect()) {
```

```

        NIOProcessor processor = MycatServer.getInstance()
            .nextProcessor();

        c.setProcessor(processor);

        c.register();

    }

} catch (Throwable e) {}

}

```

与 AIOAcceptor 的 completed 方法比较像，对建立连接后的 socket 做下一步操作，只要调用 AbstractConnection.register() 向就可以了。

6.4.4 AIOSocketWR 和 AIOReadHandler

AIOSocketWR 实现了 SocketWR 接口的 asynRead 方法，该方法的调用关系如下图：

```

▼ ● asynRead() : void - org.opencloudb.net.AIOSocketWR
  ▼ ● asynRead() : void - org.opencloudb.net.AbstractConnection
    ► ● completed(Integer, AIOSocketWR) : void - org.opencloudb.net.AIOReadHandler
    ► ● register() : void - org.opencloudb.net.BackendAIOConnection
    ► ● register() : void - org.opencloudb.net.FrontendConnection

```

- 1、前端链路接入后，先发送握手数据包，然后调用 asynRead() 等待读应答握手应答。
- 2、后端链路接入后，调用 asynRead() 等待握手数据包的到来。
- 3、AIOReadHandler 被回调时，继续下一次读。

6.4.5 AIOSocketWR 的 asynRead 方法

这个方法很简单，就是调用 channel 的 read 方法，把 AIOReadHandler 句柄传递过去。

```

@Override

public void asynRead() {

    ByteBuffer theBuffer = con.readBuffer;

    if (theBuffer == null) {

        theBuffer = con.processor.getBufferPool().allocate();

        con.readBuffer = theBuffer;

        channel.read(theBuffer, this, aioReadHandler);

    } else if (theBuffer.hasRemaining()) {

```

```

channel.read(theBuffer, this, aioReadHandler);
} else {
    throw new java.lang.IllegalArgumentException("full buffer to read ");
}
}

```

6.4.6 AIOReadHandler

AIOReadHandler 实现 CompletionHandler<V,A>, 用作在 read 事件的用户句柄回调。根据 AsynchronousSocketChannel.read() 的参数定义，对 AIOReadHandler 而言，V 已经固定为 Integer 类型表示读的字节数，A 可以自定义。

```

class AIOReadHandler implements CompletionHandler<Integer, AIOSocketWR> {
    @Override
    public void completed(final Integer i, final AIOSocketWR wr) {
        if (i > 0) {
            try {
                wr.con.onReadData(i);
                wr.con.asynRead();
            } catch (IOException e) {
                wr.con.close("handle err:" + e);
            }
        } else if (i == -1) {
            wr.con.close("client closed");
        }
    }
}

```

AIOReadHandler 的 completed 方法主要做两件事：

- 1、读 buffer 中的内容。
- 2、继续注册下一次读的回调句柄。

6.4.7 AIOSocketWR 和 AIOWriteHandler

AIOSocketWR 实现了 SocketWR 接口的 doNextWriteCheck 方法, doNextWriteCheck 又调用

asynWrite, 该方法的调用有两类:

- asynWrite(ByteBuffer) : void - org.opencloudb.net.AIOSocketWR
- ▼ ■ write0() : boolean - org.opencloudb.net.AIOSocketWR (2 matches)
 - ▼ ● doNextWriteCheck() : void - org.opencloudb.net.AIOSocketWR (2 matches)
 - ▶ ◆ onWriteFinished(int) : void - org.opencloudb.net.AIOSocketWR
 - ▶ ●F write(ByteBuffer) : void - org.opencloudb.net.AbstractConnection
 - ▼ ◆ onWriteFinished(int) : void - org.opencloudb.net.AIOSocketWR
 - ▶ ● completed(Integer, AIOSocketWR) : void - org.opencloudb.net.AIOWriteHandler

1.业务线程发起写请求操作, 当显式调用 AbstractConnection 时, 若空闲直接写, 否则放入写队列等待。

```
public void doNextWriteCheck() {
    if (!writing.compareAndSet(false, true)) {
        return;
    }
    boolean noMoreData = this.write0();
    if (noMoreData) {
        if (!con.writeQueue.isEmpty()) {
            this.write0();
        }
    }
}

private boolean write0() {
    ByteBuffer theBuffer = con.writeBuffer;

    if (theBuffer == null || !theBuffer.hasRemaining()) { // writeFinished,但要区分 bufer 是否 NULL, 不 NULL, 要
回收
        if (theBuffer != null) {
            con.recycle(theBuffer);
            con.writeBuffer = null;
        }
        ByteBuffer buffer = con.writeQueue.poll();
```



```

if (buffer != null) {
    if (buffer.limit() == 0) {
        con.recycle(buffer);
        con.writeBuffer = null;
        con.close("quit cmd");
        return true;
    } else {
        con.writeBuffer = buffer;
        asynWrite(buffer);
        return false;
    }
} else {
    writing.set(false);
    return true;
}
} else {
    theBuffer.compact();
    asynWrite(theBuffer);
    return false;
}
}

private void asynWrite(ByteBuffer buffer) {
    buffer.flip();
    this.channel.write(buffer, this, aioWriteHandler);
}

```

2.CompletionHandler 回调句柄中，对返回的 Integer 仅作计数和判断用，不像 read 那样，读出 n bytes 进行 handle 出来。异步写的逻辑是，不断循环，发现 buffer 没有写完，则 compact 后继续写；如果 buffer 已经写完，则 recycle；然后从 writeQueue 中取出其他的 buffer 继续，如果队列中也没有 buffer，则不再循环。

```
protected void onWriteFinished(int result) {  
    con.netOutBytes += result;  
    con.processor.addNetOutBytes(result);  
    con.lastWriteTime = TimeUtil.currentTimeMillis();  
    boolean noMoreData = this.write0();  
    if (noMoreData) {  
        this.doNextWriteCheck();  
    }  
}
```

第 7 章 Mycat 的路由与分发流程

7.1 路由的作用

7.1.1 为什么需要路由

还得从 Mycat 原理上来看（具体见前文 Mycat 原理）。从原理上来看，可以把 mycat 看成一个 sql 转发器。mycat 接收到前端发来的 sql，然后转发到后台的 mysql 服务器上去执行。但是后面有很多台 mysql 节点（如 dn1, dn2, dn3），该转发到哪些节点呢？这就是路由解析该做的事情了。

路由能保证 sql 转发到正确的节点。转发的范围是刚刚好，不多发也不少发。多发会出现两种问题：浪费性能和找不到表。比如一个 `select * from orders where pro= 'wuhan'` 这个语句，只有 dn1 节点，能查到数据，如果将语句同时转发到 dn1、dn2、dn3 三个节点，这样的范围就多发了，性能上是一种浪费。如果新增了一个节点 dn4，但是 orders 的 datanode 范围只是 dn1,dn2,dn3，如果同时转发到 dn1、dn2、dn3、dn4 四个节点，则发到 dn4 执行时会返回 `table orders not exists`。少发则会出现结果集不全的问题，如 `select * from orders` 如果只转发到 dn1，只会返回 dn1 上的结果集，dn2、dn3 上的结果集得不到。

7.2 路由解析器

7.2.1 解析器选型

解析器指的是 sql 解析器，mycat1.3 之前使用的解析器为 fdb parser(FoundationDB SQL Parser)，从 1.3 开始引入 druid 解析器，从 1.4 开始去掉了 fdbparser，只保留 druidparser 方式。

fdbparser 解析器存在的问题：

- 1、修改解析器源码的门槛太高。使用了 javacc 解析器，如果要修改解析器的源码必须搞清楚 javacc 的原理（修改解析器源码是有时碰到不支持的语法，要修改解析器来支持）。
- 2、没有好的 api 接口获取 ast 语法树中的表名、拆分字段条件等，所以路由解析时的代码很难有好的结构，就是写的很让人看不懂。
- 3、支持的语句太少。如 insert into On duplicate key update....，带注释的 create table 语句不支持，还有很多就不列举了。
- 4、解析性能很差。我们公司的 sql 一般都很长（select 语句），一个长点的 sql 解析花了 3、4 秒解析出 ast 语法树。这个在业务上无法让人忍受（当然，这么慢与我的开发机器有关，2 核 4G 的破机器，如果用好的服务器可能也用不了这么久）。

7.2.2 几种解析器性能对比

选择解析器时考虑从开源项目中找 java 语言开发的 sql 解析器，找到了两种：

Jsqlparser

项目地址：<https://github.com/JSQLParser/JSqlParser>

Druid SQL Parser

<https://github.com/alibaba/druid/wiki/SQL-Parser>

对 fdbparser、JSqlParser、druidparser3 种解析器做性能对比，对同一个 sql 语句，使用 3 种解析器解析出 ast 语法树（这是编译原理上的说法，在 sql 解析式可能就是解析器自定义的 statement 类型），执行 10 万次、100 万次的的时间对比。

```
import java.sql.SQLException;

import net.sf.jsqlparser.JSQLParserException;
import net.sf.jsqlparser.parser.CCJSqlParserUtil;
import net.sf.jsqlparser.statement.Statements;
import io.mycat.parser.SQLParserDelegate;
```

```

import com.alibaba.druid.sql.ast.SQLStatement;

import com.alibaba.druid.sql.dialect.mysql.parser.MySqlStatementParser;

import com.foundationdb.sql.parser.QueryTreeNode;


public class TestParser {

    public static void main(String[] args) {

        String sql = "insert into employee(id,name,sharding_id) values(5, 'wdw',10010)";

        int count = 1000000;

        long start = System.currentTimeMillis();

        System.out.println(start);

        try {

            for(int i = 0; i < count; i++) {

                QueryTreeNode ast = SQLParserDelegate.parse(sql,"utf-8" );

            }

        } catch (SQLException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

        long end = System.currentTimeMillis();

        System.out.println(count + " times parse,fdb cost:" + (end - start) + "ms");


        start = end;

        try {

            for(int i = 0; i < count; i++) {

                Statements stmt = CCJSqlParserUtil.parseStatements(sql);

            }

        } catch (JSQLParserException e) {

            // TODO Auto-generated catch block

```

```

    e.printStackTrace();
}

end = System.currentTimeMillis();

System.out.println(count + " times parse,JSQlParser cost:" + (end - start) + "ms");

start = end;

for(int i = 0; i < count; i++) {

    MySQLStatementParser parser = new MySQLStatementParser(sql);

    SQLStatement statement = parser.parseStatement();

}

end = System.currentTimeMillis();

System.out.println(count + " times parse ,druid cost:" + (end - start) + "ms");

}

}

```

**** 10 万次输出结果: ****

```

100000 times parse,fdb cost:4549ms
100000 times parse,JSQlParser cost:2892ms
100000 times parse ,druid cost:456ms

```

**** 100 万次输出结果: ****

```

1000000 times parse,fdb cost:30280ms
1000000 times parse,JSQlParser cost:18983ms
1000000 times parse ,druid cost:1912ms

```

结论:

10 万次: druid 比 fdbparser 快 10 倍, 比 JSQlParser 快 6 倍。

100 万次: druid 比 fdbparser 快 15 倍, 比 JSQlParser 快近 10 倍。

7.3 druid 路由解析的两种方式

Druid 解析有两种方式：visitor 方式和 statement 方式。

7.3.1 Visitor 方式的用法：

```
String sql = "select * from tableName" ;  
MySQLStatementParser parser = new MySQLStatementParser(sql);  
SQLStatement statement = parser.parseStatement();  
MycatSchemaStatVisitor visitor = new MycatSchemaStatVisitor();  
stmt.accept(visitor);
```

经过上面的步骤后，你可以很方便的从 visitor 中获取表名、条件、表别名 map、字段列表、值类表等信息。用这些信息就可以做路由计算了。

7.3.2 Statement 方式的用法

```
String sql = "select * from tableName" ;  
MySQLStatementParser parser = new MySQLStatementParser(sql);  
SQLStatement statement = parser.parseStatement();  
SQLSelectStatement selectStmt = (SQLSelectStatement) statement;
```

然后就可以从 selectStmt 里面得到想要的信息去了。

如果 sql = "delete from tableName" ;

就要转型为 MySQLDeleteStatement

MySQLDeleteStatement deleteStmt = (MySQLDeleteStatement) statement。

7.3.3 改写 sql

支持 insert into ... values (),()...语句

要支持该语句需要 DruidParser 在 statementParse 的过程中将 sql 做拆分，根据拆分字段的值，将一个 insert 语句拆分成多个 insert 语句，然后分别发到对应的分片执行。

做法：操作 MySQLInsertStatement，获取里面的 valuesList，根据拆分字段计算，把一个 valuesList 拆分成多个 valuesList（每个 dataNode 对应一个 valuesList）。

具体见 DruidInsertParser 类中的 parserBatchInsert 方法。如下：

```
List<ValuesClause> valueClauseList = insertStmt.getValuesList();
```

```

Map<Integer,List<ValuesClause>> nodeValuesMap = new HashMap<Integer,List<ValuesClause>>();

TableConfig tableConfig = schema.getTables().get(tableName);

AbstractPartitionAlgorithm algorithm = tableConfig.getRule().getRuleAlgorithm();

for(ValuesClause valueClause : valueClauseList) {
    if(valueClause.getValues().size() != columnNum) {
        String msg = "bad insert sql columnSize != valueSize:"
            + columnNum + " != " + valueClause.getValues().size()
            + "values:" + valueClause;

        LOGGER.warn(msg);

        throw new SQLNonTransientException(msg);
    }

    SQLExpr expr = valueClause.getValues().get(shardingCollIndex);

    String shardingValue = null;

    if(expr instanceof SQLIntegerExpr) {
        SQLIntegerExpr intExpr = (SQLIntegerExpr)expr;

        shardingValue = intExpr.getNumber() + "";
    } else if (expr instanceof SQLCharExpr) {
        SQLCharExpr charExpr = (SQLCharExpr)expr;

        shardingValue = charExpr.getText();
    }

    Integer nodeIndex = algorithm.calculate(shardingValue);

    //没找到插入的分片

    if(nodeIndex == null) {
        String msg = "can't find any valid datanode :" + tableName
            + " -> " + partitionColumn + " -> " + shardingValue;

        LOGGER.warn(msg);

        throw new SQLNonTransientException(msg);
    }
}

```

```

if(nodeValuesMap.get(nodeIndex) == null) {
    nodeValuesMap.put(nodeIndex, new ArrayList<ValuesClause>());
}
nodeValuesMap.get(nodeIndex).add(valueClause);
}

RouteResultsetNode[] nodes = new RouteResultsetNode[nodeValuesMap.size()];
int count = 0;
for(Map.Entry<Integer,List<ValuesClause>> node : nodeValuesMap.entrySet()) {
    Integer nodeIndex = node.getKey();
    List<ValuesClause> valuesList = node.getValue();
    insertStmt.setValuesList(valuesList);
    nodes[count++] = new RouteResultsetNode(tableConfig.getDataNodes().get(nodeIndex),
        rrs.getSqlType(),insertStmt.toString());
}
rrs.setNodes(nodes);
rrs.setFinishedRoute(true);

```

Select 语句添加 limit

见 DruidSelectParser 类中的以下方法：

```

if(isNeedChangeLimit(rrs, schema)) {
    Limit changedLimit = new Limit();
    changedLimit.setRowCount(new SQLIntegerExpr(limitStart + limitSize));

    if(offset != null) {
        if(limitStart < 0) {
            String msg = "You have an error in your SQL syntax; check the manual that " +
                "corresponds to your MySQL server version for the right syntax to use near '" + limitStart + "'";
            throw new SQLNonTransientException(ErrorCode.ER_PARSE_ERROR + " - " + msg);
        }
    }
}

```



```

    } else {
        changedLimit.setOffset(new SQLIntegerExpr(0));
        //TODO
    }
}

mysqlSelectQuery.setLimit(changedLimit);
rrs.changeNodeSqlAfterAddLimit(SQLParserUtils.toMySqlString(stmt));
//    rrs.setSqlChanged(true);
}

```

Select 语句加减 order by

跟加 limit 类似：

```
mysqlSelectQuery.setOrderBy(orderBy);
```

要去掉 order by, mysqlSelectQuery.setOrderBy(null);

Select 语句加减 group by

跟加 limit 类似：

```
mysqlSelectQuery.setGroupBy(groupBy);
```

去掉 group by, mysqlSelectQuery.setGroupBy(null);

Insert 语句加自增长主键

操作 MySQLInsertStatement

```
insertStmt.getColumns().add(column);
```

```
insertStmt.getValues().addValue(value);
```

7.3.4 其他改写

其他改写还有很多，可以通过 druid 的 api 自由发挥。

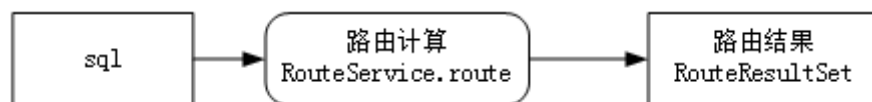
7.4 路由计算

7.4.1 路由计算接口

路由计算的入口方法为 io.mycat.route.RouteService 类中的 route 方法。方法签名如下：

```
public RouteResultset route(SystemConfig sysconf, SchemaConfig schema,int sqlType, String stmt, String charset, ServerConnection sc) throws SQLNonTransientException
```

7.4.2 路由计算简要数据流图



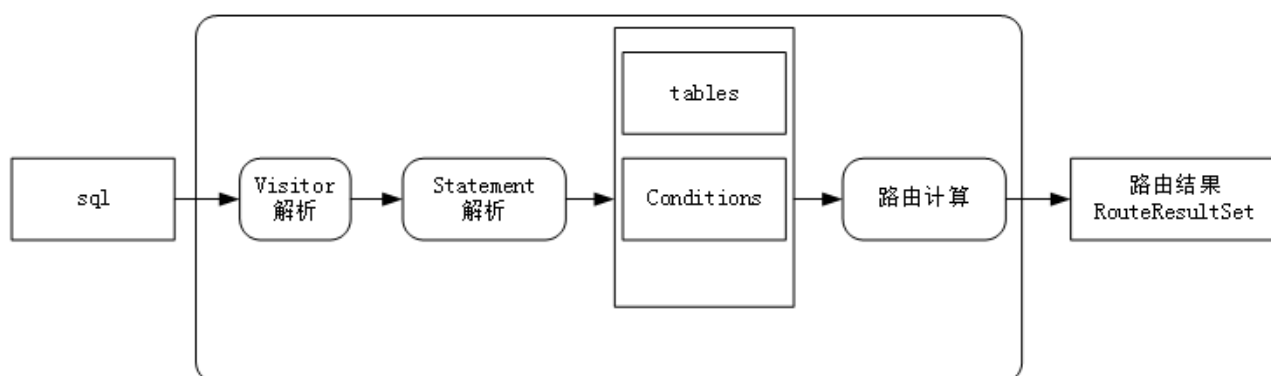
说明：输入一个 sql，经过路由计算，输出路由结果。

该图实际是对路由接口的一个简化。路由接口中还包含 SystemConfig、SchemaConfig、sqlType、charset、ServerConnection 等其他输入参数，但对于路由计算来说，这些参数都不是最主要参数。如 SystemConfig、SchemaConfig 两个参数，完全可以不用传入，我们可以直接用其他方式获取，如：

```
SystemConfig sysconf =MycatServer.getInstance().getConfig().getSystem();  
SchemaConfig schema = MycatServer.getInstance().getConfig().getSchemas().get(sc.getSchema());
```

这些参数可以理解为一些次要参数（对路由计算本身次要，但是对其他流程有用，至于具体用处此处不做为重点），另一个需要传这些参数的原因，路由计算的流程比较长，要经过很多个方法的调用，如果每个方法中都通过曲折的途径去计算获取这些参数也是一种性能损耗。

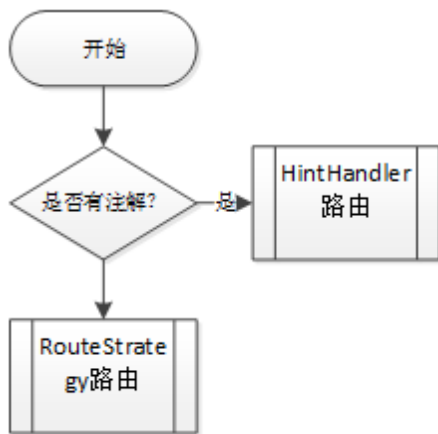
7.4.3 路由计算分解数据流图



其中 conditions 中每个 condition 为<表名、字段名、字段值>的 3 元组。

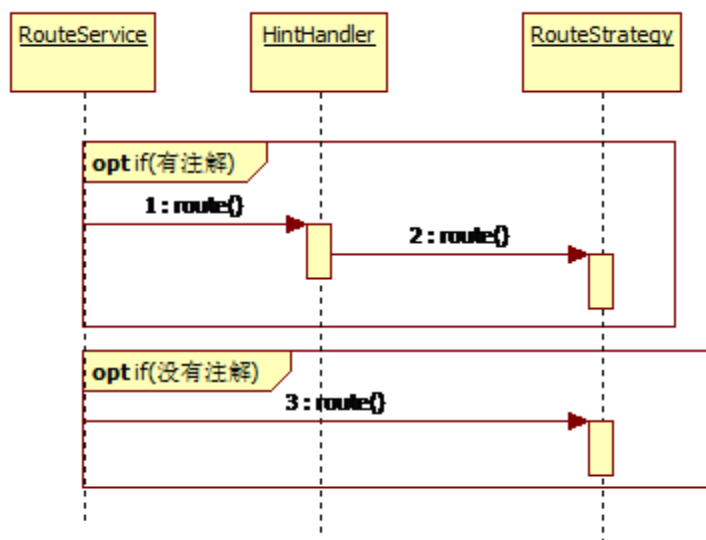
7.4.4 路由计算流程

路由解析总体流程



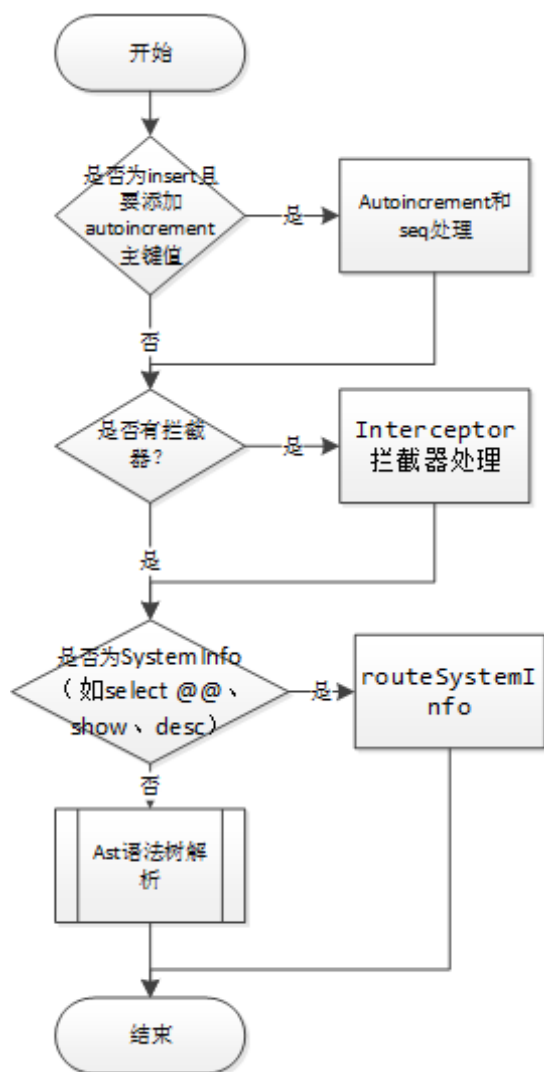
其中 RouteStrategy 路由为子流程，见 RouteStrategy 路由子流程对其展开讲解。HintHandler 路由也是子流程，但非主流程故本文不做重点讲解。

路由解析序列图



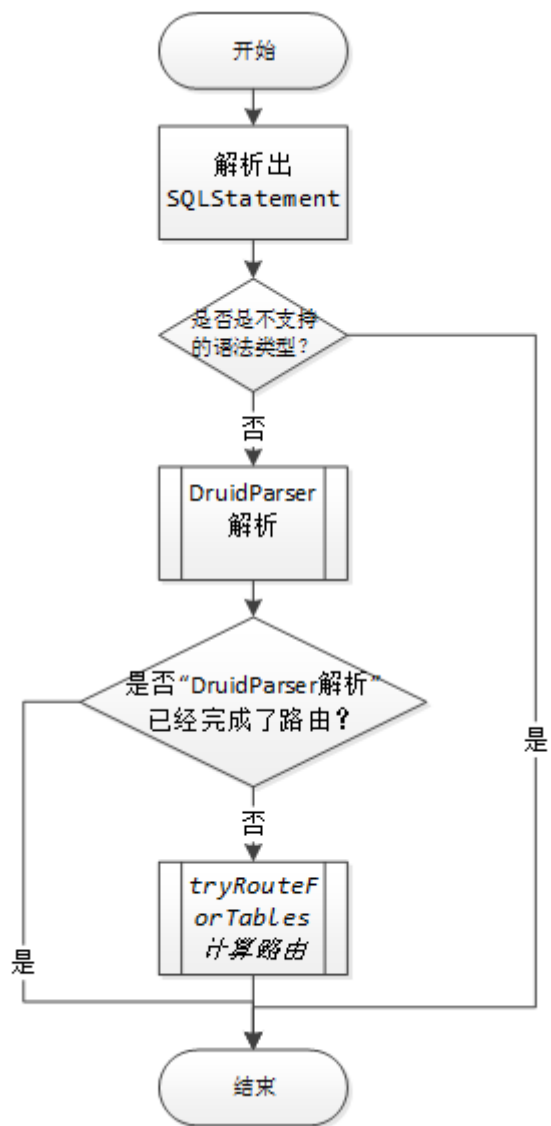
路由解析入口都从 RouteService 类的 route 方法进入，然后根据是否有注解决定是走 HintHandler 还是 RouteStrategy 进行路由解析。

RouteStrategy 路由子流程

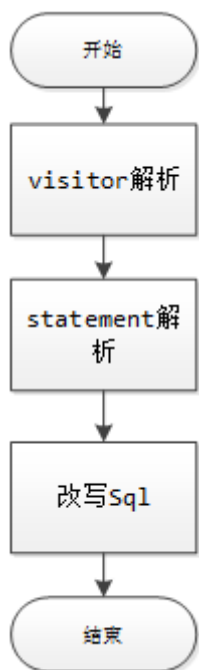


该流程是 fdbparser 和 druidparser 两种解析策略的公共流程。该流程封装在 AbstractRouteStrategy 类的 route 方法中，相当于两种策略的模板方法。子流程“Ast 语法树解析”对应 routeNormalSqlWithAST 方法，下一节将对 ast 语法树解析流程再展开讲解（以 DruidMysqlRouteStrategy 策略类为例）。

DruidMysqlRouteStrategy 的 AST 语法树解析流程



DruidParser 解析子流程



此处 DruidParser 解析的含义说明：DruidParser 解析指的是利用 ast 语法树（SQLStatement，这是 druid 解析器已经解析出来的）解析出表名、条件表达式、字段列表、值列表等信息，用于我们计算路由的过程。

该流程封装在 DefaultDruidParser 类的 parser 方法中。

7.5 路由计算的核心要素

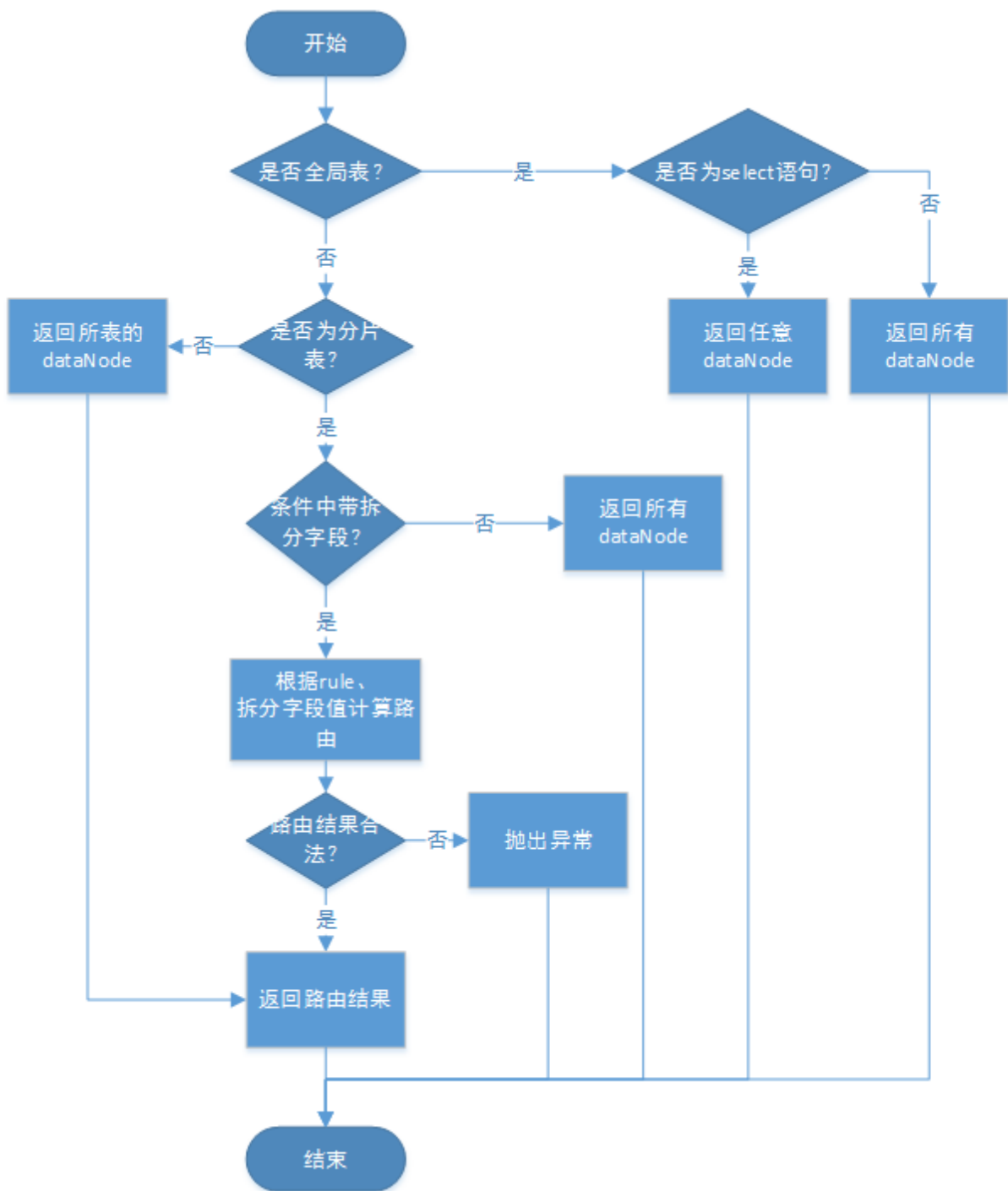
- 1、sql 中包含的表名
- 2、sql 中包含的条件（Conditons），每个 Condition 是一个<表名、字段名、字段值>的 3 元组。
- 3、表对应的 schema。
- 4、表是否分片，如果分片，分片字段是什么？分片算法是什么？第 4 点的信息都可以根据第 3 条计算获得。

有以上一些数据就能计算出路由，所以路由计算需要解决以下问题：

从 sql 语句中提取出表名、条件（字段、字段所属表、字段值）。有了表名、条件，再根据表的分片规则就可以计算出准确的路由了。

7.6 单个表的路由计算

7.6.1 单表路由计算流程



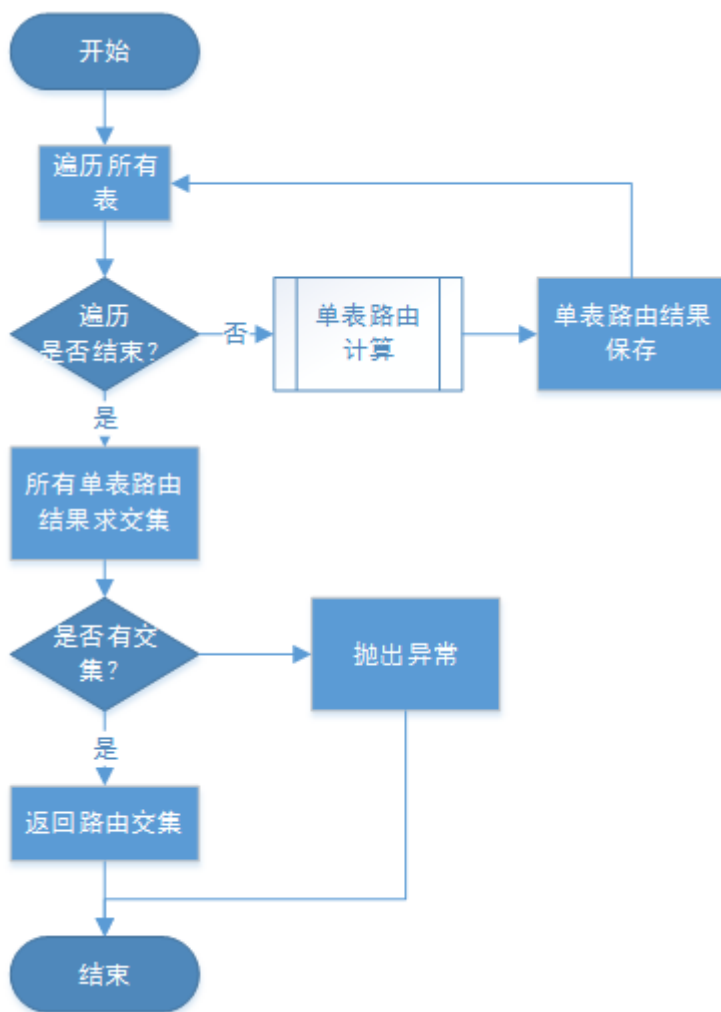
无表语句的路由计算

如 select 1 语句，返回 schema 的任意一个 dataNode 即可。

```
//没有 from 的 select 语句或其他
if(druidParser.getCtx().getTables().size() == 0) {
    return RouterUtil.routeToSingleNode(rrs, schema.getRandomDataNode(),druidParser.getCtx().getSql());
}
```

7.7 多个表的路由计算

7.7.1 多表路由计算流程



多表路由计算中有子流程“单表路由计算”，这个子流程引用了上面的单表路由计算流程。

7.8 全局表的路由计算

全局表 insert、update 语句：路由到所有节点。

全局表 select 语句：路由到任意一个节点。

```
if(tc.isGlobalTable()){//全局表
    if(isSelect){
        // global select ,not cache route result
        rrs.setCacheAble(false);
        return routeToSingleNode(rrs, tc.getRandomDataNode(),ctx.getSql());
    } else {
        return routeToMultiNode(false, rrs, tc.getDataNodes(), ctx.getSql());
    }
}
```



```
}  
  
}
```

7.9 or 语句的路由计算

or 语句的路由需要特殊设计和处理，如果使用一般的计算流程，会出现逻辑错误，导致查询结果错误。

如下面的场景：

travelrecord 表为分片表，其按照 id 范围分片，id 在 1—2000000 范围内在第一分片，id 在 2000001—5000000 在第二分片，对于 `select * from student where id = 1 or 1=1`；如果按照常规的计算方式，只能路由到第一分片，这样查询到的结果就是错误的。

7.9.1 or 语句问题解决方案思想——等价替换

解决 or 语句的路由的基本思想是等价替换。

1、使用 union 语句拆分 or 语句的等价替换

这个等价替换应该是大家都知道的。

`Select * from travelrecord where id = 1 or id = 5000001` 等价于以下语句：

`Select * from travelrecord where id = 1 union Select * from travelrecord where id = 5000001`

2、Union 语句的结果集并集 等价于路由的并集

这个等价没有明确的理论基础，但是我们可以反证法证明：

如果路由集合不同，那么结果集必然不同，所以结果集相同，路由集合必然相同。

`Select * from travelrecord where id = 1 or id = 5000001` 的路由集合

等价于 `Select * from travelrecord where id = 1` 的路由集合与 `Select * from travelrecord where id = 5000001` 的路由集合的并集。

最终演变成对 `Select * from travelrecord where id = 1` 和 `Select * from travelrecord where id = 5000001` 两个语句分别求路由，然后取并集。

7.9.2 or 语句路由解析数据结构分解

每碰到一个 where 条件，如果这个 where 条件中有 or，就把整个 where 条件作为一个单元 WhereUnit，如果这个 WhereUnit 永真（类似 `or 1=1`，`2>1` 之类的），抛弃（抛弃 where 条件后就是全路由，如 `select *`

from tableName,不带任何条件，就是路由到所有节点）。每个 WhereUnit 根据 or 拆分成多个 splitedExpr，构成 splitedExprList。每个 splitedExpr 中都是一些 and 相连的条件（如 classId= 1 and age >20）。

WhereUnit 拆分时使用逐步分解的过程，因为一个 where 条件中可能有多个 or，每个 or 都有 left 表达式和 right 表达式，left 和 right 中必然有一个是不可再拆的，而另一个可能还可再拆，所以逐步拆分，直到不可再拆分（没有了 or）。

碰到 or 语句构造 WhereUnit 的逻辑如下：

见 MycatSchemaStatVisitor 类。

```
@Override
public boolean visit(SQLBinaryOpExpr x) {
    x.getLeft().setParent(x);
    x.getRight().setParent(x);

    switch (x.getOperator()) {
        case Equality:
        case LessThanOrEqualOrGreaterThan:
        case Is:
        case IsNot:
            handleCondition(x.getLeft(), x.getOperator().name, x.getRight());
            handleCondition(x.getRight(), x.getOperator().name, x.getLeft());
            handleRelationship(x.getLeft(), x.getOperator().name, x.getRight());
            break;
        case BooleanOr:
            //永真条件，where 条件抛弃
            if(!RouterUtil.isConditionAlwaysTrue(x)) {
                hasOrCondition = true;
                WhereUnit whereUnit = new WhereUnit(x);
                whereUnits.add(whereUnit);
            }
    }
```

```

        return false;

    case Like:

    case NotLike:

    case NotEqual:

    case GreaterThan:

    case GreaterThanOrEqual:

    case LessThan:

    case LessThanOrEqual:

    default:

        break;

    }

    return true;

}

```

分解 or 语句的逻辑如下：

见 MycatSchemaStatVisitor 类。

```

/**
 * 分解条件
 */
public List<List<Condition>> splitConditions() {

    //按照 or 拆分

    for(WhereUnit whereUnit : whereUnits) {

        splitUntilNoOr(whereUnit);

    }

    //拆分后的条件块解析成 Condition 列表

    for(WhereUnit whereUnit : whereUnits) {

        List<List<Condition>> list = this.getConditionsFromWhereUnit(whereUnit);

        whereUnit.setConditionList(list);

    }

}

```

```

//多个 WhereUnit 组合:多层集合的组合
return getMergedConditionList();
}

/**
 * 条件合并: 多个 WhereUnit 中的条件组合
 * @return
 */
private List<List<Condition>> getMergedConditionList() {
    List<List<Condition>> mergedConditionList = new ArrayList<List<Condition>>();
    if(whereUnits.size() == 0) {
        return mergedConditionList;
    }
    mergedConditionList.addAll(whereUnits.get(0).getConditionList());

    for(int i = 1; i < whereUnits.size(); i++) {
        mergedConditionList = merge(mergedConditionList, whereUnits.get(i).getConditionList());
    }
    return mergedConditionList;
}

/**
 * 两个 list 中的条件组合
 * @param list1
 * @param list2
 * @return
 */
private List<List<Condition>> merge(List<List<Condition>> list1, List<List<Condition>> list2) {

```

```

if(list1.size() == 0) {
    return list2;
} else if (list2.size() == 0) {
    return list1;
}

List<List<Condition>> retList = new ArrayList<List<Condition>>();
for(int i = 0; i < list1.size(); i++) {
    for(int j = 0; j < list2.size(); j++) {
        List<Condition> listTmp = new ArrayList<Condition>();
        listTmp.addAll(list1.get(i));
        listTmp.addAll(list2.get(j));
        retList.add(listTmp);
    }
}
return retList;
}

```

```

private List<List<Condition>> getConditionsFromWhereUnit(WhereUnit whereUnit) {

```

```

    List<List<Condition>> retList = new ArrayList<List<Condition>>();

```

//or 语句外层的条件:如 where condition1 and (condition2 or condition3),condition1 就会在外层条件中,因为之前提取

```

    List<Condition> outSideCondition = new ArrayList<Condition>();

```

```

    outSideCondition.addAll(conditions);

```

```

    this.conditions.clear();

```

```

    for(SQLExpr sqlExpr : whereUnit.getSplitedExprList()) {

```

```

        sqlExpr.accept(this);

```

```

        List<Condition> conditions = new ArrayList<Condition>();

```

```

        conditions.addAll(getConditions());
    }
}

```

```

        conditions.addAll(outSideCondition);

        retList.add(conditions);

        this.conditions.clear();
    }

    return retList;
}

/**
 * 递归拆分 OR
 *
 * @param whereUnit
 * TODO:考虑嵌套 or 语句, 条件中有子查询、 exists 等很多种复杂情况是否能兼容
 */
private void splitUntilNoOr(WhereUnit whereUnit) {
    SQLBinaryOpExpr expr = whereUnit.getCanSplitExpr();

    if(expr.getOperator() == SQLBinaryOperator.BooleanOr) {
//        whereUnit.addSplitedExpr(expr.getRight());
        addExprIfNotFalse(whereUnit, expr.getRight());

        if(expr.getLeft() instanceof SQLBinaryOpExpr) {
            whereUnit.setCanSplitExpr((SQLBinaryOpExpr)expr.getLeft());

            splitUntilNoOr(whereUnit);
        } else {
            addExprIfNotFalse(whereUnit, expr.getLeft());
        }
    } else {
        addExprIfNotFalse(whereUnit, expr);
    }
}
}

```

```
private void addExprIfNotFalse(WhereUnit whereUnit, SQLExpr expr) {
    //非永假条件加入路由计算
    if(!RouterUtil.isConditionAlwaysFalse(expr)) {
        whereUnit.addSplitedExpr(expr);
    }
}
}
```

7.10 系统语句的路由计算

主要有 select @@xxx、show 语句、desc 等语句。

比如：show tables;

show full tables from databaseName;

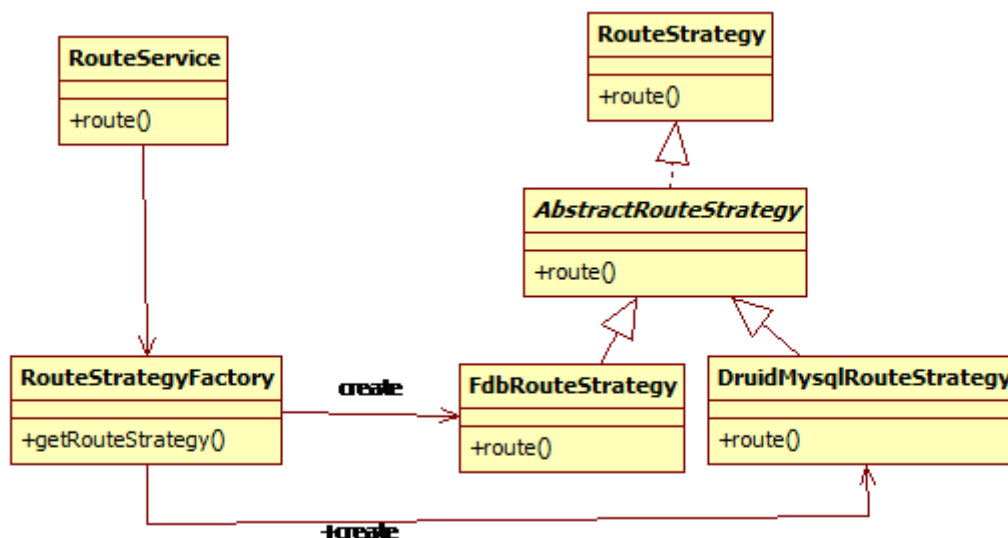
show fields from tableName;

show variables;

这些语句暂时没有使用 sql 解析器进行解析，而是通过字符串解析来特殊处理的，可以考虑使用。

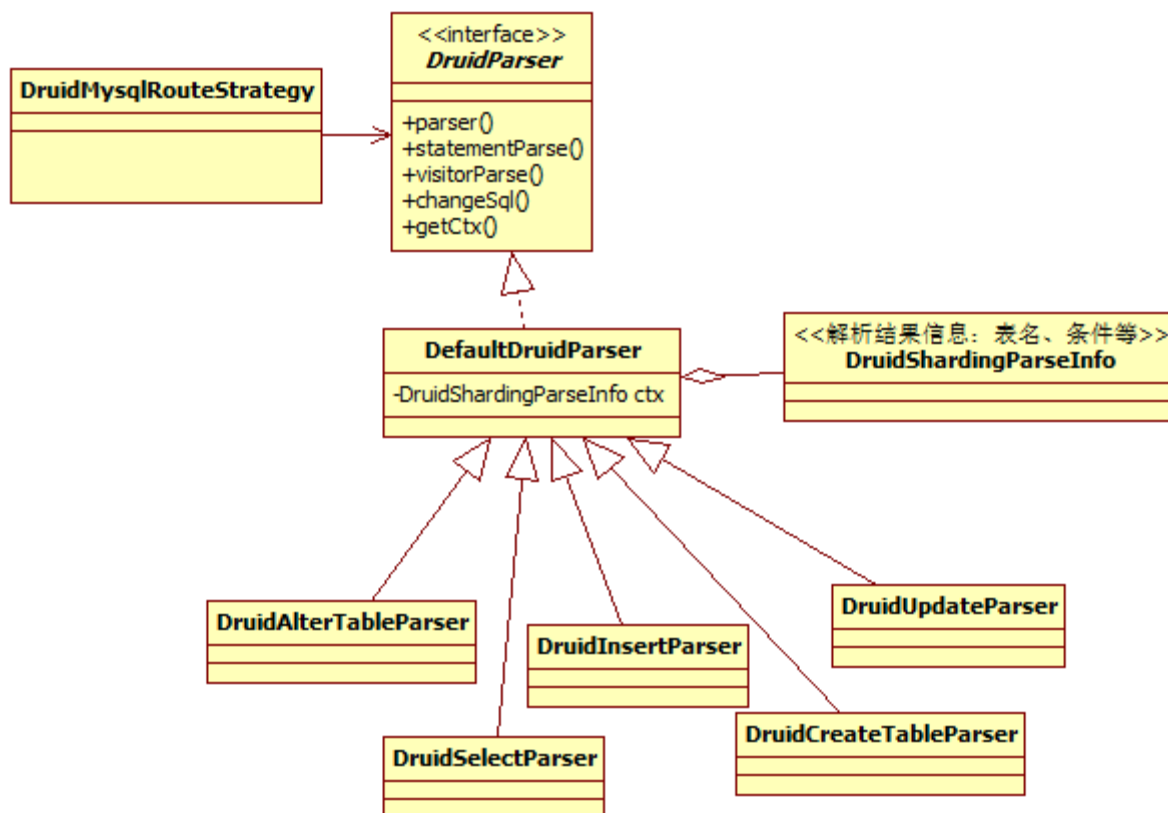
7.11 相关类图和序列图

7.11.1 路由策略相关类图



路由解析使用了策略模式，每种解析器实现一种路由策略。还可以继续扩展，如 Druid 解析再细分 Mysql、postgresql、oracle 等实现策略。本次只实现 druid 解析的 mysql 的策略，其他暂时忽略。

7.11.2 Druid 语法树解析相关类图



类图说明：DruidMysqlRouteStrategy 会根据解析出来的 Statement（AST 语法树）来调用相应的解析器进行解析，解析后的结果会存放到 DruidShardingParseInfo 类中（解析结果信息：表名、条件等），用于后面计算路由。

DruidParser 接口方法介绍（见表 1）。

DruidParser 接口有一个默认实现 DefaultDruidParser，该类相当于一个模板类，parser 方法是其模板方法。模板方法规定了解析步骤：visitorParse、statementParse、changeSql、ctx.setSql(stmt.toString())4 个步骤挨个执行。

所有的子类都继承自该模板类。

Druid 对 SQLStatement 解析时，大多数类型的 statement 通过 visitorParse 这一个方法解析完就得到了我们计算分库路由的所有信息（表名、条件字段等），如果 visitorParse 后还有信息没解析出来，就通过 statementParse，通过这两种方式的解析之后，所有的路由需要的信息都会得到。

7.11.3 每种 Statement 是否必须有一个 DruidParser 的实现类

Druid 的 SQLStatement 有很多子类，如下图，我们是否需要每种 statement 都实现一个子类呢？不需要都实现，一般的 statement 我们使用 visitorParse 方式解析就能得到我们进行路由的所有信息了，visitorParse 在模板类 DefaultDruidParser 中已经有了统一的实现。如果没有特殊需求的，让他走默认的 DefaultDruidParser 解析足矣。

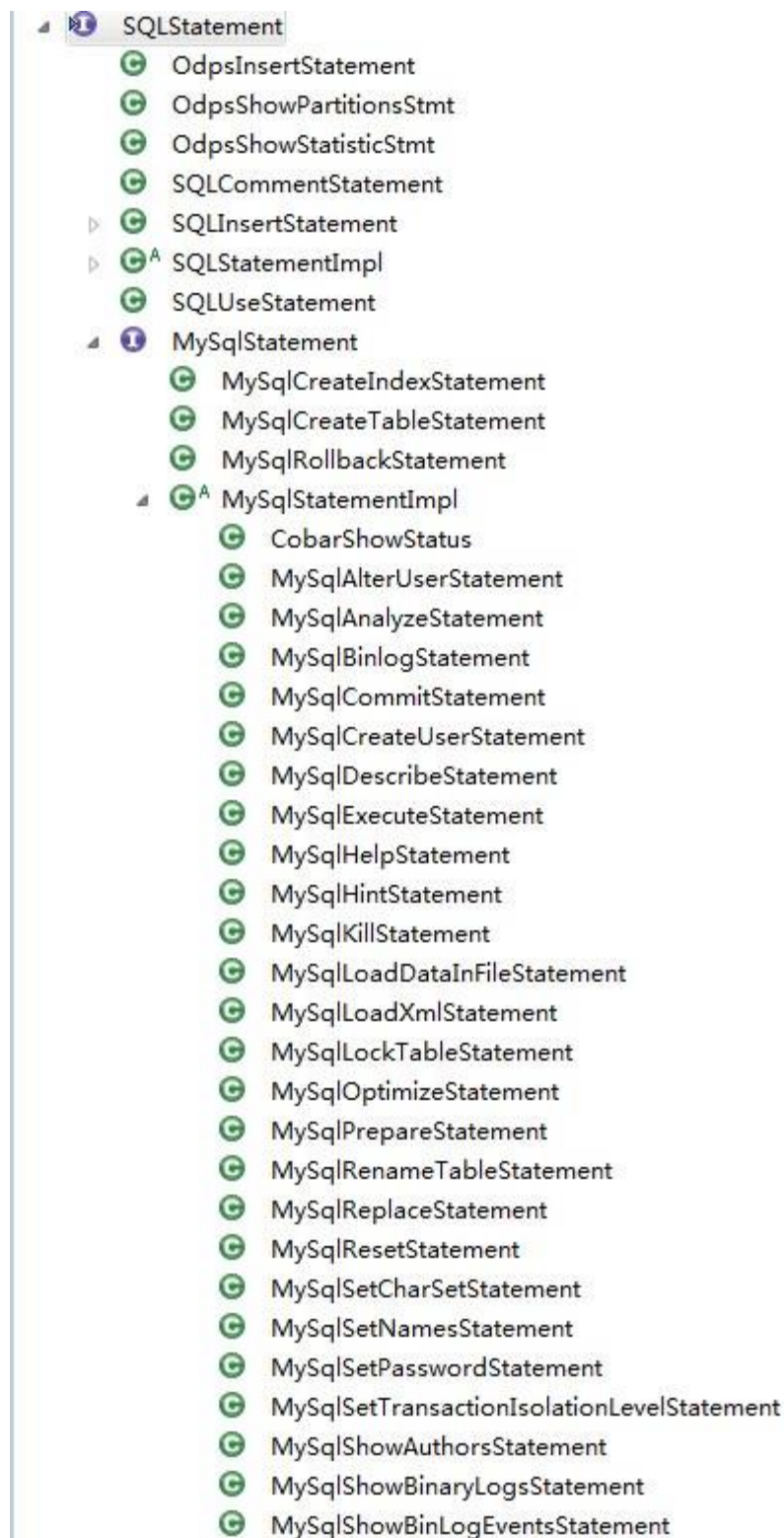


表 1 DruidParser 接口方法介绍

方法名	用途
parser	解析的入口方法
visitorParse	通过 visitor 解析，可以很方便的获取到表名、条件、字段列表、值列表等 对各种语句的 statement 都适用
visitorParse	statement 方式解析。子类覆盖该方法一般是将 SQLStatement 转型后再解析 (如转型为 MySQLInsertStatement)
changeSql	该方法用来改写 sql。如 select 语句加 limit，insert 语句加自增长值等。 主要是为了代码结构化，实际你完全可以把这里面的工作放到 statementParse 中来做
getCtx	获取解析结果。返回 DruidShardingParseInfo 对象。该对象包含解析到的表名列表 条件列表等信息。用于后续计算路由

7.12 路由解析过程中的一些控制变量

RouteResultset 是路由解析的最终返回值类型，该类中包含一些比较关键的参数，现进行列举说明。

7.12.1 isFinishedRoute

//是否完成了路由

```
private boolean isFinishedRoute = false;
```

该变量能控制路由解析流程，由于各种语句的解析流程不可能完全一样，有些简单的可能很快就解析完，直接返回路由结果，有些可能需要经过很复杂的计算才能完成，对于一些能够提前计算出路由结果的，为了防止后面的流程再做一些无用的计算，提高性能，所以设置 setFinishedRoute(true),进入下一个流程计算时，如果判断已经计算完成的，直接返回。

```
//路由计算已经完成的，直接返回
```

```
if(rrs.isFinishedRoute()) {
```

```
return rrs;  
}
```

7.12.2 canRunInReadDB

该变量能控制 mycat 的事务，前提是需要连接的客户端设置了 autocommit=false。

7.12.3 cacheAble

该变量能控制是否缓存路由结果。如果 RouteResultset.setCacheAble(true),在 RouteService 类中会根据此变量来判断是否缓存路由结果，如下：

```
if (rrs!=null && sqlType == ServerParse.SELECT && rrs.isCacheAble()) {  
    sqlRouteCache.putIfAbsent(cacheKey, rrs);  
}
```

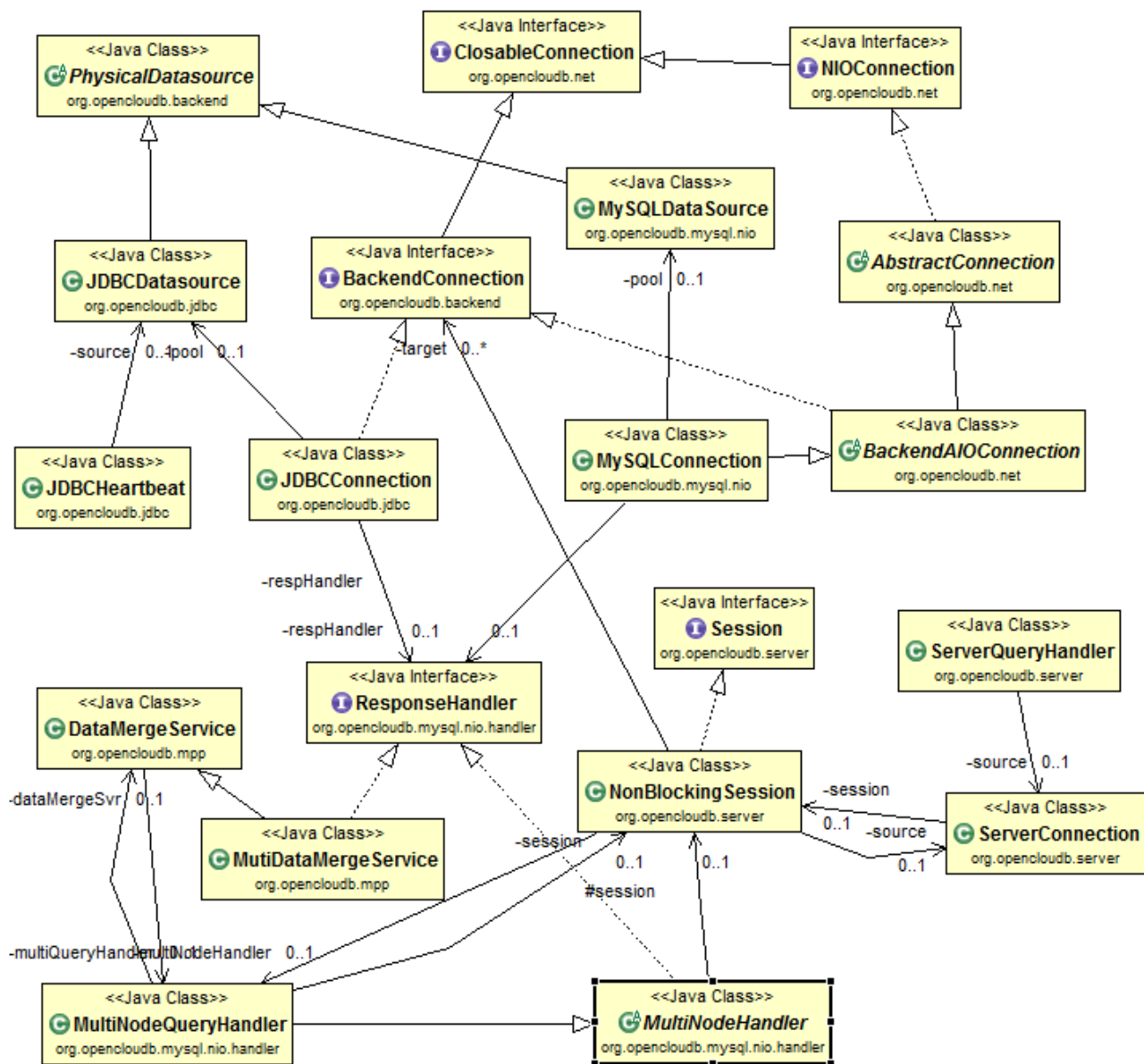
第 8 章 Mycat 的 JDBC 后端框架

8.1 JDBC 方式访问后端数据库

Mycat 对 JDBC 支持部分的代码比较简单，主要实现了下面三个类：

1. JDBCDataSource JDBC 物理数据源；
2. JDBCConnection JDBC 连接类；
3. JDBCHeartbeat JDBC 心跳类。

8.2 JDBC 相关类图



JDBCDataSource 继承 PhysicalDataSource

```
static {  
    // 加载可能的驱动  
    List<String> drivers = Lists.newArrayList("com.mysql.jdbc.Driver",  
        "io.mycat.jdbc.mongodb.MongoDriver", "oracle.jdbc.OracleDriver",  
        "com.microsoft.sqlserver.jdbc.SQLServerDriver", "org.apache.hive.jdbc.HiveDriver", "com.ibm.db2.jcc.DB2Driver
```

```

", "org.postgresql.Driver");

    for (String driver : drivers)
    {
        try
        {
            Class.forName(driver);
        } catch (ClassNotFoundException ignored)
        {
        }
    }
}
}

```

创建连接的时候，从配置文件中获取 host,port,dbtype,还有连接数据库的 url,User,Password 。

```

    public void createNewConnection(ResponseHandler handler,String schema) throws IOException {
        DBHostConfig cfg = getConfig();
        JDBCCConnection c = new JDBCCConnection();

        c.setHost(cfg.getHost());
        c.setPort(cfg.getPort());
        c.setPool(this);
        c.setSchema(schema);
        c.setDbType(cfg.getDbType());

        try {
            // TODO 这里应该有个连接池
            Connection con = getConnection();
            // c.setIdleTimeout(pool.getConfig().getIdleTimeout());

            c.setCon(con);

            // notify handler
            handler.connectionAcquired(c);

```

```

    } catch (Exception e) {

        handler.connectionError(e, c);

    }

}

```

获取连接的时候，判断是否配置的初始化语句，如果存在，就执行初始化语句，此功能可用于设置日期格式，字符集等。

Connection getConnection() throws SQLException

```

{
    DBHostConfig cfg = getConfig();

    Connection connection = DriverManager.getConnection(cfg.getUrl(), cfg.getUser(), cfg.getPassword());

    String initSql=getHostConfig().getConnectionInitSql();

    if(initSql!=null&&"".equals(initSql)) //初始化语句是否存在
    {
        Statement statement =null;

        try
        {
            statement = connection.createStatement();

            statement.execute(initSql);

        }finally
        {
            if(statement!=null)
            {
                statement.close();
            }
        }
    }

    return connection;
}

```

mycat 又从哪里创建 JDBCDataSource 的呢？

请看 io.mycat.ConfigInitializer.

判断是否 dbType 是 mysql 并且 dbDriver 是 native,使用 MySQLDataSource 连接后台数据库, 如果 dbDriver 是 jdbc 就使用 JDBCDataSource 连接后台数据库, 否则抛出异常。

```
private PhysicalDatasource[] createDataSource(DataHostConfig conf,
String hostName, String dbType, String dbDriver,
DBHostConfig[] nodes, boolean isRead) {
PhysicalDatasource[] dataSources = new PhysicalDatasource[nodes.length];
if (dbType.equals("mysql") && dbDriver.equals("native")) {
    for (int i = 0; i < nodes.length; i++) {
        nodes[i].setIdleTimeout(system.getIdleTimeout());
        MySQLDataSource ds = new MySQLDataSource(nodes[i], conf, isRead);
        dataSources[i] = ds;
    }

} else if(dbDriver.equals("jdbc"))//是 jdbc 方式
{
    for (int i = 0; i < nodes.length; i++) {
        nodes[i].setIdleTimeout(system.getIdleTimeout());
        JDBCDataSource ds = new JDBCDataSource(nodes[i], conf, isRead);
        dataSources[i] = ds;
    }
} else {
    throw new ConfigException("not supported yet !" + hostName);
}
return dataSources;
}
```

8.4 JDBCConnection

JDBCCConnection 主要做两件事情，就是执行 SQL 语句，然后把执行结果发回给 mpp(SQL 合并引擎,mycat 处理多节点结果集排序，分组，分页),需要实现 ResponseHandler 的接口。

下面来分析下执行 SQL 语句的代码：

创建线程 Runnable，在线程中执行 executeSQL 的方法，并把线程放入 MycatServer 的线程池中执行，据测试，比不用线程方式执行 SQL 语句效率提高 20%-30%。

```
public void execute(final RouteResultsetNode node, final ServerConnection source,
    final boolean autocommit) throws IOException {
    Runnable runnable=new Runnable()
    {
        @Override
        public void run()
        {
            try
            {
                executeSQL(node, source, autocommit);
            } catch (IOException e)
            {
                throw new RuntimeException(e);
            }
        }
    };

    MycatServer.getInstance().getBusinessExecutor().execute(runnable);
}
```

执行 SQL 语句的过程，先判断是 select,或 show 语句还是 ddl 语句。

1.如果是 show 指令，并且不是 mysql 数据库，执行 ShowVariables.execute，构造 mysql 的固定信息包。

2.如果是 SELECT CONNECTION_ID()语句，执行 ShowVariables.justReturnValue，也是构造 mysql 的固定信息包。

3.如果是 SELECT 语句，执行并且有返回结果数据集。

4.如果是 DDL 语句，执行并且返回 OkPacket。

```
private void executeSQL(RouteResultsetNode rrn, ServerConnection sc,
    boolean autocommit) throws IOException {
    String orgin = rrn.getStatement();
    if (!modifiedSQLExecuted && rrn.isModifySQL()) {
        modifiedSQLExecuted = true;
    }

    try {
        if (!this.schema.equals(this.oldSchema)) { //判断
            con.setCatalog(schema);
            this.oldSchema = schema;
        }

        if (!this.isSpark) { //spark sql ,hive 不支持事务
            con.setAutoCommit(autocommit);
        }

        int sqlType = rrn.getSqlType();
        //判断是否是查询或者 mysql 的 show 指令
        if (sqlType == ServerParse.SELECT || sqlType == ServerParse.SHOW ) {
            if ((sqlType == ServerParse.SHOW) && (!dbType.equals("MYSQL"))){
                ShowVariables.execute(sc, orgin, this); //show 指令的返回结果
            } else if ("SELECT CONNECTION_ID()".equalsIgnoreCase(orgin))
            {
                ShowVariables.justReturnValue(sc, String.valueOf(sc.getId()), this);
            }

            else {
```

```

        ouputResultSet(sc, orgin);//执行 select 语句， 并处理结果集
    }
} else { //sql ddl 执行
    executedddl(sc, orgin);
}

} catch (SQLException e) { //异常处理
    String msg = e.getMessage();
    ErrorPacket error = new ErrorPacket();
    error.packetId = ++packetId;
    error.errno = e.getErrorCode();
    error.message = msg.getBytes();
    //触发错误数据包的响应事件
    this.respHandler.errorResponse(error.writeToBytes(sc), this);
} finally {
    this.running = false;
}
}
}

```

ouputResultSet(sc, orgin);//执行 select 语句， 并处理结果集。

```

    stmt = con.createStatement();
rs = stmt.executeQuery(sql); 执行 sql 语句

List<FieldPacket> fieldPks = new LinkedList<FieldPacket>();//创建字段列表
    //把字段的元数据转换为 mysql 的元数据并放入 fieldPks 中， 主要是数据类型
ResultSetUtil.resultSetToFieldPacket(sc.getCharset(), fieldPks, rs, this.isSpark);

```

把字段信息封装成 mysql 的网络封包。

```

int columnCount = fieldPks.size();

ByteBuffer byteBuf = sc.allocate();

ResultSetHeaderPacket headerPkg = new ResultSetHeaderPacket();

headerPkg.fieldCount = fieldPks.size();

headerPkg.packetId = ++packetId;


byteBuf = headerPkg.write(byteBuf, sc, true);

byteBuf.flip();

byte[] header = new byte[byteBuf.limit()];

byteBuf.get(header);

byteBuf.clear();

List<byte[]> fields = new ArrayList<byte[]>(fieldPks.size());

Iterator<FieldPacket> itor = fieldPks.iterator();

while (itor.hasNext()) {

    FieldPacket curField = itor.next();

    curField.packetId = ++packetId;

    byteBuf = curField.write(byteBuf, sc, false);

    byteBuf.flip();

    byte[] field = new byte[byteBuf.limit()];

    byteBuf.get(field);

    byteBuf.clear();

    fields.add(field);

    itor.remove();

}

EOFPacket eofPkg = new EOFPacket();

eofPkg.packetId = ++packetId;

byteBuf = eofPkg.write(byteBuf, sc, false);

byteBuf.flip();

byte[] eof = new byte[byteBuf.limit()];

```

```

byteBuf.get(eof);

byteBuf.clear();

        //触发收到字段数据包结束的响应事件

this.respHandler.fieldEofResponse(header, fields, eof, this);

```

遍历结果数据集 ResultSet,并把每一条记录封装成一个数据包,数据发送完成, 还需要在封装一个行结束的数据包

```

// output row
while (rs.next()) {

    RowDataPacket curRow = new RowDataPacket(columnCount);

    for (int i = 0; i < columnCount; i++) {

        int j = i + 1;

        curRow.add(StringUtil.encode(rs.getString(j), sc.getCharset()));

    }

    curRow.packetId = ++packetId;

    byteBuf = curRow.write(byteBuf, sc, false);

    byteBuf.flip();

    byte[] row = new byte[byteBuf.limit()];

    byteBuf.get(row);

    byteBuf.clear();

        //触发收到行数据包的响应事件

    this.respHandler.rowResponse(row, this);

}

// end row

eofPkg = new EOFPacket();

eofPkg.packetId = ++packetId;

byteBuf = eofPkg.write(byteBuf, sc, false);

byteBuf.flip();

eof = new byte[byteBuf.limit()];

```

```
byteBuf.get(eof);

sc.recycle(byteBuf);

        //收到行数据包结束的响应处理

this.respHandler.rowEofResponse(eof, this);
```

8.5 JDBCHeartbeat

JDBCHeartbeat 就是定时执行 schema.xml 中 dataHost 的 heartbeat 语句。

在启动的时候判断心跳语句是否为空，如果为空则执行 stop(),后面再执行 heartbeat()方法时，直接返回。

```
public class JDBCHeartbeat extends DBHeartbeat{

    private final ReentrantLock lock;

    private final JDBCDataSource source;

    private final boolean heartbeatnull;

    public JDBCHeartbeat(JDBCDataSource source)

    {

        this.source = source;

        lock = new ReentrantLock(false);

        this.status = INIT_STATUS;

        this.heartbeatSQL = source.getHostConfig().getHeartbeatSQL().trim();

        this.heartbeatnull= heartbeatSQL.length()==0;//判断心跳语句是否为空

    }

    @Override

    public void start()//启动

    {

        if (this.heartbeatnull){

            stop();

            return;

        }

        lock.lock();

        try
```

```

{
    isStop.compareAndSet(true, false);
    this.status = DBHeartbeat.OK_STATUS;
} finally

{
    lock.unlock();
}
}

```

@Override

public void stop()//停止

```

{
    lock.lock();
    try
    {
        if (isStop.compareAndSet(false, true))
        {
            isChecking.set(false);
        }
    } finally
    {
        lock.unlock();
    }
}

```

....

@Override

public void heartbeat()//执行心跳语句

```

{

```

```
if (isStop.get())  
    return;  
  
lock.lock();  
  
try  
{  
    isChecking.set(true);  
  
    try (Connection c = source.getConnection())  
    {  
        try (Statement s = c.createStatement())  
        {  
            s.execute(heartbeatSQL);  
        }  
    }  
  
    status = OK_STATUS;  
  
} catch (SQLException ex)  
{  
    status = ERROR_STATUS;  
}  
finally  
{  
    lock.unlock();  
    this.isChecking.set(false);  
}  
}
```

第 9 章 Mycat 的事务管理机制

9.1 Mycat 事务源码分析

Mycat 的事务相关的代码逻辑，目前的实现方式如下：

用户会话 Session 中设定 autocommit=false，开启一个事务过程，这个会话中随后的所有 SQL 语句进入事务模式，ServerConnection（前端连接）中有一个变量 txInterrupted 控制是否事务异常需要回滚。

当某个 SQL 执行过程中发生错误，则设置 txInterrupted=true，表明此事务需要回滚。

当用户提交事务（commit 指令）的时候，Session 会检查事务回滚变量，若发现事务需要回滚，则取消 Commit 指令在相关节点上的执行过程，返回错误信息，Transaction need rollback，用户只能回滚事务，若所有节点都执行成功，则向每个节点发送 Commit 指令，事务结束。

从上面的逻辑来看，当前 Mycat 的事务是一种弱 XA 的事务，与 XA 事务相似的地方是，只有所有节点都执行成功（Prepare 阶段都成功），才开始提交事务，与 XA 不同的是，在提交阶段，若某个节点宕机，没有手段让此事务在故障节点恢复以后继续执行，从实际的概率来说，这个概率也是很小很小的，因此，当前事务的方式还是能满足绝大多数系统对事务的要求。

另外，Mycat 当前若 XA 的事务模式，相对 XA 还是比较轻量级，性能更好，虽然如此，也不建议一个事务中存在跨多个节点的 SQL 操作问题，这样锁定的资源更多，并发性降低很多。

前端连接中关于事务标记 txInterrupted 的方法片段：

```
public class ServerConnection extends FrontendConnection {  
  
    /**  
     * 设置是否需要中断当前事务  
     */  
    public void setTxInterrupt(String txInterruptMsg) {  
        if (!autocommit && !txInterrupted) {  
            txInterrupted = true;  
            this.txInterruptMsg = txInterruptMsg;  
        }  
    }  
  
    public boolean isTxInterrupted()
```



```

    {
        return txInterrupted;
    }
/**
 * 提交事务
 */
public void commit() {
    if (txInterrupted) {
        writeErrMessage(ErrorCode.ER_YES,
                        "Transaction error, need to rollback.");
    } else {
        session.commit();
    }
}
}

```

SQL 出错时候设置事务回滚标志：

```

public class SingleNodeHandler implements ResponseHandler, Terminatable,
LoadDataResponseHandler {
    private void backConnectionErr(ErrorPacket errPkg, BackendConnection conn) {
        endRunning();

        String errmgs = " errno:" + errPkg.errno + " "
            + new String(errPkg.message);

        LOGGER.warn("execute sql err :" + errmgs + " con:" + conn);
        session.releaseConnectionIfSafe(conn, LOGGER.isDebugEnabled(), false);
        ServerConnection source = session.getSource();
        source.setTxInterrupt(errmgs);
    }
}

```

```

        errPkg.write(source);

        recycleResources();

    }

}

```

Session 提交事务的关键代码：

```

public class NonBlockingSession implements Session {

    public void commit() {

        final int initCount = target.size();

        if (initCount <= 0) {

            ByteBuffer buffer = source.allocate();

            buffer = source.writeToBuffer(OkPacket.OK, buffer);

            source.write(buffer);

            return;

        } else if (initCount == 1) {

            BackendConnection con = target.elements().nextElement();

            commitHandler.commit(con);

        } else {

            if (LOGGER.isDebugEnabled()) {

                LOGGER.debug("multi node commit to send ,total " + initCount);

            }

            multiNodeCoordinator.executeBatchNodeCmd(SQLCmdConstant.COMMIT_CMD);

        }

    }

}

```

```
}
```

第 10 章 Mycat 的分页和跨库 Join

10.1 多数据库支持的分页机制

mycat 对多数据库分页语法的支持主要分为 2 种方式，一是 limit 语法自动转换成原生分页语法，二是直接支持对原生分页语句。目前支持的数据库分页的类型有 oracle、db2、sqlserver、PostgreSQL 等。

主要涉及的类有：

1. DruidMycatRouteStrategy 路由策略入口。
2. MycatStatementParser 扩展语句解析。
3. MycatSelectParser 扩展查询语句解析。
4. MycatExprParser 扩展支持聚合函数。
5. MycatLexer 扩展支持关键词。
6. DruidParserFactory 解析工厂类。
7. DruidSelectOracleParser oracle 分页解析。
8. DruidSelectDb2Parser db2 分页解析。
9. DruidSelectSqlServerParser sqlserver 分页解析。
10. DruidSelectPostgresqlParser PostgreSQL 分页支持。
11. RouteResultset 路由结果类。

10.1.1 DruidMycatRouteStrategy 路由策略入口

```
//这里判断当配置文件中配置了 mysql 以外的数据库类型时，才启用多数据库语法支持。
```

```
//默认只支持 mysql 语法。
```

```
if(schema.isNeedSupportMultiDBType())  
{
```

```

    parser = new MycatStatementParser(stmt);
} else
{
    parser = new MySqlStatementParser(stmt); //只有 mysql 时只支持 mysql 语法
}

MycatSchemaStatVisitor visitor = null;
SQLStatement statement;
//解析出现问题统一抛 SQL 语法错误
try {
    statement = parser.parseStatement();
    visitor = new MycatSchemaStatVisitor();
} catch (Exception t) {
    LOGGER.error("DruidMycatRouteStrategyError", t);
    throw new SQLSyntaxErrorException(t);
}

```

10.1.2 MycatStatementParser 扩展语句解析

```

//负责覆盖 SQLExprParser、SQLSelectParser
public MycatStatementParser(String sql)
{
    super(sql);
    selectExprParser = new MycatExprParser(sql);
}

public MycatStatementParser(Lexer lexer)
{
    super(lexer);
    selectExprParser = new MycatExprParser(lexer);
}

```

```

}

protected SQLExprParser selectExprParser;

@Override
public SQLSelectStatement parseSelect()
{

    MycatSelectParser selectParser = new MycatSelectParser(this.selectExprParser);
    return new SQLSelectStatement(selectParser.select(), JdbcConstants.MYSQL);
}

public SQLSelectParser createSQLSelectParser()
{
    return new MycatSelectParser(this.selectExprParser);
}

//由于 druid 默认提供的 load data 解析有 bug，所以这里进行覆盖替换为自己的解析实现
protected MySQLLoadDataInFileStatement parseLoadDataInFile()
{
    acceptIdentifier("DATA");

    MySQLLoadDataInFileStatement stmt = new MySQLLoadDataInFileStatement();

    if (identifierEquals(LowPriority.LOW_PRIORITY)) {
        stmt.setLowPriority(true);
        lexer.nextToken();
    }

    if (identifierEquals("CONCURRENT")) {

```

```

    stmt.setConcurrent(true);

    lexer.nextToken();
}

if (identifierEquals(LOCAL)) {
    stmt.setLocal(true);
    lexer.nextToken();
}

acceptIdentifier("INFILE");

SQLLiteralExpr fileName = (SQLLiteralExpr) exprParser.expr();
stmt.setFileName(fileName);

if (lexer.token() == Token.REPLACE) {
    stmt.setReplicate(true);
    lexer.nextToken();
}

if (identifierEquals(IGNORE)) {
    stmt.setIgnore(true);
    lexer.nextToken();
}

accept(Token.INTO);
accept(Token.TABLE);

SQLName tableName = exprParser.name();
stmt.setTableName(tableName);

```

```

if (identifierEquals(CHARACTER)) {
    lexer.nextToken();
    accept(Token.SET);

    if (lexer.token() != Token.LITERAL_CHARS) {
        throw new ParseException("syntax error, illegal charset");
    }

    String charset = lexer.stringVal();
    lexer.nextToken();
    stmt.setCharset(charset);
}

if (identifierEquals("FIELDS") || identifierEquals("COLUMNS")) {
    lexer.nextToken();
    if (identifierEquals("TERMINATED")) {
        lexer.nextToken();
        accept(Token.BY);
        stmt.setColumnsTerminatedBy(new SQLCharExpr(lexer.stringVal()));
        lexer.nextToken();
    }

    if (identifierEquals("OPTIONALLY")) {
        stmt.setColumnsEnclosedOptionally(true);
        lexer.nextToken();
    }

    if (identifierEquals("ENCLOSED")) {

```

```

lexer.nextToken();

accept(Token.BY);

stmt.setColumnsEnclosedBy(new SQLCharExpr(lexer.stringVal()));

lexer.nextToken();
}

if (identifierEquals("ESCAPED")) {
    lexer.nextToken();

    accept(Token.BY);

    stmt.setColumnsEscaped(new SQLCharExpr(lexer.stringVal()));

    lexer.nextToken();
}
}

if (identifierEquals("LINES")) {
    lexer.nextToken();

    if (identifierEquals("STARTING")) {
        lexer.nextToken();

        accept(Token.BY);

        stmt.setLinesStartingBy(new SQLCharExpr(lexer.stringVal()));

        lexer.nextToken();
    }

    if (identifierEquals("TERMINATED")) {
        lexer.nextToken();

        accept(Token.BY);

        stmt.setLinesTerminatedBy(new SQLCharExpr(lexer.stringVal()));

        lexer.nextToken();
    }
}

```



```

}

if (identifierEquals(IGNORE)) {
    lexer.nextToken();
    stmt.setIgnoreLinesNumber((SQLLiteralExpr) this.exprParser.expr());
    acceptIdentifier("LINES");
}

if (lexer.token() == Token.LPAREN) {
    lexer.nextToken();
    this.exprParser.exprList(stmt.getColumns(), stmt);
    accept(Token.RPAREN);
}

if (lexer.token() == Token.SET) {
    lexer.nextToken();
    this.exprParser.exprList(stmt.getSetList(), stmt);
}

return stmt;
}

```

10.1.3 MycatSelectParser 扩展查询语句解析

//这里主要负责解析多数据库语法时不会出错，目前扩展支持了 top 关键字

```

protected SQLSelectItem parseSelectItem()
{
    parseTop();
    return super.parseSelectItem();
}

```

```

public void parseTop()
{
    if (lexer.token() == Token.TOP)
    {
        lexer.nextToken();
    }
}

```

10.1.4 MycatExprParser 扩展支持聚合函数

```

//这里负责扩展聚合函数的支持，目前扩展了对 ROW_NUMBER 的支持
public static String[] max_agg_functions = {"AVG", "COUNT", "GROUP_CONCAT", "MAX", "MIN", "STDDEV",
"SUM", "ROW_NUMBER"};

```

10.1.5 MycatLexer 扩展支持关键词

```

//扩展了对关键词的支持，目前主要是 top
map.put("TOP", Token.TOP);

```

10.1.6 DruidParserFactory 解析工厂类

```

//根据配置数据库类型返回对应数据库类型的 select 解析类
if (dbTypes.contains("oracle"))
{
    parser = new DruidSelectOracleParser();
    break;
} else if (dbTypes.contains("db2"))
{
    parser = new DruidSelectDb2Parser();
    break;
} else if (dbTypes.contains("sqlserver"))
{

```

```

        parser = new DruidSelectSqlServerParser();

        break;

    } else if (dbTypes.contains("postgresql"))
    {

        parser = new DruidSelectPostgresqlParser();

        break;

    }

```

10.1.7 DruidSelectOracleParser oracle 分页解析

//解析 oracle 的 2 种分页以及通过 rownum 限制查询最大条数的语法

```

protected void parseNativePageSql(SQLStatement stmt, RouteResultset rrs, OracleSelectQueryBlock
mysqlSelectQuery, SchemaConfig schema)
{
    //第一层子查询

    SQLExpr where= mysqlSelectQuery.getWhere();
    SQLTableSource from= mysqlSelectQuery.getFrom();
    if(where instanceof SQLBinaryOpExpr &&from instanceof SQLSubqueryTableSource)
    {

        SQLBinaryOpExpr one= (SQLBinaryOpExpr) where;
        SQLExpr left=one.getLeft();

        SQLBinaryOperator operator =one.getOperator();

        //解析只有一层 rownum 限制大小

        if(one.getRight() instanceof SQLIntegerExpr &&"rownum".equalsIgnoreCase(left.toString())
&&(operator==SQLBinaryOperator.LessThanOrEqual||operator==SQLBinaryOperator.LessThan))
        {

            SQLIntegerExpr right = (SQLIntegerExpr) one.getRight();

            int firstrownum = right.getNumber().intValue();

```

```

if (operator == SQLBinaryOperator.LessThan&&firstrownum!=0) firstrownum = firstrownum - 1;

SQLSelectQuery subSelect = ((SQLSubqueryTableSource) from).getSelect().getQuery();

if (subSelect instanceof OracleSelectQueryBlock)
{
    rrs.setLimitStart(0);

    rrs.setLimitSize(firstrownum);

    mysqlSelectQuery = (OracleSelectQueryBlock) subSelect;    //为了继续解出 order by 等
    parseOrderAggGroupOracle(stmt,rrs, mysqlSelectQuery, schema);

    isNeedParseOrderAgg=false;
}
}

else //解析 oracle 三层嵌套分页

    if(one.getRight() instanceof SQLIntegerExpr &&!"rownum".equalsIgnoreCase(left.toString())
&&(operator==SQLBinaryOperator.GreaterThan||operator==SQLBinaryOperator.GreaterThanOrEqual))
    {
        parseThreeLevelPageSql(stmt, rrs, schema, (SQLSubqueryTableSource) from, one, operator);
    }

    else //解析 oracle rownumber over 分页
    {

        SQLSelectQuery subSelect = ((SQLSubqueryTableSource) from).getSelect().getQuery();

        SQLOrderBy orderBy=null;

//解析分页语句成功，把分页参数赋值到路由结果类

if (subSelect instanceof OracleSelectQueryBlock)
{
    rrs.setLimitStart(0);

    rrs.setLimitSize(firstrownum);

```

```

mysqlSelectQuery = (OracleSelectQueryBlock) subSelect; //为了继续解出 order by 等
parseOrderAggGroupOracle(stmt,rrs, mysqlSelectQuery, schema);

isNeedParseOrderAgg=false;
}

```

10.1.8 DruidSelectDb2Parser db2 分页解析

```

//由于 druid 的 db2 解析部分不够完整，所以通过继承 oracle 的解析来实现
//db2 的分页方式为 row_number 分页，解析与 oracle 类似
//通过正则表达式解析 db2 的 FETCH FIRST ROWS ONLY 语法

protected void parseNativeSql(SQLStatement stmt,RouteResultset rrs, OracleSelectQueryBlock
mysqlSelectQuery,SchemaConfig schema)
{
    String patten="FETCH(?:\\s)+FIRST(?:\\s)+(?\\d+)(?:\\s)+ROWS(?:\\s)+ONLY";

    Pattern pattern = Pattern.compile(patten,Pattern.CASE_INSENSITIVE);

    Matcher matcher = pattern.matcher(getCtx().getSql());

    while (matcher.find())
    {

        String row=  matcher.group(1);

        rrs.setLimitStart(0);

        rrs.setLimitSize(Integer.parseInt(row));

    }
}

```

10.1.9 DruidSelectSqlServerParser sqlserver 分页解析

```

//通过解析 row_number 和 top 来实现对 sqlserver 的 2 种分页语法的支持

boolean hasRowNumber=false;

```

```

        boolean hasSubTop=false;

        int subTop=0;

        SQLServerSelectQueryBlock subSelectOracle = (SQLServerSelectQueryBlock) subSelect;

        List<SQLSelectItem> sqlSelectItems=  subSelectOracle.getSelectList();

        for (SQLSelectItem sqlSelectItem : sqlSelectItems)

        {

            SQLExpr sqlExpr= sqlSelectItem.getExpr() ;

            if(sqlExpr instanceof SQLAggregateExpr )

            {

                SQLAggregateExpr agg= (SQLAggregateExpr) sqlExpr;

                if("row_number".equalsIgnoreCase(agg.getMethodName())&&agg.getOver()!=null)

                {

                    hasRowNumber=true;

                    orderBy= agg.getOver().getOrderBy();

                }

            }

        }

        if(subSelectOracle.getFrom() instanceof SQLSubqueryTableSource)

        {

            SQLSubqueryTableSource subFrom= (SQLSubqueryTableSource) subSelectOracle.getFrom();

            if (subFrom.getSelect().getQuery() instanceof SQLServerSelectQueryBlock)

            {

                SQLServerSelectQueryBlock sqlSelectQuery = (SQLServerSelectQueryBlock)

subFrom.getSelect().getQuery();

                if(sqlSelectQuery.getTop()!=null)

                {

                    SQLExpr sqlExpr= sqlSelectQuery.getTop().getExpr() ;

```

```

        if(sqlExpr instanceof SQLIntegerExpr)
        {
            hasSubTop=true;

            subTop=((SQLIntegerExpr) sqlExpr).getNumber().intValue();

            orderBy= subFrom.getSelect().getOrderBy();

        }

    }

}

}

```

10.1.10 DruidSelectPostgresqlParser PostgreSQL 分页支持

目前对 PostgreSQL 的分页语法使用 DruidSelectParser 已经可以满足需求。

10.1.11 RouteResultset 路由结果类

```

//这里通过对数据库类型的判断，来自动将 limit 语法转换成对应数据库的原生分页语法
public void changeNodeSqlAfterAddLimit(SchemaConfig schemaConfig, String sourceDbType, String sql, int
offset, int count, boolean isNeedConvert) {
    if (nodes != null)
    {

        Map<String, String> dataNodeDbTypeMap = schemaConfig.getDataNodeDbTypeMap();
        Map<String, String> sqlMapCache = new HashMap<>();
        for (RouteResultsetNode node : nodes)
        {
            String dbType = dataNodeDbTypeMap.get(node.getName());
            if (sourceDbType.equalsIgnoreCase("mysql"))
            {
                node.setStatement(sql); //mysql 之前已经加好 limit
            } else if (sqlMapCache.containsKey(dbType))

```

```

        {
            node.setStatement(sqlMapCache.get(dbType));
        } else if(isNeedConvert)
        {
            String nativeSql = PageSQLUtil.convertLimitToNativePageSql(dbType, sql, offset, count);
            sqlMapCache.put(dbType, nativeSql);
            node.setStatement(nativeSql);
        } else {
            node.setStatement(sql);
        }

        node.setLimitStart(offset);
        node.setLimitSize(count);
    }

}
}
}

```

//PageSQLUtil 类负责 limit 语法转原生分页，主要方法来自 druid，但是做了扩展和修改

//通过添加 select 0 解除 sqlserver 的 row_number 必须要有排序的限制

//修复了转换为 db2 分页时的生成 order by 的顺序不对的 bug

```
public class PageSQLUtil
```

```

{
    public static String convertLimitToNativePageSql(String dbType, String sql, int offset, int count)
    {
        if (JdbcConstants.ORACLE.equalsIgnoreCase(dbType))
        {
            OracleStatementParser oracleParser = new OracleStatementParser(sql);

```



```

SQLSelectStatement oracleStmt = (SQLSelectStatement) oracleParser.parseStatement();

return PagerUtils.limit(oracleStmt.getSelect(), JdbcConstants.ORACLE, offset, count);
} else if (JdbcConstants.SQL_SERVER.equalsIgnoreCase(dbType))
{
    SQLServerStatementParser oracleParser = new SQLServerStatementParser(sql);
    SQLSelectStatement sqlserverStmt = (SQLSelectStatement) oracleParser.parseStatement();
    SQLSelect select = sqlserverStmt.getSelect();
    SQLOrderBy orderBy= select.getOrderBy() ;
    if(orderBy==null)
    {
        SQLSelectQuery sqlSelectQuery= select.getQuery();
        if(sqlSelectQuery instanceof SQLServerSelectQueryBlock)
        {
            SQLServerSelectQueryBlock sqlServerSelectQueryBlock= (SQLServerSelectQueryBlock)
sqlSelectQuery;

            SQLTableSource from= sqlServerSelectQueryBlock.getFrom();
            if("limit".equalsIgnoreCase(from.getAlias()))
            {
                from.setAlias(null);
            }
        }
        SQLOrderBy newOrderBy=new SQLOrderBy(new SQLIdentifierExpr("(select 0)"));
        select.setOrderBy(newOrderBy);
    }

    return PagerUtils.limit(select, JdbcConstants.SQL_SERVER, offset, count) ;
}

```

```

else if (JdbcConstants.DB2.equalsIgnoreCase(dbType))
{
    DB2StatementParser db2Parser = new DB2StatementParser(sql);

    SQLSelectStatement db2Stmt = (SQLSelectStatement) db2Parser.parseStatement();

    return limitDB2(db2Stmt.getSelect(), JdbcConstants.DB2, offset, count);

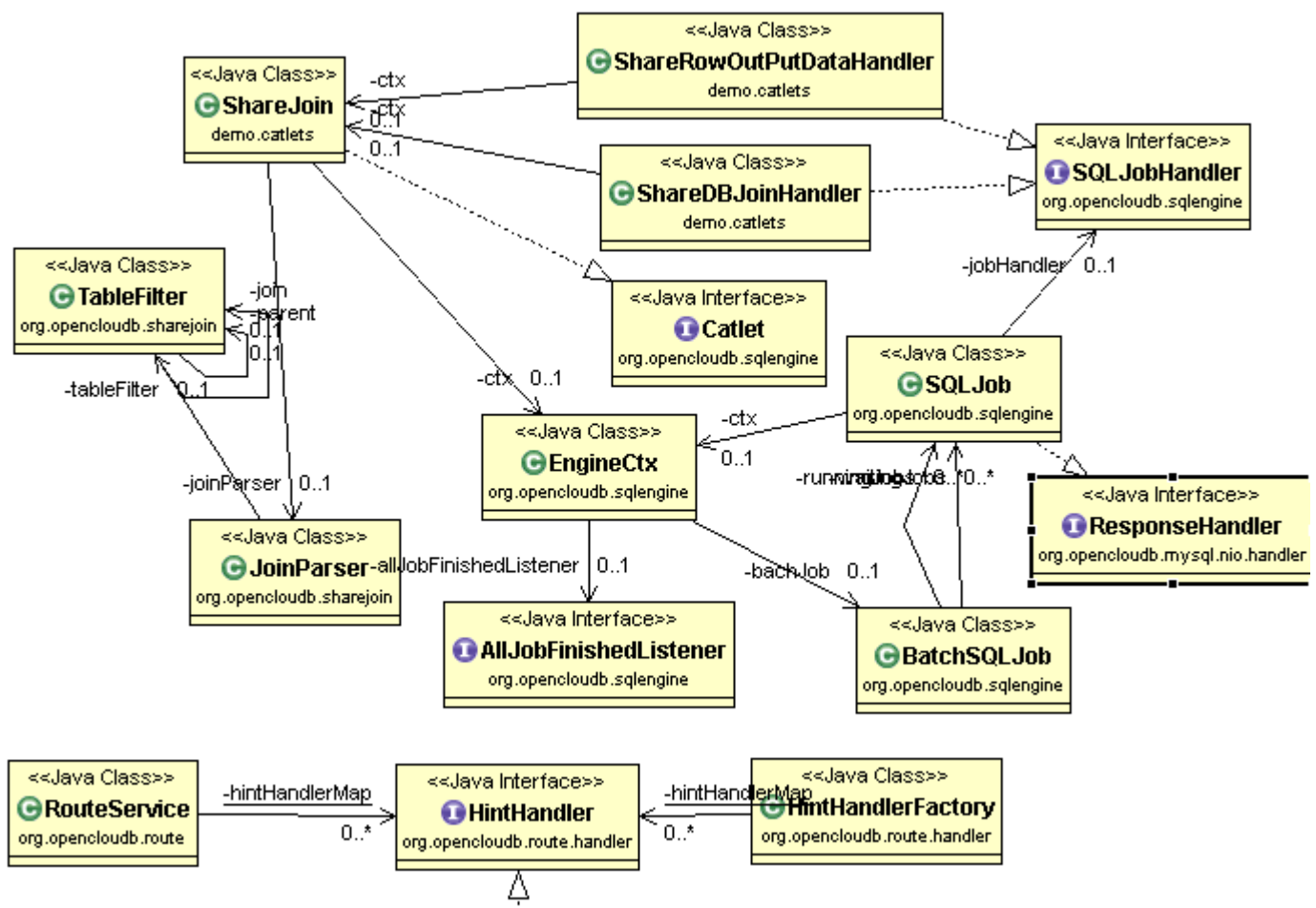
} else if (JdbcConstants.POSTGRESQL.equalsIgnoreCase(dbType))

```

10.2 ShareJoin 代码分析

10.2.1 ShareJoin

ShareJoin 是 Catlet 的一个实现，把解析出的 SQL 分次执行，并存结果集，合并结果集。



以下说的主表和子表，分别是拆分出的第一条 SQL 和第二条 SQL 语句中的表。

```

public class ShareJoin implements Catlet {

    private EngineCtx ctx; //HBT 的执行引擎

```

```

private RouteResultset rrs ;//路由结果集

private JoinParser joinParser;//Join 解析器


private Map<String, byte[]> rows = new ConcurrentHashMap<String, byte[]>();//存记录的结果集
private Map<String,String> ids = new ConcurrentHashMap<String,String>();//join 字段的值


private List<byte[]> fields; //主表的字段
private ArrayList<byte[]> allfields;//所有的字段
private boolean isMfield=false; //已经获取主表的字段了
private int mjob=0;  //job 的任务数
private int maxjob=0; //最大的任务数
private int joinindex=0;//关联 join 表字段的位置
private int sendField=0; //输出 field 的标志
private boolean childRoute=false;//是否重新路由标志
//重新路由使用

private SystemConfig sysConfig;
private SchemaConfig schema;
private int sqltype;
private String charset;
private ServerConnection sc;
private LayerCachePool cachePool;

```

第一步，获取路由的配置信息和原始 SQL 语句，Join 解析器(joinParser)解析原始语句。

```

public void route(SystemConfig sysConfig, SchemaConfig schema,int sqlType, String realSQL, String charset,
ServerConnection sc, LayerCachePool cachePool) {

    int rs = ServerParse.parse(realSQL);

    this.sqltype = rs & 0xff;

    this.sysConfig=sysConfig;

    this.schema=schema;

    this.charset=charset;

```

```

this.sc=sc;

this.cachePool=cachePool;

try {

    MySQLStatementParser parser = new MySQLStatementParser(realSQL);

    SQLStatement statement = parser.parseStatement();

    if(statement instanceof SQLSelectStatement) {

        SQLSelectStatement st=(SQLSelectStatement)statement;

        SQLSelectQuery sqlSelectQuery =st.getSelect().getQuery();

        if(sqlSelectQuery instanceof MySQLSelectQueryBlock) {

            MySQLSelectQueryBlock mysqlSelectQuery = (MySQLSelectQueryBlock)st.getSelect().getQuery();

            joinParser=new JoinParser(mysqlSelectQuery,realSQL);

            joinParser.parser();

        }

    }

} catch (Exception e) {

}

}

```

第二步执行 SQL 语句 。

```

public void processSQL(String sql, EngineCtx ctx) {

    String ssl=joinParser.getSql();//拆分的第一条 SQL 语句

    getRoute(ssl);//对第一条 SQL 语句重新路由

    RouteResultsetNode[] nodes = rrs.getNodes();//获取路由节点

    if (nodes == null || nodes.length == 0 || nodes[0].getName() == null

        || nodes[0].getName().equals("")) {

        ctx.getSession().getSource().writeErrMessage(ErrorCode.ER_NO_DB_ERROR,

            "No dataNode found ,please check tables defined in schema:"

            + ctx.getSession().getSource().getSchema());
    }
}

```

```

return;
}

this.ctx=ctx;

String[] dataNodes =getDataNodes();

maxjob=dataNodes.length;//节点数就是最大的任务数

ShareDBJoinHandler joinHandler = new ShareDBJoinHandler(this,joinParser.getJoinLkey());

    //多个节点执行第一条 SQL 语句

ctx.executeNativeSQLSequceJob(dataNodes, ssl, joinHandler);

    EngineCtx.LOGGER.info("Catlet exec:"+getDataNode(getDataNodes())+" sql:" +ssl);

    //所有任务完成的侦听器

ctx.setAllJobFinishedListener(new AllJobFinishedListener() {

    @Override

    public void onAllJobFinished(EngineCtx ctx) {

        ctx.writeEof();

        EngineCtx.LOGGER.info("发送数据 OK");

    }

});
}

```

//join 第一条 SQL 语句的字段列表，每个节点的表结构一样，只需要获取一次

```

public void putDBFields(List<byte[]> mFields){

    if (!isMfield){

        fields=mFields;

    }

}

```

//join 第一条 SQL 语句的记录结果集

```

public void putDBRow(String id,String nid, byte[] rowData,int finde){

    rows.put(id, rowData);

```

```

    ids.put(id, nid);

    joinindex=findindex;

    //ids.offer(nid);

    int batchSize = 999;

    // 满 1000 条, 发送一个查询请求

    if (ids.size() > batchSize) {

        createQryJob(batchSize);

    }

}

//join 第一条 SQL 语句的节点 job 完成

public void endJobInput(String dataNode, boolean failed){

    mjob++;

    if (mjob>=maxjob){

        createQryJob(Integer.MAX_VALUE);

        ctx.endJobInput();

    }

    // EngineCtx.LOGGER.info("完成"+mjob+":" + dataNode+" failed:"+failed);

}

//创建第二次查询的任务

private void createQryJob(int batchSize) {

    int count = 0;

    Map<String, byte[]> batchRows = new ConcurrentHashMap<String, byte[]>();

    String theId = null;

    StringBuilder sb = new StringBuilder().append('(');

    String svalue="";

    for(Map.Entry<String,String> e: ids.entrySet() ){

        theId=e.getKey();

        batchRows.put(theId, rows.remove(theId));

```

```

        if (!svalue.equals(e.getValue())){
            sb.append(e.getValue()).append(',');
        }
        svalue=e.getValue();
        if (count++ > batchSize) {
            break;
        }
    }
    /*
    while ((theld = ids.poll()) != null) {
        batchRows.put(theld, rows.remove(theld));
        sb.append(theld).append(',');
        if (count++ > batchSize) {
            break;
        }
    }
    */
    if (count == 0) {
        return;
    }
    sb.deleteCharAt(sb.length() - 1).append('');
    String sql = String.format(joinParser.getChildSQL(), sb);//获取第二条 SQL 语句
        //重新计算路由
        getRoute(sql);
        //多个节点执行第二条 SQL 语句,batchRows 主表的数据记录
    ctx.executeNativeSQLParallJob(getDataNodes(),sql, new ShareRowOutPutDataHandler(this,fields,joinindex,
batchRows));

    EngineCtx.LOGGER.info("SQLParallJob:"+getDataNode(getDataNodes())+" sql:" + sql);
}

```

```

        //sendField=1,向客户端输出字段列表

public void writeHeader(String dataNode,List<byte[]> afields, List<byte[]> bfields) {

    sendField++;

    if (sendField==1){

        ctx.writeHeader(afields, bfields);

        setAllFields(afields, bfields);

        // EngineCtx.LOGGER.info("发送字段 2:" + dataNode);

    }

}

        //所有字段放入 allfields

private void setAllFields(List<byte[]> afields, List<byte[]> bfields){

    allfields=new ArrayList<byte[]>();

    for (byte[] field : afields) {

        allfields.add(field);

    }

    //EngineCtx.LOGGER.info("所有字段 2:" +allfields.size());

    for (int i=1;i<bfields.size();i++){

        allfields.add(bfields.get(i));

    }

}

        //得到 allfields

public List<byte[]> getAllFields(){

    return allfields;

}

        //向客户端输出一条记录

public void writeRow(RowDataPacket rowDataPkg){

```



```
ctx.writeRow(rowDataPkg);  
}
```

10.2.2 ShareDBJoinHandler

第一条 SQL 语句执行 job 的事件处理。

```
class ShareDBJoinHandler implements SQLJobHandler {  
  
    private List<byte[]> fields;//表的字段列表  
  
    private final ShareJoin ctx;//ShareJoin 执行结果处理  
  
    private String joinkey;//join 的字段  
  
  
    public ShareDBJoinHandler(ShareJoin ctx,String joinField) {  
  
        super();  
  
        this.ctx = ctx;  
  
        this.joinkey=joinField;  
  
    }  
  
    //获取字段列表的事件  
  
    @Override  
  
    public void onHeader(String dataNode, byte[] header, List<byte[]> fields) {  
  
        this.fields = fields;  
  
        ctx.putDBFields(fields);//交给 ShareJoin 处理字段  
  
    }  
  
  
  
  
  
  
  
  
  
    public static int getFieldIndex(List<byte[]> fields,String fkey){  
  
        int i=0;  
  
        for (byte[] field :fields) {  
  
            FieldPacket fieldPacket = new FieldPacket();  
  
            fieldPacket.read(field);  
  
            if (ByteUtil.getString(fieldPacket.name).equals(fkey)){  
  
                return i;  
  
            }  
  
        }  
  
    }  
  
}
```

```

    }

    i++;

}

return i;
}

@Override

public boolean onRowData(String dataNode, byte[] rowData) {

    int fid=getFieldIndex(fields,joinkey);//join 字段在表字段列表的位置

    String id = ResultSetUtil.getColumnValAsString(rowData, fields, 0);//主键的值，默认 id

    String nid = ResultSetUtil.getColumnValAsString(rowData, fields, fid);//join 字段的值

    // 交给 ShareJoin 处理结果集，rowData 记录字节数组

    ctx.putDBRow(id,nid, rowData,fid);

    return false;

}

    //处理完成标志

@Override

public void finished(String dataNode, boolean failed) {

    ctx.endJobInput(dataNode,failed);//通知 ShareJoin

}

}

```

10.2.3 ShareRowOutPutDataHandler

执行第二条 SQL 语句。

```

class ShareRowOutPutDataHandler implements SQLJobHandler {

    private final List<byte[]> afields;//主表的字段

    private List<byte[]> bfields;//子表的字段

    private final ShareJoin ctx;//ShareJoin 执行结果处理

```

```

private final Map<String, byte[]> arows;//主表的记录

private int joini; //join 字段的位置

public ShareRowOutPutDataHandler(ShareJoin ctx,List<byte[]> afields,int joini,Map<String, byte[]> arows) {
    super();

    this.afields = afields;

    this.ctx = ctx;

    this.arows = arows;

    this.joini =joini;

    //EngineCtx.LOGGER.info("二次查询:" +arows.size()+ " afields:
"+FenDBJoinHandler.getFieldNames(afields));
}

    //获取字段的处理

@Override

public void onHeader(String dataNode, byte[] header, List<byte[]> bfields) {

    this.bfields=bfields;

    ctx.writeHeader(dataNode,afields, bfields);//交给 ShareJoin 处理字段
}


//不是主键，获取 join 左边的的记录

private byte[] getRow(String value,int index){

    for(Map.Entry<String,byte[]> e: arows.entrySet() ){

        String key=e.getKey();

        RowDataPacket rowDataPkg = ResultSetUtil.parseRowData(e.getValue(), afields);

        String id = ByteUtil.getString(rowDataPkg.fieldValues.get(index));

        if (id.equals(value)){

            return arows.remove(key);

        }

    }

    return null;
}

```

```

}

//获取数据记录的处理

@Override

public boolean onRowData(String dataNode, byte[] rowData) {

    RowDataPacket rowDataPkgold = ResultSetUtil.parseRowData(rowData, bfields);

    // 获取 Id 字段,

    String id = ByteUtil.getString(rowDataPkgold.fieldValues.get(0));

    // 查找 ID 对应的 A 表的记录

    byte[] arow = getRow(id,joini);//arows.remove(id);

    while (arow!=null) {

        RowDataPacket rowDataPkg = ResultSetUtil.parseRowData(arow,afields );

        for (int i=1;i<rowDataPkgold.fieldCount;i++){

            // 设置 b.name 字段

            byte[] bname = rowDataPkgold.fieldValues.get(i);

            rowDataPkg.add(bname);

            rowDataPkg.addFieldCount(1);//新增字段

        }

        ctx.writeRow(rowDataPkg);//交给 ShareJoin 处理数据记录

        arow = getRow(id,joini);// 查找 ID 对应的 A 表的记录

    }

    return false;

}

//SQL job 处理完成的事件

@Override

public void finished(String dataNode, boolean failed) {

    // EngineCtx.LOGGER.info("完成 2:" + dataNode+" failed:"+failed);

}

```

```
}
```

第 11 章 Mycat 缓存

11.1 缓存介绍及代码分析

目前的系统大部分都会使用缓存，本地缓存 `oscache`, `ehcache`, 分布式缓存 `memcached`, `redis` 等, 使用缓存会使系统的性能得到提升，详细的介绍请看各自的官网。

Mycat 缓存的数据分别是：`SQLRouteCache`(SQL 语句路由缓存), `TableID2DataNodeCache`(表主键节点缓存), `ER_SQL2PARENTID`(ER 关系缓存)。

Mycat 缓存支持 `ehcache`, `mapdb`, `leveldb`, 通过配置文件 `cacheservice.properties`，决定使用哪种缓存。缓存的代码在 `io.mycat.cache` 和 `io.mycat.cache.impl` 包中。

接口 `CachePool`。

```
public interface CachePool {  
  
    //放入缓存前先用 get 方法判断是否存在  
    public void putIfAbsent(Object key, Object value);  
  
    //判断缓存的 key 是否存在  
    public Object get(Object key);  
  
    //清理缓存  
    public void clearCache();  
  
    //缓存状态信息  
    public CacheStatic getCacheStatic();  
  
    //最大缓存大小  
    public long getMaxSize();  
}
```

缓存池工厂类。

```

public abstract class CachePoolFactory {

    /**
     * create a cache pool instance
     * @param poolName 名称
     * @param cacheSize 大小
     * @param expireSeconds -1 for not expired 失效时间秒
     * @return
     */
    public abstract CachePool createCachePool(String poolName,int cacheSize,int expireSeconds);
}

```

CacheService

缓存服务类管理缓存池。

```

public class CacheService {

    private static final Logger logger = Logger.getLogger(CacheService.class);

    //管理缓存池工厂类
    private final Map<String, CachePoolFactory> poolFactorys = new HashMap<String, CachePoolFactory>();

    //管理缓存池
    private final Map<String, CachePool> allPools = new HashMap<String, CachePool>();

    public CacheService() {

        // load cache pool defined
        try {
            init();
        } catch (Exception e) {
            if (e instanceof RuntimeException) {
                throw (RuntimeException) e;
            } else {

```

```

        throw new RuntimeException(e);
    }
}

}

public Map<String, CachePool> getAllCachePools()
{
    return this.allPools;
}

    //读取缓存 cacheservice.properties 配置文件，由配置文件决定使用那种缓存
private void init() throws Exception {
    Properties props = new Properties();
    props.load(CacheService.class
        .getResourceAsStream("/cacheservice.properties"));
    final String poolFactoryPref = "factory.";
    final String poolKeyPref = "pool.";
    final String layedPoolKeyPref = "layedpool.";
    String[] keys = props.keySet().toArray(new String[0]);
    Arrays.sort(keys);
}

```

CacheStatic 缓存状态信息类，这个很简单，看看代码就明白了。

```

public class CacheStatic {
    private long maxSize;//缓存大小
    private long memorySize;//内存大小
    private long itemSize;//key 数量
    private long accessTimes;//访问次数
    private long putTimes;//put 次数
    private long hitTimes;//命中次数
    private long lastAccesTime;//最后访问时间
    private long lastPutTime;//最后 put 时间
}

```

LayerCachePool 接口，表主键缓存使用。

DefaultLayedCachePool 分层缓存池，LayerCachePool 的实现。

MysqlDataSetCache 数据结果集缓存。

MysqlDataSetService 数据结果集缓存服务类。

11.2 SQLRouteCache

路由缓存，通过缓存 SQL 语句的路由信息，下次查询，不用再路由了，直接从缓存中获取路由信息，然后发到各个节点执行。

我们简单的看下执行一条 SQL 的变化：

通过命令查询下 mycat 的缓存信息。

```
mysql> show @@cache;
```

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT
SQLRouteCache	10000	0	0	0	0	0	0
TableID2DataNodeCache.TESTDB_ORDERS	50000	0	0	0	0	0	0
ER_SQL2PARENTID	1000	0	0	0	0	0	0

```
3 rows in set (0.05 sec)
```

执行 SQL 。

```
mysql> select * from customer;
```

id	name	company_id	sharding_id
2	xue	2	10010
1	wang	1	10000
3	feng	3	10000

3 rows in set (0.38 sec)

查看下缓存信息。

```
mysql> show @@cache;
```

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT
SQLRouteCache	10000	1	1	0	1	1429541712934	1429541713222
TableID2DataNodeCache.TESTDB_ORDERS	50000	0	0	0	0	0	0
ER_SQL2PARENTID	1000	0	0	0	0	0	0

3 rows in set (0.00 sec)

看到变化了吧，SQLRouteCache put 和 ACCESS 访问次数都加一了，再次执行 `select * from customer`。

```
mysql> show @@cache;
```

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT
SQLRouteCache	10000	1	2	1	1	1429541906269	1429541713222
TableID2DataNodeCache.TESTDB_ORDERS	50000	0	0	0	0	0	0
ER_SQL2PARENTID	1000	0	0	0	0	0	0

3 rows in set (0.00 sec)

HIT 命中次数也有了。

代码简单分析，在 `io.mycat.route.RouteService` 类中实现获取路由信息之前都会先在缓存中查询下是否存在，如果存在则直接取出。

缓存的 key 是 schema+SQL 语句。

```
public RouteResultset route(SystemConfig sysconf, SchemaConfig schema,
                             int sqlType, String stmt, String charset, ServerConnection sc)
```

```

        throws SQLNonTransientException {

        RouteResultset rrs = null;

        String cacheKey = null;

        //判断是否是查询语句

        if (sqlType == ServerParse.SELECT) {

            cacheKey = schema.getName() + stmt;//缓存的 key

            rrs = (RouteResultset) sqlRouteCache.get(cacheKey);

            if (rrs != null) {//判断是否存在缓存

                return rrs;

            }

        }

        ...

        //最后几行 put 到缓存中

        if (rrs!=null && sqlType == ServerParse.SELECT && rrs.isCacheAble()) {

            sqlRouteCache.putIfAbsent(cacheKey, rrs);

        }

        return rrs;

```

11.3 TableID2DataNodeCache

表主键 ID 的路由缓存，为每一个表建一个缓存池，命名为 TableID2DataNodeCache.TESTDB_表名,缓存的 key 是 id 的值，value 是节点名。

还是用个简单的例子说明下：

先查看缓存信息。

```
mysql> show @@cache;
```

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT
SQLRouteCache	10000	0	0	0	0	0	0

```
| TableID2DataNodeCache.TESTDB_ORDERS | 50000 | 0 | 0 | 0 | 0 | 0 | 0 |
| ER_SQL2PARENTID | 1000 | 0 | 0 | 0 | 0 | 0 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.02 sec)
```

执行 SQL 语句。

```
mysql> select * from customer where id=1;
+----+-----+-----+-----+
| id | name | company_id | sharding_id |
+----+-----+-----+-----+
| 1 | wang | 1 | 10000 |
+----+-----+-----+-----+
1 row in set (0.13 sec)
```

再次查询缓存信息。

```
mysql> show @@cache;
+-----+-----+-----+-----+-----+-----+-----+-----+
| CACHE | MAX | CUR | ACCESS | HIT | PUT | LAST_ACCESS | LAST_PUT |
+-----+-----+-----+-----+-----+-----+-----+-----+
| SQLRouteCache | 10000 | 0 | 1 | 0 | 0 | 1429544238522 | 0 |
| TableID2DataNodeCache.TESTDB_CUSTOMER | 10000 | 1 | 1 | 0 | 1 | 1429544238624 |
1429544238624 |
| TableID2DataNodeCache.TESTDB_ORDERS | 50000 | 0 | 0 | 0 | 0 | 0 | 0 |
| ER_SQL2PARENTID | 1000 | 0 | 0 | 0 | 0 | 0 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

再次执行同样的 SQL 语句。

```
mysql> show @@cache;
```

```

+-----+-----+-----+-----+-----+-----+-----+
| CACHE          | MAX | CUR | ACCESS | HIT | PUT | LAST_ACCESS | LAST_PUT |
+-----+-----+-----+-----+-----+-----+-----+
| SQLRouteCache  | 10000 | 0 | 2 | 0 | 0 | 1429544832439 | 0 |
| TableID2DataNodeCache.TESTDB_CUSTOMER | 10000 | 1 | 2 | 1 | 1 | 1429544832441 |
1429544238624 |
| TableID2DataNodeCache.TESTDB_ORDERS | 50000 | 0 | 0 | 0 | 0 | 0 | 0 |
| ER_SQL2PARENTID | 1000 | 0 | 0 | 0 | 0 | 0 | 0 |
+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

命中次数和访问次数都由变化了。

执行其他的 SQL 试试。

```
mysql> select * from customer where id=2;
```

```

+----+-----+-----+-----+
| id | name | company_id | sharding_id |
+----+-----+-----+-----+
| 2 | xue | 2 | 10010 |
+----+-----+-----+-----+
1 row in set (0.01 sec)

```

```
mysql> show @@cache;
```

```

+-----+-----+-----+-----+-----+-----+-----+
| CACHE          | MAX | CUR | ACCESS | HIT | PUT | LAST_ACCESS | LAST_PUT |
+-----+-----+-----+-----+-----+-----+-----+
| SQLRouteCache  | 10000 | 0 | 3 | 0 | 0 | 1429544916936 | 0 |
| TableID2DataNodeCache.TESTDB_CUSTOMER | 10000 | 2 | 3 | 1 | 2 | 1429544916937 |
1429544916940 |
| TableID2DataNodeCache.TESTDB_ORDERS | 50000 | 0 | 0 | 0 | 0 | 0 | 0 |

```

ER_SQL2PARENTID	1000	0	0	0	0	0	0

4 rows in set (0.00 sec)

PUT 和 CUR,ACCESS 发生变化了, 证明 id=2 的主键路由信息被缓存了。

代码分析

io.mycat.route.util.RouterUtil

判断是否缓存了主键的路由节点信息。

```

//缓存池 TESTDB_表名
String tableKey = schema.getName() + '_' + tableName;

boolean allFound = true;
for (ColumnRoutePair pair : primaryKeyPairs) { //可
能 id in(1,2,3)多主键

String cacheKey = pair.colValue; //缓存的
key 是 id 的值(主键的值)

String dataNode = (String)
cachePool.get(tableKey, cacheKey);

if (dataNode == null) { //value 是节点名
allFound = false;
continue;
} else {

if (tablesRouteMap.get(tableName) == null) {

tablesRouteMap.put(tableName, new HashSet<String>());

}

tablesRouteMap.get(tableName).add(dataNode);

continue;
}
}

```

```

    }
}

```

MultiNodeQueryHandler.java

MultiNodeQueryWithLimitHandler.java

两个类都是 put 缓存池，一个带 limit 的实现。

```

@Override
public void rowResponse(final byte[] row, final BackendConnection conn) {
    if (errorRepsonsed.get()) {
        conn.close(error);
        return;
    }
    lock.lock();
    try {
        if (dataMergeSvr != null) {
            final String dnName = ((RouteResultsetNode) conn
                .getAttachment()).getName();
            dataMergeSvr.onNewRecord(dnName, row);
        } else {
            if (primaryKeyIndex != -1) { // cache
                //
                primaryKey->
                //
                dataNode
                RowDataPacket rowDataPkg = new
                RowDataPacket(fieldCount);
                rowDataPkg.read(row);
                //主键的值
            }
        }
    } finally {
        lock.unlock();
    }
}

```

```

        String primaryKey = new String(

rowDataPkg.fieldValues.get(primaryKeyIndex));

        LayerCachePool pool = MycatServer.getInstance()

.getRouterservice().getTableId2DataNodeCache();

        //路由节点

        String dataNode = ((RouteResultsetNode) conn

        .getAttachment()).getName();

        //priamaryKeyTable 是 TESTDB_表名

        pool.putIfAbsent(priamaryKeyTable, primaryKey, dataNode);

    }

    row[3] = ++packetId;

    session.getSource().write(row);

}

} catch (Exception e) {

    handleDataProcessException(e);

} finally {

    lock.unlock();

}

}

```

11.4 ER_SQL2PARENTID

ER 关系的缓存目前只是在 Insert 语句中才会使用缓存，子表插入数据的时候，根据 joinKey 的值，判断父表所在分片，从而定位子表分片，分片信息 put 缓存，以便下次直接获取。

缓存 key 的内容是 schema + “:” + sql，例子中的 key 是 TESTDB:select customer.id from customer where customer.id=2, value 是 dn2。

例子 先查询下缓存信息。

```
mysql> show @@cache;
```

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT
SQLRouteCache	10000	0	0	0	0	0	0
TableID2DataNodeCache.TESTDB_ORDERS	50000	0	0	0	0	0	0
ER_SQL2PARENTID	1000	0	0	0	0	0	0

3 rows in set (0.02 sec)

然后执行 insert 语句。

```
mysql> insert orders (id,customer_id,note) values(2,2,'cs');
```

Query OK, 1 row affected (0.51 sec)

再次查询缓存信息。

```
mysql> show @@cache;
```

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT
SQLRouteCache	10000	0	0	0	0	0	0
TableID2DataNodeCache.TESTDB_ORDERS	50000	0	0	0	0	0	0
ER_SQL2PARENTID	1000	1	1	0	1	1429629504951	1429629505354

3 rows in set (0.00 sec)

ER 关系缓存 PUT 成功。

再次执行 insert orders (id,customer_id,note) values(3,2, 'aa')。

CACHE	MAX	CUR	ACCESS	HIT	PUT	LAST_ACCESS	LAST_PUT			
SQLRouteCache	10000	0	0	0	0	0	0			
TableID2DataNodeCache.TESTDB_ORDERS	50000	0	0	0	0	0	0	0		
ER_SQL2PARENTID	1000	1	2	1	1	1429630708823	1429630694284			

3 rows in set (0.03 sec)

ER 关系缓存的实现的代码在类 `DruidInsertParser` 和 `FetchStoreNodeOfChildTableHandler` 中实现。

DruidInsertParser

```
private RouteResultset parserChildTable(SchemaConfig schema, RouteResultset rrs,
    String tableName, MySQLInsertStatement insertStmt) throws SQLNonTransientException {
    TableConfig tc = schema.getTables().get(tableName); //子表配置信息

    String joinKey = tc.getJoinKey(); //获取子表的 Join 字段

    int joinKeyIndex = getJoinKeyIndex(insertStmt.getColumns(), joinKey); //获取子表的 Join 字段在插入语句中的
位置

    if(joinKeyIndex == -1) {
        String inf = "joinKey not provided :" + tc.getJoinKey() + "," + insertStmt;
        LOGGER.warn(inf);
        throw new SQLNonTransientException(inf);
    }

    if(isMultiInsert(insertStmt)) { //批量插入

        String msg = "ChildTable multi insert not provided" ;
        LOGGER.warn(msg);
        throw new SQLNonTransientException(msg);
    }
}
```

```

//获取 join 字段的值

String joinKeyVal = insertStmt.getValues().getValues().get(joinKeyIndex).toString();


String sql = insertStmt.toString();


// try to route by ER parent partion key
RouteResultset theRrs = RouterUtil.routeByERParentKey(sql, rrs, tc,joinKeyVal);
if (theRrs != null) {
    rrs.setFinishedRoute(true);
    return theRrs;
}


// 父表的 sql 语句(route by sql query root parent's datanode)
String findRootTBSql = tc.getLocateRTableKeySql().toLowerCase() + joinKeyVal;
if (LOGGER.isDebugEnabled()) {
    LOGGER.debug("find root parent's node sql " + findRootTBSql);
}

FetchStoreNodeOfChildTableHandler fetchHandler = new FetchStoreNodeOfChildTableHandler();

    //获取分片节点

String dn = fetchHandler.execute(schema.getName(),findRootTBSql, tc.getRootParent().getDataNodes());
if (dn == null) {
    throw new SQLNonTransientException("can't find (root) parent sharding node for sql:" + sql);
}

if (LOGGER.isDebugEnabled()) {
    LOGGER.debug("found partion node for child table to insert "+ dn + " sql : " + sql);
}

return RouterUtil.routeToSingleNode(rrs, dn, sql);
}

```

```

public class FetchStoreNodeOfChildTableHandler implements ResponseHandler {

    private static final Logger LOGGER = Logger
        .getLogger(FetchStoreNodeOfChildTableHandler.class);

    private String sql;

    private volatile String result;

    private volatile String dataNode;

    private AtomicInteger finished = new AtomicInteger(0);

    protected final ReentrantLock lock = new ReentrantLock();

    public String execute(String schema, String sql, ArrayList<String> dataNodes) {

        //缓存 key

        String key = schema + ":" + sql;

        CachePool cache = MycatServer.getInstance().getCacheService()
            .getCachePool("ER_SQL2PARENTID");

        String result = (String) cache.get(key);

        if (result != null) {

            return result;

        }

        ...

        if (dataNode != null) {

            cache.putIfAbsent(key, dataNode); //key 的分片节点信息 put 缓存

        }

        return dataNode;
    }
}

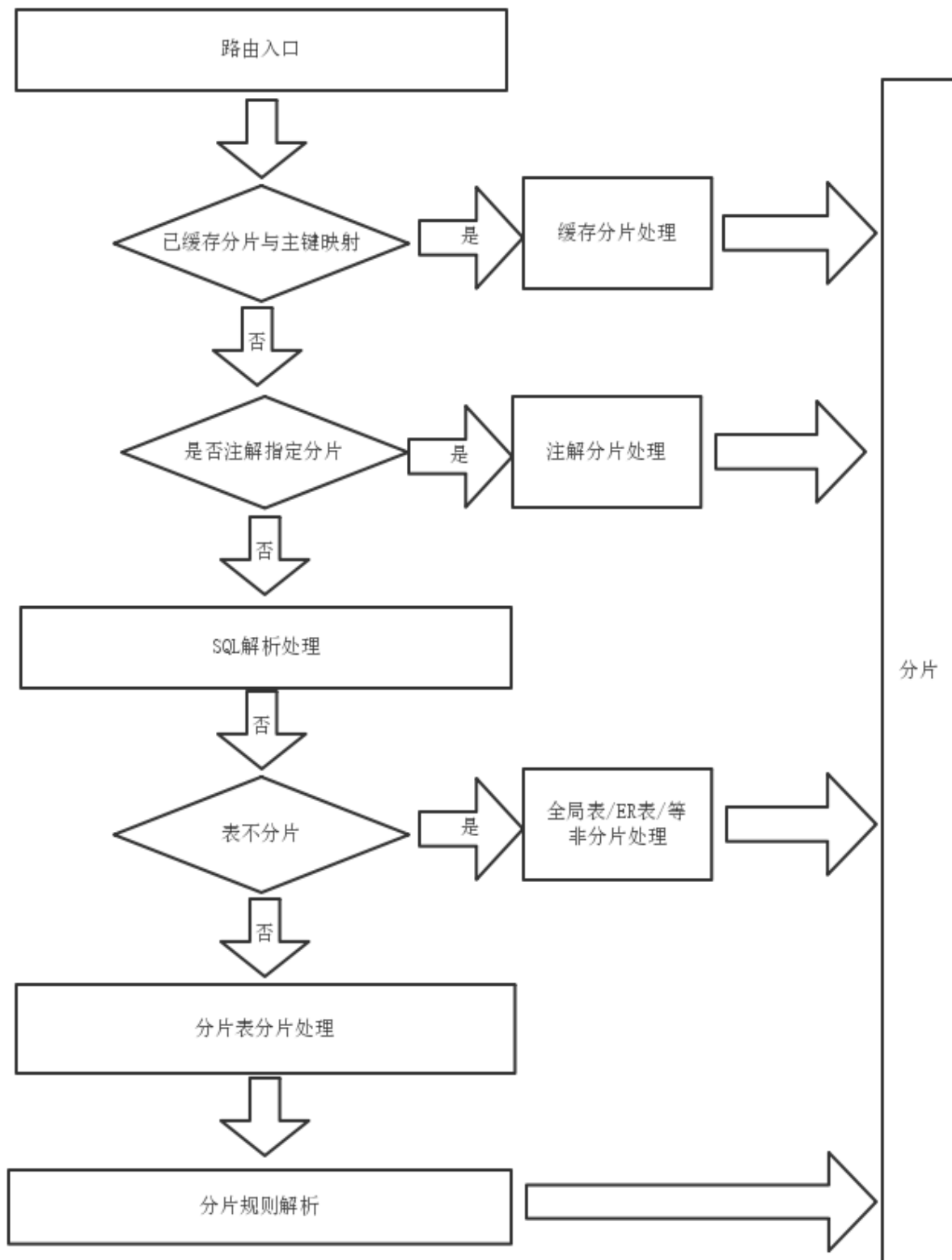
```

第 12 章 Mycat 的分片规则设计

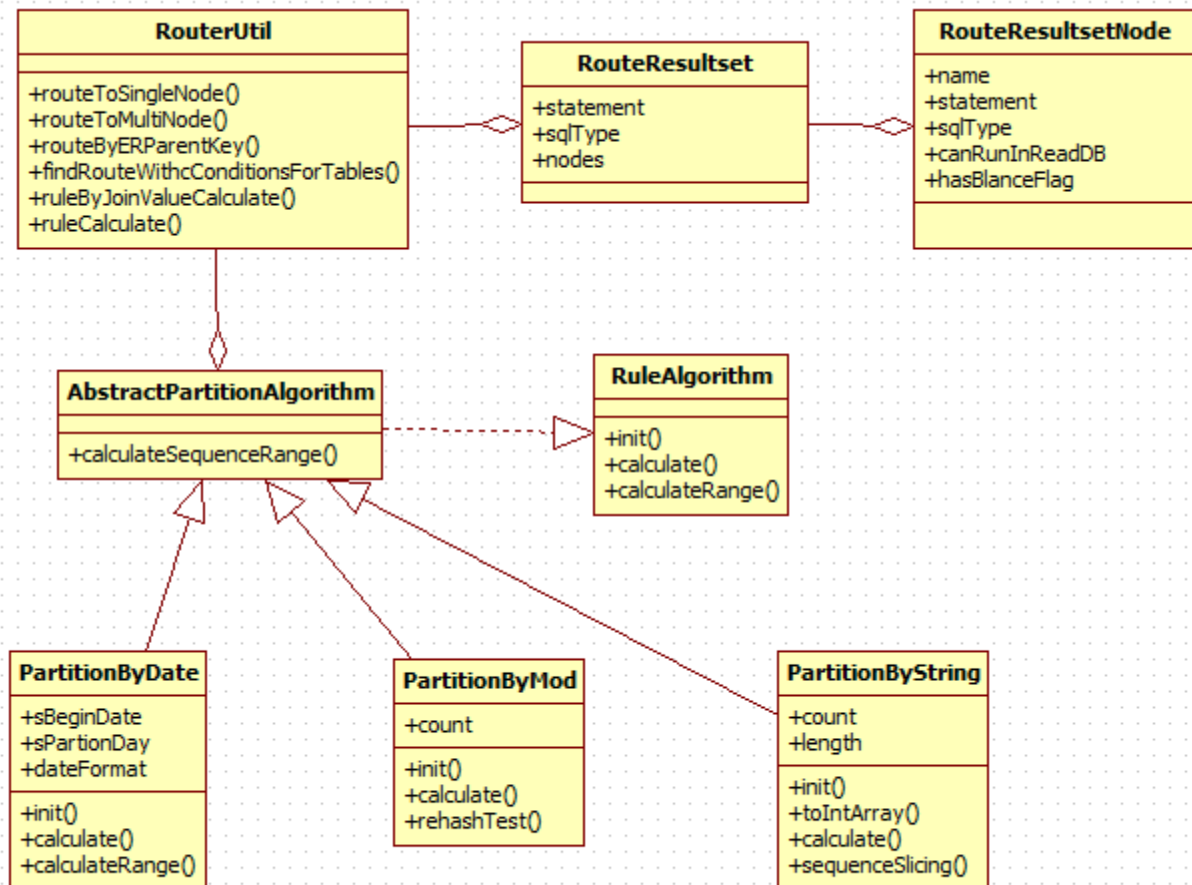
12.1 分片规则设计架构

分布式数据库系统中，分片规则用于定义数据与分片的路由关系，也就是 insert, delete, update, select 的基本 sql 操作中，如何将 sql 路由到对应的分片执行。

Mycat 的总体路由图为：



如图所示分片规则是最终解析 sql 到那个分片执行的规则，Mycat 分片的确定是根据分片字段来确定数据的分布，即根据预先配置好的分片字段（只有一个）到分片规则中解析该字段对应的值应该路由到哪个分片，然后确认 sql 到哪个分片执行，分片规则类图设计为：



RouterUtil,RouteResultset,RouteResultsetNode 几张表是解析 sql，解析出 sql 路由的节点，内部调用 AbstractPartitionAlgorithm 实现类解析分片字段，查找对应的分片。

AbstractPartitionAlgorithm：为路由规则的抽象类。

RuleAlgorithm：路由规则接口抽象，规定了分片规则的初始化（init），路由分片计算（calculate），及路由多值分片计算（calculateRange）。

分片规则中 calculate 方法是基本的分片路由计算方法，根据分片字段值，计算出分片。

分片规则中 calculateRange 方法是范围查询时分片计算，即如果查询类似：

```
select * from t_user t where t.id < 100;
```

需要解析出指定范围的所有值对应分片。

自定义的分片规则只需要继承 AbstractPartitionAlgorithm，按照自己的规则初始化配置文件，并且实现 calculate 或者 calculateRange 方法即可，路由的配置文件为：rule.xml。

route 包下面是对应的路由处理，其下面的 function 包，是分片规则的具体抽象与实现的代码位置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mycat:rule SYSTEM "rule.dtd">
```

```

<mycat:rule xmlns:mycat="http://io.mycat/">
<tableRule name="rule1">
<rule>
<columns>user_id</columns>
<algorithm>func1</algorithm>
</rule>
</tableRule>

<function name="func1" class="io.mycat.route.function.PartitionByLong">
<property name="partitionCount">2</property>
<property name="partitionLength">512</property>
</function>
</mycat:rule>

```

其中 rule 下的 columns 规定了分片字段，algorithm 为自定义分片类配置。

function 标签为分片规则配置：

name：为自定义名字。

class：自定义分片规则方法。

property：其中的参数为自定义参数配置。

12.2 分片规则自定义实现

本章节通过日期分片讲解分片规则内部实现细节：

```

package io.mycat.route.function;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import org.apache.log4j.Logger;
import io.mycat.config.model.rule.RuleAlgorithm;

```

```

/**
 * 例子按日期列分区格式 between 操作解析的范例
 *
 * @author lxy
 *
 */
public class PartitionByDate extends AbstractPartitionAlgorithm implements RuleAlgorithm {

    private static final Logger LOGGER = Logger
        .getLogger(PartitionByDate.class);

    private String sBeginDate;
    private String sPartionDay;
    private String dateFormat;

    private long beginDate;
    private long partionTime;

    private static final long oneDay = 86400000;

    @Override
    public void init() {
        try {
            beginDate = new SimpleDateFormat(dateFormat).parse(sBeginDate)
                .getTime();
        } catch (ParseException e) {
            throw new java.lang.IllegalArgumentException(e);
        }
        partionTime = Integer.parseInt(sPartionDay) * oneDay;
    }

```


@Override

```
public Integer calculate(String columnValue) {  
    try {  
        long targetTime = new SimpleDateFormat(dateFormat).parse(  
            columnValue).getTime();  
        int targetPartition = (int) ((targetTime - beginDate) / partionTime);  
        return targetPartition;  
    } catch (ParseException e) {  
        throw new java.lang.IllegalArgumentException(e);  
    }  
}
```

@Override

```
public Integer[] calculateRange(String beginValue, String endValue) {  
    return AbstractPartitionAlgorithm.calculateSequenceRange(this, beginValue, endValue);  
}
```

```
public void setsBeginDate(String sBeginDate) {  
    this.sBeginDate = sBeginDate;  
}
```

```
public void setsPartionDay(String sPartionDay) {  
    this.sPartionDay = sPartionDay;  
}
```

```
public void setDateFormat(String dateFormat) {
```

```

    this.dateFormat = dateFormat;
}

}

```

在日期分片字段配置中，分片规则类 PartitionByDate 的配置属性与类的成员变量对应一次为：

dateFormat==>private String dateFormat;

sBeginDate==>private String sBeginDate;

sPartitionDay==>private String sPartitionDay;

在 Mycat 的配置文件装载机制中，会根据 property 自动设置类的成员变量，因此只要设置了 Set...方法就可以赋值。

init 方法：

主要处理每种规则的自定义处理，例如本规则中，解析了变量 beginDate、partitionTime 。

```

try {
    beginDate = new SimpleDateFormat(dateFormat).parse(sBeginDate)
        .getTime();
} catch (ParseException e) {
    throw new java.lang.IllegalArgumentException(e);
}

partitionTime = Integer.parseInt(sPartitionDay) * oneDay;

```

calculate 方法：

计算路由分片的核心方法，本规则中通过处理传入的（目标日期-设置的开始日期间隔）/分片时间，计算出偏移量即是分片节点，所有的分片节点编号都是从 0 开始编码。

例如：每个 1 天一分片，开始日期是 2015-01-01 那么分片日期字段值假若是 2015-01-10，那么通过公式：

分片 = (2015-01-10-2015-01-01) / 1 = 9，即 dn9。

```

try {
    long targetTime = new SimpleDateFormat(dateFormat).parse(
        columnValue).getTime();
    int targetPartition = (int) ((targetTime - beginDate) / partitionTime);
}

```

```

return targetPartition;

} catch (ParseException e) {

    throw new java.lang.IllegalArgumentException(e);

}

```

calculateRange 方法:

calculateRange 方法默认根据继承的抽象类规则，可以不实现，默认实现是获取分片字段的值连续范围内的所有分片，主要用于类似：update test where id<5; 这种语句中，通过解析条件 id<15 解析出所有的 id 值域分片的对应关系，依次路由执行，[1->dn0,2->dn1,3->dn2,4->dn3]。

```

Integer begin = 0, end = 0;

begin = algorithm.calculate(beginValue);

end = algorithm.calculate(endValue);


if(begin == null || end == null){

    return new Integer[0];

}


if (end >= begin) {

    int len = end-begin+1;

    Integer [] re = new Integer[len];

    for(int i =0;i<len;i++){

        re[i]=begin+i;

    }

    return re;

}else{

```

```
return null;

}
```

第 13 章 Mycat Load Data 源码

13.1 load data 代码分析

load data infile 语句可以从一个文本文件中以很高的速度读入一个表中。性能大概是 insert 语句的几十倍。

13.1.1 ServerLoadDataInfileHandler

//客户端发送 load data 的 sql 语句会执行到 start 方法里

```
public void start(String sql)

{

    //保存解析变量，留作后续使用

    clear();

    this.sql = sql;

    SQLStatementParser parser = new MycatStatementParser(sql);

    statement = (MySQLLoadDataInFileStatement) parser.parseStatement();

    fileName = parseFileName(sql);


    schema = MycatServer.getInstance().getConfig()

        .getSchemas().get(serverConnection.getSchema());

    tableId2DataNodeCache = (LayerCachePool)

MycatServer.getInstance().getCacheService().getCachePool("TableID2DataNodeCache");

    tableName = statement.getTableName().getSimpleName().toUpperCase();

    tableConfig = schema.getTables().get(tableName);

    tempPath = SystemConfig.getHomePath() + File.separator + "temp" + File.separator +

serverConnection.getId() + File.separator;
```

```

tempFile = tempPath + "clientTemp.txt";

tempByteBuffer = new ByteArrayOutputStream();

parseLoadDataPram();

//判断为 local 参数，则向客户端发送请求文件包，客户端收到此包后会将 load data 的数据文件封包发送过来
if (statement.isLocal())
{
    isStartLoadData = true;

    //向客户端请求发送文件

    ByteBuffer buffer = serverConnection.allocate();

    RequestFilePacket filePacket = new RequestFilePacket();

    filePacket.fileName = fileName.getBytes();

    filePacket.packetId = 1;

    filePacket.write(buffer, serverConnection, true);
} else
{
    if (!new File(fileName).exists())
    {
        serverConnection.writeErrMessage(ErrorCode.ER_FILE_NOT_FOUND, fileName + " is not found!");

        clear();
    } else
    {
        //不是 local 时，直接从 mycat 服务器上的路径读取文件

        parseFileByLine(fileName, loadData.getCharset(), loadData.getLineTerminatedBy());

        RouteResultset rrs = buildResultSet(routeResultMap);

        if (rrs != null)
        {
            flushDataToFile();
        }
    }
}

```

```

        isStartLoadData = false;

        serverConnection.getSession2().execute(rrs, ServerParse.LOAD_DATA_INFILE_SQL);
    }

}

}

}

//收到客户端发送过来的 load data 的数据文件，会有多次
public void handle(byte[] data)
{

    try
    {
        if (sql == null)
        {
            serverConnection.writeErrMessage(ErrorCode.ER_UNKNOWN_COM_ERROR,
                "Unknown command");

            clear();

            return;
        }

        BinaryPacket packet = new BinaryPacket();

        ByteArrayInputStream inputStream = new ByteArrayInputStream(data, 0, data.length);

        packet.read(inputStream);

//为了性能考虑，超过 200M 才会存文件，低于 200M 的直接保存在内存中

        saveByteOrToFile(packet.data, false);

    } catch (IOException e)
    {

```

```

        throw new RuntimeException(e);
    }

}

```

客户端发送 load data 数据文件结束后，会发送一个空包，到这里

```

public void end(byte packID)
{
    isStartLoadData = false;

    this.packID = packID;

    //load in data 空包结束
    saveByteOrToFile(null, true);

    List<SQLExpr> columns = statement.getColumns();

    String tableName = statement.getTableName().getSimpleName();

    if (isHasStoreToFile)
    {
        parseFileByLine(tempFile, loadData.getCharset(), loadData.getLineTerminatedBy());
    } else
    {
        String content = new String(tempByteBuffer.toByteArray(), Charset.forName(loadData.getCharset()));

        //引入 csv 解析器来解析自定义分割符号换行符等的数据
        //如果一个字段的值中包括了分隔符、换行符之类，可以通过加引号等括起来来解决

        CsvParserSettings settings = new CsvParserSettings();

        settings.getFormat().setLineSeparator(loadData.getLineTerminatedBy());

        settings.getFormat().setDelimiter(loadData.getFieldTerminatedBy().charAt(0));

        if(loadData.getEnclose()!=null)
        {

```

```

        settings.getFormat().setQuote(loadData.getEnclose().charAt(0));
    }
    settings.getFormat().setNormalizedNewline(loadData.getLineTerminatedBy().charAt(0));
    CsvParser parser = new CsvParser(settings);
    try
    {
        parser.beginParsing(new StringReader(content));
        String[] row = null;

        while ((row = parser.parseNext()) != null)
        {
            parseOneLine(columns, tableName, row, false, null);
        }
    } finally
    {
        parser.stopParsing();
    }

}

RouteResultset rrs = buildResultSet(routeResultMap);
if (rrs != null)
{
    flushDataToFile();
    serverConnection.getSession2().execute(rrs, ServerParse.LOAD_DATA_INFILE_SQL);
}

```



```

}

//由于变量是连接级别共享的，所以提高 clear 方法来清空变量或临时文件
public void clear()
{
    isStartLoadData = false;
    tableId2DataNodeCache = null;
    schema = null;
    tableConfig = null;
    isHasStoreToFile = false;
}

```

13.1.2 FrontendCommandHandler

```

//通过判断是否已经发送过 load data 的 sql 语句来过滤判断是否是 load data 的数据包
//可以避免将 load data 的数据包误识别成其他的包
public void handle(byte[] data)
{
    if(source.getLoadDataInfileHandler()!=null&&source.getLoadDataInfileHandler().isStartLoadData())
    {
        MySQLMessage mm = new MySQLMessage(data);
        int packetLength = mm.readUB3();
        if(packetLength+4==data.length)
        {
            source.loadDataInfileData(data);
        }
        return;
    }
}

```

13.1.3 LoadDataResponseHandler

当向后端的 db 发送完 load data 的 sql 语句，后端 db 会发送请求文件包，由 LoadDataResponseHandler 负责将数据发送到后端。

```

public interface LoadDataResponseHandler
{
    /**
     * 收到请求发送文件数据包的响应处理
     */
    void requestDataResponse(byte[] row, BackendConnection conn);
}

LoadDataUtil

//发送数据内容到后端

public static void requestFileDataResponse(byte[] data, BackendConnection conn)
{

    byte packId= data[3];

    BackendAIOConnection backendAIOConnection= (BackendAIOConnection) conn;

    RouteResultsetNode rrn= (RouteResultsetNode) conn.getAttachment();

    LoadData loadData= rrn.getLoadData();

    List<String> loadDataData = loadData.getData();

    try
    {
        if(loadDataData !=null&&loadDataData.size()>0)
        {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();

            for (int i = 0, loadDataDataSize = loadDataData.size(); i < loadDataDataSize; i++)
            {

                String line = loadDataData.get(i);

                String s =(i==loadDataDataSize-1)?line: line + loadData.getLineTerminatedBy();

```

```

        byte[] bytes = s.getBytes(loadData.getCharset());

        bos.write(bytes);

    }

    packId= writeToBackConnection(packId,new
ByteArrayInputStream(bos.toByteArray()),backendAIOConnection);

    } else
    {
        //从文件读取

        packId= writeToBackConnection(packId,new BufferedInputStream(new
FileInputStream(loadData.getFileName()))),backendAIOConnection);

    }
}catch (IOException e)
{

    throw new RuntimeException(e);
} finally
{
    //结束必须发空包

    byte[] empty = new byte[] { 0, 0, 0,3 };

    empty[3]=++packId;

    backendAIOConnection.write(empty);
}

```

13.2 mysql 压缩协议代码分析

MySQLConnectionAuthenticator MySQLConnection

FrontendConnection FrontendAuthenticator

```
//判断 2 端都支持 mysql 的压缩协议时，才会启用

    // 处理认证结果

                                source.setHandler(new MySQLConnectionHandler(source));

                                source.setAuthenticated(true);

                                boolean clientCompress =

Capabilities.CLIENT_COMPRESS==(Capabilities.CLIENT_COMPRESS & packet.serverCapabilities);

//mycat 的 server.xml 中配置是否启用压缩协议的参数

                                boolean usingCompress=

MycatServer.getInstance().getConfig().getSystem().getUseCompression()==1 ;

                                if(clientCompress&&usingCompress)

                                {

                                        source.setSupportCompress(true);

                                }

}
```

13.2.1 AbstractConnection

```
//判断是否双方都支持压缩协议后，进行压缩协议的解压缩

public void handle(byte[] data) {

    if(isSupportCompress())

    {

        List<byte[]> packs= CompressUtil.decompressMysqlPacket(data,decompressUnfinishedDataQueue);

        for (byte[] pack : packs)

        {

                                if(pack.length != 0)

                                handler.handle(pack);

        }

    }

}
```

```

    }
} else
{
    handler.handle(data);
}

}

//判断是否双方都支持压缩协议后，进行压缩协议的压缩
public final void write(ByteBuffer buffer) {
    if(isSupportCompress())
    {
        ByteBuffer    newBuffer=
CompressUtil.compressMysqlPacket(buffer,this,compressUnfinishedDataQueue);
        writeQueue.offer(newBuffer);

    } else
    {
        writeQueue.offer(buffer);
    }

    // if ansyn write finishe event got lock before me ,then writing
    // flag is set false but not start a write request
    // so we check again
    try {
        this.socketWR.doNextWriteCheck();
    } catch (Exception e) {
        LOGGER.warn("write err:", e);
        this.close("write err:" + e);
    }
}

```

```

    }

//压缩协议的包头大小为 7 和普通的协议包头大小不一样
    protected final int getPacketLength(ByteBuffer buffer, int offset) {
int  headerSize =getPacketHeaderSize();

    if(isSupportCompress())
    {
        headerSize=7;
    }

```

13.2.2 CompressUtil

//将普通的 mysql 协议包压缩成压缩包，目前采取一包一压的方式，后续可以优化将多个包压缩到一个压缩包里，可以提高压缩率，减少网络传输。唯一需要仔细考虑的地方就是 package 的 id，必须要对应好。

```

private static ByteBuffer compressMysqlPacket(byte[] data, AbstractConnection
con,ConcurrentLinkedQueue<byte[]> compressUnfinishedDataQueue)
{

    ByteBuffer byteBuf = con.allocate();

    byteBuf = con.checkWriteBuffer(byteBuf, data.length, false);

    MySQLMessage msg = new MySQLMessage(data);

    while (msg.hasRemaining())
    {
        int i1 = msg.length() - msg.position();

        int i = 0;

        if (i1 > 3)
        {
            i = msg.readUB3();

            msg.move(-3);

```

```

}

if (i1 < i + 4)
{
    byte[] e = msg.readBytes(i1);
    if (e.length != 0)
    {
        compressUnfinishedDataQueue.add(e);
        //throw new RuntimeException("不完整的包");
    }
} else
{
    byte[] e = msg.readBytes(i + 4);
    if (e.length != 0)
    {

        if (e.length <= 54)
        {
            BufferUtil.writeUB3(byteBuf, e.length);
            byteBuf.put(e[3]);
            BufferUtil.writeUB3(byteBuf, 0);
            byteBuf.put(e);

        } else
        {
            byte[] compress = compress(e);
            BufferUtil.writeUB3(byteBuf, compress.length);
            byteBuf.put(e[3]);
            BufferUtil.writeUB3(byteBuf, e.length);
            byteBuf.put(compress);
        }
    }
}

```

```

    }

    }

}

return byteBuf;
}

```

//将 mysql 的压缩协议包解压成普通的协议包

//这里主要考虑的地方是一个普通的协议包可能或多个压缩包，一个压缩包里可能有多个普通包，其中有可能有不完整的包，所以利用 decompressUnfinishedDataQueue 的队列来暂时存储

```

    public static List<byte[]> decompressMysqlPacket(byte[] data, ConcurrentLinkedQueue<byte[]>
decompressUnfinishedDataQueue)
    {
        MySQLMessage mm = new MySQLMessage(data);
        int len = mm.readUB3();
        byte packetId = mm.read();
        int oldLen = mm.readUB3();
        if (len == data.length - 4)
        {
            return Lists.newArrayList(data);
        } else if (oldLen == 0)
        {
            byte[] readBytes = mm.readBytes();
            // return Lists.newArrayList(readBytes);
            return splitPack(readBytes, decompressUnfinishedDataQueue);
        } else

```



```

{
    byte[] de = decompress(data, 7, data.length - 7);
    return splitPack(de, decompressUnfinishedDataQueue);
}
}

```

//从流中分割出协议包，主要为了判断 packID

```

private static List<byte[]> splitPack(byte[] in, ConcurrentLinkedQueue<byte[]>
decompressUnfinishedDataQueue)

```

```

{
    in = mergeBytes(in, decompressUnfinishedDataQueue);

    List<byte[]> rtn = new ArrayList<>();
    MySQLMessage msg = new MySQLMessage(in);
    while (msg.hasRemaining())
    {
        int i1 = msg.length() - msg.position();
        int i = 0;
        if (i1 > 3)
        {
            i = msg.readUB3();
            msg.move(-3);
        }
        if (i1 < i + 4)
        {
            byte[] e = msg.readBytes(i1);
            if (e.length != 0)
            {
                decompressUnfinishedDataQueue.add(e);
            }
        }
    }
}

```

```

    }
} else
{
    byte[] e = msg.readBytes(i + 4);
    if (e.length != 0)
    {
        rtn.add(e);
    }
}
}

```

```

return rtn;
}

```

```

private static byte[] mergeBytes(byte[] in, ConcurrentLinkedQueue<byte[]>
decompressUnfinishedDataQueue)

```

```

{
    if (!decompressUnfinishedDataQueue.isEmpty())
    {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try
        {
            while (!decompressUnfinishedDataQueue.isEmpty())
            {
                outputStream.write(decompressUnfinishedDataQueue.poll());
            }
            outputStream.write(in);
            in = outputStream.toByteArray();

```

```
        outputStream.close();

    } catch (IOException e)
    {
        throw new RuntimeException(e);
    }
}

return in;
}
```

第 14 章 Mycat 外传-群英会

14.1 我不做大哥很多年



曾经年少时不更事，为古惑电影所迷，遂取网名为南哥，有一阶段成了传说中的南哥，后来又从传说中跑了回来，一不小心混了 10 年的 Java 编程经验。多年前被女同事称作大哥，不过已经好久没听人这么说了，哎，我不做大哥好多年，看来我是逆生长了，越活越年轻。

好老庄，曾跟道家老师学习过。想强身，也曾一时兴起练过一段时间 MMA。现就职于杭州某公司副总工程师。在公司内部经常捣鼓一些框架，把玩一些新技术。从开源项目获益良多，mycat 比较合胃口，所以贡献过多自动迁移、zk 协调切换、数据库分页语法支持、load data、压缩协议等功能，这里也要感谢那些反馈 bug 的网友。也欢迎大家一起交流 magicdoom@gmail.com QQ: 84436109

14.2 冰风影



不经意间就发现工作了十多年了，曾经精通过 Delphi,后来转 java，在一小公司干了七八年，从 coder 做到技术总监，带 15 人左右的研发团队，也曾经弄过一段时间的页游。定居在广州后，发现找管理方面的工作，大公司不要，小公司待遇低，无奈之下再次成为 coder,架构师。目前主要的研究方向是大数据，分布式技术，Hadoop,Hive,Hbase,spark,业余时间打算贡献给 mycat。

爱好广泛，摄影等都有接触过，专门学习过资本交易，对股票，基金，保险，港股，期货，外汇，股权投资都有接触。

港股开户找我(免开户费),欢迎大家一起技术交流,可邮件联系:sunsoft@qq.com。

14.3 从零开始



古语云 一生二，二生万物，而代码世界就是由 1 的世界构成，通过代码你可以构建无限可能，程序员不是屌丝，程序员有自己性格，是有热情的有志青年。

从零开始仅是代码世界普通一员，励志在技术的世界里寻得自己的天地，苦学代码六年载，然未有所成就，再此留言实属惭愧。做过电信行业财务，银企支付等系统，现在于上海某公司做养老金融，研究数据搜索与分布式处理。

我想有一所房子，面朝大海，春暖花开，10M 宽带，能叫外卖，快递直达，不还房贷。

喜欢爬山，喜欢游泳，喜欢户外大自然的气息。

博客地址就是传说中的从零开始:songwie.com

14.4 黑白咖啡



一个大雨滂沱的夜晚,灯光氤氲在雨中,黯淡了街道。我独坐在电脑前面注册人生的第一个 QQ, 为 QQ 名所伤神, 突然一道春雷在我耳边炸响, 看着黑白的世界, 黑白咖啡四个字突然跳入了我的脑海。从此与咖啡结下了不解之缘, 也因此与 Java 展开了热恋。

爱好广泛, 音乐体育动漫自是不在话下, 《黄帝内经》、《道德经》、《庄子》、《孙子兵法》都拜读过, 家中至今收藏多部武学巨著, 如易筋经洗髓经^_^曾经就职于某外包公司, 拥有近两年的电信行业的开发经验。目前就职于南京某公司, 主要的研究方向是大数据, 分布式技术。

14.5 石头狮子

尘世中学习型小码农一枚, 目前正在努力打怪升级中。

典型 90 后, 崇拜卢梭, 亚当斯密。喜欢新鲜技术, 爱折腾。

践行每天开源一小时, 读开源项目源码, 翻译些英文文档, 写点总结。:)

被 leader 忽悠成为 Mycat 团队中的酱油党, 努力向各位大牛学习中。



14.6 Rainbow



从传统企业应用管理系统入门、成长于电商系统的架构与开发实现，目前从事大型国企的私有云建设中技术架构主导。已经在 JAVA Web 领域工作 8 年有余。从 coder 到架构师，随着职位变化从事工作也发生很大改变。但目前依然在一线 coding。因为很享受 coding 所带来的快乐！目前主要研究应用架构(单一服务架构、垂直应用架构、分布式应用架构、流动计算架构)、PAAS 平台、应用系统监控与管理。在业余时间主导并参与 Mycat-web 系统开发。

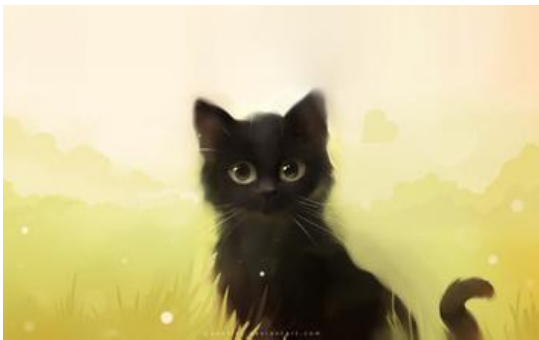
个人爱好吃、唱歌、旅游、看电影(科幻类, 感悟人生类)。

技术源于生活, 以生活思考业务, 基于业务思考技术, 以艺术家对技术落地!

有兴趣对 JAVA WEB 架构, Mycat-Web 的朋友, 请联系: QQ:270300639 e-

mail:accp_huangxin@163.com

14.7 Mycat 铁杆粉丝



你能想到写库用 InnoDB, 而读库用 MyISAM 的主从复制和读写分离方案么?

他想到了, 而且证明了这种模式的性能。

他就是 Mycat 的铁杆粉丝之一, 杨超。

杨超, 软件工程师科班毕业, 1987 年生, 祖籍福建龙岩, 目前就职于北京新华安徽数据服务有限公司, 职位研发部经理, 经过对 mycat 的研究和使用, 首创 mycat 读写分离模式的智能优化方案, 并成功将 mycat 已经正式应用于我公司的项目(社会辅助征信系统)中, 目前系统在生产应用环境中已稳定运行快半年有余, 截止今日系统未重启过。

14.8 兵临城下



从 09 年开始参加工作至今已 6 个年头了，从 GPS 定位到移动医疗再到电子商务，这一路走来让我收获颇多，也让我对人生有了一个深刻的认识，人生的路，需要自己一步一步的走过你才知道，没有对也没有错，人生需要这些挫折和成长。坚信梦想，不断努力，梦想一定可以实现。

以上来自我从业 6 年来真实的经历和感悟，目前国内某大型电商企业担任高级开发工程师。

14.9 我是谁



取名是个麻烦事，曾先后用过“runfriends”、“我是谁”、“一坨”、“哇咔咔”等网名，现在我也不知道该署哪个名了。

当年脑子进水选了生物工程专业，后来脑子水干了，做了程序员。本来以只有 C 程序员才是最牛逼的，却又稀里糊涂做了 JAVA 程序员。

不知不觉，一晃 N 年。转战 IT 领域的多个行业，曾一度迷茫，不知路在何方，今夕何夕，不知“我是谁”，最终进入互联网。

现在一家特别小的互联网公司写写代码，在各技术群里吹吹牛逼。有时也能被人叫做大神，不免有点小得意。不过，看看各个 QQ 技术群里那么多更大的神，又深觉自己功力不够。

后来技术领域令我赞叹的事情越来越多，每天在心里响起无数个“哇咔咔”。

现在 MYCAT 技术团队，打打酱油，跟各位大神学习。

欧耶。

14.10 当太极遇到 AK47



古老的东方神功太极拳，遇到了兵器之王 AK47，谁胜谁负，或者是可以合二为一？

沧海一声笑，两岸水滔滔，估计就是身怀绝技的武者手持 AK47，一排子弹扫过黄浦江时候溅起的浪花朵朵....

和木，和气和睦(木)之意，十年 IT 从业经验，JAVA 程序员、架构师、技术型产品经理。

- 精通于网络编程、高性能高并发架构设计、JAVA 性能调优。
- 敢拼敢闯，小布什打萨达姆期间上过美伊战场打过 AK-47 突击步枪。
- 特长跑步，在校、院系、公司运动会 3000、5000 米比赛中，基本上第一名或至少前三。
- 爱好太极拳，陈式太极 14 代传人，曾获得全国武术比赛太极拳青年组银牌（2014）。
- 表达能力尚可，曾在某大学担任兼职讲师，讲授了一学期的管理类课程。
- 管理能力尚可，毕业两年即担任 10 人开发团队项目经理，不过目前走技术路线，习惯于带领 3~6 人的精英

技术团队打造精英产品。

- 有一定的设计能力，曾获得华为 UCD 设计比赛一等奖。
- 有一定的创新能力，有两篇技术发明专利（全部为第一发明人）。
- 一定的算法应用能力，能把数学应用于生产，曾设计了中国电信计费网的核心路由算法。
- 团队合作能力较好，在前几个东家打工时，所在团队多次获得优秀团队、金牌团队称号。
- 综合解决能力较强，在某过千万级软妹子的项目中，单兵负责了投标、需求、设计、开发、联调、运维一整

条线近 70%工作。

开源感言：

过去十年，一直做闭源项目，未曾贡献开源力量，对自己这种只取不予的行为深感羞愧；这次应 Leader 之约，写了开发篇的第三、四、五章的内容，时间匆忙，如有错误请及时指正，本人定会虚心接受；也算是痛改前非，尽微薄之力为开源社区添砖加瓦！

14.11 传说中的 Mycat 大美女



非技术女，IDC 销售 1 枚~本着对中国开源的支持，被 Leader 成功忽悠参与 Mycat.....自传哪里是 200 字可以写得完的？不如就借一段较为接近的文字介绍一下自己吧：“忘掉远方是否可有出路 忘掉夜里月黑风高 踏雪过山双脚虽渐老 但靠两手一切达到，

见面再喝到了熏醉 风雨中细说到心里 是与非过眼似烟吹 笑泪渗进了老井里，

上路对唱过客乡里 春与秋撒满了希冀 夏与冬看透了生死 世代辈辈永远紧记，

忘掉世间万千广阔土地 忘掉命里是否悲与喜 雾里看花一生走万里 但已了解不变道理

.....”

14.12 Mycat 至尊酱油师



我，Michael（大家叫我英文名字比较好，真名不多说...），摩羯座，IT 人。

当年为了祖国的花朵健康成长不误人子弟，毅然放弃了神圣的教师职业（→_→物理学），转行开始做苦工搬代码。

纯酱油师出生，没学过什么编译原理、汇编语言、数据结构..... 一个机遇+一个 RP 爆发，成功走上了 J2EE 编程开发之路，当然不知道这是上苍眷顾我还是要惩罚我，因为编程既是一个时刻充满挑战又是一个不归路，还好在良（hu）师（peng）益（gou）友（you）

的帮助下混了个架构师职位，从事着外人看起来高大上的工作，其实依然很苦逼....，最近转行搞火热的大数据技术，这个才真算有点高大上的味道^_^。

本人不是什么大牛，只是比较好学，没事就喜欢捣腾一些新技术，也很乐意和志同道合的朋友喝茶聊天，当然不局限于技术。

目前在上海浙大网新易得任职研发总监，主要从事大数据相关工作，打个小广告有想转行搞 hadoop 相关的可以和我联系：

blog: www.micmiu.com email: sjsky007@gmail.com weibo: <http://weibo.com/ctosun>

14.13白衣公子



凭虚公子，目前于某电信软件供应商供职，负责数据架构方面工作。在进行系统去 IOE 的过程中，选择了 Mycat，通过业务场景的分离和 substring 的分片方式，实现了可在线扩容的数据库集群架构。并将 Mycat 应用于运营商系统中，目前集群中数据量约 6 亿左右，系统已在线稳定运行一年以上，后续还将在更多的去 IOE 实践中使用。目前正在计划中的大型业务系统还有两个。

14.14 他入错了行



他入错了行。

他本来是应该做营销的，传销估计也行。

他最擅长的武功大概是装作泥害的样子，把别人忽悠到发呆，然后，滔滔不绝的宣讲他的理念。

他还真做到了，于是，你才有机会看到这本书，中国第一本开源项目发起的众筹预售电子书。

他就是 Leader-us，一个极具营销意识的 S 级编程王架构师。

他忽悠出了一个 Mycat 开源社区，然后这个社区成为国内大数据编程领域最有实力的社区，这里有颜值很高的新锐小清新，也有深藏不露的资深架构师，如果有一天风头慕名而来，你也不用诧异，因为你可能也是被风头看中的一员大将。

说了这么多，还是看看 Leader-us 出神入化的忽悠神功吧，下面这句是他为网上开设的课程《大型分布式系统架构实践》的所写的无敌广告。

等学完 Leader 这门课程，同学们的营销水平就达到阿里的 P8+ 了。

"自从我跟着 Leader 花了 3 个月，挑灯夜战，把这门课学完以后，小宇宙爆发，人气爆表，漂亮妹子们潮水般的扑过来，甩都甩不掉....." ——某学员的痛苦心声。

14.15 烟花易冷-奎



多年的 JAVA WEB 开发经验，技术处女座。在恰当的场合使用恰当的互联网词汇，并且能一本正经逗你！

爱好广泛！联系方式 QQ:294548915。

14.16 海王星

外号小强。

是个很敬业的程序猿，但也是个活泼的大男孩~热爱编程~也热爱游戏~

从毕业开始一直从事这 JAVA 的开发工作，对着代码有着很深的执着~可以为了代码不吃不喝不睡。

很喜欢鼓捣一些奇奇怪怪的东西。

目前开始成为一个刚入门的架构师，开始想着众多的大牛们努力的学习。

无意间接触 Mycat,参与了 Mycat 的开发、测试。

目前是 Mycat 幕后的神秘人物，作为 QA 人员，负责代码的质量分析，自动化构建，版本规划等工作~



14.17 太极鸟人



大器当晚成，童年时贪玩还没开始记事经常爬树掏鸟窝，也从树上掉下来摔晕过，其实我是爬累了，掉下来躺着舒服就多睡了一会。小学一二年级都留级了，因为没考及格学校不让升级。第二个二年级我终于觉醒了，虽然还是天天爬树掏鸟下河抓鱼，但总能长期占据班级头名。从事软件行业后的前 3 年基本是打酱油的，2008 年毕业至今，现在也 7 年了，最近的一两年才开始发力，也许我才刚刚觉醒，一位大神即将诞生。

本人爱好比较广泛：喜爱太极拳，但还没入门，曾经在大学跟随吴氏太极拳第 4 代传人战波老师傅学习了一个月吴氏太极老架。喜爱养信鸽，小学掏鸟窝养野鸽子开始，然后转成养家鸽，最后开始养信鸽，养了十年左右，现在没条件养了，以后必然会继续。弹弓是我从小的爱好，目前兜里随时揣着一把弹弓，不敢说百步穿杨，但敢 5 米打某人头顶上的苹果，如果你敢当模特顶着苹果的话。

联系方式 QQ:152974495 邮箱: wdw1206@163.com

14.18 成都-顽石神



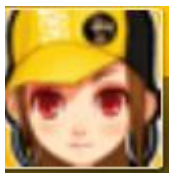
Mycat PMO 之一 他本是 DBA，却喜欢研究各种架构。从 DB 维护到数据架构，其间经历思考方式上的巨大转变，铭记领路人指点；从传统 DB 到 NOSQL，再到大数据；从图形化报表到 DB 监控工具，再到 DB 中间件，感觉工作中越来越不能没有 CODE。无意间知道 Mycat，被它的架构深深吸引，进而加入志愿者大军，开始贡献代码，成了 PMO。目前主要方向是 Mycat+Mysql 性能优化和 Mycat 推广，有兴趣的朋友可一起技术交流，Mail:250721835@qq.com。

14.19 杭州-白



Mycat PMO 之一 大学在读学生，一直怀着没被踢出群感恩的心的酱油党，努力学习中希望早日为 mycat 贡献代码，也希望 mycat 会在 leader-us 的带领下走的更远。

14.20 allnet-深海



Mycat 志愿者，高级 java 开发人员，熟悉 mycat 目前负责南京某项目 Mycat 本地方案实施。

14.21 明明 Ben



从事软件开发 10 年，号称 CODE、架构、管理、忽悠 等无一不通，北京打拼多年， 14 年回到合肥，参与创建了一家 O2O 公司“一米兼职”，虽如此，但从未远离热爱的编码。

14.22 上海-袁文华



已经做 Java 十六年了，拿过 Oracle OCP。在公司主要负责系统架构，爱好接受新鲜事务，玩过 python, ruby, css, js, android, html5。八字，风水，中医也都有研究。年纪虽长，但入门 Mycat 算最晚。幸得 Leader us 的信任，负责 Mycat eye 的核心开发。希望能在 Mycat 社区共享自己的一份力。也打个小广告，在上海，有想招人的请联系我，QQ:1935326097。

14.23 杭州-yuanfang



大学时，曾经梦想着做大型网络游戏，狂热学习 C++，读遍了当时所有的 C++ 名著。然而毕业找工作，却掉入了 Java 的坑中。工作已 6, 7 年。待过日企，做过 web 网游，维护过 Oracle(10g)，目前从事 MySQL DBA 工作。技术很杂，几乎成了所谓的“全栈”。目前工作主要还是以维护 MySQL 为主，偶尔用 Java 打打酱油。曾向 Leader-us 提过一些关于 Mycat 的建议，Leader-us 忽悠我自己实现，so, 加入了 Mycat 开源大家庭。本人热爱开源技术，尤其是 MySQL 相关技术，喜交流，有志趣相投者，请联系：digdeep@126.com；QQ:1981715364。

14.24 胡雅辉



望舒，仙剑迷，java 程序员，喜欢捣鼓新东西，现在在一湖南本土电商企业捣鼓营销中心。贡献 mycat1.6 的 pg 原始协议支持。

14.25^{KK}



KK，爱好，coding, coding 还是 coding, 经常为了一个错误奋战到半夜，为了那一个错误的解决而额外的高兴，喜欢与人分享代码，自己的一点经验，代码总是在一个又一个的坑中去成长。在 mycat 的这个编码的大家庭中，我编写了 zk 与 mycat 的相互协调的工作，QQ: 546064298，愿与更多的人一起交流代码。

14.26CrazyPig



CrazyPig，很疯狂的猪，有时也用 ZzzCrazyPig，名字只是以前玩游戏的常用名。目前就职深圳某快递科技公司。典型很宅的程序员，专攻 Java，涉猎过大数据、分布式数据库中间件等。对开源颇有好感，毕业以来有幸接触 Mycat，由此展开了一段新的历程！目前仅为 Mycat 社区贡献预处理功能代码，并为 Mycat 修复若干 BUG。希望后面跟着 Mycat 大神们，继续为 Mycat 贡献代码。

14.27传说的学霸

xué



bà



Technology Learners&&Practitioners

2010 年毕业，大部分时间在从事 Linux 下 C/C++ 开发，业余玩玩 OpenWrt，后面转行做大数据 Spark 开发，阅读过 Spark 1.5.1 内核，在 Leader-us 的领导下研究 Mycat，主要为 mycat 贡献了 Java 堆外内存 (off-heap) + 磁盘的方式处理 SQL 结果集汇聚，使用这种方式让 Mycat 有处理亿级规模 merge/order/group/limit 数据结果集的能力。

未来希望为 Mycat 开源事业在中国发展壮大贡献一份力量。

目前就职顺丰科技，主要从事 Mycat 分布式数据库中间件研发技术研究方向：

Mycat 中间件，分布式存储与计算，分布式数据库。

联系 QQ: 547557645 网名‘零’。

14.28 毛茸茸的逻辑



mycat 资深用户，原 sybase 数据库高级研发工程师，目前参加 mycat1.6 的开发。

14.29 深圳-Java-HelloWorld



mycat 源码贡献者，1.6 的 xa 分布式事务功能开发者。

qq: 826252919