

第一版

中国第一开源社区出品



# Mycat 权威指南

Mycat

开源项目组 著



# 入门篇

## Mycat开源宣言



不怕从零开始  
只怕从未启程

Mycat + You

随着技术的不断进步，是否应该有一种比公司形态更有效的组织来支撑经济的进一步发展？

这种新型组织在以有形资产为核心的，以农业经济和工业经济为主导的社会是不可能取得成功的，而在以无形资产逐渐成为核心的，以知识经济为主导的信息社会将会成为可能。如国内崛起的分布式数据库中间件产品Mycat并不是由任何一家公司主导开发的，而是由民间自发组织的由那些喜爱它的不知名的程序员共同开发，如今该产品的发展速度极快其影响力也逐渐扩大。

国内外类似的开源组织和产品还有很多，这些开源产品潜力无限，无论开发效率和质量都逐渐超越任何一家公司的产品。这也导致了一些公司试图通过收购等手段遏制开源产品的发展。那么这些开源产品爱好者和贡献者获得了什么呢？在「无私奉献」的过程中他们获得了知识——信息社会最有价值的资产，他们可以用这些知识以任何形式换来不可估量的财富，信息社会的开源组织使「按劳分配」达到了前所未有的公平与公正。

企业所采取的期权激励、扁平化管理、自由工作时间等模式，正是对公司这种生产关系「自顶向下」的改良，以适应持久技术进步带来的生产力的高速发展。但公司的本质：追求股东利益最大化，使其不可能实现真正意义的去中心化。

信息社会的开源组织形态是对原有公司模式「自底向上」的一次颠覆式创新，他们将带来生产力的极速发展。这种组织先天具有开放、共享、敏捷、去中心化等等这些可以带来高效率的特性，可以想像拥有杰出技术与高效团队的开源组织可以创造出超越一切公司的更优秀的产品。

「每一个人为改变他的状况而自然做出的努力，当其具有施展的自由和安全时，就是一个十分强有力的原则，不需要借助其他，这种个人的努力，就能给社会带来财富和繁荣」亚当斯密的这段话是工业革命中的小公司向拥有国家特许经营权的垄断企业发出的呐喊。没有工业革命就没有现代公司存在的必要性；没有现代公司的存在和发展，工业革命的快速进程也无法出现。历史总在重演，信息社会的开源组织将亚当斯密这段话原封不动的回赠给了现代公司制度。它让知识经济不再只是少数资本家的游戏，而成为普通人登台表演的机会。技术不再高高在上，而是落地生根。开源组织将成为引领信息社会进步的发动机，接下来的竞争，就看谁能在无限的数字世界里更好的发挥开源组织的能量了，一个新的时代即将到来！

随着信息技术的持续快速发展和中国经济实力的不断加强，以Mycat为代表的中国开源组织和产品的价值和发展前景不可限量！

——By 正能量

# 概述

## 数据库切分概述

### 数据切分概述

# 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

# 关系型数据库和NoSQL数据库

针对上面两类系统有多种技术方案，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

<1>关系数据库的特点是：

- 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。

<2>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。

<3>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。

#### NoSQL数据库

<1>NoSQL数据库的特点是：

- 非结构化的存储。
- 基于多维关系模型。
- 具有特有的使用场景。

<2>优点：

- 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。

<3>缺点：

- 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目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，所以就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。

## 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。

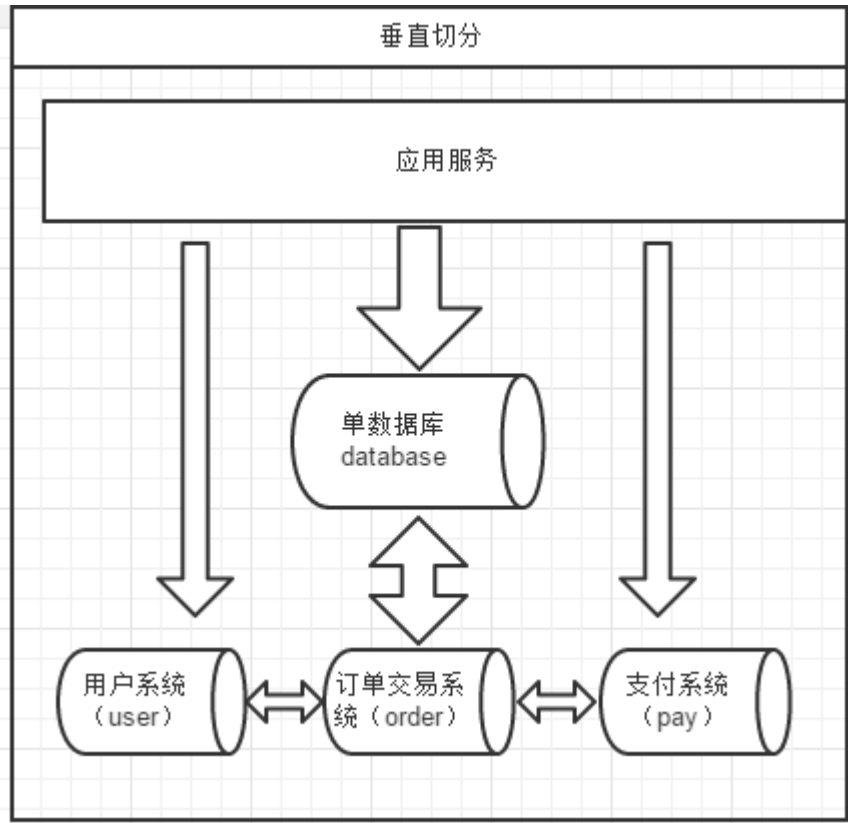
数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。一种是按照不同的表（或者Schema）来切分到不同的数据库（主机）之上，这种切可以称之为数据的垂直（纵向）切分；另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上面，这种切分称之为数据的水平（横向）切分。

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。

## 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。

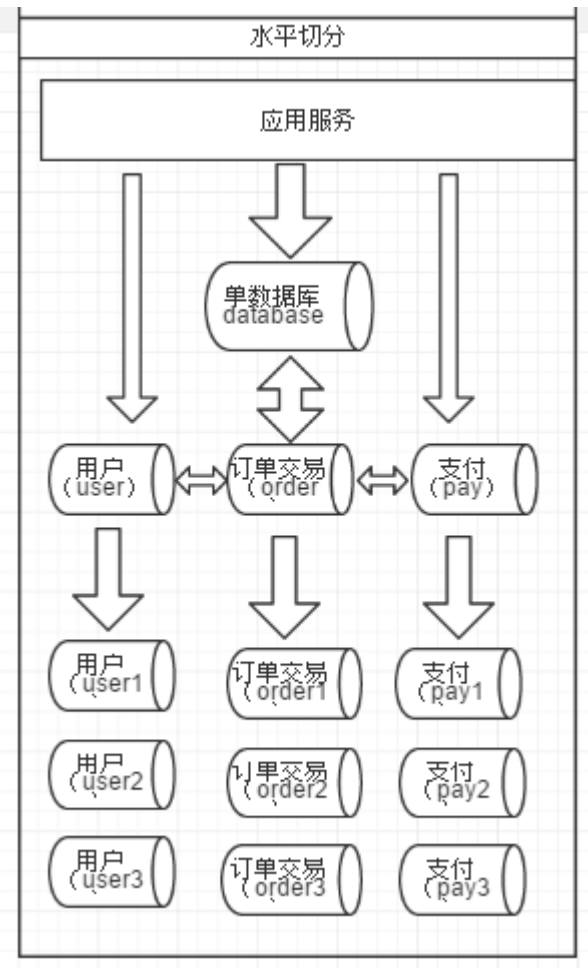
缺点：

- 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。

## 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：



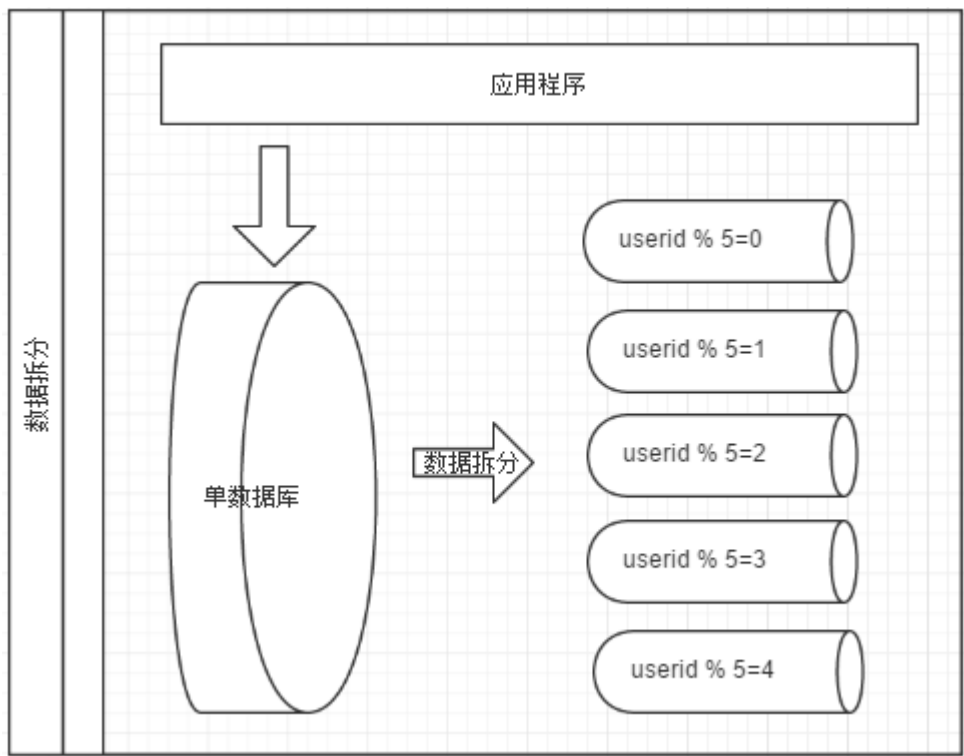
拆分数据就需要定义分片规则。关系型数据库是行列的二维模型，拆分的第一原则是找到拆分维度。比如：从会员的角度来分析，商户订单交易类系统中查询会员某天某月某个订单，那么就需要按照会员结合日期来拆分，不同的数据按照会员ID做分组，

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，则会有一定的困难。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。

优点有：

- 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。

缺点有：

- 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。

## Mycat前世今生

### 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

## 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。

## Cobar的十个秘密

## 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。

## 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。

## 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：

1. Cobar启动的时候，会用默认第一个Datasource进行数据读写操作；
2. 当第一个Datasource心跳检测失败，会切换到第二个Datasource；
3. 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？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。

## 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。

## 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>
```

## 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 $0.1\% \times 4\% =$ 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。

## 第六个秘密：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”。

## 第七个秘密：出乎意料的连接池

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。。。

## 第八个秘密：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。

## 第九个秘密：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。

## 第十个秘密：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。

## 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opencloুদ্ধdb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24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>

## Mycat概述

### 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，就好像是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：

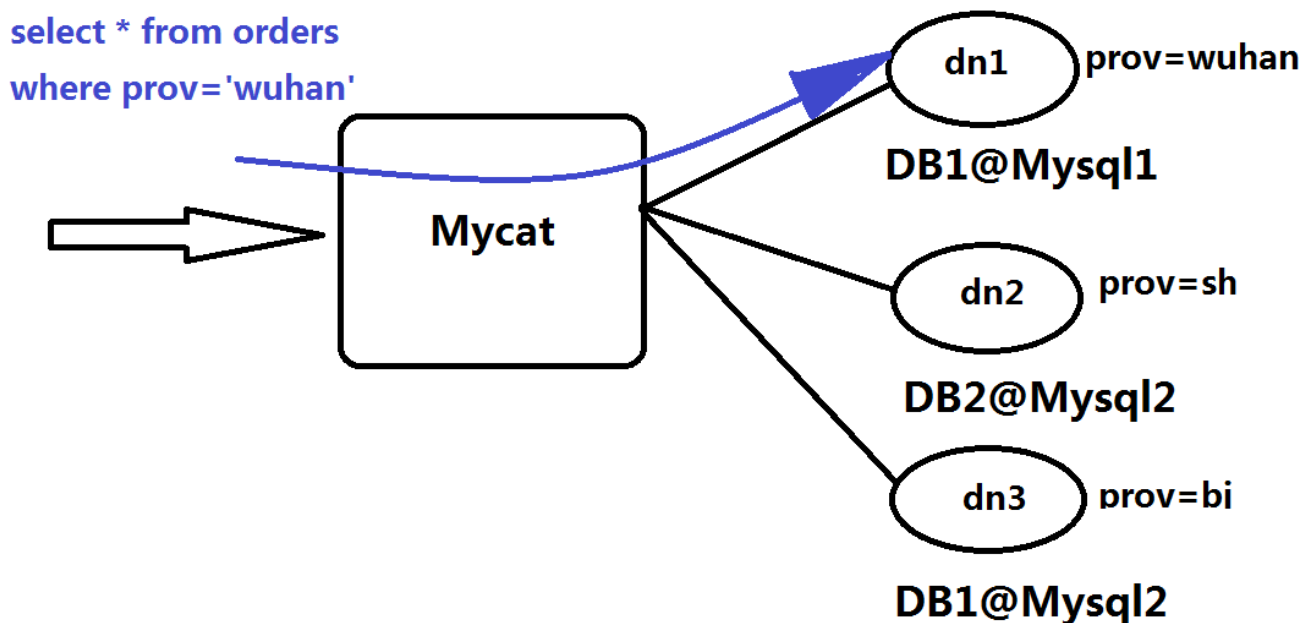
Mycat是一个强大的数据库中间件，不仅仅可以用作读写分离、以及分表分库、容灾备份，而且可以用于多租户应用开发、云平台基础设施、让你的架构具备很强的适应性和灵活性，借助于即将发布的Mycat智能优化模块，系统的数据访问瓶颈和热点一目了然，根据这些统计分析数据，你可以自动或手工调整后端存储，将不同的表映射到不同存储引擎上，而整个应用的代码一行也不用改变。

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！

## 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

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！

## 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

## Mycat长期路线图

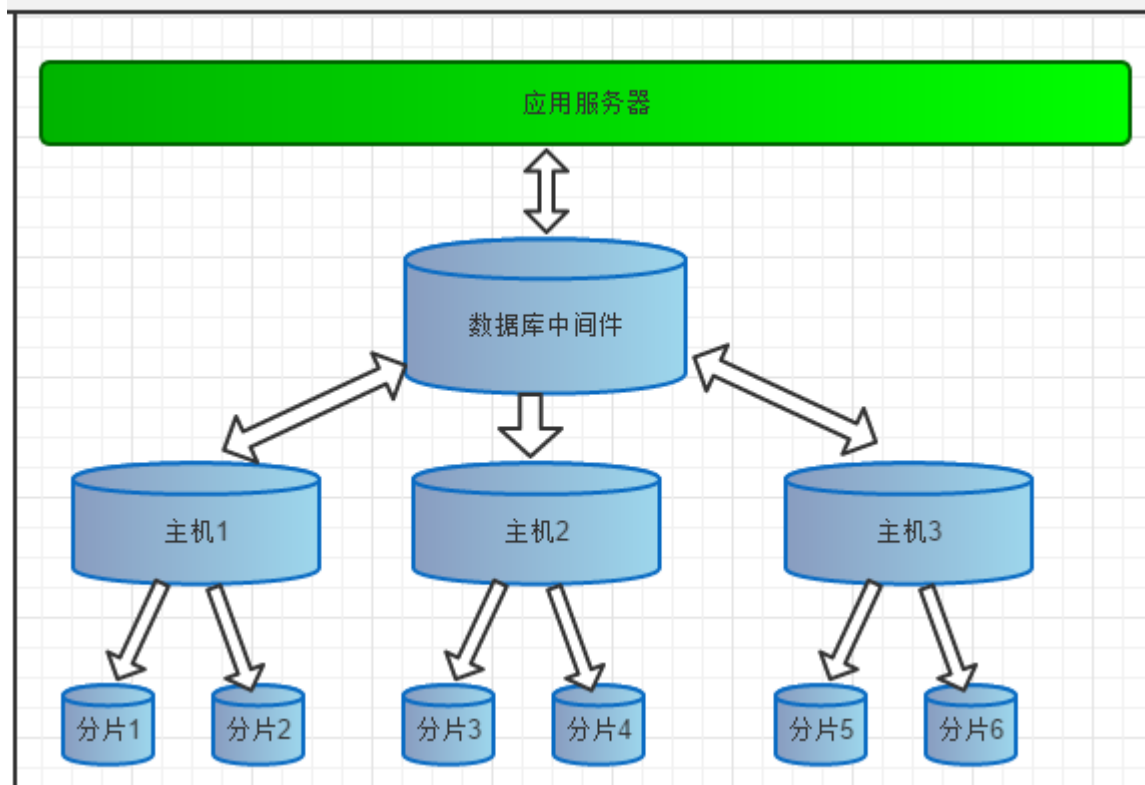
- 强化分布式数据库中间件的方面的功能，使之具备丰富的插件、强大的数据库智能优化功能、全面的系统监控能力、以及方便的数据运维工具，实现在线数据扩容、迁移等高级功能
- 进一步挺进大数据计算领域，深度结合Spark Stream和Storm等分布式实时流引擎，能够完成快速的巨表关联、排序、分组聚合等 OLAP方向的能力，并集成一些热门常用的实时分析算法，让工程师以及DBA们更容易用Mycat实现一些高级数据分析处理功能。
- 不断强化Mycat开源社区的技术水平，吸引更多的IT技术专家，使得Mycat社区成为中国的Apache，并将Mycat推到Apache基金会，成为国内顶尖开源项目，最终能够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

## Mycat中的概念

### 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。

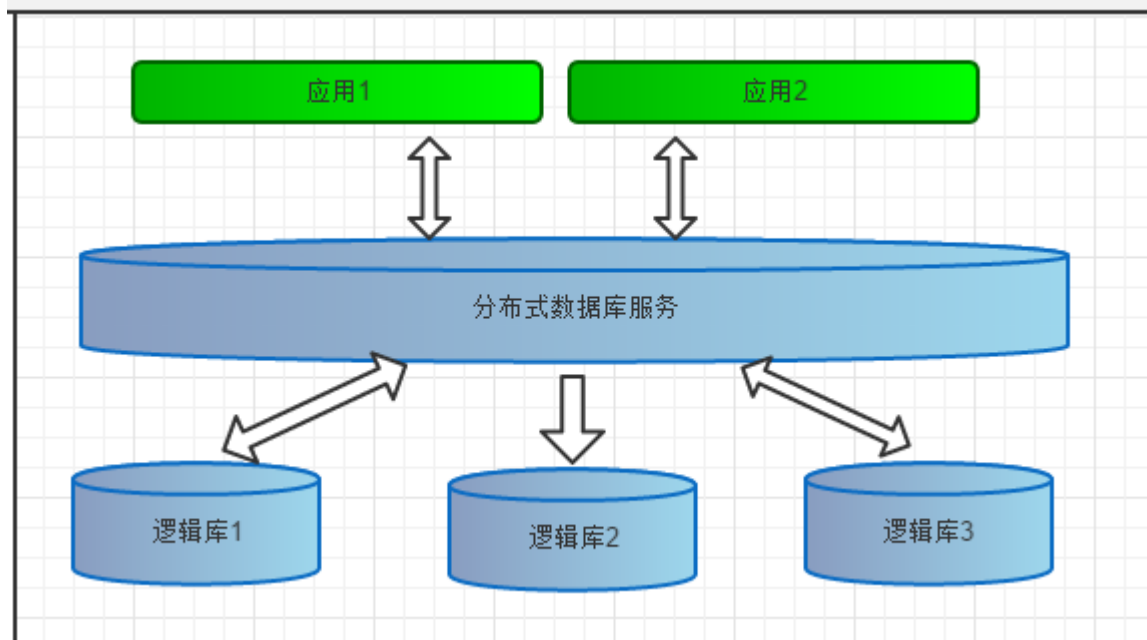
## 逻辑库(schema)

### 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。





## 逻辑表 ( table )

### 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。

### 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" />
```

### 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" />
```

## 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。

## 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。

## 分片节点(dataNode)

### 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）。

### 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）。

## 分片规则(rule)

### 分片规则

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。

## 全局序列号(sequence)

### 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）。

## 多租户

### 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

#### 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。

### 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，但一个Tenant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.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：

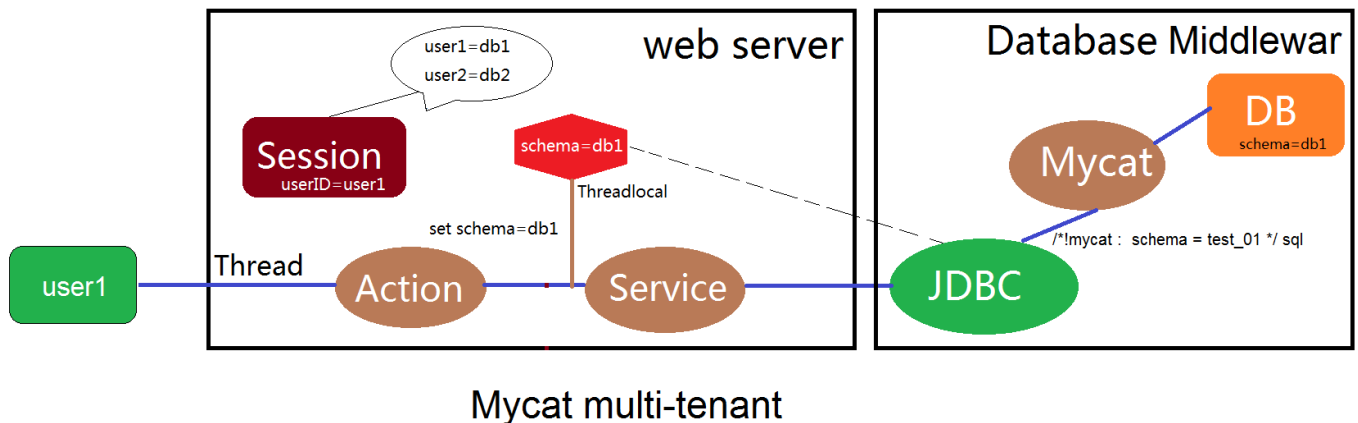
三种方案比较，第三种方案的维护和购置成本最低，允许每个数据库支持的租户数量最多。

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。



## 快速入门

### 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。

#### 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。

## 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，需要先在本地上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

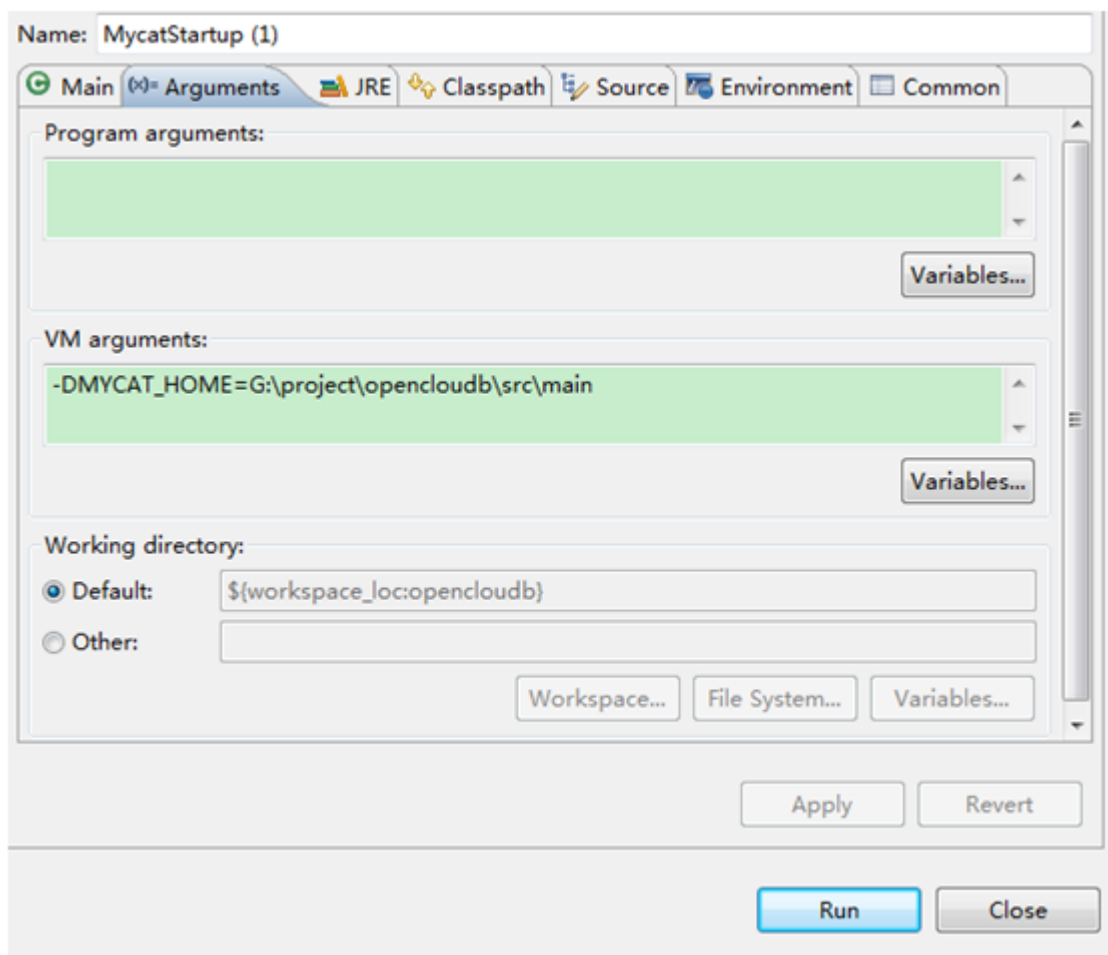
MyCAT目前主要通过配置文件的方式来定义逻辑库和相关配置：

- 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org.opencloudb.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

### 3.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：

<http://pan.baidu.com/s/1o61EXaa>

## 2.2 服务安装与配置

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

```

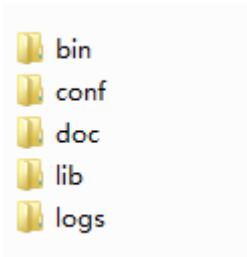
/usr/local/Mycat/
├── bin
├── conf
├── doc
├── lib
└── logs

useradd Mycat
chown -R Mycat.Mycat /usr/local/Mycat

```

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.

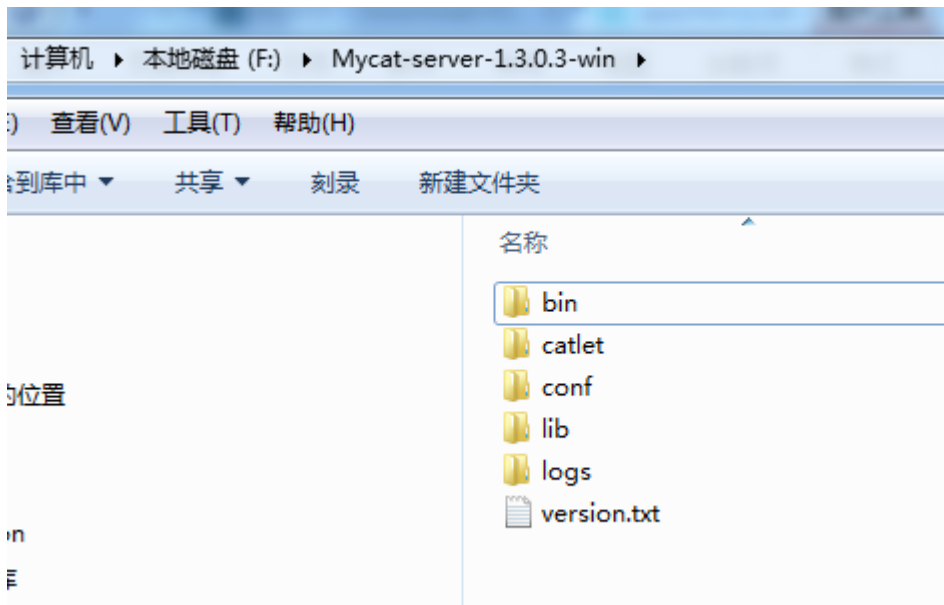
日志存放在logs/mycat.log中，每天一个文件，日志的配置是在conf/log4j.xml中，根据自己的需要，可以调整输出级别为debug，debug级别下，会输出更多的信息，方便排查问题.

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！

## 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 console 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目录下执行命令 mycat install 执行安装mycat服务.

- 2) 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.

## 2.3 服务启动与启动设置

### linux

1.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。

1. 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：

- 1) 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！

## 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：

从cmd中执行命令到达 D:\Mycat-server-1.4-win\bin 目录下，执行：

mycat install //表示执行安装MyCAT服务

mycat remove //表示执行卸载MyCAT服务

服务安装完后，就可以通过windows系统服务对MyCAT进行启动和停止了。

## 2.4 demo使用

1) springMVC+ibatis+FreeMarker 连接mycat示例：

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

## 2.5 日志分析

## 日志配置

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。

## 日志分析

### 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



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。

## 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] (DefaultLayeredCachePool.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,
maxStringLength=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=org.opencloudb.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 21:46:59.506 INFO [main] (MycatServer.java:262) -using nio network handler
04-29 21:46:59.530 INFO [main] (MycatServer.java:280) -$_MyCatManager is started and listening on 9066
04-29 21:46:59.530 INFO [main] (MycatServer.java:284) -$_MyCatServer is started and listening on 8066
04-29 21:46:59.530 INFO [main] (MycatServer.java:286) =====
04-29 21:46:59.530 INFO [main] (MycatServer.java:289) -Initialize dataHost ...
04-29 21:46:59.531 INFO [main] (PhysicalDBPool.java:267) -init backend myqsl source ,create connections
total 10 for master index :0
04-29 21:46:59.533 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 21:46:59.536 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 21:46:59.537 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 21:46:59.537 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 21:46:59.537 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 21:46:59.538 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 21:46:59.538 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 21:46:59.538 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 21:46:59.539 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 21:46:59.539 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 21:46:59.598 INFO [$ _NIOREACTOR-1-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=5, lastTime=1430315219133, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=88952, 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 21:46:59.599 INFO [$ _NIOREACTOR-0-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=4, lastTime=1430315219133, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
```

[illegible]

```
fromSlaveDB=false, threadId=17138, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:46:59.971 INFO [$NIOREACTOR-2-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=14, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17134, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:47:00.032 INFO [$NIOREACTOR-3-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=11, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17130, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:47:00.034 INFO [$NIOREACTOR-0-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=12, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17131, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:47:00.035 INFO [$NIOREACTOR-0-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=20, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17136, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:47:00.035 INFO [$NIOREACTOR-3-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=19, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17139, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:47:01.255 INFO [$NIOREACTOR-3-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=15, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17133, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 21:47:01.258 INFO [$NIOREACTOR-0-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=16, lastTime=1430315219133, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17137, charset=utf8, txIsolation=0, autocommit=true, attachment=null,
respHandler=null, host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
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=org.opencloudb.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 [schema=mycat_node1, charset=utf8, txIsolation=-1, autocommit=true]
04-29 21:58:35.413 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 21:58:35.413 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 21:58:35.413 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 21:58:35.414 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 21:58:35.414 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 21:58:35.414 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 21:58:35.415 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 21:58:35.463 INFO [$ _NIOREACTOR-0-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=4, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89015, 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 21:58:35.464 INFO [$ _NIOREACTOR-2-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=6, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89018, 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 21:58:35.463 INFO [$ _NIOREACTOR-1-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=5, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89017, 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 21:58:35.463 INFO [$ _NIOREACTOR-3-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=7, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89019, 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 21:58:35.464 INFO [$ _NIOREACTOR-1-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=1, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89013, 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 21:58:35.464 INFO [$ _NIOREACTOR-2-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=2, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89016, 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 21:58:35.465 INFO [$ _NIOREACTOR-3-RW] (GetConnectionHandler.java:66) -connected successfully
MySQLConnection [id=3, lastTime=1430315915098, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89014, 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 21:58:35.467 INFO [$ _NIOREACTOR-0-RW] (GetConnectionHandler.java:66) -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, schema=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。

## 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 22:06:10.292 DEBUG [$ _NIOREACTOR-1-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=13, lastTime=1430316370226, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=17188, charset=utf8, txIsolation=3, autocommit=true, attachment=dn2{SHOW
STATUS}, respHandler=SingleNodeHandler [node=dn2{SHOW STATUS}, packetId=60], host=116.236.223.115,
port=3307, statusSync=org.opencloudb.mysql.nio.MySQLConnection$StatusSync@7cf13e82, writeQueue=0,
modifiedSQLExecuted=false]
04-29 22:06:10.292 DEBUG [$ _NIOREACTOR-1-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=13, lastTime=1430316370226, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false,
threadId=17188, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.293 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 22:06:10.293 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 -> dn1{SHOW STATUS}
}

```

```

} rrs
04-29 22:06:10.293  DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost
04-29 22:06:10.293  DEBUG [$_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=3, lastTime=1430316370288, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false,
threadId=89066, charset=utf8, txIsolation=0, autocommit=true, attachment=dn1{SHOW STATUS},
respHandler=SingleNodeHandler [node=dn1{SHOW STATUS}, packetId=0], host=121.40.121.133, port=3306,
statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.306  DEBUG [$_NIOREACTOR-3-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=3, lastTime=1430316370288, schema=mycat_node1, old shema=mycat_node1, borrowed=true,
fromSlaveDB=false, threadId=89066, charset=utf8, txIsolation=3, autocommit=true, attachment=dn1{SHOW
STATUS}, respHandler=SingleNodeHandler [node=dn1{SHOW STATUS}, packetId=60], host=121.40.121.133, port=3306,
statusSync=org.opencldb.mysql.nio.MySQLConnection$StatusSync@4bd38cb3, writeQueue=0,
modifiedSQLExecuted=false]
04-29 22:06:10.306  DEBUG [$_NIOREACTOR-3-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=3, lastTime=1430316370288, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false,
threadId=89066, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.313  DEBUG [$_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 [$_NIOREACTOR-3-RW] (MySQLConnection.java:437) -con need syn ,total syn cmd 1
commands SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;schema change:false con:MySQLConnection
[id=1, lastTime=1430316370409, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false,
threadId=89067, charset=utf8, txIsolation=0, autocommit=true, attachment=dn1{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.423  DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost2
04-29 22:06:10.423  DEBUG [$_NIOREACTOR-3-RW] (MySQLConnection.java:437) -con need syn ,total syn cmd 1
commands SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;schema change:false con:MySQLConnection
[id=11, lastTime=1430316370409, schema=mycat_node1, old shema=mycat_node1, borrowed=true, fromSlaveDB=false,
threadId=17189, charset=utf8, txIsolation=0, autocommit=true, attachment=dn2{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.432  DEBUG [$_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=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,

```

```
host=121.40.121.133, port=3306, statusSync=org.opencldb.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], PROVINCE_CODE=ColMeta [colIndex=3, colType=253],
USER_ID=ColMeta [colIndex=0, colType=3], CREATE_TIME=ColMeta [colIndex=2, colType=12]]
04-29 22:06:10.434 DEBUG [$NIOREACTOR-1-RW] (MultiNodeQueryHandler.java:226) -on row end reponse
MySQLConnection [id=1, lastTime=1430316370409, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=89067, charset=utf8, txIsolation=3, autocommit=true, attachment=dn1{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=121.40.121.133, port=3306, statusSync=org.opencldb.mysql.nio.MySQLConnection$StatusSync@7485fef2,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.434 DEBUG [$NIOREACTOR-1-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=1, lastTime=1430316370409, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=89067, charset=utf8, txIsolation=3, autocommit=true, attachment=dn1{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=121.40.121.133, port=3306, statusSync=org.opencldb.mysql.nio.MySQLConnection$StatusSync@7485fef2,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.435 DEBUG [$NIOREACTOR-1-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=1, lastTime=1430316370409, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=89067, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.481 DEBUG [$NIOREACTOR-3-RW] (MultiNodeQueryHandler.java:165) -received ok response
,executeResponse:false from MySQLConnection [id=11, lastTime=1430316370409, schema=mycat_model, old
shema=mycat_model, borrowed=true, fromSlaveDB=false, threadId=17189, charset=utf8, txIsolation=3,
autocommit=true, attachment=dn2{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=116.236.223.115, port=3307, statusSync=org.opencldb.mysql.nio.MySQLConnection$StatusSync@6a95ec91,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.482 DEBUG [$NIOREACTOR-3-RW] (MultiNodeQueryHandler.java:226) -on row end reponse
MySQLConnection [id=11, lastTime=1430316370409, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17189, charset=utf8, txIsolation=3, autocommit=true, attachment=dn2{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=116.236.223.115, port=3307, statusSync=org.opencldb.mysql.nio.MySQLConnection$StatusSync@6a95ec91,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.482 DEBUG [$NIOREACTOR-3-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=11, lastTime=1430316370409, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17189, charset=utf8, txIsolation=3, autocommit=true, attachment=dn2{SELECT *
FROM t_user t
LIMIT 100}, respHandler=org.opencldb.mysql.nio.handler.MultiNodeQueryHandler@3ff70d3c,
host=116.236.223.115, port=3307, statusSync=org.opencldb.mysql.nio.MySQLConnection$StatusSync@6a95ec91,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.482 DEBUG [$NIOREACTOR-3-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=11, lastTime=1430316370409, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=17189, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.484 DEBUG [BusinessExecutor6] (DataMergeService.java:130) -prepare mpp merge result for
select * from t_user t
04-29 22:06:10.485 DEBUG [BusinessExecutor6] (MultiNodeQueryHandler.java:287) -output merge result ,total
data 16 start :0 end :100 package id start:6
04-29 22:06:10.485 DEBUG [BusinessExecutor6] (MultiNodeQueryHandler.java:308) -last packet id:23
04-29 22:06:10.485 DEBUG [BusinessExecutor6] (DataMergeService.java:312) -clear data
04-29 22:06:10.491 DEBUG [$NIOREACTOR-3-RW] (ServerQueryHandler.java:64) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]SHOW STATUS
04-29 22:06:10.491 DEBUG [$NIOREACTOR-3-RW] (DataMergeService.java:312) -clear data
04-29 22:06:10.492 DEBUG [$NIOREACTOR-3-RW] (NonBlockingSession.java:118) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]SHOW STATUS, route={
1 -> dn2{SHOW STATUS}
} rrs
04-29 22:06:10.492 DEBUG [$NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost2
04-29 22:06:10.492 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=12, lastTime=1430316370489, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=17186, 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 22:06:10.554 DEBUG [$_NIOREACTOR-0-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=12, lastTime=1430316370489, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17186, charset=utf8, txIsolation=3, autocommit=true, attachment=dn2{SHOW
STATUS}, respHandler=SingleNodeHandler [node=dn2{SHOW STATUS}, packetId=60], host=116.236.223.115,
port=3307, statusSync=org.opencloudb.mysql.nio.MySQLConnection$StatusSync@364c4b05, writeQueue=0,
modifiedSQLExecuted=false]
04-29 22:06:10.554 DEBUG [$_NIOREACTOR-0-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=12, lastTime=1430316370489, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=17186, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.589 DEBUG [$_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
`mycat_model`.`t_user` LIMIT 0
04-29 22:06:10.590 DEBUG [$_NIOREACTOR-3-RW] (EnchachePool.java:76) -SQLRouteCache miss cache
,key:mycatSELECT * FROM `mycat_model`.`t_user` LIMIT 0
04-29 22:06:10.592 DEBUG [$_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
`mycat_model`.`t_user` LIMIT 0, route={
    1 -> dn1{SELECT * FROM `mycat_model`.`t_user` LIMIT 0}
} rrs
04-29 22:06:10.592 DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost
04-29 22:06:10.592 DEBUG [$_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=4, lastTime=1430316370591, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=89064, charset=utf8, txIsolation=0, autocommit=true, attachment=dn1{SELECT * FROM
`mycat_model`.`t_user` LIMIT 0}, respHandler=SingleNodeHandler [node=dn1{SELECT * FROM
`mycat_model`.`t_user` LIMIT 0}, packetId=0], host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-29 22:06:10.603 DEBUG [$_NIOREACTOR-0-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=4, lastTime=1430316370591, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=89064, charset=utf8, txIsolation=3, autocommit=true, attachment=dn1{SELECT *
FROM `mycat_model`.`t_user` LIMIT 0}, respHandler=SingleNodeHandler [node=dn1{SELECT * FROM
`mycat_model`.`t_user` LIMIT 0}, packetId=6], host=121.40.121.133, port=3306,
statusSync=org.opencloudb.mysql.nio.MySQLConnection$StatusSync@4792ba63, writeQueue=0,
modifiedSQLExecuted=false]
04-29 22:06:10.603 DEBUG [$_NIOREACTOR-0-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=4, lastTime=1430316370591, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=89064, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=121.40.121.133, port=3306, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.603 DEBUG [$_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 COLUMNS FROM
`mycat_model`.`t_user`
04-29 22:06:10.603 DEBUG [$_NIOREACTOR-3-RW] (NonBlockingSession.java:118) -ServerConnection [id=1,
schema=mycat, host=127.0.0.1, user=mycat,txIsolation=3, autocommit=true, schema=mycat]SHOW COLUMNS FROM
mycat_model.t_user, route={
    1 -> dn2{SHOW COLUMNS FROM t_user}
} rrs
04-29 22:06:10.604 DEBUG [$_NIOREACTOR-3-RW] (PhysicalDBPool.java:417) -select read source master for
dataHost:jdbchost2
04-29 22:06:10.604 DEBUG [$_NIOREACTOR-3-RW] (MySQLConnection.java:437) -con need syn ,total syn cmd 1
commands SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;schema change:false con:MySQLConnection
[id=16, lastTime=1430316370591, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=17191, charset=utf8, txIsolation=0, autocommit=true, attachment=dn2{SHOW COLUMNS FROM t_user},
respHandler=SingleNodeHandler [node=dn2{SHOW COLUMNS FROM t_user}, packetId=0], host=116.236.223.115,
port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.665 DEBUG [$_NIOREACTOR-0-RW] (NonBlockingSession.java:246) -release connection
MySQLConnection [id=16, lastTime=1430316370591, schema=mycat_model, old shema=mycat_model, borrowed=true,
fromSlaveDB=false, threadId=17191, charset=utf8, txIsolation=3, autocommit=true, attachment=dn2{SHOW COLUMNS
FROM t_user}, respHandler=SingleNodeHandler [node=dn2{SHOW COLUMNS FROM t_user}, packetId=12],
host=116.236.223.115, port=3307, statusSync=org.opencloudb.mysql.nio.MySQLConnection$StatusSync@278806c4,
writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:10.665 DEBUG [$_NIOREACTOR-0-RW] (PhysicalDatasource.java:386) -release channel MySQLConnection
[id=16, lastTime=1430316370591, schema=mycat_model, old shema=mycat_model, borrowed=true, fromSlaveDB=false,
threadId=17191, charset=utf8, txIsolation=3, autocommit=true, attachment=null, respHandler=null,
host=116.236.223.115, port=3307, statusSync=null, writeQueue=0, modifiedSQLExecuted=false]
04-29 22:06:21.332 INFO [$_NIOConnector] (AbstractConnection.java:398) -close connection,reason:heartbeat
connecterr

```



```
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:06:21.334 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:06:42.333 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:06:42.334 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:07:03.326 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:07:03.344 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:07:24.327 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:07:24.345 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:07:45.332 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
04-29 22:07:45.355 INFO [$NIOConnector] (AbstractConnection.java:398) -close connection, reason:heartbeat
connecterr
, [thread=$_NIOConnector, class=MySQLDetector, host=116.236.223.115, port=33071, localPort=0, schema=null]
```

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被分配到了分片dn1、dn2上同时执行，如果某个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。

# 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
org.opencloud.db.route.impl.DruidMycatRouteStrategy.routeNormalSqlWithAST(DruidMycatRouteStrategy.java:44)
    at org.opencloud.db.route.impl.AbstractRouteStrategy.route(AbstractRouteStrategy.java:52)
    at org.opencloud.db.route.RouteService.route(RouteService.java:118)
    at org.opencloud.db.server.ServerConnection.routeEndExecuteSQL(ServerConnection.java:165)
    at org.opencloud.db.server.ServerConnection.execute(ServerConnection.java:154)
    at org.opencloud.db.server.ServerQueryHandler.query(ServerQueryHandler.java:125)
    at org.opencloud.db.net.FrontendConnection.query(FrontendConnection.java:250)
    at org.opencloud.db.net.handler.FrontendCommandHandler.handle(FrontendCommandHandler.java:56)
    at org.opencloud.db.net.FrontendConnection.handle(FrontendConnection.java:357)
    at org.opencloud.db.net.AbstractConnection.onReadData(AbstractConnection.java:276)
    at org.opencloud.db.net.NIOSocketWR.asyncRead(NIOSocketWR.java:186)
    at org.opencloud.db.net.AbstractConnection.asyncRead(AbstractConnection.java:238)
    at org.opencloud.db.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 :
'elect * from t_user t where t.',expect IDENTIFIER, actual IDENTIFIER elect
    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
org.opencloud.db.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。

## Mycat的配置

### schema.xml

#### 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。

#### 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。

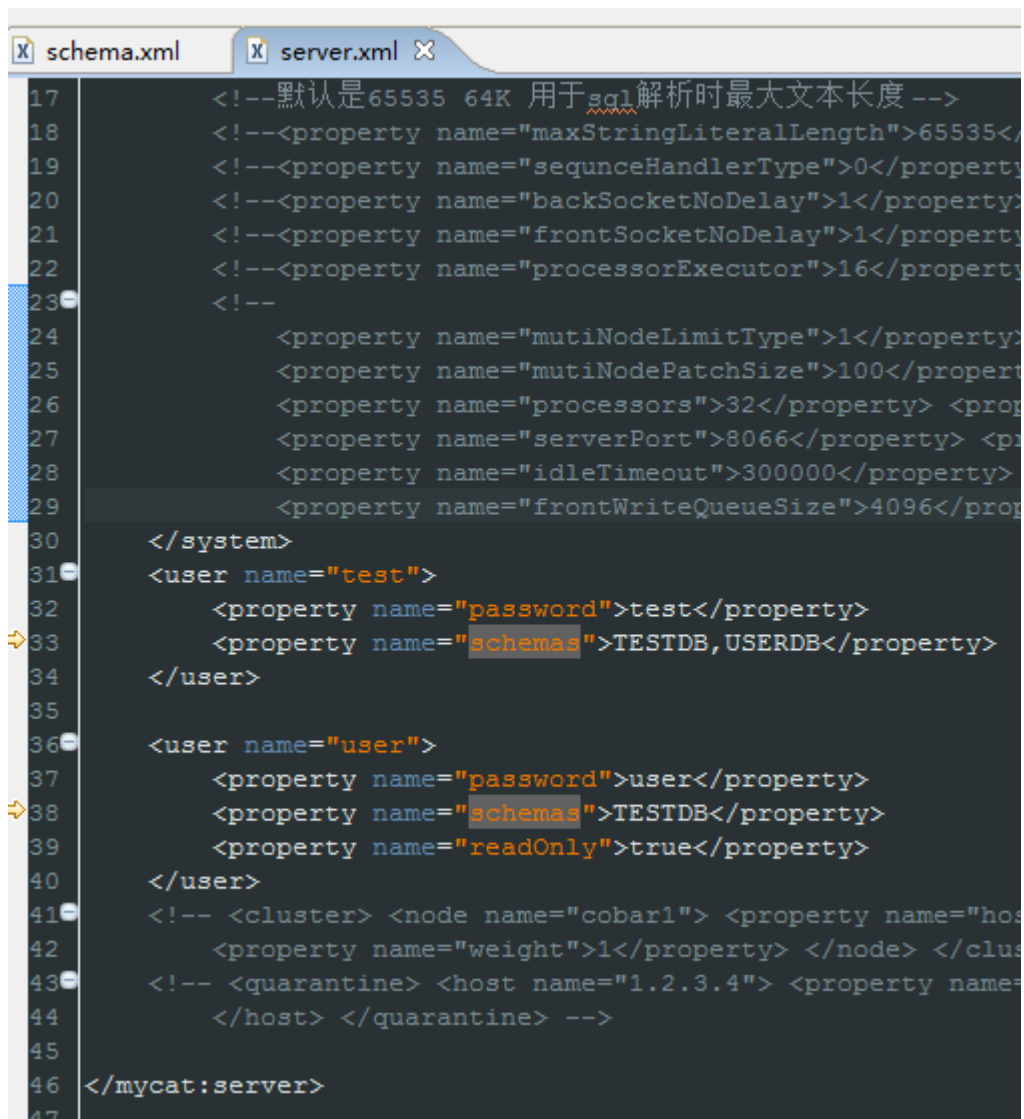
```
mysql> use USERDB;
Database changed
mysql> select * from company;
Empty set (0.09 sec)

mysql> select * from travelrecord;
ERROR 1064 (HY000): can't find table define in schema ,table:TRAVELRECORD 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="cobar1"> <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="host">127.0.0.1</property> </host> </quarantine> -->
43 <!-- <quarantine> <host name="1.2.3.4"> <property name="host">127.0.0.1</property> </host> </quarantine> -->
44 </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)

dataNode

该属性用于绑定逻辑库到某个具体的database上，如果定义了这个属性，那么这个逻辑库就不能工作在分库分表模式下了。也就是说对这个逻辑库的所有操作会直接作用到绑定的dataNode上，这个schema就可以用作读写分离和主从切换，具体如下配置:

```
<schema name="USERDB" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn1">
    <!--这里不能配置任何逻辑表信息-->
</schema>
```

那么现在USERDB就绑定到dn1所配置的具体database上，可以直接访问这个database。当然该属性只能配置绑定到一个database上，不能绑定多个dn。

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。

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。不设置该值的话，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。

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。

属性名	值	数量限制
name	String	(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。

## 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" dataHost="localhost1" database="db$0-99" ></dataNode>
<dataNode name="multipleDn2" dataHost="localhost1" database=" db$0-99" ></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。

## 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。

## 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。

## 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。

## 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globla的所有表。

## 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。

## 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。

## 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。

name	String	(1)
joinKey	String	(1)
parentKey	String	(1)
primaryKey	String	(0..1)
needAddLimit	boolean	(0..1)

## name属性

定义子表的表名。

## 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。

## 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。

## primaryKey属性

同table标签所描述的。

## needAddLimit属性

同table标签所描述的。

## 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)

### 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。

### 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name属性。

### 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，因为这里使用两个纬度来定义分片，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。

### dataHost标签

## dataHost

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)

## name属性

唯一标识dataHost标签，供上层的标签使用。

## 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。

## 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。

## balance属性

负载均衡类型，目前的取值有3种：

1. balance= “0”，所有读操作都发送到当前可用的writeHost上。
2. balance= “1”，所有读操作都随机的发送到readHost。
3. balance= “2”，所有读操作都随机的在writeHost、readhost上分发。

## writeType属性

负载均衡类型，目前的取值有3种：

1. writeType= “0”，所有写操作都发送到可用的writeHost上。
2. writeType= “1”，所有写操作都随机的发送到readHost。
3. writeType= “2”，所有写操作都随机的在writeHost、readhost上分发。

## 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。



## 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。

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。

## 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'

## 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宕机系统会自动的检测到，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)

## host属性

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

## 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或其他的dbDriver，则需要特殊指定。当使用JDBC时则可以这么写：jdbc:mysql://localhost:3306/。

## user属性

后端存储实例需要的用户名字

## password属性

后端存储实例需要的密码

## server.xml

### 优化配置

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。现在就对这个文件中的配置，一一介绍。

### user标签

```
<user name="test">
  <property name="password">test</property>
  <property name="schemas">TESTDB</property>
  <property name="readOnly">true</property>
</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>
```

### 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。

### defaultSqlParser属性

由于mycat最初是时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。

### 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。

### 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。

## 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bufferChunkSize(4096) \* processors属性 \* 1000**

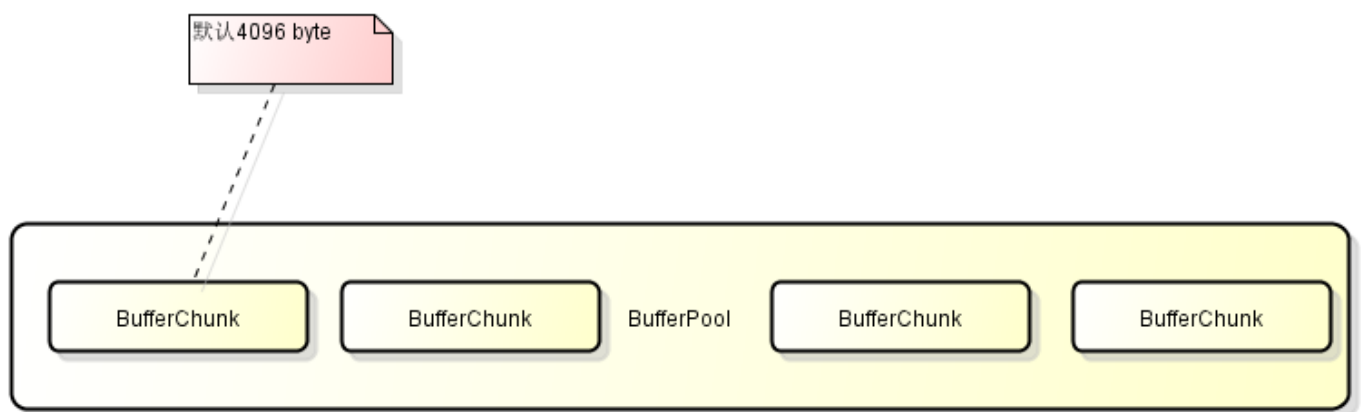
**BufferPool的总长度 = bufferPool / bufferChunk。**

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：

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

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



## 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ThreadLocalPool的长度公式如下：

$$\text{ThreadLocalPool的长度} = \text{线程缓存百分比} * \text{BufferPool长度} / 100$$

假设BufferPool的长度为 4000，其他保持默认值。

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

## 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在需要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。

## 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。默认是使用本地文件方式，文件方式主要只是用于测试使用。

## 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。

## 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。

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。

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。

## 周期间隔相关属性

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。

dataNodeIdleCheckPeriod: 对后端连接进行空闲、超时检查的时间间隔，默认是60秒。

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

## 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。

## rule.xml

### 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。

### 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。连接表规则和具体路由算法。当然，多个表规则可以连接到同一个路由算法上。

标签内使用。让逻辑表使用这个规则进行分片。

### function标签

```
<function name="hash-int"
  class="org.opencloudb.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。

# Mycat的join

## 分片join

### join概述

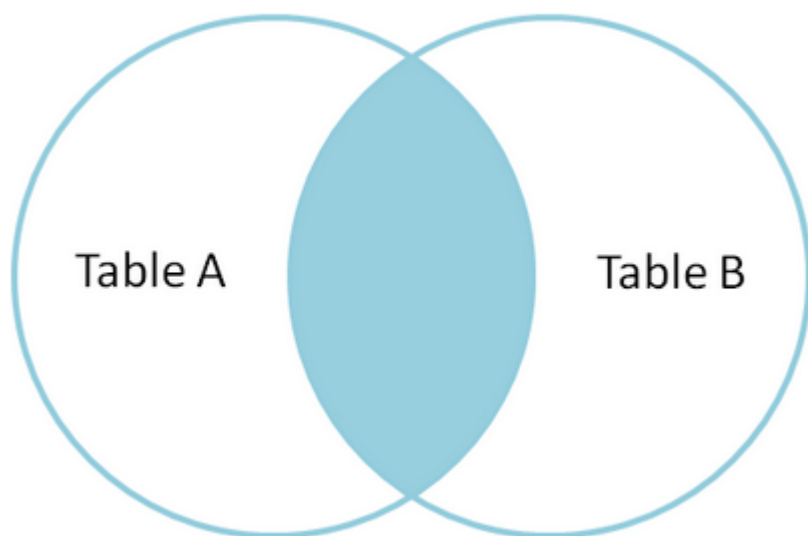
Join绝对是关系型数据库中最常用一个特性，然而在分布式环境中,跨分片的join确是最复杂的，最难解决一个问题。

下面我们简单介绍下各种Join操作。

#### 1：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。

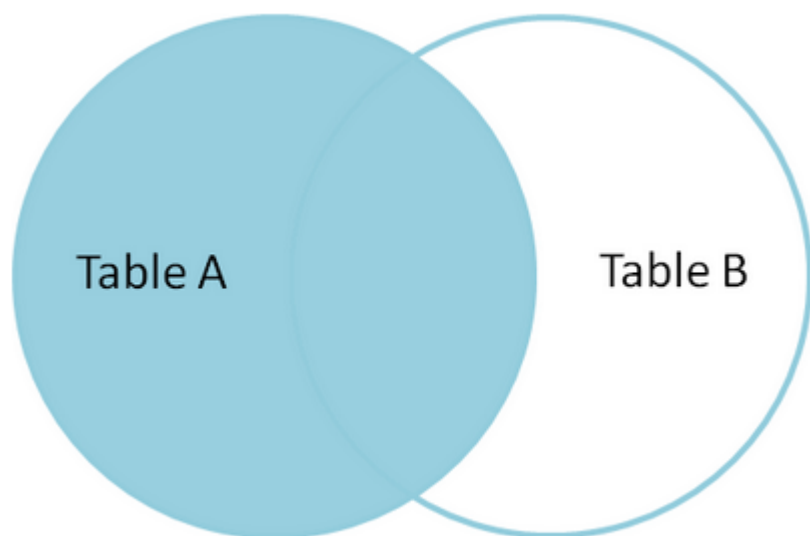
如图：



#### 2: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：



#### 3：RIGHT JOIN

同Left join,AB表互换即可。

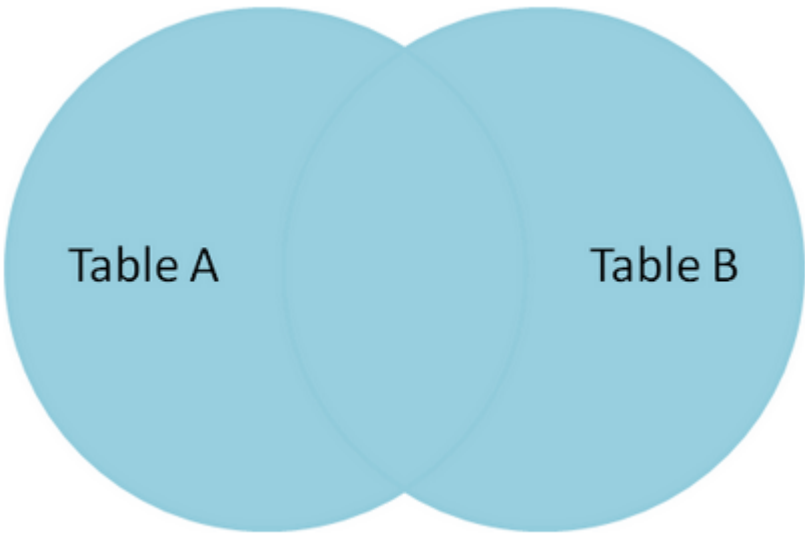
#### 4：Cross join

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。

5 : 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。



6 : 性能建议

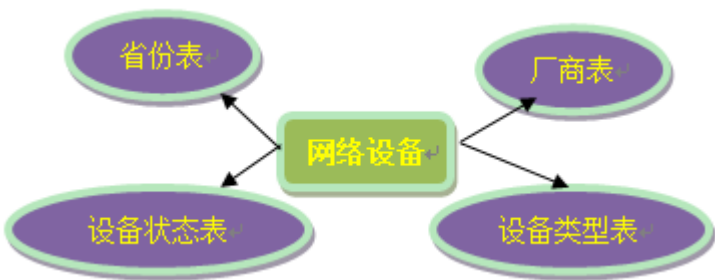
- a:尽量避免使用Left join或Right join,而用Inner join
- b:在使用Left join或Right join时，ON会优先执行，where条件在最后执行，所以在使用过程中，条件尽可能的在ON语句中判断，减少where的执行
- c:少用子查询，而用join。

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。

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”，而不理解通常为的“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：

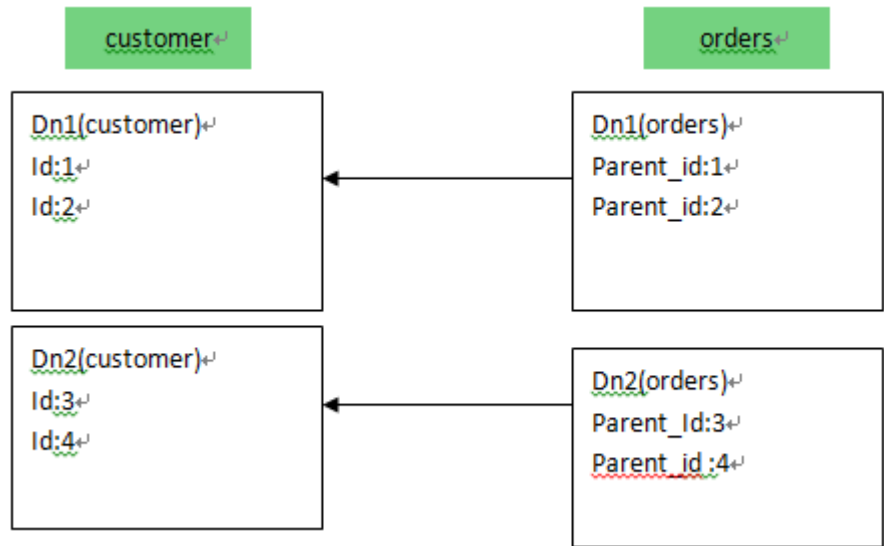
```
<table name="company" primaryKey="ID" type="global" dataNode="dn1, dn2, dn3" />
```

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。

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 and 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>
```

## 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 ;

```

## 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表上查询对应的记录，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();}});

```

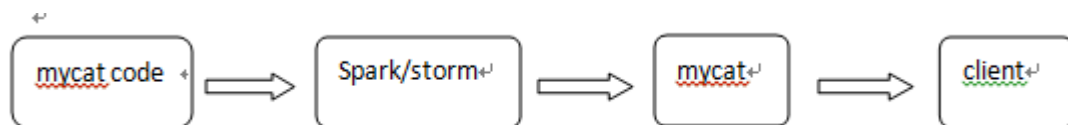
以上提供一个SQL执行框架，完全是异步的模式执行，并且以后会提供更多高质量的API，简化分布式数据处理，比如内存结合文件的数据JOIN算法，分组算法，排序算法等等，  
期待更多的牛人一起来完善。

## 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。



## 第五章 全局序列号

### 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：

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。

## 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；

2).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：

1

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;

```

3) 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' );
```

## 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

.....

## 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。

## 自增长主键

### 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)
```

Mybatis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>
```

## Mycat 分片规则

### 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。

#### a. 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" />
```

#### b. 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。

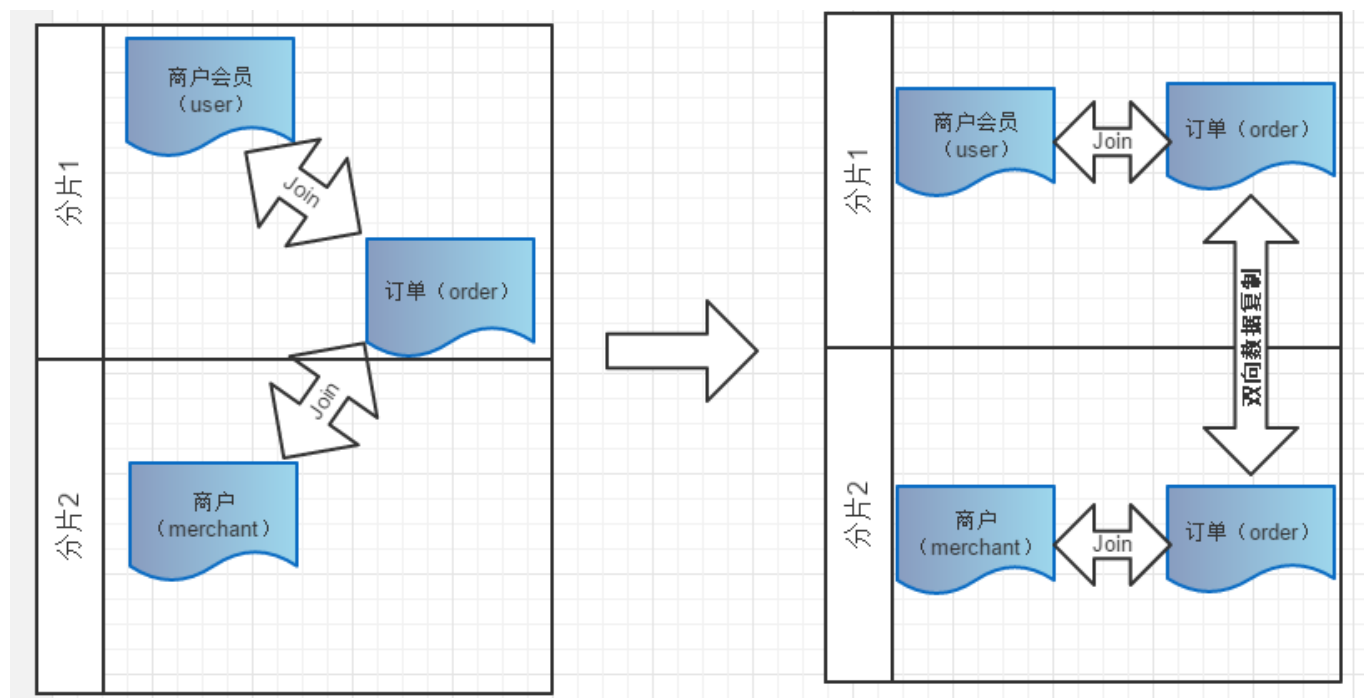
```
<table name="order" dataNode="dn$1-32" rule="mod-long">
    <childTable name="order_detail" primaryKey="id" joinKey="order_id" parentKey="order_id" />
</table>
```

</table>

### c. 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，但是如果要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：



### d. 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列号作为主键时，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：

1. 尽可能的比较均匀分布数据到各个节点上；
2. 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会把你根据主键查询的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。

## Mycat常用的分片规则

### 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="org.opencloudb.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
 */
```

### 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="org.opencloudb.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。

$1024 = \sum((count[i]*length[i]))$ . 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，partitionCount\*partitionLength=1024

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

## 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，

start <= range <= 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="org.opencloudb.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

0-10000000=0

10000001-20000000=1

## 求模

此规则为对分片字段求摸运算。

```
<tableRule name="mod-long">
  <rule>
    <columns>user_id</columns>
    <algorithm>mod-long</algorithm>
  </rule>
</tableRule>
<function name="mod-long" class="org.opencldb.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。

## 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="org.opencldb.route.function.PartitionByDate">
  <property name="dateFormat">yyyy-MM-dd</property>
  <property name="sBeginDate">2014-01-01</property>
  <property name="sPartionDay">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

```
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" ));
```

## 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="org.opencloud.db.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));
```

## ASCII码求模范围约束

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="org.opencloudb.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sum%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));
```

## 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="org.opencloudb.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size=2位数字即05，05就是获取的分区，如果没传默认分配到defaultPartition

## 字符串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="org.opencloudb.route.function.PartitionByString">
  <property name=length>512</property> <!-- zero-based -->
  <property name="count">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length代表字符串hash求模基数，count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 = "aaaabbb0000";
Assert.assertEquals(true, 0 == rule.calculate(idVal));
idVal = "aaaabbb2359";
Assert.assertEquals(true, 0 == rule.calculate(idVal));
```

## 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="org.opencloudb.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>
```

## 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="org.opencloud.db.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 = partion.calculateRange("2015020100", "2015020123");
assertTrue(span.length == 24);
assertTrue(span[0] == 0 && span[span.length - 1] == 23);
```

## 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="org.opencloud.db.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"));
```

## 权限控制

### 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：

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。

### 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：

mycat

## 常见问题与解决方案

### 常见问题与解答

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

2. Mycat出来Mysql还支持哪些数据库？

答：mongodb、oracle、sqlserver、hive、db2、postgresql。

3. Mycat目前有生产案例了么？

答：目前Mycat初步统计大概60家公司使用。

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。

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。

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。

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、以石头狮子为首的产品发布及代码管理，还有以Marshy为首的推广。

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）

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1.3.0.3，1.4为开发板，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>
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>
```

1. 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。
2. 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。
3. 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。  
启动方式，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。

1. 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。
1. 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。

1. 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。

1. 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)。

1. 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。

1. Mycat 启动包主机不存在的问题？

答：需要添加ip跟主机的映射。

1. Mycat连接会报无效数据源（Invalid datasource）？

答：如果不是配置问题，分析具体日志看出错原因，常见的有：

2. 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。

3. 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，或用用户名密码是否正确。

1. 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>

1. 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);
```

1. Mycat目前有多少人维护？

打：目前初步统计有10人以上核心人员维护。

2. 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。

## Mycat性能测试

### 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，因为它的数据库是分布在几个Datahost上的，因此插入和查询，都需要特定的工具，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），}${char([a-f,0-9]8:8)}表示从a到f以及0到9的字符串随机组成，定长为8位。
Enum:枚举，表示从指定范围内获取一个值，${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

### 读写分离

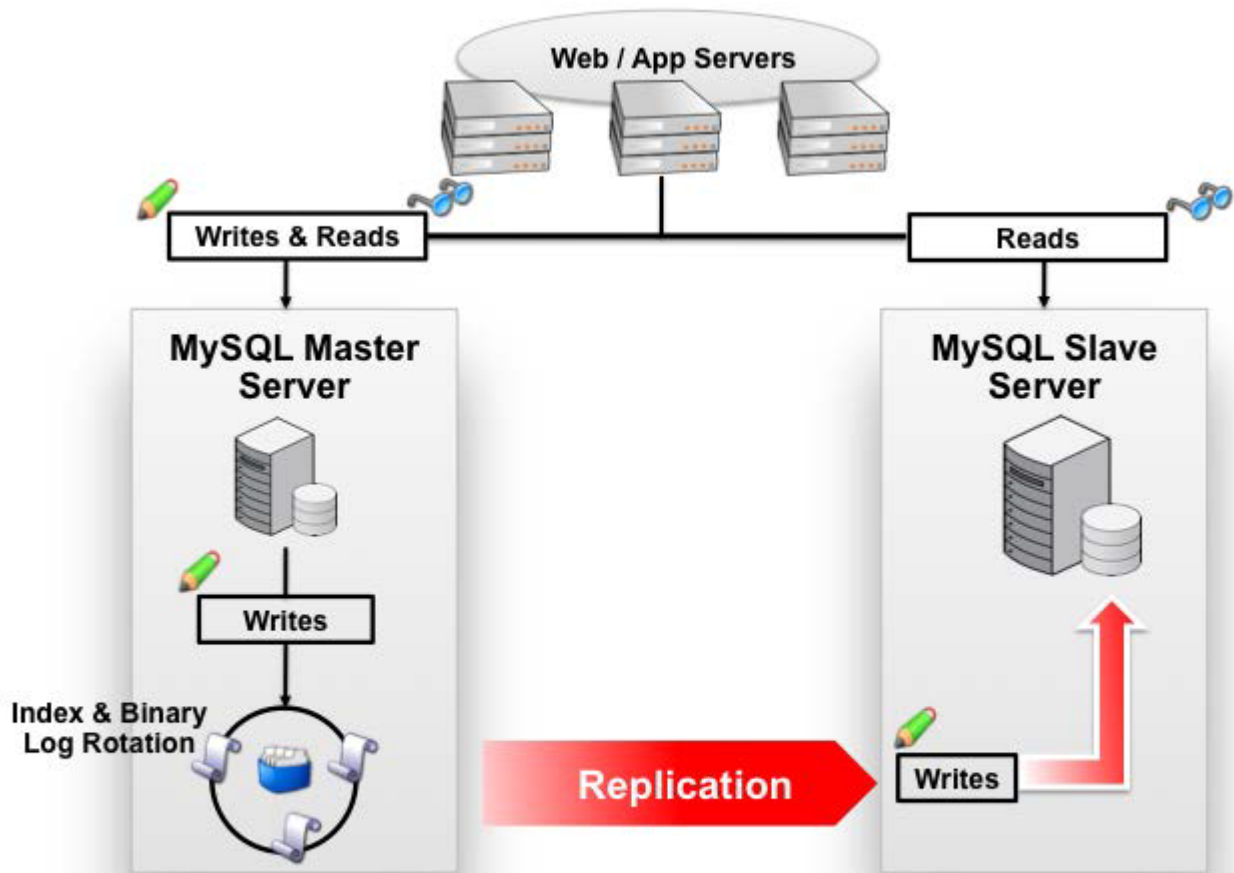
#### MySQL主从复制的几种方案

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。

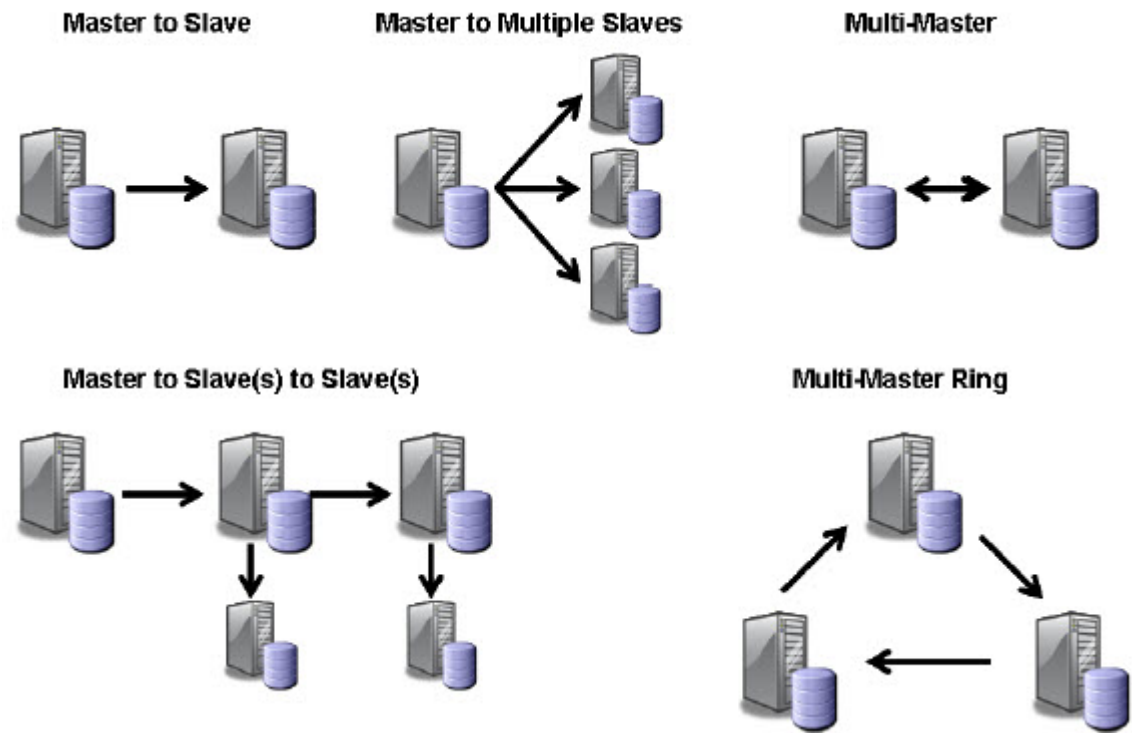
从数据库的角度来说，对于大多数应用来说，从集中到分布，最基本的一个需求不是数据存储的瓶颈，而是在于计算的瓶颈，即SQL查询的瓶颈，我们知道，正常情况下，Insert SQL就是几十个毫秒的时间内写入完成，而系统中的大多数Select SQL则要几秒到几分钟才能有结果，很多复杂的SQL，其消耗服务器CPU的能力超强，不亚于死循环的威力。在没有读写分离的系统上，很

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。

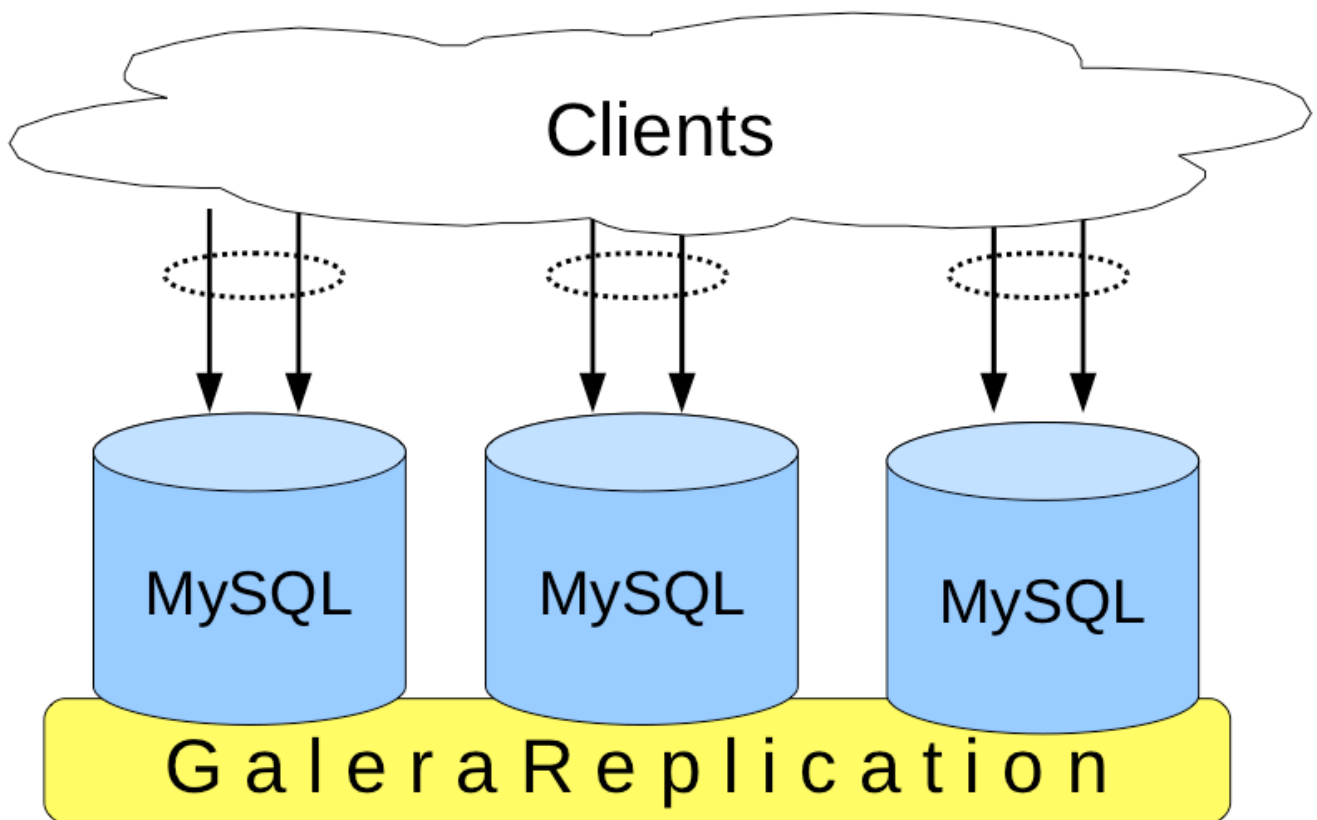
备库的SQL线程执行最后一步，该线程从中继日志中读取事件并在备库执行，从而实现备库数据的更新。当SQL线程追赶上I/O线程时，中继日志通常已经在系统缓存中，所以中继日志的开销很低。SQL线程执行的事件也可以通过配置选项来决定是否写入其自己的二进制日志中，它对于我们稍后提到的场景非常有用。这种复制架构实现了获取事件和重放事件的解耦，允许这两个过程异步进行。也就是说I/O线程能够独立于SQL线程之外工作。但这种架构也限制了复制的过程，其中最重要的一点是在主库上并发运行的查询在备库只能串行化执行，因为只有一个SQL线程来重放中继日志中的事件。后面我们将会看到，这是很多工作负载的性能瓶颈所在。虽然有一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。

\*\*



## A c t i v e   c l u s t e r

### 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研究，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。

## Mycat支持的读写分离

当MySQL按照之前的主从复制方式配置好集群以后，可以开启Mycat的读写分离机制，以以下的配置为例，表明一个从节点hostS1与一个主节点hostM1组成了标准的一主一从的读写分离模式，参数balance决定了哪些MySQL服务器参与到读SQL的负载均衡中，0为不开启读写分离，

1为全部的readHost与standby writeHost参与select语句的负载均衡，比如我们配置了1主3从的MySQL主从环境，并把第一个从节点MySQL配置为dataHost中的第二个writeHost，以便主节点宕机后，Mycat自动切换到这个writeHost上来执行写操作，此时balance=1就意味着第一个writeHost不参与读SQL的负载均衡，其他3个都参与；balance=2则表示所有的writeHost不参与，此时，只有2个readHost参与负载均衡。这里有一个细节需要你知道，readHost是从属于writeHost的，即意味着它从那个writeHost获取同步数据，因此，当它所属的writeHost宕机了，则它也不会再参与到读写分离中来，即“不工作了”，这是因为此时，它的数据已经“不可靠”了。基于这个考虑，目前mycat 1.3和1.4版本中，若想支持MySQL一主一从的标准配置，并且在主节点宕机的情况下，从节点还能读取数据，则需要在Mycat里配置为两个writeHost并设置balance=1。

```
<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>

```

writeType=1仅仅对于galera for mysql集群这种多主多节点都能写入的集群起效，此时Mycat会随机选择一个writeHost并写入数据，对于非galera for mysql集群，请不要配置writeType=1，会导致数据库不一致的严重问题。

Mycat目前支持自动方式、编程指定的两种读写分离方式：

自动方式，即一个查询SQL是自动提交模式，对应于connection.setAutocommit(true) 或者 set autocommit=1

编程指定方式，即一个查询SQL语句以/\*balance\*/注解来确定其是走读节点还是写节点。在1.3版本里，若事务内的的查询语句增加此注解，则强制其走读节点，而1.4版本里继续强化，可以在非事务内的查询语句前增加此注解，强制走写节点，这个增强是为了避免主从不同步的情况下要求查询到刚写入的数据而做的增强。

另外 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。

switchType 目前有三种选择：

- -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
- 下面为参考配置：

```

<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">
<!-- can have multi read hosts -->

</writeHost>
<writeHost host="hostS1" url="localhost:3316" user="root"
password="123456" />
</dataHost>

```

《/ddataHost>

conf/log4j.xml中配置日志输出级别为debug时，当选择节点的时候，会输出如下日志：

```

16:37:21.660 DEBUG [Processor0-E3] (PhysicalDBPool.java:333) -select read source hostM1 for
dataHost:localhost1
16:37:21.662 DEBUG [Processor0-E3] (PhysicalDBPool.java:333) -select read source hostM1 for
dataHost:localhost1

```

根据这个信息，可以确定某个SQL发往了哪个读（写）节点，据此可以分析判断是否发生了读写分离。

用MySQL客户端连接到Mycat的9066管理端口，执行show @@datanode，也能看出负载均衡的情况，其中execute字段表明该分片上执行过的SQL累计数：

![输入图片说明](http://static.oschina.net/uploads/img/201504/07212301\_4ZPx.jpg "在这里输入图片标题")

至于应用中的哪些数据查询比较适合开启读写分离，总结下来大概有以下几种：

- 列表界面，通常是浏览查询功能，这类的数据访问频繁但实时性要求比较低，有几秒几十秒的延迟，通常感觉不出来，淘宝界面里，已售出的商品个数往往比商家后台看到的数据要延迟很大，也说明了它是一个快照数据
- 某个数据的详细信息页面，通常也访问较为频繁，但事实性要求不高
- 历史时刻的数据，比如昨天的数据，上个月的，这种数据即使有修改，也概率很低

Mycat的读写分离，默认是按照该SQL是否有事务包裹，由于一些高层框架如Hibernate、Spring等往往会自动追加事务控制语句，将查询语句变成事务内的语句，当你开启Mycat Debug日志级别后，就可能很清楚的看到这一点，日志中会出现如下的序列，此时不会走读写分离，因此建议程序设计的时候，手工控制事务，让这些查询语句自动提交，这个做法也有利于加快MySQL的执行过程

```
set autocommit=0
```

```
...
```

```
select *
```

commit

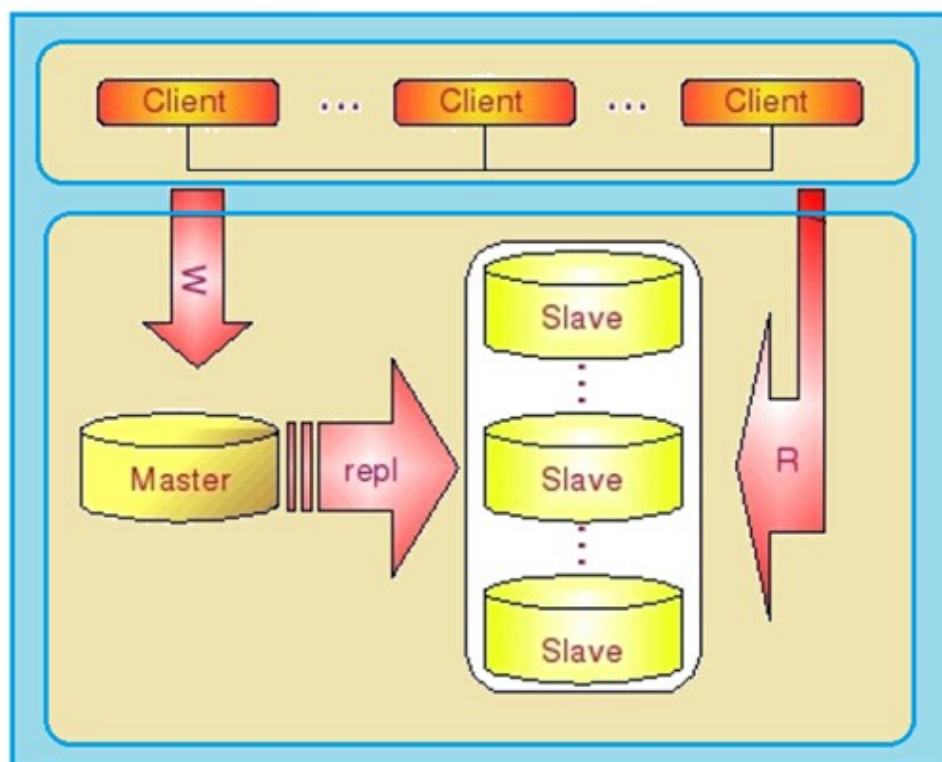
...

## 高可用与集群

### MySQL高可用的几种方案

首先我们看看MySQL高可用的几种方案：

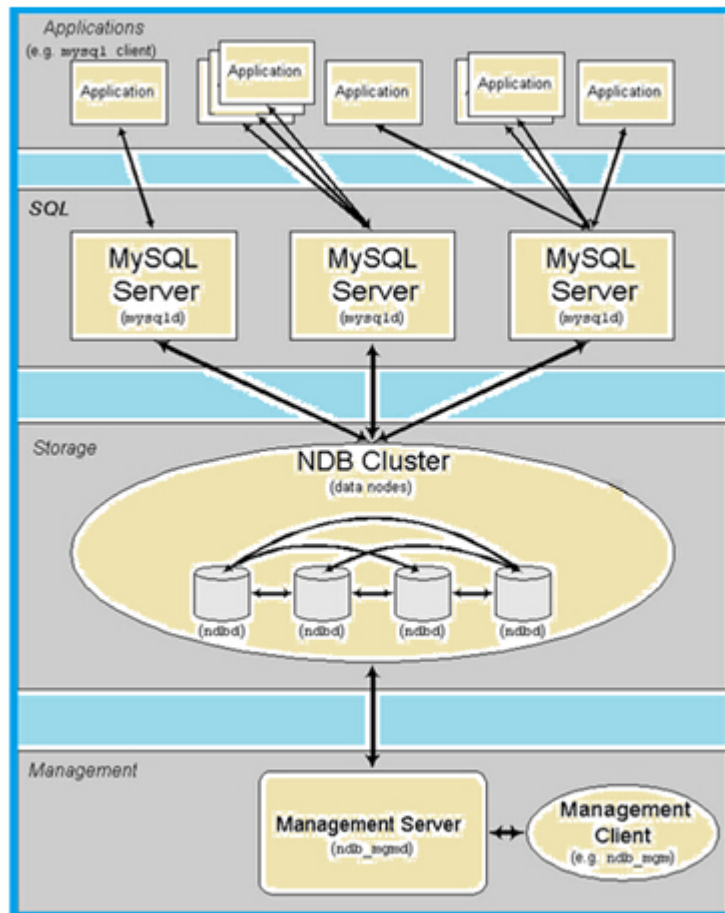
#### 主从复制 + 读写分离



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

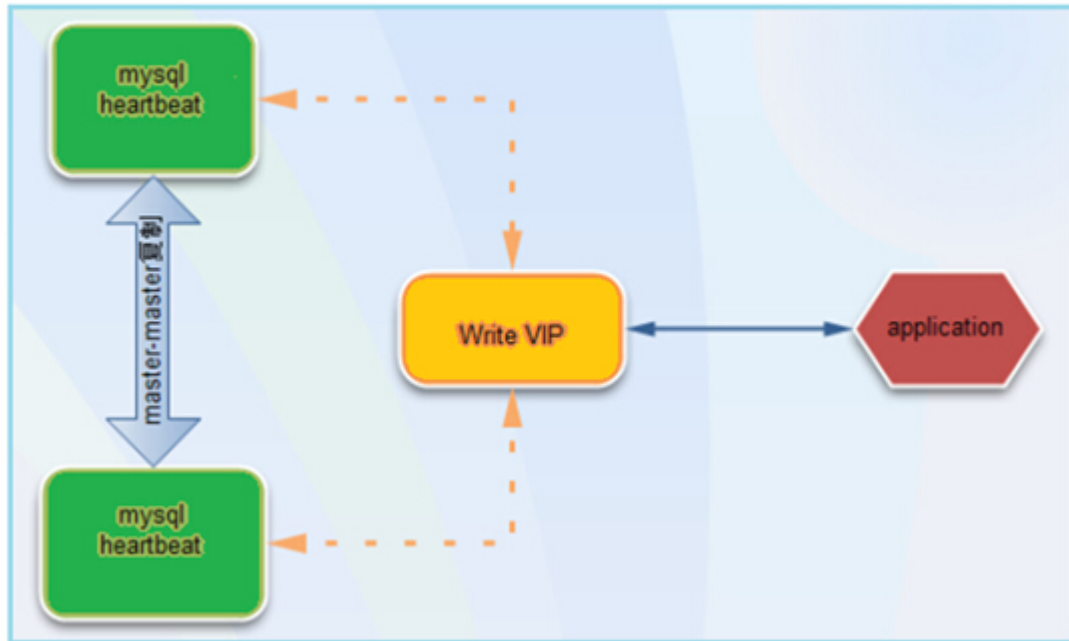


MySQL Cluster由一组计算机构成，每台计算机上均运行着多种进程，包括MySQL服务器，NDB Cluster 的数据节点，管理服务，以及（可能）专门的数据访问程序。NDB” 是一种“内存中”的存储引擎，它具有可用性高和数据一致性好的特点。

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 运营商机版本 (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。

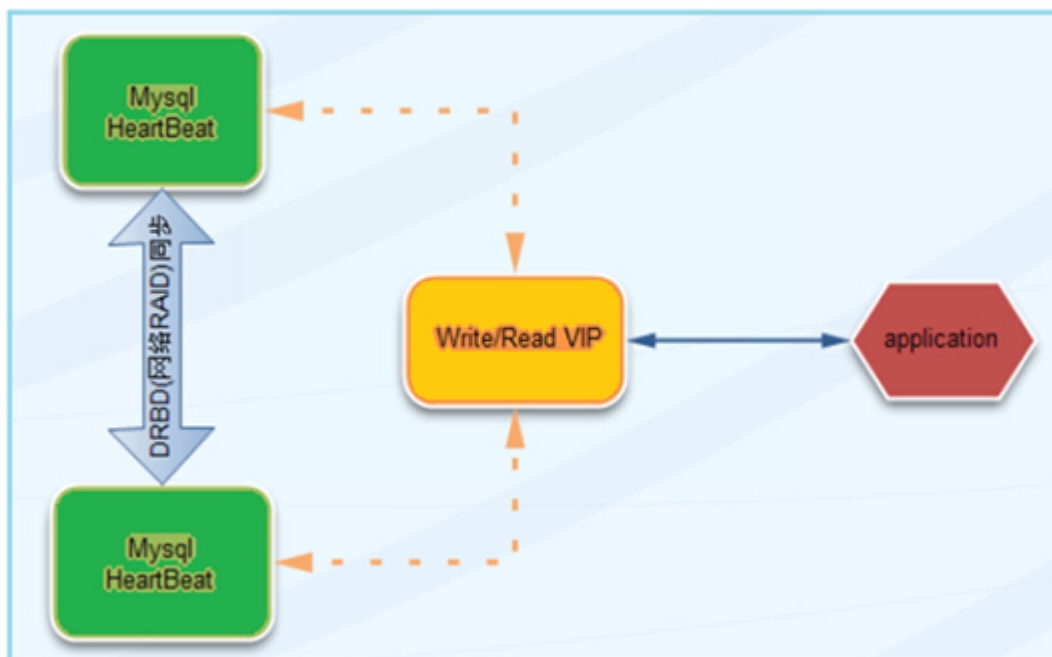


## HeartBeat+双主复制



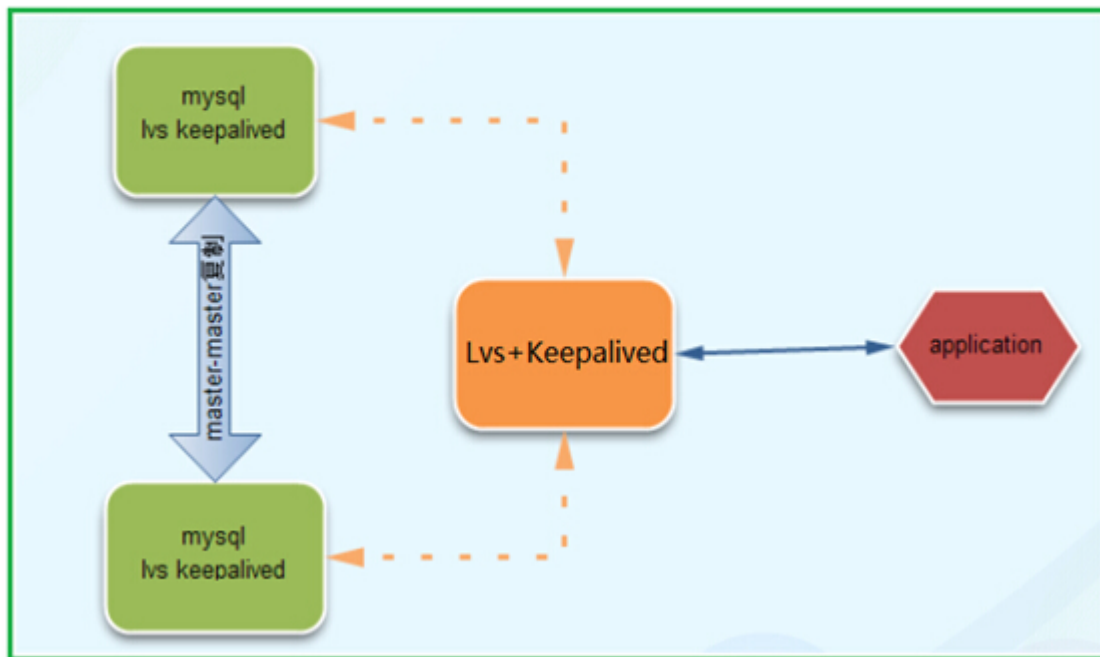
heartbeat是Linux-HA工程的一个组件,heartbeat最核心的包括两个部分：心跳监测和资源接管。在指定的时间内未收到对方发送的报文，那么就认为对方失效，这时需启动资源接管模块来接管运行在对方主机上的资源或者服务。

## HeartBeat+DRBD+MySQL



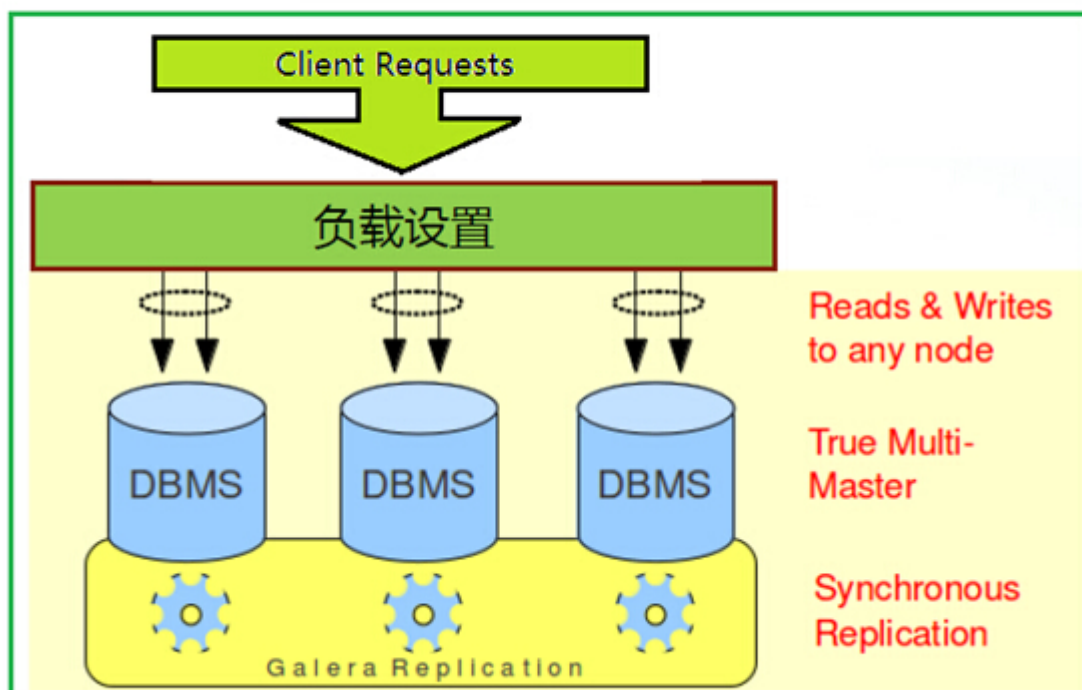
DRBD是通过网络来实现块设备的数据镜像同步的一款开源Cluster软件，它自动完成网络中两个不同服务器上的磁盘同步，相对于binlog日志同步，它是更底层的磁盘同步，理论上DRBD适合很多文件型系统的高可用。

## 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。

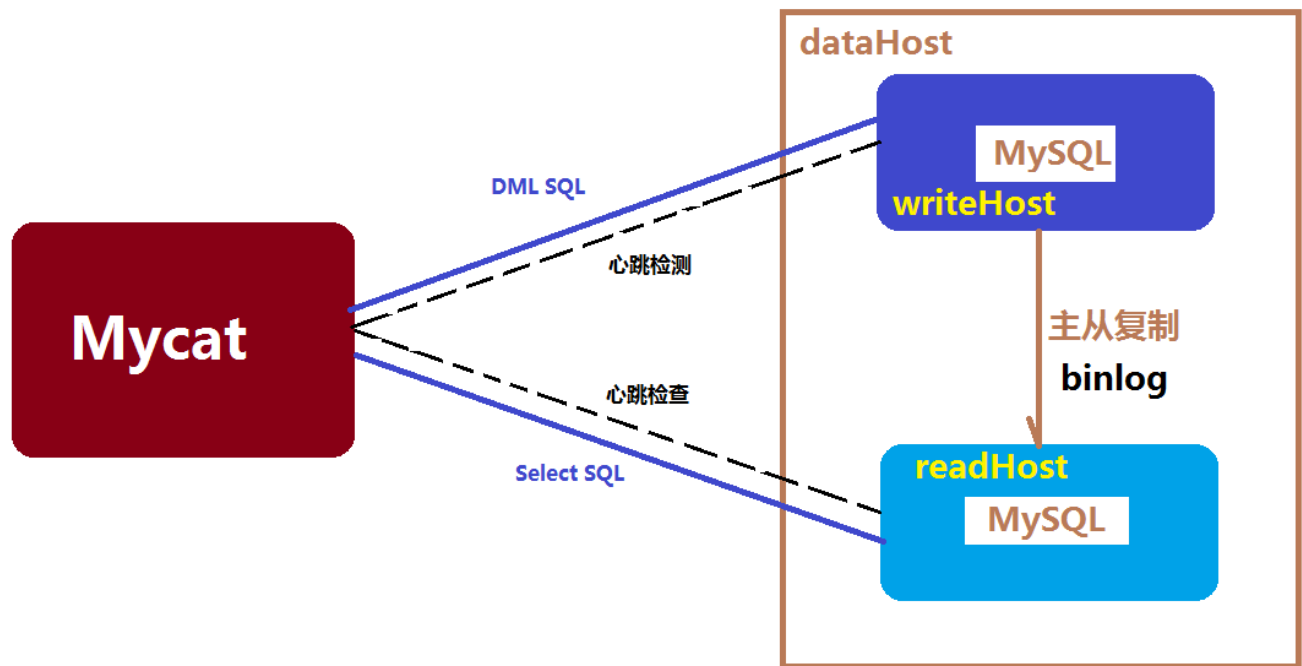
## MariaDB Galera



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。

## Mycat高可用方案

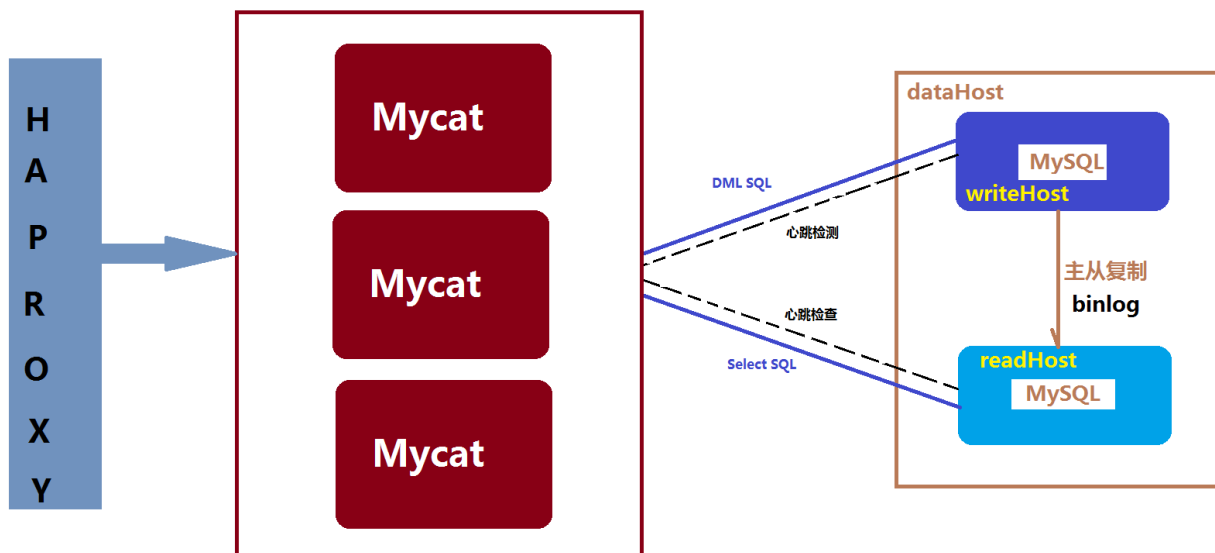
Mycat作为一个代理层中间件，Mycat系统的高可用涉及到Mycat本身的高可用以及后端MySQL的高可用，前面章节所讲的MySQL高可用方案都可以在此用来确保Mycat所连接的后端MySQL服务的高可用性。在大多数情况下，建议采用标准的MySQL主从复制高可用性配置并交付给Mycat来完成后端MySQL节点的主从自动切换。



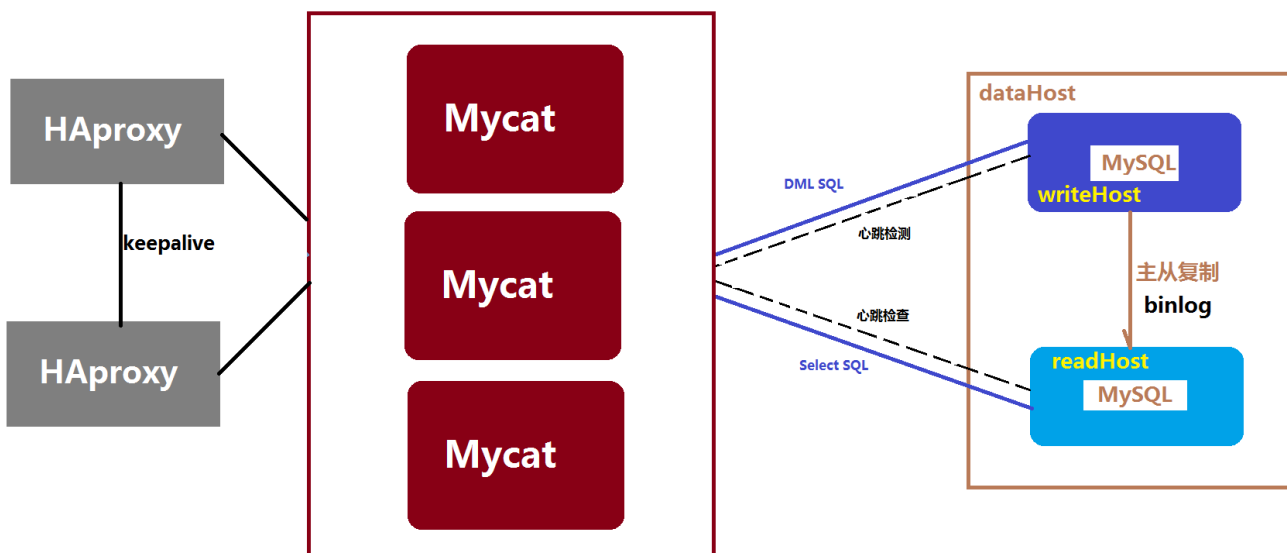
如图所示，MySQL节点开启主从复制的配置方案，并将主节点配置为Mycat的dataHost里的writeNode，从节点配置为readNode，同时Mycat内部定期对一个dataHost里的所有writeHost与readHost节点发起心跳检测，正常情况下，Mycat会将第一个writeHost作为写节点，所有的DML SQL会发送给此节点，若Mycat开启了读写分离，则查询节点会根据读写分离的策略发往readHost(+writeHost)执行，当一个dataHost里面配置了两个或多个writeHost的情况下，如果第一个writeHost宕机，则Mycat会在默认的3次心跳检查失败后，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Mycat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：



最后，Mycat还有一个项目，HA-DataSource，这是JDBC连接池，替代HAProxy，Java应用可以考虑此方案，这样节省了HAProxy的中间转发过程，并且可以定向某些数据表到某个MyCAT进行负载均衡。

<https://github.com/MyCATApache/MyCAT-Tools/tree/master/HA-DataSource>

## 事务支持

### Mycat里的数据库事务

Mycat里的事务包括以下几种情况：

单SQL不垮分片：事务中的单条SQL在单个节点上执行

单SQL跨分片：事务中的单条SQL在多个节点上执行

事务内多个SQL，在不同的分片上执行

其中，第一种情况，单一SQL仅仅在一个dataNode上执行，此时Mycat事务模式跟标准的数据库事务模式一样，要么提交要么回滚；而对于第二种和第三种的事务，Mycat执行的一种“弱XA事务”模式，此模式的逻辑如下：

首先事务内的SQL在各自的分片上执行并返回状态码，若某个分片上的返回码为ERROR，则Mycat认为事务失败，应用端只能回滚（rollback）事务，Mycat收到回滚指令后，依次回滚事务中涉及到的所有分片；若事务中的所有SQL的执行都返回成功（OK）的返回码，则应用程序提交事务的时候，Mycat会同时向事务中涉及到的节点发送提交事务的指令。

举例如下：

客户端执行如下的指令：

```
set autocommit=0
```

```
update person set name= 'xxxx' where age >18
```

```
commit
```

如果person表跨分片(dn1,dn2,dn3)，则上述SQL将触发如下的执行逻辑

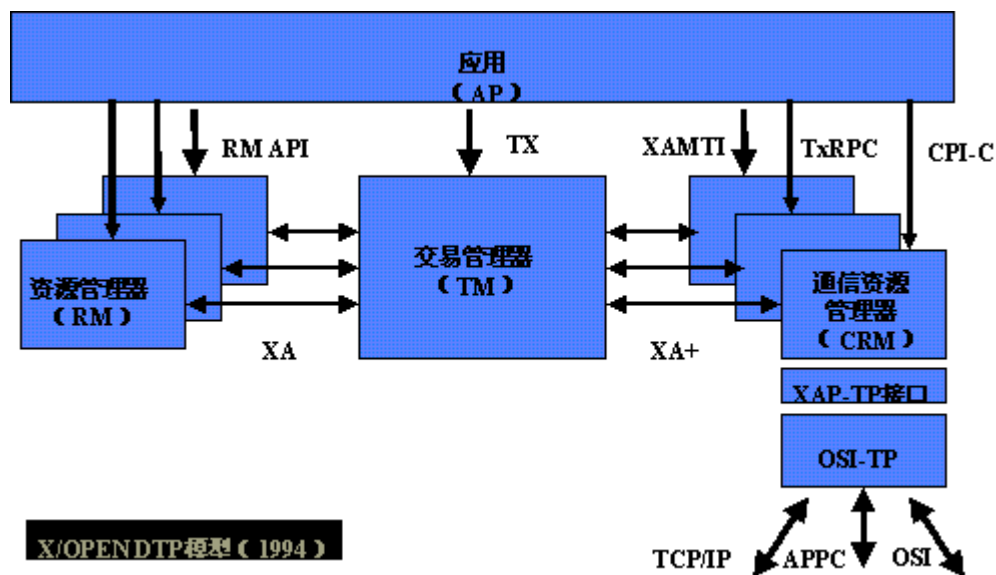
```
for ( dn1, dn2, dn3)
{
    set autocommit=0;
    update person set name='xxxx' where age >18;
}
if(allOK)
{
    for (dn1, dn2, dn3)
    {
        commit;
    }
}
```

这里称之为弱XA，是因为第二阶段Commit的时候，若某个节点出错了，也无法等节点恢复以后去做Recover操作，重新commit，但考虑到所有的节点都执行成功，但Commit指令失败的概率很小，因此这种弱XA事务也已经满足大多数应用的需求，而且性能接近普通事务。

Mycat 1.3目前还不支持MySQL的 Begin Transaction指令（后继会支持），而只支持set autocommit=0 & commit这种指令，对于JDBC程序来说，没有任何影响，其他语言的MySQL驱动应该也可以避免使用 Begin Transaction指令。

## XA事务原理

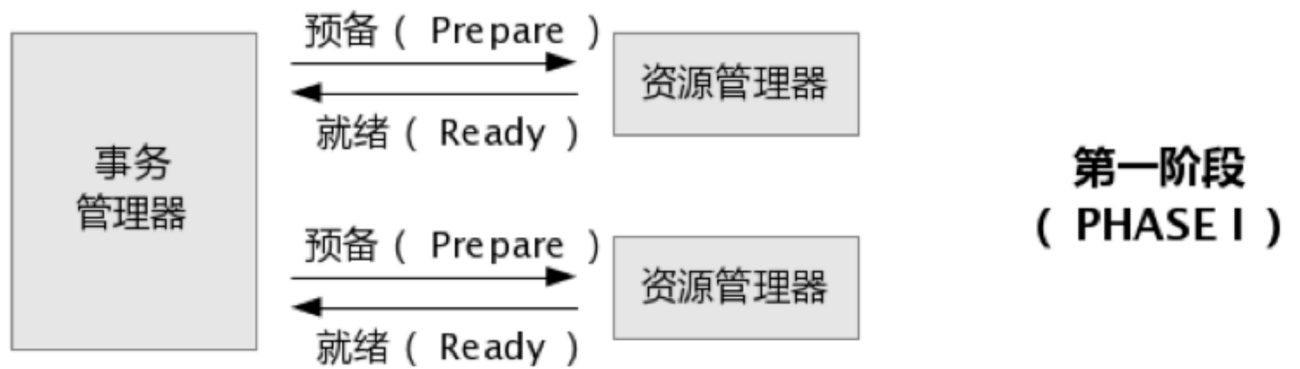
分布式事务处理（Distributed Transaction Processing，DTP）指一个程序或程序段，在一个或多个资源如数据库或文件上为完成某些功能的执行过程的集合，分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）。X/Open组织（即现在的Open Group）定义了分布式事务处理模型。X/Open DTP模型（1994）包括应用程序（AP）、事务管理器（TM）、资源管理器（RM）、通信资源管理器（CRM）四部分。一般，常见的事务管理器（TM）是交易中间件，常见的资源管理器（RM）是数据库，常见的通信资源管理器（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。

## 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.

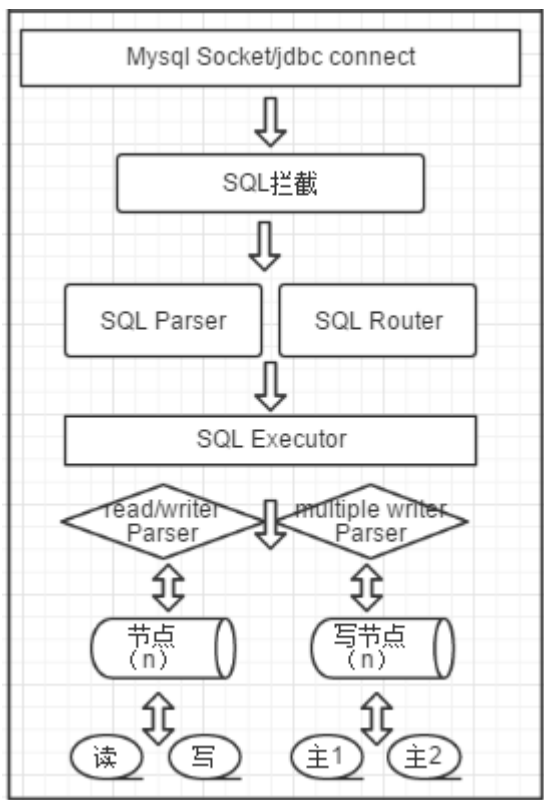
## SQL拦截

### 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">org.opencloudb.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">org.opencldb.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">org.opencldb.interceptor.impl.自定义class</property>
    <!--其他配置-->
</system>

```

## Mycat注解

### 注解的原理

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代码，这样Sql就能正确执行，这段代码称之为“注解”。注解的形式是

```

/#!mycat: sql=Sql语句*/

```

使用时将=号后的“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	
	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	
Delete	1. 对于分片表使用要删除记录的表做注解SQL	

原始Sql	注解Sql	备注
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。

## 注解使用示例

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，默认查询会到都节点，获取数据，但是有些场景需要实时获取，如果读读节点，有可能会有延时，Mycat支持通过注解/\*balance\*/来获取读写：

- 事务内的SQL，默认走写节点，以注释/\*balance\*/开头，则会根据balance=“1”或“2”去获取
- 非事务内的SQL，开启读写分离默认根据balance=“1”或“2”去获取，以注释/\*balance\*/开头则会走写 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

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. 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。

### 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。

```
/*!mycat : schema = test_01 */ sql ;
```

## Mycat Catlet

### 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。

## jdbc多数据库支持

### 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。

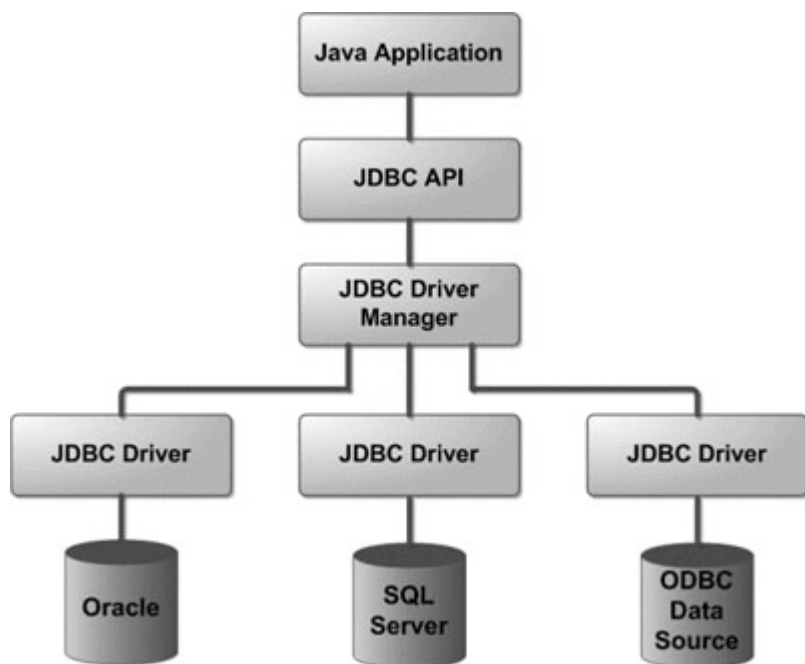
## JDBC 体系结构

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：



## 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。

## JDBC 4.0

自从核心Java语言的第一个公开发布版本起，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。

在JDBC API4.0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SQL XML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 forXML（在JSR 173规范中确立）基于Iterator模式，它与基于Observer模式的Simple API for XMLProcessing(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，还要注意到，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

## 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语句效率提高20%-30%。

## 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。

### 一、 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

### 二、 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

### 三、 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

### 四、 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

## 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。

### 1.1 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" />
```

#### 1.1.1 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。

### 1.2 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

### 1.3 支持的SQL语法

#### 1.3.1 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。

### 1.3.2 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)
```

### 1.3.3 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)
```

### 1.3.4 Select 查询语句

#### 1.3.4.1 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)
```

#### 1.3.4.2 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)
```

### 1.3.4.3 where条件

#### 1.3.4.3.1 支持等于：

select name,age from people where name= 'mongo' ;

#### 1.3.4.3.2 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)
```

#### 1.3.4.3.3 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)
```

#### 1.3.4.3.4 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)
```

#### 1.3.4.3.5 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)
```

#### 1.3.4.3.6 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)
```

#### 1.3.4.3.7 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)
```

#### 1.3.4.3.8 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)
```

#### 1.3.4.3.9 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)
```

#### 1.3.4.4 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)
```

1.3.4.5 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)
```

### 1.3.5 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)
```

### 1.3.6 Drop语句

drop table people;

删除表

## Oracle

## 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" />
```

## 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;
```

## rownum控制最大条数

以下语法控制查询结果最多5条

```
SELECT * FROM (SELECT * FROM test t) XX WHERE ROWNUM <= 5;
```

## SQL Server

### 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" />

```

## 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

```

## 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;

```

## 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
```

## 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。

## 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
```

## 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;
```

## 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）。

## 配置Mycat

修改conf下的配置schema.xml文件中的以下内容：

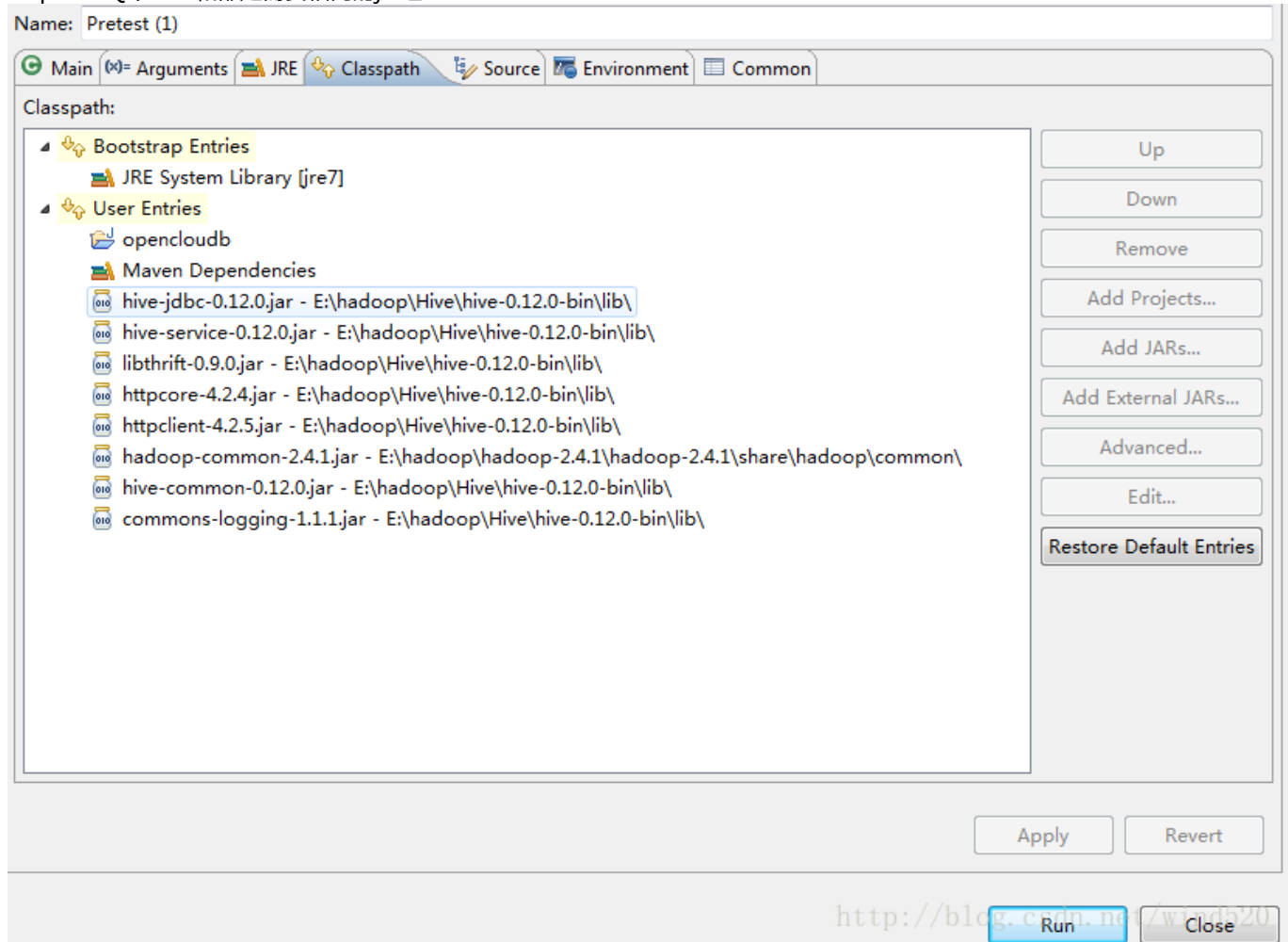
配置dataHost

在节点下在新增一个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



## 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
```

## 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
```

端口:10000 服务器:feng02

spark master:spark://feng02:7077 driver-class-path:mysql驱动包(hive配置的)

## PostgreSQL

支持limit offset分页语法以及limit控制最大条数的语法。

支持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

## 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 [base1]> 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>
```

## 管理命令与监控

### 命令行监控

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 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 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 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 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
21:34:33	false							2015-04-08

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中配置的数据Host 中name。

index：schema中配置的数据Host 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model, 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_model, old shema=mycat_model,
```

```
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 [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=26, lastTime=1428823025902, schema=mycat_nodel, old shema=mycat_nodel,
borrowed=false, fromSlaveDB=false, threadId=7061, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-12 15:21:06.620 INFO [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=30, lastTime=1428823025902, schema=mycat_nodel, old shema=mycat_nodel,
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 [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=31, lastTime=1428823025902, schema=mycat_nodel, old shema=mycat_nodel,
borrowed=false, fromSlaveDB=false, threadId=7066, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-12 15:21:06.621 INFO [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=27, lastTime=1428823025923, schema=mycat_nodel, old shema=mycat_nodel,
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 [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=33, lastTime=1428823025923, schema=mycat_nodel, old shema=mycat_nodel,
borrowed=false, fromSlaveDB=false, threadId=7069, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-12 15:21:06.622 INFO [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=25, lastTime=1428823025902, schema=mycat_nodel, old shema=mycat_nodel,
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 [$_NIOREACTOR-2-RW] (AbstractConnection.java:398) -close connection,reason:switch
datasource,MySQLConnection [id=29, lastTime=1428823025902, schema=mycat_nodel, old shema=mycat_nodel,
borrowed=false, fromSlaveDB=false, threadId=7062, charset=utf8, txIsolation=0, autocommit=true,
attachment=null, respHandler=null, host=121.40.121.133, port=3306, statusSync=null, writeQueue=0,
modifiedSQLExecuted=false]
04-12 15:21:06.622 WARN [$_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。

## 压缩协议支持

### 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。

## 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。

## 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。

## 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。**

## Mycat-Web

### Mycat-Web简介

### Mycat-web成长史

Mycat-web第一个版本

早在1年前，Mycat发起人Leader-us在群里号召要为Mycat-server提供一个Web端，对server端进行管理与监控。本人觉得自己已经有一个成型的java web开发平台。英勇的接下了这个活，而且响应很迅速。在群里招人组织一个Mycat-Web开发云团队。参与的兄弟也很积极。不到一个月的时间，就把server端的配置管理、监控管理开发完成！也很快把代码提交。后来因为本人自己工作原因，对Mycat-Web没有任何管理，这样就糊涂的混过了一年时间。在此间有热心的朋友，把jrds集成进来，原来从taobao.code迁移到github。就这样Mycat-web第一个版本就此结束了。

Mycat-web第二个版本(目前新版本)

经过第一个版本的开发，收到很多“赞美” - 这里是列表文本，界面out,使用配置复杂，基于源码进行扩展太繁琐。Leader-us也不时在群里号召Mycat-web重新开发。听到这些想着以后还怎么在这么火的开源项目见大家。不过还得感谢Leader-us的长久信任。本人再一次发起对Mycat-Web开发。设定开发原则，界面简单、时尚；使用简单；功能强大。基于这些原则成就了目前的Mycat-Web。不经历风雨，怎么见彩虹(Rainbow)

## Mycat-web架构及原理

Mycat-web 基础开发平台：基于rainbow-framework1.5.6开发。大家可能没听到过这个框架。您可能也百度不到。此框架是本人基于spring mvc+spring+mybatis经过深度封装形成的开发平台。此框架不做过多讲解，大家对此感兴趣可以专门联系本人。本人联系方式在群英传中Rainbow自我介绍有联系方式。

Mycat-web 数据库连接设计：采用了基于代码方式向spring ioc中注册一个DataSource。因此他能管理你所有的mycat、mysql服务。

Mycat-web监控：由开源的jrd实现。目前已经实现了Mycat、Mysql性能监控(jdbc连接获取)、Mycat的JVM内存、线程的监控(通过JMX获取)，Mycat,Mysql所在操作系统的CPU、内存、磁盘、网络的监控。(通过SNMP协议获取)

Mycat-web监控模板：采用freemark作为模板引擎，提供jrd所需的配置文件动态生成。

Mycat-web前端是基于bootstrap,jquery实现。

Mycat-web所有管理配置的持久化是基于内置的sqlite。

Mycat-web容器是基于jetty启动。

## Mycat-Web使用篇

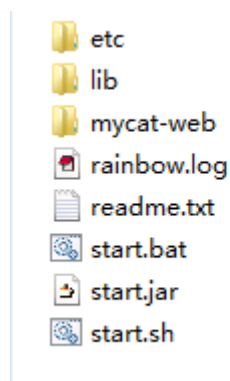
## Mycat-Web安装

### 1.Mycat-Web需要JDK1.7

jdk安装过程在此不做过多讲解，jdk安装

### 2.Mycat-Web安装与启动

解压Mycat-web-1.0.bate.zip



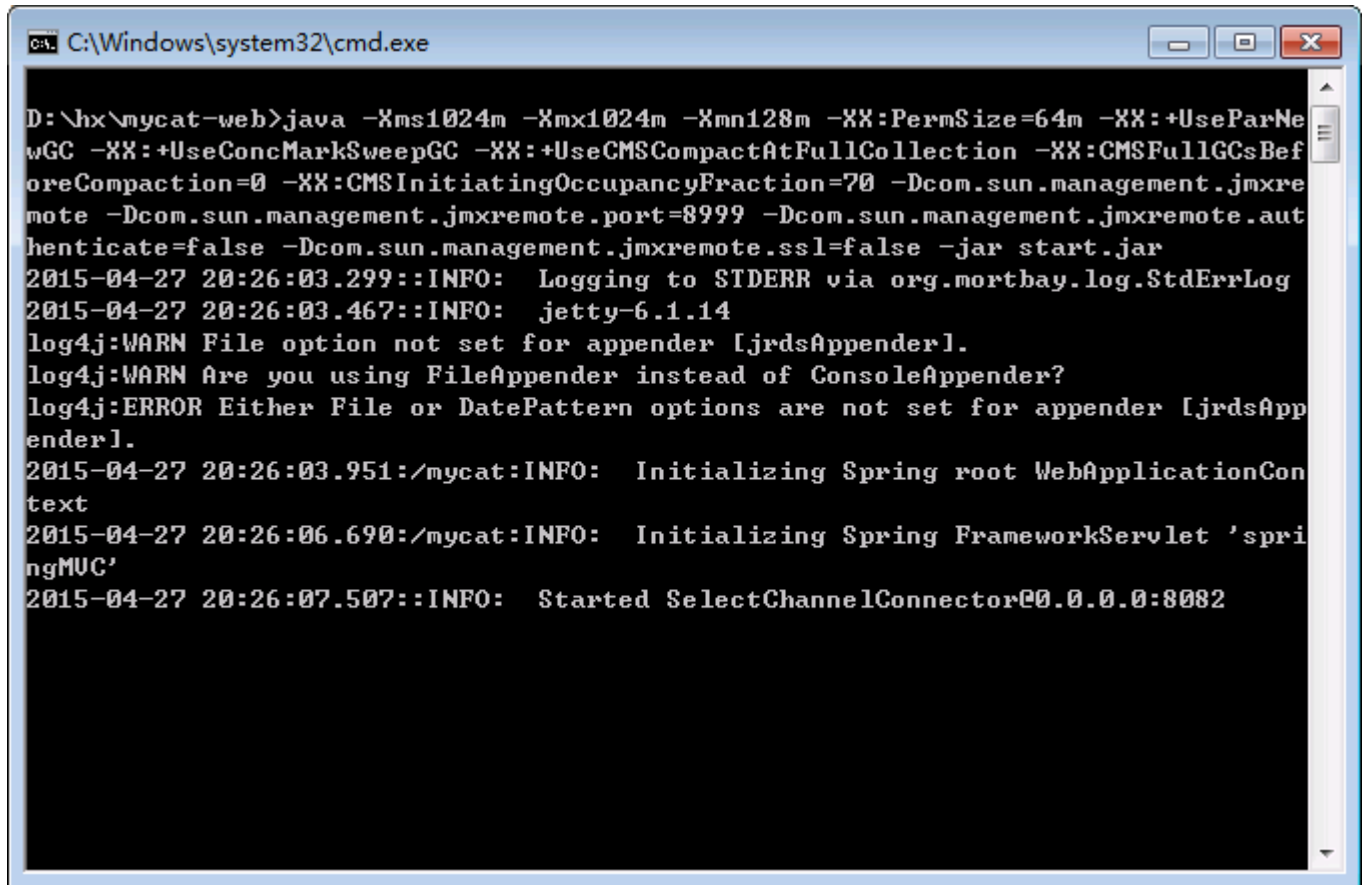
#### 2.1基于windows

双击start.bat

#### 2.2基于Linux

```
unzip Mycat-web-1.0.bate.zip /  
cd /Mycat-web-1.0.bate  
chmod 755 start.sh  
./start.sh
```

启动窗口



```
C:\Windows\system32\cmd.exe  
D:\hx\mycat-web>java -Xms1024m -Xmx1024m -Xmn128m -XX:PermSize=64m -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 -jar start.jar  
2015-04-27 20:26:03.299::INFO: Logging to STDERR via org.mortbay.log.StdErrLog  
2015-04-27 20:26:03.467::INFO: jetty-6.1.14  
log4j:WARN File option not set for appender [jrdsAppender].  
log4j:WARN Are you using FileAppender instead of ConsoleAppender?  
log4j:ERROR Either File or DatePattern options are not set for appender [jrdsAppender].  
2015-04-27 20:26:03.951:/mycat:INFO: Initializing Spring root WebApplicationContext  
2015-04-27 20:26:06.690:/mycat:INFO: Initializing Spring FrameworkServlet 'springMUC'  
2015-04-27 20:26:07.507::INFO: Started SelectChannelConnector@0.0.0.0:8082
```

### 2.3访问Mycat-Web

访问地址：<http://localhost:8082/mycat>

展示页面



# Mycat-Web功能介绍

## 一、Mycat管理

简介：

Mycat管理：配置mycat-server的jdbc连接操作。配置完成后,Mycat-web就与Mycat-server建立好连接。此处可以设置多个Mycat-server，同时也可以设置Mysql连接。为监控Mycat-server，端口配置为9066。

配置参数：

参数	描述
mycat名称	便于管理mycat，一字母或者加数字合租。
IP地址	Mycat-Server or Mysql：IP地址
端口port	Mycat：9066 MySql：3306
数据库实例名称	Mycat-Server or Mysql：数据库 如：test
用户名	Mycat-Server or Mysql：用户名
密码	Mycat-Server or Mysql：密码

## 二、Mycat性能

简介：

通过Mycat管理的配置建立好数据库连接后，在此菜单中可以通过9066所提供的监控命名来查看数据库性能参数。目前监控命令已经在第二个下拉框中列出。

三、Jmx管理

简介：

Jmx：是对JVM提供监控服务。通过JVM配置参数开启监控服务。在Mycat-Web中配置JMX，主要是生成jrds的JMX配置模板。

Jmx配置：

在java启动的设置如何参数：

- Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=8999
- Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
- Dcom.sun.management.jmxremote 开启jmx远程服务
- Dcom.sun.management.jmxremote.port=8999 远程访问端口为8999
- Dcom.sun.management.jmxremote.authenticate=false 不做身份验证
- Dcom.sun.management.jmxremote.ssl=false 不做ssl加密

JMX参数说明

参数	说明
JMX名称	为jrds监控时，显示的JMX名称，字母或字母数据组合
IP地址	JVM服务地址，Mycat-server所在的IP地址
端口	JVM开启端口 如上节中的8999
用户名	如果JMX开启身份验证，需要填写用户名称
密码	如果JMX开启身份验证，需要填写密码

四、snmp管理

简介：

snmp是：简单网络管理协议（SNMP），由一组网络管理的标准组成，包含一个应用层协议（application layer protocol）、数据库模型（database schema）和一组资源对象。该协议能够支持网络管理系统，用以 监测连接到网络上的设备是否有任何引起管理上关注的情况。在Mycat-Web中通过snmp协议对操作系统中的CPU 、内存、网络、磁盘获取监控数据，通过jrds展示。

snmp配置

目前Mycat-Web中暂时对linux系统开发监控服务。基于linux系统中安装SNMP，请查看

snmp安装

snmp参数说明

参数	说明
snmp名称	snmp标示符，字母或字母与数字组合



参数	说明
IP地址	被监控的机器ip地址
端口	默认snmp端口为161
snmp团体名称	默认为public

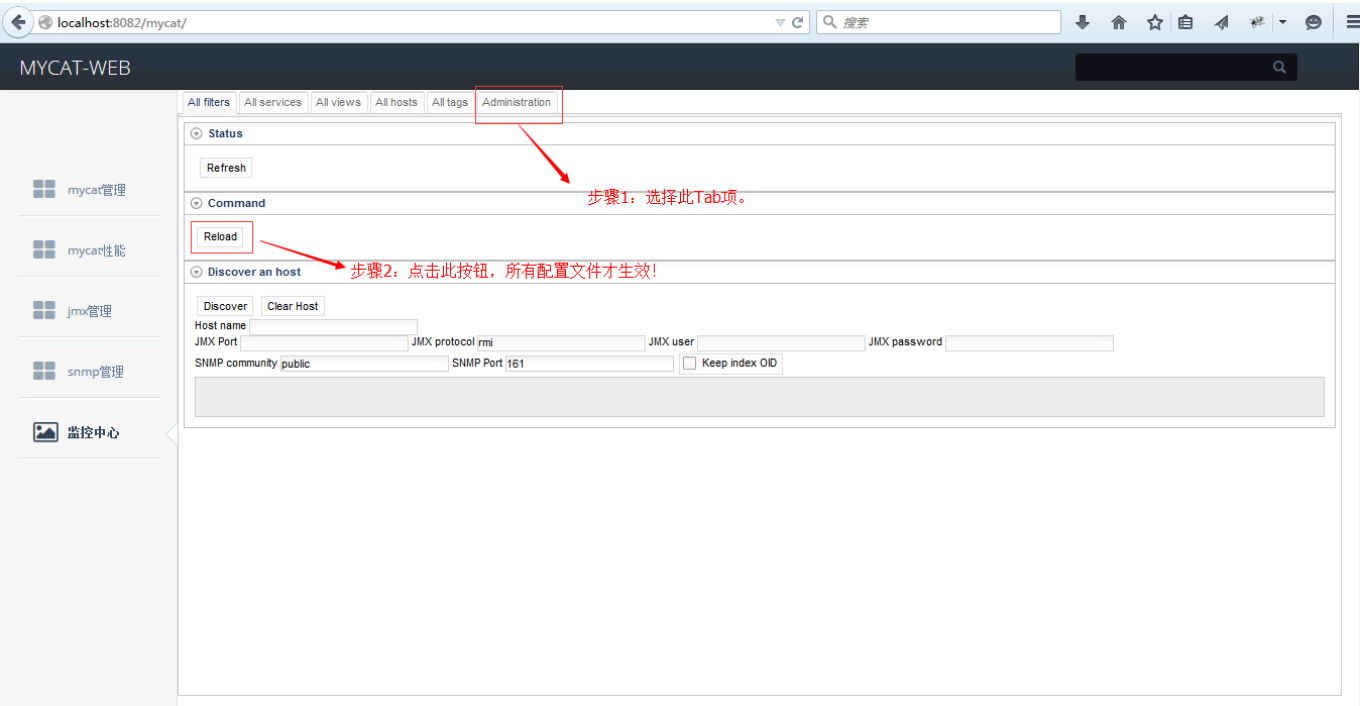
五、监控中心

简介

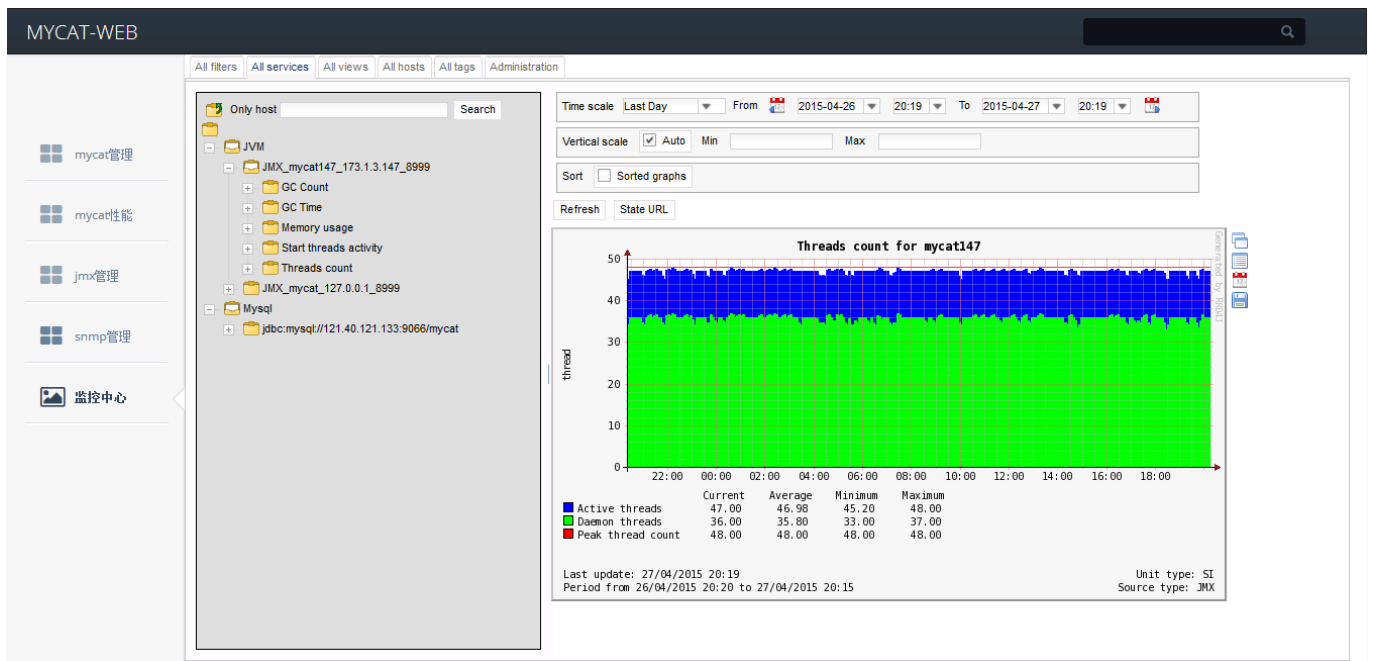
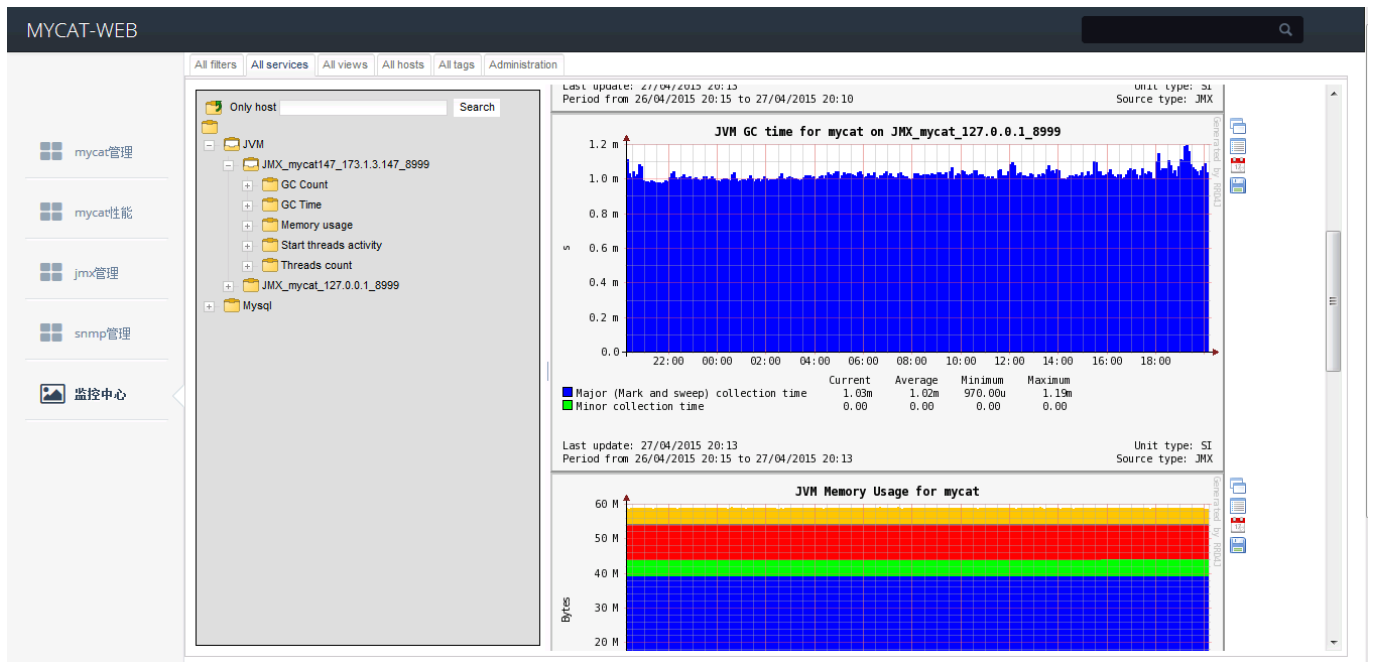
监控中心：是Mycat-Web所有监控的核心内容。基于jrdss实现。目前对Mycat-server、Mysql、JVM、OS中的CPU、内存、网络、磁盘进行监控。

使用说明

在前几节中我们说明了对Mycat配置、JMX配置、SNMP配置，在Mycat-Web中实际只在监控系统所需的配置地址中添加了文件。想真正生效需要如何操作



六、监控数据采集图



## 生产实践篇

### 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。

若不需要自动切换功能，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。

## 分表分库案例

## 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 ;`

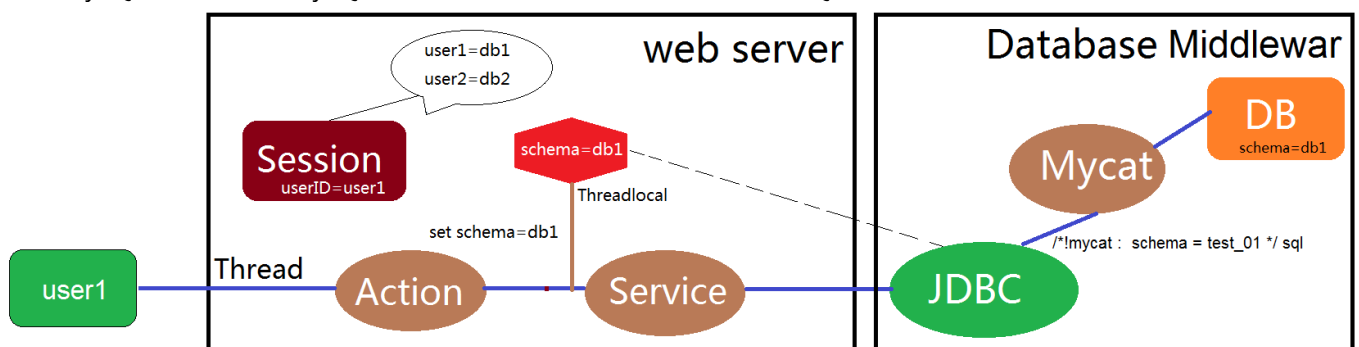
2. 在db前面建立proxy层，代理所有web过来的数据库请求。proxy层是用mycat实现的，web提交的sql过来时在注释中指定schema，proxy层根据指定的schema 转发sql请求。

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

## 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_tablespaces=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。

## 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。

## 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
[REDACTED]	[REDACTED]	[REDACTED]	200	0.55	1
	[REDACTED]	[REDACTED]	90	0.44	
	[REDACTED]	[REDACTED]	25	0.12	
	[REDACTED]	[REDACTED]	200	0.25	1
	[REDACTED]	[REDACTED]	200	0.55	1
	[REDACTED]	[REDACTED]	200	0.57	1
	[REDACTED]	[REDACTED]	75000	153.67	48
	[REDACTED]	[REDACTED]	3600	205.19	23
	[REDACTED]	[REDACTED]	500	16.12	3
	[REDACTED]	[REDACTED]	3500	7.5	22
	[REDACTED]	[REDACTED]	1500	2.45	9
	[REDACTED]	[REDACTED]			

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。

1. 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。

1. 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开了。

## 生产环境部署

### 单节点mycat部署

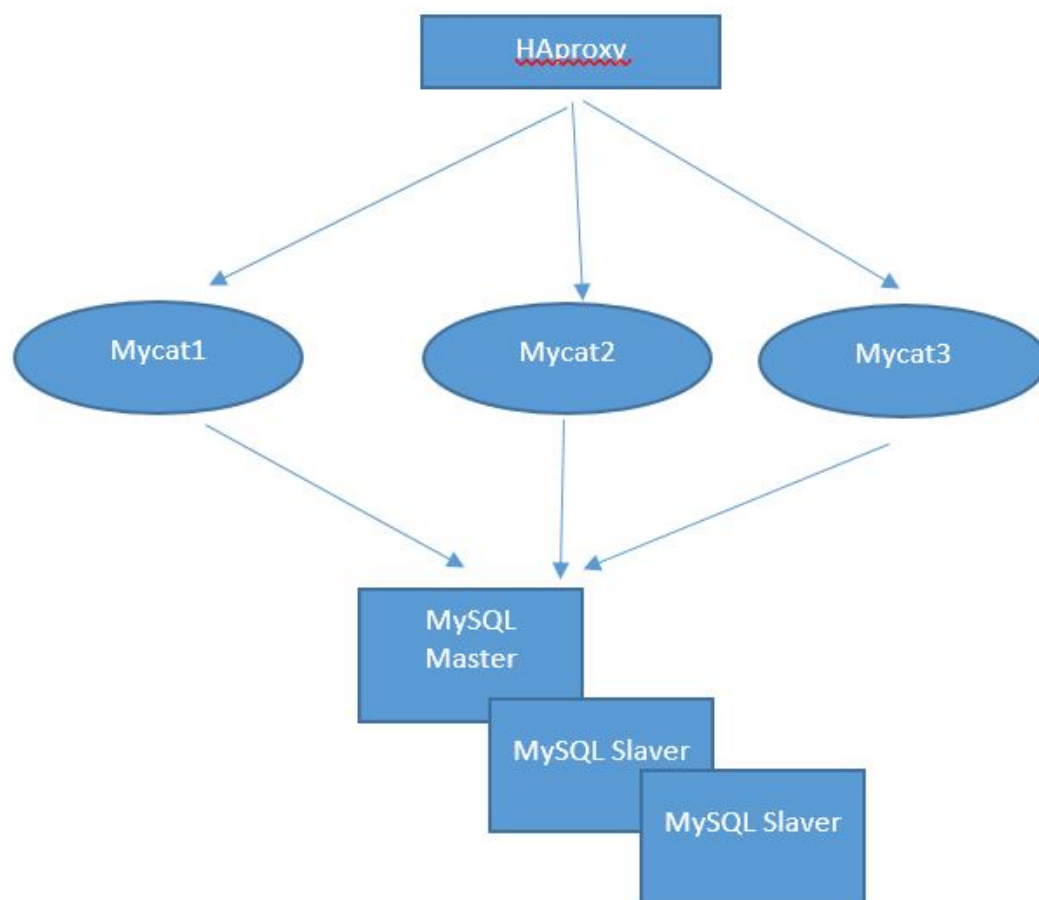
单节点mycat的部署指的是只部署一台mycat服务器，它与mycat集群部署是相对的，如果这台mycat服务器宕机了，mycat就不可用了。

### mycat的高可用与负载均衡

## 什么是高可用？

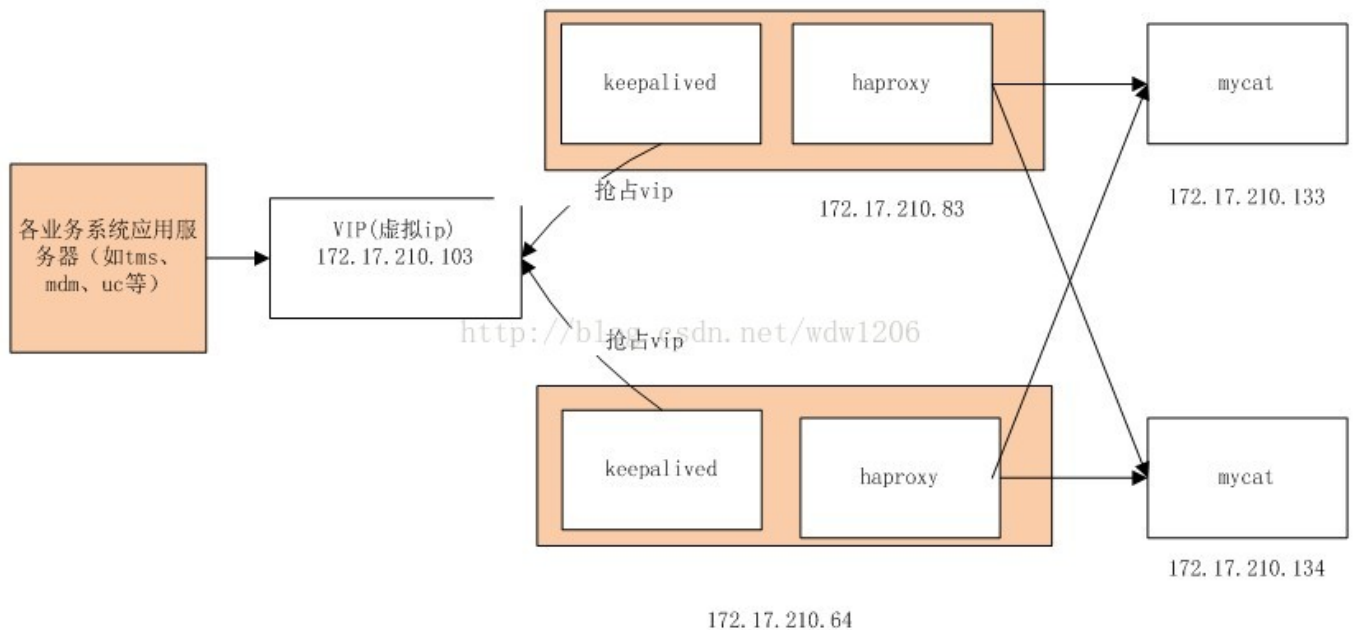
高可用通常也叫HA ( High Available )。指的是，一台服务器宕机了，照样能对外提供服务。常用的高可用软件方案有：LVS、keepalived、Heartbeat、roseHA ( roseHA为收费软件 ) 等。

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：



### 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。

## haproxy安装

```
useradd haproxy
#wget http://haproxy.lwt.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

## 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
```

#### 4、/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
```

#### 5、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
```

## 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
```

## 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)

## 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
```

## 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
```

## 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
```

## 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
vrrp_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 Fileforkeepalived
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
        auth_type PASS    用时有问题
        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
}

```

#### 粗体注意：

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是本地存在的网卡地址。有些服务器如果只有一个网卡，但被人把eth0改成eth1了，你再写eth0就找不到了。

#### 粗体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
sleep5
fi
if [ `ps -C haproxy --no-header |wc -l` -eq0 ];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}'|xargskill-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 stared ..." >> $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}'|xargskill-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 stared ..." >> $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
```

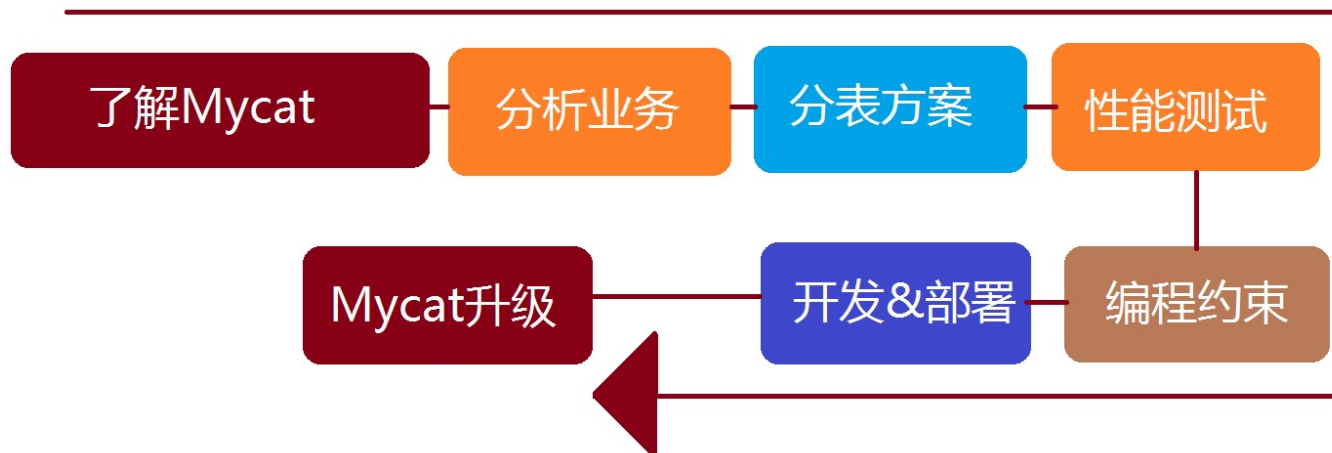
## Mycat实施指南

### 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

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。

## 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，则他们通常比较合适用时间范围分片，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

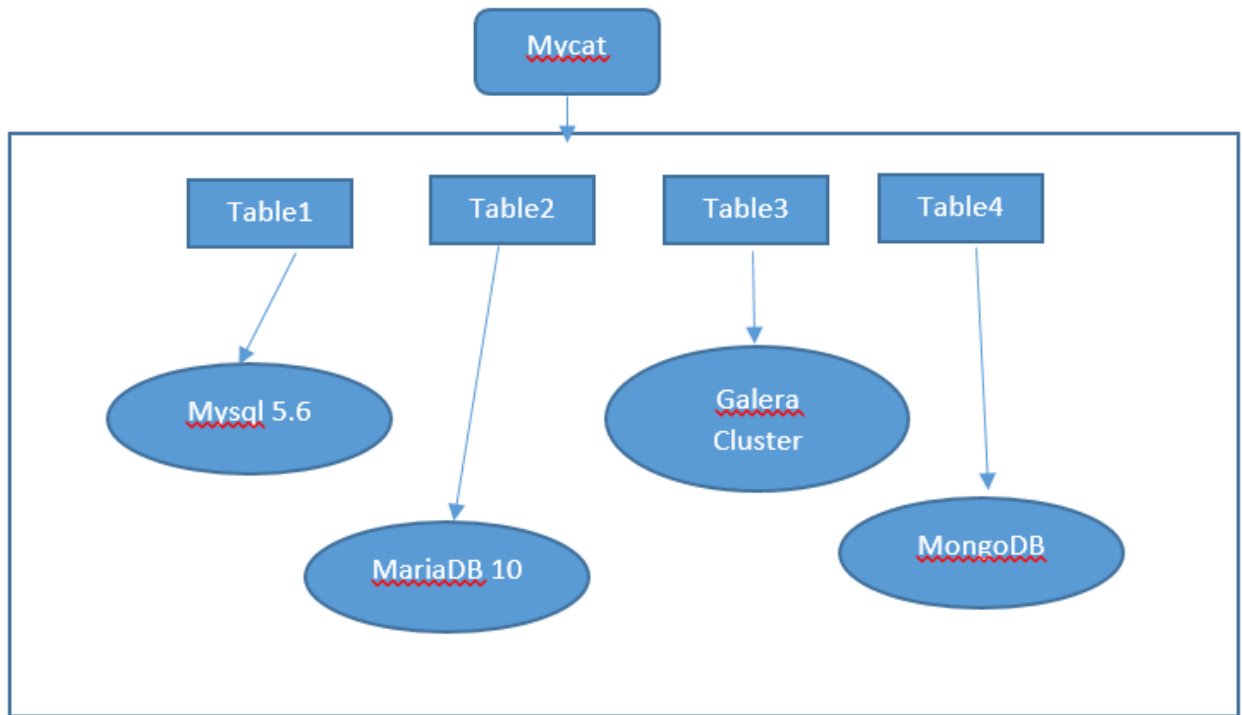
## 后端存储的选择

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。



## 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。

## DataNode的分布问题

DataNode代表MySQL数据库上的一个Database，因此一个分片表的数据节点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。

当我们将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-callldate" />
```

其中dn1xxx与dn2xxxx是分别定义在DataHost1上与DataHost2上的377个分片。

## 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SQL
- 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Hibernat，但不建议使用Hibernat，而是建议Mybatis以及直接JDBC操作，原因Hibernat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。

## 数据迁移与扩容实践

Mycat扩容：一致性Hash，范围分片等

### 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。

## 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="org.opencloudb.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://org.opencloudb/">
  <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="savle_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。

## 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="org.opencloudb.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://org.opencloudb/">
  <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="slave_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="slave_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。

## 开始迁移

使用org.opencldb.util.rehasher.RehashLauncher类进行数据迁移。参数以命令行的形式进行载入。如

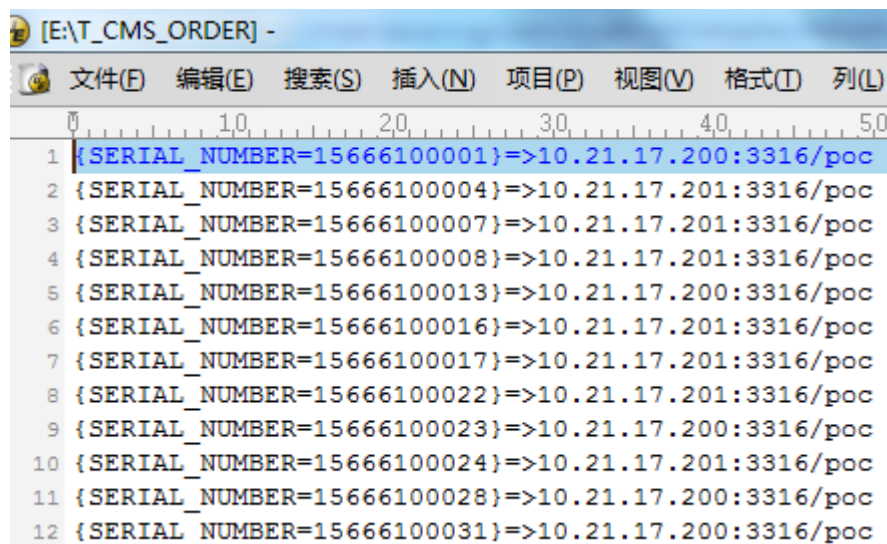
```
-jdbcDriver=xxxxx -jdbcUrl=... -host=192.168.1.1:3316 -user=xxxx -password=xxxx -database=xxxx
```

- 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

org.opencldb.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org.opencldb.util.rehasher.RehashLauncher后生成的文件格式如下：



```
[E:\T_CMS_ORDER] -
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





```

do
echo "$i"
done
echo " "

#取出rehash需要的SerNum(已经用in拼接好)
for n in `cat $order_fn`
do
    condOrder=$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

## 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：

## 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="org.opencloudb.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org.opencloudb.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
```

## 数据迁移的注意事项

### **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。

## 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。

## 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parallel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。

## 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。

## 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。

## 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。

## 语法和注意事项

语法示例：

```
load DATA LOCAL INFILE 'd:\88\mycat.txt' IGNORE INTO TABLE test character set 'utf8' (id,sid,asf);
```

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。

## 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。

## 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。

## 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;
```

## 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
```

## 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
```

## 版本选择与升级指南

### 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

## 目前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。

## 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。

## mycat1.4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中的统配符为：

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

```
<dataNode name= "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。



目前1.3 版本比1.2稳定性有很大提高，但是支持的特性也有很大提高，建议升级到1.3.0.3最新稳定版。

1.3版本中1.3.0.3为最稳定版本，1.3.0.2存在较多bug，以废弃不建议使用。

1.4 为目前的开发版本，进一步加强了新特性，如果是项目属于开发阶段可以研究1.4，等到代码开发完毕1.4预计发布稳定版本，目前的1.4开发版本存在部分bug，如果担心生产问题可以使用1.3版本，后期建议升级1.4。

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>

## 性能调优

### 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：

$BDP = \text{link bandwidth} \times RTT$

若应用程序通过一个100MB / s的局域网通信，其RTT为500ms，则BDP为：50MB / s × 0.050 / 8625M = 625KB。Linux2.6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：

$\text{throughput} = \text{window\_size} / RTT$

110 KB / 0.050 = 2.2 MB / s

使用上面计算的窗口大小，得到带宽为12.5 MB / s，即：

625 KB / 0.050 = 12.5 MB / 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。

### JVM调优

Mycat的jvm相关配置是在warrper启动中配置例如：

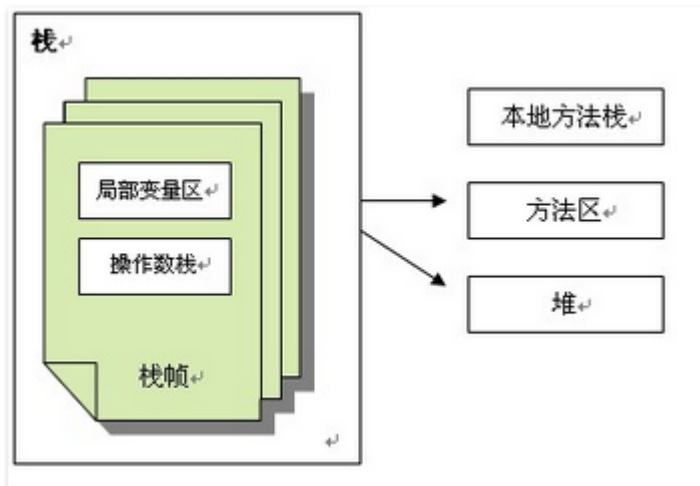
linux下 startup\_nowrap.sh

其他版本都会在对应的配置文件中配置。

## JVM结构

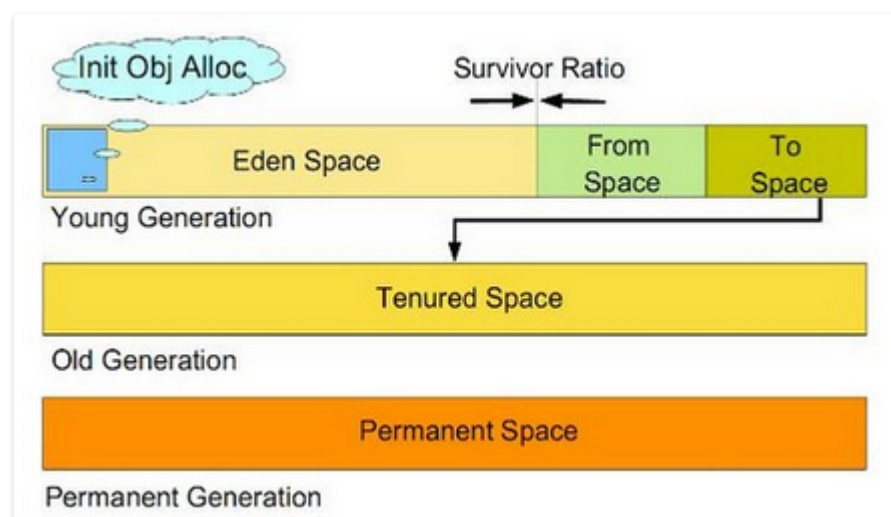
JVM内存结构由堆、栈、本地方法栈、方法区等部分组成，另外JVM分别对新生代和旧生代采用不同的垃圾回收机制。

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生代，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。

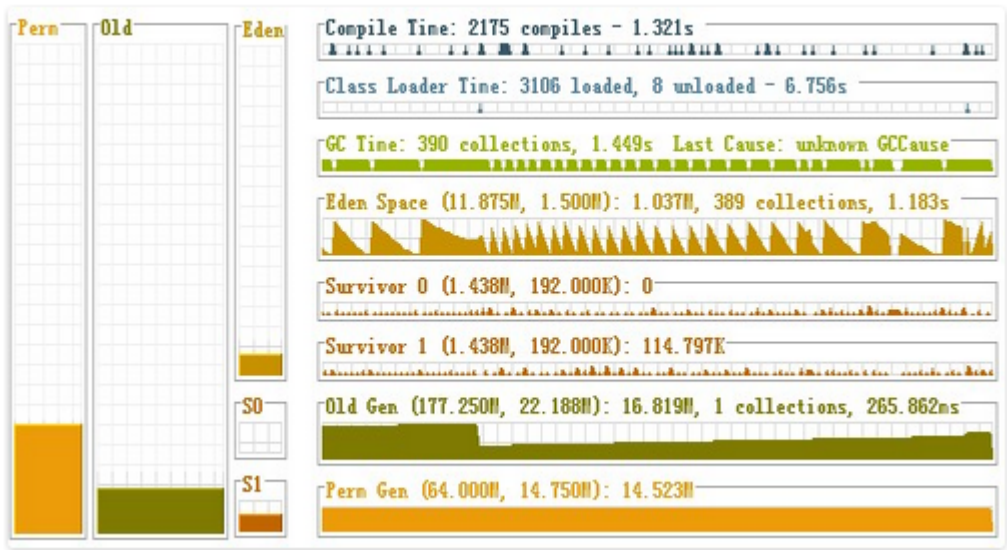
# JVM垃圾回收机制

JVM分别对新生代和旧生代采用不同的垃圾回收机制

新生代的GC：

新生代通常存活时间较短，因此基于Copying算法来进行回收，所谓Copying算法就是扫描出存活的对象，并复制到一块新的完全未使用的空间中，对应于新生代，就是在Eden和FromSpace或ToSpace之间copy。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在新生代区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。当连续分配对象时，对象会逐渐从eden到survivor，最后到旧生代，

用javavisualVM来查看，能明显观察到新生代满了后，会把对象转移到旧世代，然后清空继续装载，当旧世代也满了后，就会报outofmemory的异常，如下图所示：



在执行机制上JVM提供了串行GC(SerialGC)、并行回收GC(ParallelScavenge)和并行GC(ParNew)

1)串行GC

在整个扫描和复制过程采用单线程的方式来进行，适用于单CPU、新生代空间较小及对暂停时间要求不是非常高的应用上，是client级别默认的GC方式，可以通过-XX:+UseSerialGC来强制指定 2)并行回收GC 在整个扫描和复制过程采用多线程的方式来进行，适用于多CPU、对暂停时间要求较短的应用上，是server级别默认采用的GC方式，可用-XX:+UseParallelGC来强制指定，用-XX:ParallelGCThreads=4来指定线程数

3)并行GC

与旧生代的并发GC配合使用

旧生代的GC：

旧世代与新生代不同，对象存活的时间比较长，比较稳定，因此采用标记(Mark)算法来进行回收，所谓标记就是扫描出存活的对象，然后再进行回收未被标记的对象，回收后对用空出的空间要么进行合并，要么标记出来便于下次进行分配，总之就是要减少内存碎片带来的效率损耗。在执行机制上JVM提供了串行GC(SerialMSC)、并行GC(parallelMSC)和并发GC(CMS)，具体算法细节还有待进一步深入研究。

以上各种GC机制是需要组合使用的，指定方式由下表所示：

指定方式	新生代GC方式	旧世代GC方式
-XX:+UseSerialGC	串行GC	串行GC
-XX:+UseParallelGC	并行回收GC	并行GC
-XX:+UseConcMarkSweepGC	并行GC	并发GC
-XX:+UseParNewGC	并行GC	串行GC
-XX:+UseParallelOldGC	并行回收GC	并行GC
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC	串行GC	并发GC
不支持的组合	1、-XX:+UseParNewGC -XX:+UseParallelOldGC 2、-XX:+UseParNewGC -XX:+UseSerialGC	

## 常用垃圾回收器与方法

垃圾收集GC（Garbage Collection）是Java语言的核心技术之一，之前我们曾专门探讨过Java 7新增的垃圾回收器G1的新特性，但在JVM的内部运行机制上看，Java的垃圾回收原理与机制并未改变。垃圾收集的目的在于清除不再使用的对象。GC通过确定对象是否被活动对象引用来确定是否收集该对象。GC首先要判断该对象是否是时候可以收集。两种常用的方法是引用计数和对象引用遍历。

### 引用计数收集器

引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象（不是引用）都有一个引用计数。当一个对象被创建时，且将该对象分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（ $a = b$ , 则b引用的对象+1），但当一个对象的某个引用超过了生命周期或者被设置为一个新值时，对象的引用计数减1。任何引用计数为0的对象可以被当作垃圾收集。当一个对象被垃圾收集时，它引用的任何对象计数减1。

优点：引用计数收集器可以很快的执行，交织在程序运行中。对程序不被长时间打断的实时环境比较有利。

缺点：无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0。

### 跟踪收集器

早期的JVM使用引用计数，现在大多数JVM采用对象引用遍历。对象引用遍历从一组对象开始，沿着整个对象图上的每条链接，递归确定可到达（reachable）的对象。如果某对象不能从这些根对象的一个（至少一个）到达，则将它作为垃圾收集。在对象遍历阶段，GC必须记住哪些对象可以到达，以便删除不可到达的对象，这称为标记（marking）对象。

下一步，GC要删除不可到达的对象。删除时，有些GC只是简单的扫描堆栈，删除未标记的未标记的对象，并释放它们的内存以生成新的对象，这叫做清除（sweeping）。这种方法的问题在于内存会分成好多小段，而它们不足以用于新的对象，但是组合起

来却很大。因此，许多GC可以重新组织内存中的对象，并进行压缩（compact），形成可利用的空间。

为此，GC需要停止其他的活动活动。这种方法意味着所有与应用程序相关的工作停止，只有GC运行。结果，在响应期间增减了许多混杂请求。另外，更复杂的GC不断增加或同时运行以减少或者清除应用程序的中断。有的GC使用单线程完成这项工作，有的则采用多线程以增加效率。

#### 一些常用的垃圾收集器

##### （1）标记 - 清除收集器

这种收集器首先遍历对象图并标记可到达的对象，然后扫描堆栈以寻找未标记对象并释放它们的内存。这种收集器一般使用单线程工作并停止其他操作。并且，由于它只是清除了那些未标记的对象，而并没有对标记对象进行压缩，导致会产生大量内存碎片，从而浪费内存。

##### （2）标记 - 压缩收集器

有时也叫标记 - 清除 - 压缩收集器，与标记 - 清除收集器有相同的标记阶段。在第二阶段，则把标记对象复制到堆栈的新域中以便压缩堆栈。这种收集器也停止其他操作。

##### （3）复制收集器

这种收集器将堆栈分为两个域，常称为半空间。每次仅使用一半的空间，JVM生成的新对象则放在另一半空间中。GC运行时，它把可到达对象复制到另一半空间，从而压缩了堆栈。这种方法适用于短生存期的对象，持续复制长生存期的对象则导致效率降低。并且对于指定大小堆来说，需要两倍大小的内存，因为任何时候都只使用其中的一半。

##### （4）增量收集器

增量收集器把堆栈分为多个域，每次仅从一个域收集垃圾，也可理解为把堆栈分成一小块一小块，每次仅对某一个块进行垃圾收集。这会造成较小的应用程序中断时间，使得用户一般不能觉察到垃圾收集器正在工作。

##### （5）分代收集器

复制收集器的缺点是：每次收集时，所有的标记对象都要被拷贝，从而导致一些生命周期很长的对象被来回拷贝多次，消耗大量的时间。而分代收集器则可解决这个问题，分代收集器把堆栈分为两个或多个域，用以存放不同寿命的对象。JVM生成的新对象一般放在其中的某个域中。过一段时间，继续存在的对象(非短命对象)将获得使用期并转入更长寿命的域中。分代收集器对不同的域使用不同的算法以优化性能。

#### 并行收集器

并行收集器使用某种传统的算法并使用多线程并行的执行它们的工作。在多CPU机器上使用多线程技术可以显著的提高java应用程序的可扩展性。

## 参数优化与讲解

现在的JVM运行Java程序（和其它的兼容性语言）时在高效性和稳定性方面做的非常出色。自适应内存管理、垃圾收集、及时编译、动态类加载、锁优化，在运行时，JVM会不断的计算并优化应用或者应用的某些部分。

虽然有了这种程度的自动化（或者说有这么自动化），但是JVM仍然提供了足够多的外部监控和手动调优工具。在有错误或低性能的情况下，JVM必须能够让专家调试。顺便说一句，除了这些隐藏在引擎中的神奇功能，允许大范围的手动调优也是现代JVM的优势之一。有趣的是，一些命令行参数可以在JVM启动时传入到JVM中。一些JVM提供了几百个这样的参数，所以如果没有这方面的知识很容易迷失，本章节讲述各个参数。我们将专注于Java7的Sun/Oracle HotSpot JVM，大多数情况下，这些

参数也会适用于其他一些流行的JVM里。

`-server` and `-client`

有两种类型的 HotSpot JVM，即“server”和“client”。服务端的VM中的默认为堆提供了一个更大的空间以及一个并行的垃圾收集器，并且在运行时可以更大程度地优化代码。客户端的VM更加保守一些（校对注：这里作者指客户端虚拟机有较小的默认堆大小），这样可以缩短JVM的启动时间和占用更少的内存。有一个叫“JVM能效学”的概念，它会在JVM启动的时候根据可用的硬件和操作系统来自动的选择JVM的类型。标准表中，我们可以看到客户端的VM只在32位系统中可用。

如果我们不喜欢预选（校对注：指JVM自动选择的JVM类型）的JVM，我们可以使用`-server`和`-client`参数来设置使用服务端或客户端的VM。虽然当初服务端VM的目标是长时间运行的服务进程，但是现在看来，在运行独立应用程序时它比客户端VM有更出色的性能。当应用的性能非常重要时，我推荐使用`-server`参数来选择服务端VM。一个常见的问题：在一个32位的系统上，HotSpot JDK可以运行服务端VM，但是32位的JRE只能运行客户端VM。

`-version` and `-showversion`

我们现在可以使用`-version`参数，它可以打印出正在使用的JVM的信息。例如：

```
C:\Users\sw>java -version
java version "1.7.0_67"
Java(TM) SE Runtime Environment (build 1.7.0_67-b01)
Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)
```

输出显示的是Java版本号(1.7.0\_67)和JRE确切的build号(1.7.0\_67-b01)。我们也可以看到JVM的名字(Java HotSpot(TM))、类型(Server)和build ID (24.65-b04)。除此之外，我们还知道JVM以混合模式(mixed mode)在运行，这是HotSpot默认的运行模式，意味着JVM在运行时可以动态的把字节码编译为本地代码。

`-Xint`, `-Xcomp`, 和 `-Xmixed`

`-Xint`和`-Xcomp`参数和我们的日常工作不是很相关，但是我非常有兴趣通过它来了解下JVM。在解释模式(interpreted mode)下，`-Xint`标记会强制JVM执行所有的字节码，当然这会降低运行速度，通常低10倍或更多。`-Xcomp`参数与它（`-Xint`）正好相反，JVM在第一次使用时会把所有的字节码编译成本地代码，从而带来最大程度的优化。这听起来不错，因为这完全绕开了缓慢的解释器。然而，很多应用在使用`-Xcomp`也会有一些性能损失，当然这比使用`-Xint`损失的少，原因是`-xcomp`没有让JVM启用JIT编译器的全部功能。JIT编译器在运行时创建方法使用文件，然后一步一步的优化每一个方法，有时候会主动的优化应用的行为。这些优化技术，比如，积极的分支预测（optimistic branch prediction），如果不先分析应用就不能有效的使用。另一方面方法只有证明它们与此相关时才会被编译，也就是，在应用中构建某种热点。被调用很少（甚至只有一次）的方法在解释模式下会继续执行，从而减少编译和优化成本。

注意混合模式也有他自己的参数，`-Xmixed`。最新版本的HotSpot的默认模式是混合模式，所以我们不需要特别指定这个标记。我们来用对象填充HashMap然后检索它的结果做一个简单的用例。每一个例子，它的运行时间都是很多次运行的平均时间。

```
$ java -server -showversion Benchmark
java version "1.6.0_24" Java(TM) SE Runtime Environment (build 1.6.0_24-b07) Java HotSpot(TM) Server VM
(build 19.1-b02, mixed mode)
Average time: 0.856449 seconds
```

```
$ java -server -showversion -Xcomp Benchmark
```

```
java version "1.6.0_24" Java(TM) SE Runtime Environment (build 1.6.0_24-b07) Java HotSpot(TM) Server VM
(build 19.1-b02, compiled mode)
Average time: 0.950892 seconds
```

```
$ java -server -showversion -Xint Benchmark
java version "1.6.0_24" Java(TM) SE Runtime Environment (build 1.6.0_24-b07) Java HotSpot(TM) Server VM
(build 19.1-b02, interpreted mode)
Average time: 7.622285 seconds
```

当然也有很多使-Xcomp表现很好的例子。特别是运行时间长的应用，我强烈建议大家使用JVM的默认设置,让JIT编译器充分发挥其动态潜力，毕竟JIT编译器是组成JVM最重要的组件之一。事实上，正是因为JVM在这方面的进展才让Java不再那么慢。

## JVM 参数分类

HotSpot JVM 提供了三类参数。第一类包括了标准参数。顾名思义，标准参数中包括功能和输出的参数都是很稳定的，很可能在将来的JVM版本中不会改变。你可以用java命令（或者用 java -help）检索出所有标准参数。我们在第一部分中已经见到过一些标准参数，例如：-server。

第二类是X参数，非标准化的参数在将来的版本中可能会改变。所有的这类参数都以-X开始，并且可以用java -X来检索。注意，不能保证所有参数都可以被检索出来，其中就没有-Xcomp。

第三类是包含XX参数（到目前为止最多的），它们同样不是标准的，甚至很长一段时间内不被列出来（最近，这种情况有改变，我们将在本系列的第三部分中讨论它们）。然而，在实际情况中X参数和XX参数并没有什么不同。X参数的功能是十分稳定的，然而很多XX参数仍在实验当中（主要是JVM的开发者用于debugging和调优JVM自身的实现）。值的一读的介绍非标准参数的文档 HotSpot JVM documentation，其中明确的指出XX参数不应该在不了解的情况下使用。这是真的，并且我认为这个建议同样适用于X参数（同样一些标准参数也是）。不管类别是什么，在使用参数之前应该先了解它可能产生的影响。

用一句话来说明XX参数的语法。所有的XX参数都以“-XX:”开始，但是随后的语法不同，取决于参数的类型。

a. 对于布尔类型的参数，我们有“+”或“-”，然后才设置JVM选项的实际名称。例如，-XX:+用于激活选项，而-XX:-用于注销选项。

b. 对于需要非布尔值的参数，如string或者integer，我们先写参数的名称，后面加上“=”，最后赋值。例如，-XX:=给赋值。

```
-XX:+PrintCompilation and -XX:+CITime
```

查看JIT编译工作。通过设置-XX:+PrintCompilation，我们可以简单的输出一些关于从字节码转化成本地代码的编译过程。

```
-XX:+UnlockExperimentalVMOptions
```

有些时候当设置一个特定的JVM参数时，JVM会在输出“Unrecognized VM option”后终止。如果发生了这种情况，你应该首先检查你是否输错了参数。然而，如果参数输入是正确的，并且JVM并不识别，你或许需要设置

```
-XX:+UnlockExperimentalVMOptions 来解锁参数
```

```
-XX:+LogCompilation and -XX:+PrintOptoAssembly
```

如果你在一个场景中发现使用 -XX:+PrintCompilation，不能够给你足够详细的信息，你可以使用 -XX:+LogCompilation把扩展的编译输出写到“hotspot.log”文件中

```
-XX:+PrintFlagsFinal and -XX:+PrintFlagsInitial
```

HotSpot JVM 提供给了两个新的参数，在JVM启动后，在命令行中可以输出所有XX参数和值，表格的每一行包括五列，来表示

一个XX参数。第一列表示参数的数据类型，第二列是名称，第四列为值，第五列是参数的类别。第三列“=”表示第四列是参数的默认值，而“:=”表明了参数被用户或者JVM赋值了。

```
java -client -XX:+PrintFlagsFinal Benchmark [Global flags] uintx AdaptivePermSizeWeight = 20
{product} uintx AdaptiveSizeDecrementScaleFactor = 4 {product} uintx
AdaptiveSizeMajorGCDecayTimeScale = 10 {product} uintx AdaptiveSizePausePolicy = 0
{product}[...] uintx YoungGenerationSizeSupplementDecay = 8 {product} uintx
YoungPLABSize = 4096 {product} bool ZeroTLAB = false
{product} intx hashCode = 0 {product}
```

如果我们只想看下所有XX参数的默认值，能够用一个相关的参数，-XX:+PrintFlagsInitial。用 -XX:+PrintFlagsInitial, 只是展示了第三列为“=”的数据（也包括那些被设置其他值的参数）。

然而，注意当与-XX:+PrintFlagsFinal 对比的时候，一些参数会丢失，大概因为这些参数是动态创建的。

研究表格的内容是很有意思的，通过比较client和server VM的行为，很明显了解哪些参数会影响其他的参数。

-XX:+PrintCommandLineFlags

让我们看下另外一个参数，事实上这个参数非常有用:-XX:+PrintCommandLineFlags。这个参数让JVM打印出那些已经被用户或者JVM设置过的详细的XX参数的名称和值。

换句话说，它列举出 -XX:+PrintFlagsFinal的结果中第三列有“:=”的参数。以这种方式，我们可以用

-XX:+PrintCommandLineFlags作为快捷方式来查看修改过的参数。看下面的例子。

```
java -server -XX:+PrintCommandLineFlags Benchmark

-XX:InitialHeapSize=132425856 -XX:MaxHeapSize=2118813696 -XX:+PrintCommandLineFlags -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
```

现在如果我们每次启动java 程序的时候设置 -XX:+PrintCommandLineFlags 并且输出到日志文件上，这样会记录下我们设置的JVM参数对应用程序性能的影响。类似于 -showversion，建议 -XX:+PrintCommandLineFlags 这个参数应该总是设置在JVM启动的配置项里。因为你从不知道你什么时候会需要这些信息。

-Xms and -Xmx (or: -XX:InitialHeapSize and -XX:MaxHeapSize)

-Xms和-Xmx可以说是最流行的JVM参数，它们可以允许我们指定JVM的初始和最大堆内存大小。一般来说，这两个参数的数值单位是Byte，但同时它们也支持使用速记符号，比如“k”或者“K”代表“kilo”，“m”或者“M”代表“mega”，“g”或者“G”代表“giga”。举个例子，下面的命令启动了一个初始化堆内存为128M，最大堆内存为2G，名叫“TestJava”的Java应用程序。

```
java -Xms128m -Xmx2g TestJava
```

在实际使用过程中，初始化堆内存的大小通常被视为堆内存大小的下界。然而JVM可以在运行时动态的调整堆内存的大小，所以理论上来说我们有可能看到堆内存的大小小于初始化堆内存的大小。但是即使在非常低的堆内存使用下，我也从来没有遇到过这种情况。这种行为将会方便开发者和系统管理员，因为我们可以通过将“-Xms”和“-Xmx”设置为相同大小来获得一个固定大小的堆内存。-Xms和-Xmx实际上是-XX:InitialHeapSize和-XX:MaxHeapSize的缩写。我们也可以直接使用这两个参数，它们所起得效果是一样的：

```
java -XX:InitialHeapSize=128m -XX:MaxHeapSize=2g TestJava
```



需要注意的是，所有JVM关于初始\最大堆内存大小的输出都是使用它们的完整名称：“InitialHeapSize”和“InitialHeapSize”。所以当你查询一个正在运行的JVM的堆内存大小时，如使用-XX:+PrintCommandLineFlags参数或者通过JMX查询，你应该寻找“InitialHeapSize”和“InitialHeapSize”标志而不是“Xms”和“Xmx”。

```
-XX:+HeapDumpOnOutOfMemoryError and -XX:HeapDumpPath
```

如果我们没法为-Xmx（最大堆内存）设置一个合适的大小，那么就有可能面临内存溢出（OutOfMemoryError）的风险，这可能是最常见的内存溢出。

我们可以通过设置-XX:+HeapDumpOnOutOfMemoryError让JVM在发生内存溢出时自动的生成堆内存快照。有了这个参数，当我们不得不面对内存溢出异常的时候会节约大量的时间。默认情况下，堆内存快照会保存在JVM的启动目录下名为java\_pid.hprof的文件里（在这里就是JVM进程的进程号）。也可以通过设置-XX:HeapDumpPath=来改变默认的堆内存快照生成路径，可以是相对或者绝对路径。

虽然这一切听起来很不错，但有一点我们需要牢记。堆内存快照文件有可能很庞大，特别是当内存溢出错误发生的时候。因此，我们推荐将堆内存快照生成路径指定到一个拥有足够磁盘空间的地方。

内存溢出快照：

```
#
# There is insufficient memory for the Java Runtime Environment to continue.
# pthread_getattr_np
# Possible reasons:
#   The system is out of physical RAM or swap space
#   In 32 bit mode, the process size limit was hit
# Possible solutions:
#   Reduce memory load on the system
#   Increase physical memory or swap space
#   Check if swap backing store is full
#   Use 64 bit Java on a 64 bit OS
#   Decrease Java heap size (-Xmx/-Xms)
#   Decrease number of Java threads
#   Decrease Java thread stack sizes (-Xss)
#   Set larger code cache with -XX:ReservedCodeCacheSize=
# This output file may be truncated or incomplete.
#
# Out of Memory Error (os_linux_x86.cpp:715), pid=3644, tid=140602196883200
#
# JRE version: 6.0_33-b33
# Java VM: OpenJDK 64-Bit Server VM (23.25-b01 mixed mode linux-amd64 compressed oops)
# Derivative: IcedTea6 1.13.5
# Distribution: Ubuntu 12.04 LTS, package 6b33-1.13.5-1ubuntu0.12.04
# Failed to write core dump. Core dumps have been disabled. To enable core dumping, try "ulimit -c
unlimited" before starting Java again
#

----- T H R E A D -----

Current thread (0x00007fe084089800): VMThread [stack: 0x0000000000000000,0x0000000000000000] [id=3646]

Stack: [0x0000000000000000,0x0000000000000000], sp=0x00007fe07fffe970, free space=137306832890k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V [libjvm.so+0x83d289]
V [libjvm.so+0x3e6610]
V [libjvm.so+0x6e973a]
V [libjvm.so+0x6e97cd]
```

V [libjvm.so+0x7fad51]  
V [libjvm.so+0x841eed]  
V [libjvm.so+0x6e06c2]

----- P R O C E S S -----

Java Threads: ( => current thread )

0x00007fe084009000 JavaThread "Unknown thread" [\_thread\_blocked, id=3645,  
stack(0x00007fe08a70c000,0x00007fe08a80d000)]

Other Threads:

=>0x00007fe084089800 VMThread [stack: 0x0000000000000000,0x0000000000000000] [id=3646]

VM state:not at safepoint (normal execution)

VM Mutex/Monitor currently owned by a thread: None

Heap

def new generation total 9664K, used 172K [0x00000000cc000000, 0x00000000cca70000, 0x00000000d9e00000)  
eden space 8640K, 2% used [0x00000000cc000000, 0x00000000cc02b3a0, 0x00000000cc870000)  
from space 1024K, 0% used [0x00000000cc870000, 0x00000000cc870000, 0x00000000cc970000)  
to space 1024K, 0% used [0x00000000cc970000, 0x00000000cc970000, 0x00000000cca70000)  
tenured generation total 21376K, used 0K [0x00000000d9e00000, 0x00000000db2e0000, 0x00000000f5a00000)  
the space 21376K, 0% used [0x00000000d9e00000, 0x00000000d9e00000, 0x00000000d9e00200,  
0x00000000db2e0000)  
compacting perm gen total 21248K, used 606K [0x00000000f5a00000, 0x00000000f6ec0000, 0x0000000100000000)  
the space 21248K, 2% used [0x00000000f5a00000, 0x00000000f5a97b70, 0x00000000f5a97c00,  
0x00000000f6ec0000)  
No shared spaces configured.

Card table byte\_map: [0x00007fe0801eb000,0x00007fe08038c000] byte\_map\_base: 0x00007fe07fb8b000

Polling page: 0x00007fe08a818000

Code Cache [0x00007fe08038c000, 0x00007fe0805fc000, 0x00007fe08338c000)

total\_blobs=38 nmethods=0 adapters=17 free\_code\_cache=48869Kb largest\_free\_block=50041216

Compilation events (0 events):

No events

GC Heap History (0 events):

No events

Deoptimization events (0 events):

No events

Internal exceptions (0 events):

No events

Events (10 events):

Event: 0.017 loading class 0x00007fe084023350 done  
Event: 0.017 loading class 0x00007fe0840233b0  
Event: 0.017 loading class 0x00007fe0840233b0 done  
Event: 0.017 loading class 0x00007fe084023420  
Event: 0.018 loading class 0x00007fe084023420 done  
Event: 0.018 loading class 0x00007fe084024400  
Event: 0.018 loading class 0x00007fe084024400 done  
Event: 0.018 loading class 0x00007fe084023f30  
Event: 0.018 loading class 0x00007fe084023f30 done  
Event: 0.019 Thread 0x00007fe084009000 Thread added: 0x00007fe084009000

Dynamic libraries:

00400000-00409000 r-xp 00000000 ca:01 1185605	/usr/lib/jvm/java-6-openjdk-
amd64/jre/bin/java	
00608000-00609000 r--p 00008000 ca:01 1185605	/usr/lib/jvm/java-6-openjdk-
amd64/jre/bin/java	
00609000-0060a000 rw-p 00009000 ca:01 1185605	/usr/lib/jvm/java-6-openjdk-

amd64/jre/bin/java	
01334000-01355000 rw-p 00000000 00:00 0	[heap]
cc000000-cca70000 rw-p 00000000 00:00 0	
cca70000-d9e00000 rw-p 00000000 00:00 0	
d9e00000-db2e0000 rw-p 00000000 00:00 0	
db2e0000-f5a00000 rw-p 00000000 00:00 0	
f5a00000-f6ec0000 rw-p 00000000 00:00 0	
f6ec0000-1000000000 rw-p 00000000 00:00 0	
7fe07feff000-7fe07ff00000 ---p 00000000 00:00 0	
7fe07ff00000-7fe080000000 rw-p 00000000 00:00 0	
7fe080000000-7fe080197000 r--s 01a18000 ca:01 1058624	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/rt.jar	
7fe080197000-7fe0801a2000 rw-p 00000000 00:00 0	
7fe0801a2000-7fe0801eb000 rw-p 00000000 00:00 0	
7fe0801eb000-7fe0801f1000 rw-p 00000000 00:00 0	
7fe0801f1000-7fe08025a000 rw-p 00000000 00:00 0	
7fe08025a000-7fe080265000 rw-p 00000000 00:00 0	
7fe080265000-7fe080338000 rw-p 00000000 00:00 0	
7fe080338000-7fe080343000 rw-p 00000000 00:00 0	
7fe080343000-7fe08038b000 rw-p 00000000 00:00 0	
7fe08038b000-7fe08038c000 rw-p 00000000 00:00 0	
7fe08038c000-7fe0805fc000 rwxp 00000000 00:00 0	
7fe0805fc000-7fe08338c000 rw-p 00000000 00:00 0	
7fe08338c000-7fe083393000 r-xp 00000000 ca:01 1185638	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libzip.so	
7fe083393000-7fe083592000 ---p 00007000 ca:01 1185638	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libzip.so	
7fe083592000-7fe083593000 r--p 00006000 ca:01 1185638	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libzip.so	
7fe083593000-7fe083594000 rw-p 00007000 ca:01 1185638	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libzip.so	
7fe083594000-7fe0835a0000 r-xp 00000000 ca:01 268931	/lib/x86_64-linux-gnu/libnss_files-
2.15.so	
7fe0835a0000-7fe08379f000 ---p 0000c000 ca:01 268931	/lib/x86_64-linux-gnu/libnss_files-
2.15.so	
7fe08379f000-7fe0837a0000 r--p 0000b000 ca:01 268931	/lib/x86_64-linux-gnu/libnss_files-
2.15.so	
7fe0837a0000-7fe0837a1000 rw-p 0000c000 ca:01 268931	/lib/x86_64-linux-gnu/libnss_files-
2.15.so	
7fe0837a1000-7fe0837ab000 r-xp 00000000 ca:01 268938	/lib/x86_64-linux-gnu/libnss_nis-
2.15.so	
7fe0837ab000-7fe0839ab000 ---p 0000a000 ca:01 268938	/lib/x86_64-linux-gnu/libnss_nis-
2.15.so	
7fe0839ab000-7fe0839ac000 r--p 0000a000 ca:01 268938	/lib/x86_64-linux-gnu/libnss_nis-
2.15.so	
7fe0839ac000-7fe0839ad000 rw-p 0000b000 ca:01 268938	/lib/x86_64-linux-gnu/libnss_nis-
2.15.so	
7fe0839ad000-7fe0839c4000 r-xp 00000000 ca:01 268930	/lib/x86_64-linux-gnu/libnsl-2.15.so
7fe0839c4000-7fe083bc3000 ---p 00017000 ca:01 268930	/lib/x86_64-linux-gnu/libnsl-2.15.so
7fe083bc3000-7fe083bc4000 r--p 00016000 ca:01 268930	/lib/x86_64-linux-gnu/libnsl-2.15.so
7fe083bc4000-7fe083bc5000 rw-p 00017000 ca:01 268930	/lib/x86_64-linux-gnu/libnsl-2.15.so
7fe083bc5000-7fe083bc7000 rw-p 00000000 00:00 0	
7fe083bc7000-7fe083bcf000 r-xp 00000000 ca:01 268937	/lib/x86_64-linux-gnu/libnss_compat-
2.15.so	
7fe083bcf000-7fe083dce000 ---p 00008000 ca:01 268937	/lib/x86_64-linux-gnu/libnss_compat-
2.15.so	
7fe083dce000-7fe083dcf000 r--p 00007000 ca:01 268937	/lib/x86_64-linux-gnu/libnss_compat-
2.15.so	
7fe083dcf000-7fe083dd0000 rw-p 00008000 ca:01 268937	/lib/x86_64-linux-gnu/libnss_compat-
2.15.so	
7fe083dd0000-7fe083dfd000 r-xp 00000000 ca:01 1185656	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libjava.so	
7fe083dfd000-7fe083ffc000 ---p 0002d000 ca:01 1185656	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libjava.so	
7fe083ffc000-7fe083ffd000 r--p 0002c000 ca:01 1185656	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libjava.so	
7fe083ffd000-7fe084000000 rw-p 0002d000 ca:01 1185656	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libjava.so	
7fe084000000-7fe08408d000 rw-p 00000000 00:00 0	

7fe08408d000-7fe088000000	---p	00000000	00:00	0	
7fe08801f000-7fe08805d000	rw-p	00000000	00:00	0	
7fe08805d000-7fe088131000	rw-p	00000000	00:00	0	
7fe088131000-7fe08813f000	r-xp	00000000	ca:01	1185655	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libverify.so					
7fe08813f000-7fe08833e000	---p	0000e000	ca:01	1185655	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libverify.so					
7fe08833e000-7fe088340000	r--p	0000d000	ca:01	1185655	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libverify.so					
7fe088340000-7fe088341000	rw-p	0000f000	ca:01	1185655	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/libverify.so					
7fe088341000-7fe088348000	r-xp	00000000	ca:01	268928	/lib/x86_64-linux-gnu/librt-2.15.so
7fe088348000-7fe088547000	---p	00007000	ca:01	268928	/lib/x86_64-linux-gnu/librt-2.15.so
7fe088547000-7fe088548000	r--p	00006000	ca:01	268928	/lib/x86_64-linux-gnu/librt-2.15.so
7fe088548000-7fe088549000	rw-p	00007000	ca:01	268928	/lib/x86_64-linux-gnu/librt-2.15.so
7fe088549000-7fe08855e000	r-xp	00000000	ca:01	262198	/lib/x86_64-linux-gnu/libgcc_s.so.1
7fe08855e000-7fe08875d000	---p	00015000	ca:01	262198	/lib/x86_64-linux-gnu/libgcc_s.so.1
7fe08875d000-7fe08875e000	r--p	00014000	ca:01	262198	/lib/x86_64-linux-gnu/libgcc_s.so.1
7fe08875e000-7fe08875f000	rw-p	00015000	ca:01	262198	/lib/x86_64-linux-gnu/libgcc_s.so.1
7fe08875f000-7fe08885a000	r-xp	00000000	ca:01	268935	/lib/x86_64-linux-gnu/libm-2.15.so
7fe08885a000-7fe088a59000	---p	000fb000	ca:01	268935	/lib/x86_64-linux-gnu/libm-2.15.so
7fe088a59000-7fe088a5a000	r--p	000fa000	ca:01	268935	/lib/x86_64-linux-gnu/libm-2.15.so
7fe088a5a000-7fe088a5b000	rw-p	000fb000	ca:01	268935	/lib/x86_64-linux-gnu/libm-2.15.so
7fe088a5b000-7fe088b3d000	r-xp	00000000	ca:01	922699	/usr/lib/x86_64-linux-
gnu/libstdc++.so.6.0.16					
7fe088b3d000-7fe088d3c000	---p	000e2000	ca:01	922699	/usr/lib/x86_64-linux-
gnu/libstdc++.so.6.0.16					
7fe088d3c000-7fe088d44000	r--p	000e1000	ca:01	922699	/usr/lib/x86_64-linux-
gnu/libstdc++.so.6.0.16					
7fe088d44000-7fe088d46000	rw-p	000e9000	ca:01	922699	/usr/lib/x86_64-linux-
gnu/libstdc++.so.6.0.16					
7fe088d46000-7fe088d5b000	rw-p	00000000	00:00	0	
7fe088d5b000-7fe089732000	r-xp	00000000	ca:01	1185660	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/server/libjvm.so					
7fe089732000-7fe089931000	---p	009d7000	ca:01	1185660	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/server/libjvm.so					
7fe089931000-7fe0899b0000	r--p	009d6000	ca:01	1185660	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/server/libjvm.so					
7fe0899b0000-7fe0899d3000	rw-p	00a55000	ca:01	1185660	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/server/libjvm.so					
7fe0899d3000-7fe0899fd000	rw-p	00000000	00:00	0	
7fe0899fd000-7fe089a13000	r-xp	00000000	ca:01	262374	/lib/x86_64-linux-
gnu/libz.so.1.2.3.4					
7fe089a13000-7fe089c12000	---p	00016000	ca:01	262374	/lib/x86_64-linux-
gnu/libz.so.1.2.3.4					
7fe089c12000-7fe089c13000	r--p	00015000	ca:01	262374	/lib/x86_64-linux-
gnu/libz.so.1.2.3.4					
7fe089c13000-7fe089c14000	rw-p	00016000	ca:01	262374	/lib/x86_64-linux-
gnu/libz.so.1.2.3.4					
7fe089c14000-7fe089dc9000	r-xp	00000000	ca:01	268932	/lib/x86_64-linux-gnu/libc-2.15.so
7fe089dc9000-7fe089fc8000	---p	001b5000	ca:01	268932	/lib/x86_64-linux-gnu/libc-2.15.so
7fe089fc8000-7fe089fcc000	r--p	001b4000	ca:01	268932	/lib/x86_64-linux-gnu/libc-2.15.so
7fe089fcc000-7fe089fce000	rw-p	001b8000	ca:01	268932	/lib/x86_64-linux-gnu/libc-2.15.so
7fe089fce000-7fe089fd3000	rw-p	00000000	00:00	0	
7fe089fd3000-7fe089fd5000	r-xp	00000000	ca:01	268946	/lib/x86_64-linux-gnu/libdl-2.15.so
7fe089fd5000-7fe08ald5000	---p	00002000	ca:01	268946	/lib/x86_64-linux-gnu/libdl-2.15.so
7fe08ald5000-7fe08ald6000	r--p	00002000	ca:01	268946	/lib/x86_64-linux-gnu/libdl-2.15.so
7fe08ald6000-7fe08ald7000	rw-p	00003000	ca:01	268946	/lib/x86_64-linux-gnu/libdl-2.15.so
7fe08ald7000-7fe08aldb000	r-xp	00000000	ca:01	1185630	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/jli/libjli.so					
7fe08aldb000-7fe08a3da000	---p	00004000	ca:01	1185630	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/jli/libjli.so					
7fe08a3da000-7fe08a3db000	r--p	00003000	ca:01	1185630	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/jli/libjli.so					
7fe08a3db000-7fe08a3dc000	rw-p	00004000	ca:01	1185630	/usr/lib/jvm/java-6-openjdk-
amd64/jre/lib/amd64/jli/libjli.so					
7fe08a3dc000-7fe08a3f4000	r-xp	00000000	ca:01	268926	/lib/x86_64-linux-gnu/libpthread-
2.15.so					
7fe08a3f4000-7fe08a5f3000	---p	00018000	ca:01	268926	/lib/x86_64-linux-gnu/libpthread-

```
2. 15. so
7fe08a5f3000-7fe08a5f4000 r--p 00017000 ca:01 268926 /lib/x86_64-linux-gnu/libpthread-
2. 15. so
7fe08a5f4000-7fe08a5f5000 rw-p 00018000 ca:01 268926 /lib/x86_64-linux-gnu/libpthread-
2. 15. so
7fe08a5f5000-7fe08a5f9000 rw-p 00000000 00:00 0
7fe08a5f9000-7fe08a61b000 r-xp 00000000 ca:01 268925 /lib/x86_64-linux-gnu/ld-2.15.so
7fe08a61c000-7fe08a64e000 rw-p 00000000 00:00 0
7fe08a64e000-7fe08a704000 rw-p 00000000 00:00 0
7fe08a704000-7fe08a70c000 rw-s 00000000 ca:01 402365 /tmp/hsperfdata_root/3644
7fe08a70c000-7fe08a70f000 ---p 00000000 00:00 0
7fe08a70f000-7fe08a812000 rw-p 00000000 00:00 0
7fe08a812000-7fe08a818000 rw-p 00000000 00:00 0
7fe08a818000-7fe08a819000 r--p 00000000 00:00 0
7fe08a819000-7fe08a81b000 rw-p 00000000 00:00 0
7fe08a81b000-7fe08a81c000 r--p 00022000 ca:01 268925 /lib/x86_64-linux-gnu/ld-2.15.so
7fe08a81c000-7fe08a81e000 rw-p 00023000 ca:01 268925 /lib/x86_64-linux-gnu/ld-2.15.so
7fff1a7b8000-7fff1a7d9000 rw-p 00000000 00:00 0 [stack]
7fff1a7ff000-7fff1a800000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

#### VM Arguments:

java\_command: org.apache.catalina.util.ServerInfo  
Launcher Type: SUN\_STANDARD

#### Environment Variables:

PATH=/sbin:/usr/sbin:/bin:/usr/bin  
LD\_LIBRARY\_PATH=/usr/lib/jvm/java-6-openjdk-amd64/jre/lib/amd64/server:/usr/lib/jvm/java-6-openjdk-amd64/jre/lib/amd64:/usr/lib/jvm/java-6-openjdk-amd64/jre/./lib/amd64

#### Signal Handlers:

SIGSEGV: [libjvm.so+0x83dcf0], sa\_mask[0]=0x7ffbfefeff, sa\_flags=0x10000004  
SIGBUS: [libjvm.so+0x83dcf0], sa\_mask[0]=0x7ffbfefeff, sa\_flags=0x10000004  
SIGFPE: [libjvm.so+0x6de8f0], sa\_mask[0]=0x7ffbfefeff, sa\_flags=0x10000004  
SIGPIPE: [libjvm.so+0x6de8f0], sa\_mask[0]=0x7ffbfefeff, sa\_flags=0x10000004  
SIGXFSZ: [libjvm.so+0x6de8f0], sa\_mask[0]=0x7ffbfefeff, sa\_flags=0x10000004  
SIGILL: [libjvm.so+0x6de8f0], sa\_mask[0]=0x7ffbfefeff, sa\_flags=0x10000004  
SIGUSR1: SIG\_DFL, sa\_mask[0]=0x00000000, sa\_flags=0x00000000  
SIGUSR2: [libjvm.so+0x6ded30], sa\_mask[0]=0x00000000, sa\_flags=0x10000004  
SIGHUP: SIG\_DFL, sa\_mask[0]=0x00000000, sa\_flags=0x00000000  
SIGINT: SIG\_DFL, sa\_mask[0]=0x00000000, sa\_flags=0x00000000  
SIGTERM: SIG\_DFL, sa\_mask[0]=0x00000000, sa\_flags=0x00000000  
SIGQUIT: SIG\_DFL, sa\_mask[0]=0x00000000, sa\_flags=0x00000000

----- S Y S T E M -----

OS:Ubuntu 12.04 (precise)  
uname:Linux 3.2.0-65-generic #98-Ubuntu SMP Wed Jun 11 20:27:07 UTC 2014 x86\_64  
libc:glibc 2.15 NPTL 2.15  
rlimit: STACK 8192k, CORE 0k, NPROC 15879, NOFILE 1024, AS 1048576k  
load average:0.02 0.02 0.05

#### /proc/meminfo:

MemTotal: 2050108 kB  
MemFree: 899988 kB  
Buffers: 107976 kB  
Cached: 327296 kB  
SwapCached: 0 kB  
Active: 804996 kB  
Inactive: 272512 kB  
Active(anon): 642324 kB  
Inactive(anon): 228 kB  
Active(file): 162672 kB  
Inactive(file): 272284 kB  
Unevictable: 0 kB  
Mlocked: 0 kB  
SwapTotal: 0 kB  
SwapFree: 0 kB

```
Dirty:                80 kB
Writeback:            0 kB
AnonPages:            642396 kB
Mapped:               42920 kB
Shmem:                256 kB
Slab:                 38736 kB
SReclaimable:         28884 kB
SUnreclaim:           9852 kB
KernelStack:          1264 kB
PageTables:           3960 kB
NFS_Unstable:         0 kB
Bounce:               0 kB
WritebackTmp:         0 kB
CommitLimit:          1025052 kB
Committed_AS:         1143204 kB
VmallocTotal:         34359738367 kB
VmallocUsed:          11068 kB
VmallocChunk:         34359725180 kB
HardwareCorrupted:    0 kB
AnonHugePages:        0 kB
HugePages_Total:      0
HugePages_Free:       0
HugePages_Rsvd:       0
HugePages_Surp:       0
Hugepagesize:         2048 kB
DirectMap4k:          45056 kB
DirectMap2M:          2052096 kB
```

CPU:total 1 (32 cores per cpu, 2 threads per core) family 6 model 45 stepping 7, cmov, cx8, fxsr, mmx, sse, sse2, sse3, ssse3, sse4.1, sse4.2, popcnt, ht, tsc

```
/proc/cpuinfo:
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 45
model name : Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz
stepping : 7
microcode : 0x70d
cpu MHz : 2300.072
cache size : 15360 KB
physical id : 0
siblings : 1
core id : 0
cpu cores : 1
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat clflush mmx fxsr sse sse2 ht
syscall nx rdtscp lm constant_tsc up rep_good nopl pni ssse3 cx16 sse4_1 sse4_2 popcnt aes hypervisor lahf_lm
bogomips : 4600.14
clflush size : 64
cache alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:
```

Memory: 4k page, physical 2050108k(899988k free), swap 0k(0k free)

vm\_info: OpenJDK 64-Bit Server VM (23.25-b01) for linux-amd64 JRE (1.6.0\_33-b33), built on Oct 15 2014 12:27:16 by "build" with gcc 4.6.3

time: Sat Jan 31 21:50:03 2015  
elapsed time: 0 seconds

`-XX:OnOutOfMemoryError`

当内存溢出发生时，我们甚至可以执行一些指令，比如发个E-mail通知管理员或者执行一些清理工作。通过

`-XX:OnOutOfMemoryError` 这个参数我们可以做到这一点，这个参数可以接受一串指令和它们的参数。在这里，我们将不会深入它的细节，但我们提供了它的一个例子。在下面的例子中，当内存溢出错误发生的时候，我们会将堆内存快照写到/tmp/heapdump.hprof 文件并且在JVM的运行目录执行脚本cleanup.sh

```
java -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/heapdump.hprof -XX:OnOutOfMemoryError="sh
~/cleanup.sh" TestJava
```

`-XX:PermSize` and `-XX:MaxPermSize`

永久代在堆内存中是一块独立的区域，它包含了所有JVM加载的类的对象表示。为了成功运行应用程序，JVM会加载很多类（因为它们依赖于大量的第三方库，而这又依赖于更多的库并且需要从里面将类加载进来）这就需要增加永久代的大小。我们可以使用`-XX:PermSize` 和 `-XX:MaxPermSize` 来达到这个目的。其中`-XX:MaxPermSize` 用于设置永久代大小的最大值，`-XX:PermSize` 用于设置永久代初始大小。

这里设置的永久代大小并不会被包括在使用参数`-XX:MaxHeapSize` 设置的堆内存大小中。也就是说，通过`-XX:MaxPermSize`设置的永久代内存可能会需要由参数`-XX:MaxHeapSize` 设置的堆内存以外的一些堆内存。

`-XX:InitialCodeCacheSize` and `-XX:ReservedCodeCacheSize`

JVM往往被忽视的内存区域是“代码缓存”，它是用来存储已编译方法生成的本地代码。代码缓存确实很少引起性能问题，但是一旦发生其影响可能是毁灭性的。如果代码缓存被占满，JVM会打印出一条警告消息，并切换到interpreted-only 模式：JIT编译器被停用，字节码将不再会被编译成机器码。因此，应用程序将继续运行，但运行速度会降低一个数量级，直到有人注意到这个问题。就像其他内存区域一样，我们可以自定义代码缓存的大小。相关的参数是`-XX:InitialCodeCacheSize` 和 `-XX:ReservedCodeCacheSize`，它们的参数和上面介绍的参数一样，都是字节值。

`-XX:+UseCodeCacheFlushing`

如果代码缓存不断增长，例如，循环中不断new object，那么提高代码的缓存大小只会延缓其发生溢出。为了避免这种情况的发生，我们可以尝试一个有趣的新参数：当代码缓存被填满时让JVM放弃一些编译代码。通过使用`-XX:+UseCodeCacheFlushing` 这个参数，我们至少可以避免当代码缓存被填满的时候JVM切换到interpreted-only 模式。不过，我仍建议尽快解决代码缓存问题发生的根本原因，如找出内存泄漏并修复它。

`XX:NewSize` and `-XX:MaxNewSize`

就像可以通过参数(`-Xms` and `-Xmx`) 指定堆大小一样，可以通过参数指定新生代大小。设置 `XX:MaxNewSize` 参数时，应该考虑到新生代只是整个堆的一部分，新生代设置的越大，老年代区域就会减少。一般不允许新生代比老年代还大，因为要考虑GC时最坏情况，所有对象都晋升到老年代。(译者:会发生OOM错误) `-XX:MaxNewSize` 最大可以设置为`-Xmx/2`。

考虑性能，一般会通过参数 `-XX:NewSize` 设置新生代初始大小。如果知道新生代初始分配的对象大小(经过监控)，这样设置会有帮助，可以节省新生代自动扩展的消耗。

`XX:NewRatio`

可以设置新生代和老年代的相对大小。这种方式的优点是新生代大小会随着整个堆大小动态扩展。参数 `-XX:NewRatio` 设置老年代与新生代的比例。例如 `-XX:NewRatio=3` 指定老年代/新生代为3/1. 老年代占堆大小的 3/4，新生代占 1/4。

-XX:SurvivorRatio

参数 -XX:SurvivorRatio 与 -XX:NewRatio 类似，作用于新生代内部区域。-XX:SurvivorRatio 指定伊甸园区(Eden)与幸存者大小比例。例如，-XX:SurvivorRatio=10 表示伊甸园区(Eden)是 幸存者To 大小的10倍(也是幸存者From的10倍).所以,伊甸园区(Eden)占新生代大小的10/12, 幸存者From和幸存者To 每个占新生代的1/12 .注意,两个幸存者永远是一样大的..

设定幸存者大小有什么作用? 假设幸存者相对伊甸园区(Eden)太小, 相应新生对象的伊甸园区(Eden)永远很大空间, 我们当然希望, 如果这些对象在GC时全部被回收,伊甸园区(Eden)被清空,一切正常.然而,如果有一部分对象在GC中幸存下来, 幸存者只有很少空间容纳这些对象.结果大部分幸存对象在一次GC后, 就会被转移到老年代,这并不是我们希望的.考虑相反情况, 假设幸存者相对伊甸园区(Eden)太大,当然有足够的空间, 容纳GC后的幸存对象. 但是过小的伊甸园区(Eden),意味着空间将越快耗尽, 增加新生代GC次数, 这是不可接受的。

总之,我们希望最小化短命对象晋升到老年代的数量, 同时也希望最小化新生代GC 的次数和持续时间.我们需要找到针对当前应用的折中方案, 寻找适合方案的起点是 了解当前应用中对象的年龄分布情况。

XX:+PrintTenuringDistribution

参数 -XX:+PrintTenuringDistribution 指定JVM 在每次新生代GC时, 输出幸存者中对象的年龄分布。例如:

```
Desired survivor size 75497472 bytes, new threshold 15 (max 15)
- age 1: 19321624 bytes, 19321624 total
- age 2: 79376 bytes, 19401000 total
- age 3: 2904256 bytes, 22305256 total
```

第一行说明幸存者To大小为 75 MB. 也有关于老年代阈值(tenuring threshold)的信息, 老年代阈值, 意思是对象从新生代移动到老年代之前, 经过几次GC(即, 对象晋升前的最大年龄). 上例中,老年代阈值为15,最大也是15.

之后行表示, 对于小于老年代阈值的每一个对象年龄,本年龄中对象所占字节 (如果当前年龄没有对象,这一行会忽略). 上例中,一次GC 后幸存对象大约 19 MB, 两次GC 后幸存对象大约79 KB, 三次GC 后幸存对象大约 3 MB .每行结尾, 显示直到本年龄全部对象大小.所以,最后一行的 total 表示幸存者To 总共被占用22 MB . 幸存者To 总大小为 75 MB ,当前老年代阈值为15, 可以断定在本次GC中, 没有对象会移动到老年代。现在假设下一次GC 输出为：

```
Desired survivor size 75497472 bytes, new threshold 2 (max 15)
- age 1: 68407384 bytes, 68407384 total
- age 2: 12494576 bytes, 80901960 total
- age 3: 79376 bytes, 80981336 total
- age 4: 2904256 bytes, 83885592 total
```

对比前一次老年代分布。明显的,年龄2和年龄3 的对象还保持在幸存者中, 因为我们看到年龄3和4的对象大小与前一次年龄2和3 的相同。同时发现幸存者中,有一部分对象已经被回收,因为本次年龄2的对象大小为 12MB , 而前一次年龄1的对象大小为 19 MB。最后可以看到最近的GC中, 有68 MB 新对象, 从伊甸园区移动到幸存者。

注意,本次GC 幸存者占用总大小 84 MB -大于75 MB. 结果,JVM 把老年代阈值从15降低到2, 在下次GC时, 一部分对象会强制离开幸存者, 这些对象可能会被回收(如果他们刚好死亡)或移动到老年代。

-XX:InitialTenuringThreshold, -XX:MaxTenuringThreshold and -XX:TargetSurvivorRatio

参数 -XX:+PrintTenuringDistribution 输出中的部分值可以通过其它参数控制。通过 -XX:InitialTenuringThreshold 和 -XX:MaxTenuringThreshold 可以设定老年代阈值的初始值和最大值。另外,可以通过参数 -XX:TargetSurvivorRatio 设定幸存区的目标使用率.例如, -XX:MaxTenuringThreshold=10 -XX:TargetSurvivorRatio=90 设定老年代阈值的上限为10,幸存者空间目标使用率为90%。

有多种方式,设置新生代行为, 没有通用准则。我们必须清楚以下2中情况：

1 如果从年龄分布中发现, 有很多对象的年龄持续增长, 在到达老年代阈值之前。这表示 -XX:MaxTenuringThreshold 设置过大



2 如果 `-XX:MaxTenuringThreshold` 的值大于1，但是很多对象年龄从未大于1.应该看下幸存区的目标使用率。如果幸存区使用率从未到达，这表示对象都被GC回收，这正是我们想要的。如果幸存区使用率经常达到，有些年龄超过1的对象被移动到老年代中。这种情况，可以尝试调整幸存区大小或目标使用率。

`-XX:+NeverTenure` and `-XX:+AlwaysTenure`

最后,我们介绍2个颇为少见的参数,对应2种极端的新生代GC情况.设置参数 `-XX:+NeverTenure`，对象永远不会晋升到老年代.当我们确定不需要老年代时，可以这样设置。这样设置风险很大,并且会浪费至少一半的堆内存。相反设置参数 `-XX:+AlwaysTenure`，表示没有幸存区,所有对象在第一次GC时，会晋升到老年代。

在实践中我们发现对于大多数的应用领域，评估一个垃圾收集(GC)算法如何根据如下两个标准：

吞吐量越高算法越好

暂停时间越短算法越好

首先让我们来明确垃圾收集(GC)中的两个术语:吞吐量(throughput)和暂停时间(pause times)。JVM在专门的线程(GC threads)中执行GC。只要GC线程是活动的，它们将与应用程序线程(application threads)争用当前可用CPU的时钟周期。简单点来说，吞吐量是指应用程序线程用时占程序总用时的比例。例如，吞吐量99/100意味着100秒的程序执行时间应用程序线程运行了99秒，而在这一时间段内GC线程只运行了1秒。

术语“暂停时间”是指一个时间段内应用程序线程让与GC线程执行而完全暂停。例如，GC期间100毫秒的暂停时间意味着在这100毫秒期间内没有应用程序线程是活动的。如果说一个正在运行的应用程序有100毫秒的“平均暂停时间”，那么就是说该应用程序所有的暂停时间平均长度为100毫秒。同样，100毫秒的“最大暂停时间”是指该应用程序所有的暂停时间最大不超过100毫秒。

吞吐量 VS 暂停时间

高吞吐量最好因为这会让应用程序的最终用户感觉只有应用程序线程在做“生产性”工作。直觉上，吞吐量越高程序运行越快。低暂停时间最好因为从最终用户的角度来看不管是GC还是其他原因导致一个应用被挂起始终是不好的。这取决于应用程序的类型，有时候甚至短暂的200毫秒暂停都可能打断终端用户体验。因此，具有低的最大暂停时间是非常重要的，特别是对于一个交互式应用程序。

不幸的是“高吞吐量”和“低暂停时间”是一对相互竞争的目标（矛盾）。这样想想看，为了清晰起见简化一下：GC需要一定的前提条件以便安全地运行。例如，必须保证应用程序线程在GC线程试图确定哪些对象仍然被引用和哪些没有被引用的时候不修改对象的状态。为此，应用程序在GC期间必须停止(或者仅在GC的特定阶段，这取决于所使用的算法)。然而这会增加额外的线程调度开销：直接开销是上下文切换，间接开销是因为缓存的影响。加上JVM内部安全措施的开销，这意味着GC及随之而来的不可忽略的开销，将增加GC线程执行实际工作的时间。因此我们可以通过尽可能少运行GC来最大化吞吐量，例如，只有在不可避免的时候进行GC，来节省所有与它相关的开销。

然而，仅仅偶尔运行GC意味着每当GC运行时将有許多工作要做，因为在此期间积累在堆中的对象数量很高。单个GC需要花更多时间来完成，从而导致更高的平均和最大暂停时间。因此，考虑到低暂停时间，最好频繁地运行GC以便更快速地完成。这反过来又增加了开销并导致吞吐量下降，我们又回到了起点。

综上所述，在设计（或使用）GC算法时，我们必须确定我们的目标：一个GC算法只可能针对两个目标之一（即只专注于最大吞吐量或最小暂停时间），或尝试找到一个二者的折衷。

HotSpot虚拟机上的垃圾收集

该系列的第五部分我们已经讨论过年轻代的垃圾收集器。对于老年代，HotSpot虚拟机提供两类垃圾收集算法(除了新的G1垃圾收集算法)，第一类算法试图最大限度地提高吞吐量，而第二类算法试图最小化暂停时间。今天我们的重点是第一类，“面向吞吐量”的垃圾收集算法。

我们希望把重点放在JVM配置参数上，所以我只会简要概述HotSpot提供的面向吞吐量(throughput-oriented)垃圾收集算法。当年老代中由于缺乏空间导致对象分配失败时会触发垃圾收集器(事实上，“分配”的通常是指从年轻代提升到年老代的对象)。从所谓的“GC根”(GC roots)开始，搜索堆中的可达对象并将其标记为活着的，之后，垃圾收集器将活着的对象移到年老代的一块无碎片(non-fragmented)内存块中，并标记剩余的内存空间是空闲的。也就是说，我们不像复制策略那样移到一个不同的堆区域，像年轻代垃圾收集算法所做的那样。相反地，我们把所有的对象放在一个堆区域中，从而对该堆区域进行碎片整理。垃圾收集器使用一个或多个线程来执行垃圾收集。当使用多个线程时，算法的不同步骤被分解，使得每个收集线程大多数时候工作在自己的区域而不干扰其他线程。在垃圾收集期间，所有的应用程序线程暂停，只有垃圾收集完成之后才会重新开始。现在让我们来看看跟面向吞吐量垃圾收集算法有关的重要JVM配置参数。

-XX:+UseSerialGC 我们使用该标志来激活串行垃圾收集器，例如单线程面向吞吐量垃圾收集器。无论年轻代还是年老代都将只有一个线程执行垃圾收集。该标志被推荐用于只有单个可用处理器核心的JVM。在这种情况下，使用多个垃圾收集线程甚至会适得其反，因为这些线程将争用CPU资源，造成同步开销，却从未真正并行运行。-XX:+UseParallelGC

有了这个标志，我们告诉JVM使用多线程并行执行年轻代垃圾收集。在我看来，Java 6中不应该使用该标志因为

-XX:+UseParallelOldGC显然更合适。需要注意的是Java 7中该情况改变了一点(详见本概述)，就是-XX:+UseParallelGC能达到-XX:+UseParallelOldGC一样的效果。-XX:+UseParallelOldGC

该标志的命名有点不巧，因为“老”听起来像“过时”。然而，“老”实际上是指年老代，这也解释了为什么

-XX:+UseParallelOldGC要优于-XX:+UseParallelGC：除了激活年轻代并行垃圾收集，也激活了年老代并行垃圾收集。当期望高吞吐量，并且JVM有两个或更多可用处理器核心时，我建议使用该标志。

作为旁注，HotSpot的并行面向吞吐量垃圾收集算法通常称为“吞吐量收集器”，因为它们旨在通过并行执行来提高吞吐量。

-XX:ParallelGCThreads 通过-XX:ParallelGCThreads=我们可以指定并行垃圾收集的线程数量。例如，-XX:ParallelGCThreads=6表示每次并行垃圾收集将有6个线程执行。如果不明确设置该标志，虚拟机将使用基于可用(虚拟)处理器数量计算的默认值。决定因素是由Java Runtime.availableProcessors()方法的返回值N，如果 $N \leq 8$ ，并行垃圾收集器将使用N个垃圾收集线程，如果 $N > 8$ 个可用处理器，垃圾收集线程数量应为 $3 + 5N/8$ 。

当JVM独占地使用系统和处理器时使用默认设置更有意义。但是，如果有多个JVM(或其他耗CPU的系统)在同一台机器上运行，我们应该使用-XX:ParallelGCThreads来减少垃圾收集线程数到一个适当的值。例如，如果4个以服务器方式运行的JVM同时跑在一个具有16核处理器的机器上，设置-XX:ParallelGCThreads=4是明智的，它能使不同JVM的垃圾收集器不会相互干扰。

-XX:-UseAdaptiveSizePolicy 吞吐量垃圾收集器提供了一个有趣的(但常见，至少在现代JVM上)机制以提高垃圾收集配置的用户友好性。这种机制被看做是HotSpot在Java 5中引入的“人体工程学”概念的一部分。通过人体工程学，垃圾收集器能将堆大小动态变动像GC设置一样应用到不同的堆区域，只要有证据表明这些变动将能提高GC性能。“提高GC性能”的确切含义可以由用户通过-XX:GCTimeRatio和-XX:MaxGCPauseMillis(见下文)标记来指定。重要的是要知道人体工程学是默认激活的。这很好，因为自适应行为是JVM最大优势之一。不过，有时我们需要非常清楚对于特定应用什么样的设置是最合适的，在这些情况下，我们可能不希望JVM混乱我们的设置。每当我们发现处于这种情况时，我们可以考虑通过-XX:-UseAdaptiveSizePolicy停用一些人体工程学。

-XX:GCTimeRatio 通过-XX:GCTimeRatio=我们告诉JVM吞吐量要达到的目标值。更准确地说，-XX:GCTimeRatio=N指定目标应用程序线程的执行时间(与总的程序执行时间)达到 $N/(N+1)$ 的目标比值。例如，通过-XX:GCTimeRatio=9我们要求应用程序线程在整个执行时间中至少9/10是活动的(因此，GC线程占用其余1/10)。基于运行时的测量，JVM将会尝试修改堆和GC设置以期达到目标吞吐量。-XX:GCTimeRatio的默认值是99，也就是说，应用程序线程应该运行至少99%的总执行时间。

-XX:MaxGCPauseMillis 通过-XX:GCTimeRatio=告诉JVM最大暂停时间的目标值(以毫秒为单位)。在运行时，吞吐量收集器计算在暂停期间观察到的统计数据(加权平均和标准偏差)。如果统计表明正在经历的暂停其时间存在超过目标值的风险时，JVM会修改堆和GC设置以降低它们。需要注意的是，年轻代和年老代垃圾收集的统计数据是分开计算的，还要注意，默认情况下，最

大暂停时间没有被设置。

如果最大暂停时间和最小吞吐量同时设置了目标值，实现最大暂停时间目标具有更高的优先级。当然，无法保证JVM将一定能达到任一目标，即使它会努力去做。最后，一切都取决于手头应用程序的行为。

当设置最大暂停时间目标时，我们应注意不要选择太小的值。正如我们现在所知道的，为了保持低暂停时间，JVM需要增加GC次数，那样可能会严重影响可达到的吞吐量。这就是为什么对于要求低暂停时间作为主要目标的应用程序(大多数是Web应用程序)，我会建议不要使用吞吐量收集器，而是选择CMS收集器。

HotSpot JVM的并发标记清理收集器(CMS收集器)的主要目标就是：低应用停顿时间。该目标对于大多数交互式应用很重要，比如web应用。在我们看一下有关JVM的参数之前,让我们简要回顾CMS收集器的操作和使用它时可能出现的主要挑战。

就像吞吐量收集器(参见本系列的第6部分),CMS收集器处理老年代的对象,然而其操作要复杂得多。吞吐量收集器总是暂停应用程序线程，并且可能是相当长的一段时间，然而这能够使该算法安全地忽略应用程序。相比之下，CMS收集器被设计成在大多数时间能与应用程序线程并行执行，仅仅会有一点(短暂的)停顿时间。GC与应用程序并行的缺点就是，可能会出现各种同步和数据不一致的问题。为了实现安全且正确的并发执行，CMS收集器的GC周期被分为了好几个连续的阶段。

## CMS收集器的过程

CMS收集器的GC周期由6个阶段组成。其中4个阶段(名字以Concurrent开始的)与实际的应用程序是并发执行的，而其他2个阶段需要暂停应用程序线程。

初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。

并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。

重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。

并发清理：所有不再被应用的对象将从堆里清除掉。

并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

一个常见的误解是,CMS收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使“stop-the-world”阶段相对于并发阶段的时间很短。

应该指出，尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻代的垃圾回收时间通常是相当短的。

### 挑战

当我们在真实的应用中使用CMS收集器时，我们会面临两个主要的挑战，可能需要进行调优：

#### 堆碎片

##### 对象分配率高

堆碎片是有可能的，不像吞吐量收集器，CMS收集器并没有任何碎片整理的机制。因此，应用程序有可能出现这样的情形，即使总的堆大小远没有耗尽，但却不能分配对象——仅仅是因为没有足够连续的空间完全容纳对象。当这种事发生后，并发算法不会帮上任何忙，因此，万不得已JVM会触发Full GC。回想一下，Full GC将运行吞吐量收集器的算法，从而解决碎片问题——但却暂停了应用程序线程。因此尽管CMS收集器带来完全的并发性，但仍然有可能发生长时间的“stop-the-world”的风险。这是“设计”，而不能避免的——我们只能通过调优收集器来它的可能性。想要100%保证避免“stop-the-world”，对于交互式应用是有问题的。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。从某

某种程度上说，老年代将没有足够的可用空间来容纳一个从年轻代提升过来的对象。这种情况被称为“并发模式失败”，并且JVM会执行堆碎片整理：触发Full GC。

当这些情形之一出现在实践中时(经常会出现生产系统中)，经常被证实是老年代有大量不必要的对象。一个可行的办法就是增加年轻代的堆大小，以防止年轻代短生命的对象提前进入老年代。另一个办法就似乎利用分析器，快照运行系统的堆转储，并且分析过度的对象分配，找出这些对象，最终减少这些对象的申请。

下面我看看大多数与CMS收集器调优相关的JVM标志参数。

-XX : +UseConcMarkSweepGC

该标志首先是激活CMS收集器。默认HotSpot JVM使用的是并行收集器。

-XX : UseParNewGC

当使用CMS收集器时，该标志激活年轻代使用多线程并行执行垃圾回收。这令人很惊讶，我们不能简单在并行收集器中重用

-XX : UserParNewGC标志，因为概念上年轻代用的算法是一样的。然而，对于CMS收集器，年轻代GC算法和老年代GC算法是不同的，因此年轻代GC有两种不同的实现，并且是两个不同的标志。

注意最新的JVM版本，当使用-XX : +UseConcMarkSweepGC时，-XX : UseParNewGC会自动开启。因此，如果年轻代的并行GC不想开启，可以通过设置-XX : -UseParNewGC来关掉。

-XX : +CMSConcurrentMTEnabled

当该标志被启用时，并发的CMS阶段将以多线程执行(因此，多个GC线程会与所有的应用程序线程并行工作)。该标志已经默认开启，如果顺序执行更好，这取决于所使用的硬件，多线程执行可以通过-XX : -CMSConcurrentMTEnabled禁用。

-XX : ConcGCThreads

标志-XX : ConcGCThreads=(早期JVM版本也叫-XX:ParallelCMSThreads)定义并发CMS过程运行时的线程数。比如value=4意味着CMS周期的所有阶段都以4个线程来执行。尽管更多的线程会加快并发CMS过程，但其也会带来额外的同步开销。因此，对于特定的应用程序，应该通过测试来判断增加CMS线程数是否真的能够带来性能的提升。

如果该标志未设置，JVM会根据并行收集器中的-XX : ParallelGCThreads参数的值来计算出默认的并行CMS线程数。该公式是 $\text{ConcGCThreads} = (\text{ParallelGCThreads} + 3) / 4$ 。因此，对于CMS收集器，-XX:ParallelGCThreads标志不仅影响“stop-the-world”垃圾收集阶段，还影响并发阶段。

总之，有不少方法可以配置CMS收集器的多线程执行。正是由于这个原因,建议第一次运行CMS收集器时使用其默认设置,然后如果需要调优再进行测试。只有在生产系统中测量(或类生产测试系统)发现应用程序的暂停时间的目标没有达到,就可以通过这些标志应该进行GC调优。

-XX:CMSInitiatingOccupancyFraction 当堆满之后，并行收集器便开始进行垃圾收集，例如，当没有足够的空间来容纳新分配或提升的对象。对于CMS收集器，长时间等待是不可取的，因为在并发垃圾收集期间应用持续在运行(并且分配对象)。因此，为了在应用程序使用完内存之前完成垃圾收集周期，CMS收集器要比并行收集器更先启动。因为不同的应用会有不同对象分配模式，JVM会收集实际的对象分配(和释放)的运行数据，并且分析这些数据，来决定什么时候启动一次CMS垃圾收集周期。为了引导这一过程，JVM会在一开始执行CMS周期前作一些线索查找。该线索由-XX:CMSInitiatingOccupancyFraction=来设置，该值代表老年代堆空间的使用率。比如，value=75意味着第一次CMS垃圾收集会在老年代被占用75%时被触发。通常CMSInitiatingOccupancyFraction的默认值为68(之前很长时间的经历来决定的)。

-XX : +UseCMSInitiatingOccupancyOnly

我们用-XX+UseCMSInitiatingOccupancyOnly标志来命令JVM不基于运行时收集的数据来启动CMS垃圾收集周期。而是，当该

标志被开启时，JVM通过CMSInitiatingOccupancyFraction的值进行每一次CMS收集，而不仅仅是第一次。然而，请记住大多数情况下，JVM比我们自己能作出更好的垃圾收集决策。因此，只有当我们充足的理由(比如测试)并且对应用程序产生的对象的生命周期有深刻的认知时，才应该使用该标志。

-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。

-XX:+CMSClassUnloadingEnabled 相对于并行收集器，CMS收集器默认不会对永久代进行垃圾回收。如果希望对永久代进行垃圾回收，可用设置标志-XX:+CMSClassUnloadingEnabled。在早期JVM版本中，要求设置额外的标志

-XX:+CMSPermGenSweepingEnabled。注意，即使没有设置这个标志，一旦永久代耗尽空间也会尝试进行垃圾回收，但是收集不会是并行的，而再一次进行Full GC。 -XX:+CMSIncrementalMode

该标志将开启CMS收集器的增量模式。增量模式经常暂停CMS过程，以便对应用程序线程作出完全的让步。因此，收集器将花更长的时间完成整个收集周期。因此，只有通过测试后发现正常CMS周期对应用程序线程干扰太大时，才应该使用增量模式。由于现代服务器有足够的处理器来适应并发的垃圾收集，所以这种情况发生得很少。

-XX:+ExplicitGCInvokesConcurrent and -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

如今,被广泛接受的最佳实践是避免显式地调用GC(所谓的“系统GC”),即在应用程序中调用system.gc()。然而，这个建议是不管使用的GC算法的，值得一提的是，当使用CMS收集器时，系统GC将是一件很不幸的事，因为它默认会触发一次Full GC。幸运的是，有一种方式可以改变默认设置。标志-XX:+ExplicitGCInvokesConcurrent命令JVM无论什么时候调用系统GC，都执行CMS GC，而不是Full GC。第二个标志-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses保证当有系统GC调用时，永久代也被包括进CMS垃圾回收的范围内。因此，通过使用这些标志，我们可以防止出现意料之外的“stop-the-world”的系统GC。

-XX:+DisableExplicitGC 然而在这个问题上...这是一个很好提到-XX:+DisableExplicitGC标志的机会，该标志将告诉JVM完全忽略系统的GC调用(不管使用的收集器是什么类型)。对于我而言，该标志属于默认的标志集合中，可以安全地定义在每个JVM上运行，而不需要进一步思考。

## GC日志：

垃圾收集（GC）日志的JVM参数。GC日志是一个很重要的工具，它准确记录了每一次的GC的执行时间和执行结果，通过分析GC日志可以优化堆设置和GC设置，或者改进应用程序的对象分配模式。

-XX:+PrintGC 参数-XX:+PrintGC（或者-verbose:gc）开启了简单GC日志模式，为每一次新生代（young generation）的GC和每一次的Full GC打印一行信息。下面举例说明：

```
1 [GC 246656K->243120K(376320K), 0.0929090 secs]
```

```
2 [Full GC 243120K->241951K(629760K), 1.5589690 secs]
```

每行开始首先是GC的类型（可以是“GC”或者“Full GC”），然后是在GC之前和GC之后已使用的堆空间，再然后是当前的堆容量，最后是GC持续的时间（以秒计）。

第一行的意思就是GC将已使用的堆空间从246656K减少到243120K，当前的堆容量（译者注：GC发生时）是376320K，GC持续的时间是0.0929090秒。

简单模式的GC日志格式是与GC算法无关的，日志也没有提供太多的信息。在上面的例子中，我们甚至无法从日志中判断是否GC将一些对象从young generation移到了old generation。所以详细模式的GC日志更有用一些。

-XX:PrintGCDetails 如果不是使用-XX:+PrintGC，而是-XX:PrintGCDetails，就开启了详细GC日志模式。在这种模式下，日志格式和所使用的GC算法有关。我们首先看一下使用Throughput垃圾收集器在young generation中生成的日志。为了便于阅读这里将一行日志分为多行并使用缩进。

```
1 [GC
2   [PSYoungGen: 142816K->10752K(142848K)] 246648K->243136K(375296K), 0.0935090 secs
3 ]
4 [Times: user=0.55 sys=0.10, real=0.09 secs]
```

我们可以很容易发现：这是一次在young generation中的GC，它将已使用的堆空间从246648K减少到了243136K，用时0.0935090秒。此外我们还可以得到更多的信息：所使用的垃圾收集器（即PSYoungGen）、young generation的大小和使用情况（在这个例子中“PSYoungGen”垃圾收集器将young generation所使用的堆空间从142816K减少到10752K）。

既然我们已经知道了young generation的大小，所以很容易判定发生了GC，因为young generation无法分配更多的对象空间：已经使用了142848K中的142816K。我们可以进一步得出结论，多数从young generation移除的对象仍然在堆空间中，只是被移到了old generation：通过对比绿色的和蓝色的部分可以发现即使young generation几乎被完全清空（从142816K减少到10752K），但是所占用的堆空间仍然基本相同（从246648K到243136K）。

详细日志的“Times”部分包含了GC所使用的CPU时间信息，分别为操作系统的用户空间和系统空间所使用的时间。同时，它显示了GC运行的“真实”时间（0.09秒是0.0929090秒的近似值）。如果CPU时间（译者注：0.55秒+0.10秒）明显多于“真实”时间（译者注：0.09秒），我们可以得出结论：GC使用了多线程运行。这样的话CPU时间就是所有GC线程所花费的CPU时间的总和。实际上我们的例子中的垃圾收集器使用了8个线程。

接下来看一下Full GC的输出日志

```
1 [Full GC
2   [PSYoungGen: 10752K->9707K(142848K)]
3   [ParOldGen: 232384K->232244K(485888K)] 243136K->241951K(628736K)
4   [PSPermGen: 3162K->3161K(21504K)], 1.5265450 secs
5 ]
```

除了关于young generation的详细信息，日志也提供了old generation和permanent generation的详细信息。对于这三个generations，一样也可以看到所使用的垃圾收集器、堆空间的大小、GC前后的堆使用情况。需要注意的是显示堆空间的大小等于young generation和old generation各自堆空间的和。以上面为例，堆空间总共占用了241951K，其中9707K在young generation，232244K在old generation。Full GC持续了大约1.53秒，用户空间的CPU执行时间为10.96秒，说明GC使用了多线程（和之前一样8个线程）。

对不同generation详细的日志可以让我们分析GC的原因，如果某个generation的日志显示在GC之前，堆空间几乎被占满，那么很有可能就是这个generation触发了GC。但是在上面的例子中，三个generation中的任何一个都不是这样的，在这种情况下是什么原因触发了GC呢。对于Throughput垃圾收集器，在某一个generation被过度使用之前，GC ergonomics（参考本系列第6节）决定要启动GC。

Full GC也可以通过显式的请求而触发，可以通过应用程序，或者是一个外部的JVM接口。这样触发的GC可以很容易在日志里分辨出来，因为输出的日志是以“Full GC(System)”开头的，而不是“Full GC”。

对于Serial垃圾收集器，详细的GC日志和Throughput垃圾收集器是非常相似的。唯一的区别是不同的generation日志可能使用了不同的GC算法（例如：old generation的日志可能以Tenured开头，而不是ParOldGen）。使用垃圾收集器作为一行日志的开头可以方便我们从日志就判断出JVM的GC设置。

对于CMS垃圾收集器，young generation的详细日志也和Throughput垃圾收集器非常相似，但是old generation的日志却不是这样。对于CMS垃圾收集器，在old generation中的GC是在不同的时间片内与应用程序同时运行的。GC日志自然也和Full GC的日志不同。而且在不同时间片的日志夹杂着在此期间young generation的GC日志。但是了解了上面介绍的GC日志的基本元素，也不难理解在不同时间片内的日志。只是在解释GC运行时间时要特别注意，由于大多数时间片内的GC都是和应用程序同时运行的，所以和那种独占式的GC相比，GC的持续时间更长一些并不说明一定有问题。

正如我们在第7节中所了解的，即使CMS垃圾收集器没有完成一个CMS周期，Full GC也可能会发生。如果发生了GC，在日志中会包含触发Full GC的原因，例如众所周知的“concurrent mode failure”。

为了避免过于冗长，我这里就不详细说明CMS垃圾收集器的日志了。另外，CMS垃圾收集器的作者做了详细的说明（在这里），强烈建议阅读。

-XX:+PrintGCTimeStamps和-XX:+PrintGCDateStamps

使用-XX:+PrintGCTimeStamps可以将时间和日期也加到GC日志中。表示自JVM启动至今的时间戳会被添加到每一行中。例子如下：

```
1 0.185: [GC 66048K->53077K(251392K), 0.0977580 secs]
2 0.323: [GC 119125K->114661K(317440K), 0.1448850 secs]
3 0.603: [GC 246757K->243133K(375296K), 0.2860800 secs]
```

如果指定了-XX:+PrintGCDateStamps，每一行就添加上了绝对的日期和时间。

```
1 2014-01-03T12:08:38.102-0100: [GC 66048K->53077K(251392K), 0.0959470 secs]
2 2014-01-03T12:08:38.239-0100: [GC 119125K->114661K(317440K), 0.1421720 secs]
3 2014-01-03T12:08:38.513-0100: [GC 246757K->243133K(375296K), 0.2761000 secs]
```

如果需要也可以同时使用两个参数。推荐同时使用这两个参数，因为这样在关联不同来源的GC日志时很有帮助。

-Xloggc

缺省的GC日志时输出到终端的，使用-Xloggc:也可以输出到指定的文件。需要注意这个参数隐式的设置了参数-XX:+PrintGC和-XX:+PrintGCTimeStamps，但为了以防在新版本的JVM中有任何变化，我仍建议显示的设置这些参数。

可管理的JVM参数

一个常常被讨论的问题是在生产环境中GC日志是否应该开启。因为它所产生的开销通常都非常有限，因此我的答案是需要开启。

但并不一定在启动JVM时就必须指定GC日志参数。

HotSpot JVM有一类特别的参数叫做可管理的参数。对于这些参数，可以在运行时修改他们的值。我们这里所讨论的所有参数以及以“PrintGC”开头的参数都是可管理的参数。这样在任何时候我们都可以开启或是关闭GC日志。比如我们可以使用JDK自带的jinfo工具来设置这些参数，或者是通过JMX客户端调用HotSpotDiagnostic MBean的setVMOption方法来设置这些参数。

在CMS GC 时,使用参数-XX:+PrintGCDetails 和 -XX:+PrintGCTimeStamps 会输出很多日志信息，了解这些信息可以帮我们更好的调整参数，以获得更高的性能。

我们来看下在JDK1.4.2\_10 中CMS GC日志示例：

39.910: [GC 39.910: [ParNew: 261760K->0K(261952K), 0.2314667 secs] 262017K->26386K(1048384K), 0.2318679 secs]  
新生代使用 (ParNew 并行)回收器。新生代容量为261952K，GC回收后占用从261760K降到0,耗时0.2314667秒。(译注：262017K->26386K(1048384K), 0.2318679 secs 表示整个堆占用从262017K 降至26386K,费时0.2318679)

40.146: [GC [1 CMS-initial-mark: 26386K(786432K)] 26404K(1048384K), 0.0074495 secs]

开始使用CMS回收器进行老年代回收。初始标记(CMS-initial-mark)阶段,这个阶段标记由根可以直接到达的对象，标记期间整个应用线程会暂停。

老年代容量为786432K,CMS 回收器在空间占用达到 26386K 时被触发

40.154: [CMS-concurrent-mark-start]

开始并发标记(concurrent-mark-start) 阶段，在第一个阶段被暂停的线程重新开始运行，由前阶段标记过的对象出发，所有可到达的对象都在本阶段中标记。

40.683: [CMS-concurrent-mark: 0.521/0.529 secs]

并发标记阶段结束，占用 0.521秒CPU时间, 0.529秒墙钟时间(也包含线程让出CPU给其他线程执行的时间)

40.683: [CMS-concurrent-preclean-start]

开始预清理阶段

预清理也是一个并发执行的阶段。在本阶段，会查找前一阶段执行过程中,从新生代晋升或新分配或被更新的对象。通过并发地重新扫描这些对象，预清理阶段可以减少下一个stop-the-world 重新标记阶段的工作量。

40.701: [CMS-concurrent-preclean: 0.017/0.018 secs]

预清理阶段费时 0.017秒CPU时间，0.018秒墙钟时间。

40.704: [GC40.704: [Rescan (parallel), 0.1790103 secs]40.883: [weak refs processing, 0.0100966 secs] [1 CMS-remark: 26386K(786432K)] 52644K(1048384K), 0.1897792 secs]

Stop-the-world 阶段,从根及被其引用对象开始，重新扫描 CMS 堆中残留的更新过的对象。这里重新扫描费时0.1790103秒，处理弱引用对象费时0.0100966秒，本阶段费时0.1897792 秒。

40.894: [CMS-concurrent-sweep-start]

开始并发清理阶段，在清理阶段，应用线程还在运行。

41.020: [CMS-concurrent-sweep: 0.126/0.126 secs]

并发清理阶段费时0.126秒

41.020: [CMS-concurrent-reset-start]

开始并发重置

41.147: [CMS-concurrent-reset: 0.127/0.127 secs]



在本阶段，重新初始化CMS内部数据结构，以备下一轮 GC 使用。本阶段费时0.127秒

这是CMS正常运行周期打印的日志，现在让我们一起看一下其他的CMS日志记录：

```
197.976: [GC 197.976: [ParNew: 260872K->260872K(261952K), 0.0000688 secs]197.976: [CMS197.981:
[CMS-concurrent-sweep: 0.516/0.531 secs]
(concurrent mode failure): 402978K->248977K(786432K), 2.3728734 secs] 663850K->248977K(1048384K), 2.3733725
secs]
```

这段信息显示ParNew 收集器被请求进行新生代的回收，但收集器并没有尝试回收，因为它 预计在最糟糕的情况下，CMS 老年代中没有足够的空间容纳新生代的幸存对象。我们把这个失败称之为“完全晋升担保失败”。

因为这样，并发模式的 CMS 被中断同并且在 197.981秒时，Full GC被启动。这次Full GC，采用标记-清除-整理算法，会发生 stop-the-world，费时2.3733725秒。CMS 老年代占用从 402978K 降到248977K。

避免并发模式失败，通过增加老年代空间大小或者设置参数 CMSInitiatingOccupancyFraction 同时设置 UseCMSInitiatingOccupancyOnly为true。参数 CMSInitiatingOccupancyFraction 的值必须谨慎选择，设置过低会造成频繁发生 CMS 回收。

有时我们发现，当日志中出现晋升失败时，老年代还有足够的空间。这是因为“碎片”，老年代中的可用空间并不连续，而从新生代晋升上来的对象，需要一块连续的可用空间。CMS 收集器是一种非压缩收集器，在某种类型的应用中会发生碎片。下面博客中 Jon 详细讨论了如何处理碎片问题：[https://blogs.oracle.com/jonthecollector/entry/when\\_the\\_sum\\_of\\_the](https://blogs.oracle.com/jonthecollector/entry/when_the_sum_of_the)

从JDK 1.5 开始，CMS 收集器中的晋升担保检查策略有些变化。原来的策略是考虑最坏情况，即新生代所有对象都晋升到老年代，新的晋升担保检查策略基于最近晋升历史情况，这种预计晋升对象比最坏情况下晋升对象要少很多，因此需要的空间也会少点。如果晋升失败，新生代处于一致状态。触发一次 stop-the-world 的标记-压缩收集。如果想在 UseSerialGC 中获得这种功能，需要设置参数 -XX:+HandlePromotionFailure。

```
283.736: [Full GC 283.736: [ParNew: 261599K->261599K(261952K), 0.0000615 secs] 826554K->826554K(1048384K),
0.0003259 secs]
```

GC locker: Trying a full collection because scavenge failed

```
283.736: [Full GC 283.736: [ParNew: 261599K->261599K(261952K), 0.0000288 secs]
```

当一个JNI 关键区被释放时会发生 Stop-the-world GC。新生代因为晋升担保失败回收失败，触发一次 Full GC。

CMS 可以运行在增量模式下(i-cms)，使用参数 -XX:+CMSIncrementalMode。在增量模式下，CMS 收集器在并发阶段，不会独占整个周期，而会周期性的暂停，唤醒应用线程。收集器把并发阶段工作，划分为片段，安排在次级(minor) 回收之间运行。这对需要低延迟，运行在少量CPU服务器上的应用很有用。

以下是增量模式 CMS的日志。

```
2803.125: [GC 2803.125: [ParNew: 408832K->0K(409216K), 0.5371950 secs] 611130K->206985K(1048192K) icms_dc=4
, 0.5373720 secs]
```

```
2824.209: [GC 2824.209: [ParNew: 408832K->0K(409216K), 0.6755540 secs] 615806K->211897K(1048192K) icms_dc=4
, 0.6757740 secs]
```

新生代花费 537 毫秒 和 675 毫秒。在2次收集之间 iCMS 短暂运行期间由icms\_dc 表示，icms\_dc 表示运行的占空比。这里占空比为4%。

简单计算下，iCMS 增量阶段费时  $4/100 * (2824.209 - 2803.125 - 0.537) = 821$  毫秒，即 2次 GC 间隔时间的 4%。

在JDK 1.5 中，CMS 增加一个并发可中止预清理(concurrent abortable preclean)阶段。可中止预清理阶段，运行在并行预清理和

重新标记之间，直到获得所期望的eden空间占用率。增加这个阶段是为了避免在重新标记阶段后紧跟着发生一次垃圾清除。为了尽可能区分开垃圾清除和重新标记，我们尽量安排在两次垃圾清除之间运行重新标记阶段。

There is a second reason why we do this. Immediately following a scavenge there are likely a large number of grey objects that need rescanning. The abortable preclean phase tries to deal with such newly grey objects thus reducing a subsequent CMS remark pause.

可以通过JVM参数CMSScheduleRemarkEdenSizeThreshold 和 CMSScheduleRemarkEdenPenetration 控制 重新标记阶段。默认值是2m和50%。CMSScheduleRemarkEdenSizeThreshold 设置Eden区大小,低于此值时不启动重新标记阶段，因为回报预期为微不足道 CMSScheduleRemarkEdenPenetration 设置启动重新标记阶段时Eden区的空间占用率。(译注：根据下面描述 Eden 应该是指整个新生代)

预清理阶段后，如果Eden 空间占用大于 CMSScheduleRemarkEdenSizeThreshold 设置的值, 会启动可中止预清理，直到占用率达到 CMSScheduleRemarkEdenPenetration 设置的值, 否则，我们立即安排重新标记阶段。(译注：与上面说的正好相反，不知是不是我翻译有误)

7688.150: [CMS-concurrent-preclean-start]

7688.186: [CMS-concurrent-preclean: 0.034/0.035 secs]

7688.186: [CMS-concurrent-abortable-preclean-start]

7688.465: [GC 7688.465: [ParNew: 1040940K->1464K(1044544K), 0.0165840 secs] 1343593K->304365K(2093120K), 0.0167509 secs]

7690.093: [CMS-concurrent-abortable-preclean: 1.012/1.907 secs]

7690.095: [GC[YG occupancy: 522484 K (1044544 K)]7690.095: [Rescan (parallel), 0.3665541 secs]7690.462: [weak refs processing, 0.0003850 secs] [1 CMS-remark: 302901K(1048576K)] 825385K(2093120K), 0.3670690 secs]

上面日志中,在预清理之后, 启动可中止预清理, 之后发生年轻代垃圾回收,年轻代占用从 1040940K 下降到 1464K. 当年轻代占用率达到 522484K 即堆的50%时,发生重新标记

注意在JDK1.5中，年轻代的垃圾回收日志输出在后面的重新标记阶段

## 汇总

对JVM内存调优的时候不能只看操作系统级别Java进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为GC过后这个值是不会变化的，因此内存调优的时候要更多地使用JDK提供的内存查看工具，比如JConsole和Java VisualVM，在配置各参数之前第一原则是检查自己业务代码，缩小问题的根源。内存溢出往往是低效代码与错误配置导致。

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生代创建对象

Permanent 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生代剩余空间

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世代的大小比例，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，LargePageSizeInBytes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>

## MyCAT调优

MyCAT所有的调优参数都可以在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。

## 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。所花费的时间也多很多，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，要考滤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

## 加入Mycat

### 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/>

## 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：

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

## Mycat开发基础

### 代码调试入口

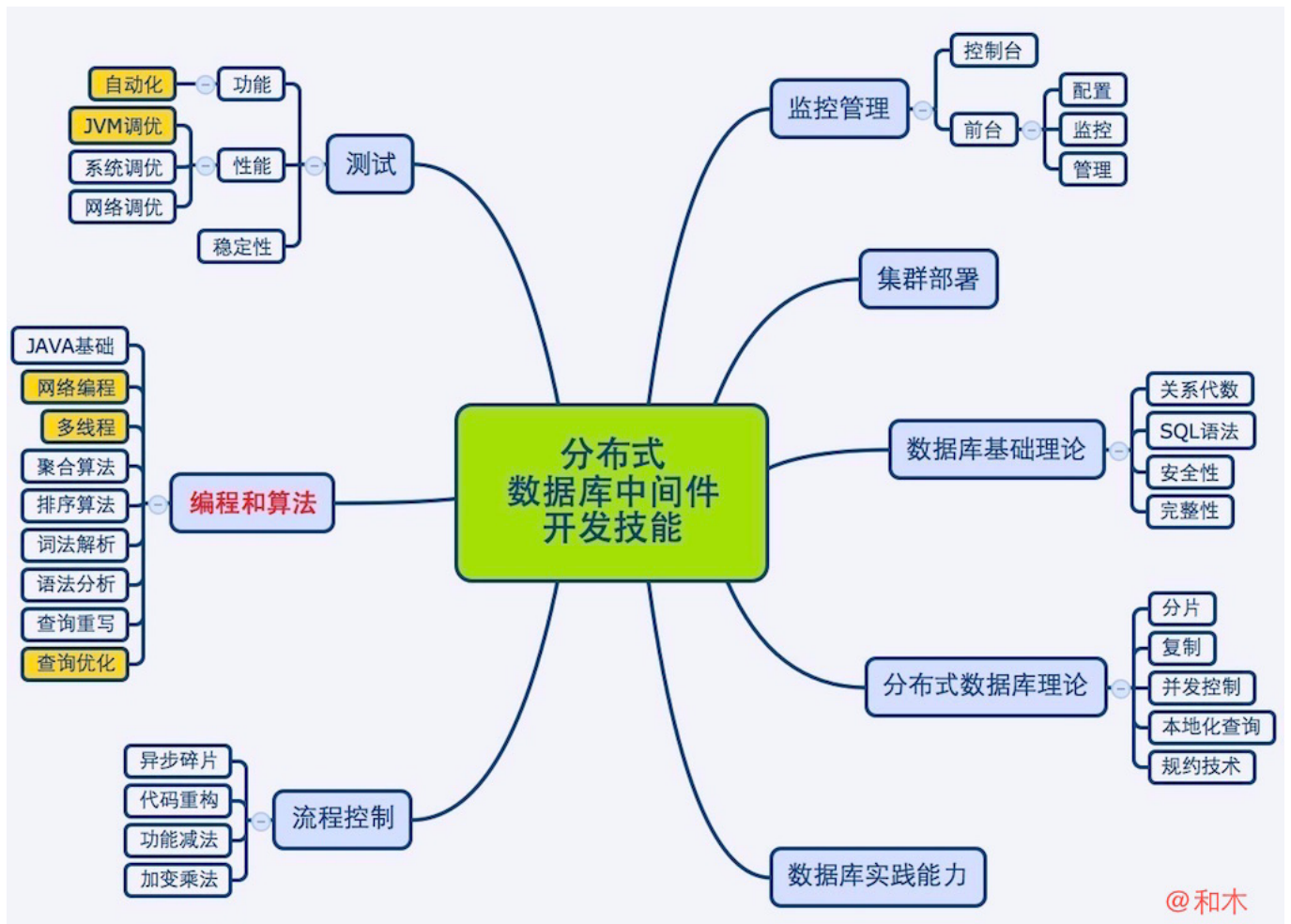
Mycat运行的main class 为MycatStartup。在获取源代码之后，导入到IDE中。配置相关的启动参数就可以在IDE中调试Mycat了。

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`。

### 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



## Mycat架构分析

### MyCAT和TDDL、Amoeba、Cobar的架构比较

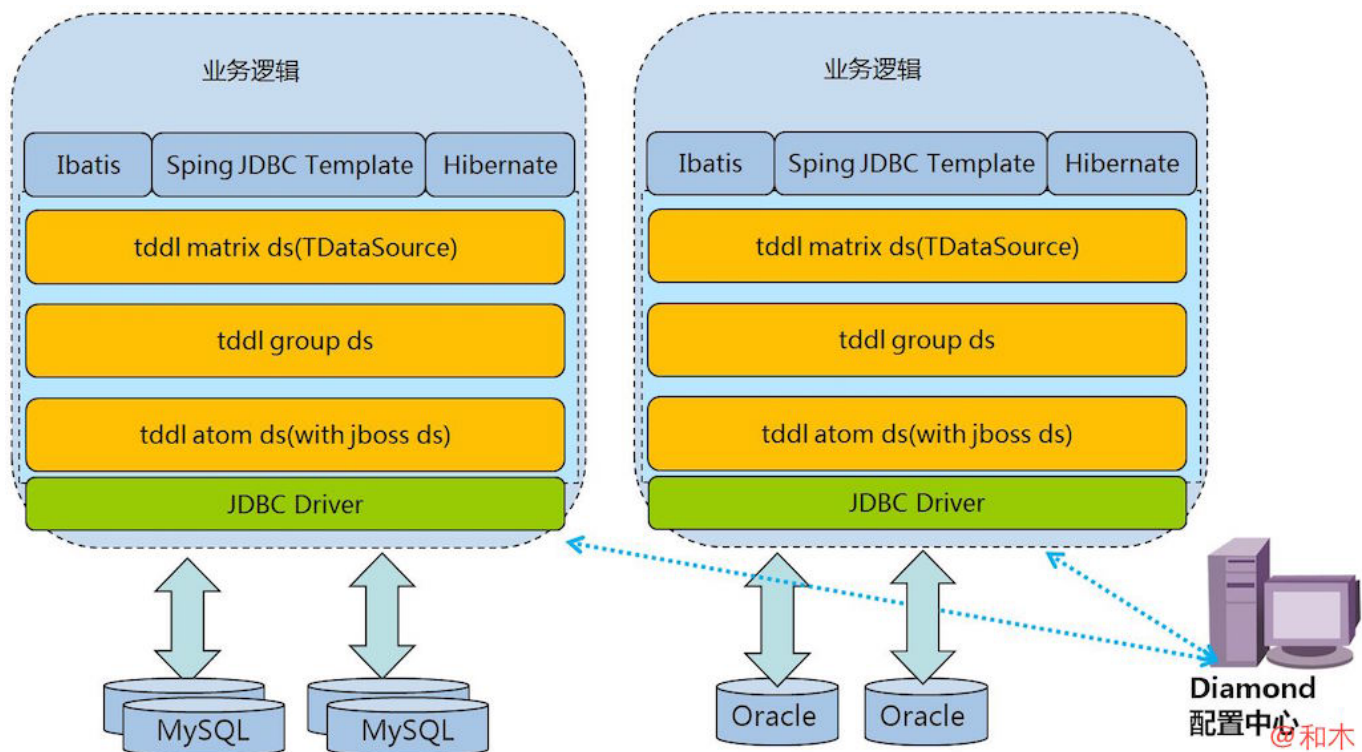
分布式数据库中间件TDDL、Amoeba、Cobar、MyCAT架构比较

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

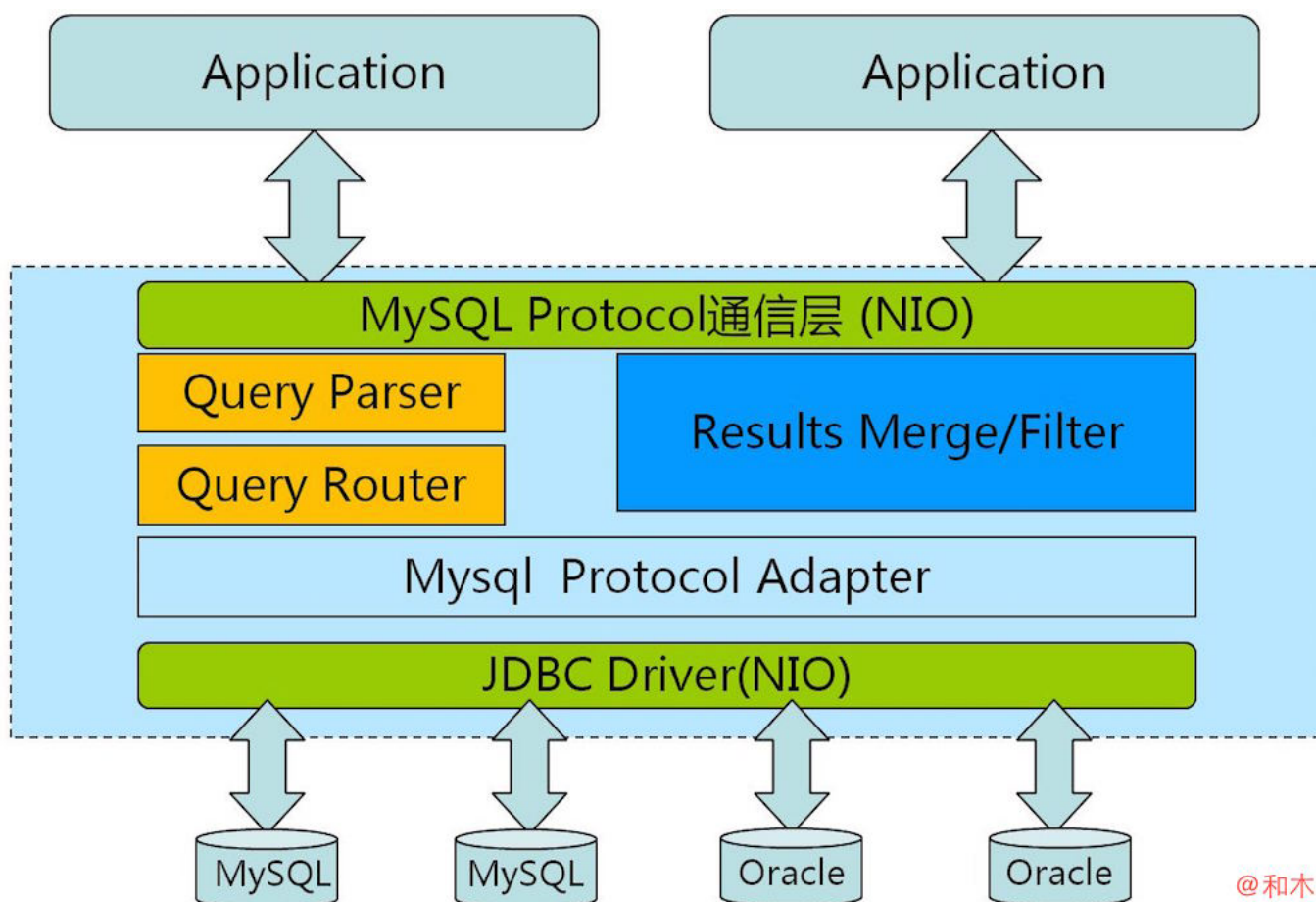
## 框架比较

### TDDL

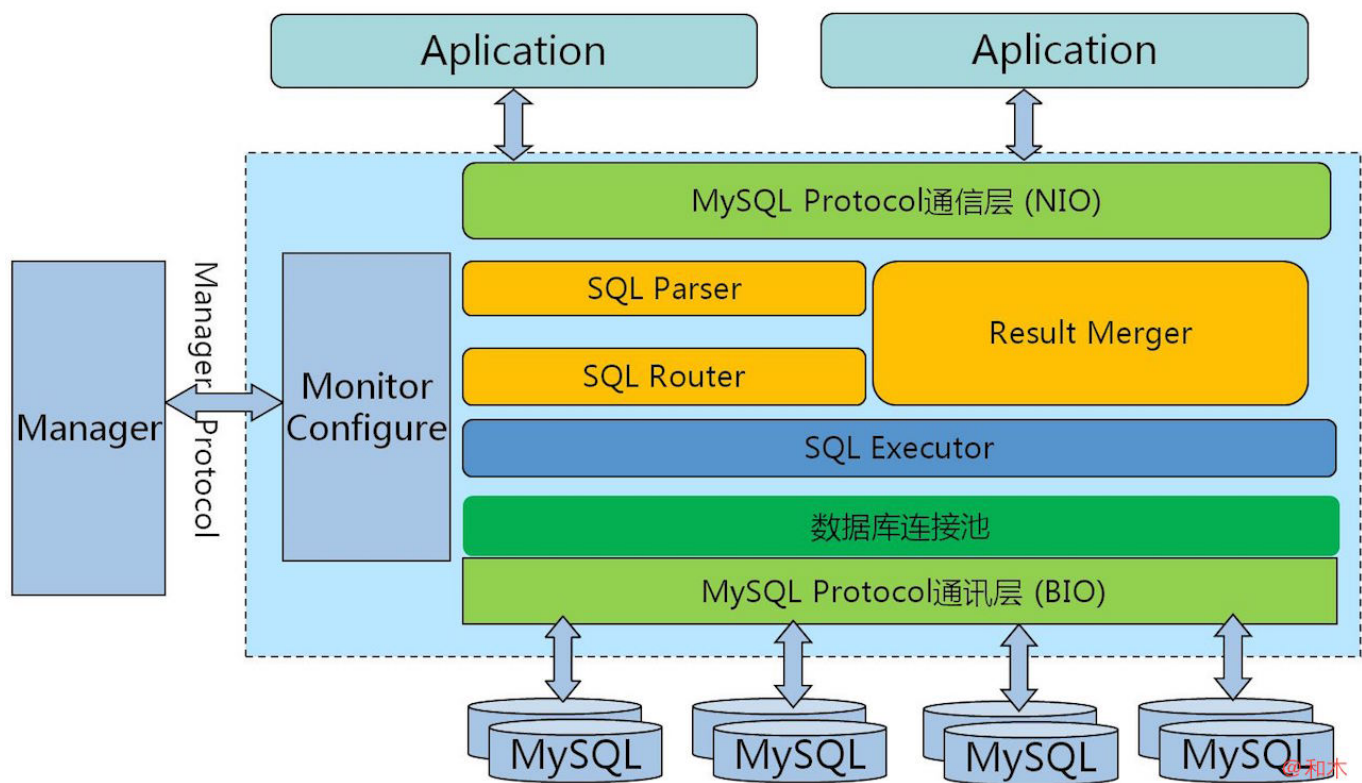




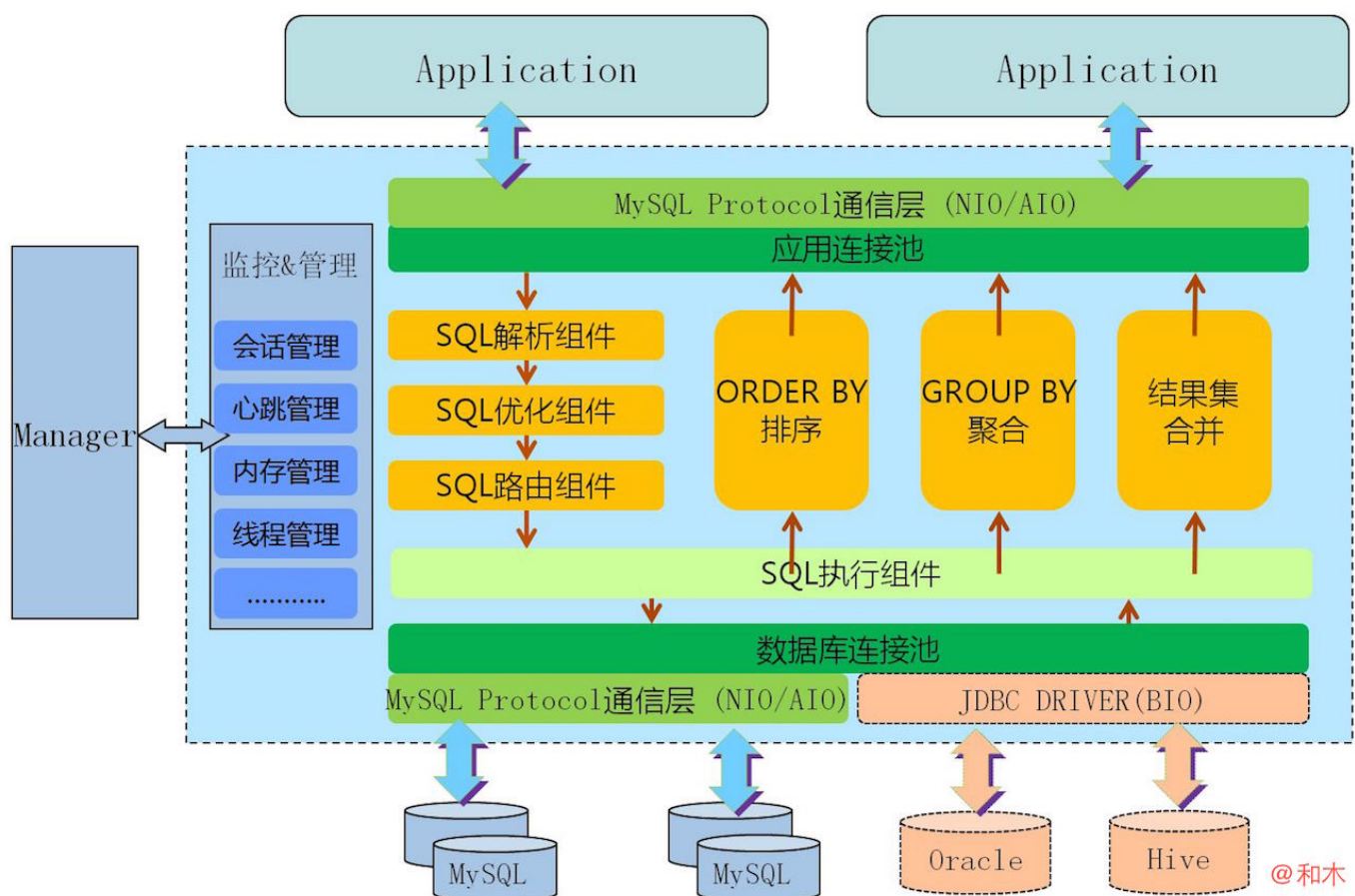
## Amoeba



## Cobar



## MyCat



## 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，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。
5. 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。
6. MyCat又是在Cobar基础上发展的版本，两个显著点是：
7. 后端由BIO改为NIO，并发量有大幅提高
8. 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）。
9. 目前社区情况：
10. TDDL处于停滞状态
11. Amoeba处于停滞状态
12. Cobar处于停滞状态
13. **MyCAT社区非常活跃**
14. 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。

## 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)

<http://songwie.com/articlelist/44>

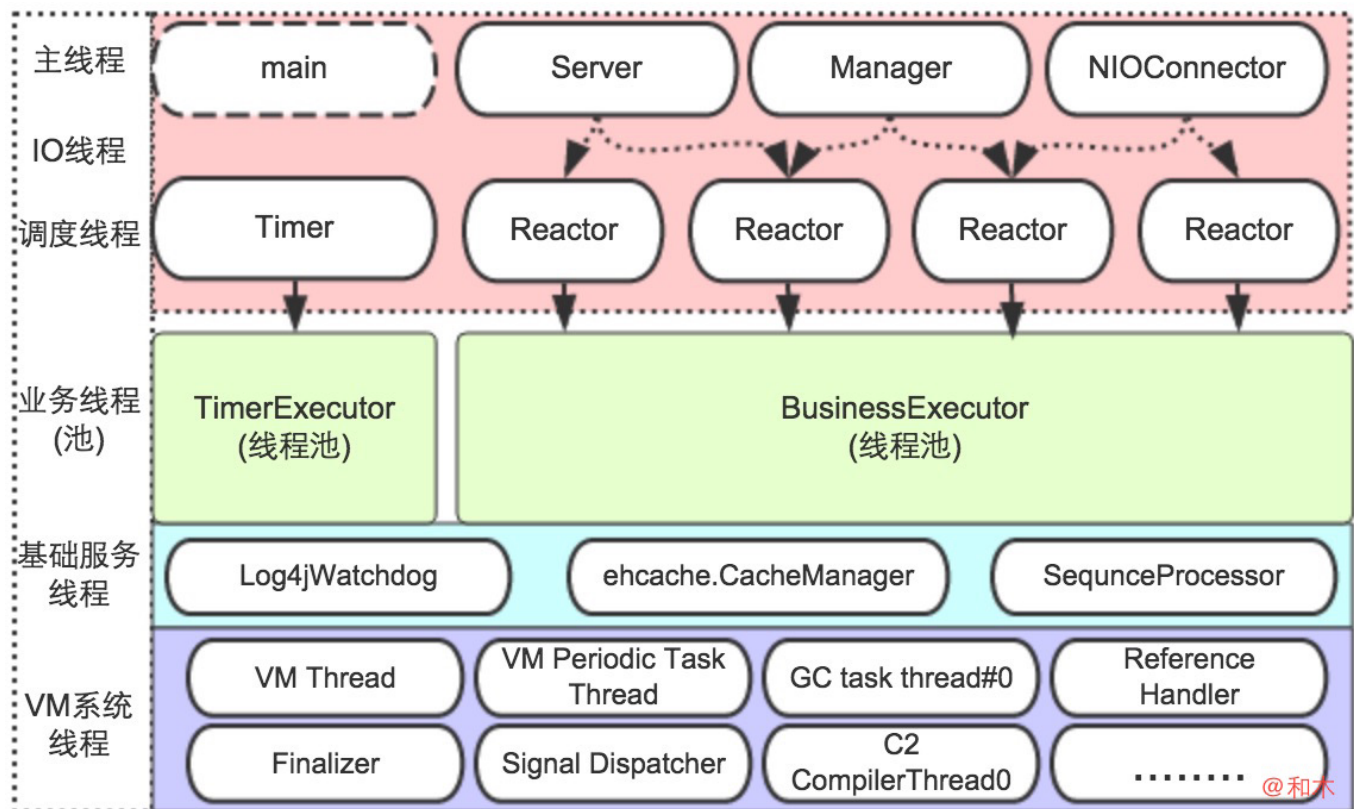
mysql中间件研究 ( Atlas, cobar, TDDL )

<http://www.guokr.com/blog/475765/>

## MyCAT线程模型分析

[TOC]

## MyCAT线程模型



## 线程介绍

### Timer

Timer单线程仅仅负责调度，任务的具体动作交给timerExecutor。

### 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。

### 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

### 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

### 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

### NIOReactor读写事件分离器



默认个数N=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

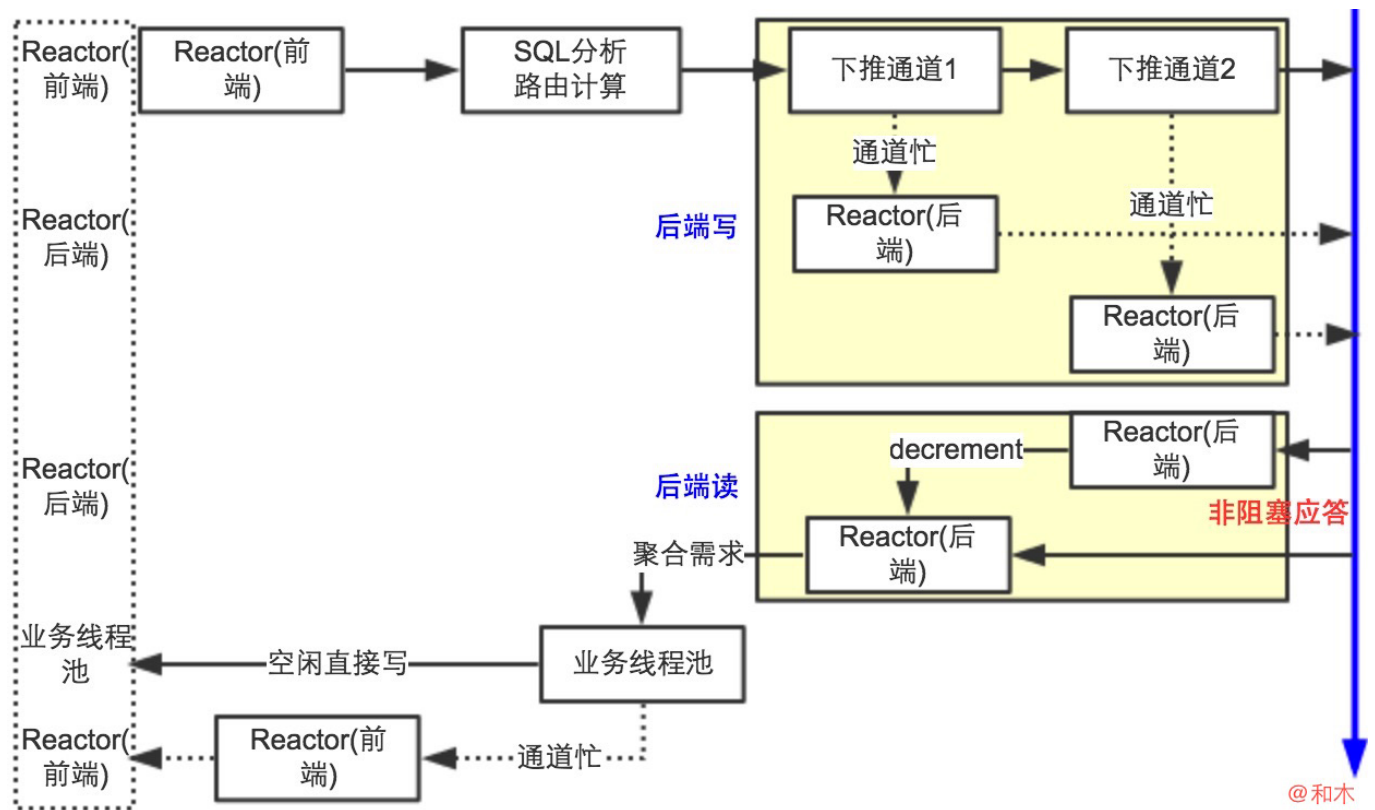
## BusinessExecutor线程池

默认大小N=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

## 一个SQL请求的线程切换



## MyCAT的线程快照

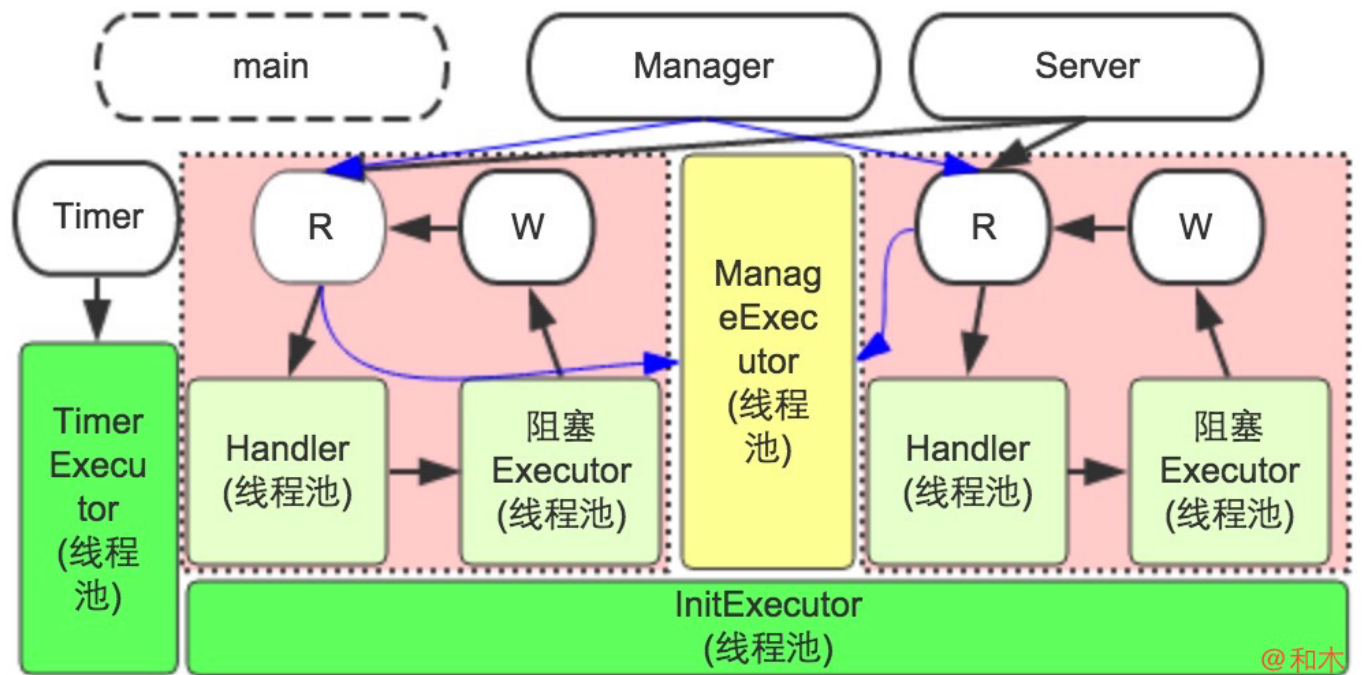
```
jstack 34179|grep prio
"Attach Listener" #32 daemon prio=9 os_prio=31 tid=0x00007f8f8ba15800 nid=0x2f07 waiting on condition
[0x0000000000000000]
"Timer1" #31 daemon prio=5 os_prio=31 tid=0x00007f8f8c0d1000 nid=0x7703 waiting on condition
[0x0000000126510000]
"Timer0" #30 daemon prio=5 os_prio=31 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
[0x00000000126001000]
"BusinessExecutor3" #24 daemon prio=5 os_prio=31 tid=0x00007f8f89937800 nid=0x6703 waiting on condition
[0x00000000125efe000]
"BusinessExecutor2" #23 daemon prio=5 os_prio=31 tid=0x00007f8f8a443800 nid=0x6503 waiting on condition
[0x00000000125dfb000]
"BusinessExecutor1" #22 daemon prio=5 os_prio=31 tid=0x00007f8f8a43c000 nid=0x6303 waiting on condition
[0x00000000125cf8000]
"BusinessExecutor0" #21 daemon prio=5 os_prio=31 tid=0x00007f8f8a3ae000 nid=0x6103 waiting on condition
[0x00000000125bf5000]
"$MyCatServer" #20 prio=5 os_prio=31 tid=0x00007f8f8c098000 nid=0x5f03 runnable [0x00000000125af2000]
"$MyCatManager" #19 prio=5 os_prio=31 tid=0x00007f8f8a8ce800 nid=0x5d03 runnable [0x000000001259ef000]
"$NIOConnector" #18 prio=5 os_prio=31 tid=0x00007f8f89956800 nid=0x5b03 runnable [0x000000001256ec000]
"$NIOREACTOR-3-RW" #17 prio=5 os_prio=31 tid=0x00007f8f898b9000 nid=0x5903 runnable [0x000000001255e9000]
"$NIOREACTOR-2-RW" #16 prio=5 os_prio=31 tid=0x00007f8f8a914800 nid=0x5703 runnable [0x000000001254e6000]
"$NIOREACTOR-1-RW" #15 prio=5 os_prio=31 tid=0x00007f8f8a8d9800 nid=0x5503 runnable [0x000000001253e3000]
"$NIOREACTOR-0-RW" #14 prio=5 os_prio=31 tid=0x00007f8f8a8d9000 nid=0x5303 runnable [0x000000001252e0000]
"Log4jWatchdog" #13 daemon prio=5 os_prio=31 tid=0x00007f8f8a305000 nid=0x5107 waiting on condition
[0x000000001251cd000]
"net.sf.ehcache.CacheManager@512ddf17" #11 daemon prio=5 os_prio=31 tid=0x00007f8f8a32d000 nid=0x4f03 in
Object.wait() [0x000000001250ca000]
"MyCatTimer" #10 daemon prio=5 os_prio=31 tid=0x00007f8f8a162800 nid=0x4d03 in Object.wait()
[0x00000000124fab000]
"Thread-0" #9 prio=5 os_prio=31 tid=0x00007f8f8b082000 nid=0x4b03 waiting on condition [0x00000000124cf1000]
"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()
[0x00000000122b34000]
"Reference Handler" #2 daemon prio=10 os_prio=31 tid=0x00007f8f8a00d800 nid=0x2b03 in Object.wait()
[0x00000000122a31000]
"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=0x00007f8f8a408000 nid=0x4903 waiting on condition

```

## Cobar线程介绍



## 线程介绍

### Timer

Timer单线程仅仅负责调度，任务的具体动作交给timerExecutor。

### 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，定时任务没有那么大的运算量。

### 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

### 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

### R读写事件分离器

客户端与Server连接后，由R线程负责读写事件（写事件大部分有W线程负责，只有在网络繁忙时才会由小部分写事件是由R线程完成的）。

### Handler和Executor线程池

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

## 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。

## 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。

## InitExecutor线程池

用来进行后端链路初始化。

## 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？

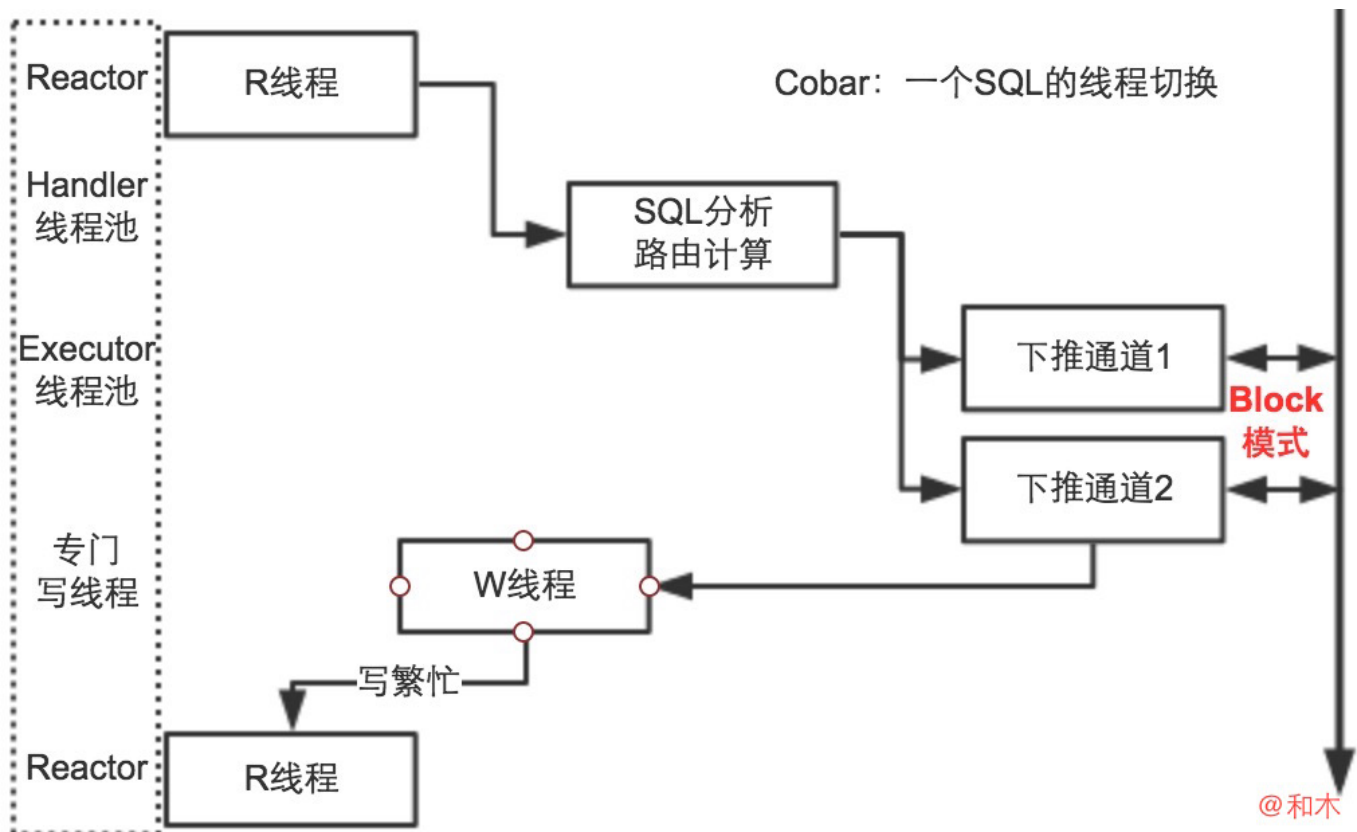
我不是Cobar的原作者，只能猜测最什么设计这么多线程池？那就是因为\*\*后端采用了BIO\*\*！

- 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。

## 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





## 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
```

## MyCAT与Cobar的比较

# 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。

## 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。

## 特别说明ER分片机制

如果涉到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。

## 数据验证

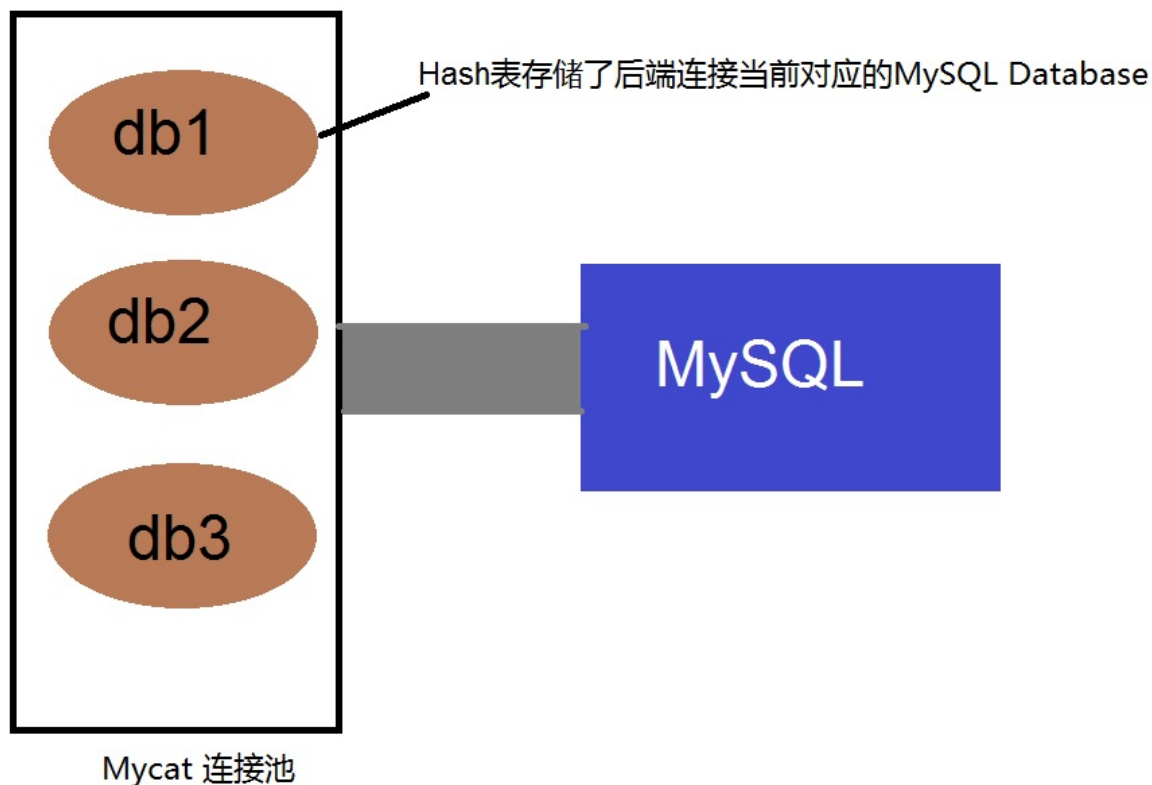
1. 测试sql从收到请求到下推的总时长，如果时间可容忍，则不必切换到线程池。**\*\*忽略ER分片\*\***
2. 对于manager端口的命令，若存在执行时间比较的，也需要改为线程池来执行
3. 对于收到的应答，大部分都不必切换到线程池。
4. 对于大量数据排序，只有在排序时，构造执行任务，切换到线程池完成。

## mycat的连接池

### Mycat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。

org.opencloudb.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activeIndex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Database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。

## Mycat的网络通信框架

### 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

## 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。

## 比对情况

	MyCAT	Cobar																																										
全局计划	一样都是模拟1500个用户。同时启动运行20分钟。																																											
	<div><div>全局计划</div><div><div><div><div></div><div></div><div></div><div></div><div></div><div></div></div><div>总数： 1500 个 Vuser</div></div></div><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></div>		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&gt; use dbtest; Database changed mysql&gt; show cobar_status; +-----+   STATUS   +-----+   ON        +-----+ 1 row in set &lt;0.00 sec&gt;  mysql&gt; show cobar_cluster; Empty set &lt;0.00 sec&gt;  mysql&gt;</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~~~

## MyCAT网络框架

# 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。

## 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，事件分离者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），事件分离者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(或者代由事件分离者发起)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。事件分离者得知了这个请求，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)，事件分离者等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)

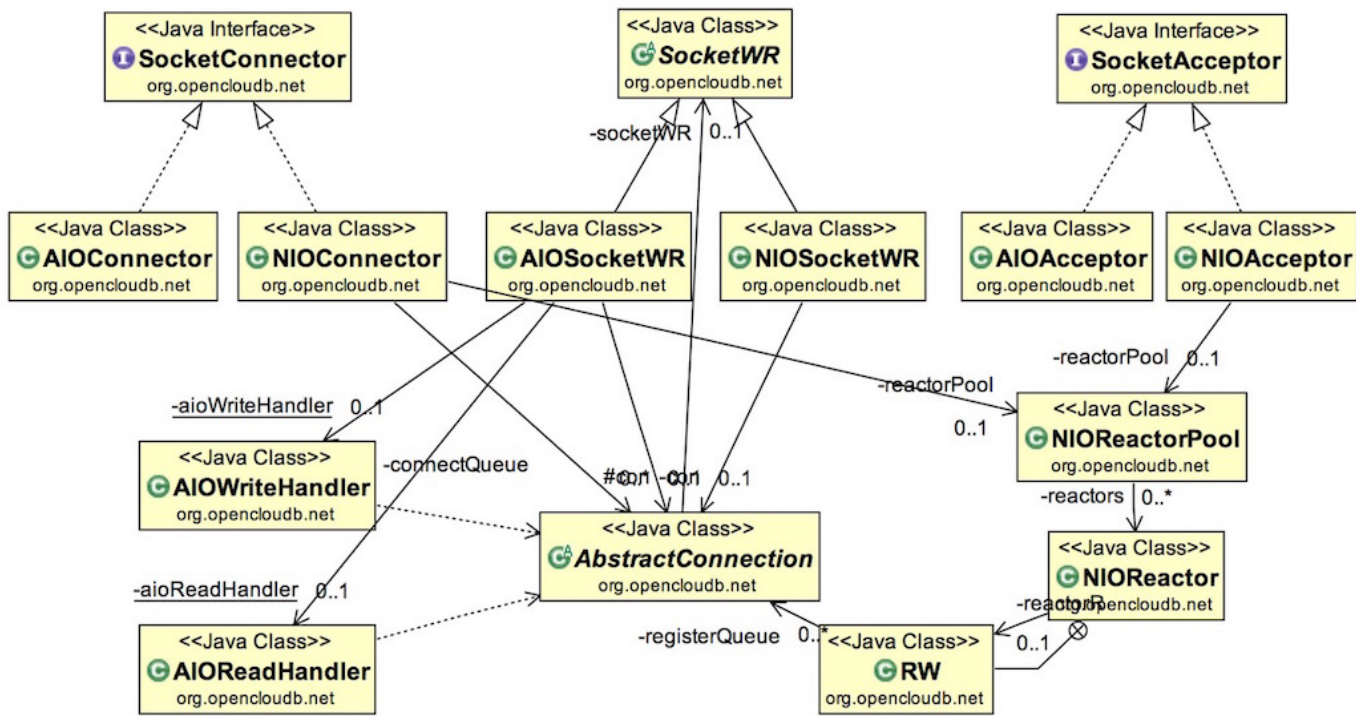
从上面可以看出，Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，而Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的IO设备。

最后结合下面的两张图更容易理解（这是别人的图，非原创）：





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：

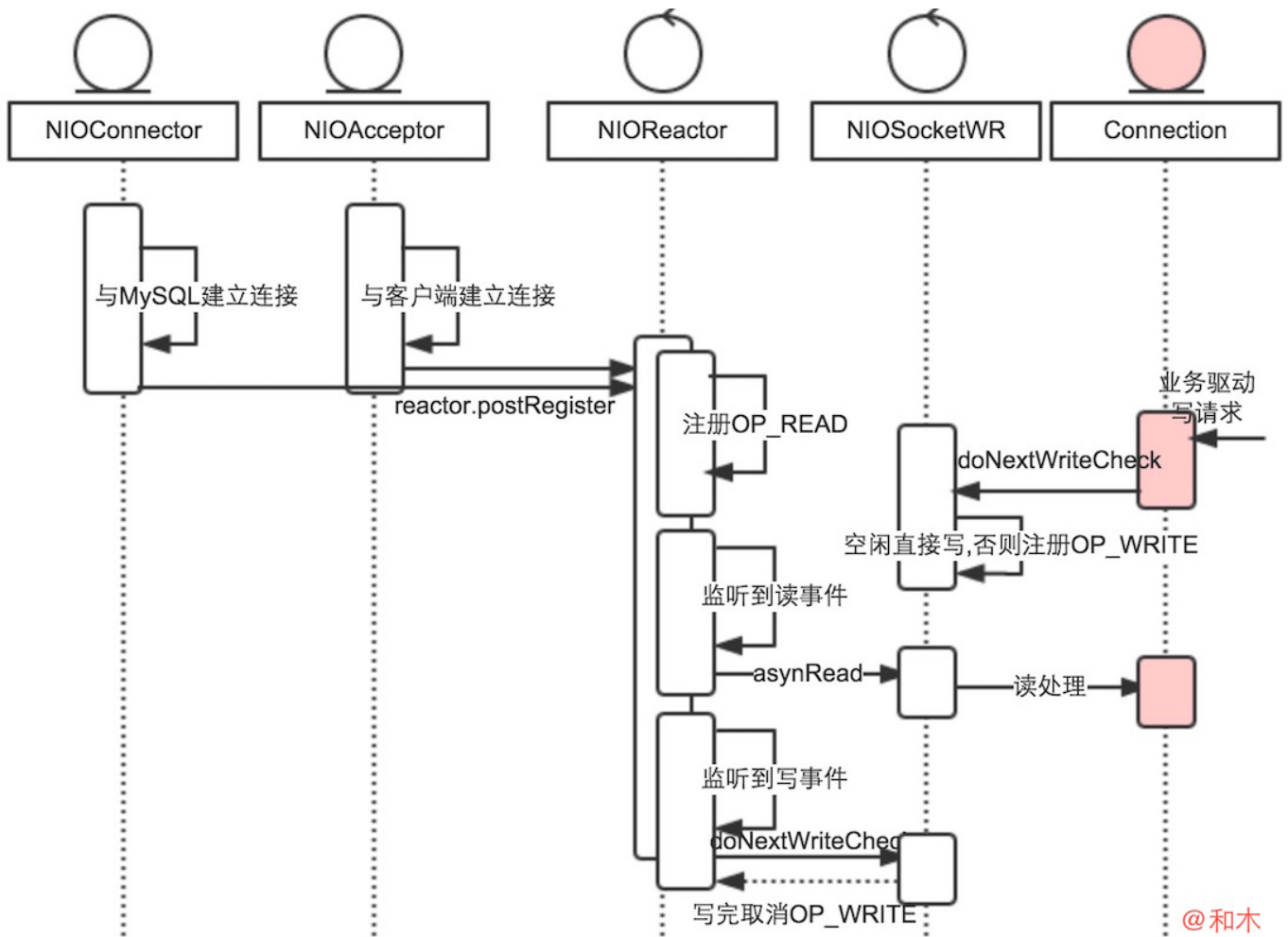
- Connect客户端连接服务端事件
- Accept 服务端接收客户端连接事件
- Read 读事件
- Write 写事件

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

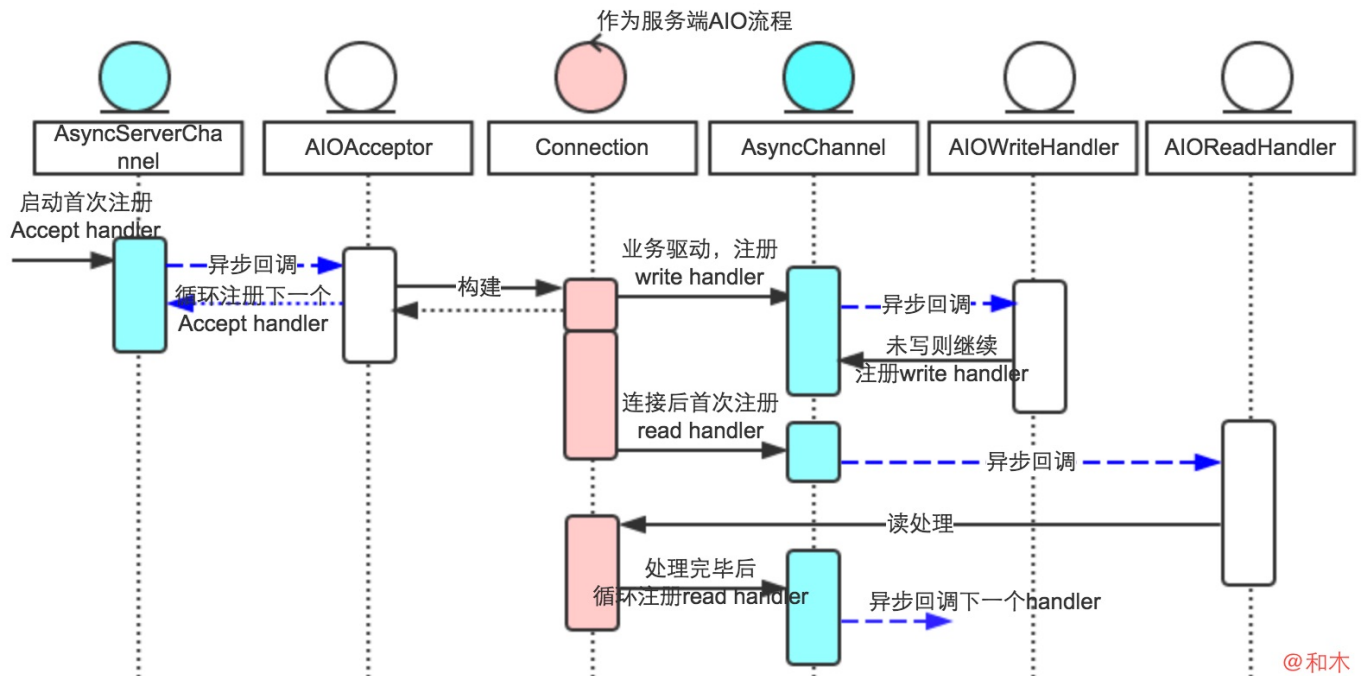
上面的类图看起来有些复杂，因为把NIO和AIO放在一起了，那么我们分开来讲

NIO主要类调用



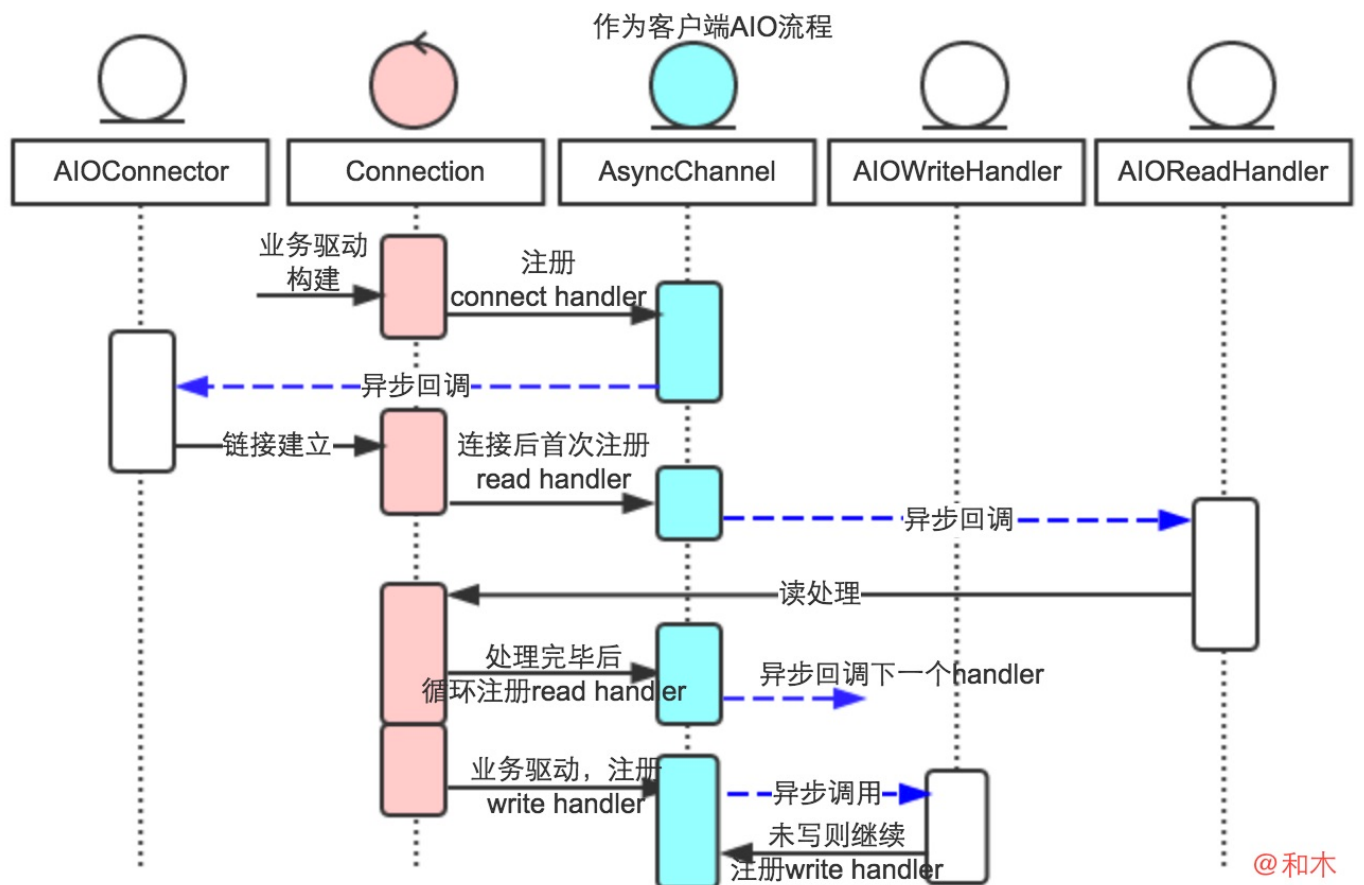
@和木

## AIO主要类调用-服务端



@和木

## 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：)

## 4.MyCAT的NIO实现

Selector（选择器）是Java NIO中能够检测到多个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

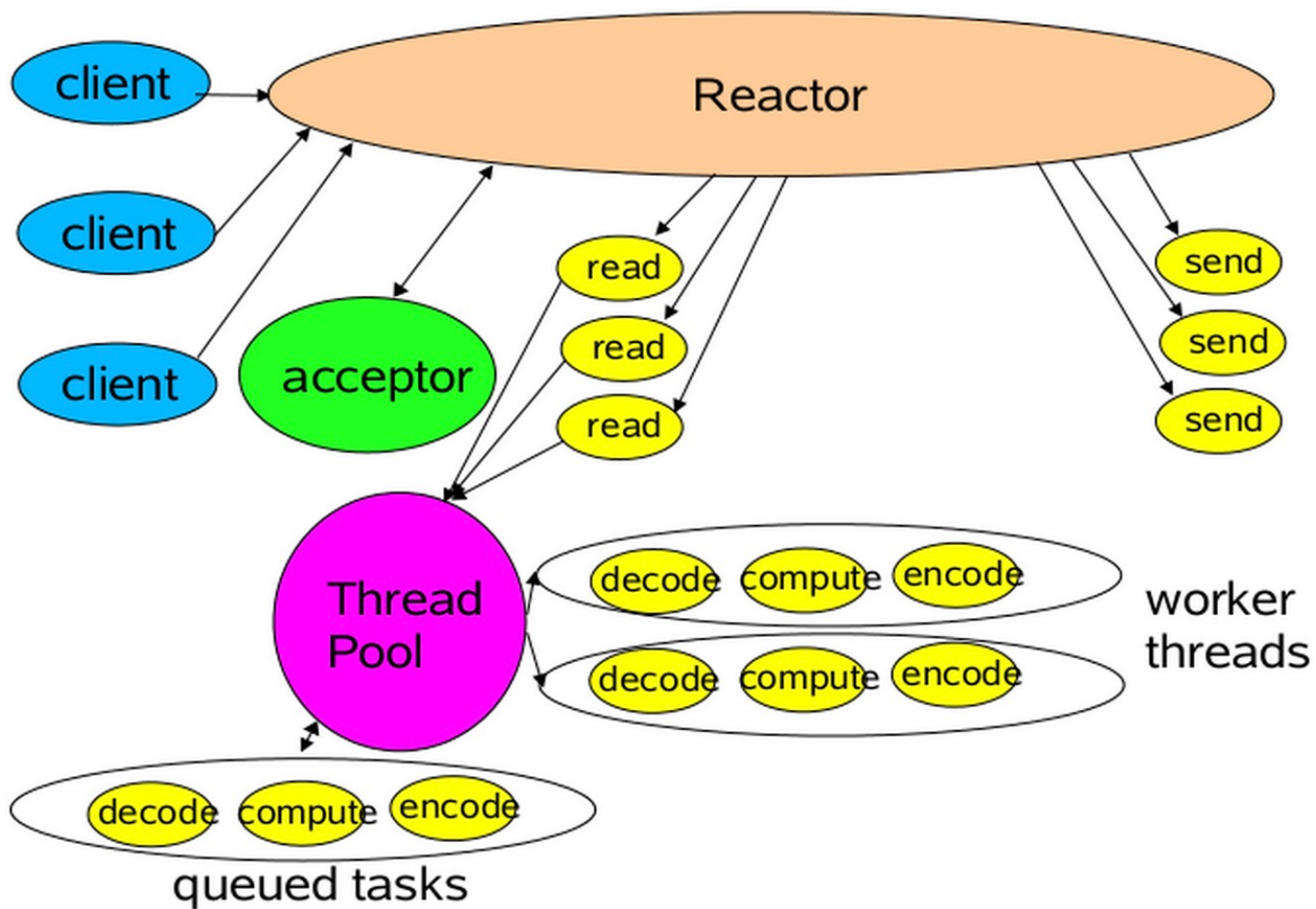
这四种事件用SelectionKey的四个常量来表示：

- SelectionKey.OP\_CONNECT
- SelectionKey.OP\_ACCEPT
- SelectionKey.OP\_READ
- SelectionKey.OP\_WRITE

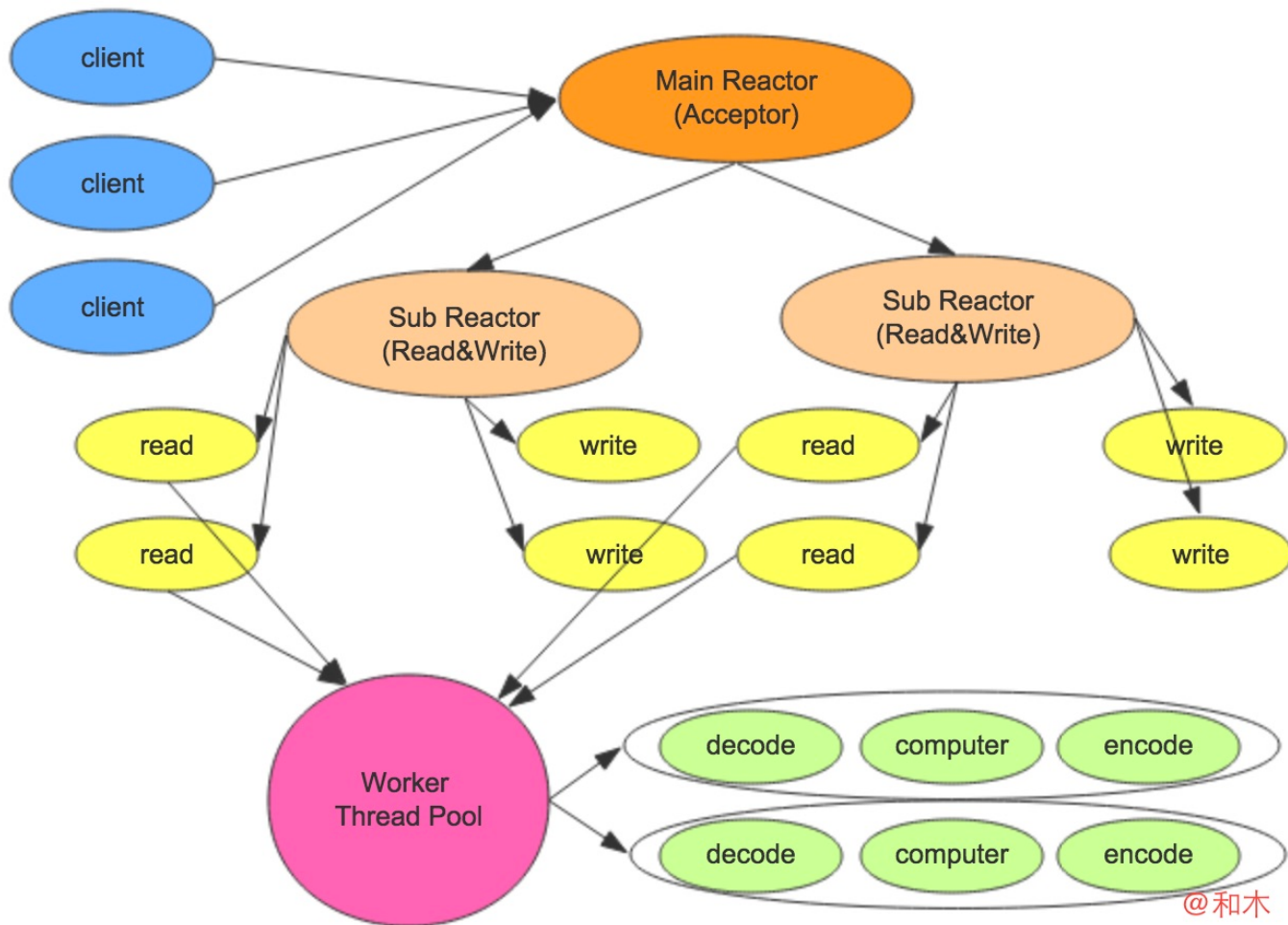
前面已经说了，NIO采用的Reactor模式：例如汽车是乘客访问的主体（Reactor），乘客上车后，到售票员（acceptor）处登记，之后乘客便可以休息睡觉去了，当到达乘客所要到达的目的地后，售票员将其唤醒即可。

典型的Reactor场景

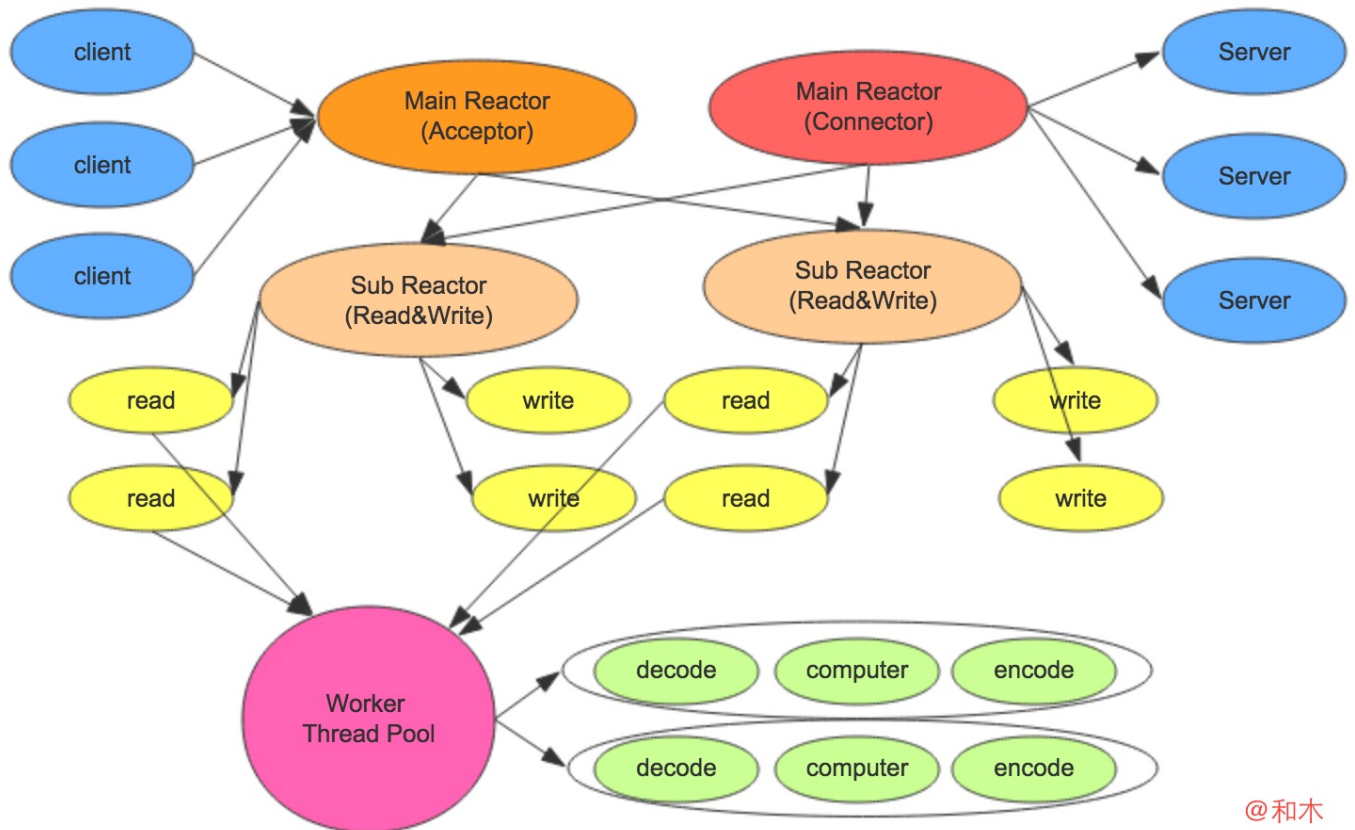




在高性能IO框架中，大都是采用多Reactor模式,即多个dispatcher，如下图所示：



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，



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();
}
```

## 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.opencloudb.net.NIOConnector
  ▼ make(MySQLDataSource, ResponseHandler, String) : MySQLConnection - org.opencloudb.mysql.nio.MySQLConnectionFactory
    ► createNewConnection(ResponseHandler, String) : void - org.opencloudb.mysql.nio.MySQLDataSource
  ▼ make(MySQLHeartbeat) : MySQLDetector - org.opencloudb.heartbeat.MySQLDetectorFactory
    ► heartbeat() : void - org.opencloudb.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);
    .....
}

```

## 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);
    .....
}

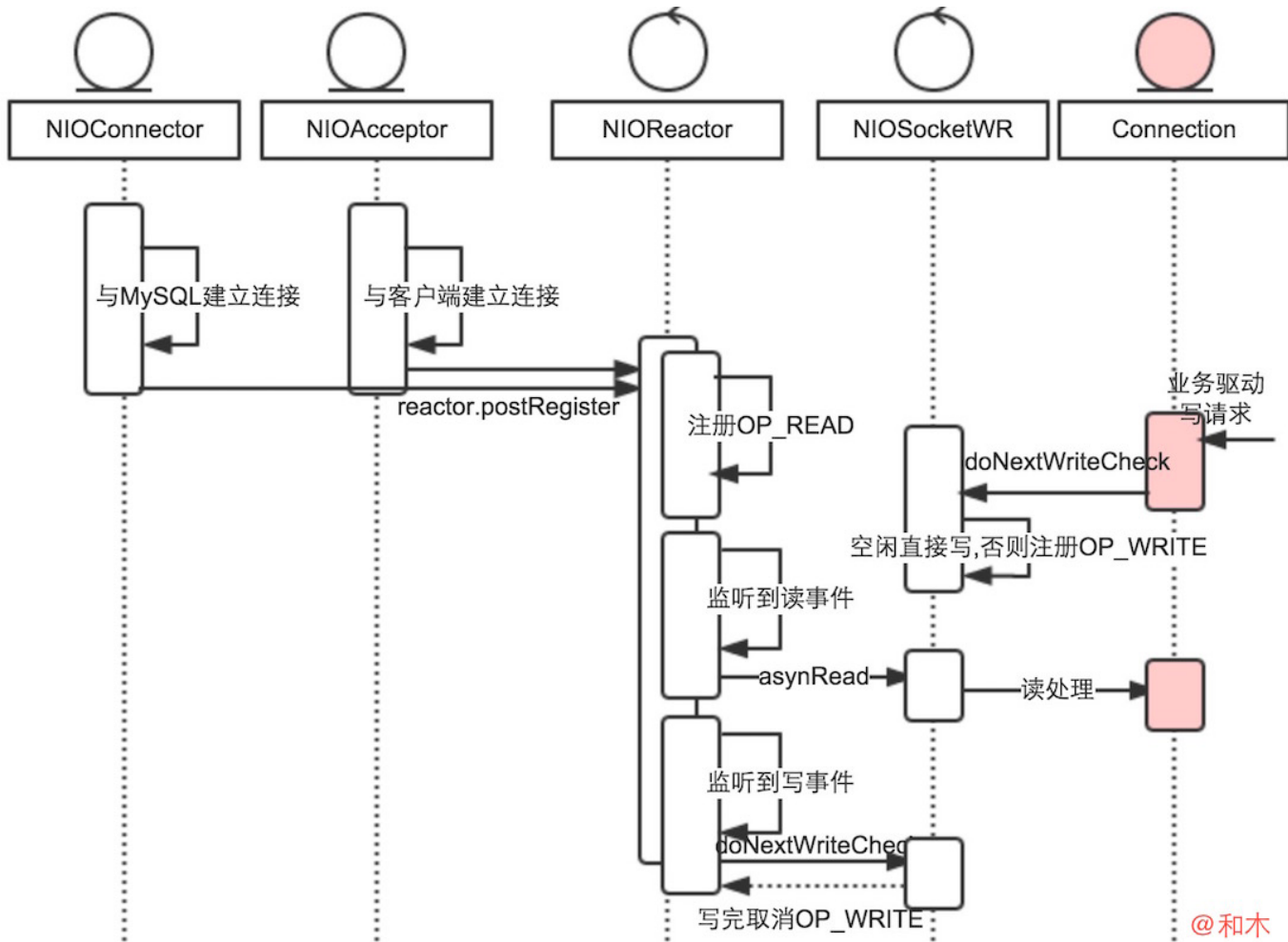
```

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。

## 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



@和木

## 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();
}

```

NIORreactor.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。

## NIORreactor.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.asyncRead();
                        }
                        if (key.isWritable()) {
                            con.doNextWriteCheck();
                        }
                    } else {
                        key.cancel();
                    }
                } catch (Throwable e) {
                }
            }
        } catch (Throwable e) {
            LOGGER.warn(name, e);
        } finally {
            if (keys != null) {
                keys.clear();
            }
        }
    }
}

```

NIORreactor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。

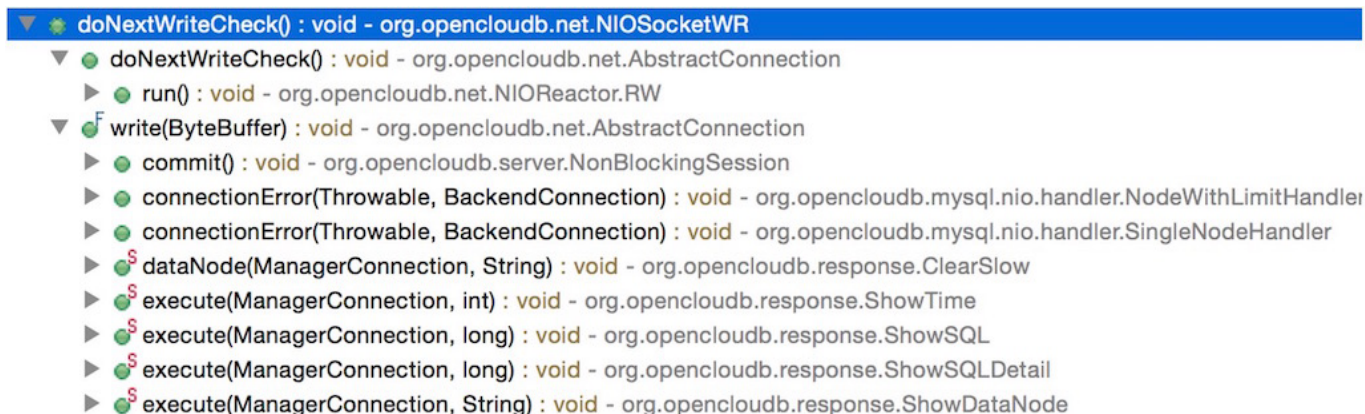
- selector监听事件，如果是读事件，就调用con.asyncRead()函数，进行字节的读取。对于asyncRead中如何提取MySQL协议包，就属于网络框架讨论的内容，可以参考其它章节。

- 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



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。

## 5.与Cobar原有NIO细节比较

### 5.1.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();
    }
}

```

## 5.2.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.MyCAT的AIO实现

### 6.1.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。

## 6.2.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

## 6.3.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();
        }
    }
}

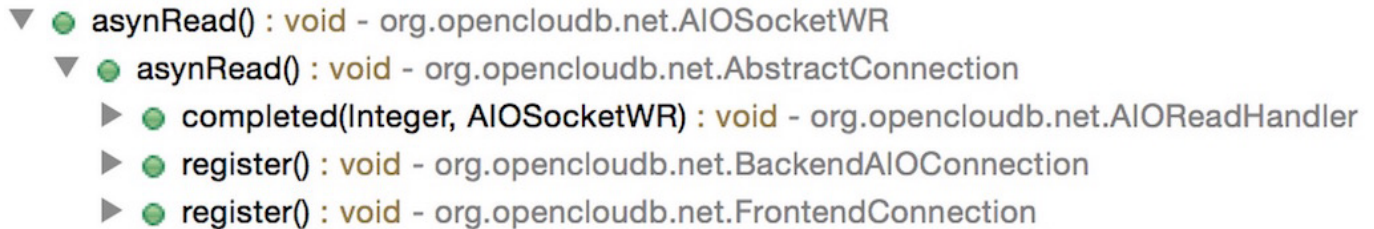
```

```
    } 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.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



- 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

### 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 ");  
    }  
}
```

### 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

## 6.5.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
    ► ● 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AbstactConnection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();
    }
}
}

```

## Mycat的路由与分发流程

### 路由的作用

### 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。

### 路由解析器

### 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）。

## 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 org.opencloudb.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,JSQParser cost:2892ms
100000 times parse ,druid cost:456ms

```

**\*\* 100万次输出结果：\*\***

```

1000000 times parse,fdb cost:30280ms
1000000 times parse,JSQParser cost:18983ms
1000000 times parse ,druid cost:1912ms

```

**\*\* 结论：\*\***

10万次：druid比fdbparser快10倍，比JSQParser快6倍；

100万次：druid比fdbparser快15倍，比JSQParser快近10倍

## druid路由解析的两种方式

Druid解析有两种方式：vistor方式和statement方式。

### Vistor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。

### 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;
```

## 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(shardingColIndex);
    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);
```

## 其他改写

其他改写还有很多，可以通过druid的api自由发挥。

### 路由计算

## 路由计算接口

路由计算的入口方法为org.opencloudb.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
```

### 路由计算简要数据流图

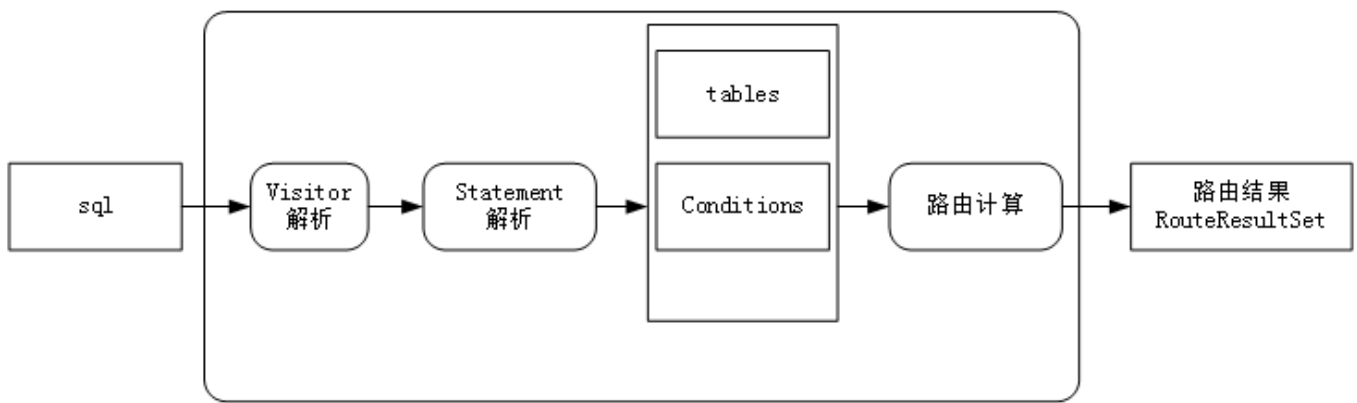


说明：输入一个sql，经过路由计算，输出路由结果。  
该图实际是对路由接口的一个简化。路由接口中还包含SystemConfig、SchemaConfig、sqlType、charset、ServerConnection等其他输入参数，但对于路由计算来说，这些参数都不是最主要参数。如SystemConfig、SchemaConfig两个参数，完全可以不用传入，我们可以直接用其他方式获取，如：

```
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。

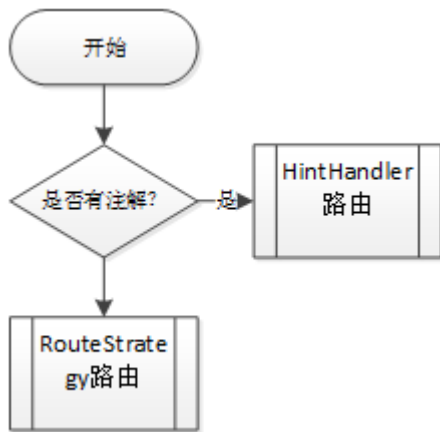
### 路由计算分解数据流图



其中conditions中每个condition为<表名、字段名、字段值>的3元组。

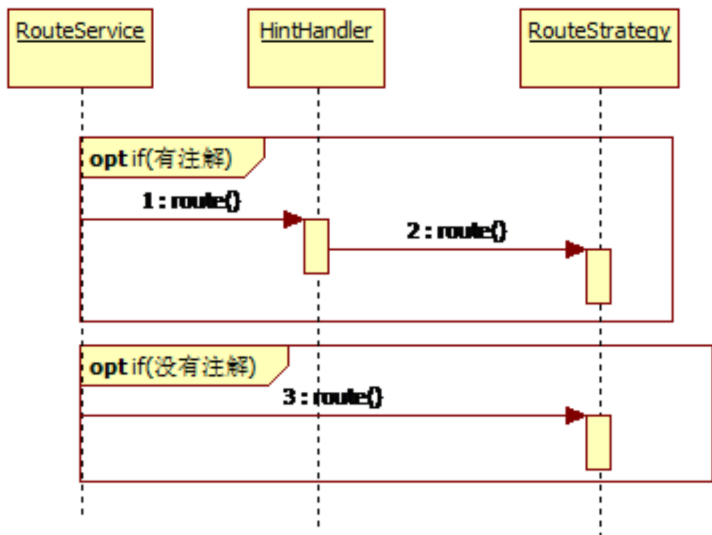
### 路由计算流程

路由解析总体流程



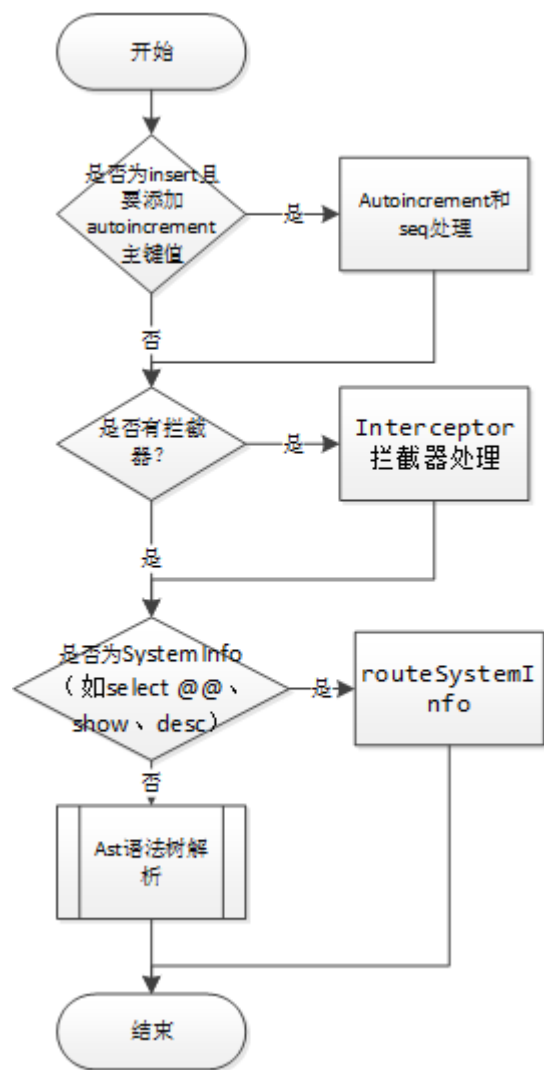
其中RouteStrategy路由为子流程，见RouteStrategy路由子流程对其展开讲解。HintHandler路由也是子流程，但非主流程故本文不做重点讲解。

路由解析序列图



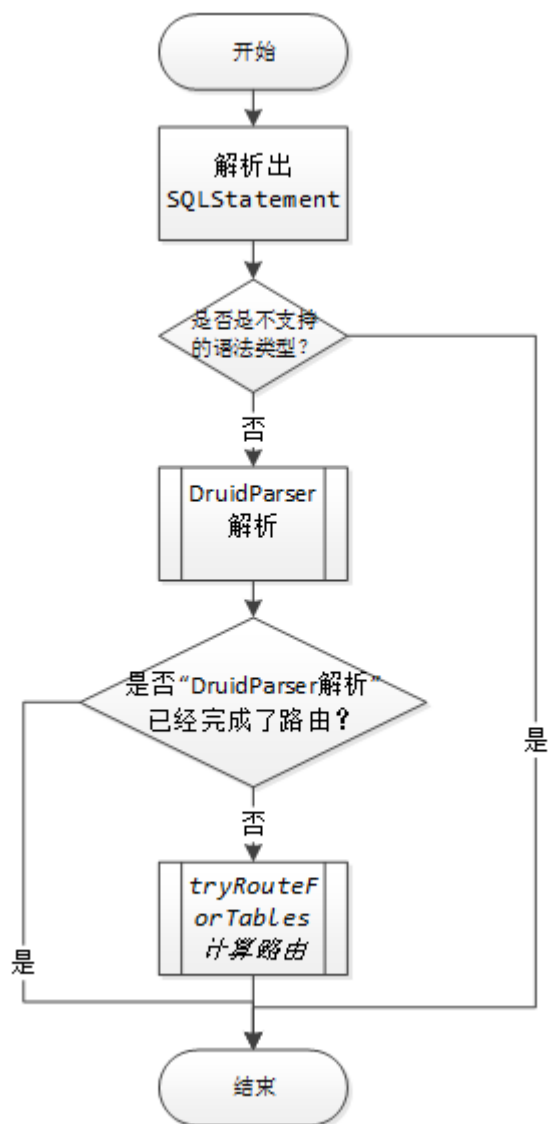
路由解析入口都从RouteService类的route方法进入，然后根据是否有注解决定是走HintHandler还是RouteStrategy进行路由解析。

RouteStrategy路由子流程

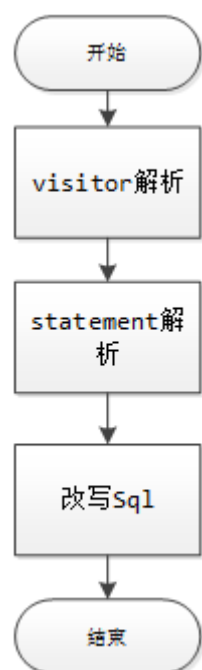


该流程是fdbparser和druidparser两种解析策略的公共流程。该流程封装在AbstractRouteStrategy类的route方法中，相当于两种策略的模板方法。子流程“Ast语法树解析”对应routeNormalSqlWithAST方法，下一节将对ast语法树解析流程再展开讲解（以DruidMysqlRouteStrategy策略类为例）。

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。

## 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（ Conditions ），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。

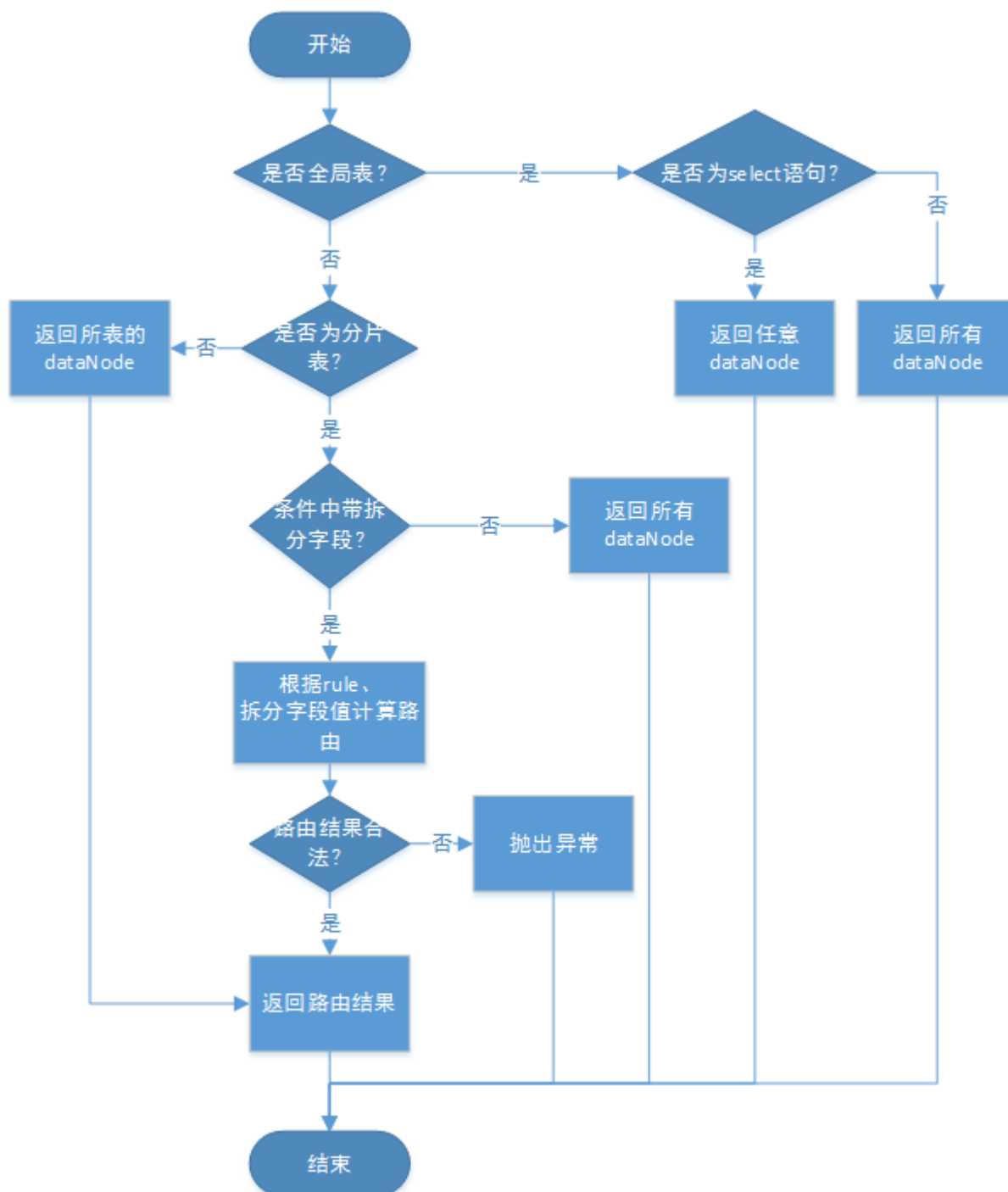
有以上一些数据就能计算出路由，所以路由计算需要解决以下问题：

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。

## 路由的计算

### 单个表的路由计算

单表路由计算流程



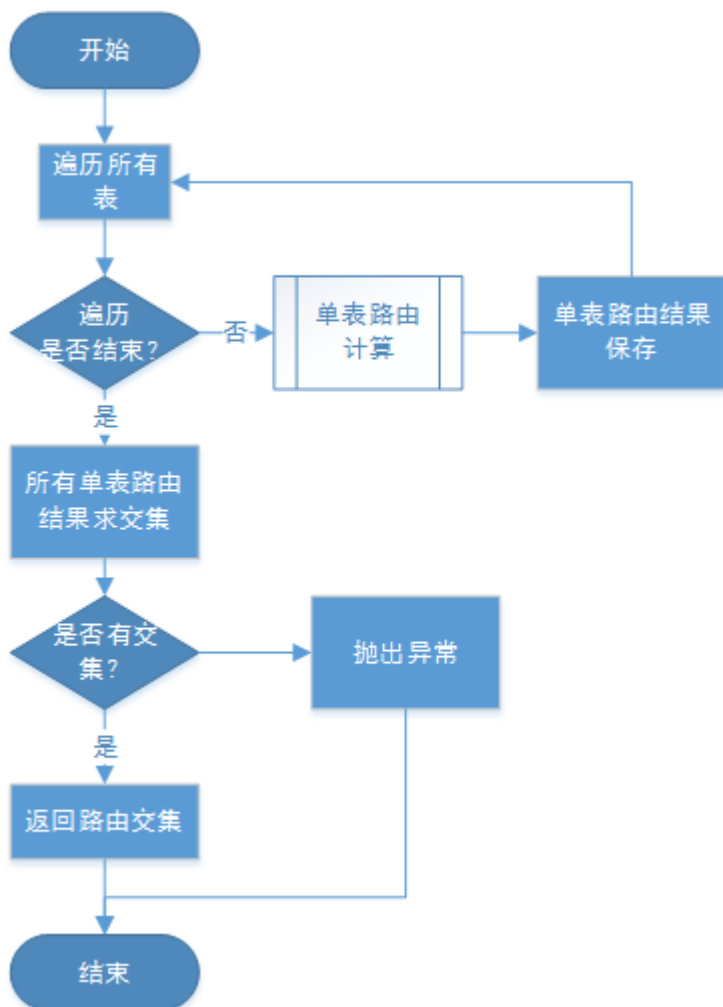
## 无表语句的路由计算

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());
}
```

## 多个表的路由计算

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。

## 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());
    }
}
  
```

## 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。



## 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。

## 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);
        }
    }
}

```

## 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;

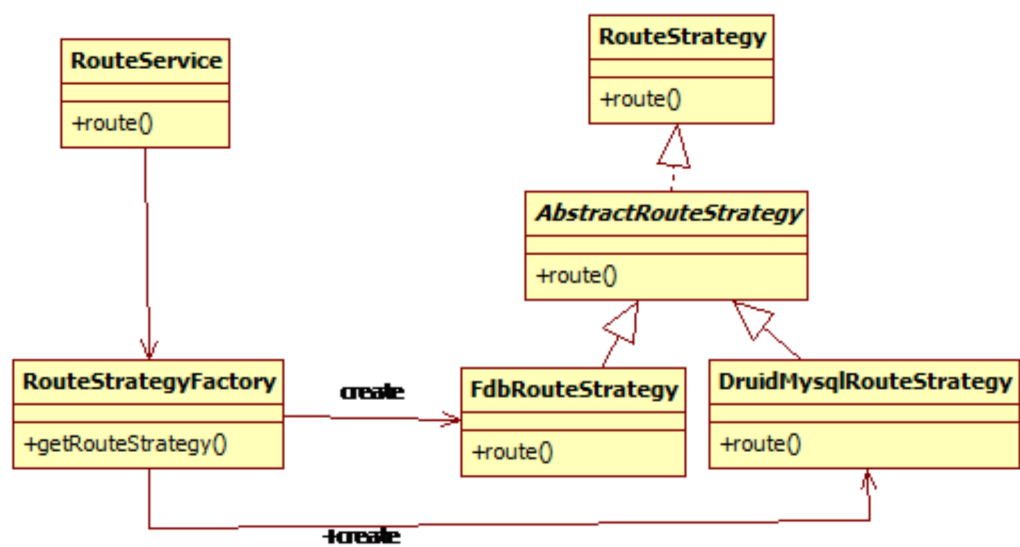
show full tables from databaseName;

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

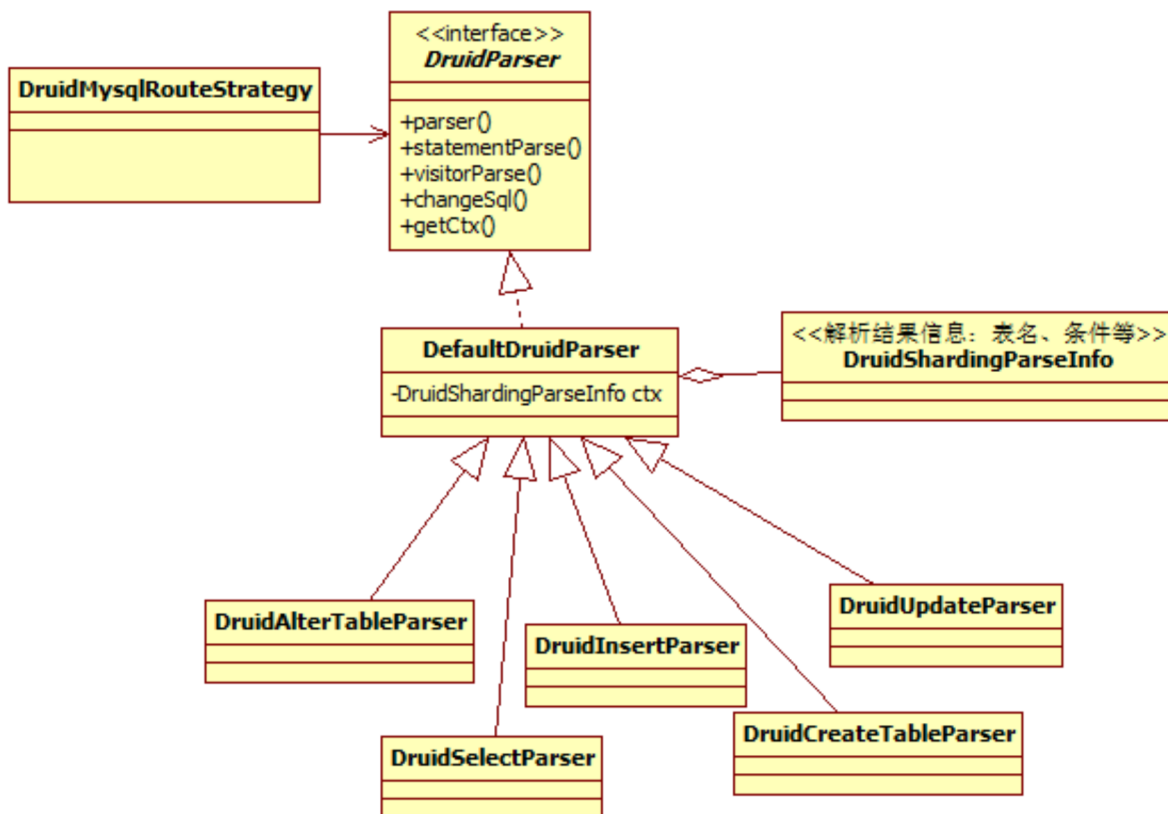
## 相关类图和序列图

### 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。

### 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。

## 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。

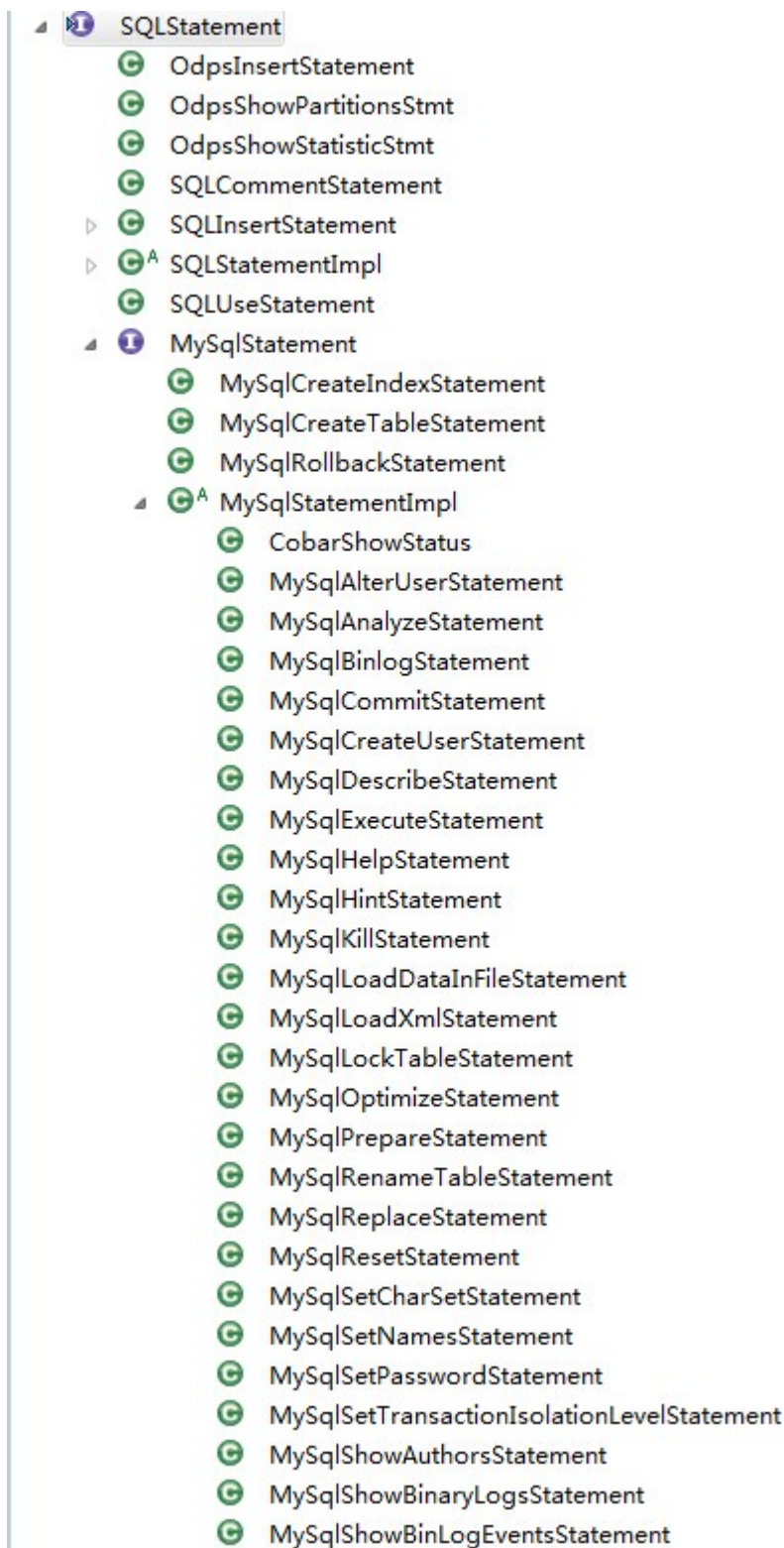


表1 DruidParser接口方法介绍

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

方法名	用途
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

## 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。

### 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;
}
```

### 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。

### 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);
}
```

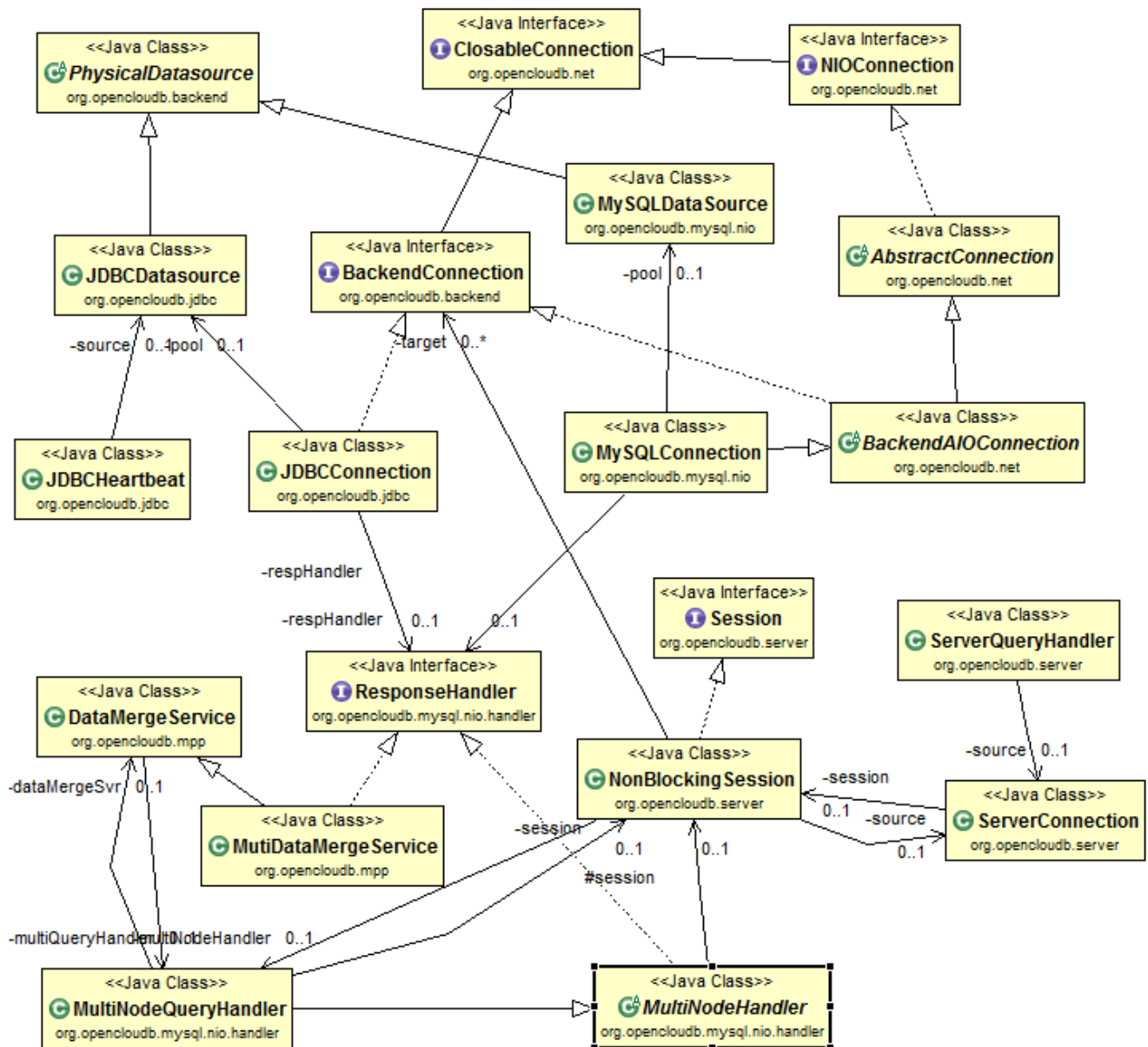
## 第七章 Mycat的JDBC后端框架

### 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

## JDBC相关类图



## JDBCDataSource

JDBCDataSource继承PhysicalDataSource

初始化的时候加载支持数据库的驱动

```
static {
    // 加载可能的驱动
    List<String> drivers = Lists.newArrayList("com.mysql.jdbc.Driver",
```



```

"org.opencloudb.jdbc.mongodb.MongoDriver", "oracle.jdbc.OracleDriver",
"com.microsoft.sqlserver.jdbc.SQLServerDriver", "org.apache.hive.jdbc.HiveDriver", "com.ibm.db2.jcc.DB2Driver", "org
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();
    JDBCConnection c = new JDBCConnection();

    c.setHost(cfg.getIp());
    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org.opencloudb.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;
}
```

## JDBCConnection

JDBCConnection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);

```

## 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);
    }
}

```

## Mycat的事务管理机制

### 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) {
            writeErrorMessage(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);
        }
    }
}
```

## Mycat的分页和跨库Join

### 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

## 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);
}
```

## 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(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 ParserException("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;
}

```

### 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();  
}
```

## 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"};
```

## 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);
```

## 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;  
}
```

## 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;
            }
        }
    }

```

## 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));
    }
}

```

## 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.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

## 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();

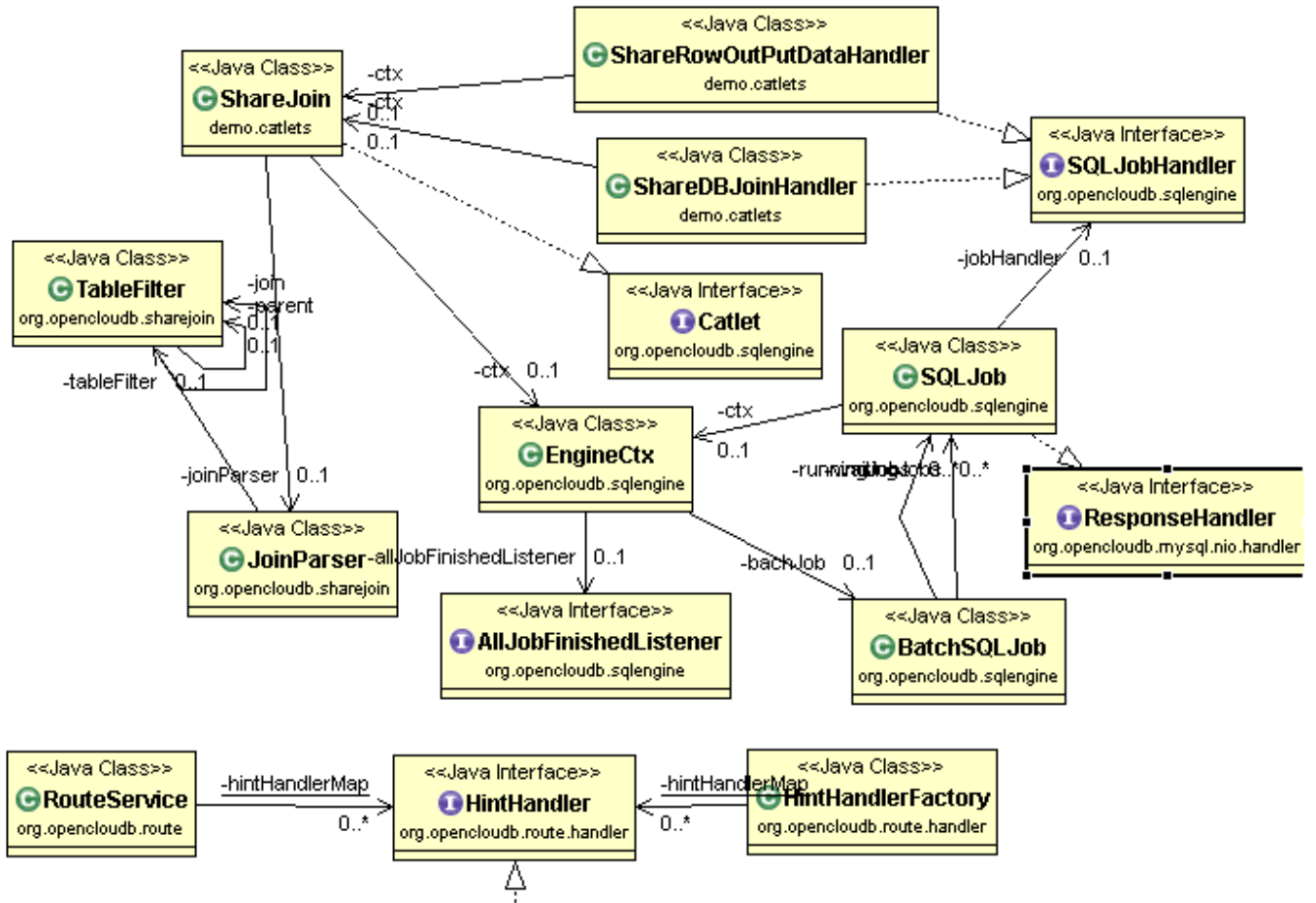
            return limitDB2(db2Stmt.getSelect(), JdbcConstants.DB2, offset, count);
        } else if (JdbcConstants.POSTGRESQL.equalsIgnoreCase(dbType))

```

## ShareJoin代码分析

### 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 ssql=joinParser.getSql(); //拆分的第一条SQL语句
    getRoute(ssql); //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().writeErrorMessage(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, ssql, joinHandler);
    EngineCtx.LOGGER.info("Catlet exec:"+getDataNode(getDataNodes())+" sql:"+ssql);
    //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 findex){
    rows.put(id, rowData);
    ids.put(id, nid);
    joinindex=findex;
    //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 ((theId = ids.poll()) != null) {
    batchRows.put(theId, rows.remove(theId));
    sb.append(theId).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);
    }
}

```

## 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
    }
}

```

```
}  
  
}
```

## 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);
}
}

```

## Mycat缓存

### 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，决定使用种缓存。

缓存的代码在org.opencloudb.cache和org.opencloudb.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 lastAcceTime;//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

## 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org.opencloudb.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;

```

## 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
SQLRouteCache	10000	0	2	0	0	1429544832439
TableID2DataNodeCache. TESTDB_CUSTOMER	10000	1	2	1	1	1429544832441
TableID2DataNodeCache. TESTDB_ORDERS	50000	0	0	0	0	0
ER_SQL2PARENTID	1000	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
SQLRouteCache	10000	0	3	0	0	1429544916936
TableID2DataNodeCache. TESTDB_CUSTOMER	10000	2	3	1	2	1429544916937
TableID2DataNodeCache. TESTDB_ORDERS	50000	0	0	0	0	0
ER_SQL2PARENTID	1000	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

org.opencloud.db.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

    String cacheKey = pair.colValue; //缓存的key是id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) {
```



```

HashSet<String>());

tablesRouteMap.put(tableName, new
}
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
                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();
    }
}

```

## 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
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 partition 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.getLocateRTTableKeySql().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 partition 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;
    }
}

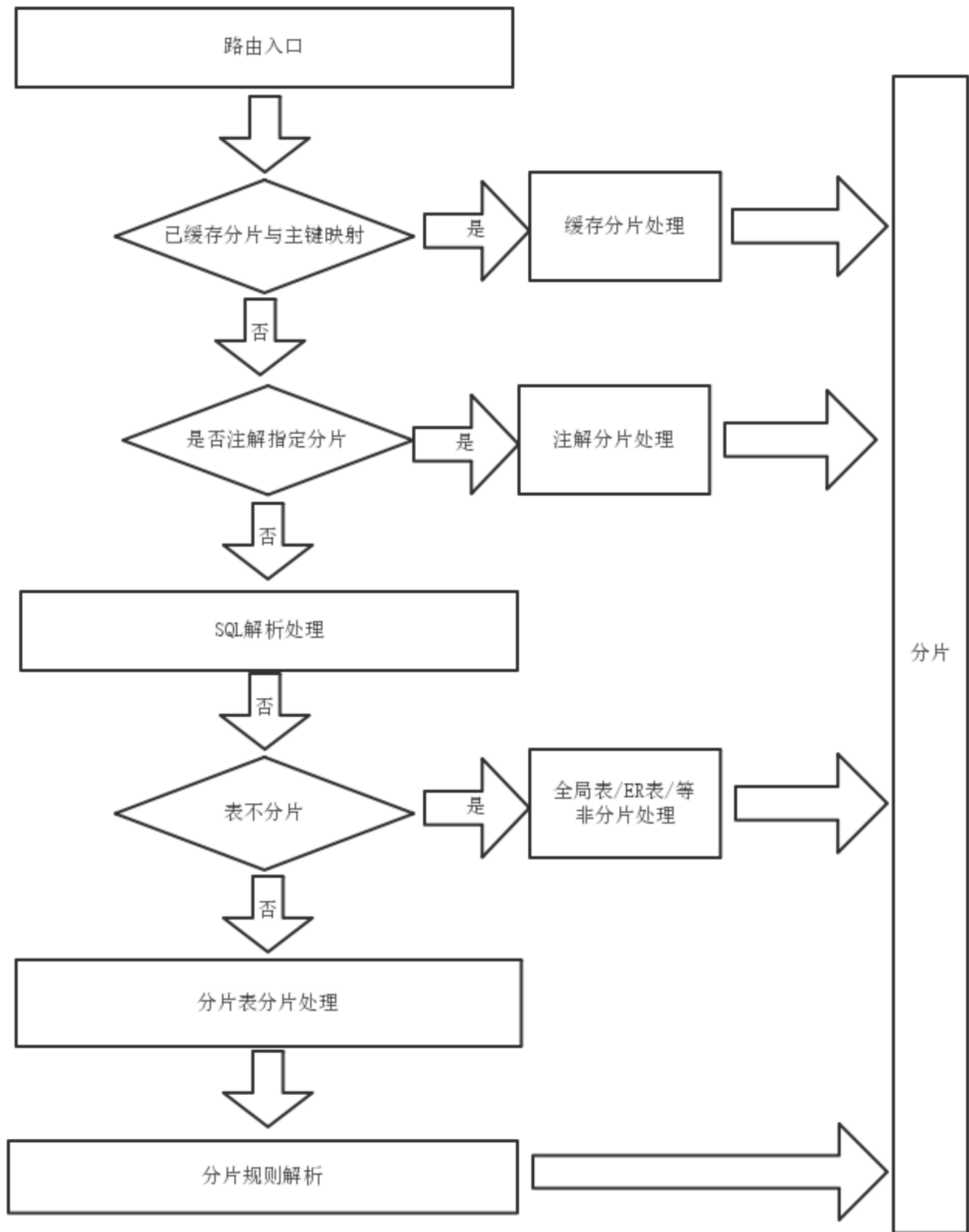
```

## Mycat 的分片规则设计

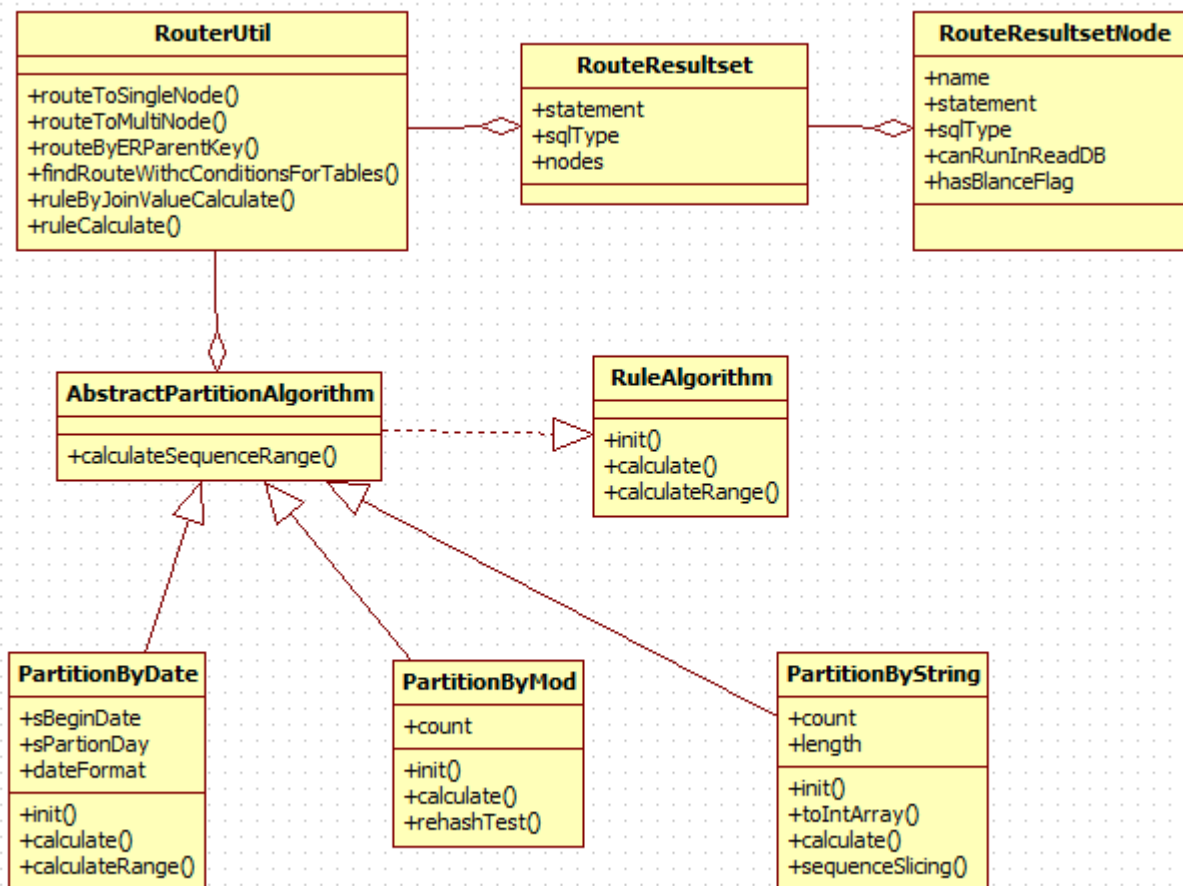
### 分片规则设计架构

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，分片规则的类图设计为：



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://org.opencloudb/">
<tableRule name="rule1">
  <rule>
    <columns>user_id</columns>
    <algorithm>func1</algorithm>
  </rule>
</tableRule>

<function name="func1" class="org.opencloudb.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。

## 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 org.opencloudb.route.function;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import org.apache.log4j.Logger;
import org.opencloudb.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 setsPartitionDay(String sPartitionDay) {
    this.sPartitionDay = sPartitionDay;
}

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;
}
```

## Mycat Load Data源码

### 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。

## 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();

    parseLoadDataParam();

    //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.writeErrorMessage(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.writeErrorMessage(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;

```

## 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;
}

```

```
}
```

## 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);
    }
}
```

## 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);
}
```

## 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;
    }
}
```

## 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 il = msg.length() - msg.position();
        int i = 0;
        if (il > 3)
        {
            i = msg.readUB3();
            msg.move(-3);
        }
        if (il < i + 4)
        {
            byte[] e = msg.readBytes(il);
            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 il = msg.length() - msg.position();
        int i = 0;
        if (il > 3)
        {
            i = msg.readUB3();
            msg.move(-3);
        }
        if (il < i + 4)
        {
            byte[] e = msg.readBytes(il);
            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;
}
```

## Mycat外传

---

### 群英会

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，公司主营文博行业、360全景、三维等。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

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。

## 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

## 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。

## 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。

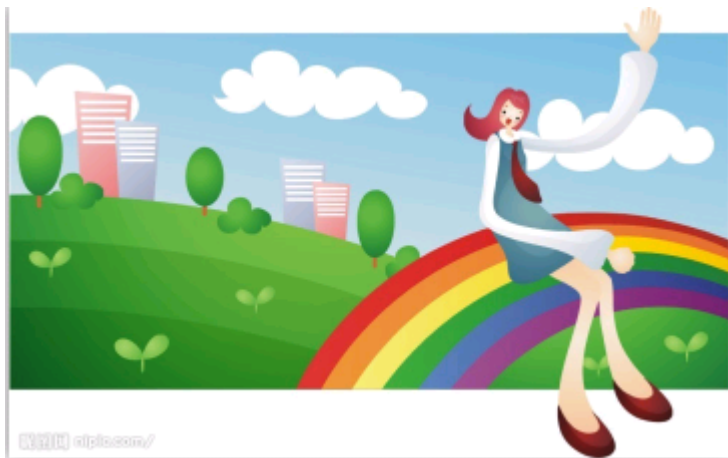
典型90后，崇拜卢梭，亚当斯密。喜欢新鲜技术，爱折腾。

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。



## Rainbow



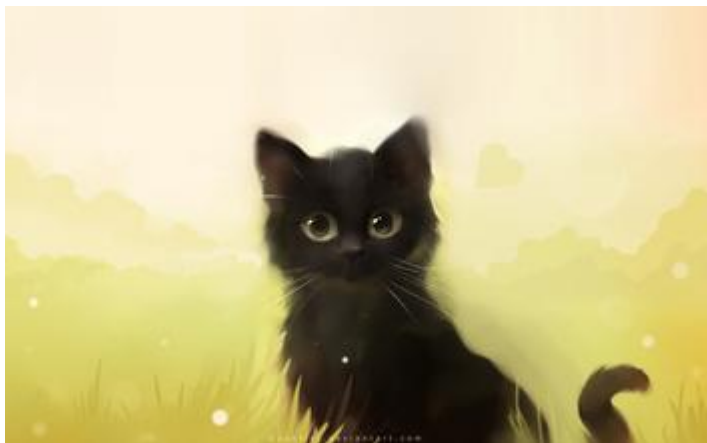
从传统企业应用管理系统入门、成长于电商系统的架构与开发实现，目前从事大型国企的私有云建设中技术架构主导。已经在JAVA Web领域工作8年有余。从coder到架构师，随着职位变化从事工作也发生很大改变。但目前依然在一线coding。因为很享受coding所带来的快乐！目前主要研究应用架构(单一服务架构、垂直应用架构、分布式应用架构、流动计算架构)、PAAS平台、应用系统监控与管理。在业余时间主导并参与Mycat-web系统开发。

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

## 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.

## 满满正能量

他写了一篇关于开源和Mycat的文章，满满的正能量，他的昵称也叫做正能量。



王金剑（正能量），网名Dreamcode，CSDN 认证专家。从事软件工作10年，曾在金融和互联网行业企业担任高级软件工程师、项目经理、高级DBA工程师职位。有银行信用卡系统和电子商务系统基础架构设计与开发经验，现在天津某跨国公司担任高

级DBA工程师，负责公司电商网站数据库设计与优化、数据库架构规划及部署规范制定、核心应用的软件设计与开发等工作。个人

博客：<http://blog.csdn.net/dreamcode>

目前为MyCat开源社区做了以下贡献：

1、撰写了《Mycat开源宣言》

2、基于Oracle和Mycat集群环境，针对Mycat对SQL92标准语句的支持情况，进行了较全面的测试。

详见文档：《SQL92标准语句Mycat支持测试（基于Oracle和MySQL）》

下载地址：<https://github.com/MyCATApache/Mycat-doc/commit/4b13dd3719e806e6ab8309d9a221676537cc66a5>

## 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。

## 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。

## 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！

## 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

.....”

## 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](http://www.micmiu.com) email: [sjsky007@gmail.com](mailto:sjsky007@gmail.com) weibo: <http://weibo.com/ctosun>



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。

## 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。

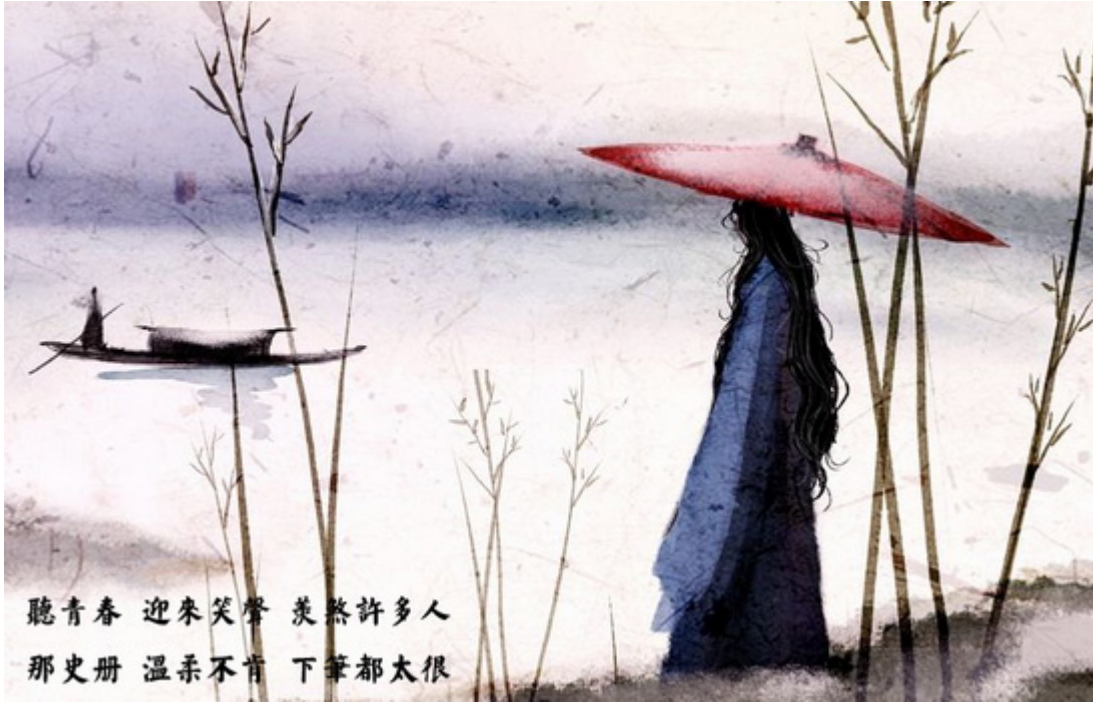
他忽悠出了一个Mycat开源社区，然后这个社区成为国内大数据编程领域最有实力的社区，这里有颜值很高的新锐小清新，也有深藏不露的资深架构师，如果有一天风头慕名而来，你也不用诧异，因为你可能也是被风头看中的一员大将。



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

## 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

## 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~



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