# Mycat

## 主要内容

1. mycat简介
2. mysql主从搭建
3. mysql的集群搭建
4. mycat的安装
5. mycat名词解释
6. mycat server.xml的配置
7. mycat schema.xml的配置
8. mycat分片详解之枚举
9. mycat分片详解之取模
10. mycat分片详解之auto-sharding-long
11. mycat分片详解之固定分片hash算法
12. mycat分片详解之字符串ID处理
13. mycat全局表和普通表的配置和测试
14. mycatER表的配置和测试
15. mycat全局ID的概述
16. mycat全局主键的生成方式
17. 雪花算法

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| mycat简介 | 熟悉 |
| mysql主从搭建 | 掌握 |
| mysql的集群搭建 | 掌握 |
| mycat的安装 | 掌握 |
| mycat名词解释 | 掌握 |
| mycat server.xml的配置 | 掌握 |
| mycat schema.xml的配置 | 熟悉 |
| mycat分片详解之枚举 | 掌握 |
| mycat分片详解之取模 | 掌握 |
| mycat分片详解之auto-sharding-long | 掌握 |
| mycat分片详解之固定分片hash算法 | 掌握 |
| mycat分片详解之字符串ID处理 | 掌握 |
| mycat全局表和普通表的配置和测试 | 掌握 |
| mycatER表的配置和测试 | 掌握 |
| mycat全局ID的概述 | 掌握 |
| mycat全局主键的生成方式 | 掌握 |
| 雪花算法 | 掌握 |

# 【熟悉】mycat简介

## 什么是MyCat

  MyCat是目前最流行的分布式数据库中间插件，是一个开源的分布式数据库系统，是一个实现了MySQL协议的服务器，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。

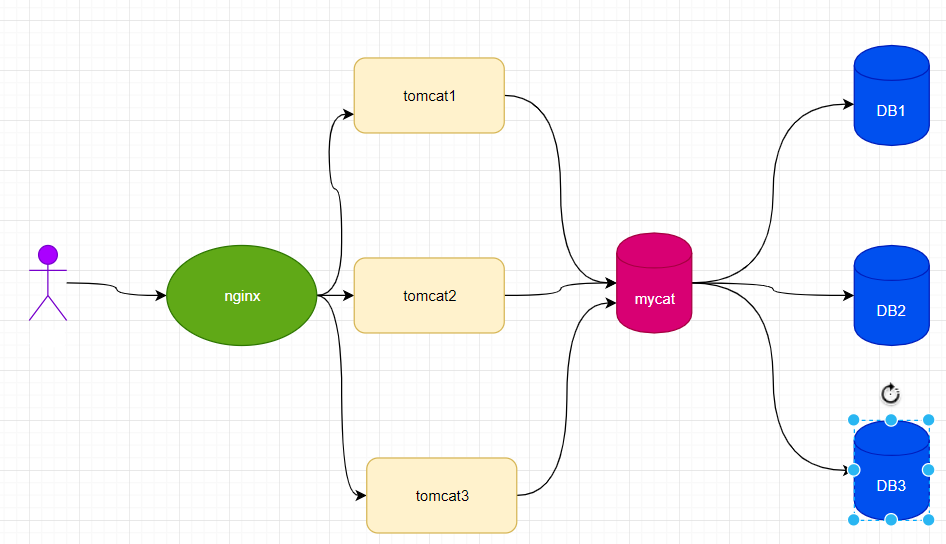
MyCat发展到目前的版本，已经不是一个单纯的MySQL代理了，它的后端可以支持MySQL、SQL Server、Oracle、DB2、PostgreSQL等主流数据库，也支持MongoDB这种新型NoSQL方式的存储，未来还会支持更多类型的存储。而在最终用户看来，无论是那种存储方式，在MyCat里，都是一个传统的数据库表，支持标准的SQL语句进行数据的操作，这样一来，对前端业务系统来说，可以大幅降低开发难度，提升开发速度



更多解释见官网：<http://www.mycat.io/> http://www.mycat.org.cn/

项目地址：<https://github.com/MyCATApache/Mycat-Server>

## 为什么使用MyCat



  如今随着互联网的发展，数据的量级也是撑指数的增长，从GB到TB到PB。对数据的各种操作也是愈加的困难，传统的关系性数据库已经无法满足快速查询与插入数据的需求。这个时候NoSQL的出现暂时解决了这一危机。它通过降低数据的安全性，减少对事务的支持，减少对复杂查询的支持，来获取性能上的提升。但是，在有些场合NoSQL一些折衷是无法满足使用场景的，就比如有些使用场景是绝对要有事务与安全指标的。这个时候NoSQL肯定是无法满足的，所以还是需要使用关系性数据库。如何使用关系型数据库解决海量存储的问题呢？此时就需要做数据库集群，为了提高查询性能将一个数据库的数据分散到不同的数据库中存储，为应对此问题就出现了——MyCat

综上所述：Mycat作用为：能满足数据库数据大量存储；提高了查询性能;实现读写分离，分库分表，如下

1：解决表的容量问题

   Mysql的表最大存储多少数据？500w条

2：解决表查询的性能问题？log(n)<https://www.cnblogs.com/glzgc/p/10831877.html>

  我的表里面的数据非常多，导致我查询很慢？

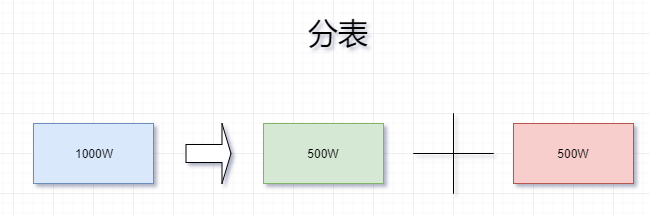
3：解决表的DML（数据库操作语言）的性能问题？

 2000条/s

4：解决mysql 挂了的问题

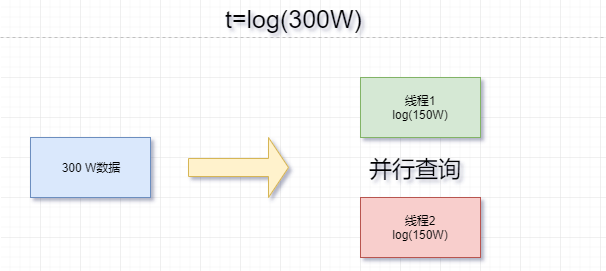
## MyCat解决问题的思路

### 容量

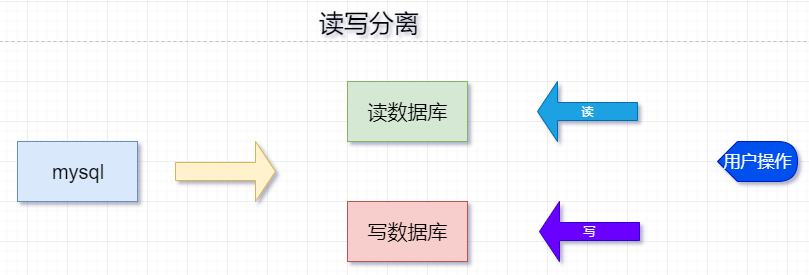


### 查询速度

最快也是O(Logn),你加什么索引都不行



### 写入问题



## Mycat和MySQL的区别。

       我们可以把上层(mycat)看作是对下层(mysql)的抽象，例如操作系统是对各类计算机硬件的抽象。那么我们什么时候需要抽象？假如只有一种硬件的时候，我们需要开发一个操作系统吗？再比如一个项目只需要一个人完成的时候不需要leader，但是当需要几十人完成时，就应该有一个管理者，发挥沟通协调等作用，而这个管理者对于他的上层来说就是对项目组的抽象。同样的，当我们的应用只需要一台数据库服务器的时候我们并不需要Mycat，而如果你需要分库甚至分表，这时候应用要面对很多个数据库的时候，这个时候就需要对数据库层做一个抽象，来管理这些数据库，而最上面的应用只需要面对一个数据库层的抽象或者说数据库中间件就好了，这就是Mycat的核心作用。所以可以这样理解：数据库是对底层存储文件的抽象，而Mycat是对数据库的抽象。

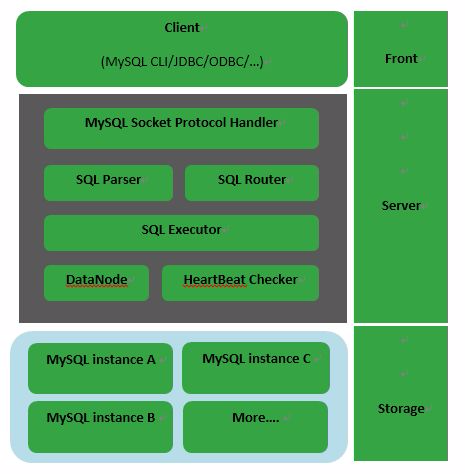
## 支持的数据库

  mysql，sqlserver，mongoDB等



## 架构

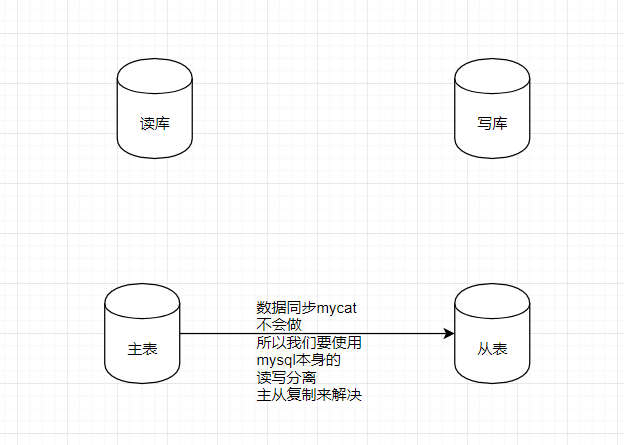
  前端用户可以把mycat看作是一个数据库代理，用mysql客户端工具（如Navicat）和命令访问，而后端支持mysql，sqlserver，oracle等主流数据库，用mysql native 协议和多个mysql服务器通信，也可用JDBC协议与大多数主流数据库服务器通信，其核心功能就是分库分表，即将一个大表水平分割为N个小表，真正的存储在后端Mysql服务器中或其它数据库中



  mycat原理：可以用“拦截”一词形容，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处理，最终返回给用户

# 【熟悉】mysql主从搭建

因为**mycat只能路由，分发，不能把多个数据库里面的数据进行同步**，所以要数据同步必做还要使用**mysql的读写分离，主从复制**



## 概述主从介绍

Mysql主从又叫Replication、AB复制。简单讲就是A与B两台机器做主从后，在A上写数据，另外一台B也会跟着写数据，实现数据实时同步

mysql主从是基于**binlog**，主上需开启binlog才能进行主从

主从过程大概有3个步骤

**主库将更改操作记录到binlog里**

**从库将主库的binlog事件（sql语句） 同步本机上并记录在relaylog里**

**从根据relaylog里面的sql语句按顺序执行**

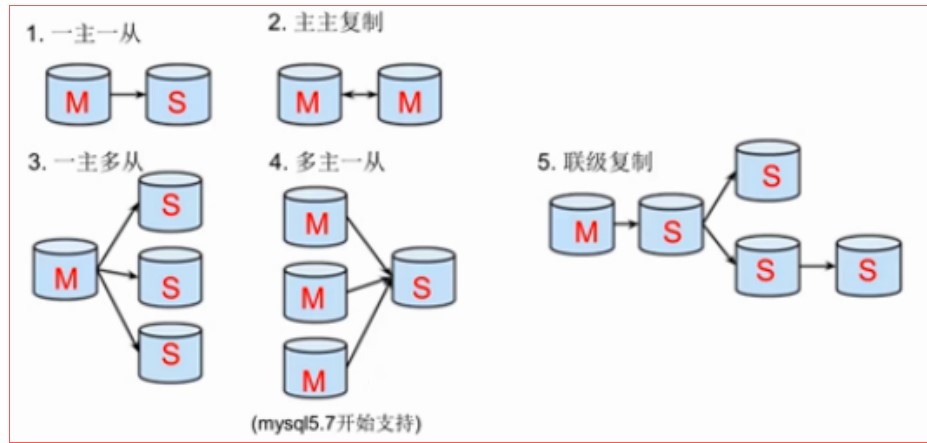
## 主从作用

**实时灾备，用于故障切换**

**读写分离，提供查询服务**

**备份，避免影响业务**

主从形式



\* 一主一从

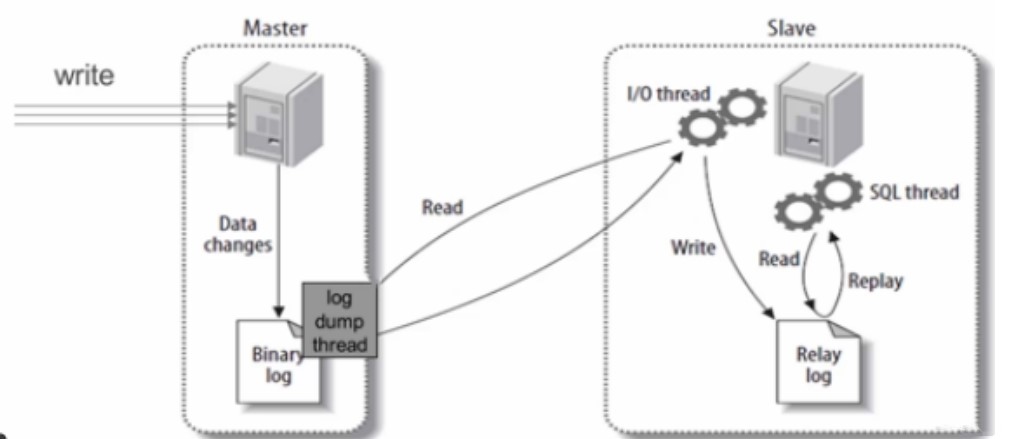
\* 主主复制

\* 一主多从---扩展系统读取的性能，因为读是在从库读取的

\* 多主一从---5.7版本开始支持

\* 联级复制

## 主从复制原理



## 主从复制步骤

①当Master节点进行insert、update、delete操作时，会按顺序写入到binlog中。

②salve从库连接master主库，Master有多少个slave就会创建多少个binlog dump线程。

③当Master节点的binlog发生变化时，binlog dump 线程会通知所有的salve节点，并将相应的binlog内容推送给slave节点。

④I/O线程接收到 binlog 内容后，将内容写入到本地的 relay-log。

⑤SQL线程读取I/O线程写入的relay-log，并且根据 relay-log 的内容对从数据库做对应的操作

## 主从复制配置步骤

1. 确保从数据库与主数据库里的数据一致
2. 在主数据库里创建一个同步账户授权给从数据库使用
3. 配置主数据库（修改配置文件）
4. 配置从数据库（修改配置文件）

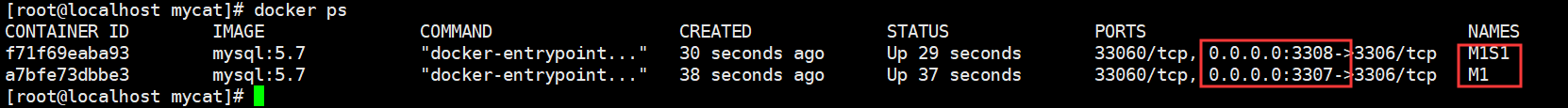
需求

搭建两台MYSQL服务器，一台作为主服务器，一台作为从服务器，主服务器进行写操作，从服务器进行读操作

## 环境说明【使用docker启动两个】

|  |  |  |
| --- | --- | --- |
| 名称 | Ip | Port |
| M1 | 192.168.149.128 | 3307 |
| M1S1 | 192.168.149.128 | 3308 |





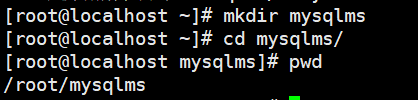
|  |
| --- |
| docker run --name M1 -p 3307:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |
| docker run --name M1S1 -p 3308:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |

**注意开放阿里云的端口(本地注意关闭防方墙并重启docker)**

## 修改配置文件

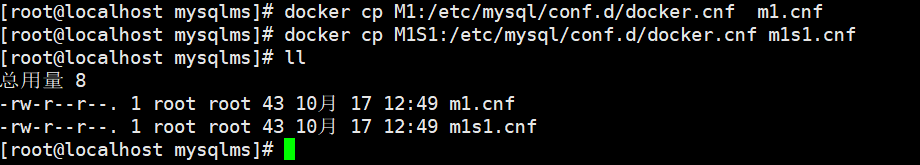
**将容器里面的配置文件复制出来,主要修改服务器的配置**

在root目录下创建一个mysqlms的目录存放从docker容器里面复制过来的配置文件



docker cp M1:/etc/mysql/conf.d/docker.cnf m1.cnf

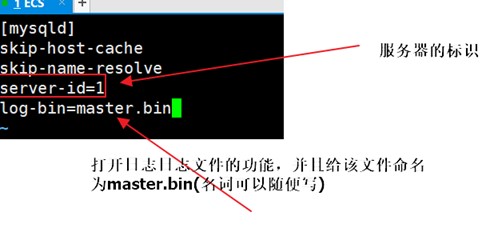
docker cp M1S1:/etc/mysql/conf.d/docker.cnf m1s1.cnf



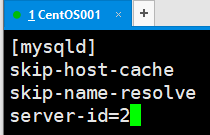
### 主机的配置m1.cnf

主机里面要记录sql 语句，以后从机会把该sql 语句偷过去

|  |
| --- |
| # 开启binlog  log-bin=master.bin  server-id=1  # 需要同步的数据库，如果不配置则同步全部数据库  binlog-do-db=test\_db  # binlog日志保留的天数，清除超过10天的日志  # 防止日志文件过大，导致磁盘空间不足  expire-logs-days=10 |



### 从机的配置m1s1.cnf



### 配置文件修改后，复制到容器里面

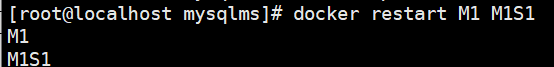
docker cp m1.cnf M1:/etc/mysql/conf.d/docker.cnf

docker cp m1s1.cnf M1S1:/etc/mysql/conf.d/docker.cnf

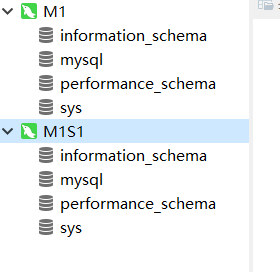


重启mysql（m1,m1s1）

docker restart M1 M1S1

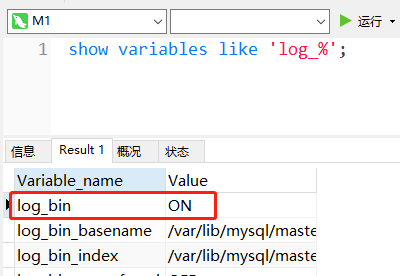


测试连接



查看M1的binlog功能是否开启

show variables like 'log\_%';

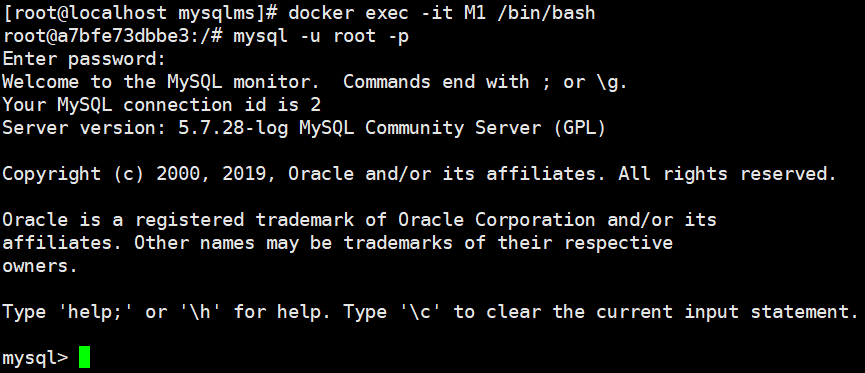


## 执行sql语句

### 进入主机里面执行相关配置

docker exec -it M1 bash

mysql -uroot -p123456



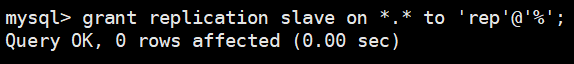
**创建用户**

**create user 'rep'@'%' identified by '123456';**



**给该用户授予权限：**

**grant replication slave on \*.\* to 'rep'@'%';**



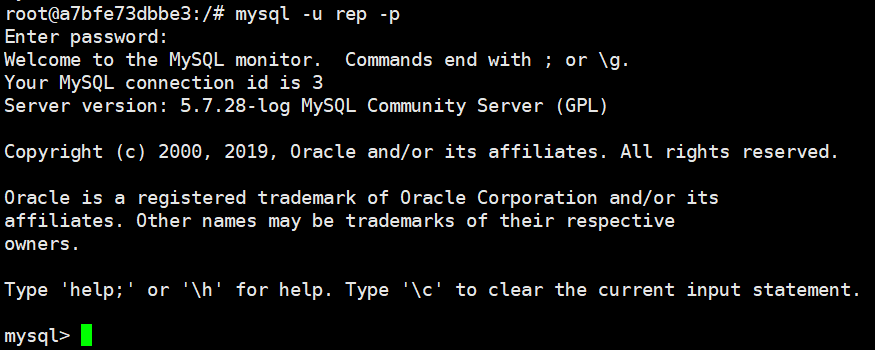
**刷新权限**

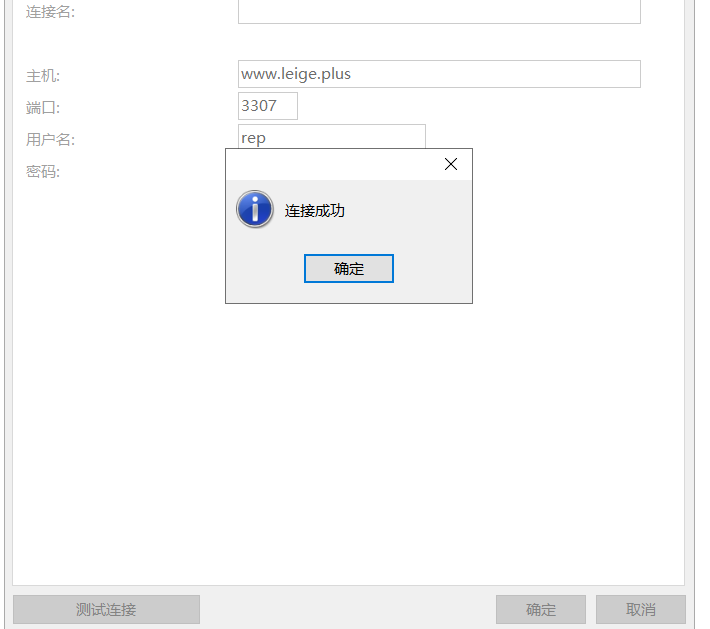
**flush privileges;**



 至此：M1 里面已经创建了一个用户：rep 123456 拥有所以库，所有表replication slave

尝试使用M1 里面的rep 用户登录：

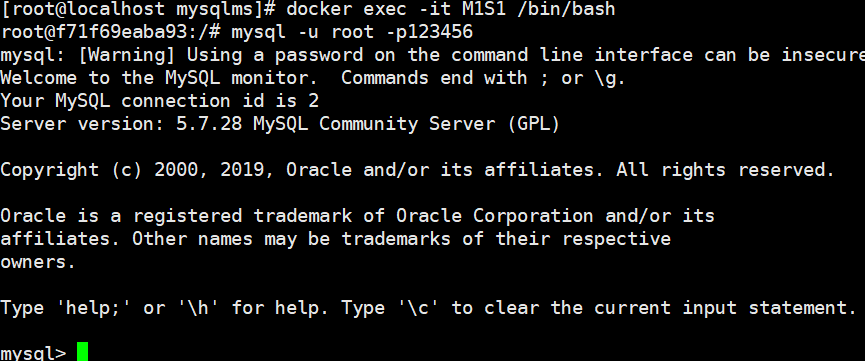




### 进入从机里面执行相关配置

docker exec -it M1S1 bash

mysql -u root -p123456



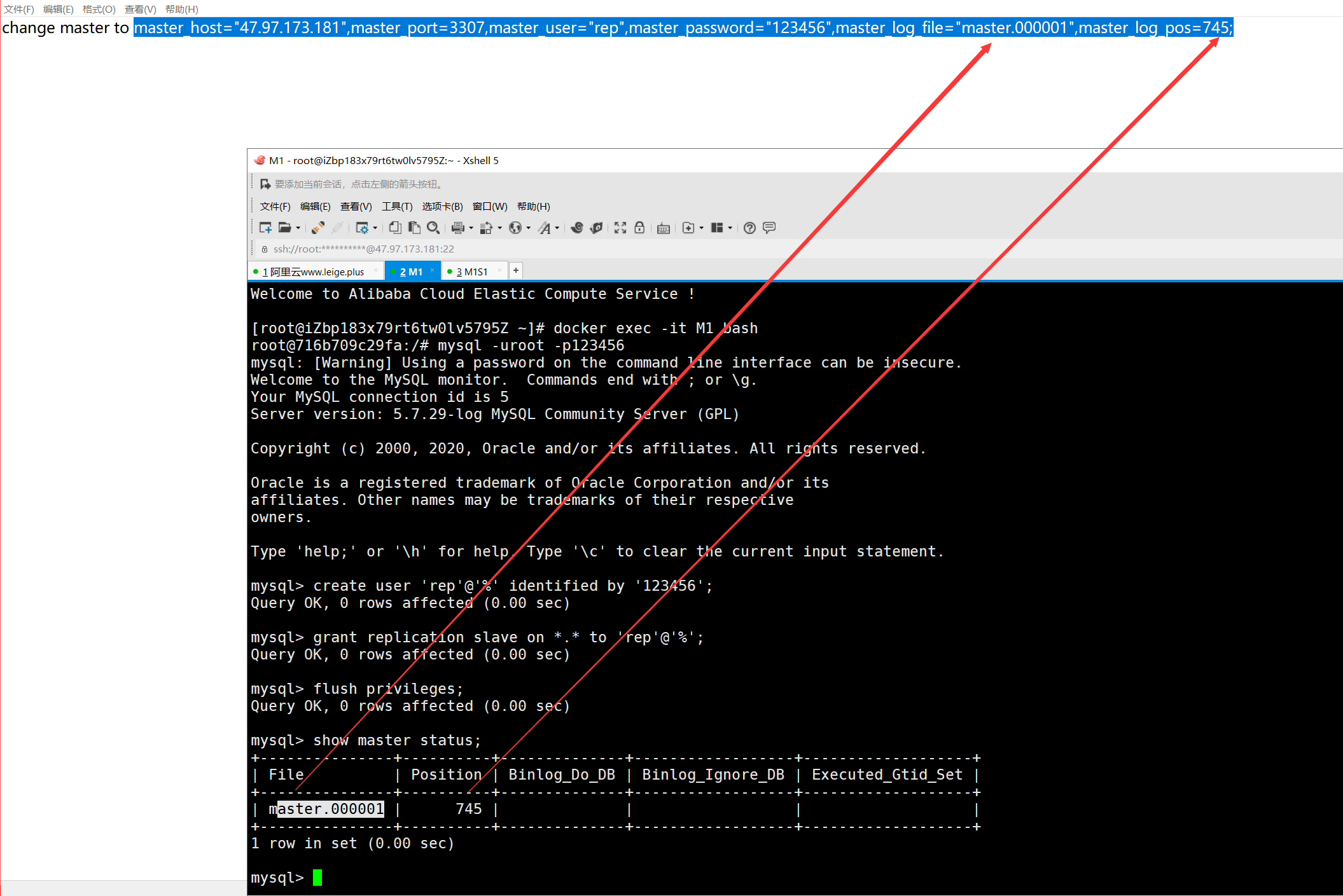
change master to master\_host="",master\_port=,master\_user="",master\_password="",master\_log\_file="",master\_log\_pos=

master\_log\_file：该文件具体叫什么名称，需要从主机里面去看看：

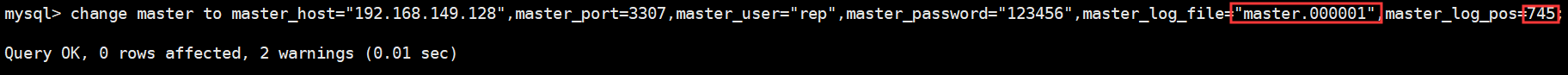
进入M1 里面使用root 用户登录M1，执行下面的sql：

show master status;





修改上面的SQL执行



change master to master\_host="**47.97.173.181**",master\_port=**3307**,master\_user="**rep**",master\_password="**123456**",master\_log\_file="**master.000001**",master\_log\_pos=**745**;

**启动主从：（在M1S1里面执行）**

**start slave ;**

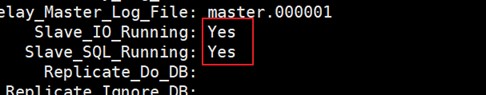


**查询主从的状态（M1S1）**

**show slave status \G;**



**成功的标志：**



## 搭建失败的原因

### 第一个不是yes,是connecting

是因为从机使用你配置的主机信息没有登陆到主机里面！

修改(从机里面)

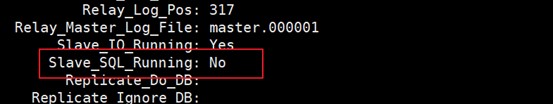
stop slave;

change master to master\_host="192.168.149.128",master\_port=3307,master\_user="rep",master\_password="123456",master\_log\_file="master.000001",master\_log\_pos=745;

start slave;

### 第二个不是yes ，是no

原因是主机和从机里的数据不一致



从机会复制主机里面的sql 语句，来自己执行！

实验时先把从机里面的db3 删除

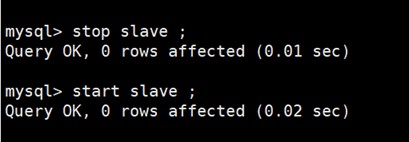
再把主机里面的db3 删除->从机里面复制该删除的命令->从机执行删除的命令（db3），事务无法提交，将一直阻塞！



现在从机里面要删除db3 ，但是没有db3，导致一直阻塞，以后的主从复制不会进行了

解决：在从机新建一个db3：

然后停止主从，启动主从：



### 第一个不是yes，是no

就是你的server-id 没有配置成功的原因，需要重新修改配置文件，复制配置文件到容器里面，然后重启就ok

### mysql主从的操作规范

**1 只能在主机里面执行DML 语句，不能在从机里面执行DML语句【会破坏主从】**

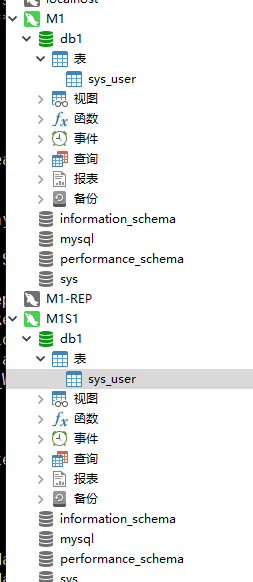
**2 在从机里面可以执行查询语句**

**3 主机只有一台，但是从机可以有多台**

## 测试

在M1 里面创建数据库

看M1S1 有没有复制过去



# 【掌握】mysql的集群搭建

## 集群搭建概述

### 是什么

 集群（cluster）技术是一种较新的技术，通过集群技术，可以在付出较低成本的情况下获得在性能、可靠性、灵活性方面的相对较高的收益，其任务调度则是集群系统中的核心技术。

        MySQL集群技术在分布式系统中为MySQL数据提供了冗余特性，增强了安全性，使得单个MySQL服务器故障不会对系统产生巨大的负面效应，系统的稳定性得到保障。

### 优点

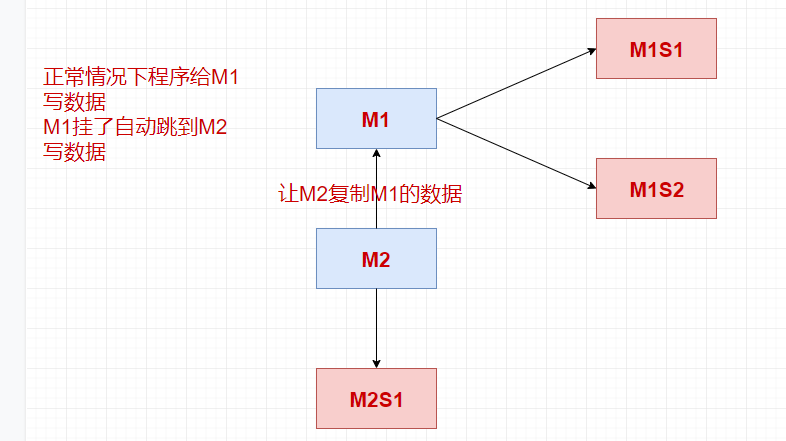
**高可伸缩性**：服务器集群具有很强的可伸缩性。 随着需求和负荷的增长，可以向集群系统添加更多的服务器。在这样的配置中，可以有多台服务器执行相同的应用和数据库操作。

**高可用性**：在不需要操作者干预的情况下，防止系统发生故障或从故障中自动恢复的能力。通过把故障服务器上的应用程序转移到备份服务器上运行，集群系统能够把正常运行时间提高到大于99.9%，大大减少服务器和应用程序的停机时间。

### 缺点

        我们知道集群中的应用只在一台服务器上运行，如果这个应用出现故障，其它的某台服务器会重新启动这个应用，接管位于共享磁盘柜上的数据区，进而使应用重新正常运转。我们知道整个应用的接管过程大体需要三个步骤：侦测并确认故障、后备服务器重新启动该应用、接管共享的数据区。因此在切换的过程中需要花费一定的时间，原则上根据应用的大小不同切换的时间也会不同，越大的应用切换的时间越长。

## 搭建思路图



## 搭建准备工作

**在搭建mysql的集群时，先清空所有机器里面的数据**

**现在有数据的时：M1 M1S1,只需要清空M1的数据，就都没有了**

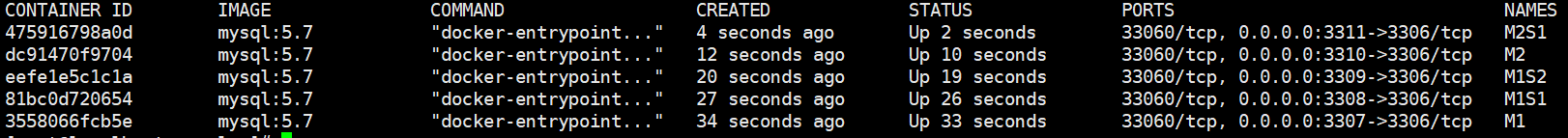
**将之前新建的db1 db2 删除就ok**

### 使用docker启动5台Mysql[至少要5台哦]

|  |  |  |
| --- | --- | --- |
| 名称 | Ip | Port |
| M1 | 192.168.149.128 | 3307 |
| M1S1 | 192.168.149.128 | 3308 |
| M1S2 | 192.168.149.128 | 3309 |
| M2 | 192.168.149.128 | 3310 |
| M2S1 | 192.168.149.128 | 3311 |

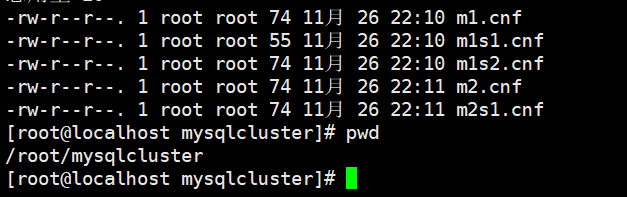
前面主从已启动了两个，所以只用启动三个就够了

|  |
| --- |
| docker run --name M1 -p 3307:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |
| docker run --name M1S1 -p 3308:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |
| docker run --name M1S2 -p 3309:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |
| docker run --name M2 -p 3310:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |
| docker run --name M2S1 -p 3311:3306 -e MYSQL\_ROOT\_PASSWORD=123456 -d mysql:5.7 --lower\_case\_table\_names=1 |



### 配置文件的修改

先准备5 个配置文件



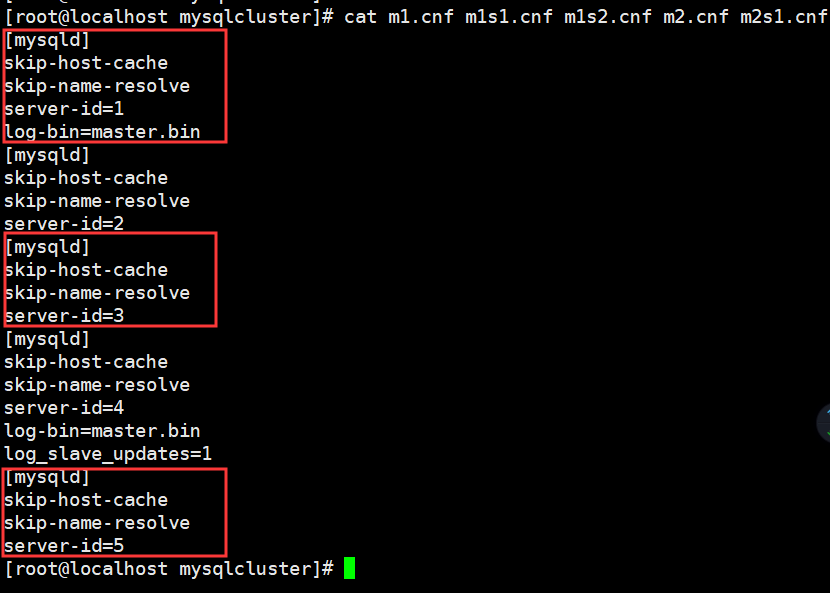
分别修改  因为在上面的主从里面已有两个配置文件，所以只用创建三个就够了m1s2.cnf   m2.cnf     m2s1.cnf

M2 需要添加一个配置项：

M2 会从M1 复制数据，但是M2 从M1复制的数据，不会记录下来，则M2S1 里面没有数据！

我们需要打开M2的级联复制功能，让M2 也能记录从M1 里面复制的数据

最后结果



配置文件修改好了后，复制到容器里面，并且重启容器！

#docker cp m1.cnf M1:/etc/mysql/conf.d/docker.cnf

#docker cp m1s1.cnf M1S1:/etc/mysql/conf.d/docker.cnf

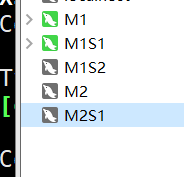
docker cp m1s2.cnf M1S2:/etc/mysql/conf.d/docker.cnf

docker cp m2.cnf M2:/etc/mysql/conf.d/docker.cnf

docker cp m2s1.cnf M2S1:/etc/mysql/conf.d/docker.cnf

### 重启新建的三个容器

docker restart M1S2 M2 M2S1



## 进入容器执行SQL

**4.1，M1修改**

不动

**4.2，M1S1修改**

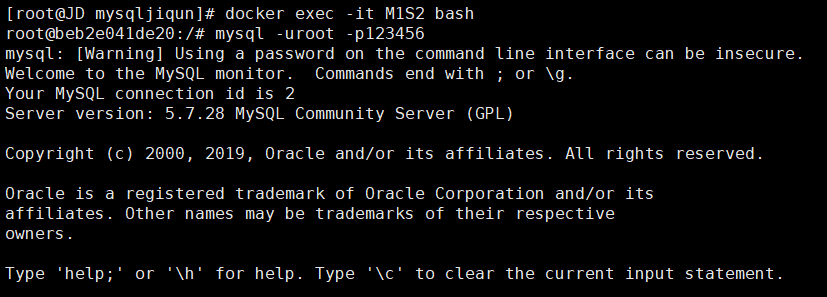
不动

**4.3，M1S2修改（从机）**

给他设置一个主机就ok

docker exec -it M1S2 bash

mysql -uroot -p123456

****

关联主机

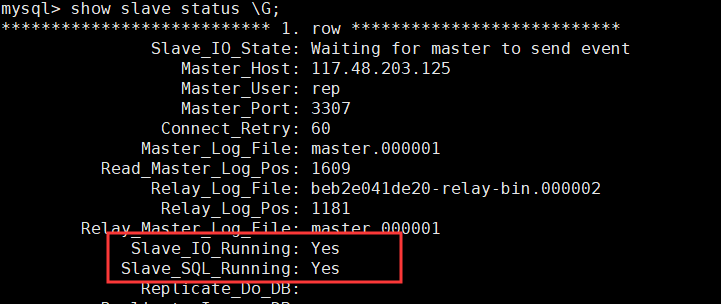
change master to master\_host="47.97.173.181",master\_port=3307,master\_user="rep",master\_password="123456",master\_log\_file="master.000001",master\_log\_pos=745;

启动主从

start slave ;

查看状态

show slave status \G;



**4.4，M2修改**

进入M2使用Root登陆

docker exec -it M2 bash

mysql -uroot -p123456

1 新建用户

create user 'rep1'@'%' identified by '123456';

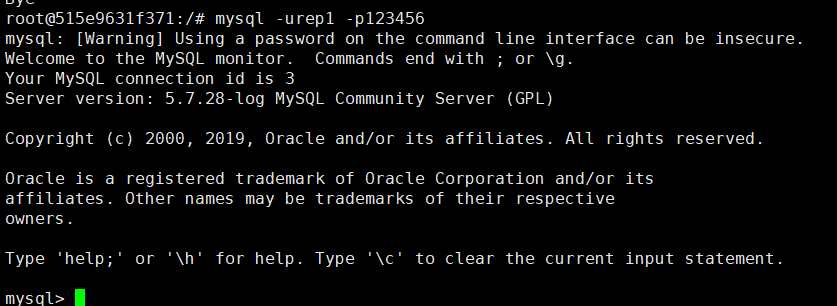
2 给用户授权

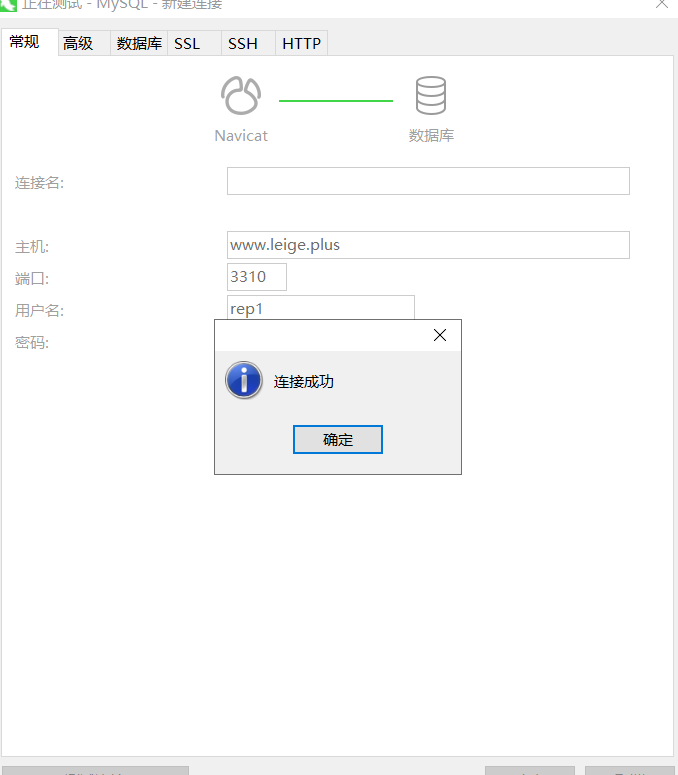
grant replication slave on \*.\* to 'rep1'@'%';

3 刷新权限

flush privileges;

4 使用新的用户尝试登录



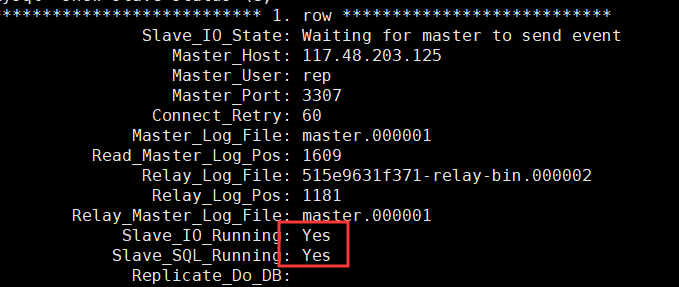


M2 做为M1的从机需要执行的sql：（**登录m2时要使用root用户**）

change master to master\_host="47.97.173.181",master\_port=3307,master\_user="rep",master\_password="123456",master\_log\_file="master.000001",master\_log\_pos=745;

start slave ;

show slave status \G;

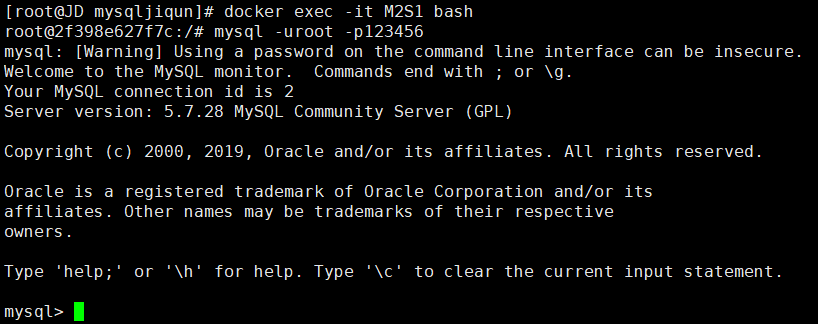


**4.5，M2S1修改**

进入容器

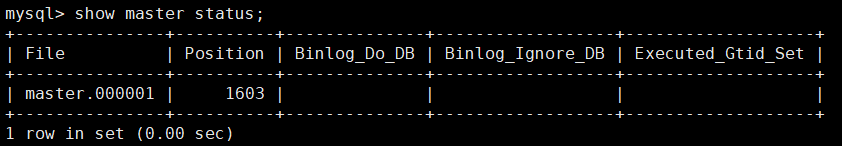
docker exec -it M2S1 bash

mysql -uroot -p123456



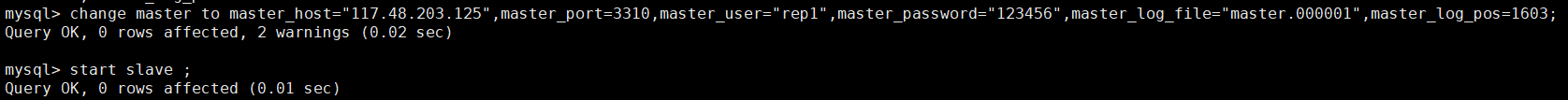
M2S1 是M2的从机，执行的sql

进入M2查看pos

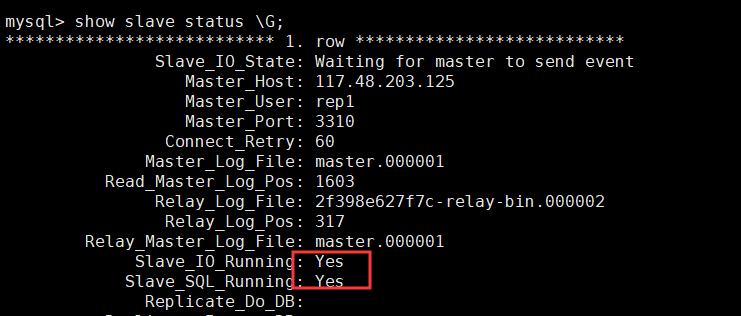


change master to master\_host="47.97.173.181",master\_port=3310,master\_user="rep1",master\_password="123456",master\_log\_file="master.000001",master\_log\_pos=1603;

start slave ;

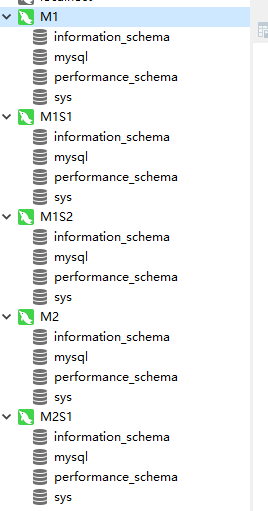


show slave status \G;



## 证集群是否成功

使用工具连接上所有的mysql

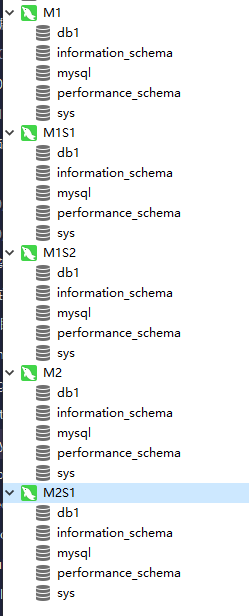


往M1 写数据 看 M2S1

在M1 新建数据库

观察M2S1

往M1 写数据，看M1S1



# 【掌握】mycat的安装

## 前言

使用mycat要安装JDK.不会的去看Linux里面的安装JDK的知识点，这是不再做说明

也可以直接使用yum install java-1.7.0-openjdk 因为mycat 基于jdk1.7开发的，所有最好安装jdk1.7的版本

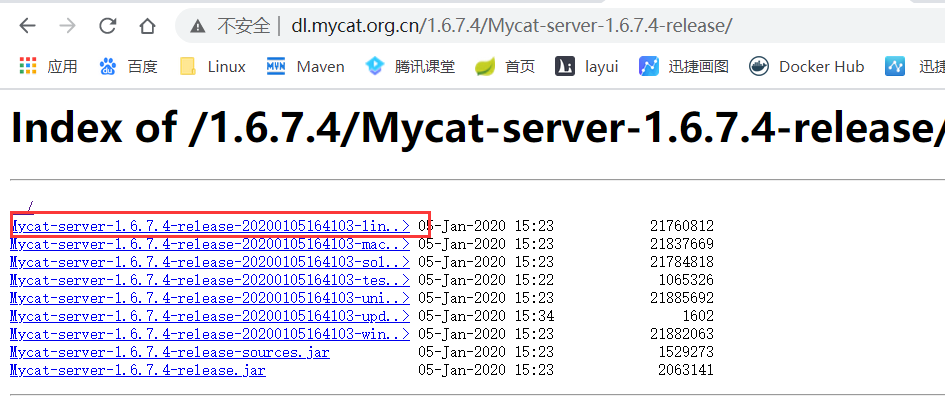
**重要说明[Mycat-server-1.6-release 版本发布](http://dl.mycat.io/1.6-RELEASE/" \t "_blank)的版本是基于jdk1.7开发的，所在要安装JDK1.7**

**[Mycat-server-1.6.7.1-release 版本发布](http://dl.mycat.io/1.6.7.1/" \t "_blank)     更新成了JDK1.8所以要安装JDK1.8下面的教程是使用JDK1.8**

## 下载压缩包或者使用wget命令

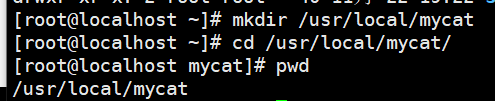
<http://www.mycat.io/>





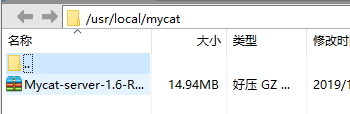
## 上传到Linux并解压压缩包

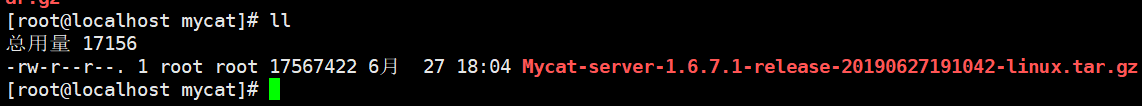
在linux里面创建目录



使用命令下载【我这里选择直接上传】

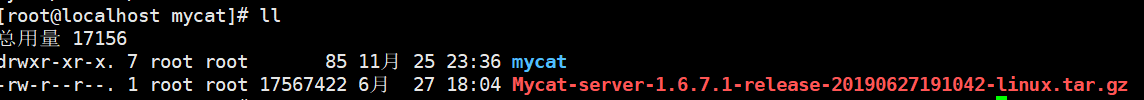
wget <http://dl.mycat.org.cn/1.6.7.4/Mycat-server-1.6.7.4-release/Mycat-server-1.6.7.4-release-20200105164103-linux.tar.gz>



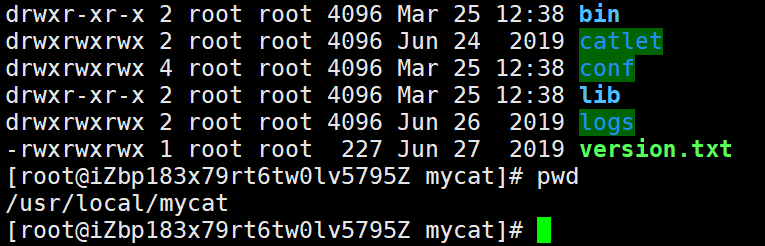


解压

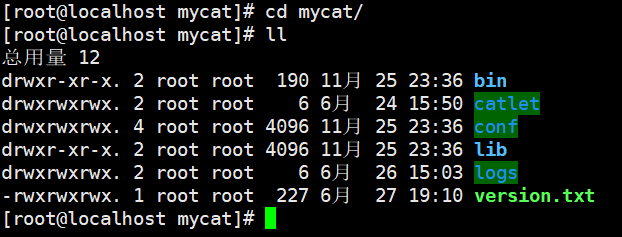




移动到usr/local/



## Mycat的目录结构



bin 执行命令的目录

conf 配置文件

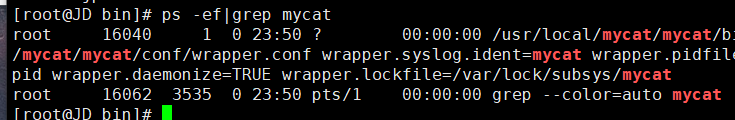
lib  依赖包

logs 日志包

## 启动Mycat

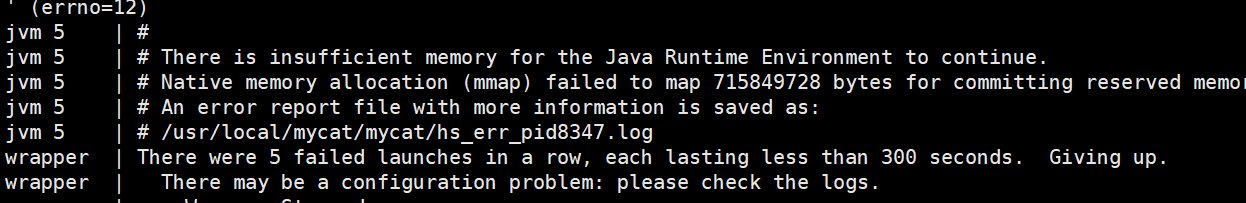
cd bin

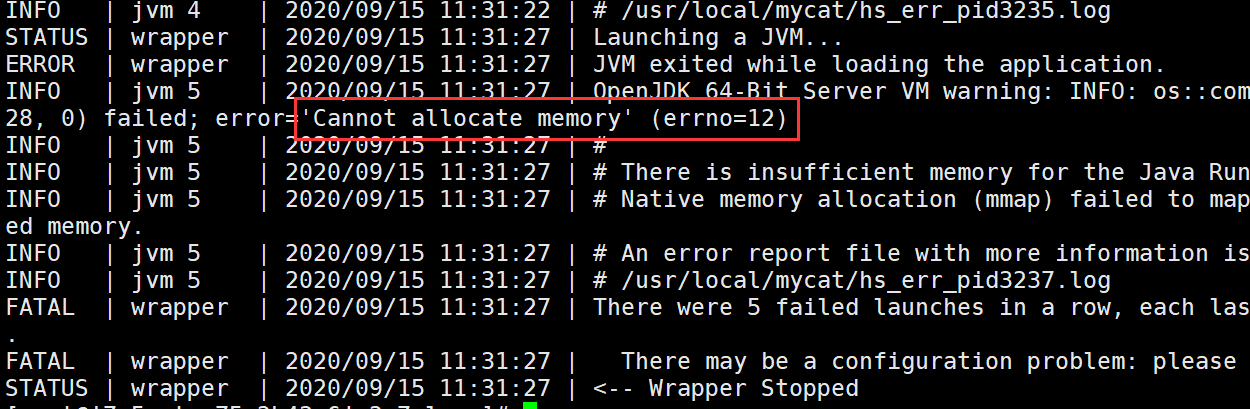
**./mycat start**|restart|stop|status|console

****

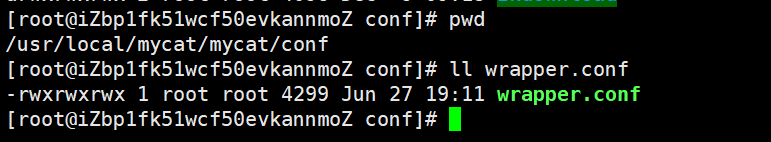
如果启动不了查看自己的JDK是不是1.8的

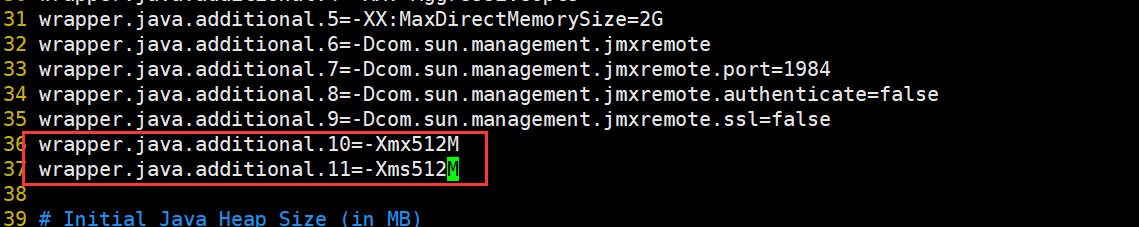
### 查看logs/wrapper.log文档看错



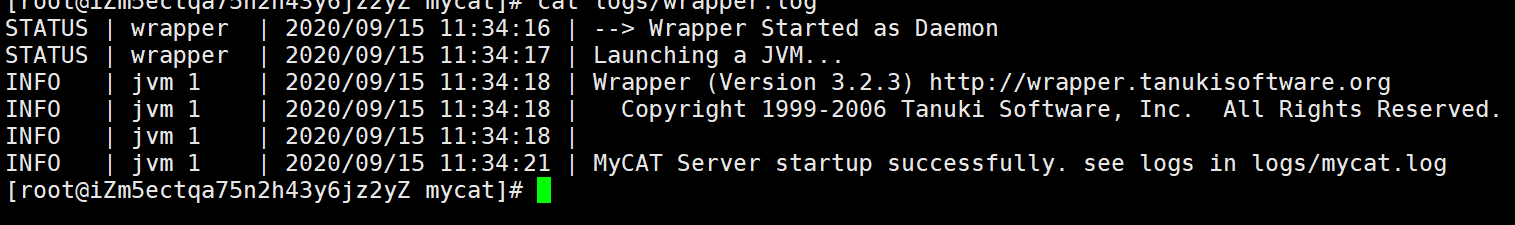


### 以上的内存不足的问修改conf/wrapper.conf





如果换成512M还不行，请换成256再启动



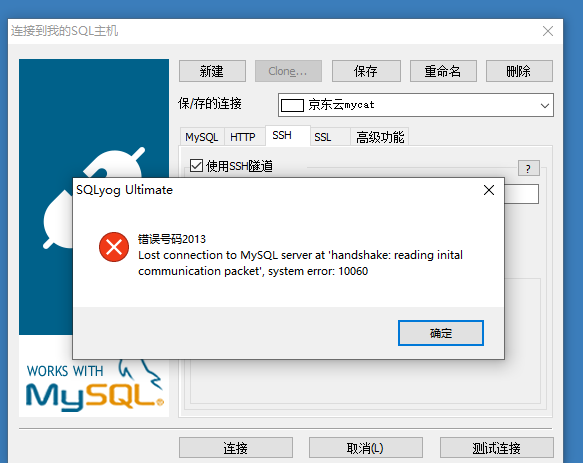
## 连接Mycat

我们在外面看Mycat，认为Mycat 就是一个Mysql，怎么连接Mysql？

1 Navicat for MySQL 对Mycat的支持查询

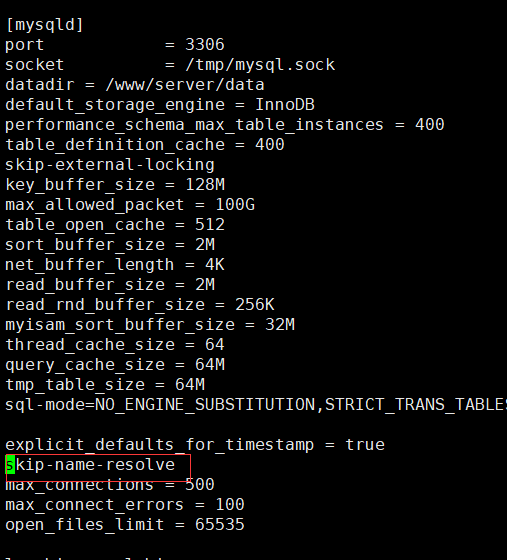
2 SqlYog 较好

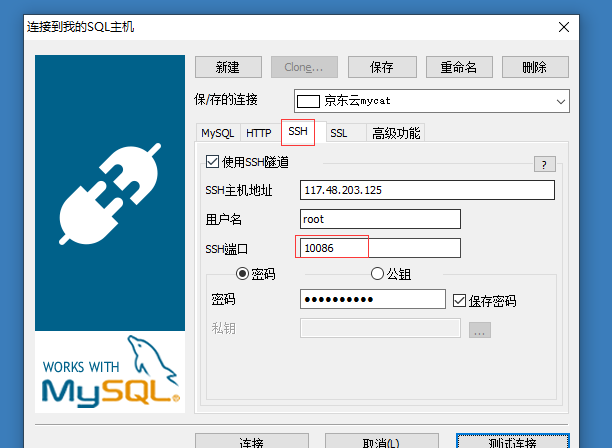


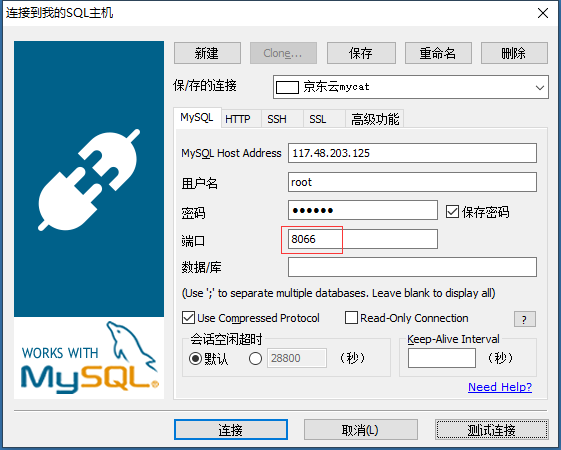
这是因为Mycat没有正常启动

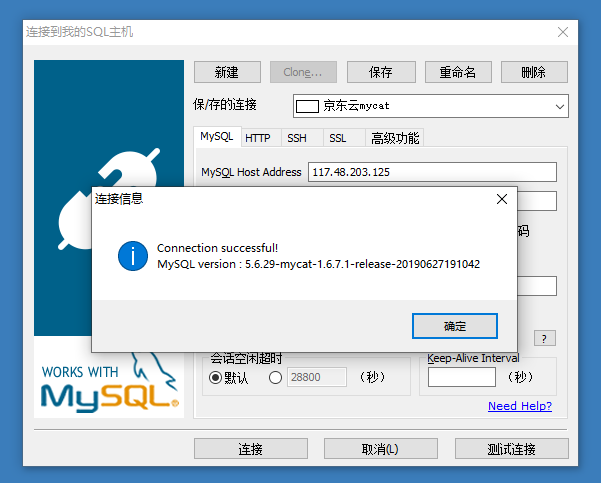
### 如果后面的error:138 是因为8066的端口没有放行

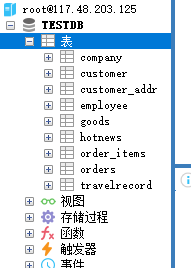
修改/etc/my.conf











用户名为root

密码为123456

可以上conf/server.xml里面设置

也可以使用 mysql链接mycat

**mysql -uroot -p123456 -P8066 -h127.0.0.1**

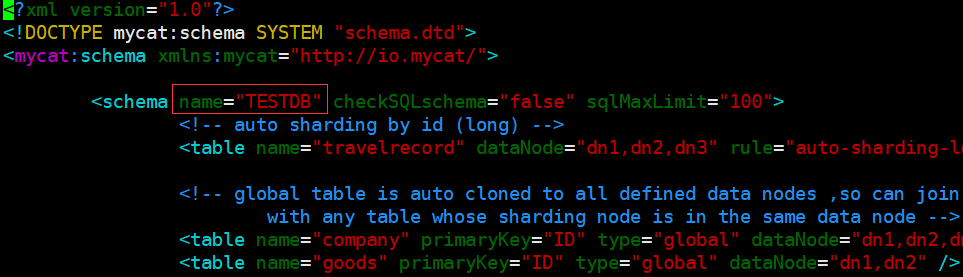
# 【掌握】mycat名词解释

## 逻辑库

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**。

如图一中，在MYCAT服务区中的TESTDB库，只是逻辑上存在的数据库，真正的数据来源还是来源MYSQL服务区中的两台实际的Mysql db实例。

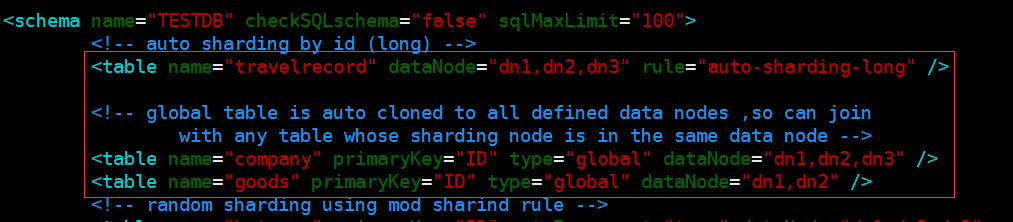
在Mycat中逻辑库在{MYCAT\_HOME}/conf/schema.xml 用<schema> 标签定义。如图：



## 逻辑表

既然有逻辑库，肯定将会存在逻辑表，**分布式数据库中，对应用来说，读写数据的表就是逻辑表**。

逻辑表的数据来源，可以是数据进行切分后，分布在一个或多个分片库中，针对不同的数据分布和管理特点，我们将逻辑表又分为**分片表、全局表、ER表、非分片表**五种逻辑表类型。在schema.xml使用<table>标签对逻辑表进行定义。如图



## 分片表

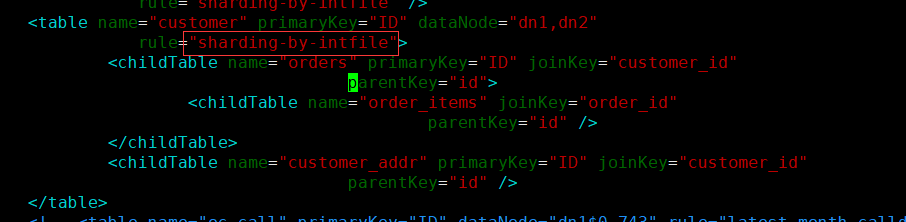
        是指那些原有的很大数据的表，需要切分到多个表，这样，每个分片都有表的一部分数据，所有分片数据的合集构成了完整的表数据，如图一种中MYCAT服务区的users表即是分片表，通过userID字段取模的方式进行数据的水平切分。

如图中用户（company）表：



## 分片规则

将大数据的表，切分到多个数据分片的策略。如图三中rule="mod-userID-long",名字为mod-userID-long引用的详细规则，将在MYCAT的rule.xml中（{MYCAT\_HOME}/conf/rule.xml）中进行定义，具体定义规则如图：



**分片规则Mycat中内置了很多种，比如按时间、按自定义数字范围、十进制取模、程序指定，字符串Hash，一致性Hash**等等，总体可将这些分片规则分为离散型和连续型两种分片规则。

离散型分片规则数据分布均衡，对数据的处理并发能力强，但是对于分片的扩缩容存在较大的挑战。连续性分片数据分布较集中，更符合业务特性，但是对数据的处理并发能力受限数据的分布，分片的扩缩容有更好的支持。

## 全局表

一个真实的业务系统中，往往存在大量的类似数据字典表的表，数据字典表具有以下几个特性：

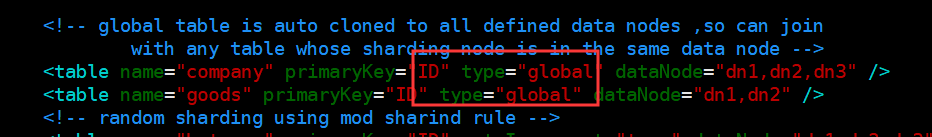
• 数据变动不频繁；

• 数据规模不大，数据量在十万以内；

• 存在跟其他表（特别是分片表）有一点的关联查询要求。

为了解决表与表的join查询，Mycat提倡大家将具有上诉特点的表通过数据冗余的方式（全局表的定义）进行解决，即所有的分片都有一份数据的拷贝。通过MYCAT对这样的表进行数据的操作时，数据的修改，新增，删除时，所有的分片数据都将受到影响。

设置方式非常简单如下图



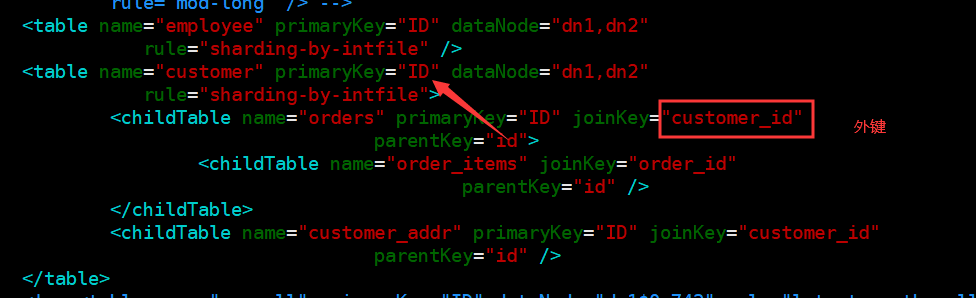
## ER表(一对多，多对一)

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。

这样一种表分组的设计方式是解决跨分片数据 join 的一种很好的思路，也是数据切分规划的重要一条规则。ER表中在schema.xml中使用<childTable>标签进行描述和定义

如图



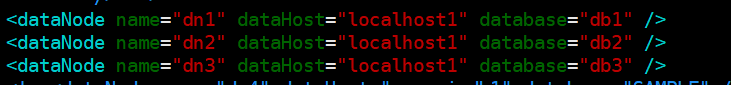
## 非分片表

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。在schema.xml中具体的定义，可参见图七：



## 分片节点

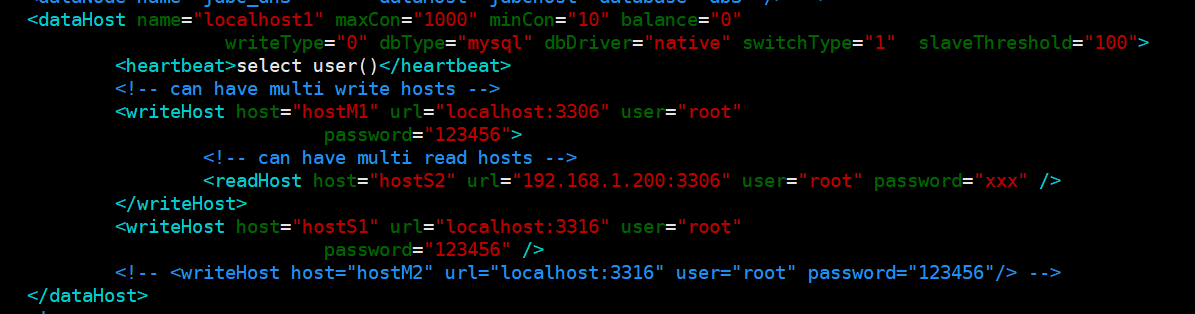
大数据表进行数据切分后，每个表分片所在的数据库就是分片节点，狭义的理解可以认为一个DB实例就是一个节点，在schema.xml中使用<dataNode>进行分片节点的定义如图八：

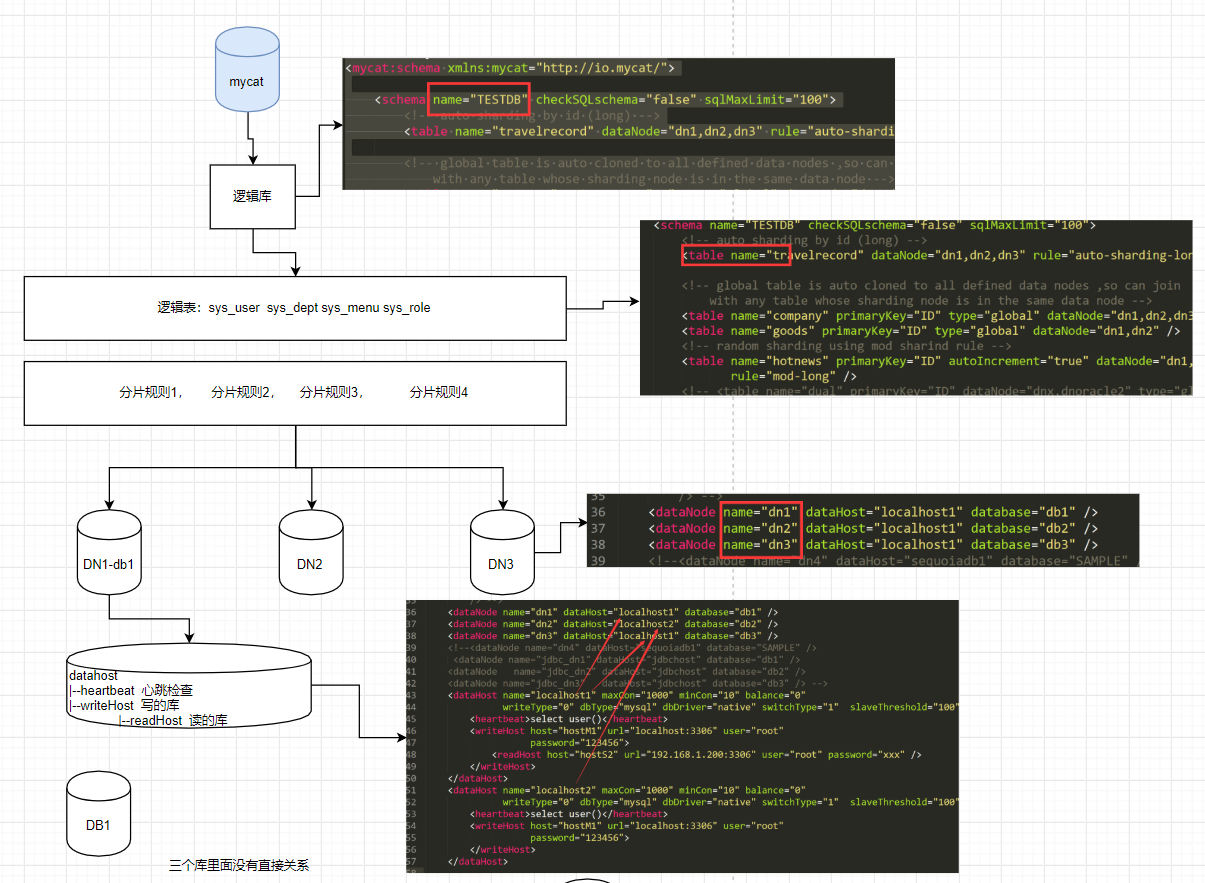


## 节点主机

数据切分后，每个分片节点（dataNode）不一定都会独占一台机器，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,为了规避单节点主机并发数限制。

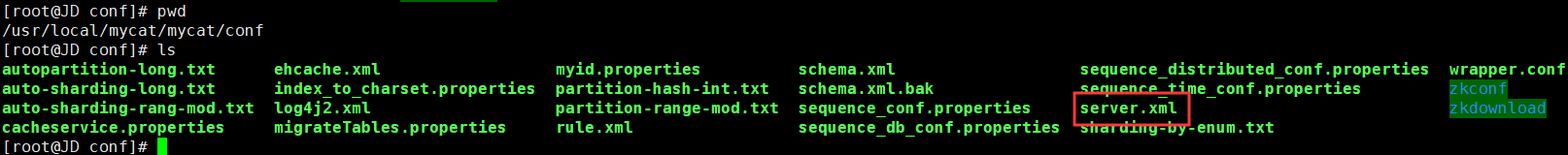
尽量将读写压力高的分片节点（dataNode）均衡的放在不同的节点主机，在schema.xml中使用<dataHost>进行分片节点的定义如图



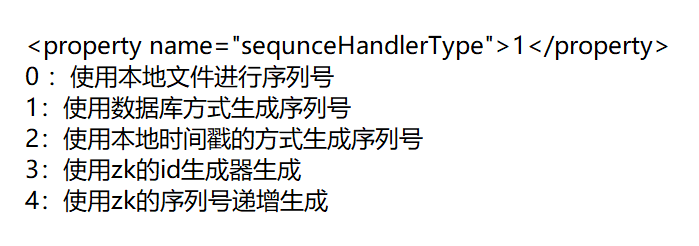


# 【掌握】mycat server.xml配置

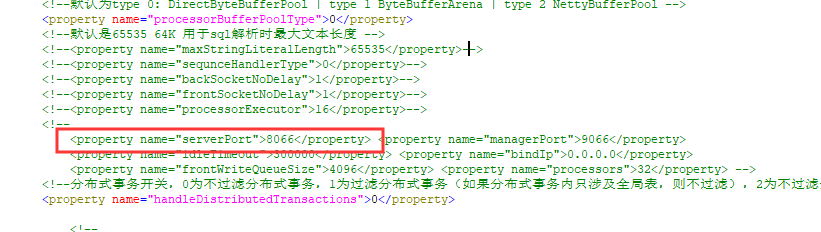
## 找到conf/server.xml



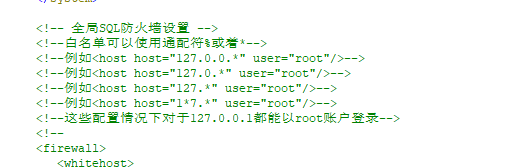
## 序列号配置



## 访问端口的配置



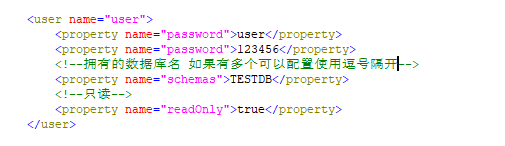
## 黑白名单的配置



## 默认账号的配置

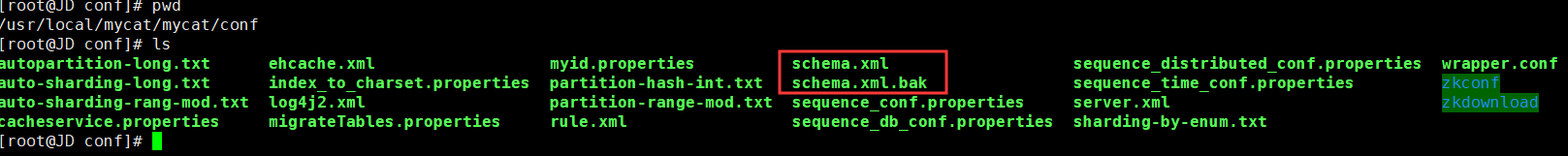


## 自定义账号的配置



# 【掌握】mycat简单配置之schema.xml详解

## 找到conf/schema.xml并备份



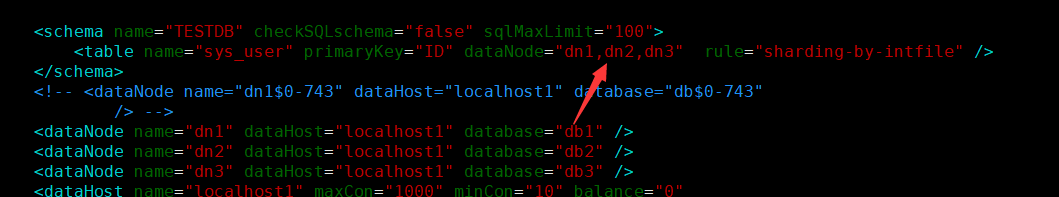
## 配置虚拟表table[在schema里面]

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

<table name="sys\_user" primaryKey="ID" dataNode="dn1,dn2,dn3" rule="sharding-by-intfile" />

</schema>

## 配置数据节点dataNode

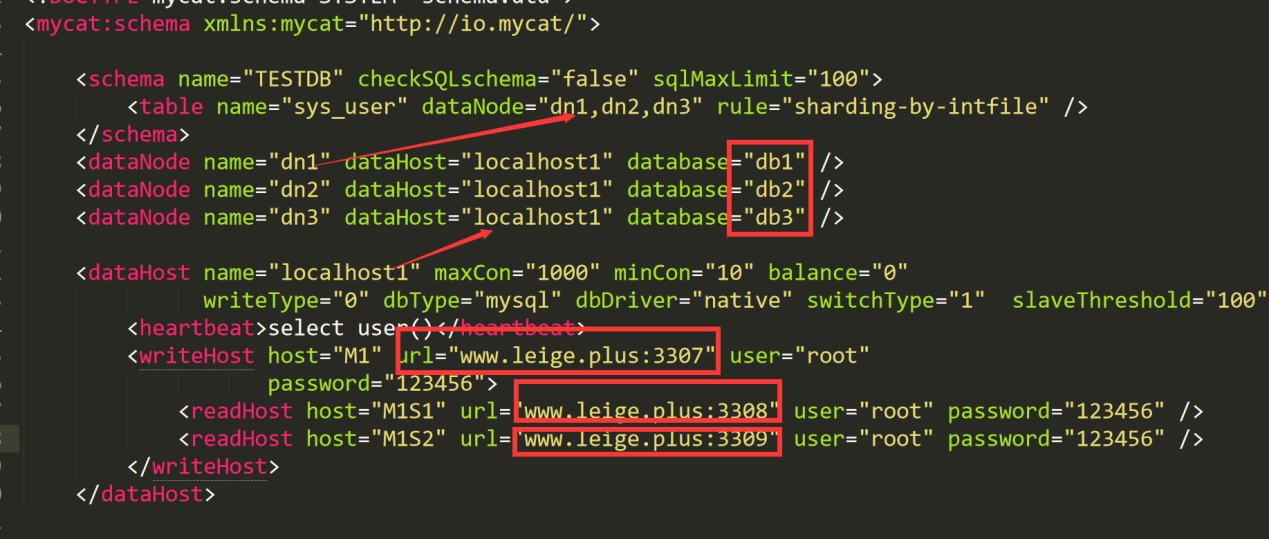


name 节点名称

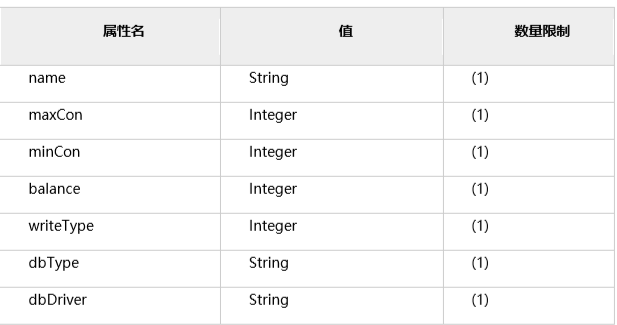
dataHost 主机名

database 数据库名

## 配置节点主机dataHost

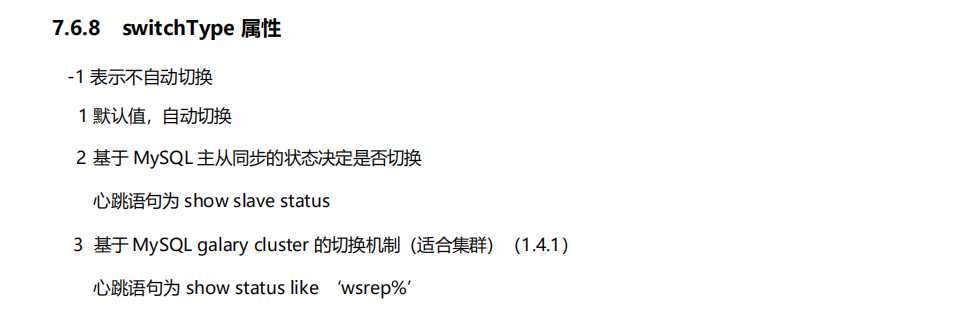


### dataHost属性说明

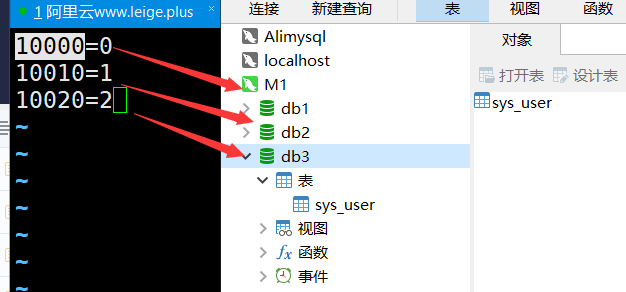


1. name   唯一标识dataHost 标签，供上层的标签使用。
2. maxCon 指定每个读写实例连接池的最大连接。也就是说，标签内嵌套的 writeHost、readHost标签都会使用这个属性的值来实例化出连接池的最大连接数。
3. minCon 指定每个读写实例连接池的最小连接，初始化连接池的大小。
4. balance  查看文档
5. writeType 查看文档
6. dbType 查看文档
7. dbDriver  查看文档

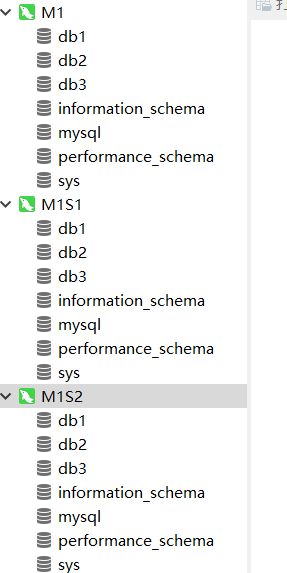




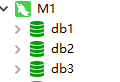
## 修改conf/partition-hash-int.txt



## 使用昨天上面配置的1主二从



## 在M1上创建三个数据库db1 db2 db3



## 启动mycat

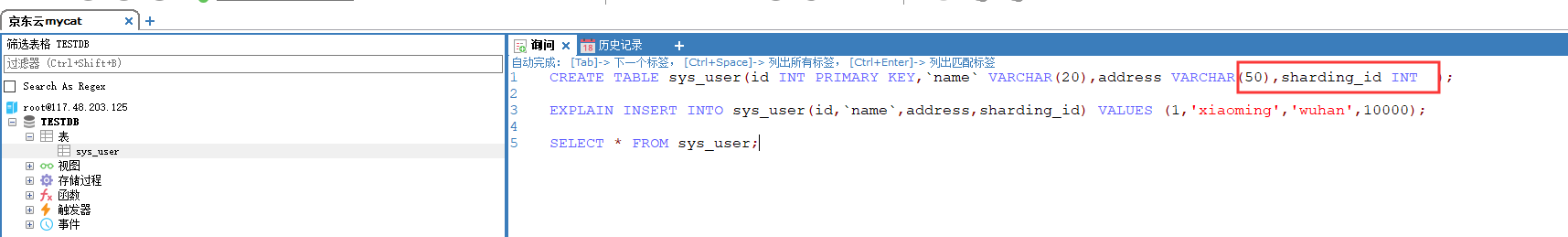


此时发现表不能查看数据

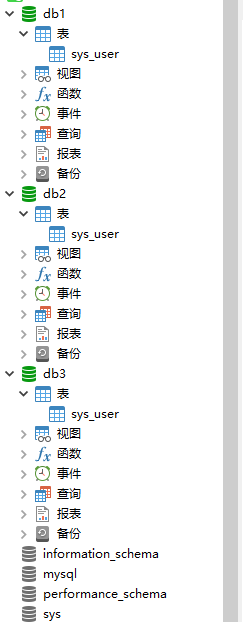
这是因为真实的mysql上还没有表

## 使用工具连接mycat创建表

create table sys\_user(id int primary key,username varchar(30),address varchar(50),sharding\_id int);



## 查看mysql里面的表已经帮我们创建好了



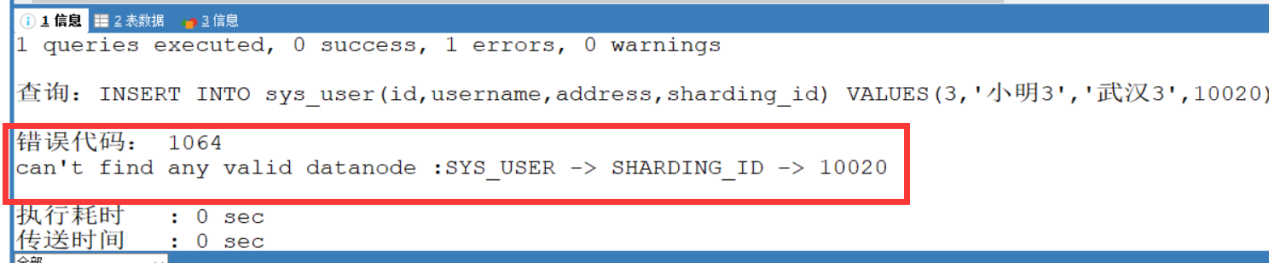
## 做添加修改删除

EXPLAIN INSERT INTO sys\_user(id,`name`,address,sharding\_id) VALUES (1,'xiaoming','wuhan',10000);

​

SELECT \* FROM sys\_user;

## 相关错误说明



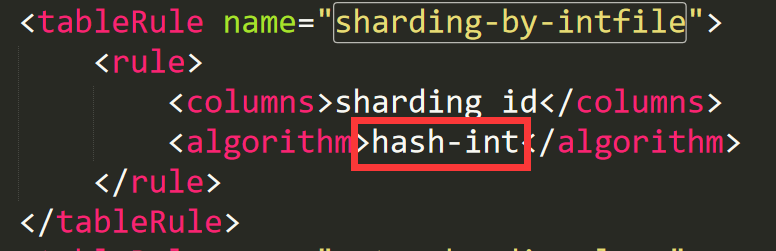
因为修改了partition-hash-int.txt文件内容之后没有重启动mycat

# 【掌握】分片详解之枚举

## 分片思路

sys\_user -🡪rule🡪sharding-by-intfile

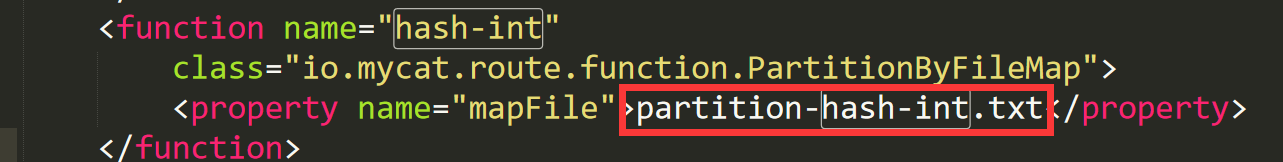
|---rule.xml

|---

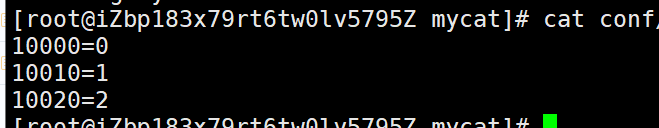
columns:代表数据库里面的字段名

Algorithm：分片算法

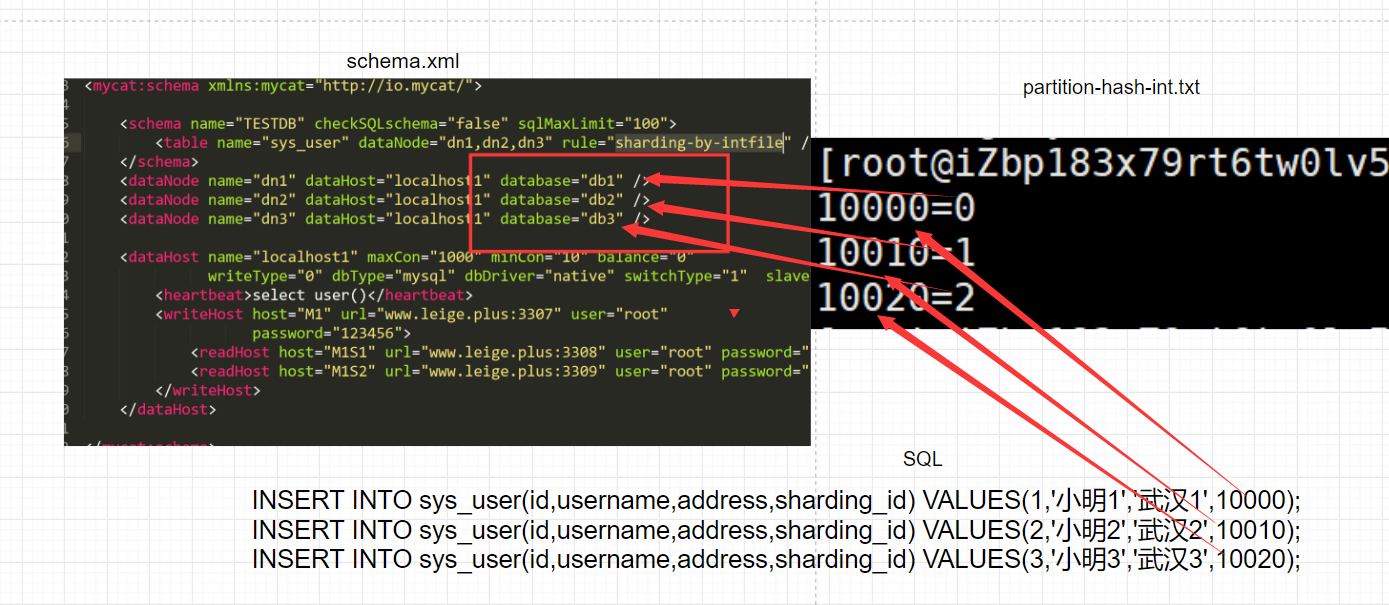
找到



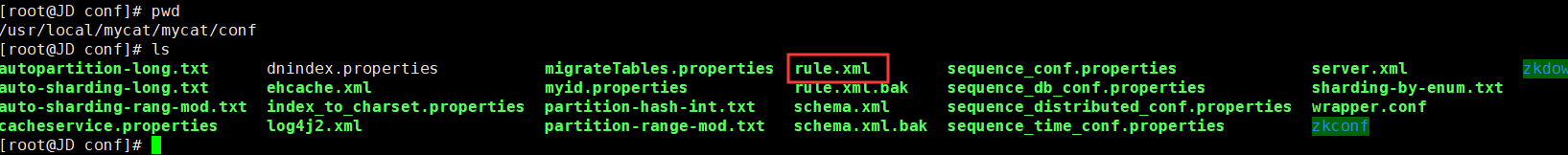
找到partition-hash-int.txt



### 整体逻辑



## 找到conf/rule.xml并备份



## 枚举路由规则

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

<tableRule name="sharding-by-intfile">

<rule>

<columns>user\_id</columns>

<algorithm>hash-int</algorithm>

</rule>

</tableRule>

<function name="hash-int" class="io.mycat.route.function.PartitionByFileMap">

<property name="mapFile">partition-hash-int.txt</property>

<property name="type">0</property>

<property name="defaultNode">1</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

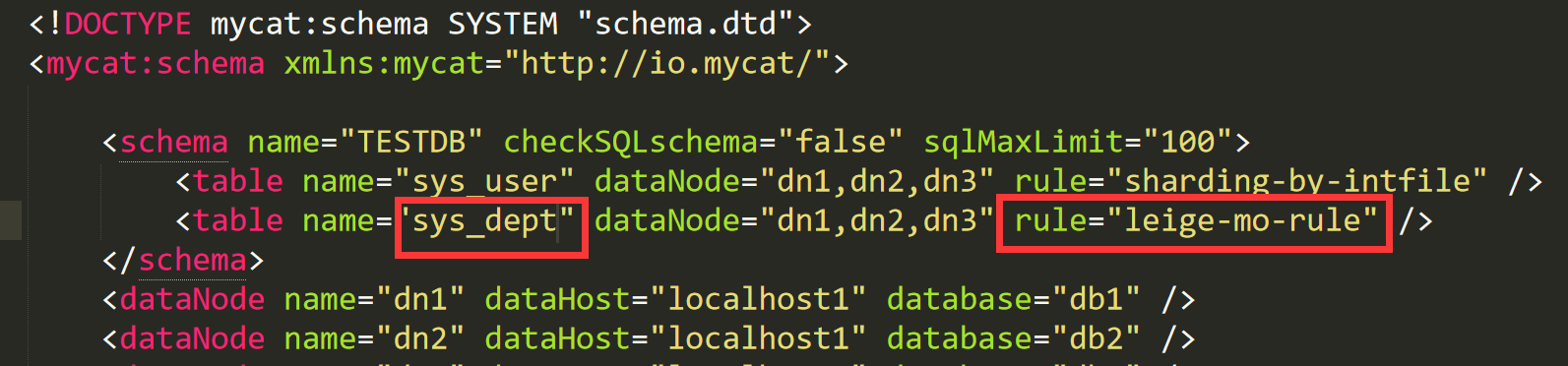
默认节点的作用：枚举分片时，如果碰到不识别的枚举值，就让它路由到默认节点

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 shardin

添加数据时给sharding\_id字 段值为10000会插入到db1数据库 10010会到db2数据库 以此类推

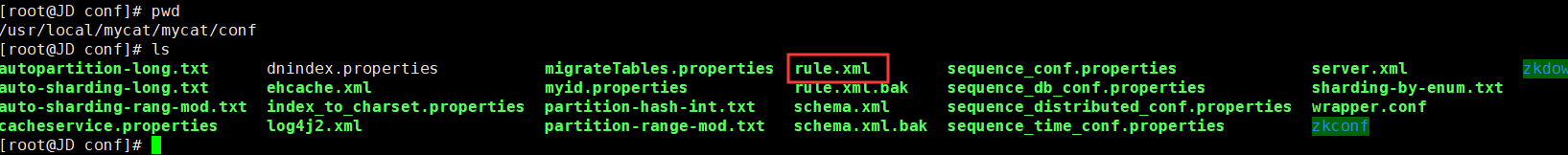
# 【掌握】分片详解之取模

## 找到conf/schema.xml修改



·

## 找到conf/rule.xml并备份



## 取模的路由规则

和轮询一样，取模有什么好处，有什么坏处？

优点：利用的写的负载均衡效果，写入速度很快

**缺点：批量写入，失败后事务的回滚有难度！代表写都成功，或都失败**

**批量写入：100**

**第99 条失败了，执行数据的回滚！**

**db1 db2 db3 (跨数据库的回滚，非常难，很耗费性能）**

此规则为对分片字段求摸运算。

|  |
| --- |
| <tableRule name="leige-mo-rule">  <rule>  <columns>id</columns>  <algorithm>leige-mo-rule-hash</algorithm>  </rule>  </tableRule>  <function name="leige-mo-rule-hash" class="io.mycat.route.function.PartitionByMod">  <!-- 有多个少datanode就配置几 -->  <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事务插入多数据分片，增大事务一致性难度

## 测试

### 创建表

CREATE TABLE sys\_dept(

id INT PRIMARY KEY ,

deptname VARCHAR(20) NOT NULL

);

EXPLAIN INSERT INTO sys\_dept(id,deptname) VALUES(2,'开发3部');

### 插入数据测试

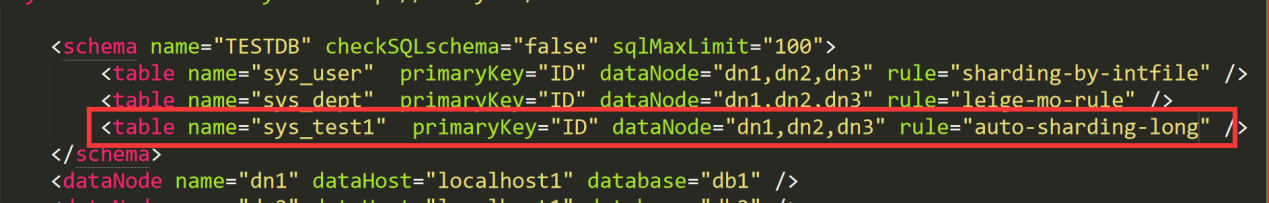
当id=1 时 1%3=1 所以数据到db2

当id=2 时 2%3=2 所以数据到db3

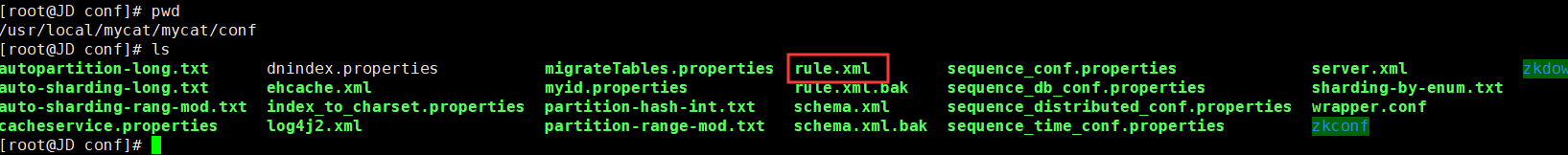
当id=3 时 3%3=0 所以数据到db1

# 【掌握】分片详解之auto-sharding-long

## 找到conf/schema.xml并备份



## 找到conf/rule.xml并备份



## auto-sharding-long的分区算法

当数据达到500w 时，才进行分库分表

db1 的表里面，写满500w，才进第二个里面写数据

缺点：没有里面写的负载均衡效果

优点：没有跨区回滚事务的风险

<tableRule name="auto-sharding-long">

<rule>

<columns>id</columns>

<algorithm>rang-long</algorithm>

</rule>

</tableRule>

<function name="rang-long"

class="io.mycat.route.function.AutoPartitionByLong">

<property name="mapFile">autopartition-long.txt</property>

</function>

​

autopartition-long.txt文件内容

# range start-end ,data node index

# K=1000,M=10000.

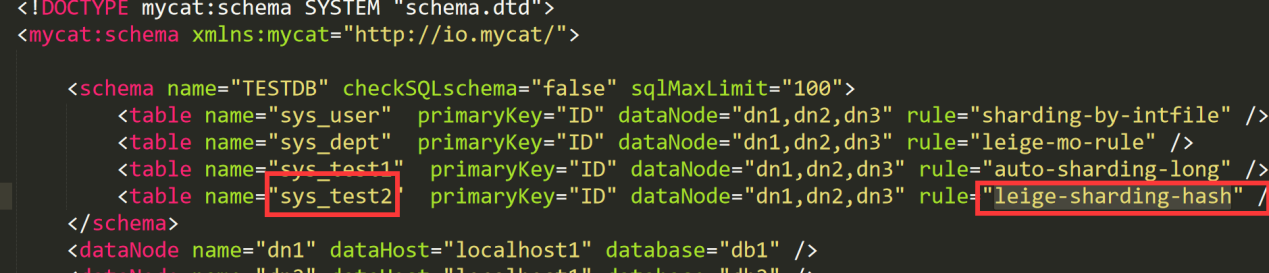
0-500M=0 #0-500W条数据在第一分区 下面依次

500M-1000M=1

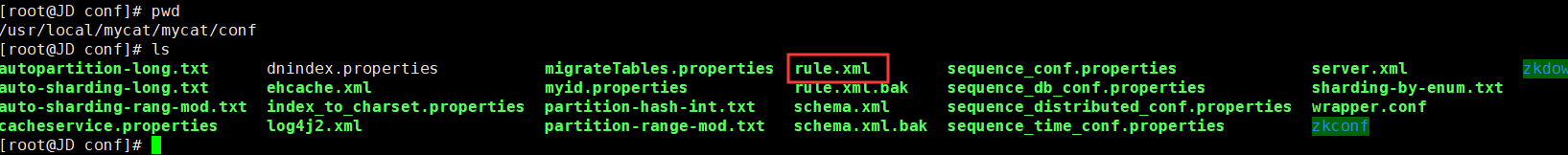
1000M-1500M=2

# 【掌握】分片详解之固定分片hash算法

## 找到conf/schema.xml并备份

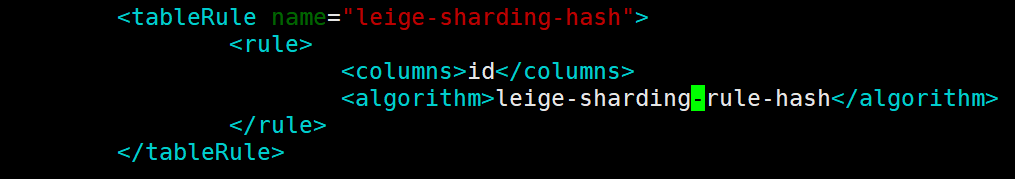


## 找到conf/rule.xml并备份



## 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 。 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。



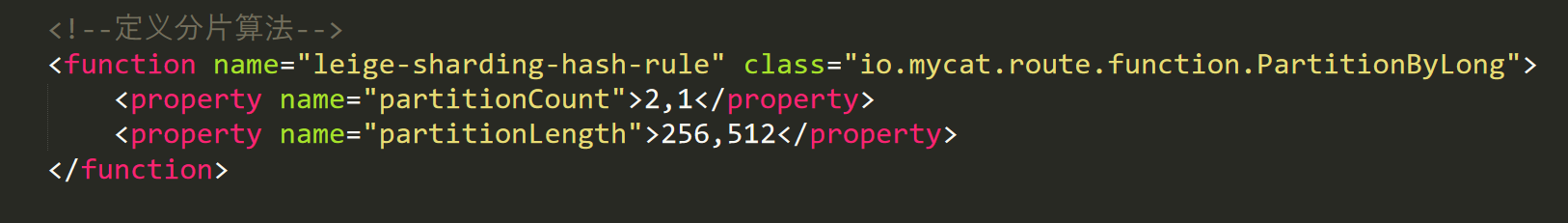
配置说明： 上面columns 标识将要分片的表字段，algorithm 分片函数， partitionCount 分片个数列表，partitionLength 分片范围列表 分区长度:默认为最大2^n=1024 ,即最大支持1024分区 约束 : count,length两个数组的长度必须是一致的。 1024 = sum((count[i]\*length[i])). count 和length两个向量的点积恒等于1024

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 };



partitionCount分片的数量级

partitionLength 分片数量级的长度

以上的配置是2\*256+1\*512=1024 那么1024就是分区的模 必须有（2+1）个datanode节点

## 测试

将sys\_customer 表设计出来，然后把它的分片规则修改为区间内轮询的概念

CREATE TABLE sys\_test2(

id INT PRIMARY KEY ,

testname VARCHAR(20) NOT NULL

);

插入测试—第一圈

1-255 dn1

256-511 dn2

512-1023 dn3

插入测试—第二圈

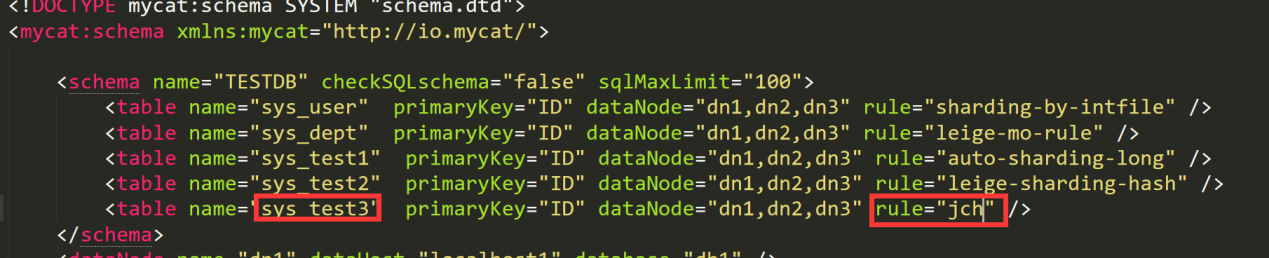
(1024)-(1024+255) dn1

(1024+256)-(1024+511) dn2

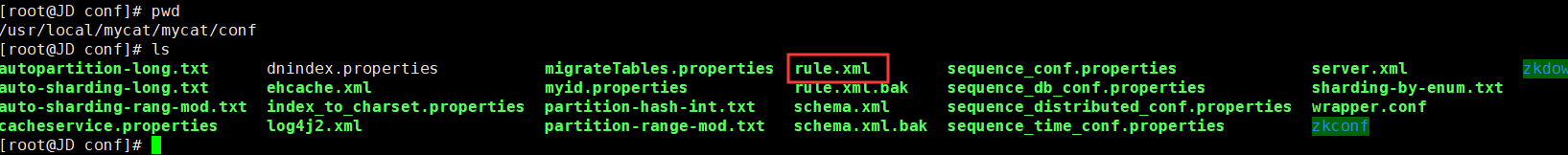
(1024+512)-(2047) dn3

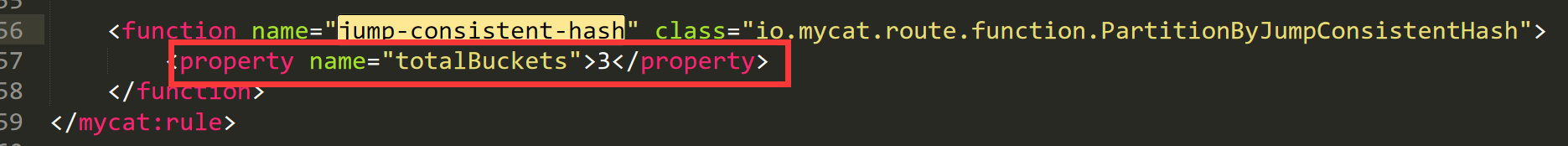
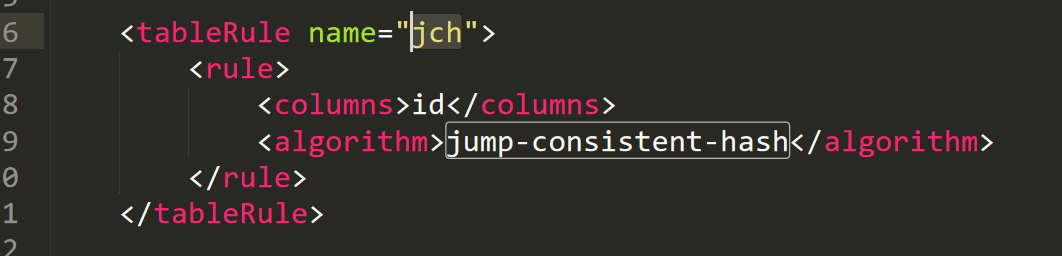
# 【熟悉】分片详解之字符串ID处理

## 找到conf/schema.xml并备份



## 找到conf/rule.xml并备份



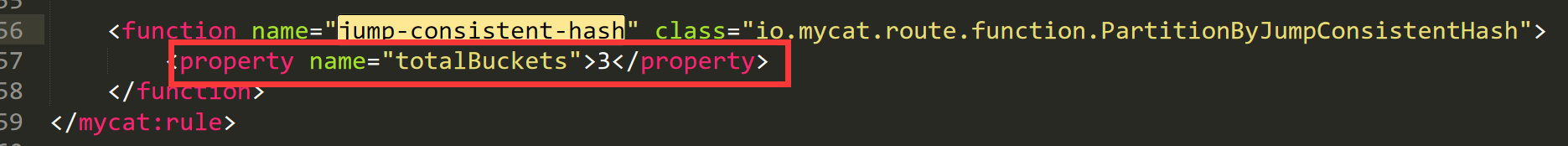


## 字符串的id怎么玩？

只有是字符串，就使用JCH

将给key 分配给n 个buckets。

jump Consistent hash:零内存消耗，均匀，快速，简洁，来自Google的一致性哈希算法



Hash 算法

将一个值映射一个区间的某个值上面！

将很多的字符串我可以平均的分配到 4 个buckets 里面就ok

Rehash 的过程，所有在选择容器存储时，我们需要预估该容器里面最大的容量值，并且使用负载因子来确定 容器的size。这样的话，我们可以减少rehash的过程

总结：

 以后只要使用到字符串的id 类型，就使用jch的算法就ok 了

## 测试

注意表的ID必须为字符串

#sys\_test3

CREATE TABLE sys\_test3(

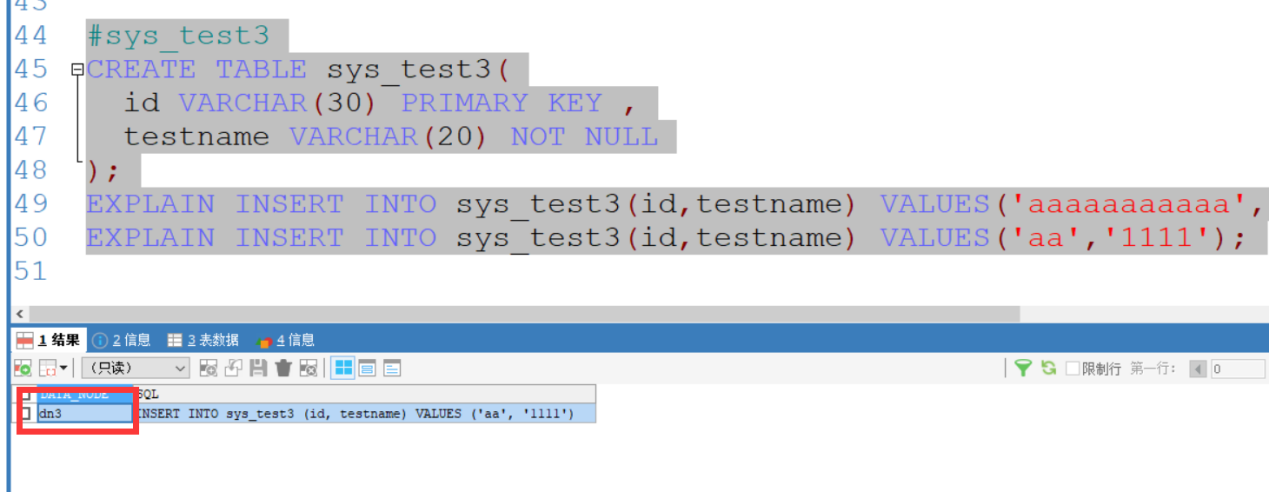
id VARCHAR(30) PRIMARY KEY ,

testname VARCHAR(20) NOT NULL

);

EXPLAIN INSERT INTO sys\_test3(id,testname) VALUES('aaaaaaaaaaa','1111');

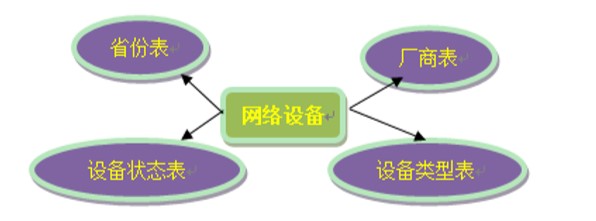
EXPLAIN INSERT INTO sys\_test3(id,testname) VALUES('aa','1111');



# 【掌握】全局表和普通表的配置和测试

## 全局表概述

一个真实的业务系统中，往往存在大量的类似字典表的表格，它们与业务表之间可能有关系，这种关系，可以理解为“标签”，而不应理解为通常的“主从关系”，这些表基本上很少变动，可以根据主键 ID进行缓存，下面这张图说明了一个典型的“标签关系”图：



在分片的情况下，当业务表因为规模而进行分片以后，业务表与这些附属的字典表之间的关联，就成了比较棘手的问题，考虑到字典表具有以下几个特性：

• 变动不频繁

• 数据量总体变化不大

• 数据规模不大，很少有超过数十万条记录。

鉴于此，MyCAT定义了一种特殊的表，称之为“全局表”，全局表具有以下特性：

• 全局表的插入、更新操作会实时在所有节点上执行，保持各个分片的数据一致性

• 全局表的查询操作，只从一个节点获取

• 全局表可以跟任何一个表进行 JOIN操作

将字典表或者符合字典表特性的一些表定义为全局表，则从另外一个方面，很好的解决了数据 JOIN的难题。

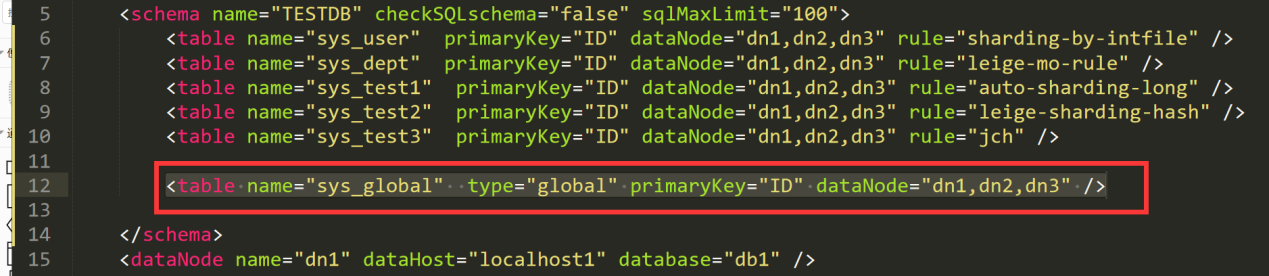
通过全局表+基于E-R关系的分片策略，MyCAT可以满足80%以上的企业应用开发。

## 配置方式

全局表配置比较简单，不用写Rule规则，如下配置即可：

### 修改schema.xml

<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。



## 测试

#全局表

CREATE TABLE sys\_global(

id INT PRIMARY KEY ,

`code` VARCHAR(30) ,

codename VARCHAR(20) NOT NULL

);

EXPLAIN INSERT INTO sys\_global(id,`code`,codename) VALUES(1,'sex:1','男');

EXPLAIN INSERT INTO sys\_global(id,`code`,codename) VALUES(2,'sex:0','女');

EXPLAIN UPDATE sys\_global SET `codename`='汉子' WHERE `code`='sex:1';

UPDATE sys\_global SET `codename`='妹子' WHERE `code`='sex:0';

EXPLAIN SELECT \* FROM sys\_global; #全局表的查询是轮询

## 说明

如查一个逻辑表不设置全局，也不设置路由规则，那么默认所有节点都会存数据

查询时会所以的节点数据全查询出，再汇总，会出现重复数据

#<table name="sys\_global2" primaryKey="ID" dataNode="dn1,dn2,dn3" />

EXPLAIN CREATE TABLE sys\_global2(

id INT PRIMARY KEY ,

`code` VARCHAR(30) ,

codename VARCHAR(20) NOT NULL

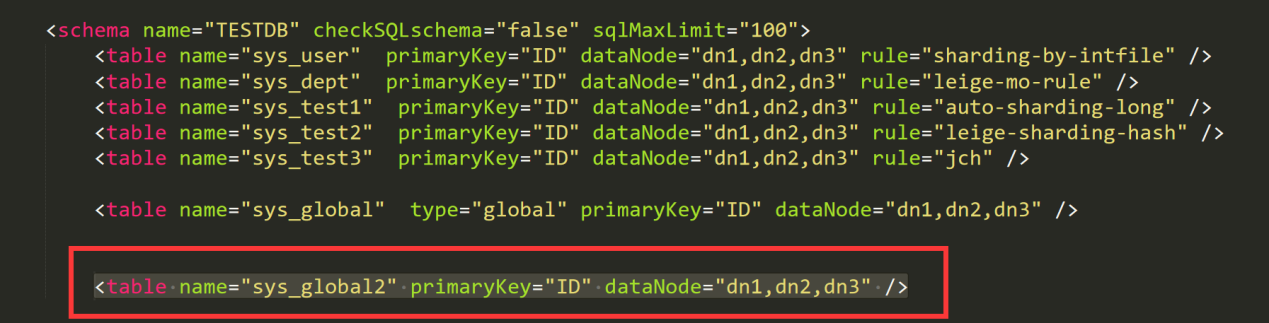
);

EXPLAIN INSERT INTO sys\_global2(id,`code`,codename) VALUES(1,'sex:1','男');

EXPLAIN INSERT INTO sys\_global2(id,`code`,codename) VALUES(2,'sex:0','女');

#好像不加全局表，不写路由，好像和全局表是样？

SELECT \* FROM sys\_global2;



# 【了解】ER表的配置和测试

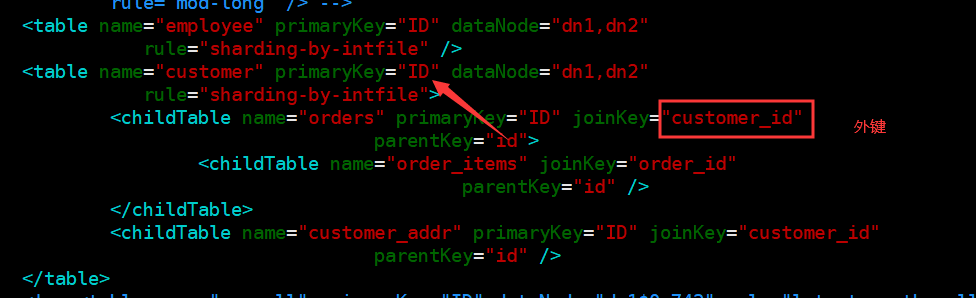
## ER表概述(一对多，多对一)

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。

这样一种表分组的设计方式是解决跨分片数据 join 的一种很好的思路，也是数据切分规划的重要一条规则。ER表中在schema.xml中使用<childTable>标签进行描述和定义

如图



## 以上说明

Customer 主键id

|--orders 主键id customer\_id映射customer表里面id

|--order\_items 无主键 order\_id映射orders 里面的主键id

# 【掌握】全局ID的概述

## 面试题引入

在集群高并发情况下如何保证分布式唯一全局ID生成？

## 为什么要全局分布式ID

在复杂的分布式系统中，需要对大量的数据和消息进行唯一标识

如在阿里，淘宝，支付，等系统中，数据日渐增长，对数据分库分表后需要有一个唯一ID来标识一条数据或消息;

还有如美团和饿了吗的骑手ID 商家ID  优惠券ID等等

从以上可以从出，一个能够生成全局唯一ID的系统是非常必要的

## ID生成规则部分硬性要求

### 全局唯一：

不能出现重复的ID号，这个是最基础的要求

### 趋势递增：

在Mysql的InnoDb引擎中使用的是聚集索引，由于多数据的RDBMS使用的是Btree的数据结构来存储索引数据，在主键的选择上面我们应该尽量敷衍有序的主键保存写入性能。

### 单调递增：

保证下一个ID一定大于上一个ID。

### 信息安全：

如果ID是连续的，恶意用户的扒取工作就很容易了，直接按顺序下载指定的URL就行，如果订单号就更危险了，竞争对手可以直接知道我们一天的单量。所以在一些应用场景下，需要ID无规则不规则，让竞争对手不好猜。

### 含时间戳：

就可以知道这个分布式 ID在什么时候生成的。

## ID生成系统的可用性要求

1，高可用：发一个获取分布式ID的请求同，服务器要保证100%的情况下给我们创建一个唯一的分布式ID

2，低延时：毫秒级的生成，如，一毫秒生成4096个

3，高QPS（Queries-per-second每秒查询率）：

## 通用的解决方案

### 自增ID-auto\_increment

使用数据库的自动增长（auto\_increment），是比较简单和常见的ID生成方案，数据库内部可以确保生成id的唯一性。

**优点：**

1、数据库自动编号，速度快，而且是增量增长，聚集型主键按顺序存放，对于检索非常有利。

2、 数字型，占用空间小，易排序，在程序中传递方便。

**缺点：**

1、不支持水平分片架构，水平分片的设计当中，这种方法显然不能保证全局唯一。

2、对数据库有依赖，每种数据库可能实现不一样，数据库切换时候，涉及到代码的修改，不利于扩展

**结论：**

**自增id做主键适用于非分布式架构。**

### ~~数据库自增主键-replace into~~

的分布式里面，数据库的自增ID机制的主要原理是：数据库自增ID和mysql数据库的replace into实现。

replace into跟insert功能类似，不同点在于，replace into首先尝试插入数据列表中，如果发现表中已级的此行数据(根据主键或唯一索引判断)则先删除，再插入。否则直接插入新数据。

REPLACE INTO 的含义是插入一行记录，如查表中唯一索引的值遇到冲突，则替换老数据。

CREATE TABLE t\_test(

id BIGINT(20) UNSIGNED NOT NULL AUTO\_INCREMENT PRIMARY KEY,

stub CHAR(1) NOT NULL DEFAULT '',

UNIQUE KEY stub (stub)

);

REPLACE INTO t\_test(stub) values(‘x’);

SELECT LAST\_INSERT\_ID();

缺点：单机性能不够，集群添加机器扩容非常麻烦【一句化，并发量大了mysql扛不住】

### UUID

UUID（Universally Unique Identifier）的标准形式包含32个16进制的数字，以连字号分为5段，形式为8-4-4-4-12个字符，

如：941e3b9a-a6a8-43fd-8b74-d0627f12323f

**优点**

性能非常高：本地生成，没有网络消耗

**缺点**

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

2、信息不安全：基于MAC地址生成UUID的算法可能会造成MAC地址泄露，这个漏洞曾被用于寻找梅丽莎病毒的制作者位置。

3、ID作为主键时在特定的环境会存在一些问题，比如需要排序的时候——UUID是无序的。

4、MySQL官方有明确的建议主键要尽量越短越好，36个字符长度的UUID不符合要求。

5、对MySQL索引不利：作为数据库主键，在InnoDB引擎下，UUID的无序性可能会引起数据位置频繁变动，严重影响性能。

### 基于Redis生成全局主键

因为Redis是单线程原子操作可以使用incr和incrby来实现

**优点**：并发能力起来了，有点NB了

**缺点**：

1，代价太大

2，redis集群扩容会出现ID不连续的问题(步长不好设置)

### 基于zookeeper生成全局主键

原理，使用数据的版本号

缺点：代价太大

# 【掌握】mycat全局主键的生成方式

## Mycat全局主键方式

Mycat提供的全局主键方式如下：

1. 本地文件方式：使用服务器本地磁盘文件的方式

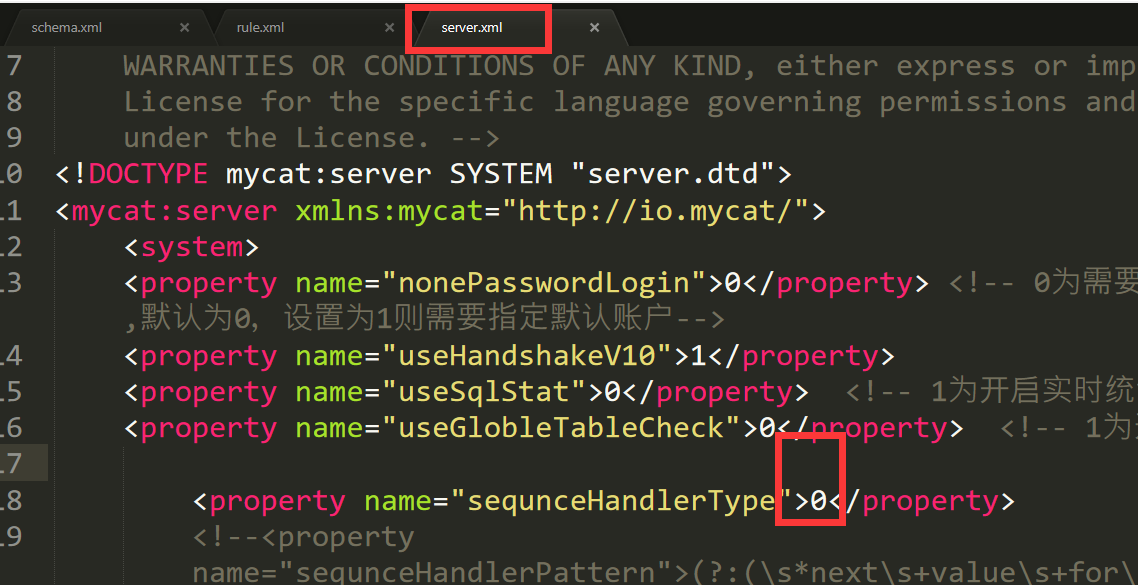
2. 数据库方式：使用数据库的方式【不推荐。要写数据库函数】

3. 本地时间戳方式：使用时间戳方式[还行]

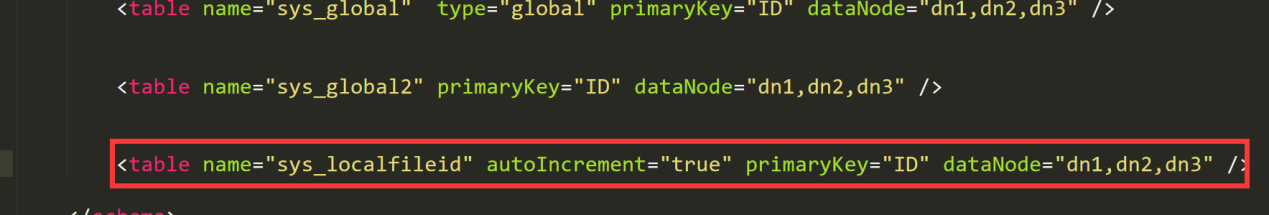
4. 分布式zookeeper生成ID【了解】

## 本地文件方式

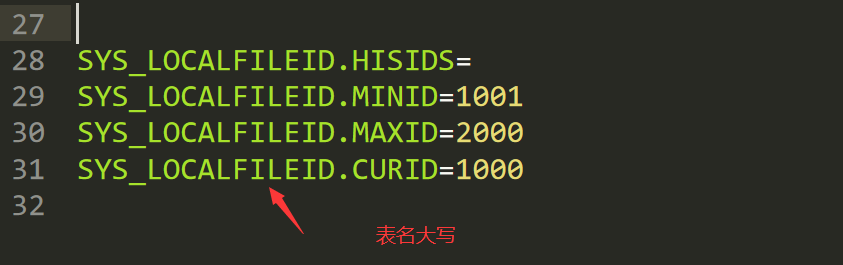
### 修改conf/server.xml



### 修改conf/schema.xml

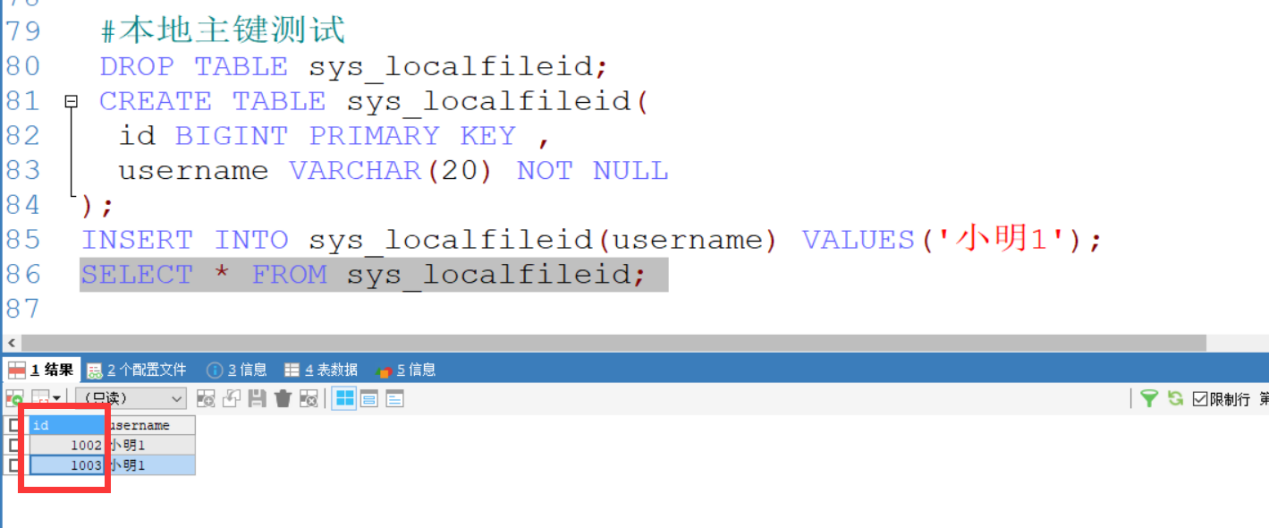


### 修改 sequence\_conf.properties注意名字



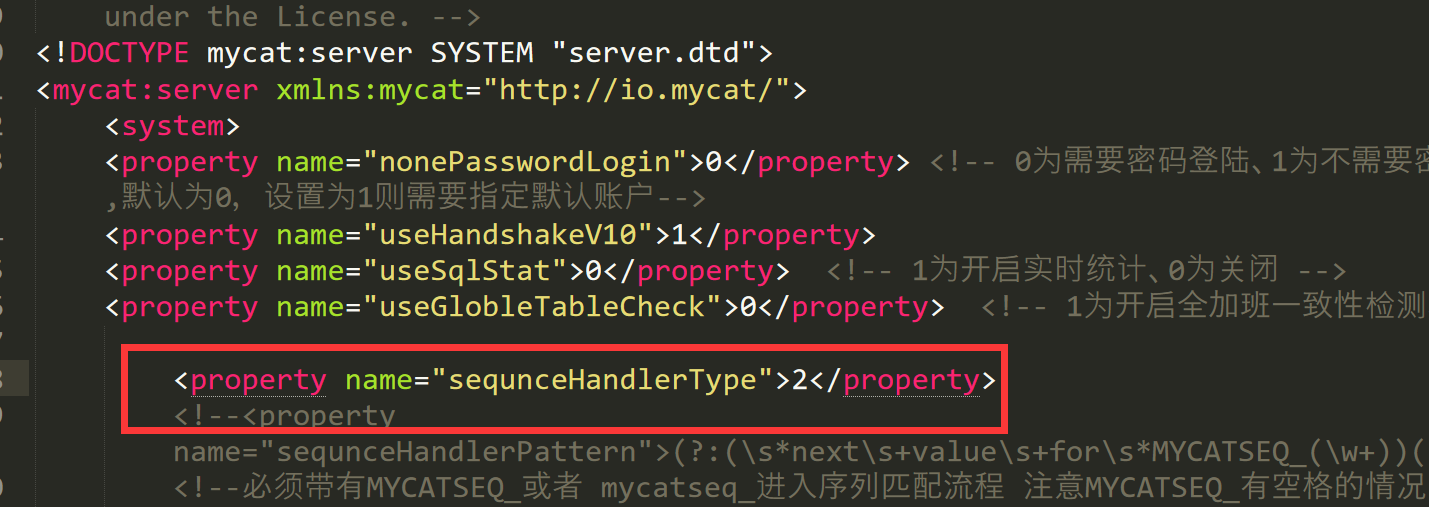
### 配置覆盖重启

### 创建表插入测试

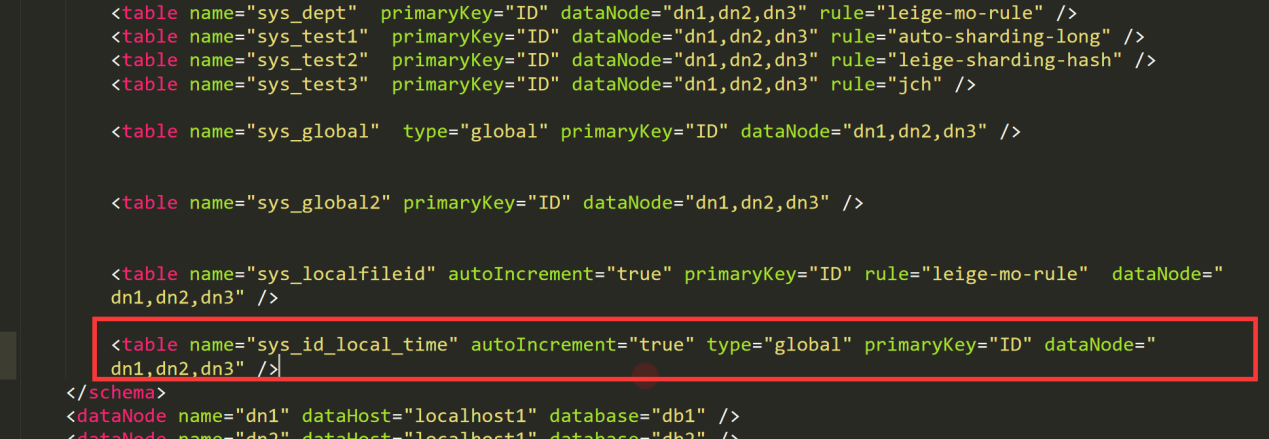


## 本地时间戳方式[还行]

### 修改conf/server.xml



### 修改conf/schema.xml



### 修改conf/sequence\_time\_conf.properties

不改

### 配置覆盖重启

### 创建表插入测试

CREATE TABLE sys\_id\_local\_time(

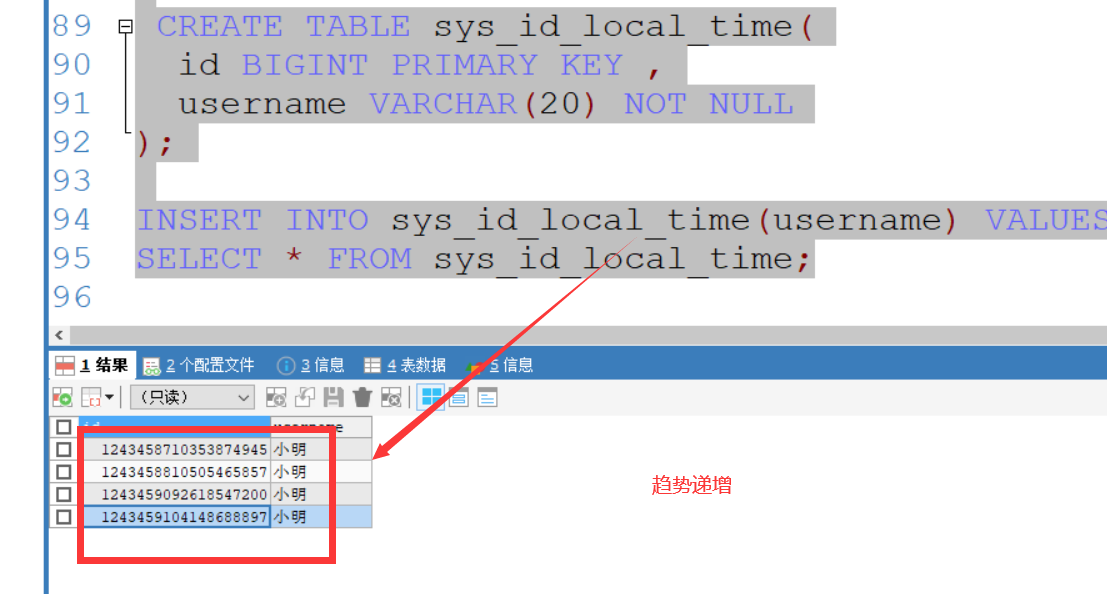
id BIGINT PRIMARY KEY ,

username VARCHAR(20) NOT NULL

);

INSERT INTO sys\_id\_local\_time(username) VALUES('小明');

SELECT \* FROM sys\_id\_local\_time;



### 优缺点

优点：不存在mycat重新发布影响seq的问题，

缺点：如果有时间有波动有问题。

# 【面试】全局主键生成方式-雪花算法

<https://github.com/twitter-archive/snowflake>

## snowflake算法【雪花算法】概述

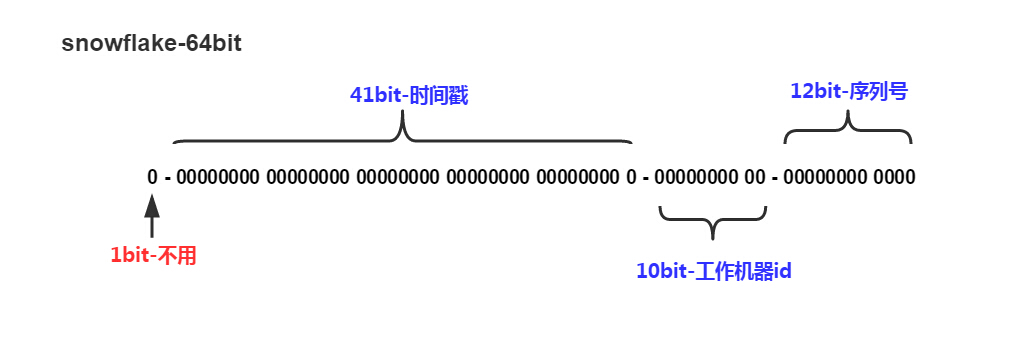
twitter开源分布式生成id算法。

优点：基本解决了所有问题

缺点：每个节点时间可能不同，生成id是整体趋势递增的

## 算法原理

snowflake的结构如下(每部分用-分开):



第一位未使用，因为二进制中最高位是符号位同，1表示负数，0表示正数。ID不可能为负数。

时间戳：41位，最后为2的41次方  大概69.73年 10==2+8

然后是5位datacenterId和5位workerId(10位的长度最多支持部署1024个节点） [**也可以是3位datacenterid和7位workerid**]

最后12位是毫秒内的计数（12位的计数顺序号支持每个节点每毫秒产生4096个ID序号）每秒生成409.6万个【可怕】

一共加起来刚好64位，为一个Long型。(转换成字符串长度为18)

<https://tool.oschina.net/hexconvert/>  可以使用这个进制转化工具测试

**优点**：

1、毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。

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

3、可以根据自身业务特性分配bit位，非常灵活。

**缺点**：

强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。

**结论**：

适用于大规模分布式架构

## 源码

|  |
| --- |
| **package** com.sxt.commons;  */\*\*  \** ***@Author:*** *尚学堂 雷哥  \*/* **public class** SnowflakeIdWorker {  *// ==============================Fields===========================================  /\*\* 开始时间截 (2015-01-01) \*/* **private final long twepoch** = 1420041600000L;   */\*\* 机器id所占的位数 \*/* **private final long workerIdBits** = 5L;   */\*\* 数据标识id所占的位数 \*/* **private final long datacenterIdBits** = 5L;   */\*\* 支持的最大机器id，结果是31 (这个移位算法可以很快的计算出几位二进制数所能表示的最大十进制数) \*/* **private final long maxWorkerId** = -1L ^ (-1L << **workerIdBits**);   */\*\* 支持的最大数据标识id，结果是31 \*/* **private final long maxDatacenterId** = -1L ^ (-1L << **datacenterIdBits**);   */\*\* 序列在id中占的位数 \*/* **private final long sequenceBits** = 12L;   */\*\* 机器ID向左移12位 \*/* **private final long workerIdShift** = **sequenceBits**;   */\*\* 数据标识id向左移17位(12+5) \*/* **private final long datacenterIdShift** = **sequenceBits** + **workerIdBits**;   */\*\* 时间截向左移22位(5+5+12) \*/* **private final long timestampLeftShift** = **sequenceBits** + **workerIdBits** + **datacenterIdBits**;   */\*\* 生成序列的掩码，这里为4095 (0b111111111111=0xfff=4095) \*/* **private final long sequenceMask** = -1L ^ (-1L << **sequenceBits**);   */\*\* 工作机器ID(0~31) \*/* **private long workerId**;   */\*\* 数据中心ID(0~31) \*/* **private long datacenterId**;   */\*\* 毫秒内序列(0~4095) \*/* **private long sequence** = 0L;   */\*\* 上次生成ID的时间截 \*/* **private long lastTimestamp** = -1L;   *//==============================Constructors=====================================  /\*\*  \* 构造函数  \** ***@param workerId*** *工作ID (0~31)  \** ***@param datacenterId*** *数据中心ID (0~31)  \*/* **public** SnowflakeIdWorker(**long** workerId, **long** datacenterId) {  **if** (workerId > **maxWorkerId** || workerId < 0) {  **throw new** IllegalArgumentException(String.*format*(**"worker Id can't be greater than %d or less than 0"**, **maxWorkerId**));  }  **if** (datacenterId > **maxDatacenterId** || datacenterId < 0) {  **throw new** IllegalArgumentException(String.*format*(**"datacenter Id can't be greater than %d or less than 0"**, **maxDatacenterId**));  }  **this**.**workerId** = workerId;  **this**.**datacenterId** = datacenterId;  }   *// ==============================Methods==========================================  /\*\*  \* 获得下一个ID (该方法是线程安全的)  \** ***@return*** *SnowflakeId  \*/* **public synchronized long** nextId() {  **long** timestamp = timeGen();   *//如果当前时间小于上一次ID生成的时间戳，说明系统时钟回退过这个时候应当抛出异常* **if** (timestamp < **lastTimestamp**) {  **throw new** RuntimeException(  String.*format*(**"Clock moved backwards. Refusing to generate id for %d milliseconds"**, **lastTimestamp** - timestamp));  }   *//如果是同一时间生成的，则进行毫秒内序列* **if** (**lastTimestamp** == timestamp) {  **sequence** = (**sequence** + 1) & **sequenceMask**;  *//毫秒内序列溢出* **if** (**sequence** == 0) {  *//阻塞到下一个毫秒,获得新的时间戳* timestamp = tilNextMillis(**lastTimestamp**);  }  }  *//时间戳改变，毫秒内序列重置* **else** {  **sequence** = 0L;  }   *//上次生成ID的时间截* **lastTimestamp** = timestamp;   *//移位并通过或运算拼到一起组成64位的ID* **return** ((timestamp - **twepoch**) << **timestampLeftShift**) *//* | (**datacenterId** << **datacenterIdShift**) *//* | (**workerId** << **workerIdShift**) *//* | **sequence**;  }   */\*\*  \* 阻塞到下一个毫秒，直到获得新的时间戳  \** ***@param lastTimestamp*** *上次生成ID的时间截  \** ***@return*** *当前时间戳  \*/* **protected long** tilNextMillis(**long** lastTimestamp) {  **long** timestamp = timeGen();  **while** (timestamp <= lastTimestamp) {  timestamp = timeGen();  }  **return** timestamp;  }   */\*\*  \* 返回以毫秒为单位的当前时间  \** ***@return*** *当前时间(毫秒)  \*/* **protected long** timeGen() {  **return** System.*currentTimeMillis*();  }   *//==============================Test=============================================  /\*\* 测试 \*/* **public static void** main(String[] args) {  SnowflakeIdWorker idWorker = **new** SnowflakeIdWorker(0, 0);  **for** (**int** i = 0; i < 1000; i++) {  **long** id = idWorker.nextId();  System.***out***.println(Long.*toBinaryString*(id));  System.***out***.println(id);  }  } } |

## 使用Hutool封装

<https://github.com/looly/hutool>

### 引入pom.xml

|  |
| --- |
| <dependency>  <groupId>cn.hutool</groupId>  <artifactId>hutool-all</artifactId>  <version>5.2.4</version>  </dependency> |

### 创建封装类

|  |
| --- |
| **package** com.bjsxt.common;  **import** cn.hutool.core.lang.Snowflake; **import** cn.hutool.core.net.NetUtil; **import** cn.hutool.core.util.IdUtil;  */\*\*  \* <功能简要> <br>  \* <>  \*  \** ***@Author*** *武汉尚学堂【雷哥】   \** ***@createTime*** *2020/11/12 11:08  \** ***@since*** *1.0.0  \*/* **public class** IdGeneratorSnowflake {   **private static long** *workId*=0; *//0-31* **private static long** *datacenterId*=1; *//0-31* **private static** Snowflake *snowflake*=**null**;   **static**{  **try**{  String localhostStr = NetUtil.*getLocalhostStr*();  **if**(localhostStr!=**null**&&!localhostStr.equals(**""**)){  *workId*=NetUtil.*ipv4ToLong*(localhostStr)%32;  }**else**{  *workId*=0;  }  }**catch** (Exception e){  System.***out***.println(**"当前程序无法得到网络信息"**);  *workId*=0;  }  *snowflake*= IdUtil.*createSnowflake*(*workId*,*datacenterId*);  }   */\*\*  \* 得到一个ID  \*/* **public static** String snowflakeId(){  **return** *snowflakeId*(**""**);  }  */\*\*  \* 得到一个带前缀的ID  \** ***@param prefix*** *前缀  \*/* **public static** String snowflakeId(String prefix){  **long** nextId = *snowflake*.nextId();  **return** prefix+nextId;  } } |

# 【总结】Mycat使用总结

## 如何使用Java连接

配置好mycat这后用法和以前java连接mysql是一样的

|  |
| --- |
| server:  port: 8088 spring:  application:  name: mycat-demo  datasource:  username: root  password: 123456  url: jdbc:mysql://**192.168.127.128:8066**/**TESTDB**?serverTimezone=GMT%2B8  driver-class-name: com.mysql.cj.jdbc.Driver mybatis:  configuration:  log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  mapper-locations: classpath:mapper/\*.xml |

## 为什么还要使用mysql集群和主从

