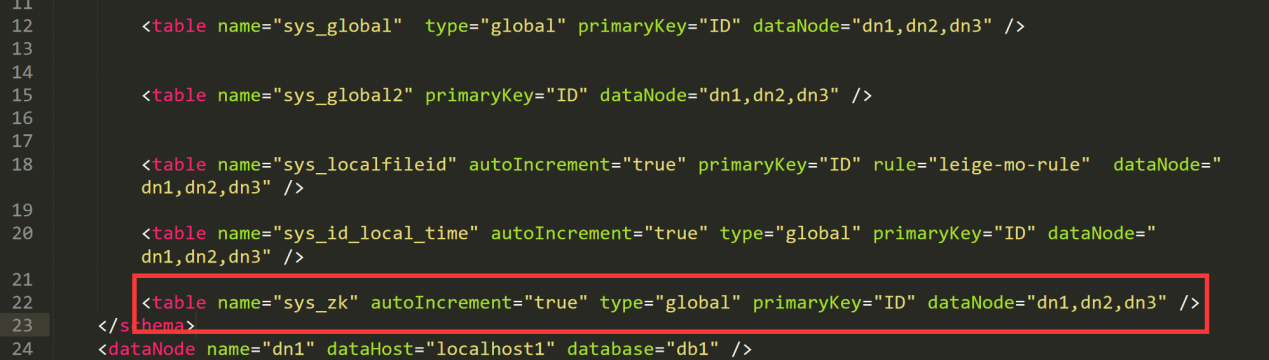
## 4，zookeeper方式

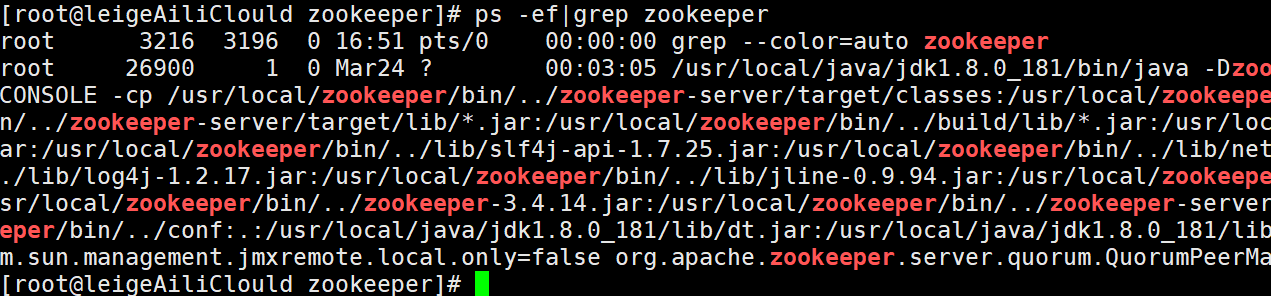
### 修改conf/server.xml



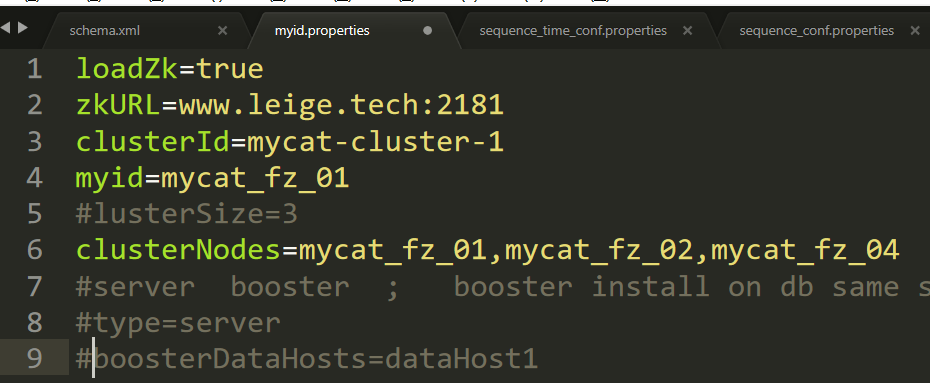
### 修改conf/schema.xml



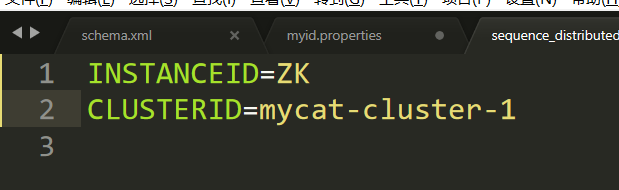
### 启动zookeeper



### 修改conf/myid.properties



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



INSTANCEID=ZK #代表使用zk

CLUSTERID=mycat-cluster-1 #与myid.properties中的CLUSTERID设置的值相同

### 配置需要使用自增ID的sequence，修改sequence\_conf.properties文件

增加一段自己的配置

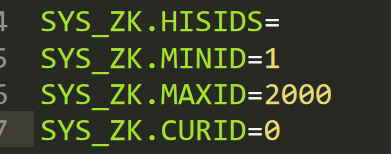
# self define sequence

SYS\_ZK.HISIDS= #可以不填写

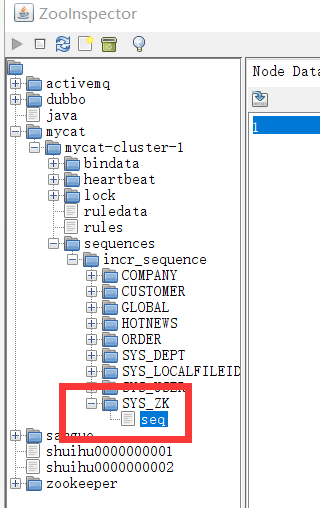
SYS\_ZK.MINID=1 #某线程当前区间内最小值

SYS\_ZK.MAXID=2000 #某线程当前区间内最大值

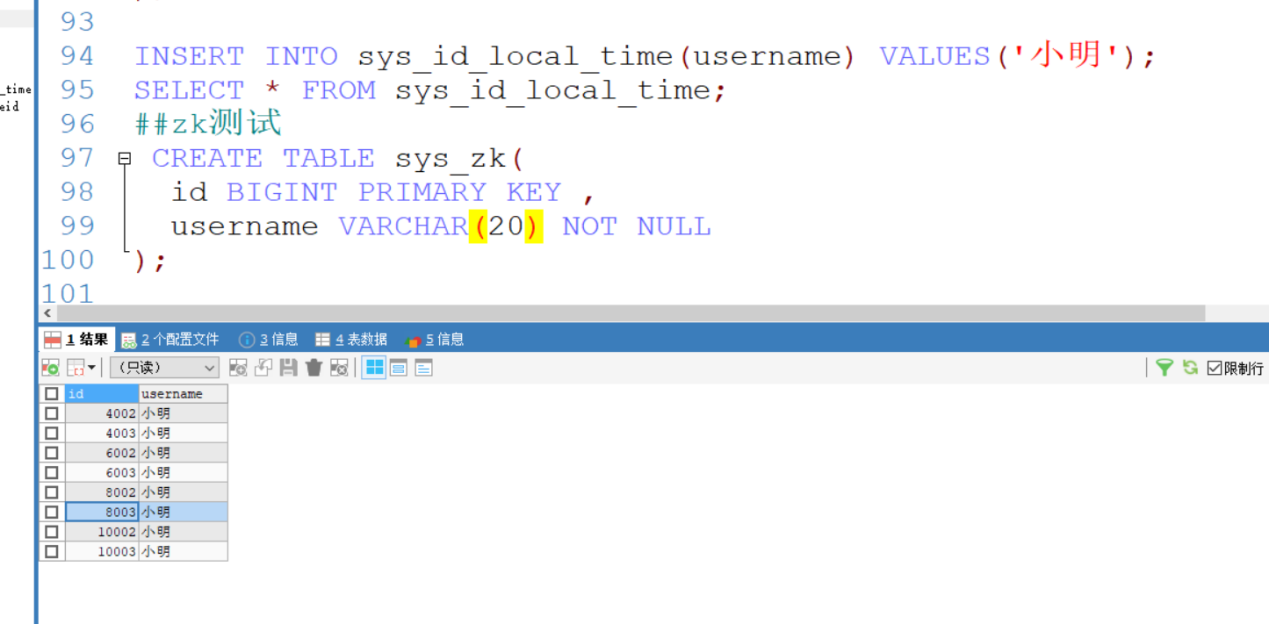
SYS\_ZK.CURID=0 #某线程当前区间内当前值



### 配置覆盖重启



### 创建表插入测试



# 18【面试】全局主键生成方式-雪花算法

## 1，snowflake算法【雪花算法】概述

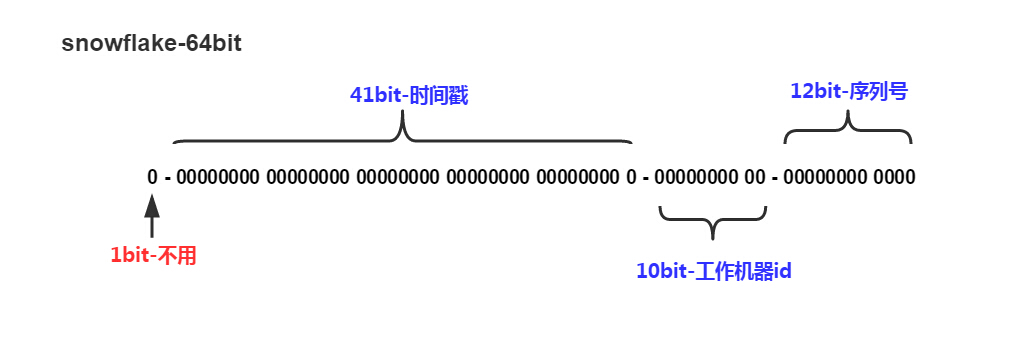
twitter开源分布式生成id算法。

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

## 2，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

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

## 3，源码

/\*\*

\* Twitter\_Snowflake<br>

\* SnowFlake的结构如下(每部分用-分开):<br>

\* 0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000 <br>

\* 1位标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0<br>

\* 41位时间截(毫秒级)，注意，41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截)

\* 得到的值），这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的（如下下面程序IdWorker类的startTime属性）。

\* 41位的时间截，可以使用69年，年T = (1L << 41) / (1000L \* 60 \* 60 \* 24 \* 365) = 69<br>

\* 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId<br>

\* 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号<br>

\* 加起来刚好64位，为一个Long型。<br>

\* SnowFlake的优点是，整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞(由数据中心ID和机器ID作区分)，并且效率较高，

\* 经测试，SnowFlake每秒能够产生26万ID左右。

\*/

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);

}

}

}

## 4，使用Hutool封装

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

### 4.1，引入pom.xml

<dependency>

<groupId>cn.hutool</groupId>

<artifactId>hutool-all</artifactId>

<version>5.2.4</version>

</dependency>

### 4.2，创建封装类

 /\*\*

\* @program: activemq-code

\* @author: 雷哥

\* @create: 2020-03-25 02:14

\*\*/

@Component

@Log4j

public class IdGeneratorSnowflake {

​

​

private long workId=0;

private long datacenterId=1;

private Snowflake snowflake= IdUtil.createSnowflake(workId,datacenterId);

​

@PostConstruct//启动项目时加载

public void init(){

try {

workId= NetUtil.ipv4ToLong(NetUtil.getLocalhostStr());

log.info("当前机工的workdId:"+workId);

}catch (Exception e){

e.printStackTrace();

log.warn("当前机器的workID获取失败",e);

workId=NetUtil.getLocalhostStr().hashCode();

}

​

}

​

/\*\*

\* 生成Id

\* @return

\*/

public synchronized long snowflakeId(){

return snowflake.nextId();

}

​

}

