**雪花算法（SnowFlake）Java实现**

有这么一种说法，自然界中并不存在两片完全一样的雪花的。每一片雪花都拥有自己漂亮独特的形状、独一无二。雪花算法也表示生成的ID如雪花般独一无二。



生成主键的方法：

1.数据库自增-缺点是分布式主键会重复

2.UUID-缺点是生成的主键是无序字符串，没有意义，而且大量数据查询比较慢

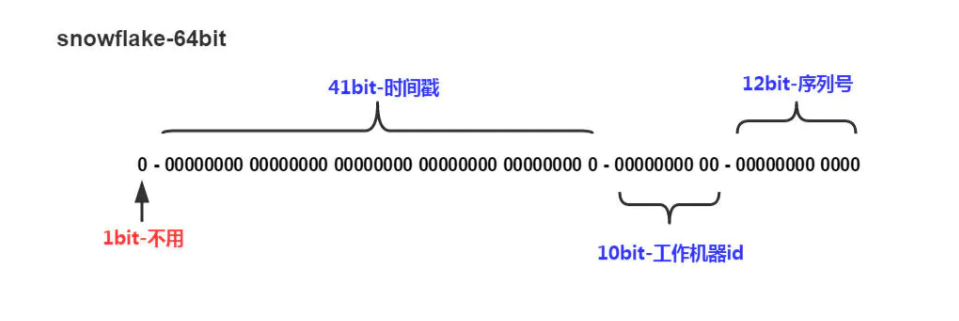
3.雪花算法，可以在1024个分布式机器上同一毫秒内，每台机器生成4096个数字ID。

各分布式生成的id不相重复。

**雪花算法的特点(自增、有序、适合分布式场景)决定了雪花算法适合分布式环境中生成ID。**

## 算法原理

SnowFlake算法生成id的结果是一个64bit大小的整数，它的结构如下图：



 **1bit符号位**，不用，因为二进制中最高位是符号位，1表示负数，0表示正数。生成的id一般都是用整数，所以最高位固定为0。

 **41bit-时间戳**，用来记录时间戳，毫秒级。  
--- 41位可以表示241-1个数字，  
--- 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0 至 241-1，减1是因为可表示的数值范围是从0开始算的，而不是1。  
---- 也就是说41位可以表示241-1个毫秒的值，转化成单位年则是(241-1)/(1000\*60\*60\*24\*365)=69年

 **10bit-工作机器id**，用来记录工作机器id。  
---- 可以部署在210=1024个节点(不同的机器上)，包括5位datacenterId和5位workerId  
---- 5位（bit）可以表示的最大正整数是25-1=31，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId

 **12bit-序列号**，序列号，用来记录同毫秒内产生的不同id。  
---- 12位（bit）可以表示的最大正整数是212-1=4095，即可以用0、1、2、3、....4095这4096个数字，来表示同一机器同一时间截（毫秒)内产生的4096个ID序号。

由于在Java中64bit的整数是long类型，所以在Java中SnowFlake算法生成的id就是long来存储的。

**SnowFlake可以保证：**

1. 所有生成的id按时间趋势递增
2. 整个分布式系统内不会产生重复id（因为有datacenterId和workerId来做区分）

## ****雪花算法的缺点****

雪花算法在单机系统上ID是递增的，但是在分布式系统多节点的情况下，所有节点的时钟并不能保证不完全同步，所以有可能会出现不是全局递增的情况。

## 算法实现（Java）

public class IdWorker{

//下面两个每个5位，加起来就是10位的工作机器id

private long workerId; //工作id

private long datacenterId; //数据id

//12位的序列号

private long sequence;

public IdWorker(long workerId, long datacenterId, long sequence){

// sanity check for workerId

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));

}

System.out.printf("worker starting. timestamp left shift %d, datacenter id bits %d, worker id bits %d, sequence bits %d, workerid %d",

timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId);

this.workerId = workerId;

this.datacenterId = datacenterId;

this.sequence = sequence;

}

//初始时间戳

private long twepoch = 1288834974657L;

//长度为5位

private long workerIdBits = 5L;

private long datacenterIdBits = 5L;

//最大值

private long maxWorkerId = -1L ^ (-1L << workerIdBits);

private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);

//序列号id长度

private long sequenceBits = 12L;

//序列号最大值

private long sequenceMask = -1L ^ (-1L << sequenceBits);

//工作id需要左移的位数，12位

private long workerIdShift = sequenceBits;

//数据id需要左移位数 12+5=17位

private long datacenterIdShift = sequenceBits + workerIdBits;

//时间戳需要左移位数 12+5+5=22位

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

//上次时间戳，初始值为负数

private long lastTimestamp = -1L;

public long getWorkerId(){

return workerId;

}

public long getDatacenterId(){

return datacenterId;

}

public long getTimestamp(){

return System.currentTimeMillis();

}

//下一个ID生成算法

public synchronized long nextId() {

long timestamp = timeGen();

//获取当前时间戳如果小于上次时间戳，则表示时间戳获取出现异常

if (timestamp < lastTimestamp) {

System.err.printf("clock is moving backwards. Rejecting requests until %d.", lastTimestamp);

throw new RuntimeException(String.format("Clock moved backwards. Refusing to generate id for %d milliseconds",

lastTimestamp - timestamp));

}

//获取当前时间戳如果等于上次时间戳（同一毫秒内），则在序列号加一；否则序列号赋值为0，从0开始。

if (lastTimestamp == timestamp) {

sequence = (sequence + 1) & sequenceMask;

if (sequence == 0) {

timestamp = tilNextMillis(lastTimestamp);

}

} else {

sequence = 0;

}

//将上次时间戳值刷新

lastTimestamp = timestamp;

/\*\*

\* 返回结果：

\* (timestamp - twepoch) << timestampLeftShift) 表示将时间戳减去初始时间戳，再左移相应位数

\* (datacenterId << datacenterIdShift) 表示将数据id左移相应位数

\* (workerId << workerIdShift) 表示将工作id左移相应位数

\* | 是按位或运算符，例如：x | y，只有当x，y都为0的时候结果才为0，其它情况结果都为1。

\* 因为个部分只有相应位上的值有意义，其它位上都是0，所以将各部分的值进行 | 运算就能得到最终拼接好的id

\*/

return ((timestamp - twepoch) << timestampLeftShift) |

(datacenterId << datacenterIdShift) |

(workerId << workerIdShift) |

sequence;

}

//获取时间戳，并与上次时间戳比较

private long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {

timestamp = timeGen();

}

return timestamp;

}

//获取系统时间戳

private long timeGen(){

return System.currentTimeMillis();

}

//---------------测试---------------

public static void main(String[] args) {

IdWorker worker = new IdWorker(1,1,1);

for (int i = 0; i < 30; i++) {

System.out.println(worker.nextId());

}

}

}

算法中大量使用位运算，这里不对位运算做过多解释，代码的详细解释参考[煲煲菜的博客](https://links.jianshu.com/go?to=https%3A%2F%2Fsegmentfault.com%2Fa%2F1190000011282426)