Twitter的SnowFlake雪花算法

# Twitter的雪花算法—SnowFlake

## SnowFlake算法背景

Twitter-Snowflake算法产生的背景相当简单，为了满足**Twitter每秒上万条消息的请求**，每条消息都必须分配一条唯一的id，这些id还需要一些大致的顺序（方便客户端排序），并且在分布式系统中不同机器产生的id必须不同。

SnowFlake --- 雪花

**flake**  英 [fleɪk] 美 [flek]

n. 小薄片，（尤指）碎片;火星，火花; vi.（成小薄片）脱落，剥落

vt. 把（鱼、食物等）切成薄片

## Twitter Snowflake算法的应用及优缺点

### 分布式应用场景

Twitter Snowflake算法是用来在**分布式场景下生成唯一ID的**。

**举个栗子**：我们有10台分布式MySql服务器，我们的系统每秒能生成10W条数据插入到这10台机器里，现在我们需要为每一条数据生成一个全局唯一的ID， 并且这些 ID 有大致的顺序。

### 优缺点

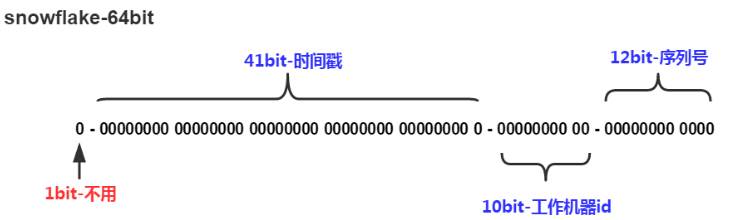
优点：性能优，不占用带宽，趋势递增；

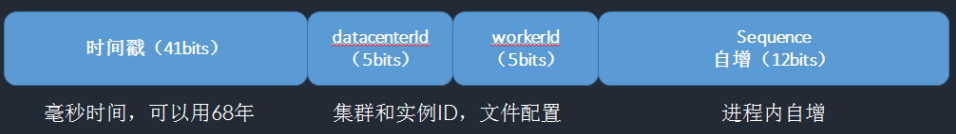
缺点：依赖服务器时间，需要时钟同步。

## ID结构

**SnowFlake算法核心**：把**时间戳，工作机器id，序列号**组合在一起。

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：





如图：最后生成的ID是一个long类型，long占64bit，符号位占1位，剩下63位，我们将这63位拆分成4段，就可以表示：某一毫秒内的某一集群内的某一机器的第几个ID。

可以分成5部分：**1位+41位+10位+12位。**

### **1位，未使用，固定为0**。二进制中最高位为1表示负数，但是生成的id一般都使用正整数，所以这个最高位固定是0；正好作为64位id的最高位，为0，即long类型值为正数；

### **41位，用来记录时间戳（毫秒）**;41位表示的数字范围可以使用69年，也就是说41位可以表示毫秒值，转化成单位年则是69年；

### **10位，用来记录节点id**；最多支持部署1024个节点，（节点一般是由5位**数据中心编号datacenterId**和5位**机器编号workerId**组成）；

5位（bit）可以表示的最大正整数是31，即可以用0、1、2、3、....31这32个数字，来表示不同的**datecenterId或workerId**；

### **12位，序列号**，用来记录**同毫秒内**产生的不同id，意味着每个节点每毫秒可以产生4096个ID序号。

12位（bit）可以表示的最大正整数是，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4095个ID序号。

在上面的字符串中，第一位为未使用（实际上也可作为**long的符号位**），接下来的41位为**毫秒级时间**，然后**5位datacenter标识位，5位机器ID**（并不算标识符，实际是为**线程标识**），然后**12位该毫秒内的当前毫秒内的计数**，加起来刚好64位，为一个Long型。

这样的好处是，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datacenter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，完全满足需要。

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：

所有生成的id按时间趋势递增；

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）；

## ID 生成策略

 \* **毫秒级时间41位** + **机器ID 10位** + **毫秒内序列12位**。

 \* 0           41     51     64

+-----------+------+------+

|time       |pc    |seq   |

+-----------+------+------+

 \* 最高位bit标记为不可用

 \*  前41bits是以微秒为单位的timestamp。

 \*  接着10bits是事先配置好的机器ID。

 \*  最后12bits是累加计数器。

 \*  macheine id(10bits)标明最多只能有1024台机器同时产生ID，sequence number(12bits)也标明1台机器1ms中最多产生4096个ID。

## 问题：Q&A

### 问题1：有人会问：为什么时间戳要占41位？sequence要占12位？而其他两个要各占5位？

### 答：这是**根据具体需求来分**的，你也可以自己再去将这63为重新拆分。例如：sequence占12位就可以在同一毫秒内的同一集群的同一机器上同时有2^12 - 1 个线程。

### **问题2：twepoch 为什么要等于1288834974657L 而不等于其他数？**

### 答： 1288834974657 是 (Thu, 04 Nov 2010 01:42:54 GMT) 这一时刻到1970-01-01 00:00:00时刻所经过的毫秒数。当前时刻减去1288834974657 的值刚好在2^41 里，因此占41位。 所以这个数是为了让时间戳占41位才特地算出来的。

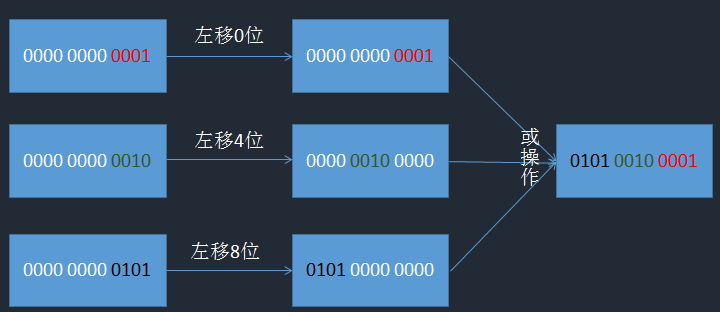
### **问题3：类似这种long maxWorkerId = -1L ^ (-1L << workerIdBits);操作是什么意思？**

### 答： **-1L ^ (-1L << n)**表示占n个bit的数字的最大值是多少。举个栗子：-1L ^ (-1L << 2)等于10进制的3 ，即二进制的11表示十进制3。

注意：计算机存放数字都是存放数字的补码，正数的原码、补码、反码都一样，负数的补码是其反码加一。符号位做取反操作时不变，做逻辑与、或、非、异或操作时要参与运算。

再来个栗子：   
-1L原码 : 1000 0001   
-1L反码 : 1111 1110   
-1L补码 : 1111 1111   
-1L<<5 : 1110 0000   
1111 1111 ^ 1110 0000 : 0001 1111   
0001 1111是正数，所以补码、反码、原码都一样，所以0001 1111是31

### **问题4：((timestamp - twepoch) << timestampLeftShift) | (datacenterId << datacenterIdShift) | (workerId << workerIdShift) | sequence是什么意思？** 答：我只发图不说话



## 扩展

在理解了这个算法之后，其实还有一些扩展的事情可以做：

### 根据自己业务修改**每个位段存储的信息**。算法是通用的，可以根据自己需求适当调整每段的大小以及存储的信息。

### **解密id**，由于id的每段都保存了特定的信息，所以拿到一个id，应该可以尝试反推出原始的每个段的信息。反推出的信息可以帮助我们分析。比如作为订单，可以知道该订单的生成日期，负责处理的数据中心等等。

## 重要代码解析

使用mask的目的是：**防止溢出**。

sequence = (sequence + 1) & sequenceMask;//防止溢出

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

return ((timestamp - twepoch) << timestampLeftShift) |  
 (datacenterId << datacenterIdShift) |  
 (workerId << workerIdShift) |  
 sequence;//利用左移运算得到最终的ID

负数的补码：方法1：**补码 = 反码 + 1**；<反码=补码-1>

方法2：补码 = （原码 - 1）再取反码。



## SnowFlake算法的Java代码

public class SnowFlakeAlgorithm {

/\*\*

\* Twitter的雪花算法SnowFlake

\* DateTime:2018-12-18 22:23:00

\* 1bit + 41bit + 5bit + 5bit + 12bit

\*

\* 时间戳(毫秒数)是根据当前时间获取的，datacenterId和workerId都是节点固定的值，

\* 因此，只需要确定sequence即可。

\* 这里最重要的就是序列号sequence的生成：需要判断是否为同一毫秒内；

\* （1）、若为同一毫秒，则sequence加一即可，若溢出(变为0)，需要等待下一毫秒的到来；

\* （2）、若不为同一毫秒，sequence置0即可。

\*

\* 代码实现原理：

\* ID由四部分组成，确定四部分即可。

\* 首先定义各个部分占用的比特位数和组合时需要左移的位数；

\* 核心方法:nextId

\*/

private long twepoch = 1288834974657L;

private long workerIdBits = 5L;//机器编号5位

private long datacenterIdBits = 5L;//数据中心编号5位

private long maxWorkerId = -1L ^ (-1L << workerIdBits);//00...00011111;即31

private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);//00...00011111;即31

//需要左移的位数

private long sequenceBits = 12L;//序列号12位

private long workerIdShift = sequenceBits;

private long datacenterIdShift = sequenceBits + workerIdBits;

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

//序列号

private long sequenceMask = -1L ^ (-1L << sequenceBits);//0...0FFF；使用mask的目的是防止溢出

//4部分：41bit + 5bit + 5bit + 12bit

private long lastTimestamp = -1L;

private long workerId;//机器编号

private long datacenterId;//数据中心编号

private long sequence;//序列号

/\*\*

\* 构造方法

\*

\* @param workerId

\* @param datacenterId

\* @param sequence

\*/

public SnowFlakeAlgorithm(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);

System.out.println();//换行

this.workerId = workerId;

this.datacenterId = datacenterId;

this.sequence = sequence;

}

/\*\*

\* 核心方法：获取下一个Id

\*

\* @return Id

\*/

public synchronized long nextId() {

long timestamp = timeGen();

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));

}

if (lastTimestamp == timestamp) {//同一个毫秒内，利用序列号区别

sequence = (sequence + 1) & sequenceMask;//防止溢出

if (sequence == 0) {

timestamp = tilNextMillis(lastTimestamp);

}

} else {//不是一个毫秒数，则序列号置0

sequence = 0;

}

lastTimestamp = timestamp;//更新上一个时间戳

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

(datacenterId << datacenterIdShift) |

(workerId << workerIdShift) |

sequence;//利用左移运算得到最终的ID

}

/\*\*

\* 直到下一毫秒到来

\* 通过while循环实现

\* @param lastTimestamp

\* @return

\*/

private long tilNextMillis(long lastTimestamp) {

long timestamp = timeGen();

while (timestamp <= lastTimestamp) {//确保下一毫秒的到来

timestamp = timeGen();

}

return timestamp;

}

private long timeGen() {

return System.currentTimeMillis();//当前系统的毫秒值

}

//---------------测试---------------

public static void main(String[] args) {

SnowFlakeAlgorithm snowFlake = new SnowFlakeAlgorithm(1, 1, 1);

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

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

}

}

}