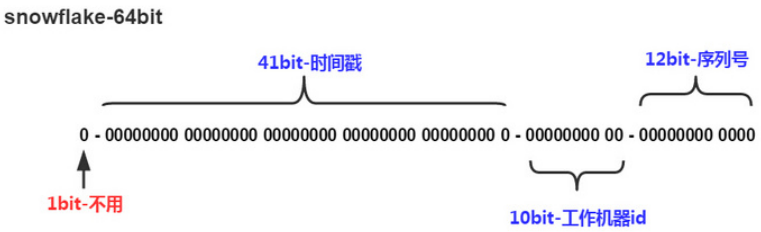
理解分布式id生成算法SnowFlake

原 分布式 java 算法 首席卖萌官 2017年09月20日发布

> 分布式id生成算法的有很多种，Twitter的SnowFlake就是其中经典的一种。

概述

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：



- 1位，不用。二进制中最高位为1的都是负数，但是我们生成的id一般都使用整数，所以这个最高位固定是0
- 41位，用来记录时间戳（毫秒）。
  - 41位可以表示$2^{41}-1$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 $2^{41}-1$，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$2^{41}-1$个毫秒的值，转化成单位年则是$(2^{41}-1)/(1000*60*60*24*365)=69$年
- 10位，用来记录工作机器id。
  - 可以部署在$2^{10}=1024$个节点，包括 5位datacenterId 和 5位workerId
  - 5位（bit）可以表示的最大正整数是$2^5-1=31$，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId
- 12位，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$2^{12}-1=4096$，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：

- 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）

Talk is cheap, show you the code

以下是Twitter官方原版的，用Scala写的，（我也不懂Scala，当成Java看即可）：

```
private[this] val rand = new Random

val twepoch = 1288834974657L

private[this] val workerIdBits = 5L
private[this] val datacenterIdBits = 5L
private[this] val maxWorkerId = -1L ^ (-1L << workerIdBits)
private[this] val maxDatacenterId = -1L ^ (-1L << datacenterIdBits)
private[this] val sequenceBits = 12L

private[this] val workerIdShift = sequenceBits
private[this] val datacenterIdShift = sequenceBits + workerIdBits
private[this] val timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits
private[this] val sequenceMask = -1L ^ (-1L << sequenceBits)

private[this] var lastTimestamp = -1L

// sanity check for workerId
if (workerId > maxWorkerId || workerId < 0) {
  exceptionCounter.incr(1)
```

```
    throw new IllegalArgumentException("worker Id can't be greater than %d or less than 0".format
  }

  if (datacenterId > maxDatacenterId || datacenterId < 0) {
    exceptionCounter.incr(1)
```

Scala是一门可以编译成字节码的语言，简单理解是在Java语法基础上加上了很多语法糖，例如不用每条语句后写分号，可以使用动态类型等等。抱着试一试的心态，我把Scala版的代码“翻译”成Java版本的，对scala代码改动的地方如下：

```
/** Copyright 2010-2012 Twitter, Inc.*/
package com.twitter.service.snowflake

import com.twitter.ostrich.stats.Stats
import com.twitter.service.snowflake.gen._
import java.util.Random
import com.twitter.logging.Logger

/**
 * An object that generates IDs.
 * This is broken into a separate class in case
 * we ever want to support multiple worker threads
 * per process
 */
class IdWorker(                                     // |
    val workerId: Long,                             // |
    val datacenterId: Long,                         // |<--这部分改成Java的构造函数形式
    private val reporter: Reporter,//日志相关，删       // |
    var sequence: Long = 0L)                        // |
       extends Snowflake.Iface { //接口找不到，删       // |

  private[this] def genCounter(agent: String) = {                    // |
    Stats.incr("ids_generated")                                      // |
    Stats.incr("ids_generated_%s".format(agent))                     // |<--错误、日志处理相关，删
```

改出来的Java版：

```
public class IdWorker{

    private long workerId;
    private long datacenterId;
    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
        }
        if (datacenterId > maxDatacenterId || datacenterId < 0) {
            throw new IllegalArgumentException(String.format("datacenter Id can't be greater than
        }
        System.out.printf("worker starting. timestamp left shift %d, datacenter id bits %d, worke
                timestampLeftShift, datacenterIdBits, workerIdBits, sequenceBits, workerId);

        this.workerId = workerId;
        this.datacenterId = datacenterId;
        this.sequence = sequence;
    }

    private long twepoch = 1288834974657L;
```

代码理解

上面的代码中，有部分位运算的代码，如：

```
sequence = (sequence + 1) & sequenceMask;

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

return ((timestamp - twepoch) << timestampLeftShift) |
        (datacenterId << datacenterIdShift) |
        (workerId << workerIdShift) |
        sequence;
```

为了能更好理解，我对相关知识研究了一下。

负数的二进制表示

在计算机中，负数的二进制是用 补码 来表示的。
假设我是用Java中的int类型来存储数字的，

int类型的大小是32个二进制位（bit），即4个字节（byte）。（1 byte = 8 bit）
那么十进制数字 3 在二进制中的表示应该是这样的：

```
00000000 00000000 00000000 00000011
// 3的二进制表示，就是原码
```

那数字 -3 在二进制中应该如何表示？
我们可以反过来想想，因为-3+3=0，
在二进制运算中 把-3的二进制看成未知数x来求解 ，
求解算式的二进制表示如下：

```
   00000000 00000000 00000000 00000011 //3，原码
+  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx //-3，补码
-----------------------------------------------
   00000000 00000000 00000000 00000000
```

反推x的值，3的二进制加上什么值才使结果变成 00000000 00000000 00000000 00000000 ？：

```
   00000000 00000000 00000000 00000011 //3，原码
+  11111111 11111111 11111111 11111101 //-3，补码
-----------------------------------------------
 1 00000000 00000000 00000000 00000000
```

反推的思路是3的二进制数从最低位开始逐位加1，使溢出的1不断向高位溢出，直到溢出到第33位。然后由于int类型最多只能保存32个二进制位，所以最高位的1溢出了，剩下的32位就成了（十进制的）0。

补码的意义就是可以拿补码和原码（3的二进制）相加，最终加出一个“溢出的0”

以上是理解的过程，实际中记住**公式**就很容易算出来：

- 补码 = 反码 + 1
- 补码 =（原码 - 1）再取反码

因此 -1 的二进制应该这样算：

```
00000000 00000000 00000000 00000001 //原码：1的二进制
11111111 11111111 11111111 11111110 //取反码：1的二进制的反码
11111111 11111111 11111111 11111111 //加1：-1的二进制表示（补码）
```

用位运算计算n个bit能表示的最大数值

比如这样一行代码：

```
    private long workerIdBits = 5L;
    private long maxWorkerId = -1L ^ (-1L << workerIdBits);
```

上面代码换成这样看方便一点：

```
long maxWorkerId = -1L ^ (-1L << 5L)
```

咋一看真的看不准哪个部分先计算，于是查了一下Java运算符的优先级表：

优先级	运算符	结合性
1	() [] .	从左到右
2	! +(正) -(负) ~ ++ --	从右向左
3	* / %	从左向右
4	+(加) -(减)	从左向右
5	<< >> >>>	从左向右
6	< <= > >= instanceof	从左向右
7	== !=	从左向右
8	&(按位与)	从左向右
9	^	从左向右
10	|	从左向右
11	&&	从左向右
12	||	从左向右
13	?:	从右向左
14	= += -= *= /= %= &= |= ^= ~= <<= >>= >>>=	从右向左

所以上面那行代码中，运行顺序是：

- -1 左移 5，得结果a

- -1 异或 a

`long maxWorkerId = -1L ^ (-1L << 5L)` 的二进制运算过程如下：

-1 左移 5，得结果a ：

```
       11111111 11111111 11111111 11111111 //-1的二进制表示（补码）
 11111 11111111 11111111 11111111 11100000 //高位溢出的不要，低位补0
       11111111 11111111 11111111 11100000 //结果a
```

-1 异或 a ：

```
       11111111 11111111 11111111 11111111 //-1的二进制表示（补码）
   ^   11111111 11111111 11111111 11100000 //两个操作数的位中，相同则为0，不同则为1
-------------------------------------------------------------------------------
       00000000 00000000 00000000 00011111 //最终结果31
```

最终结果是31，二进制 `00000000 00000000 00000000 00011111` 转十进制可以这么算：

$$2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 16 + 8 + 4 + 2 + 1 = 31$$

那既然现在知道算出来 `long maxWorkerId = -1L ^ (-1L << 5L)` 中的 `maxWorkerId = 31` ，有什么含义？为什么要用左移5来算？如果你看过 概述 部分，请找到这段内容看看：

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

`-1L ^ (-1L << 5L)` 结果是 `31` ，$2^5 - 1$的结果也是 `31` ，所以在代码中， `-1L ^ (-1L << 5L)` 的写法是 利用位运算计算出5位能表示的最大正整数是多少

用mask防止溢出

有一段有趣的代码：

```
sequence = (sequence + 1) & sequenceMask;
```

分别用不同的值测试一下，你就知道它怎么有趣了：

```
long seqMask = -1L ^ (-1L << 12L); //计算12位能耐存储的最大正整数，相当于：2^12-1 = 4095
System.out.println("seqMask: "+seqMask);
System.out.println(1L & seqMask);
System.out.println(2L & seqMask);
System.out.println(3L & seqMask);
System.out.println(4L & seqMask);
System.out.println(4095L & seqMask);
System.out.println(4096L & seqMask);
System.out.println(4097L & seqMask);
System.out.println(4098L & seqMask);


/**
seqMask: 4095
1
2
3
4
4095
0
1
2
*/
```

这段代码通过 位与 运算保证计算的结果范围始终是 0-4095 ！

用位运算汇总结果

还有另外一段诡异的代码：

```
return ((timestamp - twepoch) << timestampLeftShift) |
        (datacenterId << datacenterIdShift) |
        (workerId << workerIdShift) |
        sequence;
```

为了弄清楚这段代码，

首先 **需要计算一下相关的值：**

```
    private long twepoch = 1288834974657L; //起始时间戳，用于用当前时间戳减去这个时间戳，算出偏移量

    private long workerIdBits = 5L; //workerId占用的位数：5
    private long datacenterIdBits = 5L; //datacenterId占用的位数：5
    private long maxWorkerId = -1L ^ (-1L << workerIdBits);  // workerId可以使用的最大数值：31
    private long maxDatacenterId = -1L ^ (-1L << datacenterIdBits); // datacenterId可以使用的最大数值
    private long sequenceBits = 12L;//序列号占用的位数：12

    private long workerIdShift = sequenceBits; // 12
    private long datacenterIdShift = sequenceBits + workerIdBits; // 12+5 = 17
    private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits; // 12+5+5 = 2
    private long sequenceMask = -1L ^ (-1L << sequenceBits);//4095

    private long lastTimestamp = -1L;
```

其次 **写个测试，把参数都写死，并运行打印信息，方便后面来核对计算结果：**

```
    //---------------测试---------------
    public static void main(String[] args) {
        long timestamp = 1505914988849L;
        long twepoch = 1288834974657L;
        long datacenterId = 17L;
        long workerId = 25L;
        long sequence = 0L;

        System.out.printf("\ntimestamp: %d \n",timestamp);
        System.out.printf("twepoch: %d \n",twepoch);
        System.out.printf("datacenterId: %d \n",datacenterId);
        System.out.printf("workerId: %d \n",workerId);
        System.out.printf("sequence: %d \n",sequence);
        System.out.println();
        System.out.printf("(timestamp - twepoch): %d \n",(timestamp - twepoch));
        System.out.printf("((timestamp - twepoch) << 22L): %d \n",((timestamp - twepoch) << 22L));
        System.out.printf("(datacenterId << 17L): %d \n" ,(datacenterId << 17L));
        System.out.printf("(workerId << 12L): %d \n",(workerId << 12L));
        System.out.printf("sequence: %d \n",sequence);

        long result = ((timestamp - twepoch) << 22L) |
                (datacenterId << 17L) |
                (workerId << 12L) |
```

代入位移的值得之后，就是这样：

```
return ((timestamp - 1288834974657) << 22) |
        (datacenterId << 17) |
        (workerId << 12) |
        sequence;
```

对于尚未知道的值，我们可以先看看 概述 中对SnowFlake结构的解释，再代入在合法范围的值(windows系统可以用计算器方便计算这些值的二进制)，来了解计算的过程。
当然，由于我的测试代码已经把这些值写死了，那直接用这些值来手工验证计算结果即可：

```
        long timestamp = 1505914988849L;
        long twepoch = 1288834974657L;
        long datacenterId = 17L;
        long workerId = 25L;
        long sequence = 0L;
```

```
设：timestamp  = 1505914988849，twepoch = 1288834974657
1505914988849 - 1288834974657 = 217080014192 (timestamp相对于起始时间的毫秒偏移量)，其(a)二进制左移22位

                          |<--这里开始左右22位
00000000 00000000 000000|00 00110010 10001010 11111010 00100101 01110000 // a = 217080014192
00001100 10100010 10111110 10001001 01011100 00|000000 00000000 00000000 // a左移22位后的值(la)
                                                |<--这里后面的位补0
```

```
设：datacenterId  = 17，其（b）二进制左移17位计算过程如下：

                    |<--这里开始左移17位
00000000 00000000 0|0000000 00000000 00000000 00000000 00000000 00010001 // b = 17
00000000 00000000 00000000 00000000 00000000 0010001|0 00000000 00000000 // b左移17位后的值(lb)
                                                     |<--这里后面的位补0
```

```
设：workerId  = 25，其（c）二进制左移12位计算过程如下：

            |<--这里开始左移12位
00000000 0000|0000 00000000 00000000 00000000 00000000 00011001 // c = 25
00000000 00000000 00000000 00000000 00000000 00000001 1001|0000 00000000 // c左移12位后的值(lc)
                                                          |<--这里后面的位补0
```

```
设：sequence = 0，其二进制如下：

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 // sequence = 0
```

现在知道了每个部分左移后的值(la,lb,lc)，代码可以简化成下面这样去理解：

```
return ((timestamp - 1288834974657) << 22) |
        (datacenterId << 17) |
        (workerId << 12) |
        sequence;
-----------------------------
           |
           |简化
          \|/
-----------------------------
return (la) |
        (lb) |
        (lc) |
        sequence;
```

上面的管道符号 | 在Java中也是一个位运算符。其含义是：
x的第n位和y的第n位 只要有一个是1，则结果的第n位也为1，否则为0 ，因此，我们对四个数的 位或运算 如下：

```
 1  |                  41                      |  5  |   5  |     12

   0|0001100 10100010 10111110 10001001 01011100 00|00000|0 0000|0000 00000000 //la
   0|0000000 00000000 00000000 00000000 00000000 00|10001|0 0000|0000 00000000 //lb
   0|0000000 00000000 00000000 00000000 00000000 00|00000|1 1001|0000 00000000 //lc
or 0|0000000 00000000 00000000 00000000 00000000 00|00000|0 0000|0000 00000000 //sequence
-----------------------------------------------------------------------------------------
   0|0001100 10100010 10111110 10001001 01011100 00|10001|1 1001|0000 00000000 //结果：910499571847
```

结果计算过程：
1) 从至左列出1出现的下标（从0开始算）：

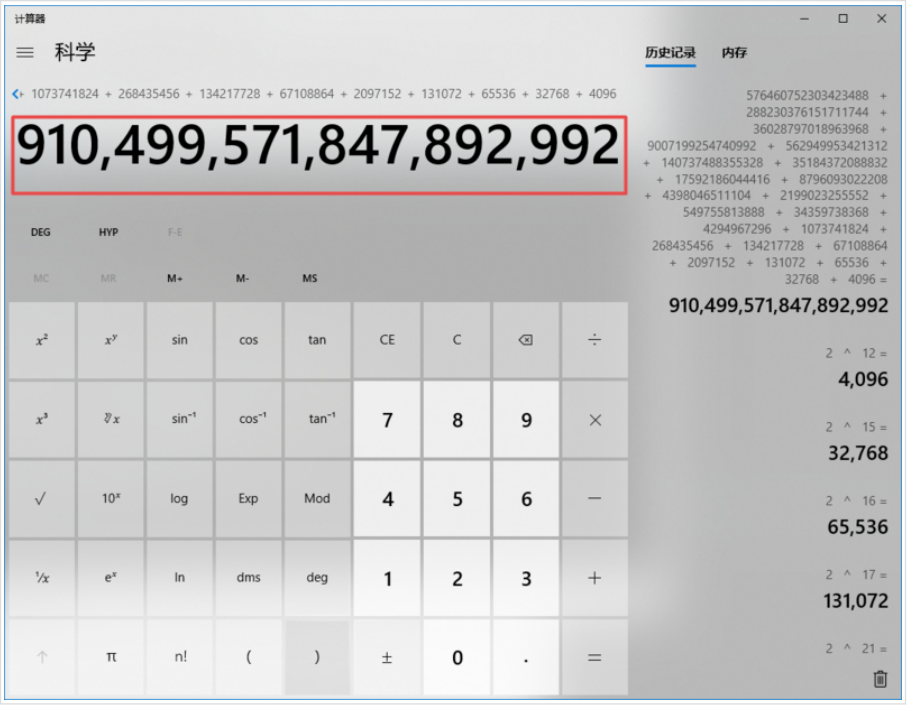
```
0000  1   1   00  1   0  1  000  1   0  1  0  1  1  1  1  1  0 1   000 1 00 1  0 1  0   1  1  1  00
      59  58      55     53      49     47    45 44 43 42 41   39      35   32   30     28 27 26
```

2) 各个下标作为2的幂数来计算，并相加：

$$2^{59} + 2^{58} + 2^{55} + 2^{53} + 2^{49} + 2^{47} + 2^{45} + 2^{44} + 2^{43} + 2^{42} + 2^{41} + 2^{39} + 2^{35} + 2^{32} + 2^{30} + 2^{28} + 2^{27} + 2^{26} + 2^{21} + 2^{17} + 2^{16} + 2$$

```
    2^59}  : 576460752303423488
    2^58}  : 288230376151711744
    2^55}  :  36028797018963968
    2^53}  :   9007199254740992
    2^49}  :    562949953421312
    2^47}  :    140737488355328
    2^45}  :     35184372088832
    2^44}  :     17592186044416
    2^43}  :      8796093022208
    2^42}  :      4398046511104
    2^41}  :      2199023255552
    2^39}  :       549755813888
    2^35}  :        34359738368
    2^32}  :         4294967296
    2^30}  :         1073741824
    2^28}  :          268435456
    2^27}  :          134217728
    2^26}  :           67108864
    2^21}  :            2097152
    2^17}  :             131072
    2^16}  :              65536
    2^15}  :              32768
+   2^12}  :               4096
----------------------------------------
            910499571847892992
```

计算截图：



跟测试程序打印出来的结果一样，手工验证完毕！

观察

```
  1  |                  41                      |  5  |   5  |     12

    0|0001100 10100010 10111110 10001001 01011100 00|     |      |             //la
    0|                                           |10001|      |             //lb
    0|                                           |     |1 1001|             //lc
 or 0|                                           |     |      |0000 00000000 //sequence
 ------------------------------------------------------------------------------------------
    0|0001100 10100010 10111110 10001001 01011100 00|10001|1 1001|0000 00000000 //结果：9104995718478
```

上面的64位我按1、41、5、5、12的位数截开了，方便观察。

- 纵向 观察发现:
  - 在41位那一段，除了la一行有值，其它行（lb、lc、sequence）都是0，（我爸其它）
  - 在左起第一个5位那一段，除了lb一行有值，其它行都是0
  - 在左起第二个5位那一段，除了lc一行有值，其它行都是0
  - 按照这规律，如果sequence是0以外的其它值，12位那段也会有值的，其它行都是0
- 横向 观察发现:
  - 在la行，由于左移了5+5+12位，5、5、12这三段都补0了，所以la行除了41那段外，其它肯定都是0
  - 同理，lb、lc、sequnece行也以此类推
  - 正因为左移的操作，使四个不同的值移到了SnowFlake理论上相应的位置，然后四行做 位或 运算（只要有1结果就是1），就把4段的二进制数合并成一个二进制数。

结论：
所以，在这段代码中

```
return ((timestamp - 1288834974657) << 22) |
        (datacenterId << 17) |
        (workerId << 12) |
        sequence;
```

左移运算是为了将数值移动到对应的段(41、5、5，12那段因为本来就在最右，因此不用左移)。

然后对每个左移后的值(la、lb、lc、sequence)做位或运算，是为了把各个短的数据合并起来，合并成一个二进制数。

最后转换成10进制，就是最终生成的id

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：

1. 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。
2. 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。

2017年09月20日发布 ···

赞 | 5　收藏 | 22

你可能感兴趣的文章

分布式事务概念梳理　77 浏览

推荐十款java开源中文分词组件　3 收藏，864 浏览

一个两年Java的面试总结　747 收藏，6.4k 浏览

3 条评论

默认排序 | 时间排序

FullStack · 3月1日

请问workid具体怎么保证唯一呢，比如我有n个docker，有些docker会挂掉然后重启

赞　回复

不好意思，这点我也没有深入研究
— **首席卖萌官** 作者 · 3月1日

添加回复

aaaaaaxxl · 3月2日

((timestamp - 1288834974657) << 32) 超过long类型数据范围，怎么办

赞　回复

文明社会，理性评论

发布评论

首席卖萌官

374 声望

关注作者

发布于专栏

不折腾会死

怀着不折腾会死的心研究技术。多折腾多学习。

4 人关注

关注专栏