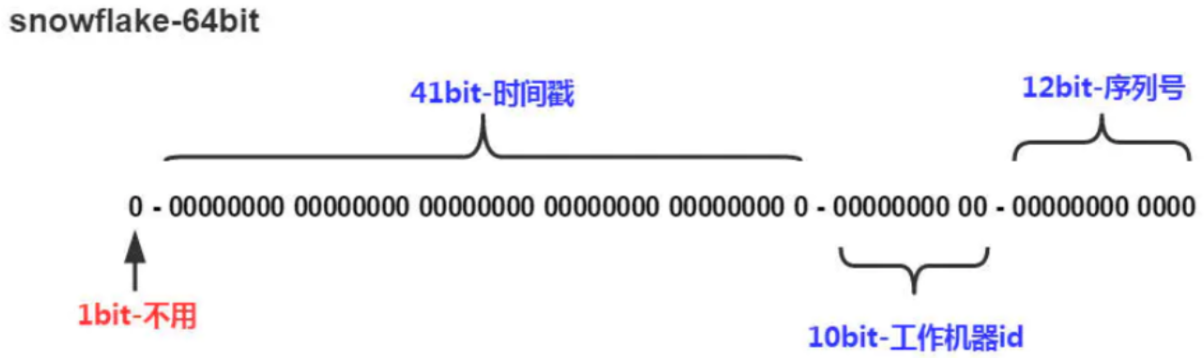
Snowflake，雪花算法是由Twitter开源的分布式ID生成算法，以划分命名空间的方式将64-bit位分割成多个部分，每个部分代表不同的含义。而 Java中64bit的整数是Long类型，所以在 Java 中 SnowFlake 算法生成的 ID 就是 long 来存储的。

位运算异或( ^ )，左移( << )，与( & )，或( | )




第1位：占用1bit，第一位为符号位，不使用。

第1部分：41位的时间戳，41-bit位可表示2^41个数，每个数代表毫秒，那么雪花算法可用的时间年限是(2^41)/(1000*60*60*24*365)=69 年的时间。

第2部分：10-bit位可表示机器数，即2^10 = 1024台机器，通常不会部署这么多台机器，（细分两部分5-bit（数据），5-bit(2^5)=32台机器）也可划分多个部分，

第3部分：12-bit位是自增序列，可表示2^12 = 4096个数。觉得一毫秒个数不够用也可以调大点

41位时间戳是固定的，时间戳转二进制的长度是41位，后面两个部分都可以灵活调正，只要注意后面位运算的位数就行.

## 一、SnowFlake代码

```
import org.springframework.util.Assert;

public class IdWorker {
 /**
 * 这两个参数可以读取配置文件
 * 这里默认写死
 *
 * @param workerId 机器标识
 * @param datacenterId 数据标识
 */
 private static SnowflakeIdWorker worker = new SnowflakeIdWorker(0, 0);

 public static long id() {
 Assert.notNull(worker, "SnowflakeIdWorker未配置!");
 return worker.nextId();
```

```
}

/**
* Twitter的分布式自增ID算法snowflake
*/
public static class SnowflakeIdWorker {
/**
* 第1部分(41位)
* 开始时间截 (2022-04-01)
*/
private final long startTime = 1648742400000L;

/**
* 第2部分（10位）
* 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;

/**
* 第3部分（12位）
* 序列在id中占的位数
*/
private final long sequenceBits = 12L;

/**
* -1L ^ (-1L << 5) = 31
* 支持的最大机器id，结果是31
*/
private final long maxWorkerId = -1L ^ (-1L << workerIdBits);

/**
* -1L ^ (-1L << 5) = 31
* 支持的最大数据标识id，结果是31
*/
private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);

```

```
56   /**
57   * -1L ^ (-1L << 12) = 4095
58   * 自增长最大值4095，0开始
59   */
60   private final long sequenceMask = -1L ^ (-1L << sequenceBits);
61
62   /**
63   * 时间截向左移22位(5+5+12)
64   */
65   private final long timestampLeftShift = sequenceBits + workerIdBits + d
atacenterIdBits;
66
67   /**
68   * 数据标识id向左移17位(12+5)
69   */
70   private final long datacenterIdShift = sequenceBits + workerIdBits;
71
72   /**
73   * 机器ID向左移12位
74   */
75   private final long workerIdShift = sequenceBits;
76
77
78   /**
79   * 工作机器ID(0~31)
80   */
81   private long workerId;
82
83   /**
84   * 数据中心ID(0~31)
85   */
86   private long datacenterId;
87
88   /**
89   * 1毫秒内序号(0~4095)
90   */
91   private long sequence = 0L;
92
93   /**
94   * 上次生成ID的时间截
95   */
```

```
96   private long lastTimestamp = -1L;
97
98   /**
99   * 构造函数
100   *
101   * @param workerId 工作ID (0~31)
102   * @param datacenterId 数据中心ID (0~31)
103   */
104   public SnowflakeIdWorker(long workerId, long datacenterId) {
105   if (workerId > maxWorkerId || workerId < 0) {
106   throw new IllegalArgumentException(String.format("worker Id can't be gr
eater than %d or less than 0", maxWorkerId));
107   }
108   if (datacenterId > maxDatacenterId || datacenterId < 0) {
109   throw new IllegalArgumentException(String.format("datacenter Id can't b
e greater than %d or less than 0", maxDatacenterId));
110   }
111   this.workerId = workerId;
112   this.datacenterId = datacenterId;
113   }
114
115   /**
116   * 获得下一个ID (该方法是线程安全的)
117   *
118   * @return SnowflakeId
119   */
120   public synchronized long nextId() {
121   long timestamp = timeGen();
122   // 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
123   if (timestamp < lastTimestamp) {
124   throw new RuntimeException(String.format("Clock moved backwards.Refusin
g to generate id for %d milliseconds", lastTimestamp - timestamp));
125   }
126   // 如果是同一时间生成的，则进行毫秒内序列
127   if (lastTimestamp == timestamp) {
128   sequence = (sequence + 1) & sequenceMask;
129   // 毫秒内序列溢出
130   //sequence == 0 ，就是1毫秒用完了4096个数
131   if (sequence == 0) {
132   // 阻塞到下一个毫秒,获得新的时间戳
```

```
133  timestamp = tilNextMillis(lastTimestamp);
134  }
135  }
136  // 时间戳改变，毫秒内序列重置
137  else {
138  sequence = 0L;
139  }
140  // 上次生成ID的时间截
141  lastTimestamp = timestamp;
142
143  // 移位并通过或运算拼到一起组成64位的ID
144  return ((timestamp - startTime) << timestampLeftShift) // 时间戳左移22位
145  | (datacenterId << datacenterIdShift) //数据标识左移17位
146  | (workerId << workerIdShift) //机器id标识左移12位
147  | sequence;
148  }
149
150  /**
151  * 阻塞到下一个毫秒，直到获得新的时间戳
152  *
153  * @param lastTimestamp 上次生成ID的时间截
154  * @return 当前时间戳
155  */
156  protected long tilNextMillis(long lastTimestamp) {
157  long timestamp = timeGen();
158  while (timestamp <= lastTimestamp) {
159  timestamp = timeGen();
160  }
161  return timestamp;
162  }
163
164  /**
165  * 返回以毫秒为单位的当前时间
166  *
167  * @return 当前时间(毫秒)
168  */
169  protected long timeGen() {
170  return System.currentTimeMillis();
171  }
172 }
```

## 二、SnowFlake代码分析，结合位运算分析

1、异或( ^ )，左移( << )

```
/**
 * -1L ^ (-1L << 5) = 31
 * 支持的最大机器id，结果是31
 */
private final long maxWorkerId = -1L ^ (-1L << workerIdBits);

/**
 * -1L ^ (-1L << 5) = 31
 * 支持的最大机房标识id，结果是31
 */
private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);

/**
 * -1L ^ (-1L << 12) = 4095
 * 自增长最大个数4095，0开始
 */
private final long sequenceMask = -1L ^ (-1L << sequenceBits);
```

这里使用到了 异或( ^ )，左移( << ) 这两个位运算

异或( ^ ) 两个数相同，结果为0，不相同则为1

左移( << ) 二进制向左移多少位，低位补0

**计算机中负数的二进制是用补码来表示的**

**补码 = 反码 + 1**

```
例如 ：1L的二进制
0000000000000000000000000000000000000000000000000000000000000001

反码
1111111111111111111111111111111111111111111111111111111111111110

补码 =反码 + 1 （-1L的二进制）
1111111111111111111111111111111111111111111111111111111111111111 //-1L的二进制表示

-1L << 5 ，向左移动5位，低位补0，超过64位的高位溢出部分舍弃
1111111111111111111111111111111111111111111111111111111111100000 //左移5位

-1L ^ (-1L << 5) ：异或( ^ ) 两个数相同，结果为0，不相同则为1
```

```
00000000000000000000000000000000000000000000000000000011111 //转为十进制就是31
```

就是16+8+4+2+1 = 31

也就是 2的5次方-1 = 31

该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，从0开始算。所以可以配置32台机器。

2、与( & )

```
// 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
//sequence == 0 ，就是1毫秒用完了4096个数
if (sequence == 0) {
// 阻塞到下一个毫秒,获得新的时间戳
timestamp = tilNextMillis(lastTimestamp);
}
 }
```

这里使用到了 与( & ) 位运算

两个数都为1，结果为1，否则为0

sequence 一毫秒内开始从0开始， sequenceMask为最大值4095。

这个方法里sequence == 0 为什么要等到下一毫秒来重置sequence的值

```
...00000000 00000000 00000000 00000001 // 1
& ...00000000 00000000 00001111 11111111 //4095
-----------------------------------------------------------------
...00000000 00000000 00000000 00000001 //1

...00000000 00000000 00000000 00000010 // 2
& ...00000000 00000000 00001111 11111111 //4095
-----------------------------------------------------------------
...00000000 00000000 00000000 00000010 // 2

 ...00000000 00000000 00001111 11111111 //4095
 & ...00000000 00000000 00001111 11111111 //4095
-----------------------------------------------------------------
 ...00000000 00000000 00001111 11111111 //4095

 ...00000000 00000000 00010000 00000000 //4096
 & ...00000000 00000000 00001111 11111111 //4095
-----------------------------------------------------------------
```

```
...00000000 00000000 00000000 00000000 //0
```

可以看出来到了4096之前，计算出来的结果都是等于本身，到了4096计算结果为0，所以sequence == 0 就是说从0开始，到4095，4096个数已经用完了。实际场景不够用可以调大位数。

3、或( | )

```
// 移位并通过或运算拼到一起组成64位的ID
return ((timestamp - startTime) << timestampLeftShift) // 时间戳左移22位
| (datacenterId << datacenterIdShift) //数据标识左移17位
| (workerId << workerIdShift) //机器id标识左移12位
| sequence;
```

这里使用到了 与( | )， 左移( < < ) 这两个位运算

与( | ) 两个数只要一个是1，结果为1，否则为0

问题：这是最后生成id的算法,为什么不同的部分要左移不同的位数呢。

模拟下数据

```
public static void main(String[] args) {
 //第一部分时间戳
 Long a = 1648742400000L;
 String aa = Long.toBinaryString(a);
 System.out.println("时间戳位数" + aa.length());
 while (aa.length() < 64) {
 aa = "0" + aa;
 }
 System.out.println("//时间戳-----------------------------------//");
  System.out.println(aa);


 //第二部分机房区分
 Long b = 5L;
 String bb = Long.toBinaryString(b);
 while (bb.length() < 64) {
 bb = "0" + bb;
 }
 System.out.println("//数据标识-----------------------------------//");
 System.out.println(bb);

 //机器区分
 Long c = 6L;
```

```
 String cc = Long.toBinaryString(c);
 while (cc.length() < 64) {
 cc = "0" + cc;
 }
 System.out.println("//机器标识----------------------------------//");
 System.out.println(cc);


 //第三部分递增数
 Long d = 1L;
 String dd = Long.toBinaryString(d);
 while (dd.length() < 64) {
 dd = "0" + dd;
 }
 System.out.println("//自增数----------------------------------//");
 System.out.println(dd);
}
```

输出结果：

```
时间戳位数41
//时间戳----------------------------------//
0000000000000000000000101111111111100000101101001000000000000000
//机房标识----------------------------------//
0000000000000000000000000000000000000000000000000000000000000101
//机器标识----------------------------------//
0000000000000000000000000000000000000000000000000000000000000110
//自增数----------------------------------//
0000000000000000000000000000000000000000000000000000000000000001
```

进行位移计算后

```
public static void main(String[] args) {
 //第一部分时间戳
 Long a = 1648742400000L;
 String aa = Long.toBinaryString(a<<22);
 while (aa.length() < 64) {
 aa = "0" + aa;
 }
 System.out.println("//位移22位后时间戳----------------------------------
-//");
 System.out.println(aa);


```

```
12  //第二部分数据区分
13  Long b = 5L;
14  String bb = Long.toBinaryString(b<<17);
15  while (bb.length() < 64) {
16  bb = "0" + bb;
17  }
18  System.out.println("//位移17位后数据标识----------------------------------
---//");
19  System.out.println(bb);
20
21  //机器区分
22  Long c = 6L;
23  String cc = Long.toBinaryString(c<<12);
24  while (cc.length() < 64) {
25  cc = "0" + cc;
26  }
27  System.out.println("//位移12位后机器标识----------------------------------
---//");
28  System.out.println(cc);
29
30
31  //第三部分递增数
32  Long d = 1L;
33  String dd = Long.toBinaryString(d);
34  while (dd.length() < 64) {
35  dd = "0" + dd;
36  }
37  System.out.println("//自增数（最后一个部分不用位移）----------------------
--------------//");
38  System.out.println(dd);
39 }
```

输出结果：

```
1  //位移22位后时间戳-----------------------------------//
2  0101111111111000001011010010000000000000000000000000000000000000
3  //位移17位后机房标识-----------------------------------//
4  0000000000000000000000000000000000000000000010100000000000000000
5  //位移12位后机器标识-----------------------------------//
6  0000000000000000000000000000000000000000000001100000000000000
7  //自增数（最后一个部分不用位移）-----------------------------------//
8  0000000000000000000000000000000000000000000000000000000000000001
```

最后使用位移后的数据进行 与（|）计算合并。两个数只要一个是1，结果为1，否则为0

```
0101111111111000001011010010000000000000000000000000000000000000 //时间戳
| 0000000000000000000000000000000000000000000010100000000000000000 //机房标识
| 0000000000000000000000000000000000000000000001100000000000000000 //机器标识
| 0000000000000000000000000000000000000000000000000000000000000001 //数据标识
```

最后可以看出，对应每个部分之间的数据就是互不影响，放在了各自对应的位数范围内，把每个部分的数据合并起来。

时间戳左移22位，因为后面有 5（数据标识）+5（机器标识）+12（自增数）=22位

数据标识左移17位，后面还需要 5（机器标识）+12（自增数）=17位

机器标识左移12位，最后留下的位数给自增数。

**总结**

优点：雪花算法提供了一个很好的设计思想，雪花算法生成的ID是趋势递增，不依赖数据库等第三方系统，生成ID的性能也是非常高的，而且可以根据自身业务特性分配bit位，非常灵活。

缺点：雪花算法强依赖机器时钟，如果机器上时钟回拨，会导致发号重复。如果恰巧回退前生成过一些ID，而时间回退后，生成的ID就有可能重复