链滴

雪花算法

大背景不讲，参考：https://segmentfault.com/a/1190000011282426#articleHeader5

小背景：我们的订单编号要求是16位，改造了一下雪花算法

```
*
* 参考Twitter Snowflake算法，按实际需求，做了部分修改，结构如下(每部分用-分开):
* 0000000000 - 10000000000000000000000000000000000000000 - 00 - 000 - 00000000
* 10位不使用，因为目的是为了最终生成16位整数，所以只使用后面的54bit
* 41位时间截(毫秒级)，存储时间截的差值（当前时间截 - 开始时间截)，41位的时间截，可以使用69
，且考虑到差值较小时，会生成不足16位的数字，因些需要选择一个合适的值
* 2位的集群ID，可以部署在4个集群
* 3位的节点ID，每个集群可以有8个节点
* 8位序列，毫秒内的计数，支持每个节点每毫秒产生256个ID序号
* 加起来刚好64位，为一个Long型
*/public class UniqueIdWorker {

  /**
* 起始时间，用于调整位数
* 这里取值 2012-12-22 00:00:00
* 以41位表示毫秒，此方案可以使用到 2082-08-28 15:47:35，订单编号从15开头，
*/  private final long baseTimestamp = 1356105600000L;

  /**
* 机器id所占的位数
*/
private final long workerIdBits = 3L;

  /**
* 集群id所占的位数
*/
private final long clusterIdBits = 2L;

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

  /**
* 支持的最大集群id
*/  private final long maxClusterId = -1L ^ (-1L << clusterIdBits);

  /**
* 序列在id中占的位数
*/
private final long sequenceBits = 8L;

  /**
* 机器ID向左移位数
*/
private final long workerIdShift = sequenceBits;

  /**
* 集群id向左移位数
*/
private final long clusterIdShift = sequenceBits + workerIdBits;
```

```
  /**
* 时间截向左移位数
*/
private final long timestampLeftShift = sequenceBits + workerIdBits + clusterIdBits;

  /**
* 生成序列的掩码
*/
private final long sequenceMask = -1L ^ (-1L << sequenceBits);

  /**
* 工作机器ID
*/  private long workerId;

  /**
* 集群ID
*/  private long clusterId;

  /**
* 毫秒内序列
*/
private long sequence = 0L;

  /**
* 上次生成ID的时间截
*/
private long lastTimestamp = -1L;

  /**
* 构造函数
*
* @param workerId
* @param clusterId
*/
public UniqueIdWorker(Long workerId, Long clusterId) {
      Preconditions.checkArgument(null != workerId && workerId > 0 && workerId < maxWor
erId, "Invalid workerId");
      Preconditions.checkArgument(null != clusterId && clusterId > 0 && clusterId < maxClust
rId, "Invalid clusterId");
      this.workerId = workerId;
      this.clusterId = clusterId;
  }

  /**
* 获得下一个ID
* * @return
*/
public synchronized long nextId() {
      long timestamp = timeGen();
      //系统时钟回退，抛出异常
if (timestamp < lastTimestamp) {
          throw new RuntimeException(String.format("Clock moved backwards. Failed to generat
 id for %d milliseconds", lastTimestamp - timestamp));
```

```
      }
      //同一毫秒内顺序递增
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);
          }
      }
      //时间戳改变重置为0
else {
          sequence = 0L;
      }
      lastTimestamp = timestamp;
      return ((timestamp - baseTimestamp) << timestampLeftShift)
              | (clusterId << clusterIdShift)
              | (workerId << workerIdShift)
              | sequence;
  }

  /**
* 阻塞到下一个毫秒的时间戳并返回
*
* @param lastTimestamp
* @return
*/
private long tilNextMillis(long lastTimestamp) {
      long timestamp = timeGen();
      while (timestamp <= lastTimestamp) {
          timestamp = timeGen();
      }
      return timestamp;
  }

  /**
* 返回当前毫秒时间戳
*
* @return
*/
private long timeGen() {
      return System.currentTimeMillis();
  }

  /**
* 根据订单ID反向解析内容
*
* @param id
* @return
*/
public String parseId(Long id) {
      if (null == id) {
          return "";
      }
```

```
    return String.format("sequence: %d, workerId: %d, clusterId: %d, timestamp: %d\n", ((id) &
~(-1L << sequenceBits))
              , ((id >> (workerIdShift)) & ~(-1L << (workerIdBits)))
              , ((id >> clusterIdShift) & ~(-1L << clusterIdBits))
              , ((id >> timestampLeftShift) + baseTimestamp));
  }

}
```

解释

41位时间戳能用几年?

```
@Test
public void test2() {
    String minTimeStampStr = "00000000000000000000000000000000000000000";
    long minTimeStamp = new BigInteger(minTimeStampStr, 2).longValue();
    String maxTimeStampStr = "11111111111111111111111111111111111111111";
    long maxTimeStamp = new BigInteger(maxTimeStampStr, 2).longValue();
    long oneYearMills = 1L * 1000 * 60 * 60 * 24 * 365;
    System.out.println((maxTimeStamp - minTimeStamp) / oneYearMills);
}
```

结果是69

前41位最小值

如果前41位太小，结果可能不满16位。

计算1000_0000_0000_0000L的前41位

```
@Test
  public void test1() {
        String str = Long.toBinaryString(
            1000_0000_0000_0000L);//00001110001101011111101010010011000110100_0000000
00000
//        String str = Long.toBinaryString(9999_9999_9999_9999L);//1000111000011011110010
1101111111000000111_1111111111111
  System.out.println(str);
        int needZero = 54 - str.length();
        str = StringUtils.repeat("0", needZero) + str;
        char[] chars = str.toCharArray();

        System.out.println("length :" + chars.length);
        for (int i = 0; i < chars.length; i++) {
            if (i != 0 && i % 41 == 0) {
                System.out.print("_");
            }
            System.out.print(chars[i]);
        }

    }
```

起始时间计算

用当前时间戳减去前41位最小值，得到的时间就是起始时间，如果需要开头从15或者20开始，也可自行计算。

```
@Test
  public void test3() {

        long curTimeStamp = System.currentTimeMillis();
        //也可以用下面的
//        long curTimeStamp1 = LocalDateTime.now().atZone(ZoneId.systemDefault()).toInstant(

//            .toEpochMilli();
  System.out.println(curTimeStamp);
//        System.out.println(curTimeStamp1);
 //差值最小为00001110001101011111101010010011000110100
  String diffTimeStampStr = "00001110001101011111101010010011000110100";
        long diffTimeStamp = new BigInteger(diffTimeStampStr, 2).longValue();
        long minTimeStamp = curTimeStamp - diffTimeStamp;
        LocalDateTime minDateTime = LocalDateTime
            .ofInstant(Instant.ofEpochMilli(minTimeStamp), ZoneId.systemDefault());
        System.out.println(minDateTime);//2015-02-15T17:59:14.079
  }
```

-1L ^ (-1L << workerIdBits)求最大机器id

```
//-1 的二进制原码1000 0001，反码 - 1111 1111
//-1 << 3也就是-8的二进制 1000 1000 反码- 1111 1000
// 1111 1111 ^ 1111 1000 =   0000 0111
```

原文链接：雪花算法