**并发生成重复订单号**

# 思路，生成订单号：时间+序号

|  |
| --- |
| **package** com.locks;  **import** java.text.SimpleDateFormat; **import** java.util.Date;  */\*\*  \** ***@author*** *iszengziqiang@163.com  \** ***@Version:*** *1.0  \** ***@desc*** *//****todo*** *\*/* **public class** OrderCodeGenerator {  *// 自增长序列* **private static int** *i* = 0;   *//生成订单号* **public** String getOrderCode() {  Date date = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd-HH-mm-ss-"**);  **return** sdf.format(date) + ++*i*;  }   **public static void** main(String[] args){  OrderCodeGenerator orderCodeGenerator = **new** OrderCodeGenerator();  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(orderCodeGenerator.getOrderCode());  }  } } |

# 多线程同时生成订单号，出现并发问题会生成相同的订单号

|  |
| --- |
| **package** com.locks;  **import** java.util.concurrent.CountDownLatch;  */\*\*  \** ***@author*** *iszengziqiang@163.com  \** ***@Version:*** *1.0  \** ***@desc*** *//****todo*** *\*/* **public class** OrderServiceImpl **implements** Runnable {   **private static** OrderCodeGenerator *ong* = **new** OrderCodeGenerator();   *//同时并发的线程数* **private static final int *NUM*** = 10;  *//按照线程数初始化倒计数器，倒计数器* **private static** CountDownLatch *cdl* = **new** CountDownLatch(***NUM***);   *// 创建订单接口* **public void** createOrder() {  String orderCode = **null**;  *//获取订单编号* orderCode = *ong*.getOrderCode();  *//业务代码* System.***out***.println(Thread.*currentThread*().getName() + **"====>"** + orderCode);   }   */\*\*  \* When an object implementing interface <code>Runnable</code> is used  \* to create a thread, starting the thread causes the object's  \* <code>run</code> method to be called in that separately executing  \* thread.  \* <p>  \* The general contract of the method <code>run</code> is that it may  \* take any action whatsoever.  \*  \** ***@see*** *Thread#run()  \*/* @Override  **public void** run() {  **try** {  *//等待其他线程初始化（只有线程数达到预定线程才会同时执行方法）  cdl*.await();  } **catch** (Exception e) {  e.printStackTrace();  }  *//创建订单* createOrder();  }   **public static void** main(String[] args) {  **for** (**int** i = 0; i < ***NUM***; i++) {  **new** Thread(**new** OrderServiceImpl()).start();  *cdl*.countDown();  }  } } |

|  |
| --- |
| 打印结果：  Thread-5====>2020-04-07-20-45-11-1  Thread-6====>2020-04-07-20-45-11-2  Thread-1====>2020-04-07-20-45-11-4  Thread-7====>2020-04-07-20-45-11-3  Thread-4====>2020-04-07-20-45-11-3  Thread-0====>2020-04-07-20-45-11-5  Thread-3====>2020-04-07-20-45-11-6  Thread-2====>2020-04-07-20-45-11-7  Thread-8====>2020-04-07-20-45-11-8  Thread-9====>2020-04-07-20-45-11-9  Process finished with exit code 0 |

可以看出线程4和线程7生成了一样的订单号，所以此方法存在问题。

优化一：

|  |
| --- |
| **加锁：**  **private static** Lock *lock* = **new** ReentrantLock(); |

方法如下：实用单体应用（分布式项目不可用）

|  |
| --- |
| **package** com.locks;  **import** java.util.concurrent.CountDownLatch; **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReentrantLock;  */\*\*  \** ***@author*** *iszengziqiang@163.com  \** ***@date*** *2020/4/7 11:31  \** ***@Version:*** *1.0  \** ***@desc*** *//****todo*** *\*/* **public class** OrderServiceImpl **implements** Runnable {   **private static** OrderCodeGenerator *ong* = **new** OrderCodeGenerator();   *//同时并发的线程数* **private static final int *NUM*** = 10;  *//按照线程数初始化倒计数器，倒计数器* **private static** CountDownLatch *cdl* = **new** CountDownLatch(***NUM***);   **private static** Lock *lock* = **new** ReentrantLock();   *// 创建订单接口* **public void** createOrder() {  String orderCode = **null**;   *lock*.lock();  **try** {  *//获取订单编号* orderCode = *ong*.getOrderCode();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  *lock*.unlock();  }   *//业务代码* System.***out***.println(Thread.*currentThread*().getName() + **"====>"** + orderCode);   }   */\*\*  \* When an object implementing interface <code>Runnable</code> is used  \* to create a thread, starting the thread causes the object's  \* <code>run</code> method to be called in that separately executing  \* thread.  \* <p>  \* The general contract of the method <code>run</code> is that it may  \* take any action whatsoever.  \*  \** ***@see*** *Thread#run()  \*/* @Override  **public void** run() {  **try** {  *//等待其他线程初始化  cdl*.await();  } **catch** (Exception e) {  e.printStackTrace();  }  *//创建订单* createOrder();  }   **public static void** main(String[] args) {  **for** (**int** i = 0; i < ***NUM***; i++) {  **new** Thread(**new** OrderServiceImpl()).start();  *cdl*.countDown();  }  } } |

# 优化二：（分布式锁）

<https://github.com/redisson/redisson/wiki/1.-%E6%A6%82%E8%BF%B0>

# 推特(Twitter)的Snowflake算法——用于生成唯一ID

<https://www.cnblogs.com/jakeylove3/p/8446798.html>

|  |
| --- |
| /\*\*  \* @author Jakeylove3  \* 2017/12/31  \*/  /\*\*  \* Twitter\_Snowflake  \* SnowFlake的结构如下(每部分用-分开):  \* 0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000  \* 1位标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0  \* 41位时间戳(毫秒级)，注意，41位时间戳不是存储当前时间的时间戳，而是存储时间戳的差值（当前时间戳 - 开始时间戳)  \* 得到的值），这里的的开始时间戳，一般是我们的id生成器开始使用的时间，由我们程序来指定的（如下面程序SnowflakeIdWorker类的startTime属性）。41位的时间戳，可以使用69年，年T = (1L << 41) / (1000L \* 60 \* 60 \* 24 \* 365) = 69  \* 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId  \* 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间戳)产生4096个ID序号  \* 加起来刚好64位，为一个Long型。  \*/  public class SnowflakeIdWorker {  /\*\* 开始时间戳 (2015-01-01) \*/  private final long twepoch = 1420041600000L;  /\*\* 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;  /\*\* 支持的最大机器id，结果是31 (这个移位算法可以很快的计算出几位二进制数所能表示的最大十进制数) \*/  private final long maxWorkerId = -1L ^ (-1L << workerIdBits);  /\*\* 支持的最大数据标识id，结果是31 \*/  private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);  /\*\* 序列在id中占的位数 \*/  private final long sequenceBits = 12L;  /\*\* 机器ID向左移12位 \*/  private final long workerIdShift = sequenceBits;  /\*\* 数据标识id向左移17位(12+5) \*/  private final long datacenterIdShift = sequenceBits + workerIdBits;  /\*\* 时间戳向左移22位(5+5+12) \*/  private final long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;  /\*\* 生成序列的掩码，这里为4095 (0b111111111111=0xfff=4095) \*/  private final long sequenceMask = -1L ^ (-1L << sequenceBits);  /\*\* 工作机器ID(0~31) \*/  private long workerId;  /\*\* 数据中心ID(0~31) \*/  private long datacenterId;  /\*\* 毫秒内序列(0~4095) \*/  private long sequence = 0L;  /\*\* 上次生成ID的时间戳 \*/  private long lastTimestamp = -1L;  //==============================Constructors=====================================  /\*\*  \* 构造函数  \* @param workerId 工作ID (0~31)  \* @param datacenterId 数据中心ID (0~31)  \*/  public SnowflakeIdWorker(long workerId, long datacenterId) {  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));  }  this.workerId = workerId;  this.datacenterId = datacenterId;  }  // ==============================Methods==========================================  /\*\*  \* 获得下一个ID (该方法是线程安全的)  \* @return SnowflakeId  \*/  public synchronized long nextId() {  long timestamp = timeGen();  //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  if (timestamp < lastTimestamp) {  throw new RuntimeException(  String.format("Clock moved backwards. Refusing to generate id for %d milliseconds", lastTimestamp - timestamp));  }  //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  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，毫秒内序列重置  else {  sequence = 0L;  }  //上次生成ID的时间戳  lastTimestamp = timestamp;  //移位并通过或运算拼到一起组成64位的ID  return ((timestamp - twepoch) << timestampLeftShift) //  | (datacenterId << datacenterIdShift) //  | (workerId << workerIdShift) //  | sequence;  }  /\*\*  \* 阻塞到下一个毫秒，直到获得新的时间戳  \* @param lastTimestamp 上次生成ID的时间戳  \* @return 当前时间戳  \*/  protected long tilNextMillis(long lastTimestamp) {  long timestamp = timeGen();  while (timestamp <= lastTimestamp) {  timestamp = timeGen();  }  return timestamp;  }  /\*\*  \* 返回以毫秒为单位的当前时间  \* @return 当前时间(毫秒)  \*/  protected long timeGen() {  return System.currentTimeMillis();  }  /\*\* 测试 \*/  public static void main(String[] args) {  System.out.println("开始："+System.currentTimeMillis());  SnowflakeIdWorker idWorker = new SnowflakeIdWorker(0, 0);  for (int i = 0; i < 50; i++) {  long id = idWorker.nextId();  System.out.println(id);  // System.out.println(Long.toBinaryString(id));  }  System.out.println("结束："+System.currentTimeMillis());  }  } |

