# 秒杀-全局唯一ID

经过前面的学习，我们已经掌握了如何利用Redis来去解决Session共享的问题，还有如何利用Redis去添加查询业务的缓存并且去应对缓存使用时出现的各种各样的问题，接下来继续学习如何使用Redis解决秒杀业务。可以说聊电商一定离不开秒杀，而Redis在整个秒杀业务中扮演了非常重要的角色！

## 全局唯一ID

需求：每个店铺都可以发布优惠券，当用户抢购的时候，就会生产订单并保存到数据库的订单表tb\_voucher\_order中，而订单表如果使用数据库自增ID就存在一些问题！

### 数据库自增ID存在的问题

#### 1：id的规律性太明显了！

比如昨天我下单的时候订单ID看到是1，今天我再下单看到订单的ID是99，这不就暴露了信息给用户了。

#### 2：受单表数据量的限制！

单张表不能保存的数据量是有限的，如果无法保存就需要分到多张表，每张表的ID都是自增的话，自增的ID是会出现重复的，而订单ID是不应该重复的！

所以不能使用数据库的自增ID策略，就要用到下面讲的全局ID生成器！

### 全局ID生成器介绍：

全局ID生成器，是一种在分布式系统下用来生产全局唯一ID的工具。就是说在同一个业务下，不管你这个分布式系统将来有多少个服务、、多少个节点，这个业务下将来分成了多少张表，最终只要你用全局ID生成器，得到的ID一定是当前业务中唯一的，不会出现冲突。当然了仅限于同一业务，你不同的业务即便冲突了也没关系。

全局ID生成器一般满足下图特性！

#### 唯一性

#### 高性能

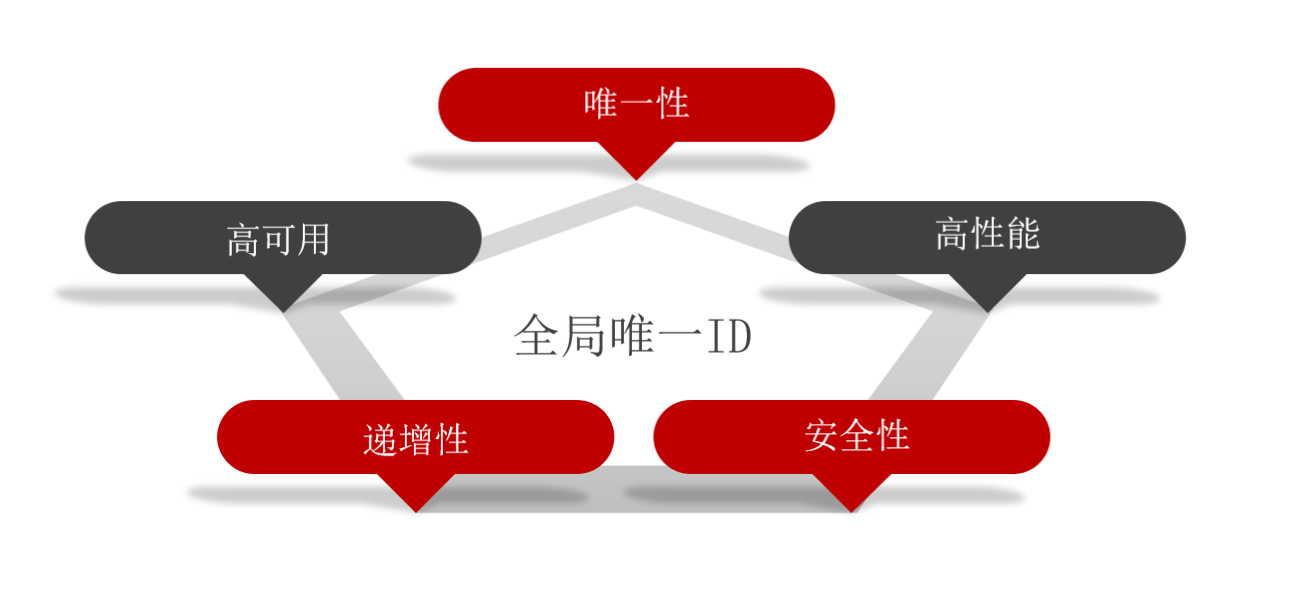
#### 高可用

#### 递增性：

全局ID生成器生成的ID要有单调递增的特性，即不断变大，因为我们的这个ID是用来替代数据库自增ID的，虽然不一定像数据库一样1234挨着增长，但是一定要确保整体逐渐变大的特性，因为这样能够有利于数据库创建索引，提高插入、查询时的速度！

#### 安全性：

ID的规律性不能太明显，让别人能猜测到你其他的订单id是不行的。



## Redis实现全局唯一ID

很明显，可以使用Redis来满足以上五个特性，得到带一种全局ID的生成方案！就是利用Redis中的String数据结构是有一个自增特性的，即incre命令。

### 唯一性

唯一性绝对可以保证，因为Redis是独立于数据库之外的，不管你数据库有几张表，或者说你有几个不同的数据库，但是我们的Redis只有一个，这个时候当所有的人都来访问Redis，自增一定是唯一的；

### 高可用

高可用更不用提，Redis的集群方案、主从方案、哨兵方案，这些方案确保高可用；

### 高性能

高性能更不用提，Redis就是以高性能著称的，比数据库快多了；

### 递增性：

incre命令就是自增

### 安全性：

如果说你采用Redis递增方案和数据库一样，都是12345这样，那就不存在安全性，因为太容易猜测出规律，所以说使用Redis的incre实现全局唯一ID的时候，不能直接把Redis的自增数值拿来当id，我们要拼接一定其他信息，让规律性不那么明显！

## Redis全局唯一ID组成：

为了增加ID的安全性，我们可以不直接使用Redis自增的数值，而是拼接一些其他信息：

为了提高数据库的性能，我们这个ID会采用数值类型，说直白点就是Java中的Long类型，然后直接插入数据库，因为数值类型在数据库里占用空间更小，建立索引更方便更快。

因为采用Long类型，所以八个字节，64位。

### 符号位：

最高位是符号位，永远是0，代表ID永远是正数；

### 时间戳：

符号位后31位是时间戳，时间戳就是用来增加全局ID的复杂性的，不是单纯的Redis自增了，但是一个时间戳。时间戳为什么是31位的，因为时间戳我们将来会以秒为单位，也就是说我们会定义一个初始时间，比如说从2024年1月1号零时起。然后时间戳就是计算当前下单时间离初始时间多少秒。31位可以表示21亿多秒，可以支撑69年，远远足够了。

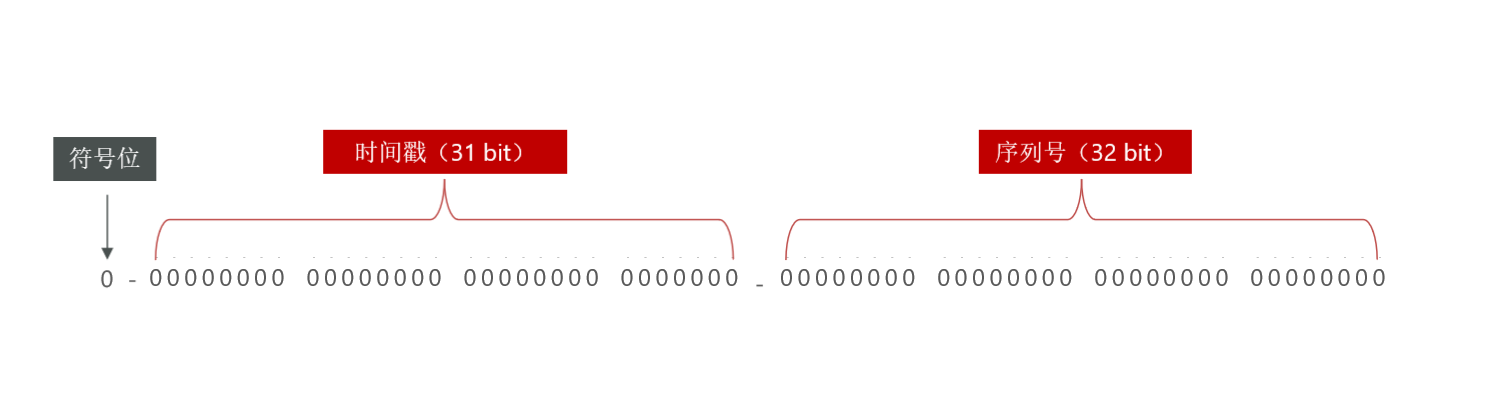
### 序列号：

最后32位就是序列号，是Redis自增的值。你可能会说很多用户同一秒内下了多个订单，你的时间戳不久重复了吗？没关系，时间戳可以冲突，时间戳冲突的情况下，序列号是不冲突的，因为理论上支持一秒钟内2^32个订单，接近43亿个订购单了，完全够用！强如淘宝在双十一每秒最多也就几十万订单，平常顶多几千订单！

唯一性：一套下来整体上一定是唯一的：时间不同，时间戳就不同，ID整体肯定不同；就算时间戳一样，序列号一定不一样，ID整体肯定不同！

递增性：ID整体肯定是单调递增的，时间戳是越来越大单调递增，序列号也是越来越大单调递增的。

安全性：因为我们的id是时间戳和序列号拼接在一起的，不是简单的自增



### 总结：ID的组成部分：

符号位：1bit，永远为0

时间戳：31bit，以秒为单位，可以使用69年

序列号：32bit，秒内的计数器，支持每秒产生2^32个不同ID

## Redis全局唯一ID编码实现

Redis单个Key的自增长对应的数值是有上限的，上限是2^64，虽然很大，但归根结底也是有上限，万一超过了怎么办？

而且我们的全局唯一ID生成策略里面，真正用来记录序列号的只有32位。Redis的key自增长对应的数值上限是64位，key自增的数值超过64位很难，超过32还是有一定可能的哦。将来如果永远是同一个key，key自增的数值超过32位，序列号这部分32位就存不下key自增的数值了。

所以说哪怕是同一个业务也不能使用同一个Key！

解决方案：在业务前缀后边拼上时间戳，比如说：inc:order:20231218，那么代表20231218这一天下的订单的id就会以这个inc:order:20231218作为key去自增。也就是说这个key自增的上限就是这一天的下的订单量，一天的订单量怎么可不可能超过2^32！

这样做还有好处，比如我想看某天的订单量直接看这个key的值就行了，即还有统计的好处。

## 总结：

### 全局唯一ID生成策略：

#### 1：UUID

利用JDK自带的UUID工具类就能生成，这种生成策略生成的是16进制的数值。16进制所以返回的是字符串结构，并且不支持单调递增的特性，因此虽然可以做唯一ID，但是不够友好，没有满足自增性。

#### Redis自增

我们采用的方案，这种方案相对来说各种特性都能满足，id的长度也不大，Long类型64位。数字类型在数据库中存储比较小，比较友好。

#### snowflake算法

雪花算法，也是采用的是long类型的64位的数字，性能比Redis好，自增方案采用的是当前机器的自增，所以需要维护一个机器ID。对于时钟以来比较高，如果时间不准确可能出现异常问题。

#### 数据库自增

这里的自增不是在新增订单到订单表中，对订单id设置成自增，而是单独整一张表，这张表专门用来做自增，这样一来不管你的订单表是十张表还是八张表，这些表的id从这张专门自增的表中来。等于n张表用的是同一张表的自增id。可以看作是Redis方案的数据库版。

这种方案性能当然比Redis方案差，所以企业使用数据库自增的时候往往采取一些方案，比如批量地去获取id在内存中缓存起来，然后再去使用，一定程度上提高性能。

### Redis自增ID策略：

#### 每天一个key，方便统计订单量

#### ID构造是 时间戳 + 计数器