UniqueID

**生成算法**

www.bzMe.work

参考：

**概述**

唯一ID(UniqueID,UID)是系统设计常遇到的问题。生成唯一ID的方法有很多，例如常用的方式是利用数据库产生唯一ID，其优点的是无论是单机还是分布式系统其生成的UID是全局不重复的，且UID是有序的，缺点是UID的生成依赖数据库。

下面主要讨论的是在一个分布式系统中，如何不依赖中心数据库生成有序的全局唯一的UID。

**一、UUID**

UUID (Universally Unique Identifier, 通用唯一识别码)，这是一个软件建构的标准，也是被开源软件基金会(Open Software Foundation, OSF)的组织应用在分布式计算环境(Distributed Computing Environment, DCE)领域的一部分。

UUID 的目的，是让分布式系统中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识资讯的指定。如此一来，每个人都可以建立不与其它人冲突的UUID。在这样的情况下，就不需考虑数据库建立时的名称重复问题。目前最广泛应用的UUID，即是微软的 Microsoft's Globally Unique Identifiers(GUIDs)

GUID（Globally Unique Identifier，全局唯一标识符）也称作UUID(Universally Unique IDentifier)。GUID是一种由算法生成的二进制长度为128位的数字标识符。GUID主要用于在拥有多个节点、多台计算机的网络或系统中。在理想情况下，任何计算机和计算机集群都不会生成两个相同的GUID。GUID的总数达到了2^128（3.4×10^38）个，所以随机生成两个相同GUID的可能性非常小，但并不为0。GUID一词有时也专指微软对UUID标准的实现。

UUID是指在一台机器上生成的UID，它保证对在同一时空中的所有机器都是唯一的。按照[开放软件基金会](http://baike.baidu.com/view/770031.htm)(OSF)制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片ID码和许多可能的数字，UUID由以下几部分的组合：

(1) 当前日期和时间，UUID的第一个部分与时间有关，如果你在生成一个UUID之后，过几秒又生成一个UUID，则第一个部分不同，其余相同。

(2) 时钟序列。

(3) 全局唯一的IEEE机器识别号，如果有网卡，从网卡MAC地址获得，没有网卡以其他方式获得。

UUID的唯一缺陷在于生成的结果是比较长的字符串，且是无序的。

UUID的格式为：xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx(8-4-4-16)，

标准的UUID格式为：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx(8-4-4-4-12)。

其中每个x是0-9或a-f范围内的一个十六进制数。

GUID则是采用标准的UUID格式：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx(8-4-4-4-12)

其中每个 x 是 0-9 或 a-f 范围内的一个十六进制数。例如：6F9619FF-8B86-D011-B42D-00C04FC964FF 即为有效的GUID值。

在软件设计中，为了解决UUID无序的问题，NHibernate开源库在其主键生成中提供了Comb (combined guid/timestamp)算法。保留GUID的10个字节，用另6个字节表示GUID生成的时间(DateTime)。C#源码如下：

|  |
| --- |
| // https://github.com/nhibernate/nhibernate-core/blob/master/src/NHibernate/Id/GuidCombGenerator.cs  // using a strategy suggested Jimmy Nilsson's(使用Jimmy Nilsson建议的策略)  // http://www.informit.com/articles/article.asp?p=25862  // http://www.informit.com  public class GuidCombGenerator  {  private static readonly long BaseDateTicks = new DateTime(1900, 1, 1).Ticks;  /// <summary>  /// Generate a new Guid using the comb algorithm.  /// 使用comb算法生成一个新的Guid。  /// </summary>  public Guid GenerateComb()  {  // 先生成GUID  byte[] guidArray = Guid.NewGuid().ToByteArray();  DateTime now = DateTime.UtcNow;  // Get the days and milliseconds which will be used to build the byte string  // 获取用于生成字节字符串的天数和毫秒数  TimeSpan days = new TimeSpan(now.Ticks - BaseDateTicks);  TimeSpan msecs = now.TimeOfDay;  // Convert to a byte array  // Note that SQL Server is accurate to 1/300th of a millisecond so we divide by 3.333333  // 转换为字节数组  // 注意，SQL Server精确到1/300毫秒，所以我们除以3.333333  byte[] daysArray = BitConverter.GetBytes(days.Days);  byte[] msecsArray = BitConverter.GetBytes((long)(msecs.TotalMilliseconds / 3.333333));  // Reverse the bytes to match SQL Servers ordering  // 反转字节以匹配SQL服务器顺序  Array.Reverse(daysArray);  Array.Reverse(msecsArray);  // Copy the bytes into the guid  // 将字节复制到GUID中  Array.Copy(daysArray, daysArray.Length - 2, guidArray, guidArray.Length - 6, 2);  Array.Copy(msecsArray, msecsArray.Length - 4, guidArray, guidArray.Length - 4, 4);  // 返回有序的GUID  return new Guid(guidArray);  }  } |

**二、Snowflake**

参考：https://blog.csdn.net/bjweimengshu/article/details/80162731

Snowflake是Twitter开源的分布式ID生成算法，结果是一个long型的ID。Snowflake由4部分组成：

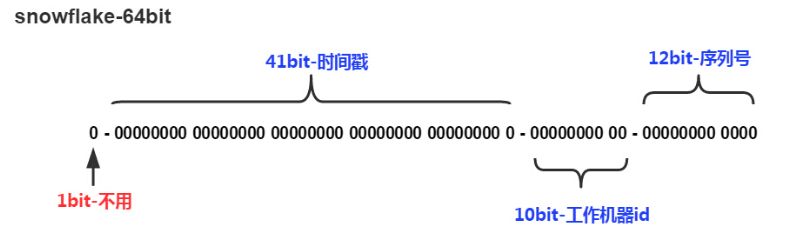
符号位：占用1bit，其值始终是0，没有实际作用。

时间戳：占用41bit，精确到毫秒，总共可以容纳约70年的时间。

机器ID：占用10bit，其中高位5bit是数据中心ID(DataCenterID)；低位5bit是工作节点ID(WorkerID)(即机器ID(MachineID))，最多可以容纳1024个节点。

序列号: 占用12bit，这个值在同一毫秒同一节点上从0开始不断累加，最多可以累加到4095(即每个节点每毫秒可以产生4096个序列号))。

4个部分总共64位用long(System.Int64)型来表示：



由上可知，SnowFlake算法在同一毫秒内最多可以生成：

1024(机器ID) \* 4096(序列号) = 4194304个全局唯一ID。

C#源码如下：

<https://github.com/Coldairarrow/IdHelper>

<https://github.com/mschuler/UniqueIdGenerator>

<https://www.cnblogs.com/wangquanyi/p/11328943.html>

<https://github.com/twitter-archive/snowflake/releases/tag/snowflake-2010>

<https://www.cnblogs.com/cider/p/11776088.html>

|  |
| --- |
| /// <summary>  /// From: https://github.com/twitter/snowflake  /// An object that generates IDs.  /// This is broken into a separate class in case  /// we ever want to support multiple worker threads  /// per process  /// </summary>  public class IdWorker  {  private long workerId;  private long datacenterId;  private long sequence = 0L;  private static long twepoch = 1288834974657L;  private static long workerIdBits = 5L;  private static long datacenterIdBits = 5L;  private static long maxWorkerId = -1L ^ (-1L << (int)workerIdBits);  private static long maxDatacenterId = -1L ^ (-1L << (int)datacenterIdBits);  private static long sequenceBits = 12L;  private long workerIdShift = sequenceBits;  private long datacenterIdShift = sequenceBits + workerIdBits;  private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;  private long sequenceMask = -1L ^ (-1L << (int)sequenceBits);  private long lastTimestamp = -1L;  private static object syncRoot = new object();  public IdWorker(long workerId, long datacenterId)  {  // sanity check for workerId  if (workerId > maxWorkerId || workerId < 0)  {  throw new ArgumentException(string.Format("worker Id can't be greater than %d or less than 0", maxWorkerId));  }  if (datacenterId > maxDatacenterId || datacenterId < 0)  {  throw new ArgumentException(string.Format("datacenter Id can't be greater than %d or less than 0", maxDatacenterId));  }  this.workerId = workerId;  this.datacenterId = datacenterId;  }  public long nextId()  {  lock (syncRoot)  {  long timestamp = timeGen();  if (timestamp < lastTimestamp)  {  throw new ApplicationException(string.Format("Clock moved backwards. Refusing to generate id for %d milliseconds", lastTimestamp - timestamp));  }  if (lastTimestamp == timestamp)  {  sequence = (sequence + 1) & sequenceMask;  if (sequence == 0)  {  timestamp = tilNextMillis(lastTimestamp);  }  }  else  {  sequence = 0L;  }  lastTimestamp = timestamp;  return ((timestamp - twepoch) << (int)timestampLeftShift) | (datacenterId << (int)datacenterIdShift) | (workerId << (int)workerIdShift) | sequence;  }  }  protected long tilNextMillis(long lastTimestamp)  {  long timestamp = timeGen();  while (timestamp <= lastTimestamp)  {  timestamp = timeGen();  }  return timestamp;  }  protected long timeGen()  {  return (long)(DateTime.UtcNow - new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc)).TotalMilliseconds;  }  } |

SnowFlake算法的优点：

1.生成ID时不依赖数据库，完全在内存生成，高性能高可用。

2.ID有序呈趋势递增，后续插入索引树的时候性能较好。

SnowFlake算法的缺点：

依赖于系统时钟的一致性，如果某台机器的系统时钟回拨，有可能造成ID冲突，或者ID乱序。

问题：

分析Snowflake算法，会发现时间戳部分用41bit来表达，总共能容纳的数值是:2^41-1=2199023255551，精确到毫秒，总共可以容纳约70年的时间：

年数=2199023255551/(365 \* 24 \* 3600 \* 1000L) = 69年

显然为了能容纳70年内时间，在生成时间戳的时候，要选择一个有效的参照时间，在上例中时间戳以1970-01-01 00：00：00为参照，需要进行修正。

为了设计Snowflake算法，需要明白纳秒和秒的关系，如下所示：

|  |
| --- |
| 1秒 = 1000毫秒，1毫秒 = 1000微妙，1微秒 = 1000纳秒。因此：  1秒 = 1000000000纳秒，1纳秒 = 0.000000001秒  1毫秒 = 1000000纳秒，1纳秒 = 0.000001毫秒  DateTime.Ticks：表示0001年1月1日午夜12:00:00以来所经历的100纳秒数。因此：  1秒 = 1000000000/100 = 10000000纳秒，1纳秒 = 0.0000001秒  1毫秒 = 1000000/100 = 10000纳秒，1纳秒 = 0.0001毫秒 |

现在我们将系统能表达的最大时间：9999-12-31 23：59：59转换成毫秒数：

|  |
| --- |
| var ticks = new DateTime(9999, 12, 31, 23, 59, 59, DateTimeKind.Utc).Ticks; // 纳秒数  var mills = (long)((double)ticks \* 0.0001d); // 转换成毫秒数  var bin1 = System.Convert.ToString(ticks, 2); // 将纳秒数转换成二进制  var bin2 = System.Convert.ToString(mills, 2); // 将毫秒数转换成二进制  // 输出：  ticks = 3155378975990000000  mills = 315537897599000  bin1 = 10101111001010001010000111010111110011100111101010100110000000 (62位)  bin2 = 00000000000001000111101111101011100100010011001011000000011000 (49位) |

这个例子告诉我们，如果想要让UID”有意义”，用纳秒数来表示显然是不行的，因为其64位的空间已经被其占完了，必须将其降级成毫秒数，且还要缩短年限，例如Snowflake算法采用70年。许多公司的UID算法都是在遵循SnowFlake算法的基础上进行的修改。