# 分布式系统唯一ID生成方案汇总 - nick hao - 博客园



## 分布式系统唯一ID生成方案汇总

系统唯一ID是我们在设计一个系统的时候常常会遇见的问题，也常常为这个问题而纠结。生成ID的方法有很多，适应不同的场景、需求以及性能要求。所以有些比较复杂的系统会有多个ID生成的策略。下面就介绍一些常见的ID生成策略。

**1. 数据库自增长序列或字段**

最常见的方式。利用数据库，全数据库唯一。

优点：

1）简单，代码方便，性能可以接受。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）不同数据库语法和实现不同，数据库迁移的时候或多数据库版本支持的时候需要处理。

2）在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。

3）在性能达不到要求的情况下，比较难于扩展。（不适用于海量高并发）

4）如果遇见多个系统需要合并或者涉及到数据迁移会相当痛苦。

5）分表分库的时候会有麻烦。

6）并非一定连续，类似MySQL，当生成新ID的事务回滚，那么后续的事务也不会再用这个ID了。这个在性能和连续性的折中。如果为了保证连续，必须要在事务结束后才能生成ID，那性能就会出现问题。

7）在分布式数据库中，如果采用了自增主键的话，有可能会带来尾部热点。分布式数据库常常使用range的分区方式，在大量新增记录的时候，IO会集中在一个分区上，造成热点数据。

优化方案：

1）针对主库单点，如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。

**2. UUID**

常见的方式。可以利用数据库也可以利用程序生成，一般来说全球唯一。UUID是由32个的16进制数字组成，所以每个UUID的长度是128位（16^32 = 2^128）。UUID作为一种广泛使用标准，有多个实现版本，影响它的因素包括时间、网卡MAC地址、自定义Namesapce等等。

优点：

1）简单，代码方便。

2）生成ID性能非常好，基本不会有性能问题。

3）全球唯一，在遇见数据迁移，系统数据合并，或者数据库变更等情况下，可以从容应对。

缺点：

1）没有排序，无法保证趋势递增。

2）UUID往往是使用字符串存储，查询的效率比较低。

3）存储空间比较大，如果是海量数据库，就需要考虑存储量的问题。

4）传输数据量大

5）不可读。

**3. UUID的变种**

1）为了解决UUID不可读，可以使用UUID to Int64的方法。及

```
/// <summary>
/// 根据GUID获取唯一数字序列
/// </summary>
public static long GuidToInt64()
{
    byte[] bytes = Guid.NewGuid().ToByteArray();
    return BitConverter.ToInt64(bytes, 0);
}
```

```
1
```

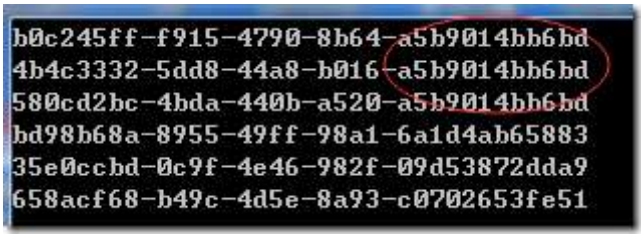
2）为了解决UUID无序的问题，NHibernate在其主键生成方式中提供了Comb算法（combined guid/timestamp）。保留GUID的10个字节，用另6个字节表示GUID生成的时间（DateTime）。

```
/// <summary>
/// Generate a new <see cref="Guid"/> using the comb algorithm.
/// </summary>
private Guid GenerateComb()
{
    byte[] guidArray = Guid.NewGuid().ToByteArray();

    DateTime baseDate = new DateTime(1900, 1, 1);
    DateTime now = DateTime.Now;

    // Get the days and milliseconds which will be used to build
    //the byte string
    TimeSpan days = new TimeSpan(now.Ticks - baseDate.Ticks);
    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
    byte[] daysArray = BitConverter.GetBytes(days.Days);
    byte[] msecsArray = BitConverter.GetBytes((long)
      (msecs.TotalMilliseconds / 3.333333));

    // Reverse the bytes to match SQL Servers ordering
    Array.Reverse(daysArray);
    Array.Reverse(msecsArray);

    // Copy the bytes into the guid
    Array.Copy(daysArray, daysArray.Length - 2, guidArray,
      guidArray.Length - 6, 2);
    Array.Copy(msecsArray, msecsArray.Length - 4, guidArray,
```

```
        guidArray.Length - 4, 4);

    return new Guid(guidArray);
}
```

```
1
```

用上面的算法测试一下，得到如下的结果：作为比较，前面3个是使用COMB算法得出的结果，最后12个字符串是时间序（统一毫秒生成的3个UUID），过段时间如果再次生成，则12个字符串会比图示的要大。后面3个是直接生成的GUID。




如果想把时间序放在前面，可以生成后改变12个字符串的位置，也可以修改算法类的最后两个Array.Copy。

**4. Redis生成ID**

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作 INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：

A：1,6,11,16,21

B：2,7,12,17,22

C：3,8,13,18,23

D：4,9,14,19,24

E：5,10,15,20,25

这个，随便负载到哪个机确定好，未来很难做修改。但是3-5台服务器基本能够满足器上，都可以获得不同的ID。但是步长和初始值一定需要事先需要了。使用Redis集群也可以方式单点故障的问题。

另外，比较适合使用Redis来生成每天从0开始的流水号。比如订单号=日期+当日自增长号。可以每天在Redis中生成一个Key，使用INCR进行累加。

优点：

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。

2）需要编码和配置的工作量比较大。

**5. Twitter的snowflake算法**

snowflake是Twitter开源的分布式ID生成算法，结果是一个long型的ID。其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID），最后还有一个符号位，永远是0。具体实现的代码可以参看https://github.com/twitter/snowflake。雪花算法支持的TPS可以达到419万左右（2^22*1000）。

雪花算法在工程实现上有单机版本和分布式版本。单机版本如下，分布式版本可以参看美团leaf算法：https://github.com/Meituan-Dianping/Leaf

C#代码如下：

```
/// <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;
        }
    }
```

测试代码如下：

```
private static void TestIdWorker()
        {
            HashSet<long> set = new HashSet<long>();
            IdWorker idWorker1 = new IdWorker(0, 0);
            IdWorker idWorker2 = new IdWorker(1, 0);
            Thread t1 = new Thread(() => DoTestIdWoker(idWorker1, set));
            Thread t2 = new Thread(() => DoTestIdWoker(idWorker2, set));
            t1.IsBackground = true;
```

```
            t2.IsBackground = true;

            t1.Start();
            t2.Start();
            try
            {
                Thread.Sleep(30000);
                t1.Abort();
                t2.Abort();
            }
            catch (Exception e)
            {
            }

            Console.WriteLine("done");
        }

        private static void DoTestIdWoker(IdWorker idWorker, HashSet<long> set)
        {
            while (true)
            {
                long id = idWorker.nextId();
                if (!set.Add(id))
                {
                    Console.WriteLine("duplicate:" + id);
                }

                Thread.Sleep(1);
            }
        }
```

snowflake算法可以根据自身项目的需要进行一定的修改。比如估算未来的数据中心个数，每个数据中心的机器数以及统一毫秒可以能的并发数来调整在算法中所需要的bit数。

优点:

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）ID按照时间在单机上是递增的。

缺点:

1）在单机上是递增的，但是由于涉及到分布式环境，每台机器上的时钟不可能完全同步，在算法上要解决时间回拨的问题。

## 6. 利用zookeeper生成唯一ID

1 zookeeper主要通过其znode数据版本来生成序列号，可以生成32位和64位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

1 很少会使用zookeeper来生成唯一ID。主要是由于需要依赖zookeeper，并且是多步调用API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不甚理想。

1

1 7. MongoDB的ObjectId

MongoDB的ObjectId和snowflake算法类似。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求。使其在分片环境中要容易生成得多。

其格式如下：


前4 个字节是从标准纪元开始的时间戳，单位为秒。时间戳，与随后的5 个字节组合起来，提供了秒级别的唯一性。由于时间戳在前，这意味着ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，如将其作为索引提高效率。这4 个字节也隐含了文档创建的时间。绝大多数客户端类库都会公开一个方法从ObjectId 获取这个信息。
接下来的3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的ObjectId，不产生冲突。
为了确保在同一台机器上并发的多个进程产生的ObjectId 是唯一的，接下来的两字节来自产生ObjectId 的进程标识符（PID）。
前9 字节保证了同一秒钟不同机器不同进程产生的ObjectId 是唯一的。后3 字节就是一个自动增加的计数器，确保相同进程同一秒产生的ObjectId 也是不一样的。同一秒钟最多允许每个进程拥有2563（16 777 216）个不同的ObjectId。

实现的源码可以到MongoDB官方网站下载。

## 8. TiDB的主键

TiDB默认是支持自增主键的，对未声明主键的表，会提供了一个隐式主键_tidb_rowid，因为这个主键大体上是单调递增的，所以也会出现我们前面说的“尾部热点”问题。

TiDB也提供了UUID函数，而且在4.0版本中还提供了另一种解决方案AutoRandom。TiDB 模仿MySQL的 AutoIncrement，提供了AutoRandom关键字用于生成一个随机ID填充指定列。

分类: 分布式系统

标签: 唯一ID生成, 分布式系统



nick (尼克) hao
关注 - 0
粉丝 - 195
+加关注