



下载APP



24 | UUID：如何高效生成全局的唯一ID？

2022-02-15 黄清昊

《算法实战高手课》

课程介绍 >



讲述：代录（老师近期生病）

时长 13:47 大小 12.64M



你好，我是微扰君。

今天我们来聊一聊在生产环境中非常常用的一个算法——全局唯一 ID 生成算法，也就是我们通常说的 UUID。

就和我们在社会中都有自己的身份证号作为自己的唯一标示一样，在互联网的应用中，很多时候，我们需要能生成一个全局唯一的 ID，去区别不同业务场景下的不同数据，比如消息 ID、用户 ID、微博内容 ID 等等。



因为我们往往需要通过这个 ID 去索引某个业务数据，所以一定要保证生成的 ID 在全局范围内是唯一的，这也是 identifier 的本意，在部分情况下，冲突概率很小可能也是可以接受的。另外，这个 ID 通常需要按照某种规则有序排列，最常用的就是基于时间进行排序。

所以全局唯一 ID 的两个核心需求就是：

1. 全局唯一性
2. 粗略有序性

那业界是如何生成满足这两大需求的 ID，又有哪些方案呢？我们开始今天的学习。

单体环境

在单体的应用中，保证 ID 的全局唯一，其实不是一个很大的问题，我们只需要提供一个在内存中的计数器，就可以完成对 ID 的颁发。

当然这样的 ID 可能会带有明确的含义，并被暴露出去了，比如在票务系统中，如果这样设计，我们能根据电子票 ID 判断出自己买的是第几张票。这对安全性要求更高的业务来说，是不可接受的，但通过一些简单的加密算法混淆，我们就能解决这个问题。

总的来说，单节点的应用，因为所有产生新业务数据，而需要产生新 ID 的地方，都是同一个地方，复杂性是很可控的。

分布式环境

但在现在的分布式环境下，每一个简单的问题都变得更复杂了一些，我们来举一个具体的例子。

假设，现在有一个票务系统，每次出票请求的产生，都会产生一个对应的电子票 ID，毫无疑问这个 ID 需要是全局唯一的，否则会出现多个同学的票无法区分的情况。那假设我们的出票服务 TPS 比较高，**为了同时让多台服务器都可以颁发不重复的 ID，我们自然需要一种机制进行多台服务器之间的协调或者分配。**

这个问题其实历史已久，解决方法也已经有很多了，我们一起来看看主流的解决方案是如何考虑的。

引入单点 ID 生成器

先来看一个最简单的，也非常容易 DIY 的思路——单点 ID 生成器。

通过这个方案，我们可以快速了解这个问题的解决思路，在接下来学习的过程中，你也可以边看边思考，对于这个全局唯一 ID 的生成，你有什么更好的改进主意。

我们在前面说了，单机里生成 ID 不是一个问题，**在多节点中，我们仍然可以尝试自己手动打造一个单点的 ID 生成器，通常可以是一个独立部署的服务**，这样的服务，我们一般也称为 ID generate service。

也就是说，所有其他需要生成 ID 的服务，在需要生成 ID 的时候都不自己生成，而是全部访问这个单点的服务。因为单点的服务只有一台机器，我们很容易通过本地时钟和计数器来保证 ID 的唯一性和有序性。

当然这里要重点注意的是，我们必须能应对时钟回拨，或者服务器异常重启之后计数器不会重复的问题。

如何解决

首先看第一个问题：时钟为什么会回拨呢？

如果你了解计算机如何计时的话就知道，计算机底层的计时主要依靠石英钟，它本身是有一定误差，所以计算机会定期地通过 NTP 服务，来同步更加接近真实时间的时间（仍然有一定的误差），这个时候就可能会产生时钟的一些跳跃。这里我们就不展开讲了，感兴趣的话。你可以自己去搜索一下 NTP 协议了解。

真正影响更大的问题其实是第二个，**如果计数器只是在内存中保存，一旦发生机器故障或者断电等情况，我们就无法知道之前的 ID 生成到什么位置了，怎么办？**

我们需要想办法有一定的持久化机制，也需要有一定的容灾备份的机制，要考虑的问题还是不少的。比如，对于单点服务挂了的情况下，首先想到可以用之前讲 Raft 和 MapReduce 的时候也提到过类似的提供一个备用服务的方案，来提高整个服务的高可用性，但这样，备用服务和主服务之间又如何同步状态，又成了新的问题，所以我们往往需要引入数据库等外部组件来解决。

哪怕解决了这两大问题，就这个设计本身来说，单点服务的一大限制是**性能不佳，如果每个请求都需要将状态持久化一下，并发量很容易遇到瓶颈。**

所以这种方案在实际生产中并不常用，具体实现就留给你做课后的思考题，你可以想想，不借助任何外部组件，自己如何独立实现一个单点的 ID 生成器服务。

基于数据库实现 ID 生成器

好我们继续想，既然直接自己写还是有许多问题需要考虑，那能不能利用现有的组件来实现呢？

我首先想到的方案就是数据库，还记得数据库中的主键吗，我们往往可以把主键设置成 `auto_increment`，这样在往数据库里插入一个元素的时候，就不需要我们提供 ID，而是数据库自动给我们生成一个呢？而且有了 `auto_increment`，我们也自然能保证字段的有序性。

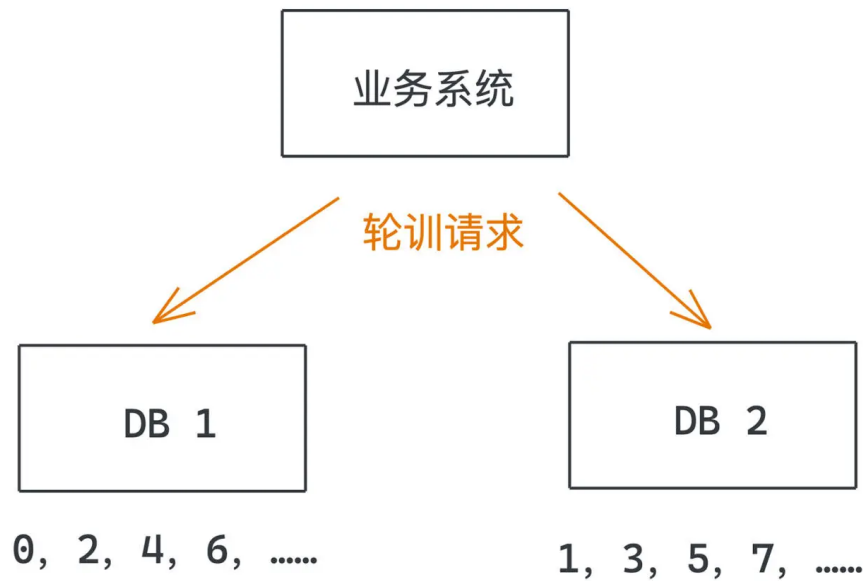
其实这正是天然的全局 ID 生成器。利用了外部组件自身的能力，我们基于数据库自增 ID 直接实现的 ID generate 非常简单，既可以保证唯一性，也可以保证有序性，ID 的步长也是可调的；而且数据库本身有非常好的可用性，能解决了我们对服务可靠性的顾虑。

但是同样有一个很大的限制，单点数据库的写入性能可能不是特别好，作为 ID 生成器，可能成为整个系统的性能瓶颈。

如何优化呢？我们一起来想一想。

水平扩展

既然单点写性能不高，我们如果扩展多个库，平均分摊流量是不是就可以了呢？这也是非常常用的提高系统吞吐量的办法。接下来的问题就是，多个库之间如何分配 ID 呢？



为了让每个库都能有独立的 ID 范围不至于产生冲突，我们可以为它们设置比数据库数量更高的值，作为 `auto_increment` 的步长，而且每个库采用不同的初始值，这样自然就可以保证每个库所能分配的 ID 是错开的。比如两个数据库，一个持有所有偶数 ID，一个持有所有奇数 ID。其他业务系统只需要轮询两个数据库，就可以得到粗略有序的全局唯一 ID 了。

为什么只是粗略有序，因为我们没办法保证所有依赖于此的服务，能按照时序轮流访问多个服务，但随着时间推移，只要负载均衡算法比较合理，整体 ID 还是在递增的。

但这样的系统也往往有一个问题：一旦数据库的数量定好了，就不太好再随意增加，必须重新划分每个数据库的初始值和步长。不过通常来说，这个问题也比较好处理，可以一开始就根据业务规模，设置足够多个数据库作为 ID 生成器，来避免扩展的需要。

但是如果直接用数据库来产生序号，会面临数据库写入瓶颈的问题。不过估计你也想到了刚才单点服务的思路，**如果我们把生成 ID 的响应服务和存储服务拆开，还是用单点对外提供 ID 发生服务，但是将 ID 状态记录在数据库中呢？**两者结合应该能获得更好的效果。

利用单点服务

具体做法就是在需要产生新的全局 ID 的时候，每次单点服务都向数据库批量申请 n 个 ID，在本地用内存维护这个号段，并把数据库中的 ID 修改为当前值 $+n$ ，直到这 n 个 ID 被耗尽；下次需要产生新的全局 ID 的时候，再次到数据库申请一段新的号段。

如果 ID 被耗尽之前，单点服务就挂了，也没关系，我们重启的时候直接向数据库申请下一次批次的 ID 就行，最多也就导致继续生成的 ID 和之前的批次不连续，这在大部分场景中都是可以接受的。

这样批量处理的设计，能大大减少数据库写的次数，把压力变成了原来的 $1/n$ ，性能大大提升，往往可以承载 10w 级的 QPS。这也是非常常见的减少服务压力的策略。


UUID

UUID (universally unique identifier) 这个词我们一开始就提到了，相信你不会很陌生，它本身就可以翻译成全局唯一 ID，但同时它也是一种常见的生成全局唯一 ID 的算法和协议。

和前面我们思考的两种方案不同，**这次的 ID 不再需要通过远程调用一个独立的服务产生，而是直接在业务侧的服务本地产生，所以 UUID 通常也被实现为一个库，供业务方直接调用。**UUID 有很多个不同的版本，网络上不同库的实现也可能会略有区别。

UUID 一共包含 32 位 16 进制数，也就是相当于 128 位二进制数，显示的时候被分为 8-4-4-4-12 几个部分，看一个例子：

```
1 0725f9ac-8cc1-11ec-a8a3-0242ac120002
```

 复制代码

我们就用 JDK 中自带的 UUID，来讲解一下第三和第四个版本的使用和主要思想，背后的逻辑主要是一些复杂的位运算，解释起来比较麻烦，对我们实际业务开发帮助不大，你感兴趣的话可以自己去看看相关的 [源代码](#)。

第三个版本的方法是基于名字计算的，名字由用户传入，它保证了不同空间不同名字下的 UUID 都具有唯一性，而相同空间相同名字下的 UUID 则是相同的：

```
1 public static UUID nameUUIDFromBytes(byte[] name)
```

[复制代码](#)

name 是用户自行传入的一段二进制，UUID 包会对其进行 MD5 计算以及一些位运算，最终得到一个 UUID。

第四个版本更加常用也更加直接一点，就是直接基于随机性进行计算，因为 UUID 非常长，所以其重复概率可以忽略不计。

```
1 public static UUID nameUUIDFromBytes(byte[] name)
```

[复制代码](#)

两个版本的使用都很简单：

```
1 UUID uuid = UUID.randomUUID();  
2 UUID uuid_ = UUID.nameUUIDFromBytes(nbyte);
```

[复制代码](#)

但 UUID 过于冗长，且主流版本完全无序，对数据库存储非常不利，这点我们之后介绍 B+ 树的时候也会展开讨论。

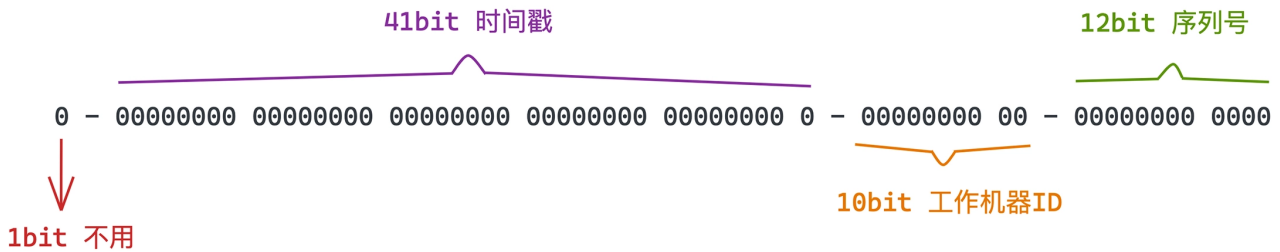
Snowflake

除了用户自己传入 name 来计算 UUID，UUID 其他几个版本里也有用到 MAC 地址，利用全球唯一性来标识不同的机器以及利用时间来保证有序性。不过 Mac 地址属于用户隐私，暴露出去不太好，也没有被广泛使用，但是思想还是可以被借鉴的。

Snowflake 就是这样一种引入了机器编号和时间信息的分布式 ID 生成算法，也是由业务方本地执行，由 twitter 开源，国内的美团和百度也都开源了基于各自业务场景的类似算法，感兴趣的同学可以搜索 leaf 和 UUID-generator，性能都很不错。

整个 Snowflake 生成的 UUID 都是 64 位的长整型，分为四个部分。

Snowflake 64bit



第一位是位保留位，置 0。

后面连续 41 位存储时间戳，可到毫秒级精度。

再后面 10 位代表机器 ID，由用户指定，相当于最多可以支持 1024 台机器。

最后 12 位表示序列号，是一个自增的序号，在时间相同的情况下，也就是 1ms 内可以支持 4096 个不同的序号，也就是在理论上来说 Snowflake 每秒可以产生 400w+ 个序号，这对于大部分业务场景来说都是绰绰有余的了。

Twitter 官方开源的版本是用 Scala 写的（网上也有人翻译了一个 [Java 版本](#)），因为思路其实很简单，所以代码也非常简洁，我这里写了点简单的注释，供你参考：

复制代码

```
1 package com.callicoder.snowflake;
2
3 import java.net.NetworkInterface;
4 import java.security.SecureRandom;
5 import java.time.Instant;
6 import java.util.Enumeraion;
7
8 /**
9  * Distributed Sequence Generator.
10  * Inspired by Twitter snowflake: https://github.com/twitter/snowflake/tree/sn
11  *
12  * This class should be used as a Singleton.
```



```
13  * Make sure that you create and reuse a Single instance of Snowflake per node
14  */
15  public class Snowflake {
16      private static final int UNUSED_BITS = 1; // Sign bit, Unused (always set
17      private static final int EPOCH_BITS = 41;
18      private static final int NODE_ID_BITS = 10;
19      private static final int SEQUENCE_BITS = 12;
20
21      private static final long maxNodeId = (1L << NODE_ID_BITS) - 1;
22      private static final long maxSequence = (1L << SEQUENCE_BITS) - 1;
23
24      // Custom Epoch (January 1, 2015 Midnight UTC = 2015-01-01T00:00:00Z)
25      private static final long DEFAULT_CUSTOM_EPOCH = 1420070400000L;
26
27      private final long nodeId;
28      private final long customEpoch;
29
30      private volatile long lastTimestamp = -1L;
31      private volatile long sequence = 0L;
32
33      // Create Snowflake with a nodeId and custom epoch
34      // 初始化需要传入节点ID和年代
35      public Snowflake(long nodeId, long customEpoch) {
36          if (nodeId < 0 || nodeId > maxNodeId) {
37              throw new IllegalArgumentException(String.format("NodeId must be b
38          }
39          this.nodeId = nodeId;
40          this.customEpoch = customEpoch;
41      }
42
43      // Create Snowflake with a nodeId
44      public Snowflake(long nodeId) {
45          this(nodeId, DEFAULT_CUSTOM_EPOCH);
46      }
47
48      // Let Snowflake generate a nodeId
49      public Snowflake() {
50          this.nodeId = createNodeId();
51          this.customEpoch = DEFAULT_CUSTOM_EPOCH;
52      }
53
54      // 这个函数用于获取下一个ID
55      public synchronized long nextId() {
56          long currentTimestamp = timestamp();
57
58          if (currentTimestamp < lastTimestamp) {
59              throw new IllegalStateException("Invalid System Clock!");
60          }
61
62          // 同一个时间戳, 我们需要递增序号
63          if (currentTimestamp == lastTimestamp) {
64              sequence = (sequence + 1) & maxSequence;
```

```
65         if(sequence == 0) {
66             // 如果序号耗尽,则需要等待到下一秒继续执行
67             // Sequence Exhausted, wait till next millisecond.
68             currentTimestamp = waitNextMillis(currentTimestamp);
69         }
70     } else {
71         // reset sequence to start with zero for the next millisecond
72         sequence = 0;
73     }
74
75     lastTimestamp = currentTimestamp;
76
77     long id = currentTimestamp << (NODE_ID_BITS + SEQUENCE_BITS)
78             | (nodeId << SEQUENCE_BITS)
79             | sequence;
80
81     return id;
82 }
83
84
85
86 // Get current timestamp in milliseconds, adjust for the custom epoch.
87 private long timestamp() {
88     return Instant.now().toEpochMilli() - customEpoch;
89 }
90
91 // 由于这样被耗尽的情况不多,且需要等待的时间也只有1ms;所以我们选择死循环进行阻塞
92 // Block and wait till next millisecond
93 private long waitNextMillis(long currentTimestamp) {
94     while (currentTimestamp == lastTimestamp) {
95         currentTimestamp = timestamp();
96     }
97     return currentTimestamp;
98 }
99
100 // 默认基于mac地址生成节点ID
101 private long createNodeId() {
102     long nodeId;
103     try {
104         StringBuilder sb = new StringBuilder();
105         Enumeration<NetworkInterface> networkInterfaces = NetworkInterface
106             .getNetworkInterfaces();
107         while (networkInterfaces.hasMoreElements()) {
108             NetworkInterface networkInterface = networkInterfaces.nextElement();
109             byte[] mac = networkInterface.getHardwareAddress();
110             if (mac != null) {
111                 for(byte macPort: mac) {
112                     sb.append(String.format("%02X", macPort));
113                 }
114             }
115         }
116         nodeId = sb.toString().hashCode();
117     } catch (Exception ex) {
```

```
117         nodeId = (new SecureRandom().nextInt());
118     }
119     nodeId = nodeId & maxNodeId;
120     return nodeId;
121 }
122
123 public long[] parse(long id) {
124     long maskNodeId = ((1L << NODE_ID_BITS) - 1) << SEQUENCE_BITS;
125     long maskSequence = (1L << SEQUENCE_BITS) - 1;
126
127     long timestamp = (id >> (NODE_ID_BITS + SEQUENCE_BITS)) + customEpoch;
128     long nodeId = (id & maskNodeId) >> SEQUENCE_BITS;
129     long sequence = id & maskSequence;
130
131     return new long[]{timestamp, nodeId, sequence};
132 }
133
134 @Override
135 public String toString() {
136     return "Snowflake Settings [EPOCH_BITS=" + EPOCH_BITS + ", NODE_ID_BIT
137         + ", SEQUENCE_BITS=" + SEQUENCE_BITS + ", CUSTOM_EPOCH=" + cus
138         + ", NodeId=" + nodeId + "];";
139 }
140 }
```

主要思路就是先根据 name，初始化 Snowflake generator 的实例，开发者需要保证 name 的唯一性；然后在需要生成新的 ID 的时候，用当前时间戳加上当前时间戳内（也就是某一毫秒内）的计数器，拼接得到 UUID，如果某一毫秒内的计数器被耗尽达到上限，会死循环直至这 1ms 过去。代码很简单，你看懂了吗。

再次说明一下，你千万不用太担心源码艰深复杂而不敢看，其实很多项目的源码还是很简单的，推荐你从这种小而美的代码开始看起，其实你看完之后，往往也会信心倍增，觉得自己也能写出来。等你之后看得多了，自然也能更好地掌握背后的编程技巧啦。

总结

主流的几种生成分布式唯一 ID 的方案我们就都学习完了，思路基本上都比较直接，大体分为两种思路：需要引入额外的系统生成 ID、在业务侧本地通过规则约束独立生成 ID。

单点生成器和基于数据库的实现都是第一种，UUID 和 Snowflake 则都是在本地根据规则约束独立生成 ID，一般来说也应用更加广泛。

你可以好好回顾感受学到的几个问题解决思想，备份节点来提高可用性、批量读写来提高系统性能、本地计算来避免性能瓶颈。之后，你自己引入外部数据库或者其他系统的时候，也要多多考虑是否会在引入的系统上发生问题和性能瓶颈。

课后作业

今天思考题就是前面说的，如果让你自己不借助外部组件实现一个单点的 ID 发生器，你会怎么做呢？

欢迎你在评论区留言与我一起讨论，如果觉得本文对你有帮助的话，也欢迎转发给你的朋友一起学习，我们下节课见～

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | Raft：分布式系统间如何达成共识？

下一篇 25 | 一致性哈希：如何在集群上合理分配流量？

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (1)

写留言



peter

2022-02-15

请教老师四个问题：

Q1：重试幂等使用的ID用什么方法生成？

Q2：个人电脑，比如我使用的笔记本，我并没有对它进行过时间同步操作，
请问这个笔记本会自动进行时间同步吗？

Q3：Snowflake算法的bit0为什么不用？maxNodeId为什么要1024减去1？...

展开

