链滴

Task Manage 分布式 ID 的思考

作者：dianjiu

原文链接：https://ld246.com/article/1593882776693

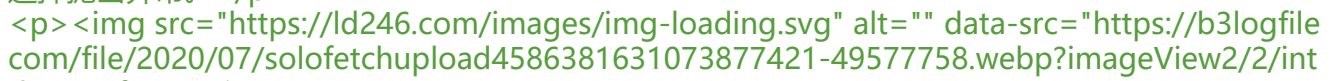
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20191114.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="一-需求背景">一、需求背景</h2>
由于系统设计设想，需满足以下场景：
任务列表中，同一任务组下有且只有一个同名任务。
即：一个系统中不允许有同名接口。
考虑到系统支持分布式部署，需要满足任务 ID 和任务组 ID 唯一。
<h2 id="二-参考资料">二、参考资料</h2>
分布式唯一 ID 有哪些特性或要求呢？
① 唯一性：生成的 ID 全局唯一，在特定范围内冲突概率极小。
② 有序性：生成的 ID 按某种规则有序，便于数据库插入及排序。
③ 可用性：可保证高并发下的可用性, 确保任何时候都能正确的生成 ID。
④ 自主性：分布式环境下不依赖中心认证即可自行生成 ID。
⑤ 安全性：不暴露系统和业务的信息, 如：订单数,用户数等。
分布式唯一 ID 有哪些生成方法呢？
总的来说，大概有三大类方法，分别是：数据库自增 ID、UUID 生成、snowflake 雪花算法。
<h2 id="1-数据库自增ID">1、数据库自增 ID</h2>
<h3 id="2-1-1-核心思想-">2.1.1 核心思想：</h3>
使用数据库的 id 自增策略（如: Mysql 的 auto_increment）。
<h3 id="2-1-2-优点-">2.1.2 优点：</h3>
① 简单，天然有序。
<h3 id="2-1-3-缺点-">2.1.3 缺点：</h3>
① 并发性不好。
② 数据库写压力大。
③ 数据库故障后不可使用。
④ 存在数量泄露风险。
<h3 id="2-1-4-优化方案-">2.1.4 优化方案:</h3>
1. 数据库水平拆分，设置不同的初始值和相同的自增步长
核心思想：将数据库进行水平拆分，每个数据库设置不同的初始值和相同的自增步长。
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2020/07/solofetchupload5874492596179439989-af1902fc.webp?imageView2/2/inter
ace/1/format/jpg">
如图所示，可保证每台数据库生成的 ID 是不冲突的，但这种固定步长的方式也会带来扩容的问
，很容易想到当扩容时会出现无 ID 初始值可分的窘境。解决方案有：
① 根据扩容考虑决定步长。
② 增加其他位标记区分扩容。
这其实都是在需求与方案间的权衡，根据需求来选择最适合的方式。
2. 批量缓存自增 ID
核心思想：如果使用单台机器做 ID 生成，可以避免固定步长带来的扩容问题（方案 1 的缺点）

具体做法是：每次批量生成一批 ID 给不同的机器去慢慢消费，这样数据库的压力也会减小到 N
之一，且故障后可坚持一段时间。
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2020/07/solofetchupload3972062550956190645-1207daa8.webp?imageView2/2/int
rlace/1/format/jpg">
如图所示，但这种做法的缺点是服务器重启、单点故障会造成 ID 不连续。
还是那句话，没有最好的方案，只有最适合的方案。
3. Redis 生成 ID
核心思想：Redis 的所有命令操作都是单线程的，本身提供像 incr 和 increby 这样的自增原子
令，所以能保证生成的 ID 肯定是唯一有序的。
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。
优化方案：
考虑到单节点的性能瓶颈，可以使用 Redis 集群来获取更高的吞吐量，并利用上面的方案（① 数据水平拆分，设置不同的初始值和相同的步长； ② 批量缓存自增 ID）来配置集群。
PS：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 + 当日增长号”，则可以每天在 Redis 中生成一个 Key，使用 INCR 进行累加。
<h2 id="2-UUID生成">2、UUID 生成</h2>
<h3 id="2-2-1-核心思想-">2.2.1 核心思想：</h3>
结合机器的网卡（基于名字空间/名字的散列值 MD5/SHA1）、当地时间（基于时间戳&钟序列）、一个随记数来生成 UUID。
其结构如下：
aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee（即包含 32 个 16 进制数字，以连字号-分为五段，最形成“8-4-4-4-12”的 36 个字符的字符串，即 32 个英数字母 +4 个连字号）。
例如：550e8400-e29b-41d4-a716-446655440000
<h3 id="2-2-2-优点-">2.2.2 优点：</h3>
① 本地生成，没有网络消耗，生成简单，没有高可用风险。
<h3 id="2-2-3-缺点-">2.2.3 缺点：</h3>
① 不易于存储：UUID 太长，16 字节 128 位，通常以 36 长度的字符串表示，很多场景不适用
② 信息不安全：基于 MAC 地址生成 UUID 的算法可能会造成 MAC 地址泄露，这个漏洞曾被用于寻梅丽莎病毒的制作者位置。
③ 无序查询效率低：由于生成的 UUID 是无序不可读的字符串，所以其查询效率低。
<h3 id="2-2-4-生成UUID的5种方式--">2.2.4 生成 UUID 的 5 种方式： </h3>
① 版本 1 - 基于时间的 UUID（date-time & MAC address）：
规则：主要依赖当前的时间戳及机器 mac 地址，因此可以保证全球唯一性。
优点：能基本保证全球唯一性。
缺点：使用了 Mac 地址，因此会暴露 Mac 地址和生成时间。
② 版本 2 - 分布式安全的 UUID（date-time & group/user id）：
规则：将版本 1 的时间戳前四位换为 POSIX 的 UID 或 GID，很少使用。
优点：能保证全球唯一性。
缺点：很少使用，常用库基本没有实现。
③ 版本 3 - 基于名字空间的 UUID-MD5 版（MD5 hash & namespace）：
规则：基于指定的名字空间/名字生成 MD5 散列值得到，标准不推荐。
优点：不同名字空间或名字下的 UUID 是唯一的；相同名字空间及名字下得到的 UUID 保持重复。
缺点：MD5 碰撞问题，只用于向后兼容，后续不再使用。
④ 版本 4 - 基于随机数的 UUID（pseudo-random number）：
规则：基于随机数或伪随机数生成。
优点：实现简单。
缺点：重复几率可计算。机率也与随机数产生器的质量有关。若要避免重复机率提高，必须要使用基密码学上的强伪随机数产生器来生成值才行。
⑤ 版本 5 - 基于名字空间的 UUID-SHA1 版（SHA-1 hash & namespace）：
规则：将版本 3 的散列算法改为 SHA1。
优点：不同名字空间或名字下的 UUID 是唯一的；相同名字空间及名字下得到的 UUID 保持重复。
缺点：SHA1 计算相对耗时。
总得来说：
① 版本 1/2 适用于需要高度唯一性且无需重复的场景。
② 版本 3/5 适用于一定范围内唯一且需要或可能会重复生成 UUID 的环境下。
③ 版本 4 适用于对唯一性要求不太严格且追求简单的场景。
<h2 id="3-雪花算法">3、雪花算法</h2>
<h3 id="3-3-1-核心思想-">3.3.1 核心思想：</h3>
把 64-bit 分别划分成多段，分开来标示机器、时间、某一并发序列等，从而使每台机器及同一

器生成的 ID 都是互不相同。
PS：这种结构是雪花算法提出者 Twitter 的分法，但实际上这种算法使用可以很灵活，根据自身务的并发情况、机器分布、使用年限等，可以自由地重新决定各部分的位数，从而增加或减少某部分量级。比如：百度的 UidGenerator、美团的 Leaf 等，都是基于雪花算法做一些适合自身业务的变化
<h3 id="3-3-2-雪花算法的4种优化方案-">3.3.2 雪花算法的 4 种优化方案：</h3>
1、Twitter 的 snowflake 算法
核心思想是：采用 bigint（64bit）作为 id 生成类型，并将所占的 64bit 划分成多段。
其结构如下：


说明：
①1 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。
②41 位时间截(毫秒级）：需要注意的是，41 位时间截不是存储当前时间的时间截，而是存储间截的差值（当前时间截 - 开始时间截）得到的值，这里的开始时间截，一般是指我们的 id 生成器始使用的时间截，由我们的程序来指定。41 位的毫秒时间截，可以使用 69 年（即 T =（1L << 1）/（1000 * 60 * 60 * 24 * 365）= 69）。
③10 位的数据机器位：包括 5 位数据中心标识 Id（datacenterId）、5 位机器标识 Id(workerId，最多可以部署 1024 个节点（即 1 << 10 = 1024）。超过这个数量，生成的 ID 就有可能会突。
④12 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，同一时间截产生 4096 个 ID 序号（即 1 << 12 = 4096）。
PS：全部结构标识（1+41+10+12=64）加起来刚好 64 位，刚好凑成一个 Long 型。
优点：
① 整体上按照时间按时间趋势递增，后续插入索引树的时候性能较好。
② 整个分布式系统内不会产生 ID 碰撞（由数据中心标识 ID、机器标识 ID 作区分）。
③ 本地生成，且不依赖数据库（或第三方组件），没有网络消耗，所以效率高（每秒能够产生 26 万 D 左右）。
缺点：
① 由于雪花算法是强依赖于时间的，在分布式环境下，如果发生时钟回拨，很可能会引起 ID 重复、ID 乱序、服务会处于不可用状态等问题。
解决方案有：
a. 将 ID 生成交给少量服务器，并关闭时钟同步。
b. 直接报错，交给上层业务处理。
c. 如果回拨时间较短，在耗时要求内，比如 5ms，那么等待回拨时长后再进行生成。
d. 如果回拨时间很长，那么无法等待，可以匀出少量位（1~2 位）作为回拨位，一旦时钟回拨，将拨位加 1，可得到不一样的 ID，2 位回拨位允许标记 3 次时钟回拨，基本够使用。如果超出了，可以选择抛出异常。


2、Mongo 的 ObjectId 算法
核心思想是：使用 12 字节（24bit）的 BSON 类型字符串作为 ID，并将所占的 24bit 划分成多。 

说明：
①4 字节（8 位）timeStamp：UNIX 时间戳（精确到秒）。
②3 字节（6 位）machine：所在主机的唯一标识符（一般是机器主机名的散列值）。
③2 字节（4 位）pid：同一台机器不同的进程产生 objectid 的进程标识符 。

④3 字节（6 位）increment：由一个随机数开始的计数器生成的自动增加的值，用来确保在同一秒产生的 objectid 也不会发现冲突，允许 256^3（16777216）条记录的唯一性。
如：ObjectID（为了方便查看，每部分使用“-”分隔）格式为：5dba76a3-d2c366-7f99-57df0
①timeStamp：5dba76a3（对应十进制为：1572501155）。
②machine：d2c366（对应十进制为：13812582）。
③pid：7f99（对应十进制为：32665）。
④increment：57dfb0（对应十进制为：5758896）。
优点：
① 本地生成，没有网络消耗，生成简单，没有高可用风险。
② 所生成的 ID 包含时间信息，可以提取时间信息。
缺点：
① 不易于存储：12 字节 24 位长度的字符串表示，很多场景不适用。
◆ 新版 ObjectId 中“机器标识码 + 进程号” 改为用随机数作为机器标识和进程号的值
mark：从 MongoDB 3.4 开始（最早发布于 2016 年 12 月），ObjectId 的设计被修改了，中间 5 字节的值由原先的 “机器标识码 + 进程号” 改为用随机数作为机器标识和进程号的值。
那问题来了，为什么不继续使用“机器标识 + 进程号”呢？
问题就在于，在这个物理机鲜见，虚拟机、云主机、容器横行的时代，机器标识和进程号不太可。
① 机器标识码：
ObjectId 的机器标识码是取系统 hostname 哈希值的前几位。那么问题来了，准备了几台虚拟机，hstname 都是默认的 localhost，谁都想着这玩意儿能有什么用，还得刻意给不同机器起不同的 hostnme？ 此外，hostname 在容器、云主机里一般默认就是随机数，也不会检查同一集群里是否有 hostame 重名。
② 进程号：
这个问题就更大了，要知道，容器内的进程拥有自己独立的进程空间，在这个空间里只用它自己这一进程（以及它的子进程），所以它的进程号永远都是 1。也就是说，如果某个服务（既可以是 mongo 实例也可以是 mongo 客户端）是使用容器部署的，无论部署多少个实例，在这个服务上生成的 ObjetId，第八第九个字节恒为 0000 0001，相当于说这两个字节废了。
综上，与其使用一个固定值来“区分不同进程实例”，且这个固定值还是人类随意设置或随机生的 hostname 加上一个可能恒为 1 的进程号，倒不如每次都随机生成一个新值。
可见，这是平台层面的架构变动影响了应用层面的设计方案，随着云、容器的继续发展，这样的事还会继续上演。
**旧版：使用主机名的散列值作用 machine、使用进程标识符作为 pid **

新版：使用随机数作为 machine、pid 的值

3、百度 UidGenerator 算法
UidGenerator 是百度开源的分布式 ID 生成器，是基于 snowflake 算法的实现，看起来感觉还行，是需要借助数据库，配置起来比较复杂。
具体可以参考官网说明：https://github.com/baidu/uid-generator/blob/master/README.zh_cn.md
4、美团 Leaf 算法
Leaf 是美团开源的分布式 ID 生成器，能保证全局唯一性、趋势递增、单调递增、信息安全，里面也到了几种分布式方案的对比，但也需要依赖关系数据库、Zookeeper 等中间件。
具体可以参考官网说明：https://ech.meituan.com/2017/04/21/mt-leaf.html

<h2 id="三-系统方案">三、系统方案</h2>
考虑到依赖问题，系统复杂度问题，业务场景，且任务信息表的 id 是自增的。
故决定 taskNo 和 groupNo 分别采用 T-UUID 和 G-UUID 的方式。