# 版本更新

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 修改时间 | 修改人 | 描述 |
| V1.0 | 2018/11/28 | stone | 新建文档 |

# 引子

在复杂的系统中，往往需要使用一个有意义的有序的序列号来作为全局唯一的ID对大量的数据如订单,账户进行标识。

# 业内方案简介

##### 时间戳方案

取当前毫秒数/微秒作为ID 如

System.currentTimeMillis()

**优点**：

本地生成ID，不需要进行远程调用，时延低；

生成的ID趋势递增；

生成的ID是整数，建立索引后查询效率高。

**缺点**：

并发量过高，会生成重复的ID。

不能高可用，存在单点故障问题。

不够灵活，不能实现对不同业务的ID隔离。

##### UUID方案

UUID(Universally Unique Identifier)的标准型式包含32个16进制数字，以连字号分为五段，形式为8-4-4-4-12的36个字符，示例：550e8400-e29b-41d4-a716-446655440000，到目前为止业界一共有5种方式生成UUID，详情见IETF发布的UUID规范 [A Universally Unique IDentifier (UUID) URN Namespace](http://www.ietf.org/rfc/rfc4122.txt)。

**优点**：

性能非常高：本地生成，没有网络消耗。

**缺点**：

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。

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找梅丽莎病毒的制作者位置。

无序，ID于ID之前毫无顺序可言。

ID作为主键时在特定的环境会存在一些问题，比如做DB主键的场景下，UUID就非常不适用：

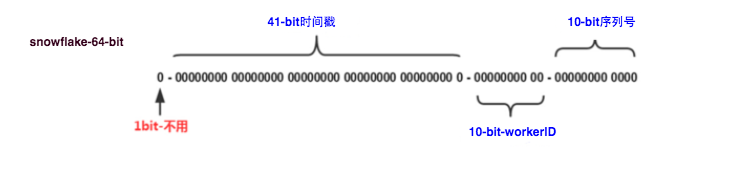
① MySQL官方有明确的建议主键要尽量越短越好[4]，36个字符长度的UUID不符合要求。

All indexes other than the clustered index are known as secondary indexes. In InnoDB, each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index. InnoDB uses this primary key value to search for the row in the clustered index. If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key.

② 对MySQL索引不利：如果作为数据库主键，在InnoDB引擎下，UUID的无序性可能会引起数据位置频繁变动，严重影响性能。

##### Snowflake方案

这种方案大致来说是一种以划分命名空间（UUID也算，由于比较常见，所以单独分析）来生成ID的一种算法，这种方案把64-bit分别划分成多段，分开来标示机器、时间等，比如在snowflake中的64-bit分别表示如下图（图片来自网络）所示：



41-bit的时间可以表示（1L<<41）/(1000L\*3600\*24\*365)=69年的时间，10-bit机器可以分别表示1024台机器。如果我们对IDC划分有需求，还可以将10-bit分5-bit给IDC，分5-bit给工作机器。这样就可以表示32个IDC，每个IDC下可以有32台机器，可以根据自身需求定义。12个自增序列号可以表示2^12个ID，理论上snowflake方案的QPS约为409.6w/s，这种分配方式可以保证在任何一个IDC的任何一台机器在任意毫秒内生成的ID都是不同的。

**优点**：

毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。

不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的。

可以根据自身业务特性分配bit位，非常灵活。

**缺点**：

强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。

不够灵活，不能实现对不同业务的ID隔离。

##### 数据库auto\_increment方案

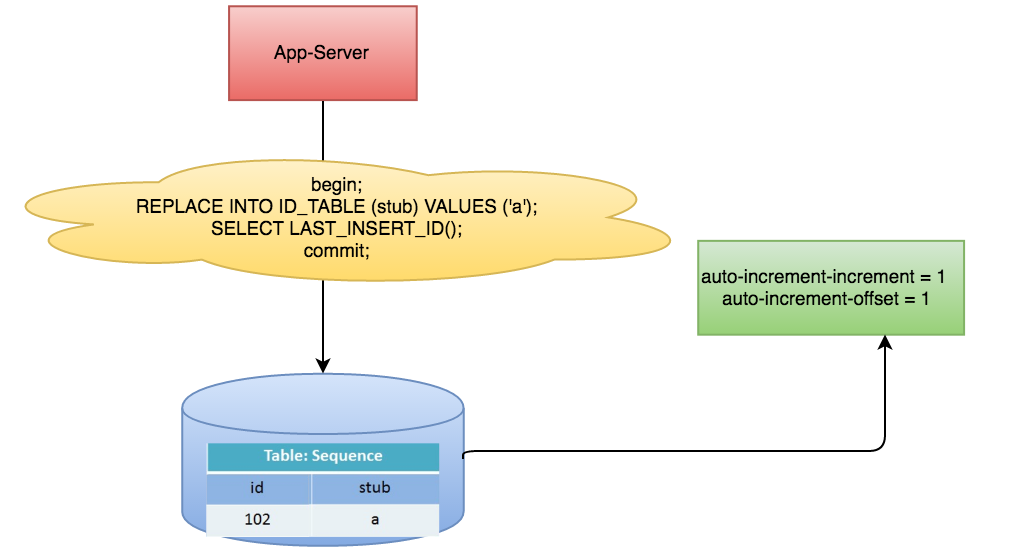
以MySQL举例，利用给字段设置auto\_increment\_increment和auto\_increment\_offset来保证ID自增，每次业务使用下列SQL读写MySQL得到ID号。

begin;

REPLACE INTO Tickets64 (stub) VALUES ('a');

SELECT LAST\_INSERT\_ID();

commit;



**优点**：

非常简单，利用现有数据库系统的功能实现，成本小，有DBA专业维护。

ID号单调自增，可以实现一些对ID有特殊要求的业务。

**缺点**：

强依赖DB，当DB异常时整个系统不可用，属于致命问题。配置主从复制可以尽可能的增加可用性，但是数据一致性在特殊情况下难以保证。主从切换时的不一致可能会导致重复发号。

ID发号性能瓶颈限制在单台MySQL的读写性能。

##### redis生成ID

Redis的所有命令操作都是单线程的，本身提供像 incr 和 increby 这样的自增原子命令，所以能保证生成的 ID 肯定是唯一有序的。

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。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

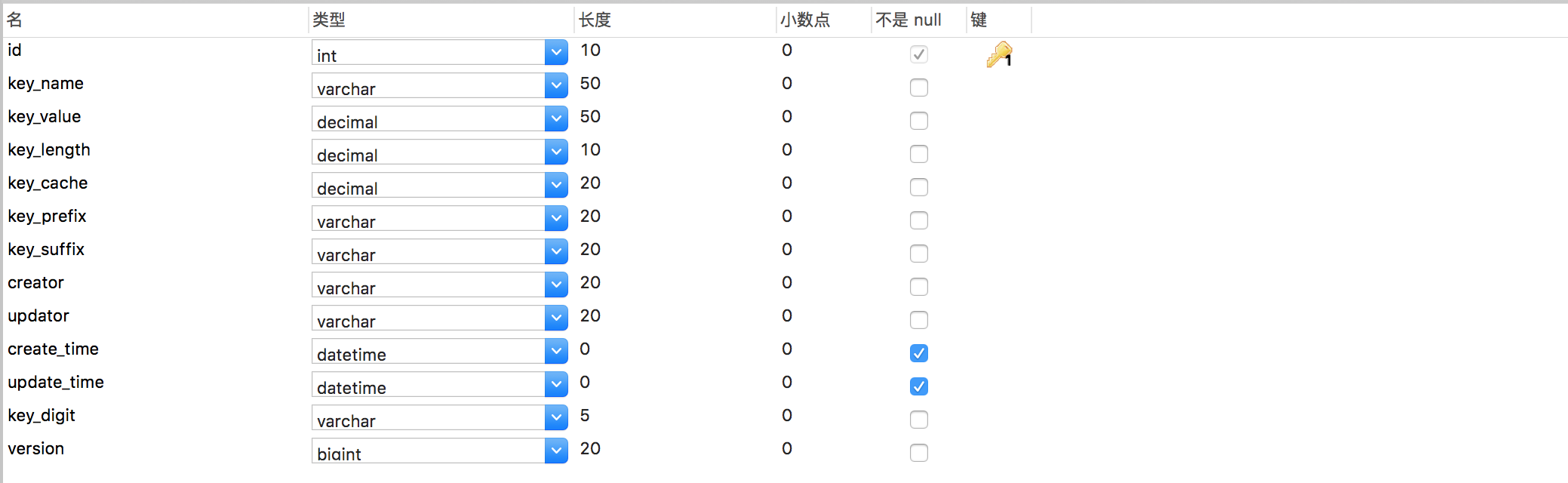
E：5, 10, 15, 20, 25

**优点**：不依赖于数据库，灵活方便，且性能优于数据库；数字ID天然排序，对分页或者需要排序的结果很有帮助；使用集群可以防止单点故障问题。

**缺点**：如果系统中没有Redis，还需要引入新的组件，增加系统复杂度；需要编码和配置的工作量比较大；步长、初始值需提前确定好且不易于扩展。

##### idgen方案

先看下数据库表设计：



字段说明：

id :数据库主键，无实际含义

key\_name :用来区分业务，不同的业务使用不同的key\_name，每个key\_name的ID相互隔离，互不影响。如果以后有性能需求需要对数据库扩容，不需要上述描述的复杂的扩容操作，只需要对biz\_tag分库分表就行。

key\_value：表示该key\_name目前所被分配的ID号段的最大值。

key\_length：生成ID的长度。

key\_cache：表示每次分配的号段长度。原来获取ID每次都需要写数据库，现在只需要把key\_cache设置得足够大，比如1000。那么只有当1000个号被消耗完了之后才会去重新读写一次数据库。读写数据库的频率从1减小到了1/step。

key\_prefix：生成ID的前缀，可配置 自定义前缀+日期部分 如：${date14}/TEST${date14}

ID前缀日期部分支持以下几种日期格式：

|  |  |  |
| --- | --- | --- |
| 前缀日期部分配置 | 日期格式 | 样例 |
| ${date6} | yyMMdd | 191011 |
| ${date8} | yyMMddHH | 19101123 |
| ${date10} | yyMMddHHmm | 1910112322 |
| ${date12} | yyMMddHHmmss | 191011232218 |
| ${date14} | yyyyMMddHHmmss | 20191011232218 |
| ${date15} | yyMMddHHmmssSSS | 191011232218079 |
| ${date16} | yyyMMddHHmmssSSS | 0191011232218079 |
| ${date17} | yyyyMMddHHmmssSSS | 20191011232218079 |

key\_suffix：生成ID的后缀，可配置亦可不配置

key\_digit：ID进制数，支持10进制 36进制 62进制

version：每条记录对应的版本号，用户更新记录。

**优点**：

很方便的线性扩展，能够支撑大多数业务场景。

生成ID规则多样，可配置且支持10进制、36进制、62进制

业务之间ID相互隔离，互不影响。

获取ID不用频繁操作数据库，快消耗完号段内ID时才会操作数据库，减轻了数据库的压力。

提前初始化号段内的ID，保证在每个号段内ID使用完之前初始化完成，避免业务使用完ID后才初始化带来的影响。

可以自定义key\_value的大小，非常方便业务从原有的ID方式上迁移过来。

容灾性高：服务内部有号段缓存，即使DB宕机，短时间内服务仍能正常对外提供服务。

# 功能介绍

该ID生成器是基于NS4框架实现的，支持分布式部署，同时生成的ID长度，前缀，后缀，步长，进制也可根据自己的业务自由的配置。

其功能可分为以下几个部分：

1、获取单个Long类型的ID如 66310

2、获取单个String类型的ID如 19011123221266311

3、获取批量String类型的ID：19011123221266312, 19011123221266313, 19011123221266314, 19011123221266315

# 请求方式

[见帮助文档](http://10.100.139.25:8181/readme.html" \l "/readme/idGen)

# 性能相关

[见帮助文档](http://10.100.139.25:8181/readme.html" \l "/readme/idGen)

# git地址

git@top-git.creditease.corp:CDC/ns4\_gear.git

# SQL脚本

见ns4\_gear\_idgen源码下gear\_key.sql