# 分布式id生成

在复杂分布式系统中，往往需要对大量的数据和消息进行唯一标识。如在美团点评的金融、支付、餐饮、酒店、猫眼电影等产品的系统中，数据日渐增长，对数据分库分表后需要有一个唯一ID来标识一条数据或消息，数据库的自增ID显然不能满足需求；特别一点的如订单、骑手、优惠券也都需要有唯一ID做标识。此时一个能够生成全局唯一ID的系统是非常必要的。业务系统对ID号的要求：

1. 全局唯一性：不能出现重复的ID号，既然是唯一标识，这是最基本的要求。
2. 趋势递增：在MySQL InnoDB引擎中使用的是聚集索引，由于多数RDBMS使用B-tree的数据结构来存储索引数据，在主键的选择上面我们应该尽量使用有序的主键保证写入性能。
3. 单调递增：保证下一个ID一定大于上一个ID，例如事务版本号、IM增量消息、排序等特殊需求。
4. 信息安全：如果ID是连续的，恶意用户的扒取工作就非常容易做了，直接按照顺序下载指定URL即可；如果是订单号就更危险了，竞对可以直接知道我们一天的单量。所以在一些应用场景下，会需要ID无规则、不规则。

上述123对应三类不同的场景，3和4需求还是互斥的，无法使用同一个方案满足。

## 常见方法介绍

### UUID

UUID出现的目的，是为了让分布式系统可以不借助中心节点，就可以生成UUID来标识一些唯一的信息；

#### 各个版本对比

##### 版本1：基于时间的UUID

通过当前时间戳、机器MAC地址生成；

由于在算法中使用了MAC地址，这个版本的UUID可以保证在全球范围的唯一性。

但与此同时，因为它暴露了电脑的MAC地址和生成这个UUID的时间，这就是这个版本UUID被诟病的地方。

##### 版本3和5：基于名字空间的UUID（MD5，SHA1）

由用户指定1个namespace和1个具体的字符串，通过MD5（SHA1）散列，来生成1个UUID；

##### 版本4：基于随机数的UUID

根据随机数，或者伪随机数生成UUID。这种UUID产生重复的概率是可以计算出来的，但随机的东西就像是买彩票：你指望它发财是不可能的，但狗屎运通常会在不经意中到来。这个版本应该是平时大家无意中用得最多的版本了。

#### 优缺点：

优点：

* 性能非常高：本地生成，没有网络消耗。

缺点：

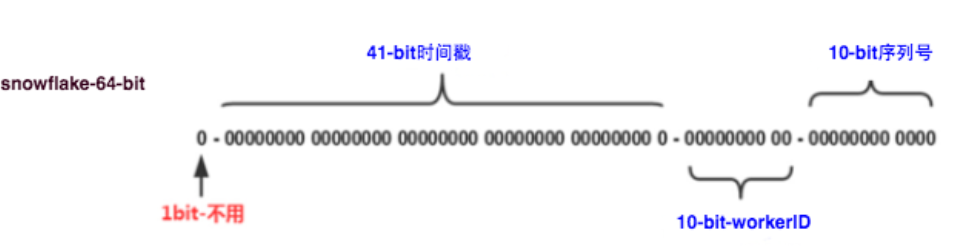
* 不易于存储：UUID太长，16字节128位，通常以36长度的字符串表示，很多场景不适用。
* 信息不安全：基于MAC地址生成UUID的算法可能会造成MAC地址泄露，这个漏洞曾被用于寻找梅丽莎病毒的制作者位置。
* ID作为主键时在特定的环境会存在一些问题，比如做DB主键的场景下，UUID就非常不适用：

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。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。（随机数方法）

### 类snowflake方案

这种方案大致来说是一种以划分命名空间来生成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：



序列号是12位，是自增的。

#### 优缺点：

优点：

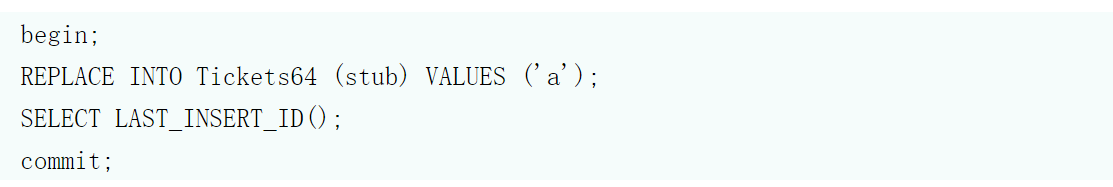
* 毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。
* 不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的。
* 可以根据自身业务特性分配bit位，非常灵活。

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。

### 数据库生成

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。



Replace into是Insert into的增强版。在向表中插入数据时，我们经常会遇到这样的情况：1、首先判断数据是否存在；2、如果不存在，则插入；3、如果存在，则更新。

#### 优缺点：

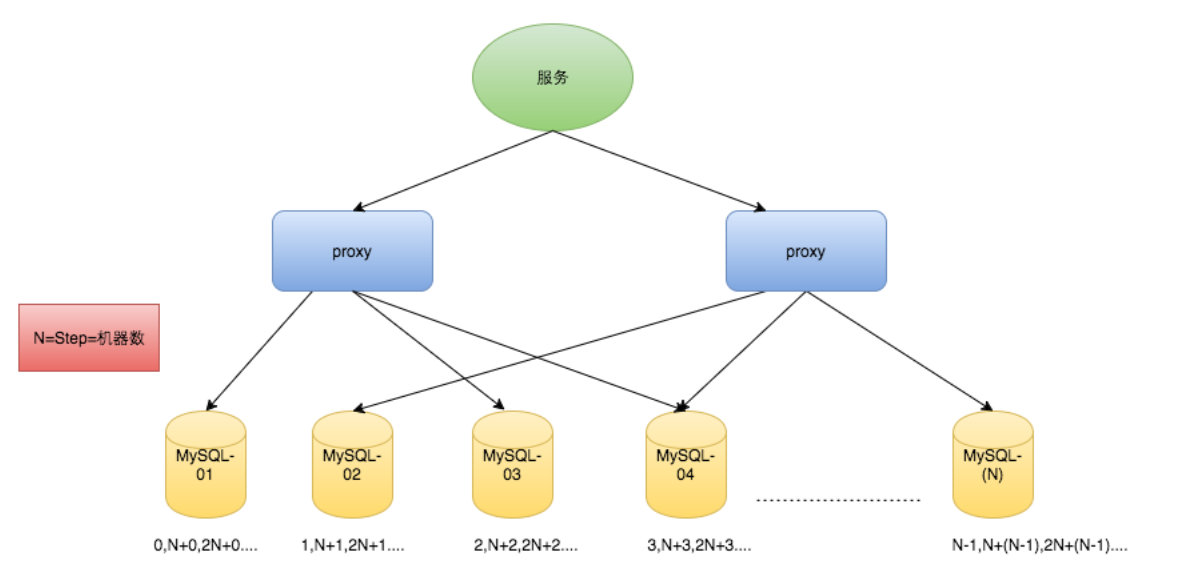
优点：

* 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。

对于MySQL性能问题，可用如下方案解决：在分布式系统中我们可以多部署几台机器，每台机器设置不同的初始值，且步长和机器数相等。比如有两台机器。设置步长step为2，TicketServer1的初始值为1（1，3，5，7，9，11...）、TicketServer2的初始值为2（2，4，6，8，10...）。这是Flickr团队在2010年撰文介绍的一种主键生成策略（[Ticket Servers: Distributed Unique Primary Keys on the Cheap](http://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/)）。如下所示，为了实现上述方案分别设置两台机器对应的参数，TicketServer1从1开始发号，TicketServer2从2开始发号，两台机器每次发号之后都递增2。



这种架构貌似能够满足性能的需求，但有以下几个缺点：

1.系统水平扩展比较困难，比如定义好了步长和机器台数之后，如果要添加机器该怎么做。

2. ID没有了单调递增的特性，只能趋势递增。

3. 数据库压力还是很大，每次获取ID都得读写一次数据库，只能靠堆机器来提高性能。

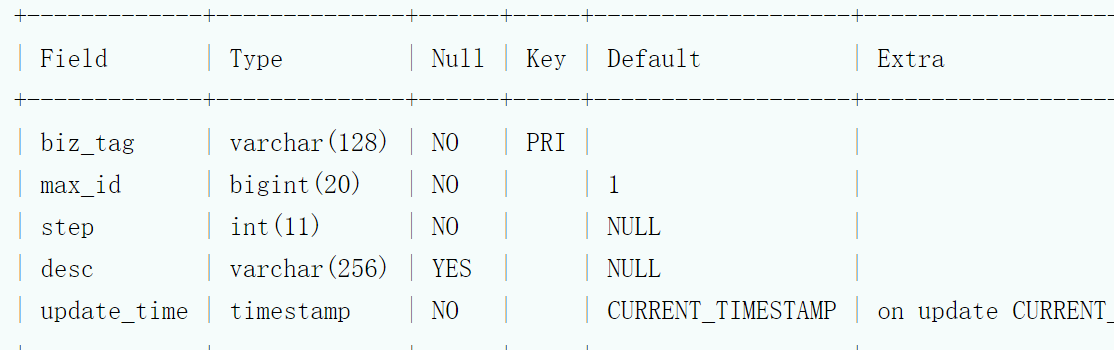
## Leaf方案实现

### Leaf-segment数据库方案

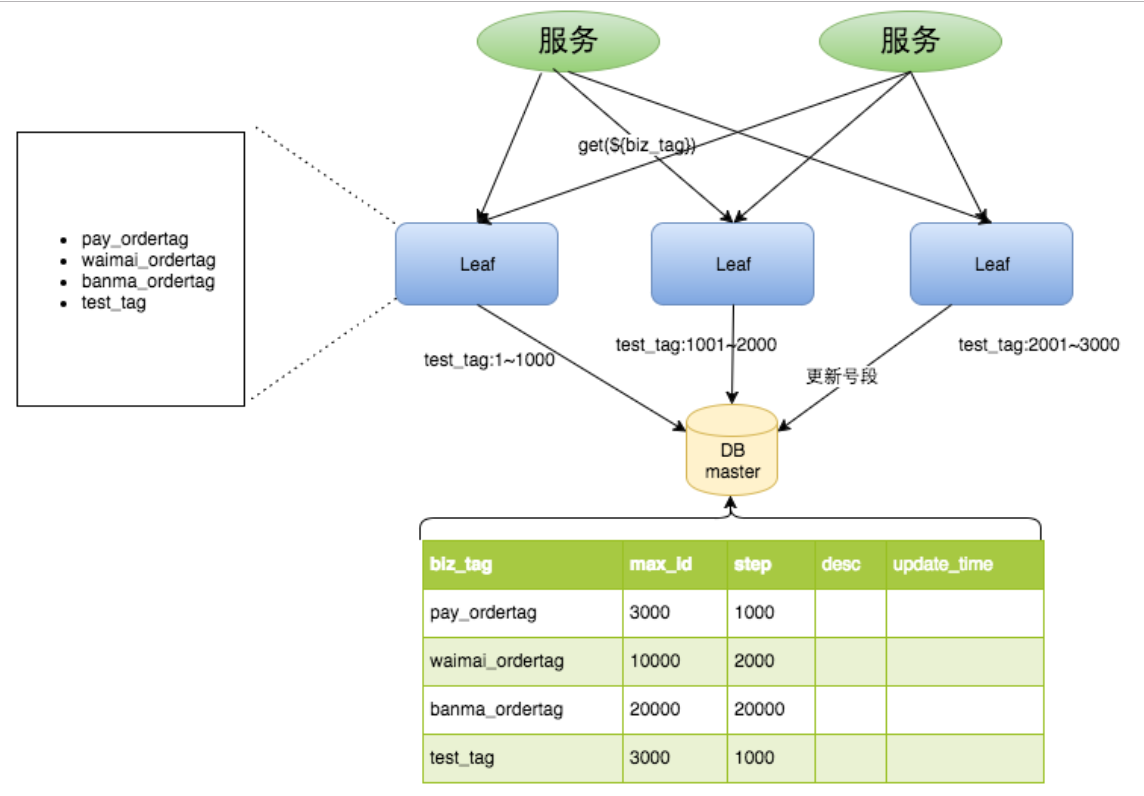
Leaf-segment方案，在使用数据库的方案上，做了如下改变：

* 原方案每次获取ID都得读写一次数据库，造成数据库压力大。改为利用proxy server批量获取，每次获取一个segment(step决定大小)号段的值。用完之后再去数据库获取新的号段，可以大大的减轻数据库的压力。
* 各个业务不同的发号需求用biz\_tag字段来区分，每个biz-tag的ID获取相互隔离，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。

数据库表设计如下：



biz\_tag用来区分业务，max\_id表示该biz\_tag目前所被分配的ID号段的最大值，step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step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。



test\_tag在第一台Leaf机器上是1~1000的号段，当这个号段用完时，会去加载另一个长度为step=1000的号段，假设另外两台号段都没有更新，这个时候第一台机器新加载的号段就应该是3001~4000。同时数据库对应的biz\_tag这条数据的max\_id会从3000被更新成4000。

#### 优缺点

优点：

* Leaf服务可以很方便的线性扩展，性能完全能够支撑大多数业务场景。
* ID号码是趋势递增的8byte的64位数字，满足上述数据库存储的主键要求。
* 容灾性高：Leaf服务内部有号段缓存，即使DB宕机，短时间内Leaf仍能正常对外提供服务。
* 可以自定义max\_id的大小，非常方便业务从原有的ID方式上迁移过来。

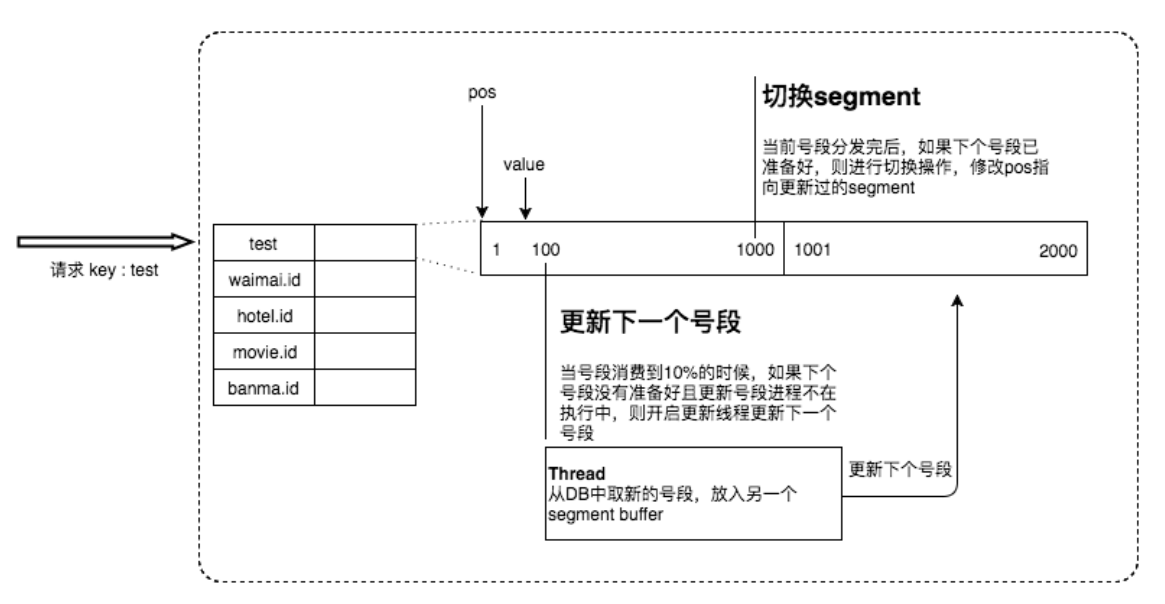
缺点：

* ID号码不够随机，能够泄露发号数量的信息，不太安全。
* TP999数据波动大，当号段使用完之后还是会hang在更新数据库的I/O上，tg999数据会出现偶尔的尖刺。
* DB宕机会造成整个系统不可用。

#### 双buffer优化

Leaf 取号段的时机是在号段消耗完的时候进行的，也就意味着号段临界点的ID下发时间取决于下一次从DB取回号段的时间，并且在这期间进来的请求也会因为DB号段没有取回来，导致线程阻塞。如果请求DB的网络和DB的性能稳定，这种情况对系统的影响是不大的，但是假如取DB的时候网络发生抖动，或者DB发生慢查询就会导致整个系统的响应时间变慢。

为此，我们希望DB取号段的过程能够做到无阻塞，不需要在DB取号段的时候阻塞请求线程，即当号段消费到某个点时就异步的把下一个号段加载到内存中。而不需要等到号段用尽的时候才去更新号段。



采用双buffer的方式，Leaf服务内部有两个号段缓存区segment。当前号段已下发10%时，如果下一个号段未更新，则另启一个更新线程去更新下一个号段。当前号段全部下发完后，如果下个号段准备好了则切换到下个号段为当前segment接着下发，循环往复。

### Leaf-snowflake方案

Leaf-segment方案可以生成趋势递增的ID，同时ID号是可计算的，不适用于订单ID生成场景，比如竞对在两天中午12点分别下单，通过订单id号相减就能大致计算出公司一天的订单量，这个是不能忍受的。面对这一问题，我们提供了 Leaf-snowflake方案。

Leaf-snowflake方案完全沿用snowflake方案的bit位设计，即是“1+41+10+12”的方式组装ID号。对于workerID的分配，当服务集群数量较小的情况下，完全可以手动配置。Leaf服务规模较大，动手配置成本太高。所以使用Zookeeper持久顺序节点的特性自动对snowflake节点配置wokerID。

snowflake是按照下面几个步骤启动的：

1. 启动Leaf-snowflake服务，连接Zookeeper，在leaf\_forever父节点下检查自己是否已经注册过（是否有该顺序子节点）。
2. 如果有注册过直接取回自己的workerID（zk顺序节点生成的int类型ID号），启动服务。
3. 如果没有注册过，就在该父节点下面创建一个持久顺序节点，创建成功后取回顺序号当做自己的workerID号，启动服务。

除了每次会去ZK拿数据以外，也会在本机文件系统上缓存一个workerID文件。当ZooKeeper出现问题，恰好机器出现问题需要重启时，能保证服务能够正常启动。这样做到了对三方组件的弱依赖。

#### 解决时钟问题

服务启动时首先检查自己是否写过ZooKeeper leaf\_forever节点：

1. 若写过，则用自身系统时间与leaf\_forever/${self}节点记录时间做比较，若小于leaf\_forever/${self}时间则认为机器时间发生了大步长回拨，服务启动失败并报警。
2. 若未写过，证明是新服务节点，直接创建持久节点leaf\_forever/${self}并写入自身系统时间，接下来综合对比其余Leaf节点的系统时间来判断自身系统时间是否准确，具体做法是取leaf\_temporary下的所有临时节点(所有运行中的Leaf-snowflake节点)的服务IP：Port，然后通过RPC请求得到所有节点的系统时间，计算sum(time)/nodeSize。
3. 若abs( 系统时间-sum(time)/nodeSize ) < 阈值，认为当前系统时间准确，正常启动服务，同时写临时节点leaf\_temporary/${self} 维持租约。
4. 否则认为本机系统时间发生大步长偏移，启动失败并报警。
5. 每隔一段时间(3s)上报自身系统时间写入leaf\_forever/${self}。

由于强依赖时钟，对时间的要求比较敏感，要么在时钟回拨的时候直接不提供服务直接返回ERROR\_CODE，等时钟追上即可。或者做一层重试，然后上报报警系统，更或者是发现有时钟回拨之后自动摘除本身节点并报警。