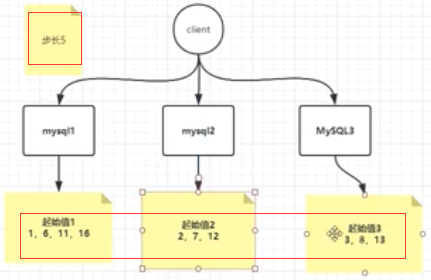
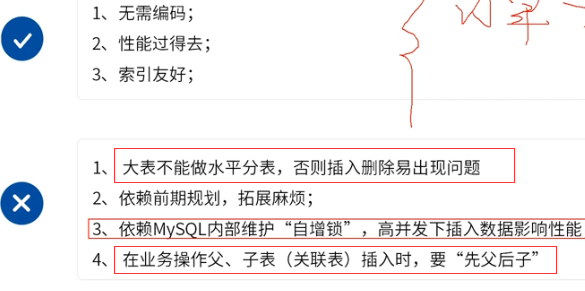
**分布式ID的生成策略**

## 为什么需要分布式ID

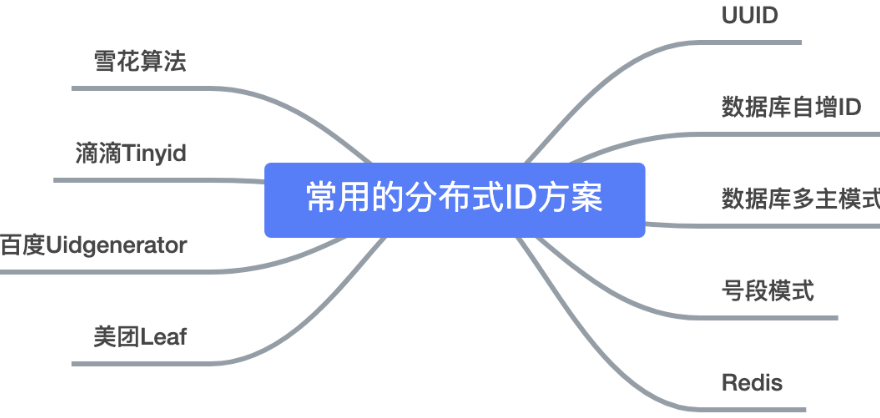
一般单机或者单数据库的项目可能规模比较小，适应的场景也比较有限，平台的访问量和业务量都较小，业务ID的生成方式比较原始但是够用，它并没有给这样的系统带来问题和瓶颈，所以这种情况下我们并没有对此给予太多的关注。但是对于大厂的那种大规模复杂业务、分布式高并发的应用场景，显然这种ID的生成方式不会像小项目一样仅仅依靠简单的数据自增序列来完成，而且在分布式环境下这种方式已经无法满足业务的需求，不仅无法完成业务能力，业务ID生成的速度或者重复问题可能给系统带来严重的故障。



## 分布式ID的特性

* 全局唯一：这是基本要求，不能出现重复。
* 具有递增的趋势:后面的ID必须比前面的大。
* 长度固定:能够提高查询效率。
* 安全: 如果ID连续生成，势必会泄露业务信息，甚至可能被猜出，所以需要无规则不规则。
* 高可用延时:ID生成快，能够扛住高并发，延时足够低不至于成为业务瓶颈。

## 分布式ID的几种常见解决方案



## 分析解决方案

### 4.1基于UUID

UUID是全球唯一的特性深入人心，熟悉MySQL数据库特性的人，应该不会用此来作为业务ID，它不可读而且过于长。

原理：使用以太网卡地址，纳秒级时间，芯片和许多可能的数字。

写sql语句验证：

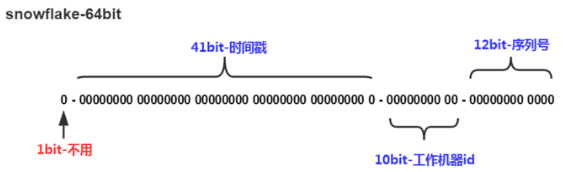
Select id, count（\*） as count from order group by id having count > 1

* 优点
* 代码实现足够简单易用。
* 本地生成没有性能问题。
* 因为具备全球唯一的特性，所以对于数据库迁移这种情况不存在问题。
* 缺点
* 每次生成的ID都是无序的，而且不是全数字，且无法保证趋势递增。
* UUID生成的是字符串，字符串存储性能差，查询效率慢。
* UUID长度过长，不适用于存储，耗费数据库性能。
* ID无一定业务含义，可读性差。
* 适用场景
* 可以用来生成如token令牌一类的场景，足够没辨识度，而且无序可读，长度足够。
* 可以用于无纯数字要求、无序自增、无可读性要求的场景。

### 4.2 基于雪花算法（Snowflake） 18位

不安全：订单号有顺序，导致数据不安全。

国内开源的基于snowflake算法思想实现的一款分布式ID生成器，snowflake雪花算法是twitter公司内部分布式项目采用的ID生成算法，现在开源并流行了起来，下面是Snowflake算法的ID构成图。



这种方案巧妙地把64位分别划分成多段，分开表示时间戳差值、机器标识和随机序列，先以此生成一个64位地二进制正整数，然后再转换成十进制进行存储。

* + 1位标识符 不使用且记为零
  + 41位时间戳 用来存储时间戳的差值
  + 10 位机器码 共可标记1024个机器节点 也可拆分为5位表示机房，5位表示机器ID 共有32 \* 32 中组合
  + 最后的12位随即序列用来记录毫秒内的计数，一个节点就能够生成4096个ID序号

综述：理论上Snowflake算法方案的QPS大约为409.6w/s，性能足够强悍了，而且这种方式，能够确保集群中每个节点生成的ID都是不同的，且区间内递增。

* 优点：
* 每秒能够生成百万个不同的ID，性能佳。
* 时间戳值在高位，中间是固定的机器码，自增的序列在地位，整个ID是趋势递增的。
* 能够根据业务场景数据库节点布置灵活挑战bit位划分，灵活度高。
* 缺点
* 强依赖于机器时钟，如果时钟回拨，会导致重复的ID生成，所以一般基于此的算法发现时钟回拨，都会抛异常处理，阻止ID生成，这可能导致服务不可用。
* 防止时钟回拨

因为机器的原因会发生时间回拨，我们的雪花算法是强依赖我们的时间的，如果时间发生回拨，有可能会生成重复的ID，在我们上面的nextId中我们用当前时间和上一次的时间进行判断，如果当前时间小于上一次的时间那么肯定是发生了回拨，普通的算法会直接抛出异常,这里我们可以对其进行优化,一般分为两个情况:

* 如果时间回拨时间较短，比如配置5ms以内，那么可以直接等待一定的时间，让机器的时间追上来。
* 如果时间的回拨时间较长，我们不能接受这么长的阻塞等待，那么又有两个策略:
* 直接拒绝，抛出异常，打日志，通知RD时钟回滚。
* 利用扩展位，上面我们讨论过不同业务场景位数可能用不到那么多，那么我们可以把扩展位数利用起来了，比如当这个时间回拨比较长的时候，我们可以不需要等待，直接在扩展位加1。2位的扩展位允许我们有3次大的时钟回拨，一般来说就够了，如果其超过三次我们还是选择抛出异常，打日志。

### 4.3基于Redis生成办法

Redis的INCR命令能够将key中存储的数字值增一，得益于此操作的原子特性，我们能够巧妙地使用此来做分布式ID地生成方案，还可以配合其他如时间戳值、机器标识等联合使用。

* 优点：
* 有序递增，可读性强。
* 能够满足一定性能。
* 缺点：
* 强依赖于Redis，可能存在单点问题。
* 占用宽带，而且需要考虑网络延时等问题带来地性能冲击。
* 适用场景:
* 对性能要求不是太高，而且规模较小业务较轻的场景，而且Redis的运行情况有一定要求，注意网络问题和单点压力问题，如果是分布式情况，那考虑的问题就更多了，所以一帮情况下这种方式用的比较少。

### 4.3基于美团的leaf方案

从上面的几种分布式ID方案可以看出，能够解决一定问题，但是都有明显缺陷，为此，美团在数据库的方案基础上做了一个优化，提出了一个叫做Leaf-segment的数据库方案。

原方案我们每次获取ID都需要去读取一次数据库，这在高并发和大数据量的情况下很容易造成数据库的压力，那能不能一次性获取一批ID呢，这样就无需频繁的造访数据库了。

Leaf-segment的方案就是采用每次获取一个ID区间段的方式来解决，区间段用完之后再去数据库获取新的号段，这样一来可以大大减轻数据库的压力，那怎么做呢？

|  |
| --- |
| +-------------+--------------+------+-----+-------------------+-----------------------------+  | Field | Type | Null | Key | Default | Extra |  +-------------+--------------+------+-----+-------------------+-----------------------------+  | biz\_tag | varchar(128) | NO | PRI | | |  | max\_id | bigint(20) | NO | | 1 | |  | step | int(11) | NO | | NULL | |  | desc | varchar(256) | YES | | NULL | |  | update\_time | timestamp | NO | | CURRENT\_TIMESTAMP | on update CURRENT\_TIMESTAMP |  +-------------+--------------+------+-----+-------------------+-----------------------------+ |

其中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，后面的desc和update\_time分别表示业务描述和上一次更新号段的时间。原来每次获取ID都要访问数据库，现在只需要把Step设置的足够合理如1000，那么现在可以在1000个ID用完之后再去访问数据库了，看起来真的很酷。

我们现在可以这样设计整个获取分布式ID的流程了：

* 用户服务在注册一个用户时，需要一个用户ID；会请求生成ID服务(是独立的应用)的接口
* 生成ID的服务会去查询数据库，找到user\_tag的id，现在的max\_id为0，step=1000
* 生成ID的服务把max\_id和step返回给用户服务，并且把max\_id更新为max\_id = max\_id + step，即更新为1000
* 用户服务获得max\_id=0，step=1000；
* 这个用户服务可以用[max\_id + 1，max\_id+step]区间的ID，即为[1，1000]
* 用户服务把这个区间保存到jvm中 即内存中
* 用户服务需要用到ID的时候，在区间[1，1000]中依次获取id，可采用AtomicLong中的getAndIncrement方法。
* 如果把区间的值用完了，再去请求生产ID的服务的接口，获取到max\_id为1000，即可以用[max\_id + 1，max\_id+step]区间的ID，即为[1001，2000]

显而易见，这种方式很好的解决了数据库自增的问题，而且可以自定义max\_id的起点，可以自定义步长，非常灵活易于扩容，于此同时，这种方式也很好的解决了数据库压力问题，而且ID号段是存储在JVM中的，性能获得极大的保障，可用性也过得去，即时数据库宕机了，因为JVM缓存的号段，系统也能够因此撑住一段时间。

* 优点：
* 扩张灵活，性能强能够撑起大部分业务场景。
* ID号码是趋势递增的，满足数据库存储和查询性能要求。
* 可用性高，即使ID生成服务器不可用，也能够使得业务在短时间内可用，为排查问题争取时间。
* 可以自定义max\_id的大小，方便业务迁移，方便机器横向扩张
* 缺点：
* ID号码不够随机，完整的顺序递增可能带来安全问题。能够猜出下一个id。
* DB宕机可能导致整个系统不可用，仍然存在这种风险，因为号段只能撑一段时间。
* 可能存在分布式环境各节点同一时间争抢分配ID号段的情况，这可能导致并发问题而出现ID重复生成。
* 缺点解决：
* 采用双Buffer减少了数据库查询，减少了网络依赖，效率更高。