全局唯一id生成策略

一、以前的做法

1）、UUID：性能非常高，本地生成，没有网络消耗

问：为什么无序的UUID会导致入库性能变差呢？

1. 无序，无法预测他的生成顺序，不能生成递增有序的数字。（首先分布式id一般都会作为主键，但是安装mysql官方推荐主键要以是越短越好，UUID 每一个都很长，所以不是很推荐。）
2. 主键，ID作为主键时，在我定的环境会存在一些问题。（比如做BD主键的场景下，UUID就非常不适用MySql，官方有明确的建议主键要以是睵短越好，36个字符长度的UUID不符合要求）
3. 索引，B+树索引分裂（既然分布式id是主键，然后主键是包含索引的，然后Mysqlr索引是通过b+树来实现的，每一次新的UUID数据的插入，为了查询的优化，都会对索引 底层的b+树进行修改，因为UUID数据是无序的，所以每一次UUID数据的插入都会对主键的b+树进行很大的修改，这一点很不好。插入完全无序，不但会导致一些中间节点产生分裂，也会白白创造出很多不饱和的节点，这样大大降低了数据库插入的性能。）

2）、mysql自增id

1、单机mysql：

在分布式里面，数据库的自增id机制的主要原理是：

1. 数据库自增id和mysql数据库的replace into 实现的。
2. 这里的replace into 跟 insert功能类似，不同点在于：replace into 首先尝试插入数据列表中，如果发现表中已经有此行数据(根据 主键或唯一索引判断)则先删除，再插入。否则直接插入新数据。
3. replace into 的含义是插入一条记录，如果表中唯一索引的值 遇到冲突，则替换老数据

2、集群mysql：

问：那数据库自增id机制适合作分布式id吗？ 不太适合

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？假设现在只有一台机器是1、2、3、4、5（步长是1），
2. 这个时候需要扩容机器一，可以这样做：把第二台机器的初始值设置得比第一台超过很多，貌似还好，现在想象一下，如果我们线上有100台机器，这个时候扩容该怎么做？简直是噩梦。所以系统水平扩展方案复杂难以实现。
3. 数据库压力还是很大，每次获取id都得读写一次数据库，非常影响性能，不符合分布式id里面的延迟低和要高QPS的规则(在高并发下，如果都去数据库里面获取id，那是非常影响性能的)

3）、redis生成全局id策略：

1. 因为reids是单线程的，天生保证原子性，可以使用原子操作INCR和INCRBY来实现。（注意：redis 6.0 以后 支持多线程了。可以关注一下。好像是核心还是单线程，只是把网络io那里优化成了多线程）。
2. 注意：在Redis集群情况下，同样和Mysql一样需要设置不同的增长步长，同时key一定要设置有效期。
3. 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。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

缺点：实现麻烦，维护麻烦，为了一个id，还要维护redis集群。

二、snowflake： Twitter的分布式自增id算法

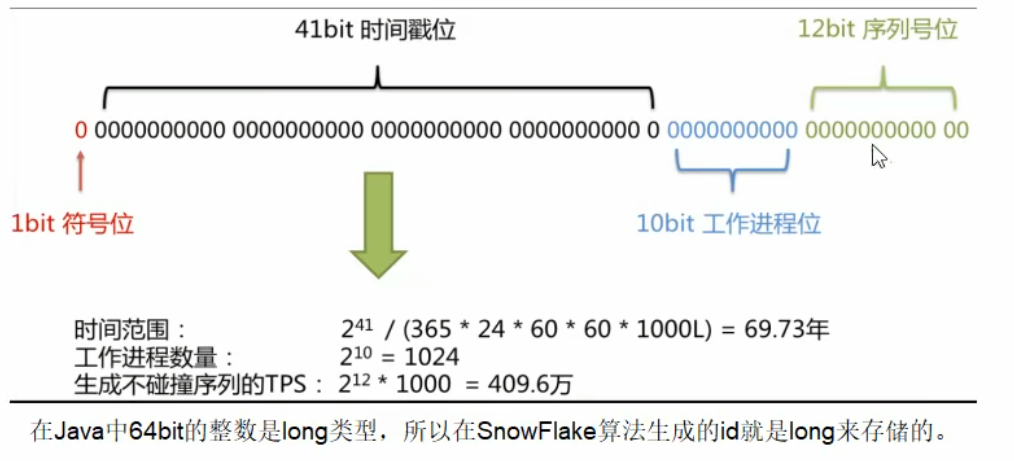
1）Twitter的分布式雪花算法 SnowFlake，经测试 snowflake 每秒能够产生26万个自增可排序的ID

1. twitter的SnowFlake生成ID能够按照时间有序生成
2. SnowFlake 算法生成id的一个64bit大小的整数，为一个Long型(转换成字符串后长度最多19)。
3. 分布式系统内不会产生ID碰撞(由datacenter和workerId作区分) 并且效率较高。

2）、分布式系统中，有一些需要使用全局唯一id的场景，生成id的基本要求：

1. 在分布式的环境下必须全局唯一。
2. 一般都需要单调递增，因为一般唯一id都会存到数据库，而Innodb的特性就是将内容存储在主键索引树上的叶子节点，而且是从左往右，递增的，所以考虑到数据库性能，一般生成的id也最好是单调递增。为了防止ID冲突可以使用36位的UUID，但是UUID有一些缺点，首先他相对比较长，另外UUID一般是无序的。
3. 可能还会需要无规则，因为如果使用唯一ID作为订单号这种，为了不让别人知道一天的订单量是多少，就需要这个规则。

3）、雪花算法的几个核心组成部分：



号段解析：

1）、1bit，不用，因为二进制中最高位是符号位，1表示负数，0表示正数。

- 生成的id一般都是用整数，所以最高位固定为0.

1. 、41bit-时间戳，用来记录时间戳，毫秒级

- 41位可以表示 2^{41} -1 个数字

- 如果只用来表示正整数，（计算机中正数包含0），可以表示的数值范围是：0至2^{41}-1，减1是因为可表示的数值范围是从0开始的。

- 也就是说41位可以表示 2^{41}-1 个毫秒的值，转化成单位年则是（2^{41}-1）/(1000 \* 60 \* 60 \* 24 \* 365) = 69年 （从1970开始算）

1. 、10bit-工作机器id，用来记录工作机器id。

- 可以部署在 2^{10} = 1024个节点，包括5位datacenterid和 5位workerId

- 5位(bit)可以表示的最大正整数是2^{5}-1 = 31,即可以用0、1、2、3、...31这32个数字，来表示不同的datecenterId或workerId。

4）、12bit-序列号，序列号，用来记录同毫秒内产生的不同id。

- 12位(bit) 可以表示的最大正整数是 2^{12}-1 = 4095，即可以用0、1、2、3、...4094这4095个数字，来表示同一机器同一时间戳（毫秒）内产生的4095个ID序列

SnowFlake可以保证：

所有生成的Id按时间趋势递增

整个分布式系统内不会产生重复id（因为有datacenterId和workerId来做区分）

三、落地经验

1、糊涂工具包

[www.github.com/looly/hutool/](http://www.github.com/looly/hutool/) hutool-captcha就包含雪花算法

四、优缺点

优点：

1. 毫秒数在高位，自增序列在低位，整个ID都是趋势递增的
2. 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。
3. 可以根据自身业务特性分配bit位，非常灵活。

缺点：

1. 依赖机器时钟，如果机器时钟回拨，会导致重复ID生成
2. 在单机 上是递增的，但是由于设计到分布式环境，每台机器上的时钟不可能完全同步，有时候会出现不是全局递增的情况。（此缺点可以认为无所谓，一般分布式ID只要求趋势递增，并不会严格要求递增，90%的需求都只要求趋势递增）