九大分布式ID生成策略

- UUID
- 数据库自增ID
- 数据库多主模式
- 号段模式
- Redis
- 雪花算法（SnowFlake）
- 百度（Uidgenerator）
- 美团（Leaf）
- 滴滴出品（TinyID）

生成策略概述

UUID可以做分布式ID, 但是不推荐；长度 16 字节128位，36位16进位数字长度的字符串
优点：生成简单，本地生成无网络消耗，具有唯一性
缺点：
 1. 没有具体的业务含义
 2. 无序的字符串，不具备趋势自增特性
 3. 存储以及查询对MySQL的性能消耗较大, 不推荐作为主键ID, 可以作为逻辑ID

格式：8-4-4-4-12
　　917c5dd2-3bab-18c7-ad21-8369704bd120
　　xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
　　四位数字 M表示 UUID 版本，数字 N的一至三个最高有效位表示 UUID 变体。在例子中，M 是 1, 而且 N 是 a（10xx），这意味着此 UUID 是“变体1”、 “版本1” UUID；即基于时间的 DCE/RFC 4122 UUID。

编码规则：
 1）1~8位采用系统时间，在系统时间上精确到毫秒级保证时间上的惟一性；
 2）9~16位采用底层的IP地址，在服务器集群中的惟一性；
 3）17~24位采用当前对象的HashCode值，在一个内部对象上的惟一性；
 4）25~32位采用调用方法的一个随机数，在一个对象内的毫秒级的惟一性。
 通过以上4种策略可以保证惟一性。在系统中需要用到随机数的地方都可以考虑采用UUID算法

UUID五个版本：
 版本由 M 字符串中指示。
 版本1 - UUID 是根据时间和节点 ID（通常是MAC地址）生成；
 版本2 - UUID 是根据标识符（通常是组或用户ID）、时间和节点ID生成；
 版本3、版本5 - 确定性UUID 通过散列（hashing）名字空间（namespace）标识符和名称生成；
 版本4 - UUID 使用随机性或伪随机性生成。
 基于时间的UUID
　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
　基于时间的UUID通过计算当前时间戳、随机数和机器MAC地址得到。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，使用MAC地址会带来安全性问题，这就是这个版本UUID受到批评的地方。如果应用只是在局域网中使用，也可以使用退化的算法，以IP地址来代替MAC地址 - - Java的UUID往往是这样实现的（当然也考虑了获取MAC的难度）。

 DCE安全的UUID
　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
　DCE（Distributed Computing Environment）安全的UUID和基于时间的UUID算法相同，但会把时间戳的前4位置换为POSIX的UID或GID。这个版本的UUID在实际中较少用到。

 基于名字的UUID（MD5）
　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
　基于名字的UUID通过计算名字和名字空间的MD5散列值得到。这个版本的UUID保证了：相同名字空间中不同名字生成的UUID的唯一性；不同名字空间中的UUID的唯一性；相同名字空间中相同名字的UUID重复生成是相同的。

 随机UUID
　优点：实现简单
　缺点：重复几率可计算
　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。

 基于名字的UUID（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
　和基于名字的UUID算法类似，只是散列值计算使用SHA1（Secure Hash Algorithm 1）算法。

```java
/**
 * @ClassName: UUID_Id
 * @Author: 99847
 * @Description:
 * @Date: 2021/6/19 18:00
 * @Version: 1.0
 */
public class UUID_Id {
    public static void main(String[] args) {


        // 版本4，使用随机性或伪随机性生成。
        String id = UUID.randomUUID().toString();
        System.out.println(id);
        System.out.println(id.replace("-", ""));

        // 版本3，使用名字空间
        System.out.println( UUID.nameUUIDFromBytes(new byte[] {12,123,12}).toString());
    }
}
```

基于数据库的auto_increment自增ID完全可以充当分布式ID，需要一个单独的MySQL实例用来生成ID
当我们需要一个ID的时候，向表中插入一条记录返回主键ID，但这种方式有一个比较致命的缺点，访问量激增时MySQL本身就是系统的瓶颈，用它来实现分布式服务风险比较大，不推荐！
优点：实现简单, ID单调自增, 天然有序, 数值类型查询速度快
缺点：
 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

单点数据库方式存在很明显的性能问题，可以对数据库进行高可用优化，搭建数据库集群模式; 担心一个主节点挂掉没法用，可以选择做双主模式集群，也就是两个Mysql实例都能单独的生产自增ID。
解决方案： 数据库水平拆分，设置不同的初始值和相同的步长(解决自增ID都从1开始)
优点：解决DB单点问题
缺点：不利于后续扩容，而且实际上单个数据库自身压力还是大，依旧无法满足高并发场景。

要使用单台机器做ID生成，避免固定步长带来的扩容问题，可以每次从数据库取出一个号段范围，例如 (1,1000] 代表1000个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分之一，且故障后可坚持一段时间。
缺点：服务器重启、单点故障会造成ID不连续

利用Redis的 incr 命令实现ID的原子性自增。
注意要考虑redis持久化的问题，AOF可以实现对每条写命令进行持久化，即使Redis挂掉了也不会出现ID重复的情况。由于incr命令的特殊性，会导致Redis重启恢复的数据时间过长。

定义一个64bit的数，对指定机器 & 同一时刻 & 某一并发序列，是唯一的，其极限QPS约为400w/s。
格式： 1-41-10-12
　　符号位,不用 - 时间戳(最长69年) - 机器id - 序列号
含义：将64 bit分为了四部分。其中时间戳有时间上限（69年）。机器id只有10位，能记录1024台机器，常用前几位表示数据中心id，后几位表示数据中心内的机器id。序列号用来对同一个毫秒之内的操作产生不同的ID，最多4095个。

备注：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，根据自身业务的并发情况、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的量级。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。

缺点：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冲突的问题
解决方法：
 1）将ID生成交给少量服务器，并关闭时钟同步。
 2）直接报错，交给上层业务处理。
 3）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。
 4）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，将回拨位加1，可得到不一样的ID，2位回拨位允许标记三次时钟回拨，基本够使用。如果超出了，可以再选择抛出异常。
 5）ID生成有规律性,信息容易泄漏

```java
package cn.xlystar.distributeId;

/**
 * @ClassName: SnowFlakeShortUrl
 * @Author: 99847
 * @Description: 雪花算法
 * Snowflake ID组成结构：
 * 正数位（占1比特）+ 时间戳（占41比特）+ 机器ID（占5比特）+ 数据中心（占5比特）+ 自增值（占12比特），
 * 总共64比特组成的一个Long类型。
 * @Date: 2021/6/19 23:41
 * @Version: 1.0
 */
public class SnowFlakeShortUrl {

    /**
     * 起始的时间戳
     */
    private final static long START_TIMESTAMP = 1480166465631L;

    /**
     * 每一部分占用的位数
     */
    private final static long SEQUENCE_BIT = 12;   //序列号占用的位数
    private final static long MACHINE_BIT = 5;     //机器标识占用的位数
    private final static long DATA_CENTER_BIT = 5; //数据中心占用的位数

    /**
     * 每一部分的最大值
     */
    private final static long MAX_SEQUENCE = -1L ^ (-1L << SEQUENCE_BIT);
    private final static long MAX_MACHINE_NUM = -1L ^ (-1L << MACHINE_BIT);
    private final static long MAX_DATA_CENTER_NUM = -1L ^ (-1L << DATA_CENTER_BIT);

    /**
     * 每一部分向左的位移
     */
    private final static long MACHINE_LEFT = SEQUENCE_BIT;
    private final static long DATA_CENTER_LEFT = SEQUENCE_BIT + MACHINE_BIT;
    private final static long TIMESTAMP_LEFT = DATA_CENTER_LEFT + DATA_CENTER_BIT;

    /**
     * 数据中心
     */
    private long dataCenterId;
    /**
     * 机器标识
     */
    private long machineId;
    /**
     * 序列号
     */
    private long sequence = 0L;
    /**
     * 上一次时间戳
     */
    private long lastTimeStamp = -1L;

    private long getNextMill() {
        long mill = getNewTimeStamp();
        while (mill <= lastTimeStamp) {
            mill = getNewTimeStamp();
        }
        return mill;
    }

    /**
     * 当前毫秒级时间戳
     */
    private long getNewTimeStamp() {
        return System.currentTimeMillis();
    }

    /**
     * 根据指定的数据中心ID和机器标志ID生成指定的序列号
     *
     * @param dataCenterId 数据中心ID
     * @param machineId    机器标志ID
     */
    public SnowFlakeShortUrl(long dataCenterId, long machineId) {
        if (dataCenterId > MAX_DATA_CENTER_NUM || dataCenterId < 0) {
            throw new IllegalArgumentException("DtaCenterId can't be greater than MAX_DATA_CENTER_NUM or
less than 0！");
        }
        if (machineId > MAX_MACHINE_NUM || machineId < 0) {
            throw new IllegalArgumentException("MachineId can't be greater than MAX_MACHINE_NUM or less than
0！");
        }
        this.dataCenterId = dataCenterId;
        this.machineId = machineId;
    }

    /**
     * 产生下一个ID
     *
     * @return
     */
    public synchronized long nextId() {
        long currTimeStamp = getNewTimeStamp();
        if (currTimeStamp < lastTimeStamp) {
            throw new RuntimeException("Clock moved backwards.  Refusing to generate id");
        }

        if (currTimeStamp == lastTimeStamp) {
            //相同毫秒内，序列号自增
            sequence = (sequence + 1) & MAX_SEQUENCE;
            //同一毫秒的序列数已经达到最大
            if (sequence == 0L) {
                currTimeStamp = getNextMill();
            }
        } else {
            //不同毫秒内，序列号置为0
            sequence = 0L;
        }

        lastTimeStamp = currTimeStamp;

        return (currTimeStamp - START_TIMESTAMP) << TIMESTAMP_LEFT //时间戳部分
                | dataCenterId << DATA_CENTER_LEFT       //数据中心部分
                | machineId << MACHINE_LEFT             //机器标识部分
                | sequence;                             //序列号部分
    }

    public static void main(String[] args) {
        SnowFlakeShortUrl snowFlake = new SnowFlakeShortUrl(2, 3);

        for (int i = 0; i < (1 << 4); i++) {
            //10进制
            System.out.println(snowFlake.nextId());
        }
    }
}
```

uid-generator是基于Snowflake算法实现的，与原始的snowflake算法不同在于，uid-generator支持自定义时间戳、工作机器ID和 序列号 等各部分的位数，而且uid-generator中采用用户自定义workId的生成策略。uid-generator需要与数据库配合使用，需要新增一个WORKER_NODE表。当应用启动时会向数据库表中去插入一条数据，插入成功后返回的自增ID就是该机器的workId数据由host，port组成。
uid-generator ID组成结构：
workId占用了22个bit位，时间占用了28个bit位，序列化占用了13个bit位，需要注意的是，和原始的snowflake不太一样，时间的单位是秒，而不是毫秒，workId也不一样，而且同一应用每次重启就会消费一个workId。
默认分配方式如下：
 sign(1bit)：固定1bit符号标识，即生成的UID为正数。
 delta seconds (28 bits)：当前时间，相对于时间基点"2016-05-20"的增量值，单位：秒，最多可支持约8.7年（注意：1. 这里的单位是秒，而不是毫秒！ 2.注意这里的用词，是“最多”可支持8.7年，为什么是“最多” ，后面会讲）
 worker id (22 bits)：机器id，最多可支持约420w次机器启动。内置实现为在启动时由数据库分配，默认分配策略为用后即弃，后续可提供复用策略。
 sequence (13 bits)：每秒下的并发序列，13 bits可支持每秒8192个并发。（注意下这个地方，默认支持qps最大为8192个）
由百度技术部开发，开源项目链接：https://github.com/baidu/uid-generator

由美团开发，开源项目链接：https://github.com/Meituan-Dianping/Leaf
Leaf同时支持号段模式和snowflake算法模式，可以切换使用。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。
Leaf的snowflake模式依赖于ZooKeeper，不同于原始snowflake算法也主要是在workId的生成上，Leaf中workId是基于ZooKeeper的顺序Id来生成的，每个应用在使用Leaf-snowflake时，启动时都会都在Zookeeper中生成一个顺序Id，相当于一台机器对应一个顺序节点，也就是一个workId。
Leaf的号段模式是对直接用数据库自增ID充当分布式ID的一种优化，减少对数据库的频率操作。相当于从数据库批量的获取自增ID，每次从数据库取出一个号段范围，例如 (1,1000] 代表1000个ID，业务服务将号段在本地生成1~1000的自增ID并加载到内存.。
特性：
 1）全局唯一，绝对不会出现重复的ID，且ID整体趋势递增。
 2）高可用，服务完全基于分布式架构，即使MySQL宕机，也能容忍一段时间的数据库不可用。
 3）高并发低延时，在CentOS 4C8G的虚拟机上，远程调用QPS可达5W+，TP99在1ms内。
 4）接入简单，直接通过公司RPC服务或者HTTP调用即可接入。
Leaf采用双buffer的方式，它的服务内部有两个号段缓存区segment。当前号段已消耗10%时，还没能拿到下一个号段，则会另启一个更新线程去更新下一个号段。

简而言之就是Leaf保证了总是会多缓存两个号段，即便哪一时刻数据库挂了，也会保证发号服务可以正常工作一段时间。

由滴滴开发，开源项目链接：https://github.com/didi/tinyid
Tinyid是在美团（Leaf）的leaf-segment算法基础上升级而来，不仅支持了数据库多主节点模式，还提供了tinyid-client客户端的接入方式，使用起来更加方便。但和美团（Leaf）不同的是，Tinyid只支持号段一种模式不支持雪花模式。Tinyid提供了两种调用方式，一种基于Tinyid-server提供的http方式，另一种Tinyid-client客户端方式。每个服务获取一个号段（1000,2000]、（2000,3000]、（3000,4000]
特性：
 1）全局唯一的long型ID
 2）趋势递增的id
 3）提供 http 和 java-client 方式接入
 4）支持批量获取ID
 5）支持生成1,3,5,7,9...序列的ID
 6）支持多个db的配置
适用场景：只关心ID是数字，趋势递增的系统，可以容忍ID不连续，可以容忍ID的浪费
不适用场景：像类似于订单ID的业务，因生成的ID大部分是连续的，容易被扫库、或者推算出订单量等信息

总结

可以发现，常用的分布式唯一ID生成思路基本是利用一个长串数字或字符串，将其分割成多个部分，分别记录时间信息、机器/名字信息、随机信息、序列信息等。时间信息部分决定了该策略能使用的时长，机器/名字信息支持了分布式环境下的独自生成唯一ID与识别能力，序列信息保证了事件的顺序记录以及同一时间单位下的并发数，而随机信息则加大了ID整体的不可识别性。

实际上如果现有的方法依然不能满足，我们完全可以依据自身业务和发展需求，来自行决定使用何种策略生成唯一ID。各种方案都有其优缺点，技术的使用没有绝对的好坏之分，主要在于是否适合使用场景：
 1）要求生成全局唯一且不会重复ID，不关心顺序 —— 使用基于时间的UUID（如游戏聊天室中不同用户的身份ID）
 2）要求生成唯一ID，具有名称不可变性，可重复生成 —— 使用基于名称哈希的UUID（如基于不可变信息生成的用户ID，若不小心删除，仍可根据信息重新生成同一ID）
 3）要求生成有序且自然增长的ID —— 使用数据库自增ID（如各业务操作流水ID，高并发下可参考优化方案）
 4）要求生成数值型无序定长ID —— 使用雪花算法（如对存储空间、查询效率、传输数据量等有较高要求的场景）