

# 几种分布式ID解决方案，总有一款适合你！

IT码徒 2023-04-27 22:09 Posted on 河南

点击“IT码徒”，关注，置顶公众号  
每日技术干货，第一时间送达！



Scan to Follow

耗时8个月联合打造 《2023年Java高薪课程》，已更新了 102G 视频，累计更新时长 500+ 个小时，需要的小伙伴可以了解下，一次购买，持续更新，无需2次付费。

## 背景

在复杂的分布式系统中，往往需要对大量的数据进行唯一标识，比如在对一个订单表进行了分库分表操作，这时候数据库的自增ID显然不能作为某个订单的唯一标识。除此之外还有其他分布式场景对分布式ID的一些要求：

- **趋势递增：** 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。
- **单调递增：** 保证下一个ID一定大于上一个ID，例如排序需求。
- **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；如果是订单号就更危险了，可以直接知道我们的单量。所以在一些应用场景下，会需要ID无规则、不规则。

就不同的场景及要求，市面诞生了很多分布式ID解决方案。本文针对多个分布式ID解决方案进行介绍，包括其优缺点、使用场景及代码示例。

## 1、UUID

UUID( **Universally Unique Identifier** )是基于当前时间、计数器（counter）和硬件标识（通常为无线网卡的MAC地址）等数据计算生成的。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，可以生成全球唯一的编码并且性能高效。

JDK提供了UUID生成工具，代码如下：

```
import java.util.UUID;

public class Test {
    public static void main(String[] args) {
        System.out.println(UUID.randomUUID());
    }
}
```

输出如下

b0378f6a-eeb7-4779-bffe-2a9f3bc76380

UUID完全可以满足分布式唯一标识，但是在实际应用过程中一般不采用，有如下几个原因：

- **存储成本高：** 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地址生成的UUID算法会暴露MAC地址，曾经梅丽莎病毒的制造者就是根据UUID寻找的。
- **不符合MySQL主键要求：** MySQL官方有明确的建议主键要尽量越短越好，因为太长对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。

## 2、数据库自增ID

利用Mysql的特性ID自增，可以达到数据唯一标识，但是分库分表后只能保证一个表中的ID的唯一，而不能保证整体的ID唯一。为了避免这种情况，我们有以下两种方式解决该问题。

### 2.1、主键表

通过单独创建主键表维护唯一标识，作为ID的输出源可以保证整体ID的唯一。举个例子：

创建一个主键表

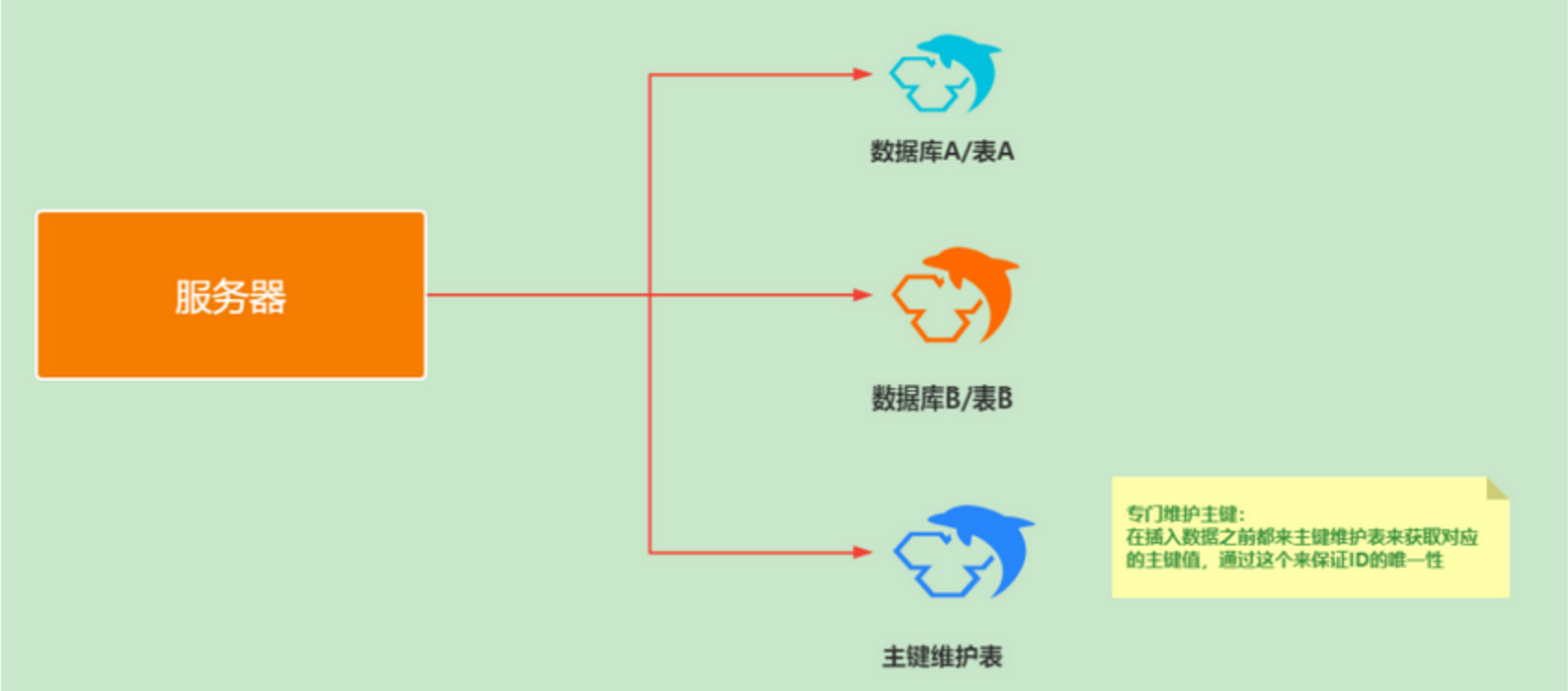
```
CREATE TABLE `unique_id` (
  `id` bigint NOT NULL AUTO_INCREMENT,
  `biz` char(1) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `biz` (`biz`)
) ENGINE = InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET =utf8;
```

业务通过更新操作来获取ID信息，然后添加到某个分表中。

```
BEGIN;

REPLACE INTO unique_id (biz) values ('o') ;
SELECT LAST_INSERT_ID();

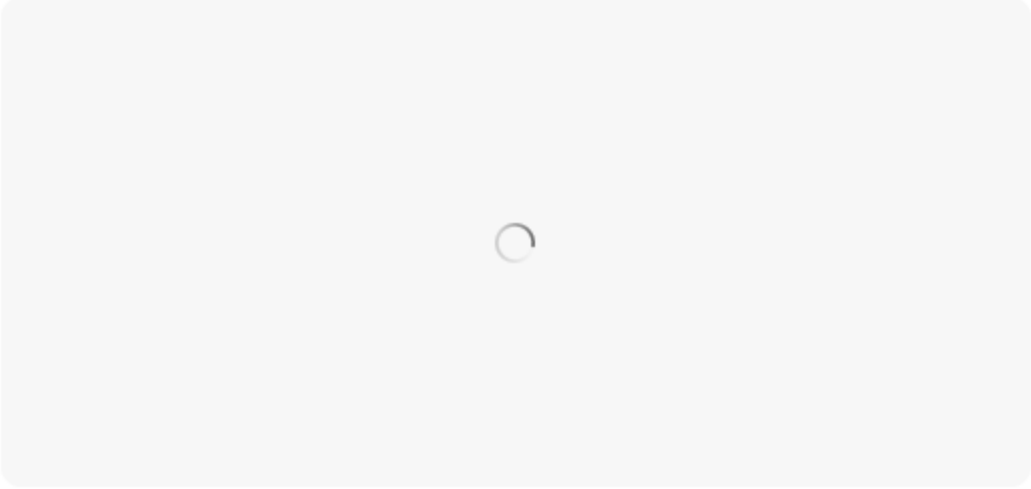
COMMIT;
```



## 2.2、ID自增步长设置

我们可以设置Mysql主键自增步长，让分布在不同实例的表数据ID做到不重复，保证整体的唯一。

如下，可以设置Mysql实例1步长为1，实例1步长为2。



查看主键自增的属性

```
show variables like '%increment%'
```

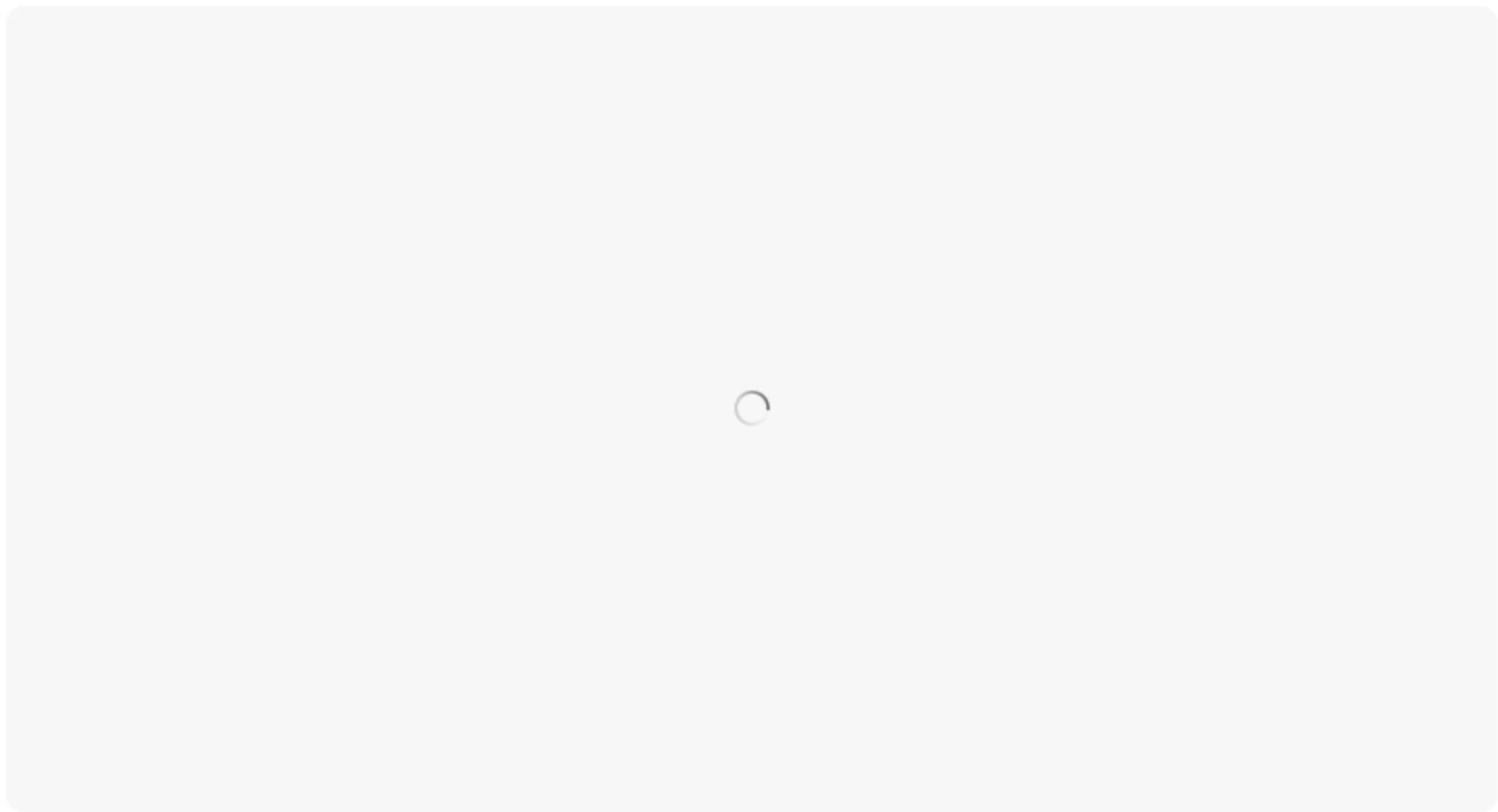
信息	结果 1	剖析	状态
Variable_name	Value		
auto_increment_incr	1		
auto_increment_offs	1		
div_precision_increment	4		
innodb_autoextend_	64		

显然，这种方式在并发量比较高的情况下，如何保证扩展性其实会是一个问题。

## 3、号段模式

号段模式是当下分布式ID生成器的主流实现方式之一。其原理如下：

- 号段模式每次从数据库取出一个号段范围，加载到服务内存中。业务获取时ID直接在这个范围递增取值即可。
- 等这批号段ID用完，再次向数据库申请新号段，对max\_id字段做一次update操作，新的号段范围是( max\_id , max\_id +step ]。
- 由于多业务端可能同时操作，所以采用版本号version乐观锁方式更新。



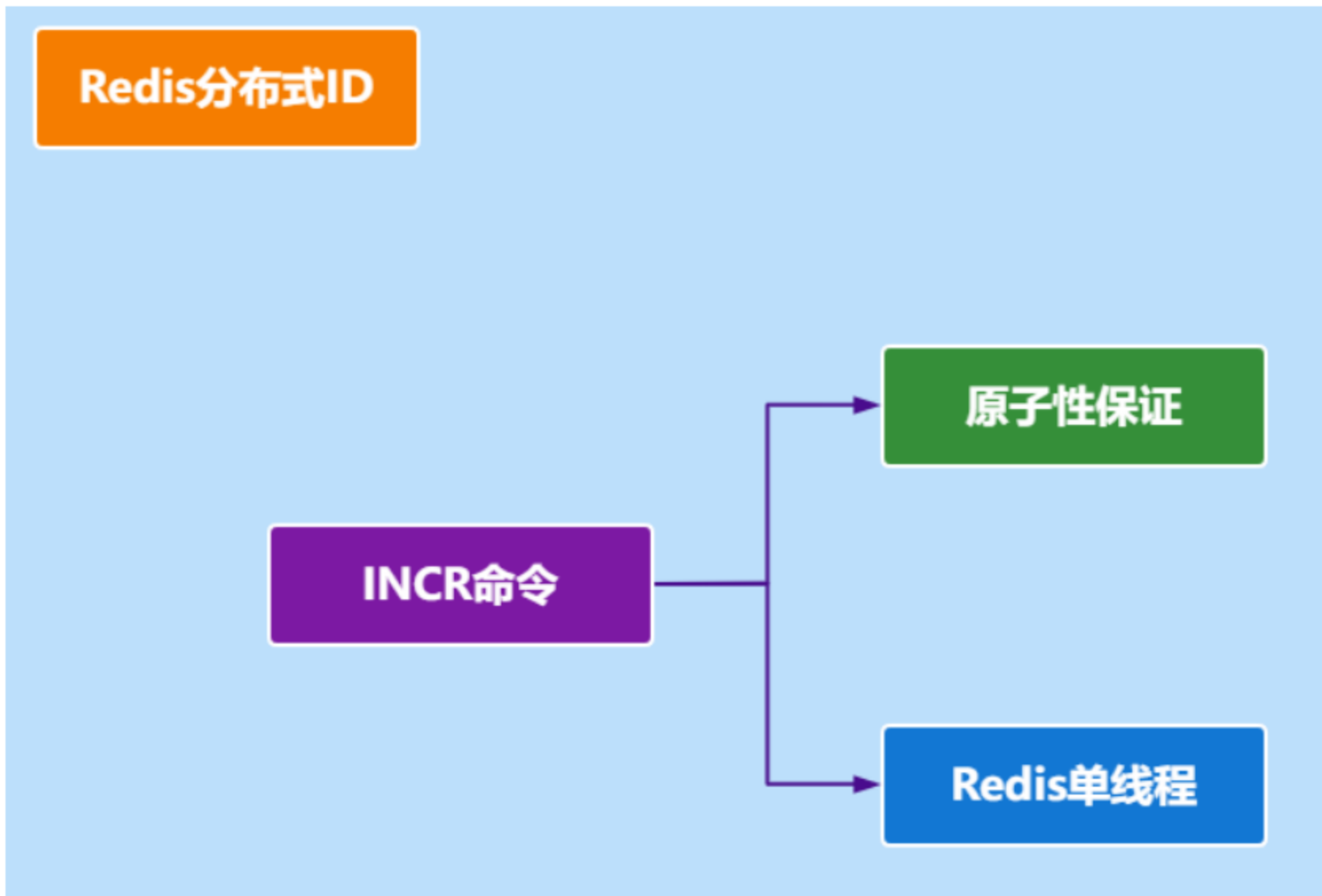
例如 (1,1000] 代表1000个ID，具体的业务服务将本号段生成1~1000的自增ID。表结构如下：

```
CREATE TABLE id_generator (
  id int(10) NOT NULL,
  max_id bigint(20) NOT NULL COMMENT '当前最大id',
  step int(20) NOT NULL COMMENT '号段的长度',
  biz_type int(20) NOT NULL COMMENT '业务类型',
  version int(20) NOT NULL COMMENT '版本号,是一个乐观锁,每次都更新version, 保证并发时数据的正确性',
  PRIMARY KEY (`id`)
)
```

这种分布式ID生成方式不强依赖于数据库，不会频繁地访问数据库，对数据库的压力小很多。但同样也会存在一些缺点比如：服务器重启，单点故障会造成ID不连续。

#### 4、Redis INCR

基于全局唯一ID的特性，我们可以通过Redis的INCR命令来生成全局唯一ID。



Redis分布式ID的简单案例

```
/**
 * Redis 分布式ID生成器
 */
@Component
public class RedisDistributedId {

    @Autowired
    private StringRedisTemplate redisTemplate;

    private static final long BEGIN_TIMESTAMP = 1659312000L;

    /**
     * 生成分布式ID
     * 符号位  时间戳[31位]  自增序号【32位】
     * @param item
     * @return
     */
    public long nextId(String item){
        // 1.生成时间戳
        LocalDateTime now = LocalDateTime.now();
        // 格林威治时间差
        long nowSecond = now.toEpochSecond(ZoneOffset.UTC);
        // 我们需要获取的 时间戳 信息
        long timestamp = nowSecond - BEGIN_TIMESTAMP;

        // 2.生成序号 --》 从Redis中获取
        // 当前当前的日期
        String date = now.format(DateTimeFormatter.ofPattern("yyyy:MM:dd"));
        // 获取对应的自增的序号
```

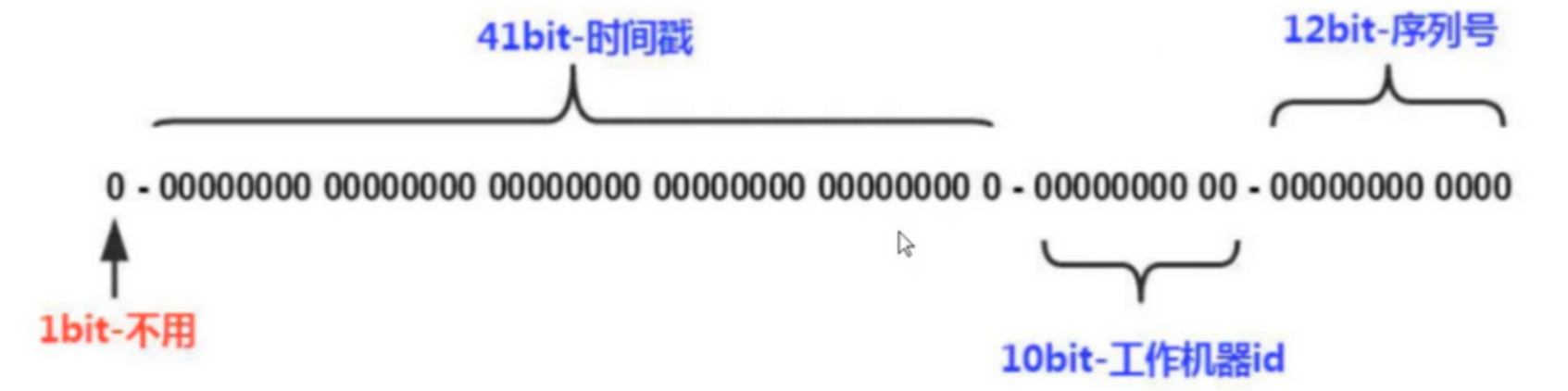


```
        Long increment = redisTemplate.opsForValue().increment("id:" + item + ":" + date);
        return timestamp << 32 | increment;
    }
}
```

同样使用Redis也有对应的缺点：ID 生成的持久化问题，如果Redis宕机了怎么进行恢复？

## 5、雪花算法

Snowflake，雪花算法是有Twitter开源的分布式ID生成算法，以划分命名空间的方式将64bit位分割成了多个部分，每个部分都有具体的不同含义，在Java中64Bit位的整数是Long类型，所以在Java中Snowflake算法生成的ID就是long来存储的。具体如下：



- **第一部分：** 占用1bit，第一位为符号位，不适用
- **第二部分：** 41位的时间戳，41bit位可以表示241个数，每个数代表的是毫秒，那么雪花算法的时间年限是  $(241)/(1000 \times 60 \times 60 \times 24 \times 365) = 69$  年
- **第三部分：** 10bit表示是机器数，即  $2^{10} = 1024$  台机器，通常不会部署这么多机器
- **第四部分：** 12bit位是自增序列，可以表示  $2^{12} = 4096$  个数，一秒内可以生成4096个ID，理论上snowflake方案的QPS约为 409.6w/s

雪花算法案例代码：

```
public class SnowflakeIdWorker {

    // =====Fields=====
    /**
     * 开始时间戳 (2020-11-03，一旦确定不可更改，否则时间被回调，或者改变，可能会造成id重复或冲突)
     */
    private final long twepoch = 1604374294980L;

    /**
     * 机器id所占的位数
     */
    private final long workerIdBits = 5L;

    /**
     * 数据标识id所占的位数
     */
    private final long datacenterIdBits = 5L;

    /**
     * 支持的最大机器id，结果是31（这个移位算法可以很快的计算出几位二进制数所能表示的最大十进制数）
     */
    private final long maxWorkerId = -1L ^ (-1L << workerIdBits);

    /**
     * 支持的最大数据标识id，结果是31
     */
    private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);

    /**
     * 序列在id中占的位数
     */
    private final long sequenceBits = 12L;

    /**
     * 机器ID向左移12位
     */
    private final long workerIdShift = sequenceBits;

    /**
     * 数据标识id向左移17位(12+5)
     */
    private final long datacenterIdShift = sequenceBits + workerIdBits;

    /**
     * 时间截向左移22位(5+5+12)
     */
    private final long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

    /**
     * 生成序列的掩码，这里为4095 (0b111111111111=0xfff=4095)
     */
}
```

```
private final long sequenceMask = -1L ^ (-1L < sequenceBits);

/**
 * 工作机器ID(0~31)
 */
private long workerId;

/**
 * 数据中心ID(0~31)
 */
private long datacenterId;

/**
 * 毫秒内序列(0~4095)
 */
private long sequence = 0L;

/**
 * 上次生成ID的时间戳
 */
private long lastTimestamp = -1L;

//=====Constructors=====

/**
 * 构造函数
 *
 */
public SnowflakeIdWorker() {
    this.workerId = 0L;
    this.datacenterId = 0L;
}

/**
 * 构造函数
 *
 * @param workerId 工作ID (0~31)
 * @param datacenterId 数据中心ID (0~31)
 */
public SnowflakeIdWorker(long workerId, long datacenterId) {
    if (workerId > maxWorkerId || workerId < 0) {
        throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0", maxWorkerId));
    }
    if (datacenterId > maxDatacenterId || datacenterId < 0) {
        throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0", maxDatacenterId));
    }
    this.workerId = workerId;
    this.datacenterId = datacenterId;
}

// =====Methods=====

/**
 * 获得下一个ID (该方法是线程安全的)
 *
 * @return SnowflakeId
 */
public synchronized long nextId() {
    long timestamp = timeGen();

    //如果当前时间小于上一次ID生成的时间戳，说明系统时钟回退过这个时候应当抛出异常
    if (timestamp < lastTimestamp) {
        throw new RuntimeException(
            String.format("Clock moved backwards. Refusing to generate id for %d milliseconds",
                lastTimestamp - timestamp));
    }

    //如果是同一时间生成的，则进行毫秒内序列
    if (lastTimestamp == timestamp) {
        sequence = (sequence + 1) & sequenceMask;
        //毫秒内序列溢出
        if (sequence == 0) {
            //阻塞到下一个毫秒, 获得新的时间戳
            timestamp = tilNextMillis(lastTimestamp);
        }
    }
    //时间戳改变，毫秒内序列重置
    else {
        sequence = 0L;
    }

    //上次生成ID的时间戳
    lastTimestamp = timestamp;

    //移位并通过或运算拼到一起组成64位的ID
    return ((timestamp - twepoch) << timestampLeftShift) //
        | (datacenterId << datacenterIdShift) //
        | (workerId << workerIdShift) //
        | sequence;
}
```

```
/**
 * 阻塞到下一个毫秒，直到获得新的时间戳
 *
 * @param lastTimestamp 上次生成ID的时间戳
 * @return 当前时间戳
 */
protected long tilNextMillis(long lastTimestamp) {
    long timestamp = timeGen();
    while (timestamp <= lastTimestamp) {
        timestamp = timeGen();
    }
    return timestamp;
}

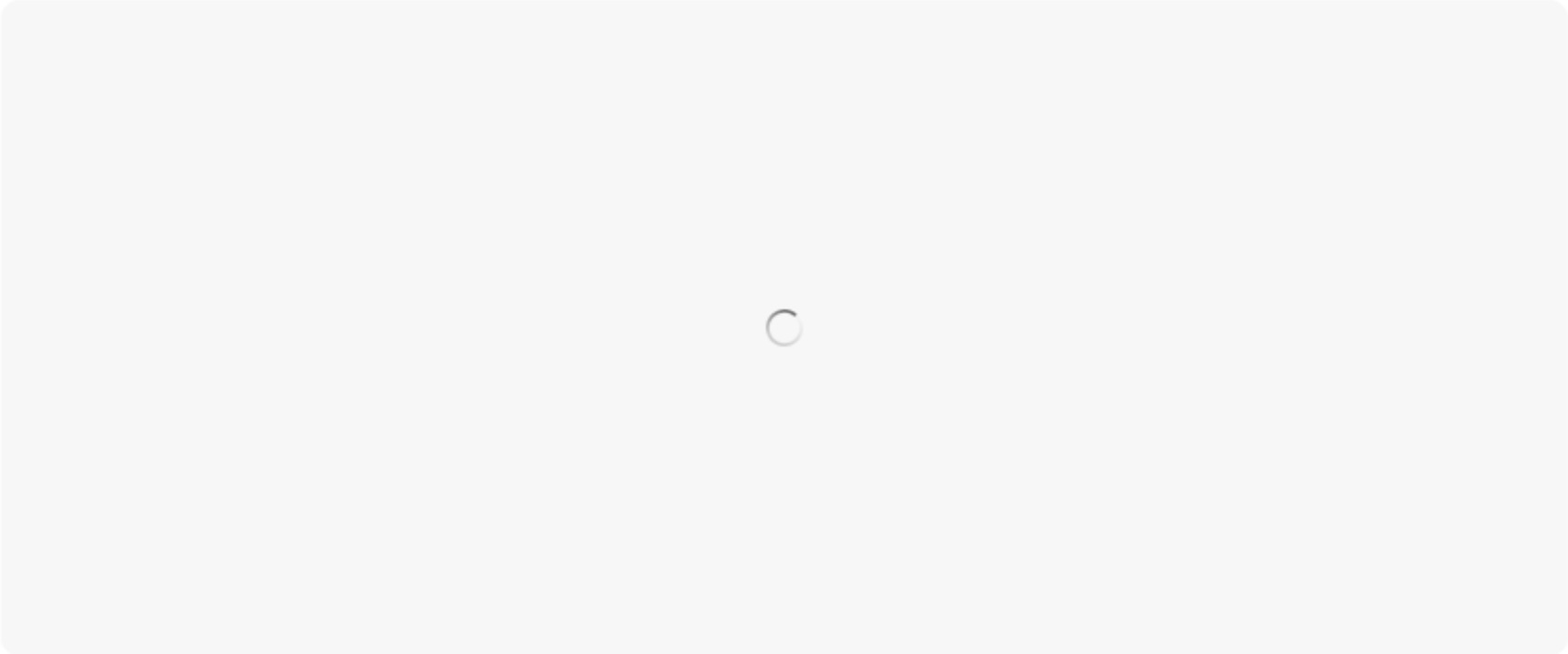
/**
 * 返回以毫秒为单位的当前时间
 *
 * @return 当前时间(毫秒)
 */
protected long timeGen() {
    return System.currentTimeMillis();
}

/**
 * 随机id生成，使用雪花算法
 *
 * @return
 */
public static String getSnowId() {
    SnowflakeIdWorker sf = new SnowflakeIdWorker();
    String id = String.valueOf(sf.nextId());
    return id;
}

//=====Test=====

/**
 * 测试
 */
public static void main(String[] args) {
    SnowflakeIdWorker idWorker = new SnowflakeIdWorker(0, 0);
    for (int i = 0; i < 1000; i++) {
        long id = idWorker.nextId();
        System.out.println(id);
    }
}
```

雪花算法强依赖机器时钟，如果机器上时钟回拨，会导致发号重复。通常通过记录最后使用时间处理该问题。



## 6、美团(Leaf)

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。

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。

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。

## 7、百度(Uidgenerator)

源码地址：

- <https://github.com/baidu/uid-generator>

中文文档地址：

- [https://github.com/baidu/uid-generator/blob/master/README.zh\\_cn.md](https://github.com/baidu/uid-generator/blob/master/README.zh_cn.md)

UidGenerator是百度开源的Java语言实现，基于Snowflake算法的唯一ID生成器。它是分布式的，并克服了雪花算法的并发限制。单个实例的QPS能超过6000000。需要的环境：JDK8+，MySQL（用于分配WorkerId）。

百度的Uidgenerator对结构做了部分的调整，具体如下：

标志位SIGN	时间戳	机器ID	自增序号
1bits	28bits	22bits	13bits

时间部分只有28位，这就意味着UidGenerator默认只能承受8.5年（ $2^{28}-1/86400/365$ ），不过UidGenerator可以适当调整delta seconds、worker node id和sequence占用位数。

## 8、滴滴(TinyID)

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。

总结比较


	Redis INCR	雪花算法	号段模式	Leaf、Uidgenerator、TinyID
	性能优于数据库、ID有序	不依赖数据库等第三方系统，性能也是非高、可以根据自身业务特性分配bit位，非常灵活	数据库的压力小	高性能、高可用、接入简单
	解决单点问题带来的数据一致性等问题使得复杂度提高	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。	单点故障ID不连续	依赖第三方组件如ZooKeeper、Mysql

作者：叫我二蛋  
来源：[wangbinguang.blog.csdn.net/article/details/129201971](https://wangbinguang.blog.csdn.net/article/details/129201971)

— END —

[【福利】2023 高薪课程，全面来袭（视频+笔记+源码）](#)

PS：防止找不到本篇文章，可以收藏点赞，方便翻阅查找哦



IT码徒

专注Java技术栈分享，多线程，JVM，io流，Spring，微服务，数据库等以及开源项目，视...

公众号

往期推荐

多线程如何实现事务回滚？一招帮你搞定！

和 XShell 说再见，这款 SSH 工具足够惊艳，还支持网页版！

不愧是神级Java框架！

真神！我又搞到一个Java神器

号称 Redis Plus，来看看 KeyDB 性能有多炸裂！

AutoGPT 爆火，两周斩获 50k+ Star：无需人类插手，自主完成任务！

还在用@Autowired和@Resource？试试@RequiredArgsConstructor吧！



[Read more](#)

People who liked this content also liked

一次src中的小技巧分享  
深夜笔记本

X



炸裂的 AutoGPT，帮我做了个网站！  
程序员鱼皮

X



Vue 官方推荐的动效库 Dynamics.js  
前端知识分享喵

X

