标点符

订单号/唯一序列号生成方案（中篇）

2016年5月17日 · 1 min read

上一篇文章介绍了 twitter snowflake，snowflake的算法还是不错的，其实本身不复杂，复杂的是你客户端怎么用。遇到的问题如下：

- 代码部署在不同的服务器上，中间的机器ID如何设置，有没有更方便的获取机器ID的方式？
- 整个算法依赖时间的连续性，但是显示环境是线上服务器都开启了ntp，ntp情况下会出现时间倒退的问题。

再来重新分析下snowflake的优缺点：

Snowflake 生成的 unique ID 的组成 (由高位到低位):

- 41 bits: Timestamp (毫秒级)
- 10 bits: 节点 ID (datacenter ID 5 bits + worker ID 5 bits)
- 12 bits: sequence number

一共 63 bits (最高位是 0)

unique ID 生成过程:

- 10 bits 的机器号, 在 ID 分配 Worker 启动的时候, 从一个 Zookeeper 集群获取 (保证所有的 Worker 不会有重复的机器号)
- 41 bits 的 Timestamp: 每次要生成一个新 ID 的时候, 都会获取一下当前的 Timestamp, 然后分两种情况生成 sequence number:
- 如果当前的 Timestamp 和前一个已生成 ID 的 Timestamp 相同 (在同一毫秒中), 就用前一个 ID 的 sequence number + 1 作为新的 sequence number (12 bits); 如果本毫秒内的所有 ID 用完, 等到下一毫秒继续 (这个等待过程中, 不能分配出新的 ID)
- 如果当前的 Timestamp 比前一个 ID 的 Timestamp 大, 随机生成一个初始 sequence number (12 bits) 作为本毫秒内的第一个 sequence number

整个过程中, 只是在 Worker 启动的时候会对外部有依赖 (需要从 Zookeeper 获取 Worker 号), 之后就可以独立工作了, 做到了去中心化.

异常情况讨论:

- 在获取当前 Timestamp 时, 如果获取到的时间戳比前一个已生成 ID 的 Timestamp 还要小怎么办? Snowflake 的做法是继续获取当前机器的时间, 直到获取到更大的 Timestamp 才能继续工作 (在这个等待过程中, 不能分配出新的 ID)

从这个异常情况可以看出, 如果 Snowflake 所运行的那些机器时钟有大的偏差时, 整个 Snowflake 系统不能正常工作 (偏差得越多, 分配新 ID 时等待的时间越久)。从 Snowflake 的官方文档 (https://github.com/twitter/snowflake/#system-clock-dependency) 中也可以看到, 它明确要求 “You should use NTP to keep your system clock accurate”. 而且最好把 NTP 配置成不会向后调整的模式. 也就是说, NTP 纠正时间时, 不会向后回拨机器时钟。

问题一：如何解决时间同步问题?

为了解决上述的时间问题，可以采取的方案：

```
import java.security.SecureRandom;

/**
 * 自定义 ID 生成器
 * ID 生成规则: ID长达 64 bits
 *
 * | 41 bits: Timestamp (毫秒) | 3 bits: 区域（机房） | 10 bits: 机器编号 |
 */
public class CustomUUID {
    // 基准时间
    private long twepoch = 1288834974657L; //Thu, 04 Nov 2010 01:42:54 G
    // 区域标志位数
    private final static long regionIdBits = 3L;
    // 机器标识位数
    private final static long workerIdBits = 10L;
    // 序列号识位数
    private final static long sequenceBits = 10L;

    // 区域标志ID最大值
    private final static long maxRegionId = -1L ^ (-1L << regionIdBits);
    // 机器ID最大值
    private final static long maxWorkerId = -1L ^ (-1L << workerIdBits);
    // 序列号ID最大值
    private final static long sequenceMask = -1L ^ (-1L << sequenceBits)

    // 机器ID偏左移10位
    private final static long workerIdShift = sequenceBits;
    // 业务ID偏左移20位
    private final static long regionIdShift = sequenceBits + workerIdBit
    // 时间毫秒左移23位
```

```
31      private final static long timestampLeftShift = sequenceBits + worker
32
33      private static long lastTimestamp = -1L;
34
35      private long sequence = 0L;
36      private final long workerId;
37      private final long regionId;
38
39      public CustomUUID(long workerId, long regionId) {
40
41          // 如果超出范围就抛出异常
42          if (workerId > maxWorkerId || workerId < 0) {
43              throw new IllegalArgumentException("worker Id can't be greate
44          }
45          if (regionId > maxRegionId || regionId < 0) {
46              throw new IllegalArgumentException("datacenter Id can't be g
47          }
48
49          this.workerId = workerId;
50          this.regionId = regionId;
51      }
52
53      public CustomUUID(long workerId) {
54          // 如果超出范围就抛出异常
55          if (workerId > maxWorkerId || workerId < 0) {
56              throw new IllegalArgumentException("worker Id can't be greate
57          }
58          this.workerId = workerId;
59          this.regionId = 0;
60      }
61
62      public long generate() {
63          return this.nextId(false, 0);
64      }
65
66      /**
67       * 实际产生代码的
68       *
69       * @param isPadding
70       * @param busId
71       * @return
72       */
73      private synchronized long nextId(boolean isPadding, long busId) {
74
75          long timestamp = timeGen();
76          long paddingnum = regionId;
77
78          if (isPadding) {
79              paddingnum = busId;
80          }
81
82          if (timestamp < lastTimestamp) {
83              try {
84                  throw new Exception("Clock moved backwards.  Refusing to
85              } catch (Exception e) {
86                  e.printStackTrace();
87              }
88          }
89
90          //如果上次生成时间和当前时间相同,在同一毫秒内
91          if (lastTimestamp == timestamp) {
92              //sequence自增，因为sequence只有10bit，所以和sequenceMask相与一下
93              sequence = (sequence + 1) & sequenceMask;
94              //判断是否溢出,也就是每毫秒内超过1024，当为1024时，与sequenceMask相
95              if (sequence == 0) {
```

```
                //自旋等待到下一毫秒
                timestamp = tailNextMillis(lastTimestamp);
            }
        } else {
            // 如果和上次生成时间不同,重置sequence，就是下一毫秒开始，sequence计
            // 为了保证尾数随机性更大一些,最后一位设置一个随机数
            sequence = new SecureRandom().nextInt(10);
        }

        lastTimestamp = timestamp;

        return ((timestamp - twepoch) << timestampLeftShift) | (paddingnu
    }

    // 防止产生的时间比之前的时间还要小（由于NTP回拨等问题）,保持增量的趋势.
    private long tailNextMillis(final long lastTimestamp) {
        long timestamp = this.timeGen();
        while (timestamp <= lastTimestamp) {
            timestamp = this.timeGen();
        }
        return timestamp;
    }

    // 获取当前的时间戳
    protected long timeGen() {
        return System.currentTimeMillis();
    }
}
```

使用自定义的这种方法需要注意的几点：

- 为了保持增长的趋势，要避免有些服务器的时间早，有些服务器的时间晚，需要控制好所有服务器的时间，而且要避免NTP时间服务器回拨服务器的时间。
- 在跨毫秒时，序列号总是归0，会使得序列号为0的ID比较多，导致生成的ID取模后不均匀，所以序列号不是每次都归0，而是归一个0到9的随机数。
- 使用这个CustomUUID类，最好在一个系统中能保持单例模式运行。

问题二：如何解决分布式部署？

Snowflake 有一些变种, 各个应用结合自己的实际场景对 Snowflake 做了一些改动. 这里主要介绍 3 种.

1）Boundary flake

https://github.com/boundary/flake

变化:

- ID 长度扩展到 128 bits:
- 最高 64 bits 时间戳;

- 然后是 48 bits 的 Worker 号 (和 Mac 地址一样长);
- 最后是 16 bits 的 Seq Number
- 由于它用 48 bits 作为 Worker ID, 和 Mac 地址的长度一样, 这样启动时不需要和 Zookeeper 通讯获取 Worker ID. 做到了完全的去中心化
- 基于 Erlang

它这样做的目的是用更多的 bits 实现更小的冲突概率, 这样就支持更多的 Worker 同时工作. 同时, 每毫秒能分配出更多的 ID

2）Simpleflake

https://github.com/SawdustSoftware/simpleflake

Simpleflake 的思路是取消 Worker 号, 保留 41 bits 的 Timestamp, 同时把 sequence number 扩展到 22 bits;

Simpleflake 的特点:

- sequence number 完全靠随机产生 (这样也导致了生成的 ID 可能出现重复)
- 没有 Worker 号, 也就不需要和 Zookeeper 通讯, 实现了完全去中心化
- Timestamp 保持和 Snowflake 一致, 今后可以无缝升级到 Snowflake

Simpleflake 的问题就是 sequence number 完全随机生成, 会导致生成的 ID 重复的可能. 这个生成 ID 重复的概率随着每秒生成的 ID 数的增长而增长。

所以, Simpleflake 的限制就是每秒生成的 ID 不能太多 (最好小于 100次/秒, 如果大于 100 次/秒的场景, Simpleflake 就不适用了, 建议切换回 Snowflake)。

3）instagram 的做法

先简单介绍一下 instagram 的分布式存储方案:

- 先把每个 Table 划分为多个逻辑分片 (logic Shard), 逻辑分片的数量可以很大, 例如 2000 个逻辑分片
- 然后制定一个规则, 规定每个逻辑分片被存储到哪个数据库实例上面; 数据库实例不需要很多. 例如, 对有 2 个 PostgreSQL 实例的系统 (instagram 使用 PostgreSQL); 可以使用奇数逻辑分片存放到第一个数据库实例, 偶数逻辑分片存放到第二个数据库实例的规则
- 每个 Table 指定一个字段作为分片字段 (例如, 对用户表, 可以指定 uid 作为分片字段)

- 插入一个新的数据时, 先根据分片字段的值, 决定数据被分配到哪个逻辑分片 (logic Shard)
- 然后再根据 logic Shard 和 PostgreSQL 实例的对应关系, 确定这条数据应该被存放到哪台 PostgreSQL 实例上

instagram unique ID 的组成:

- 41 bits: Timestamp (毫秒)
- 13 bits: 每个 logic Shard 的代号 (最大支持 8 x 1024 个 logic Shards)
- 10 bits: sequence number; 每个 Shard 每毫秒最多可以生成 1024 个 ID

生成 unique ID 时, 41 bits 的 Timestamp 和 Snowflake 类似, 这里就不细说了.

主要介绍一下 13 bits 的 logic Shard 代号 和 10 bits 的 sequence number 怎么生成.

logic Shard 代号:

- 假设插入一条新的用户记录, 插入时, 根据 uid 来判断这条记录应该被插入到哪个 logic Shard 中.
- 假设当前要插入的记录会被插入到第 1341 号 logic Shard 中 (假设当前的这个 Table 一共有 2000 个 logic Shard)
- 新生成 ID 的 13 bits 段要填的就是 1341 这个数字

sequence number 利用 PostgreSQL 每个 Table 上的 auto-increment sequence 来生成:

- 如果当前表上已经有 5000 条记录, 那么这个表的下一个 auto-increment sequence 就是 5001 (直接调用 PL/PGSQL 提供的方法可以获取到)
- 然后把 这个 5001 对 1024 取模就得到了 10 bits 的 sequence number

instagram 这个方案的优势在于:

- 利用 logic Shard 号来替换 Snowflake 使用的 Worker 号, 就不需要到中心节点获取 Worker 号了. 做到了完全去中心化
- 另外一个附带的好处就是, 可以通过 ID 直接知道这条记录被存放在哪个 logic Shard 上

同时, 今后做数据迁移的时候, 也是按 logic Shard 为单位做数据迁移的, 所以这种做法也不会影响到今后的数据迁移

其他方式：Flickr Ticket Servers

flickr是用的一个叫做ticketserver的玩意，使用纯mysql来实现的。

```
CREATE TABLE `Tickets64` (
  `id` bigint(20) unsigned NOT NULL auto_increment,
  `stub` char(1) NOT NULL default '',
  PRIMARY KEY  (`id`),
  UNIQUE KEY `stub` (`stub`)
) ENGINE=MyISAM
```

先插入一条记录，然后再用replace去获取这个id。

```
REPLACE INTO Tickets64 (stub) VALUES ('a');
SELECT LAST_INSERT_ID();
```

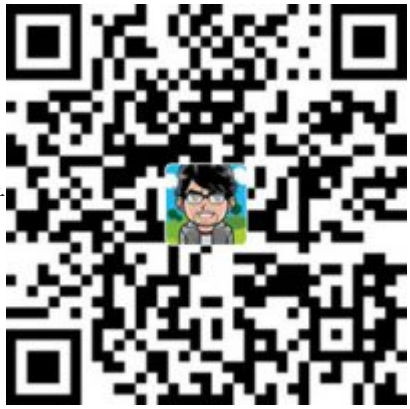
另有，mongodb自带的objectId也是一种高度唯一的序列可以利用Mongodb生成的直接拿过来用。

参考文章：

- http://darktea.github.io/notes/2013/12/08/Unique-ID
- http://www.jianshu.com/p/61817cf48cc3
- http://blog.paracode.com/2012/04/16/fast-id-generation-part-1/
- http://yellerapp.com/posts/2015-02-09-flake-ids.html
- http://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/

打赏作者

微信支付



支付宝

程序开发

#算法

« 为了效率你不应该做的7件事[译]

创业公司增长曲线与鸿沟曲线 »

0

Windows安装Docker记录

Docker是一种容器技术，可以将应用和环境等进行打包，形成一个独立的、类似于iOS的APP形式的应用。这个应用可以直接分发到任意一个支持Docker的环境中，通过简单的命令即可启动运行。Docker是一种最流行的容器化实现方案，和虚拟化技术类似，它极大的方便 ...

Aug 14, 2018 · 22 sec read

Windows下安装Tesserocr

在爬虫过程中，经常会遇到各种验证码，大多数验证码是图形验证码，先前的文章中有介绍到破解图形验证码的原理。最简单的破击验证码的方式是使用OCR。

Aug 13, 2018 · 1 min read

Selenium在Windows 上的安装

Selenium是一个用于Web应用程序自动化测试工具。Selenium测试直接运行在浏览器中，就像真正的用户在操作一样。Selenium是一款使用Apache License 2.0协议发布的开源框架。

Aug 13, 2018 · 1 min read

One Reply to “订单号/唯一序列号生成方案（中篇）”

欣 Says:

2016年6月16日 at 01:32

有没有生成10位的，而且去中心的生成

Leave a Reply

Write a response...

Name

E-mail address

Website Link

Post Comment