Twitter的分布式自增ID算法snowflake (Java版)

概述

分布式系统中，有一些需要使用全局唯一ID的场景，这种时候为了防止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。

有些时候我们希望能使用一种简单一些的ID，并且希望ID能够按照时间有序生成。

而twitter的snowflake解决了这种需求，最初Twitter把存储系统从MySQL迁移到Cassandra，因为Cassandra没有顺序ID生成机制，所以开发了这样一套全局唯一ID生成服务。

结构

snowflake的结构如下(每部分用-分开):

0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000

第一位为未使用，接下来的41位为毫秒级时间(41位的长度可以使用69年)，然后是5位datacenterId和5位workerId(10位的长度最多支持部署1024个节点） ，最后12位是毫秒内的计数（12位的计数顺序号支持每个节点每毫秒产生4096个ID序号）

一共加起来刚好64位，为一个Long型。(转换成字符串后长度最多19)

snowflake生成的ID整体上按照时间自增排序，并且整个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。经测试snowflake每秒能够产生26万个ID。

源码

(JAVA版本的源码)

```
/**
 * Twitter_Snowflake<br>
 * SnowFlake的结构如下(每部分用-分开):<br>
 * 0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000 <br>
 * 1位标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0<br>
 * 41位时间截(毫秒级)，注意，41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截)
 * 得到的值），这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的（如下下面程序IdWorker类的startTime属性）。41位的时间截，可以使用69年，年T = (1L << 41) / (1000L * 60 * 60 * 24 * 365) = 69<br>
 * 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId<br>
 * 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号<br>
 * 加起来刚好64位，为一个Long型。<br>
 * SnowFlake的优点是，整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞(由数据中心ID和机器ID作区分)，并且效率较高，经测试，SnowFlake每秒能够产生26万ID左右。
 */
public class SnowflakeIdWorker {

    // ==============================Fields===========================================
    /** 开始时间截 (2015-01-01) */
    private final long twepoch = 1420041600000L;

    /** 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的时间截 */
    private long lastTimestamp = -1L;

    //==============================Constructors=====================================
    /**
     * 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
```

随笔档案

- 2018年5月 (1)
- 2017年11月 (1)
- 2017年9月 (4)
- 2017年5月 (1)
- 2017年4月 (2)
- 2017年3月 (6)
- 2017年2月 (2)
- 2016年6月 (1)
- 2016年5月 (1)
- 2015年11月 (1)
- 2015年8月 (1)
- 2015年6月 (5)
- 2015年5月 (5)
- 2015年3月 (2)
- 2015年2月 (6)
- 2015年1月 (1)

```
     */
    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的时间截
        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的时间截
     * @return 当前时间戳
     */
    protected long tilNextMillis(long lastTimestamp) {
        long timestamp = timeGen();
        while (timestamp <= lastTimestamp) {
            timestamp = timeGen();
        }
        return timestamp;
```


最新评论

1. Re:Twitter的分布式自增ID算法snowflake (Java版)

@不敢平庸加上横杠...

--robinwyz

2. Re:Twitter的分布式自增ID算法snowflake (Java版)

UUID 不应该是32位吗？为何楼主说是36位？

--不敢平庸

3. Re:Twitter的分布式自增ID算法snowflake (Java版)

学习了 谢谢

--坦荡

```
    }

    /**
     * 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();
    }

    //==============================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(Long.toBinaryString(id));
            System.out.println(id);
        }
    }
}
```

参考

https://github.com/twitter/snowflake

分类： Java

relucent
关注 - 5
粉丝 - 7
+加关注

9 0

« 上一篇：Excel列名 字母和数字的转换

» 下一篇：rpm常用命令

posted @ 2015-11-11 10:19 relucent 阅读(49871) 评论(17) 编辑 收藏

评论列表

#1楼 2017-02-09 11:08 禅道

为什么一定要用mark

支持(0) 反对(0)

#2楼[楼主] 2017-02-17 09:57 relucent

@ 禅道
最后12位是毫秒内计数，因为是自增的，当超过12位时候计算一下（1000000000000 & 111111111111 = 0 ）可以判断出来。

支持(0) 反对(0)

#3楼 2017-04-10 09:10 埋头前进的码农

是时间戳还是时间截？

支持(0) 反对(0)

#4楼 2017-06-28 11:26 jEpac

nextId方法怎么是线程安全的了

支持(0) 反对(0)

#5楼 2017-06-29 15:08 Secondworld

@ jEpac
synchronized

支持(0) 反对(0)

#6楼 2017-06-29 15:11 jEpac

@ Secondworld
好吧 没看到 sync太重了

支持(0) 反对(0)

#7楼 2017-09-18 11:14 暴雪执行

请问，workerId 和datacenterId怎么生成

支持(0) 反对(0)

#8楼 2017-09-18 11:16 暴雪执行

假如我在不同的服务器上运行它，要让它的workerID和datacenterId不同，怎么做呢。

有劳各位回答一下。感激不尽~

支持(0) 反对(0)

#9楼[楼主] 2017-09-18 15:51 relucent

@ 暴雪执行

引用

假如我在不同的服务器上运行它，要让它的workerID和datacenterId不同，怎么做呢。
有劳各位回答一下。感激不尽~

当然是写配置文件(保证每个机器的workerId和datacenterId不同)，然后自己读取。

支持(0) 反对(0)

#10楼 2017-09-18 15:53 暴雪执行

@ relucent

嗯，谢谢回复~
那就可以在配置文件中初始化一个值，每读一次 修改一次配置文件。

谢谢，明白了~

支持(0) 反对(1)

#11楼[楼主] 2017-09-18 17:08 relucent

@ 暴雪执行

引用

@relucent
嗯，谢谢回复~
那就可以在配置文件中初始化一个值，每读一次 修改一次配置文件。
谢谢，明白了~

？配置文件不需要修改啊。
比如你有3台服务器，那么这3台服务器的配置分别设置为0,1,2 就行了。

支持(0) 反对(0)

#12楼 2017-09-30 11:27 实简

一共加起来刚好64位，为一个Long型。(转换成字符串长度为18)
System.out.println(Long.MAX_VALUE);
System.out.println((Long.MAX_VALUE+"").length());
//9223372036854775807
//19

支持(0) 反对(0)

#13楼 2017-10-09 14:44 实简

使用了synchronized,性能会下降很多啊

支持(0) 反对(0)

#14楼 2017-10-30 15:36 Tomy_Jx

很受用，算法代码也很间接，谢谢楼主

支持(0) 反对(0)

#15楼 2018-03-02 07:56 坦荡

学习了 谢谢

支持(0) 反对(0)

#16楼 2018-05-03 09:28 神龙摆尾

UUID 不应该是32位吗？为何楼主说是36位？

支持(0) 反对(0)

#17楼 2018-05-17 11:49 robinwyz

@ 不敢平庸
加上横杠

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· 店主二维码遭人掉包 获支付宝"敢收敢赔"计划赔付
· 应对微信口令消息禁令，淘宝增加了图片分享功能
· 微软市值再增74亿美元 已超过7600亿美元
· 小米公交支持167座城市：快看有你家吗？
· 技术极客：安全漏洞不会让EOS归零，BM的回应不够尊重技术
» 更多新闻...

最新知识库文章:
· 你可以把编程当做一项托付终身的职业
· 评审的艺术——谈谈现实中的代码评审
· 如何高效学习
· 如何成为优秀的程序员？
· 菜鸟工程师的超神之路 -- 从校园到职场
» 更多知识库文章...