holdno
5月8日 · 783 次阅读

记一次 golang 实现Twitter snowFlake算法 高效生成全局唯一ID

最近在着手准备一个H5游戏
因为这是我第一次接触游戏这个类目
即使量不大也想好好的做它一番
在设计表结构的时候想到了表全局唯一id这个问题
既然是游戏
那么一定是多人在线点点点(运营理想状态 哈哈哈)
一开始想使用mongoDB的objectId来作为全局唯一id
但是字符串作为索引的效率肯定不如整型来得实在

> 两者的主要差别就在于，字符类型有字符集的概念，每次从存储端到展现端之间都有一个字符集编码的过程。而这一过程主要消耗的就是CPU资源，对于In-memory的操作来说，这是一个不可忽视的消耗。如果使用整型替换可以减少CPU运算及内存和IO的开销。

所以最后考虑到理想状态下的效率及视觉效果(整型)，考虑找一个纯整型的id替代方案
无意间看到了Twitter的snowFlake算法

> 这篇内容大部分借鉴网络内容，整合在一起只为帮助自己和各位看官更好的理解snowFlake的原理

snowFlake 雪花算法

snowflake ID 算法是 twitter 使用的唯一 ID 生成算法，为了满足 Twitter 每秒上万条消息的请求，使每条消息有唯一、有一定顺序的 ID ，且支持分布式生成。

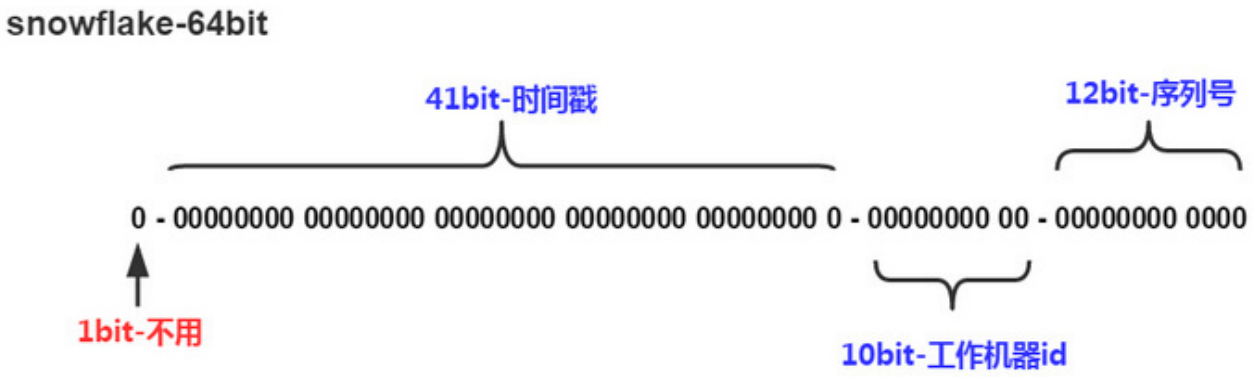
原理

其实很简单，只需要理解：某一台拥有独立标识(为机器分配独立id)的机器在1毫秒内生成带有不同序号的id
所以生成出来的id是具有时序性和唯一性的

构成

> 这里直接借鉴前人的整理，只为给大家更加清楚的讲解

snowflake ID 的结构是一个 64 bit 的 int 型数据。



- 第1位bit：
  二进制中最高位为1的都是负数，但是我们所需要的id应该都是整数，所以这里最高位应该为0
- 后面的41位bit：
  用来记录生成id时的毫秒时间戳，这里毫秒只用来表示正整数(计算机中正整数包含0)，所以可以表示的数值范围是0至2^41 - 1（这里为什么要-1很多人会范迷糊，要记住，计算机中数值都是从0开始计算而不是1）
- 再后面的10位bit：
  用来记录工作机器的id
  2^10 = 1024 所以当前规则允许分布式最大节点数为1024个节点 我们可以根据业务需求来具体分配worker数和每台机器1毫秒可生成的id序号number数
- 最后的12位：
  用来表示单台机器每毫秒生成的id序号
  12位bit可以表示的最大正整数为2^12 - 1 = 4096，即可用0、1、2、3...4095这4096(注意是从0开始计算)个数字来表示1毫秒内机器生成的序号(这个算法限定单台机器1毫秒内最多生成4096个id，超出则等待下一毫秒再生成)

最后将上述4段bit通过位运算拼接起来组成64位bit

实现

这里我们用golang来实现一下snowFlake
首先定义一下snowFlake最基础的几个常量，每个常量的用户我都通过注释来详细的告诉大家

```
// 因为snowFlake目的是解决分布式下生成唯一id 所以ID中是包含集群和节点编号在内的

const (
    numberBits uint8 = 12 // 表示每个集群下的每个节点，1毫秒内可生成的id序号的二进制位 对应上图中
```

```
    workerBits uint8 = 10 // 每台机器(节点)的ID位数 10位最大可以有2^10=1024个节点数 即每毫秒可
    // 这里求最大值使用了位运算，-1 的二进制表示为 1 的补码，感兴趣的同学可以自己算算试试 -1 ^ (-1
    workerMax int64 = -1 ^ (-1 << workerBits) // 节点ID的最大值，用于防止溢出
    numberMax int64 = -1 ^ (-1 << numberBits) // 同上，用来表示生成id序号的最大值
    timeShift uint8 = workerBits + numberBits // 时间戳向左的偏移量
    workerShift uint8 = numberBits // 节点ID向左的偏移量
    // 41位字节作为时间戳数值的话，大约68年就会用完
    // 假如你2010年1月1日开始开发系统 如果不减去2010年1月1日的时间戳 那么白白浪费40年的时间戳啊!
    // 这个一旦定义且开始生成ID后千万不要改了 不然可能会生成相同的ID
    epoch int64 = 1525705533000 // 这个是我在写epoch这个常量时的时间戳(毫秒)
)
```

上述代码中 两个偏移量 timeShift 和 workerShift 是对应图中时间戳和工作节点的位置

时间戳是在从右往左的 workerBits + numberBits (即22)位开始，大家可以数数看就很容易理解了

workerShift 同理

Worker 工作节点

因为是分布式下的ID生成算法，所以我们要生成多个Worker，所以这里抽象出一个woker工作节点所需要的基本参数

```
// 定义一个woker工作节点所需要的基本参数
type Worker struct {
    mu sync.Mutex // 添加互斥锁 确保并发安全
    timestamp int64 // 记录上一次生成id的时间戳
    workerId int64 // 该节点的ID
    number int64 // 当前毫秒已经生成的id序列号(从0开始累加) 1毫秒内最多生成4096个ID
}
```

实例化工作节点

由于是分布式情况下，我们应该通过外部配置文件或者其他方式为每台机器分配独立的id

```
// 实例化一个工作节点
// workerId 为当前节点的id
func NewWorker(workerId int64) (*Worker, error) {
    // 要先检测workerId是否在上面定义的范围内
    if workerId < 0 || workerId > workerMax {
        return nil, errors.New("Worker ID excess of quantity")
    }
    // 生成一个新节点
    return &Worker{
        timestamp: 0,
        workerId: workerId,
        number: 0,
```

```
    }, nil
}
```

> 可以通过redis来为分布式环境下的每台机子生成唯一id
>
> 该部分不包含在算法内

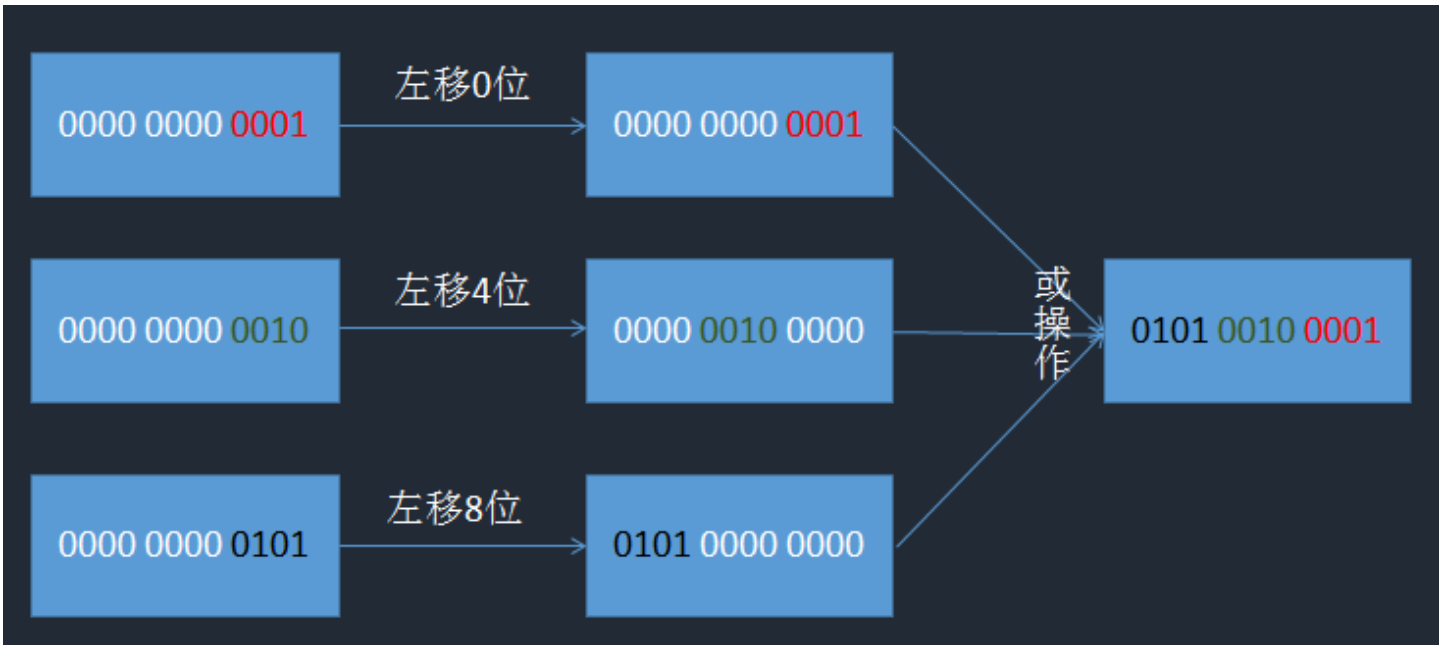
生成id

```
// 生成方法一定要挂载在某个woker下，这样逻辑会比较清晰 指定某个节点生成id
func (w *Worker) GetId() int64 {
    // 获取id最关键的一点 加锁 加锁 加锁
    w.mu.Lock()
    defer w.mu.Unlock() // 生成完成后记得 解锁 解锁 解锁

    // 获取生成时的时间戳
    now := time.Now().UnixNano() / 1e6 // 纳秒转毫秒
    if w.timestamp == now {
        w.number++

        // 这里要判断，当前工作节点是否在1毫秒内已经生成numberMax个ID
        if w.number > numberMax {
            // 如果当前工作节点在1毫秒内生成的ID已经超过上限 需要等待1毫秒再继续生成
            for now <= w.timestamp {
                now = time.Now().UnixNano() / 1e6
            }
        }
    } else {
        // 如果当前时间与工作节点上一次生成ID的时间不一致 则需要重置工作节点生成ID的序号
        w.number = 0
        // 下面这段代码看到很多前辈都写在if外面，无论节点上次生成id的时间戳与当前时间是否相同 都重新
        w.timestamp = now // 将机器上一次生成ID的时间更新为当前时间
    }

    ID := int64((now - epoch) << timeShift | (w.workerId << workerShift) | (w.number))
    return ID
}
```

很多新入门的朋友可能看到最后的ID := xxxxx << xxx | xxxxxx << xx | xxxxx 有点懵

这里是对各部分的bit进行归位并通过按位或运算(就是这个‘|’)将其整合

用一张图来解释

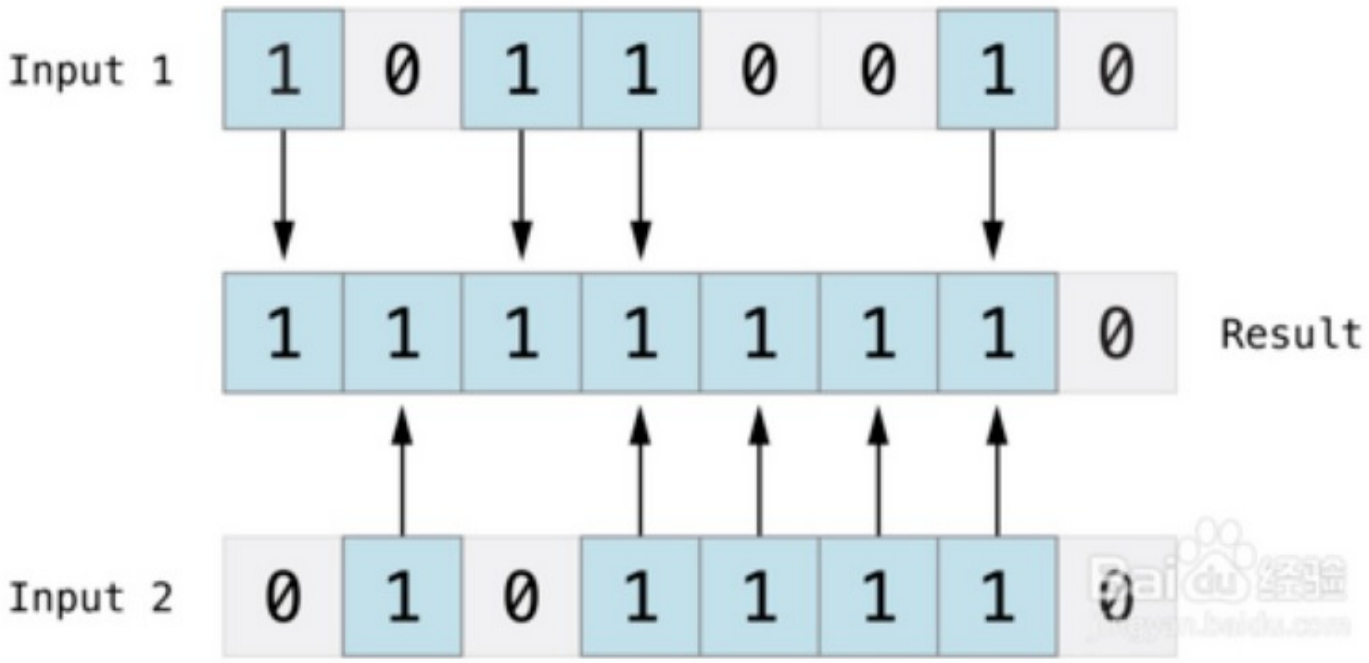


想必大家看完后就很清晰了吧

至于某一段一开始位数可能不够？ 别担心二进制空位会自动补0！

针对这个"|"多解释一下

参加运算的两个数，换算为二进制(0、1)后，进行或运算。只要相应位上存在1，那么该位就取1，均不为1，即为0



同样 看完图就很清楚啦(百度会不会说我盗图啊T.T)

Test

接下来我们用golang的测试包来测试一下我们刚才生成的代码

```
package snowFlakeByGo

import (
    "testing"
    "fmt"
)

func TestSnowFlakeByGo(t *testing.T) {
    // 测试脚本

    // 生成节点实例
    worker, err := NewWorker(1)

    if err != nil {
        fmt.Println(err)
        return
    }

    ch := make(chan int64)
    count := 10000
    // 并发 count 个 goroutine 进行 snowflake ID 生成
    for i := 0; i < count; i++ {
        go func() {
            id := worker.GetId()
            ch <- id
        }()
    }

    defer close(ch)

    m := make(map[int64]int)
    for i := 0; i < count; i++  {
        id := <- ch
        // 如果 map 中存在为 id 的 key, 说明生成的 snowflake ID 有重复
        _, ok := m[id]
        if ok {
            t.Error("ID is not unique!\n")
            return
        }
        // 将 id 作为 key 存入 map
        m[id] = i
    }
    // 成功生成 snowflake ID
    fmt.Println("All", count, "snowflake ID Get successed!")
}
```

结果

用的是17版 13寸macbook pro(非Touch Bar)进行测试

```
wbyMacBook-Pro:snowFlakeByGo xxx$ go test
All 10000 snowflake ID Get successed!
PASS
ok      github.com/holdno/snowFlakeByGo 0.031s
```

并发生成一万个id用时0.031秒

如果能跑在分布式服务器上 估计更快了~

够用了够用了

本文结合网络内容加上自己的一些小小的优化整理而成

最后附上github地址：https://github.com/holdno/sno...

觉得有用可以给颗星星哦

不早了要去洗洗睡了

晚安~