利用BitMap进行大数据排序去重

# 问题

**问题提出**：M（如10亿）个int整数，只有其中N个数重复出现过，读取到内存中并将重复的整数删除。

# 解决方案

**问题分析：**

我们肯定会先想到在计算机内存中开辟M个int整型数据数组，来one bye one读取M个int类型数组， 然后在一一比对数值，最后将重复数据的去掉。当然这在处理小规模数据是可行的。

我们考虑大数据的情况：例如在java语言下，对10亿个int类型数据排重。

java中一个int类型在内存中占4byte。那么10亿个int类型数据共需要开辟10^9次方\*4byte≈4GB的连续内存空间。以32位操作系统电脑为例，最大支持内存为4G，可用内存更是小于4G。所以上述方法在处理大数据时根本行不通。

**思维转化：**

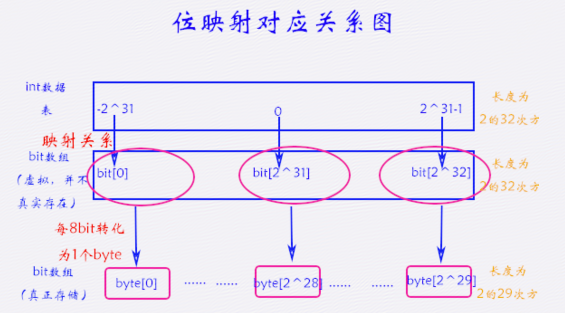
既然我们不能为所有 int 类型的数据开辟 int 类型数组，那么可以采取更小的数据类型来读取缓存 int 类型数据。考虑到计算机内部处理的数据都是 01 序列的bit，那么我们是否可以用 1bit 来表示一个 int 类型数据。

**位映射的引出：**

使用较小的数据类型指代较大的数据类型。如上所说的问题，我们可以用1个 bit 来对应一个int 整数。假如对应的 int 类型的数据存在，就将其对应的 bit 赋值为1，否则，赋值为0（boolean类型）。java中 int 范围为 -2^31 到 2^31-1. 那么所有可能的数值组成的长度为2^32. 对应的 bit 长度也为 2^32. 那么可以用这样处理之后只需要开辟2^32 bit = 2^29 byte = 512M 大小的 内存空间 。显然，这样处理就能满足要求了，虽然对内存的消耗也不太小。

**问题解决方案：**

首先定义如下图的int - byte 映射关系，当然，映射关系可以自定义。但前提要保证你的数组上下标不能越界。



但如上定义的bit[]数组显然在计算机中是不存在的，所我们需要将其转化为 java 中的一个基本数据类型存储。显然，byte[] 是最好的选择。

**将其转化为byte[] 数组方案：**

自定义的映射关系表，每个bit对应一个 int 数值，将 int 的最大值，最小值与数组的最大最小索引相对应。从上图可以看出来 int 数值与bit索引相差 2^31次方。当然，你也可以定义其他的映射关系，只是注意不要发生数组越界的情况。

bit[]索引：由于最大值可能是2^32,故用long接收: long bitIndex = num + (1l << 31);

byte[]索引: int index = (int) (bitIndex / 8); ，在字节byte[index]中的具体位置： int innerIndex = (int) (bitIndex % 8);

更新值： dataBytes[index] = (byte) (dataBytes[index] | (1 << innerIndex));

# 大数据处理-Bitmap

　　MapReduce是一种编程模型，用于大规模数据集（大于1TB）的并行运算。概念"Map（映射）"和"Reduce（归约）"

Bit-map空间压缩和快速排序去重

## Bit-map的基本思想

　　32位机器上，对于一个整型数，比如int a=1 在内存中占32bit位，这是为了方便计算机的运算。但是对于某些应用场景而言，这属于一种巨大的浪费，因为我们可以用对应的32bit位对应存储十进制的0-31个数，**而这就是Bit-map的基本思想**。Bit-map算法利用这种思想处理大量数据的排序、查询以及去重。

　　Bitmap在用户群做交集和并集运算的时候也有极大的便利。

## Bit-map应用之快速排序

　　假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）,我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0，

bitmap1：对应位设置为1；

bitmap2：遍历一遍Bit区域，将该位是1的位的编号输出（2，3，4，5，7），这样就达到了排序的目的，时间复杂度**O(n)**。

　　优点：

　　　　运算效率高，不需要进行比较和移位；

　　　　占用内存少，比如N=10000000；只需占用内存为N/8=1250000Byte=1.25M。

　　缺点：

**所有的数据不能重复。即不可对重复的数据进行排序和查找。**

## Bit-map应用之快速去重

　　2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

　　首先，根据“**内存空间不足以容纳这2.5亿个整数**”我们可以快速的联想到Bit-map。下边关键的问题就是怎么设计我们的Bit-map来表示这2.5亿个数字的状态了。其实这个问题很简单，**一个数字的状态只有三种，分别为不存在，只有一个，有重复**。因此，我们只需要2bits就可以对一个数字的状态进行存储了，假设我们设定一个数字不存在为00，存在一次01，存在两次及其以上为11。那我们大概需要存储空间几十兆左右。

　　接下来的任务就是遍历一次这2.5亿个数字，如果对应的状态位为00，则将其变为01；如果对应的状态位为01，则将其变为11；如果为11，,对应的转态位保持不变。

　　最后，我们将状态位为01的进行统计，就得到了不重复的数字个数，时间复杂度为O(n)。

## Bit-map应用之快速查询

　　同样，我们利用Bit-map也可以进行快速查询，这种情况下对于一个数字只需要一个bit位就可以了，0表示不存在，1表示存在。假设上述的题目改为，如何快速判断一个数字是够存在于上述的2.5亿个数字集合中。

　　同之前一样，首先我们先对所有的数字进行一次遍历，然后将相应的转态位改为1。遍历完以后就是查询，由于我们的Bit-map采取的是连续存储（整型数组形式，一个数组元素对应32bits），我们实际上是**采用了一种分桶的思想**。一个数组元素可以存储32个状态位，那将待查询的数字除以32，定位到对应的数组元素（桶），然后再求余（%32），就可以定位到相应的状态位。如果为1，则代表改数字存在；否则，该数字不存在。

## Bit-map扩展——Bloom Filter(布隆过滤器)

　　当一个元素被加入集合中时,通过k各散列函数将这个元素映射成一个位数组中的k个点,并将这k个点全部置为1.

**有一定的误判率**--在判断一个元素是否属于某个集合时,有可能会把不属于这个集合的元素误判为属于这个集合.因此,它不适合那些"零误判"的应用场合.在能容忍低误判的应用场景下,**布隆过滤器通过极少的误判换区了存储空间的极大节省.**

　　Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素x，第i个哈希函数映射的位置hi(x)就会被置为1（1≤i≤k）。注：如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。

Bloom Filte

在判断y是否属于这个集合时，对y应用k次哈希函数，若所有hi(y)的位置都是1（1≤i≤k），就认为y是集合中的元素，否则就认为y不是集合中的元素。

## 总结

　　使用Bit-map的思想，我们可以将存储空间进行压缩，而且可以对数字进行快速排序、去重和查询的操作。Bloom Fliter是Bit-map思想的一种扩展，它可以在允许低错误率的场景下，大大地进行空间压缩，是一种拿错误率换取空间的数据结构。

## 应用

适用范围：**可进行数据的快速查找，判重，删除**，一般来说数据范围是int的10倍以下

　基本原理及要点：使用bit数组来表示某些元素是否存在，比如8位电话号码

　扩展：bloom filter可以看做是对bit-map的扩展

问题实例：

1、已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。

8位最多99 999 999，大概需要99m个bit，大概10几M字节的内存即可。

2、在2.5亿个整数中找出不重复的整数，内存不足以容纳这2.5亿个整数。

方案1：采用**2-Bitmap**（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存232\*2bit=1GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。扫描完事后，查看bitmap，把对应位是01的整数输出即可。

# 代码

1 import java.util.Random;

3 /\*\*

4 \* **问题：M（如10亿）个int整数，只有其中N个数重复出现过，读取到内存中并将重复的整数删除。**<br/>

5 \* **使用位映射来进行海量数据的去重排序，原先一个元素用一个int现在只用一个bit， 内存占比4\*8bit:1bit=32:1**<br/>

6 \* **亦可用java语言提供的BitSet，不过其指定bit index的参数为int类型，因此在此例中将输入数转为bit index时对于较大的数会越界**<br><br/>

7 \*/

8 public class BigDataSort {

9

10 private static final int CAPACITY = 1\_000\_000;// 数据容量

11

12 public static void main(String[] args) {

13

14 testMyFullBitMap();

15

16 }

17

18 public static void testMyFullBitMap() {

19 MyFullBitMap ms = new MyFullBitMap();

20

21 byte[] bytes = null;

22

23 Random random = new Random();

24 long startTime = System.currentTimeMillis();

25 for (int i = 0; i < CAPACITY; i++) {

26 int num = random.nextInt();

27 // System.out.println("读取了第 " + (i + 1) + "\t个数: " + num);

28 bytes = ms.setBit(num);

29 }

30 long endTime = System.currentTimeMillis();

31 System.out.printf("存入%d个数，用时%dms\n", CAPACITY, endTime - startTime);

32

33 startTime = System.currentTimeMillis();

34 ms.output(bytes);

35 endTime = System.currentTimeMillis();

36 System.out.printf("取出%d个数，用时%dms\n", CAPACITY, endTime - startTime);

37 }

38 }

39

40 class MyFullBitMap {

41 // 定义一个byte数组表示所有的int数据，一bit对应一个，共2^32b=2^29B=512MB

42 private byte[] dataBytes = new byte[1 << 29];

43

44 /\*\*

45 \* 读取数据，并将对应数数据的 到对应的bit中，并返回byte数组

46 \*

47 \* @param num

48 \* 读取的数据

49 \* @return byte数组 dataBytes

50 \*/

51 public byte[] setBit(int num) {

52

53 long bitIndex = num + (1l << 31); // 获取num数据对应bit数组（虚拟）的索引

54 int index = (int) (bitIndex / 8); // bit数组（虚拟）在byte数组中的索引

55 int innerIndex = (int) (bitIndex % 8); // bitIndex 在byte[]数组索引index 中的具体位置

56

57 // System.out.println("byte[" + index + "] 中的索引：" + innerIndex);

58

59 dataBytes[index] = (byte) (dataBytes[index] | (1 << innerIndex));

60 return dataBytes;

61 }

62

63 /\*\*

64 \* 输出数组中的数据

65 \*

66 \* @param bytes

67 \* byte数组

68 \*/

69 public void output(byte[] bytes) {

70 int count = 0;

71 for (int i = 0; i < bytes.length; i++) {

72 for (int j = 0; j < 8; j++) {

73 if (((bytes[i]) & (1 << j)) != 0) {

74 count++;

75 int number = (int) ((((long) i \* 8 + j) - (1l << 31)));

76 // System.out.println("取出的第 " + count + "\t个数: " + number);

77 }

78 }

79 }

80 }

81 }