2. 位图&布隆过滤器&海量数据面试题

[本节目标]

- **1. 位图**
- **2. 布隆过滤器**
- **3. 海量数据处理面试题**

1. 位图

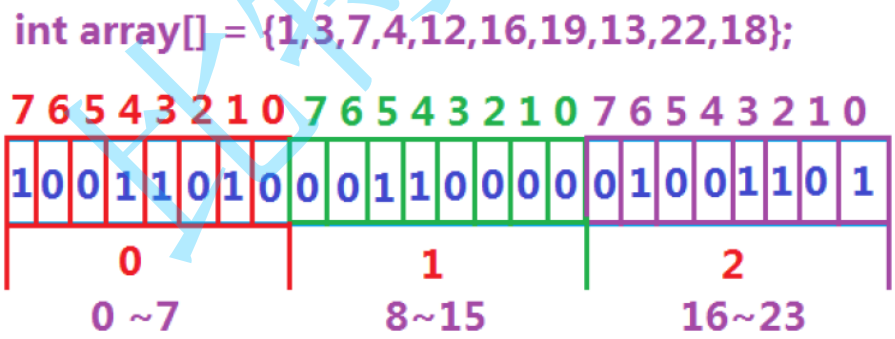
1.1 位图概念

1. 面试题

   给40亿个不重复的无符号整数，没排过序。给一个无符号整数，如何快速判断一个数是否在这40亿个数中。【腾讯】

   1. 遍历，时间复杂度O(N)
   2. 排序(O(NlogN))，利用二分查找: logN
   3. 位图解决

      数据是否在给定的整形数据中，结果是在或者不在，刚好是两种状态，那么可以使用一个二进制比特位来代表数据是否存在的信息，如果二进制比特位为1，代表存在，为0代表不存在。比如：



2. 位图概念

   所谓位图，就是用每一位来存放某种状态，适用于海量数据，数据无重复的场景。通常是用来判断某个数据存不存在的。

1.2 位图的实现

```
/*
注意：以下模拟实现中只是将Java8中BitSet中部分常用的接口进行了实现，其他同学参考帮助文档
*/
public class BitSet {
    // 因为在底层使用long的数组来保存所有的比特位，而一个long占8个字节，总共有64个比特位，即2^6

    private final static int ADDRESS_BITS_PER_WORD = 6;
```

```
// 一个long中总的比特位的个数
private final static int BITS_PER_WORD = 1 << ADDRESS_BITS_PER_WORD;

private transient int wordsInUse = 0;

private long[] words;    // 保存所有的比特位


// 给定一个比特位的索引，计算该比特位在words数组中的那个索引位置上
private static int wordIndex(int bitIndex) {
    return bitIndex >> ADDRESS_BITS_PER_WORD;
}

private void initWords(int nbits) {
    words = new long[wordIndex(nbits-1) + 1];
}

public BitSet() {
    initWords(BITS_PER_WORD);
}

public BitSet(int nbits) {
    // nbits 不能是负数
    if (nbits < 0)
        throw new NegativeArraySizeException("nbits < 0: " + nbits);

    initWords(nbits);
}


private void expandTo(int wordIndex) {
    int wordsRequired = wordIndex+1;
    if (wordsInUse < wordsRequired) {
        ensureCapacity(wordsRequired);
        wordsInUse = wordsRequired;
    }
}

private void ensureCapacity(int wordsRequired) {
    if (words.length < wordsRequired) {
        // Allocate larger of doubled size or required size
        int request = Math.max(2 * words.length, wordsRequired);
        words = Arrays.copyOf(words, request);
    }
}

public void flip(int bitIndex) {
    if (bitIndex < 0)
        throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

    int wordIndex = wordIndex(bitIndex);

    expandTo(wordIndex);
```

```
        words[wordIndex] ^= (1L << bitIndex);
    }

    // 将bitIndex比特位设置为1
    public void set(int bitIndex) {
        // 注意: bitIndex不能为负数
        if (bitIndex < 0)
            throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

        // 计算bitIndex在words中的下标
        int wordIndex = wordIndex(bitIndex);
        expandTo(wordIndex);   // 如果下标超了，对words进行扩容

        // 将words中第bitIndex个比特位设置为1
        words[wordIndex] |= (1L << bitIndex); // Restores invariants
    }

    // 将bitIndex个比特位置为value，value: true置为1，false置为0
    public void set(int bitIndex, boolean value) {
        if (value)
            set(bitIndex);
        else
            clear(bitIndex);
    }

    // 将bitIndex个比特位清零
    public void clear(int bitIndex) {
        if (bitIndex < 0)
            throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

        int wordIndex = wordIndex(bitIndex);
        // 位图中不存在该比特位，直接返回
        if (wordIndex >= wordsInUse)
            return;

        words[wordIndex] &= ~(1L << bitIndex);
    }

    // 将所有的比特位全部清零
    public void clear() {
        while (wordsInUse > 0)
            words[--wordsInUse] = 0;
    }

    // 获取bitIndex的比特位是0还是1
    public boolean get(int bitIndex) {
        if (bitIndex < 0)
            throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

        int wordIndex = wordIndex(bitIndex);
        return (wordIndex < wordsInUse)

                && ((words[wordIndex] & (1L << bitIndex)) != 0);
```

```
    }

    // 检测位图中是否存在有效比特位
    public boolean isEmpty() {
        return wordsInUse == 0;
    }


    // 返回实际总的比特位数
    public int size() {
        return words.length * BITS_PER_WORD;
    }
}
```

BitSet实现原理

1.3 位图的应用

1. 快速查找某个数据是否在一个集合中
2. 排序 + 去重
3. 求两个集合的交集、并集等
4. 操作系统中磁盘块标记

2. 布隆过滤器

2.1 布隆过滤器提出

日常生活中，包括在设计计算机软件时，我们**经常要判断一个元素是否在一个集合中**。比如在**字处理软件中，需要检查一个英语单词是否拼写正确**（也就是要判断它是否在已知的字典中）；在 FBI，**一个嫌疑人的名字是否已经在嫌疑名单上**；在网络爬虫里，**一个网址是否被访问过等等**。最直接的方法就是**将集合中全部的元素存在计算机中，遇到一个新元素时，将它和集合中的元素直接比较即可。**

一般来讲，计算机中的**集合是用哈希表（hash table）来存储的。它的好处是快速准确，缺点是费存储空间。**当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来了。

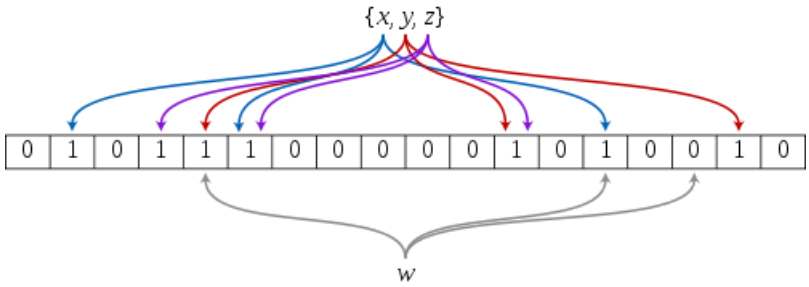
比如说，一个像 Yahoo,Hotmail 和 Gmai 那样的公众电子邮件（email）提供商，总是需要**过滤来自发送垃圾邮件的人（spamer）的垃圾邮件。一个办法就是记录下那些发垃圾邮件的 email 地址。由于那些发送者不停地在注册新的地址，全世界少说也有几十亿个发垃圾邮件的地址，将他们都存起来则需要大量的网络服务器。**

如果用哈希表，每存储一亿个 email 地址， 就需要 1.6GB 的内存（用哈希表实现的具体办法是将每一个 email 地址对应成一个八字节的信息指纹,然后将这些信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此一个 email 地址需要占用十六个字节。一亿个地址大约要 1.6GB， 即十六亿字节的内存）。因此存贮几十亿个邮件地址可能需要上百 GB 的内存。除非是超级计算机，一般服务器是无法存储的。

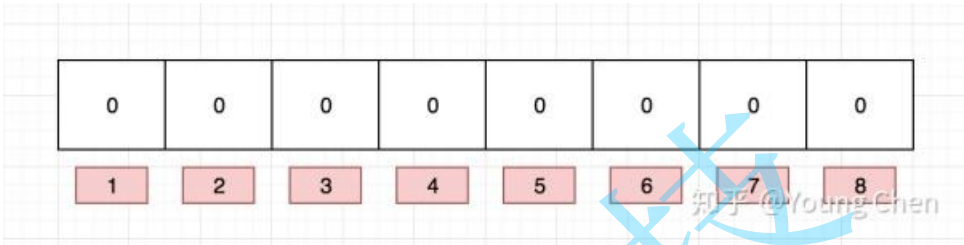
1. 用哈希表存储用户记录，缺点：浪费空间
2. 用位图存储用户记录，缺点：位图一般只能处理整形，如果内容编号是字符串，就无法处理了。
3. 将哈希与位图结合，即布隆过滤器

2.2布隆过滤器概念

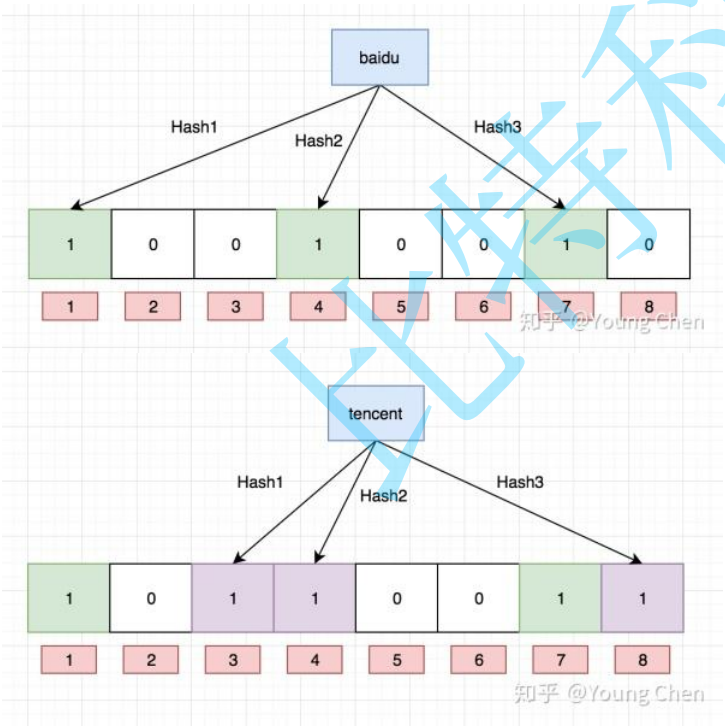
> **布隆过滤器是**由布隆（Burton Howard Bloom）在1970年提出的一种紧凑型的、比较巧妙的**概率型数据结构**，特点是**高效地插入和查询，可以用来告诉你 “某样东西一定不存在或者可能存在”**，它是用多个哈希函数，将一个数据映射到位图结构中。此种方式**不仅可以提升查询效率，也可以节省大量的内存空间。**



2.3 布隆过滤器的插入



向布隆过滤器中插入："baidu"



2.4 布隆过滤器的查找

> 布隆过滤器的思想是将一个元素用多个哈希函数映射到一个位图中，因此被映射到的位置的比特位一定为1。所以可以按照以下方式进行查找：**分别计算每个哈希值对应的比特位置存储的是否为零，只要有一个为零，代表该元素一定不在哈希表中，否则可能在哈希表中。**
>
> **注意：布隆过滤器如果说某个元素不存在时，该元素一定不存在，如果该元素存在时，该元素可能存在，因为有些哈希函数存在一定的误判。**

> 比如：在布隆过滤器中查找"alibaba"时，假设3个哈希函数计算的哈希值为：1、3、7，刚好和其他元素的比特位重叠，此时布隆过滤器告诉该元素存在，但实该元素是不存在的。

2.5 布隆过滤器模拟实现

```
import java.util.BitSet;

public class  BloomFilter {
    private static final  int  DEFAULT_SIZE  =2 << 24 ;
    private static final  int [] seeds = new  int []{5,7, 11 , 13 , 31 , 37 , 61};

    private BitSet bits;     // 用来存储元素
    private SimpleHash[]  func;  // 哈希函数所对应类
    private int size = 0;

    public  BloomFilter() {
        bits= new  BitSet(DEFAULT_SIZE);
        func = new  SimpleHash[seeds.length];

        for( int  i= 0 ; i< seeds.length; i ++ ) {
            func[i]=new  SimpleHash(DEFAULT_SIZE, seeds[i]);
        }
    }

    public void  add(String value) {
        if(null == value)
            return;

        for(SimpleHash f : func) {
            bits.set(f.hash(value));
        }

        size++;
    }

    public boolean  contains(String value) {
        if(value ==null ) {
            return false;
        }

        for(SimpleHash f : func) {
            if(!bits.get(f.hash(value))){
                return false;
            }
        }
        return true;
    }

    // 构建哈希函数
    public static class SimpleHash {
        private int  cap;
        private int  seed;

        public  SimpleHash( int cap, int seed) {
```

```
            this.cap= cap;
            this.seed =seed;
        }

        public int hash(String value) {
            int  result=0 ;
            int  len= value.length();
            for  (int i= 0 ; i< len; i ++ ) {
                result =seed* result + value.charAt(i);
            }
            return (cap - 1 ) & result;
        }
    }

    public static void  main(String[] args) {
        String s1  = "欧阳锋";
        String s2  = "欧阳克";
        String s3  = "金轮法王";
        String s4  = "霍都";

        BloomFilter filter=new  BloomFilter();

        filter.add(s1);
        filter.add(s2);
        filter.add(s3);
        filter.add(s4);

        System.out.println(filter.contains("杨过"));
        System.out.println(filter.contains("金轮法王"));
    }
}
```

2.6 布隆过滤器删除

> **布隆过滤器不能直接支持删除工作，因为在删除一个元素时，可能会影响其他元素。**
>
> 比如：删除上图中"tencent"元素，如果直接将该元素所对应的二进制比特位置0，“baidu”元素也被删除了，因为这两个元素在多个哈希函数计算出的比特位上刚好有重叠。
>
> 一种支持删除的方法：将布隆过滤器中的每个比特位扩展成一个小的计数器，插入元素时给k个计数器(k个哈希函数计算出的哈希地址)加一，删除元素时，给k个计数器减一，通过多占用几倍存储空间的代价来增加删除操作。
>
> 缺陷：
>
> 1. 无法确认元素是否真正在布隆过滤器中
> 2. 存在计数回绕

2.7 布隆过滤器优点

> 1. 增加和查询元素的时间复杂度为:O(K), (K为哈希函数的个数，一般比较小)，与数据量大小无关
> 2. 哈希函数相互之间没有关系，方便硬件并行运算
> 3. 布隆过滤器不需要存储元素本身，在某些对保密要求比较严格的场合有很大优势
> 4. 在能够承受一定的误判时，布隆过滤器比其他数据结构有这很大的空间优势

5. 数据量很大时，布隆过滤器可以表示全集，其他数据结构不能
6. 使用同一组散列函数的布隆过滤器可以进行交、并、差运算

2.8 布隆过滤器缺陷

1. 有误判率，即存在假阳性(False Position)，即不能准确判断元素是否在集合中(补救方法：再建立一个白名单，存储可能会误判的数据)
2. 不能获取元素本身
3. 一般情况下不能从布隆过滤器中删除元素
4. 如果采用计数方式删除，可能会存在计数回绕问题

3. 海量数据面试题

3.1 哈希切割

给一个超过100G大小的log file, log中存着IP地址, 设计算法找到出现次数最多的IP地址？ 与上题条件相同，如何找到top K的IP？如何直接用Linux系统命令实现？

3.2 位图应用

1. 给定100亿个整数，设计算法找到只出现一次的整数？
2. 给两个文件，分别有100亿个整数，我们只有1G内存，如何找到两个文件交集？
3. 位图应用变形：1个文件有100亿个int，1G内存，设计算法找到出现次数不超过2次的所有整数

3.3 布隆过滤器

1. 给两个文件，分别有100亿个query，我们只有1G内存，如何找到两个文件交集？分别给出精确算法和近似算法
2. 如何扩展BloomFilter使得它支持删除元素的操作

扩展阅读：

一致性哈希

哈希与加密