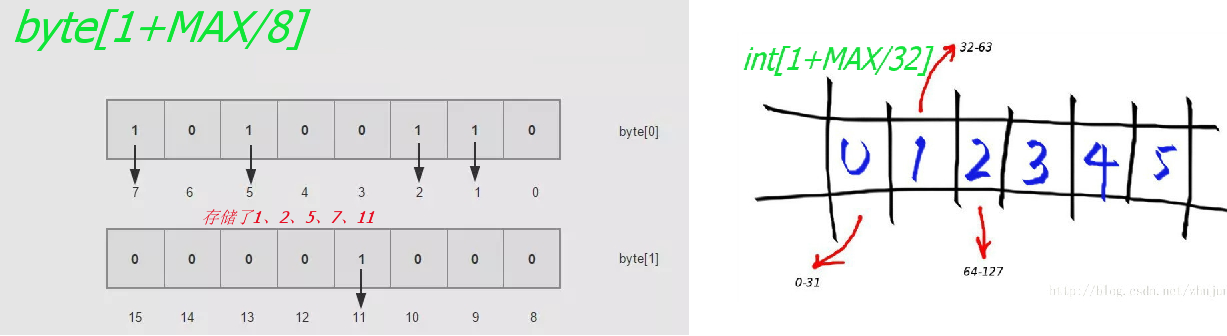
哈希表在查找定位操作上具有O(1)的常量时间，常用于做性能优化，但是内存毕竟是有限的，当数据量太大时用哈希表就会内存溢出了。而考虑对这些大数据进行存盘分批处理又有IO上的开销，性能又不能满足要求。这个时候我们就得介绍BitMap算法了。

|  |
| --- |
| **bitMap原理介绍** |
| BitMap算法是基于**位映射**的，对于内存中一段连续的二进制位，其中每一位的值（0或1）代表了**值为该二进制位索引的元素**【正整数】是否存在。这相当于**用bit位来存储数据**，因而大大的节省了内存空间。  >>对于存储操作，只需要根据元素的值找到相应的二进制位，并将该位的值置1即可。  >>对于查找操作，只需要根据元素的值找到相应的二进制位，并判断该位的值是否为1即可。  >>对于删除操作，只需要根据元素的值找到相应的二进制位，并将该位的值置0即可。 |

举一个例子，要对一个**超大的无序有界int序列做去重、排序、存储和查找**，比如是10亿个值在0到10亿之间的可能重复的数，就说存储那些整数吧，一个整数占4字节，10亿个就是3.72g，在加上需要这个数据量来一次排序，内存如何受的了。而用BitMap算法，我们可以对内存中连续的10亿个二进制位进行映射，其中每个二进制位的值（0或1）用来表示值等于该二进制位索引的整数是否存在，如下图所示（这里是用byte[]、int[]、long[]等来表示内存中连续的二进制位）。将这10亿个数依次添加到bitmap结构中，就相当于只用了(3.72/32)g的内存空间就存储了10亿数据，一个占用32bit的数据现在只占用了1bit，节省了不少的空间；而排序就更不用说了，二进制位的索引已经维护了顺序了；去重嘛，添加的时候就已经去重了；查找呢，按元素值查看相应的二进制位的值是否为1就是了。



|  |
| --- |
| **bitMap排序复杂度分析** |
| Bitmap排序需要的时间复杂度和空间复杂度依赖于数据中最大的数字。  bitmap排序的时间复杂度不是O(N)的，而是取决于待排序整数序列中的最大值MAX，但是因为bitmap存储的数据之间是没有关联性，你可以用**多线程的方式加速读取**，比如开10个线程去读，那么复杂度就为O(Max/10)。 |

|  |
| --- |
| **bitMap缺点** |
| 1>>bitmap不可存储重复数据，不可对重复数据排序。（少量重复数据查找可以用2-bitmap）  2>>存储和排序稀疏数据时（如1、20、1000、1w、50w、1ww），会浪费空间和时间。比如存储，这种问题可以通过引入 **Roaring BitMap** 来解决。 |

|  |
| --- |
| **bitMap算法扩展** |
| BitMap算法使用场景很广，可以运用在数据的**快速查找、排序、删除、去重、压缩**等。比如说 Oracle、[Redis](https://cloud.tencent.com/product/crs) 中都有用到 BitMap，当然更多的系统会有比 BitMap 稍微复杂一些的算法，比如 Bloom Filter、Counting Bloom Filter。  1>>**Bloom filter**：更大数据量的有一定误差的用来判断映射是否重复的算法  2>>**Counting Bloom Filter**  … |

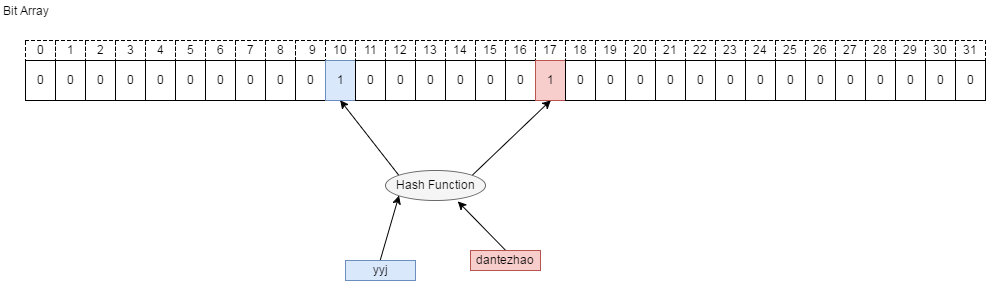
|  |
| --- |
| **应用实例** |
| **1>>使用位图法判断整形数组是否存在重复**  判断集合中是否存在重复元素是常见编程任务之一，当集合中数据量比较大时我们通常希望少进行几次扫描，这时双重循环法就不可取了。位图法比较适合于这种情况，它的做法是按照集合中最大元素max创建一个长度为max+1的新数组，然后再次扫描原数组，遇到几就给新数组的第几位置上1（如遇到 5就给新数组的第六个元素置1），这样下次再遇到5想置位时发现新数组的第六个元素已经是1了，这说明这次的数据肯定和以前的数据存在着重复。这种给新数组初始化时置零其后置一的做法类似于位图的处理方法故称位图法。它的运算次数最坏的情况为2N。如果已知数组的最大值即能事先给新数组定长的话效率还能提高一倍。 |
| **2>>在2.5亿个整数中找出不重复的整数（注：内存不足以容纳这2.5亿个整数）**  **2.1>>一个序列里除了一个元素，其他元素都会重复出现3次，设计一个时间复杂度与空间复杂度最低的算法，找出这个不重复的元素。**  解法一：将bit-map扩展一下，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）。依次扫描这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。所描完事后，查看bitmap，把对应位是01的整数输出即可。  或者我们不用2bit来进行表示，我们用两个bit-map即可模拟实现这个2-bitmap，都是一样的道理。  ==========================================================  解法二：分治法。 |
| **3>>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。**  号码最大的数是99999999，用1亿个bit即可存储，遍历文件中的电话号码，依次将相应的二进制位的值置为1，最后统计值为1的二进制位的数量即为不同号码的总数。 |
| **4>>给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？**  bitmap算法就好办多了。申请512M的内存，一个bit位代表一个unsigned int值，读入40亿个数，设置相应的bit位；读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。（Note: unsigned int最大数为2^32 - 1，所以需要2^32 - 1个位，也就是(2^32 - 1) / 8 /10 ^ 9G = 0.5G内存。） |
| >> 爬虫系统中url去重  >> 解决全组合问题 |

|  |
| --- |
| **java中的BitSet简单的实现了BitMap** |
| **package** org.liws;  **import** java.util.BitSet;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  BitSet bitSet = **new** BitSet();    **try** {  bitSet.set(-2);  } **catch** (Exception e) {  System.***err***.println("bitset不能存储负数");  }  bitSet.set(3);  bitSet.set(4);  System.***out***.println("3是否存在：" + bitSet.get(3));  System.***out***.println("5是否存在：" + bitSet.get(5));  }  /\* out:  \* bitset不能存储负数  \* 3是否存在：true  \* 5是否存在：false  \*/  } |

下面给出一个BitMap简单实现及相应应用的代码示例：

|  |
| --- |
| **package** org.liws.util;  **public** **class** BitMap {    /\*\*  \* 保存数据的连续内存，1个byte能存储8个数据。  \* 7 6 5 4 3 2 1 0  \* 15 14 13 12 11 10 9 8  \* 23 22 21 20 19 18 17 16  \* ...  \*/  **private** **byte**[] bytes;    /\*\*  \* 这里是简单的Bitmap实现，BitMap比较完备的实现是带有自动扩容处理的。  \* **@param** capacity 能够存储的数据量  \*/  **public** BitMap(**int** capacity){  // capacity个数据需要多少个bit呢，capacity/8 + 1  bytes = **new** **byte**[(capacity >> 3) + 1];  }    /\*\*  \* 通过num/8来得到存储num的二进制位所在的字节在bytes数组中的索引  \* **@param** num  \* **@return**  \*/  **private** **int** getIndexInByteArray(**int** num) {  **int** arrayIndex = num >> 3; // num / 8  **return** arrayIndex;  }  /\*\*  \*  \* 通过num%8来得到存储num的二进制位在所属的字节中的postion  \* **@param** num  \* **@return** 0~7  \*/  **private** **int** getPostionInTheByte(**int** num) {  **int** position = num & 0x07; // num % 8  **return** position;  }    **public** **void** add(**int** ... nums) {  **for** (**int** num : nums) {  addOne(num);  }  }  /\*\*  \* 添加一个数  \* **@param** num  \*/  **private** **void** addOne(**int** num){  // 要将二进制数A中倒数第三个二进制位的值置1，就需要"A | 0x00000100"  bytes[getIndexInByteArray(num)] |= 1 << getPostionInTheByte(num);  }  /\*\*  \* 数是否存在  \* **@param** num  \* **@return**  \*/  **public** **boolean** contains(**int** num){  // 判断二进制数A中倒数第三个二进制位的值是否为0，就需要看"A & 0x00000100"结果是否等于0  **return** (bytes[getIndexInByteArray(num)] & (1 << getPostionInTheByte(num))) !=0;  }    /\*\*  \* 删除一个数  \* **@param** num  \*/  **public** **void** remove(**int** num){  // 要将二进制数A中倒数第三个二进制位的值置0，就需要 "A & 0x11111011"，即 "A & (~0x00000100)"  bytes[getIndexInByteArray(num)] &= ~(1 << getPostionInTheByte(num));  }  } |
| **package** org.liws.util;  **import** org.junit.Test;  **public** **class** T\_BitMap {  /\*\*  \* 按大小顺序打印正整数数组{7, 2, 13, 1, 55, 78, 99, 66, 0, 33, 45, 71, 85}，注意要求O(n)  \*/  @Test  **public** **void** test\_1() {  **int** capacity = 100;  BitMap bitMap = **new** BitMap(100); // 简单实现没有自动扩容，需要指定容量  bitMap.add(7, 2, 13, 1, 55, 78, 99, 66, 0, 33, 45, 71, 85); // 这一步是O(n)的  **for** (**int** i = 0; i < capacity; i++) { // 遍历也是O(n)的  **if**(bitMap.contains(i)) {  System.***out***.print(i+", ");  }  }  System.***out***.println(); // out: 0, 1, 2, 7, 13, 33, 45, 55, 66, 71, 78, 85, 99,  }    /\*\*  \* 找到正整数数组{7, 2, 13, 1, 55, 78, 99, 66, 0, 33, 45, 71, 85}中未出现的最小整数3，注意要求O(n)  \*/  @Test  **public** **void** test\_2() {  **int** capacity = 100;  BitMap bitMap = **new** BitMap(100);  bitMap.add(7, 2, 13, 1, 55, 78, 99, 66, 0, 33, 45, 71, 85); // 这一步是O(n)的  **for** (**int** i = 0; i < capacity; i++) { // 遍历也是O(n)的  **if**(!bitMap.contains(i)) {  System.***out***.println(i);  **break**;  }  } // out: 3  }  } |

**BitMap 也可以用来表述字符串类型的数据**，但是需要有一层Hash映射，如下图，通过一层映射关系，可以表述字符串是否存在。当然这种方式会有数据碰撞的问题，但可以通过 **Bloom Filter** 做一些优化，Bloom Filter 使用多个 Hash 函数来减少冲突的概率。Bloom filter可以看做是对bit-map的扩展，更大数据量的有一定误差的用来判断映射是否重复的算法。



参考：

[经典算法系列之(一) - BitMap](https://www.jianshu.com/p/6082a2f7df8e)：https://www.jianshu.com/p/6082a2f7df8e

[BitMap 的基本原理和实现](https://cloud.tencent.com/developer/article/1006113)：https://cloud.tencent.com/developer/article/1006113

BitMap算法：https://blog.csdn.net/pipisorry/article/details/62443757