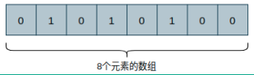
BitMap

定义

BitMap很类似于数组，不过是利用计算机底层使用二进制存储数据的特性所实现的"数组"

对于十进制数1234而言，将其转换成二进制的结果为 0100 1101 0010，一共12位



如果将二进制数中的0看作是False，1看做是True，那么这12位的二进制数就可以看做是一个标志位数组

使用数字来代替数组，可大幅度的节省空间，如果用数组来存储标志位，最少需要8字节空间(Java中的布尔值占一个字节)

但是如果使用BitMap进行存储的话，每一个标志位使用1bit表示，8个标志位只需要8bit，也就是一个字节的占用空间

若使用BitMap来存储值为6的元素，其结果为: 0000 0010，index为6的bit设置为1即可

从上图中可以看出，BitMap只能保存类似于True或者是False的标志位，不能表示其它的数字。所以，BitMap多应用于某元素是否存在于某个海量数据中的判断，进而衍生出对大数据的去重

简单实现

Java实现

基本定义

```java
public class BitMap {
    private char[] bytes;
    private int nbits;

    public BitMap(int nbits) {
        // nbits表示数据集合中最大的值，而不是数据容量
        this.nbits = nbits;
        this.bytes = new char[nbits/16+1];
    }
```

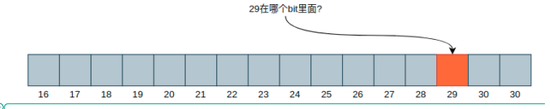
对于char[0]而言，存在16个bit，可以表示0~15，但不包括16，所以当nbits为16时，需要额外的16个bit

设置bit位

假设将数值K于BitMap中进行设置，首先需要找到K理应所在的数组元素下标

通过上图可以直接看出，数值K在数组中的元素索引其实就是 k / 16，向下取整

在获取到数组索引之后，剩下的就是定位具体的bit位



bit = K % 16，表示具体的bit位

将bit位中的数值置位1

移位操作

移位是一个二元运算符，用来将一个二进制数中的每一位全部都向一个方向移动指定位，溢出的部分将被舍弃，而空缺的部分填入0

```
  0001 (十进制1)
<<  3 (左移3位)
= 1000 (十进制8)
```

bytes[n] = bytes[n] | 1 << bit

set

```java
public void set(int k) {
    if (k > nbits) return;
    int index = k / 16;
    int bit = k % 16;
    bytes[index] |= (1 << bit);
}
```

完整实现

get

```java
public boolean get(int k) {
    if (k > nbits) return false;
    int index = k / 16;
    int bit = k % 16;
    return (bytes[index] & (1 << bit)) != 0;
}
```

BitMap的应用

大数据去重

BitMap+哈希函数可构建一占用内存空间极少的去重器，例如爬虫系统中的URL去重

URL去重

首先使用哈希函数将URL进行编码，而后根据某种规则获取其整数表示，在BitMap中设置该值，平均时间复杂度为O(1)

对后续URL进行爬取时，按照同样的规则计算，并于BitMap中获取该值，以决定是否需要进行请求并解析，平均时间复杂度同样为O(1)

但是，BitMap并不是全能的，其内存空间占用完全取决于最大的那个整数

假如需要存储10个数，但是这10个数的最大值为10亿，那么BitMap必须要能够存储10亿这一数字，占用空间将达到120MB，而使用数组只需要40个字节

因此，为了解决此问题，衍生出了布隆过滤器，具体细节可以Google

用于数据重复较少、最大值较小并且数据量庞大的排序

BitMap具有天然的顺序性，所以可以应用于数据的排序，但是正如前面所提到的缺陷: 排序的前提就是序列中的最大值不是特别大

如果最大值为1000亿，但是序列中只有少量的几十万数据的话，使用BitMap将会浪费大量的内存空间

所以，BitMap用于排序适用的场景较少，条件比较苛刻，但是一旦碰上了，将会是一个杀手级的利器

```java
public void sort() {
    for (int i = 0; i <= nbits; i++) {
        if (get(i)) {
            System.out.println(i);
        }
    }
}
```

直接循环取数，对于重复的数据可以维护一个散列表，来记录该元素重复的次数，遍历时将其加上即可

实现标签的交集、并集操作