

数据结构：位图法

一、定义

位图法就是bitmap的缩写。所谓bitmap，就是用每一位来存放某种状态，适用于大规模数据，但数据状态又不是很多的情况。通常是用来判断某个数据存不存在的。在STL中有一个bitset容器，其实就是位图法，引用bitset介绍：

A `bitset` is a special container class that is designed to store bits (elements with only two possible values: 0 or 1, true or false, ...). The class is very similar to a regular array, but optimizing for space allocation: each element occupies only one bit (which is eight times less than the smallest elemental type in C++: `char`). Each element (each bit) can be accessed individually: for example, for a given `bitset` named `mybitset`, the expression `mybitset[3]` accesses its fourth bit, just like a regular array accesses its elements.

二、数据结构

`unsigned int bit[N];`

在这个数组里面，可以存储 $N * \text{sizeof(int)} * 8$ 个数据，但是最大的数只能是 $N * \text{sizeof(int)} * 8 - 1$ 。假如，我们要存储的数据范围为 0-15，则我们只需要使得 $N=1$ ，这样就可以把数据存进去。如下图：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

数据为 `[5, 1, 7, 15, 0, 4, 6, 10]`，则存入这个结构中的情况为

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	1	1	1	1	0	0	1	1

三、相关操作

1，写入数据

定义一个数组：`unsigned char bit[8 * 1024];`这样做，能存 $8K * 8 = 64K$ 个 `unsigned short` 数据。`bit` 存放的字节位置和位位置（字节 0~8191，位 0~7）比如写 1234，字节序：1234/8 = 154；位序：1234 &0b111 = 2，那么 1234 放在 `bit` 的下标 154 字节处，把该字节的 2 号位（0~7）置为 1

字节位置： int nBytePos = 1234/8 = 154;
位位置： int nBitPos = 1234 & 7 = 2;

```
// 把数组的 154 字节的 2 位置为 1
unsigned short val = 1<<nBitPos;
bit[nBytePos] = bit[nBytePos] | val; // 写入 1234 得到arrBit[154]=0b00000100
```

再比如写入 1236 ,
字节位置： int nBytePos = 1236/8 = 154;
位位置： int nBitPos = 1236 & 7 = 4

```
// 把数组的 154 字节的 4 位置为 1
val = 1<<nBitPos; arrBit[nBytePos] = arrBit[nBytePos] | val;
// 再写入 1236 得到arrBit[154]=0b00010100
```

函数实现：

```
#define SHIFT 5
#define MAXLINE 32
#define MASK 0x1F
void setbit(int *bitmap, int i){
    bitmap[i >> SHIFT] |= (1 << (i & MASK));
}
```

2 , 读指定位

```
bool getbit(int *bitmap1, int i){
    return bitmap1[i >> SHIFT] & (1 << (i & MASK));
}
```

四、位图法的缺点

1. 可读性差
2. 位图存储的元素个数虽然比一般做法多，但是存储的元素大小受限于存储空间的大小。位

图存储性质：存储的元素个数等于元素的最大值。比如，1K字节内存，能存储8K个值大小上限为8K的元素。（元素值上限为8K，这个局限性很大！）比如，要存储值为65535的数，就必须要有 $65535/8=8K$ 字节的内存。这就导致了位图法根本不适合存unsigned int类型的数（大约需要 $2^{32}/8=5$ 亿字节的内存）。

3. 位图对有符号类型数据的存储，需要2

位来表示一个有符号元素。这会让位图能存储的元素个数，元素值大小上限减半。比如8K字节内存空间存储short类型数据只能存 $8K*4=32K$ 个，元素值大小范围为-32K~32K。

五、位图法的应用

1、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。首先，将这40亿个数字存储到bitmap中，然后对于给出的数，判断是否在bitmap中即可。

2、使用位图法判断整形数组是否存在重复

遍历数组，一个一个放入bitmap，并且检查其是否在bitmap中出现过，如果没出现放入，否则即为重复的元素。

3、使用位图法进行整形数组排序

首先遍历数组，得到数组的最大最小值，然后根据这个最大最小值来缩小bitmap的范围。这里需要注意对于int的负数，都要转化为unsigned int来处理，而且取位的时候，数字要减去最小值。

4、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

参考的一个方法是：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）。其实，这里可以使用两个普通的Bitmap，即第一个Bitmap存储的是整数是否出现，如果再次出现，则在第二个Bitmap中设置即可。这样的话，就可以使用简单的1-Bitmap了。

六、实现

要求在[这篇文章](#)里面

```
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>

#define SHIFT 5
#define MAXLINE 32
#define MASK 0x1F
```

```
using namespace std;

// w397090770
// wyphao.2007@163.com
// 2012.11.29

void setbit(int *bitmap, int i){
    bitmap[i >> SHIFT] |= (1 << (i & MASK));
}

bool getbit(int *bitmap1, int i){
    return bitmap1[i >> SHIFT] & (1 << (i & MASK));
}

size_t getFileSize(ifstream &in, size_t &size){
    in.seekg(0, ios::end);
    size = in.tellg();
    in.seekg(0, ios::beg);
    return size;
}

char * fillBuf(const char *filename){
    size_t size = 0;
    ifstream in(filename);
    if(in.fail()){
        cerr<< "open " << filename << " failed!" << endl;
        exit(1);
    }
    getFileSize(in, size);

    char *buf = (char *)malloc(sizeof(char) * size + 1);
    if(buf == NULL){
        cerr << "malloc buf error!" << endl;
        exit(1);
    }

    in.read(buf, size);
    in.close();
    buf[size] = '\0';
    return buf;
}

void setBitMask(const char *filename, int *bit){
    char *buf, *temp;
    temp = buf = fillBuf(filename);
    char *p = new char[11];
    int len = 0;
```

```
while(*temp){
    if(*temp == '\n'){
        p[len] = ' ';
        len = 0;
        //cout<<p<<endl;
        setbit(bit, atoi(p));
    }else{
        p[len++] = *temp;
    }
    temp++;
}
delete buf;
}

void compareBit(const char *filename, int *bit, vector &result){
char *buf, *temp;
temp = buf = fillBuf(filename);
char *p = new char[11];
int len = 0;
while(*temp){
    if(*temp == '\n'){
        p[len] = ' ';
        len = 0;
        if(getbit(bit, atoi(p))){
            result.push_back(atoi(p));
        }
    }else{
        p[len++] = *temp;
    }
    temp++;
}
delete buf;
}

int main(){
vector result;
unsigned int MAX = (unsigned int)(1 << 31);
    unsigned int size = MAX >> 5;
int *bit1;

bit1 = (int *)malloc(sizeof(int) * (size + 1));
if(bit1 == NULL){
    cerr<<"Malloc bit1 error!"<<endl;
    exit(1);
}
```

```
memset(bit1, 0, size + 1);
setBitMask("file1", bit1);
compareBit("file2", bit1, result);
delete bit1;

cout<<result.size();
sort(result.begin(), result.end());
vector< int >::iterator it = unique(result.begin(), result.end());

ofstream of("result");
ostream_iterator output(of, "Wn");
copy(result.begin(), it, output);

return 0;
}
```

本博客文章除特别声明，全部都是原创！

原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。

本文链接: [【】\(\)](#)