CS216 Algorithm Design - Project1-Zip Compressor

---

name: 廖铭骞

SID: 12012919

Part1 Introduction

Zip 是一种数据文件压缩的格式，由 Phil Katz 发明，zip 压缩是通过重新对指定文件内部的数据进行重新编码之后实现压缩，以此可以减少大文件的传输时的带宽需求，也可以缩短磁盘访问I/O的时间，在得到一个zip 文件之后，可以通过一定的规则进行解压缩，得到压缩前的文件。

本次 project，我使用了 C++ 对zip 进行了实现， 工作主要分为三阶段：第一个阶段先是通过阅读zip 格式的文档了解了zip各个字段编码方式及其作用，然后通过 C++ 的文件I/O操作对相应字段进行正确的填充；第二个阶段是实现 zip 当中的压缩算法，即 deflate 算法，deflate 算法的实现结合了 LZ77 的编码方法以及哈夫曼压缩的编码思想，在使用 LZ77 编码重新编码文件之后，再使用哈夫曼编码进行字节流的再编码，进而得到最终的压缩文件流，填充到相应的字段当中；第三个阶段则是对实现了zip压缩之后的性能进行测试与改进，探究并分析了文件压缩率以及文件压缩时间与文件大小的关系，以及通过改进优化代码提升压缩的性能以及效率的尝试。

Part2 Analysis and Code

Zip 格式

字段解析

对于整体的 zip 文件格式，可以用下列的字段结构进行阐释

```
[local file header 1]
[encryption header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[encryption header n]
[file data n]
[data descriptor n]
[archive decryption header]
[archive extra data record]
[central directory header 1]
.
.
```

```
.
[central directory header n]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
```

从上表可以看出，`ZIP` 文件主要由三部分构成，分别为压缩源文件数据区、核心目录以及核心目录的结束。

一开始可以通过查看一个简单的 zip 文件的内部编码以及结合zip 编码的文档了解到各个字段表示的含义，下图是通过压缩一个只包含了 `Hello` 的文本文件得到的zip格式文件：

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$ ./zip hello.txt hello.zip
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$ xxd hello.zip
00000000: 504b 0304 1400 0000 0800 387b 8e54 8289  PK........8{.T..
00000010: d1f7 0700 0000 0500 0000 0900 0000 6865  ..............he
00000020: 6c6c 6f2e 7478 74f3 48cd c9c9 0700 504b  llo.txt.H.....PK
00000030: 0102 0a00 0a00 0000 0800 387b 8e54 8289  ..........8{.T..
00000040: d1f7 0700 0000 0500 0000 0900 0000 0000  ................
00000050: 0000 0100 2000 0000 0000 0000 6865 6c6c  .... .......hell
00000060: 6f2e 7478 7450 4b05 0600 0000 0001 0001  o.txtPK.........
00000070: 0037 0000 002e 0000 0000 00              .7.........
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$
```

在ZIP当中采用的是低字节优先的编码方式，即在一个字节里面低位优先，所以实际的字节顺序会和以上显示的顺序相反。

本地文件头（**Local file header**）

位于一开始的字段是本地文件头字段，对于每一个需要压缩的文件，zip 都会为它们分别产生一个本地文件头字段。

以该zip文件为例，下表展示了压缩源文件数据区的文件头部分的内容以及含义

字段大小	字段内容	字段含义
4 字节	0x04034b50	标识了本地文件头字段的开始
2 字节	0x0014	规定了解压该文件所需要的最低版本
2字节	0x0000	通用位的标记
2字节	0x0008	压缩方法（0x0008表示使用deflate 压缩）
2字节	0x7b38	压缩文件最后修改时间
2字节	0x548b	压缩文件最后修改日期
4字节	0xf7d18982	文件的CRC32标识
4字节	0x00000007	文件压缩后大小
4字节	0x00000005	文件压缩前大小
2字节	0x0009	文件名长度
2字节	0x0000	扩展区的长度

在文件头部分之后对应字段是文件的名字，由于该zip文件解析的文件名字是`hello.txt`，所以对应字段对应的是其ASCII码值。

文件名字段之后是压缩后的文件内容，在文件内容开始之前，还有3比特的加密头部（encryption header），由于该文件的压缩采用了deflate 的压缩方式，且由于文件比较小，总共只有一个块，所以前三个比特填充为 `0b101`，之后是文件内容压缩之后的字节流。

而对于数据描述符记录（data descriptor），当使用中央目录加密方法时，数据描述符记录不是必需的，但可以使用。如果存在，则通用位字段的第 3 位设置为1来指示其存在，且数据描述符记录的字段中的值必须设置为二进制零。

对于存档解密头字段（archive decryption header)以及存档解密附加字段（Archive Extra Data Record），两个记录的存在是为了支持文档规范当中的中央目录加密功能。 当中央目录结构数据被加密时，解密头以及解密附加数据段必须在加密数据段之前，提供解密相关数据信息。

核心目录（**Central directory**）

之后是 zip 格式的第二部分——核心目录，字段 `0x02014b50`标识着核心目录的开头，下表展示了核心目录部分的内容以及含义

字段大小	字段内容	字段含义
4字节	0x02014b50	核心目录起始标识
2字节	0x000a	压缩使用的PKWare 版本
2字节	0x000a	解压缩需要的PKWare 版本
2字节	0x0000	通用位标记
2字节	0x0008	压缩方法（0x0008表示使用deflate 压缩）
2字节	0x7b38	压缩文件最后修改时间
2字节	0x548b	压缩文件最后修改日期
4字节	0xf7d18982	文件的CRC32标识
4字节	0x00000007	文件压缩后大小
4字节	0x00000005	文件压缩前大小
2字节	0x0009	文件名长度
2字节	0x0000	扩展区的长度
2字节	0x0000	文件注释长度
2字节	0x0000	文件开始位置磁盘的序号
2字节	0x0001	文件内部属性
4字节	0x00000020	文件外部属性
4字节	0x00000000	本地文件头的相对位移

核心目录结束（**End of central directory record**）

Zip 格式的第三部分标识用于标记压缩的目录数据的结束，每个压缩文件必须有且只有一个核心目录结束记录。

的起始由 `0x06054b50` 标识，字段内容以及含义由下表所示

字段大小	字段内容	字段含义
4字节	0x06054b50	核心目录结束字段起始标识
2字节	0x0000	当前磁盘编号
2字节	0x0000	核心目录起始磁盘编号
2字节	0x0001	磁盘记录的核心目录数量
2字节	0x0001	核心目录结构的总数
4字节	0x00000037	核心目录的大小
4字节	0x0000002e	核心目录起始位置相对磁盘起始位置的位移
2字节	0x0000	文件注释长度

时间日期编码

对于文件最后一次修改的时间以及日期的编码转换，可以通过使用 `fstream` 库当中对文件最后一次修改时间的获取方法

```
_NODISCARD inline file_time_type last_write_time(const path& _Path)
```

传入文件的路径作为参数调用，函数将会返回一个含有时间信息的宏，通过返回的宏计算文件最后一次修改时间与系统起始时间的差值，最终通过该差值得到一个含有完整时间日期信息的结构体 `tm* whole_time`

```
struct tm
{
    int tm_sec;   // seconds after the minute - [0, 60] including
leap second
    int tm_min;   // minutes after the hour - [0, 59]
    int tm_hour;  // hours since midnight - [0, 23]
    int tm_mday;  // day of the month - [1, 31]
    int tm_mon;   // months since January - [0, 11]
    int tm_year;  // years since 1900
    int tm_wday;  // days since Sunday - [0, 6]
    int tm_yday;  // days since January 1 - [0, 365]
    int tm_isdst; // daylight savings time flag
};
```

首先是文件时间的转换，由于在 zip 相应字段当中只有两个字节来表示时间信息，其中小时以及分钟都是比较重要的信息，需要完全保留，所以使用5比特来编码小时，使用6比特来编码分钟，这样一来，表示60秒只有剩下的5比特了，所以精度不能精确到每一秒，这也可以看做是不重要项向重要项的在计算机科学当中的让步。这个设计也确实是妙，如果一定要将秒的精度精确到每一秒，则总共需要2个比特来编码时间信息，同时为了字节对齐以及后续数据的整齐，又需要额外的1个比特来进行字节对齐，这样压缩一个文件就需要额外花费一个字节，这样对于一个文件来说是不必要的，因为2秒的精确度在实际应用当中已经足够了，所以最终编码秒数的时候只能对秒数除以2进行保存。

在表示时间的字段当中最高的5比特位保存小时，次高的6比特位保存分钟，最低5位保存秒。通过以下的位运算可以实现时间的编码。

```
        //File last modification time
        size_t s = whole_time->tm_sec / 2;
        size_t m = whole_time->tm_min;
        size_t h = whole_time->tm_hour;
        h <<= 6; // 将小时放到最高位，并左移6位来存放分钟
        h |= m;
        h <<= 5; // 左移5位来存放秒数
        h |= s;
```

对于文件最后修改日期的编码，和时间的编码类似，注意到在 `tm` 结构体当中，年份是从1900年开始计算的，而在 zip 字段当中，年份是从1980年开始计算的，所以在存储编码日期的时候需要将年份再减去80得到从1980年开始计算的年份，由于该结构体当中的月份是从0开始计算的，所以在实际编码的时候需要加上1个月。编码日期的 C++ 代码如下所示。

```
        //File last modification date
        int d = whole_time->tm_mday;
        int mon = whole_time->tm_mon + 1;
        int y = whole_time->tm_year - 80;

        y <<= 4; // 将年份存储在最高位，左移4位，用来存储月份
        y |= mon;
        y <<= 5;// 左移五位用来存储日
        y |= d;
```

文件I/O操作

填充 zip 相应字段涉及到文件字节流的读写，使用 C++ 当中的 `fstream` 库可以实现文件字节流的读写操作。

通过

```
    fstream fin(read_filename);
    ofstream fout(write_filename);
```

分别创建 fstream 对象来管理输入流、创建 ofstream 流来管理输出流，之后将这两个对象与指定的文件关联起来，便可以使用该对象读取以及写入文件。 使用ifstream对象打开文件的时候有可能发生文件不存在指定目录当中或者文件与流关联失败， 即没能成功打开文件的情况，所以在使用该对象关联文件之后要通过使用方法is_open()确认文件是否与流关联成功，如果没有关联成功便要输出错误信息并及时清理流对象，返回 false 结束函数，这样可以避免对空的流对象进行读操作而在后续操作中陷入更大的错误，提升了程序的可靠新，相应代码如下所示

```
if(!fin.is_open()){
        cout << "Could not open the file " << read_filename <<
endl;
        cerr << "Error in line " << __LINE__ << ", in function \""
<< __FUNCTION__ << "\" of file \"" << __FILE__ << "\"" <<endl;
        fin.clear();
        return false;
    }
```

关联成功之后，先通过 `fstream` 对象中的

```
basic_istream& __CLR_OR_THIS_CALL read(_Elem* _Str, streamsize
_Count)
```

方法，将文件当中的数据以字节流的方式读取到一个字符指针 `buffer` 指向的内存当中，注意在实际读取之前，需要通过

```
memset(buffer, 0, sizeof(char) * (length + 1));
```

将新分配的内存清理一遍，防止原来内存的内容影响新文件内容的读取。

在读取完整个文件的内容之后， `fstream` 当中的文件指针将会指向文件的尾部，此时可以通过

```
pos_type __CLR_OR_THIS_CALL tellg()
```

方法通过读取文件指针此时的位置来指示未压缩文件的原始大小。

对于输出 zip 文件的写入，则可以通过

```
basic_ostream& __CLR_OR_THIS_CALL write(const _Elem* _Str,
streamsize _Count)
```

传入需要写入的字节流以及写入的字节数量来进行文件的写入，每写入 `_Count` 个字节，文件指针便会往下移 `_Count` 个字节。

由于压缩后的大小需要在实际压缩完成之后才可以获得，所以在经过需要填写压缩文件大小信息的字段当中，先使用 `tellp()` 方法记录下此时文件指针的位置 `pos`，之后在压缩完成之后再使用 `seekp(pos, ios::beg)` 方法使用该位置作为偏移量回到该字段所在的位置进行重新填写。

在文件读写操作完成之后，使用`close()`方法关闭与当前对象关联的文件，将其与流分离。同时为了防止内存泄漏，还需要将诸如`buffer`这样，指向通过 `new`分配内存的指针所指向的内存进行释放。

LZ77 算法

对于 deflate 算法而言，LZ77是它的核心，因为后续所有的编码优化都是基于 LZ77编码之后的字节流之上，下面对LZ77算法的工作原理进行阐述。

对于一个具有逻辑的文本来说，其中常常会包括比较长串的重复的短语字段，如果不使用任何压缩方式，就相当于把同一段话在第一次出现之后都频繁地复制，这样编码的效率很低，这也是LZ77算法的着眼点，LZ77算法通过使用 距离（distance）+ 长度（length）与 文本（literal）相结合的方式对一大段文本进行编码，通过将大段重复的文本替换成以 距离 + 长度来进行编码。

LZ77 使用一个前向缓冲区以及一个滑动窗口来维护短语信息，可以将滑动窗口内部的字符称为字典，前向缓冲区以及滑动窗口都具有一定的大小，通过不断地把数据载入前向缓冲区，并且会不断遍历前向缓冲区里面的所有字符，并且将该字符与滑动窗口当中的字符进行比对，查看是否有重复的字符：如果该字符没有出现在滑动窗口当中，则直接将该字符放在缓缓冲区里面；如果出现在滑动窗口当中，则遍历所有出现过该字符的地方，找到匹配以该字符为开头的字符串的最长长度，并记录最长匹配字符串的起始位置，之后将在前向缓冲区中匹配的字符串全部放进滑动窗口当中。期间如果滑动窗口满了，就开始不断往右移动，使得滑动窗口的大小保持最大值内。

可以看出LZ77算法的主要就是在前向缓冲区当中不断地寻找字典中字符匹配的最长字符串，从该算法的原理也可以看出来，LZ77算法的压缩率以及压缩时间取决于许多因素，例如，滑动窗口、前向缓冲区的大小以及数据的短语重复程度。这些因素对压缩的影响会在Part4进行探究。同时LZ77算法对数据的压缩率也依赖于匹配的字符串数量以及长度，如果匹配的字符串越多、越长，那么可以减少的编码也就越多。

用LZ77算法压缩数据是非常耗时的，因为要花很多时间寻找窗口中的匹配短语。然而在通常情况下，LZ77的解压缩过程要比霍夫曼编码的解压缩过程耗时要少。LZ77的解压缩过程非常快是因为每个标记都明确地告诉我们在缓冲区中哪个位置可以读取到所需要的符号。事实上，我们最终只从滑动窗口中读取了与原始数据数量相等的符号而已。

下面对LZ77算法的时间复杂度进行分析，由于滑动窗口以及前向缓冲区的大小均为常数，设为 c_1, c_2，且对于每一个位于前向缓冲区的字符，最差的情况是需要在滑动窗口的每个字符都比对一遍是否有最大匹配字符串，同时每一个字节流当中的字节都会进入一遍前向缓冲区，这样的话假设字节的总数是 `n`，那么算法的复杂度就是 $O(n)$.

最长字符串匹配

在 LZ77 算法当中，如果对于前向缓冲区的每一个字符，都需要遍历字典当中的每一字符来确定是否存在字符串匹配以及确定匹配字符串的最大长度，这样的效率是比较低的。并且由于对于每个重复的字符串，都至少需要两个字节来保存距离以及长度的相关编码信息，所以对于每一个前向缓冲区里面的字符来说，只有匹配的长度大于4的时候才进行 LZ77编码转换能更加显著地提升编码的效率，即如果匹配字符串长度小于4，我们直接将它作为普通不匹配字符处理。

此时可以通过构建字符串哈希的形式，在哈希表当中存储以第 `i` 个字符为起始字符的长度为4字符串的哈希值。

哈希计算方式如下

```
size_t HashCode (const std::string &str) {
    size_t h = 0;
    for (size_t i = 0; i < str.size(); ++i){
        h = h * 31 + size_t(str[i]);
    }
    return h;
}
```

对于前向缓冲区里面的每一个字符而言，计算以它为起始字符的长度为4的字符串的哈希值，并在字典当中以 $O(1)$ 的时间进行查找，确认在字典当中是否存在一个长度至少为4的字符串与其哈希值相同。如果不存在，则直接将该字符放入字典当中，滑动窗口做相应的滑动，如果存在这样的哈希值，则说明有很大可能存在匹配的字符串，此时遍历所有哈希值相同的字符，计算匹配字符串的最长长度，如果该长度`len`大于4，则使用 距离 + 长度的方式进行编码，滑动窗口移动 `len`个字节。

在实现方面，使用两个哈希表，一个哈希表用于检索滑动窗口当中是否存在与其相等的哈希值，另一个哈希表用于存储该哈希值对应的队列，队列的元素是哈希值匹配字符的起始位置，从小到大进行排序，遍历队列的时候取出队头元素，并计算和缓冲区字符相匹配的最长字符串长度，每次取出队头元素，在计算完成之后重新入队，这样在一次遍历之后依然可以保持队列的有序性。如果在计算的过程中存在长度至少为4的字符串匹配，则返回匹配的最大长度以及起始字符的位置，否则，返回 -1。

```
/*
@parameter buffer: 缓冲区对应的字符串
@parameter str: 待匹配的字符串
@parameter str_start_pos: 待匹配字符串在原有字节流的起始位置
*/
pair<size_t, size_t> str_match(const string& buffer, const string&
str, int str_start_pos){

    size_t hash = HashCode(str.substr(0, 4));
    if(hash_pos_map.count(hash)){ // 如果缓冲区有匹配的字符串
        queue<int> que = hash_pos_map[hash];
        int que_size = que.size();
        int idx = 0, length = 0;
        for(int i = 0; i < que_size; i++){
            int pos = que.front();
            que.pop();
            int cnt = 0;
            int str_len = str.size();
            while(cnt < str_len && buffer[pos + cnt] == str[cnt] &&
pos + cnt < str_start_pos){
                cnt ++;
            }
            if(cnt >= length){ // 记录最长字符串的起始位置以及长度
                length = cnt;
                idx = pos;
            }
            que.push(pos);
        }
        if(length < 4){ // 只记录长度至少为4 的字符串信息，不满足则返回
{-1, -1}
            return {-1, -1};
        }
        else{
            return {idx, length};
        }
    }else{
        return {-1, -1};
    }
}
```

由于滑动窗口存在大小限制，所以当位于前向缓冲区里面的字符位置距离滑动窗口起始位置距离大于滑动窗口的限制大小时，将哈希表当中起始位置位于滑动窗口起始边界的字符串的哈希值从表中清除（erase），保证每次检索都是在一个常数大小的范围当中检索。每次更新哈希表的时候，都计算以当前位置为起始位置长度为4字符串的哈希值，并放入对应队列当中，如果有匹配的字符串，还需要将位于前向缓冲区的匹配的字符串中以每一个字符为起始字符的长度为4的字符串计算出哈希值，并放入相应的字典队列当中，并将滑动窗口移动相应的长度。

```
void update_hash(string buffer, int i){
    // 遍历滑动窗口之外的元素进行删除
    for(auto iter = pos_hash_map.rbegin(); iter !=
pos_hash_map.rend() && int(iter -> first) < i - 32768; iter++){

        size_t hash = iter -> second;
        queue<int>q = hash_pos_map[hash];
        while(!q.empty() && q.front() < i - 32768){// 遍历匹配的队列
进行滑动窗口之外元素的删除
            q.pop();
        }
        pos_hash_map.erase(iter->first);
    }

    size_t hashcode = HashCode(buffer.substr(i, 4));
    pos_hash_map[i] = hashcode;

    hash_pos_map[hashcode].push(i);

    return;
}
```

静态哈夫曼编码

在使用 LZ77 算法对文件重新编码完成之后，文本信息主要分为3类（文本（literal）、距离（distance）长度。一种简便的表示方法是采用定长的方式编码，对每一种信息都使用8比特进行存储，这样无疑是一种合理的方式。但是在实际压缩文本的过程中，每一类当中的所有元素出现都是有频率的信息的，一般来说，距离或者长度较小的出现频率就会比较大出现的概率更高，因为在一段有逻辑的文本当中，重复的话语一般不会间隔太远。这时候，如果依然对每一个元素都一视同仁地进行编码，会浪费很多不必要的字节去保存相关信息，考虑哈夫曼的压缩方式，就是将频率越高的元素以更加短的码长进行编码，由哈夫曼编码的正确性证明，这样确实可以更加合理高效地分配编码空间。

根据静态哈夫曼编码的算法，在得到3类文本信息之后，可以直接根据字符的 ASCII 码进行编码，由于字符的 ASCII 码位于 144 之后的字符出现频率总体更小，并且匹配字符串的长度很少超过256个字节，所以将文本信息与匹配字符串的长度信息放在一个哈夫曼编码树当中进行编码会比放在两棵编码树上占用的编码空间更小。由于 ASCII 码加上距离的编码位数已经超过了256，所以需要使用9比特来存储码树信息，但是如何合理地分配用于编码信息的比特变成了一个值得思考的问题。

对于文本信息，每一个字符都有对应的 ASCII 码映射，而对于距离而言，如果每一个距离都使用8比特来进行编码，则使用一个最多只包含9比特的码树是不足以做到这一点的。每个距离肯定对应唯一一个码字，使用哈夫曼编码可以得到所有码字，但是因为距离比较多，而码树可以表示的位数有限，可以把距离划分成多个区间，每个区间当做一个整数来看，这个整数称为 length code 。当一个长度落到某个区间，则相当于是出现了相应的 length code，此时可以将多个长度对应于一个length dode，此时长度总数较多，但 length code 可以划分得很少，根据这样的想法，我们只需要对长度的每一个区间进行统一编码，对于同一个区间内部的距离区分，可以使用额外的位数来进行细分。

考虑如果全部使用8比特来表示文本（literal），则表示距离的信息不得不使用9比特来进行编码，对于一些较小的距离而言，这样是不合理的。能不能使用更少的位数来编码常用的距离信息，并且同时满足哈夫曼编码的前缀码（任何一个字符的编码都不能是其他字符编码的前缀）条件呢？

通过观察静态哈夫曼编码树中 literal + length 分配的码字，可以发现编码树优先为较小的距离区间分配了满足条件的7比特的最短距离编码，这样可以大大减少存储小距离需要的空间代价，小距离区间的编码从 0b0000000 到 0b0010111 共23个区间，在7比特分配完之后，考虑到前缀码特性，在 ASCII 码在144 之前的常用字符分配了从 0b0011000 开始到 0b10111111 的8位的比特编码长度，剩下的8比特编码分配给不常用到的、长度大于114的信息，最后为不常用的 ASCII 码在 144 之后的编码分配了 9 比特的编码长度。综合在一棵码树上考虑，具体的编码信息如下表所示：

文本（**LITERAL**）/ 长度（**LENGTH**） 编码树对应区间值	位数	哈夫曼编码
0 ~ 143	8	00110000 ~ 10111111
144 ~ 255	9	110010000 ~ 111111111
256 ~ 279	7	0000000 ~ 0010111
280 ~ 287	8	11000000 ~ 11000111

对字节信息的编码如下所示：

```
void literal_to_bits(ofstream& fout, const unsigned char* chars,
int len){

    for(int i = 0; i < len; i++){
        unsigned char ch = chars[i];
```

```
            int num = ch;
            if(num >= 0 && num <= 143){
                num = (num + 0b00110000);

                emitBits(fout, num, 8);
            }else{

                num -= 144;
                num = (num + 0b110010000) ;

                emitBits(fout, num, 9);
            }
        }

}
```

在完成对文本（literal）以及长度（length）的编码之后，对于距离（distance） 的编码也与之类似，但是由于距离的值相对与长度最大只为 256 来说要大得多，最大可以达到 32768，所以使用单独的一棵码树来对其进行编码，同样地，考虑到距离越大，出现的频率相对于小的距离也会更低，所以和长度的区间划分类似，将距离分为30个大小不等的区间，称为 distance code，当一个距离落到某个区间，则相当于是出现了相应的 distance code，此时可以将多个距离对应于一个编码区间，此时虽然距离总数较多，但距离编码区间可以划分得很少，根据这样的想法，我们只需要对距离的每一个区间进行统一编码，对于同一个区间内部的距离区分，可以使用额外的位数来进行细分，这点和长度的编码如出一辙，每一个区间的 distance code 统一用 5 bit 表示，从 0b00000 到 0b11101。

在 deflate 编码的最后需要输出 0b0000000 ，此字节标识着编码的结束。

具体方法的实现如下所示：

```
/*
@parameter fout: 输出流对象
@parameter buffer: 需要进行lz77 编码的字节流
@return bool: 标识转换是否成功
*/

bool lz77(ofstream& fout, const char * buffer){

    int data_header = 0b110;
    emitBits(fout, data_header, 3); // 标识文件块信息以及 deflate 编码信息的比特
```

```
    size_t max_len = strlen(buffer);

    string buffer_str = buffer;

    string out = "";
    if(!buffer){ // 判断传入空指针的情况
        cerr << "The input char pointer is nullptr" << endl;
        cout << "Error in line " << __LINE__ << " in function " <<
__FUNCTION__ << " of file " << __FILE__ << endl;
        return false;
    }
    for(size_t i = 0; i < max_len;){
        size_t idx = 0, len = 0;
        if(i + 5 >= max_len){// 剩余空间 <= 5, 直接发射
            int len = max_len - i;
            literal_to_bits(fout, buffer + i, len);
            break;

        }

        else if(str_match(buffer_str, buffer_str.substr(i),
i).first != -1){ // 字符串匹配

            idx = str_match(buffer_str, buffer_str.substr(i),
i).first; // 返回最长匹配字符串起始位置的索引
            len = str_match(buffer_str, buffer_str.substr(i),
i).second; // 返回最长匹配字符串的长度
            idx = i-idx;// 将索引转换成为距离

            distance_len_to_bits(fout, idx, len); // 将长度以及距离进
行编码后发射
            for(int j = 0; j < len; j ++){
                update_hash(buffer_str, i + j);
            }

            i += len;
        }
        else{ // 字符串不匹配，直接发射 literal
            literal_to_bits(fout, buffer + i, 1);
            update_hash(buffer_str, i);
            i++;
        }
    }

     //结束
```

```
    emitBits(fout, 0, 7); // 标识文件块结束

    // 字节补齐
    int remain_bit = (total_bit / 8 + 1) * 8 - total_bit;
    if(remain_bit){
        emitBits(fout, 0, remain_bit);
    }

    return true;
}
```

比特发射器

在 deflate 压缩算法当中，对每一个编码完成的信息而言，都需要通过比特的方式发射到字节流当中，但是 C++ 当中并没有支持比特读写的方法，只有进行位运算的操作符。由于支持文件读写的最小单位是字节，即长度至少是8个比特，于是思考使用一个32位的整型变量`bits`来存储每一个等待发送的比特，使用整型变量来记录待发送的比特长度，每当待发送的长度 ≥ 8 时，就将有效的高8比特作为一个字节打出到待写入缓冲区当中，并将`bits`右移8位，等待接收新的比特。

具体实现如下

```
// remain_len represents the length of bits waiting to emit
void emitBytes(ofstream& fout)
{
  while (remain_len >= 8)
  {
    int num = (bits >> (remain_len - 8)) & 0xff;
    char *c = (char *)&num;
    fout.write(c, sizeof(char) * 1);
    remain_len -= 8;
  }
}
/*
@parameter fout: the file stream object
@parameter bit: the binary number that represents bit stream
@parameter l: the length of bit
*/
void emitBits(ofstream& fout, int bit, int l)
{
  bits <<= l;
```

```
  bits |= bit;
  remain_bit += 1;
  emitBytes(fout);
}
```

调试方法

在分成三个阶段实现 zip 的过程中，对于每一个阶段，都需要保证写出来的子模块尽可能减少bug，因此需要不同的策略进行bug 的调试。

第一个阶段是zip 字段的填充，在这一个阶段当中，目标是通过将没有经过压缩的文件填写进 zip 文件的文件内容字段当中，使得一个文件能够被转换成 zip 格式，发现当文本内容比较少的时候，使用系统内部压缩方式对文件压缩也不会使用任何的压缩算法，这为字段比对工作提供了便利。通过依照系统得到的 zip 文件和自己使用代码转换得到 zip 文件的对应字段比对，一边进行填充，一边进行字段含义的理解，最终在理解了每一个字段含义的基础上完成了零压缩转换 zip，完成了计划的第一步。

第二个阶段是实现 LZ77 编码，通过人为地创造一段具有少量重复的文本，观察编码得到的（距离 + 长度）编码是否符合 LZ77 的编码规则，如果能够符合规则，则进入哈夫曼编码，通过哈夫曼编码规则完成对 LZ77 再压缩之后，综合第一个阶段得到的 zip 转换器方法，综合成一个 zip 文件，观察是否能正常解压成为原来的文件，如果不能就还是比对标准压缩文件的对应字段是否编码错误。

在完成小文件压缩之后探究大文件的压缩是否能成功，依然是比对标准产生的 zip 文件对应的字段，完成调试之后第二阶段的工作就基本完成了。

Cmake 管理与运行

由于在此次 project 当中涉及到多个文件，每次编译链接涉及到的文件较多，所以使用 Cmake 管理代码，分别将第一阶段实现的 zip 转换函数以及第二阶段实现的压缩算法封装到两个源文件`zip.cpp` 和 `deflate.cpp`当中。并且将它们作为动态链接库。

```
cmake_minimum_required(VERSION 3.16)
project(zip)

# Enable C++17
set(CMAKE_CXX_STANDARD 17)

set(SOURCE
    src/main.cpp
)

add_library(zipper
    SHARED
        src/zip.cpp
```

```
        src/deflate.cpp
)

target_include_directories(zipper
    PUBLIC
        ${PROJECT_SOURCE_DIR}/include/static
)

add_executable(zip ${SOURCE})

target_link_libraries(zip
    PRIVATE
        zipper
)
```

这里选择作为动态链接库而不是静态库的原因是相对于源文件规模较大，使用动态链接库在多个文件引用同一个库的时候只会存在一个库文件，使用的空间代价更加小，且修改库的时候不需要重新编译文件，对于后续对库代码的优化过程更加友好。

编译完成之后，可以通过在命令行键入需要转换的文件名以及转换完成后的文件名对文件进行转换，转换后的 zip 文件输出到当前目录下

```
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$ cmake .
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/c/大二课程/算法设计与分析/Project1
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$ make
[ 60%] Built target zipper
[100%] Built target zip
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$ ./zip hello.txt hello.zip
lmq@LAPTOP-4MG6A2H2:/mnt/c/大二课程/算法设计与分析/Project1$
```

计时方式分析

由于本程序是在Linux环境下运行，所以使用Linux环境下的计时函数gettimeofday()进行程序 的计时，该计时函数精度较高，可以达到微秒级别。 通过查看计时函数的结构可以发现，该结构体的定义为

```
struct timeval
{
__time_t tv_sec; /* Seconds. */
__suseconds_t tv_usec; /* Microseconds. */
};
```

这个函数获取从1970年1月1日到现在经过的时间和时区（UTC时间），但是按照Linux的官方文 档，该时区已经不再使用，所以在使用的时候传入NULL即可。 在需要计时的函数开头以及结尾分别获取一次当前时间，它们之间的差值即为程序运行的时间。 操作代码如下

```
gettimeofday(&read_t11, NULL);
//function
gettimeofday(&read_t12, NULL);
read_time_use += (read_t12.tv_sec - read_t11.tv_sec) + (double)
(read_t12.tv_usec -
read_t11.tv_usec) / 1000000.0;
```

在本次 project 当中对执行整个 zip 转换、文件 I/O 以及 压缩的时间均进行了测量，所以涉及到三段时间的计算。

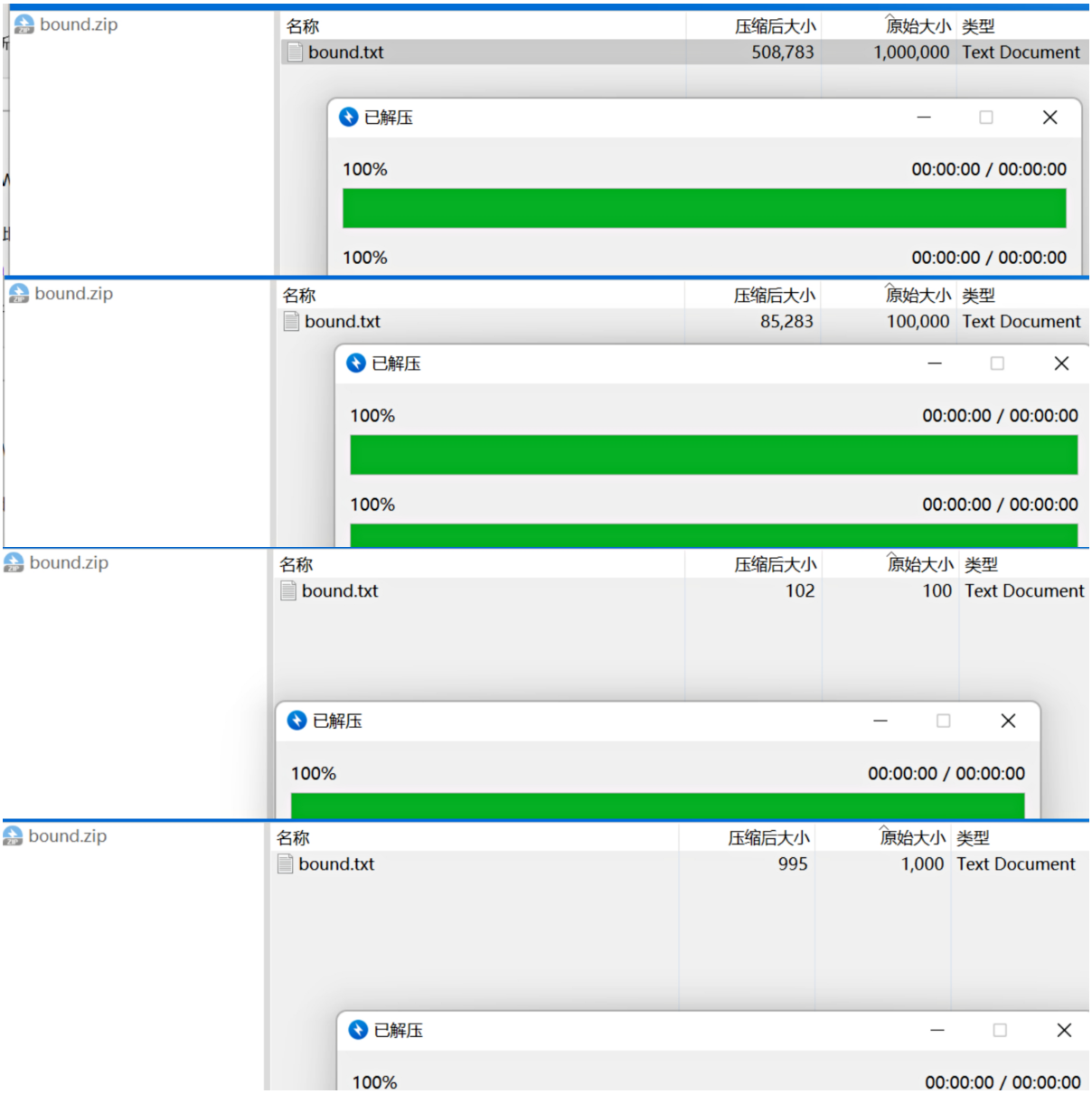
Part3 Experiments and Improvements

代码正确性检验

在完成代码的编写之后，为了验证代码的正确性，编写了一个可以指定大小的随机文本生成器

```
srand(time(NULL));
    ofstream fout("bound.txt");
    for (int i = 0; i < 300; i++)
    {
        int x = rand();
        // printf("x = %d\n", x);
        x %= 25;
        char c = x + 'a';
        // cout << c << endl;
        char* ch = new char[1];
        ch[0] = c;
        fout.write(ch, sizeof(char) * 1);
        delete(ch);
    }
    fout.close();
```

将代码的压缩算法应用于不同大小（1000KB、10000KB、100000KB、1000000KB）的文件，压缩的结果如下所示：



以上文件均能正常解压，以此说明代码的正确性。

LZ77算法参数对压缩影响

滑动窗口大小对压缩影响

通过上文对 LZ77 编码算法的分析可以发现，滑动窗口的大小会影响对前向缓冲区的字符串匹配速度。下面对于同一段大小为 1 MB的随机文本进行压缩，探究当其它变量不变的条件下，LZ77 中滑动窗口的选择对于压缩率以及压缩时间的影响

滑动窗口大小/**BYTE**	压缩 **1MB** 文本所用总时间/**S**
100	95.97
1000	80

滑动窗口大小/BYTE	压缩 1MB 文本所用总时间/S
10000	75.91
100000	62

通过上图可以发现，当滑动窗口越小，压缩的效果相对就会变差，而压缩所需要的时间也会相对增多，这和 LZ77 算法的原理不太相符，按照LZ77算法的原理，当滑动窗口越小，对于前向缓冲区中的每一个字符而言所需要比对的哈希值越少，但是随着滑动窗口的变小，匹配到的字符串最长长度就会受限，在一定程度上也会导致压缩率会降低。

但是在实际实现当中反而发现随着滑动窗口越来越大，压缩所需时间反而越小，回过头去看自己的代码，发现在每一次滑动窗口的更新的时候，都重新遍历了一遍滑动窗口外部的字符，并且逐个删去，这样毫无疑问增加了很多不必要的运行时间，经过思考，可以通过哈希缓删除的方式进行更新，具体优化方法见下文“代码改进”部分，经过改进再次运行程序，得到压缩时间关系如下表所示。

滑动窗口大小/BYTE	压缩 1MB 文本所用总时间/S	压缩后文本大小/BYTE
10	73	999965
100	95	999523
1000	90	996000
10000	77	971830
100000	93	729509
500000	184	422709
1000000	216	230338

从表格中数据分析，以随机生成的文本为例，滑动窗口为10时，可以看做 LZ77 编码几乎没有发挥作用，虽然此时的运行时间是最快的，但是压缩率几乎为0，当滑动窗口不断增加，压缩率不断提高，因为可匹配的字符串范围不断增大，压缩效果也逐步上升，最后当滑动窗口为文件大小是，压缩率可以达到80%，但是运行时间却需要滑动窗口为10时的三倍，这在实际的压缩过程中也是必须考虑的因素。

最终的实验结果符合 LZ77 的原理，即总体来说，滑动窗口越小，压缩文本所需要的时间越少，但是压缩率便会降低，滑动窗口越大，压缩文本所需要的时间越多，但是压缩率便会提高。

所以在实际的压缩过程中，对于滑动窗口的选择要兼顾压缩文件的压缩率以及压缩文件所需要的时间。

文本内容对压缩影响

由于压缩较大文件所需的时间较长，此处以多篇名著文本与和他大小相等的随机生成的文本对比为例，探究文本内容对压缩时间以及压缩率的影响，实验结果如下表所示。

名著/随机文本	压缩前大小	压缩后大小	压缩率	压缩时间/S

名著/随机文本	压缩前大小	压缩后大小	压缩率	压缩时间/S
Alice	152KB	79KB	48.02%	5.24
Alice random	152KB	146KB	4%	2.14
Harry(节选)	611KB	334KB	45.10%	104
Harry(节选) ranodm	611KB	603KB	2%	36
Sherlock(节选)	237KB	135KB	43%	13
Sherlock(节选) random	237KB	234KB	2%	5

通过实验可以发现，在有逻辑的小说文本当中，由于常常会出现重复的人名，话语以及地点名称，所以相对于使用 deflate 算法带来的压缩率会更加可观，对于探究的这几篇文本来说，压缩率均比相等大小的随机文本要高出10倍以上， 这样的代价也很显然，由于经常出现重复的字符串，所以位于前向缓冲区的字符串常常会在滑动窗口中找到对应的匹配，这样会带来很大的压缩时间代价，对于进行试验测试的三篇文文本来说，相比于同等大小的随机文本，压缩的时间至少是两倍以上。

代码优化

哈希缓删除

在开始的实现版本当中，每经过一个前向缓冲区里面的字符，就会遍历一遍哈希表进行滑动窗口的更新，通过实际的实验发现，这样实现的方式效率比较低，对于大文本压缩来说非常慢。此时可以采用哈希缓删除的方法进行解决，即在实际匹配的过程中找到了位于滑动窗口之内的匹配字符串，就先判断一下是否位于当前滑动窗口以内，如果不位于当前的滑动窗口之内，就从哈希记录当中删除对应的元素。通过哈希缓删除的实现机制，有效地提升了程序的运行效率。核心代码如下所示。

```
size_t hash = HashCode(str.substr(0, 4));
    if(hash_pos_map.count(hash)){ // 如果缓冲区有匹配的字符串
        queue<int> que = hash_pos_map[hash]; // 找到匹配的位置队列
        for(int i = 0; i < que_size; i++){
            int pos = que.front();
            que.pop();
            // 如果目标位置位于滑动窗口之外，
            //就直接从对应哈希表当中删除对应元素
            if(pos < str_start_pos - WINDOW_SIZE){
                pos_hash_map.erase(pos);
                continue;
            }
        }
    }
```

最终的实验结果符合 LZ77 的原理，即总体来说，滑动窗口越小，压缩文本所需要的时间越少，但是压缩率便会降低，滑动窗口越大，压缩文本所需要的时间越多，但是压缩率便会提高。

Part4 Difficultis and Solutions

字节对齐问题及解决

在完成 deflate 流编码之后进行字节输出，由于静态哈夫曼编码过程中，对于距离以及长度的编码都是不定长的，所以在 deflate 流的最后有可能会导致字节没有对齐，并且由于除了文件中的 deflate 部分，所有的字节都是对齐的，如果 deflate 流的字段没有对齐，一方面会导致 deflate 流的信息输出不完全，另一方面可能会导致后续字段无法正常被解压器解析，最终都会导致编码出现错误。

所以在文件块写入完成之后需要查看仍然在比特缓存区没有发射的比特，并将剩余的比特填充0至满足一个字节的发送条件，最后将填充完成的字节输出到 zip 文件当中。

CRC校验错误问题及解决

在完成第二阶段工作之后，尝试着将使用代码压缩的文件进行解压，但是发现 CRC 校验错误问题，经过查看静态哈夫曼的编码方式，发现是实现静态哈夫曼的时候没注意正常的 Huffman code 是按大端编码的，而 extra bit 是按照小端编码的，所以导致最终产生的文件无法解析成功，注意到这一点之后，改变了 extra bit 的编码方式，最终解决了CRC 校验错误的问题，成功解压。

中文压缩问题

在调试过程当中，发现压缩中文文本时候会导致解压不能成功的问题，经过原因排查，发现是在读入字节流的时候使用的类型是 `char*` 类型，这样会导致实际处理的过程中中文的字符的值会变成负数，导致静态哈夫曼编码错误，将字节流类型替换成 `unsigned char*` 之后，将中文字符看做无符号处理，这样就解决了中文压缩失败的问题。经过思考，在实际编码的过程中，由于 ASCII 码能够表达的字数量有限，一些特殊的字符如果超过了一个字节能够编码的范围就会导致内容出现负数，最终使得编码不能成功，所以在读入字节流的时候应当使用无符号字符进行读入。

Part 5 Summary

通过这一次 zip 大项目，我从对 zip 完全不了解开始，通过逐步查文档、阅读文件内部编码方式、阅读 github 工程代码等方式一步步地学习，推理、编码、调试，最终成功实现了一个简单版本的 zip 压缩器，用了将近两周的时间进行实现。

开始时，通过了解 zip 的各个字段，认识到对于一个文件而言，需要完成编码解码所需要的信息不只是文件内容而已，还要将该文件所在目录的位置、所在目录位于磁盘的位置、文件压缩的时间日期、文件压缩前后大小以及校验码相关的信息全部存储包装起来，只有这样在最后才能被操作系统识别并成功解压，这个过程让我更加透彻地了解到文件的编码方式以及组成结构。

在实现的过程当中，最惊奇的是 deflate 算法对哈夫曼编码的透彻运用，在完成LZ77算法对原有字节流的重新编排之后，依然可以把距离，长度，文本作为一个新的集合进行重新编码压缩，将使用频率最高的元素使用最短的比特位进行编码，并且不断地对现有的编码进行压缩，从而达到压缩效率近乎极致的做法。

在实际转换整个文件的过程中，更加熟悉了 C++ 语言的文件流操作，从微观角度上看，文件本身就是一个字节流，通过操纵这个字节流以及制定相关的规则，可以将文件转换成任何的格式，进行合理的各种压缩，在压缩的过程中，也要时刻注意字节对齐，不然就可能使得文件信息遗漏导致不能正常解码。

在完成 deflate 转换阶段之后，也进行了一些压缩相关的实验，通过观察实际运行的时间以及压缩前后文件大小的变化，发现对于有逻辑的文本而言，由于常常会出现重复的人名，话语以及地点名称，所以使用 deflate 算法带来的压缩率会更加可观，对于探究的这几篇小说文本来说，压缩率均比相等大小的随机文本要高出10倍以上，并且压缩率越高，所需的时间代价也更高，这在一定程度上也是计算机科学当中的“时间换空间”思想的体现，空间节省的越多，时间的代价便更加昂贵，LZ77 滑动窗口的选择也体现了这一点，所以在实际使用 deflate 算法的过程中，要做得更好，还需要对整体文件的重复文本做一个整体的评估再确定时间和空间权衡之后的选择。

在本次项目中，也通过 Cmake 来管理代码，将源文件作为动态链接库使用，使得更新代码以及管理代码更加便捷。

Part 6 Reference

- ZIP 维基百科 https://en.wikipedia.org/wiki/ZIP_(file_format)
- ZIP 编码格式文档　https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT
- LZ77算法 https://cloud.tencent.com/developer/news/279864
- 静态哈夫曼编码表格来源 https://github.com/ebiggers/libdeflate/blob/master/lib/deflate_compress.c