

RLE压缩算法

RLE压缩算法是对连续的字节进行压缩, 把数据分为连续的相同字节和连续的不同字节, 并在前面加上连续部分的长度. 可以预见, 这种压缩算法在随机数据中表现并不会很好, 甚至可能适得其反, 但是对于重复数据比较多的文件会很有效.

因为记录连续部分长度的只有一个字节, 并且首位需要标记是重复还是非重复字节, 所以允许的最长字节数仅为127. 当连续部分长度超过127时, 需要强制分成两段.

在大文件压缩解压时, 如果一次将文件全部读入, 会产生大量的内存消耗. 对此我采用了一种分块处理的策略, 文件会每10MB一次的读进内存. 这样的策略的问题就是一段跨过两个块的数据会比较难处理.

对于压缩, 这样的问题可以忽略, 完全可以在一个块结束时强制结束这一个连续字节段, 再下一个块的开始再开始新的字节段.

但是对于解压, 这个问题就比较棘手, 完全有可能出现一段连续的数据跨在了两块之间, 甚至可能标志字节在一块内存的最后, 而信息字节在下一个内存块的开始. 这给编码带来了不小的挑战.

为了方便解决这个问题, 我对读入一个字节这个操作进行了封装, 在这个函数里处理分块的关系, 并且每次返回读入的下一个字节. 这样在解压过程中就可以不用考虑分块的逻辑, 安心处理解压的问题.

```
//读入下一个字节, 处理分块等问题
bool getnxtbyte(ifstream& ins, unsigned char &ch)
{
    static vector<unsigned char> buf;
    static int tip = 0;
    static int ttip = 0;
    if(tip >= buf.size())
    {
        cout << (ttip++) * TRUNK_SIZE << "Bytes" << endl;
        buf.clear();
        tip = 0;
        int cnt = 0;
        while (cnt < TRUNK_SIZE && ins.read(reinterpret_cast<char*>(&ch),
1)) buf.push_back(ch), cnt++;
        if(cnt == 0) return false;
    }
    ch = buf[tip++];
    return true;
}
```

解压时先读入标志字节, 判断是否为重复段, 如果为重复段就把下一字节内容重复对应多次, 否则就直接输出后面的不重复字节.

```

//解压字节
void jy(ifstream& ins, ofstream& ous)
{
    int tip = 0;
    unsigned char ch;
    while(getnxtbyte(ins, ch))
    {
        if(((1 << 7) & ch) > 0)
        {
            ch -= (1 << 7);
            unsigned char tch;
            getnxtbyte(ins, tch);
            for(int i = 0;i < (int)ch; i++)
            {
                ous << tch;
            }
        } else{
            for(int i = 1;i <= (int)ch; i++)
            {
                unsigned char tch;
                getnxtbyte(ins, tch);
                ous << tch;
            }
        }
    }
    ous.flush();
    return ;
}

```

压缩时逻辑比较复杂，需要判断下面两个字节是否重复，如果重复就当作重复段处理，否则当作非重复段处理。

```

//压缩字节
vector<unsigned char> ys(vector<unsigned char>& buf)
{
    auto bufo = vector<unsigned char>();
    int tip = 0;
    while(true)
    {
        if(tip == buf.size()) break;
        int ttip = tip;
        int ctip = 0;
        bufo.push_back(0);
        bool ok = false;
        while(tip+1 < buf.size() && buf[tip] == buf[tip+1] && tip-ttip+1
< MAX_COMP_LENGTH)
        {
            if(!ok)

```

```

    {
        bufo.push_back(buf[tip]);
        ctip++;
    }
    ok = true;
    tip++;
}
if(tip == ttip)
{
    while(tip+1 < buf.size() && buf[tip] != buf[tip+1] && (tip+2
>= buf.size() || buf[tip+1] != buf[tip+2]) && tip-ttip+1 <
MAX_COMP_LENGTH)
    {
        bufo.push_back(buf[tip]);
        ctip++;
        tip++;
    }
}
//cout << bufo.size()-1-(tip-ttip)<< endl;
if(ok) //repeat
{
    bufo[bufo.size()-1-ctip] = (1 << 7) + tip-ttip+1;
} else{
    bufo[bufo.size()-1-ctip] = tip-ttip+1;
}
if(!ok)
    bufo.push_back(buf[tip]);
tip++;
}
return bufo;
}

//对文件进行压缩
void compress(ifstream &ins, ofstream& ous)
{
    int tcnt = 0;
    while(true)
    {
        cout << (tcnt++) * TRUNK_SIZE << "Bytes" << endl;
        vector<unsigned char> indata;
        unsigned char ch;
        int cnt = 0;
        while (cnt < TRUNK_SIZE && ins.read(reinterpret_cast<char*>
(&ch), 1)) indata.push_back(ch), cnt++;

        auto rle = ys(indata);
        for (auto& c : rle)
        {
            ous << c;
        }
    }
}

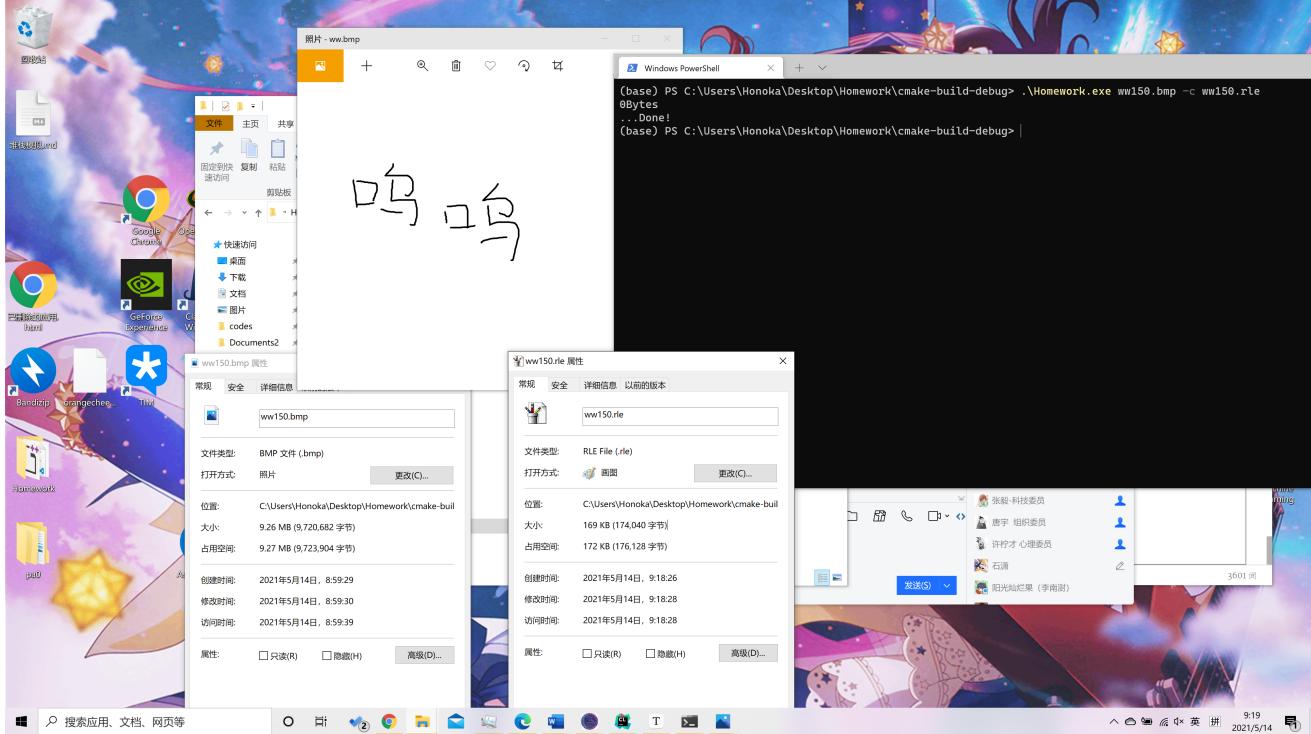
```

```

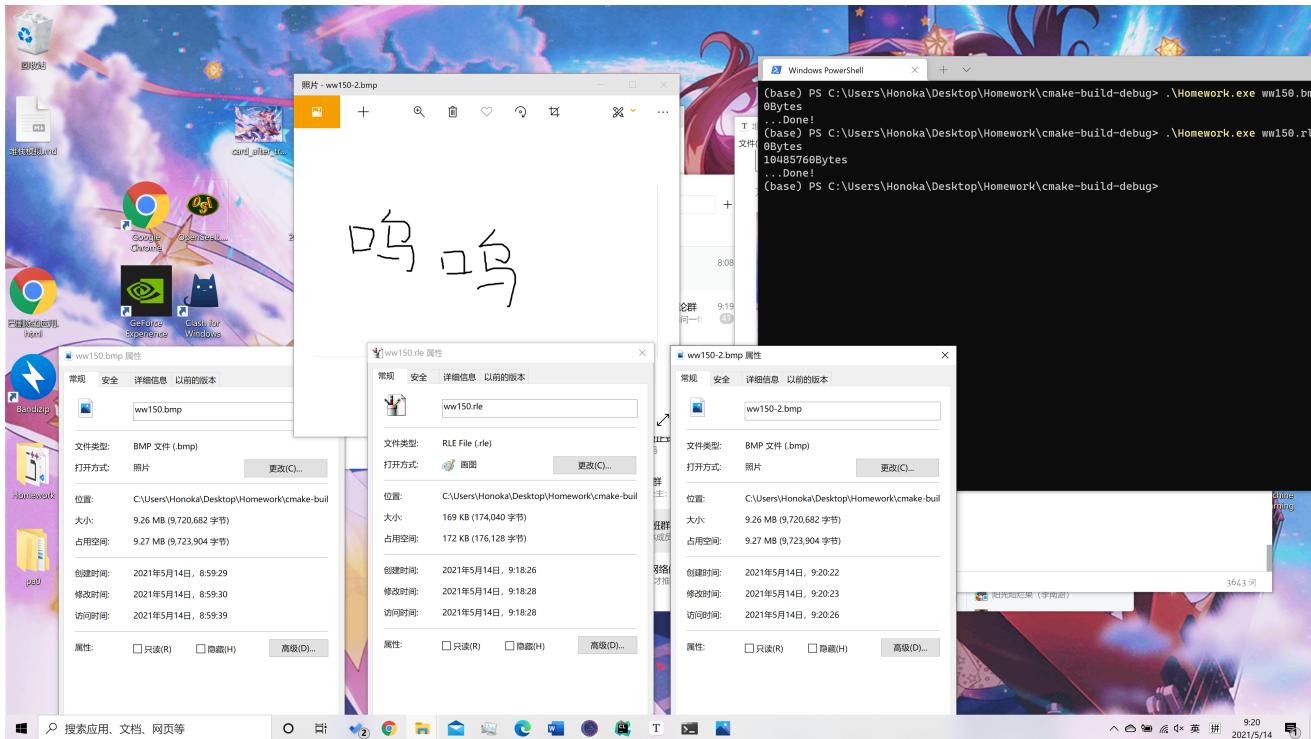
        }
        ous.flush();
        if(cnt < TRUNK_SIZE) break;
    }
}

```

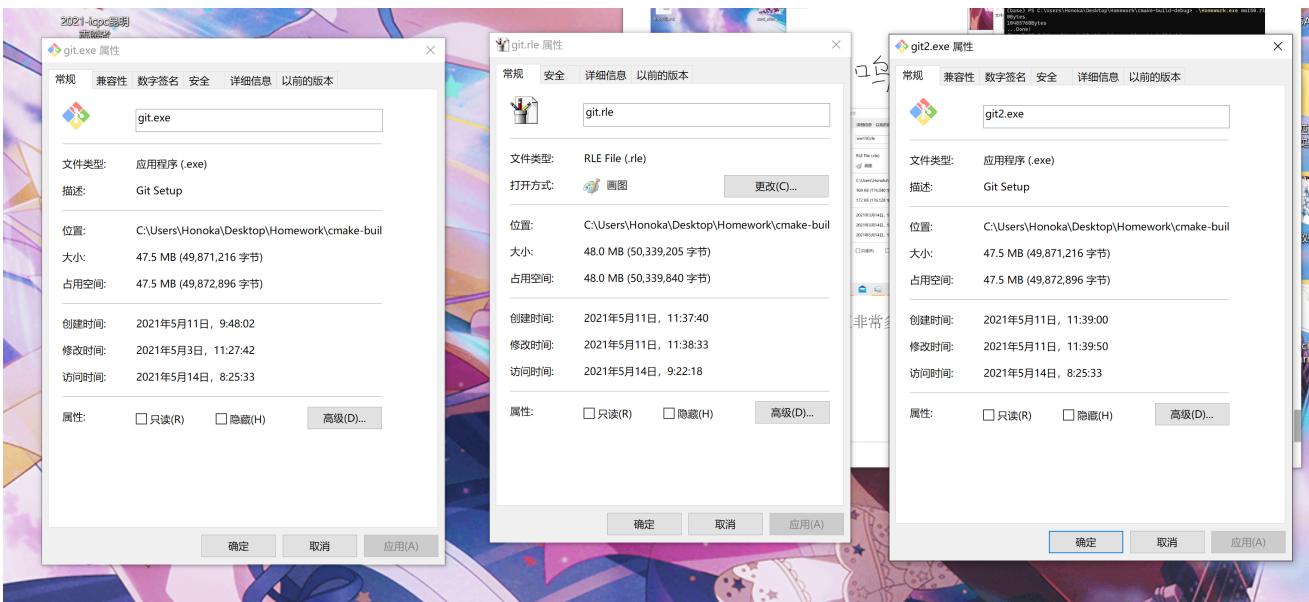
对位图放大后的图片进行压缩,可以看到原来9MB的文件被压缩为了169KB.



解压后,文件并没有被破坏.



这是一个小文件,并且数据重复非常多的情况.对于大随机文件,我也进行了测试.



这是一个47.5MB的exe文件，压缩后变为了48MB，可以看到这种算法对于随机文件的效果很差。

小结

RLE压缩算法属于一种比较简单的压缩算法，仅仅对于连续的重复数据比较多的文件有较好的效果。对于一般的文件压缩效果并不理想，所以还需要更加高效的压缩算法。

封装的思想在这个程序中也得到了体现，为了避免在实现压缩解压算法时考虑复杂的分块逻辑，将读入字节封装成一个函数，简化了代码。