LZ77与huffman的结合

1. 能否采用huffman树直接压缩LZ77的结果呢?

> 通过前文可以知道，LZ77的输出结果中包含：原字符、长度距离对以及标记信息。
>
> 因为huffman树也是一种基于字节的通用压缩算法，因此是可以直接对LZ77的结果再次进行压缩的，但是不太好，因为直接采用huffman树对LZ77的结果进行压缩，会对最终的压缩结果产生很大的影响。
>
> 同学们可以思考下为什么？提示：LZ77的压缩结果中有接近1/8大小为原字符和长度的比特位标记信息

2. Huffman压缩LZ77的结果

> 由于LZ77压缩结果中包含原字符、长度以及距离对，而原字符和长度都是当个字节的，距离占两个字节，因此：**GZIP将原字符和长度采用同一棵huffman树压缩，距离采用另一棵huffman树单独压缩。**

2.1 距离的压缩

> LZ77在查找缓冲区中找匹配时，最长的距离不会超过32K，即最大的距离为32768，即距离的范围是[1, 32768]，距离会非常多，虽然不会达到32768个，但是**如果对于一个比较大的文件进行LZ编码，distance上千还是很正常的，因此会导致huffman树非常大，计算量和内存消耗都会超过当时的硬件条件**，怎么办呢?
>
> GZIP提供了一种非常好的方式，**将distance划分成多个区间，每个区间当做一个整数来看**，该整数称为Distance Code。**当一个distance落到某个区间，则相当于出现了那个Code，虽然distance很多，Distance Code可以划分少一点，即多个distance对应一个Distance Code，最后只需要对Distance Code进行huffman编码即可。得到Code后，Distance Code再根据一定规则扩展出来。**GZIP最终将distance划分成了30个区间，如下图：

Code	Extra bits	Distance	Code	Extra bits	Distance	Code	Extra bits	Distance
0	0	1	10	4	33–48	20	9	1025–1536
1	0	2	11	4	49–64	21	9	1537–2048
2	0	3	12	5	65–96	22	10	2049–3072
3	0	4	13	5	97–128	23	10	3073–4096
4	1	5,6	14	6	129–192	24	11	4097–6144
5	1	7,8	15	6	193–256	25	11	6145–8192
6	2	9–12	16	7	257–384	26	12	8193–12288
7	2	13–16	17	7	385–512	27	12	12289–16384
8	3	17–24	18	8	513–768	28	13	16385–24576
9	3	25–32	19	8	769–1024	29	13	24577–32768

Code表示区间编号[0,29]，总共是30个区间，每个区间容纳distance的个数刚好是2的n次幂，**huffman树只对0~29这30个Code进行编码，得到编码，Extra bits表示distance的编码需要再Code的编码基础上扩展的比特位个数。**比如：0表示不扩展，13表示要扩展13位，因为最大的区间中包含的distance数量为8192个。比如：17~24这个区间的huffman编码为110，因为这个区间有8个整数，于是按照上述表格的规则就可以得到所有distance的编码：

17-----> 110 000

18-----> 110 001

19-----> 110 010

20-----> 110 011

21-----> 110 100

22-----> 110 101

23-----> 110 110

24-----> 110 111

这样就可以将树的高度降低，计算的时间和空间复杂度都降低了，而且扩展起来也比较简单。

2.2 原字符和长度的压缩

原字符表示在LZ77中未匹配的字符，长度表示重复字符串的个数，都占了一个字节，因此GZIP将其压缩合二为一了，即**对于原字符和距离采用同一棵huffman树进行处理。**原字符的范围是[0, 255]，距离是[3, 258](注意：实际存储时候距离减去了个3，即存储的也是[0,255])，那如何区分原字符和长度呢？

GZIP用整数0~255表示原字符，256表示结束标志，即解码以后是256表示解码结束，从257开始表示距离，比如：257表示重复3个字符，258重复4个字符，但**GZIP并没有一直这么一一对应**，而是采用了和distance类似的方式进行分区，**将长度划分成了29个区间**，如下图：

Code	Extra bits	Lengths	Code	Extra bits	Lengths	Code	Extra bits	Lengths
257	0	3	267	1	15,16	277	4	67–82
258	0	4	268	1	17,18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	35–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11,12	275	3	51–58	285	0	258
266	1	13,14	276	3	59–66			

即**原字符和长度的huffman编码的输入元素一共有286个，当解码器接收到一个比特流的时候，首先可以按照literal/length这个码表来解码，如果解出来是0-255，就表示原字符，如果是256，那就表示块结束，如果是257-285之间，则表示length，把后面扩展比特加上形成length后，后面的比特流肯定就表示distance**，因此，实际上通过**一个Huffman码表，对各类情况进行了统一，而不是通过加一个什么标志来区分到底是literal还是重复字符串。**

到此GZIP的主体压缩过程基本出来了，第一步：先是采用LZ77对源文件进行压缩，第二步采用huffman对LZ77的压缩结果进行再次压缩，因为原字符和长度使用一棵huffman树，将其称为huffman码表1，distance对应huffman树称为huffman码表2，而最终的huffman树信息只需要使用码字长度保存即可，称之为CL(Code Length)，即两个码表长度分别为：CL1、CL2。码树记录下来，对原字符的编码比特流称为LIT比特流，对distance编码的比特流称为DIST比特流。按照上面的方法，LZ的编码结果就变成四块：CL1、CL2、LIT比特流、DIST比特流 。

3. LZ77与huffman结合时需要注意问题

1. LZ77压缩结果可以直接统计出字符出现次数，然后直接用来创建huffman树
2. 不是等LZ77全部压缩完成之后才进行huffman压缩，而是分块来进行压缩的，因为交给huffman树的数据不是非常大的情况下，字节的种类越少，生成的huffman编码越短，最终压缩的效果越好。
3. huffman压缩LZ77的结果时，采用了静态编码、动态编码和不压缩总共三种方式，即：静态编码压缩效果更好，则采用静态编码进行压缩，动态编码压缩效果更好则采用动态编码进行压缩，如果不论采用静态还是动态编码都会使文件压缩结果变大，则采用直接存储即不压缩---->本次不考虑这么多，同学们自己可以进行扩展，本文直接采用动态编码进行压缩，因此压缩效率肯定会有一定的影响
4. huffman压缩完LZ77之后，需要保存码字长度用以解压缩，Gzip采用了游程编码对码字长度再次进行了压缩，可以让压缩效率进一步提升，本文暂不考虑，同学们下去可以自己扩展
5. 对于游程编码压缩完成之后的结果，Gzip还对该结果采用huffman进行了进一步压缩，可以使压缩率进一步提高，本文暂不考虑，同学们下去可以自己扩展
6. huffman树的高度不能超过15层(猜测是为了后序的游程编码使用的)，否则树会非常大，因此当huffman树的高度超过15时，需要对树中的部分节点进行合并

4. 扩展：CL的游程编码

编码的长度即CL也是一堆数字，该部分信息理论也可以使用huffman树再次压缩，但是GZIP并没有对其使用huffman树进行压缩，而是使用了游程编码。

游程，即一段完全相同的数的序列。游程编码，即对一段连续相同的数，记录这个数一次，紧接着记录出现了多少个。比如CL序列如下：

4, 4, 4, 4, 4, 3, 3, 3, 6, 6, 6, 6, 6, 6, 6, 6, 6, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2

那么，游程编码的结果为：

4, 16, 01（二进制）, 3, 3, 3, 6, 16, 11（二进制）, 16, 00（二进制）, 17,011（二进制）, 2, 16, 00（二进制）

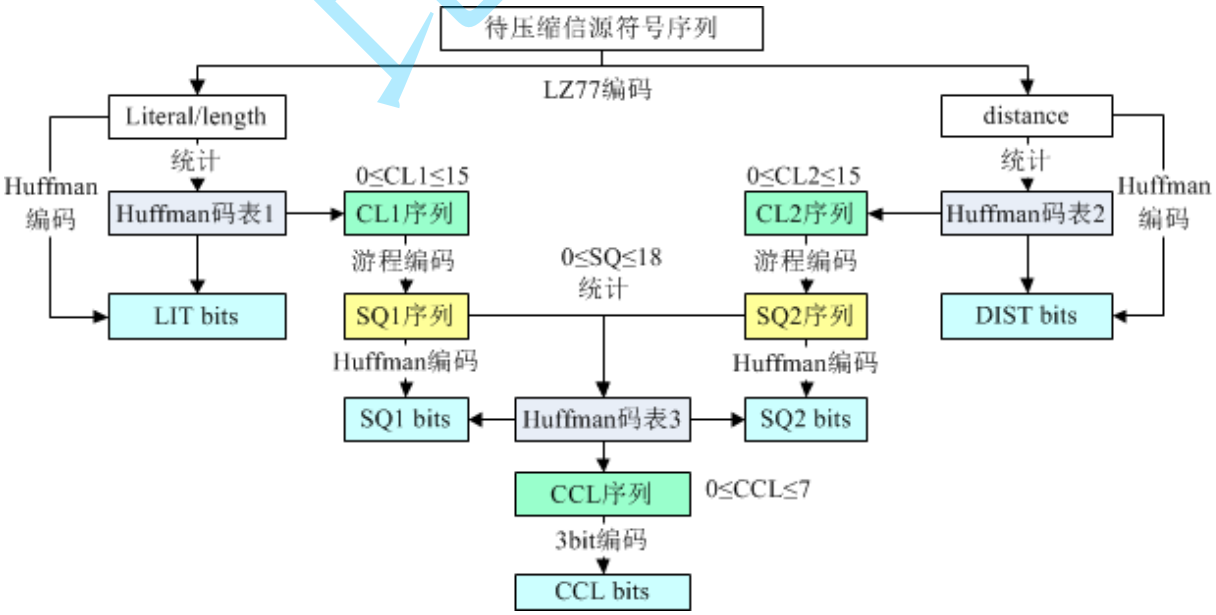
这是什么意思呢？因为CL的范围是0-15，GZIP认为重复出现**2次太短就不用游程编码了，所以游程长度从3开始。用16这个特殊的数表示重复出现3、4、5、6个这样一个游程，分别后面跟着00、01、10、11表示**（实际存储的时候需要低比特优先存储，需要把比特倒序来存，博文的一些例子有时候会忽略这点，实际写程序的时候一定要注意，否则会得到错误结果）。于是**4,4,4,4,4,这段游程记录为4,16,01，也就是说，4这个数，后面还会连续出现了4次。**6,16,11,16,00表示6后面还连续跟着6个6，再跟着3个6；因为**连续的0出现的可能很多，所以用17、18这两个特殊的数专门表示0游程，17后面跟着3个比特分别记录长度为3-10（总共8种可能）的游程；18后面跟着7个比特表示11-138（总共128种可能）的游程。**17,011（二进制）表示连续出现6个0；18,0111110（二进制）表示连续出现62个0。总之记住，0-15是CL可能出现的值，16表示除了0以外的其它游程；17、18表示0游程。因为二进制实际上也是个整数，所以上面的序列用整数表示为：

4, 16, 1, 3, 3, 3, 6, 16, 3, 16, 0, 17, 3, 2, 16, 0

5. GZip的压缩流程

原字符和长度的编码符号总共有286个(256个原字符+1个结束标记+29个长度区间)，distance编码区间总共30个，因此这棵树不会特别深，huffman编码后的码字长度不会特别长，不会超过15，即树的深度不会超过15，因此CL1和CL2这两个序列的任意整数的值的范围是0-15,0表示没有出现，故GZIP对CL1和CL2使用了游程编码。

因为**游程编码之后整数值的范围是0-18，这个序列称之为SQ，因为码字长度有CL1、CL2，因此最后有SQ1和SQ2两组数据。GZIP采用第三个huffman树对SQ1和SQ2再次进行huffman压缩。**通过统计各个整数（0-18范围内）的出现次数，按照相同的思路，对SQ1和SQ2进行了Huffman编码，得到的码流记为SQ1 bits和SQ2 bits。同时，这里又需要记录第三个码表，称为Huffman码表3。同理，这个码表也用相同的方法记录，也等效为一个码长序列，称为CCL。到此GZIP压缩才算真正结束，这个算法命名为Deflate算法：



参考资料：

ZIP压缩算法详细分析及解压实例解释

GZip压缩原理分析

Zip算法源码分析

范式huffman树在文件压缩中的应用

ZIP算法作者短暂而饱受折磨的一生