无损数据压缩是一件奇妙的事情，想一想，一串任意的数据能够根据一定的规则转换成只有原来 1/2 - 1/5 长度的数据，并且能够按照相应的规则还原到原来的样子，听起来真是很酷。  
半年前，苦熬过初学 vc 时那段艰难的学习曲线的我，对 MFC、SDK 开始失望和不满，这些虽然不算易学，但和 DHTML 没有实质上的区别，都是调用微软提供的各种各样的函数，不需要你自己去创建一个窗口，多线程编程时，也不需要你自己去分配 CPU 时间。我也做过驱动，同样，有DDK（微软驱动开发包），当然，也有 DDK 的“参考手册”，连一个最简单的数据结构都不需要你自己做，一切都是函数、函数……   
微软的高级程序员编写了函数让我们这些搞应用的去调用，我不想在这里贬低搞应用的人，正是这些应用工程师连接起了科学和社会之间的桥梁，将来可以做销售，做管理，用自己逐渐积累起来的智慧和经验在社会上打拼。  
但是，在技术上来说，诚实地说，这并不高深，不是吗？第一流的公司如微软、Sybase、Oracle 等总是面向社会大众的，这样才能有巨大的市场。但是他们往往也是站在社会的最顶层的：操作系统、编译器、数据库都值得一代代的专家去不断研究。这些帝国般的企业之所以伟大，恐怕不是“有经验”、“能吃苦”这些中国特色的概念所能涵盖的，艰深的技术体系、现代的管理哲学、强大的市场能力都是缺一不可的吧。我们既然有志于技术，并且正在起步阶段，何必急不可耐地要转去做“管理”，做“青年才俊”，那些所谓的“成功人士”的根底能有几何，这样子浮躁，胸中的规模和格局能有多大？  
  
在我发现vc只是一个用途广泛的编程工具，并不能代表“知识”、“技术”的时候，我有些失落，无所不能的不是我，而是 MFC、SDK、DDK，是微软的工程师，他们做的，正是我想做的，或者说，我也想成为那种层次的人，现在我知道了，他们是专家，但这不会是一个梦，有一天我会做到的，为什么不能说出我的想法呢。  
那时公司做的系统里有一个压缩模块，领导找了一个 zlib 库，不让我自己做压缩算法，站在公司的立场上，我很理解，真的很理解，自己做算法要多久啊。但那时自己心中隐藏的一份倔强驱使我去寻找压缩原理的资料，我完全没有意识到，我即将打开一扇大门，进入一个神奇的“数据结构”的世界。“计算机艺术”的第一线阳光，居然也照到了我这样一个平凡的人的身上。  
  
上面说到“计算机艺术”，或者进一步细化说“计算机编程艺术”，听起来很深奥，很高雅，但是在将要进入专业的压缩算法的研究时，我要请大家做的第一件事情是：忘掉自己的年龄、学历，忘掉自己的社会身份，忘掉编程语言，忘掉“面向对象”、“三层架构”等一切术语。把自己当作一个小孩，有一双求知的眼睛，对世界充满不倦的、单纯的好奇，唯一的前提是一个正常的具有人类理性思维能力的大脑。  
下面就让我们开始一段神奇的压缩算法之旅吧：  
  
  
1. 原理部分：  
　　有两种形式的重复存在于计算机数据中，zip 就是对这两种重复进行了压缩。  
　　一种是短语形式的重复，即三个字节以上的重复，对于这种重复，zip用两个数字：1.重复位置距当前压缩位置的距离；2.重复的长度，来表示这个重复，假设这两个数字各占一个字节，于是数据便得到了压缩，这很容易理解。  
　　一个字节有 0 - 255 共 256 种可能的取值，三个字节有 256 \* 256 \* 256 共一千六百多万种可能的情况，更长的短语取值的可能情况以指数方式增长，出现重复的概率似乎极低，实则不然，各种类型的数据都有出现重复的倾向，一篇论文中，为数不多的术语倾向于重复出现；一篇小说，人名和地名会重复出现；一张上下渐变的背景图片，水平方向上的像素会重复出现；程序的源文件中，语法关键字会重复出现（我们写程序时，多少次前后copy、paste？），以几十 K 为单位的非压缩格式的数据中，倾向于大量出现短语式的重复。经过上面提到的方式进行压缩后，短语式重复的倾向被完全破坏，所以在压缩的结果上进行第二次短语式压缩一般是没有效果的。  
　　第二种重复为单字节的重复，一个字节只有256种可能的取值，所以这种重复是必然的。其中，某些字节出现次数可能较多，另一些则较少，在统计上有分布不均匀的倾向，这是容易理解的，比如一个 ASCII 文本文件中，某些符号可能很少用到，而字母和数字则使用较多，各字母的使用频率也是不一样的，据说字母 e 的使用概率最高；许多图片呈现深色调或浅色调，深色（或浅色）的像素使用较多（这里顺便提一下：png 图片格式是一种无损压缩，其核心算法就是 zip 算法，它和 zip 格式的文件的主要区别在于：作为一种图片格式，它在文件头处存放了图片的大小、使用的颜色数等信息）；上面提到的短语式压缩的结果也有这种倾向：重复倾向于出现在离当前压缩位置较近的地方，重复长度倾向于比较短（20字节以内）。这样，就有了压缩的可能：给 256 种字节取值重新编码，使出现较多的字节使用较短的编码，出现较少的字节使用较长的编码，这样一来，变短的字节相对于变长的字节更多，文件的总长度就会减少，并且，字节使用比例越不均匀，压缩比例就越大。  
　　在进一步讨论编码的要求以及办法前，先提一下：编码式压缩必须在短语式压缩之后进行，因为编码式压缩后，原先八位二进制值的字节就被破坏了，这样文件中短语式重复的倾向也会被破坏（除非先进行解码）。另外，短语式压缩后的结果：那些剩下的未被匹配的单、双字节和得到匹配的距离、长度值仍然具有取值分布不均匀性，因此，两种压缩方式的顺序不能变。  
　　在编码式压缩后，以连续的八位作为一个字节，原先未压缩文件中所具有的字节取值不均匀的倾向被彻底破坏，成为随机性取值，根据统计学知识，随机性取值具有均匀性的倾向（比如抛硬币试验，抛一千次，正反面朝上的次数都接近于 500 次）。因此，编码式压缩后的结果无法再进行编码式压缩。  
　　短语式压缩和编码式压缩是目前计算机科学界研究出的仅有的两种无损压缩方法，它们都无法重复进行，所以，压缩文件无法再次压缩（实际上，能反复进行的压缩算法是不可想象的，因为最终会压缩到 0 字节）。  
　　短语式重复的倾向和字节取值分布不均匀的倾向是可以压缩的基础，两种压缩的顺序不能互换的原因也说了，下面我们来看编码式压缩的要求及方法：  
  
压缩文件无法再次压缩是因为：  
1. 短语式压缩去掉了三个字节以上的重复，压缩后的结果中包含的是未匹配的单双字节，和匹配距离、长度的组合。这个结果当然仍然可能包含三个字节以上的重复，但是概率极低。因为三个字节有 256 \* 256 \* 256 共一千六百多万种可能的情况。  
所以只要把原始文件中“自然存在”的短语式重复倾向压掉就可以了，一千六百万分之一的概率再去压缩没有必要。  
2.编码式压缩利用各个单字节使用频率不一样的倾向，使定长编码变为不定长编码，给使用频率高的字节更短的编码，使用频率低的字节更长的编码，起到压缩的效果。如果把编码式压缩的“结果”按照8位作为1字节，重新统计各字节的使用频率，应该是大致相等的。因为新的字节使用频率是随机的。相等的频率再去变换字节长短是没有意义的，因为变短的字节没有比变长的字节更多。  
所以压掉了原始文件中“自然存在”的单字节使用频率不均匀的倾向后，随机的使用频率再去压缩也失去了意义。  
  
首先，为了使用不定长的编码表示单个字符，编码必须符合“前缀编码”的要求，即较短的编码决不能是较长编码的前缀，反过来说就是，任何一个字符的编码，都不是由另一个字符的编码加上若干位 0 或 1 组成，否则解压缩程序将无法解码。  
看一下前缀编码的一个最简单的例子：  
  
  
符号 编码  
A 0  
B 10  
C 110  
D 1110  
E 11110  
  
有了上面的码表，你一定可以轻松地从下面这串二进制流中分辨出真正的信息内容了：  
  
1110010101110110111100010 - DABBDCEAAB  
  
要构造符合这一要求的二进制编码体系，二叉树是最理想的选择。考察下面这棵二叉树：  
  
　　　　　　　　根(root)  
　　　　　　 0　　|　　 1  
　　　　 　 +-------+--------+  
　　　　0　 |　1　　 0 　|　 1  
　 　 +-----+------+　+----+----+  
　 　 |　　　　　|　|　 　　 |  
　 　 a　　 　　 |　d 　　　 e  
　　　　　0　　|　　1  
　　　　　+-----+-----+  
　　　　　|　　　　 |  
　　　　　b　　 　　c  
  
要编码的字符总是出现在树叶上，假定从根向树叶行走的过程中，左转为0，右转为1，则一个字符的编码就是从根走到该字符所在树叶的路径。正因为字符只能出现在树叶上，任何一个字符的路径都不会是另一字符路径的前缀路径，符合要求的前缀编码也就构造成功了：  
  
a - 00 b - 010 c - 011 d - 10 e - 11  
  
  
接下来来看编码式压缩的过程：  
为了简化问题，假定一个文件中只出现了 a，b，c，d ，e四种字符，它们的出现次数分别是  
a : 6次  
b : 15次  
c : 2次  
d : 9次  
e : 1次  
如果用定长的编码方式为这四种字符编码： a : 000 b : 001 c : 010 d : 011 e : 100  
那么整个文件的长度是 3\*6 + 3\*15 + 3\*2 + 3\*9 + 3\*1 = 99  
  
用二叉树表示这四种编码(其中叶子节点上的数字是其使用次数，非叶子节点上的数字是其左右孩子使用次数之和):  
  
　　　　　　　　 　根  
　　　　　　　　 　 |  
　　　　　　+---------33---------+  
　　　　　　|　　　　　　 　|  
　　 +----32---+　　　　 +----1---+  
　　 |　　　　|　 　　　|　　　 |  
+-21-+ 　 +-11-+ 　 　+--1--+　　   
|　　 | 　 |　　 | 　 　|　　 |  
6　 15 　2　　9 　 　1　　　  
  
（如果某个节点只有一个子节点，可以去掉这个子节点。）  
  
　　　　　　　　 根  
　　　　　　　　　|  
　 　 　　　 +------33------+  
　　　　　　 |　　　　　|  
　 　 +-----32----+ 　 　 1  
　 　 |　　　 　 |  
　 +--21--+ 　+--11--+  
　 |　　　|　 |　　　|  
　 6 　　15　2　 　 9  
  
现在的编码是： a : 000 b : 001 c : 010 d : 011 e : 1 仍然符合“前缀编码”的要求。  
  
第一步：如果发现下层节点的数字大于上层节点的数字，就交换它们的位置，并重新计算非叶子节点的值。  
先交换11和1，由于11个字节缩短了一位，1个字节增长了一位，总文件缩短了10位。  
  
　　　　　　　　　　 根  
　 　 　　　　　　　　|  
　　　　　　 +----------33---------+  
　　　　　　 |　　　　　　　　|  
　　 +-----22----+　　　　　+----11----+  
　　 | 　　 　　|　　　　　|　　　　 |  
+--21--+　　　 1　 　 　　2　　　　 9  
| 　 　 |  
6　　 15  
  
再交换15和1、6和2，最终得到这样的树：  
  
　　　　　　　　　　 根  
　　　　　 　　　　 　|  
　　　　　　 +----------33---------+  
　　　　　　 |　　　　　　　　|  
　　 　 +-----18----+　　　 +----15----+  
　　　 | 　　 　　|　　　 |　　　　 |  
　　+--3--+ 　 　15　　　6　　　　 9  
　　|　 　|  
　　2　　 1  
  
这时所有上层节点的数值都大于下层节点的数值，似乎无法再进一步压缩了。但是我们把每一层的最小的两个节点结合起来，常会发现仍有压缩余地。  
  
第二步：把每一层的最小的两个节点结合起来，重新计算相关节点的值。  
  
在上面的树中，第一、二、四三层都只有一或二个节点，无法重新组合，但第三层上有四个节点，我们把最小的3和6结合起来，并重新计算相关节点的值，成为下面这棵树。  
  
　　　　　　　　　　 根  
　　　　　 　　　　 　|  
　　　　　　 +----------33---------+  
　　　　　　 | 　　　　　　　 |  
　　　 +------9-----+　　　 +----24----+  
　　　 |　 　　 　|　　　 |　　　　 |  
　　 +--3--+　 　 6　　　15　　　　9  
　　 |　　 |  
　　2　　1  
  
然后，再重复做第一步。  
这时第二层的9小于第三层的15，于是可以互换，有9个字节增长了一位，15个字节缩短了一位，文件总长度又缩短了6位。然后重新计算相关节点的值。  
  
　　　　　　　　　　 根  
　　　　 　　　　　　 |  
　　　　　　 +----------33---------+  
　　　　　　 |　　　　　　　　|  
　　　　　　 15　　　　 +----18----+　  
　　　　　　　　　　　　|　　　　|  
　　　　　　　　　+------9-----+　　 9  
　　　　　　　　　|　 　　　 |  
　 　　　　　 　+--3--+　　　6  
　　 　　　　 　|　　 |  
　　　　 　　　 2　　1  
  
这时发现所有的上层节点都大于下层节点，每一层上最小的两个节点被并在了一起，也不可能再产生比同层其他节点更小的父节点了。  
  
这时整个文件的长度是 3\*6 + 1\*15 + 4\*2 + 2\*9 + 4\*1 = 63  
  
这时可以看出编码式压缩的一个基本前提：各节点之间的值要相差比较悬殊，以使某两个节点的和小于同层或下层的另一个节点，这样，交换节点才有利益。  
所以归根结底，原始文件中的字节使用频率必须相差较大，否则将没有两个节点的频率之和小于同层或下层其他节点的频率，也就无法压缩。反之，相差得越悬殊，两个节点的频率之和比同层或下层节点的频率小得越多，交换节点之后的利益也越大。  
  
在这个例子中，经过上面两步不断重复，得到了最优的二叉树，但不能保证在所有情况下，都能通过这两步的重复得到最优二叉树，下面来看另一个例子：  
  
　　　　　　　　　　　　　　　　　　　　　　　　　根  
　　　　　　　　　　　　　　　　　　　　　　　　　｜  
　　　　　　　　　　　　　　＋－－－－－－－－－１９－－－－－－－－＋  
　　　　　　　　　　　　　　｜　　　　　　　　　　　　　　　　　　　｜  
　　　　　　＋－－－－－－１２－－－－－－＋　　　　　　　　　　　　７  
　　　　　　｜　　　　　　　　　　　　　　｜  
　　＋－－－５－－－＋　　　　　　＋－－－７－－－＋  
　　｜　　　　　　　｜　　　　　　｜　　　　　　　｜  
＋－２－＋　　　＋－３－＋　　＋－３－＋　　　＋－４－＋  
｜　　　｜　　　｜　　　｜　　｜　　　｜　　　｜　　　｜  
１　　　１　　　１　　　２　　１　　　２　　　２　　　２  
  
这个例子中，所有上层节点都大于等于下层节点，每一层最小的两个节点结合在了一起，但仍然可以进一步优化：  
  
  
　　　　　　　　　　　　　　　　　　　　　　　　　根  
　　　　　　　　　　　　　　　　　　　　　　　　　｜  
　　　　　　　　　　　　　　＋－－－－－－－－－１９－－－－－－－－＋  
　　　　　　　　　　　　　　｜　　　　　　　　　　　　　　　　　　　｜  
　　　　　　＋－－－－－－１２－－－－－－＋　　　　　　　　　　　　７  
　　　　　　｜　　　　　　　　　　　　　　｜  
　　＋－－－４－－－＋　　　　　　＋－－－８－－－＋  
　　｜　　　　　　　｜　　　　　　｜　　　　　　　｜  
＋－２－＋　　　＋－２－＋　　＋－４－＋　　　＋－４－＋  
｜　　　｜　　　｜　　　｜　　｜　　　｜　　　｜　　　｜  
１　　　１　　　１　　　１　　２　　　２　　　２　　　２  
  
通过最低一层的第４第５个节点对换，第３层的８大于第２层的７。  
到这里，我们得出这样一个结论：一棵最优二叉编码树（所有上层节点都无法和下层节点交换），必须符合这样两个条件：  
１．所有上层节点都大于等于下层节点。  
２．某节点，设其较大的子节点为ｍ，较小的子节点为ｎ，ｍ下的任一层的所有节点都应大于等于ｎ下的该层的所有节点。  
  
当符合这两个条件时，任一层都无法产生更小的节点去和下层节点交换，也无法产生更大的节点去和上层节点交换。  
  
上面的两个例子是比较简单的，实际的文件中，一个字节有256种可能的取值，所以二叉树的叶子节点多达256个，需要不断的调整树形，最终的树形可能非常复杂，有一种非常精巧的算法可以快速地建起一棵最优二叉树，这种算法由D.Huffman（戴·霍夫曼）提出，下面我们先来介绍霍夫曼算法的步骤，然后再来证明通过这么简单的步骤得出的树形确实是一棵最优二叉树。  
  
  
霍夫曼算法的步骤是这样的：  
  
·从各个节点中找出最小的两个节点，给它们建一个父节点，值为这两个节点之和。  
·然后从节点序列中去除这两个节点，加入它们的父节点到序列中。  
  
重复上面两个步骤，直到节点序列中只剩下唯一一个节点。这时一棵最优二叉树就已经建成了，它的根就是剩下的这个节点。  
  
仍以上面的例子来看霍夫曼树的建立过程。  
最初的节点序列是这样的：  
a(6) 　b(15) 　c(2) 　d(9) 　e(1)  
  
把最小的c和e结合起来  
　　　　　　　　　　　　 　　 　 　| (3)  
a(6) 　　b(15) 　　d(9)　　　+------+------+  
　　　　　　　　　　　　　　|　　 　　　|  
　　　　　　　　　　　　　　c　　　　　e  
  
不断重复，最终得到的树是这样的：  
  
　　 　　　 根  
　 　 　　　 |  
　 　+-----33-----+  
　　　|　　　　　|  
　　　15　　 +----18----+　　　   
　　　　　　 | 　 　 　 |  
　　　　　　 9　 +------9-----+  
　 　　 　　　　　|　　 　 　 |  
　　　　　　　　 6 　　 　+--3--+  
　　 　　　　　　　　　 　|　　 |  
　　　　 　　　 　　　　　2　　1  
  
这时各个字符的编码长度和前面我们说过的方法得到的编码长度是相同的，因而文件的总长度也是相同的： 3\*6 + 1\*15 + 4\*2 + 2\*9 + 4\*1 = 63  
  
考察霍夫曼树的建立过程中的每一步的节点序列的变化：  
  
6 　15　2　9　1  
6　 15　9　3  
15　9　 9  
15　18  
33  
  
下面我们用逆推法来证明对于各种不同的节点序列，用霍夫曼算法建立起来的树总是一棵最优二叉树：  
  
对霍夫曼树的建立过程运用逆推法：  
当这个过程中的节点序列只有两个节点时（比如前例中的15和18），肯定是一棵最优二叉树，一个编码为0，另一个编码为1，无法再进一步优化。  
然后往前步进，节点序列中不断地减少一个节点，增加两个节点，在步进过程中将始终保持是一棵最优二叉树，这是因为：  
1.按照霍夫曼树的建立过程，新增的两个节点是当前节点序列中最小的两个，其他的任何两个节点的父节点都大于（或等于）这两个节点的父节点，只要前一步是最优二叉树，其他的任何两个节点的父节点就一定都处在它们的父节点的上层或同层，所以这两个节点一定处在当前二叉树的最低一层。  
2.这两个新增的节点是最小的，所以无法和其他上层节点对换。符合我们前面说的最优二叉树的第一个条件。  
3.只要前一步是最优二叉树，由于这两个新增的节点是最小的，即使同层有其他节点，也无法和同层其他节点重新结合，产生比它们的父节点更小的上层节点来和同层的其他节点对换。它们的父节点小于其他节点的父节点，它们又小于其他所有节点，只要前一步符合最优二叉树的第二个条件，到这一步仍将符合。  
  
这样一步步逆推下去，在这个过程中霍夫曼树每一步都始终保持着是一棵最优二叉树。  
  
由于每一步都从节点序列中删除两个节点，新增一个节点，霍夫曼树的建立过程共需 (原始节点数 - 1) 步，所以霍夫曼算法不失为一种精巧的编码式压缩算法。  
  
  
附：对于 huffman 树，《计算机程序设计艺术》中有完全不同的证明，大意是这样的：  
１．二叉编码树的内部节点（非叶子节点）数等于外部节点（叶子节点）数减１。  
２．二叉编码树的外部节点的加权路径长度（值乘以路径长度）之和，等于所有内部节点值之和。（这两条都可以通过对节点数运用数学归纳法来证明，留给大家做练习。）  
３．对 huffman 树的建立过程运用逆推，当只有一个内部节点时，肯定是一棵最优二叉树。  
４．往前步进，新增两个最小的外部节点，它们结合在一起产生一个新的内部节点，当且仅当原先的内部节点集合是极小化的，加入这个新的内部节点后仍是极小化的。（因为最小的两个节点结合在一起，并处于最低层，相对于它们分别和其他同层或上层节点结合在一起，至少不会增加加权路径长度。）  
５．随着内部节点数逐个增加，内部节点集合总维持极小化。  
  
  
  
  
２．实现部分  
　　如果世界上从没有一个压缩程序，我们看了前面的压缩原理，将有信心一定能作出一个可以压缩大多数格式、内容的数据的程序，当我们着手要做这样一个程序的时候，会发现有很多的难题需要我们去一个个解决，下面将逐个描述这些难题，并详细分析 zip 算法是如何解决这些难题的，其中很多问题带有普遍意义，比如查找匹配，比如数组排序等等，这些都是说不尽的话题，让我们深入其中，做一番思考。  
  
待续。。。。