昵称： 随O心
园龄： 7年11个月
粉丝： 0
关注： 0
+加关注

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

Html(1)
Meta(1)
zip(1)
数据压缩(1)

随笔分类

English
HTML5+CSS
JQUERY
PHP学习
Thinkphp
thinkphp思想
wps(1)

随笔档案

2018年1月(1)
2012年6月(2)

文章分类

51单片机
C/++(1)
HTML5+CSS(2)
JQUERY
PHP(5)
PHP学习
Thinkphp
thinkphp编程思想(5)
编程术语(1)
单片机
数据库

博客园 | 首页 | 新随笔 | 联系 | 订阅 | 管理

ZIP压缩算法详细分析及解压实例解释

最近自己实现了一个ZIP压缩数据的解压程序，觉得有必要把ZIP压缩格式进行一下详细总结，数据压缩是一门通信原理和计算机科学都会涉及到的学科，在通信原理中，一般称为信源编码，在计算机科学里，一般称为数据压缩，两者本质上没啥区别，在数学家看来，都是映射。一方面在进行通信的时候，有必要将待传输的数据进行压缩，以减少带宽需求；另一方面，计算机存储数据的时候，为了减少磁盘容量需求，也会将文件进行压缩，尽管现在的网络带宽越来越高，压缩已经不像90年代初那个时候那么迫切，但在很多场合下仍然需要，其中一个原因是压缩后的数据容量减小后，磁盘访问IO的时间也缩短，尽管压缩和解压缩过程会消耗CPU资源，但是CPU计算资源增长得很快，但是磁盘IO资源却变化得很慢，比如目前主流的SATA硬盘仍然是7200转，如果把磁盘的IO压力转化到CPU上，总体上能够提升系统运行速度。压缩作为一种非常典型的技术，会应用到很多很多场合下，比如文件系统、数据库、消息传输、网页传输等等各类场合。尽管压缩里面会涉及到很多术语和技术，但无需担心，博主尽量将其描述得通俗易懂。另外，本文涉及的压缩算法非常主流并且十分精巧，理解了ZIP的压缩过程，对理解其它相关的压缩算法应该就比较容易了。

1、引子

压缩可以分为无损压缩和有损压缩，有损，指的是压缩之后就无法完整还原原始信息，但是压缩率可以很高，主要应用于视频、话音等数据的压缩，因为损失了一点信息，人是很难察觉的，或者说，也没必要那么清晰照样可以看可以听；无损压缩则用于文件等等必须完整还原信息的场合，ZIP自然就是一种无损压缩，在通信原理中介绍数据压缩的时候，往往是从信息论的角度出发，引出香农所定义的熵的概念，这方面的介绍实在太多，这里换一种思路，从最原始的思想出发，为了达到压缩的目的，需要怎么去设计算法。而ZIP为我们提供了相当好的案例。

尽管我们不去探讨信息论里面那些复杂的概念，不过我们首先还是要从两位信息论大牛谈起。因为是他们奠基了今天大多数无损数据压缩的核心，包括ZIP、RAR、GZIP、GIF、PNG等等大部分无损压缩格式。这两位大牛的名字分别是Jacob Ziv和Abraham Lempel，是两位以色列人，在1977年的时候发表了一篇论文《A Universal Algorithm for Sequential Data Compression》，从名字可以看出，这是一种通用压缩算法，所谓通用压缩算法，指的是这种压缩算法没有对数据的类型有什么限定。不过论文我觉得不用仔细看了，因为博主作为一名通信专业的PHD，看起来也焦头烂额，不过我们后面可以看到，它的思想还是很简单的，之所以看起来复杂，主要是因为IEEE的某些杂志就是这个特点，需要从数学上去证明，这种压缩算法到底有多优，比如针对一个各态历经的随机序列（不用追究什么叫各态历经随机序列），经过这样的压缩算法后，是否可以接近信息论里面的极限（也就是前面说的熵的概念）等等，不过在理解其思想之前，个人认为没必要深究这些东西，除非你要发论文。这两位大牛提出的这个算法称为LZ77，两位大牛过了一年又提了一个类似的算法，称为LZ78，思想类似，ZIP这个算法就是基于LZ77的思想演变过来的，但ZIP对LZ77编码之后的结果又继续进行压缩，直到难以压缩为止。除了LZ77、LZ78，还有很多变种的算法，基本都以LZ开头，如LZW、LZO、LZMA、LZSS、LZR、LZB、LZH、LZC、LZT、LZMW、LZJ、LZFG等等，非常多，LZW也比较流行，GIF那个动画格式记得用了LZW。我也写过解码程序，以后有时间可以再写一篇，但感觉跟LZ77这些类似，写的必要性不大。

ZIP的作者是一个叫Phil Katz的人，这个人算是开源界的一个具有悲剧色彩的传奇人物。虽然二三十年前，开源这个词还没有现在这样风起云涌，但是总有一些具有黑客精神的牛人，内心里面充满了自由，无论他处于哪个时代。Phil Katz这个人是个牛逼程序员，成名于DOS时代，我个人也没有经历过那个时代，我是从Windows98开始接触电脑的，只是从书籍中得知，那个时代网速很慢，拨号使用的是只有几十Kb（比特不是字节）的猫，56Kb实际上是这种猫的最高速度，在ADSL出现之后，这种技术被迅速淘汰。当时记录文件的也是硬盘，但是在电脑之间拷贝文件的是软盘，这个东西我大一还用过，最高容量记得是1.44MB，这还是200X年的软盘，以前的软盘容量具体多大就不知道了，Phil Katz上网的时候还不到1990年，WWW实际上就没出现，浏览器当然是没有的，当时上网干嘛呢？基本就是类似于网管敲各种命令，这样实际上也可以聊天、上论坛不是吗，传个文件不压缩的话肯定死慢死慢的，所以压缩在那个时代很重要。当时有个商业公司提供了一种称为ARC的压缩软件，可以让你在那个时代聊天更快，当然是要付费的，Phil Katz就感觉到不爽，于是写了一个PKARC，免费的，看名字知道是兼容ARC的，于是网友都用

PKARC了，ARC那个公司自然就不爽，把哥们告上了法庭，说牵涉了知识产权等等，结果Phil Katz坐牢了。。。牛人就是牛人， 在牢里面冥思苦想，决定整一个超越ARC的牛逼算法出来，牢里面就是适合思考，用了两周就整出来的，称为PKZIP，不仅免费，而且这次还开源了，直接公布源代码，因为算法都不一样了，也就不涉及到知识产权了，于是ZIP流行开来，不过Phil Katz这个人没有从里面赚到一分钱，还是穷困潦倒，因为喝酒过多等众多原因，2000年的时候死在一个汽车旅馆里。英雄逝去，精神永存，现在我们用UE打开ZIP文件，我们能看到开头的两个字节就是PK两个字符的ASCII码。

2、一个案例的入门思考

好了，Phil Katz在牢里面到底思考了什么？用什么样的算法来压缩数据呢？我们想一个简单的例子：

生，容易。活，容易。生活，不容易。

上面这句话假如不压缩，如果使用Unicode编码，每个字会用2个字节表示。为什么是两个字节呢？Unicode是一种国际标准，把常见各国的字符，比如英文字符、日文字符、韩文字符、中文字符、拉丁字符等等全部制定了一个标准，显然，用2个字节可以最多表示2^16=65536个字符，那么65536就够了吗？生僻字其实是很多的，比如光康熙字典里面收录的汉字就好几万，所以实际上是不够的，那么是不是扩到4个字节？也可以，这样空间倒是变大了，可以收录更多字符，但一方面扩到4个字节就一定保证够吗？另一方面，4个字节是不是太浪费空间了，就为了那些一般情况都不会出现的生僻字？所以，一般情况下，使用2个字节表示，当出现生僻字的时候，再使用4个字节表示。这实际上就体现了信息论中数据压缩基本思想，出现频繁的那些字符，表示得短一些；出现稀少的，可以表示得长些（反正一般情况下也不会出现），这样整体长度就会减小。除了Unicode，ASCII编码是针对英文字符的编码方案，用1个字节即可，除了这两种编码方案，还有很多地区性的编码方案，比如GB2312可以对中文简体字进行编码，Big5可以对中文繁体字进行编码。两个文件如果都使用一种编码方案，那是没有问题的，不过考虑到国际化，还是尽量使用Unicode这种国际标准吧。不过这个跟ZIP没啥关系，纯属背景介绍。

好了，回到我们前面说的例子，一共有17个字符（包括标点符号），如果用普通Unicode表示，一共是17*2=34字节。可不可以压缩呢？所有人一眼都可以看出里面出现了很多重复的字符，比如里面出现了好多次容易（实际上是容易加句号三个字符）这个词，第一次出现的时候用普通的Unicode，第二次出现的“容易。”则可以用（距离、长度）表示，距离的意思是接下来的字符离前面重复的字符隔了几个，长度则表示有几个重复字符，上面的例子的第二个“容易。”就表示为（5,3），就是距离为5个字符，长度是3，在解压缩的时候，解到这个地方的时候，往前跳5个字符，把这个位置的连续3个字符拷贝过来就完成了解压缩，这实际上不就是指针的概念？没有错，跟指针很类似，不过在数据压缩领域，一般称为字典编码，为什么叫字典呢，当我们去查一个字的时候，我们总是先去目录查找这个字在哪一页，再翻到那一页去看，指针不也是这样，指针不就是内存的地址，要对一个内存进行操作，我们先拿到指针，然后去那块内存去操作。所谓的指针、字典、索引、目录等等术语，不同的背景可能称呼不同，但我们要理解他们的本质。如果使用（5,3）这种表示方法，原来需要用6个字节表示，现在只需要记录5和3即可。那么，5和3怎么记录呢？一种方法自然还是可以用Unicode，那么就相当于节省了2个字节，但是有两个问题，第一个问题是解压缩的时候怎么知道是正常的5和3这两个字符，还是这只是一个特殊标记呢？所以前面还得加一个标志来区分一下，到底接下来的Unicode码是指普通字符，还是指距离和长度，如果是普通Unicode，则直接查Unicode码表，如果是距离和长度，则往前面移动一段距离，拷贝即可。第二个问题，还是压缩程度不行，这么一弄，感觉压缩不了多少，如果重复字符比较长那倒是比较划算，因为反正“距离+长度”就够了，但比如这个例子，如果5和3前面加一个特殊字节，岂不是又是3个字节，那还不如不压缩。咋办呢？能不能对（5,3）这种整数进行再次压缩？这里就利用了我们前面说的一个基本原则：出现的少的整数多编一些比特，出现的多的整数少编一些比特。那么，比如3、4、5、6、7、8、9这些距离谁出现得多？谁出现的少呢？谁知道？

压缩之前当然不知道，不过扫描一遍不就知道了？比如，后面那个重复的字符串“容易。”按照前面的规则可以表示为（7,3）,即离前面重复的字符串距离为7，长度为3。（7,3）指着前面跟自己一样那个字符串。那么，为什么不指着第一个“容易。”要指着第二个“容易。”呢？如果指着第一个，那就不是（7,3）了，就是（12，3）了。当然，表示为（12,3）也可以解压缩，但是有一个问题，就是12这个值比7大，大了又怎么了？我们在生活中会发现一些普遍规律，重复现象往往具有局部性。比如，你跟一个人说话，你说了一句话以后，往往很快会重复一遍，但是你不会隔了5个小时又重复这句话，这种特点在文件里面也存在着，到处都是这种例子，比如你在编程的时候，你定义了一个变量int nCount，这个nCount一般你很快就会用到，不会离得很远。我们前面所说的距离代表了你隔了多久再说这句话，这个距离一般不大，既然如此，应该以离当前字符串距离最近的那个作为记录的依据（也就是指向离自己最近那个重复字符串），这样的话，所有的标记都是一些短距离，比如都是3、4、5、6、7而不会是3、5、78、965等等，如果大多数都是一些短距离，那么这些短距离就可以用短一些的比特表示，长一些的距离不太常见，则用一些长一些的比特表示。这样， 总体的表示长度就会减少。好了，我们前面得到了（5,3）、（7、3）这种记录重复的表示，距离有两种：5、7；长度只有1种：3。咋编码？越短越好。

既然表示的比特越短越好，3表示为0、5表示为10、7表示为11，行不行？这样（5,3），(7,3）就只需要表示为100、110，这样岂不是很短？貌似可以，貌似很高效。

但解压缩遇到10这两个比特的时候，怎么知道10表示5呢？这种表示方法是一个映射表，称为码表。我们设计的上面这个例子的码表如下：

3-->0

5-->10

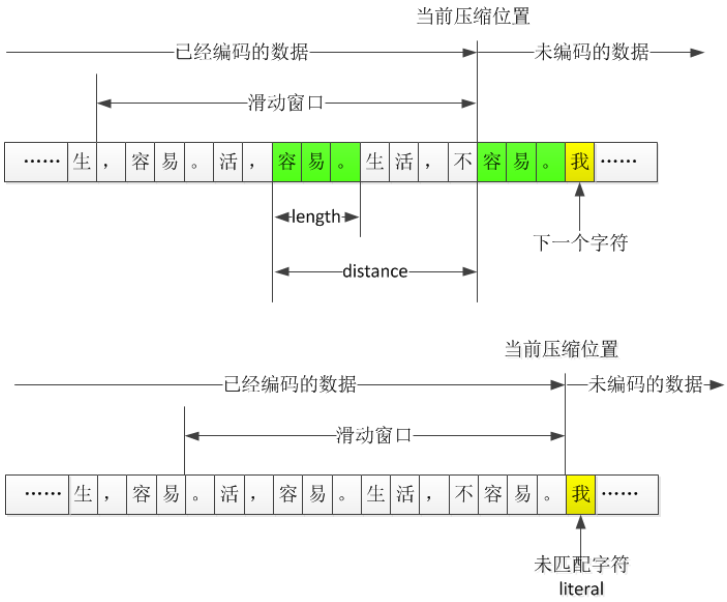
7-->11

这个码表也得传过去或者记录在压缩文件里才行啊，否则无法解压缩，但会不会记录了码表以后整体空间又变大了，会不会起不到压缩的作用？而且一个码表怎么记录？码表记录下来也是一堆数据，是不是也需要编码？码表是否可以继续压缩？那岂不是又需要新的码表？压缩会不会是一个永无止境的过程？作为一个入门级的同学，大概想到这儿就不容易想下去了。

3、ZIP中的LZ编码思想

上面我们说的重复字符串用指针标记记录下来，这种方法就是LZ这两个人提出来的，理解起来比较简单。后面分析（5,3）这种指针标记应该怎么编码的时候，就涉及到一种非常广泛的编码方式，Huffman编码，Huffman大致和香农是一个时代的人，这种编码方式是他在MIT读书的时候提出来的。接下来，我们来看看ZIP是怎么做的。

以上面的例子，一个很简单的示意图如下：



可以看出，ZIP中使用的LZ77算法和前面分析的类似，当然，如果仔细对比的话，ZIP中使用的算法和LZ提出来的LZ77算法其实还是有差异的，不过我建议不用仔细去扣里面的差异，思想基本是相同的，我们后面会简要分析一下两者的差异。LZ77算法一般称为“滑动窗口压缩”，我们前面说过，该算法的核心是在前面的历史数据中寻找重复字符串，但如果要压缩的文件有100MB，是不是从文件头开始找？不是，这里就涉及前面提过的一个规律，重复现象是具有局部性的，它的基本假设是，如果一个字符串要重复，那么也是在附近重复，远的地方就不用找了，因此设置了一个滑动窗口，ZIP中设置的滑动窗口是32KB，那么就是往前面32KB的数据中去找，这个32KB随着编码不断进行而往前滑动。当然，理论上讲，把滑动窗口设置得很大，那样就有更大的概率找到重复的字符串，压缩率不就更高了？初看起来如此，找的范围越大，重复概率越大，不过仔细想想，可能会有问题，一方面，找的范围越大，计算量会增大，不顾一切地增大滑动窗口，甚至不设置滑动窗口，那样的软件可能不可用，你想想，现在这种方式，我们在压缩一个大文件的时候，速度都已经很慢了，如果增大滑动窗口，速度就更慢，从工程实现角度来说，设置滑动窗口是必须的；另一方面，找的范围越大，距离越远，出现的距离很多，也不利于对距离进行进一步压缩吧，我们前面说过，距离和长度最好出现的值越少越好，那样更好压缩，如果出现的很多，如何记录距离和长度可能也存在问题。不过，我相信滑动窗口设置得越大，最终的结果应该越好一些，不过应该不会起到特别大的作用，比如压缩率提高了5%，但计算量增加了10倍，这显然有些得不偿失。

在第一个图中，“容易。”是一个重复字符串，距离distance=5，字符串长度length=3。当对这三个字符压缩完毕后，接下来滑动窗口向前移动3个字符，要压缩的是“我...”这个字符串，但这个串在滑动窗口内没找到，所以无法使用distance+length的方式记录。这种结果称为literal。literal的中文含义是原义的意思，表示没有使用distance+length的方式记录的那些普通字符。literal是不是就用原始的编码方式，比如Unicode方式表示？ZIP里不是这么做的，ZIP把literal认为也是一个数，尽管不能用distance+length表示，但不代表不可以继续压缩。另外，如果“我”出现在了滑动窗口内，是不是就可以用distance+length的方式表示？也不是，因为一个字出现重复，不值得用这种方式表示，两个字呢？distance+length就是两个整数，看起来也不一定值得，ZIP中确实认为2个字节如果在滑动窗口内找到重复，也不管，只有3个字节以上的重复字符串，才会用distance+length表示，重复字符串的长度越长越好，因为不管多长，都用distance+length表示就行了。

这样的话，一段字符串最终就可以表示成literal、distance+length这两种形式了。LZ系列算法的作用到此为止，下面，Phil Katz考虑使用Huffman对上面的这些LZ压缩后的结果进行二次压缩。个人认为接下来的过程才是ZIP的核心，所以我们要熟悉一下Huffman编码。

4、ZIP中的Huffman编码思想

上面LZ压缩结果有三类（literal、distance、length），我们拿出distance一类来举例。distance代表重复字符串离前一个一模一样的字符串之间的距离，是一个大于0的整数。如何对一个整数进行编码呢？一种方法是直接用固定长度表示，比如采用计算机里面描述一个4字节整数那样去记录，这也是可以的，主要问题当然是浪费存储空间，在ZIP中，distance这个数表示的是重复字符串之间的距离，显然，一般而言，越小的距离，出现的数量可能越多，而越大的距离，出现的数量可能越少，那么，按照我们前面所说的原则，小的值就用较少比特表示，大的值就用较多比特表示，在我们这个场景里，distance当然也不会无限大，比如不会超过滑动窗口的最大长度，假如对一个文件进行LZ压缩后，得到的distance值为：

3、6、4、3、4、3、4、3、5

这个例子里，3出现了4次，4出现了3次，5出现了1次，6出现了1次。当然，不同的文件得到的结果不同，这里只是举一个典型的例子，因为只有4种值，所以我们没有必要对其它整数编码。只需要对这4个整数进行编码即可。

那么，怎么设计一个算法，符合3的编码长度最短？6的编码长度最长这种直观上可行的原则（我们并没有说这是理论上最优的方式）呢？

看起来似乎很难想出来。我们先来简化一下，用固定长度表示。这里有4个整数，只要使用2个比特表示即可。于是这样表示就很简单：

00-->3； 01-->4； 10-->5；11-->6。

00、01这种编码结果称为码字，码字的平均长度是2。上面这个对应关系即为码表，在压缩时，需要将码表放在最前面，后面的数字就用码字表示，解码时，先把码表记录在内存里，比如用一个哈希表记录下来，解压缩时，遇到00，就解码为3等等。

因为出现了9个数，所以全部码字总长度为18个比特。（我们暂时不考虑记录码表到底要占多少空间）

想要编码结果更短，因为3出现的最多，所以考虑把3的码字缩短点，比如3是不是可以用1个比特表示，这样才算缩短吧，因为0和1只是二进制的一个标志，所以用0还是1没有本质区别，那么，我们暂定把3用比特0表示。那么，4、5、6还能用0开头的码字表示呢？

这样会存在问题，因为4、5、6的编码结果如果以0开头，那么，在解压缩的时候，遇到比特0，就不知道是表示3还是表示4、5、6了，就无法解码，当然，似乎理论上也不是不可以，比如可以往后解解看，比如假定0表示3的条件下往后解，如果无效则说明这个假设不对，但这种方式很容易出现两个字符串的编码结果是一样的，这个谁来保证？所以，4、5、6都得以1开头才行，那么，按照这个原则，4用1个比特也不行，因为5、6要么以0开头，要么以1开头，就无法编码了，所以我们将4的码字增加至2个比特，比如10，于是我们得到了部分码表：

0-->3；10-->4。

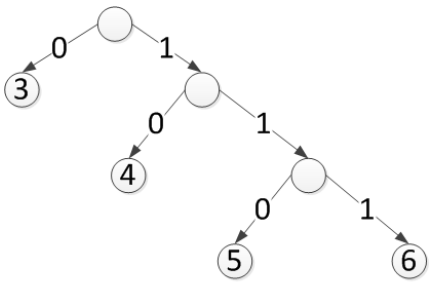
按照这个道理，5、6既不能以0开头，也不能以10开头了，因为同样存在无法解码的问题，所以5应该以11开头，就定为11行不行呢？也不行，因为6就不知道怎么编码了，6也不能以0开头，也不能以10、11开头，那就无法表示了，所以，迫不得已，我们必须把5扩展一位，比如110，那么，6显然就可以用111表示了，反正也没有其他数了。于是我们得到了最终的码表：

0-->3；10-->4；110-->5；111-->6。

看起来，编码结果只能是这样了，我们来算一下，码字的总长度减少了没有，原来的9个数是3、6、4、3、4、3、4、3、5，分别对应的码字是：

0、111、10、0、10、0、10、0、110

算一下，总共16个比特，果然比前面那种方式变短了。我们在前面的设计过程中，是按照这些值出现次数由高到底的顺序去找码字的，比如先确定3，再确定4、5、6等等。按照一个码字不能是另一个码字的前缀这一规则，逐步获得所有的码字。这种编码规则有一个专用术语，称为前缀码。Huffman编码基本上就是这么做的，把出现频率排个序，然后逐个去找，这个逐个去找的过程，就引入了二叉树。不过Huffman的算法一般是从频率由低到高排序，从树的下面依次往上合并，不过本质上没区别，理解思想即可。上面的结果可以用一颗二叉树表示为下图：



这棵树也称为码树，其实就是码表的一种形式化描述，每个节点（除了叶子节点）都会分出两个分支，左分支代表比特0，右边分支代表1，从根节点到叶子节点的一个比特序列就是码字。因为只有叶子节点可以是码字，所以这样也符合一个码字不能是另一个码字的前缀这一原则。说到这里，可以说一下另一个话题，就是一个映射表map在内存中怎么存储，没有相关经验的可以跳过，map实现的是key-->value这样的一个表，map的存储一般有哈希表和树形存储两类，树形存储就可以采用上面这棵树，树的中间节点并没有什么含义，叶子节点的值表示value，从根节点到叶子节点上分支的值就是key，这样比较适合存储那些key由多个不等长字符组成的场合，比如key如果是字符串，那么把二叉树的分支扩展很多，成为多叉树，每个分支就是a,b,c,d这种字符，这棵树也就是Trie树，是一种很好使的数据结构。利用树的遍历算法，就实现了一个有序Map。

好了，我们理解了Huffman编码的思想，我们来看看distance的实际情况。ZIP中滑动窗口大小固定为32KB，也就是说，distance的值范围是1-32768。那么，通过上面的方式，统计频率后，就得到32768个码字，按照上面这种方式可以构建出来。于是我们会遇到一个最大的问题，那就是这棵树太大了，怎么记录呢？

好了，个人认为到了ZIP的核心了，那就是码树应该怎么缩小，以及码树怎么记录的问题。

5、ZIP中Huffman码树的记录方式

分析上面的例子，看看这个码表：

0-->3；10-->4；110-->5；111-->6。

我们之前提过，0和1就是二进制的一个标志，互换一下其实根本不影响编码长度，所以，下面的码表其实是一样的：

1-->3；00-->4；010-->5；011-->6。

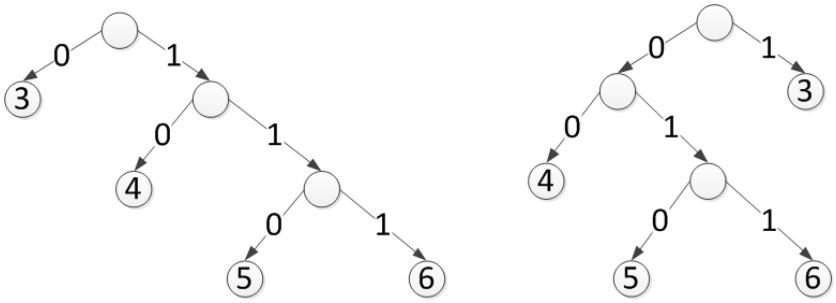
1-->3；01-->4；000-->5；001-->6。

0-->3；11-->4；100-->5；101-->6。

。。。。。

这些都可以，而且编码长度完全一样，只是码字不同而已。

对比一下第一个和第二个例子，对应的码树是这个样子：



也就是说，我们把码树的任意节点的左右分支旋转（0、1互换），也可以称为树的左右互换，其实不影响编码长度，也就是说，这些码表其实都是一样好的，使用哪个都可以。

这个规律暗示了什么信息呢？暗示了码表可以怎么记录呢？Phil Katz当年在牢里也深深地思考了这一问题。

为了体会Phil Katz当时的心情，我们有必要盯着这两棵树思考几分钟：怎么把一颗树用最少的比特记录下来？

Phil Katz当时思考的逻辑我猜是这样的，既然这些树的压缩程度都一样，那干脆使用最特殊的那棵树，反正压缩程度都一样，只要ZIP规定了这棵树的特殊性，那么我记录的信息就可以最少，这种特殊化的思想在后面还会看到。不同的树当然有不同的特点，比如数据结构里面常见的平衡树也是一类特殊的树，他选的树就是左边那棵，这棵树有一个特点，越靠左边越浅，越往右边越深，是这些树中最不平衡的树。ZIP里的压缩算法称为Deflate算法，这棵树也称为Deflate树，对应的解压缩算法称为Inflate，Deflate的大致意思是把轮胎放气了，意为压缩；Inflate是给轮胎打气的意思，意为解压。那么，Deflate树的特殊性又带来什么了？

揭晓答案吧，Phil Katz认为换来换去只有码字长度不变，如果规定了一类特殊的树，那么就只需要记录码字长度即可。比如，一个有效的码表是0-->3；10-->4；110-->5；111-->6。但只需要记录这个对应关系即可：

3　4　5　6

1　2　3　3

也就是说，把1、2、3、3记录下来，解压一边照着左边那棵树的形状构造一颗树，然后只需要1、2、3、3这个信息自然就知道是0、10、110、111。这就是Phil Katz想出来的ZIP最核心的一点：这棵码树用码字长度序列记录下来即可。

当然，只把1、2、3、3这个序列记录下来还不行，比如不知道111对应5还是对应6？

所以，构造出树来只是知道了有哪些码字了，但是这些码字到底对应哪些整数还是不知道。

Phil Katz于是又出现了一个想法：记录1、2、3、3还是记录1、3、2、3，或者3、3、2、1，其实都能构造出这棵树来，那么，为什么不按照一个特殊的顺序记录呢？这个顺序就是整数的大小顺序，比如上面的3、4、5、6是整数大小顺序排列的，那么，记录的顺序就是1、2、3、3。而不是2、3、3、1。

好了，根据1、2、3、3这个信息构造出了码字，这些码字对应的整数一个比一个大，假如我们知道编码前的整数就是3、4、5、6这四个数，那就能对应起来了，不过到底是哪四个还是不知道啊？这个整数可以表示距离啊，距离不知道怎么去解码LZ？

Phil Katz又想了，既然distance的范围是1-32768，那么就按照这个顺序记录。上面的例子1和2没有，那就记录长度0。所以记录下来的码字长度序列为：

0、0、1、2、3、3、0、0、0、0、0、。。。。。。。。。。。。。

这样就知道构造出来的码字对应哪个整数了吧，但因为distance可能的值很多（32768个），但实际出现的往往不多，中间会出现很多0（也就是根本就没出现这个距离），不过这个问题倒是可以对连续的0做个特殊标记，这样是不是就行了呢？还有什么问题？

我们还是要站在时代的高度来看待这个问题。我们明白，每个distance肯定对应唯一一个码字，使用Huffman编码可以得到所有码字，但是因为distance可能非常多，虽然一般不会有32768这么多，但对一个大些的文件进行LZ编码，distance上千还是很

正常的，所以这棵树很大，计算量、消耗的内存都容易超越了那个时代的硬件条件，那么怎么办呢？这里再次体现了Phil Katz对Huffman编码掌握的深度，他把distance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Distance Code。当一个distance落到某个区间，则相当于是出现了那个Code，多个distance对应于一个Distance Code，Distance虽然很多，但Distance Code可以划分得很少，只要我们对Code进行Huffman编码，得到Code的编码后，Distance Code再根据一定规则扩展出来。那么，划分多少个区间？怎么划分区间呢？我们分析过，越小的距离，出现的越多；越大的距离，出现的越少，所以这种区间划分不是等间隔的，而是越来越稀疏的，类似于下面的划分：

1	2	3	4	5,6	7,8	9-12	13-16	17-24	……

1、2、3、4这四个特殊distance不划分，或者说1个Distance就是1个区间；5,6作为一个区间；7、8作为一个区间等等，基本上，区间的大小都是1、2、4、8、16、32这么递增的，越往后，涵盖的距离越多。为什么这么分呢？首先自然是距离越小出现频率越高，所以距离值小的时候，划分密一些，这样相当于一个放大镜，可以对小的距离进行更精细地编码，使得其编码长度与其出现次数尽量匹配；对于距离较大那些，因为出现频率低，所以可以适当放宽一些。另一个原因是，只要知道这个区间Code的码字，那么对于这个区间里面的所有distance，后面追加相应的多个比特即可，比如，17-24这个区间的Huffman码字是110，因为17-24这个区间有8个整数，于是按照下面的规则即可获得其distance对应的码字：

17-->110 000

18-->110 001

19-->110 010

20-->110 011

21-->110 100

22-->110 101

23-->110 110

24-->110 111

这样计算复杂度和内存消耗是不是很小了，因为需要进行Huffman编码的整数一下字变少了，这棵树不会多大，计算起来时间和空间复杂度降低，扩展起来也比较简单。当然，从理论上来说，这样的编码方式实际上将编码过程分为了两级，并不是理论上最优的，把所有distance当作一个大空间去编码才可能得到最优结果，不过还是那句话，工程实现的限制，在压缩软件实现上，我们不能用压缩率作为衡量一个算法优劣的唯一指标，其实耗费的时间和空间同样是指标，所以需要看综合指标。很多其他软件也一样，扩展性、时间空间复杂度、稳定性、移植性、维护的方便性等等是工程上很重要的东西。我没有看过RAR是如何压缩的，有可能是在类似的地方进行了改进，如果如此，那也是站在巨人的肩膀上，而且硬件条件不同，进行一些改进也并不奇怪。

具体来说，Phil Katz把distance划分为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

这个图是我从David Salomon的《Data Compression The Complete Reference》这本书（第四版）中拷贝出来的，下面的有些图也是，如果需要对数据压缩进行全面的了解，这本书几乎是最全的了，强烈推荐。

当然，你要问为什么是30个区间，我也没分析过，也许是复杂度和压缩率经过试验之后的一种折中吧。

其中，左边的Code表示区间的编号，是0-29，共30个区间，这只是个编号，没有特别的含义，但Huffman就是对0-29这30个Code进行编码的，得到区间的码字；

bits表示distance的码字需要在Code的码字基础上扩展几位，比如0就表示不扩展，最大的13表示要扩展13位，因此，最大的区间包含的distance数量为8192个。

Distance一列则表示这个区间涵盖的distance范围。

理解了码树如何有效记录，以及如何缩小码树的过程，我觉得就理解了ZIP的精髓。

6、ZIP中literal和length的压缩方式

说完了distance，LZ编码结果还有两类：literal和length。这两类也利用了类似于distance的方式进行压缩。

前面分析过，literal表示未匹配的字符，我们前面之所以拿汉字来举例，完全是为了便于理解，ZIP之所以是通用压缩，它实际上是针对字节作为基本字符来编码的，所以一个literal至多有256种可能。

length表示重复字符串长度，length=1当然不会出现，因为一个字符不值得用distance+length去记录，重复字符串当然越长越好，Phil Katz（下面还是简称PK了，拷贝太麻烦）认为，length=2也不值得用这种方式记录，还是太短了，所以PK把length最小值认为是3，必须3个以上字符的字符串出现重复才用distance+length记录。那么，最大的length是多少呢？理论上当然可以很长很长，比如一个文件就是连续的0，这个重复字符串长度其实接近于这个文件的实际长度。但是PK把length的范围做了限制，限定length的个数跟literal一样，也只有256个，因为PK认为，一个重复字符串达到了256个已经很长了，概率非常小；另外，其实哪怕超过了256，我还是认为是一段256再加上另外一段，增加一个distance+length就行了嘛，并不影响结果。而且这样做，我想同样也考虑了硬件条件吧。

初看有点奇怪的在于，将literal和length二者合二为一，什么意思呢？就是对这两种整数（literal本质上是一个字节）共用一个Huffman码表，一会儿解释为什么。PK对Huffman的理解我觉得达到了炉火纯青的地步，前面已经看到，后面还会看到。他认为Huffman编码的输入反正说白了就是一个集合的元素就行，无论这个元素是啥，所以多个集合看做一个集合当作Huffman编码的输入没啥问题。literal用整数0-255表示，256是一个结束标志，解码以后结果是256表示解码结束；从257开始表示length，所以257这个数表示length=3，258这个数表示length=4等等，但PK也不是一直这么一一对应，和distance一样，也是把length（总共256个值）划分为29个区间，其结果如下图：

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			

其中的含义和distance类似，不再赘述，所以literal/length这个Huffman编码的输入元素一共285个，其中256表示解码结束标志。为什么要把二者合二为一呢？因为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，就表示未匹配字符，如果是256，那自然就结束，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。

好了，理解了上面的过程，就理解了ZIP压缩的第二步，第一步是LZ编码，第二步是对LZ编码后结果（literal、distance、length）进行的再编码，因为literal/length是一个码表，我称其为Huffman码表1，distance那个码表称为Huffman码表2。前面我们已经分析了，Huffman码树用一个码字长度序列表示，称为CL（Code Length），记录两个码表的码字长度序列分别记为CL1、CL2。码树记录下来，对literal/length的编码比特流称为LIT比特流；对distance的编码比特流称为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。CL1、CL2是码字长度的序列，这个序列说白了就是一堆正整数，因此，PK继续深挖，认为这个序列还应该继续压缩，也就是说，对码表进行压缩。

7、ZIP中对CL进行再次压缩的方法

这里仍然沿用Huffman的想法，因为CL也是一堆整数，那么当然可以再次应用Huffman编码。不过在这之前，PK对CL序列进行了一点处理。这个处理也是很精巧的。

CL序列表示一系列整数对应的码字长度，对于literal/length来说，总共有0-285这么多符号，所以这个序列长度为286，每个符号都有一个码字长度，当然，这里面可能会出现大段连续的0，因为某些字符或长度不存在，尤其是对英文文本编码的时候，非ASCII字符就根本不会出现，length较大的值出现概率也很小，所以出现大段的0是很正常的；对于distance也类似，也可能出现大段的0。PK于是先进行了一下游程编码。在说什么是游程编码之前，我们谈谈PK对CL序列的认识。

literal/length的编码符号总共286个（回忆：256个Literal+1个结束标志+29个length区间），distance的编码符号总共30个（回忆：30个区间），所以这颗码树不会特别深，Huffman编码后的码字长度不会特别长，PK认为最长不会超过15，也就是树的深度不会超过15，这个是否是理论证明我还没有分析，有兴趣的同学可以分析一下。因此，CL1和CL2这两个序列的任意整数值的范围是0-15。0表示某个整数没有出现（比如literal=0x12, length Code=8, distance Code=15等等）。

什么叫游程呢？就是一段完全相同的数的序列。什么叫游程编码呢？说起来原理更简单，就是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即可。David的书中举了这个例子，比如CL序列如下：

4, 4, 4, 4, 4, 3, 3, 3, 6, 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，PK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

我们又看到了一串整数，这串整数的值的范围是0-18。这个序列称为SQ（Sequence的意思）。因为有两个CL1、CL2，所以对应的有两个SQ1、SQ2。

针对SQ1、SQ2，PK用了第三个Huffman码表来对这两个序列进行编码。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，因为至多有0-18个，PK认为树的深度至多为7，于是CCL的范围是0-7。

当得到了CCL序列后，PK决定不再折腾，对这个序列用普通的3比特定长编码记录下来即可，即000代表0,111代表7。但实际上还有一点小折腾，就是最后这个序列如果全部记录，那就需要19*3=57个比特，PK认为CL序列里面CL范围为0-15，特殊的几个值是16、17、18，如果把CCL序列位置置换一下，把16、17、18这些放前面，那么这个CCL序列就很可能最后面跟着一串0（因为CL=14,15这些很可能没有），所以最后还引入了一个置换，其示意图如下，分别表示置换前的CCL序列和置换后的CCL。可以看出，16、17、18对应的CCL被放到了前面，这样如果尾部出现了一些0，就只需要记录CCL长度即可，后面的0不记录。可以继续节省一些比特，不过这个例子尾部置换后只有1个0：

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CCL:	4	5	5	1	5	0	5	0	0	0	0	0	0	0	0	0	2	4	0

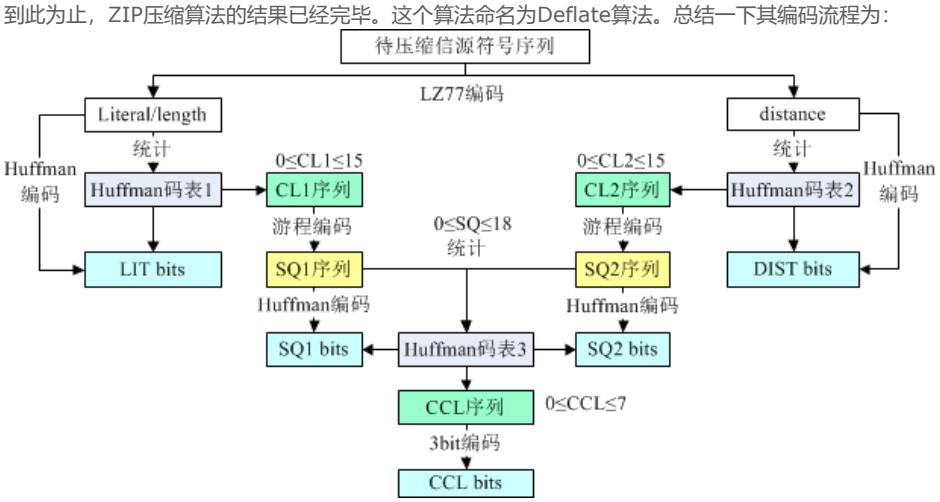
Position:	16	17	18	0	8	7	9	6	10	5	11	4	12	3	13	2	14	1	15
CCL:	2	4	0	4	0	0	0	5	0	0	0	5	0	1	0	5	0	5	0

不过粗看起来，这个置换效果并不好，我一开始接触这个置换的时候，我觉得应该按照16、17、18、0、1、2、3、。。。这样的顺序来存储，如果按照我理解的，那么置换后的结果如下：

2、4、0、4、5、5、1、5、0、6、0、0、0、0、0、0、0、0

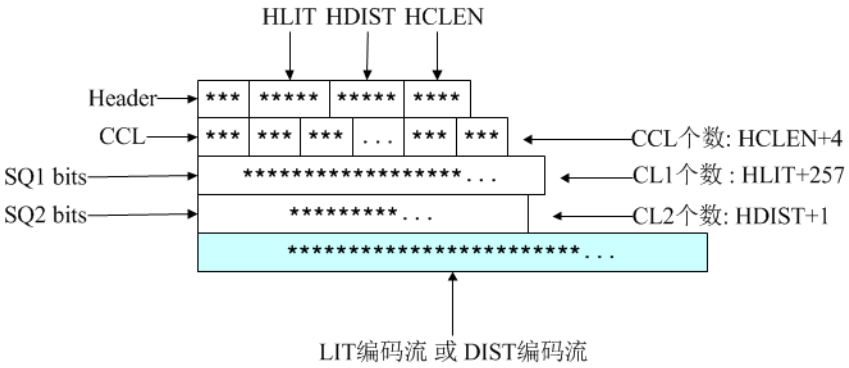
这样后面的一大串0直接截断，比PK的方法更短。但PK却按照上面的顺序。我总是认为，我觉得牛人可能出错了的时候，往往是我自己错了，所以我又仔细想了一下，上面的顺序特点比较明显，直观上看，PK认为CL为0和中间的值出现得比较多（放在了前面），但CL比较小的和比较大的出现得比较少（1、15、2、14这些放在了后面，你看，后面交叉着放），在文件比较小的时候，这种方法效果不算好，上面就是一个典型的例子，但文件比较大了以后，CL1、CL2码树比较大，码字长度普遍比较长，大部分很可能接近于中间值，那么这个时候PK的方法可能就体现出优势了。不得不说，对一个算法或者数据结构的优化程度，简直完全取决于程序员对那个东西细节的理解的深度。当我仔细研究了ZIP压缩算法的过程之后，我对PK这种深夜埋头冥思苦想的大牛佩服得五体投地。

到此为止，ZIP压缩算法的结果已经完毕。这个算法命名为Deflate算法。总结一下其编码流程为：



8、Deflate压缩数据格式

ZIP的格式实际上就是Deflate压缩码流外面套了一层文件相关的信息，这里先介绍Deflate压缩码流格式。其格式为：



Header：3个比特，第一个比特如果是1，表示此部分为最后一个压缩数据块；否则表示这是.ZIP文件的某个中间压缩数据块，但后面还有其他数据块。这是ZIP中使用分块压缩的标志之一；第2、3比特表示3个选择：压缩数据中没有使用Huffman、使用静态Huffman、使用动态Huffman，这是对LZ77编码后的literal/length/distance进行进一步编码的标志。我们前面分析的都是动态Huffman，其实Deflate也支持静态Huffman编码，静态Huffman编码原理更为简单，无需记录码表（因为PK自己定义了一个固定的码表），但压缩率不高，所以大多数情况下都是动态Huffman。

HLIT：5比特，记录literal/length码树中码长序列（CL1）个数的一个变量。后面CL1个数等于HLIT+257（因为至少有0-255总共256个literal，还有一个256表示解码结束，但length的个数不定）。

HDIST：5比特，记录distance码树中码长序列（CL2）个数的一个变量。后面CL2个数等于HDIST+1。哪怕没有1个重复字符串，distance都为0也是一个CL。

HCLEN：4比特，记录Huffman码表3中码长序列（CCL）个数的一个变量。后面CCL个数等于HCLEN+4。PK认为CCL个数不会低于4个，即使对于整个文件只有1个字符的情况。

接下来是3比特编码的CCL，一共HCLEN+4个，用以构造Huffman码表3；

接下来是对CL1（码长）序列经过游程编码（SQ1：缩短的整数序列）后，并对SQ1继续用Huffman编码后的比特流。包含HLIT+257个CL1，其解码码表为Huffman码表3，用以构造Huffman码表1；

接下来是对CL2（码长）序列经过游程编码（SQ2：缩短的整数序列）后，并对SQ2继续用Huffman编码后的比特流。包含HDIST+1个CL2，其解码码表为Huffman码表3，用于构造Huffman码表2；

总之，上面的数据都是为了构造LZ解码需要的2个Huffman码表。

接下来才是经过Huffman编码的压缩数据，解码码表为Huffman码表1和码表2。
最后是数据块结束标志，即literal/length这个码表输入符号位256的编码比特。
对倒数第1、2内容块进行解码时，首先利用Huffman码表1进行解码，如果解码所得整数位于0-255之间，表示literal未匹配字符，接下来仍然利用Huffman码表1解码；如果位于257-285之间，表示length匹配长度，之后需要利用Huffman码表2进行解码得到distance偏移距离；如果等于256，表示数据块解码结束。

9、ZIP文件格式解析

上面各节对ZIP的原理进行了分析，这一节我们来看一个实际的例子，为了更好地描述，我们尽量把这个例子举得简单一些。下面是我随便从一本书拷贝出来的一段较短的待压缩的英文文本数据：

As mentioned above,there are many kinds of wireless systems other than cellular.

这段英文文本长度为80字节。经过ZIP压缩后，其内容如下：

```
00000000h: 50 4B 03 04 14 00 00 00 08 00 8E 4D 25 45 3C 43 ; PK........峪%E<C
00000010h: AD 54 48 00 00 00 50 00 00 00 08 00 00 00 54 65 ; 瀍H...P.......Te
00000020h: 73 74 2E 74 78 74 15 CA D1 0D 80 20 0C 45 D1 55 ; st.txt.恃.€ .E裹
00000030h: DE 00 C6 1D 1C A5 CA 33 10 4B 49 68 D5 B0 BD F8 ; ??.ナ3.KIh瞻进
00000040h: 71 7F 4E EE E6 A8 B4 28 CD 98 20 7B 7B B8 44 66 ; q N钸ù(蜆 {{窪f
00000050h: 27 64 56 C5 06 AE 62 C9 D1 4E BC A5 53 E9 0E 1F ; 'dV?産裳N讥S?.
00000060h: 1E AC D3 FE 13 91 C5 70 50 F5 56 E9 EB 07 50 4B ; .  ?懪pP鲋嫔.PK
00000070h: 01 02 14 00 14 00 00 00 08 00 8E 4D 25 45 3C 43 ; ..........峪%E<C
00000080h: AD 54 48 00 00 00 50 00 00 00 08 00 00 00 00 00 ; 瀍H...P.........
00000090h: 00 00 01 00 20 00 80 81 00 00 00 00 54 65 73 74 ; .... .€?...Test
000000a0h: 2E 74 78 74 50 4B 05 06 00 00 00 00 01 00 01 00 ; .txtPK.........
000000b0h: 36 00 00 00 6E 00 00 00 00 00                   ; 6...n....
```

可以看到，第1、2字节就是PK。看着怎么比原文还长，这怎么叫压缩？实际上，这里面大部分内容是ZIP的文件标记开销，真正压缩的内容（也就是我们前面提到的Deflate数据，划线部分都是ZIP文件开销）其实肯定要比原文短（否则ZIP不会启用压

缩），我们这个例子是个短文本，但对于更长的文本而言，那ZIP文件总体长度肯定是要短于原始文本的。上面的这个ZIP文件，可以看到好几个以PK开头的区域，也就是不同颜色的划线区域，这些其实都是ZIP文件本身的开销。

所以，我们首先来看一看ZIP的格式，其格式定义为：

[local file header 1]
[file data 1]
[data descriptor 1]
.....
[local file header n]
[file data n]
[data descriptor n]
[archive decryption header]
[archive extra data record]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
local file header+file data+data descriptor这是一段ZIP压缩数据，在一个ZIP文件里，至少有一段，至多那就不好说了，假如你要压缩的文件一共有10个，那这个地方至少会有10段，ZIP对每个文件进行了独立压缩，RAR在此进行了改进，将多个文件联合起来进行压缩，提高了压缩率。local file header的格式如下：

Local file header			
偏移量	占用字节	含义	译
0	4	Signature = 0x04034b50 (little-endian)	文件头标识，值固定(0x04034b50)
4	2	Version needed to extract (minimum)	解压文件所需最低版本
6	2	General purpose bit flag	通用位标记
8	2	Compression method	压缩方法
10	2	File last modification time	文件最后修改时间
12	2	File last modification date	文件最后修改日期
14	4	CRC-32	说明采用的算法。
18	4	Compressed size	压缩后的大小。
22	4	Uncompressed size	非压缩的大小。
26	2	File name length (*n*)	文件名长度
28	2	Extra field length (*m*)	扩展区长度
30	n	File name	文件名
30+n	m	Extra field	扩展区

可见，起始的4个字节就是0x50（P）、0x4B（K）、0x03、0x04，因为是低字节优先，所以Signature=0x03044B50.接下来的内容按照上面的格式解析，十分简单，这个区域在上面ZIP数据的那个图里面是红色划线区域，之后则是压缩后的Deflate数据。在文件的尾部，还有ZIP尾部数据，上面这个例子包含了central directory和end of central directory record，一般这两部分也是必须的。central directory以0x50、0x4B、0x01、0x02开头；end of central directory record以0x50、0x4B、0x05、0x06开头，其含义比较简单，分别对应于上面ZIP数据那个图的蓝色和绿色部分，下面是两者的格式：

Central directory file header			
偏移量	占用字节	Description	译
0	4	signature = 0x02014b50	核心目录文件 header 标识=（0x02014b50）
4	2	Version made by	压缩所用的 pkware 版本
6	2	Version needed to extract (minimum)	解压所需 pkware 的最低版本
8	2	General purpose bit flag	通用位标记
10	2	Compression method	压缩方法
12	2	File last modification time	文件最后修改时间
14	2	File last modification date	文件最后修改日期
16	4	CRC-32	CRC-32 算法
20	4	Compressed size	压缩后大小
24	4	Uncompressed size	未压缩的大小
28	2	File name length (*n*)	文件名长度
30	2	Extra field length (*m*)	扩展域长度
32	2	File comment length (*k*)	文件注释长度
34	2	Disk number where file starts	文件开始位置的磁盘编号
36	2	Internal file attributes	内部文件属性
38	4	External file attributes	外部文件属性
42	4	Relative offset of local file header.	本地文件 header 的相对位移。
46	*n*	File name	目录文件名
46+*n*	*m*	Extra field	扩展域
46+*n*+*m*	*k*	File comment	文件注释内容

end of central directory record格式:

End of central directory record			
偏移量	占用字节	Description	译
0	4	Signature = 0x06054b50	核心目录结束标记（0x06054b50）
4	2	Number of this disk	当前磁盘编号
6	2	Disk where central directory starts	核心目录开始位置的磁盘编号
8	2	Number of central directory records on this disk	该磁盘上所记录的核心目录数量
10	2	Total number of central directory records	核心目录结构总数
12	4	Size of central directory (bytes)	核心目录的大小
16	4	Offset of start of central directory, relative to start of archive	核心目录开始位置相对于 archive 开始的位移
20	2	Comment length (*n*)	注释长度
22	*n*	Comment	注释内容

这几张图是我从网上找的，写得比较清晰。对于其中的含义，解释起来也比较简单，我分析的结果如下：注意ZIP采用的低字节优先，在一个字节里面低位优先，需要反过来看。

Local File Header: (38B,304b)
00001010110100101100000000100000 (signature)
0000000000010100 (version:20)
0000000000000000 (generalBitFlag)
0000000000001000 (compressionMethod:8)
0100110110001110 (lastModTime:19854)
0100010100100101 (lastModDate:17701)
01010100101011010100001100111100 (CRC32)
00000000000000000000000001001000 (compressedSize:72)
00000000000000000000000001010000 (uncompressedSize:80)
0000000000001000 (filenameLength:8)
0000000000000000 (extraFieldLength:0)
00101010101001101100111000101110011101000010111000011111000101110 (fileName:Test.txt)
(extraField)

Central File Header: (54B,432b)
00001010110100101000000001000000 (signature)
0000000000010100 (versionMadeBy:20)
0000000000010100 (versionNeeded:20)
0000000000000000 (generalBitFlag)

0000000000001000 (compressionMethod:8)
0100110110001110 (lastModTime:19854)
0100010100100101 (lastModDate:17701)
01010100101011010100001100111100 (CRC32)
00000000000000000000000001001000 (compressedSize:72)
00000000000000000000000001010000 (uncompressedSize:80)
0000000000001000 (filenameLength:8)
0000000000000000 (extraFieldLength:0)
0000000000000000 (fileCommenLength:0)
0000000000000000 (diskNumberStart)
0000000000000001 (internalFileAttr)
10000001100000000000000000100000 (externalFileAttr)
00000000000000000000000000000000 (relativeOffsetLocalHeader)
00101010101001101100111000101110011101000010111000011111000101110 (fileName:Test.txt)
(extraField)
(fileComment)

end of Central Directory Record: (22B,176b)
00001010111010010101000001100000 (signature)
0000000000000000 (numberOfThisDisk:0)
0000000000000000 (numberDiskCentralDirectory:0)
0000000000000001 (EntriesCentralDirectDisk:1)
0000000000000001 (EntriesCentralDirect:1)
00000000000000000000000000110110 (sizeCentralDirectory:54)
00000000000000000000000001101110 (offsetStartCentralDirectory:110)
0000000000000000 (fileCommentLength:0)
(fileComment)

Local File Header Length:304
Central File Header Length:432
End Central Directory Record Length:176

可见，开销总的长度为38+54+22=114字节，整个文件长度为186字节，因此Deflate压缩数据长度为72字节（576比特）。尽管这里看起来只是从80字节压缩到72字节，那是因为这是一段短文本，重复字符串出现较少，但如果文本较长，那压缩率就会增加，这里只是举个例子。

下面对其中的关键部分，也就是Deflate压缩数据进行解析。

10，Deflate解码过程实例分析

我们按照ZIP格式把Deflate压缩数据（72字节）提取出来，如下（每行8字节）：

10101000010100111000101110110000000000010000010000110000010100010
10001011101010100111101100000000011000111011100000111000101001010
01010011110011000000100011010010100100100001011010101011000011010
10111101000111111000111011111111001110010011101110110011100010101
00101101000101001011001100011001000001001101111011011111000011101
00100010011001101110010000100110011010101010001101100000001110101
01000110100100111000101101110010001111011010010111001010100101111
01110000111110000111100000110101110010110111111111100100010001001
10100011000011100000101010101111011010101001011111010111111000000

Deflate格式除了上面的介绍，也可以参考RFC1951，解析如下：

Header:101，第一个比特是1，表示此部分为最后一个压缩数据块；后面的两个比特01表示采用动态哈夫曼、静态哈夫曼、或者没有编码的标志，01表示采用动态Huffman；在RFC1951里面是这么说明的：

00 - no compression

01 - compressed with fixed Huffman codes

10 - compressed with dynamic Huffman codes

11 - reserved (error)

注意，这里需要按照低比特在先的方式去看，否则会误以为是静态Huffman。

接下来：
HLIT:01000,记录literal/length码树中码长序列个数的一个变量，表示HLIT=2（低位在前），说明后面存在HLIT + 257=259个CL1，CL1即0-258被编码后的长度，其中0-255表示Literal，256表示无效符号，257、258分别表示Length=3、4（length从3开始）。因此，这里实际上只出现了两种重复字符串的长度，即3和4。回顾这个图可以更清楚：

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			

继续：
HDIST:01010,记录distance码树中码长序列个数的一个变量，表示HDIST=10，说明后面存在HDIST+1=11个CL2，CL2即Distance Code=0-10被编码的长度。

继续：

HCLEN:0111,记录Huffman码树3中码长序列个数的一个变量，表示HCLEN=14（1110二进制），即说明紧接着跟着HCLEN+4=18个CCL，前面已经分析过，CCL记录了一个Huffman码表，这个码表可以用一个码长序列表示，根据这个码长序列可以得到码表。于是接下来我们把后面的18*3=54个比特拷贝出来，上面的码流目前解析为下面的结果：

101(Header) 01000(HLIT) 01010(HDIST) 0111(HCLEN)
000 101 110 110 000 000 000 010 000 010 000 110 000 101 000 101 000 101 (CCL)
110101010011110110000000001100011101110000011100010100101
010100111100110000001000110100101001001000010110101010110000 1101
101111010001111110001110111111100111001001110111011001110001 0101
001011010001010010110011000110010000010011011110110111000011101
001000100110011011100100001001100110101010100011011000000 1110101
010001101001001110001011011100100011110110100101110010101001 0111
011100001111100001111000001101011100101101111111110010001000 1001
101000110000111000001010101011110110101010010111110101111110 0000

标准的CCL长度为19（回忆一下：CCL范围为0-18，按照整数大小排序记录各自的码字长度），因此最后一个补0。得到序列：

000 101 110 110 000 000 000 010 000 010 000 110 000 101 000 101 000 101 000

其长度分别为（低位在前）：
0、5、3、3、0、0、0、2、0、2、0、3、0、5、0、5、0、5、0
前面已经分析过，这个CCL序列实际上是经过一次置换操作得到的，需要进行相反的置换，置换后为：

3、5、5、5、3、2、2、0、0、0、0、0、0、0、0、0、0、5、3
这个就是对应于0-18的码字长度序列。
根据Deflate树的构造方式，得到下面的码表（Huffman码表3）：

```
00    <-->  5
01    <-->  6
100   <-->  0
101   <-->  4
110   <-->  18
11100  <-->1
11101  <-->2
11110  <-->3
11111  <-->17
```

接下来就是CL1序列，按照前面的指示，一共有259个，分别对应于literal/length：0-258对应的码字长度序列，我们队跟着CCL后面的比特按照上面获得的码表进行逐步解码，在解码之前，实际上并不知道CL1的比特流长度有多少，需要根据259这个数字来判定，解完了259个整数，表明解析CL1完毕：

101(Header) 01000(HLIT) 01010(HDIST) 0111(HCLEN)
000 101 110 110 000 000 000 010 000 010 000 110 000 101 000 101 000 101 (CCL)

110（18）1010100（7比特，记录连续的11-138个0，此处一共0010101b=21，即记录21+11=32个0）

11110（3）110（18）0000000（7比特，记录连续的11-138个0，此处为全0，即记录0+11=11个0）

01（6）100（0）01（6）110（18）1110000（7比特，记录连续的11-138个0，此处为111b=7，即记录7+11=18个0）

01（6）110（18）0010100（7比特，记录连续的11-138个0，此处为10100b=20，即记录20+11=31个0）

101（4）01（6）01（6）00（5）11110（3）01（6）100（0）00（5）00（5）100（0）01（6）101（4）

00（5）101（4）00（5）100（0）100（0）00（5）101（4）101（4）01（6）01（6）01（6）100（0）

00（5）110（18）1101111（7比特，记录连续的11-138个0，此处为1111011b=123，即记录123+11=134个0）

统计一下，上面已经解了32+11+18+31+134+30=256个数了，因为总共259个，还差三个：

01（6）00（5）01（6）

好了，CL1比特流解析完毕了，得到的CL1码长序列为：

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **3** 0 0 0 0 0 0 0
0 0 0 0 6 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 6 6 5 3 6 0 5 5 0 6 4 5 4 5 0 0 5 4 4 6 6 6
0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **6 5 6**

总共259个，每行40个。根据这个序列，同样按照Deflate树构造方法，得到literal/length码表（Huffman码表1）为：

000 --> (System.Char)（看前面的CL1序列，空格对应的ASCII为0x20=32，码字长度3，即上面序列中第一个3）
001 -->e(System.Char)
0100 -->a(System.Char)
0101 -->l(System.Char)
0110 -->n(System.Char)
0111 -->s(System.Char)
1000 -->t(System.Char)
10010 -->d(System.Char)
10011 -->h(System.Char)
10100 -->i(System.Char)
10101 -->m(System.Char)
10110 -->o(System.Char)
10111 -->r(System.Char)
11000 -->y(System.Char)
11001 -->3(System.Int32)（看前面的CL1序列，对应257，码字长度5）
110100 -->,(System.Char)
110101 -->.(System.Char)
110110 -->A(System.Char)
110111 -->b(System.Char)
111000 -->c(System.Char)
111001 -->f(System.Char)
111010 -->k(System.Char)
111011 -->u(System.Char)
111100 -->v(System.Char)
111101 -->w(System.Char)
111110 -->-1(System.Int32)（看前面的CL1序列，对应256，码字长度6）
111111 -->4(System.Int32)（看前面的CL1序列，对应258，码字长度6）

可以看出，码表里存在两个重复字符串长度3和4，当解码结果为-1（上面进行了处理，即256），或者说遇到111110的时候，表示Deflate码流结束。

按照同样的道理，对CL2序列进行解析，前面已经知道HDIST=10，即有11个CL2整数序列：

11111（17）000（3比特，记录连续的3-10个0，此处为0，即记录3个0）

11101（2）11111（17）100（3比特，记录连续的3-10个0，此处为001b=1，即记录4个0）

11100（1）100（0）11101（2）

已经结束，总共11个。

于是CL2序列为：

0 0 0 2 0 0 0 0 1 0 2

分别记录的是distance码为0-10的码字长度，根据下面的对应关系，需要进行扩展：

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

比如，第1个码长2记录的是Code=3的长度，即Distance=4对应的码字为：

10 -->4(System.Int32)

第1个码长1记录的是Code=8的长度（码字为0，扩展三位000-111），即Distance=17-24对应的码字为（注意，低比特优先）：

0 000 -->17(System.Int32)
0 100 -->18(System.Int32)
0 010 -->19(System.Int32)
0 110 -->20(System.Int32)
0 001 -->21(System.Int32)
0 101 -->22(System.Int32)
0 011 -->23(System.Int32)
0 111 -->24(System.Int32)

注意，扩展的时候还是低比特优先。

最后1个码长2记录的是Code=10的长度（其实是码字：11，扩展四位0000-1111），即Distance=33-48对应的码字为：

11 0000 -->33(System.Int32)
11 1000 -->34(System.Int32)
11 0100 -->35(System.Int32)
11 1100 -->36(System.Int32)
11 0010 -->37(System.Int32)
11 1010 -->38(System.Int32)
11 0110 -->39(System.Int32)
11 1110 -->40(System.Int32)
11 0001 -->41(System.Int32)
11 1001 -->42(System.Int32)
11 0101 -->43(System.Int32)
11 1101 -->44(System.Int32)
11 0011 -->45(System.Int32)
11 1011 -->46(System.Int32)
11 0111 -->47(System.Int32)
11 1111 -->48(System.Int32)

至此为止，Huffman码表1、Huffman码表2已经还原出来，接下来是对LZ压缩所得到的literal、distance、length进行解码，目前剩余的比特流如下，先按照Huffman码表1解码，如果解码结果是长度（>256），则接下来按照Huffman码表2解码，逐步解码即可：

[As]：110110（A）0111（s）000（空格）

[mentioned]：10101（m）001（e）0110（n）1000（t）10100（i）10110（o）0110（n）001（e）10010（d）000（空格）

[above,]：0100（a）110111（b）10110（o）111100（v）001（e）110100（,）

[there]：1000（t）10011（h）001（e）10111（r）001（e）000（空格）

[are]：0100（a）11001（长度3，表示下一个需要用Huffman解码）10（Distance=4，即重复字符串为re空格）

[many]：10101（m）0100（a）0110（n）11000（y）000（空格）

[kinds]：111010（k）10100（i）0110（n）10010（d）0111（s）000（空格）

[of]：10110（o）111001（f）000（空格）

[wireless]：111101（w）10100（i）10111（r）001（e）0101（l）001（e）0111（s）0111（s）000（空格）

[systems o]：0111（s）11000（y）0111（s）1000（t）001（e）10101（m）11001（长度指示=3，接下来根据distance解码）0110（Distance=20,即重复字符串为s o）

[ther]：111111（长度指示=4，接下来根据distance解码）111001（Distance=42,即重复字符串为ther）000（空格）

[than]：1000（t）10011（h）0100（a）0110（n）000（空格）

[cellular.]：111000（c）001（e）0101（l）0101（l）111011（u）0101（l）0100（a）10111（r）110101（.）

[256，结束标志]111110（结束标志）0000（字节补齐的0）

于是解压缩结果为：

As mentioned above,there are many kinds of wireless systems other than cellular.

再来回顾我们的解码过程：

译码过程：
1、根据HCLEN得到截尾信息，并参照固定置换表，根据CCL比特流得到CCL整数序列；
2、根据CCL整数序列构造出等价于CCL的二级Huffman码表3；
3、根据二级Huffman码表3对CL1、CL2比特流进行解码，得到SQ1整数序列,SQ2整数序列；
4、根据SQ1整数序列,SQ2整数序列,利用游程编码规则得到等价的CL1整数序列、CL2整数序列；
5、根据CL1整数序列、CL2整数序列分别构造两个一级Huffman码表：literal/length码表、distance码表；
6、根据两个一级Huffman码表对后面的LZ压缩数据进行解码得到literal/length/distance流；
7、根据literal/length/distance流按照LZ规则进行解码。

Deflate码流长度总共为72字节=576比特，其中：

3比特Header；

5比特HLIT；

5比特HDIST；

4比特HCLEN；

54比特CCL序列码流；

133比特CL1序列码流；

34比特CL2序列码流；

338比特LZ压缩后的literal/length/distance码流。

11、ZIP的其它说明

上面各个环节已经详细分析了ZIP压缩的过程以及解码流程，通过对一个实例的解压缩过程分析，可以彻底地掌握ZIP压缩和解压缩的原理和过程。还有一些情况需要说明：

（1）上面的算法复杂度主要在于压缩一端，因为需要统计literal/length/distance，创建动态Huffman码表，相反解压只需要还原码表后，逐比特解析即可，这也是压缩软件的一个典型特点，解压速度远快于压缩速度。

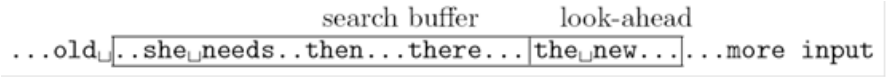
（2）上面我们分析了动态Huffman，对于LZ压缩后的literal/length/distance，也可以采用静态Huffman编码，这主要取决于ZIP在压缩中看哪种方式更节省空间，静态Huffman编码不需要记录码表，因为这个码表是固定的，在RFC1951里面也有说明。对于literal/length码表来说，需要对0-285进行编码，其码表为：

edoc	Bits	Prefix codes
0–143	8	00110000–10111111
144–255	9	110010000–111111111
256–279	7	0000000–0010111
280–287	8	11000000–11000111

对于Distance来说，需要对Code=0-29的数进行编码，则直接采用5比特表示。Distance和动态Huffman一样，在此基础上进行扩展。

（3）ZIP中使用的LZ77算法是一种改进的LZ77。主要区别有两点：

1）标准LZ77在找到重复字符串时输出三元组(length, distance, 下一个未匹配的字符)（有兴趣可以关注LZ77那篇论文）；Deflate在找到重复字符串时仅输出双元组(length, distance)。
2）标准LZ77使用"贪婪"的方式解析，寻找的都是最长匹配字符串。Deflate中不完全如此。David Salomon的书里给了一个例子：



对于上面这个例子，标准LZ77在滑动窗口中查找最长匹配字符串，找到的是"the"与前面的there的前三个字符匹配，这种贪婪解析方式逻辑简单，但编码效率不一定最高。Deflate则不急于输出，跳过t继续往后查看，发现"th ne"这5个字符存在重复字符串，因此，Deflate算法会选择将t作为未匹配字符输出，而对后面的匹配字符串用(length, distance)编码输出。显然，这样就提高了压缩效率，因为标准的LZ77找到的重复字符串长度为3，而Deflate找到的是5。换句话说，Deflate算法并不是简单的寻找最长匹配后输出，而是会权衡几种可行的编码方式，用其中最高效的方式输出。

12、总结

本篇博文对ZIP中使用的压缩算法进行了详细分析，从一个简单地例子出发，一步步地分析了PK设计Deflate算法的思路。最后，通过一个实际例子，分析了其解压缩流程。总的来看，ZIP的核心在于如何对LZ压缩后的literal、length、distance进行Huffman编码，以及如何以最小空间记录Huffman码表。整个过程充满了对数据结构尤其是树的深入优化利用。按照上面的分析，如果要对ZIP进行进一步改进，可以考虑的地方也有不少，典型的有：

（1）扩大LZ编码的滑动窗口的大小；

（2）将Huffman编码改进为算术编码等压缩率更高的方法，毕竟，Huffman的码字长度必须为整数，这就从理论上限制了它的压缩率只能接近于理论极限，但难以达到。我记得在JPEG图像编码领域，以前的JPEG采用了DCT变换编码+Huffman的方式，现在JPEG2000将其改为小波变换+算数编码，所以数据压缩也可以尝试类似的思路；

（3）将多个文件进行合并压缩，ZIP中，不同的文件压缩过程没有关系，独立进行，如果将它们合并起来一起进行压缩，压缩率可以得到进一步提高。

描述分析有误的地方，敬请指正。针对数据压缩相关的话题，后续会对HBase列压缩等等进行分析，看看ZIP这种文件压缩和HBase这种数据库数据压缩的区别和联系。

分类: C/++

标签: 数据压缩, zip

随O心
关注 - 0
粉丝 - 0
+加关注

发表于 2018-10-09 10:45 随O心 阅读(1953) 评论(1) 编辑 收藏

评论

#1楼

生，容易。活，容易。生活，不容易。

上面这句话假如不压缩，如果使用Unicode编码，每个字会用2个字节表示。

这句话不严谨，Unicode只是一个字符集，不是编码方式；
事实上根本不会有人用Unicode直接存储内容，因为nicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。
比如UTF-8和UTF-16都是Unicode的实现方式；
另外Unicode表示汉字也不是每个字都会是2个字节，因为汉字有十万多，根本不够呢

支持(0) 反对(0)

无心流泪 评论于 2019-07-01 20:16

相关博文：
· ZIP压缩算法详细分析及解压实例解释
· 解压Zip
· Linux下的压缩zip,解压缩unzip命令详解及实例
· ZIP压缩算法原理分析及解压实例代码
· Linux下的压缩zip,解压缩unzip命令具体解释及实例
» 更多推荐...

最新 IT 新闻：
· 以魔鬼鱼为灵感 科学家研制新型航天器探索金星黑暗面
· 3个搞物理的颠覆了数学常识，数学天才陶哲轩：我开始压根不相信
· 5G、云计算、物联网与边缘计算的相辅相承
· 新研究有助揭示日冕高温之谜
· 对话联想童夫尧："超算+AI+5G"将成未来超算研究新方向
» 更多新闻...