

# 基于GZIP思想的文件压缩

---

## [本节目标]

- 1. 数据压缩的概念
- 2. 为什么需要压缩
- 3. 压缩的分类
- 4. ZIP压缩的历史
- 5. GZIP压缩算法的原理
- 6. 常见压缩算法的对比
- 7. 面试相关

## 1. 数据压缩的概念

数据压缩是指在不丢失有用信息的前提下，缩减数据量以减少存储空间，提高其传输、存储和处理效率，或按照一定的算法对数据进行重新组织，减少数据的冗余和存储的空间的一种技术方法。

## 2. 为什么需要压缩

1. 紧缩数据存储容量，减少存储空间
2. 可以提高数据传输的速度，减少带宽占用量，提高通讯效率
3. 对数据的一种加密保护，增强数据在传输过程中的安全性

## 3. 压缩的分类

- 有损压缩

有损压缩是利用了人类对图像或声波中的某些频率成分不敏感的特性，允许压缩过程中损失一定的信息；虽然不能完全恢复原始数据，但是所损失的部分对理解原始图像的影响缩小，却换来了大得多的压缩比，即指使用压缩后的数据进行重构，重构后的数据与原来的数据有所不同，但不影响人对原始资料表达的信息造成误解。

- 无损压缩

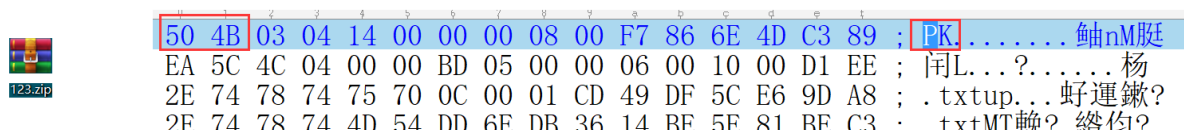
对文件中数据按照特定的编码格式进行重新组织，压缩后的压缩文件可以被还原成与源文件完全相同的格式，不会影响文件内容，对于数码图像而言，不会使图像细节有任何损失。



## 4. ZIP压缩的历史

1977年，两位以色列人Jacob Ziv和Abraham Lempel，发表了一篇论文《A Universal Algorithm for Sequential Data Compression》，一种通用的数据压缩算法，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，该算法奠基了今天大多数无损数据压缩的核心，为了纪念两位科学家，该算法被称为LZ77，过了一年他们又提了一个类似的算法，称为LZ78。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。在LZ77、LZ78基础上变种的算法很多，基本都以LZ开头，如LZW、LZO、LZMA、LZSS、LZR、LZB、LZH、LZC、LZT、LZMW、LZJ、LZFG等等。

ZIP的作者是Phil Katz，他算是开源界的一个具有悲剧色彩的传奇人物。Phil Katz是个牛逼程序员，成名于DOS时代，那个时代网速很慢，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，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，但是要付费，Phil Katz就感觉到不爽，于是写了一个PKARC压缩工具，不仅免费，而且兼容ARC，于是网友都用PKARC了，ARC那个公司自然就不爽，把Phil Katz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。



```

00000000  50 4B 03 04 14 00 00 00 08 00 F7 86 6E 4D C3 89 ; PK..... 魷nM脰
00000010  EA 5C 4C 04 00 00 BD 05 00 00 06 00 10 00 D1 EE ; 閏L...?...... 杨
00000020  2E 74 78 74 75 70 0C 00 01 CD 49 DF 5C E6 9D A8 ; .txtup... 野運鏃?
00000030  2F 74 78 74 4D 54 DD 6F DB 36 14 BF 5F 81 BF C3 ; +x+MT 轡? 縹恁?
  
```

## 5. GZIP压缩算法的原理

GZIP压缩算法经历了两个阶段，第一个阶段使用改进的LZ77压缩算法对上下文中的重复语句进行压缩，第二阶段，采用huffman编码思想对第一阶段压缩完成的数据进行字节上的压缩，从而实现对数据的高效压缩存储。

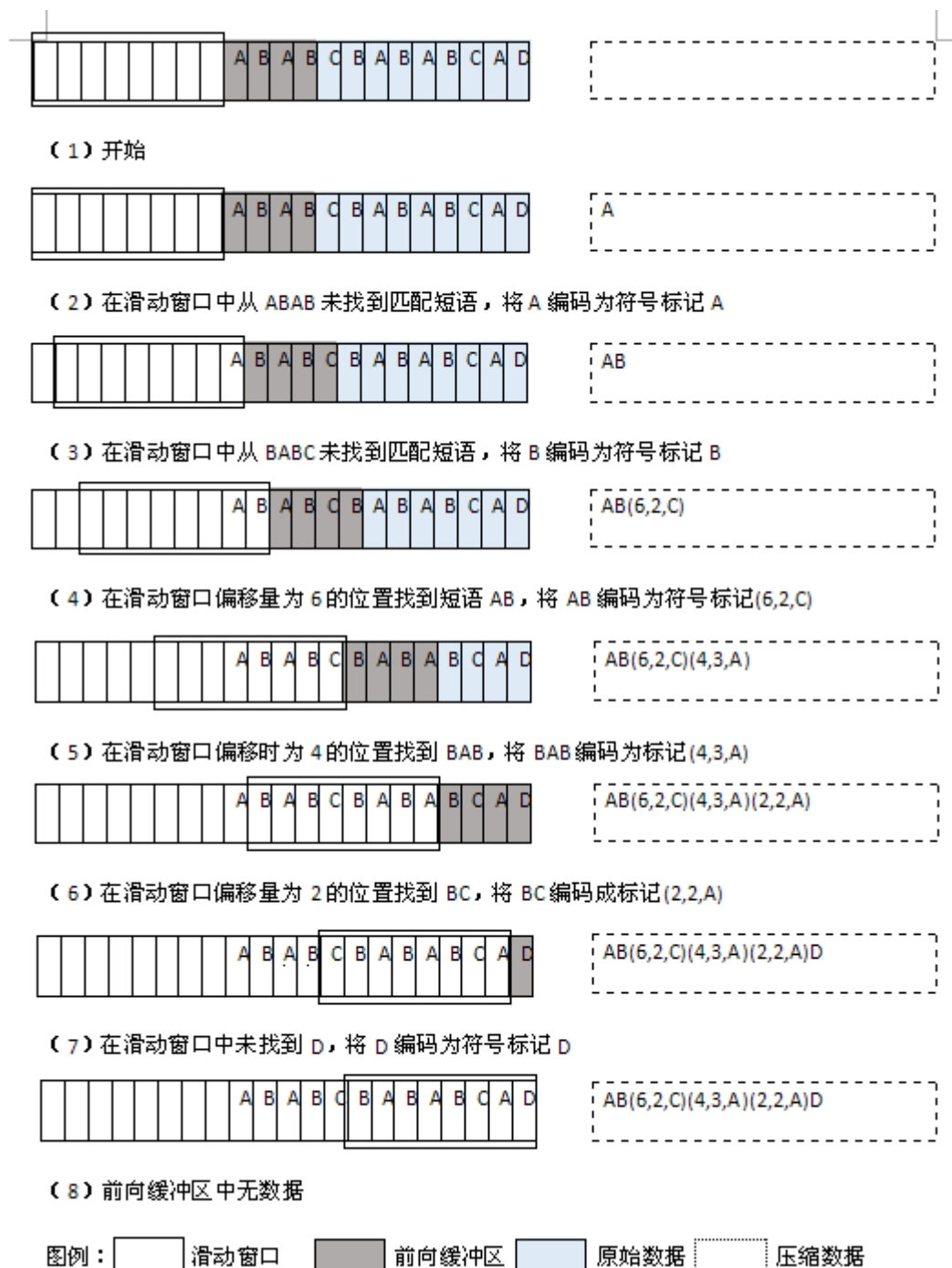
### 5.1 LZ77压缩算法

LZ77是一种基于字典的算法，它将长字符串（也称为短语）编码成短小的标记，用小标记代替字典中的短语，从而达到压缩的目的。

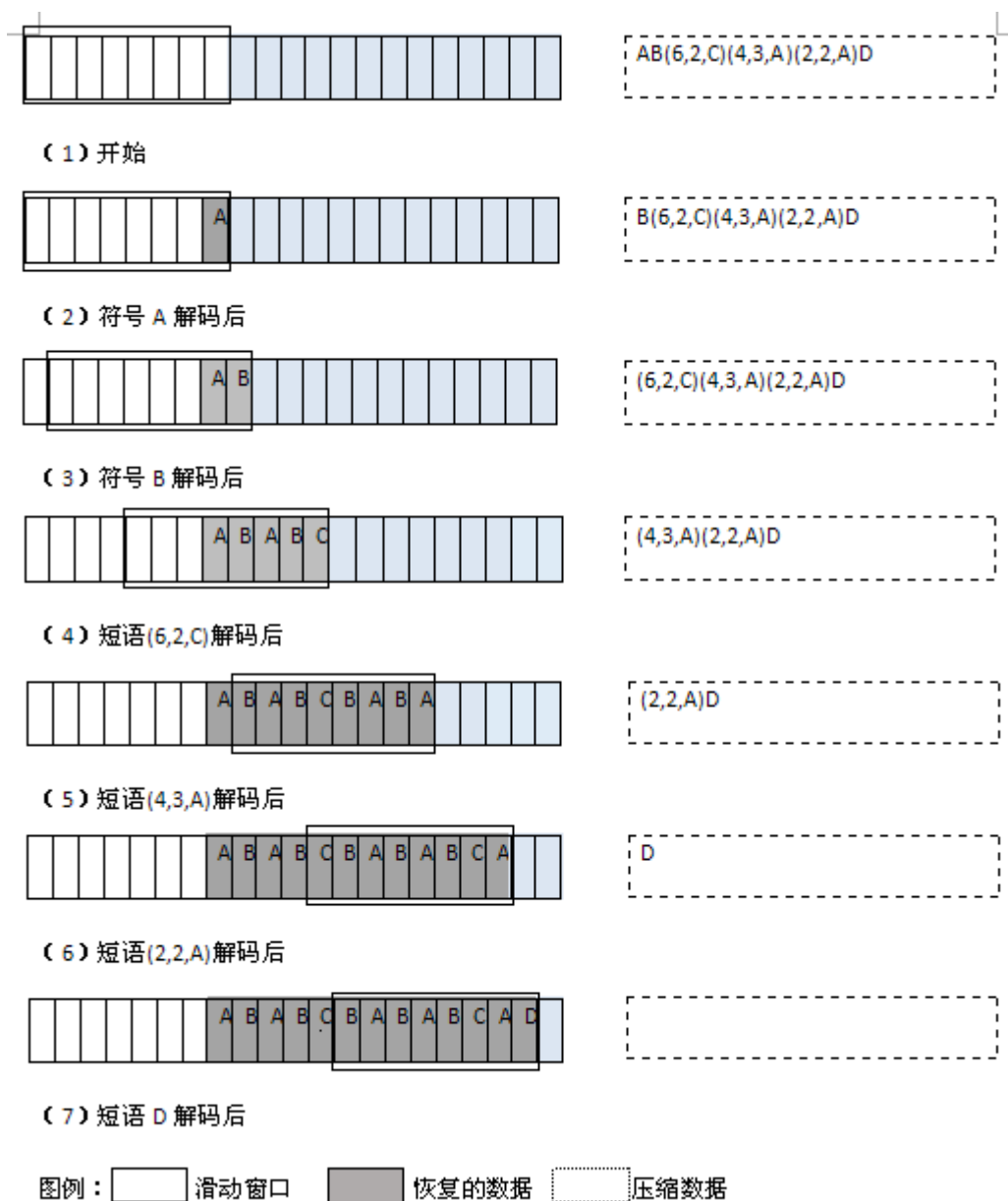
**LZ77使用的是一个前向缓冲区和一个滑动窗口来维护字典。**它首先将一部分数据载入前向缓冲区,一旦数据中的短语通过前向缓冲区,那么它将移动到滑动窗口中,并变成字典的一部分,随着滑动窗口的移动,字典中的内容也在不断的更新,也就是说它是变更新字典,边进行压缩。

假设字典已经建立,每读到一个字符,就向前扫描,在字典中查看是否有重复出现的短语,若搜索出有重复出现的短语,将其编码成短语标记。短语标记由三部分组成:**滑动窗口中的偏移量(从头部到匹配开始的前一个字符)、匹配中的符号个数、匹配结束后,前向缓冲区中的第一个符号---(offset, lenght, nextChar)。**当没有找到匹配时,将未匹配的符号编码成符号标记。这个符号标记仅仅包含符号本身,没有压缩过程。一旦把n个符号编码并生成相应的标记,就将这n个符号从滑动窗口的一端移出,并用前向缓冲区中同样数量的符号来代替它们。然后,重新填充前向缓冲区。

下图展示了LZ77的压缩过程,假设滑动窗口大小为8个字节,前向缓冲区大小为4个字节:

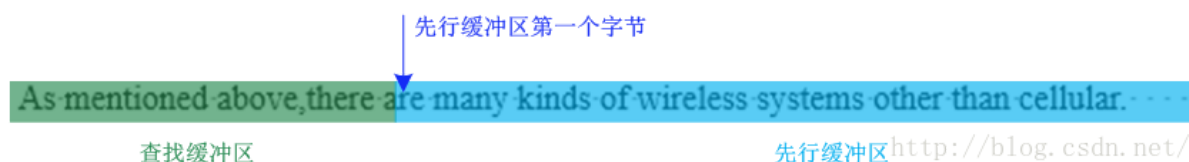


当解码每个标记时，将标记编码成字符拷贝到滑动窗口中。每当遇到一个短语标记时，就在滑动窗口中查找相应的偏移量，同时查找在那里发现的指定长度的短语。每当遇到一个符号标记时，就生成标记中保存的一个符号：



## 5.2 GZIP 中的LZ77思想

在GZIP算法中，也使用到LZ77的算法思想，但是对其进行了改进，主要是**对于短语标记的改进：只使用“长度+距离”的二元组进行表示，匹配的查找是在查找缓冲区中进行的，即字典。**



注意：查找缓冲区的数据是已经被扫描过，建立的字典中的数据，先行缓冲区即为带压缩数据

当前先行缓冲区第一个字节就是字符“r”，要做的就是找到以“r”为起始字符，在先行缓冲区中由连续字符组成的字符串在查找缓冲区中的最长匹配。例如，“re”在绿色部分中有，“re（这里有个空格）”在绿色部分中也有，但是后者比前者多一个字符，所以就选后者。

现在有四个问题需要处理：

1. 从“r”开始，先行缓冲区中的字符串由几个字符构成？有什么规定吗？还是说只要能找到匹配，不管几个字符都行？
2. 如何高效的在查找缓冲区中找到匹配串？
3. 如何找到最长匹配？
4. 找不到匹配怎么办？

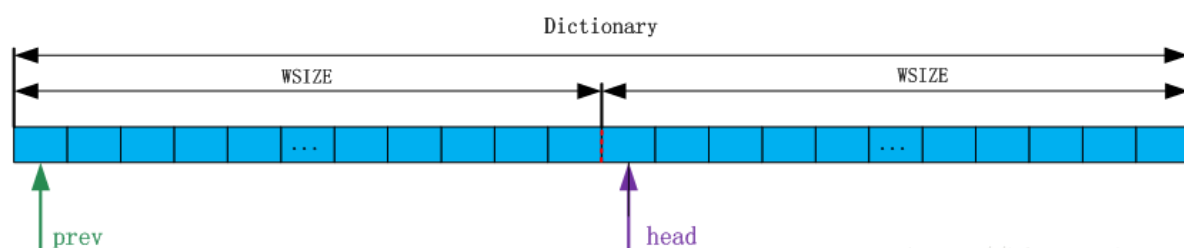
### 5.2.1 匹配串中由几个字符构成？

1. 如果对单个字符用长度距离对儿替换，那还不如不替换
2. 如果对两个连续字符用长度距离对儿替换，那是否替换又有什么关系，反正替换前后都要用两个数（字符其实也是个数）
3. 能够替换的前提就是字符串至少要由3个位于先行缓冲区中的连续字符组成，而且还必须以先行缓冲区中第一个字节为开始

因此：先行缓冲区中的字符串最少要由3个连续的字符构成。也就是说，当先行缓冲区中的字符串至少由3个连续的字符组成并且在查找缓冲区中找到了匹配，此时先行缓冲区中的这个字符串才能用“长度+距离”对儿替换。

### 5.2.2 如何高效的查找匹配串

一个字符串必须至少由三个连续字符组成，此时该字符串才有资格在查找缓冲区中寻找匹配串。**压缩使用字典(哈希表)来提高查找速度。**在压缩中，“字典”由一整块连续的内存构成，该内存大小为64KB，分成两部分，每部分大小为一个WSIZE，如下图所示



<http://blog.csdn.net/>

指针 $head = prev + WSIZE$ ， $prev$ 指向该字典整个内存的起始位置。既然内存是连续的，所以 $prev$ 和 $head$ 可以看作两个数组，即 $prev[]$ 和 $head[]$ 。

压缩就是利用这三个连续字符来计算哈希值，这个哈希值就是 $head[]$ 数组的索引。三个字符不同，排列的次序不同，计算出的哈希值就不同，总共有 $2^{24}$ 种结果，而哈希值的范围是： $[0, 32767]$  ( $head$ 数组的大小为32K)，因此不能做到每个哈希值对应一种字符串，肯定会存在冲突，而 $prev$ 就是来解决冲突的，而且该数组还可以帮助找到最长匹配串。

#### 1. 字典构建

所谓构建字典，就是将字符串插入到字典中，压缩是逐字节进行的，每处理一个字节，查找缓冲区扩大一个字节(如果已经达到32KB，那就沿着先行缓冲区的方向滑动一个字节)，先行缓冲区缩小一个字节，先行缓冲区的第一个字符在不断的变化。**每次先行缓冲区的第一个字节改变的时候，就要用“由当前的先行缓冲区第一个字节开始，在先行缓冲区中连续的三个字节”组成的字符串来计算哈希值 $ins\_h$ ，该哈希值是数组 $head[]$ 的索引，所以用 $head[ins\_h]$ 来记录该字符串的出现位置，该字符串的出现位置就是这个字符串第一个字符（也就是当前先行缓冲区第一个字节）在当前window中的位置，这就是把字符串插入字典的过程。**

有“abcdefg”，假设先行缓冲区中第一个字符为c

“cde” ---> 哈希值: ins\_h1, “cde”首字符c在“abcdefg”中起始位置为2, 其插入伪代码为: head[ins\_h1] = 2

“def” ---> 哈希值: ins\_h2, “def”首字符d在“abcdefg”中起始位置为3, 其插入伪代码为: head[ins\_h1] = 2

## 2. 查找匹配串

“mnoabc~~zxy~~uvabc~~zxy~~defgh”

假设第一个“abc”已经插入到字典中，哈希值为ins\_h1, head[ins\_h1] = 3, 先行缓冲区第一个字符为第二个“abc”中的‘a’, 计算“abc”的哈希值为ins\_h2, ins\_h2与ins\_h1必然相同，即在head[ins\_h2]不为空，找到匹配串“abc”，此时匹配不会停止，“abc”和“abc”分别向后继续匹配，直到遇到不能匹配的字符为止，因此最终找到的匹配串为abczxy。第一个“abc”中首字符‘a’的位置为3，第二个“abc”中首字符‘a’的位置为11，两个的差值8就是两个字符串之间的距离，于是将字符串“abcxyz”使用长度距离对替换为: “mnoabcxyzuv(6,8)defgh”

注意：将字符串插入字典，实际是将字符串的起始位置插入到字典中，哈希只是得到一次初步匹配，用于找到符合这种初步匹配的位置，利用这个位置找到更长的匹配才是目的。

### 5.2.3 如何找到最长匹配

#### 1. 插入时字典冲突怎么办？

插入过程就是用head[ins\_h]来记录当前字符串的出现位置，ins\_h是当前字符串的哈希值，head[]数组的索引。如果将当前字符串的出现位置插到某个head[ins\_h]的时候，head[ins\_h]不为空怎么办？比如：

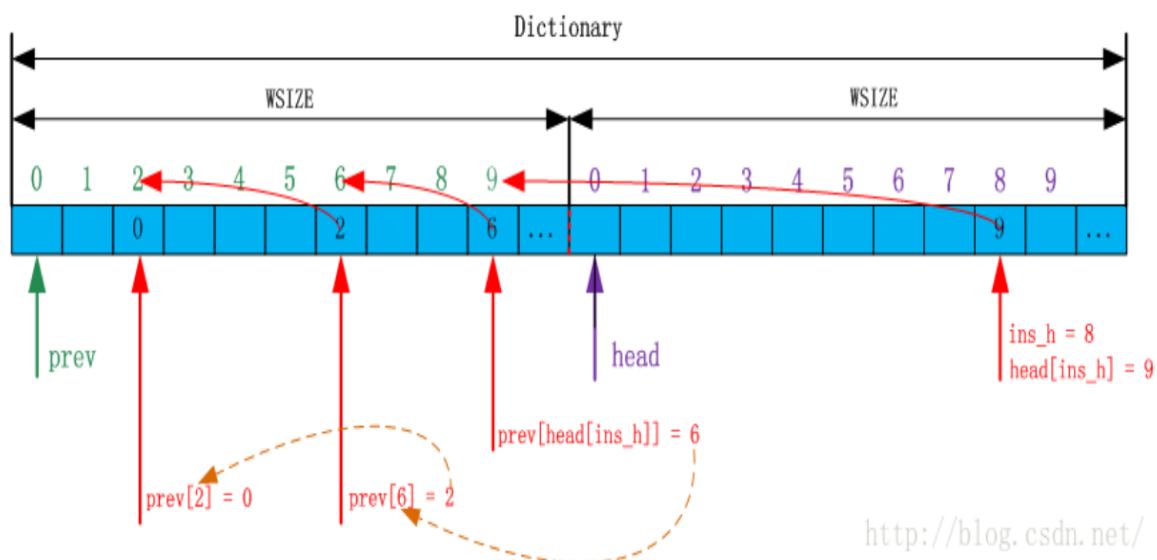
“mnoabc~~zxy~~uvwabc123456abc~~zxy~~defgh”

如果在插入第二个“abc”时，直接将字典中第一个“abc”的位置覆盖掉(因为两个字符串计算的哈希地址相同),对最终找最长子串可能会造成影响。此时就需要用到prev数组

假设：strstart是先行缓冲区第一个字符在window中的位置，数组元素prev[strstart]是前一个字符串的位置。插入按照 ([ins\_h] = strstart; prev[strstart] = head[ins\_h];

```
prev[strstart] = head[ins_h]; // 将上一个字符串的位置保存
head[ins_h] = strstart;       // 将当前字符串位置保存在head[ins_h]中
```

注意：strstart逐渐增大，所以能够保证每次对prev[strstart]赋值的时候不会把旧的内容覆盖掉，每次赋值之前，prev[strstart]一定都是全新的。



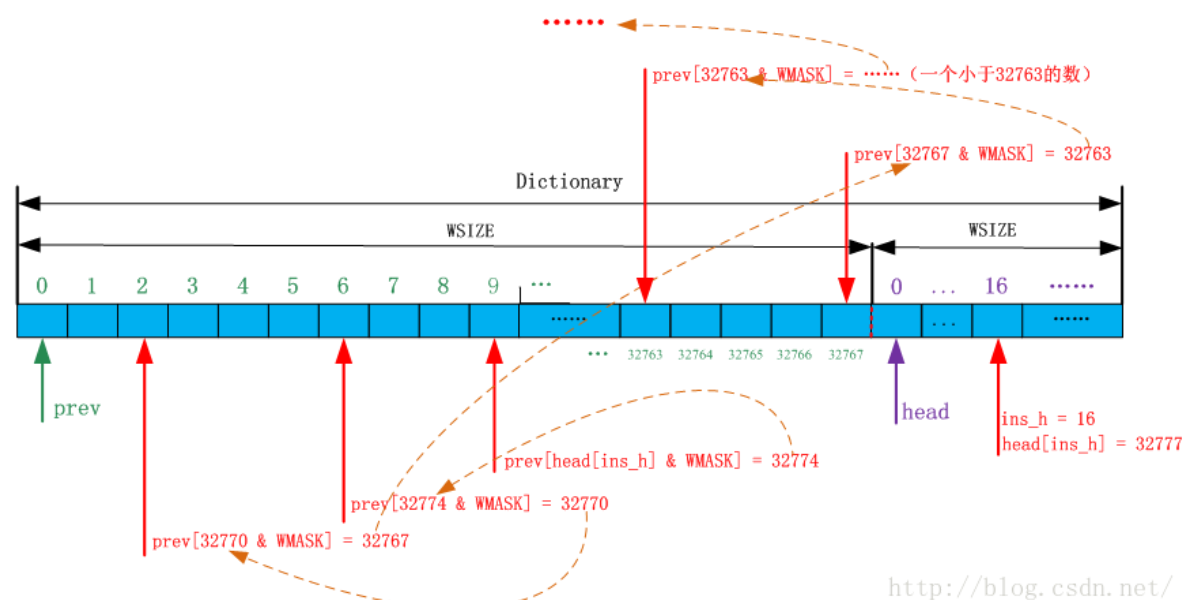


这样就形成了一种“链式”关系，这种链式关系通过prev[]数组的索引和数组元素值将同一哈希值但是位置不同的字符串链在了一起。

注意：压缩之前会把head[]数组全部初始化为0，但是不会初始化prev[]数组。prev[]数组的初始化会动态进行。因此，head[x]和prev[x]使用0作为“空”，即没有匹配串的标志，他俩的值的本身的含义是字符串的位置，但window的第0个字符组成的字符串位置也是0，这就产生歧义了，该歧义的解决方法非常简单粗暴，就是干脆不让window中的首个字符串参与匹配过程。

## 2. 先行缓冲区第一个字符位置大于32K怎么办？

随着压缩的逐字节进行，先行缓冲区的第一个字符strstart在不断推进(增大)，势必会大于32K(因为滑动窗口的大小为64K)，当strstart超过32K时，直接用其作用prev数组的下标就会出错。源码的解决办法是：让strstart与WMASK按位与运算的结果作prev[]的索引，而不是直接用strstart作prev[]的索引，其中，WMASK的值是十进制的32768，即，prev[strstart & WMASK]。这样就保证了在strstart的取值范围内，prev[]的索引与strstart都是完全对应的。



## 3. 匹配链造成死循环怎么办？

当strstart大于32K的时候，匹配链有可能出现死循环。

源码使用max\_chain\_length(表示最大链长度256)解决该问题，表示在顺着匹配链查找时，最多查找的个数。max\_chain\_length值越小，搜索的匹配节点越少，压缩速度越快，但是压缩率可能会相对低一些。

## 4. 懒匹配

源码使用两种方式找最长匹配：懒匹配和long\_match。懒匹配针对不同字符串找最长匹配，而longest\_match找当前字符串的最长匹配。

所谓懒匹配就是，虽然当前字符串找到了最适合自己的那个匹配串，但是并不急于让长度距离对儿将其替换，而是用两个变量prev\_length和prev\_match分别将匹配长度和匹配位置记录下来。然后让strstart往先行缓冲区方向挪动一个字符，现在strstart到了新的位置，当前串更新。此时再找最适合当前串的匹配串。假设找到了最适合当前串的匹配串，得到了匹配位置以及匹配长度，此时就是懒匹配大显身手的时候了。用当前匹配串的匹配长度和prev\_length做比较，如果前者大于后者，则用前者将后者更新，同时用当前的匹配位置更新prev\_match，而（当前strstart - 1）位置处的那个字符就作为一个未匹配字符来处理，当prev\_length和prev\_match完成更新后，strstart再往先行缓冲区的方向挪动一个字符并继续进行同样的处理，这种处理方式就是懒匹配；如果前者小于后者，即上次的匹配长度（prev\_length）大于这次的，那么懒匹配停止，并用上次的匹配长度和匹配位置组成长度距离对

**儿二元组完成替换。**懒惰匹配停止之后，将prev\_length和prev\_match初始化，将strstart挪到新的位置（被替换字符串之外），准备开始新一轮的懒惰匹配过程。注意，**被替换字符串中每一个字符所组成的字符串仍然要插入字典。**

懒惰匹配：“1abc23bcdefghijklm456abcdefghijklmnopq” 假设strstart在第二个“abc”中a的位置。

1. 针对第二个“abc”找匹配串，刚好在前面可以找到，匹配串位置为1，长度为3，此时先不替换，用prev\_match=1, prev\_length=3比较该次匹配，让strstart向后挪动一个字节，看接下来匹配是否比第一次长
2. strstart在第二个“abc”中b的位置且找到匹配串，匹配串位置为6，长度为12，改长度大于prev\_length,因此用新的匹配更新上一次匹配(即prev\_length=12, prev\_match=6),将位置21出的'a'作为未匹配字符处理，暂不替换，继续让strstart向后挪动一个字节，看接下来匹配是否比第一次长
3. strstart在第二个“abc”中c的位置且找到匹配串，匹配串位置为7，长度为11，该长度小于prev\_length,因此懒惰匹配终止开始替换“1abc23bcdefghijklm456a(12, 16)nopq”
4. 被替换的字符串“bcdefghijklm”每三个字符插入到字典中，后面匹配可能需要用到
5. 重新初始化prev\_length和prev\_match，不在用于后续懒惰匹配

## 5. longest\_match

longest\_match就是遍历匹配链，在匹配链中找最长子串，当然该最长子串不一定是最优的，需要结合懒惰匹配才能找到最优。

longest\_match: 在一条链中找最长匹配

“1 1abc2902abc3 3abc4564abc5klm4564abc6nopq”

注意：为了方便说明，一条链下的字符串前都增加了斜体数字，该数字只是标记，不在真正的字符串中

1. 当strstart到了4的a时，开始遍历匹配链。假设字符串 5 和“abc”的哈希值相同，那么此时通过哈希值找到的第一个匹配就是 5，但是从a和k开始逐字节比较，发现根本不同，所以5这个“匹配”放弃，
2. 继续遍历哈希链，到了 3；新的匹配是 3，从这两个字符串的首个字符a开始，逐个比较，发现一口气可以比到字符m，记录这个长度，继续往前遍历匹配链；
3. 后续匹配串分别是2 和 1，但是匹配长度都不如 3，所以就拿 3 作为当前strstart的最适合匹配串。

注意：

1. 匹配节点的遍历不是无限进行的，有一个最大匹配链节点个数限制
2. 匹配串的匹配长度也不是无限的，一个匹配串的最大匹配长度为MAX\_MATCH(258)
3. 匹配距离不是无限的，匹配距离不能超过MAX\_DIST
4. 最长匹配串的获得由longest\_match和懒惰匹配共同完成。前者负责找到当前strstart的最长匹配，后者负责在连续几个strstart的最长匹配中找到匹配串长度最长的那个作为最终的、真正的最长匹配。

### 5.2.4 找不到最长匹配怎么办？

对于压缩结果中为匹配的字符串，或者距离对与源字符如何区分？比如：

“1abc23bcdefghijklm456a(12, 16)nopq”

注意：实际在压缩字符串中距离对是没有扩招的。那么问题来了：23不是距离对，是源字符，而12,16代表距离对，那如何区分呢？

解决方式：用1个比特位来进行标记，比如1代表距离对，0代表源字符。

### 5.2.5 解压缩



“As mentioned above,there a(3,4)many kinds of wireless system(3,20)(4,42) than cellular.” 。

在解压缩时，一个一个字节进行解析：

1. 如果当前字节最高位是0，代表源字节，直接写入文件
2. 如果当前字节最高位是1，代表距离对，取出距离对，然后按照距离对的匹配起始位置以及长度解析出有效字符串。

### 5.3 GZIP中哈夫曼思想

通过前面LZ77变形思想对源数据进行语句的重复压缩之后，语句层面的重复性已经解决，但并不代表压缩效果已经达到最佳，字节层面可能也有大量重复的。比如：“BCDCDDBDDCADCBDC”

一个字节占8个比特位，那如果能对所有字节找到小于8个比特位的编码，然后用找到的编码对源文件中对应字节重新进行改写，也可以让源文件更小。那如何找编码呢？

#### 5.3.1 静态等长编码

每个字符的编码长度都相等，比如：

字符	编码
A	00
B	01
C	10
D	11

用等长编码对上述源数据进行压缩：01101110 11110111 11100011 10011110，压缩完成后的结果只占4个字节，压缩率还是比较高的。

#### 5.3.2 动态不等长编码

每个字符的编码根据具体的字符情况来确定，比如：

字符	编码
A	100
B	101
C	11
D	0

使用不等长编码对源数据进行压缩：10111011 00101001 11000111 01011

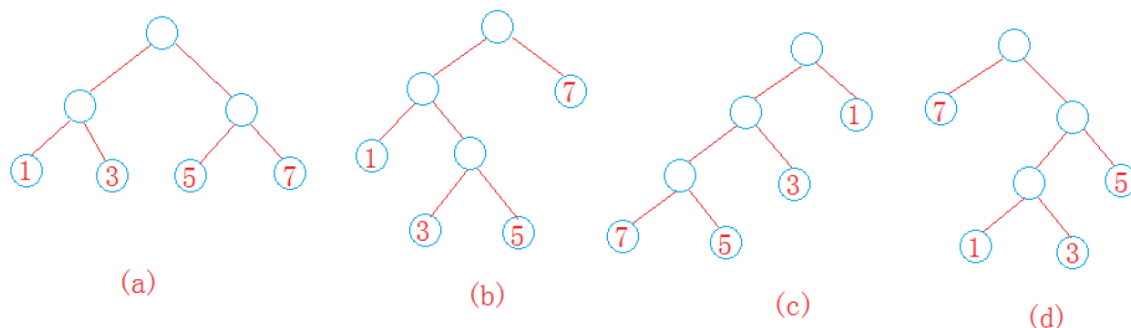
压缩完成后最后一个字节没有用完，还剩余3个比特位，显然动态不等长编码比等长编码压缩率能好点。

那动态不等长编码如何获取到呢？

#### 5.3.3 huffman编码

## 1. huffman树

从二叉树的根结点到二叉树中所有叶结点的路径长度与相应权值的乘积之和为该二叉树的带权路径长度WPL。



具有相同叶结点和不同带权路径长度的二叉树

上述四棵树的带权路径长度分别为：

- $WPLa = 1 * 2 + 3 * 2 + 5 * 2 + 7 * 2 = 32$
- $WPLb = 1 * 2 + 3 * 3 + 5 * 3 + 7 * 1 = 33$
- $WPLc = 7 * 3 + 5 * 3 + 3 * 2 + 1 * 1 = 43$
- $WPLd = 1 * 3 + 3 * 3 + 5 * 2 + 7 * 1 = 29$

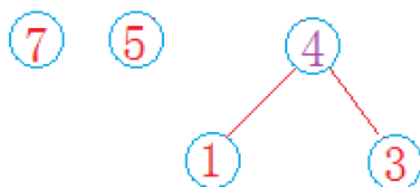
把带权路径最小的二叉树称为Huffman树。

## 2. huffman树构建

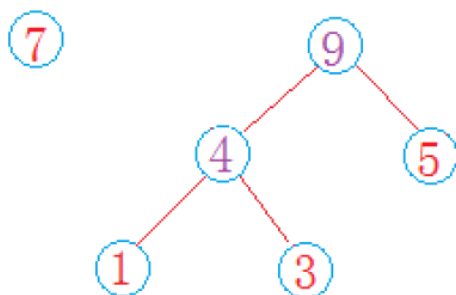
1. 由给定的n个权值{  $w_1, w_2, w_3, \dots, w_n$ }构造n棵只有根节点的二叉树森林 $F=\{T_1, T_2, T_3, \dots, T_n\}$ , 每棵二叉树 $T_i$ 只有一个带权值 $w_i$ 的根节点, 左右孩子均为空。
2. 重复以下步骤, 直到F中只剩下一棵树为止
  - 在F中选取两棵根节点权值最小的二叉树, 作为左右子树构造一棵新的二叉树, 新二叉树根节点的权值为其左右子树根节点的权值之和
  - 在F中删除这两棵二叉树
  - 把新的二叉树加入到F中



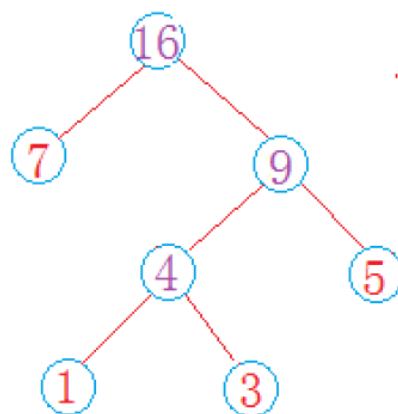
第一步



第二步



第三步



第四步

### 3. 获取huffman编码

BCDCDDDBDDCADCBDC

A→1 B→3 C→5 D→7

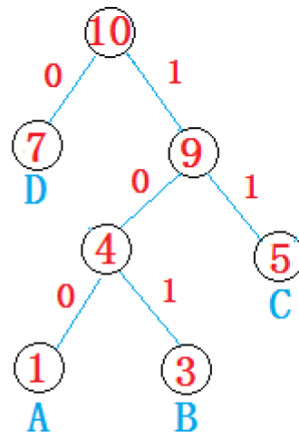
1. 以字符串中每个字符出现的总次数为权值构建huffman树
2. 另huffman树中左分支用0代替，右分支用1代替
3. 所有权值节点都在叶子位置，遍历每条到叶子节点的路径

获取字符的编码 A:100

B:101

C:11

D:0



问题：如果不等长编码中出现一个编码是另一个编码的前缀怎么办？该情况是否会出现？

### 5.3.4 利用huffman编码对源文件进行压缩

1. 统计源文件中每个字符出现的次数
2. 以字符出现的次数为权值创建huffman树
3. 通过huffman树获取每个字符对应的huffman编码
4. 读取源文件，对源文件中的每个字符使用获取的huffman编码进行改写，将改写结果写到压缩文件中，直到文件结束。

### 5.3.4 压缩文件格式

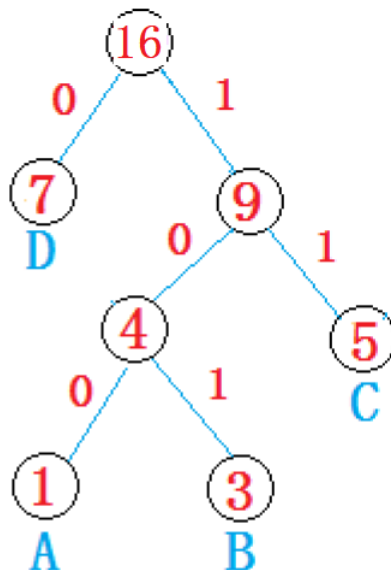
压缩文件中只保存压缩之后的数据可以吗？

答案是不行的，因为在解压缩时，没有办法进行解压缩。比如：10111011 00101001 11000111 01011，只有压缩数据是没办法进行解压缩的，因此压缩文件中除了要保存压缩数据，还必须保存解压缩需要用到的信息：

1. 源文件的后缀
2. 字符次数对的总行数
3. 字符以及字符出现次数(为简单期间，每个字符放置一行)
4. 压缩数据

### 5.3.5 解压缩

1. 从压缩文件中获取源文件的后缀
2. 从压缩文件中获取字符次数的总行数
3. 获取每个字符出现的次数
4. 重建huffman树
5. 解压缩



从压缩文件中一个一个字节的获取压缩数据，每获取到一个字节的压缩数据，从根节点开始，按照该字节的二进制比特位信息遍历huffman树，该比特位是0，取当前节点的左孩子，否则取右孩子，直到遍历到叶子节点位置，该字符就被解析成功。继续此过程，直到所有的数据解析完毕。

## 6. 与其他压缩算法的比较

### 7. 面试

1. 已经有这么多成熟压缩工具，为什么自己要在写一个？
2. 讲下你所实现的压缩算法的原理？
3. 你是如何对你自己的压缩算法进行测试的？
4. 压缩率怎么样？
5. 有没有对压缩结果进行二次压缩？二次压缩压缩率如何？为什么？
6. 还有没有改进的空间？