

Quake3 自适应 Huffman 编码

王京

v0.94, 2006 年 12 月 9 日

在 John Carmark 自由精神的鼓舞下, 本文以 GFDL 的形式发布。今后, 本人还有计划写出更多的关于 quake3 源代码分析的文章, 希望能对国内与我一样研究游戏编程的朋友们有小小的帮助, 也希望能够得到大家的鼓励与帮助。

感谢您的阅读! 如果您在阅读本文时发现了任何您认为有问题的地方, 那一定是本人在写作过程中头脑不清, 思维混乱所致! 请立即发一封 email 给我, 或直接在我的 blog 上面留言指出我的错误, 我会尽快对其进行修正。

我的 email 地址: <mailto:geeningwang@263.net>

我的 blog 空间: <http://spaces.msn.com/geeningwang>

感谢热心的 陈雄杰 对本文提出的宝贵意见!

Copyright© 2006 Wang Jing, All rights reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

本文介绍 quake3 源代码中的 huffman.c 文件，该源代码实现了自适应 Huffman 算法。源代码的注释表明，此源代码是根据 Sayood 关于数据压缩的书中的描述实现的。

1 介绍

Huffman 算法是一种用于数据压缩的算法，由 D.A.Huffman 最先提出。该算法的核心部分为 Huffman 编码树 (Huffman coding tree)，一棵满二叉树。所有可能的输入符号（通常对应为字节）在 Huffman 编码树上对应为一个叶节点，叶节点的位置就是该符号的 Huffman 编码。具体来说，一个符号对应的 Huffman 编码就是从根节点开始，沿左子节点 (0) 或右子节点 (1) 前进，一直找到该符号叶节点为止的路径对应的二进制编码。在 Huffman 编码树的基础上，该算法的编码部分输入一系列的符号，根据 Huffman 树对符号进行翻译，以符号在 Huffman 树上的位置作为编码结果。解码部分反之，根据输入的 Huffman 编码，通过查询 Huffman 树翻译回原始符号。

下面假定我们已经拥有一棵 Huffman 树，举例说明编码和解码的过程。

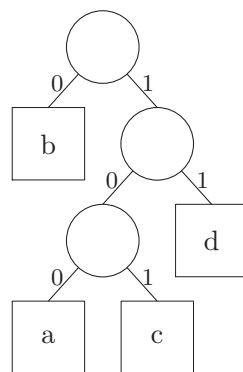


图 1 一棵 Huffman 编码树

我们以 abcd dbb 作为待编码的原始数据串为例说明利用 Huffman 编码树进行编码的过程。第一个出现的字符为 a，从 Huffman 的根节点出发，经过右子树、左子树、左子树三步后，我们到达了包含字符 a 的叶节点。同时，我们获得了 a 字符的二进制编码：100。下一个字符为 b，b 在 Huffman 编码树上的位置非常高，仅从根节点出发向左子树前进一步即可达到，因此编码也最短，为 0。类似的，我们将所有的原始数据经过这一编码过程后，得到了最终的编码结果：1000101111100。

反过来，假如我们首先得到了 Huffman 编码结果，能不能顺利的恢复出原始数据呢？让我们试验一下。对于一个二进制 bit 流 1000101111100，逐 bit 进行处理。我们首先将出发位置放置于根节点，并读入一个 bit，值 1 代表向右子树前进一步。然后是 0，向左子树前进一步。然后又是 0，再向左子树前进一步。这时，我们遇到了一个叶节点 a，这就是给定

得 Huffman 编码恢复出的第一个符号。接着，我们将当前位置恢复至根节点，并继续读入二进制 bit 流。下一个读入的值为 0，代表向左子树前进一步。由于直接发现了叶节点，所以立刻恢复出符号 b。类似的，逐一处理所有的 bit，将恢复出我们所期望的原始符号数据串 abcd dbb。

在这个例子中，我们已经可以发现，通过对出现几率高的符号赋以较短的编码、对出现几率低的符号赋以较长的编码，可以有效的对输入数据进行压缩，这也就是 Huffman 编码提出的目的。而如何构造出编码树，便成为一个重要的问题。在下一节中，我们将具体介绍编码树的构造方案。

原始的 Huffman 算法给出了一种静态的编码树构造方案，要求在实际编码之前统计被编码对象中符号出现的几率，并据此进行编码树的构造。但应用此方案时必须对输入符号流进行两遍扫描，这在许多实际的应用场合是不能接受的。另外，静态编码树构造方案不能对符号流的局部统计规律变化做出反应，因为它从始至终都使用完全不变的编码树。因此，后人（谁？）提出了自适应 Huffman 编码方案。这种方案在不需要事先构造 Huffman 树，而是随着编码的进行，逐步构造 Huffman 树。同时，这种编码方案对符号的统计也动态进行，随着程序的运行，同一个符号的编码可能发生改变（变得更长或更短）。这样就解决了静态编码树面临的主要问题。（这里隐含了一个假设，就是在一个数据块中，已经出现的符号出现概率越高，则未来可能出现的概率也越高，这个假设在大多数情况下是成立的。）

本文主要介绍两种 Huffman 编码树的构造方案，并对第二种 Huffman 编码树的构造方案，即自适应 Huffman 编码方案的一种特定实现进行了分析。在讨论构造方案时，虽然给出了 Huffman 编码树的定义，但并不证明为什么通过某种构造方案构造出的编码树就是 Huffman 编码树。如果读者需要进行理论方面的研究，请进一步参考相关的资料。

2 Huffman 编码树的构造原理

通过第一节的介绍，可以发现，Huffman 算法之中的编码和解码过程都相对简单，而如何构造 Huffman 编码树成为问题的关键。在这一节中，首先介绍静态 Huffman 编码树的构造，然后在此基础上，介绍自适应 Huffman 编码树的构造。两种 Huffman 编码树的构造都使用具体的例子进行说明，以便读者理解。

所谓 Huffman 编码树，就是一棵具有最小加权路径长度（weighted path length, WPL）的一棵满二叉树。而树的加权路径长度，就是树中所有的叶节点的权重值（通常为符号出现的频率）乘上其到根节点的路径长度（根节点本身具有的路径长度为 0，叶节点到根节点的路径长度为叶节点的层数）。树的加权路径长度记为：

$$WPL = W_1 \cdot L_1 + W_2 \cdot L_2 + \cdots + W_n \cdot L_n$$

其中 N 个权重值 $W_i (i = 1, 2, \dots, n)$ 构成一棵有 N 个叶节点的二叉树，相应的叶节点的路径长度为 $L_i (i = 1, 2, \dots, n)$ 。

值得注意的是，虽然一棵 Huffman 树一定是满二叉树，但一棵满二叉树却不一定是 Huffman 树（你能举例说明么？）。某一种特定的满二叉树构造方案必须经过证明满足 *WPL* 最小的条件，才能说这种方案是 Huffman 树构造方案。另外，所有 *WPL* 最小的编码方案之间都是等价的（可以通过有限次的左子树与右子树之间的交换，得到完全相同的两棵树）。可以证明（但本文不证明☺），使用以下两种构造方案构造出的 Huffman 编码树的 *WPL* 都是最小的。

2.1 静态 Huffman 编码

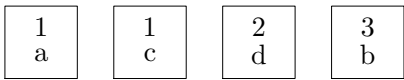
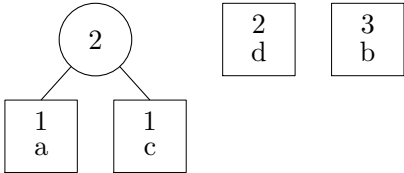
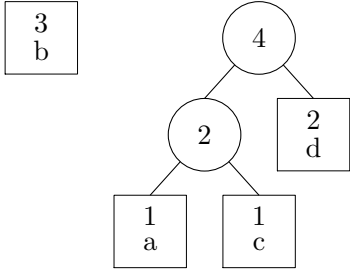
建立静态 Huffman 编码树的过程相对简单。首先，按照权重值的大小将符号集合排序。接着，取出权重值最小的两个集合元素作为叶节点组成一棵子树，子树的权重值为两个叶节点的权重值之和。然后将新产生的子树放回原来的集合，并保持集合有序。重复上述步骤，直至集合中只剩下一个元素，则 Huffman 编码树构造完成。

下面我们同样以 `abcd dbb` 作为待编码的原始数据串为例，构造静态 Huffman 编码树。首先，我们需要统计出 `a`, `b`, `c`, `d` 四个符号分别在原始数据串中的出现频率。统计结果如表 1 所示：

符号	a	b	c	d
频率	1	3	1	2

表 1 数据串 `abcd dbb` 的频率统计

然后，按照前面提到的构造方法，经过图 2 的四个步骤，即可获得起基于表 1 频率统计的静态 Huffman 编码树。

步骤 1		初始状态，将每个符号看作一棵只有一个叶节点子树，按权重值排序。
步骤 2		将权重值最小的两个元素 <code>a</code> , <code>c</code> 组合为一棵子树，并按权重值排序。
步骤 3		再次将权重值最小的两个元素组合为一棵子树，并按权重值排序。

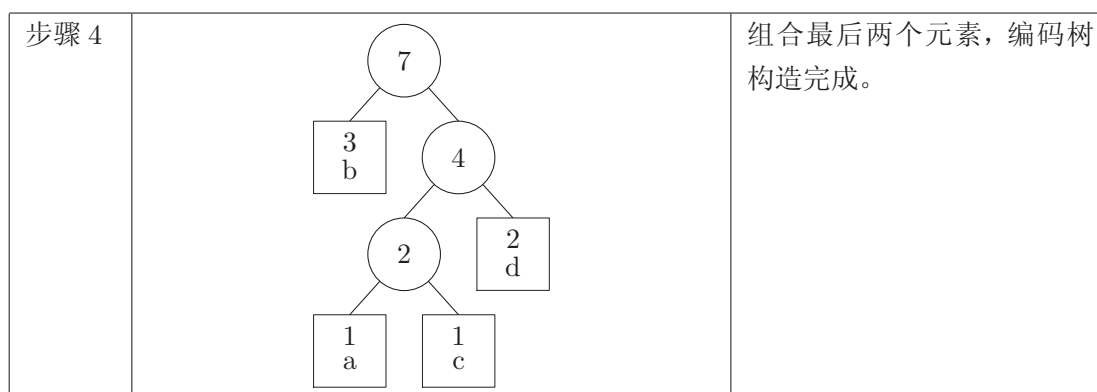


图 2 建立静态 Huffman 编码树

到此为止，我们建立起了给定符号串的 Huffman 编码树。可以发现，此树与图 1 中的 Huffman 编码树等同，第一节中编码、解码的例子使用的就是此 Huffman 编码树。

2.2 自适应 Huffman 编码

上文已经提到，使用基于静态 Huffman 编码树的算法对输入的符号流进行编码，必须进行两遍扫描。第一遍扫描统计被编码对象中符号出现的几率，并创建 Huffman 编码树，第二遍扫描按照 Huffman 编码树对输入符号进行编码。并且，在储存或传输 Huffman 编码结果之前，还必须先储存或传输 Huffman 编码树。这些问题使静态 Huffman 编码树并不实用，尤其是对于短小的符号流来说，加上 Huffman 编码树的编码结果在尺寸上很可能更大，甚至大出很多。

为了解决这些问题，自适应 Huffman 编码应运而生。严格的说，自适应 Huffman 编码不仅涉及到编码树的构造问题，还与编码、解码过程相关。由于其实用性大大提高，因而应用领域也非常广泛。这看似复杂的自适应 Huffman 编码方案背后的概念却十分简单，如表 2 所示。

```

初始化编码树;
对每一个输入符号
{
    对符号编码;
    更新编码树;
}

```

表 2 自适应 Huffman 编码算法伪代码

解码算法与编码算法类似，当编码方和解码方都使用相同的初始化状态，以及相同的更新编码树策略时，在编码方输入的符号将正确的在解码方还原。现在，问题变为如何更新编

码树。为了让压缩编码过程具有自适应性，我们应该在每次处理完一个符号时调整该符号的权重值（加一），并重新建立 Huffman 编码树。但是很明显，这样的算法将有极差的效率。为了增加算法的效率，在实际的自适应 Huffman 编码算法中使用了一个技巧：每次只更新受影响的那一部分编码树结构。

初始化编码树时，由于只允许对待编码数据流进行单遍扫描，因此不可能预先知道各种符号的出现频率。为了对所有符号一致对待，编码树的初始状态只包含一个叶节点，包含符号 NYT (Not Yet Transmitted, 尚未传送)，权重值为 0。NYT 是一个逸出码 (escape code)，不同于任何一个将要传送的符号。当有一个尚未包含在编码树中的符号需要被编码时，系统就输出 NYT 编码，然后跟着符号的原始表达。当解码器解出一个 NYT 之后，它就知道下面的内容暂时不再是 Huffman 编码，而是一个从未在编码数据流中出现过的原始符号。这样，任何符号都可以在增加到编码树之前进行传送，虽然这样做不仅没有对数据进行压缩，反而使其更加冗长，但至少是一种对不存在于编码树之中的符号进行编码的手段。

包含 NYT 符号的节点还有另外一个作用，就是作为新符号的插入点。在需要插入一个新符号时，总是先构造一个新的子树，子树包含 NYT 符号与新符号两个叶节点，然后将旧的 NYT 节点由这个子树替代。由于包含 NYT 符号的节点权重值为 0，而包含新符号的叶节点的权重值为 1，因此最终效果相当于原 NYT 节点位置的权重值由 0 变为 1。因此，下一步将试图对其父节点执行权重值“加一操作”。

对符号编码的过程与第一节中给出的方法完全一致。每次符号编码完成之后，也将试图对包含符号的节点执行权重值“加一操作”。

将一个新的符号插入编码树或者输出某一个已编码符号后，相应的符号的出现次数增加了 1，继而编码树中各种符号的出现频率发生了改变。此时，原有的 Huffman 编码树已经不一定符合具有最小加权路径长度这一条件，因此必须进行调整，以使其继续满足这一条件，保持其合法 Huffman 编码树的状态。这一操作，在本文中称为“加一操作”。

在说明“加一操作”的调整方法之前，我们必须给每个节点引入两个新的属性：节点编号 (node number) 和所属块 (block)。其中节点编号是一个全局唯一的值，不同的节点拥有不同的节点编号，而块指具有相同权重的一组节点。节点编号具有一些特点：

- (1) 权重值较大的节点，节点编号也较大；
- (2) 父节点的节点编号总是大于子节点的节点编号。

以上两点称为兄弟属性 (sibling property)。在每一次调整节点权重值时，都需要相应的调整节点编号，以避免兄弟属性被破坏。在对某一个节点权重值进行“加一操作”时，应该首先检查该节点是否具有所在的块中的最大节点编号，如果不是，则应该将该节点与所在块中具有最大节点编号的节点交换位置。然后再对节点的权重值加一，这样，由于该节点的节点编号已经处于原来所属块中的最大值，因此权重值加一之后兄弟属性仍然得到满足。最后，由于节点的权重发生了变化，必须递归地对节点的父节点进行“加一操作”。

图 3 就是对一个输入符号进行编码并更新编码树的流程图，此图取自参考文献 [5]。

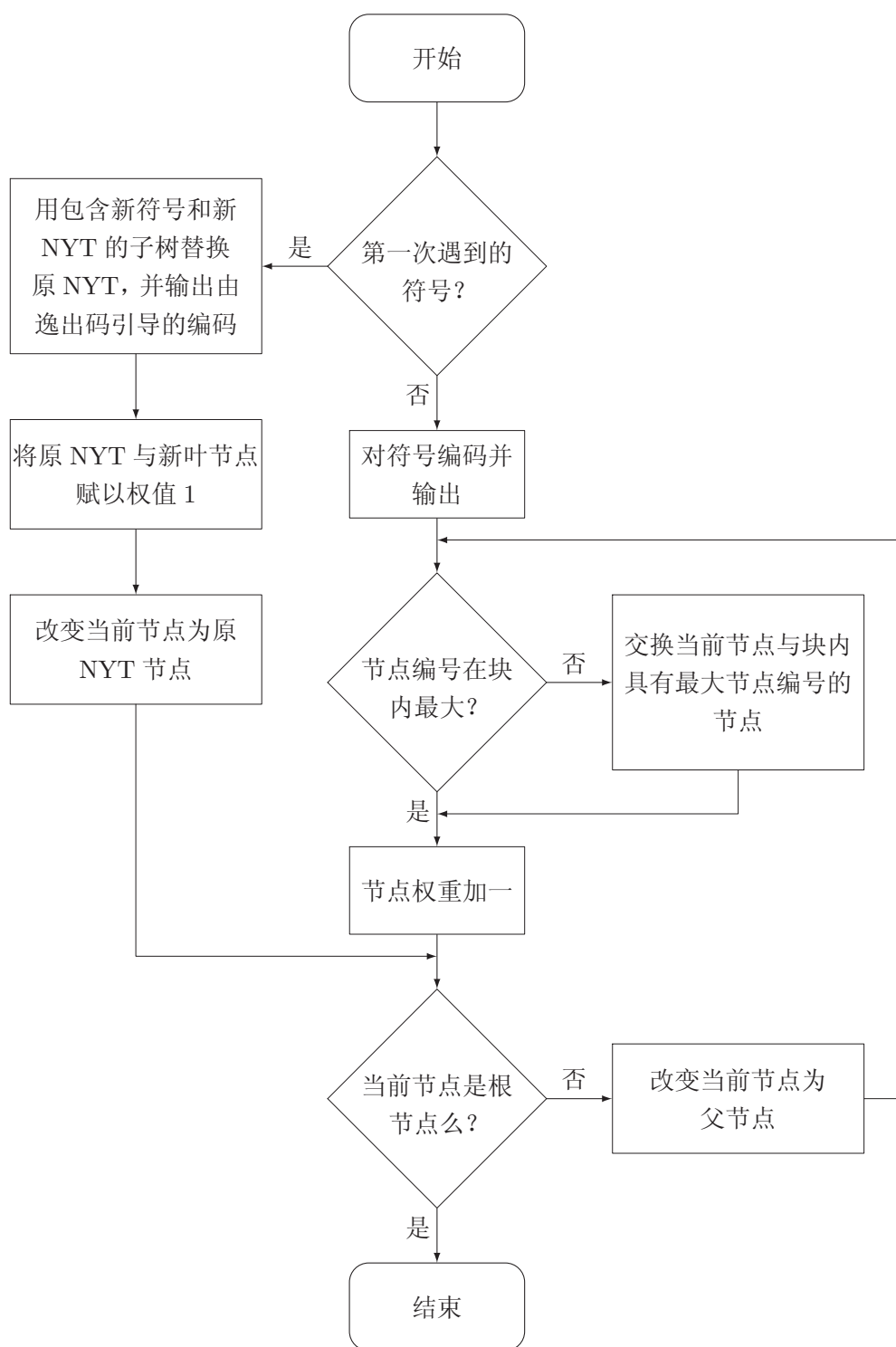

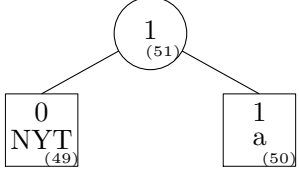
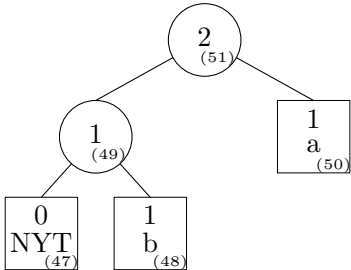
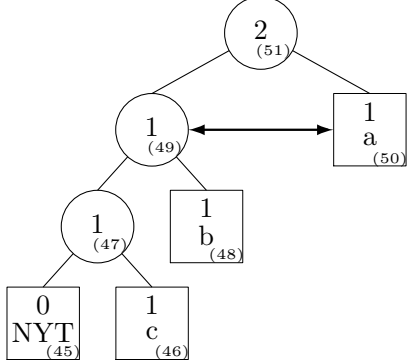
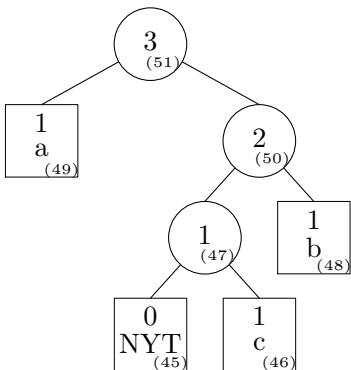
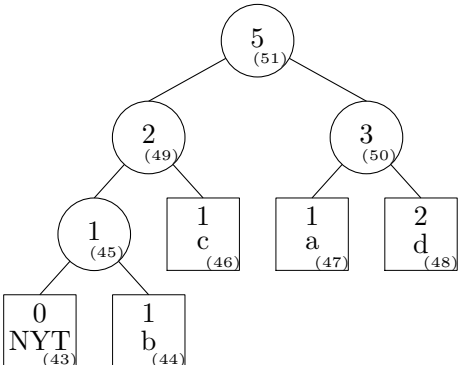
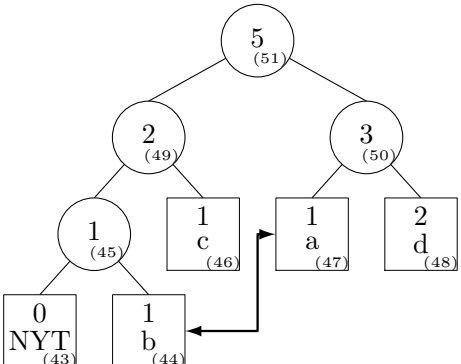
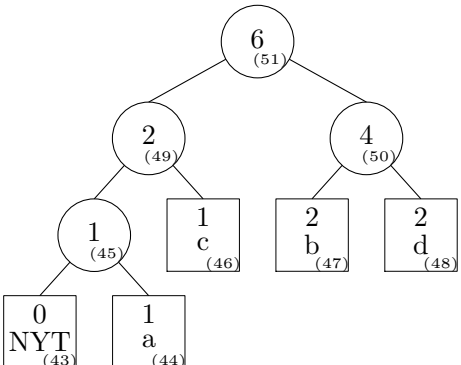
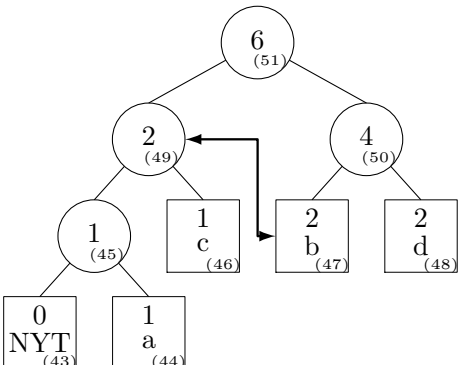


图 3 自适应 Huffman 编码算法对一个输入符号进行编码并更新编码树流程图

下面我们仍以 abcd dbb 作为待编码的原始数据串为例，演示自适应 Huffman 编码过程。整个编码过程如图 4 所示。为了与参考文献 [4] 保持一致，节点的从 51 开始向下增长。在本文第四节中，可以看到节点编号在 quake3 中的具体实现方式。

步骤 1		初始状态，仅有唯一的 NYT 节点，NYT 节点的权重为 0。
步骤 2		输入符号：a 输出编码：a 使用包含新 NYT 节点和字符 a 节点的子树，替换旧的 NYT 节点。
步骤 3		输入符号：b 输出编码：0b 使用包含新 NYT 节点和字符 b 节点的子树，替换旧的 NYT 节点。 对 51 号节点（根节点）执行权重值“加一操作”。
步骤 4		输入符号：c 输出编码：00c 使用包含新 NYT 节点和字符 c 节点的子树，替换旧的 NYT 节点。 将对 49 号节点执行权重值“加一操作”，但 49 号节点不具有所在的块中的最大节点编号，因此需要先与 50 号节点进行交换位置操作。
步骤 5		新的 50 号节点权重值加一。 对 51 号节点执行权重值“加一操作”。

<p>步骤 6</p>		<p>输入符号: d 输出编码: 100d 使用包含新 NYT 节点和字符 d 节点的子树, 替换旧的 NYT 节点。 将要对 47 号节点执行权重值“加一操作”, 但 47 号节点不具有所在的块中的最大节点编号, 因此需要先与 49 号节点进行交换位置操作。</p>
<p>步骤 7</p>		<p>新的 49 号节点权重值加一。 对 51 号节点执行权重值“加一操作”。</p>
<p>步骤 8</p>		<p>输入符号: d 输出编码: 001 将要对 44 号节点执行权重值“加一操作”, 但 44 号节点不具有所在的块中的最大节点编号, 因此需要先与 48 号节点进行交换位置操作。</p>

步骤 9		<p>新的 48 号节点权重值加一。</p> <p>对 50 号节点执行权重值“加一操作”。</p> <p>对 51 号节点执行权重值“加一操作”。</p>
步骤 10		<p>输入符号: b</p> <p>输出编码: 001</p> <p>将要对 44 号节点执行权重值“加一操作”，但 44 号节点不具有所在的块中的最大节点编号，因此需要先与 47 号节点进行交换位置操作。</p>
步骤 11		<p>新的 47 号节点权重值加一。</p> <p>对 50 号节点执行权重值“加一操作”。</p> <p>对 51 号节点执行权重值“加一操作”。</p>
步骤 12		<p>输入符号: b</p> <p>输出编码: 10</p> <p>将要对 47 号节点执行权重值“加一操作”，但 47 号节点不具有所在的块中的最大节点编号，因此需要先与 49 号节点进行交换位置操作。</p>

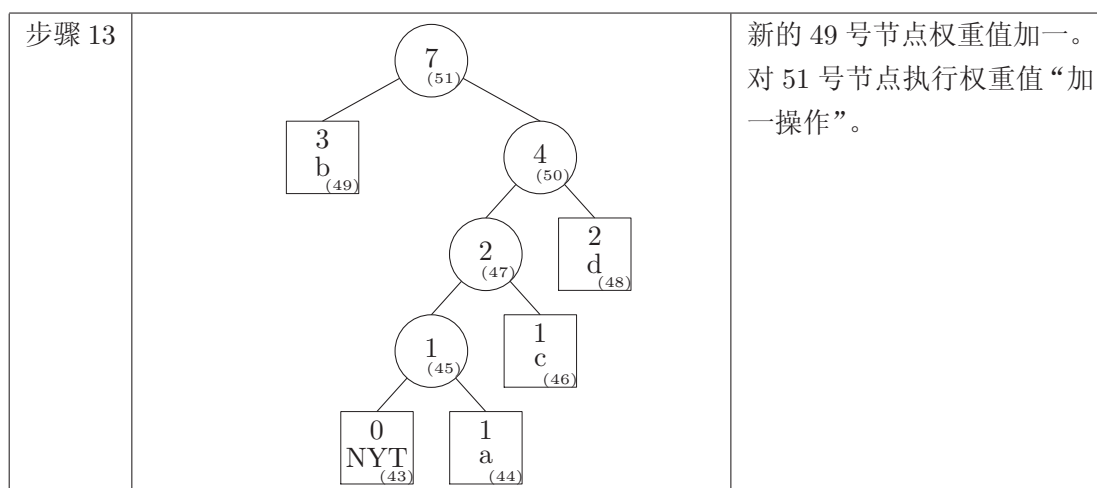


图 4 自适应 Huffman 编码过程

通过观察以上步骤，容易发现自适应 Huffman 编码的几个特征：

- (1) 在步骤 13 得到的编码树与静态 Huffman 编码树基本相同，除了 NYT 节点和符号 a 节点组成的子树替代了静态 Huffman 编码树中的符号 a 的叶节点之外；
- (2) 在每一次输入新的符号之前，Huffman 树都处于完整可用的正常状态；
- (3) 同一个输入符号，可能产生多种不同的输出。例如三次输入的符号 b，产生的输出分别为 0b、001 和 10；
- (4) 同样的输出结果，可能由不同的输入产生。例如第二次输入的符号 d 与第二次输入的符号 b，都产生了 001 作为输出结果。

这些特征首先说明了自适应 Huffman 编码树与静态 Huffman 编码树等同，完全符合 Huffman 树的定义。同时，由于每一个输入符号都对编码树产生了影响，因此解码过程无法从 Huffman 编码数据流的某一个中间位置开始进行，而必须从头至尾逐 bit 处理。

由于自适应 Huffman 编码算法采用了先编码，后调整编码树的方案，相应的解码算法比较简单。解码算法也使用仅有唯一的 NYT 节点的编码树作为初始状态，然后根据 Huffman 编码数据流，对符号进行还原。每次处理完一个符号，就使用这个符号调整编码树。这样，在每一次输入新的符号之前，Huffman 树都处于与进行编码时使用的 Huffman 树完全相同的状态，保证了解码的正确性。

3 接口

经过艰苦努力，以上两节终于将自适应 Huffman 算法介绍完毕。在下面的三节中，我们将研究 quake3 源代码中关于 Huffman 算法的具体实现。在本节中，首先介绍函数接口，然后在第四节中介绍具体用于实现 Huffman 算法的数据结构，最后在第五节中逐函数进行说

明。

huffman.c 的调用接口包括：

Huff_Compress	压缩一个数据块；
Huff-Decompress	解压缩一个数据块；
Huff_Init	初始化数据结构；
Huff_addRef	增加对某一个字符的引用；
Huff_offsetReceive	解压缩一个字节，从缓冲区的某一个指定位置；
Huff_offsetTransmit	压缩一个字节，从缓冲区的某一个指定位置；
Huff_putBit	向缓冲区输出一个 bit；
Huff_getBit	由缓冲区输入一个 bit；

表 3 huffman.c 调用接口

从 quake3 整体观察，对 huffman.c 的调用有两种形式。第一种形式，就是对一个数据块进行普通的自适应 Huffman 编码，这种形式的仅需要使用前两个函数即可完成。虽然在自适应 Huffman 编码中对编码树的调整非常快速，但仍然需要一定的时间。因此，在 msg.c 模块中，对 huffman.c 的调用基于第二种形式。在这种形式中，程序首先调用 Huff_Init 初始化编码树，然后根据预先统计好的符号出现概率调用 addRef，构造 Huffman 编码树，然后在运行时使用 Huff_offsetReceive 和 Huff_offsetTransmit 函数进行编码和解码，而不再调整 Huffman 编码树的结构。这种方式本质上就是静态 Huffman 编码。

另外，从上述接口声明可以看出 huffman.c 的编码似乎不太认真，函数名称大小写不太规范。同时，由于 huffman.c 模块与 msg.c 模块之间功能分割不清晰，msg.c 中对 Huffman 编码缓冲区有直接操作，才需要最后两个函数接口，Huff_putBit 和 Huff_getBit。

4 数据结构

每个节点包含 8 项数据，其中实际数据包括 symbol 和 weight 两项。symbol 表示了节点代表的符号，有 258 种可能的取值。0-255 表示了一个真实的字节值；NYT(not yet transmitted, 256) 具有两重含义：首先，它是一个逃逸码 (escape code)，在编码流中代表其后跟随的 8 bit 不再是编码，而是一个新的符号；另外，内存中的 NYT 节点代表新节点的插入位置；INTERNAL_NODE(257) 表示该节点不是一个叶节点，不包含实际的符号。weight 表示了某一个节点的权重。

每个节点使用 left, right, parent 三个变量形成了二叉树结构。left 指向左子树，right 指向右子树，parent 指向父节点。

节点编号和块这两个概念没有使用直接的数字表示，而是使用了 next, prev 以及 head 三个指针进行描述。我想这里需要稍微详细一点，虽然在第二节的例子中，节点编号使用

了一个数字表示，但实际上数值对于节点编号而言是没有意义的。本质上来说，节点编号仅表示了节点之间的一个顺序关系。因此，在 huffman.c 中，节点编号使用一个双向链表表示，next 和 prev 就表示了双向链表的前向和后向指针。变量 head 是一个双重指针，处于同一个块的节点，head 指向了同一个位置。该位置是一个节点指针，又返回来指向该块中节点编号最大的节点。这一关系图 5 所示。

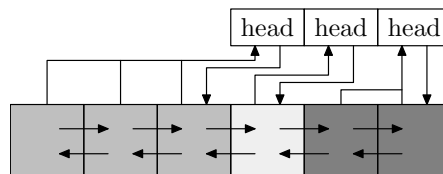


图 5 以 head 指针标示出的 block

实际上，这一双向链表完全可以使用一个固定数组代替。因为这里实际只需要两个操作：从追加新节点和交换两个节点。当出现新字符时，原 NYT 需要扩展成两个节点。如果将节点编号从大到小存放，则新的节点只会在最后面追加。而交换节点，只需要交换节点的值和部分指针。这些操作在固定数组上实现就足够了。（仍然有一个问题：为什么不把 head 更简单的设计为一个指针，直接指向节点所处块中节点编号最大的节点？）

整个节点结构如表 4 所示，定义了节点结构之后，还需要定义整个编码器（或解码器）所需要的数据结构。其中包括：

- (1) 树结构根节点指针 tree;
- (2) 节点存储空间 nodeList 以及当前分配位置 blocNode;
- (3) 节点的符号索引 loc;
- (4) 用于表示节点编号的双向链表的头指针 lhead 和尾指针 ltail（实际上没有使用）;
- (5) 用于保存指向每一个块中具有最大节点编号的指针的空间 nodePtrs 以及当前分配位置 blocPtrs，另外这块空间可能从中间释放或再分配，因此使用了空闲链表结构，空闲链表的头指针为 freelist;

```
typedef struct nodetype {
    struct nodetype *left, *right, *parent; /* tree structure */
    struct nodetype *next, *prev;          /* doubly-linked list */
    struct nodetype **head;                /* highest ranked node in block */
    int    weight;
    int    symbol;
} node_t;
```

表 4 节点结构

以上就是 huffman.c 中所用到的所有数据项。由于以 8 bit 字节作为符号表，整个树可能占用的空间是有限的，因此 huffman.c 中没有使用任何动态分配内存，所有的可能用到的内存都静态分配完成。（nodeList 与 nodePtrs 都使用了 3×256 ，也就是 768 作为下标，是不是过多了？理论上的最大值是多少？）

5 函数实现

5.1 Huff_putBit

此函数将一个 bit 放入缓冲区，并调整作为函数参数的 offset 的值。实际上，此函数与 Huffman 算法无关。

5.2 Huff_getBit

此函数从缓冲区读出一个 bit，并调整作为函数参数的 offset 的值。实际上，此函数与 Huffman 算法无关。

5.3 add_bit

此函数将一个 bit 放入缓冲区。

5.4 get_bit

此函数从缓冲区读出一个 bit。

5.5 get_ppnode

从 nodePtrs 中分配一个单元（node_t ** 类型）以备用。如果可能，先从 freelist 列表中分配；否则从未分配的空间中（由 blocPtrs 指出）分配。

5.6 free_ppnode

将一个单元（node_t ** 类型）还回 nodePtrs。总是还回 freelist 列表中。

5.7 swap

交换编码树上的两棵子树。两棵子树不能为父子关系。

5.8 swaplist

交换双向链表上的两个节点。实际上就是交换了两个节点的节点编号。

5.9 increment

实现了“加一操作”。分以下几步完成：

- (1) 检查节点是否有效，无效则直接返回。这是递归中止条件。
- (2) 检查节点编号在块内是否最大。如果不是，则交换当前节点与块内具有最大节点编号的节点在编码树上的位置，同时交换它们的节点编号。
- (3) 如果节点处于一个块中，那么现在它必然处于块的尾部，令节点从块中脱离出来。
- (4) 对节点权重值加一。
- (5) 检查权重值增加之后的节点是否与下一个块具有相同的权重值，如果是，则加入下一个块。
- (6) 对父节点递归执行“加一操作”。由于父节点总是在子节点之后进行“加一操作”，因此有必要在父节点完成操作之后再次调整父子节点之间的节点编号关系，以符合兄弟属性。

5.10 Huff_addRef

使用某一个符号更新编码树。如果符号已经存在于编码树之中，则直接调用 increment 函数进行处理。否则，需要在原 NYT 符号节点处创建一棵子树，包含新 NYT 以及新符号两个叶节点。

将原有 NYT 节点由一个子树代替这一操作，可以由插入一个新的叶节点以及一个新的子树节点两个操作完成。实际上的 NYT 节点没有发生销毁再分配的操作，只是改变了在编码树上的位置。由于 NYT 节点的权重值总为 0 而且是编码树中唯一的权重值为 0 的节点，所以它总处于节点编号列表的最前面。新建的符号节点以及子树节点的权重值都为 1，所以两个新建节点在节点编号列表中总是插入在 NYT 节点之后。按照两者的父子关系，总是先插入子树节点，然后插入新符号节点。这样，最后形成的节点编号列表形式为：

NYT 节点→新符号节点→子树节点→……

在创建新符号节点完成之后，编码树中一些节点的权重值不再正确，需要对新子树的父节点进行“加一操作”以恢复编码树的正确性。

5.11 Huff_Receive

解码一个字节。从缓冲区中逐一读出 Huffman 编码 bit，并在编码树中进行查找。直到找到一个叶节点位置。

5.12 Huff_offsetReceive

从缓冲区的一个指定位置解码一个字节。从缓冲区中逐一读出 Huffman 编码 bit，并在编码树中进行查找。直到找到一个叶节点位置。

5.13 send

对一个节点进行编码。递归调用自身。

5.14 Huff_transmit

编码一个字节。如果尚不在符号内，则编码为“NYT，符号”的形式。

5.15 Huff_offsetTransmit

从缓冲区的一个指定位置编码一个字节。

5.16 Huff-Decompress

对数据缓冲区解码。

5.17 Huff_Compress

对数据缓冲区进行编码。

5.18 Huff_Init

初始化 Huffman 编码树。

参考文献

- [1] 笨笨数据压缩教程, <http://www.contextfree.net/wangyg/a/tutorial/benben.html>, 王咏刚, 2003. (很不错的教程, 可惜作者只完成了前半, 没有写完)
- [2] *Huffman* 编码简介. (网上到处都是, 实在不能考证出原作者了, 如果哪位读者知道确切作者, 请一定与我联系)
- [3] 数据结构与算法分析, Clifford A. Shaffer, 张铭、刘晓丹译, 电子工业出版社, 1998.
- [4] *Adaptive Huffman Compression*, <http://www.cs.sfu.ca/cs/CC/365/li/squeeze/AdaptiveHuff.html>, Ze-Nian Li, 2006. (在这个页面上, 有一个用 java 写出的 applet 小程序, 以图形界面动态的显示了自适应 Huffman 树的构造过程, 非常有助于加深理解, 值得一看!)
- [5] *Introduction to Data Compression, 2nd Edition*, Sayood Khalid, 2000. (还没真的看到呢☺)