声明

本论文题目：区间编码算法的分析与实现，作者：叶叶，于 2010 年 10 月 15 日在编程论坛上发表。页面地址：http://programbbs.com/bbs/view12-29351-1.htm 。本论文全文及相关配套程序可以在上述页面中下载。请尊重他人劳动成果，转载或引用时请注明出处。

目录

区间编码算法的分析与实现

作者：叶叶（网名：yeye55）

摘要：全面介绍了区间编码算法的理论基础和实现方式。详细讨论了区间编码原理、正规化操作、区间编码实现、二进制索引树的理论和实现等技术。并给出了一个切实可行的应用程序。

关键词：区间编码；算术编码；二进制索引树；Delphi

中图分类号：TP301.6

1 前言

区间编码是一种基于统计模型的无损压缩算法。G. N. N. Martin 在 1979 年的视频和数据记录会议（Video & Data Recording Conference）上提交了一篇论文：《区间编码：去除数字信息中冗余的算法[1]》（Range encoding: an algorithm for removing redundancy from a digitised message.），第一次提出了区间编码算法的思想。目前，区间编码的实现都是基于该论文[1]中描述的方法。尽管从本质上说区间编码与算术编码是相同的，但是根据该论文[1]的发表年代，通常认为区间编码算法不受与算术编码算法相关的专利约束。正是因为如此，越来越多的研究人员将目光转向了区间编码算法。

与经典的哈夫曼编码相比，区间编码可以获得更高的压缩率。因为传统上的哈夫曼编码是以位作为单位为符号分配编码。即使一个符号具有非常高的频度，哈夫曼编码也只能为其分配一个位的编码。这限制了压缩率的进一步提升。与哈夫曼编码不同，区间编码将所有的数据映射到一个整数区间内。然后输出一个属于该区间的整数作为输出编码。这意味着区间编码可以无限的接近数据的熵极限。另外，区间编码由于其特点可以很好的与高阶模型相配合。

目前，区间编码已经开始大量的应用。本文将详细的分析区间编码的理论基础和实现方式，并给出一个在 Delphi 7.0 下开发，使用区间编码算法压缩数据的应用程序。

2 理论

2.1 编码

区间编码使用一个整数区间，并在整数区间上进行整数运算。对于任意的一个整数可以表示为基数 b 的 w 次方，即：b^w。在这里为了说明和运算的方便，将基数 b 设为 10。同时，设有一个整数区间(i | L ≤ i ≤ H)，即：[L,H]。L 称之为区间的下沿，H 称之为区间的上沿。那么区间范围设为 R，即：R = H - L + 1。另外，令 T 为所有符号的总计频度（Total Frequency）。令 f_S 为符号 S 的频度（Frequency）。令 F_S 为符号 S 的累积频度（Cumulative Frequency）。累积频度就是符号值小于 S 的其它符号的频度总和。累积频度可以用以下公式进行计算。

$$F_S = \sum_{x<S} f_x$$

假设有一份数据由 4 种符号组成，这 4 种符号为“A”、“B”、“C”、“D”。现在，先假设编码仅有一个符号的数据“D”，让我们来看看编码是如何进行的。

在编码前先要进行一些初始化。首先，为了防止运算中出现 0 频的问题，令所有符号初始时的频度都为 1。然后，令初始时区间的范围 $R = b^w$。这里设 w 为 2，即：初始区间范围 R = 100，区间为[0,99]。编码时可以根据每个符号的频度、累积频度以及总计频度，计算出符号在区间[L,H]中的映射区间[L’,H’]。其计算公式如下所示。

$$R' = R \text{ div } T * f_S \qquad \text{（公式 1）}$$
$$L' = L + R \text{ div } T * F_S \qquad \text{（公式 2）}$$
$$H' = L + R \text{ div } T * (F_S + f_S) - 1 = L' + R' - 1 \qquad \text{（公式 3）}$$

其中 div 表示整除运算。R’表示映射区间[L’,H’]的范围。在当前的例子里，总计频度 T 为 4。根据以上公式计算后的结果如表 2.1 所示。

符号	频度	累积频度	映射区间	区间范围
A	1	0	[00,24]	25
B	1	1	[25,49]	25
C	1	2	[50,74]	25
D	1	3	[75,99]	25

表 2.1 映射区间的计算

编码后可以输出一个整数数值 V，使得 V 属于区间[L’,H’]，即：L’ ≤ V ≤ H’。如果 V 可以分解成 $V = V' * b^n$，那么实际上只需要输出 V’即可，b^n 部分可以省略。例如，对于符号“D”，其映射区间为[75,99]。这里可以选择 V = 80，80 属于区间[75,99]，80 可以分解成 $80 = 8 * 10^1$。最后可以输出整数数值 8。该数值就是数据“D”的编码。

2.2 解码

解码前进行的初始化必需同编码前进行的初始化完全一致，即：初始时符号的频度和区间的范围必需同编码时一致。解码时只输入一个数值 V，V 属于区间[L’,H’]。根据这个关系，以及公式 2 和公式 3 可以推导出解码时逆运算的公式。推导过程如下。

$\because$ L' $\leqslant$ V $\leqslant$ H'

$\therefore$ L + R div T * F_S $\leqslant$ V $\leqslant$ L + R div T * (F_S + f_S) – 1

=> L + R div T * F_S $\leqslant$ V $<$ L + R div T * (F_S + f_S)

=> R div T * F_S $\leqslant$ V - L $<$ R div T * (F(S) + f_S)

=> F_S $\leqslant$ (V - L) div (R div T) $<$ F_S + f_S

令 F'_S = (V - L) div (R div T) （公式 4）

可得 F_S $\leqslant$ F'_S $<$ F_S + f_S

根据输入数值 V、当前的区间[L,H]以及总计频度 T，利用公式 4 就可以计算出 F'_S。F'_S 是一个估算出来的累积频度（以下简称“估算频度（Estimate Frequency）”），属于区间 [F_S,F_S + f_S)。利用估算频度 F'_S 在表 2.1 中进行查找，当满足 F_S $\leqslant$ F'_S $<$ F_S + f_S 的条件时就可以解码出一个符号。在我们的例子里，输入数值是 8，乘以省略的 b^n 后为 80。输入数值 80 可以计算出估算频度 3（注意，这里进行的是整数运算，如果用实数运算答案是 3.2）。根据估算频度 3 可以解码出符号“D”。

2.3 更大的范围

前面的例子中只编码了仅有一个符号的数据。如果编码一个符号后的映射区间范围足够的大，那么可以利用这个区间继续编码下一个符号。当编码多个符号的数据时，为了有足够大的映射区间，必需加大初始时区间的范围。现在来看一个例子，假设编码具有两个符号的数据“DC”。初始时设 w = 3，即初始区间范围 R = 1000。映射区间的计算结果如表 2.2 所示。

符号	输入“D”，区间[000,999]				输入“C”，区间[750,999]			
	频度	累积频度	映射区间	区间范围	频度	累积频度	映射区间	区间范围
A	1	0	[000,249]	250	1	0	[750,799]	50
B	1	1	[250,499]	250	1	1	[800,849]	50
C	1	2	[500,749]	250	1	2	[850,899]	50
D	1	3	[750,999]	250	2	3	[900,999]	100

表 2.2 数据“DC”映射区间的计算

首先说明一点，在频度统计方面这里使用的是自适应方法。也就是说，初始时所有符号的频度都设为 1。输入一个符号，计算了该符号的映射区间后再将该符号的频度加 1。另外，在编码第 1 个符号时，使用的是初始区间。在编码下一个符号时使用的是上一个符号的映射区间。这样可以确保最终输出的数值 V 落在每一个编码符号的映射区间内。在当前的例子中，最后可以输出 850，省略后为 85。数值 85 就是数据“DC”的编码。

当然在编码的时候，没必要将每个符号的映射区间都计算出来。只需要计算当前输入符号的映射区间即可。现在再来看一个更多符号的例子“DCBDDDAADCB”。这份数据的长度为 11 个符号。我们使用一个庞大的初始区间对其进行压缩。初始时设 w=11，即：初始区间范围 R = 100000000000。编码时映射区间的变化如表 2.3 所示。

输入数据	输入符号	映射区间	区间范围
“”		[00000000000, 99999999999]	100000000000
“D”	D	[75000000000, 99999999999]	25000000000
“DC”	C	[85000000000, 89999999999]	5000000000
“DCB”	B	[85833333333, 86666666665]	833333333
“DCBD”	D	[86428571428, 86666666665]	238095238
“DCBDD”	D	[86577380948, 86666666659]	89285712
“DCBDDD”	D	[86626984118, 86666666653]	39682536
“DCBDDDA”	A	[86626984118, 86630952370]	3968253
“DCBDDDAA”	A	[86626984118, 86627705617]	721500
“DCBDDDAAD”	D	[86627404993, 86627705617]	300625
“DCBDDDAADC”	C	[86627520618, 86627566867]	46250
“DCBDDDAADCB”	B	[86627530527, 86627537132]	6606

表 2.3 数据“DCBDDDAADCB”的编码过程

通过表 2.3 中的数据可以发现，区间编码可以将一份数据“DCBDDDAADCB”映射到一个区间[86627530527,86627537132]中。使用该区间中的一个数值就可以表示整个数据。在这里可以使用数值 86627531000，省略后为 86627531。观察表 2.3 中的数据可以发现，所有经过编码的符号对应的映射区间中都包含数值 86627531000。这意味着只要使用该数值，就可以解码出所有的符号。

3 正规化

上述算法实际上是无法实现的。观察表 2.3 中的数据可以发现，每次编码一个符号后，映射区间就会缩减到一个更小的区间中。随着编码符号越来越多，映射区间的范围最后会趋向 0。如果增加初始区间的范围，那么初始区间的范围将趋向无穷大。

解决这一问题的方法是使用正规化（Renormalization，又称“归一化”）。正规化就是使用一个较小的区间，以模拟无限区间的运算。每当区间满足一定的条件时，就将一些数字从

区间中移出，并对区间进行一次扩展。这样做虽然会有精度上的损失，但是可以模拟出无限区间的运算。下面就来介绍一下正规化的具体实现。

首先区间的运算要限制在一个较小的区间内。我们令区间范围的最大值为 R_{max}，令 $R_{max} = b^w$。同时我们令区间范围的最小值为 R_{min}，则 $R_{min} = b^{w-1}$。在这里我们假设 w = 3，即 R_{max} = 1000，R_{min} = 100。假设有一个区间[1319314,1320105]，那么我们可以将这个区间中的数字分为 3 个部分。如下所示。

区间	[13 13	19 20	314, 105]
部分	1	2	3

区间的第 1 个部分，是两个完全一样的数字 13。在前面的算法描述中我们知道，每次编码一个符号都是从上一个符号的映射区间开始的。如果一个区间的第 1 部分完全一样，那么在后续的运算中这一部分将不会再改变。那么可以将这部分的数字移出区间并输出。我们称这部分数字为不变数字，用 c 表示。

区间的第 2 个部分，是两个仅相差 1 的数字 19 和 20。这两个数字可以用公式 $(d + 1) * b^{n-1} - 1$ 和 $(d + 1) * b^{n-1}$ 来表示。我们称 d 为延迟数字，称 n 为延迟长度。在这个例子里：d = 1，n = 2。一般的，当 n = 0 时 d = 0。当出现延迟数字时，即：$n \neq 0$，表明区间的上沿和下沿靠的很近。如果区间的上下沿靠的过近，即 R 偏小，会严重的影响运算精度。所以此时，也需要将延迟数字移出区间。但是，当出现延迟数字时，我们通常无法知道区间的变化是趋向 d 还是趋向 d + 1。所以延迟数字暂时不能输出。当区间的变化趋势可以确定时，再输出延迟数字。

区间的第 3 个部分，是两个长度为 w 的数字。这两个数字可以形成一个区间，使得区间范围 R 满足条件：$R_{min} < R \leqslant R_{max}$。在这个区间上可以进行下一轮的编码操作。

根据上面的描述我们可以发现，不变数字 c 和延迟数字 d 都是可以移出区间的。那么我们可以确定的一个移出数字 x，x 可以由下述公式进行计算。

$$x = c * b^n + (d + 1) * b^{n-1}$$

现在我们可以将区间[L,H]转换为 c,(d,n),[L”,H”]的形式。其中的数学关系如下所示。

$$L” = L - x * R_{max}$$
$$H” = H - x * R_{max}$$

这样我们可以根据 R_{max} 限制[L”,H”]区间的大小。区间[L”,H”]同时满足下列条件。

$$-R_{max} < L” < H” < R_{max}$$
$$R_{min} < H” - L” + 1 \leqslant R_{max}$$

根据上面的公式，我们可以将区间[1319314,1320105]转换为 13,(2,2),[-686,104]的形式。这样，我们就可以使用一个较小的区间[L”,H”]来模拟无限区间[L,H]上的运算。每当区间满足条件时，就将数字从区间中移出。现在我们仍就对数据“DCBDDDAADCB”进行压缩，不过这次将使用正规化。压缩前设 R_{max} = 1000，R_{min} = 100，初始区间范围 $R = R_{max}$。编码时映射区间的变化，以及正规化操作如表 3.1 所示。

输入数据	输入符号	映射区间	区间范围	正规化		区间范围
“”		[000, 999]	1000	,(0,0),	[000, 999]	1000
“D”	D	[750, 999]	250	,(0,0),	[750, 999]	250
“DC”	C	[850, 899]	50	8,(0,0),	[500, 999]	500
“DCB”	B	[583, 665]	83	8,(5,1),	[-170, 659]	830
“DCBD”	D	[420, 655]	236	8,(5,1),	[420, 655]	236
“DCBDD”	D	[565, 651]	87	86,(5,1),	[-350, 519]	870
“DCBDDD”	D	[130, 513]	384	86,(5,1),	[130, 513]	384
“DCBDDDA”	A	[130, 167]	38	8661,(0,0),	[300, 679]	380
“DCBDDDAA”	A	[300, 367]	68	86613,(0,0),	[000, 679]	680
“DCBDDDAAD”	D	[392, 671]	280	86613,(0,0),	[392, 671]	280
“DCBDDDAADC”	C	[497, 538]	42	86613,(4,1),	[-030, 389]	420
“DCBDDDAADCB”	B	[060, 119]	60	866135,(0,1),	[-400, 199]	600

表 3.1 数据“DCBDDDAADCB”的正规化编码过程

完成编码后可以输出数值 866135100，省略后为 8661351。观察表 3.1 中的数据可以发现 c,(d,n),[L”,H”]的形式是对[L,H]形式的有限模拟。也就是说模拟运算时会产生精度损失，不过这些损失还是在许可的范围内。运算产生的精度损失不会影响到编解码的结果。

正规化操作就是在区间满足一定的条件时，将数字从区间中移出，同时对区间进行扩展。观察表 3.1 中数据编码的过程，可以发现正规化操作有以下几种类型。

情况 1，“D”[750,999]。当编码完数据“D”时区间范围 R 的大小有 250。此时 $R > R_{min}$，所以不需要进行正规化操作。可以直接利用当前的区间，编码下一个符号。值得注意的是，编码完数据“DCBD”时也是这种情况。虽然此时有一个还未输出的延迟数字，但是只要 R 足够大就不进行正规化操作。

情况 2，“DC”[850,899]。当编码完数据“DC”时，此时区间的上沿和下沿的最高位数字相同。即：出现了不变数字。此时可以将不变数字移出区间并输出，同时将区间扩展。区间扩展的计算公式如下。

$$L'' = L'' * b \bmod R_{max}$$
$$H'' = (H'' * b + b-1) \bmod R_{max}$$

其中 mod 表示取余运算。在当前的例子里，区间[850,899]输出数字 8 后，对区间进行扩展可以得到区间[500,999]。正规化后区间范围由 50 扩展到 500。可以发现扩展时区间范围满足公式：R = R * b。

情况 3，“DCB”[583,665]。当编码完数据“DCB”时，虽然此时区间的上下沿最高位数字不相同，但是此时区间范围 $R \leqslant R_{min}$。为了防止 R 趋向 0，此时必需进行正规化操作。即：出现了延迟数字。在当前的例子里，区间的上下沿最高位数字分别是 5 和 6。区间后期的变化可能使最高位数字趋向 5，也有可能趋向 6。根据区间上下沿之间的关系，我们可以断定，当出现延迟数字时区间的上下沿最高位数字只相差 1。此时我们可以将 5 记录下来延迟输出。等到区间变化的趋势确定后再输出。此时 d = 5，n = 1。另外，此时同样也要对区间进行扩展。区间扩展的计算公式如下。

$$L'' = (L'' - R_{max}) * b \bmod R_{max}$$
$$H'' = (H'' * b + b-1) \bmod R_{max}$$

在当前例子里，将区间[583,665]的下沿最高位数字 5 记录下来，然后对区间进行扩展可以得到区间[-170,659]。扩展后原区间的最高位数字相当于被抛弃了。区间下沿 L”也由 $0 \leqslant L'' < R_{max}$ 调整到$- R_{max} < L'' \leqslant 0$。但是，可以发现扩展时区间范围同样满足公式：R = R * b。

情况 4，“DCBDD”[565,651]。在编码完数据“DCBDD”时，此时的区间需要进行正规化操作，同时还有一个未输出的延迟数字需要处理。对于当前有未输出的延迟数字的情况，根据区间的位置又可以分为 3 种类型。

情况 4.1，$H'' \leqslant 0$。此时区间位于 0 的下方，可以确定延迟数字趋向下沿。此时可以先输出 d，然后输出 n - 1 个数字 b - 1。

情况 4.2，$L'' \geqslant 0$。此时区间位于 0 的上方，可以确定延迟数字趋向上沿。此时可以先输出 d + 1，然后输出 n - 1 个数字 0。

情况 4.3，$H'' > 0$ 同时 $L'' < 0$。此时区间的上沿位于 0 的上方，区间的下沿位于 0 的下方。此时无法判断延迟数字的变化趋势，即：趋向未定。那么可以继续保留延迟数字。同时将 n 加 1，表示延迟长度的增加。然后按照情况 2 那样扩展区间。只不过最高位数字不输出，而是直接抛弃。

以上 3 种情况中，只有情况 4.3 扩展了区间。所以情况 4.1 和情况 4.2 在输出延迟数字后，还需要对区间进行判断以便扩展区间。在当前的例子中，根据此时的区间已经可以确定延迟数字趋向上沿，即：输出数字 6。输出数字 6 后，可以发现此时的区间属于情况 3。即：输出了一个旧的延迟数字后，又出现了一个新的的延迟数字。按照情况 3 的处理方式操作，最后区间正规化为[-350,519]。

综上所述，正规化的操作只有在 $R \leqslant R_{min}$ 时进行。每编码完一个符号，都要检查是否需要进行正规化操作。正规化的时候要先检查是否有延迟数字，然后再检查是否有不变数字。从上面的例子中可以发现，不同的情况可能一同出现。所以正规化的操作要不断的进行直到满足条件 $R > R_{min}$。满足条件后才完成了区间调整的全部工作。

另外，在解码的时候也要进行正规化操作。而且解码时正规化操作要与编码时相一致。唯一不同的是，扩展区间时需要输入数字以填补移出的空间。

4 实现

4.1 编码

从前面的算法描述中可以发现，当出现延迟数字时区间的上下沿可能会出现负数。在实现的时候为了简化运算，防止上下沿出现负数可以将区间向上平移 R_{max} 位置。平移的公式如下。

$$L" = L" + R_{max}$$
$$H" = H" + R_{max}$$

平移后的区间[L",H"]将满足条件 $0 < L" < H" < 2R_{max}$。实现时，将 R_{max} 设为 L"所能表示的最大的 2 的次方数。为了能正常的编解码，在将数字从区间中移出或者移入时、以及判断不变数字时，需要忽略 L"和 H"的最高位。相当于将区间平移回
$-R_{max} < L" < H" < R_{max}$ 的位置。

在前面的例子中将基数 b 设为 10，在实现的时候可以将 b 设为 256。这样每次正规化输出数字的时候，正好是以字节为单位。每次输出一个字节。另外，平移后的区间总是大于等于 0，所以可以用 32 位无符号整形变量来保存 L"和 H"。在 Delphi 中可以用 Cardinal 变量类型。在实现代码之前我们需要定义一些常量和变量，如下所示。

```
01  const
02      Rmax = Cardinal(1 shl 31);
03      Rmin = Cardinal(1 shl 23);
04  var
05      d, n, L, H, R, V : Cardinal;
```

代码 4.1 常量与变量的声明

代码 4.1 中用变量 L 表示 L"，用变量 H 表示 H"。由于使用 Cardinal 类型，区间的最大范围 R_{max} 设定为$80000000。至此，我们已经可以写出区间编码的编码程序。程序如下所示。

```
01  procedure RangeCoder_Encode(cf, f, T : Cardinal);
02  var
03      i : Integer;
04  begin
05      //区间计算
06      R := R div T;
07      Inc(L, cf * R);
08      R := f * R;
09      //调整区间
10      while R <= Rmin do
11      begin
12          H := L + R - 1;
```

```
13          //判断是否有延迟数字
14          if n <> 0 then
15          begin
16              if H <= Rmax then
17              begin
18                  //趋向下沿
19                  OutputCode(d);
20                  for i := 1 to n - 1 do OutputCode($FF);
21                  n := 0;
22                  Inc(L, Rmax);
23              end
24              else if L >= Rmax then
25              begin
26                  //趋向上沿
27                  OutputCode(d + 1);
28                  for i := 1 to n - 1 do OutputCode($00);
29                  n := 0;
30              end
31              else
32              begin
33                  //趋向未定
34                  Inc(n);
35                  //扩展区间
36                  L := (L shl 8) and (Rmax - 1);
37                  R := R shl 8;
38                  continue;
39              end;
40          end;
41          //判断最高位数字
42          if ((L xor H) and ($FF shl 23)) = 0 then
43          begin
44              //出现不变数字
45              OutputCode(L shr 23);
46          end
47          else
48          begin
49              //出现延迟数字
50              Dec(L, Rmax);
51              d := L shr 23;
52              n := 1;
53          end;
54          //扩展区间
55          L := ((L shl 8) and (Rmax - 1)) or (L and Rmax);
56          R := R shl 8;
```

```
57          end;
58          //完成
59      end;
```

代码 4.2 编码程序

代码 4.2 中用变量 cf 表示 F_S，用变量 f 表示 f_S。其它变量名与本文中设定的相一致。OutputCode 是一个输出函数，用以输出一个字节的数据。代码 4.2 中的函数完成了输入一个符号时所要进行的编码工作。这里需要说明的是，在实现时我特意将编码部分与模型部分相分离。当区间编码编码一个符号时，只需要知道这个符号的累积频度 F_S、频度 f_S、总计频度 T。至于这些数据是如何被统计出来的，编码部分可以不用关心。这些工作将由模型部分来完成。这样分离的好处是，我可以使用不同的模型来进行压缩。甚至可以使用高阶的复杂模型，例如 PPM。所以代码 4.2 中的函数只接收 3 个参数。

观察代码 4.2 中的代码可以发现，编码时区间计算的代码只占用了很小的一部分，从第 6 行到第 8 行。更多的代码用以正规化操作，从第 10 行到第 57 行。正规化操作的代码主要以情况判断为主。判断当前区间属于哪种情况，然后进行相应的处理。处理的方法对应本文第 3 节的内容，这里不再重复。

所有数据都编码完成后，需要输出最后区间[L",H"]中的一个数值。从理论上来说，这个数值可以省略部分的数字。但是在实现的时候为了简化运算和对齐数据，一般不进行省略。而是将最后区间的下沿 L"完整输出。注意，输出 L"时同样需要忽略最高位。另外，如果此时有一个未输出的延迟数字。那么要先输出延迟数字再输出 L"。延迟数字的趋势可以用 L"来判断，如果 L" ＜ R_{max} 那么延迟数字趋向下沿，反之趋向上沿。收尾操作的代码如下所示。

```
01  procedure RangeCoder_FinishEncode;
02  var
03      i : Integer;
04  begin
05      //输出剩余延迟数字
06      if n <> 0 then
07      begin
08          if L < Rmax then
09          begin
10              //趋向下沿
11              OutputCode(d);
12              for i := 1 to n - 1 do OutputCode($FF);
13          end
14          else
15          begin
16              //趋向上沿
17              OutputCode(d + 1);
18              for i := 1 to n - 1 do OutputCode($00);
19          end;
20      end;
```

```
21          //输出剩余编码
22          L := L shl 1;
23          i := 32;
24          repeat
25              Dec(i, 8);
26              OutputCode(L shr i);
27          until i <= 0;
28          //完成
29      end;
```

代码 4.3 编码的收尾操作

4.2 解码

解码时不需要处理不变数字和延迟数字，所以不需要对区间进行平移。但是为了和编码时的操作相一致，仍然需要忽略 L”和 H”的最高位。对于输入数值 V 也一样，忽略最高位只有低位参与运算。由于编码时一次输出一个字节，输入数据到 V 中的时候也可以按字节读入。到要运算的时候再将 V 右移 1 位。虽然不需要对区间进行平移，但是 V 有时会小于 L”。因为编码时 L”属于 $0 < L" < H" < 2R_{max}$。所以此时需要对 V 进行修正，即加上 R_{max}。相当于将 V 临时向上平移 R_{max}。

另外，由于将编码部分与模型部分相分离。在解码的时候需要两个函数来完成。第一个函数根据当前的区间[L”,H”]和总计频度 T 计算出估算频度。这个估算频度交给模型部分解码出符号。根据解码出的符号可以统计出累积频度 F_S 和频度 f_S。第二个函数根据累积频度 F_S 和频度 f_S 计算映射区间。这样才能保证区间的变化与编码时相一致。第一个函数的代码如下。

```
01  function RangeCoder_DecodeTarget(T : Cardinal): Cardinal;
02  var
03      Code : Cardinal;
04  begin
05      Code := V shr 1;
06      //计算估算频度
07      R := R div T;
08      if Code < L then Result := (Code + Rmax - L) div R
09      else             Result := (Code - L) div R;
10      //完成
11  end;
```

代码 4.4 解码程序 1

代码 4.4 中的函数返回估算频度，这个估算频度交给模型部分解码。另外，代码 4.4 第 7 行计算得到的 R 并不是区间范围，而是一个临时的值。这个值将在第二个函数中用到，这里保存这个值可以防止重复运算。第二个函数的代码如下。

```
01  procedure RangeCoder_Decode(cf, f : Cardinal);
02  begin
03      //区间计算
04      Inc(L, cf * R);
05      R := f * R;
06      //调整区间
07      while R <= Rmin do
08      begin
09          L := (L shl 8) and (Rmax - 1);
10          R := R shl 8;
11          V := (V shl 8) or InputCode;
12      end;
13      //完成
14  end;
```

代码 4.5 解码程序 2

代码 4.5 中的函数主要对区间进行重新计算和调整，同时还会输入数据。InputCode 是一个输入函数，每次调用时会返回一个字节的数据。观察代码 4.5 中的代码可以发现，虽然解码时也要进行正规化操作。但是由于不需要处理不变数字和延迟数字，正规化操作的代码变的相当简单。每次左移时高位的数字就被抛弃了。对于 V 来说左移后需要输入一个字节的数据填补到低位。

自此，编码部分的实现已经全部讲解完毕。可以发现，虽然区间编码的理论基础非常复杂，但是实现代码却相当简单。这可以算是区间编码的优点。

5 频度统计

5.1 理论

在上一节中主要讲解了编码部分的实现，现在来讲解一下模型部分的实现。模型部分的主要工作就是对符号的频度进行统计和更新。除此之外还有一点需要注意。从前面描述的算法中可以发现，总计频度 T 在区间计算中总是以分母的形式出现。随着编码符号越来越多，T 也会越来越多大。如果 T 过大会使运算精度下降，从而降低压缩率。所以需要对 T 设定一个最大值（以下简称“最大频度（Max Frequency）”）的限制。当总计频度 T 达到最大频度时，要对符号频度进行削减。通常的做法是对所有符号的频度减半。另一方面，最大频度也不能太小。太小的最大频度会使频度削减操作频繁的进行。从而造成符号频度差别的丢失，这同样会降低压缩率。在实现的时候，我们进行的是 32 位无符号整数运算，此时将最大频度设定为$FFFF 是比较合适的。

模型部分实现时最主要的问题就是使用何种数据结构来保存频度数据。在压缩算法实现的过程中，模型的实现在算法耗时中占有很大的一部分比重。所以说，一个优秀的数据结构非常重要。一种比较简单的结构就是使用一个符号容量大小的数组，每个数组元素保存对应符号的频度。这种结构在符号容量比较小的情况下是可以使用的。但对于较大的符号容量这

种结构将会非常的慢。

Peter M. Fenwick 在 1994 年提出了一个名为二进制索引树（Binary Indexed Tree，以下简称“BIT 结构”）的结构[2]。BIT 结构的基本思想就是将累积频度保存在一棵树上。树中每个节点按一定规则保存若干个连续符号的频度之和。通过对树中节点的访问，完成频度与累积频度的统计操作。目前，在统计累计频度的数据结构中，BIT 结构是速度最快的。同时它的内存占用也是最低的。本文将介绍 BIT 结构的实现方式。

在 BIT 结构中使用两棵树来完成相关的操作。这两棵树是询问树（Interrogation Tree）和更新树（Updating Tree）。询问树用来统计频度和累积频度，更新树用来更新频度。这两棵树都可以隐含在同一个数组中。只要使用一个符号容量大小的数组就可以表示这两棵树。现在，先来看一个例子。假设有一个具有 16 个符号的符号集 S。其中的符号依次为 $S_0, S_1, \ldots, S_{15}$，对应的符号频度为 $f_0, f_1, \ldots, f_{15}$。本文中使用 $f_a \ldots f_b$ (a ＜ b)来表示连续符号的频度之和。例如：$f_9 \ldots f_{12} = f_9 + f_{10} + f_{11} + f_{12}$。另外，我们令符号 S_{16} 为结束控制符 EOM（End of Massage）。设保存数据的数组为 Tree。那么我们可以得到 BIT 结构中询问树的数据结构如图 5.1 所示。

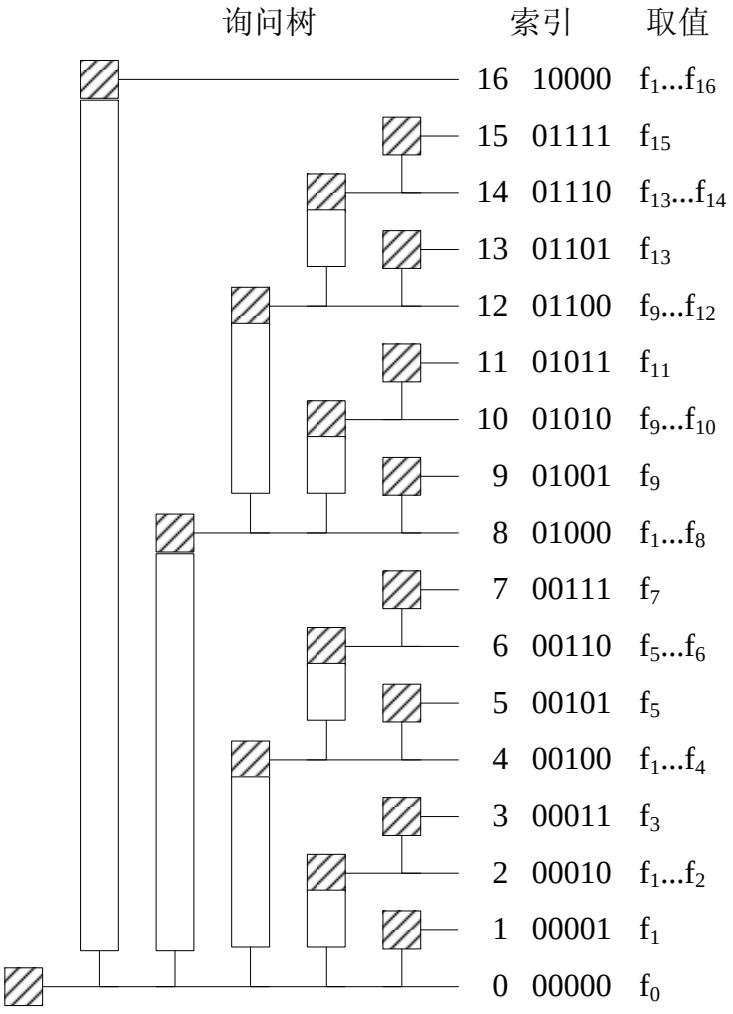

图 5.1 询问树的数据结构

图 5.1 中的阴影矩形表示树的节点，空白矩形表示节点的取值范围。索引表示树节点在数组 Tree 中的保存位置。将图 5.1 中的询问树画成一般的形式如图 5.2 所示。

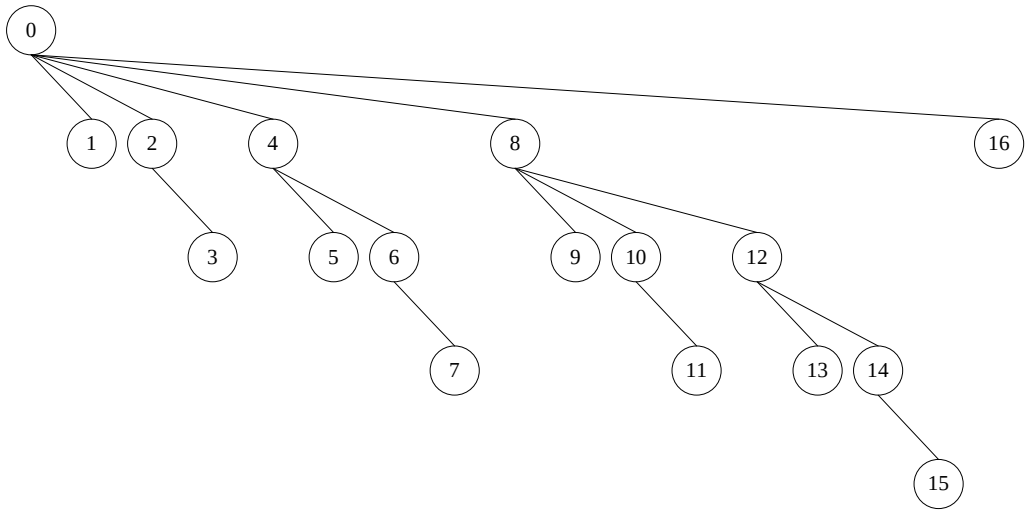

图 5.2 询问树

观察图 5.1 中的数据结构可以发现，树节点的索引与符号值相对应。而且，树节点的取值范围和树节点相互之间的关系都与树节点的索引有着密切的关系。树节点的取值范围就是对应的父节点索引向上到达该节点的所有索引对应的频度之和。例如，Tree[8]的父节点是 Tree[0]，那么 Tree[8]的取值就是 $f_1 \ldots f_8$。从父节点索引向上的 1 到 8。

另外，树节点相互之间的关系与索引值中二进制最低位的 1（以下简称“低位 1”）有关系。对于一个树节点，将其索引值的低位 1 减去，得到的就是该节点的父节点的索引。例如树节点 Tree[12]，其索引为 12 (01100)。减去低位 1 后为 8 (01000)，Tree[8]就是 Tree[12]的父节点。同样，8 (01000)减去低位 1 后为 0 (00000)。Tree[0]就是 Tree[8]的父节点。对于唯一不存在低位 1 的索引 0，是询问树的根节点。利用询问树统计频度和累积频度的过程，其实就是不断查询指定节点的父节点的过程。

与询问树相对应的是更新树。更新树的结构相当于是询问树的镜像。更新树也可以隐含在数组 Tree 中。更新树的数据结构如图 5.3 所示。将图 5.3 中的更新树画成一般的形式如图 5.4 所示。

观察图 5.3 中的数据结构可以发现，对于更新树来说，树节点相互之间的关系同样与索引值中的低位 1 有关系。对于更新树上的一个树节点，将其索引值加上对应的低位 1，得到的就是该节点的父节点的索引。例如树节点 Tree[10]，其索引为 10 (01010)。加上低位 1 后为 12 (01100)，Tree[12]就是 Tree[10]的父节点。同样，12 (01100)加上低位 1 后为 16 (10000)。Tree[16]就是 Tree[12]的父节点。另外，可以发现更新树上父节点的取值范围总是包含子节点的取值范围。例如，Tree[12]的取值范围 $f_9 \ldots f_{12}$ 包含 Tree[10]的取值范围 $f_9 \ldots f_{10}$。所以，利用更新树更新频度的过程，其实就是不断查询指定节点的父节点的过程。这一点与询问树的使用相似。

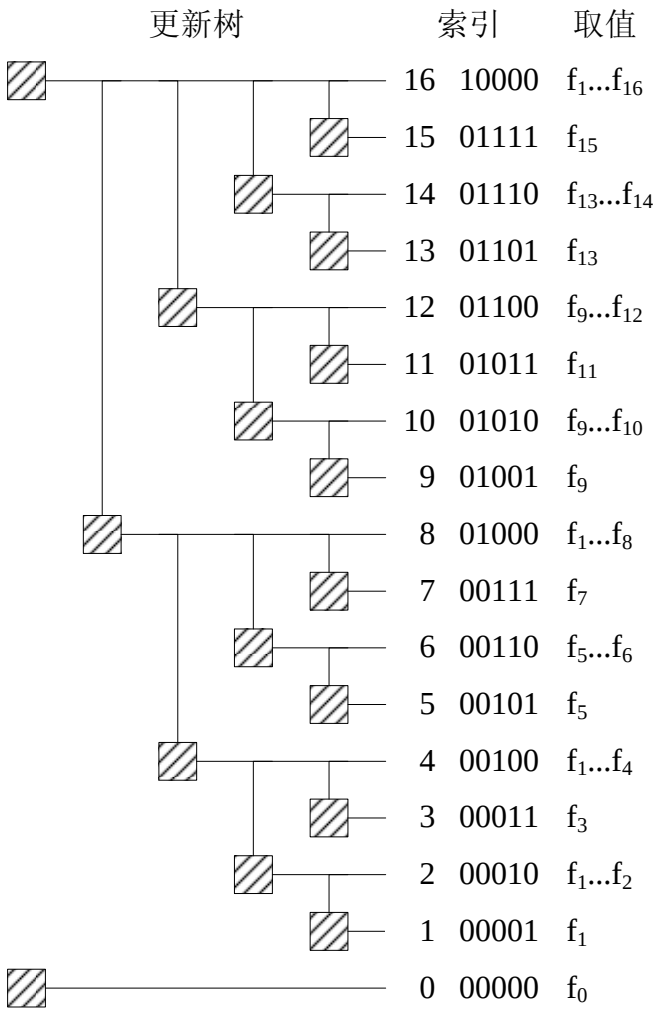

图 5.3 更新树的数据结构

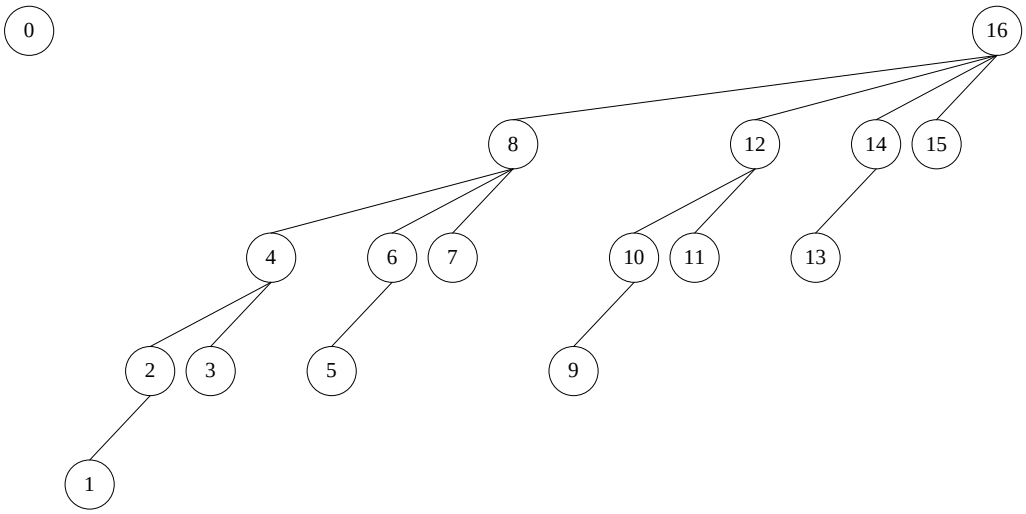

图 5.4 更新树

综上所述，询问树和更新树是隐含在数组 Tree 中的。按询问树结构访问节点时，将指定节点的索引值减去对应的低位 1 可以得到父节点的索引值。按更新树结构访问节点时，将指定节点的索引值加上对应的低位 1 可以得到父节点的索引值。下面的章节将详细的分析 BIT 结构的具体实现。

另外注意一点，符号值和索引值是一一对应的。访问时可以将符号值作为索引，访问 Tree 中的节点。

5.2 统计累积频度

实现前先说说低位 1 的计算。由于现在计算机系统都是使用补码来表示负数，而数组索引值都是正数。假设保存索引值的变量为 Id，那么使用代码 Id and –Id 就可以取得 Id 的低位 1。减去或加上低位 1 的代码如下。

```
01    Dec(Id, Id and -Id);
02    Inc(Id, Id and -Id);
```

代码 5.1 减去或加上低位 1

另外，在实现代码之前我们需要定义一些常量和变量，如下所示。

```
01  const
02        SYMBOL_NO = 256; //符号容量
03        EOM = SYMBOL_NO; //结束符
04        MAX_FREQ = $FFFF; //最大频度
05  var
06        T : Cardinal; //总计频度
07        Tree : array [0..SYMBOL_NO] of Word;
```

代码 5.2 常量与变量的声明

统计累积频度就是对于一个指定的符号，统计出符号值小于它的其它符号的频度总计。统计累积频度是在询问树上完成的，按照前面提到的询问树的特点可以得到统计累积频度的方法。统计累积频度时，从指定索引的下一个节点开始。沿着父节点向上到达根节点，将路过节点的值相加就可以统计出累积频度。

例如，要统计符号 S_8 的累积频度。从 Tree[8]的下一个节点 Tree[7]开始，Tree[7]的父节点是 Tree[6]，Tree[6]的父节点是 Tree[4]，等等。最后可以得到 S_8 的累积频度 $F_8 = f_7 + f_5 \ldots f_6 + f_1 \ldots f_4 + f_0$。实现的代码如下所示。

```
01  function Model_GetCumFreq(Sym : Integer): Cardinal;
02  begin
03      //排除符号 0
04      if Sym = 0 then
05      begin
06          Result := 0; exit;
07      end;
08      //统计累积频度
09      Result := Tree[0];
10      Dec(Sym);
11      while Sym > 0 do
12      begin
13          Inc(Result, Tree[Sym]);
14          Dec(Sym, Sym and -Sym);
15      end;
16      //完成
17  end;
```

代码 5.3 统计累积频度

注意，观察图 5.1 中的取值范围可以发现，符号 S_0 被独立了出来。对于所有索引大于 0 的节点，其取值范围都没有包含 f_0。而且，对于符号值最小的符号，S_0 的累积频度始终为 0。所以在实现的时候，通常将符号 S_0 独立出来处理。先排除符号值为 0 的情况，再处理其它符号值的情况。

5.3 统计频度

统计频度也是在询问树上完成的。统计频度的过程同统计累积频度类似。统计频度时，同样从指定索引的下一个节点开始。沿着父节点向上到达指定索引的父节点，将路过节点的值从指定索引节点的值中减去，就可以统计出频度。注意，相减的节点中不包含指定索引的父节点

例如，要统计符号 S_8 的累积频度。从 Tree[8]的下一个节点 Tree[7]开始，Tree[7]的父节点是 Tree[6]，Tree[6]的父节点是 Tree[4]。Tree[4]的父节点是 Tree[0]，这同时也是 Tree[8]的父节点。不包含 Tree[0]节点，将其它节点的值从 Tree[8]中减去。最后可以得到 S_8 的频度 $f_8 = f_1 \ldots f_8 - f_7 - f_5 \ldots f_6 - f_1 \ldots f_4$。实现的代码如下所示。

```
01  function Model_GetFreq(Sym : Integer): Cardinal;
02  var
03      Parent : Integer;
04  begin
05      Result := Tree[Sym];
06      //排除符号 0
07      if Sym = 0 then exit;
```

```
08          //统计频度
09          Parent := Sym - (Sym and -Sym);
10          Dec(Sym);
11          while Sym <> Parent do
12          begin
13              Dec(Result, Tree[Sym]);
14              Dec(Sym, Sym and -Sym);
15          end;
16          //完成
17      end;
```

代码 5.4 统计频度

先回顾一下 4.1 节和 4.2 节。在进行编码和解码的时候，累积频度 F_S 和频度 f_S 总是同时使用。注意代码 4.2 和代码 4.5，在调整区间的时候总是同时使用 F_S 和 f_S。现在再来观察代码 5.3 和代码 5.4。可以发现，在统计累积频度和统计频度时，部分计算是相同的。既然 F_S 和 f_S 需要同时使用。那么可以将统计累积频度和统计频度的代码合并到同一个函数中，同时计算 F_S 和 f_S。这样可以减少算法步骤，减少代码冗余。合并后的代码如下所示。

```
01  procedure Model_GetFreq(Sym : Integer; var cf, f : Cardinal);
02  var
03      Id, Parent : Integer;
04      Sum : Cardinal;
05  begin
06      //排除符号 0
07      if Sym = 0 then
08      begin
09          cf := 0;
10          f := Tree[0];
11          exit;
12      end;
13      //统计累积频度和频度
14      Id := Sym;
15      Parent := Id - (Id and -Id);
16      Dec(Id);
17      Sum := 0;
18      while Id <> Parent do
19      begin
20          Inc(Sum, Tree[Id]);
21          Dec(Id, Id and -Id);
22      end;
23      f := Tree[Sym] - Sum;
24      while Id > 0 do
25      begin
```

```
26              Inc(Sum, Tree[Id]);
27              Dec(Id, Id and -Id);
28          end;
29          cf := Tree[0] + Sum;
30          //完成
31      end;
```

代码 5.5 统计累积频度和频度

5.4 更新频度

当编码完成一个符号，需要对该符号的频度加 1。更新频度是在更新树上完成的。前面讲过，在更新树上父节点的取值范围总是包含子节点的取值范围。所以更新频度时，从指定索引的节点开始。沿着父节点向上到达根节点，将路过节点的值加 1，就可以完成更新频度的操作。

例如，更新符号 S_9。从 Tree[9]开始沿着父节点向上，分别是 Tree[10] 、Tree[12]、Tree[16]。将这些节点的值加 1，就可以完成更新。实现的代码如下所示。

```
01  procedure Model_IncFreq(Sym : Integer);
02  begin
03      //更新频度
04      if Sym = 0 then Inc(Tree[0])
05      else
06      begin
07          repeat
08              Inc(Tree[Sym]);
09              Inc(Sym, Sym and -Sym);
10          until Sym > SYMBOL_NO;
11      end;
12      Inc(T);
13      //完成
14  end;
```

代码 5.6 更新频度

另外注意一点，在更新频度之前通常需要检查总计频度 T。如果总计频度 T 达到最大频度，需要对所有符号的频度进行削减操作。

5.5 削减频度

削减频度的操作由两部分组成，统计频度和更新频度。操作时对所有的符号进行遍历。首先统计出符号的频度。然后计算出削减后的频度。最后根据削减量进行更新。另外，计算

削减后频度时，使用代码(f + 1) shr 1。这样在对频度减半的过程中可以防止出现 0 频的情况。实现的代码如下所示。

```
01  procedure Model_DecFreq;
02  var
03      i, Id, Parent : Integer;
04      Freq, NewFreq : Cardinal;
05  begin
06      //削减频度
07      Tree[0] := (Tree[0] + 1) shr 1;
08      T := Tree[0];
09      for i := 1 to SYMBOL_NO do
10      begin
11          //统计频度
12          Id := i;
13          Freq := Tree[Id];
14          Parent := Id - (Id and -Id);
15          Dec(Id);
16          while Id <> Parent do
17          begin
18              Dec(Freq, Tree[Id]);
19              Dec(Id, Id and -Id);
20          end;
21          //计算差值
22          NewFreq := (Freq + 1) shr 1;
23          Inc(T, NewFreq);
24          Dec(Freq, NewFreq);
25          if Freq = 0 then continue;
26          //更新频度
27          Id := i;
28          repeat
29              Dec(Tree[Id], Freq);
30              Inc(Id, Id and -Id);
31          until Id > SYMBOL_NO;
32      end;
33      //完成
34  end;
```

代码 5.7 削减频度

在代码 5.7 中第 12 行到第 20 行是统计频度的代码，相当于前面介绍的代码 5.4。在代码 5.7 中第 27 行到第 31 行是更新频度的代码，相当于前面介绍的代码 5.6。

5.6 查找估算频度

在解码的时候，解码器会计算出一个估算频度 F'_S。利用估算频度 F'_S 进行查找，当满足 $F_S \leqslant F'_S < F_S + f_S$ 的条件时可以解码出一个符号。如果用一个数组保存符号的累积频度。将累积频度根据符号值放入数组对应的单元中，就象表 2.1 的第 3 列一样。那么这个数组就是一个有序数组。在有序数组上进行查找，最高效的算法是折半查找法。保存 BIT 结构的数组 Tree 当然不是有序数组，但是却具有相同的性质。所以在数组 Tree 上也可以进行折半查找法。

观察图 5.1 可以发现，此时数组的中点是 8。除 0 外，索引值小于等于 8 的节点取值范围都在 $f_1 \ldots f_8$ 之内。除 16 外，索引值大于 8 的节点取值范围都在 $f_9 \ldots f_{15}$ 之内。查找时可以先将要查找的值减去 Tree[0]。此时，如果该值小于等于 Tree[8]，那么从 1 至 7 的范围内重新开始查找。如果该值大于 Tree[8]，那么将该值减去 Tree[8]然后从 9 至 15 的范围内重新开始查找。当查找的范围只剩下一个节点时，查找结束。这样就在数组 Tree 上实现了折半查找法。

在实现的时候为了方便运算，可以设置一个掩码和一个基数索引。当前需要比较的数组索引等于掩码加上基数索引。初始时掩码为符号容量的一半，基数索引为 0。每完成一轮查找，掩码减半，根据比较结果调整基数索引。掩码为 0 时查找结束，基数索引加 1 就是对应的符号值。实现的代码如下所示。

```
01  function Model_FindEstFreq(ef : Cardinal): Integer;
02  var
03      Mask, Id : Integer;
04  begin
05      //排除符号 0 和结束符
06      if ef < Tree[0] then
07      begin
08          Result := 0; exit;
09      end;
10      Dec(ef, Tree[0]);
11      if ef >= Tree[SYMBOL_NO] then
12      begin
13          Result := EOM; exit;
14      end;
15      //查找
16      Result := 0;
17      Mask := SYMBOL_NO shr 1;
18      while Mask <> 0 do
19      begin
20          Id := Result + Mask;
21          if ef >= Tree[Id] then
22          begin
23              Result := Id;
24              Dec(ef, Tree[Id]);
```

```
25                  end;
26                  Mask := Mask shr 1;
27              end;
28              Inc(Result);
29              //完成
30      end;
```

代码 5.8 查找估算频度

代码 5.8 中用 ef 表示 F’s。另外需要注意，代码 5.8 中的函数若要能正常运行必需满足两个条件。第一，所有符号的频度都不能为 0。如果有频度为 0 的符号，会使查找出现错误。第二，符号容量必需是 2 的次方数。如果不是，查找时可能会出现数组访问越界。在编写实现代码的时候，要满足上述两点并不困难。前面的代码都是按照这样的要求编写的。

6 区间编码与算术编码

区间编码将所有数据映射到一个整数区间中，并输出一个很大的整数来表示数据。算术编码将所有数据映射到一个位于[0,1)的实数区间中，并输出一个很长的小数来表示数据。区间编码和算术编码在本质上是一致的。只不过是对同一事物稍微不同的两种理解方法。

1987 年 Ian H. Witten、Radford M. Neal 和 John G. Cleary 发表了一篇论文[3]，提出了一种基于整数运算的算术编码实现算法（以下简称“WNC 算法”）。由于算术编码输出的总是一个小于 1 的小数。WNC 算法省略了小数开始的 0 和小数点，只输出小数的尾数部分。这相当于输出了一个很大的整数。另外，在实现的时候 WNC 算法也是在一个整数区间中模拟实数区间的运算。从实现方式上看 WNC 算法和区间编码非常的类似。

目前，大多数算术编码的实现方式都是以 WNC 算法为基础的。许多研究人员将它与区间编码混为一谈。区间编码和算术编码还是有区别的，主要集中在正规化的操作上。

对于区间编码来说，实际应用中的实现方式一般以 Martin 论文[1]中描述的方法为基础。其特点是每次正规化时，以字节（8 位）为单位。每次正规化一个字节。

对于算术编码来说，实际应用中的实现方式一般以 Witten 等人论文[3]中描述的方法为基础。其特点是每次正规化时，以位为单位。每次正规化一个位。

由于区间编码每次正规化一个字节，而不是一个位。所以区间编码的运算精度比算术编码低。这会造成压缩率的稍微降低。但是正因为一次处理一个字节，区间编码的速度比算术编码要快很多。

另外，根据 Martin 论文[1]的发表年代，通常认为区间编码算法不受与算术编码算法相关的专利约束。正是基于这一点，才激起了研究人员尤其是开放源码社区对于区间编码的兴趣。而且，区间编码与算术编码之间只有细微的区别，二者本质上是一致的。所以许多应用在算术编码上的技术也可以应用在区间编码上，例如 PPM。

7 测试与分析

本文的配套项目在 Delphi 7.0 下开发，编译通过并调试正常。测试程序的计算机使用 Intel 2.66GHz 单核 CPU，DDR400 512MB 内存，Windows XP SP3 操作系统。测试程序压缩文件的时候，将文件全部读入内存进行压缩。压缩后的数据再写入到文件中。解压缩的过程也是

如此。算法计时的时候，读写文件的时间并没有计算在内。只计算单纯算法的耗时。另外，程序中使用 API 函数 GetTickCount 进行计时。按照微软官方的说法，该函数有 15 毫秒的误差。

为了进行比较测试，在本文的配套项目中我编写一个算术编码的实现程序。该实现程序以 WNC 算法为基础，并做了一些修改。我修改了 WNC 算法的区间计算方法，使得修改后的算法支持更大的最大频度。原先最大频度只能为$3FFF，现在设定为$FFFF。另外我将编码部份与模型部分分离。原先 WNC 算法的模型采用 move-to-front 的算法实现，现在修改成本文论述的 BIT 结构的实现方式。

除了算术编码，测试时还使用了哈夫曼编码进行比较测试。该哈夫曼编码的实现程序来自我写的另一篇论文[4]。注意一点，该哈夫曼编码的实现程序使用的是静态模型，而进行比较测试的区间编码程序和算术编码程序使用的都是自适应模型。

在基于统计模型的无损压缩算法领域，哈夫曼编码算法、区间编码算法和算术编码算法是最为常用的三种算法。现在就使用卡尔加里语料库（The Calgary Corpus[5]）中的 18 个文件，以及大型语料库（The Large Corpus[5]）中的 3 个文件，对这三个算法进行压缩率测试。测试结果如表 7.1 和表 7.2 所示。

文件	输入大小	哈夫曼编码		区间编码		算术编码	
		输出大小	bpc	输出大小	bpc	输出大小	bpc
bib	111261	72824	5.236	72626	5.222	72616	5.221
book1	768771	438444	4.563	435641	4.533	435572	4.533
book2	610856	368364	4.824	365309	4.784	365256	4.784
geo	102400	72648	5.676	72448	5.660	72440	5.659
news	377109	246456	5.228	244719	5.191	244684	5.191
obj1	21504	16156	6.010	16126	5.999	16124	5.999
obj2	246814	194212	6.295	191693	6.213	191672	6.213
paper1	53161	33400	5.026	33359	5.020	33356	5.020
paper2	82199	47684	4.641	47521	4.625	47512	4.624
paper3	46526	27332	4.700	27379	4.708	27376	4.707
paper4	13286	7920	4.769	8003	4.819	8000	4.817
paper5	11954	7492	5.014	7564	5.062	7564	5.062
paper6	38105	24088	5.057	24094	5.058	24092	5.058
pic	513216	106676	1.663	76744	1.196	76700	1.196
progc	39611	25972	5.245	25974	5.246	25972	5.245
progl	71646	43044	4.806	42979	4.799	42972	4.798
progp	49379	30280	4.906	30298	4.909	30296	4.908
trans	93695	65288	5.575	64920	5.543	64912	5.542
总计	3251493	1828280	4.498	1787397	4.398	1787116	4.397

表 7.1 卡尔加里语料库压缩率测试

文件	输入大小	哈夫曼编码		区间编码		算术编码	
		输出大小	bpc	输出大小	bpc	输出大小	bpc
bible.txt	4047392	2218508	4.385	2195047	4.339	2194680	4.338
E.coli	4638690	1159688	2.000	1164283	2.008	1163864	2.007
world192.txt	2473400	1558664	5.041	1543039	4.991	1542816	4.990
总计	11159482	4936860	3.539	4902369	3.514	4901360	3.514

表 7.2 大型语料库压缩率测试

注意，表 7.1 和表 7.2 中的输入大小与输出大小一列是以字节为单位。观察表 7.1 和表 7.2 中的测试数据可以发现，虽然很多资料声称区间编码可以获得比哈夫曼编码大的多的压缩率。但实际测试表明，就总体而言，区间编码只是提高了少量的压缩率。部分文件如“paper4”等，哈夫曼编码的压缩率还更高。但是也有部分文件如“pic”，区间编码获得比哈夫曼编码高得多的压缩率。另外，区间编码由于运算精度上的问题，其压缩率会比算术编码低。但是这种压缩率的降低很微小。比较测试结果，从 bpc 上看压缩率的降低幅度仅仅位于 bpc 小数点后的第 3 位。所以这种压缩率的降低是可以接受的。

现在对哈夫曼编码算法、区间编码算法和算术编码算法，进行压缩耗时和解压缩耗时的测试。测试结果如表 7.3 和表 7.4 所示。

文件	哈夫曼编码		区间编码		算术编码	
	压缩耗时	解压耗时	压缩耗时	解压耗时	压缩耗时	解压耗时
bib	16	0	16	15	31	31
book1	15	31	93	156	203	219
book2	16	16	94	109	156	172
geo	0	0	16	16	32	31
news	15	0	47	78	109	109
obj1	0	0	0	15	0	16
obj2	0	0	31	47	78	62
paper1	0	0	16	0	16	15
paper2	0	0	0	15	16	15
paper3	0	0	0	15	15	15
paper4	0	0	15	0	0	15
paper5	0	0	0	0	0	15
paper6	0	0	0	16	15	0
pic	15	16	47	47	63	63
progc	0	0	0	0	16	16
progl	0	0	0	16	15	15
progp	0	0	0	15	15	16
trans	0	0	16	16	16	31

表 7.3 卡尔加里语料库算法耗时测试

文件	哈夫曼编码		区间编码		算术编码	
	压缩耗时	解压耗时	压缩耗时	解压耗时	压缩耗时	解压耗时
bible.txt	110	94	516	734	1047	1063
E.coli	110	125	485	718	828	906
world192.txt	78	62	328	469	687	688

表 7.4 大型语料库算法耗时测试

注意，表 7.3 和表 7.4 中的耗时数据都是以毫秒为单位的。观察表 7.3 和表 7.4 中的测试数据可以发现，哈夫曼编码的速度无疑是最快的，其次是区间编码，算术编码最慢。区间编码利用运算上精度的损失换取了速度上的优势。压缩数据时，算术编码的耗时几乎是区间编码耗时的 2 倍。相比之下，区间编码压缩率的损失非常的值得。另外，区间编码在压缩数据时，需要进行频度统计、区间计算、正规化等操作。在解压缩时，除了要进行上述的操作外，还要进行估算频度的查找。这就意味着，区间编码的解压缩算法比压缩算法更耗时。这点可以从表 7.4 中比较明显的看出来。解压耗时比压缩耗时多了大约 100 毫秒到 200 毫秒左右。

综合分析压缩率和算法耗时，哈夫曼编码的综合性能最高，其次是区间编码，最后是算术编码。区间编码的压缩率比哈夫曼编码略高，但是算法耗时比哈夫曼编码慢很多。当然，这并不是说区间编码不如哈夫曼编码。当使用自适应模型或者是高阶模型的时候，哈夫曼编码的性能下降的很快。相反，此时区间编码却可以获得不错的性能。哈夫曼编码经过数年的发展在性能上已经很难再提高了。而区间编码由于其特点，可以很好的与高阶模型相配合。比如 PPM。所以，区间编码以及高阶模型已经成为目前无损压缩技术的研究重点。

8 结束语

区间编码由于其无版权限制、更高的压缩率以及更适用于高阶模型，已经被大量的使用。本文全面介绍了区间编码算法的理论基础和实现方式。详细讨论了区间编码原理、正规化操作、区间编码实现、二进制索引树的理论和实现等技术。并给出了一个切实可行的应用程序。希望本文能够对压缩算法的研究人员有所帮助。

参考文献

[1] G. N. N. Martin, Range encoding: an algorithm for removing redundancy from a digitised message, Video & Data Recording Conference, Southampton, 1979, July 24-27.

[2] Peter M. Fenwick, A New Data Structure for Cumulative Frequency Tables, Software-Practice and Experience, 1994, 24(3), 327-336.

[3] Ian H. Witten, Radford M. Neal, John G. Cleary, Arithmetic Coding for Data Compression, Communications of the ACM, 1987, 30(6), 520-541.

[4] 叶叶，范式哈夫曼算法的分析与实现，编程论坛，2010, http://programbbs.com/bbs/view12-29332-1.htm .

[5] The Canterbury Corpus, http://corpus.canterbury.ac.nz/descriptions/ .

附录：项目说明

1 程序项目

本程序项目是论文《区间编码算法的分析与实现》（以下简称“《区》”）的配套程序。除 ArithmeticCoding.pas 文件外，本程序项目中的其它文件作为免费开源程序进行发布。这些文件中的代码无需任何授权或许可即可用于个人和商业目的。使用者一切后果自负。ArithmeticCoding.pas 文件仅供算法研究使用。ArithmeticCoding.pas 文件中实现的算法如具有版权的，由版权所有人拥有。

本程序项目在 Delphi 7.0 下开发，编译通过并且调试正常。本程序项目提供了一个文件到文件的，区间编码算法的压缩程序。本人发表过另一篇论文《范式哈夫曼算法的分析与实现》（以下简称“《范》”）。本程序项目中的部分文件来自于论文《范》中的配套文件。

本程序项目中的主要文件及相关说明如表 1.1 所示。

文件	说明
Project1.dpr	项目主文件
Project1.exe	项目可执行文件
Unit1.pas	主窗口单元文件
Unit1.dfm	主窗口窗体文件
MemoryBuffer.pas	内存缓冲区管理单元文件
CanonicalHuffman.pas	范式哈夫曼算法实现单元文件
RangeCoding.pas	区间编码算法实现单元文件
ArithmeticCoding.pas	算术编码算法实现单元文件

表 1.1 项目主要文件说明

在程序项目中 MemoryBuffer.pas 和 CanonicalHuffman.pas 两个文件来自于论文《范》中的配套文件。这两个文件的详细说明请参考论文《范》中的项目说明。另外，RangeCoding.pas 是核心文件。在这个文件中实现了论文《区》中描述的区间编码的算法。同时，这个文件可以与 MemoryBuffer.pas 文件一起独立使用。将这两个文件添加到其它项目中，调用其中的相关函数就可以实现数据压缩。类似的，ArithmeticCoding.pas 文件也可以与 MemoryBuffer.pas 文件一起独立使用。注意，ArithmeticCoding.pas 文件中的算法可能带有版权。

2 RangeCoding.pas文件

我在这个单元文件里编写了区间编码算法的实现代码。在这个单元文件中一共定义了 3 个类。其中，TRCCoder 实现了编解码过程，TRCModel 实现了统计模型，TRCCoding 提供

了对 TRCCoder 和 TRCModel 的简单封装。

TRCCoder 类实现了区间编码算法的编解码过程。由于将编码部分与模型部分相分离，TRCCoder 只负责编解码的实现。它需要模型部分为它提供频度的统计信息，才可以正常工作。在 TRCCoder 类中的 Encode 方法和 FinishEncode 方法实现了编码的过程。对应于论文《区》中的第 4.1 节。TRCCoder 类中的 DecodeTarget 方法和 Decode 方法实现了解码的过程。对应于论文《区》中的第 4.2 节。TRCCoder 类的其它方法说明见单元文件中的相关注释。

TRCModel 类实现了统计模型部分的过程。TRCModel 主要是对 BIT 结构的封装，实现了频度信息的统计，并调用 TRCCoder 实例完成编解码。在 TRCCoder 类中的 GetFreq 方法实现了统计指定符号频度的过程。对应于论文《区》中的第 5.3 节。TRCCoder 类中的 IncFreq 方法实现了更新指定符号频度的过程。对应于论文《区》中的第 5.4 节和第 5.5 节。TRCCoder 类中的 FindEstFreq 方法实现了查找估算频度的过程。对应于论文《区》中的第 5.6 节。TRCModel 类的其它方法说明见单元文件中的相关注释。

TRCCoding 类实现了区间编码算法的编解码过程。TRCCoding 类是对 TRCCoder 和 TRCModel 的简单封装。TRCCoding 类提供了一个缓冲区到缓冲区的压缩与解压缩过程。TRCCoding 类中的 Encode 方法是以字节为单位从缓冲区中读取数据进行压缩，然后保存到另一缓冲区中。TRCCoding 类中的 Decode 方法是从缓冲区中读取压缩数据进行解压缩，然后保存到另一缓冲区中。在 Unit1.pas 单元文件中的测试程序就是调用 TRCCoding 类中的相关方法完成文件压缩与解压缩的。

3 ArithmeticCoding.pas文件

ArithmeticCoding.pas 文件的结构与 RangeCoding.pas 文件完全一致。其中，不同的地方在于 TACCoder 类的 Encode 方法、FinishEncode 方法、DecodeTarget 方法和 Decode 方法。虽然方法命名一样，参数也一样。但是 TACCoder 类中实现的是算术编码，TRCCoder 类中实现的是区间编码。关于这两种算法的区别，请参考论文《区》中的第 6 节。另外，TACModel 类与 TRCModel 类完全一致，只有命名上的不同。

最后注意，ArithmeticCoding.pas 文件仅供算法研究使用。ArithmeticCoding.pas 文件中实现的算法如具有版权的，由版权所有人拥有。

叶叶

2010 年 8 月 30 日