# 基于64位整型实现的rANS范围编码算法：原理、优化与C++实践

### 1. 引言：熵编码的演进与rANS的定位

#### 1.1 传统熵编码回顾：Huffman编码与算术编码的优缺点

在无损数据压缩领域，熵编码是核心组成部分，其目的是利用符号的概率分布来为高频符号分配较短的编码，为低频符号分配较长的编码，从而逼近香农熵的理论极限。长期以来，两种主流方法主导了该领域：霍夫曼编码（Huffman Coding）和算术编码（Arithmetic Coding）。

霍夫曼编码是一种基于贪心策略的算法，通过构建一棵二叉树来生成前缀码。它的最大优势在于实现简单且速度极快，解码过程通常可以通过简单的查表操作完成2。然而，霍夫曼编码存在一个根本性的局限性：它为每个符号分配的编码长度必须是整数位。当符号的概率不是2的幂次方时，这会导致显著的压缩效率损失，无法达到最优的香农熵界限2。

与此相对，算术编码则通过将整个消息编码为一个单一的、介于0.0和1.0之间的浮点数来实现压缩。它通过维护一个区间（由上下两个界限定义），并根据每个符号的概率不断缩小该区间。这种方法能够为每个符号分配非整数位长度的编码，从而能无限逼近香农熵的理论极限，在压缩率上达到最优1。然而，其实现复杂，通常需要进行浮点运算或高精度定点运算，这导致其计算开销大，速度相对较慢2。

这两种传统方法代表了压缩技术在**速度-压缩率**权衡上的两个极端。霍夫曼编码牺牲压缩率以换取极致速度，而算术编码则牺牲速度以换取最优压缩率。这种二元对立驱动了对新一代熵编码算法的探索，以期找到一个更优的平衡点。

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 压缩率（逼近香农熵程度） | 速度 | 实现复杂度 |
| **Huffman编码** | 良好（但有整数位编码损失） | 极快（查表） | 低 |
| **算术编码** | 优（可无限逼近理论极限） | 慢（浮点/高精度运算） | 高 |
| **rANS** | 优（接近理论极限） | 极快（整数运算） | 中 |

#### 1.2 不对称数制系统（ANS）的诞生：兼具速度与压缩率的混合体

不对称数制系统（Asymmetric Numeral Systems, ANS）是由Jarosław Duda于2007年提出的一类新型熵编码家族。其核心思想是，它不再像算术编码那样操作浮点区间，而是利用一个单一的、代表整个编码消息的**整数状态变量 x** 来进行编码。这种方法将编码过程从浮点运算转化为了高效的整数运算，从而在根本上提升了性能2。

rANS（Range ANS）是ANS家族中的一个重要成员，因其操作方式与传统的范围编码（Range Encoding）有相似之处而得名2。rANS的独特之处在于，它通过巧妙的整数运算设计，成功地结合了算术编码的优越压缩率和霍夫曼编码的卓越速度4。它被设计为在保持接近香农熵理论极限的压缩效率的同时，达到甚至超越霍夫曼编码的解码速度2。这使得rANS成为了一种理想的混合体，完美地解决了传统熵编码技术在性能上的权衡难题。

### 2. rANS核心原理：从数学到算法

#### 2.1 不对称数制系统：状态变量 x 的“基数转换”

rANS的核心思想是将数据编码为单个整数状态 x。从数学上看，这种编码方法可以被理解为一种**可逆的基数转换**。rANS首先将所有自然数分割成多个不相交的子集，每个子集对应一个特定的符号。这些子集的大小比例与对应符号的频率或概率完全一致3。例如，在一个由符号'A'和'B'组成的语言中，如果'A'的概率是1/2，'B'的概率是1/2，那么自然数集合就会被平均分配给'A'和'B'，例如所有偶数分配给'A'，所有奇数分配给'B'3。

编码操作的本质是将一个整数状态 x，根据要编码的新符号 s，映射到一个新的整数状态 x\_{new}。这个过程就像在数学上将 x 转换到了一种新的、包含了新符号信息的数制。解码过程则是完全相反的，通过 x\_{new} 还原出原始的符号 s 和上一状态 x。这种纯粹的整数操作特性是rANS能够高效实现的根本原因。

#### 2.2 编码原理：状态扩展与符号信息嵌入

rANS的编码过程是一个状态扩展的过程。每编码一个新符号，当前状态 x 都会被更新为一个更大的值 x\_{new}。这个过程通过一个核心公式来实现：

xnew​=⌊x/freq(s)⌋⋅M+start(s)+(x(modfreq(s)))

* 其中 x 是当前的整数状态。
* s 是要编码的符号。
* freq(s) 是符号 s 的频率。
* start(s) 是符号 s 的累积频率（即所有在 s 之前的符号的频率之和）。
* M 是所有符号的总频率5。

这个公式的作用是，将当前状态 x 映射到一个更大的、由 start(s) 和 freq(s) 定义的子区间内。这就像是在一个更大的数制空间中为 x 找到一个"新家"。这种操作将新符号的信息巧妙地嵌入到了 x 的最低有效位部分，从而实现了信息的叠加6。

随着编码的持续进行，状态变量 x 的值会呈指数级增长。为了防止其超过计算机整数类型的表示范围（例如64位整型的上限），rANS引入了\*\*归一化（Renormalization）\*\*机制6。当

x 达到预设的阈值时，编码器会将其高位（例如32位）写入输出数据流中，然后将 x 缩小回一个可控的范围。这个过程确保了编码器可以在不损失信息的情况下，持续处理任意长度的输入数据流7。

#### 2.3 解码原理：状态回溯与信息提取

rANS的解码过程是编码的精确逆过程7。给定当前状态

x，解码器需要从 x 中提取出原始符号 s 和上一状态 x\_{prev}。这个过程通过以下逆向公式实现7：

1. 从 x 中提取槽位信息：  
   slot = x (mod M)  
   其中 M 是总频率。slot 值的范围在 [0,M−1] 之间。
2. 通过槽位查找符号：  
   slot 值与符号的累积频率相关。解码器通过查找预先建立的累积频率表，找到满足 start(s) ≤ slot < start(s)+freq(s) 的符号 s7。
3. 计算上一状态 x\_{prev}：  
   x\_prev ​= ⌊x/M⌋ ⋅ freq(s) + slot − start(s)  
   其中 freq(s) 和 start(s) 对应于步骤2中找到的符号 s。

值得注意的是，rANS的解码顺序具有\*\*后进先出（LIFO）\*\*的特性2。这是因为编码器在每次编码时，都将新符号的信息附加到状态

x 的最低有效位上。因此，为了还原原始数据，解码器必须从最后编码的符号开始，逐步向后回溯，这与编码的顺序正好相反。这要求编码器在处理完整个数据流后，要么将编码后的数据流反转，要么从尾部开始写入和读取，以供解码器按LIFO顺序处理4。

与编码过程类似，解码过程也需要进行**归一化**。当状态 x 在解码过程中变得过小时，解码器会从输入数据流中读取高位数据，并将其与当前的 x 合并，从而将状态“扩展”回操作范围，这个操作与编码时的归一化正好相反7。

#### 2.4 符号表与累积频率：编码与解码的基石

rANS的编码和解码都依赖于一个**概率模型**。对于静态模型，这个模型就是符号的频率统计。一个关键的数据结构是累积频率表，它存储了每个符号的累积频率 start 值，即该符号在字母表中的位置之前的所有符号的频率总和5。

在解码过程中，如何高效地从 slot 值找到对应的符号 s 至关重要。朴素的实现可能会使用线性查找或二分查找，其时间复杂度分别为$O(N)$和$O(\log\_2 N)$（其中N是字母表大小）5。然而，为了追求极致性能，许多高性能实现会使用预计算的查找表，例如别名方法（alias method），从而实现\*\*O(1)\*\*的查找时间5。这种优化是rANS能与查表式霍夫曼编码在速度上竞争的核心技术之一。

### 3. 64位整型实现的优势与技术挑战

#### 3.1 64位架构的天然优势：更大的状态空间与高效数据处理

将rANS算法实现为64位版本（使用uint64\_t类型）带来了显著的性能和设计优势。

首先，64位整型能够表示 264 种不同的状态，这远超32位整型所能表示的 232 种状态8。这为rANS的状态变量

x 提供了巨大的工作空间，使得它可以在归一化前累积更多的信息。这意味着编码器可以更长时间地运行，而无需将中间结果写入输出流，从而减少了昂贵的I/O和归一化操作的频率9。

其次，现代64位处理器原生支持64位寄存器和指令，可以一次性处理64位数据8。对于rANS这种以位操作和整数运算为核心的算法，其核心计算（如乘法、位移、加法）可以直接在硬件层面以更高的效率完成。这种直接利用64位架构的计算能力，是选择64位实现的根本原因，因为它直接影响了算法的性能上限和数据吞吐量。

#### 3.2 高性能优化：用乘法替代除法

rANS理论公式中的除法和模运算在现代CPU上是相对较慢的指令。当除数是常数时，编译器通常会利用预先计算的倒数将除法优化为更快的乘法和位移操作。rANS的高性能实现正是利用了这一技术5。

这一优化的核心在于利用64位平台提供的128位乘法指令。例如，在GCC中可以使用unsigned \_\_int128，在MSVC中可以使用\_\_umulh内部函数，来获取两个64位整数相乘后128位结果的高64位10。通过预先计算好每个符号频率的倒数

rcp\_freq 和位移量 rcp\_shift，理论公式中的 $\lfloor x / \text{freq}(s) \rfloor$ 就可以被转换为 Rans64MulHi(x, rcp\_freq) >> rcp\_shift。这种方法能够精确地模拟整数除法，且不会产生舍入误差5。

这一优化是rANS能与霍夫曼在速度上竞争的决定性因素。它将原本包含慢速除法/模运算的编码/解码循环，转化为了由快速的整数乘法、加法和位移构成的纯粹的整数运算，极大地提升了数据处理速度。

#### 3.3 归一化策略：64位状态与32位数据流的交互

在实际实现中，一个巧妙的工程权衡是使用64位的状态变量，但每次归一化时只写入/读取32位数据10。

**编码归一化：** 编码器维护一个64位的状态 x。当 x 的值达到或超过一个预设的阈值 x\_max 时，编码器将 x 的低32位写入输出缓冲区，然后将 x 整体右移32位。例如，阈值可以被设置为 x\_max = ((RANS64\_L >> scale\_bits) << 32) \* freq，其中 RANS64\_L 是一个基本状态下界10。

**解码归一化：** 解码器在状态 x 的值降到某个阈值（例如RANS64\_L）以下时，会从输入流中读取一个32位整数，并将其左移32位后与当前的 x 进行或操作（x = (x << 32) | input\_word），从而将状态 x 扩展回操作范围10。

这种策略利用了64位状态的大空间来减少归一化频率，从而提高了编码效率，同时又保持了与32位数据流的兼容性，实现了高效的I/O操作。

### 4. 64位rANS编码器核心代码（C++）

为了更好地理解rANS的工作原理，下面将展示其64位整型版本的C++代码实现，并附带中文注释。

#### 4.1 环境配置与符号表数据结构

首先，我们需要定义一些基本类型和函数，特别是针对不同编译器的128位乘法高位获取函数。

C++

#**include** <cstdint>  
#**include** <vector>  
  
// 状态变量类型  
using Rans64State = uint64\_t;  
  
// 检查是否需要归一化的下限阈值  
const uint64\_t RANS64\_L = 1ULL << 32;  
  
#**if** defined(\_MSC\_VER)  
#**include** <intrin.h>  
// MSVC编译器: 使用 \_\_umulh 内部函数获取64位乘法结果的高64位  
static inline uint64\_t Rans64MulHi(uint64\_t a, uint64\_t b) {  
 return \_\_umulh(a, b);  
}  
#**elif** defined(\_\_GNUC\_\_)  
// GNU编译器: 使用 \_\_int128 类型进行128位乘法并右移  
static inline uint64\_t Rans64MulHi(uint64\_t a, uint64\_t b) {  
 return (uint64\_t) (((unsigned \_\_int128)a \* b) >> 64);  
}  
#**else**  
#**error** "Unknown/unsupported compiler!"  
#**endif**

编码器需要一个预先计算的符号表，以利用倒数乘法优化。这个结构体包含了所有必要的参数。

|  |  |  |
| --- | --- | --- |
| 参数名 | 数据类型 | 作用描述 |
| freq | uint32\_t | 符号的频率 |
| start | uint32\_t | 符号的累积频率 |
| rcp\_freq | uint64\_t | 频率的64位定点倒数乘法参数 |
| rcp\_shift | uint32\_t | 倒数乘法后的右移位数 |
| bias | uint32\_t | 偏差值，用于校正倒数乘法结果 |

C++

// 编码器符号数据结构  
struct Rans64EncSymbol {  
 uint64\_t rcp\_freq;  
 uint32\_t freq;  
 uint32\_t bias;  
 uint32\_t cmpl\_freq; // Complement of frequency: (1 << scale\_bits) - freq  
 uint32\_t rcp\_shift;  
};  
  
// 初始化编码器符号表  
void Rans64EncSymbolInit(Rans64EncSymbol\* s, uint32\_t start, uint32\_t freq, uint32\_t scale\_bits) {  
 // 核心步骤：根据freq计算rcp\_freq, rcp\_shift和bias  
 //...  
 // 该部分代码逻辑复杂，但其目标是确保：  
 // x\_new = (x/freq)\*M + start + (x%freq)  
 // 能够通过以下优化公式实现：  
 // x\_new = bias + x + q\*(M - freq)  
 // 其中 q = mul\_hi(x, rcp\_freq) >> rcp\_shift  
 // 这将理论上的除法/模运算转换为了纯粹的加法、乘法和位移。  
 //...  
}

#### 4.2 编码函数 Rans64EncPut

该函数负责将一个符号编码到当前状态中，并在必要时进行归一化。

C++

// 编码函数  
// r: rANS状态指针  
// pptr: 输出缓冲区指针  
// sym: 符号信息  
// scale\_bits: 频率规模的位宽（例如12，表示总频率M=4096）  
void Rans64EncPut(Rans64State\* r, uint32\_t\*\* pptr, const Rans64EncSymbol\* sym, uint32\_t scale\_bits) {  
 uint64\_t x = \*r;  
  
 // 归一化检查  
 // 当x达到x\_max时，写入低32位数据并右移  
 uint64\_t x\_max = ((RANS64\_L >> scale\_bits) << 32) \* sym->freq;  
 if (x >= x\_max) {  
 // 将状态的低32位写入缓冲区  
 \*(--(\*pptr)) = (uint32\_t)x;  
 // 状态右移32位  
 x >>= 32;  
 }  
  
 // 核心编码逻辑：使用倒数乘法优化  
 // 计算商q = floor(x/freq)  
 uint64\_t q = Rans64MulHi(x, sym->rcp\_freq) >> sym->rcp\_shift;  
  
 // 计算余数r = x % freq  
 // 使用 r = x - q\*freq  
 uint64\_t rem = x - q \* sym->freq;  
  
 // 更新状态  
 // x\_new = q \* M + start + rem  
 // M = (1 << scale\_bits)  
 x = q \* (1ULL << scale\_bits) + sym->start + rem;  
  
 \*r = x;  
}  
  
// 编码流程封装  
// 由于rANS解码是LIFO，所以编码时需要从输入数据的末尾开始向开头编码。  
std::vector<uint32\_t> Rans64Encode(const std::vector<uint8\_t>& data,  
 const std::vector<Rans64EncSymbol>& syms,  
 uint32\_t scale\_bits) {  
 std::vector<uint32\_t> output\_buffer(data.size() + 4); // 预估大小  
 uint32\_t\* pptr = output\_buffer.data() + output\_buffer.size();  
   
 Rans64State x = RANS64\_L; // 编码器初始状态  
  
 // 从后往前遍历输入数据进行编码  
 for (int i = data.size() - 1; i >= 0; --i) {  
 uint8\_t symbol = data[i];  
 const Rans64EncSymbol& sym = syms[symbol];  
 Rans64EncPut(&x, &pptr, &sym, scale\_bits);  
 }  
  
 // 刷新剩余状态  
 while (x > 0) {  
 \*(--pptr) = (uint32\_t)x;  
 x >>= 32;  
 }  
  
 // 调整缓冲区大小并返回  
 output\_buffer.erase(output\_buffer.begin(), pptr);  
 return output\_buffer;  
}

### 5. 64位rANS解码器核心代码（C++）

解码是编码的逆过程，核心在于从状态中提取符号并回溯到前一状态。

#### 5.1 解码器状态初始化 Rans64DecInit

解码器必须从编码数据流的末尾开始读取，因为最后编码的符号信息位于数据流的开头。

C++

// 解码器状态数据结构  
struct Rans64DecSymbol {  
 uint32\_t start;  
 uint32\_t freq;  
};  
  
// 初始化解码器状态  
// r: rANS状态指针  
// pptr: 输入缓冲区指针  
void Rans64DecInit(Rans64State\* r, uint32\_t\*\* pptr) {  
 uint64\_t x;  
 // 从缓冲区读取初始状态  
 // 编码时最后写入的32位是最高位  
 x = (uint64\_t)((\*pptr)) << 32;  
 x |= (uint64\_t)((\*pptr));  
 \*pptr += 2;  
 \*r = x;  
}

#### 5.2 符号查找与解码函数 Rans64DecAdvance

这个函数是解码循环的核心，它从当前状态中提取符号，并根据逆公式更新状态。

C++

// 查找符号的辅助函数  
uint8\_t FindSymbol(uint32\_t slot, const std::vector<Rans64DecSymbol>& dec\_syms) {  
 // 示例：这里使用简单的二分查找  
 // 实际高性能实现会使用O(1)查找表  
 //...  
 int low = 0, high = dec\_syms.size() - 1;  
 while (low <= high) {  
 int mid = low + (high - low) / 2;  
 if (slot >= dec\_syms[mid].start && slot < dec\_syms[mid].start + dec\_syms[mid].freq) {  
 return mid;  
 } else if (slot < dec\_syms[mid].start) {  
 high = mid - 1;  
 } else {  
 low = mid + 1;  
 }  
 }  
 return -1; // 错误处理  
}  
  
// 解码函数  
// r: rANS状态指针  
// pptr: 输入缓冲区指针  
// dec\_syms: 解码器符号表  
// scale\_bits: 频率规模的位宽  
uint8\_t Rans64DecAdvance(Rans64State\* r, uint32\_t\*\* pptr, const std::vector<Rans64DecSymbol>& dec\_syms, uint32\_t scale\_bits) {  
 uint64\_t x = \*r;  
 uint64\_t M = 1ULL << scale\_bits;  
  
 // 1. 获取槽位信息  
 uint32\_t slot = x % M;  
  
 // 2. 查找符号  
 uint8\_t symbol = FindSymbol(slot, dec\_syms);  
 const Rans64DecSymbol& sym = dec\_syms[symbol];  
  
 // 3. 计算上一状态  
 x = (x / M) \* sym.freq + slot - sym.start;  
  
 // 4. 归一化检查  
 if (x < RANS64\_L) {  
 x = (x << 32) | \*\*pptr;  
 (\*pptr)++;  
 }  
  
 \*r = x;  
 return symbol;  
}  
  
// 解码流程封装  
// 由于rANS解码是LIFO，所以解码出的符号序列需要反转  
std::vector<uint8\_t> Rans64Decode(const std::vector<uint32\_t>& encoded\_data,  
 size\_t original\_size,  
 const std::vector<Rans64DecSymbol>& dec\_syms,  
 uint32\_t scale\_bits) {  
 std::vector<uint8\_t> decoded\_data;  
 decoded\_data.reserve(original\_size);  
   
 const uint32\_t\* pptr = encoded\_data.data();  
 Rans64State x;  
 Rans64DecInit(&x, const\_cast<uint32\_t\*\*>(&pptr));  
  
 // 解码符号  
 for (size\_t i = 0; i < original\_size; ++i) {  
 decoded\_data.push\_back(Rans64DecAdvance(&x, const\_cast<uint32\_t\*\*>(&pptr), dec\_syms, scale\_bits));  
 }  
   
 // 由于解码顺序是LIFO，所以最终的输出序列需要反转  
 std::reverse(decoded\_data.begin(), decoded\_data.end());  
   
 return decoded\_data;  
}

### 6. 结论与性能展望

#### 6.1 rANS的综合评估：压缩比、速度与内存占用

综合来看，rANS提供了一个在压缩比和速度之间实现了卓越平衡的熵编码解决方案。

**压缩比**方面，由于其基于概率的整数状态操作，rANS能够实现接近算术编码的压缩效率，能够有效地利用香农熵的理论极限，即便在符号概率不是2的幂次方时，也不会有显著的压缩损失2。

**速度**方面，通过利用64位架构的天然优势和巧妙的倒数乘法优化，rANS的核心计算完全由快速的整数指令完成，避免了浮点运算的开销。这使得其数据吞吐量可以与查表式的霍夫曼编码相媲美，甚至在某些硬件和特定应用场景下表现更佳2。

**内存占用**方面，rANS需要存储符号频率和累积频率表，这与霍夫曼编码类似，但其核心数据结构是一个单一的整数状态，比算术编码需要维护的浮点区间或高精度定点数更加轻量和高效5。

#### 6.2 适用场景：为何rANS在现代压缩格式中日益重要

由于其在性能上的优越平衡，rANS已成为许多现代高性能压缩库的首选熵编码器，例如Facebook的Zstd和Lz4的某些变体。它特别适合于那些对实时性有高要求的应用场景，例如：

* **高吞吐量数据流：** 例如数据库、日志文件和大数据处理。
* **游戏和多媒体：** 游戏资源包、纹理数据和实时视频流。
* **系统级压缩：** 操作系统内核、文件系统和内存管理。

rANS的整数状态操作使其比传统的算术编码更易于并行化，这在多核处理器日益普及的今天，是其相比于其他算法的又一大优势。

#### 6.3 未来的研究方向与潜在优化空间

尽管rANS已经非常高效，但仍有潜在的研究方向和优化空间。

* **自适应模型：** 当前的静态模型需要预先统计符号频率，这对于不可预测的数据流或小文件压缩并不理想。未来的研究将集中于开发更高效的自适应rANS变体，使其能够动态更新概率模型，从而更好地适应输入数据流的变化。
* **并行化：** rANS的整数状态更新具有链式依赖性，但其归一化操作相对独立，这为并行化提供了可能性。未来的工作将探索如何更有效地对rANS进行并行化，以充分利用多核处理器的计算能力，进一步提升其数据吞吐量。

#### 引用的著作

1. Arithmetic coding - Wikipedia, 访问时间为 八月 30, 2025， <https://en.wikipedia.org/wiki/Arithmetic_coding>
2. Understanding the ANS Compressor - Kedar Tatwawadi, 访问时间为 八月 30, 2025， <https://kedartatwawadi.github.io/post--ANS/>
3. Asymmetric numeral systems - Wikipedia, 访问时间为 八月 30, 2025， <https://en.wikipedia.org/wiki/Asymmetric_numeral_systems>
4. Rescaling of Symbol Counts for Adaptive rANS Coding - EURASIP, 访问时间为 八月 30, 2025， <https://eurasip.org/Proceedings/Eusipco/Eusipco2023/pdfs/0000585.pdf>
5. rANS with static probability distributions - The ryg blog - WordPress.com, 访问时间为 八月 30, 2025， <https://fgiesen.wordpress.com/2014/02/18/rans-with-static-probability-distributions/>
6. Entropy coding by a beginner for beginners - Part 7: ANS (rANS, FSE/tANS) | Akreson, 访问时间为 八月 30, 2025， <https://akreson.github.io/en/posts/entropy-encoding-part7/>
7. Asymmetric Numeral Systems - GitHub Pages, 访问时间为 八月 30, 2025， <https://stanforddatacompressionclass.github.io/notes/lossless_iid/ans.html>
8. Benefits of a 64-bit application - MicroSurvey Helpdesk, 访问时间为 八月 30, 2025， <https://helpdesk.microsurvey.com/article/1072-benefits-of-a-64-bit-application>
9. What Are the Advantages of 64-Bit Systems? How Does It Affect Gaming? - Lenovo, 访问时间为 八月 30, 2025， <https://www.lenovo.com/us/en/glossary/64-bit-system/>
10. ryg\_rans/rans64.h at master - GitHub, 访问时间为 八月 30, 2025， <https://github.com/rygorous/ryg_rans/blob/master/rans64.h>