深入理解并实现基本的霍夫曼编码（Huffman Coding）算法

黄梓淳

Sep 30, 2025

从信息论基础到手写代码，掌握这一经典无损压缩技术的核心

在数字世界中，数据压缩技术无处不在，从常见的 ZIP 或 RAR 文件到 MP3 音乐和 JPEG 图片，这些格式都依赖于高效的压缩算法来减少存储空间和传输带宽。数据压缩主要分为无损压缩和有损压缩两大类；无损压缩确保原始数据可以完全恢复，而有损压缩则通过牺牲部分信息来换取更高的压缩率。霍夫曼编码作为一种经典的无损压缩算法，由 David A. Huffman 在 1952 年提出，它结合了贪婪算法和最优前缀码的特性，至今仍是许多现代压缩算法如 DEFLATE 的基石。本文的目标不仅是解释霍夫曼编码的原理，还将通过代码实现带领读者从理论走向实践，真正掌握这一技术。

1　预备知识：信息论与编码的基石

定长编码如 ASCII 码为所有字符分配相同长度的编码，这在字符分布不均的文本中效率低下，因为它无法利用高频字符的优势。变长编码的引入解决了这一问题，其核心思想是为出现频率高的字符分配短码，为频率低的字符分配长码，从而降低整体编码长度。前缀码是实现变长编码的关键；它定义为任何一个字符的编码都不是另一个字符编码的前缀，这保证了编码的唯一可译性，解码时无需分隔符也不会产生歧义。举例来说，如果一个编码方案中 A 的编码为 0，B 的编码为 01，那么解码字符串 01 时会出现歧义，因为它可能代表 A 后跟 B 或单独一个 B；而前缀码如 A=0 和 B=10 则能避免这种问题，确保解码过程清晰无误。

2　霍夫曼编码的核心：霍夫曼树的构建

霍夫曼编码的核心在于构建一棵霍夫曼树，这棵树通过贪婪算法逐步合并节点来实现最优编码。首先，我们需要统计待编码数据中每个字符出现的频率，这可以通过遍历数据并记录每个字符的出现次数来完成。接着，将每个字符视为一个只有根节点的二叉树，节点的权值即为该字符的频率，所有这样的树构成一个初始森林。然后，进入循环合并阶段：当森林中树的数目大于一时，每次从中选出两个权值最小的根节点，创建一个新的内部节点，其权值为这两个节点权值之和，并将选出的节点作为新节点的左右孩子（通常约定权值小的在左，权值大的在右，但这不影响压缩效率）。将新树放回森林并移除原来的两棵树，重复此过程直到只剩下一棵树，这棵树就是霍夫曼树。为了可视化这一过程，我们可以以字符串 ABRACADABRA 为例：首先统计频率，A 出现 5 次，B 和 R 各 2 次，C 和 D 各 1 次；然后从森林中合并最小权值节点，例如先合并 C 和 D（权值各为 1），形成权值为 2 的新节点，再与 B 或 R 合并，依此类推，最终构建出完整的霍夫曼树。

3 从霍夫曼树生成编码表

一旦霍夫曼树构建完成，就可以从中生成编码表。从根节点出发，向左子树走标记为 0，向右子树走标记为 1，遍历每个叶子节点（即字符节点）的路径所组成的 0 和 1 序列，即为该字符的霍夫曼编码。例如，如果字符 A 的路径是左-左，那么它的编码就是 00。这种生成方式自动满足了前缀码的性质，因为所有字符都位于叶子节点，任何字符的编码路径都不会成为另一个字符路径的前缀，从而确保了解码的唯一性。通过深度优先遍历霍夫曼树，我们可以递归地记录路径，并在到达叶子节点时将路径存入字典，形成完整的编码表。

4 实战：手把手实现霍夫曼编码

在实战部分，我们将使用 Python 语言来实现霍夫曼编码的关键步骤。首先，设计一个节点类来表示霍夫曼树中的节点，每个节点包含字符、频率以及左右子节点的引用。

```
class Node:
    def __init__(self, char, freq):
        self.char = char # 字符，内部节点可为 None
        self.freq = freq # 频率
        self.left = None # 左孩子
        self.right = None # 右孩子
```

这段代码定义了一个节点类，其中 char 存储字符，对于内部节点，它可以为 None；freq 存储频率；left 和 right 分别指向左右子节点。这为构建霍夫曼树提供了基础数据结构。

接下来，实现核心算法。首先统计输入字符串中每个字符的频率，构建一个字符-频率字典。

```
def build_frequency_dict(data):
    freq_dict = {}
    for char in data:
        freq_dict[char] = freq_dict.get(char, 0) + 1
    return freq_dict
```

这段代码遍历输入字符串 data，使用字典 freq_dict 记录每个字符的出现次数。get 方法用于安全地获取并更新频率，如果字符不存在则初始化为 0。

然后，构建一个优先级队列（使用最小堆）来高效管理节点。我们将所有字符节点放入堆中，按频率排序。

```
import heapq

def build_huffman_tree(freq_dict):
    heap = []
    for char, freq in freq_dict.items():
        heapq.heappush(heap, (freq, Node(char, freq)))
    while len(heap) > 1:
        freq1, node1 = heapq.heappop(heap)
```

```
        freq2, node2 = heapq.heappop(heap)
        merged_node = Node(None, freq1 + freq2)
        merged_node.left = node1
        merged_node.right = node2
        heapq.heappush(heap, (merged_node.freq, merged_node))
    return heapq.heappop(heap)[1]
```

这段代码首先将每个字符节点压入最小堆，堆根据频率排序。然后循环合并：每次弹出两个最小频率的节点，创建一个新节点作为它们的父节点，权值为频率之和，并将新节点压回堆中。循环直到堆中只剩一个节点，即霍夫曼树的根节点。这体现了贪婪算法的思想，总是合并当前最小的两个权重。

生成编码表时，通过深度优先遍历霍夫曼树，递归记录路径。

```
def build_code_table(root):
    code_table = {}
    def traverse(node, code):
        if node is None:
            return
        if node.char is not None:
            code_table[node.char] = code
        traverse(node.left, code + '0')
        traverse(node.right, code + '1')
    traverse(root, '')
    return code_table
```

这段代码定义了一个递归函数 traverse，从根节点开始，向左走添加 0，向右走添加 1。当遇到叶子节点（char 不为 None）时，将当前路径存入 code_table。这确保了每个字符的编码唯一且符合前缀码规则。

最后，编码数据：遍历原始字符串，将每个字符替换为其霍夫曼编码。

```
def encode_data(data, code_table):
    encoded = ''
    for char in data:
        encoded += code_table[char]
    return encoded
```

这段代码简单地将输入字符串中的每个字符映射到编码表中对应的二进制字符串，并拼接成最终的编码结果。

解码过程同样重要，它需要根据霍夫曼树逐位解析编码字符串。

```
def decode_data(encoded, root):
    decoded = ''
    current = root
    for bit in encoded:
        if bit == '0':
            current = current.left
```

```
        else:
            current = current.right
        if current.char is not None:
            decoded += current.char
            current = root
    return decoded
```

这段代码从根节点开始，根据编码字符串的每一位（0 或 1）遍历霍夫曼树：遇到 0 走向左子树，遇到 1 走向右子树。当到达叶子节点时，输出对应字符并重置到根节点，继续解码剩余部分。这确保了编码字符串可以被准确还原为原始数据。

5 分析与展望

霍夫曼编码之所以是最优前缀码，是因为它通过贪婪算法总是合并当前最小的两个权重，从而最小化整体编码长度，这可以从信息论的角度证明，但本文不展开复杂数学推导。霍夫曼编码的优点包括无损压缩、最优前缀码特性以及原理简单易懂；然而，它也有一些缺点，例如需要预先统计整个数据的频率分布，这可能导致两次遍历数据（统计和编码），并且必须将编码表与压缩数据一起存储或传输，对于小文件来说，编码表的开销可能较大。此外，霍夫曼编码对数据变化敏感，如果频率分布发生变化，就需要重新构建树。在实际应用中，霍夫曼编码被广泛用于 GZIP、PKZIP、JPEG 和 MP3 等格式；同时，自适应霍夫曼编码作为变种，可以在编码过程中动态更新频率模型，无需提前统计，提高了灵活性。

回顾霍夫曼编码的核心思想，它通过字符频率决定码长，并利用二叉树生成前缀码，实现了高效的无损压缩。作为计算机科学中的经典算法，霍夫曼编码不仅在理论上具有启发性，还在实际应用中发挥着重要作用。鼓励读者动手实现代码，并尝试将其应用于简单的文件压缩任务，以加深对数据压缩技术的理解。通过本文的讲解和示例，希望读者能够真正掌握霍夫曼编码的精髓，并在实践中灵活运用。