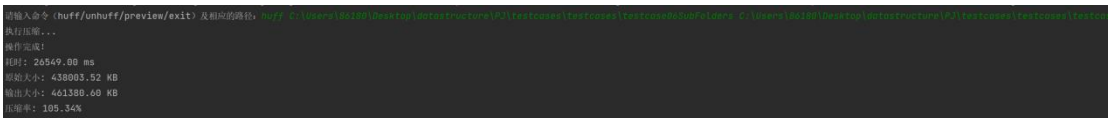
开发文档

压缩成功截图：



一、代码结构

Main.java:

main 函数首先用一个 while 循环来实现以参数的形式指定输入输出，不断等待用户的新的指令，当用户输入 exit 时才退出程序。

接着调用 processCommand 函数，根据输入的参数进入 switch 中的不同 case。如果只有一个参数，则输出"参数不足，请重新输入"，并重新输入。如果第一个参数为 huff，则调用 Compression 函数；如果第一个参数为 unhuff，则调用 Decompression 函数；如果第一个参数为 preview，则调用 FolderDecompression 中的 previewCompressedStructure 函数；如果是其他参数，则输出"未知命令，请使用 'huff', 'unhuff', 'preview' 或 'exit'"，并重新输入。

1、Compression 函数

这个函数负责处理文件或文件夹的压缩操作。它首先检查是否提供了足够的参数（至少三个参数：命令、输入路径、输出路径）。如果参数不足，它会提示用户输入输出路径。

函数测量压缩操作的开始和结束时间，以计算压缩过程所需的总时间。

它根据输入文件是单个文件还是文件夹来调用相应的压缩方法。对于文件夹，调用 FolderCompression 中的 finalFolderCompression 方法；对于单个文件，调用 HuffmanCompression 中的 finalHuffmanCompression 方法。

压缩完成后，它调用 displayCompressionDetails 函数来显示压缩操作的细节，如原始大小、输出大小、压缩率和耗时。

2、Decompression 函数：

这个函数处理压缩文件的解压缩操作。它首先检查指定的压缩文件是否存在。如果文件不存在，会输出相应的错误信息。

函数读取文件的 magic number 以确定文件类型（单个文件或文件夹压缩）。根据 magic number 的值，它调用相应的解压缩方法：HuffmanDecompression 中的 finalHuffmanDecompression 或 FolderDecompression 中的 finalFolderDecompression。

函数还测量解压缩操作的开始和结束时间，并显示操作完成的信息和耗时。解压缩时不显示压缩细节（如原始大小、输出大小、压缩率）。

3、displayCompressionDetails 函数：

这个函数显示压缩或解压缩操作的详细信息。它计算操作的耗时，然后输出。

对于压缩操作，它还计算原始文件/文件夹和压缩后文件的大小（以 KB 为单位），并计算压缩率（以百分比表示），然后输出这些信息。对于解压缩操作，这些额外信息不会显示。

4、calculateFolderSize 函数的作用：

这个函数计算给定文件夹的总大小。它递归地遍历文件夹内的所有文件和子文件夹，累加它们的大小。对于每个遇到的文件夹，它递归调用自身以获取该文件夹内所有文件的总大小。对于文件，它直接添加文件的大小。返回值是文件夹内所有文件的总大小（以字节为单位）。

HuffmanCompression.java：

1、FILE_MAGIC_NUMBER 和 FOLDER_MAGIC_NUMBER：

这两个数组定义了用于标识压缩文件类型的 magic number。FILE_MAGIC_NUMBER 用于标识单个文件的压缩，而 FOLDER_MAGIC_NUMBER 用于标识文件夹的压缩。

2、Node 类：

这是一个内部类，用于构造哈夫曼树。每个 Node 对象代表树中的一个节点，包含数据、频率及其左右子节点的引用。compareTo 方法用于在优先队列中比较节点。

3、buildHuffmanTree 函数：

此函数根据文件数据构造哈夫曼树。首先统计每个字节的频率，然后使用优先队列基于频率构造树。

4、generateHuffmanCodes 函数：

这个函数根据哈夫曼树生成哈夫曼编码表。它使用递归方式遍历树，并为每个叶节点生成一个唯一的二进制编码。

5、writeHuffmanCodesToStream 函数：

此函数将哈夫曼编码表写入数据输出流。这对于稍后解压缩文件时重建哈夫曼树是必要的。

6、compressBinaryData 函数：

这个函数读取原始数据，应用哈夫曼编码，并将编码后的数据写入输出流。处理二进制数据，确保每 8 位写入一个字节。

7、convertInputStreamToByteArray 函数：

此函数将 BufferedInputStream 转换为字节数组，用于构建哈夫曼树和编码表。

8、compressFile 函数：

这是文件压缩的主要函数。它读取输入文件，构建哈夫曼树，生成编码表，应用哈夫曼编码，然后将编码后的数据写入输出文件。

9、finalHuffmanCompression 函数：

这个函数增加了与用户的交互。它检查输入文件是否存在，输出文件是否已存在，并根据用户的选择决定是否覆盖现有文件。然后执行压缩操作。

HuffmanDecompression.java：

1、rebuildHuffmanTree 函数：

这个函数根据哈夫曼编码表重建哈夫曼树，这是解压缩过程中的关键步骤。它创建了一个新的树根节点，并根据编码表中的每个编码逐步构建树的分支。

2、isLeaf 函数：

这个函数用于检测一个节点是否是叶节点。如果一个节点既没有左子节点也没有右子节点，则被视为叶节点。

3、readHuffmanCodesFromStream 函数：

这个函数从压缩文件的数据输入流中读取哈夫曼编码表。它首先读取编码表的大小，然后逐个读取每个字符及其对应的编码。

4、decompressBinaryData 函数：

此函数使用重建的哈夫曼树对二进制数据进行解码。它逐位地遍历编码数据，沿着哈夫曼树向下移动，直到达到叶节点，然后将叶节点的数据写入输出流。

5、decompressFile 函数：

这是解压缩单个文件的主要函数。它读取压缩文件中的哈夫曼编码表和编码后的数据，重建哈夫曼树，然后解码数据并写入到输出文件。

6、finalHuffmanDecompression 函数：

这个函数提供了用户交互，允许用户决定是否覆盖同名的输出文件。它首先从压缩文件中获取原始文件名，然后检查输出文件夹中是否存在同名文件。

7、getOriginalFileName 函数：

这个函数从压缩文件中读取原始文件名。这对于判断输出文件夹中是否已存在同名文件是必要的。

9、FILE_MAGIC_NUMBER 和 checkMagicNumber 函数：

FILE_MAGIC_NUMBER 定义了用于标识压缩文件的 magic number。checkMagicNumber 方法用于验证读取的文件是否为正确的哈夫曼压缩文件。

FolderCompression.java：

1、compressFolder 函数：

这是主函数，用于启动文件夹的压缩流程。首先，它检查并设置输出文件路径，然后打开输出文件的输出流。接下来，它写入用于识别文件夹压缩文件的 magic number，以及原始文件夹的名称。最后，它调用 compressFolderRecursive 方法来递归压缩文件夹内的所有内容。

2、compressFolderRecursive 函数：

这是递归函数，用于遍历文件夹中的所有文件和子文件夹。对于每个文件夹，它写入一个标识符和相对路径，并递归调用自身。对于每个文件，它同样写入一个标识符和相对路径，然后调用 compressFile 方法来压缩文件。

3、compressFile 函数：

这个函数负责压缩单个文件。它首先读取文件内容，构建哈夫曼树，并生成哈夫曼编码表。然后，它重置输入流，并将文件数据通过哈夫曼编码转换为压缩数据。最后，它将压缩数据的长度、有效位数、哈夫曼编码表和压缩数据本身写入输出流。

4、compressBinaryData 函数：

这个函数将输入流中的数据应用哈夫曼编码并转换为压缩数据。它逐字节读取数据，应用哈夫曼编码，并将编码后的数据写入一个临时输出流。该函数还处理最后一个字节可能不满 8 位的情况。

5、finalFolderCompression 函数：

这个函数提供了用户交互界面，允许用户决定是否覆盖已存在的输出文件。它检查输入文件夹是否存在，询问用户是否覆盖已存在的输出文件夹，并根据用户的选择进行相应的操作。

FolderDecompression.java：

1、decompressFolder 函数：

这是主函数，用于启动文件夹的解压缩流程。首先，它打开输入文件的输入流，并检查 magic number 以确认文件格式。接着，它读取原始文件夹的名称，构建输出文件夹的路径，并调用 decompressFolderRecursive 方法来递归解压缩文件夹内的所有内容。

2、decompressFolderRecursive 函数：

这是递归函数，用于遍历压缩文件中的所有文件和子文件夹。对于每个文件夹，它在输出路径中创建相应的文件夹。对于每个文件，它调用 decompressFile 方法来解压缩文件。

3、decompressFile 函数：

这个函数负责解压缩单个文件。它首先读取文件的压缩长度和有效位数，然后读取哈夫曼编码表。接着，它读取压缩数据，并使用 rebuildHuffmanTree 方法重构哈夫曼树。最后，它使用 decompressBinaryData 方法解码数据，并将解码后的数据写入输出文件。

4、decompressBinaryData 函数：

这个函数将压缩的二进制数据解码为原始数据。它逐字节处理压缩数据，根据哈夫曼树解码每个比特，并将解码后的数据写入输出流。

5、finalFolderDecompression 函数：

这个函数提供了用户交互界面，允许用户决定是否覆盖已存在的输出文件夹。它检查压缩文件是否存在，读取解压缩文件夹的名称，并询问用户是否覆盖已存在的同名文件夹。根据用户的选择，它调用 decompressFolder 方法进行解压缩。

6、previewCompressedStructure 函数：

这个函数实现了对压缩文件夹内部结构的预览功能。它读取并确认文件夹的 magic number，然后递归地读取每个文件和子文件夹的路径和类型。通过 printFolderStructure 和 skipCompressedFileData 方法，它在不解压缩的情况下显示文件夹的树形结构。

二、项目“核心需求”与“其他需求”的设计与实现

1、核心需求

（1）文件的压缩与解压 (30%)

①正常压缩/解压非空文件 (20%)

- 设计：使用哈夫曼编码算法对文件进行压缩，将文件中的字节转换为有效的压缩代码，从而减小文件大小。考虑大文件的处理，应避免整型溢出，可能需要使用 long 类型来处理文件大小。
- 实现：首先读取整个文件，计算每个字节的频率并构建哈夫曼树。然后，生成哈夫曼编码表并应用这些编码来压缩文件数据。在解压时，重构哈夫曼树并解码数据以恢复原始文件。

②压缩/解压空文件 (10%)

- 设计：空文件的处理应确保文件系统的有效性。即使文件为空，也要确保在压缩和解压时能够正确处理。
- 实现：之前的设计已经可以正常压缩/解压空文件，无需修改。

③指定压缩包名称与还原文件名 (5%)

- 设计：压缩时允许用户指定输出文件名。解压时，从压缩文件中提取原始文件名，而不依赖于压缩文件的名称。
- 实现：在压缩过程中，将原始文件名作为元数据写入压缩文件。解压时，读取此元数据以确定输出文件名。

（2）文件夹的压缩与解压 (20%)

①压缩/解压非空文件夹 (10%)

- 设计：文件夹压缩应能处理任意深度的文件夹结构。应递归地访问所有子文件夹和文件，并对每个文件应用压缩算法。
- 实现：使用递归方法遍历文件夹和子文件夹。每当遇到文件夹，将其路径和类型（文件夹）记录到压缩文件中。对于文件，应用上述文件压缩方法。

②压缩/解压空文件夹 (5%)

- 设计：空文件夹的压缩应保留其结构。解压时应还原空文件夹。
- 实现：在递归遍历过程中，即使文件夹为空，也应将其路径和类型记录到压缩文件中。在解压时，根据记录的信息创建相应的空文件夹。

③还原文件名、文件夹名 (5%)

- 设计：解压文件夹时，应还原所有文件和子文件夹的原始名称，包括其在文件夹结构中的相对位置。
- 实现：在压缩过程中记录文件和文件夹的相对路径。解压时，根据这些路径信息重建文件夹结构，并使用原始名称创建文件和子文件夹。

2、其他需求

（1）用户交互 (4%)

- 设计：实现一个简单的命令行界面，允许用户通过命令行参数来指定操作（压缩或解压缩），输入文件/文件夹路径和输出路径。提供清晰的使用说明和错误提示。
- 实现：在 main 函数中设置一个 while 循环解析命令行参数，根据参数执行相应的压缩或解压缩操作，直到输入 exit 退出。在操作完成后，显示相关信息如耗时、压缩率等。

（2）检验压缩包来源是否是自己的压缩工具 (4%)

- 设计：使用特定的“magic number”来标识压缩文件，以确保只解压由本工具创建的压缩包。
- 实现：在压缩时，在文件开始处写入一个独特的字节序列。在解压缩时，首先检查这个字节序列，如果不匹配，则给出错误提示并停止操作。

（3）文件覆盖问题 (4%)

- 设计：在执行可能导致文件覆盖的操作前，检查目标路径是否已存在文件。如果存在，提示用户并询问是否覆盖。
- 实现：在压缩或解压缩之前，检查输出路径。如果文件已存在，提示用户并请求确认是否覆盖。根据用户的响应决定是继续操作还是中断。

（4）压缩包预览 (8%)

- 设计：实现一个预览功能，允许用户查看压缩文件或文件夹内的内容结构，而无需解压整个文件。
- 实现：为压缩文件添加一个读取文件结构的功能，不实际解压数据，只展示文件和文件夹的层次结构。如果压缩的是文件夹，显示其内部结构；如果是单个文件，仅显示文件名。使用树形格式来展示结构，如用缩进或特殊字符来表示层级关系。在读取文件结构时，跳过实际的数据部分，只处理文件和文件夹的名字和类型信息。

三、开发环境/工具，以及如何编译/运行项目

1、开发环境：JDK20

2、开发工具：IntelliJ IDEA

3、编译/运行项目：

- 编译: 打开命令行工具，导航到项目的根目录。使用 javac 命令来编译.java 文件。例如:javac allpackage/*.java ，这会编译 allpackage 中的所有文件。
- 运行: 使用 java 命令来运行编译后的程序。确保你在包含 allpackage 的目录中运行它，例如: java allpackage.Main ，接着会提示“请输入命令（huff/unhuff/preview/exit）及相应的路径：”，以参数的形式指定输入输出，不断等待用户的新的指令，并显示压缩时间、压缩率等相关信息。

四、性能测试结果

1、单个文件

（1）huffman

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase01EmptyFile\empty.txt	0	0.02	\	0.020	0
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase02NormalSingleFile\1.txt	1938.49	1086.70	56.06%	0.220	0.063
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase02NormalSingleFile\2.pdb	33.70	15.80	46.88%	0.031	0.015
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase02NormalSingleFile\20.mov	1647.65	1566.12	95.05%	0.196	0.140
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\1.jpg	20261.96	20210.47	99.75%	2.153	1.586
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\2.csv	431821.02	270798.66	62.71%	22.407	14.441
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase08Speed\1.csv	628332.07	402067.12	63.99%	26.154	10.383
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase09Ratio\1.csv	431417.58	270562.16	62.71%	20.533	11.628

对于其他压缩工具，小文件速度过快无法测量，因此只选择了大文件；

（2）MacOS 自带 Zip 压缩软件（归档实用工具）

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\1.jpg	20261.96	20226.66	99.83%	1.310	0.86
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\2.csv	431417.58	101217.30	23.46%	10.890	2.23
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase08Speed\1.csv	628332.07	73480.14	11.69%	9.050	2.54
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase09Ratio\1.csv	431417.58	101217.30	23.46%	10.890	2.23

（3）7-Zip

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\1.jpg	20261.96	20235.01	99.87%	1.55	1.05
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\2.csv	431417.58	68785.02	15.94%	29.29	1.26
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase08Speed\1.csv	628332.07	54325.98	8.65%	29.44	1.46
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase09Ratio\1.csv	431417.58	66736.00	15.47%	28.97	1.23

（4）Windows 自带压缩软件（Zip）

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\1.jpg	20261.96	20227.00	99.87%	1.03	0.98
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase03XLargeSingleFile\2.csv	431417.58	106098.00	24.59%	10.94	3.52
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase08Speed\1.csv	628332.07	74550.00	11.86%	9.94	3.67
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase09Ratio\1.csv	431417.58	103720.00	24.04%	11.25	3.53

2、文件夹

（1）Huffman

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase4EmptyFolder	0	0.04	\	0.032	0.018
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase5NomalFolder	5806.10	4227.86	72.82%	0.399	0.287
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase06SubFolders	438003.72	461380.81	105.34%	35.992	15.085
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase07XlargeSubFolders	1074189.58	681998.08	63.49%	41.796	21.061

对于其他压缩工具，小文件夹速度过快无法策略，因此只选择了大文件夹；

（2）MacOS 自带压缩软件（归档实用工具）

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase06SubFolders	438003.72	434626.71	99.23%	16.160	3.88
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase07XlargeSubFolders	1074189.58	178189.43	16.59%	20.160	3.05

（3）7-Zip

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase06SubFolders	438003.72	428881.02	97.92%	14.96	6.19
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase07XlargeSubFolders	1074189.58	122742.01	11.43%	50.95	1.78

（4）Windows 自带压缩软件（Zip）

testcase	Original (KB)	Compressed (KB)	ratio	Compress time (s)	Decompress time(s)
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase06SubFolders	438003.72	432433.00	98.73%	20.45	70.45
C:\Users\86180\Desktop\datastructure\PJ\testcases\testcases\testcase07XlargeSubFolders	1074189.58	181731.00	16.92%	20.04	8.87

3、对比分析：

- 压缩率：Huffman 编码在压缩率方面表现一般，平均只能达到 60%左右，可能是因为 Huffman 算法与其他工具相比的性能更差。对于已高度压缩的文件（如 JPG 图片），其压缩率与其他工具相似，因为这些文件类型已经通过其他算法进行了优化压缩。
- 压缩时间：Huffman 编码在处理非常大的文件时的压缩时间相对较长，尤其是与 MacOS Zip 和 Windows Zip 相比。这可能是由于 Huffman 算法在处理大文件时的计算复杂性所致。
- 解压时间：Huffman 编码的解压时间相对较长，这可能是由于算法在解压时需要重建整个哈夫曼树和解码过程的复杂性。而 MacOS Zip 和 Windows Zip 的解压速度最快。
- 其他工具性能：7-Zip 使用的 LZMA 算法在处理大型文件时非常有效，压缩率最低。MacOS Zip 和 Windows Zip 在小文件的快速压缩和解压方面有优势，压缩/解压的速度最快。

总结来说，Huffman 编码适合于需要较高压缩率的场景，特别是在文件内容具有高冗余度时，其压缩率、压缩/解压速度都比较差。对于已经高度压缩的文件格式，Huffman 编码可能不会提供显著的压缩效益，这可能是因为压缩算法的不同导致的。另一方面，对于大文件和包含多种类型文件的文件夹，Huffman 编码在压缩和解压时间方面的性能可能比较差。

五、遇到的问题与解决方案

1、异常

（1）FileNotFoundException

- 问题: 在指定路径找不到文件时出现。
- 解决方案: 确保提供的路径是正确的，并且程序有足够的权限访问该文件，需要用特定的语句提高权限。

（2）EOFException 和 IOException

- 问题: 在数据流处理过程中遇到这些异常，通常是由于文件格式不正确或读取数据时的逻辑错误。
- 解决方案: 对文件格式进行严格检查，并调整数据读取逻辑。增加判断条件来自己输出错误提示，以让用户了解输入的正确格式。

（3）UTFDataFormatException

- 问题: 读取格式不正确的 UTF 字符串时遇到。
- 解决方案: 确保数据的正确性和合法性，可能需要对数据编码或格式进行修改。压缩和解压时写入和读取的数据顺序及大小应该保持完全一致。

2、文件夹递归压缩

- 问题: 在实现文件夹及其子文件夹的递归压缩时遇到困难。
- 解决方案: 使用递归方法来遍历文件夹结构，并正确维护相对路径。并且需要将每个文件的长度存进去，以判断单个文件的结束。

3、验证压缩包来源，以及是文件还是文件夹

- 问题: 确保只解压由该工具创建的压缩文件。
- 解决方案: 在压缩文件中写入特定的标识符（magic number），以识别文件来源，并判断是文件还是文件夹。

4、文件预览功能

- 问题: 实现不解压的情况下预览压缩包内文件结构的功能时无法正确还原结构。
- 解决方案: 读取文件头信息，利用栈和递归方法来渲染树形结构，不用栈的方式的话则会一直向深入，无法判断文件夹的结束。

5、内存消耗过大

- 问题：在压缩或解压文件时，如果一次性将整个文件读取到内存中，尤其是对于大型文件，可能会导致内存消耗过大，从而引发性能问题甚至程序崩溃。
- 解决方案：为了防止内存消耗过大，应采取分块读取文件的策略。具体来说，在压缩或解压时，应该使用缓冲流（如 BufferedInputStream 和 BufferedOutputStream）来处理文件。这种方法可以每次仅处理文件的一小部分，而不是整个文件。例如，在压缩过程中，可以使用固定大小的缓冲区（如 1024 字节），每次从文件中读取这么多字节，然后对这些字节进行处理（如应用哈夫曼编码）。在处理完这些字节后，再继续读取下一个数据块，直到文件结束。这样可以有效控制内存的使用，防止因大文件处理而导致的内存问题。

6、文件夹压缩与文件压缩的区别

在哈夫曼编码的单文件压缩中，不需要单独记录最后一个字节的有效位数，因为哈夫曼编码本身是一种前缀码，它保证任何一个字符的编码都不会是另一个字符编码的前缀。这意味着，即使最后一个字节没有完全填满，解码器也能正确地识别出所有的字符编码。

当解压缩时，解码器会顺着数据流逐个解析编码，直到达到数据流的末尾。因为哈夫曼

编码的这种唯一性，最后一个字节即使不完整（不足 8 位），也不会引起歧义，因为没有任何有效的编码会以这个“不完整”的编码作为前缀。

在文件夹压缩的情况下，可能需要更复杂的数据结构来处理多个文件和它们之间的关系。这时候，记录额外的信息（如最后一个字节的有效位数）可能是为了确保在解压缩时可以正确地界定文件的边界，尤其是在文件数据直接连续存储的情况下。这样可以确保解压缩过程知道每个文件结束的确切位置，从而正确地处理后续文件。

简而言之，在单文件压缩中，由于哈夫曼编码的特性，最后一个字节的不完整性不会引起歧义，因此不需要单独记录有效位数。而在文件夹压缩中，可能由于处理多文件和文件结构的需要，记录额外信息如最后一个字节的有效位数可能是为了正确处理文件边界。

最后感谢开发过程中两位助教老师辛苦解答我的问题和疑惑！