PJ 开发文档

22302010011 黄宸一

2023 年 12 月 10 日

1 代码结构概要

类设置

我设置了 FileNode、HuffmanNode、HuffmanTree、Operation 四个类。

其中:FileNode 类用来存储文件夹的结构与所带信息 (如文件的 Huffman 树);HuffmanNode 与 HuffmanTree 类用来构建与存储对于单一文件的 Huffman 树结构；抽象类 Operation 用来处理程序运行当中的一些命令 (压缩、解压等)，检测命令格式。

Operation 下有子类 OperationQuit(处理退出) 与 OperationFile(处理文件相关命令，文件覆盖等)，OperationFile 下再细分子类 OperationZip、OperationUnzip、OperationShow。

结构设置/正常运行过程

在程序开始时，反复读取用户输入的一行命令到 String 数组 inputs 中，根据 inputs[0] 即命令名，新建相对应的 Operation 子类 op，调用 op.run() 进入命令执行部分。

在每一个 Operation 中，有共通的 checkVariableNum() 函数用来检查命令中参数数量是否正确，且有抽象方法 checkCommand() 与 run() 需要被重写对于每个不同的命令不同的格式检查与运行。

OperationQuit 在当用户输入“quit”或“q”时被调用，在 run() 中检测参数数量，返回 true 到 Boolean 变量 ifQuit 中，退出 while 循环，退出程序 (其他命令类运行时都返回 false)。

其他情况均为对于文件进行处理的 OperationFile 类，其中有共通的检查格式函数 checkCommand() 与检查是否需要进行文件覆盖的函数 checkCover()，且有抽象方法 printInformation() 需要被重写对于不同命令不同的操作信息输出。

OperationZip 在当用户输入命令名为“zip”或“z”时被调用，在 run() 中将被压缩文件 sourceFile 与压缩目标文件 destFile 赋为命令中的第二个与第一个参数，调用上述函数方法检查格式正误，处理覆盖之后对于 FileNode 类根文件节点变量，生成文件夹树，生成每个文件的 Huffman 树，将此根节点写入目标文件后调用 FileNode 类中真正的压缩函数，最后输出压缩用时、压缩率等，完成压缩。

OperationUnzip 在当用户输入命令名为“unzip”或“u”时被调用，在 run() 中将被解压文件 sourceFile 赋为命令中的参数，调用上述函数方法检查格式正误后从 sourceFile 中读取出 destFile，处理覆盖之后调用 FileNode 类中真正的解压函数，最后输出解压用时等，完成解压。

OperationShow 在当用户输入命令名为“show”或“s”时被调用，在 run() 中将被检查文件 sourceFile 赋为命令中的参数，调用上述函数方法检查格式正误后从 sourceFile 中读取出 rootFileNode，调用 FileNode 类中的 showFiles 函数输出文件夹结构即可。

以上为代码大致结构的说明，更多细节实现在下面“各需求实现思路”模块进一步说明。

2 各需求实现思路

核心需求

文件与文件夹的压缩与解压

整体压缩解压指令的接收输出等上面已做过说明,此处着重简述 FileNode.zipFile() 与 FileNode.unzipFile() 内部及相关方法的实现。

压缩：

首先读取到被压缩文件，将其储存在 FileNode 类变量 rootFileNode 中作为根节点。调用 generateFileTree()，通过 java 中的 File 类方法遍历所有的文件或文件夹进行处理，若处理到 file 为文件夹，则将其子文件或文件夹对应节点存在 ArrayList<FileNode> sonNodeList 中，若 file 为文件，则将该文件的大小 (长度) 存储在 fileLength 中，这样生成文件节点的树。

接着调用 generateHuffmanTree()，对于每个 file 为文件的节点，遍历该文件生成该文件对应的 Huffman 树 (空文件也有 Huffman 树，不过根节点为空)。Huffman 树的生成方法与中期不变，文件中所有出现过的字符都有其对应的节点 HuffmanNode，随机选择文件 1000 个位置来更新权重，大致代表各字符的出现频率，最后根据生成规则进行树的建构即可。

其中特别注意，这样生成的文件节点树中的每一个节点，可能是文件也可能是文件夹，通过 FileNode 类可以统一实现就不用麻烦地分类写代码。关于文件夹与文件的区分以及是否为空文件夹空文件：判断 file 是否为文件夹与 fileNode 的 Huffman 树是否为空等价，而不是和 fileNode 的子节点列是否为空等价；判断 file 是否为空文件与 file 的 HuffmanTree 的根节点是否为空等价，如下表：

	文件夹	空文件夹	文件	空文件
sonNodeList == null	false	false	false	false
sonNodeList.isEmpty()	false	true	true	true
huffmanTree == null	true	true	false	false
huffmanTree.getRoot() == null	-	-	false	true

正式开始向压缩文件中写入数据。首先将根文件节点 rootFileNode 通过 ObjectOutputStream 写入文件，这样存储下整个文件的结构与大小，之后对 Huffman 树中的每个叶节点生成对应的 Huffman 编码存在 HashMap 类变量 nodeList 中，这样就可以直接在该 list 中寻找而不用每次都遍历整棵树，提高效率。

最后以根后序的顺序依次写入各文件压缩后的数据。对于每个单个文件，遍历文件，将读到的字符依次转换对应为 Huffman 码连成 01 串，每 8 位转换为一个 byte 通过 BufferedOutputStream 写入文件直到该被压缩文件结束，最后不满 8 位则补满 0，处理到下一个文件从一个新的 byte 开始存储即可。

解压：

由上述压缩过程可知，Huffman 文件的整体结构为，文件开头储存着 rootFileNode 这一对象，代表整个文件的结构大小与各文件的 Huffman 树，这一对象后的所有数据都是压缩后的 01 串转 byte 数据。

则解压时，首先读出 rootFileNode，根据文件结构按根后序顺序遍历：遍历到文件夹则创建文件夹 (无需考虑是否为空)，进入其子节点；遍历到文件则判断是否为空文件，若是则直接创建该文件后继续遍历，是有内容文件，则将文件中的 byte 依次转 01 串，读到 0 则令 nodeNow 赋为其左子节点，读到 1 则令 nodeNow 赋为其右子节点，若 nodeNow 为叶节点则则将此节点 (此 Huffman 编码) 对应的原字符写入文件，nodeNow 回归 root，直到正在写入的文件长度与之前记录在 fileLength 中的原文件长度相同则结束该单文件解压，读取下一个 byte，解压下一个文件。如此反复即可完成解压。

性能

在我的笔记本且插电情况下，对于所给 testcases 这一 3.5G 文件夹的压缩时间约为 4.2min，解压时间约为 2.5min，压缩率约为 71%。其他更多不同文件大小等相关数据详见下面“性能测试”模块。

其他需求

用户交互

已实现以参数的形式指定输入输出，“zip targetFile.huffman sourceFile”表示将 sourceFile 压缩为 targetFile.huffman，“unzip sourceFile.huffman”表示解压 sourceFile.huffman，“show sourceFile.huffman”表示预览 sourceFile.huffman 中的文件结构，“quit”表示退出程序。(仅用首字母输入命令也可行)

在每次操作完毕后都会显示该操作的用时，这是由在操作前后获取当前时间相减实现的。

压缩完毕后会显示压缩率，这是由通过遍历原文件 (夹) 结构获取该文件总大小 (长度)，与压缩后文件大小相除得到的，压缩率越低说明压缩结果越小。

检验压缩包来源 & 输入命令错误反馈

已实现在尝试解压一个奇怪的，不是由我们的压缩工具创建的文件时，在终端显示“你的输入中含有我无法处理的 xxx 文件”。

另外，也实现了在对应情况下输出错误提示“你的输入中含有无法找到的文件 xxx”“你的输入中含有错误的变量数”。并且在以上这些提示后都会显示“正确的输入格式为：xxx，请重新输入”。

这些提示是由检测到对应错误情况后抛出自定义异常类 WrongInputFormatException(message, type)，在 main 函数的 while 循环中获取并根据错误信息 message 与操作类型 type 输出相应提示来实现。

文件覆盖问题

在“代码结构概要”模块中已进行了部分说明，Boolean 类型的 checkCover() 函数会检查 destFile 是否已经存在，并做出相应的响应，返回是否需要进行真正的操作。

压缩包预览

在“代码结构概要”模块中已进行了部分说明，下面简要说明 rootFileNode.showFiles() 内部的实现。该函数主体是由 isLast 这一 ArrayList<Boolean> 来代表每一级文件 (夹) 是否已经是上一级文件夹中的最后一个，以此来决定此文件名前的前缀应该如何输出，在遍历文件结构时更新 isLast 数组并输出即可。效果如下图所示：

```
z Folder.huffman Folder
当前压缩进度: 100.0%
压缩已结束
压缩用时: 1.99s
压缩率: 46%
s Folder.huffman
Folder
├── file5.txt
├── SubFolder1
│   ├── file2.txt
│   ├── file3.txt
│   └── SubFolder3
│       └── file1.txt
└── SubFolder2
    └── file4.txt
预览已生成
预览用时: 0.02s
```

其他实现

实时压缩进度

我额外实现了在压缩、解压过程中显示当前解压进度百分比，这是由设置一个已读取数据量 progressAll 与总数据量 (文件文件夹总长度) 相比简单实现。

3 开发环境工具

IDE：VS Code version 1.84

编程语言：Java

平台：Windows 11

4 性能测试

测试用例	初始大小	压缩后大小	压缩率	压缩时间	解压时间
EmptyFile	0B	438B	-	0.00s	0.00s
NormalSingleFile(9.htm)	495KB	355KB	71%	0.15s	0.13s
XLargeSingleFile(2.csv)	421MB	268MB	63%	30.32s	19.04s
EmptyFolder	0B	272B	-	0.00s	0.00s
NomalFolder(2)	4.03MB	2.79MB	69%	1.28s	0.33s
SubFolders(2)	7.43MB	5.58MB	75%	0.65s	0.47s
XlargeSubFolders(3)	1.01GB	674MB	64%	64.81s	43.76s
Speed(1.csv)	613MB	399MB	65%	38.33s	25.32s
Ratio(1.csv)	421MB	269MB	64%	26.88s	18.69s

5 其他压缩工具比较

统一压缩整个 testcases 文件夹，数据如下表：

压缩工具	压缩率	压缩时间	解压时间
Huffman 压缩工具	73.0%	4min15s	2min30s
Win11 自带压缩工具	27.4%	1min15s	1min58s
360zip	25.9%	4min	16s

原因分析：

可以看到我的压缩工具的压缩率更高，压缩后文件更大，且压缩与解压时间都比其他工具长很多。

这些主流压缩工具的底层相比于我这一简单压缩可能采取了更为高效的 LZ77 算法与哈夫曼编码相结合的 DEFLATE 等算法，因此能够做到生成更小的压缩文件；且在算法上相比于我这一简陋版本更为优化，时间复杂度更低，也可能调用了多线程进行压缩，实现大幅提升程序运行速度，降低时间。

6 开发过程中的问题与解决

1. 如何将文件与文件夹统一在一起处理。单个文件压缩解压完成后开始思考这个问题，解决办法如上述说明，使用统一的 FileNode 类，不再说明。

2. 压缩内含多文件的文件夹时，如何判定或设置两个不同文件间的分界线，使解压时互不干扰。在最初但文件实现时我采用在文件最后补反 (即如最后剩余 110 则补为 11011111) 来判定处理到哪一位时文件结束，但这种方法不适用于多个文件相连。之后尝试想在文件间插入某个特殊节点来判定，但由于生成的压缩文件中的 byte 可以取遍所有-128127，所以完全没法区分。最后采用预存储每个文件长度的方法来实现。

3. 如何大幅减少压缩时间与解压时间。除了参考文档中提到的采用缓冲区读写，我在代码修改过程中相比于中期还用到了 (1) 直接使用 int 类型 + 位运算取各位 01 来替代原先的使用字符串 + 调用 Math.Pow 这一更耗时的操作 (2) 使用 HashMap 与 PriorityQueue 代替 Arraylist 实现相同效果，这样效率有显著提升。

4. 如何大幅减少压缩率。尝试过使用多次压缩来进一步减小文件大小，但由于第一次压缩后的文件中的内容各字符频率相近，Huffman 树就没有显著优势了，放弃这一想法。不过现在离主流压缩工具的压缩程度还有很大差距。