









数据结构PJ说明文档

22302010019 陈星宇





















一、代码结构概要说明

1.包结构

该项目的包结构如下：

- ▼  src
 - >  com.huffman
 - >  com.huffman.bitstream
 - >  com.huffman.compressed_data
 - >  com.huffman.helper
 - >  com.huffman.method
 - >  com.huffman.myexception
 - >  com.huffman.treenode

2.类结构

- ▼  src
 - ▼  com.huffman
 - >  Main.java
 - ▼  com.huffman.bitstream
 - >  BitInputStream.java
 - >  BitOutputStream.java
 - ▼  com.huffman.compressed_data
 - >  HuffmanFileData.java
 - ▼  com.huffman.helper
 - >  FileHelper.java
 - >  FolderHelper.java
 - >  MyReader_Writer.java
 - ▼  com.huffman.method
 - >  HuffmanDecode.java
 - >  HuffmanEncode.java
 - ▼  com.huffman.myexception
 - >  UnknownObjectTypeException.java
 - ▼  com.huffman.treenode
 - >  FolderTreeNode.java
 - >  HuffmanTreeNode.java

对以上各个类的功能简述如下：

Main: 主控类，获取用户希望执行的操作等基本信息

BitInputStream: 使用位运算，提供了从给定字节数组中读取位序列的基本机制

BitOutputStream: 使用位运算，用于将二进制数据流按位写入数据到字节数组

HuffmanFileData: 存储压缩后的文件信息，如文件内容、文件地址等（不包括压缩的文件数据）

FileHelper: 文件处理相关的方法集合

FolderHelper: 文件夹处理相关的方法集合

MyReader_Writer: Reader把输入文件转化为二进制文件，Writer把将ArrayList中的每个字节写入文件

HuffmanDncode: 哈夫曼解压缩的实现

HuffmanEncode: 哈夫曼压缩的实现

UnknownObjectTypeException: 一个自定义异常类（未知类型异常）

FolderTreeNode: 压缩与解压缩文件夹时，用于表示文件夹结构的树的节点

HuffmanTreeNode: 哈夫曼树节点

二、各评分项实现思路

1.文件的压缩与解压

(1) 文件压缩

文件压缩功能主要由HuffmanEncode类实现，该类大纲如下：

```
● encode(String, String) : void
■ encodeFolder(File) : void
■ buildFolderTree(File, boolean) : FolderTreeNode
■ WriteFolderData(FolderTreeNode) : void
■ encodeFile(String) : void
■ HuffmanFrequencyCounter(String) : void
✓ ■ cp : Comparator<HuffmanTreeNode>
  > new Comparator() {...}
■ createHuffmanTree() : void
■ buildHuffmanCodes(HuffmanTreeNode, String) : void
■ writeOringalDataToFile(String, String, String) : HuffmanFileData
■ isEncodingEmptyFile(String, String, ObjectOutputStream) : boolean
■ encodeMsgPrinter(long, String) : void
```

仅encode方法为外部可见，根据文件类型，其中给出两条分支——encodeFile与encodeFolder。

对其中一些值得说明的方法的实现说明如下：

createHuffmanTree: 构建哈夫曼树，使用优先队列存储各个字符与对应的频率，取出头两个元素（频率最低），合并后插回优先队列，使构建哈夫曼树的时间复杂度为 $O(N \cdot \log N)$

writeOriginalDataToFile: 第二次遍历输入文件的每个字节,转化为huffmanCode组成的01串 (01串过大时, 及时写入并销毁对象

) ,再由bitOutputStream转化为字节写入文件

encodeFile的核心代码如下, 各个方法的作用见图中注释:

```
// 统计字符频率
HuffmanFrequencyCounter(binaryFilePath);
// 构建哈夫曼树
createHuffmanTree();
// 构建哈夫曼编码
buildHuffmanCodes(huffmanTree, "");
compressedHFileData = new HuffmanFileData(originalFileFullName, huffmanTree, totalBytes, huffmanCodes);
// 若在解压文件夹, 构建完huffmanHFileData就可以返回了
if (isCompressFolder)
    return;
// 若在解压文件, 先把该对象写入文件头
oos.writeObject(compressedHFileData);
// 使用哈夫曼编码压缩文件, 把压缩数据写入压缩文件地址
writeOriginalDataToFile(binaryFilePath, compressedFilePath, originalFileFullName);
// 删除二进制文件 (过河拆桥)
FileHelper.deleteFile(binaryFilePath);
// 打印相关信息
encodeMsgPrinter(startTime, compressedFilePath);
```

注: 对文件...//test.Extension,压缩后的文件名为...//test.huffman

(2) 文件解压缩

文件解压缩功能主要由HuffmanDncode类实现, 该类大纲如下:

- **s decode(String) : void**
- **s decodeFolder() : void**
- **s createFolderStructure(FolderTreeNode, File, boolean) : void**
- **s decodeFile(String) : void**
- **s isDecodingEmptyFile() : boolean**
- **s writeCompressedDataToFile(BufferedOutputStream, long) : void**
- **s decodeMsgPrinter(long) : void**
- **s generateDecodeFilePath() : String**
- **s isCoverFile(String) : void**
- **s getFolderTree(File) : FolderTreeNode**

仅decode(解压文件)与getFolderTree (用于预览压缩文件结构) 两个方法为外部可见, 根据文件类型, 其中给出两条分支——decodeFile与decodeFolder。

对其中一些值得说明的方法的实现说明如下:

writeCompressedDataToFile: 根据哈夫曼编码, 把01串转化为字节数组, 再写入文件。其中的代码经过多次修改以提高效率, 详见第六节: 遇到的问题和解决方案; 压缩文件夹时, 在压缩文件中区分哪一片段的字节属于哪一文件, 通过的是HuffmanFileData中存储的压缩前的字节数 (还原出的字节数等于原有字节数时, 说明该文件解压缩完成), 而压缩文件中的字节写入顺序与构建文件夹树的顺序有关, 反向还原即可, 不用担心顺序问题。

generateDecodeFilePath: 压缩 (非文件夹) 文件时, 由用户输入决定解压缩文件名, 若输入“\d”表示用原文件名+“ (decode) ”解压缩

注: 对原文件...//test.Extension,解压缩后的文件名为...//*.Extension * 为用户指定的解压缩文件名

2.文件夹的压缩与解压

(1) 文件夹压缩

对文件夹的操作建立在对文件的操作之上，由于文件夹可能有多层嵌套，用递归方法遍历文件夹是一个基本想法。

以建立文件夹树结构为例：

```
1 private static void writeFolderData(FolderTreeNode node) {
2     try {
3
4         for (HuffmanFileData fileData : node.getCompressedFiles()) {
5             String originalFileFullName = fileData.getFileName();
6             String binaryFilePath = FileHelper.changeFileName(
7                 FileHelper.changeFileExtension(originalFileFullName,
8                 "bin"),
9             FileHelper.getFileNameWithoutExtension(originalFileFullName));
10
11             huffmanCodes=fileData.getHuffmanCodes();
12             writeOringalDataToFile(binaryFilePath, compressedFilePath,
13             originalFileFullName);
14             // 删除二进制文件（过河拆桥）
15             FileHelper.deleteFile(binaryFilePath);
16         }
17
18         // 递归遍历子文件夹
19         for (FolderTreeNode child : node.getChildren()) {
20             writeFolderData(child);
21         }
22     } catch (Exception e) {
23         System.err.println("解压文件夹时出错: "+e.getMessage());
24         return;
25     }
26 }
```

注：对文件夹...//test,压缩后的文件名为...//test(huffman)

(2) 文件夹解压缩

createFolderStructure类是实现文件夹解压缩的主要方法，逻辑仍为使用递归方法，根据压缩的文件夹树结构，对node下的非文件夹文件逐一还原，对node的子节点作为其子文件夹逐一创建，还原出原有结构。

```
1 private static void createFolderStructure(FolderTreeNode node, File
2     parentFolder, boolean isRoot)
3     throws IOException {
4     File currentFolder;
5     if (isRoot) {
6         rootFolder = node.getFolder().getName();
7         currentFolder = new File(parentFolder, rootFolder);
8     } else
9         currentFolder = new File(parentFolder,
10         node.getFolder().getName());
```

```

9      currentFolder.mkdirs();
10     // 还原当前文件夹中的文件
11     for (HuffmanFileData huffmanFileData : node.getCompressedFiles()) {
12
13         //完成准备工作并开始压缩
14         //.....
15         //.....
16
17     }
18
19     // 递归创建子文件夹
20     for (FolderTreeNode child : node.getChildren()) {
21         createFolderStructure(child, currentFolder, false);
22     }
23 }

```

注：对原文件夹...//test,解压缩后的文件名为...//test, 若...//test已存在, 会向用户发出覆盖警告

3.用户交互

(1) 运行方式

程序以控制台方式运行, 可以连续完成各项不同任务

用户在控制台先输入想要进行的操作, 如下所示 (以压缩为例) :

请输入操作 (压缩文件-'e', 解压文件-'d', 预览文件夹压缩包内容-'c', 退出-'q') :

e

准备执行压缩文件操作

接下来需输入待操作文件的绝对地址 (地址是否带双引号无影响) :

请输入操作 (压缩文件-'e', 解压文件-'d', 预览文件夹压缩包内容-'c', 退出-'q') :

e

准备执行压缩文件操作

请输入待压缩文件的绝对地址 (压缩文件将存放在同一个目录下) :

"D:\Code\Javacode\Data Structure
Project\TestFiles\test"

压缩/解压缩时, 若发现存在同名文件, 会要求用户决定是否覆盖已存在文件:

```
请输入待压缩文件的绝对地址 (压缩文件将存放在
同一个目录下) :
"D:\Code\Javacode\Data Structure
Project\TestFiles\test"
文件 (可能是过时的) :D:\Code\Javacode
\Data Structure Project\TestFiles
\test (huffman) 已存在!
是否覆盖此文件? (Y/N)
y
```

若选择确定覆盖，会执行压缩操作，覆盖已有文件，并在结束时给出压缩信息。

(2) 错误输入处理

用户的错误输入均可妥善处理，不会使程序异常终止，此处不——展示，仅给出部分示例：

```
请输入操作（压缩文件-'e'，解压文件-'d'，预
览文件夹压缩包内容-'c'，退出-'q'）：
```

```
wrong input
```

输入错误，请重新输入。

```
请输入操作（压缩文件-'e'，解压文件-'d'，预
览文件夹压缩包内容-'c'，退出-'q'）：
```

```
请输入解压缩文件名 (输入\d将以原文件名解压
缩)：
```

```
jpg_wrongName\
```

```
不可使用特殊字符 | < > * : ? \ / " 为
文件命名!请重新输入!
```

(3) 压缩、解压缩结束信息的展示

压缩：

```
-----
File compressed successfully.
压缩率： -0.29%
压缩时间： 0.745 秒
-----
```

解压缩：

File decompressed successfully.

解压时间： 0.455 秒

4.检验压缩包来源

若不是该程序的压缩文件，会引起StreamCorruptedException，会向用户报告错误，表示无法用此程序解压缩

```
1      try (FileInputStream fis = new FileInputStream(compressedFilePath);
2          ObjectInputStream inputStream = new ObjectInputStream(fis))
3      {
4          bitInputStream = new BitInputStream(fis);
5          Object object = inputStream.readObject();
6          if (object instanceof HuffmanFileData) { // 解压普通文件
7              isDecodingFolder = false;
8              decodeHFileData = (HuffmanFileData) object;
9              decodeFile(compressedFilePath);
10         } else if (object instanceof FolderTreeNode) { // 解压文件夹
11             isDecodingFolder = true;
12             folderTreeNode = (FolderTreeNode) object;
13             decodeFolder();
14         } else
15             throw new UnknownObjectTypeException("读取对象时出错：文件中含有
16             未知的类型");
17         } catch (FileNotFoundException e) {
18             System.err.println("文件未找到" + e.getMessage());
19             return;
20         } catch (IOException | ClassNotFoundException e) {
21             if (e instanceof StreamCorruptedException)
22                 System.err.println("解压缩过程已终止\n" + "文件：" +
23                 compressedFilePath + "不可用此程序解压!");
24             else
25                 System.err.println("读取文件对象时发生错误：" + e.getMessage());
26             return;
27         } catch (UnknownObjectTypeException e) {
28             e.printStackTrace();
29             return;
30         }
31     }
```

如下图，试图用该程序解压一个.rar文件，无法执行：

请输入操作（压缩文件-'e'，解压文件-'d'，预览文件夹压缩包内容-'c'，退出-'q'）：

d

准备执行解压文件操作

请输入待解压文件的绝对地址（解压文件将存放在同一个目录下）：

"D:\Code\Javacode\Data Structure Project\TestFiles\其他压缩文件\1.rar"

解压缩过程已终止

文件：D:\Code\Javacode\Data Structure Project\TestFiles\其他压缩文件\1.rar不可用此程序解压！

请输入操作（压缩文件-'e'，解压文件-'d'，预览文件夹压缩包内容-'c'，退出-'q'）：

当然，若把文件后缀简单改为.huffman（本程序生成的默认压缩文件后缀），也会抛出StreamCorruptedException，使本次解压缩操作终止。

5.文件覆盖问题

是否覆盖交由用户决定：

```
1 // 检测文件是否已经存在，并由用户决定是否覆盖之
2 public static boolean isCoverExistingFile(String filePath) {
3     Path path = Paths.get(filePath);
4     boolean FileExists = Files.exists(path);
5     if (FileExists) {
6         //文件已存在，输入y/Y,返回true，输入N/n，返回false（代码略）
7     }
8 }
```

6.压缩包预览

获取文件夹树根节点的方法在HuffmanDncode的getFolderTree方法中实现。对于非文件夹的压缩文件，不可预览。

FolderHelper中的printFolderTree可以接受一个FolderTreeNode node，根据其中信息打印出文件夹结构：

```
1 public static void printFolderTree(FolderTreeNode node, int depth) {
2     if(node==null)
3         return;
4     System.out.println(getDepthIndent(depth) +
5 node.getFolder().getName() + "（文件夹）");
6     for(HuffmanFileData huffmanFileData:node.getCompressedFiles())
7         // 打印该文件夹之下的所有子文件
8         System.out.println(getDepthIndent(depth + 1)
9 +FileHelper.getFileNameWithExtension(huffmanFileData.getFileName()));
10    for (FolderTreeNode child : node.getChildren())
11        // 递归调用该方法，打印子文件夹信息
12        printFolderTree(child, depth + 1);
13    }
14
15    private static String getDepthIndent(int depth) {
16        StringBuilder indent = new StringBuilder();
17        for (int i = 0; i < depth; i++)
18            indent.append("--"); // 用两个空格表示一层深度
19        return indent.toString();
20    }
```

信息显示如下：


```
请输入操作（压缩文件-'e'，解压文件-'d'，预览文件夹压缩包内容-'c'，退出-'q'）：
c
准备预览文件夹压缩包结构
请输入待预览文件的绝对地址：
"D:\Code\Javacode\Data Structure Project\TestFiles\test(huffman)"
test（文件夹）
--1.jpg
--nest0_1.txt
--nest0_2.txt
--nest0_3.txt
--nest1（文件夹）
----nest1.txt
----nest2（文件夹）
-----empty.txt
-----nest1_2.txt
--nest3（文件夹）
----nest3_0.txt
----nest4（文件夹）
-----nest4.txt
```

三、开发环境以及运行项目方法的说明

1.开发环境

1.1 Java Development Kit (JDK)

- 版本：

```
java version "11.0.6" 2020-01-14 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.6+8-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.6+8-LTS, mixed mode)
```

1.2 集成开发环境 (IDE)

- IDE: Eclipse
- IDE版本: 2022-12 (4.26.0)

2.运行项目方法的说明

详见二-3，用户交互一节。

四、性能测试结果

小文件：一个451 Kb的.txt文件（txt内为一段英文文本），压缩时间0.095s，压缩率45.29%；解压缩时间0.026s

中等大小文件：一个86.6 MB的视频，压缩时间8.662s，压缩率-0.01%（负压缩率表示压缩文件更大了）；解压缩时间4.812s

大文件：一个3.50 GB的文件夹（PDF中给出的testcases），压缩时间 3分26秒08，压缩率30.57%

五、与其他压缩工具的压缩率和压缩时间比较

对于一个大小为3.50 GB(3,769,450,636 字节)的文件夹（PDF中给出的testcases）：

用时		压缩后文件大小
----	--	---------

本程序：	3分26秒08	2.43 GB (2,619,036,369 字节)
------	---------	----------------------------

zip: 2分33秒22 | 953 MB (999,950,235 字节)

WinRAR: 1分44秒58 | 815 MB (855,562,785 字节)

分析: zip对LZ压缩(“滑动窗口压缩”)后的结果通过huffman方法进行二次压缩,压缩的文件大小会显著小于本程序单独用huffman压缩的结果;经过多次试验证明,WinRAR的RAR格式一般要比WinZIP的ZIP格式高出10%~30%的压缩率,尤其是它还提供了可选择的、针对多媒体数据的压缩算法——对WAV、BMP声音及图像文件可以用独特的多媒体压缩算法大大提高压缩率,故压缩后文件大小比zip更小。(作者Eugene Roshal有条件的公开了WinRAR解码程序的源代码,但是编码程序仍然是私有的,压缩算法未知)

六、遇到的问题和解决方案

1.哈夫曼解压缩耗时明显不为线性

删除前缀的操作 `String_0_1.delete(0, index)` 的时间复杂度实际上是 $O(N)$ 。所以下列代码的时间复杂度是 $O(N^2)$, 在文件变大时运行速度显著下降。

```
1 private static Byte[] String_0_1_ToByteArray(StringBuilder String_0_1) {
2     int index = 0;
3     StringBuilder currentCode = new StringBuilder();
4     Byte[] myByteArray = new Byte[decodeHFileData.count];
5     int next = 0;
6     while (String_0_1.length() > 0) { // &&index<String_0_1.length()
7         // 逐步截取01串
8         currentCode.append(String_0_1.charAt(index++));
9         // 查看当前截取的部分是否匹配哈夫曼编码表
10        Byte decodedByte = huffmanCodes.get(currentCode.toString());
11
12        // 如果匹配成功,将对应的字节值存储到字节数组中
13        if (decodedByte != null) {
14            myByteArray[next++] = decodedByte;
15            String_0_1.delete(0, index);
16            index = 0;
17            currentCode.setLength(0); // 清空当前截取的部分
18        }
19    }
20    return myByteArray;
21 }
```

第一次修改: 修改后方法的时间复杂度为 $O(N)$, 用移动截取子串(时间复杂度 $O(1)$) 代替了删除操作(时间复杂度 $O(N)$), 大大减少了对大文件的解压缩时间

```
1 //时间复杂度为O(N)
2 private static Byte[] String_0_1_ToByteArray(StringBuilder String_0_1) {
3     int index = 0;
4     Byte[] myByteArray = new
5     Byte[decodeHFileData.getCompressedDataLength()];
6     int subLength = 1;
7     int next = 0;
8     int subStart = 0;
9     while (String_0_1.length() != subStart) {
10        // 逐步截取01串,查看当前截取的部分是否匹配哈夫曼编码表
```

```

10         Byte decodedByte =
huffmanCodes.get(String_0_1.substring(subStart, subStart + (subLength++)));
11         index++;
12         // 如果匹配成功，将对应的字节值存储到字节数组中
13         if (decodedByte != null) {
14             subLength = 1; // 子串长度重置为1
15             subStart = index; // 从新位置开始截取
16             myByteArray[next++] = decodedByte;
17         }
18     }
19     return myByteArray;

```

第二次修改：在压缩文件中不再存储哈夫曼编码表（HashMap），而是直接存储哈夫曼树，尽管这样在空间上的开销略有增加，但是在解码时对速度有显著提升，遇到1向右走，遇到0向左走，减少了用HashMap存储时大量的不匹配的时间浪费。这是一个常数级优化。

```

1 private static Byte[] String_0_1_ToByteArray(StringBuilder String_0_1) {
2     int index = 0;
3     Byte[] myByteArray = new
Byte[decodeHFileData.getCompressedDataLength()];
4     int next = 0;
5     HuffmanTreeNode nowHuffmanTreeNode=huffmanTree;
6     while (index<String_0_1.length()) {
7
8         // 遇到1向右走，遇到0向左走
9         if(String_0_1.charAt(index++)=='1')
10             nowHuffmanTreeNode=nowHuffmanTreeNode.getright();
11         else
12             nowHuffmanTreeNode=nowHuffmanTreeNode.getleft();
13
14         // 如果是叶节点，将对应的字节值存储到字节数组中
15         if (nowHuffmanTreeNode.isLeaf()) {
16             myByteArray[next++] = nowHuffmanTreeNode.getdata();
17             nowHuffmanTreeNode=huffmanTree;
18         }
19     }
20     return myByteArray;
21 }

```

第三次修改：不再把整个压缩文件还原出的01串全部保存在 `StringBuilder string_0_1` 中，分段读取 bit、写入 byte。详见OOM问题的解决。

```

1 private static void writeCompressedDataToFile(BufferedOutputStream bos, long
totalBytes) throws IOException {
2     HuffmanTreeNode nowHuffmanTreeNode = huffmanTree;
3     long bytenums = 0;
4     while (true) {
5         int bit=bitInputStream.readBit();
6         // 遇到1向右走，遇到0向左走
7         if (!huffmanTree.isLeaf()) {
8             if (bit == 1)
9                 nowHuffmanTreeNode = nowHuffmanTreeNode.getright();
10             else if (bit == 0)
11                 nowHuffmanTreeNode = nowHuffmanTreeNode.getleft();

```

```

12         else if (bit == -1) // 到文件末尾
13             break;
14         else
15             System.err.println("错误的比特读入:" + bit);
16
17     }
18     // 如果是叶节点, 将对应的字节值存储到字节数组中
19     if (nowHuffmanTreeNode.isLeaf()) {
20         bytenums++;
21         bos.write(nowHuffmanTreeNode.getdata());
22         nowHuffmanTreeNode = huffmanTree;
23         // 若当前还原的字节数与原文件原有的字节数一致, 退出
24         if (bytenums == totalBytes) {
25             bitInputStream.clear();
26             break;
27         }
28     }
29 }
30 bos.flush();
31 }
32

```

2.解压缩、压缩大文件时, 出现OOM错误

出现OOM错误主要原因在于, 在一开始的版本中, 由于只考虑了将一个最终对象写入文件的方法, 压缩文件的所有数据、相关信息都是由一个对象装载(即HuffmanFileData), 在压缩数据极大时, 会出现OOM问题, 提示堆空间不足。在4天时间里, 本人花费了20小时以上的时间重构代码, 最终的解决方案是把不包含文件压缩数据的HuffmanFileData写入文件, 再用BufferOutputStream向文件中写入字节。这一方法及时地完成了对象的销毁, 不仅避免了OOM问题, 也大大提升了运行速度。