YYCompression文档梗概

1. 概括压缩/解压功能的实现原理
2. 对每个功能模块简述实现方式
3. 描述性能，附与市面常见压缩软件的对比
4. 介绍几个开发中遇到的问题

概括压缩/解压功能的实现原理

采用Huffman编码对字节再编码，为高频字节赋予更短的编码，减少空间使用

故压缩功能步骤为：统计文件中各字节频次、根据结果建立HuffmanTree、根据Huffman编码重新编码文件内容

解压功能步骤为：根据Huffman编码的对应关系反向编码压缩文件内容

对于文件夹的压缩，则先构建文件夹树(有下级文件夹数组和下级文件数组两个孩子)，先存入文件夹结构，再按遍历顺序逐个压缩文件

解压时先读入文件夹结构，再按照遍历顺序逐个解压文件

对每个功能模块简述实现方式

频率统计

Statistic接口实现统计功能，用HashMap统计文件中各字节的频率

再根据频率生成Huffman树结点，并按顺序放入优先队列

Huffman编码

编码目的：让高频字节拥有短编码，且任何字节的编码不是其他字节编码的前缀(否则在解压时将出现歧义)

Huffman编码构建二叉树，以从根节点到叶子结点的路径为编码内容，故字节位于叶子结点、编码长度为结点层数

从下向上构建HuffmanTree，先构建的节点层数更高，故应先取频率低的结点

遵循此步骤，永远将频率最低的两结点取出构成小树，并以频率之和标记树根，继续和其他结点比较(每一步都保证把频率最低的结点放在最下面，得到总长度最小的编码方式)

Tree接口规定了Tree类的功能，在TreeImpl中实现树的构建与使用

- 树的构建：按照上述原理，不断将频率最小的结点结成树，直到每个最初的节点都成为叶子
- 树的使用：从根节点爬树，向左记0向右记1，爬到叶子结点时，即得到了叶子结点对应的编码

压缩与解压

压缩与解压在YYCompress接口中规定功能，在YYCompressImpl中实现

压缩与解压设计了类似结构：

- 一个分发函数：判断压缩or解压目标是文件还是文件夹，并分发给对应函数
- 处理单个文件：对压缩，是将单个文件压缩到目标地址，解压则是读出压缩文件中的下一个文件放入目标地址
- 处理文件夹：本身并不进行压缩，负责文件夹树的生成/读取，通过遍历文件夹树的方式逐个调用处理单个文件的函数

压缩实现

将HuffmanTree转化成HashMap，避免重复爬树

由于在字节最均匀且挤满256个的情况下，Huffman编码将白费力气，将8位字节全部再编码成8位字节。除此之外，压缩文件的内容部分都将小于原文件

故能确定压缩文件≤借助FileInputStream.available获取的原文件长度，可以按此最大长度创建byte数组并记录(不满8位后方补0)

逐个查表填入后，再复制精确大小的数组。由此避免了ArrayList等数据结构在扩容过程中反复复制的损失

所有写入采用了ObjectOutputStream，对不定长的文件夹树和HuffmanTree，但对数组其实有空间和速率的损失。(空间上，额外记录了数组的非内容信息。速度上，额外增加了最后一次数组复制的工作)

解压实现

从文件中读出HuffmanTree和编码后内容，将byte转为String(这里用StringBuilder、用charAt而非subString，否则性能浪费严重)

解码时便不适合采用HashMap，不定长的编码若用HashMap查询对应定长编码，需要反复操作字符串。且Hash查询比起爬树，也有常量倍数的时间浪费

故解码时逐个根据char爬树，爬到叶子时写入内容，直到文件大小恢复(需要避免不满8位的补0被误识别为Huffman编码)

描述性能

测试采用了两台电脑：A为台式机，B为轻薄本

WinRAR和HaoZip的测试只在A机上运行

A机配置：Core i7-9700K 3.60GHz;16G;500G固态

测试样例	大小 MB	压缩速度 MB/s	WinRAR	HaoZip	压缩比%	WinRAR	HaoZip	解压速度 MB/s	WinRAR	HaoZip
testcases	3594.83	21.44	19.97	111.4	69.45	22	27	14.77	299.50	114.5
testcase05NomalFolder	5.67	17.55	太快看不清	太快看不清	75.72	36	38	10.92	太快看不清	太快看不清
testcase07XlargeSubFolders	1049.01	22.97	21.85	99.9	63.52	11	17	16.55	349.67	271.2
testcase08Speed	613.61	23.27	30.68	73.0	63.99	7	12	14.09	大概500	458.1
testcase09Ratio	421.31	21.12	16.20	40.5	62.71	16	23	16.61	400不到	288.5

B机配置：Core i7-8550U 1.80GHz;16G;512G固态

测试样例	大小	压缩速度	压缩比	解压速度
testcases	3594.83	11.58	69.45	4.95
testcase05NomalFolder	5.67	11.62	75.72	9.51
testcase07XlargeSubFolders	1049.01	12.85	63.52	8.06
testcase08Speed	613.61	10.10	63.99	7.97
testcase09Ratio	421.31	11.91	62.71	8.74

开发中遇到的问题

数据结构的选择

由于压缩将改变文件长度，压缩后的文件长度未知，最初采用ArrayList储存压缩后的Byte

然而，文件长度的可能上限是可知的，可以在一开始就做好可能的扩容(且这个空间也会比ArrayList一步步翻倍最终占用的空间小)

于是，整个文件的压缩过程中只需要一次复制(从最大长度复制到压缩后获知的精确长度中)，如果不使用writeObject()而是逐字节写入，可以将这次复制也省去(来不及改啦！)

字符串操作的性能减损

最初将HashMap直接写入文件当做编码表，比起存入HuffmanTree可以节省大约一半的空间(4KB->2KB)

在压缩过程中，查表也比反复爬树要更节省时间

然而解压时，如果仍采用查表方式，则不得不反复使用subString，以检验缓冲区中的前几位是否在编码表中

该为存入HuffmanTree，使用charAt不断爬树的方式节省了算法中绝大多数的字符串操作

ObjectIn/OutputStream的使用

为便于储存文件夹结构，压缩中写入才用Object读写

起初的设计中，将地址作为参数，每次在函数内创建流对象，用append模式写入。解压的时候则共用流对象，防止丢失读取进度

然而，ObjectOutputStream将在构造时写入序列化流头，确保构造ObjectInputStream时不会阻塞。Java-API ObjectOutputStream构造方法

所以，如果多次new对象输出流，将导致无法顺利读取除首个写入对象以外的所有内容

对象输出流和输入流都必须做成共用同一个实例

压缩/解压大文件导致JavaFx无响应

查API后发现，JavaFx的界面是由单独的UI线程维护的，如果再此线程中执行耗时操作，UI将无响应。画面更新也发生在执行结束后，这就意味着很难从UI上提示用户等待、几乎必然造成困惑

为了在耗时操作过程中避免无响应，同时通过UI告知用户等待，就需要使用新的线程

起初尝试使用Thread类，但无法向其传参，既执行不了压缩解压、也无法更改UI界面

JavaFx团队将Thread包装成Task类，转为JavaFx设置后台运行功能。然而同样存在传参问题，引用虽不会报错，但根本引用不到想要的内容(如输出字符串为空内容)

查询JavaFx-API Task类发现，Task只能接受final类型作为参数

通过final传参，执行压缩解压的问题得到解决，但仍无法更改UI，因为Controller中的各个属性不可能以final传入并生效

进一步查询API得知，子线程想要修改UI界面，只能通过请求主线程来实现。JavaFx团队提供了Platform.runlater()函数，在里面更改UI就可以