2019-DS-project1开发文档

18302010047 戴予淳

1、项目构思

由于本PJ指定了文件的压缩算法为利用huffman编码进行的压缩，故在算法部分不存在太大问题，因byte为计算机存储的最小单位，且取值只有256种，很适合作为哈夫曼树中的叶子节点。于是压缩文件的流程可以如下描述：

```
1、将目标文件按byte读入
2、统计每个byte的出现次数，并作为哈夫曼树的初始权值
3、将出现次数大于1的byte的信息加入堆中，构建哈夫曼树，得到哈夫曼编码
4、按照哈夫曼编码将原byte数组进行转换，并拼接成新的byte数组
5、将解压需要的信息以及新的byte数组输入压缩文件，压缩结束
```

下面再考虑解压过程，解压过程的特点在于其不仅要得到压缩文件的内容部分，还需解析出其哈夫曼树部分的信息。具体来说，再压缩时要记入压缩文件的信息有：**压缩文件内容的大小，哈夫曼树的节点个数，哈夫曼树根节点编号，哈夫曼树每个节点的左右儿子编号。**解压流程描述如下：

```
1、将目标文件按byte读入
2、按照压缩时的存储方式将哈夫曼树的结构进行读出并进行建树
3、读取内容信息，按位对哈夫曼树进行匹配，即如果匹配位为1走向右儿子，否则左儿子，初始在根节点
4、当匹配到叶子节点时代表还原出一个byte，写入输入文件，匹配位回到根节点
5、重复3，4过程直到内容匹配完毕，解压结束
```

再考虑文件夹压缩与解压，这相当于是文件压缩的一个复合操作，可以利用dfs遍历该目录下所有的文件，然后将每个文件进行压缩后打包在一起。同时还需要记录下文件目录结构和文件名以便进行还原。文件夹压缩描述如下：

```
1、利用dfs获取文件目录结构和文件信息
2、利用单文件压缩算法将每个文件进行压缩，存储压缩文件
3、将文件目录结构、文件信息写入压缩文件
4、将各文件的压缩文件整合入压缩文件，压缩结束
```

相应的，文件夹解压算法描述如下：

```
1、将目标文件按byte读入
2、解析文件开头关于文件目录和文件内容的信息，并在目标路径下建立相关目录
3、读取单个文件的压缩内容，利用单文件解压算法进行解压
4、将解压结果写入相应目录下
5、对所有文件执行完毕后，解压结束
```

这里只是较为简单地描述了算法流程，实现算法流程后再将算法内核与UI进行绑定即可完成压缩软件的功能。

具体实现可以看代码。

2、实现中遇到的问题和解决方案

2.1 IO加速

自己大一的时候做课程PBL时的课题是文件IO，于是知道在Java IO流中速度`ioStream < bufferedStream < nioStream < nioMemoryStream`，在此PJ中选择的IO方式是利用`RandomAccessFile`进行文件访问，然后利用nio的特性对文件建立`FileChannel`，再利用nio中的`ByteBuffer`对channel进行读写。实际测试中速度确实快于利用`BufferedStream`进行读写的同学。

另外，根据自己电脑的配置条件适当调大缓冲区的大小，也可以进行IO的加速。

2.2 压缩加速

在经典的OOP编程中，我们可以对哈夫曼树节点建立一个类，对哈夫曼树建立一个类，利用各类指针来组织树的结构，调用各种方法进行建树。思想很好，但是略显复杂，速度也偏慢。其实此处直接用面向过程式的编程进行解决就行了：对于基本的byte，直接将其值与节点编号相对应，对于新建的节点，利用一个变量记录当前最大的节点编号，每次进行+1即可。然后进行出堆时将新节点的左右儿子分别设置为出堆的两个节点。这些操作都可以通过数组进行实现。最后将哈夫曼树信息写入文件时也只需写入这些数组文件。解压建立哈夫曼树时同理。

2.3 UI

UI选定的是Javafx+Jfoenix组件，利用SceneBuilder进行构建，使用起来比较方便，因为之前有UI的开发经验所以上手比较快。功能实现的话在UI的controller里将UI与算法内核进行绑定即可。

2.4 解压失败原因及解决方案

遇到过两次解压失败的情况。

第一次：解压出来最开始一部分结果正确，从某一个byte开始之后开始错误。

原因：我读入的缓冲区大小为100MB，即会对100MB的文件进行哈夫曼编码的转换，拼成新的byte数组写入文件，于是此时可能会留下几个bit需要和下一批凑成byte写入文件。我之前对于余下bit数的计算方式是`x=bits_num/8`，bits_num表示压缩后的bit数。但是多次读入时的公式应该为`x=(x+bits_num)/8`，要考虑上次遗留下的bit。

解决方案：更改余下bit的计算公式，保证正确写入。

第二次：解压出来的文件与源文件几乎一致，只是在末尾加入了1~2个byte的内容。

原因：考虑哈夫曼编码后的bit数不一定是8的倍数，这即意味着内容的最后一个byte的最后几位可能并非原内容的信息，但是这几位也可以在哈夫曼树上进行匹配并且匹配到叶子节点，这即造成了在文件尾有多余字符。

解决方案：记录源文件的byte数，记录在压缩文件中。解压时只解压出这么多数量的byte，剩余的信息不做处理。

3、压缩表现

压缩对象	压缩时间	解压时间	压缩率
1.txt	243ms	183ms	56.6%
2.pdq	16ms	125ms	56.7%
3.evy	12ms	51ms	127.9%
4.gz	5ms	47ms	945.5%
5.hpgl	53ms	56ms	47.8%
6.ma	8ms	51ms	58.5%
7.pdf	8ms	50ms	81.3%
8.sgml	5ms	112ms	70.7%
9.htm	14ms	127ms	69.8%
10.cgm	5ms	42ms	114.3%
11.g3f	5ms	38ms	124.1%
12.gif	2ms	36ms	197.3%
13.jpg	6ms	40ms	114.3%
14.png	2ms	37ms	284.5%
15.ps	6ms	48ms	60.2%
16.svg	2ms	43ms	84.6%
17.tif	4ms	48ms	115.4%
18.xbm	2ms	42ms	53.2%
19.msh	8ms	128ms	65.5%
20.mov	8ms	163ms	94.9%
21.mepg	7ms	50ms	100.9%
22.igs	26ms	85ms	41.0%
23.v5d	25ms	83ms	94.1%
24.wrl	4ms	48ms	66.3%
25.aiff	4ms	48ms	75.6%
26.au	4ms	117ms	93.2%
27.mp3	38ms	198ms	99.4%
28.ra	5ms	43ms	110.0%
29.wav	2ms	40ms	159.1%
30.ram	2ms	40ms	845.7%

压缩对象	压缩时间	解压时间	压缩率
31.aiff	5ms	42ms	95.4%
32.aiff	8ms	51ms	90.4%
33.aifc	14ms	58ms	88.1%
34.tsv	2ms	50ms	52.0%
35.avi	6ms	51ms	96.1%
folder1	885ms	2051ms	62.5%
folder2	2480ms	5358ms	64.4%
folder3	1060ms	2226ms	65.5%
large 1.jpg	613ms	1742ms	99.7%
large 2.csv	2906ms	6159ms	64.1%
large 3.csv	2697ms	5988ms	63.7%
emptyfile	2ms	23ms	100%
emptyfolder	29ms	2ms	100%

运行环境：Cpu: Amd 3600 Gpu: Nvidia rtx2060 Ram: 32GB

4、性能对比

主要判断依据为大文件的压缩效果。结论为：压缩速度与360压缩差不多甚至更快，压缩结果的大小大于360压缩（压缩算法比较naive的结果）

对比效果：

文件大小：116596836 byte	压缩时间	压缩率
我的压缩	6159ms	64.1%
360压缩	11000ms	25.1%