JAVA 的输入输出

姓名：陈德创　　学号：19030500217

西安电子科技大学

日期：2020 年 5 月 23 日

目录

1 小组名单

学号	姓名	工作
19030500217	陈德创	完成程序、联系例程、调试、小组讨论

2 题目

1. 实验楼练习
2. 程序设计，程序可对文件进行操作，要求实现：
   1). 针对给定（英文）文档，统计文档的段落数、单词数及每个字母出现的次数
   2). 针对一个文件夹（文件夹下全部文件及子文件夹）下全部文件，根据给定字符串，检索出全部包含该字符串的全部文件，并将结果列表。
   3). 针对给定源文件名及目的文件名（文件名以 main 函数参数方式给定），实现将源文件拷贝至目的文件。

3 练习

3.1 随机访问文件

RandomAccessFile 类可以进行文件的随机访问。感觉就像是可以控制一个光标，将光标移动到指定位置 (通过 *seek* 方法)，然后对光标处的文件内容进行操作。

3.2 FileDemo

原数组进行倒序输出。这里 $randf.seek(i * 4L)$ 找位置很重要，注意这里乘以了 $4L$，因为 *seek* 接收一个长整型为参数并且是以字节为单位的。去掉这个语句将导致 $EOFException$。

3.3 文件拷贝

Files 类是 *File* 的工具类，在 JDK7 中开始提供。而 *Path* 类是 *File* 类的替代类。想来也是，本来 *File* 类表示的就是一个虚拟路径，用 *Path* 类表示也更加贴合原意。而且 *Path* 通过静态方法 *get* 获得实例，可以更好地对路径的合法性做出检查。

Files 类的 copy 方法提供了对文件的复制，并且给出了更多的可选项，也可以说是很人性化了。默认情况下，如果目标文件已存在或者是符号链接，则复制将失败，除非源和目标是 same 文件，在这种情况下，方法完成而不复制文件。

3.4 文件移动

和文件拷贝的使用方法大同小异。可以直接使用 *move* 方法间接实现 *raname*，即保持原目录即可。

3.5 文件删除

Files 的 *delete* 方法可以是实现文件的删除。该方法声明了 *IOException* 异常。*File* 类也有成员方法 *delete*，这两者的主要差别是 *File* 的成员方法 *delete* 如果删除失败返回 *false*，否则返回 *true*；*Files* 类的静态 *delete* 如果删除成功则什么都不做，否则抛出相应异常，比如文件不存在抛出 *NoSuchFileException*。

3.6 文件属性

这就没啥好说的了，不管是文件，以后也可能遇见其他东西，这些属性能提供的 JAVA 肯定提供相应方法了，毕竟万物皆对象。更多方法可以查阅相应文档（这个不是官方的，但是是中文的）。

不过 *Arrays.stream*(*file.list*()).*forEach*(*System.out* :: *println*); 这个输出方式是真的秀。

3.7 ReadDir

遍历一个文件夹及其所有的子目录等，这个本质上就是一个搜索遍历。实验楼中给出的是 *dfs*，也更好写一些。

3.8 目录树

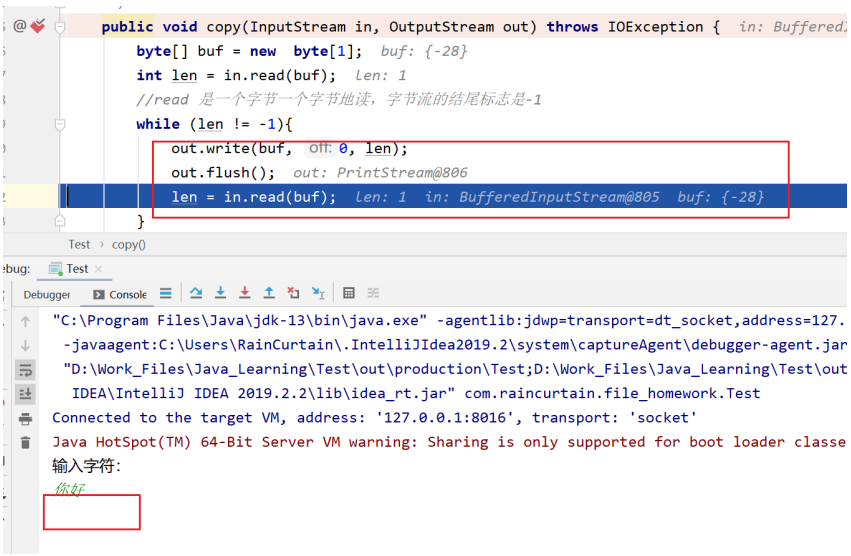
这个还是要 *dfs* 实现，用广搜的子文件和父级文件夹对应不上。在原来的遍历方法中引入 *dep* 参数参数就可以了，然后根据 *dep* 进行对齐。

3.9 字节流

将原代码改成如下：（将缓存数组长度改为 1）

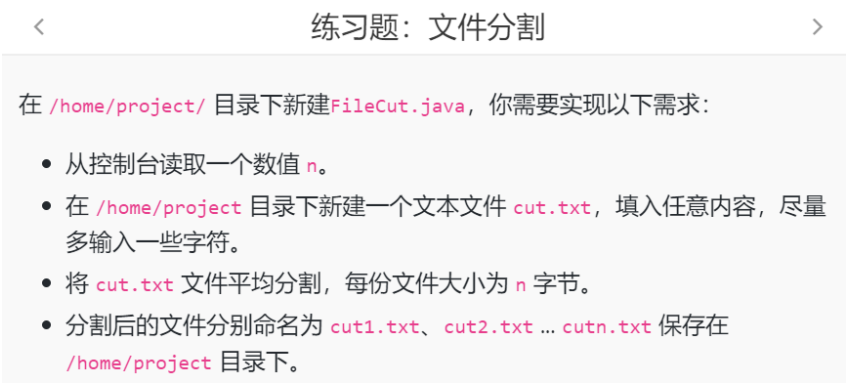
```
public void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buf = new  byte[1];
    int len = in.read(buf);
    //read 是一个字节一个字节地读，字节流的结尾标志是-1
    while (len != -1){
        out.write(buf,  off: 0, len);
        len = in.read(buf);
    }
}
```

但是在读入中文时并没有出现乱码，说明 *System.out* 的 *write* 方法是优化过的，并不会直接写出。经过我们调试，确实是这样。



3.10 文件分割

这里题目表述有误：分成 n 份，不一定每份都是 n 字节。



题目已经给出提示了，就是获取到文件的总字节数（通过 *File* 类的成员方法 *length*，将会返回字节数），然后平均一下。每次读入 $\lfloor length/n \rfloor$ 个字节，一共读入 n 或者 $n+1$ 次（这取决于是否可以整除），然后写入相应文件。

3.11 缓冲流

缓冲流是个好东西，一次读一行不好太爽，而且效率也高了。通过 *BufferedReader* 类可以对字符输入流进行修饰（对应的 *BufferedWriter* 类可以对字符输出流进行修饰，字节流也有相应的对象。其实 JAVA IO 类的类设计还是很有规则的，*Reader/Writer* 是字符，*IuputStream* 和 *OutputStream* 是字节，节点流对应源，处理流用什么套什么，很舒服。

3.12 数据流

数据流是一种很精细地对文件数据进行操作的 IO 流了吧，可以定制读入/写出各种基本类型，看起来总像是一些对内存外存把控极好的大佬们用的。

3.13 读写对象

没啥说的吧，就是被读写的类要实现 *Serizlizable* 接口，其实这个接口没有任何需要实现的方法。读对象的时候注意强制类型转换，

4 题目分析

4.1 统计段落、单词、字母次数

对于给定文件，我们有两个思路，一个是逐个读入，统计字母，然后判断是不是空格或者换行。如果空格则单词数 ++，如果是换行则段落数 ++。需要注意的是可能多个空格和换行连在一起，所以需要维护一个“preChar”，也就是上一次输入的字符，如果不是有效字符，那么单词或者段落数就不能增加。这样的好处是时间复杂度常数比较低，毕竟一次读入就完了，编程也不算麻烦。

因为有空行，所以我本想偷个懒。用 $BufferedRead$ 的 $readLine$，每次读入一行判断是不是空就可以了。然后利用 $splite$(" ") 分割统计单词个数，然后遍历每个单词的每个字母统计字母出现次数。本想偷懒，去弄巧成拙，更麻烦了。

4.2 查找字符串

要求查找一个文件夹下所有文件（包括子文件夹的文件等）中是否包含所要查找的字符。查找的字符可能出现在如下几个地方：文件名、文件内容。

首先需要找到一种方法遍历所有的子文件即文件夹中的子文件。可以用 dfs 或者 bfs，我们选用 bfs，主要是因为 dfs 开栈成本太高而且我们希望可以用一下前缀数组加快一下查找速度，而 dfs 会导致重复计算前缀数组（当然可以另开一个方法，但是这不是麻烦么）。

首先求一下目标字符串的 byte 数组的前缀数组。对于遍历到的某一文件，我们采用字节流处理，然后用 KMP 算法的在线做法来匹配 byte 数组就可以了。假设 i 为已经匹配到的 byte 数组下标，则匹配成功的条件为 $i == byte[].length$。

4.3 文件拷贝

这个就简单了，这个实验楼的实验中也有涉及到。开两个字节流，循环一个读入一个读出就可以了。主要是要判断一下拷贝的条件。即若源文件名与目的文件名相同、目的文件已存在、源文件不存在。不执行拷贝。

4.4 人机交互

这次人机交互有点特别，使用命令行参数进行交互的。三个操作命令，switch 选择一下，没有命令就输出提示信息。

5 程序实现

5.1 统计段落、单词、字母次数

设计静态方法 *Map statistics(String path)*，用 Map 返回结果（用整数数组完全可以）。我们用 *paraNum* 表示段落数，*wordsNum* 表示单词数，*lettersNum*[] 表示字母数。其中 *letterNum*[*X* − *′A′*] 表示字母 X 出现的次数，我们不区分大小写。*while* 循环，每次利用 *BufferedRead* 每次读入一行，当读入为 *null* 时退出循环（如果是空行就 *continue*）。

利用 *splite*(" ") 分割成若干单词，更新单词个数。*foreach* 遍历每个单词，对于每个单词，可以利用字符串的 *toCharArray* 方法将字符串变为字符数组。然后依次统计就可以了。注意一下每次统计之前要先判断字符是不是字母，直接用 *lettersNum*[*char* − *′A′*] + + 的话可能导致数组溢出。

补充一下，这里判断我用的 *if*(*char* >= *′A′ and char* <= *′Z′*)，不知道为什么用 *Character.isLetter*(*char*) 会数组越界。

5.2 查找字符串

因为我们不一定对 txt 文件进行操作，所以我们匹配的方式为字节数组。当然本质上和字符数组没什么不同。所以我们要先把目标字符串转化为字节数组，好在有 *getBytes* 方法可以很方便地调用。

我们设计如下的方法：

```
1   /**
2    * 用于计算给定byte数组的前缀数组
3    * @param tars 需要计算前缀数组的byte数组
4    * @return byte数组的前缀数组
5    */
6   static private int[] getPrefix(byte[] tars);
7   /**
8    * 用于查找文件file内有无符合tars数组的字节串
9    * @param file 目标文件
10   * @param tars 目标byte串
11   * @param pi 目标byte串的前缀数组
12   * @return true当含有, false当不含有
13   */
14  static private boolean kmp(File file, final byte[] tars, final int[] pi);
15  /**
16     * 用于查找path所表示的文件夹（或文件）及其子文件等中是否含有目标字符串,
17     * 包括文件夹（文件）名字和文件内容
18     * @param path 目标文件夹（文件）
19     * @param tar 目标字符串
20     * @return null当path不合法, 找到的含有目标字符串的文件（文件夹）当其他情
           况
21     */
22  static private List<File> findString(String path, final String tar);
```

该功能的入口方法为 *findString* 方法，该方法会首先判断路径地合法性，然后调用 *getPrefix*(*tar*) 方法，获得目标字符串的字节数组的前缀数组。随后对目标文件夹进行 bfs 搜索，对于搜索到的文件夹，会将其子文件加入队列，对于搜索到的文件，会调用 *kmp* 方法来判断文件中是否还有目标字节数组。*kmp* 的内部实现就是在线版的 kmp 算法，每次读入一个字节然后进行判断。含有的条件在前面已经给出了

需要注意的是一些细节，比如先判断路径合法。对于每次操作的文件（文件夹）要先判断名字中是否含有目标字符串（*contians* 方法即可），如果含有并且是文件的话则不用进入 *kmp* 算法。

最后实现的时候，注意读入的时候是一个整型，而我们比较的对象是 byte 型，我们是想进行位比较而不是值比较，所以要对读到的整型进行强制类型转换为 byte，即去掉高位。以开始没注意，真的是崩溃。

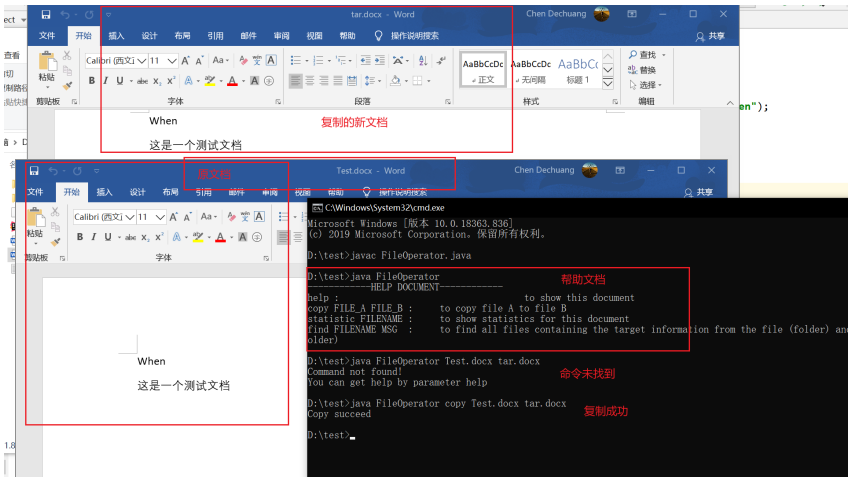
还有就是这个程序对 word 之类的还是无能为力，将 docx 解码把里面的 xml 单独复制倒是可以找到。emmmmm 可能是编码问题？（啊我死了）

5.3 文件拷贝和人机交互

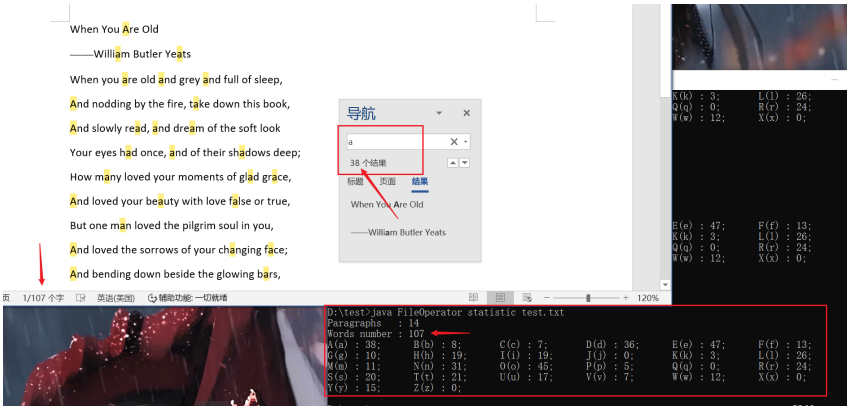
文件拷贝，很经典的练习应用，也很简单。设计一个 *copyFile* 方法，返回一个布尔值表示拷贝成功与否，内部就是循环读入写出。人机交互也不多说了，和之前的大同小异。不过这次是用命令行参数进行人机交互的，注意下参数的判断逻辑就可以了。

6 执行结果

6.1 Copy



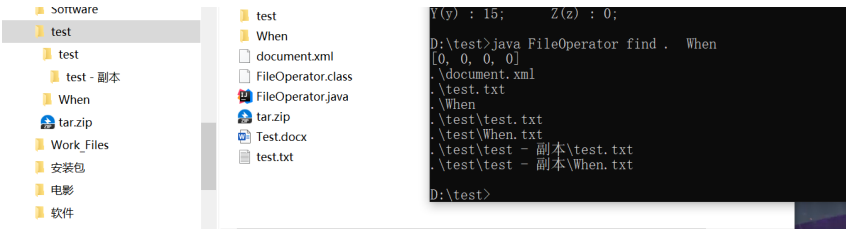
6.2 Statistic



这个效果还是不错的，素材取自 Yeats 著名的《When You Are Old》。

6.3 Find

Find 是重头戏啊。结果如上，设置了三层目录。我们搜索“When”，模拟了文件夹名字中带



有关键字、文件名字中带有关键字、文件内容中带有关键字以及多层目录。

其中 docx 文档中也是由“When”关键字的，但是搜索不出，而解压出来的 xml 可以被搜索出。

等等！这是因为被压缩了么？

7 个人总结

我原以为这回报告也可以写得少一点，现在发现也不少。我要学会写报告了，精简一下自己的语言和表达。

在异常之后我学的就没这么好了，有点虎头蛇尾的感觉。这很不好，要改正。（其实文件在之前学 C++ 的时候我就学的不好，只会个重定向能输入测试文件就行了）

行百里者半九十，不能放弃啊。