软工实践作业（二）

GitHub

PSP表格

PSP2.1	Personal Software Process Stages	预估耗时（分钟）	实际耗时（分钟）
Planning	**计划**	**65**	**68**
· Estimate	· 估计这个任务需要多少时间	65	68
Development	**开发**	**510**	**558**
· Analysis	· 需求分析 (包括学习新技术)	150	180
· Design Spec	· 生成设计文档	30	24
· Design Review	· 设计复审	20	11
· Coding Standard	· 代码规范（为目前的开发制定合适的规范）	10	5
· Design	· 具体设计	30	10
· Coding	· 具体编码	120	158
· Code Review	· 代码复审	40	26
· Test	· 测试（自我测试，修改代码，提交修改）	120	144
Reporting	**报告**	**90**	**65**
· Test Report	· 测试报告	40	18
· Size Measurement	· 计算工作量	20	14
· Postmortem & Process Improvement Plan	· 事后总结, 并提出过程改进计划	30	33
	合计	**665**	**691**

需求分析

基本功能点：

- 程序可通过**命令行**读取输入文件；

- 程序可统计文件的**字符数**，具体要求：
  - 只需要统计**Ascll码**，汉字不需考虑；
  - 空格，水平制表符，换行符，**均算字符**；
- 程序可统计文件的**单词数**，具体要求：
  - 单词[4]：至少以4个**英文字母**[1]开头，跟上**字母数字符号**[2]，单词以**分隔符**[3]分割，**不区分大小写**；
- 程序可统计文件的**有效行数**，具体要求：
  - 任何包含**非空白字符**的行，都需要统计；
- 程序可统计文件的**单词词频**，具体要求：
  - 最终只输出**频率最高的10个**；
  - 频率相同的单词，优先输出**字典序靠前**的单词；
- 按照**字典序**输出结果至文件result.txt，具体要求：
  - 输出的单词统一为**小写格式**；
  - 需按**格式**[5]输出.

非功能性需求：

- 对三个核心功能**统计字符数**、**统计单词数**、**统计最多的10个单词及其词频**进行封装；
- 使用Github进行源代码管理，**代码有进展即签入Github。**根据需求划分功能后，每做完一个功能，编译成功后，应至少commit一次；
- 至少应采用白盒测试用例设计方法来设计测试用例，并设计至少10个测试用例.

备注：

1、英文字母：A-Z，a-z；

2、字母数字符号：A-Z， a-z，0-9；

3、分割符：空格，非字母数字符号；

4、例：file123是一个单词，123file不是一个单词。file，File和FILE是同一个单词；

5、输出格式示例：

```
characters: number
words: number
lines: number
```

```
<word1>: number
<word2>: number
...
```

解题思路

看到这个题目后，我其实第一想法是用MapReduce....这也算是MapReduce的Hello World了。不过题目是在单机上测试，所以用分布式框架毫无意义（之后应该会补充基于MapReduce的WordCount）。

这次需要实现的功能其实主要是两个部分：**字词计数**和**文件读写**。下面进行具体描述：

对于**文件读写**，因为很多计数处理在读文件时可以一起完成，所以我选择将文件读取放进计数模块中。而写文件则独立出来，避免过多功能写在一起显得太臃肿。

对于核心的**计数模块**，其实字符和行数还是比较好实现的。但在**字符计数**中也碰到了一个问题，用readLine读取文件时，无法将换行符读取进来，更改成read一个一个读就没问题了。对于**单词**的读取，我一开始想直接用split进行切分，但又有些担心正则的效率。。经过测试，最后还是选择了stringTokenizer进行切分，正则用来匹配。不过官方并不推荐用stringTokenizer，，但简单切分还是蛮好用的。

关于怎么做**词频排序**，我起初想了几个方案：转换为list直接sort、建堆、BFPTR加快排。实测BFPTR加快排还是会比堆快一点的。但最终实现时，我还是用了sort，写起来干净方便。。其实也是有些地方没修好，因为很少用java写算法，所以虽然能跑起来，但中间冗余部分还是有点多，看着非常别扭，于是弃用了。很难说这样扯出来的代码性能究竟怎么样，因为时间有限，所以没有再进行对比测试，之后修复好还是得多试试。

代码规范

代码规范我用的是实验室的代码规范：阿里巴巴的码出高效，并加上了一些补充。

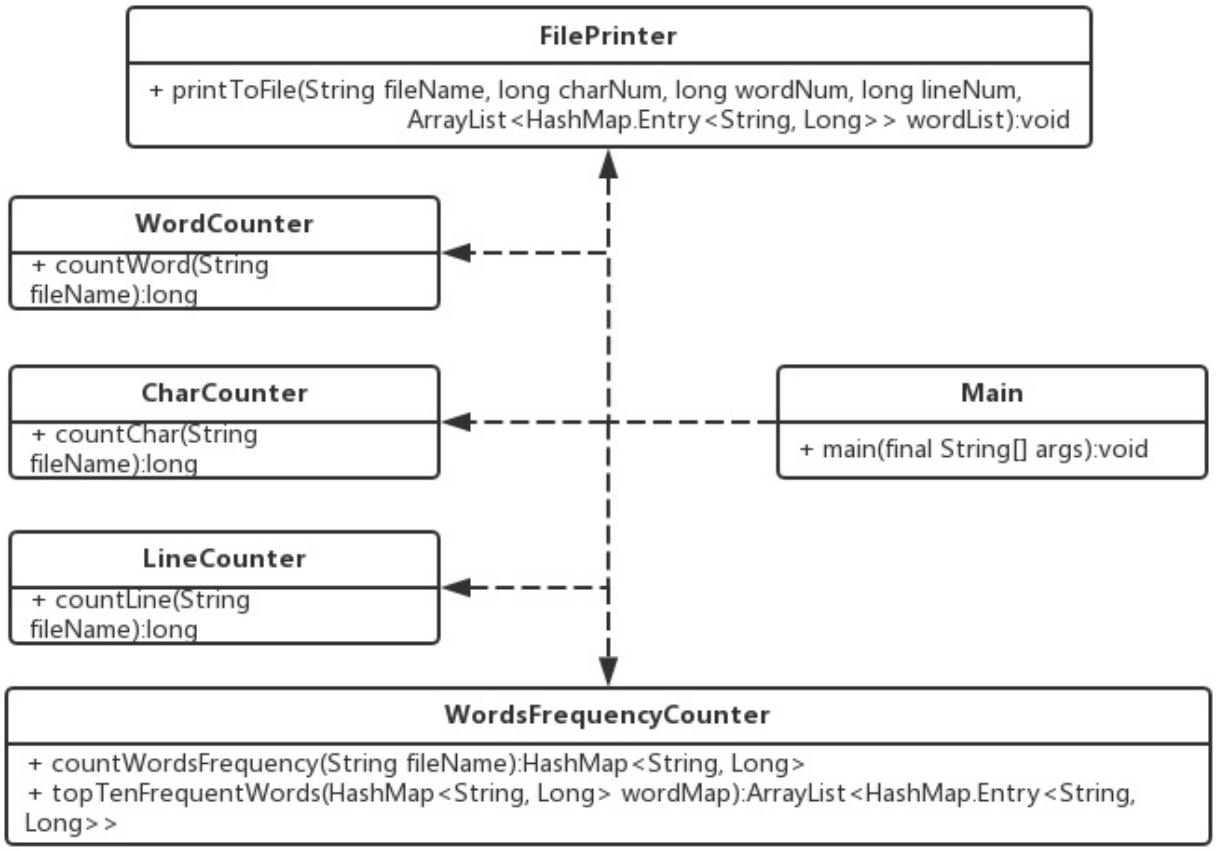
设计说明

总体设计简述

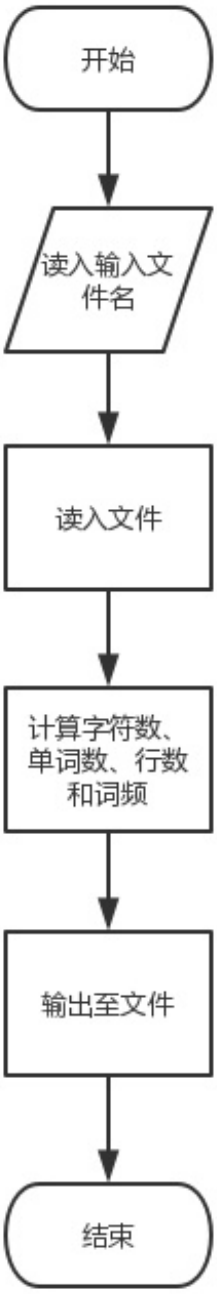
整体由一个计数模块提供字词计数功能，分为字符计数、单词计数、行数计数、词频计数四个部分.

类图及流程图

类图


FilePrinter
+ printToFile(String fileName, long charNum, long wordNum, long lineNum,
ArrayList<HashMap.Entry<String, Long>> wordList):void
WordCounter
+ countWord(String
fileName):long
CharCounter
+ countChar(String
fileName):long
Main
+ main(final String[] args):void
LineCounter
+ countLine(String
fileName):long
WordsFrequencyCounter
+ countWordsFrequency(String fileName):HashMap<String, Long>
+ topTenFrequentWords(HashMap<String, Long> wordMap):ArrayList<HashMap.Entry<String,
Long>>

流程图



模块设计

- 计数模块
  - 模块说明
  - 类说明

计数模块

模块说明

通过传入文件名，提供统计字符总数、单词总数、总行数和总词频的功能.

类说明

CharCounter

(1) countChar(String fileName):long
功能：计算字符数
输入：fileName：文件名
输出：文件总字符数

WordCounter

(1) countWord(String fileName):long
功能：计算单词数
输入：fileName：文件名
输出：文件总单词数

LineCounter

(1) countLine(String fileName):long
功能：计算行数
输入：fileName：文件名
输出：文件总行数

WordsFrequencyCounter

(1) countWordsFrequency(String fileName):long
功能：计算单词词频
输入：fileName：文件名
输出：各单词词频

(2) topTenFrequentWords(HashMap wordMap):ArrayList>
功能：求出频率最高的10个单词
输入：wordMap：各单词词频

输出：频率最高的10个单词

关键代码

词频计算器部分，使用StringTokenizer分词，然后用regex匹配，存入HashMap中，再转换为ArrayList进行排序。

```
/**
 * 词频计算器，包括计算文件中各单词词频，只输出频率最高的10个.
 * 频率相同的单词，优先输出字典序靠前的单词.
 *
 * @author xyy
 * @version 1.0 2018/9/12
 * @since 2018/9/11
 */
public class WordsFrequencyCounter {
    /**
     * 读取并计算文件词频.
     *
     * @param fileName 文件名
     * @return 各单词词频
     */
    public static HashMap<String, Long> countWordsFrequency(String fileName) {
        InputStreamReader inputStreamReader = null;
        BufferedReader bufferedReader = null;
        String in = null;
        String regex = "[a-zA-Z]{4,}[a-zA-Z0-9]*";
        String delim = " ,.!?-=*/()[]{}\\\"\\';:\\n\\r\\t“”‘’·——… () 【】 {} \\0";
        String word = "";
        HashMap<String, Long> wordMap = new HashMap<String, Long>(16);

        //读入文件
        try {
            inputStreamReader = new InputStreamReader(new FileInputStream(fileName));
        } catch (FileNotFoundException e) {
            System.out.println("找不到此文件");
            e.printStackTrace();
```

```
            }
            if (inputStreamReader != null) {
                bufferedReader = new BufferedReader(inputStreamReader);
            }
            //计算单词词频
            try {
                while ((in = bufferedReader.readLine()) != null) {
                    in = in.toLowerCase();
                    //根据分隔符分割
                    StringTokenizer tokenizer = new StringTokenizer(in, del
im);
                    while (tokenizer.hasMoreTokens()) {
                        word = tokenizer.nextToken();
                        //匹配单词
                        if (word.matches(regex)) {
                            if (wordMap.get(word) != null) {
                                wordMap.put(word, wordMap.get(word) + 1);
                            } else {
                                wordMap.put(word, 1L);
                            }
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    inputStreamReader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return wordMap;
        }

        /**
         * 求频率最高的10个单词
         *
         * @param wordMap 各单词词频
         * @return 频率最高的10个单词
         */
        public static ArrayList<HashMap.Entry<String, Long>> topTenFrequent
Words(HashMap<String, Long> wordMap) {
            ArrayList<HashMap.Entry<String, Long>> wordList =
                    new ArrayList<HashMap.Entry<String, Long>>(wordMap.entr
```

```
ySet());
74.         Collections.sort(wordList, new Comparator<HashMap.Entry<String,
Long>>() {
75.             public int compare(Map.Entry<String, Long> o1, Map.Entry<St
ring, Long> o2) {
76.                 if (o1.getValue() < o2.getValue()) {
77.                     return 1;
78.                 } else {
79.                     if (o1.getValue().equals(o2.getValue())) {
80.                         if (o1.getKey().compareTo(o2.getKey()) > 0) {
81.                             return 1;
82.                         } else {
83.                             return -1;
84.                         }
85.                     } else {
86.                         return -1;
87.                     }
88.                 }
89.             }
90.         });
91.         return wordList;
92.     }
93. }
```

Main部分，建立线程池，并行运行四个任务，然后输出至文件。

```
1.  /**
2.   * 主函数类，包括提交计数任务、打印结果.
3.   *
4.   * @author xyy
5.   * @version 1.0 2018/9/12
6.   * @since 2018/9/11
7.   */
8.  public class Main {
9.      public static void main(final String[] args) {
10.         ExecutorService executor = Executors.newCachedThreadPool();
11.
12.         //计算字符数
13.         Future<Long> futureChar = executor.submit(new Callable<Long>()
{
14.             public Long call() {
15.                 return CharCounter.countChar(args[0]);
16.             }
17.         });
```

```
18.
19.             //计算单词数
20.             Future<Long> futureWord = executor.submit(new Callable<Long>()
    {
21.                 public Long call() {
22.                     return WordCounter.countWord(args[0]);
23.                 }
24.             });
25.
26.             //计算行数
27.             Future<Long> futureLine = executor.submit(new Callable<Long>()
    {
28.                 public Long call() {
29.                     return LineCounter.countLine(args[0]);
30.                 }
31.             });
32.
33.             //计算单词词频
34.             Future<ArrayList<HashMap.Entry<String, Long>>>
    futureWordFrequnency = executor.submit(
35.                     new Callable<ArrayList<HashMap.Entry<String, Long>>>()
    {
36.                         public ArrayList<HashMap.Entry<String, Long>> call(
    ) {
37.                             return
    WordsFrequencyCounter.topTenFrequentWords(
38.
    WordsFrequencyCounter.countWordsFrequency(args[0]));
39.                         }
40.                     });
41.
42.             //输出至文件
43.             try {
44.                 FilePrinter.printToFile("result.txt",
45.                         futureChar.get(), futureWord.get(), futureLine.get(
    ), futureWordFrequnency.get());
46.                 executor.shutdown();
47.             } catch (ExecutionException e) {
48.                 e.printStackTrace();
49.             } catch (InterruptedException e) {
50.                 e.printStackTrace();
51.             }
52.         }
53.     }
```

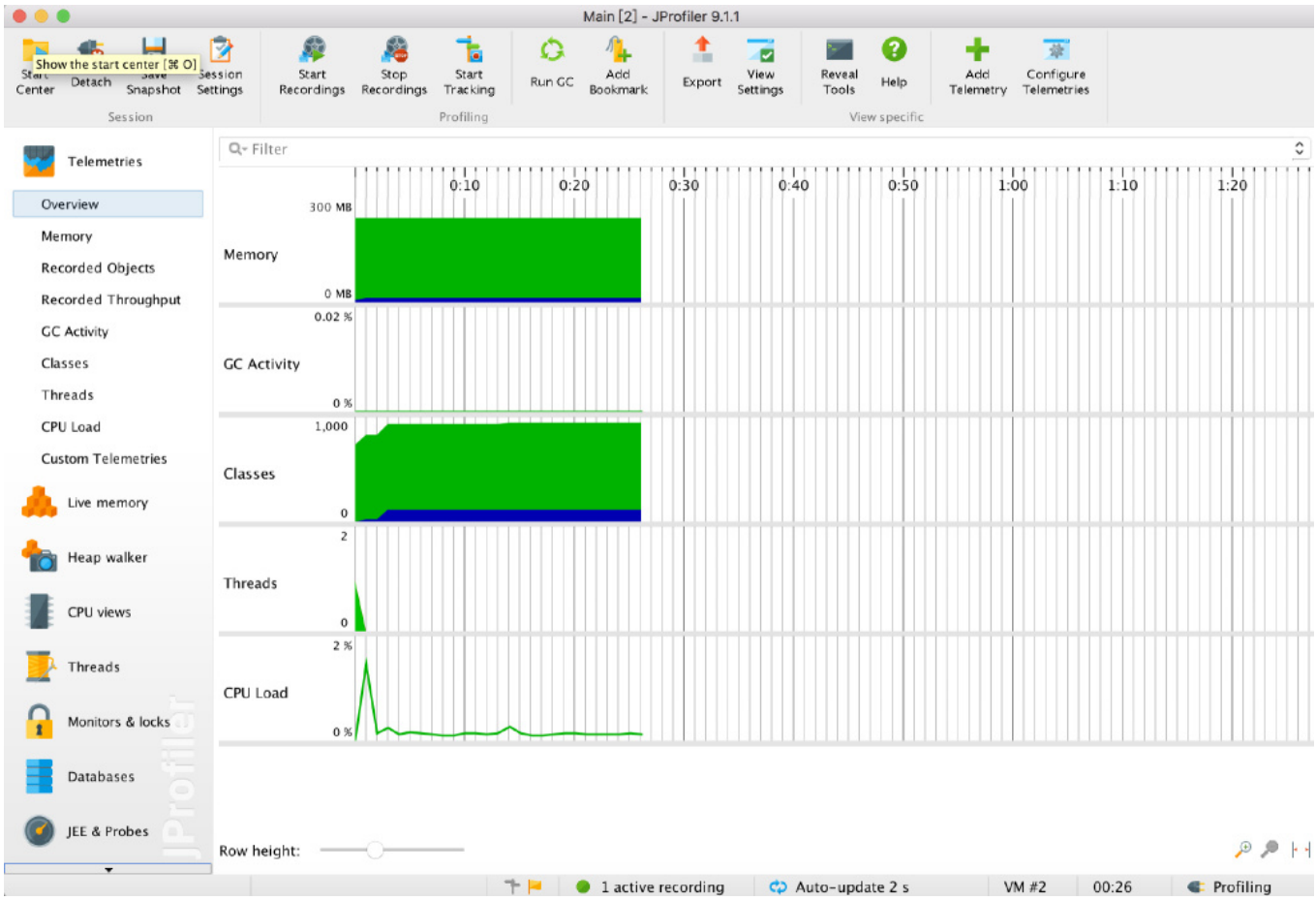
异常处理

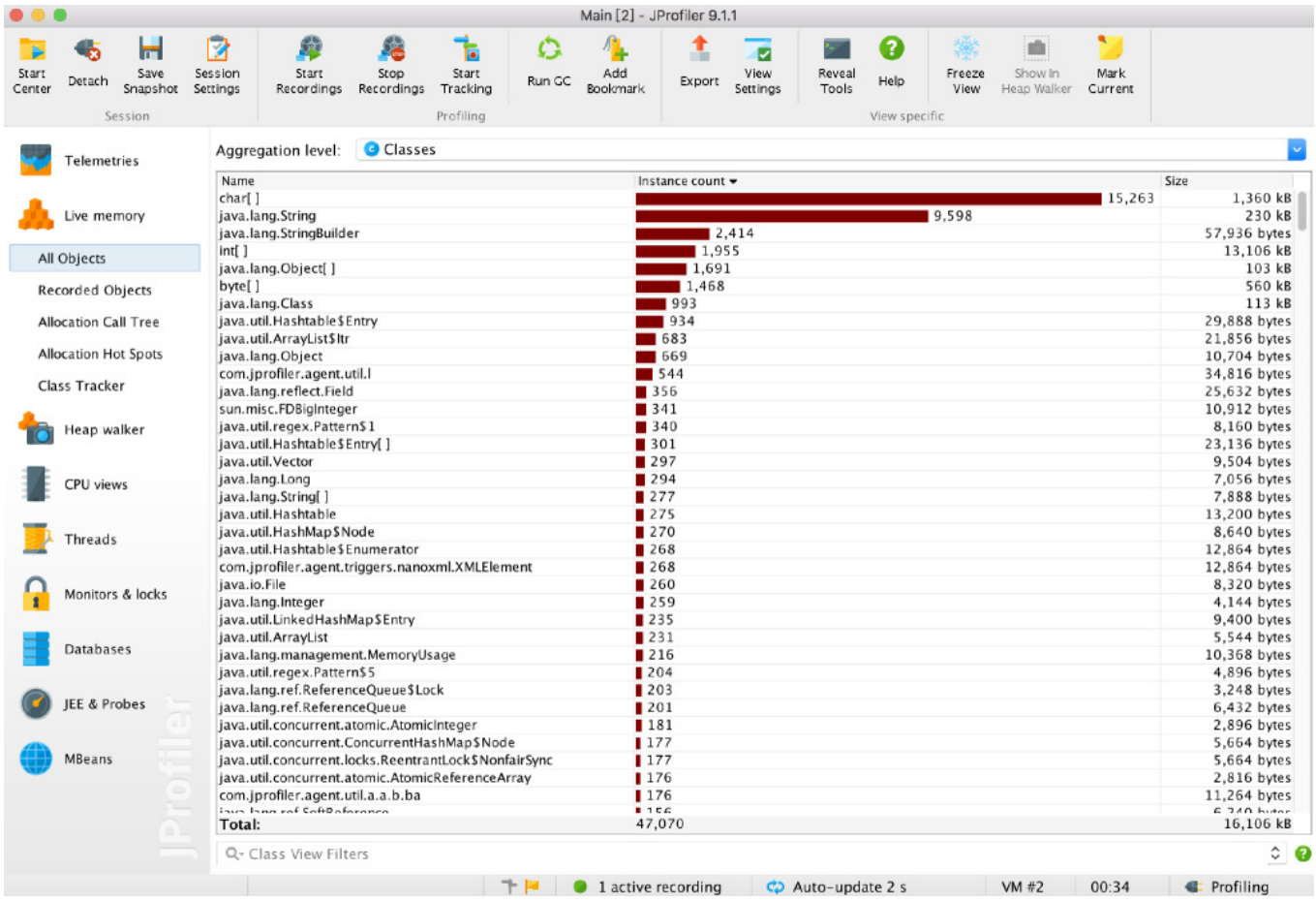
对于各个异常情况都会打印异常信息，如读取文件时，如果找不到对应文件：

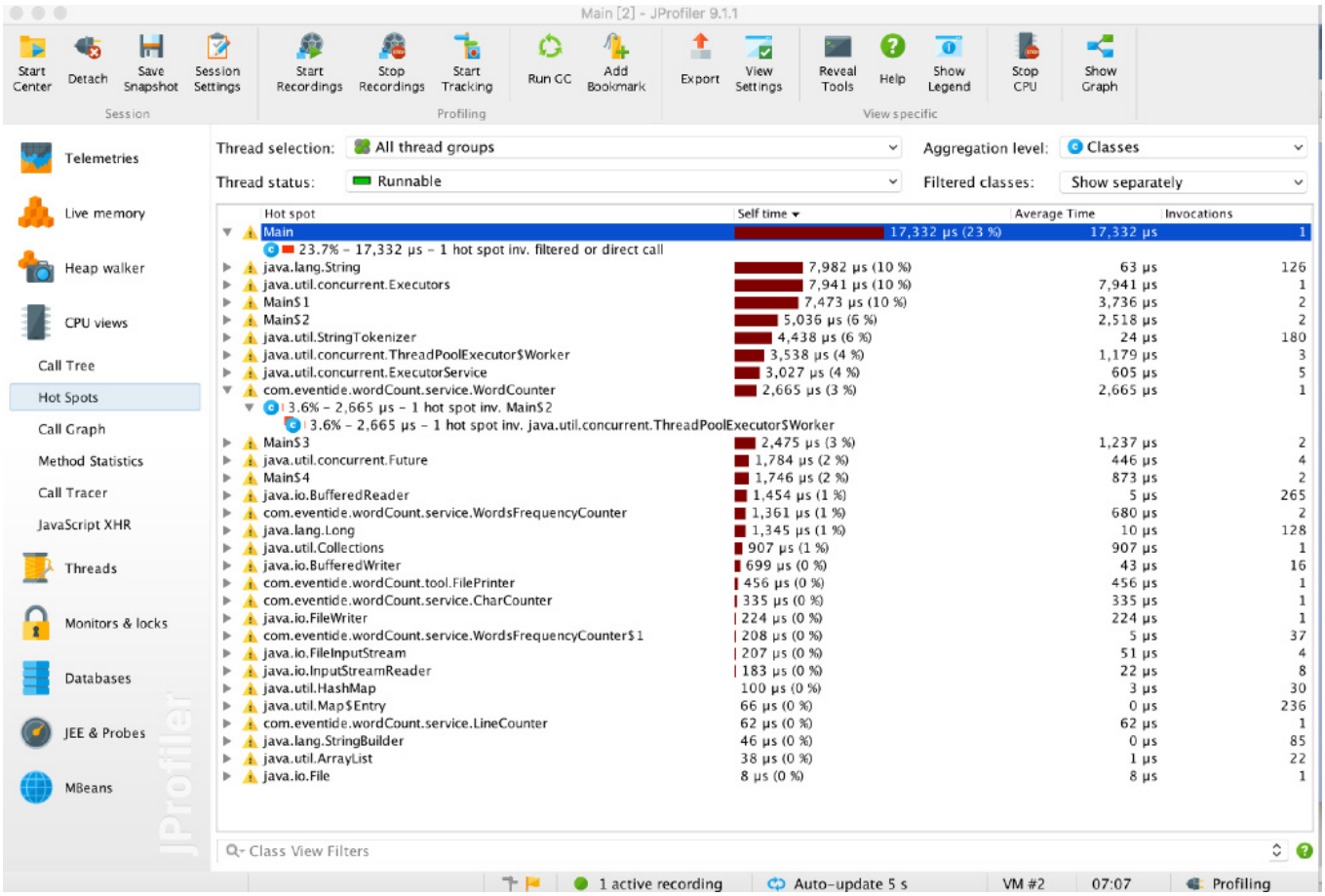
```
try {
    inputStreamReader = new InputStreamReader(new FileInputStream(fileN
ame));
} catch (FileNotFoundException e) {
    System.out.println("找不到此文件");
    e.printStackTrace();
}
```

性能分析

可见最大开销来源于多线程并行以及单词计数部分。


Main [2] - JProfiler 9.1.1
Show the start center [⌘ O]
Start Center
Detach
Save Snapshot
Session Settings
Session
Start Recordings
Stop Recordings
Start Tracking
Profiling
Run GC
Add Bookmark
Export
View Settings
Reveal Tools
Help
View specific
Add Telemetry
Configure Telemetries
Telemetries
Overview
Memory
Recorded Objects
Recorded Throughput
GC Activity
Classes
Threads
CPU Load
Custom Telemetries
Live memory
Heap walker
CPU views
Threads
Monitors & locks
Databases
JEE & Probes
JProfiler
Filter
0:10
0:20
0:30
0:40
0:50
1:00
1:10
1:20
300 MB
Memory
0 MB
0.02 %
GC Activity
0 %
1,000
Classes
0
2
Threads
0
2 %
CPU Load
0 %
Row height:
1 active recording
Auto-update 2 s
VM #2
00:26
Profiling


Main [2] - JProfiler 9.1.1
Start Center
Detach
Save Snapshot
Session Settings
Session
Start Recordings
Stop Recordings
Start Tracking
Profiling
Run GC
Add Bookmark
Export
View Settings
Reveal Tools
Help
Freeze View
Show In Heap Walker
Mark Current
View specific
Telemetries
Live memory
All Objects
Recorded Objects
Allocation Call Tree
Allocation Hot Spots
Class Tracker
Heap walker
CPU views
Threads
Monitors & locks
Databases
JEE & Probes
MBeans
JProfiler
Aggregation level: Classes
Name
Instance count
Size
char[ ] 15,263 1,360 kB
java.lang.String 9,598 230 kB
java.lang.StringBuilder 2,414 57,936 bytes
int[ ] 1,955 13,106 kB
java.lang.Object[ ] 1,691 103 kB
byte[ ] 1,468 560 kB
java.lang.Class 993 113 kB
java.util.Hashtable$Entry 934 29,888 bytes
java.util.ArrayList$Itr 683 21,856 bytes
java.lang.Object 669 10,704 bytes
com.jprofiler.agent.util.I 544 34,816 bytes
java.lang.reflect.Field 356 25,632 bytes
sun.misc.FDBigInteger 341 10,912 bytes
java.util.regex.Pattern$1 340 8,160 bytes
java.util.Hashtable$Entry[ ] 301 23,136 bytes
java.util.Vector 297 9,504 bytes
java.lang.Long 294 7,056 bytes
java.lang.String[ ] 277 7,888 bytes
java.util.Hashtable 275 13,200 bytes
java.util.HashMap$Node 270 8,640 bytes
java.util.Hashtable$Enumerator 268 12,864 bytes
com.jprofiler.agent.triggers.nanoxml.XMLElement 268 12,864 bytes
java.io.File 260 8,320 bytes
java.lang.Integer 259 4,144 bytes
java.util.LinkedHashMap$Entry 235 9,400 bytes
java.util.ArrayList 231 5,544 bytes
java.lang.management.MemoryUsage 216 10,368 bytes
java.util.regex.Pattern$5 204 4,896 bytes
java.lang.ref.ReferenceQueue$Lock 203 3,248 bytes
java.lang.ref.ReferenceQueue 201 6,432 bytes
java.util.concurrent.atomic.AtomicInteger 181 2,896 bytes
java.util.concurrent.ConcurrentHashMap$Node 177 5,664 bytes
java.util.concurrent.locks.ReentrantLock$NonfairSync 177 5,664 bytes
java.util.concurrent.atomic.AtomicReferenceArray 176 2,816 bytes
com.jprofiler.agent.util.a.a.b.ba 176 11,264 bytes
Total: 47,070 16,106 kB
Class View Filters
1 active recording
Auto-update 2 s
VM #2
00:34
Profiling



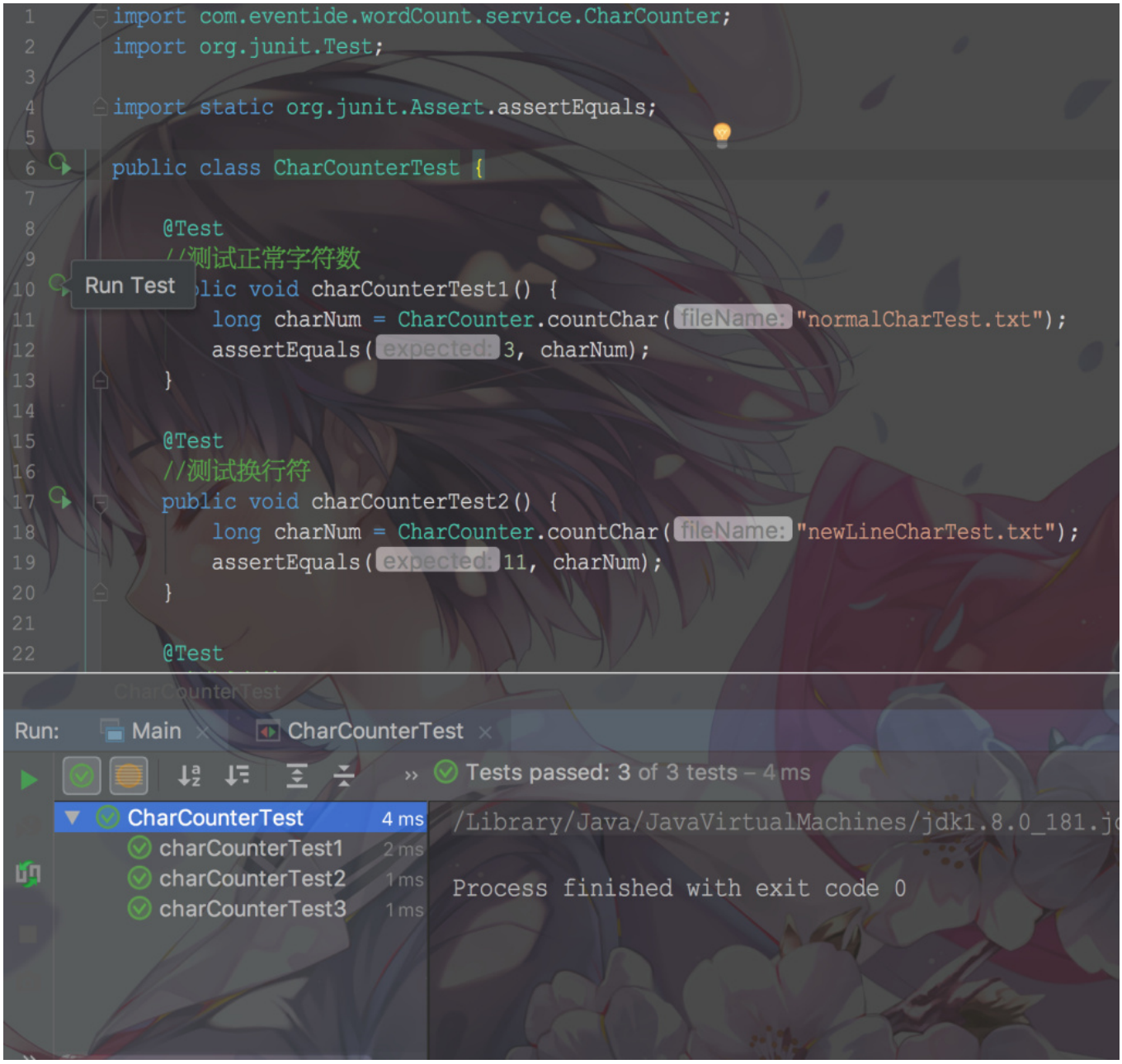
单元测试

单元测试框架用的是JUnit4。

我总共设计了十一个单元测试，其中Main一个，三个字词计数部分各三个，词频计数部分一个。

单元测试	测试项	被测试代码
CharCounterTest	分别测试普通字符、换行符和空格	CharCounter.java
WordCounterTest	分别测试普通单词、特殊单词和大小写单词	WordCounter.java
LineCounterTest	分别测试普通行、空白行和混合行	LineCounter.java

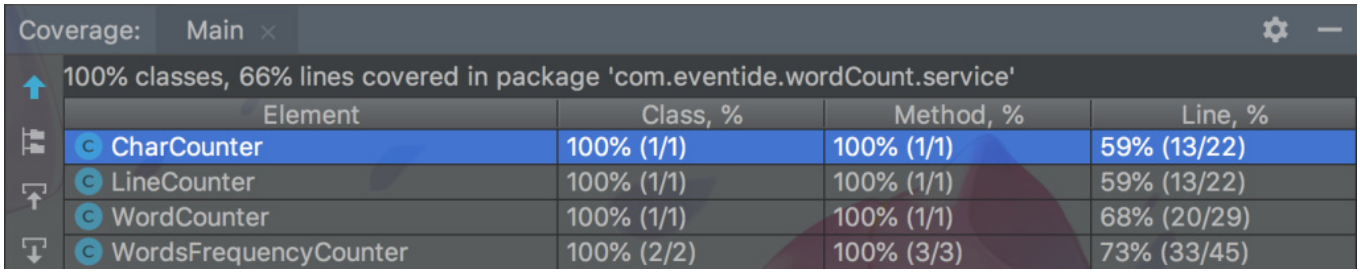
单元测试	测试项	被测试代码
WordFrequencyCounterTest	测试混合单词	WordFrequencyCounter.java
MainTest	测试空白文件	Main.java

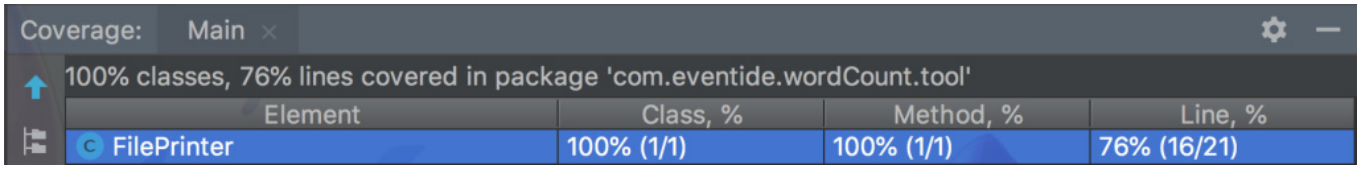


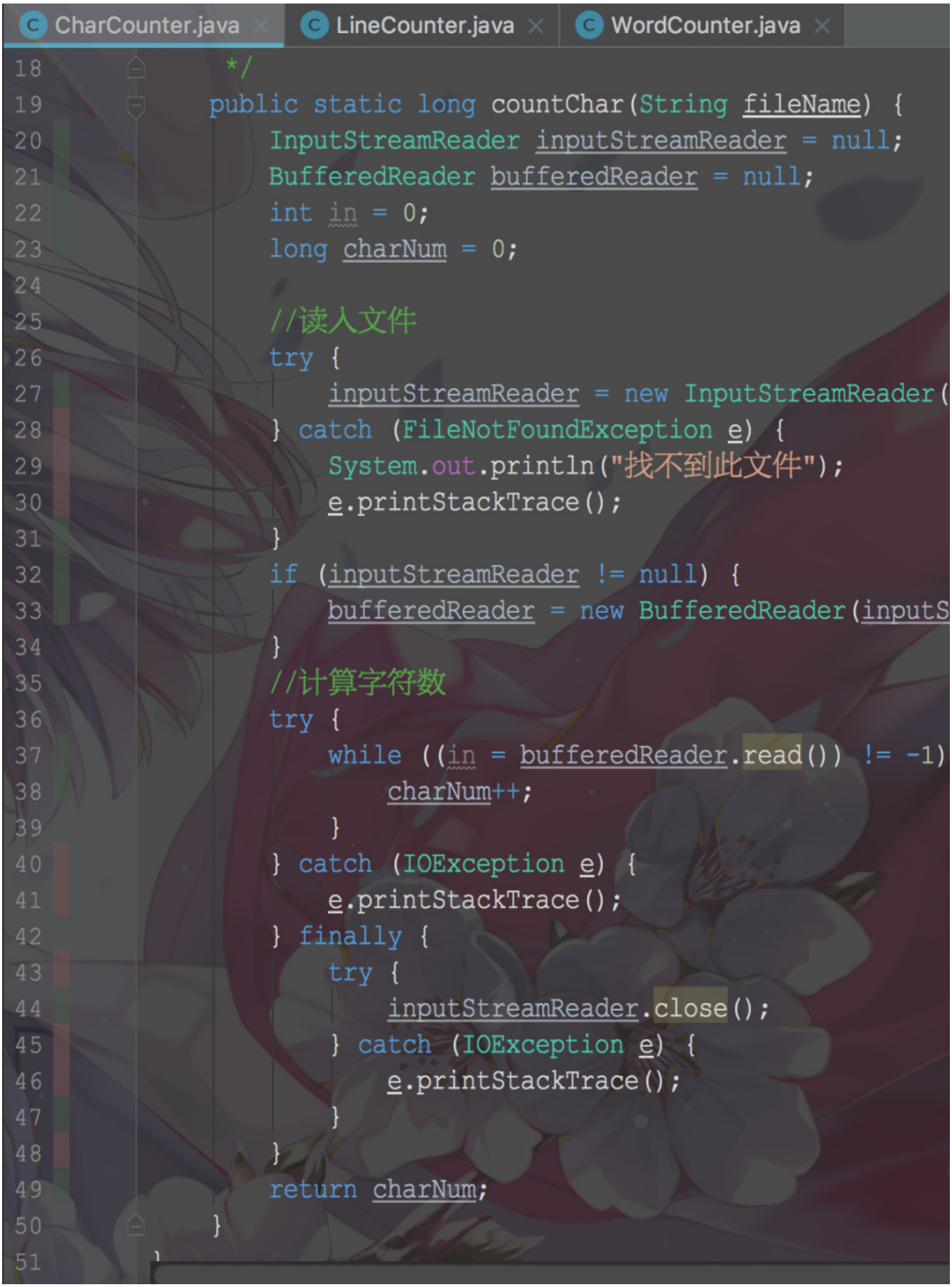
代码覆盖率

检测覆盖率使用的是IDEA的Coverage，截图如下：

因为异常处理并没有单独提出来，而是当场处理了，所以总的代码覆盖率并不高。尤其是功能比较简单的字词行计数部分，许多代码都用来处理读写文件异常了。






CharCounter.java
LineCounter.java
WordCounter.java
*/
public static long countChar(String fileName) {
InputStreamReader inputStreamReader = null;
BufferedReader bufferedReader = null;
int in = 0;
long charNum = 0;
//读入文件
try {
inputStreamReader = new InputStreamReader(
} catch (FileNotFoundException e) {
System.out.println("找不到此文件");
e.printStackTrace();
}
if (inputStreamReader != null) {
bufferedReader = new BufferedReader(inputS
}
//计算字符数
try {
while ((in = bufferedReader.read()) != -1)
charNum++;
}
} catch (IOException e) {
e.printStackTrace();
} finally {
try {
inputStreamReader.close();
} catch (IOException e) {
e.printStackTrace();
}
}
return charNum;
}

感想

这次最大的感想就是差点没赶上deadline。。虽然时间预估看上去没有出现太多问题，但这实际上算是用工程质量的下降换来的，有许多地方没有达到原先预想的水平。因为之前有了几次做小项目的经验，所以我很重视需求分析和设计文档，事前也做了许多学习，但实际上手时，还是遇到比较多的问题。很多问题还是源于我对java编程和各个工具的使用还不够熟练，特别是异常处理和单元测试部分，非常不满意。。
也因为还不熟练，很多知识需要当场查阅学习，浪费了很多时间。最后实际编码时间其实不长，一次编码中也遗留了一些小问题，到测试时才再一一解决。

通过这次的作业，我也对单元测试有了个大概的理解。之前做测试都是手动编写一些样例进行测试，就像做算法一样。不过比较糟糕的是我是在编码结束后才编写单元测试的。。在学习相关内容时，我才了解到单元测试最好在设计时就写好，或者至少也应该跟程序一起写了。而且我编写的单元测试也比较简单，有许多用法还在学习。

还有一点就是对GitHub的使用，其实也是对代码的管理。我之前是不常用Git的，常常是按自己的习惯在本地进行保存和版本管理。做实验室的项目时，也没有很好地利用svn，经常是完成了几个部分才一起提交，但并这不符合实际软件工程的要求。而且我还学会了怎么更好地书写commit message，对比之前惨不忍睹的提交记录。。。

这次也算是第一次像点样子的完成了整个软件开发的工程，深感自己在编码和时间把控上还非常不足，希望在之后的结对和组队中能够有所提高。

参考链接

git commit 规范指南
现代软件工程讲义 2 开发技术 - 单元测试 & 回归测试
在IntelliJ IDEA中查看代码覆盖率结果
IDEA 单元测试覆盖技巧
Java 比较字符串之间大小
BFPRT算法O(n)解决第k小的数

Java的简单单元测试例子

Java正则表达式的语法与示例

正则表达式匹配解析过程探讨分析（正则表达式匹配原理）