

# 一、实验名称

---

第6次上机——第12、13、14章内容

## 二、实验内容

---

page104 第12章 基础训练1: 了解字节流与字符流的使用

目标: 理解字节流与字符流处理文件的差异性

page 106 第12章 基础训练4: 从文件中读写对象

目标: 理解对象串行化特点。

page 107 第12章 编程练习(2): 从一个文本文件中读取若干学生成绩, 每个学生成绩占1行, 统计所有学生成绩的平均分。

page114 第13章 基础训练2: 测试列表和集合的差异

目标: 理解 `ArrayList` 和 `HashSet` 的使用差异

page115 第13章 基础训练4: 堆栈和队列的操作方法

目标: 掌握 `ArrayDeque` 用作对栈和队列的操作方法

page123 第14章 基础训练1: Lambda 表达式的使用

目标: 掌握 Lambda 表达式的表示形式, 了解典型函数式接口的使用

page124 第14章 基础训练3: 字符串中单词分离处理

目标: 熟悉 `Stream` 的各种操作

page127 第14章 作业编程题 (5) :

利用随机函数产生数值范围为 1 ~ 20 的 500 个整数构成的流。对该流进行如下处理:

1. 统计每个数值的出现次数。
2. 求所有元素的平均值。

## 三、实验环境

---

调试工具: Version: 1.99.0 Commit: 4437686ffebaf200fa4a6e6e67f735f3edf24ada Date: 2025-04-02T21:35:19.530Z Electron: 34.3.2 ElectronBuildId: 11161073 Chromium: 132.0.6834.210 Node.js: 20.18.3 V8: 13.2.152.41-electron.0 OS: Windows\_NT x64 10.0.26100

服务配置: openjdk 21.0.6 2025-01-21 OpenJDK Runtime Environment (build 21.0.6+7-Ubuntu-124.04.1) OpenJDK 64-Bit Server VM (build 21.0.6+7-Ubuntu-124.04.1, mixed mode, sharing)

## 四、实验设计

---

### page104 第12章 基础训练1

(1) 调试如下程序, 显示源程序文件内容

创建 `TypeFile.java`, 编码内容如下:

```
package Lab06;

import java.io.*;

public class TypeFile {
    public static void main(String[] args) throws Exception {
        FileInputStream infile = new FileInputStream("TypeFile.java");
        int byteRead = infile.read();
        while (byteRead != -1) {
            System.out.print((char) byteRead); // 判断是否读到文件的末尾
            byteRead = infile.read(); // 将字节转化为字符显示
        }
    }
}
```

然后再将这份文件复制到根目录下, 运行原来的java程序, 在终端输出为如下结果

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.TypeFile
package Lab06;

// C12P104T1

import java.io.*;

public class TypeFile {
    public static void main(String[] args) throws Exception {
        FileInputStream infile = new FileInputStream("TypeFile.java");
        int byteRead = infile.read();
        while (byteRead != -1) {
            // åæ æè~»å°æä»¶ç æ«å°%
            System.out.print((char) byteRead); //å°å è½-ää,°å ¶|æçæ°
            byteRead = infile.read();
        }
    }
}%
```

(2) 将程序中的 `FileInputStream` 改成 `FileReader` , 其他不变。

重新编译运行得到输出结果如下:

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.TypeFile
package Lab06;

import java.io.*;

// C12P104T1

public class TypeFile {
    public static void main(String[] args) throws Exception {
        FileInputStream infile = new FileInputStream("TypeFile.java");
        int byteRead = infile.read();
        while (byteRead != -1) {
            // 判断是否读到文件的末尾
            System.out.print((char) byteRead); //将字节转化为字符显示
            byteRead = infile.read();
        }
    }
}%
```

更改后的代码能够正常输出汉字的原因是: `FileReader` 是基于 **字符流** 的, 它会根据文件的默认字符集自动将字节解码为字符。而 `FileInputStream` 是基于 **字节流** 的, 直接读取文件的原始字

节内容，并没有进行字符编码的转换。

### (3) 修改程序的输出语句，删除其中的强制转换

删除强制转换后重新编译运行终端输出结果如下：

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-  
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin  
Lab06.TypeFile  
112,97,99,107,97,103,101,32,76,97,98,48,54,59,10,10,105,109,112,111,114,116,32,106,97,  
118,97,46,105,111,46,42,59,10,10,112,117,98,108,105,99,32,99,108,97,115,115,32,67,49,5  
0,80,49,48,52,84,49,32,123,10,32,32,32,32,112,117,98,108,105,99,32,115,116,97,116,105,  
99,32,118,111,105,100,32,109,97,105,110,40,83,116,114,105,110,103,91,93,32,97,114,103,  
115,41,32,116,104,114,111,119,115,32,69,120,99,101,112,116,105,111,110,32,123,10,32,32  
,32,32,32,32,32,70,105,108,101,73,110,112,117,116,83,116,114,101,97,109,32,105,110,  
102,105,108,101,32,61,32,110,101,119,32,70,105,108,101,73,110,112,117,116,83,116,114,1  
01,97,109,40,34,67,49,50,80,49,48,52,84,49,46,106,97,118,97,34,41,59,10,32,32,32,32,32  
,32,32,32,105,110,116,32,98,121,116,101,82,101,97,100,32,61,32,105,110,102,105,108,101  
,46,114,101,97,100,40,41,59,10,32,32,32,32,32,32,32,32,119,104,105,108,101,32,40,98,12  
1,116,101,82,101,97,100,32,33,61,32,45,49,41,32,123,32,32,32,32,32,32,32,32,32,32,3  
2,32,32,32,32,47,47,32,21028,26029,26159,21542,35835,21040,25991,20214,30340,26411,236  
14,10,32,32,32,32,32,32,32,32,32,32,83,121,115,116,101,109,46,111,117,116,46,112  
,114,105,110,116,40,40,99,104,97,114,41,32,98,121,116,101,82,101,97,100,41,59,32,32,47  
,47,23558,23383,33410,36716,21270,20026,23383,31526,26174,31034,10,32,32,32,32,32,32,3  
2,32,32,32,32,32,98,121,116,101,82,101,97,100,32,61,32,105,110,102,105,108,101,46,114,  
101,97,100,40,41,59,10,32,32,32,32,32,32,32,32,125,10,32,32,32,32,125,10,125,%
```

其中汉字对应的编码值大致是

21028,26029,26159,21542,35835,21040,25991,20214,30340,26411,23614 和  
23558,23383,33410,36716,21270,20026,23383,31526,26174,31034 两块, 因为汉字在 UniCode 中的  
编码非常靠后 (且汉字使用多字节编码)

### (4) 将 `FileReader` 恢复到 `FileInputStream` , 其他不变。

恢复带 `FileInputStream` 之后重新编译运行, 终端得到输出结果如下:

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.TypeFile
112,97,99,107,97,103,101,32,76,97,98,48,54,59,10,10,105,109,112,111,114,116,32,106,97,
118,97,46,105,111,46,42,59,10,10,112,117,98,108,105,99,32,99,108,97,115,115,32,67,49,5
0,80,49,48,52,84,49,32,123,10,32,32,32,32,112,117,98,108,105,99,32,115,116,97,116,105,
99,32,118,111,105,100,32,109,97,105,110,40,83,116,114,105,110,103,91,93,32,97,114,103,
115,41,32,116,104,114,111,119,115,32,69,120,99,101,112,116,105,111,110,32,123,10,32,32
,32,32,32,32,32,70,105,108,101,73,110,112,117,116,83,116,114,101,97,109,32,105,110,
102,105,108,101,32,61,32,110,101,119,32,70,105,108,101,73,110,112,117,116,83,116,114,1
01,97,109,40,34,67,49,50,80,49,48,52,84,49,46,106,97,118,97,34,41,59,10,32,32,32,32,32
,32,32,32,105,110,116,32,98,121,116,101,82,101,97,100,32,61,32,105,110,102,105,108,101
,46,114,101,97,100,40,41,59,10,32,32,32,32,32,32,32,119,104,105,108,101,32,40,98,12
1,116,101,82,101,97,100,32,33,61,32,45,49,41,32,123,32,32,32,32,32,32,32,32,32,32,3
2,32,32,32,32,47,47,32,229,136,164,230,150,173,230,152,175,229,144,166,232,175,187,229
,136,176,230,150,135,228,187,182,231,154,132,230,156,171,229,176,190,10,32,32,32,32,32
,32,32,32,32,32,32,32,83,121,115,116,101,109,46,111,117,116,46,112,114,105,110,116,40,
40,99,104,97,114,41,32,98,121,116,101,82,101,97,100,41,59,32,32,47,47,229,176,134,229,
173,151,232,138,130,232,189,172,229,140,150,228,184,186,229,173,151,231,172,166,230,15
2,190,231,164,186,10,32,32,32,32,32,32,32,32,32,32,32,98,121,116,101,82,101,97,100,
32,61,32,105,110,102,105,108,101,46,114,101,97,100,40,41,59,10,32,32,32,32,32,32,32,32
,125,10,32,32,32,32,125,10,125,%
```

可以发现原来对应汉字的位置较大的编码变成了较小的编码

**总结:** `FileInputStream` 和 `FileReader` 的使用差异

| 特性   | FileInputStream                                        | FileReader                                      |
|------|--------------------------------------------------------|-------------------------------------------------|
| 数据类型 | 基于字节流（InputStream），读取文件的原始字节内容。                        | 基于字符流（Reader），自动将字节解码为字符，支持文本文件的读取。             |
| 编码处理 | 不处理字符编码，直接读取字节。需要手动处理字符编码转换（如通过InputStreamReader指定编码）。 | 根据系统默认字符集（或指定的编码）自动解码字节为字符。                     |
| 适用场景 | 适用于读取二进制文件（如图片、音频等）或需要对字节进行低级操作的场景。                    | 适用于读取文本文件，尤其是包含多字节字符（如中文、日文等）的文件。               |
| 输出结果 | 输出的是字节值，若直接转为字符可能会导致乱码（特别是多字节编码的字符）。                   | 输出的是正确解码后的字符，能够正常显示文本内容。                        |
| 灵活性  | 更灵活，可以结合InputStreamReader和BufferedReader使用，支持自定义编码。    | 灵活性较低，默认使用系统字符集，若需要指定其他编码则需改用InputStreamReader。 |

page 106 第12章 基础训练4

(1) 调试如下程序，查看产生的数据文件位置。

```

package Lab06;

import java.io.*;

// WriteObj

public class WriteObj {
    public static void main(String[] args) throws Exception{
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("storedate.dat"));
        out.writeObject("me");
        out.writeObject(new WriteObj());
        System.out.println("写入完毕");
    }
}

```

执行以上程序之后, 可以看到在项目的根目录下生成了一个 `storedate.dat` 的文件

```

> /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java -
XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin Lab06.WriteObj
Exception in thread "main" java.io.NotSerializableException: Lab06.WriteObj
    at
    java.base/java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1200)
    at java.base/java.io.ObjectOutputStream.writeObject(ObjectOutputStre
> ls ~/Code/Java/HW
bin  TypeFile.java  doc  img  lib  README.md  src  storedate.dat

```

但是抛出了一个 `NotSerializableException` 异常, 因为该对象的类没有实现 `Serializable` 接口。

`ObjectOutputStream` 的 `writeObject` 方法用于将对象序列化并写入到输出流中。要使一个对象能够被序列化, 它的类必须实现 `Serializable` 接口。否则, 运行时会抛出 `NotSerializableException` 异常。

## (2) 修改程序类头部分, 增加子句 `implements Serializable`

修改后的部分程序如下:

```

...
public class WriteObj implements Serializable {
    ...
}

```

再次编译运行该程序, 得到终端的输出结果如下:

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-  
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin  
Lab06.WriteObj  
写入完毕
```

说明正确向 `storedate.dat` 文件写入

### (3) 调试以下程序，从文件 `storedate.dat` 中读取先前写入的对象数据并输出

```
package Lab06;  
  
import java.io.*;  
  
// C12P106T4  
  
public class ReadObj {  
    public static void main(String[] args) {  
        try {  
            ObjectInputStream in = new ObjectInputStream(  
                new FileInputStream("storedate.dat"));  
            while (true) {  
                System.out.println(in.readObject());  
            }  
        } catch (Exception e) {}  
    }  
}
```

通过编译运行以上程序, 得到输出结果如下:

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-  
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin  
Lab06.ReadObj  
me  
Lab06.WriteObj@7a07c5b4
```

也就是说, 之前 `WriteObj.java` 程序中成功向 `storedate.dat` 文件中写入 `me`, 然后成功由 `ReadObj.java` 文件读取并正常输出.

**总结:** 对象串行化的数据读写特点:

- 要序列化的类必须实现 `Serializable` 接口。否则, 尝试序列化时会抛出 `NotSerializableException`。
- 通过 `ObjectOutputStream` 可以将对象的状态保存到文件中, 实现了对象的持久化存储。

- 使用 `ObjectInputStream` 可以从文件中读取并恢复对象状态
- 如果未重写对象的 `toString()` 方法，反序列化后打印对象时会显示默认格式（如 类名@哈希码），例如 `Lab06.WriteObj@7a07c5b4`。

---

## page 107 第12章 编程练习(2)

从一个文本文件中读取若干学生成绩，每个学生成绩占1行，统计所有学生成绩的平均分。首先创建文本文件，放置于项目根目录下的 `data` 文件夹中，命名为 `students.txt`，内容如下

```
学号,姓名,成绩
1,张三,60
2,李四,70
3,王五,80
4,赵六,90
5,agjio,abc
```

完成上述要求的代码如下:

```

package Lab06;

import java.io.*;
import java.util.*;

public class C12P107E2 {
    public static void main(String[] args) throws FileNotFoundException {
        String filePath = "data/students.txt";

        int sum = 0;
        int cnt = 0;
        int invalidcnt = 0;

        try (Scanner sc = new Scanner(new File(filePath))) {
            // 跳过表头
            if (sc.hasNextLine()) sc.nextLine();

            while (sc.hasNextLine()) {
                String line = sc.nextLine().trim();
                if (!line.isEmpty()) {
                    String[] parts = line.split(",");

                    if (parts.length == 3) { // 确保是完整的数据
                        try {
                            int score = Integer.parseInt(parts[2].trim());
                            sum += score;
                            cnt++;
                        } catch (NumberFormatException e) {
                            invalidcnt++;
                            System.out.println("成绩不合法: " + line);
                        }
                    } else {
                        invalidcnt++;
                        System.out.println("数据格式不正确: " + line);
                    }
                }
            }

            if (cnt > 0) {
                double ave = (double) sum / cnt;
                System.out.println("-----");
                System.out.println("有效成绩数量: " + cnt);
                System.out.println("无效成绩数量: " + invalidcnt);
                System.out.println("总成绩: " + sum);
                System.out.println("平均成绩: " + ave);
            } else {
                System.out.println("没有有效的成绩数据。");
            }
        } catch (FileNotFoundException e) {
            System.out.println("文件未找到: " + filePath);
        }
    }
}

```

```

    }
}
}

```

编译运行该程序在终端输出结果如下:

```

> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.C12P107E2
成绩不合法: 5,agjio,abc
-----
有效成绩数量: 4
无效成绩数量: 1
总成绩: 300
平均成绩: 75.0

```

## page114 第13章 基础训练2

```

package Lab06;

import java.util.*;

public class C13P114T2 {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<>();
        a.add("one");
        a.add("three");
        a.add("one");
        a.add("two");
        System.out.println(a);
    }
}

```

以上程序运行输出结果如下:

```

> /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java -
XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin Lab06.C13P114T2
[one, three, one, two]

```

**思考:** 将上面程序中的 `ArrayList` 改为 `HashSet`, 重新编译运行, 终端输出结果如下:

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-  
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin  
Lab06.C13P114T2  
[one, three, two]
```

总结: 列表 ( ArrayList ) 与集合 ( HashSet ) 的差异

| 特性   | ArrayList      | HashSet        |
|------|----------------|----------------|
| 存储顺序 | 按插入顺序存储        | 由哈希算法决定        |
| 允许重复 | 允许重复元素         | 不允许重复元素        |
| 适用场景 | 需要顺序存储或允许重复的数据 | 需要唯一性且不关心顺序的数据 |

## page115 第13章 基础训练4

按照书中代码编译运行, 终端输出结果如下:

```
> /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java -  
XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin Lab06.C13P115T4  
北京大学  
清华大学  
湖南大学  
四川大学
```

如果队列中允许混合存放字符串和整数两类数据, 那么 `ArrayDeque` 的泛型参数应该改成 `Object` .

因为 `Object` 是所有类的基类, 因此 `ArrayDeque<Object>` 可以存储任何类型的对象.

修改之后的程序如下:

```

package Lab06;

import java.util.*;

public class C13P115T4 {
    public static void main(String[] args) {
        // ArrayDeque <String> c = new ArrayDeque<>();
        ArrayDeque <Object> c = new ArrayDeque<>();
        c.push("清华大学");
        c.push("吉林大学");
        c.pop();
        c.push("北京大学");
        while (!c.isEmpty()) {
            System.out.println(c.pop());
        }
        c.offer("湖南大学");
        c.offer("四川大学");
        c.offer(985);
        c.offer(211);
        while(!c.isEmpty()) {
            System.out.println(c.poll());
        }
    }
}

```

得到输出结果为:

```

> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.C13P115T4
北京大学
清华大学
湖南大学
四川大学
985
211

```

## page123 第14章 基础训练1

设有一批整数，分别按以下条件进行过滤，输出过滤后的数据

1. 保留偶数数据
2. 保留能被 3 或 5 整除的数

```

package Lab06;

import java.util.*;
import java.util.function.Predicate;

public class C14P123T1 {
    public static void main(String[] args) {
        List<Integer> x = Arrays.asList(3, 6, 8, 23, 6, 67, 34);
        System.out.println(filter(x, e -> e % 2 == 0));
        System.out.println(filter(x, e -> e % 3 == 0 || e % 5 == 0));
    }

    public static List<Integer> filter(List<Integer> data, Predicate<Integer> c) {
        List<Integer> r = new ArrayList<Integer>();
        for (int s : data) {
            if (c.test(s)) {
                r.add(s);
            }
        }
        return r;
    }
}

```

以上程序输出如下:

```

> /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java -
XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin Lab06.C14P123T1
[6, 8, 6, 34]
[3, 6, 6]

```

## page124 第14章 基础训练3

从给定句子中返回单词长度大于 3 的单词列表，按长度倒序输出，限制仅输出 3 个。

```

package Lab06;

import java.util.*;
import java.util.stream.Collectors;

public class C14P124T3 {
    public static void main(String[] args) {
        String sentence = "how are you, welcome you to china";
        List<String> x = Arrays.stream(sentence.split("[\\s,]+"))
            .filter(word->word.length() >= 3)
            .sorted((a, b) -> a.length() - b.length())
            .limit(3)
            .collect(Collectors.toList());
        System.out.println(x);
    }
}

```

以上程序输出结果为:

```

> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.C14P124T3
[how, are, you]

```

将程序中的单词长度限制数据改为 2，并将输出限制数量改为 5 后, 输出如下:

```

> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin
Lab06.C14P124T3
[to, how, are, you, you]

```

在该代码中用于单词分离的正则表达式的含义为:

- `[\\s,]`：表示匹配空格（`\\s`）或逗号（`,`）。
- `+`：表示匹配一个或多个连续的空格或逗号。

## page127 第14章 作业编程题（5）

利用随机函数产生数值范围为 1 ~ 20 的 500 个整数构成的流。对该流进行如下处理：

1. 统计每个数值的出现次数。
2. 求所有元素的平均值。

【说明】为实现两方面的统计要求，可以对同一个数据源，两次形成各自的流进行处理，统计每个数值的出现次数，可以先得到有哪些数值在流中出现，使用流的 `distinct()` 方法就可以得到，然后针对结果流中的数据在原集合中的出现次数进行统计处理。求平均值可以将数据集合构成的流转换为原生流，利用原生流的 `average()` 方法来求平均值。

按照要求与说明最终编码如下:

```
package Lab06;

import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class C14P127E5 {
    public static void main(String[] args) {
        // 使用随机函数生成 500 个范围为 1~20 的整数
        Random random = new Random();
        List<Integer> numbers = IntStream.generate(() -> random.nextInt(20) + 1)
            .limit(500)
            .boxed()
            .collect(Collectors.toList());

        // 1. 统计每个数值的出现次数，并按出现频率排序
        System.out.println("每个数值的出现次数(按频率降序排列): ");

        numbers.stream()
            .collect(Collectors.groupingBy(n -> n, Collectors.counting()))
            .entrySet().stream()
            .sorted(Map.Entry.<Integer, Long>comparingByValue().reversed()) // 按频率降序排序
            .forEach(entry -> {
                System.out.println("数值 " + entry.getKey() + " 出现了 " +
                    entry.getValue() + " 次");
            });

        // 2. 求所有元素的平均值
        OptionalDouble average = numbers.stream()
            .mapToInt(Integer::intValue)
            .average();

        System.out.printf("所有元素的平均值: " + average.getAsDouble());
    }
}
```

由于数据是随机生成的, 每次结果执行均不一样, 下面是一次执行的结果

```
> cd /home/ /Code/Java/HW ; /usr/bin/env /usr/lib/jvm/java-21-openjdk-  
amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/ /Code/Java/HW/bin  
Lab06.C14P127E5
```

每个数值的出现次数 (按频率降序排列):

数值 1 出现了 33 次  
数值 13 出现了 31 次  
数值 14 出现了 31 次  
数值 6 出现了 29 次  
数值 5 出现了 28 次  
数值 15 出现了 27 次  
数值 12 出现了 26 次  
数值 20 出现了 26 次  
数值 7 出现了 24 次  
数值 8 出现了 24 次  
数值 11 出现了 24 次  
数值 19 出现了 24 次  
数值 10 出现了 23 次  
数值 16 出现了 23 次  
数值 17 出现了 23 次  
数值 9 出现了 22 次  
数值 18 出现了 22 次  
数值 3 出现了 21 次  
数值 2 出现了 20 次  
数值 4 出现了 19 次  
所有元素的平均值: 10.526%

可以看出数据出现频率是相对平均的, 最终的平均值也非常接近 10.5

## 五、实验心得

通过本次实验, 我深入学习了Java的文件IO操作、集合框架和函数式编程, 获得了以下收获:

### 文件IO操作

- **字节流与字符流差异:** 理解了 `FileInputStream` 处理原始字节与 `FileReader` 自动处理字符编码的根本区别
- **文本文件处理:** 掌握了如何正确读取和解析文本文件中的数据, 包括处理无效数据的异常情况
- **对象序列化:** 学习了如何通过实现 `Serializable` 接口将对象持久化到文件中, 以及使用 `ObjectInputStream` / `ObjectOutputStream` 进行对象的读写操作

### 集合框架

- **列表与集合区别**：通过对比 `ArrayList` 和 `HashSet`，理解了两者在数据存储、顺序维护和元素唯一性上的差异
- **栈和队列操作**：掌握了使用 `ArrayDeque` 作为栈和队列的基本操作方法，了解了其在不同场景下的应用方式
- **集合类型选择**：明确了根据需求选择合适集合类型的原则，如需要元素唯一性时选择 `Set`，需要保持插入顺序时选择 `List`

## 函数式编程

- **Lambda表达式**：学习了Lambda表达式的基本语法和使用方式，体会到了它如何简化函数式接口的实现
- **Stream API**：掌握了流操作的基本方法，如过滤( `filter` )、映射( `map` )、收集( `collect` )、统计( `count` 、 `average` )等
- **函数式思维**：理解了函数式编程与传统命令式编程的差异，体会到使用Stream API处理集合数据的简洁性和表达力

## 遇到的问题

1. 在page124 第14章 基础训练3中, 遇到了正则表达式, 对正则表达式的语法规则并不了解, 通过文档结合AI来快速入门正则表达式的语法. 由AI整理得来的正则表达式基本语法与用法如下:

| 语法     | 描述              | 示例      | 匹配示例                     |
|--------|-----------------|---------|--------------------------|
| .      | 匹配任意单个字符（换行符除外） | a.c     | "abc", "a1c", "a-c"      |
| \      | 转义特殊字符          | \.      | 匹配字符串中的 "."              |
| [abc]  | 匹配括号内任意一个字符     | [aeiou] | "a", "e"                 |
| [^abc] | 匹配不在括号内的任意字符    | [^0-9]  | "a", "@"（排除数字）           |
| \d     | 匹配数字（0-9）       | \d{3}   | "123", "456"             |
| \w     | 匹配字母、数字、下划线     | \w+     | "hello", "user123"       |
| \s     | 匹配空格、制表符、换行符    | \s+     | " "（连续空格）                |
| *      | 匹配前一个元素 0次或多次   | ab*c    | "ac", "abbc"             |
| +      | 匹配前一个元素 1次或多次   | ab+c    | "abc", "abbc"（排除 "ac"）   |
| ?      | 匹配前一个元素 0次或1次   | colou?r | "color", "colour"        |
| {n}    | 匹配前一个元素恰好n次     | a{3}    | "aaa"                    |
| {n,m}  | 匹配前一个元素 n到m次    | a{2,4}  | "aa", "aaaa"             |
| (abc)  | 分组（可后续引用或处理）    | (ab)+   | "abab"                   |
| ^      | 匹配字符串开始         | ^Hello  | "Hello world" 中的 "Hello" |
| \$     | 匹配字符串结束         | end\$   | "The end" 中的 "end"       |
|        | 或运算符，匹配左侧或右侧    | cat dog | "cat", "dog"             |

2. 对Stream API的操作链使用非常不熟练, 通过多次增删与排列组合初步掌握基本操作的用法. 同样地, 以下是AI整理的常用Stream API的用法

### 中间操作

| 方法                      | 语法示例                                                                  | 描述                                 | 示例说明                                                                                       |
|-------------------------|-----------------------------------------------------------------------|------------------------------------|--------------------------------------------------------------------------------------------|
| <code>filter()</code>   | <code>stream.filter(Predicate&lt;T&gt;)</code>                        | 过滤元素，保留满足条件的元素                     | <code>list.stream().filter(n -&gt; n &gt; 10)</code> → 过滤大于10的元素                           |
| <code>map()</code>      | <code>stream.map(Function&lt;T, R&gt;)</code>                         | 将元素转换为另一种类型                        | <code>list.stream().map(String::length)</code> → 字符串转为长度                                   |
| <code>flatMap()</code>  | <code>stream.flatMap(Function&lt;T, Stream&lt;R&gt;&gt;)</code>       | 将元素转换为流，并合并所有流（扁平化）                | <code>list.stream().flatMap(List::stream)</code> → 合并多个列表为一个流                              |
| <code>sorted()</code>   | <code>stream.sorted()</code> 或 <code>stream.sorted(Comparator)</code> | 对元素排序（默认自然顺序或自定义比较器）               | <code>list.stream().sorted()</code> → 自然顺序<br><code>sorted((a, b) -&gt; b - a)</code> → 降序 |
| <code>distinct()</code> | <code>stream.distinct()</code>                                        | 去重（依赖元素的 <code>equals()</code> 方法） | <code>list.stream().distinct()</code> → 去除重复元素                                             |
| <code>limit(n)</code>   | <code>stream.limit(long n)</code>                                     | 截取前 $n$ 个元素                        | <code>list.stream().limit(3)</code> → 取前3个元素                                               |
| <code>skip(n)</code>    | <code>stream.skip(long n)</code>                                      | 跳过前 $n$ 个元素                        | <code>list.stream().skip(2)</code> → 跳过前2个元素                                               |
| <code>peek()</code>     | <code>stream.peek(Consumer&lt;T&gt;)</code>                           | 对元素执行操作（常用于调试）                     | <code>list.stream().peek(System.out::println)</code> → 打印每个元素                              |

终止操作

| 方法                       | 语法示例                                                | 描述                                                                      |                                                                              |
|--------------------------|-----------------------------------------------------|-------------------------------------------------------------------------|------------------------------------------------------------------------------|
| <code>forEach()</code>   | <code>stream.forEach(Consumer&lt;T&gt;)</code>      | 对每个元素执行操作<br>(无返回值)                                                     | <code>list.stream()</code><br><code>System.out.println();</code><br>→ 打印所有元素 |
| <code>collect()</code>   | <code>stream.collect(Collector)</code>              | 将流转换为集合 (如<br><code>List</code> 、 <code>Set</code> 、 <code>Map</code> ) | <code>list.stream()</code><br>→ 转为 <code>List</code>                         |
| <code>reduce()</code>    | <code>stream.reduce(BinaryOperator&lt;T&gt;)</code> | 将元素归约为一个值<br>(如求和、最大值)                                                  | <code>list.stream()</code><br>→ 求和                                           |
| <code>min()/max()</code> | <code>stream.min(Comparator)</code>                 | 返回最小/最大值 (需<br>提供比较器)                                                   | <code>list.stream()</code><br>→ 返回最大值                                        |
| <code>count()</code>     | <code>stream.count()</code>                         | 返回流中元素个数                                                                | <code>list.stream()</code><br>→ 统计正数个数                                       |
| <code>anyMatch()</code>  | <code>stream.anyMatch(Predicate&lt;T&gt;)</code>    | 判断是否有元素满足<br>条件 (返回<br><code>boolean</code> )                           | <code>list.stream()</code><br><code>s.contains('a')</code><br>→ 检查是否包含 'a'   |
| <code>findFirst()</code> | <code>stream.findFirst()</code>                     | 返回第一个元素 (返<br>回 <code>Optional&lt;T&gt;</code> )                        | <code>list.stream()</code><br>→ 获取第一个元素                                      |
| <code>toArray()</code>   | <code>stream.toArray()</code>                       | 将流转换为数组                                                                 | <code>list.stream()</code><br>→ 转为字符串数组                                      |

通过这次实验，我不仅加深了对Java IO、集合和函数式编程的理解，也认识到了这些技术在实际编程中的重要性。特别是Stream API的学习，让我体会到了现代编程范式，代码更加简洁、可读且高效。