

深圳大学实验报告

课程名称：Java 程序设计

实验项目名称：课程实验 3：常用集合类和线程

学院：计算机与软件学院

专业：数计班

指导教师：潘微科

报告人：詹耿羽 学号：2023193026 班级：数计

实验时间：2024 年 11 月 1 日（周五）-2024 年 11 月 20 日（周三）

实验报告提交时间：2024/11/15

教务部制

实验目的与要求:

实验目的: 掌握常用的集合类, 能够较为熟练的查阅 Java 提供的常见的类, 并进行程序设计, 掌握 Java 程序设计中的线程同步等技术。

实验要求:

Part 1 (25 分)

(1.1). 编写 Java 应用程序, 实现浮点数 (float) 稀疏矩阵的乘法和加法运算, 其中稀疏矩阵是指矩阵中的绝大部分元素的值为 0。在命令行读入和输出矩阵中的元素的时候, 采用三元组的方式, 即行号、列号和数值, 例如“第 2 行、第 3 列、数值 3.2”表示为 2 3 3.2。要求以 1 个 3*4 的矩阵 ([1.2 0 0 0; 0 0 3.1 0; 0 0 0 2.2]) 和一个 4*5 的矩阵 ([0 1.1 1.5 0 0; 0 0 3.2 0 0; 0 1.3 0 0 -3.2; -1.0 6.2 0 0 0]) 相乘, 以及 1 个 3*4 的矩阵 ([1.2 0 0 0; 0 0 3.1 0; 0 0 0 2.2]) 和 1 个 3*4 的矩阵 ([1.2 -3 -5.3 0; 0 1 0.1 -0.4; 2 2 1 0.2]) 相加为例, 在报告中附上程序截图、完整的运行结果截图和简要文字说明。(10 分)

(1.2). 有 12 个国家 (美国、中华人民共和国、德国、日本、英国、印度、法国、意大利、加拿大、韩国、以色列、俄罗斯), 其属性有 name、GDP2023 和 Olympics2024, 分别表示国家名称、世界银行公布的 2023 年的国内生产总值 (单位: 百万美元) 和在 2024 年巴黎奥运会上获得的奖牌数量。

编写一个 Java 应用程序, 要求使用 TreeSet。(i) 按照 Olympics2024 从大到小排序输出这些国家的信息; (ii) 按照 GDP2023 从大到小排序输出这些国家的信息。要求以上(i)和(ii)两小题都通过以下两种方式实现: 通过实现 Comparator 接口或通过实现 Comparable 接口。在报告中附上程序截图、完整的运行结果截图和详细的文字说明。(15 分)

Part 2 (25 分)

(2.1) 将第 8 章讲义 (JavaPD-Ch08) 中的 5 个应用程序 (Example8_1, Example8_2, Example8_3, Example8_4, Example8_6) 在 Eclipse 中运行, 如运行结果不唯一, 则需要运行多次并至少得到两个不同的结果。对重要语句加上注释。在报告中附上程序截图、运行结果截图和简要文字说明 (对运行结果做出解释)。(5 分)

(2.2). 运行以下三个程序 (每个程序运行 5 次), 并对输出结果给出分析。在报告中附上程序截图和简要的文字说明 (包括对结果的分析)。(10 分)

程序 1:

```

// The task for printing a character a specified number of times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The number of times to repeat

    /** Construct a task with a specified character and number of
     *  times to print the character
     */
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    @Override /** Override the run() method to tell the system
     *  what task to perform
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
    private int lastNum;

    /** Construct a task for printing 1, 2, ..., n */
    public PrintNum(int n) {
        lastNum = n;
    }

    @Override /** Tell the thread how to run */
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}

public class TaskThreadDemo {
    public static void main(String[] args) {
        // Create tasks
        Runnable printA = new PrintChar('a', 100);
        Runnable printB = new PrintChar('b', 100);
        Runnable print100 = new PrintNum(100);

        // Create threads
        Thread thread1 = new Thread(printA);
        Thread thread2 = new Thread(printB);
        Thread thread3 = new Thread(print100);

        // Start threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}

```

程序 2:

```

// The task for printing a character a specified number of times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The number of times to repeat

    /** Construct a task with a specified character and number of
     * times to print the character
     */
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    @Override /** Override the run() method to tell the system
     * what task to perform
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
    private int lastNum;

    /** Construct a task for printing 1, 2, ..., n */
    public PrintNum(int n) {
        lastNum = n;
    }

    @Override /** Tell the thread how to run */
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}

```

```
import java.util.concurrent.*;
```

```

public class ExecutorDemo {
    public static void main(String[] args) {
        // Create a fixed thread pool with maximum three threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit runnable tasks to the executor
        executor.execute(new PrintChar('a', 100));
        executor.execute(new PrintChar('b', 100));
        executor.execute(new PrintNum(100));

        // Shut down the executor
        executor.shutdown();
    }
}

```

程序 3:

```
import java.util.concurrent.*;

public class AccountWithoutSync {
    private static Account account = new Account();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }

        executor.shutdown();

        // Wait until all tasks are finished
        while (!executor.isTerminated()) {
        }

        System.out.println("What is balance? " + account.getBalance());
    }

    // A thread for adding a penny to the account
    private static class AddAPennyTask implements Runnable {
        public void run() {
            account.deposit(1);
        }
    }

    // An inner class for account
    private static class Account {
        private int balance = 0;

        public int getBalance() {
            return balance;
        }

        public void deposit(int amount) {
            int newBalance = balance + amount;

            // This delay is deliberately added to magnify the
            // data-corruption problem and make it easy to see.
            try {
                Thread.sleep(5);
            }
            catch (InterruptedException ex) {
            }

            balance = newBalance;
        }
    }
}
```

(2.3) 第 8 章讲义 (JavaPD-Ch08) 中的第 5 个应用程序 (Example8_5) 存在线程间不同步的问题, 请修改该程序, 以解决不同步的问题。在报告中附上程序截图、运行结果截图和详细的文字说明 (包括设计的思路和合理性分析)。(10 分)

Part 3 (30 分)

(3.1). 编写 Java 应用程序实现如下功能: 第一个线程不停地随机生成[0,1)之间的浮点数 (float) 并输出到屏幕, 第二个线程将第一个线程输出的第 1-5 个浮点数的和与平均值输出到屏幕 (紧跟在第一个线程输出的第 5 个浮点数之后)、将第一个线程输出的第 6-10 个[0,1)之间的浮点数的和与平均值输出到屏幕 (紧跟在第一个线程输出的第 10 个浮点数之后)…。要求线程间实现通信。要求采用实现 Runnable 接口和 Thread 类的构造方法的方式创建线程, 而不是通过 Thread 类的子类的方式。在报告中附上程序截图、

运行结果截图和详细的文字说明（包括设计的思路和合理性分析）。（10 分）

(3.2). 编写 Java 应用程序实现如下功能：创建工作线程，模拟银行现金账户取款和存款操作。多个线程同时执行取款和存款操作时，如果不使用同步处理，会造成账户余额混乱，要求使用 `synchronized` 关键字同步代码块，以保证多个线程同时执行取款和存款操作时，银行现金账户取款和存款的有效和一致。要求采用实现 `Runnable` 接口和 `Thread` 类的构造方法的方式创建线程，而不是通过 `Thread` 类的子类的方式。在报告中附上程序截图（假设银行存款有 100 元，有 3 个取款线程和 2 个存款线程，每次取款和存款均为 10 元）、运行结果截图（显示每次存取款操作后的余额等信息，以说明线程间同步正确）和详细的文字说明。（10 分）

(3.3). 有一座南北向的桥，只能容纳一个人，桥的南边有 1000 个人（记为 $S_1, S_2, \dots, S_{1000}$ ）和桥的北边有 1000 个人（记为 $N_1, N_2, \dots, N_{1000}$ ），编写 Java 应用程序让这些到达对岸，每个人用一个线程表示，桥为共享资源，在过桥的过程中输出谁正在过桥（不同人之间用逗号隔开）。运行 10 次，分别统计南边的 1000 人和北边的 1000 人先全部到达对岸的次数（第 i 行输出格式为：第 i 次运行，南边/北边先完成过桥）。要求采用实现 `Runnable` 接口和 `Thread` 类的构造方法的方式创建线程，而不是通过 `Thread` 类的子类的方式。在报告中附上程序截图、运行结果截图和详细的文字说明（包括对结果的分析）。（10 分）

报告写作。要求：主要思路有明确的说明，重点代码有详细的注释，行文逻辑清晰可读性强，报告整体写作较为专业。（20 分）

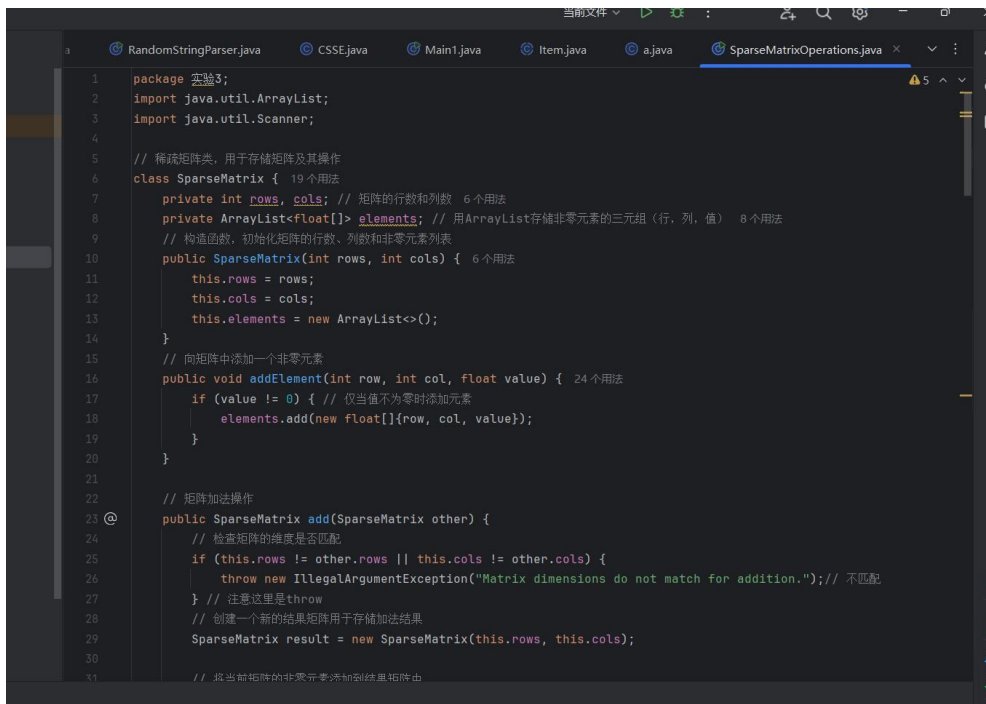
说明：

- (1) 本次实验课作业满分为 100 分，占总成绩的比例 7%。
- (2) 本次实验课作业截至时间 2024 年 11 月 20 日（周三）21:59。
- (3) 报告正文：请在**指定位置填写**，本次实验**不需要单独提交源程序文件**。
- (4) 个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的**姓名和学号**；实验报告的首页，请**准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”**等信息。
- (5) 提交方式：截至时间前，请在 Blackboard 平台中提交。
- (6) 发现抄袭（包括复制&粘贴整句话、整张图），**抄袭者和被抄袭者的成绩记零分**。
- (7) 延迟提交，不得分；如有特殊情况，请于截止日期之后的**48 小时内**发邮件到 panweike@szu.edu.cn，并在邮件中注明课程名称、作业名称、姓名、学号等信息，以及特殊情况的说明，我收到后会及时回复。
- (8) 期末考试阶段补交无效。

Part 1 (25 分)

(1.1). 编写 Java 应用程序，实现浮点数（float）稀疏矩阵的乘法和加法运算，其中稀疏矩阵是指矩阵中的绝大部分元素的值为 0。在命令行读入和输出矩阵中的元素的时候，采用三元组的方式，即行号、列号和数值，例如“第 2 行、第 3 列、数值 3.2”表示为 2 3 3.2。要求以 1 个 3*4 的矩阵 ([1.2 0 0 0; 0 0 3.1 0; 0 0 0 2.2]) 和一个 4*5 的矩阵 ([0 1.1 1.5 0 0; 0 0 3.2 0 0; 0 1.3 0 0 -3.2; -1.0 6.2 0 0 0]) 相乘，以及 1 个 3*4 的矩阵 ([1.2 0 0 0; 0 0 3.1 0; 0 0 0 2.2]) 和 1 个 3*4 的矩阵 ([1.2 -3 -5.3 0; 0 1 0.1 -0.4; 2 2 1 0.2]) 相加为例，在报告中附上程序截图、完整的运行结果截图和简要文字说明。（10 分）

• 程序截图



```
1 package 实验3;
2 import java.util.ArrayList;
3 import java.util.Scanner;
4
5 // 稀疏矩阵类，用于存储矩阵及其操作
6 class SparseMatrix { 19 个用法
7     private int rows, cols; // 矩阵的行数和列数 6 个用法
8     private ArrayList<float[]> elements; // 用 ArrayList 存储非零元素的三元组 (行, 列, 值) 8 个用法
9     // 构造函数，初始化矩阵的行数、列数和非零元素列表
10    public SparseMatrix(int rows, int cols) { 6 个用法
11        this.rows = rows;
12        this.cols = cols;
13        this.elements = new ArrayList<>();
14    }
15    // 向矩阵中添加一个非零元素
16    public void addElement(int row, int col, float value) { 24 个用法
17        if (value != 0) { // 仅当值不为零时添加元素
18            elements.add(new float[]{row, col, value});
19        }
20    }
21
22    // 矩阵加法操作
23    @ public SparseMatrix add(SparseMatrix other) {
24        // 检查矩阵的维度是否匹配
25        if (this.rows != other.rows || this.cols != other.cols) {
26            throw new IllegalArgumentException("Matrix dimensions do not match for addition."); // 不匹配
27        } // 注意这里是 throw
28        // 创建一个新的结果矩阵用于存储加法结果
29        SparseMatrix result = new SparseMatrix(this.rows, this.cols);
30
31        // 将当前矩阵的非零元素添加到结果矩阵中
```

• 完整代码

```
1. package 实验3;
2. import java.util.ArrayList;
3. import java.util.Scanner;
4.
5. // 稀疏矩阵类，用于存储矩阵及其操作
6. class SparseMatrix {
7.     private int rows, cols; // 矩阵的行数和列数
8.     private ArrayList<float[]> elements; // 用 ArrayList 存储非零
       元素的三元组 (行, 列, 值)
9.     // 构造函数，初始化矩阵的行数、列数和非零元素列表
10.    public SparseMatrix(int rows, int cols) {
11.        this.rows = rows;
12.        this.cols = cols;
13.        this.elements = new ArrayList<>();
14.    }
15.    // 向矩阵中添加一个非零元素
```

```
16.     public void addElement(int row, int col, float value) {
17.         if (value != 0) { // 仅当值不为零时添加元素
18.             elements.add(new float[]{row, col, value});
19.         }
20.     }
21.
22.     // 矩阵加法操作
23.     public SparseMatrix add(SparseMatrix other) {
24.         // 检查矩阵的维度是否匹配
25.         if (this.rows != other.rows || this.cols != other.cols)
26.         {
27.             throw new IllegalArgumentException("Matrix dimensions do not match for addition."); // 不匹配
28.         } // 注意这里是 throw
29.         // 创建一个新的结果矩阵用于存储加法结果
30.         SparseMatrix result = new SparseMatrix(this.rows, this.cols);
31.         // 将当前矩阵的非零元素添加到结果矩阵中
32.         for (float[] element : this.elements) {
33.             result.addElement((int) element[0], (int) element[1], element[2]); // 新矩阵加入非零元素
34.         }
35.
36.         // 将另一个矩阵的非零元素与当前矩阵相同位置的值相加并存储到结果矩阵中
37.         for (float[] element : other.elements) {
38.             result.addElement((int) element[0], (int) element[1],
39.                 result.getValue((int) element[0], (int) element[1]) + element[2]);
40.         }
41.         return result; // 返回相加后的结果矩阵
42.     }
43.
44.     // 矩阵乘法操作
45.     public SparseMatrix multiply(SparseMatrix other) {
46.         // 检查是否符合矩阵乘法的条件(当前矩阵的列数等于另一个矩阵的行数)
47.         if (this.cols != other.rows) {
48.             throw new IllegalArgumentException("Matrix dimensions do not match for multiplication.");
49.         } // throw
50.         // 创建一个新的结果矩阵用于存储乘法结果
```



```

51.         SparseMatrix result = new SparseMatrix(this.rows, other
.cols);
52.
53.         // 遍历当前矩阵的非零元素
54.         for (float[] elementA : this.elements) {
55.             // 遍历另一个矩阵的非零元素
56.             for (float[] elementB : other.elements) {
57.                 // 检查当前矩阵的列索引和另一个矩阵的行索引是否相同
(满足相乘条件)
58.                 if ((int) elementA[1] == (int) elementB[0]) {
59.                     int row = (int) elementA[0]; // 当前矩阵的行
索引
60.                     int col = (int) elementB[1]; // 另一个矩阵的
列索引
61.                     float value = elementA[2] * elementB[2]; //
计算相乘后的值
62.                     // 将乘积加到结果矩阵的相应位置
63.                     result.addElement(row, col, result.getValue
(row, col) + value);
64.                 }
65.             }
66.         }
67.         return result; // 返回相乘后的结果矩阵
68.     }
69.
70.     // 获取矩阵指定位置的元素值
71.     public float getValue(int row, int col) {
72.         // 遍历所有非零元素，查找是否存在指定位置的元素
73.         for (float[] element : elements) {
74.             if ((int) element[0] == row && (int) element[1] ==
col) {
75.                 return element[2]; // 返回找到的元素值
76.             }
77.         }
78.         return 0; // 如果没有找到该位置的元素，返回0
79.     }
80.
81.     // 打印矩阵的所有非零元素（行、列、值的三元组）
82.     public void printMatrix() {
83.         for (float[] element : elements) {
84.             System.out.println((int) element[0] + " " + (int) e
lement[1] + " " + element[2]);
85.         }
86.     }

```

```
87.
88.     // 从命令行输入读取矩阵的非零元素并生成稀疏矩阵
89.     public static SparseMatrix readMatrix(Scanner scanner, int
    rows, int cols) {
90.         SparseMatrix matrix = new SparseMatrix(rows, cols);
91.         System.out.println("Enter the number of non-zero elemen
    ts:");
92.         int nonZeroElements = scanner.nextInt(); // 读取非零元素
    的数量
93.         System.out.println("Enter elements in format: row col v
    alue");
94.
95.         // 循环读取每个非零元素的行、列和值
96.         for (int i = 0; i < nonZeroElements; i++) {
97.             int row = scanner.nextInt();
98.             int col = scanner.nextInt();
99.             float value = scanner.nextFloat();
100.            matrix.addElement(row, col, value); // 添加元素到矩
    阵中
101.        }
102.        return matrix; // 返回生成的矩阵
103.    }
104. }
105.
106. // 主类，用于执行矩阵的加法和乘法操作
107. public class SparseMatrixOperations {
108.     public static void main(String[] args) {
109.         Scanner scanner = new Scanner(System.in);
110.
111.         // 定义第一个 3x4 矩阵并初始化非零元素
112.         SparseMatrix matrixA = new SparseMatrix(3, 4);
113.         matrixA.addElement(0, 0, 1.2f);
114.         matrixA.addElement(1, 2, 3.1f);
115.         matrixA.addElement(2, 3, 2.2f);
116.
117.         // 定义第二个 4x5 矩阵并初始化非零元素
118.         SparseMatrix matrixB = new SparseMatrix(4, 5);
119.         matrixB.addElement(0, 1, 1.1f);
120.         matrixB.addElement(0, 2, 1.5f);
121.         matrixB.addElement(1, 2, 3.2f);
122.         matrixB.addElement(2, 1, 1.3f);
123.         matrixB.addElement(2, 4, -3.2f);
124.         matrixB.addElement(3, 0, -1.0f);
125.         matrixB.addElement(3, 1, 6.2f);
```

```

126.
127.      // 定义第三个 3x4 矩阵, 用于矩阵加法操作
128.      SparseMatrix matrixC = new SparseMatrix(3, 4);
129.      matrixC.addElement(0, 0, 1.2f);
130.      matrixC.addElement(0, 1, -3.0f);
131.      matrixC.addElement(0, 2, -5.3f);
132.      matrixC.addElement(1, 1, 1.0f);
133.      matrixC.addElement(1, 2, 0.1f);
134.      matrixC.addElement(1, 3, -0.4f);
135.      matrixC.addElement(2, 0, 2.0f);
136.      matrixC.addElement(2, 1, 2.0f);
137.      matrixC.addElement(2, 2, 1.0f);
138.      matrixC.addElement(2, 3, 0.2f);
139.
140.      // 打印矩阵 A
141.      System.out.println("Matrix A:");
142.      matrixA.printMatrix();
143.
144.      // 打印矩阵 B
145.      System.out.println("Matrix B:");
146.      matrixB.printMatrix();
147.
148.      // 打印矩阵 C
149.      System.out.println("Matrix C:");
150.      matrixC.printMatrix();
151.
152.      // 进行矩阵加法并打印结果
153.      System.out.println("\nMatrix A + Matrix C:");
154.      SparseMatrix resultAdd = matrixA.add(matrixC);
155.      resultAdd.printMatrix();
156.
157.      // 进行矩阵乘法并打印结果
158.      System.out.println("\nMatrix A * Matrix B:");
159.      SparseMatrix resultMultiply = matrixA.multiply(matrixB)
160.      ;
161.      resultMultiply.printMatrix();
162.
163.      // 关闭扫描器
164.      scanner.close();
165.  }
166. }

```

• 运行结果

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
```

```
Matrix A:
```

```
0 0 1.2
```

```
1 2 3.1
```

```
2 3 2.2
```

```
Matrix B:
```

```
0 1 1.1
```

```
0 2 1.5
```

```
1 2 3.2
```

```
2 1 1.3
```

```
2 4 -3.2
```

```
3 0 -1.0
```

```
3 1 6.2
```

```
Matrix C:
```

```
0 0 1.2
```

```
0 1 -3.0
```

```
0 2 -5.3
```

```
1 1 1.0
```

```
1 2 0.1
```

```
1 3 -0.4
```

```
2 0 2.0
```

```
2 1 2.0
```

```
2 2 1.0
```

```
2 3 0.2
```

```
Matrix A + Matrix C:
```

```
0 0 1.2
1 2 3.1
2 3 2.2
0 0 2.4
0 1 -3.0
0 2 -5.3
1 1 1.0
1 2 3.1999998
1 3 -0.4
2 0 2.0
2 1 2.0
2 2 1.0
2 3 2.4
```

```
Matrix A * Matrix B:
```

```
0 1 1.32
0 2 1.8000001
1 1 4.0299997
1 4 -9.92
2 0 -2.2
2 1 13.64
```

```
进程已结束，退出代码为 0
```

• 详细说明

类 `SparseMatrix`：实现了稀疏矩阵的存储和基本操作，包括添加元素、获取元素、矩阵加法和矩阵乘法。

方法 `addElement`：仅在元素不为零时添加到稀疏矩阵，避免存储多余的零值。

方法 `add`：进行矩阵加法，将两个矩阵对应位置的元素相加。

方法 `multiply`：实现矩阵乘法，依据矩阵的维度要求，将符合条件的元素相乘并累加到结果矩阵。

方法 `printMatrix`：以三元组格式输出矩阵的所有非零元素。

矩阵定义：

`Matrix A` 是一个 3×4 的稀疏矩阵，包含非零元素。

`Matrix B` 是一个 4×5 的稀疏矩阵，用于乘法。

`Matrix C` 是另一个 3×4 的稀疏矩阵，用于加法。

矩阵运算结果：

加法：`Matrix A + Matrix C` 的结果输出为新的稀疏矩阵，显示每个非零元素的行、列和值。

乘法：`Matrix A * Matrix B` 的结果输出也为稀疏矩阵格式。

命令行交互：程序在命令行中以三元组形式展示结果，如行号、列号和数值。

(1.2). 有 12 个国家（美国、中华人民共和国、德国、日本、英国、印度、法国、意大利、加拿大、韩国、以色列、俄罗斯），其属性有 name、GDP2023 和 Olympics2024，分别表示国家名称、世界银行公布的 2023 年的国内生产总值（单位：百万美元）和在 2024 年巴黎奥会上获得的奖牌数量。

编写一个 Java 应用程序，要求使用 TreeSet。（i）按照 Olympics2024 从大到小排序输出这些国家的信息；（ii）按照 GDP2023 从大到小排序输出这些国家的信息。要求以上(i)和(ii)两小题都通过以下两种方式实现：通过实现 Comparator 接口或通过实现 Comparable 接口。在报告中附上程序截图、完整的运行结果截图和详细的文字说明。（15 分）

• 资料检索

(1.2). 有 12 个国家（美国、中华人民共和国、德国、日本、英国、印度、法国、意大利、加拿大、韩国、以色列、俄罗斯），其属性有 name、GDP2023 和 Olympics2024.							
2023年世界国家和地区GDP总量（IMF版）							
数据来自国际货币基金组织（IMF）2024年4月发布				制表：Gong众号：冲之星云			
		2023年GDP总量 (亿美元)	2022年GDP总量 (亿美元)	名义增量	名义增速	人口 (万)	2023年人均GDP (美元)
1	美国	273578	257441	16137	6.3%	33514	81632
2	中国	176620	178485	-1865	-1.0%	141140	12514
3	德国	44574	40857	3717	9.1%	8454	52727
4	日本	42129	42564	-435	-1.0%	12462	33806
5	印度	35721	33535	2186	6.5%	142863	2500
6	英国	33447	31001	2446	7.9%	6812	49099
7	法国	30318	27804	2513	9.0%	6591	46001
8	意大利	22555	20686	1869	9.0%	5885	38326
9	巴西	21737	19518	2218	11.4%	20425	10642
10	加拿大	21401	21615	-214	-1.0%	3997	53548
11	俄罗斯	19970	22723	-2752	-12.1%	14633	13648
12	墨西哥	17889	14633	3256	22.2%	13114	13642
13	澳大利亚	17419	17249	170	1.0%	2662	65434
14	韩国	17128	16739	389	2.3%	5160	33192
15	西班牙	15812	14189	1622	11.4%	4781	33071
16	印度尼西亚	13712	13191	521	3.9%	27743	4942
17	荷兰	11171	10102	1069	10.6%	1781	62719
18	土耳其	11085	9058	2026	22.4%	8627	12849
19	沙特阿拉伯	10676	11086	-410	-3.7%	3282	32530
20	瑞士	8851	8186	665	8.1%	882	100413

一、以色列GDP情况

华经产业研究院数据显示：2023年世界GDP为1054350.4亿美元，同比增长2.72%；以色列GDP为5099.01亿美元，比上年减少了151.01亿美元，同比增长2%；与2010年相比增长了2715.37亿美元，相比2010年下降了3.67个百分点。



顺序	NOC	金	银	铜	
1	美利坚合众国	40	44	42	126
2	中国	40	27	24	91
3	日本	20	12	13	45
4	澳大利亚	18	19	16	53
5	法国	16	26	22	64
6	荷兰	15	7	12	34
7	英国	14	22	29	65
8	大韩民国	13	9	10	32
9	意大利	12	13	15	40

10	德国	12	13	8	33
11	新西兰	10	7	3	20
12	加拿大	9	7	11	27

国内版国际版

Microsoft Bing

2024巴黎奥运会以色列奖牌

网页 图片 视频 学术 词典 地图 更多 工具

约 1,770,000 个结果

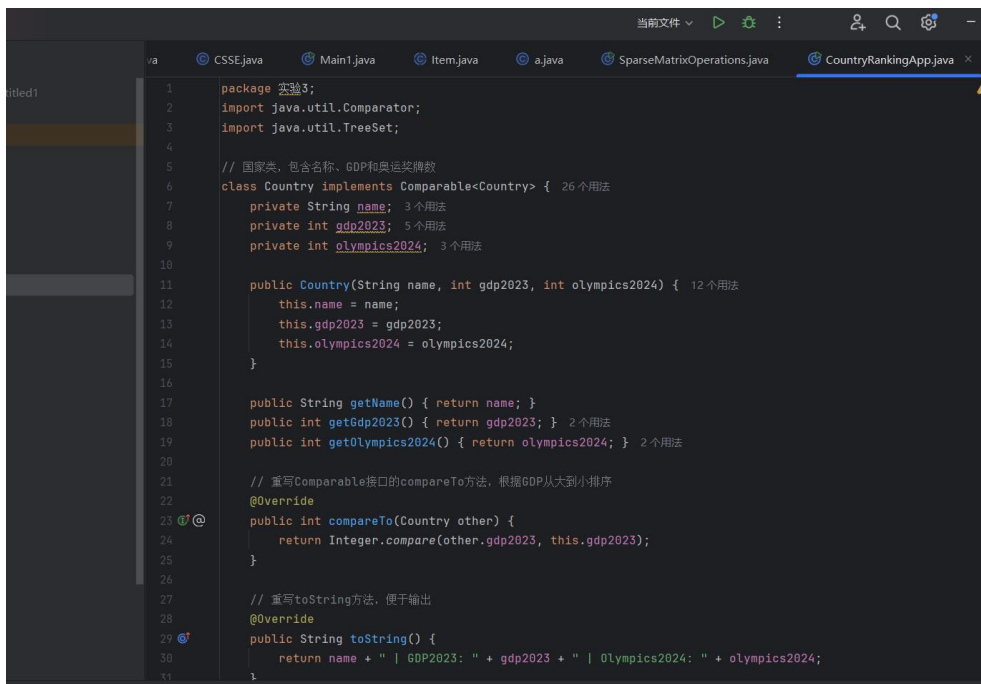
10枚奖牌

在2024年巴黎奥运会上，在战争背景下，以色列运动员获得了**10枚奖牌**，成为20年来最好的参赛成绩。今年，以色列代表团超越了2020年东京奥运会的成绩，还赢得了历史性的奖牌，例如女子门球银牌，这是36年来在团体项目中获得的第一枚奖牌。

以色列以 10 枚奖牌结束了历史性的巴黎奥运会。
aurora-israel.co.il/zh-CN/%E4%BB%A5%E8%89%B2%E5%88%97%E5%9C%...

注意：俄罗斯和奖牌数设为0（由于俄乌冲突）。

• 程序截图



```
1 package 实验3;
2 import java.util.Comparator;
3 import java.util.TreeSet;
4
5 // 国家类，包含名称、GDP和奥运奖牌数
6 class Country implements Comparable<Country> { 26个用法
7     private String name; 3个用法
8     private int gdp2023; 5个用法
9     private int olympics2024; 3个用法
10
11     public Country(String name, int gdp2023, int olympics2024) { 12个用法
12         this.name = name;
13         this.gdp2023 = gdp2023;
14         this.olympics2024 = olympics2024;
15     }
16
17     public String getName() { return name; }
18     public int getGdp2023() { return gdp2023; } 2个用法
19     public int getOlympics2024() { return olympics2024; } 2个用法
20
21     // 重写Comparable接口的compareTo方法，根据GDP从大到小排序
22     @Override
23     public int compareTo(Country other) {
24         return Integer.compare(other.gdp2023, this.gdp2023);
25     }
26
27     // 重写toString方法，便于输出
28     @Override
29     public String toString() {
30         return name + " | GDP2023: " + gdp2023 + " | Olympics2024: " + olympics2024;
31     }
32 }
```

• 完整代码

```
1. package 实验3;
2. import java.util.Comparator;
3. import java.util.TreeSet;
4.
5. // 国家类，包含名称、GDP 和奥运奖牌数
6. class Country implements Comparable<Country> {
7.     private String name;
8.     private int gdp2023;
9.     private int olympics2024;
10.
11.     public Country(String name, int gdp2023, int olympics2024)
12.     {
13.         this.name = name;
14.         this.gdp2023 = gdp2023;
15.         this.olympics2024 = olympics2024;
16.     }
17.
18.     public String getName() { return name; }
19.     public int getGdp2023() { return gdp2023; }
20.     public int getOlympics2024() { return olympics2024; }
21.
22.     // 重写Comparable 接口的 compareTo 方法，根据 GDP 从大到小排序
23.     @Override
24.     public int compareTo(Country other) {
25.         return Integer.compare(other.gdp2023, this.gdp2023);
26.     }
27. }
```

```
26.
27.     // 重写 toString 方法，便于输出
28.     @Override
29.     public String toString() {
30.         return name + " | GDP2023: " + gdp2023 + " | Olympics20
31.         24: " + olympics2024;
32.     }
33.
34.     // 自定义 Comparator，用于根据奥运奖牌数从大到小排序
35.     class OlympicsComparator implements Comparator<Country> {
36.         @Override
37.         public int compare(Country c1, Country c2) {
38.             int result = Integer.compare(c2.getOlympics2024(), c1.g
39.             etOlympics2024());
40.             if (result == 0) {
41.                 // 如果奖牌数相同，进一步按国家名称排序以保证唯一性
42.                 return c1.getName().compareTo(c2.getName());
43.             }
44.             return result;
45.         }
46.
47.         // 自定义 Comparator，用于根据 GDP 从大到小排序
48.         class GDPComparator implements Comparator<Country> {
49.             @Override
50.             public int compare(Country c1, Country c2) {
51.                 int result = Integer.compare(c2.getGdp2023(), c1.getGdp
52.                 2023());
53.                 if (result == 0) {
54.                     // 如果 GDP 相同，进一步按国家名称排序以保证唯一性
55.                     return c1.getName().compareTo(c2.getName());
56.                 }
57.                 return result;
58.             }
59.         }
60.         // 主类
61.         public class CountryRankingApp {
62.             public static void main(String[] args) {
63.                 // 创建国家对象并添加到集合中
64.                 TreeSet<Country> countriesByOlympics = new TreeSet<>(ne
65.                 w OlympicsComparator());
66.                 TreeSet<Country> countriesByGDP = new TreeSet<>(new GDP
67.                 Comparator());
```

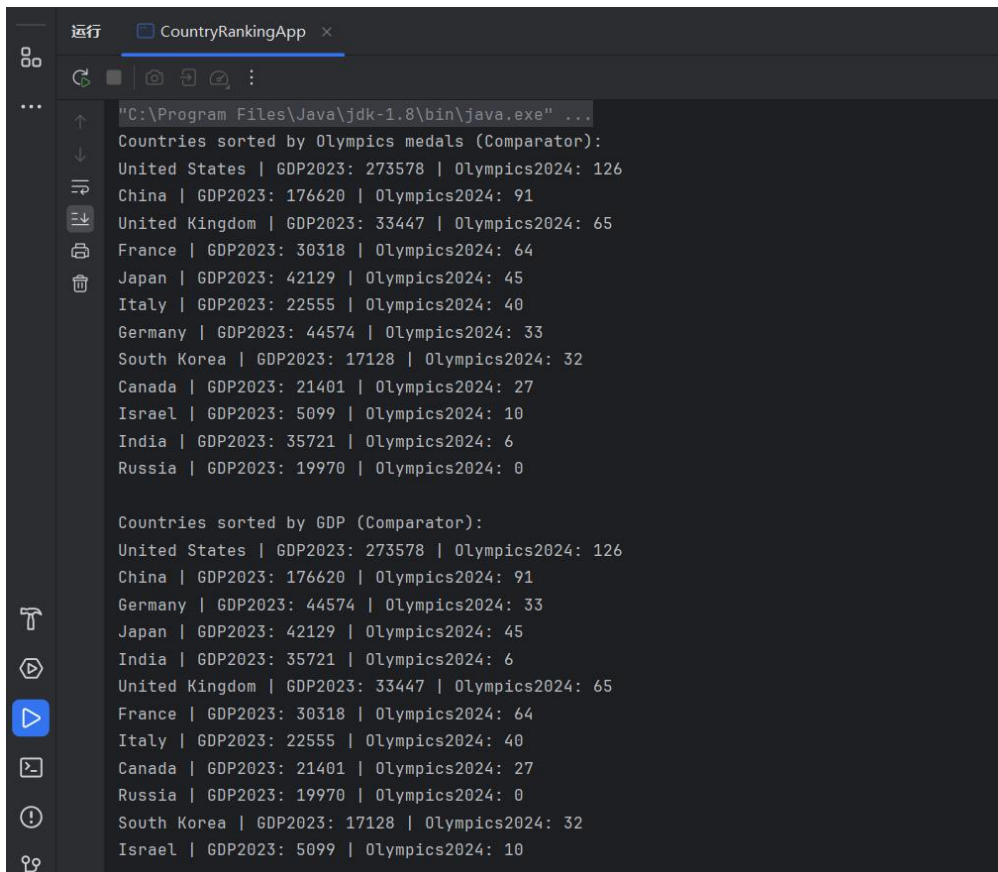
```
65.
66.         // 添加国家数据
67.         countriesByOlympics.add(new Country("United States", 27
3578, 126));
68.         countriesByOlympics.add(new Country("China", 176620, 91
));
69.         countriesByOlympics.add(new Country("Germany", 44574, 3
3));
70.         countriesByOlympics.add(new Country("Japan", 42129, 45)
);
71.         countriesByOlympics.add(new Country("United Kingdom", 3
3447, 65));
72.         countriesByOlympics.add(new Country("India", 35721, 6))
;
73.         countriesByOlympics.add(new Country("France", 30318, 64
));
74.         countriesByOlympics.add(new Country("Italy", 22555, 40)
);
75.         countriesByOlympics.add(new Country("Canada", 21401, 27
));
76.         countriesByOlympics.add(new Country("South Korea", 1712
8, 32));
77.         countriesByOlympics.add(new Country("Israel", 5099, 10)
);
78.         countriesByOlympics.add(new Country("Russia", 19970, 0)
);
79.
80.         // 使用OlympicsComparator 按奥运奖牌数从大到小排序输出
81.         System.out.println("Countries sorted by Olympics medals
(Comparator):");
82.         for (Country country : countriesByOlympics) {
83.             System.out.println(country);
84.         }
85.
86.         System.out.println("\nCountries sorted by GDP (Comparat
or):");
87.         // 使用GDPComparator 按GDP 从大到小排序
88.         countriesByGDP.addAll(countriesByOlympics);
89.         for (Country country : countriesByGDP) {
90.             System.out.println(country);
91.         }
92.
93.         // 使用Comparable 接口按GDP 排序
94.         System.out.println("\nCountries sorted by GDP (Comparab
```

```

        le:");
95.         TreeSet<Country> countriesByComparable = new TreeSet<>(  
            countriesByOlympics);
96.         for (Country country : countriesByComparable) {
97.             System.out.println(country);
98.         }
99.     }
100. }

```

• 运行结果

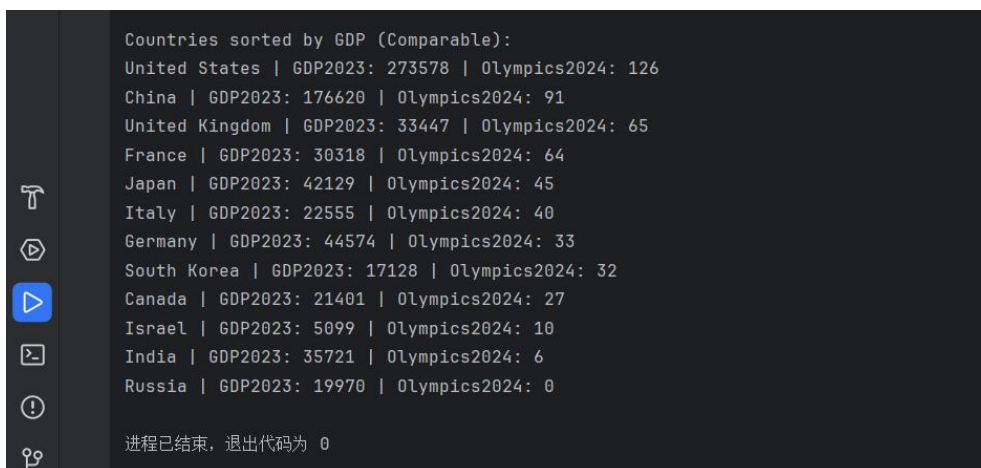


```

运行 CountryRankingApp x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Countries sorted by Olympics medals (Comparator):
United States | GDP2023: 273578 | Olympics2024: 126
China | GDP2023: 176620 | Olympics2024: 91
United Kingdom | GDP2023: 33447 | Olympics2024: 65
France | GDP2023: 30318 | Olympics2024: 64
Japan | GDP2023: 42129 | Olympics2024: 45
Italy | GDP2023: 22555 | Olympics2024: 40
Germany | GDP2023: 44574 | Olympics2024: 33
South Korea | GDP2023: 17128 | Olympics2024: 32
Canada | GDP2023: 21401 | Olympics2024: 27
Israel | GDP2023: 5099 | Olympics2024: 10
India | GDP2023: 35721 | Olympics2024: 6
Russia | GDP2023: 19970 | Olympics2024: 0

Countries sorted by GDP (Comparator):
United States | GDP2023: 273578 | Olympics2024: 126
China | GDP2023: 176620 | Olympics2024: 91
Germany | GDP2023: 44574 | Olympics2024: 33
Japan | GDP2023: 42129 | Olympics2024: 45
India | GDP2023: 35721 | Olympics2024: 6
United Kingdom | GDP2023: 33447 | Olympics2024: 65
France | GDP2023: 30318 | Olympics2024: 64
Italy | GDP2023: 22555 | Olympics2024: 40
Canada | GDP2023: 21401 | Olympics2024: 27
Russia | GDP2023: 19970 | Olympics2024: 0
South Korea | GDP2023: 17128 | Olympics2024: 32
Israel | GDP2023: 5099 | Olympics2024: 10

```



```

Countries sorted by GDP (Comparable):
United States | GDP2023: 273578 | Olympics2024: 126
China | GDP2023: 176620 | Olympics2024: 91
United Kingdom | GDP2023: 33447 | Olympics2024: 65
France | GDP2023: 30318 | Olympics2024: 64
Japan | GDP2023: 42129 | Olympics2024: 45
Italy | GDP2023: 22555 | Olympics2024: 40
Germany | GDP2023: 44574 | Olympics2024: 33
South Korea | GDP2023: 17128 | Olympics2024: 32
Canada | GDP2023: 21401 | Olympics2024: 27
Israel | GDP2023: 5099 | Olympics2024: 10
India | GDP2023: 35721 | Olympics2024: 6
Russia | GDP2023: 19970 | Olympics2024: 0

进程已结束，退出代码为 0

```

• 详细说明

总体实现了一个国家排名应用程序 `CountryRankingApp`，其中展示了如何使用 `TreeSet` 数据结构结合 `Comparator` 和 `Comparable` 来按不同的条件（如 GDP 和奥运奖牌数）对国家进行排序。

Country 类：

表示一个国家，包含三个属性：名称、2023 年的 GDP 值和 2024 年的奥运奖牌数。
`Country` 类实现了 `Comparable<Country>` 接口，以便能够按照 GDP 值进行排序。

属性：

- **name:** 国家的名称（字符串类型）。
- **gdp2023:** 2023 年该国家的 GDP 值（整数类型）。
- **olympics2024:** 2024 年该国家获得的奥运奖牌数（整数类型）。

构造函数：

- **Country(String name, int gdp2023, int olympics2024):** 用于初始化国家的名称、GDP 和奥运奖牌数。

Getter 方法：

- **getName():** 返回国家名称。
- **getGdp2023():** 返回国家的 GDP 值。
- **getOlympics2024():** 返回国家的奥运奖牌数。

compareTo 方法：

- **Comparable<Country>**接口的实现，用于按照 GDP 从大到小对国家进行排序。
- 使用 `Integer.compare(other.gdp2023, this.gdp2023)` 实现，从大到小的排序规则。

toString 方法：

- 用于返回一个字符串表示该国家的所有信息，包括国家名称、GDP 和奥运奖牌数。
方便在输出时查看该对象的详细信息。

OlympicsComparator 类：

一个自定义的 `Comparator<Country>` 实现，用于根据国家的奥运奖牌数从大到小对国家进行排序。

compare 方法：

- 首先，按奥运奖牌数进行比较，使用 `Integer.compare(c2.getOlympics2024(), c1.getOlympics2024())`，确保按奖牌数从大到小排序。
- 如果两个国家的奥运奖牌数相同，进一步根据国家名称（字母顺序）进行排序，使用 `c1.getName().compareTo(c2.getName())`。

GDPComparator 类：

一个自定义的 `Comparator<Country>` 实现，用于根据国家的 GDP 从大到小进行排序。

compare 方法：

- 按 GDP 进行比较，使用 `Integer.compare(c2.getGdp2023(), c1.getGdp2023())`，确保按 GDP 从大到小排序。
- 如果两个国家的 GDP 相同，进一步按国家名称排序。

CountryRankingApp 类（主类）：

应用程序的入口点，包含 `main` 方法，负责创建国家对象并按照不同的排序规则进行排序和输出。

步骤：

1) 创建集合：

- 创建一个 `TreeSet` 集合，用于存储按奥运奖牌数排序的国家。`TreeSet` 会根据

提供的 `Comparator` 进行自动排序。

- `TreeSet<Country> countriesByGDP = new TreeSet<>(new GDPComparator());`: 创建一个 `TreeSet` 集合，用于存储按 GDP 排序的国家。

2) 添加数据:

- 通过 `countriesByOlympics.add()` 和 `countriesByGDP.addAll()` 方法向集合中添加 'Country' 对象。每个国家对象包含名称、GDP 值和奥运奖牌数。

3) 输出按奥运奖牌数排序的结果:

- 使用 `OlympicsComparator` 排序，遍历 `countriesByOlympic` 集合，按奥运奖牌数从大到小打印国家信息。

4) 输出按 GDP 排序的结果:

- 将 `countriesByOlympics` 集合中的国家数据添加到 `countriesByGDP` 集合，使用 `GDPComparator` 对其进行排序，并打印结果。

5) 输出使用 `Comparable` 排序的结果:

- 创建一个新的 `TreeSet<Country> countriesByComparable`，并传入 `countriesByOlympics`（因为 `Country` 类实现了 `Comparable` 接口，默认按 GDP 排序）。
- 打印 `countriesByComparable` 中的国家信息。
- `Comparable` 和 `Comparator` 在 Java 中用于自定义对象的排序规则。
- `Comparable` 是对象自身决定排序规则，而 `Comparator` 是外部提供排序规则。
- `TreeSet` 会根据给定的排序规则自动对元素进行排序，因此可以灵活使用不同的排序方式（通过 `Comparator` 或 `Comparable`）。

Part 2 (25 分)

(2.1) 将第 8 章讲义 (JavaPD-Ch08) 中的 5 个应用程序 (`Example8_1`, `Example8_2`, `Example8_3`, `Example8_4`, `Example8_6`) 在 Eclipse 中运行，如运行结果不唯一，则需要运行多次并至少得到两个不同的结果。对重要语句加上注释。在报告中附上程序截图、运行结果截图和简要文字说明（对运行结果做出解释）。（5 分）

- 程序截图

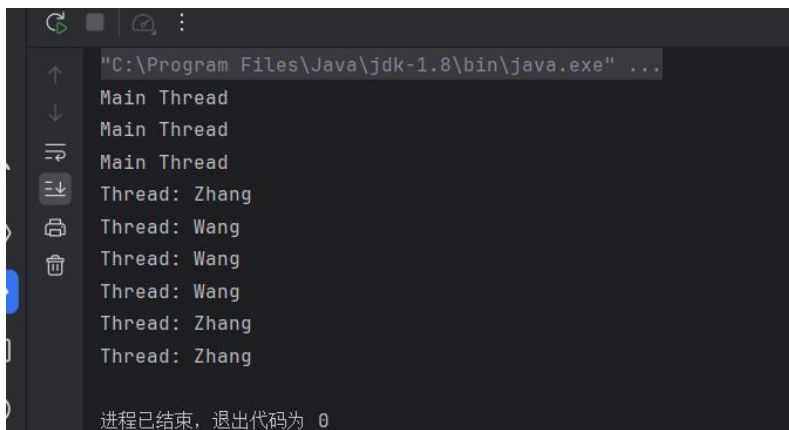
-example8_1:

```

1 package 实验3;
2
3 class WriteWordThread extends Thread { 3个用法
4     WriteWordThread(String s) { 2个用法
5         setName(s);
6     }
7
8     public void run() {
9         for (int i = 1; i <= 3; i++)
10             System.out.println("Thread: " + getName());
11     }
12 }
13 public class Example8_1
14 {
15     public static void main(String args[])
16     {
17         WriteWordThread zhang, wang;
18         zhang = new WriteWordThread(s: "Zhang"); //新建线程
19         wang = new WriteWordThread(s: "Wang"); //新建线程
20         zhang.start(); //启动线程
21         for(int i=1; i<=3; i++)
22         {
23             System.out.println("Main Thread");
24         }
25         wang.start(); //启动线程
26     }
27 }

```

• 运行结果:



```

"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Main Thread
Main Thread
Main Thread
Thread: Zhang
Thread: Wang
Thread: Wang
Thread: Wang
Thread: Zhang
Thread: Zhang
进程已结束，退出代码为 0

```



```
运行 Example8_1 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Main Thread
Main Thread
Main Thread
Thread: Zhang
Thread: Zhang
Thread: Zhang
Thread: Wang
Thread: Wang
Thread: Wang
进程已结束，退出代码为 0
```

```
运行 Example8_1 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Main Thread
Main Thread
Main Thread
Thread: Wang
Thread: Wang
Thread: Wang
Thread: Zhang
Thread: Zhang
Thread: Zhang
进程已结束，退出代码为 0
```

• 代码注释+说明:

```
1. package 实验3;
2. // 定义一个继承自 Thread 的类，表示一个写字的线程
3. class WriteWordThread extends Thread {
4.     // 构造函数，接受一个字符串参数并将其设置为线程的名字
5.     WriteWordThread(String s) {
6.         setName(s); // 设置线程名称
7.     }
8.     // 重写 run 方法，run 方法中的代码就是线程启动后要执行的内容
9.     public void run() {
10.        // 在 run 方法中，循环打印三次线程名称
11.        for (int i = 1; i <= 3; i++) {
12.            // 输出当前线程的名字
13.            System.out.println("Thread: " + getName());
14.        }
15.    }
16. }
17. public class Example8_1 {
18.     public static void main(String args[]) {
19.        // 创建两个 WriteWordThread 对象，分别代表张和王
20.        WriteWordThread zhang, wang;
21.        zhang = new WriteWordThread("Zhang"); // 创建一个名为
```

Zhang 的线程

```
22.         wang = new WriteWordThread("Wang"); // 创建一个名为
           Wang 的线程
23.         // 启动张线程
24.         zhang.start(); // 调用 start() 方法启动线程，线程开始执行
           run() 方法中的内容
25.         // 主线程打印三次"Main Thread"
26.         for (int i = 1; i <= 3; i++) {
27.             System.out.println("Main Thread");
28.         }
29.         // 启动王线程
30.         wang.start(); // 启动王线程，执行王线程的 run() 方法
31.     }
32. }
```

WriteWordThread 类:

继承自 Thread，意味着它是一个可以独立运行的线程。

在构造函数中，通过 setName() 方法设置线程的名字。线程名字会在输出中显示出来，方便区分不同线程的执行。

run() 方法是线程的执行体，其中包含了一个 for 循环，循环打印出当前线程的名字 3 次。

Example8_1 类:

main 方法是程序的入口。在这里，创建了两个线程对象 zhang 和 wang。

调用 zhang.start() 启动了 zhang 线程，线程开始执行 run() 方法中的代码（打印线程名字）。

主线程进入 for 循环，打印 "Main Thread" 三次。

然后调用 wang.start() 启动了 wang 线程，线程同样开始执行自己的 run() 方法。

- 程序截图

Example8_2:

```

1 package 实验3;
2 class WriteWordThread extends Thread { 3个用法
3     int n = 0; 2个用法
4     WriteWordThread(String s, int n) { 2个用法
5         setName(s);
6         this.n = n;
7     }
8     public void run() {
9         for (int i = 1; i <= 3; i++) {
10             System.out.println("Thread: " + getName());
11             try {
12                 sleep(n);
13             } catch (InterruptedException e) {
14             }
15         }
16     }
17 }
18 public class Example8_2
19 {
20     public static void main(String args[])
21     {
22         WriteWordThread zhang, wang;
23         zhang = new WriteWordThread("Zhang", 200);
24         wang = new WriteWordThread("Wang", 100);
25         zhang.start();
26         wang.start();
27     }
28 }

```

- 运行代码

```

"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Thread: Wang
Thread: Zhang
Thread: Wang
Thread: Zhang
Thread: Wang
Thread: Zhang

进程已结束，退出代码为 0

```

```

运行 Example8_2 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Thread: Wang
Thread: Zhang
Thread: Wang
Thread: Wang
Thread: Zhang
Thread: Zhang

进程已结束，退出代码为 0

```

- 完整代码

1. package 实验3;

```

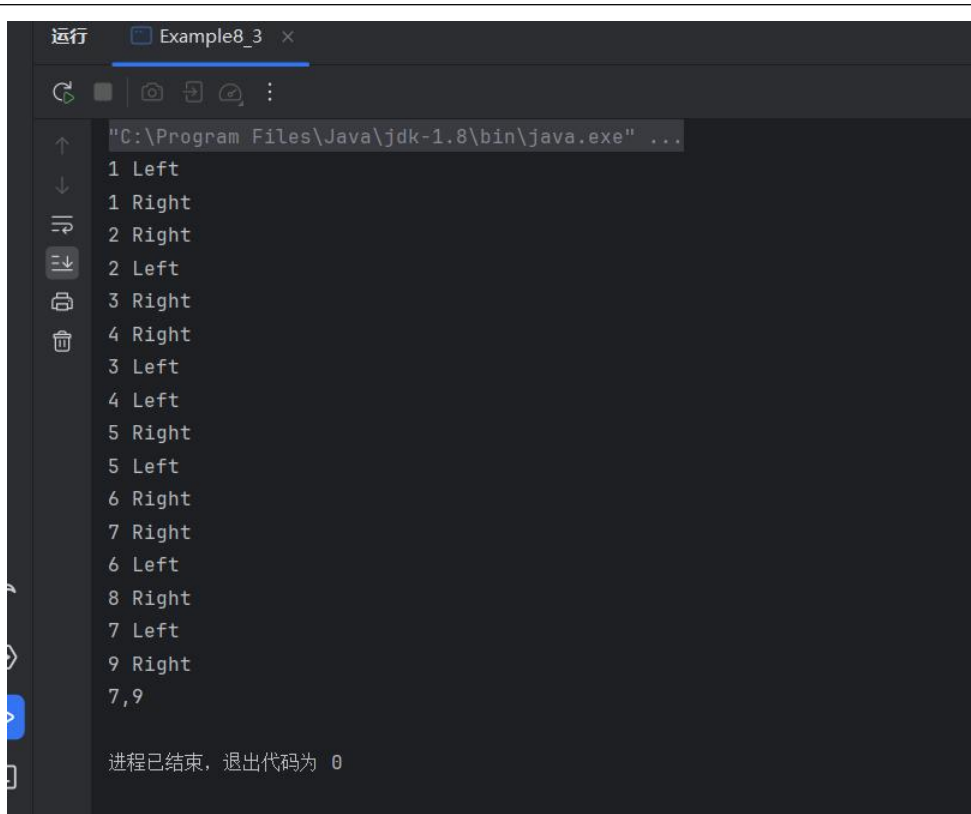
2.
3.  class WriteWordThread extends Thread {
4.      int n = 0;
5.
6.      // 构造函数，设置线程名称和延迟时间
7.      WriteWordThread(String s, int n) {
8.          setName(s); // 设置线程名称
9.          this.n = n; // 线程休眠时间
10.     }
11.
12.     public void run() {
13.         for (int i = 1; i <= 3; i++) {
14.             System.out.println("Thread: " + getName()); // 输出
线程名称
15.             try {
16.                 sleep(n); // 线程休眠，模拟不同线程的延迟
17.             } catch (InterruptedException e) {
18.                 e.printStackTrace();
19.             }
20.         }
21.     }
22. }
23.
24. public class Example8_2 {
25.     public static void main(String args[]) {
26.         // 创建两个线程对象，设置不同的休眠时间
27.         WriteWordThread zhang = new WriteWordThread("Zhang", 20
0); // 张线程，延迟 200ms
28.         WriteWordThread wang = new WriteWordThread("Wang", 100)
; // 王线程，延迟 100ms
29.
30.         // 启动线程
31.         zhang.start(); // 启动张线程
32.         wang.start(); // 启动王线程
33.     }
34. }

```

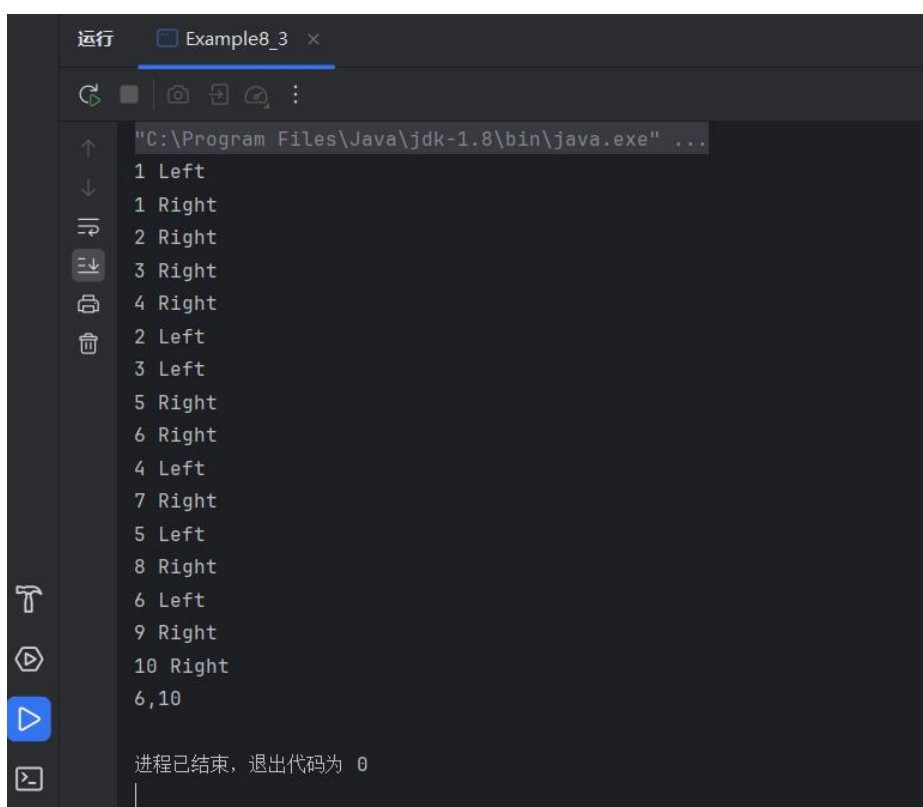
- 程序截图

-Example8_3

- 运行结果



```
运行 Example8_3 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
1 Left
1 Right
2 Right
2 Left
3 Right
4 Right
3 Left
4 Left
5 Right
5 Left
6 Right
7 Right
6 Left
8 Right
7 Left
9 Right
7,9
进程已结束，退出代码为 0
```



```
运行 Example8_3 x
"C:\bin\java.exe" ...
1 Left
1 Right
2 Right
3 Right
4 Right
2 Left
3 Left
5 Right
6 Right
4 Left
7 Right
5 Left
8 Right
6 Left
9 Right
10 Right
6,10
进程已结束，退出代码为 0
```

• 完整代码

1. `package` 实验 3;
- 2.
3. `class` Left `extends` Thread {

```
4.     int n = 0;
5.
6.     public void run() {
7.         while (true) {
8.             n++; // 递增计数器
9.             System.out.println(n + " Left"); // 输出计数器值
10.            try {
11.                sleep((int) (Math.random() * 100)); // 随机休眠
时间
12.            } catch (InterruptedException e) {
13.                e.printStackTrace();
14.            }
15.        }
16.    }
17. }
18.
19. class Right extends Thread {
20.     int n = 0;
21.
22.     public void run() {
23.         while (true) {
24.             n++; // 递增计数器
25.             System.out.println(n + " Right"); // 输出计数器值
26.             try {
27.                 sleep((int) (Math.random() * 100)); // 随机休眠
时间
28.             } catch (InterruptedException e) {
29.                 e.printStackTrace();
30.             }
31.        }
32.    }
33. }
34.
35. public class Example8_3 {
36.     public static void main(String args[]) {
37.         // 创建并启动线程
38.         Left left = new Left();
39.         Right right = new Right();
40.         left.start();
41.         right.start();
42.
43.         // 检查计数器条件，确保某一线程计数达到8
44.         while (true) {
45.             try {
```

```

46.         Thread.sleep(100);
47.     } catch (InterruptedException e) {
48.         e.printStackTrace();
49.     }
50.     if (left.n >= 8 || right.n >= 8) {
51.         System.out.println(left.n + "," + right.n); //
        输出最终计数值
52.         System.exit(0); // 终止程序
53.     }
54. }
55. }
56. }

```

- 程序截图

-example8_4

```

1  package 实验3;
2  class TaskBank implements Runnable 2个用法
3  {
4      private int money = 0; String name1,name2; 7个用法
5      TaskBank(String s1, String s2){ name1 = s1; name2 = s2; } 1个用法
6      public void setMoney(int mount){ money = mount; } 1个用法
7      public void run()
8      {
9          while(true)
10         {
11             money = money-10;
12             if(Thread.currentThread().getName().equals(name1)){
13                 System.out.println(name1 + ": " + money);
14                 if(money<=100){
15                     System.out.println(name1 + ": Finished");
16                     return;
17                 }
18             }
19             else
20                 if(Thread.currentThread().getName().equals(name2)){
21                     System.out.println(name2 + ": " + money);
22                     if(money<=60){
23                         System.out.println(name2 + ": Finished");
24                         return;

```

- 运行结果

运行 Example8_4 x

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
treasurer zhang: 100
treasurer zhang: Finished
cashier cheng: 100
cashier cheng: 90
cashier cheng: 80
cashier cheng: 70
cashier cheng: 60
cashier cheng: Finished

进程已结束，退出代码为 0
```

Example8_4 x

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
cashier cheng: 100
treasurer zhang: 100
treasurer zhang: Finished
cashier cheng: 90
cashier cheng: 80
cashier cheng: 70
cashier cheng: 60
cashier cheng: Finished

进程已结束，退出代码为 0
```

• 完整代码

```
1. package 实验3;
2.
3. class TaskBank implements Runnable {
4.     private int money = 0;
5.     String name1, name2;
6.
7.     TaskBank(String s1, String s2) {
8.         name1 = s1; // 线程名称
```

```
9.         name2 = s2;
10.     }
11.
12.     public void setMoney(int amount) {
13.         money = amount; // 设置初始金额
14.     }
15.
16.     public void run() {
17.         while (true) {
18.             money -= 10; // 每次减少10
19.             if (Thread.currentThread().getName().equals(name1))
20.             {
21.                 System.out.println(name1 + ": " + money); // 输出当前金额
22.                 if (money <= 100) {
23.                     System.out.println(name1 + ": Finished");
24.                     return;
25.                 }
26.             } else if (Thread.currentThread().getName().equals(name2)) {
27.                 System.out.println(name2 + ": " + money); // 输出当前金额
28.                 if (money <= 60) {
29.                     System.out.println(name2 + ": Finished");
30.                     return;
31.                 }
32.             }
33.             try {
34.                 Thread.sleep(800); // 休眠
35.             } catch (InterruptedException e) {
36.                 e.printStackTrace();
37.             }
38.         }
39.     }
40.
41.     public class Example8_4 {
42.         public static void main(String args[]) {
43.             // 创建任务并设置初始金额
44.             String s1 = "treasurer zhang";
45.             String s2 = "cashier cheng";
46.             TaskBank taskBank = new TaskBank(s1, s2);
47.             taskBank.setMoney(120);
48.         }
```

```

49.         // 创建并启动线程
50.         Thread zhang = new Thread(taskBank);
51.         Thread cheng = new Thread(taskBank);
52.         zhang.setName(s1);
53.         cheng.setName(s2);
54.         zhang.start();
55.         cheng.start();
56.     }
57. }

```

- 程序截图

-example8_6:

```

package 实验3;

class TaskAddSub implements Runnable { 2个用法
    String s1, s2; 4个用法
    TaskAddSub(String s1, String s2) { 1个用法
        this.s1 = s1;
        this.s2 = s2;
    }
    public void run() {
        int i = 0;
        while (true) {
            if (Thread.currentThread().getName().equals(s1)) {
                i += 1; // i值增加
                System.out.println(s1 + ":" + i);
                if (i >= 4) {
                    System.out.println(s1 + " Finished");
                    return;
                }
            } else if (Thread.currentThread().getName().equals(s2)) {
                i -= 1; // i值减少
                System.out.println(s2 + ":" + i);
                if (i <= -4) {
                    System.out.println(s2 + " Finished");
                    return;
                }
            }
            try {
                Thread.sleep(800); // 线程休眠
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

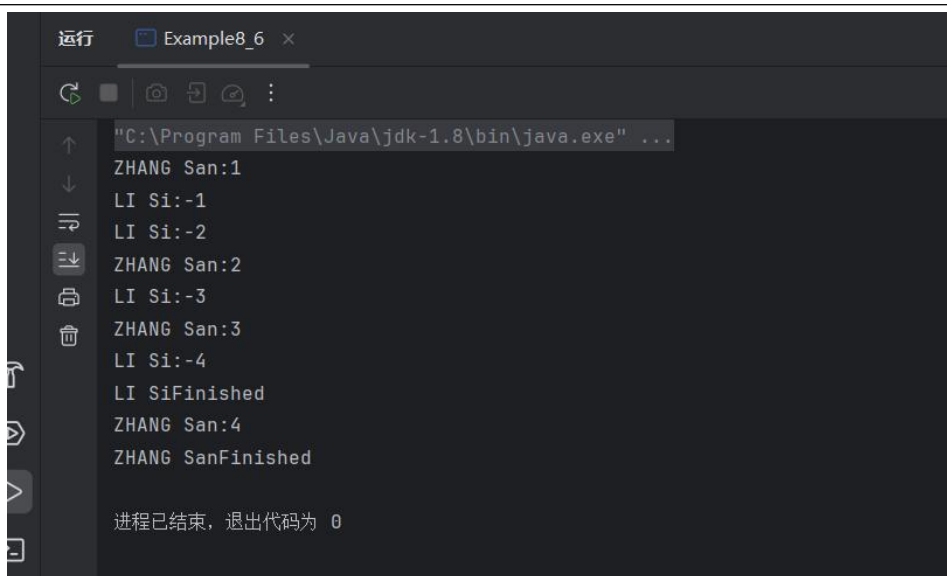
- 运行结果

```
运行 Example8_6 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
LI Si:-1
ZHANG San:1
LI Si:-2
ZHANG San:2
ZHANG San:3
LI Si:-3
ZHANG San:4
ZHANG SanFinished
LI Si:-4
LI SiFinished

进程已结束，退出代码为 0
```

```
运行 Example8_6 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
LI Si:-1
ZHANG San:1
LI Si:-2
ZHANG San:2
LI Si:-3
ZHANG San:3
LI Si:-4
LI SiFinished
ZHANG San:4
ZHANG SanFinished

进程已结束，退出代码为 0
|
```



```
运行 Example8_6 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
ZHANG San:1
LI Si:-1
LI Si:-2
ZHANG San:2
LI Si:-3
ZHANG San:3
LI Si:-4
LI SiFinished
ZHANG San:4
ZHANG SanFinished

进程已结束，退出代码为 0
```

• 完整代码:

```
1. package 实验3;
2.
3. class TaskAddSub implements Runnable {
4.     String s1, s2;
5.     TaskAddSub(String s1, String s2) {
6.         this.s1 = s1;
7.         this.s2 = s2;
8.     }
9.     public void run() {
10.        int i = 0;
11.        while (true) {
12.            if (Thread.currentThread().getName().equals(s1)) {
13.                i += 1; // i 值增加
14.                System.out.println(s1 + ":" + i);
15.                if (i >= 4) {
16.                    System.out.println(s1 + " Finished");
17.                    return;
18.                }
19.            } else if (Thread.currentThread().getName().equals(
20.                s2)) {
21.                i -= 1; // i 值减少
22.                System.out.println(s2 + ":" + i);
23.                if (i <= -4) {
24.                    System.out.println(s2 + " Finished");
25.                    return;
26.                }
27.            }
28.            try {
29.                Thread.sleep(800); // 线程休眠
```

```

29.         } catch (InterruptedException e) {
30.             e.printStackTrace();
31.         }
32.     }
33. }
34. }
35.
36. public class Example8_6 {
37.     public static void main(String args[]) {
38.         // 创建并启动加减任务线程
39.         String s1 = "ZHANG San";
40.         String s2 = "LI Si";
41.         TaskAddSub taskAddSub = new TaskAddSub(s1, s2);
42.         Thread zhang = new Thread(taskAddSub);
43.         Thread li = new Thread(taskAddSub);
44.         zhang.setName(s1);
45.         li.setName(s2);
46.         zhang.start();// 线程1 开始
47.         li.start(); // 线程2 开始
48.     }
49. }

```

总的说明：

Example8_1 - 简单的多线程

创建了两个线程对象，每个线程会输出自己的名称三次。程序执行如下：

类 WriteWordThread：

WriteWordThread 继承自 Thread 类，因此可以直接使用 Thread 的特性。

构造函数 WriteWordThread(String s)接受一个字符串并使用 setName(s)将其设置为线程名称。

run()方法是线程的执行体，线程启动后会执行该方法的代码。在 run()中，使用 for 循环输出当前线程的名字三次。

类 Example8_1：

在 main 方法中，创建了两个 WriteWordThread 对象 zhang 和 wang。

zhang.start()启动张线程，该线程会执行 run()中的内容。

for 循环使主线程输出三次 Main Thread。

然后调用 wang.start()启动王线程，两个线程可以并发执行。

说明：

由于多线程的运行顺序是非确定性的，因此每次运行可能得到不同的输出顺序。

Example8_2 - 加入线程延迟

在上一个代码的基础上增加了延迟，使线程输出的间隔不同。

类 WriteWordThread：

在构造函数中增加了一个 int n 参数，用于指定线程的休眠时间。

在 run()方法中，使用 Thread.sleep(n)让线程休眠 n 毫秒，从而在每次输出之间产生延迟。

catch 语句捕获 InterruptedException 异常，确保线程的稳定性。

类 Example8_2:

创建了两个 WriteWordThread 对象，分别设置不同的延迟时间（张线程延迟 200 毫秒，王线程延迟 100 毫秒）。

启动两个线程后，由于不同的延迟时间会导致每个线程的输出间隔不同，结果将产生更大的随机性。

说明:

线程的休眠时间不同，使输出顺序更加多样，运行结果不唯一。

Example8_3 - 竞争线程的递增计数器

此代码使用两个线程竞争一个递增计数器，同时主线程监控它们的状态，当其中任一线程的计数达到 8 时结束程序。

类 Left 和 Right:

Left 和 Right 类都继承自 Thread，并在 run() 方法中通过无限循环让各自的 n 变量递增。

每次循环中打印计数器的值并附加 "Left" 或 "Right" 标记，以区分是哪一个线程输出。

sleep((int) (Math.random() * 100)) 随机休眠一段时间，模拟多线程的不同运行速度。

类 Example8_3:

创建 Left 和 Right 线程并启动。

主线程通过一个 while 循环不断检查 left.n 和 right.n 的值。

如果任一计数器达到或超过 8，主线程打印计数器的值并使用 System.exit(0) 终止程序。

说明:

每次运行结果会有所不同，因为每个线程的休眠时间随机分配，因此各线程的输出顺序和计数进展不一致。

Example8_4 - 银行任务线程

该代码模拟了一个简单的银行系统，两个线程分别表示会计（treasurer zhang）和出纳（cashier cheng），同时对账户余额进行操作，直到满足终止条件。

类 TaskBank:

实现了 Runnable 接口，并在构造函数中接受两个字符串参数，表示两个线程的名称。setMoney 方法设置账户的初始金额。

run() 方法中使用一个 while 循环模拟账户余额的操作:

每次循环中，账户余额减少 10。

根据当前线程的名称输出不同的信息，并设置不同的终止条件:

如果是 treasurer zhang 线程，当余额小于等于 100 时终止。

如果是 cashier cheng 线程，当余额小于等于 60 时终止。

Thread.sleep(800) 让线程每次休眠 800 毫秒，减少运行速度。

类 Example8_4:

创建并设置初始金额，生成两个线程。

启动线程，模拟银行任务。

说明:

每次运行中，账户余额的减少速度可能有所不同，结果输出的顺序和终止条件的满足时间可能会有变化。

Example8_6 - 加减任务线程

该代码模拟了两个线程竞争同一变量 i，分别执行加和减操作，并设定不同的终止条

件。

类 TaskAddSub:

构造函数接受两个字符串参数 String s1 和 String s2, 表示两个线程的名称。

run()方法中包含一个 while 循环, 判断当前线程的名称并执行相应操作:

如果当前线程名称为 s1 (“ZHANG San”), 则对变量 i 执行加 1 操作, 并在 i 达到 4 时终止。

如果名称为 s2 (“LI Si”), 则对变量 i 执行减 1 操作, 并在 i 达到 -4 时终止。

Thread.sleep(800)使线程休眠 800 毫秒以减缓操作速度。

类 Example8_6:

创建一个 TaskAddSub 对象, 传递线程名称。

启动两个线程并分别设置其名称。

ZHANG San 线程和 LI Si 线程竞争 i 变量的值, 并分别按不同方向修改, 直到满足各自的终止条件。

说明:

每次运行结果可能不同, 因为两个线程交替对 i 进行加减操作, 结果受线程的执行顺序影响。

(2.2). 运行以下三个程序 (每个程序运行 5 次), 并对输出结果给出分析。在报告中附上程序截图和简要的文字说明 (包括对结果的分析)。(10 分)

程序 1:

• 程序截图

```
1 package 实验3;
2 class PrintChar implements Runnable { 2个用法
3     private char charToPrint; 2个用法
4     private int times; 2个用法
5     public PrintChar(char charToPrint, int times) { 2个用法
6         this.charToPrint = charToPrint;
7         this.times = times;
8     }
9     public void run() {
10         for (int i = 0; i < times; i++) {
11             System.out.print(charToPrint);
12         }
13     }
14 }
15 class PrintNum implements Runnable { 1个用法
16     private int lastnum; 2个用法
17     public PrintNum(int lastnum) { 1个用法
18         this.lastnum = lastnum;
19     }
20     public void run() {
21         for (int i = 0; i < lastnum; i++) {
22             System.out.print(" "+i);
23         }
24     }
25 }
```

• 运行结果

[illegible]

• 文字说明

线程的并发性:

每个线程独立运行，其任务执行顺序由 JVM 和操作系统的线程调度器决定。

因此，字符 'a'、字符 'b' 和数字 0-99 的输出顺序是随机的，不同的运行可能产生不同的混合结果。

输出混乱的原因:

线程之间没有同步机制，打印操作是并发执行的。当多个线程同时尝试打印时，控制台的输出流被多个线程争用，因此会出现交错的打印结果。

线程安全性:

本程序没有共享资源（如变量或数据结构），每个线程的任务独立完成，因此无需担心线程安全问题。

性能及公平性:

由于任务量较小（打印 100 次字符或数字），线程切换的开销不明显。

线程调度由 JVM 和操作系统控制，可能会导致某些线程被优先执行，但长期来看

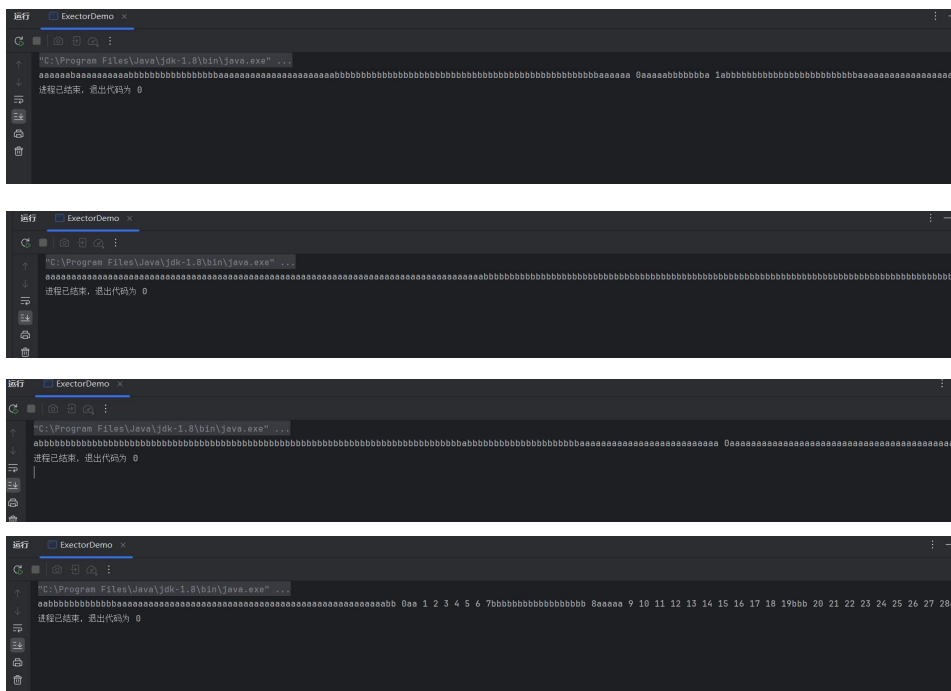
线程的执行是公平的。

程序 2:

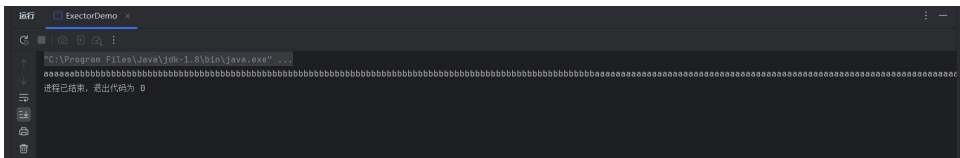
- 程序截图

```
1 package 实验3;
2 import java.util.concurrent.*;
3 class PrintChar implements Runnable { 2个用法
4     private char charToPrint; 2个用法
5     private int times; 2个用法
6     public PrintChar(char charToPrint, int times) { 2个用法
7         this.charToPrint = charToPrint;
8         this.times = times;
9     }
10    @Override
11    public void run() {
12        for (int i = 0; i < times; i++) {
13            System.out.print(charToPrint);
14        }
15    }
16 }
17 class PrintNum implements Runnable { 1个用法
18     private int lastnum; 2个用法
19     public PrintNum(int lastnum) { 1个用法
20         this.lastnum = lastnum;
21     }
22    public void run() {
23        for (int i = 0; i < lastnum; i++) {
24            System.out.print(" "+i);
25        }
26    }
27 }
28 public class ExecutorDemo {
29     public static void main(String[] args) {
30         ExecutorService executor = Executors.newFixedThreadPool( nThreads: 3);
31         executor.execute(new PrintChar( charToPrint: 'a', times: 100));
32         executor.execute(new PrintChar( charToPrint: 'b', times: 100));
33         executor.execute(new PrintNum( lastnum: 100));
34         executor.shutdown();
35     }
36 }
```

- 运行结果



The first screenshot shows the program starting and printing a long string of 'a's followed by 'b's. The second screenshot shows the program completing its execution and printing the exit code 0. The third screenshot shows the program starting and printing a long string of 'a's followed by 'b's. The fourth screenshot shows the program completing its execution and printing the exit code 0.



- 文字说明

这段代码的功能与前一个例子类似，也是通过多线程打印字符 'a'、字符 'b' 和数字 0 到 99。主要区别在于这次使用了 Java 的线程池 来管理线程的创建与执行，而不是直接创建线程对象。这种方式更加适合实际应用场景，尤其是需要执行大量并发任务时。

线程池机制

线程池：

使用了 `ExecutorService` 创建了一个固定大小为 3 的线程池 (`Executors.newFixedThreadPool(3)`)。

线程池会复用线程资源，避免了频繁创建和销毁线程带来的性能开销。

任务提交：

调用 `executor.execute()` 将 `Runnable` 任务提交到线程池。线程池会从任务队列中取出任务并交由可用的线程执行。

程序中三个任务分别打印 'a'、'b' 和数字 0-99。

线程池关闭：

调用了 `executor.shutdown()`，表示不再接受新任务，线程池会在所有已提交任务完成后关闭。

程序 3:

- 程序截图

```
1 package 实验3;
2 import java.util.*;
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executors;
5 public class AccountWithoutSync{
6     private static Account account=new Account(); 2个用法
7     public static void main(String[] args){
8         ExecutorService executor = Executors.newCachedThreadPool();
9
10        for(int i=0;i<100;i++){
11            executor.execute(new AddAPennyTask());
12        }
13        executor.shutdown();
14        while (!executor.isTerminated()){ }
15        System.out.println("What is balance?" + account.getBalance());
16    }
17    private static class AddAPennyTask implements Runnable{ 1个用法
18
19        public void run() {
20            account.deposit(1);
21        }
22    }
23    private static class Account { 2个用法
24        private int balance = 0; 3个用法
25        public int getBalance() { 1个用法
26            return balance;
27        }
28    }
29 }
```

```

28  public void deposit(int amount) { 1 个用法
    int newBalance = balance + amount;
30  try{
31      Thread.sleep( millis: 5);
32  }catch(InterruptedException ex){}
33  balance = newBalance;
34  }
35  }
36  }

```

- 运行结果

```

运行 AccountWithoutSync x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
what is balance?1
进程已结束，退出代码为 0

```

```

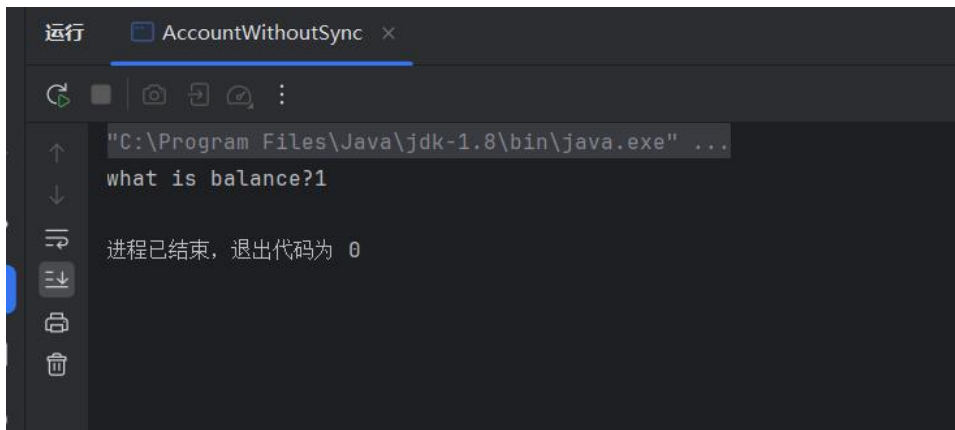
运行 AccountWithoutSync x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
what is balance?1
进程已结束，退出代码为 0

```

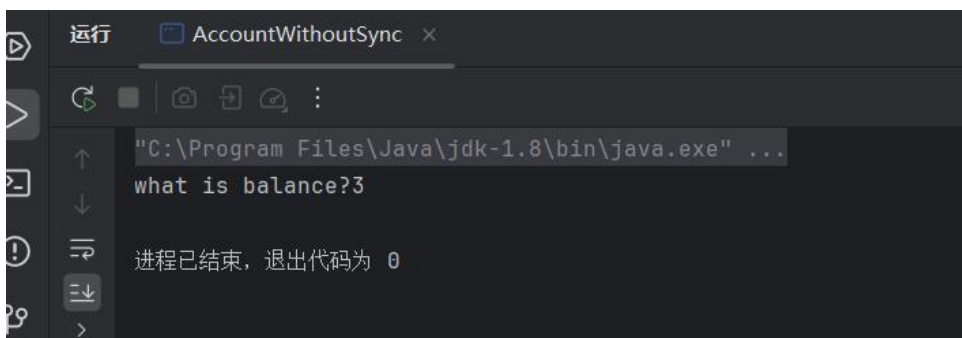
```

运行 AccountWithoutSync x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
what is balance?1
进程已结束，退出代码为 0

```



```
运行 AccountWithoutSync x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
what is balance?1
进程已结束, 退出代码为 0
```



```
运行 AccountWithoutSync x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
what is balance?3
进程已结束, 退出代码为 0
```

- 文字说明

程序结构

Account 类:

维护账户的余额 balance。

提供了 getBalance 方法获取余额。

提供了 deposit 方法向账户存钱，模拟一个非线程安全的存款操作：

存款时，先读取余额 balance，然后计算新的余额并赋值。

中间使用 Thread.sleep(5) 模拟处理延迟。

AddAPennyTask 类:

实现了 Runnable 接口。

任务的功能是调用账户的 deposit 方法存入 1 单位金额。

主方法:

使用线程池 (ExecutorService) 创建了 100 个线程。

每个线程执行 AddAPennyTask 任务，即向账户存入 1 单位金额。

在所有线程完成任务后，输出账户的最终余额。

结果分析

由于 deposit 方法不是线程安全的，程序运行可能导致最终余额小于 100。这是因为多个线程可能同时读取并修改余额，造成以下问题：

竞争条件：

多个线程可能同时读取 balance 的值。

在某些线程更新 balance 前，其他线程已经计算了新的余额，这些操作会互相覆盖。

线程切换：

Thread.sleep(5) 增大了线程切换的机会，使多个线程更可能并发执行。

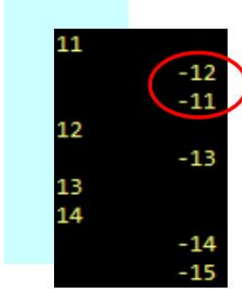
一个线程尚未完成更新操作时，另一个线程可能已经修改了余额，导致错误的结果。

因此，得到的结果为 1（有时候为 3 或者 4，基本都是 1），（比 100 小）这是由于多线程竞争问题。

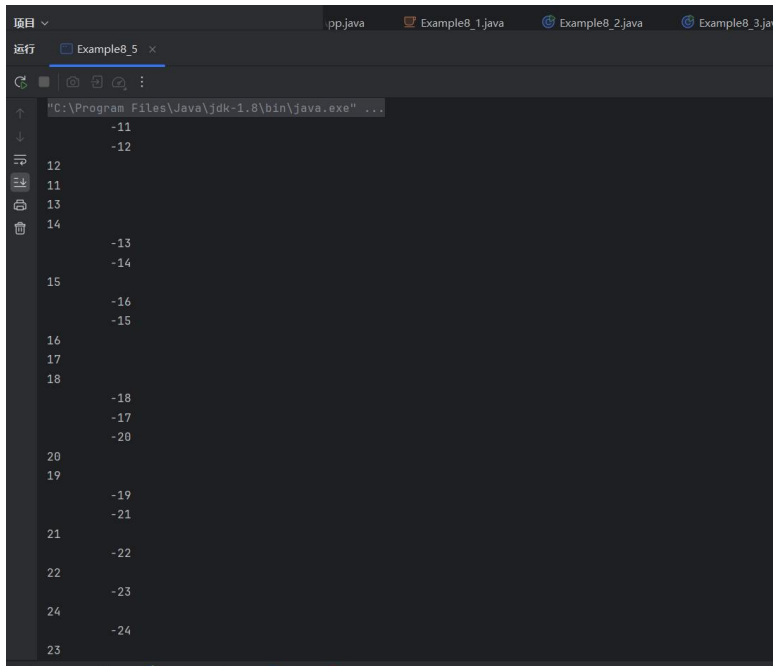
(2.3) 第 8 章讲义（JavaPD-Ch08）中的第 5 个应用程序（Example8_5）存在线程间不同步的问题，请修改该程序，以解决不同步的问题。在报告中附上程序截图、运行结果截图和详细的文字说明（包括设计的思路和合理性分析）。（10 分）

- 初始的程序中，存在线程间不同步的问题。

现在运行代码，可以看到他的输出存在一些问题：



```
11
12
13
14
-12
-11
-13
-14
-15
```



```
Example8_5.java
11
12
13
14
15
16
17
18
19
20
21
22
23
24
-11
-12
-13
-14
-15
-16
-17
-18
-19
-20
-21
-22
-23
-24
```

```
项目
运行 Example8_5 x
"C:\Program Files\Java\jdk-1.
-11
-12
12
11
13
14
-13
-14
15
-16
-15
16
17
18
-18
-17
-20
20
19
-19
-21
21
-22
22
-23
24
-24
23
```

针对此，进行改进：

- 程序截图：

```
java Example8_1.java Example8_2.java Example8_3.java Example8_4.java Example8_6.java Example8_5.java x
package 实验3;

class Task implements Runnable { 4个用法
    private int number = 0; 5个用法

    public void setNumber(int n) { 2个用法
        number = n;
    }

    public void run() {
        while (true) {
            synchronized (this) { // 使用同步代码块，确保对number变量的操作是线程安全的
                if (Thread.currentThread().getName().equals("add")) {
                    number++;
                    System.out.printf("%d\n", number);
                }
                if (Thread.currentThread().getName().equals("sub")) {
                    number--;
                    System.out.printf("%12d\n", number);
                }
            }
            try {
                Thread.sleep(1000); // 休眠1000毫秒，模拟操作间的时间间隔
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Example8_5 {
```

- 完整代码：

1. package 实验3;
- 2.

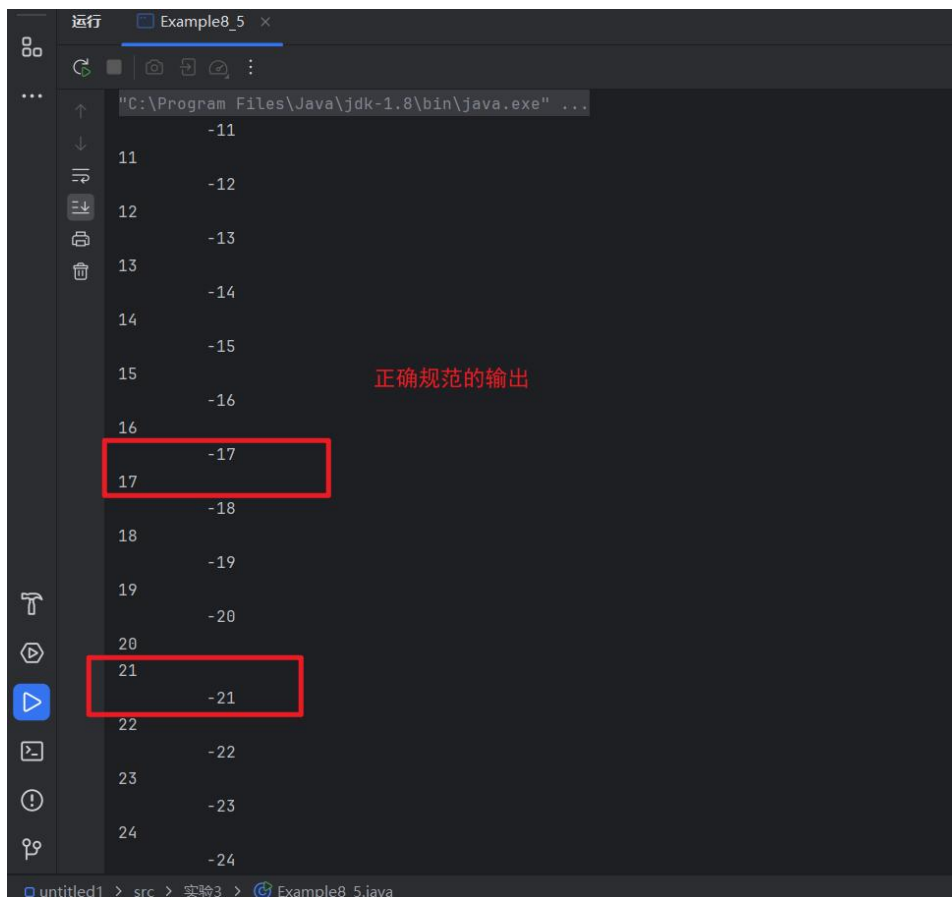

```
3.  class Task implements Runnable {
4.      private int number = 0;
5.
6.      public void setNumber(int n) {
7.          number = n;
8.      }
9.
10.     public void run() {
11.         while (true) {
12.             synchronized (this) { // 使用同步代码块, 确保对 number
                变量的操作是线程安全的
13.                 if (Thread.currentThread().getName().equals("ad
                    d")) {
14.                     number++;
15.                     System.out.printf("%d\n", number);
16.                 }
17.                 if (Thread.currentThread().getName().equals("su
                    b")) {
18.                     number--;
19.                     System.out.printf("%12d\n", number);
20.                 }
21.             }
22.             try {
23.                 Thread.sleep(1000); // 休眠 1000 毫秒, 模拟操作间的
                时间间隔
24.             } catch (InterruptedException e) {
25.                 e.printStackTrace();
26.             }
27.         }
28.     }
29. }
30.
31. public class Example8_5 {
32.     public static void main(String args[]) {
33.         Task taskAdd = new Task();
34.         taskAdd.setNumber(10);
35.         Task taskSub = new Task();
36.         taskSub.setNumber(-10);
37.
38.         // 创建四个线程
39.         Thread threadA, threadB, threadC, threadD;
40.         threadA = new Thread(taskAdd);
41.         threadB = new Thread(taskAdd);
42.         threadA.setName("add");
```

```

43.         threadB.setName("add");
44.
45.         threadC = new Thread(taskSub);
46.         threadD = new Thread(taskSub);
47.         threadC.setName("sub");
48.         threadD.setName("sub");
49.
50.         // 启动线程
51.         threadA.start();
52.         threadB.start();
53.         threadC.start();
54.         threadD.start();
55.     }
56. }

```

- 运行结果截图（改善后）



```

运行 Example8_5 x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
-11
11 -12
12 -13
13 -14
14 -15
15 -16
16 -17
17 -18
18 -19
19 -20
20 -21
21 -22
22 -23
23 -24
24

```

正确规范的输出

- 改善思路

（问题所在）在 Example8_5 中，由于多个线程同时访问和修改共享变量 `number`，产生了线程间不同步的问题（线程安全问题），因为线程可能会在操作 `number` 变量时发生竞态条件，即多个线程可能会同时读取或修改变量，导致数据不一致的情况。

为了解决这个问题，可以采用同步机制，确保每次只有一个线程能够访问 `number` 变量。Java 提供了同步代码块和同步方法来实现这种互斥访问。

修改方案

在 run 方法中，使用 synchronized 关键字对 number 变量的增减操作进行同步，以保证线程安全。

设计思路和合理性分析

设计思路：通过将 number 的增减操作放在 synchronized 块中，我们确保每次只有一个线程可以执行这些操作。Java 中的 synchronized 关键字可以确保同一时刻只有一个线程进入同步代码块，这样可以防止线程并发修改共享变量导致的数据不一致问题。

合理性分析：

互斥访问：synchronized 块保证了对共享资源 number 的互斥访问，避免了竞态条件。

线程安全性：由于 number++ 和 number-- 操作是原子性的，所有对 number 的操作都是线程安全的，避免了多线程环境下的错误。

性能考虑：虽然 synchronized 会导致一定的性能开销，但在此场景中，由于操作简单，且线程数较少，因此开销可忽略不计。

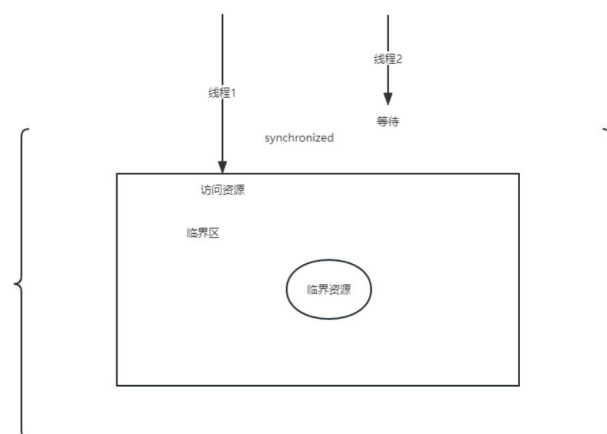
• 针对这个 synchronized 的用法，是在博客看到的：[【多线程与高并发】- synchronized 锁的认知 synchronized 能否锁住多台-CSDN 博客](#)

3.1、修饰一个代码块

1) 一个线程访问一个对象中的 synchronized(this) 同步代码块时，其他试图访问该对象的线程将被阻塞。

```
1 class SyncThread implements Runnable {
2     private static int count;
3
4     public SyncThread() {
5         count = 0;
6     }
7
8     public void run() {
9         synchronized(this) {
10             for (int i = 0; i < 5; i++) {
11                 try {
12                     System.out.println(Thread.currentThread().getName() + ":" + (count++));
13                     Thread.sleep(100);
14                 } catch (InterruptedException e) {
15                     e.printStackTrace();
16                 }
17             }
18         }
19     }
20
21     public int getCount() {
22         return count;
23     }
24 }
```

使得线程有序进行

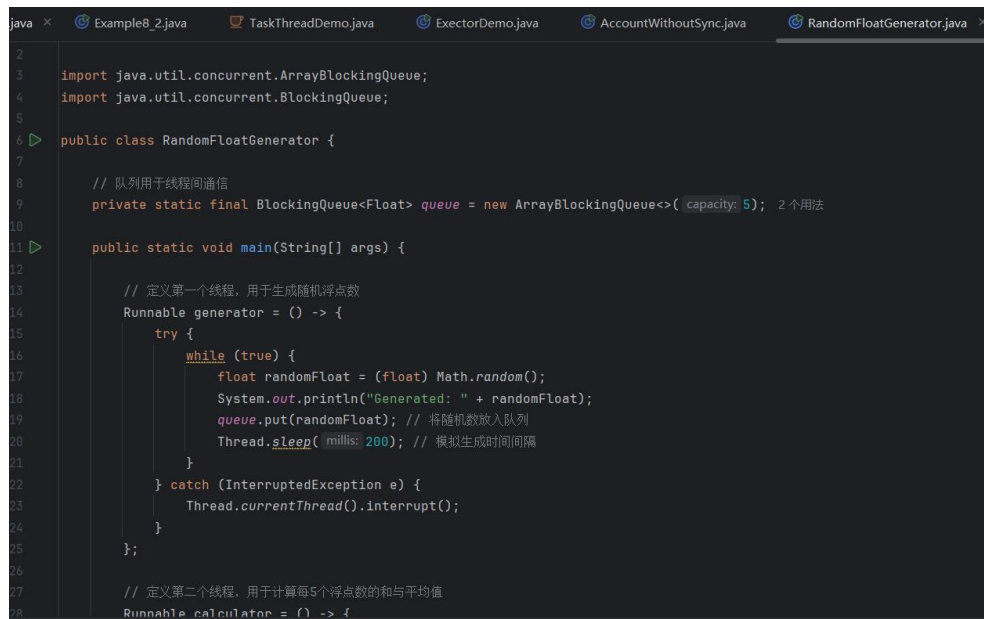


图：synchronized 锁的示意图

Part 3 (30 分)

(3.1). 编写 Java 应用程序实现如下功能：第一个线程不停地随机生成[0,1)之间的浮点数（float）并输出到屏幕，第二个线程将第一个线程输出的第 1-5 个浮点数的和与平均值输出到屏幕（紧跟在第一个线程输出的第 5 个浮点数之后）、将第一个线程输出的第 6-10 个[0,1)之间的浮点数的和与平均值输出到屏幕（紧跟在第一个线程输出的第 10 个浮点数之后）...。要求线程间实现通信。要求采用实现 Runnable 接口和 Thread 类的方式创建线程，而不是通过 Thread 类的子类的方式。在报告中附上程序截图、运行结果截图和详细的文字说明（包括设计的思路 and 合理性分析）。（10 分）

• 程序截图



```
1  java × Example8_2.java TaskThreadDemo.java ExecutorDemo.java AccountWithoutSync.java RandomFloatGenerator.java ×
2
3  import java.util.concurrent.ArrayBlockingQueue;
4  import java.util.concurrent.BlockingQueue;
5
6  public class RandomFloatGenerator {
7
8      // 队列用于线程间通信
9      private static final BlockingQueue<Float> queue = new ArrayBlockingQueue<>(capacity: 5); // 2个用法
10
11  public static void main(String[] args) {
12
13      // 定义第一个线程，用于生成随机浮点数
14      Runnable generator = () -> {
15          try {
16              while (true) {
17                  float randomFloat = (float) Math.random();
18                  System.out.println("Generated: " + randomFloat);
19                  queue.put(randomFloat); // 将随机数放入队列
20                  Thread.sleep(200); // 模拟生成时间间隔
21              }
22          } catch (InterruptedException e) {
23              Thread.currentThread().interrupt();
24          }
25      };
26
27      // 定义第二个线程，用于计算每5个浮点数的和与平均值
28      Runnable calculator = () -> {
```

• 运行结果

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
```

```
Generated: 0.8164453
Generated: 0.5136489
Generated: 0.7111905
Generated: 0.4722319
Generated: 0.39043674
Sum of 5 numbers: 2.90395, Average: 0.58079
Generated: 0.8999206
Generated: 0.13870494
Generated: 0.9906658
Generated: 0.1538251
Generated: 0.58861625
Sum of 5 numbers: 2.77173, Average: 0.55435
Generated: 0.9589518
Generated: 0.528258
Generated: 0.30266607
Generated: 0.31876552
Generated: 0.7251358
Sum of 5 numbers: 2.83378, Average: 0.56676
Generated: 0.9656103
Generated: 0.76109314
Generated: 0.5063842
Generated: 0.8674605
Generated: 0.82732886
Sum of 5 numbers: 3.92788, Average: 0.78558
```

• 完整代码

```
1. package 实验3;
2. import java.util.concurrent.ArrayBlockingQueue;
3. import java.util.concurrent.BlockingQueue;
4.
5. public class RandomFloatGenerator {
6.
7.     // 队列用于线程间通信
8.     private static final BlockingQueue<Float> queue = new Array
        BlockingQueue<>(5);
9.
10.    public static void main(String[] args) {
11.
12.        // 定义第一个线程，用于生成随机浮点数
13.        Runnable generator = () -> {
14.            try {
15.                while (true) {
16.                    float randomFloat = (float) Math.random();
17.                    System.out.println("Generated: " + randomFl
                        oat);
18.                    queue.put(randomFloat); // 将随机数放入队列
19.                    Thread.sleep(200); // 模拟生成时间间隔
```

```

20.         }
21.     } catch (InterruptedException e) {
22.         Thread.currentThread().interrupt();
23.     }
24. };
25.
26.     // 定义第二个线程，用于计算每5个浮点数的和与平均值
27.     Runnable calculator = () -> {
28.         try {
29.             int count = 0;
30.             float sum = 0;
31.
32.             while (true) {
33.                 float num = queue.take(); // 从队列中取出一个随机数
34.                 sum += num;
35.                 count++;
36.
37.                 if (count == 5) {
38.                     // 输出前5个数的和与平均值
39.                     float average = sum / 5;
40.                     System.out.printf("Sum of 5 numbers: %.5f, Average: %.5f\n", sum, average);
41.                     // 重置计数器和总和
42.                     count = 0;
43.                     sum = 0;
44.                 }
45.             }
46.         } catch (InterruptedException e) {
47.             Thread.currentThread().interrupt();
48.         }
49.     };
50.
51.     // 启动两个线程
52.     new Thread(generator).start();
53.     new Thread(calculator).start();
54. }
55. }
56.

```

• 文字说明

程序说明

线程间通信：使用 `BlockingQueue`（具体为 `ArrayBlockingQueue`）实现两个线程的通信，第一个线程将生成的浮点数放入队列中，第二个线程从队列中获取浮点数。这样实现了线程之间的同步，因为 `take()` 方法会阻塞，直到有元素可用。

生成和计算：第一个线程每隔 200 毫秒生成一个 $[0, 1)$ 的随机浮点数并输出到屏幕。第二个线程每读取 5 个数后，计算这 5 个数的和与平均值，并立即输出到屏幕。

循环处理：第二个线程持续进行读取并在满足条件时输出计算结果。通过重置计数器和总和，程序能够不断地处理后续生成的数据。

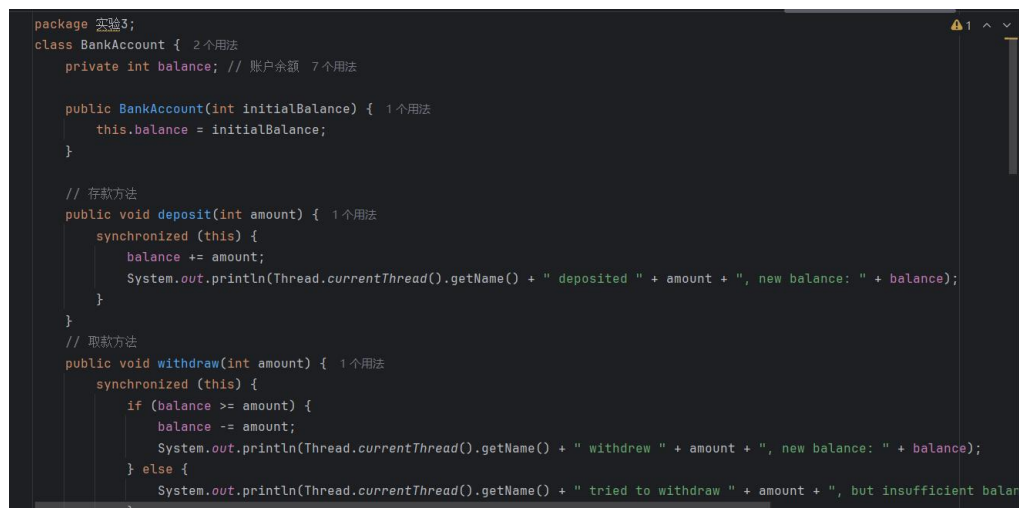
合理性分析

线程安全：BlockingQueue 保证了线程安全的操作，因此在多线程环境下无需担心资源竞争。

同步控制：使用阻塞队列控制线程的同步，不会出现丢失数据或多余等待的问题。

(3.2). 编写 Java 应用程序实现如下功能：创建工作线程，模拟银行现金账户取款和存款操作。多个线程同时执行取款和存款操作时，如果不使用同步处理，会造成账户余额混乱，要求使用 synchronized 关键字同步代码块，以保证多个线程同时执行取款和存款操作时，银行现金账户取款和存款的有效和一致。要求采用实现 Runnable 接口和 Thread 类的构造方法的方式创建线程，而不是通过 Thread 类的子类的方式。在报告中附上程序截图（假设银行存款有 100 元，有 3 个取款线程和 2 个存款线程，每次取款和存款均为 10 元）、运行结果截图（显示每次存取款操作后的余额等信息，以说明线程间同步正确）和详细的文字说明。（10 分）

• 程序截图



```
package 实验3;
class BankAccount { 2 个用法
    private int balance; // 账户余额 7 个用法

    public BankAccount(int initialBalance) { 1 个用法
        this.balance = initialBalance;
    }

    // 存款方法
    public void deposit(int amount) { 1 个用法
        synchronized (this) {
            balance += amount;
            System.out.println(Thread.currentThread().getName() + " deposited " + amount + ", new balance: " + balance);
        }
    }

    // 取款方法
    public void withdraw(int amount) { 1 个用法
        synchronized (this) {
            if (balance >= amount) {
                balance -= amount;
                System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ", new balance: " + balance);
            } else {
                System.out.println(Thread.currentThread().getName() + " tried to withdraw " + amount + ", but insufficient balar
            }
        }
    }
}
```

• 运行截图

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Withdraw-1 withdrew 10, new balance: 90
Deposit-1 deposited 10, new balance: 100
Withdraw-3 withdrew 10, new balance: 90
Withdraw-2 withdrew 10, new balance: 80
Deposit-2 deposited 10, new balance: 90
Withdraw-1 withdrew 10, new balance: 80
Withdraw-3 withdrew 10, new balance: 70
Deposit-2 deposited 10, new balance: 80
Withdraw-2 withdrew 10, new balance: 70
Deposit-1 deposited 10, new balance: 80
Withdraw-2 withdrew 10, new balance: 70
Deposit-1 deposited 10, new balance: 80
Deposit-2 deposited 10, new balance: 90
Withdraw-3 withdrew 10, new balance: 80
Withdraw-1 withdrew 10, new balance: 70
Withdraw-3 withdrew 10, new balance: 60
Withdraw-1 withdrew 10, new balance: 50
Deposit-1 deposited 10, new balance: 60
Deposit-2 deposited 10, new balance: 70
Withdraw-2 withdrew 10, new balance: 60
Withdraw-3 withdrew 10, new balance: 50
Withdraw-2 withdrew 10, new balance: 40
Withdraw-1 withdrew 10, new balance: 30
Deposit-2 deposited 10, new balance: 40
Deposit-1 deposited 10, new balance: 50

进程已结束，退出代码为 0
```

• 完整代码

```
1. package 实验3;
2. class BankAccount {
3.     private int balance; // 账户余额
4.
5.     public BankAccount(int initialBalance) {
6.         this.balance = initialBalance;
7.     }
8.
9.     // 存款方法
10.    public void deposit(int amount) {
11.        synchronized (this) {
12.            balance += amount;
13.            System.out.println(Thread.currentThread().getName()
14.                + " deposited " + amount + ", new balance: " + balance);
15.        }
16.    }
17.    // 取款方法
18.    public void withdraw(int amount) {
```



```
18.         synchronized (this) {
19.             if (balance >= amount) {
20.                 balance -= amount;
21.                 System.out.println(Thread.currentThread().getNa
me() + " withdrew " + amount + ", new balance: " + balance);
22.             } else {
23.                 System.out.println(Thread.currentThread().getNa
me() + " tried to withdraw " + amount + ", but insufficient bala
nce. Current balance: " + balance);
24.             }
25.         }
26.     }
27. }
28.
29. public class BankSimulation {
30.     public static void main(String[] args) {
31.         BankAccount account = new BankAccount(100); // 初始账户
余额为 100
32.
33.         // 定义取款线程
34.         Runnable withdrawTask = () -> {
35.             for (int i = 0; i < 5; i++) { // 每个线程尝试取
款 5 次
36.                 account.withdraw(10);
37.                 try {
38.                     Thread.sleep(100); // 模拟处理时间
39.                 } catch (InterruptedException e) {
40.                     Thread.currentThread().interrupt();
41.                 }
42.             }
43.         };
44.
45.         // 定义存款线程
46.         Runnable depositTask = () -> {
47.             for (int i = 0; i < 5; i++) { // 每个线程尝试存
款 5 次
48.                 account.deposit(10);
49.                 try {
50.                     Thread.sleep(100); // 模拟处理时间
51.                 } catch (InterruptedException e) {
52.                     Thread.currentThread().interrupt();
53.                 }
54.             }
55.         };
```

```

56.
57.         // 创建 3 个取款线程
58.         Thread withdrawThread1 = new Thread(withdrawTask, "With
draw-1");
59.         Thread withdrawThread2 = new Thread(withdrawTask, "With
draw-2");
60.         Thread withdrawThread3 = new Thread(withdrawTask, "With
draw-3");
61.
62.         // 创建 2 个存款线程
63.         Thread depositThread1 = new Thread(depositTask, "Deposi
t-1");
64.         Thread depositThread2 = new Thread(depositTask, "Deposi
t-2");
65.
66.         // 启动所有线程
67.         withdrawThread1.start();
68.         withdrawThread2.start();
69.         withdrawThread3.start();
70.         depositThread1.start();
71.         depositThread2.start();
72.     }
73. }
74.

```

• 文字说明

程序说明

线程安全性：

使用 `synchronized` 同步代码块，确保存款和取款方法在同一时刻只能有一个线程操作余额，避免了竞争条件和数据不一致问题。

设计逻辑：

取款线程：3 个线程，每次取款 10 元，共尝试 5 次。

存款线程：2 个线程，每次存款 10 元，共尝试 5 次。

在存取款操作时，打印当前线程的操作和操作后的账户余额。

同步合理性：

将同步块限制在余额修改的核心逻辑内，减少锁的粒度，保证了性能和安全的平衡。

运行结果分析：

每次存取款操作后的余额显示一致性，即不会出现多个线程同时修改余额导致的混乱。

当余额不足时，系统会提示，并拒绝取款操作。

使用方法

编译代码：将代码保存为 `BankSimulation.java`，并通过命令 `javac BankSimulation.java` 进行编译。

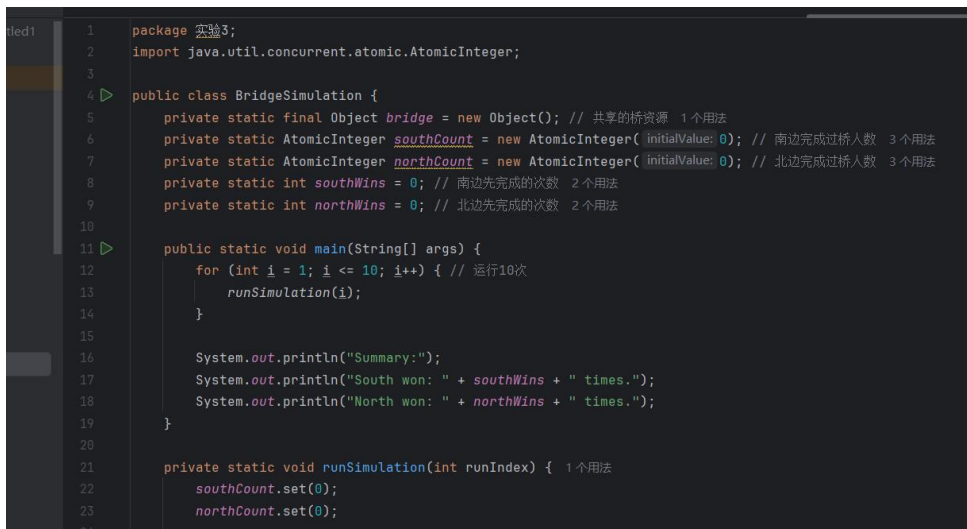
运行程序：使用命令 `java BankSimulation` 执行程序。

验证结果：观察输出，确认每次存取款后的余额计算正确，且不会出现线程竞争导

致的错误。

(3.3). 有一座南北向的桥，只能容纳一个人，桥的南边有 1000 个人（记为 S1,S2,...,S1000）和桥的北边有 1000 个人（记为 N1,N2,...,N1000），编写 Java 应用程序让这些到达对岸，每个人用一个线程表示，桥为共享资源，在过桥的过程中输出谁正在过桥（不同人之间用逗号隔开）。运行 10 次，分别统计南边的 1000 人和北边的 1000 人先全部到达对岸的次数（第 i 行输出格式为：第 i 次运行，南边/北边先完成过桥）。要求采用实现 Runnable 接口和 Thread 类的构造方法的方式创建线程，而不是通过 Thread 类的子类的方式。在报告中附上程序截图、运行结果截图和详细的文字说明（包括对结果的分析）。（10 分）

- 程序截图



```
1 package 实验3;
2 import java.util.concurrent.atomic.AtomicInteger;
3
4 public class BridgeSimulation {
5     private static final Object bridge = new Object(); // 共享的桥资源 1个用法
6     private static AtomicInteger southCount = new AtomicInteger(0); // 南边完成过桥人数 3个用法
7     private static AtomicInteger northCount = new AtomicInteger(0); // 北边完成过桥人数 3个用法
8     private static int southWins = 0; // 南边先完成的次数 2个用法
9     private static int northWins = 0; // 北边先完成的次数 2个用法
10
11     public static void main(String[] args) {
12         for (int i = 1; i <= 10; i++) { // 运行10次
13             runSimulation(i);
14         }
15
16         System.out.println("Summary:");
17         System.out.println("South won: " + southWins + " times.");
18         System.out.println("North won: " + northWins + " times.");
19     }
20
21     private static void runSimulation(int runIndex) { 1个用法
22         southCount.set(0);
23         northCount.set(0);
```

- 运行结果

```
运行 BridgeSimulation x
S872 is crossing the bridge.
S871 is crossing the bridge.
S870 is crossing the bridge.
S869 is crossing the bridge.
S868 is crossing the bridge.
S867 is crossing the bridge.
S865 is crossing the bridge.
S619 is crossing the bridge.
S863 is crossing the bridge.
S862 is crossing the bridge.
S615 is crossing the bridge.
S614 is crossing the bridge.
S859 is crossing the bridge.
S858 is crossing the bridge.
N171 is crossing the bridge.
S856 is crossing the bridge.
S855 is crossing the bridge.
S608 is crossing the bridge.
N160 is crossing the bridge.
S852 is crossing the bridge.
S851 is crossing the bridge.
N158 is crossing the bridge.
Run 10: South finished first.
Summary:
South won: 10 times.
North won: 0 times.
进程已结束, 退出代码为 0
```

• 完整代码

```
1. package 实验3;
2. import java.util.concurrent.atomic.AtomicInteger;
3.
4. public class BridgeSimulation {
5.     private static final Object bridge = new Object(); // 共享
    的桥资源
6.     private static AtomicInteger southCount = new AtomicInteger
    (0); // 南边完成过桥人数
7.     private static AtomicInteger northCount = new AtomicInteger
    (0); // 北边完成过桥人数
8.     private static int southWins = 0; // 南边先完成的次数
9.     private static int northWins = 0; // 北边先完成的次数
10.
11.     public static void main(String[] args) {
```

```
12.         for (int i = 1; i <= 10; i++) { // 运行10次
13.             runSimulation(i);
14.         }
15.
16.         System.out.println("Summary:");
17.         System.out.println("South won: " + southWins + " times.
18.             ");
19.         System.out.println("North won: " + northWins + " times.
20.             ");
21.     }
22.
23.     private static void runSimulation(int runIndex) {
24.         southCount.set(0);
25.         northCount.set(0);
26.
27.         Thread[] southThreads = new Thread[1000];
28.         Thread[] northThreads = new Thread[1000];
29.
30.         // 创建南边的人线程
31.         for (int i = 0; i < 1000; i++) {
32.             final int id = i + 1;
33.             southThreads[i] = new Thread(new CrossBridgeRunnabl
34.                 e("S" + id, southCount));
35.         }
36.
37.         // 创建北边的人线程
38.         for (int i = 0; i < 1000; i++) {
39.             final int id = i + 1;
40.             northThreads[i] = new Thread(new CrossBridgeRunnabl
41.                 e("N" + id, northCount));
42.         }
43.
44.         // 启动所有线程
45.         for (Thread t : southThreads) {
46.             t.start();
47.         }
48.         for (Thread t : northThreads) {
49.             t.start();
50.         }
51.
52.         // 等待所有线程完成
53.         try {
54.             for (Thread t : southThreads) {
55.                 t.join();
56.             }
57.             for (Thread t : northThreads) {
58.                 t.join();
59.             }
60.         } catch (InterruptedException e) {}
61.     }
62. }
```

```
52.         }
53.         for (Thread t : northThreads) {
54.             t.join();
55.         }
56.     } catch (InterruptedException e) {
57.         Thread.currentThread().interrupt();
58.     }
59.
60.     // 判断哪边先完成
61.     if (southCount.get() == 1000) {
62.         southWins++;
63.         System.out.println("Run " + runIndex + ": South finished first.");
64.     } else if (northCount.get() == 1000) {
65.         northWins++;
66.         System.out.println("Run " + runIndex + ": North finished first.");
67.     }
68. }
69.
70. // 实现 Runnable 接口的类，用于表示过桥的人
71. private static class CrossBridgeRunnable implements Runnable {
72.     private String person;
73.     private AtomicInteger counter;
74.
75.     public CrossBridgeRunnable(String person, AtomicInteger counter) {
76.         this.person = person;
77.         this.counter = counter;
78.     }
79.
80.     @Override
81.     public void run() {
82.         synchronized (bridge) { // 同步代码块，确保同一时刻只有一个人能过桥
83.             System.out.println(person + " is crossing the bridge.");
84.             try {
85.                 Thread.sleep(1); // 模拟过桥时间
86.             } catch (InterruptedException e) {
87.                 Thread.currentThread().interrupt();
88.             }
89.             counter.incrementAndGet(); // 统计已完成过桥的人
```

数

90. }

91. }

92. }

93. }

• 文字说明

程序说明

核心逻辑：使用 `synchronized` 保证桥的线程安全，确保同一时刻只有一个人能过桥。每个人通过一个线程表示，南北两边各有 1000 个线程。使用 `AtomicInteger` 统计已过桥的人数。

同步控制：通过 `synchronized (bridge)` 锁定桥，确保线程安全。通过 `Thread.sleep(1)` 模拟过桥时间，增加竞争性。

运行与结果：程序运行 10 次，每次判断南边或北边哪一方先完成过桥，并统计南北两边的胜利次数。

合理性分析

线程安全：使用 `synchronized` 和 `AtomicInteger` 保证线程安全，避免竞态条件。

公平性：线程调度由 JVM 控制，程序运行结果不确定，体现了竞争性和公平性。

统计可靠性：使用 `AtomicInteger` 保证计数准确，确保最终结果的可靠性。

结果分析

竞争性：每次运行的结果可能不同，取决于线程调度和竞争情况。

统计：通过 10 次运行统计南北两边谁先完成过桥，最终输出南北两边各自先完成的次数。

+++++

其他（例如感想、建议等等）。

在完成本次实验后，我深刻感受到了 Java 集合类以及线程同步技术的强大和实用。通过学习和实践，我不仅掌握了常用的集合类，例如 ArrayList、LinkedList、HashMap 等，还学会了如何有效地查阅 Java 文档，以便在程序设计中选择合适的类和接口。

实验过程中，我体会到了集合类在数据处理中的高效性和灵活性，它们为 Java 程序提供了丰富的数据结构，使得数据操作更加简便。同时，通过线程同步技术的学习，我了解了如何在多线程环境下保证数据的一致性和安全性，这对于编写高效、稳定的并发程序至关重要。

这次实验让我认识到，熟练掌握 Java 提供的常用类和线程同步技术，对于提升编程能力、解决复杂问题具有重要意义。在未来的学习和实践中，我将继续深化对这些技术的理解和应用，以提升自己的编程水平。

指导教师批阅意见:

成绩评定:

指导教师签字:

2024 年 月 日

备注:

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。