实验二、多线程矩阵相乘

实验环境：

Linux 平台或 Wnidows 平台

实验时间：

2 小时

实验目的：

掌握多线程编程技术，理解多线程的优点和缺点。

实验目标：

编制一个程序，采用单线程、4 线程和 16 线程来计算两个矩阵 A 和 B 的乘积。其中，矩阵大小分别为 16 * 16,128 * 128，512*512。

实验步骤：

1、开发一个单线程的程序，分别计算 16 * 16,128 * 128，512 * 512矩阵的乘积，并记录矩阵相乘花费的时间；

2、开发一个 4线程的程序，分别计算 16 * 16,128 * 128，512*512矩阵的乘积，并记录矩阵相乘花费的时间；

3、开发一个 16 线程的程序，分别计算 16 * 16,128 * 128，512*512 矩阵的乘积，并记录矩阵相乘花费的时间。

代码实现

1. FileToMatrices.java 读取文件生成矩阵

```
1    package com.os.zlj;
2
3    import java.io.*;
4    import java.util.ArrayList;
5    import java.util.Arrays;
```

```java

 public class FileToMatrices {
     private float[][] A, B;

     public FileToMatrices(String fileA, String fileB) {
         // 读取文件
         A = toMatrices(fileA);
         B = toMatrices(fileB);
     }

     public FileToMatrices() {
         // 默认构造测试数据
         this("./src/resources/M64A.txt", "./src/resources/M64B.txt");
     }

     public static void main(String[] args) {
         File file = new File(".");
         System.out.println(file.getAbsolutePath());
         FileToMatrices fileToMatrices = new FileToMatrices();
         System.out.println(fileToMatrices);
     }

     public float[][] getA() {
         return A;
     }

     public void setA(float[][] a) {
         A = a;
     }

     public float[][] getB() {
         return B;
     }

     public void setB(float[][] b) {
         B = b;
     }


     private float[][] toMatrices(String fileName) {
         float[][] array = new float[0][];
         // 读取文件
         try (
                 Reader reader = new FileReader(fileName);
                 BufferedReader bufferedReader = new
BufferedReader(reader)
         ) {
             ArrayList<String> arrayList = new ArrayList<>();
             String line;
             while ((line = bufferedReader.readLine()) != null) {
                 arrayList.add(line);
             }

             // 创建数组
             int high = arrayList.size();
             int width = arrayList.get(0).split(" ").length;
             array = new float[high][width];

```

```
                String[] splitLine;
                for (int i = 0; i < high; i++) {
                    splitLine = arrayList.get(i).split(" ");
                    for (int j = 0; j < width; j++) {
                        array[i][j] = Float.parseFloat(splitLine[j]);
                    }
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }

            return array;
        }

        @Override
        public String toString() {
            return "FileToMatrices{" +
                    "\nA=" + Arrays.deepToString(A) +
                    ", \nB=" + Arrays.deepToString(B) +
                    "\n}";
        }
    }
```

2. MatricesMul.java 使用线程池实现多线程，并包含主函数进行实验

```
package com.os.zlj;

import sun.awt.windows.WPrinterJob;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MatricesMul {
    float[][] A, B;

    public MatricesMul(FileToMatrices ma) {
        A = ma.getA();
        B = ma.getB();
    }

    public void run() {
        run(1);
    }
    public float[][] run(int threadNum) {
        float[][] C = new float[A.length][B[0].length];
        long startTime, endTime;
        System.out.printf(
                "线程数：%2d, A: [%4d][%4d], B: [%4d][%4d],",
                threadNum,
                A.length,
                A[0].length,
                B.length,
                B[0].length
        );

        // 开始时间
```

```
34            startTime = System.currentTimeMillis();
35
36            // 使用线程池
37            ExecutorService pool = Executors.newFixedThreadPool(threadNum);
38            for (int i = 0; i < A.length; i++) {
39                int finalI = i;
40                // 把每一行加入线程池中
41                pool.submit(() -> {
42                    for (int j = 0; j < A[0].length; j++) {
43                        for (int k = 0; k < B[0].length; k++) {
44                            C[finalI][j] += A[finalI][k] * B[k][j];
45                        }
46                    }
47                });
48            }
49
50
51            // 等待线程池完成
52            pool.shutdown();
53            while(! pool.isTerminated());
54
55     //         System.out.println(Arrays.toString(C[0]));
56            // 结束时间
57            endTime = System.currentTimeMillis();
58            System.out.println("计算完成" + ",用时：" + (endTime - startTime)
    + "ms");
59
60            return C;
61        }
62        public static void main(String[] args) {
63            // 文件相对路径
64            String[][] fileNames = new String[][]{
65                    {"./src/resources/M64A.txt",
    "./src/resources/M64B.txt"},
66                    {"./src/resources/M128A.txt",
    "./src/resources/M128B.txt"},
67                    {"./src/resources/M512A.txt",
    "./src/resources/M512B.txt"},
68                    {"./src/resources/M1024A.txt",
    "./src/resources/M1024B.txt"}
69            };
70
71            System.out.println("在多次运行统一代码逻辑时，因为 java 内部的机制，第一
    次运行的的时间总会大于之后运行的时间 \n" +
72                    "在此先全部运行一次，但不记录结果，消除对之后进程的影响");
73            // 循环遍历，分别进行答单、四、十六线程
74            for (String[] files :
75                    fileNames) {
76                System.out.println(files[0] + ", " + files[1]);
77                FileToMatrices fileToMatrices1 = new
    FileToMatrices(files[0], files[1]);
78                MatricesMul matricesMul1 = new MatricesMul(fileToMatrices1);
79                matricesMul1.run(1);
80                matricesMul1.run(4);
81                matricesMul1.run(16);
82            }
83
84            System.out.println("-------------正式实验部分-----------------");
```

```
            int[] index = new int[]{1, 4, 16};
            for (int i : index) {
                System.out.println("------- 线程数为" + i + " ------");
                for (String[] files : fileNames) {
                    FileToMatrices fileToMatrices1 = new
    FileToMatrices(files[0], files[1]);
                    MatricesMul matricesMul1 = new
    MatricesMul(fileToMatrices1);
                    matricesMul1.run(i);
                }
            }

            // 循环遍历，分别进行答单、四、十六线程
            for (String[] files :
                    fileNames) {
                System.out.println(files[0] + ", " + files[1]);
                FileToMatrices fileToMatrices1 = new
    FileToMatrices(files[0], files[1]);
                MatricesMul matricesMul1 = new MatricesMul(fileToMatrices1);
                matricesMul1.run(1);
                matricesMul1.run(4);
                matricesMul1.run(16);
            }
        }
    }
```

实验分析：

1、列出步骤一的实验结果，并比较所花费的时间，讨论原因；

原因：计算矩阵的算法时间复杂度为O(n^3)，当矩阵的大小越大时，所用时间将会更长。

```
------- 线程数为1 ------
线程数:  1, A: [  64][  64], B: [  64][  64],计算完成,用时: 2ms
线程数:  1, A: [ 128][ 128], B: [ 128][ 128],计算完成,用时: 5ms
线程数:  1, A: [ 512][ 512], B: [ 512][ 512],计算完成,用时: 535ms
线程数:  1, A: [1024][1024], B: [1024][1024],计算完成,用时: 4472ms
```

2、列出步骤二的实验结果，并比较所花费的时间，讨论原因；

原因：计算矩阵的算法时间复杂度为O(n^3)，当矩阵的大小越大时，所用时间将会更长。

```
------- 线程数为4 -----
线程数:  4, A: [  64][  64], B: [  64][  64],计算完成,用时: 1ms
线程数:  4, A: [ 128][ 128], B: [ 128][ 128],计算完成,用时: 4ms
线程数:  4, A: [ 512][ 512], B: [ 512][ 512],计算完成,用时: 182ms
线程数:  4, A: [1024][1024], B: [1024][1024],计算完成,用时: 2935ms
```

3、列出步骤三的实验结果，并比较所花费的时间，讨论原因；

原因：计算矩阵的算法时间复杂度为O(n^3)，当矩阵的大小越大时，所用时间将会更长。

```
------- 线程数为16 ------
线程数：16, A: [  64][  64], B: [  64][  64],计算完成,用时：3ms
线程数：16, A: [ 128][ 128], B: [ 128][ 128],计算完成,用时：5ms
线程数：16, A: [ 512][ 512], B: [ 512][ 512],计算完成,用时：237ms
线程数：16, A: [1024][1024], B: [1024][1024],计算完成,用时：2799ms
```

4、针对 16 * 16,128 * 128，512*512矩阵，分别比较单线程、4线程和 16线程的耗时并进行分析。

分析：结果见下图，可以发现，对于16 * 16、128 * 128、512 * 512矩阵，所用时间并没有完全随着线程数目增加而减少，特别时对于16 * 16的矩阵来说，所用时间与线程数成正比。因为，对于16 * 16、128 * 128、512 * 512矩阵来说，计算量并不算大，当线程数目变多时，创建线程的消耗大于计算的消耗，使得存在时间随着线程数目增加而增加的情况。

```
./src/resources/M64A.txt, ./src/resources/M64B.txt
线程数：  1, A: [  64][  64], B: [  64][  64],计算完成,用时：1ms
线程数：  4, A: [  64][  64], B: [  64][  64],计算完成,用时：1ms
线程数：16, A: [  64][  64], B: [  64][  64],计算完成,用时：4ms
./src/resources/M128A.txt, ./src/resources/M128B.txt
线程数：  1, A: [ 128][ 128], B: [ 128][ 128],计算完成,用时：5ms
线程数：  4, A: [ 128][ 128], B: [ 128][ 128],计算完成,用时：3ms
线程数：16, A: [ 128][ 128], B: [ 128][ 128],计算完成,用时：4ms
./src/resources/M512A.txt, ./src/resources/M512B.txt
线程数：  1, A: [ 512][ 512], B: [ 512][ 512],计算完成,用时：429ms
线程数：  4, A: [ 512][ 512], B: [ 512][ 512],计算完成,用时：175ms
线程数：16, A: [ 512][ 512], B: [ 512][ 512],计算完成,用时：188ms
./src/resources/M1024A.txt, ./src/resources/M1024B.txt
线程数：  1, A: [1024][1024], B: [1024][1024],计算完成,用时：4422ms
线程数：  4, A: [1024][1024], B: [1024][1024],计算完成,用时：3514ms
线程数：16, A: [1024][1024], B: [1024][1024],计算完成,用时：4442ms
```

实验总结：

线程是进程中执行运算的最小单位，是资源调度的基本单位。使用线程可以提高并发性，相比于进程来说创建线程的资源更少，创建速度更快。但是使用多个线程，并不会一定使程序运行的速度更快，对于计算量较小的程序，使用多线程时，关于线程的操作的时间反而会大于计算的时间，使程序运行的速度反而更慢。所以，我们在使用多线程时，要根据实际情况来决定是否使用线程以及使用线程的数目。

遇到的问题

由于 JVM 虚拟机垃圾回收机制，同一操作消耗的时间可能不同。

例如，运行同样的代码三次

```
new MatricesMul(new FileToMatrices(fileNames[0][0], fileNames[0][1])).run();
new MatricesMul(new FileToMatrices(fileNames[0][0], fileNames[0][1])).run();
new MatricesMul(new FileToMatrices(fileNames[0][0], fileNames[0][1])).run();
```

得到的结果却截然不同，且越来越小

```
线程数:  1, A: [  64][  64], B: [  64][  64],计算完成,用时: 152ms
线程数:  1, A: [  64][  64], B: [  64][  64],计算完成,用时: 11ms
线程数:  1, A: [  64][  64], B: [  64][  64],计算完成,用时: 1ms
```

解决方案：

1. 使用平均数，但是对于64 * 64 这样较小的矩阵来说，运行时间本来就很短，并没有效果。
2. 多次计算直到临近的两个结果的插值小于一个极小数，但是运行效率太慢。

最终选择了一种比较简单且效率高的方式，先运用一遍代码，不记录结果，再运行一次即可。