JAVA 进阶第四次实验：矩阵相乘

一 实验要求

• 编写矩阵随机生成类 MatrixGenerator 类，随机生成任意大小的矩阵，矩阵单元使用 double 存储。

• 使用串行方式实现矩阵乘法。

• 使用多线程方式实现矩阵乘法。

• 比较串行和并行两种方式使用的时间，利用第三次使用中使用过的 jvm状态查看命令，分析产生时间差异的原因是什么。

二 实验过程

串行的计算矩阵比较简单，就是一个位置一个位置地计算，只要遵循矩阵乘法就可以了。

```
//首先算一波串行计算的结果
        startTime = System.currentTimeMillis();
        System.out.println(startTime);
        for(int i = 0; i< m1.length;i++) {
            for(int j = 0; j < m2[0].length;j++) {
                for(int k=0;k < m1[0].length;k++) {
                    presult[i][j] += m1[i][k]*m2[k][j];
                }
            }
        }
        endTime = System.currentTimeMillis();
        System.out.println(endTime);
```

然后就是使用多线程的方式实现矩阵乘法。为了描述方便，把第一个参与计算的矩阵称为A，第二个参与计算的矩阵称为B。我选择一条线程负责A的一行数据与B整个矩阵相乘，A有多少行就有多少条线程。首先实现最重要的类MatrixCalculation，由它来实现两个矩阵相乘的过程。

```
public class MatrixCalculation {
    double[][] matrix1 = null;//第一个矩阵
    double[][] matrix2 = null;//第二个矩阵
    double[][] result = null;//存放矩阵相乘结果
    public static int line = 0;//记录当前参与计算的行数有多少

    //定义构造方法
    public MatrixCalculation() {}

    public MatrixCalculation(double[][] m1,double[][] m2) {

        this.matrix1 = m1;
        this.matrix2 = m2;
        result = new double[matrix1.length][matrix2[0].length];
```

```
    }
}
```

以上就是MatrixCalculation类的构造方法。其中因为有很多线程参与并行计算，所以很有可能出现有的线程计算完了但是有的线程还在计算的情况。因为我需要输出计算结果与串行计算比较来验证正确性，如果直接使用getResult()方法，出来的结果可能不正确。所以line这个静态变量相当于一个计数器，每一行计算完了就加一，直到line与A的行数相等的时候再输出结果。

所以得到了getResult()函数：（写在MatrixCalculation类里面）

```
//返回矩阵相乘的结果
    public double[][] getResult() {
        try {
            while(MatrixCalculation.line<matrix1.length) {
                Thread.sleep(1);
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
        return this.result;
    }
```

接下来定义一行一行计算的函数。（写在MatrixCalculation类里面）

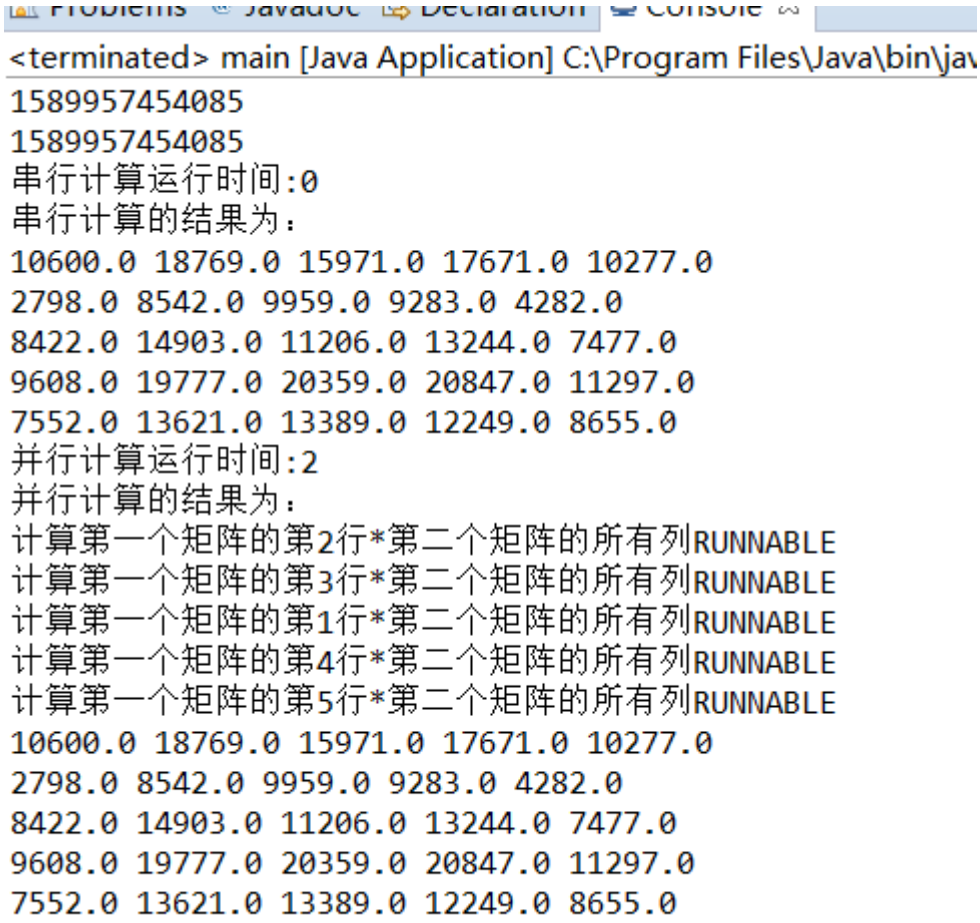
```
public void operate() {
        MatrixCalculation.line += 1;//记录这行已经相乘

        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-1][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-1][i] = sum;
        }
    }
```

MatrixCalculation类完成了，接下来写ThreadOperate类，用来调度线程工作。这里需要把MatrixCalculation类作为ThreadOperate的一个属性，因为重写run方法需要使用到MatrixCalculation类的operate()函数。

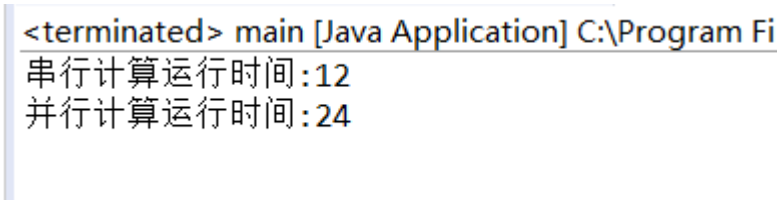
```
public class ThreadOperate extends Thread{
    private MatrixCalculation m = null;

    public ThreadOperate() {
        super();
    }

    public ThreadOperate(MatrixCalculation m,String name) {
        super(name);
        this.m = m;
    }
```

```
    @Override
    public void run() {
        try {

System.out.println(Thread.currentThread().getName()+Thread.currentThread().getState());//调用getState方法查看各个线程的状态，方便以后分析
        }catch(Exception e) {
            e.printStackTrace();
        }
        this.m.operate();
    }

}
```

接下来就是main测试类了，一开始用的数字比较小，都是5*5的矩阵，因为要打印串行和并行的运行结果，太大了控制台输出不方便观察。



可以看到并行串行的结果都是一样的。说明并行计算方法是一样的。但是并行计算的运行时间居然比串行时间还要长。一开始我以为是矩阵大小的关系，矩阵太小体现不出多线程的优势，于是我计算了100*100的矩阵。（因为矩阵大了控制台输出太多了，所以这次只输出运行时间）



可以看到并行计算运行时间远远大于串行计算。而且反而矩阵越大相差的时间越来越多。所以我猜测应该是我启动的线程太多了。因为每一行都启动一个线程，随着矩阵的变大，创建这么多线程和上下文切换太耗费时间了。所以我改进了代码，在operate函数里面不再是一个线程负责一行，而是一个线程负责两行。

修改之后的operate（）函数：

```
public void operate() {
        MatrixCalculation.line += 2;//一次加两行

        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-2][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-2][i] = sum;
        }
        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-1][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-1][i] = sum;
        }
    }
```

使用10*10的矩阵先看一下这种计算方法有没有错误：

```
1589958630022
1589958630022
串行计算运行时间:0
串行计算的结果为：
26761.0 35562.0 25541.0 26486.0 32604.0 33248.0 28077.0 21423.0 21909.0 25818.0
24313.0 22962.0 22880.0 19285.0 24076.0 26049.0 21568.0 25719.0 22811.0 19734.0
34470.0 48322.0 37080.0 35099.0 38804.0 44484.0 35754.0 33761.0 27028.0 33386.0
29668.0 39286.0 29568.0 26134.0 34834.0 37780.0 30968.0 29892.0 25146.0 30174.0
24301.0 32542.0 24122.0 21395.0 27486.0 30311.0 27247.0 27217.0 17990.0 23172.0
18723.0 24165.0 21207.0 19120.0 25048.0 24591.0 24825.0 23148.0 17011.0 24026.0
20184.0 29901.0 21356.0 21093.0 26864.0 28620.0 26681.0 21063.0 16893.0 24064.0
30763.0 35195.0 30468.0 30435.0 39818.0 34620.0 34748.0 27753.0 32003.0 27899.0
27632.0 32352.0 32249.0 29847.0 31416.0 33570.0 27199.0 29058.0 27310.0 29259.0
21186.0 30632.0 25977.0 21703.0 25070.0 28217.0 23286.0 18038.0 18148.0 25625.0
计算第一个矩阵的第1行*第二个矩阵的所有列RUNNABLE
计算第一个矩阵的第3行*第二个矩阵的所有列RUNNABLE
并行计算运行时间:2
计算第一个矩阵的第2行*第二个矩阵的所有列RUNNABLE
计算第一个矩阵的第5行*第二个矩阵的所有列RUNNABLE
并行计算的结果为：
计算第一个矩阵的第4行*第二个矩阵的所有列RUNNABLE
26761.0 35562.0 25541.0 26486.0 32604.0 33248.0 28077.0 21423.0 21909.0 25818.0
24313.0 22962.0 22880.0 19285.0 24076.0 26049.0 21568.0 25719.0 22811.0 19734.0
34470.0 48322.0 37080.0 35099.0 38804.0 44484.0 35754.0 33761.0 27028.0 33386.0
29668.0 39286.0 29568.0 26134.0 34834.0 37780.0 30968.0 29892.0 25146.0 30174.0
24301.0 32542.0 24122.0 21395.0 27486.0 30311.0 27247.0 27217.0 17990.0 23172.0
18723.0 24165.0 21207.0 19120.0 25048.0 24591.0 24825.0 23148.0 17011.0 24026.0
20184.0 29901.0 21356.0 21093.0 26864.0 28620.0 26681.0 21063.0 16893.0 24064.0
30763.0 35195.0 30468.0 30435.0 39818.0 34620.0 34748.0 27753.0 32003.0 27899.0
27632.0 32352.0 32249.0 29847.0 31416.0 33570.0 27199.0 29058.0 27310.0 29259.0
21186.0 30632.0 25977.0 21703.0 25070.0 28217.0 23286.0 18038.0 18148.0 25625.0
```

两行的计算结果依然正确，接下来还是计算100*100的：

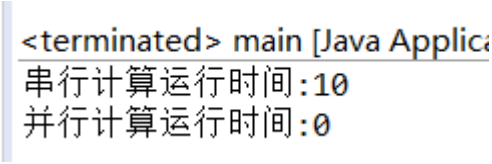
```
<terminated> main [Java Applicati
串行计算运行时间:10
并行计算运行时间:10
```

这次两种方法直接相等了，说明我上面的猜想正确。这次把线程改成一次计算5行，计算结果：

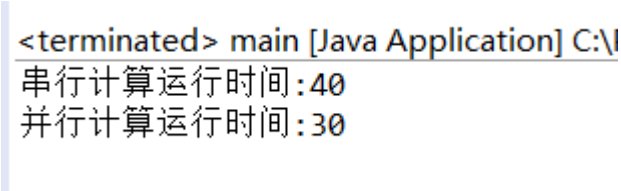
50*50的两个矩阵相乘：

```
<terminated> main [Java Applicatic
串行计算运行时间:9
并行计算运行时间:0
```

100*100的两个矩阵相乘：



200*200的两个矩阵相乘：



三 实验结果

通过上面的分析，我们得到

当矩阵一样大的时候，线程数越多，矩阵相乘运行的时间会越短。但是线程的数量并不是越多越好，因为频繁地创造线程以及切换上下文会使额外的时间增加，总的时间反而会比较少线程的程序运行的慢。

当线程数一样的多的时候，矩阵越大，无论并行串行，矩阵相乘的时间会越长。而且运行的时间随着矩阵的长或者宽呈指数增长。

至于为什么会出现这种结果，使用 jvm状态查看命令可以知道如果是串行，那么结果就是一个线程一个一个算出来的。只有等待上一个数算完了下一个数才能开始算。如果是并行，就可以同时计算好几个数，不必等待上个数算完。

四 代码附录

```
package MatrixCalculation;

public class MatrixCalculation {
    double[][] matrix1 = null;//第一个矩阵
    double[][] matrix2 = null;//第二个矩阵
    double[][] result = null;//存放矩阵相乘结果
    public static int line = 0;//记录当前参与计算的是第一个矩阵的第几行

    //定义构造方法
    public MatrixCalculation() {}

    public MatrixCalculation(double[][] m1,double[][] m2) {

        this.matrix1 = m1;
        this.matrix2 = m2;
        result = new double[matrix1.length][matrix2[0].length];
    }

    //返回矩阵相乘的结果
    public double[][] getResult() {
        try {
            while(MatrixCalculation.line<matrix1.length) {
                Thread.sleep(1);
```

```
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
        return this.result;
    }

    public void operate() {
        MatrixCalculation.line += 5;

        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-5][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-5][i] = sum;
        }
        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-4][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-4][i] = sum;
        }
        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-3][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-3][i] = sum;
        }
        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-2][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-2][i] = sum;
        }
        for(int i = 0; i < matrix1[0].length;i++) {
            double sum = 0;
            for(int j = 0; j < matrix2.length; j++) {
                sum += matrix1[MatrixCalculation.line-1][j]*matrix2[j][i];
            }
            result[MatrixCalculation.line-1][i] = sum;
        }
    }

}
```

```
package MatrixCalculation;

public class ThreadOperate extends Thread{
    private MatrixCalculation m = null;
```

```
    public ThreadOperate() {
        super();
    }

    public ThreadOperate(MatrixCalculation m,String name) {
        super(name);
        this.m = m;
    }

    @Override
    public void run() {
        try {

//System.out.println(Thread.currentThread().getName()+Thread.currentThread().getState());
        }catch(Exception e) {
            e.printStackTrace();
        }
        this.m.operate();
    }

}
```

```
package MatrixCalculation;

public class main {
    public static void main(String args[]) {
        int m1row = 200;
        int m1col = 200;
        int m2col = 200;
        double[][] m1 = new double[m1row][m1col];
        double[][] m2 = new double[m1col][m2col];
        double[][] presult = new double[m1row][m2col];
        MatrixCalculation m = new MatrixCalculation(m1,m2);
        long startTime;
        long endTime;

        for (int i = 0; i < m1row; i++)
            for (int j = 0; j < m1col; j++) {
                m1[i][j] = (int) (Math.random() * 100);
            }
        for (int i = 0; i < m1col; i++)
            for (int j = 0; j < m2col; j++) {
                m2[i][j] = (int) (Math.random() * 100);
            }
        //首先算一波串行计算的结果
        startTime = System.currentTimeMillis();
        //System.out.println(startTime);
        for(int i = 0; i< m1.length;i++) {
            for(int j = 0; j < m2[0].length;j++) {
                for(int k=0;k < m1[0].length;k++) {
                    presult[i][j] += m1[i][k]*m2[k][j];
                }
```

```
            }
        }
        endTime = System.currentTimeMillis();
        //System.out.println(endTime);
        System.out.println("串行计算运行时间:" + (endTime - startTime));
        //System.out.println("串行计算的结果为: ");
        //for(int i = 0; i < m1row; i++) {
        //  for(int j =0; j < m2col;j++) {
        //      System.out.print(presult[i][j]+" ");
        //  }
        //  System.out.println();
        //}

        //并行计算
        startTime = System.currentTimeMillis();
        for(int i = 0; i <m1.length/5;i++) {
            new ThreadOperate(m,"计算第一个矩阵的第"+(i+1)+"行*第二个矩阵的所有列").start();
        }
        endTime = System.currentTimeMillis();
        System.out.println("并行计算运行时间:" + (endTime - startTime));
        //System.out.println("并行计算的结果为: ");
        //presult = m.getResult();
        //for(int i = 0; i < m1row; i++) {
        //  for(int j =0; j<m2col;j++) {
        //      System.out.print(presult[i][j]+" ");
        //  }
        //  System.out.println();
        //}
    }
}
```