第一题

**试编写一个多线程的程序：分别用Thread子类和Runnable接口的方式，启动4个线程。其中两个循环100次,每次将同一个成员变量（或静态成员变量）加一个整数,另两个循环100次,每次将此变量减一个相同的整数。请输出该变量的变化结果。（可以增加循环次数，计算之间用sleep增加不同时长的间隔等等进行尝试。如果加减的整数是1，加减采用++，--操作符的效果又是怎样？）分析一下结果**

package hw8.proj1;

class SharedCounter {

    private int count = 0;

    // synchronized to prevent race conditions

    public synchronized void add(int value) {

        count += value;

        System.out.println("Added: " + value + " | Current count: " + count);

    }

    public synchronized void subtract(int value) {

        count -= value;

        System.out.println("Subtracted: " + value + " | Current count: " + count);

    }

    public int getCount() {

        return count;

    }

}

class AddThread extends Thread {

    private SharedCounter counter;

    public AddThread(SharedCounter counter) {

        this.counter = counter;

    }

    @Override

    public void run() {

        for (int i = 0; i < 100; i++) {

            counter.add(1);

            try {

                Thread.sleep((long) (Math.random() \* 100));

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

class SubtractThread extends Thread {

    private SharedCounter counter;

    public SubtractThread(SharedCounter counter) {

        this.counter = counter;

    }

    @Override

    public void run() {

        for (int i = 0; i < 100; i++) {

            counter.subtract(1);

            try {

                Thread.sleep((long) (Math.random() \* 100));

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

class AddRunnable implements Runnable {

    private SharedCounter counter;

    public AddRunnable(SharedCounter counter) {

        this.counter = counter;

    }

    @Override

    public void run() {

        for (int i = 0; i < 100; i++) {

            counter.add(1);

            try {

                Thread.sleep((long) (Math.random() \* 100));

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

class SubtractRunnable implements Runnable {

    private SharedCounter counter;

    public SubtractRunnable(SharedCounter counter) {

        this.counter = counter;

    }

    @Override

    public void run() {

        for (int i = 0; i < 100; i++) {

            counter.subtract(1);

            try {

                Thread.sleep((long) (Math.random() \* 100));

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

public class MultiThreadDemo {

    public static void main(String[] args) {

        SharedCounter counter = new SharedCounter();

        // Using Thread subclasses

        AddThread addThread1 = new AddThread(counter);

        AddThread addThread2 = new AddThread(counter);

        SubtractThread subThread1 = new SubtractThread(counter);

        SubtractThread subThread2 = new SubtractThread(counter);

        // Using Runnable interface

        Thread addRunnableThread1 = new Thread(new AddRunnable(counter));

        Thread addRunnableThread2 = new Thread(new AddRunnable(counter));

        Thread subRunnableThread1 = new Thread(new SubtractRunnable(counter));

        Thread subRunnableThread2 = new Thread(new SubtractRunnable(counter));

        // Start threads

        addThread1.start();

        addThread2.start();

        subThread1.start();

        subThread2.start();

        addRunnableThread1.start();

        addRunnableThread2.start();

        subRunnableThread1.start();

        subRunnableThread2.start();

    }

}

运行结果如图1-1所示：

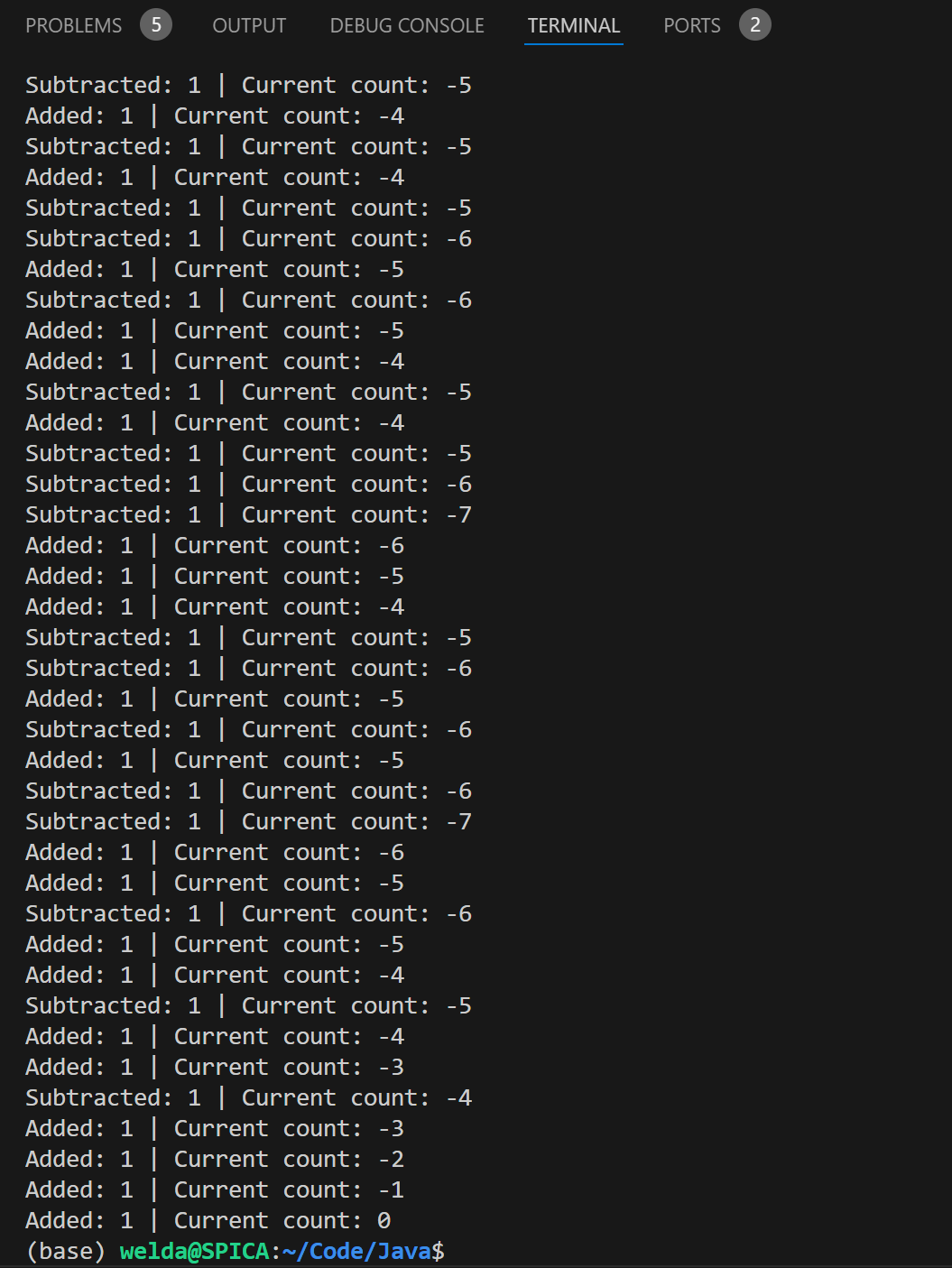


图1-1

分析：在上述代码中，通过启动多个线程对共享变量 count 进行加减操作。在没有 synchronized 的情况下，可能会出现竞态条件，导致 count 的值不可预测。如果移除 synchronized 关键字，结果会非常混乱，因为线程会互相打断执行。

如果使用++或—操作符，仍然需要使用synchronized来确保正确的行为，多线程环境下如果直接使用，很可能会导致错误结果。

**第二题**

**开放性作业**

**（1）尝试修改主业1的程序，加各种锁，观察并简单解释运行结果**

**（2）将作业1的结果（任选一实现多线程的方式）简化为2个线程。用wait和notify配合synchronized控制线程的工作方式，记录并简单解释运行结果**

(1)

代码：

package hw8.proj2;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class SharedCounterWithLock {

    private int count = 0;

    private Lock lock = new ReentrantLock();

    public void add(int value) {

        lock.lock();

        try {

            count += value;

            System.out.println("Added: " + value + " | Current count: " + count);

        } finally {

            lock.unlock();

        }

    }

    public void subtract(int value) {

        lock.lock();

        try {

            count -= value;

            System.out.println("Subtracted: " + value + " | Current count: " + count);

        } finally {

            lock.unlock();

        }

    }

    public int getCount() {

        return count;

    }

}

public class MultiThreadWithLockDemo {

    public static void main(String[] args) {

    SharedCounterWithLock counter = new SharedCounterWithLock();

    // 创建线程

    Thread addThread1 = new Thread(() -> {

        for (int i = 0; i < 100; i++) {

            counter.add(1);

            try {

                Thread.sleep((long) (Math.random() \* 100));

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    });

    Thread subThread1 = new Thread(() -> {

        for (int i = 0; i < 100; i++) {

            counter.subtract(1);

            try {

                Thread.sleep((long) (Math.random() \* 100));

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    });

    // 启动线程

    addThread1.start();

    subThread1.start();

    try {

        addThread1.join();

        subThread1.join();

    } catch (InterruptedException e) {

        e.printStackTrace();

    }

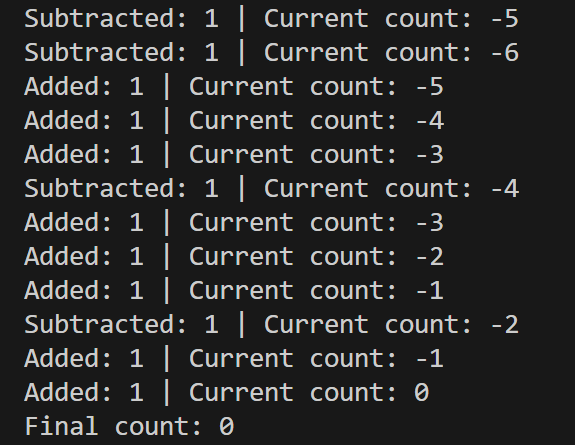
    System.out.println("Final count: " + counter.getCount());

    }

}

结果分析：

没有同步机制时，当多个线程同时访问共享变量的时候，会导致最终count的值不稳定。使用synchronized后，保证每次对共享变量的操作是原子性的，ReentrantLock同样也可以和synchronized一样确保线程的同步。



（2）

package hw8.proj2;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class SharedCounterWithWaitNotify {

    private int count = 0;

    private boolean isAddTurn = true; // 控制哪个线程应该执行

    private final Object lock = new Object();

    public void add(int value) {

        synchronized (lock) {

            while (!isAddTurn) {

                try {

                    lock.wait(); // 等待直到轮到加法线程执行

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

            count += value;

            System.out.println("Added: " + value + " | Current count: " + count);

            isAddTurn = false; // 改变标志，让减法线程运行

            lock.notify(); // 通知另一个线程

        }

    }

    public void subtract(int value) {

        synchronized (lock) {

            while (isAddTurn) {

                try {

                    lock.wait(); // 等待直到轮到减法线程执行

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

            count -= value;

            System.out.println("Subtracted: " + value + " | Current count: " + count);

            isAddTurn = true; // 改变标志，让加法线程运行

            lock.notify(); // 通知另一个线程

        }

    }

    // Add the missing getCount() method

    public int getCount() {

        return count;

    }

}

public class WaitNotifyDemo {

    public static void main(String[] args) {

        SharedCounterWithWaitNotify counter = new SharedCounterWithWaitNotify();

        Thread addThread = new Thread(() -> {

            for (int i = 0; i < 100; i++) {

                counter.add(1);

            }

        });

        Thread subThread = new Thread(() -> {

            for (int i = 0; i < 100; i++) {

                counter.subtract(1);

            }

        });

        // 启动线程

        addThread.start();

        subThread.start();

        try {

            addThread.join();

            subThread.join();

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        System.out.println("Final count: " + counter.getCount());

    }

}

结果分析：程序中 isAddTurn 标志位确保了加法和减法操作的线程能够有序地执行，每个线程在不该自己执行时进入等待状态。这种方法不仅解决了同步问题，还确保了线程的交替性，避免了两个线程同时修改共享变量的情况。

