第 2 次作业

一. 实验目的：

1. 练习 Java 多线程编程技术。
2. 练习实现网络并发服务的编程技术。
3. 学习如何实现多线程间的相互同步和相互协作。
4. 理解什么是线程安全。

二. 设计要求

1. 功能概述：实现一个支持并发服务的**网络运算服务器**程序。该服务器能够同时接收来自于多个客户端的运算请求，然后根据运算类型和请求参数完成实际的运算，最后把运算结果返回给客户端。

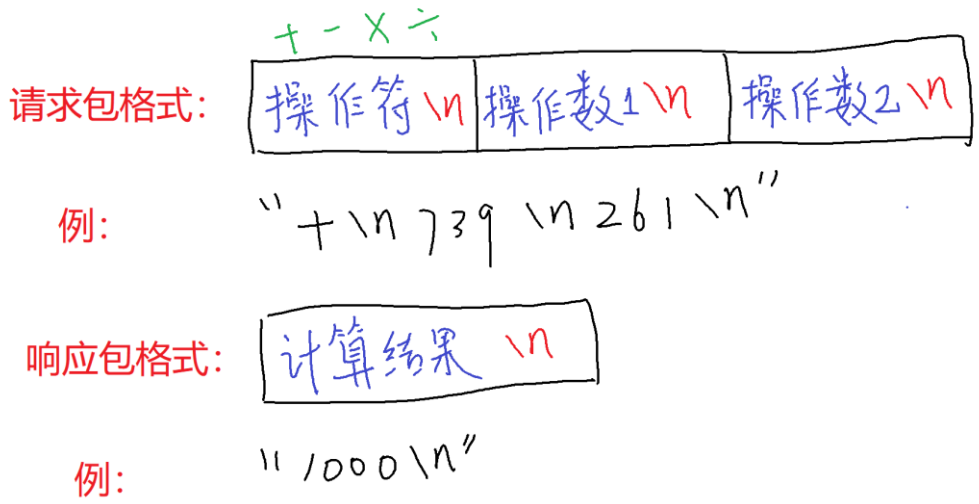
2. 具体要求：

（1）至少支持加、减、乘、除四种基本运算。

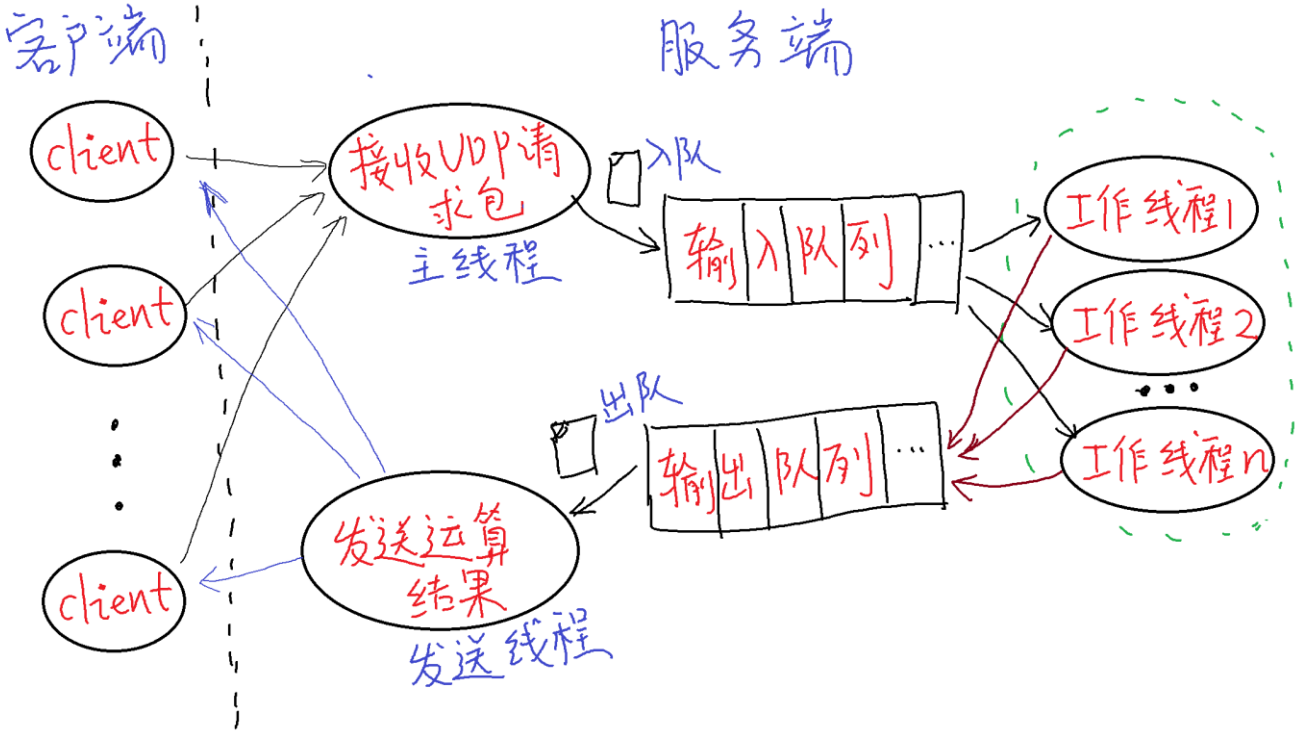
（2）服务器端能够分别记录已经成功处理的不同运算类型请求的个数。

（2）客户端与服务器端之间基于 UDP 协议进行通信。

（3）应用层协议**自行设计**。例如请求数据包、响应数据包可以采用如下格式：



（4）服务器端程序**必须采用如下结构：**



三. 提交要求

1. **3 月 15 日**前将**源程序（包含客户端程序和服务端程序）**和**简单的设计报告**打包后发送至邮箱：xd_distri_comp@163.com。设计报告内容主要包括：（1）设计思想，把应用层协议设计描述清楚。（2）遇到的问题和解决方法。报告中不要大段地粘贴源代码。

2. 邮件标题风格：第 2 次作业+学号+姓名

3. 打包文件命名方式：第 2 次作业+学号+姓名.zip

四. 技术提示

1. 关于线程安全

一个对象或对象某个成员方法（函数）被多个线程同时访问时还能保持正确性，即还能完成预定义的功能，则称该对象或该成员方法是**线程安全**的。

下面定义了一个用于记录系统成功处理的交易数量的 TransactionCounter 类。其使用方法为：创建 TransactionCounter 类的一个实例 tc 作为公共变量。当处理完一个交易后就调用 tc 对象的 increase()方法将交易计数器加一。以上设计逻辑上没有问题，但不幸的是 increase()方法不是线程安全的。（为什么？）

```
public class TransactionCounter {
    private int counter;
    public TransactionCounter() {
        counter=0;
    }
    public int increase() {
        return counter++;
    }
}
```

要想让某个类的成员方法称为线程安全的，一种简单粗暴的办法是在该成员方法定义前面加上“**synchronized**”关键字。用 synchronized 修饰的成员方法在同一时间内只能有 1 个线程进入该方法，如果同时有两个线程调用该方法，则其中一个必须等待先进入的退出后才能进入该方法。这种在同一时间内只能有 1 个线程进入的代码段叫“临界区”（Critical Section）。

```
public class TransactionCounter {
    private int counter;
    public TransactionCounter() {
        counter=0;
    }
    public synchronized int increase() {        // Thread Safe
        return counter++;
    }
}
```

2. 关于阻塞队列

Java 中提供的常用的队列结构：java.util.LinkedList 不是线程安全的。实现作业中要求的功能时可以使用线程安全的并且支持阻塞机制的 LinkedBlockingQueue 队列类。当一个线程对已经满了的 LinkedBlockingQueue 队列进行入队操作（调用 put 函数）时，会被阻塞，除非另外一个线程进行了出队操作。或者当一个线程对一个空的 LinkedBlockingQueue 队列进行出队操作（调用 take 函数）的时候，会被阻塞，除非另外一个线程进行了入队的操作。

使用 LinkedBlockingQueue 队列的示例代码如下。

```
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class Demo {
	public static void main(String[] args) {
		BlockingQueue<String> blockingQueue = new LinkedBlockingQueue();
		new Thread(new Producer(blockingQueue), "First Producer").start();
		new Thread(new Consumer(blockingQueue), "First Consumer").start();
		new Thread(new Consumer(blockingQueue), "Second Consumer").start();
	}
}

// Producer
class Producer implements Runnable {
	private	BlockingQueue<String> queue;
	public Producer(BlockingQueue<String> queue) {
		this.queue = queue;
	}
	public void run() {
		int i = 0;
		while (true) {
			try {
				String product = String.valueOf(i) ;
				queue.put(product);
				System.out.println("Thread(" + Thread.currentThread().getName() + ")
produced: " + product + "; We now have " + queue.size() + " products.");
				try {
					Thread.sleep(500);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			i++;
```

```
            }
        }
}

// Consumer
class Consumer implements Runnable {
        private    BlockingQueue<String> queue;
        public Consumer(BlockingQueue<String> queue) {
            this.queue = queue;
        }

        public void run() {
            String tempStr = null;
             while (true) {
                  try {
                        tempStr = queue.take();
                        System.out.println("Thread(  "  +  Thread.currentThread().getName()  +  ") consumed:" + tempStr + "; We now have " + queue.size() + " products.");
                        try {
                              Thread.sleep(2000);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }

            }
        }
}
```

3. 创建线程的常用方法 1：通过扩展 Thread 类来创建多线程

a) 创建继承于 Thread 类（JDK 实现的基类）的子类。子类需要重载 run 函数，作为自己的线程主函数。

b) 利用 Thread 子类的构造函数接收一些初始化状态或初始化参数。这些初始化参数一般

被保存在 Thread 子类对象的私有变量中。线程主函数（run 函数）运行时可以访问这些参数，因此这相当于间接地为 run 函数传递运行参数。

```
public class MutliThreadDemo {
    public static void main(String [] args){
        MutliThread m1=new MutliThread(100, "Window 1");
        MutliThread m2=new MutliThread(200, "Window 2");
        MutliThread m3=new MutliThread(150, "Window 3");
        m1.start();
        m2.start();
        m3.start();
    }
}
class MutliThread extends Thread{
    private int ticket;//每个线程都拥有的票数
    MutliThread(int t, String name){
        this.ticket = t;
        super(name);//调用父类带参数的构造方法，name 参数设置了线程的名字
    }
    public void run(){
        while(ticket>0){
            System.out.println(ticket--+" is saled by "+Thread.currentThread().getName());
        }
    }
}
```

4. 创建线程的常用方法 2：通过实现 Runnable 接口来创建多线程

a) 创建一个实现 **Runnable** 接口的 Java 类，在该类中重载 run 函数，如下面示例中的 RunFunction 类。这个 Java 类相当于对 run 函数做了一个封装，并且可以利用私有成员为 run 函数保存一些运行参数和运行状态（如下面示例中的 ticket 变量、name 变量。）。

```
public class MutliThreadDemo2 {
    public static void main(String [] args){
        RunFunction r1=new RunFunction(100, "Window 1");
        RunFunction r2=new RunFunction(200, "Window 2");
        RunFunction r3=new RunFunction(150, "Window 3");
        Thread t1=new Thread(r1);
```

```
            Thread t2=new Thread(r2);
            Thread t3=new Thread(r3);
            t1.start();
            t2.start();
            t3.start();
        }
}
class RunFunction implements Runnable{
        private int ticket;
        private String name;
        RunFunction(int t, String name){
            this.name=name;
           this.ticket=t;
        }
        public void run(){
            while(ticket>0){
                System.out.println(ticket--+" is saled by "+name);
            }
        }
}
```