实验九：多线程扫描技术

——1801090006 郭盈盈

一、实验题目

请结合本次实验的资料，进一步优化上次实验中的端口扫描工具，**使其可以利用多线程技术**实现端口扫描功能。本次实验**编程语言不限，必须要有界面**，不要提交整个工程文件。提交时请上传**实验报告、源代码**（.cpp、.py 或.java 文件等）。

二、相关知识

线程

线程提供了运行一个任务的机制。对于 Java 而言，可以在一个程序中并发地启动多个线程。这些线程可以在多处理器系统上同时运行。线程是为了**同步**完成多项任务，不是为了**提高运行效率**，而是为了提高资源使用效率来提高系统的效率。线程是在**同一时间需要完成多项任务**的时候实现的。

最简单的比喻多线程就像火车的每一节车厢，而进程则是火车。车厢离开火车是无法跑动的，同理火车也不可能只有一节车厢。多线程的出现就是为了提高效率。同时它的出现也带来了一些问题。

同步与互斥

相交进程之间的关系主要有两种，同步与互斥。所谓互斥，是指散步在不同进程之间的若干程序片断，当某个进程运行其中一个程序片段时，其它进程就不能运行它 们之中的任一程序片段，只能等到该进程运行完这个程序片段后才可以运行。所谓同步，是指散步在不同进程之间的若干程序片断，它们的运行必须严格按照规定的 某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。

显然，同步是一种更为复杂的互斥，而互斥是一种特殊的同步。

也就是说互斥是两个线程之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步也是不能同时运行，但他是必须要安照某种次序来运行相应的线程（也是一种互斥）！

总结：互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

代码部分

端口多线程扫描实现，先插入一个内部类 ScanThread。

```
class ScanThread implements Runnable{
    private String IP;
    private int startPort,endPort;//开始端口，结束端口
    private int threadNumber, serial, timeout; // 线程数，这是第几个线程，超时时间
    private String openPort = "";//开放的端口信息

    public ScanThread(String IP,int startPort,int endPort,
            int threadNumber,int serial,int timeout) {
        this.IP = IP;
        this.startPort = startPort;
        this.endPort = endPort;
        this.threadNumber = threadNumber;
        this.serial = serial;
        this.timeout = timeout;
        this.openPort = "";
    }

    public void run() {
    }
```

run()的实现就是上次实验里单个 IP 地址下端口扫描的端口扫描实现代码。

接着用线程池实现多线程的执行。

```
int threadNumber = 5;
long startTime = System.currentTimeMillis(); //获取开始时间
ExecutorService threadPool = Executors.newFixedThreadPool(threadNumber);
for (int i = 0; i < threadNumber; i++) {
    ScanThread scanThread = new ScanThread(currIP,
        Integer.parseInt(tfBegin.getText().trim()),
        Integer.parseInt(tfEnd.getText().trim()),
        threadNumber, i, Integer.parseInt(tfTime.getText().trim()));
    threadPool.execute(scanThread);
}
threadPool.shutdown();
// 每秒中查看一次是否已经扫描结束
try {
    while (!threadPool.isTerminated()) {
        Thread.sleep(1000);
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

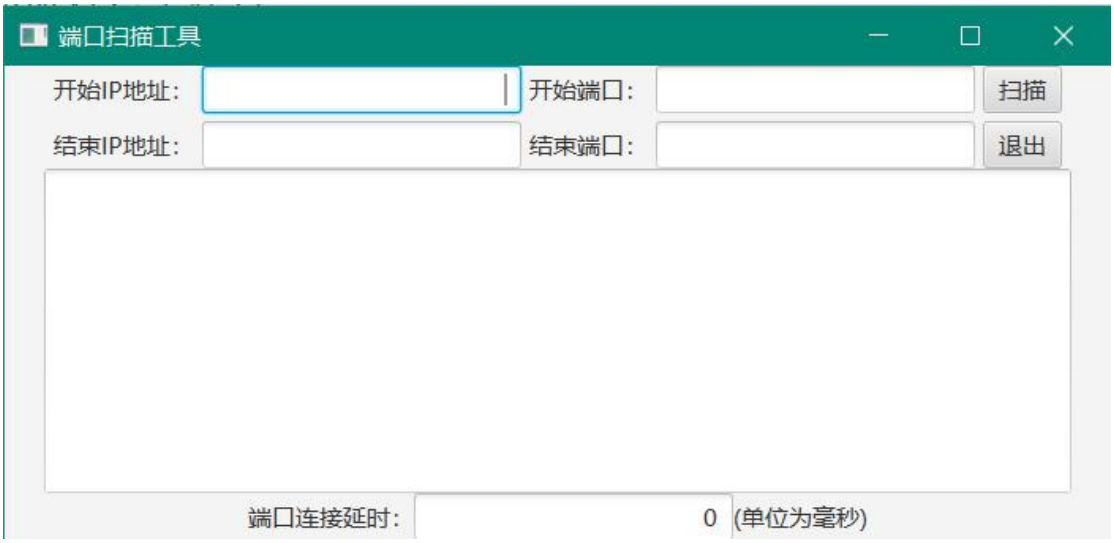
记得用 isTerminated()检查是否扫描结束，不然会一直没有调用下一个线程。

最后由扫描一个 IP 地址改为扫描一段连续的 IP 地址段。将 IP 地址以“.”切割成 4 个整数，然后在连接起来。

```
private int part1,part2,part3,part4;
private String ip[];
public MyIP(String address) {
    String ip[] = address.split("\\.");
    this.part1 = Integer.parseInt(ip[0]);
    this.part2 = Integer.parseInt(ip[1]);
    this.part3 = Integer.parseInt(ip[2]);
    this.part4 = Integer.parseInt(ip[3]);
}
public static String getIP(int part1,int part2,
        int part3,int part4 ) {
    return "" + part1 + "." +
            part2 + "." + part3 + "." + part4;
}
```

三、实验步骤与结果分析

界面



输入效果

运行显示

四、实验收获与总结

1.遇到了 java.util.concurrent.RejectedExecutionException 的问题

从异常名称里分析出是提交的任务被线程池拒绝了。在分析 java.util.concurrent.RejectedExecutionException 之前，需要深入学习一下 ThreadPoolExecutor 的使用。

核心池和最大池的大小

TreadPoolExecutor 将根据 corePoolSize 和 maximumPoolSize 设置的边界自动调整池大小。当新任务在方法 execute(java.lang.Runnable)中提交时，如果运行的线程少于 corePoolSize，则创建新线程来处理请求，即使其他辅助线程是空闲的。如果运行的线程多于 corePoolSize 而少于 maximumPoolSize，则仅当队列满时才创建新的线程。如果设置的 corePoolSize 和 maximumPoolSize 相同，则创建了固定大小的线程池。如果将 maximumPoolSize 设置为基本的无界值（如 Integer.MAX_VALUE），则允许线程池适应任意数量的并发任务。

保持活动时间

如果池中当前有多于 corePoolSize 的线程，则这些多出的线程在空闲时间超过 keepAliveTime 时将会终止。

排队

所有 BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：

如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝（抛出 RejectedExecutionException）。

排队有三种通用策略：

1. 直接提交。工作队列的默认选项是 synchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增加的可能性。
2. 无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize（因此，maximumPoolSize 的值也就无效了）。
3. 有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度的降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞，则系统可能为超过您许可的更多线程安排时间，使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样可会降低吞吐量。

终止

程序不再引用的池没有剩余线程会自动 shutdown。如果希望确保回收取消引用的池（即使用户忘记调用 shutdown()），则必须安排未使用的线程最终终止。

分析

通过对 ThreadPoolExecutor 类分析，引发 java.util.concurrent.RejectedException 主要有两种原因：

1. 线程池显示的调用了 shutdown()之后，再向线程池提交任务的时候，如果你配置的拒绝策略是 ThreadPoolExecutor.AbortPolicy 的话，这个异常就被会抛出来。

2. 当你的排队策略为有界队列，并且配置的拒绝策略是ThreadPoolExecutor.AbortPolicy，当线程池的线程数量已经达到了maximumPoolSize的时候，你再向它提交任务，就会抛出ThreadPoolExecutor.AbortPolicy异常。