第一节 详解NIO的前世今生-原理解析

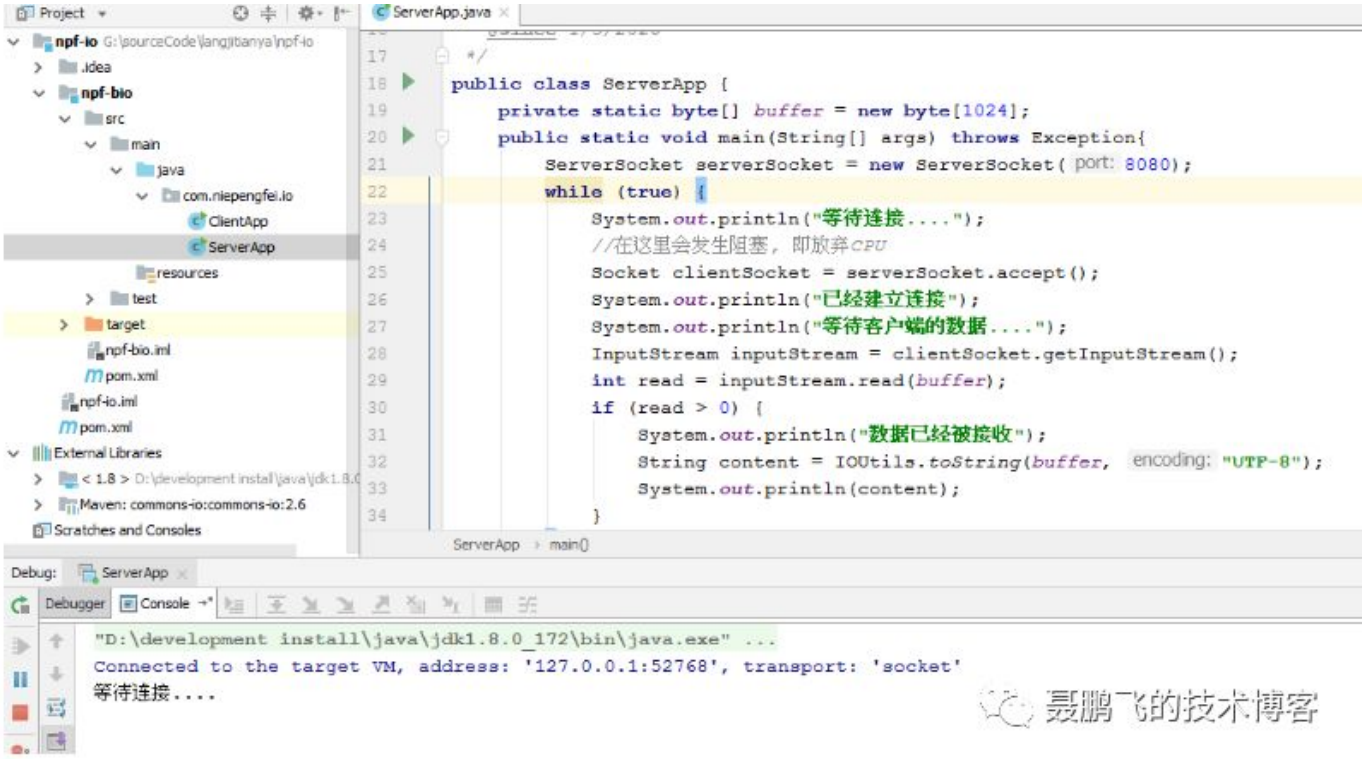
1. 初步了解BIO

说起BIO，我们就来谈一下网络编程，其实网络编程，很多人学的都很差。网络编程的基本模型是C/S模型，即两个进程间的通信。服务端提供IP和监听端口，客户端通过连接操作向服务端监听的地址发起连接请求，通过三次握手连接，如果连接成功建立，双方就可以通过套接字进行通信。传统的同步阻塞模型开发中，ServerSocket负责绑定IP地址，启动监听端口；Socket负责发起连接操作。连接成功后，双方通过输入和输出流进行同步阻塞式通信。
本文的git连接：
https://github.com/pengfeinie/npf-io

1.1 服务端

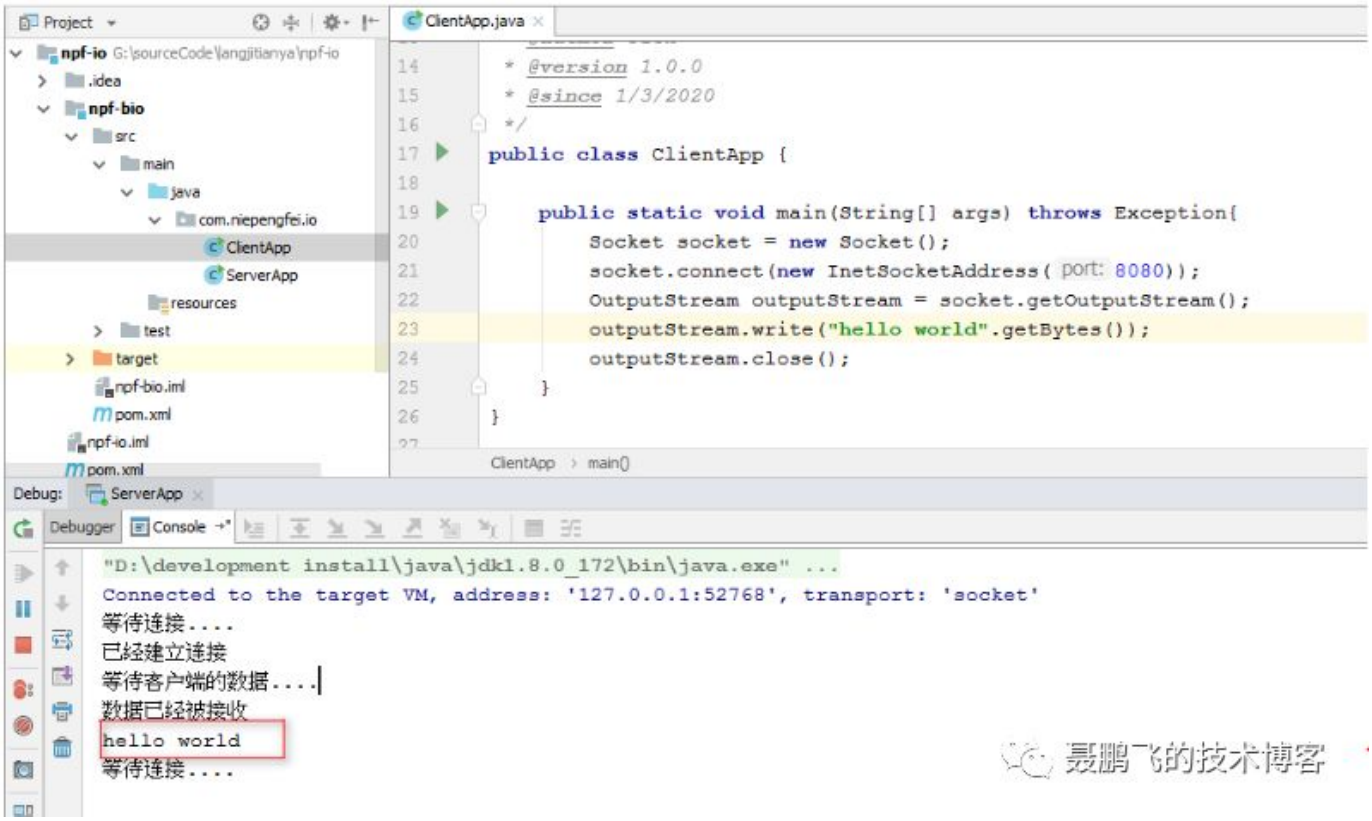
现在我们来写一个很简单的BIO实现的服务端。当我们把服务端启动的时候，服务端将会阻塞在下面的这行代码，目的是等待客户端来连接。

```
serverSocket.accept();
```



1.2 客户端

当我们运行客户端，服务端将打印出客户端发来的数据，然后程序再次阻塞，等待客户端来连接，因为我们的服务端采用了while 循环。

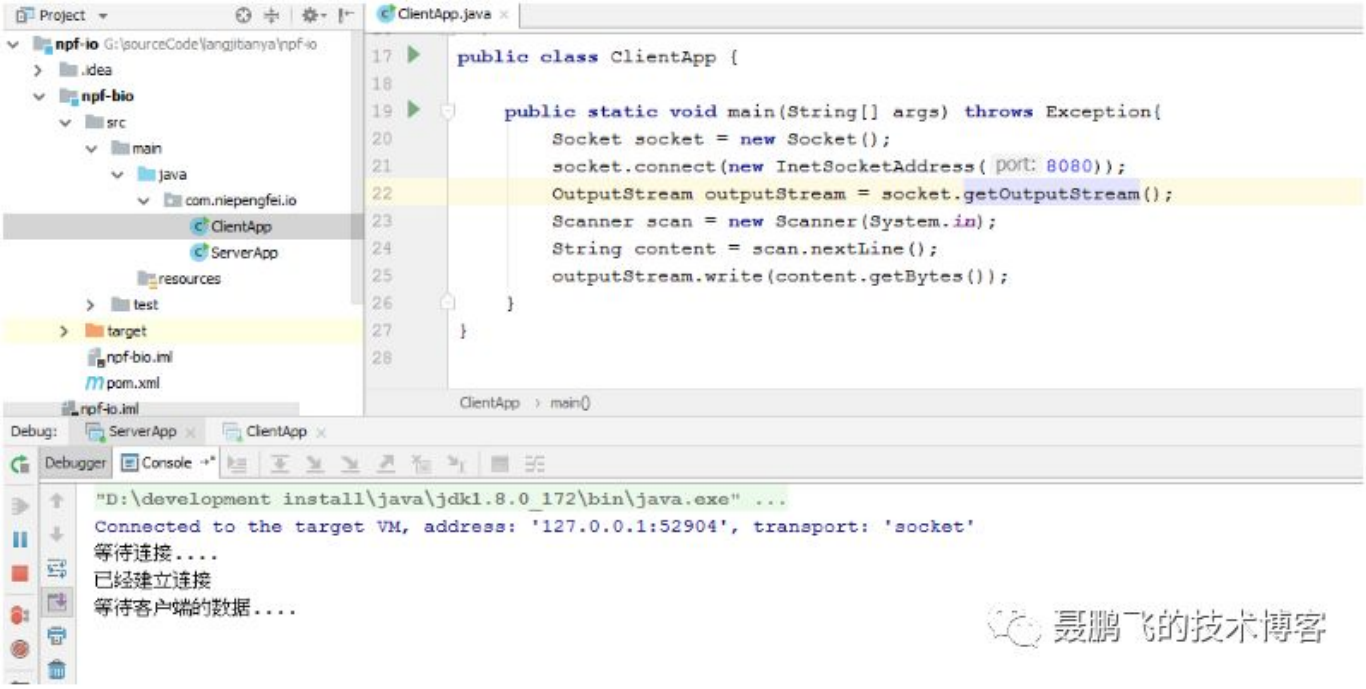


1.3 BIO在单线程情况下无法解决并发问题

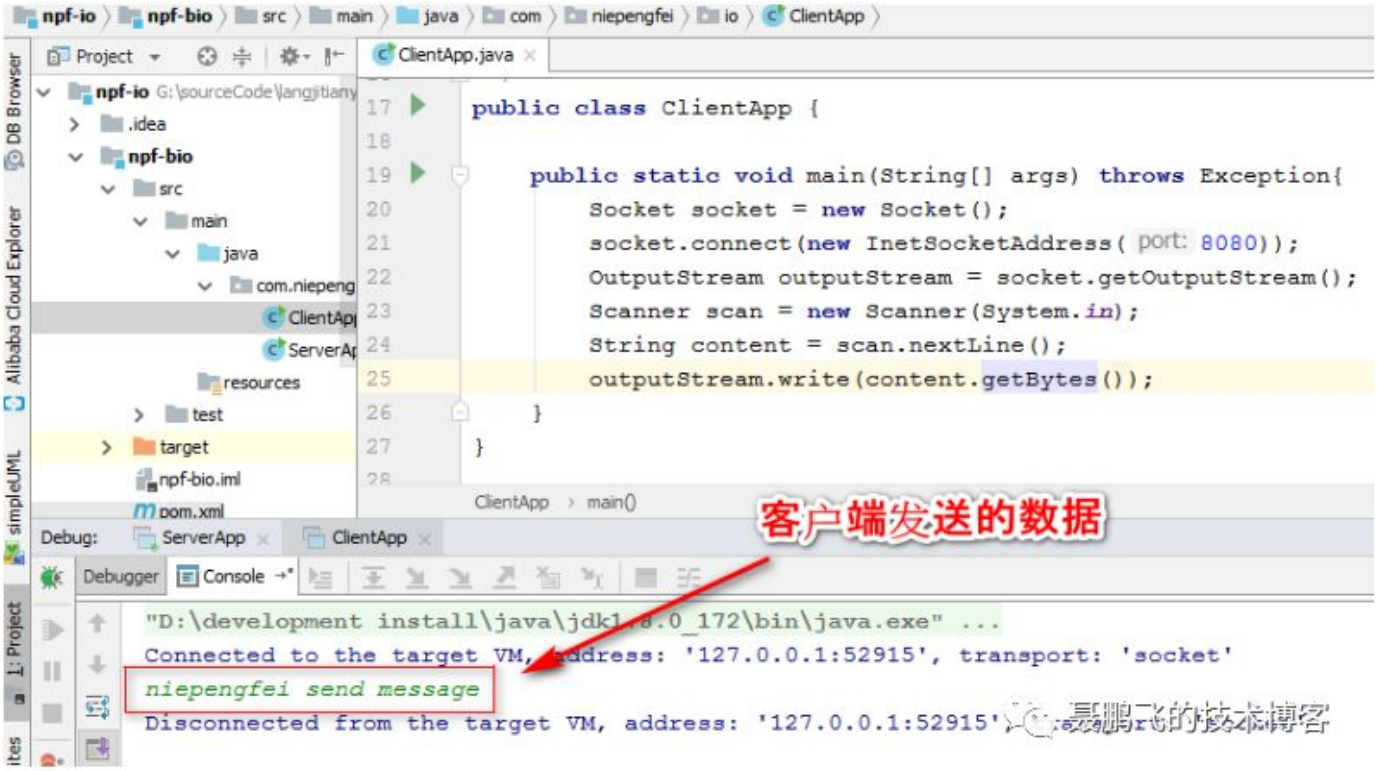
客户端假设如上面所写那样，你永远都体会不到一点，那就是服务端的以下代码也是阻塞的，即read方法是阻塞的。为什么这么说呢？上述那个客户端的例子一点都不好，可以说非常的差劲，但是几乎所有学习BIO编程的Java程序员都是从这个例子中学习过来的，我真想把最开始写这个例子程序的人摁到地上摩擦，简直是误人子弟。

inputStream.read(buffer);

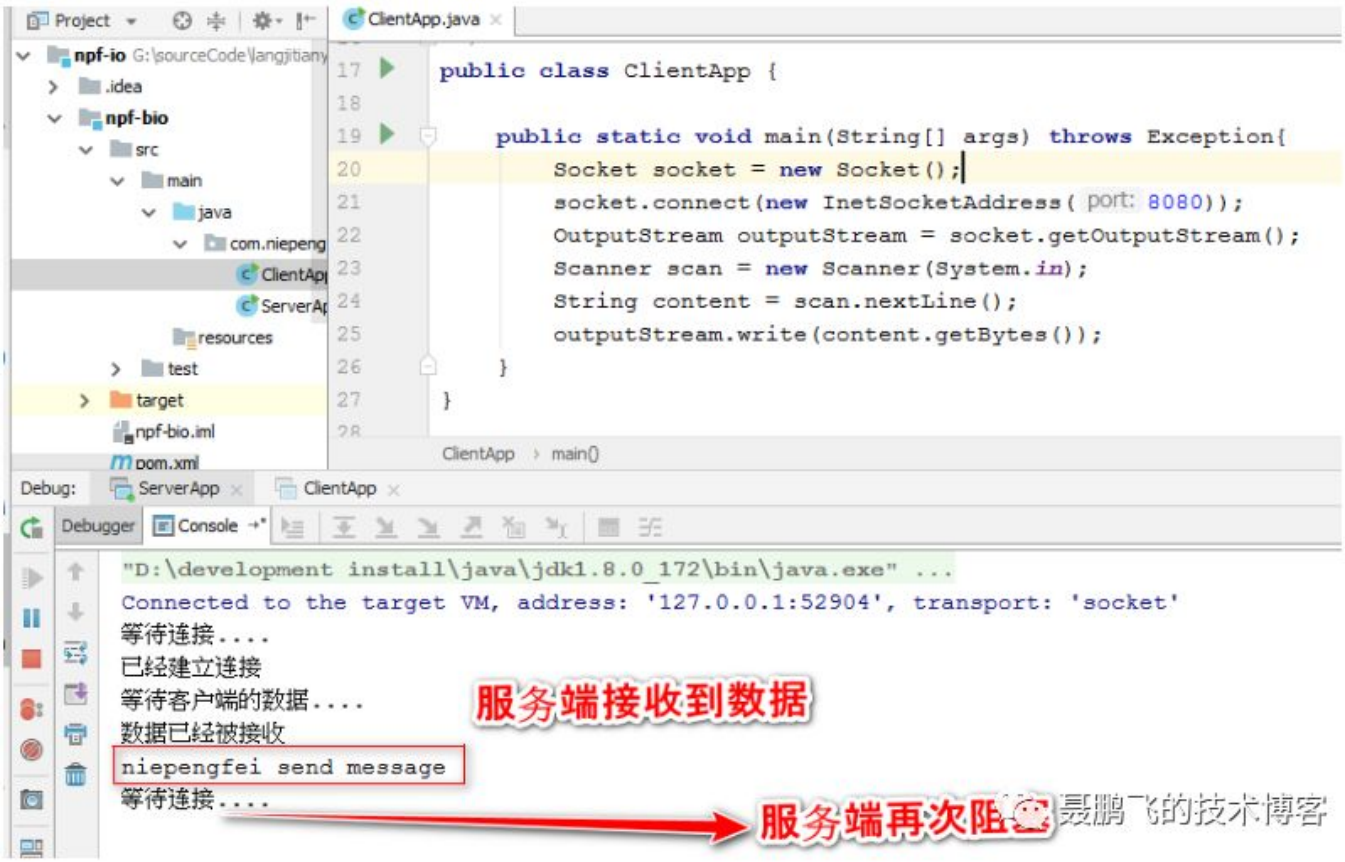
我们来看一下，我把客户端的代码修改一下。让客户端连接上服务端的时候，不要立即去发数据，而是在那里等待一会，这样你就可以很明显的看到，服务端将会阻塞在read方法上面。不信你可以看看，当我们运行客户端的时候，服务端的控制台将会打印如下信息，这就足以说明服务端已经阻塞在read方法上面，等待客户端发送数据。



之后，我在客户端的控制台发送数据，随即你可以在服务端看到接收的数据，然后服务端打印出来之后，再次阻塞在accept方法上面，等待客户端的链接。



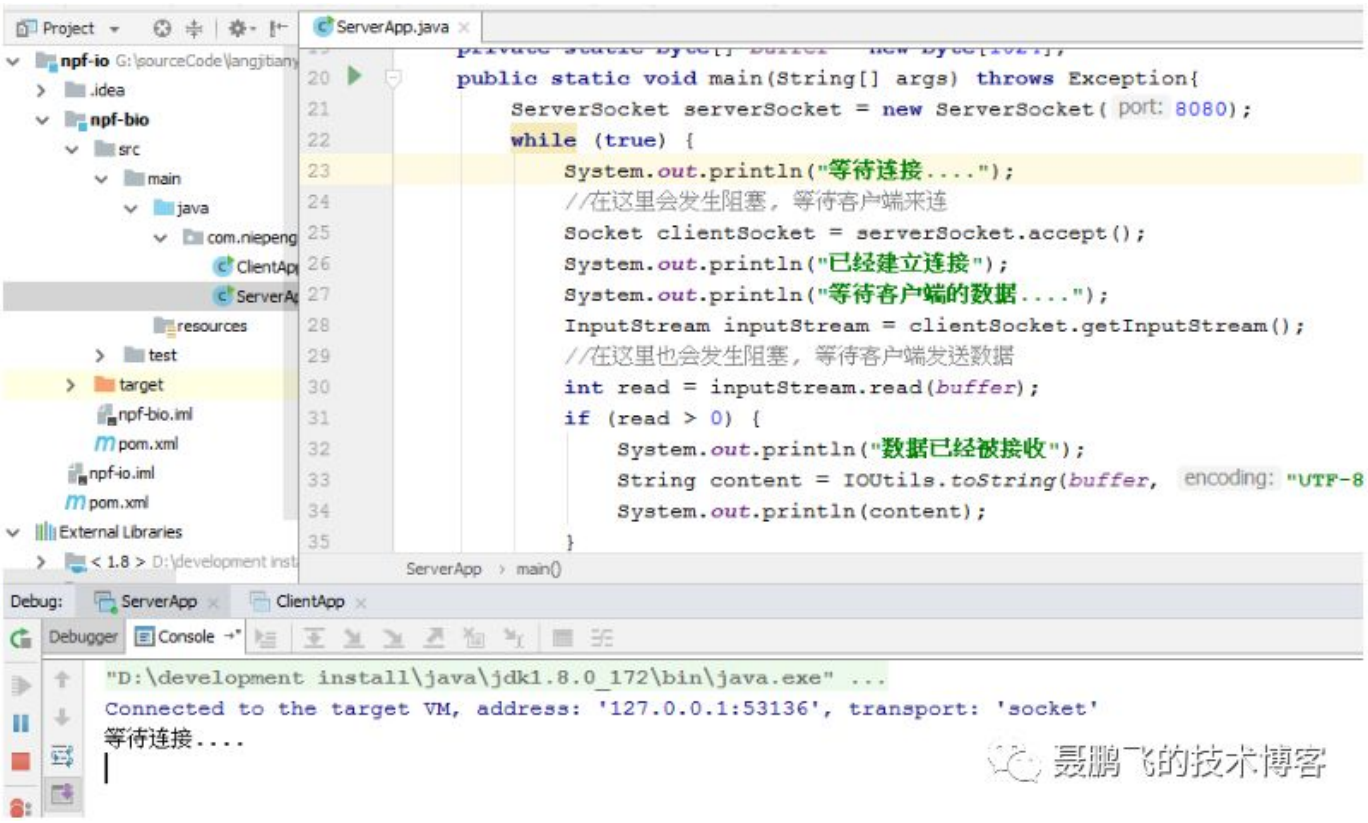
然后我们再到服务的控台看下：



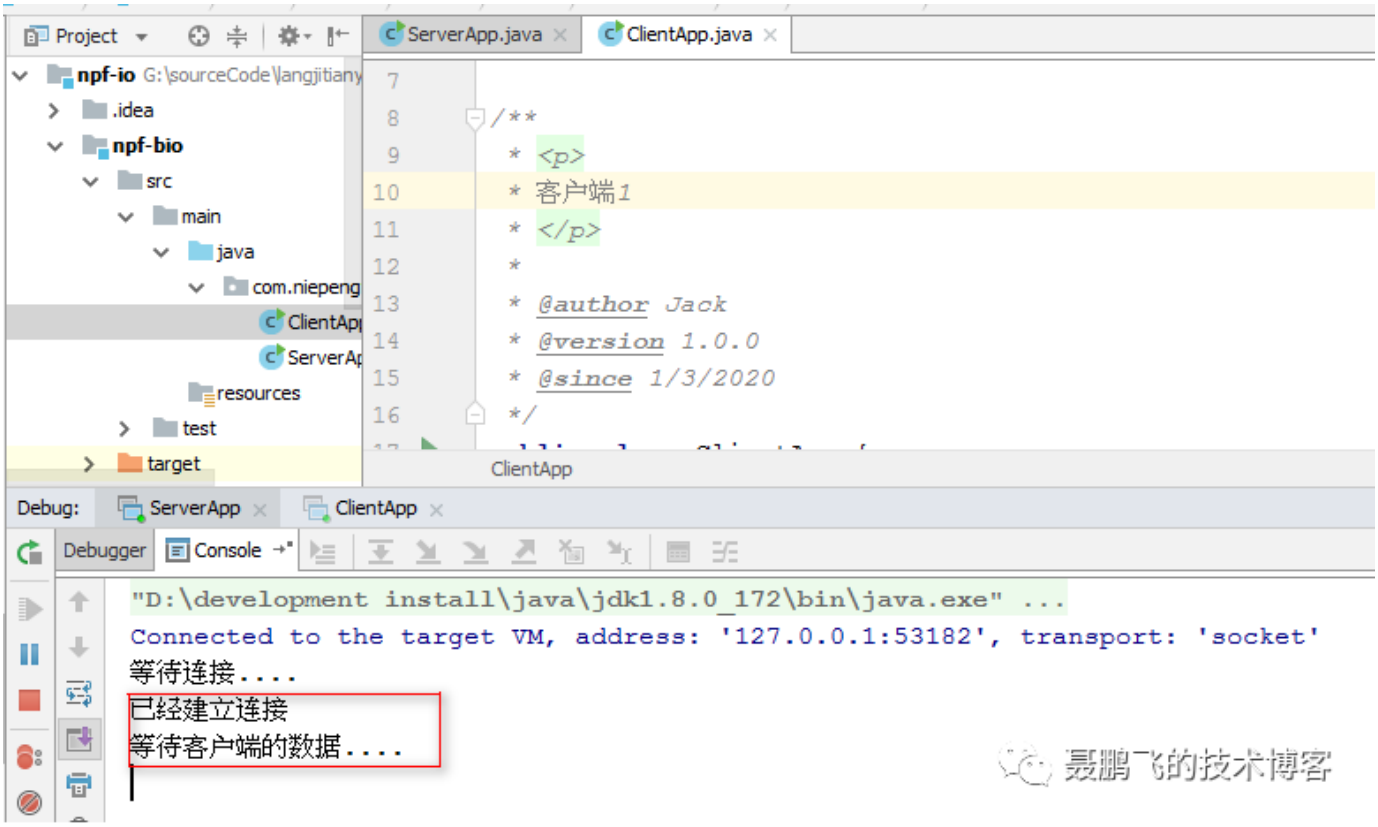
通过以上分析，采用传统的BIO编程的话，会造成两个地方阻塞。第一个就是accept，等待客户端来连。第二个就是read，等待客户端发送数据。这两个阻塞会造成什么问题呢？那就决定了如果要让BIO实现并发，那么就必须要借助多线程。写到这里，可能还是有人不明白，我真的是醉了，好吧。如果你还不相信，那么下面我就用例子来证明这些理论，好吧。

1.3.1 如何证明

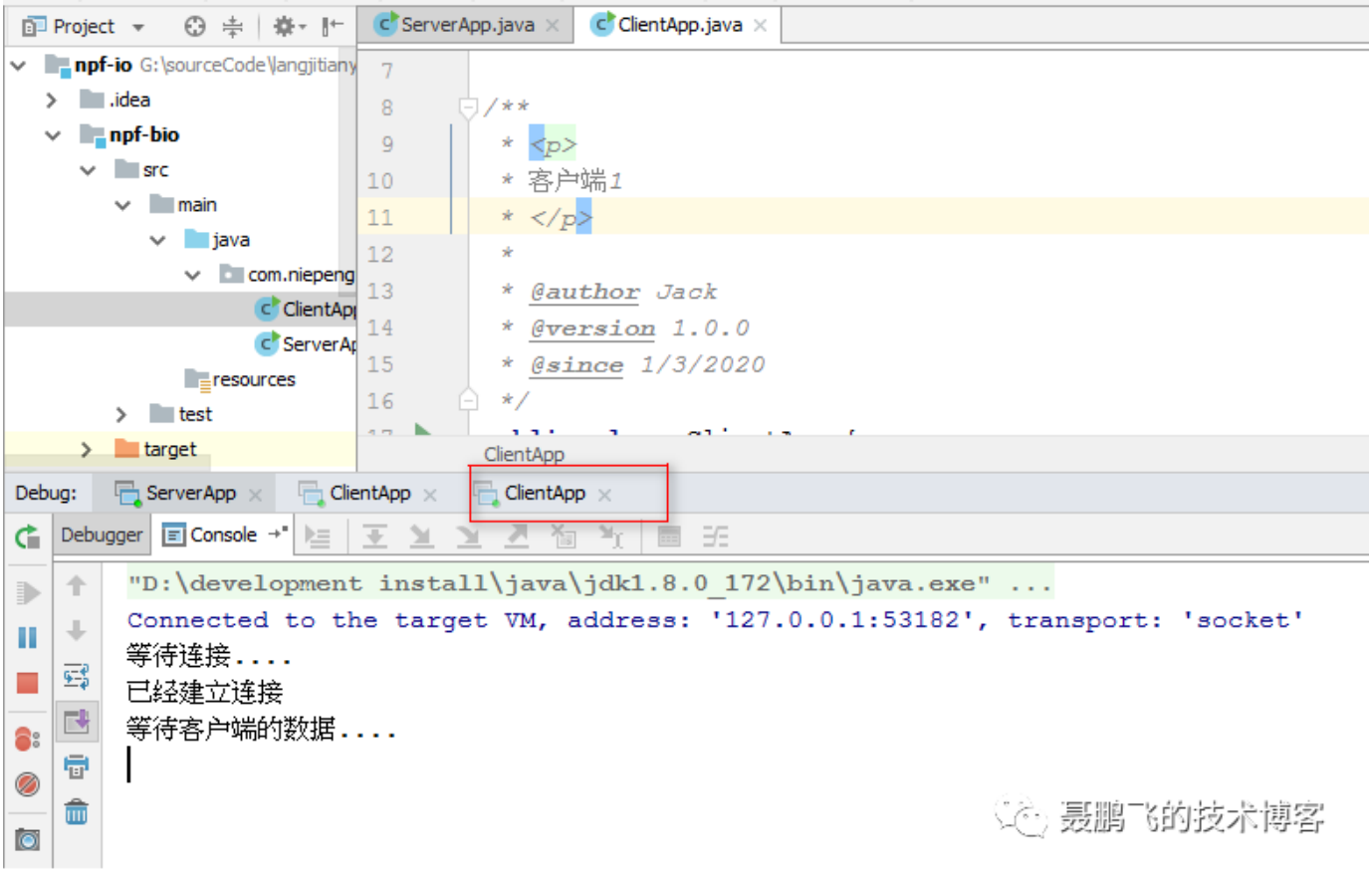
第一步：我们启动服务端。那么服务端必须阻塞在accept方法上面，等待客户端来连。



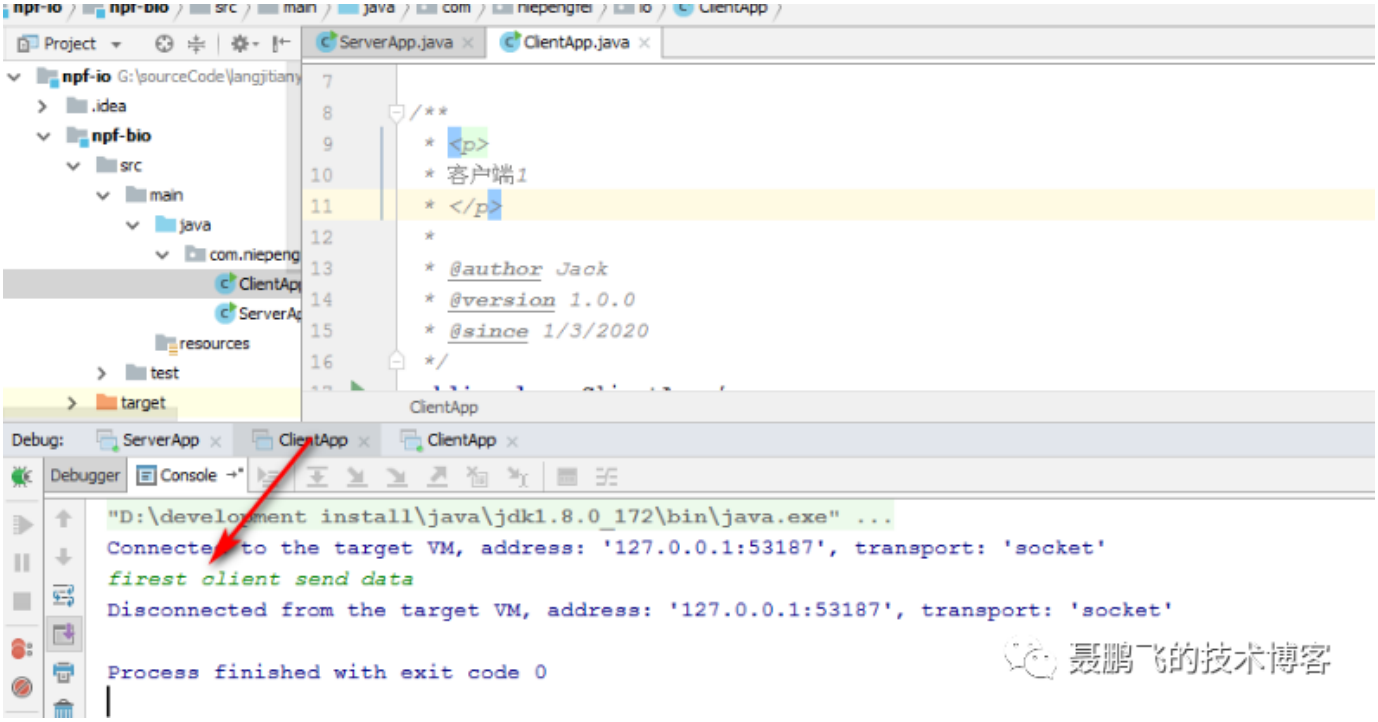
第二步：我们启动客户端，连接到服务端之后，一直不发送数据，让服务端一直阻塞在read方法上面。如果能在服务端控制台看到如下信息，那就证明我们的客户端连接上来了。



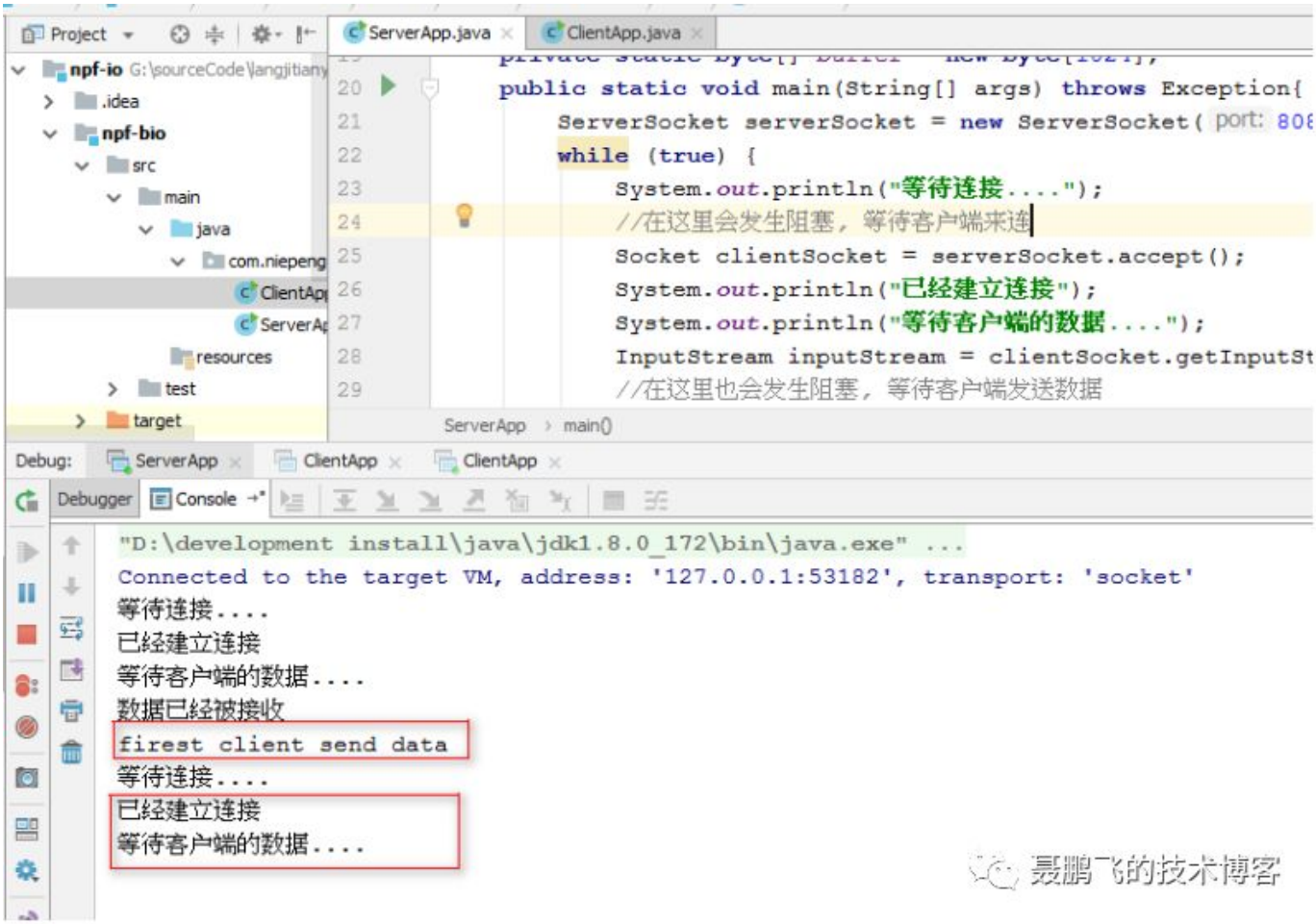
第三步：我们再启动另一个客户端，发现怎么也连接不上服务端。因为服务端没有打印任何信息，目前服务端控制台出现的信息还是第一个客户端连接上来的时候打印的。那么这个服务端相当于瘫痪了，因为没有任何客户端可以连接上来了。直到我们的第一个客户端发数据上来了，第二个客户端才可以连接上，不信你可以继续往下看。



第四步：第一个客户端发送数据给服务端。



这个时候，你请看服务端的控台。第一个客户端的数据已经接收到了，第二个客户端也连接上来了。

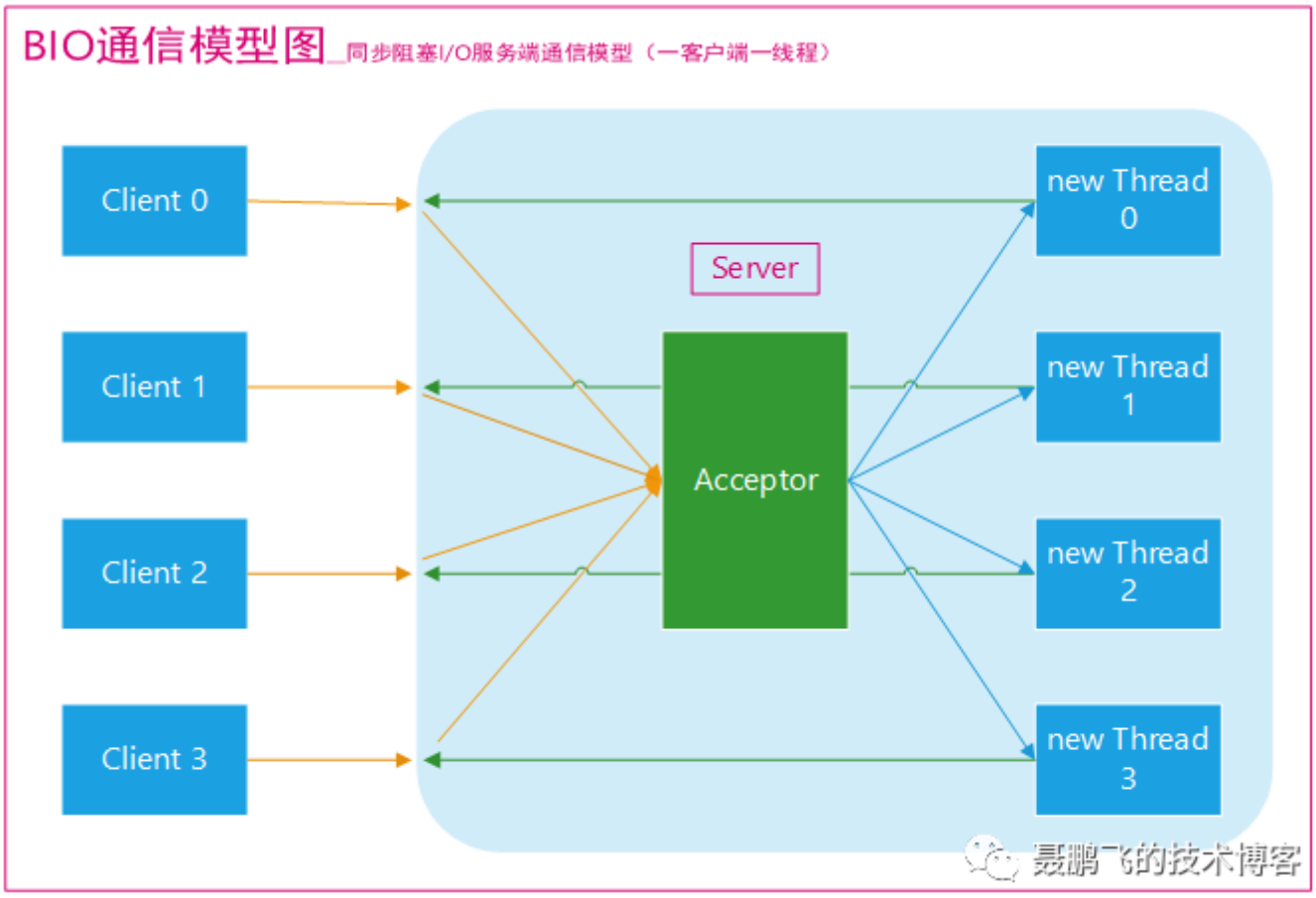


说白了，BIO在单线程的情况下，是不能实现并发的，因为它在accept和read方法上面阻塞了。那如何解决呢？请看下面。

1.4 BIO在多线程情况下可以解决并发问题

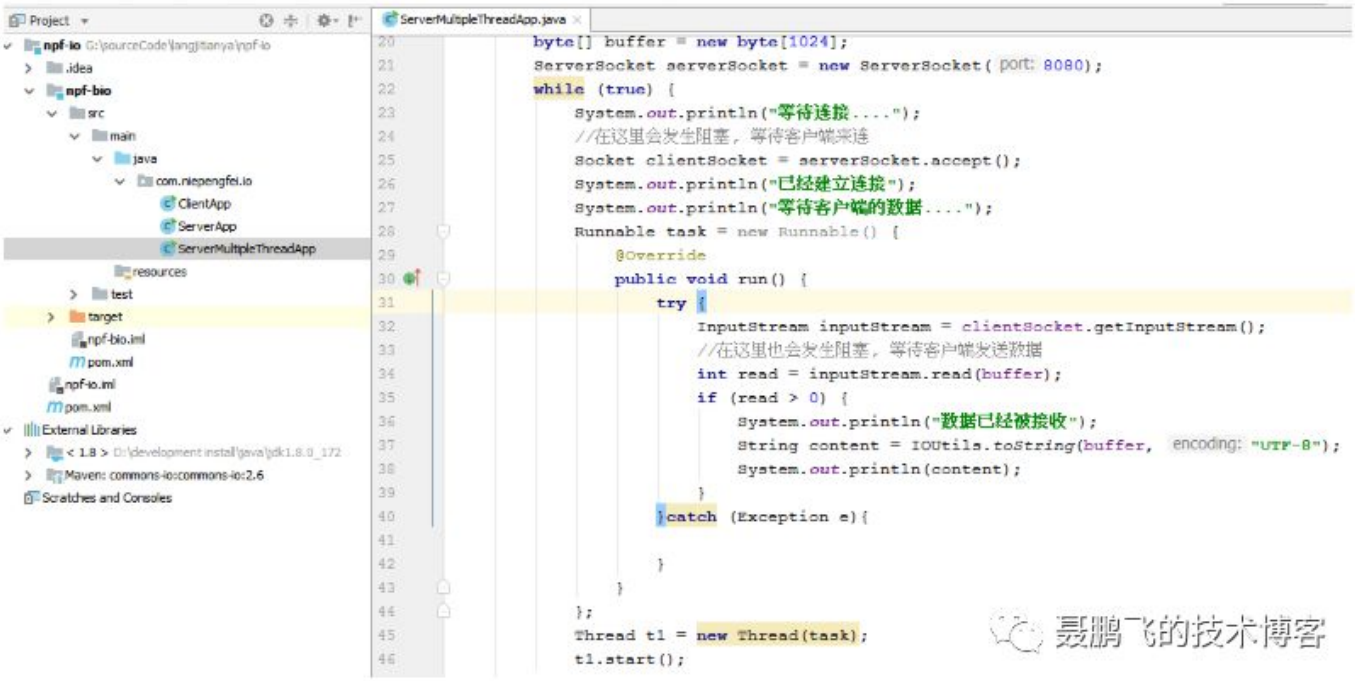
采用 BIO **通信模型** 的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接。我们一般通过在`while(true)` 循环中，服务端会调用 `accept()` 方法等待接收客户端的连接的方式监听请求，一旦接收到一个连接请求，就可以建立通信套接字在这个通信套接字上进行读写操作，此时不能再接收其他客户端连接请求，只能等待同当前连接的客户端的操作执行完成， 不过可以通过多线程来支持多个客户端的连接，如下图所示。

如果要让 BIO **通信模型** 能够同时处理多个客户端请求，就必须使用多线程（主要原因是`socket.accept()`、`socket.read()`、`socket.write()` 涉及的三个主要函数都是同步阻塞的），也就是说它在接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的 **一请求一应答通信模型** 。



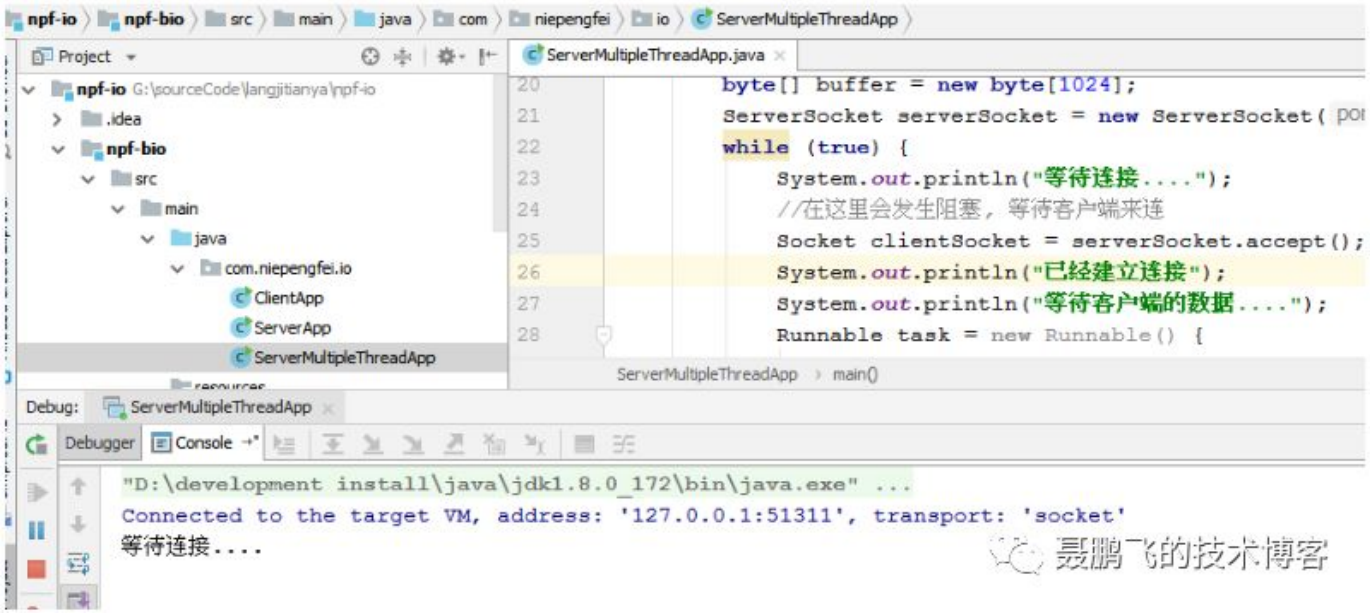
1.4.1 代码层面如何解决

服务端采用多线程方式。如下所示：

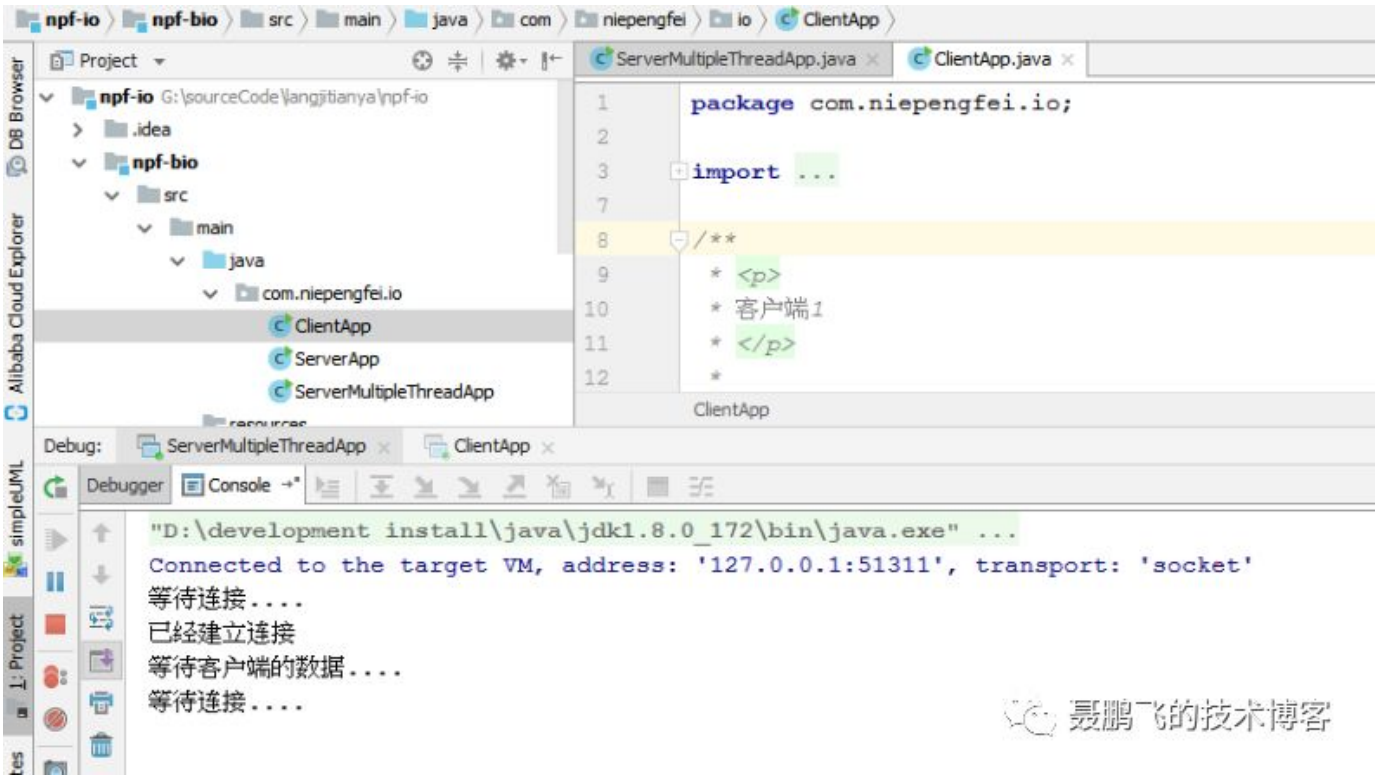


客户端保持不变，因为能否解决并发问题，主要是在服务端。我们进行如下测试。

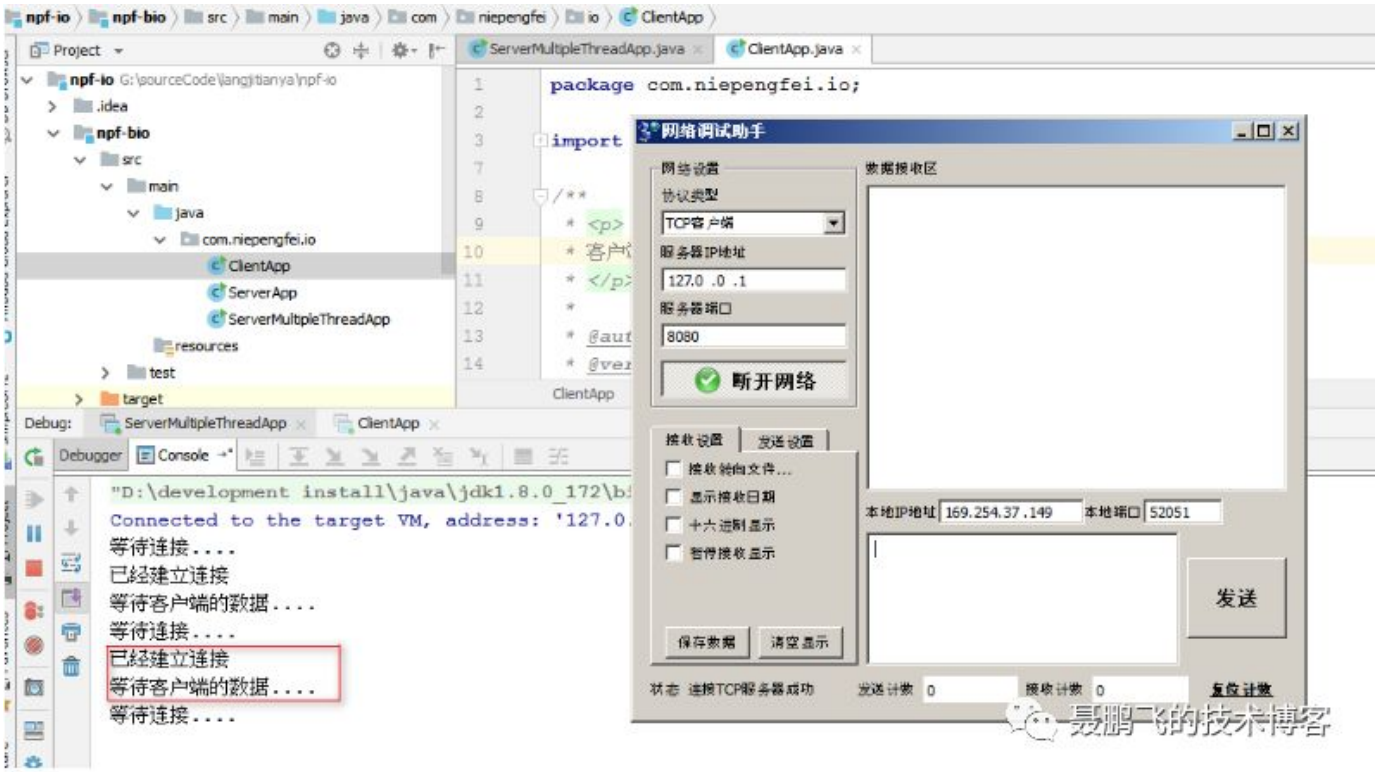
第一步：启动服务端。



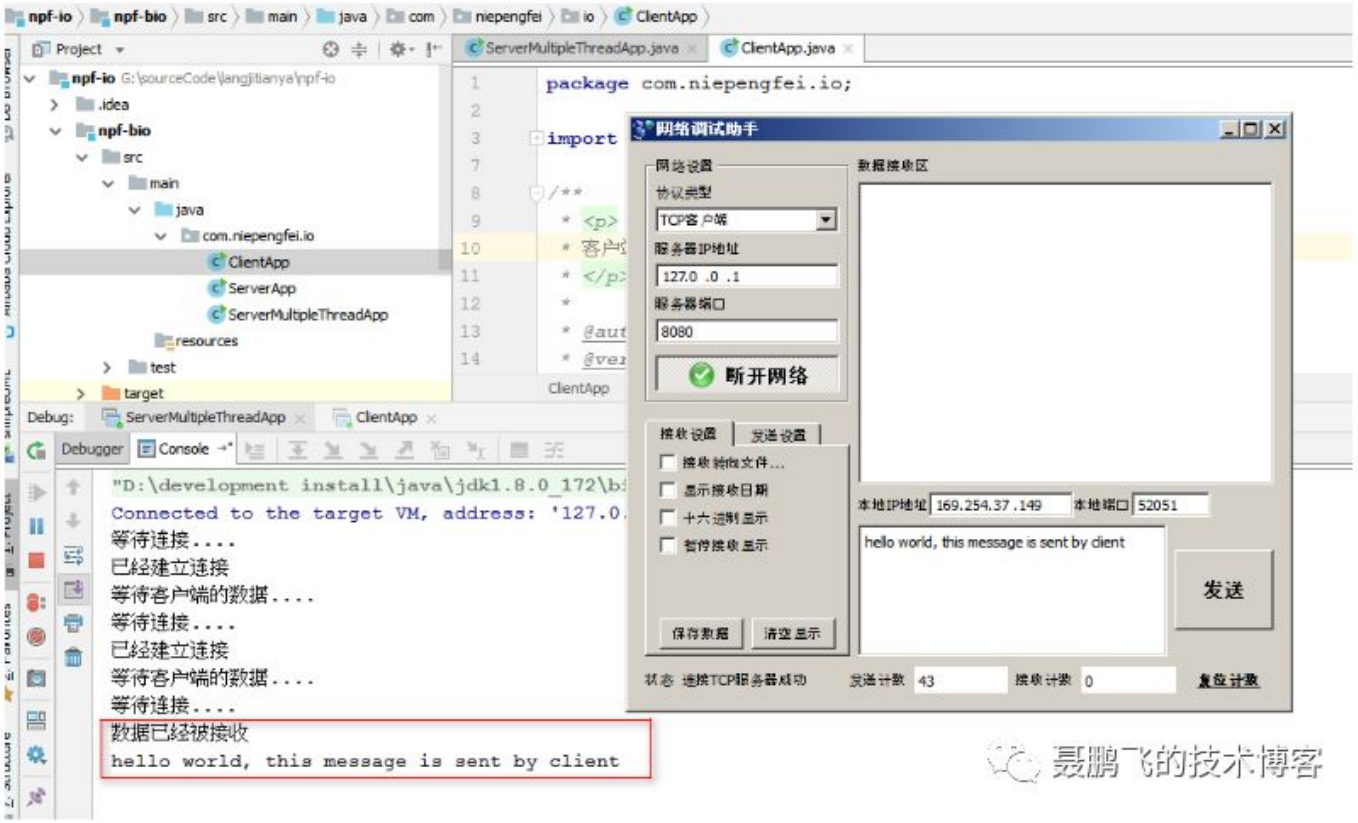
第二步：启动第一个客户端。让这个客户端和服务端建立连接，但是不发送数据。



第三步：让另一个客户端和服务端建立连接。



很明显，这个客户端显然连接上了服务端。此时我们发送数据看看。



从这里可以看出，BIO在多线程情况下，确实是可以解决并发问题的。

1.4.2 存在的问题

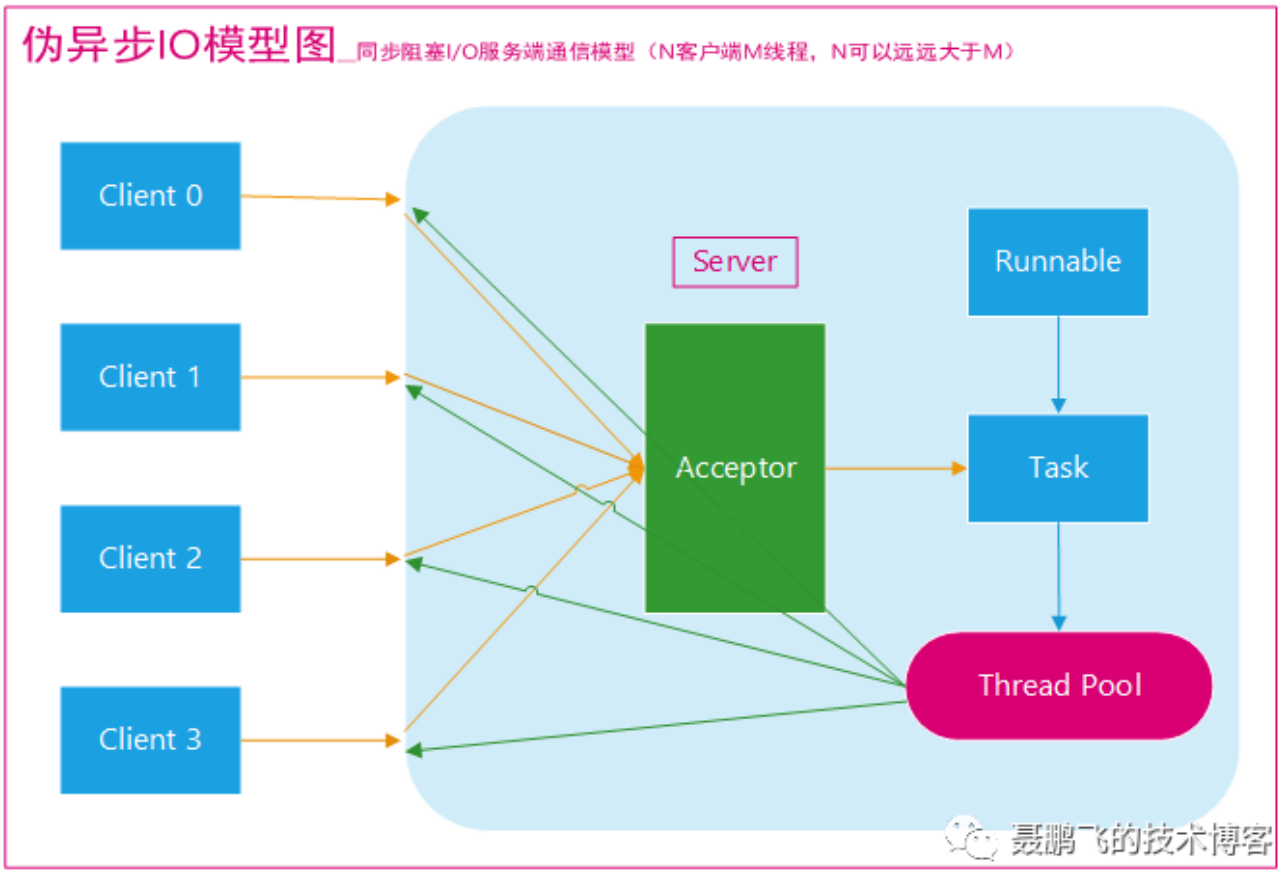
我们可以设想一下，如果连接到服务端的客户端不做任何事情的话就会造成不必要的线程开销，不过我们可以通过 **线程池机制** 改善，线程池还可以让线程的创建和回收成本相对较低。使用`FixedThreadPool`

可以有效的控制了线程的最大数量，保证了系统有限的资源的控制，实现了N(客户端请求数量):M(处理客户端请求的线程数量)的I/O模型（N 可以远远大于 M），即BIO在多线程情况下可以解决并发问题。

我们再设想一下当客户端并发访问量增加后这种模型会出现什么问题?

在 Java 虚拟机中，线程是宝贵的资源，线程的创建和销毁成本很高，除此之外，线程的切换成本也是很高的。尤其在 Linux 这样的操作系统中，线程本质上就是一个进程，创建和销毁线程都是重量级的系统函数。如果并发访问量增加会导致线程数急剧膨胀可能会导致线程堆栈溢出、创建新线程失败等问题，最终导致进程宕机或者僵死，不能对外提供服务。

为了解决BIO面临的一个客户端请求需要一个线程处理的问题，后来有人对它的线程模型进行了优化——后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数N：线程池最大线程数M的比例关系，其中N可以远远大于M。通过线程池可以灵活地调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。



采用线程池和任务队列可以让BIO解决并发问题，它的模型图如上图所示。当有新的客户端接入时，将客户端的 Socket 封装成一个Task（该任务实现java.lang.Runnable接口）投递到后端的线程池中进行处理，JDK 的线程池维护一个消息队列和 N 个活跃线程，对消息队列中的任务进行处理。

由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。该方案采用了线程池实现，因此避免了为每个请求都创建一个独立线程造成的线程资源耗尽问题。不过因为它的底层仍然是同步阻塞的BIO模型，因此无法从根本上解决问题。在活动连接数不是特别高的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲

一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

2. 理解NIO

2.1 NIO的设计初衷

基于BIO的缺陷，NIO被设计出来，就是为了在单线程情况下，可以解决并发问题。基于前面的分析，BIO之所以在单线程情况下不能解决并发问题的实质是accept和read方法都是阻塞的。那么很简单，NIO就是让这两个方法不阻塞不就可以了嘛。

2.2 代码层面实现

可以看到如下的代码，详细的注释已在代码中说明。例如：

第一步：当client1来连接的时候，accept方法获取连接立即返回，假设此时client1没有发数据过来的话，此时read方法也不会阻塞的。程序继续执行，再次进入while循环。

第二步：当再次进入while循环的时候，此时若还没有其他的客户端来连，那么此时accept方法立即返回null。

第三步：程序继续执行，若此时client1发来了数据，然后我们会发现，服务端已经接收不到数据了，因为我们已经丢失了client1的那个socket连接。那么怎么办呢?

```
/**
 * @author Jack
 * @version 1.0.0
 * @since 1/11/2020
 */
public class ServerNioApp {

    private static ByteBuffer buffer = ByteBuffer.allocate(1024);
    public static void main(String[] args) throws Exception{
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.bind(new InetSocketAddress( port: 8080));
        //设置服务端等待连接 非阻塞，意思是在调用accept方法时，不会再阻塞
        //如果有客户端来连，那么服务端就会创建一个socket与当前连接上来的客户端进行通信
        //如果没有客户端来连，那么该方法立即返回,且返回值是null
        serverSocketChannel.configureBlocking(false);
        while (true){
            SocketChannel clientSocket = serverSocketChannel.accept();①
            //设置读取客户端发送过来的数据的动作是非阻塞的
            //如果客户端有发送数据过来,那么自然会读取到
            //如果客户端没有发送数据过来,那么read方法立即返回,不会阻塞
            clientSocket.configureBlocking(false);
            int read = clientSocket.read(buffer);②
            if (read > 0) {
                System.out.println(new String(buffer.array()));
            }
        }
    }
```

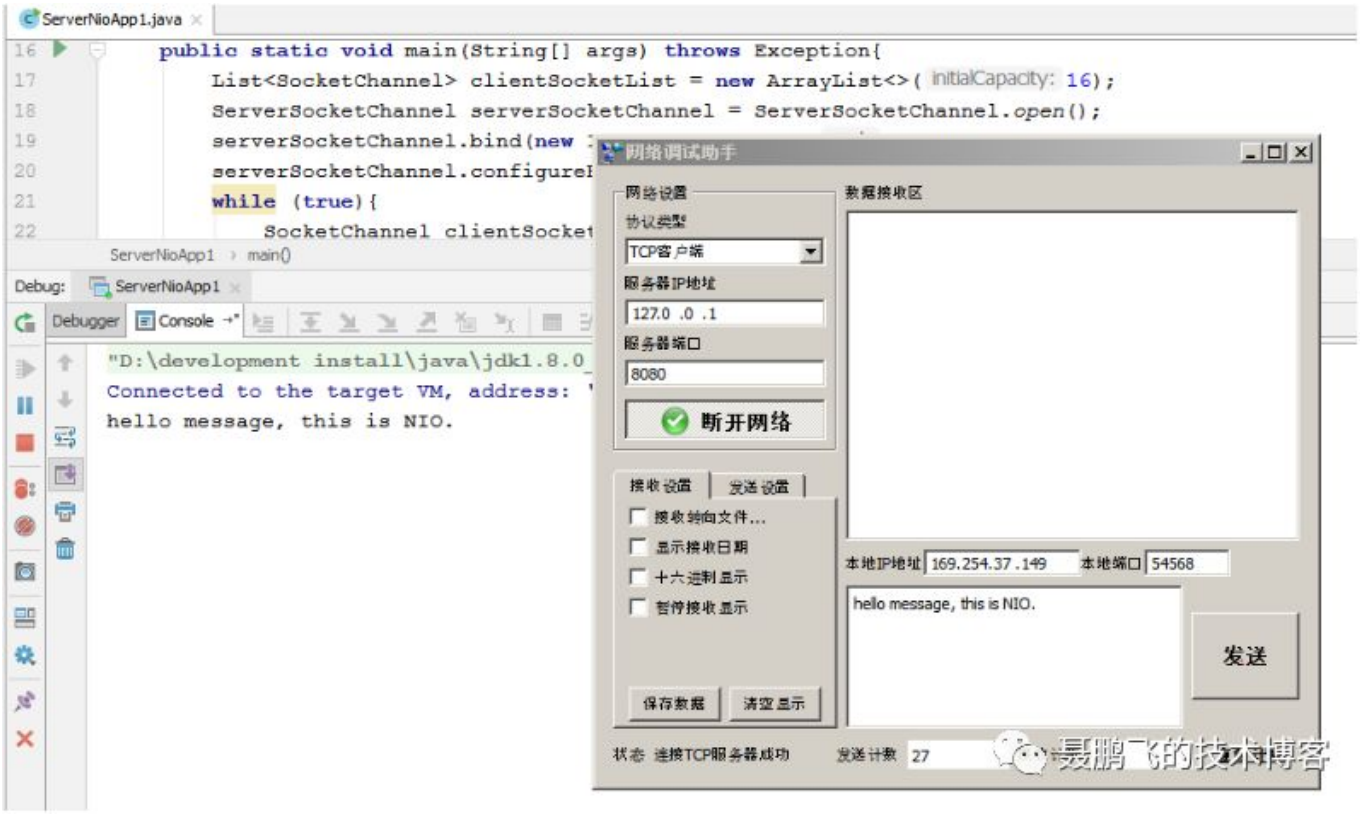
针对上面的问题，我们如何改进呢？那么我们是不是要有一个集合来存这个已经连上来的客户端，是的，我们的分析是没有错的，那么请看下面的代码。

ServerNioApp1.java

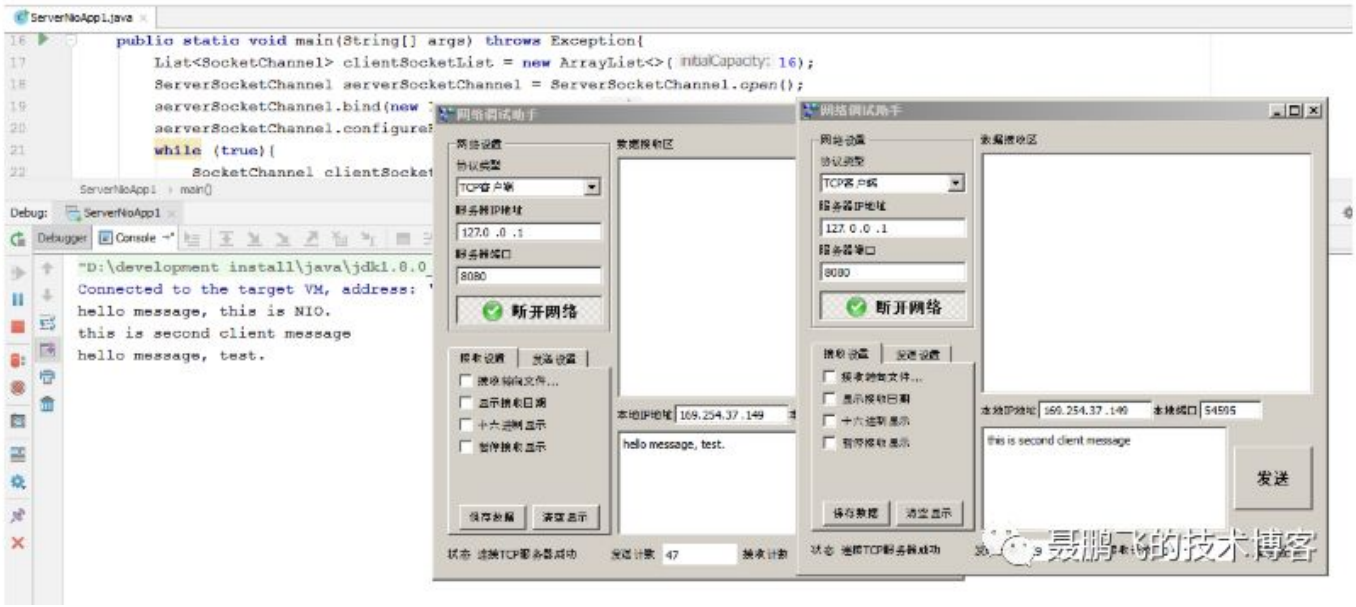
```
16      public static void main(String[] args) throws Exception{
17          List<SocketChannel> clientSocketList = new ArrayList<>( initialCapacity: 16);
18          ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
19          serverSocketChannel.bind(new InetSocketAddress( port: 8080));
20          serverSocketChannel.configureBlocking(false);
21          while (true){
22              SocketChannel clientSocket = serverSocketChannel.accept();
23              if (clientSocket == null) {
24                  List<SocketChannel> clientList = new ArrayList<>(clientSocketList);
25                  for (SocketChannel socketChannel : clientList ) {
26                      ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
27                      int read = socketChannel.read(byteBuffer);
28                      byteBuffer.flip();
29                      if (read > 0) {System.out.println(new String(byteBuffer.array()));}
30                      else if (read < 0){ clientSocketList.remove(socketChannel); }
31                  }
32              } else {
33                  clientSocket.configureBlocking(false);
34                  clientSocketList.add(clientSocket);
35                  List<SocketChannel> clientList = new ArrayList<>(clientSocketList);
36                  for (SocketChannel socketChannel : clientList ) {
37                      ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
38                      int read = socketChannel.read(byteBuffer);
39                      byteBuffer.flip();
40                      if (read > 0) { System.out.println(new String(byteBuffer.array())); }
41                      else if (read < 0){ clientSocketList.remove(socketChannel); }
42                  }
```


我们有一个list集合来存储连接到服务端的客户端，如果有人连接上来，那么就放入到集合当中，随后遍历该集合，判断是否有客户端发送数据过来。如果没有人连接上来，那么也会遍历集合，目的是判断之前连接上来的客户端是否有发送数据过来。

我们启动服务端，并在客户端进行测试，如下：



接着，我们再开另外一个客户端，发现也是可以连接上来的，并且第二个客户端发数据到服务端，服务端是可以接收到的，随后第一个客户端再发了一条数据，服务端同样是可以接收到的。



2.3 问题的所在

2.3.1 natvie方法在哪里

根据上面的分析，应用程序需要不断的去循环那个list集合。我们不应该将这个集合放在应用程序中去执行，应该放在操作系统的内核中去执行。那该怎么去解决呢？通过调用操作系统底层的select函数，就是将这个集合交给操作系统去循环。select是操作系统函数，可以实现该功能。那么我问你，Java是怎么调

用操作系统函数呢？很简单，Java可以通过JNI方式调用native方法，而在native方法中去手动的调用操作系统的函数。不信你可以看我的分析。当你写这行代码的时候。

```
ServerSocket serverSocket = new ServerSocket(8080);
```

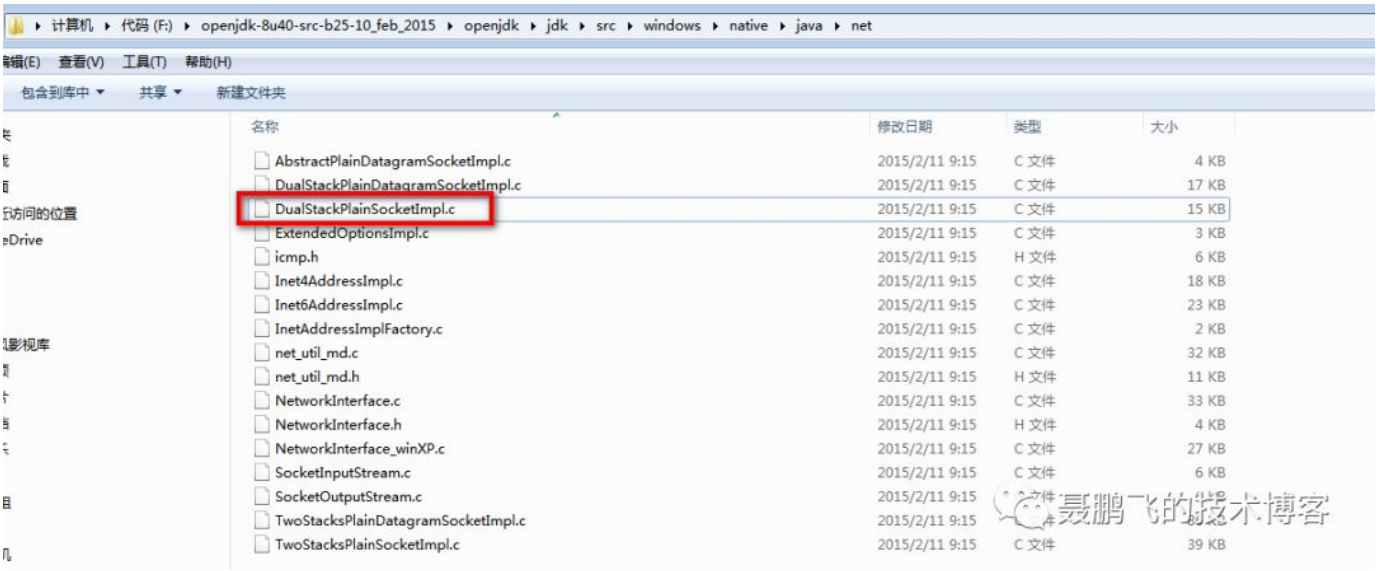

实际底层会调用如下的代码：

DualStackPlainSocketImpl.java

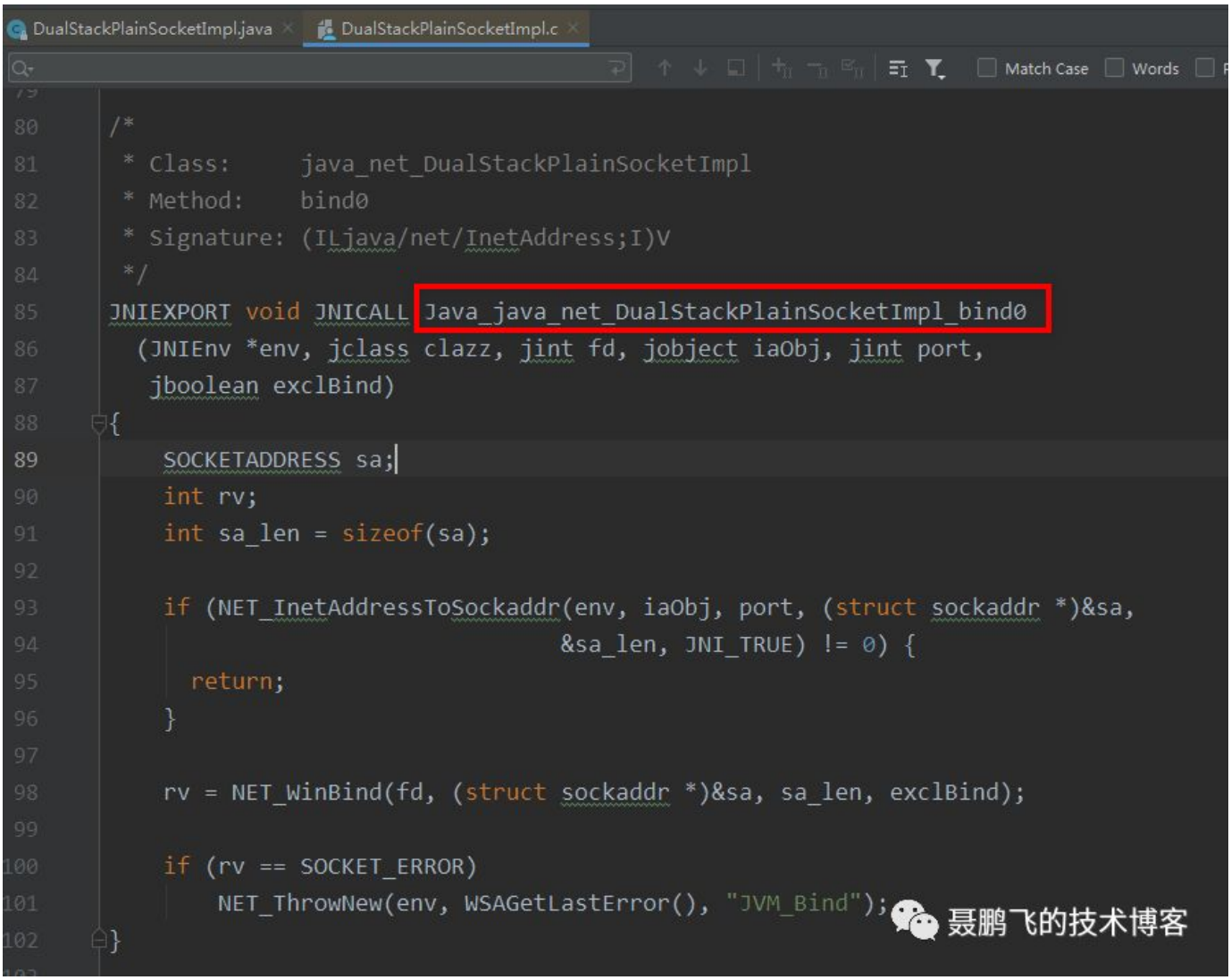
```
98      }
99
100 @   void socketBind(InetAddress address, int port) throws IOException {
101         int nativefd = checkAndReturnNativeFD();
102
103         if (address == null)
104             throw new NullPointerException("inet address argument is null.");
105
106         bind0(nativefd, address, port, exclusiveBind);
107         if (port == 0) {
108             localport = localPort0(nativefd);
109         } else {
110             localport = port;
111         }
112
113         this.address = address;
114     }
```


即bind0方法，意为本地方法，即c语言实现的方法，那这个方法到底存在哪里呢？即JVM当中，那什么是JVM呢，说白了，就是你装好的那个java.exe程序，那么这个java.exe就是一个编译好的程序，那这个java.exe所属的项目到底是哪个呢？那就是openjdk，源码的下载地址如下http://hg.openjdk.java.net/ ， 现在的问题是，我们只要下载好这个openjdk的源码，然后在源码里面去找这个bind0方法就好了。然后去看看bind0方法内部是如何实现的。

windows平台中的实现

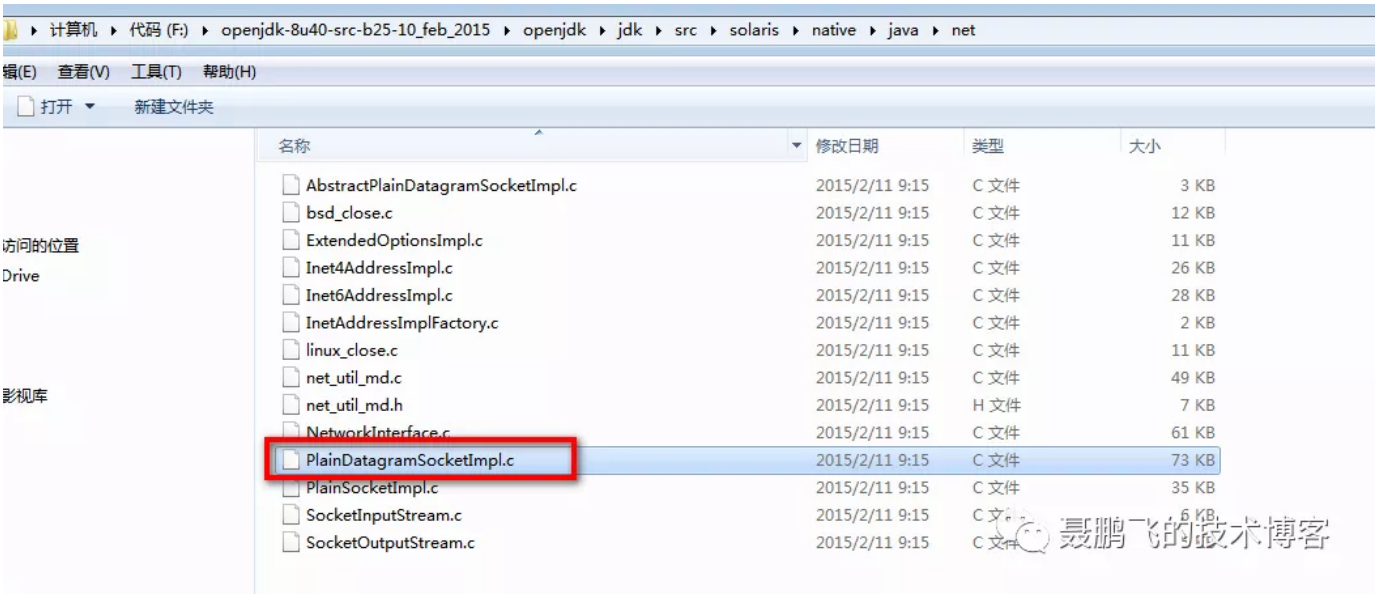


我们打开这个文件，具体查看bind0到底是如何实现的。

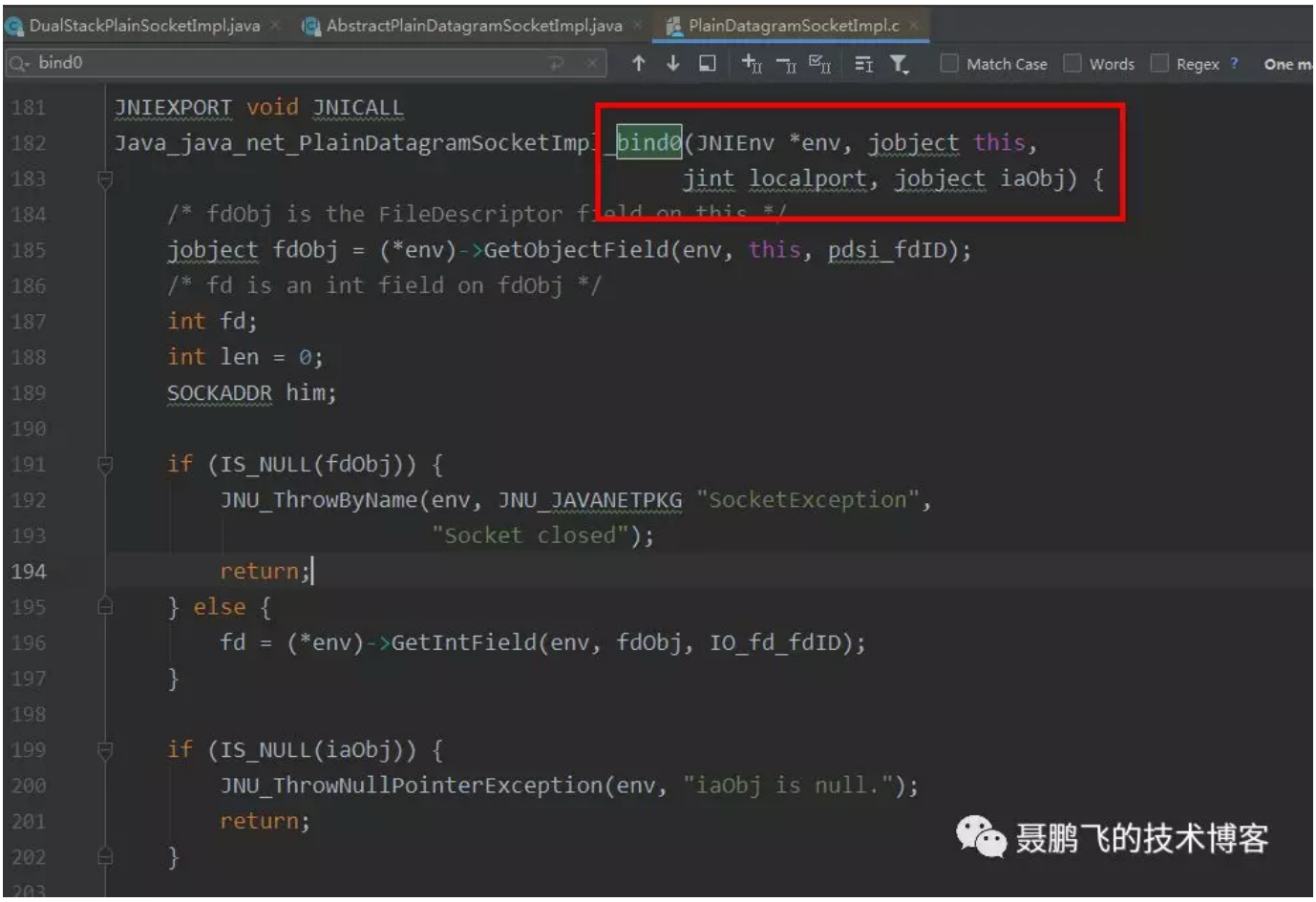


由于我们查看的是windows的bind0的实现，所以在源码中，你可以很清楚的看到调用了windows操作系统的函数。

linux平台中的实现



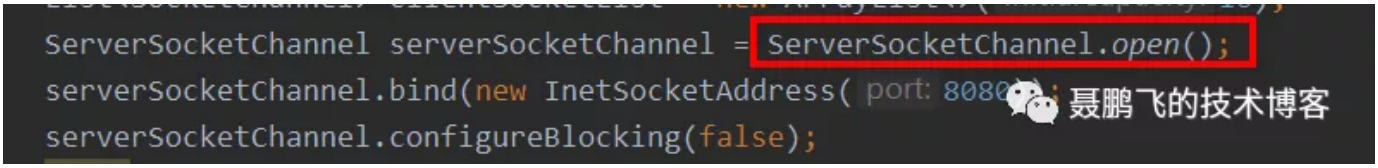
我们打开这个文件，具体查看bind0到底是如何实现的。



2.3.2 重点分析

windows平台的分析

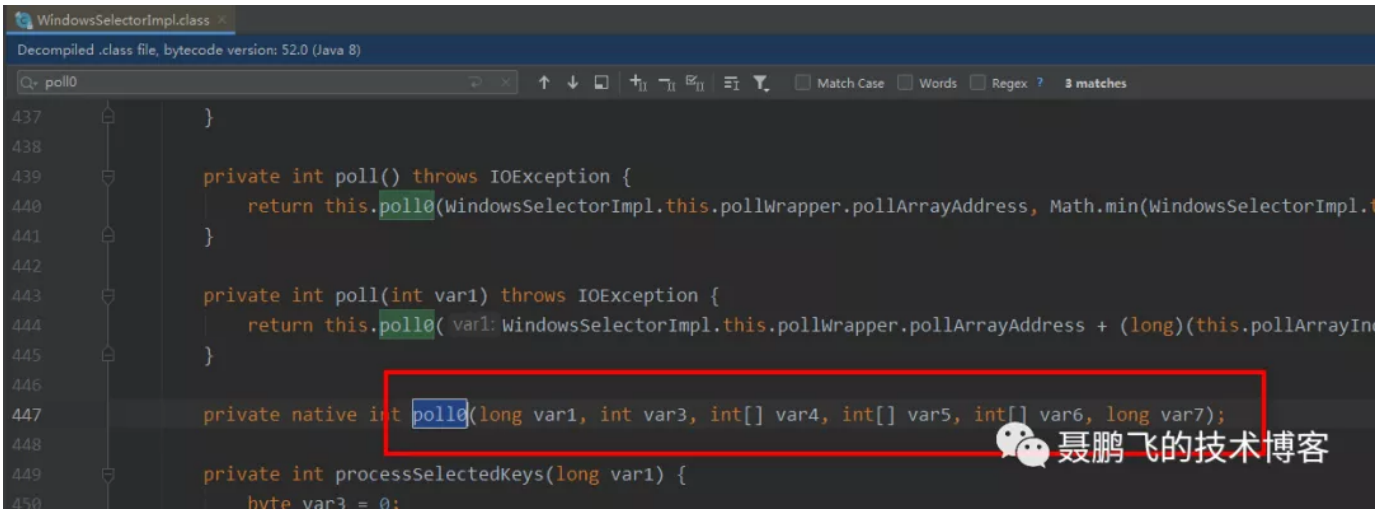
当你调用这个方法的时候，底层是如何实现的呢？



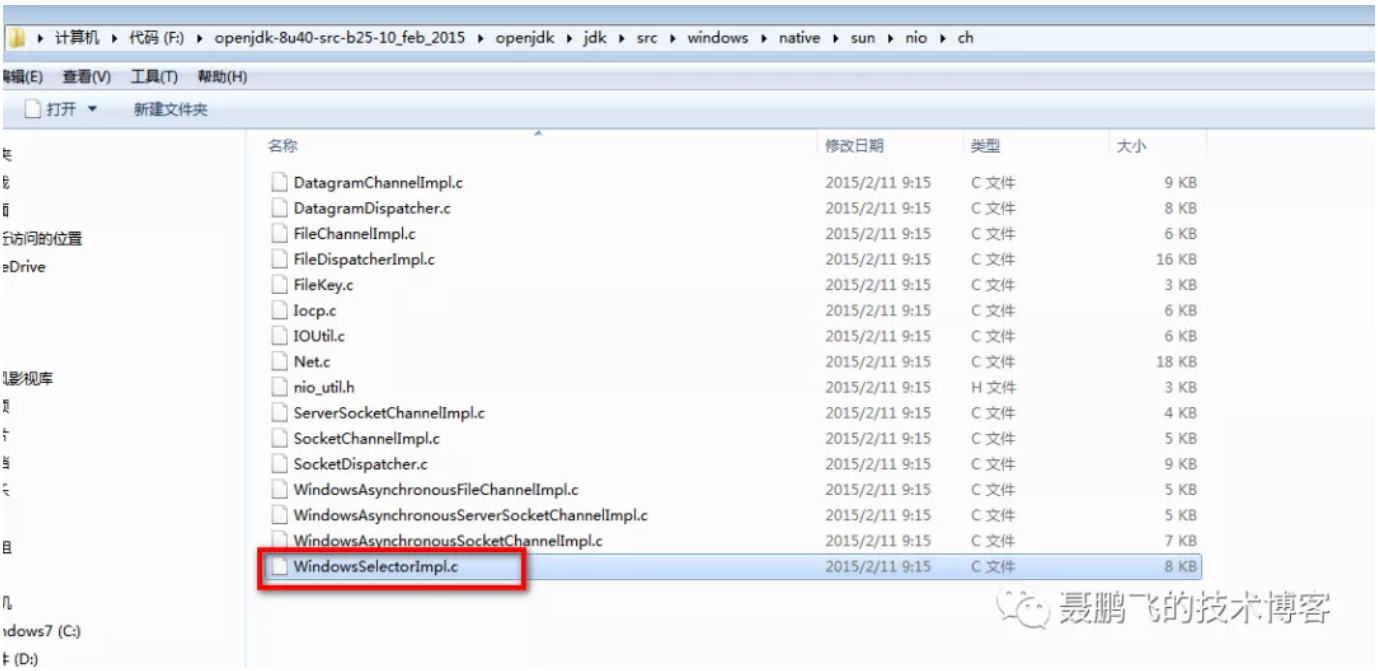
追踪到源码，底层调用的代码如下：

```
package sun.nio.ch;

import java.nio.channels.spi.SelectorProvider;

public class DefaultSelectorProvider {
    private DefaultSelectorProvider() {
    }

    public static SelectorProvider create() {
        return new WindowsSelectorProvider();
    }
}
```

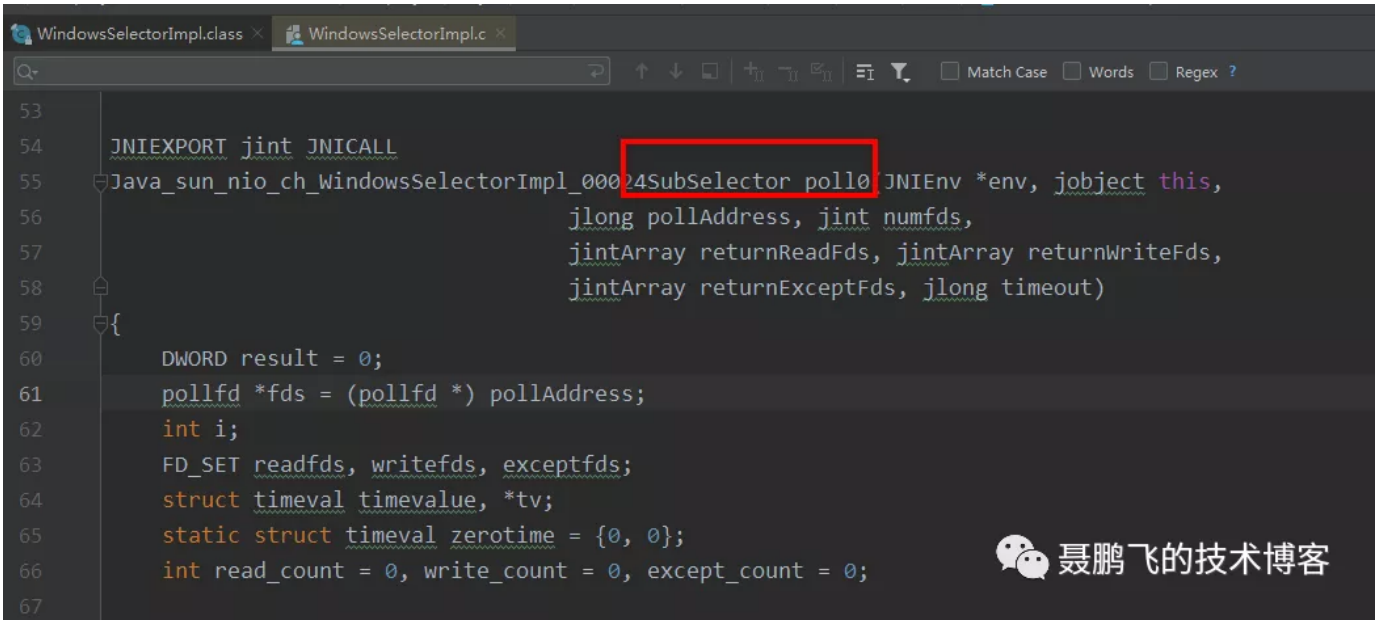

可以发现，在windows平台下，直接就new出来了一个WindowsSelectorProvider对象。分析到这里，你肯定也能猜到，在windows平台下，肯定是这个对象里面调用了native方法。



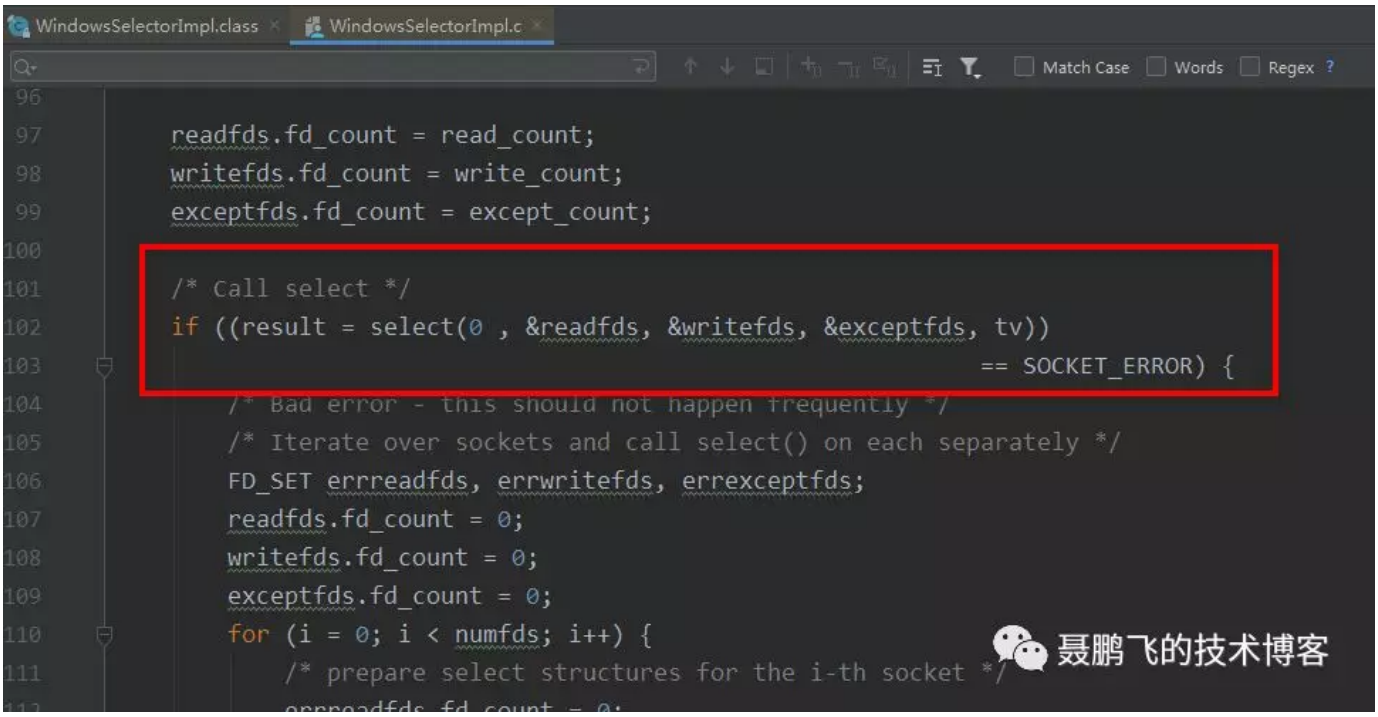
即在这个poll0方法中，调用了windows平台下的select函数。说白了，在windows底层，就是调用了这个方法，将上述我们一直讨论的那个list集合交给操作系统的。接下来我们具体查看一下该函数的实现。



我们打开这个文件，内容如下：

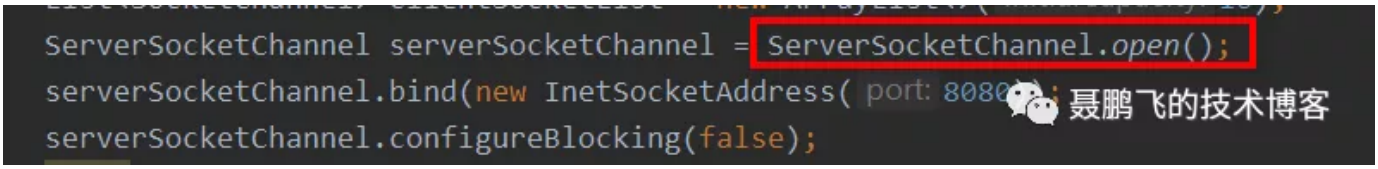


并且在poll0方法内部，显示的调用了select函数。如下所示：

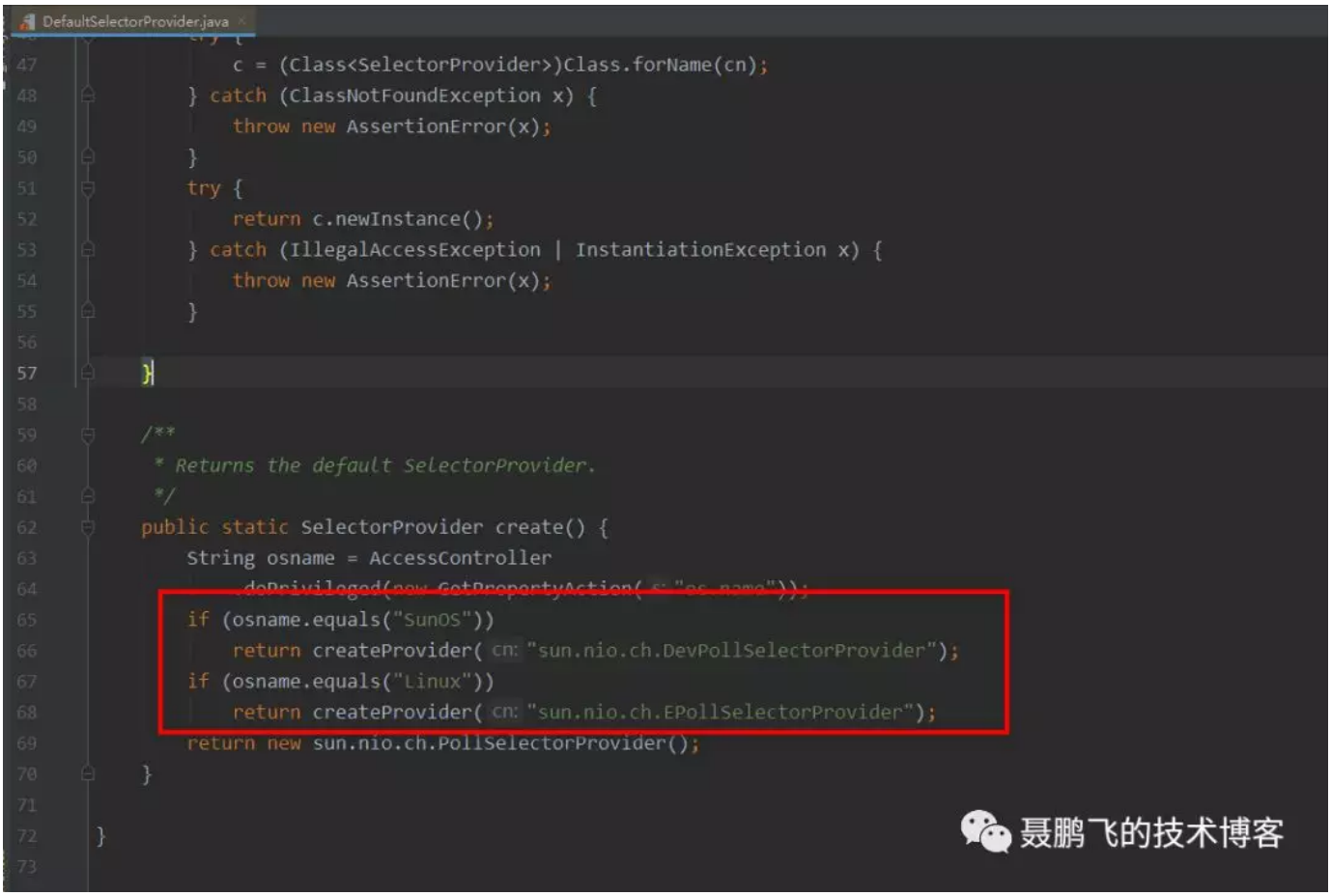


linux平台的分析

当你调用这个方法的时候，底层是如何实现的呢?



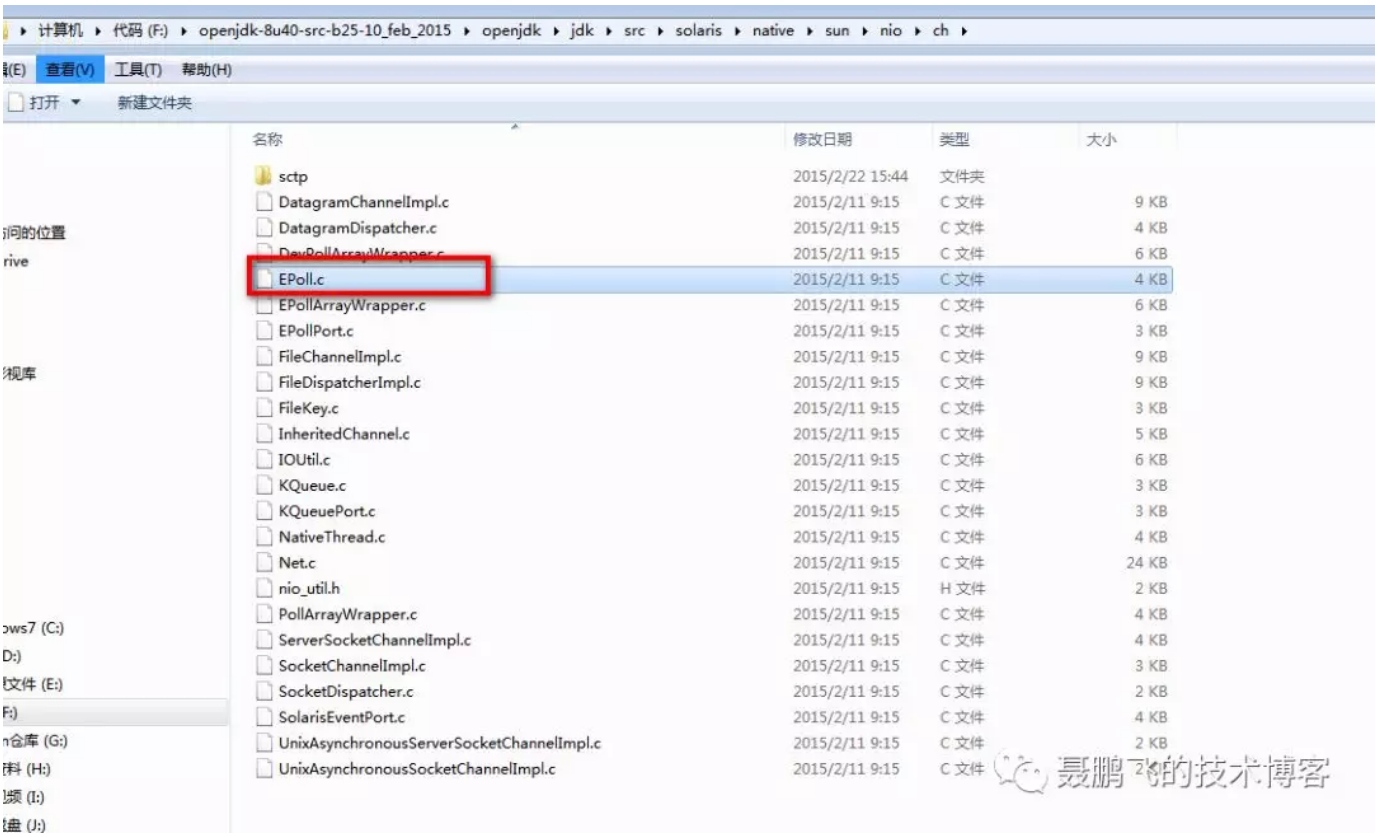
linux平台下追踪到源码，底层调用的代码如下：



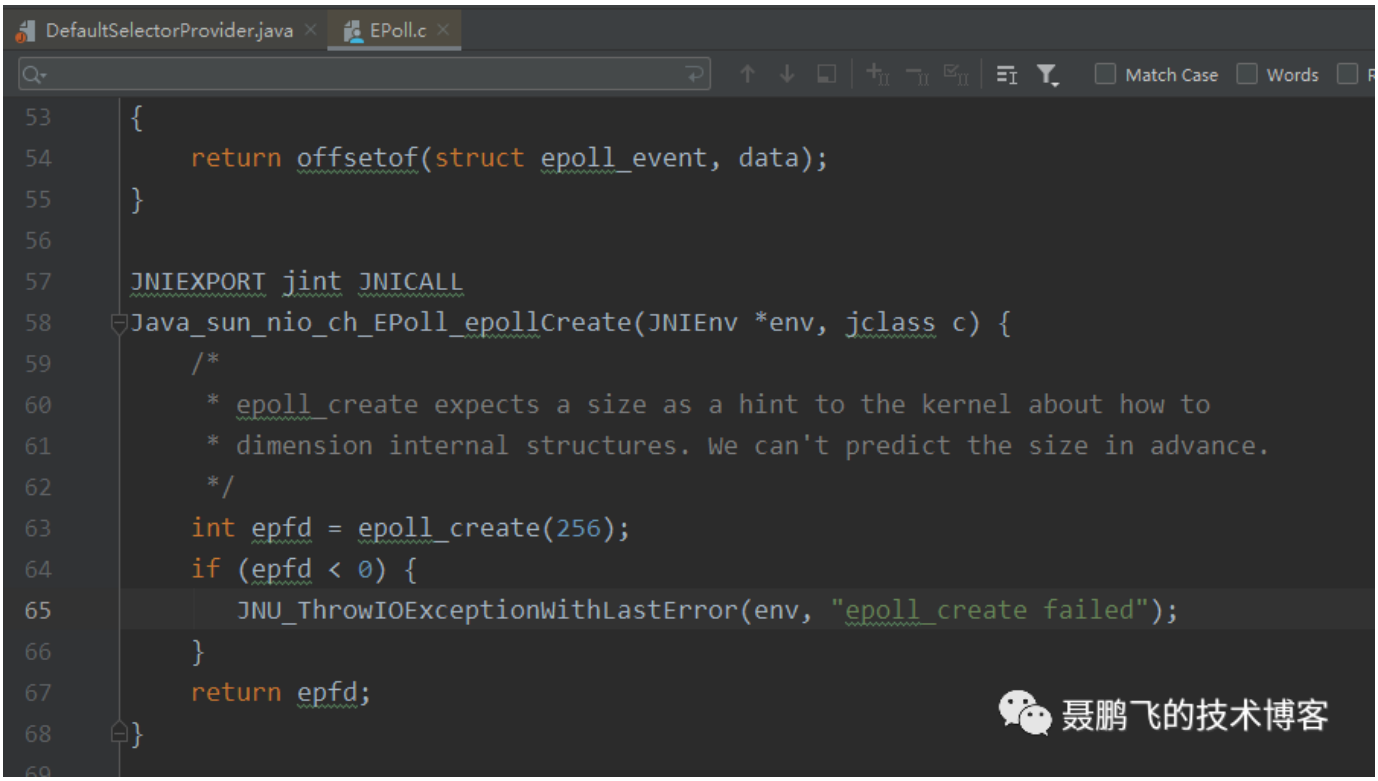
如果程序运行在Linux平台的话，就会采用

sun.nio.ch.EPollSelectorProvider

说明在高版本的JDK当中，linux平台中已经去除了select，采用了epoll。



我们打开这个文件，进行查看。



2.3.3 如何理解与查看系统函数

接下来我给你看一下操作系统的函数。整个linux系统，你可以简单的理解为是c语言写的，那么既然是c语言写的，那么linux系统肯定对外提供了c语言函数，即当前操作系统的函数。

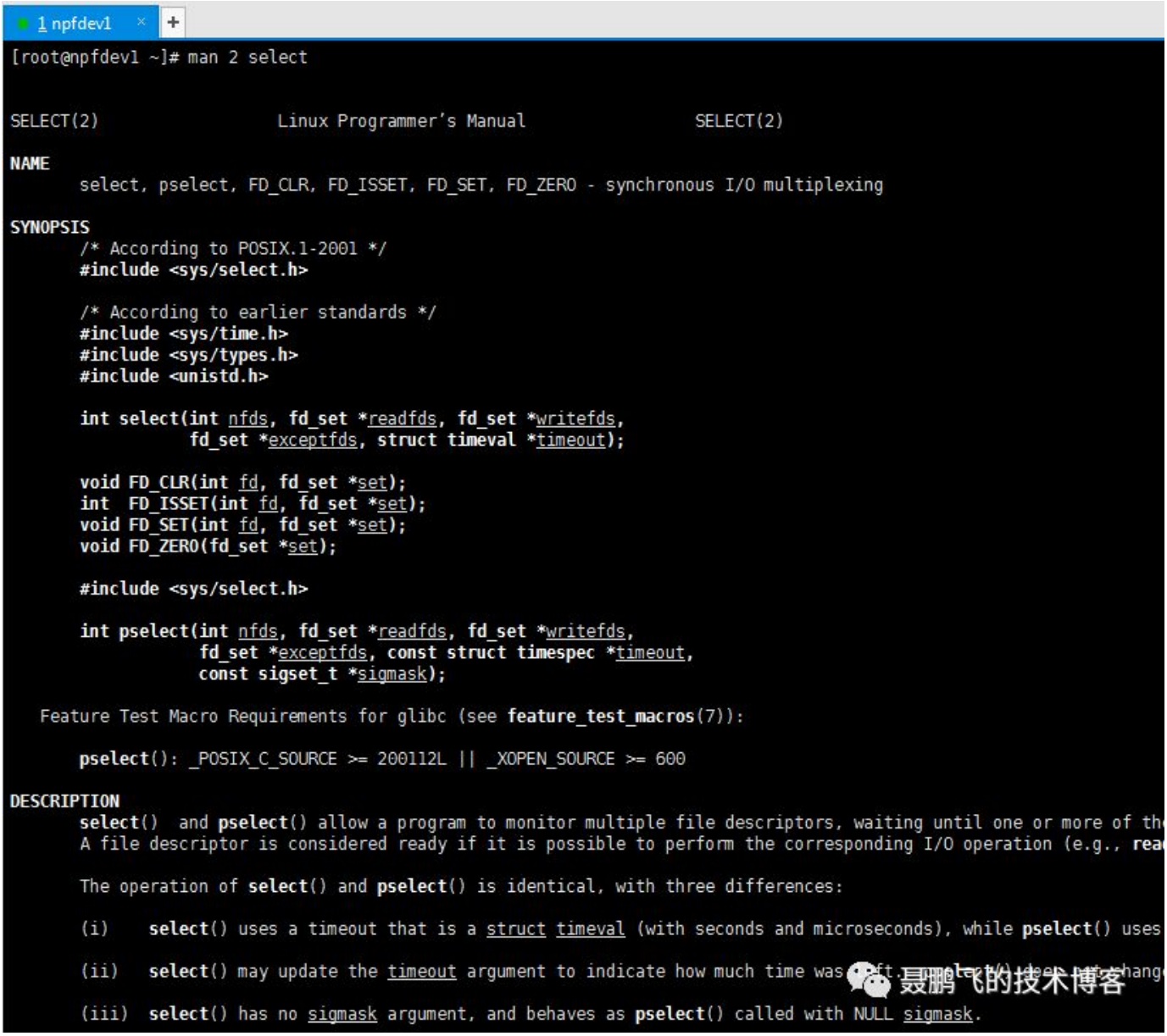
第一：查看socket函数。使用命令：man 2 socket

这个socket函数的意思，linux已经说的很清楚了。即：

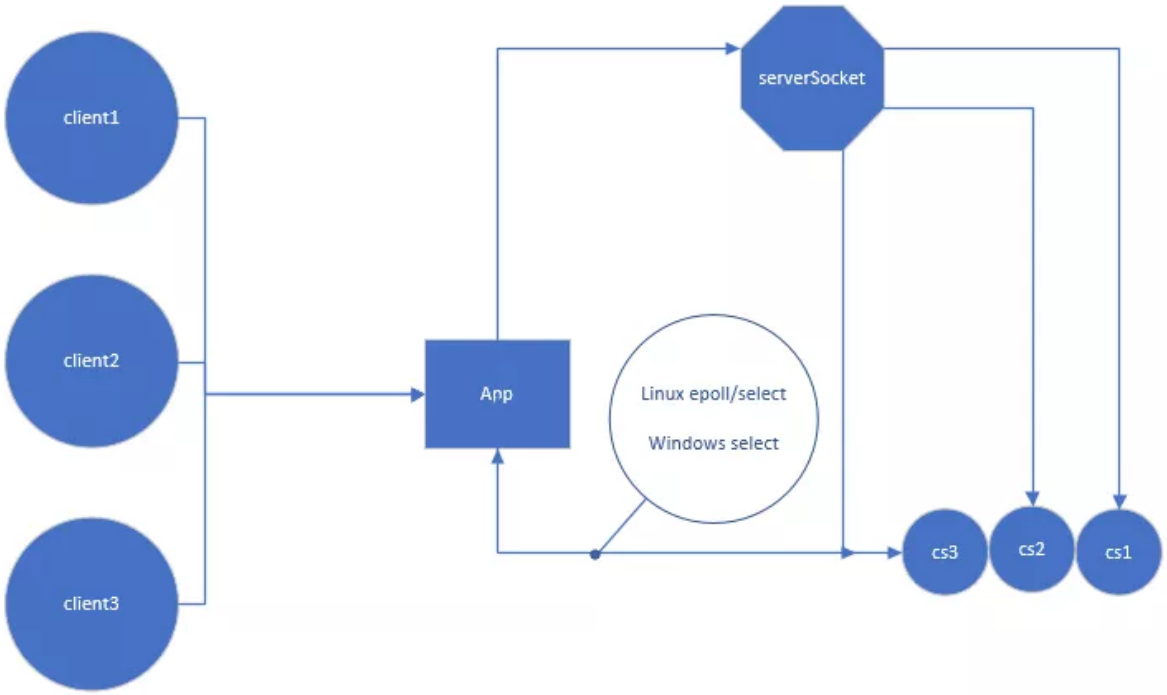
socket – create an endpoint for communication

创建一个终端以用于进行通信。

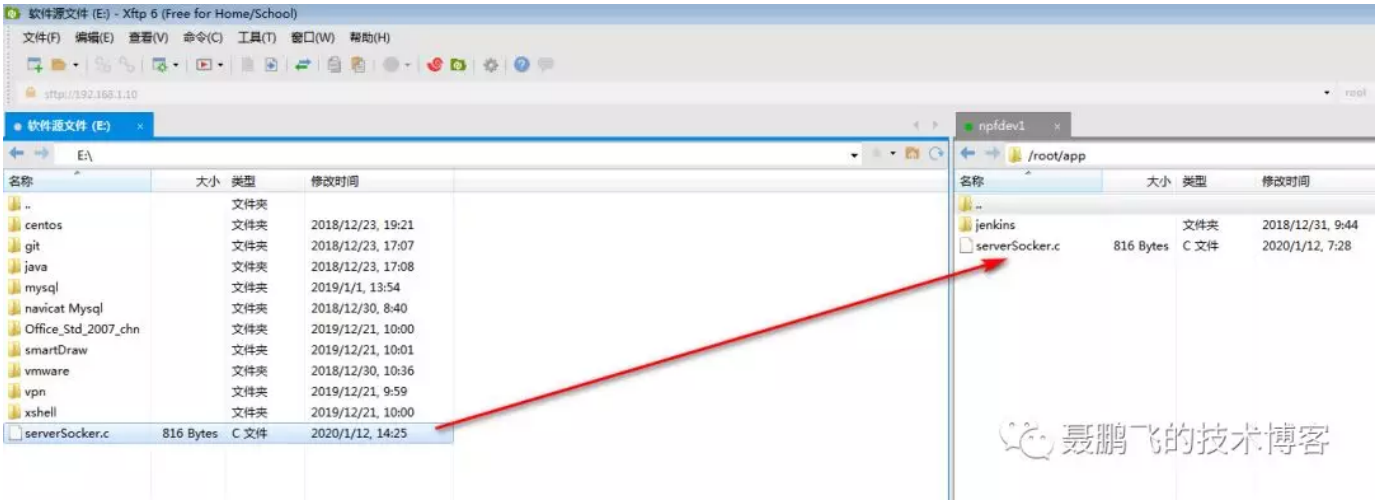
第二：查看select函数。使用命令： man 2 select



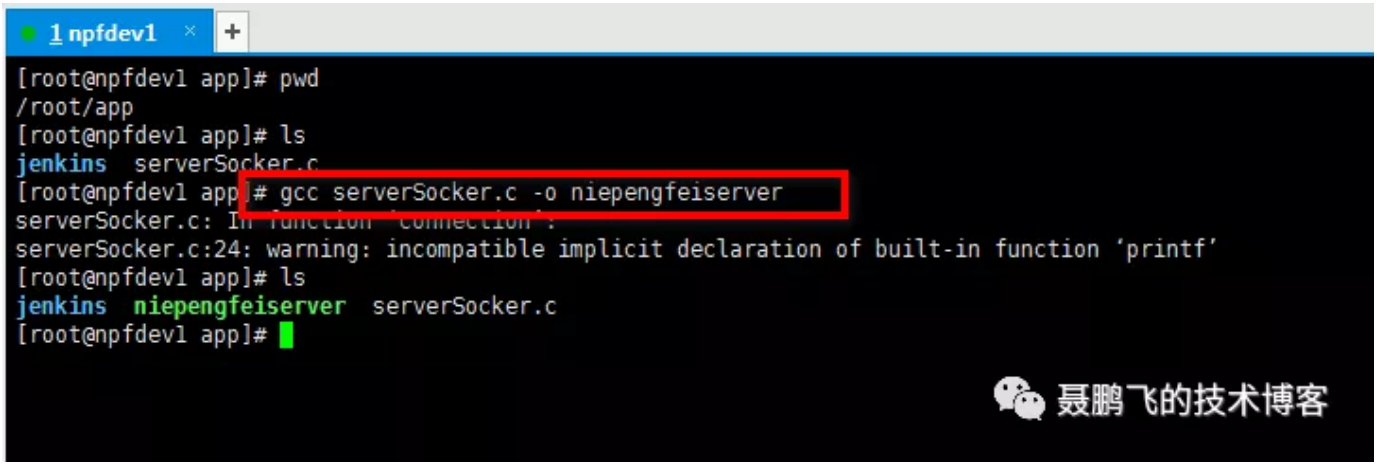
谈到这个select函数，我们不得不提一下NIO的思想，就是说sun公司的那帮人是怎么想到将list集合交给操作系统去循环的。



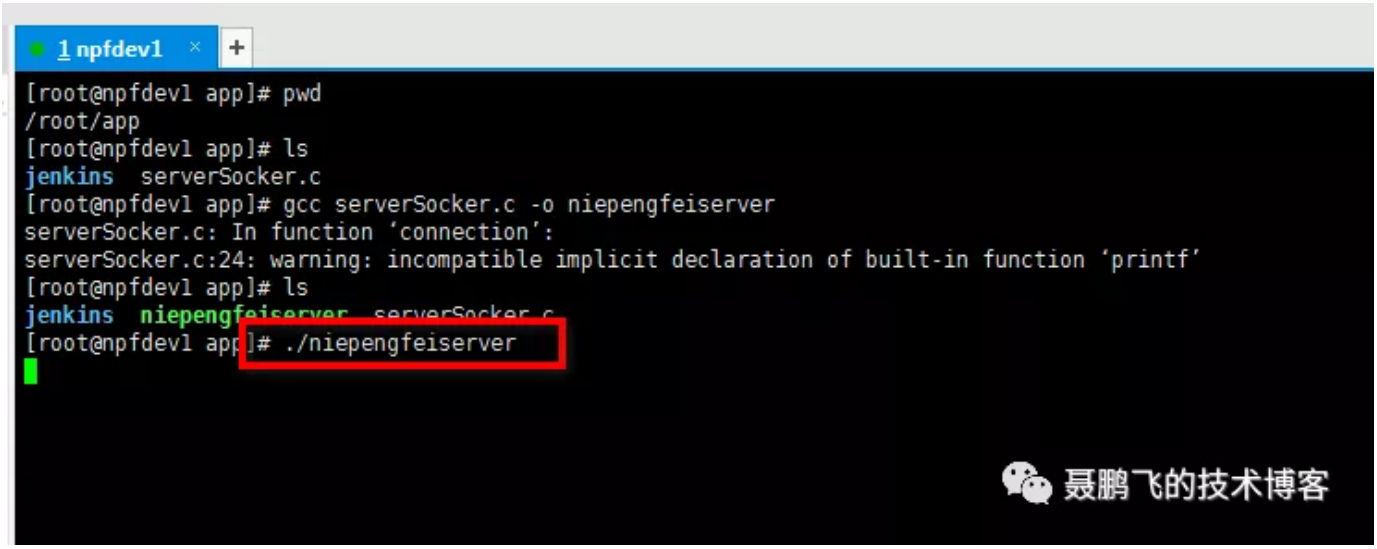
接下来我们来自己写一个c语言程序，去调用linux系统函数。那么请看我如何写的，耐心一点好吧，我接下来就写了。



运行如下命令进行编译：

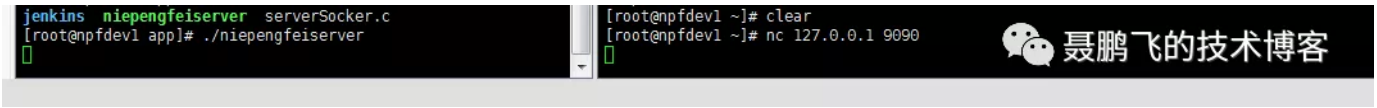


现在我们运行刚才编译好的程序，如下：

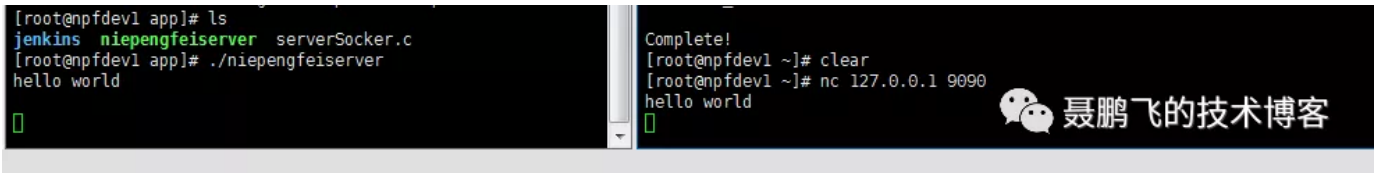


目前程序已经运行成功，正在等待连接。我们开一个客户端，运行命令：

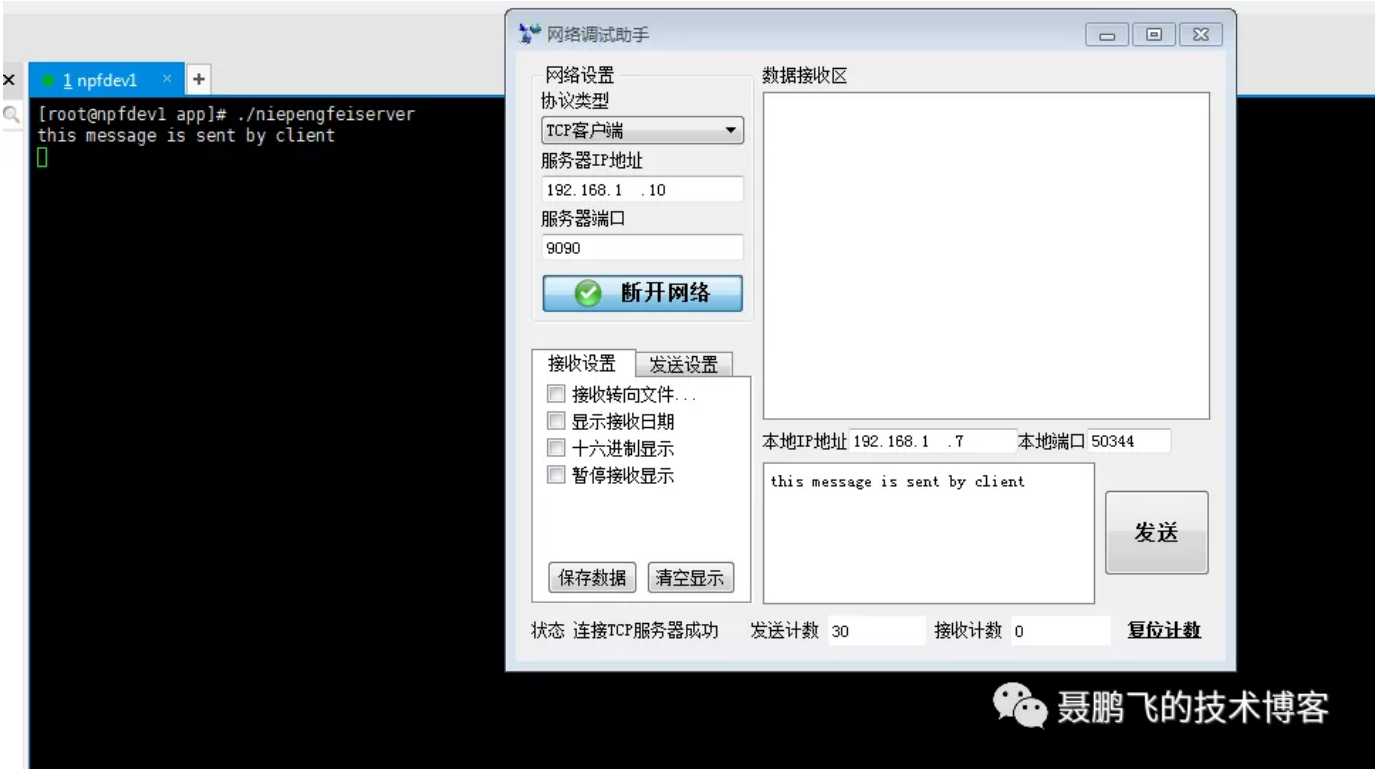
nc 127.0.0.1 9090



从上图可以看出，已经连接成功，之后我们在利用客户端向服务端发送一条数据，进行测试。



以下是通过客户端工具进行测试。



到此，我们在linux平台写出了一个服务端。