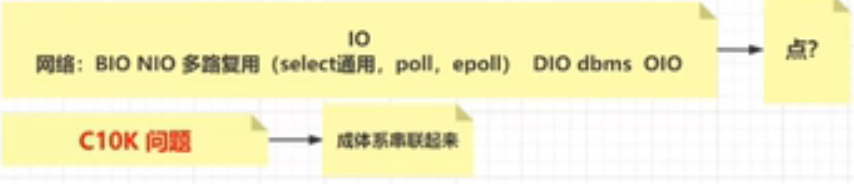
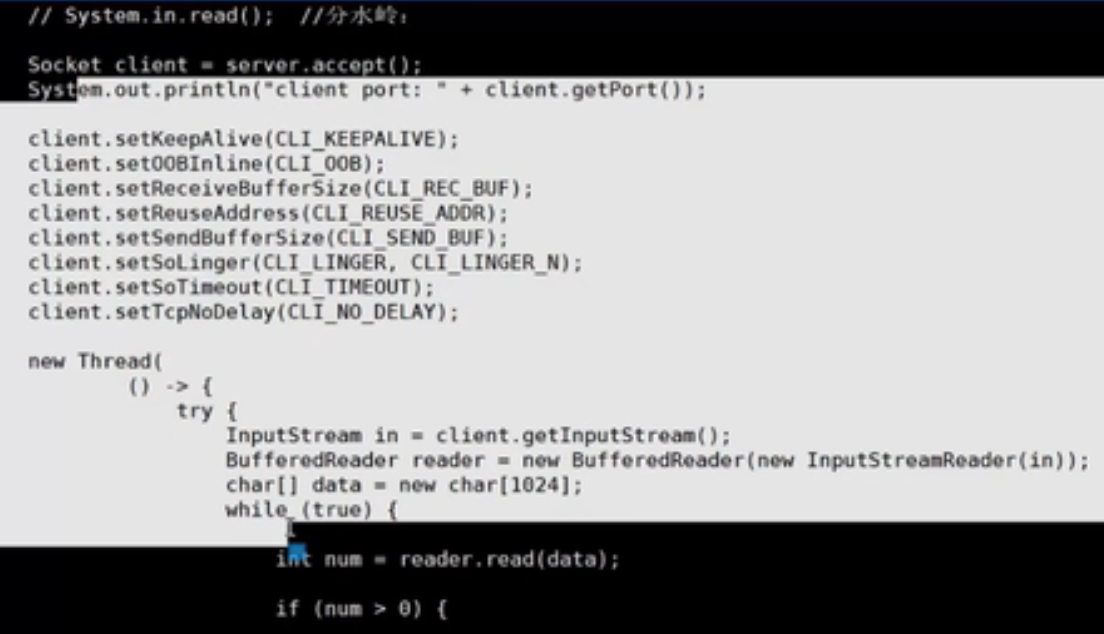
一、演示



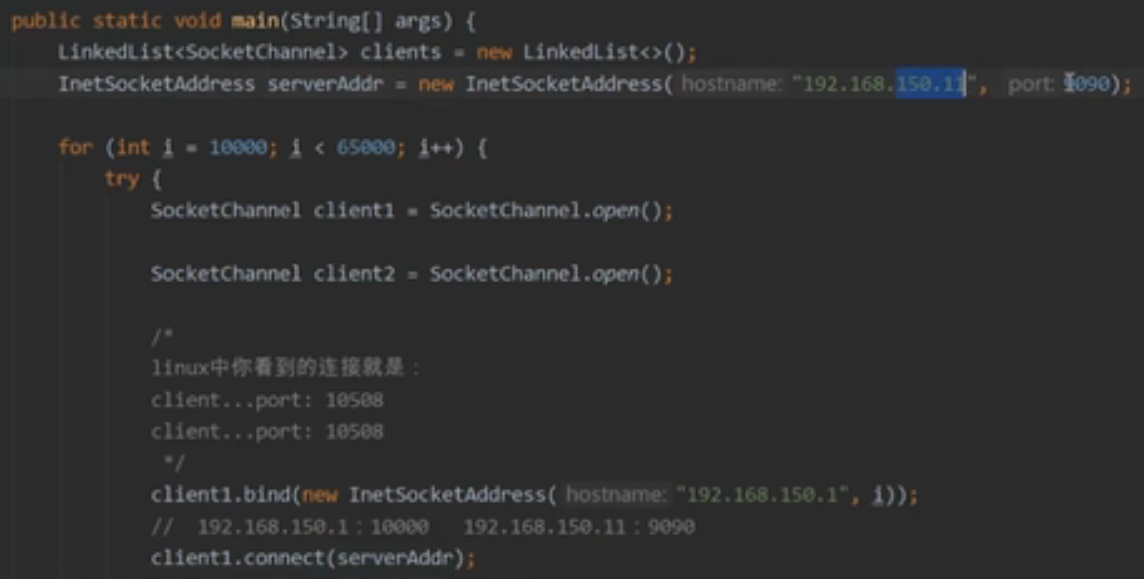
1. BIO 每连接，每线程

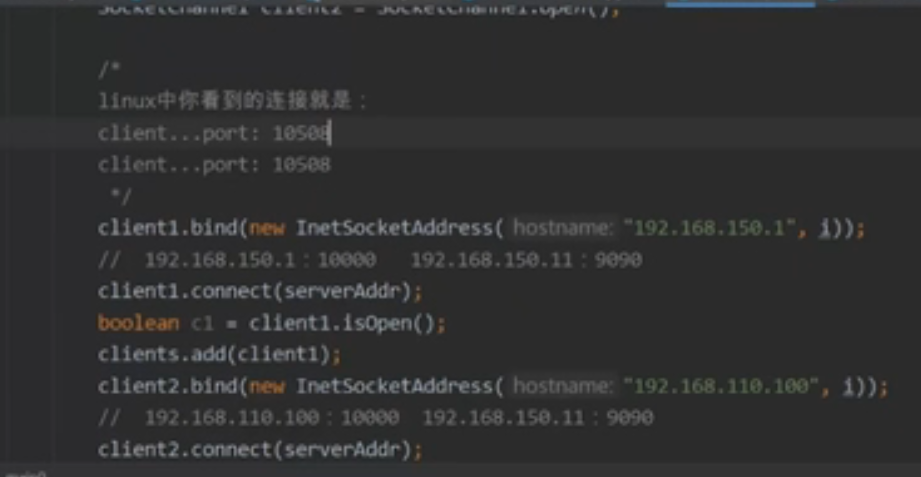
服务器：（阻塞，抛新线程）

阻塞原因：等待客户端发数据... => 所以需要抛线程



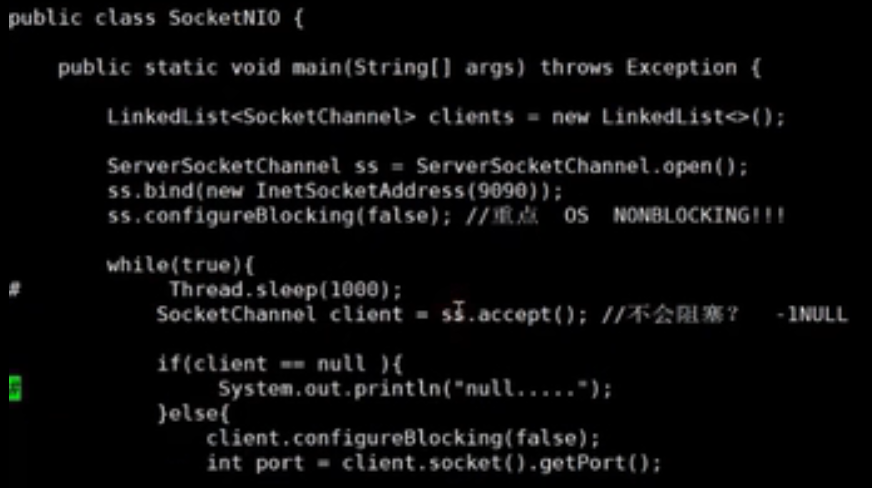
客户端：

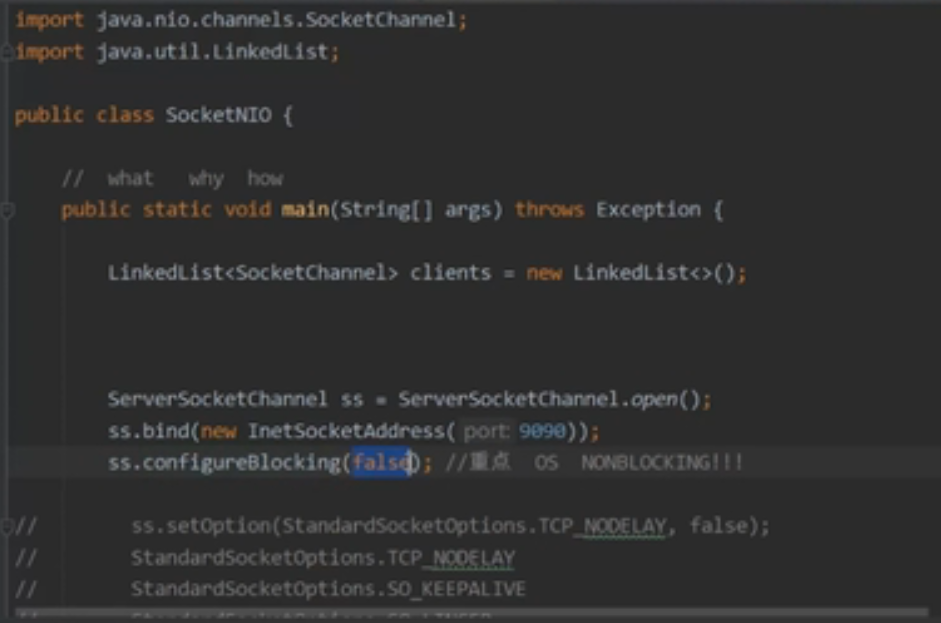




1. NIO

服务器（不阻塞，不抛新线程）

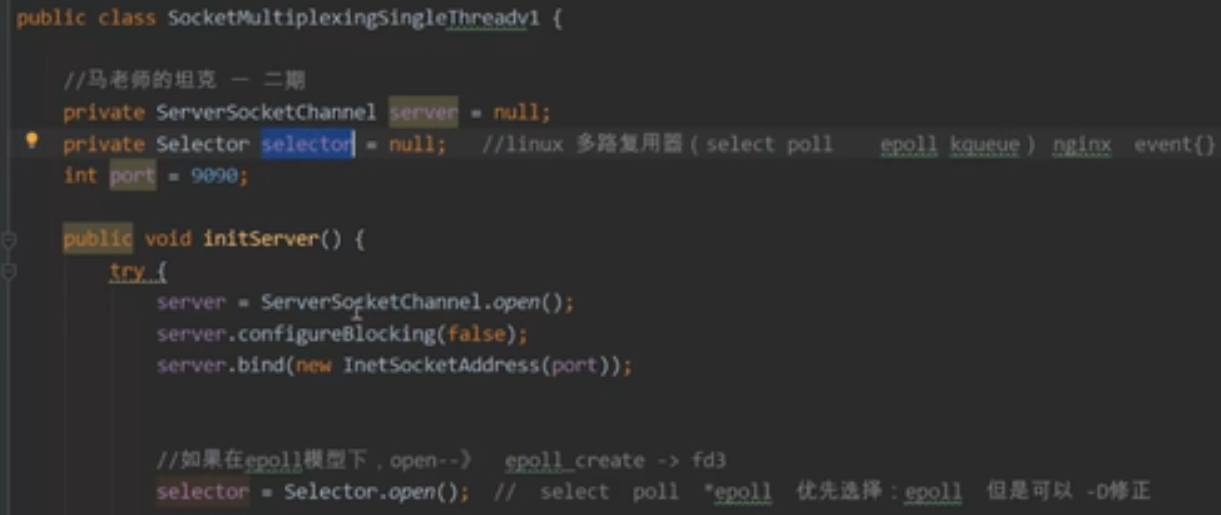


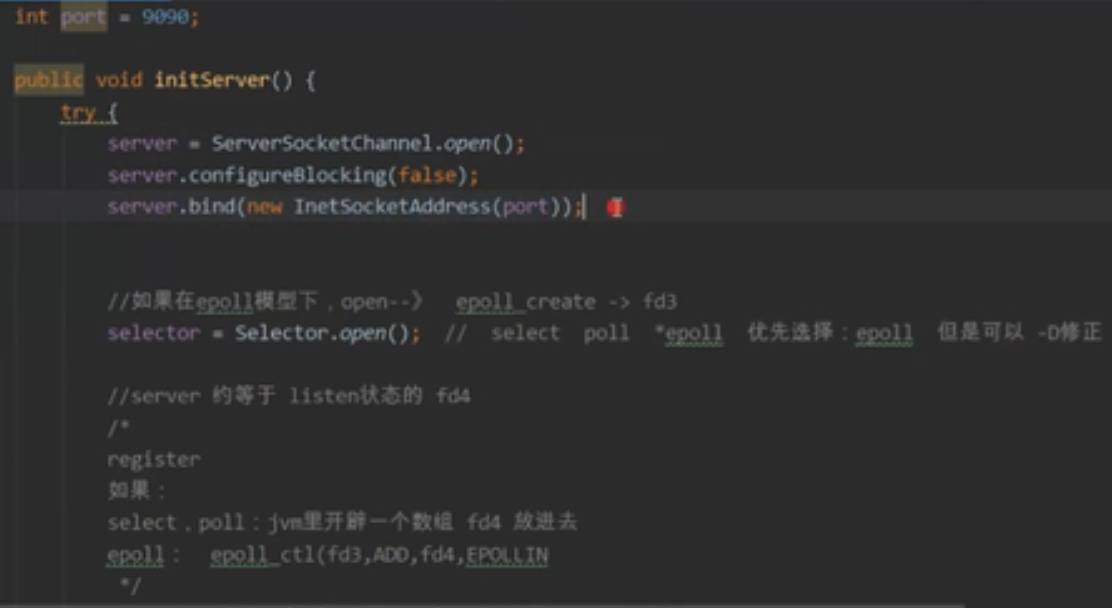


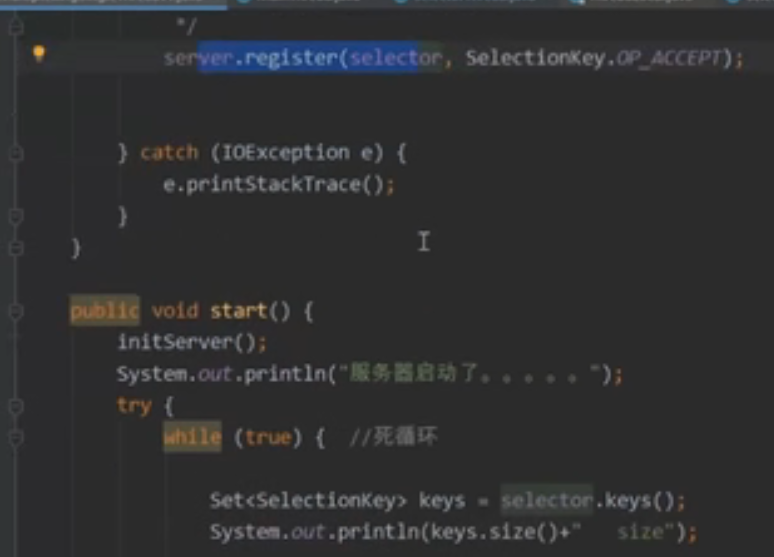
客户端：同上

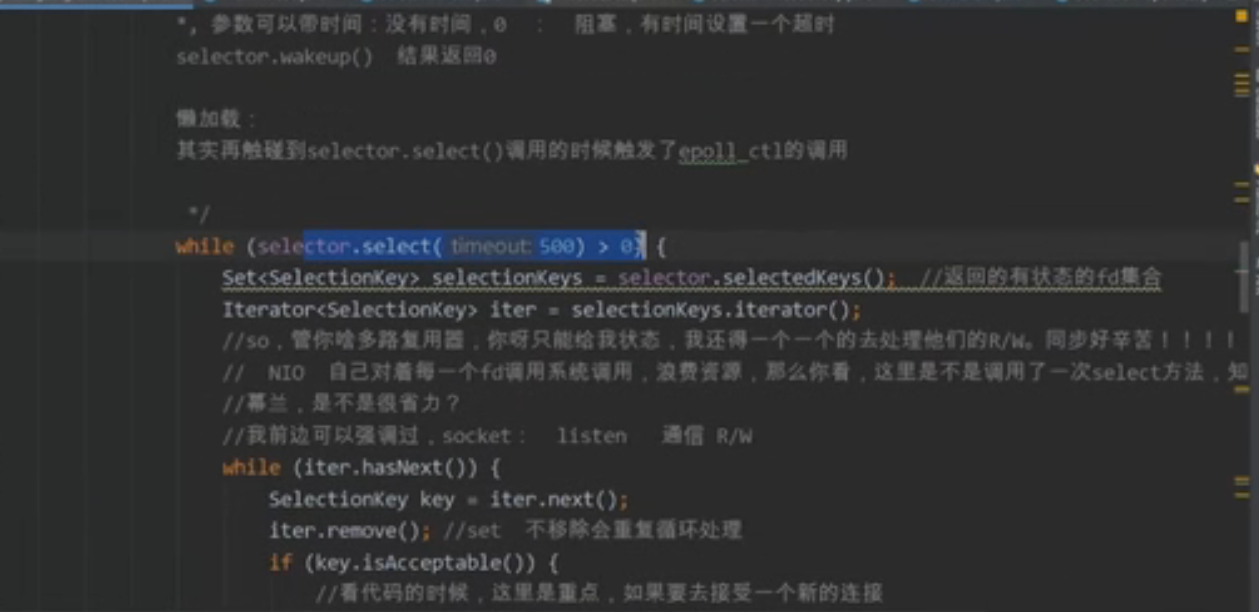
1. 多路复用（比NIO快了非常多）

服务器：

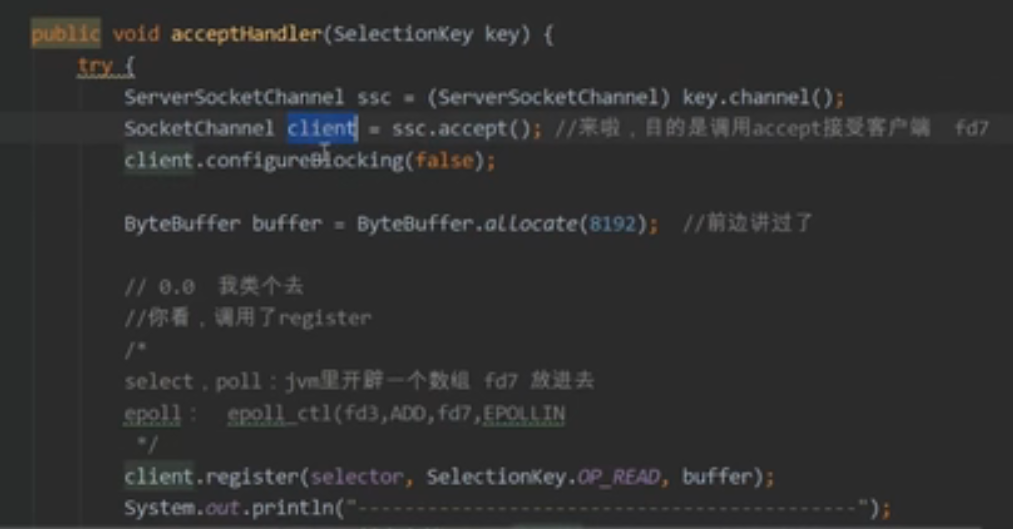


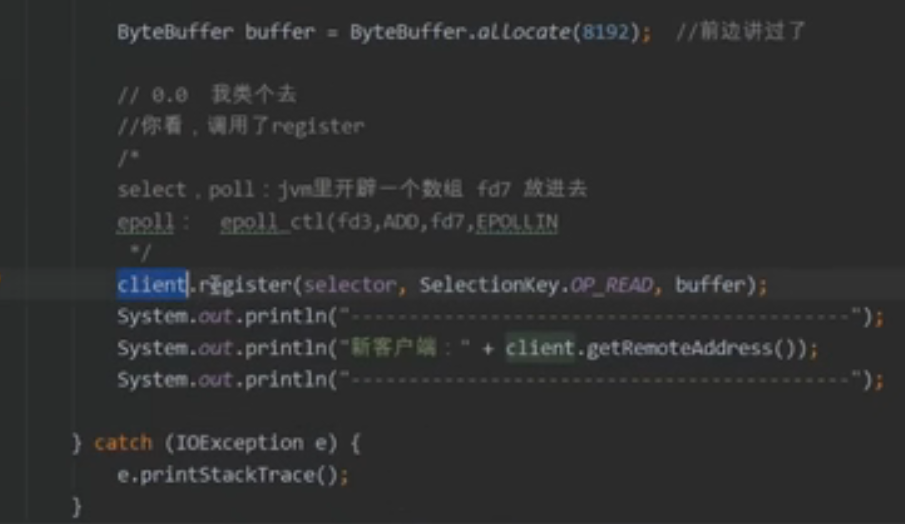






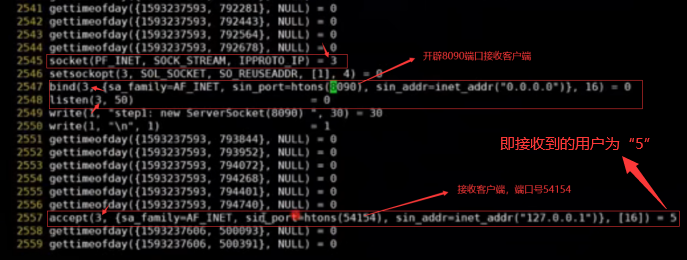




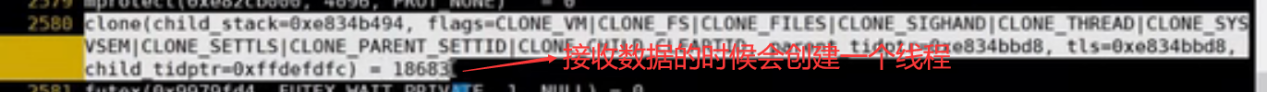


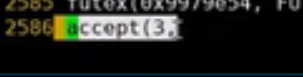
客户端：同上

1. 服务器端和底层交互
2. BIO

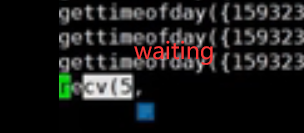


其中：

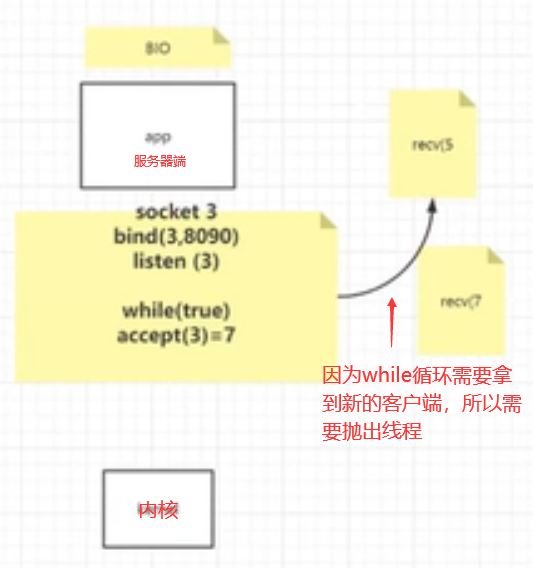


最后因为没有数据阻塞了

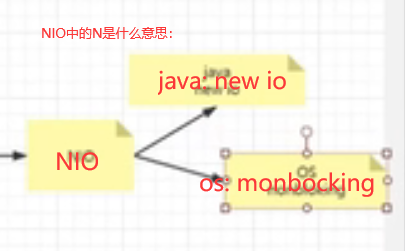
打开接收数据的18683：

 正在等待客户端发数据 --> 抛出

综上：

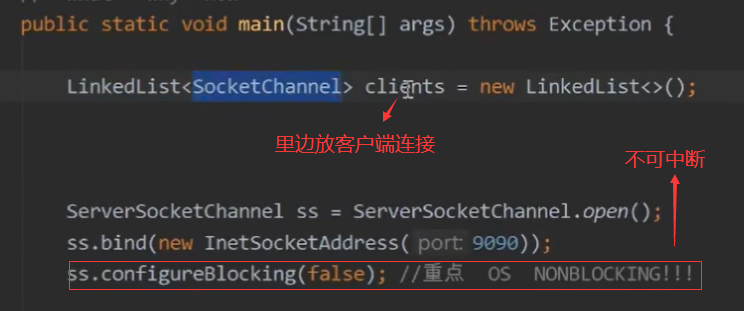
慢，资源消耗大，因为阻塞抛线程

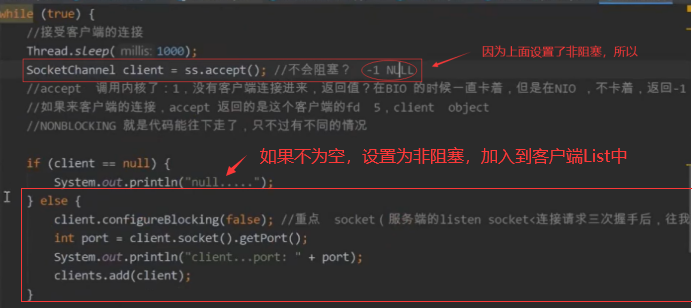
1. NIO

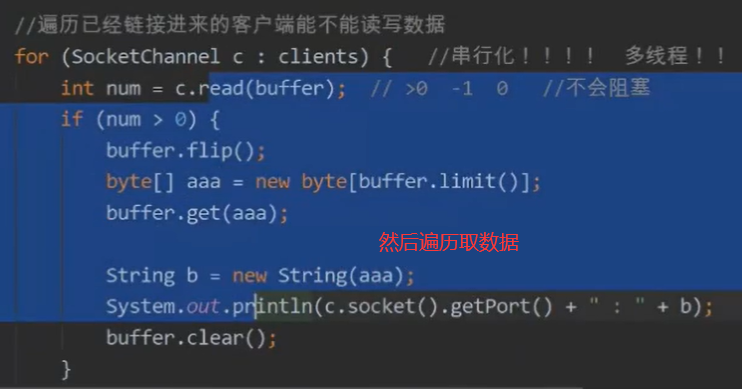
（os: 操作系统）



看代码：







这样每一次连接都会对应一次底层调用，并不是有效连接才进行处理，所以也会浪费空间时间 -----> 一次读多个，谁能读到东西就再读哪个，调用底层（得有一个方法，传递所有fd(文件)可r/w(读写)） ----> 这个方法：多路复用器（select poll）（epoll），返回的是状态，你程序得自己去accept, recv，即此时为同步IO模型

（异步IO模型: 有一个方法，你调了，除了状态，数据都给你写入到用户程序里，你程序不需要recv ----> windows.iocp）

1. 多路复用

看代码调用(select)：

