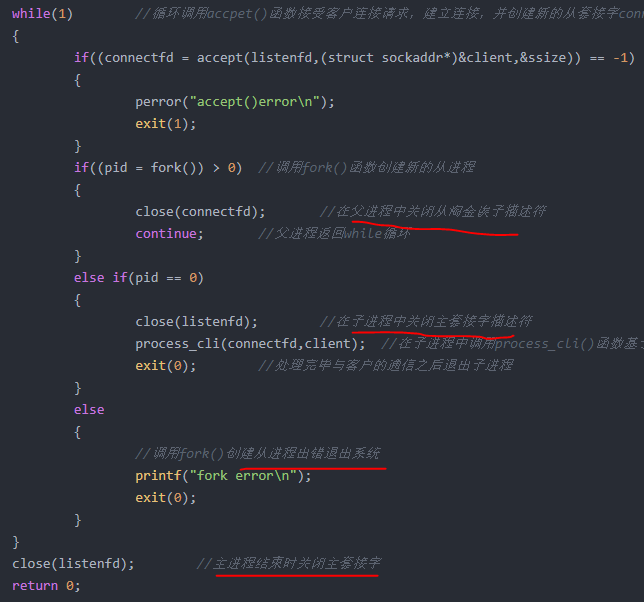
1. Socket编程

果然凡事都要动手才会.jpg。

首先，流程没啥问题，跟理论一样，服务器端，socket,bind,listen,accept，阻塞，然后双方均可读写。但是！！！细节和想的不同。

1. socket通信有两种域，通常学的是基于AF\_INET域的，典型的TCP/IP四层，但正因如此，它消耗CPU，并且数据传输需要经过网卡，受到网卡带宽限制。另一种是AF\_UNIX域，它是用于本地进程间通信的，类似于管道，依赖路径名称标识发送方和接收方，节省cpu，且不经过网卡所以不受限制。编程时，除socket传递不同的域，bind地址结构不同，最后AF\_INET关闭监听套接字，它关闭socket对象。
2. 服务器端，两个套接字（监听套接字，连接套接字），两个地址（自己的，自己设，传给bind；客户端的，accept接收得到）；客户端，只有一个套接字，一个地址（服务器地址，设好才能连上嘛）。
3. 确实有缓冲，自己设置的。。。
4. 针对并发操作，每次服务器跟某一客户端连接结束，则需要关闭其连接套接字；程序结束前，关闭监听套接字。
5. 基于多进程的高并发

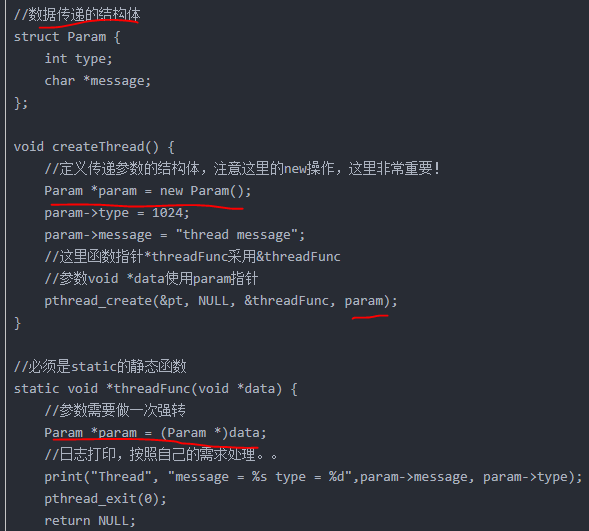


客户端是单独的，不变。

1. 基于多线程的高并发



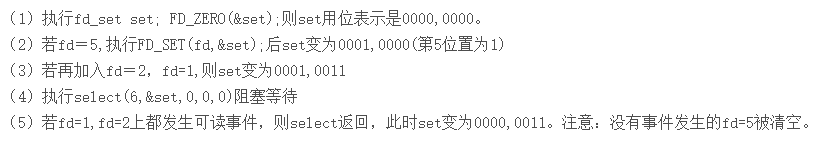
主要考察C++多线程。每次建立一次连接，建立一个线程，处理完以后，关闭连接套接字，把该线程删了，以免满了。头文件是<pthread.h>，如果传参不止一个参数，把需要的参数全打包进struct,传过去再转回来，如下：



注意！Linux下pthread不是默认的库，编译时须在末尾加上-lpthread.

1. 基于select的并发服务器

原理大家都懂，就是select可监控文件集中的fd，如果有读，写申请或错误，就会返回正数（检查原理如下），但不能精确到是谁申请的，于是需要查看（listenfd）或者遍历（cnndfd)。



操作:

FD\_ZERO,清空set。FD\_SET，加入。FD\_ISSET，判断是否该fd有申请。FD\_CLR，从set中删除该fd。

注意：

1. 每次申请都有监控，所以不能一次申请后，读写n次，所以之前写的process那种不行。
2. 由于select每次会清空set,所以传入select前需全部重新传。
3. 如果不想遍历所有fd（包括空的），可设一个maxi，但注意maxi=max(maxi,i+1)。
4. 由于需要遍历所有fd，所以需要自己设一个数组存放连接fd，可初始化为-1,检测到第一个-1就存值，释放后，就再令回-1。
5. 由于一直阻塞等着，所有read结果为0是不可能的，所以可以自己设一个退出键，如传“exit”.退出时必须FD\_CLR以及close(client[i])。

步骤：

Socket->bind->listen->FD\_ZERO->FD\_SET(listenfd,&allset)->

rset=allset->nr=select->

if(FD\_ISSET(listenfd,allset)) accept->FD\_SET(conntfd,allset)->client[i]->maxi->nr--

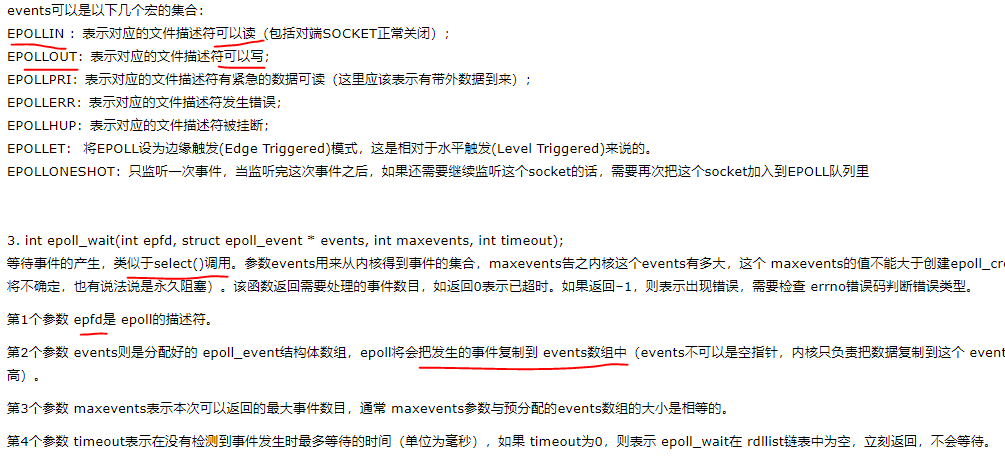
Else read->if(strncmp(“exit”,buf,n)) FD\_CLR close->else 该干啥该啥

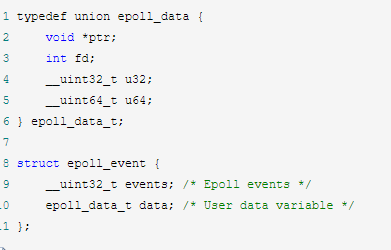
Poll差不多，就不写了，区别就是不用每次都全部更新set。

1. 基于epoll的并发服务器

由于是针对每个fd进行注册，因此可直接遍历，处理一条龙。







步骤：

Socket->bind->listen->epoll\_create(创建一个总的描述符，所以文件加这儿）->

Tmp.event=EPOLLIN(告诉它监听什么活动),tmp.data.fd=listenfd

epoll\_ctl(epfd,EPOLLL\_CTL\_ADD,listenfd,&tmp)->n=epoll\_wait(epfd,e)->for(int i=0;i<n;++i)

->if(!(e[i].events&EPOLLIN)) continue;->

if(e[i].data.fd==listenfd) accept->epoll\_ctl(ADD)

Else 读写->退出时可用select那种，但也必须epoll\_ctl(DEL)和close。