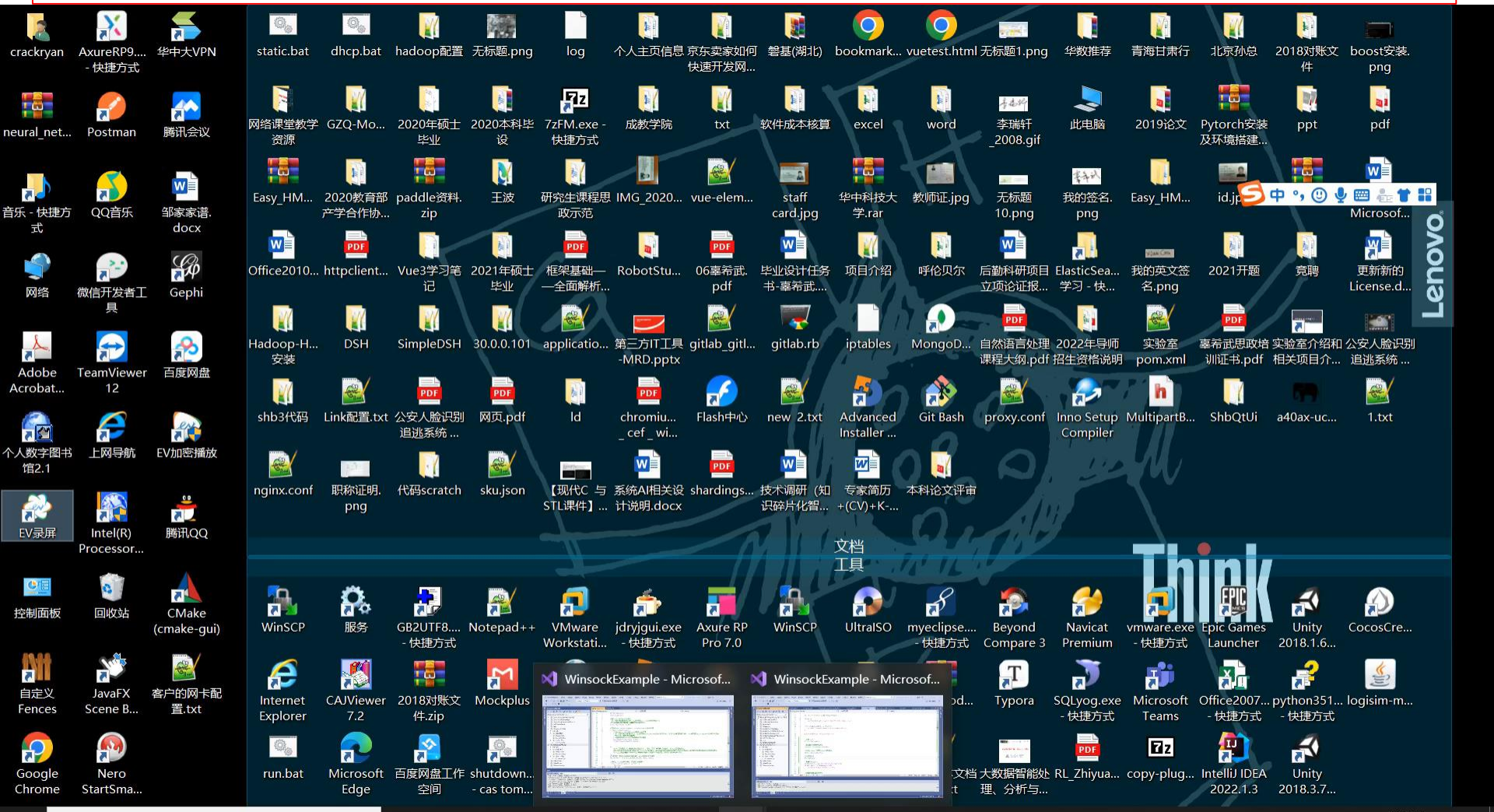
Socket编程实例

VerySimpleServer和VerySimpleClient的实现

VerySimpleServer和VerySimpleClient的运行

- 这个Demo只支持一个Client连接到Server。



VerySimpleServer

- Server.cpp

```
void main(){
        WSADATA wsaData;

    fd_set rfds;

    fd_set wfds;

    bool first_connetion = true;

    …

}
```

select()机制中提供的fd_set的数据结构，实际上是long类型的数组，每一个数组元素都能与一打开的文件句柄（不管是socket句柄，还是其他文件或命名管道或设备句柄）建立联系，建立联系的工作由程序员完成.
当调用select()时，由内核根据IO状态修改fd_set的内容，由此来通知执行了select()的进程哪个socket或文件句柄发生了可读或可写事件。

用于检查socket是否有数据到来的的文件描述符，用于socket非阻塞模式下等待网络事件通知（有数据到来）

用于检查socket是否可以发送的文件描述符，用于socket非阻塞模式下等待网络事件通知（可以发送数据）

是否为用户的第一次请求

VerySimpleServer

- Server.cpp

```
void main(){
        WSADATA wsaData;
        fd_set rfds;
        fd_set wfds;
        bool first_connetion = true;

        int nRc = WSAStartup(0x0202,&wsaData);

        if(nRc){
                printf("Winsock  startup failed with error!\n");
        }

        if(wsaData.wVersion != 0x0202){
                printf("Winsock version is not correct!\n");
        }

        printf("Winsock  startup Ok!\n");
        …
}
```

初始化Winsock环境。八股文，照写就是

VerySimpleServer

- Server.cpp

```
void main(){
        //监听socket
        SOCKET srvSocket;

        //服务器地址和客户端地址
        sockaddr_in addr, clientAddr;

        //会话socket，负责和client进程通信
        SOCKET sessionSocket;

        //ip地址长度
        int addrLen;

        //创建监听socket
        srvSocket = socket(AF_INET,SOCK_STREAM,0);
        if(srvSocket != INVALID_SOCKET)
        printf("Socket create Ok!\n");
        …
}
```

监听socket

服务器实际上是有多个Socket：对于每个客户端，都会创建一个会话socket

创建监听Socket。第二个参数指定使用TCP。八股文，照写

VerySimpleServer

- Server.cpp

```
void main(){
        //设置服务器的端口和地址
        addr.sin_family = AF_INET;
        addr.sin_port = htons(5050);
        addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY); //主机上任意一块网卡的IP地址

        //binding
        int rtn = bind(srvSocket,(LPSOCKADDR)&addr, sizeof(addr));
        if(rtn != SOCKET_ERROR)
        printf("Socket bind Ok!\n");
        //监听
        rtn = listen(srvSocket,5);
        if(rtn != SOCKET_ERROR)
        printf("Socket listen Ok!\n");

        clientAddr.sin_family =AF_INET;
        addrLen = sizeof(clientAddr);
        //设置接收缓冲区
        char recvBuf[4096];
        …
}
```

八股文，照写

VerySimpleServer

- Server.cpp

```
void main(){
        u_long blockMode = 1;          //将srvSock设为非阻塞模式以监听客户连接请求

        if ((rtn = ioctlsocket(srvSocket, FIONBIO, &blockMode) == SOCKET_ERROR))    {
                //FIONBIO：允许或禁止套接口s的非阻塞模式。
                cout << "ioctlsocket() failed with error!\n";
                return;
        }
        cout << "ioctlsocket() for server socket ok! Waiting for client connection and data\n";

        …
}
```

调用ioctlsocket，将srvSocket改为非阻塞模式，改成反复检查fd_set元素的状态，看每个元素对应的句柄是否可读或可写。也就是把阻塞模式改为轮询模式。

VerySimpleServer

Server.cpp

```
void main(){
        while(true){
                //清空rfds和wfds数组
                FD_ZERO(&rfds);
                FD_ZERO(&wfds);

                FD_SET(srvSocket, &rfds);

                //如果first_connetion为true，sessionSocket还没有产生
                if (!first_connetion) {
                        if (sessionSocket != INVALID_SOCKET) {
                                FD_SET(sessionSocket, &rfds);
                                FD_SET(sessionSocket, &wfds);
                        }
                }
                …
        }

}
```

每次循环开始，清空rfds和wfds数组。

将srvSocket加入rfds数组
即：当客户端连接请求到来时，rfds数组里srvSocket对应的的状态为可读因此这条语句的作用就是：设置等待客户连接请求

如果sessionSocket是有效的，将sessionSocket加入rfds数组和wfds数组
当客户端发送数据过来时，rfds数组里sessionSocket的对应的状态为可读；当可以发送数据到客户端时，wfds数组里sessionSocket的对应的状态为可写。因此上面二条语句的作用就是：设置等待会话SOKCET可接受数据或可发送数据

VerySimpleServer

Server.cpp

```
void main(){
        while(true){

                int nTotal = select(0, &rfds, &wfds, NULL, NULL);

                …
        }
}
```

select工作原理：传入要监听的文件描述符集合（可读、可写，有异常）开始监听，select处于阻塞状态。
当有可读写事件发生或设置的等待时间timeout到了就会返回，返回之前自动去除集合中无事件发生的文件描述符，返回时传出有事件发生的文件描述符集合。
但select传出的集合并没有告诉用户集合中包括哪几个就绪的文件描述符，需要用户后续进行遍历操作(通过FD_ISSET检查每个文件描述符的状态)。

开始等待，等待rfds里是否有输入事件，wfds里是否有可写事件
The select function returns the total number of socket handles that are ready and contained in the fd_set structure
返回总共可以读或写的文件描述符个数

VerySimpleServer

Server.cpp

如果srvSock收到连接请求，接受客户连接请求

因为客户端请求到来也算可读事件，因此-1，剩下的就是真正有可读事件的句柄个数（即有多少个socket收到了数据）

把会话SOCKET设为非阻塞模式

设置等待会话SOKCET可接受数据或可发送数据

已经产生了会话socket，因此设为false

```
void main(){
        while(true){
            if (FD_ISSET(srvSocket, &rfds)) {
                nTotal--;
                //产生会话SOCKET
                sessionSocket = accept(srvSocket, (LPSOCKADDR)&clientAddr, &addrLen);
                if (sessionSocket != INVALID_SOCKET)
                    printf("Socket listen one client request!\n");

                if ((rtn = ioctlsocket(sessionSocket, FIONBIO, &blockMode) == SOCKET_ERROR)) {
                    cout << "ioctlsocket() failed with error!\n";
                    return;
                }
                cout << "ioctlsocket() for session socket ok! Waiting for client connection and data\n";

                FD_SET(sessionSocket, &rfds);
                FD_SET(sessionSocket, &wfds);

                first_connetion = false;
            }
            …
        }
}
```

VerySimpleServer

Server.cpp

如果还有可读事件

如果会话SOCKET有数据到来，则接受客户的数据

```
void main(){
        while(true){
            if (nTotal > 0) {
                if (FD_ISSET(sessionSocket, &rfds)) {
                    //receiving data from client
                    memset(recvBuf, '\0', 4096);
                    rtn = recv(sessionSocket, recvBuf, 256, 0);
                    if (rtn > 0) {
                        printf("Received %d bytes from client: %s\n", rtn, recvBuf);
                    }
                    else {
                        printf("Client leaving ...\n");
                        closesocket(sessionSocket); //既然client离开了，就关闭sessionSocket
                        nTotal--;       //因为客户端离开也属于可读事件，所以需要-1
                        sessionSocket = INVALID_SOCKET; //把sessionSocket设为INVALID_SOCKET
                    }
                }
            }
        }
}
```

否则是收到了客户端断开连接的请求，也算可读事件。但是这种情况下FD_ISSET(sessionSocket, &rfds)返回false

VerySimpleServer

- 从这个例子可以看到
- 如果Socket Server不采用多线程，则fd_set、ioctlsocket、select对于Socket Server的实现非常重要
- 当然一个真正的服务器是需要用多线程实现的：对于每个客户端请求，会产生一个Session Socket，并并且将这个Session Socket传递给一个子线程，在子线程里通过Session Socket和客户端通信。在子线程里可以用while(true)循环，这时可以不用fd_set、ioctlsocket、select这套机制
- 为什么在不采用多线程的情况下，fd_set、ioctlsocket、select对于Socket Server的实现非常重要？
- 这要从阻塞式函数谈起。

阻塞式函数调用

- 当我们的应用程序发出对操作系统的一个调用时，通常这个调用是阻塞式的。例如我们打开一个文件：

```
void main(){

        open（文件）

        …

}
```

当调用open函数打开文件时，main函数的执行被阻塞了：即停留在open语句，等待操作系统打开文件，返回文件句柄。知道open函数返回，才会执行下一条语句。

阻塞式函数调用

- Socket API里面的函数都是阻塞式的，特别是：
  - accept：等待客户端的连接请求
  - recv：等待客户端数据到来
- 那么阻塞式的Socket API给我们实现Server带来什么问题呢?(不采用多线程的情况下)
- Server程序的代码逻辑通常是放在一个while（true）循环里，如果我们在while（true）循环里直接调用accept、 recv会发生什么情况呢？请看代码

阻塞式函数SOCKETAPI调用

- 这是我们的Server端程序

```
void main(){
    while（true）{

        …
        sessionSocket = accept(srvSocket, (LPSOCKADDR)&clientAddr, &addrLen);

        …

        recv(sessionSocket, recvBuf, 256, 0);


    }

}
```

等待客户端连接，阻塞式。因此这里是一个阻塞点

等待客户端的数据到来，阻塞式。因此这里是第二个阻塞点

在while循环里出现了二个阻塞点。假设Client1的连接请求已经到来，产生了sessionSocket，然后执行到recv语句，假设Client1的数据迟迟不到，那么while循环就被阻塞在recv语句。假设这时第二个Client发出连接请求，由于while循环被阻塞在recv语句，无法执行到上面的accept语句，因此这时服务器是无法对第二个Client的连接请求作出相应的。这就是单线程服务的问题。

阻塞式函数调用

- 如果我们采用fd_set、ioctlsocket、select这套机制，把Socket改为非阻塞模式，那么accept函数、recv函数都不是阻塞式的了
- 而阻塞点被全部集中到select函数了
- 看代码

阻塞式函数SOCKETAPI调用

- 这是我们的Server端程序

```
void main(){
    while（true）{

        …
        ioctlsocket(srvSocket, FIONBIO, &blockMode) == SOCKET_ERROR
        FD_SET(srvSocket, &rfds);
        int nTotal = select(0, &rfds, &wfds, NULL, NULL);
        if (FD_ISSET(srvSocket, &rfds)) {
            sessionSocket = accept(srvSocket, (LPSOCKADDR)&clientAddr, &addrLen);
            ioctlsocket(sessionSocket, FIONBIO, &blockMode)
            FD_SET(sessionSocket, &rfds);
            FD_SET(sessionSocket, &wfds);
            if (FD_ISSET(sessionSocket, &rfds)) {
                recv(sessionSocket, recvBuf, 256, 0);
            }
    }
}
```

现在把监听socket和会话socket全部改成非阻塞模式，那么accept和recv函数不再阻塞。While循环里就剩下一个阻塞点select，每次循环都通过select去集中轮询所有的可读事件（是否有可读端请求、是否有客户端数据到来，可以避免前面出现的问题）。