# C++Socket网络通信

# 目录

[C++Socket网络通信 1](#_Toc33298471)

[目录 1](#_Toc33298472)

[管理C++项目 6](#_Toc33298473)

[调试方法 7](#_Toc33298474)

[打印数据来调试 7](#_Toc33298475)

[普通断点 7](#_Toc33298476)

[条件断点 9](#_Toc33298477)

[条件追踪 9](#_Toc33298478)

[单步调试 9](#_Toc33298479)

[添加监视 9](#_Toc33298480)

[基础知识 9](#_Toc33298481)

[网络分层 9](#_Toc33298482)

[OSI 7层参考模型 9](#_Toc33298483)

[TCP/IP 4层模型 9](#_Toc33298484)

[TCP协议头字段 10](#_Toc33298485)

[三次握手和四次挥手 11](#_Toc33298486)

[链接、可靠、字节流、传输层 11](#_Toc33298487)

[Socket基础Api 12](#_Toc33298488)

[网络库与网络头文件 12](#_Toc33298489)

[WSAStartup()详解 12](#_Toc33298490)

[参数1：WORD wVersionRequested 13](#_Toc33298491)

[参数2：LPWSADATA lpWSAData 13](#_Toc33298492)

[返回值 13](#_Toc33298493)

[代码范例 14](#_Toc33298494)

[版本校验 15](#_Toc33298495)

[什么是socket 16](#_Toc33298496)

[socket()详解 16](#_Toc33298497)

[参数1：int af 16](#_Toc33298498)

[参数2：int type 16](#_Toc33298499)

[参数3：int protocol 17](#_Toc33298500)

[返回值 17](#_Toc33298501)

[代码范例 17](#_Toc33298502)

[bind()详解 17](#_Toc33298503)

[参数1：SOCKET s 18](#_Toc33298504)

[参数2：const sockaddr \*addr 18](#_Toc33298505)

[参数3：int namelen 18](#_Toc33298506)

[返回值 18](#_Toc33298507)

[代码范例 19](#_Toc33298508)

[listen() 19](#_Toc33298509)

[参数1：SOCKET s 19](#_Toc33298510)

[参数2：int backlog 19](#_Toc33298511)

[WSAAPI 20](#_Toc33298512)

[返回值 20](#_Toc33298513)

[代码范例 20](#_Toc33298514)

[accept() 20](#_Toc33298515)

[参数1：SOCKET s 21](#_Toc33298516)

[参数2：sockaddr \*addr 21](#_Toc33298517)

[参数3：int \*addrlen 21](#_Toc33298518)

[返回值 21](#_Toc33298519)

[代码范例 21](#_Toc33298520)

[recv() 22](#_Toc33298521)

[参数1：SOCKET s 22](#_Toc33298522)

[参数2：char \*buf 22](#_Toc33298523)

[参数3：int len 22](#_Toc33298524)

[参数4：int flags 22](#_Toc33298525)

[返回值 23](#_Toc33298526)

[代码范例 24](#_Toc33298527)

[send() 24](#_Toc33298528)

[参数1：SOCKET s 24](#_Toc33298529)

[参数2：const char \*buf 24](#_Toc33298530)

[参数3：int len 25](#_Toc33298531)

[参数4：int flags 25](#_Toc33298532)

[返回值 25](#_Toc33298533)

[代码范例 26](#_Toc33298534)

[服务端 26](#_Toc33298535)

[客户端 26](#_Toc33298536)

[connect() 26](#_Toc33298537)

[参数1：SOCKET s 27](#_Toc33298538)

[参数2：const sockaddr \*name 27](#_Toc33298539)

[参数3：int namelen 27](#_Toc33298540)

[返回值 27](#_Toc33298541)

[客户端connect()代码范例 27](#_Toc33298542)

[完善C/S模型 30](#_Toc33298543)

[select模型 30](#_Toc33298544)

[fd\_set 31](#_Toc33298545)

[select() 31](#_Toc33298546)

[参数1：int nfds 31](#_Toc33298547)

[参数2：fd\_set \*readfds 31](#_Toc33298548)

[参数3：fd\_set \*writefds 32](#_Toc33298549)

[参数4：fd\_set \*exceptfds 32](#_Toc33298550)

[参数5：const timeval \*timeout 32](#_Toc33298551)

[返回值 32](#_Toc33298552)

[代码范例 32](#_Toc33298553)

[select()处理参数3 34](#_Toc33298554)

[select()处理参数4 35](#_Toc33298555)

[select模型总结 36](#_Toc33298556)

[释放socket集合 36](#_Toc33298557)

[控制台关闭事件处理 36](#_Toc33298558)

[Windows消息机制和事件机制 37](#_Toc33298559)

[消息机制 37](#_Toc33298560)

[事件机制 37](#_Toc33298561)

[事件选择模型的处理逻辑 38](#_Toc33298562)

[WSACreateEvent() 38](#_Toc33298563)

[WSACloseEvent() 38](#_Toc33298564)

[WSAResetEvent() 38](#_Toc33298565)

[WSASetEvent() 39](#_Toc33298566)

[WSAEventSelect() 39](#_Toc33298567)

[参数1 ：SOCKET s 39](#_Toc33298568)

[参数2：WSAEVENT hEventObject 39](#_Toc33298569)

[参数3：long lNetworkEvents 39](#_Toc33298570)

[返回值 40](#_Toc33298571)

[WSAWaitForMultipleEvents() 40](#_Toc33298572)

[参数1：DWORD cEvents 40](#_Toc33298573)

[参数2：const WSAEVENT \*lphEvents 40](#_Toc33298574)

[参数3：BOOL fWaitAll 40](#_Toc33298575)

[参数4：DWORD dwTimeout 40](#_Toc33298576)

[参数5：BOOL fAlertable 41](#_Toc33298577)

[返回值 41](#_Toc33298578)

[WSAEnumNetworkEvents() 41](#_Toc33298579)

[参数1：SOCKET s 41](#_Toc33298580)

[参数2：WSAEVENT hEventObject 41](#_Toc33298581)

[参数3：LPWSANETWORKEVENTS lpNetworkEvents 41](#_Toc33298582)

[返回值 42](#_Toc33298583)

[事件分类处理逻辑 42](#_Toc33298584)

[switch处理以及else if处理以及if处理的区别 42](#_Toc33298585)

[有序处理优化 42](#_Toc33298586)

[增加处理事件的数量 42](#_Toc33298587)

[释放 42](#_Toc33298588)

[select模型与事件选择模型对比 43](#_Toc33298589)

[异步选择模型 43](#_Toc33298590)

[逻辑 43](#_Toc33298591)

[窗口 43](#_Toc33298592)

[WSAAsyncSelect() 44](#_Toc33298593)

[参数1：SOCKET s 44](#_Toc33298594)

[参数2：HWND hWnd 44](#_Toc33298595)

[参数3：u\_int wMsg 44](#_Toc33298596)

[参数4：long lEvent 44](#_Toc33298597)

[返回值 44](#_Toc33298598)

[异步选择模型问题分析 45](#_Toc33298599)

[重叠IO模型-事件通知 45](#_Toc33298600)

[重叠IO介绍 45](#_Toc33298601)

[重叠IO逻辑 46](#_Toc33298602)

[事件通知实现逻辑 46](#_Toc33298603)

[WSASocket() 47](#_Toc33298604)

[参数1：int af 47](#_Toc33298605)

[参数2：int type 47](#_Toc33298606)

[参数3：int protocol 47](#_Toc33298607)

[参数4：LPWSAPROTOCOL\_INFOA lpProtocolInfo 47](#_Toc33298608)

[参数5：GROUP g 47](#_Toc33298609)

[参数6：DWORD dwFlags 47](#_Toc33298610)

[返回值 48](#_Toc33298611)

[AcceptEx() 48](#_Toc33298612)

[参数1：SOCKET sListenSocket 48](#_Toc33298613)

[参数2：SOCKET sAcceptSocket 48](#_Toc33298614)

[参数3：PVOID lpOutputBuffer 48](#_Toc33298615)

[参数4：DWORD dwReceiveDataLength 49](#_Toc33298616)

[参数5：DWORD dwLocalAddressLength 49](#_Toc33298617)

[参数6：DWORD dwRemoteAddressLength 49](#_Toc33298618)

[参数7：LPDWORD lpdwBytesReceived 49](#_Toc33298619)

[参数8：LPOVERLAPPED lpOverlapped 49](#_Toc33298620)

[返回值 49](#_Toc33298621)

[WSARecv() 49](#_Toc33298622)

[参数1：SOCKET s 50](#_Toc33298623)

[参数2：LPWSABUF lpBuffers 50](#_Toc33298624)

[参数3：DWORD dwBufferCount 50](#_Toc33298625)

[参数4：LPDWORD lpNumberOfBytesRecvd 50](#_Toc33298626)

[参数5：LPDWORD lpFlags 50](#_Toc33298627)

[参数6：LPWSAOVERLAPPED lpOverlapped 51](#_Toc33298628)

[参数7：LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine 51](#_Toc33298629)

[返回值 51](#_Toc33298630)

[WSAGetOverlappedResult() 51](#_Toc33298631)

[参数1：SOCKET s 51](#_Toc33298632)

[参数2：LPWSAOVERLAPPED lpOverlapped 51](#_Toc33298633)

[参数3：LPDWORD lpcbTransfer 51](#_Toc33298634)

[参数4：BOOL fWait 52](#_Toc33298635)

[参数5：LPDWORD lpdwFlags 52](#_Toc33298636)

[返回值 52](#_Toc33298637)

[WSASend() 52](#_Toc33298638)

[参数1：SOCKET s 52](#_Toc33298639)

[参数2：LPWSABUF lpBuffers 52](#_Toc33298640)

[参数3：DWORD dwBufferCount 52](#_Toc33298641)

[参数4：LPDWORD lpNumberOfBytesSent 53](#_Toc33298642)

[参数5：DWORD dwFlags 53](#_Toc33298643)

[参数6：LPWSAOVERLAPPED lpOverlapped 53](#_Toc33298644)

[参数7：LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine 53](#_Toc33298645)

[返回值 53](#_Toc33298646)

[重叠IO模型-完成例程 53](#_Toc33298647)

[完成例程回调函数 53](#_Toc33298648)

[参数1：DWORD dwError 54](#_Toc33298649)

[参数2：DWORD cbTransferred 54](#_Toc33298650)

[参数3：LPWSAOVERLAPPED lpOverlapped 54](#_Toc33298651)

[参数4：DWORD dwFlags 54](#_Toc33298652)

[重叠IO模型总结 54](#_Toc33298653)

[完成端口模型 55](#_Toc33298654)

[完成端口逻辑 55](#_Toc33298655)

[CreateIoCompletionPort() 55](#_Toc33298656)

[参数1：HANDLE FileHandle 55](#_Toc33298657)

[参数2：HANDLE ExistingCompletionPort 56](#_Toc33298658)

[参数3：ULONG\_PORT CompletionKey 56](#_Toc33298659)

[参数4：DWORD NumberOfConcurrentThreads 56](#_Toc33298660)

[返回值 56](#_Toc33298661)

[CreateThread() 56](#_Toc33298662)

[参数1：LPSECURITY\_ATTRIBUTES lpThreadAttributes 56](#_Toc33298663)

[参数2：SIZE\_T dwStackSize 56](#_Toc33298664)

[参数3：LPTHREAD\_START\_ROUTINE lpStartAddress 57](#_Toc33298665)

[参数4：LPVOID lpParameter 57](#_Toc33298666)

[参数5：DWORD dwCreationFlags 57](#_Toc33298667)

[参数6：LPDWORD lpThreadld 57](#_Toc33298668)

[返回值 57](#_Toc33298669)

[GetQueuedCompletionStatus() 57](#_Toc33298670)

[参数1：HANDLE CompletionPort 57](#_Toc33298671)

[参数2：LPDWORD lpNumberOfBytes 58](#_Toc33298672)

[参数3：PULONG\_PTR lpCompletionKey 58](#_Toc33298673)

[参数4：LPOVERLAPPED \*lpOverlapped 58](#_Toc33298674)

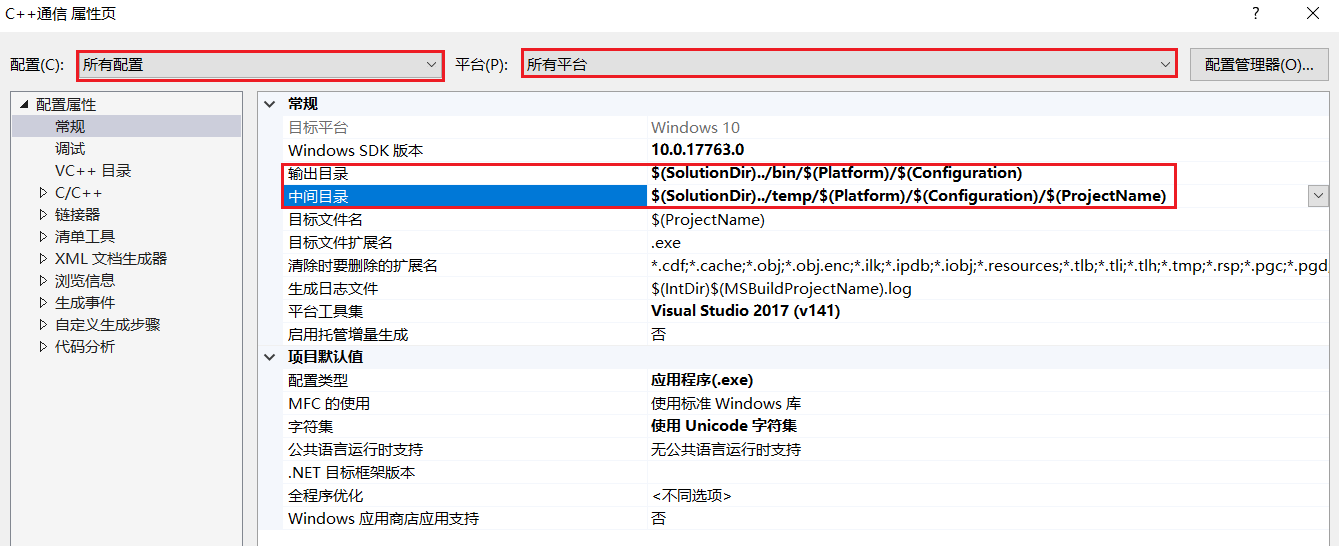
[参数5：DWORD dwMillseconds 58](#_Toc33298675)

[返回值 58](#_Toc33298676)

[分类处理 58](#_Toc33298677)

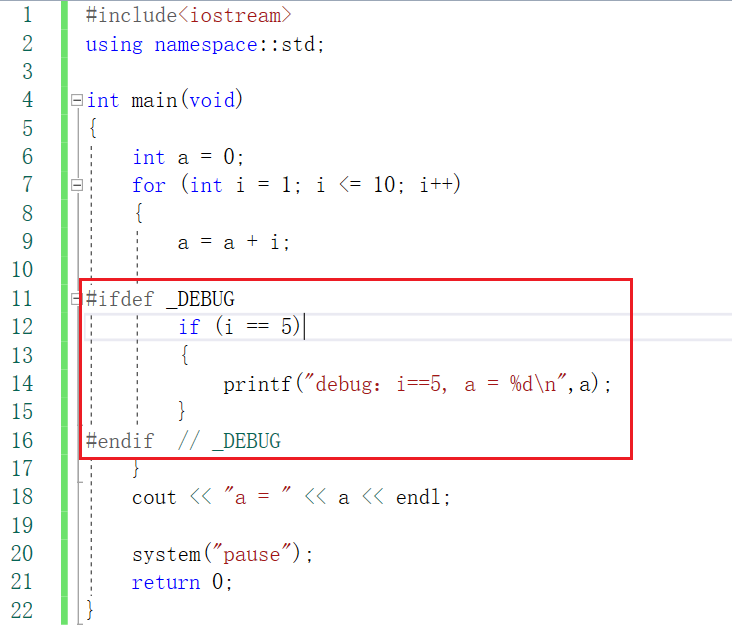
[完成端口模型总结 59](#_Toc33298678)

# 管理C++项目



# 调试方法

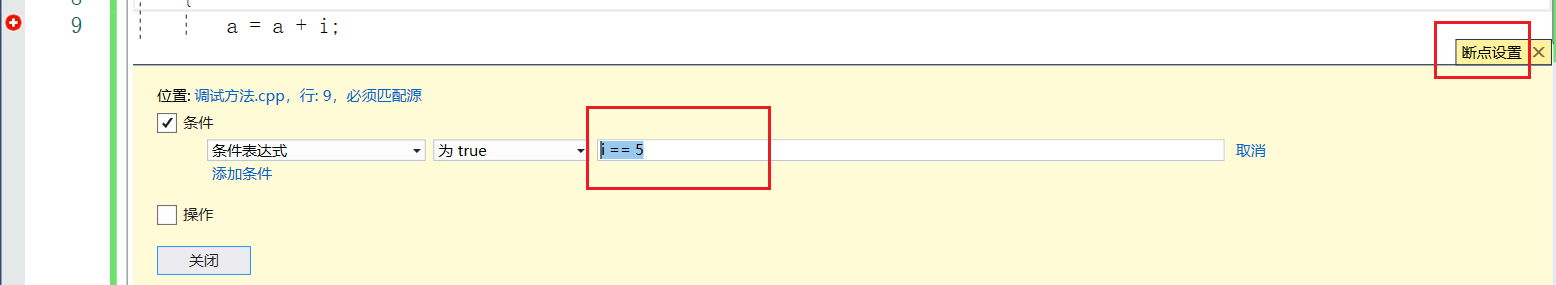
## 打印数据来调试



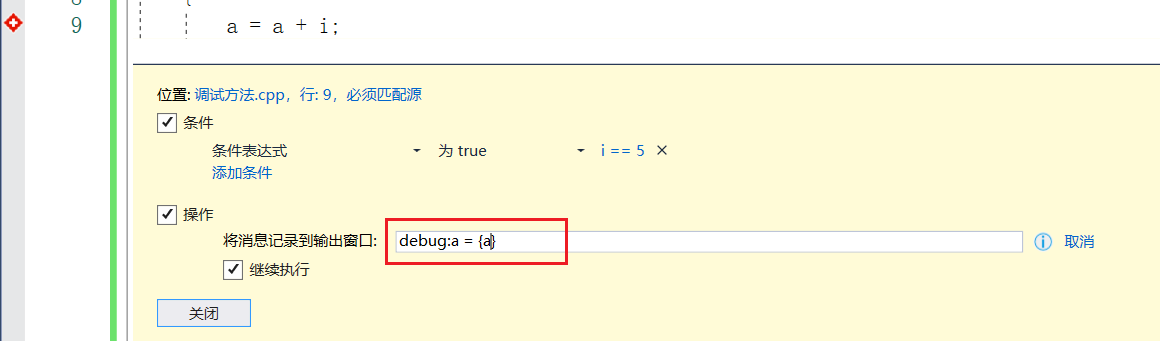
Release状态下，#ifdef \_DEBUG…#endif中间的代码不编译。

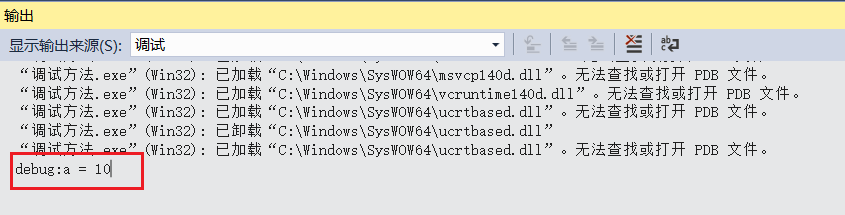
## 普通断点

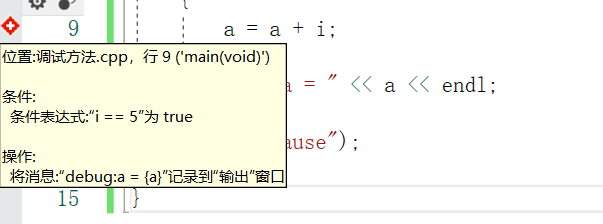
断点设置：



当i == 5时才触发断点。







## 条件断点

## 条件追踪

## 单步调试

## 添加监视

# 基础知识

## 网络分层

### OSI 7层参考模型

模型理解：发送方是打包的过程从上而下；接收放是拆包的过程从下而上。

应用层：为程序员提供编码调用的API接口；给程序员用的；将复杂的协议封装成函数。

表示层：双方通信数据的摆布规则，比如，数据加密解密，压缩解压，数据的语法表示问题。

会话层：建立、管理、终止表示层的实体会话；删除，存储，管理客户端的socket。

传输层：将我们的代码包上一层协议的包头，即指定了该数据包的发送方式。

网络层：给数据包绑上目标地址，即IP地址。

数据链路层：将软件层面打包好的数据，变成硬件层面可发送的数据。比如，将数据切成很多小段发送（帧），匹配当前的带宽，平衡通信双方的通信质量，双方通信之前的信息交换，比如，连接，本地资源准备好。

物理层：硬件设备，传输数据的物理通路。

### TCP/IP 4层模型

应用层：应用、表示、会话。

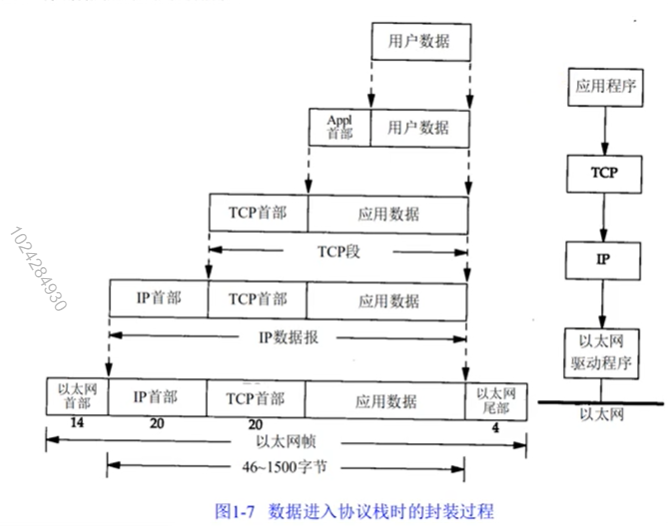
传输层：

网络层：

数据链路层：逻辑链路层，包含物理层，网卡。

## TCP协议头字段

也就是在传输层，这个会包到咱们数据的头部。



发送序列号（seq）：数据段首字节编号，比如，一次发送500字节（头+数据），第一个字节编号，可能是5000，那么这一段最后一个字节编号就是54999。

接收时的确认序列号（ack）：接收方告诉咱们的，我下一次接收的数据的编号；如果咱们的500个数据被完整收到，那么这里就是5500；如果该段数据没有被完整收到，丢帧了，那么则是5000。

偏移值：4为二进制数，最大是15，单位4字节，即最大偏移60字节。

保留：暂时没定义意义。

URG：发送端设置，表示我发送的这组数据你要立刻处理，别等待后续的数据包。紧急指针，目前很少使用。

ACK：确认接收到了TCP数据包。

PSH：发送端设置，表示我收到的这段包没了，你那边把收到的组合吧。主要是最后一段数据置1。

RST：网络重启。

SYN：虚拟电路接通，咱俩建立通信路的意思。发起连接。客户端和服务器连接的时候使用。

FIN：结束连接，正常下线，释放连接。

窗口大小：表示我最大一次能接收的数据，首先于自己的带宽。

校验和：整个数据包的字节数，接收端也会验证。

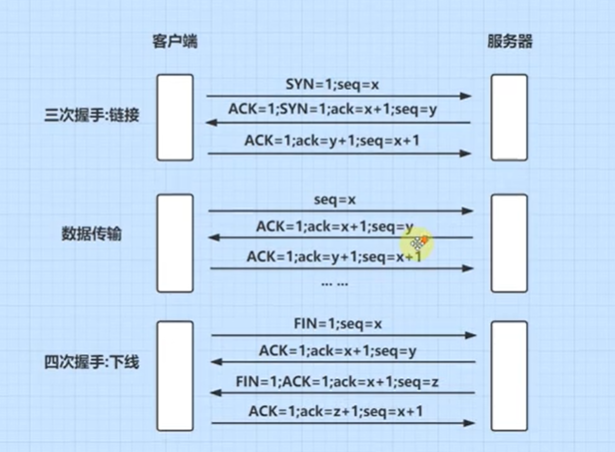
紧急指针：它是一个偏移量，配合URG使用，和序号字段中的值相加表示紧急数据最后一个字节的序号。该段数据紧急处理。

可选项：自定义自己关心的数据，时间点等。

链接建立：三次握手。

链接断开：四次握手。

## 三次握手和四次挥手



## 链接、可靠、字节流、传输层

全双工：可以同时发送和接收数据，这意味着数据流在一个方向上的传输是独立于另一个方向上的传输的。TCP就是全双工的。

半双工：发的时候不能收，收的时候不能发。

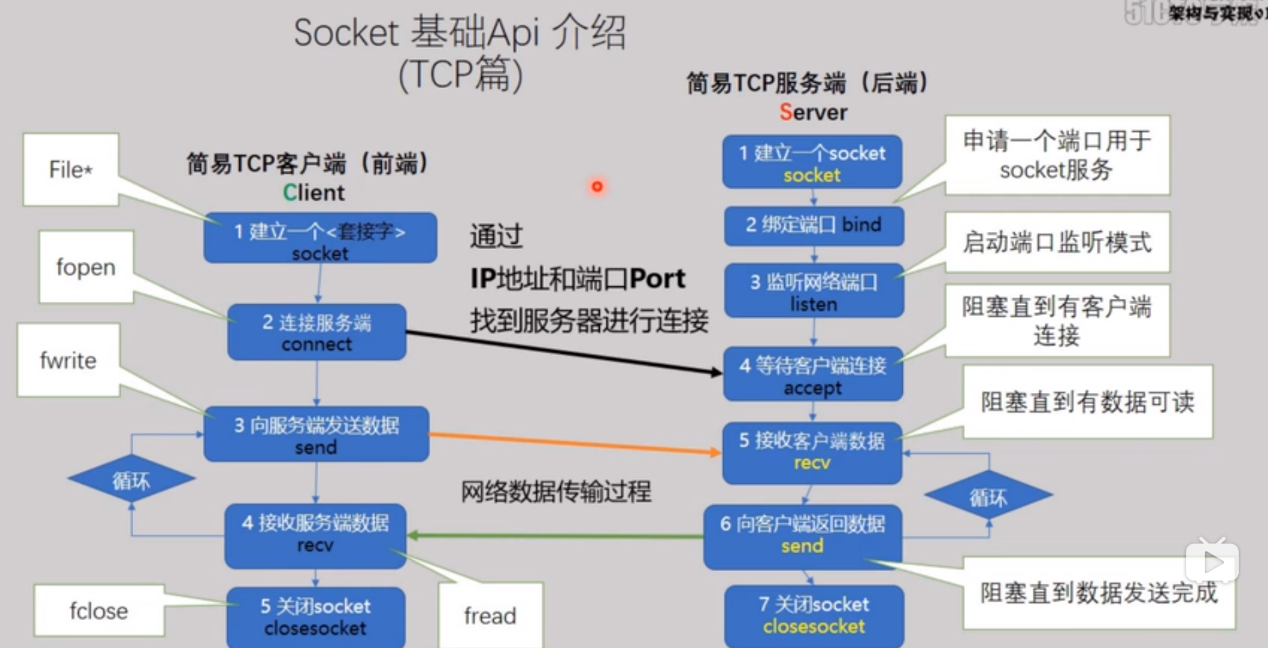
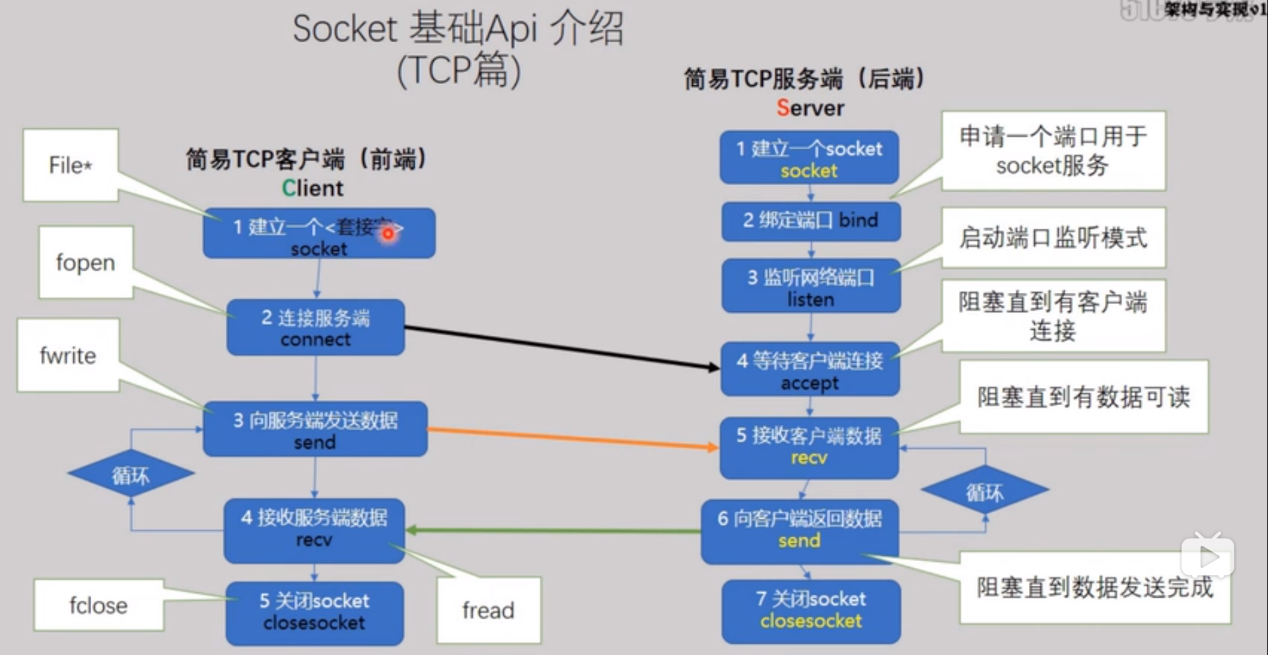
链接：connect()、accept()，三次握手，四次握手。

可靠：对发送的每个字节进行编号，校验和。三次握手保证了开始通信成功。接收方验证。如果数据丢失，那么接收确认ack就是上次的编号。接收方再次返回该段首顺序号，发送方就重发这段。四次握手保证最后接收成功。

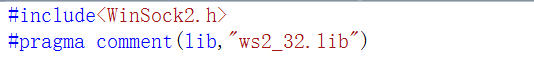
字节流：按字节操作。将数据包组合或剪碎以适合当前带宽。可以缓存，调整合并为有序再传到应用层。

传输层：因为处于网络分层中的传输层，控制传输格式。

# Socket基础Api



# 网络库与网络头文件



我们的开发环境支持哪个版本？？？

打开头文件，WinSock2.h，查看当前编译器环境支持的最高版本。

库文件：编译好的二进制代码。

不管是64编译环境还是32编译环境，都使用ws2\_32.lib这个库文件，没有ws2\_64.lib。

# WSAStartup()详解

作用：打开网络库，启动网络库。

int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);

功能：打开网络库/启动网络库，启动了这个库，这个库里的函数才能使用。

W：window。

S：socket。

A：Asynchronous（异步）。

## 参数1：WORD wVersionRequested

WORD：unsigned short。

MAKEWORD(主版本,副版本)：MAKEWORD(2,2)。**主版本在低数据位，副版本在高数据位。在内存中如何存储？？？，小端存储，低数据位存储在低地址位，高数据位存储在高地址位**。

wVersionRequested：数据高位/高地址是副版本；数据低位/低地址是主版本。

## 参数2：LPWSADATA lpWSAData

**启动网络库之后，系统会把网络库的一些基本信息传递给我们，这个参数就是用来保存系统传递过来的网络库的基本信息的**。

struct WSADATA

{

WORD wVersion;

WORD wHighVersion;

unsigned short iMaxSockets;

unsigned short iMaxUdpDg;

char \*lpVendorInfo;

char szDescription[WSADESCRIPTION\_LEN+1];

char szSystemStatus[WSASYS\_STATUS\_LEN+1];

}

wVersion：我们要使用的版本。

wHighVersion：系统能提供给我们的最高版本。

iMaxSockets：返回可用的socket数量。2版本之后就没用了。

iMaxUdpDg：UDP数据报信息的大小。2版本之后就没用了。

lpVendorInfo：供应商特定的信息。2版本之后就没用了。

szDescription：当前库的描述信息。

szSystemStatus：

当输入的版本不存在时：

1. 有主版本，没有副版本：得到该主版本的最大副版本。
2. 超过最大版本号：使用系统能提供的最大的版本。
3. 主版本是0：网络库打开失败，不支持请求的套接字版本。

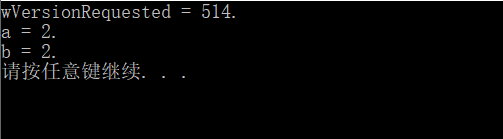
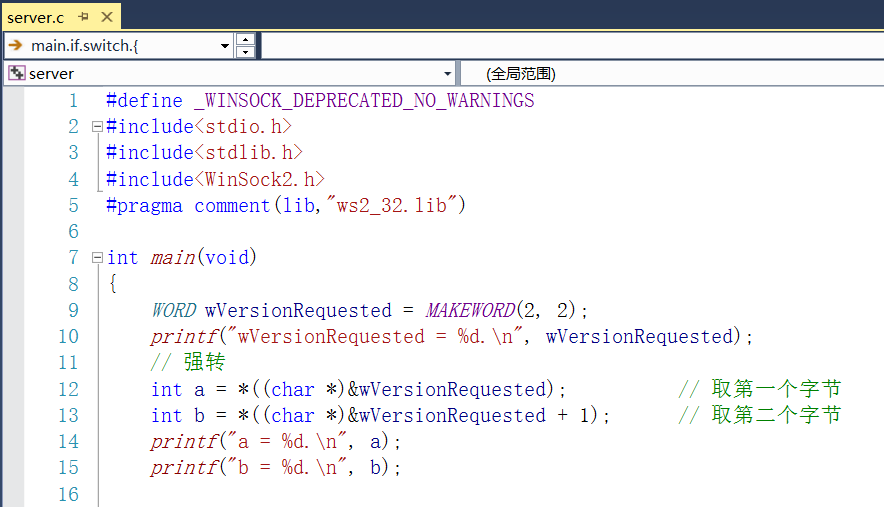
## 返回值

返回0为执行正确。

返回非0执行失败：

1. WSASYSNOTREADY（10091）：底层网络子系统尚未准备好进行网络通信。
2. WSAVERNOTSUPPORTED（10092）：此特定Windows套接字实现不提供所请求的Windows套接字支持的版本。
3. WSAEPROCLIM（10067）：已达到对Windows套接字实现支持的任务数量的限制。
4. WSAEINPROGRESS（10036）：正在阻止Windows Socket X.X操作。
5. WSAEFAULT（10014）：lpWSAData参数不是有效指针。

## 代码范例





# 版本校验

HIBYTE是高位，副版本。

LOBYTE是低位，主版本。



# 什么是socket

将底层的复杂协议体系，执行流程，进行了封装，封装完的结果就是一个socket。也就是说，socket是我们调用协议进行通信的操作接口。

将复杂的协议过程与编程人员分开，编程人员直接操作一个简单的socket就行了，对于底层协议过程细节，完全不用知道，这样大大方便了编程人员。

socket本质就是一种数据类型，就是一个整数，但是这个数是唯一的，标识着当前的应用程序，协议特点等信息。

typedef UINT\_PTR SOCKET;

typedef \_W64 unsigned int UINT\_PTR, \*PUINT\_PTR;

# socket()详解

作用：创建socket。

SOCKET socket(int af, int type, int protocol);

## 参数1：int af

地址的类型。

AF\_INET，ipv4，4字节，32位的地址，就是无符号int类型的范围0-4294967295。

AF\_INET6，ipv6，16字节，128位地址。

## 参数2：int type

套接字的类型。

SOCK\_STREAM，一种套接字类型，提供带有OOB数据传输机制的顺序，可靠，双向，基于连接的字节流。此套接字类型使用传输控制协议TCP作为Internet地址系列（AF\_INET或AF\_INET6）。

## 参数3：int protocol

协议的类型。

IPPROTO\_TCP，传输控制协议TCP。当af参数为AF\_INET或AF\_INET6且类型参数为SOCK\_STREAM时，这是一个可能的值。

参数3一般可以填写0，系统会自动帮我们选择协议类型。

## 返回值

成功返回可用的socket。

不用的socket一定要销毁，使用closesocket(socket)。

失败返回INVALID\_SOCKET。

可用WSAGetLastError()返回错误码。

## 代码范例



# bind()详解

int bind(SOCKET s, const sockaddr \*addr, int namelen);

作用：给socket绑定端口号与具体地址。

地址：找到我们的电脑。

端口号：找到我们机器上对应的软件，每一种通信的端口号是唯一的，同一个软件可能占用多个端口。

## 参数1：SOCKET s

上一个函数创建的socket，绑定了协议信息（地址类型，套接字类型，协议类型），咱们bind()函数就是绑定实质的地址和端口号。

## 参数2：const sockaddr \*addr

struct sockaddr

{

u\_short sa\_family;

char sa\_data[14];

}

struct sockaddr\_in

{

short sin\_family;

ushort sin\_port;

struct in\_addr sin\_addr;

char sin\_zero[8];

}

端口号：理论上只要在这个范围0-65535都可以；实际上，介于0-1023，为系统保留占用端口号；所以我们不能填写这个范围的值。

如何查看自己要用的端口有没有被占用呢？？？

查看被使用的所有端口：netstat -ano

检查我们要使用的端口号是否被占用了：netstat -ano|findstr “12345”，如果被使用了就会显示使用的程序，未被使用就什么也不显示。

## 参数3：int namelen

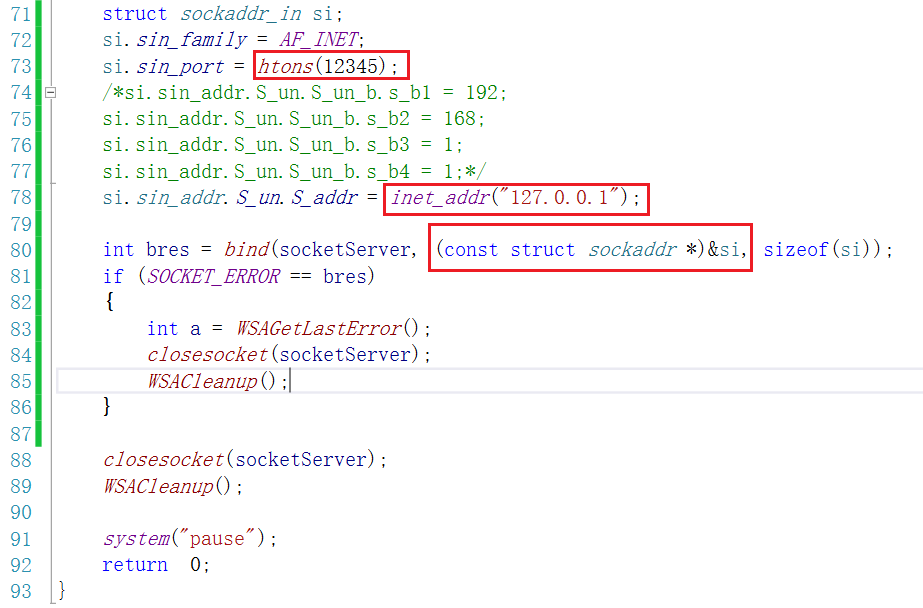
sizeof(sockaddr\_in)

## 返回值

成功返回0。

失败返回SOCKET\_ERROR。

## 代码范例



# listen()

int WSAAPI listen(SOCKET s, int backlog);

作用：将套接字置于正在侦听传入连接的状态。

## 参数1：SOCKET s

服务器端的socket，也就是socket()函数创建的。

## 参数2：int backlog

挂起连接队列的最大长度。就是说，比如有100个用户连接请求，但是系统一次只能处理20个，那么剩下80个不能不理人家，所以系统就创建个队列记录这些暂时不能处理，一会处理的连接请求，依先后顺序处理，那这个队列到底有多大？就是这个参数设置的，比如2，那么就允许两个新连接排队。这个肯定不能无限大，内存不够用。

一般填写这个参数为SOMAXCONN，作用是让系统自动选择最合适的个数。

## WSAAPI

调用约定。

这是系统管的，跟咱们没关系。

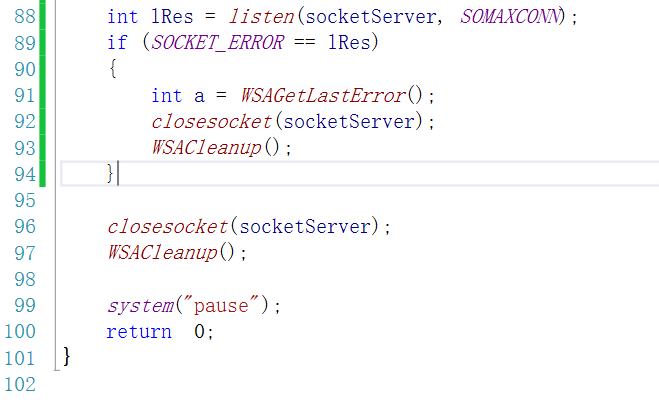
用来决定：函数名字的编译方式；参数的入栈顺序；函数的调用时间。

## 返回值

成功返回0。

失败返回SOCKET\_ERROR。

## 代码范例



# accept()

SOCKET WSAAPI accept(SOCKET s, sockaddr \*addr, int \*addrlen);

创建客户端SOCKET。

accept()函数允许在套接字上进行传入连接尝试。

listen()函数监听客户端的连接，accept()将客户端的信息绑定到一个socket上，也就是给客户端创建一个socket，通过返回值返回给我们客户端的socket。

一次只能创建一个，有几个客户端连接，就要调用几次。

## 参数1：SOCKET s

我们上面创建的自己的socket。

socket先处于监听状态，然后来的连接都由这个管理，我们取客户端的信息，就是通过这个我们自己的socket。

## 参数2：sockaddr \*addr

客户端的地址端口信息结构体。

跟bind()的第二个参数一样。

系统帮我们监视客户端的动态，肯定会记录客户端的信息，也就是IP地址，端口号，并通过这个结构体记录。

参数2，3也能都设置成NULL，那就是不直接得到客户端的地址和端口号。

此时可以通过函数得到客户端的信息：getpeername()。

得到本地服务器信息：getsockname()。

## 参数3：int \*addrlen

参数2的大小。

## 返回值

成功，返回值就是给客户端包好的socket，与客户端连接就靠这个。

失败，返回INVALID\_SOCKET。

## 代码范例



# recv()

int recv(SOCKET s, char \*buf, int len, int flags);

作用：得到指定客户端/服务器（参数1）发来的消息。

原理：recv()的本事是复制。数据的接收都是由协议本身做的，也就是SOCKET底层做的，系统会有一段缓冲区，存储接收到的数据。咱们外面调用recv()的作用，就是通过socket找到这个缓冲区，并把数据复制进咱们的参数2中。只要客户端发来消息，不管recv()有没有调用，系统一定会接收客户端发来的消息的，并在接收到的消息放在系统的缓冲区中。自己想要的时候，就调用recv()将客户端发送来的数据从系统的缓冲区中取出来。

## 参数1：SOCKET s

目标的SOCKET，如果是服务器，一定是接收客户端的SOCKET；如果是客户端，一定是接收服务器的SOCKET。

客户端/服务器的socket，每个客户端/服务器对应唯一的socket。

服务器可以连接很多个客户端，每个客户端都对应一个socket，recv()参数1中放哪个socket，就接收哪个发送过来的数据。recv()是一对一的。

## 参数2：char \*buf

存放接收到的消息。是个字符数组。

这个一般不高于1500个字节。

网络传输的最大单元，1500字节，也就是客户端发过来的数据，一次最大就是1500字节，这是协议规定的，这个数值也是根据很多情况，总结出来的最优值。

## 参数3：int len

想要读取的字节个数。一般是参数2的字节数-1，把\0字符串结尾留出来。

## 参数4：int flags

数据的读取方式。

一般情况下，填写0。

正常逻辑下，我们从系统缓冲区把数据读到我们的buf中，读到buf中后，系统缓冲区中被读出来的数据应该被删除了，不然就浪费空间，毕竟，通信时间长了，空间不够用。我们将缓冲区的数据读到我们自己的buf中，根据需要处理相应的数据，这是我们可控的。系统缓冲区的数据，咱们操作不了。

读出来就删的话，由很多好处：1、系统缓冲区读到的数据，比我们buf中的数据多，那么我们读出来的，系统删除，从而我们就可以依次地把所有数据读完了。2、可以计数收到了多少字节，返回值就是本次读出来的数据。

MSG\_PEEK：窥视传入的数据，数据将被复制到缓冲区中，但不会从输入队列中删除。不建议使用，原因是：读数据不行，无法计数。

MSG\_OOB：带外数据，就是传输一段数据，在外带一个额外的特殊数据。不建议使用，原因是，既然是两种数据，那咱们就send()两次，另一方recv()两次就行了。

MSG\_WAITALL：直到系统缓冲区中的字节数满足参数3所请求的字节数，才开始读取。

## 返回值

读出来字节数的大小。

读没了，怎么办？？？

返回值为0，代表客户端下线，需要释放客户端socket。

recv()也是阻塞的，跟accept()一样，从TCP协议缓冲区读数据，客户端处于连接状态，如果客户端没有发送数据过来，recv()就一直等着。

执行失败，返回SOCKET\_ERROR。WSAGetLastError()得到错误码，根据错误码信息做相应处理：重启、等待、不用理会。

## 代码范例



# send()

int WSAAPI send(SOCKET s, const char \*buf, int len, int flags)

作用：向目标发送数据。

本质：send()将我们的数据（参数2）复制进系统的协议发送缓冲区，计算机伺机发送出去。

## 参数1：SOCKET s

目标的socket，每个客户端对应唯一的socket。

## 参数2：const char \*buf

给对方发送的字节串，这个参数不要超过1500字节。发送的时候，协议要进行包装，加上协议信息，也叫协议头，或者叫包头。打包的过程就是给数据加上目标端的信息。打完包之后，不要超过1500字节。这个大小不同的协议不一样，链路层14字节，ip头20字节，tcp头20字节，数据结尾还要有状态确认，加起来也几十个字节，所以实际上咱们的数据位，不能写1500字节，最多1400字节。

超过1500字节，怎么办？？？系统会将数据分片处理，分次发送。这样处理造成的结果是：1、系统要给咱们分包再打包，再发送，客户端接收到之后，还要拆包，组合数据，从而增加了系统的工作，降低了效率。2、有的协议，就把分片后的二包直接丢了。

## 参数3：int len

要决定发送字节的个数。

## 参数4：int flags

数据的发送方式。

一般情况下，填写0。

MSG\_OOB：带外数据，不建议使用。

MSG\_DONTROUTE：指定数据不受路由限制，Windows套接字服务提供程序可以选择忽略此标志。

## 返回值

执行成功，返回写入的字节数（len）。

执行失败，返回SOCKET\_ERROR。

## 代码范例



# 服务端

1、打开网络库

2、校验版本

3、创建SOCKET

4、绑定地址与端口

5、开始监听

6、创建客户端SOCKET/接收连接

7、与客户端收发消息

# 客户端

1、打开网络库

2、校验版本

3、创建SOCKET

4、连接到服务器

5、与服务器收发消息

# connect()

int WSAAPI connect(SOCKET s, const sockaddr \*name, int namelen)

作用：连接服务器并把服务器信息与服务器SOCKET绑定到一起。

## 参数1：SOCKET s

服务器socket。

## 参数2：const sockaddr \*name

服务器ip地址端口号结构体。

## 参数3：int namelen

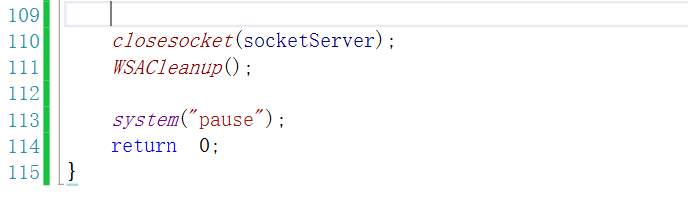
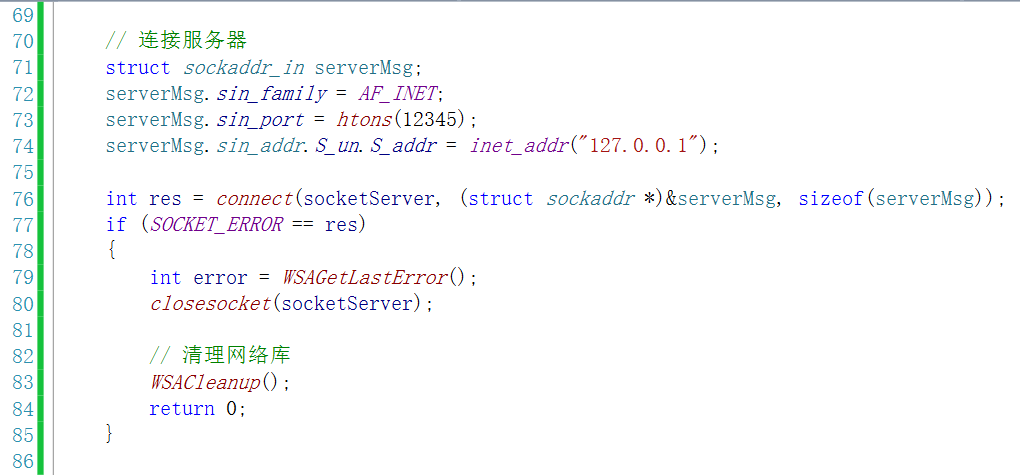
参数2结构体大小。

## 返回值

成功，返回0。

失败，返回SOCKET\_ERROR。

## 客户端connect()代码范例



# 完善C/S模型

在客户端和服务端都加上循环，将收发放到循环里面处理。

问题：

由于accept()，recv()是阻塞的，做其中一件事，另外一件事就做不了，所以假设有多个客户端的情况下，当前的模型是，我先等，我不管将来是什么请求，我先等。造成的结果是：咱们在这等收消息recv()，结果来了个连接请求，那就无法处理，连接只能accept()。另外，我们等的socket可能没有发请求，那我们就完了，只能一直等待下去。从而这个服务器，就废掉了。

分析：

我们直接跟系统要，要什么？？？要有请求的socket，哪个有请求，就给我找哪个。结果：得到连接请求，我们就直接accept()，得到是发来了消息，我们就recv()。

得到：

select模型，它把请求的套接字给我们挑选出来，我们直接就去处理这些套接字。

select就是处理accept()和recv()阻塞问题的。

send()本身也是阻塞的，不是长阻塞，是短阻塞。

# select模型

解决基本C/S模型中，accept()和recv()傻等的问题。

执行阻塞：send()，recv()，accept()。在执行复制的过程中都是阻塞的。

注意：select模型是解决傻等问题的，不解决这几个函数本身的阻塞问题。

实现多个客户端连接，与多个客户端分别通信。

用于服务器，客户端就不用这个，因为客户端只有一个socket（服务器的socket）。客户端recv()时，也不能send()？？？，此时只需要单独创建一个线程，recv()放线程里，就可以了。

服务端select模型代码逻辑：

1. 打开网络库。
2. 校验版本。
3. 创建SOCKET。
4. 绑定地址与端口。
5. 开始监听。
6. select。

select模型处理逻辑:

1. 每个客户端都有socket，服务器也有自己的socket，将所有的socket装进一个数据结构里，数组中。
2. 通过select()函数遍历1中socket数组，当某个socket有响应，select就会通过其参数/返回值反馈出来。
3. 我们就做相应的处理。如果监测到的是服务器socket，那就是有客户端连接，调用accept()。如果监测到的是客户端socket，那就是客户端请求通信，send()或recv()。

# fd\_set

typedef struct fd\_set

{

u\_int fd\_count; /\* how many are SET? \*/

SOCKET fd\_array[FD\_SETSIZE]; /\* an array of SOCKETs \*/

} fd\_set;

FD\_SETSIZE默认为64，大家可以在Winsock2.h前声明这个宏，给它更大的值。

四个操作fd\_set的参数宏：

FD\_ZERO：将fd\_set清零。

FD\_SET：向集合中添加一个socket。当数量不足64，并且不存在的时候。

FD\_CLR：删除集合中指定的socket。删除之后，需要手动释放closesocket()。

FD\_ISSET：判断一个socket是否在集合中，不在返回0，在返回非0。

# select()

int WSAAPI select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, const timeval \*timeout)

作用：监视socket集合，如果某个socket发生事件（连接或收发数据），通过返回值以及参数告诉我们。然后根据socket类型进行分类处理，类型就两种一个是服务器socket（调用accept），一个是客户端socket（调用recv()或send()）。

## 参数1：int nfds

忽略，一般填写0。

这个参数仅仅是为了兼容Berkeley sockets。

## 参数2：fd\_set \*readfds

检查是否有可读的socket，检查是否有消息recv()，即客户端发来消息了，该socket就会被设置。

初始化所有的socket，通过select投放给系统，系统将有事件发生的socket再复制回来，调用后，这个参数就只剩下有请求的socket。

这个参数是双向的。

## 参数3：fd\_set \*writefds

检查是否有可写的socket，就是可以给哪些客户端套接字发消息，即send()，只要连接成功建立起来了，那该客户端套接字就是可写的。

初始化所有的socket，通过select投放给系统，系统将可写的socket再复制回来，调用后，这个参数就是装着可以被send()数据的客户端socket。

一般我们就直接send()，所以这个参数在逻辑上，用的不是很多。

## 参数4：fd\_set \*exceptfds

检查套接字上的异常错误。将有异常错误的套接字重新装进来，反馈给我们。

得到异常套接字上的具体错误码：getsockopt(socket, SOL\_SOCKET, SO\_ERROR, buf, buflen)。

## 参数5：const timeval \*timeout

最大等待时间。

比如，当客户端没有请求时，那么select()可以等一会，一段时间过后，还没有，就继续执行select()下面的语句，如果有了，就立即执行下面的语句。

TIMEVAL：两个成员，tv\_sec：秒，tv\_usec：微秒。

0 0：非阻塞状态，不用等待，立刻返回。

3 4：那就在无客户端响应的情况下等待3秒4微秒。

NULL：select完全阻塞，死等，直到客户端有反应，我才继续。

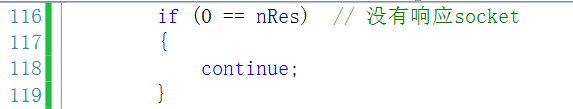
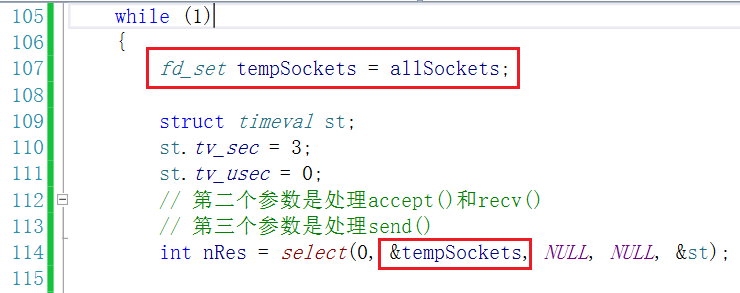
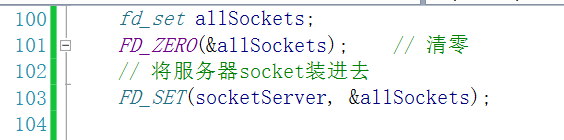
## 返回值

返回0：客户端在等待时间内没有反应，continue，继续下一次等待。

返回值>0：有客户端请求交流。

返回SOCKET\_ERROR：发生错误了，得到错误码WSAGetLastError()。

## 代码范例



# select()处理参数3

可写的socket集合。

哪些客户端可以被send()消息了。



# select()处理参数4

异常的socket集合。



# select模型总结

# 释放socket集合



# 控制台关闭事件处理

在主函数中投递一个监视。



# Windows消息机制和事件机制

Windows处理用户行为的两种方式：消息机制，事件机制。

## 消息机制

核心是消息队列。

处理过程：所有的用户操作，比如点鼠标，摁键盘，点软件上的按钮…等等，所有操作均依此按顺序被记录，装进一个队列。

特点：消息队列由系统维护，咱们做的操作，然后把消息取出来，分类处理。有先后顺序。

## 事件机制

核心是事件集合。

处理过程：根据需求，我们为用户的特定操作绑定一个事件，事件由我们自己调用API创建，需要多少创建多少。将事件投递给系统，系统就帮咱们监视着，所以不能无限创建，太多了系统运行会卡。

如果操作发生了，比如用户点击鼠标，那么对应的事件就会被置成有信号，我们直接获取到有信号的事件，然后处理。

特点：所有事件都是咱们自己定义的，系统只是帮咱们置有无信号，所以我们自己掌管定义。无序的。

# 事件选择模型的处理逻辑

整体逻辑跟select差不多，你看这个叫WSAEventSelect()，名字上看，就是select的进化版。

第一步：创建一个事件对象（变量），WSACreateEvent()。

第二步：为每一个事件对象绑定一个socket以及操作，accept()、read()、close()，并投递给系统。

第三步：查看事件是否有信号，WSAWaitForMultipleEvents()。

第四步：有信号的话就分类处理，WSAEnumNetworkEvents()。

# WSACreateEvent()

WSAEVENT WSAAPI WSACreateEvent()。

创建事件对象。

执行成功，返回一个事件。

执行失败，返回WSA\_INVALID\_EVENT。WSAGetLastError()获取错误码。执行失败，需要关闭，closesocket()，WSACleanup()。

HANDLE，句柄：

ID

内核对象：由系统在内核申请，由操作系统访问，我们不能定位其内容，也不能修改，创建内核对象时通过调用函数实现，不用的时候要调用函数释放。

内核对象（Kernel Objects）有哪些：socket。

# WSACloseEvent()

BOOL WSAAPI WSACloseEvent(WSAEVENT hEvent)。

关闭/释放事件句柄。不用就要释放。

# WSAResetEvent()

BOOL WSAAPI WSAResetEvent(WSAEVENT hEvent)。

重置WSAEventSelect()函数使用的事件对象状态的正确方法，是将事件对象的句柄传递给hEventObject参数中的WSAEnumNetworkEvents()函数。这将重置事件对象并以原子方式调整套接字上活动FD事件的状态。

# WSASetEvent()

BOOL WSAAPI WSASetEvent(WSAEVENT hEvent)。

将指定事件主动置成有信号。

# WSAEventSelect()

int WSAAPI WSAEventSelect(SOCKET s, WSAEVENT hEventObject, long lNetworkEvents)。

绑定并投递。

功能：给事件绑定上socket与操作码，并投递给操作系统。

## 参数1 ：SOCKET s

被绑定的socket，最终，每个socket都会被绑定一个事件。

## 参数2：WSAEVENT hEventObject

事件对象。逻辑上，就是将参数1和参数2绑定在一起。

## 参数3：long lNetworkEvents

具体事件。

FD\_ACCEPT，有客户端连接，与服务器socket绑定。

FD\_READ，有客户端发来消息，与客户端socket绑定，可多个属性并列（用|）。

FD\_CLOSE，客户端下线，与客户端socket绑定，包含强制下线，正常下线。

FD\_WRITE，可以给客户端发送消息，与客户端socket绑定，会在accept()后立即主动产生该信号，可以说明，客户端连接成功，即可随时send()。

FD\_CONNECT，客户端一方，给服务器绑定这个。

0，取消事件监视。

FD\_OOB，带外数据，不建议使用。

FD\_QOS，套接字服务质量状态发送变化消息通知。

FD\_GROUP\_QOS。

## 返回值

执行成功，返回0。

执行失败，返回SOCKET\_ERROR，WSACloseEvent(wsaEvent)、closesocket(socketServer)、WSACleanup()。

# WSAWaitForMultipleEvents()

DWORD WSAAPI WSAWaitForMultipleEvents(DWORD cEvents, const WSAEVENT \*lphEvents, BOOL fWaitAll, DWORD dwTimeout, BOOL fAlertable)

询问事件。

作用：获取发生信号的事件。

## 参数1：DWORD cEvents

事件个数，定义事件列表（数组）个数是，WSA\_MAXIMUM\_WAIT\_EVENTS(64)，该函数参数1最大为64，也可以变的比64更大。

## 参数2：const WSAEVENT \*lphEvents

事件列表。

## 参数3：BOOL fWaitAll

事件等待方式。

TRUE，所有事件都产生信号时，才返回。

FALSE，任何一个事件产生信号时，就立即返回。返回值减去WSA\_WAIT\_EVENT\_0，表示事件对象的索引，其状态导致函数返回。如果在调用期间发出多个事件对象的信号，则这是信号事件对象的数组索引，其中返回所有信号事件对象的索引值中最小的。

## 参数4：DWORD dwTimeout

超时间隔。以毫秒为单位。

等待dwTimeout时间，超时就返回WSA\_WAIT\_TIMEOUT。超时的话就continue继续下一次等待。

如果该参数填写0，表示检查事件对象的状态并立即返回，不管有没有信号。

如果该参数填写WSA\_INFINITE，等待直到事件发生。

## 参数5：BOOL fAlertable

TRUE，重叠I/O模型使用。

FALSE，此事件模型填写FALSE。

## 返回值

返回数组下标的运算值。

如果参数3为TRUE，所有事件均有信号。

如果参数3为FALSE，返回值减去WSA\_WAIT\_EVENT\_0 == 数组中事件的下标。

WSA\_WAIT\_IO\_COMPLETION，参数5为TRUE，才会返回这个值。

WSA\_WAIT\_TIMEOUT，超时，continue即可。

# WSAEnumNetworkEvents()

Int WSAAPI WSAEnumNetworkEvents(SOCKET s, WSAEVENT hEventObject, LPWSANETWORKEVENTS lpNetworkEvents)

列举事件。

获取事件的类型，并将事件上的信号量重置。

accept()、recv()、close()等等。

## 参数1：SOCKET s

对应的SOCKET。

## 参数2：WSAEVENT hEventObject

对应的事件。

## 参数3：LPWSANETWORKEVENTS lpNetworkEvents

该参数是一个结构体指针。

触发的事件类型在这里装着。

Struct \_WSANETWORKEVENTS

{

long lNetworkEvents;

int iErrorCode[FD\_MAX\_EVENTS];

}

成员1：具体操作，一个信号可能包含两个消息，以按位或的形式存在。

成员2：错误码数组，FD\_ACCEPT事件错误码在FD\_ACCEPT\_BIT下标里，没有错误，对应的就是0。

## 返回值

执行成功，返回0。

执行失败，返回SOCKET\_ERROR，WSAGetLastError()得到错误码。

# 事件分类处理逻辑

# switch处理以及else if处理以及if处理的区别

else if是多选一。

# 有序处理优化

最开始就一个一个检测，不绝对公平，只是相对公平，相对有序。

让大家在一轮循环中都能得到处理。

但是并不完全解决顺序问题，只是达到相对公平。

所以，事件选择模型不能用于大用户，多访问。

# 增加处理事件的数量

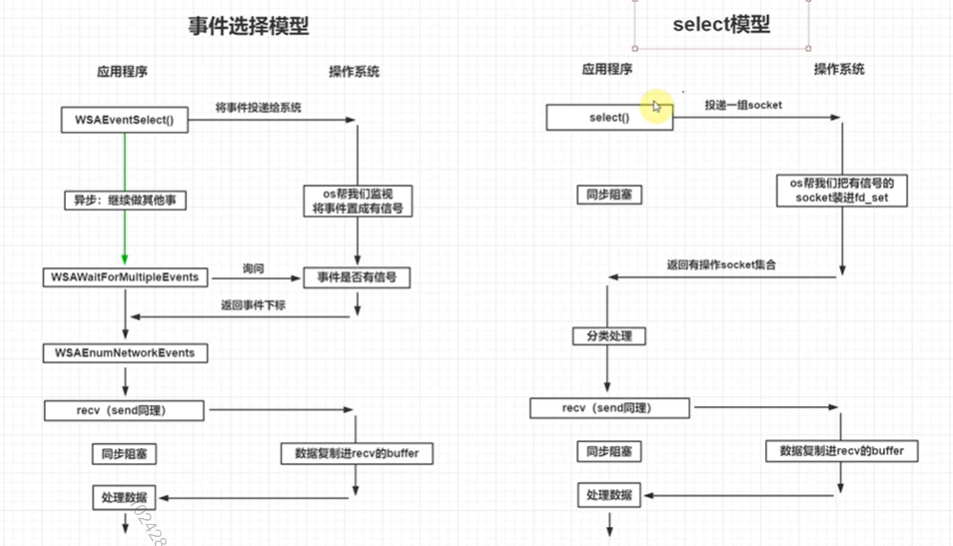
一个一个来，一个大数组就行了。

一组一组来，单线程，一组一组按顺序处理就好了；创建多个线程，每个线程处理一个事件表，最大64。

# 释放

释放socket和对应的事件。

# select模型与事件选择模型对比



# 异步选择模型

## 逻辑

核心：消息队列。

操作系统为每个窗口创建一个消息队列并维护。所以我们想要使用消息队列，那就要创建一个窗口。

第一步：将我们的socket，操作绑定在一个消息上，并投递给操作系统，WSAAsyncSelect()。

第二步：取消息分类处理，如果有操作了，那就会得到指定的消息。

该模型只能用于windows，学习该模型的处理思想。

## 窗口

第一步：创建窗口结构体，WNDCLASSEX。

第二步：注册窗口结构体，RegisterClassEx。

第三步：创建窗口，CreateWindowEx。

第四步：显示窗口，ShowWindow。

第五步：消息循环，GetMessage，TranslateMessage，DispatchMessage。

第六步：回调函数。

# WSAAsyncSelect()

int WSAAsyncSelect(SOCKET s, HWND hWnd, u\_int wMsg, long lEvent);

功能：将socket与消息绑定在一起，并投递给操作系统。

## 参数1：SOCKET s

socket。

## 参数2：HWND hWnd

窗口句柄。

绑定到哪个窗口上，本质就是窗口的ID。

## 参数3：u\_int wMsg

消息编号。

自定义消息，本质就是一个数。

## 参数4：long lEvent

消息类型。

FD\_ACCEPT：有客户端连接，与服务器socket绑定。

FD\_WRITE：可以给客户端发消息，与客户端绑定，会在accept()后立即主动产生该信号，可以说明，客户端连接成功，即随时send()。

FD\_READ：有客户端发来消息，与客户端绑定，多个属性并列使用，用|。

FD\_CLOSE：客户端下线，与客户端绑定，包含强制下线，正常下线。

FD\_CONNECT：客户端一方，给服务器绑定这个。

0：取消事件监视。

FD\_OOB：带外数据。

FD\_QOS

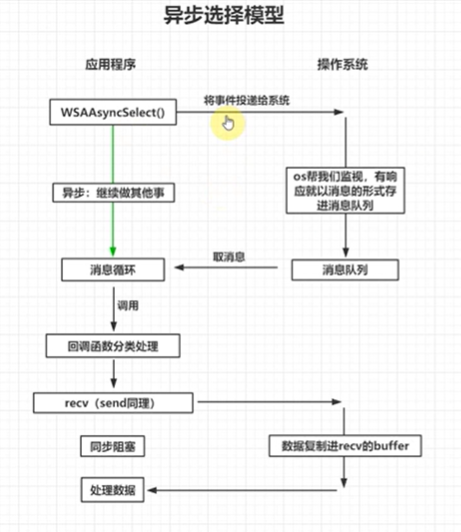
FD\_GROUP\_QOS

## 返回值

执行成功，返回0。

执行失败，返回SOCKET\_ERROR。

# 异步选择模型问题分析



# 重叠IO模型-事件通知

## 重叠IO介绍

重叠IO是windows提供的一种异步读写文件的机制。

作用：正常读写文件（socket本质就是文件操作），如recv()，是阻塞的，等协议缓冲区中的数据全部复制进咱们的buffer里，函数才结束并返回复制的个数，写也是一样，同一时间只能读写一个，其它的全都靠边等。

重叠IO机制读写，将读的指令以及咱们的buffer投递给操作系统，然后函数直接返回，操作系统独立开个线程，将数据复制进咱们的buffer，数据复制期间，咱们就去做其他事情，两不耽误，即读写过程变成了异步，可以同时投递多个读写操作。即将accept()、recv()、send()优化成了异步过程，被AcceptEx()、WSARecv()、WSASend()函数代替了。

本质，是一个结构体，定义一个该结构体的变量与socket绑定。

struct \_WSAOVERLAPPED

{

DWORD Internal;

DWORD InternalHigh;

DWORD Offset;

DWORD OffsetHigh;

WSAEVENT hEvent;

}

前四个成员系统使用，咱们不需要使用。

成员5：事件对象，操作完成时，它会被置成有信号。

使用：重叠IO模型把重叠结构与socket绑定在一起，然后系统以重叠IO机制处理反馈。

重叠IO反馈两种方式：事件通知；完成例程（回调函数）。

## 重叠IO逻辑

事件通知：调用AcceptEx()、WSARecv()、WSASend()投递；被完成的操作，事件信号置成有信号；调用WSAWaitForMultipleEvents()获取事件信号。

回调函数：调用AcceptEx()、WSARecv()、WSASend()投递；完成后自动调用回调函数。

## 事件通知实现逻辑

1. 创建事件数组，socket数组，重叠结构体数组；下标相同的绑定成一组。
2. 创建重叠IO模型使用的socket，WSASocket()。
3. 投递AcceptEx()：
   1. 立即完成：
      1. 对客户端套接字投递：WSARecv()
         1. 立即完成：处理信息；多客户端套接字投递WSARecv()。
         2. 延迟完成：循环等信号。
      2. 根据需求对客户端套接字投递WSASend()：
         1. 立即完成：处理信息；根据需求对客户端套接字投递WSASend()。
         2. 延迟完成：循环等信号。
      3. 对服务器套接字继续投递AcceptEx()，重复上述操作。
   2. 延迟完成：循环等信号。
4. 循环等信号：
   1. 等信号：WSAWaitForMultipleEvents()。
   2. 有信号：
      1. 获取重叠结构上的信息：WSAGetOverlapperResult()。
      2. 客户端退出：删除服务端的信息。
      3. 接收连接：投递AcceptEx()。
      4. 接收信息：处理消息；对客户端套接字投递WSARecv()。
      5. 发送信息：根据需求对客户端套接字投递WSASend()。

## WSASocket()

SOCKET WSAAPI WSASocket(int af, int type, int protocol, LPWSAPROTOCOL\_INFOA lpProtocolInfo, GROUP g, DWORD dwFlags);

作用：创建一个用于异步操作的socket。

## 参数1：int af

地址的类型。AF\_INET。

## 参数2：int type

套接字类型。SOCK\_STREAM。

## 参数3：int protocol

协议的类型。IPPROTO\_TCP。

## 参数4：LPWSAPROTOCOL\_INFOA lpProtocolInfo

设置套接字详细的属性。

指向WSAPROTOCOL\_INFO结构的指针。比如，发送数据是否需要连接、是否保证数据完整达到、参数3填0，那么匹配哪个协议、设置传输接收字节数、设置套接字权限、还有很多保留字段，供以后扩展使用。

不适用，填写NULL。

## 参数5：GROUP g

一组socket的组ID，大概是想一次操作多个socket。

保留，暂时没用，填0。

## 参数6：DWORD dwFlags

指定套接字的属性。

WSA\_FLAG\_OVERLAPPED：创建一个供重叠IO模型使用的socket。

WSA\_FLAG\_MULTIPOINT\_C\_ROOT、WSA\_FLAG\_MULTIPOINT\_C\_LEAF、WSA\_FLAG\_MULTIPOINT\_D\_ROOT、WSA\_FLAG\_MULTIPOINT\_D\_ROOT：用于多播协议。

WSA\_FLAG\_ACCESS\_SYSTEM\_SECURITY：设置套接字权限，配合参数4使用。意义：可以对套接字send()设置权限，那么在send()时，就会触发相关的权限设置。

WSA\_FLAG\_NO\_HANDLE\_INHERIT：套接字不可被继承。在多线程开发中，子线程会继承父线程的socket，即主线程创建了一个socket，那么子线程有两种使用方式：第一种是直接用父类的，第二种是把父类的这个socket复制一份，自己用。

## 返回值

执行成功，返回可用的socket。

执行失败，返回INVALID\_SOCKET。

# AcceptEx()

BOOL AcceptEx(SOCKET sListenSocket, SOCKET sAcceptSocket, PVOID lpOutputBuffer, DWORD dwReceiveDataLength, DWORD dwLocalAddressLength, DWORD dwRemoteAddressLength, LPDWORD lpdwBytesReceived, LPOVERLAPPED lpOverlapped);

功能：投递服务器socket，异步接收连接。

## 参数1：SOCKET sListenSocket

服务器socket。

## 参数2：SOCKET sAcceptSocket

连接服务器的客户端的socket。

要调用WSASocket()亲自创建。

AcceptEx()将客户端的IP端口号绑定在这个socket上。

## 参数3：PVOID lpOutputBuffer

缓冲区的指针，接收在新连接上发送的第一个数据：客户端第一次send()，由这个函数接收。第二次以后就由WSARecv()接收。

## 参数4：DWORD dwReceiveDataLength

填写0，取消了参数3的功能。

填写参数3的大小。

## 参数5：DWORD dwLocalAddressLength

为本地地址信息保留的字节数。此值必须至少比使用的传输协议的最大地址长度长16个字节。

## 参数6：DWORD dwRemoteAddressLength

为远程地址信息保留的字节数。此值必须至少比使用的传输协议的最大地址长度长16个字节。

## 参数7：LPDWORD lpdwBytesReceived

该函数可以接收第一次客户端发来的信息，如果这个信息刚好是调用的时候接收到了，也即立即接收到了（客户端连接的同时发送了信息），这时候装着接收到的字节数。

可以填写NULL，不想获得这个字节数。

也可以填写DWORD变量的地址。

## 参数8：LPOVERLAPPED lpOverlapped

重叠结构。

## 返回值

返回值TRUE：立即完成返回，刚执行就已经有客户端到了。

返回值FALSE：出错。int a = WSAGetLastError()，a == ERROR\_IO\_PENDING（异步等待，客户端还没来）。

# WSARecv()

int WSAAPI WSARecv(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount, LPDWORD lpNumberOfBytesRecvd, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped, LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine);

作用：投递异步接收信息。

## 参数1：SOCKET s

客户端socket。

## 参数2：LPWSABUF lpBuffers

存储接收到的信息。

struct \_WSABUF

{

ULONG len;

CHAR \*buf;

}

成员1：字节数。

成员2：指向字符数组的指针。

## 参数3：DWORD dwBufferCount

WSABUF的个数。

## 参数4：LPDWORD lpNumberOfBytesRecvd

接收成功，这里装的是成功接收到的字节数。

参数6重叠结构不为NULL时，此参数可以设置成NULL。

## 参数5：LPDWORD lpFlags

指向用于修改WSARecv()函数调用行为的标志的指针。

MSG\_PEEK：将协议缓冲区中的数据复制出来，但是不删除。

MSG\_OOB：带外数据。

MSG\_PUSH\_IMMEDIATE：通知传送尽快完成。比如，传输10M数据，一次只能传1M，这个包要拆分成10份发送，第一份发送中，后面9份要排队等着，指定了这个标记，那么表示快点发送，不要等了，那么没被发送的包就被舍弃了，从而造成了数据发送失败。该参数不建议用于多数据发送。提示系统尽快处理，所以能减少一定的延迟。

MSG\_WAITALL：呼叫者提供的缓冲区已满；连接已关闭；请求已取消或发生错误。

MSG\_PARTIAL：传出的。表示咱们此次接收到的数据是客户端发来的一部分，接下来接收下一部分。

## 参数6：LPWSAOVERLAPPED lpOverlapped

重叠结构

## 参数7：LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine

回调函数。

## 返回值

如果立刻完成，返回0。

其他情况返回SOCKET\_ERROR。

# WSAGetOverlappedResult()

BOOL WSAAPI WSAGetOverlappedResult(SOCKET s, LPWSAOVERLAPPED lpOverlapped, LPDWORD lpcbTransfer, BOOL fWait, LPDWORD lpdwFlags);

功能：获取对应socket上的具体情况。

## 参数1：SOCKET s

有信号的socket。

## 参数2：LPWSAOVERLAPPED lpOverlapped

对应的重写结构。

## 参数3：LPDWORD lpcbTransfer

由发送或接收的实际字节数。

0表示客户端下线。

## 参数4：BOOL fWait

仅当重叠操作选择了基于事件的完成通知时，才能将fWait参数设置为TRUE。

选择事件通知，填写TRUE。

## 参数5：LPDWORD lpdwFlags

装WSARecv的参数5 lpflags。

不能是NULL。

## 返回值

执行成功返回TRUE。

执行失败返回FALSE。

# WSASend()

int WSAAPI WSASend(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount, LPDWORD lpNumberOfBytesSent, DWORD dwFlags, LPWSAOVERLAPPED lpOverlapped, LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine);

作用：投递异步发送消息。

## 参数1：SOCKET s

客户端socket。

## 参数2：LPWSABUF lpBuffers

存储接收到信息。

## 参数3：DWORD dwBufferCount

参数2的个数。

## 参数4：LPDWORD lpNumberOfBytesSent

接收成功的话，这里装的是发送的字节数。

参数6重叠结构不为NULL时，此参数可以设置为NULL。

## 参数5：DWORD dwFlags

函数调用行为的标志。

## 参数6：LPWSAOVERLAPPED lpOverlapped

重叠结构。

## 参数7：LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionRoutine

回调函数。

## 返回值

立即发生，返回0。

失败，返回SOCKET\_ERROR。

# 重叠IO模型-完成例程

本质：为我们的socket，重叠结构绑定一个函数，当异步操作完成时，系统异步自动调用这个函数，send()绑定一个，recv()绑定一个，完成就调用各自的函数，自动分类。

事件通知：需要咱们自己调用WSAGetOverlappedResult()，得到结果，然后根据逻辑，自己分类，分类的逻辑很多。

区别：分类方式上，完成例程性能更好，系统自己调用。

# 完成例程回调函数

void CALLBACK 函数名(DWORD dwError, DWORD cbTransferred, LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags)

## 参数1：DWORD dwError

错误码。

## 参数2：DWORD cbTransferred

发送或接收到的字节数。

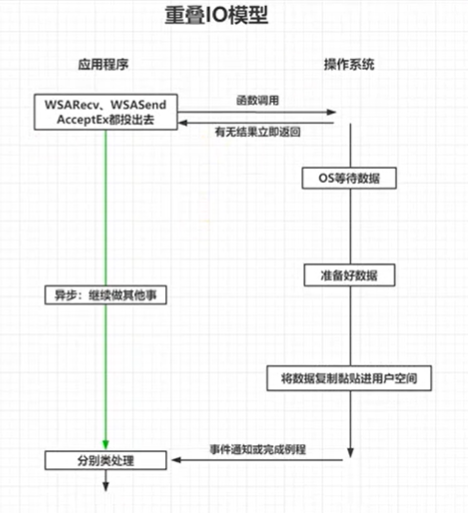
## 参数3：LPWSAOVERLAPPED lpOverlapped

重叠结构。

## 参数4：DWORD dwFlags

函数执行的方式。

# 重叠IO模型总结



# 完成端口模型

完成端口也是windows的一种机制，是在重叠IO模型上的一种优化。

完成端口：模仿消息队列，创建一个通知队列，系统创建，保证有序，不做无用功。创建最佳数量的线程，充分利用CPU的性能。

# 完成端口逻辑

将重叠套接字（客户端+服务器）与一个完成端口（一个类型的变量）绑定在一起。

使用AcceptEx()、WSARecv()、WSASend()投递请求。

当系统异步完成请求，就会把通知存进一个队列，我们就叫它通知队列，该队列由操作系统创建和维护。

完成端口可以理解为这个队列的头，我们通过GetQueuedCompletionStatus()从队列头往外拿，一个一个处理。

1. 创建完成端口：CreateCompletionPort()。
2. 将完成端口与socket绑定：CreateCompletionPort()。
3. 创建指定数目的线程：
   1. CPU核数一样，CreateThread()。
   2. 线程内部：
      1. GetQueuedCompletionStatus()。
      2. 分类处理。
4. 使用AcceptEx()、WSARecv()、WSASend()投递请求。
5. 主线程阻塞。

# CreateIoCompletionPort()

HANDLE CreateIoCompletionPort(HANDLE FileHandle, HANDLE ExistingCompletionPort, ULONG\_PORT CompletionKey, DWORD NumberOfConcurrentThreads)

功能：创建一个完成端口；将完成端口与SOCKET绑定在一起。

## 参数1：HANDLE FileHandle

创建完成端口：INVALID\_HANDLE\_VALUE，此时参数2必须为NULL，参数3忽略，填写0。

绑定：服务器socket。

## 参数2：HANDLE ExistingCompletionPort

创建完成端口：NULL。

绑定：完成端口变量。

## 参数3：ULONG\_PORT CompletionKey

创建完成端口：0。

绑定：再次传递socketServer，也可以传递一个下标；与系统接收到的对应的数据关联在一起。

## 参数4：DWORD NumberOfConcurrentThreads

创建完成端口：允许此端口上最多同时运行的线程数量。填CPU的核数即可，自己获取，GetSystemInfo()。

绑定：填0。

## 返回值

执行成功，当参数2为NULL，返回一个新的完成端口；当参数2不为NULL，返回自己；当参数1为socket，返回与socket绑定后的端口。

执行失败，返回0。

# CreateThread()

HANDLE CreateThread(LPSECURITY\_ATTRIBUTES lpThreadAttributes, SIZE\_T dwStackSize, LPTHREAD\_START\_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadld)

## 参数1：LPSECURITY\_ATTRIBUTES lpThreadAttributes

线程句柄是否能被继承，NULL不能被继承。

指定线程的执行权限，NULL默认的。

## 参数2：SIZE\_T dwStackSize

线程栈大小，填0，系统使用默认大小1M。

字节为单位。

## 参数3：LPTHREAD\_START\_ROUTINE lpStartAddress

线程函数地址，DWORD WINAPI ThreadProc(LPVOID lpParameter)。

参数，就是参数4传递进来的数据。

## 参数4：LPVOID lpParameter

外部给线程传递的数据。

## 参数5：DWORD dwCreationFlags

CREATE\_SUSPENDED，线程创建完挂起状态，调用ResumeThread()启动函数。

STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION，设置了，参数2就是栈保留大小，虚拟内存上栈的大小，1M；未设置，就是栈提交大小，物理内存上栈的大小，4KB。

## 参数6：LPDWORD lpThreadld

线程ID，可以填NULL。

## 返回值

执行成功，返回线程句柄，内核对象，最后要释放（CloseHandle()）。

执行失败，返回NULL。

# GetQueuedCompletionStatus()

BOOL WINAPI GetQueuedCompletionStatus(HANDLE CompletionPort, LPDWORD lpNumberOfBytes, PULONG\_PTR lpCompletionKey, LPOVERLAPPED \*lpOverlapped, DWORD dwMillseconds)

无通知时，线程挂起，不占用CPU的时间。

功能：尝试从指定的IO完成端口处列出IO完成数据包。

## 参数1：HANDLE CompletionPort

完成端口。从主函数传进来。

## 参数2：LPDWORD lpNumberOfBytes

接收或发送的字节数。

## 参数3：PULONG\_PTR lpCompletionKey

完成端口函数的参数3传进来。

## 参数4：LPOVERLAPPED \*lpOverlapped

重叠结构。

## 参数5：DWORD dwMillseconds

等待时间，当没有客户响应的时候，通知队列里什么都没有，咱这里也get不到任何东西，那么等一会，还是一直等。

INFINITE，一直等。

## 返回值

成功返回TRUE。

失败返回FALSE。

# 分类处理

accept()：给socket绑上完成端口；投递recv；投递accept。

recv()：if(0 == lpNumberOfBytes)，客户端退出；打印数据；继续投递recv。

send()：

# 完成端口模型总结

