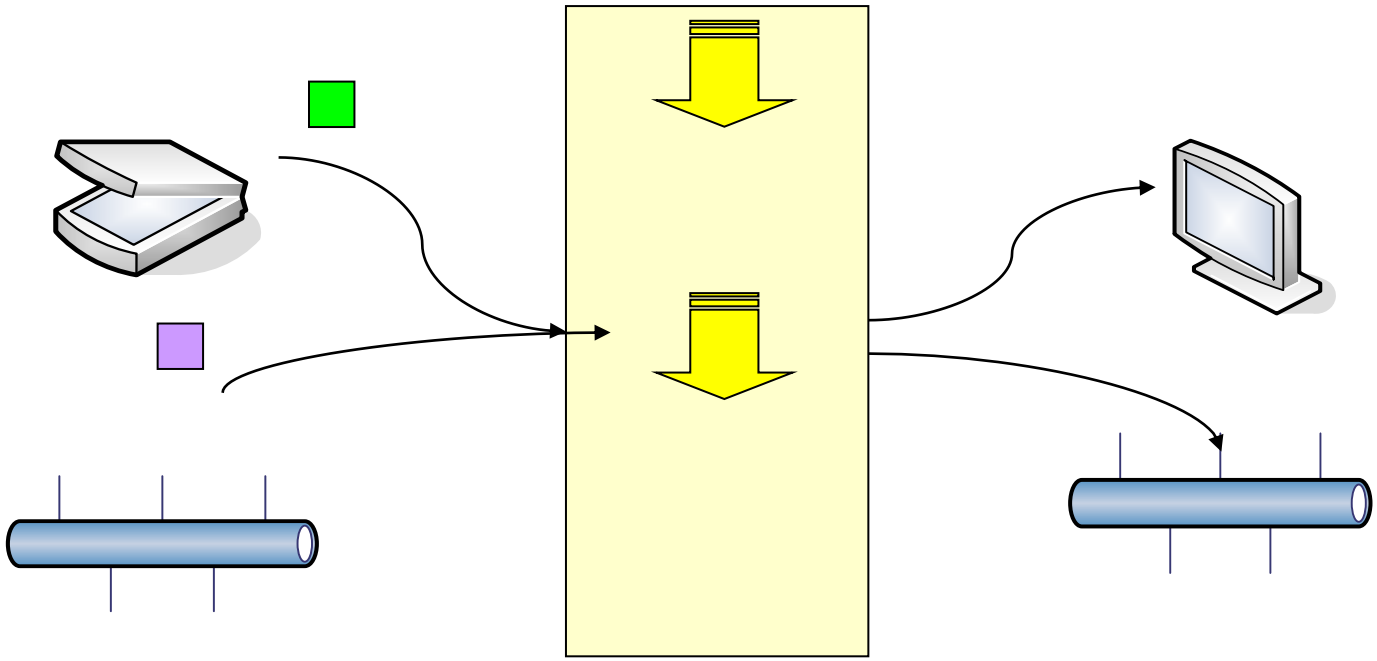
网络程序设计

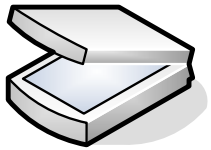
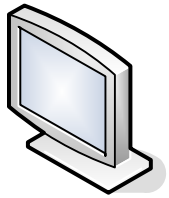
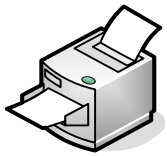
网络通信中的I/O操作



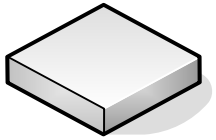
- I/O设备有哪些?
- 如何处理等待?
- 如何处理多个I/O请求?
- 如何编程实现网络通信过程中的I/O操作?

1、I/O设备

- I/O类外设




……

- 存储类型的设备




……

- 网络通信设备




……

1、I/O设备

- **I/O设备之间的差别**
  - **数据率**
  - **传送单位**
  - **数据表示**
  - **……**

很难实现一种统一的、一致的
输入/输出方法！！

问题：网络通讯中，套接字在哪些操作中等待？

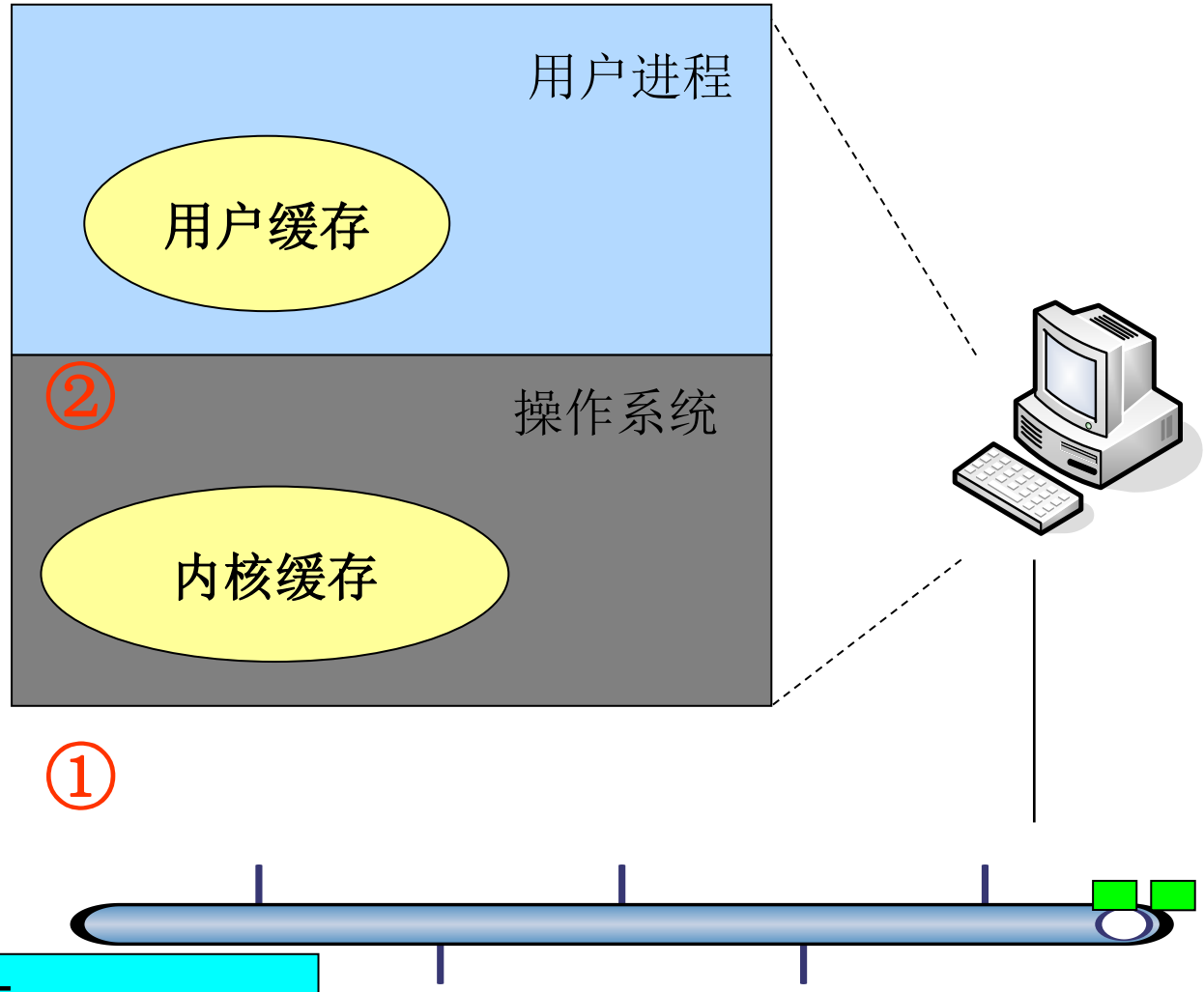
- **Input Operations**：Sleeping until some data arrives the socket receive buffer.

- **Output Operations**：Sleeping until there is enough room in the socket send buffer.

- **Accepting incoming connection**：Sleeping until there is a new connection available.

- **Initiating outgoing connections**：Sleeping until the client receives the ACK of its SYN.

Recvfrom操作



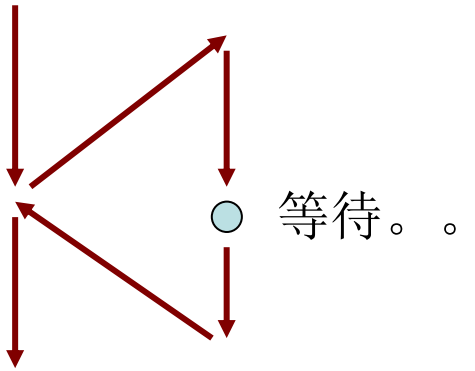
① 等待数据到达

② 等待数据从内核到进程

2、I/O操作

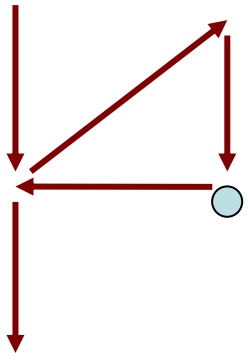
- 同步**I/O (Synchronous I/O)**

必须等待**I/O**操作完成后，控制权才返回给用户进程。



- 异步**I/O (Asynchronous I/O)**

无需等待**I/O**操作完成，就将控制权返回给用户进程。



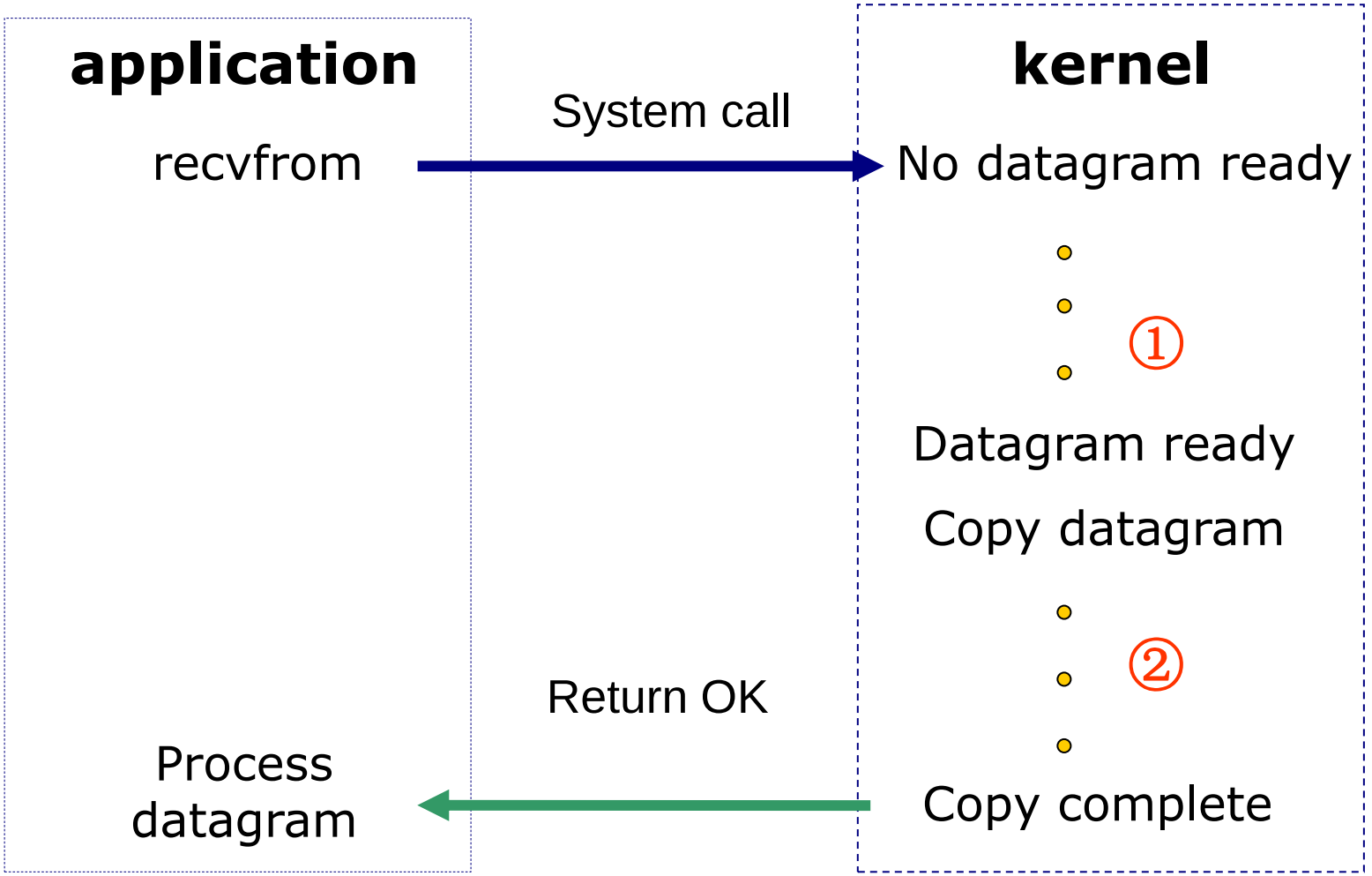
同步和异步

- 同步和异步与消息的通知机制有关
  - 同步：由处理消息者自己去等待消息是否触发
  - 异步：由触发机制来通知处理消息者，然后进行消息的处理。
- 从消息处理机制来看，套接字分为阻塞和非阻塞两种I/O模式
- 同步操作可以是非阻塞的，比如轮询处理I/O事件
- 异步操作也可以是阻塞的，比如I/O复用模型

网络通信7种I/O模型

- 阻塞模型
- 非阻塞模型
- I/O复用模型
- WSAAsynSelect模型
- WSAEventSelect模型
- 重叠I/O模型
- 完成端口模型

（1）阻塞I/O模型(Blocking I/O Model)

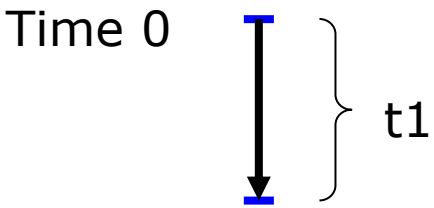


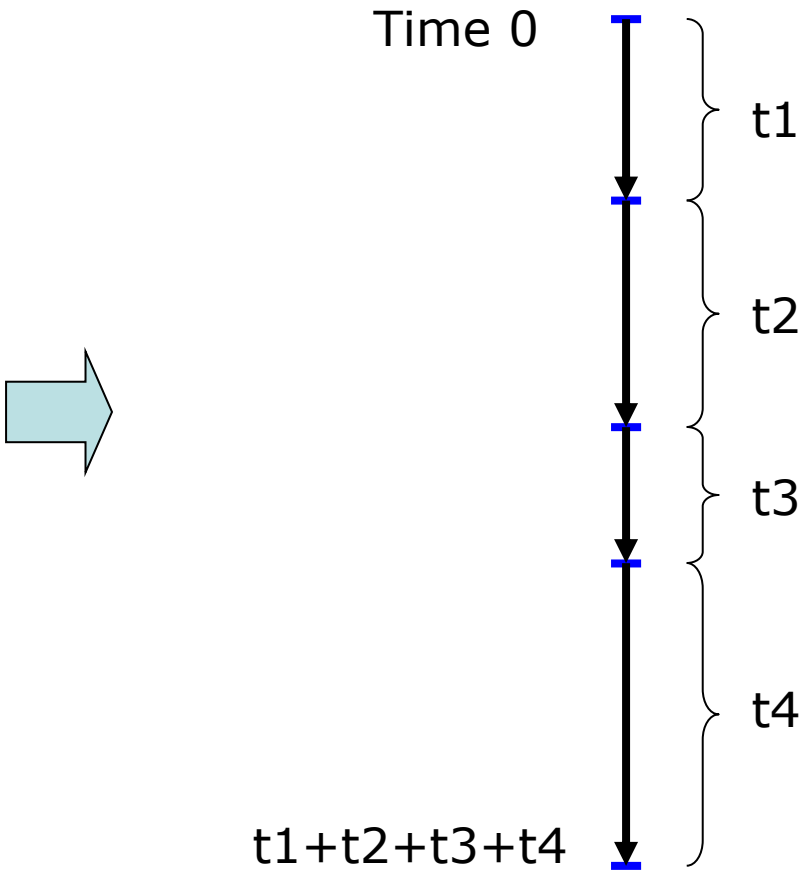
优点：简单、直接

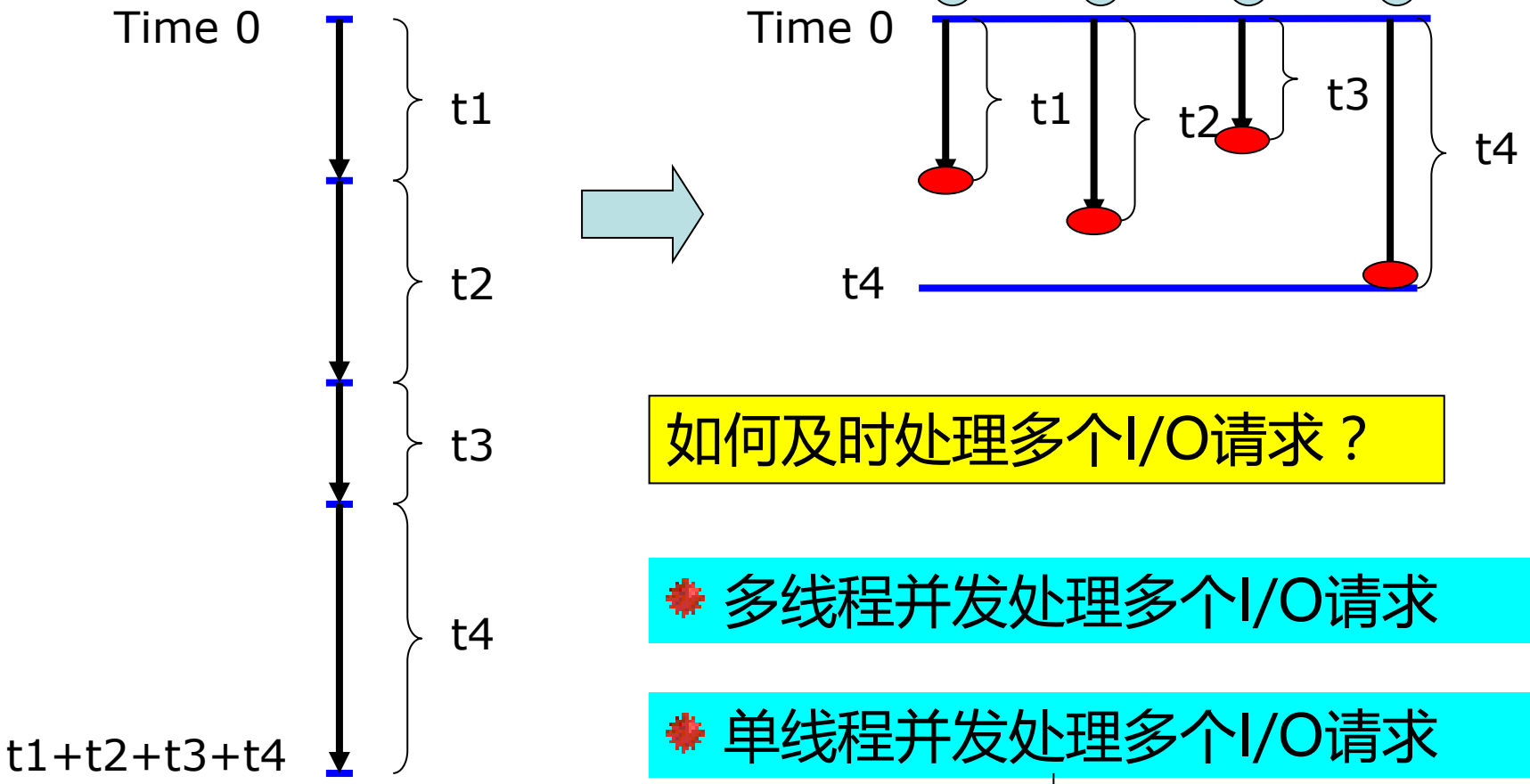
缺点：不适合多个I/O请求

单个套接字的I/O请求

多个套接字的I/O请求







如何及时处理多个I/O请求？

- 多线程并发处理多个I/O请求
- 单线程并发处理多个I/O请求
  - 非阻塞I/O模型
  - I/O复用模型

关键：如何确定网络事件何时发生？

基于阻塞I/O模型的套接字通信服务器示例

服务器设计为循环服务器，每次处理单个客户的请求，当一个客户断开连接后再处理下一个客户的请求。

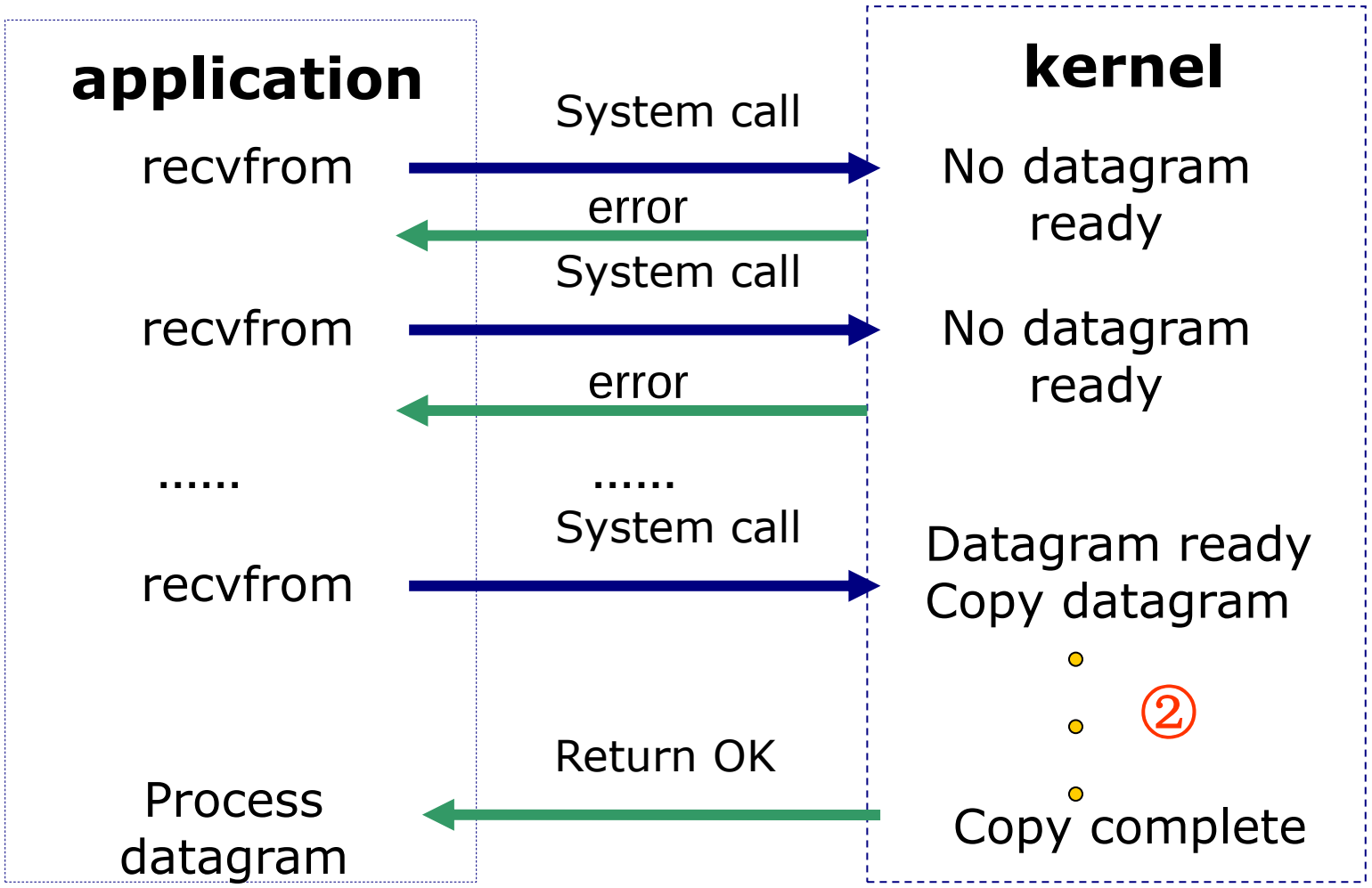
```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>

// 连接到winsock2对应的lib文件： Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT 27015  //默认服务器端口号为27015
int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;
    sockaddr_in addrClient;
    int addrClientlen = sizeof(sockaddr_in);

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }
```

```
// 创建用于监听的套接字
ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (ServerSocket == INVALID_SOCKET)
{
    printf("socket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

// 为套接字绑定地址和端口号
SOCKADDR_IN addrServ;
addrServ.sin_family = AF_INET;
addrServ.sin_port = htons(DEFAULT_PORT);          // 监听端口为DEFAULT_PORT
addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
if (iResult == SOCKET_ERROR)
{
    printf("bind failed with error: %d\n", WSAGetLastError());
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}

// 监听套接字
iResult = listen(ServerSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR)
{
    printf("listen failed !\n");
    closesocket(ServerSocket);
    WSACleanup();
    return -1;
}
printf("TCP server starting\n");
```

```
// 循环等待
while (true)
{
    AcceptSocket = accept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen);
    if (AcceptSocket == INVALID_SOCKET)
    {
        printf("accept failed !\n");
        closesocket(ServerSocket);
        WSACleanup();
        return 1;
    }
    //循环接收数据
    while (TRUE)
    {
        memset(recvbuf, 0, recvbuflen);
        iResult = recv(AcceptSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0)
        {
            //情况1：成功接收到数据
            printf("\nBytes received: %d\n", iResult);
            //处理数据请求
            //....
            //继续接收
            continue;
        }
        else if (iResult == 0)
        {
            //情况2：连接关闭
            printf("Current Connection closing, waiting for the next connection...\n");
            closesocket(AcceptSocket);
            break;
        }
        else
        {
            //情况3：接收发生错误
            printf("recv failed with error: %d\n", WSAGetLastError());
            closesocket(AcceptSocket);
            closesocket(ServerSocket);
            WSACleanup();
            return 1;
        }
    }
}
```

```
    // cleanup
    closesocket(ServerSocket);
    closesocket(AcceptSocket);
    WSACleanup();
    return 0;
}
```

（2）非阻塞I/O模型（Nonblocking I/O Model)



缺点1：多次系统调用消耗资源！

缺点2：处理时机滞后！

非阻塞模型的实现

- 用ioctlsocket()函数将套接字设置为非阻塞模式
- IO命令用FIONBIO
- 以下示例实现了基于非阻塞I/O模型的面向连接通信服务器，该服务器的主要功能是接收客户使用TCP协议发来的数据，打印接收到的数据的字节数。

```c
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>

// 连接到winsock2对应的lib文件： Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT 27015  //默认服务器端口号为27015
int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;
    sockaddr_in addrClient;
    int addrClientlen = sizeof(sockaddr_in);

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }
```

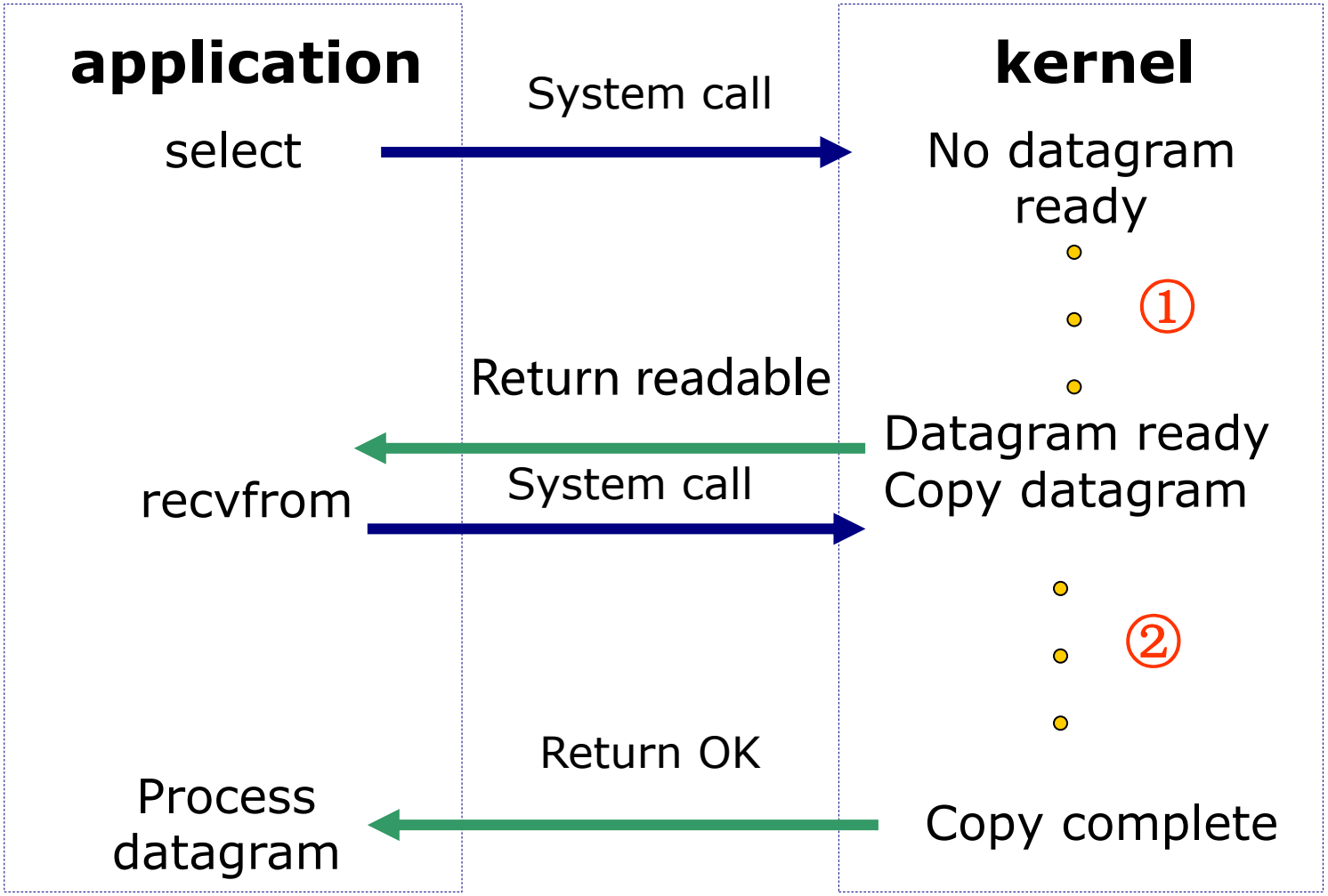
```
// 创建用于监听的套接字
ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (ServerSocket == INVALID_SOCKET)
{
    printf("socket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

// 为套接字绑定地址和端口号
SOCKADDR_IN addrServ;
addrServ.sin_family = AF_INET;
addrServ.sin_port = htons(DEFAULT_PORT);        // 监听端口为DEFAULT_PORT
addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
if (iResult == SOCKET_ERROR)
{
    printf("bind failed with error: %d\n", WSAGetLastError());
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}

//设置套接字为非阻塞模式
int iMode = 1;
iResult = ioctlsocket(ServerSocket, FIONBIO, (u_long*)&iMode);
if (iResult == SOCKET_ERROR)
{
    printf("ioctlsocket failed with error: %ld\n", WSAGetLastError());
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}
// 监听套接字
iResult = listen(ServerSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR)
{
    printf("listen failed !\n");
    closesocket(ServerSocket);
    WSACleanup();
    return -1;
}
printf("TCP server starting\n");
```

```
int err;
// 循环等待
while (true)
{   //注意，因为传入的ServerSocket非阻塞，所以成功返回的AcceptSocket也是非阻塞的
    AcceptSocket = accept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen);
    if (AcceptSocket == INVALID_SOCKET)
    {
        err = WSAGetLastError();
        if (err == WSAEWOULDBLOCK)          // 无法立即完成非阻塞套接字上的操作
        {
            Sleep(1000);
            continue;
        }
        else
        {
            printf("accept failed !\n");
            closesocket(ServerSocket);
            WSACleanup();
            return 1;
        }
    }
```

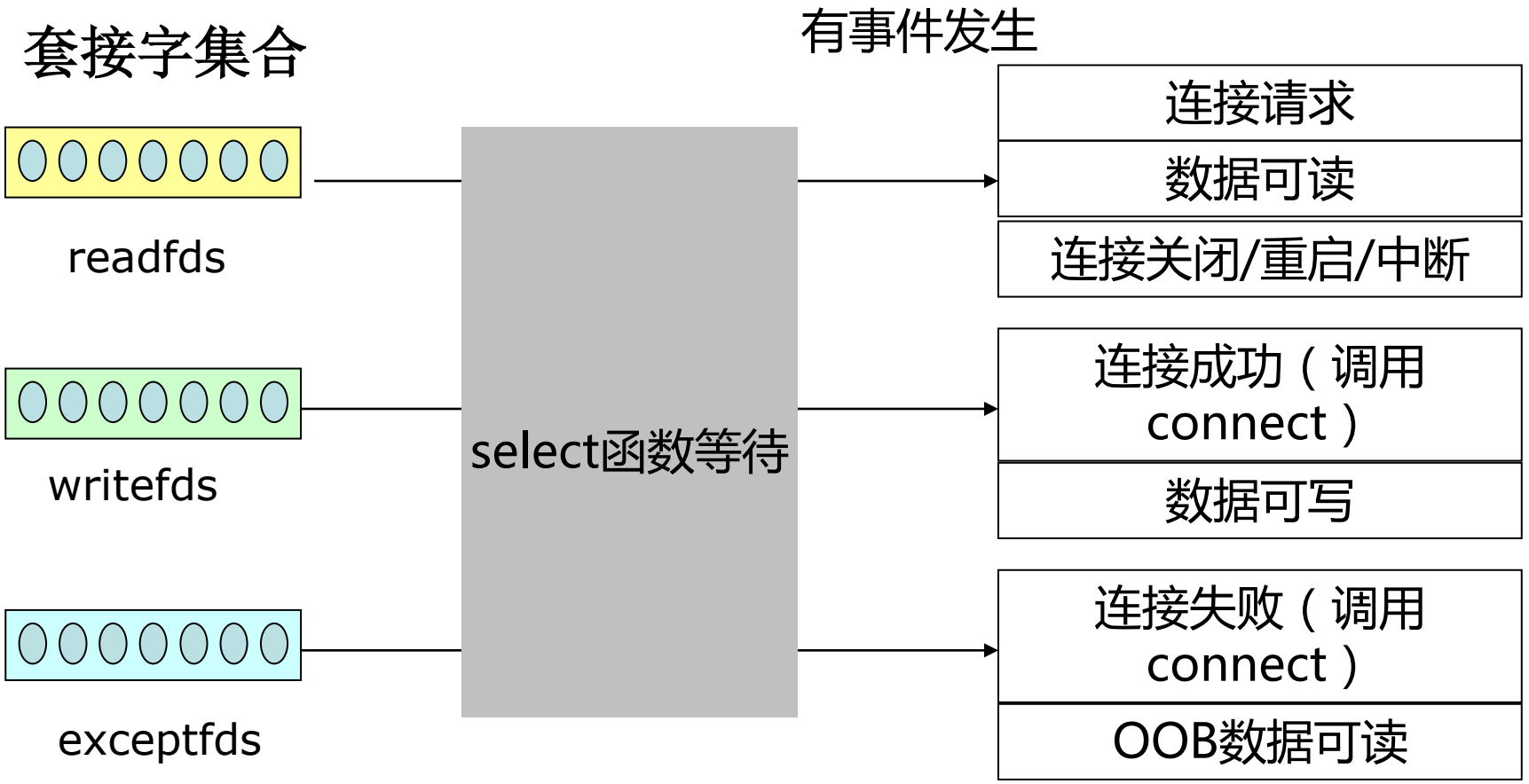
```
    //循环接收数据
    while (TRUE)
    {
        memset(recvbuf, 0, recvbuflen);
        iResult = recv(AcceptSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0)
        {
            //情况1：成功接收到数据
            printf("\nBytes received: %d\n", iResult);
            //处理数据请求
            //....
            //跳出轮询接收
            continue;

        }
        else if (iResult == 0)
        {
            //情况2：连接关闭
            printf("Current Connection closing, waiting for the next connection...\n");
            closesocket(AcceptSocket);
            break;
        }
        else
        {
            //情况3：接收发生错误
            err = WSAGetLastError();
            if (err = WSAEWOULDBLOCK)
            {
                //无法立即完成非阻塞套接字上的操作
                Sleep(1000);
                printf("\n当前I/O不满足，等待1000毫秒轮询");
                continue;
            }
            else
            {
                printf("recv failed with error: %d\n", err);
                closesocket(AcceptSocket);
                closesocket(ServerSocket);
                WSACleanup();
                return 1;
            }
        }
    }
}
```

```
    // cleanup
    closesocket(ServerSocket);
    closesocket(AcceptSocket);
    WSACleanup();
    return 0;
}
```

（3）I/O复用模型（I/O Multiplexing Model）



（3） I/O复用模型（I/O Multiplexing Model）



优点：避免了阻塞模式下的线程膨胀问题

缺点：可监控的套接字数量有限

I/O复用模型

- I/O复用模型也叫Select模型继承自 BSD UNIX的Berkeley Sockets，该模型是因为使用select()函数来管理I/O而得名。
- 程序通过调用select()函数可以获取一组指定套接字的状态，这样可以保证及时捕捉到最先得到满足的网络I/O事件，从而可保证对各套接字I/O操作的及时性。
- 这里I/O事件是指监听套接字上有用户请求到达、非监听套接字接受到数据、套接字已准备好可以发送数据等事件。

套接字集合fd_set

- select函数使用套接字集合fd_set来管理多个套接字，因此在学习使用select()函数之前，需要先了解套接字集合fd_set。
- 套接字集合fd_set是一个结构体，用于保存一系列的特定套接字.

- 其定义如下：

```
typedef struct fd_set
{
    unsigned int  fd_count;
    SOCKET fd_array[FD_SETSIZE];
} fd_set;
```

- fd_count用来保存集合中套接字的数目，套接字数组fd_array用于存储集合中各个套接字的描述符。
- FD_SETSIZE是一个常量，在WinSock2.h中定义，其值为64。

套接字集合相关的宏

- 为了方便编程，winsock提供了四个宏来对套接字集合进行操作。
- FD_ZERO(*set)
  - 初始化set为空集合。套接字集在使用前总是应该清空。
- FD_CLR(s,*set)
  - 从set移除套接字s
- FD_ISSET(s,*set)
  - 检查s是不是set的成员，如果是返回TRUE
- FD_SET(s,*set)
  - 添加套接字到集合

select函数

```
int select(
    int nfds, //忽略，仅为了与berkaley套接字兼容
    fd_set * readfds, //一个套接字集合，用于检查可读性
    fd_set * writefds, //一个套接字集合，用于检查可写性
    fd_set * exceptfds, //一个套接字集合，用于检查错误
    const struct timeval * timeout //指定此函数等待的最长时间，若为NULL，则最长时间为无限大
)
```

- 返回值：负值表示select错误；正值表示某些套接字可读写或出错；0表示timeout指定的时间内没有可读写或出错误的套接字 。

- 参数 timeout指向一个结构体(struct timeval)变量，该结构体定义如下：

```
typedef struct timeval
{
    long tv_sec; //指示等待多少秒
    long tv_usec; //指示等待多少毫秒
} timeval;
```

- 该结构体指针指向的结构体变量指定了select()函数等待的最长时间。如果为NULL，select()将会无限阻塞，直至有网络事件发生。如果将这个结构设置为(0,0)，select ()函数会马上返回。

select()对三个套接字集合的操作

- select()函数的三个参数指向的三个套接字集合分别用于保存要检查可读性（readfds）、可写性（writefds）和是否出错（exceptfds）的套接字。
- 当select()返回时，它将移除这三个套接字集合中没有发生相应I/O事件的套接字。

- Select()返回时，如果有下列事件发生，对应的套接字不会被删除
  - 对于readfds，主要有以下事件：
    - 数据可读
    - 连接已经关闭、重启或中断
    - listen已经调用，并且有一个连接请求到达，accept函数将成功
  - 对于writefds，主要有以下事件：
    - 数据能够发送
    - 如果一个非阻塞连接调用正在被处理，连接已经成功
  - 对于exceptfds，主要有以下事件：
    - 如果一个非阻塞连接调用正在被处理，连接企图失败
    - OOB数据可读

使用Select模型编程的方法

- 根据select()函数的工作过程，不难得出使用Select模型编写程序的基本步骤：

  ①用FD_ZERO宏来初始化需要的fd_set；

  ②用FD_SET宏来将套接字句柄分配给相应的fd_set，例如，如果要检查一个套接字是否有需要接收的数据，则可用FD_SET宏把该套接字的描述符加入可读性检查套接字集合中（第二个参数指向的套接字集合）；

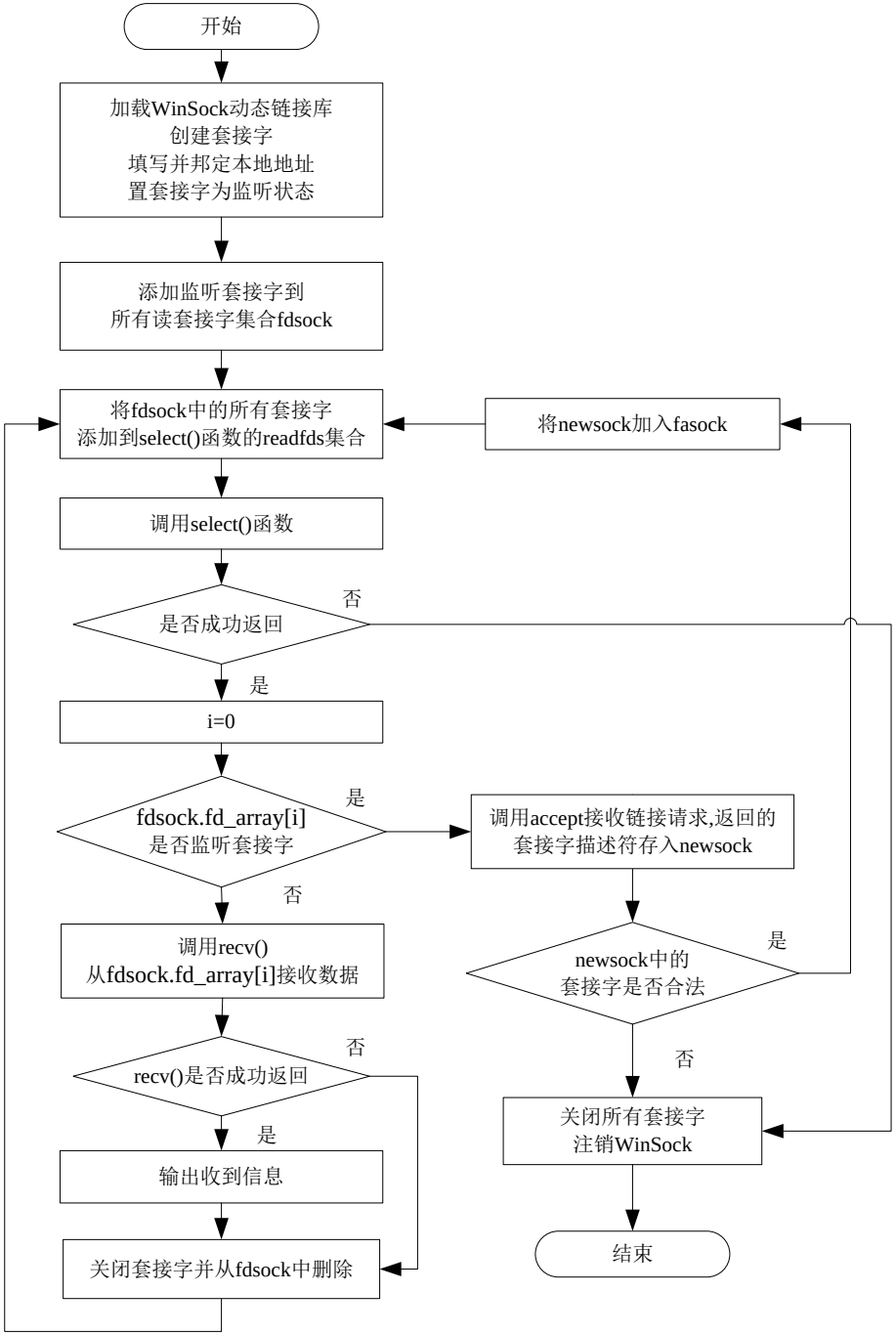
③调用select()函数，该函数将会阻塞直到满足返回条件，返回时，各集合中无网络I/O事件发生的套接字将被删除。

例如，对可读性检查集合readfds中的套接字，如果select()函数返回时接受缓冲区中没有数据需要接收，select()函数则会把套接字从集合中删除掉；

④用FD_ISSET对套接字句柄进行检查，如果被检查的套接字仍然在开始分配的那个fd_set里，则说明马上可以对该套接字进行相应的IO操 作。

例如，一个分配给可读性检查套接字集合readfds的套接字，在select()函数返回后仍然在该集合中，则说明该套接字已有数据已经到来， 马上调用recv函数可以读取成功。

- 事实上，实际的应用程序通常不会只有一次网络I/O，因此不会只有一次select()函数调用，而应该是上述过程的一个循环。


开始
加载WinSock动态链接库
创建套接字
填写并邦定本地地址
置套接字为监听状态
添加监听套接字到
所有读套接字集合fdsock
将fdsock中的所有套接字
添加到select()函数的readfds集合
将newsock加入fasock
调用select()函数
是否成功返回
否
是
i=0
fdsock.fd_array[i]
是否监听套接字
是
否
调用accept接收链接请求,返回的
套接字描述符存入newsock
调用recv()
从fdsock.fd_array[i]接收数据
newsock中的
套接字是否合法
是
否
recv()是否成功返回
否
是
输出收到信息
关闭套接字并从fdsock中删除
关闭所有套接字
注销WinSock
结束

```c
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>

// 连接到winsock2对应的lib文件： Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT 27015    //默认服务器端口号为27015
int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;
    sockaddr_in addrClient;
    int addrClientlen = sizeof(sockaddr_in);
    char strIP[16];

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }
```

```
// 创建用于监听的套接字
ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
if (ServerSocket == INVALID_SOCKET)
{
    printf("socket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

// 为套接字绑定地址和端口号
SOCKADDR_IN addrServ;
addrServ.sin_family = AF_INET;
addrServ.sin_port = htons(DEFAULT_PORT);        // 监听端口为DEFAULT_PORT
addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
if (iResult == SOCKET_ERROR)
{
    printf("bind failed with error: %d\n", WSAGetLastError());
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}

// 监听套接字
iResult = listen(ServerSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR)
{
    printf("listen failed !\n");
    closesocket(ServerSocket);
    WSACleanup();
    return -1;
}
printf("TCP server starting\n");
```

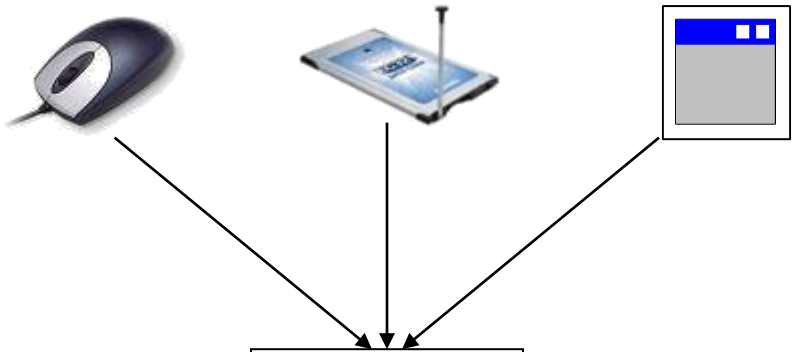
```
fd_set fdRead, fdSocket;
FD_ZERO(&fdSocket);
FD_SET(ServerSocket, &fdSocket);

while (TRUE)
{
    //通过select等待数据到达事件，如果有事件发生，select函数移除fdRead集合中没有未决I/O操作的套接字句柄，然后返回
    fdRead = fdSocket;
    iResult = select(0, &fdRead, NULL, NULL, NULL);
    if (iResult > 0)
    {
        //有网络事件发生
        //确定有哪些套接字有未决的I/O，并进一步处理这些I/O
        for (int i = 0; i < (int)fdSocket.fd_count; i++)
        {
            if (FD_ISSET(fdSocket.fd_array[i], &fdRead))
            {
                if (fdSocket.fd_array[i] == ServerSocket)
                {
                    if (fdSocket.fd_count < FD_SETSIZE)
                    {
                        //同时复用的套接字数量不能大于FD_SETSIZE
                        //有新的连接请求
                        AcceptSocket = accept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen);
                        if (AcceptSocket == INVALID_SOCKET)
                        {
                            printf("accept failed !\n");
                            closesocket(ServerSocket);
                            WSACleanup();
                            return 1;
                        }
                        //增加新的连接套接字进行复用等待
                        FD_SET(AcceptSocket, &fdSocket);                                    ;
                        printf("接收到新的连接：%s\n", inet_ntop(AF_INET, &(addrClient.sin_addr), strIP, 16));
                    }
                    else
                    {
                        printf("连接个数超限!\n");
                        continue;
                    }
                }
```

```
            else
            {
                //有数据到达
                memset(recvbuf, 0, recvbuflen);
                iResult = recv(fdSocket.fd_array[i], recvbuf, recvbuflen, 0);
                if (iResult > 0)
                {
                    //情况1：成功接收到数据
                    printf("\nBytes received: %d\n", iResult);
                    //处理数据请求
                    //....
                }
                else if (iResult == 0)
                {
                    //情况2：连接关闭
                    printf("Current Connection closing...\n");
                    closesocket(fdSocket.fd_array[i]);
                    FD_CLR(fdSocket.fd_array[i], &fdSocket);
                }
                else
                {
                    //情况3：接收失败
                    printf("recv failed with error: %d\n", WSAGetLastError());
                    closesocket(fdSocket.fd_array[i]);
                    FD_CLR(fdSocket.fd_array[i], &fdSocket);
                }
            }
        }
        //如果还有其它类等待的套接字，需要依次判断
        //……
    }
}
```

```
        else
        {
            printf("select failed with error: %d\n", WSAGetLastError());
            break;
        }
    }

    // cleanup
    closesocket(ServerSocket);
    WSACleanup();
    return 0;
}
```

（4）异步I/O复用模型（WSAAsyncSelect Model）

- **Windows消息机制**




```
WinMain()
{
    while(1)
    {
        等待;
        取出消息;
        调用消息处理函数;
    }
}
```

消息 ⟷ 消息处理函数

（4） 异步I/O复用模型（WSAAsyncSelect Model）

函数定义

int WSAAsyncSelect(SOCKET *s*, HWND *hWnd*, unsigned int *wMsg*, long *lEvent*)

值	含义
FD_READ	期望在套接字s上收到数据时接到通知
FD_WRITE	期望在套接字s上可发送数据时接到通知
FD_OOB	期望在套接字s上有带外数据到达时接到通知
FD_ACCEPT	期望在套接字s上有外来连接时接到通知
FD_CONNECT	期望在套接字s上连接建立完成时接到通知
FD_CLOSE	期望在套接字s关闭时接到通知

WSAAsyncSelect()函数

- WSAAsyncSelect模型是基于Windows的消息机制实现的，当网络事件发生时，Windows系统将发送一条消息给应用程序，应用程序将根据消息做出相应的处理。

- WSAAsyncSelect模型中常用的网络事件包括FD_READ网络事件、FD_WRITE事件、FD_ACCEPT事件、FD_CONNECT事件、FD_CLOSE事件等。
  - FD_READ网络事件：有数据到达但还没有发送FD_READ网络事件；调用recv()或者recvfrom()函数后，如果仍然有可读数据。
  - FD_WRITE事件：调用connect()或者accept()函数后，连接已经建立；调用send()或者sendto()函数返回WSAEWOULDBLOCKE错误后，再次调用send()或者sendto()函数可能成功(缓冲区变得可用)。

- FD_ACCEPT事件：当前有连接请求需要接受，即有当连接请求到达但还没有发送FD_ACCEPT网络事件；调用accept()函数后，如果还有另外连接请求需要接受时。
- FD_CONNECT事件：调用connect()函数后，建立连接完成
- FD_CLOSE事件仅对面向连接套接字有效，在下面情况下发送FD_CLOSE事件：对方执行了套接字关闭并且没有数据可读，如果数据已经到达并等待读取，FD_CLOSE事件不会被发送，直到所有的数据都被接收。

- WSAAsyncSelect模型的核心是WSAAsyncSelect()函数，该函数的主要功能是针对一个指定的套接字向系统注册一个或多个应用程序关注的网络事件。
- 注册网络事件时需要指定事件发生时需要发送的消息以及处理该消息的窗口的句柄。
- 程序运行时，一旦该事件发生时，系统将向指定的窗口发送指定的消息。

WSAAsyncSelect 函数原形

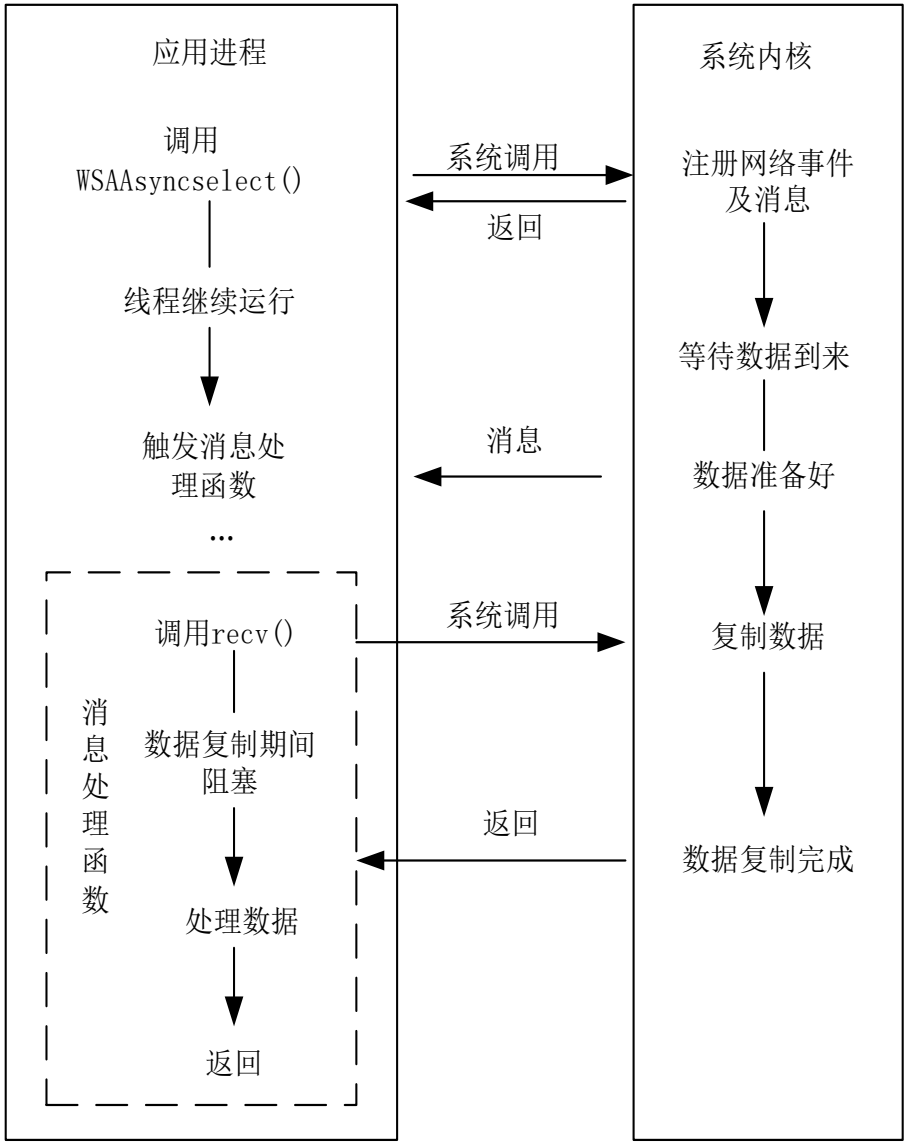
- int WSAAsyncSelect(
  SOCKET s,
  HWND hWnd，
  unsigned int wMsg,
  long lEvent
)
- 参数
  - s：需要事件通知的套接字。
  - hWnd: 当网络事件发生时接收消息的窗口句柄。
  - wWsg: 当网络事件发生时向窗口发送的用户自定义消息。
  - lEvent：要注册的应用程序感兴趣的套接字s的网络事件集合。
- 函数返回值：应用程序感兴趣的事件注册成功，则返回0；如果注册失败，则返回SOCKET_ERROR。

```
if(WSAAsyncSelect(m_acceptSocket,m_ hWnd, MsgRecv,
 FD_READ)!=0)
	{
		CString str3("套接字异步事件注册失败！ ");
		MessageBox(str3);
		closesocket(m_acceptSocket);
		closesocket(m_ListenSocket);
		m_SendButton.EnableWindow(0);
	}
```

- WSAAsyncSelect模型是非阻塞的，在应用程序中调用WSAAsyncSelect()函数后，该函数将向系统注册完成参数lEvent指定的网络事件后立即返回。
- WSAAsyncSelet模型是异步的，当已被注册的网络事件发生时，系统将向应用程序发送消息，该消息将由参数hWnd指定的窗口的相应消息处理函数进行处理，编写相应的消息处理函数是程序编写的主要工作之一。

使用WSAAsyncSelect模型接收数据的过程

- 调用recv()函数接收数据前，首先调用WSAAsyncselect()函数注册网络事件、事件发生时发出的用户自定义消息及处理消息的窗口。
- 当系统收到数据时，系统将向应用程序发送消息。
- 应用程序接收到这个消息后，将在消息对应的消息处理函数中调用recv()函数接收数据并处理数据。



- 应用程序需要注册哪些网络事件，取决于实际的需求。如果应用程序同时对多个网络事件感兴趣。需要对网络事件类型执行按位或（|）运算。然后将它们赋值给lEvent参数。

```
SAAsyncSelect(s,
    hWnd,WM_SOCKET,FD_CONNECT|FD_READ|FD_CLOSE);
```

  - 当套接字s连接完成、有数据可读或者套接字关闭的网络事件事件发生时，就会有WM_SOCKET消息发送给窗口句柄为hWnd窗口。

•如果要取消某个套接字的所有已注册的网络事件，需要以参数IEvent值为0来调用WSAAsyncSelect()函数，其格式如下。

WSAAsyncSelect(s, hWnd, 0, 0);

– s为要被取消注册网络事件的套接字，hWnd为注册这些事件时指定的接收网络事件消息的窗口的句柄。

•取消网络事件的注册之后，系统将不再为该套接字发送任何与网络事件相关的消息。

- 需要特别强调，WSAAsyncSelect模型应用在Windows环境下，使用该模型时必须创建窗口。
- 而Slelect模型广泛应用在Unix系统和Windows系统，使用该模型不需要创建窗口。
- 应用程序调用WSAAsyncSelect()函数后，自动将套接字设置为非阻塞模式，而应用程序中调用select()函数后，并不能改变该套接字的工作方式。

- 如果已对一个套接口进行了WSAAsynSelect() 操作，则任何用ioctlsocket()来把套接口重新设置成阻塞模式的试图将以WSAEINVAL失败。为了把套接口重新设置成阻塞模式，应用程序必须首先用WSAAsynSelect()调用（IEvent参数置为0）来禁止WSAAsynSelect()。

- 在同一个套接字上，多次调用WSAAsyncSelect()函数注册不同的网络事件，后一次函数调用将取消前一次注册的网络事件。
- 例如，下面的两行代码：

```
WSAAsyncSelect(s, hWnd, wMsg, FD_READ);
WSAAsyncSelect(s, hWnd,wMsg, FD_WRITE);
```

第一行调用WSAAsyncSelect()函数为套接字s注册FD_READ网络事件，然后又再次调用WSAAsyncSelect()函数为同一个套接字s注册FD_WRITE 网络事件，那么此后应用程序将只能接收到套接字s的FD_WRITE网络事件。

- 在同一个套接字上多次调用WSAAsyncSelect()函数为不同的网络事件注册不同的消息，后一次的函数调用也将取消前面注册的网络事件。例如，下面的代码中，第二次函数调用将会取消第一次函数调用的作用。只有FD_WRITE网络事件能过wMsg2消息通知到窗口，而FD_READ事件则无法触发wMsg1消息。

```
WSAAsyncSelect(s,hWnd,wMsg1,FD_READ);
WSAAsyncSelect(s,hWnd,wMsg2,FD_WRITE);
```

- 如果为一个监听套接字注册了FD_ACCEPT、FD_READ和FD_WRITE网络事件，则在该监听套接字上调用accept()函数接受连接请求所创建的任何套接字，也会触发FD_ACCEPT、FD_READ和FD_WRITE网络事件，即相当于为这写套接字也注册了同样的网络事件。
- 这是因为调用accept()函数接受的套接字和监听套接字具有同样的属性。所以，任何为监听套接字设置的网络事件对接受的套接字同样起作用。若需要不同的消息和网络事件，应用程序应该调用WSAAsyncSelect()函数，为套接字注册不同的网络事件和消息。

- 在程序中为一个FD_READ网络事件一般不要多次调用recv()函数来接受数据，如果应用程序为一个FD_READ网络事件调用了多个recv()函数，可能会使得该应用程序接收到多个FD_READ网络事件。
- 例如，假设一开始套接字接收到了300字节的数据，这时系统将向应用程序发送FD_READ事件通知，如果应用程序的相应消息处理函数中连续调用三次recv()函数，每次都只接收100字节数据，前两次的recv()调用都将导致系统发送FD_READ网络事件通知，第三次调用recv()时将会把剩余数据接收完，因而不会发送FD_READ。前两次发送的FD_READ都会引发系统再次调用该消息处理函数。

- 当套接字上注册的网络事件发生时，系统将向指定的窗口发送指定的用户自定义消息，进而触发对该消息的消息处理函数的调用。
- 编写程序时，消息处理函数可使用MFC的类向导添加，函数的具体功能则由编程者实现。函数的名字在添加该函数时是由编程者指定。

- 应用程序不必在收到FD_READ消息时读进所有可读的数据。每接收到一次FD_READ网络事件，应用程序调用一次recv()函数是恰当的。

- 网络事件消息的处理函数具有类似下面代码所示的原型。

```
afx_msg LRESULT OnSocketMsg(WPARAM
    wParam, LPARAM lParam)
```

- wParam参数存放发生网络事件的套接字的句柄，
- lParam参数的低16位存放的是发生的网络事件，高16位则用于存放网络事件发生错误时的错误码。
- 函数的参数个数和类型是由系统规定的，它们的值在因消息到达而触发函数运行时由系统传入。

- 为了便于编程者获取网络事件或网络事件的错误信息，WinSock提供了WSAGETSELECTEVENT和WSAGETSELECTERROR两个宏，这两个宏的使用格式如下所示。

```
WORD wEvent，wError;
wEvent=WSAGETSELECTEVENT(lParam);
wError=WSAGETSELECTERROR(lParam);
```

WSAAsyncSelect模型的编程方法

- WSAAsyncSelect模型是基于Windows消息机制的，而其WSAAsyncSelect()函数要求消息的接收对象必须是一个窗口，因此基于WSAAsyncSelect模型的应用程序一般都是图形界面的窗口应用程序。
- 程序的编写可以分为两大部分：建立并完善应用程序框架、编写消息处理函数。

- 建立并完善应用程序框架需要完成如下任务：
  - ①使用应用程序向导创建对话框应用程序框架；
  - ②设计程序界面，主要是绘制控件并设置相关属性等；
  - ③为相关控件添加控件变量；
  - ④将通信所必须的套接字变量作为成员变量添加到窗口类中；
  - ⑤添加WSAAsyncSelect()函数在为套接字注册网络事件时发送的自定义消息；
  - ⑥在窗口类的成员函数OnInitDialog()中添加程序代码，完成创建套接字、给套接字绑定地址、使套接字处于监听状态、调用WSAAsyncSelect()函数为套接字注册网络事件等功能。

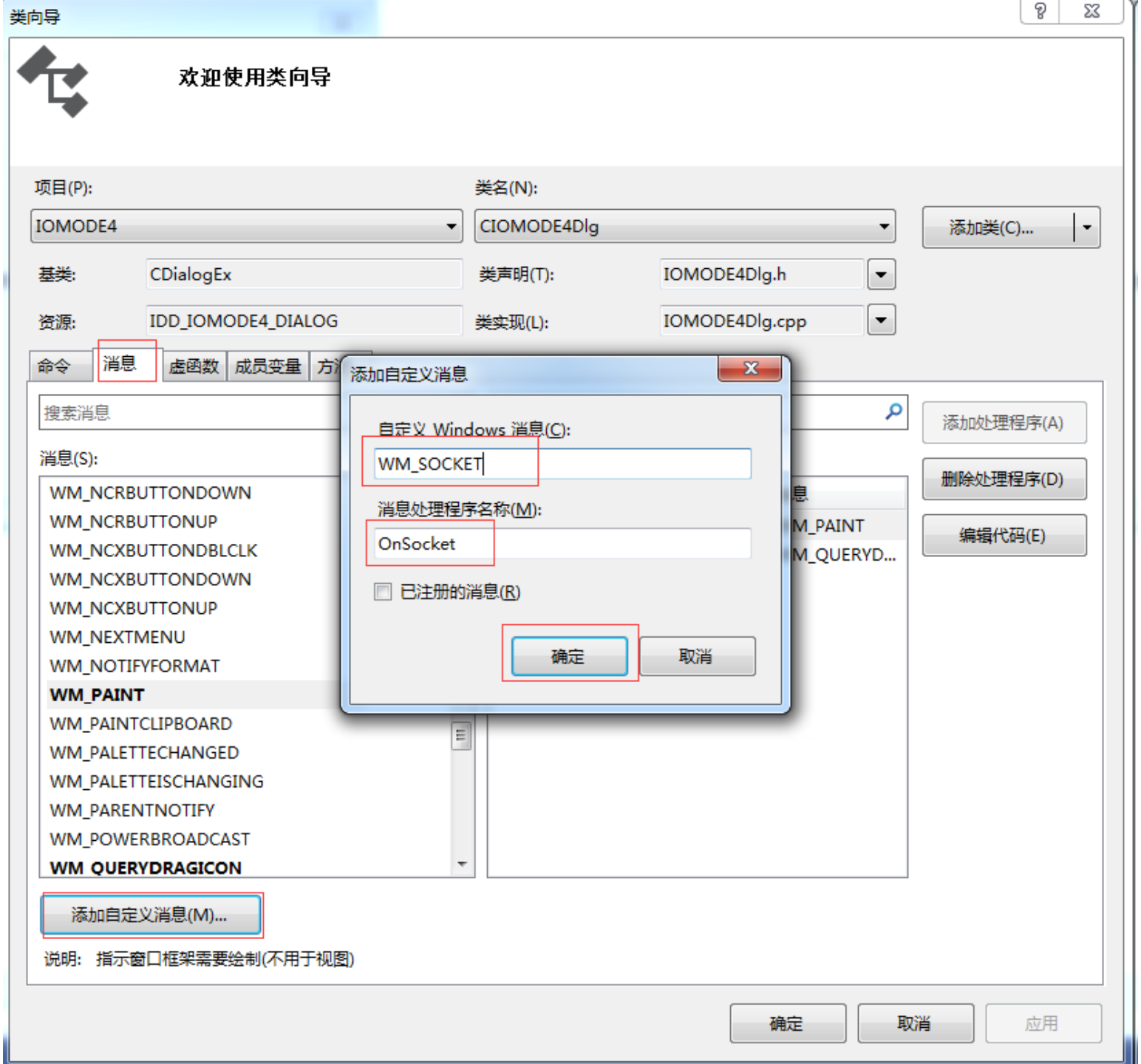
- 编写消息处理函数是程序设计的主要工作，除了编写相关控件消息的处理函数外，最主要的就是为套接字编写与网络事件关联的自定义消息的处理函数，在这些处理函数中要调用相关的套接字函数完成相关的IO处理。
- 现在的VS版本建议用WSAEventSelect()取代WSAAnsycSelect()

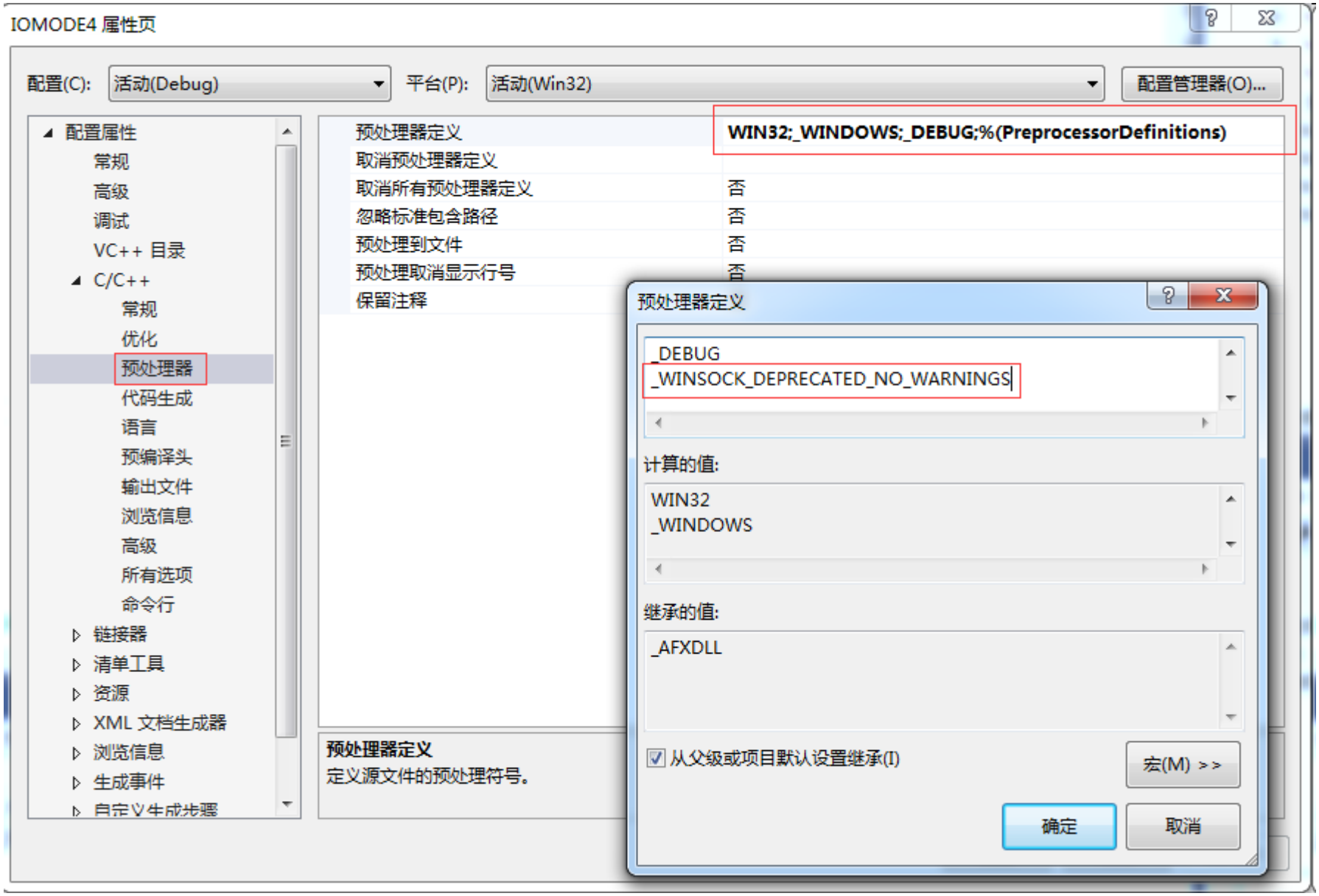
IOMODE4Dlg.h | IOMODE4.rc - ID..._DIALOG - Dialog | IOMODE4.cpp

IOMODE4 | (全局范围)

```
4
5       #pragma once
6
7       #define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
8       #define DEFAULT_PORT 27015  //默认服务器端口号为27015
9       #define WM_SOCKET WM_USER+101
10
11      // CIOMODE4Dlg 对话框
12      class CIOMODE4Dlg : public CDialogEx
13      {
14      // 构造
15      public:
16          CIOMODE4Dlg(CWnd* pParent = nullptr);   // 标准构造函数
17
18      // 对话框数据
19      #ifdef AFX_DESIGN_TIME
20          enum { IDD = IDD_IOMODE4_DIALOG };
21      #endif
22
23          protected:
24          virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV 支持
25
26
27      // 实现
28      protected:
29          HICON m_hIcon;
30
31          // 生成的消息映射函数
32          virtual BOOL OnInitDialog();
33          afx_msg void OnPaint();
34          afx_msg HCURSOR OnQueryDragIcon();
35          DECLARE_MESSAGE_MAP()
36          afx_msg LRESULT OnSocket(WPARAM wParam, LPARAM lParam);
37      public:
38          CListBox list;
39          afx_msg void OnBnClickedStartServer();
40      };
41
```

在对话框头文件中定义用户自定义消息WM_SOCKET


类向导
欢迎使用类向导
项目(P):
IOMODE4
类名(N):
CIOMODE4Dlg
添加类(C)...
基类:
CDialogEx
类声明(T):
IOMODE4Dlg.h
资源:
IDD_IOMODE4_DIALOG
类实现(L):
IOMODE4Dlg.cpp
命令
消息
虚函数
成员变量
搜索消息
消息(S):
WM_NCRBUTTONDOWN
WM_NCRBUTTONUP
WM_NCXBUTTONDBLCLK
WM_NCXBUTTONDOWN
WM_NCXBUTTONUP
WM_NEXTMENU
WM_NOTIFYFORMAT
WM_PAINT
WM_PAINTCLIPBOARD
WM_PALETTECHANGED
WM_PALETTEISCHANGING
WM_PARENTNOTIFY
WM_POWERBROADCAST
WM_QUERYDRAGICON
添加处理程序(A)
删除处理程序(D)
编辑代码(E)
添加自定义消息(M)...
说明: 指示窗口框架需要绘制(不用于视图)
确定
取消
应用
添加自定义消息
自定义 Windows 消息(C):
WM_SOCKET
消息处理程序名称(M):
OnSocket
已注册的消息(R)
确定
取消

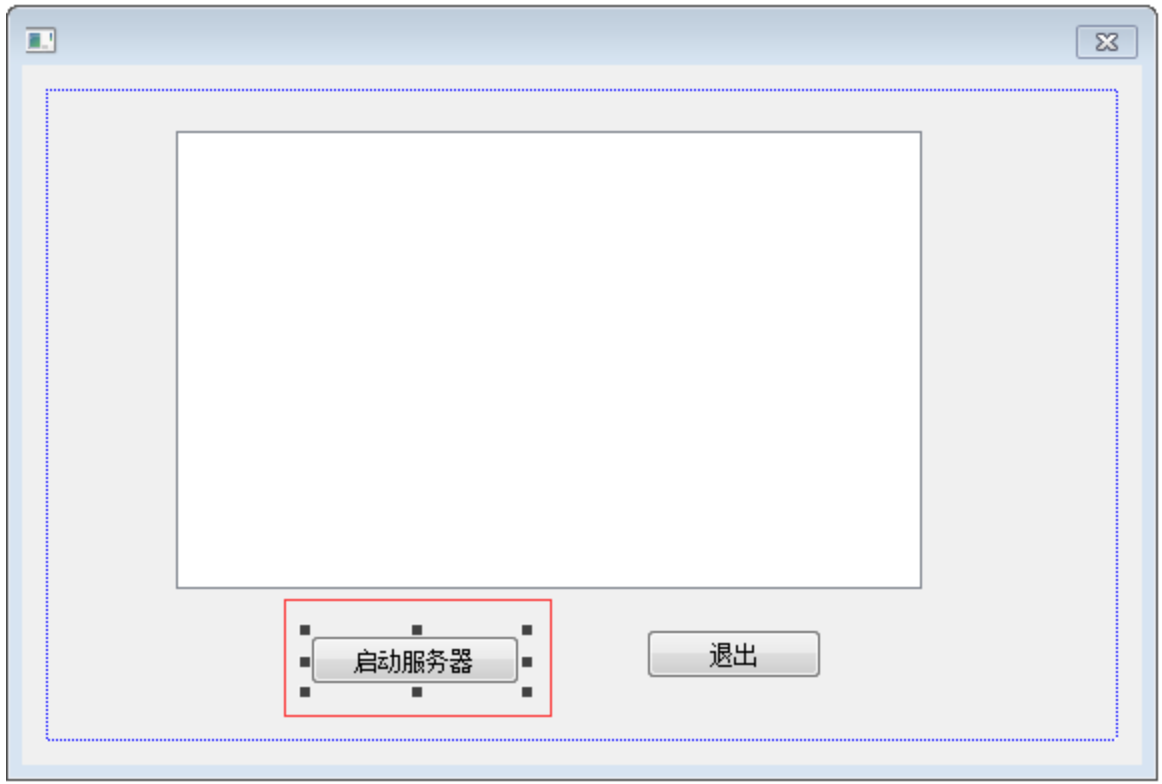


WSAAsyncSelect()已被废弃，为能继续使用，需设置预处理宏**_WINSOCK_DEPRECATED_NO_WARNINGS**

消息处理函数

```
afx_msg LRESULT CIOMODE4Dlg::OnSocket(WPARAM wParam, LPARAM lParam)
{
    //自定义的关闭与缓冲区有消息
    int iResult;
    sockaddr_in addrClient;
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;
    int addrClientlen = sizeof(sockaddr_in);
    SOCKET s, AcceptSocket;

    CString str;

    s = static_cast<SOCKET>(wParam);      //取有事件发生的套接字句柄
    if (WSAGETSELECTERROR(lParam))
        list.InsertString(0, "套接字错误。");
    else
    {
        switch (WSAGETSELECTEVENT(lParam))
        {
            //判断消息类型
        case FD_ACCEPT:
            //检测到有新的连接进入
            AcceptSocket = accept(s, (sockaddr FAR*) & addrClient, &addrClientlen);
            if (AcceptSocket == INVALID_SOCKET)
            {
                list.InsertString(0, "accept failed !");
                closesocket(s);
            }
            str.Format("接收到新的连接：%s\n", inet_ntoa(addrClient.sin_addr));
            list.InsertString(0, str);
            //增加新的连接套接字进行等待
            iResult = WSAAsyncSelect(AcceptSocket, m_hWnd, WM_SOCKET, FD_READ | FD_WRITE | FD_CLOSE);
            if (iResult == SOCKET_ERROR)
            {
                list.InsertString(0, "WSAAsyncSelect设定失败!");
            }
            break;
```

```
        case FD_READ:
            //有数据到达
            iResult = recv(s, recvbuf, recvbuflen, 0);
            if (iResult >= 0)
            {
                //情况1：成功接收到数据
                str.Format("Bytes received: %d", iResult);
                list.InsertString(0, str);
                //处理数据请求
                //....
            }
            else
            {
                //情况2：接收失败
                str.Format("recv failed with error: %d\n", WSAGetLastError());
                list.InsertString(0, str);
                closesocket(s);
            }
            break;
        case FD_WRITE:
        {}
        break;
        case FD_CLOSE:
            //情况3：连接关闭
            list.InsertString(0, "Current Connection closing...\n");
            closesocket(s);
            break;
        default:
            break;
        }
    }
    return TRUE;
}
```

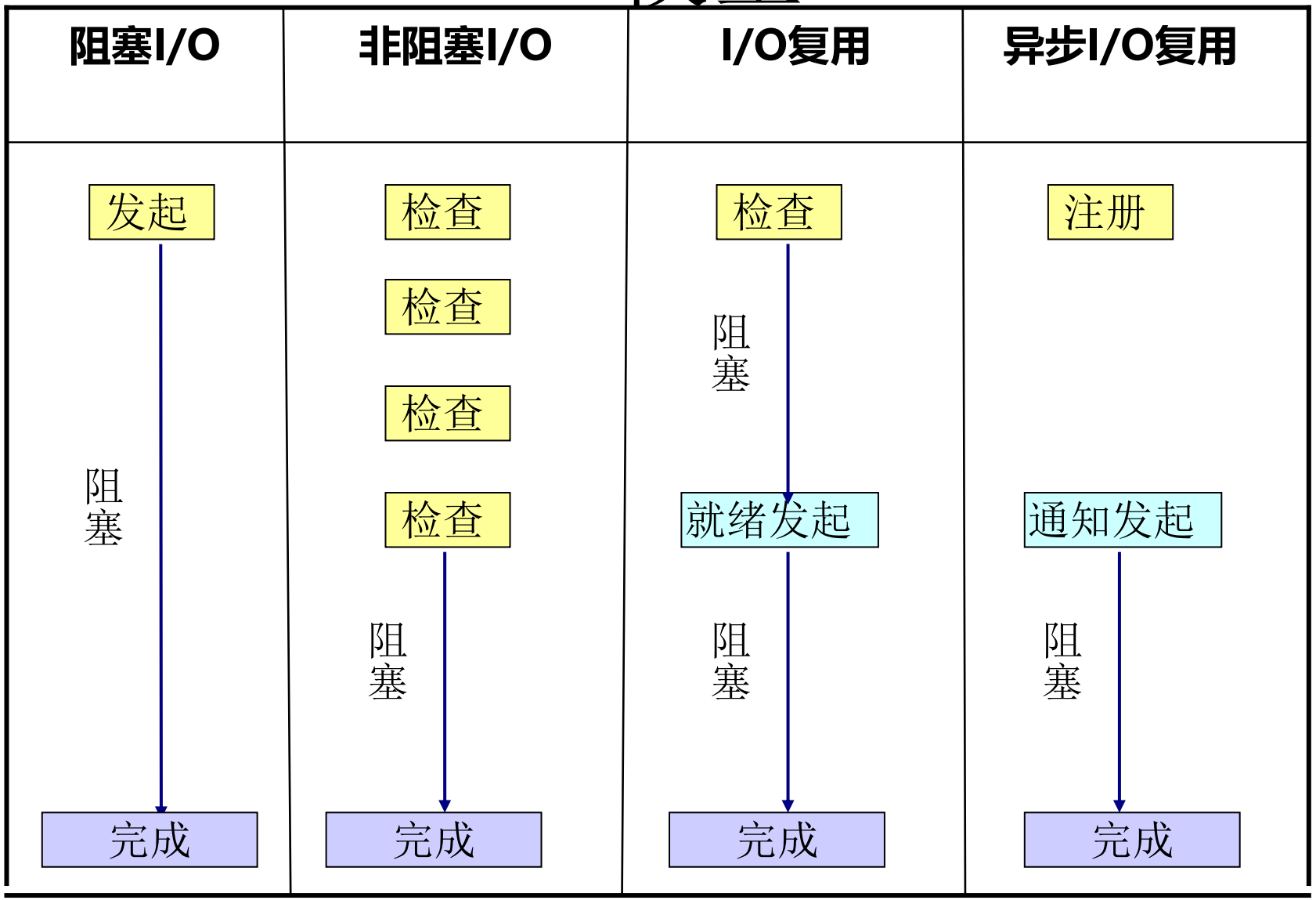

启动服务器
退出

启动服务

```
void CIOMODE4Dlg::OnBnClickedStartServer()
{
    // TODO: 在此添加控件通知处理程序代码
    // TODO: 在此添加控件通知处理程序代码
    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        list.InsertString(0, "WSAStartup函数出错。");
        return;
    }

    // 创建用于监听的套接字
    ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (ServerSocket == INVALID_SOCKET)
    {
        list.InsertString(0, "socket函数出错。");
        WSACleanup();
        return;
    }

    // 为套接字绑定地址和端口号
    SOCKADDR_IN addrServ;
    addrServ.sin_family = AF_INET;
    addrServ.sin_port = htons(DEFAULT_PORT);        // 监听端口为DEFAULT_PORT
    addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
    if (iResult == SOCKET_ERROR)
    {
        list.InsertString(0, "bind函数出错。");
        closesocket(ServerSocket);
        WSACleanup();
        return;
    }
```

```
    // 监听套接字
    iResult = listen(ServerSocket, SOMAXCONN);
    if (iResult == SOCKET_ERROR)
    {
        list.InsertString(0, "listen函数出错。");
        closesocket(ServerSocket);
        WSACleanup();
        return;
    }
    list.InsertString(0, "服务器启动。。。");

    // 产生相应传递给窗口的消息为WM_SOCKET ，这是自定义消息
    iResult = WSAAsyncSelect(ServerSocket, m_hWnd, WM_SOCKET, FD_ACCEPT);
    if (iResult == SOCKET_ERROR)
    {
        list.InsertString(0, "WSAAsyncSelect设定失败!");
        return;
    }
    return;
}
```

I/O模型


阻塞I/O
非阻塞I/O
I/O复用
异步I/O复用
发起
检查
检查
检查
检查
检查
注册
阻塞
阻塞
阻塞
阻塞
就绪发起
通知发起
完成
完成
完成
完成

基于事件的WSAEventSelect模型

- 基于事件的WSAEventSelect模型是用另一种Windows机制实现的异步I/O模型，与基于WSAAsyncSelect的异步I/O模型的最主要区别是网络事件发生时系统通知应用程序的方式不同。
- WSAEventSelect模型允许在多个套接字上以接收事件为基础的网络事件通知。
- 现在VS推荐用WSAEventSelect取代WSAAsyncSelect

事件对象

- 事件对象属于内核，它包含三个主要内容：
  - 使用计数
  - 工作模式(BOOL)：自动重置或人工重置
  - 工作状态(BOOL)：已授信(signaled)状态,未授信(nonsignaled)状态

WSACreateEvent function

2018/12/05 • 2 分钟阅读时长

The **WSACreateEvent** function creates a new event object.

Syntax

C++ 复制

```
WSAEVENT WSAAPI WSACreateEvent();
```

Parameters

This function has no parameters.

Return value

If no error occurs, **WSACreateEvent** returns the handle of the event object. Otherwise, the return value is WSA_INVALID_EVENT. To get extended error information, call WSAGetLastError.

使用WSACreateEvent()函数创建事件对象，成功则函数返回事件对象句柄，失败则返回WSA_INVALID_EVENT
调用WSACreateEvent()函数创建的事件对象处于人工重置模式和未授信状态。

WSAResetEvent function

2018/12/05 • 2 分钟阅读时长

The **WSAResetEvent** function resets the state of the specified event object to nonsignaled.

Syntax

C++ 复制

```
BOOL WSAAPI WSAResetEvent(
  WSAEVENT hEvent
);
```

Parameters

`hEvent`

A handle that identifies an open event object handle.

Return value

If the **WSAResetEvent** function succeeds, the return value is **TRUE**. If the function fails, the return value is **FALSE**. To get extended error information, call WSAGetLastError.

当网络事件发生时，与套接字关联的事件对象被从未授信状态转为授信状态。
调用WSAResetEvent()函数可以将事件对象从已授信状态修改为未授信状态。

WSASetEvent function

2018/12/05 • 2 分钟阅读时长

The **WSASetEvent** function sets the state of the specified event object to signaled.

Syntax

```
C++

BOOL WSAAPI WSASetEvent(
  WSAEVENT hEvent
);
```

Parameters

`hEvent`

Handle that identifies an open event object.

Return value

If the function succeeds, the return value is **TRUE**.

If the function fails, the return value is **FALSE**. To get extended error information, call WSAGetLastError.

WSASetEvent()函数可以将给定的事件对象设置为已授信状态

WSACloseEvent function

2018/12/05 • 2 分钟阅读时长

The **WSACloseEvent** function closes an open event object handle.

Syntax

```
C++

BOOL WSAAPI WSACloseEvent(
  WSAEVENT hEvent
);
```

Parameters

`hEvent`

Object handle identifying the open event.

Return value

If the function succeeds, the return value is **TRUE**.

If the function fails, the return value is **FALSE**. To get extended error information, call WSAGetLastError.

处理完网络事件后，需要调用WSACloseEvent()函数关闭事件对象句柄，释放事件对象占用的资源。

WSAEventSelect function

2018/12/05 • 11 分钟阅读时长

The **WSAEventSelect** function specifies an event object to be associated with the specified set of FD_XXX network events.

Syntax

```
C++

int WSAAPI WSAEventSelect(
  SOCKET   s,
  WSAEVENT hEventObject,   //与网络事件集合相关联的事件对象句柄
  long     lNetworkEvents  //感兴趣的网络事件集合，每个为1的比特代表一个事件
);
```

Parameters

`s`

A descriptor identifying the socket.

`hEventObject`

A handle identifying the event object to be associated with the specified set of FD_XXX network events.

`lNetworkEvents`

A bitmask that specifies the combination of FD_XXX network events in which the application has interest.

如果函数执行成功，则返回0。否则返回SOCKET_ERROR

WSAEventSelect

- WSAEventSelect()函数对网络事件的注册方法与WSAAsnycSelect()函数类似
- iResult=WSAEventSelect(s,hEventObject,FD_READ|FD_CLOSE);
- 如果要取消指定套接字上的所有通知事件，则可以在调用WSAEventSelect()函数时将事件参数IEvent设置为0
- iResult=WSAEventSelect(s,hEventObject,0);
- WSAEventSelect()函数会将套接字设置为非阻塞状态，如果需要将套接字设置回默认的阻塞状态，则必须清除与套接字相关联的注册事件，然后调用ioctlsoket()或WSAIoctl()将套接字设置为阻塞模式。

WSAWaitForMultipleEvents

```
DWORD WSAAPI WSAWaitForMultipleEvents(
  DWORD          cEvents,   //lphEvents包含的事件句柄的数量
  const WSAEVENT *lphEvents, //指向事件句柄数组的指针
  BOOL           fWaitAll,
  DWORD          dwTimeout, //指定超时时间,以毫秒计数
  BOOL           fAlertable //当完例程在系统队列中排队等待执行时函数是否返回
);
```

fWaitAll: 为TRUE,则数组lphEvents中包含的所有事件对象都变成已授信状态才返回；为FALSE,则只要有一个事件对象变成已授信状态就返回。

dwTimeout:如果该参数为0，则函数检查事件对象的状态后立即返回；如果该参数为WSA_INFINITE,则该函数会无限期等待，直到满足参数fWaitAll指定条件。

fAlertable:为TRUE，说明函数返回时完成例程已经被执行；为FALSE,则说明该函数返回时完成例程还没被执行。该参数主要应用于重叠I/O模型。

WSAWaitForMultipleEvents

Return value

If the **WSAWaitForMultipleEvents** function succeeds, the return value upon success is one of the following values.

Return Value	Meaning
WSA_WAIT_EVENT_0 to (WSA_WAIT_EVENT_0 + cEvents - 1)	If the *fWaitAll* parameter is **TRUE**, the return value indicates that all specified event objects is signaled. If the *fWaitAll* parameter is **FALSE**, the return value minus **WSA_WAIT_EVENT_0** indicates the *lphEvents* array index of the signaled event object that satisfied the wait. If more than one event object became signaled during the call, the return value indicates the *lphEvents* array index of the signaled event object with the smallest index value of all the signaled event objects.
WSA_WAIT_IO_COMPLETION	The wait was ended by one or more I/O completion routines that were executed. The event that was being waited on is not signaled yet. The application must call the WSAWaitForMultipleEvents function again. This return value can only be returned if the *fAlertable* parameter is **TRUE**.
WSA_WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the *fWaitAll* parameter were not satisfied. No I/O completion routines were executed.

If the **WSAWaitForMultipleEvents** function fails, the return value is **WSA_WAIT_FAILED**. The following table lists values that can be used with WSAGetLastError to get extended error information.

WSA_WAIT_TIMEOUT:超时
WSA_WAIT_FAILED:出错，此时需检查cEvents和lphEvents两个参数是否有效
如果事件数组中有某一个事件被授信了，函数就会返回这个事件的索引值，这个索引值需要减去预定义值WSA_WAIT_EVENT_0才是这个事件在事件数组中的位置。

WSAEnumNetworkEvents function

2018/12/05 • 5 分钟阅读时长

The **WSAEnumNetworkEvents** function discovers occurrences of network events for the indicated socket, clear internal network event records, and reset event objects (optional).

Syntax

C++

```
int WSAAPI WSAEnumNetworkEvents(
  SOCKET             s,
  WSAEVENT           hEventObject, //用于标识需要复位的相应事件对象句柄
  LPWSANETWORKEVENTS lpNetworkEvents //WSANETWORKEVENTS结构的数组
);
```

WSAEnumNetworkEvents()函数获取给定套接字上发生的网络事件，可以使用该函数来发现自上次调用该函数后指定套接字上发生的网络事件。

hEventObject: 如果指定了此参数，函数会重置这个事件对象的状态

lpNetWorkEvents:数组，每一个元素记录了一个网络事件和相应的错误代码

如果成功则返回0，否则将返回SOCKET_ERROR错误

WSANETWORKEVENTS structure

2018/12/05 • 2 分钟阅读时长

The **WSANETWORKEVENTS** structure is used to store a socket's internal information about network events.

Syntax

C++ 复制

```
typedef struct _WSANETWORKEVENTS {
  long lNetworkEvents;
  int  iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, *LPWSANETWORKEVENTS;
```

Members

`lNetworkEvents` //已经发生的网络事件

Indicates which of the FD_XXX network events have occurred.

`iErrorCode` //数组每个成员对应一个网络事件的出错代码

Array that contains any associated error codes, with an array index that corresponds to the position of event bits in **lNetworkEvents**. The identifiers FD_READ_BIT, FD_WRITE_BIT and others can be used to index the **iErrorCode** array.

WSAEventSelect模型基本流程

- 1)初始化Windows Socket环境，并创建用于网络通信的套接字
- 2)创建事件对象
- 3)将新建的事件对象与等待网络事件的套接字相关联，并注册改套接字关心的网络事件集合
- 4)调用WSAWaitForMultipleEvents()等待素有事件对象上发生的注册的网络事件
- 5)如果有事件发生，使用WSAEnumNetworkEvents()函数了解所发生的网络事件，从而进行相应的处理。

基于WSAEventSelect模型的套接字通信服务器示例

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>

#pragma  comment(lib,"ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT 27015    //默认服务器端口号为27015

int _tmain(int argc, _TCHAR* argv[])
{
    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;
    char strIP[16];

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    // 创建用于监听的套接字
    ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (ServerSocket == INVALID_SOCKET)
    {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
```

```
// 为套接字绑定地址和端口号
SOCKADDR_IN addrServ;
addrServ.sin_family = AF_INET;
addrServ.sin_port = htons(DEFAULT_PORT);          // 监听端口为DEFAULT_PORT
addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
if (iResult == SOCKET_ERROR)
{
    printf("bind failed with error: %d\n", WSAGetLastError());
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}

// 监听套接字
iResult = listen(ServerSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR)
{
    printf("listen failed !\n");
    closesocket(ServerSocket);
    WSACleanup();
    return -1;
}
printf("TCP server starting\n");
```

```
// 创建事件对象，并关联到套接字ServerSocket上，注册FD_READ事件
WSAEVENT Event = WSACreateEvent();
int iIndex = 0, i;
int iEventTotal = 0;
// 事件句柄和套节字句柄表
WSAEVENT eventArray[WSA_MAXIMUM_WAIT_EVENTS];
SOCKET sockArray[WSA_MAXIMUM_WAIT_EVENTS];
WSAEventSelect(ServerSocket, Event, FD_ACCEPT);
// 将新建的事件Event保存到eventArray数组中
eventArray[iEventTotal] = Event;
// 将套接字ServerSocket保存到sockArray数组中
sockArray[iEventTotal] = ServerSocket;
iEventTotal++;
```

```
    // 处理网络事件

    sockaddr_in addrClient;
    int addrClientlen = sizeof(sockaddr_in);
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    while (TRUE)
    {
        // 在所有事件对象上等待，只要有一个事件对象变为已授信状态，则函数返回
        iIndex = WSAWaitForMultipleEvents(iEventTotal, eventArray, FALSE, WSA_INFINITE, FALSE);
        // 对每个事件调用WSAWaitForMultipleEvents函数，以便确定它的状态
        //发生的事件对象的索引，一般是句柄数组中最前面的那一个，然后再用循环依次处理后面的事件对象
        iIndex = iIndex - WSA_WAIT_EVENT_0;
        for (i = iIndex; i < iEventTotal; i++)
        {
            iResult = WSAWaitForMultipleEvents(1, &eventArray[i], TRUE, 1000, FALSE);
            if (iResult == WSA_WAIT_FAILED || iResult == WSA_WAIT_TIMEOUT)
            {
                continue;
            }
            else { ... }
        }
    }
    return 0;
}
```

```
else
{
    // 获取到来的通知消息，WSAEnumNetworkEvents函数会自动重置受信事件
    WSANETWORKEVENTS newevent;
    WSAEnumNetworkEvents(sockArray[i], eventArray[i], &newevent);
    if (newevent.lNetworkEvents & FD_ACCEPT)                // 处理FD_ACCEPT通知消息
    {
        if (newevent.iErrorCode[FD_ACCEPT_BIT] == 0)        // 如果处理FD_ACCEPT消息时没有错误
        {
            if (iEventTotal > WSA_MAXIMUM_WAIT_EVENTS)          // 连接太多，暂时不处理
            {
                printf(" Too many connections! \n");
                continue;
            }
            // 接收连接请求，得到与客户端进行通信的套接字AcceptSocket
            SOCKET AcceptSocket = accept(sockArray[i], (sockaddr FAR*) & addrClient, &addrClientlen);
            printf("接收到新的连接：%s\n", inet_ntop(AF_INET, &(addrClient.sin_addr), strIP, 16));
            WSAEVENT newEvent1 = WSACreateEvent();              // 为新套接字创建事件对象
            // 将新建的事件对象newEvent1关联到套接字AcceptSocket上，注册FD_READ|FD_CLOSE|FD_WRITE网络事件
            WSAEventSelect(AcceptSocket, newEvent1, FD_READ | FD_CLOSE | FD_WRITE);
            // 将新建的事件newEvent1保存到eventArray数组中
            eventArray[iEventTotal] = newEvent1;
            // 将新建的套接字sNew保存到sockArray数组中
            sockArray[iEventTotal] = AcceptSocket;
            iEventTotal++;
        }
    }
    if (newevent.lNetworkEvents & FD_READ) { ... }
    if (newevent.lNetworkEvents & FD_CLOSE) { ... }
    if (newevent.lNetworkEvents & FD_WRITE) { ... }
}
```

```
if (newevent.lNetworkEvents & FD_READ)                    // 处理FD_READ通知消息
{
    if (newevent.iErrorCode[FD_READ_BIT] == 0)        // 如果处理FD_READ消息时没有错误
    {
        //有数据到达
        memset(recvbuf, 0, recvbuflen);
        iResult = recv(sockArray[i], recvbuf, recvbuflen, 0);
        if (iResult > 0)
        {
            //情况1：成功接收到数据
            printf("\nBytes received: %d\n", iResult);
            //处理数据请求
            //....
        }
        else
        {
            //情况2：接收失败
            printf("recv failed with error: %d\n", WSAGetLastError());
            closesocket(sockArray[i]);
        }
    }
}
```

```
if (newevent.lNetworkEvents & FD_CLOSE)          // 处理FD_CLOSE通知消息
{
    //进行套接字关闭
    printf("Current Connection closing...\n");
    closesocket(sockArray[i]);
}
if (newevent.lNetworkEvents & FD_WRITE)          // 处理FD_WRITE通知消息
{
    //进行数据发送
}
```

WSAEventSelect模型评价

- WSAEventSelect模型不依赖于消息，所以可以在没有窗口的环境下比较简单地实现对网络通信的异步操作。
- 该模型的缺点是等待的事件对象的总数是有限制的（每次只能等待64个事件）

重叠I/O模型

- 重叠I/O是Win32文件操作的一项技术，其基本设计思想是允许应用程序使用重叠数据结构一次投递一个或者多个异步I/O请求。
- Windows引进了重叠I/O的概念，它能够同时以多线程处理多个I/O,而且系统内部对I/O的处理在性能上有很大的优化。
- 重叠模型的核心是一个重叠数据结构WSAOVERLAPPED,该结构与OVERLAPPED结构兼容。
- 至于为什么叫Overlapped？Jeffrey Richter的解释是因为“执行I/O请求的时间与线程执行其他任务的时间是重叠(overlapped)的”

WSAOVERLAPPED structure

2018/12/05 • 2 分钟阅读时长

The **WSAOVERLAPPED** structure provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion. The **WSAOVERLAPPED** structure is compatible with the Windows OVERLAPPED structure.

Syntax

C++ 复制

```
typedef struct _WSAOVERLAPPED {
  DWORD     Internal;     //系统内部使用的字段
  DWORD     InternalHigh; //系统内部使用的字段
  DWORD     Offset;       //使用套接字情况下忽略该字段
  DWORD     OffsetHigh;   //使用套接字情况下忽略该字段
  WSAEVENT hEvent;        //关联一个事件对象句柄
} WSAOVERLAPPED, *LPWSAOVERLAPPED;
```

重叠I/O模型

- 在重叠I/O模式下，对套接字的读写调用会立即返回，这时候程序可以去做其他的工作，系统会自动完成具体的I/O操作。
- 应用程序可以同时发出多个读写调用，当系统完成I/O操作时，会将WSAOVERLAPPED中的hEvents置为授信状态，可以通过调用WSAWaitForMultipleEvents()函数来等待这个I/O通知完成，在得到通知信号后，就可以调用WSAGetOverlappedResult()函数来查询I/O操作的结果，并进行相关处理。
- WSAOVERLAPPED结构在一个重叠I/O请求的初始化及其后续的完成之间，提供了一种沟通或通信机制。

重叠I/O模型相关函数

- 1)套接字创建函数：WSASocket()
- 2)数据发送函数：WSASend()和WSASendTo()
- 3)数据接收函数：WSARecv()和WSARecvFrom()
- 4)重叠操作结果获取函数：GetOverlappedResult()

WSASocket

```
SOCKET WSASocket(
  int af,
  int type,
  int protocol,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  GROUP g,
  DWORD dwFlags
);
```

//协议信息，与前三个参数互斥使用，即其为NULL时，前三个参数才生效

//套接字组，保留，目前未定义用法

//声明一组对套接字属性的描述，在重叠I/O模型中，该参数需要被设置为WSA_FLAG_OVERLAPPED

详细

如果函数执行成功，则返回新建套接字的句柄，否则返回INVALID_SOCKET

注意，工程项目字符集要设定为Unicode，否则编译通不过。

WSASend()和WSASendTo()

- WinSock环境下，WSASend()函数和WSASendTo()函数提供了在重叠套接字上进行数据发送的能力，并在以下两个方面有所增强：
  - 1)用于重叠socket上，完成重叠发送操作
  - 2)一次发送多个缓冲区中的数据，完成集中写入操作。

WSASend()

```
int WSASend(
  SOCKET s,
  LPWSABUF lpBuffers,                  //指向WSABUF数组的指针
  DWORD dwBufferCount,                 //lpBuffers指向的数组的成员数
  LPDWORD lpNumberOfBytesSent,//一个返回值
  DWORD dwFlags,                       //标志位，同send()调用的flag域
  LPWSAOVERLAPPED lpOverlapped,        //指向WSAOVERLAPPED结构的指针
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine //指向完成例程，即发送操作完成后调用的完成例程
);
```

如果重叠操作立即完成，则WSASend()函数返回0，并且参数lpNumberOfBytesSent被更新为发送数据的字节数；

如果重叠操作被成功初始化，并且将在稍后完成，则WSASend()函数返回SOCKET_ERROR,错误代码为WSA_IO_PENDING。

当重叠操作完成后，可以通过下面两种方式获取传输数据的数量：

1)如果指定了完成例程，则通过完成例程的cbTransferred参数获取;

2)通过WSAGetOverlappedResult()函数的lpcbTransfer参数获取

WSABUF structure

2018/12/05 • 2 分钟阅读时长

The **WSABUF** structure enables the creation or manipulation of a data buffer used by some Winsock functions.

Syntax

C++

```
typedef struct _WSABUF {
  ULONG len;
  CHAR  *buf;
} WSABUF, *LPWSABUF;
```

Members

`len`

The length of the buffer, in bytes.

`buf`

A pointer to the buffer.

WSASendTo

```
int WSAAPI WSASendTo(
  SOCKET                             s,
  LPWSABUF                           lpBuffers,
  DWORD                              dwBufferCount,
  LPDWORD                            lpNumberOfBytesSent,
  DWORD                              dwFlags,
  const sockaddr                     *lpTo,  //指向sockaddr结构的目的地址
  int                                iTolen, //lpTo指向的数据结构的长度
  LPWSAOVERLAPPED                    lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

WSASendTo()提供了在非连接模式下使用重叠I/O进行数据发送的能力。

WSARecv()和WSARecvFrom()

- 在WinSock环境下，WSARecv()函数和WSARecvFrom()函数提供了在重叠套接字上进行数据接收的能力，并在以下两个方面有所增强：
- 1)用于重叠socket，完成重叠接收的操作
- 2)一次将数据接收到多个缓冲区中，完成集中读出操作。

WSARecv()

```
int WSAAPI WSARecv(
  SOCKET                             s,
  LPWSABUF                           lpBuffers,  //指向WSABUF结构数组的指针
  DWORD                              dwBufferCount,//记录lpBuffers数组中的WSABUF结构的数目
  LPDWORD                            lpNumberOfBytesRecvd, //一个返回值
  LPDWORD                            lpFlags,  //标志位，与recv()调用的flag域类似
  LPWSAOVERLAPPED                    lpOverlapped,  //指向WSAOVERLAPPED结构的指针
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine //指向完成例程，即接收操作完成后调用的完成例程
);
```

如果重叠操作立即完成，则WSARecv()函数返回0，并且参数lpNumberOfBytesRecvd被更新为接收数据的字节数；

如果重叠操作被成功初始化，并且将在稍后完成，则返回SOCKET_ERROR, 错误代码为WSA_IO_PENDING

WSARecvFrom()

```
int WSAAPI WSARecvFrom(
  SOCKET                             s,
  LPWSABUF                           lpBuffers,
  DWORD                              dwBufferCount,
  LPDWORD                            lpNumberOfBytesRecvd,
  LPDWORD                            lpFlags,
  sockaddr                           *lpFrom, //返回值，指向sockadrr结构
  LPINT                              lpFromlen,  //指向来源地址长度的指针
  LPWSAOVERLAPPED                    lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

WSARecvFrom()函数提供了在非连接模式下使用重叠I/O进行数据接收的能力

WSAGetOverlappedResult()

```
BOOL WSAAPI WSAGetOverlappedResult(
  SOCKET          s,
  LPWSAOVERLAPPED lpOverlapped,//指向重叠操作开始时指定的WSAOVERLAPPED结构
  LPDWORD         lpcbTransfer,//本次重叠操作实际接收(或发送)的字节数
  BOOL            fWait,    //指定函数是否等待挂起的重叠操作结束
  LPDWORD         lpdwFlags //存放完成状态的附加标志位
);
```

如果函数成功，则返回值为TRUE。它意味着重叠操作已经完成，lpcbTransfer所指向的值已经被刷新。
如果函数失败，则返回值为FALSE。它意味着要么重叠操作未完成，要么由于一个或多个参数的错误导致无法决定完成状态。失败时lpcbTransfer指向的值不会被刷新。

使用事件通知方式进行重叠I/O

- 当I/O完成时，系统更改WSAOVERLAPPED结构对应的事件对象的授信状态，使其从“未授信”变成“已授信”。
- 调用WSAWaitForMultipleEvents()函数，从而判断出一个(或一些)重叠I/O在什么时候完成。
- 通过WSAWaitForMultipleEvents()函数返回的索引可以知道这个重叠I/O完成事件是在哪个Socket上发生的。

使用事件通知方式进行重叠I/O

- 1)套接字初始化，设置为重叠I/O模式
- 2)创建套接字网络事件对应的用户事件对象
- 3)初始化重叠结构，为套接字关联事件对象
- 4)异步接收数据，无论能否接收到数据，都会直接返回；
- 5)调用WSAWaitForMultiEvents()函数在所有事件对象上等待，只要有一个事件对象变为已授信状态，则函数返回；
- 6)调用WSAGetOverlappedResult()函数获取套接字上的重叠操作的状态，并保存到重叠结构中。
- 7)根据重置事件的状态进行处理
- 8)重置已授信的事件对象、重叠结构、标志位和缓冲区
- 9)回到步骤4

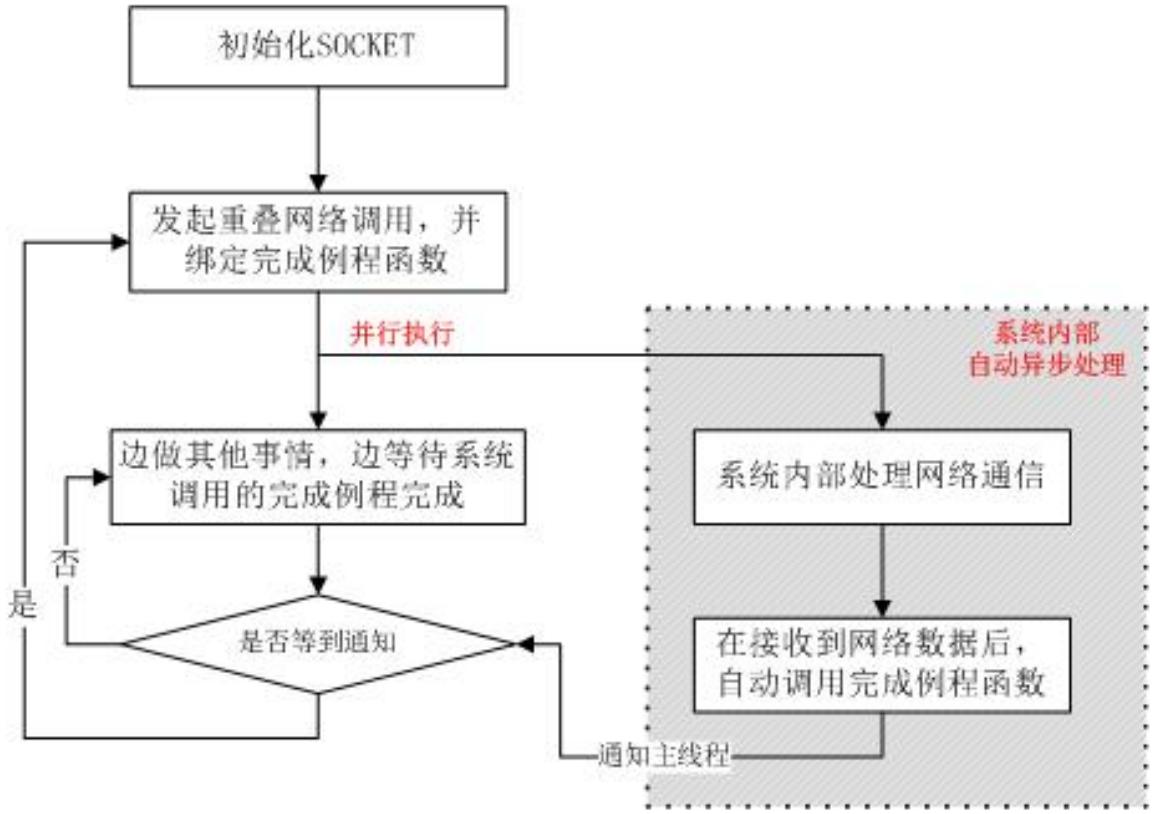
使用完成例程方式进行重叠I/O

- 完成例程的函数原型

```
void CALLBACK CompletionRoutine(
  IN DWORD dwError,  //重叠操作的完成状态
  IN DWORD cbTransferred,//传送的字节数
  IN LPWSAOVERLAPPED lpOverlapped,//重叠操作结构
  IN DWORD dwFlags //没定义用法，通常置0
);
```

WSARecv()、WSARecvFrom()、WSASend()、WSASendTo()最后一个参数lpCompletionROUTINE是个可选指针。若指定此参数，则hEvent参数将会被忽略，上下文信息将传送给完成例程函数，然后调用WSAGetOverlappedResult()函数查询重叠操作结果。

使用完成例程方式进行重叠I/O



使用完成例程方式进行重叠I/O

- 1)套接字初始化,设置为重叠I/O模型
- 2)初始化重叠结构
- 3)异步传输数据，将重叠结构作为输入参数，并指定一个完成例程对应于数据传输后的处理;
- 4)调用WSAWaitForMultiEvents()函数或SleepEx()函数将自己的线程置为一种可警告等待状态，等待一个重叠I/O请求完成，重叠请求完成后，完成例程会自动执行，在完成例程内，可随一个完成例程一起投递另一个重叠I/O操作。
- 5)回到步骤3

使用事件通知方式进行重叠I/O的示例

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>

#pragma  comment(lib,"ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT 27015    //默认服务器端口号为27015

int _tmain(int argc, _TCHAR* argv[])
{
    // 声明和初始化变量
    WSABUF DataBuf;                          // 发送和接收数据的缓冲区结构体
    char buffer[DEFAULT_BUFLEN];             // 缓冲区结构体DataBuf中
    DWORD EventTotal = 0,                    // 记录事件对象数组中的数据
        RecvBytes = 0,                             // 接收的字节数
        Flags = 0,                                 // 标识位
        BytesTransferred = 0;                  // 在读、写操作中实际传输的字节数

    // 数组对象数组
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;     // 重叠结构体

    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;
    char strIP[16];

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }
```

```
// 创建用于监听的套接字
ServerSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_IP, NULL, 0, WSA_FLAG_OVERLAPPED);
if (ServerSocket == INVALID_SOCKET)
{
    printf("WSASocket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

// 为套接字绑定地址和端口号
SOCKADDR_IN addrServ;
addrServ.sin_family = AF_INET;
addrServ.sin_port = htons(DEFAULT_PORT);        // 监听端口为DEFAULT_PORT
addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
if (iResult == SOCKET_ERROR)
{
    printf("bind failed with error: %d\n", WSAGetLastError());
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}

// 监听套接字
iResult = listen(ServerSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR)
{
    printf("listen failed !\n");
    closesocket(ServerSocket);
    WSACleanup();
    return -1;
}
printf("TCP server starting\n");
```

```
// 创建事件对象，建立重叠结构
EventArray[EventTotal] = WSACreateEvent();
ZeroMemory(buffer, DEFAULT_BUFLEN);
ZeroMemory(&AcceptOverlapped, sizeof(WSAOVERLAPPED));        // 初始化重叠结构
AcceptOverlapped.hEvent = EventArray[EventTotal];            // 设置重叠结构中的hEvent字段
DataBuf.len = DEFAULT_BUFLEN;                                // 设置缓冲区
DataBuf.buf = buffer;
EventTotal++;
sockaddr_in addrClient;
int addrClientlen = sizeof(sockaddr_in);
SOCKET AcceptSocket;
```

```
    // 循环处理客户端的连接请求
    while (true)
    {
        AcceptSocket = accept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen);
        if (AcceptSocket == INVALID_SOCKET)
        {
            printf("accept failed !\n");
            closesocket(ServerSocket);
            WSACleanup();
            return 1;
        }
        printf("接收到新的连接：%s\n", inet_ntop(AF_INET, &(addrClient.sin_addr), strIP, 16));
        // 处理在套接字上接收到数据
        while (true) { ... }
    }
    return 0;
}
```

```
// 处理在套接字上接收到数据
while (true)
{
    DWORD Index;        // 保存处于授信状态的事件对象句柄
    // 调用WSARecv()函数在ServerSocket套接字上以重叠I/O方式接收数据，保存到DataBuf缓冲区中
    iResult = WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes, &Flags, &AcceptOverlapped, NULL);
    if (iResult == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
            printf("Error occured at WSARecv():%d\n", WSAGetLastError());
    }
    // 等待完成的重叠I/O调用
    Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA_INFINITE, FALSE);
    // 决定重叠事件的状态
    WSAGetOverlappedResult(AcceptSocket, &AcceptOverlapped, &BytesTransferred, FALSE, &Flags);
    // 如果连接已经关闭，则关闭AcceptSocket套接字
    if (BytesTransferred == 0) {
        printf("Closing Socket %d\n", AcceptSocket);
        closesocket(AcceptSocket);
        break;
    }
    //成功接收到数据
    printf("Bytes received: %d\n", BytesTransferred);
    //处理数据请求
    //....

    // 重置已授信的事件对象
    WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
    // 重置Flags变量和重叠结构
    Flags = 0;
    ZeroMemory(&AcceptOverlapped, sizeof(WSAOVERLAPPED));
    ZeroMemory(buffer, DEFAULT_BUFLEN);
    AcceptOverlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];
    // 重置缓冲区
    DataBuf.len = DEFAULT_BUFLEN;
    DataBuf.buf = buffer;
}
```

完成例程方式进行重叠I/O示例

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>

#pragma  comment(lib,"ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT  27015   //默认服务器端口号为27015

// I/O操作的数据
typedef struct
{
    WSAOVERLAPPED overlap;                  // 重叠结构体
    WSABUF Buffer;                          // 缓冲区对象
    char szMessage[DEFAULT_BUFLEN]; // 缓冲区字符数组
    DWORD NumberOfBytesRecvd;               // 接收字节数
    DWORD Flags;                            // 标识位
    SOCKET sClient;                         // 套接字
} PER_IO_OPERATION_DATA, * LPPER_IO_OPERATION_DATA;
```

```
// 完成例程函数
void CALLBACK CompletionROUTINE(DWORD dwError,  // 重叠操作的完成状态
    DWORD cbTransferred,                        // 发送的字节数
    LPWSAOVERLAPPED lpOverlapped,               // 指定重叠操作的结构体
    DWORD dwFlags)                              // 标识位
{
    LPPER_IO_OPERATION_DATA lpPerIOData = (LPPER_IO_OPERATION_DATA)lpOverlapped;    // 保存I/O操作的数据
    // 如果发生错误或者没有数据传输，则关闭套接字，释放资源
    if (dwError != 0 || cbTransferred == 0)
    {
        closesocket(lpPerIOData->sClient);
        HeapFree(GetProcessHeap(), 0, lpPerIOData);
    }
    else
    {
        //成功接收到数据
        printf("Bytes received: %d\n", cbTransferred);
        //处理数据请求
        //....

        lpPerIOData->szMessage[cbTransferred] = '\0';     // 标识接收数据的结束
        // 执行另一个异步操作，接收数据
        memset(&lpPerIOData->overlap, 0, sizeof(WSAOVERLAPPED));
        lpPerIOData->Buffer.len = DEFAULT_BUFLEN;
        lpPerIOData->Buffer.buf = lpPerIOData->szMessage;
        // 接收数据
        WSARecv(lpPerIOData->sClient,
            &lpPerIOData->Buffer,
            1,
            &lpPerIOData->NumberOfBytesRecvd,
            &lpPerIOData->Flags,
            &lpPerIOData->overlap,
            CompletionROUTINE);
    }
}
```

完成例程实际上和投递进程处于同一个进程

```
int _tmain(int argc, _TCHAR* argv[])
{
    // 声明和初始化变量
    DWORD   RecvBytes = 0,                  // 接收的字节数
        Flags = 0,                          // 标识位
        BytesTransferred = 0;               // 在读、写操作中实际传输的字节数

    // 数组对象数组
    WSADATA wsaData;
    int iResult;
    SOCKET ServerSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;
    char strIP[16];

    // 初始化 Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0)
    {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }
```

```
// 创建用于监听的套接字
ServerSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_IP, NULL, 0, WSA_FLAG_OVERLAPPED);
if (ServerSocket == INVALID_SOCKET)
{
    printf("WSASocket failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

// 为套接字绑定地址和端口号
SOCKADDR_IN addrServ;
sockaddr_in addrClient;
int addrClientlen = sizeof(sockaddr_in);
addrServ.sin_family = AF_INET;
addrServ.sin_port = htons(DEFAULT_PORT);         // 监听端口为DEFAULT_PORT
addrServ.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
iResult = bind(ServerSocket, (const struct sockaddr*) & addrServ, sizeof(SOCKADDR_IN));
if (iResult == SOCKET_ERROR)
{
    printf("bind failed with error\n");
    closesocket(ServerSocket);
    WSACleanup();
    return 1;
}

// 监听套接字
iResult = listen(ServerSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR)
{
    printf("listen failed !\n");
    closesocket(ServerSocket);
    WSACleanup();
    return -1;
}
printf("TCP server starting\n");
```

```
// 循环处理客户端的连接请求
while (true)
{
    AcceptSocket = accept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen);
    if (AcceptSocket == INVALID_SOCKET)
    {
        printf("accept failed !\n");
        closesocket(ServerSocket);
        WSACleanup();
        return 1;
    }
    printf("\n接收到新的连接：%s\n", inet_ntop(AF_INET, &(addrClient.sin_addr), strIP, 16));

    // 处理在套接字上接收到数据
    LPPER_IO_OPERATION_DATA lpPerIOData = NULL;      // 保存I/O操作的数据
    // 为新的连接执行一个异步操作
    // 为LPPER_IO_OPERATION_DATA结构体分配堆空间
    lpPerIOData = (LPPER_IO_OPERATION_DATA)HeapAlloc(
        GetProcessHeap(),
        HEAP_ZERO_MEMORY,
        sizeof(PER_IO_OPERATION_DATA));
    // 初始化结构体lpPerIOData
    lpPerIOData->Buffer.len = DEFAULT_BUFLEN;
    lpPerIOData->Buffer.buf = lpPerIOData->szMessage;
    lpPerIOData->sClient = AcceptSocket;
    ZeroMemory(lpPerIOData->Buffer.buf, DEFAULT_BUFLEN);
    ZeroMemory(&(lpPerIOData->overlap), sizeof(WSAOVERLAPPED));
    // 接收数据
    iResult = WSARecv(lpPerIOData->sClient,      // 接收数据的套接字
        &lpPerIOData->Buffer,                    // 接收数据的缓冲区
        1,                                       // 缓冲区对象的数量
        &lpPerIOData->NumberOfBytesRecvd,        // 接收数据的字节数
        &lpPerIOData->Flags,                     // 标识位
        &lpPerIOData->overlap,                   // 重叠结构
        CompletionROUTINE);                      // 完成例程函数，将会在接收数据完成的时候进行相应的调用
    if (iResult == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
            printf("Error occured at WSARecv():%d\n", WSAGetLastError());
    }
    SleepEx(1000, TRUE);
}
}
```

//accpet导致的线程阻塞实际上是使线程进入了警觉等待状态

//这个SleepEx实际上可以删掉

SleepEx()

```
DWORD SleepEx(
  DWORD dwMilliseconds,  //休眠时间，毫秒
  BOOL  bAlertable   //是否因I/O事件完成停止休眠
);
```

bAlertable: 为FALSE，则只有休眠时间到函数才返回。
为TRUE,则以下三种情况之一发生，函数就返回。
1）SleepEx与扩展I/O函数(ReadFileEx or WriteFileEx)是在同一个线程，当休眠超时或I/O回调函数出现 ,函数就会立即返回
2)如果APC(异步函数调用)被插入线程，该函数不论当前线程是否超时都会立即返回，而且APC函数也会被调用。
3) 休眠超时

当线程休眠超时函数返回0。
如果函数返是由于I/O回调函数导致，那么返回值是WAIT_IO_COMPLETION，这只会出现在当bAlertable设置TRUE时的情况。
如果是其他返回值，那就坏事了，说明网络通信出现了其他异常，程序就可以报错退出了……

重叠I/O模型评价

- 使用重叠模型的应用程序通知缓冲区收发系统直接使用数据。
- 重叠I/O的效率优势就在于减少了一次从I/O缓冲区到应用程序缓冲区的拷贝

完成端口模型

- 完成端口是Windows下伸缩性最好的I/O模型。
- 完成端口内部提供了线程池的管理，可以避免反复创建线程的开销。

传统并发服务器设计的不足

- 1)服务器能够创建的线程数量是有限的。
- 2)操作系统对线程的管理和调度会占用CPU资源，进而降低系统的响应速度。
- 3)频繁地创建线程和结束线程涉及反复的资源分配与释放，会浪费大量的系统资源。
- 解决方法：线程池

线程池

- 线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件）,则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。
- 线程池的使用即限制了工作线程的数量，又避免了反复创建线程的开销，减少了线程调度的开销，从而提高了服务程序的性能。

完成端口模型

- 完成端口模型使用线程池对线程进行管理，预先创建和维护线程，并规定了并行线程的数量。
- 可以把完成端口看成是系统维护的一个队列，操作系统把重叠I/O操作完成的事件通知放到该队列里。当某项I/O操作完成时，系统会向服务器完成端口发送一个I/O完成数据报。应用程序收到I/O完成数据包后，完成端口队列中的一个线程被唤醒，为客户提供服务。服务完成后，该线程会继续在完成端口上等待后续I/O请求事件的通知。
- 一个套接字被创建后，可以在任何时刻和一个完成端口联系起来。

工作线程与完成端口

- 要创建多少个线程为完成端口提供服务？
- 依赖于程序的总体设计情况。
- 理想情况下，对应一个CPU创建一个线程。
- 考虑到阻塞操作情况，可以多创建几个线程，以便在发生阻塞的时候充分发挥系统的潜力。

单句柄数据和单I/O操作数据

- 单句柄数据对应着与某个套接字关联的数据，用来把客户数据和对应的完成通知关联起来。
- 可以为单句柄数据定义一个数据结构来保存其关联的信息
- 单I/O操作数据，记录了每次I/O通知信息，可以是追加到一个OVERLAPPED结构末尾的任意长度字节。
- 可以为单I/O操作数据定义一个数据结构来保存具体的操作类型和数据，并将OVERLAPPED结构作为新结构的第一个成员。

CreateIoCompletionPort()

```
HANDLE CreateIoCompletionPort(
  HANDLE    FileHandle,  //套接字或INVALID_HANDLE_VALUE
  HANDLE    ExistingCompletionPort, //已存在的完成端口句柄
  ULONG_PTR CompletionKey,   //指向单句柄数据的指针
  DWORD     NumberOfConcurrentThreads //允许并发处理I/O完成数据包的最大线程数量，0表示等同于处理器个数。如果ExistingCompletionPort不为NULL,则该参数就会被忽略
);
```

这个函数用于两个明显不同的目的：

1.用于创建一个完成端口对象（第4个参数）

2.用于将一个句柄同完成端口关联起来(第2个参数)

FileHandle:是重叠I/O关联的套接字，如果被指定为INVALIND_HANDLE_VALUE,则创建一个与套接字无关的I/O完成端口。此时ExistingCompletionPort 必须为NULL,且CompletionKey 参数被忽略。

ExistingCompletionPort:是已经存在的完成端口。如果指定一个已存在的完成端口句柄，则函数将其关联到FileHandle指定的套接字上。如果为NULL，则新建一个与FileHandle套接字关联的新的IO完成端口。

函数执行成功，则返回IO完成端口句柄，失败则返回NULL.

GetQueuedCompletionStatus()

- 实现从指定的IOCP获取CP。当CP队列为空时，对此函数的调用将被阻塞，而不是一直等待I/O的完成。当CP队列不为空时，被阻塞的线程将以后进先出（LIFO）顺序被释放。对于IOCP机制，它允许多线程并发调用GetQueuedCompletionStatus函数，最大并发数是在调用CreateIoCompletionPort函数时指定的，超出最大并发数的调用线程，将被阻塞。

GetQueuedCompletionStatus()

```
BOOL GetQueuedCompletionStatus(
  HANDLE       CompletionPort, //完成端口对象句柄
  LPDWORD      lpNumberOfBytesTransferred,//获取已经完成的I/O操作中传输的字节数
  PULONG_PTR   lpCompletionKey, //获取与已经完成的IO操作的套接字相关联的单句柄数据
  LPOVERLAPPED *lpOverlapped, //在完成的IO操作开始时指定的重叠结构地址，及单I/O操作数据
  DWORD        dwMilliseconds//函数在完成端口上等待时间
);
```

dwMilliseconds:如果在等待时间内没有I/O操作完成通知包到达完成端口，则函数返回FALSE,lpOverlapped的值为NULL。如果该参数为INFINITE,则函数不会出现调用超时的情况。如果该参数为0，则函数立即返回。

如果函数从完成端口获取到一个成功的IO操作完成通知包，则函数返回非0值。

如果函数从完成端口获取到一个失败的IO操作完成通知包，则函数返回0。

如果函数调用超时，则返回0。

完成端口模型的基本流程

- 主程序
  - 1)判断系统中安装了多少个处理器，创建n个工作线程,n一般为处理器个数。工作线程的主要功能是检测完成端口的状态，如果有来自客户的数据，则接收数据，处理请求。
  - 2)初始化Windows Sockets环境，初始化套接字。
  - 3)创建完成端口对象，将待处理网络请求的套接字与完成端口对象关联；
  - 4)异步接收数据，无论能否接收到数据，都会直接返回。
- 工作线程
  - 1)调用GetQueuedCompletionStatus()函数检查完成端口的状态;
  - 2)根据GetQueuedCompletionStatus()返回的数据和状态进行具体的请求处理。

完成端口模型示例

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <tchar.h>
#include <thread>

#pragma  comment(lib,"ws2_32.lib")

#define DEFAULT_BUFLEN 512    //默认缓冲区长度为512
#define DEFAULT_PORT 27015  //默认服务器端口号为27015

// 定义I/O操作的结构体
typedef struct
{
    OVERLAPPED Overlapped;                    // 重叠结构
    WSABUF DataBuf;                           // 缓冲区对象
    CHAR Buffer[DEFAULT_BUFLEN];                  // 缓冲区数组
    DWORD BytesRECV;                          // 接收的字节数
} PER_IO_DATA, * LPPER_IO_DATA;

// 套接字句柄结构体
typedef struct
{
    SOCKET Socket;
} PER_HANDLE_DATA, * LPPER_HANDLE_DATA;
```

```
// 服务器端工作线程
DWORD WINAPI ServerWorkerThread(LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE)CompletionPortID;   // 完成端口句柄
    DWORD BytesTransferred;                                    // 数据传输的字节数
    LPPER_HANDLE_DATA PerHandleData;                    // 套接字句柄结构体
    LPPER_IO_DATA PerIoData;                // I/O操作结构体
    DWORD RecvBytes;                                   // 接收的数量
    DWORD Flags;                                                        // WSARecv()函数中的标识位

    while (TRUE) { ... }
}
```

```
while (TRUE)
{
    // 检查完成端口的状态
    if (GetQueuedCompletionStatus(CompletionPort, &BytesTransferred,
        (LPDWORD)&PerHandleData, (LPOVERLAPPED*)&PerIoData, INFINITE) == 0)
    {
        printf("GetQueuedCompletionStatus failed with error %d\n", GetLastError());
        return 0;  //这里改为continue更合适
    }

    // 如果数据传送完了，则退出
    if (BytesTransferred == 0)
    {
        printf("Closing socket %d\n", PerHandleData->Socket);
        // 关闭套接字
        if (closesocket(PerHandleData->Socket) == SOCKET_ERROR)
        {
            printf("closesocket() failed with error %d\n", WSAGetLastError());
            return 0;  //正常情况不应该return
        }
        // 释放结构体资源
        GlobalFree(PerHandleData);
        GlobalFree(PerIoData);
        continue;
    }
```

```
while (TRUE)
{
    // 检查完成端口的状态
    if (GetQueuedCompletionStatus(CompletionPort, &BytesTransferred,
        (LPDWORD)&PerHandleData, (LPOVERLAPPED*)&PerIoData, INFINITE) == 0) { ... }

    // 如果数据传送完了，则退出
    if (BytesTransferred == 0) { ... }
    // 如果还没有记录接收的数据数量，则将收到的字节数保存在PerIoData->BytesRECV中
    if (PerIoData->BytesRECV == 0)
    {
        PerIoData->BytesRECV = BytesTransferred;
    }

    //成功接收到数据
    printf("\nBytes received: %d\n", BytesTransferred);
    //处理数据请求
    //....
    PerIoData->BytesRECV = 0;
    Flags = 0;
    ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
    PerIoData->DataBuf.len = DEFAULT_BUFLEN;
    PerIoData->DataBuf.buf = PerIoData->Buffer;

    if (WSARecv(PerHandleData->Socket, &(PerIoData->DataBuf), 1, &RecvBytes, &Flags,
        &(PerIoData->Overlapped), NULL) == SOCKET_ERROR)
    {
        if (WSAGetLastError() != ERROR_IO_PENDING)
        {
            printf("WSARecv() failed with error %d\n", WSAGetLastError());
            return 0;
        }
    }
}
```

```cpp
int _tmain(int argc, _TCHAR* argv[])
{
    SOCKADDR_IN InternetAddr;                       // 服务器地址
    SOCKET ServerSocket = INVALID_SOCKET;           // 监听套接字
    SOCKET AcceptSocket = INVALID_SOCKET;           // 与客户端进行通信的套接字
    HANDLE CompletionPort;                          // 完成端口句柄
    SYSTEM_INFO SystemInfo;                         // 获取系统信息（这里主要用于获取CPU数量）
    LPPER_HANDLE_DATA PerHandleData;                // 套接字句柄结构体
    LPPER_IO_DATA PerIoData;                        // 定义I/O操作的结构体
    DWORD RecvBytes;                                // 接收到的字节数
    DWORD Flags;                                    // WSARecv()函数中指定的标识位
    WSADATA wsaData;                                // Windows Socket初始化信息
    DWORD Ret;                                      // 函数返回值
    char strIP[16];

    // 初始化Windows Sockets环境
    if ((Ret = WSAStartup(0x0202, &wsaData)) != 0)
    {
        printf("WSAStartup failed with error %d\n", Ret);
        return -1;
    }
    // 创建新的完成端口
    if ((CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0)) == NULL)
    {
        printf("CreateIoCompletionPort failed with error: %d\n", GetLastError());
        return -1;
    }
    // 获取系统信息
    GetSystemInfo(&SystemInfo);
    // 根据CPU数量启动线程
    for (unsigned int i = 0; i < SystemInfo.dwNumberOfProcessors * 2; i++)
    {
        std::thread thread(ServerWorkerThread, CompletionPort);
        thread.detach();
    }
```

```
    // 创建监听套接字
    if ((ServerSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
        WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        printf("WSASocket() failed with error %d\n", WSAGetLastError());
        return -1;
    }
    // 绑定到本地地址的指定端口
    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(DEFAULT_PORT);
    if (bind(ServerSocket, (PSOCKADDR)&InternetAddr, sizeof(InternetAddr)) == SOCKET_ERROR)
    {
        printf("bind() failed with error %d\n", WSAGetLastError());
        return -1;
    }
    // 开始监听
    if (listen(ServerSocket, 5) == SOCKET_ERROR)
    {
        printf("listen() failed with error %d\n", WSAGetLastError());
        return -1;
    }
    printf("TCP server starting\n");

    // 监听端口打开，就开始在这里循环，一有socket连上，WSAAccept就创建一个socket，
    // 这个socket 和完成端口联上
    sockaddr_in addrClient;
    int addrClientlen = sizeof(sockaddr_in);
    while (TRUE) { ... }
    return 0;
}
```

```
while (TRUE)
{
    // 等待客户连接
    if ((AcceptSocket = WSAAccept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen, NULL, 0)) == SOCKET_ERROR)
    {
        printf("WSAAccept() failed with error %d\n", WSAGetLastError());
        return -1;
    }
    printf("\n接收到新的连接：%s\n", inet_ntop(AF_INET, &(addrClient.sin_addr), strIP, 16));
    // 分配并设置套接字句柄结构体
    if ((PerHandleData = (LPPER_HANDLE_DATA)GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA))) == NULL)
    {
        printf("GlobalAlloc() failed with error %d\n", GetLastError());
        return -1;
    }
    PerHandleData->Socket = AcceptSocket;

    // 将与客户端进行通信的套接字Accept与完成端口CompletionPort相关联
    if (CreateIoCompletionPort((HANDLE)AcceptSocket, CompletionPort, (DWORD)PerHandleData,
        0) == NULL)
    {
        printf("CreateIoCompletionPort failed with error %d\n", GetLastError());
        return -1;
    }
```

```
while (TRUE)
{
    // 等待客户连接
    if ((AcceptSocket = WSAAccept(ServerSocket, (sockaddr FAR*) & addrClient, &addrClientlen, NULL, 0)) == SOCKET_ERROR) { ... }
    printf("\n接收到新的连接：%s\n", inet_ntop(AF_INET, &(addrClient.sin_addr), strIP, 16));
    // 分配并设置套接字句柄结构体
    if ((PerHandleData = (LPPER_HANDLE_DATA)GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA))) == NULL) { ... }
    PerHandleData->Socket = AcceptSocket;

    // 将与客户端进行通信的套接字Accept与完成端口CompletionPort相关联
    if (CreateIoCompletionPort((HANDLE)AcceptSocket, CompletionPort, (DWORD)PerHandleData,
        0) == NULL) { ... }
    // 为I/O操作结构体分配内存空间
    if ((PerIoData = (LPPER_IO_DATA)GlobalAlloc(GPTR, sizeof(PER_IO_DATA))) == NULL)
    {
        printf("GlobalAlloc() failed with error %d\n", GetLastError());
        return -1;
    }
    // 初始化I/O操作结构体
    ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
    PerIoData->BytesRECV = 0;
    PerIoData->DataBuf.len = DEFAULT_BUFLEN;
    PerIoData->DataBuf.buf = PerIoData->Buffer;
    Flags = 0;

    // 接收数据，放到PerIoData中，而perIoData又通过工作线程中的ServerWorkerThread函数取出，
    if (WSARecv(AcceptSocket, &(PerIoData->DataBuf), 1, &RecvBytes, &Flags,
        &(PerIoData->Overlapped), NULL) == SOCKET_ERROR)
    {
        if (WSAGetLastError() != ERROR_IO_PENDING)
        {
            printf("WSARecv() failed with error %d\n", WSAGetLastError());
            return -1;
        }
    }
}
```

完成端口模型评价

- 完成端口模型是应用程序使用线程池处理异步I/O请求的一种机制，在Windows服务平台上比较成熟，是伸缩性最好的I/O模式。
- 当应用程序需要管理上千个套接字时，利用完成端口模型往往可以达到最佳的系统性能。

如何正确关闭I/O完成端口--特别是同时运行一个或多个线程，在几个不同的套接字上执行I/O操作时。要注意的一个主要问题是，在进行重叠I/O操作时，应避免强行释放OVERLAPPED结构。要想不出现这种情况，最好的办法是针对每个套接字句柄，调用closesocket函数，则任何尚未进行的重叠I/O操作都会完成。
一旦所有套接字句柄都已关闭，便须在完成端口上终止所有工作器线程的运行。要想做到这一点可以使用PostQueuedCompletionStatus函数，向每个工作器线程都发送一个特殊的完成数据包。该函数会提示每个线程立即结束并退出

完成端口更完善的例子