网络程序设计

流式套接字编程

流式套接字

- 流式套接字依托TCP协议提供面向连接的、可靠的数据传输服务，该服务将保证数据能够实现无差错、无重复发送，并按顺序接收。
- 基于流的特点，使用流式套接字传输的数据形态是没有记录边界的有序数据流。

流式套接字编程的适用场合

- 流式套接字只适合一对一的数据传输
- 适用场合
  - 大数据量的数据传输应用
  - 可靠性要求高的传输应用

流式套接字的通信过程

服务器通信过程：

①socket初始化；

②创建套接字，指定使用TCP（可靠的传输服务）进行通信；

③指定本地地址和通信端口；

④等待客户端的连接请求；

⑤进行数据传输；

⑥关闭套接字；

⑦结束对windows sockets dll的使用。

流式套接字的通信过程

客户端通信过程：

①socket初始化；

②创建套接字，指定使用TCP（可靠的传输服务）进行通信；

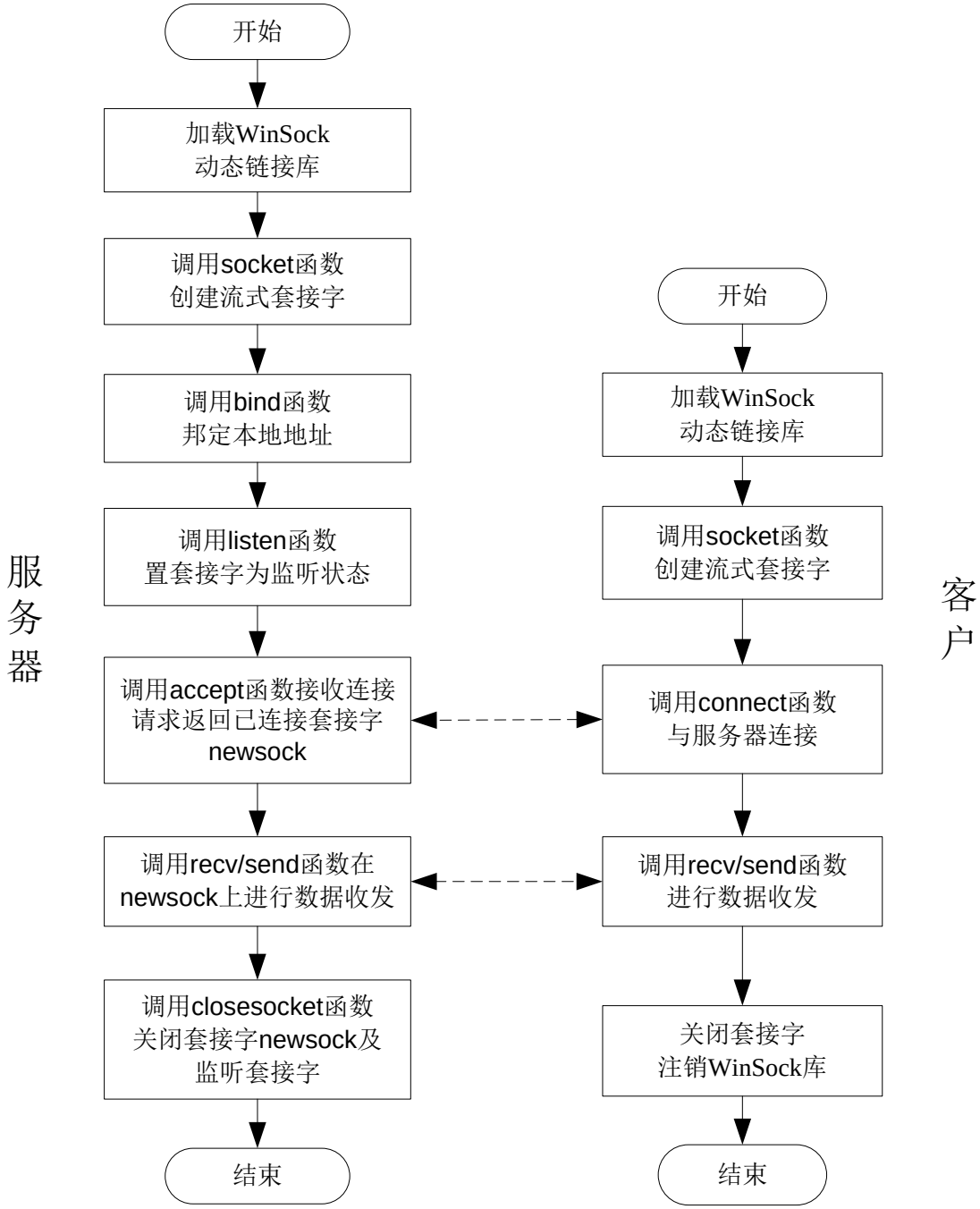
③指定服务器地址和通信端口；

④向服务器发送连接请求；

⑤进行数据传输；

⑥关闭套接字；

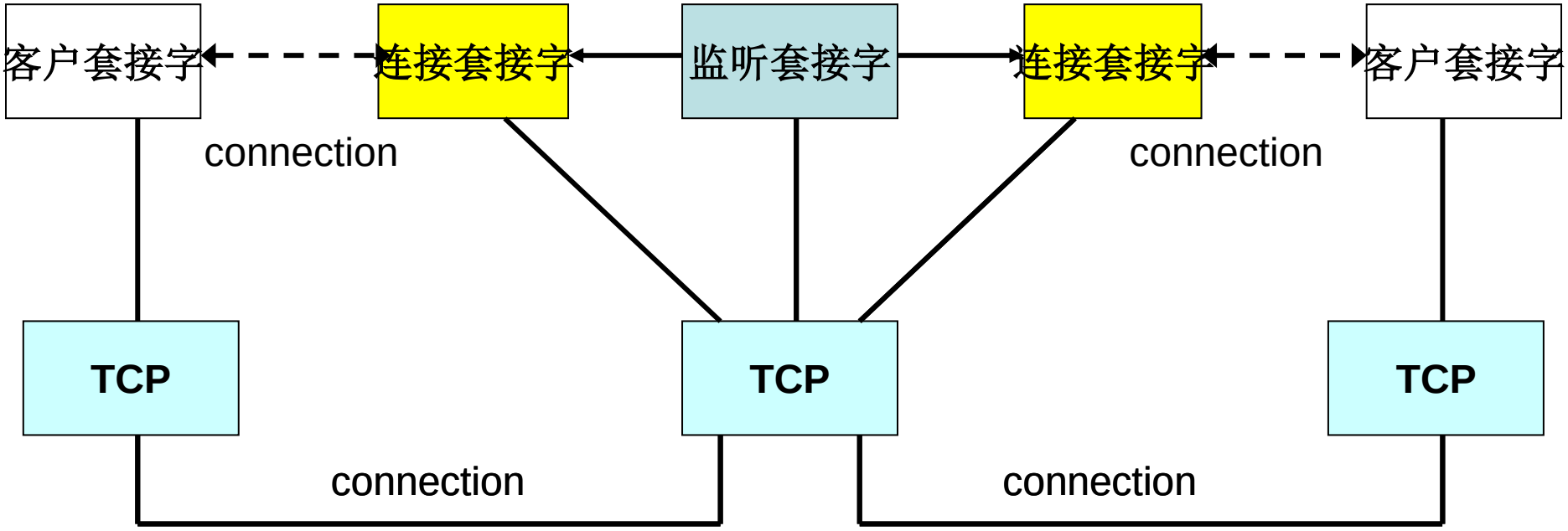
⑦结束对windows sockets dll的使用。


服务器
开始
加载WinSock
动态链接库
调用socket函数
创建流式套接字
调用bind函数
邦定本地地址
调用listen函数
置套接字为监听状态
调用accept函数接收连接
请求返回已连接套接字
newsock
调用recv/send函数在
newsock上进行数据收发
调用closesocket函数
关闭套接字newsock及
监听套接字
结束
客户
开始
加载WinSock
动态链接库
调用socket函数
创建流式套接字
调用connect函数
与服务器连接
调用recv/send函数
进行数据收发
关闭套接字
注销WinSock库
结束

思考

- 1）为什么服务器需要绑定操作，而在客户没有进行绑定操作？客户如何使用唯一的端点地址与服务器通信？
- 2）在服务器和客户的通信过程中、面向连接服务器是如何处理多个客户服务请求的呢？

流式套接字服务器工作原理



流式套接字服务器的工作原理

- 从服务器的并发角度看，服务器的两种设计：
  - 1）循环服务器：一次只服务一个客户，当一个客户服务完后，服务器才会处理另一个客户的服务请求。
  - 2）并发服务：服务器为每一个客户创建一个单独的子进程或线程，同时为多个客户服务

套接字函数的使用方法

- 1. 创建套接字
  - WinSock2主要用两个函数创建套接字
    - socket(): 实现同步传输套接字的创建
    - WSASocket():实现异步传输套接字的创建

相关套接字函数的使用方法

1. 创建套接字——socket()函数

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

   – 用于创建一个套接字。
   – af：标识一个地址家族，通常为AF_INET。
   – type：标识套接字类型，SOCK_STREAM表示流式套字；SOCK_DGRAM表示数据报套接字；SOCK_RAW表示原始套接字。
   – protocol：标识一个特殊的协议被用于这个套接字，通常为0，表示采用默认的TCP/IP协议

- socket()函数返回的套接字描述符的类型符SOCKET，它是Winsock中专门定义的一种新的数据类型，其定义为

  typedef u_int SOCKET;

  是一个无符号整型数。
- 创建套接字失败返回的常量INVALID_SOCKET的定义如下：

  #define INVALID_SOCKET (SOCKET)~0

  是各二进制位全为1的一个无符号整数。不难看出，合法的套接字描述符的取值可以是0~INVALID_SOCKET-1之间的任何值。

```
/*建立一个socket。*/
if ((sock_server = socket(AF_INET,SOCK_STREAM,0))<0) {
    printf("创建套接字失败！ \n");
    return 0;
}
else
{
    printf("socket created .\n");     //socket建立成功。
    printf("socked id: %d \n",sock_server);
}
```

WSASocket

```
SOCKET WSASocket(
  int af,
  int type,
  int protocol,
  LPWSAPROTOCOL_INFO lpProtocolInfo,  //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
  GROUP g,  //套接字组，保留，目前未定义用法
  DWORD dwFlags  //声明一组对套接字属性的描述
);
```

详细

Socket()函数和WSASocket()函数的返回值是通信实例的句柄(handle),类似于文件描述符。
在源于UNIX的系统上，该句柄是一个函数：非负值表示成功，-1表示失败

套接字关闭

- 当利用套接字完成应用程序的网络操作是，调用closesocket()完成套接字的关闭操作。

```
int closesocket(
  SOCKET s
);
```

如果成功，返回0，否则返回-1

套接字绑定

2. 绑定地址——bind函数

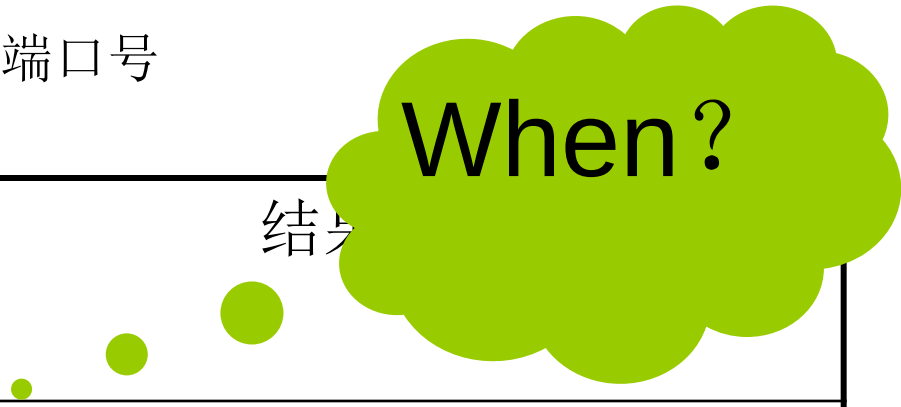
socket()函数在创建套接字时并没有为创建的套接字分配地址，因此服务器软件在创建了监听套接字之后，需要调用bind函数为其指定本机地址、协议和端口号。

int bind(SOCKET s,struct sockaddr *name, int namelen);

- 将套接字绑定到一个已知的地址上。如果函数执行成功，返回值为0，否则为SOCKET_ERROR。
- s：是一个套接字。
- name：是一个sockaddr结构指针，该结构中包含了要绑定的地址和端口号。
- namelen：确定name缓冲区的长度。

关于 bind函数的几点说明

- 地址种类
  - 常规地址：特定主机地址，特定端口号
  - 通配地址： INADDR_ANY，0



进程指定		结
IP地址	端口	
通配地址	0	内核选择IP和端口
通配地址	非0	内核选择IP，进程指定端口
本地IP地址	0	进程指定IP，内核选择端口
本地IP地址	非0	进程指定IP和端口

```
//绑定IP端口
struct sockaddr_in addr；
addr.sin_family =AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
                                    //允许套接字使用本机的任何IP
if(bind(sock_server,(LPSOCKADDR)&addr,sizeof(addr))!=0)
{
    printf("邦定失败！\n");
    return 0;
}
```

关于bind函数的几点说明

- *client端的socket是否需要关联本地地址的？*
  - *不建议*
    - *没有必要使用固定端口号*
    - *强行绑定端口号，可能这个端口号已经被使用，会产生冲突*
  - 当客户调用connect和sendto时，系统会随机选一个未使用的端口号，隐式调用一次bind()

关于bind函数的几点说明

- *如果由系统选择地址或端口，如何获得套接字的双方地址？*
  - getsockname函数：获得本地与套接字关联的IP地址和端口号
  - getpeername函数：获得通信对等端与套接字关联的IP地址和端口号

```
int getsockname(
  SOCKET    s,
  sockaddr *name,
  int      *namelen
);
```

执行成功返回0，出错返回SOCKET_ERROR

详细

```
int getpeername(
  SOCKET    s,
  sockaddr *name,
  int      *namelen
);
```

执行成功返回0，出错返回SOCKET_ERROR

详细

监听连接listen

3. 开始监听——listen()函数

- listen()函数是只能由服务器端使用的函数，而且只适用于流式套接字，listen()函数用于将套接字置为监听模式。

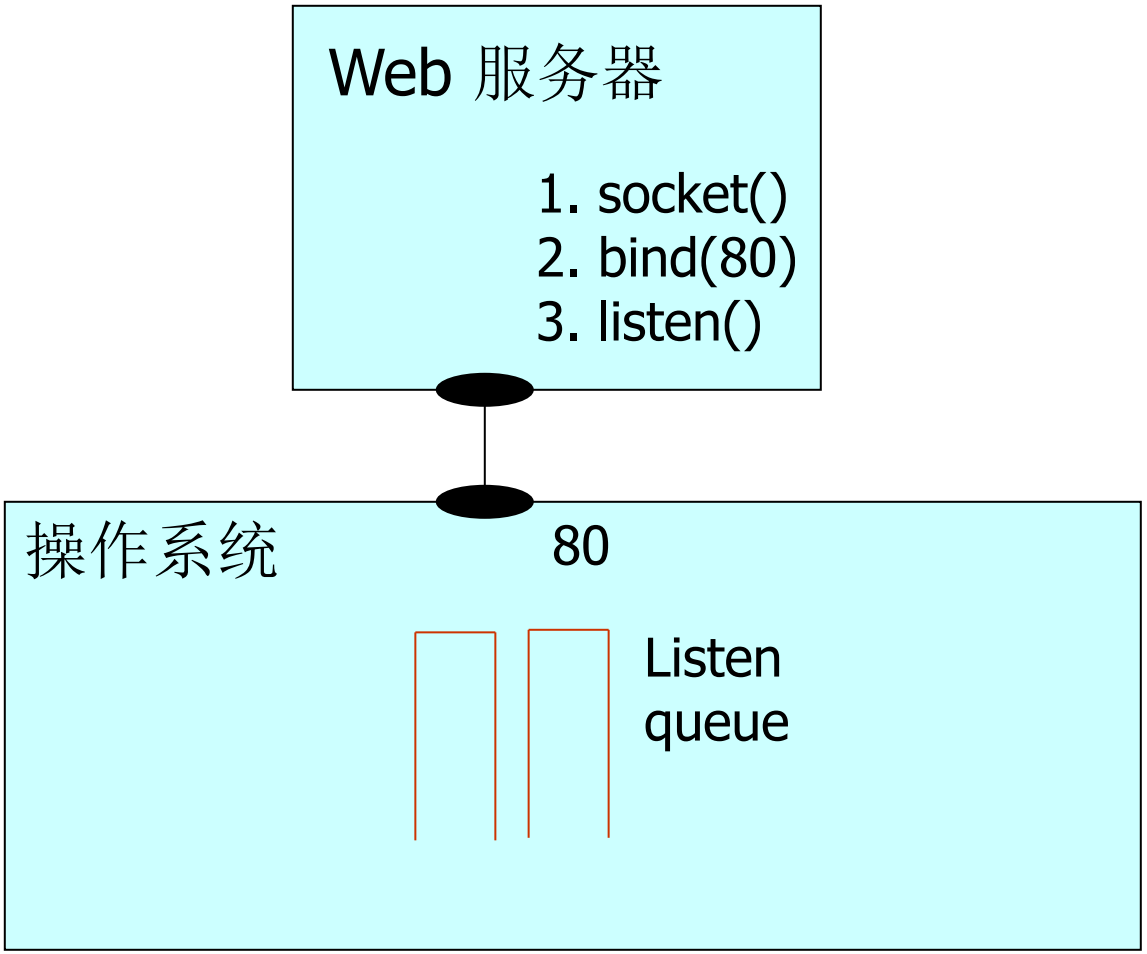
  int listen (SOCKET s, int backlog);

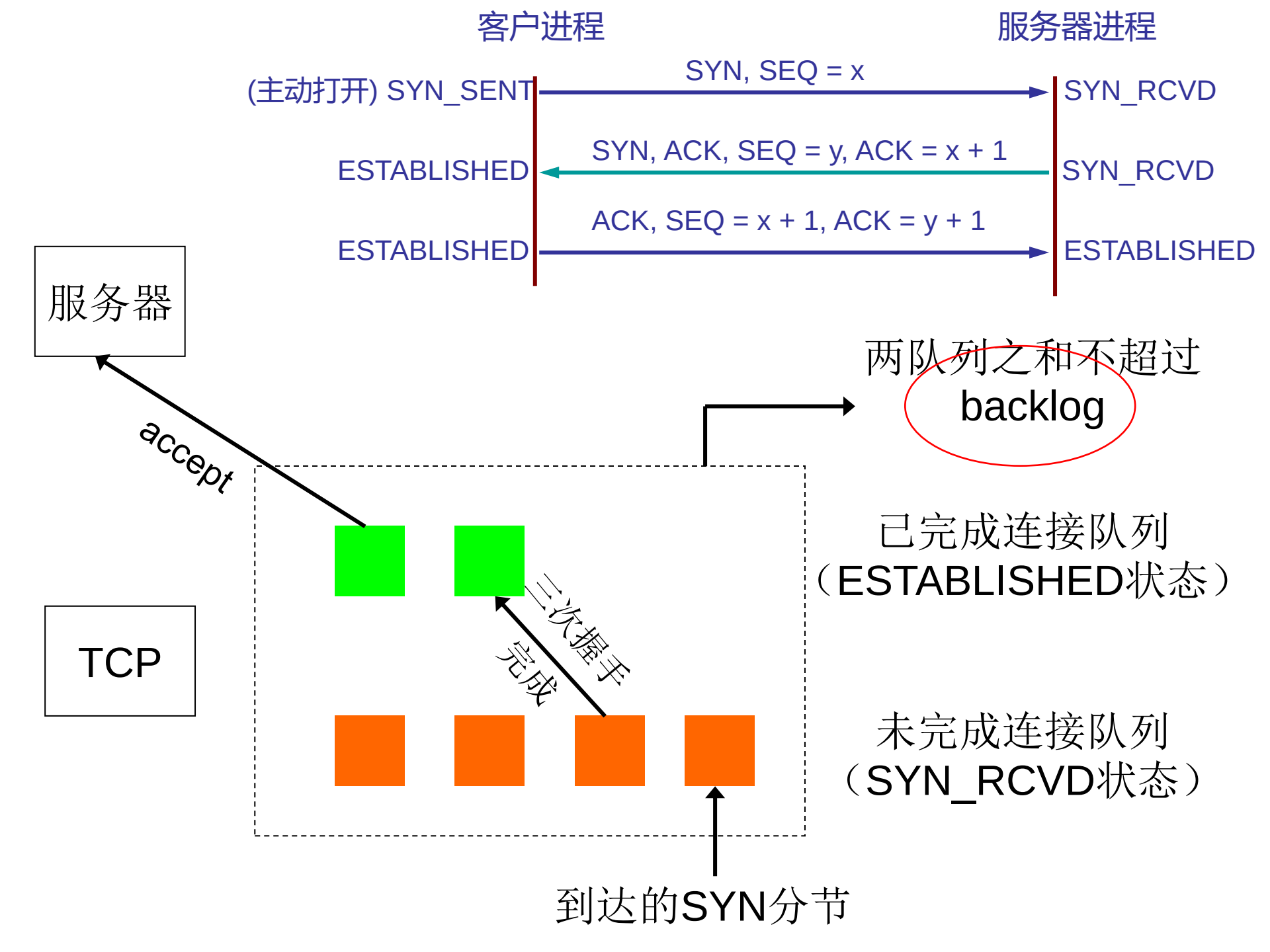
  – s：套接字。

  – backlog：表示等待连接的最大队列长度。例，若设置backlog为3，当同时收到4个客户连接请求时，则前3个客户连接请求会放置在等待队列中，第4个客户端会得到错误信息。该参数值通常设置为常量SOMAXCONN，表示将连接等待队列的最大长度值设为一个最大的“合理”值，该值有底层开发者指定，WinSock2中，该值为5。

```
if(listen(sock_server,SOMAXCONN)!=0)
{
  printf("listen函数调用失败！\n");
  return 0;
}
```

服务器初始化




客户进程
服务器进程
(主动打开) SYN_SENT
SYN, SEQ = x
SYN_RCVD
ESTABLISHED
SYN, ACK, SEQ = y, ACK = x + 1
SYN_RCVD
ESTABLISHED
ACK, SEQ = x + 1, ACK = y + 1
ESTABLISHED
服务器
两队列之和不超过
backlog
accept
已完成连接队列
（ESTABLISHED状态）
TCP
三次握手
完成
未完成连接队列
（SYN_RCVD状态）
到达的SYN分节

连接请求connect

4.客户端发送连接请求——connect()函数

int connect (SOCKET s, const struct sockaddr *name, int namelen);

- s：标识一个套接字。
- connect函数用于发送一个连接请求。若成功则返回0，否则为SOCKET_ERROR。用户可以通过WSAGetLastError得到其错误描述。
- name：套接字s想要连接的主机地址和端口号。
- namelen：name缓冲区的长度。

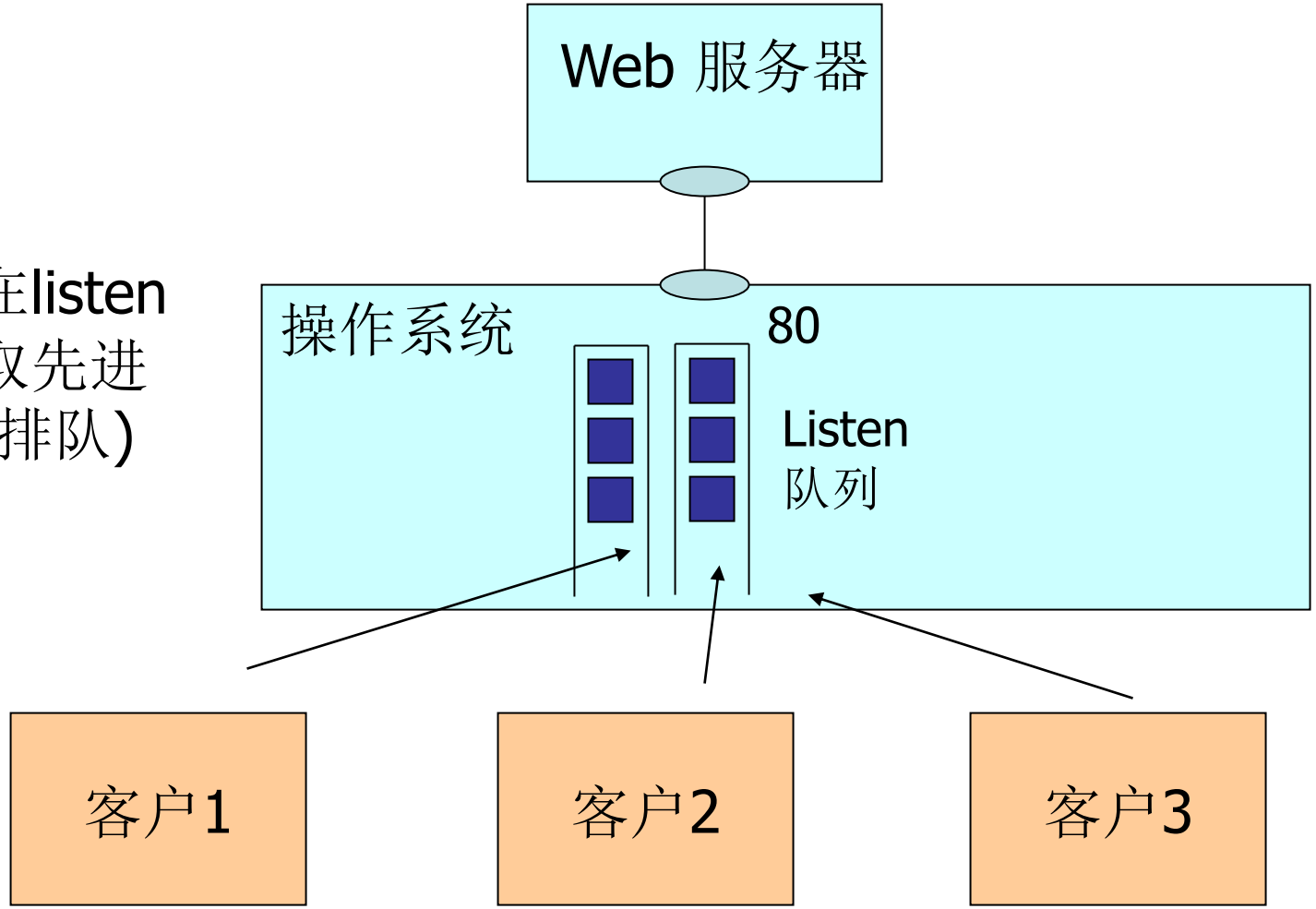
```
// Create a SOCKET for connecting to server
SOCKET ConnectSocket;
ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (ConnectSocket == INVALID_SOCKET) {
    wprintf(L"socket function failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
//----------------------
// The sockaddr_in structure specifies the address family,
// IP address, and port of the server to be connected to.
sockaddr_in clientService;
clientService.sin_family = AF_INET;
clientService.sin_addr.s_addr = inet_addr("127.0.0.1");
clientService.sin_port = htons(27015);

//----------------------
// Connect to server.
iResult = connect(ConnectSocket, (SOCKADDR *) & clientService, sizeof (clientService));
if (iResult == SOCKET_ERROR) {
    wprintf(L"connect function failed with error: %ld\n", WSAGetLastError());
    iResult = closesocket(ConnectSocket);
    if (iResult == SOCKET_ERROR)
        wprintf(L"closesocket function failed with error: %ld\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
```

关于connect函数的几点说明

- connect函数常见的若干错误
  - WSAETIMEDOUT错误：若TCP客户重发几次SYN请求后仍然没有收到SYN分节的响应。
  - WSAECONNREFUSED错误：若服务器对客户的SYN响应是RST，表示该服务器主机在客户端指定的端口上没有进程提供服务。
  - WSAEHOSTUNREACH错误或ENETUNREACH错误：客户端发出SYN后收到ICMP目的不可达报文，在重试若干次后无效。
- 注意：若connect失败，则套接口不再可用，必须关闭，再次调用connect函数是无效的。

服务器忙

客户请求在listen
队列中获取先进
先出服务(排队)



接收连接请求accept

5. 接收连接请求——accpet()函数

SOCKET accept (SOCKET s, struct sockaddr *addr,

int FAR* addrlen);

- s：是一个套接字，它应处于监听状态。
- addr：是一个sockaddr_in结构指针，包含一组客户端的端口号、IP地址等信息。
- addrlen：用于接收参数addr的长度。

- 用于接受客户端的连接请求。

accept

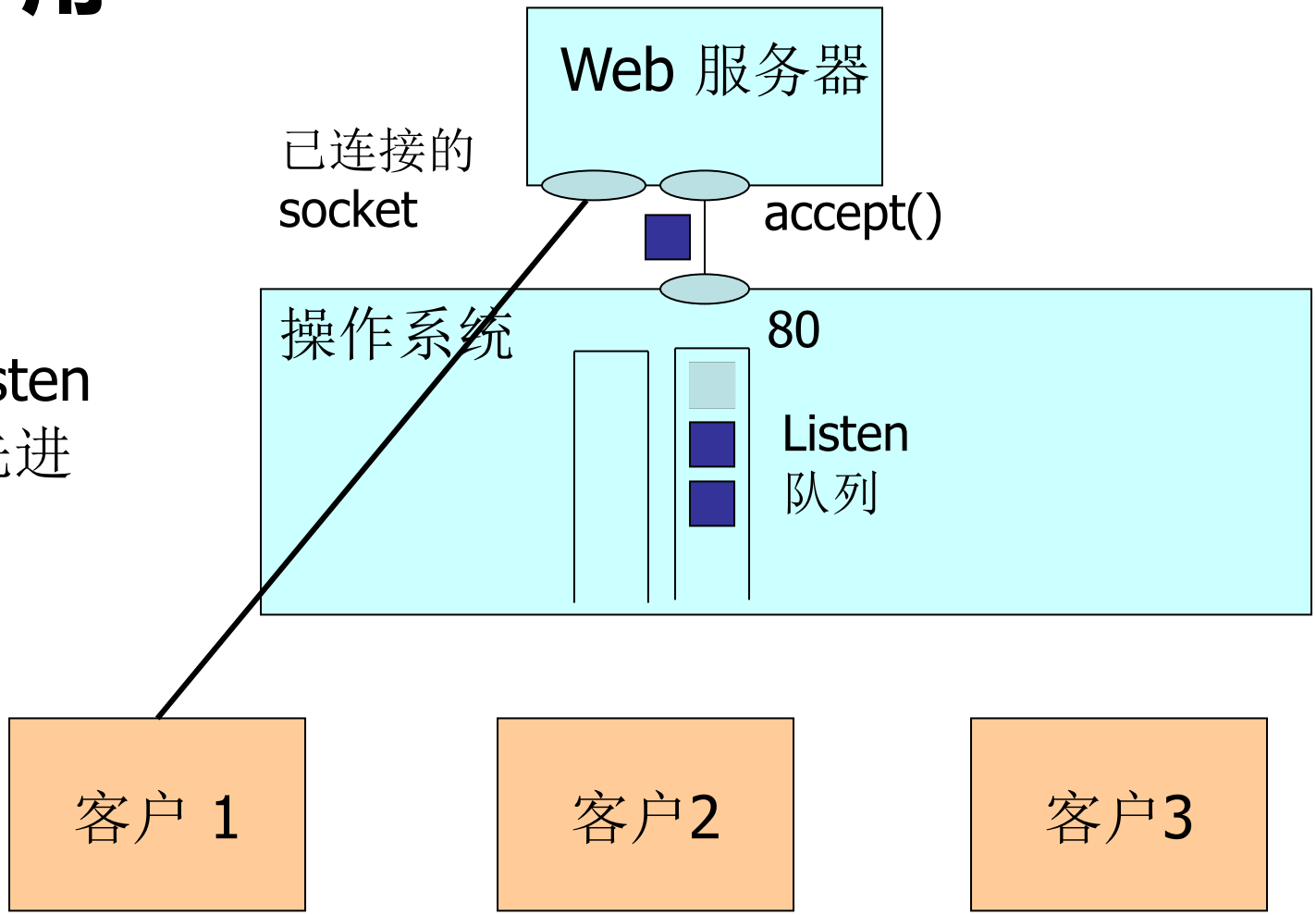
- accept()函数返回一个已建立连接的新的套接字的描述符，即已连接套接字的描述符，服务器与本连接所对应客户端的所有后续通信，都应使用该套接字。原来的监听套接字仍然处于监听状态，可以继续接受其他客户的连接请求。

- 默认情况下，如果调用accept()时还没有客户的连接请求到达，accept()将不会返回，进程将阻塞，直到有客户与服务器建立了连接才会返回。

```
SOCKET newsock = accept ( sock_server,
          (LPSOCKADDR)&client_addr,&addr_len);
 if(newsock <0)
 {
    printf("listen函数调用失败！ \n");
 }
```

accept() 调用

客户请求在listen队列中夺取先进先出服务



accept函数的执行情况

- 阻塞套接字
  - 当连接队列上没有等待的连接，**accept**进入阻塞状态
- 非阻塞套接字
  - 当连接队列上没有等待的连接，**accept**返回错误**WSAEWOULDBLOCK**

发送数据send

6. 发送数据send()函数

int send (SOCKET s, const char *buf, int len, int flag s);

- 用于在已经建立连接的套接字上发送数据。
- s： 标识一个套接字
- buf ：是存放要发送数据的缓冲区
- len：标识缓冲区长度
- flags ：用于控制数据发送的方式，通常取0，表示正常发送数据；如果取值为宏MSG_DONTROUT，则表示目标主机就在本地网络中，也就是与本机在同一个IP网段上，数据分组无须路由可直接交付目的主机，如果传输协议的实现不支持该选项则忽略该标志；如果该参数取值为宏MSG_OOB，则表示数据将按带外数据发送。
- 返回值指示了实际发送的字节总数。出错则返回SOCKET_ERROR
- 详细

- 带外数据（Out Of Band，OOB）本义是指那些使用与普通数据不同的另外的信道传送的一些特殊数据。带外数据一般都是一些重要的数据，通信协议通常能将这些数据快速地发送到对方。
- TCP协议试图通过紧急数据机制来实现带外数据的功能，但紧急数据并不是真正意义上的带外数据，因为TCP并没有真正开辟一个新的信道进行数据传输，而是将这些数据放在普通的数据流中一起传输。
- 现在，TCP的带外数据功能基本已经是废弃的功能了。2011年发表的有关TCP的紧急数据的因特网标准RFC6093就强烈建议，新的应用不要再使用紧急数据机制。

- 发送数据的例子

```
char msgbuf[]="The message to be sent";
send(s,msgbuf, sizeof(msg), 0);
```

关于send函数的几点说明

- 使用场合
  - 流式套接字
  - 已建立连接的数据报套接字
- 发送长度
  - < 套接字允许的最大长度
  - 基于消息的SOCKET最大的发送包大小为SO_MAX_MSG_SIZE(默认为65535, 即64K),超过此值，则返回**WSAEMSGSIZE**错误，数据不会被发送。
- 问题1：发送成功是否意味着数据传送到达？
  - 否

关于send函数的几点说明

- 问题2：在SOCKE_STREAM中，send函数如何获知数据的目的主机地址？
  - Server：accept函数获得对方地址
  - Client：connect函数注册对方地址
- 问题3：如果传送数据的缓存区空间不够保存需传送的数据，如何处理？
  - 阻塞模式：等待
  - 非阻塞模式：实际写的数据可能在1到所需大小之间，其值取决于本地和远端主机的缓冲区大小。通过异步处理确定何时能够进一步发送数据

关于send函数的几点说明

- 问题4：根据数据长度、网络允许最大长度和系统缓存的情况，实际返回的发送长度有哪些情况？

  设待发送的数据长度为X，实际发送的字节总数为X’，分以下几种情况讨论：

  - 1）X>最大长度: error
  - 2）Sysbuf size< X <最大长度:
    - 阻塞模式：等待，成功发送后X’=X
    - 非阻塞模式： X’=min{本地系统缓存大小，远端系统缓存大小}
  - 3）X<sysbuf : X’=X

接收数据recv

（7）recv函数

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

- 用于从连接的套接字中返回数据。该函数执行成功返回实际从套接字s读入到buf中的字节数。连接终止则返回0；否则返回SOCKET_ERROR错。
- s：标识一个套接字
- buf：是接收数据的缓冲区
- len：是buf的长度
- flags：表示函数的调用方式，一般取值0

关于recv函数的几点说明

- 使用场合
  - 面向连接：调用前必须先建立连接
  - 无连接：调用前套接字必须绑定
- 问题1：在流式套接字中，recv如何获知数据来源？
  - Server：accept函数获得对方地址
  - Client：connect函数注册对方地址
- 问题2：要接收的数据超过缓冲区大小，如何处理？
  - 最多不超过缓冲区的大小；
  - SOCK_DGRAM：超出的那部分数据会丢失；
  - SOCK_STREAM：通过循环调用recv，接收到所有数据；

- 接收数据的例子

```
char msgbuffer[1000];   //保存收到数据的缓存区
if(recv(s, msgbuffer, sizeof(msgbuffer),0)<=0)
      std::cout<<"接受信息失败\n";
else
     std::cout<<"The message from client:\n" << msgbuffer
                    <<endl; //输出收到的信息
```

断开连接shutdown

★ **中断连接——shutdown**

功能：禁止某些操作

函数定义：**int shutdown(SOCKET s, int how)**

输入参数：

s：套接字标识

how：描述禁止哪些操作（**SD_RECEIVE**，**SD_SEND**，**SD_BOTH)**

返回值：

成功：**0**；　失败：**SOCKET_ERROR**

说明：本函数并不关闭套接字，且套接字所占有的资源将被一直保持到**closesocket**（）调用。

一个简单的例子

- 客户端向服务端发送数据
- 服务端向客户端回送其所发送的数据

服务端代码

```c
#include <WinSock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib,"Ws2_32.lib")

//自定义的默认端口号,缓冲区长度
#define DEFAULT_PORT 27015
#define DEFAULT_BUFLEN 512

int main()
{
    WSADATA wsaData; //用于Winsock初始化的数据结构
    int iResult; //执行结果返回值. 为0表示成功
    SOCKET listenSocket = INVALID_SOCKET; //用来监听的套接字
    SOCKADDR_IN localAddr; //本地地址信息
    SOCKET clientSocket = INVALID_SOCKET; //用来和客户端通信的套接字
    char recvbuf[DEFAULT_BUFLEN];  //接收数据缓冲区
    int recvbuflen = DEFAULT_BUFLEN; //接收数据缓冲区的长度
    int iSendResult;  //调用send时，成功发送数据的字节数

    //初始化套接字
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup执行失败，错误码：%d\n", iResult);
        return 1;
    }
```

```
//为面向连接的服务器创建套接字
listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (listenSocket == INVALID_SOCKET) {
    printf("socket执行错误，错误码:%d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

//配置本地地址信息
localAddr.sin_family = AF_INET;
localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
localAddr.sin_port = htons(DEFAULT_PORT);

//为套接字绑定地址和端口号
iResult = bind(listenSocket, (LPSOCKADDR)&localAddr, sizeof(localAddr));
if (iResult == SOCKET_ERROR) {
    printf("bind失败，错误码:%d\n", WSAGetLastError());
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}

//监听连接请求
iResult = listen(listenSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR) {
    printf("listen失败，错误码:%d\n", WSAGetLastError());
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}
```

```
    //接受客户端的连接请求，返回连接套接字
    clientSocket = accept(listenSocket, NULL, NULL);
    if (clientSocket == INVALID_SOCKET) {
        printf("accept失败，错误码:%d\n", WSAGetLastError());
        closesocket(listenSocket);
        WSACleanup();
        return 1;
    }
    //已经不需监听了，释放监听套接字
    closesocket(listenSocket);

    //持续接收数据，直到对方关闭连接
    do { ... } while (iResult > 0);

    //关闭连接
    iResult = shutdown(clientSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown失败，错误码:%d\n", WSAGetLastError());
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
    //关闭套接字，释放资源
    closesocket(clientSocket);
    WSACleanup();
    return 0;
}
```

```
//持续接收数据，直到对方关闭连接
do {
    iResult = recv(clientSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0) {
        //成功收到数据
        printf("接收字节数：%d", iResult);
        //将缓冲区的内容回送给客户端
        iSendResult = send(clientSocket, recvbuf, iResult, 0);
        if (iSendResult == SOCKET_ERROR) {
            printf("send失败，错误码:%d\n", WSAGetLastError());
            closesocket(clientSocket);
            WSACleanup();
            return 1;
        }
        printf("发送字节数：%d", iSendResult);
    }
    else if (iResult == 0) {
        //连接关闭
        printf("连接关闭...");
    }
    else {
        //接收发生错误
        printf("recv失败，错误码:%d\n", WSAGetLastError());
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
} while (iResult > 0);
```

服务端代码
使用getaddrinfo

```c
#include <WinSock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib,"Ws2_32.lib")

//自定义的默认端口号,缓冲区长度
#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLEN 512

int main()
{
    WSADATA wsaData; //用于Winsock初始化的数据结构
    int iResult; //执行结果返回值. 为0表示成功
    struct addrinfo* result = NULL,hints; //地址信息结构
    SOCKET listenSocket = INVALID_SOCKET; //用来监听的套接字
    SOCKET clientSocket = INVALID_SOCKET; //用来和客户端通信的套接字
    char recvbuf[DEFAULT_BUFLEN];  //接收数据缓冲区
    int recvbuflen = DEFAULT_BUFLEN; //接收数据缓冲区的长度
    int iSendResult;  //调用send时，成功发送数据的字节数

    //初始化套接字
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup执行失败，错误码：%d\n", iResult);
        return 1;
    }
```

```
//解析服务器地址和端口号
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE; //当地址无法解析时就置为0.0.0.0，便于bind
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo执行错误，错误码：%d\n", iResult);
    WSACleanup();
    return 1;
}
//为面向连接的服务器创建套接字
listenSocket = socket(result->ai_family,(int)result->ai_socktype,result->ai_protocol);
if (listenSocket==INVALID_SOCKET) {
    printf("socket执行错误，错误码:%d\n",WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}
//为套接字绑定地址和端口号
iResult = bind(listenSocket,result->ai_addr,(int)result->ai_addrlen);
if (iResult==SOCKET_ERROR) {
    printf("bind失败，错误码:%d\n",WSAGetLastError());
    freeaddrinfo(result);
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}
freeaddrinfo(result);
```

```
//监听连接请求
iResult = listen(listenSocket,SOMAXCONN);
if (iResult==SOCKET_ERROR) {
    printf("listen失败，错误码:%d\n", WSAGetLastError());
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}

//接受客户端的连接请求，返回连接套接字
clientSocket = accept(listenSocket,NULL,NULL);
if (clientSocket==INVALID_SOCKET) {
    printf("accept失败，错误码:%d\n", WSAGetLastError());
    closesocket(listenSocket);
    WSACleanup();
    return 1;
}
//已经不需监听了，释放监听套接字
closesocket(listenSocket);
```

```
//持续接收数据，直到对方关闭连接
do {
    iResult = recv(clientSocket,recvbuf,recvbuflen,0);
    if (iResult > 0) {
        //成功收到数据
        printf("接收字节数：%d",iResult);
        //将缓冲区的内容回送给客户端
        iSendResult = send(clientSocket,recvbuf,iResult,0);
        if (iSendResult == SOCKET_ERROR) {
            printf("send失败，错误码:%d\n", WSAGetLastError());
            closesocket(clientSocket);
            WSACleanup();
            return 1;
        }
        printf("发送字节数：%d", iSendResult);
    }
    else if (iResult==0) {
        //连接关闭
        printf("连接关闭...");
    }
    else {
        //接收发生错误
        printf("recv失败，错误码:%d\n", WSAGetLastError());
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
} while (iResult>0);
```

```
    //关闭连接
    iResult = shutdown(clientSocket,SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown失败，错误码:%d\n", WSAGetLastError());
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
    //关闭套接字，释放资源
    closesocket(clientSocket);
    WSACleanup();
    return 0;
}
```

客户端程序

```c
#include <WinSock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib,"Ws2_32.lib")

//自定义的默认端口号,缓冲区长度
#define DEFAULT_PORT 27015
#define DEFAULT_BUFLEN 512

int main(int argc, char** argv)
{
    WSADATA wsaData;
    int iResult;
    SOCKET connectSocket = INVALID_SOCKET;
    SOCKADDR_IN serverAddr; //服务器地址
    const char* sendbuf = "This is a test"; //要发送的测试数据
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    //验证参数合法性
    if (argc != 2) {
        printf("命令用法： %s 服务器域名或ip\n", argv[0]);
        return 1;
    }
    //初始化套接字
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup执行失败，错误码：%d\n", iResult);
        return 1;
    }
```

```
//配置服务器地址信息
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(DEFAULT_PORT);
iResult = inet_pton(AF_INET, argv[1], &(serverAddr.sin_addr.s_addr));
if (iResult <= 0) { //地址信息无法解释或出错
    printf("服务端地址有错\n");
    WSACleanup();
    return 1;
}

//创建套接字
connectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (connectSocket == INVALID_SOCKET) {
    printf("socket执行失败,错误码:%d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
//向服务器请求连接
iResult = connect(connectSocket, (LPSOCKADDR)&serverAddr, sizeof(serverAddr));
if (iResult == SOCKET_ERROR) {
    closesocket(connectSocket);
    connectSocket = INVALID_SOCKET;
    printf("无法连接到服务器!\n");
    WSACleanup();
    return 1;
}
```

```
    //发送缓冲区中的测试数据
    iResult = send(connectSocket, sendbuf, (int)strlen(sendbuf), 0);
    if (iResult == SOCKET_ERROR) {
        printf("发送失败,错误码:%d\n", WSAGetLastError());
        closesocket(connectSocket);
        WSACleanup();
        return 1;
    }
    printf("发送字节数:%ld\n", iResult);
    //数据发送结束，调用shutdown()函数声明不再发送数据，此时客户端仍然可以接收数据
    iResult = shutdown(connectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown执行出错，错误码：%d\n", WSAGetLastError());
        closesocket(connectSocket);
        WSACleanup();
        return 1;
    }
    //持续接收数据，直到服务器关闭连接
    do {
        iResult = recv(connectSocket, recvbuf, recvbuflen, 0);
        if (iResult > 0)
            printf("接收字节数:%d\n", iResult);
        else if (iResult == 0)
            printf("连接关闭\n");
        else
            printf("接收数据失败，错误码：%d\n", WSAGetLastError());
    } while (iResult > 0);

    //关闭套接字
    closesocket(connectSocket);
    //释放资源
    WSACleanup();
    return 0;

}
```

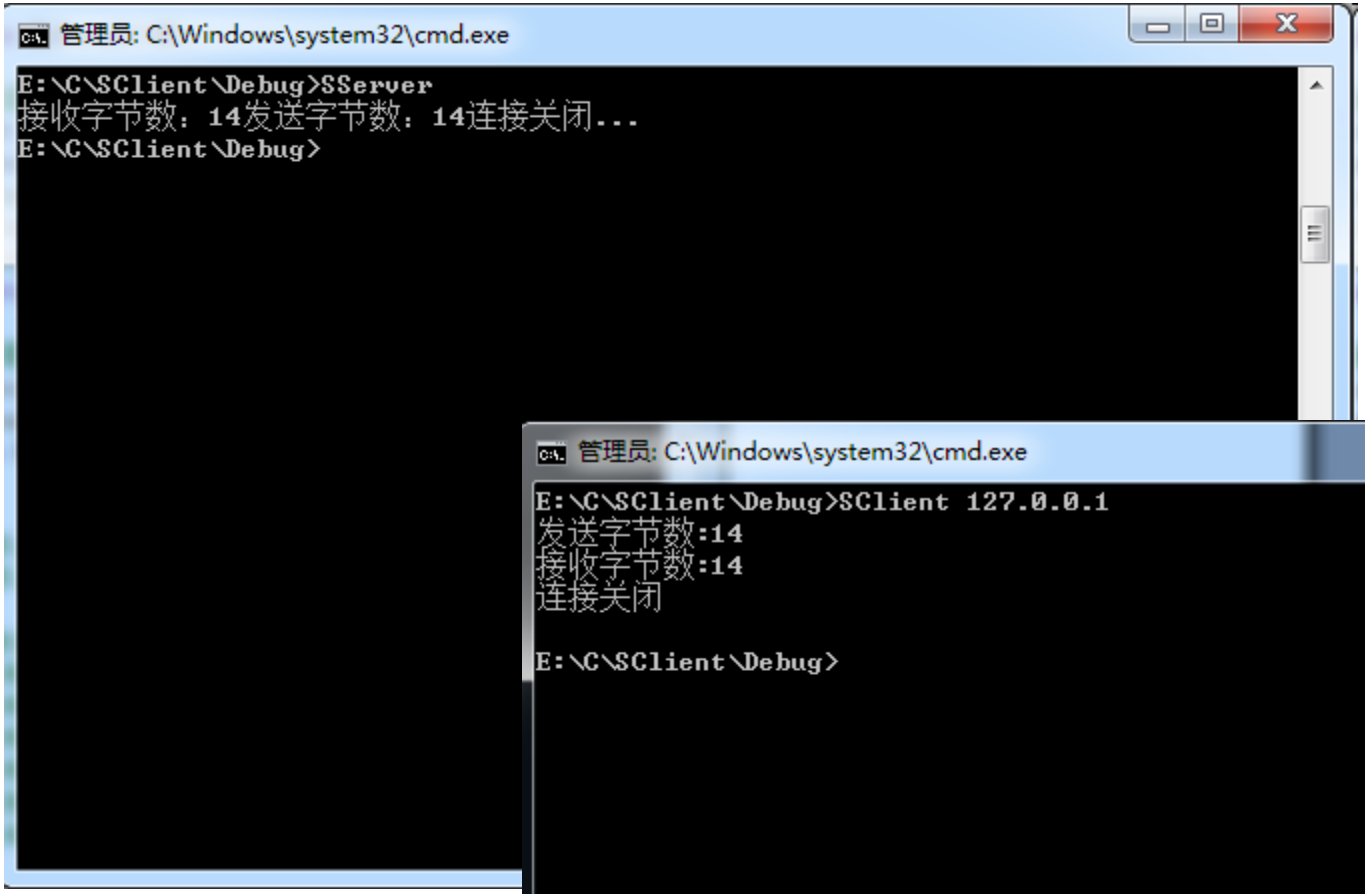
客户端程序
使用getaddrinfo

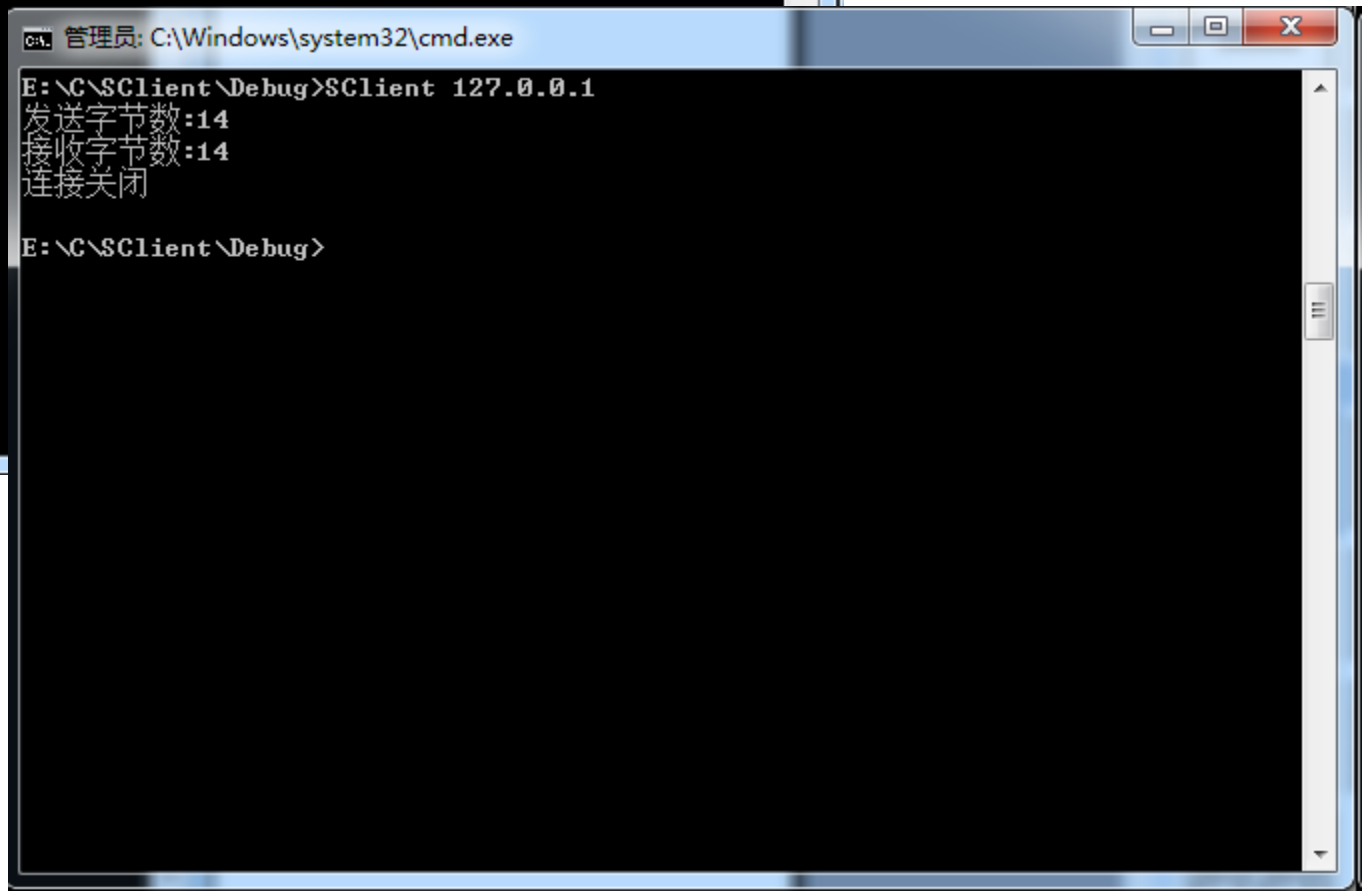
```c
#include <WinSock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib,"Ws2_32.lib")

//自定义的默认端口号,缓冲区长度
#define DEFAULT_PORT "27015"
#define DEFAULT_BUFLEN 512

int main(int argc,char **argv)
{

    WSADATA wsaData;
    int iResult;
    struct addrinfo* result = NULL, * ptr = NULL, hints;
    SOCKET connectSocket = INVALID_SOCKET;
    const char *sendbuf = "This is a test"; //要发送的测试数据
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    //验证参数合法性
    if (argc!=2) {
        printf("命令用法： %s 服务器域名或ip\n",argv[0]);
        return 1;
    }
    //初始化套接字
    iResult = WSAStartup(MAKEWORD(2,2),&wsaData);
    if (iResult != 0) {
        printf("WSAStartup执行失败，错误码：%d\n",iResult);
        return 1;
    }
```

```c
//解析服务器地址和端口号
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
iResult = getaddrinfo(argv[1],DEFAULT_PORT,&hints,&result);
if (iResult != 0) {
    printf("getaddrinfo执行错误，错误码：%d\n",iResult);
    WSACleanup();
    return 1;
}
//尝试连接服务器地址，直到成功
for (ptr = result; ptr != NULL;ptr=ptr->ai_next) {
    //创建套接字
    connectSocket = socket(ptr->ai_family,ptr->ai_socktype,ptr->ai_protocol);
    if (connectSocket==INVALID_SOCKET) {
        printf("socket执行失败,错误码:%d\n",WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //向服务器请求连接
    iResult = connect(connectSocket,ptr->ai_addr,(int)ptr->ai_addrlen);
    if (iResult==SOCKET_ERROR) {
        closesocket(connectSocket);
        connectSocket = INVALID_SOCKET;
        continue;
    }
    break;
}
freeaddrinfo(result);
if (connectSocket== INVALID_SOCKET) {
    printf("无法连接到服务器!\n");
    WSACleanup();
    return 1;
}
```

```
//发送缓冲区中的测试数据
iResult = send(connectSocket,sendbuf,(int)strlen(sendbuf),0);
if (iResult==SOCKET_ERROR) {
    printf("发送失败,错误码:%d\n",WSAGetLastError());
    closesocket(connectSocket);
    WSACleanup();
    return 1;
}
printf("发送字节数:%ld\n",iResult);
//数据发送结束，调用shutdown()函数声明不再发送数据，此时客户端仍然可以接收数据
iResult = shutdown(connectSocket,SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown执行出错，错误码：%d\n",WSAGetLastError());
    closesocket(connectSocket);
    WSACleanup();
    return 1;
}
//持续接收数据，直到服务器关闭连接
do {
    iResult = recv(connectSocket,recvbuf,recvbuflen,0);
    if (iResult > 0)
        printf("接收字节数:%d\n", iResult);
    else if (iResult == 0)
        printf("连接关闭\n");
    else
        printf("接收数据失败，错误码：%d\n",WSAGetLastError());
} while (iResult>0);
```

```
    //关闭套接字
    closesocket(connectSocket);
    //释放资源
    WSACleanup();
    return 0;
}
```


管理员: C:\Windows\system32\cmd.exe
E:\C\SClient\Debug>SServer
接收字节数：14发送字节数：14连接关闭...
E:\C\SClient\Debug>


管理员: C:\Windows\system32\cmd.exe
E:\C\SClient\Debug>SClient 127.0.0.1
发送字节数:14
接收字节数:14
连接关闭
E:\C\SClient\Debug>

TCP的流传输控制

内容提要

TCP的流传送特点

- 字节流的特点
- TCP对字节流的处理
- 数据接收的可能情况

从套接字接口层观察流的传送

- 套接字接口层的位置与内容
- 数据发送和接收过程中的缓冲现象

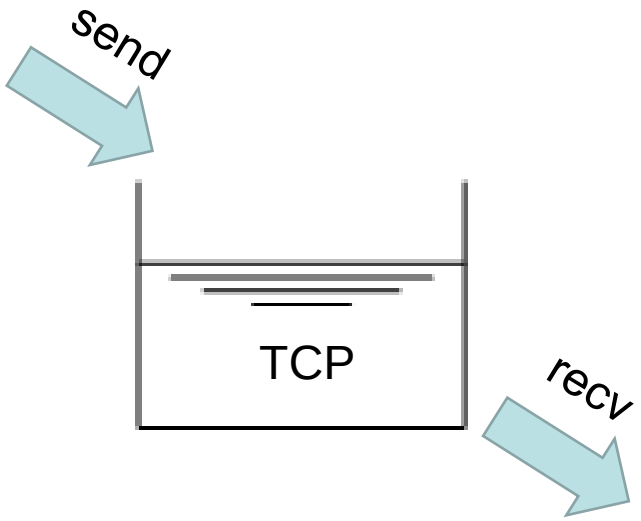
合理处理流数据的接收

- 合理判断接收返回值
- 接收定长数据
- 接收变长数据

1、TCP的流传送特点

1.1字节流的特点

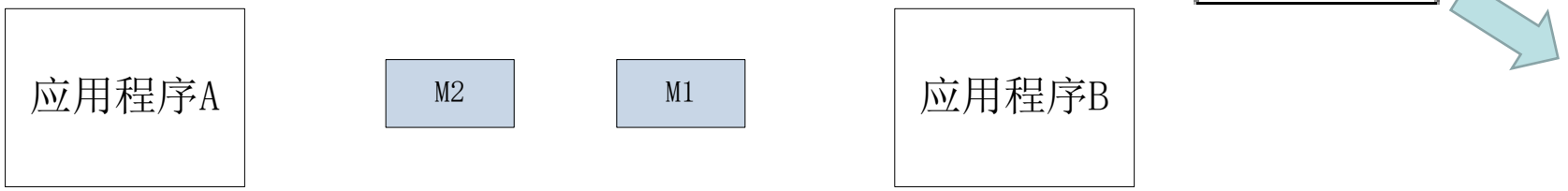
- 无边界
- 传送时机未知
- 写入、读出、传送形态各异



1、TCP的流传送特点

1.2 TCP对字节流的处理

✓从传输方的角度来观察数据



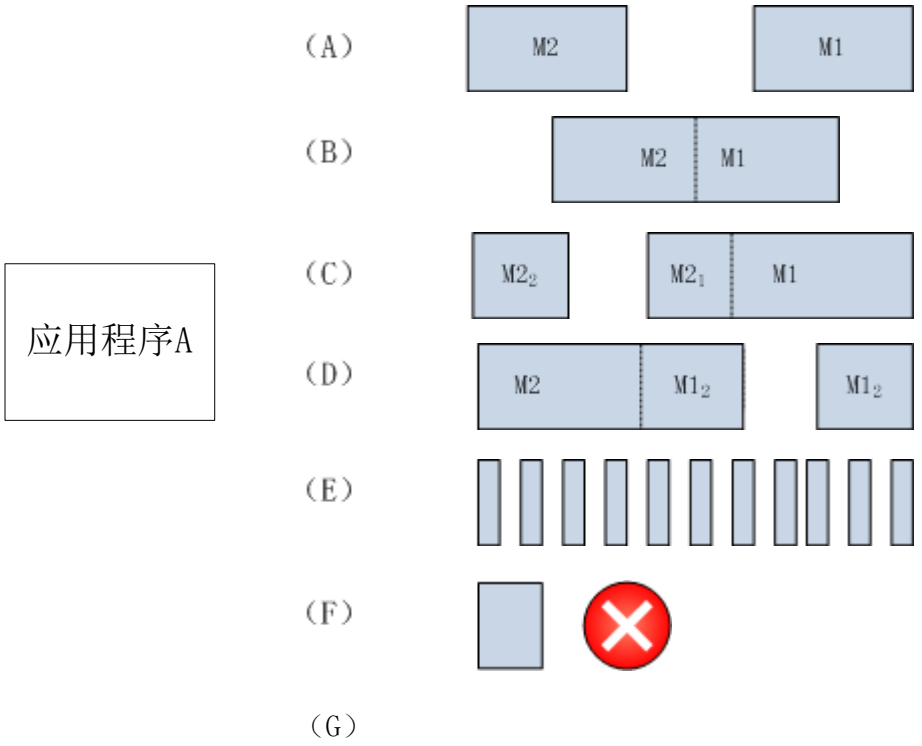
TCP如何决定字节流的处理？

- 当前接收方希望接收的数据量
- 网络是否存在阻塞
- 网络路径MTU
- 本地连接的输出队列上有多少数据正在排队或未接收到应答
- ……

1、TCP的流传送特点

1.2 TCP对字节流的处理

✓从传输方的角度来观察数据

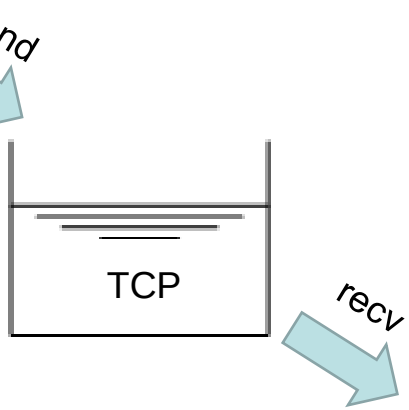


当程序调用发送操作时，由于主机和网络当前的状态不同，会使得数据传送的形式产生很大的差别！

1、TCP的流传送特点

1.3 数据接收的可能情况

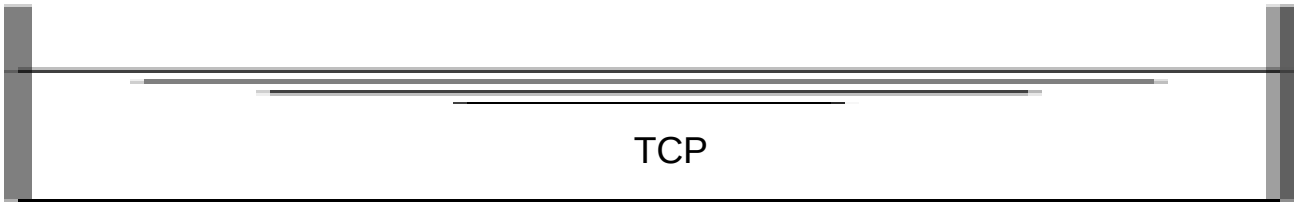
✓从接收方的角度来观察数据



没有数据

接收到已交付的全部数据

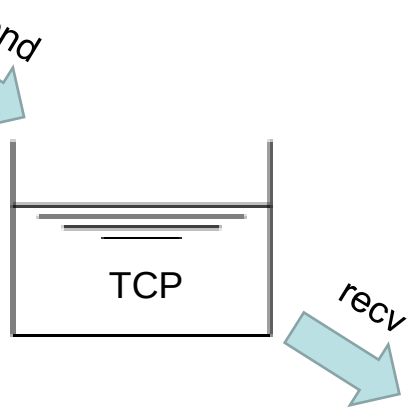
接收到已交付的部分数据



1、TCP的流传送特点

1.3 数据接收的可能情况

✓从接收方的角度来观察数据



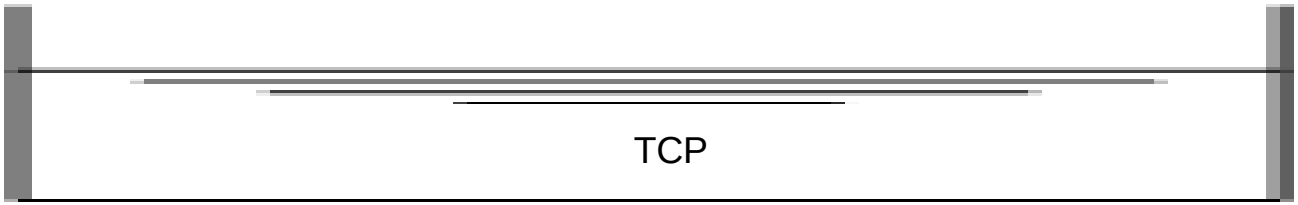
（F）

（G）

没有数据

接收到已交付的全部数据

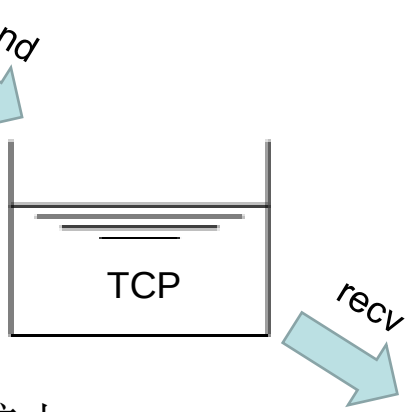
接收到已交付的部分数据



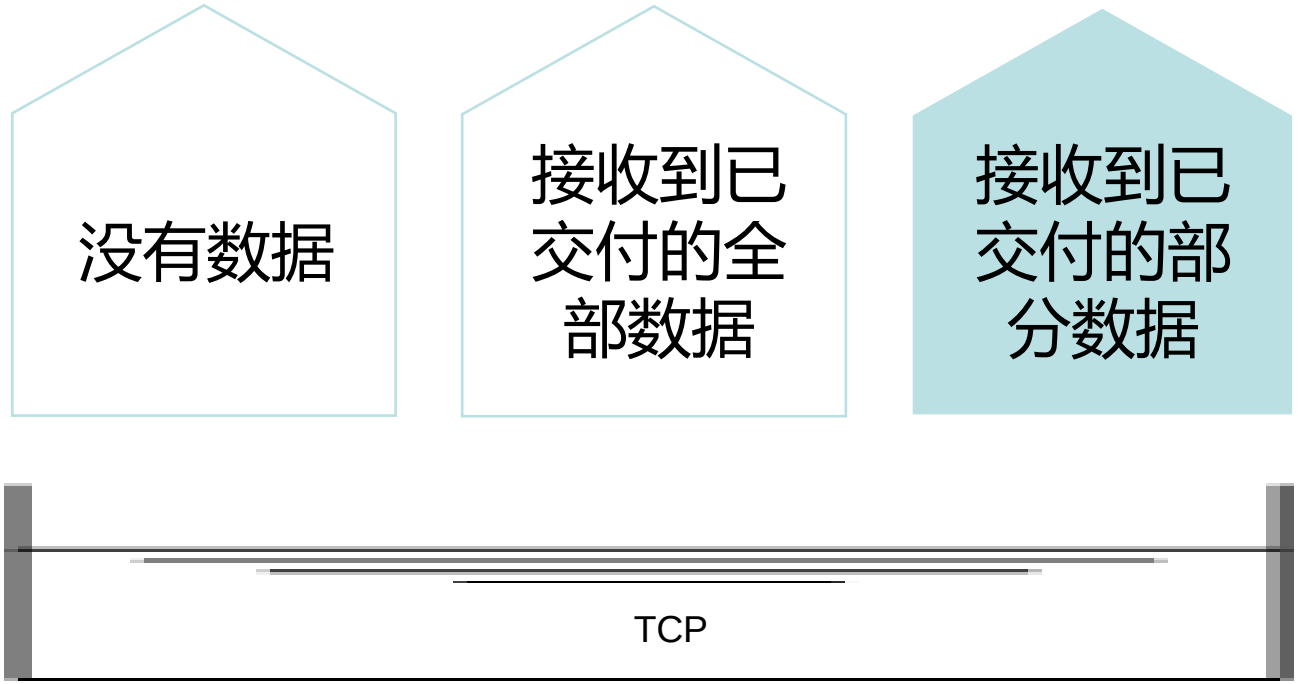
1、TCP的流传送特点

1.3 数据接收的可能情况

✓从接收方的角度来观察数据

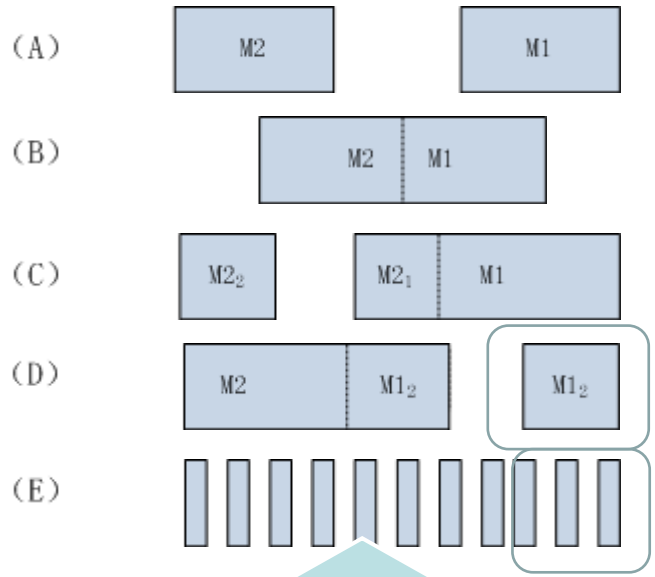


接收缓存较小



1、TCP的流传送特点

部分M1



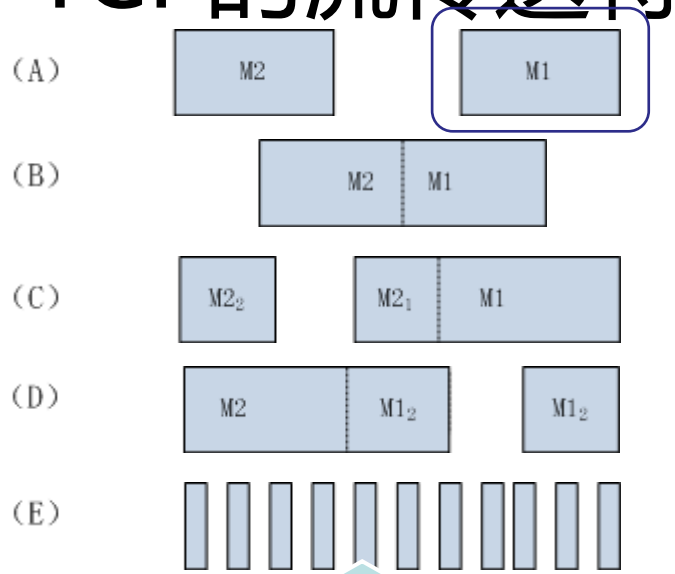
没有数据

接收到已交付的全部数据

接收到已交付的部分数据

TCP

1、TCP的流传送特点



M1

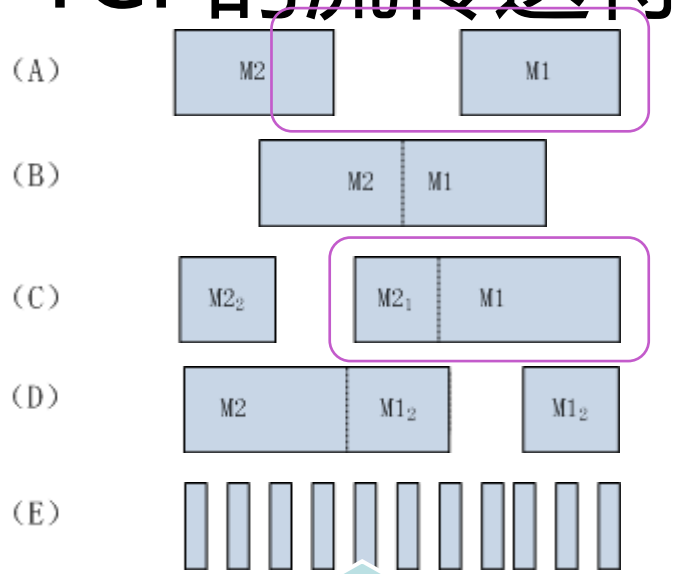
没有数据

接收到已交付的全部数据

接收到已交付的部分数据

TCP

1、TCP的流传送特点



M1+部分M2

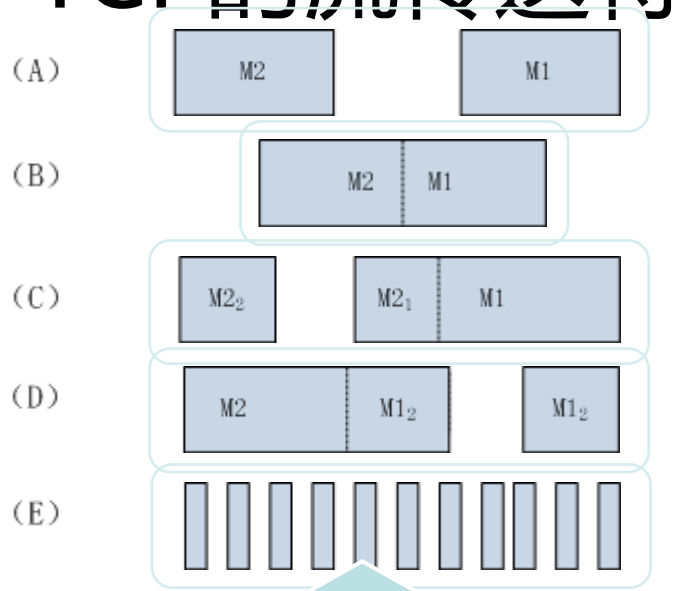
没有数据

接收到已交付的全部数据

接收到已交付的部分数据

TCP

1、TCP的流传送特点



M1+M2

没有数据

接收到已交付的全部数据

接收到已交付的部分数据

TCP

1、TCP的流传送特点

- *总结*

TCP是一个流协议，TCP如何对数据打包跟调用send()函数传递多少数据给TCP没有直接的关系。

对于使用TCP的应用程序来说，没有“数据边界”的概念，接收操作返回的时机和数量是不可预测的，必须在应用程序中正确处理。

内容提要

TCP的流传送特点

- 字节流的特点
- TCP对字节流的处理
- 数据接收的可能情况

从套接字接口层观察流的传送

- 套接字接口层的位置与内容
- 数据发送和接收过程中的缓存现象

合理处理流数据的接收

- 合理判断接收返回值
- 接收定长数据
- 接收变长数据

2、从套接字接口层观察流的传送

2.1 套接字接口层的位置与内容

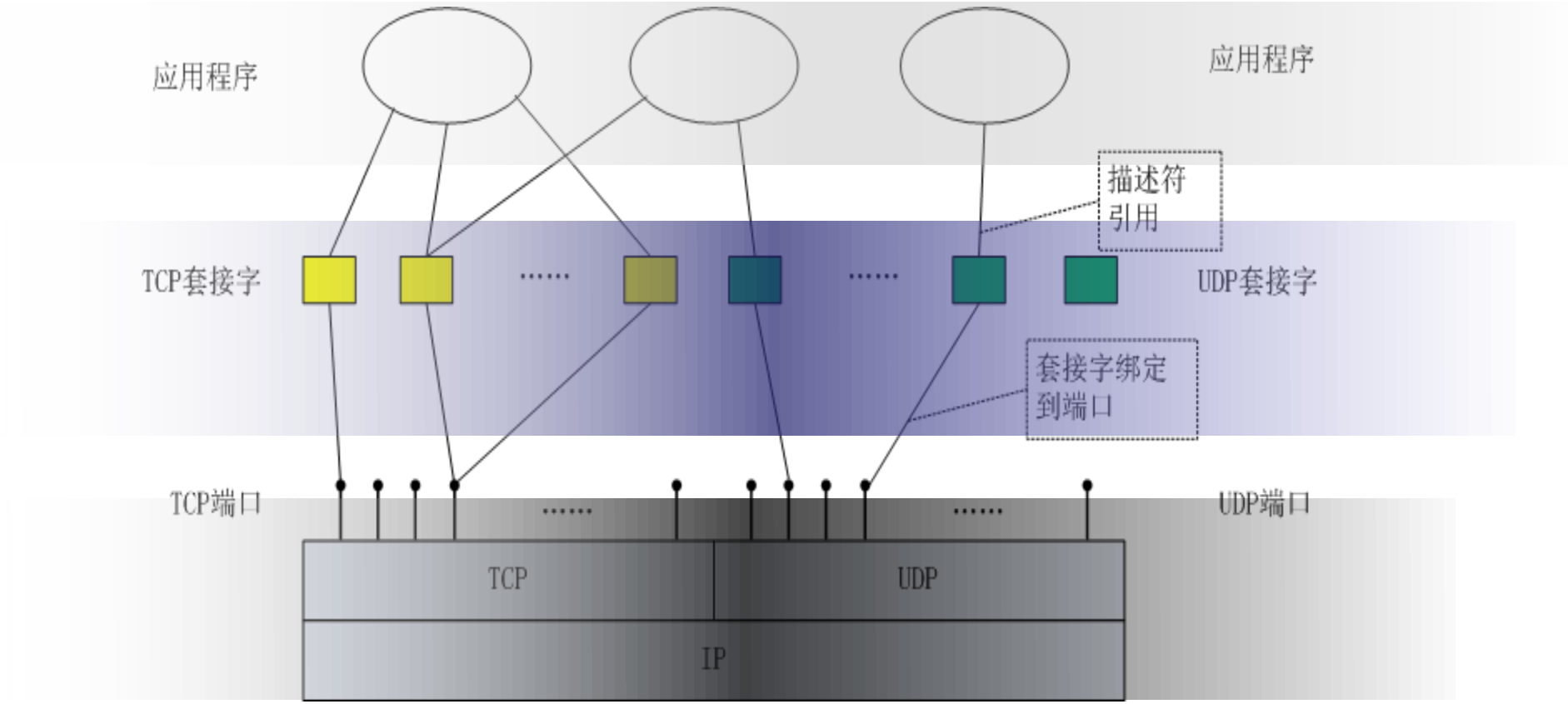
应用程序

套接字接口层

协议实现

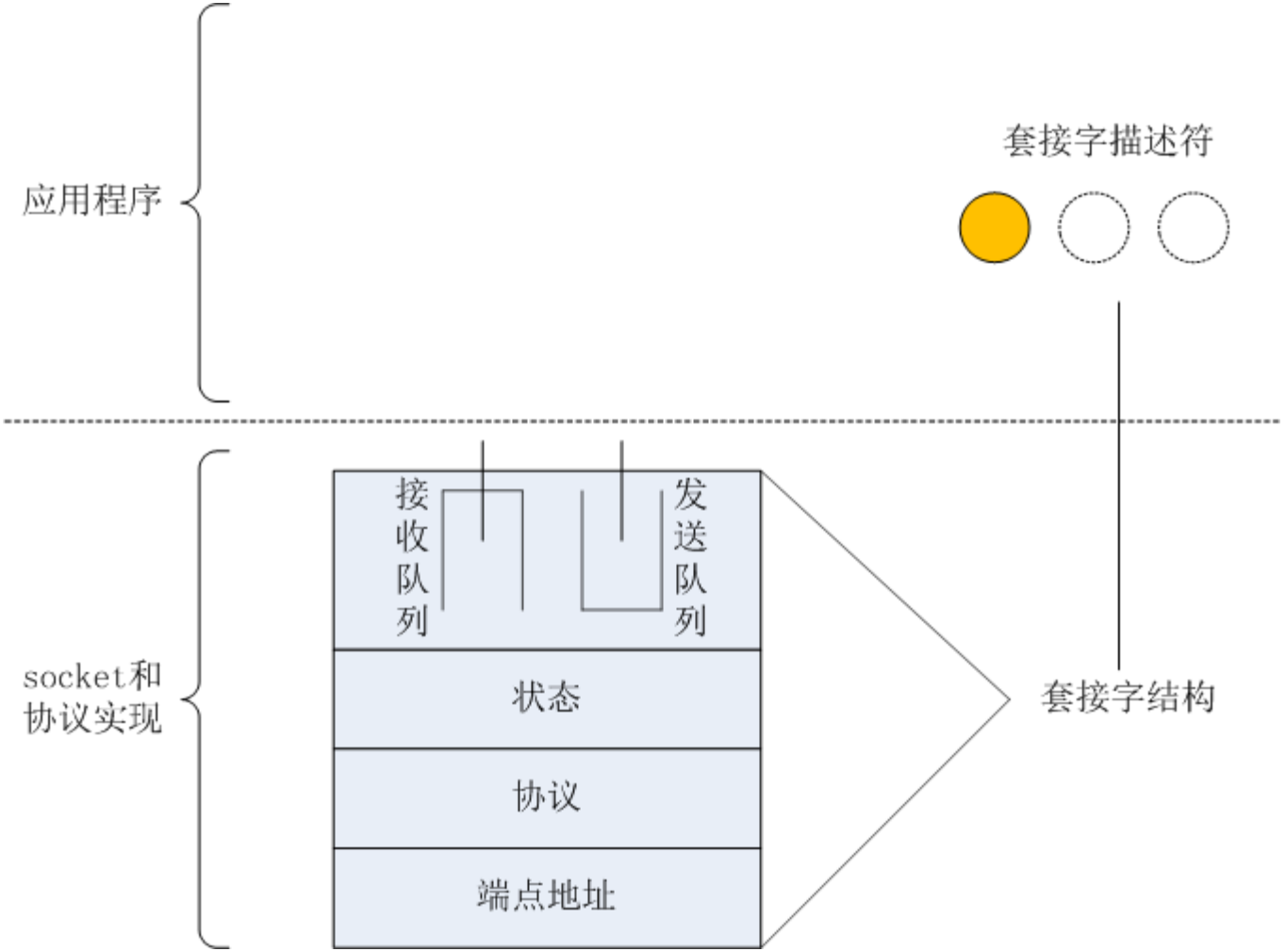
2、从套接字接口层观察流的传送

2.1 套接字接口层的位置与内容



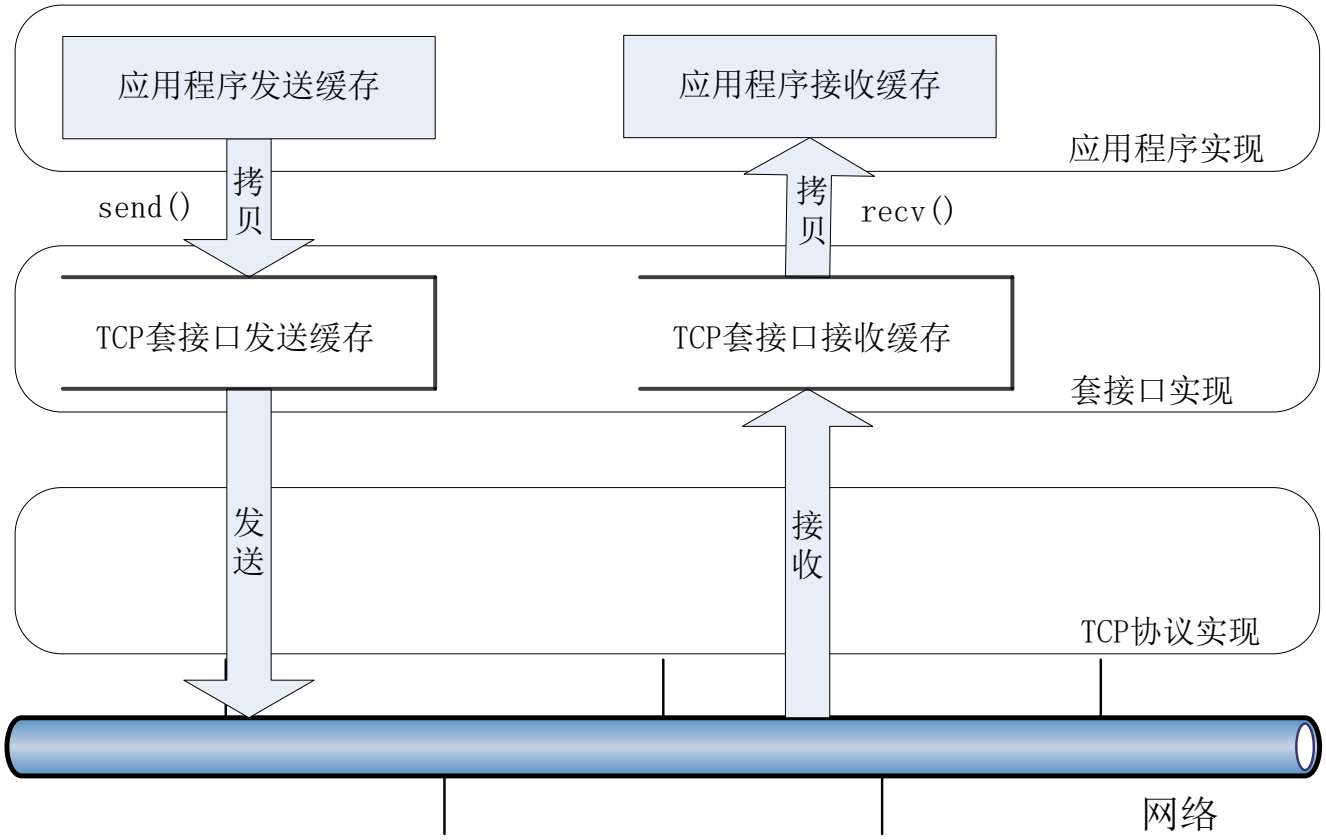
2、从套接字接口层观察流的传送

2.1 套接字接口层的位置与内容



2、从套接字接口层观察流的传送

2.2 数据发送和接收过程中的缓存现象



2、从套接字接口层观察流的传送

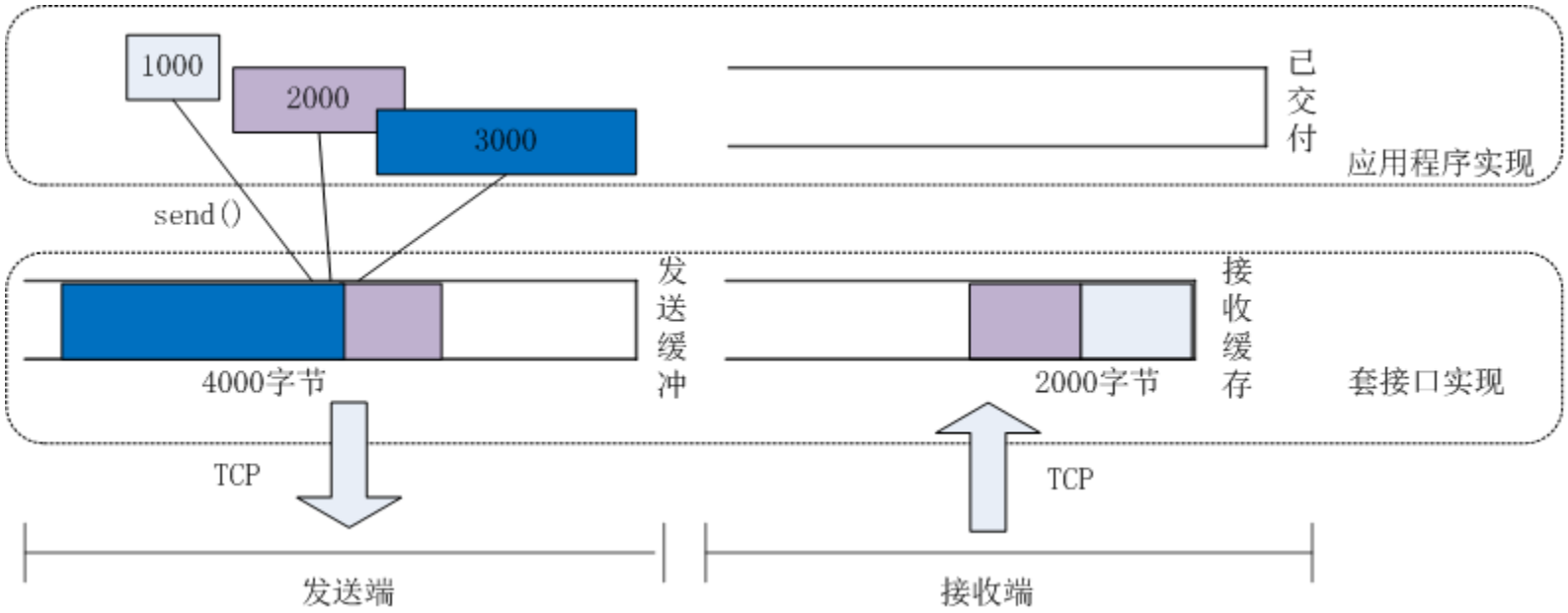
2.2 数据发送和接收过程中的缓存现象

```
……
iSendResult = send(ClientSocket, buffer0, 1000, 0);
……
iSendResult = send(ClientSocket, buffer1, 2000, 0);
……
iSendResult = send(ClientSocket, buffer2, 3000, 0);
……
```

2、从套接字接口层观察流的传送

2.2 数据发送和接收过程中的缓存现象

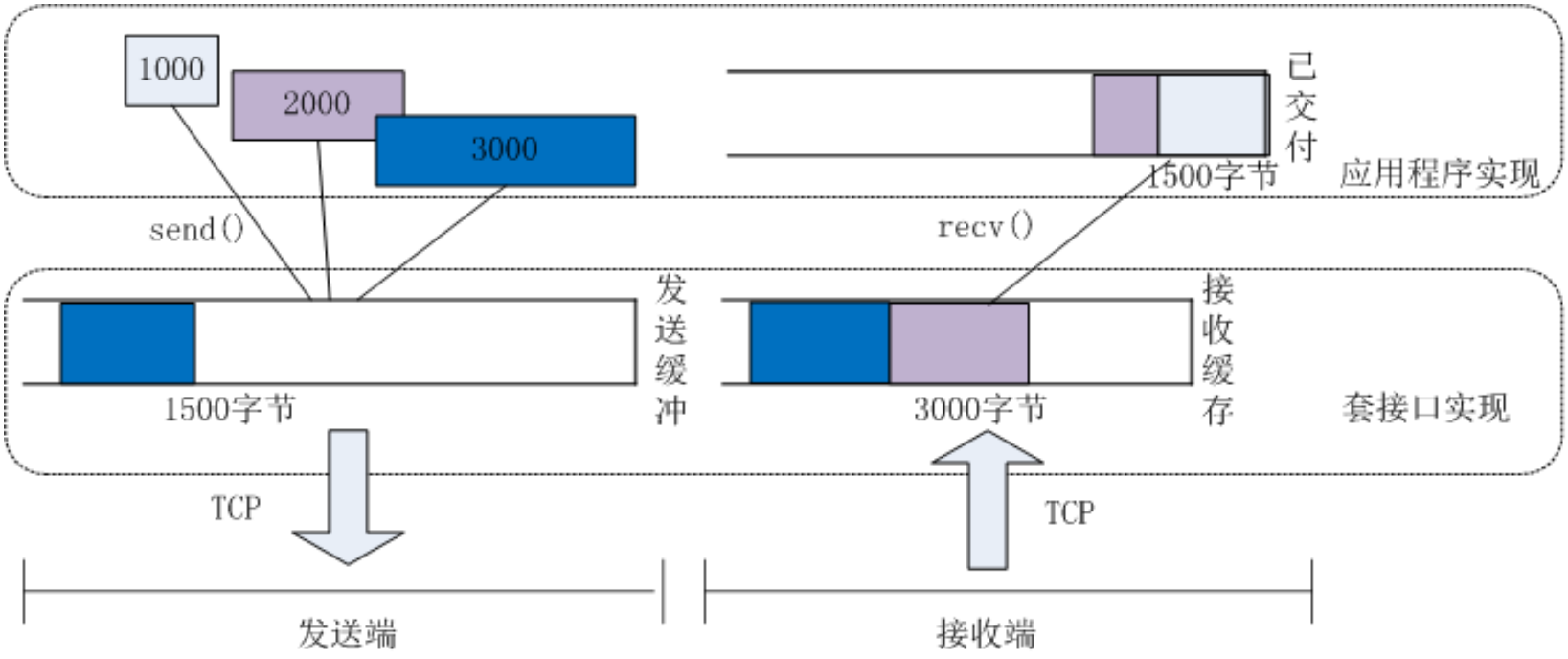
接收端尚未执行任何recv()之前的一种可能的状态。



2、从套接字接口层观察流的传送

2.2 数据发送和接收过程中的缓存现象

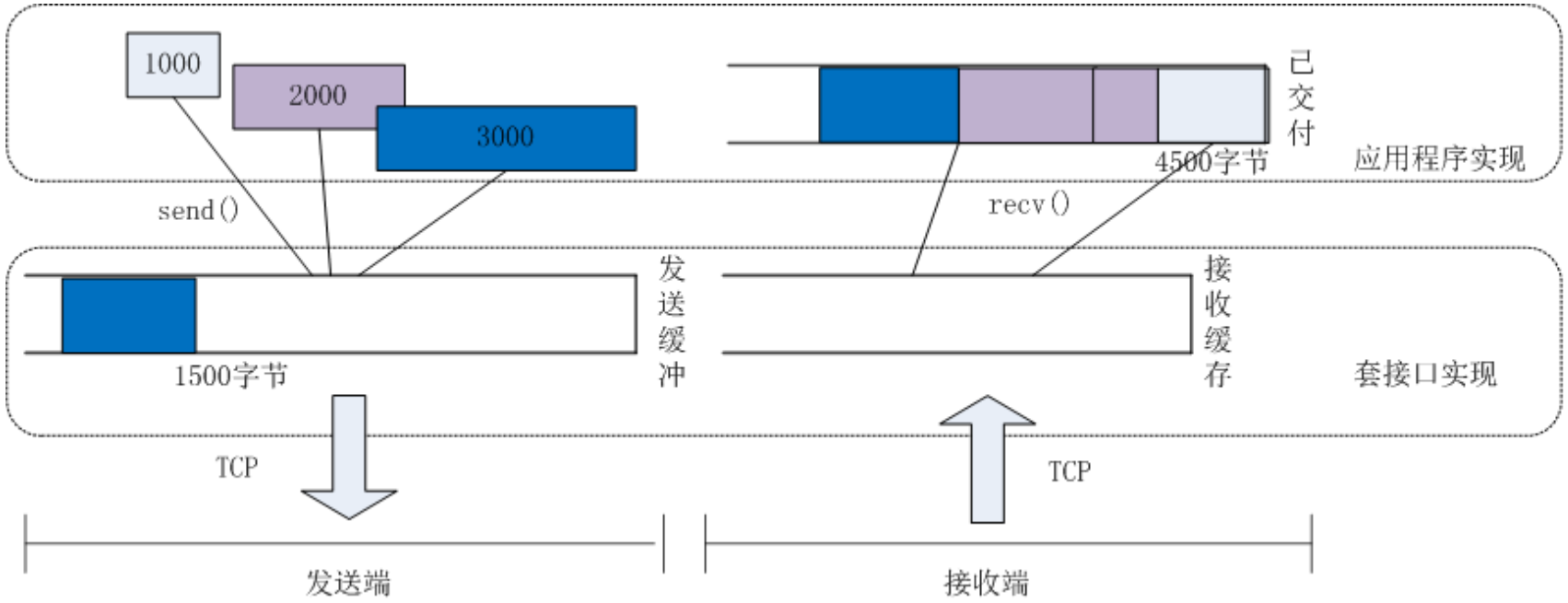
接收端执行一次recv()后的情况，接收端的应用程序缓冲区长度<当前接收缓存的数据长度。



2、从套接字接口层观察流的传送

2.2 数据发送和接收过程中的缓存现象

接收端执行下一次recv()后的情况，接收端的应用程序缓冲区长度>当前接收缓存的数据长度。



内容提要

TCP的流传送特点

- 字节流的特点
- TCP对字节流的处理
- 数据接收的可能情况

从套接字接口层观察流的传送

- 套接字接口层的位置与内容
- 数据发送和接收过程中的缓存现象

合理处理流数据的接收

- 完整接收流数据
- 接收定长数据
- 接收变长数据

3、合理处理流数据的接收

3.1 完整接收流数据

```
SOKCET ConnectSocket;
int recvbuflen;
char recvbuf [MSGSZ];
recv(ConnectSocket, recvbuf, recvbuflen, 0);
```

3、合理处理流数据的接收

3.1完整接收流数据

数据的接收处理需要考虑到网络中的各种可能性！

第一，一次接收不能保证接收到一次发送的所有数据！

3、合理处理流数据的接收

3.1 完整接收流数据

数据的接收处理需要考虑到网络中的各种可能性！

第二，接收函数的调用结果有很多种！

结果判断

返回结果	原因	后续处理
iResult == recvbuflen	接收到与缓冲区长度相等的数据	应处理数据或继续接收
iResult < recvbuflen	表明到达接收方缓存的数据量少于接收缓存长度	应处理数据或继续接收
iResult == 0	表明对方关闭了连接，发送FIN标志的TCP段	应退出循环等待接收过程
iResult == SOCK_ERROR	接收出现错误	应处理错误

3、合理处理流数据的接收

3.1完整接收流数据

```
int iResult, recvbuflen;
char recvbuf [MSGSZ];
do {
        iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
        if ( iResult > 0 )
            printf("Bytes received: %d\n", iResult);
        else {
            if ( iResult == 0 )
                printf("Connection closed\n");
            else
                printf("recv failed with error: %d\n", WSAGetLastError());
            }
} while( iResult > 0 );
```

3、合理处理流数据的接收

字节流的传输特点使得应用程序在数据接收时存在以下困惑：

（1）预分配的应用程序缓冲区recvbuf的取值难以确定。

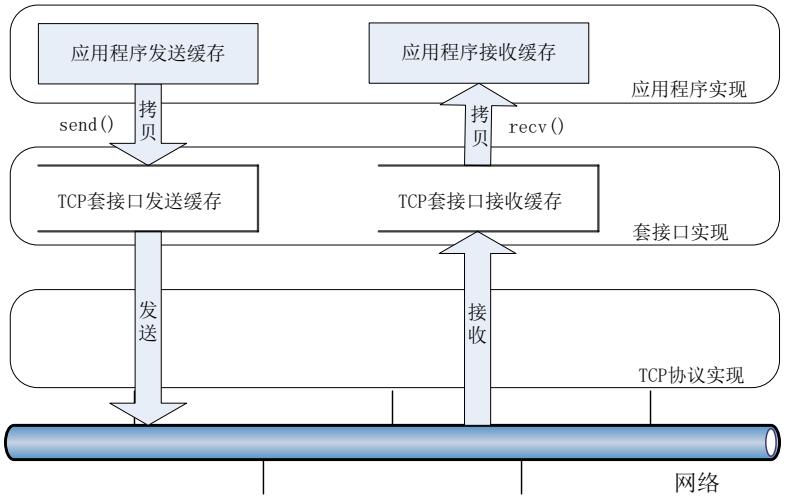
过小的recvbuf　　频繁地调用，消耗资源

过大的recvbuf　　浪费系统的存储资源

（2）接收数据的数量不可预测。

发送端和接收端都等待数据　　死锁！

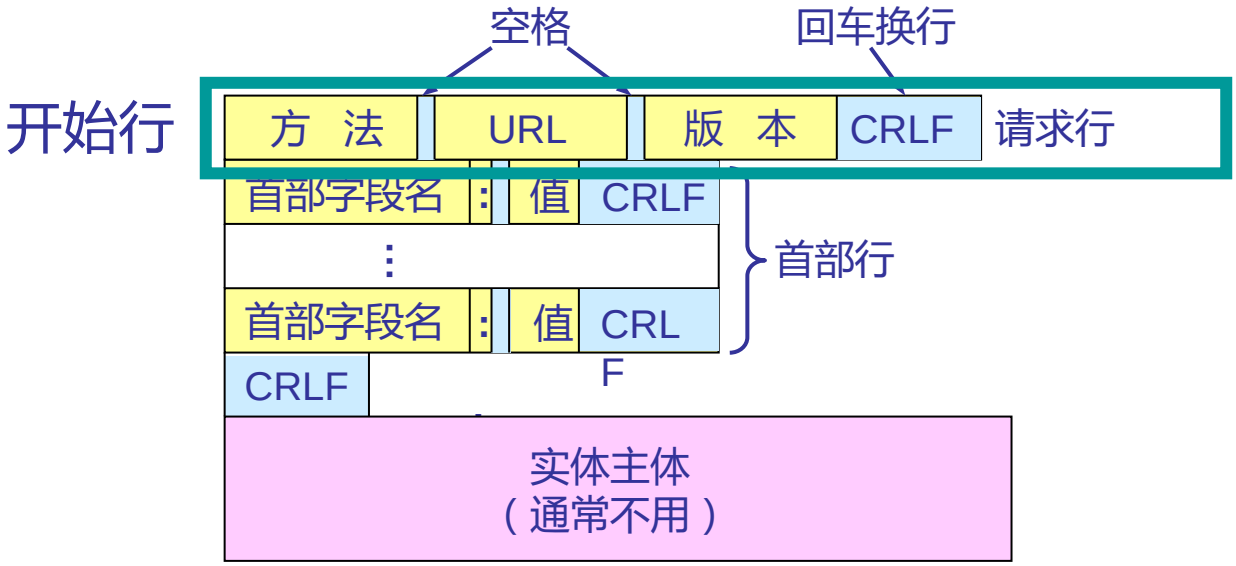
武断截断消息　　错误！



接收方对数据流的逻辑分割

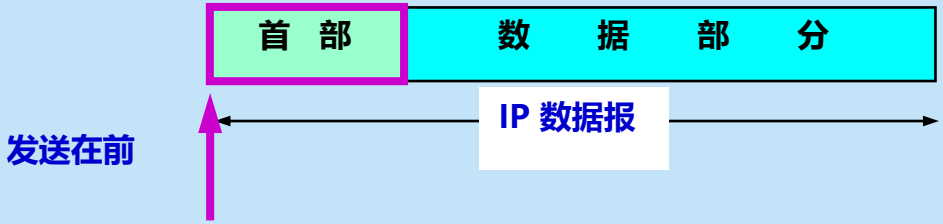
- 网络应用程序的两种数据：
  - 文本数据（比如HTTP协议）
  - 二进字数据（比如IP报文）

HTTP 的报文结构（请求报文）



报文由三个部分组成，即开始行、首部行和实体主体。
在请求报文中，开始行就是请求行。

IP 数据报由首部和数据两部分组成



文本数据的接收

- 文本数据的接收主要要处理好两个方面：
  - 字符编码的约定，收发双方使用的字符编码要一致。
  - 逻辑分割符双方要约定一致，一般使用CRLF(回车换行)

字符编码约定

- ASCII、ISO-8859-1 这种用1个字节编码的字符集，叫做单字节字符集（**SBCS** - Single-Byte Character Set）
- GB 系列这种用1-2、4个不等字节编码的字符集，叫做多字节字符集（**MBCS** - Multi-Byte Character Set）

字符编码约定

- 由 SBCS 编码的数据可以随机访问，从任意字节偏移开始解编码都能保证解析出的字符和后继字符是正确的。
- 而由MBCS编码的数据，只能将其作为字节流进行解析，如果从随机任意的字节偏移开始解编码，有可能定位到切断一个字符的中间位置，导致后继解析出的字符连续出错。作为字节流时，是从某个标识位置进行解析字符，比如从数据的开始位置，或从每个新行符 '/n' 之后开始解析字符。

字符编码约定

- Unicode 至少涉及3个方面：Code Point，UCS，UTF
- 在实际编码之前先给每个穷举到的字符指定一个序号，叫它 **Code Point**，它是数学概念，和用几个字节存储无关。
- 编号时有一些原则，就是越常用的字符越靠前，编号到一定数量后，发现差不多了，常用字符都编完了，截止于此将之前的编号组成的子集叫做基本多文种平面（**BMP** - Basic Multilingual Plane），在 BMP 里的字符，只要4位16进制数就可以表示，当然在 BMP 以外的字符则需要使用5位或更多16进制数表示。

字符编码约定

- 有了 Code Point 后就可以规定它的字符集，叫做 **UCS** - Unicode Character Set，它和存储有关，用2个字节存储 Code Point 叫做 **UCS-2**，用4个字节存储的叫做 **UCS-4**
- UCS 和 Native ANSI 字符集采用的 MBCS 编码是不同的，UCS 不将 ASCII 作为自己的子集，无论什么情况 UCS 总使用定长的字节来编码字符，UCS-2 使用2个字节，UCS-4 使用4个字节

字符编码约定

- 当存储多字节编码的数据并且不将其作为字节流解析时，就要考虑保存数据的大小端问题（**Endian**），可以使用 **BOM**（Byte Order Mark）标识一个 UCS 字符数据块是采用 Big Endian 还是 Little Endian 进行存储。
- 在 Unicode 概念中有一个字符，它的 Code Point 为**U+FEFF**，实际上它不映射到任何地区、国家中的可能字符，即是一个不可能存在字符的 Code Point（Unicode 标准对它的注释为：ZERO WIDTH NO-BREAK SPACE）

字符编码约定

- 当开始处理 UCS 数据块时，UCS 标准建议先处理这个 ZERO WIDTH NO-BREAK SPACE 字符，比如 UCS-2 数据块，如果一开始读到/写入的字节序列是 FF FE（8 进制：377 376），那么说明后续的 UCS-2 按 Little Endian 存储；如果是 FE FF（8进制：376 377），则说明后续的 UCS-2 按 Big Endian 存储。

字符编码约定

- 但是 UCS 也有缺点，
  - 一是有些浪费
  - 二是和 ASCII 不兼容
- UTF - Unicode Transformation Format，作为 Unicode 的传输编码，是对 UCS 再次编码映射得到的字符集，能够一定程度上解决上面 UCS 的2个缺点。

字符编码约定

- 有一种方法可以让数据块标识自己使用的是 UTF-8 编码，这个标识方法就是使用 UCS-2 中 BOM 的 UTF-8 编码，其1字节流为：EF BB BF（8 进制：357 273 277）。当数据块的开始有这个流时就说明后续字符采用 UTF-8 编码。
- UTF-8 的BOM 已经失去其在 UCS-2 中作为标识字节序大小端的作用，仅作为 UTF-8 编码的标识功能（Magic），有时就叫它 **UTF-8 Signature**。

字符编码约定

- UTF-8最适合用来作为字符串网络传输的编码格式。UTF-16最适合当作本地字符串编码格式。
- 如果定义好了网络传输协议，应用限于WINDOWS平台。那么UTF-16也非常合适当作网络字符串传输的编码格式，特别是中文等远东地区字符集。比起UTF-8来说，节省一点点流量。

源文件字符集的影响

- UTF-16 LE是windows上默认的Unicode编码方式，使用wchar_t表示。所有wchar_t *类型的字符串(包括硬编码在.h/.cpp里的字符串字面值)，VC都自动采用UTF-16的编码
- char *类型的字面值，最终内存使用何种编码方式完全取决于当前文件的编码方式。也就是说当前文件如果是GBK编码的，那么文件里char * str = "中午"，str指向的内存字符串二进制是使用GBK编码的。如果文件编码是UTF-8，那么内存是使用UTF-8编码。所以为什么一直要强调字符串应该放在资源文件里，而不是硬编码在.h/.cpp文件里！

字符集转换函数

- **MultiByteToWideChar**从其它编码转为UTF-16
- **WideCharToMultiByte**从UTF-16转为其它编码

```
int MultiByteToWideChar(
  [in]            UINT                                CodePage,
  [in]            DWORD                               dwFlags,
  [in]            _In_NLS_string_(cbMultiByte)LPCCH   lpMultiByteStr,
  [in]            int                                 cbMultiByte,
  [out, optional] LPWSTR                              lpWideCharStr,
  [in]            int                                 cchWideChar
);
```

CodePage:表示从哪个码页转为UTF-16, UTF-8取值**CP_UTF8**，936为简体中文GB2312代码页

dwFlags：用来指示转换类型，一般为0.

lpMultiByteStr：指向待转换的字符串

cbMultiByte：指定由参数lpMultiByteStr指向的字符串中字节的个数。如果lpMultiByteStr指定的字符串以空字符终止，可以设置为-1

lpWideCharStr：指向存储转换后的字符串的缓冲区

cchWideChar：指定由参数lpWideCharStr指向的缓冲区的宽字符个数。若此值为零，函数返回缓冲区所必需的宽字符数，在这种情况下，lpWideCharStr中的缓冲区不被使用。

返回值：如果函数运行成功，并且cchWideChar不为零，返回值是由lpWideCharStr指向的缓冲区中写入的宽字符数；如果函数运行成功，并且cchWideChar为零，返回值是接收到待转换字符串的缓冲区所需求的宽字符数大小。如果函数运行失败，返回值为零。若想获得更多错误信息，请调用GetLastError函数。

详细

```
int WideCharToMultiByte(
  [in]            UINT                                CodePage,
  [in]            DWORD                               dwFlags,
  [in]            _In_NLS_string_(cchWideChar)LPCWCH lpWideCharStr,
  [in]            int                                 cchWideChar,
  [out, optional] LPSTR                               lpMultiByteStr,
  [in]            int                                 cbMultiByte,
  [in, optional]  LPCCH                               lpDefaultChar,
  [out, optional] LPBOOL                              lpUsedDefaultChar
);
```

CodePage:转换后的码页
dwFlags:转换类型，一般为0
lpWideCharStr:指向待转换的宽字符串的长度，如果这个值为-1，字符串将被设定为以NULL为结束符的字符串，并且自动计算长度。
lpMultiByteStr: 指向接收缓冲区
cbMultiByte:指定由参数lpMultiByteStr指向的缓冲区最大值（用字节来计量）。若此值为零，函数返回接收缓冲区所必需的字节数，在这种情况下，lpMultiByteStr参数通常为NULL。
lpDefaultChar:如果宽字节字符不能被转换，该函数便使用lpDefaultChar参数指向的字符。如果该参数是NULL（这是大多数情况下的参数值），那么该函数使用系统的默认字符
lpUsedDefaultChar:指向一个布尔变量，用来标识转换的结果是否有用到DefaultChar.
返回值：如果函数运行成功，并且cchMultiByte不为零，返回值是由 lpMultiByteStr指向的缓冲区中写入的字节数；如果函数运行成功，并且cchMultiByte为零，返回值是接收到待转换字符串的缓冲区所必需的字节数。如果函数运行失败，返回值为零。若想获得更多错误信息，请调用GetLastError函数。

详细

```
//UTF8转ANSI
void UTF8toANSI(CString &strUTF8)
{
 //获取转换为多字节后需要的缓冲区大小，创建多字节缓冲区
 UINT nLen = MultiByteToWideChar(CP_UTF8,NULL,strUTF8,-1,NULL,NULL);
 WCHAR *wszBuffer = new WCHAR[nLen+1];
 nLen = MultiByteToWideChar(CP_UTF8,NULL,strUTF8,-1,wszBuffer,nLen);
 wszBuffer[nLen] = 0;

 nLen = WideCharToMultiByte(936,NULL,wszBuffer,-1,NULL,NULL,NULL,NULL);
 CHAR *szBuffer = new CHAR[nLen+1];
 nLen = WideCharToMultiByte(936,NULL,wszBuffer,-1,szBuffer,nLen,NULL,NULL);
 szBuffer[nLen] = 0;

 strUTF8 = szBuffer;
 //清理内存
 delete []szBuffer;
 delete []wszBuffer;
}
```

```
//ANSI转UTF8
void ANSItoUTF8(CString &strAnsi)
{
 //获取转换为宽字节后需要的缓冲区大小，创建宽字节缓冲区，936为简体中文GB2312代码页
 UINT nLen = MultiByteToWideChar(936,NULL,strAnsi,-1,NULL,NULL);
 WCHAR *wszBuffer = new WCHAR[nLen+1];
 nLen = MultiByteToWideChar(936,NULL,strAnsi,-1,wszBuffer,nLen);
 wszBuffer[nLen] = 0;
 //获取转为UTF8多字节后需要的缓冲区大小，创建多字节缓冲区
 nLen = WideCharToMultiByte(CP_UTF8,NULL,wszBuffer,-1,NULL,NULL,NULL,NULL);
 CHAR *szBuffer = new CHAR[nLen+1];
 nLen = WideCharToMultiByte(CP_UTF8,NULL,wszBuffer,-1,szBuffer,nLen,NULL,NULL);
 szBuffer[nLen] = 0;

 strAnsi = szBuffer;
 //内存清理
 delete []wszBuffer;
 delete []szBuffer;
}
```

换行

- 在 Web 的发展过程中，UNIX 系统从一开始到现在一直是主要的开发和运行平台，但是在 HTTP 协议中，各个 header 之间用的却是 CRLF (\r\n)这样的 Windows/DOS 换行方式，而不是 UNIX 普遍的的 LF (\n)换行方式，看上去十分冗长和诡异，为什么会这样？

换行

- Carriage return，简称CR，Carriage return翻译过来就是“小车回位”（将打印头小车推回行首），即“回车”。
- LINE FEED，即LF，作用是将打印纸向上推动一行，即换行。
- RFC158出现将命令统一为CR+LF，微软和大多数互联网标准用CRLF做换行符。但很多厂商由于历史原因仍然不遵守。
- \r的ascii 码是：13
- \n的ascii 码是：10

\0字符的处理

- \0字符被C语言看作是字符串的结尾
- 尽量不要传输\0字符，除非使用\0字符做分割符。

流数据的接收

3.2 接收定长数据

模拟定长数据包的形态处理底层提交的字节流数据。

- 预先给定了接收数据的总长度

- 增加接收总长度的判断

流数据的接收

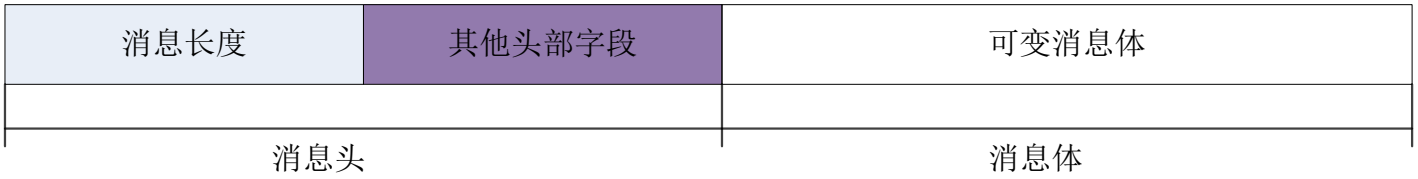
```
int recvn(SOCKET s, char * recvbuf, unsigned int fixedlen)
{
    int iResult;    //存储单次recv操作的返回值
    int cnt;        //用于统计相对于固定长度，剩余多少字节尚未接收
    cnt = fixedlen;
    while ( cnt > 0 ) {
        iResult = recv(s, recvbuf, cnt, 0);
        if ( iResult < 0 ){
            //数据接收出现错误，返回失败
            printf("接收发生错误: %d\n", WSAGetLastError());
            return -1;
        }
        if ( iResult == 0 ){
            //对方关闭连接，返回已接收到的小于fixedlen的字节数
            printf("连接关闭\n");
            return fixedlen - cnt;
        }
        //printf("接收到的字节数: %d\n", iResult);
        //接收缓存指针向后移动
        recvbuf +=iResult;
        //更新cnt值
        cnt -=iResult;
    }
    return fixedlen;
}
```

流数据的接收

3.3 接收变长数据

用长度字段显式指明字节流的数据长度。

- 设计消息格式存储长度字段



- 两次定长接收数据

流数据的接收

3.3接收变长数据

```
int recvvl(SOCKET s, char * recvbuf, unsigned int recvbuflen)
{
    int iResult;//存储单次recv操作的返回值
    unsigned int reclen; //用于存储报文头部存储的长度信息
    //获取接收报文长度信息
    iResult = recvn(s, ( char * )&reclen, sizeof( unsigned int ));
    if ( iResult !=sizeof ( unsigned int ) {
        //如果长度字段在接收时没有返回一个整型数据就返回（连接关闭）或-1（发生错误）
        if ( iResult == -1 ) {
            printf("接收发生错误: %d\n", WSAGetLastError());
            return -1;
        }
        else {
             printf("连接关闭\n");
             return 0;
        }
    }
    //转换网络字节顺序到主机字节顺序
    reclen = ntohl( reclen );
```

流数据的接收

3.3接收变长数据

```
if ( reclen > recvbuflen ) {
        //如果recvbuf没有足够的空间存储变长消息，则接收该消息并丢弃，返回错误
        while ( reclen > 0){
            iResult = recvn( s, recvbuf, recvbuflen );
            if ( iResult != recvbuflen ) {
                //如果变长消息在接收时没有返回足够的数据就返回（连接关闭）或-1（发生错误）
                if ( iResult == -1 ) {
                    printf("接收发生错误: %d\n", WSAGetLastError());
                    return -1;
                }
                else {
                    printf("连接关闭\n");
                    return 0;
                }
            }
            reclen -= recvbuflen;
            //处理最后一段数据长度
            if ( reclen < recvbuflen )
                recvbuflen = reclen;
        }
        printf("可变长度的消息超出预分配的接收缓存\r\n");
        return -1;
    }
```

流数据的接收

3.3接收变长数据

```
//接收可变长消息
    iResult = recvn( s, recvbuf, reclen );
    if ( iResult != reclen ){
        //如果消息在接收时没有返回足够的数据就返回（连接关闭）或-1（发生错误）
        if ( iResult == -1 ) {
            printf("接收发生错误: %d\n", WSAGetLastError());
            return -1;
        }
        else {
            printf("连接关闭\n");
            return 0;
        }
    }
    return iResult;
}
```

内容总结

TCP的流传送特点

- 字节流的特点
- TCP对字节流的处理
- 数据接收的可能情况

从套接字接口层观察流的传送

- 套接字接口层的位置与内容
- 数据发送和接收过程中的缓冲现象

合理处理流数据的接收

- 合理判断接收返回值
- 接收定长数据
- 接收变长数据

面向连接程序的可靠性保护

发送成功不等于发送有效

- 数据发送涉及两个层次的写操作：
  - 从应用程序发送缓冲区拷贝数据到TCP套接字的发送缓存
  - 从TCP套接字的发送缓存将数据发送到网络中
- 流式套接字的send()函数调用成功仅仅表示我们可以重新使用应用进程缓冲区，并不意味着数据已经发出主机，更不能理解为对方已经接收到数据。

从应用程序角度观察发送操作

- send()函数调用结果：
  - 1）阻塞：send()函数一直等到TCP套接字发送缓冲区中的原有数据被释放，能够容纳应用程序即将发送的数据时，send()函数才会成功返回.
  - 2）返回错误。通常是由非法操作或网络异常导致。
  - 3）成功返回。这时，send函数完成如下操作：
    - 从应用程序的发送缓冲区移动数据到TCP套接字的发送缓存中
    - 通知TCP协议实现该应用程序有新的数据等待发送。

从应用程序角度观察发送操作

- send()操作成功返回后，数据发送的可能情况。
  - 数据立即发送。
  - 数据排队等待传输。(比如Nagle算法的影响)
  - 数据被传输一部分。(套接字发送缓冲区不够大)
  - 发送失败。要等接下来的数据接收操作时才获得接收错误。因此，发送函数的成功返回并不能保证发送的最终结果是成功的。

从TCP观察发送操作

- (1) MSS对数据发送的影响
  - MSS的取值派生于网络MTU
  - 如果发送的数据大于MSS,TCP会按照MSS限制来拆分数据，然后依次发送，并在最后一个报文中带上PSH标志
  - 如果发送的数据小于MSS,TCP会为发出的数据带上PSH标志
  - 发送到网络中的TCP分段长度受限于数据本身的长度和MSS值。

从TCP观察发送操作

- (2)TCP流量控制对数据发送的影响
  - 滑动窗口机制
  - 发送的数据长度受限于对等方的接收缓冲区大小
- (3)TCP拥塞控制对数据发送的影响
  - 拥塞窗口和对等方的接收窗口共同限制了TCP发送的数据长度。

Nagle算法

- Nagle算法的基本定义是任意时刻，最多只能有一个未被确认的小段。所谓“小段”，指的是小于MSS尺寸的数据块，所谓“未被确认”，是指一个数据块发送出去后，没有收到对方发送的ACK确认该数据已收到。

Nagle算法

- 若发送应用进程把要发送的数据逐个字节地送到 TCP 的发送缓存，则发送方就把第一个数据字节先发送出去，把后面到达的数据字节都缓存起来。
- 当发送方收到对第一个数据字符的确认后，再把发送缓存中的所有数据组装成一个报文段发送出去，同时继续对随后到达的数据进行缓存。
- 只有在收到对前一个报文段的确认后才继续发送下一个报文段。
- 当到达的数据已达到发送窗口大小的一半或已达到报文段的最大长度时，就立即发送一个报文段。

Nagle算法

- Nagle算法的规则（可参考tcp_output.c文件里tcp_nagle_check函数注释）：
  - （1）如果包长度达到MSS，则允许发送；
  - （2）如果该包含有FIN，则允许发送；
  - （3）设置了TCP_NODELAY选项，则允许发送；
  - （4）未设置TCP_CORK选项时，若所有发出去的小数据包（包长度小于MSS）均被确认，则允许发送；
  - （5）上述条件都未满足，但发生了超时（一般为200ms），则立即发送。
- 注： TCP_CORK选项。开启时，内核将阻塞不完整的报文，不完整指的是应用层发送的数据长度不足一个MSS长度。TCP_CORK最多只能将数据阻塞200毫秒
- TCP_CORK选项是Linux特有的

正确处理TCP的失败模式

- 1.TCP的可靠性服务
  - TCP提供的可靠性仅仅是在传输层两个端点之间的可靠性。
  - 从应用程序来看，数据被可靠接收并不意味着数据已经成功交付给应用程序了，也不意味着它一定会传递到。
  - 应用程序的可靠性需要应用程序自己提供。

正确处理TCP的失败模式

- 2.使用TCP传输的失败模式
  - 失败现象一: 在正常的TCP连接上，TCP确认的数据实际上有可能不会到达它的目的应用程序。
    - 解决方法：应用程序对等方发送确认信息。
  - 失败现象二：服务器的TCP实现不确认接收到了数据。通常，当连接中断时这类失败故障就会发生。
- 导致TCP连接故障的可能事件有。
  - 发生永久的或暂时的网络紊乱
  - 对等方的应用程序崩溃
  - 对等方的应用程序运行的主机崩溃

正确处理TCP的失败模式
3.程序对网络紊乱现象的处理

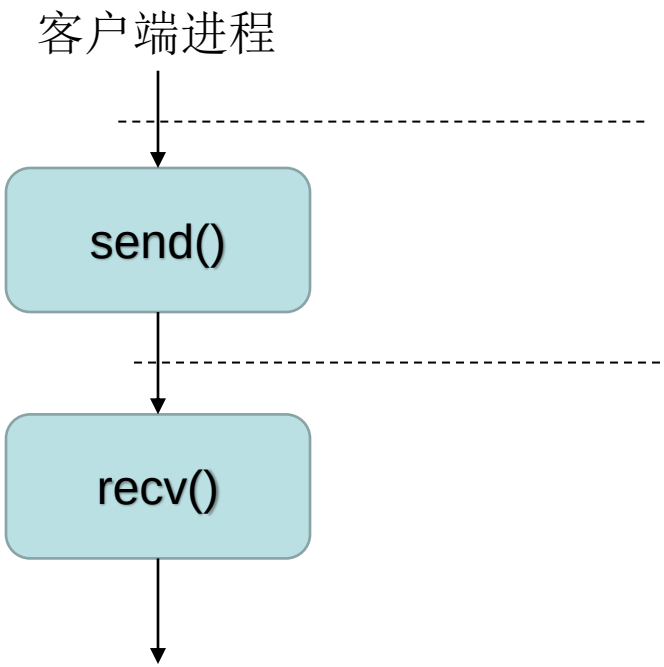
- 当网络紊乱发生时，从发送方来看，可能会出现两种情况
  - 1）路由器发送网络或主机不可达错误。
  - 2）发送超时或发送错误。
  - 以上两种情况，应用程序的发送操作都会以错误返回。程序中应增加对发送错误的判断和相应处理。
- 当网络发生紊乱时，从接收方角度看，接收程序可能会长时间保持接收的阻塞状态。
  - 解决方法：增加心跳机制或更改TCP的Keep Alive选项。

正确处理TCP的失败模式

4.程序对对等方应用程序崩溃现象的处理

- 当对等方应用程序B崩溃或中断时，从协议角度观察，应用程序B的TCP会发送FIN报文段给连接的另一方。
- 从应用程序A的角度观察，应用程序B主动关闭连接和程序崩溃的表现是一样的，需要对FIN的接收做正确的处理。

对等方应用程序崩溃的不同场景



场景2：在客户调用send()前服务器崩溃

场景1：在客户调用recv()前服务器崩溃

正确处理TCP的失败模式

4.程序对对等方应用程序崩溃现象的处理

- 场景1：在客户调用recv()函数前服务器崩溃
- 客户能够收到服务器发来的正常数据和结束标志。
- 如果服务器在发送回客户的正常响应之后崩溃，则崩溃现象跟服务器主动调用closesocket()函数产生的TCP行为一致。

正确处理TCP的失败模式
4.程序对对等方应用程序崩溃现象的处理

- 场景2：在客户调用send()函数前服务器崩溃
- 服务器崩溃会导致TCP实现发送FIN标志给客户，由于客户没有调用recv()，并不知晓从服务器发来的结束标志。
- 客户调用send()，正常返回，服务器返回一个RST标志。
- 若这时客户调用recv()就会接收到这个RST,函数返回连接重置错误(WSAENETRESET)
- 若这时客户调用send(),就会收到发送时的连接重置错误(WSAENETRESET)。这是因为TCP实现收到对方发来的RST标志，会主动释放本方已保存的该连接信息。

正确处理TCP的失败模式

4.程序对对等方应用程序崩溃现象的处理

- 服务器崩溃所产生的FIN标志以及之后对客户端请求的RST标志会使得客户在调用send函数或recv函数时发生返回值的变化。
- 如果客户程序能够对这些返回值和差错类型进行合理判断，就能够正确处理网络通信中的这类失败模式。

正确处理TCP的失败模式
5.程序对对等方主机崩溃现象的处理

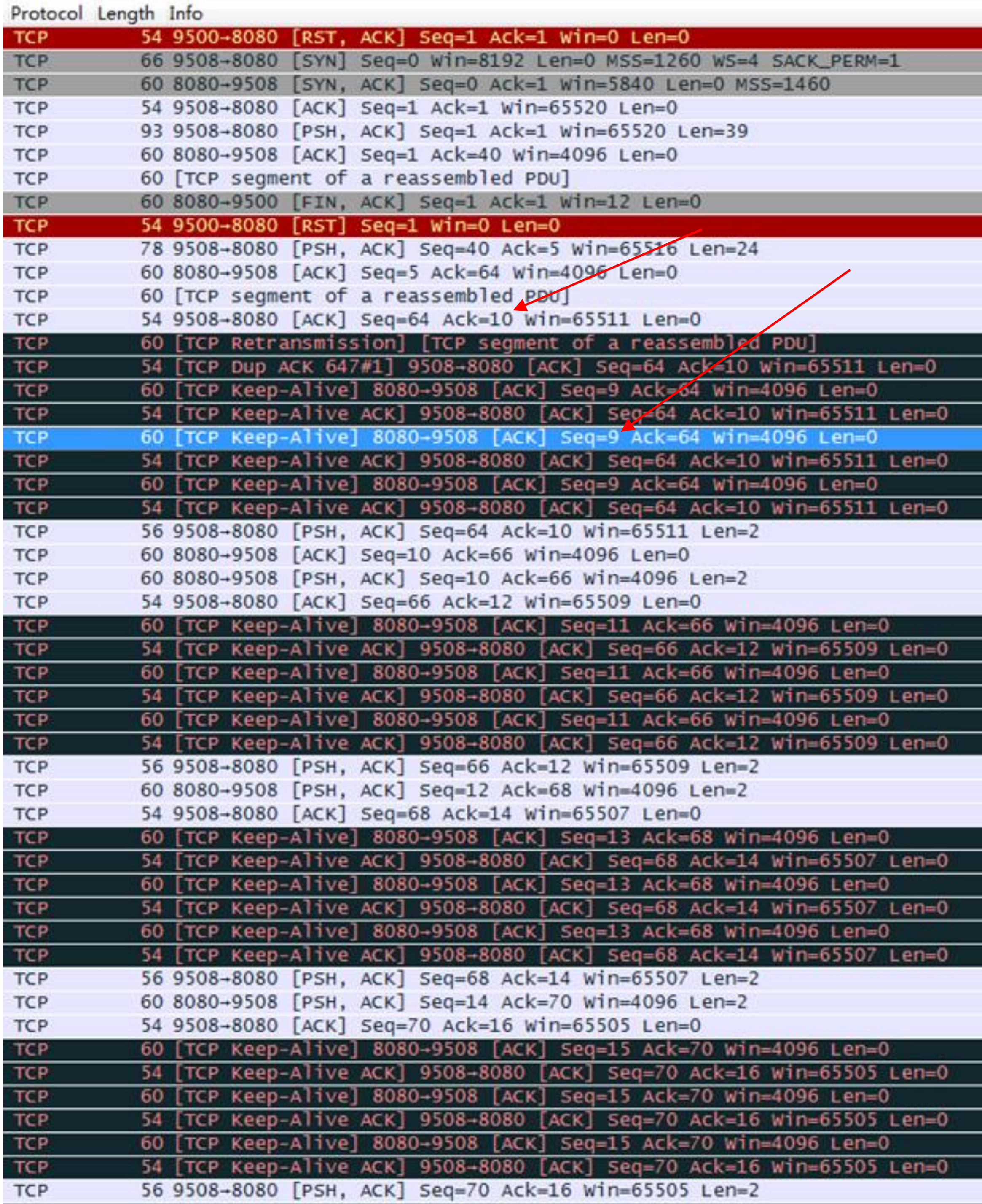
- 对等方主机崩溃使得其TCP来不及发送FIN来通过对方应用程序已经不再运行。
- 在崩溃的对等主机重启前，这种现象与网络紊乱失败模式很相似。
- 如果对等主机不重启，发送方最后会放弃并返回发送超时错误给本方应用程序。
- 在崩溃的对等方重启后，若发送方主机的TCP尚未放弃和撤销TCP连接，发送方发送数据会得到RST应答，发送方的应用程序会得到连接重置错误(WSAENETRESET).

检测无即时通知的死连接

- 假如应用程序不再发送数据，而是阻塞等待一个无效连接上的请求，那么这种等待可能会一直持续下去。
- 目前，对于TCP连接监控可采用的方法主要有以下两类
  - 1)利用Keep Alive机制实现TCP连接监控
  - 2)利用心跳机制实现TCP连接监控

TCP KeepAlive

1. TCP Keepalive虽不是标准规范，但操作系统一旦实现，默认情况下须为关闭，可以被上层应用开启和关闭。
2. TCP Keepalive必须在没有任何数据（包括ACK包）接收之后的周期内才会被发送，允许配置，默认值不能够小于2个小时
3. 不包含数据的ACK段在被TCP发送时没有可靠性保证，意即一旦发送，不确保一定发送成功。系统实现不能对任何特定探针包作死连接对待
4. 规范建议keepalive保活包不应该包含数据，但也可以包含1个无意义的字节，比如0x0。
5. SEG.SEQ = SND.NXT-1，即TCP保活探测报文序列号将前一个TCP报文序列号减1。SND.NXT = RCV.NXT，即下一次发送正常报文序号等于ACK序列号；总之保活报文不在窗口控制范围内



Protocol	Length	Info
TCP	54	9500→8080 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
TCP	66	9508→8080 [SYN] Seq=0 Win=8192 Len=0 MSS=1260 WS=4 SACK_PERM=1
TCP	60	8080→9508 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460
TCP	54	9508→8080 [ACK] Seq=1 Ack=1 Win=65520 Len=0
TCP	93	9508→8080 [PSH, ACK] Seq=1 Ack=1 Win=65520 Len=39
TCP	60	8080→9508 [ACK] Seq=1 Ack=40 Win=4096 Len=0
TCP	60	[TCP segment of a reassembled PDU]
TCP	60	8080→9500 [FIN, ACK] Seq=1 Ack=1 Win=12 Len=0
TCP	54	9500→8080 [RST] Seq=1 Win=0 Len=0
TCP	78	9508→8080 [PSH, ACK] Seq=40 Ack=5 Win=65516 Len=24
TCP	60	8080→9508 [ACK] Seq=5 Ack=64 Win=4096 Len=0
TCP	60	[TCP segment of a reassembled PDU]
TCP	54	9508→8080 [ACK] Seq=64 Ack=10 Win=65511 Len=0
TCP	60	[TCP Retransmission] [TCP segment of a reassembled PDU]
TCP	54	[TCP Dup ACK 647#1] 9508→8080 [ACK] Seq=64 Ack=10 Win=65511 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=9 Ack=64 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=64 Ack=10 Win=65511 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=9 Ack=64 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=64 Ack=10 Win=65511 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=9 Ack=64 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=64 Ack=10 Win=65511 Len=0
TCP	56	9508→8080 [PSH, ACK] Seq=64 Ack=10 Win=65511 Len=2
TCP	60	8080→9508 [ACK] Seq=10 Ack=66 Win=4096 Len=0
TCP	60	8080→9508 [PSH, ACK] Seq=10 Ack=66 Win=4096 Len=2
TCP	54	9508→8080 [ACK] Seq=66 Ack=12 Win=65509 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=11 Ack=66 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=66 Ack=12 Win=65509 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=11 Ack=66 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=66 Ack=12 Win=65509 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=11 Ack=66 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=66 Ack=12 Win=65509 Len=0
TCP	56	9508→8080 [PSH, ACK] Seq=66 Ack=12 Win=65509 Len=2
TCP	60	8080→9508 [PSH, ACK] Seq=12 Ack=68 Win=4096 Len=2
TCP	54	9508→8080 [ACK] Seq=68 Ack=14 Win=65507 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=13 Ack=68 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=68 Ack=14 Win=65507 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=13 Ack=68 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=68 Ack=14 Win=65507 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=13 Ack=68 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=68 Ack=14 Win=65507 Len=0
TCP	56	9508→8080 [PSH, ACK] Seq=68 Ack=14 Win=65507 Len=2
TCP	60	8080→9508 [PSH, ACK] Seq=14 Ack=70 Win=4096 Len=2
TCP	54	9508→8080 [ACK] Seq=70 Ack=16 Win=65505 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=15 Ack=70 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=70 Ack=16 Win=65505 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=15 Ack=70 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=70 Ack=16 Win=65505 Len=0
TCP	60	[TCP Keep-Alive] 8080→9508 [ACK] Seq=15 Ack=70 Win=4096 Len=0
TCP	54	[TCP Keep-Alive ACK] 9508→8080 [ACK] Seq=70 Ack=16 Win=65505 Len=0
TCP	56	9508→8080 [PSH, ACK] Seq=70 Ack=16 Win=65505 Len=2

TCP KeepAlive

1. 不太好的TCP堆栈实现，可能会要求保活报文必须携带有1个字节的数据负载
2. TCP Keepalive应该在服务器端启用，客户端不做任何改动；若单独在客户端启用，若客户端异常崩溃或出现连接故障，存在服务器无限期的为已打开的但已失效的文件描述符消耗资源的严重问题。但在特殊的NFS文件系统环境下，需要客户端和服务器端都要启用Tcp Keepalive机制。
3. TCP Keepalive不是TCP规范的一部分，有三点需要注意：
    1. 在短暂的故障期间，它们可能引起一个良好连接（good connection）被释放（dropped）
    2. 它们消费了不必要的宽带
    3. 在以数据包计费的互联网消费（额外）花费金钱

Linux系统内核参数

1. tcp_keepalive_time，在TCP保活打开的情况下，最后一次数据交换到TCP发送第一个保活探测包的间隔，即允许的持续空闲时长，或者说每次正常发送心跳的周期，默认值为7200s（2h）。
2. tcp_keepalive_probes 在tcp_keepalive_time之后，没有接收到对方确认，继续发送保活探测包次数，默认值为9（次）
3. tcp_keepalive_intvl，在tcp_keepalive_time之后，没有接收到对方确认，继续发送保活探测包的发送频率，默认值为75s。

```cpp
#include <iostream>
#include <WinSock2.h>
#include <mstcpip.h>
#include "WinSockTool.h"
#pragma comment(lib,"Ws2_32.lib")
```

```cpp
//
// Argument structure for SIO_KEEPALIVE_VALS.
//
struct tcp_keepalive {
    ULONG onoff;
    ULONG keepalivetime;
    ULONG keepaliveinterval;
};
```

按毫秒计时

```cpp
BOOL WinSockTool::enableKeepAlive(SOCKET socket_handle)
{
    //开启Keep Alive选项
    BOOL bKeepAlive = TRUE;
    int nRet = setsockopt(socket_handle,SOL_SOCKET,SO_KEEPALIVE,(char *)&bKeepAlive,sizeof(bKeepAlive));
    if (nRet == SOCKET_ERROR) {
        return FALSE;
    }
    //设置Keep Alive参数
    tcp_keepalive alive_in = { 0 };
    tcp_keepalive alive_out = { 0 };
    alive_in.keepalivetime = 5000; //开始首次Keep Alive探测前的TCP空闲时间
    alive_in.keepaliveinterval = 1000; //两次Keep Alive探测间的时间间隔
    alive_in.onoff = TRUE;
    unsigned long ulBytesReturn = 0;
    nRet = WSAIoctl(socket_handle,SIO_KEEPALIVE_VALS,&alive_in,sizeof(alive_in),&alive_out,sizeof(alive_out),&ulBytesReturn,NULL,NULL);
    if (nRet==SOCKET_ERROR) {
        return FALSE;
    }
}
```

TCP Keepalive 引发的错误

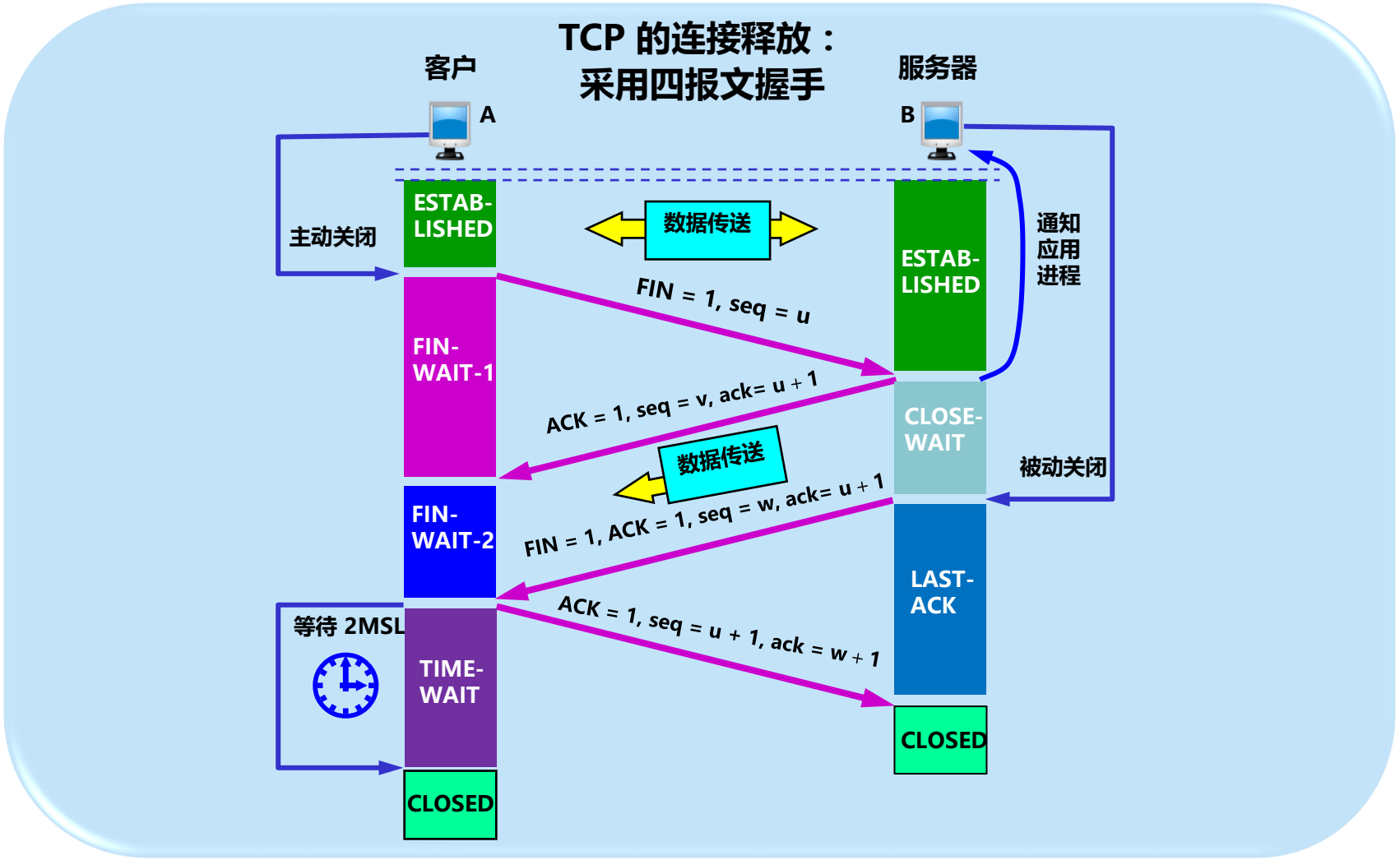
- ETIMEOUT 超时错误，在发送一个探测保护包经过(tcp_keepalive_time + tcp_keepalive_intvl * tcp_keepalive_probes)时间后仍然没有接收到ACK确认情况下触发的异常，套接字被关闭
- EHOSTUNREACH host unreachable(主机不可达)错误，这个应该是ICMP汇报给上层应用的。
- 链接被重置，终端可能崩溃死机重启之后，接收到来自服务器的报文，然物是人非，前朝往事，只能报以无奈重置宣告之。

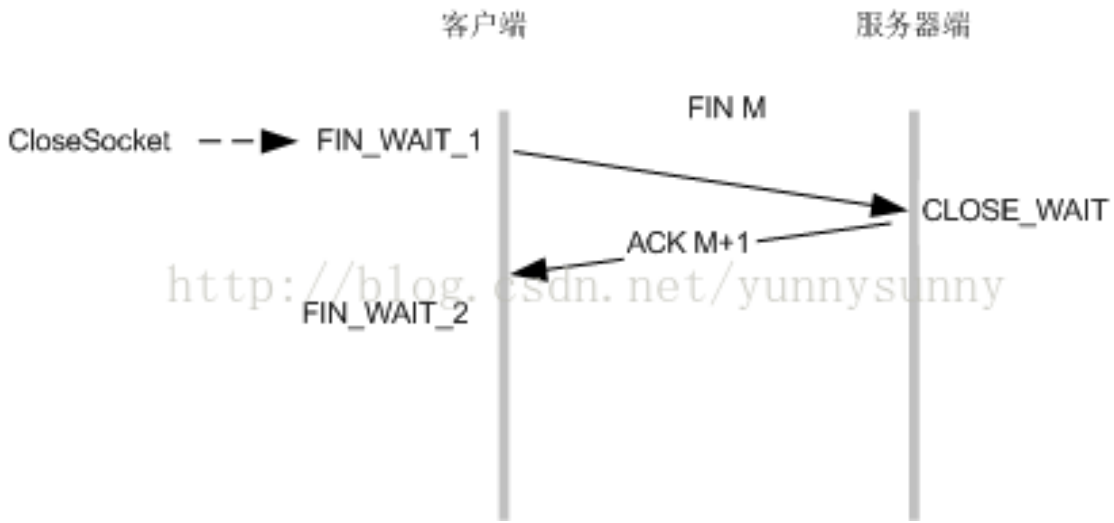
利用心跳机制实现TCP连接监控

- 由应用程序自己发送心跳信息来检测连接的健康性。
- TCP的心跳机制只能说明协议栈正常。但存在协议栈正常应用层死锁的情况，对这类应用需要应用层的心跳机制。
- 依赖于应用程序传送的数据形态不同，心跳机制的实现方式也会有所不同
  - 客户和服务器交换几个不同类型的消息。
    - 引入一个新的消息类型
  - 客户和服务器交换的内容是字节流，没有内在的记录或消息概念
    - 使用独立连接进行通信

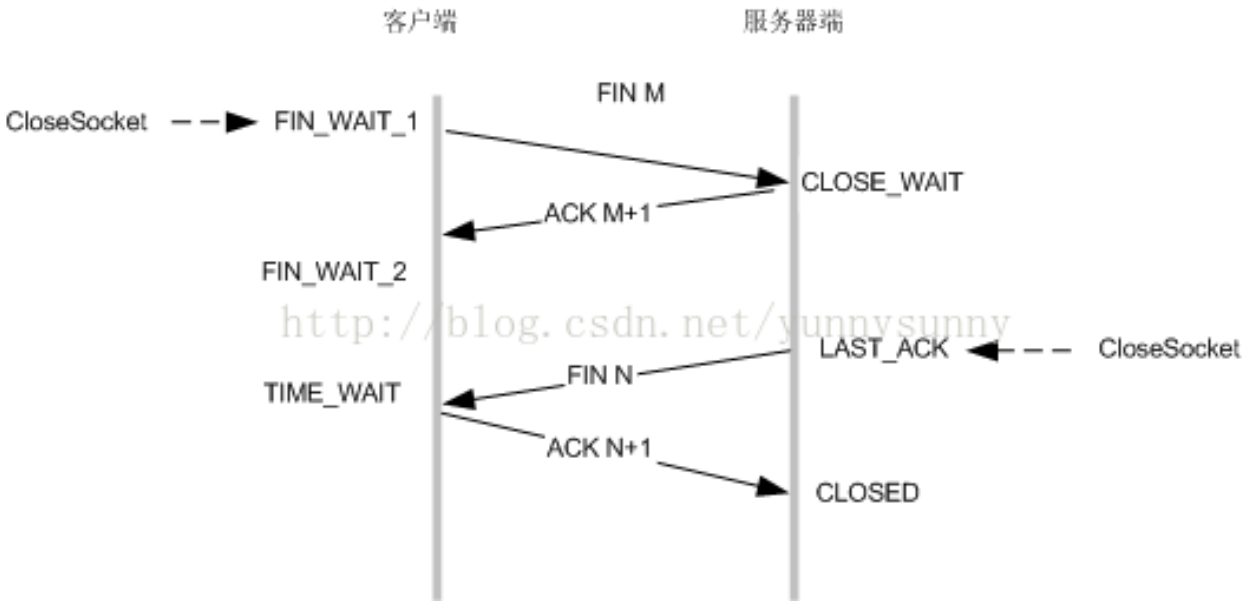
顺序释放连接
1.closesocket操作与潜在弊端

- 调用closesocket可以释放一个已经被打开的套接字句柄的资源，以后对该套接字的访问均以WSAENOTSOCK错误返回。
- closesocket()调用可能的负面影响就是数据丢失。
- 对流式套接字而言，在调用closesocket()时，选项SO_LINGER和SO_DONTLINGER的配置决定了closesocket()的操作过程.
- closesocket详细内容


TCP 的连接释放：
采用四报文握手
客户
服务器
A
B
主动关闭
ESTAB-
LISHED
数据传送
ESTAB-
LISHED
通知
应用
进程
FIN = 1, seq = u
FIN-
WAIT-1
ACK = 1, seq = v, ack= u + 1
CLOSE-
WAIT
数据传送
被动关闭
FIN-
WAIT-2
FIN = 1, ACK = 1, seq = w, ack= u + 1
LAST-
ACK
等待 2MSL
ACK = 1, seq = u + 1, ack = w + 1
TIME-
WAIT
CLOSED
CLOSED



客户端调用CloseSocket,服务端未调用CloseSocket情形。
客户端并不会一直处在 `FIN_WAIT_2`状态，因为这个状态有一个超时时间。
客户端超时后就进入TIME_WAIT状态。
服务器端一直处于 `CLOSE_WATI`状态的情况，句柄得不到释放。



服务端通过recv返回值为0得知客户端调用closesocket。
服务端调用CloseSocket使得连接正常结束。

```
typedef struct linger {
  u_short l_onoff;  //声明调用closesocket()之后是否等待一段时间再关闭套接字
  u_short l_linger; //声明具体延迟等待的秒数
} LINGER, *PLINGER, *LPLINGER;
```

详细信息

- 设置选项SO_DONTLINGER值为0或设置选项SO_LINGER时把l_onoff参数设置为1是相同效果
- 默认情况下SO_DONTLINGER非0

调用closesocket()后TCP实现的行为

- 1)l_onoff为0: 关闭LINGER选项，l_linger值将被忽略，TCP采用默认设置对closesocket()进行操作。
- 即立刻执行连接停止操作，closesocket()立刻返回，在TCP实现中，已经提交待发送的数据将会继续由协议栈发送出去，然后关闭套接字。closesocket()函数的返回不关心之前发送的数据是否被对等方TCP确认。(优雅关闭)
- 特殊情况下可能会发生数据丢失现象：客户的closesocket()可能在服务器接收套接字缓冲区中还剩余数据之前返回，服务器应用程序尚未来得及读取完这些剩余数据，服务主机就崩溃了。客户应用进程并不会获知之前已发生的数据已丢失。

调用closesocket()后TCP实现的行为

- 2)l_onoff为非0，l_linger为0. 强制或失效关闭
- Closesocket()不被阻塞立即执行，TCP将丢弃保留在套接字发送缓冲区中的任何数据，并发送一个RST给对等方。远端的recv()调用将以WSAECONNRESET出错。

so_linger.l_onoff = 1;
so_linger.l_linger = 0;
将TCP的四次挥手直接禁用掉

服务器端如果接受到这种RST消息，会直接把对应的socket句柄回收掉。

有一些socket程序不想让TCP出现`TIME_WAIT`状态，会选择直接使用RST方式关闭socket，以保证socket句柄在最短的时间内得到回收，当然前提是接受有可能被丢弃老的数据包这种情况的出现。

如果socket通信的前后数据包的关联性不是很强的话，换句话说每次通信都是一个单独的事务，那么可以考虑直接发送RST信号来快速关闭连接。

调用closesocket()后TCP实现的行为

- 3)l_onoff非0，l_linger非0
- 阻塞模式下，closesocket()调用阻塞进程，直到数据发送完毕或超时。成功返回，说明先前发送的数据和FIN已由对等方的TCP确认了。若超时，则套接字发送缓冲区中的任何残留数据都被丢弃。
- 非阻塞模式下，closesocket()调用将以WSAEWOULDBLOCK错误返回，应再次调用closesocket()函数等待数据发送完毕关闭。
- 在非阻塞模式下将l_onoff和l_linger参数都设为非0在MSDN中是不推荐的。

l_onoff	l_linger	**Type of close**	**Wait for close?**
zero	Do not care	Graceful close	No
nonzero	zero	Hard	No
nonzero	nonzero	Graceful if all data is sent within timeout value specified in the **l_linger** member. Hard if all data could not be sent within timeout value specified in the **l_linger** member.	Yes

closesocket()操作限制

- Closesocket把描述符的引用计数减1.引用计数减为0时才会发送FIN消息给对方，这个通知有可能滞后
- closesocket()终止读和写两个方向的数据传送。单方面调用closesocket(),可能会因为TCP实现尚未来的及发送已提交的数据或对方最后发送的数据尚未接收，就释放资源，从而导致数据丢失。且这种数据丢失对发送方而言是不可见的。

连接停止操作的引入

- closesocket()操作的局限可以通过主动的连接停止操作来避免。
- 函数shutdown()和函数WSASendDisconnect()都可以用于启动连接停止序列。
- 让客户知道服务器已读取其数据的一个方法：调用shutdown()(参数设为SD_SEND)开始本方的连接停止序列，然后调用接收操作，等待接收到对等方的FIN后返回

closesocket与shutdown的区别

- close把描述符的引用计数减1，仅在该计数变为0时关闭套接字。shutdown可以不管引用计数就激发TCP的正常连接终止序列
- close终止读和写两个方向的数据发送。TCP是全双工的，有时候需要告知对方已经完成了数据传送，即使对方仍有数据要发送给我们。
- **调close是把文件和协议栈资源一起释放。而调shutdown是先释放协议栈中资源。再调用close就不发送fin了，只关socket文件描述符。**

顺序释放连接过程

- 目的：为了在连接撤销之前保证双方接收到来自对等方的所有数据。
- 思路：客户、服务分别调用一次shutdown()

连接停止与关闭的操作过程示例

客户	服务器
调用shutdown(s,SD_SEND)声明会话结束，客户不再发送数据。	
	获知客户已关闭连接，接收客户已发送的数据
	发送剩余的响应数据
调用recv()函数接收服务器发回的应答数据	调用shutdown(s,SD_SEND)声明服务器不再发送数据
获知对方关闭连接	调用closesocket()函数，释放套接字相关资源
调用closesocket()函数，释放套接字相关资源	

避免TCP传输控制对性能的影响
制约TCP传输性能的原因分析

- 消耗传输时间的因素
  - 每一次发送函数的调用至少需要两个上下文切换
  - Nagle算法会影响传输时间
- Nagle算法与TCP的延迟确认机制一起工作时，在一些特殊场景下(Write-Write-Read)，可能会大大影响数据的传输效率。图5-26

延迟确认机制

- 延迟确认机制的目的是为了减少网络传输的段的数量，当对等方的段到达时，TCP延迟发送ACK消息，希望应用程序对刚刚接收到的数据做出响应，以便ACK消息可以在新数据段中捎带出去。
- 该延迟的取值在不同的操作系统中默认值不同。
  - 在Linux系统中，该值通常为40ms，在BSD实现中，这个延迟的取值通常为200ms，Windows系统也默认使用200ms

避免TCP传输控制对性能的影响
选择合适的发送方式

- 场景一：实时、单向小段数据发送的网络操作
- Nagle算法与TCP的延迟确认机制互相影响而产生的200ms延迟不能满足实时监控的应用需求，因此我们希望消除这种影响带来的响应延迟
- Nagle算法可以通过TCP层次上的套接字选项的设置来禁用
- 在可以用其他方法提高TCP响应时间的情况下，尽量不要禁用Nagle算法

关闭Nagle算法

```
static void _set_tcp_nodelay(int fd) {
    int enable = 1;
    setsockopt(fd, IPPROTO_TCP, TCP_NODELAY, (void*)&enable, sizeof(enable));
}
```

避免TCP传输控制对性能的影响
选择合适的发送方式

- 场景二:“发送-发送-…-接收”型的网络操作
- 思路：将接收操作之前的多次发送操作聚合为一次发送请求
- 可以利用WSASend()函数来简化思路的实现
- WSASend()覆盖标准send()函数，并在下面两个方面有所增强
  - 1)可以用于重叠套接字进行重叠发送操作
  - 2)可以一次发送多个缓冲区中的数据来进行集中写入

```
int WSAAPI WSASend(
  SOCKET                             s,
  LPWSABUF                           lpBuffers,
  DWORD                              dwBufferCount,
  LPDWORD                            lpNumberOfBytesSent,
  DWORD                              dwFlags,
  LPWSAOVERLAPPED                    lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

lpBuffers：一个指向WSABUF结构数组的指针。每个WSABUF结构包含缓冲区的指针和缓冲区的大小。

dwBufferCount：lpBuffers数组中WSABUF结构的数目。

lpNumberOfBytesSent：返回值，如果发送操作立即完成，则为一个指向所发送数据字节数的指针。

dwFlags：标志位

lpOverlapped：指向WSAOVERLAPPED结构的指针（对于非重叠套接口则忽略）。

lpCompletionRoutine：一个指向发送操作完成后调用的完成例程的指针。（对于非重叠套接口则忽略）。

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

```
char buf1[10], buf2[10];
WSABUF lpBuffer[2];
lpBuffer[0].buf = buf1;
lpBuffer[0].len = 10;
lpBuffer[1].buf = buf2;
lpBuffer[1].len = 10;
WSARecv(socket, lpBuffer, 2, &flag, NULL, NULL);
memset(buf1, 0, 10);
memset(buf2, 0, 10);
WSARecv(socket, lpBuffer, 2, &flag, NULL, NULL);
```

设置合适的缓冲区大小

- 较合适的有效缓冲区大小能够降低应用程序执行发送和接收的系统调用次数，从而降低上下文切换开销，较大的缓冲区有助于降低发送流量控制的可能性，并且将提高CPU效率。
- 如果缓冲区太大，并且应用程序处理数据的速度不够快，页面调度操作就会增加
- Windows系统默认缓冲区大小8KB,最大是8MB
- 发送缓冲区和接收缓冲区的大小可以通过TCP选项设置进行修改。

获取缓冲区大小示例

```
int WinSockTool::getRecvBuffOpt(SOCKET fd)
{
    int rcvbuf_len;
    int len = sizeof(rcvbuf_len);
    if (getsockopt(fd,SOL_SOCKET,SO_RCVBUF,(char *)&rcvbuf_len,&len) < 0) {
        printf("getsockopt error\n");
        return -1;
    }
    printf("接收缓冲区大小:%d\n", rcvbuf_len);

    return 0;
}
```

设置缓冲区大小示例

```
int WinSockTool::setSendBuffOpt(SOCKET fd)
{
    int sendbuf_len = 10 * 1024; //10K
    int len = sizeof(sendbuf_len);
    if (setsockopt(fd, SOL_SOCKET, SO_SNDBUF, (char*)&sendbuf_len, len) < 0) {
        printf("setsockopt error\n");
        return -1;
    }
    return 0;
}
```

总结

```
int setsockopt(
  [in] SOCKET      s,
  [in] int         level,
  [in] int         optname,
  [in] const char *optval,
  [in] int         optlen
);
```

详细

本章涉及的套接字选项

level = **SOL_SOCKET**

Value	Type	Description
SO_DONTLINGER	BOOL	Does not block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with **l_onoff** set to zero.
SO_KEEPALIVE	BOOL	Enables sending keep-alive packets for a socket connection. Not supported on ATM sockets (results in an error).
SO_LINGER	LINGER	Lingers on close if unsent data is present.
SO_RCVBUF	int	Specifies the total per-socket buffer space reserved for receives.
SO_SNDBUF	int	Specifies the total per-socket buffer space reserved for sends.

本章涉及的套接字选项

level = **IPPROTO_TCP**

Value	Type	Description
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.This socket option is included for backward compatibility with Windows Sockets 1.1

ioctlsocket

```
int ioctlsocket(
  SOCKET s,    //套接字句柄
  long   cmd,  //套接字命令,详细
  u_long *argp  //指向命令参数的指针
);
```

成功执行返回0，否则返回SOCKET_ERROR
详情

WSAIoctl

```
int WSAAPI WSAIoctl(
  SOCKET                             s,                    //套接字句柄
  DWORD                              dwIoControlCode,      //操作的控制代码
  LPVOID                             lpvInBuffer,          //指向输入参数缓冲区
  DWORD                              cbInBuffer,           //输入参数缓冲区大小
  LPVOID                             lpvOutBuffer,         //指向输出缓冲区
  DWORD                              cbOutBuffer,          //输出缓冲区大小
  LPDWORD                            lpcbBytesReturned,    //输出参数，返回输出实际字节数的地址
  LPWSAOVERLAPPED                    lpOverlapped,         //指向WSAOVERLAPPED的指针
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine   //指向操作结束后调用的例程指针
);
```

最后两个参数在使用重叠I/O时才用

成功执行返回0，否则返回SOCKET_ERROR

详情

本章使用的IO命令

SIO_KEEPALIVE_VALS 详细用法

仅能用WSAIoctl调用，不能用ioctlsocket调用

```
int WSAIoctl(
  (socket) s,              // descriptor identifying a socket
  SIO_KEEPALIVE_VALS,                  // dwIoControlCode
  (LPVOID) lpvInBuffer,    // pointer to tcp_keepalive struct
  (DWORD) cbInBuffer,      // length of input buffer
  NULL,         // output buffer
  0,       // size of output buffer
  (LPDWORD) lpcbBytesReturned,    // number of bytes returned
  (LPWSAOVERLAPPED) lpOverlapped,   // OVERLAPPED structure
  (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine,  // completion rout
);
```

```
/* Argument structure for SIO_KEEPALIVE_VALS */
struct tcp_keepalive {
    u_long  onoff;
    u_long  keepalivetime;
    u_long  keepaliveinterval;
};
```