错误代码：

WSANOTINITIALISED：WSAStartup 没有调用或者没有调用成功.

WSAENETDOWN：网络系统出错

WSAEADDRINUSE：地址已经在用，并且其套接字并没有设置地址重用选项 SO_REUSEADDR。

WSAEINPROGRESS：一个 WINSOCKET1.1 版本的阻塞调用正在处理或者服务提供者正在处理一个回调函数

WSAEINVAL：该套接字已经绑定到一个地址

WSAEISCONN：该套接字已经连接

WSAEMFILE：没有可用的套接字

WSAENOBUFS：没有足够的缓冲区

WSAENOTSOCK：该描述符不是一个套接字

WSAEOPNOTSUPP：该套接字不支持 listen 操作

10. Accept()函数

Accept()接受到来的连接请求。它从等待连接该套接字的请求队列里提取第一个连接，然后创建一个新的套接字，并返回它的指向该套接字的描述符。新创建的套接字拥有与处理该连接的套接字一样的属性，包括异步时间处理特性，但是新创建的套接字不能处理新的连接请求。如果处理连接请求的套接字是阻塞方式的，那么 accept()将使进程一直处于阻塞状态，直到有连接请求到来；如果处理连接请求的套接字是非阻塞方式的，那么 accept()不会使进程阻塞，若有请求到来，则进行处理，否则返回一个错误，进程则照常运行，以后当有连接请求到来时，则创建一个新的套接字，并且发出一个相应的指示，如设置事件对象等。

accept()函数既可处理面向连接（如：SOCK_STREAM）的请求，又可处理面向无连接的请求(如：SOCK_DGRAM)。

函数原型：

```
SOCKET accept (
  SOCKET s,
  struct sockaddr FAR* addr,
  int FAR* addrlen
);
```

表 13-15 参数说明

参 数	说 明
S	已经建立并且处于监听状态的套接字
Addr	获取并填充连接请求方的地址
Addrlen	返回 addr 的长度
输出参数	成功则返回新创建的套接字描述符，否则返回 INVALID_SOCKET，可以使用 WSAGetLastError 函数来判定发生错误的类型

错误代码：

WSANOTINITIALISED：WSAStartup 没有调用或者没有调用成功。

WSAENETDOWN：网络系统出错。

WSAEFAULT：参数 addrlen 太小或 addr 不是用户空间的有效地址

WSAEINTR：由于调用 WSACancelBlockingCall.函数导致（阻塞）Windows Socket 1.1 调用被取消。

WSAEINPROGRESS：一个 WINSOCKET1.1 版本的阻塞调用正在处理或者服务提供者正在处理一个回调函数。

WSAEINVAL：在调用 accept 函数之前并没有调用 listen 函数。

WSAEMFILE：队列中有连接请求存在，但是没有足够的套接字描述符。

WSAENOBUFS：没有足够的缓冲区。

WSAENOTSOCK：该描述符不是一个套接字。

WSAEOPNOTSUPP：引用的套接字不支持面向连接的服务类型。

WSAEWOULDBLOCK：该套接字是非阻塞的，同时当前没有连接请求。

以上都是常用的和 BSD 兼容的套接字函数，还有一些其他的函数如：

select()：提供对多个套接字进行管理，判断其中每个套接字的读、写和异常状态

getsockopt()：取得套接字选项。

setsockopt()：设置套接字选项。

ioctlsocket()：控制套接字 I/O 模式。

getsockname()：取得本地地址。

getpeername()：取得对方地址。

gethostbyname()：按名字取得主机信息。

gethostbyaddr()：按地址取得主机信息。

对于 Windows 操作系统，它本身有一套相应的 SOCKET 操作函数。其中

WSASocket() 对应于 socket()；

WSAAccept() 对应于 accept();
WSASend() 对应于 send();
WSASendTo() 对应于 sendto();
WSARecv() 对应于 recv();
WSARecvFrom() 对应于 recvfrom();
WSAListen() 对应于 listen();
WSABind() 对应于 bind();
WSAConnect() 对应于 connect();
WSACloseSocket() 对应于 closesocket()。

13.5 Socket 的编程框架

通过上面的介绍，我们基本熟悉了 WinSocket 函数，下面就来介绍如何编写 Windows Socket 程序。最通用和流行的的程序就是客户机/服务器模式，服务器是能够提供某种或某些功能的程序或进程；客户机是用户想使用服务器的某种或某些功能的程序或进程。在客户机/服务器模式中，应首先启动服务器进程，然后客户机通过网络（也可以是本机）访问服务器资源，以完成相应的操作。

通常的网络通信有两种方式，这里主要针对传输层而言，即：面向连接方式，如采用 TCP 协议；面向无连接方式，如 UDP 方式。但是有时为了编程方便，在使用 UDP 协议时采用面向连接的处理方式。

另外针对如何处理网络请求网络程序又可分为交互方式和并发方式。交互方式的特点是编程简单，不容易出错，但是执行效率不高；相反，并发方式则相对复杂，不容易控制，但是执行效率较高。对于并发的实现有多种方式，其中主要的两种方式如下：

（1）创建线程为客户机服务，这里也同样有以下几种方式实现：

① 为每一个客户端创建一个线程。

② 服务器创建一定量的线程，形成一个线程池，当有客户端请求时，则分配一个线程为其服务，若没有线程可用，则客户端不得不等待，直到有其他客户端释放连接为止。

③ 服务器并不事先创建线程，但是当有客户连接到来时，则创建之，线程的数量受到服务器预置值的限制。处理客户的请求类似第二种方法。

（2）使用事件驱动方式。在一个线程内可以通过异步 I/O 操作方式，当有事件发生时，就触发相应的过程来处理。

那么我们看看 Windows Sockets 建立网络进程间对话的过程。

13.5.1 面向连接方式

在 TCP/IP 协议栈中，TCP 就是采用面向连接方式来进行通信的。对于 UDP 协议来说，为了减少每次 recvfrom/sendto 参数中指定目的地址的麻烦，也可以采用面向连接的方式。

1. 首先启动服务器方，以提供相应服务：

（1）首先使用 WSAStartup 函数来初始化网络环境。

（2）调用 socket（AF_INET, SOCK_STREAM, 0）函数来创建一个套接字。

（3）调用 bind 函数将本地地址与刚创建的套接字关联起来。

（4）调用 listen 函数监听发向该套接字的连接请求。

（5）客户端的连接请求放在连接请求队列里，服务器调用 accept 函数从连接请求队列中取出第一个请求，创建一个为之服务的新的套接字，该套接字处理所有与该客户交互操作。而服务器进程的监听套接字这时继续处理来自其他客户的连接请求，直到因队列空而等待新的连接请求的到来。

（6）调用 closesocket 关闭监听套接字，释放套接字资源。

（7）调用 WSACleanup 函数释放相应资源。

2. 客户端进程：

（1）首先使用 WSAStartup 函数来初始化网络环境。

（2）调用 socket（AF_INET, SOCK_STREAM, 0）函数来创建一个套接字。

（3）调用 connect 函数连接远程服务器，以请求服务。

（4）服务器响应连接请求后，此时客户进程开始与服务器的交互操作，直到请求结束为止。

（5）调用 closesocket 关闭套接字，释放套接字资源。

（6）调用 WSACleanup 释放相应资源。

下面的图分别显示了服务器和客户端的编程框架。

对于服务器的设计有两种方式，即重复式和并发式。

重复式服务器：

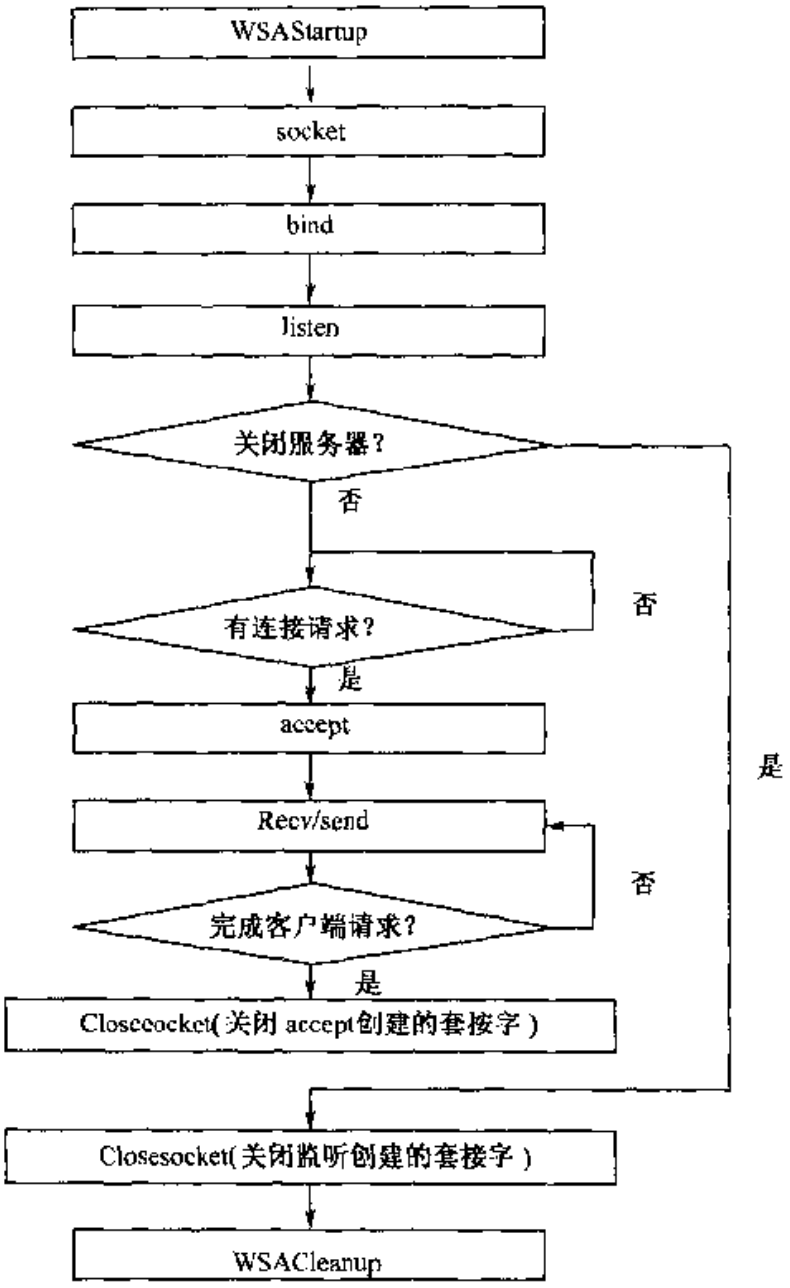

图 13-2　重复式服务器结构

并发式服务器结构：

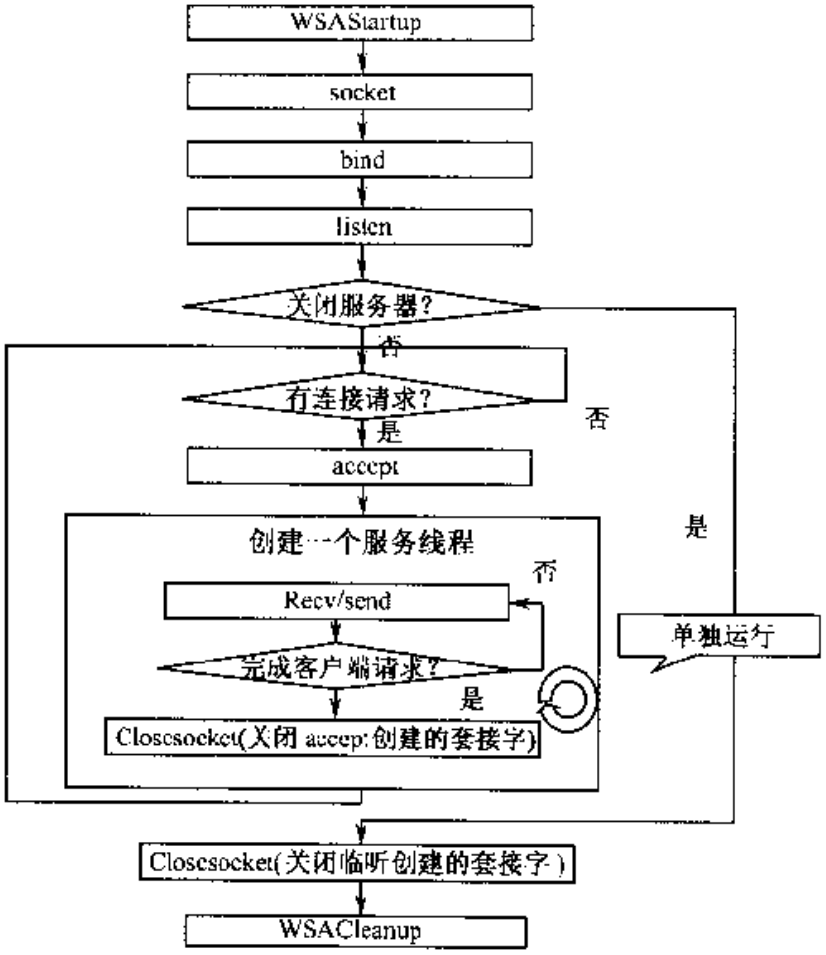

图 13-3 并发式服务器结构

客户机结构：

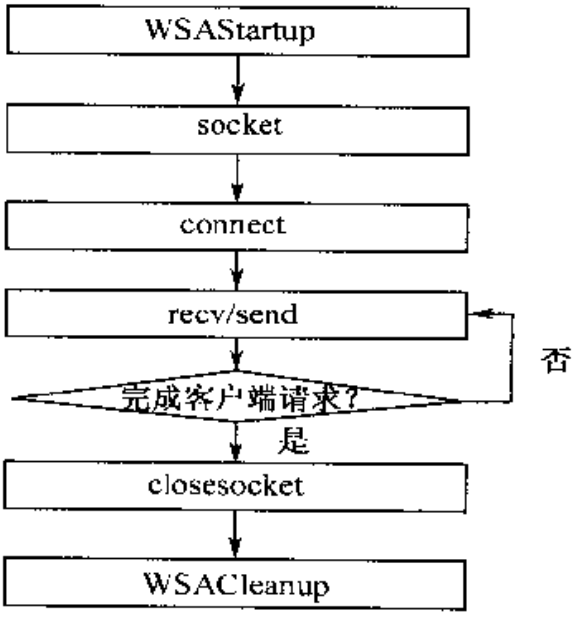

图 13-4 客户机结构

13.5.2 面向无连接方式（如使用 UDP 协议等）

服务器端结构：

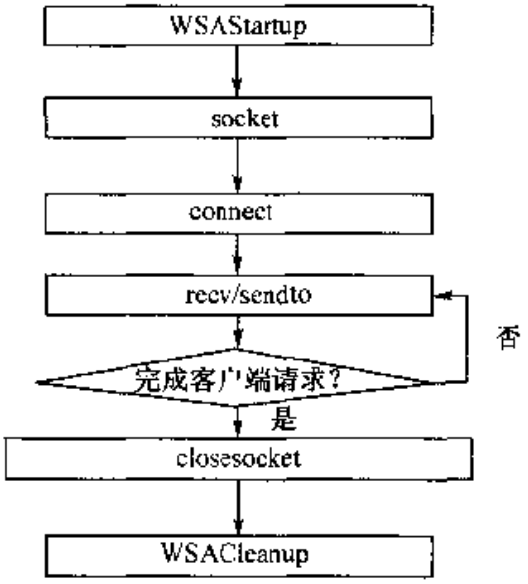

图 13-5 面向无连接的服务器结构

客户机端结构：

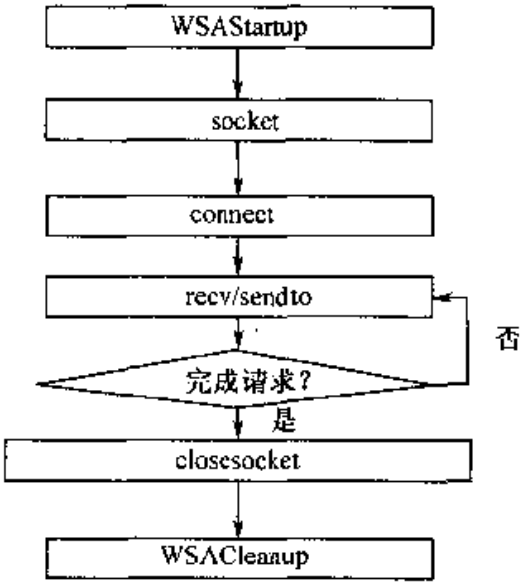

图 13-6 面向无连接的客户机结构

用一个经典的例子(摘自 Douglas E. Comer 与 David L. Stevens 合著的 INTERNETWORKING WITH TCP/IP VOLUME III CLIENT SERVER PROGRAMMING AND APPLICATIONS 一书)来描述客户机/服务器编程模式。本例说明回显服务（ECHO）是如何实现的，服务器采用的是并发模式，即对每

一个客户的请求都用一个独立的线程来处理，这样能极大地提高系统效率，减少连接队列中连接请求的大小。

服务器部分代码如下：

```
#include <winsock2.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>

#define QLEN    5
#define STKSIZE 16536
#define BUFSIZE 4096
#define WSVERS  MAKEWORD(2, 0)

SOCKET msock, ssock;

int TCPEchod(SOCKET);
SOCKET passivesock(const char *service, const char *transport, int
qlen);
SOCKET passiveTCP(const char *, int);
SOCKET passiveUDP(const char * service);

int main( int argc, char *argv[])
{
    char *service = "echo";
    struct sockaddr_in fsin;
    int alen;
    WSADATA wsadata;

    switch(argc)
    {
    case 1:
        break;
    case 2:
```

```
        service = argv[1];
        break;
    default:
        printf("Usage: TcpEchod [port]\n");
        return -1;
    }

    if(WSAStartup(WSVERS, &wsadata) != 0)
    {
        printf("WSAStartup failed!\n");
        return -1;
    }

    msock = passiveTCP(service, QLEN);

    while (true)
    {
        alen = sizeof(fsin);
        ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
        if(ssock == INVALID_SOCKET)
        {
            printf("Accept: error number \n", WSAGetLastError());
            return -1;
        }
        if(_beginthread((void (*) (void *))TCPEchod, STKSIZE, (void
        *) ssock) < 0)
        {
            printf("_beginthread: %s\n" , strerror(errno));
            return -1;
        }

    }

    WSACleanup();
```

```
    return 1;
}

int TCPEchod(SOCKET fd)
{
    char buf[BUFSIZE];
    int cc;

    cc = recv(fd, buf, sizeof(buf), 0);

    while(cc != SOCKET_ERROR && cc > 0 )
    {
        if (send(fd, buf, cc, 0) == SOCKET_ERROR)
        {
            printf("echo send error : %d\n", WSAGetLastError());
            break;
        }
        cc = recv(fd, buf, sizeof(buf), 0);
    }

    if( cc= SOCKET_ERROR)
        printf("echo recv error :%d\n", WSAGetLastError());

    closesocket(fd);

    return 0;
}

SOCKET passivesock(const char *service, const char *transport, int
qlen)
{
 struct servent *pse;
 struct protoent *ppe;
```