# 实 验 六 TCP 多连接管理

## 6.1 实验目的

本实验项目实现了 TCP 多连接的建立与稳妥断开。服务端执行监听客户端主动连接建立连接，数据通信任务完成后，通信过程中配对的 Socket 对象的任一方都可主动发起请求断开连接。

## 6.2 TCP 连接中多个 Socket 对象的管理

将客户端与服务端的线程对象结合 TCP 通信流程模式可由图6-2进行描绘，它展示了 TCP 通信时服务端两线程的 Socket 对象工作原理及联系。

网络通信的驱动程序在 TCP 通信过程中执行特别的任务，网络出现的连接请求数据包经驱动程序识别后，创建用于数据通信的 Socket 对象。驱动程序以链表的形式管理 TCP 服务端的多个通信的 Socket 对象，用户程序通过 Accept 方法获得这个 Socket 对象。用户程序调用 Socket 对象的 Close 方法后，资源句柄值仍存在，但对应的资源已被驱动程序回收，这时 Socket 对象不为 null 但是对象成员变为不可用。

处于连接状态的 Socket 对象具有 LocalEndPoint 与 RemoteEndPoint 属性，客户端与服务端进行通信的两个 Socket 对象形成配对。图6-3指出 Socket 对象间的 IP 与 PORT 分配关系，在服务端所有已连接的 Socket 对象其本地 Port 与 IP 都是一样的，且 Port 值就是监听端口值。同个主机可向相同的服务器端口发起多个连接，客户端 Socket 对象本地 IP 值都相同端口值必须不相同。连到同个服务器的不同主机客户端 Socket 对象的本地端口值随机出现。

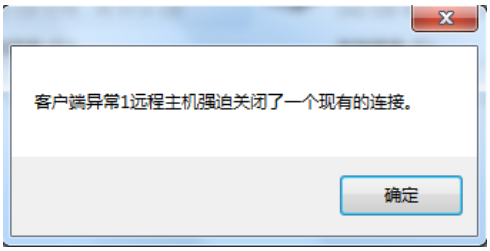


图 6-1 TCP 的非正常断开

## 6.3 TCP 连接的稳妥断开

TCP 实现的是数据双向通信，通信两端调用 Socket 对象的 Shutdown 方法分别指定传输断开。TCP 通信中配对的 Socket 对象实现稳妥断开，应按照表6-1列出的操作序列执行 Shutdown

方法。配对的 Socket 对象没有执行 ShutDown 方法，直接执行 Close 方法将引起程序异常，如图??。

表 6-1 TCP 连接的稳妥断开

| 客户端 | 服务端 |
|---|---|
| 调用 ShutDown 方法，客户端表示不再发送数据 | |
| | 服务端收到 FD_CLOSE 消息，开始断开连接，无后续可接收数据 |
| 客户端此时仍可继续接收数据 | 服务端继续向客户端发送后续数据。 |
| | 服务端调用 ShutDown 方法，表示不再发送数据。 |
| 客户端收到 FD_CLOSE 消息 | |
| 客户端调用 Close 方法清除资源 | 服务端调用 Close 方法清除资源 |

通信的断开由任一方主动发起，另一方收到断开请求后仍可继续发送数据，双方都无数据发送且都执行了 Shutdown 方法后连接稳妥断开。Socket 对象具有 Connected 属性指示连接状态，false 表示非连接状态。Connected 属性是只读的，无法通过对其赋值控制 Socket 对象的连接状态，尝试对其赋值将引发异常。

## 6.4 实验内容

本小节包含项目 TcpManC 实现客户端任务，TcpManS 实现服务端任务，互相配合实现十个 TCP 连接演示连接建立与稳妥断开操作过程。因为线程对象运行时消耗资源较多，客户端与服务端由一个工作线程用数组管理通信的十个 Socket 对象。

网络函数任务返回时间是不确定的，程序以异步回调的方式将更加灵活，TCP 通信中常用的函数有 Accept、Connect、Receive、Send 等，以 Receive 函数说明采用异步方式实现网络任务的几个关键步骤：

1. 自定义数据结构，封装需要的变量及内存空间，本实验数据结构名称为 StateObject。
2. 编写回调的任务处理函数，本实验中命名为 ReceiveCallback；
3. ReceiveCallback 函数中必须调用 EndReceive 方法完成一次回调任务周期；
4. 线程调用异步函数 BeginReceive，并按规定传入变量；

接收数据的回调任务处理函数 ReceiveCallback 中必须调用 EndReceive 方法完成一次成功的数据接收，EndReceive 方法的返回整型数值表示接收到的应用数据的字节数。当一方执行 Shutdown 方法后驱动程序发送不包含用户数据的包，包的头部含有 FD_CLOSE 消息的标志位，回调函数 ReceiveCallback 仍被调用，这时 EndReceive 方法的返回值为 0。一般用户数据长度是任意的，回调函数 ReceiveCallback 每次接收的数据长度不能超过最大值，函数 ReceiveCallback 应根据情况继续调用 BeginReceive 函数完成后续数据的接收。实现代码在后面的小节提供。

函数 ReceiveCallback 的参数必须是 IAsyncResult 接口类型，这个函数将由驱动程序在数据到达时调用，并传入封装好的 IAsyncResult 类型对象，用户自定义数据结构对象 StateObject 作为 BeginReceive 函数的参数被传入，代码 (StateObject)ar.AsyncState 获取这个结构对象。

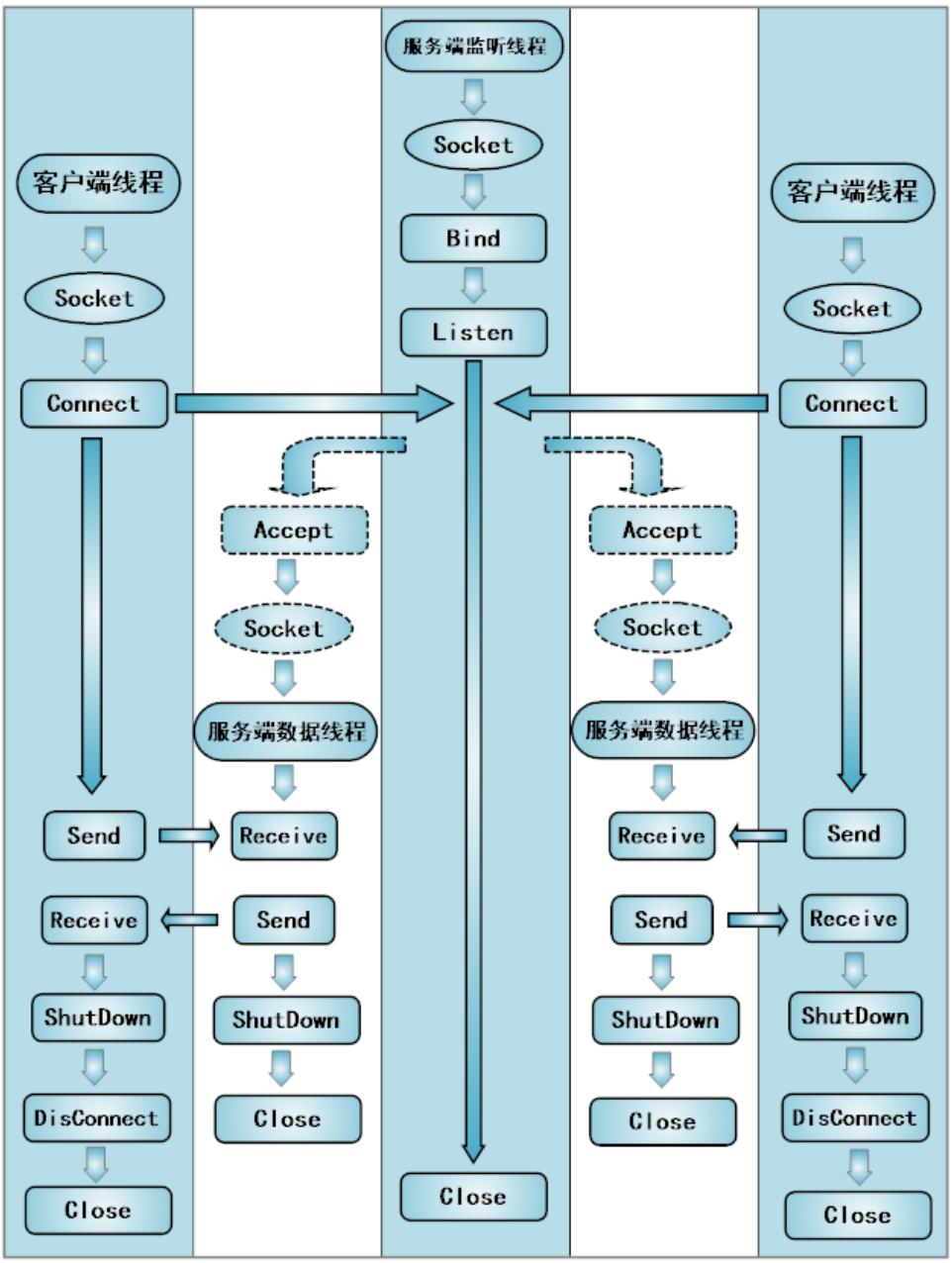


图 6-2 TCP 两线程连接模型

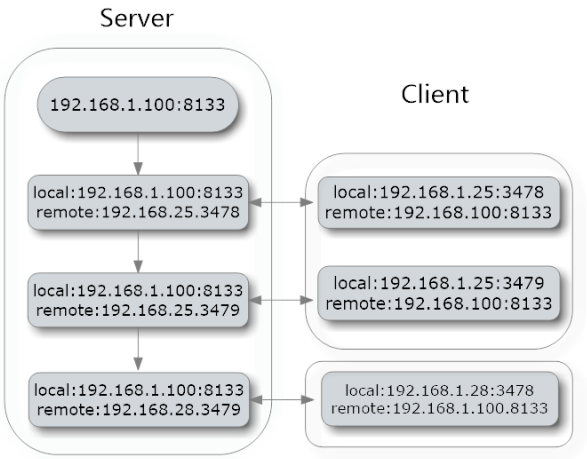


图 6-3 TCP 多连接中 IP 与 Port 匹配特点

StateObject 数据结构的典型定义请参考客户端与服务端代码。

Socket 对象 BeginReceive 函数的原型声明如下：

```
public IAsyncResult BeginReceive(
    byte[] buffer,//接收数据的缓冲区
    int offset, //缓冲区偏移量
    int size, //缓冲区长度
    SocketFlags socketFlags,//Socket 标志
    AsyncCallback callback,//回调函数，ReceiveCallback
    Object state //状态参数 StateObject-> ar.AsyncState
)
```

其中 buffer 变量值即是 StateObject 的内存空间，回调函数值是 ReceiveCallback，而 state 变量值则是 StateObject 对象。

本实验中客户端通过 Connect 方法主动向服务端发起连接，执行简单的文本传输任务，客户端采用 Shutdown 方法主动发起断开请求，服务端的 Receive 方法判断接收数据的长度，长度为 0 表明客户执行了 Shutdown 方法，服务端也执行 Shutdown 方法，从而实现 TCP 连接的稳妥断开。为了标识每个 Socket 对象及其相应的文本信息，通过对 Socket 对象进行编号，保证了数据通信过程的一致性，某个 TCP 连接的断开不会影响其它连接的通信过程。

图6-4是参考的客户端运行界面，图6-5是参考的服务端运行界面。

### 6.4.1 客户端程序

客户端程序用到的全局变量声明：

```
public static string[] Infos;
public static IntPtr mainWndHandle;
public static IPEndPoint remoteEP;
```


图 6-4 TCP 的多连接客户端界面


图 6-5 TCP 的多连接服务端界面

```
public static StateObject[] ClientState;
public const int UPDATE_INFO = 0x502;
public const int UPDATE_SOCKINFO = 0x503;
public static int socketid;
public static int selectedIndex;
public static Socket[] ClientSock;
```

自定义消息发送 API 声明：

```
[DllImport("User32.dll", EntryPoint = "SendMessage")]
private static extern int SendMessage(
    IntPtr hWnd,
    int Msg,
    int wParam,
    int lParam);
```

回调函数使用的传递多个变量的自定义数据结构 StateObject，它在客户端与服务端程序中是很接近的。

```
// State object of Client Socket for receiving data from remote device.
public class StateObject
{
    //Socket 对象编号，对应某个 TCP 连接
    public int clientNum;
```

```
    // Client socket.
    public Socket workSocket = null;
    // Size of receive buffer.
    public const int BufferSize = 1024;
    // Send buffer.
    public byte[] buffer = new byte[BufferSize];
    //Data length
    public int Datalen;
    // Received data string.
    public StringBuilder sb = new StringBuilder();
}
```

FrmMainC 窗体的 Load 事件中对全局变量进行赋值：

```
private void FrmMainC_Load(object sender, EventArgs e)
{
    mainWndHandle = this.Handle;
    Infos=new string[11];
    ClientSock=new Socket[10];
    ClientState=new StateObject[10];
    for (int i = 0; i < 10;i++ )
    {
        ClientSock[i]=new Socket(AddressFamily.InterNetwork,
            SocketType.Stream,ProtocolType.Tcp);
        ClientState[i]=new StateObject();
        ClientState[i].buffer=new byte[1024];
        ClientState[i].clientNum = i;
        ClientState[i].workSocket = ClientSock[i];
        Infos[i] = string.Format(" 第 {0} 个 Socket 对象创建成功，未连接", i);
    }
    selectedIndex = -1;
    remoteEP = new IPEndPoint(IPAddress.Parse("172.16.201.85"), Int32.Parse("8133"));
    ReFreshInfo();
}
```

刷新显示字符串信息的 ReFreshInfo 函数：

```
public void ReFreshInfo()
{
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 11;i++ )
    {
        sb.AppendLine(Infos[i]);
    }
```

```
  textBox2.Text = sb.ToString();
}
```

重载窗体的消息处理函数 DefWndProc，处理线程发来的自定义消息：

```
protected override void DefWndProc(ref Message m)
{
  switch (m.Msg)
  {
    //显示信息更新
    case UPDATE_INFO:
      ReFreshInfo();
      break;
    //Socket 信息更新
    case UPDATE_SOCKINFO:
      listBox1.Items[socketid]=ClientSock[socketid].LocalEndPoint.ToString();
      ReFreshInfo();
      break;
    default:
      base.DefWndProc(ref m);
      break;
  }
}
```

列表框的点击用于选择特定的 Socket 对象，其事件代码如下：

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  selectedIndex=listBox1.SelectedIndex;
}
```

发起 TCP 连接按钮的事件代码：

```
private void button1_Click(object sender, EventArgs e)
{
  if(selectedIndex!=-1)
  {
    //检测是否已经连接
    if (!ClientSock[selectedIndex].Connected)
    {
    ClientSock[selectedIndex].BeginConnect(remoteEP,
      new AsyncCallback(ConnectCallback), ClientState[selectedIndex]);
    }
  }
}
    //发起连接的回调函数
```

```
private static void ConnectCallback(IAsyncResult ar)
{
  try
  {
    // Retrieve the socket from the state object.
    StateObject cltso = (StateObject)ar.AsyncState;
    // Complete the connection.
    cltso.workSocket.EndConnect(ar);
    Infos[cltso.clientNum]=string.Format("Socket connected to {0}",
      cltso.clientNum.ToString());
    socketid=cltso.clientNum;
    SendMessage(mainWndHandle, UPDATE_SOCKINFO, 100, 100);
    cltso.workSocket.BeginReceive(cltso.buffer,0,StateObject.BufferSize,0,
      new AsyncCallback(ReceiveCallback), cltso);
  }
  catch (Exception e)
  {
    MessageBox.Show(e.ToString());
  }
}
```

实现多次数据接收的回调函数 ReceiveCallback：

```
public static void ReceiveCallback(IAsyncResult ar)
{
  StateObject state = (StateObject)ar.AsyncState;
  Socket handler=state.workSocket;
  SocketError sckErr;
  try
  {
    int bytesRead = handler.EndReceive(ar, out sckErr);
    if (bytesRead > 0)
    {
      //接收到应用数据，应根据逻辑继续调用 BeginReceive，待完善。
    }
    else {//bytesRead==0
      Infos[state.clientNum]=string.Format(" 服务端已经断开连接 {0}",state.clientNum);
      state.workSocket.Close();
      SendMessage(mainWndHandle,UPDATE_INFO,100,100);
    }
  }catch(Exception sockError)
  {
```

```
        MessageBox.Show(sockError.Message);
    }
}
```

向服务端发送文本信息的按钮事件代码：

```
private void button11_Click(object sender, EventArgs e)
{
    //发送数据
    if (selectedIndex != -1)
    {
        //检测是否已经连接
        if (ClientSock[selectedIndex].Connected)
        {
            // Convert the string data to byte data using UTF8 encoding.
            byte[] byteData = Encoding.UTF8.GetBytes(textBox1.Text);
            byte[] bdataLen = BitConverter.GetBytes(byteData.Length);
            ClientState[selectedIndex].Datalen=byteData.Length;
            Array.Copy(bdataLen, ClientState[selectedIndex].buffer, 4);
            Array.Copy(byteData, 0, ClientState[selectedIndex].buffer, 4, byteData.Length);
            //Clear bytes of StringBuilder
            ClientState[selectedIndex].sb.Remove(0, ClientState[selectedIndex].sb.Length);
            ClientState[selectedIndex].sb.Append(textBox1.Text);
            // Begin sending the data to the remote device.
            ClientState[selectedIndex].workSocket.BeginSend(ClientState[selectedIndex].buffer, 0, byte-
Data.Length + 4, 0,
                new AsyncCallback(SendDataCallback), ClientState[selectedIndex]);
        }
    }
}
```

发送数据的回调函数 SendDataCallback：

```
private static void SendDataCallback(IAsyncResult ar)
{
    try
    {
        // Retrieve the StateObject from the state object.
        StateObject cltso = (StateObject)ar.AsyncState;
        // Complete sending the data to the remote device.
        int bytesSent = cltso.workSocket.EndSend(ar);
        Infos[cltso.clientNum] = string.Format(" 发送：{0}", cltso.sb.ToString());
        SendMessage(mainWndHandle, UPDATE_INFO, 100, 100);
    }
```

```
    catch (Exception e)
    {}
}
```

发起断开指定 TCP 连接的按钮事件代码：

```
private void button2_Click(object sender, EventArgs e)
{
    //发送数据
    if (selectedIndex != -1)
    {
        //检测是否已经连接
        if (ClientSock[selectedIndex].Connected)
        {
            ClientSock[selectedIndex].Shutdown(SocketShutdown.Send);
        }
    }
}
```

### 6.4.2 服务端程序

服务端程序总体上要比客户端多一个执行监听的线程，多个连接的管理与客户端非常接近。

```
    //Socket 已连接数
public static int indexOfConnectedSockets;
//主窗体句柄
public static IntPtr mainWndhandle;
//定义消息常数
public const int UPDATE_INFO = 0x502;
public const int BEGIN_LISTEN = 0x503;
public const int END_LISTEN = 0x504;
public const int WINDOW_RESTORE = 0x505;
//信息字符串数组
public static string[] Infos;
public static Socket listenSocket;
public static IPEndPoint localEndPoint;
//终止监听事件控制变量
public static ManualResetEvent mrEventTermListen;
```

工作线程发送自定义消息的 API 声明：

```
[DllImport("User32.dll", EntryPoint = "SendMessage")]
private static extern int SendMessage(
IntPtr hWnd, // handle to destination window
```

```
int Msg, // message
int wParam, // first message parameter
int lParam // second message parameter
);
```

回调函数使用的传递多个变量的自定义数据结构 StateObject，客户端与服务端程序中的定义是很接近的。 public static StateObject[] ClientStateData;

```
public class StateObject
{
  public int clientNum;
  public int datalen;
  //与客户端通信的 Socket 对象
  public Socket workSocket = null;
  public const int BufferSize = 1024;
  //与客户端通信的缓冲区
  public byte[] buffer = new byte[BufferSize];
  public StringBuilder sb = new StringBuilder();
}
```

窗体的 Load 事件中对变量的初始化：

```
private void FrmMainS_Load(object sender, EventArgs e)
{
  mainWndhandle = this.Handle;
  Infos = new string[11];

  mrEventTermListen = new ManualResetEvent(false);
  localEndPoint = new IPEndPoint(IPAddress.Parse("172.16.201.85"), 8133);
  listenSocket = new Socket(localEndPoint.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
  //创建对象数组仅为 10 个空指针
  ClientStateData = new StateObject[10];
  //实例化 StateObject 对象
  for (int i = 0; i < 10; i++)
  {
    ClientStateData[i] = new StateObject();
    //分配数据缓冲内存空间
    ClientStateData[i].buffer = new byte[1024];
    //初始时数据长度设为 0
    ClientStateData[i].datalen = 0;
  }
  indexOfConnectedSockets = -1;
}
```

TCP 端口监听的线程代码：

```
static void threadListen()
{
    LingerOption _lingerOption = new LingerOption(true, 3);
    listenSocket.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.Linger, _lingerOption);
    listenSocket.Blocking = false;//设定其为异步
    listenSocket.Bind(localEndPoint);
    listenSocket.Listen(10);
    listenSocket.BeginAccept(
        new AsyncCallback(AcceptCallback),
        listenSocket);
    Infos[0] = " 正在监听";
    SendMessage(mainWndhandle, UPDATE_INFO, 100, 200);
    //在用户未设置终止信号前保持为监听状态
    mrEventTermListen.WaitOne();
    Infos[0] = " 监听结束";
    SendMessage(mainWndhandle, UPDATE_INFO, 100, 200);
    listenSocket.Close();
}
```

负责响应客户连接请求建立新连接的回调函数：

```
public static void AcceptCallback(IAsyncResult ar)
{
    Socket dataHandle = listenSocket.EndAccept(ar);
    Interlocked.Increment(ref indexOfConnectedSockets);
    StateObject so = ClientStateData[indexOfConnectedSockets];
    //新客户连接, 使用一个 StateObject 对象, 每次设置一个编号
    so.clientNum = indexOfConnectedSockets;
    so.workSocket = dataHandle;
    Infos[indexOfConnectedSockets] = string.Format(" 第 {0} 个客户连接到服务端，本地 Sock 信息为 {1}",
      indexOfConnectedSockets, dataHandle.LocalEndPoint.ToString());
    SendMessage(mainWndhandle, UPDATE_INFO, 100, 200);
    so.workSocket.BeginReceive(so.buffer, 0, StateObject.BufferSize, 0,
    new AsyncCallback(ReceiveCallback), so);
    //持续接收后续的客户连接
    listenSocket.BeginAccept(
      new AsyncCallback(AcceptCallback),
      listenSocket);
}
```

实现数据多次接收的回调函数 ReceiveCallback：

```
public static void ReceiveCallback(IAsyncResult ar)
{
  //数据到来
  // Retrieve the state object and the handler socket
  // from the asynchronous state object.
  StateObject state = (StateObject)ar.AsyncState;
  Socket handler = state.workSocket;
  SocketError sckErr;
  // Read data from the client socket. Accquire the SocketError
  try
  {
    int bytesRead = handler.EndReceive(ar, out sckErr);
    if (bytesRead > 0)
    {
      //获取数据总长度
      state.datalen = BitConverter.ToInt32(state.buffer, 0);
      //Clear the buffer
      state.sb.Remove(0, state.sb.Length);
      Infos[state.clientNum] = handler.RemoteEndPoint.ToString() + ":"+
        Encoding.UTF8.GetString(state.buffer, 4, state.datalen);
      SendMessage(mainWndhandle, UPDATE_INFO, 100, 200);
      //继续接收后续的网络数据
      handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
        new AsyncCallback(ReceiveCallback), state);
    }
    else
    {//数据长度为 0，FD_CLOSE 消息，.net 平台无法接收。
      if(handler.Connected)
      {
        Infos[state.clientNum] = string.Format("{0} 已发送 ShutDown 请求，已关闭与 {0}
的 TCP 连接",
          handler.RemoteEndPoint.ToString());
        state.workSocket.Shutdown(SocketShutdown.Send);
        //Close 方法将释放资源，但句柄标识仍在，指针值不为空，Connected 属性为 false
        state.workSocket.Close();
        SendMessage(mainWndhandle, UPDATE_INFO, 100, 200);
      }
    }
  }
```

```
    catch (Exception sockError)
    {
        MessageBox.Show(sockError.Message);
    }
}
```

向客户端发送数据的回调函数：

```
private static void SendCallback(IAsyncResult ar)
{
    try
    {
        // Retrieve the socket from the state object.
        Socket handler = (Socket)ar.AsyncState;
        // Complete sending the data to the remote device.
        int bytesSent = handler.EndSend(ar);
    }
    catch (Exception sockError)
    {
        MessageBox.Show(sockError.Message);
    }
}
```

服务端启动监听的按钮事件：

```
private void button1_Click(object sender, EventArgs e)
{
    button1.Enabled = false;
    button2.Enabled = true;
    //启动监听的线程
    ThreadStart workStart = new ThreadStart(threadListen);
    Thread workThread = new Thread(workStart);
    workThread.IsBackground = true;
    workThread.Start();
}
```

刷新显示字符串信息的 ReFreshInfo 函数：

```
public void ReFreshInfo()
{
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 11; i++)
    {
        sb.AppendLine(Infos[i]);
    }
    textBox1.Text = sb.ToString();
```

```
}
```

重载窗体的消息处理函数 DefWndProc，用于处理线程发来的自定义消息。

```
protected override void DefWndProc(ref System.Windows.Forms.Message m)
{
  switch (m.Msg)
  {
    case UPDATE_INFO:
      ReFreshInfo();
      break;
    default:
      base.DefWndProc(ref m);
      break;
  }
}
```

终止监听线程：

```
private void button2_Click(object sender, EventArgs e)
{
  button1.Enabled = true;
  //结束监听
  mrEventTermListen.Set();
}
```

## 6.5 实验作业

1. 调试并完善程序代码，完成本实验中的程序项目。
2. 添加服务端向客户端发送文本的功能代码。