火龙果•整理
uml.org.cn

第六章

C#套接字编程

主要内容

- .NET 网络编程基本操作
- 同步通信开发
  - 同步服务器开发
  - 同步客户机开发
- 异步通信开发
  - 异步服务器开发
  - 异步客户机开发
- TcpListener/TcpClient 通信开发
- UDP通信

1.概述

- .NET 环境下的网络编程仍然是基于套接字的编程。
- 套接字编程接口使用了 System.Net.Sockets 名字空间。其提供的Socket类为Win32下的套接字的托管版本。
- .NET 网络编程提供了很多支持：
  - Socket；
  - TcpClient/UdpClient/TcpListener；
  - WebRequest/WebResponse；
  - HTTP/FTP/File+ WebRequest/WebResponse等。

火龙果•整理
uml.org.cn

.NET与Win32套接字对比

.Net	Win32
Accept/BeginAccept/EndAccept	**accept**
Bind	**bind**
Close	**closesock**
Listen	**listen**
Select	**select**
Connect/BeginConnect/EndConnect	**connect**
GetSocketOption/SetSocketOption	**getsockopt/setsockopt**
Receive/BeginReceive/EndReceive	**recv**
Send/BeginSend/EndSend	**send**
ReceiveFrom/BeginReceiveFrom/EndReceiveFrom	**recvfrom**
SendTo/BeginSendTo/EndSendTo	**sendto**
Shutdown	**shutdown**

.NET网络中常用的类

- **Socket**：实现 Berkeley 套接字接口；
- **Dns**：提供简单的域名解析功能；
- **IPAddress**：提供网际协议 (IP) 地址；
- **IPEndPoint**：将网络端点表示为 IP 地址和端口号；
- **IPHostEntry**：为 Internet 主机地址信息提供容器类;
- **NetworkStream**：提供用于网络访问的基础数据流。
- **TcpLinster**：从 TCP 网络客户端侦听连接。
- **TcpClient**：为 TCP 网络服务提供客户端连接
- **UdpClient**：提供用户数据报 (UDP) 网络服务

火龙果•整理
uml.org.cn

1.1 .NET套接字概述

- 名字空间：System.Net.Sockets.Socket，System.Net 命名空间中的所有其他网络访问类都建立在该套接字实现的基础之上。
- Socket 类支持两种基本模式：
  - 同步：对执行网络操作的函数（如 Send 和 Receive）的调用一直等到操作完成后才将控制返回给调用程序。
  - 异步：网络操作的函数调用立即返回。

1.2 .NET套接字构造

- Socket 的构造函数原型：

```
public Socket (
        AddressFamily addressFamily,
        SocketType socketType,
        ProtocolType protocolType
)
```

  - **参数1：指定 Socket 类用来解析网络地址的标准址族**（如：AddressFamily.InterNetwork 成员指定 IP 版本 4 地址族）。
  - **参数2：指定套接字的类型**（如：SocketType.Stream 成员表示一个用于发送和接收数据并且支持流控制的标准套接字）。
  - **参数3：指定在 Socket 上通信时使用的网络协议**（如：ProtocolType.Tcp 指示套接字使用 TCP）。

1.2.1 TCP/UDP套接字构造

- 构造TCP套接字

```
Socket s = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp);
```

- 构造UDP套接字

```
Socket s = new Socket(AddressFamily.InterNetwork,
    SocketType.Dgram, ProtocolType.Udp);
```

套接字的常用属性

- Socket 常用属性：
  - **Blocking**：设置或者获取套接字阻塞模式；
  - **Connected**：获取当前套接字是否与远程主机相连；
  - **LocalEndPoint**：得到套接字的本地 EndPoint 对象；
  - **RemoteEndPoint**：得到套接字的远程 EndPoint 对象；
  - **SocketType**：套接字类型。

套接字的常用方法

- Socket 常用同步的方法：
  - **Accept；**
  - Bind；
  - Close；
  - **Connect；**
  - **Disconnect；**
  - Listen；
  - **Receive/ReciveFrom；**
  - Select；
  - **Send/SendTo/SendFile；**
  - SetSocketOption/GetSocketOption；
  - Shutdown；

  说明：红色的方法有对应的异步方法。

火龙果•整理
uml.org.cn

1.2.2 使用套接字

- 创建套接字后：
  - **在客户端**，通过Connect方法连接到指定的服务器，并通过Send/SendTo或者Receive/ReceiveFrom从服务端接收数据；
  - **在服务器端**，使用Bind方法绑定所指定的接口使Socket与一个本地终结点相联，并通过Listen方法侦听该接口上的请求，当侦听到用户端的连接时，调用Accept完成连接的操作，创建新的Socket以处理传入的连接请求。
  - **使用完 Socket 后**，使用 Shutdown 方法结束通信，并使用 Close 方法关闭 Socket。
- **上面的是同步方式的通信，还有相对应的异步的Begin/End连接与通信方式。**

1.2.3 Bind方法

- 套接字服务器的绑定端点需要使用到 Socket.Bind() 方法。
- 在调用 Bind 之前，必先创建本地 IPEndPoint。可以用 IPAddress.Any 作为地址参数创建一个 IPEndPoint，这样，将会使用所有可能的网络地址。
- 通过调用 LocalEndPoint 获知所分配的本地网络地址和端口号。

1.2.4 Listen 方法

- Socket.Listen() 将套接字置于监听状态。

```
public void Listen (
        int backlog
)
```

  – backlog 挂起连接队列的最大长度。

1.2.5 SetSocketOption

- 设置或者取消套接字的对应的属性。

```
public void SetSocketOption (
        SocketOptionLevel optionLevel,
        SocketOptionName optionName,
        bool optionValue
)
```

  - 参数1：可以是Socket, IP, TCP, UDP;
  - 参数2：各种与参数1对应的属性，如：Broadcast, RecevieBuffer, ReceiveTimeout等待。
  - 参数3：true/false。true－使能；false－取消。

火龙果•整理
uml.org.cn

1.3 地址/端口信息

- TCP/IP 使用一个网络地址和一个服务端口号来唯一标识设备。
  - 网络地址标识网络上的特定设备；
  - 端口号标识要连接到的该设备上的特定服务。
- 网络地址和服务端口的组合称为网络端点，它在 .NET中由 EndPoint 类表示。对于 IP 地址族，该类为 **IPEndPoint**。

1.3.1 IPEndPoint 使用

- 构造函数两种：

```
public IPEndPoint (
        IPAddress address,
        int port
)
```

```
public IPEndPoint (
        long address,
        int port
)
```

- 其IPEndPoint的使用

```
IPAddress myIP = IPAddress.Parse("127.0.0.1");
IPEndPoint myServer = new IPEndPoint (myIP, myPort);
```

- **问题：在 Win32 中记录IP地址和端口信息是使用的什么？**

火龙果•整理
uml.org.cn

1.3.2 地址处理

- IPAddress 在 System.Net名字空间中。
- 它包含计算机在 IP 网络上的地址相关操作和属性。
- 根据字符串获取IPAddress信息：
  - **Parse ();**
- 提供的常用地址：
  - **Any;**
  - **Broadcast;**
  - **Loopback;**
  - **None;**

火龙果•整理
uml.org.cn

1.3.3 主机解析

- Dns 类是一个静态类，它从域名系统中检索关于特定主机的信息。
- IPHostEntry 类将主机名、别名和一组匹配的 IP 地址关联。常和Dns类一起使用。
- 在 IPHostEntry 类的实例中返回来自 DNS 查询的主机信息。如果指定的主机在 DNS 数据库中有多个入口，则 IPHostEntry 包含多个 IP 地址和别名。
- **问题：在 Win32 中记录地址和机器名信息是使用的什么？**

1.3.4 主机解析用法

- Dns.Resove用于解析主机名到 IPHostEntry。

```
public static IPHostEntry Resolve (
        string hostName
)
```

- 使用：

```
IPHostEntry host = Dns.Resolve ("fun");
Console.WriteLine (host.AddressList[0].ToString ());
```

1.3.5 Dns类常用方法

- GetHostName()可以获得主机名。

```
public static string GetHostName();
string hostname = Dns.GetHostName();
```

- GetHostByName() 方法根据主机名向DNS服务器查询主机信息。

```
public static IPHostEntry GetHostByName (
    string name;
)
```

- GetHostByAddress() 方法根据地址信息查询主机信息。

```
public static IPHostEntry GetHostByAddress (
    string ip;
)
```

```
public static IPHostEntry GetHostByAddress (
    IPAddress ip;
)
```

1.3.6 示例1——地址信息

```
try {
    IPHostEntry hostInfo = Dns.GetHostByName(hostName);
    IPAddress[] address = hostInfo.AddressList;
    String[] alias = hostInfo.Aliases;

    Console.WriteLine("Host name : " + hostInfo.HostName);
    Console.WriteLine("\nAliases : ");
    for(int index=0; index < alias.Length; index++) {
      Console.WriteLine(alias[index]);
    }
    Console.WriteLine("\nIP address list : ");
    for(int index=0; index < address.Length; index++) {
       Console.WriteLine(address[index]);
    }
 }
 catch(SocketException e) {
    Console.WriteLine("SocketException caught!!!");
    Console.WriteLine("Source : " + e.Source);
    Console.WriteLine("Message : " + e.Message);
 }
 catch(Exception e)  {
     Console.WriteLine("Exception caught!!!");
     Console.WriteLine("Source : " + e.Source);
     Console.WriteLine("Message : " + e.Message);
 }
```

1.3.7 示例2——构造服务器

```
IPHostEntry local = Dns.GetHostByName(Dns.GetHostName());
IPEndPoint iep = new IPEndPoint(local.AddressList[0],
                                8000);
Socket newserver = new Socket(AddressFamily.InterNetwork,
                              SocketType.Stream,
                              ProtocolType.Tcp);
newserver.Bind(iep);
newserver.Listen(5);
```

- **Win32如何创建并绑定一个服务器套接字的？**

1.4 出错/异常信息

- SocketException 用来抛出Socket相关异常信息。
- 当网络发生错误时，Socket 和 Dns 引发 SocketException。
- 使用 SocketException.ErrorCode 可以获取特定的错误代码, Message 可以获得错误消息。

```
try {
    // ...
}
catch (SocketException e) {
    Console.WriteLine("{0} Error code: {1}.",
                      e.Message,
                      e.ErrorCode);
    return (e.ErrorCode);
}
return 0;
```

火龙果•整理
uml.org.cn

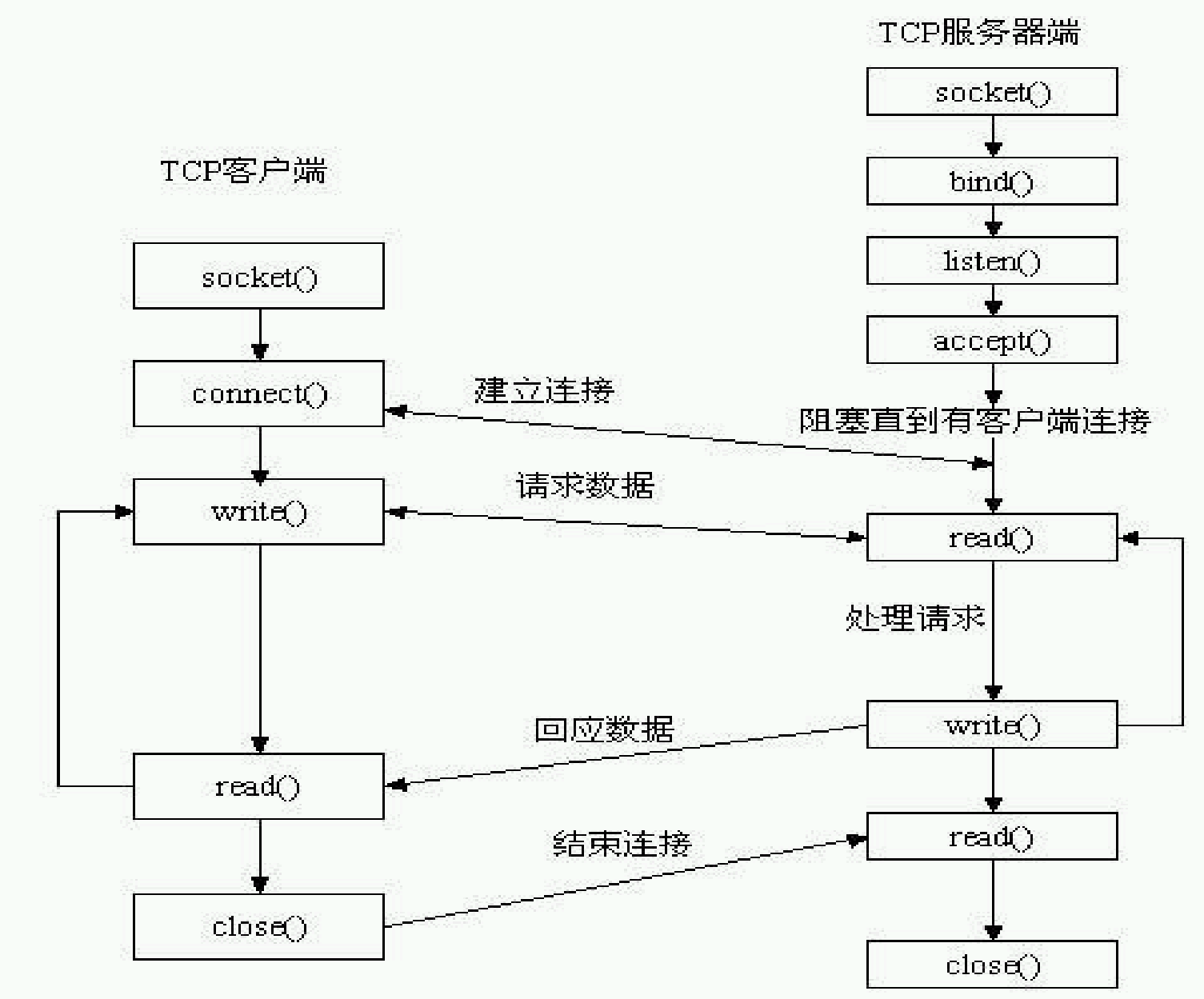
2. 同步服务器/客户机

- 讨论一种常见网络服务器模式：
  - 同步阻塞服务器；
  - 同步阻塞客户机；
- 和Win32下的阻塞模型类似，对每一个接受的请求产生新的线程，并在此线程中完成所有的与客户机的数据交互。

2.1 同步阻塞服务器开发

- 这是使用起来最简单的一种模型。
- 任何时候调用一个绑定I/O的.NET套接字方法，如果套接字上没有等候处理的数据，该调用就会阻塞。
- 如果需要做其他的事情，或者为其它的连接请求提供服务，就需要在应用程序中创建额外的线程。
- 和Win32下的阻塞模型类似。

同步阻塞模型


TCP服务器端
socket()
bind()
listen()
accept()
TCP客户端
socket()
connect()
建立连接
阻塞直到有客户端连接
请求数据
write()
read()
处理请求
回应数据
write()
read()
结束连接
read()
close()
close()

2.1.1 同步Accept

- 其函数定义：`public Socket Accept ()`
- **Accept** 以**同步**方式从侦听套接字的连接请求队列中提取第一个挂起的连接请求，然后创建并返回新的 Socket。
- 此处使用 Socket.**RemoteEndPoint**来获取连接机器的IP和端口信息。
- **问题：Win32 中何处获取请求连接机器的地址和端口信息？**

同步Accept 接受连接

- 在阻止模式中，Accept 将一直处于阻止状态，直到传入的连接尝试排入队列。连接被接受后，原来的 Socket 继续将传入的连接请求排入队列，直到您关闭它。

```
ListeningSocket = new Socket(
        AddressFamily.InterNetwork,
        SocketType.Stream,
        ProtocolType.IP);
IPEndPoint LocalEndPoint = new IPEndPoint(
        IPAddress.Any,
        Port);
ListeningSocket.Bind(LocalEndPoint);
ListeningSocket.Listen(5);
newCnnSocket = ListeningSocket.Accept();
```

2.1.2 同步Send发送数据

- 将数据发送到连接的 Socket：

```
public int Send (
        byte[] buffer
)
```

```
public int Send (
        byte[] buffer,
        int size,
        SocketFlags socketFlags
)
```

- SocketFlags 通常使用SocketFlags.None，发送字节数的默认值为缓冲区的大小。
- Send 对面向连接的协议和无连接协议均适用。

```
byte[] sndbuf = new byte[512];
clntsock.Send(sndbuf, sndbuf.Length, SocketFlags.None);
```

火龙果•整理
uml.org.cn

同步Send注意事项

- 如果您使用的是面向连接的协议，则除非使用Socket.SendTimeout 设置了超时值，否则，Send 将一直处于阻止状态，直到发送完缓冲区中的所有字节。
- **应由应用程序负责跟踪已发送的字节数并重试操作，直到应用程序发送完想要发送的缓冲区中的字节数为止。**
- 不能保证发送的数据会立即出现在网络上。为提高网络效率，**基础系统可能会延迟传输**，直到收集了足够多的传出数据后才开始发送。

2.1.3 同步Receive发送数据

- 从Socket中接收数据(常用第3个)：

```
public int Receive(byte[]);
public int Receive(byte[], SocketFlags);
public int Receive(byte[], int, SocketFlags);
public int Receive(byte[], int, int, SocketFlags);
```

- Receive 方法将数据读入第一个参数中，并返回成功读取的字节数。第一个int是接收多少数据到缓冲区，第二个int是指明缓冲区的偏移量。
- 如果没有可读取的数据，则 Receive 方法将一直处于阻止状态，直到数据可用，除非使用 ReceiveTimeout 设置了超时值。如果超过超时值，Receive 调用将引发 SocketException。

同步Send/Receive示例

```
byte[] msg = Encoding.UTF8.GetBytes("This is a test");
byte[] bytes = new byte[256];
try {
    // 阻塞直到发送完成
    int i = server.Send(msg);
    Console.WriteLine("Sent {0} bytes.", i);

    // 获取数据
    i = server.Receive(bytes);
    Console.WriteLine(Encoding.UTF8.GetString(bytes));
}
catch (SocketException e){
    Console.WriteLine("{0} Error code: {1}.", e.Message, e.ErrorCode)
    return (e.ErrorCode);
}
```

2.1.4 另一种方式收发数据

- System.Net.Sockets.NetworkStream 提供了用于网络访问的基础数据流。
- NetworkStream 提供的 Write 和 Read 方法用于简单的单线程同步阻止 I/O。
- 另外NetworkStream还提供异步的发送和接收数据的方法：
  - BeginWrite/EndWrite
  - BeginRead/EndRead

Write发送数据

- Write 函数定义：

```
public override void Write (
        byte[] buffer,
        int offset,
        int size
)
```

  - 参数1：byte数组，包含即将写入到NetworkStream的数据。
  - 参数2：buffer中开始写入数据的位置。
  - 参数3：准备写入NetworkStream的字节数。

Read接收数据

- 函数定义：
```
public override int Read (
        byte[] buffer,
        int offset,
        int size
    )
```
  - 参数1：用于存储从 NetworkStream 读取的数据。
  - 参数2：buffer中的开始存放数据的偏移量。
  - 参数3：buffer长度。
  - 返回值：成功读取的字节数。

Write/Read——示例

```
Socket newcnn = sock.Accept();
NetworkStream ns = new NetworkStream(newcnn);
string wmsg = "aaaaaaaaaaaaaa";
Byte[] wbuf = Encoding.BigEndianUnicode.GetBytes(wmsg);

// 发送数据
ns.Write(wbuf, 0, buf.Length);

// 接收
Byte[] rbuf = new Byte[512];
ns.Read(rbuf, 0, rbuf.Length);
string rmsg = Encoding.BigEndianUnicode.GetString(rbuf);
```

2.2 同步客户机

- 客户机需要通过三次握手完成到服务器的连接。.NET 环境提供了同步 Connect 函数完成到服务器的连接。
- TCP 连接中Connect 方法会同步建立 **LocalEndPoint** 与指定远程终结点之间的网络连接，同时保存本地地址和端口信息。
- 远程结点的信息存放在 Socket.**RemoteEndPoint** 中。
- 客户机与服务器的通信也是使用Send/Receive 或 NetworkStream 的 Read/Write 完成。

2.2.1 客户机 Connect

- .NET环境下常使用的Connect原型有：

```
public void Connect (EndPoint remoteEP);
public void Connect (IPAddress address,int port);
public void Connect (string host, int port);
```

- 其中后面两个为 .NET 2.0 新增。
- 使用示例：

```
string ip = "127.0.0.1";
int port = 8765;
IPAddress ipaddr = IPAddress.Parse(ip);
IPEndPoint ep = new IPEndPoint(ipaddr, port);
sock.Connect(ep);
```

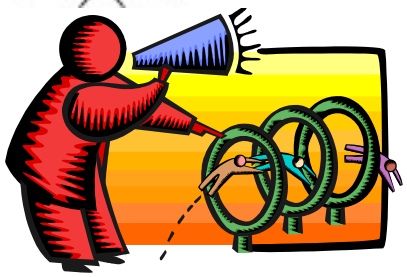
火龙果•整理
uml.org.cn

2.3 服务器/客户机示例

- 服务器(多线程同步阻塞服务器，为每一个客户新建一个线程来处理)：
  - **FirstNetwork.csproj**
- 客户机：
  - **ClientNetwork.csproj**

火龙果•整理
uml.org.cn

3. 异步服务器/客户机



- **在.NET 套接字中，异步模型是用在一个或者多个套接字上管理 I/O 的最佳途径。**
- 因为**异步模型**采用的和 Win32 下的完成端口模型类似，而且在 NT 系统中，其内部实现上使用了完成端口 I/O 模型，因此是**最有效率**的一种。
- **回忆：Win32 下有那些 I/O 模型？**

火龙果•整理
uml.org.cn

Table 6-3 *I/O Method Performance Compar*

I/O Model	Attempted/Connected	Memory Used (KB)	Non-Paged Pool	CPU Usage	Threads	Throughput (Send/ Rec Per Second)
Blocking	7000/ 1008	25,632	36,121	10 - 60%	2016	2,198,148/ 2,198,148
	12,000/ 1008	25,408	36,352	5 - 40%	2016	404,227/ 402,227
Non- blocking	7000/ 4011	4208	135,123	95 - 100%*	1	0/0
	12,000/ 5779	5224	156,260	95 - 100%*	1	0/0
WSA- Async Select	7000/ 1956	3640	38,246	75 - 85%	3	1,610,204/ 1,637,819
	12,000/ 4077	4884	42,992	90 - 100%	3	652,902/ 652,902
WSA- Event Select	7000/ 6999	10,502	36,402	65 - 85%	113	4,921,350/ 5,186,297
	12,000/ 11,080	19,214	39,040	50 - 60%	192	3,217,493/ 3,217,493
	46,000/ 45,933	37,392	121,624	80 - 90%	791	[illegible]
Over- lapped (events)	7000/ 5558	21,844	34,944	65 - 85%	66	
	12,000/12,000	60,576	48,060	35 - 45%	195	
	49,000/48,997	241,208	155,480	85 - 95%	792	
Over- lapped (comple- tion port)	7000/ 7000	36,160	31,128	40 - 50%	2	
	12,000/12,000	59,256	38,862	40 - 50%	2	
	50,000/49,997	242,272	148,192	55 - 65%	2	

服务器为Pentinum4 1.7G Xeon，内存 768M，100M网络，系统WinXP

火龙果•整理
uml.org.cn

3.1 普通程序异步执行

- C# 有多种常用的异步调用步骤，其中有两种分别是：
  - 进行某些操作，然后调用 **EndInvoke** 一直阻塞到调用完成。
  - 将用于回调方法的委托传递给 **BeginInvoke**。该方法在异步调用完成后在 **ThreadPool** 线程上执行，它可以调用 **EndInvoke**。

- **代码：Asyn-1.cs Asyn-4.cs**

3.2 网络相关异步方法

- 前面提到几种用来异步处理的 I/O 方法：
  - **BeginAccept/EndAccept;**
  - **BeginConnect/EndConnect;**
  - **BeginReceive/EndReceive;**
  - **BeginSend/EndSend;**
  - **BeginReciveFrom/EndReceiveFrom;**
  - **BeginSendTo/EndSendTo;**
  - 都是有Accept、Connect、Receive、Send等都是增加了一对 BeginXXX/EndXXX

3.3 异步方法的工作机制

- 异步套接字使用**系统线程池中的线程**处理传入的连接。
- 一些线程负责接受连接，另一些线程用于处理每个传入的连接，还有一些线程负责接收连接数据。这些线程可以是同一个线程，具体取决于线程池所分配的线程。
- System.Threading.ManualResetEvent 类挂起主线程的执行并在执行可以继续时发出信号。

火龙果•整理
uml.org.cn

3.4 性能改善

- 异步调用的执行上下文（安全上下文、模拟用户及调用上下文）会被 .NET 环境缓存，以用于 Socket 异步方法。
- 在首次使用一个特定上下文（特定 Socket 异步方法、特定 Socket 实例及特定回调）后，该上下文的后续使用将得到性能改善。

火龙果•整理
uml.org.cn

3.5 异步方法的使用

- 要异步调用一个 I/O 方法，必须调用与同步方法中对应的方法 “BeginXXX”，并在它的调用中提供一个回调方法。
- BeginXXX 调用完成后，调用线程可以继续做其它事情，而提供的回调方法则会去处理 I/O 读写。
- 在回调方法内部，当 Socket 完成 I/O 操作后，可以使用对等的 EndXXX 来获取完成的 I/O 果。

3.6 IAsyncResult 接口

- IAsyncResult 接口由包含可异步操作的方法的类实现。
- 它是启动异步操作的方法的返回类型，如 BeginAccept，也是结束异步操作的方法的参数的类型，如 EndAccept。
- IAsyncResult 也将传递给由 AsyncCallback 委托调用的方法。它的 AsyncState 属性能够获取用户定义的对象，就是在 BeginXXX 中传入的Object 对象。

```
mainSock.BeginAccept (new AsyncCallback (acceptCallback),
                                        mainSock);
void acceptCallback(IAsyncResult ar);
```

3.7 常用异步方法——Accept

- BeginAccept/EndAccept原型：

```
public IAsyncResult BeginAccept(AsyncCallback, Object);
public Socket EndAccept(IAsyncResult);
```

  - AsynCallback：指明回调方法，异步执行体；
  - Object：一个包含此请求的状态信息的对象；
  - IAsynResult：存储此异步操作的状态信息以及任何用户定义数据。
  - EndAccept：返回连接客户机的套接字。

3.7.1 异步Accept

- 为开始接受网络连接请求，其步骤如下：
  1) 初始化 Socket，并完成绑定和监听。
  2) 使用 BeginAccept 方法开始接受新连接。
  3) 当套接字上接收到新连接请求时，将调用接受回调方法。
  4) 回调方法通过 EndAccept 来获取处理客户机连接请求的 Socket 句柄，并将该 Socket 提交给将处理请求的线程。回调方法实现 **AsyncCallback** 委托；它返回 void，并带一个 IAsyncResult 类型的参数。

```
static void Main (string[] args) {
    FServer server = new FServer ();
    server.initSocket ();
    if (false == server.bindListenSocket ()) {
        Console.WriteLine ("ERR, Server will be exitting...");
        return;
    }
    server.doClient ();
}

// 开始处理客户连接请求
public bool doClient() {
    mainSock.BeginAccept (
            new AsyncCallback(acceptCallback),
            mainSock);
}

// 异步接收连接请求
void acceptCallback(IAsyncResult ar) {
    Socket sock = (Socket) ar.AsyncState;
    ClientSocket cs = new ClientSocket ();
    cs.clntsock = sock.EndAccept (ar);

    // 异步具体处理
    // ...

    // 重新标记异步接收，也可以以循环替代
    mainSock.BeginAccept (
            new AsyncCallback (acceptCallback),
            mainSock);
}
```

3.8 常用异步方法——Send

- 将数据异步的发送到连接的Socket。
- BeginSend/EndSend原型：

```
public IAsyncResult BeginSend (
    byte[] buffer,
    int offset,
    int size,
    SocketFlags socketFlags,
    AsyncCallback callback,
    Object state
)

public int EndSend (
    IAsyncResult asyncResult
)
```

常用异步方法——Send

- 参数说明：
  - buffer：存放发送数据的缓冲区；
  - offset：缓冲区中准备发送的数据的开始处；
  - size：发送数据的长度；
  - sockFlags：Socket相关标识，常用 0；
  - callback：指明回调方法；
  - state：一个包含此请求的状态信息的对象；
  - IAsynResult：存储此异步操作的状态信息；
  - EndSend：返回成功发送的数据长度。

BeginSend

- BeginSend 方法可对在 Connect、BeginConnect、Accept 或 BeginAccept 方法中建立的远程主机启动异步发送操作。
- 如果没有首先调用 Accept、BeginAccept、Connect 或 BeginConnect，则 BeginSend 将会引发异常。
- 调用 BeginSend 方法将使您能够在单独的执行线程中发送数据。

火龙果•整理
uml.org.cn

回调方法——AsyncCallback

- 创建一个实现 **AsyncCallback** 委托的回调方法并将它的名称传递给 BeginSend 方法。
- 因此，state 参数至少必须包含用于通信的已连接或默认 Socket。**如果回调需要更多信息，则可以创建一个小型类或结构，用于保存 Socket 和其他所需的信息。**通过 state 参数将此类的一个实例传递给 BeginSend 方法。

EndSend

- 回调方法应调用 EndSend 方法。
- 当应用程序调用 BeginSend 时，系统将使用一个单独的线程来执行指定的回调方法，并阻止 EndSend，直到 Socket 发送了请求的字节数或引发了异常为止。
- 如果希望在调用 BeginSend 方法之后使原始线程阻止，请使用 WaitHandle.**WaitOne** 方法。当需要原始线程继续执行时，请在回调方法中调用 System.Threading.**ManualResetEvent** 的 **Set** 方法。

```
// 异步接收数据
void receiveCallback (IAsyncResult ar) {
    int rcvnum = sock.EndReceive (ar);
    // ...
    Send (cs);
    // ...
}

// 发送数据
void Send (ClientSocket acs) {
    ClientSocket cs = acs;
    Socket sock = cs.clntsock;
    StringBuilder sb = cs.sb;

    // 发送数据的内容
    sb.Remove (0, sb.Length);
    sb.Append ("This is server response.");
    cs.buf_out = Encoding.ASCII.GetBytes (sb.ToString ());

    // 开始发送数据到客户机
    sock.BeginSend (cs.buf_out, 0, cs.buf_out.Length,
                    SocketFlags.None,
                    new AsyncCallback (sendCallback), cs);
}

// 异步发送数据
void sendCallback (IAsyncResult ar) {
    ClientSocket cs = (ClientSocket) ar.AsyncState;
    Socket sock = cs.clntsock;
    int sndnum = sock.EndSend (ar);
}
```

3.9 常用异步方法—Receive

- BeginReceive/EndReceive原型：

```
public IAsyncResult BeginReceive (
    byte[] buffer,
    int offset,
    int size,
    SocketFlags socketFlags,
    AsyncCallback callback,
    Object state
)

public int EndReceive (
    IAsyncResult asyncResult
)
```

常用异步方法—Receive

- 参数说明：
  - buffer：存放接收数据的缓冲区；
  - offset：缓冲区中准备接收的数据的开始处；
  - size：缓冲区的长度；
  - sockFlags：Socket相关标识，常用 0；
  - callback：指明回调方法；
  - state：一个包含此请求的状态信息的对象；
  - IAsynResult：它存储此异步操作的状态信息以及所有用户定义数据。
  - EndReceive：返回成功接收的数据长度。

异步接收方法使用

- 异步 BeginReceive 操作必须通过调用 EndReceive 方法来完成。通常，该方法由 callback 委托调用。
- 在 EndReceive 调用中会返回接收到的数据长度，接收到的数据填充在由 BeginReceive 中传入的缓冲区中。

火龙果•整理
uml.org.cn

```
// 异步接收连接请求
void acceptCallback(IAsyncResult ar) {
    Socket sock = (Socket) ar.AsyncState;
    ClientSocket cs = new ClientSocket ();

    cs.clntsock = sock.EndAccept (ar);
    cs.clntsock.BeginReceive (cs.buf_in, 0, ClientSocket.BUF_LEN,
                              SocketFlags.None,
                              new AsyncCallback (receiveCallback),
                              cs);
    // 重新标记，也可以以循环替代
    mainSock.BeginAccept (new AsyncCallback (acceptCallback),
                          mainSock);
}

// 异步接收数据
void receiveCallback (IAsyncResult ar) {
    ClientSocket cs = (ClientSocket) ar.AsyncState;
    int rcvnum = sock.EndReceive (ar);

    Send (cs);

    // 处理多次接收
    sock.BeginReceive (cs.buf_in, 0, ClientSocket.BUF_LEN,
                       SocketFlags.None,
                       new AsyncCallback (receiveCallback),
                       cs);
}
```

3.10 常用异步方法—Connect

- BeginConnect/EndConnect 原型：

```
public IAsyncResult BeginConnect (
    EndPoint remoteEP,
    AsyncCallback callback,
    Object state
)

public void EndConnect (
    IAsyncResult asyncResult
)
```

常用异步方法—Connect

- 参数说明：
  - remoteEP：目标的地址和IP信息；
  - callback：回调方法
  - state：一个包含此请求的状态信息的对象；
  - IAsynResult：它存储此异步操作的状态信息以及所有用户定义数据。

```
private static void StartClient() {
    client.BeginConnect( remoteEP,
            new AsyncCallback(ConnectCallback),
            client);
    connectDone.WaitOne();

    // 收发数据
    // Send/Receive
}

private static void ConnectCallback(IAsyncResult ar) {
    Socket client = (Socket) ar.AsyncState;

    // Complete the connection.
    client.EndConnect(ar);
    Console.WriteLine("Socket connected to {0}",
        client.RemoteEndPoint.ToString());

    // Signal that the connection has been made.
    connectDone.Set();
}
```

3.11 完整示例

- 异步服务器：
  - **Server.csproj**
- 异步客户机：
  - **Client.csproj**

4. TcpListener/TcpClient

- 应用程序可以通过 TCPClient、TCPListener 类来使用传输控制协议 (TCP) 服务。
- TcpListener 类提供一些简单方法，用于在同步模式下侦听和接受传入连接请求。可使用 TcpClient 或 Socket 来连接 TcpListener。
- 要使用异步 Socket 方法，可以使用 NetworkStream 类提供的异步方法。NetworkStream 也支持同步的数据读写。

4.1 TcpListener

- 可使用 IPEndPoint、本地 IP 地址及端口号或者仅使用端口号，来创建 TcpListener。
- TcpListerner.Start 方法用来开始侦听传入的连接请求。Start 将对传入连接进行排队，直至您调用 Stop 方法或它已经完成 MaxConnections 排队为止。
- 可使用 AcceptSocket 或 AcceptTcpClient 从传入连接请求队列提取连接。

4.2 TcpClient

- TcpClient 类用于在同步阻止模式下通过网络来连接、发送和接收流数据。
- 为使 TcpClient 连接并交换数据，可以使用下面两种方法之一连接到该服务器端：
  - 创建一个 TcpClient，并调用三个可用的 Connect 方法之一。
  - 使用远程主机的主机名和端口号创建 TcpClient。此构造函数将自动尝试一个连接。

4.3 示例

- 服务器：
  - TcpLinstener.cs
- 客户机：
  - TcpClient.cs

4.4 TcpListener.Pending

- public bool Pending ();
- 用于确定是否有到来的连接请求。
  - 有：返回 true；
  - 没有：返回 false；

```
TcpListener tl =  new TcpListener(ipaddr, port);
tl.Start();

if (!tl.Pending())
    Console.WriteLine("Sorry, no connection has arrived");
else{
    TcpClient tc = tl.AcceptTcpClient();
```

4.5 服务端的线程池

- 线程池使用系统的 ThreadPool 对象来处理连接请求。
- 托管线程池中的线程全部为**后台线程**，即它们的 IsBackground 属性为 true。
- 若要请求由线程池中的一个线程来处理工作项，请调用 QueueUserWorkItem 方法。

服务端的线程池——示例

- 示例源文件：
  - **ThreadPoolServer.cs**

火龙果•整理
uml.org.cn

5. UDP 通信

- UDP不能提供确保可靠的数据传输服务，但是它能使用较少的网络和系统资源进行通信，以及具备将数据发送到多个目标的能力。
- UDP的服务器的初始化和套接字创建工作和TCP的相同，只是具体参数不同。

火龙果•整理
uml.org.cn

5.1 使用UDP

- 要使用 UDP 完成通信只需要完成下面一些步骤即可：
  1) 创建支持 UDP 的套接字；
  2) 将UDP套接字绑定到本地的 IPEndPoint ；
  3) 调用 SendTo 或者 ReceiveFrom 进行通信。

5.2 创建并绑定

- UDP 创建和绑定如下：

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
                                 9050);
Socket newsock = Socket(AddressFamily.InterNetwork,
                        SocketType.Dgram,
                        ProtocolType.Udp);
newsock.Bind(ipep);
```

- TCP 创建和绑定

```
mainSock = new Socket(AddressFamily.InterNetwork,
                      SocketType.Stream,
                      ProtocolType.Tcp);
```

5.3 SendTo

- 常用SendTo 原型：

```
public int SendTo (
        byte[] buffer,
        int size,
        SocketFlags socketFlags,
        EndPoint remoteEP
)
```

  – 参数1：发送缓冲区，存放准备发送的数据；
  – 参数2：发送数据的长度；
  – 参数3：套接字标识，常用 SocketFlags.None；
  – 参数4：对方端点信息(IP/Port)；
  – 返回值：成功发送的数据长度。

5.4 ReceiveFrom

- 常用ReceiveFrom：

```
public int ReceiveFrom (
        byte[] buffer,
        ref EndPoint remoteEP
)
```

  – 参数1：接收缓冲区
  – 参数2：对方端点信息(IP/Port)；
  – 返回值：接收到的数据长度；

5.5 UDP 服务器示例

```
int recv;
byte[] data = new byte[1024];
IPEndPoint ipep = new IPEndPoint (IPAddress.Any, 9050);
Socket newsock = new Socket (AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);
newsock.Bind (ipep);

Console.WriteLine ("Waiting for a client...");
IPEndPoint sender = new IPEndPoint (IPAddress.Any, 0);
EndPoint Remote = (EndPoint) (sender);
recv = newsock.ReceiveFrom (data, ref Remote);
Console.WriteLine ("Message received from {0}:", Remote.ToString ());
Console.WriteLine (Encoding.ASCII.GetString (data, 0, recv));

string welcome = "Welcome to my test server";
data = Encoding.ASCII.GetBytes (welcome);
newsock.SendTo (data, data.Length, SocketFlags.None, Remote);

while (true) {
    data = new byte[1024];
    recv = newsock.ReceiveFrom (data, ref Remote);

    Console.WriteLine (Encoding.ASCII.GetString (data, 0, recv));
    newsock.SendTo (data, recv, SocketFlags.None, Remote);
}
```

5.6 UDP 客户机示例

火龙果•整理
uml.org.cn

```
byte[] data = new byte[1024];
string input, stringData;
IPEndPoint ipep = new IPEndPoint (
        IPAddress.Parse ("127.0.0.1"), 9050);
Socket server = new Socket (AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);

string welcome = "Hello, are you there?";
data = Encoding.ASCII.GetBytes (welcome);
server.SendTo (data, data.Length, SocketFlags.None, ipep);

IPEndPoint sender = new IPEndPoint (IPAddress.Any, 0);
EndPoint Remote = (EndPoint) sender;
data = new byte[1024];
int recv = server.ReceiveFrom (data, ref Remote);
Console.WriteLine ("Message received from {0}:", Remote.ToString ());
Console.WriteLine (Encoding.ASCII.GetString (data, 0, recv));

while (true) {
    input = Console.ReadLine ();
    if (input == "exit") break;
    server.SendTo (Encoding.ASCII.GetBytes (input), Remote);

    data = new byte[1024];
    recv = server.ReceiveFrom (data, ref Remote);
    stringData = Encoding.ASCII.GetString (data, 0, recv);
    Console.WriteLine (stringData);
}
Console.WriteLine ("Stopping client");
server.Close ();
```

火龙果•整理
uml.org.cn

5.6 一个改进的UDP传达

```
int AdvSndRcvData (Socket s, byte[] msg, EndPoint ep) {
    int recv = 0;
    int retry = 0;
    while (true) {
        Console.WriteLine ("Attempt #{0}", retry);
        try {
            s.SendTo (msg, msg.Length,
                SocketFlags.None, ep);
            data = new byte[size];
            recv = s.ReceiveFrom (data, ref Remote);
        }
        catch (SocketException e) {
            if (e.ErrorCode == 10054) // 连接被重置
                recv = 0;
            else if (e.ErrorCode == 10040) { // 消息太长
                Console.WriteLine ("Error receiving packet");
                size += 10;
                recv = 0;
            }
        }
        if (recv > 0) return recv;
        else {
            retry++;
            if (retry > 4) return 0;
        }
    }
}
```

一个改进的UDP传送

- 为什么要改进UDP通信？
- 改进的UDP传送所提供的功能：
  - 发出一个到远程主机的消息；
  - 它等待一个时间(超时时间)来接收对方的回应(通过设置 SetSocketOption 中的 SocketOptionName.ReceiveTimeout)；
  - 如果没有消息收到，它再次重发原始消息并等待接收(UDP可能会丢包)；
  - 如果接收的消息太长超过了定义的数据缓冲区，它将增大缓冲区长度并重发原始消息并等待接收。

火龙果•整理
uml.org.cn

5. UDP广播

- 要实现UDP广播主要有下面几方面需要设定的：
  1) 创建使用 UDP 的套接字；
  2) 调用套接字的 SetSocketOption()，设置其 SocketOptionName.Broadcast 为 true；
  3) 指定 IPEndPoint 的地址为广播地址：IPAddress.Broadcast；
  4) 调用 SendTo() 向广播地址上发出广播。

火龙果•整理
uml.org.cn

```
public static void Main () {
    Advertise server = new Advertise ();
}

public Advertise () {
    Thread advert = new Thread (new ThreadStart (sendPackets));
    advert.IsBackground = true;
    advert.Start ();
    Console.Write ("Press Enter to stop");
    string data = Console.ReadLine ();
}

void sendPackets () {
    Socket sock = new Socket (AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);
    sock.SetSocketOption (SocketOptionLevel.Socket,
                SocketOptionName.Broadcast, 1);
    IPEndPoint iep = new IPEndPoint (IPAddress.Broadcast, 9050);
    string hostname = Dns.GetHostName ();
    byte[] data = Encoding.ASCII.GetBytes (hostname);
    while (true) {
        sock.SendTo (data, iep);
        Thread.Sleep (60000);
    }
}
```

火龙果•整理
uml.org.cn

- *Thank You!*