目录

1、C#编写简单的聊天程序.....2
1.1、版权声明.....2
1.2、内容详情.....2
2、C#网络编程.....20
2.1、版权声明.....20
2.2、内容详情.....20
2.2.1 基本概念和操作.....20
2.2.2 同步传输字符串.....32
2.2.3 异步传输字符串.....44
2.2.4 订立协议和发送文件.....56
2.2.5 接收文件.....70
3、Scoket 处理HTTP请求响应.....77
3.1、版权声明.....77
3.2、内容详情.....77
4、浅谈C# Socket编程及多线程.....84
4.1、版权声明.....84
4.2、内容详情.....84
5、Socket编程笔记.....90
5.1、版权声明.....90
5.2、内容详情.....90
6、深入探析C# Socket.....98
6.1、版权声明.....98
6.2、内容详情.....98
7、最基本的Socket编程.....110
7.1、版权声明.....110
7.2、内容详情.....110

1、C#编写简单的聊天程序

1.1、版权声明

文章出处：http://www.cnblogs.com/JimmyZhang/archive/2008/09/07/1286299.html
文章作者：Jimmy Zhang

1.2、内容详情

引言

这是一篇基于 Socket 进行网络编程的入门文章，我对于网络编程的学习并不够深入，这篇文章是对于自己知识的一个巩固，同时希望能为初学的朋友提供一点参考。文章大体分为四个部分：程序的分析与设计、C#网络编程基础(篇外篇)、聊天程序的实现模式、程序实现。

程序的分析与设计

1.明确程序功能

如果大家现在已经参加了工作，你的经理或者老板告诉你，“小王，我需要你开发一个聊天程序”。那么接下来该怎么做呢？你是不是在脑子里有个雏形，然后就直接打开 VS2005 开始设计窗体，编写代码了呢？在开始之前，我们首先需要进行软件的分析与设计。就拿本例来说，如果只有这么一句话“一个聊天程序”，恐怕现在大家对这个“聊天程序”的概念就很模糊，它可以是像 QQ 那样的非常复杂的一个程序，也可以是很简单的聊天程序；它可能只有在对方在线的时候才可以进行聊天，也可能进行留言；它可能每次将消息只能发往一个人，也可能允许发往多个人。它还可能有一些高级功能，比如向对方传送文件等。所以我们首先需要进行分析，而不是一上手就开始做，而分析的第一步，就是搞清楚程序的功能是什么，它能够做些什么。在这一步，我们的任务是了解程序需要做什么，而不是如何去做。

了解程序需要做什么，我们可以从两方面入手，接下来我们分别讨论。

1.1 请求客户提供更详细信息

我们可以做的第一件事就是请求客户提供更加详细的信息。尽管你的经理或老板是你的上司，但在这个例子中，他就是你的客户（当然通常情况下，客户是公司外部委托公司开发软件的人或单位）。当遇到上面这种情况，我们只有少得可怜的一条信息“一个聊天程序”，首先可以做的，就是请求客户提供更加确切的信息。比如，你问经理“对这个程序的功能能不能提供一些更具体的信息？”。他可能会像这样回答：“哦，很简单，可以登录聊天程序，登录的时候能够通知其他在线用户，然后与在线的用户进行对话，如果不想对话了，就注销或者直接关闭，就这些吧。”

有了上面这段话，我们就又可以得出下面几个需求：

- 程序可以进行登录。
- 登录后可以通知其他在线用户。
- 可以与其他用户进行对话。
- 可以注销或者关闭。

1.2 对于用户需求进行提问，并进行总结

经常会有这样的情况：可能客户给出的需求仍然不够细致，或者客户自己本身对于需求就很模糊，此时我们需要做的就是针对用户上面给出的信息进行提问。接下来我就看看如何对上面的需求进行提问，我们至少可以向经理提出以下问题：

NOTE：这里我穿插一个我在见到的一个印象比较深刻的例子：客户往往向你表达了强烈的意愿他多么多么想拥有一个属于自己的网站，但是，他却没有告诉你网站都有哪些内容、栏目，可以做什么。而作为开发者，我们显然关心的是后者。

- 登录时需要提供哪些内容？需不需要提供密码？
- 允许多少人同时在线聊天？
- 与在线用户聊天时，可以将一条消息发给一个用户，还是可以一次将消息发给多个用户？
- 聊天时发送的消息包括哪些内容？
- 注销和关闭有什么区别？
- 注销和关闭对对方需不需要给对方提示？

由于这是一个范例程序，而我在为大家讲述，所以我只能再充当一下客户的角色，来回答上面的问题：

- 登录时只需要提供用户名称就可以了，不需要输入密码。
- 允许两个人在线聊天。（这里我们只讲述这种简单情况，允许多人聊天需要使用多线程）
- 因为只有两个人，那么自然是只能发给一个用户了。
- 聊天发送的消息包括：用户名称、发送时间还有正文。
- 注销并不关闭程序，只是离开了对话，可以再次进行连接。关闭则是退出整个应用程序。
- 注销和关闭均需要给对方提示。

好了，有了上面这些信息我们基本上就掌握了程序需要完成的功能，那么接下来做什么？开始编码了么？上面的这些属于业务流程，除非你对它已经非常熟悉，或者程序非常的小，那么可以对它进行编码，但是实际中，我们最好再编写一些用例，这样会使程序的流程更加的清楚。

1.3 编写用例

通常一个用例对应一个功能或者叫需求，它是程序的一个执行路径或者执行流程。编写用例的思路是：假设你已经有了这样一个聊天程序，那么你应该如何使用它？我们的使用步骤，就是一个用例。用例的特点就每次只针对程序的一个功能编写，最后根据用例编写代码，最终完成程序的开发。我们这里的需求只有简单的几个：登录，发送消息，接收消息，注销或关闭，上面的分析是对这几点功能的一个明确。接下来我们首先编写第一个用例：登录。

在开始之前，我们先明确一个概念：客户端，服务端。因为这个程序只是在两个人（机器）之间聊天，那么我们大致可以绘出这样一个图来：

我们期望用户 A 和用户 B 进行对话，那么我们就需要在它们之间建立起连接。尽管“用户 A”和“用户 B”的地位是对等的，但按照约定俗称的说法：我们将发起连接请求的一方称为客户端（或叫本地），另一端称为服务端（或叫远程）。所以我们的登录过程，就是“用户 A”连接到“用户 B”的过程，或者说客户端（本地）连接到服务端（远程）的过程。在分析这个程序的过程中，我们总是将其分为两部分，一部分为发起连接、发送消息的一方（本地），一方为接受连接、接收消息的一方（远程）。

登录和连接（本地）	
主路径	可选路径
1.打开应用程序，显示登录窗口	
2.输入用户名	
3.点击“登录”按钮，登录成功	3.“登录”失败 如果用户名为空，重新进入第2步。
4.显示主窗口，显示登录的用户名称	
5.点击“连接”，连接至远程	
6.连接成功 6.1提示用户，连接已经成功。	6.连接失败 6.1 提示用户，连接不成功
5.在用户界面变更控件状态 5.2连接为灰色，表示已经连接 5.3注销为亮色，表示可以注销 5.4发送为亮色，表示可以发消息	

这里我们的用例名称为登录和连接，但是后面我们又打了一个括号，写着“本地”，它

的意思是说，登录和连接是客户端，也就是发起连接的一方采取的动作。同样，我们需要写下当客户端连接至服务端时，服务端采取的动作。

登录和连接（远程）	
主路径	可选路径
1-4 同客户端	
5.等待连接	
6.如果有连接，**自动**在用户界面显示“远程主机连接成功”	

接下来我们来看发送消息。在发送消息时，已经是登录了的，也就是“用户 A”、“用户 B”已经做好了连接，所以我们现在就可以只关注发送这一过程：

发送消息（本地）	
主路径	可选路径
1.输入消息	
2.点击发送按钮	2.没有输入消息，重新回到第1步
3.在用户界面上显示发出的消息	3.服务端已经断开连接或者关闭 3.1在客户端用户界面上显示错误消息

然后我们看一下接收消息，此时我们只关心接收消息这一部分。

接收消息（远程）	
主路径	可选路径
1.侦听到客户端发来的消息，**自动**显示在用户界面上。	

注意到这样一点：当远程主机向本地返回消息时，它的用例又变为了上面的用例“发送消息（本地）”。因为它们的角色已经互换了。

最后看一下注销，我们这里研究的是当我们在本地机器点击“注销”后，双方采取的动作：

注销（本地主动）	
主路径	可选路径
1.点击注销按钮，断开与远程的连接	
2.在用户界面显示已经注销	
3.更改控件状态 3.1注销为灰色，表示已经注销 3.2连接为亮色，表示可以连接 3.3发送为灰色，表示无法发送	

与此对应，服务端应该作出反应：

注销（远程被动）	
主路径	可选路径
1.**自动**显示远程用户已经断开连接。	

注意到一点：当远程主动注销时，它采取的动作为上面的“本地主动”，本地采取的动作则为这里的“远程被动”。

至此，应用程序的功能分析和用例编写就告一段落了，通过上面这些表格，之后再继续编写程序变得容易了许多。另外还需要记得，用例只能为你提供一个操作步骤的指导，在实现的过程中，因为技术等方面的原因，可能还会有少量的修改。如果修改量很大，可以重新修改用例；如果修改量不大，那么就可以直接编码。这是一个迭代的过程，也没有一定的标准，总之是以高效和合适为标准。

2.分析与设计

我们已经很清楚地知道了程序需要做些什么，尽管现在还不知道该如何去做。我们甚至可以编写出这个程序所需要的接口，以后编写代码的时候，我们只要去实现这些接口就可以了。这也符合面向接口编程的原则。另外我们注意到，尽管这是一个聊天程序，但是却可以明确地划分为两部分，一部分发送消息，一部分接收消息。另外注意上面标识为自动的语句，它们暗示这个操作需要通过事件的通知机制来完成。关于委托和事件，可以参考这两篇文章：

- C#中的委托和事件 - 委托和事件的入门文章，同时捎带讲述了 Observer 设计模式和.NET 的事件模型
- C#中的委托和事件(续) - 委托和事件更深入的一些问题，包括异常、超时的处理，以及使用委托来异步调用方法。

2.1 消息 Message

首先我们可以定义消息，前面我们已经明确了消息包含三个部分：用户名、时间、内容，所以我们可以定义一个结构来表示这个消息：

```
public struct Message {
    private readonly string userName;
    private readonly string content;
    private readonly DateTime postDate;

    public Message(string userName, string content) {
        this.userName = userName;
        this.content = content;
        this.postDate = DateTime.Now;
    }

    public Message(string content) : this("System", content) { }

    public string UserName {
        get { return userName; }
    }

    public string Content {
        get { return content; }
    }

    public DateTime PostDate {
        get { return postDate; }
    }

    public override string ToString() {
        return String.Format("{0}[{1}]:    {2} ", userName, postDate, content);
    }
}
```

2.2 消息发送方 IMessageSender

从上面我们可以看出，消息发送方主要包含这样几个功能：登录、连接、发送消息、注销。另外在连接成功或失败时还要通知用户界面，发送消息成功或失败时也需要通知用户界面，因此，我们可以让连接和发送消息返回一个布尔类型的值，当它为真时表示连接或发送成功，反之则为失败。因为登录没有任何的业务逻辑，仅仅是记录控件的值并进行显示，所以我不打算将它写到接口中。因此我们可以得出它的接口大致如下：

```
public interface IMessageSender {
    bool Connect(IPAddress ip, int port);          // 连接到服务端
    bool SendMessage(Message msg);                 // 发送用户
```

```
    void SignOut();                                    // 注销系统
}
```

2.3 消息接收方 IMessageReceiver

而对于消息接收方，从上面我们可以看出，它的操作全是被动的：客户端连接时自动提示，客户端连接丢失时显示自动提示，侦听到消息时自动提示。注意到上面三个词都用了“自动”来修饰，在 C#中，可以定义委托和事件，用于当程序中某种情况发生时，通知另外一个对象。在这里，程序即是我们的 IMessageReceiver，某种情况就是上面的三种情况，而另外一个对象则为我们的用户界面。因此，我们现在首先需要定义三个委托：

```
public delegate void MessageReceivedEventHandler(string msg);
public delegate void ClientConnectedEventHandler(IPEndPoint endPoint);
public delegate void ConnectionLostEventHandler(string info);
```

接下来，我们注意到接收方需要侦听消息，因此我们需要在接口中定义的方法是 StartListen()和 StopListen()方法，这两个方法是典型的技术相关，而不是业务相关，所以从用例中是看不出来的，可能大家现在对这两个方法是做什么的还不清楚，没有关系，我们现在并不写实现，而定义接口并不需要什么成本，我们写下 IMessageReceiver 的接口定义：

```
public interface IMessageReceiver {
    event MessageReceivedEventHandler MessageReceived; // 接收到发来的消息
    event ConnectionLostEventHandler ClientLost;          // 远程主动断开连接
    event ClientConnectedEventHandler ClientConnected;   // 远程连接到了本地
    void StartListen();          // 开始侦听端口
    void StopListen();           // 停止侦听端口
}
```

我记得曾经看过有篇文章说过，最好不要在接口中定义事件，但是我忘了他的理由了，所以本文还是将事件定义在了接口中。

2.4 主程序 Talker

而我们的主程序是既可以发送，又可以接收，一般来说，如果一个类像获得其他类的能力，以采用两种方法：继承和复合。因为 C#中没有多重继承，所以我们无法同时继承实现了 IMessageReceiver 和 IMessageSender 的类。那么我们可以采用复合，将它们作为类成员包含在 Talker 内部：

```
public class Talker {
    private IMessageReceiver receiver;
    private IMessageSender sender;

    public Talker(IMessageReceiver receiver, IMessageSender sender) {
```

```
            this.receiver = receiver;
            this.sender = sender;
        }
}
```

现在，我们的程序大体框架已经完成，接下来要关注的就是如何实现它，现在让我们由设计走入实现，看看实现一个网络聊天程序，我们需要掌握的技术吧。

编写程序代码

如果你已经看完了上面一节 C#网络编程，那么本章完全没有讲解的必要了，所以我只列出代码，对个别值得注意的地方稍微地讲述一下。首先需要了解的就是，我们采用的是三个模式中开发起来难度较大的一种，无服务器参与的模式。还有就是我们没有使用广播消息，所以需要提前知道连接到的远程主机的地址和端口号。

1.实现 IMessageSender 接口

```
public class MessageSender : IMessageSender {

    TcpClient client;
    Stream streamToServer;

    // 连接至远程
    public bool Connect(IPAddress ip, int port) {
        try {
            client = new TcpClient();
            client.Connect(ip, port);
            streamToServer = client.GetStream();        // 获取连接至远程的流
            return true;
        } catch {
            return false;
        }
    }

    // 发送消息
    public bool SendMessage(Message msg) {
        try {
            lock (streamToServer) {
                byte[] buffer = Encoding.Unicode.GetBytes(msg.ToString());
                streamToServer.Write(buffer, 0, buffer.Length);
                return true;
            }
        } catch {
            return false;
```

```
            }
        }

        // 注销
        public void SignOut() {
            if (streamToServer != null)
                streamToServer.Dispose();
            if (client != null)
                client.Close();
        }
}
```

这段代码可以用朴实无华来形容，所以我们直接看下一段。

2.实现 IMessageReceiver 接口

```
public delegate void PortNumberReadyEventHandler(int portNumber);

public class MessageReceiver : IMessageReceiver {

    public event MessageReceivedEventHandler MessageReceived;
    public event ConnectionLostEventHandler ClientLost;
    public event ClientConnectedEventHandler ClientConnected;

    // 当端口号 Ok 的时候调用 -- 需要告诉用户界面使用了哪个端口号在侦听
    // 这里是业务上体现不出来，在实现中才能体现出来的
    public event PortNumberReadyEventHandler PortNumberReady;

    private Thread workerThread;
    private TcpListener listener;

    public MessageReceiver() {
        ((IMessageReceiver)this).StartListen();
    }

    // 开始侦听：显示实现接口
    void IMessageReceiver.StartListen() {
        ThreadStart start = new ThreadStart(ListenThreadMethod);
        workerThread = new Thread(start);
        workerThread.IsBackground = true;
        workerThread.Start();
    }

    // 线程入口方法
```

```
private void ListenThreadMethod() {
    IPAddress localIp = IPAddress.Parse("127.0.0.1");
    listener = new TcpListener(localIp, 0);
    listener.Start();

    // 获取端口号
    IPEndPoint endPoint = listener.LocalEndpoint as IPEndPoint;
    int portNumber = endPoint.Port;
    if (PortNumberReady != null) {
        PortNumberReady(portNumber);            // 端口号已经 OK，通知用户界面
    }

    while (true) {
        TcpClient remoteClient;
        try {
            remoteClient = listener.AcceptTcpClient();
        } catch {
            break;
        }
        if (ClientConnected != null) {
            // 连接至本机的远程端口
            endPoint = remoteClient.Client.RemoteEndPoint as IPEndPoint;
            ClientConnected(endPoint);          // 通知用户界面远程客户连接
        }

        Stream streamToClient = remoteClient.GetStream();
        byte[] buffer = new byte[8192];

        while (true) {
            try {
                int bytesRead = streamToClient.Read(buffer, 0, 8192);
                if (bytesRead == 0) {
                    throw new Exception("客户端已断开连接");
                }
                string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);

                if (MessageReceived != null) {
                    MessageReceived(msg);           // 已经收到消息
                }
            } catch (Exception ex) {
                if (ClientLost != null) {
                    ClientLost(ex.Message);         // 客户连接丢失
                    break;                          // 退出循环
                }
```

```
                    }
                }
            }
        }

        // 停止侦听端口
        public void StopListen() {
            try {
                listener.Stop();
                listener = null;
                workerThread.Abort();
            } catch { }
        }
}
```

这里需要注意的有这样几点：我们 StartListen()为显式实现接口，因为只能通过接口才能调用此方法，接口的实现类看不到此方法；这通常是对于一个接口采用两种实现方式时使用的，但这里我只是不希望 MessageReceiver 类型的客户调用它，因为在 MessageReceiver 的构造函数中它已经调用了 StartListen。意思是说，我们希望这个类型一旦创建，就立即开始工作。我们使用了两个嵌套的 while 循环，这个它可以为多个客户端的多次请求服务，但是因为是同步操作，只要有一个客户端连接着，我们的后台线程就会陷入第二个循环中无法自拔。所以结果是：如果有一个客户端已经连接上了，其它客户端即使连接了也无法对它应答。最后需要注意的就是四个事件的使用，为了向用户提供侦听的端口号以进行连接，我又定义了一个 PortNumberReadyEventHandler 委托。

3.实现 Talker 类

Talker 类是最平庸的一个类，它的全部功能就是将操作委托给实际的 IMessageReceiver 和 IMessageSender。定义这两个接口的好处也从这里可以看出来：如果日后想重新实现这个程序，所有 Windows 窗体的代码和 Talker 的代码都不需要修改，只需要针对这两个接口编程就可以了。

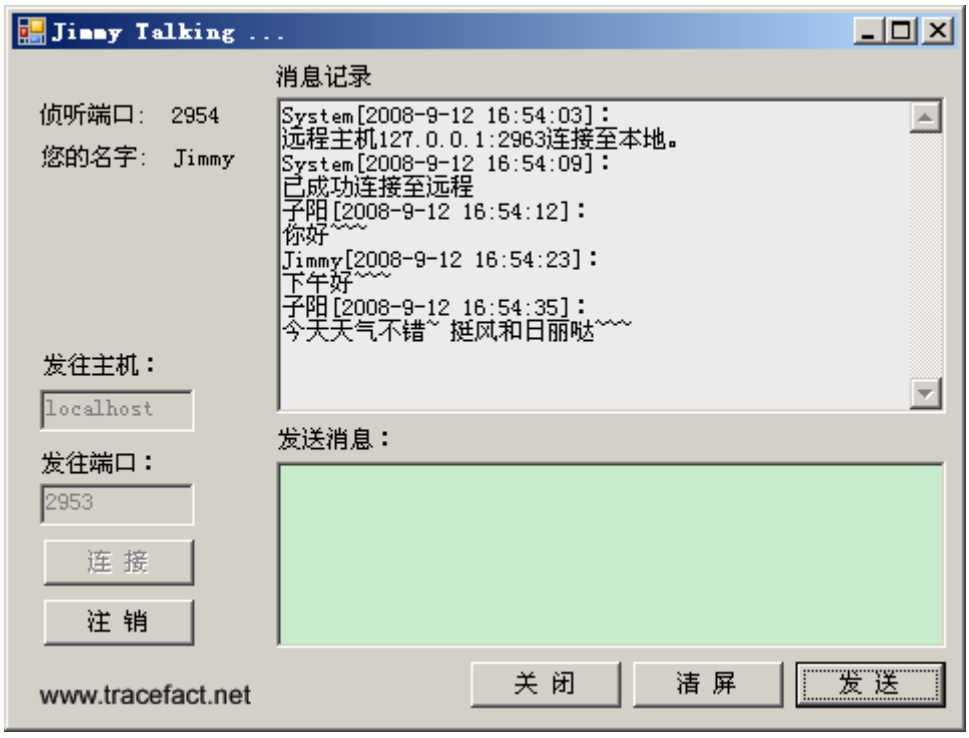
```
public class Talker {
    private IMessageReceiver receiver;
    private IMessageSender sender;

    public Talker(IMessageReceiver receiver, IMessageSender sender) {
        this.receiver = receiver;
        this.sender = sender;
    }

    public Talker() {
        this.receiver = new MessageReceiver();
        this.sender = new MessageSender();
```

```
}

public event MessageReceivedEventHandler MessageReceived {
    add {
        receiver.MessageReceived += value;
    }
    remove {
        receiver.MessageReceived -= value;
    }
}

public event ClientConnectedEventHandler ClientConnected {
    add {
        receiver.ClientConnected += value;
    }
    remove {
        receiver.ClientConnected -= value;
    }
}

public event ConnectionLostEventHandler ClientLost {
    add {
        receiver.ClientLost += value;
    }
    remove {
        receiver.ClientLost -= value;
    }
}

// 注意这个事件
public event PortNumberReadyEventHandler PortNumberReady {
    add {
        ((MessageReceiver)receiver).PortNumberReady += value;
    }
    remove {
        ((MessageReceiver)receiver).PortNumberReady -= value;
    }
}

// 连接远程 - 使用主机名
public bool ConnectByHost(string hostName, int port) {
    IPAddress[] ips = Dns.GetHostAddresses(hostName);
    return sender.Connect(ips[0], port);
}
```

```
    // 连接远程 - 使用 IP
    public bool ConnectByIp(string ip, int port) {
        IPAddress ipAddress;
        try {
            ipAddress = IPAddress.Parse(ip);
        } catch {
            return false;
        }
        return sender.Connect(ipAddress, port);
    }

    // 发送消息
    public bool SendMessage(Message msg) {
        return sender.SendMessage(msg);
    }

    // 释放资源，停止侦听
    public void Dispose() {
        try {
            sender.SignOut();
            receiver.StopListen();
        } catch {
        }
    }

    // 注销
    public void SignOut() {
        try {
            sender.SignOut();
        } catch {
        }
    }
}
```

4.设计窗体，编写窗体事件代码

现在我们开始设计窗体，我已经设计好了，现在可以先进行一下预览：



这里需要注意的就是上面的侦听端口，是程序接收消息时的侦听端口，也就是 IMessageReceiver 所使用的。其他的没有什么好说的，下来我们直接看一下代码，控件的命名是自解释的，我就不多说什么了。唯一要稍微说明下的是 txtMessage 指的是下面发送消息的文本框，txtContent 指上面的消息记录文本框：

```
public partial class PrimaryForm : Form {

    private Talker talker;
    private string userName;

    public PrimaryForm(string name) {
        InitializeComponent();
        userName = lbName.Text = name;
        this.talker = new Talker();
        this.Text = userName + " Talking ...";
        talker.ClientLost +=
            new ConnectionLostEventHandler(talker_ClientLost);
        talker.ClientConnected +=
            new ClientConnectedEventHandler(talker_ClientConnected);
        talker.MessageReceived +=
            new MessageReceivedEventHandler(talker_MessageReceived);
        talker.PortNumberReady +=
            new PortNumberReadyEventHandler(PrimaryForm_PortNumberReady);
    }

    void ConnectStatus() {    }
    void DisconnectStatus() { }

    // 端口号 OK
    void PrimaryForm_PortNumberReady(int portNumber) {
        PortNumberReadyEventHandler del = delegate(int port) {
            lbPort.Text = port.ToString();
        };
        lbPort.Invoke(del, portNumber);
    }

    // 接收到消息
    void talker_MessageReceived(string msg) {
        MessageReceivedEventHandler del = delegate(string m) {
            txtContent.Text += m;
        };
        txtContent.Invoke(del, msg);
    }

    // 有客户端连接到本机
    void talker_ClientConnected(IPEndPoint endPoint) {
        ClientConnectedEventHandler del = delegate(IPEndPoint end) {
            IPHostEntry host = Dns.GetHostEntry(end.Address);
```

```
                txtContent.Text +=
                    String.Format("System[{0}]：远程主机{1}连接至本地。 ", DateTime.Now,
end);
            };
            txtContent.Invoke(del, endPoint);
        }

        // 客户端连接断开
        void talker_ClientLost(string info) {
            ConnectionLostEventHandler del = delegate(string information) {
                txtContent.Text +=
                    String.Format("System[{0}]：  {1} ", DateTime.Now, information);
            };
            txtContent.Invoke(del, info);
        }

        // 发送消息
        private void btnSend_Click(object sender, EventArgs e) {
            if (String.IsNullOrEmpty(txtMessage.Text)) {
                MessageBox.Show("请输入内容！ ");
                txtMessage.Clear();
                txtMessage.Focus();
                return;
            }

            Message msg = new Message(userName, txtMessage.Text);
            if (talker.SendMessage(msg)) {
                txtContent.Text += msg.ToString();
                txtMessage.Clear();
            } else {
                txtContent.Text +=
                    String.Format("System[{0}]：远程主机已断开连接 ", DateTime.Now);
                DisconnectStatus();
            }
        }


        // 点击连接
        private void btnConnect_Click(object sender, EventArgs e) {

            string host = txtHost.Text;
            string ip = txtHost.Text;
            int port;
```

```
        if (String.IsNullOrEmpty(txtHost.Text)) {
            MessageBox.Show("主机名称或地址不能为空");
        }

        try{
            port = Convert.ToInt32(txtPort.Text);
        }catch{
            MessageBox.Show("端口号不能为空，且必须为数字");
            return;
        }

        if (talker.ConnectByHost(host, port)) {
            ConnectStatus();
            txtContent.Text +=
                String.Format("System[{0}]：已成功连接至远程 ", DateTime.Now);
            return;
        }

        if(talker.ConnectByIp(ip, port)){
            ConnectStatus();
            txtContent.Text +=
                String.Format("System[{0}]：已成功连接至远程 ", DateTime.Now);
        }else{
            MessageBox.Show("远程主机不存在，或者拒绝连接！");
        }

        txtMessage.Focus();
    }

    // 关闭按钮点按
    private void btnClose_Click(object sender, EventArgs e) {
        try {
            talker.Dispose();
            Application.Exit();
        } catch {
        }
    }

    // 直接点击右上角的叉
    private void PrimaryForm_FormClosing(object sender, FormClosingEventArgs e) {
        try {
            talker.Dispose();
            Application.Exit();
        } catch {
```

```
        }
    }

    // 点击注销
    private void btnSignout_Click(object sender, EventArgs e) {
        talker.SignOut();
        DisconnectStatus();
        txtContent.Text +=
            String.Format("System[{0}]：已经注销 ",DateTime.Now);
    }

    private void btnClear_Click(object sender, EventArgs e) {
        txtContent.Clear();
    }
}
```

在上面代码中，分别通过四个方法订阅了四个事件，以实现自动通知的机制。最后需要注意的就是 SignOut()和 Dispose()的区分。SignOut()只是断开连接，Dispose()则是离开应用程序。

总结

这篇文章简单地分析、设计及实现了一个聊天程序。这个程序只是对无服务器模式实现聊天的一个尝试。我们分析了需求，随后编写了几个用例，并对本地、远程的概念做了定义，接着编写了程序接口并最终实现了它。这个程序还有很严重的不足：它无法实现自动上线通知，而必须要事先知道端口号并进行手动连接。为了实现一个功能强大且开发容易的程序，更好的办法是使用集中型服务器模式。

感谢阅读，希望这篇文章能对你有所帮助。

源码下载：http://www.tracefact.net/SourceCode/CSharp-Chat.rar

2、C#网络编程

2.1、版权声明

文章出处：http://www.cnblogs.com/JimmyZhang/category/101698.html
文章作者：Jimmy Zhang

2.2、内容详情

2.2.1 基本概念和操作

引言

C#网络编程系列文章计划简单地讲述网络编程方面的基础知识，由于本人在这方面功力有限，所以只能提供一些初步的入门知识，希望能对刚开始学习的朋友提供一些帮助。如果想要更加深入的内容，可以参考相关书籍。

本文是该系列第一篇，主要讲述了基于套接字（Socket）进行网络编程的基本概念，其中包括 TCP 协议、套接字、聊天程序的三种开发模式，以及两个基本操作：侦听端口、连接远程服务端；第二篇讲述了一个简单的范例：从客户端传输字符串到服务端，服务端接收并打印字符串，将字符串改为大写，然后再将字符串回发到客户端，客户端最后打印传回的字符串；第三篇是第二篇的一个强化，讲述了第二篇中没有解决的一个问题，并使用了异步传输的方式来完成和第二篇同样的功能；第四篇则演示了如何在客户端与服务端之间收发文件；第五篇实现了一个能够在线聊天并进行文件传输的聊天程序，实际上是对前面知识的一个综合应用。

与本文相关的还有一篇文章是：C#编写简单的聊天程序，但这个聊天程序不及本系列中的聊天程序功能强大，实现方式也不相同。

网络编程基本概念

1.面向连接的传输协议：TCP

对于 TCP 协议我不想说太多东西，这属于大学课程，又涉及计算机科学，而我不是“学院派”，对于这部分内容，我觉得作为开发人员，只需要掌握与程序相关的概念就可以了，不需要做太艰深的研究。

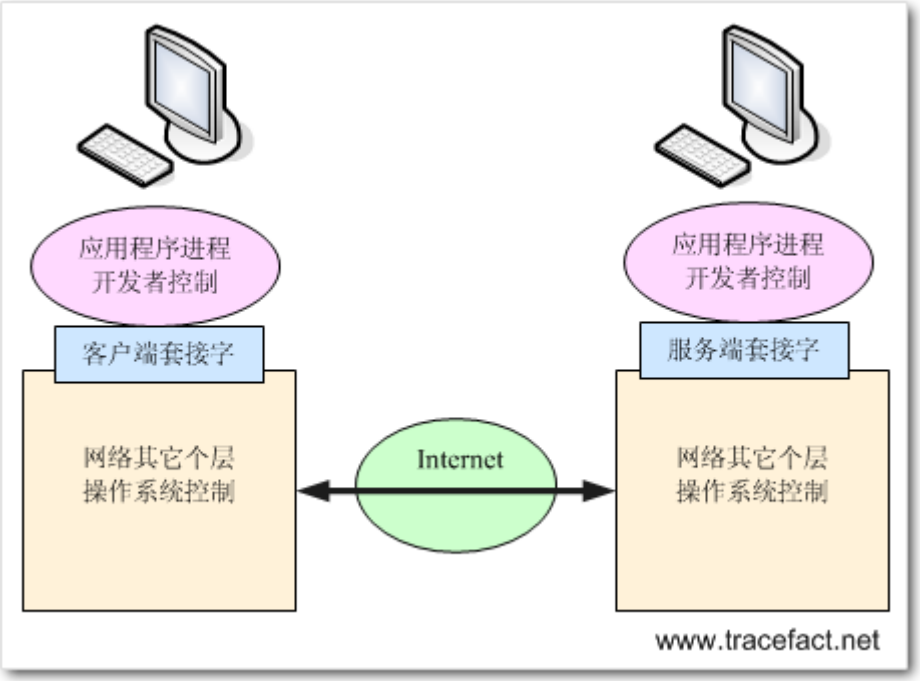
我们首先知道 TCP 是面向连接的，它的意思是说两个远程主机（或者叫进程，因为实际上远程通信是进程之间的通信，而进程则是运行中的程序），必须首先进行一个握手过程，确认连接成功，之后才能传输实际的数据。比如说进程 A 想将字符串“It's a fine day today”

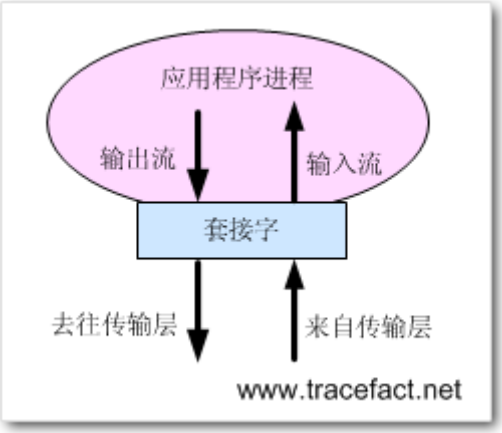
发给进程 B，它首先要建立连接。在这一过程中，它首先需要知道进程 B 的位置（主机地址和端口号）。随后发送一个不包含实际数据的请求报文，我们可以将这个报文称之为“hello”。如果进程 B 接收到了这个“hello”，就向进程 A 回复一个“hello”，进程 A 随后才发送实际的数据“It's a fine day today”。

关于 TCP 第二个需要了解的，就是它是全双工的。意思是说如果两个主机上的进程（比如进程 A、进程 B），一旦建立好连接，那么数据就既可以由 A 流向 B，也可以由 B 流向 A。除此以外，它还是点对点的，意思是说一个 TCP 连接总是两者之间的，在发送中，通过一个连接将数据发给多个接收方是不可能的。TCP 还有一个特性，就是称为可靠的数据传输，意思是连接建立后，数据的发送一定能够到达，并且是有序的，就是说发的时候你发了 ABC，那么收的一方收到的也一定是 ABC，而不会是 BCA 或者别的什么。

编程中与 TCP 相关的最重要的一个概念就是套接字。我们应该知道网络七层协议，如果我们将上面的应用程、表示层、会话层笼统地算作一层（有的教材便是如此划分的），那么我们编写的网络应用程序就位于应用层，而大家知道 TCP 是属于传输层的协议，那么我们在应用层如何使用传输层的服务呢（消息发送或者文件上传下载）？大家知道在应用程序中我们用接口来分离实现，在应用层和传输层之间，则是使用套接字来进行分离。它就像是传输层为应用层开的一个小口，应用程序通过这个小口向远程发送数据，或者接收远程发来的数据；而这个小口以内，也就是数据进入这个口之后，或者数据从这个口出来之前，我们是不知道也不需要知道的，我们也不会关心它如何传输，这属于网络其它层次的工作。

举个例子，如果你想写封邮件发给远方的朋友，那么你如何写信、将信打包，属于应用层，信怎么写，怎么打包完全由我们做主；而当我们将信投入邮筒时，邮筒的那个口就是套接字，在进入套接字之后，就是传输层、网络层等（邮局、公路交管或者航线等）其它层次的工作了。我们从来不会去关心信是如何从西安发往北京的，我们只知道写好了投入邮筒就 OK 了。可以用下面这两幅图来表示它：

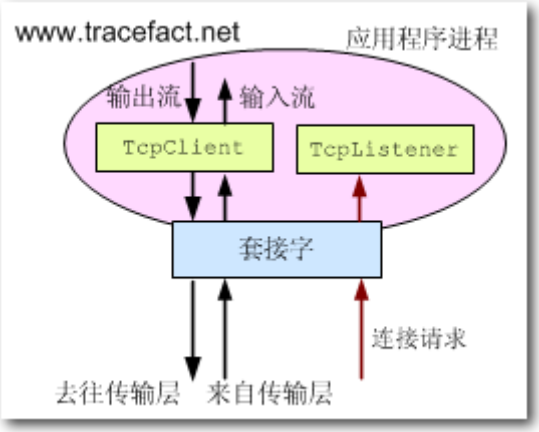




注意在上面图中，两个主机是对等的，但是按照约定，我们将发起请求的一方称为客户端，将另一端称为服务端。可以看出两个程序之间的对话是通过套接字这个出入口来完成的，实际上套接字包含的最重要的也就是两个信息：连接至远程的本地的端口信息(本机地址和端口号)，连接到的远程的端口信息(远程地址和端口号)。注意上面词语的微妙变化，一个是本地地址，一个是远程地址。

这里又出现了了一个名词端口。一般来说我们的计算机上运行着非常多的应用程序，它们可能都需要同远程主机打交道，所以远程主机就需要有一个 ID 来标识它想与本地机器上的哪个应用程序打交道，这里的 ID 就是端口。将端口分配给一个应用程序，那么来自这个端口的数据则总是针对这个应用程序的。有这样一个很好的例子：可以将主机地址想象为电话号码，而将端口号想象为分机号。

在.NET 中，尽管我们可以直接对套接字编程，但是.NET 提供了两个类将对套接字的编程进行了一个封装，使我们的使用能够更加方便，这两个类是 TcpClient 和 TcpListener，它与套接字的关系如下：



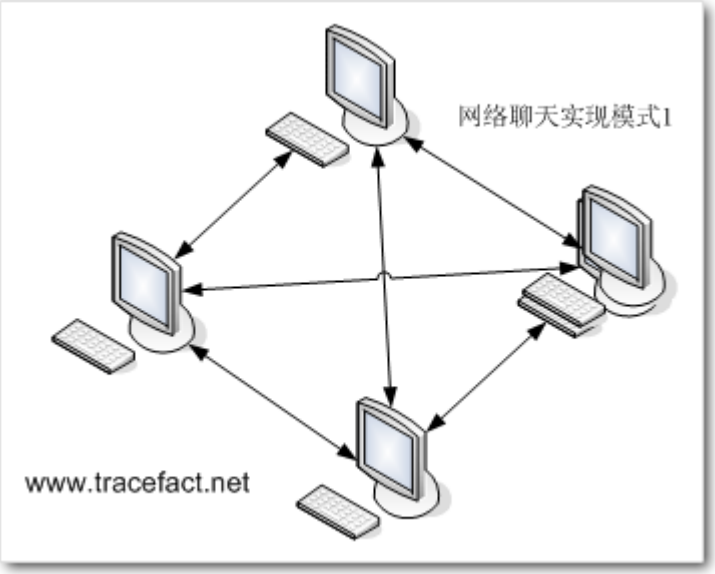
从上面图中可以看出 TcpClient 和 TcpListener 对套接字进行了封装。从中也可以看出，TcpListener 位于接收流的位置，TcpClient 位于输出流的位置（实际上 TcpListener 在收到一个请求后，就创建了 TcpClient，而它本身则持续处于侦听状态，收发数据都可以由 TcpClient

完成。这个图有点不够准确，而我暂时没有想到更好的画法，后面看到代码时会更加清楚一些）。

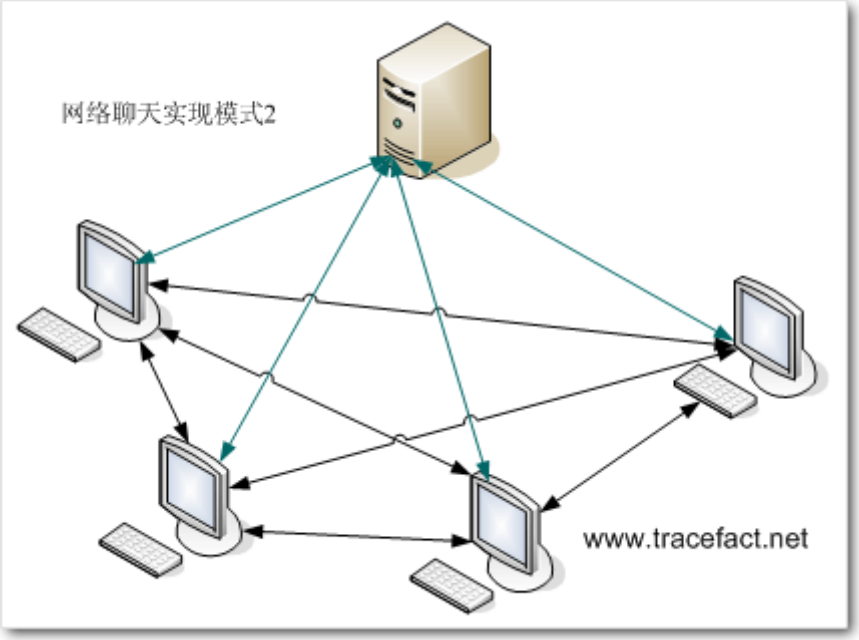
我们考虑这样一种情况：两台主机，主机 A 和主机 B，起初它们谁也不知道谁在哪儿，当它们想要进行对话时，总是需要有一方发起连接，而另一方则需要对本机的某一端口进行侦听。而在侦听方收到连接请求、并建立起连接以后，它们之间进行收发数据时，发起连接的一方并不需要再进行侦听。因为连接是全双工的，它可以使用现有的连接进行收发数据。而我们前面已经做了定义：将发起连接的一方称为客户端，另一段称为服务端，则现在可以得出：总是服务端在使用 TcpListener 类，因为它需要建立起一个初始的连接。

2.网络聊天程序的三种模式

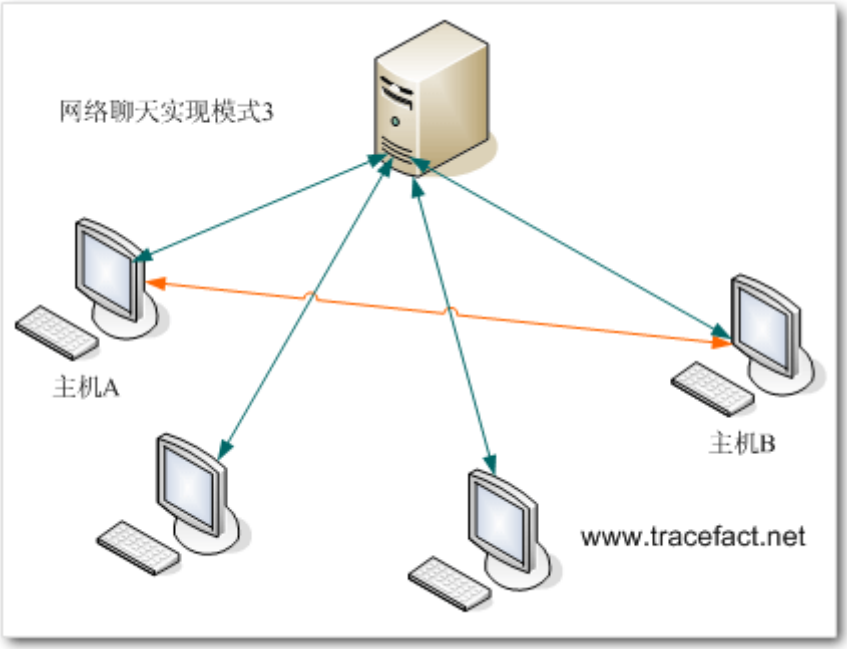
实现一个网络聊天程序本应是最后一篇文章的内容，也是本系列最后的一个程序，来作为一个终结。但是我想后面更多的是编码，讲述的内容应该不会太多，所以还是把讲述的东西都放到这里吧。



当采用这种模式时，即是所谓的完全点对点模式，此时每台计算机本身也是服务器，因为它需要进行端口的侦听。实现这个模式的难点是：各个主机（或终端）之间如何知道其它主机的存在？此时通常的做法是当某一主机上线时，使用 UDP 协议进行一个广播（Broadcast），通过这种方式来“告知”其它主机自己已经在线并说明位置，收到广播的主机发回一个应答，此时主机便知道其他主机的存在。这种方式我个人并不喜欢，但在 C#编写简单的聊天程序 这篇文章中，我使用了这种模式，可惜的是我没有实现广播，所以还很不完善。



第二种方式较好的解决了上面的问题，它引入了服务器，由这个服务器来专门进行广播。服务器持续保持对端口的侦听状态，每当有主机上线时，首先连接至服务器，服务器收到连接后，将该主机的位置（地址和端口号）发往其他在线主机（绿色箭头标识）。这样其他主机便知道该主机已上线，并知道其所在位置，从而可以进行连接和对话。在服务器进行了广播之后，因为各个主机已经知道了其他主机的位置，因此主机之间的对话就不再通过服务器（黑色箭头表示），而是直接进行连接。因此，使用这种模式时，各个主机依然需要保持对端口的侦听。在某台主机离线时，与登录时的模式类似，服务器会收到通知，然后转告给其他的主机。



第三种模式是我觉得最简单也最实用的一种，主机的登录与离线与第二种模式相同。注

意到每台主机在上线时首先就与服务器建立了连接，那么从主机 A 发往主机 B 发送消息，就可以通过这样一条路径，主机 A --> 服务器 --> 主机 B，通过这种方式，各个主机不需要在对端口进行侦听，而只需要服务器进行侦听就可以了，大大地简化了开发。

而对于一些较大的文件，比如说图片或者文件，如果想由主机 A 发往主机 B，如果通过服务器进行传输效率会比较低，此时可以临时搭建一个主机 A 至主机 B 之间的连接，用于传输大文件。当文件传输结束之后再关闭连接（桔红色箭头标识）。

除此以外，由于消息都经过服务器，所以服务器还可以缓存主机间的对话，即是说当主机 A 发往主机 B 时，如果主机 B 已经离线，则服务器可以对消息进行缓存，当主机 B 下次连接到服务器时，服务器自动将缓存的消息发给主机 B。

本系列文章最后采用的即是此种模式，不过没有实现过多复杂的功能。接下来我们的理论知识告一段落，开始下一阶段——漫长的编码。

基本操作

1.服务端对端口进行侦听

接下来我们开始编写一些实际的代码，第一步就是开启对本地机器上某一端口的侦听。首先创建一个控制台应用程序，将项目名称命名为 ServerConsole，它代表我们的服务端。如果想要与外界进行通信，第一件要做的事情就是开启对端口的侦听，这就像为计算机打开了一个“门”，所有向这个“门”发送的请求（“敲门”）都会被系统接收到。在 C#中可以通过下面几个步骤完成，首先使用本机 Ip 地址和端口号创建一个 System.Net.Sockets.TcpListener 类型的实例，然后在该实例上调用 Start()方法，从而开启对指定端口的侦听。

```
using System.Net;                    // 引入这两个命名空间，以下同
using System.Net.Sockets;
using ... // 略

class Server {
    static void Main(string[] args) {
        Console.WriteLine("Server is running ... ");
        IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });
        TcpListener listener = new TcpListener(ip, 8500);

        listener.Start();            // 开始侦听
        Console.WriteLine("Start Listening ...");

        Console.WriteLine("\n\n 输入\"Q\"键退出。");
        ConsoleKey key;
        do {
            key = Console.ReadKey(true).Key;
```

```
            } while (key != ConsoleKey.Q);
        }
}

// 获得 IPAddress 对象的另外几种常用方法：
IPAddress ip = IPAddress.Parse("127.0.0.1");
IPAddress ip = Dns.GetHostEntry("localhost").AddressList[0];
```

上面的代码中，我们开启了对 8500 端口的侦听。在运行了上面的程序之后，然后打开“命令提示符”，输入“netstat-a”，可以看到计算机器中所有打开的端口的状态。可以从中找到 8500 端口，看到它的状态是 LISTENING，这说明它已经开始了侦听：

```
  TCP    jimmy:1030          0.0.0.0:0          LISTENING
  TCP    jimmy:3603          0.0.0.0:0          LISTENING
  TCP    jimmy:8500          0.0.0.0:0          LISTENING
  TCP    jimmy:netbios-ssn   0.0.0.0:0          LISTENING
```

在打开了对端口的侦听以后，服务端必须通过某种方式进行阻塞（比如 Console.ReadKey()），使得程序不能够因为运行结束而退出。否则就无法使用“netstat -a”看到端口的连接状态，因为程序已经退出，连接会自然中断，再运行“netstat -a”当然就不会显示端口了。所以程序最后按“Q”退出那段代码是必要的，下面的每段程序都会含有这个代码段，但为了节省空间，我都省略掉了。

2.客户端与服务端连接

2.1 单一客户端与服务端连接

当服务器开始对端口侦听之后，便可以创建客户端与它建立连接。这一步是通过在客户端创建一个 TcpClient 的类型实例完成。每创建一个新的 TcpClient 便相当于创建了一个新的套接字 Socket 去与服务端通信，.Net 会自动为这个套接字分配一个端口号，上面说过，TcpClient 类不过是对 Socket 进行了一个包装。创建 TcpClient 类型实例时，可以在构造函数中指定远程服务器的地址和端口号。这样在创建的同时，就会向远程服务端发送一个连接请求（“握手”），一旦成功，则两者间的连接就建立起来了。也可以使用重载的无参数构造函数创建对象，然后再调用 Connect()方法，在 Connect()方法中传入远程服务器地址和端口号，来与服务器建立连接。

这里需要注意的是，不管是使用有参数的构造函数与服务器连接，或者是通过 Connect()方法与服务器建立连接，都是同步方法（或者说是阻塞的，英文叫 block）。它的意思是说，客户端在与服务端连接成功、从而方法返回，或者是服务端不存、从而抛出异常之前，是无法继续进行后继操作的。这里还有一个名为 BeginConnect()的方法，用于实施异步的连接，这样程序不会被阻塞，可以立即执行后面的操作，这是因为可能由于网络拥塞等问题，连接需要较长时间才能完成。网络编程中有非常多的异步操作，凡事都是由简入难，关于异步操作，我们后面再讨论，现在只看同步操作。

创建一个新的控制台应用程序项目，命名为 ClientConsole，它是我们的客户端，然后添加下面的代码，创建与服务器的连接：

```
class Client {
    static void Main(string[] args) {

        Console.WriteLine("Client Running ...");
        TcpClient client = new TcpClient();
        try {
            client.Connect("localhost", 8500);          // 与服务器连接
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
            return;
        }
        // 打印连接到的服务端信息
        Console.WriteLine("Server Connected！{0} --> {1}",
            client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

        // 按 Q 退出
    }
}
```

上面带代码中，我们通过调用 Connect()方法来与服务端连接。随后，我们打印了这个连接消息：本机的 Ip 地址和端口号，以及连接到的远程 Ip 地址和端口号。TcpClient 的 Client 属性返回了一个 Socket 对象，它的 LocalEndPoint 和 RemoteEndPoint 属性分别包含了本地和远程的地址信息。先运行服务端，再运行这段代码。可以看到两边的输出情况如下：

```
// 服务端：
Server is running ...
Start Listening ...

// 客户端：
Client Running ...
Server Connected！127.0.0.1:4761 --> 127.0.0.1:8500
```

我们看到客户端使用的端口号为 4761，上面已经说过，这个端口号是由.NET 随机选取的，并不需要我们来设置，并且每次运行时，这个端口号都不同。再次打开“命令提示符”，输入“netstat -a”，可以看到下面的输出：

```
  TCP    jimmy:8500          0.0.0.0:0           LISTENING
  TCP    jimmy:8500          localhost:4761      ESTABLISHED
  TCP    jimmy:4761          localhost:8500      ESTABLISHED
```

从这里我们可以得出几个重要信息：1、端口 8500 和端口 4761 建立了连接，这个 4761 端口便是客户端用来与服务端进行通信的端口；2、8500 端口在与客户端建立起一个连接后，仍然继续保持在监听状态。这也就是说一个端口可以与多个远程端口建立通信，这是显然的，大家众所周之的 HTTP 使用的默认端口为 80，但是一个 Web 服务器要通过这个端口与多少个浏览器通信啊。

2.2 多个客户端与服务端连接

那么既然一个服务器端口可以应对多个客户端连接，那么接下来我们就看一下，如何让多个客户端与服务端连接。如同我们上面所说的，一个 TcpClient 就是一个 Socket，所以我们只要创建多个 TcpClient，然后再调用 Connect()方法就可以了：

```
class Client {
    static void Main(string[] args) {

        Console.WriteLine("Client Running ...");
        TcpClient client;

        for (int i = 0; i <= 2; i++) {
            try {
                client = new TcpClient();
                client.Connect("localhost", 8500);          // 与服务器连接
            } catch (Exception ex) {
                Console.WriteLine(ex.Message);
                return;
            }

            // 打印连接到的服务端信息
            Console.WriteLine("Server Connected！{0} --> {1}",
                client.Client.LocalEndPoint, client.Client.RemoteEndPoint);
        }

        // 按 Q 退出
    }
}
```

上面代码最重要的就是 client = new TcpClient()这句，如果你将这个声明放到循环外面，再循环的第二趟就会发生异常，原因很显然：一个 TcpClient 对象对应一个 Socket，一个 Socket 对应着一个端口，如果不使用 new 操作符重新创建对象，那么就相当于使用一个已经与服务端建立了连接的端口再次与远程建立连接。

此时，如果在“命令提示符”运行“netstat -a”，则会看到类似下面的的输出：

```
  TCP    jimmy:8500             0.0.0.0:0              LISTENING
```

TCP	jimmy:8500	localhost:10282	ESTABLISHED
TCP	jimmy:8500	localhost:10283	ESTABLISHED
TCP	jimmy:8500	localhost:10284	ESTABLISHED
TCP	jimmy:10282	localhost:8500	ESTABLISHED
TCP	jimmy:10283	localhost:8500	ESTABLISHED
TCP	jimmy:10284	localhost:8500	ESTABLISHED

可以看到创建了三个连接对，并且 8500 端口持续保持侦听状态，从这里以及上面我们可以推断出 TcpListener 的 Start()方法是一个异步方法。

3.服务端获取客户端连接

3.1 获取单一客户端连接

上面服务端、客户端的代码已经建立起了连接，这通过使用“netstat -a”命令，从端口的状态可以看出来，但这是操作系统告诉我们的。那么我们现在需要知道的就是：服务端的程序如何知道已经与一个客户端建立起了连接？

服务器端开始侦听以后，可以在 TcpListener 实例上调用 AcceptTcpClient()来获取与一个客户端的连接，它返回一个 TcpClient 类型实例。此时它所包装的是由服务端去往客户端的 Socket，而我们在客户端创建的 TcpClient 则是由客户端去往服务端的。这个方法是一个同步方法（或者叫阻断方法，block method），意思就是说，当程序调用它以后，它会一直等待某个客户端连接，然后才会返回，否则就会一直等下去。这样的话，在调用它以后，除非得到一个客户端连接，不然不会执行接下来的代码。一个很好的类比就是 Console.ReadLine()方法，它读取输入在控制台中的一行字符串，如果有输入，就继续执行下面代码；如果没有输入，就会一直等待下去。

```
class Server {
    static void Main(string[] args) {
        Console.WriteLine("Server is running ... ");
        IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });
        TcpListener listener = new TcpListener(ip, 8500);

        listener.Start();                    // 开始侦听
        Console.WriteLine("Start Listening ...");

        // 获取一个连接，中断方法
        TcpClient remoteClient = listener.AcceptTcpClient();

        // 打印连接到的客户端信息
        Console.WriteLine("Client Connected！{0} <-- {1}",
            remoteClient.Client.LocalEndPoint, remoteClient.Client.RemoteEndPoint);

        // 按 Q 退出
```

```
    }
}
```

运行这段代码，会发现服务端运行到 listener.AcceptTcpClient()时便停止了，并不会执行下面的 Console.WriteLine()方法。为了让它继续执行下去，必须有一个客户端连接到它，所以我们现在运行客户端，与它进行连接。简单起见，我们只在客户端开启一个端口与之连接：

```
class Client {
    static void Main(string[] args) {

        Console.WriteLine("Client Running ...");
        TcpClient client = new TcpClient();
        try {
            client.Connect("localhost", 8500);          // 与服务器连接
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
            return;
        }
        // 打印连接到的服务端信息
        Console.WriteLine("Server Connected！{0} --> {1}",
            client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

        // 按 Q 退出
    }
}
```

此时，服务端、客户端的输出分别为：

```
// 服务端
Server is running ...
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:5188

// 客户端
Client Running ...
Server Connected！127.0.0.1:5188 --> 127.0.0.1:8500
```

3.2 获取多个客户端连接

现在我们再接着考虑，如果有多个客户端发动对服务器端的连接会怎么样，为了避免你将浏览器向上滚动，来查看上面的代码，我将它拷贝了下来，我们先看下客户端的关键代码：

```
TcpClient client;
```

```
for (int i = 0; i <=2; i++) {
    try {
        client = new TcpClient();
        client.Connect("localhost", 8500);        // 与服务器连接
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
        return;
    }

    // 打印连接到的服务端信息
    Console.WriteLine("Server Connected！{0} --> {1}",
        client.Client.LocalEndPoint, client.Client.RemoteEndPoint);
}
```

如果服务端代码不变，我们先运行服务端，再运行客户端，那么接下来会看到这样的输出：

```
// 服务端
Server is running ...
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:5226

// 客户端
Client Running ...
Server Connected！127.0.0.1:5226 --> 127.0.0.1:8500
Server Connected！127.0.0.1:5227 --> 127.0.0.1:8500
Server Connected！127.0.0.1:5228 --> 127.0.0.1:8500
```

就又回到了本章第 2.2 小节“多个客户端与服务端连接”中的处境：尽管有三个客户端连接到了服务端，但是服务端程序只接收到了一个。这是因为服务端只调用了一次 listener.AcceptTcpClient()，而它只对应一个连往客户端的 Socket。但是操作系统是知道连接已经建立了的，只是我们程序中没有处理到，所以我们当我们输入“netstat -a”时，仍然会看到 3 对连接都已经建立成功。

为了能够接收到三个客户端的连接，我们只要对服务端稍稍进行一下修改，将 AcceptTcpClient 方法放入一个 do/while 循环中就可以了：

```
Console.WriteLine("Start Listening ...");

while (true) {
    // 获取一个连接，同步方法
    TcpClient remoteClient = listener.AcceptTcpClient();
    // 打印连接到的客户端信息
```

```
        Console.WriteLine("Client Connected！{0} <-- {1}",
            remoteClient.Client.LocalEndPoint, remoteClient.Client.RemoteEndPoint);
}
```

这样看上去是一个死循环，但是并不会让你的机器系统资源迅速耗尽。因为前面已经说过了，AcceptTcpClient()再没有收到客户端的连接之前，是不会继续执行的，它的大部分时间都在等待。另外，服务端几乎总是要保持在运行状态，所以这样做并无不可，还可以省去“按 Q 退出”那段代码。此时再运行代码，会看到服务端可以收到 3 个客户端的连接了。

```
Server is running ...
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:5305
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:5306
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:5307
```

本篇文章到此就结束了，接下来一篇我们来看看如何在服务端与客户端之间收发数据。

源码下载：http://www.tracefact.net/SourceCode/Network-Part1-2.rar

2.2.2 同步传输字符串

服务端客户端通信

在与服务端的连接建立以后，我们就可以通过此连接来发送和接收数据。端口与端口之间以流（Stream）的形式传输数据，因为几乎任何对象都可以保存到流中，所以实际上可以在客户端与服务端之间传输任何类型的数据。对客户端来说，往流中写入数据，即为向服务器传送数据；从流中读取数据，即为从服务端接收数据。对服务端来说，往流中写入数据，即为向客户端发送数据；从流中读取数据，即为从客户端接收数据。

同步传输字符串

我们现在考虑这样一个任务：客户端打印一串字符串，然后发往服务端，服务端先输出它，然后将它改为大写，再回发到客户端，客户端接收到以后，最后再次打印一遍它。我们将它分为两部分：1、客户端发送，服务端接收并输出；2、服务端回发，客户端接收并输出。

1.客户端发送，服务端接收并输出

1.1 服务端程序

我们可以在 TcpClient 上调用 GetStream()方法来获得连接到远程计算机的流。注意这里我用了远程这个词，当在客户端调用时，它得到连接服务端的流；当在服务端调用时，它获

得连接客户端的流。接下来我们来看一下代码，我们先看服务端（注意这里没有使用 do/while 循环）：

```
class Server {
    static void Main(string[] args) {
        const int BufferSize = 8192;      // 缓存大小，8192 字节

        Console.WriteLine("Server is running ... ");
        IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });
        TcpListener listener = new TcpListener(ip, 8500);

        listener.Start();                 // 开始侦听
        Console.WriteLine("Start Listening ...");

        // 获取一个连接，中断方法
        TcpClient remoteClient = listener.AcceptTcpClient();
        // 打印连接到的客户端信息
        Console.WriteLine("Client Connected！{0} <-- {1}",
            remoteClient.Client.LocalEndPoint, remoteClient.Client.RemoteEndPoint);

        // 获得流，并写入 buffer 中
        NetworkStream streamToClient = remoteClient.GetStream();
        byte[] buffer = new byte[BufferSize];
        int bytesRead = streamToClient.Read(buffer, 0, BufferSize);
        Console.WriteLine("Reading data, {0} bytes ...", bytesRead);

        // 获得请求的字符串
        string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
        Console.WriteLine("Received: {0}", msg);

        // 按 Q 退出
    }
}
```

这段程序的上半部分已经很熟悉了，我就不再解释。remoteClient.GetStream()方法获取到了连接至客户端的流，然后从流中读出数据并保存在了 buffer 缓存中，随后使用 Encoding.Unicode.GetString()方法，从缓存中获取到了实际的字符串。最后将字符串打印在了控制台上。这段代码有个地方需要注意：在能够读取的字符串的总字节数大于 BufferSize 的时候会出现字符串截断现象，因为缓存中的数目总是有限的，而对于大对象，比如说图片或者其它文件来说，则必须采用“分次读取然后转存”这种方式，比如这样：

```
// 获取字符串
byte[] buffer = new byte[BufferSize];
int bytesRead;              // 读取的字节数
```

```
MemoryStream msStream = new MemoryStream();
do {
    bytesRead = streamToClient.Read(buffer, 0, BufferSize);
    msStream.Write(buffer, 0, bytesRead);
} while (bytesRead > 0);

buffer = msStream.GetBuffer();
string msg = Encoding.Unicode.GetString(buffer);
```

这里我没有使用这种方法，一个是因为不想关注在太多的细节上面，一个是因为对于字符串来说，8192 字节已经很多了，我们通常不会传递这么多的文本。当使用 Unicode 编码时，8192 字节可以保存 4096 个汉字和英文字符。使用不同的编码方式，占用的字节数有很大的差异，在本文最后面，有一段小程序，可以用来测试 Unicode、UTF8、ASCII 三种常用编码方式对字符串编码时，占用的字节数大小。

现在对客户端不做任何修改，然后运行先运行服务端，再运行客户端。结果我们会发现这样一件事：服务端再打印完“Client Connected！127.0.0.1:8500 <-- 127.0.0.1:xxxxx”之后，再次被阻塞了，而没有输出“Reading data, {0} bytes ...”。可见，与 AcceptTcpClient()方法类似，这个 Read()方法也是同步的，只有当客户端发送数据的时候，服务端才会读取数据、运行此方法，否则它便会一直等待。

1.2 客户端程序

接下来我们编写客户端向服务器发送字符串的代码，与服务端类似，它先获取连接服务器端的流，将字符串保存到 buffer 缓存中，再将缓存写入流，写入流这一过程，相当于将消息发往服务端。

```
class Client {
    static void Main(string[] args) {
        Console.WriteLine("Client Running ...");
        TcpClient client;

        try {
            client = new TcpClient();
            client.Connect("localhost", 8500);          // 与服务器连接
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
            return;
        }
        // 打印连接到的服务端信息
        Console.WriteLine("Server Connected！{0} --> {1}",
            client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

        string msg = "\"Welcome To TraceFact.Net\"";
```

```
            NetworkStream streamToServer = client.GetStream();

            byte[] buffer = Encoding.Unicode.GetBytes(msg);        // 获得缓存
            streamToServer.Write(buffer, 0, buffer.Length);        // 发往服务器
            Console.WriteLine("Sent: {0}", msg);

            // 按 Q 退出
        }
}
```

现在再次运行程序，得到的输出为：

```
// 服务端
Server is running ...
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:7847
Reading data, 52 bytes ...
Received: "Welcome To TraceFact.Net"
输入"Q"键退出。
// 客户端
Client Running ...
Server Connected！127.0.0.1:7847 --> 127.0.0.1:8500
Sent: "Welcome To TraceFact.Net"
输入"Q"键退出。
```

再继续进行之前，我们假设客户端可以发送多条消息，而服务端要不断的接收来自客户端发送的消息，但是上面的代码只能接收客户端发来的一条消息，因为它已经输出了“输入 Q 键退出”，说明程序已经执行完毕，无法再进行任何动作。此时如果我们再开启一个客户端，那么出现的情况是：客户端可以与服务器建立连接，也就是 netstat-a 显示为 ESTABLISHED，这是操作系统所知道的；但是由于服务端的程序已经执行到了最后一步，只能输入 Q 键退出，无法再采取任何的动作。

回想一个上面我们需要一个服务器对应多个客户端时，对 AcceptTcpClient()方法的处理办法，将它放在了 do/while 循环中；类似地，当我们需要一个服务端对同一个客户端的多次请求服务时，可以将 Read()方法放入到 do/while 循环中。

现在，我们大致可以得出这样几个结论：

- 如果不使用 do/while 循环，服务端只有一个 listener.AcceptTcpClient()方法和一个 TcpClient.GetStream().Read()方法，则服务端只能处理到同一客户端的一条请求。
- 如果使用一个 do/while 循环，并将 listener.AcceptTcpClient() 方法和 TcpClient.GetStream().Read()方法都放在这个循环以内，那么服务端将可以处理多个客户端的一条请求。
- 如果使用一个 do/while 循环，并将 listener.AcceptTcpClient()方法放在循环之外，将

TcpClient.GetStream().Read()方法放在循环以内，那么服务端可以处理一个客户端的多条请求。

- 如果使用两个 do/while 循环，对它们进行分别嵌套，那么结果是什么呢？结果并不是可以处理多个客户端的多条请求。因为里层的 do/while 循环总是在为一个客户端服务，因为它会中断在 TcpClient.GetStream().Read()方法的位置，而无法执行完毕。即使可以通过某种方式让里层循环退出，比如客户端往服务端发去“exit”字符串时，服务端也只能挨个对客户端提供服务。如果服务端想执行多个客户端的多个请求，那么服务端就需要采用多线程。主线程，也就是执行外层 do/while 循环的线程，在收到一个 TcpClient 之后，必须将里层的 do/while 循环交给新线程去执行，然后主线程快速地重新回到 listener.AcceptTcpClient()的位置，以响应其它的客户端。

对于第四种情况，实际上是构建一个服务端更为通常的情况，所以需要专门开辟一个章节讨论，这里暂且放过。而我们上面所做的，即是列出的第一种情况，接下来我们再分别看一下第二种和第三种情况。

对于第二种情况，我们按照上面的叙述先对服务端进行一下改动：

```
do {
    // 获取一个连接，中断方法
    TcpClient remoteClient = listener.AcceptTcpClient();
    // 打印连接到的客户端信息
    Console.WriteLine("Client Connected！{0} <-- {1}",
        remoteClient.Client.LocalEndPoint, remoteClient.Client.RemoteEndPoint);

    // 获得流，并写入 buffer 中
    NetworkStream streamToClient = remoteClient.GetStream();
    byte[] buffer = new byte[BufferSize];
    int bytesRead = streamToClient.Read(buffer, 0, BufferSize);
    Console.WriteLine("Reading data, {0} bytes ...", bytesRead);

    // 获得请求的字符串
    string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
    Console.WriteLine("Received: {0}", msg);
} while (true);
```

然后启动多个客户端，在服务端应该可以看到下面的输出（客户端没有变化）：

```
Server is running ...
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:8196
Reading data, 52 bytes ...
Received: "Welcome To TraceFact.Net"
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:8199
Reading data, 52 bytes ...
```

```
Received: "Welcome To TraceFact.Net"
```

由第 2 种情况改为第 3 种情况，只需要将 do 向下挪动几行就可以了：

```
// 获取一个连接，中断方法
TcpClient remoteClient = listener.AcceptTcpClient();
// 打印连接到的客户端信息
Console.WriteLine("Client Connected！{0} <-- {1}",
    remoteClient.Client.LocalEndPoint, remoteClient.Client.RemoteEndPoint);
// 获得流，并写入 buffer 中
NetworkStream streamToClient = remoteClient.GetStream();

do {
    byte[] buffer = new byte[BufferSize];
    int bytesRead = streamToClient.Read(buffer, 0, BufferSize);
    Console.WriteLine("Reading data, {0} bytes ...", bytesRead);

    // 获得请求的字符串
    string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
    Console.WriteLine("Received: {0}", msg);
} while (true);
```

然后我们再改动一下客户端，让它发送多个请求。当我们按下 S 的时候，可以输入一行字符串，然后将这行字符串发送到服务端；当我们输入 X 的时候则退出循环：

```
NetworkStream streamToServer = client.GetStream();
ConsoleKey key;
Console.WriteLine("Menu: S - Send, X - Exit");
do {
    key = Console.ReadKey(true).Key;

    if (key == ConsoleKey.S) {
        // 获取输入的字符串
        Console.Write("Input the message: ");
        string msg = Console.ReadLine();

        byte[] buffer = Encoding.Unicode.GetBytes(msg);        // 获得缓存
        streamToServer.Write(buffer, 0, buffer.Length);        // 发往服务器
        Console.WriteLine("Sent: {0}", msg);
    }
} while (key != ConsoleKey.X);
```

接下来我们先运行服务端，然后再运行客户端，输入一些字符串，来进行测试，应该能

够看到下面的输出结果：

```
// 服务端
Server is running ...
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:11004
Reading data, 44 bytes ...
Received: 欢迎访问我的博客：TraceFact.Net
Reading data, 14 bytes ...
Received: 我们一起进步！
//客户端
Client Running ...
Server Connected！127.0.0.1:11004 --> 127.0.0.1:8500
Menu: S - Send, X - Exit
Input the message: 欢迎访问我的博客：TraceFact.Net
Sent: 欢迎访问我的博客：TraceFact.Net
Input the message: 我们一起进步！
Sent: 我们一起进步！
```

这里还需要注意一点，当客户端在 TcpClient 实例上调用 Close()方法，或者在流上调用 Dispose()方法，服务端的 streamToClient.Read()方法会持续地返回 0，但是不抛出异常，所以会产生一个无限循环；而如果直接关闭掉客户端，或者客户端执行完毕但没有调用 stream.Dispose()或者 TcpClient.Close()，如果服务器端此时仍阻塞在 Read()方法处，则会在服务器端抛出异常："远程主机强制关闭了一个现有连接"。因此，我们将服务端的 streamToClient.Read()方法需要写在一个 try/catch 中。同理，如果在服务端已经连接到客户端之后，服务端调用 remoteClient.Close()，则客户端会得到异常"无法将数据写入传输连接: 您的主机中的软件放弃了一个已建立的连接。"；而如果服务端直接关闭程序的话，则客户端会得到异常"无法将数据写入传输连接: 远程主机强迫关闭了一个现有的连接。"。因此，它们的读写操作必须都放入到 try/catch 块中。

2.服务端回发，客户端接收并输出

2.2 服务端程序

我们接着再进行进一步处理，服务端将收到的字符串改为大写，然后回发，客户端接收后打印。此时它们的角色和上面完全进行了一下对调：对于服务端来说，就好像刚才的客户端一样，将字符串写入到流中；而客户端则同服务端一样，接收并打印。除此以外，我们最好对流的读写操作加上 lock，现在我们直接看代码，首先看服务端：

```
class Server {
    static void Main(string[] args) {
        const int BufferSize = 8192;      // 缓存大小，8192Bytes
        ConsoleKey key;
```

```
Console.WriteLine("Server is running ... ");
IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });
TcpListener listener = new TcpListener(ip, 8500);

listener.Start();                    // 开始侦听
Console.WriteLine("Start Listening ...");

// 获取一个连接，同步方法，在此处中断
TcpClient remoteClient = listener.AcceptTcpClient();

// 打印连接到的客户端信息
Console.WriteLine("Client Connected！{0} <-- {1}",
    remoteClient.Client.LocalEndPoint, remoteClient.Client.RemoteEndPoint);

// 获得流
NetworkStream streamToClient = remoteClient.GetStream();

do {
    // 写入 buffer 中
    byte[] buffer = new byte[BufferSize];
    int bytesRead;
    try {
        lock(streamToClient){
            bytesRead = streamToClient.Read(buffer, 0, BufferSize);
        }
        if (bytesRead == 0) throw new Exception("读取到 0 字节");
        Console.WriteLine("Reading data, {0} bytes ...", bytesRead);

        // 获得请求的字符串
        string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
        Console.WriteLine("Received: {0}", msg);

        // 转换成大写并发送
        msg = msg.ToUpper();
        buffer = Encoding.Unicode.GetBytes(msg);
        lock(streamToClient){
            streamToClient.Write(buffer, 0, buffer.Length);
        }
        Console.WriteLine("Sent: {0}", msg);
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
        break;
    }
} while (true);
```

```
            streamToClient.Dispose();
            remoteClient.Close();

            Console.WriteLine("\n\n 输入\"Q\"键退出。");
            do {
                key = Console.ReadKey(true).Key;
            } while (key != ConsoleKey.Q);
        }
}
```

接下来是客户端：

```
class Client {
    static void Main(string[] args) {
        Console.WriteLine("Client Running ...");
        TcpClient client;
        ConsoleKey key;
        const int BufferSize = 8192;

        try {
            client = new TcpClient();
            client.Connect("localhost", 8500);          // 与服务器连接
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
            return;
        }

        // 打印连接到的服务端信息
        Console.WriteLine("Server Connected！{0} --> {1}",
            client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

        NetworkStream streamToServer = client.GetStream();
        Console.WriteLine("Menu: S - Send, X - Exit");

        do {
            key = Console.ReadKey(true).Key;

            if (key == ConsoleKey.S) {
                // 获取输入的字符串
                Console.Write("Input the message: ");
                string msg = Console.ReadLine();

                byte[] buffer = Encoding.Unicode.GetBytes(msg);        // 获得缓存
```

```
                    try {
                        lock(streamToServer){
                            streamToServer.Write(buffer,  0,  buffer.Length);        //  发往服务
器
                        }
                        Console.WriteLine("Sent: {0}", msg);

                        int bytesRead;
                        buffer = new byte[BufferSize];
                        lock(streamToServer){
                            bytesRead = streamToServer.Read(buffer, 0, BufferSize);
                        }
                        msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
                        Console.WriteLine("Received: {0}", msg);

                    } catch (Exception ex) {
                        Console.WriteLine(ex.Message);
                        break;
                    }
                }
            } while (key != ConsoleKey.X);

            streamToServer.Dispose();
            client.Close();

            Console.WriteLine("\n\n 输入\"Q\"键退出。");
            do {
                key = Console.ReadKey(true).Key;
            } while (key != ConsoleKey.Q);
        }
}
```

最后我们运行程序，然后输入一串英文字符串，然后看一下输出：

```
//  客户端
Client is running ...
Server Connected！127.0.0.1:12662 --> 127.0.0.1:8500
Menu: S - Send, X - Exit
Input the message: Hello, I'm jimmy zhang.
Sent: Hello, I'm jimmy zhang.
Received: HELLO, I'M JIMMY ZHANG.

//  服务端
Server is running ...
```

```
Start Listening ...
Client Connected！127.0.0.1:8500 <-- 127.0.0.1:12662
Reading data, 46 bytes ...
Received: Hello, I'm jimmy zhang.
Sent: HELLO, I'M JIMMY ZHANG.
```

看到这里，我想你应该对使用 TcpClient 和 TcpListener 进行 C#网络编程有了一个初步的认识，可以说是刚刚入门了，后面的路还很长。本章的所有操作都是同步操作，像上面的代码也只是作为一个入门的范例，实际当中，一个服务端只能为一个客户端提供服务的情况是不存在的，下面就让我们来看看上面所说的第四种情况，如何进行异步的服务端编程。

附录：ASCII、UTF8、Uncicode 编码下的中英文字符大小

```
private static void ShowCode() {
    string[] strArray = { "b", "abcd", "乙", "甲乙丙丁" };
    byte[] buffer;
    string mode, back;

    foreach (string str in strArray) {

        for (int i = 0; i <= 2; i++) {
            if (i == 0) {
                buffer = Encoding.ASCII.GetBytes(str);
                back = Encoding.ASCII.GetString(buffer, 0, buffer.Length);
                mode = "ASCII";
            } else if (i == 1) {
                buffer = Encoding.UTF8.GetBytes(str);
                back = Encoding.UTF8.GetString(buffer, 0, buffer.Length);
                mode = "UTF8";
            } else {
                buffer = Encoding.Unicode.GetBytes(str);
                back = Encoding.Unicode.GetString(buffer, 0, buffer.Length);
                mode = "Unicode";
            }

            Console.WriteLine("Mode: {0}, String: {1}, Buffer.Length: {2}",
                mode, str, buffer.Length);

            Console.WriteLine("Buffer:");
            for (int j = 0; j <= buffer.Length - 1; j++) {
                Console.Write(buffer[j] + " ");
            }

            Console.WriteLine("\nRetrived: {0}\n", back);
```

```
            }
        }
}
```

输出为：

```
Mode: ASCII, String: b, Buffer.Length: 1
Buffer: 98
Retrived: b

Mode: UTF8, String: b, Buffer.Length: 1
Buffer: 98
Retrived: b

Mode: Unicode, String: b, Buffer.Length: 2
Buffer: 98 0
Retrived: b

Mode: ASCII, String: abcd, Buffer.Length: 4
Buffer: 97 98 99 100
Retrived: abcd

Mode: UTF8, String: abcd, Buffer.Length: 4
Buffer: 97 98 99 100
Retrived: abcd

Mode: Unicode, String: abcd, Buffer.Length: 8
Buffer: 97 0 98 0 99 0 100 0
Retrived: abcd

Mode: ASCII, String: 乙, Buffer.Length: 1
Buffer: 63
Retrived: ?

Mode: UTF8, String: 乙, Buffer.Length: 3
Buffer: 228 185 153
Retrived: 乙

Mode: Unicode, String: 乙, Buffer.Length: 2
Buffer: 89 78
Retrived: 乙

Mode: ASCII, String: 甲乙丙丁, Buffer.Length: 4
Buffer: 63 63 63 63
```

```
Retrived: ????

Mode: UTF8, String:  甲乙丙丁, Buffer.Length: 12
Buffer: 231 148 178 228 185 153 228 184 153 228 184 129
Retrived:  甲乙丙丁

Mode: Unicode, String:  甲乙丙丁, Buffer.Length: 8
Buffer: 50 117 89 78 25 78 1 78
Retrived:  甲乙丙丁
```

大体上可以得出这么几个结论：

- ASCII 不能保存中文(貌似谁都知道=_-`)。
- UTF8 是变长编码。在对 ASCII 字符编码时，UTF 更省空间，只占 1 个字节，与 ASCII 编码方式和长度相同；Unicode 在对 ASCII 字符编码时，占用 2 个字节，且第 2 个字节补零。
- UTF8 在对中文编码时需要占用 3 个字节；Unicode 对中文编码则只需要 2 个字节。

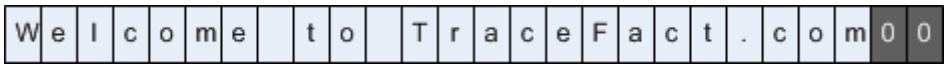
源码下载：http://www.tracefact.net/SourceCode/Network-Part1-2.rar

2.2.3 异步传输字符串

这篇文章我们将前进一大步，使用异步的方式来对服务端编程，以使它成为一个真正意义上的服务器：可以为多个客户端的多次请求服务。但是开始之前，我们需要解决上一节中遗留的一个问题。

消息发送时的问题

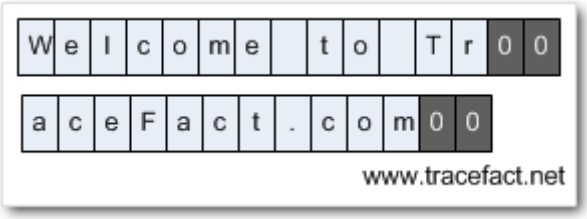
这个问题就是：客户端分两次向流中写入数据（比如字符串）时，我们主观上将这两次写入视为两次请求；然而服务端有可能将这两次合起来视为一条请求，这在两个请求间隔时间比较短的情况下尤其如此。同样，也有可能客户端发出一条请求，但是服务端将其视为两条请求处理。下面列出了可能的情况，假设我们在客户端连续发送两条“Welcome to Tracefact.net!”，则数据到达服务端时可能有这样三种情况：



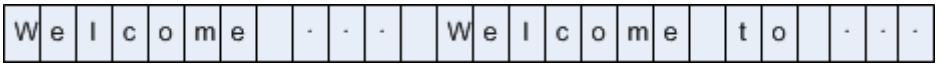
NOTE：在这里我们假设采用 ASCII 编码方式，因为此时上面的一个方框正好代表一个字节，而字符串到达末尾后为持续的 0（因为 byte 是值类型，且最小为 0）。

上面的第一种情况是最理想的情况，此时两条消息被视为两个独立请求由服务端完整地

接收。第二种情况的示意图如下，此时一条消息被当作两条消息接收了：



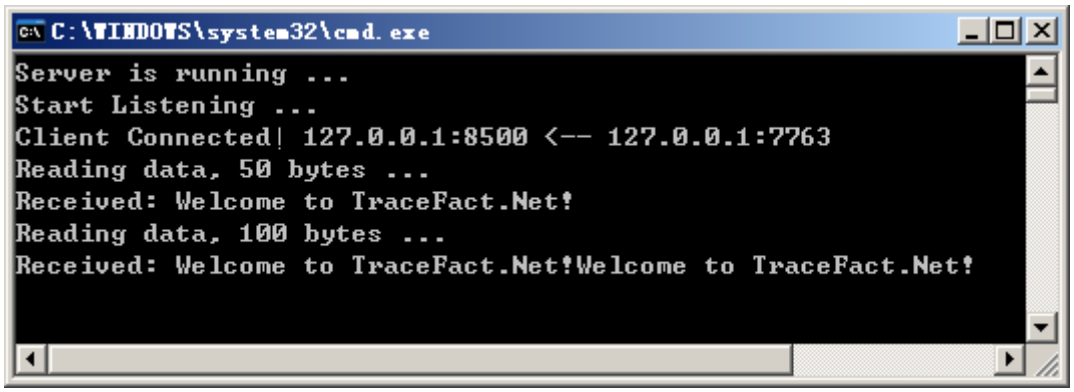
而对于第三种情况，则是两条消息被合并成了一条接收：

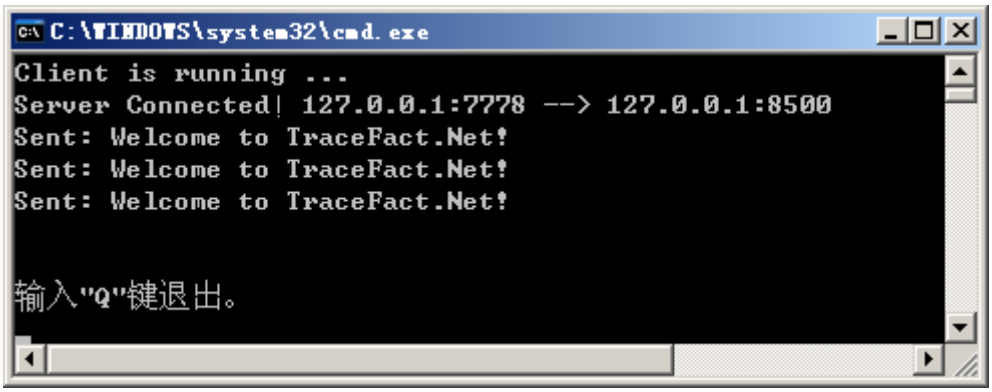


如果你下载了上一篇文章所附带的源码，那么将 Client2.cs 进行一下修改，不通过用户输入，而是使用一个 for 循环连续的发送三个请求过去，这样会使请求的间隔时间更短，下面是关键代码：

```
string msg = "Welcome to TraceFact.Net!";

for (int i = 0; i <= 2; i++) {
    byte[] buffer = Encoding.Unicode.GetBytes(msg);        // 获得缓存
    try {
        streamToServer.Write(buffer, 0, buffer.Length); // 发往服务器
        Console.WriteLine("Sent: {0}", msg);
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
        break;
    }
}
```

运行服务端，然后再运行这个客户端，你可能会看到这样的结果：





可以看到，尽管上面将消息分成了三条单独发送，但是服务端却将后两条合并成了一条。对于这些情况，我们可以这样处理：就好像 HTTP 协议一样，在实际的请求和应答内容之前包含了 HTTP 头，其中是一些与请求相关的信息。我们也可以订立自己的协议，来解决这个问题，比如说，对于上面的情况，我们就可以定义这样一个协议：

[length=XXX]：其中 xxx 是实际发送的字符串长度（注意不是字节数组 buffer 的长度），那么对于上面的请求，则我们发送的数据为："[length=25]Welcome to TraceFact.Net!"。而服务端接收字符串之后，首先读取这个"元数据"的内容，然后再根据"元数据"内容来读取实际的数据，它可能有下面这样两种情况：

NOTE：我觉得这里借用"元数据"这个术语还算比较恰当，因为"元数据"就是用来描述数据的数据。

- "[" "]"中括号是完整的，可以读取到 length 的字节数。然后根据这个数值与后面的字符串长度相比，如果相等，则说明发来了一条完整信息；如果多了，那么说明接收的字节数多了，取出合适的长度，并将剩余的进行缓存；如果少了，说明接收的不够，那么将收到的进行一个缓存，等待下次请求，然后将两条合并。
- "[""]"中括号本身就不完整，此时读不到 length 的值，因为中括号里的内容被截断了，那么将读到的数据进行缓存，等待读取下次发送来的数据，然后将两次合并之后再按上面的方式进行处理。

接下来我们来看下如何来进行实际的操作，实际上，这个问题已经不属于 C#网络编程的内容了，而完全是对字符串的处理。所以我们不再编写服务端/客户端代码，直接编写处理这几种情况的方法：

```
public class RequestHandler {
    private string temp = string.Empty;

    public string[] GetActualString(string input) {
        return GetActualString(input, null);
    }

    private string[] GetActualString(string input, List<string> outputList) {
```

```
if (outputList == null)
    outputList = new List<string>();

if (!String.IsNullOrEmpty(temp))
    input = temp + input;

string output = "";
string pattern = @"(?<=^\[length=)(\d+)(?=\])";
int length;

if (Regex.IsMatch(input, pattern)) {

    Match m = Regex.Match(input, pattern);

    // 获取消息字符串实际应有的长度
    length = Convert.ToInt32(m.Groups[0].Value);

    // 获取需要进行截取的位置
    int startIndex = input.IndexOf(']') + 1;

    // 获取从此位置开始后所有字符的长度
    output = input.Substring(startIndex);

    if (output.Length == length) {
        // 如果 output 的长度与消息字符串的应有长度相等
        // 说明刚好是完整的一条信息
        outputList.Add(output);
        temp = "";
    } else if (output.Length < length) {
        // 如果之后的长度小于应有的长度，
        // 说明没有发完整，则应将整条信息，包括元数据，全部缓存
        // 与下一条数据合并起来再进行处理
        temp = input;
        // 此时程序应该退出，因为需要等待下一条数据到来才能继续处理

    } else if (output.Length > length) {
        // 如果之后的长度大于应有的长度，
        // 说明消息发完整了，但是有多余的数据
        // 多余的数据可能是截断消息，也可能是多条完整消息

        // 截取字符串
        output = output.Substring(0, length);
        outputList.Add(output);
        temp = "";
```

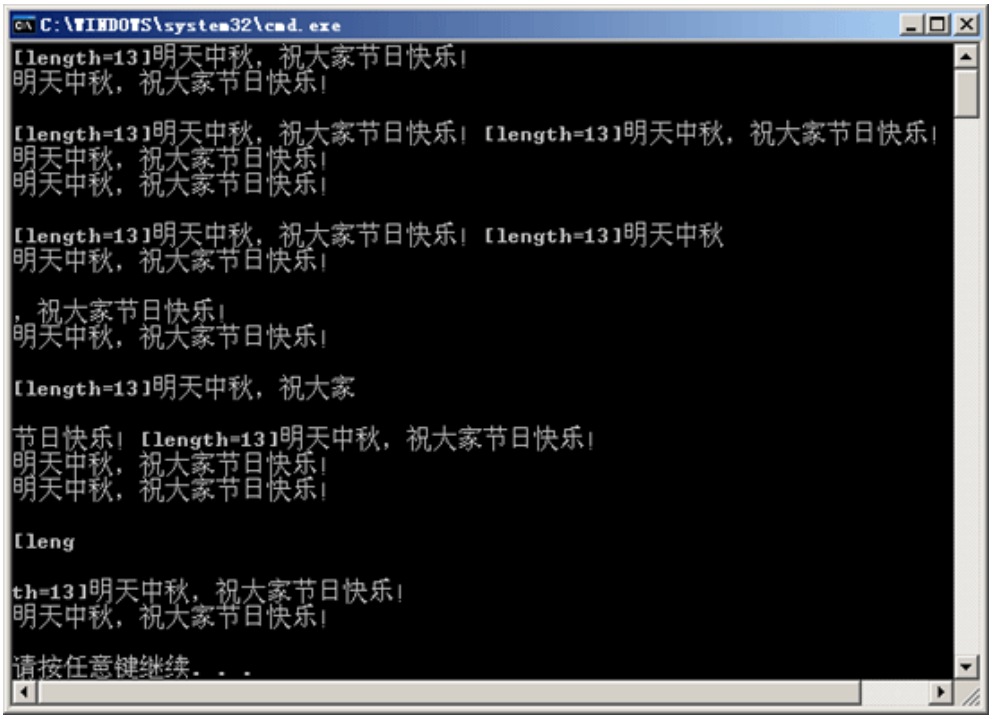
```
                    // 缩短 input 的长度
                    input = input.Substring(startIndex + length);

                    // 递归调用
                    GetActualString(input, outputList);
                }
            } else {      // 说明“[”，“]”就不完整
                temp = input;
            }

            return outputList.ToArray();
        }
}
```

这个方法接收一个满足协议格式要求的输入字符串，然后返回一个数组，这是因为如果出现多次请求合并成一个发送过来的情况，那么就将它们全部返回。随后简单起见，我在这个类中添加了一个静态的 Test()方法和 PrintOutput()帮助方法，进行了一个简单的测试，注意我直接输入了 length=13，这个是我提前计算好的。

```
public static void Test() {
    RequestHandler handler = new RequestHandler();
    string input;

    // 第一种情况测试 - 一条消息完整发送
    input = "[length=13]明天中秋，祝大家节日快乐！";
    handler.PrintOutput(input);

    // 第二种情况测试 - 两条完整消息一次发送
    input = "明天中秋，祝大家节日快乐！";
    input = String.Format
        ("[length=13]{0}[length=13]{0}", input);
    handler.PrintOutput(input);

    // 第三种情况测试 A - 两条消息不完整发送
    input = "[length=13]明天中秋，祝大家节日快乐！[length=13]明天中秋";
    handler.PrintOutput(input);

    input = "，祝大家节日快乐！";
    handler.PrintOutput(input);

    // 第三种情况测试 B - 两条消息不完整发送
    input = "[length=13]明天中秋，祝大家";
    handler.PrintOutput(input);
```

```
        input = "节日快乐！[length=13]明天中秋，祝大家节日快乐！";
        handler.PrintOutput(input);


        // 第四种情况测试 - 元数据不完整
        input = "[leng";
        handler.PrintOutput(input);        // 不会有输出

        input = "th=13]明天中秋，祝大家节日快乐！";
        handler.PrintOutput(input);

    }

    // 用于测试输出
    private void PrintOutput(string input) {
        Console.WriteLine(input);
        string[] outputArray = GetActualString(input);
        foreach (string output in outputArray) {
            Console.WriteLine(output);
        }
        Console.WriteLine();
    }
```

运行上面的程序，可以得到如下的输出：



OK，从上面的输出可以看到，这个方法能够满足我们的要求。对于这篇文章最开始提出的问题，可以很轻松地通过加入这个方法来解决，这里就不再演示了，但在本文所附带的源代码含有修改过的程序。在这里花费了很长的时间，接下来让我们回到正题，看下如何使用异步方式完成上一篇中的程序吧。

异步传输字符串

在上一篇中，我们由简到繁，提到了服务端的四种方式：服务一个客户端的一个请求、服务一个客户端的多个请求、服务多个客户端的一个请求、服务多个客户端的多个请求。我们说到可以将里层的 while 循环交给一个新建的线程去让它来完成。除了这种方式以外，我们还可以使用一种更好的方式－－使用线程池中的线程来完成。我们可以使用 BeginRead()、BeginWrite()等异步方法，同时让这 BeginRead()方法和它的回调方法形成一个类似于 while 的无限循环：首先在第一层循环中，接收到一个客户端后，调用 BeginRead()，然后为该方法提供一个读取完成后的回调方法，然后在回调方法中对收到的字符进行处理，随后在回调方法中接着调用 BeginRead()方法，并传入回调方法本身。

由于程序实现功能和上一篇完全相同，我就不再细述了。而关于异步调用方法更多详细内容，可以参见 C#中的委托和事件(续)。

1.服务端的实现

当程序越来越复杂的时候，就需要越来越高的抽象，所以从现在起我们不再把所有的代码全部都扔进 Main()里，这次我创建了一个 RemoteClient 类，它对于服务端获取到的 TcpClient 进行了一个包装：

```
public class RemoteClient {
    private TcpClient client;
    private NetworkStream streamToClient;
    private const int BufferSize = 8192;
    private byte[] buffer;
    private RequestHandler handler;

    public RemoteClient(TcpClient client) {
        this.client = client;

        // 打印连接到的客户端信息
        Console.WriteLine("\nClient Connected！{0} <-- {1}",
            client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

        // 获得流
        streamToClient = client.GetStream();
        buffer = new byte[BufferSize];
```

```
        // 设置 RequestHandler
        handler = new RequestHandler();

        // 在构造函数中就开始准备读取
        AsyncCallback callBack = new AsyncCallback(ReadComplete);
        streamToClient.BeginRead(buffer, 0, BufferSize, callBack, null);
    }

    // 再读取完成时进行回调
    private void ReadComplete(IAsyncResult ar) {
        int bytesRead = 0;
        try {
            lock (streamToClient) {
                bytesRead = streamToClient.EndRead(ar);
                Console.WriteLine("Reading data, {0} bytes ...", bytesRead);
            }
            if (bytesRead == 0) throw new Exception("读取到 0 字节");

            string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
            Array.Clear(buffer,0,buffer.Length);            // 清空缓存，避免脏读

            string[] msgArray = handler.GetActualString(msg);     // 获取实际的字符串

            // 遍历获得到的字符串
            foreach (string m in msgArray) {
                Console.WriteLine("Received: {0}", m);
                string back = m.ToUpper();

                // 将得到的字符串改为大写并重新发送
                byte[] temp = Encoding.Unicode.GetBytes(back);
                streamToClient.Write(temp, 0, temp.Length);
                streamToClient.Flush();
                Console.WriteLine("Sent: {0}", back);
            }

            // 再次调用 BeginRead()，完成时调用自身，形成无限循环
            lock (streamToClient) {
                AsyncCallback callBack = new AsyncCallback(ReadComplete);
                streamToClient.BeginRead(buffer, 0, BufferSize, callBack, null);
            }
        } catch(Exception ex) {
            if(streamToClient!=null)
                streamToClient.Dispose();
            client.Close();
```

```
                Console.WriteLine(ex.Message);          // 捕获异常时退出程序
            }
        }
}
```

随后，我们在主程序中仅仅创建 TcpListener 类型实例，由于 RemoteClient 类在构造函数中已经完成了初始化的工作，所以我们在下面的 while 循环中我们甚至不需要调用任何方法：

```
class Server {
    static void Main(string[] args) {
        Console.WriteLine("Server is running ... ");
        IPAddress ip = new IPAddress(new byte[] { 127, 0, 0, 1 });
        TcpListener listener = new TcpListener(ip, 8500);

        listener.Start();                   // 开始侦听
        Console.WriteLine("Start Listening ...");

        while (true) {
            // 获取一个连接，同步方法，在此处中断
            TcpClient client = listener.AcceptTcpClient();
            RemoteClient wapper = new RemoteClient(client);
        }
    }
}
```

好了，服务端的实现现在就完成了，接下来我们再看一下客户端的实现：

2.客户端的实现

与服务端类似，我们首先对 TcpClient 进行一个简单的包装，使它的使用更加方便一些，因为它是服务端的客户，所以我们将类的名称命名为 ServerClient：

```
public class ServerClient {
    private const int BufferSize = 8192;
    private byte[] buffer;
    private TcpClient client;
    private NetworkStream streamToServer;
    private string msg = "Welcome to TraceFact.Net!";

    public ServerClient() {
        try {
            client = new TcpClient();
```

```
            client.Connect("localhost", 8500);          // 与服务器连接
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
            return;
        }
        buffer = new byte[BufferSize];

        // 打印连接到的服务端信息
        Console.WriteLine("Server Connected！{0} --> {1}",
            client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

        streamToServer = client.GetStream();
    }

    // 连续发送三条消息到服务端
    public void SendMessage(string msg) {

        msg = String.Format("[length={0}]{1}", msg.Length, msg);

        for (int i = 0; i <= 2; i++) {
            byte[] temp = Encoding.Unicode.GetBytes(msg);     // 获得缓存
            try {
                streamToServer.Write(temp, 0, temp.Length); // 发往服务器
                Console.WriteLine("Sent: {0}", msg);
            } catch (Exception ex) {
                Console.WriteLine(ex.Message);
                break;
            }
        }

        lock (streamToServer) {
            AsyncCallback callBack = new AsyncCallback(ReadComplete);
            streamToServer.BeginRead(buffer, 0, BufferSize, callBack, null);
        }
    }

    public void SendMessage() {
        SendMessage(this.msg);
    }

    // 读取完成时的回调方法
    private void ReadComplete(IAsyncResult ar) {
        int bytesRead;
```

```
            try {
                lock (streamToServer) {
                    bytesRead = streamToServer.EndRead(ar);
                }
                if (bytesRead == 0) throw new Exception("读取到 0 字节");

                string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
                Console.WriteLine("Received: {0}", msg);
                Array.Clear(buffer, 0, buffer.Length);          // 清空缓存，避免脏读

                lock (streamToServer) {
                    AsyncCallback callBack = new AsyncCallback(ReadComplete);
                    streamToServer.BeginRead(buffer, 0, BufferSize, callBack, null);
                }
            } catch (Exception ex) {
                if(streamToServer!=null)
                    streamToServer.Dispose();
                client.Close();

                Console.WriteLine(ex.Message);
            }
        }
}
```

在上面的 SendMessage()方法中，我们让它连续发送了三条同样的消息，这么仅仅是为了测试，因为异步操作同样会出现上面说过的：服务器将客户端的请求拆开了的情况。最后我们在 Main()方法中创建这个类型的实例，然后调用 SendMessage()方法进行测试：

```
class Client {
    static void Main(string[] args) {
        ConsoleKey key;

        ServerClient client = new ServerClient();
        client.SendMessage();

        Console.WriteLine("\n\n 输入\"Q\"键退出。");
        do {
            key = Console.ReadKey(true).Key;
        } while (key != ConsoleKey.Q);
    }
}
```

是不是感觉很清爽？因为良好的代码重构，使得程序在复杂程度提高的情况下依然可以在一定程度上保持良好的阅读性。

3.程序测试

最后一步，我们先运行服务端，接着连续运行两个客户端，看看它们的输出分别是什么：

```
C:\WINDOWS\system32\cmd.exe
Server is running ...
Start Listening ...

Client Connected| 127.0.0.1:8500 <-- 127.0.0.1:10829
Reading data, 72 bytes ...
Received: Welcome to TraceFact.Net!
Sent: WELCOME TO TRACEFACT.NET!
Reading data, 144 bytes ...
Received: Welcome to TraceFact.Net!
Sent: WELCOME TO TRACEFACT.NET!
Received: Welcome to TraceFact.Net!
Sent: WELCOME TO TRACEFACT.NET!

Client Connected| 127.0.0.1:8500 <-- 127.0.0.1:10832
Reading data, 72 bytes ...
Received: Welcome to TraceFact.Net!
Sent: WELCOME TO TRACEFACT.NET!
Reading data, 144 bytes ...
Received: Welcome to TraceFact.Net!
Sent: WELCOME TO TRACEFACT.NET!
Received: Welcome to TraceFact.Net!
Sent: WELCOME TO TRACEFACT.NET!
```

```
C:\WINDOWS\system32\cmd.exe
Server Connected| 127.0.0.1:10829 --> 127.0.0.1:8500
Sent: [length=25]Welcome to TraceFact.Net!
Sent: [length=25]Welcome to TraceFact.Net!
Sent: [length=25]Welcome to TraceFact.Net!


输入"Q"键退出。
Received: WELCOME TO TRACEFACT.NET!
Received: WELCOME TO TRACEFACT.NET!WELCOME TO TRACEFACT.NET!
```

```
C:\WINDOWS\system32\cmd.exe
Server Connected| 127.0.0.1:10832 --> 127.0.0.1:8500
Sent: [length=25]Welcome to TraceFact.Net!
Sent: [length=25]Welcome to TraceFact.Net!
Sent: [length=25]Welcome to TraceFact.Net!


输入"Q"键退出。
Received: WELCOME TO TRACEFACT.NET!WELCOME TO TRACEFACT.NET!
Received: WELCOME TO TRACEFACT.NET!
```

大家可以看到，在服务端，我们可以连接多个客户端，同时为它们服务；除此以外，由接收的字节数发现，两个客户端均有两个请求被服务端合并成了一条请求，因为我们在其中加入了特殊的协议，所以在服务端可以对这种情况进行良好的处理。

在客户端，我们没有采取类似的处理，所以当客户端收到应答时，仍然会发生请求合并的情况。对于这种情况，我想大家已经知道该如何处理了，就不再多费口舌了。

使用这种定义协议的方式有它的优点，但缺点也很明显，如果客户知道了这个协议，有意地输入[length=xxx]，但是后面的长度却不匹配，此时程序就会出错。可选的解决办法是对“[”和“]”进行编码，当客户端有意输入这两个字符时，我们将它替换成“\[”和“\]”或者别的字符，在读取后再将它还原。

关于这个范例就到此结束了，剩下的两个范例都将采用异步传输的方式，并且会加入更多的协议内容。下一篇我们将介绍如何向服务端发送或接收文件。

源码下载：http://www.tracefact.net/SourceCode/Network-Part3.rar

2.2.4 订立协议和发送文件

文件传输

前面两篇文章所使用的范例都是传输字符串，有的时候我们可能会想在服务端和客户端之间传递文件。比如，考虑这样一种情况，假如客户端显示了一个菜单，当我们输入 S1、S2 或 S3（S 为 Send 缩写）时，分别向服务端发送文件 Client01.jpg、Client02.jpg、Client03.jpg；当我们输入 R1、R2 或 R3 时（R 为 Receive 缩写），则分别从服务端接收文件 Server01.jpg、Server02.jpg、Server03.jpg。那么，我们该如何完成这件事呢？此时可能有这样两种做法：

- 类似于 FTP 协议，服务端开辟两个端口，并持续对这两个端口侦听：一个用于接收字符串，类似于 FTP 的控制端口，它接收各种命令（接收或发送文件）；一个用于传输数据，也就是发送和接收文件。
- 服务端只开辟一个端口，用于接收字符串，我们称之为控制端口。当接到请求之后，根据请求内容在客户端开辟一个端口专用于文件传输，并在传输结束后关闭端口。

现在我们只关注于上面的数据端口，回忆一下在第二篇中我们所总结的，可以得出：当我们使用上面的方法一时，服务端的数据端口可以为多个客户端的多次请求服务；当我们使用方法二时，服务端只为一个客户端的一次请求服务，但是因为每次请求都会重新开辟端口，所以实际上还是相当于可以为多个客户端的多次请求服务。同时，因为它只为一次请求服务，所以我们在数据端口上传输文件时无需采用异步传输方式。但在控制端口我们仍然需要使用异步方式。

从上面看出，第一种方式要好得多，但是我们将采用第二种方式。至于原因，你可以回顾一下 Part.1（基本概念和操作）中关于聊天程序模式的讲述，因为接下来一篇文章我们将

创建一个聊天程序，而这个聊天程序采用第三种模式，所以本文的练习实际是对下一篇的一个铺垫。

1.订立协议

1.1 发送文件

我们先看一下发送文件的情况，如果我们想将文件 client01.jpg 由客户端发往客户端，那么流程是什么：

- 客户端开辟数据端口用于侦听，并获取端口号，假设为 8005。
- 假设客户端输入了 S1，则发送下面的控制字符串到服务端：[file=Client01.jpg, mode=send, port=8005]。
- 服务端收到以后，根据客户端 ip 和端口号与该客户端建立连接。
- 客户端侦听到服务端的连接，开始发送文件。
- 传送完毕后客户端、服务端分别关闭连接。

此时，我们订立的发送文件协议为：[file=Client01.jpg, mode=send, port=8005]。但是，由于它是一个普通的字符串，在上一篇中，我们采用了正则表达式来获取其中的有效值，但这显然不是一种好办法。因此，在本文及下一篇文章中，我们采用一种新的方式来编写协议：XML。对于上面的语句，我们可以写成这样的 XML：

```
<protocol><file name="client01.jpg" mode="send" port="8005" /></protocol>
```

这样我们在服务端就会好处理得多，接下来我们来看一下接收文件的流程及其协议。

NOTE：这里说发送、接收文件是站在客户端的立场说的，当客户端发送文件时，对于服务器来收，则是接收文件。

1.2 接收文件

接收文件与发送文件实际上完全类似，区别只是由客户端向网络流写入数据，还是由服务端向网络流写入数据。

- 客户端开辟数据端口用于侦听，假设为 8006。
- 假设客户端输入了 R1，则发送控制字符串：<protocol><file name="Server01.jpg" mode="receive" port="8006" /></protocol>到服务端。
- 服务端收到以后，根据客户端 ip 和端口号与该客户端建立连接。
- 客户端建立起与服务端的连接，服务端开始网络流中写入数据。
- 传送完毕后服务端、客户端分别关闭连接。

2.协议处理类的实现

和上面一章一样，在开始编写实际的服务端客户端代码之前，我们首先要编写处理协议

的类，它需要提供这样两个功能：1、方便地帮我们获取完整的协议信息，因为前面我们说过，服务端可能将客户端的多次独立请求拆分或合并。比如，客户端连续发送了两条控制信息到服务端，而服务端将它们合并了，那么则需要先拆开再分别处理。2、方便地获取我们所想要的属性信息，因为协议是 XML 格式，所以还需要一个类专门对 XML 进行处理，获得字符串的属性值。

2.1 ProtocalHandler 辅助类

我们先看下 ProtocalHandler，它与上一篇中的 RequestHandler 作用相同。需要注意的是必须将它声明为实例的，而非静态的，这是因为每个 TcpClient 都需要对应一个 ProtocalHandler，因为它内部维护的 patialProtocal 不能共享，在协议发送不完整的情况下，这个变量用于临时保存被截断的字符串。

```
public class ProtocolHandler {

    private string partialProtocal; //  保存不完整的协议

    public ProtocolHandler() {
        partialProtocal = "";
    }

    public string[] GetProtocol(string input) {
        return GetProtocol(input, null);
    }

    //  获得协议
    private string[] GetProtocol(string input, List<string> outputList) {
        if (outputList == null)
            outputList = new List<string>();

        if (String.IsNullOrEmpty(input))
            return outputList.ToArray();

        if (!String.IsNullOrEmpty(partialProtocal))
            input = partialProtocal + input;

        string pattern = "(^<protocol>.*?</protocol>)";

        //  如果有匹配，说明已经找到了，是完整的协议
        if (Regex.IsMatch(input, pattern)) {

            //  获取匹配的值
            string match = Regex.Match(input, pattern).Groups[0].Value;
            outputList.Add(match);
```

```
                partialProtocal = "";

                // 缩短 input 的长度
                input = input.Substring(match.Length);

                // 递归调用
                GetProtocol(input, outputList);

            } else {
                // 如果不匹配，说明协议的长度不够，
                // 那么先缓存，然后等待下一次请求
                partialProtocal = input;
            }

            return outputList.ToArray();
        }
}
```

因为现在它已经不是本文的重点了，所以我就不演示对于它的测试了，本文所附带的代码中含有它的测试代码（我在 ProtocolHandler 中添加了一个静态类 Test()）。

2.2 FileRequestType 枚举和 FileProtocol 结构

因为 XML 是以字符串的形式在进行传输，为了方便使用，我们最好构建一个强类型来对它们进行操作，这样会方便很多。我们首先可以定义 FileRequestMode 枚举，它代表是发送还是接收文件：

```
public enum FileRequestMode {
    Send = 0,
    Receive
}
```

接下来我们再定义一个 FileProtocol 结构，用来为整个协议字符串提供强类型的访问，注意这里覆盖了基类的 ToString()方法，这样在客户端我们就不需要再手工去编写 XML，只要在结构值上调用 ToString()就 OK 了，会方便很多。

```
public struct FileProtocol {
    private readonly FileRequestMode mode;
    private readonly int port;
    private readonly string fileName;

    public FileProtocol
        (FileRequestMode mode, int port, string fileName) {
```

```
            this.mode = mode;
            this.port = port;
            this.fileName = fileName;
        }

        public FileRequestMode Mode {
            get { return mode; }
        }

        public int Port {
            get { return port; }
        }

        public string FileName {
            get { return fileName; }
        }

        public override string ToString() {
            return String.Format("<protocol><file name=\"{0}\" mode=\"{1}\" port=\"{2}\" />></protocol>", fileName, mode, port);
        }
}
```

2.3 ProtocolHelper 辅助类

这个类专用于将 XML 格式的协议映射为我们上面定义的强类型对象，这里我没有加入 try/catch 异常处理，因为协议对用户来说是不可见的，而且客户端应该总是发送正确的协议，我觉得这样可以让代码更加清晰：

```
public class ProtocolHelper {

        private XmlNode fileNode;
        private XmlNode root;

        public ProtocolHelper(string protocol) {
            XmlDocument doc = new XmlDocument();
            doc.LoadXml(protocol);
            root = doc.DocumentElement;
            fileNode = root.SelectSingleNode("file");
        }

        // 此时的 protocal 一定为单条完整 protocal
        private FileRequestMode GetFileMode() {
            string mode = fileNode.Attributes["mode"].Value;
```

```
            mode = mode.ToLower();
            if (mode == "send")
                return FileRequestMode.Send;
            else
                return FileRequestMode.Receive;
        }

        // 获取单条协议包含的信息
        public FileProtocol GetProtocol() {
            FileRequestMode mode = GetFileMode();
            string fileName = "";
            int port = 0;

            fileName = fileNode.Attributes["name"].Value;
            port = Convert.ToInt32(fileNode.Attributes["port"].Value);

            return new FileProtocol(mode, port, fileName);
        }
}
```

OK，我们又耽误了点时间，下面就让我们进入正题吧。

3.客户端发送数据

3.1 服务端的实现

我们还是将一个问题分成两部分来处理，先是发送数据，然后是接收数据。我们先看发送数据部分的服务端。如果你从第一篇文章看到了现在，那么我觉得更多的不是技术上的问题而是思路，所以我们不再将重点放到代码上，这些应该很容易就看懂了。

```
class Server {
    static void Main(string[] args) {
        Console.WriteLine("Server is running ... ");
        IPAddress ip = IPAddress.Parse("127.0.0.1");
        TcpListener listener = new TcpListener(ip, 8500);

        listener.Start();                // 开启对控制端口 8500 的侦听
        Console.WriteLine("Start Listening ...");

        while (true) {
            // 获取一个连接，同步方法，在此处中断
            TcpClient client = listener.AcceptTcpClient();
            RemoteClient wapper = new RemoteClient(client);
            wapper.BeginRead();
```

```
            }
        }
    }

    public class RemoteClient {
        private TcpClient client;
        private NetworkStream streamToClient;
        private const int BufferSize = 8192;
        private byte[] buffer;
        private ProtocolHandler handler;

        public RemoteClient(TcpClient client) {
            this.client = client;

            // 打印连接到的客户端信息
            Console.WriteLine("\nClient Connected！{0} <-- {1}",
                client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

            // 获得流
            streamToClient = client.GetStream();
            buffer = new byte[BufferSize];

            handler = new ProtocolHandler();
        }

        // 开始进行读取
        public void BeginRead() {
            AsyncCallback callBack = new AsyncCallback(OnReadComplete);
            streamToClient.BeginRead(buffer, 0, BufferSize, callBack, null);
        }

        // 再读取完成时进行回调
        private void OnReadComplete(IAsyncResult ar) {
            int bytesRead = 0;
            try {
                lock (streamToClient) {
                    bytesRead = streamToClient.EndRead(ar);
                    Console.WriteLine("Reading data, {0} bytes ...", bytesRead);
                }
                if (bytesRead == 0) throw new Exception("读取到 0 字节");

                string msg = Encoding.Unicode.GetString(buffer, 0, bytesRead);
                Array.Clear(buffer,0,buffer.Length);            // 清空缓存，避免脏读
```

```
            // 获取 protocol 数组
            string[] protocolArray = handler.GetProtocol(msg);
            foreach (string pro in protocolArray) {
                // 这里异步调用，不然这里可能会比较耗时
                ParameterizedThreadStart start =
                    new ParameterizedThreadStart(handleProtocol);
                start.BeginInvoke(pro, null, null);
            }

            // 再次调用 BeginRead()，完成时调用自身，形成无限循环
            lock (streamToClient) {
                AsyncCallback callBack = new AsyncCallback(OnReadComplete);
                streamToClient.BeginRead(buffer, 0, BufferSize, callBack, null);
            }
        } catch(Exception ex) {
            if(streamToClient!=null)
                streamToClient.Dispose();
            client.Close();
            Console.WriteLine(ex.Message);          // 捕获异常时退出程序
        }
    }

    // 处理 protocol
    private void handleProtocol(object obj) {
        string pro = obj as string;
        ProtocolHelper helper = new ProtocolHelper(pro);
        FileProtocol protocol = helper.GetProtocol();

        if (protocol.Mode == FileRequestMode.Send) {
            // 客户端发送文件，对服务端来说则是接收文件
            receiveFile(protocol);
        } else if (protocol.Mode == FileRequestMode.Receive) {
            // 客户端接收文件，对服务端来说则是发送文件
            // sendFile(protocol);
        }
    }

    private void receiveFile(FileProtocol protocol) {
        // 获取远程客户端的位置
        IPEndPoint endpoint = client.Client.RemoteEndPoint as IPEndPoint;
        IPAddress ip = endpoint.Address;

        // 使用新端口号，获得远程用于接收文件的端口
        endpoint = new IPEndPoint(ip, protocol.Port);
```

```
        // 连接到远程客户端
        TcpClient localClient;
        try {
            localClient = new TcpClient();
            localClient.Connect(endpoint);
        } catch {
            Console.WriteLine("无法连接到客户端 --> {0}", endpoint);
            return;
        }

        // 获取发送文件的流
        NetworkStream streamToClient = localClient.GetStream();

        // 随机生成一个在当前目录下的文件名称
        string path =
            Environment.CurrentDirectory + "/" + generateFileName(protocol.FileName);

        byte[] fileBuffer = new byte[1024]; // 每次收 1KB
        FileStream fs = new FileStream(path, FileMode.CreateNew, FileAccess.Write);

        // 从缓存 buffer 中读入到文件流中
        int bytesRead;
        int totalBytes = 0;
        do {
            bytesRead = streamToClient.Read(buffer, 0, BufferSize);
            fs.Write(buffer, 0, bytesRead);
            totalBytes += bytesRead;
            Console.WriteLine("Receiving {0} bytes ...", totalBytes);
        } while (bytesRead > 0);

        Console.WriteLine("Total {0} bytes received, Done!", totalBytes);

        streamToClient.Dispose();
        fs.Dispose();
        localClient.Close();
    }

    // 随机获取一个图片名称
    private string generateFileName(string fileName) {
        DateTime now = DateTime.Now;
        return String.Format(
            "{0}_{1}_{2}_{3}", now.Minute, now.Second, now.Millisecond, fileName
        );
```

```
    }
}
```

这里应该没有什么新知识，需要注意的地方有这么几个：

- 在 OnReadComplete()回调方法中的 foreach 循环，我们使用委托异步调用了 handleProtocol()方法，这是因为 handleProtocol 即将执行的是一个读取或接收文件的操作，也就是一个相对耗时的操作。
- 在 handleProtocol()方法中，我们深切体会了定义 ProtocolHelper 类和 FileProtocol 结构的好处。如果没有定义它们，这里将是不堪入目的处理 XML 以及类型转换的代码。
- handleProtocol()方法中进行了一个条件判断，注意 sendFile()方法我屏蔽掉了，这个还没有实现，但是我想你已经猜到它将是后面要实现的内容。
- receiveFile()方法是实际接收客户端发来文件的方法，这里没有什么特别之处。需要注意的是文件存储的路径，它保存在了当前程序执行的目录下，文件的名称我使用 generateFileName()生成了一个与时间有关的随机名称。

3.2 客户端的实现

我们现在先不着急实现客户端 S1、R1 等用户菜单，首先完成发送文件这一功能，实际上，就是为上一节 SendMessage()加一个姐妹方法 SendFile()。

```
class Client {
    static void Main(string[] args) {
        ConsoleKey key;

        ServerClient client = new ServerClient();
        string filePath = Environment.CurrentDirectory + "/" + "Client01.jpg";

        if(File.Exists(filePath))
            client.BeginSendFile(filePath);

        Console.WriteLine("\n\n 输入\"Q\"键退出。");
        do {
            key = Console.ReadKey(true).Key;
        } while (key != ConsoleKey.Q);
    }
}

public class ServerClient {
    private const int BufferSize = 8192;
    private byte[] buffer;
    private TcpClient client;
    private NetworkStream streamToServer;
```

```
public ServerClient() {
    try {
        client = new TcpClient();
        client.Connect("localhost", 8500);          // 与服务器连接
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
        return;
    }
    buffer = new byte[BufferSize];

    // 打印连接到的服务端信息
    Console.WriteLine("Server Connected！{0} --> {1}",
        client.Client.LocalEndPoint, client.Client.RemoteEndPoint);

    streamToServer = client.GetStream();
}

// 发送消息到服务端
public void SendMessage(string msg) {

    byte[] temp = Encoding.Unicode.GetBytes(msg);     // 获得缓存
    try {
        lock (streamToServer) {
            streamToServer.Write(temp, 0, temp.Length); // 发往服务器
        }
        Console.WriteLine("Sent: {0}", msg);
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
        return;
    }
}

// 发送文件 - 异步方法
public void BeginSendFile(string filePath) {
    ParameterizedThreadStart start =
        new ParameterizedThreadStart(BeginSendFile);
    start.BeginInvoke(filePath, null, null);
}

private void BeginSendFile(object obj) {
    string filePath = obj as string;
    SendFile(filePath);
}
```

```
// 发送文件 -- 同步方法
public void SendFile(string filePath) {

    IPAddress ip = IPAddress.Parse("127.0.0.1");
    TcpListener listener = new TcpListener(ip, 0);
    listener.Start();

    // 获取本地侦听的端口号
    IPEndPoint endPoint = listener.LocalEndpoint as IPEndPoint;
    int listeningPort = endPoint.Port;

    // 获取发送的协议字符串
    string fileName = Path.GetFileName(filePath);
    FileProtocol protocol =
        new FileProtocol(FileRequestMode.Send, listeningPort, fileName);
    string pro = protocol.ToString();

    SendMessage(pro);          // 发送协议到服务端

    // 中断，等待远程连接
    TcpClient localClient = listener.AcceptTcpClient();
    Console.WriteLine("Start sending file...");
    NetworkStream stream = localClient.GetStream();

    // 创建文件流
    FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read);
    byte[] fileBuffer = new byte[1024];        // 每次传 1KB
    int bytesRead;
    int totalBytes = 0;

    // 创建获取文件发送状态的类
    SendStatus status = new SendStatus(filePath);

    // 将文件流转写入网络流
    try {
        do {
            Thread.Sleep(10);                  // 为了更好的视觉效果，暂停 10 毫秒:-)
            bytesRead = fs.Read(fileBuffer, 0, fileBuffer.Length);
            stream.Write(fileBuffer, 0, bytesRead);
            totalBytes += bytesRead;                  // 发送了的字节数
            status.PrintStatus(totalBytes); // 打印发送状态
        } while (bytesRead > 0);
        Console.WriteLine("Total {0} bytes sent, Done!", totalBytes);
    } catch {
```

```
                Console.WriteLine("Server has lost...");
            }

            stream.Dispose();
            fs.Dispose();
            localClient.Close();
            listener.Stop();
        }
}
```

接下来我们来看下这段代码，有这么两点需要注意一下：

- 在 Main()方法中可以看到，图片的位置为应用程序所在的目录，如果你跟我一样处于调试模式，那么就在解决方案的 Bin 目录下的 Debug 目录中放置三张图片 Client01.jpg、Client02.jpg、Client03.jpg，用来发往服务端。
- 我在客户端提供了两个 SendFile()方法，和一个 BeginSendFile()方法，分别用于同步和异步传输，其中私有的 SendFile()方法只是一个辅助方法。实际上对于发送文件这样的操作我们几乎总是需要使用异步操作。
- SendMessage()方法中给 streamToServer 加锁很重要，因为 SendFile()方法是多线程访问的，而在 SendFile()方法中又调用了 SendMessage()方法。
- 我另外编写了一个 SendStatus 类，它用来记录和打印发送完成的状态，已经发送了多少字节，完成度是百分之多少，等等。本来这个类的内容我是直接写入在 Client 类中的，后来我觉得它执行的工作已经不属于 Client 本身所应该执行的领域之内了，我记得这样一句话：当你觉得类中的方法与类的名称不符的时候，那么就应该考虑重新创建一个类。我觉得用在这里非常恰当。

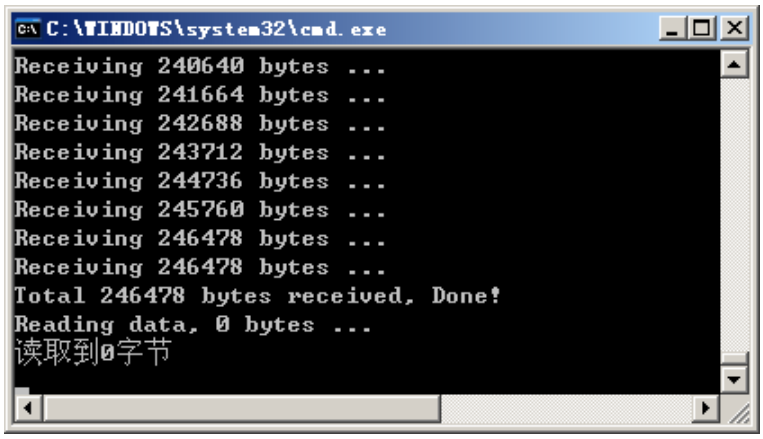
下面是 SendStatus 的内容：

```
// 即时计算发送文件的状态
public class SendStatus {
    private FileInfo info;
    private long fileBytes;

    public SendStatus(string filePath) {
        info = new FileInfo(filePath);
        fileBytes = info.Length;
    }

    public void PrintStatus(int sent) {
        string percent = GetPercent(sent);
        Console.WriteLine("Sending {0} bytes, {1}% ...", sent, percent);
    }

    // 获得文件发送的百分比
```

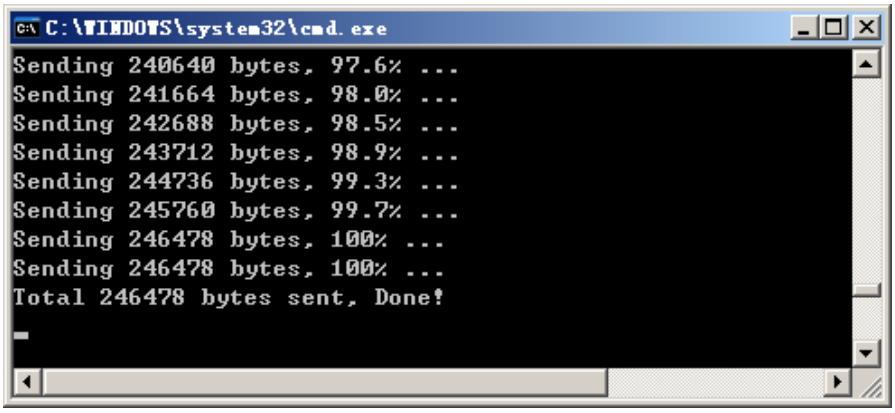
```
    public string GetPercent(int sent){

        decimal allBytes = Convert.ToDecimal(fileBytes);
        decimal currentSent = Convert.ToDecimal(sent);

        decimal percent = (currentSent / allBytes) * 100;
        percent = Math.Round(percent, 1);     //保留一位小数

        if (percent.ToString() == "100.0")
            return "100";
        else
            return percent.ToString();
    }
}
```

3.3 程序测试

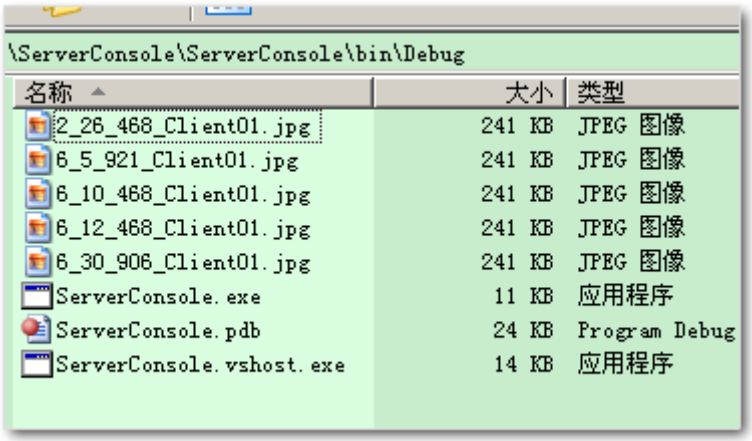
接下里我们运行一下程序，来检查一下输出，首先看下服务端：



接着是客户端，我们能够看到发送的字节数和进度，可以想到如果是图形界面，那么我们可以通过扩展 SendStatus 类来创建一个进度条：



最后我们看下服务端的 Bin\Debug 目录，应该可以看到接收到的图片：



本来我想这篇文章就可以完成发送和接收，不过现在看来没法实现了，因为如果继续下去这篇文章就太长了，我正尝试着尽量将文章控制在 15 页以内。那么我们将在下篇文章中再完成接收文件这一部分。

源码下载：http://www.tracefact.net/SourceCode/Network-Part4.rar

2.2.5 接收文件

这篇文章将完成 Part.4 中剩余的部分，它们本来是一篇完整的文章，但是因为上一篇比较长，合并起来页数太多，浏览起来可能会比较不方便，我就将它拆为两篇了，本文便是它的后半部分。我们继续进行上一篇没有完成的步骤：客户端接收来自服务端的文件。

4.客户端接收文件

4.1 服务端的实现

对于服务端，我们只需要实现上一章遗留的 sendFile()方法就可以了，它起初在 handleProtocol 中是注释掉的。另外，由于创建连接、获取流等操作与 receiveFile()是没有区别的，所以我们将它提出来作为一个公共方法 getStreamToClient()。下面是服务端的代码，只包含新增改过的代码，对于原有方法我只给出了签名：

```
class Server {
    static void Main(string[] args) {
        Console.WriteLine("Server is running ... ");
        IPAddress ip = IPAddress.Parse("127.0.0.1");
        TcpListener listener = new TcpListener(ip, 8500);
```

```
            listener.Start();                    // 开启对控制端口 8500 的侦听
            Console.WriteLine("Start Listening ...");

            while (true) {
                // 获取一个连接，同步方法，在此处中断
                TcpClient client = listener.AcceptTcpClient();
                RemoteClient wapper = new RemoteClient(client);
                wapper.BeginRead();
            }
        }
}

public class RemoteClient {
        // 字段 略

        public RemoteClient(TcpClient client) {}

        // 开始进行读取
        public void BeginRead() { }

        // 再读取完成时进行回调
        private void OnReadComplete(IAsyncResult ar) { }

        // 处理 protocol
        private void handleProtocol(object obj) {
            string pro = obj as string;
            ProtocolHelper helper = new ProtocolHelper(pro);
            FileProtocol protocol = helper.GetProtocol();

            if (protocol.Mode == FileRequestMode.Send) {
                // 客户端发送文件，对服务端来说则是接收文件
                receiveFile(protocol);
            } else if (protocol.Mode == FileRequestMode.Receive) {
                // 客户端接收文件，对服务端来说则是发送文件
                sendFile(protocol);
            }
        }

        // 发送文件
        private void sendFile(FileProtocol protocol) {
            TcpClient localClient;
            NetworkStream streamToClient = getStreamToClient(protocol, out localClient);
```

```
            // 获得文件的路径
            string filePath = Environment.CurrentDirectory + "/" + protocol.FileName;

            // 创建文件流
            FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read);
            byte[] fileBuffer = new byte[1024];        // 每次传 1KB
            int bytesRead;
            int totalBytes = 0;

            // 创建获取文件发送状态的类
            SendStatus status = new SendStatus(filePath);

            // 将文件流转写入网络流
            try {
                do {
                    Thread.Sleep(10);                  // 为了更好的视觉效果，暂停 10 毫秒:-)
                    bytesRead = fs.Read(fileBuffer, 0, fileBuffer.Length);
                    streamToClient.Write(fileBuffer, 0, bytesRead);
                    totalBytes += bytesRead;                  // 发送了的字节数
                    status.PrintStatus(totalBytes); // 打印发送状态
                } while (bytesRead > 0);
                Console.WriteLine("Total {0} bytes sent, Done!", totalBytes);
            } catch {
                Console.WriteLine("Server has lost...");
            }

            streamToClient.Dispose();
            fs.Dispose();
            localClient.Close();
        }

        // 接收文件
        private void receiveFile(FileProtocol protocol) { }

        // 获取连接到远程的流 -- 公共方法
        private NetworkStream getStreamToClient(FileProtocol protocol, out TcpClient localClient)
{
            // 获取远程客户端的位置
            IPEndPoint endpoint = client.Client.RemoteEndPoint as IPEndPoint;
            IPAddress ip = endpoint.Address;

            // 使用新端口号，获得远程用于接收文件的端口
            endpoint = new IPEndPoint(ip, protocol.Port);
```

```
            // 连接到远程客户端
            try {
                localClient = new TcpClient();
                localClient.Connect(endpoint);
            } catch {
                Console.WriteLine("无法连接到客户端 --> {0}", endpoint);
                localClient = null;
                return null;
            }

            // 获取发送文件的流
            NetworkStream streamToClient = localClient.GetStream();
            return streamToClient;
        }

        // 随机获取一个图片名称
        private string generateFileName(string fileName) {}
}
```

服务端的 sendFile 方法和客户端的 SendFile()方法完全类似，上面的代码几乎是一次编写成功的。另外注意我将客户端使用的 SendStatus 类也拷贝到了服务端。接下来我们看下客户端。

4.2 客户端的实现

首先要注意的是客户端的 SendFile()接收的参数是文件全路径，但是在写入到协议时只获取了路径中的文件名称。这是因为服务端不需要知道文件在客户端的路径，所以协议中只写文件名；而为了使客户端的 SendFile()方法更通用，所以它接收本地文件的全路径。

客户端的 ReceiveFile()的实现也和服务端的 receiveFile()方法类似，同样，由于要保存到本地，为了避免文件名重复，我将服务端的 generateFileName()方法复制了过来。

```
public class ServerClient :IDisposable {
    // 字段略

    public ServerClient() {}

    // 发送消息到服务端
    public void SendMessage(string msg) {}

    // 发送文件 - 异步方法
    public void BeginSendFile(string filePath) {        }

    private void SendFile(object obj) { }
```

```
// 发送文件 -- 同步方法
public void SendFile(string filePath) {}

// 接收文件 -- 异步方法
public void BeginReceiveFile(string fileName) {
    ParameterizedThreadStart start =
        new ParameterizedThreadStart(ReceiveFile);
    start.BeginInvoke(fileName, null, null);
}

public void ReceiveFile(object obj) {
    string fileName = obj as string;
    ReceiveFile(fileName);
}

// 接收文件 -- 同步方法
public void ReceiveFile(string fileName) {

    IPAddress ip = IPAddress.Parse("127.0.0.1");
    TcpListener listener = new TcpListener(ip, 0);
    listener.Start();

    // 获取本地侦听的端口号
    IPEndPoint endPoint = listener.LocalEndpoint as IPEndPoint;
    int listeningPort = endPoint.Port;

    // 获取发送的协议字符串
    FileProtocol protocol =
        new FileProtocol(FileRequestMode.Receive, listeningPort, fileName);
    string pro = protocol.ToString();

    SendMessage(pro);          // 发送协议到服务端

    // 中断，等待远程连接
    TcpClient localClient = listener.AcceptTcpClient();
    Console.WriteLine("Start sending file...");
    NetworkStream stream = localClient.GetStream();

    // 获取文件保存的路劲
    string filePath =
        Environment.CurrentDirectory + "/" + generateFileName(fileName);

    // 创建文件流
```

```
            FileStream fs = new FileStream(filePath, FileMode.CreateNew, FileAccess.Write);
            byte[] fileBuffer = new byte[1024];        // 每次传 1KB
            int bytesRead;
            int totalBytes = 0;

            // 从缓存 buffer 中读入到文件流中
            do {
                bytesRead = stream.Read(buffer, 0, BufferSize);
                fs.Write(buffer, 0, bytesRead);
                totalBytes += bytesRead;
                Console.WriteLine("Receiving {0} bytes ...", totalBytes);
            } while (bytesRead > 0);

            Console.WriteLine("Total {0} bytes received, Done!", totalBytes);

            fs.Dispose();
            stream.Dispose();
            localClient.Close();
            listener.Stop();
        }

        // 随机获取一个图片名称
        private string generateFileName(string fileName) {}

        public void Dispose() {
            if (streamToServer != null)
                streamToServer.Dispose();
            if (client != null)
                client.Close();
        }
}
```

上面关键的一句就是创建协议那句，注意到将 mode 由 Send 改为了 Receive，同时传去了想要接收的服务端的文件名称。

4.3 程序测试

现在我们已经完成了所有收发文件的步骤，可以看到服务端的所有操作都是被动的，接下来我们修改客户端的 Main()程序，创建一个菜单，然后根据用户输入发送或者接收文件。

```
class Program {
    static void Main(string[] args) {

        ServerClient client = new ServerClient();
```

```
            string input;
            string path = Environment.CurrentDirectory + "/";

            do {
                Console.WriteLine("Send File:       S1 - Client01.jpg, S2 - Client02.jpg, S3 - Client03.jpg");
                Console.WriteLine("Receive File: R1 - Server01.jpg, R1 - Server02.jpg, R3- Server03.jpg");
                Console.WriteLine("Press 'Q' to exit. ");
                Console.Write("Enter your choice: ");
                input = Console.ReadLine();
                switch(input.ToUpper()){
                    case "S1":
                        client.BeginSendFile(path + "Client01.jpg");
                        break;
                    case "S2":
                        client.BeginSendFile(path + "Client02.jpg");
                        break;
                    case "S3":
                        client.BeginSendFile(path + "Client02.jpg");
                        break;
                    case "R1":
                        client.BeginReceiveFile("Server01.jpg");
                        break;
                    case "R2":
                        client.BeginReceiveFile("Server01.jpg");
                        break;
                    case "R3":
                        client.BeginReceiveFile("Server01.jpg");
                        break;
                }
            } while (input.ToUpper() != "Q");
            client.Dispose();
        }
}
```

由于这是一个控制台应用程序，并且采用了异步操作，所以这个菜单的出现顺序有点混乱。我这里描述起来比较困难，你将代码下载下来后运行一下就知道了:-)

程序的运行结果和上一节类似，这里我就不再贴图了。接下来是本系列的最后一篇，将发送字符串与传输文件的功能结合起来，创建一个可以发送消息并能收发文件的聊天程序，至于语音聊天嘛...等我学习了再告诉你 >_<、

源码下载：http://www.tracefact.net/SourceCode/Network-Part5.rar

3、Socket 处理HTTP请求响应

3.1、版权声明

文章出处：http://www.cnblogs.com/and/archive/2010/05/19/1739482.html
文章作者：贤达

3.2、内容详情

套接字（socket）是通信的基石，是支持 TCP/IP 协议的网络通信的基本操作单元。

它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的 IP 地址，本地进程的协议端口，远地主机的 IP 地址，远地进程的协议端口。套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

1．HTTP 请求格式：

```
<request line>
<headers>
<blank line>
[<request-body>]
```

2.了解 Socket，TCP,HTTP,直接的关系

HTTP 协议(Hypertext Transfer Protocol)，是 Web 联网的基础，也是手机联网常用的协议之一，HTTP 协议是建立在 TCP 协议之上的一种应用。创建 Socket 连接时，可以指定使用的传输层协议，Socket 可以支持不同的传输层协议（TCP 或 UDP），当使用 TCP 协议进行连接时，该 Socket 连接就是一个 TCP 连接。

.NetFrameWork 为 Socket 通讯提供了 System.Net.Socket 命名空间，在这个命名空间里面有以下几个常用的重要类分别是:

- Socket 类这个低层的类用于管理连接，WebRequest，TcpClient 和 UdpClient 在内部使用这个类。
- NetworkStream 类这个类是从 Stream 派生出来的，它表示来自网络的数据流
- TcpClient 类允许创建和使用 TCP 连接
- TcpListener 类允许监听传入的 TCP 连接请求
- UdpClient 类用于 UDP 客户创建连接(UDP 是另外一种 TCP 协议，但没有得到广泛的使用，主要用于本地网络)
- IPAddress 类 提供网际协议 (IP) 地址。

- IPEndPoint 继承 EndPoint 将网络端点表示为 IP 地址和端口号。

简单的 Http 请求服务处理类

```
class MyWebServer
{
    private readonly int port =
Convert.ToInt32(ConfigurationManager.AppSettings["port"].ToString());
    private readonly string host = ConfigurationManager.AppSettings["host"];
    private readonly string sMyWebServerRoot = ConfigurationManager.AppSettings["dir"];
    private TcpListener tcplistener = null;
    public MyWebServer()
    {
        try
        {
            ///创建终结点（EndPoint）
            IPAddress ip = IPAddress.Parse(host);//把 ip 地址字符串转换为 IPAddress 类型的实例
            //TcpListener 类对象，监听端口
            tcplistener = new TcpListener(ip, port);
            tcplistener.Start();
            Console.WriteLine("Web Server Running... Press ^C to Stop...");
            //同时启动一个兼听进程 ''StartListen''
            Thread th = new Thread(new ThreadStart(StartListen));
            th.Start();
        }
        catch (Exception e)
        {
            Console.WriteLine("监听端口时发生错误 :" + e.ToString());
        }
    }

    /// <summary>
    ///设置请求的标头
    ///</summary>
    ///<param name="sHttpVersion"></param>
    ///<param name="sMIMEHeader"></param>
    ///<param name="iTotBytes"></param>
    ///<param name="sStatusCode"></param>
    ///<param name="mySocket"></param>
    public void SendHeader(string sHttpVersion, string sMIMEHeader, int iTotBytes, string
sStatusCode, ref Socket mySocket)
    {
        String sBuffer = "";
```

```
        if (sMIMEHeader.Length == 0)
        {
            sMIMEHeader = "text/html"; // 默认 text/html
        }
        sBuffer = sBuffer + sHttpVersion + sStatusCode + "\r\n";
        sBuffer = sBuffer + "Server: cx1193719-b\r\n";
        sBuffer = sBuffer + "Content-Type: " + sMIMEHeader + "\r\n";
        sBuffer = sBuffer + "Accept-Ranges: bytes\r\n";
        sBuffer = sBuffer + "Content-Length: " + iTotBytes + "\r\n\r\n";
        Byte[] bSendData = Encoding.ASCII.GetBytes(sBuffer);
        SendToBrowser(bSendData, ref mySocket);
        Console.WriteLine("Total Bytes : " + iTotBytes.ToString());
    }

    public void SendToBrowser(String sData, ref Socket mySocket)
    {
        SendToBrowser(Encoding.ASCII.GetBytes(sData), ref mySocket);
    }
    /// <summary>
    /// 负责向客户端发起数据
    /// </summary>
    /// <param name="bSendData">字节数组</param>
    /// <param name="mySocket">Soket 对象！</param>
    public void SendToBrowser(Byte[] bSendData, ref Socket mySocket)
    {
        int numBytes = 0;
        try
        {
            if (mySocket.Connected)
            {
                if ((numBytes = mySocket.Send(bSendData, bSendData.Length, 0)) == -1)
                    Console.WriteLine("Socket Error cannot Send Packet");
                else
                {
                    Console.WriteLine("No. of bytes send {0}", numBytes);
                }
            }
            else
                Console.WriteLine("连接失败....");
        }
        catch (Exception e)
        {
            Console.WriteLine("发生错误 : {0} ", e);
        }
```

```
    }

    public static void Main()
    {
        MyWebServer MWS = new MyWebServer();
    }
    public void StartListen()
    {
        int iStartPos = 0;
        String sRequest;
        String sDirName;
        String sRequestedFile;
        String sErrorMessage;
        String sLocalDir;
        String sPhysicalFilePath = "";
        String sFormattedMessage = "";
        String sResponse = "";
        //进入监听循环
        while (true)
        {
            //接受新连接
            Socket mySocket = tcplistener.AcceptSocket();
            Console.WriteLine("Socket Type " + mySocket.SocketType);
            if (mySocket.Connected)
            {
                Console.WriteLine("\nClient    Connected!!\n==================\nCLient    IP
{0}\n", mySocket.RemoteEndPoint);
                //将请求转化成字节数组!
                // 为读取数据而准备缓存
                Byte[] bReceive = new Byte[1024];
                int i = mySocket.Receive(bReceive, bReceive.Length, 0);
                //转换成字符串类型
                string sBuffer = Encoding.ASCII.GetString(bReceive);
                Console.WriteLine(sBuffer);
                //只处理"get"请求类型
                if (sBuffer.Substring(0, 3) != "GET")
                {
                    Console.WriteLine("只处理 get 请求类型..");
                    mySocket.Close();
                    return;
                }
                // 查找 "HTTP" 的位置
                iStartPos = sBuffer.IndexOf("HTTP", 1);
                string sHttpVersion = sBuffer.Substring(iStartPos, 8);
```

```
                // 得到请求类型和文件目录文件名
                sRequest = sBuffer.Substring(0, iStartPos - 1);
                sRequest.Replace("\\", "/");
                //如果结尾不是文件名也不是以"/"结尾则加"/"
                if ((sRequest.IndexOf(".") < 1) && (!sRequest.EndsWith("/")))
                {
                    sRequest = sRequest + "/";
                }
                //得带请求文件名
                iStartPos = sRequest.LastIndexOf("/") + 1;
                sRequestedFile = sRequest.Substring(iStartPos);
                //得到请求文件目录
            sDirName = sRequest.Substring(sRequest.IndexOf("/"), sRequest.LastIndexOf("/") - 3);
                //获取虚拟目录物理路径
                sLocalDir = sMyWebServerRoot;
                Console.WriteLine("请求文件目录 : " + sLocalDir);
                if (sLocalDir.Length == 0)
                {
                    sErrorMessage = "<H2>Error!! Requested Directory does not exists</H2><Br>";
                    SendHeader(sHttpVersion, "", sErrorMessage.Length, " 404 Not Found",
 ref mySocket);
                    SendToBrowser(sErrorMessage, ref mySocket);
                    mySocket.Close();
                    continue;
                }
                if (sRequestedFile.Length == 0)
                {
                    // 取得请求文件名
                    sRequestedFile = "index.html";
                }
                ////////////////////////////////////////////////////////////////////
                // 取得请求文件类型（设定为 text/html）
                ////////////////////////////////////////////////////////////////////
                String sMimeType = "text/html";
                sPhysicalFilePath = sLocalDir + sRequestedFile;
                Console.WriteLine("请求文件: " + sPhysicalFilePath);
                if (File.Exists(sPhysicalFilePath) == false)
                {
                    sErrorMessage = "<H2>404 Error! File Does Not Exists...</H2>";
                    SendHeader(sHttpVersion, "", sErrorMessage.Length, " 404 Not Found",
ref mySocket);
                    SendToBrowser(sErrorMessage, ref mySocket);
                    Console.WriteLine(sFormattedMessage);
                }
```

```
                else
                {
                    int iTotBytes = 0;
                    sResponse = "";
                    FileStream fs = new FileStream(sPhysicalFilePath, FileMode.Open,
FileAccess.Read, FileShare.Read);
                    BinaryReader reader = new BinaryReader(fs);
                    byte[] bytes = new byte[fs.Length];
                    int read;
                    while ((read = reader.Read(bytes, 0, bytes.Length)) != 0)
                    {
                        sResponse = sResponse + Encoding.ASCII.GetString(bytes, 0, read);
                        iTotBytes = iTotBytes + read;
                    }
                    reader.Close();
                    fs.Close();
                    SendHeader(sHttpVersion, sMimeType, iTotBytes, " 200 OK", ref mySocket);
                    SendToBrowser(bytes, ref mySocket);
                    //mySocket.Send(bytes, bytes.Length,0);
                }
                mySocket.Close();
            }
        }
    }
}
```

客户端请求类

```
static void Main(string[] args)
{
    int port = 1280;
    string host = "127.0.0.1";
    IPAddress ip = IPAddress.Parse(host);
    //  string hotByname = "www.baidu.com";
    // IPHostEntry gist = Dns.GetHostByName(hotByname);
    //IPAddress ip = gist.AddressList[0];
    /**/
    ///创建终结点 EndPoint
    //IPAddress ipp = new IPAddress("127.0.0.1");
    IPEndPoint ipe = new IPEndPoint(ip, port);//把 ip 和端口转化为 IPEndpoint 实例
    /**/
    ///创建 socket 并连接到服务器
    Socket c = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);//
创建 Socket
```

```
    Console.WriteLine("Conneting…");
    c.Connect(ipe);//连接到服务器
    ///向服务器发送信息
    //{GET /index.php HTTP/1.0Content-Type: application/x-www-form-urlencoded
    StringBuilder buf = new StringBuilder();
    buf.Append("GET ").Append("/index.php").Append(" HTTP/1.0\r\n");
    buf.Append("Content-Type: application/x-www-form-urlencoded\r\n");
    buf.Append("\r\n");
    byte[] bs = Encoding.ASCII.GetBytes(buf.ToString());//把字符串编码为字节
    Console.WriteLine("Send Message");
    c.Send(bs);//发送信息
    /**/
    ///接受从服务器返回的信息
    string recvStr = "";
    byte[] recvBytes = new byte[1024];
    int bytes;

    do
    {
        bytes = c.Receive(recvBytes, recvBytes.Length, 0);//从服务器端接受返回信息
        recvStr += Encoding.Default.GetString(recvBytes, 0, bytes);
        Console.WriteLine("client get message:{0}", recvStr);//显示服务器返回信息
    } while (bytes!=0);
    /**/
    ///一定记着用完 socket 后要关闭
    c.Close();
    Console.ReadLine();
}
```

以上有一个问题注意：

```
GET /xxx.xxx HTTP/1.1
Host: xxx
Connection: Close    //没有关闭请求！
```

只有加入这最后一行，对方发送完数据才会关闭连接，只有对方关闭连接，我们这边才能 recv 到一个 0，从而根据 recv 返回的 0 判断接收数据完毕，然后我们就可以退出拼接数据包的循环。

4、浅谈C# Socket编程及多线程

4.1、版权声明

文章出处：http://pcajax.iteye.com/blog/1067854
文章作者：pcajax

4.2、内容详情

去年暑假学习了几个月 asp.net 最后几个星期弄了下 C# socket .也算知道了个一知半解了，好久没动 C#了，虽然这语言高级的让我对他没兴趣，不过还是回忆回忆，忘了一干二净就不好了.

C# Socket:

建议初学 C# socket 的菜鸟朋友不要使用 TcpListenner, TcpClient 这些 MS 封装好的类库,这些封装好的类用起来的确方便，但你用完了你又学到了什么了？那该用什么了，只用 Socket 这一个类. 不错,这样会麻烦点的,

但是，在 C#里面，就连 Socket, MS 都进行了一翻封装,使得 Socket 使用起来也是十分的简单，我刚学的时候写过一个很菜的 TCP 聊天程序，两人对聊的. 大家可以去尝试下一 S 多 C 的聊天程序,TCP 会了可以去做个 UDP 的.

UDP 会了,学学 SMTP, 哎, SMTP 也是封装的太厉害，都成傻瓜式的了，然后大家可以看下 MultiThread,也就是多线程，这些都差不多了，可以取尝试写个 Proxy. 我当时就是这样学的，呵呵，不过我只是个菜鸟，现在搞 asm/c/C++去了,就把这些忘了差不多.

首先必须包含的两个命名空间:

```
Using System.Net;
Using System.Net.Sockets;
```

几个常用的类:(这些东西,查下 MSDN 就很详细了)

IPHostEntry, Dns,IPAddress,IPEndPoint,还有最重要的 Socket

IPEndPoint: 这个是网络终结点,很好理解,就是网络上一个固定的地址:一个 IP 与一个端口的组合.

下面我还是以我以前写的一个很简单的聊天程序做示例吧, (很短代码的)

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;    //这个和上面的是使用 Socket 必须的.
using System.IO;
using System.Threading;       //这个是使用多线程必需的.

namespace OnlySocket
{
    public partial class Form1 : Form            //partial 表示这块代码只是 Form1 类的部分, Form1
类继承自 Form 类
    {
        public Form1()
        {
            InitializeComponent();     //构造函数, 初始化容器.
        }
        Socket sock;           //定义一个 Socket 类的对象 (默认为 protected)
        Thread th;              //定义一个 Thread 类的对象
        //

        public static IPAddress GetServerIP()         //静态函数, 无需实例化即可调用.
        {
            IPHostEntry ieh = Dns.GetHostByName(Dns.GetHostName());
//不多说了, Dns 类的两个静态函数

            //或用 DNS.Resolve()代替 GetHostName()
            return ieh.AddressList[0];
//返回 Address 类的一个实例. 这里 AddressList 是数组并不奇怪,一个 Server 有 N 个 IP 都有可能
        }


        private void BeginListen()                //Socket 监听函数, 等下作为创建新线程的参数
        {
            IPAddress serverIp = GetServerIP();
//调用本类静态函数 GetServerIP 得到本机 IPAddress.
            IPEndPoint iep = new IPEndPoint(serverIp, Convert.ToInt32(tbPort.Text));
```

```
 //本地终结点
            sock     =     new     Socket(AddressFamily.InterNetwork,     SocketType.Stream,
ProtocolType.Tcp);    //实例化内成员 sock
            Byte[] byteMessage = new Byte[100];
 //存放消息的字节数组缓冲区，注意数组表示方法,和 C 不同的.
            this.lbIep.Text = iep.ToString();
            sock.Bind(iep);                          //Socket 类的一个重要函数，绑定一个 IP,
            while (true)                        //这里弄了个死循环来监听端口，有人会问死循环了，
那程序不卡住了，注意这只是个类，这里还没有 main 函数呢.
            {
                try
                {
                    sock.Listen(5);
//好了,sock 绑定了本地终结点就可以开始监听了,5 表示最大连接数为 5
                    Socket newSock = sock.Accept();
//这里又有 Socket 类的一个重要的方法:Accept，该方法接受来自外面的 Socket 连接请求，并返回一个
Socket 套接字，这个套接字就开始处理这一个 client 与 Server 之间的对话
                    newSock.Receive(byteMessage); //接受 client 发送过来的数据保存到缓冲区.
                    string msg = "From [" + newSock.RemoteEndPoint.ToString() + "]:" +
System.Text.Encoding.UTF8.GetString(byteMessage) + "\n";
//GetString()函数将 byte 数组转换为 string 类型.
                    rtbTalk.AppendText(msg + "\n");         //显示在文本控件里
                }
                catch (SocketException se)              //捕捉异常,
                {
                    lbState.Text = se.ToString();       //将其显示出来，在此亦可以自定义错误.
                }
            }
        }

        private void btConnect_Click(object sender, EventArgs e)
//连接按钮触发的事件：连接 Server
        {
            btConnect.Enabled = false;
            btStopConnect.Enabled = true;
            try
            {
                th = new Thread(new ThreadStart(BeginListen));
 //创建一个新的线程专门用于处理监听,这句话可以分开写的,比如：ThreadStart ts=new
ThreadStart(BeginListen); th=new Thread (ts); 不过要注意，ThreadStart 的构造函数的参数一定要
是无参数的函数. 在此函数名其实就是其指针，这里是委托吗?
                th.Start();                             //启动线程
                lbState.Text = "Listenning...";
            }
```

```
            catch (SocketException se)            //处理异常
            {
                MessageBox.Show(se.Message, "出现问题", MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
            catch (ArgumentNullException ae)    //参数为空异常
            {
                lbState.Text = "参数错误";
                MessageBox.Show(ae.Message, "错误", MessageBoxButtons.OK, MessageBoxIcon.Warning);
            }

        }

        private void btStopConnect_Click(object sender, EventArgs e)  //停止监听
        {
            btStopConnect.Enabled = false;
            btConnect.Enabled = true;
            sock.Close();                        //关闭套接字
            th.Abort();                            //终止监听线程

            lbState.Text = "Listenning stopped";
        }

        private void btExit_Click(object sender, EventArgs e)
        {
            sock.Close();
            th.Abort();
            Dispose();              //清理资源,就是释放内存
            this.Close();           //关闭对话框, 退出程序
        }

        private void btSend_Click(object sender, EventArgs e)
        {
            try
            {
                IPAddress clientIp = IPAddress.Parse(tbTargetIp.Text);
 //类 IPAddress 的静态函数 Parse() :将 Text 转化为 IPAddress 的一个实例.
                int clientPort = Convert.ToInt32(tbPort.Text);
//C#的这些转化函数很方便的,不像 C++那样麻烦
                IPEndPoint clientIep = new IPEndPoint(clientIp, clientPort);
//这里用 client 表示不是很好....,
                Byte[] byte_Message;
                Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
实例化的时候还有很多参数的, 这个是Tcp的. Tcp的SocketType是Stream:数据流, 如果协议类型是UDP,
则是数据包传送, QQ 就是用的 UDP.
```

```
                socket.Connect(clientIep);
//Socket 的又一个函数 Connect(IPEndPoint) .连接远程套接字
                byte_Message = System.Text.Encoding.UTF8.GetBytes(rtbWords.Text);
//发现 UTF8 可支持中文,就用之
                socket.Send(byte_Message);
                rtbTalk.AppendText("\n"+"My words:" + rtbWords.Text + "\n");
                socket.Shutdown(SocketShutdown.Both);
                socket.Close();
            }
            catch (ArgumentNullException ae)
            {
                MessageBox.Show(ae.Message,"参数为",MessageBoxButtons.OKCancel,MessageBoxIcon.Information);
            }
            catch (SocketException se)
            {
                MessageBox.Show(se.Message, "出现问题", MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
        }

    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace OnlySocket
{
    static class Program
    {
        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main()        //这儿才是 main 函数
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

写了半天了，够累的了，虽然都是很基础的东西，我自己写的时候也复习了一边 ，呵呵.

其实多线程我自己也不是很熟练，记得去年暑假写过一个多线程扫描器，也不知为啥，线程开到50以上就异常，很郁闷的. 其实当时我就是用的new Thread=Thread(new ThreadStart(Fun))实现的，方法感觉很笨拙,呵呵.

大致代码好像是这样的吧:

先写个Scan类:

```
public class Scan
{
    try{ public Scan(){   ...Init...   }
    public void Scan{ ..task 循环扫描... } //task 结构体里面有 IP，端口，是否已扫描标记 fLag}
    catch{}
}
```

然后主函数里面可以这样搞:

```
Scan[] scanner = new Scan[XX]
Thread[] thread = new Thread[XX];
            for (int i = 0; i < XX;i++)
            {
                scanner[i] = new Scan(this, i);
                thread[i] = new Thread(new ThreadStart(scanner[i].StartScan));
                thread[i].Start();

            }
```

其实这样就可以简单的实现多线程了.

5、Socket编程笔记

5.1、版权声明

文章出处：http://www.cnblogs.com/stg609/archive/2008/11/15/1333889.html
文章作者：stg609

5.2、内容详情

到这个题目，是不是很眼熟？在博客园里搜下，保证会发现关于这个东东的文章实在是太多了~~~真得是没有写得必要，而且我也有点懒得去琢磨字句。（看到这，肯定得来个转折的了，不然就看不到下文了，不是吗）但是，为了自己下一篇要写的文章做参考，还是有必要先补充一下 socket 基础知识。

注意：如果你已经接触过 socket，那就没什么必要耽误时间看下去了。另外，如果发现其中任何错误，欢迎直接指出。

1.按惯例先来介绍下 socket

Windows 中的很多东西都是从 Unix 领域借鉴过来的，Socket 也是一样。在 Unix 中，socket 代表了一种文件描述符（在 Unix 中一切都是以文件为单位），而这里这个描述符则是用于描述网络访问的。什么意思呢？就是程序员可以通过 socket 来发送和接收网络上的数据。你也可以理解成是一个 API。有了它，你就不用直接去操作网卡了，而是通过这个接口，这样就省了很多复杂的操作。

在 C#中，MS 为我们提供了 System.Net.Sockets 命名空间，里面包含了 Socket 类。

2.有了 socket，那就可以用它来访问网络了

不过你不要高兴得太早，要想访问网络，还得有些基本的条件（和编程无关的我就不提了）：a. 要确定本机的 IP 和端口，socket 只有与某一 IP 和端口绑定，才能发挥强大的威力。b. 得有协议吧（否则谁认得你这发送到网络的是什么呀）。想要复杂的，我们可以自己来定协议。但是这个就不在这篇里提了，我这里介绍两种大家最熟悉不过的协议：TCP & UDP。（别说你不知道，不然...不然...我不告诉你）

如果具备了基本的条件，就可以开始用它们访问网络了。来看看步骤吧：

a. 建立一个套接字

b. 绑定本机的 IP 和端口

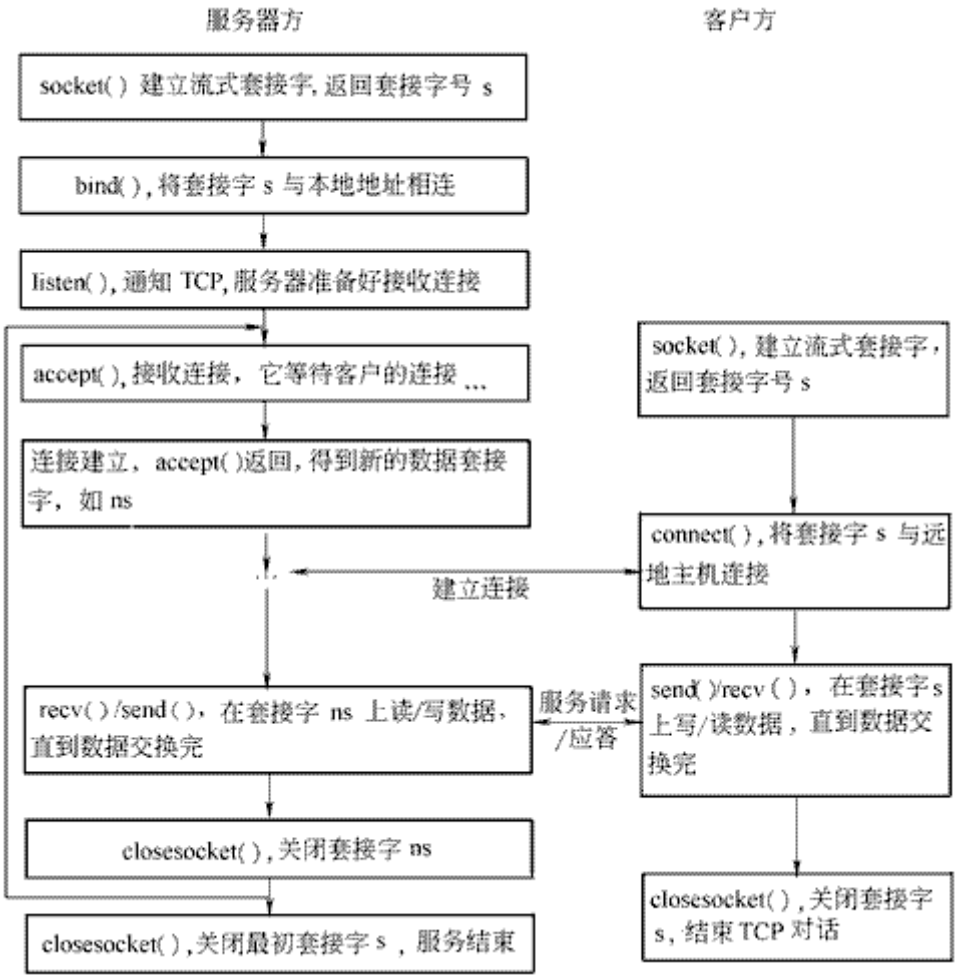
c. 如果是 TCP，因为是面向连接的，所以要利用 ListenO（）方法来监听网络上是否有人给自己发东西;如果是 UDP，因为是无连接的，所以来者不拒。

d. TCP 情况下，如果监听到一个连接，就可以使用 accept 来接收这个连接，然后就可以利用 Send/Receive 来执行操作了。而 UDP，则不需要 accept， 直接使用

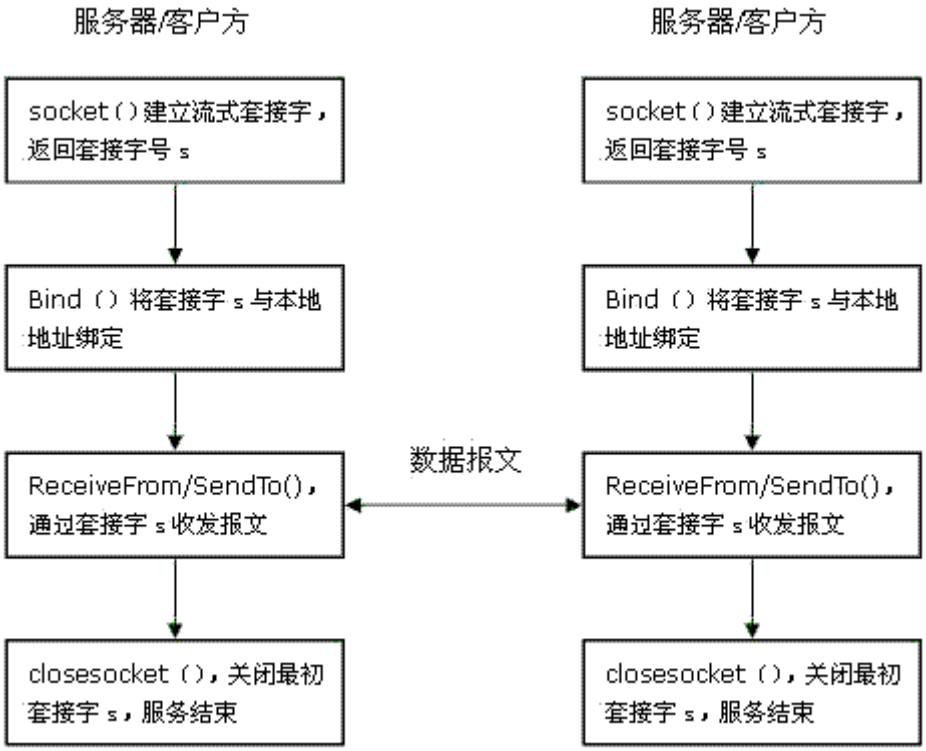
SendTo/ReceiveFrom 来执行操作。（看清楚哦，和 TCP 的执行方法有区别，因为 UDP 不需要建立连接，所以在发送前并不知道对方的 IP 和端口，因此需要指定一个发送的节点才能进行正常的发送和接收）

e. 如果你不想继续发送和接收了，就不要浪费资源了。能 close 的就 close 吧。

如果看了上面文字，你还不清楚的话，就来看看图好了：



面向连接的套接字系统调用时序



无连接的套接字系统调用时序

3.开始动手敲~~代码（简单的代码）

首先我们来写个面向连接的

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace tcpserver
{
    /// <summary>
    /// Class1 的摘要说明。
    /// </summary>
    class server
    {
        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
```

```
            //
            // TODO: 在此处添加代码以启动应用程序
            //
            int recv;//用于表示客户端发送的信息长度
            byte[] data = new byte[1024];//用于缓存客户端所发送的信息,通过 socket 传递的信息必须为字节数组
            IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);//本机预使用的 IP 和端口
            Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
            newsock.Bind(ipep);//绑定
            newsock.Listen(10);//监听
            Console.WriteLine("waiting for a client");
            Socket client = newsock.Accept();//当有可用的客户端连接尝试时执行，并返回一个新的 socket,用于与客户端之间的通信
            IPEndPoint clientip = (IPEndPoint)client.RemoteEndPoint;
            Console.WriteLine("connect with client:" + clientip.Address + " at port:" + clientip.Port);
            string welcome = "welcome here!";
            data = Encoding.ASCII.GetBytes(welcome);
            client.Send(data, data.Length, SocketFlags.None);//发送信息
            while (true)
            {//用死循环来不断的从客户端获取信息
                data = new byte[1024];
                recv = client.Receive(data);
                Console.WriteLine("recv=" + recv);
                if (recv == 0)//当信息长度为 0，说明客户端连接断开
                    break;
                Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
                client.Send(data, recv, SocketFlags.None);
            }
            Console.WriteLine("Disconnected from" + clientip.Address);
            client.Close();
            newsock.Close();

        }
    }
}
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace tcpclient
```

```
{
    /// <summary>
    /// Class1 的摘要说明。
    /// </summary>
    class client
    {
        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: 在此处添加代码以启动应用程序
            //
            byte[] data = new byte[1024];
            Socket  newclient  =  new  Socket(AddressFamily.InterNetwork,  SocketType.Stream,
ProtocolType.Tcp);
            Console.Write("please input the server ip:");
            string ipadd = Console.ReadLine();
            Console.WriteLine();
            Console.Write("please input the server port:");
            int port = Convert.ToInt32(Console.ReadLine());
            IPEndPoint ie = new IPEndPoint(IPAddress.Parse(ipadd), port);//服务器的 IP 和端口
            try
            {
                //因为客户端只是用来向特定的服务器发送信息，所以不需要绑定本机的 IP 和端口。不
需要监听。
                newclient.Connect(ie);
            }
            catch (SocketException e)
            {
                Console.WriteLine("unable to connect to server");
                Console.WriteLine(e.ToString());
                return;
            }
            int recv = newclient.Receive(data);
            string stringdata = Encoding.ASCII.GetString(data, 0, recv);
            Console.WriteLine(stringdata);
            while (true)
            {
                string input = Console.ReadLine();
                if (input == "exit")
                    break;
```

```
                newclient.Send(Encoding.ASCII.GetBytes(input));
                data = new byte[1024];
                recv = newclient.Receive(data);
                stringdata = Encoding.ASCII.GetString(data, 0, recv);
                Console.WriteLine(stringdata);
            }
            Console.WriteLine("disconnect from sercer");
            newclient.Shutdown(SocketShutdown.Both);
            newclient.Close();

        }
    }
}
```

下面在给出无连接的(实在是太懒了，下面这个是直接复制别人的)

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace SimpleUdpSrvr
{
    class Program
    {
        static void Main(string[] args)
        {
            int recv;
            byte[] data = new byte[1024];
            IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);//定义一网络端点
            Socket newsock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);//定义一个 Socket
            newsock.Bind(ipep);//Socket 与本地的一个终结点相关联
            Console.WriteLine("Waiting for a client..");

            IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);//定义要发送的计算机的地址
            EndPoint Remote = (EndPoint)(sender);//
            recv = newsock.ReceiveFrom(data, ref Remote);//接受数据
            Console.WriteLine("Message received from{0}:", Remote.ToString());
            Console.WriteLine(Encoding.ASCII.GetBytes(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);
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace SimpleUdpClient
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] data = new byte[1024];//定义一个数组用来做数据的缓冲区
            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);//将数据发送到指定的终结
点

            IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
            EndPoint Remote = (EndPoint)sender;
            data = new byte[1024];
            int recv = server.ReceiveFrom(data, ref Remote);//接受来自服务器的数据

            Console.WriteLine("Message received from{0}:", Remote.ToString());
            Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
            while (true)//读取数据
            {
                input = Console.ReadLine();//从键盘读取数据
                if (input == "text")//结束标记
                {
```

```
                    break;
                }
                server.SendTo(Encoding.ASCII.GetBytes(input), Remote);//将数据发送到指定的终
结点 Remote
                data = new byte[1024];
                recv = server.ReceiveFrom(data, ref Remote);//从 Remote 接受数据
                stringData = Encoding.ASCII.GetString(data, 0, recv);
                Console.WriteLine(stringData);
            }
            Console.WriteLine("Stopping client");
            server.Close();
        }
    }
}
```

上面的示例只是简单的应用了 socket 来实现通信，你也可以实现异步 socket、IP 组播 等等。

MS 还为我们提供了几个助手类：TcpClient 类、TcpListener 类、UDPClient 类。这几个类简化了一些操作，所以你也可以利用这几类来写上面的代码，但我个人还是比较习惯直接用 socket 来写。

既然快写完了，那我就再多啰嗦几句。在需要即时响应的软件中，我个人更倾向使用 UDP 来实现通信，因为相比 TCP 来说，UDP 占用更少的资源，且响应速度快，延时低。至于 UDP 的可靠性，则可以通过在应用层加以控制来满足。当然如果可靠性要求高的环境下，还是建议使用 TCP。

6、深入探析C# Socket

6.1、版权声明

文章出处：http://www.cnblogs.com/tianzhiliang/archive/2010/09/08/1821623.html
文章作者：田志良

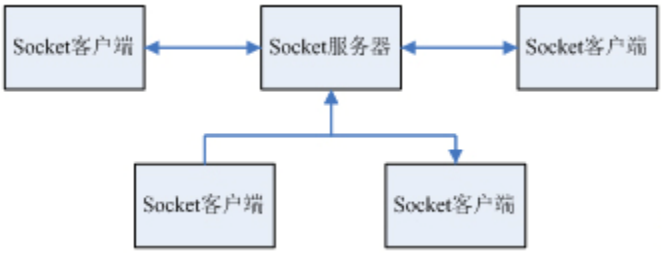
6.2、内容详情

最近浏览了几篇有关 Socket 发送消息的文章，发现大家对 Socket Send 方法理解有所偏差，现将自己在开发过程中对 Socket 的领悟写出来，以供大家参考。

（一）架构

基于 TCP 协议的 Socket 通信，架构类似于 B/S 架构，一个 Socket 通信服务器，多个 Socket 通信客户端。Socket 通信服务器启动时，会建立一个侦听 Socket，侦听 Socket 将侦听到的 Socket 连接传给接受 Socket，然后由接受 Socket 完成接受、发送消息，当 Socket 存在异常时，断开连接。在实际开发项目中，往往要求 Socket 通信服务器能提供高效、稳定的服务，一般会用到以下技术：双工通信、完成端口、SAEA、池、多线程、异步等。特别是池，用的比较多，池一般包括一下几种：

1）Buffer 池，用于集中管控 Socket 缓冲区，防止内存碎片。
2）SAEA 池，用于集中管控 Socket，重复利用 Socket。
3）SQL 池，用于分离网络服务层与数据访问层（SQL 的执行效率远远低于网络层执行效率）。
4）线程池，用于从线程池中调用空闲线程执行业务逻辑，进一步提高网络层运行效率。



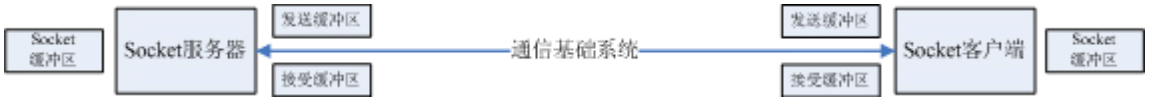
（二）Send

主服务器接受 Socket 为一端口，客户端 Socket 为一端口，这两个端口通过 TCP 协议建立连接，通信基础系统负责管理此连接，它有两个功能：

1）发送消息

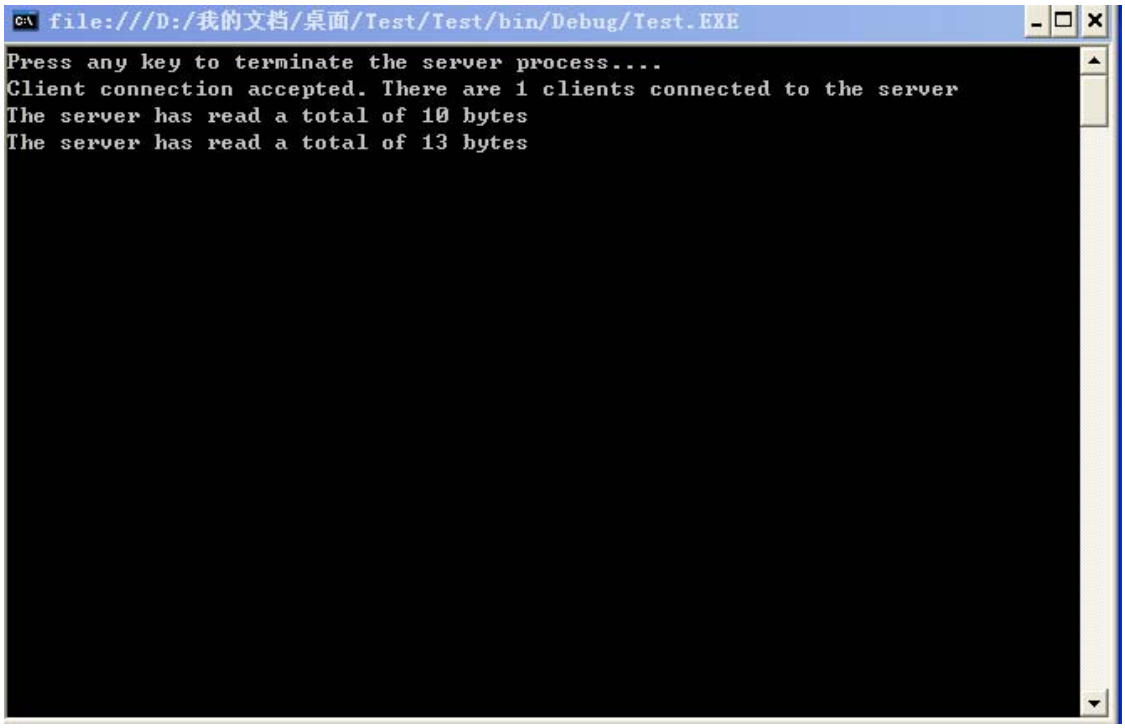
2）接受消息

Socket 的 Send 方法，并非大家想象中的从一个端口发送消息到另一个端口，它仅仅是拷贝数据到基础系统的发送缓冲区，然后由基础系统将发送缓冲区的数据到连接的另一端口。值得一说的是，这里的拷贝数据与异步发送消息的拷贝是不一样的，同步发送的拷贝，是直接拷贝数据到基础系统缓冲区，拷贝完成后返回，在拷贝的过程中，执行线程会 IO 等待，此种拷贝与 Socket 自带的 Buffer 空间无关，但异步发送消息的拷贝，是将 Socket 自带的 Buffer 空间内的所有数据，拷贝到基础系统发送缓冲区，并立即返回，执行线程无需 IO 等待，所以异步发送在发送前必须执行 SetBuffer 方法，拷贝完成后，会触发你自定义回调函数 ProcessSend，在 ProcessSend 方法中，调用 SetBuffer 方法，重新初始化 Buffer 空间。



口说无凭，下面给个例子：

服务器端：



客户端：

解释：

客户端第一次发送数据：1234567890。 客户端第一个接受数据：1234567890，该数据由服务端用 Send 同步方法发送返回。 客户端第二个接受数据：1234567890，该数据由服务端用 Send 异步方法发送返回。

、

以上似乎没什么异常，好，接下来，我只发送 abc。

客户端第一个接受数据：abc，理所当然，没什么问题。 客户端第二个接受数据：abc4567890！为什么呢？应该是 abc 才对呀！

好，现在为大家解释一下：

异步发送是将其 Buffer 空间中所有数据拷贝到基础系统发送缓冲区，第一次拷贝 1234567890 到发送缓冲区，所以收到 1234567890，第二次拷贝 abc 到发送缓冲区，替换了先前的 123，所以收到 abc4567890，大家明白的？

源代码：

```
using System.Collections.Generic;
```

```
using System.Net.Sockets;
// This class creates a single large buffer which can be divided up
// and assigned to SocketAsyncEventArgs objects for use with each
// socket I/O operation.
// This enables bufffers to be easily reused and guards against
// fragmenting heap memory.
//
// The operations exposed on the BufferManager class are not thread safe.
class BufferManager{
    int m_numBytes;
    // the total number of bytes controlled by the buffer pool
    byte[] m_buffer;
    // the underlying byte array maintained by the Buffer Manager
    Stack<int> m_freeIndexPool;
    //     int m_currentIndex;
    int m_bufferSize;
    public BufferManager(int totalBytes, int bufferSize)
    {
        m_numBytes = totalBytes;
        m_currentIndex = 0;
        m_bufferSize = bufferSize;
        m_freeIndexPool = new Stack<int>();
    }
    // Allocates buffer space used by the buffer pool
    public void InitBuffer()
    {
        // create one big large buffer and divide that
        // out to each SocketAsyncEventArg object
        m_buffer = new byte[m_numBytes];
    }
    // Assigns a buffer from the buffer pool to the
    // specified SocketAsyncEventArgs object
    //
    // <returns>true if the buffer was successfully set, else false</returns>
    public bool SetBuffer(SocketAsyncEventArgs args)
    {
        if (m_freeIndexPool.Count > 0)
        {
            args.SetBuffer(m_buffer, m_freeIndexPool.Pop(), m_bufferSize);
        }
        else
        {
            if ((m_numBytes - m_bufferSize) < m_currentIndex)
            {
```

```
                return false;
            }
            args.SetBuffer(m_buffer, m_currentIndex, m_bufferSize);
            m_currentIndex += m_bufferSize;
        }
        return true;
    }
    // Removes the buffer from a SocketAsyncEventArg object.
    // This frees the buffer back to the buffer pool
    public void FreeBuffer(SocketAsyncEventArgs args)
    {
        m_freeIndexPool.Push(args.Offset);
        args.SetBuffer(null, 0, 0);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Net.Sockets;
// Represents a collection of reusable SocketAsyncEventArgs objects.
class SocketAsyncEventArgsPool
{
    Stack<SocketAsyncEventArgs> m_pool;
    // Initializes the object pool to the specified size
    //
    // The "capacity" parameter is the maximum number of
    // SocketAsyncEventArgs objects the pool can hold
    public SocketAsyncEventArgsPool(int capacity)
    {
        m_pool = new Stack<SocketAsyncEventArgs>(capacity);
    }
    // Add a SocketAsyncEventArg instance to the pool
    //
    //The "item" parameter is the SocketAsyncEventArgs instance
    // to add to the pool
    public void Push(SocketAsyncEventArgs item)
    {
        if (item == null) {
            throw new ArgumentNullException("Items added to a SocketAsyncEventArgsPool cannot
be null");
        }
        lock (m_pool)
        {
```

```
            m_pool.Push(item);
        }
    }
    // Removes a SocketAsyncEventArgs instance from the pool
    // and returns the object removed from the pool
    public SocketAsyncEventArgs Pop()
    {
        lock (m_pool)
        {
            return m_pool.Pop();
        }
    }
    // The number of SocketAsyncEventArgs instances in the pool
    public int Count
    {
        get { return m_pool.Count;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;

class AsyncUserToken
{
    public Socket Socket;
}
```

```
using System;
using System.Threading;
using System.Net.Sockets;
using System.Net;
using System.Text;
// Implements the connection logic for the socket server.
// After accepting a connection, all data read from the client
// is sent back to the client. The read and echo back to the client pattern
// is continued until the client disconnects.
class Server{
```

```
    private int m_numConnections;
    // the maximum number of connections the sample is designed to handle simultaneously
    private int m_receiveBufferSize;// buffer size to use for each socket I/O operation
    BufferManager m_bufferManager;  // represents a large reusable set of buffers for all socket
operations
    const int opsToPreAlloc = 2;    // read, write (don't alloc buffer space for accepts)
    Socket listenSocket;            // the socket used to listen for incoming connection requests
    // pool of reusable SocketAsyncEventArgs objects for write, read and accept socket operations
    SocketAsyncEventArgsPool m_readWritePool;
    int m_totalBytesRead;
    // counter of the total # bytes received by the server
    int m_numConnectedSockets;      // the total number of clients connected to the server
    Semaphore m_maxNumberAcceptedClients;    // Create an uninitialized server instance.
    // To start the server listening for connection requests
    // call the Init method followed by Start method
    //
    // <param name="numConnections">the maximum number of connections the sample is designed to
handle simultaneously</param>
    // <param name="receiveBufferSize">buffer size to use for each socket I/O operation</param>
    public Server(int numConnections, int receiveBufferSize)
    {
        m_totalBytesRead = 0;
        m_numConnectedSockets = 0;
        m_numConnections = numConnections;
        m_receiveBufferSize = receiveBufferSize;
        // allocate buffers such that the maximum number of sockets can have one outstanding read and
        //write posted to the socket simultaneously
        m_bufferManager = new BufferManager(receiveBufferSize * numConnections * opsToPreAlloc,
receiveBufferSize);
        m_readWritePool = new SocketAsyncEventArgsPool(numConnections);
        m_maxNumberAcceptedClients = new Semaphore(numConnections, numConnections);
    }
    // Initializes the server by preallocating reusable buffers and
    // context objects.  These objects do not need to be preallocated
    // or reused, but it is done this way to illustrate how the API can
    // easily be used to create reusable objects to increase server performance.
    //
    public void Init()
    {
        // Allocates one large byte buffer which all I/O operations use a piece of.  This gaurds
        // against memory fragmentation
        m_bufferManager.InitBuffer();
        // preallocate pool of SocketAsyncEventArgs objects
        SocketAsyncEventArgs readWriteEventArg;
```

```
        for (int i = 0; i < m_numConnections; i++)
        {
            //Pre-allocate a set of reusable SocketAsyncEventArgs
            readWriteEventArg = new SocketAsyncEventArgs();
            readWriteEventArg.Completed += new
EventHandler<SocketAsyncEventArgs>(IO_Completed);
            readWriteEventArg.UserToken = new AsyncUserToken();
            // assign a byte buffer from the buffer pool to the SocketAsyncEventArg object
            m_bufferManager.SetBuffer(readWriteEventArg);
            // add SocketAsyncEventArg to the pool
            m_readWritePool.Push(readWriteEventArg);
        }
    }
    // Starts the server such that it is listening for
    // incoming connection requests.
    //
    // <param name="localEndPoint">The endpoint which the server will listening
    // for connection requests on</param>
    public void Start(IPEndPoint localEndPoint)
    {
        // create the socket which listens for incoming connections
        listenSocket = new Socket(localEndPoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
        listenSocket.Bind(localEndPoint);
        // start the server with a listen backlog of 100 connections
        listenSocket.Listen(100);        // post accepts on the listening socket
        StartAccept(null);             //Console.WriteLine("{0} connected sockets with one
outstanding receive posted to each....press any key", m_outstandingReadCount);
        Console.WriteLine("Press any key to terminate the server process....");
        Console.ReadKey();
    }
    // Begins an operation to accept a connection request from the client
    //
    // <param name="acceptEventArg">The context object to use when issuing
    // the accept operation on the server's listening socket</param>
    public void StartAccept(SocketAsyncEventArgs acceptEventArg)
    {
        if (acceptEventArg == null)
        {
            acceptEventArg = new SocketAsyncEventArgs();
            acceptEventArg.Completed += new
EventHandler<SocketAsyncEventArgs>(AcceptEventArg_Completed);
        }
        else
```

```
        {
            // socket must be cleared since the context object is being reused
            acceptEventArg.AcceptSocket = null;
        }
        m_maxNumberAcceptedClients.WaitOne();
        bool willRaiseEvent = listenSocket.AcceptAsync(acceptEventArg);
        if (!willRaiseEvent)
        {
            ProcessAccept(acceptEventArg);
        }
    }
    // This method is the callback method associated with Socket.AcceptAsync
    // operations and is invoked when an accept operation is complete
    //
    void AcceptEventArg_Completed(object sender, SocketAsyncEventArgs e)
    {
        ProcessAccept(e);
    }
    private void ProcessAccept(SocketAsyncEventArgs e)
    {
        Interlocked.Increment(ref m_numConnectedSockets);
        Console.WriteLine("Client connection accepted. There are {0} clients connected to the
server",m_numConnectedSockets);
        // Get the socket for the accepted client connection and put it into the
        //ReadEventArg object user token
        SocketAsyncEventArgs readEventArgs = m_readWritePool.Pop();
        ((AsyncUserToken)readEventArgs.UserToken).Socket = e.AcceptSocket;
        // As soon as the client is connected, post a receive to the connection
        bool willRaiseEvent = e.AcceptSocket.ReceiveAsync(readEventArgs);
        if (!willRaiseEvent)
        {
            ProcessReceive(readEventArgs);
        }
        // Accept the next connection request
        StartAccept(e);
    }
    // This method is called whenever a receive or send operation is completed on a socket
    //
    // <param name="e">SocketAsyncEventArg associated with the completed receive operation</param>
    void IO_Completed(object sender, SocketAsyncEventArgs e)
    {
        // determine which type of operation just completed and call the associated handler
        switch (e.LastOperation)
        {
```

```
            case SocketAsyncOperation.Receive:
                ProcessReceive(e);
                break;
            case SocketAsyncOperation.Send:
                ProcessSend(e);
                break;
            default:
                throw new ArgumentException("The last operation completed on the socket was not
a receive or send");
        }
    }
    // This method is invoked when an asynchronous receive operation completes.
    // If the remote host closed the connection, then the socket is closed.
    // If data was received then the data is echoed back to the client.
    //
    private void ProcessReceive(SocketAsyncEventArgs e)
    {
        // check if the remote host closed the connection
        AsyncUserToken token = (AsyncUserToken)e.UserToken;
        if (e.BytesTransferred > 0 && e.SocketError == SocketError.Success)
        {
            //increment the count of the total bytes receive by the server
            Interlocked.Add(ref m_totalBytesRead, e.BytesTransferred);
            Console.WriteLine("The server has read a total of {0} bytes", m_totalBytesRead);
            Int32 BytesToProcess = e.BytesTransferred;
            Byte[] bt = new Byte[BytesToProcess];
            Buffer.BlockCopy(e.Buffer, e.Offset, bt, 0, BytesToProcess);
            string strReceive = Encoding.Default.GetString(bt);
            Send(token.Socket, bt, 0, bt.Length, 1000);
            Thread.Sleep(1000);
            //echo the data received back to the client
            //e.SetBuffer(e.Offset, e.BytesTransferred);
            bool willRaiseEvent = token.Socket.SendAsync(e);
            if (!willRaiseEvent)
            {
                ProcessSend(e);
            }
        }
        else
        {
            CloseClientSocket(e);
        }
    }
    public static void Send(Socket socket, byte[] buffer, int offset, int size, int timeout)
```

```
{
    socket.SendTimeout = 0;
    int startTickCount = Environment.TickCount;
    int sent = 0;  // how many bytes is already sent
    do
    {
        if (Environment.TickCount > startTickCount + timeout)
        //throw new Exception("Timeout.");
            try
            {
                sent += socket.Send(buffer, offset + sent, size - sent, SocketFlags.None);
            }
            catch (SocketException ex)
            {
                if (ex.SocketErrorCode == SocketError.WouldBlock
                    ||ex.SocketErrorCode == SocketError.IOPending
                    ||  ex.SocketErrorCode == SocketError.NoBufferSpaceAvailable)
                {
                    // socket buffer is probably full, wait and try again
                    Thread.Sleep(30);
                }
                else
                    throw ex;  // any serious error occurr
            }
    } while (sent < size);
}    // This method is invoked when an asynchronous send operation completes.
// The method issues another receive on the socket to read any additional
// data sent from the client
//
// <param name="e"></param>
private void ProcessSend(SocketAsyncEventArgs e)
{
    if (e.SocketError == SocketError.Success)
    {
        //e.SetBuffer(e.Offset, 10);
        // done echoing data back to the client
        AsyncUserToken token = (AsyncUserToken)e.UserToken;
        // read the next block of data send from the client
        bool willRaiseEvent = token.Socket.ReceiveAsync(e);
        if (!willRaiseEvent)
        {
            ProcessReceive(e);
        }
    }
```

```
        else
        {
            CloseClientSocket(e);
        }
    }
    private void CloseClientSocket(SocketAsyncEventArgs e)
    {
        AsyncUserToken token = e.UserToken as AsyncUserToken;
        // close the socket associated with the client
        try
        {
            token.Socket.Shutdown(SocketShutdown.Send);
        }
        // throws if client process has already closed
        catch (Exception)
        { }
        token.Socket.Close();
        // decrement the counter keeping track of the total number of clients connected to the server
        Interlocked.Decrement(ref m_numConnectedSockets);
        m_maxNumberAcceptedClients.Release();
        Console.WriteLine("A client has been disconnected from the server. There are {0} clients
connected to the server", m_numConnectedSockets);
        // Free the SocketAsyncEventArg so they can be reused by another client        m_readWritePool.Push(e);
    }
}
```

```
using System;
using System.Net;
using System.Collections.Generic;
using System.IO;
class Program {
    static void Main(string[] args)
    {
        IPEndPoint iep = new IPEndPoint(IPAddress.Parse("10.1.20.6"), 1333);
        Server objServer = new Server(1000, 10);
        objServer.Init();
        objServer.Start(iep);
    }
}
```

7、最基本的Socket编程

7.1、版权声明

文章出处：http://www.cnblogs.com/bingyun84/archive/2009/10/16/1584393.html
文章作者：冰云

7.2、内容详情

说明：此示例在.net2005"xp 下运行通过

示例程序是同步套接字程序，功能很简单，只是客户端发给服务器一条信息，服务器向客户端返回一条信息；这里只是一个简单的示例，是一个最基本的 socket 编程流程，在接下来的文章中，会依次记录套接字的同步和异步，以及它们的区别。

下面是示例程序的简单步骤说明

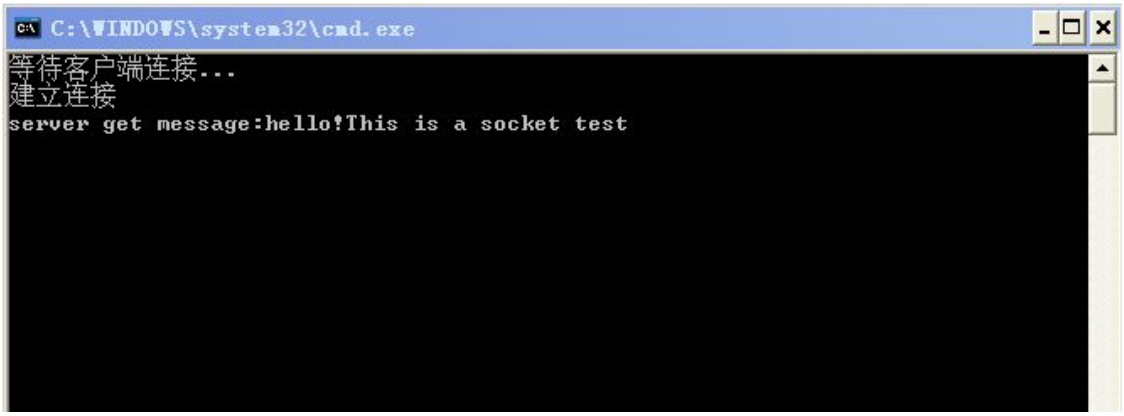
服务器端：

第一步：用指定的端口号和服务器的 ip 建立一个 EndPoint 对像；
第二步：建立一个 Socket 对像；
第三步：用 socket 对像的 Bind()方法绑定 EndPoint；
第四步：用 socket 对像的 Listen()方法开始监听；
第五步：接受到客户端的连接，用 socket 对像的 Accept()方法创建新的 socket 对像用于和请求的客户端进行通信;
第六步：通信结束后一定记得关闭 socket;

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace server
{
    class Program
    {
        static void Main(string[] args)
        {
            int port = 2000;
            string host = "127.0.0.1";
```

```
            ///创建终结点（EndPoint）
            IPAddress ip = IPAddress.Parse(host);//把 ip 地址字符串转换为 IPAddress 类型的实例
            IPEndPoint ipe = new IPEndPoint(ip, port);//用指定的端口和 ip 初始化 IPEndPoint 类
的新实例
            ///创建 socket 并开始监听
            Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
            //创建一个 socket 对像，如果用 udp 协议，则要用 SocketType.Dgram 类型的套接字
            s.Bind(ipe);//绑定 EndPoint 对像（2000 端口和 ip 地址）
            s.Listen(0);//开始监听
            Console.WriteLine("等待客户端连接");
            ///接受到 client 连接，为此连接建立新的 socket，并接受信息
            Socket temp = s.Accept();//为新建连接创建新的 socket
            Console.WriteLine("建立连接");
            string recvStr = "";
            byte[] recvBytes = new byte[1024];
            int bytes;
            bytes = temp.Receive(recvBytes, recvBytes.Length, 0);//从客户端接受信息
            recvStr += Encoding.ASCII.GetString(recvBytes, 0, bytes);
            ///给 client 端返回信息
            Console.WriteLine("server get message:{0}", recvStr);//把客户端传来的信息显示出来
            string sendStr = "ok!Client send message successful!";
            byte[] bs = Encoding.ASCII.GetBytes(sendStr);
            temp.Send(bs, bs.Length, 0);//返回信息给客户端
            temp.Close();
            s.Close();
            Console.ReadLine();
        }
    }
}
```

Server 结果：



客户端：

第一步：用指定的端口号和服务器的 ip 建立一个 EndPoint 对像；
第二步：建立一个 Socket 对像；
第三步：用 socket 对像的 Connect()方法以上面建立的 EndPoint 对像做为参数，向服务器发出连接请求；
第四步：如果连接成功，就用 socket 对像的 Send()方法向服务器发送信息；
第五步：用 socket 对像的 Receive()方法接受服务器发来的信息 ；
第六步：通信结束后一定记得关闭 socket；

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace Client
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int port = 2000;
                string host = "127.0.0.1";
                ///创建终结点 EndPoint
                IPAddress ip = IPAddress.Parse(host);
                //IPAddress ipp = new IPAddress("127.0.0.1");
                IPEndPoint ipe = new IPEndPoint(ip, port);//把 ip 和端口转化为 IPEndpoint 实例
                ///创建 socket 并连接到服务器
                Socket c = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
                    //创建 Socket
                Console.WriteLine("Conneting…");
                c.Connect(ipe);//连接到服务器
                ///向服务器发送信息
                string sendStr = "hello!This is a socket test";
                byte[] bs = Encoding.ASCII.GetBytes(sendStr);//把字符串编码为字节
                Console.WriteLine("Send Message");
                c.Send(bs, bs.Length, 0);//发送信息
                ///接受从服务器返回的信息
                string recvStr = "";
                byte[] recvBytes = new byte[1024];
                int bytes;
                bytes = c.Receive(recvBytes, recvBytes.Length, 0);//从服务器端接受返回信息
```

```
                recvStr += Encoding.ASCII.GetString(recvBytes, 0, bytes);
                Console.WriteLine("client get message:{0}", recvStr);//显示服务器返回信息
                ///一定记着用完 socket 后要关闭
                c.Close();
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("argumentNullException: {0}", e);
            }
            catch (SocketException e)
            {
                Console.WriteLine("SocketException:{0}", e);
            }
            Console.WriteLine("Press Enter to Exit");
        }
    }
}
```

Client 端结果：