21 世纪高等学校计算机应用技术系列规划教材

C#网络应用高级编程

马　骏　编著

人民邮电出版社

内容提要

本书是《C#网络应用编程基础》的姊妹篇。该书在《C#网络应用编程基础》介绍的基础知识之上，进一步讲解了利用 C#进行各种高级应用编程的方法和技巧。全书语言简洁，重点突出，思路清晰，实用性强。教材紧跟 C#和 Microsoft Visual Studio 开发工具最新版本的变化而及时更新。同时有与本书配套的实验指导、电子教案、所有例题与习题的源程序和全部习题参考解答。

本书共分 8 章。其中第 1～5 章介绍了 C#多线程处理、TCP、UDP、SMTP 和 POP3 协议的高级应用以及 P2P 应用编程，第 6 章介绍了网络数据加密与解密的方法，第 7 章介绍了三维设计与多媒体编程方法，第 8 章为与本书配套的上机实验指导。

本书可作为高等院校计算机及相关专业的高年级学生教材，也适用于有一定的 C#语言编程基础，而想利用 C#和 VS2005 开发工具进行更复杂的高级应用编程的人员阅读。

目　　录

前　言

在 Internet 和计算机技术高速发展的时代，处处都有网络的影子。在网络无处不在的环境下，任何一种开发和编程都不可避免的涉及到网络应用。网络办公、网络游戏、网络电视、电子商务、网络资源搜索、邮件处理、千姿百彩的网站、杂志订阅、远程控制……，以及其它各种网络相关的应用软件，都是网络应用编程的实际应用。早期的网络编程难度大、效率低，使初学者望而生畏，而 C#（读作 C Sharp）和.NET（读作 dot net）平台大大地简化了这些技术，使过去困难的网络应用编程变得非常轻松。

目前，很多高校计算机及相关专业都开设了网络应用编程方面的课程。在就业压力日益增大以及学生对所学知识与社会实际需求紧密结合的迫切要求的情况下，本书涉及的内容和知识面显得尤为重要。

Microsoft Visual Studio 2005（简称 VS2005）是微软公司最新推出的先进的可视化开发工具。2005 年底，微软隆重发布了 VS2005 英文版，并于 2006 年发布了 VS2005 简体中文版。本书以 VS2005 Professional 简体中文版为开发环境，以易学易用为重点，充分考虑了学以致用和社会实际需求，用大量的实例，引导读者快速掌握用 C#进行各类网络应用程序的编程方法和技巧。使学生学习本书内容后能立即编写出与企事业实际应用紧密结合的程序，避免说起来似懂非懂，做起来寸步难行的尴尬局面。

C#是在 C、C++和 Java 基础上开发的在.NET 平台上运行的为适应 Internet、Intranet 和各类网络应用而设计的编程语言，该语言综合了 C、C++和 Java 以及其他高级语言的优点，是一种语法优雅、类型安全、完全面向对象的编程语言。由于 C#是专门为.NET 平台而设计的开发语言，并于 2001 年由 ECMA（European Computer Manufactures Association，欧洲计算机制造商协会）规定为高级语言开发标准（ECMA-334），2003 年被 ISO（International Standards Organization，国际标准化组织）规定为国际标准（ISO/IEC 23270），因此它比任何在.NET 平台下提供的其他开发语言都有无可比拟的优越性。毫无疑问，随着.NET 技术的普及，C#必将成为开发 Internet 和企业级应用程序的首选程序设计语言。

本书的所有程序均在 VS2005 Professional 简体中文版开发环境下调试通过。是一本非常实用的学习 C#网络高级应用编程的教材。读者通过阅读、上机练习和调试运行，能快速理解用 C#进行各类高级应用编程的方法和技巧。另外，由于本书的内容是在《C#网络应用编程基础》（马骏主编，人民邮电出版社）介绍的基础知识之上进一步介绍更为复杂的高级应用编程方法和技巧，因此需要读者有一定的 C#语言编程基础。

本书由马骏编著。参与编写和代码调试等工作的还有陈明、王芳、杨韶华、张瑞青、郑珂等。人民邮电出版社的邹文波编辑对本书编写给予了大力支持、指导和帮助，在此表示深深的谢意。

由于编者水平有限，书中难免存在错误之处，敬请读者批评指正。

编　者

2006 年

第 1 章　进程、线程与网络协议

在 Windows 应用程序中，基于客户端和服务器的各种网络编程技术应用非常广泛，本书将逐步介绍这类应用程序的编程方法。

在《C#网络应用编程基础》（马骏主编，人民邮电出版社）一书中，我们已经学习了 C#面向对象编程的基础知识以及基本控件的用法。本书假定读者对相关基本内容比较熟悉，并在此基础上，进一步学习更为复杂的基于 Windows 窗体的高级应用开发，而不再介绍 C#语法以及控件的用法等基础内容。如果读者对 C#相关知识不太熟悉，请首先阅读《C#网络应用编程基础》一书。

本章主要介绍进程、线程、IP 地址与端口、套接字以及网络流等基本知识，这部分内容是学习本书后面章节的基础，希望读者能很好的理解和掌握。

1.1　进程和线程

进程是对一段静态指令序列（程序）的动态执行过程，是系统进行资源分配和调度的一个基本单位。与进程相关的信息包括进程的用户标志、正在执行的已经编译好的程序、进程程序和数据在存储器中的位置等等。同一个进程又可以划分为若干个独立的执行流，我们称之为线程。线程是 CPU 调度和分配的基本单位。在 Windows 环境下，用户可以同时运行多个应用程序，每个执行的应用程序就是一个进程。例如一台电脑上同时打开两个 QQ 时，每个运行的 QQ 就是一个进程；而用一个 QQ 和多个人聊天时，每个聊天窗口就是一个线程。

进程和线程概念的提出，对提高软件的并行性有着重要的意义。并行性的主要特点就是并发处理。在一个单处理器系统中，可以通过分时处理来获得并发，这种情况下，系统为每个线程分配一个 CPU 时间片，每个线程只有在分配的时间片内才拥有对 CPU 的控制权，其他时间都在等待。即同一时间只有一个线程在运行。由于系统为每个线程划分的时间片很小（20 毫秒左右），所以在用户看来，好像是多个线程在同时运行。

为什么要使用多线程呢？考虑这样一种情况：在 C/S 模式下，服务器需要不断监听来自各个客户端的请求，这时，如果采用单线程机制的话，服务器将无法处理其他事情，因为这个线程要不断的循环监听请求而无暇对其他请求做出响应。实际上，当要花费大量时间进行连续的操作时，或者等待网络或其他 I/O 设备响应时，都可以使用多线程技术。

在 C#中，有两个专门用于处理进程和线程的类：Process 类和 Thread 类。

1.1.1　Process 类

Process 类位于 System.Diagnostics 命名空间下，用于完成进程的相关处理任务。可以在本地计算机上启动和停止进程，也可以查询进程的相关信息。在自己的程序中运行其他的应用程序，实际上就是对进程进行管理。如果希望在自己的进程中启动和停止其他进程，首先要创建 Process 类的实例，并设置对象的 StartInfo 属性，然后调用该对象的 Start 方法启动进程。

【例 1-1】启动、停止和观察进程。

(1) 新建一个名为 ProcessExample 的 Windows 应用程序，设计界面如图 1-1 所示。

(2) 展开工具箱中的【组件】选项卡，然后将 Process 组件拖放到设计窗体上。

(3) 在代码的开始部分添加命名空间引用：

```
using System.Diagnostics;
using System.Threading;
```

(4) 分别在【启动记事本】、【停止记事本】和【观察所有进程】三个按钮的 Click 事件中添加代码：

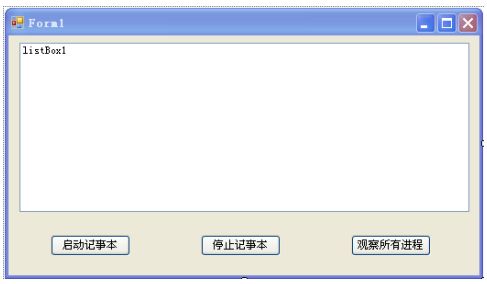


图1-1　测试进程界面

```
private void buttonStart_Click(object sender, EventArgs e)
{
    process1.StartInfo.FileName = "notepad.exe";
    //启动 Notepad.exe 进程.
    process1.Start();
}
private void buttonStop_Click(object sender, EventArgs e)
{
    //创建新的 Process 组件的数组,并将它们与指定的进程名称（Notepad）的所有进程资源相关联.
    Process[] myprocesses;
    myprocesses = Process.GetProcessesByName("Notepad");
    foreach (Process instance in myprocesses)
    {
        //设置终止当前线程前等待 1000 毫秒
        instance.WaitForExit(1000);
        instance.CloseMainWindow();
    }
}
private void buttonView_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    //创建 Process 类型的数组,并将它们与系统内所有进程相关联
    Process[] processes;
    processes = Process.GetProcesses();
    foreach (Process p in processes)
    {
        //Idle 指显示 CPU 空闲率的进程名称
        //由于访问 Idle 的 StartTime 会出现异常，所以将其排除在外
        if (p.ProcessName != "Idle")
        {
            //将每个进程名和进程开始时间加入 listBox1 中
            this.listBox1.Items.Add(
            string.Format("{0,-30}{1:h:m:s}", p.ProcessName, p.StartTime));
        }
    }
}
```

(5) 按<F5>键编译并执行，单击几次【启动记事本】按钮，观察打开的每个进程，然后单击【停止记事本】，观察依次停止进程的情况。

由于安装 Windows 操作系统后，notepad.exe 就已经安装到系统文件夹下，而且在任何一个文件夹中均可以直接运行，所以在这个例子中，我们选择了调用 notepad.exe 作为演示的例

子。实际上，任何一个可执行文件均可以通过这种方法调用，读者可以自行尝试调用其他可执行文件，并观察执行效果。

1.1.2 Thread 类

在 System.Threading 命名空间下，包含了用于创建和控制线程的 Thread 类。对线程的常用操作有：启动线程、终止线程、合并线程和让线程休眠等。

1. 启动线程

在使用线程前，首先要创建一个线程。其一般形式为：

```
Thread t=new Thread(enterPoint);
```

其中 enterPoint 为线程的入口，即线程开始执行的方法。在托管代码中，通过委托处理线程执行的代码。例如：

```
Thread t=new Thread(new ThreadStart(methodName));
```

创建线程实例后，就可以调用 Start 方法启动线程了。

2. 终止线程

线程启动后，当不需要某个线程继续执行的时候，有两种终止线程的方法。

一种是事先设置一个布尔变量，在其他线程中通过修改该变量的值作为传递给该线程是否需要终止的判断条件，而在该线程中循环判断该条件，以确定是否退出线程，这是结束线程的比较好的方法，实际编程中一般使用这种方法。

第二种方法是通过调用 Thread 类的 Abort 方法强行终止线程。例如：

```
t.Abort();
```

Abort 方法没有任何参数，线程一旦被终止，就无法再重新启动。由于 Abort 通过抛出异常强行终止结束线程，因此在实际编程中，应该尽量避免采用这种方法。

调用 Abort 方法终止线程时，公共语言运行库（CLR）会引发 ThreadAbortException 异常，程序员可以在线程中捕获 ThreadAbortException 异常，然后在异常处理的 Catch 块或者 Finally 块中作释放资源等代码处理工作；但是，线程中也可以不捕获 ThreadAbortException 异常，而由系统自动进行释放资源等处理工作。

注意，如果线程中捕获了 ThreadAbortException 异常，系统在 finally 子句的结尾处会再次引发 ThreadAbortException 异常，如果没有 finally 子句，则会在 Catch 子句的结尾处再次引发该异常。为了避免再次引发异常，可以在 finally 子句的结尾处或者 Catch 子句的结尾处调用 System.Threading.Thread.ResetAbort 方法防止系统再次引发该异常。

使用 Abort 方法终止线程，调用 Abort 方法后，线程不一定会立即结束。这是因为系统在结束线程前要进行代码清理等工作，这种机制可以使线程的终止比较安全，但清理代码需要一定的时间，而我们并不知道这个工作将需要多长时间。因此，调用了线程的 Abort 方法后，如果系统自动清理代码的工作没有结束，可能会出现类似死机一样的假象。为了解决这个问题，可以在主线程中调用子线程对象的 Join 方法，并在 Join 方法中指定主线程等待子线程结束的等待时间。

3. 合并线程

Join 方法用于把两个并行执行的线程合并为一个单个的线程。如果一个线程 t1 在执行的过程中需要等待另一个线程 t2 结束后才能继续执行，可以在 t1 的程序模块中调用 t2 的 join() 方法。例如：

t2.Join();

这样 t1 在执行到 t2.Join()语句后就会处于阻塞状态，直到 t2 结束后才会继续执行。

但是假如 t2 一直不结束，那么等待就没有意义了。为了解决这个问题，可以在调用 t2 的 Join 方法的时候指定一个等待时间，这样 t1 这个线程就不会一直等待下去了。例如，如果希望将 t2 合并到 t1 后，t1 只等待 100 毫秒，然后不论 t2 是否结束，t1 都继续执行，就可以在 t1 中加上语句：

t2.Join(100);

Join 方法通常和 Abort 一起使用。

由于调用某个线程的 Abort 方法后，我们无法确定系统清理代码的工作什么时候才能结束，因此如果希望主线程调用了子线程的 Abort 方法后，主线程不必一直等待，可以调用子线程的 Join 方法将子线程连接到主线程中，并在连接方法中指定一个最大等待时间，这样就能使主线程继续执行了。

4. 让线程休眠

在多线程应用程序中，有时候并不希望某一个线程继续执行，而是希望该线程暂停一段时间，等待其他线程执行之后再继续执行。这时可以调用 Thread 类的 Sleep 方法，即让线程休眠。例如：

```
Thread.Sleep(1000);
```

这条语句的功能是让当前线程休眠 1000 毫秒。

注意，调用 Sleep 方法的是类本身，而不是类的实例。休眠的是该语句所在的线程，而不是其他线程。

5. 线程优先级

当线程之间争夺 CPU 时间片时，CPU 是按照线程的优先级进行服务的。在 C#应用程序中，可以对线程设定五个不同的优先级，由高到低分别是 Highest、AboveNormal、Normal、BelowNormal 和 Lowest。在创建线程时如果不指定其优先级，则系统默认为 Normal。假如想让一些重要的线程优先执行，可以使用下面的方法为其赋予较高的优先级：

```
Thread t=new Thread(new ThreadStart(enterpoint));
t.priority=ThreadPriority.AboveNormal;
```

通过设置线程的优先级可以改变线程的执行顺序，所设置的优先级仅仅适用于这些线程所属的进程。

注意，当把某线程的优先级设置为 Highest 时，系统上正在运行的其他线程都会终止，所以使用这个优先级别时要特别小心。

6. 线程池

线程池是一种多线程处理形式，为了提高系统性能，在许多地方都要用到线程池技术。例如，在一个 C/S 模式的应用程序中的服务器端，如果每到一个请求就创建一个新线程，然后在新线程中为其请求服务的话，将不可避免的造成系统开销的增大。实际上，创建太多的线程可能会导致由于过度使用系统资源而耗尽内存。为了防止资源不足，服务器端应用程序应采取一定的办法来限制同一时刻处理的线程数目。

线程池为线程生命周期的开销问题和资源不足问题提供了很好的解决方案。通过对多个任务重用线程，线程创建的开销被分摊到了多个任务上。其好处是，由于请求到达时线程已经存在，所以无意中也就消除了线程创建所带来的延迟。这样，就可以立即为新线程请求服务，使

其应用程序响应更快。而且，通过适当地调整线程池中的线程数目，也就是当请求的数目超过了规定的最大数目时，就强制其他任何新到的请求一直等待，直到获得一个线程来处理为止，从而可以防止资源不足。

线程池适用于需要多个线程而实际执行时间又不多的场合，比如有些常处于阻塞状态的线程。当一个应用程序服务器接受大量短小线程的请求时，使用线程池技术是非常合适的，它可以大大减少线程创建和销毁的次数，从而提高服务器的工作效率。但是如果线程要求运行的时间比较长的话，那么此时线程的运行时间比线程的创建时间要长得多，仅靠减少线程的创建时间对系统效率的提高就不是那么明显了，此时就不适合使用线程池技术，而需要借助其他的技术来提高服务器的服务效率。

7. 同步

同步是多线程中一个非常重要的概念。所谓同步，是指多个线程之间存在先后执行顺序的关联关系。如果一个线程必须在另一个线程完成某个工作后才能继续执行，则必须考虑如何让其保持同步，以确保在系统上同时运行多个线程而不会出现逻辑错误。

当两个线程 t1 和 t2 有相同的优先级，并且同时在系统上运行时，如果先把时间片分给 t1 使用，它在变量 variable1 中写入某个值，但如果在时间片用完时它仍没有完成写入，这时由于时间片已经分给 t2 使用，而 t2 又恰好要尝试读取该变量，它可能就会读出错误的值。这时，如果使用同步仅允许一个线程使用 variable1，在该线程完成对 variable1 的写入工作后再让 t2 读取这个值，就可以避免出现此类错误。

为了对线程中的同步对象进行操作，C#提供了 lock 语句锁定需要同步的对象。lock 关键字确保当一个线程位于代码的临界区时，另一个线程不进入临界区。如果其他线程试图进入锁定的代码，则它将一直等待（即被阻塞），直到该对象被释放。比如线程 t1 对 variable1 操作时，为了避免其他线程也对其进行操作，可以使用 lock 语句锁定 variable1，实现代码为：

```
lock(variable1)
{
    variable1++;
}
```

注意，锁定的对象一定要声明为 private，不要锁定 public 类型的对象，否则将会使 lock 语句无法控制，从而引发一系列问题。

另外还要注意，由于锁定一个对象之后，其他任何线程都不能访问这个对象，需要使用该对象的线程就只能等待该对象被解除锁定后才能使用。因此如果在锁定和解锁期间处理的对象过多，就会降低应用程序的性能。

还有，如果两个不同的线程同时锁定两个不同的变量，而每个线程又都希望在锁定期间访问对方锁定的变量，那么两个线程在得到对方变量的访问权之前都不会释放自己锁定的对象，从而产生死锁。在编写程序时，要注意避免这类操作引起的问题。

【例 1-2】线程的基本用法。

(1) 新建一个名为 ThreadExample 的 Windows 应用程序，界面设计如图 1-2 所示。

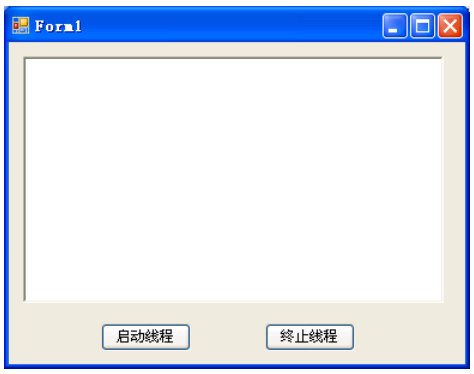


图1-2　例1-2的设计界面

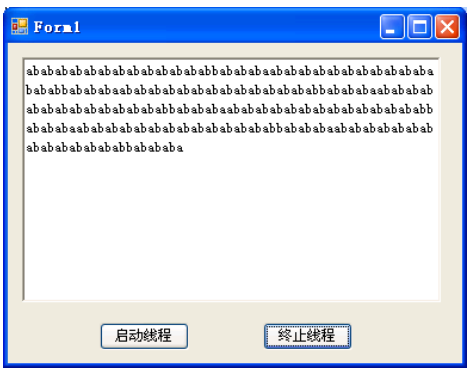


图1-3　例1-2的运行界面

(2) 向设计窗体拖放一个 Timer 组件，不改变自动生成的对象名。

(3) 添加命名空间引用：

```
using System.Threading;
```

(4) 在构造函数上方添加字段声明：

```
StringBuilder sb = new StringBuilder();
Thread thread1;
Thread thread2;
```

(5) 直接添加代码：

```
private void AppendString(string s)
{
    lock(sb)
    {
        sb.Append(s);
    }
}
public void Method1()
{
    while(true)
    {
        Thread.Sleep(100);     //线程休眠 100 毫秒
        AppendString("a");
    }
}
public void Method2()
{
    while(true)
    {
        Thread.Sleep(100);     //线程休眠 100 毫秒
        AppendString("b");
    }
}
```

(6) 分别在【启动线程】和【终止线程】按钮的 Click 事件中添加代码：

```
private void buttonStart_Click(object sender, EventArgs e)
{
    sb.Remove(0, sb.Length);
```

```
        timer1.Enabled = true;
        thread1 = new Thread(new ThreadStart(Method1));
        thread2 = new Thread(new ThreadStart(Method2));
        thread1.Start();
        thread2.Start();
    }
    private void buttonAbort_Click(object sender, EventArgs e)
    {
        thread1.Abort();
        thread1.Join(10);
        thread2.Abort();
        thread2.Join(10);
    }
```

(7) 在 timer1 的 Tick 事件中添加代码：

```
    private void timer1_Tick(object sender, EventArgs e)
    {
        if (thread1.IsAlive == true || thread2.IsAlive == true)
        {
            richTextBox1.Text = sb.ToString();
        }
        else
        {
            timer1.Enabled = false;
        }
    }
```

(8) 按<F5>键编译并执行，单击【启动线程】后，再单击【终止线程】，从图 1-3 所示的运行结果中可以看到，两个具有相同优先级的线程同时执行时，在 richTextBox1 中添加的字符个数基本上相同。

1.1.3 在一个线程中操作另一个线程的控件

默认情况下，C#不允许在一个线程中直接操作另一个线程中的控件，这是因为访问 Windows 窗体控件本质上不是线程安全的。如果有两个或多个线程操作某一控件的状态，则可能会迫使该控件进入一种不一致的状态。还可能出现其他与线程相关的 bug，以及不同线程争用控件引起的死锁问题。因此确保以线程安全方式访问控件非常重要。

在调试器中运行应用程序时，如果创建某控件的线程之外的其他线程试图调用该控件，则调试器会引发一个 InvalidOperationException 异常，并提示消息：“从不是创建控件的线程访问它”。

但是在 Windows 应用程序中，为了在窗体上显示线程中处理的信息，我们可能需要经常在一个线程中引用另一个线程中的窗体控件。比较常用的办法之一是使用委托（delegate）来完成这个工作。

为了区别是否是创建控件的线程访问该控件对象，Windows 应用程序中的每一个控件对象都有一个 InvokeRequired 属性，用于检查是否需要通过调用 Invoke 方法完成其他线程对该控件的操作，如果该属性为 true，说明是其他线程操作该控件，这时可以创建一个委托实例，然后调用控件对象的 Invoke 方法，并传入需要的参数完成相应操作，否则可以直接对该控件对象进行操作，从而保证了安全代码下线程间的互操作。例如：

```
delegate void AppendStringDelegate(string str);
private void AppendString(string str)
{
    if (richTextBox1.InvokeRequired)
    {
        AppendStringDelegate d = new AppendStringDelegate(AppendString);
        richTextBox1.Invoke(d, "abc");
    }
    else
    {
        richTextBox1.Text += str;
    }
}
```

这段代码中，首先判断是否需要通过委托调用对 richTextBox1 的操作，如果需要，则创建一个委托实例，并传入需要的参数完成 else 代码块的功能；否则直接执行 else 代码块中的内容。

实际上，由于我们在编写程序时就已经知道控件是在哪个线程中创建的，因此也可以在不是创建控件的线程中直接调用控件对象的 Invoke 方法完成对该线程中的控件的操作。

注意，不论是否判断 InvokeRequired 属性，委托中参数的个数和类型必须与传递给委托的方法需要的参数个数和类型完全相同。

【例 1-3】一个线程操作另一个线程的控件的方法。

(1) 新建一个名为 ThreadControlExample 的 Windows 应用程序，界面设计如图 1-4 所示。

图1-4　例1-3的设计界面

图1-5　例1-3的运行界面

(2) 添加命名空间引用：

```
using System.Threading;
```

(3) 在构造函数上方添加字段声明，并在构造函数中初始化对象：

```
Thread thread1;
Thread thread2;
delegate void AppendStringDelegate(string str);
AppendStringDelegate appendStringDelegate;
public Form1()
{
    InitializeComponent();
    appendStringDelegate = new AppendStringDelegate(AppendString);
}
```

(4) 直接添加代码：

```
private void AppendString(string str)
{
    richTextBox1.Text += str;
}
private void Method1()
{
    while (true)
    {
        Thread.Sleep(100);     //线程 1 休眠 100 毫秒
        richTextBox1.Invoke(appendStringDelegate, "a");
    }
}
private void Method2()
{
    while (true)
    {
        Thread.Sleep(100);     //线程 2 休眠 100 毫秒
        richTextBox1.Invoke(appendStringDelegate, "b");
    }
}
```

(5) 分别在【启动线程】和【终止线程】按钮的 Click 事件中添加代码：

```
private void buttonStart_Click(object sender, EventArgs e)
{
    richTextBox1.Text = "";
    thread1 = new Thread(new ThreadStart(Method1));
    thread2 = new Thread(new ThreadStart(Method2));
    thread1.Start();
    thread2.Start();
}

private void buttonStop_Click(object sender, EventArgs e)
{
    thread1.Abort();
    thread1.Join();
    thread2.Abort();
    thread2.Join();
    MessageBox.Show("线程 1、2 终止成功");
}
```

(6) 按<F5>键编译并执行，单击【启动线程】后，再单击【终止线程】，运行结果如图 1-5 所示。

1.2 IP 地址与端口

IP（Internet Protocol）是 Internet 网络设备之间传输数据的一种协议。本节所讲的端口是逻辑意义上的端口，即 TCP/IP 协议中的端口。这一节我们主要学习 IPAddress、IPHostEntry、IPEndPoint 等 System.Net 命名空间中的几个类，为以后的应用打下基础。

1.2.1 TCP/IP 网络协议

网络协议是网络上所有设备（网络服务器、计算机及交换机、路由器、防火墙等）之间通信规则的集合，它定义了通信时信息必须采用的格式以及这些格式的含义。网络协议使网络上各种设备能够相互交换信息。

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）是一组网络通信协议的总称，它规范了网络上的所有通信设备，尤其是一个主机与另一个主机之间的数据交换格式以及传送方式。对普通用户来说，并不需要了解网络协议的整个结构，仅需了解 IP 的地址格式，即可与世界各地进行网络通信。

IP 地址就是给每个连接在因特网上的主机（或路由器）分配一个在全世界范围内惟一的标识符。一个 IP 地址主要由两部分组成：一部分是用于标识该地址所从属的网络号，另一部分用于指明该网络上某个特定主机的主机号。网络号由因特网权力机构分配，主机地址由各个网络的管理员统一分配。因此，网络地址的惟一性与网络内主机地址的惟一性确保了 IP 地址的全球惟一性。

目前，大多数 IP 编址方案仍采用 IPv4 编址方案，即使用 32 位的二进制地址进行识别，我们常见的形式是将 32 位的 IP 地址分成 4 个字节，然后把 4 个字节分别用十进制表示，中间用圆点分开，这种方法叫做点分十进制表示法。

使用 IP 地址的点分十进制表示法，网络类的范围划分如下：

A 类：0.x.x.x—127.x.x.x　　（32 位二进制最高位为 0）

B 类：128.x.x.x—191.x.x.x　　（32 位二进制最高 2 位为 10）

C 类：192.x.x.x—223.x.x.x　　（32 位二进制最高 3 位为 110）

D 类：224.x.x.x—239.x.x.x　　（32 位二进制最高 4 位为 1110）

E 类：240.x.x.x—255.x.x.x　　（32 位二进制最高 5 位为 11110）

其中，D 类地址用于组播，E 类地址保留用于扩展或实验用。

在这些网络类中，每类网络又可以进一步分成不同的网络，或者叫子网。每个子网必须用一个公共的网址把它与该类网络中的其他子网分开。为了识别 IP 地址的网络部分，又为特定的子网定义了子网掩码。子网掩码用于区分哪些位作为网络地址部分，哪些位作为主机地址部分。把所有的网络位用 1 来标识，主机位用 0 来标识，就得到了子网掩码。

从表面上看，好像知道了服务器的 IP 地址，客户端就能够和服务器相互通信。其实，真正相互完成通信功能的不是两台计算机，而是两台计算机上的进程。IP 地址仅仅能够具体标识到某台主机，而不能标识某台计算机上的进程。如果要标识具体的进程，需要引入新的地址空间，这就是端口（Port）。在网络技术中，端口大致有两种意思：一是物理意义上的端口，比如，ADSL Modem、集线器、交换机、路由器上连接其他网络设备的接口，如 RJ-45 端口、SC 端口等等。二是逻辑意义上的端口，一般是指 TCP/IP 协议中的端口，端口号的范围从 0 到 65535，比如用于浏览网页服务的 80 端口，用于 FTP 服务的 21 端口等等。我们这里介绍的是逻辑意义上的端口。定义端口是为了解决与多个应用程序同时进行通信的问题；它主要扩充了 IP 地址的概念。假设一台计算机正在同时运行多个应用程序，并通过网络接收到了一个数据包，这时就可以利用一个独有的端口号（该端口号在建立连接时确定）来标识目标进程。因此，如果客户端 A 要与服务器 B 相互通信，客户端 A 不仅要知道服务器 B 的 IP 地址，而且要知道服务器 B 提供具体服务的端口号。

由于使用 16 位二进制表示端口地址，因此可用端口地址的范围是 0～65535。

1.2.2 IPAddress 类与 Dns 类

在 System.Net 命名空间中，IPAddress 类提供了对 IP 地址的转换、处理等功能。该类提供的 Parse 方法可将 IP 地址字符串转换为 IPAddress 实例。例如：

```
IPAddress ip = IPAddress.Parse("192.168.1.1");
```

IPAddress 类还提供了 7 个只读字段，分别代表程序中使用的特殊 IP 地址：

Any　　表示本地系统可用的任何 IPv4 地址

Broadcast　　表示本地 IPv4 网络广播地址

IPv6Any　　Socket.Bind 方法用此字段指出本地系统可用的 IPv6 地址

IPv6Loopback 表示系统的 IPv6 回送地址

IPv6None　　表示系统上没有可用的 IPv6 网络接口

Loopback　　表示系统的 IPv4 回送地址

None　　表示系统上没有可用的 IPv4 网络接口

System.Net 命名空间下，还有一个 Dns 类，该类提供了一系列静态的方法，用于获取提供本地或远程域名等功能。常用方法有：

1) GetHostAddresses 方法

获取指定主机的 IP 地址，返回一个 IPAddress 类型的数组。函数原形为：

```
public static IPAddress[] GetHostAddresses(string hostNameOrAddress);
```

例如：

```
IPAddress[] ip=Dns.GetHostAddresses("www.cctv.com");
listBox1.Items.AddRange(ip);
```

2) GetHostName 方法

获取本机主机名。例如：

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

1.2.3 IPHostEntry 类

IPHostEntry 类的实例对象中包含了 Internet 主机的相关信息。常用属性有两个：一个是 AddressList 属性，另一个是 HostName 属性。

AddressList 属性的作用是获取或设置与主机关联的 IP 地址列表，是一个 IPAddress 类型的数组，包含了指定主机的所有 IP 地址；HostName 属性则包含了服务器的主机名。

在 Dns 类中，有一个专门获取 IPHostEntry 对象的方法，通过 IPHostEntry 对象，可以获取本地或远程主机的相关 IP 地址。例如：

```
listBox1.Items.Add("搜狐新闻所用的服务器 IP 地址有：");
IPAddress[] ip = Dns.GetHostEntry("news.sohu.com").AddressList;
listBox1.Items.AddRange(ip);
listBox1.Items.Add("本机 IP 地址为：");
ip = Dns.GetHostEntry(Dns.GetHostName()).AddressList;
listBox1.Items.AddRange(ip);
```

1.2.4 IPEndPoint 类

要与远程主机进行通信，仅有 IP 地址是不够的。在 Internet 中，TCP/IP 使用一个网络地

址和一个服务端口号来唯一标识设备和服务。网络地址标识网络上的设备；端口号标识该设备上的特定服务。网络地址和服务端口的组合称为端点。在 C#中，使用 IPEndPoint 类表示这个端点，该类包含了应用程序连接到主机上的服务所需的 IP 地址和端口信息。

IPEndPoint 类常用的构造函数为：

```
public IPEndPoint(IPAddress, int);
```

其中第一个参数指定 IP 地址，第二个参数指定端口号。

例 1-4 演示了上述四个类的使用方法。在这个例子中，单击“显示本机 IP 信息”按钮可以显示主机名及相关的 IP 地址；单击“显示服务器信息”按钮可以显示中央电视台服务器的 IP 地址信息。

【例 1-4】IPAddress 类、Dns 类、IPHostEntry 类和 IPEndPoint 类的使用方法。

(1) 创建一个名为 IPExample 的 Windows 应用程序项目。

(2) 从【工具箱】中向设计窗体拖放一个 ListBox 控件、两个 Button 控件，设计界面效果如图 1-6 所示。

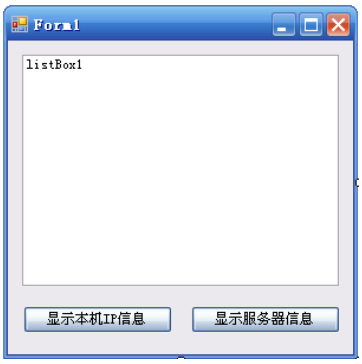


图1-6　例1-4的设计界面

(3) 切换到 Form1 的代码编辑模式，添加命名空间的引用：

```
using System.Net;
```

(4) 分别在两个按钮的 Click 事件中添加代码：

```
private void buttonLocal_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    string name = Dns.GetHostName();
    listBox1.Items.Add("本机主机名：" + name);
    IPHostEntry me = Dns.GetHostEntry(name);
    listBox1.Items.Add("本机所有 IP 地址：");
    foreach (IPAddress ip in me.AddressList)
    {
        listBox1.Items.Add(ip);
    }
    IPAddress localip = IPAddress.Parse("127.0.0.1");
    IPEndPoint iep = new IPEndPoint(localip, 80);
    listBox1.Items.Add("The IPEndPoint is: " + iep.ToString());
    listBox1.Items.Add("The Address is: " + iep.Address);
    listBox1.Items.Add("The AddressFamily is: " + iep.AddressFamily);
    listBox1.Items.Add("The max port number is: " + IPEndPoint.MaxPort);
    listBox1.Items.Add("The min port number is: " + IPEndPoint.MinPort);
private void buttonRemote_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    IPHostEntry remoteHost = Dns.GetHostEntry("www.cctv.com");
    IPAddress[] remoteIP = remoteHost.AddressList;
    listBox1.Items.Add("中央电视台：");
    foreach (IPAddress ip in remoteIP)
    {
        IPEndPoint iep = new IPEndPoint(ip, 80);
        listBox1.Items.Add(iep);
    }
}
```

(5) 按<F5>键编译并执行，观察本机和远程的 IP 地址情况。

1.3 套接字

套接字是支持 TCP/IP 协议的网络通信的基本操作单元。可以将套接字看作不同主机间的进程进行双向通信的端点，它构成了单个主机内及整个网络间的编程界面。套接字存在于通信域中，通信域是为了处理一般的线程通过套接字通信而引进的一种抽象概念。套接字通常和同一个域中的套接字交换数据（数据交换也可能穿越域的界限，但这时一定要执行某种解释程序）。各种进程使用这个相同的域互相之间用 Internet 协议进行通信。

套接字可以根据通信性质分类，这种性质对于用户是可见的。应用程序一般仅在同一类的套接字间进行通信。不过只要底层的通信协议允许，不同类型的套接字间也照样可以通信。套接字有两种不同的类型：流套接字和数据报套接字。

要通过互联网进行通信，至少需要一对套接字，其中一个运行于客户端，我们称之为 ClientSocket，另一个运行于服务器端，我们称之为 ServerSocket。

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听是指服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

客户端请求是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后再向服务器端套接字提出连接请求。

连接确认是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的信息发给客户端，一旦客户端确认了此信息，连接即可建立。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

使用套接字处理数据有两种基本模式：同步套接字和异步套接字。

1. 同步套接字

同步套接字的特点是在通过 Socket 进行连接、接收、发送操作时，客户机或服务器在接收到对方响应前会处于阻塞状态，即一直等到接收到对方请求时才继续执行下面的语句。可见，同步套接字适用于数据处理不太多的场合。当程序执行的任务很多时，长时间的等待可能会让用户无法忍受。

2. 异步套接字

在通过 Socket 进行连接、接收、发送操作时，客户机或服务器不会处于阻塞方式，而是利用 callback 机制进行连接、接收和发送处理，这样就可以在调用发送或接收的方法后直接返回，并继续执行下面的程序。可见，异步套接字特别适用于进行大量数据处理的场合。

使用同步套接字进行编程相对比较简单，而异步套接字则比较复杂。如果读者对套接字编程还不太熟悉，最好先掌握同步套接字编程方法。真正理解了同步套接字的编程思想后，再进一步学习异步套接字编程就不会感到太困难了。

1.3.1 Socket 类

Socket 类包含在 System.Net.Sockets 命名空间中。一个 Socket 实例包含了一个本地或者一个远程端点的套接字信息。Socket 类的构造函数为：

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

其中，addressFamily 为网络类型，指定 Socket 使用的寻址方案，例如 AddressFamily.InterNetwork 表明为 IP 版本 4 的地址；socketType 指定 Socket 的类型，例如 SocketType.Stream 表明连接是基于流套接字的，而 SocketType.Dgram 表示连接是基于数据报套接字的。protocolType 指定 Socket 使用的协议，例如 ProtocolType.Tcp 表明连接协议是 TCP 协议，而 ProtocolType.Udp 则表明连接协议是 UDP 协议。

Socket 构造函数的三个参数中，对于网络上的 IP 通信来说，AddressFamily 总是使用 AddressFamily.InterNetwork 枚举值。而 SocketType 参数则与 ProtocolType 参数配合使用，不允许其他的匹配形式，也不允许混淆匹配。表 1-1 列出了可用于 IP 通信的组合。

表 1-1　IP 套接字定义组合

SocketType	ProtocolType	说明
Dgram	Udp	无连接通信
Stream	Tcp	面向连接的通信
Raw	Icmp	Internet控制报文协议
Raw	Raw	简单IP包通信

熟悉了构造函数的参数含义，我们就可以创建套接字的实例：

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

套接字被创建后，我们就可以利用 Socket 类提供的一些属性方便的设置或检索信息。表 1-2 列出了常用的属性。

表 1-2　套接字常用属性

属性	说明
AddressFamily	获取套接字的Address family
Avilable	从网络中获取准备读取的数据数量
Blocking	获取或设置表示套接字是否处于阻塞模式的值
Connected	获取一个值，该值表明套接字是否与最后完成发送或接收操作的远程设备得到连接
LocalEndPoint	获取套接字的本地EndPoint对象
ProtocolType	获取套接字的协议类型
RemoteEndPoint	获取套接字的远程EndPoint对象
SocketType	获取套接字的类型

在实际应用中，还可以通过调用 Socket 对象的 SetSocketOption 方法设置套接字的各种选项，它有四种重载的形式：

```
public void SetSocketOption(SocketOptionLevel ol, SocketOptionName on, boolean value)
public void SetSocketOption(SocketOptionLevel ol, SocketOptionName on, byte[] value)
public void SetSocketOption(SocketOptionLevel ol, SocketOptionName on, int value)
public void SetSocketOption(SocketOptionLevel ol, SocketOptionName on, object value)
```

其中，ol 定义套接字选项的类型，可选类型有：IP、IPv6、Socket、Tcp、Udp。

on 指定套接字选项的值，表 1-3 列出了套接字常用的选项值。

表 1-3　套接字常用选项值

SocketOptionLevel	SocketOptionName	说明
IP	AddMembership	增加一个IP组成员
IP	HeaderIncluded	指出发送到套接字的数据将包括IP头
IP	IPOptions	指定IP选项插入到输出的数据包中
IP	MulticastInterface	设置组播包使用的接口
IP	MultiLoopBack	IP组播回送
IP	PacketInformation	返回关于接收包的信息
IP	UnBlockSource	设置套接字为无阻塞模式
Socket	AcceptConnection	如果为真，表明套接字正在侦听
Socket	Broadcast	如果为真，表明允许在套接字上发送广播消息
Socket	MaxConnections	设置使用的最大队列长度
Socket	PacketInformation	返回接收到的套接字信息
Socket	ReceiveBuffer	接收套接字的缓存大小
Socket	ReceiveTimeout	接收套接字的超时时间
Socket	SendBuffer	发送套接字的缓存大小
Socket	SendTimeout	发送套接字的超时时间
Socket	Type	获取套接字的类型
Socket	UseLookback	使用回传
Tcp	NoDelay	为发送合并禁用 Nagle 算法
Udp	ChecksumConverage	设置或获取 UDP 校验和覆盖
Udp	NoChecksum	发送校验和设置为零的 UDP 数据报

Value 参数指定所使用的套接字选项名 SocketOptionName 的值。例如：

```
socket.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.SendTimeout, 1000);
```

该语句设置套接字发送超时时间为 1000 毫秒。

1.3.2　面向连接的套接字

IP 连接领域有两种通信类型：面向连接的（connection-oriented）和无连接的（connectionless）。在面向连接的套接字中，使用 TCP 协议来建立两个 IP 地址端点之间的会话。一旦建立了这种连接，就可以在设备之间可靠的传输数据。为了建立面向连接的套接字，服务器和客户端必须分别进行编程，如图 1-7 所示。

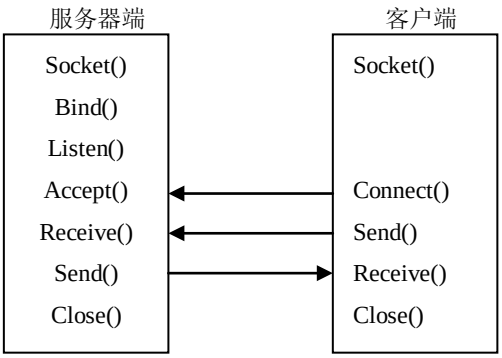


图1-7　面向连接的套接字编程

对于服务器端程序，建立的套接字必须绑定到用于 TCP 通信的本地 IP 地址和端口上。Bind 方法用于完成绑定工作：

```
Bind(IPEndPoint address)
```

Address 为 IPEndPoint 的实例，该实例包括一个本地 IP 地址和一个端口号。在套接字绑定到本地之后，就用 Listen 方法等待客户机发出的连接尝试：

```
Listen(int backlog)
```

Backlog 参数指出系统等待用户程序服务排队的连接数，超过连接数的任何客户都不能与服务器进行通信。

在 Listen 方法执行之后，服务器已经做好了接收任何引进连接的准备，这是用 Accept 方法来完成的，当有新客户进行连接时，该方法就返回一个新的套接字描述符。下面是完成上述步骤的服务器端部分代码的例子：

```
IPHostEntry local = Dns.GetHostByName(Dns.GetHostName());
IPEndPoint iep = new IPEndPoint(local.AddressList[0], 80);
Socket localSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
localSocket.Bind(iep);
locatSocket.Listen(10);
Socket clientSocket = localSocket.Accept();
```

程序执行到 Accept 方法时会处于阻塞状态，直到有客户机请求连接，一旦有客户机连接到服务器，clientSocket 对象将包含该客户机的所有连接信息。而 localSocket 对象仍然绑定到原来的 IPEndPoint 对象，并可以通过增加循环语句继续用 Accept 方法接收新的客户端连接。如果没有继续调用 Accept 方法，服务器就不会再响应任何新的客户机连接。

在接受客户机连接之后，客户机和服务器就可以开始传递数据了。表 1-4 和 1-5 分别列出了可用的 Receive 方法和 Send 方法的格式以及 SocketFlag 的值。

表 1-4　Receive 方法和 Send 方法

方法	说明
Receive(byte[] data)	接收数据放到接收缓冲器中
Receive (Generic IList data)	接收数据放到接收缓冲器列表中
Receive(byte[] data, SocketFlgs sf)	接收数据放到接收缓冲器中，并设置套接字标志
Receive (Generic IList data, SocketFlags sf)	接收数据放到接收缓冲器列表中，并设置套接字标志
Receive(byte[] data, int size, SocketFlgs sf)	接收指定容量的数据放到接收缓冲器中，并设置套接字标志
Receive(byte[] data, int offset, int size, SocketFlgs sf)	接收一定容量的数据放到接收缓冲器中指定偏移量位置，并设置套接字标志

Send(byte[] data)	向一个已连接的套接字发送数据
Socket.Send (Generic IList data)	向一个已连接的套接字发送在列表中的磁盘缓存块数
Send (byte[] data, SocketFlgs sf)	向一个已连接的套接字发送数据，并设置套接字标志
Socket.Send (Generic IList data, SocketFlags sf)	向一个已连接的套接字发送在列表中的磁盘缓存块数，并设置套接字标志
Send (byte[] data, int size, SocketFlgs sf)	发送一定容量的指定类型的数据，并设置套接字标志
Send (byte[] data, int offset, int size, SocketFlgs sf)	发送从某处开始的一定容量的指定类型的数据，并设置套接字标志

表 1-5　SocketFlag 的值

值	说明
DontRoute	不用内部路由表发送数据
MaxIOVectorLength	给用于发送和接收数据的WSABUF结构数提供一个标准值
None	对这次调用不使用标志
OutOfBind	处理带外的数据
Partial	部分的发送或接收信息
Peek	只对进入的消息取数

对于客户端程序，客户机也必须把一个地址绑定到创建的 Socket 对象，不过它不使用 Bind 方法，而是使用 Connect 方法：

```
IPAddress remoteHost = IPAddress.Parse("192.168.0.1");
PEndPoint iep = new IPEndPoint(remoteHost, 80);
Socket localSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
localSocket.Connect(iep);
```

程序运行后，客户端在与服务器建立连接之前，系统不会执行 Connect 方法下面的语句，而是处于阻塞方式。一旦客户端与服务器建立连接，客户机就可以像服务器收发数据使用的方法一样，使用 Send 和 Receive 方法进行通信。注意通信完成后，必须先用 Shutdown 方法停止会话，然后关闭 Socket 实例。表 1-6 说明了 Socket.Shutdown 方法可以使用的值。

表 1-6　Socket.ShutDown 值

值	说明
SocketShutdown.Receive	防止在套接字上接收数据，如果收到额外的数据，将发出一个RST信号
SocketShutdown.Send	防止在套接字上发送数据，在所有存留在缓冲器中的数据发送出去之后，发出一个FIN信号
SocketShutdown.Both	在套接字上既停止发送也停止接收

下面是关闭连接的典型用法：

```
sock.Shutdown(SocketShutdown.Both);
sock.Close();
```

该方法允许 Socket 对象一直等待，直到将内部缓冲区的数据发送完为止。

1.3.3　无连接的套接字

UDP 协议使用无连接的套接字，无连接的套接字不需要在网络设备之间发送连接信息。因此，很难确定谁是服务器谁是客户机。如果一个设备最初是在等待远程设备的信息，则套接字就必须用 Bind 方法绑定到一个本地地址/端口对上。完成绑定之后，该设备就可以利用套接字接收数据了。由于发送设备没有建立到接收设备地址的连接，所以收发数据均不需要 Connect 方法。图 1-8 为无连接套接字编程示意图。

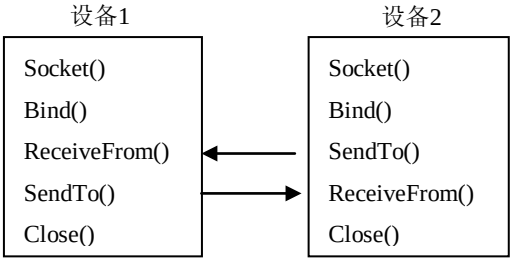


图1-8　无连接套接字编程

由于不存在固定的连接，所以可以直接使用 SendTo 方法和 ReceiveFrom 方法发送和接收数据，在两个设备之间的通信结束之后，可以像 TCP 中使用的方法一样，对套接字使用 Shutdown 和 Close 方法。

再次提醒读者注意，需要接收数据时，必须使用 Bind 方法将套接字绑定到一个本地地址/端口对上之后才能使用 ReceiveFrom 方法接收数据，如果只发送而不接收，则不需要使用 Bind 方法。

实际上，为了简化复杂的网络编程，.NET Framework 除了提供可以灵活控制的套接字类以外，还在此基础上提供了对套接字封装后的基于不同协议的更易于使用的类。至于封装后的类的具体用法，我们将在后面的章节中逐步学习。

1.4　网络流

流（stream）是对串行传输的数据的一种抽象表示，底层的设备可以是文件、外部设备、主存、网络套接字等等。

流有三种基本的操作：写入、读取和查找。

如果数据从内存缓冲区传输到外部源，这样的流叫作“写入流”。

如果数据从外部源传输到内存缓冲区，这样的流叫作“读取流”。

在网络上传输数据时，使用的是网络流（NetworkStream）。网络流的意思是数据在网络的各个位置之间是以连续的形式传输的。为了处理这种流，C#在 System.Net.Sockets 命名空间中提供了一个专门的 NetworkStream 类，用于通过网络套接字发送和接收数据。

NetworkStream 类支持对网络数据的同步或异步访问，它可以被视为在数据来源端和接收端之间架设了一个数据通道，这样我们读取和写入数据就可以针对这个通道来进行。

对于 NetworkStream 流，写入操作是指从来源端内存缓冲区到网络上的数据传输；读取操作是从网络上到接收端内存缓冲区（如字节数组）的数据传输。如图 1-9 所示。

图1-9　NetworkStream流的数据传输

构造 NetworkStream 对象的常用形式为：

```
Socket socket=new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
NetWorkStream networkStream=new NetworkStream(socket);
```

一旦构造了一个 NetworkStream 对象，就不需要使用 Socket 对象了。也就是说，在关闭

网络连接之前就一直使用 NetworkStream 对象发送和接收网络数据。表 1-7 列出了 NetworkStream 类提供的常用属性。

表 1-7　NetworkStream 类的常用属性

属性	说明
CanRead	指示NetworkStream是否支持读操作，默认值为True
CanWrite	指示NetworkStream是否支持写操作，默认值为True
CanSeek	指示NetworkStream流是否支持查找，该属性总是返回False
DataAvailable	指示NetworkStream上是否有可用的数据，有则为真
Position	获取或设置流中的当前位置，此属性始终引发NotSupportedException
Readable	指示NetworkStream流是否可读，为真时可读；假时不可读
Writeable	指示NetworkStream流是否可写，为真时可写；假时不可写

在这个表中，比较常用的一个属性就是 DataAvailable,通过这个属性，可以迅速查看在缓冲区中是否有数据等待读出。

注意：网络流没有当前位置的概念，因此不支持查找和对数据流的随机访问，相应属性 CanSeek 始终返回 false，而读取 Position 属性和调用 Seek 方法时，都将引发 NotSupportedException 异常。

表 1-8 列出了 NetworkStream 类的常用方法。

表 1-8　NetworkStream 类的常用方法

方法	说明
BeginRead方法	从NetworkStream流开始异步读取
BeginWrite方法	开始向NetworkStream流异步写入
EndRead方法	结束对一个NetworkStream流的异步读取
EndWrite方法	结束向一个NetworkStream流的异步写入
Read方法	从NetworkStream流中读取数据
Write方法	向NetworkStream流中写入数据
ReadByte方法	从NetworkStream流中读取一个字节的数据
WriteByte方法	向NetworkStream流中写入一个字节的数据
Flush方法	从NetworkStream流中取走所有数据
Close方法	关闭NetworkStream对象
Dispose方法	释放NetworkStream占用的资源
Seek方法	查找NetworkStream流的当前位置，此方法将引发NotSupportedException

网络数据传输完成后，不要忘记用 Close 方法关闭 NetworkStream 对象。

1.5　习题 1

1. 简要回答下列问题。
   1) 进程和线程有什么区别？
   2) 线程是如何创建的？怎样设置线程的优先级？
   3) 什么是套接字？

2. 使用 DNS 类和 IPHostEntry 类创建一个如图 1-10 所示的域名解析器。用户输入主机名或者 DNS 域名以后，能在下面的列表框中显示与主机或者 DNS 域名对应的 IP 地址和别名。

图1-10　第2题设计界面

第 2 章　TCP 应用编程

TCP 是 Transmission Control Protocol（传输控制协议）的简称，是 TCP/IP 体系中面向连接的运输层协议，在网络中提供全双工的和可靠的服务。一旦通信双方建立了 TCP 连接，连接中的任何一方都能向对方发送数据和接收对方发送来的数据。发送数据时，程序员可以通过程序不断将数据流陆续写入 TCP 的发送缓冲区中，然后 TCP 自动从发送缓冲区中取出一定数量的数据，将其组成 TCP 报文段逐个发送给 IP 层，再通过 IP 层发送出去。接收端从 IP 层接收到 TCP 报文段后，将其暂时保存在接收缓冲区中，这时程序员就可以通过程序依次读取接收缓冲区中的数据，从而达到相互通信的目的。

TCP 协议最主要的特点是：

1) 是一种基于连接的协议。

2) 保证数据准确到达。

3) 保证各数据到达的顺序与数据发出的顺序相同。

4) 传输的数据无消息边界。

利用 TCP 协议开发应用程序时，.NET 框架提供有两种工作方式，一种是同步工作方式，另一种是异步工作方式。

同步工作方式是指利用 TCP 协议进行编程时程序执行到发送、接收和监听语句时，在未完成工作前不再继续往下执行，即处于阻塞状态，直到该语句完成某个工作后才继续执行下一条语句；异步工作方式是指程序执行到发送、接收和监听语句时，不论工作是否完成，都会继续往下执行。例如对于接收数据来说，在同步工作方式下，接收方执行到接收语句后将处于阻塞方式，只有接收到对方发来的数据后才继续执行下一条语句；而如果采用异步工作方式，则在程序执行到接收语句后，无论接收方是否接收到对方发来的数据，程序都会继续往下执行。

与同步工作方式和异步工作方式相对应，利用 Socket 类进行编程时系统也都提供有相应的方法，采用相应的方法进行编程分别称为同步套接字编程和异步套接字编程。但是使用套接字编程比较复杂，涉及到很多底层的细节。为了简化套接字编程，.NET 框架又专门提供了两个类：TcpClient 类与 TcpListener 类。由于这两个类与套接字一样也分别有各自的同步和异步工作方式及其对应的方法，而我们在编程时，三个类都有可能使用，因此为简化起见，无论使用的是哪个类，我们统统从工作方式上将其称为同步 TCP 和异步 TCP，所以其编程方式也有两种，一种是同步 TCP 编程，另一种是异步 TCP 编程。

注意这里的同步 TCP 和异步 TCP 仅仅指工作方式，它和线程间的同步不是一个概念。线程间的同步指不同线程或不同线程使用的某些资源具有先后关联的关系，它决定着逻辑执行的顺序。比如有 A 和 B 两个资源，实际应用中要求只有对资源 A 处理后才能处理资源 B，就说 A 和 B 存在同步关系。如果执行顺序不正确，变为先处理资源 B 再处理资源 A，得到的结果就是错误的。所以，线程间的同步主要关注的是一种逻辑关系。而同步 TCP 和异步 TCP 则仅仅指 TCP 协议编程中采用那种工作方式而言，即是从执行到发送、接收和监听语句时，程序是否继续往下执行这个角度来说的。

从逻辑关系上看，无论是同步 TCP 应用编程还是异步 TCP 应用编程，在实际应用中既可能要求不同线程间的同步，也可能不要求同步。

2.1 同步 TCP 应用编程

不论是多么复杂的 TCP 应用程序，双方通信的最基本前提就是客户端要先和服务器端进行 TCP 连接，然后才可以在此基础上相互收发数据。由于服务器需要对多个客户端同时服务，因此程序相对复杂一些。在服务器端，程序员需要编写程序不断的监听客户端是否有连接请求，并通过套接字区分是哪个客户；而客户端与服务器连接则比较简单，只需要指定连接的是哪个服务器即可。一旦双方建立了连接并创建了对应的套接字，就可以相互收发数据了。在程序中，发送和接收数据的方法都是一样的，区别仅是方向不同。

在同步 TCP 应用编程中，发送、接收和监听语句均采用阻塞方式工作。使用同步 TCP 编写服务器端程序的一般步骤为：

1) 创建一个包含采用的网络类型、数据传输类型和协议类型的本地套接字对象，并将其与服务器的 IP 地址和端口号绑定。这个过程可以通过 Socket 类或者 TcpListener 类完成。

2) 在指定的端口进行监听，以便接受客户端连接请求。

3) 一旦接受了客户端的连接请求，就根据客户端发送的连接信息创建与该客户端对应的 Socket 对象或者 TcpClient 对象。

4) 根据创建的 Socket 对象或者 TcpClient 对象，分别与每个连接的客户进行数据传输。

5) 根据传送信息情况确定是否关闭与对方的连接。

使用同步 TCP 编写客户端程序的一般步骤为：

1) 创建一个包含传输过程中采用的网络类型、数据传输类型和协议类型的 Socket 对象或者 TcpClient 对象。

2) 使用 Connect 方法与远程服务器建立连接。

3) 与服务器进行数据传输。

4) 完成工作后，向服务器发送关闭信息，并关闭与服务器的连接。

为了让读者大概了解套接字编程和封装后的 TcpClient 及 TcpListener 的区别，下面我们分别对三者在程序中实现的关键代码做一个简单介绍。

2.1.1 使用套接字发送和接收数据

在网络中，数据是以字节流的形式进行传输的。服务器与客户端双方建立连接后，程序中需要先将要发送的数据转换为字节数组，然后使用 Socket 对象的 Send 方法发送数据，或者使用 Receive 方法接收数据。注意，要发送的字节数组并不是直接发送到了远程主机，而是发送到了本机的 TCP 发送缓冲区中；同样道理，接收数据也是如此，即程序中是从 TCP 接收缓冲区接收数据。可以使用 Socket 类的 SendBufferSize 属性获取或者设置发送缓冲区的大小，使用 ReceiveBufferSize 属性获取或者设置接收缓冲区的大小，也可以使用其默认大小。至于系统什么时候将缓冲区数据通过网络发送到远程主机，受到哪些因素的影响，就不需要在程序中考虑了。

1. 服务器端编程关键代码

在服务器端程序中，使用套接字收发数据的关键代码为：

```
using System.Net;
```

```
using System.Net.Sockets;
……
IPAdress ip=IPAdress.Prase("服务器 IP 地址");
IPEndPoint iep=new IPEndPoint (ip,可用端口号);
Socket socket=new Socket(AdressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
socket.Bind(iep);
socket.Listen(最大客户端连接数);
Socket clientSocket=socket.Accept();
……
//通过 clientSocket 向该客户发送数据
string message = "发送的数据";
byte[] sendbytes = System.Text.Encoding.UTF8.GetBytes(message);
int successSendBytes = clientSocket.Send(sendbytes , sendbytes.Length , SocketFlags.None );
……
//通过 clientSocket 接收该客户发来的数据
byte[] receivebytes = new byte [1024];
int successReceiveBytes = clientSocket.Receive(receivebytes);
```

由于 TCP 协议是面向连接的，因此在发送数据前，程序首先应该将套接字与本机 IP 地址和端口号绑定，并使之处于监听状态，然后通过 Accept 方法监听是否有客户端连接请求。使用套接字是为了指明使用哪种协议；和本机绑定是为了在指定的端口进行监听，以便识别客户端连接信息；调用 Accept 方法的目的是为了得到对方的 IP 地址、端口号等套接字需要的信息。因为只有得到对方的 IP 地址和端口号等相关信息后，才能与对方进行通信。当程序执行到 Accept 方法时，会处于阻塞状态，直到接收到客户端到服务器端的连接请求才继续执行下一条语句。服务器一旦接受了该客户端的连接，Accept 方法就返回一个与该客户端通信的新的套接字，套接字中包含了对方的 IP 地址和端口号，然后就可以用返回的套接字和该客户进行通信了。

Send 方法的整型返回值表示成功发送的字节数。正如本节开始所说的那样，该方法并不是把要发送的数据立即传送到网络上，而是传送到了 TCP 发送缓冲区中。但是，在阻塞方式下，如果由于网络原因导致原来 TCP 发送缓冲区中的数据还没有来得及发送到网络上，接收方就无法继续接收发送给它的所有字节数，因此该方法返回实际上成功向 TCP 发送缓冲区发送了多少字节。

即使不是因为网络原因，也不能保证数据一定能一次性全部传送到了 TCP 发送缓冲区。这是因为 TCP 发送缓冲区一次能接收的数据取决于其自身的大小，也就是说，Send 方法要发送的数据如果超过了 TCP 发送缓冲区的有效值，那么调用一次 Send 方法就不能将数据全部成功发送到缓存中。所以，实际编写程序时，程序中应该通过一个循环进行发送，并检测成功发送的字节数，直到数据全部成功发送完毕为止。当然，如果 Send 方法中发送的数据小于 TCP 发送缓冲区的有效值，调用一次 Send 方法就可能全部发送成功。

与发送相反，Receive 方法则是从 TCP 接收缓冲区接收数据，Receive 方法的整型返回值表示实际接收到的字节数，但是如果远程客户端关闭了套接字连接，而且此时有效数据已经被完全接收，那么 Receive 方法的返回值将会是 0 字节。

但有一点需要注意，如果 TCP 接收缓冲区内没有有效的数据可读时，在阻塞模式下，Receive 方法将会被阻塞；但是在非阻塞模式下，Receive 方法将会立即结束并抛出套接字异常。要避免这种情况，我们可以使用 Available 属性来预先检测数据是否有效，如果 Available 属性

值不为 0，那么就可以重新尝试接收操作。

2. 客户端编程关键代码

对于客户端程序，只需要通过服务器的 IP 地址与端口号与服务器建立连接，一旦连接成功，就可以通过套接字与服务器相互传输数据，关键代码为：

```
IPAddress ip=IPAddress.Parse("服务器 IP 地址");
IPEndPoint iep=new IPEndPoint(ip,服务器监听端口号);
Socket serverSocket=new Socket(AddressFamily.InterNetwork, SocketType.Stream,ProtocolType.Tcp);
serverSocket.Connect(iep);
……
//通过 serverSocket 向服务器发送数据
string message = "发送的数据";
byte[] sendbytes = System.Text.Encoding.UTF8.GetBytes(message);
int successSendBytes = serverSocket.Send(sendbytes , sendbytes.Length , SocketFlags.None );
……
//通过 serverSocket 接收服务器发来的数据
byte[] receivebytes = new byte [1024];
int successReceiveBytes = serverSocket.Receive(receivebytes);
```

可见，对客户端程序来说，收发数据的方法和服务器端使用的收发数据的方法相似，区别只是创建的套接字不同。

2.1.2 使用 NetworkStream 对象发送和接收数据

NetworkStream 对象专门用于对网络流数据进行处理。创建了 NetworkStream 对象后，就可以直接使用该对象接收和发送数据。例如：

```
NetworkStream networkStream = new NetworkStream(clientSocket );
……
//发送数据
string message = "发送的数据";
byte[] sendbytes = System.Text.Encoding.UTF8.GetBytes(message );
networkStream.Write(sendbytes ,0, sendbytes.Length );
……
//接收数据
byte[] readbytes = new byte[1024];
int i = networkStream.Read(readbytes, 0, readbytes.Length);
```

与套接字的 Send 方法不同，NetworkStream 对象的 Write 方法的返回值为 void，之所以不返回实际发送的字节数，是因为 Write 方法能保证字节数组中的数据全部发送到 TCP 发送缓冲区中，避免了使用 Socket 类的 Send 方法发送数据时所遇到的调用一次不一定能全部发送成功的问题，从而在一定程度上简化了编程工作量。但是所有的这一切操作必须在 NetworkStream 对象的 Writeable 属性值有效时才行，因此在使用 NetworkStream 对象的 Write 方法前应该检测 NetworkStream 对象的 Writeable 属性是否为 True。

如果发送的全部是单行文本信息，创建 NetworkStream 对象后，使用 StreamReader 类和 StreamWriter 类的 ReadLine 和 WriteLine 方法更简单，而且也不需要编程进行字符串和数组之间的转换。

与 Write 方法相对应，调用 NetworkStream 类的 Read 方法前应确保 NetworkStream 对象的 CanRead 属性值有效。在此前提下，该方法一次将所有的有效数据读入到接收缓冲变量中，并返回成功读取的字节数。

注意，Read 方法之所以也有一个整型的返回值，是因为有可能 TCP 接收缓冲区还没有接收到对方发送过来的指定长度的数据。也就是说，接收到的数据可能没有指定的那么多。在后面的内容中，我们将进一步学习解决这个问题的方法。

在 Read 方法中，有一点与 Socket 类的 Receive 方法类似，即如果远程主机关闭了套接字连接，并且此时有效数据已经被完全接收，那么 Read 方法的返回值将会是 0 字节。

2.1.3 TcpClient 与 TcpListener 类

在 System.Net.Sockets 命名空间下，TcpClient 类与 TcpListener 类是两个专门用于 TCP 协议编程的类。这两个类封装了底层的套接字，并分别提供了对 Socket 进行封装后的同步和异步操作的方法，降低了 TCP 应用编程的难度。

TcpClient 类用于连接、发送和接收数据，TcpListener 类则用于监听是否有传入的连接请求。

1. TcpClient 类

TcpClient 类归类在 System.Net 命名空间下。利用 TcpClient 类提供的方法，可以通过网络进行连接、发送和接收网络数据流。该类的构造函数有四种重载形式：

1) TcpClient()

该构造函数创建一个默认的 TcpClient 对象，该对象自动选择客户端尚未使用的 IP 地址和端口号。创建该对象后，即可用 Connect 方法与服务器端进行连接。例如：

```
TcpClient tcpClient=new TcpClient();
tcpClient.Connect("www.abcd.com", 51888);
```

2) TcpClient(AddressFamily family)

该构造函数创建的 TcpClient 对象也能自动选择客户端尚未使用的 IP 地址和端口号，但是使用 AddressFamily 枚举指定了使用哪种网络协议。创建该对象后，即可用 Connect 方法与服务器端进行连接。例如：

```
TcpClient tcpClient = new TcpClient(AddressFamily.InterNetwork);
tcpClient.Connect("www.abcd.com", 51888);
```

3) TcpClient(IPEndPoint iep)

iep 是 IPEndPoint 类型的对象，iep 指定了客户端的 IP 地址与端口号。当客户端的主机有一个以上的 IP 地址时，可使用此构造函数选择要使用的客户端主机 IP 地址。例如：

```
IPAddress[] address = Dns.GetHostAddresses(Dns.GetHostName());
IPEndPoint iep = new IPEndPoint(address[0], 51888);
TcpClient tcpClient = new TcpClient(iep);
tcpClient.Connect("www.abcd.com", 51888);
```

4) TcpClient(string hostname,int port)

这是使用最方便的一种构造函数。该构造函数可直接指定服务器端域名和端口号，而且不需使用 connect 方法。客户端主机的 IP 地址和端口号则自动选择。例如：

```
TcpClient tcpClient=new TcpClient("www.abcd.com", 51888);
```

表 2-1 和表 2-2 分别列出了 TcpClient 类的常用属性和方法。

表 2-1　TcpClient 类的常用属性

属性	含义
Client	获取或设置基础套接字

LingerState	获取或设置套接字保持连接的时间
NoDelay	获取或设置一个值，该值在发送或接收缓冲区未满时禁用延迟
ReceiveBufferSize	获取或设置Tcp接收缓冲区的大小
ReceiveTimeout	获取或设置套接字接收数据的超时时间
SendBufferSize	获取或设置Tcp发送缓冲区的大小
SendTimeout	获取或设置套接字发送数据的超时时间

表 2-2　TcpClient 类的常用方法

方法	含义
Close	释放TcpClient实例，而不关闭基础连接
Connect	用指定的主机名和端口号将客户端连接到TCP主机
BeginConnect	开始一个对远程主机连接的异步请求
EndConnect	异步接受传入的连接尝试
GetStream	获取能够发送和接收数据的NetworkStream对象

2. TcpListener 类

TcpListener 类用于监听和接收传入的连接请求。该类的构造函数有：

1) TcpListener(IPEndPoint iep)

其中 iep 是 IPEndPoint 类型的对象，iep 包含了服务器端的 IP 地址与端口号。该构造函数通过 IPEndPoint 类型的对象在指定的 IP 地址与端口监听客户端连接请求。

2) TcpListener(IPAddress localAddr, int port)

建立一个 TcpListener 对象，在参数中直接指定本机 IP 地址和端口，并通过指定的本机 IP 地址和端口号监听传入的连接请求。

构造了 TcpListener 对象后，就可以监听客户端的连接请求了。与 TcpClient 相似，TcpListener 也分别提供了同步和异步方法，在同步工作方式下，对应有 AcceptTcpClient 方法、AcceptSocket 方法、Start 方法和 Stop 方法。

AcceptSocket 方法用于在同步阻塞方式下获取并返回一个用来接收和发送数据的套接字对象。该套接字包含了本地和远程主机的 IP 地址与端口号，然后通过调用 Socket 对象的 Send 和 Receive 方法和远程主机进行通信。

AcceptTcpClient 方法用于在同步阻塞方式下获取并返回一个可以用来接收和发送数据的封装了 Socket 的 TcpClient 对象。

Start 方法用于启动监听，构造函数为：

public void Start(int backlog)

整型参数 backlog 为请求队列的最大长度，即最多允许的客户端连接个数。Start 方法被调用后，把自己的 LocalEndPoint 和底层 Socket 对象绑定起来，并自动调用 Socket 对象的 Listen 方法开始监听来自客户端的请求。如果接受了一个客户端请求，Start 方法会自动把该请求插入请求队列，然后继续监听下一个请求，直到调用 Stop 方法停止监听。当 TcpListener 接受的请求超过请求队列的最大长度或小于 0 时，等待接受连接请求的远程主机将会抛出异常。

Stop 方法用于停止监听请求，构造函数为：

public void Stop()

程序执行 Stop 方法后，会立即停止监听客户端连接请求，并关闭底层的 Socket 对象。等

待队列中的请求将会丢失，等待接受连接请求的远程主机会抛出套接字异常。

2.1.4 解决 TCP 协议的无消息边界问题

虽然采用 TCP 协议通信时，接收方能够按照发送方发送的顺序接收数据，但是在网络传输中，可能会出现发送方一次发送的消息与接收方一次接收的消息不一致的现象。例如：发送方第一次发送的字符串数据为“12345”，第二次发送的字符串数据为“abcde”；正常情况下，接收方接收的字符串应该是第一次接收：“12345”，第二次接收：“abcde”。

但是，当收发信息速度非常快时，接收方也可能一次接收到的内容就是“12345abcde”，即两次或者多次发送的内容一起接收。

还有一种最极端的情况，就是接收方可能会经过多次才能接收到发送方发送的消息。例如第一次接收到“1234”，第二次为“45ab”,第三次为“cde”。

这主要是因为 TCP 协议是字节流形式的、无消息边界的协议，由于受网络传输中的不确定因素的影响，因此不能保证单个 Send 方法发送的数据被单个 Receive 方法读取。

如果需要解析发送方发送的命令，为了保证接收方不出现解析错误，编程时必须要考虑消息边界问题，否则就可能会出现丢失命令等错误结果。例如对于两次发送的消息一次全部接收的情况，虽然接收的内容并没有少，但是如果在程序中认为每次接收的都是一条命令，就会丢失另一条命令，从而引起逻辑上的错误。就像网络象棋程序，丢失了一步，整个逻辑关系就全乱套了。

实际应用中，解决 TCP 协议消息边界问题的方法有三种：

第一种方法是发送固定长度的消息。该方法适用于消息长度固定的场合。

具体实现时，可以通过 System.IO 命名空间下的 BinaryReader 对象每次向网络流发送一个固定长度的数据，例如每次发送一个 int 类型的 32 位整数。BinaryReader 和 BinaryWriter 对象提供了多种重载方法，发送和接收具有固定长度类型的数据非常方便。

第二种方法是将消息长度与消息一起发送。例如本机在每次发送的消息前面用 4 个字节表明本次消息的长度，然后将包含消息长度的消息发送到远程主机；远程主机接收到消息后，首先从消息的头 4 个字节获取消息长度，然后根据消息长度值依次循环接收发送方发送的消息数据。这种方法可以适用于任何场合。

BinaryReader 对象和 BinaryWriter 对象同样是实现这种方法的最方便的途径。BinaryWriter 对象提供了很多重载的 Write 方法，可以适用于任何场合。例如向网络流写入字符串时，该方法会自动计算出字符串占用的字节数，并使用 4 个字节作为字符串前缀将其附加到字符串的前面；接收方使用 ReadString 方法接收字符串时，它会首先读取字符串前缀，然后自动根据字符串前缀指定的长度读取字符串。

如果是二进制的流数据，比如使用 TCP 协议通过网络传递二进制文件，需要在程序中通过代码实现计算长度的功能。本书网络数据加密与解密一章中，通过网络传递加密数据的例子中解决边界问题使用的就是这种方法。

第三种方法是使用特殊标记分隔消息。例如用回车换行（\r\n）作为分隔符。这种方法主要用于消息中不包含特殊标记的场合。

对于字符串处理，实现这种方法最方便的途径是通过 StreamWriter 对象和 StreamReader 对象。发送方每次使用 StreamWriter 对象的 WriteLine 方法将发送的字符串写入网络流中，接

收方每次只需要用 StreamReader 对象的 ReadLine 方法将用回车换行作为标记的字符串从网络流中读出即可。本章的例子均使用这种方法。

三种处理消息边界的方法各有优缺点，也不能说哪种方式好哪种方式不好。编程时应该根据实际情况选择一种合适的方法。

2.2 利用同步 TCP 编写网络游戏

在流行的网络应用编程中，利用 TCP 协议编写的程序非常多，例如网络游戏、网络办公、股票交易、网络通信等等。本节通过编写一个稍微复杂的“吃棋子”游戏，说明利用 TCP 协议和同步套接字以及多线程编写流行的网络应用程序的方法。

【例 2-1】编写一个可以通过因特网对弈的“吃棋子”游戏。游戏功能要求：

1）服务器可以同时服务多桌，每桌允许两个玩家通过因特网对弈。

2) 允许玩家自由选择坐在哪一桌的哪一方。如果两个玩家坐在同一桌，双方应都能看到对方的状态。两个玩家均单击“开始”按钮，游戏就开始了。

3) 某桌游戏开始后，服务器以固定的时间间隔同时在 15×15 的棋盘方格内向该桌随机地发送黑白两种颜色的棋子位置，客户端程序接收到服务器发送的棋子位置和颜色后，在 15×15 棋盘的相应位置显示棋子。

4) 玩家坐到游戏桌座位上后，不论游戏是否开始，该玩家都可以随时调整服务器发送棋子的时间间隔。

5) 游戏开始后，客户端程序响应鼠标单击。每当玩家单击了某个棋子，该棋子就会从棋盘上消失，同时具有相应颜色的玩家得 1 分。注意，如果玩家单击了对方颜色的棋子，则对方得 1 分。

6) 如果两个相同颜色的棋子在水平方向或垂直方向是相邻的，那么就认为这两个棋子是相邻的。这里不考虑对角线相邻的情况。如果相同颜色的棋子出现在相邻的位置，游戏就结束了。该颜色对应的玩家就是失败者。

7) 同一桌的两个玩家可以聊天。

这个游戏虽然比较简单，但却是“麻雀虽小、五脏俱全”。如果读者真正理解了编写的方法，就可以轻松编写各类基于 C/S 模式的因特网应用程序。

2.2.1 服务器端编程

根据游戏的要求，服务器需要提供一个“吃棋子”游戏室，游戏室内可以根据需要开辟多个小房间，每个小房间内有一个游戏桌，游戏桌两边各有一个座位，分别用编号 0 和 1 表示，0 代表黑方，1 代表白方。每个小房间内最多只允许两个玩家，即不提供旁观功能。

玩家进入游戏室后，应该可以看到各个小房间的游戏桌两边是否有人的情况，而且可以决定是否坐到某个座位上。坐到座位上后，才能看到游戏桌上的棋盘。玩家也可以随时离开座位，回到游戏室。

服务器启动服务后，需要创建一个线程专门用于监听玩家的连接请求。

在监听线程中，服务器一旦接受一个连接，就创建一个与该玩家对应的线程，用于接收该玩家发送的信息，并根据该玩家发送的信息提供相应的服务。有多少个玩家连接，就创建多少

个对应的线程。玩家退出游戏室，其对应的线程自动终止。

当然，玩家越多，需要创建的线程也越多。因此，服务器必须限制进入游戏室的玩家数量，具体可以同时进入多少玩家，可以根据服务器内存容量以及服务器运行速度决定。

在与每个玩家对应的线程中，服务器收到对应玩家发送的字符串信息后，需要解析字符串的含义，并决定服务器需要的操作。字符串分为命令部分和参数部分，命令部分和参数部分之间以及参数部分的参数之间均以逗号分隔。客户端发送给服务器的命令规定为：

- Login：玩家请求进入游戏室。服务器接受连接后，检查游戏室总人数，如果人数已满，回送“Sorry”命令；否则回送“Tables”命令，并在参数中指明各小房间的游戏桌情况，每桌有两个座，第一座表示黑方，第二座表示白方，每座的 0 表示无人，1 表示有人。
- Logout：玩家退出游戏室。服务器收到此命令后，清除该玩家对应的信息，结束与该玩家对应的线程。
- SitDown：玩家坐到某个小房间的游戏桌上。服务器接收到此命令后，将该玩家的昵称、TcpClient 对象、桌号、座位号保存到对应的数组中，并将该玩家入座的信息同时告诉该游戏桌的两个玩家。另外，由于座位数已经发生变化，所以还需要将各桌是否有人的情况同时发送给连接的所有玩家。
- GetUp：玩家离开小房间的游戏桌座位，回到游戏室。服务器收到此信息后，需要停止向该桌发送棋子位置及颜色信息，并作离座处理。同时，由于坐到座位上的人数已经发生变化，还要将各桌是否有人的情况发送给在线的所有玩家。
- Level：设置难度级别。本游戏共有五级难度，级别越高，发送棋子的速度就越快。服务器收到此信息后，直接设置发送棋子的时间间隔即可。
- Talk：表示客户发出的谈话内容。由于谈话中可能包括逗号，因此需要单独处理。服务器收到此信息后，直接将接收的信息同时发送给两个玩家。
- Start：表示该玩家已经单击了【开始】按钮，此时服务器需要检测双方是否都单击了【开始】按钮，如果双方都开始了，就连续向双方发送随机产生的棋子位置及颜色信息。
- UnsetDot：玩家单击了棋盘上的棋子。服务器收到此信息后，将棋盘上对应的位置设置为无棋子标记，同时修改对应的成绩，并将结果发送给两个玩家。

具体实现步骤为：

(1) 创建一个名为 GameServer 的 Windows 应用程序，将 Form1.cs 换名为 FormServer.cs，设计界面如图 2-1 所示。

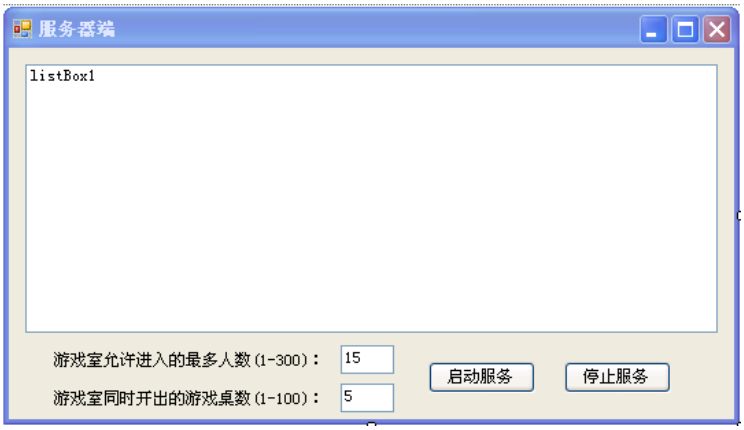


图2-1　吃棋子游戏服务器端界面设计

(2) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 User.cs，用于保存每个玩家的基本信息。代码如下：

```
//-------------User.cs----------------//
using System.Net.Sockets;
using System.IO;
namespace GameServer
{
    class User
    {
        public TcpClient client;
        public StreamReader sr;
        public StreamWriter sw;
        public string userName;
        public User(TcpClient client)
        {
            this.client = client;
            this.userName = "";
            NetworkStream netStream = client.GetStream();
            sr = new StreamReader(netStream, System.Text.Encoding.UTF8);
            sw = new StreamWriter(netStream, System.Text.Encoding.UTF8);
        }
    }
}
```

(3) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 Player.cs，用于保存已经坐到游戏桌座位上玩家的情况。代码如下：

```
//---------------Player.cs-------------//
using System;
namespace GameServer
{
    class Player
    {
        public User user;        //User 类的实例
        public bool started;   //是否已经开始
        public int grade;        //成绩
```

```
        public bool someone;   //是否有人坐下
        public Player()
        {
            someone = false;
            started = false;
            grade = 0;
            user = null;
        }
    }
}
```

(4) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 Service.cs，用于提供公用的方法。代码如下：

```
//---------------Service.cs-------------------//
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.IO;
namespace GameServer
{
    class Service
    {
        private ListBox listbox;
        //用于线程间互操作
        private delegate void SetListBoxCallback(string str);
        private SetListBoxCallback setListBoxCallback;
        public Service(ListBox listbox)
        {
            this.listbox = listbox;
            setListBoxCallback = new SetListBoxCallback(SetListBox);
        }
        public void SetListBox(string str)
        {
            //比较调用 SetListBox 方法的线程和创建 listBox1 的线程是否同一个线程
            //如果不是，则 listBox1 的 InvokeRequired 为 true
            if (listbox.InvokeRequired)
            {
                //结果为 true，则通过代理执行 else 中的代码,并传入需要的参数
                listbox.Invoke(setListBoxCallback, str);
            }
            else
            {
                //结果为 false，直接执行
                listbox.Items.Add(str);
                listbox.SelectedIndex = listbox.Items.Count - 1;
                listbox.ClearSelected();
            }
        }
        public void SendToOne(User user, string str)
        {
```

```
                try
                {
                    user.sw.WriteLine(str);
                    user.sw.Flush();
                    SetListBox(string.Format("向{0}发送{1}", user.userName, str));
                }
                catch
                {
                    SetListBox(string.Format("向{0}发送信息失败", user.userName));
                }
            }
            public void SendToBoth(GameTable gameTable, string str)
            {
                for (int i = 0; i < 2; i++)
                {
                    if (gameTable.gamePlayer[i].someone == true)
                    {
                        SendToOne(gameTable.gamePlayer[i].user, str);
                    }
                }
            }
            public void SendToAll(System.Collections.Generic.List<User> userList, string str)
            {
                for (int i = 0; i < userList.Count; i++)
                {
                    SendToOne(userList[i], str);
                }
            }
        }
    }
```

(5) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 GameTable.cs，负责向某桌发送棋子以及判断胜负等。代码如下：

```
//------------------GameTable.cs-----------------//
using System;
using System.Timers;
using System.Windows.Forms;
using System.Net.Sockets;
using System.IO;
namespace GameServer
{
    class GameTable
    {
        private const int None = -1;          //无棋子
        private const int Black = 0;          //黑色棋子
        private const int White = 1;          //白色棋子
        public Player[] gamePlayer;
        private int[,] grid = new int[15, 15];          //15*15 的方格
        private System.Timers.Timer timer;          //用于定时产生棋子
        private int NextdotColor = 0;                //应该产生黑棋子还是白棋子
        private ListBox listbox;
        Random rnd = new Random();
```

```
Service service;
public GameTable(ListBox listbox)
{
    gamePlayer = new Player[2];
    gamePlayer[0] = new Player();
    gamePlayer[1] = new Player();
    timer = new System.Timers.Timer();
    timer.Elapsed += new ElapsedEventHandler(timer_Elapsed);
    timer.Enabled = false;
    this.listbox = listbox;
    service = new Service(listbox);
    ResetGrid();
}
public void ResetGrid()
{
    for (int i = 0; i <= grid.GetUpperBound(0); i++)
    {
        for (int j = 0; j <= grid.GetUpperBound(1); j++)
        {
            grid[i, j] = None;
        }
    }
    gamePlayer[0].grade = 0;
    gamePlayer[1].grade = 0;
}
public void StartTimer()
{
    timer.Start();
}
public void StopTimer()
{
    timer.Stop();
}
public void SetTimerLevel(int interval)
{
    timer.Interval = interval;
}
private void timer_Elapsed(object sender, EventArgs e)
{
    int x, y;
    //随机产生一个格内没有棋子的单元格位置
    do
    {
        x = rnd.Next(15);   //产生一个小于 15 的非负整数
        y = rnd.Next(15);
    } while (grid[x, y] != None);
    //放置棋子:x 坐标,y 坐标,颜色
    SetDot(x, y, NextdotColor);
    NextdotColor = (NextdotColor + 1) % 2;
}
private void SetDot(int i, int j, int dotColor)
```

```
{
    //向两个用户发送产生的棋子信息，并判断是否有相邻棋子
    //发送格式：SetDot,行,列,颜色
    grid[i, j] = dotColor;
    service.SendToBoth(this, string.Format("SetDot,{0},{1},{2}", i, j, dotColor));
    /*----------以下判断当前行是否有相邻点----------*/
    int k1, k2;     //k1:循环初值，k2:循环终值
    if (i == 0)
    {
        //如果是首行，只需要判断下边的点
        k1 = k2 = 1;
    }
    else if (i == grid.GetUpperBound(0))
    {
        //如果是最后一行，只需要判断上边的点
        k1 = k2 = grid.GetUpperBound(0) - 1;
    }
    else
    {
        //如果是中间的行，上下两边的点都要判断
        k1 = i - 1; k2 = i + 1;
    }
    for (int x = k1; x <= k2; x += 2)
    {
        if (grid[x, j] == dotColor)
        {
            ShowWin(dotColor);
        }
    }
    /*-------------以下判断当前列是否有相邻点------------------*/
    if (j == 0)
    {
        k1 = k2 = 1;
    }
    else if (j == grid.GetUpperBound(1))
    {
        k1 = k2 = grid.GetUpperBound(1) - 1;
    }
    else
    {
        k1 = j - 1; k2 = j + 1;
    }
    for (int y = k1; y <= k2; y += 2)
    {
        if (grid[i, y] == dotColor)
        {
            ShowWin(dotColor);
        }
    }
}
//出现相邻点的颜色为 dotColor
```

```
        private void ShowWin(int dotColor)
        {
            timer.Enabled = false;
            gamePlayer[0].started = false;
            gamePlayer[1].started = false;
            this.ResetGrid();
            //发送格式：Win,相邻点的颜色,黑方成绩,白方成绩
            service.SendToBoth(this, string.Format("Win,{0},{1},{2}",
                dotColor, gamePlayer[0].grade, gamePlayer[1].grade));
        }
        public void UnsetDot(int i, int j, int color)
        {
            //向两个用户发送消去棋子的信息
            //格式：UnsetDot,行,列,黑方成绩,白方成绩
            grid[i, j] = None;
            gamePlayer[color].grade++;
            string str = string.Format("UnsetDot,{0},{1},{2},{3}",
                i, j, gamePlayer[0].grade, gamePlayer[1].grade);
            service.SendToBoth(this, str);
        }
    }
}
```

(6) 切换到 FormServer 的代码编辑方式下，添加对应按钮的 Click 事件以及其他代码，源程序如下：

```
//-------------FormServer.cs-----------------//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;
namespace GameServer
{
    public partial class FormServer : Form
    {
        //游戏室允许进入的最多人数
        private int maxUsers;
        //连接的用户
        System.Collections.Generic.List<User> userList = new List<User>();
        //游戏室开出的桌数。
        private int maxTables;
        private GameTable[] gameTable;
        //使用的本机 IP 地址
        IPAddress localAddress;
        //监听端口
```

```
private int port = 51888;
private TcpListener myListener;
private Service service;
public FormServer()
{
    InitializeComponent();
    service = new Service(listBox1);
}
//加载窗体时触发的事件
private void FormServer_Load(object sender, EventArgs e)
{
    listBox1.HorizontalScrollbar = true;
    IPAddress[] addrIP = Dns.GetHostAddresses(Dns.GetHostName());
    localAddress = addrIP[0];
    buttonStop.Enabled = false;
}
//【开始服务】按钮的 Click 事件
private void buttonStart_Click(object sender, EventArgs e)
{
    if (int.TryParse(textBoxMaxTables.Text, out maxTables) == false
        || int.TryParse(textBoxMaxUsers.Text, out maxUsers) == false)
    {
        MessageBox.Show("请输入在规定范围内的正整数");
        return;
    }
    if (maxUsers < 1 || maxUsers > 300)
    {
        MessageBox.Show("允许进入的人数只能在 1-300 之间");
        return;
    }
    if (maxTables < 1 || maxTables > 100)
    {
        MessageBox.Show("允许的桌数只能在 1-100 之间");
        return;
    }
    textBoxMaxUsers.Enabled = false;
    textBoxMaxTables.Enabled = false;
    //初始化数组
    gameTable = new GameTable[maxTables];
    for (int i = 0; i < maxTables; i++)
    {
        gameTable[i] = new GameTable(listBox1);
    }
    myListener = new TcpListener(localAddress, port);
    myListener.Start();
    service.SetListBox(string.Format("开始在{0}:{1}监听客户连接", localAddress, port));
    //创建一个线程监听客户端连接请求
    ThreadStart ts = new ThreadStart(ListenClientConnect);
    Thread myThread = new Thread(ts);
    myThread.Start();
    buttonStart.Enabled = false;
```

```
        buttonStop.Enabled = true;
    }
    //【停止服务】按钮的 Click 事件
    private void buttonStop_Click(object sender, EventArgs e)
    {
        //停止向游戏桌发送棋子
        for (int i = 0; i < maxTables; i++)
        {
            gameTable[i].StopTimer();
        }
        service.SetListBox(string.Format("目前连接用户数：{0}", userList.Count));
        service.SetListBox("开始停止服务，并依次使用户退出!");
        for (int i = 0; i < userList.Count; i++)
        {
            //关闭后，客户端接收字符串为 null,
            //使接收该客户的线程 ReceiveData 接收的字符串也为 null,
            //从而达到结束线程的目的
            userList[i].client.Close();
        }
        //通过停止监听让 myListener.AcceptTcpClient()产生异常退出监听线程
        myListener.Stop();
        buttonStart.Enabled = true;
        buttonStop.Enabled = false;
        textBoxMaxUsers.Enabled = true;
        textBoxMaxTables.Enabled = true;
    }
    //接收客户端连接
    private void ListenClientConnect()
    {
        while (true)
        {
            TcpClient newClient = null;
            try
            {
                //等待用户进入
                newClient = myListener.AcceptTcpClient();
            }
            catch
            {
                //当单击“停止监听”或者退出此窗体时 AcceptTcpClient()会产生异常
                //因此可以利用此异常退出循环
                break;
            }
            //每接受一个客户端连接,就创建一个对应的线程循环接收该客户端发来的信息
            ParameterizedThreadStart pts = new ParameterizedThreadStart(ReceiveData);
            Thread threadReceive = new Thread(pts);
            User user = new User(newClient);
            threadReceive.Start(user);
            userList.Add(user);
            service.SetListBox(string.Format("{0}进入", newClient.Client.RemoteEndPoint));
            service.SetListBox(string.Format("当前连接用户数：{0}", userList.Count));
```

```
        }
    }
    //接收、处理客户端信息，每客户 1 个线程，参数用于区分是哪个客户
    private void ReceiveData(object obj)
    {
        User user = (User)obj;
        TcpClient client = user.client;
        //是否正常退出接收线程
        bool normalExit = false;
        //用于控制是否退出循环
        bool exitWhile = false;
        while (exitWhile == false)
        {
            string receiveString = null;
            try
            {
                receiveString = user.sr.ReadLine();
            }
            catch
            {
                //该客户底层套接字不存在时会出现异常
                service.SetListBox("接收数据失败");
            }
            //TcpClient 对象将套接字进行了封装，如果 TcpClient 对象关闭了，
            //但是底层套接字未关闭，并不产生异常，但是读取的结果为 null
            if (receiveString == null)
            {
                if (normalExit == false)
                {
                    //如果停止了监听，Connected 为 false
                    if (client.Connected == true)
                    {
                        service.SetListBox(string.Format(
                            "与{0}失去联系，已终止接收该用户信息",
                              client.Client.RemoteEndPoint));
                    }
                    //如果该用户正在游戏桌上，则退出游戏桌
                    RemoveClientfromPlayer(user);
                }
                //退出循环
                break;
            }
            service.SetListBox(string.Format("来自{0}：{1}", user.userName, receiveString));
            string[] splitString = receiveString.Split(',');
            int tableIndex = -1;       //桌号
            int side = -1;              //座位号
            int anotherSide = -1;     //对方座位号
            string sendString = "";
            switch (splitString[0])
            {
                case "Login":
```

```
            //格式：Login,昵称
            //该用户刚刚登录
            if (userList.Count > maxUsers)
            {
                sendString = "Sorry";
                service.SendToOne(user, sendString);
                service.SetListBox("人数已满，拒绝" +
                    splitString[1] + "进入游戏室");
                exitWhile = true;
            }
            else
            {
                //将用户昵称保存到用户列表中
                //由于是引用类型,因此直接给 user 赋值也就是给 userList 中对
                //应的 user 赋值
                //用户名中包含其 IP 和端口的目的是为了帮助理解，实际游戏
                //中一般不会显示 IP 的
                user.userName = string.Format("[{0}--{1}]", splitString[1],
                    client.Client.RemoteEndPoint);
                //允许该用户进入游戏室，即将各桌是否有人情况发送给该用户
                sendString = "Tables," + this.GetOnlineString();
                service.SendToOne(user, sendString);
            }
            break;
        case "Logout":
            //格式：Logout
            //用户退出游戏室
            service.SetListBox(string.Format("{0}退出游戏室", user.userName));
            normalExit = true;
            exitWhile = true;
            break;
        case "SitDown":
            //格式：SitDown,桌号,座位号
            //该用户坐到某座位上
            tableIndex = int.Parse(splitString[1]);
            side = int.Parse(splitString[2]);
            gameTable[tableIndex].gamePlayer[side].user = user;
            gameTable[tableIndex].gamePlayer[side].someone = true;
            service.SetListBox(string.Format(
                "{0}在第{1}桌第{2}座入座",
                user.userName, tableIndex + 1, side + 1));
            //得到对家座位号
            anotherSide = (side + 1) % 2;
            //判断对方是否有人
            if (gameTable[tableIndex].gamePlayer[anotherSide].someone == true)
            {
                //先告诉该用户对家已经入座
                //发送格式：SitDown,座位号,用户名
                sendString = string.Format("SitDown,{0},{1}", anotherSide,
                   gameTable[tableIndex].gamePlayer[anotherSide].user.userName);
                service.SendToOne(user, sendString);
```

```
            }
            //同时告诉两个用户该用户入座(也可能对方无人)
            //发送格式：SitDown,座位号,用户名
            sendString = string.Format("SitDown,{0},{1}", side, user.userName);
            service.SendToBoth(gameTable[tableIndex], sendString);
            //重新将游戏室各桌情况发送给所有用户
            service.SendToAll(userList, "Tables," + this.GetOnlineString());
            break;
        case "GetUp":
            //格式：GetUp,桌号,座位号
            //用户离开座位,回到游戏室
            tableIndex = int.Parse(splitString[1]);
            side = int.Parse(splitString[2]);
            service.SetListBox(
                  string.Format("{0}离座,返回游戏室", user.userName));
            gameTable[tableIndex].StopTimer();
            //将离座信息同时发送给两个用户，以便客户端作离座处理
            //发送格式：GetUp,座位号,用户名
            service.SendToBoth(gameTable[tableIndex],
                string.Format("GetUp,{0},{1}", side, user.userName));
            //服务器进行离座处理
            gameTable[tableIndex].gamePlayer[side].someone = false;
            gameTable[tableIndex].gamePlayer[side].started = false;
            gameTable[tableIndex].gamePlayer[side].grade = 0;
            anotherSide = (side + 1) % 2;
            if (gameTable[tableIndex].gamePlayer[anotherSide].someone == true)
            {
                gameTable[tableIndex].gamePlayer[anotherSide].started = false;
                gameTable[tableIndex].gamePlayer[anotherSide].grade = 0;
            }
            //重新将游戏室各桌情况发送给所有用户
            service.SendToAll(userList, "Tables," + this.GetOnlineString());
            break;
        case "Level":
            //格式：Time,桌号,难度级别
            //设置难度级别
            tableIndex = int.Parse(splitString[1]);
            gameTable[tableIndex].SetTimerLevel((6 - int.Parse(splitString[2])) * 100);
            service.SendToBoth(gameTable[tableIndex], receiveString);
            break;
        case "Talk":
            //格式：Talk,桌号,对话内容
            tableIndex = int.Parse(splitString[1]);
            //由于说话内容可能包含逗号，所以需要特殊处理
            sendString = string.Format("Talk,{0},{1}", user.userName,
                receiveString.Substring(splitString[0].Length +
                splitString[1].Length));
            service.SendToBoth(gameTable[tableIndex], sendString);
            break;
        case "Start":
            //格式：Start,桌号,座位号
```

```
                    //该用户单击了开始按钮
                    tableIndex = int.Parse(splitString[1]);
                    side = int.Parse(splitString[2]);
                    gameTable[tableIndex].gamePlayer[side].started = true;
                    if (side == 0)
                    {
                        anotherSide = 1;
                        sendString = "Message,黑方已开始。";
                    }
                    else
                    {
                        anotherSide = 0;
                        sendString = "Message,白方已开始。";
                    }
                    service.SendToBoth(gameTable[tableIndex], sendString);
                    if (gameTable[tableIndex].gamePlayer[anotherSide].started == true)
                    {
                        gameTable[tableIndex].ResetGrid();
                        gameTable[tableIndex].StartTimer();
                    }
                    break;
                case "UnsetDot":
                    //格式：UnsetDot,桌号,座位号,行,列,颜色
                    //消去客户单击的棋子
                    tableIndex = int.Parse(splitString[1]);
                    side = int.Parse(splitString[2]);
                    int xi = int.Parse(splitString[3]);
                    int xj = int.Parse(splitString[4]);
                    int color = int.Parse(splitString[5]);
                    gameTable[tableIndex].UnsetDot(xi, xj, color);
                    break;
                default:
                    service.SendToAll(userList, "什么意思啊：" + receiveString);
                    break;
            }
        }
        userList.Remove(user);
        client.Close();
        service.SetListBox(string.Format("有一个退出，剩余连接用户数：{0}", userList.Count));
    }
    // 循环检测该用户是否坐到某游戏桌上,如果是,将其从游戏桌上移除，并终止该桌游戏
    private void RemoveClientfromPlayer(User user)
    {
        for (int i = 0; i < gameTable.Length; i++)
        {
            for (int j = 0; j < 2; j++)
            {
                if (gameTable[i].gamePlayer[j].user != null)
                {
                    //判断是否同一个对象
                    if (gameTable[i].gamePlayer[j].user == user)
```

```
                    {
                        StopPlayer(i, j);
                        return;
                    }
                }
            }
        }
    }
    //停止第 i 桌游戏
    private void StopPlayer(int i, int j)
    {
        gameTable[i].StopTimer();
        gameTable[i].gamePlayer[j].someone = false;
        gameTable[i].gamePlayer[j].started = false;
        gameTable[i].gamePlayer[j].grade = 0;
        int otherSide = (j + 1) % 2;
        if (gameTable[i].gamePlayer[otherSide].someone == true)
        {
            gameTable[i].gamePlayer[otherSide].started = false;
            gameTable[i].gamePlayer[otherSide].grade = 0;
            if (gameTable[i].gamePlayer[otherSide].user.client.Connected == true)
            {
                //发送格式：Lost,座位号,用户名
                service.SendToOne(gameTable[i].gamePlayer[otherSide].user,
                    string.Format("Lost,{0},{1}",
                     j, gameTable[i].gamePlayer[j].user.userName));
            }
        }
    }
    //获取每桌是否有人的字符串，每座用一位表示，0 表示无人，1 表示有人
    private string GetOnlineString()
    {
        string str = "";
        for (int i = 0; i < gameTable.Length; i++)
        {
            for (int j = 0; j < 2; j++)
            {
                str += gameTable[i].gamePlayer[j].someone == true ? "1" : "0";
            }
        }
        return str;
    }
    //关闭窗体前触发的事件
    private void FormDDServer_FormClosing(object sender, FormClosingEventArgs e)
    {
        //未单击开始服务就直接退出时，myListener 为 null
        if (myListener != null)
        {
            buttonStop_Click(null, null);
        }
    }
```

```
        }
    }
```

(7) 按<F5>键编译并运行，保证无编译错误，然后退出。

2.2.2 客户端编程

对客户端来说，由于只需要与服务器打交道，因此相对来说比较简单。客户端与服务器连接成功后，也需要创建一个接收线程，用于接收服务器发送的信息。

在接收线程中，客户端收到服务器发送的字符串信息后，与服务器接收的字符串相似，客户端也需要解析字符串的含义，并决定需要的操作。接收的字符串同样分为命令部分和参数部分，命令部分和参数部分之间以及参数部分的参数之间均以逗号分隔。具体规定为：

- Sorry：服务器游戏室人员已满。客户端接收到此命令后，由于无法进入游戏室，继续运行程序已经没有意义，因此直接结束接收线程，并退出整个客户端程序。
- Tables：各小房间的游戏桌情况。客户端接收到此信息后，需要计算对应的桌数，并以 CheckBox 的形式显示出各桌是否有人的情况，供玩家选择座位。
- SitDown：玩家坐到某个小房间的游戏桌上。客户端接收到此信息后，需要判断是自己还是对家，并在窗体中显示相应信息。
- GetUp：玩家离开小房间的游戏桌，回到游戏室。客户端收到此信息后，需要判断是对家离座了还是自己离座了，如果对家离座，显示一个对话框；如果是自己离座，除了设置对应的标志外，不做其他任何处理。
- Lost：对家与服务器失去联系。由于游戏无法继续进行，直接从游戏桌返回到游戏室即可。
- StopService：服务器停止服务。由于服务器已经退出接收线程，继续运行客户端程序已经没有意义，所以此时不论是否正在进行游戏，都要立即结束。
- Talk：谈话内容。由于谈话中可能包括逗号，因此需要单独处理。
- Message：服务器向游戏桌玩家发送的信息，比如入座信息等。
- Level：难度级别。客户端收到此信息后，需要显示出相应的级别。
- SetDot：在棋盘上自动产生棋子的信息，信息中包含了棋子位置以及颜色。客户端收到此信息后，需要在棋盘相应位置将对应颜色的棋子显示出来。
- UnsetDot：消去棋子的信息。客户端收到此信息后，需要将棋盘上对应位置的棋子去掉，并更新对应的成绩。
- Win：有一家已经出现相邻棋子，未出现相邻棋子的为胜方。

除了负责接收的线程外，客户端还需要根据服务器发送的命令，及时更新客户端程序的运行界面。具体设计步骤为：

(1) 选择一种作图软件，设计如图 2-2 所示的棋子和棋盘，分别保存为 black.gif、white.gif 和 grid.gif。

注意，由于计算棋盘中棋子的位置是以像素为单位的，因此要求棋盘中每条线之间的距离相等，程序要求棋盘中纵横方格线之间的距离均为 20 像素，其中 15×15 方格的左方边线与边的距离以及上方边线与边的距离也必须是 20 像素，否则会引起判断失误。

(2) 创建一个名为 GameClient 的 Windows 应用程序项目，将 Form1.cs 换名为 FormRoom.cs，设计界面如图 2-3 所示。其中昵称右边的文本框【Name】属性为 textBoxName，

登录按钮的下面是一个 Panel 控件对象。

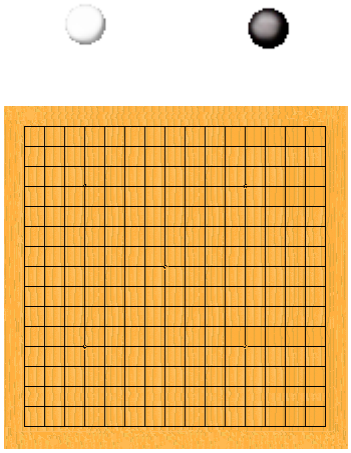


图2-2　棋子与棋盘

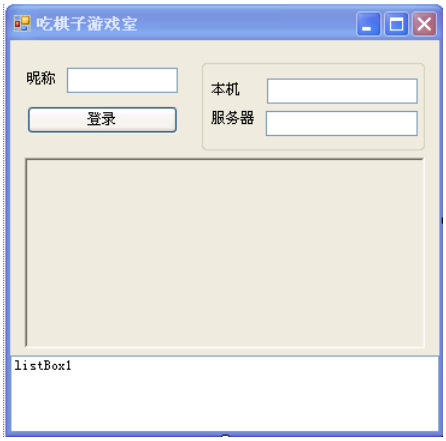


图2-3　FormRoom设计界面

(3) 在解决方案资源管理器中，鼠标右键单击项目名，选择【添加】→【现有项】，然后将 black.gif 和 white.gif 添加到项目中，并设置其【复制到输出目录】属性为“始终复制”，用于在棋盘上显示棋子。

(4) 在解决方案资源管理器中，鼠标右键单击项目名，选择【添加】→【类】，添加一个类文件 DotColor.cs，用于提供棋子的颜色。代码如下：

```
//----------DotColor.cs---------------//
namespace GameClient
{
    class DotColor
    {
        public const int None = -1;     //背景色
        public const int Black = 0;     //黑色
        public const int White = 1;     //白色
    }
}
```

注意，由于显示棋子以及判断棋盘上是否有棋子均使用了这个类中定义的常量，所以这些常量的值不是随便定义的，如果把常量的值变为其他数字，就会引起判断失误。

(5) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 Service.cs，用于提供公用的方法。代码如下：

```
//------Service.cs----------//
using System;
using System.Windows.Forms;
using System.IO;
namespace GameClient
{
    class Service
    {
        ListBox listbox;
        StreamWriter sw;
```

```
        public Service(ListBox listbox, StreamWriter sw)
        {
            this.listbox = listbox;
            this.sw = sw;
        }
        public void SendToServer(string str)
        {
            try
            {
                sw.WriteLine(str);
                sw.Flush();
            }
            catch
            {
                SetListBox("发送数据失败");
            }
        }
        delegate void ListBoxCallback(string str);
        public void SetListBox(string str)
        {
            if (listbox.InvokeRequired == true)
            {
                ListBoxCallback d = new ListBoxCallback(SetListBox);
                listbox.Invoke(d, str);
            }
            else
            {
                listbox.Items.Add(str);
                listbox.SelectedIndex = listbox.Items.Count - 1;
                listbox.ClearSelected();
            }
        }
    }
}
```

这个类中提供了两个方法，一个用于向服务器发送数据，另一个用于在对应的界面上显示相关信息。

(6) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【Windows 窗体】，添加一个 Windows 窗体文件 FormPlaying.cs，设计界面如图 2-4 所示。

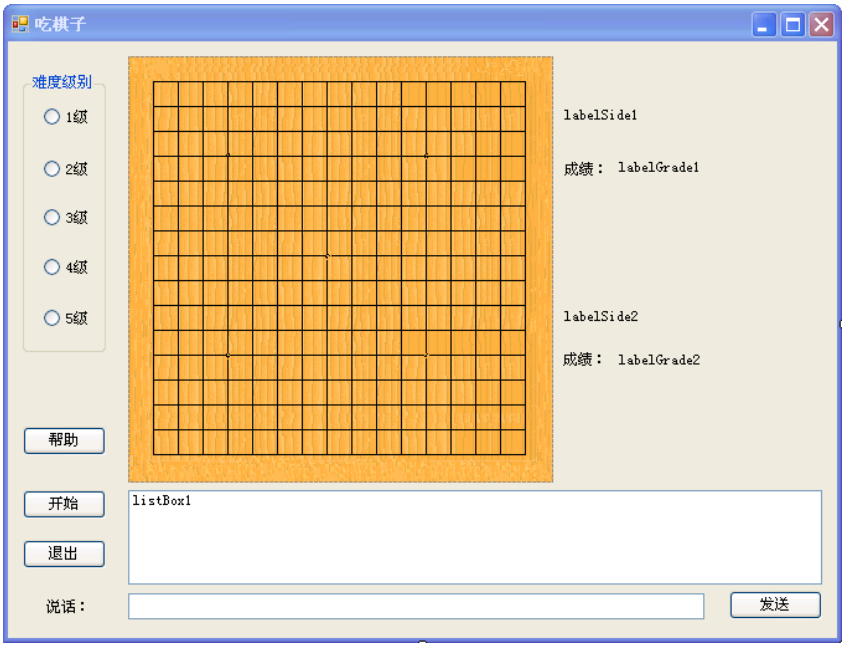


图2-4　客户端FormPlaying.cs的设计界面

图中显示的棋盘使用的是 PictureBox 控件对象，该对象的具体属性为：

【Name】属性：pictureBox1。

【Image】属性：从步骤(1)设计的棋盘 grid.gif 中得到的图像。

【SizeMode】属性：AutoSize。

这是玩家坐到游戏桌座位后显示的界面。界面使用 listBox1 显示双方的状态、对话内容以及来自服务器的数据，目的是为了便于理解收发的过程。实际应用中，一般不会显示收发的数据，而只是显示必要的信息。

(7) 在 FormPlaying.cs 的代码编辑器中添加对应的代码和事件，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;
namespace GameClient
{
    public partial class FormPlaying : Form
    {
        private int tableIndex;
        private int side;
        private int[,] grid = new int[15, 15]; //保存颜色，用于消点时进行判断
```

```
private Bitmap blackBitmap;
private Bitmap whiteBitmap;
//是否从服务器接收的命令
private bool isReceiveCommand = false;
private Service service;
delegate void LabelDelegate(Label label, string str);
delegate void ButtonDelegate(Button button, bool flag);
delegate void RadioButtonDelegate(RadioButton radioButton, bool flag);
delegate void SetDotDelegate(int i, int j, int dotColor);
LabelDelegate labelDelegate;
ButtonDelegate buttonDelegate;
RadioButtonDelegate radioButtonDelegate;
public FormPlaying(int TableIndex, int Side, StreamWriter sw)
{
    InitializeComponent();
    this.tableIndex = TableIndex;
    this.side = Side;
    labelDelegate = new LabelDelegate(SetLabel);
    buttonDelegate = new ButtonDelegate(SetButton);
    radioButtonDelegate = new RadioButtonDelegate(SetRadioButton);
    blackBitmap = new Bitmap("black.gif");
    whiteBitmap = new Bitmap("white.gif");
    service = new Service(listBox1, sw);
}
private void FormPlaying_Load(object sender, EventArgs e)
{
    radioButton3.Checked = true;
    for (int i = 0; i <= grid.GetUpperBound(0); i++)
    {
        for (int j = 0; j <= grid.GetUpperBound(1); j++)
        {
            grid[i, j] = DotColor.None;
        }
    }
    labelSide0.Text = "";
    labelSide1.Text = "";
    labelGrade0.Text = "";
    labelGrade1.Text = "";
}
public void SetLabel(Label label, string str)
{
    if (label.InvokeRequired == true)
    {
        this.Invoke(labelDelegate, label, str);
    }
    else
    {
        label.Text = str;
    }
}
private void SetButton(Button button, bool flag)
```

```
{
    if (button.InvokeRequired == true)
    {
        this.Invoke(buttonDelegate, button, flag);
    }
    else
    {
        button.Enabled = flag;
    }
}
private void SetRadioButton(RadioButton radioButton, bool flag)
{
    if (radioButton.InvokeRequired == true)
    {
        this.Invoke(radioButtonDelegate, radioButton, flag);
    }
    else
    {
        radioButton.Checked = flag;
    }
}
public void SetDot(int i, int j, int dotColor)
{
    service.SetListBox(string.Format("{0},{1},{2}", i, j, dotColor));
    grid[i, j] = dotColor;
    pictureBox1.Invalidate();
}
public void Restart(string str)
{
    MessageBox.Show(str, "", MessageBoxButtons.OK, MessageBoxIcon.Warning);
    ResetGrid();
    SetButton(buttonStart, true);
}
private void ResetGrid()
{
    SetLabel(labelGrade0, "");
    SetLabel(labelGrade1, "");
    for (int i = 0; i <= grid.GetUpperBound(0); i++)
    {
        for (int j = 0; j <= grid.GetUpperBound(1); j++)
        {
            grid[i, j] = DotColor.None;
        }
    }
    pictureBox1.Invalidate();
}
public void UnsetDot(int x, int y)
{
    grid[x / 20 - 1, y / 20 - 1] = DotColor.None;
    pictureBox1.Invalidate();
}
```

```
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    for (int i = 0; i <= grid.GetUpperBound(0); i++)
    {
        for (int j = 0; j <= grid.GetUpperBound(1); j++)
        {
            if (grid[i, j] != DotColor.None)
            {
                if (grid[i, j] == DotColor.Black)
                {
                    g.DrawImage(blackBitmap, (i + 1) * 20, (j + 1) * 20);
                }
                else
                {
                    g.DrawImage(whiteBitmap, (i + 1) * 20, (j + 1) * 20);
                }
            }
        }
    }
}
public void SetLevel(string ss)
{
    isReceiveCommand = true;
    switch (ss)
    {
        case "1":
            SetRadioButton(radioButton1, true);
            break;
        case "2":
            SetRadioButton(radioButton2, true);
            break;
        case "3":
            SetRadioButton(radioButton3, true);
            break;
        case "4":
            SetRadioButton(radioButton4, true);
            break;
        case "5":
            SetRadioButton(radioButton5, true);
            break;
    }
    isReceiveCommand = false;
}
private void radioButton_CheckedChanged(object sender, EventArgs e)
{
    //isReceiveCommand 为 true 表明是接收服务器设置的难度级别触发的此事件
    //就不需要再向服务器发送
    if (isReceiveCommand == false)
    {
        RadioButton radiobutton = (RadioButton)sender;
```

```
            if (radiobutton.Checked == true)
            {
                //设置难度级别
                //格式：Time,桌号,难度级别
                service.SendToServer(string.Format("Level,{0},{1}",
                    tableIndex, radiobutton.Name[radiobutton.Name.Length - 1]));
            }
        }
    }
    private void buttonSend_Click(object sender, EventArgs e)
    {
        //字符串格式：Talk,桌号,对话内容
        service.SendToServer(string.Format("Talk,{0},{1}", tableIndex, textBox1.Text));
    }
    //对话内容改变时触发的事件
    private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {
        if (e.KeyChar == (char)Keys.Enter)
        {
            //字符串格式：Talk,桌号,对话内容
            service.SendToServer(string.Format("Talk,{0},{1}", tableIndex, textBox1.Text));
        }
    }
    private void buttonHelp_Click(object sender, EventArgs e)
    {
        string str =
            "\n 本游戏每两人为一组。游戏玩法：用鼠标点击你所在方颜色的点，每\n\n" +
            "消失一个，得一分。当任何一方在同行或同列出现两个相邻的点时，游\n\n" +
            "戏就结束了，此时得分多的为胜方。注意：点击对方的点无效。\n";
        MessageBox.Show(str, "帮助信息");
    }
    private void buttonStart_Click(object sender, EventArgs e)
    {
        service.SendToServer(string.Format("Start,{0},{1}", tableIndex, side));
        this.buttonStart.Enabled = false;
    }
    private void buttonExit_Click(object sender, EventArgs e)
    {
        this.Close();
    }
    //关闭窗体前触发的事件
    private void FormPlaying_FormClosing(object sender, FormClosingEventArgs e)
    {
        //格式：GetUp,桌号,座位号
        service.SendToServer(string.Format("GetUp,{0},{1}", tableIndex, side));
    }
    //FormRoom 中的线程调用此方法关闭此窗体
    public void StopFormPlaying()
    {
        Application.Exit();
    }
```

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
    int x = e.X / 20;
    int y = e.Y / 20;
    if (!(x < 1 || x > 15 || y < 1 || y > 15))
    {
        if (grid[x - 1, y - 1] != DotColor.None)
        {
            int color = grid[x - 1, y - 1];
            //发送格式：UnsetDot,桌号,座位号,行,列,颜色
            service.SendToServer(string.Format(
                "UnsetDot,{0},{1},{2},{3},{4}", tableIndex, side, x - 1, y - 1, color));
        }
    }
}
//参数格式：座位号，labelSide 显示的信息，listbox 显示的信息
public void SetTableSideText(string sideString, string labelSideString, string listBoxString)
{
    string s = "白方";
    if (sideString == "0")
    {
        s = "黑方：";
    }
    //判断自己是黑方还是白方
    if (sideString == side.ToString())
    {
        SetLabel(labelSide1, s + labelSideString);
    }
    else
    {
        SetLabel(labelSide0, s + labelSideString);
    }
    service.SetListBox(listBoxString);
}
//参数格式：grade0 为黑方成绩，grade1 为白方成绩
public void SetGradeText(string str0, string str1)
{
    if (side == DotColor.Black)
    {
        SetLabel(labelGrade1, str0);
        SetLabel(labelGrade0, str1);
    }
    else
    {
        SetLabel(labelGrade0, str0);
        SetLabel(labelGrade1, str1);
    }
}
public void ShowTalk(string talkMan, string str)
{
    service.SetListBox(string.Format("{0}说：{1}", talkMan, str));
```

```
        }
        public void ShowMessage(string str)
        {
            service.SetListBox(str);
        }
    }
}
```

(8) 在 FormRoom.cs 的代码编辑器中添加对应的代码和事件，源程序如下：

```
//---------FormRoom.cs----------//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading;
namespace GameClient
{
    public partial class FormRoom : Form
    {
        private int maxPlayingTables;
        private CheckBox[,] checkBoxGameTables;
        private TcpClient client = null;
        private StreamWriter sw;
        private StreamReader sr;
        private Service service;
        private FormPlaying formPlaying;
        //是否正常退出接收线程
        private bool normalExit = false;
        //是否从服务器接收的命令
        private bool isReceiveCommand = false;
        //所坐的游戏桌座位号，-1 表示未入座
        private int side = -1;
        public FormRoom()
        {
            InitializeComponent();
        }
        private void FormRoom_Load(object sender, EventArgs e)
        {
            maxPlayingTables = 0;
            textBoxName.MaxLength = 6;
            textBoxLocal.ReadOnly = true;
            textBoxServer.ReadOnly = true;
            groupBox1.Visible = false;
        }
        //【登录】按钮的 Click 事件
```

```
private void buttonConnect_Click(object sender, EventArgs e)
{
    if (textBoxName.Text.Trim().Length == 0)
    {
        MessageBox.Show("请输入昵称。", "",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        return;
    }
    else if (textBoxName.Text.IndexOf(',') != -1)
    {
        MessageBox.Show("昵称中不能包含逗号", "",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        return;
    }
    try
    {
        //实际使用时要将 Dns.GetHostName()改为服务器名
        client = new TcpClient(Dns.GetHostName(), 51888);
    }
    catch
    {
        MessageBox.Show("与服务器连接失败", "",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        return;
    }
    groupBox1.Visible = true;
    textBoxLocal.Text = client.Client.LocalEndPoint.ToString();
    textBoxServer.Text = client.Client.RemoteEndPoint.ToString();
    buttonConnect.Enabled = false;
    //获取网络流
    NetworkStream netStream = client.GetStream();
    sr = new StreamReader(netStream, System.Text.Encoding.UTF8);
    sw = new StreamWriter(netStream, System.Text.Encoding.UTF8);
    service = new Service(listBox1, sw);
    //登录服务器，获取服务器各桌信息
    //格式：Login,昵称
    service.SendToServer("Login," + textBoxName.Text.Trim());
    Thread threadReceive = new Thread(new ThreadStart(ReceiveData));
    threadReceive.Start();
}
private void ReceiveData()
{
    bool exitWhile = false;
    while (exitWhile == false)
    {
        string receiveString = null;
        try
        {
            receiveString = sr.ReadLine();
        }
        catch
```

```
{
    service.SetListBox("接收数据失败");
}
if (receiveString == null)
{
    if (normalExit == false)
    {
        MessageBox.Show("与服务器失去联系，游戏无法继续！");
    }
    if (side != -1)
    {
        ExitFormPlaying();
    }
    side = -1;
    normalExit = true;
    //结束线程
    break;
}
service.SetListBox("收到：" + receiveString);
string[] splitString = receiveString.Split(',');
switch (splitString[0])
{
    case "Sorry":
        MessageBox.Show("连接成功，但游戏室人数已满，无法进入。");
        exitWhile = true;
        break;
    case "Tables":
        //字符串格式：Tables,各桌是否有人的字符串
        //其中每位表示一个座位，1 表示有人，0 表示无人
        string s = splitString[1];
        //如果 maxPlayingTables 为 0，说明尚未创建 checkBoxGameTables
        if (maxPlayingTables == 0)
        {
            //计算所开桌数
            maxPlayingTables = s.Length / 2;
            checkBoxGameTables = new CheckBox[maxPlayingTables, 2];
            isReceiveCommand = true;
            //将 CheckBox 对象添加到数组中，以便管理
            for (int i = 0; i < maxPlayingTables; i++)
            {
                AddCheckBoxToPanel(s, i);
            }
            isReceiveCommand = false;
        }
        else
        {
            isReceiveCommand = true;
            for (int i = 0; i < maxPlayingTables; i++)
            {
                for (int j = 0; j < 2; j++)
                {
```

```
                    if (s[2 * i + j] == '0')
                    {
                        UpdateCheckBox(checkBoxGameTables[i, j], false);
                    }
                    else
                    {
                        UpdateCheckBox(checkBoxGameTables[i, j], true);
                    }
                }
            }
            isReceiveCommand = false;
        }
        break;
    case "SitDown":
        //格式：SitDown,座位号,用户名
        formPlaying.SetTableSideText(splitString[1], splitString[2],
                string.Format("{0}进入", splitString[2]));
        break;
    case "GetUp":
        //格式：GetUp,座位号,用户名
        //自己或者对方离座
        if (side == int.Parse(splitString[1]))
        {
            //自己离座
            side = -1;
        }
        else
        {
            //对方离座
            formPlaying.SetTableSideText(splitString[1], "",
                string.Format("{0}退出", splitString[2]));
            formPlaying.Restart("敌人逃跑了，我方胜利了");
        }
        break;
    case "Lost":
        //格式：Lost,座位号,用户名
        //对家与服务器失去联系
        formPlaying.SetTableSideText(splitString[1], "",
            string.Format("[{0}]与服务器失去联系", splitString[2]));
        formPlaying.Restart("对家与服务器失去联系，游戏无法继续");
        break;
    case "Talk":
        //格式：Talk,说话者,对话内容
        if (formPlaying != null)
        {
            //由于说话内容可能包含逗号，所以需要特殊处理
            formPlaying.ShowTalk(splitString[1],
                receiveString.Substring(splitString[0].Length +
                splitString[1].Length + splitString[2].Length + 3));
        }
        break;
```

```
                case "Message":
                    //格式：Message,内容
                    //服务器自动发送的一般信息（比如进入游戏桌入座等）
                    formPlaying.ShowMessage(splitString[1]);
                    break;
                case "Level":
                    //设置难度级别
                    //格式：Time,桌号,难度级别
                    formPlaying.SetLevel(splitString[2]);
                    break;
                case "SetDot":
                    //产生的棋子位置信息
                    //格式：Setdot,行,列,颜色
                    formPlaying.SetDot(
                        int.Parse(splitString[1]),
                        int.Parse(splitString[2]),
                        int.Parse(splitString[3]));
                    break;
                case "UnsetDot":
                    //消去棋子的信息
                    //格式：UnsetDot,行,列,黑方成绩,白方成绩
                    int x = 20 * (int.Parse(splitString[1]) + 1);
                    int y = 20 * (int.Parse(splitString[2]) + 1);
                    formPlaying.UnsetDot(x, y);
                    formPlaying.SetGradeText(splitString[3], splitString[4]);
                    break;
                case "Win":
                    //格式：Win,相邻棋子的颜色,黑方成绩,白方成绩
                    string winner = "";
                    if (int.Parse(splitString[1]) == DotColor.Black)
                    {
                        winner = "黑方出现相邻点，白方胜利！";
                    }
                    else
                    {
                        winner = "白方出现相邻点，黑方胜利！";
                    }
                    formPlaying.ShowMessage(winner);
                    formPlaying.Restart(winner);
                    break;
            }
        }
        //接收线程结束后，游戏继续进行已经没有意义，所以直接退出程序
        Application.Exit();
    }
    delegate void ExitFormPlayingDelegate();
    private void ExitFormPlaying()
    {
        if (formPlaying.InvokeRequired == true)
        {
            ExitFormPlayingDelegate d = new ExitFormPlayingDelegate(ExitFormPlaying);
```

```
            this.Invoke(d);
        }
        else
        {
            formPlaying.Close();
        }
    }
    delegate void PanelCallback(string s, int i);
    private void AddCheckBoxToPanel(string s, int i)
    {
        if (panel1.InvokeRequired == true)
        {
            PanelCallback d = new PanelCallback(AddCheckBoxToPanel);
            this.Invoke(d, s, i);
        }
        else
        {
            Label label = new Label();
            label.Location = new Point(10, 15 + i * 30);
            label.Text = string.Format("第{0}桌：", i + 1);
            label.Width = 70;
            this.panel1.Controls.Add(label);
            CreateCheckBox(i, 0, s, "黑方");
            CreateCheckBox(i, 1, s, "白方");
        }
    }
    delegate void CheckBoxDelegate(CheckBox checkbox, bool isChecked);
    private void UpdateCheckBox(CheckBox checkbox, bool isChecked)
    {
        if (checkbox.InvokeRequired == true)
        {
            CheckBoxDelegate d = new CheckBoxDelegate(UpdateCheckBox);
            this.Invoke(d, checkbox, isChecked);
        }
        else
        {
            if (side == -1)
            {
                checkbox.Enabled = !isChecked;
            }
            else
            {
                //已经坐到某游戏桌上，不允许再选其他桌
                checkbox.Enabled = false;
            }
            //注意改变 Checked 属性会触发 checked_Changed 事件
            checkbox.Checked = isChecked;
        }
    }
    private void CreateCheckBox(int i, int j, string s, string text)
    {
```

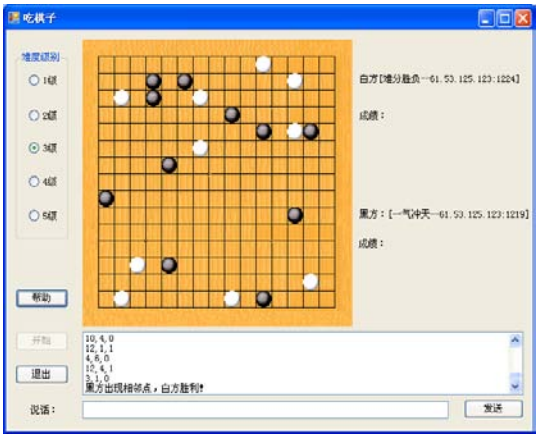
```
        int x = j == 0 ? 100 : 200;
        checkBoxGameTables[i, j] = new CheckBox();
        checkBoxGameTables[i, j].Name = string.Format("check{0:0000}{1:0000}", i, j);
        checkBoxGameTables[i, j].Width = 60;
        checkBoxGameTables[i, j].Location = new Point(x, 10 + i * 30);
        checkBoxGameTables[i, j].Text = text;
        checkBoxGameTables[i, j].TextAlign = ContentAlignment.MiddleLeft;
        if (s[2 * i + j] == '1')
        {
            //1 表示有人
            checkBoxGameTables[i, j].Enabled = false;
            checkBoxGameTables[i, j].Checked = true;
        }
        else
        {
            //0 表示无人
            checkBoxGameTables[i, j].Enabled = true;
            checkBoxGameTables[i, j].Checked = false;
        }
        this.panel1.Controls.Add(checkBoxGameTables[i, j]);
        checkBoxGameTables[i, j].CheckedChanged +=
            new EventHandler(checkBox_CheckedChanged);
    }
    //每个 CheckBox 的 Checked 属性发生变化都会触发此事件
    private void checkBox_CheckedChanged(object sender, EventArgs e)
    {
        //是否为服务器更新各桌
        if (isReceiveCommand == true)
        {
            return;
        }
        CheckBox checkbox = (CheckBox)sender;
        //Checked 为 true 表示玩家坐到第 i 桌第 j 位上
        if (checkbox.Checked == true)
        {
            int i = int.Parse(checkbox.Name.Substring(5, 4));
            int j = int.Parse(checkbox.Name.Substring(9, 4));
            side = j;
            //字符串格式：SitDown,昵称,桌号,座位号
            //只有坐下后，服务器才保存该玩家的昵称
            service.SendToServer(string.Format("SitDown,{0},{1}", i, j));
            formPlaying = new FormPlaying(i, j, sw);
            formPlaying.Show();
        }
    }
    private void FormRoom_FormClosing(object sender, FormClosingEventArgs e)
    {
        //未与服务器连接前 client 为 null
        if (client != null)
        {
            //不允许玩家从游戏桌直接退出整个程序
```

```
                    //只允许从游戏桌返回游戏室，再从游戏室退出
                    if (side != -1)
                    {
                        MessageBox.Show("请先从游戏桌站起，返回游戏室，然后再退出");
                        //不让退出
                        e.Cancel = true;
                    }
                    else
                    {
                        //服务器停止服务时,normalExit 为 true，其他情况为 false
                        if (normalExit == false)
                        {
                            normalExit = true;
                            //通知服务器，用户从游戏室退出
                            service.SendToServer("Logout");
                        }
                        //通过关闭 TcpClient 对象，使服务器接收字符串为 null
                        //服务器结束接收线程后,本程序的接收线程接收字符串也为 null
                        //从而实现退出程序功能
                        client.Close();             //关闭 client 对象
                    }
                }
            }
        }
    }
```

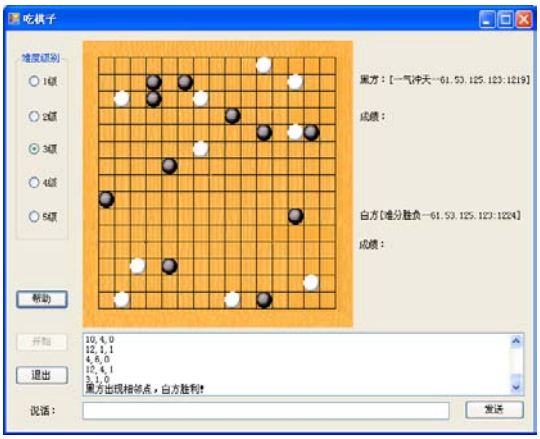
(9) 分别运行服务器端程序和客户端程序，部分运行界面效果如图 2-5、2-6 所示。

(a) 服务器端界面示例　　(b) 客户端游戏室界面示例

图2-5 服务器和客户端的部分运行界面效果



(c)　黑方玩家的界面示例



(d)　白方玩家的界面示例

图2-6　玩家坐到座位上后双方对弈的界面效果

2.3　异步 TCP 应用编程

利用 TcpListener 和 TcpClient 类在同步方式下接收、发送数据以及监听客户端连接时，在操作没有完成之前一直处于阻塞状态，这对于接收、发送数据量不大的情况或者操作用时较短的情况下是比较方便的。但是，对于执行完成时间可能较长的任务，如传送大文件等，使用同步操作可能就不太合适了，这种情况下，最好的办法是使用异步操作。

所谓异步操作方式，就是我们希望让某个工作开始以后，能在这个工作尚未完成的时候继续处理其他工作。就像我们（主线程）安排 A（子线程 A）负责处理客人来访时办理一系列登记手续。在同步工作方式下，如果没有人来访，A 就只能一直在某个房间等待，而不能同时做其他工作，显然这种方式不利于并行处理。我们希望的是，没有人来访时，A 不一定一直在这个房间等待，也可以到别处继续做其他事，而把这个工作交给总控室人员完成，这里的总控室就是 Windows 操作系统本身。总控室如何及时通知 A 呢？可以让 A 先告诉总控室一个手机号 F（callback 需要的方法名 F），以便有人来访时总控室可以立即电话通知 A（callback）。这样一来，一旦有客人来访，总控室人员（委托）就会立即给 A 打电话（通过委托自动运行方法 F），A 接到通知后，再处理客人来访时需要的登记手续（在方法 F 中完成需要的工作）。

异步操作的最大优点是可以在一个操作没有完成之前同时进行其他的操作。.NET 框架提供了一种称为 AsyncCallback（异步回调）的委托，该委托允许启动异步的功能，并在条件具备时调用提供的回调方法（是一种在操作或活动完成时由委托自动调用的方法），然后在这个方法中完成并结束未完成的工作。

使用异步 TCP 应用编程时，除了套接字有对应的异步操作方式外，TcpListener 和 TcpClient 类也提供了异步操作的方法。

异步操作方式下，每个 Begin 方法都有一个匹配的 End 方法。在程序中利用 Begin 方法开始执行异步操作，然后由委托在条件具备时调用 End 方法完成并结束异步操作。

表 2-3 列出了 TcpListener、TcpClient 以及套接字提供的部分异步操作方法。

表2-3　TcpListener和TcpClient及Socket提供的部分异步操作方法

类	提供的方法	说明
TcpListener	BeginAcceptTcpClient	开始一个异步操作接受一个传入的连接尝试
	EndAcceptTcpClient	异步接受传入的连接尝试，并创建新的TcpClient处理远程主机通信
TcpClient	BeginConnect	开始一个对远程主机连接的异步请求
	EndConnect	异步接受传入的连接尝试
Socket	BeginReceive	开始从连接的Socket中异步接收数据
	EndReceive	结束挂起的异步读取
	BeginSend	将数据异步发送到连接的Socket
	EndSend	结束挂起的异步发送

2.3.1　EventWaitHandle 类

虽然我们可以利用异步操作并行完成一系列功能，但是现实中的很多工作是相互关联的，某些工作必须要等另一个工作完成后才能继续。这个问题就是异步操作中的同步问题。

EventWaitHandle 类用于在异步操作时控制线程间的同步，即控制一个或多个线程继续执行或者等待其他线程完成。考虑这样一种情况：假设有两个线程，一个是写线程，一个是读线程，两个线程是并行运行的。下面是实现代码：

```
using System;
using System.Threading;
class Program
{
    private int n1, n2, n3;
    static void Main(string[] args)
    {
        Program p = new Program();
        Thread t0 = new Thread(new ThreadStart(p.WriteThread));
        Thread t1 = new Thread(new ThreadStart(p.ReadThread));
        t0.Start();
        t1.Start();
        Console.ReadLine();
    }
    private void WriteThread()
    {
        Console.WriteLine("t1");
        n1 = 1;
        n2 = 2;
        n3 = 3;
    }
    private void ReadThread()
    {
        Console.WriteLine("{0}+{1}+{2}={3}", n1, n2, n3, n1 + n2 + n3);
    }
}
```

运行这个程序，输出结果为：

```
t1
```

0+0+0=0

按照一般的思维逻辑，读线程执行的结果应该是 1+2+3=6，可实际运行的结果却是0+0+0=0。显然读线程输出的内容是在写线程尚未写入新值之前得到的结果。如果把这个问题一般化，即某些工作是在线程内部完成的，同时启动多个线程后，我们无法准确判断线程内部处理这些工作的具体时间，而又希望保证一个线程完成某些工作后，另一个线程才能在这个基础上继续运行，最好的办法是什么呢？

这个问题实际上就是如何同步线程的问题。在 System.Threading 命名空间中，有一个EventWaitHandle 类，它能够让操作系统通过发出信号完成多个线程之间的同步，需要同步的线程可以先阻塞当前线程，然后根据 Windows 操作系统发出的信号，决定是继续阻塞等待其他工作完成，还是不再等待而直接继续执行。

这一节涉及到的 EventWaitHandle 类提供的方法有：

Reset 方法：将信号的状态设置为非终止状态，即不让操作系统发出信号，从而导致等待收到信号才能继续执行的线程阻塞。

Set 方法：将事件状态设置为终止状态，这样等待的线程将会收到信号，从而继续执行而不再等待。

WaitOne 方法：阻塞当前线程，等待操作系统为其发出信号，直到收到信号才解除阻塞。

操作系统发出信号的方式有两种：

1) 发一个信号，使某个等待信号的线程解除阻塞，继续执行。

2) 发一个信号，使所有等待信号的线程全部解除阻塞，继续执行。

这种机制类似于面试，所有等待的线程都是等待面试者，所有等待的面试者均自动在外面排队等待。操作系统让考官负责面试，考官事先告诉大家他发的信号“继续”的含义有两个，一个是对某个等待面试者而言的，考官每次发信号“继续”，意思是只让一个面试者进去面试，其他面试者必须继续等待，至于谁进去，要看排队情况了，一般是排在最前面的那个人进去，这种方式叫自动重置（AutoResetEvent）；另一个是对所有面试者而言的，考官每次发信号“继续”，意思是让所有正在门外等待的面试者全部进来面试，当然对不等待的面试者无效，这种方式叫手动重置（ManualResetEvent）。

为什么说“每次”发信号呢？因为不一定所有考生都在外面等待，可能有些考生没有等在门外，所以他这次发出的“继续”只能对等待的面试者起作用，也许他发出这个信号后，又有面试者到了门外，因此可能需要多次发出“继续”的信号。

考官也可以不发任何信号，这样所有正在等待的面试者只能一直等待。

程序员可以认为是控制考官和面试者的“管理员”，程序员既可以告诉考官“不要发信号”（调用 EventWaitHandle 的 Reset 方法），也可以告诉考官“发信号”（调用 EventWaitHandle 的 Set 方法），同时还可以决定面试者什么时候去参加面试（调用 EventWaitHandle 的 WaitOne 方法）。

利用 EventWaitHandle 类，我们可以将上面的代码修改为：

```
using System;
using System.Threading;
class Program
{
    private int n1, n2, n3;
    //将信号状态设置为非终止，使用手动重置
```

```
        EventWaitHandle myEventWaitHandle =
            new EventWaitHandle(false, EventResetMode.ManualReset);
        static void Main(string[] args)
        {
            Program p = new Program();
            Thread t0 = new Thread(new ThreadStart(p.WriteThread));
            Thread t1 = new Thread(new ThreadStart(p.ReadThread));
            t0.Start();
            t1.Start();
            Console.ReadLine();
        }
        private void WriteThread()
        {
            //允许其他需要等待的线程阻塞
            myEventWaitHandle.Reset();
            Console.WriteLine("t1");
            n1 = 1;
            n2 = 2;
            n3 = 3;
            //允许其他等待的线程继续
            myEventWaitHandle.Set();
        }
        private void ReadThread()
        {
            //阻塞当前线程，直到收到信号
            myEventWaitHandle.WaitOne();
            Console.WriteLine("{0}+{1}+{2}={3}", n1, n2, n3, n1 + n2 + n3);
        }
    }
```

程序中增加了一个 EventWaitHandle 类型的对象 myEventWaitHandle，在 WriteThread 线程开始时，首先让调用 WaitOne 方法的线程阻塞，然后继续执行该线程，当任务完成时，向所有调用 WaitOne 方法的线程发出可以继续执行的事件句柄信号。而 ReadThread 一开始就将自己阻塞了，当 WriteThread 执行 Set 方法后才继续往下执行，因此其 WriteLine 语句输出的结果为 1+2+3=6，达到了我们预期的效果。

在异步操作中，为了让具有先后关联关系的线程同步，即让其按照希望的顺序执行，均可以调用 EventWaitHandle 类提供的 Reset、Set 和 WaitOne 方法。

2.3.2 AsyncCallback 委托

AsyncCallback 委托用于引用异步操作完成时调用的回调方法。在异步操作方式下，由于程序可以在启动异步操作后继续执行其他代码，因此必须有一种机制，以保证该异步操作完成时能及时通知调用者。这种机制可以通过 AsyncCallback 委托实现。

异步操作的每一个方法都有一个 Begin…方法和 End…方法，例如 BeginAcceptTcpClient 和 EndAcceptTcpClient。程序调用 Begin…方法时，系统会自动在线程池中创建对应的线程进行异步操作，从而保证调用方和被调用方同时执行，当线程池中的 Begin…方法执行完毕时，会自动通过 AsyncCallback 委托调用在 Begin…方法的参数中指定的回调方法。

回调方法是在程序中事先定义的，在回调方法中，通过 End…方法获取 Begin…方法的返

回值和所有输入/输出参数，从而达到异步操作方式下完成参数传递的目的。

2.3.3 BeginAcceptTcpClient 方法和 EndAcceptTcpClient 方法

BeginAcceptTcpClient 和 EndAcceptTcpClient 方法包含在 System.Net.Sockets 命名空间下的 TcpListener 类中。在异步 TCP 应用编程中，服务器端可以使用 TcpListener 类提供的 BeginAcceptTcpClient 方法开始接收新的客户端连接请求。在这个方法中，系统自动利用线程池创建需要的线程，并在操作完成时利用异步回调机制调用提供给它的方法，同时返回相应的状态参数，其方法原型为：

public IAsyncResult BeginAcceptTcpClient(AsyncCallback callback, Object state)

其中：参数 1 为 AsyncCallback 类型的委托；参数 2 为 Object 类型，用于将状态信息传递给委托提供的方法。例如：

```
AsyncCallback callback = new AsyncCallback(AcceptTcpClientCallback);
tcpListener.BeginAcceptTcpClient(callback, tcpListener);
```

程序执行 BeginAcceptTcpClient 方法后，即在线程池中自动创建需要的线程，同时在自动创建的线程中监听客户端连接请求。一旦接受了客户连接请求，就自动通过委托调用提供给委托的方法，并返回状态信息。这里我们给委托自动调用的方法命名为 AcceptTcpClientCallback，状态信息定义为 TcpListener 类型的实例 tcpListener。在程序中，定义该方法的格式为：

```
void AcceptTcpClientCallback( IAsyncResult ar)
{
    回调代码
}
```

方法中传递的参数只有一个，而且必须是 IAsyncResult 类型的接口，它表示异步操作的状态。由于我们定义了委托提供的方法（即 AcceptTcpClientCallback 方法），因此系统会自动将该状态信息从关联的 BeginAcceptTcpClient 方法传递到 AcceptTcpClientCallback 方法。注意在回调代码中，必须调用 EndAcceptTcpClient 方法完成客户端连接。关键代码为：

```
void AcceptTcpClientCallback( IAsyncResult ar)
{
    ……
    TcpListener myListener = (TcpListener)ar.AsyncState;
    TcpClient client = myListener.EndAcceptTcpClient(ar);
    ……
}
```

程序执行 EndAcceptTcpClient 方法后，会自动完成客户端连接请求，并返回包含底层套接字的 TcpClient 对象，接下来我们就可以利用这个对象与客户端进行通信了。

默认情况下，程序执行 BeginAcceptTcpClient 方法后，在该方法返回状态信息之前，不会像同步 TCP 方式那样阻塞等待客户端连接，而是继续往下执行。如果我们希望在其返回状态信息之前阻塞当前线程的执行，可以调用 ManualResetEvent 对象的 WaitOne 方法。

2.3.4 BeginConnect 方法和 EndConnect 方法

BeginConnect 方法和 EndConnect 方法包含在命名空间 System.Net.Sockets 下的 TcpClient 类和 Socket 类中，这里我们只讨论 TcpClient 类中的方法。

在异步 TCP 应用编程中，BeginConnect 方法通过异步方式向远程主机发出连接请求。该

方法有三种重载的形式，方法原型为：

public IAsyncResult BeginConnect(IPAddress address, int port, AsyncCallback requestCallback, Object state);

public IAsyncResult BeginConnect(IPAddress[] addresses, int port, AsyncCallback requestCallback, Object state);

public IAsyncResult BeginConnect(string host, int port, AsyncCallback requestCallback, Object state);

其中 address 为远程主机的 IPAddress 对象；port 为远程主机的端口号；requestCallback 为 AsyncCallback 类型的委托；state 为包含连接操作的相关信息，当操作完成时，此对象会被传递给 requestCallback 委托。

BeginConnect 方法在操作完成前不会阻塞，程序中调用 BeginConnect 方法时，系统会自动用独立的线程来执行该方法，直到与远程主机连接成功或抛出异常。如果在调用 BeginConnect 方法之后想阻塞当前线程，可以调用 ManualResetEvent 对象的 WaitOne 方法。

异步 BeginConnect 方法只有在调用了 EndConnect 方法之后才算执行完毕。因此程序中需要在提供给 requestCallback 委托调用的方法中调用 TcpClient 对象的 EndConnect 方法。关键代码为：

```
……
AsyncCallback requestCallback = new AsyncCallback(RequestCallback);
tcpClient.BeginConnect(远程主机 IP 或域名,远程主机端口号 , requestCallback, tcpClient);
……
void RequestCallback(IAsyncResult ar)
{
    ……
    tcpClient = (TcpClient)ar.AsyncState;
    client.EndConnect(ar);
    ……
}
```

在自定义的 RequestCallback 中，通过获取的状态信息得到新的 TcpClient 类型的对象，并调用 EndConnect 结束连接请求。

2.3.5 发送数据

在异步 TCP 应用编程中，如果本机已经和远程主机建立连接，就可以用 System.Net.Sockets 命名空间下 NetworkStream 类中的 BeginWrite 方法发送数据。其方法原型为：

public override IAsyncResult BeginWrite(byte[] buffer, int offset, int size, AsyncCallback callback, Object state);

其中 buffer 是一组 Byte 类型的值，用来存放要发送的数据。offset 用来存放发送的数据在发送缓冲区中的起始位置。size 用来存放发送数据的字节数。callback 是异步回调类型的委托，state 包含状态信息。

BeginWrite 方法用于向一个已经成功连接的套接字异步发送数据。程序中调用 BeginWrite 方法后，系统会自动在内部产生的单独执行的线程中发送数据。

使用 BeginWrite 方法异步发送数据，程序必须创建实现 AsyncCallback 委托的回调方法，并将其名称传递给 BeginWrite 方法。在 BeginWrite 方法中，传递的 state 参数必须至少包含 NetworkStream 对象。如果回调需要更多信息，则可以创建一个小型类或结构，用于保存

NetworkStream 和其他所需的信息，并通过 state 参数将结构或类的实例传递给 BeginWrite 方法。

在回调方法中，必须调用 EndWrite 方法。程序调用 BeginWrite 后，系统自动使用单独的线程来执行指定的回调方法，并在 EndWrite 上一直处于阻塞状态，直到 NetworkStream 对象发送请求的字节数或引发异常。关键代码为：

```
……
NetworkStream stream = tcpClient.GetStream();
……
byte[] bytesData = System.Text.Encoding.UTF8.GetBytes(str + "\r\n");
stream.BeginWrite(bytesData, 0, bytesData.Length, new AsyncCallback(SendCallback), stream);
stream.Flush();
……
private void SendCallback(IAsyncResult ar)
{
    ……
    stream.EndWrite(ar);
    ……
}
```

如果希望在 BeginWrite 方法得到传递的状态信息之前使当前线程（即调用 BeginWrite 方法的线程）阻塞，可以使用 ManualResetEvent 对象 WaitOne 方法。并在回调方法中调用 Set 使当前线程继续执行。

2.3.6 接收数据

与发送数据相似，如果本机已经和远程主机建立了连接，就可以用 System.Net.Sockets 命名空间下 NetworkStream 类中的 BeginRead 方法接收数据。其方法原型为：

```
public override IAsyncResult BeginRead(byte[] buffer, int offset, int size, AsyncCallback callback,
Object state);
```

其中 buffer 为字节数组，存储从 NetworkStream 读取的数据；offset 为 buffer 中开始存储数据的位置；size 为从 NetworkStream 中读取的字节数；callback 是在 BeginRead 完成时执行的 AsyncCallback 委托；state 包含用户定义的任何附加数据的对象。

BeginRead 方法启动从传入网络缓冲区中异步读取数据的操作。调用 BeginRead 方法后，系统自动在单独的执行线程中接收数据。

在程序中，必须创建实现 AsyncCallback 委托的回调方法，并将其名称传递给 BeginRead 方法。state 参数必须至少包含 NetworkStream 对象。一般情况下，我们希望在回调方法内获得所接收的数据，因此应创建小型的类或结构来保存读取缓冲区以及其他任何有用的信息，并通过 state 参数将结构或类的实例传递给 BeginRead 方法。

在回调方法中，必须调用 EndRead 方法完成读取操作。系统执行 BeginRead 时，将一直等待直到数据接收完毕或者遇到错误，从而得到可用的字节数，然后自动使用一个单独的线程来执行指定的回调方法，并阻塞 EndRead 方法，直到所提供的 NetworkStream 对象将可用数据读取完毕，或者达到 size 参数指定的字节数。关键代码为：

```
……
NetworkStream stream = tcpClient.GetStream();
……
```

```
ReadObject readObject = new ReadObject(stream, client.ReceiveBufferSize);
stream.BeginRead(readObject.bytes, 0, readObject.bytes.Length, ReadCallback, readObject);
……
void ReadCallback(IAsyncResult ar)
{
    ……
    ReadObject readObject = (ReadObject)ar.AsyncState;
    int count = readObject.netStream.EndRead(ar);
    //处理读取的保存在 ReadObject 类中的字节数组
    ……
}
```

如果希望在调用 BeginRead 方法之后使当前线程（即调用 BeginRead 的线程）阻塞，可以使用 ManualResetEvent 对象的 WaitOne 方法，并在回调方法中调用该对象的 Set 方法使当前线程继续执行。

2.4 异步 TCP 聊天程序

本节通过一个简单的可以同时与多个客户进行异步聊天的系统说明编写异步 TCP 应用程序的方法。

【例 2-2】编写一个异步聊天系统，系统只允许客户与服务器相互传送字符串数据，不实现任意客户之间的直接信息传递。服务器要允许多个客户同时向服务器发送字符串，服务器也可以随时向任意一个指定的客户发送信息。

这个系统虽然功能简单，实现的代码也不多，但是代码中涉及的概念要比同步 TCP 编程复杂很多，理解起来也相对比较困难。可是如果真正把相关概念搞明白了，也没有什么难点可言。由于基于 TCP 协议的大型应用系统多数都采用异步工作方式，因此希望读者尽可能掌握本节的主要设计思想，为编写复杂的网络应用程序打下基础。

2.4.1 服务器端设计

根据系统要求，服务器必须能识别不同的客户，而且需要指明与哪个客户通信，具体编写步骤为：

(1) 创建一个名为 AsyncTcpServer 的 Windows 应用程序项目，将 Form1.cs 换名为 FormServer.cs，设计界面如图 2-7 所示。

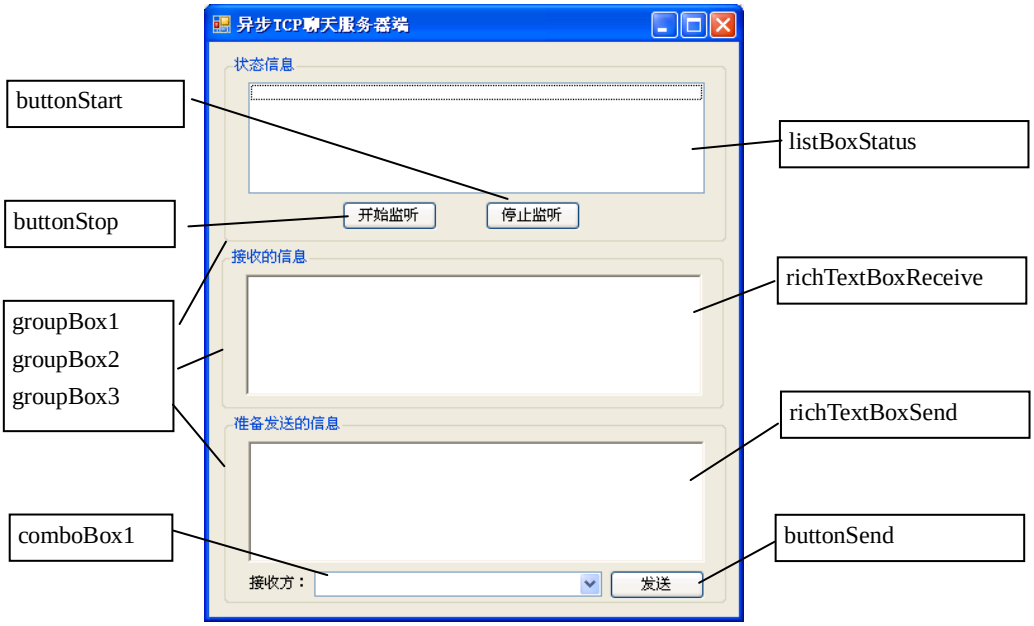


图2-7　FormServer.cs的设计界面

(2) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 ReadWriteObject.cs，用于保存接收和发送数据需要的参数。代码如下：

```
//--------ReadWriteObject.cs---------//
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
namespace AsyncTcpServer
{
    class ReadWriteObject
    {
        public TcpClient client;
        public NetworkStream netStream;
        public byte[] readBytes;
        public byte[] writeBytes;
        public ReadWriteObject(TcpClient client)
        {
            this.client = client;
            netStream = client.GetStream();
            readBytes = new byte[client.ReceiveBufferSize];
            writeBytes = new byte[client.SendBufferSize];
        }
        public void InitReadArray()
        {
            readBytes = new byte[client.ReceiveBufferSize];
        }
        public void InitWriteArray()
        {
```

```
                writeBytes = new byte[client.SendBufferSize];
            }
        }
    }
```

(3) 切换到 FormServer 的代码编辑方式下，添加对应按钮的 Click 事件以及其他代码，源程序如下：

```
//-----FormServr.cs------//
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;
namespace AsyncTcpServer
{
    public partial class FormServer : Form
    {
        private bool isExit = false;
        //保存连接的所有客户端
        System.Collections.ArrayList clientList = new System.Collections.ArrayList();
        TcpListener listener;
        //用于线程间互操作
        private delegate void SetListBoxCallback(string str);
        private SetListBoxCallback setListBoxCallback;
        private delegate void SetRichTextBoxCallback(string str);
        private SetRichTextBoxCallback setRichTextBoxCallback;
        private delegate void SetComboBoxCallback(string str);
        private SetComboBoxCallback setComboBoxCallback;
        private delegate void RemoveComboBoxItemsCallback(ReadWriteObject readWriteObject);
        private RemoveComboBoxItemsCallback removeComboBoxItemsCallback;
        //用于线程同步，初始状态设为非终止状态，使用手动重置方式
        private EventWaitHandle  allDone =
            new EventWaitHandle(false, EventResetMode.ManualReset);
        public FormServer()
        {
            InitializeComponent();
            listBoxStatus.HorizontalScrollbar = true;
            setListBoxCallback = new SetListBoxCallback(SetListBox);
            setRichTextBoxCallback = new SetRichTextBoxCallback(SetReceiveText);
            setComboBoxCallback = new SetComboBoxCallback(SetComboBox);
            removeComboBoxItemsCallback =
                new RemoveComboBoxItemsCallback(RemoveComboBoxItems);
        }
        private void buttonStart_Click(object sender, EventArgs e)
        {
```

```
        //由于服务器要为多个客户服务，所以需要创建一个线程监听客户端连接请求
        ThreadStart ts = new ThreadStart(AcceptConnect);
        Thread myThread = new Thread(ts);
        myThread.Start();
        buttonStart.Enabled = false;
        buttonStop.Enabled = true;
    }
    private void AcceptConnect()
    {
        //获取本机所有 IP 地址
        IPAddress[] ip = Dns.GetHostAddresses(Dns.GetHostName());
        listener = new TcpListener(ip[0], 51888);
        listener.Start();
        while (isExit == false)
        {
            try
            {
                //将事件的状态设为非终止
                allDone.Reset();
                //引用在异步操作完成时调用的回调方法
                AsyncCallback callback = new AsyncCallback(AcceptTcpClientCallback);
                listBoxStatus.Invoke(setListBoxCallback, "开始等待客户连接");
                //开始一个异步操作接受传入的连接尝试
                listener.BeginAcceptTcpClient(callback, listener);
                //阻塞当前线程，直到收到客户连接信号
                allDone.WaitOne();
            }
            catch (Exception err)
            {
                listBoxStatus.Invoke(setListBoxCallback, err.Message);
                break;
            }
        }
    }
    //ar 是 IAsyncResult 类型的接口，表示异步操作的状态
    //是由 listener.BeginAcceptTcpClient(callback, listener)传递过来的
    private void AcceptTcpClientCallback(IAsyncResult ar)
    {
        try
        {
            //将事件状态设为终止状态，允许一个或多个等待线程继续
            allDone.Set();
            TcpListener myListener = (TcpListener)ar.AsyncState;
            //异步接收传入的连接，并创建新的 TcpClient 对象处理远程主机通信
            TcpClient client = myListener.EndAcceptTcpClient(ar);
            listBoxStatus.Invoke(setListBoxCallback,
                "已接受客户连接：" + client.Client.RemoteEndPoint);
            comboBox1.Invoke(setComboBoxCallback,
                client.Client.RemoteEndPoint.ToString());
            ReadWriteObject readWriteObject = new ReadWriteObject(client);
            clientList.Add(readWriteObject);
```

```
            SendString(readWriteObject, "服务器已经接受连接，请通话！");
            readWriteObject.netStream.BeginRead(readWriteObject.readBytes,
                0, readWriteObject.readBytes.Length, ReadCallback, readWriteObject);
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
            return;
        }
    }
    private void ReadCallback(IAsyncResult ar)
    {
        try
        {
            ReadWriteObject readWriteObject = (ReadWriteObject)ar.AsyncState;
            int count = readWriteObject.netStream.EndRead(ar);
            richTextBoxReceive.Invoke(setRichTextBoxCallback,
                string.Format("[来自{0}]{1}", readWriteObject.client.Client.RemoteEndPoint,
                System.Text.Encoding.UTF8.GetString(readWriteObject.readBytes,
                0, count)));
            if (isExit == false)
            {
                readWriteObject.InitReadArray();
                readWriteObject.netStream.BeginRead(readWriteObject.readBytes,
                    0, readWriteObject.readBytes.Length, ReadCallback, readWriteObject);
            }
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
        }
    }
    private void SendString(ReadWriteObject readWriteObject, string str)
    {
        try
        {
            readWriteObject.writeBytes = System.Text.Encoding.UTF8.GetBytes(str + "\r\n");
            readWriteObject.netStream.BeginWrite(readWriteObject.writeBytes,
                0, readWriteObject.writeBytes.Length,
                new AsyncCallback(SendCallback), readWriteObject);
            readWriteObject.netStream.Flush();
            listBoxStatus.Invoke(setListBoxCallback,
                string.Format("向{0}发送：{1}",
                readWriteObject.client.Client.RemoteEndPoint, str));
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
        }
    }
    private void SendCallback(IAsyncResult ar)
```

```
{
    ReadWriteObject readWriteObject = (ReadWriteObject)ar.AsyncState;
    try
    {
        readWriteObject.netStream.EndWrite(ar);
    }
    catch (Exception err)
    {
        listBoxStatus.Invoke(setListBoxCallback, err.Message);
        comboBox1.Invoke(removeComboBoxItemsCallback, readWriteObject);
    }
}
private void RemoveComboBoxItems(ReadWriteObject readWriteObject)
{
    int index = clientList.IndexOf(readWriteObject);
    comboBox1.Items.RemoveAt(index);
}
private void SetListBox(string str)
{
    listBoxStatus.Items.Add(str);
    listBoxStatus.SelectedIndex = listBoxStatus.Items.Count - 1;
    listBoxStatus.ClearSelected();
}
private void SetReceiveText(string str)
{
    richTextBoxReceive.AppendText(str);
}
private void SetComboBox(object obj)
{
    comboBox1.Items.Add(obj);
}
//【停止监听】按钮的 Click 事件
private void buttonStop_Click(object sender, EventArgs e)
{
    //使线程自动结束
    isExit = true;
    //将事件状态设置为终止状态，允许一个或者多个等待线程继续
    //从而使线程正常结束
    allDone.Set();
    buttonStart.Enabled = true;
    buttonStop.Enabled = false;
}
//【发送】按钮的 Click 事件
private void buttonSend_Click(object sender, EventArgs e)
{
    int index = comboBox1.SelectedIndex;
    if (index == -1)
    {
        MessageBox.Show("请先选择接收方，然后再单击［发送］");
    }
    else
```

```
            {
                ReadWriteObject obj = (ReadWriteObject)clientList[index];
                SendString(obj, richTextBoxSend.Text);
                richTextBoxSend.Clear();
            }
        }
        //关闭窗体前触发的事件
        private void FormServer_FormClosing(object sender, FormClosingEventArgs e)
        {
            buttonStop_Click(null, null);
        }
    }
}
```

(4) 按<F5>键编译并运行，保证无编译错误，然后退出。

2.4.2 客户端设计

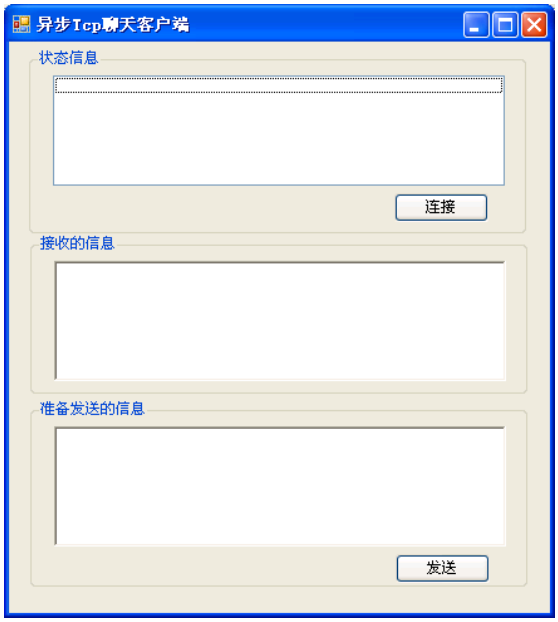


图2-8 FormClient.cs的设计界面

与同步 TCP 应用编程相似，异步 TCP 客户端同样需要先和服务器建立连接，然后才能接收和发送数据，具体实现步骤为：

(1) 创建一个名为 AsyncTcpClient 的 Windows 应用程序项目，将 Form1.cs 换名为 FormClient.cs，设计界面如图 2-8 所示。

(2) 在解决方案资源管理器中，鼠标右击项目名，选择【添加】→【类】，添加一个类文件 ReadObject.cs，用于保存接收和发送数据需要的参数。代码如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
namespace AsyncTcpClient
{
    //用于回调参数
    public class ReadObject
    {
        public NetworkStream netStream;
        public byte[] bytes;
        public ReadObject(NetworkStream netStream, int bufferSize)
        {
            this.netStream = netStream;
            bytes = new byte[bufferSize];
        }
    }
}
```

(3) 切换到 FormClient 的代码编辑方式下，添加对应按钮的 Click 事件以及其他代码，源程序如下：

```
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;
using System.Threading;
namespace AsyncTcpClient
{
    public partial class FormClient : Form
    {
        private bool isExit = false;
        //用于线程间互操作
        private delegate void SetListBoxCallback(string str);
        private SetListBoxCallback setListBoxCallback;
        private delegate void SetRichTextBoxReceiveCallback(string str);
        private SetRichTextBoxReceiveCallback setRichTextBoxReceiveCallback;
        private TcpClient client;
        private NetworkStream networkStream;
        //用于线程同步，初始状态设为非终止状态，使用手动重置方式
        private EventWaitHandle  allDone =
            new EventWaitHandle(false, EventResetMode.ManualReset);
        public FormClient()
        {
            InitializeComponent();
            listBoxStatus.HorizontalScrollbar = true;
            setListBoxCallback = new SetListBoxCallback(SetListBox);
            setRichTextBoxReceiveCallback =
                new SetRichTextBoxReceiveCallback(SetRichTextBoxReceive);
        }
        private void buttonConnect_Click(object sender, EventArgs e)
        {
            //使用 IPv4
            client = new TcpClient(AddressFamily.InterNetwork);
            //实际使用时要将 Dns.GetHostName()变为服务器域名或 IP 地址
            IPAddress[] serverIP = Dns.GetHostAddresses(Dns.GetHostName());
            //创建一个委托，让其引用在异步操作完成时调用的回调方法
            AsyncCallback requestCallback = new AsyncCallback(RequestCallback);
            //将事件的状态设为非终止状态
            allDone.Reset();
            //开始一个对远程主机的异步请求
            client.BeginConnect(serverIP[0], 51888, requestCallback, client);
            listBoxStatus.Invoke(setListBoxCallback, string.Format("本机 EndPoint：{0}",
                client.Client.LocalEndPoint));
            listBoxStatus.Invoke(setListBoxCallback, "开始与服务器建立连接");
            //阻塞当前线程，即窗体界面不再响应任何用户操作，等待 BeginConnect 完成
            //这样做的目的是为了与服务器连接有结果（成功或失败）时，才能继续
            //当 BeginConnect 完成时，会自动调用 RequestCallback，
```

```
        //通过在 RequestCallback 中调用 Set 方法解除阻塞
        allDone.WaitOne();
    }
    //ar 是 IAsyncResult 类型的接口，表示异步操作的状态
    //是由 listener.BeginAcceptTcpClient(callback, listener)传递过来的
    private void RequestCallback(IAsyncResult ar)
    {
        //异步操作能执行到此处，说明调用 BeginConnect 已经完成，
        //并得到了 IAsyncResult 类型的状态参数 ar，但 BeginConnect 尚未结束
        //此时需要解除阻塞,以便能调用 EndConnect
        allDone.Set();
        //调用 Set 后，事件状态变为终止状态，当前线程继续，
        //buttonConnect_Click 执行结束，
        //同时窗体界面可以响应用户操作
        try
        {
            //获取连接成功后得到的状态参数
            client = (TcpClient)ar.AsyncState;
            //异步接受传入的连接尝试，使 BeginConnect 正常结束
            client.EndConnect(ar);
            listBoxStatus.Invoke(setListBoxCallback,
                 string.Format("与服务器{0}连接成功", client.Client.RemoteEndPoint));
            //获取接收和发送数据的网络流
            networkStream = client.GetStream();
            //异步接收服务器发送的数据，BeginRead 完成后，会自动调用 ReadCallback
            ReadObject readObject =
                new ReadObject(networkStream, client.ReceiveBufferSize);
            networkStream.BeginRead(readObject.bytes,
                0, readObject.bytes.Length, ReadCallback, readObject);
            // allDone.WaitOne();
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
            return;
        }
    }
    private void ReadCallback(IAsyncResult ar)
    {
        //异步操作能执行到此处，说明调用 BeginRead 已经完成
        try
        {
            ReadObject readObject = (ReadObject)ar.AsyncState;
            int count = readObject.netStream.EndRead(ar);
            richTextBoxReceive.Invoke(setRichTextBoxReceiveCallback,
                 System.Text.Encoding.UTF8.GetString(readObject.bytes, 0, count));
            if (isExit == false)
            {
                //重新调用 BeginRead 进行异步读取
                readObject = new ReadObject(networkStream, client.ReceiveBufferSize);
                networkStream.BeginRead(readObject.bytes,
```

```
                    0, readObject.bytes.Length, ReadCallback, readObject);
            }
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
        }
    }
    private void SendString(string str)
    {
        try
        {
            byte[] bytesData = System.Text.Encoding.UTF8.GetBytes(str + "\r\n");
            networkStream.BeginWrite(bytesData,
                0, bytesData.Length, new AsyncCallback(SendCallback), networkStream);
            networkStream.Flush();
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
        }
    }
    private void SendCallback(IAsyncResult ar)
    {
        try
        {
            networkStream.EndWrite(ar);
        }
        catch (Exception err)
        {
            listBoxStatus.Invoke(setListBoxCallback, err.Message);
        }
    }
    private void SetListBox(string str)
    {
        listBoxStatus.Items.Add(str);
        listBoxStatus.SelectedIndex = listBoxStatus.Items.Count - 1;
        listBoxStatus.ClearSelected();
    }
    private void SetRichTextBoxReceive(string str)
    {
        richTextBoxReceive.AppendText(str);
    }
    private void buttonSend_Click(object sender, EventArgs e)
    {
        SendString(richTextBoxSend.Text);
        richTextBoxSend.Clear();
    }
    private void FormClient_FormClosing(object sender, FormClosingEventArgs e)
    {
        isExit = true;
```

```
                allDone.Set();
            }
        }
    }
```

(4) 分别运行服务器端程序和客户端程序，运行效果如图 2-9 所示。

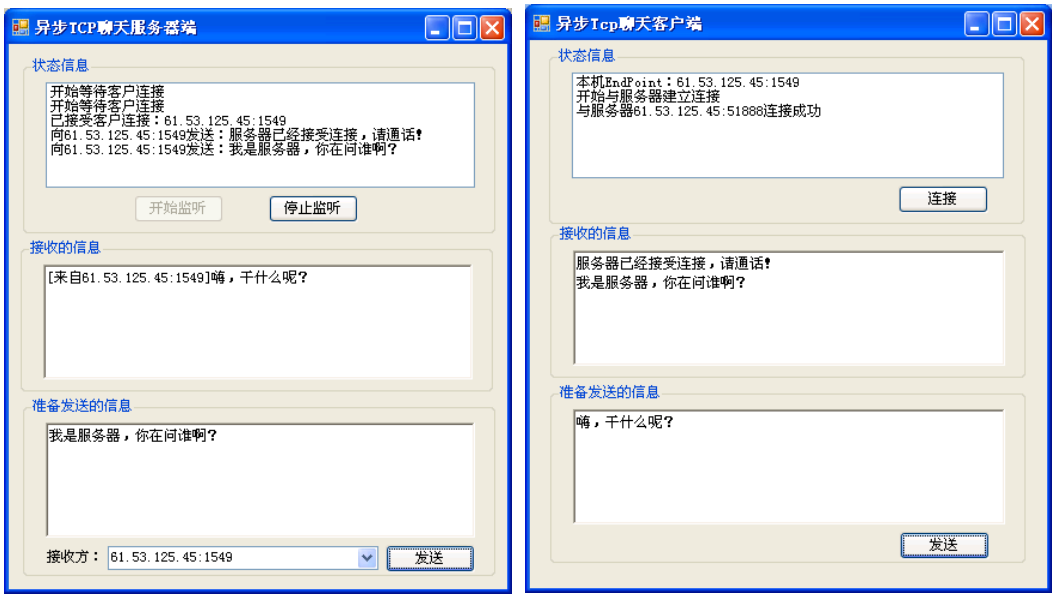


图2-9　异步TCP聊天服务器和客户端部分运行效果

在服务器端程序运行中，由于执行 AcceptTcpClientCallback 中的 allDone.Set();导致线程 AcceptConnect 解除阻塞，而该线程继续执行后，输出“开始等待客户连接”的语句比 AcceptTcpClientCallback 中输出“已接受客户连接”的语句要早，所以服务器端运行界面显示了两次“开始等待客户连接”，实际上第二次显示的含义是等待第二个客户连接。如果不想让其多次显示，只需要将语句 listBoxStatus.Invoke(setListBoxCallback, "开始等待客户连接");放到 while 循环的上面即可。例子中故意放在此处，目的是为了让读者进一步理解不同线程同时执行时各自在窗体上显示的效果。

2.5　习题 2

1. 简要回答使用 TCP 协议进行同步套接字编程中，服务器端和客户端的工作流程。
2. 已知本地主机的名称为 hostName，写出获取本地主机 IP 地址的代码。
3. 如果在服务器进行监听的同时进行其他操作，异步套接字需要提供那些方法?

第 3 章　UDP 应用编程

UDP 是 User Datagram Protocol 的缩写，意思是用户数据报协议。本章首先介绍 UDP 协议的特点、工作方式以及与 TCP 协议的区别，然后通过一些例子介绍如何利用 UDP 协议编写网络应用程序。

3.1　UDP 协议基础知识

UDP 是一个简单的、面向数据报的无连接协议，提供了快速但不一定可靠的传输服务。与 TCP 一样，UDP 也是构建于底层 IP 协议之上的传输层协议。所谓“无连接”是在正式通信前不必与对方先建立连接，不管对方状态如何就直接发送过去。这与发手机短信非常相似，只要输入对方的手机号码就可以了，不用考虑对方手机处于什么状态。

利用 UDP 协议可以使用广播的方式同时向子网上的所有设备发送信息，也可以使用组播的方式同时向网络上的多个设备发送信息。比如可以使用 UDP 协议向某网络发送广告，也可以使用 UDP 协议向指定的客户发送订阅的新闻或通知。

与 TCP 相比，UDP 有如下一些特点：首先，UDP 协议是基于无连接的协议，它能够消除生成连接的系统延迟，所以速度比 TCP 更快。对于强调传输性能而不是传输完整性的应用（例如音频和多媒体应用），UDP 是最好的选择；其次，UDP 不但支持一对一连接，而且也支持一对多连接，可以使用广播的方式多地址发送，而 TCP 仅支持一对一的通信；第三，UDP 与 TCP 的报头比是 8：20，这使得 UDP 消耗的网络带宽更少。最后，UDP 协议传输的数据有消息边界，而 TCP 协议没有消息边界。

由于 UDP 是一种无连接的协议，缺乏双方的握手信号，因此发送方无法了解数据报是否已经到达目标主机。如果在从发送方到接收方的传递过程中出现了数据报的丢失，协议本身并不能做出任何检测或提示，因此可靠性不如 TCP。

UDP 没有任何对双方会话的支持，当接收多个数据报时，不能保证各数据包到达的顺序与发出的顺序相同。当然，UDP 协议的这种乱序性基本上很少出现，通常只会在网络非常拥挤的情况下才可能发生。

3.2　UDP 应用编程技术

编写 UDP 应用程序时，有两种技术，一种是直接使用 Socket 类，另一种是使用 UdpClient 类。UdpClient 类对基础 Socket 进行了封装，发送和接收数据时不必考虑底层套接字收发时必须处理的一些细节问题，从而简化了 UDP 应用编程的难度，提高了编程效率，。

这一节我们先了解 UdpClient 类的相关知识，然后通过一个具体例子学习通过 UdpClient 对象发送和接收数据的基本技术。

3.2.1　UdpClient 类

System.Net.Sockets 名称空间下的 UdpClient 类简化了 UDP 套接字编程。与 TCP 协议有

TcpListener 类和 TcpClient 类不同，UDP 协议只有 UdpClient 类，这是因为 UDP 协议是无连接的协议，所以只需要一种 Socket。该类提供了发送和接收无连接的 UDP 数据报的方便的方法。其建立默认远程主机的方式有两种：一是使用远程主机名和端口号作为参数创建 UdpClient 类的实例；另一种是先创建不带参数的 UdpClient 类的实例，然后调用 Connect 方法指定默认远程主机。

1. UdpClient 的构造函数

UdpClient 类提供了以下几种常用格式的构造函数：

1) UdpClient()

创建一个新的 UdpClient 对象，并自动分配合适的本地 IPv4 地址和端口号。例如：

```
UdpClient udpClient = new UdpClient();
//指定默认远程主机和端口号
udpClient.Connect("www.contoso.com", 51666);
Byte[] sendBytes = System.Text.Encoding.Unicode.GetBytes("你好!");
//发送给默认远程主机
udpClient.Send(sendBytes, sendBytes.Length);
```

2) UdpClient(int port)

创建一个与指定的端口绑定的新的 UdpClient 实例，并自动分配合适的本地 IPv4 地址。例如：

```
UdpClient udpClient = new UdpClient(51666);
```

3) UdpClient(IPEndPoint localEp)

创建一个新的 UdpClient 实例，该实例与包含本地 IP 地址和端口号的 IPEndPoint 实例绑定。例如：

```
IPAddress address = IPAddress. Parse("127.0.0.1");
IPEndPoint iep = new IPEndPoint(address, 51666);
UdpClient udpClient =new UdpClient(iep);
```

4) UdpClient(string remoteHost,int port)

创建一个新的 UdpClient 实例，自动分配合适的本地 IP 地址和端口号，并将它与指定的远程主机和端口号联合。例如：

```
UdpClient udpClient =new UdpClient("www.contoso.com",8080);
```

使用这种构造函数，一般不必再调用 Connect 方法。

2. UdpClient 的常用方法和属性

表 3-1 列出了 UdpClient 的常用方法

表 3-1　UdpClient 的常用方法

方法	含义
Send()	发送数据报
Receive()	接收数据报
BeginSend()	开始从连接的socket中异步发送数据报
BeginReceive()	开始从连接的socket中异步接收数据报
EndSend()	结束挂起的异步发送数据报
EndReceive()	结束挂起的异步接收数据报
JoinMulticastG	添加多地址发送,用于连接一个多组播

roup()	
DropMulticast Group()	除去多地址发送,用于断开UdpClient与一个多组播的连接
Close()	关闭
Dispose()	释放资源

表 3-2 列出了 UdpClient 的常用属性

表 3-2　UdpClient 的常用属性

属性	含义
Active	获取或设置一个值指示是否已建立默认远程主机。
Available	获取或设置缓冲器中可用数据报的数量
Client	获取或设置基础网络套接字
EnableBroadcast	是否接收或发送广播包
ExclusiveAddressUSe	是否仅允许一个客户端使用指定端口

3.2.2　发送和接收数据的方法

编写基于 UDP 协议的应用程序时，关键在于如何实现数据的发送和接收。由于 UDP 协议不需要建立连接，因此可以在任何时候直接向网络中的任意主机发送 UDP 数据。在同步阻塞方式下，可以使用 UdpClient 对象的 Send 方法和 Receive 方法。

1. 发送数据

可以通过调用 UdpClient 对象的 Send 方法直接将数据发送到远程主机，该方法返回数据的长度可用于检查数据是否已被正确发送。

Send 方法有几种不同的重载形式，使用哪种方式取决于以下两点：一是 UdpClient 是如何连接到远程端口的，二是 UdpClient 实例是如何创建的。如果在调用 Send 方法以前没有指定任何远程主机的信息，则需要在调用中包括该信息。

1) Send(byte[] data, int length, IPEndPoint iep)

这种重载形式用于知道了远程主机 IP 地址和端口的情况下，它有三个参数：数据、数据长度、远程 IPEndPoint 对象。例如：

```
UdpClient    udpClient =new UdpClient();
//实际使用时应将 127.0.0.1 改为远程 IP
IPAddress remoteIPAdress = IPAddress.Parse("127.0.0.1");
IPEndPoint remoteIPEndPoint = new IPEndPoint(remoteIPAdress,51666);
byte[] sendBytes=System.Text.Encoding.Unicode.GetBytes("你好! ");
udpClient.Send(sendBytes , sendBytes.Length, remoteIPEndPoint);
```

2) Send(byte[] data, int length, string remoteHostName, int port)

这种重载形式用于知道了远程主机名和端口号的情况下，利用 Send 方法直接把 UDP 数据报发送到远程主机。例如：

```
UdpClient    udpClient =new UdpClient();
byte[] sendBytes= System.Text.Encoding.Unicode.GetBytes("你好!");
udpClient.Send(sendBytes , sendBytes.Length, "Host", 51666);
```

3) Send(byte[] data, int length)

这种重载形式假定 UDP 客户端已经通过 Connect 方法指定了默认的远程主机，因此，只

要用 Send 方法指定发送的数据和数据长度即可。例如：

```
UdpClient    udpClient =new UdpClient("remoteHost", 51666);
byte[] sendByte= System.Text.Encoding.Unicode.GetBytes("你好!") ;
udpClient.Send(sendBytes ,sendBytes.Length);
```

2. 接收数据

UdpClient 对象的 Receive 方法能够在指定的本地 IP 地址和端口上接收数据，该方法带一个引用类型的 IPEndPoint 实例，并将接收到的数据作为 byte 数组返回。例如：

```
IPEndPoint remoteIpEndPoint = new IPEndPoint(IPAddress.Any, 51666);
UdpClient udpClient = new UdpClient(remoteIpEndPoint);
IPEndPoint iep=new IPEndPoint(IPAddress.Any,0);
Byte[] receiveBytes = udpClient.Receive(ref    iep);
string receiveData = System.Text.Encoding.Unicode.GetString(receiveBytes);
Console.WriteLine("接收到信息："+receiveData);
```

使用 UdpClient 对象的 Receive 方法的优点是：当本机接收的数据报容量超过分配给它的缓冲区大小时，该方法能够自动调整缓冲区大小。而使用 Socket 对象遇到这种情况时，将会产生 SocketException 异常。可见，使用 UdpClient 的 Receive 方法轻而易举地解决了大量程序设计上的麻烦，提高了编程效率。

为了对这些相互配合的方式做个更具体的说明，下面开发一个简单的 UDP 网络聊天工具说明如何使用 UdpClient 类发送和接收数据。

【例 3-1】UdpClient 使用示例——UDP 网络聊天工具。

(1) 创建一个名为 UdpChatExample 的 Windows 应用程序，修改 Form1.cs 为 FormChat.cs，设计界面如图 3-1 所示。

图3-1 例3-1设计界面

(2) 添加对应的代码，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
```

```
using System.Net;
using System.Net.Sockets;
using System.Threading;
namespace UdpChatExample
{
    public partial class FormChat : Form
    {
        delegate void AddListBoxItemCallback(string text);
        AddListBoxItemCallback listBoxCallback;
        //使用的接收端口号
        private int port = 8001;
        private UdpClient udpClient;
        public FormChat()
        {
            InitializeComponent();
            listBoxCallback = new AddListBoxItemCallback(AddListBoxItem);
        }
        private void AddListBoxItem(string text)
        {
            //如果 listBoxReceive 被不同的线程访问则通过委托处理；
            if (listBoxReceive.InvokeRequired)
            {
                this.Invoke(listBoxCallback, text);
            }
            else
            {
                listBoxReceive.Items.Add(text);
                listBoxReceive.SelectedIndex = listBoxReceive.Items.Count - 1;
            }
        }
        /// <summary>
        ///  在后台运行的接收线程
        /// </summary>
        private void ReceiveData()
        {
            //在本机指定的端口接收
            udpClient = new UdpClient(port);
            IPEndPoint remote = null;
            //接收从远程主机发送过来的信息；
            while (true)
            {
                try
                {
                    //关闭 udpClient 时此句会产生异常
                    byte[] bytes = udpClient.Receive(ref remote);
                    string str = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
                    AddListBoxItem(string.Format("来自{0}：{1}", remote, str));
                }
                catch
                {
                    //退出循环，结束线程
```

```
                    break;
                }
            }
        }
        /// <summary>
        /// 发送数据到远程主机
        /// </summary>
        private void sendData()
        {
            UdpClient myUdpClient = new UdpClient();
            IPAddress remoteIP;
            if (IPAddress.TryParse(textBoxRemoteIP.Text, out remoteIP) == false)
            {
                MessageBox.Show("远程 IP 格式不正确");
                return;
            }
            IPEndPoint iep = new IPEndPoint(remoteIP, port);
            byte[] bytes = System.Text.Encoding.UTF8.GetBytes(textBoxSend.Text);
            try
            {
                myUdpClient.Send(bytes, bytes.Length, iep);
                textBoxSend.Clear();
                myUdpClient.Close();
                textBoxSend.Focus();
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message, "发送失败");
            }
            finally
            {
                myUdpClient.Close();
            }
        }
        private void FormChat_Load(object sender, EventArgs e)
        {
            //设置 listBox 样式
            listBoxReceive.HorizontalScrollbar = true;
            listBoxReceive.Dock = DockStyle.Fill;
            //获取本机第一个可用 IP 地址
            IPAddress myIP = (IPAddress)Dns.GetHostAddresses(Dns.GetHostName()).GetValue(0);
            //为了在同一台机器调试，此 IP 也作为默认远程 IP
            textBoxRemoteIP.Text = myIP.ToString();
            //创建一个线程接收远程主机发来的信息
            Thread myThread = new Thread(new ThreadStart(ReceiveData));
            //将线程设为后台运行
            myThread.IsBackground = true;
            myThread.Start();
            textBoxSend.Focus();
        }
        /// <summary>
```

```
        /// 单击发送按钮触发的事件
        /// </summary>
        private void buttonSend_Click(object sender, EventArgs e)
        {
            sendData();
        }
        /// <summary>
        /// 在 textBoxSend 中按下并释放 Enter 键后触发的事件
        /// </summary>
        private void textBoxData_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)Keys.Enter)
                sendData();
        }
        private void FormChat_FormClosing(object sender, FormClosingEventArgs e)
        {
            udpClient.Close();
        }
    }
}
```

(3) 按<F5>键编译并执行，向默认远程主机发送一些信息，运行效果如图 3-2 所示。

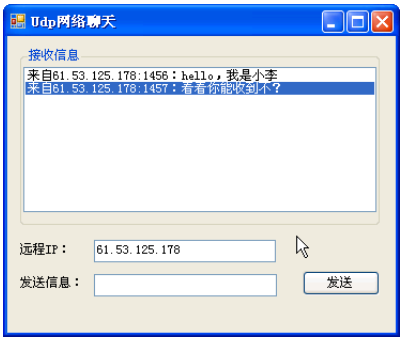


图3-2　例3-1的运行效果

3.3　利用 UDP 协议进行广播和组播

在上一节中，发送数据使用的是一对一（单播）的通信模式，即只将数据发送到某一台远程主机。UDP 协议的另外一个重要用途是可以通过广播和组播实现一对多的通信模式，即可以把数据发送到一组远程主机中。

3.3.1　通过 Internet 实现群发功能

通过一对多的方式，可以将数据发送到多台远程主机中，从而完成发送网络会议通知、广告、网络信息公告等群发功能。通过 Internet 实现群发功能的形式有两种，一种是利用广播向子网中的所有客户发送消息，比如各类通知、单位公告、集体活动日程安排等；另外一种是利用组播向 Internet 网上不同的子网发送消息，比如集团向其所属的公司或用户子网发布信息公告等。

不论采用哪种形式，发送和接收数据的程序编写的思路都是一样的，区别仅仅是一些实现细节有所不同。

1. 利用广播实现群发功能

所谓广播，就是指同时向子网中的多台计算机发送消息，并且所有子网中的计算机都可以接收到发送方发来的消息。每个广播消息包含一个特殊的 IP 地址。广播消息地址分为两种类型：本地广播和全球广播。

通过本地广播向子网中的所有计算机发送广播消息时，其他网络不会受到本地广播的影响。在前面的学习中，我们已经知道了 IP 地址分为两部分，网络地址和主机地址，标准网络地址部分组成了本地网络地址的第一部分，字节地址中全部为 1 的部分用于主机地址部分（即十进制的 255）。例如，对于 B 类网络 192.168.0.0，使用子网掩码 255.255.0.0，则本地广播地址是 192.168.255.255，用二进制表示为 11000000、10101000、11111111、11111111。其中前两个字节为网络地址，后两个字节为主机地址。

仍以 192.168.0.0 为例，如果子网掩码为 255.255.255.0，则本地广播地址是 192.168.0.255。192.168.0 为网络地址，255 代表 192.168.0 子网中的主机地址。

全球广播使用四个字节所有位全为 1 的 IP 地址，即点分十进制的 255.255.255.255，这个特定的广播地址表明数据报的目的地是网络上的所有设备。但是由于路由器会自动过滤掉全球广播，所以使用这个地址没有实际意义。

下面的例子分别说明了发送和接收广播数据报的方法。

【例 3-2】编写一个 Windows 应用程序，向子网发送广播信息，同时接收子网中的任意主机发送的广播信息。

(1) 创建一个名为 BroadcastExample 的 Windows 应用程序，修改 Form1.cs 为 FormBroadcast.cs，设计界面如图 3-3 所示。

图3-3　例3-2的设计界面

(2) 添加对应的代码，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

```
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
namespace BroadcastExample
{
    public partial class FormBroacast : Form
    {
        delegate void AppendStringCallback(string text);
        AppendStringCallback appendStringCallback;
        //使用的接收端口号
        private int port = 8001;
        private UdpClient udpClient;
        public FormBroacast()
        {
            InitializeComponent();
            appendStringCallback = new AppendStringCallback(AppendString);
        }
        private void AppendString(string text)
        {
            if (richTextBox1.InvokeRequired == true)
            {
                this.Invoke(appendStringCallback, text);
            }
            else
            {
                richTextBox1.AppendText(text + "\r\n");
            }
        }
        /// <summary>
        ///  在后台运行的接收线程
        /// </summary>
        private void ReceiveData()
        {
            //在本机指定的端口接收
            udpClient = new UdpClient(port);
            IPEndPoint remote = null;
            //接收从远程主机发送过来的信息;
            while (true)
            {
                try
                {
                    //关闭 udpClient 时此句会产生异常
                    byte[] bytes = udpClient.Receive(ref remote);
                    string str = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
                    AppendString(string.Format("来自{0}：{1}", remote, str));
                }
                catch
                {
                    //退出循环，结束线程
                    break;
```

```
                }
            }
        }
        private void buttonSend_Click(object sender, EventArgs e)
        {
            UdpClient myUdpClient = new UdpClient();
            try
            {
                //让其自动提供子网中的 IP 广播地址
                IPEndPoint iep = new IPEndPoint(IPAddress.Broadcast, 8001);
                //允许发送和接收广播数据报
                myUdpClient.EnableBroadcast = true;
                //将发送内容转换为字节数组
                byte[] bytes = System.Text.Encoding.UTF8.GetBytes(textBox1.Text);
                //向子网发送信息
                myUdpClient.Send(bytes, bytes.Length, iep);
                textBox1.Clear();
                textBox1.Focus();
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message, "发送失败");
            }
            finally
            {
                myUdpClient.Close();
            }
        }
        private void FormBroacast_Load(object sender, EventArgs e)
        {
            Thread myThread = new Thread(new ThreadStart(ReceiveData));
            //将线程设为后台运行
            myThread.IsBackground = true;
            myThread.Start();
        }
        private void FormBroacast_FormClosing(object sender, FormClosingEventArgs e)
        {
            udpClient.Close();
        }
    }
}
```

(3) 按<F5>键编译并执行，发送一些信息，观察结果。

2. 利用组播实现群发功能

广播的通信模式虽然能够实现一对多的通信需要，但是，由于广播是向子网中的所有计算机用户发送消息，没有目的性，不但增加了网络传输负担，而且资源消耗较高。组播的出现，较好地解决了这个问题。组播也叫多路广播。所谓组播是将消息从一台计算机发送到本网或全网内选择的计算机子集上，即发送到那些加入指定组播组的计算机上。组播组是开放的，每台计算机都可以通过程序随时加入到组播组中，也可以随时离开。

组播组是分享一个组播地址的一组设备。与 IP 广播类似，IP 组播使用特殊的 IP 地址范围

来表示不同的组播组。组播地址是范围在 224.0.0.0 到 239.255.255.255 的 D 类 IP 地址。任何发送到组播地址的消息都会被发送到组内的所有成员设备上。组可以是永久的，也可以是临时的。大多数组播组是临时的，仅在有成员的时候才存在。用户创建一个新的组播组时只需从地址范围内选出一个地址，然后为这个地址构造一个对象，就可以开始发送消息了。

使用组播时，应注意的是 TTL(生存周期 Time To Live)值的设置。TTL 值是允许路由器转发的最大数目，当达到这个最大值时，数据包就会被丢弃。如果使用默认值(默认值为 1)，则只能在子网中发送。可以通过 UdpClient 对象的 Ttl 属性直接设置 TTL 值，例如：

```
UdpClient myUdpClient = new UdpClient();
myUdpClient.Ttl = 50;
```

该语句设置 TTL 值为 50，即最多允许 50 次路由器转发。

在 UdpClient 类中，使用 JoinMulticastGroup 方法将 UdpClient 对象和 TTL 一起加入组播组，使用 DropMulticastGroup 退出组播组。例如：

```
//创建 UdpClient 的实例并设置使用的本地端口号
UdpClient udpClient=new UdpClient(8001);
udpClient.JoinMulticastGroup(IPAddress.Parse("224.100.0.1"));
```

或者：

```
UdpClient udpClient=new UdpClient(8001);
udpClient.JoinMulticastGroup(IPAddress.Parse("224.100.0.1"), 50);
```

其中 50 为 TTL 值。

【例 3-3】编写一个 Windows 应用程序，利用组播技术向子网发送组播信息，同时接收组播的信息。

(1) 创建一个名为 MulticastExample 的 Windows 应用程序，修改 Form1.cs 为 FormMulticast.cs，设计界面如图 3-4 所示。

图3-4　例3-3的设计界面

(2) 添加对应的代码，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```

```
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
namespace MulticastExample
{
    public partial class FormMulticast : Form
    {
        delegate void AppendStringCallback(string text);
        AppendStringCallback appendStringCallback;
        //使用的接收端口号
        private int port = 8001;
        private UdpClient udpClient;
        public FormMulticast()
        {
            InitializeComponent();
            appendStringCallback = new AppendStringCallback(AppendString);
        }
        private void AppendString(string text)
        {
            if (richTextBox1.InvokeRequired == true)
            {
                this.Invoke(appendStringCallback, text);
            }
            else
            {
                richTextBox1.AppendText(text + "\r\n");
            }
        }
        private void ReceiveData()
        {
            //在本机指定的端口接收
            udpClient = new UdpClient(port);
            //必须使用组播地址范围内的地址
            udpClient.JoinMulticastGroup(IPAddress.Parse("224.0.0.1"));
            udpClient.Ttl = 50;
            IPEndPoint remote = null;
            //接收从远程主机发送过来的信息;
            while (true)
            {
                try
                {
                    //关闭 udpClient 时此句会产生异常
                    byte[] bytes = udpClient.Receive(ref remote);
                    string str = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
                    AppendString(string.Format("来自{0}：{1}", remote, str));
                }
                catch
                {
```

```
                    //退出循环，结束线程
                    break;
                }
            }
        }
        private void buttonSend_Click(object sender, EventArgs e)
        {
            UdpClient myUdpClient = new UdpClient();
            //允许发送和接收广播数据报
            myUdpClient.EnableBroadcast = true;
            //必须使用组播地址范围内的地址
            IPEndPoint iep = new IPEndPoint(IPAddress.Parse("224.0.0.1"), 8001);
            //将发送内容转换为字节数组
            byte[] bytes = System.Text.Encoding.UTF8.GetBytes(textBox1.Text);
            try
            {
                //向子网发送信息
                myUdpClient.Send(bytes, bytes.Length, iep);
                textBox1.Clear();
                textBox1.Focus();
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message, "发送失败");
            }
            finally
            {
                myUdpClient.Close();
            }
        }
        private void FormMulticast_Load(object sender, EventArgs e)
        {
            Thread myThread = new Thread(new ThreadStart(ReceiveData));
            myThread.Start();
        }
        private void FormMulticast_FormClosing(object sender, FormClosingEventArgs e)
        {
            udpClient.Close();
        }
    }
}
```

(3) 按<F5>键编译并执行，发送一些信息，观察结果。

3.3.2 在 Internet 上举行网络会议讨论

网络会议是基于局域网或 Internet 网的实时的、交互的计算机应用系统，由于它具有低成本、少失误、可扩展的优点，已经被广泛应用于各个领域中。本节通过网络会议讨论说明组播技术的进一步应用。在这个例子中，网络中的任何人都可以加入讨论，所有发言者发送的信息都发送到会议讨论组，讨论组的每个人都可以看到任何人发送的信息。

【例 3-4】编写一个 Windows 应用程序，利用组播技术进行网络会议讨论。

(1) 创建一个名为“NetMeetingExample”的 Windows 应用程序，修改 Form1.cs 为 FormMeeting.cs，设计界面如图 3-5 所示。

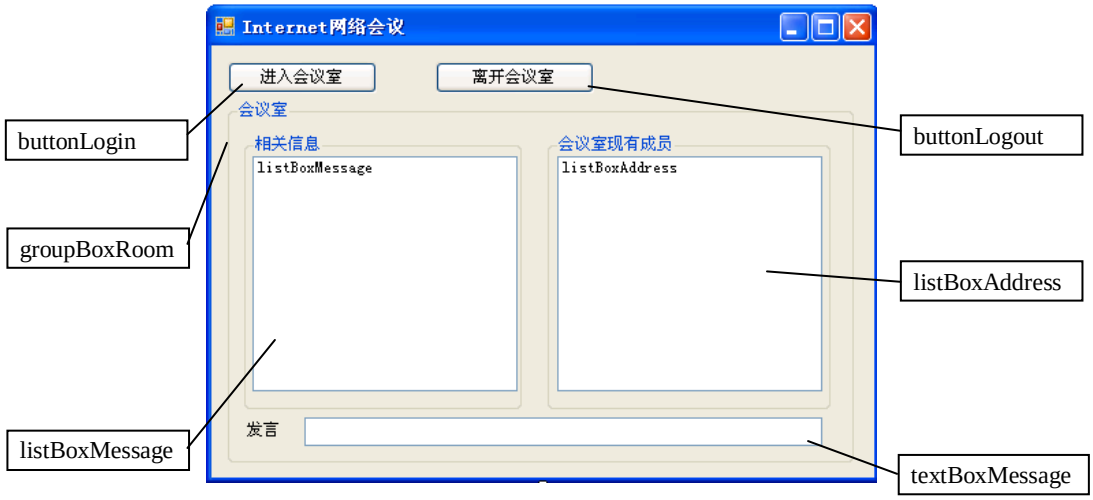


图3-5　例3-4的设计界面

(2) 添加对应的代码，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
namespace NetMeetingExample
{
    public partial class FormMeeting : Form
    {
        private enum ListBoxOperation { AddItem, RemoveItem };
        private delegate void SetListBoxItemCallback(
            ListBox listbox, string text, ListBoxOperation operation);
        SetListBoxItemCallback listBoxCallback;
        //使用的 IP 地址
        private IPAddress ip = IPAddress.Parse("224.100.0.1");
        //使用的接收端口号
        private int port = 8001;
        private UdpClient udpClient;
        public FormMeeting()
        {
            InitializeComponent();
            listBoxCallback = new SetListBoxItemCallback(SetListBoxItem);
        }
        private void SetListBoxItem(ListBox listbox, string text, ListBoxOperation operation)
```

```
{
    if (listbox.InvokeRequired == true)
    {
        this.Invoke(listBoxCallback, listbox, text, operation);
    }
    else
    {
        if (operation == ListBoxOperation.AddItem)
        {
            if (listbox == listBoxAddress)
            {
                if (listbox.Items.Contains(text) == false)
                {
                    listbox.Items.Add(text);
                }
            }
            else
            {
                listbox.Items.Add(text);
            }
            listbox.SelectedIndex = listbox.Items.Count - 1;
            listbox.ClearSelected();
        }
        else if (operation == ListBoxOperation.RemoveItem)
        {
            listbox.Items.Remove(text);
        }
    }
}
private void SendToAll(string sendString)
{
    UdpClient myUdpClient = new UdpClient();
    //允许发送和接收广播数据报
    myUdpClient.EnableBroadcast = true;
    //必须使用组播地址范围内的地址
    IPEndPoint iep = new IPEndPoint(ip, port);
    //将发送内容转换为字节数组
    byte[] bytes = System.Text.Encoding.UTF8.GetBytes(sendString);
    try
    {
        //向子网发送信息
        myUdpClient.Send(bytes, bytes.Length, iep);
    }
    catch (Exception err)
    {
        MessageBox.Show(err.Message, "发送失败");
    }
    finally
    {
        myUdpClient.Close();
    }
```

```
}
private void FormMeeting_Load(object sender, EventArgs e)
{
    listBoxMessage.HorizontalScrollbar = true;
    buttonLogin.Enabled = true;
    buttonLogout.Enabled = false;
    groupBoxRoom.Enabled = false;
}
/// <summary>
///  接收线程
/// </summary>
private void ReceiveMessage()
{
    udpClient = new UdpClient(port);
    //必须使用组播地址范围内的地址
    udpClient.JoinMulticastGroup(ip);
    udpClient.Ttl = 50;
    IPEndPoint remote = null;
    while (true)
    {
        try
        {
            //关闭 udpClient 时此句会产生异常
            byte[] bytes = udpClient.Receive(ref remote);
            string str = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
            string[] splitString = str.Split(',');
            int s = splitString[0].Length;
            switch (splitString[0])
            {
                case "Login":   //进入会议室
                    SetListBoxItem(listBoxMessage,
                        string.Format("[{0}]进入。",
                        remote.Address), ListBoxOperation.AddItem);
                    string userListString = "List," + remote.Address.ToString();
                    for (int i = 0; i < listBoxAddress.Items.Count; i++)
                    {
                        userListString += "," + listBoxAddress.Items[i].ToString();
                    }
                    SendToAll(userListString);
                    break;
                case "List": //参加会议人员名单
                    for (int i = 1; i < splitString.Length; i++)
                    {
                        SetListBoxItem(listBoxAddress,
                            splitString[i], ListBoxOperation.AddItem);
                    }
                    break;
                case "Message":   //发言内容
                    SetListBoxItem(listBoxMessage,
                        string.Format("[{0}]说：{1}", remote.Address, str.Substring(8)),
                        ListBoxOperation.AddItem);
```

```
                    break;
                case "Logout": //退出会议室
                    SetListBoxItem(listBoxMessage,
                        string.Format("[{0}]退出。", remote.Address),
                        ListBoxOperation.AddItem);
                    SetListBoxItem(listBoxAddress,
                        remote.Address.ToString(), ListBoxOperation.RemoveItem);
                    break;
            }
        }
        catch
        {
            //退出循环，结束线程
            break;
        }
    }
}
private void textBoxMessage_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == (char)Keys.Return)
    {
        if (textBoxMessage.Text.Trim().Length > 0)
        {
            SendToAll("Message," + textBoxMessage.Text);
            textBoxMessage.Text = "";
        }
    }
}
//窗体已关闭并指定关闭原因前触发的事件
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    if (buttonLogout.Enabled == true)
    {
        MessageBox.Show("请先离开会议室，然后再退出！", "",
            MessageBoxButtons.OK, MessageBoxIcon.Warning);
        //不关闭窗体
        e.Cancel = true;
    }
}
//单击进入会议室按钮触发的事件
private void buttonLogin_Click(object sender, EventArgs e)
{
    Cursor.Current = Cursors.WaitCursor;
    Thread myThread = new Thread(new ThreadStart(ReceiveMessage));
    myThread.Start();
    //等待接收线程准备完毕
    Thread.Sleep(1000);
    SendToAll("Login");
    buttonLogin.Enabled = false;
    buttonLogout.Enabled = true;
    groupBoxRoom.Enabled = true;
```

```
            Cursor.Current = Cursors.Default;
        }
        //单击退出会议室按钮触发的事件
        private void buttonLogout_Click(object sender, EventArgs e)
        {
            Cursor.Current = Cursors.WaitCursor;
            SendToAll("Logout");
            //等待接收线程处理完毕
            Thread.Sleep(1000);
            //结束接收线程
            udpClient.Close();
            buttonLogin.Enabled = true;
            buttonLogout.Enabled = false;
            groupBoxRoom.Enabled = false;
            Cursor.Current = Cursors.Default;
        }
    }
}
```

(3) 同时在几台计算机上运行该程序，观察运行效果。

3.4 习题 3

1. UDP 协议和 TCP 协议的主要区别有哪些？
2. 什么是广播、组播？两者有什么区别？简要回答 UDP 协议是如何实现广播和组播的？

第 4 章　P2P 应用编程

近年来，P2P 的发展非常迅速。采用 P2P 方式实现的软件也越来越多，涉及到通信、互动游戏、媒体播放等多种网络应用。目前人们普遍认为，P2P 在加强网络上人的交流、文件交换、深度搜索、分布计算以及协同工作等方面大有前途。

4.1　P2P 基础知识

P2P 是 Peer-to-Peer 的缩写，也叫对等互联或点对点技术。与 TCP、UDP 不同，P2P 并不是一种新的协议，而是利用现有网络协议实现网络数据或资源信息共享的技术，它使用的不一定是 TCP 协议，也可能是 UDP 协议或者其他协议。使用 P2P 技术，可以让一台计算机与另一台计算机直接交换数据，通过 Internet 直接使用对方的文件，而不必像传统 C/S 模式全部通过服务器处理。

P2P 的特点主要有：

1．对等模式

与传统的服务器/客户端模式不同，使用 P2P 技术实现的每个计算机节点既是客户端，也是服务器，其功能的提供是对等的，人们可以直接连接到安装了相同的 P2P 软件的其他用户的计算机上进行数据交互，而不需要提供专门的服务器。

2．分布式网络数据存储结构

相对于目前流行的 C/S、B/S 的“集中式”网络数据存储结构，P2P 最大的特点在于“分散”。网络中所有的计算、存储和网络连接能力都分布在非集中式网络的“对等伙伴”上。例如，以前客户端下载文件都是从服务器上下载，而 P2P 技术则改变了以服务器为中心的状态，每个节点可以各下载一部分，然后互相从对方或者其他节点下载。采用这种方式，大量用户同时下载不但不会形成服务器网络带宽瓶颈，造成网络堵塞，反而加快了下载速度。这种方法被认为最能发挥互联网的优势。

分布式计算是继“服务器/客户端”结构后新兴的网络应用模式。在传统的“服务器/客户端”应用系统中，客户端与服务器有明确的分界，常常发生客户端能力过剩，服务器能力不足或网路堵塞的现象。P2P 系统中的使用者能同时扮演客户端和服务器的多重角色，使两个使用者之间能不通过服务器而直接进行信息分享，以构建具有自主、开放、异质、延展等特性的分布式网际网络应用系统。

从计算模式上看，P2P 更加符合分布式计算的理念。其所倡导的计算能力边缘化、计算资源共享等思想，刚好与网格技术不谋而合。通过 P2P 技术，人们可以在不改变原有基础设施的基础上，实现对底层计算资源的控制和调用。

P2P 的设计模式可以分为两大类：

一种是单纯型 P2P 架构，没有专用的服务器；另一种是混合型 P2P 架构，即单纯型和专用服务器相结合的架构。

单纯型 P2P 架构没有中央服务器，各个节点之间直接交互信息。这种方式的优点是使用方便，任何一台计算机只要安装了同一个 P2P 应用软件，就可以和其他安装这个软件的计算机直接通信。而正是由于没有中央服务器参与协调，这种方式的使用范围就比较有限了。原因很简单，一台计算机要和另一台计算机连接，必须要知道对方的 IP 地址和监听端口，否则就无法向对方发送信息。而这个工作只能通过人来处理，即通过软件提供的手工操作功能将对方的 IP 地址和端口加入到搜索范围内，无法利用计算机自动搜索扩展。

从原理上说，互联网最基本的协议 TCP/IP 并没有客户机和服务器的概念，所有的设备都是通讯的平等的一端。起初，所有的互联网上的系统都同时具有服务器和客户机的功能。当然，后来发展的软件的确采用了客户机/服务器的结构：浏览器和 Web 服务器，邮件客户端和邮件服务器。但是，对于服务器来说，它们之间仍然是对等联网的。以 E-mail 为例，互联网上并没有一个巨大的、唯一的邮件服务器来处理所有的 E-mail，而是对等联网的邮件服务器相互协作把 E-mail 传送到相应的服务器上去。

混合型 P2P 架构则是将 P2P 和客户/服务器模式相结合，此时的中央服务器仅起到促成各节点协调和扩展的功能。安装了 P2P 软件的各个计算机开始全部和索引服务器连接，以便告知自己监听的 IP 地址和端口，然后再通过索引服务器告知其他与自己连接的计算机，每一台计算机的连接和断开连接都通过服务器通知网络上有联系的计算机。这样就减轻了每台计算机搜索其他计算机的负担，扩展也比较方便，而真正的信息交互则仍然通过点对点直接完成。但带来的缺点就是服务器必须正常工作才能搜索到其他计算机。

1999 年，Napster 首先发掘了 P2P 在文件共享方面的潜力，推出面向全球互联网用户的 MP3 自由下载服务。仅 1 年间，其注册会员就达到 300 万。可以说，Napster 唤醒了深藏在互联网背后的对等联网。实际上，文件共享功能在局域网中是再平常不过的事情。但是 Napster 的成功促使人们认识到把这种“对等联网”拓展到整个互联网范围的可能性。

事实上，网络上现有的许多服务都可以归入 P2P 的行列。即时通信系统例如 ICQ、Yahoo Pager、微软的 MSN Messenger 以及国内的 OICQ、POPO 等都是最流行的 P2P 应用。

目前比较流行的下载类 P2P 软件的典型应属 BT(BitTorrent)，它采用一种结构化网络结构，利用分布式哈希表(DHT)技术，使每个独立节点都不需要维护整个网络信息，只在节点中存储其临近的后继节点信息，就可以有效地找到其他目标节点。

近年来悄然兴起的互联网视频直播软件也是 P2P 的应用之一。这类软件使用网状结构，支持多种格式的流媒体文件，节点间动态查找就近连接。

其工作原理是创建系统内部的虚拟组播技术，利用用户的剩余处理能力和带宽为下游客户提供客户端代理，每一个授权用户都可以将收到的媒体流进行复制或分裂成多个流发送到其他用户，网络中每个用户都可以是媒体的使用者同时也是媒体的分发者。用虚拟组播技术媒体流可以不断的被增加和复制，并一个接一个地传递到网络的其他用户，从而在网络边界形成一个媒体流的虚拟交叉网络。当用户需要播放某一频道节目时，组播系统就会将它加入到相应组播组中，使用户迅速得到服务。

利用 P2P 的虚拟组播技术实现了在线用户越多，视频播放越流畅的特性。由此可见，现

有 Internet 网络架构下点播式的视频业务，使用 P2P 技术非常合适。

随着计算机技术的发展和宽带网的进一步普及，P2P 应用的优势越来越明显。它可以充分利用互联网的边缘资源，即用户的计算能力、存储能力和带宽，甚至每个计算机硬盘的内容。由于 P2P 网络的非中心化和自发组织的体系结构特性，使其具有非常好的健壮性。也正是 P2P 网络的灵活性和易操作性提升了互联网用户的共享和参与热情，从而带动互联网的进一步发展。

4.2 P2P 应用举例

本节首先通过单纯型 P2P 架构实现一个简单的聊天程序，然后介绍通过 P2P 技术设计实际应用程序的思路。

利用这个程序，可以在网络中向已知计算机发送、接收字符串信息。程序同时既作为服务器端，又作为客户端。任何一台计算机只要安装了这个程序，就可以通过 Internet 与加入到好友列表中的任何一个好友即时聊天。

【例 4-1】使用 P2P 技术设计一个简易聊天程序，要求不使用专用的主服务器，只要将好友添加到好友列表中，就能检测到好友是否在线，并相互发送聊天信息。

(1) 创建一个名为 P2PExample 的 Windows 应用程序，将 Form1.cs 换名为 FormP2P.cs，然后在该设计窗体内设计如图 4-1 所示的界面。

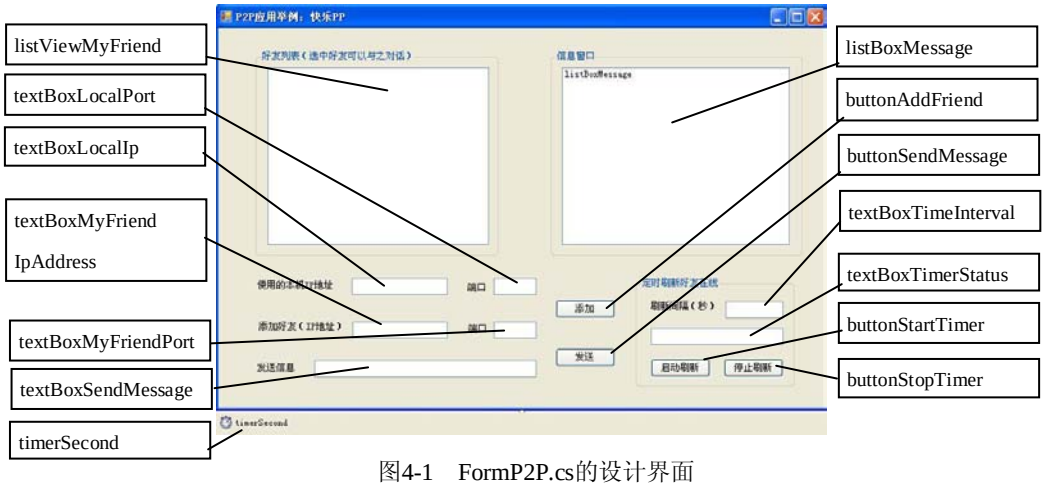


图4-1 FormP2P.cs的设计界面

这是完成聊天功能的主界面。由于不存在主服务器，所以添加好友时，需要提供好友所用计算机的 IP 地址和端口号。为了方便起见，程序中自动生成并显示出本机当前所用的 IP 地址和端口，如果程序运行时 IP 地址没有显示出来，就无法和别的计算机连接。

(2) 添加命名空间引用：

```
using System.IO;
using System.Net.Sockets;
using System.Net;
using System.Threading;
```

(3) 在构造函数上方添加字段声明，并在构造函数中添加代码：

```
private Thread myThread;
private TcpListener tcpListener;
```

```
private IPAddress myIPAddress;
private int myPort;
private System.Diagnostics.Stopwatch secondWatch;
public FormP2P()
{
    InitializeComponent();
    secondWatch = new System.Diagnostics.Stopwatch();
    ColumnHeader ipColumn = new ColumnHeader();
    ipColumn.Text = "IP 地址";
    ipColumn.Width = 136;
    ColumnHeader portColumn = new ColumnHeader();
    portColumn.Text = "端口号";
    ColumnHeader onlineColumn = new ColumnHeader();
    onlineColumn.Text = "是否在线";
    onlineColumn.Width = 71;
    listViewMyFriend.View = View.Details;
    listViewMyFriend.Columns.AddRange(
        new ColumnHeader[] { ipColumn, portColumn, onlineColumn });
}
```

(4) 添加 Load 事件代码：

```
private void FormP2P_Load(object sender, EventArgs e)
{
    //启动秒表
    secondWatch.Start();
    timerSecond.Enabled = true;
    buttonStartTimer.Enabled = false;
    buttonStopTimer.Enabled = true;
    //使用代理指定在线程上执行的方法
    ThreadStart myThreadStartDelegate = new ThreadStart(Listening);
    //创建一个用于监听的线程对象,通过代理执行线程中的方法
    myThread = new Thread(myThreadStartDelegate);
    //启动线程
    myThread.Start();
}
```

(5) 添加线程执行的方法：

```
//该方法是通过代理调用执行的
private void Listening()
{
    Socket socket = null;
    //获取本机第一个可用 IP 地址
    myIPAddress = (IPAddress)Dns.GetHostAddresses(Dns.GetHostName()).GetValue(0);
    Random r = new Random();
    while (true)
    {
        try
        {
            //随机产生一个 0-65535 之间的端口号
            myPort = r.Next(65535);
            //创建 TcpListener 对象，在本机的 IP 和 port 端口监听连接到该 IP 和端口的请求
            tcpListener = new TcpListener(myIPAddress, myPort);
            tcpListener.Start();
```

```
            //显示 IP 地址和端口
            ShowLocalIpAndPort();
            //在信息窗口中显示成功信息
            ShowMyMessage(string.Format("尝试用端口{0}监听成功", myPort));
            ShowMyMessage(string.Format(
                "<message>[{0}]{1:h 点 m 分 s 秒}开始在{2}端口监听与本机的连接",
                myIPAddress, DateTime.Now, myPort));
            break;
        }
        catch
        {
            //继续 while 循环，以便随机找下一个可用端口号，同时显示失败信息
            ShowMyMessage(string.Format("尝试用端口{0}监听失败", myPort));
        }
    }
    while (true)
    {
        try
        {
            //使用阻塞方式接收客户端连接,根据连接信息创建 TcpClient 对象
            //注意：AcceptSocket 接收到新的连接请求才会继续执行其后的语句
            socket = tcpListener.AcceptSocket();
            //如果往下执行，说明已经根据客户端连接请求创建了套接字
            //使用创建的套接字接收客户端发送的信息
            NetworkStream stream = new NetworkStream(socket);
            StreamReader sr = new StreamReader(stream);
            string receiveMessage = sr.ReadLine();
            int i1 = receiveMessage.IndexOf(",");
            int i2 = receiveMessage.IndexOf(",", i1 + 1);
            int i3 = receiveMessage.IndexOf(",", i2 + 1);
            string ipString = receiveMessage.Substring(0, i1);
            string portString = receiveMessage.Substring(i1 + 1, i2 - i1 - 1);
            string messageTypeString = receiveMessage.Substring(i2 + 1, i3 - i2 - 1);
            string messageString = receiveMessage.Substring(i3 + 1);
            ShowMyMessage(ipString, portString, messageTypeString, messageString);
        }
        catch
        {
            //通过停止 TcpListener 使 AcceptSocket()出现异常
            //在异常处理中关闭套接字并终止线程
            if (socket != null)
            {
                if (socket.Connected)
                {
                    socket.Shutdown(SocketShutdown.Receive);
                }
                socket.Close();
            }
            myThread.Abort();
        }
    }
```

```
}
```

在第一个 while 循环中，首先随机生成一个端口号，然后使用本机第一个 IP 地址和产生的端口进行监听，如果不出现异常，说明该 IP 地址和端口号可用，退出 while 循环；否则继续随机产生下一个端口，直到找到可用的端口为止。

在第二个 while 循环中，使用 AcceptSocket 方法接收到该端口的连接请求，在接收到连接请求之前，该方法一直处于阻塞方式，一旦接收到新的连接请求，就创建一个对应的套接字对象，然后就可以利用这个套接字对象创建网络流对象，再利用 StreamReader 从网络流中一直读取字符，直到遇到回车换行为止。

注意，这里使用回车换行作为每条信息之间的分隔符。在介绍 TCP 协议时，读者已经知道 TCP 协议是没有消息边界的，如果不考虑这个问题，就无法保证发送的每条信息和接收的每条信息一一对应，例如可能会出现发送的几条信息一块接收的情况。

考虑下面几条语句：

```
byte[] buffer = new byte[socket.ReceiveBufferSize];
int i = socket.Receive(buffer, buffer.Length, SocketFlags.None);
string receiveMessage = System.Text.Encoding.UTF8.GetString(buffer, 0, i);
```

如果不仔细考虑，读者可能会认为这几条语句和使用 StreamReader 对象从网络流读取方式完成的功能相同，其实这样使用会出现问题的。由于边界问题和网络中传输的影响，使用 socket.Receive 方法接收到的不一定刚好是发送的一条信息，可能是一部分，也可能是几条信息都在缓冲区内，而这段代码中只接收了一次，自然无法保证和发送的一条信息完全对应。

另外，如果在一台机器上调试，由于没有网络传输的影响，自然也就无法发现这段代码中存在的问题。即使在一个局域网中调试，网络传输的影响很小，也很难发现边界问题。

而使用 StreamReader 对象的 ReadLine 方法，则可以一直读取直到遇到回车换行为止，从而保证与发送的以回车换行结尾的每条信息对应。

接收到一条信息以后，根据发送时在字符串中插入的逗号分隔符，将其分开，分别得到对方的 IP 地址、端口号、信息类型以及信息。然后调用 SendMyMessage 方法分别处理。

还有一点要注意，不能在第二个循环中试图用一个布尔型变量作为是否退出循环的判断标准，这是因为程序执行到 AcceptSocket 方法时会处于阻塞方式，无法保证及时响应该布尔值的变化，也就失去了判断的意义。

(6) 添加代理及相应的方法：

```
delegate void ShowMessageDelegate1(string str);
private void ShowMyMessage(string str)
{
    //比较调用的线程和创建的线程是否同一个线程
    //如果不是，结果为 true
    if (this.listBoxMessage.InvokeRequired == true)
    {
        //如果结果为 true，则自动通过代理执行 else 中语句的功能（注意：是 else，不是 if）
        //这里只需要传入参数 str 即可
        //但是执行的功能会始终与 else 中的指定的功能相同，区别仅是通过代理完成
        ShowMessageDelegate1 messageDelegate = new ShowMessageDelegate1(ShowMyMessage);
        this.Invoke(messageDelegate, new object[] { str });
    }
    else
    {
```

```
            //在这里指定如果是同一个线程需要完成什么功能
            //如果是不同线程，系统会自动通过代理实现这里指定的功能
            listBoxMessage.Items.Add(str);
        }
}

delegate void ShowMessageDelegate2(string ipString, string portString, string messageTypeString, string
messageString);
private void ShowMyMessage(string ipString, string portString, string messageTypeString, string
messageString)
{
    if (this.listBoxMessage.InvokeRequired == true)
    {
        ShowMessageDelegate2 messageDelegate = new ShowMessageDelegate2(ShowMyMessage);
        this.Invoke(messageDelegate, new object[] { ipString, portString, messageTypeString,
messageString });
    }
    else
    {
        //messageType 的含义为：
        //<connect>前缀：第 i 次连接
        //<check>前缀：检查接收者是否在线
        //<message>前缀：聊天信息
        int myfriendIndex = CheckMyFriend(ipString);
        switch (messageTypeString)
        {
            case "connect":
                if (myfriendIndex == -1)
                {
                    ListViewItem myFriendItem = new ListViewItem(
                        new string[] { ipString, portString, "是" });
                    listViewMyFriend.Items.Add(myFriendItem);
                }
                listBoxMessage.Items.Add(string.Format("[{0}:{1}]说：{2}",
                    ipString, portString, messageString));
                break;
            case "check":
                if (myfriendIndex == -1)
                {
                    ListViewItem myFriendItem = new ListViewItem(
                        new string[] { ipString, portString, "是" });
                    listViewMyFriend.Items.Add(myFriendItem);
                }
                //不需要显示
                break;
            case "message":
                listBoxMessage.Items.Add(string.Format("[{0}:{1}]说：{2}",
                    ipString, portString, messageString));
                break;
            default:
                listBoxMessage.Items.Add(string.Format("什么意思呀：“{0}”", messageString));
```

```
                    break;
                }
            }
        }

        delegate void ShowIpAndPortDelegate();
        private void ShowLocalIpAndPort()
        {
            if (this.listBoxMessage.InvokeRequired)
            {
                ShowIpAndPortDelegate messageDelegate =
                    new ShowIpAndPortDelegate(ShowLocalIpAndPort);
                this.Invoke(messageDelegate);
            }
            else
            {
                textBoxLocalIp.Text = myIPAddress.ToString();
                textBoxLocalPort.Text = myPort.ToString();
            }
        }
```

使用代理的目的是为了解决一个线程无法在托管模式下直接调用另一个线程的控件问题。当然，如果通过非托管模式，也可以直接调用，但可能会引起死锁等问题。

(7) 添加代码，检查好友是否在好友列表中：

```
        private int CheckMyFriend(string remoteIpString)
        {
            //在 listViewMyFriend 中检查指定的 ip 是否存在
            ListViewItem item = listViewMyFriend.FindItemWithText(remoteIpString);
            if (item == null)
            {
                return -1;
            }
            else
            {
                return item.Index;
            }
        }
```

代码中的 FindItemWithText 方法检查 ListView 中的每一项中是否包含指定的字符串，如果是，则返回该项，否则返回 null。由于好友列表中的其他列不可能有和指定的 IP 地址相同的内容，所以这里也就相当于检查第一列是否有指定的 IP。

(8) 添加 SendMessage 方法：

```
        private void SendMessage(string remoteIpString, string remotePortString, string strType, string str)
        {
            IPAddress remoteIP = IPAddress.Parse(remoteIpString);
            int remotePort = int.Parse(remotePortString);
            NetworkStream networkstream = null;
            TcpClient tcpclient = null;
            try
            {
                tcpclient = new TcpClient(remoteIpString, remotePort);
```

```
            //得到一个用于发送和接收数据的网络流
            networkstream = tcpclient.GetStream();
            //使用默认编码和缓冲区大小初始化 StreamWriter 对象
            StreamWriter streamWriter = new StreamWriter(networkstream);
            //使用回车换行作为每次发送的分隔符
            streamWriter.WriteLine(
                string.Format("{0},{1},{2},{3}", myIPAddress, myPort, strType, str));
            //将缓冲区内容全部发送出去
            streamWriter.Flush();
            if (strType != "check")
            {
                listBoxMessage.Items.Add(
                    string.Format("向[{0}:{1}]发送成功，信息：{2}",
                    remoteIpString, remotePortString, str));
            }
        }
        catch (Exception err)
        {
            int i = CheckMyFriend(remoteIP.ToString());
            if (i != -1)
            {
                listViewMyFriend.Items[i].SubItems[2].Text = "否";
            }
            if (strType != "check")
            {
                listBoxMessage.Items.Add(
                    string.Format("向[{0}:{1}]发送信息失败，原因：{2}",
                    remoteIpString, remotePortString, err.Message));
            }
        }
        finally
        {
            if (networkstream != null)
            {
                networkstream.Close();
            }
            if (tcpclient != null)
            {
                tcpclient.Close();
            }
        }
    }
```

这部分代码完成发送字符串功能，代码中首先根据目标 IP 和端口号创建一个 TcpClient 对象，然后利用该对象的 GetStream 方法创建网络流。使用 StreamWriter 对象发送的每条信息均以回车换行结束，便于接收方区分是否到了一条信息的结尾。

如果发送的信息类型为 check，说明仅仅是为了检查对方是否在线，如果对方在线，发送中就不会出现异常，否则会出现异常，表明对方已经断开连接。

对于发送信息为文件的情况，解决边界问题的最好办法是先发送该文件的容量，即该文件占用的字节数，然后再发送文件内容。这样，接收方就可以根据文件大小确定什么时候接收完

毕。

(9) 双击【添加】按钮，添加 Click 事件代码：

```
private void buttonAddFriend_Click(object sender, EventArgs e)
{
    IPAddress myFriendIpAddress;
    if (IPAddress.TryParse(textBoxMyFriendIpAddress.Text, out myFriendIpAddress) == false)
    {
        MessageBox.Show("IP 地址格式不正确!");
        return;
    }
    int myFriendPort;
    if (int.TryParse(textBoxMyFriendPort.Text, out myFriendPort) == false)
    {
        MessageBox.Show("端口号格式不正确!");
        return;
    }
    else
    {
        if (myFriendPort < 1000 || myFriendPort > 65535)
        {
            MessageBox.Show("端口号范围不正确!,必须在 1000-65535 之间");
            return;
        }
    }
    int i = CheckMyFriend(textBoxMyFriendIpAddress.Text);
    if (i != -1)
    {
        MessageBox.Show("该好友已经在列表中");
    }
    else
    {
        ListViewItem friendItem = new ListViewItem(
            new string[] { textBoxMyFriendIpAddress.Text, textBoxMyFriendPort.Text, "是" });
        listViewMyFriend.Items.Add(friendItem);
        //向对方发送连接信息
        SendMessage(textBoxMyFriendIpAddress.Text, textBoxMyFriendPort.Text,
            "connect", "哈哈，我上线了");
    }
}
```

(10) 双击【发送】按钮，添加 Click 事件代码：

```
private void buttonSendMessage_Click(object sender, EventArgs e)
{
    if (listViewMyFriend.SelectedItems.Count > 0)
    {
        //可以群发
        for (int i = 0; i < listViewMyFriend.SelectedItems.Count; i++)
        {
            if (listViewMyFriend.SelectedItems[i].SubItems[2].Text == "是")
            {
                string remoteIpString = listViewMyFriend.SelectedItems[i].SubItems[0].Text;
```

```
                    string remotePortString = listViewMyFriend.SelectedItems[i].SubItems[1].Text;
                    SendMessage(remoteIpString,
                        remotePortString, "message", textBoxSendMessage.Text);
                }
            }
        }
        else
        {
            MessageBox.Show("请先选择发送的好友", "提示");
        }
    }
```

这段代码中使用 for 循环依次向对方发送信息，相当于实现了群发功能。

(11) 双击【启动刷新】按钮，添加 Click 事件代码：

```
    private void buttonStartTimer_Click(object sender, EventArgs e)
    {
        secondWatch.Reset();
        int i;
        if (int.TryParse(textBoxTimeInterval.Text, out i) == true)
        {
            if (i <= 0)
            {
                MessageBox.Show("刷新间隔必须是正整数", "范围不正确");
            }
            else
            {
                // timerSecond.Interval = i * 1000;
                timerSecond.Enabled = true;
                secondWatch.Start();
                // timer1.Start();
                // textBoxTimerStatus.Text = "已经启动定时刷新";
                buttonStartTimer.Enabled = false;
                buttonStopTimer.Enabled = true;
            }
        }
        else
        {
            MessageBox.Show("刷新间隔必须是整数", "格式不正确");
        }
    }
```

(12) 双击【停止刷新】按钮，添加 Click 事件代码：

```
    private void buttonStopTimer_Click(object sender, EventArgs e)
    {
        timerSecond.Enabled = false;
        secondWatch.Stop();
        textBoxTimerStatus.Text = "已经停止定时刷新";
        buttonStartTimer.Enabled = true;
        buttonStopTimer.Enabled = false;
    }
```

(13) 添加每次到指定的时间间隔触发的事件代码：

```
    private void timerSecond_Tick(object sender, EventArgs e)
```

```
{
    if (secondWatch.ElapsedMilliseconds / 1000 == int.Parse(textBoxTimeInterval.Text))
    {
        textBoxTimerStatus.Text = "刷新";
        timerSecond.Stop();
        secondWatch.Reset();
        for (int i = 0; i < listViewMyFriend.Items.Count; i++)
        {
            string remoteIpString = listViewMyFriend.Items[i].SubItems[0].Text;
            string remotePortString = listViewMyFriend.Items[i].SubItems[1].Text;
            SendMessage(remoteIpString, remotePortString, "check", "看看你还在线没？");
        }
        timerSecond.Start();
        secondWatch.Start();
    }
    else
    {
        textBoxTimerStatus.Text = string.Format("{0}", secondWatch.ElapsedMilliseconds / 1000);
    }
}
```

(14) 添加关闭窗体前触发的事件代码：

```
private void FormP2P_FormClosing(object sender, FormClosingEventArgs e)
{
    tcpListener.Stop();
}
```

(15) 按<F5>键编译并执行。

从这个例子可以看出，没有专用服务器一样可以实现任意基于 TCP/IP 协议的计算机之间的通信，然而在实际应用中，许多 P2P 网络还是有专用的索引服务器或中央服务器，但是这台服务器只负责存储登录到该网络上的所有用户的信息、客户端的 IP 地址以及用户提供的供共享的文件，而不是像传统 C/S 模式那样，所有传递的内容都经过服务器。

在实际应用中，一般都有专用的索引服务器。这时，当客户端 A 希望查找 P2P 网络上其他客户端提供的文件信息时，可以按照下面的思路实现：

1) 让客户端 A 以自己的用户名登录到索引服务器上。

2) 获取客户端 A 向服务器提供的可以供其他用户共享的文件名，并保存到数据库中，以便其他用户能够查找到这些文件。

3) 接收客户端 A 向服务器发出的希望查找的文件名。

4) 索引服务器在其数据库中搜索客户端 A 查找的文件名，并将搜索到的如下结果返回给客户端 A：

① 提供该文件的所有客户端的 IP 地址、端口号，例如客户端 B、C、D。

② 搜索到的文件名。

③ 需要在客户端 A 显示的有关客户端 B、C、D 的连接信息和文件的相关信息。

5) 如果客户端 A 选择了下载选项，客户端 A 就可以根据该文件的块号，使用搜索返回的 IP 地址、端口号采用多线程分别与客户端 B、C、D 建立连接。一旦成功建立连接，就可以通知对方开始发送指定文件的某一块了。例如从客户端 B 下载第 0 块，从客户端 C 下载第 1 块，从客户端 D 下载第 2 块等等。

下载完成后，客户端 A 通知索引服务器，以便索引服务器将客户端 A 的相关信息添加到数据库中，供其他客户端下载需要的该文件的某一个或多个模块。

4.3 习题 4

1．叙述 P2P 技术的特点。
2．比较 P2P 技术和传统 C/S 模式相比有哪些优缺点。

第 5 章　SMTP 与 POP3 应用编程

随着互联网的普及，电子邮件已经成为人们日常工作、生活中必不可少的通讯工具。本章主要介绍如何利用 SMTP 与 POP3 协议实现邮件的发送和接收。

5.1　通过应用程序发送电子邮件

不论是 Windows 应用程序还是 Web 应用程序，实现电子邮件的收发都是常用的功能，比如要求用户以邮件方式提供反馈信息；在办公系统中定时检测用户指定的邮件，并及时提醒用户查看等。

5.1.1　SMTP 协议

电子邮件是通过 SMTP 服务器进行发送的，SMTP 是英文 Simple Mail Transfer Protocol 的缩写，意为简单邮件传输协议，默认端口为 25。使用 SMTP 协议发送邮件时，有两种形式，一种是不使用客户端认证，即客户端可以使用匿名方式发送邮件，这种方式即是一般的 SMTP 协议；另一种要求客户端必须提供用户名密码，这种方式称为 ESMTP 协议，即 Extended SMTP，或者叫扩展 SMTP。ESMTP 与 SMTP 的区别除了是否需要认证以外，其他均相同。

客户端发送电子邮件过程是：先通过客户端软件将邮件发送到 SMTP 邮件服务器，然后再由 SMTP 邮件服务器发送到目标 SMTP 邮件服务器。

提供客户端软件的方式有两种，一种是 Windows 应用程序客户端软件，例如安装 Windows 操作系统时自动安装的 Outlook Express；另一种是提供 SMTP 服务的公司提供的 Web 应用程序，该 Web 应用程序相对于邮件客户是服务器，但相对于 SMTP 服务器则是客户端。例如 www.126.com 网站提供的邮件收发服务。

对于邮件接收者来说，接收邮件时，首先通过 POP3 协议与 SMTP 邮件服务器连接，POP 的意思是 Post Office Protocol，即邮局协议，用于电子邮件的接收。现在常用的是第三版，简称 POP3。通过 POP3 协议，客户机登录到服务器后，可以对指定的邮件进行删除或是下载到本地。

为了避免或者减少垃圾邮件，目前大部分 SMTP 邮件服务器一般均采用用户名密码认证的方式。

在 SMTP 协议中，电子邮件由三部分组成，信封、首部和正文。

1) 信封

信封包括发信人的邮件地址和接收人的邮件地址，用两条 SMTP 命令指明。

① MAIL FROM:<发信人的地址>，告诉 SMTP 服务器发信人的地址。

② RCPT TO:<收信人的地址>，告诉 SMTP 服务器收信人的地址。

2) 首部

首部中常用命令：

① FROM：<姓名><邮件地址>，表明邮件发送者是谁。

② TO：<姓名><邮件地址>，表明邮件接收者是谁。

③ SUBJECT：<邮件标题>，表明邮件的主题。

④ DATE：<时间>，表明发邮件的时间。

⑤ REPLY-TO：<邮件地址>，表明邮件的回复地址。

⑥ Content-Type：<邮件类型>，表明邮件包含文本、HTML 超文本和附件的哪些类型。

⑦ X-Priority：<邮件优先级>，表明邮件的发送优先级。

⑧MIME-Version：<版本>，MIME 的意思是 Multipurpose Internet Mail Extensions，即多用途 Internet 邮件扩展标准，它对传输内容的消息、附件及其他的内容定义了格式。

3) 正文

正文是邮件的内容。首部以一个空行结束，再下面就是正文部分。

4) 结束符号

邮件以“.”结束。

5.1.2 发送邮件

从 SMTP 协议的介绍可以看出，发送和接收邮件的内部实现过程还是比较复杂的，如果全部从底层进行编程，需要的代码就比较多。因此在.NET 框架 2.0 的 System.Net.Mail 命名空间中提供了专门对邮件进行处理的类，从而使邮件的发送变得非常简单。

对于运行在没有专用邮件服务器的大多数客户程序而言，向 SMTP 服务器发送邮件需要提供用户名和密码，服务器验证成功后，才能进行发送或接收。因此在应用程序中发送邮件，需要使用下面几个类：

1) System.Net 命名空间下的 NetworkCredential 类

该类用于提供客户端身份验证机制的凭据。其中包括标准 Internet 身份验证方法（基本、简要、协商、NTLM 和 Kerberos 身份验证）以及可以创建的自定义方法。在邮件发送中，我们需要使用这个类提供 SMTP 服务器需要的用户名和密码，用法为：

```
NetworkCredential myCredentials = new NetworkCredential("发件人邮件地址", "密码");
```

2) System.Net.Mail 命名空间下的 MailAddress 类

该类用于提供发件人和收件人的邮件地址，常用形式为：

```
MailAddress from = new MailAddress("发件人邮件地址");
MailAddress to = new MailAddress("收件人邮件地址");
```

3) System.Net.Mail 命名空间下的 MailMessage 类

该类用于提供邮件的信息，包括主题、内容、附件、信息类型等，常用形式为：

```
MailMessage message = new MailMessage(from, to);
message.Subject = "主题";
message.SubjectEncoding = System.Text.Encoding.UTF8;
message.Body = "邮件内容";
message.BodyEncoding = System.Text.Encoding.UTF8;
```

4) System.Net.Mail 命名空间下的 Attachment 类

该类用于提供附件对象，常用形式为：

```
Attachment attachFile = new Attachment("文件名");
message.Attachments.Add(attachFile);
```

5) SmtpClient 类

该类用于发送邮件，常用形式为：

```
SmtpClient client = new SmtpClient("邮件服务器地址");
client.Send(message);
```

下面通过具体例子说明发送邮件的方法。

【例 5-1】设计一个 Windows 应用程序，实现发送邮件的功能。要求利用正则表达式验证用户输入的信息，当输入信息符合要求时，才允许将邮件发送到邮件服务器。

(1) 创建一个名为 SendMailExample 的 Windows 应用程序，修改 Form1.cs 为 FormSendMail.cs，设计界面如图 5-1 所示。

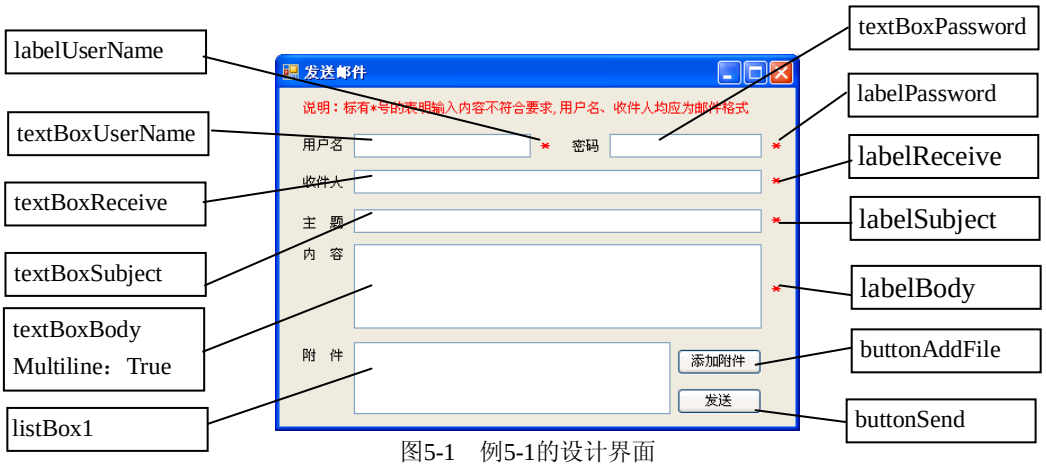


图5-1 例5-1的设计界面

(2) 切换到代码方式，添加对应的命名空间引用、事件与方法，源代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Text.RegularExpressions;
using System.Net;
using System.Net.Mail;
namespace SendMailExample
{
    public partial class FormSendMail : Form
    {
        public FormSendMail()
        {
            InitializeComponent();
        }
        //用户名（发件人地址）改变时触发
```

```
private void textBoxUserName_TextChanged(object sender, EventArgs e)
{
    //要求满足电子邮件格式
    labelUserName.Visible = !Regex.IsMatch(textBoxUserName.Text,
        @"^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$");
}
//密码改变时触发
private void textBoxPassword_TextChanged(object sender, EventArgs e)
{
    //要求满足 5-20 个英文字母或数字的组合
    labelPassword.Visible = !Regex.IsMatch(textBoxPassword.Text, @"^\w{5,20}$");
}
//收件人地址改变时触发
private void textBoxReceive_TextChanged(object sender, EventArgs e)
{
    //要求满足电子邮件格式
    labelReceive.Visible = !Regex.IsMatch(textBoxReceive.Text,
        @"^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$");
}
//主题改变时触发
private void textBoxSubject_TextChanged(object sender, EventArgs e)
{
    //不能为空
    labelSubject.Visible = !Regex.IsMatch(textBoxSubject.Text, @"^.{1,}$");
}
//发送内容改变时触发
private void textBoxBody_TextChanged(object sender, EventArgs e)
{
    //不能为空
    labelBody.Visible = !Regex.IsMatch(textBoxBody.Text, @"^.{1,}$");
}
//单击【发送】按钮触发的事件
private void buttonSend_Click(object sender, EventArgs e)
{
    string invalidString = "";
    if (labelUserName.Visible == true) invalidString += "用户名、";
    if (labelPassword.Visible == true) invalidString += "口令、";
    if (labelReceive.Visible == true) invalidString += "收件人、";
    if (labelSubject.Visible == true) invalidString += "主题、";
    if (labelBody.Visible == true) invalidString += "邮件内容、";
    if (invalidString.Length > 0)
    {
        MessageBox.Show(invalidString.TrimEnd('、') +
            "不能为空或者有不符合规定的内容");
    }
    else
    {
        //发件人和收件人地址
        MailAddress from = new MailAddress(textBoxUserName.Text);
        MailAddress to = new MailAddress(textBoxReceive.Text);
        //邮件主题、内容
```

```
            MailMessage message = new MailMessage(from, to);
            message.Subject = textBoxSubject.Text;
            message.SubjectEncoding = System.Text.Encoding.UTF8;
            message.Body = textBoxBody.Text;
            message.BodyEncoding = System.Text.Encoding.UTF8;
            //添加附件
            if (listBox1.Items.Count > 0)
            {
                for (int i = 0; i < listBox1.Items.Count; i++)
                {
                    Attachment attachFile = new Attachment(listBox1.Items[i].ToString());
                    message.Attachments.Add(attachFile);
                }
            }
            try
            {
                //大部分邮件服务器均加 smtp.前缀
                SmtpClient client = new SmtpClient("smtp." + from.Host);
                SendMail(client, from, textBoxPassword.Text, to, message);
                MessageBox.Show("发送成功");
            }
            catch (SmtpException err)
            {
                //如果错误原因是没有找到服务器，则尝试不加 smtp.前缀的服务器
                if (err.StatusCode == SmtpStatusCode.GeneralFailure)
                {
                    try
                    {
                        //有些邮件服务器不加 smtp.前缀
                        SmtpClient client = new SmtpClient(from.Host);
                        SendMail(client, from, textBoxPassword.Text, to, message);
                        MessageBox.Show("发送成功");
                    }
                    catch (SmtpException err1)
                    {
                        MessageBox.Show(err1.Message, "发送失败");
                    }
                }
                else
                {
                    MessageBox.Show(err.Message, "发送失败");
                }
            }
        }
    }
    //根据指定的参数发送邮件
    private void SendMail( SmtpClient client, MailAddress from, string password,
        MailAddress to, MailMessage message)
    {
        //不使用默认凭证,注意此句必须放在 client.Credentials 的上面
        client.UseDefaultCredentials = false;
```

```
                //指定用户名、密码
                client.Credentials = new NetworkCredential(from.Address, password);
                //邮件通过网络发送到服务器
                client.DeliveryMethod = SmtpDeliveryMethod.Network;
                try
                {
                    client.Send(message);
                }
                catch
                {
                    throw;
                }
                finally
                {
                    //及时释放占用的资源
                    message.Dispose();
                }
            }
            //单击【添加附件】按钮触发的事件
            private void buttonAddAttachment_Click(object sender, EventArgs e)
            {
                OpenFileDialog myOpenFileDialog = new OpenFileDialog();
                myOpenFileDialog.CheckFileExists = true;
                //只接收有效的文件名
                myOpenFileDialog.ValidateNames = true;
                //允许一次选择多个文件作为附件
                myOpenFileDialog.Multiselect = true;
                myOpenFileDialog.ShowDialog();
                if (myOpenFileDialog.FileNames.Length > 0)
                {
                    listBox1.Items.AddRange(myOpenFileDialog.FileNames);
                }
            }
        }
    }
```

(3) 按〈F5〉键编译并执行，在用户名中输入类似于 myname@126.com 形式的电子邮件地址，然后输入密码、收件人地址等信息。单击【发送】按钮后，程序首先根据发件人的电子邮件地址找到 SMTP 邮件服务器，并设置邮件服务器需要的验证信息，然后调用 SmtpClient 对象的 Send 方法，指定的邮件即可发送到邮件服务器中。

注意，例子中并没有对发送失败的具体原因进行处理，而只是简单的将错误信息显示出来，也没有显示附件的大小、类型等信息，但是基本功能都已经具备了。读者可以在此基础上进一步完善，做出符合实际需要的邮件发送程序。

5.2 利用同步 TCP 接收电子邮件

与发送电子邮件不同，接收电子邮件主要是利用 POP（Post Office Protocol）协议，现在常用的是第三版，简称为 POP3，默认端口为 110。通过 POP3 协议，客户机登录到服务器后，

可以对自己的邮件进行删除或下载，下载后，电子邮件客户端软件就可以在本地对邮件进行处理，随 Windows 操作系统一块安装的 Outlook Express 就是这种工作方式。

实际上，收发邮件使用了两种协议，一种是 TCP 协议，用于收发数据；另一种是 POP3 协议，用于解析传送的命令。

5.2.1 POP3 工作原理

凡是提供邮件服务的系统，除了有 SMTP 服务器外，还有 POP3 服务器。这两个服务器可能是同一台计算机，也可能是两台计算机。

POP3 邮件服务器通过侦听 TCP 端口 110 提供 POP3 服务。客户端软件读取邮件之前，需要事先与服务器建立 TCP 连接。连接成功后，POP3 服务器会向该客户端发送确认消息。然后客户端根据服务器回送的信息决定下一步的操作。

客户端每次向 POP3 服务器发送命令后，都要等待服务器响应，并处理接收的信息，然后再接着发送下一个命令，如此往复多次，一直持续到连接终止。这个过程经历了三个状态：授权（AUTHORIZATION）状态、操作（TRANSACTION）状态和更新（UPDATE）状态。

在 POP3 协议中，规定的命令只有十几条。每条命令均由命令和参数两大部分组成，而且每条命令都以回车换行结束。命令和参数之间由空格间隔。命令部分由三到四个字母组成，参数部分可达 40 个字符长度。

POP3 服务器回送的响应信息由一个状态码和一个可能跟有附加信息的命令组成。所有响应也以回车换行结束。状态码有两种：“确定”（“+OK”）和“失败”（“-ERR”）。对于客户端发送的每一条命令，服务器都会回送状态码。因此在客户端程序中，可以通过服务器回送的状态码对应的字符，即判断第一个字符是“+”号还是“-”号来确定服务器是否正确响应客户端发送的命令。

1. 授权状态

客户端首先与 POP3 服务器建立 TCP 连接，服务器接收后发送一个单行的确认信息。例如“+OK POP3 server ready”，此时 POP3 会话就进入了授权状态。在授权状态，客户需要向服务器发送用户名和密码进行确认。

假设用 C 表示客户端（Client），S 表示服务器端（Server），下面是客户端接收邮件前需要与服务器传输的信息。

1) 发送用户名。

语法形式：**USER <用户名>**

功能：将客户的用户名发送到服务器。

服务器返回：+OK 正确的用户名；-ERR 错误的用户名。

示例：C：USER myname@126.com

S：+OK welcome on this server.

上述两行代码的含义为：客户端发送“USER myname@126.com”，服务器端回送信息“+OK welcome on this server.”。

2) 用户名确认成功后，需要输入密码。

语法形式：**PASS <密码>**

功能：将客户的密码发送给服务器。

服务器返回：+OK 正确的用户名；-OK 错误的用户名。

示例：C：PASS *****

S：+OK myname logged in at 19:04

授权成功后，进入操作状态。

2. 操作状态

客户端向服务器成功确认了自己的身份后，POP3 会话将进入操作状态，客户就可以执行 POP3 命令进行相应的操作。对于每个命令，服务器都会返回应答信息。下面是在操作状态中使用的命令。

(1) STAT 命令

语法形式：**STAT**

功能：从服务器中获取邮件总数和总字节数。

服务器返回：邮件总数和总字节数。

示例：C：STAT

S：+OK 2 320

(2) LIST 命令

语法形式：**LIST**

功能：从服务中获得邮件列表和大小。

服务器返回：列出邮件列表和大小。

示例：C：LIST

S：+OK 2 messages (320 octets)

S：1　120

S：2　200

S：　.

(3) RETR 命令

语法形式：**RETR　<邮件的序号>**

功能：从服务器中获得一个邮件。

服务器返回：+OK 成功；-ERR 错误。

示例：C：　　RETR 1

S：+OK 120 octets

S：<服务器发送信件 1 内容>

S：　.

注意，这里的“.”是单独发送的。

(4) DELE 命令

语法形式：**DELE <邮件的序号>**

功能：服务器将邮件标记为删除，当执行 QUIT 命令时才真正删除。

服务器返回：+OK 成功；-ERR 错误。

示例：C：DELE 1

S：+OK 1 Deleted

当客户发送 QUIT 命令时，会话进入更新状态。

3. 更新状态

当客户在操作状态下发送 QUIT 命令后，会话进入更新状态。

QUIT 命令

语法形式：**QUIT**

功能：关闭与服务器的连接。

服务器返回：+OK；-ERR。

示例：C：QUIT

S：+OK

然后服务器自动断开与该客户端的 TCP 连接。

5.2.2 邮件接收处理

从上面的介绍可以看出，POP3 协议中传送的命令并不多，而具体传送数据的过程则是通过 TCP 协议。

下面通过一个具体例子说明邮件的接收处理过程。在这个例子中，我们重点关心的应该是如何利用同步 TCP 发送和接收数据，之所以选择邮件处理作为例子，是因为在这个例子中只需要编写客户端程序，而不需要编写服务器端程序即可正常运行。而且例子中没有采用其他线程，因此比较容易理解。

但是目前流行的大部分 TCP 应用程序，均使用多线程进行处理。因此我们的目标应该是掌握多线程 TCP 应用程序是如何设计的，这也是学习基于 C/S 编程模式的一个难点。理解了这个比较简单的邮件接收程序以后，我们再进一步介绍另外一个稍微复杂的简化的多线程网络游戏程序，到时就不会感到太困难了。

【例 5-2】利用 POP3 协议和同步 TCP 编写一个简单的邮件接收客户端程序。

(1) 新建一个名为 ReceiveMailExample 的 Windows 应用程序项目，修改 Form1.cs 为 FormReceiveMail.cs，设计窗体如图 5-2 所示。

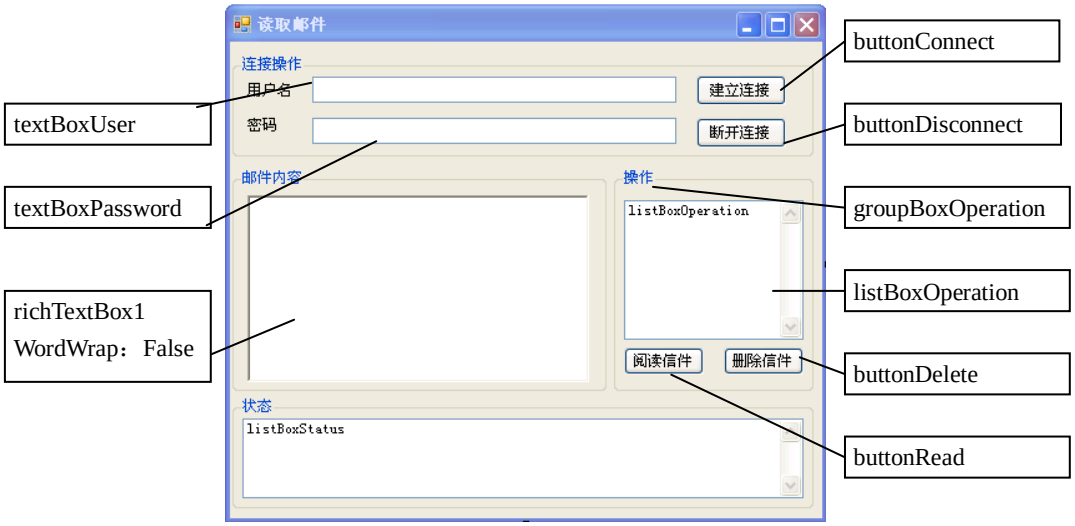


图5-2　例5-2的设计界面

(2) 切换到代码方式，添加对应的引用、字段声明、方法和事件，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.IO;
namespace ReceiveMailExample
{
    public partial class FormReceiveMail : Form
    {
        private TcpClient tcpClient;
        private NetworkStream networkStream;
        private StreamReader sr;
        private StreamWriter sw;
        public FormReceiveMail()
        {
            InitializeComponent();
        }
        //单击建立连接按钮触发的事件
        private void buttonConnect_Click(object sender, EventArgs e)
        {
            //设置鼠标光标为等待状态
            Cursor.Current = Cursors.WaitCursor;
            try
            {
                int index = textBoxUser.Text.IndexOf('@');
                //大部分 pop3 服务器都加前缀 pop3.，这里仅获取这种服务器
                //对不以 pop3.为前缀的情况，这个例子未进行处理
                string pop3Server = "pop3." + textBoxUser.Text.Substring(index + 1);
                //建立与 POP3 服务器的连接，使用默认端口 110
                tcpClient = new TcpClient(pop3Server, 110);
                listBoxStatus.Items.Add("与 pop3 服务器连接成功");
            }
            catch
            {
                MessageBox.Show("与服务器连接失败");
                return;
            }
            string str;
            listBoxStatus.Items.Clear();
            //获取  Networkstream 对象，以便通过建立好的连接发送和接收数据
            networkStream = tcpClient.GetStream();
            //得到读对象，并查找字节顺序标记，防止显示乱码
            sr = new StreamReader(networkStream);
            //得到写对象
            sw = new StreamWriter(networkStream);
```

```
        //读取服务器回送的连接信息
        if (ReadDataFromServer() == null) return;
        //向服务器发送用户名，请求确认
        if (SendDataToServer("USER " + textBoxUser.Text) == false) return;
        if (ReadDataFromServer() == null) return;
        //向服务器发送密码，请求确认
        if (SendDataToServer("PASS " + textBoxPassword.Text) == false) return;
        if (ReadDataFromServer() == null) return;
        //向服务器发送 STAT 命令，请求获取邮件总数和总字节数
        if (SendDataToServer("LIST") == false) return;
        if ((str = ReadDataFromServer()) == null) return;
        string[] splitString = str.Split(' ');
        //从字符串中取子串获取邮件总数
        int count = int.Parse(splitString[1]);
        //判断邮箱中是否有邮件
        if (count > 0)
        {
            //设置对应状态信息
            buttonRead.Enabled = true;
            buttonDelete.Enabled = true;
            listBoxOperation.Items.Clear();
            groupBoxOperation.Text = "信箱中共有 " + splitString[1] + " 封邮件";
            //向邮件列表框中添加邮件
            for (int i = 0; i < count; i++)
            {
                if ((str = ReadDataFromServer()) == null) return;
                splitString = str.Split(' ');
                listBoxOperation.Items.Add(string.Format(
                    "第{0}封邮件，{1}字节", splitString[0], splitString[1]));
            }
            listBoxOperation.SelectedIndex = 0;
            //读出结束符
            if ((str = ReadDataFromServer()) == null) return;
        }
        else
        {
            groupBoxOperation.Text = "信箱中没有邮件";
            buttonRead.Enabled = false;
            buttonDelete.Enabled = false;
        }
        buttonConnect.Enabled = false;
        buttonDisconnect.Enabled = true;
        //设置鼠标光标为默认光标
        Cursor.Current = Cursors.Default;
    }
    //向服务器发送信息
    private bool SendDataToServer(string str)
    {
        try
        {
            sw.WriteLine(str);
```

```
            sw.Flush();
            listBoxStatus.Items.Add("发送：" + str);
            return true;
        }
        catch (Exception err)
        {
            MessageBox.Show(err.Message, "发送 " + str + " 失败");
            return false;
        }
    }
    //从网络流中读取服务器回送的信息
    private string ReadDataFromServer()
    {
        //从流中读取服务器返回的信息，写入信息列表框
        string str = null;
        try
        {
            str = sr.ReadLine();
            listBoxStatus.Items.Add("收到：" + str);
            //如果是-ERR，表明有错
            if (str[0] == '-')
            {
                MessageBox.Show(str, "有错了");
                str = null;
            }
        }
        catch (Exception err)
        {
            MessageBox.Show(err.Message, "无法读取");
        }
        return str;
    }
    //单击断开连接按钮触发的事件
    private void buttonDisconnect_Click(object sender, EventArgs e)
    {
        //向服务器发送 QUIT 命令
        SendDataToServer("QUIT");
        sr.Close();
        sw.Close();
        networkStream.Close();
        tcpClient.Close();
        listBoxOperation.Items.Clear();
        richTextBox1.Clear();
        listBoxStatus.Items.Clear();
        buttonConnect.Enabled = true;
        buttonDisconnect.Enabled = false;
    }
    //单击阅读信件按钮触发的事件
    private void buttonRead_Click(object sender, EventArgs e)
    {
        //设置鼠标光标为等待状态
```

```
            Cursor.Current = Cursors.WaitCursor;
            string mailMessage = listBoxOperation.SelectedItem.ToString();
            mailMessage = mailMessage.Substring(1, mailMessage.IndexOf("封") - 1);
            if (SendDataToServer("RETR " + mailMessage) == false) return;
            string receiveData = sr.ReadToEnd();
            //例子未对邮件内容进行解码处理
            //而仅仅将源代码原样显示出来，所以看到的涉及具体内容的信息将会是乱码
            richTextBox1.Text = "源代码：\r\n";
            richTextBox1.AppendText(receiveData);
            //设置鼠标光标为默认光标
            Cursor.Current = Cursors.Default;
        }
        //单击删除信件按钮触发的事件
        private void buttonDelete_Click(object sender, EventArgs e)
        {
            string str = listBoxOperation.SelectedItem.ToString();
            str = str.Substring(1, str.IndexOf("封") - 1);
            if (SendDataToServer("DELE " + str) == true)
            {
                richTextBox1.Clear();
                int j = listBoxOperation.SelectedIndex;
                listBoxOperation.Items.Remove(listBoxOperation.Items[j].ToString());
                MessageBox.Show("删除成功", "恭喜");
            }
        }
    }
}
```

(3) 按<F5>键编译并运行，在用户名文本框中输入电子邮件地址，例如 myuser@126.com，然后输入用户密码，单击【建立连接】按钮，观察邮件接收情况。

注意，这个例子中是根据电子邮件地址自动寻找 POP3 服务器的，但是由于并不是所有 POP3 邮件服务器对应的域名均在前面加上前缀 pop3.，因此对某些随便起名字的服务器可能会出现找不到服务器的现象。在实际应用中，客户端应用程序均要求事先配置客户端程序，即使用固定的 SMTP 服务器和 POP3 服务器，也就是先告诉客户端程序申请合法的帐户名的邮件服务器域名。例如用户在 126 申请了一个帐户 myuser@126.com，则其 SMTP 服务器域名就是 smtp.126.com，POP3 服务器域名就是 pop3.126.com。

还有，为了让读者将注意力集中在对接收过程的处理上，因此例子中也没有对电子邮件的合法性进行验证。读者可以在此基础上利用学习过的正则表达式，增加邮件地址输入的验证功能。

5.3 习题 5

1．简述在 VS2005 中如何发送电子邮件。

2. 填空。

1) 电子邮件属（　　）通信方式。

2) 电子邮件由（　　）、首部和（　　）三部分组成。

3) 发送邮件经常使用的协议是（　　）。

4) 接收邮件时，客户与 POP3 服务器之间经历了三个状态，分别是（　　）状态、（　　）状态和（　　）状态。

5) 从 POP3 服务器中获得邮件列表的命令是（　　）。

第 6 章　网络数据加密与解密

数据在网络传输过程中的保密性是网络安全中重点要考虑的问题之一。由于通过网络传递数据是在不安全的信道上进行传输的，因此通信双方要想确保任何可能正在侦听的人无法理解通信的内容，而且希望确保接收方接收的信息没有在传输期间被任何人修改，最好的办法就是在传输数据前对数据进行加密，接收方接收到加密的数据后再进行解密处理，从而保证数据的安全性。

在.NET 库的 System.Security.Cryptography 命名空间中，包含多种加密数据的类，涉及多种加密算法。加密方法主要分为两大类：对称加密和不对称加密。

6.1　对称加密

对称加密也称为私钥加密，采用私钥算法，加密和解密数据使用同一个密钥。由于具有密钥的任意一方都可以使用该密钥解密数据，因此必须保证该密钥不能被攻击者获取，否则就失去了加密的意义。

私钥算法以块为单位加密数据，一次加密一个数据块。因此对称加密支持数据流，是加密流数据的理想方式。

.NET 类库使用的私钥算法有 RC2、DES、TripleDES 和 Rijndael。这些算法通过加密将 n 字节的输入块转换为加密字节的输出块。如果要加密或解密字节序列，必须逐块进行。由于 n 很小（对于 RC2、DES 和 TripleDES 算法，n 的值为 8 字节、16 字节或 24 字节，默认值为 16 字节；对于 Rijndael 算法，n 的值为 32 字节），因此每次加密的块的大小必须大于 n。实际上，一次读入的数据块是否符合私钥算法要求的块的大小，如果不符合应该如何填充使其符合要求等情况，.NET 类库提供的算法类本身会自动处理，编写程序时不需要考虑这些问题。

为了保证数据的安全，.NET 基类库中提供的私钥算法类使用称作密码块链（CBC，Cipher Block Chaining）的链模式，算法使用一个密钥和一个初始化向量（IV，Initialization Vector）对数据执行加密转换。密钥和初始化向量 IV 一起决定如何加密数据，以及如何将数据解密为原始数据。通信双方都必须知道这个密钥和初始化向量才能够加密和解密数据。

为什么要使用初始化向量 IV 呢？因为初始化向量是一个随机生成的字符集，使用它可以确保任何两个原始数据块都不会生成相同的加密后的数据块。举例来说，对于给定的私钥 k，如果不用 IV，相同的明文输入块就会加密为同样的密文输出块。显然，如果在明文流中有重复的块，那么在密文流中也会存在重复的块。对于攻击者来说，知道有关明文块结构的任何信

息，就可以使用这些信息解密已知的密文块并有可能发现密钥。为了解决这个问题，.NET Framework 中的私钥算法类将上一个块中的信息混合到下一个块的加密过程中。这样，两个相同的明文块的输出就会不同。由于该技术使用上一个块加密下一个块，因此使用了一个 IV 来加密数据的第一个块。使用这种加密技术，非法用户即使知道了公共消息标头，也无法用于对密钥进行反向工程处理，从而使数据的安全系数大大提高。

对称加密算法的优点是保密强度高，加、解密速度快，适合加密大量数据。攻击者如果对加密后的数据进行破译，惟一的办法就是对每个可能的密钥执行穷举搜索。而采用这种加密技术，即使使用最快的计算机执行这种搜索，耗费的时间也相当长。如果使用较大的密钥，破译将会更加困难。在实际应用中，加密数据采用的密钥一般都有时效性，比如几天更换一次密钥和 IV，如果攻击者采用穷举法试图破译加密后的数据，等到好不容易试出了密钥，加密者早已采用新的密钥对网络中传输的数据进行加密了，因此利用穷举搜索的方法破译加密后的数据实际上是没有意义的。

在.NET Framework 中，公共语言运行时 CLR（Common Language Runtime）使用面向流的设计实现对称加密，该设计的核心是 CryptoStream，实现 CryptoStream 的任何被加密的对象都可以和实现 Stream 的任何对象链接起来。实现对称加密算法的类有四种：

- DESCryptoServiceProvider
- RC2CryptoServiceProvider
- RijndaelManaged
- TripleDESCryptoServiceProvider

表 6-1 列出了四种对称加密类的主要特点。

表 6-1　四种对称加密类的主要特点

类	可用密钥长度（bit）	加密算法
DESCryptoServiceProvider	64	DES加密算法
RC2CryptoServiceProvider	40-128（每 8 位递增）	RC2 加密算法
RijndaelManaged	128-256（每 64 位递增）	Rijndael加密算法
TripleDESCryptoServiceProvider	128-192（每 64 位递增）	三重DES加密算法

这里仅介绍 TripleDES 加密算法的相关知识和使用方法，其他对称加密类的用法与此相似。

TripleDES 使用 DES 算法的三次连续迭代，支持从 128 位到 192 位（以 64 位递增）的密钥长度，其安全性比 DES 更高。DES 的含义是 Data Encryption Standard，是美国 1977 年公布的一种数据加密标准，DES 算法在各超市零售业、银行自动取款机、磁卡及 IC 卡、加油站、高速公路收费站等领域被广泛应用，以此来实现关键数据的保密，如信用卡持卡人的 PIN 的加密传输，IC 卡的认证、金融交易数据包的 MAC 校验等，均用到 DES 算法。DES 算法具有非常高的安全性，到目前为止，除了用穷举搜索法对 DES 算法进行攻击外，还没有发现更有效的办法。而 56 位长的密钥的穷举空间为 256，这意味着如果一台计算机的速度是每一秒种检测一百万个密钥，则它搜索全部密钥就需要将近 2285 年的时间。可见，攻击的难度是非常大的。但是，随着科学技术的发展，当出现超高速计算机后，以及用多台计算机同时进行穷举搜索，会大大缩短破译的时间，因此，为了增大攻击者破译的难度，TripleDES 在 DES 的基础上又进行了三次迭代，密钥的长度最大可达 192 位，使保密程度得到更进一步的提高。

表 6-2 列出了 TripleDESCryptoServiceProvider 类常用的属性和方法。

表 6-2　TripleDESCryptoServiceProvider 类常用的属性和方法

名称	解释
BlockSize属性	获取或设置加密操作的快大小，以位为单位
Key属性	获取或设置TripleDES算法的机密密钥
IV属性	获取或设置TripleDES算法的初始化向量
KeySize属性	获取或设置TripleDES算法所用密钥的大小，以位为单位
CreateEncryptor方法	创建TripleDES加密器对象
CreateDecryptor方法	创建TripleDES解密器对象
GenerateIV方法	生成用于TripleDES算法的随机初始化向量IV
GenerateKey方法	生成用于TripleDES算法的随机密钥

为了使用流进行加密解密处理，.NET Framework 还提供了 CryptoStream 类，该类用于定义将数据流链接到加密转换的流。实现 CryptoStream 的任何加密对象均可以和实现 Stream 的任何对象链接起来，因此一个对象的流式处理输出可以馈送到另一个对象的输入，而不需要分别存储中间结果，即不需要存储第一个对象的输出。

CryptoStream 对象的用法和其他流的用法相似，这里不再重复介绍。但是要注意，完成 CryptoStream 对象的使用后，不要忘了调用 Close 方法关闭该对象。Close 方法会刷新流并使所有剩余的数据块都被 CryptoStream 对象处理。由于在调用 Close 方法前对流的读写操作有可能会出现异常，所以为确保流处理能够正常关闭，一般在 try/catch 语句的 finally 块中调用 Close 方法。

例 6-1 说明了该类的使用方法。为了让读者将注意力集中在如何加密和解密上，这个例子没有通过网络传递加密后的数据，而是全部在同一台计算机上进行加密解密处理。

【例 6-1】使用 TripleDES 加密算法对输入的字符串进行加密，并输出加密后的字符串和解密后的结果。

(1) 新建一个名为 TdesEncryptExample 的 Windows 应用程序，修改 Form1.cs 为 FormTdesEncrypt.cs，设计界面如图 6-1 所示。

图6-1　例6-1的设计界面

(2) 添加对应的命名空间引用、方法和事件，源程序如下：

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

```
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Security.Cryptography;
using System.IO;
namespace TdesEncryptExample
{
    public partial class FormTdesEncrypt : Form
    {
        public FormTdesEncrypt()
        {
            InitializeComponent();
        }
        private void FormTdesEncrypt_Load(object sender, EventArgs e)
        {
            textBoxEncrypt.ReadOnly = true;
            textBoxDecrypt.ReadOnly = true;
        }
        private void buttonOK_Click(object sender, EventArgs e)
        {
            string str = textBoxInput.Text;
            if (str.Length == 0)
            {
                MessageBox.Show("请输入被加密的字符串");
                return;
            }
            //加密
            try
            {
                TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();
                //随机生成密钥 Key 和初始化向量 IV
                tdes.GenerateKey();
                tdes.GenerateIV();
                textBoxKey.Text = Encoding.UTF8.GetString(tdes.Key);
                //得到加密后的字节流
                byte[] encryptedBytes = EncryptText(str, tdes.Key, tdes.IV);
                //显示加密后的字符串
                textBoxEncrypt.Text = Encoding.UTF8.GetString(encryptedBytes);
                //解密
                string decryptString = DecryptText(encryptedBytes, tdes.Key, tdes.IV);
                //显示解密后的字符串
                textBoxDecrypt.Text = decryptString;
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message, "出错");
            }
        }
        private byte[] EncryptText(string str, byte[] Key, byte[] IV)
```

```
{
    //创建一个内存流
    MemoryStream memoryStream = new MemoryStream();
    //使用传递的私钥和 IV 创建加密流
    CryptoStream cryptoStream = new CryptoStream(memoryStream,
        new TripleDESCryptoServiceProvider().CreateEncryptor(Key, IV),
        CryptoStreamMode.Write);
    //将传递的字符串转换为字节数组
    byte[] toEncrypt = Encoding.UTF8.GetBytes(str);
    try
    {
        //将字节数组写入加密流,并清除缓冲区
        cryptoStream.Write(toEncrypt, 0, toEncrypt.Length);
        cryptoStream.FlushFinalBlock();
        //得到加密后的字节数组
        byte[] encryptedBytes = memoryStream.ToArray();
        return encryptedBytes;
    }
    catch (CryptographicException err)
    {
        throw new Exception("加密出错：" + err.Message);
    }
    finally
    {
        cryptoStream.Close();
        memoryStream.Close();
    }
}
private string DecryptText(byte[] dataBytes, byte[] Key, byte[] IV)
{
    //根据加密后的字节数组创建一个内存流
    MemoryStream memoryStream = new MemoryStream(dataBytes);
    //使用传递的私钥、IV 和内存流创建解密流
    CryptoStream cryptoStream = new CryptoStream(memoryStream,
        new TripleDESCryptoServiceProvider().CreateDecryptor(Key, IV),
        CryptoStreamMode.Read);
    //创建一个字节数组保存解密后的数据
    byte[] decryptBytes = new byte[dataBytes.Length];
    try
    {
        //从解密流中将解密后的数据读到字节数组中
        cryptoStream.Read(decryptBytes, 0, decryptBytes.Length);
        //得到解密后的字符串
        string decryptedString = Encoding.UTF8.GetString(decryptBytes);
        return decryptedString;
    }
    catch (CryptographicException err)
    {
        throw new Exception("解密出错：" + err.Message);
    }
    finally
```

```
                {
                    cryptoStream.Close();
                    memoryStream.Close();
                }
            }
        }
    }
```

(3) 按<F5>键编译并执行，输入一些字符串，然后单击开始加密和解密按钮，运行效果如图 6-2 所示。

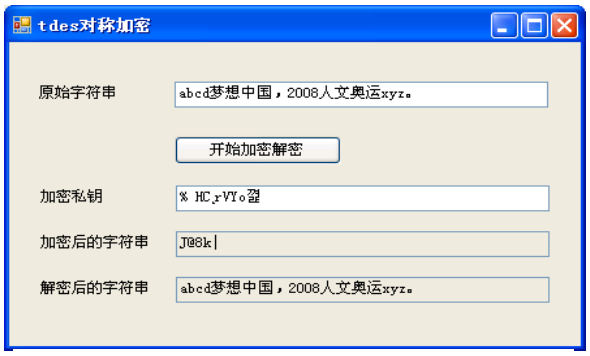


图6-2　例6-1的运行效果

6.2　不对称加密

对称加密的缺点是双方使用相同的密钥和 IV 进行加密、解密。由于接收方必须知道密钥和 IV 才能解密数据，因此发送方需要先将密钥和 IV 传递给接收方。这就有一个问题，如果攻击者截获了密钥和 IV，也就等于知道了如何解密数据！如何保证发送方传递给接收方的密钥和 IV 不被攻击者截获并破译呢？

不对称加密也叫公钥加密，这种技术使用不同的加密密钥与解密密钥，是一种“由已知加密密钥推导出解密密钥在计算上是不可行的”密码体制。不对称加密产生的主要原因有两个，一是对称加密的密钥分配问题，另一个是由于对数字签名的需求。

不对称加密使用一个需要保密的私钥和一个可以对任何人公开的公钥，即使用公钥/私钥对来加密和解密数据。公钥和私钥都在数学上相关联，用公钥加密的数据只能用私钥解密，反之，用私钥加密的数据只能用公钥解密。两个密钥对于通信会话都是惟一的。公钥加密算法也称为不对称算法，原因是需要用一个密钥加密数据而需要用另一个密钥来解密数据。

私钥加密算法使用长度可变的缓冲区，而公钥加密算法使用固定大小的缓冲区，无法像私钥算法那样将数据链接起来成为流，因此无法使用与对称操作相同的流模型。这是编写程序时必须注意的问题。

为什么不对称加密更不容易被攻击呢？关键在于对私钥的管理上。在对称加密中，发送方必须先将解密密钥传递给接收方，接收方才能解密。如果避免通过不安全的网络传递私钥，不就解决这个问题了吗？

不对称加密的关键就在于此。使用不对称加密算法加密数据后，私钥不是发送方传递给接收方的，而是接收方先生成一个公钥/私钥对，在接收被加密的数据前，先将该公钥传递给发

送方；注意，从公钥推导出私钥是不可能的，所以不怕通过网络传递时被攻击者截获公钥。发送方得到此公钥后，使用此公钥加密数据，再将加密后的数据通过网络传递给接收方；接收方收到加密后的数据后，再用私钥进行解密。由于没有传递私钥，从而保证了数据安全性。

.NET Framework 提供以下实现不对称加密算法的类：

- DSACryptoServiceProvider
- RSACryptoServiceProvider

下面以 RSACryptoServiceProvider 类为例介绍具体的使用方法。

RSACryptoServiceProvider 类使用加密服务提供程序提供的 RSA 算法实现不对称加密和解密。加密服务提供程序 CSP（Cryptographic Service Provider）是微软在 Windows 操作系统中内置的加密处理模块，RSACryptoServiceProvider 类已经对其提供的相关接口和参数进行了封装，所以即使我们不知道 CSP 内部是如何实现的，也一样可以使用其提供的功能。表 6-3 列出了 RSACryptoServiceProvider 类的部分属性和方法。

表 6-3　RSACryptoServiceProvider 类的部分属性和方法

名称	解释
CspKeyContainerInfo 属性	检索关于加密密钥对的相关信息，比如密钥是否可导出、密钥容器名称以及提供程序的信息等
PersistKeyInCsp属性	获取或设置一个值，该值指示密钥是否应该永久驻留在加密服务提供程序(CSP)中。当在CspParameter对象中指定密钥容器名称并用其初始化RSACryptoServiceProvider对象时，PersistKeyInSsp属性自动设置为true
PublicOnly属性	获取一个值，该值指示RSACryptoServiceProvider对象是否仅包含一个公钥。如果RSACryptoServiceProvider对象仅包含一个公钥，则为true，否则为false
Encrypt方法	使用RSA算法对数据进行加密。该方法有两个参数，第一个参数是被加密的字节数组；第二个参数是填充方式（true表示使用OAEP方式填充，false表示使用PKCS#1 1.5 版填充），如果操作系统是Windows XP及其以上版本，可以使用true，如果是Windows 2000 及其以上版本，使用false
Decrypt方法	使用RSA算法对数据进行解密
ImportParameters方法	导入指定的RSAParameters。RSAParameters表示RSA算法涉及的相关参数
ExportParameters方法	导出指定的RSAParameters
FromXmlString方法	通过XML字符串中的密钥信息初始化RSA对象。该XML字符串是使用ToXmlString方法生成的。FromXmlString方法既可接受包含公钥的XML字符串，也可接受包含公钥和私钥的XML字符串
ToXmlString方法	创建并返回包含当前RSA对象的密钥的XML字符串。该方法有一个布尔型参数，true表示同时包含RSA公钥和私钥，false表示仅包含公钥

例 6-2 演示了利用 RSACryptoServiceProvider 类加密和解密数据的方法。与例 6-1 的思路一样，为了让读者将注意力集中在如何加密和解密上，这个例子中仍然采用在同一台计算机上进行加密和解密处理。

【例 6-2】利用不对称加密算法加密指定的字符串，并输出加密和解密后的结果。

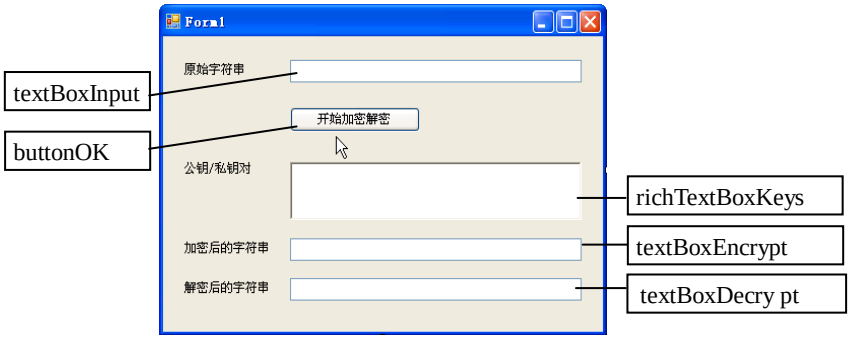


图6-3　例6-2的设计界面

(1) 新建一个名为 RsaEncryptExample 的 Windows 应用程序，修改 Form1.cs 为 FormRsaEncrypt.cs，设计界面如图 6-3 所示。

(2) 添加对应的命名空间引用、方法和事件，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.IO;
using System.Security.Cryptography;
namespace RsaEncryptExample
{
    public partial class FormRsaEncrypt : Form
    {
        public FormRsaEncrypt()
        {
            InitializeComponent();
            this.Text = "RSA 加密解密";
            textBoxEncrypt.ReadOnly = true;
            textBoxDecrypt.ReadOnly = true;
        }
        private void buttonOK_Click(object sender, EventArgs e)
        {
            //使用默认密钥创建 RSACryptoServiceProvider 对象
            RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
            //显示包含公钥/私钥对的 XML 表示形式，如果只显示公钥，将参数改为 false 即可
            richTextBoxKeys.Text = rsa.ToXmlString(true);
            //将被加密的字符串转换为字节数组
            byte[] dataToEncrypt = Encoding.UTF8.GetBytes(textBoxInput.Text);
            try
            {
                //得到加密后的字节数组
                byte[] encryptedData = rsa.Encrypt(dataToEncrypt, false);
                textBoxEncrypt.Text = Encoding.UTF8.GetString(encryptedData);
```

```
                //得到解密后的字节数组
                byte[] decryptedData = rsa.Decrypt(encryptedData, false);
                textBoxDecrypt.Text = Encoding.UTF8.GetString(decryptedData);
            }
            catch (Exception err)
            {
                MessageBox.Show(err.Message);
            }
        }
    }
}
```

(3) 按<F5>键编译并执行，输入一些字符串，然后单击开始加密和解密按钮，运行效果如图 6-4 所示。

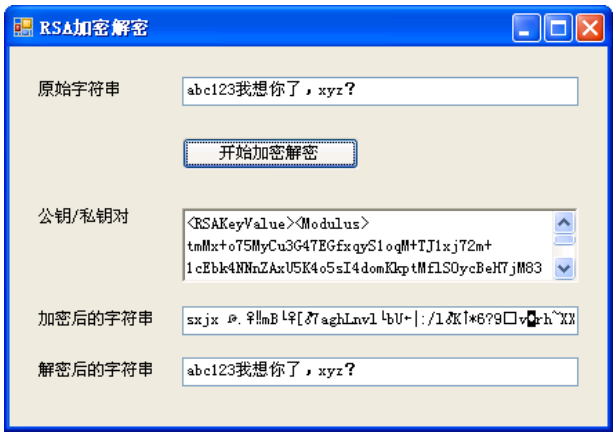


图6-4　例6-2的运行效果

从这个例子可以看出，使用不对称加密类 RSACryptoServiceProvider 加密和解密数据的过程并不复杂。但是，在网络应用编程中，RSACryptoServiceProvider 类的主要用途是加密和解密通过网络传递的对称加密用的私钥。而加密大量数据则应该使用对称加密算法。通过不安全的网络将用对称加密算法加密的数据传递到接收方后，接收方必须知道对称加密用的密钥才能够解密。而且实际应用中的密钥还需要经常更换，以避免攻击者破译。为了解决传递对称加密密钥的安全问题，发送方可以用不对称加密算法加密对称加密算法用的密钥，并将加密后的密钥传递给接收方，以便接收方用此密钥解密数据。可要完成这个功能，接收方必须在接收对称加密算法加密的数据前先将不对称加密的公钥传递给发送方，以便发送方据此加密对称加密的密钥。可见，使用 RSACryptoServiceProvider 类的关键是如何导出公钥，并将公钥通过网络传递给发送方。

还有一点要注意，本节的例子是使用 XML 的格式通过网络传递公钥的，这是因为公钥不需要保密。但是千万不要将私钥以 XML 形式通过网络传递，也不要将私钥以 XML 形式存储在本地计算机上，如果确实需要保存公钥/私钥对，应该使用密钥容器。

顾名思义，密钥容器就是保存密钥用的容器，使用密钥容器的目的是为了保证密钥的安全性。由于密钥容器可以有多个，为了区分是哪一个密钥容器，一般都要给每个密钥容器起一个名称。在 System.Security.Cryptography 命名空间中，有一个 CspParameters 类，可以通过该类提供的属性设置或获取密钥容器的名称。

6.3 通过网络传递加密数据

虽然不对称加密解决了用对称加密传递消息必须传递密钥的问题，但是由于不对称加密无法使用流进行处理，因此与对称加密相比效率较低，不适用于加密大量数据的场合。在实际应用中，一般将两种加密方法配合使用。其基本思想是：用不对称加密算法加密对称加密算法的密钥，用对称加密算法加密实际数据。

具体设计思路可以简单描述为：A 和 B 相互传递加密的数据前，B 首先生成一个不对称加密算法使用的公钥/私钥对，假定公钥为 publicKey，私钥为 privateKey，然后 B 将公钥 publicKey 通过网络传递给 A；A 接收到此公钥后，根据此公钥初始化不对称加密对象，并用此对象加密使用对称加密算法的密钥 key，并将加密后的密钥 key 通过网络传递给 B；这样，A 和 B 都有了一个共同使用的对称加密的密钥，然后双方用此密钥加密数据，并将加密后的数据传递给对方，对方收到加密后的数据后，再用密钥 key 解密数据。

通过这种方式，在不安全的网络上传递加密后的数据时，虽然攻击者可以截获公钥，但是由于用公钥加密的数据只能用私钥解密，而私钥并没有通过网络传递，因此攻击者无法通过公钥 publicKey 破译加密后的密钥 key，因此也无法破译加密的消息。

在实际应用中，一般使用 TCP 协议通过网络传输数据。对于比较重要的数据，必须进行加密解密处理。一般实现方案为：

1) 传输双方均各自生成一个公钥/私钥对。
2) 通过 TCP 协议交换公钥。
3) 双方各自生成一个对称加密用的私钥，并使用对方的公钥加密新创建的私钥。
4) 双方将加密后的对称加密用的私钥发送给对方，以便对方利用此私钥解密。
5) 双方使用对称加密进行会话。

采用 TCP 协议进行网络数据传输时，注意不要忘了解决 TCP 协议的消息边界问题。对于发送大量数据的场合，一般的解决办法是，将数据发送到网络流之前，先计算出每个加密后的数据包的长度，然后将数据包的长度和数据全部发送到网络流中。图 6-5 说明了发送方和接收方的网络传输的过程。从图中可以看出，在通过网络传输数据之前，发送方先读取一个数据块，进行加密，并将加密后的数据保存在内存流中，然后计算加密后的数据长度，最后将数据长度和内存流中的数据转换成字节序列，通过网络流发送给接收方；接收方接收数据时，首先从网络流中获取要读取的加密后的数据量的大小值，然后根据获取的要读取的字节数，从网络流中读取数据，并解密这些数据到内存流中，再把内存流中的数据转换成字节序列，从而形成原始数据。对于较大的不能一次传输的数据，循环执行这个过程，直到数据全部传输完毕。

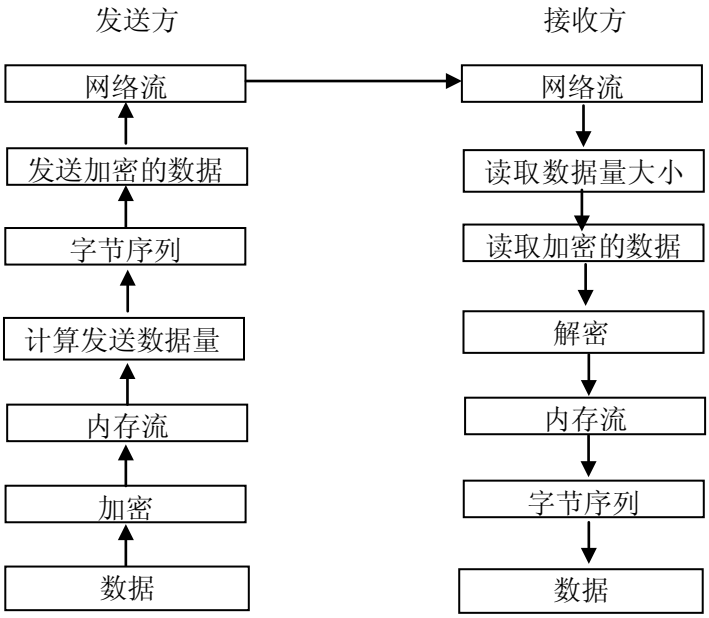


图6-5　数据加密与解密的网络传输过程

下面通过一个例子说明具体的实现方法，为了不使问题复杂化，以便读者容易理解，这个例子对实际的实现方案进行了简化处理，简化后的设计思路为：

1) 客户端生成一个使用 RSA 算法的不对称加密的公钥/私钥对，然后通过 TCP 协议将公钥发送到服务器端。

2) 服务器端用客户端发送的公钥初始化 RSA 对象。然后利用此对象加密使用 TripleDES 算法的对称加密的密钥。

3) 服务器端将加密后的对称加密的密钥发送到客户端，客户端利用 RSA 的私钥解密 TripleDES 密钥，并用此密钥初始化 TripleDES 对象。

4) 双方使用对称加密算法加密对话内容，并将加密后的对话内容发送给对方。

5) 接收方接收到加密后的对话内容后，利用对称加密算法解密对话内容，并显示解密前和解密后的结果。

【例 6-3】利用同步 TCP 传递会话数据。要求使用不对称加密算法加密对称加密算法使用的私钥，使用对称加密算法加密会话信息。

1. 服务器端设计

(1) 新建一个名为 EncryptedTcpServer 的 Windows 应用程序，修改 Form1.cs 为 FormServer.cs，设计界面如图 6-6 所示。

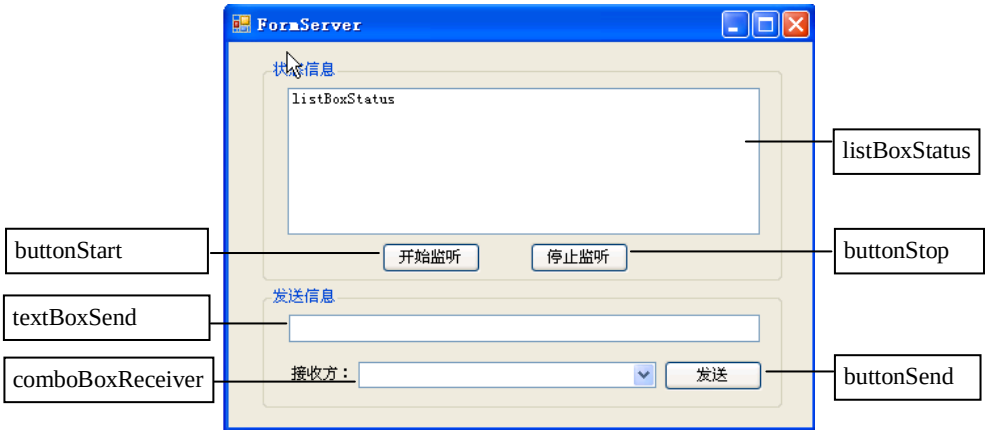


图6-6　服务器端设计界面

(2) 添加一个类文件 User.cs，源程序如下：

```
//-------------User.cs----------------//
using System.Net.Sockets;
using System.IO;
using System.Text;
using System.Security.Cryptography;
namespace EncryptedTcpServer
{
    class User
    {
        public TcpClient client;
        public BinaryReader br;
        public BinaryWriter bw;
        //对称加密
        public TripleDESCryptoServiceProvider tdes;
        //不对称加密
        public RSACryptoServiceProvider rsa;
        public User(TcpClient client)
        {
            this.client = client;
            NetworkStream networkStream = client.GetStream();
            br = new BinaryReader(networkStream, Encoding.UTF8);
            bw = new BinaryWriter(networkStream, Encoding.UTF8);
            tdes = new TripleDESCryptoServiceProvider();
            //随机生成密钥 Key 和初始化向量 IV,也可以不用此两句，而使用默认的 Key 和 IV
            //tdes.GenerateKey();
            //tdes.GenerateIV();
            rsa = new RSACryptoServiceProvider();
        }
    }
}
```

(3) 在 FormServer.cs 中添加对应的代码，源程序如下：

```
//-------------FormServer.cs----------------//
using System;
using System.Collections.Generic;
```

```
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;
using System.Security.Cryptography;
namespace EncryptedTcpServer
{
    public partial class FormServer : Form
    {
        //连接的用户
        System.Collections.Generic.List<User> userList = new List<User>();
        private delegate void SetListBoxCallback(string str);
        private SetListBoxCallback setListBoxCallback;
        private delegate void SetComboBoxCallback(User user);
        private SetComboBoxCallback setComboBoxCallback;
        //使用的本机 IP 地址
        IPAddress localAddress;
        //监听端口
        private int port = 51888;
        private TcpListener myListener;
        public FormServer()
        {
            InitializeComponent();
            listBoxStatus.HorizontalScrollbar = true;
            setListBoxCallback = new SetListBoxCallback(SetListBox);
            setComboBoxCallback = new SetComboBoxCallback(AddComboBoxitem);
            IPAddress[] addrIP = Dns.GetHostAddresses(Dns.GetHostName());
            localAddress = addrIP[0];
            buttonStop.Enabled = false;
        }
        //【开始监听】按钮的 Click 事件
        private void buttonStart_Click(object sender, EventArgs e)
        {
            myListener = new TcpListener(localAddress, port);
            myListener.Start();
            SetListBox(string.Format("开始在{0}:{1}监听客户连接", localAddress, port));
            //创建一个线程监听客户端连接请求
            ThreadStart ts = new ThreadStart(ListenClientConnect);
            Thread myThread = new Thread(ts);
            myThread.Start();
            buttonStart.Enabled = false;
            buttonStop.Enabled = true;
        }
        //接收客户端连接的线程
        private void ListenClientConnect()
```

```
{
    while (true)
    {
        TcpClient newClient = null;
        try
        {
            //等待用户进入
            newClient = myListener.AcceptTcpClient();
        }
        catch
        {
            //当单击“停止监听”或者退出此窗体时 AcceptTcpClient()会产生异常
            //因此可以利用此异常退出循环
            break;
        }
        //每接受一个客户端连接,就创建一个对应的线程循环接收该客户端发来的信息
        ParameterizedThreadStart pts = new ParameterizedThreadStart(ReceiveData);
        Thread threadReceive = new Thread(pts);
        User user = new User(newClient);
        threadReceive.Start(user);
        userList.Add(user);
        AddComboBoxitem(user);
        SetListBox(string.Format("[{0}]进入", newClient.Client.RemoteEndPoint));
        SetListBox(string.Format("当前连接用户数：{0}", userList.Count));
    }
}
//接收、处理客户端信息的线程，每客户 1 个线程，参数用于区分是哪个客户
private void ReceiveData(object obj)
{
    User user = (User)obj;
    TcpClient client = user.client;
    //是否正常退出接收线程
    bool normalExit = false;
    //用于控制是否退出循环
    bool exitWhile = false;
    while (exitWhile == false)
    {
        //保存接收的命令字符串
        string receiveString = null;
        //解析命令用
        //每条命令均带有一个参数，值为 true 或者 false，表示是否有紧跟的字节数组
        string[] splitString = null;
        byte[] receiveBytes = null;
        try
        {
            //从网络流中读出命令字符串
            //此方法会自动判断字符串长度前缀，并根据长度前缀读出字符串
            receiveString = user.br.ReadString();
            splitString = receiveString.Split(',');
            if (splitString[1] == "true")
            {
```

```
            //先从网络流中读出 32 位的长度前缀
            int bytesLength = user.br.ReadInt32();
            //然后读出指定长度的内容保存到字节数组中
            receiveBytes = user.br.ReadBytes(bytesLength);
        }
    }
    catch
    {
        //底层套接字不存在时会出现异常
        SetListBox("接收数据失败");
    }
    if (receiveString == null)
    {
        if (normalExit == false)
        {
            //如果停止了监听，Connected 为 false
            if (client.Connected == true)
            {
                SetListBox(string.Format(
                    "与[{0}]失去联系，已终止接收该用户信息",
                    client.Client.RemoteEndPoint));
            }
        }
        break;
    }
    SetListBox(string.Format("来自[{0}]：{1}",
        user.client.Client.RemoteEndPoint, receiveString));
    if (receiveBytes != null)
    {
        SetListBox(string.Format("来自[{0}]：{1}",
            user.client.Client.RemoteEndPoint,
            Encoding.Default.GetString(receiveBytes)));
    }
    switch (splitString[0])
    {
        case "rsaPublicKey":
            //使用传递过来的公钥重新初始化该客户端对
            //应的 RSACryptoServiceProvider 对象，
            //然后就可以使用这个对象加密对称加密的私钥了
            user.rsa.FromXmlString(Encoding.Default.GetString(receiveBytes));
            //加密对称加密的私钥
            try
            {
                //使用 RSA 算法加密对称加密算法的私钥 Key
                byte[] encryptedKey = user.rsa.Encrypt(user.tdes.Key, false);
                SendToClient(user, "tdesKey,true", encryptedKey);
                //加密 IV
                byte[] encryptedIV = user.rsa.Encrypt(user.tdes.IV, false);
                SendToClient(user, "tdesIV,true", encryptedIV);
            }
            catch (Exception err)
```

```
                    {
                        MessageBox.Show(err.Message);
                    }
                    break;
                case "Logout":
                    //格式：Logout
                    SetListBox(string.Format("[{0}]退出",
                        user.client.Client.RemoteEndPoint));
                    normalExit = true;
                    exitWhile = true;
                    break;
                case "Talk":
                    //解密
                    string talkString = DecryptText(receiveBytes, user.tdes.Key, user.tdes.IV);
                    if (talkString != null)
                    {
                        SetListBox(string.Format("[{0}]说：{1}",
                            client.Client.RemoteEndPoint, talkString));
                    }
                    break;
                default:
                    SetListBox("什么意思啊：" + receiveString);
                    break;
            }
        }
        userList.Remove(user);
        client.Close();
        SetListBox(string.Format("当前连接用户数：{0}", userList.Count));
    }
    //使用对称加密加密字符串
    private byte[] EncryptText(string str, byte[] Key, byte[] IV)
    {
        //创建一个内存流
        MemoryStream memoryStream = new MemoryStream();
        //使用传递的私钥和 IV 创建加密流
        CryptoStream cryptoStream = new CryptoStream(memoryStream,
            new TripleDESCryptoServiceProvider().CreateEncryptor(Key, IV),
            CryptoStreamMode.Write);
        //将传递的字符串转换为字节数组
        byte[] toEncrypt = Encoding.UTF8.GetBytes(str);
        try
        {
            //将字节数组写入加密流,并清除缓冲区
            cryptoStream.Write(toEncrypt, 0, toEncrypt.Length);
            cryptoStream.FlushFinalBlock();
            //得到加密后的字节数组
            byte[] encryptedBytes = memoryStream.ToArray();
            return encryptedBytes;
        }
        catch (Exception err)
        {
```

```
            SetListBox("加密出错：" + err.Message);
            return null;
        }
        finally
        {
            cryptoStream.Close();
            memoryStream.Close();
        }
    }
    //使用对称加密算法解密接收的字符串
    private string DecryptText(byte[] dataBytes, byte[] Key, byte[] IV)
    {
        //根据加密后的字节数组创建一个内存流
        MemoryStream memoryStream = new MemoryStream(dataBytes);
        //使用传递的私钥、IV 和内存流创建解密流
        CryptoStream cryptoStream = new CryptoStream(memoryStream,
            new TripleDESCryptoServiceProvider().CreateDecryptor(Key, IV),
            CryptoStreamMode.Read);
        //创建一个字节数组保存解密后的数据
        byte[] decryptBytes = new byte[dataBytes.Length];
        try
        {
            //从解密流中将解密后的数据读到字节数组中
            cryptoStream.Read(decryptBytes, 0, decryptBytes.Length);
            //得到解密后的字符串
            string decryptedString = Encoding.UTF8.GetString(decryptBytes);
            return decryptedString;
        }
        catch (Exception err)
        {
            SetListBox("解密出错：" + err.Message);
            return null;
        }
        finally
        {
            cryptoStream.Close();
            memoryStream.Close();
        }
    }
    //发送信息到客户端
    private void SendToClient(User user, string command, byte[] bytes)
    {
        //每条命令均带有一个参数，值为 true 或者 false，表示是否有紧跟的字节数组
        string[] splitCommand = command.Split(',');
        try
        {
            //先将命令字符串写入网络流，此方法会自动附加字符串长度前缀
            user.bw.Write(command);
            SetListBox(string.Format("向[{0}]发送：{1}",
                user.client.Client.RemoteEndPoint, command));
            if (splitCommand[1] == "true")
```

```
            {
                //先将字节数组的长度（32 位整数）写入网络流
                user.bw.Write(bytes.Length);
                //然后将字节数组写入网络流
                user.bw.Write(bytes);
                user.bw.Flush();
                SetListBox(string.Format("向[{0}]发送：{1}",
                    user.client.Client.RemoteEndPoint, Encoding.UTF8.GetString(bytes)));
                if (splitCommand[0] == "Talk")
                {
                    SetListBox("加密前内容：" + textBoxSend.Text);
                }
            }
        }
        catch
        {
            SetListBox(string.Format("向[{0}]发送信息失败",
                user.client.Client.RemoteEndPoint));
        }
    }
    private void AddComboBoxitem(User user)
    {
        if (comboBoxReceiver.InvokeRequired == true)
        {
            this.Invoke(setComboBoxCallback, user);
        }
        else
        {
            comboBoxReceiver.Items.Add(user.client.Client.RemoteEndPoint);
        }
    }
    private void SetListBox(string str)
    {
        if (listBoxStatus.InvokeRequired == true)
        {
            this.Invoke(setListBoxCallback, str);
        }
        else
        {
            listBoxStatus.Items.Add(str);
            listBoxStatus.SelectedIndex = listBoxStatus.Items.Count - 1;
            listBoxStatus.ClearSelected();
        }
    }
    //单击停止监听按钮触发的事件
    private void buttonStop_Click(object sender, EventArgs e)
    {
        SetListBox(string.Format("目前连接用户数：{0}", userList.Count));
        SetListBox("开始停止服务，并依次使用户退出!");
        for (int i = 0; i < userList.Count; i++)
        {
```

```
                    comboBoxReceiver.Items.Remove(userList[i].client.Client.RemoteEndPoint);
                    userList[i].bw.Close();
                    userList[i].br.Close();
                    userList[i].client.Close();
                }
                //通过停止监听让 myListener.AcceptTcpClient()产生异常退出监听线程
                myListener.Stop();
                buttonStart.Enabled = true;
                buttonStop.Enabled = false;
            }
            //单击【发送】按钮的 Click 事件
            private void buttonSend_Click(object sender, EventArgs e)
            {
                int index = comboBoxReceiver.SelectedIndex;
                if (index == -1)
                {
                    MessageBox.Show("请先选择接收方，然后再单击［发送］");
                }
                else
                {
                    User user = (User)userList[index];
                    //加密 textBoxSend.Text 的内容
                    byte[] encryptedBytes = EncryptText(textBoxSend.Text, user.tdes.Key, user.tdes.IV);
                    if (encryptedBytes != null)
                    {
                        SendToClient(user, "Talk,true", encryptedBytes);
                        textBoxSend.Clear();
                    }
                }
            }
            private void FormServer_FormClosing(object sender, FormClosingEventArgs e)
            {
                //未单击开始监听就直接退出时，myListener 为 null
                if (myListener != null)
                {
                    buttonStop_Click(null, null);
                }
            }
            //textBoxSend 获得焦点并释放按键后触发的事件
            private void textBoxSend_KeyPress(object sender, KeyPressEventArgs e)
            {
                if (e.KeyChar == (char)Keys.Return)
                {
                    buttonSend_Click(null, null);
                }
            }
        }
    }
```

(4) 按<F5>键编译并运行，确保没有语法错误。

2. 客户端设计

(1) 新建一个名为 EncryptedTcpClient 的 Windows 应用程序，修改 Form1.cs 为 FormClient.cs，设计界面如图 6-7 所示。

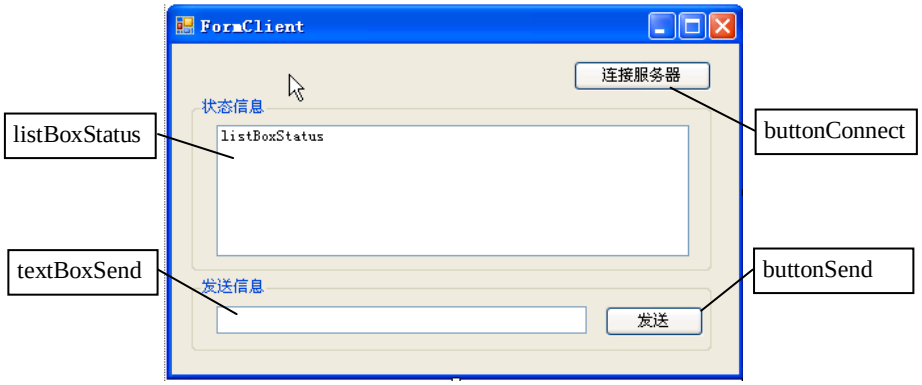


图6-7　客户端设计界面

(2) 添加对应的代码，源程序如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;
using System.Security.Cryptography;
namespace EncryptedTcpClient
{
    public partial class FormClient : Form
    {
        private bool isExit = false;
        private delegate void SetListBoxCallback(string str);
        private SetListBoxCallback setListBoxCallback;
        private TcpClient client;
        private BinaryReader br;
        private BinaryWriter bw;
        //对称加密
        private TripleDESCryptoServiceProvider tdes;
        //不对称加密
        private RSACryptoServiceProvider rsa;
        public FormClient()
        {
            InitializeComponent();
            listBoxStatus.HorizontalScrollbar = true;
            setListBoxCallback = new SetListBoxCallback(SetListBox);
        }
```

```
//单击连接服务器按钮触发的事件
private void buttonConnect_Click(object sender, EventArgs e)
{
    try
    {
        //实际使用时要将 Dns.GetHostName()改为服务器域名
        client = new TcpClient(Dns.GetHostName(), 51888);
        SetListBox(string.Format("本机 EndPoint：{0}", client.Client.LocalEndPoint));
        SetListBox("与服务器建立连接成功");
    }
    catch
    {
        SetListBox("与服务器连接失败");
        return;
    }
    buttonConnect.Enabled = false;
    //获取网络流
    NetworkStream networkStream = client.GetStream();
    //将网络流作为二进制读写对象
    br = new BinaryReader(networkStream);
    bw = new BinaryWriter(networkStream);
    Thread threadReceive = new Thread(new ThreadStart(ReceiveData));
    threadReceive.Start();
    //使用默认密钥创建对称加密对象
    tdes = new TripleDESCryptoServiceProvider();
    //使用默认密钥创建不对称加密对象
    rsa = new RSACryptoServiceProvider();
    //导出不对称加密密钥的 xml 表示形式，false 表示不包括私钥
    string rsaPublicKey = rsa.ToXmlString(false);
    //将导出的公钥发送到服务器，公钥可以对任何人公开
    SendData("rsaPublicKey,true", Encoding.Default.GetBytes(rsaPublicKey));
}
//接收线程
private void ReceiveData()
{
    while (isExit == false)
    {
        //保存接收的命令字符串
        string receiveString = null;
        //解析命令用
        //每条命令均带有一个参数，值为 true 或者 false，表示是否有紧跟的字节数组
        string[] splitString = null;
        byte[] receiveBytes = null;
        try
        {
            //从网络流中读出命令字符串
            //此方法会自动判断字符串长度前缀，并根据长度前缀读出字符串
            receiveString = br.ReadString();
            splitString = receiveString.Split(',');
            if (splitString[1] == "true")
            {
```

```
                    //先从网络流中读出 32 位的长度前缀
                    int bytesLength = br.ReadInt32();
                    //然后读出指定长度的内容保存到字节数组中
                    receiveBytes = br.ReadBytes(bytesLength);
                }
            }
            catch
            {
                //底层套接字不存在时会出现异常
                SetListBox("接收数据失败");
            }
            if (receiveString == null)
            {
                if (isExit == false)
                {
                    MessageBox.Show("与服务器失去联系！");
                }
                break;
            }
            SetListBox("收到：" + receiveString);
            if (receiveBytes != null)
            {
                SetListBox(string.Format("收到：{0}",
                    Encoding.Default.GetString(receiveBytes)));
            }
            switch (splitString[0])
            {
                case "Talk":
                    //解密
                    string talkString = DecryptText(receiveBytes, tdes.Key, tdes.IV);
                    if (talkString != null)
                    {
                        SetListBox(string.Format("服务器说：{0}", talkString));
                    }
                    break;
                case "tdesKey":
                    //解密
                    tdes.Key = rsa.Decrypt(receiveBytes, false);
                    break;
                case "tdesIV":
                    //解密
                    tdes.IV = rsa.Decrypt(receiveBytes, false);
                    break;
                default:
                    SetListBox("什么意思啊：" + receiveString);
                    break;
            }
        }
        Application.Exit();
    }
    //使用对称加密加密字符串
```

```
private byte[] EncryptText(string str, byte[] Key, byte[] IV)
{
    //创建一个内存流
    MemoryStream memoryStream = new MemoryStream();
    //使用传递的私钥和 IV 创建加密流
    CryptoStream cryptoStream = new CryptoStream(memoryStream,
        new TripleDESCryptoServiceProvider().CreateEncryptor(Key, IV),
        CryptoStreamMode.Write);
    //将传递的字符串转换为字节数组
    byte[] toEncrypt = Encoding.UTF8.GetBytes(str);
    try
    {
        //将字节数组写入加密流,并清除缓冲区
        cryptoStream.Write(toEncrypt, 0, toEncrypt.Length);
        cryptoStream.FlushFinalBlock();
        //得到加密后的字节数组
        byte[] encryptedBytes = memoryStream.ToArray();
        return encryptedBytes;
    }
    catch (CryptographicException err)
    {
        SetListBox("加密出错：" + err.Message);
        return null;
    }
    finally
    {
        cryptoStream.Close();
        memoryStream.Close();
    }
}
//使用对称加密算法解密接收的字符串
private string DecryptText(byte[] dataBytes, byte[] Key, byte[] IV)
{
    //根据加密后的字节数组创建一个内存流
    MemoryStream memoryStream = new MemoryStream(dataBytes);
    //使用传递的私钥、IV 和内存流创建解密流
    CryptoStream cryptoStream = new CryptoStream(memoryStream,
        new TripleDESCryptoServiceProvider().CreateDecryptor(Key, IV),
        CryptoStreamMode.Read);
    //创建一个字节数组保存解密后的数据
    byte[] decryptBytes = new byte[dataBytes.Length];
    try
    {
        //从解密流中将解密后的数据读到字节数组中
        cryptoStream.Read(decryptBytes, 0, decryptBytes.Length);
        //得到解密后的字符串
        string decryptedString = Encoding.UTF8.GetString(decryptBytes);
        return decryptedString;
    }
    catch (CryptographicException err)
    {
```

```
            SetListBox("解密出错：" + err.Message);
            return null;
        }
        finally
        {
            cryptoStream.Close();
            memoryStream.Close();
        }
    }
    //发送信息到服务器
    private void SendData(string command, byte[] bytes)
    {
        //每条命令均带有一个参数，值为 true 或者 false，表示是否有紧跟的字节数组
        //如果不带参数，也可以实现，但是会导致接收方判断代码复杂化
        string[] splitCommand = command.Split(',');
        try
        {
            //先将命令字符串写入网络流，此方法会自动附加字符串长度前缀
            bw.Write(command);
            SetListBox(string.Format("发送：{0}", command));
            if (splitCommand[1] == "true")
            {
                //先将字节数组的长度（32 位整数）写入网络流
                bw.Write(bytes.Length);
                //然后将字节数组写入网络流
                bw.Write(bytes);
                bw.Flush();
                SetListBox(string.Format("发送：{0}", Encoding.UTF8.GetString(bytes)));
                if (splitCommand[0] == "Talk")
                {
                    SetListBox("加密前内容：" + textBoxSend.Text);
                }
            }
        }
        catch
        {
            SetListBox("发送失败!");
        }
    }
    private void SetListBox(string str)
    {
        if (listBoxStatus.InvokeRequired == true)
        {
            this.Invoke(setListBoxCallback, str);
        }
        else
        {
            listBoxStatus.Items.Add(str);
            listBoxStatus.SelectedIndex = listBoxStatus.Items.Count - 1;
            listBoxStatus.ClearSelected();
        }
```

```
        }
        //单击发送按钮触发的事件
        private void buttonSend_Click(object sender, EventArgs e)
        {
            //加密 textBoxSend.Text 的内容
            byte[] encryptedBytes = EncryptText(textBoxSend.Text, tdes.Key, tdes.IV);
            if (encryptedBytes != null)
            {
                SendData("Talk,true", encryptedBytes);
                textBoxSend.Clear();
            }
        }
        private void FormClient_FormClosing(object sender, FormClosingEventArgs e)
        {
            //未与服务器连接前 client 为 null
            if (client != null)
            {
                SendData("Logout,false", null);
                isExit = true;
                br.Close();
                bw.Close();
                client.Close();
            }
        }
        //textBoxSend 获得焦点并释放按键后触发的事件
        private void textBoxSend_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)Keys.Return)
            {
                buttonSend_Click(null, null);
            }
        }
    }
}
```

(3) 同时执行服务器端程序和客户端程序，运行效果如图 6-8 所示。

图6-8　例6-3的运行效果

6.4 Hash 算法与数字签名

通过 Internet 下载文件后，怎样知道下载的文件是否和原始文件完全相同呢？或者说，发送方通过 Internet 发送数据后，接收方如何验证接收的数据是否和原始数据完全相同呢？这就是数字签名的用途。

数字签名是利用不对称加密和 Hash 算法共同实现的。为了真正理解数字签名的实现原理，还必须简单介绍一下 Hash 算法。

Hash 算法也叫散列算法，其功能是把任意长度的二进制值映射为较小的固定长度的二进制值，实现原理就是提供一种数据内容和数据存放地址之间的映射关系。利用 Hash 算法得到的这个固定长度的较小的二进制值叫 Hash 值。

Hash 算法具有如下特点：

1) 散列效果好。即使原始数据只发生一个小小的改动，数据的散列也会发生非常大的变化。假如两个单词非常相似，比如只有一个字母不同，使用 Hash 算法得到的结果也相差甚远。甚至根本看不出二者之间有什么相似之处。

2) 散列函数不可逆。即不可能从散列结果推导出原始数据。

3) 对不同的数据进行 Hash 运算不可能生成相同的 Hash 值。

Hash 算法的用途主要有两大类：一类是将 Hash 值作为消息身份验证代码（MAC，Message Authentication Code），用于和数字签名一起实现对消息数据进行身份验证；另一类是将 Hash 值作为消息检测代码（MDC，Message Detection Code），用于检测数据完整性。

在应用程序中，可以利用数字签名实现数据身份验证和数据完整性验证。数据身份验证是为了验证数据是不是持有私钥的人发送的；数据完整性验证则用于验证数据在传输过程中是否被修改过。

验证数据完整性的实现原理是：发送方先使用 Hash 算法对数据进行 Hash 运算得到数据的 Hash 值，然后将数据和 Hash 值一块儿发送给接收方；接收方接收到数据和 Hash 值后，对接收的数据进行和发送方相同的 Hash 运算，然后将计算得到的 Hash 值和接收的 Hash 值进行比较，如果二者一致，说明收到的数据肯定与发送方发送的原始数据相同，从而说明数据是完整的。

.NET Framework 提供以下实现数字签名的类：

- DSACryptoServiceProvider
- RSACryptoServiceProvider

可见，这两个类既能实现加密解密数据的功能，也能实现数字签名的功能。具体使用方法并不复杂，这里不再赘述。

保证数据在网络传递中的安全性和完整性，涉及的技术很多，本章只是简单地介绍了一下相关的知识，更深入的内容需要读者自己研究。从技术选择上，主要考虑以下情况：

1) 如果需要使用一种方法验证数据在传输过程中是否被修改，可以使用 Hash 值。

2) 如果要证明实体知道机密但不相互发送机密，或者想使用简单的 Hash 值以防止在传输过程中被截获，可以使用加密的 Hash 值。

3) 如果要隐藏通过不安全的媒介发送的数据或者要永久性保留数据，可以使用加密。

4) 如果要验证声称是公钥所有者的人员的身份，可以使用证书。

5) 如果双方事先均知道准备使用的密钥，可以使用对称加密以提高速度。

6) 如果想通过不安全的媒介安全地交换数据，可以使用非对称加密。

7) 如果要进行身份验证和实现不可否认性，可以使用数字签名。

8) 如果为了防范穷举搜索而进行的攻击，可以使用加密技术生成的随机数。

设计一个安全系统时，应该根据安全第一、性能第二的原则来选择实现的技术。在.NET 环境下，对数据进行加密，可以解决数据的保密性，完整性和身份验证等重要安全问题。对称加密和不对称加密各有优缺点，一般情况下，要将两种加密方法结合使用，这样可以提高数据的保密性和数据传输效率。.NET 提供的 HASH 算法类主要用于数字签名，利用.NET 提供的数字签名技术，接收者可以核实发送者对报文的签名，防止发送者抵赖对数据的签名，同时也避免了攻击者伪造对报文的签名。

6.5 习题 6

1. 简要回答对称加密和不对称加密各有哪些特点。

2. 如果有两个内容相同的原始数据块，使用.NET 提供的对称加密类得到的加密后的两个数据块内容相同吗？

3. 数字签名有什么意义？简要回答如何实现数字签名？

第 7 章　三维设计与多媒体编程

目前流行的各种大型网络游戏，以及各种高端的图形图像应用设计，大部分都使用了 3D 技术。流行的三维设计编程接口（API）有两大类，一类是 OpenGL，主要用于 Java 编程；另一类是 DirectX，主要用于 Windows 环境下的各类三维软件编程。

DirectX 其实是一个组件套件，是由微软公司开发的应用程序编程接口，包含有 Direct Graphics、Direct Input、Direct Play（针对游戏/网络）、Direct Sound（针对 3D 声音功能）、Direct Show 等多个组件，它提供了一整套的多媒体接口方案。而今已发展成为对整个多媒体系统的各个方面都有决定性影响的接口。

DirectX 是 1995 年诞生的，1996 年在 DirectX 中加入了 Direct3D，用于访问 3D 硬件加速的高级图形功能，通过提供通用的 COM（Component Object Model，组件对象模型）编程接口使硬件和设计分离，随后很快发展到 DirectX8.0，并从 DirectX9.0 开始支持托管代码。自 2002

年.NET 和 C#诞生以后，DirectX 的应用也迅速得到提升，很多优秀的性能也逐步加入到托管 DirectX 中。由于 DirectX 的使用非常普遍，所以目前所有的显卡无一例外的都支持 DirectX 编程接口，给各类三维应用设计带来了很大方便。

注意，本章的所有例子全部是在安装 DirectX9.0c SDK December 2005 后调试通过的。要调试本章的所有例子，必须先安装 VS2005，然后再安装 DirectX9.0 SDK December 2005（几百 MB），如果仅仅为了直接运行已经编译后的.exe 文件，而不进行调试，也可以只安装 DirectX9.0c（几十 MB）。例子中调试使用的具体版本为：

Microsoft.DirectX.dll 版本 1.0.2902.0
Microsoft.DirectX.Direct3D.dll 版本 1.0.2902.0
Microsoft.DirectX.Direct3DX.dll 版本 1.0.2908.0

使用 DirectX 编写程序，首先必须在资源管理器中添加对它的引用。步骤为：在解决方案资源管理器中，鼠标右键单击【引用】→【添加引用】，然后在弹出窗口的.NET 选项中选择上面列出的版本，单击【确定】。选择时要仔细观察引用的是哪个版本，不要选错。一定要注意，本章的例子必须在引用上面列出的版本下才能正常调试，在低版本下或者高版本下，由于 SDK 提供的属性及方法不一定相同，所以不保证能正常运行。

由于三维设计涉及面很广，而且包含很多复杂的模型计算以及数学处理等方面的内容，因此这里不准备在短短一章中把 Direct3D 的内容全部介绍，而是只关心简单的、最基本的三维设计思路，并通过介绍 DirectX 提供的 Primitive 和 Mesh 对象的基本用法，为复杂的三维设计入门起到一个抛砖引玉的作用。

7.1 简单的 3D 设计入门

与一般 Windows 应用程序不同，为了提高显示速率和展现真实的三维效果，三维设计有一个特定的基本编程模式。为了让读者首先有一个感性认识，这一节先不介绍 DirectX 提供的各种功能，而是通过一个简单例子介绍利用 DirectX 编程接口进行三维设计的一般方法，然后再介绍涉及到的相关概念和技术。

【例 7-1】使用 DirectX 设计一个能够旋转的三角形，并将本机显卡和显示器的相关参数显示出来。

(1) 新建一个名为 SimpleDirect3DExample 的 Windows 应用程序，使用默认的窗体名称。

(2) 在【解决方案资源管理器】中，鼠标右键单击【引用】→【添加引用】→在【.net】选项下同时选中 Microsoft.DirectX、Microsoft.DirectX.Direct3D 和 Microsoft.DirectX.Direct3DX，注意选择的版本要和本章开头列出的版本一致，单击【确定】。

(3) 在命名空间的上方，添加引用代码：

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

(4) 在构造函数上方添加字段声明：

```
private Device device = null;       //Device 指显卡适配器，一个显卡至少有一个适配器
private VertexBuffer vertexBuffer = null;
private Microsoft.DirectX.Direct3D.Font d3dfont;
private string adapterInformationString;
private bool showAdapterString = true;
private float angle = 0.0f;
```

```
private float incrementAngle = 0.1f;
private bool enableRotator = true;
```

(5) 直接添加代码：

```
public bool InitializeGraphics()
{
    try
    {
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed = true;
        presentParams.SwapEffect = SwapEffect.Discard;
        device = new Device(0, DeviceType.Hardware, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
        return true;
    }
    catch (DirectXException)
    {
        return false;
    }
}
```

这段代码中，首先在 Try 中创建一个代表显卡适配器的 Device 对象，如果显卡驱动程序正确并支持 3D，就不会出现异常。然后就利用这个 Device 对象和显存交换信息，也可以通过这个对象获取或设置显卡适配器的相关配置信息。

顺便提一下，由于目前很多显卡都支持多显示器，即一块显卡可以连接一个以上的显示器，这就相当于一块显卡有多个适配器。在编写 3D 程序时，要支持多显示器，就必须为同一块显卡的每一个适配器各创建一个 Device。在本章的所有例子中，均假定只有一个显卡适配器，所以只创建了一个 Device。

(6) 在窗体的 Load 事件中添加代码：

```
private void Form1_Load(object sender, EventArgs e)
{
    //设置窗体显示方式
    this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque, true);
    //为了能响应键盘事件，该属性必须设为 true
    this.KeyPreview = true;
    adapterInformationString = "F1：显示/隐藏提示信息\n" +
        "<F2>：旋转/不旋转\n" +
        "上箭头：提高转速\n" +
        "下箭头：降低转速\n" +
        "<Esc>：退出\n\n\n";
    AdapterDetails adapterDetails = Manager.Adapters.Default.Information;
    adapterInformationString += string.Format(
        "显卡驱动程序：{0}\n", adapterDetails.DriverName);
    adapterInformationString += string.Format(
        "显卡驱动程序版本：{0}\n", adapterDetails.DriverVersion);
    DisplayMode displayMode = Manager.Adapters.Default.CurrentDisplayMode;
    adapterInformationString += string.Format(
        "显示器当前分辨率：{0} X {1}\n", displayMode.Width, displayMode.Height);
    adapterInformationString += string.Format(
        "显示器当前颜色质量：{0}\n", displayMode.Format);
```

```
            adapterInformationString += string.Format(
                "显示器当前刷新频率（Hz）：{0}\n", displayMode.RefreshRate);
            //创建 3D 字体对象，显示字符用,这两句不能放在场景中，否则会很慢
            System.Drawing.Font winFont = new System.Drawing.Font("Arial", 9, FontStyle.Regular);
            d3dfont = new Microsoft.DirectX.Direct3D.Font(device, winFont);
            d3dfont.PreloadText(adapterInformationString);
            //创建顶点缓冲
            vertexBuffer = new VertexBuffer(typeof(CustomVertex.PositionColored),
                3, device, Usage.Dynamic | Usage.WriteOnly,
                CustomVertex.PositionColored.Format, Pool.Default);
            vertexBuffer.Created += new EventHandler(OnVertexBufferCreate);
            OnVertexBufferCreate(vertexBuffer, null);
        }
```

代码中首先设置窗体样式，目的是为了重绘窗体时能得到连续变化的窗体图像；然后利用 Manager 命名空间下提供的类判断显卡和显示器相关的配置参数；第三步是定义一个三维字体对象，以便显示配置信息；最后在显存中创建了可以容纳 3 个顶点的顶点缓冲，并添加了 OnVertexBufferCreate 事件，该事件在每次重建顶点缓冲时都会自动触发，但是第一次需要直接调用。

为什么必须通过事件实现呢？这是因为在运行时，如果改变窗口的大小，设备就会自动重置。当创建的资源位于默认的内存池时（比如顶点缓冲），重置设备会释放缓冲。所以当改变窗口大小的时候，重置了 device，释放了顶点缓冲。虽然 Managed DirectX 在重置 device 之后会自动的重建顶点缓冲。但是，由于原来的顶点缓冲已经被释放，所以重建的顶点缓冲中已经没有了数据，也就无法将原来的内容重新绘制出来。

在重建顶点缓冲并准备好填充数据时，会自动触发顶点缓冲的 Created 事件，所以可以利用这个事件，让其重建缓冲时自动调用 OnVertexBufferCreate 重建顶点数据。这样就能看到连续的图像了。

(7) 直接添加代码：

```
        private void OnVertexBufferCreate(object sender, EventArgs e)
        {
            //锁定顶点缓冲-->定义顶点-->解除锁定。
            VertexBuffer buffer = (VertexBuffer)sender;
            CustomVertex.PositionColored[] verts = (CustomVertex.PositionColored[])buffer.Lock(0, 0);
            verts[0].Position = new Vector3(0.0f, 1.0f, 1.0f);
            verts[0].Color = Color.BlueViolet.ToArgb();
            verts[1].Position = new Vector3(-1.0f, -1.0f, 1.0f);
            verts[1].Color = Color.GreenYellow.ToArgb();
            verts[2].Position = new Vector3(1.0f, -1.0f, 1.0f);
            verts[2].Color = Color.Red.ToArgb();
            buffer.Unlock();
        }
```

在这个事件中，首先通过 VertexBuffer 对象的 Lock 方法锁定顶点缓冲，然后在顶点缓冲中构造一个三角形，三角形的三个顶点分别定义了不同的颜色，构造完成后再调用 Unlock 方法解除顶点缓冲。

每次顶点缓冲需要重新构造其数据时，系统都会自动触发这个事件。

(8) 添加设置矩阵参数的代码：

```
private void SetupCamera()
{
    //--------设置世界矩阵
    Vector3 world = new Vector3(angle, angle / 2.0f, angle / 4.0f);
    device.Transform.World = Matrix.RotationAxis(world, angle);
    if (enableRotator)
    {
        angle += incrementAngle / (float)(Math.PI);
    }
    //--------设置投影矩阵
    //纵横比
    float aspectRatio = 1;
    //只能显示 nearPlane 到 farPlane 之间的场景
    float nearPlane = 1;
    float farPlane = 100;
    //视界
    float fieldOfView = (float)Math.PI / 4.0f;
    device.Transform.Projection = Matrix.PerspectiveFovLH(fieldOfView, aspectRatio, nearPlane,
farPlane);
    //--------设置视图矩阵
    Vector3 cameraPosition = new Vector3(0, 0, -5);
    Vector3 cameraTarget = new Vector3(0, 0, 0);
    Vector3 upDirection = new Vector3(0, 1, 0);
    device.Transform.View = Matrix.LookAtLH(cameraPosition, cameraTarget, upDirection);
    //--------不进行背面剔除
    device.RenderState.CullMode = Cull.None;
    //--------不要灯光
    device.RenderState.Lighting = false;
}
```

这段代码中用到了世界矩阵、投影矩阵和视图矩阵，不进行背面剔除，不使用灯光。

(9) 在窗体的 Paint 事件中添加代码：

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    device.Clear(ClearFlags.Target, System.Drawing.Color.AliceBlue, 1.0f, 0);
    SetupCamera();
    //--------------场景处理
    device.BeginScene();
    device.VertexFormat = CustomVertex.PositionColored.Format;
    device.SetStreamSource(0, vertexBuffer, 0);
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
    if (showAdapterString == true)
    {
        d3dfont.DrawText(null, adapterInformationString, 25, 30, Color.Green);
    }
    device.EndScene();
    //---------------发送场景
    device.Present();
    //---------------强制重新调用 Form1_Paint 事件
    if (WindowState != FormWindowState.Minimized)
    {
        this.Invalidate();
```

```
        }
    }
```

在介绍 GDI+时，我们已经知道很多图形图像操作都是在窗体的 Paint 事件中处理的，在三维设计中，一样可以让所有操作都在 Paint 事件中处理。和 GDI+编程不同，三维编程不是直接在屏幕坐标上绘制，而是和录像相似，首先要确定拍摄对象，设置好摄像机位置，选择拍摄的场景视界，然后才可以录制场景信息，此时在显示器上观察到的内容就相当于拍摄场景时在摄像机中观察到的内容。与实际拍摄不同的是显示的场景是我们自己设计的。实际上，在摄像机中看到的场景就是向顶点缓冲发送的内容。

在 Paint 事件中，使用 Clear 方法清除窗口的所有内容，并将其填充为实心的颜色。该方法的第一个参数指定了要填充的对象，第二个参数是要填充的颜色。

例子中使用了 DrawPrimitives 方法绘制场景，DrawPrimitives 方法有三个参数，第一个参数是要绘制的基本图形的类型，以便知道每个基本图形需要几个顶点，例子中指定绘制的类型为三角形列，这样它就知道每个三角形需要三个顶点来绘制；第二个参数是从哪个顶点开始绘制，这个例子指定从第一个顶点开始（索引号为 0）；第三个参数是要绘制的基本图形的数量，由于本例中只画一个三角形，所以设为 1。

设置场景后，必须使用 Present 方法更新显示。

调用 EndSence 方法的目的是通知 Direct3D 不再绘图了。每次调用 BeginSence 之后，在结束绘图前都必须调用这个方法。

这个事件的最后一句调用了 Invalidate 方法，这是三角形旋转后更新显示的关键。每个场景发送后，通过 Invalidate 方法使其不停地触发 Paint 事件，从而可以周而复始的重绘窗体。再利用不断变换角度，完成连续旋转的目的。

还有一个关键的地方，在默认模式下，Invalidate 方法只会重新绘制代码中指定的部分，例子中使用的是 this，即告诉系统只需要重新绘制当前窗体部分就行了。为了提高绘制效率，它不会重新处理其他的绘制过程。因此在默认模式下，看到的效果是闪烁的、不连续的。为了解决这个问题，必须修改默认模式，即在 Form1 的 Load 事件中添加：

```
this.SetStyle(ControlStyles.AllPaintingInWmPaint | ConstolStyles.Opaque, true);
```

这样就得到希望的效果了。

(10) 添加窗体的 KeyDown 事件代码：

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    switch (e.KeyCode)
    {
        case Keys.Escape:
            this.Close();
            break;
        case Keys.F1:
            showAdapterString = !showAdapterString;
            break;
        case Keys.F2:
            enableRotator = !enableRotator;
            break;
        case Keys.Up:
            if (enableRotator)
            {
```

```
                incrementAngle += 0.01f;
            }
            break;
        case Keys.Down:
            if (enableRotator && incrementAngle > 0.02f)
            {
                incrementAngle -= 0.01f;
            }
            break;
    }
}
```

(11) 修改 Program.cs 中的 Main 方法为如下代码：

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Form1 frm = new Form1();
    if (frm.InitializeGraphics() == false)
    {
        MessageBox.Show("显卡不支持 3D 或者未安装配套的显卡驱动程序！");
        return;
    }
    Application.Run(frm);
}
```

(12) 按<F5>键编译并运行，一个漂亮的不停旋转的三角形就展现出来了。运行效果如图 7-1 所示。鼠标拖动右下角改变一下窗体的大小，观察三角形有什么变化。

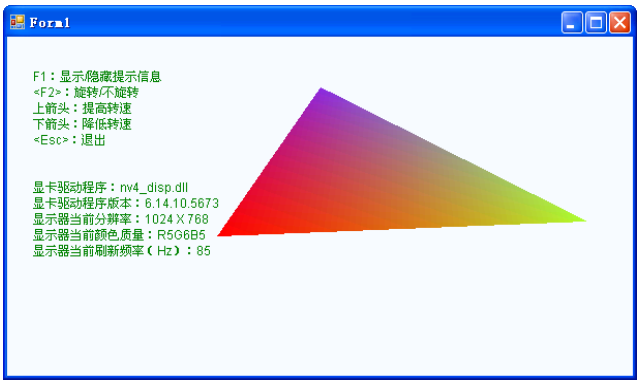


图7-1　例7-1的运行效果

观察运行的程序界面，可以看到显示器颜色位数显示的是一个字符串，字符串的字母和数字交替出现。字母表示了数据的类型，数字表示这种类型的数据所占的位数。常见字母有：

A—alpha、B—blue、X—unused、L—luminance、R—red、P—palette、G—green

每种类型所占的位数加起来，就是这种格式的总大小。比如 X8R8G8B8，就是 32 位的格式，红、绿、蓝各占 8 位，还有 8 位没有使用。

我们可能已经从这个例子中感觉到了，三维设计涉及到的新概念比较多，学习 3D 技术的关键是要先搞清楚这些常用的概念及其用法，否则的话，就很难看懂代码，或者说很难入门，自然也就不知道编写 3D 程序时应该如何下手了。

7.2 DirectX 基础知识

从例 7-1 中可以看出，理解三维设计中涉及的概念是非常重要的。如果对这些概念似懂非懂，即使花费再多的时间，也很难真正掌握实际的设计方法，进一步提高编写三维程序的能力也就成了一句空话。

本节从数学和 Direct3D 相结合的角度，介绍需要掌握的基本知识。

7.2.1 左手坐标系与右手坐标系

3D 系统中的每一个点都由 x，y，z 三个值组成。一般情况下，x 轴的正方向向右，y 轴的正方向向上。然后根据 z 轴的正方向划分为两种坐标系，如果 z 轴正方向为远离观察者而去的方向，即朝着屏幕里面的方向，这种坐标系叫左手坐标系，反之叫做右手坐标系。

Direct3D 默认使用左手坐标系。

在 Direct3D 中，用 Vector3 表示一个三维向量，例如：

```
Vector3 a = new Vector3(1, 2, 2);
```

7.2.2 设备

在 Direct3D 中，设备（Device）是指显卡提供的接口，可以把这个类假想为显卡中的某个适配器。场景的所有图形对象都依赖于 Device 对象。一台计算机中至少有一个 Device 对象，在 Managed Direct3D 里，可以控制任意多个 Device 对象。所有三维图形图像等信息均可以通过 Device 对象处理，该类常用的构造函数为：

```
public Device(int adapter, DeviceType deviceType, Control renderWindow,
    CreateFlags behaviorFlags, params PresentParameters[] presentationParameters);
```

各参数的含义如下：

adapter：表示使用显卡的第几个适配器，一般使用第一个（索引号为 0）。

deviceType：告诉 Direct3D 要创建哪种类型的 Device。一般用 DeviceType.Hardware；另一个选项是 DeviceType.Reference，即使用参考光栅器（reference rasterizer），此时所有的效果由 Direct3D 运行时来实现，这个选项会使运行速度慢得多，一般仅在显卡不支持 3D 时使用这个选项。注意：参考光栅器只在 DirectX SDK 里提供，而 DirectX 运行时是不提供这个特性的；第三个选项是 DeviceType.Software，指使用自定义的软件光栅器，一般不用。

renderWindow：表示设备绑定到的对象窗口。一般使用当前窗口。

behaviorFlags：描述创建 device 之后处理顶点的方式。常用方式有两种，一种是使用 HardwareVertexProcessing，即使用显卡自带的图形处理器（GPU，Graphic Processing Unit），使用这个选项处理速度非常快，不过这类显卡价格较贵，一般的显卡没有这个功能；另一种是使用 SoftwareVertexProcessing，即利用本机内存模拟显卡图形处理功能，这个选项对显卡的要求不高，使用一般的显卡即可，但是执行速度要慢一些。

在本章 Direct3D 的例子中，全部使用 SoftwareVertexProcessing 选项，以便只有一般显卡的计算机也能正常运行 3D 程序。

presentationParameters：表示设备把数据呈现到显示器的方式。这个选项比较多，目前我们只关心 Windowed 属性和 SwapEffect 属性。

1) Windowed 属性：呈现形式是全屏还是窗口模式。

2) SwapEffect 属性：控制显存中缓冲区处理的方式。常用的是 SwapEffect.Discard，即如果上次要显示的缓冲区数据还没有处理完毕，就自动丢弃，并直接填充新内容。

另一个常用的方式是 SwapEffect.Flip，运行时会创建一个后缓冲区，在显示场景时采用前后缓冲区快速翻动方式。

为了能理解前后缓冲区的作用，我们先看看电影播放的原理，电影本来是一幅幅图片组成的，当它以每秒 24 幅的速度连续播放时，如果每幅图像之间的差别很小，由于人眼的滞留作用，人们看到的画面就是连续的。DirectX 的工作原理与此类似。可以事先定义两个缓冲区页面，一个叫前缓冲区页面，用于显示，另一个叫后缓冲区页面，用于在显示前缓冲区的同时填充下一个将要显示的内容。当把前缓冲区的页面内容送显示器时，系统同时将下一幅要显示的内容绘制到后缓冲区页面中，然后快速的把它翻动到前缓冲区，并重复这个过程。这样在显示器上看到的就是连续的图像了。与电影不同的是，计算机每秒能绘制的页面数要远远大于 24 幅。

所以，程序需要不停地循环填充后缓冲区的内容，每次循环都要清除后缓冲区，并把应该绘制的内容绘制到该缓冲区，然后由系统自动把它翻动到前缓冲区，再进入下一次循环，直到结束。

7.2.3 顶点与顶点缓冲

在数学上，一般只需要描述点的位置，但是在 Direct3D 中，除了点的位置外，还需要保存该点的颜色以及法线（Normal）、纹理（Texture）等其他信息，为了与数学上一般的点区分，Direct3D 把坐标系中的每一个点称之为顶点（Vertex）。

顶点缓冲指在显存或内存中开辟的用于保存顶点的缓冲区。顶点缓冲可以保存任何的顶点类型。将顶点保存在顶点缓冲区中后，显卡就可以根据提供的参数对其进行各种变换、绘制等操作。

创建顶点缓冲的常用形式为：

```
public VertexBuffer( Type vertexType, int numVerts, Device device, Usage usage,
    VertexFormats vertexFormat, Pool pool);
```

其中：

vertexType：指定要创建的顶点类型。该值可以是 CustomVertex 类中的顶点结构类型，也可以是自定义的顶点类型；

numVerts：顶点个数。

device：使用的显卡对象。

usage：如何使用顶点缓冲。

vertexFormat：定义储存在顶点缓冲中的顶点格式。

pool：定位顶点缓冲使用的内存池位置。

7.2.4 Mesh 对象

有了顶点，再通过顶点之间封闭的连线，就可以构成面了。任何一个物体，无论表面多么平滑，其视图都可以用三角形构成的面组合得到。如果只画出三角形的连线，就可以看出一个物体的框架是由许多个三角形构成的网格组成，Direct3D 把由三角形网格组成的对象叫做

Mesh 对象。

7.2.5 法线

在数学上，法线（Normal）指垂直于面的向量。在 Direct3D 中，每个顶点也都可以有对应的法线，而且还可以设置每个顶点法线的方向。顶点法线和面法线对计算光照以及着色模式（Shading mode）非常重要。面法线一般用于 Flat 着色，得到物体的立体感效果；而顶点法线则一般用于 Gouraud 着色，用于控制光照及纹理，使物体看起来很圆滑。

7.2.6 纹理与纹理映射

在 20 世纪 80 年代和 90 年代初的 3D 程序中，很多物体的外表看起来都是发亮的塑胶质感的样子，感觉很不真实。在 Direct3D 中，使用纹理（Texture）可以轻松地解决这个问题，甚至物体很小的细节部分，例如子弹孔、不规则表面、生锈的部分、裂缝、指纹、钉在木板上的图钉等等都可以通过纹理实现。通过纹理，还可以将一幅图贴在几何体的表面提供复杂的视觉效果。从一般概念上讲，纹理指由各种色彩的像素构成的简单位图。但是在 Direct3D 中，纹理则指可以映射到一个或者多个面上的特殊的图像。把纹理映射到目标几何体上的过程叫纹理映射（Texture-Mapping）。

因为纹理是二维的，所以仅需要两个坐标值：u 和 v。u 是横向坐标值，v 是纵向坐标值，u 和 v 的取值应该在 0 和 1 之间，左上角是（0，0）,右下角是（1，1）。

7.2.7 世界矩阵、投影矩阵与视图矩阵

矩阵是由数字组成的矩形数组，矩阵中的每个元素都有一个数字或表达式。通过矩阵的加减乘除运算，可以对 3D 对象中的顶点进行移动、旋转、缩放等操作。

在 DirectX 中，每个矩阵都由 4 行 4 列组成。构造一个向量前，必须先构造一个变换矩阵。将一个 4×4 的矩阵乘以一个表示 3D 数据的向量，比如与一个顶点向量相乘，即得到新的变换后的向量。

如果需要进行一系列变换，可以为每一个动作（缩放、旋转、平移）各创建一个临时矩阵，然后对向量分别进行相应的变换；还有一个办法就是先将这些矩阵相乘，然后把乘积的结果作为变换矩阵。

为了简化编程，DirectX 中已经分别提供了缩放、旋转、平移等方法，我们只需要调用这些方法得到变换后的向量即可。

完成既缩放又旋转这种复合变换，则需要两个矩阵：一个用来旋转，一个用来缩放；然后把两个矩阵相乘，得到一个新的复合矩阵；最后利用这个新的矩阵来变换顶点。

注意，矩阵相乘不满足交换律。

在 DirectX 中，有三种特殊类型的矩阵：世界矩阵（World Matrix）、视图矩阵（View Matrix）和投影矩阵（Projection Matrix）。

我们可以通过修改对象在世界矩阵中的坐标实现旋转、缩放或平移操作。所有这些变换都是相对于原点（0, 0, 0）的。单一的变换（例如平移或缩放等）可以组合起来，形成更复杂的变换；但要注意各种变换的顺序，矩阵 1 乘以矩阵 2 和矩阵 2 乘以矩阵 1 结果是不同的。

要旋转两个对象，一个关于 x 轴，一个关于 y 轴，则应该先完成 x 轴的变换，然后渲染第

一个对象；再完成 y 轴的变换，然后渲染第二个对象。

与屏幕坐标相比，可以把世界坐标设想为一个无限大的三维笛卡儿坐标。对象可以被放到这个“世界”的任意位置。

视图矩阵指摄像机拍摄的场景，或者说人们在摄像机前看到的图像。摄像机在世界空间中有一个位置，还有一个观察点。例如，可以把摄像机悬置于某个对象的上面（摄像机位置），把镜头对准那个对象的中心（观察点）。也可以指定哪面是上面，例如指定 y 轴的正方向是上面。

投影矩阵可以被想象成摄像机的镜头，该矩阵指定了观察区域和前、后裁剪平面。

投影变换定义了场景被怎样投影到显示器。最简单的产生投影矩阵的方法就是使用 Matrix 类的 PerspectiveFovLH 方法。即使用左手坐标系创建一个正对场景的透视投影变换。

投影变换描绘了场景的视界，即可见部分。视界是可视角度和前裁剪面（Near Plane）与后裁剪面（Far Plane）中间的一段区域，在这个区域内的即是可见部分。

例如图 7-2，摄像机位于“o”点，可见部分为 ABCD--A'B'C'D'，ABCD 为后裁剪面，A'B'C'D' 为前裁剪面。farPlane 就是锥体的底面，而 nearPlane 则是横截面。fieldOfView 参数描绘了锥体的角度。

aspectRatio 类似于电视的高宽比，比如，宽银幕电视的高宽比是 1:85。可以用可视区域的宽度来比上高度得出这个值。DirectX3D 只绘制在这个可见区域中的物体。

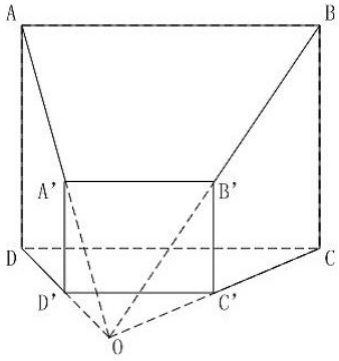


图7-2　前剪裁面与后剪裁面

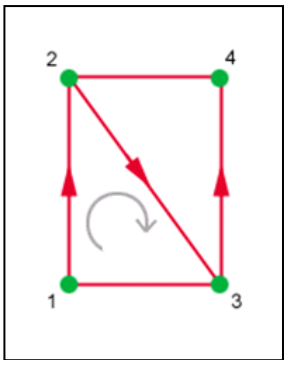


图7-3　三角形面

7.2.8　背面剔除

在介绍 Mesh 对象时已经提过，所有的几何形状都可以由三角形顶点构成的面组成。背面剔除（Backface Culling）的实际含义就是如何剔除每一个由顶点按照顺时针或逆时针方向依次连线构成的三角形面。剔除方式有三种：一种是不剔除，即全部渲染；第二种方式是剔除顺时针构成的面；第三种方式是剔除逆时针构成的面。

如果不指定剔除方式，Direct3D 默认剔除逆时针方向构成的面，即只渲染按顺时针方向排列的顶点构成的三角形面。

例如，对于图 7-3，如果采用逆时针剔除，则只显示 1、2、3 构成的三角形面；如果采用顺时针剔除，则只显示 2、3、4 构成的三角形面；如果不剔除，则全部显示。

背面剔除在 Direct3D 中有什么用呢？我们知道，当观察一个不透明的物体时，背面的部分是看不见的。系统在渲染物体的表面时，在正面是顺时针的三角形，旋转到背面时就变成了

逆时针的，因此也就不需要再渲染，即可以被剔除掉，从而节省了大量内存，提高了程序执行的效率。

7.3 Primitive

Primitive 指 Dirct3D 支持的最基本的图形，这些基本图形是构成其他各种复杂图形的基本元素，因此也叫图元。在 Direct3D 中，有六种 Primitive，分别是：点列、线列、线带、三角形列、三角形带和三角扇形。

PointList：点列。顶点序列中的每个顶点均单独绘制。

LineList：线列。绘制时顶点序列中的每两个顶点作为一组连成一条单独的直线，序列中的每个顶点只使用一次。

LineStrip：线带。绘制时从第一个顶点开始，依次将顶点序列中的各个顶点用直线连接起来。除第一个顶点和最后一个顶点只使用一次外，中间的其他顶点均使用两次。

TriangleList：三角形列。绘制时顶点序列中的每三个顶点作为一组组成一个单独的三角形，序列中的每个顶点只使用一次。

TriangleStrip：三角形带。绘制的结果为一系列相连的三角形，每两个相邻的三角形共享两个顶点。

TriangleFan：三角扇形。与三角形带相似，不过所有的三角形都共享一个顶点。

图 7-4 直观的画出了由六个顶点组成的各类图元的形式。

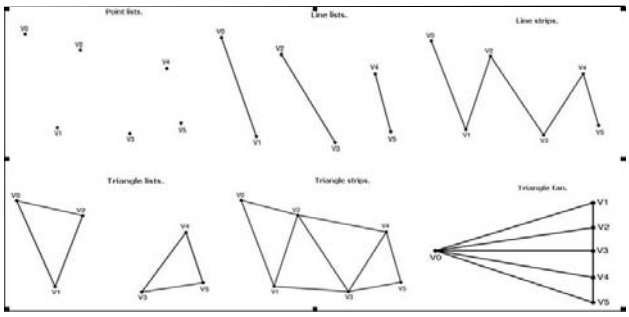


图7-4　Direct3D中的六种Primitive

【例 7-2】演示 Primitive 的绘制方法。

(1) 新建一个名为 PrimitiveExample 的 Windows 应用程序，使用默认的窗体名称。

(2) 在【解决方案资源管理器】中，鼠标右键单击【引用】→【添加引用】→在【.net】选项下同时选中 Microsoft.DirectX、Microsoft.DirectX.Direct3D 和 Microsoft.DirectX.Direct3DX，注意选择的版本要和本章开头列出的版本一致，单击【确定】。

(3) 在命名空间的上方，添加引用代码：

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

(4) 在构造函数上方添加字段声明：

```
private Device device = null;
private VertexBuffer vertexBuffer = null;
private const int vertexNumber=18;
private string primitivesType="点列";
```

```
//3D 字体，用于显示提示信息的字体
private Microsoft.DirectX.Direct3D.Font d3dfont;
//要显示的字符串
private string helpString;
//是否在窗体中显示字符串
private bool showHelpString = true;
private float angle = 0.0f;
private bool enableRotator = false;
private int rotatorXYZ = 0;
private bool enableCullMode=false;
```

在这个例子中，我们选择的顶点个数是 24，选择的原则是顶点的个数要同时能被 2 和 3 整除。以保证无论那种图元都能都被正确渲染。

(5) 直接添加代码：

```
public bool InitializeGraphics()
{
    try
    {
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed = true;
        presentParams.SwapEffect = SwapEffect.Discard;
        device = new Device(0, DeviceType.Hardware, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
        return true;
    }
    catch (DirectXException)
    {
        return false;
    }
}
private void SetupCamera()
{
    //设置投影矩阵
    float fieldOfView = (float)Math.PI / 4;
    float aspectRatio = this.Width/this.Height;
    float nearPlane = 1.0f;
    float farPlane = 100.0f;
    device.Transform.Projection =
        Matrix.PerspectiveFovLH(fieldOfView, aspectRatio, nearPlane, farPlane);
    //设置视图矩阵
    Vector3 cameraPosition = new Vector3(0, 0, -30.0f);
    Vector3 cameraTarget = new Vector3(0, 0, 0);
    Vector3 upDirection = new Vector3(0, 1, 0);
    device.Transform.View = Matrix.LookAtLH(cameraPosition, cameraTarget, upDirection);
    //点的大小
    device.RenderState.PointSize = 4.0f;
    //不要灯光
    device.RenderState.Lighting = false;
    //背面剔除
    if (enableCullMode == true)
    {
```

```
            device.RenderState.CullMode = Cull.CounterClockwise;
        }
        else
        {
            device.RenderState.CullMode = Cull.None;
        }
    }
    private void OnVertexBufferCreate(object sender, EventArgs e)
    {
        //锁定顶点缓冲-->定义顶点-->解除锁定。
        VertexBuffer buffer = (VertexBuffer)sender;
        CustomVertex.PositionColored[] verts = (CustomVertex.PositionColored[])buffer.Lock(0, 0);
        Random random = new Random();
        for (int i = 0; i < vertexNumber; i++)
        {
            float x = (float)(vertexNumber * (random.NextDouble() - 0.5f));
            float y = (float)(vertexNumber * (random.NextDouble() - 0.5f));
            float z = (float)(vertexNumber * (random.NextDouble() - 0.5f));
            verts[i].Position = new Vector3(x, y, z);
            Color color = Color.FromArgb(random.Next(byte.MaxValue),
                random.Next(byte.MaxValue), random.Next(byte.MaxValue));
            verts[i].Color = color.ToArgb();
        }
        buffer.Unlock();
    }
```

(6) 在窗体的 Load 事件中添加代码：

```
    private void Form1_Load(object sender, EventArgs e)
    {
        this.Width = 570;
        this.Height = 430;
        this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque, true);
        this.KeyPreview = true;
        helpString = "<Esc>：退出\n\n\n<F1>：显示/隐藏提示信息\n" +
            "<F2>：变换顶点\n" +
            "<F3>：旋转/不旋转\n" +
            "<F4>：旋转轴（x 轴、y 轴、z 轴）\n" +
            "<F5>：背面剔除/不剔除\n\n" +
            "<1>：点列\n" +
            "<2>：线列\n" +
            "<3>：线带\n" +
            "<4>：三角形列\n" +
            "<5>：三角形带\n" +
            "<6>：三角扇形\n";
        System.Drawing.Font winFont = new System.Drawing.Font("宋体", 9, FontStyle.Regular);
        d3dfont = new Microsoft.DirectX.Direct3D.Font(device, winFont);
        d3dfont.PreloadText(helpString);
        //创建顶点缓冲
        vertexBuffer = new VertexBuffer(typeof(CustomVertex.PositionColored),
            vertexNumber,
            device,Usage.Dynamic | Usage.WriteOnly,
            CustomVertex.PositionColored.Format,Pool.Default);
```

```
        vertexBuffer.Created += new EventHandler(OnVertexBufferCreate);
        OnVertexBufferCreate(vertexBuffer, null);
    }
```

(7) 在窗体的 Paint 事件中添加代码：

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        SetupCamera();
        device.Clear(ClearFlags.Target, System.Drawing.Color.AliceBlue, 1.0f, 0);
        device.BeginScene();
        device.VertexFormat = CustomVertex.PositionColored.Format;
        device.SetStreamSource(0, vertexBuffer, 0);
        switch (primitivesType)
        {
            case "点列":
                device.DrawPrimitives(PrimitiveType.PointList, 0, vertexNumber);
                break;
            case "线列":
                device.DrawPrimitives(PrimitiveType.LineList, 0, vertexNumber / 2);
                break;
            case "线带":
                device.DrawPrimitives(PrimitiveType.LineStrip, 0, vertexNumber - 2);
                break;
            case "三角形列":
                device.DrawPrimitives(PrimitiveType.TriangleList, 0, vertexNumber / 3);
                break;
            case "三角形带":
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, vertexNumber - 2);
                break;
            case "三角扇形":
                device.DrawPrimitives(PrimitiveType.TriangleFan, 0, vertexNumber - 2);
                break;
        }
        if (showHelpString == true)
        {
            d3dfont.DrawText(null, helpString, 25, 30, Color.Green);
        }
        if (enableRotator == true)
        {
            Vector3 world;
            if (rotatorXYZ == 0)
            {
                world = new Vector3(angle, 0, 0);
            }
            else if (rotatorXYZ == 1)
            {
                world = new Vector3(0, angle, 0);
            }
            else
            {
                world = new Vector3(0, 0, angle);
            }
```

```
            device.Transform.World = Matrix.RotationAxis(world, angle);
            angle += 0.05f / (float)Math.PI;
        }
        device.EndScene();
        device.Present();
        if (WindowState != FormWindowState.Minimized)
        {
            this.Invalidate();
        }
    }
```

(8) 在窗体的 KeyDown 事件中添加代码：

```
    private void Form1_KeyDown(object sender, KeyEventArgs e)
    {
        switch (e.KeyCode)
        {
            case Keys.Escape:
                this.Close();
                break;
            case Keys.F1:
                showHelpString = !showHelpString;
                break;
            case Keys.F2:
                OnVertexBufferCreate(vertexBuffer, null);
                break;
            case Keys.F3:
                enableRotator = !enableRotator;
                break;
            case Keys.F4:
                rotatorXYZ = (rotatorXYZ + 1) % 3;
                break;
            case Keys.F5:
                enableCullMode = !enableCullMode;
                break;
            case Keys.D1:
                primitivesType = "点列";
                break;
            case Keys.D2:
                primitivesType = "线列";
                break;
            case Keys.D3:
                primitivesType = "线带";
                break;
            case Keys.D4:
                primitivesType = "三角形列";
                break;
            case Keys.D5:
                primitivesType = "三角形带";
                break;
            case Keys.D6:
                primitivesType = "三角扇形";
                break;
```

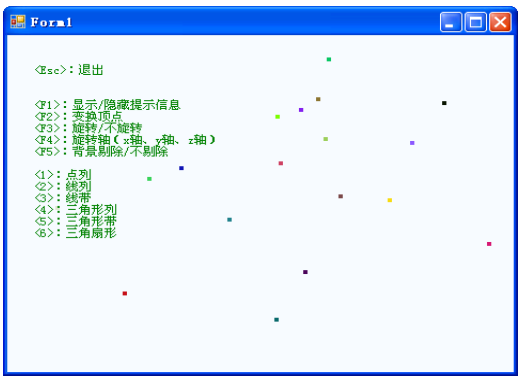
```
        }
    }
```

(9) 修改 Program.cs 中的 Main 方法为如下代码：

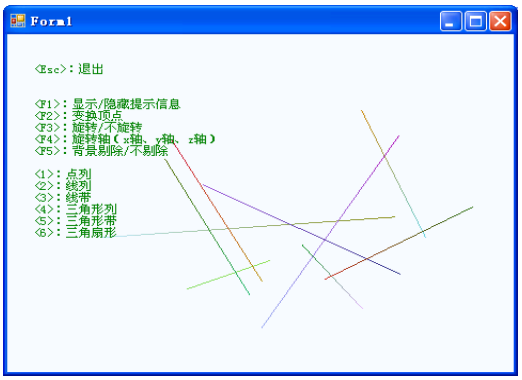
```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Form1 frm = new Form1();
    if (frm.InitializeGraphics() == false)
    {
        MessageBox.Show("显卡不支持 3D 或者未安装配套的显卡驱动程序！");
        return;
    }
    Application.Run(frm);
}
```

(10) 按<F5>键编译并运行。选择数字 1-6 观察绘制的基本图形，每次均用鼠标拖动右下角改变一下窗体的大小，观察图形有什么变化。

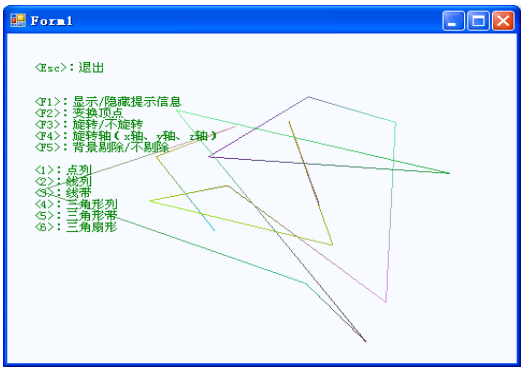
图 7-5 为运行该程序得到的部分图形效果。



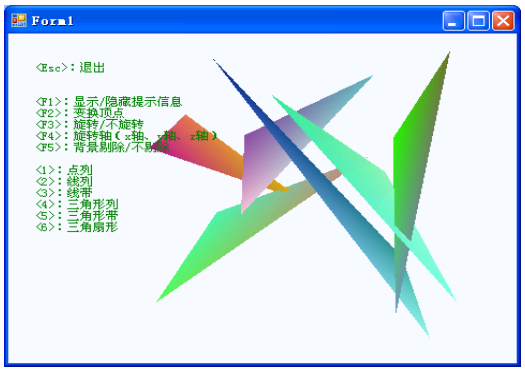
(a) 点列



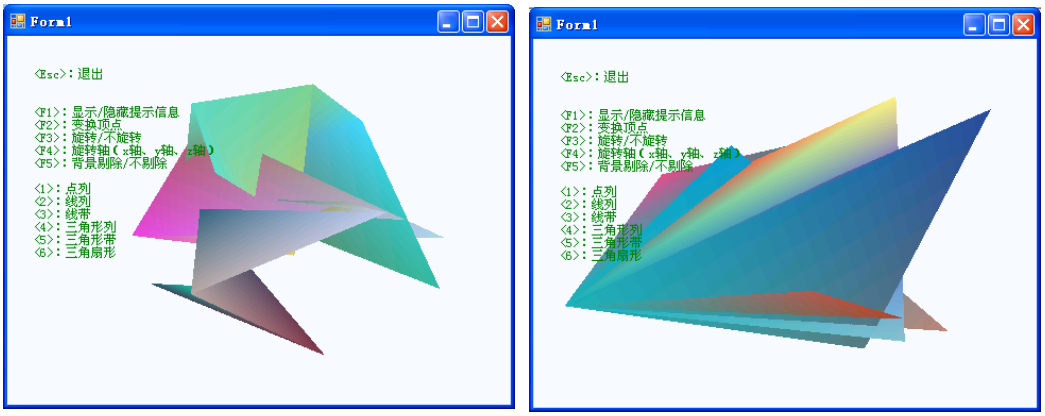
(b) 线列



(c) 线带



(d) 三角形列



(e) 三角形带　　(f) 三角扇形

图7-5　六种基本的Primitive运行效果

(a) 点列　(b) 线列　(c) 线带　(d) 三角形列　(e) 三角形带　(f) 三角扇形

这时，如果让绘制的图形旋转起来，然后分别选择 4、5、6 观察旋转效果，我们会发现一个问题：绘制的图形在旋转时看起来眼花缭乱，感觉层次感不是很强，看不清到底是如何旋转的。

为了解决这个问题，需要引入一个新概念：深度缓冲。

深度缓冲（Depth Buffer）也就是通常所说的 z-buffer。深度信息用于在光栅化时决定像素之间的替代关系，平常我们都是用视点到观察点的距离来衡量“深度”，这样带有较大深度值的像素就会被带有较小深度值的像素替代，即远处的物体被近处的物体遮挡住了。在 Direct3D 中，“深度”的方向为从摄像机位置到观察点的方向，一般为从 z 轴的负坐标到正坐标的方向，即指向显示器屏幕里面的方向。

在本章以前的例子中，程序都没有使用深度缓冲，即光栅化时没有遮挡住任何象素，当然没有层次感了。

(11) 将 InitializeGraphics 方法的代码修改为如下形式：

```
public bool InitializeGraphics()
{
    try
    {
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed = true;
        presentParams.SwapEffect = SwapEffect.Discard;
        presentParams.AutoDepthStencilFormat = DepthFormat.D16;
        presentParams.EnableAutoDepthStencil = true;
        device = new Device(0, DeviceType.Hardware, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
        return true;
    }
    catch (DirectXException)
    {
        return false;
```

```
        }
    }
```

然后将 Form1_Paint 中的 device.Clear 方法改为：

```
device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.AliceBlue, 1.0f, 0);
```

按<F5>键重新运行程序，观察各个基本图形旋转的效果。

可见，把 EnableAutoDepthStencil 设置为 true 就可以为 device 打开深度缓冲，使用 DepthFormat 指定 AutoDepthStencilFormat 枚举成员。在 DepthFormat 枚举中，有很多枚举选项，选择的深度缓冲越大，能存储的深度数据也越多，但这是以牺牲性能为代价的。一般情况下，使用最小的值就可以了。由于目前大部分显卡都支持最小的 16 位深度缓冲，所以只需要使用 DepthFormat.D16 即可。

加入深度缓冲后，还应该在调用 device 的 clear 方法时同时清除深度缓冲，这样才能得到希望的效果。

7.4 Mesh

任何一个物体，不论是表面多么平滑的物体，其视图都可以用三角形构成的面组合得到。如果只画出三角形的连线，可以看出一个物体的框架是由许多个三角形构成的网格组成，Direct3D 把由三角形网格组成的对象叫做 Mesh 对象。

在 Direct3D 中，有一个 Mesh 类，使用它可以存储任何类型的图形数据，但一般主要用它封装复杂的模型。Mesh 类提供了构造几何体的方法，同时也提供了用来提高渲染物体性能的方法。

在本章开头介绍的版本中，Mesh 对象位于 Direct3D 扩展库中。要使用 Mesh 类，必须添加对 Direct3DX.dll 程序集的引用。

Mesh 类中提供了几个静态方法，可以用来直接创建或加载不同的 Mesh 对象模型。

1) TextFromFont 方法。创建包含指定文本的 Mesh 对象。常用形式为：

```
TextFromFont(Device device, System.Drawing.Font font, string text, float deviation, float extrusion);
```

其中：

device：代表显卡适配器的 Device 对象；

font：创建 Mesh 的 System.Drawing.Font 字体对象；

text：文本字符串；

deviation：立体字偏转大小；

extrusion：立体字阴影部分的大小。

2) Box 方法。创建立方体盒子。常用形式为：

```
Box(Device device, float width, float height, float depth);
```

其中：

device：代表显卡适配器的 Device 对象；

width：x 轴坐标大小；

height：y 轴坐标大小；

depth：z 轴坐标大小。

3) Cylinder 方法。创建圆筒。常用形式为：

```
Cylinder(Device device, float radius1, float radius2, float length, int slices, int stacks);
```

其中：

device：代表显卡适配器的 Device 对象；

radius1：z 轴负方向端点的半径，该值必须大于或等于零；

radius2：z 轴正方向端点的半径，该值必须大于或等于零；

length：沿 z 轴方向的圆筒长度；

slices：垂直于 z 轴方向绕 z 轴构成圆筒面的线的条数；

stacks：沿 z 轴方向构成圆筒面的线的条数。

4) Polygon 方法。创建多边形。常用形式为：

```
Polygon(Device device, float length, int sides);
```

其中：

device：代表显卡适配器的 Device 对象；

length：每条边的长度；

sides：边数，该值必须大于或等于 3。

5) Sphere 方法。创建球形。常用形式为：

```
Sphere(Device device, float radius, int slices, int stacks);
```

其中：

device：代表显卡适配器的 Device 对象；

radius：小球半径；

slices：垂直于 z 轴方向绕 z 轴构成球表面的线的条数；

stacks：沿 z 轴方向构成球表面的线的条数。

6) Torus 方法。创建圆环。常用形式为：

```
Torus(Device device, float innerRadius, float outerRadius, int sides, int rings);
```

其中：

device：代表显卡适配器的 Device 对象；

innerRadius：圆环内半径；

outerRadius：圆环外半径；

sides：边数，该值必须大于或等于 3；

rings：环数，该值必须大于或等于 3。

如何渲染 mesh 呢？在渲染 Mesh 对象时，系统会自动根据属性缓冲的大小，将 Mesh 对象划分为一系列的子集（subsets），同时使用一个叫做 DrawSubset 的方法来渲染该对象。例如：

```
mesh.DrawSubset(0);
```

注意，使用 Mesh 类直接创建的基本图形只有一个索引号为零的子集。

【例 7-3】演示 Mesh 对象的绘制方法。

(1) 新建一个名为 MeshExample 的 Windows 应用程序，使用默认的窗体名称。

(2) 在【解决方案资源管理器】中，鼠标右键单击【引用】→【添加引用】→在【.net】选项下同时选中 Microsoft.DirectX、Microsoft.DirectX.Direct3D 和 Microsoft.DirectX.Direct3DX，注意选择的版本要和本章开头列出的版本一致，单击【确定】。

(3) 在命名空间的上方，添加引用代码：

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

(4) 在构造函数上方添加字段声明：

```
private Device device = null;
private Mesh boxMesh = null;
private Mesh cylinderMesh = null;
private Mesh polygonMesh = null;
private Mesh sphereMesh = null;
private Mesh teapotMesh = null;
private Mesh torusMesh = null;
private Mesh textMesh = null;
private Microsoft.DirectX.Direct3D.Font d3dfont;
private string helpString;
private bool showHelpString = true;
private float angle = 0.0f;
private int rotatorXYZ = 0;
private bool enableRotator = true;
private bool enableCullMode = false;
private bool enableSolidMode = false;
```

(5) 直接添加代码：

```
public bool InitializeGraphics()
{
    try
    {
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed = true;
        presentParams.SwapEffect = SwapEffect.Discard;
        device = new Device(0, DeviceType.Hardware, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
        return true;
    }
    catch (DirectXException)
    {
        return false;
    }
}

private void SetupCamera()
{
    float fieldOfView = (float)Math.PI / 4;
    float aspectRatio = this.Width / this.Height;
    float nearPlane = 1.0f;
    float farPlane = 100.0f;
    device.Transform.Projection =
        Matrix.PerspectiveFovLH(fieldOfView, aspectRatio, nearPlane, farPlane);
    Vector3 cameraPosition = new Vector3(0, 0, -18.0f);
    Vector3 cameraTarget = new Vector3(0, 0, 0);
    Vector3 upDirection = new Vector3(0, 1, 0);
    device.Transform.View = Matrix.LookAtLH(cameraPosition, cameraTarget, upDirection);
    device.RenderState.Lighting = false;
    device.RenderState.CullMode = (enableCullMode ? Cull.CounterClockwise : Cull.None);
    device.RenderState.FillMode = (enableSolidMode ? FillMode.Solid : FillMode.WireFrame);
}
private void SetWorldTransform(float x, float y, float z)
```

```
{
    Vector3 world;
    if (rotatorXYZ == 0)
    {
        world = new Vector3(angle, 0, 0);
    }
    else if (rotatorXYZ == 1)
    {
        world = new Vector3(0, angle, 0);
    }
    else
    {
        world = new Vector3(0, 0, angle);
    }
    device.Transform.World = Matrix.RotationAxis(world, angle) * Matrix.Translation(x, y, z);
    if (enableRotator == true)
    {
        angle += 0.03f / (float)Math.PI;
    }
}
```

(6) 在窗体的 Load 事件中添加代码：

```
private void Form1_Load(object sender, EventArgs e)
{
    this.Width = 575;
    this.Height = 475;
    this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque, true);
    this.KeyPreview = true;
    helpString = "<Esc>：退出\n\n\n" +
        "<F1>：显示/隐藏提示信息\n" +
        "<F2>：实心/线框\n" +
        "<F3>：旋转/不旋转\n" +
        "<F4>：旋转轴（x 轴、y 轴、z 轴）\n" +
        "<F5>：背景剔除/不剔除\n\n";
    System.Drawing.Font winFont = new System.Drawing.Font("宋体", 9, FontStyle.Regular);
    d3dfont = new Microsoft.DirectX.Direct3D.Font(device, winFont);
    d3dfont.PreloadText(helpString);
    textMesh = Mesh.TextFromFont(
       device, new System.Drawing.Font("宋体", 12), "Mesh 对象展示", 1.0f, 0.3f);
    boxMesh = Mesh.Box(device, 2.0f, 2.0f, 2.0f);
    cylinderMesh = Mesh.Cylinder(device, 1.0f, 0.5f, 2.0f, 10, 10);
    polygonMesh = Mesh.Polygon(device, 0.4f, 20);
    sphereMesh = Mesh.Sphere(device, 1.0f, 20, 10);
    teapotMesh = Mesh.Teapot(device);
    torusMesh = Mesh.Torus(device, 0.3f, 1.0f, 20, 20);
}
```

(7) 在窗体的 Paint 事件中添加代码：

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    SetupCamera();
    device.Clear(ClearFlags.Target, Color.DarkSeaGreen, 1.0f, 0);
    device.BeginScene();
```

```
        // 绘制文本
        SetWorldTransform(0, 5, 0);
        textMesh.DrawSubset(0);
        // 绘制立方体
        SetWorldTransform(-5, 0, 0);
        boxMesh.DrawSubset(0);
        // 绘制圆筒
        SetWorldTransform(0, 0, 0);
        cylinderMesh.DrawSubset(0);
        // 绘制多边形
        SetWorldTransform(5, 0, 0);
        polygonMesh.DrawSubset(0);
        // 绘制球形
        SetWorldTransform(-5, -5, 0);
        sphereMesh.DrawSubset(0);
        // 绘制茶壶
        SetWorldTransform(0, -5, 0);
        teapotMesh.DrawSubset(0);
        // 绘制圆环
        SetWorldTransform(5, -5, 0);
        torusMesh.DrawSubset(0);
        if (showHelpString == true)
        {
            d3dfont.DrawText(null, helpString, 25, 30, Color.Tomato);
        }
        device.EndScene();
        device.Present();
        if (WindowState != FormWindowState.Minimized)
        {
            this.Invalidate();
        }
    }
```

(8) 在窗体的 KeyDown 事件中添加代码：

```
    private void Form1_KeyDown(object sender, KeyEventArgs e)
    {
        switch (e.KeyCode)
        {
            case Keys.Escape:
                this.Close();
                break;
            case Keys.F1:
                showHelpString = !showHelpString;
                break;
            case Keys.F2:
                enableSolidMode = !enableSolidMode;
                break;
            case Keys.F3:
                enableRotator = !enableRotator;
                break;
            case Keys.F4:
                rotatorXYZ = (rotatorXYZ + 1) % 3;
```

```
                break;
            case Keys.F5:
                enableCullMode = !enableCullMode;
                break;
        }
    }
```

(9) 修改 Program.cs 中的 Main 方法为如下代码：

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Form1 frm = new Form1();
    if (frm.InitializeGraphics() == false)
    {
        MessageBox.Show("显卡不支持 3D 或者未安装配套的显卡驱动程序！");
        return;
    }
    Application.Run(frm);
}
```

(10) 按<F5>键编译并运行，图 7-6 为运行该程序得到的界面效果。分别选择功能键<F1>到<F5>观察绘制的基本图形有什么变化。

图7-6　例7-3的运行界面

如果按<F2>键将 Mesh 对象变为实心，即渲染各个表面，我们会发现显示的效果就非常差了，几乎看不出球形是如何旋转的，而且所有 Mesh 对象的颜色也全部是白色的，在代码中也找不到更改的办法。

因此，我们还必须再学习一个新的概念：灯光与材质。

7.5　灯光与材质

在 DirectX 中，我们可以给场景创建不同类型的光，使场景看起来更真实。当然，DirectX

中的光只是在近似地模拟自然界的光。

自然界中，光由光源（如灯泡）发出，然后沿直线传播，直到光能消耗完毕或传入眼睛；光在遇到物体时会发生反射，每次反射都会有能量的消耗；实际上，光可以在物体间反射千百万次，而且光滑表面会比不光滑的表面反射更多的光。这一切如果全算进来的话，计算量是巨大的。所以，为了提高效率和速度，DirectX 只是在近似地模仿自然界的光。

1. 光的类型

在 DirectX 中，有四种类型的光，分别是环境光、直射光、点光源和聚光灯。

人们看到的物体的颜色可以认为是光线照射到物体上后反射到人的眼睛的颜色。如果没有任何光线，我们看到的将是漆黑一片。为了能看见计算机屏幕上显示的物体，Direct3D 首先需要提供一种没有位置、没有方向、没有衰减、所有光线都均匀不变地从各个方向照射到物体上的光，这就是环境光。

在 DirectX 中，默认情况下 Device 对象的灯光是打开的，但是为了使程序看起来更清晰，一般在代码中加上：

```
device.RenderState.Lighting = true;
```

灯光打开后，系统默认提供的光就是黑色的环境光，即相当于夜晚。这种情况下，如果物体本身不发光，不论物体本身的颜色是什么，我们看到的都会是黑色。当然，我们希望看到的是类似于白天的情况，因此，需要设置环境色为白色，例如：

```
device.RenderState.Ambient = Color.White;
```

为了体现各种光照效果，系统默认的环境光比较暗，类似于在没有其他灯光的房间里感觉的光线效果。在实际环境中，如果希望房间内比较亮一些，还要打开日光灯。在 DirectX 中，与实际环境相似，要得到较好的效果，还需要设置其他类型的光，可选的灯光类型有：直射光、点光源和聚光灯。灯光的个数可以根据亮度需要使用一个或者多个，具体可以同时使用多少个灯光，则受材质个数限制。

三种类型的光中，均有以下两种属性：

- Diffuse 属性：表示漫射光颜色，是一种有方向的光，其方向就是 Direction 属性指定的方向。例如：device.Lights[0].Diffuse = Color.White;
- Ambient 属性：环境光颜色，是一种无方向的光，例如：

```
device.Lights[0].Ambient = Color.Black;
```

注意：如果指定了灯光的环境光颜色，则实际光的颜色为灯光的环境光颜色和通用的环境光颜色的混合色，至于混合后是什么颜色，由三基色原理决定。

为了避免使用复杂，一般情况下，不论是哪种类型的光，我们均设置其颜色为白色。这样，物体的实际颜色就由它自身提供的反射色决定了。否则，我们就必须要知道哪种颜色和哪种颜色混合，从而得到什么颜色，这样才能正确的使用。

除了以上两种颜色外，对不同类型的灯光，还需要设置与类型相关的其他属性。

1) 直射光

直射光有方向，没有位置，也没有范围和衰减。场景中的所有对象都会从同样的方向接收到同样的直射光。一般情况下，只需要为直射光设置 Direction 属性，例如：

```
device.Lights[0].Direction = new Vector3(0, 0, 1);
```

该语句的意思是设置首个直射光的方向为 z 轴正方向。

2) 点光源

点光源没有方向，但是有光照范围，有光的衰减。例如灯泡发出的光就可以认为是点光源。

一般情况下，只需要为点光源设置位置属性，例如：

```
device.Lights[0].Position = new Vector3(0, 0, 0);
```

该语句的意思是设置首个点光源的位置为世界坐标的原点。

3) 聚光灯

聚光灯有位置、方向、范围、发光内径和发光外径，光照强度还会随距离而衰减。聚光灯的恰当的例子就是手电筒或者剧场里用来照亮舞台上人物的灯光。

注意，三种类型的光中，不论使用哪种光，都会增加系统的计算复杂度，按照复杂度从小到大排序分别为：直射光、点光源、聚光灯。

2. 材质

材质（Material）用于描述物体的反光性能，使它的表面看起来有没有光泽和带有什么颜色。可以认为材质的颜色就是人们在自然界中看到的物体的颜色。在不同类型的光照射下，使用的材质不同，物体反射的光也不同。例如，如果把红色的灯光照在淡蓝色的表面上，它就会呈献出柔和的紫色。

创建一个材质并设置光照射到物体表面上反射的颜色后，就可以设置 Device 对象的 Material 属性，让其使用创建的材质。这样 Direct3D 就知道渲染时使用哪种材质数据。

材质的环境反射色（Ambient）表示物体本身在光照射下反射的颜色，其他灯光照射到物体表面上后，物体实际的颜色则是灯光颜色、环境颜色和材质颜色的混合色。

注意，当材质的环境反射色为黑色时，表示物体的颜色由物体本身的颜色决定，其他光不影响材质本身的颜色；当环境反射色变浅时，其他光就会照亮材质，并将光的颜色和材质的颜色混和起来，从而得到人们看到的混合后的颜色。

描述材质的常见属性有：

Diffuse：物体的漫反射色，如果灯光为白色，则漫反射色就是灯光照射部分物体的颜色。

Ambient：物体的环境反射色，如果环境色为白色，反射色为非黑色的其他颜色，则物体的颜色由环境反射色决定。

Specular：物体的镜面反射与高光强度。设用这个属性可以使物体看起来发亮。

Emission：物体自发光颜色。

【例 7-4】灯光、材质、Primitive 和 Mesh 综合举例。

(1) 新建一个名为 LineSphereExample 的 Windows 应用程序，使用默认的窗体名称。

(2) 在【解决方案资源管理器】中，鼠标右键单击【引用】→【添加引用】→在【.net】选项下同时选中 Microsoft.DirectX、Microsoft.DirectX.Direct3D 和 Microsoft.DirectX.Direct3DX，注意选择的版本要和本章开头列出的版本一致，单击【确定】，将对三个文件的引用加入到当前项目中。

(3) 在命名空间的上方，添加引用代码：

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

(4) 在构造函数上方添加字段声明：

```
private Device device = null;
private Microsoft.DirectX.Direct3D.Font d3dfont;
private string helpString;
private bool showHelpString = true;
```

```
private float angle = 0;
private bool enableRotator = true;
private int rotatorXYZ = 0;
private float rotateSpeed = 0.01f;              //旋转速度
private bool enableSolidMode = true;
private bool enableCullMode = true;
private Mesh[] sphereMeshs;
private float sphereRadius = 1.5f;              //小球半径
private int sphereNumber = 18;
private Matrix[] spherePositions;     //原始位置变换矩阵
private float xRadius = 15.0f;              //x 方向圆的半径
private int ySpacing = 10;              //y 方向相对空间大小
private VertexBuffer vertexBuffer = null; //顶点缓冲
private Material[] sphereMaterial;              //小球材质
private Material[] lineMaterial;                //线的材质
private bool enableEmissive = false;        //是否允许物体本身发光
private Material commonSphereMaterial = new Material();
private Material commonLineMaterial = new Material();
private bool multiMaterial = true;
private bool isPointLight = false;
private bool enableLight = true;
```

(5) 直接添加代码：

```
public bool InitializeGraphics()
{
    try
    {
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed = true;
        presentParams.SwapEffect = SwapEffect.Discard;
        presentParams.AutoDepthStencilFormat = DepthFormat.D16;
        presentParams.EnableAutoDepthStencil = true;
        device = new Device(0, DeviceType.Hardware, this,
            CreateFlags.SoftwareVertexProcessing, presentParams);
        device.RenderState.ZBufferEnable = true;
        return true;
    }
    catch (DirectXException)
    {
        return false;
    }
}
public void BuildScene()
{
    this.buildMeshs();                      //创建 Mesh 对象
    this.BuildspherePositions();            //构造小球位置
    this.BuildLineVertexBuffer();           //创建小球间的连线顶点缓冲
    BuildMaterials();                       //创建材质
}
private void buildMeshs()
{
    //一定要及时释放资源，不能靠垃圾回收自动回收，
```

```
        //否则退出后会很长时间无反应，像死机一样
        if (sphereMeshs != null)
        {
            for (int i = 0; i < sphereMeshs.Length; i++)
            {
                sphereMeshs[i].Dispose();
            }
        }
        sphereMeshs = new Mesh[sphereNumber];
        for (int i = 0; i < sphereNumber; i++)
        {
            sphereMeshs[i] = Mesh.Sphere(device, sphereRadius, 30, 15);
        }
    }
    private void BuildspherePositions()
    {
        spherePositions = new Matrix[sphereNumber];
        float alfa = 360.0f / (sphereNumber - 2);
        for (int i = 0; i < sphereNumber; i++)
        {
            if (i == 0)
            {
                spherePositions[i] = Matrix.Translation(0, ySpacing + 5, 0);
            }
            else if (i == sphereNumber - 1)
            {
                spherePositions[i] = Matrix.Translation(0, -ySpacing, 0);
            }
            else
            {
                //将小球按圆周在水平面平均分布
                float xx = (float)(xRadius * Math.Sin(i * alfa * Math.PI / 180.0f));
                float zz = (float)(xRadius * Math.Cos(i * alfa * Math.PI / 180.0f));
                spherePositions[i] = Matrix.Translation(xx, 0, zz);
            }
        }
    }
    private void BuildLineVertexBuffer()
    {
        vertexBuffer = new VertexBuffer(typeof(CustomVertex.PositionOnly),
            (sphereNumber - 2) * 4 + 2,
            device, 0, CustomVertex.PositionOnly.Format, Pool.Default);
        vertexBuffer.Created += new System.EventHandler(this.OnCreateVertexBuffer);
        this.OnCreateVertexBuffer(vertexBuffer, null);
    }
    public void OnCreateVertexBuffer(object sender, EventArgs e)
    {
        VertexBuffer buffer = (VertexBuffer)sender;
        CustomVertex.PositionOnly[] verts =
            (CustomVertex.PositionOnly[])buffer.Lock(0, 0);
        Random random = new Random();
```

```
        for (int i = 0; i < sphereNumber - 2; i++)
        {
            Vector3 vector = new Vector3(spherePositions[i + 1].M41,
                spherePositions[i + 1].M42, spherePositions[i + 1].M43);
            verts[i * 2].Position = vector;
            verts[i * 2 + 1].Position = new Vector3(0, ySpacing, 0);
            verts[2 * (sphereNumber - 2) + (i * 2)].Position = vector;
            verts[2 * (sphereNumber - 2) + (i * 2 + 1)].Position =
                new Vector3(0, -ySpacing, 0);
        }
        verts[sphereNumber - 1].Position = new Vector3(spherePositions[0].M41,
            spherePositions[0].M42, spherePositions[0].M43);
        verts[sphereNumber - 2].Position = new Vector3(0, ySpacing, 0);
        vertexBuffer.Unlock();
    }
    private void SetupCamera()
    {
        float fieldOfView = (float)Math.PI / 4;
        float aspectRatio = (float)this.Width / (float)this.Height;
        float nearPlane = 1.0f;
        float farPlane = 200.0f;
        device.Transform.Projection =
            Matrix.PerspectiveFovLH(fieldOfView, aspectRatio, nearPlane, farPlane);
        Vector3 cameraPosition = new Vector3(0, 0, -50.0f);
        Vector3 cameraTarget = new Vector3(0, 0, 0);
        Vector3 upDirection = new Vector3(0, 1, 0);
        device.Transform.View = Matrix.LookAtLH(cameraPosition, cameraTarget, upDirection);
        device.RenderState.FillMode = (enableSolidMode ? FillMode.Solid : FillMode.WireFrame);
        device.RenderState.CullMode = (enableCullMode ? Cull.CounterClockwise : Cull.None);
    }
    private void BuildMaterials()
    {
        Random random = new Random();
        Color color;
        sphereMaterial = new Material[sphereNumber];
        for (int i = 0; i < sphereNumber; i++)
        {
            color = Color.FromArgb(random.Next(byte.MaxValue),
                random.Next(byte.MaxValue), random.Next(byte.MaxValue));
            sphereMaterial[i].Diffuse = color;
            //注意：设置 Emissive 的目的是为了让物体本身发光，以方便演示光照效果
            //如果不设置此属性，物体看起来会更逼真
            if (enableEmissive == true)
            {
                sphereMaterial[i].Emissive = color;
            }
        }
        int lines = (sphereNumber - 2) * 2 + 1;
        lineMaterial = new Material[lines];
        for (int i = 0; i < lines; i++)
        {
```

```
            color = Color.FromArgb(random.Next(byte.MaxValue),
                random.Next(byte.MaxValue), random.Next(byte.MaxValue));
            lineMaterial[i].Ambient = color;
        }
    }
    private void SetupLights()
    {
        device.RenderState.Ambient = Color.White;
        if (isPointLight == false)
        {
            device.Lights[0].Type = LightType.Directional;
            device.Lights[0].Direction = new Vector3(0, 0, 1); //指向 z 轴正方向
        }
        else
        {
            device.Lights[0].Type = LightType.Point;
            device.Lights[0].Position = new Vector3(6, 0, 0); //旋转轴中心位置
        }
        device.Lights[0].Range = 500.0f;
        device.Lights[0].Enabled = enableLight;
    }
    public void RendScene()
    {
        if (enableRotator)
        {
            angle += rotateSpeed;
        }
        ////画小球
        for (int i = 0; i < sphereNumber; i++)
        {
            //设置材质
            if (multiMaterial)
            {
                device.Material = sphereMaterial[i];
            }
            else
            {
                device.Material = commonSphereMaterial;
            }
            SetWorldTransform(spherePositions[i].M41, spherePositions[i].M42, spherePositions[i].M43);
            sphereMeshs[i].DrawSubset(0);
        }
        //重新进行矩阵变换
        SetWorldTransform(0, 0, 0);
        //画线
        int lines = (sphereNumber - 2) * 2 + 1;
        device.SetStreamSource(0, vertexBuffer, 0);
        device.VertexFormat = CustomVertex.PositionOnly.Format;
        for (int i = 0; i < lines; i++)
        {
            if (multiMaterial)
```

```
                {
                    device.Material = lineMaterial[i];
                }
                else
                {
                    device.Material = commonLineMaterial;
                }
                device.DrawPrimitives(PrimitiveType.LineList, i * 2, 1);
            }
            if (showHelpString == true)
            {
                d3dfont.DrawText(null, helpString, 25, this.Height - 290, Color.White);
            }
        }
        private void SetWorldTransform(float x, float y, float z)
        {
            Vector3 world;
            if (rotatorXYZ == 0)
            {
                world = new Vector3(angle, 0, 0);
            }
            else if (rotatorXYZ == 1)
            {
                world = new Vector3(0, angle, 0);
            }
            else
            {
                world = new Vector3(0, 0, angle);
            }
            device.Transform.World =
                Matrix.Translation(x, y, z) *
                Matrix.RotationAxis(world, angle) * Matrix.Translation(6, 0, 0);
        }
```

(6) 在窗体的 Load 事件中添加代码：

```
        private void Form1_Load(object sender, EventArgs e)
        {
            this.Width = 700;
            this.Height = 520;
            this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque, true);
            this.KeyPreview = true;// ↑ ↓ ←→
            helpString = "<Esc>：退出\n\n" +
                "<F1>：显示/隐藏提示信息\n" +
                "<F2>：实心/线框\n" +
                "<F3>：背面剔除/不剔除\n" +
                "<F4>：旋转/不旋转\n" +
                "<F5>：旋转轴（x 轴、y 轴、z 轴）\n\n" +
                "<0>：物体本身发光/物体本身不发光\n" +
                "<1>：关闭/打开灯光\n" +
                "<2>：直射光\n" +
                "<3>：点光源\n" +
                "<4>：单一材质/多种材质\n" +
```

```
            "<5>：变换小球和连线材质\n\n" +
            "上下左右箭头键：增加、减少小球个数\n" +
            "<Alt>+箭头键  ：提高、降低旋转速度\n" +
            "<Ctrl>+箭头键 ：增加、减少大圆半径\n" +
            "<Shift>+箭头键：增加、减少小球半径";
        System.Drawing.Font winFont = new System.Drawing.Font("宋体", 9, FontStyle.Regular);
        d3dfont = new Microsoft.DirectX.Direct3D.Font(device, winFont);
        d3dfont.PreloadText(helpString);
        BuildScene();          //创建小球及连线
        device.RenderState.Lighting = true;
        //设置环境光颜色
        device.RenderState.Ambient = Color.White;
        //单一材质时，所有小球使用漫射光，受有方向的光照影响
        commonSphereMaterial.Diffuse = Color.Red;
        commonSphereMaterial.Ambient = Color.Black;
        //单一材质时，所有连线均使用连线材质的环境反射色
        //当环境光为白色时，连线的颜色就是指定的环境反射色（此句中连线也使用白色）
        commonLineMaterial.Ambient = Color.White;
        SetupLights();        //设置灯光
    }
```

(7) 在窗体的 Paint 事件中添加代码：

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.DarkSeaGreen, 1.0f, 0);
        SetupCamera();
        device.BeginScene();
        this.RendScene();
        device.EndScene();
        device.Present();
        if (WindowState != FormWindowState.Minimized)
        {
            this.Invalidate();
        }
    }
```

(8) 在窗体的 KeyDown 事件中添加代码：

```
    private void Form1_KeyDown(object sender, KeyEventArgs e)
    {
        switch (e.KeyCode)
        {
            case Keys.Escape:
                this.Close(); break;
            case Keys.F1:
                showHelpString = !showHelpString; break;
            case Keys.F2:
                enableSolidMode = !enableSolidMode; break;
            case Keys.F3:
                enableCullMode = !enableCullMode; break;
            case Keys.F4:
                enableRotator = !enableRotator; break;
            case Keys.F5:
                rotatorXYZ = (rotatorXYZ + 1) % 3; break;
```

```
            case Keys.D0:
                enableEmissive = !enableEmissive;
                BuildMaterials();
                break;
            case Keys.D1:
                enableLight = !enableLight; SetupLights(); break;
            case Keys.D2:
                isPointLight = false; SetupLights(); break;
            case Keys.D3:
                isPointLight = true; SetupLights(); break;
            case Keys.D4:
                multiMaterial = !multiMaterial; break;
            case Keys.D5:
                BuildMaterials(); break;
            case Keys.Up:
            case Keys.Right:
                if (e.Alt == true) rotateSpeed += 0.005f;
                else if (e.Shift == true) sphereRadius += 0.05f;
                else if (e.Control == true) xRadius += 0.5f;
                else sphereNumber += 2;
                if (e.Alt == false)
                {
                    BuildScene();
                }
                break;
            case Keys.Down:
            case Keys.Left:
                if (e.Alt == true)
                    rotateSpeed =
                        (rotateSpeed > 0.01 ? rotateSpeed - 0.005f : rotateSpeed);
                else if (e.Shift == true)
                    sphereRadius =
                        (sphereRadius > 0.1f ? sphereRadius - 0.05f : sphereRadius);
                else if (e.Control == true)
                    xRadius = (xRadius > 0.5f ? xRadius - 0.5f : xRadius);
                else
                    sphereNumber = (sphereNumber > 6 ? sphereNumber - 2 : sphereNumber);
                if (e.Alt == false)
                {
                    BuildScene();
                }
                break;
        }
    }
```

(9) 修改 Program.cs 中的 Main 方法为如下代码：

```
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Form1 frm = new Form1();
        if (frm.InitializeGraphics() == false)
```

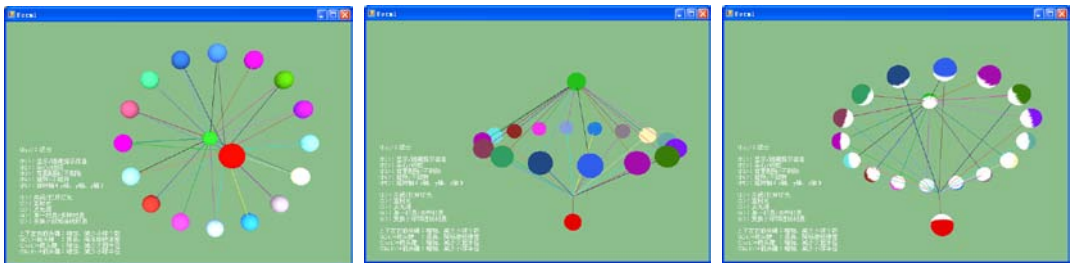
```
        {
            MessageBox.Show("显卡不支持 3D 或者未安装配套的显卡驱动程序！");
            return;
        }
        Application.Run(frm);
    }
```

(10) 按<F5>键编译并运行。分别选择功能键和数字键，观察绘制的图形有什么变化。图 7-7 为运行该程序得到的部分界面效果。

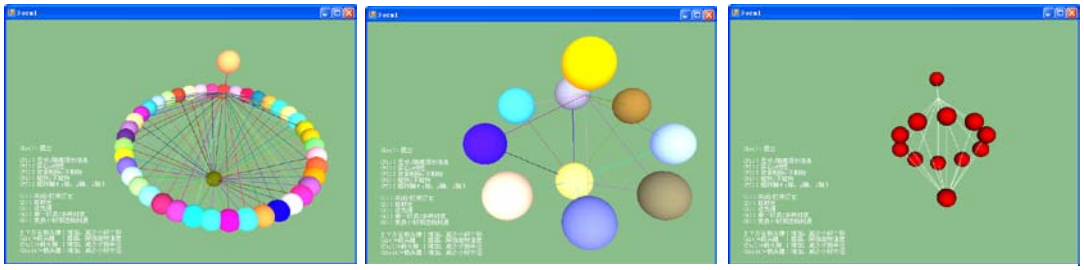
图 7-7(a)是多材质、直射光的效果。这种情况下，每一个小球都有对应的小球材质，小球材质的个数与小球的个数相同；每一条线也都有对应的连线材质，连线材质的个数与连线的个数相同。小球材质和连线材质的颜色都是随机产生的，当然也可以直接指定某个小球或者连线的颜色。由于在这个图中使用了物体本身发光选项，所以场景看起来比较鲜艳。

图 7-7(b)是多材质、无外加灯光、但是物体本身发光的效果。此时，由于每个小球本身发光，所以仍然能看到每个小球的颜色。另外，场景中没有增加其他灯光，因此小球色彩显得较暗。还有一点要注意，如果选择小球本身也不发光，看到的小球颜色将会是黑色的。

图 7-7(c)中，使用了多材质、点光源照射。可以看出，每个小球一半是白色点光源照射的效果，另一半是点光源照射不到，但是物体本身发光的效果。另外，由于所有小球都是围绕（6,0,0）旋转的，所以把点光源放在了（6,0,0）处。



(a) 多材质、直射光　(b) 多材质、无灯光　(c) 多材质、点光源



(d) 增加小球后的效果　(e) 增大小球半径后效果　(f) 单一材质效果

图7-7　例7-4的部分运行界面

7.6　音频与视频

在 DirectX 的 AudioVideoPlayback 命名空间中，提供了简单的音频视频处理功能，同时也在其他命名空间中提供了复杂的处理功能。本节主要通过一个具体例子，介绍如何使用 AudioVideoPlayback 命名空间下的 Audio 类和 Video 类，实现简单的音频和视频播放，以及显

示播放时间和进度等功能。

【例 7-5】制作一个音频视频播放器，具有播放、暂停、停止等功能。

(1) 新建一个名为 AudioVideoExample 的 Windows 应用程序，使用默认的窗体名称。

(2) 在【解决方案资源管理器】中，鼠标右键单击【引用】→【添加引用】→在【.net】选项下同时选中 Microsoft.DirectX、Microsoft.DirectX.AudioVideoPlayback，注意选择的版本要和本章开头列出的版本一致，单击【确定】。

(3) 设计如图 7-8 所示的窗体界面。

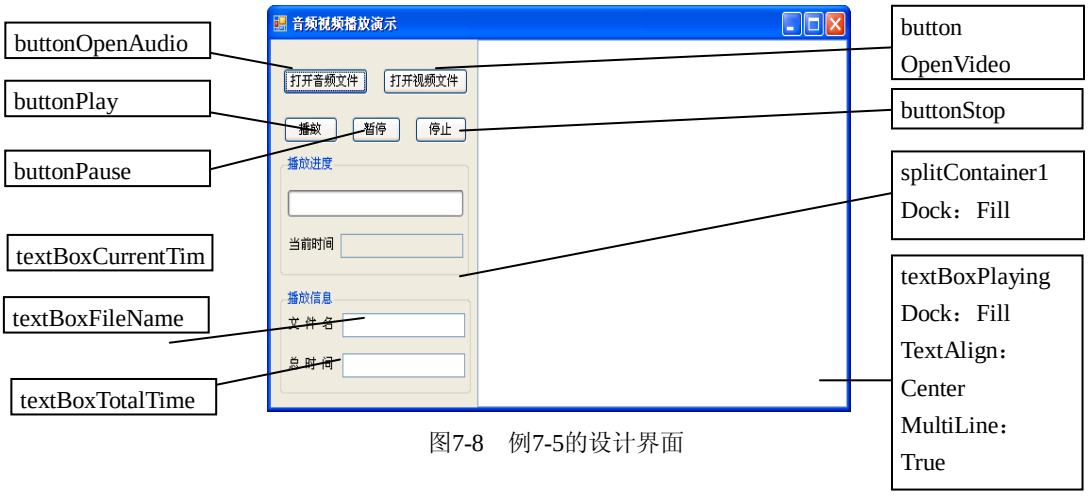


图7-8　例7-5的设计界面

注意，在图 7-8 中，有一个看不见的名为“splitContainer1”的 SplitContainer 控件，该控件是其他控件的容器，它由两个 Panel 部分组成。左边看得到的控件都被包含在 SplitContainer 控件的 panel1 内，右边的 textBoxPlaying 控件被包含在 SplitContainer 控件的 panel2 内。这样做的目的是为了设置 textBoxPlaying 的【Dock】属性为“Fill”，以便视频影片能够按 textBoxPlaying 的大小自动填充，即实现影片缩放功能。使用 SplitContainer 控件可以防止 textBoxPlaying 填满整个窗口，否则左边的部分就无法看到了。

还有一点要注意，在视频播放中，textBoxPlaying 仅起到一个容器的作用，之所以在这个例子中选择文本框控件，是因为音频播放时还需要在其上面显示相关内容。如果只是为了作为视频播放的容器，选择任何一个控件均可。

(4) 在命名空间的上方，添加引用代码：

```
using Microsoft.DirectX.AudioVideoPlayback;
```

(5) 在构造函数上方添加字段声明：

```
Audio audio = null;
Video video = null;
System.Diagnostics.Stopwatch watch = new System.Diagnostics.Stopwatch();
Timer timer1 = new Timer();
```

(6) 在窗体的 Load 事件中添加代码：

```
private void Form1_Load(object sender, EventArgs e)
{
    timer1.Tick += new EventHandler(Timer1_Tick);
    timer1.Interval = 100;
    timer1.Enabled = false;
}
```

(7) 直接添加代码：

```
private void Timer1_Tick(object sender, EventArgs e)
{
    int s = (int)watch.ElapsedMilliseconds;
    if (s <= progressBar1.Maximum)
    {
        progressBar1.Value = s;
    }
    else
    {
        watch.Reset();
        watch.Start();
        progressBar1.Value = 0;
        if (audio != null)
        {
            audio.CurrentPosition = 0;
        }
        else
        {
            video.CurrentPosition = 0;
        }
    }
    textBoxCurrentTime.Text = string.Format("{0}", watch.Elapsed);
}
```

(8) 在 buttonOpenAudio 的 Click 事件中添加代码：

```
private void buttonOpenAudio_Click(object sender, EventArgs e)
{
    OpenFileDialog audioFile = new OpenFileDialog();
    audioFile.CheckFileExists = true;
    if (audioFile.ShowDialog() == DialogResult.OK)
    {
        try
        {
            video = null;
            audio = new Audio(audioFile.FileName);
            textBoxFileName.Text = audioFile.FileName;
            //audio.Duration 的单位为秒，TimeSpan 的单位为 100 纳秒
            textBoxTotalTime.Text = string.Format(
                "{0}", new TimeSpan((long)audio.Duration * 10000000));
            textBoxPlaying.Text = "\r\n\r\n\r\n 请按[播放]按钮开始播放。";
            progressBar1.Minimum = 0;
            progressBar1.Maximum = (int)audio.Duration * 1000;
            watch.Reset();
        }
        catch
        {
            audio = null;
            MessageBox.Show("文件格式不正确");
            textBoxPlaying.Text = "\r\n\r\n\r\n 请单击相应按钮选择播放文件。";
        }
    }
```

```
}
```

(9) 在 buttonOpenVideo 的 Click 事件中添加代码：

```
private void buttonOpenVideo_Click(object sender, EventArgs e)
{
    OpenFileDialog videoFile = new OpenFileDialog();
    videoFile.CheckFileExists = true;
    if (videoFile.ShowDialog() == DialogResult.OK)
    {
        try
        {
            audio = null;
            video = new Video(videoFile.FileName);
            video.Owner = this.textBoxPlaying;
            video.Size = this.textBoxPlaying.Size;
            textBoxFileName.Text = videoFile.FileName;
            textBoxTotalTime.Text = string.Format(
                "{0}", new TimeSpan((long)video.Duration * 10000000));
            textBoxPlaying.Text = "\r\n\r\n\r\n 请单击[播放]按钮开始播放。";
            progressBar1.Minimum = 0;
            progressBar1.Maximum = (int)video.Duration * 1000;
            watch.Reset();
        }
        catch
        {
            video = null;
            MessageBox.Show("文件格式不正确");
            textBoxPlaying.Text = "\r\n\r\n\r\n 请单击相应按钮选择播放文件。";
        }
    }
}
```

(10) 在 buttonPlay 的 Click 事件中添加代码：

```
private void buttonPlay_Click(object sender, EventArgs e)
{
    if(audio==null && video==null)
    {
        MessageBox.Show("请先打开播放文件！");
        return;
    }
    else if (audio != null)
    {
        audio.Play();
    }
    else
    {
        video.Play();
    }
    watch.Start();
    timer1.Enabled = true;
    textBoxPlaying.Text = "\r\n\r\n\r\n 正在播放……";
}
```

(11) 在 buttonPause 的 Click 事件中添加代码：

```
private void buttonPause_Click(object sender, EventArgs e)
{
    if (audio != null)
    {
        audio.Pause();
    }
    else if (video != null)
    {
        video.Pause();
    }
    watch.Stop();
    timer1.Enabled = false;
    textBoxPlaying.Text = "\r\n\r\n\r\n 暂停，再单击[播放]可以继续！";
}
```

(12) 在 buttonStop 的 Click 事件中添加代码：

```
private void buttonStop_Click(object sender, EventArgs e)
{
    if (audio != null)
    {
        audio.Stop(); ;
    }
    else if (video != null)
    {
        video.Stop();
    }
    watch.Reset();
    timer1.Enabled = false;
    progressBar1.Value = 0;
    textBoxPlaying.Text = "\r\n\r\n\r\n 停止！";
    textBoxCurrentTime.Text = string.Format("{0}", watch.Elapsed);
}
```

(13) 按<F5>键编译并执行，打开硬盘上的某个.mp3 音频文件或者.mpg 视频文件，测试播放效果。例如选择视频文件“D:\VCD\森林公园.mpg”，运行效果如图 7-9 所示。

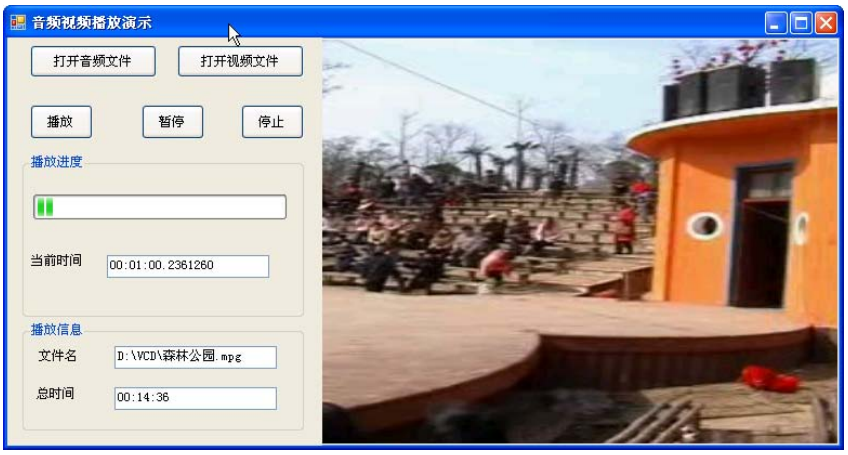


图7-9　例7-5的运行效果

(14) 在视频播放模式下，拖动视频左边的分割条使其左右移动，观察视频缩放情况。

7.7　直接使用 SoundPlayer 类播放 WAV 音频文件

除了使用 DirectX 进行音频视频处理外，在.NET 框架 2.0 的 System.Media 命名空间下还提供了一个 SoundPlayer 类，用于播放 WAV 类型的音频文件以及 WAV 类型的音频流。由于 Windows 操作系统本身支持 WAV 类型的音频播放，因此不需要安装 DirectX 就可以直接使用 SoundPlayer 类在程序中添加声音。

SoundPlayer 类支持从文件路径、URL、包含 WAV 文件的流或者包含 WAV 文件的嵌入资源来载入 WAV 类型的声音。使用 SoundPlayer 类来播放 WAV 文件，需要事先指定文件的路径，然后调用 Play 方法进行播放。WAV 文件的路径是通过设置 SoundLocation 属性实现的。指定路径后，还需要用 Load 方法载入播放文件，然后调用 Play 方法播放。

既可以选择同步方式也可以选择异步方式从流或 URL 载入 WAV 文件。如果调用同步的 Load 或 Play 方法，调用的线程会一直等待，直到方法返回，这样会中断 painting 和其他事件。异步调用 Load 或 Play 方法可以允许线程继续而不中断其他事件。当 SoundPlayer 完成一个 WAV 文件的载入时，会触发 LoadCompleted 事件。

当更改音频资源的路径或 URL 时，会自动触发 SoundLocationChanged 事件。因此可以通过事件中 AsyncCompletedEventArgs 类型的参数来检查载入是否成功。

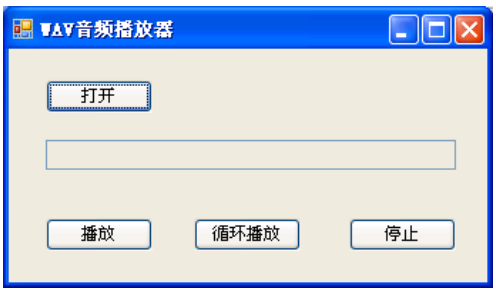


图7-10　例7-6的设计界面

【例 7-6】演示 SoundPlayer 的使用方法。

(1) 新建一个名为 SoundPlayerExample 的 Windows 应用程序，使用默认的窗体名称。设计如图 7-10 所示的界面。

(2) 切换到代码方式，在命名空间的上方添加引用代码：

using System.Media;

(3) 在构造函数上方添加字段声明：

```
SoundPlayer player = new SoundPlayer();
```

(4) 添加【打开】按钮的 Click 事件代码：

```
private void buttonFilePath_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.CheckFileExists = true;
    dlg.Filter = "WAV files (*.wav)|*.wav";
    dlg.DefaultExt = ".wav";
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        textboxFilepath.Text = dlg.FileName;
        player.SoundLocation = textboxFilepath.Text;
        player.Load();
    }
}
```

(5) 添加【播放】按钮的 Click 事件代码：

```
private void buttonPlay_Click(object sender, EventArgs e)
{
    try
    {
        player.Play();
    }
    catch (Exception err)
    {
        MessageBox.Show(err.Message);
    }
}
```

(6) 添加【循环播放】按钮的 Click 事件代码：

```
private void buttonLoopPlay_Click(object sender, EventArgs e)
{
    try
    {
        player.PlayLooping();
    }
    catch (Exception err)
    {
        MessageBox.Show(err.Message);
    }
}
```

(7) 添加【停止】按钮的 Click 事件代码：

```
private void buttonStop_Click(object sender, EventArgs e)
{
    try
    {
        player.Stop ();
    }
    catch (Exception err)
    {
```

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

(8) 按<F5>键编译并运行，选择.WAV 文件，测试各种播放效果。

7.8 习题 7

1. Direct3D 提供的基本图元有哪些？
2. 什么是 Mesh 对象？可以用顶点缓冲构造一个自定义 Mesh 对象吗？
3. 在 Direct3D 中，有几种类型的灯光？各有何特点？

第 8 章 上机实验指导

本章的上机实验指导是在该书的姊妹篇《C#网络应用编程基础》上机实验指导的基础之上，又增加了三个与本书内容配套的高级编程实验内容。

注意，考虑到学习者学习的进度，在安排上机实验内容时，功能需求及实现步骤均与实际的业务处理有非常大的差别，而且有些步骤是多余的，或者实现的办法不是最简单的。但是完成实验的目的不仅仅是为了实现需要的功能，还要通过这些步骤领会涉及到的编程技巧。

另外还要注意，虽然实验都有参考解答，但是实验者一定不要一开始就看参考解答是如何实现的，正确的做法应该是先搞明白功能需求，想一想如何完成这些功能，然后按照实验步骤逐步实现，并思考通过这些步骤能受到什么启发。实在不知道如何实现某个步骤时再看参考解答中对应的内容，只有这样才能体会深刻，并能通过实验举一反三，从而收到比较好的学习效果。

上机实验环境

操作系统：WindowsXP SP2

开发工具：VS2005 Professional 简体中文版（或者 VS2005 Team Suite 简体中文版）

内存要求：至少 256MB。

实验报告要求

1. 使用专用的统一的实验报告纸，每个实验一套，要求字迹工整，内容清晰，注意填写必要的信息：学号、姓名、班级、辅导教师。

2. 必须认真填写实验题目、实验目的等；实验步骤中要求列出当次实验中自己认为有意义的操作过程及各种必要的数据输入输出情况；写出主要的功能模块划分、设计界面及关键源代码，上机调试过程中遇到的问题和解决办法。

8.1 实验一 简单网络聊天系统

实验目的

1. 练习 TcpClient 和 TcpListener 的用法。
2. 练习 NetworkStream 的用法。
3. 练习 BinaryRead 和 BinaryWriter 的用法。
4. 练习线程的创建和使用方法。
5. 练习解决 TCP 协议消息边界问题的另一种方法。

实验内容

开发一个简单的基于 TCP 协议的网络聊天系统，服务器端和客户端设计界面分别如图 8-1 和 8-2 所示。

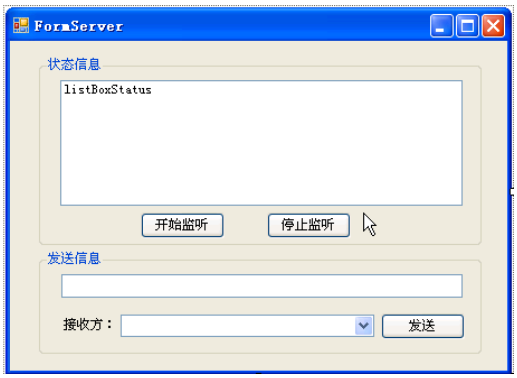


图8-1 服务器端设计界面

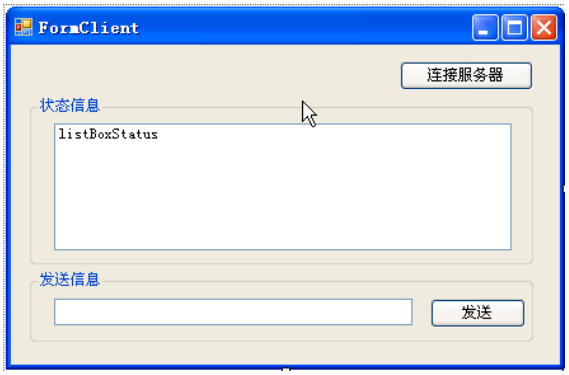


图8-2 客户端设计界面

实验要求

先看懂参考解答的源程序，然后尝试独立完成实验内容。注意一定不要采用复制粘贴参考解答中的源代码的办法完成实验，否则就失去了实验的意义。

实验步骤提示

1. 创建一个名为 ChatServer 的 Windows 应用程序项目，设计服务器端程序。
2. 创建一个名为 ChatClient 的 Windows 应用程序项目，设计客户端程序。
3. 分别运行服务器端程序和客户端程序，验证程序是否正确。

实验报告中要求回答的问题

1. 写出你认为有必要解释的关键步骤和代码。
2. 写出调试中遇到的问题及解决方法。
3. 你从这个实验中受到了哪些启发？

8.2 实验二 网络呼叫应答提醒系统

实验目的

1. 练习 UDP 应用编程的方法。
2. 练习动画窗体的设计方法。
3. 练习自动显示和隐藏窗体的方法。
4. 通过实验步骤学习系统托盘的设计方法。

实验内容

开发一个简单的基于 UDP 协议的网络呼叫应答及时提醒系统。程序运行后，系统自动弹出一个逐渐变大的动画窗体，同时自动在屏幕右下方（即任务栏内）显示一个托盘图标。窗体变化到最大程度后的界面效果如图 8-3 所示。

用户关闭窗体后，窗体消失，但并不结束程序运行。

关闭窗体后，如果用户双击系统托盘图标，系统仍然会自动弹出与启动程序时效果相同的逐渐变大的动画窗体。

如果用户用鼠标右键单击本系统的托盘，系统会自动弹出一个快捷菜单，菜单中提供“呼叫对方”和“退出程序”两个功能，托盘和快捷菜单运行效果如图 8-4 所示。

图8-3 呼叫窗体

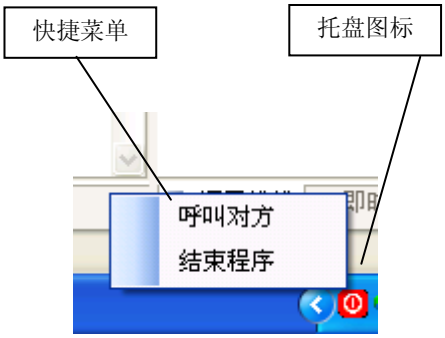


图8-4 屏幕右下方显示的托盘图标

系统弹出快捷菜单后，如果用户选择“呼叫对方”，则程序自动弹出动画窗体，并向接收方发出自己的 IP 地址和呼叫信息；如果用户选择“退出程序”，则结束整个系统运行。

程序运行后，系统会自动创建一个线程在端口 8001 监听网络呼叫信息，当接收到某人呼叫时，弹出一个对话框，提示呼叫方发送的信息。

实验要求

先运行参考解答的源程序，观察运行效果，然后按照实验步骤独立完成实验内容。注意一定不要采用复制粘贴参考解答中的源代码的办法完成实验，否则就失去了实验的意义。

实验步骤提示

1. 创建一个名为 MessageAwake 的 Windows 应用程序项目，修改 Form1.cs 为 FormMain.cs，然后设计图 8-3 所示的界面。
2. 向设计窗体拖放一个 Timer 控件，以便控制窗体动画效果。
3. 向设计窗体拖放一个 ContextMenuStrip 控件，设计如图 8-4 所示的快捷菜单。
4. 在解决方案资源管理器中，鼠标右键单击项目名，选择【添加】→【新建项】，在模板

中选择“图标文件”，文件名为“demo.ico”，设计一个托盘图标。

5. 在解决方案资源管理器中，鼠标右键单击 demo.ico 文件，在快捷菜单中选择【属性】，然后将其【复制到输出目录】属性改为“始终复制”，【生成操作】属性设置为“无”。

6. 分析下列源代码，完成实验要求的功能。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//添加的命名空间引用
using System.Net;
using System.Net.Sockets;
using System.Threading;
namespace MessageAwake
{
    public partial class FormMain : Form
    {
        private System.Windows.Forms.NotifyIcon myNotifyIcon;
        private bool isExit = false;
        private int formHeight;
        //使用的接收端口号
        private int port = 8001;
        private UdpClient udpClient;
        public FormMain()
        {
            InitializeComponent();
            formHeight = 0;
            this.Height = formHeight;
            timer1.Enabled = true;
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            //在当前窗体的容器中创建托盘图标 NotifyIcon 的实例
            myNotifyIcon = new NotifyIcon(this.components);
            //指定托盘图标
            myNotifyIcon.Icon =    new Icon("demo.ico");
            //鼠标悬停在托盘图标上方时显示的内容
            myNotifyIcon.Text = "网络呼叫提醒\n";
            //设置关联的上下文菜单
            myNotifyIcon.ContextMenuStrip = this.contextMenuStrip1;
            //显示托盘图标
            myNotifyIcon.Visible = true;
            //添加用户双击任务栏中的托盘图标时触发的事件
            myNotifyIcon.DoubleClick += new EventHandler(myNotifyIcon_DoubleClick);
            //获取本机第一个可用 IP 地址
            IPAddress myIP = (IPAddress)Dns.GetHostAddresses(Dns.GetHostName()).GetValue(0);
            //为了在同一台机器调试，此 IP 也作为默认远程 IP
            textBoxRemoteIP.Text = myIP.ToString();
```

```
        //创建一个线程接收远程主机发来的信息
        Thread myThread = new Thread(new ThreadStart(ReceiveData));
        myThread.Start();
        textBoxSendMessage.Focus();
}
void myNotifyIcon_DoubleClick(object sender, EventArgs e)
{
        ShowThisForm();
}
private void ShowThisForm()
{
        this.Height = 0;
        timer1.Enabled = true;
        this.Show();
}
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
        if (isExit == false)
        {
                //不关闭窗体
                e.Cancel = true;
                //隐藏窗体
                this.Hide();
        }
}
private void  结束程序 ToolStripMenuItem_Click(object sender, EventArgs e)
{
        isExit = true;
        udpClient.Close();
        Application.Exit();
}
private void  呼叫对方 ToolStripMenuItem_Click(object sender, EventArgs e)
{
        ShowThisForm();
}
/// <summary>
///  接收线程
/// </summary>
private void ReceiveData()
{
        //在本机指定的端口接收
        udpClient = new UdpClient(port);
        IPEndPoint remote = null;
        //接收从远程主机发送过来的信息;
        while (true)
        {
                try
                {
                        //关闭 udpClient 时此句会产生异常
                        byte[] bytes = udpClient.Receive(ref remote);
                        string str = Encoding.UTF8.GetString(bytes, 0, bytes.Length);
```

```
                MessageBox.Show(str, string.Format("收到来自[{0}]的呼叫", remote));
            }
            catch
            {
                //退出循环，结束线程
                break;
            }
        }
    }
    /// <summary>
    /// 发送数据到远程主机
    /// </summary>
    private void sendData()
    {
        UdpClient myUdpClient = new UdpClient();
        IPAddress remoteIP;
        if (IPAddress.TryParse(textBoxRemoteIP.Text, out remoteIP) == false)
        {
            MessageBox.Show("远程 IP 格式不正确");
            return;
        }
        IPEndPoint iep = new IPEndPoint(remoteIP, port);
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(textBoxSendMessage.Text);
        try
        {
            myUdpClient.Send(bytes, bytes.Length, iep);
            textBoxSendMessage.Clear();
            myUdpClient.Close();
            textBoxSendMessage.Focus();
        }
        catch (Exception err)
        {
            MessageBox.Show(err.Message, "发送失败");
        }
        finally
        {
            myUdpClient.Close();
        }
    }
    private void buttonSend_Click(object sender, EventArgs e)
    {
        sendData();
    }
    private void timer1_Tick(object sender, EventArgs e)
    {
        if (formHeight < 235)
        {
            this.Height = formHeight;
            formHeight += 5;
        }
        else
```

```
                {
                    timer1.Enabled = false;
                    formHeight = 0;
                }
            }
        }
    }
```

实验报告中要求回答的问题

1. 写出你认为有必要解释的关键步骤和代码。
2. 写出调试中遇到的问题及解决方法。
3. 你从这个实验中受到了哪些启发？

8.3 实验三　文件数据加密与解密

实验目的

1. 练习数据加密与解密的实现方法。
2. 练习将加密数据保存到文件中的方法。
3. 练习从文件中读取加密后的数据的方法。

实验内容

设计一个 Windows 应用程序，实现下列功能：

1. 使用某种加密算法加密窗体上 TextBox 控件中显示的文字，然后将其保存到文件 MyData.txt 中。

2. 程序开始运行时，自动判断加密后的文件是否存在，如果文件存在，则根据以某种方式保存到加密后的文件中的密钥对加密内容进行解密，并将解密后的结果显示在窗体的 TextBox 控件中。程序运行效果如图 8-6 所示。

实验步骤提示

要顺利完成本实验，需要解决四个问题：一是采用哪种算法进行加密；二是如何将加密后的数据保存到文件中；三是如何保存加密密钥；四是如何将加密后的内容读取出来并进行解密处理。

由于只有知道加密密钥才能进行解密，而实验中要求程序开始运行时就判断是否有加密后的文件，并对文件进行解密。因此如何在加密的同时将加密密钥也保存下来是本完成本实验的关键。

这个实验的实现方法有多种，参考解答仅仅是其中比较简单的一种。设计步骤为：

1. 创建一个名为 DataEncrypt 的 Windows 应用程序，修改 Form1.cs 为 FormMain.cs。

2. 向窗体拖放一个 TextBox 控件，一个【保存】按钮，一个【打开】按钮。界面设计效果如图 8-5 所示。

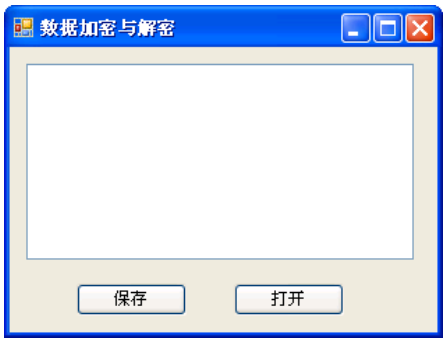


图8-5　实验三设计界面

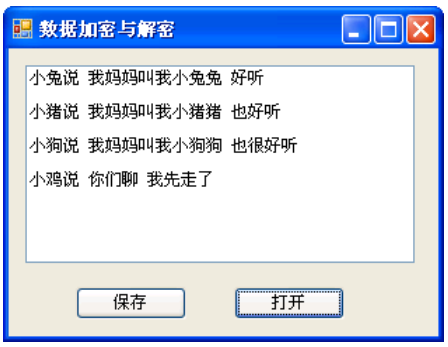


图8-6　实验三运行界面

3. 在【保存】按钮的 Click 事件中，创建 MyData.txt 文件，先将加密密钥写入文件中，然后再写入加密后的文本内容。

4. 编写一个 DecryptFromFile 方法，先读出加密密钥，再读出加密后的文本，并根据读出的密钥进行解密，然后将解密结果显示出来。

采用这种办法，既可以实现保存密钥的目的，也起到了加密的作用。虽然该方法将加密密钥和实际数据保存在一起感觉不太安全，但是即使攻击者打开加密文件，也很难将加密密钥分离出来。当然，如果希望让加密效果更复杂些，也可以先写入几个字节的随机数，然后再保存加密密钥，读出时只需要将这几个字节先读出来，然后再读密钥即可。或者将加密密钥和被加密的数据交叉保存也会使破解的难度增大。对于一般的应用来说，这些方法已经够了。

5. 在窗体的 Load 事件和【打开】按钮的 Click 事件中分别调用 DecryptFromFile 方法，完成解密的功能。

6. 运行程序，输入一些文字信息，单击【保存】，然后打开项目 bin 目录下加密后的 MyData.txt 文件，观察加密后的效果。

实验报告中要求回答的问题

1. 写出你认为有必要解释的关键代码。
2. 写出调试中遇到的问题及解决方法。
3. 你从这个实验中受到了哪些启发？