C#.Net 网络程序开发-Socket 篇

Microsoft.Net Framework 为应用程序访问 Internet 提供了分层的、可扩展的以及受管辖的网络服务，其名字空间 System.Net 和 System.Net.Sockets 包含丰富的类可以开发多种网络应用程序。.Net 类采用的分层结构允许应用程序在不同的控制级别上访问网络，开发人员可以根据需要选择针对不同的级别编制程序，这些级别几乎囊括了 Internet 的所有需要--从 socket 套接字到普通的请求/响应，更重要的是，这种分层是可以扩展的，能够适应 Internet 不断扩展的需要。

抛开 ISO/OSI 模型的 7 层构架，单从 TCP/IP 模型上的逻辑层面上看，.Net 类可以视为包含 3 个层次：请求/响应层、应用协议层、传输层。WebReqeust 和 WebResponse 代表了请求/响应层，支持 Http、Tcp 和 Udp 的类组成了应用协议层，而 Socket 类处于传输层。

传输层位于这个结构的最底层，当其上面的应用协议层和请求/响应层不能满足应用程序的特殊需要时，就需要使用这一层进行 Socket 套接字编程。

而在.Net 中，System.Net.Sockets 命名空间为需要严密控制网络访问的开发人员提供了 Windows Sockets (Winsock) 接口的托管实现。System.Net 命名空间中的所有其他网络访问类都建立在该套接字 Socket 实现之上，如 TCPClient、TCPListener 和 UDPClient 类封装有关创建到 Internet 的 TCP 和 UDP 连接的详细信息；NetworkStream 类则提供用于网络访问的基础数据流等，常见的许多 Internet 服务都可以见到 Socket 的踪影，如 Telnet、Http、Email、Echo 等，这些服务尽管通讯协议 Protocol 的定义不同，但是其基础的传输都是采用的 Socket。

其实，Socket 可以象流 Stream 一样被视为一个数据通道，这个通道架设在应用程序端（客户端）和远程服务器端之间，而后，数据的读取（接收）和写入（发送）均针对这个通道来进行。

可见，在应用程序端或者服务器端创建了 Socket 对象之后，就可以使用 Send/SentTo 方法将数据发送到连接的 Socket,或者使用 Receive/ReceiveFrom 方法接收来自连接 Socket 的数据；

针对 Socket 编程，.NET 框架的 Socket 类是 Winsock32 API 提供的套接字服务的托管代码版本。其中为实现网络编程提供了大量的方法，大多数情况下，Socket 类方法只是将数据封送到它们的本机 Win32 副本中并处理任何必要的安全检查。如果你熟悉 Winsock API 函数，那么用 Socket 类编写网络程序会非常容易，当然，如果你不曾接触过，也不会太困难，跟随下面的解说，你会发觉使用 Socket 类开发 windows 网络应用程序原来有规可寻，它们在大多数情况下遵循大致相同的步骤。

在使用之前，你需要首先创建 Socket 对象的实例，这可以通过 Socket 类的构造方法来实现：

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

其中，addressFamily 参数指定 Socket 使用的寻址方案，socketType 参数指定 Socket 的类型，protocolType 参数指定 Socket 使用的协议。

下面的示例语句创建一个 Socket，它可用于在基于 TCP/IP 的网络（如 Internet）上通讯。

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

若要使用 UDP 而不是 TCP，需要更改协议类型，如下面的示例所示：

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

一旦创建 Socket，在客户端，你将可以通过 Connect 方法连接到指定的服务器，并通过 Send/SendTo 方法向远程服务器发送数据，而后可以通过 Receive/ReceiveFrom 从服务端接收数据；而在服务器端，你需要使用 Bind 方法绑定所指定的接口使 Socket 与一个本地终结点相联，并通过 Listen 方法侦听该接口上的请求，当侦听到用户端的连接时，调用 Accept 完成连接的操作，创建新的 Socket 以处理传入的连接请求。使用完 Socket 后，记住使用 Shutdown 方法禁用 Socket，并使用 Close 方法关闭 Socket。其间用到的方法/函数有：

Socket.Connect 方法:建立到远程设备的连接
public void Connect(EndPoint remoteEP)（有重载方法）
Socket.Send 方法:从数据中的指示位置开始将数据发送到连接的 Socket。
public int Send(byte[], int, SocketFlags);(有重载方法)
Socket.SendTo 方法 将数据发送到特定终结点。
public int SendTo(byte[], EndPoint);（有重载方法）
Socket.Receive 方法:将数据从连接的 Socket 接收到接收缓冲区的特定位置。
public int Receive(byte[],int,SocketFlags);
Socket.ReceiveFrom 方法：接收数据缓冲区中特定位置的数据并存储终结点。
public int ReceiveFrom(byte[], int, SocketFlags, ref EndPoint);
Socket.Bind 方法：使 Socket 与一个本地终结点相关联：
public void Bind(EndPoint localEP);
Socket.Listen 方法：将 Socket 置于侦听状态。
public void Listen(int backlog);
Socket.Accept 方法:创建新的 Socket 以处理传入的连接请求。
public Socket Accept();
Socket.Shutdown 方法:禁用某 Socket 上的发送和接收
public void Shutdown(SocketShutdown how);
Socket.Close 方法:强制 Socket 连接关闭
public void Close();

可以看出，以上许多方法包含 EndPoint 类型的参数，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。网络地址标识网络上的特定设备；端口号标识要连接到的该设备上的特定服务。网络地址和服务端口的组合称为终结点，在 .NET 框架中正是由 EndPoint 类表示这个终结点，它提供表示网络资源或服务的抽象，用以标志网络地址等信息。.Net 同时也为每个受支持的地址族定义了 EndPoint 的子代；对于 IP 地址族，该类为 IPEndPoint。IPEndPoint 类包含应用程序连接到主机上的服务所需的主机和端口信息，通过组合服务的主机 IP 地址和端口号， IPEndPoint 类形成到服务的连接点。

用到 IPEndPoint 类的时候就不可避免地涉及到计算机 IP 地址，.Net 中有两种类可以得到 IP 地址实例：

IPAddress 类：IPAddress 类包含计算机在 IP 网络上的地址。其 Parse 方法可将 IP 地址字符串转换为 IPAddress 实例。下面的语句创建一个 IPAddress 实例：

```
IPAddress myIP = IPAddress.Parse("192.168.1.2");
```

Dns 类：向使用 TCP/IP Internet 服务的应用程序提供域名服务。其 Resolve 方法查询 DNS 服务器以将用户友好的域名（如"host.contoso.com"）映射到数字形式的 Internet 地址（如 192.168.1.1）。Resolve 方法 返回一个 IPHostEnty 实例，该实例包含所请求名称的地址和别名的列表。大多数情况下，可以使用 AddressList 数组中返回的第一个地址。下面的代码获取一个 IPAddress 实例，该实例包含服务器 host.contoso.com 的 IP 地址。

```
IPHostEntry ipHostInfo = Dns.Resolve("host.contoso.com");
IPAddress ipAddress = ipHostInfo.AddressList[0];
```

你也可以使用 GetHostName 方法得到 IPHostEntry 实例：

```
IPHosntEntry hostInfo=Dns.GetHostByName("host.contoso.com")
```

在使用以上方法时，你将可能需要处理以下几种异常：

SocketException 异常：访问 Socket 时操作系统发生错误引发

ArgumentNullException 异常：参数为空引用引发

ObjectDisposedException 异常：Socket 已经关闭引发

在掌握上面得知识后，下面的代码将该服务器主机（ host.contoso.com 的 IP 地址与端口号组合，以便为连接创建远程终结点：

```
IPEndPoint ipe = new IPEndPoint(ipAddress,11000);
```

确定了远程设备的地址并选择了用于连接的端口后，应用程序可以尝试建立与远程设备的连接。下面的示例使用现有的 IPEndPoint 实例与远程设备连接，并捕获可能引发的异常：

```
try {
s.Connect(ipe);//尝试连接
}
//处理参数为空引用异常
catch(ArgumentNullException ae) {
Console.WriteLine("ArgumentNullException : {0}", ae.ToString());
}
//处理操作系统异常
catch(SocketException se) {
Console.WriteLine("SocketException : {0}", se.ToString());
}
catch(Exception e) {
Console.WriteLine("Unexpected exception : {0}", e.ToString());
}
```

需要知道的是：Socket 类支持两种基本模式：同步和异步。其区别在于：在同步模式中，对执行网络操作的函数（如 Send 和 Receive）的调用一直等到操作完成后才将控制返回给调用程序。在异步模式中，这些调用立即返回。

另外，很多时候，Socket 编程视情况不同需要在客户端和服务器端分别予以实现，在客户端编制应用程序向服务端指定端口发送请求，同时编制服务端应用程序处理该请求，这个过程在上面的阐述中已经提及；当然，并非所有的 Socket 编程都需要你严格编写这两端程序；视应用情况不同，你可以在客户端构造出请求字符串，服务器相应端口捕获这个请求，交由其公用服务程序进行处理。以下事例语句中的字符串就向远程主机提出页面请求：

```
string Get = "GET / HTTP/1.1\r\nHost: " + server + "\r\nConnection: Close\r\n\r\n";
```

远程主机指定端口接受到这一请求后，就可利用其公用服务程序进行处理而不需要另行编制服务器端应用程序。

综合运用以上阐述的使用 Visual C#进行 Socket 网络程序开发的知识，下面的程序段完整地实现了 Web 页面下载功能。用户只需在窗体上输入远程主机名（Dns 主机名或以点分隔的四部分表示法格式的 IP 地址）和预保存的本地文件名，并利用专门提供 Http 服务的 80 端口，就可以获取远程主机页面并保存在本地机指定文件

中。如果保存格式是.htm 格式，你就可以在 Internet 浏览器中打开该页面。适当添加代码，你甚至可以实现一个简单的浏览器程序。

实现此功能的主要源代码如下：

```
//"开始"按钮事件
private void button1_Click(object sender, System.EventArgs e) {
//取得预保存的文件名
string fileName=textBox3.Text.Trim();
//远程主机
string hostName=textBox1.Text.Trim();
//端口
int port=Int32.Parse(textBox2.Text.Trim());
//得到主机信息
IPHostEntry ipInfo=Dns.GetHostByName(hostName);
//取得 IPAddress[]
IPAddress[] ipAddr=ipInfo.AddressList;
//得到 ip
IPAddress ip=ipAddr[0];
//组合出远程终结点
IPEndPoint hostEP=new IPEndPoint(ip,port);
//创建 Socket 实例
Socket socket=new
Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
try
{
//尝试连接
socket.Connect(hostEP);
}
catch(Exception se)
{
MessageBox.Show("连接错误"+se.Message,"提示信息
,MessageBoxButtons.RetryCancel,MessageBoxIcon.Information);
}
//发送给远程主机的请求内容串
string sendStr="GET / HTTP/1.1\r\nHost: " + hostName +
"\r\nConnection: Close\r\n\r\n";
//创建 bytes 字节数组以转换发送串
byte[] bytesSendStr=new byte[1024];
//将发送内容字符串转换成字节 byte 数组
bytesSendStr=Encoding.ASCII.GetBytes(sendStr);
try
{
```

```
//向主机发送请求
socket.Send(bytesSendStr,bytesSendStr.Length,0);
}
catch(Exception ce)
{
MessageBox.Show("发送错误:"+ce.Message,"提示信息
,MessageBoxButtons.RetryCancel,MessageBoxIcon.Information);
}
//声明接收返回内容的字符串
string recvStr="";
//声明字节数组，一次接收数据的长度为 1024 字节
byte[] recvBytes=new byte[1024];
//返回实际接收内容的字节数
int bytes=0;
//循环读取，直到接收完所有数据
while(true)
{
bytes=socket.Receive(recvBytes,recvBytes.Length,0);
//读取完成后退出循环
if(bytes〈=0)
break;
//将读取的字节数转换为字符串
recvStr+=Encoding.ASCII.GetString(recvBytes,0,bytes);
}
//将所读取的字符串转换为字节数组
byte[] content=Encoding.ASCII.GetBytes(recvStr);
try
{
//创建文件流对象实例
FileStream fs=new
FileStream(fileName,FileMode.OpenOrCreate,FileAccess.ReadWrite);
//写入文件
fs.Write(content,0,content.Length);
}
catch(Exception fe)
{
MessageBox.Show("文件创建/写入错误:"+fe.Message,"提示信息
",MessageBoxButtons.RetryCancel,MessageBoxIcon.Information);
}
//禁用 Socket
socket.Shutdown(SocketShutdown.Both);
//关闭 Socket
socket.Close();
```

```
}
}
```