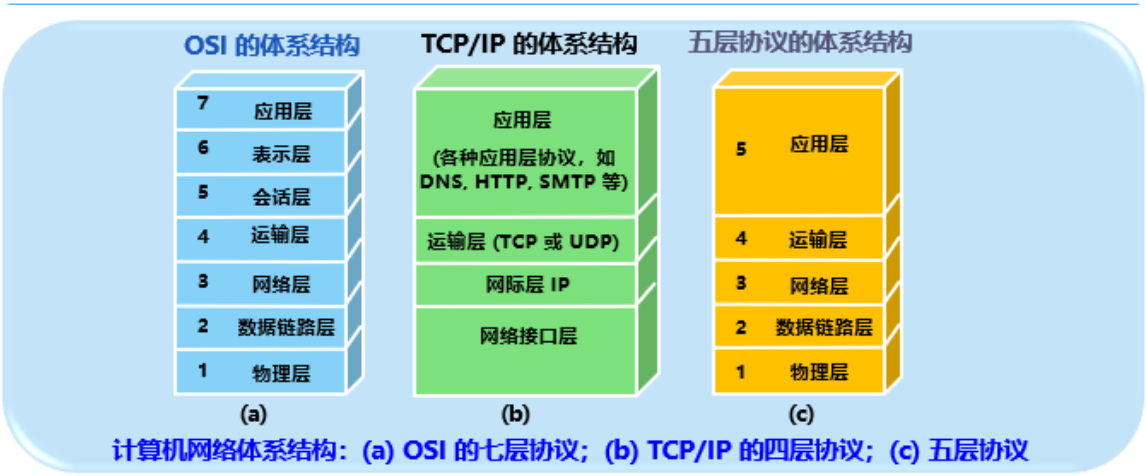
# C#网络应用编程

# 第一章

## 三种网络体系结构




TCP和UDP协议位于传输层

## socket套接字

**套接字位于应用层和传输层之间**，一个套接字实例中保存有本机的IP地址和端口、对方的IP地址和端口、双方通信采用的网络协议等信息。

三种类型：

1. 流式套接字：实现面向连接的TCP通信
2. 数据报套接字：实现无连接的UDP通信
3. 原始套接字：实现IP数据包通信

## TCP应用编程

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）

服务端使用TcpListener类，客户端使用TcpClient类；或者全部使用Socket实现

## UDP应用编程

UdpClient类是在UDP层面对套接字编程的进一步封装。

## IPAddress类

提供了对IP地址的转换和处理功能

- Parse方法将IP地址字符串转换为IPAddress的实例：`IPAddress ip = IPAddress.Parse("143.24.20.36");`

- AddressFamily属性可判断该IP地址是IPv6还是IPv4：`if (ip.AddressFamily == AddressFamily.InterNetworkV6)`

## IPEndPoint类

描述应用程序连接到主机上的服务所需的主机和端口信息

常见的构造函数：`public IPEndPoint(IPAddress address, int port);`其中第一个参数指定IP地址，第二个参数指定端口号

## IPHostEntry类

IPHostEntry类将一个域名系统（DNS）的主机名与一组别名和一组匹配的IP地址关联。该类一般和Dns类一起使用

- AddressList属性：获取或设置与主机关联的IP地址列表（包括IPv4和IPv6）
- HostName属性：包含了指定主机的主机名

## DNS类

- GetHostEntry静态方法:用于在DNS服务器中查询与某个主机名或IP地址关联的IP地址列表：`IPAddress[] ips = Dns.GetHostEntry("news.sohu.com").AddressList;`
- GetHostAddresses方法：获取指定主机的IP地址，该方法返回一个IPAddress类型的数组：`IPAddress[] ips = Dns.GetHostAddresses("www.cctv.com");`（如果括号内是ip地址，则返回此地址；如果为空，返回本机所有ipv4和ipv6地址）
- GetHostName方法：用于获取本机主机名：`string hostname = Dns.GetHostName( );`

## 网卡信息检测相关类

- NetworkInterface类：提供了网络适配器的配置和统计信息。可以利用这个类检测本机有多少个网络适配器、网络适配器型号以及网络适配器的速度等：`NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();`

| 属性及方法 | 说明 |
| --- | --- |
| Name属性 | 获取网络适配器的名称 |
| Speed属性 | 获取网络适配器的速度（bit/秒） |
| GetAllNetworkInterfaces方法 | 返回描述本地计算机上的所有网络适配器对象 |
| GetIPProperties方法 | 回描述此网络适配器配置的对象 |
| GetIsNetworkAvailable方法 | 指示是否有任何可用的网络连接 |
| GetPhysicalAddress方法 | 返回此适配器的媒体访问控制(MAC)地址 |
| Supports方法 | 指示接口是否支持指定的协议（IPv4或IPv6） |

- IPInterfaceProperties类：检测本机所有网络适配器支持的各种地址，IPInterfaceProperties类是抽象类，不能实例化。可以通过NetworkInterface对象的GetIPProperties()获得其实例

| 属性及方法 | 说　明 |
| --- | --- |
| **AnycastAddresses**属性 | 获取分配给此接口的任意广播IP地址 |
| **DhcpServerAddresses**属性 | 获取此接口的动态主机配置协议（**DHCP**）服务器的地址 |
| **DnsAddresses**属性 | 获取此接口的域名系统（DNS）服务器的地址 |
| **DnsSuffix**属性 | 获取与此接口关联的域名系统（DNS）后缀 |
| **GatewayAddresses**属性 | 获取此接口的网关地址 |
| **MulticastAddresses**属性 | 获取分配给此接口的多路广播地址 |
| **UnicastAddresses**属性 | 获取分配给此接口的单播地址 |
| **GetIPv4Properties**方法 | 获取此网络接口的Internet协议版本4（IPv4）配置数据 |
| **GetIPv6Properties**方法 | 获取此网络接口的Internet协议版本6（IPv6）配置数据 |

## 网络流量检测相关类

- IPGlobalProperties类：提供了本地计算机网络连接和通信统计数据的信息：`IPGlobalProperties properties =IPGlobalPropeties.GetIPGlobalProperties( );`

| 名　　称 | 说　　明 |
| --- | --- |
| GetActiveTcpConnections | 返回有关本地计算机上的 Internet 协议版本 4 (IPV4) 传输控制协议 (TCP) 连接的信息 |
| GetActiveTcpListeners | 返回有关本地计算机上的 Internet 协议版本 4 (IPV4) 传输控制协议 (TCP) 侦听器的终结点信息 |
| GetActiveUdpListeners | 返回有关本地计算机上的 Internet 协议版本 4 (IPv4) 用户数据报协议 (UDP) 侦听器的信息 |
| GetIPv4GlobalStatistics | 提供本地计算机的 Internet 协议版本 4 (IPv4) 统计数据 |
| GetIPv6GlobalStatistics | 提供本地计算机的 Internet 协议版本 6 (IPv6) 统计数据 |
| GetTcpIPv4Statistics | 提供本地计算机的传输控制协议/Internet 协议版本 4 (TCP/IPv4) 统计数据 |
| GetTcpIPv6Statistics | 提供本地计算机的传输控制协议/Internet 协议版本 6 (TCP/IPv6) 统计数据 |

- TcpConnectionInformation类：提供本机传输控制协议(TCP)连接的信息：`TcpConnectionInformation[] connections = properties.GetActiveTcpConnections();`

# 第三章

## 进程

- 进程是操作系统级别的一个基本概念，可以将其简单地理解为“正在运行的程序”。
- 进程是资源调度和分配的基本单位
- 进程之间是相互独立的。
- 在操作系统级别的管理中，利用Process类可启动、停止本机或远程进程。

# 线程

- 将一个进程划分为若干个独立的执行流，每一个执行流均称为一个线程
- 线程是CPU调度和分配的基本单位
- 一个进程中既可以只包含一个线程，也可以同时包含多个线程。
- 线程共享进程的资源

线程并发执行：线程宏观上并行、微观上串行

线程异步执行：各个线程之间执行时各自运行，无前后关系，无相互等待，执行先后不可知

# 逻辑内核数

利用System.Environment类提供的静态ProcessorCount属性，可获取本机可用逻辑内核的数量

# 进程管理（Process类）

## 启动进程

1. 创建一个Process组件的实例，例如：`Process p = new Process();`
2. 指定要运行的应用程序名以及传递的参数：

   ```
   p.StartInfo.FileName = "文件名";
   p.StartInfo.Arguments = "参数";
   p.StartInfo.WindowStyle = ProcessWindowStyle.Normal;
   ```

3. 调用该实例的Start方法启动该进程：`myProcess.Start();`

或者：`Process.Start(“Notpad.exe”);`

## 停止进程

1. Kill方法和CloseMainWindow方法：前者用于强行终止进程，后者只是“请求”终止进程。
2. HasExited属性：HasExited属性用于判断启动的进程是否已停止运行。
3. WaitForInputIdle方法：仅适用于具有用户界面的进程，它可以使Process等待关联进程进入空闲状态。
4. WaitForExit方法：设置等待关联进程退出的时间
5. ExitCode属性和ExitTime属性：ExitCode属性用于获取关联进程终止时指定的值 （0成功 非零错误）
   ExitTime属性用于获取关联进程退出的时间。
   这两个属性只能在HasExited属性为true时才能检测。
6. EnableRaisingEvents属性
   EnableRaisingEvents属性用于获取或设置在进程终止时是否应引发Exited事件。当关联进程终止时引发Exited事件则为true，否则为false

# 获取所有进程信息

Process类的GetProcesses静态方法用于创建新的Process数组，并将该数组与本地计算机上的所有进程资源相关联。

- 获取本地计算机的所有进程：`Process[] myProcesses = Process.GetProcesses();`
- 获取远程计算机的所有进程：

```
Process[] myProcesses =Process.GetProcesses (remoteMachineName);
```

# 获取指定进程信息

- Process的GetProcessById静态方法会自动创建Process对象，并将其与本地计算机上的进程相关联，同时将进程Id传递给该Process对象。 `Process  p = Process.GetProcessesById(5152);`
- GetProcessesByName静态方法返回一个包含所有关联进程的数组。

  获取本地计算机上指定名称的进程：

  ```
  Process[] myProcesses = Process.GetProcessesByName("进程名称");
  ```

  获取远程计算机上指定名称的进程：

  ```
  Process[] myProcesses =Process.GetProcessesByName( "远程进程名称",remoteMachineName);
  ```

# 线程管理

## 主线程和辅助线程

- 无论何种类型的应用程序，当将这些程序作为进程来运行时，系统都会为该进程创建一个默认的线程，该线程称为主线程。
- 主线程用于执行Main方法中的代码，当Main方法返回时，主线程也自动终止。
- 在一个进程中，除了主线程之外的其他线程都称为辅助线程。

## 前台线程与后台线程

- 一个线程要么是前台线程要么是后台线程。
- 后台线程不会影响进程的终止，而前台线程则会影响进程的终止。
- 利用Thread对象的IsBackground属性，可以设置或判断一个线程是后台线程还是前台线程
- 将某个线程的IsBackground属性设置为true，使其变为后台线程。
- 默认情况下，属于托管线程池的线程都是后台线程（即其IsThreadPoolThread属性为true），通过创建并启动新的Thread对象而生成的线程都是前台线程。

## 创建和启动线程

- 通过Thread创建一个单独的线程，常用形式为：
  `Thread t=new Thread(<方法名>);`
- 线程通过委托来实现的，委托由定义的方法是否带参数而定。
  不带参数方法使用ThreadStart委托：`Thread t3 = new Thread(new ThreadStart(MethodC));`
  带参数方法使用ParameterizedThreadStart委托：`Thread t3 = new Thread(new ParameterizedThreadStart(MethodC));`
- Thread创建的线程默认为前台线程。
- 线程启动调用该实例的Start方法
  `t.Start();` //调用不带参数的方法
  `t.Start(“abc”);` //调用带参数的方法

## 线程终止和取消

- 第1种方法是先设置一个修饰符为volatile的布尔型的字段表示是否需要正常结束该线程，称为终止线程。（该字段将不再被编译器优化。这样可以确保该字段在任何时间呈现的都是最新的值。）
- 第2种方法是在其他线程中调用Thread实例的Abort方法终止当前线程，该方法的最终效果是强行终止该线程的执行，属于非正常终止的情况，称为取消线程的执行。

## 休眠进程

调用Thread类提供的静态Sleep方法，可使当前线程暂停一段时间：`Thread.Sleep(1000);`（Sleep方法是暂停的是该语句所在的线程，而不是其他线程；并且无法从一个线程中暂停其他的线程。

## 获取或设置线程的优先级

创建线程时，默认优先级为Normal

使用下面的方法可为线程赋予较高的优先级：

```
Thread t1 = new Thread(MethodName);
t1.priority = ThreadPriority.AboveNormal
```

## 线程池

1. 托管线程池中的线程都是后台线程。
2. 添加到线程池中的任务不一定会立即执行。
3. 线程池可自动重用已创建过的线程。一旦池中的某个线程完成任务，它将返回到等待线程队列中，等待被再次使用，而不是直接销毁它。
4. 开发人员可设置线程池的最大线程数。
5. 从.NET框架4.0开始，线程池的默认大小由多个因素决定;线程池中的线程都是利用多核处理技术来实现的。
6. 在传统的编程模型中，开发人员一般是直接用ThreadPool.QueueUserWorkItem方法向线程池中添加工作项。
   ```
   ThreadPool.QueueUserWorkItem(new WaitCallback(Method1));
   ThreadPool.QueueUserWorkItem(new WaitCallback(Method2));
   ```
7. ThreadPool只提供了一些静态方法，不能通过创建该类的实例来使用线程池。

## 线程池多线程编程中的资源同步

- 同步执行： 执行某语句时，在该语句完成之前不会执行其后面的代码，这种执行方式称为同步执行。
- 异步执行:执行某语句时，不管该语句是否完成，都会继续执行其后面的语句，这种执行方式称为异步执行。
- 在某个线程中启动另一个或多个线程后，这些线程会同时执行，称为并行（准确说是并发）。
- 并行执行的多个线程同时访问某些资源时，必须考虑如何让多个线程保持同步。

- 死锁的典型例子是两个线程都停止响应，并且都在等待对方完成，从而导致任何一个线程都不能继续执行。
- 争用就是程序结果取决于两个或多个线程中的哪一个先到达某一特定代码块是出现的一种错误。

## 实现资源同步的方式

- 用volatile修饰符锁定公共或私有字段。
  利用该修饰符可直接访问内存中的字段，而不是将字段缓存在某个处理器的寄存器中。这样做的好处是所有处理器都可以访问该字段最新的值。
- 用Interlocked类提供的静态方法锁定局部变量。
  System.Threading.Interlocked类通过加锁和解锁提供了原子级别的静态操作方法，对并行执行过程中的某个局部变量进行操作时，可采用这种办法实现同步。
- 用lock语句锁定代码块
  直接用C#提供的lock语句将包含局部变量的代码块锁定，退出被锁定的代码块后会自动解锁。

## WPF中的多线程编程模型

默认情况下，.NET框架都不允许在一个线程中直接访问另一个线程中的控件

为了解决死锁以及异步执行过程中的同步问题，WPF中的每个元素（包括根元素）都有一个Dispatcher属性

要在后台线程中与用户界面交互，可以通过向WPF控件的Dispatcher注册工作项来完成。常用方法有两种：Invoke方法和InvokeAsync方法。

1. Invoke方法是同步调用，即直到在线程池中实际执行完该委托它才返回。
2. InvokeAsync是异步调用。

# 应用程序域

定义：一个主进程中，可包含一个或多个“子进程”，每个“子进程”所占用的内存范围（或者叫边界）都称为一个应用程序域。

## 应用程序域与线程的关系

1. 应用程序域为安全性、版本控制、可靠性和托管代码的卸载形成隔离边界，执行应用程序时，所有托管代码均加载到一个应用程序域中，由一个或多个托管线程来运行。
2. 应用程序域和线程之间不具有一对一的相关性。
3. 应用程序域之间是相互隔离的，一个应用程序域无法直接访问另一个应用程序域的资源。

## 应用程序域与进程的关系

1. 可将应用程序进程中的每个应用程序域都看作是一个“子进程”。
2. 一个进程既可以只包含一个应用程序域，也可以同时包含多个相互隔离的应用程序域。
3. 多进程是操作系统级别使用的功能，资源消耗较大，细节控制复杂；应用程序域是在应用程序级别使用的功能，比直接用多进程来实现进程管理速度快、资源消耗少而且更安全，是为应用程序开发人员提供的轻量级的进程管理。

## 程序集

.NET框架应用程序的生成块，为CLR提供识别和实现类型所需的信息。程序集包含模块、模块包含类型、类型包含成员

## 反射

提供封装程序集、模块和类型的对象。

1. 利用反射可以动态创建类的实例、将类绑定到对象或从现有对象获得类，并调用类的方法或访问其字段和属性。
2. 利用反射可以从程序集中找到元数据信息。

# 第四章 数据流与数据的加密和解密

## 数据编码和解码

编码：将字符序列转换为字节序列的过程。
解码：将字节序列转换为字符序列的过程。

常见的字符集编码方式：

- ASCII
- Unicode
- UTF-8
- GB2312和GB18030

### Encoding类

| | |
|---|---|
| Default属性 | 获取系统的当前ANSI代码页的编码 |
| BodyName属性 | 获取可与邮件正文标记一起使用的编码名称。如果当前Encoding无法使用，则为空字符串 |
| HeaderName属性 | 获取可与邮件标题标记一起使用的编码名称。如果当前Encoding无法使用，则为空字符串 |
| Unicode属性 | 获取Unicode格式的编码（UTF-16） |
| UTF8属性 | 获取Unicode格式的编码（UTF-8） |
| ASCII属性 | 获取ASCII字符集的编码 |
| Convert方法 | 将字节数组从一种编码转换为另一种编码 |
| GetBytes方法 | 将一组字符编码为一个字节序列 |
| GetString方法 | 将一个字节序列解码为一个字符串 |
| GetEncoding方法 | 返回指定格式的编码 |

获取所有编码名称及其描述信息：使用Encoding类静态的GetEncodings方法可得到一个包含所有编码的EncodingInfo类型的数组

```
foreach ( EncodingInfo ei in Encoding.GetEncodings( ))
{
    Encoding en = ei.GetEncoding( );
}
```

利用Encoding类的Convert方法可将字节数组从一种编码转换为另一种编码，转换结果为一个byte类型的数组

```
byte[] b = Encoding.Convert(unicode, utf8, unicode.GetBytes(s));
```

## 数据流

1. FileStream类、MemoryStream类、NetworkStream类、CryptoStream类
2. 用于文本读写的StreamReader和StreamWriter类
3. 用于二进制读写的BinaryReader和BinaryWriter类等。

1.利用构造函数创建FileStream对象：

```
FileStream (string path, FileMode mode, FileAccess access)
```

- FileMode枚举的可选值：CreateNew、Create、Open、OpenOrCreate、Truncate、Append
- FileAccess枚举的可选值有：Read、Write、ReadWrite

2.利用File类创建FileStream对象：

- 利用OpenRead方法创建仅读取的文件流；
  ```
  FileStream fs= File.OpenRead(@"D:\ls\File1.txt");
  ```
- 利用OpenWrite方法创建仅写入的文件流。
  ```
  FileStream fs= File.OpenWrite(@"D:\ls\File1.txt");
  ```

# 内存流

利用System.IO命名空间下的MemoryStream类，可以按内存流的方式对保存在内存中的字节数组进行操作：

- 利用Write方法将字节数组写入到内存流中
- 利用Read方法将内存流中的数据读取到字节数组中

MemoryStream的用法与文件流的用法相似，支持对数据流的查找和随机访问：

- CanSeek属性值默认为true
- 通过Position属性获取内存流的当前位置。

MemoryStream的使用场合：

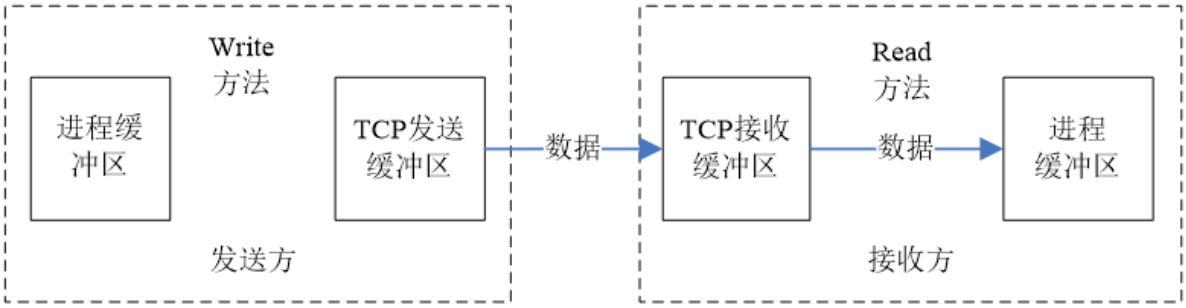
- 在数据加密以及对长度不定的数据进行缓存等场合。内存流的容量可自动增长
- 从数据库中读取照片类型的数据，显示到图形控件内时。字节数组→内存流→ImageSource

# 网络流

可以将NetworkStream看作在数据源和接收端之间架设了一个数据通道，读取和写入数据就可以针对这个通道来进行。

> 注意：NetworkStream类仅支持面向连接的套接字。


## 利用网络流发送及接收TCP数据的流程：




1. Write方法负责将字节数组从进程缓冲区发送到本机的TCP发送缓冲区
2. 然后TCP/IP协议栈再通过网络适配器把数据真正发送到网络上
3. 最终到达接收方的TCP接收缓冲区。

### 获取NetworkStream对象

1. 利用TcpClient对象的GetStream方法得到网络流对象。
```
TcpClient tcpClient=new TcpClient( );
tcpClient.Connect("www.abcd.com", 51888);
NetworkStream networkStream = tcpClient.GetStream( );
```
2. 利用Socket得到网络流对象。
```
NetworkStream myNetworkStream = new NetworkStream(mySocket);
```

### 发送数据

Write方法为同步方法，在将数据写入到网络流之前，Write方法将一直处于阻塞状态，直到发送成功或者返回异常为止

```
byte[] writeBuffer = Encoding.UTF8.GetBytes("Hello");
   myNetworkStream.Write(writeBuffer, 0, writeBuffer.Length);
```

### 接收数据

调用Read方法将数据从接收缓冲区读取到进程缓冲区，完成读取操作

```
numberOfBytesRead =  myNetworkStream.Read(readBuffer, 0,readBuffer.Length);
```

## StreamReader和StreamWriter类

### 1.创建StreamReader和StreamWriter的实例

- 如果数据来源是文件流、内存流或者网络流，可以利用StreamReader和StreamWriter对象的构造函数得到读写流。
```
NetworkStream networkStream = client.GetStream( );

StremReader sr = new StremReader (networkStream);
StreamWriter sw = new StreamWriter (networkStream);
```
- 如果需要处理的是文件流，还可以直接利用文件路径创建StreamWriter对象。
```
StreamWriter sw= new StreamWriter ("C:\\file1.txt");
```
- 与该方法等价的有File及FileInfo类提供的CreateText方法。
```
StreamWriter sw = File.CreateText ("C:\\file1.txt");
```

### 2.读写文本数据

- 利用StreamWriter类的Write和WriteLine方法写入文本数据
- 利用StreamReader类的ReadLine方法读取文本数据。

> 注意：不要忘记用Close方法关闭流，或者用using语句让系统自动关闭它。

## BinaryReader和BinaryWriter类

System.IO命名空间还提供了BinaryReader和BinaryWriter类以二进制模式读写流，更方便于对图像文件、压缩文件等二进制数据进行操作。

对于BinaryReader中的每个读方法，在BinaryWriter中都有一个与之对应的写方法。

# 第五章 异步编程

## Lambda表达式

Lambda表达式是一个可用于创建委托或表达式树类型的匿名函数。

### 基本用法定义

(输入参数列表)⇒{表达式或语句块}

```
x ⇒ x * x
```

### 在LINQ to Objects中使用Lambda表达式

```
var q2 = from i in numberList where i < 4 select i;
```

## Action和Func委托

- Action<[T1,T2,……,T16]>: 封装不带返回值的方法，即返回值是void

```
class Program
{
    static void Main(string[] args)
    {
        Action<string> a = ShowMessage;
        a("OK");
        Show Message();
    }
    private static void ShowMessage(string message)
    {
        Console.WriteLine(message);
    }}
```

```
class Program
{
    static void Main(string[] args)
    {
        //使用匿名方法实现
        Action a=() => Console.WriteLine("OK");
        Action<string> b = (s)=> Console.WriteLine(s);
        b("OK");
        Show Message();
    }
```

- Func<[T1,T2,……,T16,] TResult>：封装带返回值的方法，返回值类型为Tresult

### 元组（Tuple类）

元组是一种数据结构，其中的元素具有固定的数目和序列。

提供对数据集的轻松访问和操作

在.NET框架中，可通过Tuple. Create方法直接创建具有1到7个元素的元组，另外，还可以通过嵌套的元组创建具有更多元素的对象。元组中的元素可通过Tuple对象的ItemN（N=1,2,3,……,7）属性得到。

优点：用一个参数就能把多个值传递给某个方法

## 异步编程的实现方式

### 传统的异步编程模型（APM）

使用Item设计模式的异步操作是通过名为 Begin--- 和 End---的两个方法来实现的，这两个方法分别指代开始和结束异步操作。
例如：网络流对象的Begin Read、End Read方法

> 异步控制复杂，目前该技术已经被淘汰。

### 基于事件的异步编程设计模式（EAP）

该模式用事件驱动模型实现异步方法。

通常由有一个或若干个--- Async方法和一个对应的---Completed 事件。

例如：Background Worker组件、 Picture Box控件等。

> 目前该技术已经被淘汰。

### 基于任务的异步模式（TAP）

.NET 4.0框架，异步编程建议的异步编程技术

### 改进的基于任务的异步模式（async、await、Task. Run和TAP）

TAP和C#关键字的结合使用，目前建议采用的异步编程技术。

## 任务

任务(Task) 表示一个异步操作。任务运行的时候需要使用线程，通用语言运行时（CLR）会创建必要的线程来支持任务执行的需求。

### Task类

表示一个没有返回值的异步操作：`Task t = Task. Run( () =>`
`{ Console.WriteLine("this is a task"); } );`

### Task类

表示一个可以返回值的异步操作：

```
var t = Task<int>.Run( () =>
{
      int num=5;  num=DateTime.Now.Year+5;
      return num;
 });
```

### Task. Delay方法

- Delay(Int32)  //延时指定的毫秒数
- Delay(Timespan)  //延时指定的时间（年、月、日、时、分、秒、毫秒等）
- Delay(Int32, Cancellation Token)  //延时指定的毫秒数后取消任务操作
- Delay(Timespan, Cancellation Token)  //延时指定的时间后取消任务操作

## 异步操作关键字

带async修饰的方法称为异步方法。带async修饰的事件处理程序称为异步事件处理程序。二者也可以统称为异步方法。

> 异步方法的命名约定：方法后面以"Async"做为后缀

### async修饰符

对于普通方法：

- 如果方法没有返回值，则用async和Task共同签名。
- 如果方法有返回值，则用async和Task共同签名。

对于事件处理程序：

- 用async和void共同签名。

### await运算符

- await运算符表示等待异步执行的结果。实质上对方法的返回值进行操作。
- 使用await异步等待任务完成时，不会执行其后面的代码，但也不会影响用户对UI的操作。
- await运算符必须放在异步方法内部。

# 创建任务

## 定义任务执行的方法

- 用普通方法定义任务

```
public void Method1() {...}
private async void Thread. Sleep(...)
{
    await Task. Run(()=>Method1());
}
```

普通方法要用Task. Run方法去调用，或者用Task、Task类的构造函数显示创建Task实例，然后再启动。

- 用异步方法定义任务

```
public async Task Method1Async() {.... }
private async void Thread. Sleep(...)
{
    await Method1Async();
}
```

- 用匿名方法定义任务

```
private async void Thread. Sleep(...)
{
    await  Task. Run(()=>
{
Console.WriteLine("acai");
});
}
```

## 利用Task. Run方法隐式创建和执行任务

Task. Run方法是.NET框架4.5提供的功能，它会在线程池中用单独的线程执行某个任务。

Run方法的几种形式：

- Run(Funk)  用默认调度程序在线程池中执行不带返回值的任务
- Run(Funk<Task>) 用默认调度程序在线程池中执行带返回值的任务

- Run(Funk, Cancellation Token)  执行任务过程中可侦听取消通知
- Run(Funk<Task>, Cancellation Token) 执行任务过程中可侦听取消通知

## 利用async和await关键字隐式创建异步任务

async和await关键字是C# 5.0提供的功能，仅包含async和await关键字的异步方法不会创建新线程，它只是表示在当前线程中异步执行指定的任务。

## 利用WPF控件的调度器隐式创建和执行任务

只能在WPF应用程序中使用，且异步过程间同步实现比较复杂。

## 通过显式调用Task或Task的构造函数创建任务

用法：先创建任务，然后调用Start方法启动任务。与Thread的用法类似。

- Task类的构造函数。

```
//不传递对象的任务
public Task(Action action， Cancellation Token
cancellationToken，TaskCreationOptions creationOptions)
//传递对象的任务
public Task(Action<object> action，object state,
Cancellation Token cancellationToken,
TaskCreationOptions creationOptions)
```

- Task类的构造函数

```
Task task = new Task(TaskMethod);
task.Start();
//任务执行的方法
static void TaskMethod()
{
   for (int i = 0; i < 10; i++)
    {  Console.WriteLine("Running in a task.”+i);
       Thread. Sleep(500);
    }
}
```

# 取消或终止任务的执行

.NET框架引入的Cancellation TokenSource类和Cancellation Token结构用于协同实现多个线程、线程池工作项或Task对象的取消操作。

## Cancellation TokenSource类和Cancellation Token结构

- Cancellation TokenSource用于创建取消通知，称为**取消源**。

  ```
  CancellationTokenSource cts = new CancellationTokenSource();
  ```

- Cancellation Token结构用于传播应取消操作的通知，称为**取消令牌**。

  ```
  Cancellation Token ct=cts.Token;
  ```

> CancellationTokenSource对象的Cancel方法发出取消通知，然后将Cancellation Token对象的IsCancellationRequested属性设置为true。

## 执行任务的方法接收到取消通知后，可以用以下方式之一终止操作

- 在任务代码中，简单地从委托中返回，任务状态值为RanToCompletion（正常完成）。

  可以通过任务的Status属性获得任务状态。类似线程终止时设定的布尔变量方式

- 在任务代码中，引发OperationCanceledException异常，并将其传递到在其上请求了取消的标记。

  调用Cancellation Token对象的ThrowIfCancellationRequested方法

## 冷任务和热状态

用Task类或者Task类的构造函数显式创建的任务称为冷任务（cold task）。

> 冷任务必须通过Start方法来启动。

任务在生命周期内的执行情况称为热状态。

## Status属性和TaskStatus枚举

利用任务实例的Status属性获取任务执行的状态。任务执行的状态用TaskStatus枚举表示。

TaskStatus的枚举值有：

- Created：该任务已初始化，但尚未进入调度计划。
- WaitingForActivation：该任务已进入调度计划，正在等待被调度程序激活。
- WaitingToRun：该任务已被调度程序激活，但尚未开始执行。
- Running：该任务正在运行，但尚未完成。
- RanToCompletion：该任务已成功完成。
- Canceled：该任务由于被取消而完成（任务自身引发OperationCanceledException异常，或者在该任务执行之前调用方已向该任务的Cancellation Token发出了信号）。
- Faulted：该任务因为出现未经处理的异常而完成。
- WaitingForChildrenToComplete：该任务本身已完成，正等待附加的子任务完成。

## 任务完成情况相关的属性

- IsCompleted属性：表示任务是否完成。任务状态为RanToCompletion时，任务成功完成。
- IsCanceled属性：表示任务是否因为取消而完成。
- IsFaulted属性：表示任务是否因出现未处理的异常而完成。
- 取消和完成之间的关系：取消是向任务传递一种信号，希望任务尽快结束。

完成是任务执行结束了。

## 报告任务执行的进度

**可以用Progress类报告任务执行的进度。**
Progress类是通过IProgress接口来实现的，该类的声明方式如下：

```
public class Progress<T> : IProgress<T>
{
   public Progress();
   public Progress(Action<T> handler);
   protected virtual void OnReport(T value);
   public event EventHandler<T> ProgressChanged; //事件
}
```

# 第六章 并行编程

从业务实现的角度看并行策略：并行编程模型分为**数据并行**与**任务并行**。

从硬件实现的角度看并行策略：并行又分为**单机多核并行**和**多机多核并行**。

# 任务并行库（TPL）及其分类

TPL分类：

- 数据并行：对源集合或者数组中的元素同时执行相同操作。借助Parallel类的For或Foreach方法来实现。
- 任务并行：借助Parallel类提供的静态方法Invoke实现任务并行。
- 并行查询：并行实现LINQ to Objects查询，即PLINQ。

## TPL与传统多线程编程模型相比的优势

- TPL编程模型使用CLR线程池执行多个任务，并能自动处理工作分区、线程调度和取消、状态管理以及其他低级别的细节操作。
- TPL还会动态地按比例调节并发程度，从而最有效地使用所有可用的处理器。
- TPL比Thread更具智能性，当它通过试探法来预判任务集不会从并行运行中获得性能优势时，还会自动选择按顺序运行。

# Parallel类

- Parallel.For方法用于并行执行for循环。
- Parallel.Foreach方法用于并行执行foreach循环。
- Parallel.Invoke方法用于任务并行。

# Parallel帮助器类

- ParallelOptions类：为并行方法提供操作选项。
  CancellationToken：获取或设置取消标志
  TaskScheduler：默认值为null
- ParallelLoopState类：将Parallel循环的迭代与其他迭代交互。
  Break方法：告知Parallel循环尽早停止执行当前迭代之外的迭代。
  Stop方法：告知Parallel循环尽早停止执行。
- ParallelLoopResult：提供Parallel循环的完成状态。
  IsCompleted：获取该循环是否已经完成。

# 数据并行

对源集合或者数组中的元素同时执行相同操作。
实现方法：Parallel.For或Parallel.Foreach方法

## Parallel.For方法

Parallel.For方法用于并行执行for循环

简单的Parallel.For循环：Parallel.For(<开始索引>,<结束索引>,<每次迭代执行的委托>)

带并行选项的Parallel.For循环：Parallel.For(<开始索引>,<结束索引>,<并行选项>,<每次迭代执行的委托>)

带并行循环状态的Parallel.For循环：

```
public static ParallelLoopResult For(
    int  fromInclusive,                    //开始索引（包含）
    int  toExclusive,                      //结束索引（不包含）
    Action<int, ParallelLoopState> body    //每个迭代调用的委托
)
```

带线程局部变量的Parallel.For循环，线程局部变量是指某个线程内的局部变量，其他线程无法访问。线程局部变量保存的数据称为线程本地数据：

```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive,                  //开始索引（包含）
    int toExclusive,                    //结束索引（不包含）
    Func<TLocal> localInit,             //返回每个任务初始化的状态
    Func<int, ParallelLoopState, TLocal, TLocal> body, //每个迭代调用一次
    Action<TLocal> localFinally         //对每个任务执行一个最终操作
)
```

### 利用Parallel.ForEach方法实现数据并行

简单的Parallel.ForEach循环：`ForEach<TSource>(IEnumerable<TSource>, Action<TSource>)`

## 任务并行

任务并行是指同时运行一个或多个独立的任务，而且并行的任务都是异步执行的。

### Parallel.Invoke方法

Parallel.Invoke方法用于任务并行。重载形式有：

- `public static void Invoke(Action[] actions )`
- `public static void Invoke(ParallelOptions parallelOptions, Action[] actions )`

> 这两种方式都是尽可能并行执行提供的操作，采用第二种重载形式还可以取消操作。

# 第七章 WCF入门

## XML

XML（Extensible Markup Language，可扩展的标记语言）是一套用文本来定义语义标记的元标记语言，具有与平台无关、可灵活的定义数据和结构信息、便于网络传递等优势。

特点：

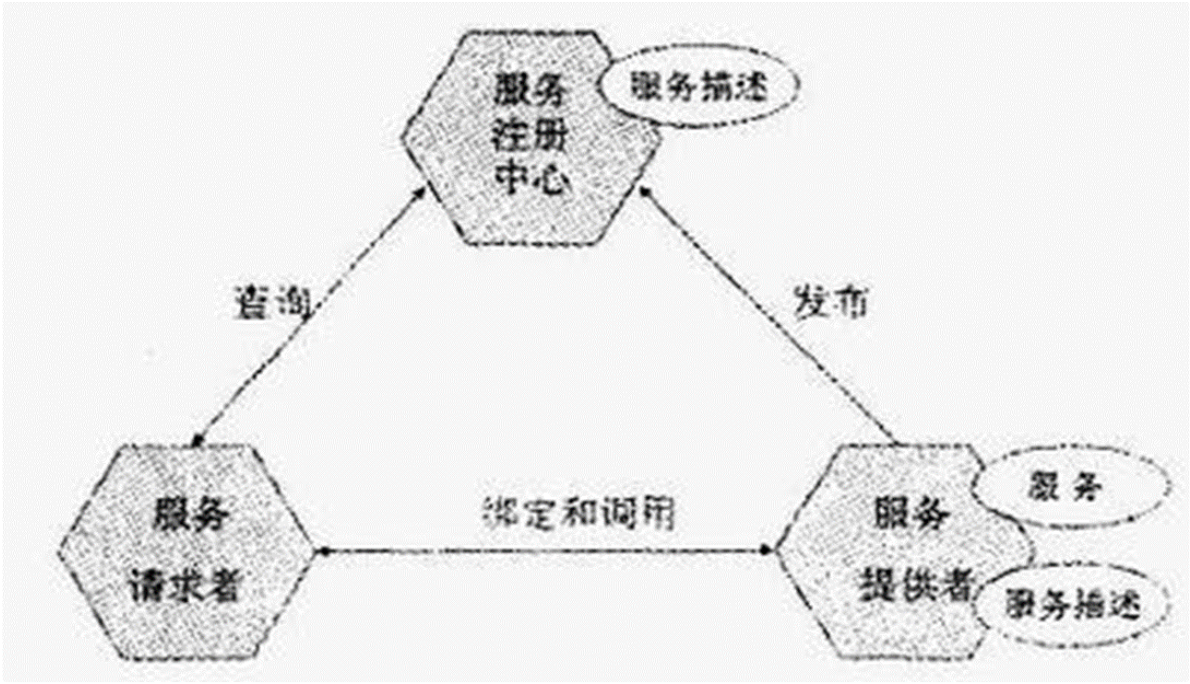
- XML是文本编码，可在任何网络中正常传输。
- 不受所选用的操作系统、对象模型和编程语言的影响
- XML中的所有标记都是自定义的，通过这些自定义的标记，可描述某种数据的不同部分及其嵌套的层次结构。
- XML规定所有标记都必须有开始和结束标志。

## Web Service

Web Service也叫Web服务

根据数据交换格式的不同，Web Service又进一步分为XML Web Service和JSON Web Service等

Web服务的体系结构基于**服务提供者、服务请求者、服务注册中心**三个角色，利用**发布、发现、绑定**三个操作来构建的




SOAP （Simple Object Access Protocol）：基于XML的，以HTTP作为基础传输协议的消息交换协议，定义了客户端与Web服务交换数据的格式。

WSDL（Web Service Description Language）：描述Web服务提供的方法以及调用这些方法的各种方式。

## 客户端与Web服务通信的过程

1. 阶段1：序列化：

   （1）客户端应用程序创建Web服务代理类的一个实例。

   （2）客户端应用程序调用代理类的方法。

   （3）客户端基础架构将Web服务所需要的参数序列化为SOAP消息，并通过网络将其发送给Web服务器。

2. 阶段2：反序列化：

   （4）Web服务器接收SOAP消息并反序列化该XML，同时创建实现Web服务的实例，再调用Web服务提供的方法，并将反序列化后的XML作为参数传递给该方法。

   （5）Web服务器执行Web服务提供的方法，得到返回值和各种输出参数

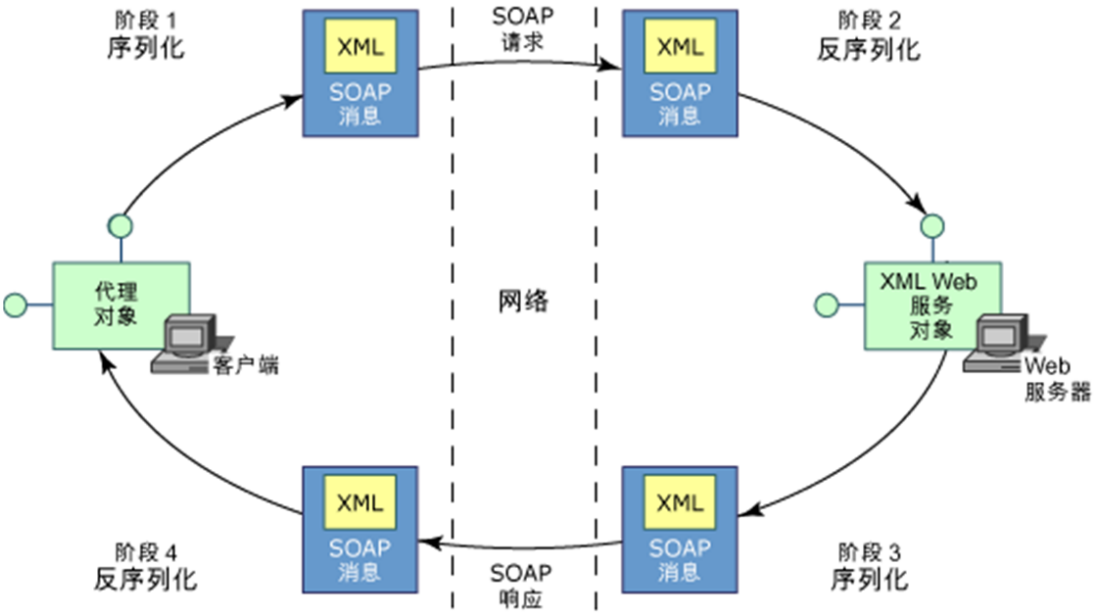
3. 阶段3：序列化：

   （6）Web服务器将返回值和输出参数序列化为SOAP消息，并通过网络将其返回给客户端基础架构。

4. 阶段4：反序列化：

   （7）客户端基础架构接收返回的SOAP消息，将XML反序列化为返回值和输出参数，并将其传递给代理类的实例。

   （8）客户端应用程序接收返回值和输出参数。




# 消息队列（MSMQ）

MQ（Message Queue）是在多个不同的应用程序之间实现相互通信的一种基于队列和事务处理的异步传输模式。
其实现原理是：消息发送者把要发送的信息放入一个容器中（称为Message），然后把它保存至一个系统公用的消息队列（Message Queue）中；本地或者是异地的消息接收程序再从该队列中取出发给它的消息进行处理。

# 面向服务的体系结构（SOA）

其基本思想就是希望用一种统一的、以“服务”为中心的模型来整合各种不同的技术，而不是仅仅限于Web服务。
对于.NET开发人员来说，这个基于SOA的具体实现就是WCF。

# WCF

特点：

- 以服务为中心
- 支持多种消息交换模式
- 支持多种传输协议和编码方式
- 支持工作流、事务以及持久性的消息处理
- 统一性、安全性和可扩展性

## 终结点（EndPoint）

终结点（EndPoint）用于确定网络通信目标，用EndPoint类来实现，在配置文件中用配置节来指定。
对于WCF来说，终结点由地址、协定和绑定组成，三者缺一不可。其中，地址（Address）用于公开服务的位置，协定（Contract）用于公开提供的是哪种具体服务。

```
<endpoint
    address="http://localhost:2338/Service1.svc"
    binding="basicHttpBinding"
    contract="WcfService.IService1" />
```

### 地址（Address）

WCF中的地址用于确定终结点的位置。地址可以是URL、FTP、网络路径或本地路径。WCF规定的地址格式为：`[传输协议]://[位置][:端口][/服务路径]`

例如：`http://www.mytest.com:50001/MyService`

WCF服务可以在各种不同的基础网络协议（例如TCP、UDP、HTTP等）之间传输。

例如：`net.tcp://localhost:50001/MyService`

### 绑定（Binding）

WCF通过绑定来定义WCF客户端与WCF服务通信的方式。
WCF提供了多种绑定方式：

`BasicHttpBinding`、`WSHttpBinding`、`NetTcpBinding`、`NetNamedPipeBinding`、`NetMsmqBinding`

### 协定（Contract）

协定表示客户端和服务端之间的信息交换规则。如果不指定协定，就无法在客户端和服务端之间进行通信。
例如：**服务协定、数据协定、消息协定**等。

服务协定：指WCF对客户端公开哪些服务。包括ServiceContract特性（定义服务协定）和OperationContract特性（定义操作协定）。

数据协定：数据协定是服务端与客户端之间交换数据的约定，即用某种抽象方式描述要交换的数据并将其传输到客户端。利用DataContract特性（定义那些类可以被序列化）和DataMember特性声明（声明类中的哪些成员可被序列化）

消息协定：在有些情况下，需要用单个类型来表示整个消息。使用消息协定可以避免在XML序列化时产生不必要的包装。MessageHeader特性和MessageBodyMember特性。在消息协定的内部，通过MessageHeader特性（MessageHeaderAttribute类）指定消息头，通过MessageBodyMember特性（MessageBodyMemberAttribute类）指定消息体。并且可以对所有字段、属性和事件应用MessageHeader特性和MessageBodyMember特性

> 如果类型中既包含消息协定又包含数据协定，则只处理消息协定

> WCF规定实现服务的接口中只能包含方法声明，不允许在接口中声明属性和字段。换言之，属性和字段是在实现接口的类中通过数据协定来公开的

## 承载WCF的方式

- 利用IIS或者WAS承载WCF服务（WCF服务应用程序采用这种模式）
- 利用Windows服务承载WCF服务（WCF服务库）
- 自承载方式（自己编写代码实现承载WCF）

> 编写WCF服务端程序有4个主要步骤：选择承载方式、设计和实现协定、配置服务、承载服务

## 终结点绑定方式

`basicHttpBinding`，`wsHttpBinding`，`wsDualHttpBinding`，`netTcpBinding`，`udpBinding`、`netNamedPipeBinding`，`netMsmqBinding`以及`netPeerTcpBinding`

# 第八章 WCF和HTTP应用编程

## HTTP简介

HTTP（HyperText Transfer Protocol，超文本传输协议）在TCP/IP体系结构中，HTTP属于应用层协议，位于TCP/IP的顶层。

HTTP以TCP方式工作：HTTP客户端首先与服务器建立TCP连接，然后客户端通过套接字发送HTTP请求，并通过套接字接收HTTP响应

HTTP是无状态的：“无状态”的含义是，客户端发送一次请求后，服务器并没有存储关于该客户端的任何状态信息。即使客户端再次请求同一个对象，服务器仍会重新发送这个对象，而不管原来是否已经向该客户端发送过这个对象。

HTTP使用元信息作为标头：HTTP通过添加标头（Header）的方式向服务器提供本次HTTP请求的相关信息，即在主要数据前添加一部分信息，称为元信息（Metainformation）。例如，传送的对象属于哪种类型，采用的是哪种编码等。

## HTTP请求

GET请求：GET请求是最为常见的一种请求，表示客户端告诉服务器获取哪些资源。GET请求后面跟随一个网页的位置。除了页面位置作参数之外，这种请求还可以跟随协议的版本如HTTP/1.0等作为参数，以发送给服务器更多的信息。

POST请求：POST请求要求服务器接收大量的信息。与GET请求相比，POST请求不是将请求参数附加在URL后面，而是在请求主体中为服务器提供附加信息。一般用于客户端填写包含在Web表单（Form）中的内容后，将这些填入的数据以POST请求的方式发送给服务器。

HEAD请求：HEAD请求在客户端程序和服务器端之间进行交流，而不会返回具体的文档。因此HEAD方法通常不单独使用，而是和其他的请求方法一起起到辅助作用。

## HTTP编程相关类（了解）

### WebRequest与HttpWebRequest

- WebRequest：请求/响应模型的抽象（abstract）基类。用于访问Internet数据。它允许使用该请求/响应模型的应用程序可以用协议不可知的方式从Internet请求数据。
- HttpWebRequest：是针对于HTTP的特定实现。该类通过HTTP协议和服务器交互。 使用Create方法初始化新的WebRequest实例 `HttpWebRequest request = (HttpWebRequest)WebRequest.Create(uriString);`

### HttpWebResponse类

通过调用WebRequest实例的GetReponse方法来创建，并不通过构造函数创建HttpWebResponse实例

```
HttpWebResponse response=(HttpWebResponse)request.GetResponse();
```

### 基本HTTP绑定（BasicHttpBinding类）

基本HTTP绑定用BasicHttpBinding类来实现，在配置文件中用basicHttpBinding元素来配置

### 安全HTTP绑定（WSHttpBinding类）

WSHttpBinding定义一个适合于非双工服务的安全、可靠且可互操作的绑定。该绑定实现了WS-ReliableMessaging规范（保证了可靠性）和WS-Security规范（保证了消息安全性和身份验证）。

### 双工安全HTTP绑定（WSDualHttpBinding类）

WSDualHttpBinding类也是使用HTTP作为基础传输协议，在服务端和客户端配置文件中用wsDualHttpBinding元素来配置。该绑定也是使用“文本/XML”作为默认的消息编码。但是，它仅支持SOAP安全模式，且需要可靠的消息传递。

## WCF客户端和服务端的消息交换模式

### 请求应答模式（Action/Reply）

客户端向WCF服务端发送请求后，服务端执行服务操作，并将操作结果返回到客户端。客户端如果不是通过异步操作来调用的，在服务端返回服务操作结果之前，客户端代码将处于阻塞状态。

### 单向模式（IsOneWay）

客户端调用WCF服务操作时，服务端不向客户端返回操作结果。即使服务端出现执行错误，它也不会向客户端返回结果。

### 双工通信

双工（duplex）是指客户端和服务端都可以主动呼叫对方。在这种通信模式中，WCF利用双向绑定实现服务端和客户端相互公开终结点的信息

# 第9章 WCF和TCP应用编程

## TCP的特点

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，在网络中提供双工和可靠的服务。

特点：

- 一对一通信。
- 安全顺序传输。
- 通过字节流收发数据。
- 传输的数据无消息边界

### 解决TCP无消息边界问题的办法

- 发送固定长度的消息，这种办法适用于消息长度固定的场合。
- 将消息长度与消息一起发送，一般在每次发送的消息前面用4个字节表明本次消息的长度，然后将其和消息一起发送到对方；对方接收到消息后，首先从前4个字节获取实际的消息长度，再根据消息长度值依次接收发送方发送的数据。这种办法适用于任何场合。
- 使用特殊标记分隔消息，使用特殊分隔符对消息进行分隔。这种办法主要用于消息本身不包含特殊标记的场合。

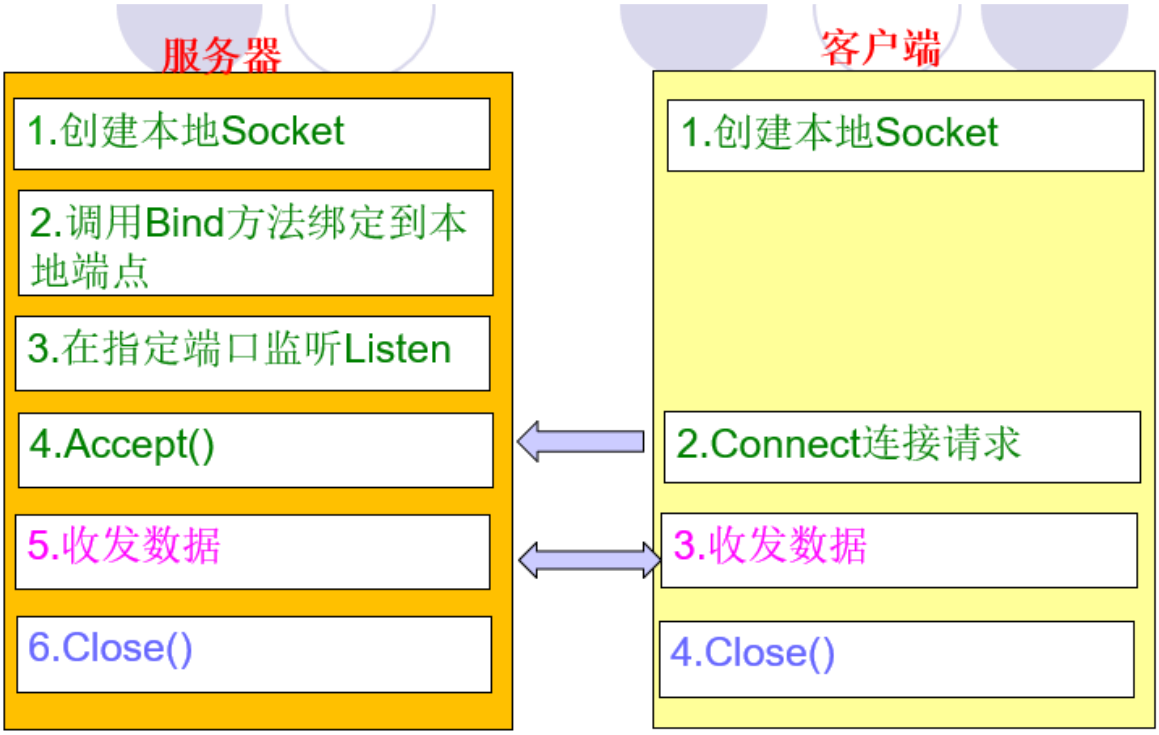
# TCP应用编程的技术选择

1. 用Socket类实现
   TCP通信过程中的所有细节全部通过自己编写的程序来控制。
   特点：方式灵活，但需要程序员编写的代码多。
   建议：定义一些新的协议或者对底层的细节进行更灵活的控制时使用此技术。
2. 用TcpClient和TcpListener以及多线程实现
   TcpClient和TcpListener类对Socket进一步封装后的类，在一定程度上简化了用Socket编写TCP程序的难度，但灵活性也受到一定的限制。
   特点：TCP数据传输过程中的监听和通信细节（比如消息边界问题）仍然需要程序员自己通过代码去解决。
3. 用TcpClient和TcpListener以及多任务实现
   编写TCP应用程序时，不需要开发人员考虑多线程创建、管理以及负载平衡等实现细节，只需要将多线程看作是多个任务来实现即可。
4. 用WCF实现
   监听和无消息边界等问题均有WCF内部自动完成。
   程序员只需要考虑传输过程中的业务逻辑即可。另外，利用WCF还可以实现自定义的协议。

## 套接字

面向连接套接字编程有三个步骤：

1. 服务器与客户端建立连接（建立连接阶段）
2. 服务器与客户端之间收发消息（收发消息阶段）
3. 断开连接，关闭套接字（断开连接阶段）




1.建立连接（服务器）

```
IPHostEntry local=Dns.GetHostByName(Dns.GetHostName());
IPEndPoint iep = new IPEndPoint(local.AddressList[0], 1180);
          //创建套接字
Socket  serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
             //绑定
            serverSocket.Bind(iep);
```

```
            serverSocket.Listen(10); //监听
Socket clientSocket = serverSocket .Accept();
```

1.建立连接（客户端）

```
IPAddress remoteHost = IPAddress.Parse("192.168.0.1");
IPEndPoint iep = new IPEndPoint(remoteHost, 1180);
Socket localSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
 localSocket.Connect(iep); //连接
```

2.发送、接收信息（服务器）

```
Socket clientSocket = serverSocket .Accept();
    //建立连接后，利用Send方法向客户端发送信息
    clientSocket.Send(Encoding.ASCII.GetBytes("server send Hello"));
    //接收客户端信息
    byte[] myresult = new Byte[1024];
    int receiveNum = clientSocket.Receive(myresult);
    Console.WriteLine("接收客户端消息：{0}", Encoding.ASCII.GetString(myresult));
```

2.发送、接收信息（客户端）

```
//建立连接成功后，向服务器发送信息
string sendMessage = "client send Message Hello"+DateTime.Now;
localSocket.Send(Encoding.ASCII.GetBytes(sendMessage));
Console.WriteLine("向服务器发送消息:{0}", sendMessage);
//接收服务器信息
byte[] result = new Byte[1024];
localSocket.Receive(result);
Console.WriteLine("接收服务器消息：{0}", Encoding.ASCII.GetString(result));
```

3.断开连接

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

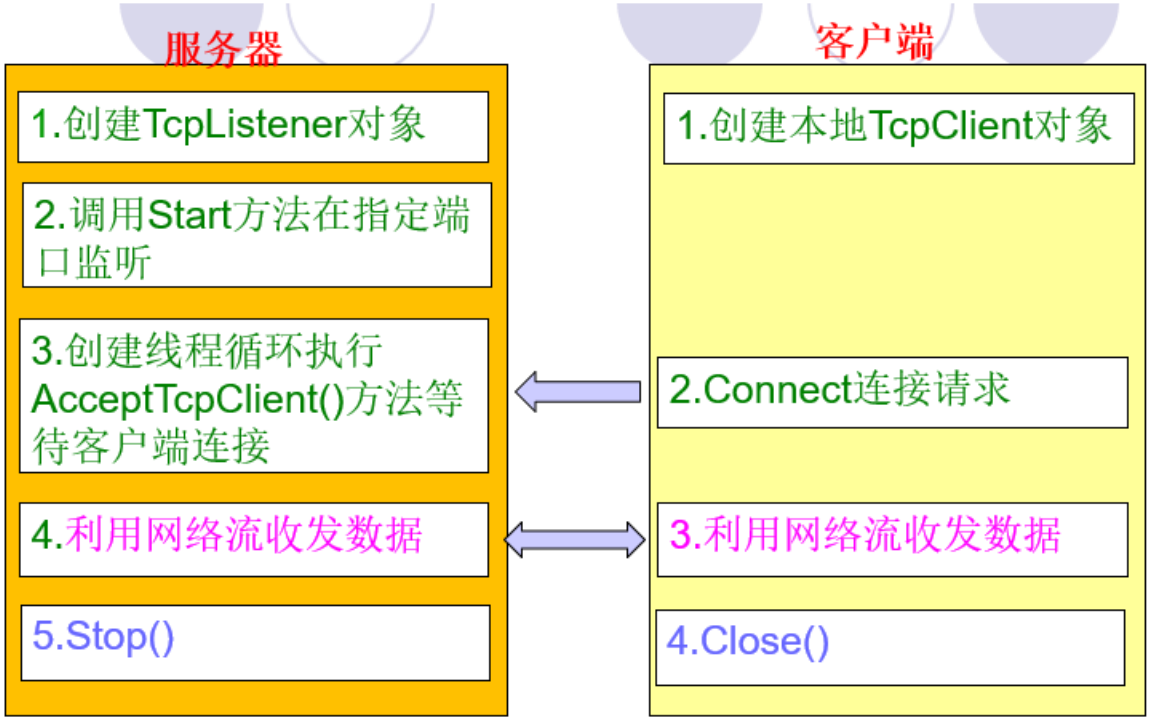
# 利用传统技术实现TCP应用编程

## TcpClient类和TcpListener类

为了简化网络编程的复杂度，.NET对套接字进行了封装，封装后的类就是System.Net.Sockets命名空间下的TcpListener类和TcpClient类

- TcpListener类用于监听客户端连接请求。
- TcpClient类用于提供本地主机和远程主机的连接信息 `TcpClient tcpClient = new TcpClient("www.abcd.com", 51888);`

## TCPListener和TCPClient的编程流程




## WCF与TCP相关的绑定

利用WCF编写TCP应用程序时，只需要在服务端配置文件中设置相关的绑定，就可以轻松实现相应的功能，而且不容易出错。

### NetTcpBinding

NetTcpBinding类用于将WCF和TCP绑定在一起，并以服务的形式提供TCP服务端和客户端之间的通信。
在服务端配置文件中，用元素来配置

元素默认配置如下：

- 安全模式：Transport（保证传输安全）。
- 消息编码方式：Binary（采用二进制消息编码器）。
- 传输协议：TCP。
- 会话：提供传输会话（也可以配置为可靠对话）。
- 事务：无事务处理功能（也可以配置为支持事务处理）。
- 双工：支持。

# 第10章 WCF和UDP应用编程

## UDP基本知识

UDP（User Datagram Protocol，用户数据报协议）是简单的、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。
UDP的主要作用是将网络数据流量压缩成数据报的形式，每一个数据报用8个字节描述报头信息，剩余字节包含具体的传输数据。
UDP特点：

- UDP可以一对多传输，UDP不但支持一对一通信，而且支持一对多通信。或者说，利用UDP可以使用多播技术同时向多个接收方发送信息。

- UDP传输速度比TCP快，由于UDP不需要先与对方建立连接，也不需要传输确认，因此其数据传输速度比TCP快得多。
- UDP有消息边界，使用UDP不需要考虑消息边界问题。
- UDP不保证有序传输，UDP不确保数据的发送顺序和接收顺序一致。对于突发性的数据报，有可能会乱序。但是，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有可能发生。
- UDP可靠性不如TCP，UDP不提供数据传送的保证机制。

# 单播、广播和多播

单播是指只向某个指定的远程主机发送信息，这种方式本质上属于一对一的通信。

广播是指同时向子网中的多台计算机发送消息，分为本地广播和全球广播。本地广播是指向子网中的所有计算机发送广播消息，其他网络不会受到本地广播的影响。全球广播是指使用所有位全为1的IP地址（对于Ipv4来说指255.255.255.255），但是，由于路由器默认会自动过滤掉全球广播，所以使用这个地址没有实际意义。

多播也叫多路广播，由于多播是分组的，所以也叫组播。对于IPv4来说，多播是指在224.0.0.0到239.255.255.255的D类IP地址范围内进行广播（第1个字节在224～239之间）。或者说，发送方程序通过这些范围内的某个地址发送数据，接收方程序也监听并接收来自这些地址范围的数据。

# UDP应用编程的技术选择

## 用Socket类实现

直接用System.Net.Sockets命名空间下的Socket类来实现。采用这种方式时，需要程序员编写的代码最多，所有底层处理的细节都需要程序员自己去考虑

## 用UdpClient和多线程实现

用System.Net.Sockets命名空间下的UdpClient类和Thread类来实现。UdpClient类对基础Socket进行了封装，发送和接收数据时不必考虑套接字收发时必须处理的细节问题，在一定程度上降低了用Socket编写UDP应用程序的难度，提高了编程效率

## 用UdpClient和多任务实现

用UdpClient类以及基于任务的编程模型（Task类）来实现。用多任务实现比直接用多线程实现更有优势

## 用WCF实现

用WCF来实现。即将WCF和UDP通过配置绑定在一起，这是对Socket进行的另一种形式的封装

# 利用UdpClient类发送和接收数据

TCP有TcpListener类和TcpClient类，而UDP只有UdpClient类，这是因为UDP是无连接的协议，所以只需要一种Socket。

由于UDP不需要发送方和接收方先建立连接，因此发送方可以在任何时候直接向指定的远程主机发送UDP数据报。在这种模式中，发送方是客户端，具有监听功能的接收方是服务端

UdpClient类提供了多种重载的构造函数，分别用于IPv4和IPv6的数据收发。在这些构造函数中，最常用的重载形式就是带本地终结点参数的构造函数，语法如下。

```
public UdpClient(IPEndPoint localEp)
```

用这种构造函数创建的UdpClient对象会自动与参数中指定的本地终结点绑定在一起。绑定的目的是为了监听来自其他远程主机的数据报。例如：

```
IPEndPoint localEndPoint = new IPEndPoint(localAddress, 51666);
UdpClient client =new UdpClient(localEndPoint);
```

发送数据：

- 用UdpClient对象的Send方法同步发送数据时，该方法返回已发送的字节数。
- Send方法有多种重载形式，这里只介绍最常用的重载形式，语法如下：

```
public int Send(byte[] data, int length, IPEndPoint remoteEndPoint)
```

接收数据：

- UdpClient对象的Receive方法用于获取来自远程主机的UDP数据报，语法如下。

```
public byte[] Receive(ref IPEndPoint remoteEndPoint)
```

### 异步发送和接收数据

对于执行时间可能较长的任务，或者无法预测用时到底有多长的任务，最好用基于任务的异步编程来实现（调用UdpClient对象的SendAsync方法和ReceiveAsync方法）。使用这种办法的好处是收发数据时，用户界面不会出现停顿现象。
异步发送数据，例如：

```
await client.SendAsync(bytes, bytes.Length, remoteEndPoint);
```

异步接收数据，例如：

```
public async void ReceiveDataAsync()
{
    while (true)
    {
        var result = await client.ReceiveAsync();
        string s = Encoding.Unicode.GetString(result.Buffer);
        textBlock1.Dispatcher.Invoke(() =>
        {
            textBlock1.Text += string.Format(
                "来自{0}：{1}\n", result.RemoteEndPoint, s);
        });
    }
    }
```

## 利用UdpClient实现群发功能

### 加入和退出多播组

多播组可以是永久的，也可以是临时的。在实际应用中，大多数多播组都是临时的，即仅在多播组中有成员的时候才存在。
凡是加入到多播组的接收端都可以接收来自多播发送端发送的数据。但是，如果不加入多播组，则无法接收多播数据。

向多播组发送数据时，需要先创建一个UdpClient对象：`UdpClient udpclient = new UdpClient("224.0.0.1", 8001);`

使用多播时，应注意的是该对象TTL值的设置。利用UdpClient对象的Ttl属性可修改TTL的默认值，例如

```
udpClient.Ttl = 50;
```

利用UdpClient对象的JoinMulticastGroup方法可加入到指定的多播组中：

```
udpClient.JoinMulticastGroup(IPAddress.Parse("224.0.0.1"));
```

利用UdpClient的DropMulticastGroup方法可以退出多播组。参数中指出要退出多播组的IPAddress对象：

```
udpClient.DropMulticastGroup(IPAddress.Parse("224.100.0.1"));
```

### 是否允许接收多播

利用UdpClient对象的MulticastLoopback属性可控制是否允许接收多播信息。该属性默认为true，即允许接收多播

## 利用WCF实现UDP应用编程

> WCF中与UDP相关的绑定只有，对应的类为UdpBinding类

> 用WCF实现多播时，只需要一个项目。该项目既是服务端，也是客户端，并且只能用自承载方式来实现