# 一、C#基础知识(1)

## 1、[C#中 foreach 用法](http://blog.csdn.net/eckelwei/article/details/8264949)

foreach循环用于列举出集合中所有的元素，foreach语句中的表达式由关键字in隔开的两个项组成。in右边的项是集合名，in左边的项是变量名，用来存放该集合中的每个元素。

该循环的运行过程如下：每一次循环时，从集合中取出一个新的元素值。放到只读变量中去，如果括号中的整个表达式返回值为true，foreach块中的语句就能够执行。一旦集合中的元素都已经被访问到，整个表达式的值为false，控制流程就转入到foreach块后面  
的执行语句。

foreach语句经常与数组一起使用，下面实例将通过foreach语句读取数组的值并进行显示。

**数组的属性：Array.Length数组的容量**

利用这个属性，我们可以取得数组对象允许存储的容量值，也就是数组的长度、元素个数,这个比较好理解，数组还有其他的属性，比如数组的维数等，属性的用法比较简单，学会一种，其他的格式基本一致，这里我们就不举例了。

当数组的维数、容量较多时，C#提供了foreach语句，专门用来读取集合/数组中的所有元素，我们把这种功能叫做遍历。语法书写如下：

**遍历数组：foreach（type objName in collection/Array）**

这段语句会逐一检查数组中的所存储的变量值，并且一一将其取出，其中的type是你所要读取的数组对象将要存储在objName变量的数据类型，而objName是定义了一个type类型的变量名，代表每一次从集合和数组（collection/Array）中取得的元素，collection/Array则是所要存取的数组对象。用这种方法只需写一个foreach就可以遍历出除交错数组以外的所有维数的数组。

例子：用foreach循环一次性遍历a数组

          int[,,] a = new int[2, 2, 2] { {{ 1, 2 }, { 3,4}},{{ 5, 6 }, { 7,8}} };//定义一个2行2列2纵深的3维数组a  
                      foreach(int i in a)  
                      {  
                          Console .WriteLine (i);  
                      }

这两种代码执行的结果是一样的都是 每行一个元素，共8行，元素分别是1 2 3 4 5 6 7 8

string[]数组里面是存放string型的值，List<string>是存放string类型的对象

## 2、[C#中的InitializeComponent()方法](http://blog.csdn.net/ycigwy/article/details/51711575)

 InitializeComponent()用于初始化窗体上控件。

在每一个窗体生成的时候，都会针对于当前的窗体定义InitializeComponent()方法，该方法实际上是由系统生成的对于窗体界面的定义方法。  
//位于.cs文件之中的InitializeComponent()方法  
public Form011()          
{              
InitializeComponent();  
}   
在每一个Form文件建立后，都会同时产生程序代码文件.CS文件，以及与之相匹配的.Designer.CS文件，业务逻辑以及事件方法等被编写在.CS文件之中，而界面设计规则被封装在.Designer.CS文件里，下面代码为.Designer.CS文件的系统自动生成的脚本代码。  
namespace Example\_FormTest  
{  
    partial class frmStudent  
    {  
        /// <summary>  
        /// 必需的设计器变量。  
        /// </summary>  
        private System.ComponentModel.IContainer components = null;

        /// <summary>  
        /// 清理所有正在使用的资源。  
        /// </summary>  
        /// <param name="disposing">如果应释放托管资源，为 true；否则为 false。</param>  
        protected override void Dispose(bool disposing)  
        {  
            if (disposing && (components != null))  
            {  
                components.Dispose();  
            }  
            base.Dispose(disposing);  
        }

        #region Windows 窗体设计器生成的代码

        /// <summary>  
        /// 设计器支持所需的方法 - 不要  
        /// 使用代码编辑器修改此方法的内容。  
        /// </summary>  
        private void InitializeComponent()  
        {  
            this.SuspendLayout();  
            //   
            // frmStudent  
            //   
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);  
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;  
            this.ClientSize = new System.Drawing.Size(292, 273);  
            this.Name = "frmStudent";  
            this.Text = "学生信息";  
            this.MouseClick += new System.Windows.Forms.MouseEventHandler(this.frmStudent\_MouseClick);  
            this.KeyPress += new System.Windows.Forms.KeyPressEventHandler(this.frmStudent\_KeyPress);  
            this.ResumeLayout(false);

        }

        #endregion  
    }  
}

在代码之中，可以很容易发现InitializeComponent()方法和Dispose()方法，前者为界面设计的变现内容，后者为表单释放系统资源时候执行编码。  
InitializeComponent()方法反映了窗体设计器中窗体和控件的属性。通常不会在世界上个文件中修改代码。如果更改InitializeComponent()方法中的相关属性参数，在窗体设计器界面上也会显示出来。  
我们发现建一个c#窗体项目，在Form1.cs和Form1.Designer.cs里分别有一个InitializeComponent()函数  
Form1.Designer.cs 里的是定义 （函数名后面有大括号包含定义内容）  
Form1.cs 里的是调用 （函数名后面分号结尾）  
partial 表示以下创建的是分布类代码  
也就是说 一个类的定义代码 可以写在两个不同的页面  
Form1.cs和Form1.Designer.cs   
注意：InitializeComponent()方法写在前面和后面是有区别的。InitializeComponent()方法是用来初始化窗体组件的，在InitializeComponent之前，窗体的各个控件还没有进行初始化，比如说窗体上有一个TextBoxUserName，在InitializeComponent之前它还只是一个TextBox类型的变量而已，如果在InitializeComponent之前调用这个TextBox的属性或方法，就会出错。再比如说你拖上取一个TextBox,它放在Form的位置，TextBox的一些属性。包括new 这个TextBox都放在那个函数里面处理的。  
我们可以看一下Form.Designer.cs，在那里找到InitializeComponent方法，您会发现窗体对象的初始化都是在那里做的。

## 3、[C# 里面的 #region 是什么意思?](http://blog.csdn.net/wjm371002515/article/details/6895816)

#region是c#预处理器指令。

    #region是一个分块预处理命令，它主要用于编辑代码的分段，在编译时会被自动删除。

    #region使你可以在使用VS代码编辑器的大纲显示功能时指定可展开或折叠的代码块。

    例如：

http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif#region  MyClass definition

http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gifpublic class MyClass   
        http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif{  
        http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    static void Main()   
        http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
        http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
        http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif}  
        http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif#endregion

    备注：#region块必须以#endregion指令终止。

              #region块不能与#if块重叠。但是，可以将#region块嵌套在#if块内，或将#if块嵌套在#region块内。

* 使用#region就是将一些有关联的代码组织在一起，然后折叠起来。这样你就在一个函数中看不到很长的令人头痛的代码段。
* 当然这段代码很简单。在实际项目中，你可能看到上百行甚至更多的代码在一个#region中。如果把它折叠起来。看起来就会很整洁。
* region 不适合在大方法中使用，当你在一个方法中使用#region 的时候，停下来想想你刚刚写了什么代码？大多数时候，你可以将这些代码段独立成一个函数。

## 4、C#中this的作用

在C#中,this关键字代表当前实例,我们可以用this.来调用当前实例的成员方法,变量,属性,字段等; 也可以用this来做为参数状当前实例做为参数传入方法. 还可以通过this[]来声明索引器。下面介绍C#中this的作用

**一、C# this指针的几种用法**

**1、限定被相似的名称隐藏的成员**

**C# 代码   复制**

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gifpublic class ThisName

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public string name = "张三";

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public int num = 55;

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif public ThisName() { }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public void GetThisName(string name, int num)

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif name = this.name;//调取全局变量name

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif num = this.num;//调取全局变量num

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif HttpContext.Current.Response.Write("参数name的值为:"+name+";参数num的值为:"+num);//输出结果为"参数name的值为张三;参数num值为:55"

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockEnd.gif }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedBlockEnd.gif }

**2、将对象作为参数传递到别的方法中**

**C# 代码   复制**

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gifpublic class ThisFF

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public ThisFF()

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockEnd.gif }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public string shuju()

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif return "This指针当做方法传递";

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockEnd.gif }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public void f1(ThisFF ff)

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif HttpContext.Current.Response.Write(ff.shuju());

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockEnd.gif }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif public void f()

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif f1(this);//this在这里可以理解为 ThisFF ff=news ThisFF();当前类的实例

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockEnd.gif }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedBlockEnd.gif }http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif ThisFF ff=news ThisFF();//方法实例

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif ff.f();//调用方法 输出结果为： This指针当做方法传递

**3、声明索引器**

**C# 代码   复制**

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gifprivate int \_Sy;

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif public int this[int sy]

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedBlockStart.gif {

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif get { return \_Sy;}

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedSubBlockStart.gif set { sy=\_Sy;}

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/InBlock.gif

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/ExpandedBlockEnd.gif }

http://www.studyofnet.com/Codefan-Controls/OutliningIndicators/None.gif

**二、C#中this的总结**

1、this关键字引用被访问成员所在的当前实例。静态成员函数没有this指针。this关键字可以用来从构造函数，实例方法和实例化访问器中访问成员，不能在静态方法。静态属性访问器或者域声明的变量初始化程序中使用this关键字，这将会产生错误。

2、在类的构造函数中出现的this作为一个值类型表示对正在构造的对象本身的引用。

3、在类的方法中出现this作为一个值类型表示对调用该方法的对象的引用。

4、在结构的构造函数中出现的this作为一个变量类型表示对正在构造的结构的引用。

5、在结构的方法中出现的this作为一个变量类型表示对调用该方法的结构。

## 5、[c#.net]SuspendLayout 和 ResumeLayout

在Designer中会看到这两个方法，究竟是做什么用的呢？

一般在加载所有控件之前SuspendLayout方法，挂起所有控件在界面的逻辑布局事件，等又有布局代码完后再调用 ResumeLayout事件吧刚才挂起的逻辑布局一并执行。

例如我们在设置初始化控件时候，都会用到Size、Location、Anchor 或 Dock属性，而每一个属性都会去改变控件在界面上的布局，使得控件，每执行一段代码就就要界面重绘一次，那么我们先挂起所有事件，之后一起执行，这样就可以提高带有许多控件的应用程序的性能。

控件的布局逻辑被挂起，直到调用 ResumeLayout 方法为止。

当调整控件的多个属性时，将先后使用 SuspendLayout 和 ResumeLayout 方法取消多个 Layout 事件。例如，通常先调用 SuspendLayout 方法，然后设置控件的 Size、Location、Anchor或 Dock 属性，最后调用 ResumeLayout 方法以使更改生效。

**注意**：为了防止layout事件的发生。这个事件将引发重新绘图的事件，如果介面上控件较多的话，改变控件某些属性时，由于不断地重画，性能变低，所以用suspendlayout来阻止。  
resumelayou可以结束上面的限制。

## 6、[C#中的set和get方法](http://blog.csdn.net/u013095889/article/details/52804360)

在面向对象编程（OOP）中，是不允许外界直接对类的成员变量直接访问的，既然不能访问，那定义这些成员变量还有什么意义呢？所以C#中就要用set和get方法来访问私有成员变量，它们相当于外界访问对象的一个通道，一个“接口”。先来看一段代码：

class Employee  
  {  
        private string name;  
        private byte age;  
        public string Name  
        {  
            get { return name; }  
            set { name = value; }  
        }  
        public byte Age  
        {  
            get { return age; }  
            set { age = value; }  
        }  
  }

　　代码中定义了两个私有变量name和age，当我们不想让外界随意访问该私有变量时，可以使用属性来访问，语法为：

public <返回类型（要与被访问变量的类型相同）> <属性名（不能与被访问变量同名）>  
  {  
        get{ return <被访问变量>;}  
        set{ <被访问变量> = value;}  
  }

　　当我们使用属性来访问私有成员变量时就会调用里面的get方法，当我们要修改该变量时就会调用set方法，当然在定义的时候可以只定义一个get方法或只定义一个set方法。如果只定义get方法，那么这个相应变量就是“只读”的；如果只定义set方法，那么相应变量就是“只写”的。

class Employee  
    {  
        private string name;  
        private byte age;  
        public string Name  
        {  
            get { return name; }  
            set { name = value; }  
        }  
        //\*\*\*\*修改后\*\*\*\*↓↓↓↓↓↓↓↓  
        public byte Age  
        {  
            get { return age; }    
            set  
            {                  
                if (value > 10 && value<=100)   //一般在公司雇员的年龄都在10到100岁之间  
                age = value;     
            }  
        }  
        //\*\*\*\*修改后\*\*\*\*↑↑↑↑↑↑↑↑  
    }

上例中，set就像一位门卫大叔一样，只有好人才能进来。可以通过属性来控制对员变量的读写，防止对成员变量的非法赋值等。

再举个小例子，以下是个温度计的类：

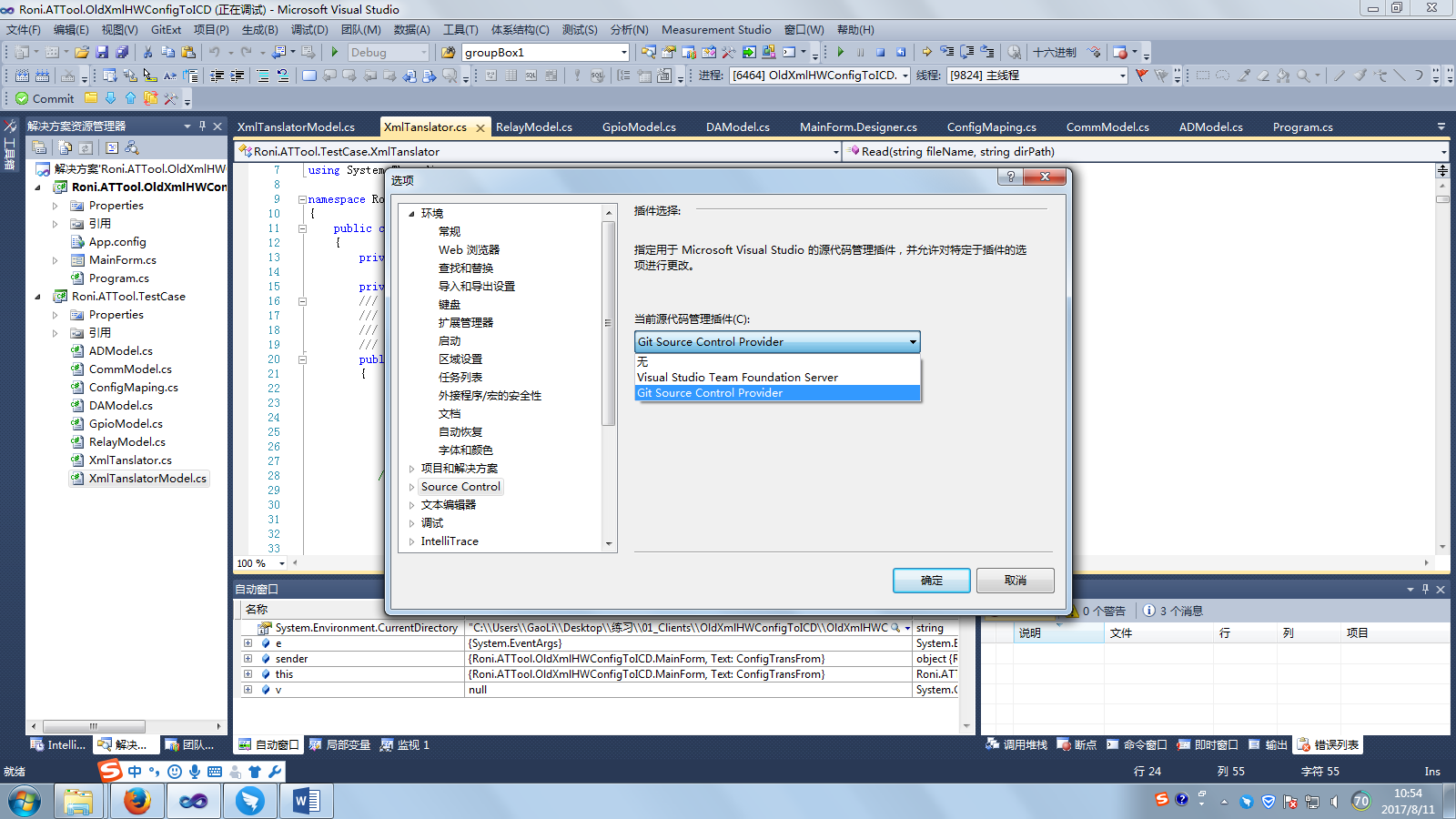
class Thermometer  
    {  
        private double temperature;  
        public Thermometer(double temperature)  //构造函数  
        {  
            this.temperature = temperature;  
        }  
        public double Temperature  
        {  
            get { return temperature; }  
            set { temperature = value; }  
        }  
    }

假设这里的温度单位表示的是摄氏度（℃），如果在需求分析阶段误解了客户的原始意愿或者客户在日后变更了需求，需要把系统中所有表示温度的变量temperature都用来表示开尔文温度（K）。那么不必动系统中成百上千个Temperature属性，只需在get和set函数中稍微修改一下代码即可：

class Thermometer  
    {  
        private double temperature;  
        public Thermometer(double temperature)  //构造函数  
        {  
            this.temperature = temperature;  
        }  
        public double Temperature  
        {  
            //\*\*\*\*修改后\*\*\*\*↓↓↓↓↓↓↓↓  
            get { return temperature-273.15; }  
            set { temperature = value+273.15; }  
            //\*\*\*\*修改后\*\*\*\*↑↑↑↑↑↑↑↑  
        }  
    }

建一个控制台应用程序来[**测试**](http://lib.csdn.net/base/softwaretest)一下，主函数为：

class Test  
    {  
        static void Main(string[] args)  
        {  
            Thermometer a = new Thermometer(40);  
            Console.WriteLine(a.Temperature);  
        }  
    }



## 7、[override和new的区别](http://www.cnblogs.com/xiaoyusmile/archive/2012/06/18/2553851.html)

**override**

1. override是派生类用来重写基类中方法的；

2. override不能重写非虚方法和静态方法；

3. override只能重写用virtual、abstract、override修饰的方法；

4. 不能使用修饰符 new、static、virtual 或 abstract 来修改 override 方法。

**new**

1. new是派生类用来隐藏基类中的方法的；也就是说在派生类中“看不到”基类中的方法；

2. 如果要在派生类中隐藏（不是重写）基类中的方法，而没有使用new关键字，编译时会出现一个警告，提示如果是要隐藏基类中的方法，请使用new关键字；

3. 派生类可以隐藏基类中的虚方法，也可以隐藏基类中的普通方法。

4. 如果在派生类中用private来修饰new 方法，那么只在该派生类中隐藏了基类中的方法，在该派生类之外，相当于没有隐藏基类中的方法；

5. 如果在派生类中隐藏了基类中的方法，在该派生类的派生类中，将延续对该派生类对基类方法的隐藏。

代码：

[复制代码](javascript:void(0);)

class A

{

public virtual void Func() // 注意virtual,表明这是一个虚拟函数

{

Console.WriteLine("Func In A");

}

}

class B : A // 注意B是从A类继承,所以A是父类,B是子类

{

public override void Func() // 注意override ,表明重新实现了虚函数

{

Console.WriteLine("Func In B");

}

}

class C : B // 注意C是从B类继承,所以B是父类,C是子类

{

}

class D : A // 注意D是从A类继承,所以A是父类,D是子类

{

public new void Func() // 注意new，表明覆盖父类里的同名类，而不是重新实现

{

Console.WriteLine("Func In D");

}

}

class E : D // 注意E是从D类继承,所以D是父类,E是子类

{

}

class F : A

{

private new void Func() //注意new关键字前有private修饰符，故该隐藏只在F类内有效

{

Console.WriteLine("Func In F");

}

public void Func2()

{

Func(); //在F类内隐藏了基类的Func方法，故此处调用的private new void Func()

}

}

static void Main(string[] args)

{

A a; // 定义一个a这个A类的对象.这个A就是a的申明类

A b; // 定义一个b这个A类的对象.这个A就是b的申明类

A c; // 定义一个c这个A类的对象.这个A就是c的申明类

A d; // 定义一个d这个A类的对象.这个A就是d的申明类

A e; // 定义一个e这个A类的对象.这个A就是e的申明类

A f; // 定义一个f这个A类的对象.这个A就是f的申明类

a = new A(); // 实例化a对象,A是a的实例类

b = new B(); // 实例化b对象,B是b的实例类

c = new C(); // 实例化c对象,C是c的实例类

d = new D(); // 实例化d对象,D是d的实例类

e = new E(); // 实例化e对象,E是e的实例类

f = new F(); // 实例化f对象,F是f的实例类

Console.WriteLine("a.Func();");

a.Func(); // 执行a.Func：1.先检查申明类A 2.检查到是虚拟方法 3.转去检查实例类A，就为本身 4.执行实例类A中的方法 5.输出结果 Func In A

Console.WriteLine("b.Func();");

b.Func(); // 执行b.Func：1.先检查申明类A 2.检查到是虚拟方法 3.转去检查实例类B，有重载的 4.执行实例类B中的方法 5.输出结果 Func In B

Console.WriteLine("c.Func();");

c.Func(); // 执行c.Func：1.先检查申明类A 2.检查到是虚拟方法 3.转去检查实例类C，无重载的 4.转去检查类C的父类B，有重载的 5.执行父类B中的Func方法 5.输出结果 Func In B

Console.WriteLine("d.Func();");

d.Func(); // 执行d.Func：1.先检查申明类A 2.检查到是虚拟方法 3.转去检查实例类D，无重载的（这个地方要注意了，虽然D里有实现Func()，但没有使用override关键字，所以不会被认为是重载） 4.转去检查类D的父类A，就为本身 5.执行父类A中的Func方法 5.输出结果 Func In A

Console.WriteLine("e.Func();");

e.Func(); // 执行e.Func：E继承D，E.Func没有重写父类中的方法，相当于执行父类D中的Func方法，输出结果 Func In A

Console.WriteLine("f.Func();");

f.Func(); // 执行f.Func：F类中虽然隐藏了基类中的Func方法，但是有private修饰符，该隐藏只在F类范围内有效。执行f.Func相当于执行其基类中的Func方法，输出结果 Func In A

D d1 = new D();

Console.WriteLine("d1.Func();");

d1.Func(); // 执行D类里的Func()，输出结果 Func In D

E e1 = new E();

Console.WriteLine("e1.Func();");

e1.Func(); // 执行E类里的Func()，输出结果 Func In D

F f1 = new F();

Console.WriteLine("f1.Func();");

f1.Func(); // 执行F类里的Func()，输出结果 Func In A

Console.WriteLine("f1.Func2();");

f1.Func2(); // 执行F类里的Func2()，输出结果 Func In F

Console.ReadLine();

}

## 8、[MarshalAs的使用](http://blog.csdn.net/xiaobai1593/article/details/7025775)

**作用：**

MarshalAs属性指示如何在托管代码和非托管代码之间封送数据。

**使用方法：**

[MarshalAs(UnmanagedType unmanagedType, 命名参数)]

实际上相当于构造一个MarshalAsAttribute类的对象

**常用的UnmanagedType枚举值：（详细内容查MSDN）**

BStr   长度前缀为双字节的 Unicode 字符串；

LPStr  单字节、空终止的 ANSI 字符串。；

LPWStr  一个 2 字节、空终止的 Unicode 字符串；

ByValArray 用于在结构中出现的内联定长字符数组，应始终使用MarshalAsAttribute的SizeConst字段来指示数组的大小。

注意：

在用Marshal.SizeOf()，即获取对象的非托管大小时，获得的是自己定义的大小；

但在实际处理的时候，是按照实际的大小来获取的

**示例：**

定义一个固定大小的结构体，代码如下：

结构的声明：

**[csharp]** [view plain](http://blog.csdn.net/xiaobai1593/article/details/7025775) [copy](http://blog.csdn.net/xiaobai1593/article/details/7025775)

1. struct Info
2. {
3. [MarshalAs(UnmanagedType.ByValArray, SizeConst = 16)]
4. public char[] name;
5. [MarshalAs(UnmanagedType.ByValArray, SizeConst = 16)]
6. public char[] cipher;
7. [MarshalAs(UnmanagedType.ByValArray, SizeConst = 256)]
8. public char[] signature;
9. }

结构的使用：

**[csharp]** [view plain](http://blog.csdn.net/xiaobai1593/article/details/7025775) [copy](http://blog.csdn.net/xiaobai1593/article/details/7025775)

1. Info myInfo;
2. myInfo.name = name.ToCharArray();
3. myInfo.cipher = cipher.ToCharArray();
4. myInfo.signature = signature.ToCharArray();

注意：

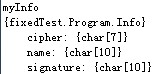
**[csharp]** [view plain](http://blog.csdn.net/xiaobai1593/article/details/7025775) [copy](http://blog.csdn.net/xiaobai1593/article/details/7025775)

1. int size = Marshal.SizeOf(myInfo);

size=16+16+256

可见，获取到的非托管大小为288

但是，查看myInfo对象可以看到其实际大小如下所示：

  
问题：

这种实际大小和固定大小的不一致性，导致了在用Marshal类进行托管对象和非托管对象的转换时，会有如下错误提示：“未能封送类型，因为嵌入数组实例的长度与布局中声明的长度不匹配。”

**小结：**  
MarshalAs这个属性很难用，很容易用错，用好需要对C#、C++和COM数据的布局方式有一定的了解才能做。所以做好使用一些工具来帮你，可以参照我下面的文章：  
<http://blog.csdn.net/Donjuan/archive/2009/02/05/3865026.aspx>  
如果你只是感兴趣的话，那就忘了这个属性吧，在.NET 4.0以后，微软会尽量解决掉这个属性。

## 9、[实例化](http://blog.csdn.net/xiaobai1593/article/details/7025775)

**实例化**是指在[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1)的编程中，通常把用类创建对象的过程称为实例化。格式化有类名，对象名 = new 类名（参数1，参数2...参数n）。

在[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1)的编程中，通常把用类创建对象的过程称为实例化，其格式如下：

如 Date date=new Date();就是用日期类创建了一个日期的对象，就叫对象的实例化。

////类名是你用来定义类的，类相当于一个模板。比如说世界上很多人，我们就可以定义一个Person类:  
Public Person  
{  
Public String Name {get; set;}  
Public String Age {get; set;}  
}  
这是类的创建。  
每个人又是有区别的，那我们就可以根据Person类来创建不同的人，比如说：  
Person p1 = New Person() { Name = "A", Age = "22" }  
Person [p2](https://www.baidu.com/s?wd=p2&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLm1nkmWc4uWTznvDzPyfL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjnkP101P1Tv) = New Person() { Name = "B", Age = "23" }

## [10、C#中的异常处理入门 try catch throw](http://www.cnblogs.com/thinking-jiang/p/5642282.html)

软件运行过程中，如果出现了软件正常运行不应该出现的情况，软件就出现了异常。这时候我们需要去处理这些异常。或者让程序终止，避免出现更严重的错误。或者提示用户进行某些更改让程序可以继续运行下去。

C#编程语言本身就为我们提供了这种异常处理机制。

C# 中的异常是对程序运行时出现的特殊情况的一种响应，比如尝试除以零。或者试图将一个字符串"aaa"转换成整数。

异常提供了一种把程序控制权从某个部分转移到另一个部分的方式。C# 异常处理时建立在四个关键词之上的：**try**、**catch**、**finally** 和 **throw**。

* **try**：一个 try 块标识了一个将被激活的特定的异常的代码块。后跟一个或多个 catch 块。
* **catch**：程序通过异常处理程序捕获异常。catch 关键字表示异常的捕获。
* **finally**：finally 块用于执行给定的语句，不管异常是否被抛出都会执行。例如，如果您打开一个文件，不管是否出现异常文件都要被关闭。
* **throw**：当问题出现时，程序抛出一个异常。使用 throw 关键字来完成。

***语法***

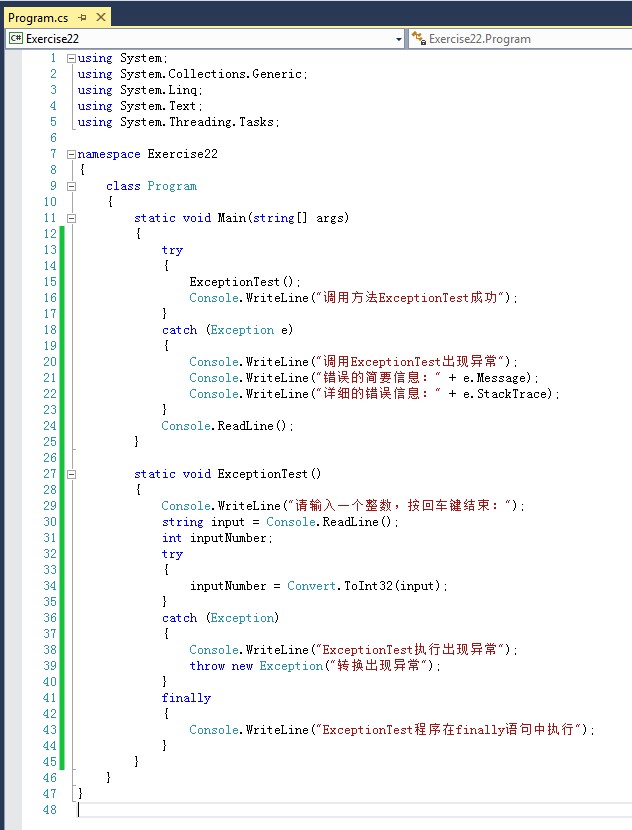
假设一个块将出现异常，一个方法使用 try 和 catch 关键字捕获异常。try/catch 块内的代码为受保护的代码，使用 try/catch 语法如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | try              {                  // 引起异常的语句              }              catch (ExceptionName e1)              {                  // 错误处理代码              }              catch (ExceptionName e2)              {                  // 错误处理代码              }              catch (ExceptionName eN)              {                  // 错误处理代码              }              finally              {                  // 要执行的语句              } |

C#提供了很多种异常，我们在这里只要掌握最基本的异常Exception就可以啦。在catch语句中我们只处理Exception这个最基本的异常。

下面我们看一个具体的例子，来更加清楚的理解一下C#的异常处理。

你创建一个项目。解决方案名称为Sloution22，项目名称为Exercise22，项目类型为控制台程序。点击Program.cs. 在里边加入这些代码。



我在这里简要的说明一下这段代码。

1. 27行-45行 我们新建了一个ExceptionTest 方法。在这个方法中要求用户输入一个整数。如果用户输入一个字符串，我们的程序会产生异常。

2. 39行 我们用throw抛出了一个我们自己定义的异常。在里边加入了我们想传递的一些信息。

3. 41行-44行 finally语句是可选的。只有当需要做一些例如资源释放的操作的时候，才是需要的。不管有没有异常产生 finally部分都会执行。

4. 13行-23行 调用ExceptionTest方法。同时捕获异常。

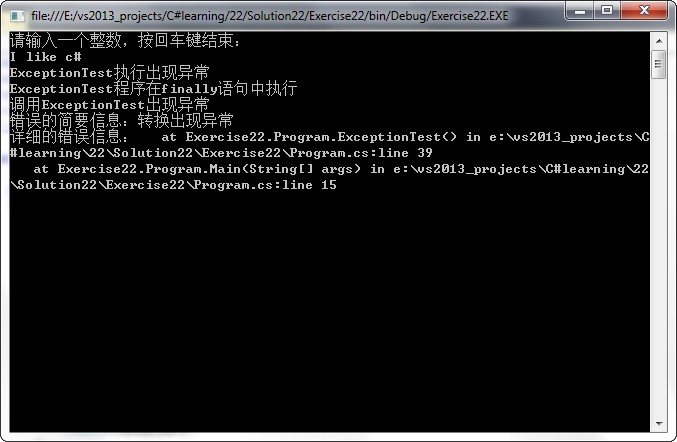
5. 16行 如果调用ExceptionTest方法出现异常，这条语句不会执行。否则正常执行。

6. 21行 输出异常的简要信息

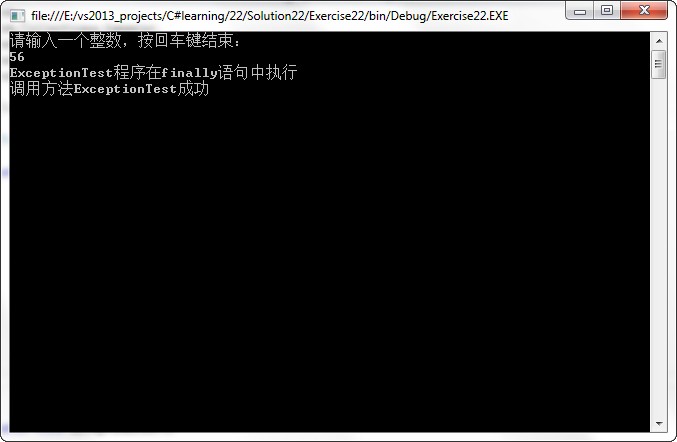
7. 22行 输出异常的详细信息。例如出错的代码位置。

**运行结果**

1. 用户输入字符串，产生异常



2. 用户输入整数，程序正常执行



Eg:

看下面的程序段:  
 try {  
 return a/b;  
   
 } catch (Exception e) {  
 //当b为0的时候这里可以打印提示:除数不能为0  
 }  
异常的通俗原理在于,当出现某些错误的时候,你又不想直接把程序直接关闭,那么就在你捕捉到的异常里进行处理,这样程序可以继续运行,而不是马上关闭.  
上面你返回a/b,正常情况下b不是0,那么就直接返回,不会执行catch里的东西,但是当你的b是0,就无法返回数据,那么异常了,异常然后被catch捕捉到,就执行catch里的语句,然后继续运行下面的代码.

## 11、[C# String.split()用法小结](http://www.cnblogs.com/mingforyou/p/3952357.html)

第一种方法

string s=abcdeabcdeabcde;

string[] sArray=s.Split('c') ;

foreach(string i in sArray)

Console.WriteLine(i.ToString());

输出下面的结果:

ab

deab

deab

de

第二种方法

我们看到了结果是以一个指定的字符进行的分割。使用另一种构造方法对多个字

符进行分割:

string s="abcdeabcdeabcde";

string[] sArray1=s.Split(new char[3]{'c','d','e'}) ;

foreach(string i in sArray1)

Console.WriteLine(i.ToString());

可以输出下面的结果

ab

ab

ab

第三种方法

除了以上的这两种方法以外,第三种方法是使用正则表达式。新建一个控制台项

目。然后先添加 using System.Text.RegularExpressions;

System.Text.RegularExpressions

string content=agcsmallmacsmallgggsmallytx;

string[]resultString=Regex.Split(content,small,RegexOptions.IgnoreCas

e) foreach(string i in resultString)

Console.WriteLine(i.ToString());

输出下面的结果:

agc

mac

ggg

ytx

第四种方法

string str1=我\*\*\*\*\*是\*\*\*\*\*一\*\*\*\*\*个\*\*\*\*\*教\*\*\*\*\*师;

string[] str2;

str1=str1.Replace(\*\*\*\*\*,\*) ;

str2=str1.Split(\*) ;

foreach(string i in str2)

Console.WriteLine(i.ToString());

第五种方法

string str1=我\*\*是\*\*\*\*\*一\*\*\*\*\*个\*\*\*\*\*教\*\*\*\*\*师;

我希望显示的结果为:我是一个教师。

我如果采用上面的第四种方法来做就会产生下面的错误，我   是一个教师。中

间有空格输出，所以输出结果并不是希望的结果，这就又回到了正则表达式了

这时可以采用下面的第五种方法

string str1=我\*\*是\*\*\*\*\*一\*\*\*\*\*个\*\*\*\*\*教\*\*\*\*\*师;

string[] str2 = System.Text.RegularExpressions.Regex.Split(str1,@[\*]+);

foreach(string i in str2)

Console.WriteLine(i.ToString());

## 12、C#异常类相关总结

C#异常类一、基类Exception

C#异常类二、常见的异常类

**1、SystemException类**:该类是System命名空间中所有其他异常类的基类。（建议：公共语言运行时引发的异常通常用此类）

**2、ApplicationException类：**该类表示应用程序发生非致命错误时所引发的异常（建议：应用程序自身引发的异常通常用此类）

**C#异常类三、与参数有关的异常类**

此类异常类均派生于SystemException，用于处理给方法成员传递的参数时发生异常

1、ArgumentException类：该类用于处理参数无效的异常，除了继承来的属性名，此类还提供了string类型的属性ParamName表示引发异常的参数名称。

2、FormatException类：该类用于处理参数格式错误的异常。

**C#异常类四、与成员访问有关的异常**

1、MemberAccessException类：该类用于处理访问类的成员失败时所引发的异常。失败的原因可能的原因是没有足够的访问权限，也可能是要访问的成员根本不存在（类与类之间调用时常用）

2、MemberAccessException类的直接派生类：

i、FileAccessException类：该类用于处理访问字段成员失败所引发的异常

ii、MethodAccessException类：该类用于处理访问方法成员失败所引发的异常

iii、MissingMemberException类：该类用于处理成员不存在时所引发的异常

**C#异常类五、与数组有关的异常**

以下三个类均继承于SystemException类

1、IndexOutOfException类：该类用于处理下标超出了数组长度所引发的异常

2、ArrayTypeMismatchException类：该类用于处理在数组中存储数据类型不正确的元素所引发的异常

3、RankException类：该类用于处理维数错误所引发的异常

**C#异常类六、与IO有关的异常**

1、IOException类：该类用于处理进行文件输入输出操作时所引发的异常。

2、IOException类的5个直接派生类：

i、DirectionNotFoundException类：该类用于处理没有找到指定的目录而引发的异常。

ii、FileNotFoundException类：该类用于处理没有找到文件而引发的异常。

iii、EndOfStreamException类：该类用于处理已经到达流的末尾而还要继续读数据而引发的异常。

iv、FileLoadException类：该类用于处理无法加载文件而引发的异常。

v、PathTooLongException类：该类用于处理由于文件名太长而引发的异常。

**C#异常类七、与算术有关的异常**

1、ArithmeticException类：该类用于处理与算术有关的异常。

2、ArithmeticException类的派生类：

i、DivideByZeroException类：表示整数货十进制运算中试图除以零而引发的异常。

ii、NotFiniteNumberException类：表示浮点数运算中出现无穷打或者非负值时所引发的异常。

## 13、[C#中的Dictionary字典类介绍](http://www.cnblogs.com/txw1958/archive/2012/11/07/csharp-dictionary.html)

**说明**  
    必须包含名空间System.Collection.Generic   
    Dictionary里面的每一个元素都是一个键值对(由二个元素组成：键和值)   
    键必须是唯一的,而值不需要唯一的   
    键和值都可以是任何类型(比如：string, int, 自定义类型，等等)   
    通过一个键读取一个值的时间是接近O(1)   
    键值对之间的偏序可以不定义

**使用方法**：

//定义

Dictionary<string, string> openWith = new Dictionary<string, string>();

//添加元素

openWith.Add("txt", "notepad.exe");

openWith.Add("bmp", "paint.exe");

openWith.Add("dib", "paint.exe");

openWith.Add("rtf", "wordpad.exe");

//取值

Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

//更改值

openWith["rtf"] = "winword.exe";

Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

//遍历key

foreach (string key in openWith.Keys)

{

Console.WriteLine("Key = {0}", key);

}

[复制代码](javascript:void(0);)

//遍历value

foreach (string value in openWith.Values)

{

Console.WriteLine("value = {0}", value);

}

//遍历value, Second Method

Dictionary<string, string>.ValueCollection valueColl = openWith.Values;

foreach (string s in valueColl)

{

Console.WriteLine("Second Method, Value = {0}", s);

}

//遍历字典

foreach (KeyValuePair<string, string> kvp in openWith)

{

Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);

}

//添加存在的元素

try

{

openWith.Add("txt", "winword.exe");

}

catch (ArgumentException)

{

Console.WriteLine("An element with Key = \"txt\" already exists.");

}

//删除元素

openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))

{

Console.WriteLine("Key \"doc\" is not found.");

}

//判断键存在

if (openWith.ContainsKey("bmp")) // True

{

Console.WriteLine("An element with Key = \"bmp\" exists.");

}

**参数为其它类型**

//参数为其它类型

Dictionary<int, string[]> OtherType = new Dictionary<int, string[]>();

OtherType.Add(1, "1,11,111".Split(','));

OtherType.Add(2, "2,22,222".Split(','));

Console.WriteLine(OtherType[1][2]);

**参数为自定义类型**

首先定义类

class DouCube

{

public int Code { get { return \_Code; } set { \_Code = value; } } private int \_Code;

public string Page { get { return \_Page; } set { \_Page = value; } } private string \_Page;

}

然后

//声明并添加元素

Dictionary<int, DouCube> MyType = new Dictionary<int, DouCube>();

for (int i = 1; i <= 9; i++)

{

DouCube element = new DouCube();

element.Code = i \* 100;

element.Page = "http://www.doucube.com/" + i.ToString() + ".html";

MyType.Add(i, element);

}

//遍历元素

foreach (KeyValuePair<int, DouCube> kvp in MyType)

{

Console.WriteLine("Index {0} Code:{1} Page:{2}", kvp.Key, kvp.Value.Code, kvp.Value.Page);

}

**常用属性**

    名称    说明  
    Comparer     获取用于确定字典中的键是否相等的 IEqualityComparer<T>。  
    Count        获取包含在 Dictionary<TKey, TValue> 中的键/值对的数目。  
    Item         获取或设置与指定的键相关联的值。  
    Keys         获取包含 Dictionary<TKey, TValue> 中的键的集合。  
    Values       获取包含 Dictionary<TKey, TValue> 中的值的集合。

**常用方法**  
    名称    说明  
    Add                 将指定的键和值添加到字典中。  
    Clear               从 Dictionary<TKey, TValue> 中移除所有的键和值。  
    ContainsKey         确定 Dictionary<TKey, TValue> 是否包含指定的键。  
    ContainsValue       确定 Dictionary<TKey, TValue> 是否包含特定值。  
    Equals(Object)      确定指定的 Object 是否等于当前的 Object。 （继承自 Object。）  
    Finalize            允许对象在“垃圾回收”回收之前尝试释放资源并执行其他清理操作。 （继承自 Object。）  
    GetEnumerator       返回循环访问 Dictionary<TKey, TValue> 的枚举器。  
    GetHashCode         用作特定类型的哈希函数。 （继承自 Object。）  
    GetObjectData       实现 System.Runtime.Serialization.ISerializable 接口，并返回序列化 Dictionary<TKey, TValue> 实例所需的数据。  
    GetType             获取当前实例的 Type。 （继承自 Object。）  
    MemberwiseClone     创建当前 Object 的浅表副本。 （继承自 Object。）  
    OnDeserialization    实现 System.Runtime.Serialization.ISerializable 接口，并在完成反序列化之后引发反序列化事件。  
    Remove              从 Dictionary<TKey, TValue> 中移除所指定的键的值。  
    ToString            返回表示当前对象的字符串。 （继承自 Object。）  
    TryGetValue         获取与指定的键相关联的值

附：

Dictionary<string, string>是一个泛型

他本身有集合的功能有时候可以把它看成数组

他的结构是这样的：Dictionary<[key], [value]>

他的特点是存入对象是需要与[key]值一一对应的存入该泛型

通过某一个一定的[key]去找到对应的值

举个例子：

//实例化对象

Dictionary<int, string> dic = new Dictionary<int, string>();

//对象打点添加

dic.Add(1, "one");

dic.Add(2, "two");

dic.Add(3, "one");

//提取元素的方法

string a = dic[1];

string b = dic[2];

string c = dic[3];

//1、2、3是键，分别对应“one”“two”“one”

//上面代码中分别把值赋给了a,b,c

//注意,键相当于找到对应值的唯一标识，所以不能重复

//但是值可以重复

如果你还看不懂我最后给你举一个通俗的例子

有一缸米，你想在在每一粒上都刻上标记，不重复，相当于“键”当你找的时候一一对应不会找错，这就是这个泛型的键的-作用，而米可以一样，我的意思你明白了吧？

c# 对dictionary类进行排序用什么接口实现

如果使用.Net Framework 3.5的话，事情就很简单了。呵呵。

如果不是的话，还是自己写排序吧。

using System;

using System.Collections.Generic;

using System.Text;

using System.Linq;

namespace DictionarySorting

{

class Program

{

static void Main(string[] args)

{

Dictionary<int, string> dic = new Dictionary<int, string>();

dic.Add(1, "HaHa");

dic.Add(5, "HoHo");

dic.Add(3, "HeHe");

dic.Add(2, "HiHi");

dic.Add(4, "HuHu");

var result = from pair in dic orderby pair.Key select pair;

foreach (KeyValuePair<int, string> pair in result)

{

Console.WriteLine("Key:{0}, Value:{1}", pair.Key, pair.Value);

}

Console.ReadKey();

}

}

}

【执行结果】

Key:1, Value:HaHa

Key:2, Value:HiHi

Key:3, Value:HeHe

Key:4, Value:HuHu

Key:5, Value:HoHo

Dictionary的基本用法。假如

需求：现在要导入一批数据，这些数据中有一个称为公司的字段是我们数据库里已经存在了的，目前我们需要把每个公司名字转为ID后才存入数据库。

分析：每导一笔记录的时候，就把要把公司的名字转为公司的ID，这个不应该每次都查询一下数据库的，因为这太耗数据库的性能了。

解决方案：在业务层里先把所有的公司名称及相应的公司ID一次性读取出来，然后存放到一个Key和Value的键值对里，然后实现只要把一个公司的名字传进去，就可以得到此公司相应的公司ID，就像查字典一样。对，我们可以使用字典Dictionary操作这些数据。

示例：SetKeyValue()方法相应于从数据库里读取到了公司信息。

/// <summary>  
/// 定义Key为string类型，Value为int类型的一个Dictionary  
/// </summary>  
/// <returns></returns>  
protected Dictionary<string, int> SetKeyValue()  
{  
Dictionary<string, int> dic = new Dictionary<string, int>();  
dic.Add("公司1", 1);  
dic.Add("公司2", 2);  
dic.Add("公司3", 3);  
dic.Add("公司4", 4);  
return dic;  
}

/// <summary>  
/// 得到根据指定的Key行到Value  
/// </summary>  
protected void GetKeyValue()  
{  
Dictionary<string, int> myDictionary = SetKeyValue();  
//测试得到公司2的值  
int directorValue = myDictionary["公司2"];  
Response.Write("公司2的value是：" + directorValue.ToString());  
}

using System

using System.Threading;

**Theradsleep（）用法**

namespace ThreadTest

  {

 public class A  
    {  
        public static void B()  
        {  
            while (true)  
            {  
                Console.WriteLine("A.B is running in its own thread.");  
            }  
        }  
    }

    public class C  
    {  
        public static void Main()  
        {  
            Thread t = new Thread(new ThreadStart(A.B));  
            t.Start();  
            Thread.Sleep(5);  
            t.Abort();  
            t.Join();  
            Console.WriteLine("A.B has finished");  
            Console.ReadLine();  
        }  
    }

}

运行后显示若干行"A.B is running in its own thread."和一行"A.B has finished";

若取消程序中的“Thread.Sleep(5);”句，则运行后只显示"A.B has finished"。

理解：如无“Thread.Sleep(5);”句，则线程oThread一开始后即被中止，根本没有执行到其中的输入字符串的代码，但增加此句后，使主线程(执行Main方法的线程)休眠(sleep)了5毫秒才执行t.Abort();，这段5毫秒的时间使t线程有时间执行了输出字符串代码,由此可见该程序运行完t.start().后,有两个并行的线程,主线程和t线程,两者各自运行,可视为并行,而主线程中有控制t线程的语句,主线程休眠5毫秒时,t线程继续运行.而如果没有休眠指令,在主线程中由于t.start()和指令的下一条就是t.Abort()和t.Join(),所以使得已经在同步运行t线程立即停止了.通过该段小程序,可以帮助我们了解多线程的概念.

## 14、多线程有什么用？

1。单进程单线程：一个人在一个桌子上吃菜。  
2。单进程多线程：多个人在同一个桌子上一起吃菜。  
3。多进程单线程：多个人每个人在自己的桌子上吃菜。

多线程的问题是多个人同时吃一道菜的时候容易发生争抢，例如两个人同时夹一个菜，一个人刚伸出筷子，结果伸到的时候已经被夹走菜了。。。此时就必须等一个人夹一口之后，在还给另外一个人夹菜，也就是说资源共享就会发生冲突争抢。

1。对于 Windows 系统来说，【开桌子】的开销很大，因此 Windows 鼓励大家在一个桌子上吃菜。因此 Windows 多线程学习重点是要大量面对资源争抢与同步方面的问题。

2。对于 Linux 系统来说，【开桌子】的开销很小，因此 Linux 鼓励大家尽量每个人都开自己的桌子吃菜。这带来新的问题是：坐在两张不同的桌子上，说话不方便。因此，Linux 下的学习重点大家要学习进程间通讯的方法。

--  
补充：有人对这个开桌子的开销很有兴趣。我把这个问题推广说开一下。

开桌子的意思是指创建进程。开销这里主要指的是时间开销。  
可以做个实验：创建一个进程，在进程中往内存写若干数据，然后读出该数据，然后退出。此过程重复 1000 次，相当于创建/销毁进程 1000 次。在我机器上的测试结果是：   
UbuntuLinux：耗时 0.8 秒   
Windows7：耗时 79.8 秒   
两者开销大约相差一百倍。

这意味着，在 Windows 中，进程创建的开销不容忽视。换句话说就是，Windows 编程中不建议你创建进程，如果你的程序架构需要大量创建进程，那么最好是切换到 Linux 系统。

大量创建进程的典型例子有两个，一个是 gnu autotools 工具链，用于编译很多开源代码的，他们在 Windows 下编译速度会很慢，因此软件开发人员最好是避免使用 Windows。另一个是服务器，某些服务器框架依靠大量创建进程来干活，甚至是对每个用户请求就创建一个进程，这些服务器在 Windows 下运行的效率就会很差。这"可能"也是放眼全世界范围，Linux 服务器远远多于 Windows 服务器的原因。

--  
再次补充：如果你是写服务器端应用的，其实在现在的网络服务模型下，开桌子的开销是可以忽略不计的，因为现在一般流行的是按照 CPU 核心数量开进程或者线程，开完之后在数量上一直保持，进程与线程内部使用协程或者异步通信来处理多个并发连接，因而开进程与开线程的开销可以忽略了。

另外一种新的开销被提上日程：核心切换开销。

现代的体系，一般 CPU 会有多个核心，而多个核心可以同时运行多个不同的线程或者进程。

当每个 CPU 核心运行一个进程的时候，由于每个进程的资源都独立，所以 CPU 核心之间切换的时候无需考虑上下文。

当每个 CPU 核心运行一个线程的时候，由于每个线程需要共享资源，所以这些资源必须从 CPU 的一个核心被复制到另外一个核心，才能继续运算，这占用了额外的开销。换句话说，在 CPU 为多核的情况下，多线程在性能上不如多进程。

因而，当前面向多核的服务器端编程中，需要习惯多进程而非多线程

## 15 Thead

### [C#中的线程(一) 线程同步基础](http://www.cnblogs.com/miniwiki/archive/2010/06/18/1760611.html)

#### 概述与概念

   C#支持通过多线程并行地执行代码，一个线程有它独立的执行路径，能够与其它的线程同时地运行。一个C#程序开始于一个单线程，这个单线程是被CLR和操作系统（也称为“主线程”）自动创建的，并具有多线程创建额外的线程。这里的一个简单的例子及其输出：

*除非被指定，否则所有的例子都假定以下命名空间被引用了：*   
   ***using System;   
   using System.Threading;***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class ThreadTest {    static void Main() {      Thread t = new Thread (WriteY);      t.Start();                          // Run WriteY on the new thread      while (true) Console.Write ("x");   // Write 'x' forever    }      static void WriteY() {      while (true) Console.Write ("y");   // Write 'y' forever    }  } |

[](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_2.png)

   主线程创建了一个新线程“t”，它运行了一个重复打印字母"y"的方法，同时主线程重复但因字母“x”。CLR分配每个线程到它自己的内存堆栈上，来保证局部变量的分离运行。在接下来的方法中我们定义了一个局部变量，然后在主线程和新创建的线程上同时地调用这个方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static void Main() {    new Thread (Go).Start();      // Call Go() on a new thread    Go();                         // Call Go() on the main thread  }    static void Go() {    // Declare and use a local variable - 'cycles'    for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');  } |

[image](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_4.png)

   变量**cycles**的副本分别在各自的内存堆栈中创建，输出也一样，可预见，会有10个问号输出。当线程们引用了一些公用的目标实例的时候，他们会共享数据。下面是实例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ThreadTest {   bool done;     static void Main() {     ThreadTest tt = new ThreadTest();   // Create a common instance     new Thread (tt.Go).Start();     tt.Go();   }     // Note that Go is now an instance method   void Go() {     if (!done) { done = true; Console.WriteLine ("Done"); }   }  }  因为在相同的<b>ThreadTest</b>实例中，两个线程都调用了<b>Go()</b>，它们共享了<b>done</b>字段，这个结果输出的是一个"Done"，而不是两个。 |
| 1 | <a href="<http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_6.png>"><img src="<http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_thumb_2.png>" alt="image" title="image" style="display: inline; margin-left: 0px; margin-right: 0px; border-width: 0px;" width="640" height="45" border="0" align="left"></a> |

  静态字段提供了另一种在线程间共享数据的方式，下面是一个以**done**为静态字段的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class ThreadTest {   static bool done;    // Static fields are shared between all threads     static void Main() {     new Thread (Go).Start();     Go();   }     static void Go() {     if (!done) { done = true; Console.WriteLine ("Done"); }   }  } |

  上述两个例子足以说明， 另一个关键概念， 那就是*线程安全*(或反之，它的不足之处! ) 输出实际上是不确定的：它可能(虽然不大可能) ， "Done" ，可以被打印两次。然而，如果我们在**Go**方法里调换指令的顺序， "Done"被打印两次的机会会大幅地上升：

|  |  |
| --- | --- |
| 1  2  3 | static void Go() {    if (!done) { Console.WriteLine ("Done"); done = true; }  } |

[](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_8.png)

问题就是一个线程在判断if块的时候，正好另一个线程正在执行WriteLine语句——在它将done设置为true之前。

补救措施是当读写公共字段的时候，提供一个*排他锁*；C#提供了*lock*语句来达到这个目的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ThreadSafe {    static bool done;    static object locker = new object();      static void Main() {      new Thread (Go).Start();      Go();    }      static void Go() {      lock (locker) {        if (!done) { Console.WriteLine ("Done"); done = true; }      }    }  } |

   当两个线程争夺一个锁的时候（在这个例子里是locker），一个线程等待，或者说被*阻止*到那个锁变的可用。在这种情况下，就确保了在同一时刻只有一个线程能进入临界区，所以"Done"只被打印了1次。代码以如此方式在不确定的多线程环境中被叫做*线程安全*。

   临时暂停，或阻止是多线程的协同工作，同步活动的本质特征。等待一个排它锁被释放是一个线程被阻止的原因，另一个原因是线程想要暂停或*Sleep*一段时间：

|  |  |
| --- | --- |
| 1 | Thread.Sleep (TimeSpan.FromSeconds (30));         // Block for 30 seconds |

一个线程也可以使用它的Join方法来等待另一个线程结束：

|  |  |
| --- | --- |
| 1  2  3 | Thread t = new Thread (Go);           // Assume Go is some static method  t.Start();  t.Join();                             // Wait (block) until thread t ends |

一个线程，一旦被阻止，它就不再消耗CPU的资源了。

**线程是如何工作的**

   线程被一个线程协调程序管理着——一个CLR委托给操作系统的函数。线程协调程序确保将所有活动的线程被分配适当的执行时间；并且那些等待或阻止的线程——比如说在排它锁中、或在用户输入——都是不消耗CPU时间的。

   在单核处理器的电脑中，线程协调程序完成一个时间片之后迅速地在活动的线程之间进行切换执行。这就导致“波涛汹涌”的行为，例如在第一个例子，每次重复的X 或 Y 块相当于分给线程的时间片。在Windows XP中时间片通常在10毫秒内选择要比CPU开销在处理线程切换的时候的消耗大的多。(即通常在几微秒区间)

   在多核的电脑中，多线程被实现成混合时间片和真实的并发——不同的线程在不同的CPU上运行。这几乎可以肯定仍然会出现一些时间切片, 由于操作系统的需要服务自己的线程，以及一些其他的应用程序。

   线程由于外部因素（比如时间片）被中断被称为被抢占，在大多数情况下,一个线程方面在被抢占的那一时那一刻就失去了对它的控制权。

**线程 vs. 进程**

属于一个单一的应用程序的所有的线程逻辑上被包含在一个进程中，进程指一个应用程序所运行的操作系统单元。

    线程于进程有某些相似的地方：比如说进程通常以时间片方式与其它在电脑中运行的进程的方式与一个C#程序线程运行的方式大致相同。二者的关键区别在于进程彼此是完全隔绝的。线程与运行在相同程序其它线程共享(堆heap)内存，这就是线程为何如此有用：一个线程可以在后台读取数据，而另一个线程可以在前台展现已读取的数据。

**何时使用多线程**

    多线程程序一般被用来在后台执行耗时的任务。主线程保持运行，并且工作线程做它的后台工作。对于Windows Forms程序来说，如果主线程试图执行冗长的操作，键盘和鼠标的操作会变的迟钝，程序也会失去响应。由于这个原因，应该在工作线程中运行一个耗时任务时添加一个工作线程，即使在主线程上有一个有好的提示“处理中...”，以防止工作无法继续。这就避免了程序出现由操作系统提示的“没有相应”，来诱使用户强制结束程序的进程而导致错误。模式对话框还允许实现“取消”功能，允许继续接收事件，而实际的任务已被工作线程完成。*BackgroundWorker*恰好可以辅助完成这一功能。

   在没有用户界面的程序里，比如说Windows Service， 多线程在当一个任务有潜在的耗时，因为它在等待另台电脑的响应（比如一个应用服务器，数据库服务器，或者一个客户端）的实现特别有意义。用工作线程完成任务意味着主线程可以立即做其它的事情。

   另一个多线程的用途是在方法中完成一个复杂的计算工作。这个方法会在多核的电脑上运行的更快，如果工作量被多个线程分开的话（使用Environment.ProcessorCount属性来侦测处理芯片的数量）。

   一个C#程序称为多线程的可以通过2种方式：明确地创建和运行多线程，或者使用.NET framework的暗中使用了多线程的特性——比如*BackgroundWorker*类, *线程池*，*threading timer*，远程服务器，或Web Services或ASP.NET程序。在后面的情况，人们别无选择，必须使用多线程；一个单线程的ASP.NET web server不是太酷，即使有这样的事情；幸运的是，应用服务器中多线程是相当普遍的；唯一值得关心的是提供适当锁机制的静态变量问题。

**何时不要使用多线程**

    多线程也同样会带来缺点，最大的问题是它使程序变的过于复杂，拥有多线程本身并不复杂，复杂是的线程的交互作用，这带来了无论是否交互是否是有意的，都会带来较长的开发周期，以及带来间歇性和非重复性的bugs。因此，要么多线程的交互设计简单一些，要么就根本不使用多线程。除非你有强烈的重写和调试欲望。

当用户频繁地分配和切换线程时，多线程会带来增加资源和CPU的开销。在某些情况下，太多的I/O操作是非常棘手的，当只有一个或两个工作线程要比有众多的线程在相同时间执行任务块的多。稍后我们将实现生产者/耗费者 队列，它提供了上述功能。

#### 2. 创建和开始使用多线程

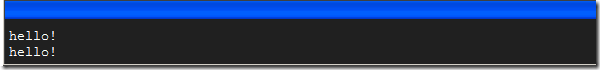
   线程用Thread类来创建， 通过ThreadStart委托来指明方法从哪里开始运行，下面是ThreadStart委托如何定义的：

|  |  |
| --- | --- |
| 1 | public delegate void ThreadStart(); |

   调用Start方法后，线程开始运行，线程一直到它所调用的方法返回后结束。下面是一个例子，使用了C#的语法创建TheadStart委托：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class ThreadTest {    static void Main() {      Thread t = new Thread (new ThreadStart (Go));      t.Start();   // Run Go() on the new thread.      Go();        // Simultaneously run Go() in the main thread.    }    static void Go() { Console.WriteLine ("hello!"); } |

在这个例子中，线程t执行Go()方法，大约与此同时主线程也调用了Go()，结果是两个几乎同时hello被打印出来：

[](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_10.png)

一个线程可以通过C#堆委托简短的语法更便利地创建出来：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static void Main() {    Thread t = new Thread (Go);    // No need to explicitly use ThreadStart    t.Start();    ...  }  static void Go() { ... }  在这种情况，ThreadStart被编译器自动推断出来，另一个快捷的方式是使用匿名方法来启动线程： |
| 1  2  3  4 | static void Main() {    Thread t = new Thread (delegate() { Console.WriteLine ("Hello!"); });    t.Start();  } |

  线程有一个IsAlive属性，在调用Start()之后直到线程结束之前一直为true。一个线程一旦结束便不能重新开始了。

#### 3 将数据传入ThreadStart中

话又说回来，在上面的例子里，我们想更好地区分开每个线程的输出结果，让其中一个线程输出大写字母。我们传入一个状态字到Go中来完成整个任务，但我们不能使用ThreadStart委托，因为它不接受参数，所幸的是，.NET framework定义了另一个版本的委托叫做ParameterizedThreadStart， 它可以接收一个单独的object类型参数：

|  |  |
| --- | --- |
| 1  2 | public delegate void ParameterizedThreadStart (object obj);  之前的例子看起来是这样的： |
| 1 |  |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class ThreadTest {    static void Main() {      Thread t = new Thread (Go);      t.Start (true);             // == Go (true)      Go (false);    }    static void Go (object upperCase) {      bool upper = (bool) upperCase;      Console.WriteLine (upper ? "HELLO!" : "hello!");    } |

[](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_12.png)

  在整个例子中，编译器自动推断出ParameterizedThreadStart委托，因为Go方法接收一个单独的object参数，就像这样写：

|  |  |
| --- | --- |
| 1  2 | Thread t = new Thread (new ParameterizedThreadStart (Go));  t.Start (true); |

ParameterizedThreadStart的特性是在使用之前我们必需对我们想要的类型（这里是bool）进行装箱操作，并且它只能接收一个参数。

  一个替代方案是使用一个匿名方法调用一个普通的方法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | static void Main() {    Thread t = new Thread (delegate() { WriteText ("Hello"); });    t.Start();  }  static void WriteText (string text) { Console.WriteLine (text); } |

  优点是目标方法（这里是WriteText），可以接收任意数量的参数，并且没有装箱操作。不过这需要将一个外部变量放入到匿名方法中，向下面的一样：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static void Main() {    string text = "Before";    Thread t = new Thread (delegate() { WriteText (text); });    text = "After";    t.Start();  }  static void WriteText (string text) { Console.WriteLine (text); } |

[image](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_14.png)

  匿名方法打开了一种怪异的现象，当外部变量被后来的部分修改了值的时候，可能会透过外部变量进行无意的互动。有意的互动（通常通过字段）被认为是足够了！一旦线程开始运行了，外部变量最好被处理成只读的——除非有人愿意使用适当的锁。

  另一种较常见的方式是将对象实例的方法而不是静态方法传入到线程中，对象实例的属性可以告诉线程要做什么，如下列重写了原来的例子：

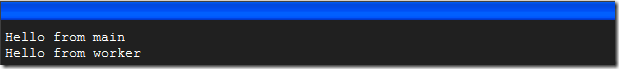
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class ThreadTest {    bool upper;      static void Main() {      ThreadTest instance1 = new ThreadTest();      instance1.upper = true;      Thread t = new Thread (instance1.Go);      t.Start();      ThreadTest instance2 = new ThreadTest();      instance2.Go();        // 主线程——运行 upper=false    }      void Go() { Console.WriteLine (upper ? "HELLO!" : "hello!"); } |

**命名线程**

  线程可以通过它的Name属性进行命名，这非产有利于调试：可以用Console.WriteLine打印出线程的名字，Microsoft Visual Studio可以将线程的名字显示在调试工具栏的位置上。线程的名字可以在被任何时间设置——但只能设置一次，重命名会引发异常。

  程序的主线程也可以被命名，下面例子里主线程通过CurrentThread命名：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class ThreadNaming {    static void Main() {      Thread.CurrentThread.Name = "main";      Thread worker = new Thread (Go);      worker.Name = "worker";      worker.Start();      Go();    }    static void Go() {      Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);    }  } |

[](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_12936/image_16.png)

 **前台和后台线程**

  线程默认为前台线程，这意味着任何前台线程在运行都会保持程序存活。C#也支持后台线程，当所有前台线程结束后，它们不维持程序的存活。

  改变线程从前台到后台不会以任何方式改变它在CPU协调程序中的优先级和状态。

  线程的IsBackground属性控制它的前后台状态，如下实例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class PriorityTest {    static void Main (string[] args) {      Thread worker = new Thread (delegate() { Console.ReadLine(); });      if (args.Length > 0) worker.IsBackground = true;      worker.Start();    }  } |

### [C#中的线程(二) 线程同步基础](http://www.cnblogs.com/miniwiki/archive/2010/06/18/1760611.html)

#### ****1 同步要领****

下面的表格列展了.NET对协调或同步线程动作的可用的工具：

**简易阻止方法**

|  |  |
| --- | --- |
| **构成** | **目的** |
| Sleep | 阻止给定的时间周期 |
| Join | 等待另一个线程完成 |

**锁系统**

|  |  |  |  |
| --- | --- | --- | --- |
| **构成** | **目的** | **跨进程？** | **速度** |
| lock | 确保只有一个线程访问某个资源或某段代码。 | 否 | 快 |
| Mutex | 确保只有一个线程访问某个资源或某段代码。可被用于防止一个程序的多个实例同时运行。 | 是 | 中等 |
| Semaphore | 确保不超过指定数目的线程访问某个资源或某段代码。 | 是 | 中等 |

                   （同步的情况下也提够自动锁。）

**信号系统**

|  |  |  |  |
| --- | --- | --- | --- |
| **构成** | **目的** | **跨进程？** | **速度** |
| EventWaitHandle | 允许线程等待直到它受到了另一个线程发出信号。 | 是 | 中等 |
| Wait 和 Pulse\* | 允许一个线程等待直到自定义阻止条件得到满足。 | 否 | 中等 |

**非阻止同步系统\***

|  |  |  |  |
| --- | --- | --- | --- |
| **构成** | **目的** | **跨进程？** | **速度** |
| Interlocked\* | 完成简单的非阻止原子操作。 | 是(内存共享情况下) | 非常快 |
| volatile\* | 允许安全的非阻止在锁之外使用个别字段。 | 非常快 |  |

\* 代表页面将转到第四部分

**1.1   阻止 (Blocking)**

     当一个线程通过上面所列的方式处于等待或暂停的状态，被称为被阻止。一旦被阻止，线程立刻放弃它被分配的

CPU时间，将它的ThreadState属性添加为WaitSleepJoin状态，不在安排时间直到停止阻止。停止阻止在任意四种

情况下发生（关掉电脑的电源可不算！）：

* 阻止的条件已得到满足
* 操作超时（如果timeout被指定了）
* 通过Thread.Interrupt中断了
* 通过Thread.Abort放弃了

   当线程通过（不建议）Suspend 方法暂停，不认为是被阻止了。

**1.2  休眠 和 轮询**

    调用Thread.Sleep阻止当前的线程指定的时间（或者直到中断）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | static void Main() {      Thread.Sleep (0); // 释放CPU时间片      Thread.Sleep (1000); // 休眠1000毫秒      Thread.Sleep (TimeSpan.FromHours (1)); // 休眠1小时      Thread.Sleep (Timeout.Infinite); // 休眠直到中断  } |

    更确切地说，Thread.Sleep放弃了占用CPU，请求不在被分配时间直到给定的时间经过。Thread.Sleep(0)放弃

CPU的时间刚刚够其它在时间片队列里的活动线程（如果有的话）被执行。

    Thread.Sleep在阻止方法中是唯一的暂停汲取Windows Forms程序的Windows消息的方法，或COM环境中用于

单元模式。这在Windows Forms程序中是一个很大的问题，任何对主UI线程的阻止都将使程序失去相应。因此一般避

免这样使用，无论信息汲取是否被“技术地”暂定与否。由COM遗留下来的宿主环境更为复杂，在一些时候它决定停止，

而却保持信息的汲取存活。微软的 Chris Brumm 在他的[博客](http://www.zxproxy.com/browse.php?u=QnY4aUx6a2lNM1pob0tBeG92NXdvMjBpTDJXbHFKMWdNRD09&b=6)中讨论这个问题。（搜索： 'COM "Chris Brumme"'）

    线程类同时也提供了一个SpinWait方法，它使用轮询CPU而非放弃CPU时间的方式，保持给定的迭代次数进行“无用

地繁忙”。50迭代可能等同于停顿大约一微秒,虽然这将取决于CPU的速度和负载。从技术上讲，SpinWait并不是一个阻

止的方法：一个处于spin-waiting的线程的ThreadState不是WaitSleepJoin状态，并且也不会被其它的线程过早的中断

(Interrupt)。SpinWait很少被使用，它的作用是等待一个在极短时间（可能小于一微秒）内可准备好的可预期的资源，

而不用调用Sleep方法阻止线程而浪费CPU时间。不过，这种技术的优势只有在多处理器计算机：对单一处理器的电脑，

直到轮询的线程结束了它的时间片之前,一个资源没有机会改变状态，这有违它的初衷。并且调用SpinWait经常会花费较

长的时间这本身就浪费了CPU时间。

**1.3  阻止 vs. 轮询**

    线程可以等待某个确定的条件来明确轮询使用一个轮询的方式，比如：

|  |  |
| --- | --- |
| 1 | while (!proceed); |

     或者：

|  |  |
| --- | --- |
| 1 | while (DateTime.Now < nextStartTime); |

   这是非常浪费CPU时间的：对于CLR和操作系统而言，线程进行了一个重要的计算，所以分配了相应的资源！在这种状态

下的轮询线程不算是阻止，不像一个线程等待一个EventWaitHandle（一般使用这样的信号任务来构建）。

   阻止和轮询组合使用可以产生一些变换：

|  |  |
| --- | --- |
| 1 | while (!proceed) Thread.Sleep (x); // "轮询休眠!" |

   x越大，CPU效率越高，折中方案是增大潜伏时间，任何20ms的花费是微不足道的，除非循环中的条件是极其复杂的。

   除了稍有延迟，这种轮询和休眠的方式可以结合的非常好。（但有并发问题，在第四部分讨论）可能它最大的用处在于

程序员可以放弃使用复杂的信号结构 来工作了。

**1.4   使用Join等待一个线程完成**

    你可以通过Join方法阻止线程直到另一个线程结束：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class JoinDemo {        static void Main() {        Thread t = new Thread (delegate() { Console.ReadLine();});        t.Start();        t.Join(); // 等待直到线程完成        Console.WriteLine ("Thread t's ReadLine complete!");        }    } |

    Join方法也接收一个使用毫秒或用TimeSpan类的超时参数，当Join超时是返回false，如果线程已终止，则返回true 。

Join所带的超时参数非常像Sleep方法，实际上下面两行代码几乎差不多:

|  |  |
| --- | --- |
| 1  2  3 | Thread.Sleep (1000);    Thread.CurrentThread.Join (1000); |

  （他们的区别明显在于单线程的应用程序域与COM互操作性，源于先前描述Windows信息汲取部分：在阻止时，Join

保持信息汲取，Sleep暂停信息汲取。）

#### ****2.  锁和线程安全****

   锁实现互斥的访问，被用于确保在同一时刻只有一个线程可以进入特殊的代码片段，考虑下面的类：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class ThreadUnsafe {      static int val1, val2;      static void Go() {          if (val2 != 0) Console.WriteLine (val1 / val2);              val2 = 0;      }    } |

   这不是线程安全的：如果Go方法被两个线程同时调用，可能会得到在某个线程中除数为零的错误，因为val2可能被一个

线程设置为零，而另一个线程刚好执行到if和Console.WriteLine语句。

   下面用lock来修正这个问题：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class ThreadSafe {      static object locker = new object();      static int val1, val2;      static void Go() {        lock (locker) {        if (val2 != 0) Console.WriteLine (val1 / val2);            val2 = 0;            }        }    } |

    在同一时刻只有一个线程可以锁定同步对象（在这里是locker），任何竞争的的其它线程都将被阻止，直到这个锁被释放。如果有大于一个的线程竞争这个锁，那么他们将形成称为“就绪队列”的队列，以先到先得的方式授权锁。互斥锁有时被称之对由锁所保护的内容强迫串行化访问，因为一个线程的访问不能与另一个重叠。在这个例子中，我们保护了Go方法的逻辑，以及val1 和val2字段的逻辑。

    一个等候竞争锁的线程被阻止将在ThreadState上为WaitSleepJoin状态。稍后我们将讨论一个线程通过另一个线程调用

Interrupt或Abort方法来强制地被释放。这是一个相当高效率的技术可以被用于结束工作线程。

   C#的lock 语句实际上是调用Monitor.Enter和Monitor.Exit，中间夹杂try-finally语句的简略版，下面是实际发生在之前例

子中的Go方法:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Monitor.Enter (locker);    try {      if (val2 != 0) Console.WriteLine (val1 / val2);    val2 = 0;  }  finally { Monitor.Exit (locker);  } |

   在同一个对象上,在调用第一个之前Monitor.Enter而先调用了Monitor.Exit将引发异常。

   Monitor 也提供了TryEnter方法来实现一个超时功能——也用毫秒或TimeSpan，如果获得了锁返回true，反之没有获得返回false，因为超时了。TryEnter也可以没有超时参数，“测试”一下锁，如果锁不能被获取的话就立刻超时。

**2.1  选择同步对象**

   任何对所有有关系的线程都可见的对象都可以作为同步对象，但要服从一个硬性规定：它必须是引用类型。也强烈建议同步对象最好私有在类里面（比如一个私有实例字段）防止无意间从外部锁定相同的对象。服从这些规则，同步对象可以兼对象和保护两种作用。比如下面List ：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class ThreadSafe {    List <string> list = new List <string>();    void Test() {    lock (list) {    list.Add ("Item 1");    ... |

    一个专门字段是常用的(如在先前的例子中的locker) , 因为它可以精确控制锁的范围和粒度。用对象或类本身的类型作为一个同步对象，即：

lock (this) { ... }

或：

lock (typeof (Widget)) { ... }    // 保护访问静态

   是不好的，因为这潜在的可以在公共范围访问这些对象。

   锁并没有以任何方式阻止对同步对象本身的访问，换言之，x.ToString()不会由于另一个线程调用lock(x) 而被阻止，两者都要调用ock(x) 来完成阻止工作。

**2.2   嵌套锁定**

   线程可以重复锁定相同的对象，可以通过多次调用Monitor.Enter或lock语句来实现。当对应编号的Monitor.Exit被调用或最外面的lock语句完成后，对象那一刻被解锁。这就允许最简单的语法实现一个方法的锁调用另一个锁：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | static object x = new object();      static void Main() {      lock (x) {      Console.WriteLine ("I have the lock");          Nest();      Console.WriteLine ("I still have the lock");    }          在这锁被释放  }  static void Nest() {      lock (x) {           ...      }            释放了锁？没有完全释放！    } |

   线程只能在最开始的锁或最外面的锁时被阻止。

**2.3  何时进行锁定**

     作为一项基本规则，任何和多线程有关的会进行读和写的字段应当加锁。甚至是极平常的事情——单一字段的赋值操作，都必须考虑到同步问题。在下面的例子中Increment和Assign 都不是线程安全的：

|  |  |
| --- | --- |
| 1  2  3  4  5 | class ThreadUnsafe {      static int x;      static void Increment() { x++; }      static void Assign() { x = 123; }  } |

  下面是Increment 和 Assign 线程安全的版本：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class ThreadUnsafe {      static object locker = new object();      static int x;      static void Increment() { lock (locker) x++; }      static void Assign() { lock (locker) x = 123; }    } |

   作为锁定另一个选择，在一些简单的情况下，你可以使用非阻止同步，在第四部分讨论（即使像这样的语句需要同步的原因）。

**2.4   锁和原子操作**

    如果有很多变量在一些锁中总是进行读和写的操作，那么你可以称之为原子操作。我们假设x 和 y不停地读和赋值，他们在锁内通过

locker锁定：

    lock (locker) { if (x != 0) y /= x; }

   你可以认为x 和 y 通过原子的方式访问，因为代码段没有被其它的线程分开 或 抢占，别的线程改变x 和 y是无效的输出，你永远不会得到除数为零的错误，保证了x 和 y总是被相同的排他锁访问。

**2.5  性能考量**

   锁定本身是非常快的，一个锁在没有堵塞的情况下一般只需几十纳秒（十亿分之一秒）。如果发生堵塞，任务切换带来的开销接近于数微秒（百万分之一秒）的范围内，尽管在线程重组实际的安排时间之前它可能花费数毫秒（千分之一秒）。而相反，与此相形见绌的是该使用锁而没使用的结果就是带来数小时的时间，甚至超时。如果耗尽并发，锁定会带来反作用，死锁和争用锁，耗尽并发由于太多的代码被放置到锁语句中了，引起其它线程不必要的被阻止。死锁是两线程彼此等待被锁定的内容，导致两者都无法继续下去。争用锁是两个线程任一个都可以锁定某个内容，如果“错误”的线程获取了锁，则导致程序错误。

   对于太多的同步对象死锁是非常容易出现的症状，一个好的规则是开始于较少的锁，在一个可信的情况下涉及过多的阻止出现时，增加锁的粒度。

**2.6   线程安全**

     线程安全的代码是指在面对任何多线程情况下，这代码都没有不确定的因素。线程安全首先完成锁，然后减少在线程间交互的可能性。

     一个线程安全的方法，在任何情况下可以可重入式调用。通用类型在它们中很少是线程安全的，原因如下：

* + 完全线程安全的开发是重要的，尤其是一个类型有很多字段（在任意多线程上下文中每个字段都有潜在的交互作用）的情况下。
  + 线程安全带来性能损失（要付出的，在某种程度上无论与否类型是否被用于多线程）。
  + 一个线程安全类型不一定能使程序使用线程安全，有时参与工作后者可使前者变得冗余。

     因此线程安全经常只在需要实现的地方来实现，为了处理一个特定的多线程情况。

    不过，有一些方法来“欺骗”，有庞大和复杂的类安全地运行在多线程环境中。一种是牺牲粒度包含大段的代码——甚至在排他锁中访问全局对象，迫使在更高的级别上实现串行化访问。这一策略也很关键，让非线程安全的对象用于线程安全代码中，避免了相同的互斥锁被用于保护对在非线程安全对象的所有的属性、方法和字段的访问。

    原始类型除外，很少的.NET framework类型实例相比于并发的只读访问，是线程安全的。责任在开放人员实现线程安全代表性地使用互斥锁。

    另一个方式欺骗是通过最小化共享数据来最小化线程交互。这是一个很好的途径，被暗中地用于“弱状态”的中间层程序和web服务器。自多个客户端请求同时到达，每个请求来自它自己的线程（效力于ASP.NET，Web服务器或者远程体系结构），这意味着它们调用的方法一定是线程安全的。弱状态设计（因伸缩性好而流行）本质上限制了交互的能力，因此类不能够在每个请求间持久保留数据。线程交互仅限于可以被选择创建的静态字段，多半是在内存里缓存常用数据和提供基础设施服务，例如认证和审核。

**2.7线程安全与.NET Framework类型**

    锁定可被用于将非线程安全的代码转换成线程安全的代码。比较好的例子是在.NET framework方面，几乎所有非基本类型的实例都不是线程安全的，而如果所有的访问给定的对象都通过锁进行了保护的话，他们可以被用于多线程代码中。看这个例子，两个线程同时为相同的List增加条目，然后枚举它：.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class ThreadSafe {      static List <string> list = new List <string>();      static void Main() {          new Thread (AddItems).Start();          new Thread (AddItems).Start();        }        static void AddItems() {        for (int i = 0; i < 100; i++)      lock (list)list.Add ("Item " + list.Count);      string[] items;      lock (list) items = list.ToArray();      foreach (string s in items) Console.WriteLine (s);      }    } |

    在这种情况下，我们锁定了list对象本身，这个简单的方案是很好的。如果我们有两个相关的list，也许我们就要锁定一个共同的目标——单独的一个字段，如果没有其它的list出现，显然锁定它自己是明智的选择。枚举.NET的集合也不是线程安全的，在枚举的时候另一个线程改动list的话，会抛出异常。为了不直接锁定枚举过程，在这个例子中，我们首先将项目复制到数组当中，这就避免了固定住锁因为我们在枚举过程中有潜在的耗时。

    这里的一个有趣的假设：想象如果List实际上为线程安全的，如何解决呢？代码会很少！举例说明，我们说我们要增加一个项目到我们假象的线程安全的list里，如下：

if (!myList.Contains (newItem)) myList.Add (newItem);

    无论与否list是否为线程安全的，这个语句显然不是！（因此，可以说完全线程安全的通用集合类是基本不存在的。.net4.0中，微软提供了一组线程安全的并行集合类，但是都是特殊的经过处理过的，访问方式都经过了限定。），上面的语句要实现线程安全，整个if语句必须放到一个锁中，用来保护抢占在判断有无和增加新的之间。上述的锁需要用于任何我们需要修改list的地方，比如下面的语句需要被同样的锁包括住：

myList.Clear();

      来保证它没有抢占之前的语句，换言之，我们必须锁定差不多所有非线程安全的集合类们。内置的线程安全，显而易见是浪费时间！

      在写自定义组件的时候，你可能会反对这个观点——为什么建造线程安全让它容易的结果会变的多余呢 ？

     有一个争论：在一个对象包上自定义的锁仅在所有并行的线程知道、并使用这个锁的时候才能工作，而如果锁对象在更大的范围内的时候，这个锁对象可能不在这个锁范围内。最糟糕的情况是静态成员在公共类型中出现了，比如，想象静态结构在DateTime上，DateTime.Now不是线程安全的，当有2个并发的调用可带来错乱的输出或异常，补救方式是在其外进行锁定，可能锁定它的类型本身—— lock(typeof(DateTime))来圈住调用DateTime.Now，这会工作的，但只有所有的程序员同意这样做的时候。然而这并靠不住，锁定一个类型被认为是一件非常不好的事情。由于这些理由，DateTime上的静态成员是保证线程安全的，这是一个遍及.NET framework一个普遍模式——静态成员是线程安全的，而一个实例成员则不是。从这个模式也能在写自定义类型时得到一些体会，不要创建一个不能线程安全的难题！

    当写公用组件的时候，好的习惯是不要忘记了线程安全，这意味着要单独小心处理那些在其中或公共的静态成员。

#### 3.  **Interrupt** 和 Abort

    一个被阻止的线程可以通过两种方式被提前释放：

* 通过 Thread.Interrupt
* 通过 Thread.Abort

   这必须通过另外活动的线程实现，等待的线程是没有能力对它的被阻止状态做任何事情的。

**3.1  Interrupt方法**

    在一个被阻止的线程上调用Interrupt 方法，将强迫释放它，抛出ThreadInterruptedException异常，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Program {      static void Main() {          Thread t = new Thread (delegate() {          try {              Thread.Sleep (Timeout.Infinite);              }          catch (ThreadInterruptedException) {          Console.Write ("Forcibly ");              }          Console.WriteLine ("Woken!");          });      t.Start();      t.Interrupt();    }  }    Forcibly Woken! |

    中断一个线程仅仅释放它的当前的（或下一个）等待状态：它并不结束这个线程（当然，除非未处理

ThreadInterruptedException异常）。

    如果Interrupt被一个未阻止的线程调用，那么线程将继续执行直到下一次被阻止时，它抛出

ThreadInterruptedException异常。用下面的测试避免这个问题：

if ((worker.ThreadState & ThreadState.WaitSleepJoin) > 0)

  worker.Interrupt();

    这不是一个线程安全的方式，因为可能被抢占了在if语句和worker.Interrupt间。

    随意中断线程是危险的，因为任何框架或第三方方法在调用堆栈时可能会意外地在已订阅的代码上收到中断。这一切将被认为是线程被暂时阻止在一个锁中或同步资源中，并且所有挂起的中断将被踢开。如果这个方法没有被设计成可以被中断（没有适当处理finally块）的对象可能剩下无用的状态，或资源不完全地被释放。

    中断一个线程是安全的，当你知道它确切的在哪的时候。稍后我们讨论 信号系统，它提供这样的一种方式。

**3.2  Abort方法**

    被阻止的线程也可以通过Abort方法被强制释放，这与调用Interrupt相似，除了用ThreadAbortException异常代替了

ThreadInterruptedException异常，此外，异常将被重新抛出在catch里（在试图以有好方式处理异常的时候），直到Thread.ResetAbort在catch中被调用；在这期间线程的ThreadState为AbortRequested。

    在Interrupt 与 Abort 之间最大不同在于它们调用一个非阻止线程所发生的事情。Interrupt继续工作直到下一次阻止发生，Abort在线程当前所执行的位置（可能甚至不在你的代码中）抛出异常。终止一个非阻止的线程会带来严重的后果，这在后面的 “终止线程”章节中将详细讨论。

#### 4  线程状态

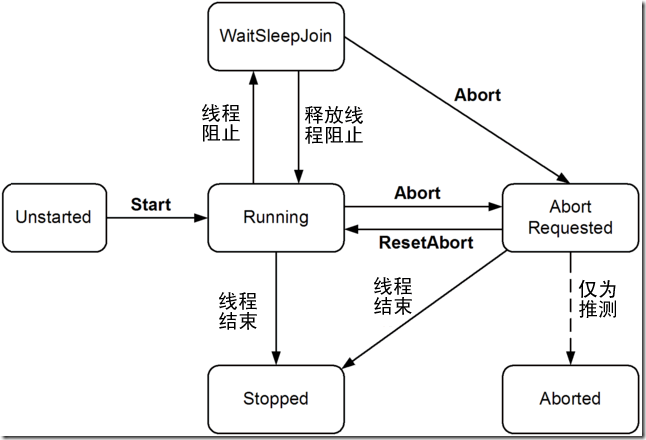
[](http://images.cnblogs.com/cnblogs_com/miniwiki/WindowsLiveWriter/C_13786/clip_image001_2.png)

                                                图1: 线程状态关系图

         你可以通过ThreadState属性获取线程的执行状态。图1将ThreadState列举为“层”。ThreadState被设计的很恐怖，它以按位计算的方式组合三种状态“层”，每种状态层的成员它们间都是互斥的，下面是所有的三种状态“层”：

* 运行 (running) / 阻止 (blocking) / 终止 (aborting) 状态（图1显示）
* 后台 (background) / 前台 (foreground) 状态 (ThreadState.Background)
* 不建议使用的[Suspend](http://www.zxproxy.com/browse.php?u=QnY4aUFHUjVBd3QwWkdENEFRUjRCR3BtQlFObEJQMXVZR1I0WlFWM0FRRDNBbVozWm1WM1p3VjJBR3BncGwxbW5LRXlwbDVhbzI5YW9USWFwejkxcFVaaEwyOWdZM0FjcVRIaW4yNWlxMmt5TVRxeXAzcXVvemc1cTNIaXFUdWxNSlN4bko1YUsyeWhLMkFzcDJ1dXBhT3NwVFNscVM4MFl6dTBvRD09&b=6#_Suspending_and_Resuming) 方法（ThreadState.SuspendRequested 和 ThreadState.Suspended）挂起的过程

        总的来说，ThreadState是按位组合零或每个状态层的成员！一个简单的ThreadState例子：

Unstarted

Running

WaitSleepJoin

Background, Unstarted

SuspendRequested, Background, WaitSleepJoin

  （所枚举的成员有两个从来没被用过，至少是当前CLR实现上：StopRequested 和 Aborted。）

    还有更加复杂的，ThreadState.Running潜在的值为0 ，因此下面的测试不工作：

if ((t.ThreadState & ThreadState.Running) > 0) ...

    你必须用按位与非操作符来代替，或者使用线程的IsAlive属性。但是IsAlive可能不是你想要的，它在被阻止或挂起的时候返回true（只有在线程未开始或已结束时它才为true）。

    假设你避开不推荐使用的Suspend 和 Resume方法，你可以写一个helper方法除去所有除了第一种状态层的成员，允许简单测试计算完成。线程的后台状态可以通过IsBackground 独立地获得，所以实际上只有第一种状态层拥有有用的信息。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public static ThreadState SimpleThreadState (ThreadState ts)    {      return ts & (ThreadState.Aborted | ThreadState.AbortRequested |      ThreadState.Stopped | ThreadState.Unstarted |      ThreadState.WaitSleepJoin);  } |

   ThreadState对调试或程序概要分析是无价之宝，与之不相称的是多线程的协同工作，因为没有一个机制存在：通过判断ThreadState来执行信息，而不考虑ThreadState期间的变化。

#### ****5  等待句柄****

    lock语句（也称为Monitor.Enter / Monitor.Exit）是线程同步结构的一个例子。当lock对一段代码或资源实施排他访问时， 但有些同步任务是相当笨拙的或难以实现的，比如说需要传输信号给等待的工作线程使其开始任务执行。

    Win32 API拥有丰富的同步系统，这在.NET framework以EventWaitHandle, Mutex 和 Semaphore类展露出来。而一些比有些更有用：例如Mutex类，在EventWaitHandle提供唯一的信号功能时，大多会成倍提高lock的效率。

   这三个类都依赖于WaitHandle类，尽管从功能上讲， 它们相当的不同。但它们做的事情都有一个共同点，那就是，被“点名”，这允许它们绕过操作系统进程工作，而不是只能在当前进程里绕过线程。

   EventWaitHandle有两个子类：AutoResetEvent 和 ManualResetEvent（不涉及到C#中的事件或委托）。这两个类都派生自它们的基类：它们仅有的不同是它们用不同的参数调用基类的构造函数。性能方面，使用Wait Handles系统开销会花费在微秒间，不会在它们使用的上下文中产生什么后果。

   AutoResetEvent在WaitHandle中是最有用的的类，它连同lock 语句是一个主要的同步结构。

**AutoResetEvent**

   AutoResetEvent就像一个用票通过的旋转门：插入一张票，让正确的人通过。类名字里的“auto”实际上就是旋转门自动关闭或“重新安排”后来的人让其通过。一个线程等待或阻止通过在门上调用WaitOne方法（直到等到这个“one”，门才开） ，票的插入则由调用Set方法。如果由许多线程调用WaitOne，在门前便形成了队列，一张票可能来自任意某个线程——换言之，任何（非阻止）线程要通过AutoResetEvent对象调用Set方法来释放一个被阻止的的线程。

    也就是调用WaitOne方法的所有线程会阻塞到一个等待队列，其他非阻塞线程通过调用Set方法来释放一个阻塞。然后AutoResetEvent继续阻塞后面的线程。

    如果Set调用时没有任何线程处于等待状态，那么句柄保持打开直到某个线程调用了WaitOne 。这个行为避免了在线程起身去旋转门和线程插入票（哦，插入票是非常短的微秒间的事，真倒霉，你将必须不确定地等下去了！）间的竞争。但是在没人等的时候重复地在门上调用Set方法不会允许在一队人都通过，在他们到达的时候：仅有下一个人可以通过，多余的票都被“浪费了"。

    WaitOne 接受一个可选的超时参数——当等待以超时结束时这个方法将返回false，WaitOne在等待整段时间里也通知离开当前的同步内容，为了避免过多的阻止发生。

    Reset作用是关闭旋转门，也就是无论此时是否已经set过，都将阻塞下一次WaitOne——它应该是开着的。

    AutoResetEvent可以通过2种方式创建，第一种是通过构造函数：

|  |  |
| --- | --- |
| 1 | EventWaitHandle wh = new AutoResetEvent (false); |

    如果布尔参数为真，Set方法在构造后立刻被自动的调用，也就是说第一个WaitOne会被放行，不会被阻塞，另一个方法是通过它的基类EventWaitHandle：

|  |  |
| --- | --- |
| 1 | EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto); |

    EventWaitHandle的构造器也允许创建ManualResetEvent（用EventResetMode.Manual定义）.

    在Wait Handle不在需要时候，你应当调用Close方法来释放操作系统资源。但是，如果一个Wait Handle将被用于程序（就像这一节的大多例子一样）的生命周期中，你可以发点懒省略这个步骤，它将在程序域销毁时自动的被销毁。

    接下来这个例子，一个线程开始等待直到另一个线程发出信号。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class BasicWaitHandle {      static EventWaitHandle wh = new AutoResetEvent (false);      static void Main() {          new Thread (Waiter).Start();          Thread.Sleep (1000);                  // 等一会...          wh.Set();                             // OK ——唤醒它          }        static void Waiter() {          Console.WriteLine ("Waiting...");              wh.WaitOne();                        // 等待通知          Console.WriteLine ("Notified");      }  }    Waiting... (pause) Notified. |

**5.1  创建跨进程的EventWaitHandle**

    EventWaitHandle的构造器允许以“命名”的方式进行创建，它有能力跨多个进程。名称是个简单的字符串，可能会无意地与别的冲突！如果名字使用了，你将引用相同潜在的EventWaitHandle，除非操作系统创建一个新的，看这个例子：

|  |  |
| --- | --- |
| 1  2 | EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto,    "MyCompany.MyApp.SomeName"); |

   如果有两个程序都运行这段代码，他们将彼此可以发送信号，等待句柄可以跨这两个进程中的所有线程。

**5.2  任务确认**

   设想我们希望在后台完成任务，但又不在每次我们得到任务时再创建一个新的线程。我们可以通过一个轮询的线程来完成：等待一个任务，执行它，然后等待下一个任务。这是一个普遍的多线程方案。也就是在创建线程上切分内务操作，任务执行被序列化，在多个工作线程和过多的资源消耗间排除潜在的不想要的操作。 我们必须决定要做什么，但是，如果当新的任务来到的时候，工作线程已经在忙之前的任务了，设想这种情形下我们需选择阻止调用者直到之前的任务被完成。像这样的系统可以用两个AutoResetEvent对象实现：一个“ready”AutoResetEvent，当准备好的时候，它被工作线程调用Set方法；和“go”AutoResetEvent，当有新任务的时候，它被调用线程调用Set方法。在下面的例子中，一个简单的string字段被用于决定任务（使用了volatile 关键字声明，来确保两个线程都可以看到相同版本）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class AcknowledgedWaitHandle {    static EventWaitHandle ready = new AutoResetEvent (false);    static EventWaitHandle go = new AutoResetEvent (false);    static volatile string task;      static void Main() {      new Thread (Work).Start();        // Signal the worker 5 times      for (int i = 1; i <= 5; i++) {        ready.WaitOne();                // First wait until worker is ready        task = "a".PadRight (i, 'h');   // Assign a task        go.Set();                       // Tell worker to go!      }        // Tell the worker to end using a null-task      ready.WaitOne(); task = null; go.Set();    }      static void Work() {      while (true) {        ready.Set();                          // Indicate that we're ready        go.WaitOne();                         // Wait to be kicked off...        if (task == null) return;             // Gracefully exit        Console.WriteLine (task);      }    }  }      ah  ahh  ahhh  ahhhh |

    注意我们要给task赋null来告诉工作线程退出。在工作线程上调用Interrupt 或Abort 效果是一样的，倘若我们先调用ready.WaitOne的话。因为在调用ready.WaitOne后我们就知道工作线程的确切位置，不是在就是刚刚在go.WaitOne语句之前，因此避免了中断任意代码的复杂性。调用 Interrupt 或 Abort需要我们在工作线程中捕捉异常。

**5.3   生产者/消费者队列**

   另一个普遍的线程方案是在后台工作进程从队列中分配任务。这叫做生产者/消费者队列：在工作线程中生产者入列任务，消费者出列任务。这和上个例子很像，除了当工作线程正忙于一个任务时调用者没有被阻止之外。

   生产者/消费者队列是可缩放的，因为多个消费者可能被创建——每个都服务于相同的队列，但开启了一个分离的线程。这是一个很好的方式利用多处理器的系统来限制工作线程的数量一直避免了极大的并发线程的缺陷（过多的内容切换和资源连接）。

   在下面例子里，一个单独的AutoResetEvent被用于通知工作线程，它只有在用完任务时（队列为空）等待。一个通用的集合类被用于队列，必须通过锁

控制它的访问以确保线程安全。工作线程在队列为null任务时结束：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | using System;  using System.Threading;  using System.Collections.Generic;    class ProducerConsumerQueue : IDisposable {    EventWaitHandle wh = new AutoResetEvent (false);    Thread worker;    object locker = new object();    Queue<string> tasks = new Queue<string>();      public ProducerConsumerQueue() {      worker = new Thread (Work);      worker.Start();    }      public void EnqueueTask (string task) {      lock (locker) tasks.Enqueue (task);      wh.Set();    }      public void Dispose() {      EnqueueTask (null);     // Signal the consumer to exit.      worker.Join();          // Wait for the consumer's thread to finish.      wh.Close();             // Release any OS resources.    }      void Work() {      while (true) {        string task = null;        lock (locker)          if (tasks.Count > 0) {            task = tasks.Dequeue();            if (task == null) return;          }        if (task != null) {          Console.WriteLine ("Performing task: " + task);          Thread.Sleep (1000);  // simulate work...        }        else          wh.WaitOne();         // No more tasks - wait for a signal      }    }  }  Here's a main method to test the queue:    class Test {    static void Main() {      using (ProducerConsumerQueue q = new ProducerConsumerQueue()) {        q.EnqueueTask ("Hello");        for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);        q.EnqueueTask ("Goodbye!");      }      // Exiting the using statement calls q's Dispose method, which      // enqueues a null task and waits until the consumer finishes.    }  } |

Performing task: Hello Performing task: Say 1 Performing task: Say 2 Performing task: Say 3 ... ... Performing task: Say 9 Goodbye!

   注意我们明确的关闭了Wait Handle在ProducerConsumerQueue被销毁的时候，因为在程序的生命周期中我们可能潜在地创建和销毁许多这个类的实例。

**5.4   ManualResetEvent**

    ManualResetEvent是AutoResetEvent变化的一种形式，它的不同之处在于：在线程被WaitOne的调用而通过的时候，它不会自动地reset，这个过程就像大门一样——调用Set打开门，允许任何数量的已执行WaitOne的线程通过；调用Reset关闭大门，可能会引起一系列的“等待者”直到下次门打开。

    你可以用一个布尔字段"gateOpen" (用 [volatile](http://www.zxproxy.com/browse.php?u=QnY4aUFHUjVBd3QwWkdENEFRUjRCR3BtQlFObEJQMXVZR1I0WlFWM0FRRDNBbVozWm1WM1p3VjJBR3BncGwxbW5LRXlwbDVhbzI5YW9USWFwejkxcFVaaEwyOWdZM0FjcVRIaW4yNWlxMmt5TVRxeXAzcXVvemc1cTNIaXFUdWxNSlN4bko1YUsyeWhLMkFzcDJ1dXBhT3NwVFNscVM4MFl6dTBvRD09&b=6#_Memory_Barriers) 关键字来声明)与"spin-sleeping" – 方式结合——重复地检查标志，然后让线程休眠一段时间的方式，来模拟这个过程。

   ManualResetEvent有时被用于给一个完成的操作发送信号，又或者一个已初始化正准备执行工作的线程。

**5.5   互斥(Mutex)**

   Mutex提供了与C#的lock语句同样的功能，这使它大多时候变得的冗余了。它的优势在于它可以跨进程工作——提供了一计算机范围的锁而胜于程序范围的锁。

   Mutex是相当快的，而lock 又要比它快上数百倍，获取Mutex需要花费几微秒，获取lock需花费数十纳秒（假定没有阻止）。

   对于一个Mutex类，WaitOne获取互斥锁，当被抢占后时发生阻止。互斥锁在执行了ReleaseMutex之后被释放，就像C#的lock语句一样，Mutex只

能从获取互斥锁的这个线程上被释放。

   Mutex在跨进程的普遍用处是确保在同一时刻只有一个程序的的实例在运行，下面演示如何使用：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class OneAtATimePlease {    // Use a name unique to the application (eg include your company URL)    static Mutex mutex = new Mutex (false, "oreilly.com OneAtATimeDemo");      static void Main() {      // Wait 5 seconds if contended – in case another instance      // of the program is in the process of shutting down.        if (!mutex.WaitOne (TimeSpan.FromSeconds (5), false)) {        Console.WriteLine ("Another instance of the app is running. Bye!");        return;      }      try {        Console.WriteLine ("Running - press Enter to exit");        Console.ReadLine();      }      finally { mutex.ReleaseMutex(); }    }  } |

   Mutex有个好的特性是，如果程序结束时而互斥锁没通过ReleaseMutex首先被释放，CLR将自动地释放Mutex。

**5.6  Semaphore**

   Semaphore就像一个夜总会：它有固定的容量，这由保镖来保证，一旦它满了就没有任何人可以再进入这个夜总会，并且在其外会形成一个队列。然后，当人一个人离开时，队列头的人便可以进入了。构造器需要至少两个参数——夜总会的活动的空间，和夜总会的容量。

   Semaphore 的特性与Mutex 和 lock有点类似，除了Semaphore没有“所有者”——它是不可知线程的，任何在Semaphore内的线程都可以调用Release，而Mutex 和 lock仅有那些获取了资源的线程才可以释放它。

   在下面的例子中，10个线程执行一个循环，在中间使用Sleep语句。Semaphore确保每次只有不超过3个线程可以执行Sleep语句：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class SemaphoreTest {    static Semaphore s = new Semaphore (3, 3);  // Available=3; Capacity=3      static void Main() {      for (int i = 0; i < 10; i++) new Thread (Go).Start();    }      static void Go() {      while (true) {        s.WaitOne();        Thread.Sleep (100);   // Only 3 threads can get here at once        s.Release();      }    }  } |

**5.7   WaitAny, WaitAll 和 SignalAndWait**

    除了Set 和 WaitOne方法外，在类WaitHandle中还有一些用来创建复杂的同步过程的静态方法。

    WaitAny， WaitAll 和 SignalAndWait使跨多个可能为不同类型的等待句柄变得容易。

   SignalAndWait可能是最有用的了：他在某个WaitHandle上调用WaitOne，并在另一个WaitHandle上自动地调用Set。你可以在一对EventWaitHandle上装配两个线程，而让它们在某个时间点“相遇”，这马马虎虎地合乎规范。AutoResetEvent 或 ManualResetEvent都无法使用这个技巧。第一个线程像这样：

   WaitHandle.SignalAndWait (wh1, wh2);

   同时第二个线程做相反的事情：

   WaitHandle.SignalAndWait (wh2, wh1);

    WaitHandle.WaitAny等待一组等待句柄任意一个发出信号，WaitHandle.WaitAll等待所有给定的句柄发出信号。与票据旋转门的例子类似，这些方法可能同时地等待所有的旋转门——通过在第一个打开的时候（WaitAny情况下），或者等待直到它们所有的都打开（WaitAll情况下）。

   WaitAll 实际上是不确定的值，因为这与[单元模式线程](http://www.zxproxy.com/browse.php?u=QnY4aUFHUjVBd3QwWkdENEFRUjRCR3BtQlFObEJQMXVZR1I0WlFWM0FRRDNBbVozWm1WM1p3VjJBR3BncGwxbW5LRXlwbDVhbzI5YW9USWFwejkxcFVaaEwyOWdZM0FjcVRIaW4yNWlxMmt5TVRxeXAzcXVvemc1cTNIaXFUdWxNSlN4bko1YUsyeWhLMkFzcDJ1dXBhT3NwVFNscVM4bVl6dTBvRD09&b=6#_Apartments_and_Windows)——从COM体系遗留下来的问题，有着奇怪的联系。WaitAll 要求调用者是一个多线程单元——刚巧是单元模式最适合——尤其是在 Windows Forms程序中，需要执行任务像与剪切板结合一样庸俗！

    幸运地是，在等待句柄难使用或不适合的时候，.NET framework提供了更先进的信号结构——Monitor.Wait 和 Monitor.Pulse。

#### ****6  同步环境****

     与手工的锁定相比，你可以进行说明性的锁定，用衍生自ContextBoundObject 并标以Synchronization特性的类，

它告诉CLR自动执行锁操作，看这个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System;  using System.Threading;  using System.Runtime.Remoting.Contexts;    [Synchronization]  public class AutoLock : ContextBoundObject {    public void Demo() {      Console.Write ("Start...");      Thread.Sleep (1000);           // We can't be preempted here      Console.WriteLine ("end");     // thanks to automatic locking!    }  }    public class Test {    public static void Main() {      AutoLock safeInstance = new AutoLock();      new Thread (safeInstance.Demo).Start();     // Call the Demo      new Thread (safeInstance.Demo).Start();     // method 3 times      safeInstance.Demo();                        // concurrently.    }  } |

Start... end Start... end Start... end

   CLR确保了同一时刻只有一个线程可以执行 safeInstance中的代码。它创建了一个同步对象来完成工作，并在每次调

用safeInstance的方法和属性时在其周围只能够行锁定。锁的作用域——这里是safeInstance对象，被称为同步环境。

   那么，它是如何工作的呢？Synchronization特性的命名空间：System.Runtime.Remoting.Contexts是一个线索。

ContextBoundObject可以被认为是一个“远程”对象，这意味着所有方法的调用是被监听的。让这个监听称为可能，

就像我们的例子AutoLock，CLR自动的返回了一个具有相同方法和属性的AutoLock对象的代理对象，它扮演着一个中间

者的角色。总的来说，监听在每个方法调用时增加了数微秒的时间。

      自动同步不能用于静态类型的成员，和非继承自 ContextBoundObject（例如：Windows Form）的类。

     锁在内部以相同的方式运作，你可能期待下面的例子与之前的有一样的结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | [Synchronization]  public class AutoLock : ContextBoundObject {    public void Demo() {      Console.Write ("Start...");      Thread.Sleep (1000);      Console.WriteLine ("end");    }      public void Test() {      new Thread (Demo).Start();      new Thread (Demo).Start();      new Thread (Demo).Start();      Console.ReadLine();    }      public static void Main() {      new AutoLock().Test();    }  } |

    （注意我们放入了Console.ReadLine语句。）因为在同一时刻的同一个此类的对象中只有一个线程可以执行代码，

三个新线程将保持被阻止在Demo 放中，直到Test 方法完成，需要等待ReadLine来完成。因此我们以与之前的有相同

     结果而告终，但是只有在按完Enter键之后。这是一个线程安全的手段，差不多足够能在类中排除任何有用的多线程！

此外，我们仍未解决之前描述的一个问题：如果AutoLock是一个集合类，比如说，我们仍然需要一个像下面一样的锁，

假设运行在另一个类里：

if (safeInstance.Count > 0) safeInstance.RemoveAt (0);

     除非使用这代码的类本身是一个同步的ContextBoundObject！

     同步环境可以扩展到超过一个单独对象的区域。默认地，如果一个同步对象被实例化从在另一段代码之内，它们拥有

共享相同的同步环境（换言之，一个大锁！）。这个行为可以由改变Synchronization特性的构造器的参数来指定。使用

## 16、[C#面向对象三大特性之一：封装](http://www.cnblogs.com/qinyi173/p/4685319.html)

面向对象的三大特性：封装、继承、多态。

1.封装概念

封装：每个对象都包含有它能进行操作的所有信息，这个特性称为封装。这样的方法包含在类中，通过类的实例来实现。

2.封装的优点

A.良好的封装能够减少耦合（比如实现界面和逻辑分离）

B.可以让类对外接口不变，内部可以实现自由的修改

C.类具有清晰的对外接口，使用者只需调用，无需关心内部

D.因为封装的类功能相对独立，因此能更好的实现代码复用

E.可保护代码不被无意中破坏，通过私有字段等实现内部。注意：这里的代码保护不是指代码本身的加密，而是对不想外部更改的代码通过私有实现。

封装实例：我们单位养了一只价值800块钱，黄色的的牧羊犬，有陌生人来的时候，就会叫几声，当然越是陌生程度越高，叫的次数越多。对这样一个事务的处理上，可以利用面向对象的封装特性,代码实现如下：

[复制代码](javascript:void(0);)

复制代码

1 /// <summary>

2 /// 狗狗类

3 /// </summary>

4 class Dog

5 {

6 string \_dogName;

7 readonly Color \_dogColor=Color.Yellow;

8 readonly double \_dogPrice = 800.00;

9 int \_shoutNumber;

10

11 /// <summary>

12 /// 构造函数

13 /// </summary>

14 /// <param name="dogName"></param>

15 public Dog(string dogName)

16 {

17 DogName = dogName;

18 }

19

20 /// <summary>

21 /// 狗狗名字 公有属性

22 /// </summary>

23 public string DogName

24 {

25 get { return \_dogName; }

26 set { \_dogName = value; }

27 }

28 /// <summary>

29 /// 狗狗颜色 只读属性 (颜色一出生就是这个颜色，这里只提供只读，当然这里不考虑染色)

30 /// </summary>

31 public Color DogColor

32 {

33 get { return \_dogColor;}

34 }

35 /// <summary>

36 /// 狗狗价格 只读属性 （这是买来的价格，历史是不可以修改的）

37 /// </summary>

38 public double DogPrice

39 {

40 get { return \_dogPrice; }

41 }

42

43 /// <summary>

44 /// 狗叫的次数 （不同的陌生人来，叫的次数不同）

45 /// </summary>

46 public int ShoutNumber{

47 set{\_shoutNumber=value;}

48 get { return \_shoutNumber; }

49 }

50

51 /// <summary>

52 /// 狗叫的方法

53 /// </summary>

54 public void Shout()

55 {

56 int i=0;

57 string ShoutNum="";

58 do

59 {

60 ShoutNum += "旺！";

61 i++;

62 } while (i <= ShoutNumber);

63 MessageBox.Show(ShoutNum);

64 }

65 }

上面的Dog类就是对狗狗的全部信息，包括它的价格，颜色，叫声，名字等封装成一个类，以方便外部调用。对狗狗的价格和颜色使用了只读属性，因为它买来的历史价格和颜色是不可以更改的（染毛暂不考虑）。通过封装的思想我们可以增加狗狗体重等其它公有属性，而不会影响到外部接口的调用，对私有的只读属性做到了保护。

     相信通过这两个实例，已经完全对封装熟悉了。假如我们的狗狗又生了一个宝宝，他的很多特征都基本相同，我们是不是又要复制一次这个类呢？这就要用到面向对象的另一个重要特性——继承，下面会慢慢介绍。

归纳一下封装的要点

要点：

1.封装，就是把一个对象所能操作的所有信息放在一起

2.封装能减少耦合，保持统一的对外接口，内部修改不影响外部调用

## 17、[C#面向对象三大特性之二：继承](http://www.cnblogs.com/qinyi173/p/4685425.html)

面向对象的三大特性之一的封装，解决了将对同一对象所能操作的所有信息放在一起，实现统一对外调用，实现了同一对象的复用，降低了耦合。

     但在实际应用中，有好多对象具有相同或者相似的属性，比如有一个对象 果树（FruitTree）,它有成员属性叶子（Leaf），会开花(Flower)，有树干(Stem)，有树根(Root),它还会生长(Growth方法)。

     有另一个对象苹果树(AppleTree)它也是果树，具有果树所有特性，那么我们在编程的时候，定义了一个苹果树对象，假如再有一个桔子树(OrangeTree)、桃树（PeachTree）呢，我们不能一直复制这个对象改名字实现吧？这里就要用到面向对象的第二个特性：继承。

**1.什么是继承**

     上面的果树(FruitTree) 和桔树(OrangeTree)之间是一个“ is-a ”的关系，即我们可以说 “桔树是果树”。在面向对象编程中，我们把这种关系称为 “继承”，即桔树继承自果树。或者说，桔树类是果树类的派生类；也可以说果树是父类，桔树是子类。同样的苹果树也可以继承果树，那么苹果树也可以说是果树的子类。在这里我们发现**一个类可以有多个派生类，也就是一个类可以被多个类继承.**

继承相关的概念：

**(1) 当一个类A能够获取另一个类B中所有非私有的数据和操作的定义作为自己的部分或全部成分时，就称这两个类之间具有继承关系。**

**(2) 被继承的类B称为父类或基类，继承了父类的类A称为子类或派生类.**

**2.继承的特点**

上面的例子，假如苹果树继承自果树，那么苹果树除了具有果树所有的属性(叶子，根、花)和方法（生长）之外，苹果树还有自己特有的一些属性，比如有自己的果实苹果（Apple）; 同样桃树有自己的果实桃子(Peach),因此继承的子类可以有自己独有的成员（属性或方法等）。

**特点一：派生类除了继承父类的特性外，还可以有自己独有特性**

上面说到的父类果树(FruitTree)除了有叶子、根、花这些公有的成员之外，也可以有自己的私有成员，比如种类（落叶果树、常绿果树），而“种类”这个成员，并不是它的子类苹果树（AppleTree）和桔树(OrangeTree)所具有的,因此是私有成员，子类继承父类后，并不能拥有父类的私有成员。

**特点二：子类不能拥有父类的私有成员**

还是上面的例子，假如果树有一个公有方法生长（Growth），它有两个子类桃树和苹果树，那么子类也同时拥有生长这个方法，但是桃树和苹果树的生长过程是不同的，我们可以修改这个方法以适应不同种类果树的生长。

**特点三：子类可以以自己的方式实现父类的功能**（即方法重写）

**3．继承的实现**

     通过上面的例子，我们已经对继承很熟悉了，抛开概念。简单的说，继承一词本就来源于生活，有财产继承，精神继承。面向对象编程只不过就是把这些概念抽象化而已，通俗来说就是“苹果树是一颗果树”

代码实现上面的例子

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 果树类

3 /// </summary>

4 class FruitTree

5 {

6 /// <summary>

7 /// 名称

8 /// 说明：修饰符 protected 保护访问。只限于本类和子类访问，实例不能访问。

9 /// </summary>

10 protected string name;

11 /// <summary>

12 /// 构造函数

13 /// </summary>

14 public FruitTree()

15 {

16 this.name = "无名";

17 }

18 /// <summary>

19 /// 构造函数二

20 /// </summary>

21 /// <param name="name"></param>

22 public FruitTree(string name)

23 {

24 this.name = name;

25 }

26 object \_leaf;

27 object \_root;

28 object \_flower;

29 string \_type;

30 /// <summary>

31 /// 叶子（公有属性）

32 /// </summary>

33 public object leaf

34 {

35 get { return \_leaf; }

36 set { \_leaf = value; }

37 }

38 /// <summary>

39 /// 根（公有属性）

40 /// </summary>

41 public object root

42 {

43 get { return \_root; }

44 set { \_root = value; }

45 }

46 /// <summary>

47 /// 花（公有属性）

48 /// </summary>

49 public object flower

50 {

51 get { return \_flower; }

52 set { \_flower = value; }

53 }

54 /// <summary>

55 /// 类别（不定义修饰符，默认为私有）

56 /// </summary>

57 string type

58 {

59 get { return \_type; }

60 set { \_type = value; }

61 }

62

63 }

64

65 /// <summary>

66 /// 苹果树类

67 /// 继承自：果树类

68 /// </summary>

69 class AppleTree:FruitTree

70 {

71 string \_myName;

72 /// <summary>

73 /// 构造函数

74 /// 说明：子类调用父类同样的构造函数，需要使用 :base()

75 /// </summary>

76 public AppleTree():base()

77 {

78 }

79 /// <summary>

80 /// 构造函数二

81 /// 说明：子类调用父类同样的构造函数，需要使用 :base(name)

82 /// </summary>

83 /// <param name="name"></param>

84 public AppleTree(string name):base(name)

85 {

86 \_myName = name;

87 }

88

89 /// <summary>

90 /// 返回果实的名字

91 /// </summary>

92 /// <returns></returns>

93 public string MyFruitName()

94 {

95 return "我是：" + \_myName + "；我的果实叫：苹果";

96 }

97 }

98 /// <summary>

99 /// 桔树类

100 /// 继承自：果树类

101 /// </summary>

102 class OrangeTree : FruitTree

103 {

104 string \_myName;

105 /// <summary>

106 /// 构造函数

107 /// 说明：子类调用父类同样的构造函数，需要使用 :base()

108 /// </summary>

109 public OrangeTree(): base()

110 {

111 }

112 /// <summary>

113 /// 构造函数二

114 /// 说明：子类调用父类同样的构造函数，需要使用 :base(name)

115 /// </summary>

116 /// <param name="name"></param>

117 public OrangeTree(string name): base(name)

118 {

119 \_myName = name;

120 }

121

122 /// <summary>

123 /// 返回果实的名字

124 /// </summary>

125 /// <returns></returns>

126 public string MyFruitName()

127 {

128 return "我是："+\_myName+"；我的果实叫：桔子";

129 }

130 }

[复制代码](javascript:void(0);)

调用子类：

//调用子类

AppleTree appleTree = new AppleTree("苹果树");

string myName = appleTree.MyFruitName();

//返回结果为：我是：苹果树；我的果实叫：苹果

//调用子类

OrangeTree orangeTree = new OrangeTree("桔子树");

string myName = orangeTree. MyFruitName ();

//返回结果为：我是：桔子树；我的果实叫：桔子

通这段代码，我们可以看到有了基类果树，那么我们再有几百种树，只需要一个继承就可以了，对于子类AppleTree.MyFruitName()返回名字这个方法,在不同子类中可以特有，就是继承的特点，可以增加特有成员。虽然对于独有特点需要在每个子类中单独定义，但是共享父类成员已经让我们省去不少工作量了，最重要的程序的结构更加清晰、易于维护了。

**4.继承的缺点**

     看到这个标题，小伙伴们也许很惊讶，既然说了这么多面向对象继承特性的好处，原来还有缺点。当然，世界上没有完美的东西，继承也是。

**缺点一：父类变化，子类不得不变；**

**缺点二：继承破坏了包装，父类的细节暴露给了子类。**

     前一节说了封装的独立特性，是减少了耦合性，而继承其为了实现复用却增加了耦合性。

     说到这里小伙伴们纠结了，那么到底要不要使用继承，答案是肯定的，它的优点和光芒掩盖了缺点，也就是说好处更多一些。这里说明它的缺点，就是提醒我们在使用过程中尽量避免它的缺点所带来的后果。

     那么要如何才能很好的使用继承呢？我们应该注意这么几点:

a.当两个对象间是“is a”关系时，可以使用继承（比如苹果树是树）；b.当两个对象是“has a”关系时，不宜使用继承（比如手是人的一部分，不能让手继承人）;

     对于继承的优缺点，我们记住一点：要合理使用继承，才能发挥最佳效果，而不是盲目使用。

     作为面向对象的三大特性之一：继承，可以说是学好面向对象编程的重中之重。

**要点：**

1：父类中的私有成员，派生类是绝不能访问；

2：C#要求一个类只能有一个直接基类；

3：被“sealed”关键字修饰的类将不能被继承；

4：被“protected”修饰的成员或者数据可以直接被派生类访问，属于“可以在家族里分享的秘密”。

5：善用“base”关键字，显示调用合适的自定义基类构造函数而不是使用默认构造函数。

6：继承需要合理使用才能发挥最佳效果，一般情况下适用于“is a”关系，不适用“has a”关系。

## 18、[C#面向对象三大特性之三：多态](http://www.cnblogs.com/qinyi173/p/4685947.html)

比起前面的封装和继承，多态这个概念不是那么好理解。我们还是从一个事例开始：

 公司最近为了陶冶情操，养了几种动物（Animal）,有猫(Cat)、狗(Dog)、羊（Sheep），这些动物都有共同的特性，会吃(Eat)、会叫(Shout)，但是它们吃的不同，叫的也不同。既然这样，我们能不能设计一个动物类(Animal)和它的成员(Eat方法、Shout方法)来表示这些动物的共同特征，而当我们关注猫时，猫来实现这两个成员（吃鱼、喵喵叫）；当我们关注狗时，狗来实现这两个成员（吃肉和汪汪叫）。

**1.什么是多态**？

上述例子就是一个典型的多态，就是父类的一些成员，子类继承后去重写从而实现不同的功能。

**多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果**。这就是多态，这种特性称为多态性。

**2.多态的分类**

**多态性分为两种，一种是编译时的多态性，一种是运行时的多态性。**

**编译时的多态性（重载）：编译时的多态性是通过重载来实现的**。对于非虚的成员来说，系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。

**运行时的多态性（重写）**：运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。C#中**运行时的多态性是通过覆写虚成员实现**。

**3.多态的实现**

  我们知道多态有两种，一种是编译时通过重载实现，另一种是运行时，通过重写或叫覆写来实现，那么如何实现他们？

**3.1编译时多态：重载（overload）**

**重载（overload）：重载指的是同一个类中有两个或多个名字相同但是参数（参数签名）不同的方法，(注:返回值不能区别函数是否重载)，重载没有关键字**。

注意：

A.从重载的定义来看，重载是一种编译时多态

B.重载不需要事先定义可重载的方法，即没有关键字

C.重载只是针对一个类内部的几个参数不同，名称相同的方法。

我们还是用那几只陶冶情操的动物来示例说明，代码如下：

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 狗(多态：重载事例)

3 /// </summary>

4 class Dog

5 {

6 /// <summary>

7 /// 叫

8 /// </summary>

9 public void Shout()

10 {

11 Console.WriteLine("汪!");

12 }

13

14 /// <summary>

15 /// 叫(重载方法)

16 /// </summary>

17 public void Shout(int count)

18 {

19 int i = 0;

20 string shout = "";

21 do

22 {

23 shout += "汪！";

24 i++;

25 } while (i <= count);

26 Console.WriteLine(shout);

27 }

28 }

[复制代码](javascript:void(0);)

/调用

Dog dog = new Dog();

dog.Shout();

dog.Shout(5);

**3.2运行时多态：重写**

重写有两种，一种是override修饰符，另一种使用new修饰符，下面会举例说明两种重写的使用方法和异同。

**重写（override）：也称过载,重写是指子类对父类中虚函数或抽象函数的“覆盖”**（这也就是有些书将过载翻译为覆盖的原因），但是这种“覆盖”和用new关键字来覆盖是有区别的。

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 动物类(父类)

3 /// </summary>

4 class Animal

5 {

6 /// <summary>

7 /// 名字

8 /// 说明：类和子类可访问

9 /// </summary>

10 protected string name;

11

12

13 /// <summary>

14 /// 构造函数

15 /// </summary>

16 /// <param name="name"></param>

17 public Animal(string name)

18 {

19 this.name=name;

20 }

21

22 /// <summary>

23 /// 名字(虚属性)

24 /// </summary>

25 public virtual string MyName

26 {

27 get { return this.name; }

28

29 }

30

31 /// <summary>

32 /// 吃（虚方法）

33 /// </summary>

34 public virtual void Eat()

35 {

36 Console.WriteLine("我会吃!");

37 }

38

39 /// <summary>

40 /// 叫（虚方法）

41 /// </summary>

42 public virtual void Shout()

43 {

44 Console.WriteLine("我会叫！");

45 }

46 }

47

48 /// <summary>

49 /// 狗（子类）

50 /// </summary>

51 class Dog:Animal

52 {

53 string myName;

54 public Dog(string name): base(name)

55 {

56 myName = name;

57 }

58

59 /// <summary>

60 /// 名字(重写父类属性)

61 /// </summary>

62 public override string MyName

63 {

64 get { return "我是：狗狗，我叫："+this.name; }

65

66 }

67

68

69 /// <summary>

70 /// 吃(重写父类虚方法)

71 /// </summary>

72 public override void Eat()

73 {

74 Console.WriteLine("我喜欢吃肉!");

75 }

76

77 /// <summary>

78 /// 叫（重写父类方法）

79 /// </summary>

80 public override void Shout()

81 {

82 Console.WriteLine("汪!汪!汪！");

83 }

84 }

[复制代码](javascript:void(0);)

//调用方法

Animal dog = new Dog("旺财");

string myName=dog.MyName;

Console.WriteLine(myName);

dog.Eat();

dog.Shout();

//运行结果如下：

我是：狗狗，我叫：旺财

我喜欢吃肉！

汪！汪！汪！

**重写（new）**

**new：覆盖**指的是不同类中（基类或派生类）有两个或多个返回类型、方法名、参数都相同，但是方法体不同的方法。但是这种**覆盖是一种表面上的覆盖，所以也叫隐藏**，被覆盖的父类方法是可以调用得到的。

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 动物类(父类)

3 /// </summary>

4 class Animal

5 {

6 /// <summary>

7 /// 名字

8 /// 说明：类和子类可访问

9 /// </summary>

10 protected string name;

11

12

13 /// <summary>

14 /// 构造函数

15 /// </summary>

16 /// <param name="name"></param>

17 public Animal(string name)

18 {

19 this.name=name;

20 }

21

22 /// <summary>

23 /// 名字(虚属性)

24 /// </summary>

25 public virtual string MyName

26 {

27 get { return this.name; }

28

29 }

30

31 /// <summary>

32 /// 吃（虚方法）

33 /// </summary>

34 public virtual void Eat()

35 {

36 Console.WriteLine("我会吃!");

37 }

38

39 /// <summary>

40 /// 叫（虚方法）

41 /// </summary>

42 public virtual void Shout()

43 {

44 Console.WriteLine("我会叫！");

45 }

46 }

47

48 /// <summary>

49 /// 狗（子类）

50 /// </summary>

51 class Dog:Animal

52 {

53 string myName;

54 public Dog(string name): base(name)

55 {

56 myName = name;

57 }

58 /// <summary>

59 /// 名字(重写父类属性)

60 /// </summary>

61 public override string MyName

62 {

63 get { return "我是：狗狗，我叫："+this.name; }

64 }

65

66 /// <summary>

67 /// 吃(重写父类虚方法)

68 /// </summary>

69 new public void Eat()

70 {

71 Console.WriteLine("我喜欢吃肉!");

72 }

73

74 /// <summary>

75 /// 叫（重写父类方法）

76 /// </summary>

77 public new void Shout()

78 {

79 Console.WriteLine("汪!汪!汪！");

80 }

81 }

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//调用方法 使用new重写，则只调用父类的方法

Animal dog = new Dog("旺财");

string myName=dog.MyName;

Console.WriteLine(myName);

dog.Eat();

dog.Shout();

//执行结果如下：

我是：狗狗，我叫：旺财

我会吃！

我会叫！

[复制代码](javascript:void(0);)

如下改一下调用方法：

[复制代码](javascript:void(0);)

//调用方法

Dog dog = new Dog("旺财");

string myName=dog.MyName;

Console.WriteLine(myName);

dog.Eat();

dog.Shout();

//执行结果如下：

我是：狗狗，我叫：旺财！

我爱吃肉！

汪！汪！汪！

[复制代码](javascript:void(0);)

可以看出，当派生类Dog的Eat()方法使用new修饰时，Dog的对象转换为Animal对象后，调用的是Animal类中的Eat()方法。其实可以理解为，使用new关键字后，使得Dog中的Eat()方法和Animal中的Eat()方法成为毫不相关的两个方法，只是它们的名字碰巧相同而已。所以， Animal类中的Eat()方法不管用还是不用virtual修饰，也不管访问权限如何，或者是没有，都不会对Dog的Eat()方法产生什么影响（只是因为使用了new关键字，如果Dog类没用从Animal类继承Eat()方法，编译器会输出警告）。

我想这是设计者有意这么设计的，因为有时候我们就是要达到这种效果。**严格的说，不能说通过使用new来实现多态，只能说在某些特定的时候碰巧实现了多态的效果。**

**3.3 要点：**

a.**多态是面向对象的重要特性之一，指同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。**

b.**多态分为两种**：一种是编译时多态，使用重载实现；另一种是运行时多态，使用重写实现；

c.**重写有两种**，一种使用override关键词，另一种使用new关键词

d.**new重写实际上是对父类方法的隐藏，被覆盖的父类方法可以调用得到**。因此new可以重写（或说是隐藏）的父类方法不一定要定义为虚方法或抽象方法。只是如果父类方法是虚方法或抽象方法时会覆盖父类方法，如果不是，则隐藏。

**e.重载和覆盖的发生条件**：  
**重载,必然发生在一个类中**,函数名相同,参数类型或者顺序不同构成重载,与返回类型无关  
**重写,必然发生在基类和派生类中**,其类函数用virtual修饰,派生类用override修饰  
覆盖,在子类中写一个和基类一样名字(参数不同也算)的非虚函数,会让基类中的函数被隐藏,编译后会提示要求使用New关键字 new 修饰

隐藏，在子类中可以通过new 隐藏父类的方法

f.new覆盖与重写、重载的区别：

当子类与父类的参数不同时

当基类函数不是虚函数时，基类函数将被隐藏。（因为子类和基类不在同一范围内，所以不是重载）

当基类函数是虚函数时，基类函数将被隐藏。（因为子类和基类不在同一范围内，所以不是重载；因为参数不同，所以不是重写）

当子类与父类的参数相同时

当基类函数不是虚函数时，基类函数将被隐藏。（因为子类和基类不在同一范围内，所以不是重载，因为基类不是虚函数，所以是隐藏不是重写）

当基类函数是虚函数时，基类函数将被覆盖。（因为子类和基类不在同一范围内，所以不是重载）

## 19、术语表& [C# .ToString() 格式化](http://www.cnblogs.com/bignjl/articles/1935645.html)

***术语表***

generics：泛型  
type-safe：类型安全  
collection: 集合  
compiler：编译器  
run time：程序运行时  
object: 对象  
.NET library：.Net类库  
value type: 值类型  
box: 装箱  
unbox: 拆箱   
implicity: 隐式  
explicity: 显式  
linked list: 线性链表  
node: 结点  
indexer: 索引器

[***C# .ToString() 格式化***](http://www.cnblogs.com/bignjl/articles/1935645.html)

c# ToString() 格式化字符串  格式化数值：有时，我们可能需要将数值以一定的格式来呈现，就需要对数值进行格式化。我们使用格式字符串指定格式。格式字符串采用以下形式：Axx，其中 A 为格式说明符，指定格式化类型，xx 为精度说明符，控制格式化输出的有效位数或小数位数。   
格式说明符 说明 示例 输出   
C 货币 2.5.ToString("C") ￥2.50    
D 十进制数 25.ToString("D5") 00025   
E 科学型 25000.ToString("E") 2.500000E+005   
F 固定点 25.ToString("F2") 25.00   
G 常规 2.5.ToString("G") 2.5   
N 数字 2500000.ToString("N") 2,500,000.00   
X 十六进制 255.ToString("X") FF

**C#的String.Format格式化举例**  
stringstr1 =string.Format("{0:N1}",56789);               //result: 56,789.0  
stringstr2 =string.Format("{0:N2}",56789);               //result: 56,789.00  
stringstr3 =string.Format("{0:N3}",56789);               //result: 56,789.000  
stringstr8 =string.Format("{0:F1}",56789);               //result: 56789.0  
stringstr9 =string.Format("{0:F2}",56789);               //result: 56789.00  
stringstr11 =(56789 / 100.0).ToString("#.##");           //result: 567.89  
stringstr12 =(56789 / 100).ToString("#.##");             //result: 567

C 或 c  
货币  
Console.Write("{0:C}", 2.5);  //$2.50  
Console.Write("{0:C}", -2.5); //($2.50)

D 或 d  
十进制数  
Console.Write("{0:D5}", 25);  //00025  
   
E 或 e  
科学型  
Console.Write("{0:E}", 250000);  //2.500000E+005  
   
F 或 f  
固定点  
Console.Write("{0:F2}", 25);  //25.00  
Console.Write("{0:F0}", 25);  //25  
   
G 或 g  
常规  
Console.Write("{0:G}", 2.5);  //2.5  
   
N 或 n  
数字  
Console.Write("{0:N}", 2500000);  //2,500,000.00  
   
X 或 x  
十六进制  
Console.Write("{0:X}", 250);  //FA  
Console.Write("{0:X}", 0xffff);  //FFFF

## 20 、Queue队列&[C#的new关键字的几种用法](http://blog.csdn.net/wu5101608/article/details/27826141)

***Eg1***

　Queue队列就是先进先出。它并没有实现 IList，ICollection。所以它不能按索引访问元素，不能使用Add和Remove。下面是 Queue的一些方法和属性

　　Enqueue():在队列的末端添加元素

　　Dequeue():在队列的头部读取和删除一个元素，注意，这里读取元素的同时也删除了这个元素。如果队列中不再有任何元素。就抛出异常

　　Peek():在队列的头读取一个元素，但是不删除它

　　Count：返回队列中的元素个数

　　TrimExcess():重新设置队列的容量，因为调用Dequeue方法读取删除元素后不会重新设置队列的容量。

　　Contains():确定某个元素是否在队列中

　　CopyTo():把元素队列复制到一个已有的数组中

　　ToArray():返回一个包含元素的新数组

做一个小例子来说明下队列的用法:

首先建立一个实体类

[复制代码](javascript:void(0);)

[Serializable]

public class Person:IEquatable<Person>

{

private string name;

public string Name

{

get { return name; }

set { name = value; }

}

private string phone;

public string Phone

{

get { return phone; }

set { phone = value; }

}

private bool? isGet;

public bool? IsGet

{

get { return isGet; }

set { isGet = value; }

}

public Person() { }

public Person(string name, string phone,bool? isGet)

{

this.name = name;

this.phone = phone;

this.isGet = isGet;

}

public bool Equals(Person person)

{

if (person == null)

{

return false;

}

if (this.name == person.name && this.phone == person.phone)

{

return true;

}

else

{

return false;

}

}

}

***Eg2***

[(C#)使用队列(Queue)解决简单的并发问题](http://blog.csdn.net/xuejunling/article/details/46909111)

有一个场景：一个抢购的项目，假设有5件商品，谁先抢到谁可以买，但是如果此时此刻（这里的此时此刻假设是相同的时间），有100人去抢这个商品，如果使用平时的方法会出现什么情况呢？你懂的，这里所说是就是有关并发的问题。

平时我们去超市购物去结账的时候就是排队，这里我们先让抢购人排好队，按时间，谁先点击的抢购按钮谁就排在前面，这样就形成了一个队列，然后我们再对这个队列处理，这样就不会出现并发的问题了。（至少可以处理这样简单的并发，这里不讨论太复杂的并发）

案例：

要求：有一个发布文章的接口，每发布一篇文章，调用一下接口。（这里不用批量发布，为了讲解这个）

建立一个这样的处理程序类，BusinessInfoHelper.cs

**[csharp]** [view plain](http://blog.csdn.net/xuejunling/article/details/46909111) [copy](http://blog.csdn.net/xuejunling/article/details/46909111)

1. namespace MyNameSpace
3. {
4. //队列临时类
5. public class QueueInfo
6. {
7. public string medias { get; set; }
8. public string proids { get; set; }
9. public string host { get; set; }
10. public string userid { get; set; }
11. public string feedid { get; set; }
12. }
14. public class BusinessInfoHelper
15. {
16. #region 解决发布时含有优质媒体时，前台页面卡住的现象
17. //原理：利用生产者消费者模式进行入列出列操作
19. public readonly static BusinessInfoHelper Instance = new BusinessInfoHelper();
20. private BusinessInfoHelper()
21. { }
23. private Queue<QueueInfo> ListQueue = new Queue<QueueInfo>();
25. public void AddQueue(string medias, string proids, string host, string userid, string feedid) //入列
26. {
27. QueueInfo queueinfo = new QueueInfo();
29. queueinfo.medias = medias;
30. queueinfo.proids = proids;
31. queueinfo.host = host;
32. queueinfo.userid = userid;
33. queueinfo.feedid = feedid;
34. ListQueue.Enqueue(queueinfo);
35. }
37. public void Start()//启动
38. {
39. Thread thread = new Thread(threadStart);
40. thread.IsBackground = true;
41. thread.Start();
42. }
44. private void threadStart()
45. {
46. while (true)
47. {
48. if (ListQueue.Count > 0)
49. {
50. try
51. {
52. ScanQueue();
53. }
54. catch (Exception ex)
55. {
56. LO\_LogInfo.WLlog(ex.ToString());
57. }
58. }
59. else
60. {
61. //没有任务，休息3秒钟
62. Thread.Sleep(3000);
63. }
64. }
65. }
67. //要执行的方法
68. private void ScanQueue()
69. {
70. while (ListQueue.Count > 0)
71. {
72. try
73. {
74. //从队列中取出
75. QueueInfo queueinfo = ListQueue.Dequeue();
77. //取出的queueinfo就可以用了，里面有你要的东西
78. //以下就是处理程序了
79. //。。。。。。
81. }
82. catch (Exception ex)
83. {
84. throw;
85. }
86. }
87. }

90. #endregion
91. }

[***C#的new关键字的几种用法***](http://blog.csdn.net/wu5101608/article/details/27826141) ***1***

在 C# 中，new 关键字可用作运算符、修饰符或约束。

**new 运算符**

用于创建对象和调用构造函数。

**new 修饰符**

用于向基类成员隐藏继承成员。

**new 约束**

用于在泛型声明中约束可能用作类型参数的参数的类型。

**new 修饰符（C# 参考）**

在用作修饰符时，new 关键字可以显式隐藏从基类继承的成员。隐藏继承的成员意味着该成员的派生版本将替换基类版本。在不使用 new 修饰符的情况下隐藏成员是允许的，但会生成警告。使用 new 显式隐藏成员会取消此警告，并记录代之以派生版本这一事实。

若要隐藏继承的成员，请使用相同名称在派生类中声明该成员，并使用 new 修饰符修饰该成员

**new 运算符（C# 参考）**

**1.用于创建对象和调用构造函数。例如：**

Class1 o = new Class1();

**2.也用于为值类型调用默认的构造函数**

　例：int myInt = new int();

　myInt 初始化为 0，它是 int 类型的默认值。该语句的效果等同于：int myInt = 0;

**3.不能重载 new 运算符。**

**4.如果 new 运算符分配内存失败，则它将引发 OutOfMemoryException 异常**

**new 约束（C# 参考）**

new 约束指定泛型类声明中的任何类型参数都必须有公共的无参数构造函数。当泛型类创建类型的新实例时，将此约束应用于类型参数，如下面的示例所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class ItemFactory<T> where T : new()  {  public T GetNewItem()  {  return new T();  }  } |

通过继承隐藏名称采用下列形式之一：

　　1.引入类或结构中的常数、指定、属性或类型隐藏具有相同名称的所有基类成员。

　　2.引入类或结构中的方法隐藏基类中具有相同名称的属性、字段和类型。同时也隐藏具有相同签名的所有基类方法。

　　3.引入类或结构中的索引器将隐藏具有相同名称的所有基类索引器。

　　4.在同一成员上同时使用 new 和 override 是错误的。

　　注意：在不隐藏继承成员的声明中使用 new 修饰符将生成警告。

　示例

　　在该例中，嵌套类 MyClass 隐藏了基类中具有相同名称的类。该例不仅说明了如何使用完全限定名访问隐藏类成员，同时也说明了如何使用 new 修饰符消除警告消息。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | using System;   public class MyBaseC   {   public class MyClass   {   public int x = 200;   public int y;   }   }   public class MyDerivedC : MyBaseC   {   new public class MyClass // nested type hiding the base type members   {   public int x = 100;   public int y;   public int z;   }   public static void Main()  {   // Creating object from the overlapping class:   MyClass S1 = new MyClass();   // Creating object from the hidden class:  MyBaseC.MyClass S2 = new MyBaseC.MyClass();   Console.WriteLine(S1.x);   Console.WriteLine(S2.x);   }   }    输出  100  200 |

[***C#的new关键字的几种用法***](http://blog.csdn.net/wu5101608/article/details/27826141) ***2***

1.new 运算符用于创建对象和调用构造函数：

 Test01 test = new Test01();

new 运算符还可以用于创建值类型的默认构造函数：

int i = new int();

i被初始化为0，因为它是int类型的默认值。该语句等同于：int i = 0;

如果new运算符分配内存失败，则它将引发OutOfMemoryException异常.

不能重载new 运算符

请记住，为结构声明默认的构造函数是错误的，因为每一个值类型都隐式具有一个公共的默认构造函数。可以在结构类型上声明参数化构造函数以设置其初始值，但是，只有在需要默认值之外的值时才必须这样做。

值类型对象（例如结构）是在堆栈上创建的，而引用类型对象（例如类）是在堆上创建的。两种类型的对象都是自动销毁的，但是，基于值类型的对象是在超出范围时销毁，而基于引用类型的对象则是在对该对象的最后一个引用被移除之后在某个不确定的时间销毁。对于占用固定资源（例如大量内存、文件句柄或网络连接）的引用类型，有时需要使用确定性终止以确保对象被尽快销毁

2.new 修饰符可以显式隐藏从基类继承的成员，隐藏继承的成员时，该成员的派生版本将替换基类版本。虽然可以在不使用new修饰符的情况下隐藏成员，但会生成警告，如果用new显示隐藏成员，则会取消该警告，并记录要替换为派生版本这一事实。如要隐藏继承的成员，请使用相同名称在派生类中声明该成员，并使用new修饰符修饰该成员。例如：

    class Program

    {

        public int x;

        public void TestMethod()

        {

            Console.WriteLine("我是基类的方法");

        }

    }

    class Test:Program

    {

        new public void TestMethod()

        {

            Console.WriteLine("我是派生类的方法");

        }

    }

在此示例中，Test.TestMethod隐藏了Program.TestMethod。字段x不受影响，因为它没有被类似名称的字段隐藏。

通过继承隐藏名称采用以下形式之一：

1。引入类或结构中的常数,指定，属性或类型隐藏具有相同名称的所有基类成员。

2。引入类或结构中的方法隐藏基类中具有相同名称的属性，字段和类型，同时也隐藏具有相同签名的所有基类方法。

3。引入类或结构中的索引器将具有相同名称的所有基类索引器。

对同一成员同时使用new和override是错误的做法，因为这两个修饰符的含义互斥，new修饰符会用同样的名称创建一个新成员并使原始成员变为隐藏的。override修饰符会扩展继承成员的实现，在不隐藏继承成员的声明中使用new修饰符将会生成警告。

3.new 约束指定泛型类声明中的任何类型参数都必须有公共的无参数构造函数，如果使用new约束，则该类型不能为抽象类型。

当泛型类型创建类型的新实例，请将new 约束应用于类型参数，如下面示例所示

 class ItemFactory<T> where T : new()

    {

        public T GetNewItem()

        {

            return new T();

        }

    }

当与其他约束一起使用时，new（）约束必须最后指定：

  public class ItemFactory2<T>

     where T : IComparable, new()

    {

    }

## 21、[分分钟用上C#中的委托和事件](http://www.cnblogs.com/yinqixin/p/5056307.html)

每一个初学C#的程序猿，在刚刚碰到委托和事件的概念时，估计都是望而却步，茫然摸不到头脑的。百度一搜，关于概念介绍的文章大把大把的，当然也不乏深入浅出的好文章。可看完这些文章，大多数新手，估计也只是信心满满的觉得自己看懂了，一旦要在自己的程序中用上委托和事件，就傻眼了，根本不知道什么情况下该用，也不知道具体怎么用。

本文没有任何概念介绍，旨在帮助同学们快速用上委托和事件，进入C#的更高境界。

* **首先，我们需要知道，到底在什么情况下必须使用委托和事件呢？**

请看下面的场景：首领A要搞一场鸿门宴，吩咐部下B和C各自带队埋伏在屏风两侧，约定以杯为令：若左手举杯，则B带队杀出；若右手举杯，则C带队杀出；若直接摔杯，则B和C同时杀出。B和C袭击的具体方法，首领A并不关心。(完整代码在全文最后。)

有脑子反应快的同学，马上说，这个逻辑，用一个IF条件判断语句不就搞定了吗：

[复制代码](javascript:void(0);)

1 if(左手举杯)

2 {

3 B带队杀出;

4 }

5 else if(右手举杯)

6 {

7 C带队杀出;

8 }

9 else if(直接摔杯)

10 {

11 B带队杀出;

12 C带队杀出;

13 }

14 else

15 {

16 按兵不动

17 }

[复制代码](javascript:void(0);)

如果真这么简单，那我还写这篇文章干嘛呢。请这位同学认真想想：首领A会什么时候发出信号呢？估计连他自己都不知道。那么上面这段伪码，你打算插入到哪里呢？难道B和C就一直不停的用While循环做上述判断吗？显然不合适吧。更重要的，如果扩展一下，部下不止B和C，而是B、C、D、E等等，每个人都有自己不同的响应领导号召的方法，那么对首领A而言，要分别去跟每个部下打交道，上述的伪码，会进行无尽的扩展和完善，显然也不是我们所希望的。

正确的逻辑应该是，B和C不管宴席上发生的任何其他事情，只等首领发出举杯或者摔杯的信号，一旦首领A发出信号，相当于通知了所有部下，凡是约定好的部下，都立马各自行动！

我们定义三个类，分别模拟首领A和部下B、C：

[复制代码](javascript:void(0);)

/// <summary>

/// 首领A

/// </summary>

public class A

{

}

/// <summary>

/// 部下B

/// </summary>

public class B

{

}

/// <summary>

/// 部下C

/// </summary>

public class C

{

}

[复制代码](javascript:void(0);)

首领A的类有举杯和摔杯两种方法，其中，举杯方法带有一个形参，用于传递是左手举杯，还是右手举杯。部下B和C的类，各自有一种攻击方法。

[复制代码](javascript:void(0);)

/// <summary>

/// 首领A

/// </summary>

public class A

{

/// <summary>

/// 举杯

/// </summary>

/// <param name="hand">手：左、右</param>

public void Raise(string hand)

{

Console.WriteLine("首领A{0}手举杯", hand);

}

/// <summary>

/// 摔杯

/// </summary>

public void Fall()

{

Console.WriteLine("首领A摔杯");

}

}

/// <summary>

/// 部下B

/// </summary>

public class B

{

/// <summary>

/// 攻击

/// </summary>

public void Attack()

{

Console.WriteLine("部下B发起攻击");

}

}

/// <summary>

/// 部下C

/// </summary>

public class C

{

/// <summary>

/// 攻击

/// </summary>

public void Attack()

{

Console.WriteLine("部下C发起攻击");

}

}

[复制代码](javascript:void(0);)

至此，三个独立的类，构造完毕。那么，怎么让部下B和C，根据首领的暗语才去相应的行动呢？  
我们需要在首领A类中，采用一种方法，把其意图传递出去。在首领A类之前，分别定义一个带形参的举杯委托RaiseEventHandler和一个不带形参的摔杯委托FallEventHandler。命名规则是在准备传递的方法名后加上EventHandler，不要问我为什么，介绍为什么的文章多得是，大家大可以去查阅。本文的宗旨是，教会大家如何快速的使用委托和事件。当然，你也可以不遵循此命名规则，对程序的运行没有任何影响，只是会增加自己或他人日后阅读代码的难度。

delegate void RaiseEventHandler(string hand);

delegate void FallEventHandler();

然后，在首领A类中，定义两个事件。这个类似于大家在WinForm中拖放一个Button后，双击该Button，即可编辑其事件。

[复制代码](javascript:void(0);)

/// <summary>

/// 首领A举杯事件

/// </summary>

public event RaiseEventHandler RaiseEvent;

/// <summary>

/// 首领A摔杯事件

/// </summary>

public event FallEventHandler FallEvent;

[复制代码](javascript:void(0);)

好了，做完上面两个步骤后，就可以在首领A的举杯和摔杯方法中，调用上述两个事件了。这样，如果B和C中订阅该事件，便可自动执行了。

[复制代码](javascript:void(0);)

/// <summary>

/// 举杯

/// </summary>

/// <param name="hand">手：左、右</param>

public void Raise(string hand)

{

Console.WriteLine("首领A{0}手举杯", hand);

// 调用举杯事件，传入左或右手作为参数

if (RaiseEvent!=null)

{

RaiseEvent(hand);

}

}

/// <summary>

/// 摔杯

/// </summary>

public void Fall()

{

Console.WriteLine("首领A摔杯");

// 调用摔杯事件

if (FallEvent!=null)

{

FallEvent();

}

}

[复制代码](javascript:void(0);)

同样，不要问为什么这样写。按照上述操作即可。一旦用熟这种方法，体会其中含义，自然就明白为何这么写了。

在部下B和C心中，必须存在首领A，才能执行A的暗示吧。所以，在B类和C类中，需要声明一个A，该声明可以通过B和C的构造函数进行实例化。实例化之后，便可在类B和类C中订阅类A的事件了。

完整代码如下：

[复制代码](javascript:void(0);)

1 class Program

2 {

3 static void Main(string[] args)

4 {

5 A a = new A(); // 定义首领A

6

7 B b = new B(a); // 定义部下B

8

9 C c = new C(a); // 定义部下C

10

11 // 首领A左手举杯

12 a.Raise("左");

13

14 // 首领A右手举杯

15 //a.Raise("右");

16

17 // 首领A摔杯

18 //a.Fall();

19

20 Console.ReadLine();

21 // 由于B和C订阅了A的事件，所以无需任何代码，B和C均会按照约定进行动作。

22 }

23 }

24

25 /// <summary>

26 /// 首领A举杯委托

27 /// </summary>

28 /// <param name="hand">手：左、右</param>

29 public delegate void RaiseEventHandler(string hand);

30 /// <summary>

31 /// 首领A摔杯委托

32 /// </summary>

33 public delegate void FallEventHandler();

34 /// <summary>

35 /// 首领A

36 /// </summary>

37 public class A

38 {

39 /// <summary>

40 /// 首领A举杯事件

41 /// </summary>

42 public event RaiseEventHandler RaiseEvent;

43 /// <summary>

44 /// 首领A摔杯事件

45 /// </summary>

46 public event FallEventHandler FallEvent;

47

48 /// <summary>

49 /// 举杯

50 /// </summary>

51 /// <param name="hand">手：左、右</param>

52 public void Raise(string hand)

53 {

54 Console.WriteLine("首领A{0}手举杯", hand);

55 // 调用举杯事件，传入左或右手作为参数

56 if (RaiseEvent!=null)

57 {

58 RaiseEvent(hand);

59 }

60 }

61 /// <summary>

62 /// 摔杯

63 /// </summary>

64 public void Fall()

65 {

66 Console.WriteLine("首领A摔杯");

67 // 调用摔杯事件

68 if (FallEvent!=null)

69 {

70 FallEvent();

71 }

72 }

73 }

74 /// <summary>

75 /// 部下B

76 /// </summary>

77 public class B

78 {

79 A a;

80

81 public B(A a)

82 {

83 this.a = a;

84 a.RaiseEvent += new RaiseEventHandler(a\_RaiseEvent); // 订阅举杯事件

85 a.FallEvent += new FallEventHandler(a\_FallEvent); // 订阅摔杯事件

86 }

87 /// <summary>

88 /// 首领举杯时的动作

89 /// </summary>

90 /// <param name="hand">若首领A左手举杯，则B攻击</param>

91 void a\_RaiseEvent(string hand)

92 {

93 if (hand.Equals("左"))

94 {

95 Attack();

96 }

97 }

98

99 /// <summary>

100 /// 首领摔杯时的动作

101 /// </summary>

102 void a\_FallEvent()

103 {

104 Attack();

105 }

106

107 /// <summary>

108 /// 攻击

109 /// </summary>

110 public void Attack()

111 {

112 Console.WriteLine("部下B发起攻击，大喊：猛人张飞来也！");

113 }

114 }

115 /// <summary>

116 /// 部下C

117 /// </summary>

118 public class C

119 {

120 A a;

121 public C(A a)

122 {

123 this.a = a;

124 a.RaiseEvent += new RaiseEventHandler(a\_RaiseEvent); // 订阅举杯事件

125 a.FallEvent += new FallEventHandler(a\_FallEvent); // 订阅摔杯事件

126 }

127 /// <summary>

128 /// 首领举杯时的动作

129 /// </summary>

130 /// <param name="hand">若首领A右手举杯，则攻击</param>

131 void a\_RaiseEvent(string hand)

132 {

133 if (hand.Equals("右"))

134 {

135 Attack();

136 }

137 }

138

139 /// <summary>

140 /// 首领摔杯时的动作

141 /// </summary>

142 void a\_FallEvent()

143 {

144 Attack();

145 }

146 /// <summary>

147 /// 攻击

148 /// </summary>

149 public void Attack()

150 {

151 Console.WriteLine("部下C发起攻击，一套落英神掌打得虎虎生威~");

152 }

153 }

## 22 [NULL，""，String.Empty三者在C#中的区别](http://www.cnblogs.com/liuyaozhi/p/5809521.html)

(1)NULL  
null 关键字是表示不引用任何对象的空引用的文字值。null 是引用类型变量的默认值。那么也只有引用型的变量可以为NULL，如果int i=null,的话，是不可以的，因为Int是值类型的。

(2)""和String.Empty  
   这两个都是表示空字符串。只不过""理论上重新开辟内存空间，而String.Empty指向一处。不过优化器会优化的！

   string.Empty不分配存储空间, ""分配一个长度为空的存储空间,所以一般用string.Empty,为了以后跨平台，还是用string.empty。

在C# 中，大多数情况下 "" 和 string.Empty 可以互换使用。比如：

string s = "";

string s2 = string.Empty;

if (s == string.Empty) {//} if语句成立

（3）判定为空字符串的几种写法，按照性能从高到低的顺序是：

s.Length == 0 优于 s == string.Empty 优于 s == ""

注意：

   1.string str1="" 和 string str2=null 的区别。

str1是一个空字符串，空字符串是一个特殊的字符串，只不过这个字符串的值为空，在内存中是有准确的指向的，

string str2=null,这样定义后，只是定义了一个string 类的引用，str2并没有指向任何地方，在使用前如果不实例化的话，都将报错。

23[**c#中params关键字应用**](http://www.cnblogs.com/yiyi20120822/archive/2012/11/29/2794009.html)

c#params应用

params 是C#开发语言中关键字， params主要的用处是在给函数传参数的时候用，就是当函数的参数不固定的时候。 在方法声明中的 params 关键字之后不允许任何其他参数，并且在方法声明中只允许一个 params 关键字。 关于参数数组，需掌握以下几点。

（1）若形参表中含一个参数数组，则该参数数组必须位于形参列表的最后；

（2）参数数组必须是一维数组；

（3）不允许将params修饰符与ref和out修饰符组合起来使用；

（4）与参数数组对应的实参可以是同一类型的数组名，也可以是任意多个与该数组的元素属于同一类型的变量；

（5）若实参是数组则按引用传递，若实参是变量或表达式则按值传递。

（6）用法：可变的方法参数，也称数组型参数，适合于方法的参数个数不知的情况，用于传递大量的数组集合参数；当使用数组参数时，可通过使用params关键字在形参表中指定多种方法参数，并在方法的参数表中指定一个数组，

形式为：方法修饰符　返回类型　方法名（params　类型［］　变量名） 如带有参数的SQL 语句，不同的表的字段数量也不同， 当你更新修改的时候就可以用。例如：

params using System;

public class MyClass

{

    public static void UseParams(params int[] list)

   {

       for (int i = 0 ; i < list.Length; i++)

      { Console.WriteLine(list[i]); }

      Console.WriteLine();

   }

   public static void UseParams2(params object[] list)

   {

       for (int i = 0 ; i < list.Length; i++)

      { Console.WriteLine(list[i]); }

      Console.WriteLine();

   }

   static void Main()

   {

　　　　UseParams(1, 2, 3);

　　　　UseParams2(1, 'a', "test");

　　　　 // An array of objects can also be passed, as long as

　　　　// the array type matches the method being called.

　　　　int[] myarray = new int[3] {10,11,12};

　　　　UseParams(myarray);

　　}

}

　　输出：1231atest101112

## 23 C#中窗体程序中的this.SuspendLayout()和this.ResumeLayout()

   挂起布局逻辑与恢复布局逻辑

    有时候，需要重新布置整个窗口里的控件的位置布局，如刚刚初始化时就是一个例子。这样在每增加一个有dock等属性的控件时，由于这些布局都是相对性布局，都需要重新计算位置什么的，并实际执行布局。如果一个窗口上有50个控件，这个布局处理就要执行50次，并触发50 \* n次的布局有关的事件。

而实际上，在定义布局方案过程中，这些布局是没有必要每个控件定义之后就立即执行布局的，也没有必要处理因此发出的布局事件。更合理的办法，是在全部定义完所有控件的布局方案之后，才一次性计算布局结果，并一次执行。于是，就设计了这两个函数。第一个函数用于暂停实际的布局，而第二个函数用于恢复实际布局。在程序里，如果你需要大规模调整布局方案时，就可以先调用suspendlayout，暂停布局执行，然后定义谁在前，谁在后，谁在上，谁在下，谁填充左，谁填充右。布置完毕。resumelayout，于是，一次性处理完毕。

如果注释这两个方法的调用，效果上也不会有什么变化，只是在性能上会造成不必要的开销。

# 一、C#基础知识(2)

## 1.0 [C#中substring ()的用法](http://www.cnblogs.com/nextsoft/articles/782336.html)

String.SubString(int   index,int   length)     
  index:开始位置，从0开始       
  length:你要取的子字符串的长度     
  
示例：

using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace str\_sub  
{  
class Program  
{  
static void Main(string[] args)  
{  
string myString = "Hello Word!";  
  
//Substring()在C#中有两个重载函数  
//分别如下示例  
  
string subString1 = myString.Substring(0);  
  
//如果传入参数为一个长整, 且大于等于0,  
//则以这个长整的位置为起始,  
//截取之后余下所有作为字串.  
//如若传入值小于0,  
//系统会抛出ArgumentOutOfRange异常  
//表明参数范围出界  
  
  
  
string subString2 = myString.Substring(0, 5);  
  
//如果传入了两个长整参数,  
//前一个为参数子串在原串的起始位置  
//后一个参数为子串的长度  
//如不合条件同样出现上述异常  
  
  
Console.WriteLine(subString1);  
Console.WriteLine(subString2);  
Console.ReadLine();  
}  
}  
}  
  
程序输出的结果:  
  
Hello Word!  
Hello

拓展

[**C#中substr和substring的区别**](http://www.cnblogs.com/shlcn/archive/2011/07/21/2112869.html)

这两个方法比较有意思，很像，但结果却是相差十万八千里。我也是从误用它们开始，慢慢进入到它们的世界。因为很巧的是当start是为0的时候，二者的结果是一样，而我在这之前就看过substr这个方法。所以让我产生一种错觉，就是只有substr，没有substring。当我发现是两个方法来的时候，我会傻笑自己的无知。下面介绍这两个方法：

substr(start[,length])表示从start位置开始取length个字符串  
substring(start,end)表示从start，到end之间的字符串，包括start位置的字符但是不包括end位置的字符

Js的substring和C#的Substring的作用都是从一个字符串中截取出一个子字符串，但它们的使用方法却有很大的不同，下边我们来比较看看：

Js的substring

语法：

程序代码  
String.substring(start, end)

说明：  
返回一个从start开始到end(不包含end)的子字符串。

示例：

程序代码  
var str="abcdefgh";  
document.write(str.substring(0,1));//return:a  
document.write(str.substring(2,5));//return:cde  
document.write(str.substring(7,8));//return:h

C#的Substring

语法：

程序代码  
String.Substring(int startIndex)  
String.Substring(int startIndex, int length)

说明：  
返回一个从startIndex开始到结束的子字符串，或返回一个从startIndex开始，长度为length的子字符串。

示例：

程序代码  
string str = "abcdefgh";  
Response.Write(str.Substring(0,1));//return:a  
Response.Write(str.Substring(2,3));//return:cde  
Response.Write(str.Substring(7,1));//return:h  
Response.Write(str.Substring(7));//return:h  
Response.Write(str.Substring(10));//error:startIndex 不能大于字符串长度。  
Response.Write(str.Substring(7,10));//error:索引和长度必须引用该字符串内的位置。

经过上边的说明对它们的使用应该有个比较清楚的认识了，但对Js的substring还有几点要说明：

1.start不一定就是第一个参数，end也不一定就是第二个参数，substring(3,1)时，开始位置是1，结束位置是3；  
2.当要返回的子字符串是从开始位置到结束时，end的值必须大于等于字符串的长度，如上边的str.substring(7,8)，按照索引从0开始算的话end的最大值为7，但这边却用8，当然，使用大于8的数返回的结果也是一样的，这点比较有意思。

## 2.0 [C#中IndexOf的使用](http://www.cnblogs.com/yuerdongni/archive/2011/10/09/2204806.html)

IndexOf()   
查找字串中指定字符或字串首次出现的位置,返首索引值，如：   
str1.IndexOf("字")； //查找“字”在str1中的索引值（位置）   
str1.IndexOf("字串")；//查找“字串”的第一个字符在str1中的索引值（位置）   
str1.IndexOf("字",start,end)；//从str1第start+1个字符起，查找end个字符，查找“字”在字符串STR1中的位置[从第一个字符算起]注意:start+end不能大于str1的长度

indexof参数为string,在字符串中寻找参数字符串第一次出现的位置并返回该位置。如string s="0123dfdfdf";int i=s.indexof("df");这时i==4。   
如果需要更强大的字符串解析功能应该用Regex类，使用正则表达式对字符串进行匹配。

indexof() ：在字符串中从前向后定位字符和字符串；所有的返回值都是指在字符串的绝对位置，如为空则为- 1

string test="asdfjsdfjgkfasdsfsgfhgjgfjgdddd";

test.indexof(’d’) =2 //从前向后 定位 d 第一次出现的位置  
test.indexof(’d’,1) =2 //从前向后 定位 d 从第三个字符串 第一次出现的位置  
test.indexof(’d’,5,2) =6 //从前向后 定位 d 从第5 位开始查，查2位，即 从第5位到第7位；

lastindexof() ：在字符串中从后向前定位字符和字符串；、  
用法和 indexof() 完全相同。

下面介绍 IndexOfAny ||lastindexofany

他们接受字符数组做为变元，其他方法同上，返回数组中任何一个字符最早出现的下标位置

如下

char[] bbv={’s’,’c’,’b’};  
string abc = "acsdfgdfgchacscdsad";  
  
Response.Write(abc.IndexOfAny(bbv))=1  
Response.Write(abc.IndexOfAny(bbv, 5))=9  
Response.Write(abc.IndexOfAny(bbv, 5, 3))=9

lastindexofany 同上。

## 3.0 [\r \r\n \t的区别](http://www.cnblogs.com/f158w058/p/6548073.html)

**\n 软回车**：  
      在Windows 中表示换行且回到下一行的最开始位置。相当于Mac OS 里的 \r 的效果。  
      在Linux、unix 中只表示换行，但不会回到下一行的开始位置。  
  
**\r 软空格：**      在Linux、unix 中表示返回到当行的最开始位置。  
      在Mac OS 中表示换行且返回到下一行的最开始位置，相当于Windows 里的 \n 的效果。  
 **\t 跳格（移至下一列）。**

它们在双引号或定界符表示的字符串中有效，在单引号表示的字符串中无效。  
\r\n 一般一起用，用来表示键盘上的回车键，也可只用 \n。  
\t表示键盘上的“TAB”键。  
  
就像你使用 enter和shift+enter的区别，如果要显示在页面上的效果还要转化为HTML[代码](http://www.39g.com/)或用...

**文件中的换行符号：**   
  
linux,unix:     \r\n   
windows    :     \n     
Mac OS   ：   \r  
  
**对应：**  
\n LF或ASCII中的0x0A(10)   
\r CR或ASCII中的0x0D(13)   
\t 水平制表符-HT或ASCII中的0x09（9）   
\\ 反斜杠   
\$ 美圆符   
\" 双引号   
\' 单引号

**有关它们的来历并引起分歧垢原由：**  
  
    “回车”（carriage return）和“换行”（line feed）这两个概念的来历和区别了。  
在计算机还没有出现之前，有一种叫做电传打字机（Teletype Model 33）的玩意，每秒钟可以打10个字符。但是它有一个问题，就是打完一行换行的时候，要用去0.2秒，正好可以打两个字符。要是在这0.2秒里面，又有新的字符传过来，那么这个字符将丢失。  
  
    于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个叫做“回车”，告诉打字机把打印头定位在左边界；另一个叫做“换行”，告诉打字机把纸向下移一行。  
  
    这就是“换行”和“回车”的来历，从它们的英语名字上也可以看出一二。  
  
    后来，计算机发明了，这两个概念也就被般到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。  
  
    Unix系统里，每行结尾只有“<换行>”，即“\n”；Windows系统里面，每行结尾是“<换行><回车>”，即“\n\r”；Mac系统里，每行结尾是“<回车>”。一个直接后果是，Unix/Mac系统下的文件在Windows里打开的话，所有文字会变成一行；而Windows里的文件在Unix/Mac下打开的话，在每行的结尾可能会多出一个^M符号。  
  
    c语言编程时（windows系统）  
  
    \r 就是return 回到 本行 行首 这就会把这一行以前的输出 覆盖掉   
  
如：   
int main() {   
  
cout << "hahaha" << "\r" << "xixi" ;   
  
  
}   
    最后只显示 xixi 而 hahaha 背覆盖了   
  
    \n 是回车＋换行 把光标 先移到 行首 然后换到下一行 也就是 下一行的行首拉   
  
    int main() {   
  
    cout << "hahaha" << "\n" << "xixi" ;   
}   
  
则 显示 ：  
  
hahaha  
xixi

## 4.0 C# typeof() 和 GetType()区别

总得来说他们都是为了获取某个实例具体引用的数据类型System.Type。

1、GetType()方法继承自Object，所以C#中任何对象都具有GetType()方法，x.GetType()，其中x为变量名

2、typeof(x)中的x，必须是具体的类名、类型名称等，不可以是变量名称

3、System.Type.GetType()，有两个重载方法  
  
比如有这样一个变量i:   
Int32 i = new Int32();   
  
使用GetType()，i.GetType()返回值是Int32的类型，但是无法使用typeof(i)，因为i是一个变量，

使用typeof()，则只能：typeof(Int32)，返回的同样是Int32的类型。

下面我们具体来看下相关类的介绍：

**System.Type类**  
System.Type类定义了很多成员，可以用来检查某个类型的元数据，它们返回的类型大多位于System.Reflection命名空间中。举例来说，Type.GetMethods()返回一个MethodInfo类型的数组，Type.GetFields返回一个FieldInfo类型的数组等。System.Type提供的完整的成员组是很容易扩展的。

**使用System.Object.GetType()得到Type引用**可以用多种方法得到一个Type类的实例。但是，由于Type是一个抽象类，所以不能直接使用new关键字创建一个Type对象。对此我们的首选是：使用System.Object定义的GetType()方法，它返回一个表示当前对象元数据的Type类的实例：  
  
//使用一个SportsCar实例得到类型信息  
SportsCar sc = new SportsCar();  
Type t = sc.GetType();  
  
显而易见，要想使用这个方法，必须得到类型的编译时信息(这里是SportsCar类)，并且当前在内存中类型实例。

**使用typeof()得到Type引用**另一个取类型信息的方法是使用C# typeof操作符：  
  
//使用typeof得到类型  
Type t = typeof(SportsCar);  
  
类似System.Object.GetType()，使用typeof操作符，我们不需要建立一个实例来提取类型信息。但是，仍然需要知道类型的编译时信息，因为typeof需要的是类型的强类型名称，而不是文本表示。

**使用System.Type.GetType()得到Type引用**为了以更灵活的方式得到类型信息，我们可以调用System.Type类的静态成员GetType()，然后指定类型的完全限定名。采用这种方法，我们不需要得到正从中提取元数据的类型的编译时信息，

1）Type.GetType()方法被重载一：

允许我们指定两个布尔类型的参数，一个用来控制当类型找不到时是否抛出异常，另一个用来指示是否区分字符串大小写。  
  
例如：  
//使用静态的Type.GetType()方法获取类型信息(如果SportsCar没有找到，则忽略不抛出异常信息)  
Type t = Type.GetType("CarLibrary.SportsCar",false,true);  
  
2）Type.GetType()方法被重载二：

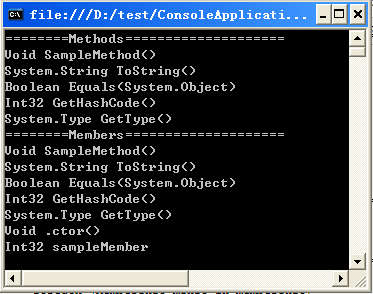
在上面的例子中，注意传入GetType()的字符串没有包含类型所在的程序集信息。在这种情况下，该类型便被认为是定义在当前执行的程序集中。但是，当希望得到一个外部私有程序集的类型元数据时，字符串参数必须使用类型完全限定名，加上类型所在的程序集的名字(每一个都用逗号隔开):  
  
//得到外部程序集中类型的类型信息  
Type t= Type.GetType("CarLibrary.SportsCar,CarLibrary");

**示例代码**

**[csharp]** [view plain](http://blog.csdn.net/letianok/article/details/7257117) [copy](http://blog.csdn.net/letianok/article/details/7257117)

1. **using** System;
2. **using** System.Reflection;
3. **public** **class** Program
4. {
5. **public** **int** sampleMember;
6. **public** **void** SampleMethod() { }
7. **static** **void** Main()
8. {
9. Type t = **typeof**(Program);
10. //也可通过下面这种方式操作
11. //Program obj = new Program();
12. //Type t = obj.GetType();
13. Console.WriteLine("========Methods====================");
14. MethodInfo[] methodInfo = t.GetMethods();
15. **foreach** (MethodInfo mInfo **in** methodInfo)
16. Console.WriteLine(mInfo.ToString());
17. Console.WriteLine("========Members====================");
18. MemberInfo[] memberInfo = t.GetMembers();
19. **foreach** (MemberInfo mInfo **in** memberInfo)
20. Console.WriteLine(mInfo.ToString());
21. Console.ReadLine();
22. }
23. }

运行结果如下



## C#接口

接口的相关陈述  
１．一个接口定义了一个契约。

２．接口可以包容方法、C#属性、事件、以及索引器。

３．在一个接口声明中，我们可以声明零个或者多个成员。

４．所有接口成员的默认访问类型都是public。

５．如果在接口成员声明中包括了任何修饰符，那么会产生一个编译器错误。

６．与一个非抽象类类似，一个抽象类必须提供接口中所有成员的实现，只要这些成员在这个类的基类中出现过。

接口的理解  
1. 面向接口编程利用OO的一个基本性质--多态，相同方法不同表现。可以这样想一下，client编写自己程序的时候，如果直接面向一个具体类写程序，那这个程序有个风吹草动的，那client就要受到影响，但如果面向一个接口就不同了，某个具体类变了，只知接口，不知具体类的client就可以完全不动。都说上层领导比较好当，因为可以干的事通常对老百姓来说是虚的，越虚就越不容易错。  
这个道理在OO中也是适用的。

2. 换个视角看，面向接口编程反映OO的另一个方面--封装，接口将具体实现封装了起来，可以不影响客户的情况下切换实现

3. 接口的作用，一言以蔽之，就是标志类的类别（type of class）。把不同类型的类归于不同的接口，可以更好的管理他们。OO的精髓，我以为，是对对象的抽象，最能体现这一点的就是接口。为什么我们讨论设计模式都只针对具备了抽象能力的语言（比如c++、java、c#等），就是因为设计模式所研究的，实际上就是如何合理的去抽象。（cowboy的名言是"抽象就是抽去像的部分"，看似调侃，实乃至理）。

空接口的使用  
在接口使用的时候,空接口有2种情况:  
1.类似于ObjectBuilder中的IBuilderPolicy,他们往往是做一个标记,表示需要某个功能.当然你也可以这么用,来表示你的类具有某个功能,实现了你的某个接口.

namespace Microsoft.Practices.ObjectBuilder

{

///

/// Represents a builder policy interface. Since there are no fixed requirements

/// for policies, it acts as a marker interface from which to derive all other

/// policy interfaces.

///

public interface IBuilderPolicy

{

}

}

2.你的接口继承了别的接口(非空),你的接口本身没有声明函数.这种情况一般是你不希望用户使用父接口作为参数类型,因为他们的用途可能不同,此时就可以用空接口来实现.

interface Text

{

    string getText();

}

interface SqlText : Text

{

}

可以看到,Text接口是用于返回一个字符串.而SqlText是一个空接口,它继承了Text接口.也就是说SqlText也是一种Text.但是我们可以知道,任何一个字符串不一定是Sql字符串,所以此时声明了一个SqlText接口来用于表名当前的字符串是一个Sql字符串.你的函数可以这样声明:

public void doQuery(SqlText sqlText)

而不是这样：

public void doQuery(Text text)

避免用户产生歧义的想法,一眼看去,就明白应该传入一个Sql字符串.

接口的成员为什么没有委托  
我们都知道C#的接口是可以包含事件的，其实当我们看到事件的时候，很容易就会想到委托，委托是事件的基础，如果对委托和事件不是特别清楚的程序员就一定不会明白，为什么C#接口中可以包含事件而不能有委托呢。其实简单的说法就是委托也是类型，delegate关键字引入的是一个新的类型，所以一个C#接口无法包容一个委托并把它当作成员；而event关键字引入的是一个新的成员，因此事件可以归人接口。理解这点，我们要从C#接口的使命说起，C#接口是一个契约，规范了接口实现者的行为，而不是要有些什么。很简单，例如"党员"是个接口，它肯定有个动作是"为人民服务"，"某某党员"实现了"党员" 这个接口，那么"某某党员"肯定也要"为人民服务"，至于你"某某党员"是否必须拥用"电脑"、"小孩"。那么"党员"这个接口中肯定不会有规定。这也就是接口的目的，规范了实现者的一些行为。所以C#接口的成员都是方法，不会有其它了。稍有c#常识的程序员都明白，c#中的属性，其实就是两个方法，一个Set方法，一个Get方法，同样事件和索引器也都是方法，请看下面的接口：

public interface IDrawingObject

{

event EventHandler OnDraw;

string Name

{

   get;

   set;

}

int this[int index]

{

   get;

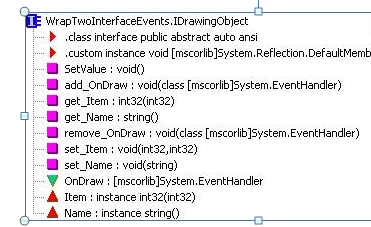
   set;

}

void SetValue();

}

该接口包含了c#接口所能接纳的所有成员，事件，属性，索引器，方法。把该接口编译后，我们用MSIL Disassembler工具查看一下：



这下大家都明白了，其实属性Name对应于Get\_Name(),Set\_Name()这两个方法，事件OnDraw对应于add\_OnDraw(),remove\_OnDraw()这两个方法，索引器对应于get\_Item(),set\_Item()这两个方法。在看下面的委托和类的定义：

public delegate void TestEventDelegate(object sender, System.EventArgs e);

class TestClass

{

  public void SetValue()

  { }

}

 看到了吧，定义一个委托和定义一个类是没有什么区别的，都是定义了个新的类型。所以C#接口是不能有委托的，除非微软告诉我们C#接口中是可以定义类的。

## 5.0 堆和栈

堆和栈的区别可以用如下的比喻来看出：      
  使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就 走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自  由度小。      
  使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由  度大。   (经典！)

在计算机领域，堆栈是一个不容忽视的概念，[堆](http://baike.baidu.com/view/249120.htm)[栈](http://baike.baidu.com/subview/38877/12246229.htm)是 两种数据结构。堆栈都是一种数据项按序排列的数据结构，只能在一端(称为栈顶(top))对数据项进行插入和删除。在单片机应用中，堆栈是个特殊的存储 区，主要功能是暂时存放数据和地址，通常用来保护断点和现场。要点：堆，队列优先,先进先出（FIFO—first in first out）。栈，先进后出(FILO—First-In/Last-Out)。

   一般情况下，如果有人把堆栈合起来说，那它的意思是栈，可不是堆。。。。。

  他们之间区别，对于初学者来说总是混淆不清，笔者自己就是，因而觉得还是有必要对此稍加整理，最起码自己在心里也有个概念。能力问题，写得不好，对此问题，各IT类的BBS，blog都有写得很好的，大家可以学习。百度百科上对堆和栈进行了对比分析：

**堆栈空间分配**

栈（操作系统）：由操作系统自动分配释放 ，存放函数的[参数值](http://baike.baidu.com/view/1703315.htm)，[局部变量](http://baike.baidu.com/view/552847.htm)的值等。其操作方式类似于数据结构中的栈。

堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表

**堆栈缓存方式**

栈使用的是[一级缓存](http://baike.baidu.com/view/16882.htm)， 他们通常都是被调用时处于存储空间中，调用完毕立即释放。

堆则是存放在[二级缓存](http://baike.baidu.com/view/27650.htm)中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

**堆栈数据结构区别**

堆（数据结构）：堆可以被看成是一棵树，如：堆排序。

栈（数据结构）：一种先进后出的数据结构。

例如：顺序栈AStack的类定义

template < class T >

class AStack {

private:

int size ; // 数组的规模

T \* stackArray ; // 存放堆栈元素的数组

int top ; // 栈顶所在数组元素的下标

public:

AStack ( int MaxStackSize ) // 构造函数

{ size = MaxStackSize ; stackArray = new T [MaxStackSize] ; top = -1 ; }

~AStack ( ) { delete [ ] stackArray ; } // 析构函数

bool Push ( const T& item ) ; // 向栈顶压入一个元素

bool Pop ( T & item ) ; // 从栈顶弹出一个元素

bool Peek ( T & item ) const ; // 存取栈顶元素

int IsEmpty ( void ) const { return top = = -1 ; }

// 检测栈是否为空

int IsFull ( void ) const { return top   size-1 ; }

// 检测栈是否为满

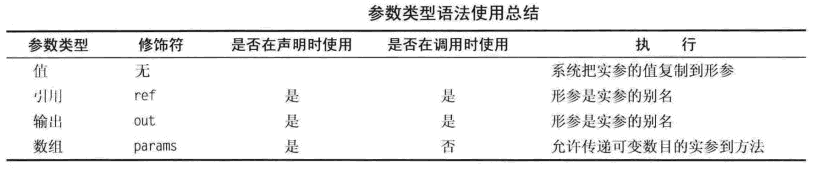
void clear ( void ) { top-1 ; } // 清空栈

} ;

    首先，我们举一个例子：  
    void f() { int\* p=new int[5]; }   
    这 条短短的一句话就包含了堆与栈，看到new，我们首先就应该想到，我们分配了一块堆内存，那么指针p呢？他分配的是一块栈内存，所以这句话的意思就是：在 栈内存中存放了一个指向一块堆内存的指针p。在程序会先确定在堆中分配内存的大小，然后调用operator new分配内存，然后返回这块内存的首地 址，放入栈中，他在VC6下的汇编代码如下：  
    00401028   push        14h  
    0040102A   call        operator new (00401060)  
    0040102F   add         esp,4  
    00401032   mov         dword ptr [ebp-8],eax  
    00401035   mov         eax,dword ptr [ebp-8]  
    00401038   mov         dword ptr [ebp-4],eax  
    这里，我们为了简单并没有释放内存，那么该怎么去释放呢？是delete p么？澳，错了，应该是delete []p，这是为了告诉编译器：我删除的是一个数组，VC6就会根据相应的Cookie信息去进行释放内存的工作。  
    好了，我们回到我们的主题：堆和栈究竟有什么区别？ 笔者为此做出鄙陋总结  
    主要的区别由以下几点：  
    1、管理方式不同；  
    2、空间大小不同；  
    3、能否产生碎片不同；  
    4、生长方向不同；  
    5、分配方式不同；  
    6、分配效率不同；  
    管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。  
    空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M（好像是，记不清楚了）。当然，我们可以修改：      
    打开工程，依次操作菜单如下：Project->Setting->Link，在Category 中选中Output，然后在Reserve中设定堆栈的最大值和commit。  
注意：reserve最小值为4Byte；commit是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。  
    碎 片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因 为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可 以参考数据结构，这里我们就不再一一讨论了。  
    生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。  
    分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。  
    分 配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较 高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内 存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到 足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。  
    从这里我们可以看到，堆和栈相比，由于大量new/delete的使用，容 易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广 泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用 堆。  
    虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。  
    无论是堆还是 栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程 序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候debug可是相当困难的：）

## 6.0 C#方法中参数ref和out的解析

**一、C#方法中参数类型**

有4种参数类型，有时候很难记住它们的不同特征，下图对它们做一个总结，使之更容易比较和对照。 

**二、C#方法中的参数**

**1、值参数**

   使用值参数，通过复制实参的值到形参的方式把数据传递到方法。方法调用时，系统做如下操作：

    · 在栈中为形参分配空间

    · 复制实参到形参

**注：栈（先进后出）**是编译期间就分配好的内存空间，因此你的代码中必须就栈的大小有明确的定义；

**堆（队列优先,先进先出）**是程序运行期间动态分配的内存空间，你可以根据程序的运行情况确定要分配的堆内存的大小。

/// <summary>

/// 声明方法

/// </summary>

/// <param name="value">参数</param>

/// <returns>返回值</returns>

static float FuncData(float value )

{

float i = 1.5F;

float j = 2.5F;

float value1 = FuncData(i); //方法调用

float value2 = FuncData(i+j); //方法调用（一个值参数的实参不一定是变量。它可以是任何能计算成相应数据类型的表达式）

return value1 + value2;

}

**2、引用参数**

  · 使用引用参数时，必须在方法的声明和调用中都使用ref修饰符

  · 实参必须是变量，在用作实参前必须被赋值，如果是引用类型变量，可以赋值为一个引用或者null值

/// <summary>

/// 声明方法

/// </summary>

/// <param name="value">参数</param>

/// <returns>返回值</returns>

static void FuncData(float value )

{

//方法调用

int temp=0; //实参变量

FuncData(ref temp); //包含修饰符ref

//FuncData(ref temp+1); //错误，必须使用变量

}

/// <summary>

/// 方法声明

/// </summary>

/// <param name="value">参数</param>

/// <returns>返回值</returns>

static int FuncData(ref int value)

{

return value = 100;

}

**3、输出参数**

  · 必须在声明和调用中都使用修饰符。输出参数的修饰符是out不是ref

  · 和引用参数相似，实参必须是变量，而不能是其他类型的表达式。（因为方法需要内存位置保存返回值）

/// <summary>

/// 声明方法

/// </summary>

/// <param name="value">参数</param>

/// <returns>返回值</returns>

static void FuncData(float value )

{

//方法调用

int temp=0; //实参变量

FuncData(out temp); //包含修饰符out

//FuncData(out temp+1); //错误，必须使用变量

}

/// <summary>

/// 方法声明

/// </summary>

/// <param name="value">参数</param>

/// <returns>返回值</returns>

static int FuncData(out int value)

{

return value = 100;

}

**与引用参数不同，输出参数有以下要求：**

  · 在方法内部，输出参数在被读取之前必须被赋值。（意味着参数的初始值是无关的，而且没有必要在方法调用之前为实参赋值）

  · 在方法返回之前，方法内部的任何贯穿的可能路径都必须为所有输出参数进行一次赋值。

class MyClass {

public int val = 100;

}

class Program

{

/// <summary>

/// 方法声明

/// </summary>

static void FuncData(out MyClass myclass,out int temp)

{

myclass = new MyClass(); //创建一个类变量

myclass.val = 50; //赋值字段

temp=20; //赋值int参数

}

/// <summary>

/// 方法调用

/// </summary>

static void Main()

{

MyClass my = null;

int temp;

FuncData(out my,out temp);//调用方法

}

static void FuncTest(out int value)

{

var test = value+1;//错误，在方法赋值之前无法读取输出变量

}

}

**4、参数数组**

  ·在一个参数列表中只能有一个参数数组

  ·如果有，它必须是列表中的最后一个

**声明一个参数数组必须做的事**

  ·在数据类型前使用Params修饰符

  ·在数据类型后放置一组空的方括号

/// <summary>

/// 声明方法

/// </summary>

/// <param name="Array">参数</param>

static void FuncTest(params int[] Array)

{

...

}

  · 数组是一组整齐的相同类型的数据项

  · 数组使用一个数字索引进行访问

  · 数组是一个引用类型，因此它的所有数据项都保存在堆中

**三、C#中方法参数ref和out区别**

   1、使用ref型参数时，传入的参数必须先被初始化。对out而言，必须在方法中对其完成初始化

   2、使用ref和out时，在方法的参数和执行方法时，都要加Ref或Out关键字，以满足匹配。

   3、out适合用在需要retrun多个返回值的地方，而ref则用在需要被调用的方法修改调用者的引用的时候。

class TestApp

{

static void outTest(out int x, out int y)

{

//离函数前必须xy赋值否则报错

//y = x;

//上面行报错使用outxy都清空需要重新赋值即使调用函数前赋值行

x = 1;

y = 2;

}

static void refTest(ref int x, ref int y)

{

x = 1;

y = x;

}

public static void Main()

{

//正确 （out test）

int a, b;

//out使用前变量赋值

outTest(out a, out b);

Console.WriteLine("a={0};b={1}", a, b);

//在使用out关键字时，不需要在此处初始化，初始化也不会影响到方法内部的值，所以你初始化没用

int c = 11, d = 22;

outTest(out c, out d);

Console.WriteLine("c={0};d={1}", c, d);

//错误 （ref test）

int m, n;

refTest(ref m, ref n);

//上面行错ref使用前变量必须赋值

//正确（ref test）

int o = 11, p = 22;

refTest(ref o, ref p);

Console.WriteLine("o={0};p={1}", o, p);

}

}

**四、总结**

  1、**ref的使用：**使用ref进行参数的传递时，该参数在创建时，必须设置其初始值，且ref侧重于修改；

  2、**out的使用：**采用out参数传递时，该参数在创建时，可以不设置初始值，但是在方法中必须初始化，out侧重于输出；

**注释：**当希望方法返回多个值时，可以用out，并且一个方法中的参数可以有一个或多个out参数；使用out参数，必须将参数作为out参数显式传递到方法中，但是out 参数的值不会被传递到 方法中，且属性不是变量，不能作为 out 参数传递。

**ref是有进有出，而out是只出不进。**

## 7.0 [C# Array类的浅复制Clone()与Copy()的区别](http://blog.csdn.net/wangshubo1989/article/details/47002667)

**1 Array.Clone方法**

命名空间：System

程序集：mscorlib

语法：

public Object Clone()

Array的浅表副本仅复制Array的元素，无论他们是引用类型还是值类型，但是不负责这些引用所引用的对象。

新Array中的引用与原始Array的引用指向相同的对象。

例：

int[] intArray1 = {1, 2};

int [] intArray2 = (int [])intArray1.Clone();

这里需要说明的是，需要使用强制类型转换，原因在于Clone()返回的类型为Object

**2 Array.Copy方法**

命名空间：System

程序集：mscorlib

Copy有几个重载函数：

//从第一个元素开始复制Array中的一系列元素，将它们粘贴到另一Array中（从第一个元素开始）。长度为32位整数

public static void Copy(Array sourceArray, Array destinationArray, int length)

//从第一个元素开始复制Array中的一系列元素，将它们粘贴到另一Array中（从第一个元素开始）。长度为64位整数

public static void Copy(Array sourceArray, Array destinationArray, long length)

//从指定的源索引开始，复制Array中的一系列元素，将它们粘贴到另一Array中（从指定的目标索引开始）。长度和索引指定为32位整数

public static void Copy(Array sourceArray, int sourceIndex，Array destinationArray, int destinationIndex，int length)

//从指定的源索引开始，复制Array中的一系列元素，将它们粘贴到另一Array中（从指定的目标索引开始）。长度和索引指定为64位整数

public static void Copy(Array sourceArray, long sourceIndex，Array destinationArray, long destinationIndex，long length)

例： Array myIntArray=Array.CreateInstance( typeof(System.Int32), 5 );

for ( int i = myIntArray.GetLowerBound(0); i <= myIntArray.GetUpperBound(0); i++ )

{myIntArray.SetValue( i+1, i );}

Array myObjArray = Array.CreateInstance( typeof(System.Object), 5 );

for ( int i = myObjArray.GetLowerBound(0); i <= myObjArray.GetUpperBound(0); i++ )

{myObjArray.SetValue( i+26, i );}

// Copies the first element from the Int32 array to the Object array.

Array.Copy( myIntArray, myIntArray.GetLowerBound(0), myObjArray, myObjArray.GetLowerBound(0), 1 );

// Copies the last two elements from the Object array to the Int32 array.

Array.Copy( myObjArray, myObjArray.GetUpperBound(0) - 1, myIntArray, myIntArray.GetUpperBound(0) - 1, 2 );

**区别：**

Clone()返回值是Object，Copy返回值为void

Clone()是非静态方法，Copy为静态方法。

Clone()会创建一个新数组，Copy方法必须传递阶数相同且有足够元素的已有数组。

## 8.0 [C#命名空间大全详细教程](http://www.cnblogs.com/viky0724/p/5267909.html)

C#命名空间大全详细教程  
  
System 命名空间包含了定义数据类型、事件和事件处理程序等基本类；  
  
System.Data 命名空间包含了提供数据访问功能的命名空间和类；  
  
System.IO 命名空间包含了数据流读写相关功能的类；  
  
System.Windows.Forms 命名空间包含了Windows 窗体开发所需要的工具和窗体的类；  
  
System.Web 命名空间包含了所有网站开发相关的命名空间和类。  
  
Microsoft.Csharp  
包含支持用C#语言进行编译的代码  
Microsoft.Jscript  
包含支持用Jscript语言进行编译和代码生成的Jscript运行库和类  
Microsoft.VisualBasic  
包含Visual Basic.NET运行库。此运行库于Visual Basic.NET语言一起使用。此命名空间包含支持Visual Basic.NET语言进行编译和代码生成类。  
Microsoft.Win32  
提供两种类型的类：处理由操作系统引发的事件的类和对系统注册表进行操作的类。  
System  
最重要的类，包含用于定义常用值和引用数据类型、事件和事件处理程序、接口、属性、和处理异常的基础类和基类。  
System.CodeDom  
包含用于表示源代码文档的元素和结构的类  
System.CodeDom.Compiler  
包含源代码模型的结构，管理源代码所生成的编译的类。  
System.Collections  
包含定义各种对象集合（如列表、队列、位数组、散列表和词典）的接口和类。  
System.Collections.Specialized  
包含中用的强类型集合；例如，链接表词典、位向量以及值包含字符串的集合。  
System.CommponentModel  
提供用于实现组件和空间的运行时和设计时行为的类。此命名空间包括用于属性和类型转换器的事件、数据源绑定和组件授权的基类和接口。  
System.ComponentModel.Design  
使开发人员可以生成自定义用户控件，并将这些控件包括在设计时环境中以便与供应商空间一起使用。  
System.ComponentModel.Design.Serialization  
提供设计器所进行的组件序列化支持。此命名空间中的类可用于提哦能够自定义序列化程序、管理特定类型的序列化、管理设计器加载和设计器序列化，以及优化设计器重新加载。  
www.51rgb.com  
System.Configuration  
提供以变成方式访问.NET空间配置和处理配置文件（.config文件）中的错误的类和接口。  
System.Configuration.Assemblies  
包含用于配置程序集的类。  
System.Configuration.Install  
提供为组件编写自定义安装程序的类。Installer类是.NET框架中所有自定义安装程序的基类。  
System.Data  
基本上由构成ADO.NET结构的类组成。使用ADO.NET结构可以生成用于有效管理多个数据源中的数据组件。在断开连接的方案（如Internet）中，ADO.NET提供可以在多城系统中请求、更新和协调数据的工具。ADO.NET结构也可以在客户端应用程序或ASP.NET创建的HTML页中实现。  
System.Data.Common  
包含由.NET数据提供程序共享的类。.NET数据提供程序描述用于在托管空间中访问数据源的类的集合。  
System.Data.OleDb  
封装OLE DB.NET数据提供程序。.NET数据提供程序描述用于在托管空间中访问数据源的类的集合  
System.Data.SqlClient  
封装SQL Server.NET数据提供程序。.NET数据提供描述用于在托管空间中访问数据源的类的集合。  
System.Data.SqlTypes  
提供用于SQL Sserver中本机数据类型的类。这些类提供其他数据类型更安全、更快速的替代物。使用此命名空间中的类有助于防止在可能发生精度损失的情况中出现的类行转换错误。  
System.Diagnostics  
提供允许与系统进程、事件日志和性能计数器进行交互的类。此命名空间还提供可以调试应用程序和跟踪代码执行的类。  
System.Diagnostics.SymbolStore  
提供允许读取和写入调适符号信息的类。面向.NET框架的编译器可以将调试符号信息存储到程序员的数据库（PDB）文件中。调试器和代码分析器工具可以在运行时读取调试符号信息。  
System.DirectioyServices  
提供从托管代码轻松访问Active Directory的方法。  
System.Drawing  
提供对GDI+基本形功能的访问。System.Drawing.Drawing2D,System.Drawing.Imaging和System.Drawing.Text命名空间提供了更高级的功能  
System.Drawing.Design  
包含扩展设计时用户界面（UI）逻辑和绘制的类。可以进一步扩展此设计时功能，以创建自定义工具箱项、类型特定的值编辑器（可编辑和以图形方式表示所支持的类型的值）或类型转换器（可在特定类型之间转换值）。  
System.Drawing.Drawing2D  
提供高级的二维和向量图形功能。此命名空间包括渐变笔、Matrix类（用于定义几何转换）和GraphicsPath类。  
System.Drawing.Imaging  
提供高级的GDI+图象处理功能  
System.Drawing.Printing  
提供与打印相关的服务。  
System.Drawing.Text  
提供高级的GDI+版式功能。此命名空间中的类使用户可以创建和使用字体集合。  
System.EnterpriseServices  
为企业级应用程序提供重要的基础结构。COM+为企业级环境中部署的组件编程模型提供服务结构。此命名空间为.NET框架对象提供对COM+服务的访问，使.NET框架对象更适用于企业级应用程序。  
System.EnterpriseServices.CompensatingResourceManager  
提供在托管代码中使用补偿资源管理器（CRM）的类。CRM是由COM+提供的一项服务，使用户可以在Microsoft分布式事务处理协调器（DTC）事务中包括非事务对象。虽然CRM不提供完整资源管理器的功能，但它们却通过恢复日志提供事务性和持久性。  
System.Globalization  
包含定义区域性相关信息的类，这些信息包括语言、国家/地区、正在使用的日历、日期的格式模式、货币、数字以及字符串的排序顺序。  
System.IO  
包含允许对数据流和文件进行同步和异步读写的类型。  
System.IO.IsolatedStorage  
包含允许船舰和使用独立存储区的类型。通过使用这些存储区，可以读写信任度较低的代码无法访问的数据，防止公开可保存在文件系统其他位置的敏感信息。数据存储在独立于当前用户和代码所在的程序集的数据仓中。  
System.ManageMent  
提供对一组丰富的管理信息和管理事件（有关符合Windows管理规范（WMI）基础机构的系统、设备和应用程序的）的访问。  
System.Management.Instrumentation  
提供在规范应用程序管理并通过WMI向潜在用户公开管理信息和事件时必须的类。这样，Microsoft Application Center 或 Microsoft Operations Manager 等用户者就可以轻松地管理您的应用程序。而管理员脚本或其他应用程序（托管应用程序和非过关应用程序）也可以监视和配置您的应用程序。  
System.Messaging  
提供用户连接、监视和管理网络上的消息列队以及发送、和接受或查看消息的类。  
System.Net  
为当前网络采用的多种协议提供简单的编程接口。WebRequest和WebResponse类构成所谓的可插接式协议的基础，该协议式一种网络服务的实现，它使您可以开发使用Internet资源的应用程序，而不必考虑各个协议的具体细节。  
System.Net.Sockets  
为需要严格控制网络访问的开发人员提供Windows套接字（Winsock）接口的托管实现。  
System.Refection  
包含提供已加载类型、方法和字段的托管视图的类和接口，并具由动态创建和调用类型的能力。  
System.Reflection.Emit  
包含允许编译器或工具发出元数据和Micosoft中间语言（MSIL）并在磁盘上生成PE（可选）的类。这些类的主要客户端是脚本引擎和编译器。  
System.Runtime.CompilerServices  
为使用托管代码的编译编写器提供功能，以影响在公共语言运行库运行时行为的元数据中制定的属性。此命名空间中的类值用于编译器编译器。  
System.Runtime.InteropServices  
提供用于通过.NET访问COM对象和本机API的类的集合。此命名空间中的类型分为一下功能区：属性、异常、COM类型的托管定义、包装、类型转换器和Marshal类。  
System.Runtime.InteropServices.Expando  
包含Iexpando接口，此接口允许通过添加活移除对象的成员来修改对象。  
System.Runtime.Remoting  
提供允许开发人员创建和配置分布式应用程序的接口。  
System.Runtime.Remoting.Activation  
提供支持服务器和客户端远程对象激活的类和对象。  
System.Runtime.Remoting.Channels  
包含支持和处理信道和信道接收器的类，这些信道和信道接受器在客户端远程对象调用方法时用作传输媒介。  
System.Runtime.Remoting.Channels.Http  
包含使用HTTP协议与远程位置之间相互传输消息和对象的信道。默认情况下，HTTP信道以SOAP格式对对象和方法调用进行编码以便传输，但在信道的配置属性中也可以指定他编码和解码格式化程序接收器。  
System.Runtime.Remoting.Channels.Tcp  
包含使用TCP协议与远程位置之间相互传输消息和对象的信道。默认情况下，TCP信道以二进制格式对对象和方法调用进行编码以便传输，但在信道的配置属性中也可以指定其编码和解码格式化程序接收器。  
System.Runtime.Remoting.Contexts  
包含定义所有对象所驻留的上下文的对象。上下文时个有序的属性序列，用与定义其中的对象所处的环境。上下文时在对象的激活过程中创建的，这些对象被配置为要求某些自定服务，如同步，事务，实时（JIT）激活、安全性。多个对象可以存留在以个上下文内。  
System.Runtimg.Remoting.Lifetime  
包含管理远程对象生存期的类。传统上，分布式垃圾回收功能使用引用计数和Ping来控制对象的生存期、这种机制在每一项服务只有较少的客户端时可以正常工作，但是当每一项服务有几千个客户端时就不能正常工作了。远程处理生存期服务将每一项服务与一个租约关联，当租约到期时，就会删除该服务。生存期服务可以起到传统的分布式垃圾回收器的作用，并在每一项服务的客户端数量增加时很好的调整。  
System.Reuntime.Remoting.Metadata  
包含可用于为对象和字段定义SOAP的生成和处理的类和属性。此命名空间中的类可用于指示SOAPAction、类型输出、XML元素名和SML命名空间URL方法。  
System.Runtime.Remoting.Metadata.W3cXsd2001  
包含有WWW联合会（W3C）在2001年定义的XML架构定义（XSD）。W3C中的“XML Schema Part2:Data type”（XML架构第二布冯：数据类型）规范确定了各种数据类型的格式和行为。此命名空间包含符合W3C规范的数据类型的包装类。所有日期和时间类型都符合ISO标准规范。  
System.Runtime.Remoting.MetadataServices  
包含利用Soapsuds.exe命令行工具和用户代码在元数据和远程处理基础结构的XML架构之间相互转的类。  
System.Runtime.Remoting.Proxies  
包含控制和提供代理功能的类。代理时作为远程对象映像的本地对象。代理使客户端可以跨远程处理边界访问对象。  
System.Runtime.Remoting.Services  
包含为.NET框架提供功能的服务类。  
System.Runtime.Serialization  
包含可以用于序列化和反序列化对象的类。序列化是对象或对象图转换为线性的字节序列以存储或传输到其他位置的过程。反序列化是接受存储的信息并用这些信息重新建立对象的过程。  
System.Runtime.Serialization.Formatters  
提供由序列化格式化程序使用的通用枚举、接口和类。  
System.Runtime.Serialization.Formatters.Binary  
包含可以用二进制格式序列化和反序列化对象的BinaryFormatter类。  
System.Runtime.Serialization.Formatters.Soap  
包含可以用SOAP格式序列化和反序列化对象的SoapForamtter类。  
System.Security  
提供公共语言运行库安全系统的基础结构，包括权限的基类。  
System.Security.Cryptography  
提供加密服务，包括数据的安全编码和解码，以及其他许多操作，如散列处理、随即生成和消息身份验证。  
System.Security.Cryptography.X509Certificates  
包含Authenticode X.509 v.3证书的公共语言运行库实现。此证书用唯一明确标识证书持有者的私钥签名。  
System.Securiyt.Cryptography.XML  
包含在.NET框架安全系统中供独占使用的XML模型。此XML模型不用于常规用途。此模型允许对XML对象进行数字签名。  
System.Security.Permaissions  
定义根据策略控制操作和资源访问的类。  
System.Security.Policy  
包含代码组、成员条件和证据。这三种类性的类用于创建由公共语言运行库安全策略系统应用的规则。证据类是安全策略的输入，成员条件是开关；二者共同创建策略语句并确定授予的权限集。策略级别代码组是策略层次的结构。代码组是规则的封装并且在策略级别中分层排列。  
System.Security.Principal  
定义表示运行代码的安全上下文的用户对象。  
System.ServiceProcess  
提供使您可以实现、安装和控制Windows服务应用程序的类。服务是不需要用户界面、长期运行的可执行文件。实现服务的过程包括：从ServiceBase类继承，定义在传入启动、停止、暂定和继续命令时处理的特定行为，以及定义当前系统关闭时所采取的自定义行为和操作。  
System.Text  
包含表示ASCII、Unicode、UTF-7和UTF-8字符编码的类，用于在字符块和字节块之间相互转换的抽象基类，以及不需要创建字符串的中间实例就可以操作和格式化字符串对象的帮助器类。  
System.Text.RegularExpressions  
提供对.NET框架正则表达式引擎的访问的类。此命名空间提供可在Microsoft.NET框架上运行的任何平台或语言中使用的正则表达式功能。  
System.Threading  
提供支持多线程编程的类和接口。此命名空间包括管理线程组的ThreadPool类、允许在指定的一段时间后调用委托的Time类，以及用于同步互斥的线程的Mutex类。此命名空间还提供用于线程安排、等待通知和死锁的类。  
System.Timers  
提供允许以指定的间隔引发时间的Timer组件。  
System.Web  
提供支持浏览器/服务器通信的类和接口。此命名空间包括提供有关当前HTTP请求的大量信息的HTTPResponse类、管理HTTP到客户端的输出的HTTPResponse类，以及提供对服务器端实用工具和进程访问的HTTPServerUitility对象。System.Web还包括用于Cookie操作、文件传输、异常信息和输出缓存控制的类。  
System.Web.Caching  
提供用于在服务器上缓存常用资源的类。这些资源包括ASP.NET页、Web服务和用户空间。另外，缓存词典可以供您存储常用资源，如散列表和其他数据结构。  
System.Web.Configuration  
包含用于设置ASP.NET配置的类。  
System.Web.Hosting  
提供对Microsoft Internet Information Server(IIS)外部的托管应用程序承载ASP.NET应用程序功能。  
System.Web.Mail  
包含使用CDOSYS消息组建构造和发送消息的类。邮件消息通过Microsoft Windows 2000中内置的SMTP邮件服务或任意的SMTP服务器发送。此命名空间中的类可从ASP.NET或任何托管应用程序中使用。  
System.Web.Security  
包含用于在Web服务器应用程序中实现ASP.NET安全的类。  
System.Web.Services  
包含可以生成和使用Web服务的类。Web服务式驻留在Web服务器上并通过标准Internet协议公开的可编程实体（XML、SOAP）。  
System.Web.Services.Configuration  
由一些类组成，这些类配置用ASP.NET创建的XML Web services的运行方式。  
System.Web.Service.Description  
由一些类组成，这些类使您能够使用Web服务描述语言（WSDL）公开描述SML Web services。此命名空间中的每一个类对应于WSDL规范中的一个特定元素，并且类层次机构对应于有效的WSDL文档的XML结构。  
System.WebServices.Discovery  
由一些类组成，这些类允许XML Web Services客户端通过称为“XML Web services发现”的进程来定位Web服务其上可用的XML Web services。  
System.Web.Services.Protocols  
由一些类组成，这些类定义在通信期间通过网络在XML Web services 客户端和用ASP.NET创建的XML Web services之间传输数据的协议。  
System.Web.SessionState  
提供支持在服务器上存储特定于Web应用程序中的单个客户端的数据的类和接口。会话转台数据用于向客户端提供与应用程序的持久连接的外观。状态信息可以存储在本地进程内存中，或者，对于网络场配置来说，可以使用ASP.NET状态服务或SQL Server数据库状态信息存储进程外。  
System.Web.UI  
提供创建以Web页上用户界面形式出现在Web应用程序中的空间和页的类和接口。此命名空间包括Control类，该类为所有空间（不论使HTML空间、Web空间还是用户控件）提供一组通用功能。它还包括Page空间，每当对Web应用程序中的页发出请求时，都会自动生成此空间。另外还提供了一些类，这些类提供Web窗体服务器空间数据绑定功能、保存给定控件或页的视图状态的能力，以及对可编程控件和文本空间都适用的分析通能。  
System.Web.UI.Design  
包含用于扩展Web用户界面设计时的支持类。  
System.Web.UI.Design.WebControls  
包含用于扩展Web服务器控件设计时所支持的类。  
System.Web.UI.HtmlControls  
提供可以在Web页上创建HTML服务器控件的类。HTML服务器控件在服务器上运行，并直接映射到所有浏览器支持的标准HTML标记。这使您能够以编程方式控制Web页上的HTML元素。  
System.Web.UI.WebControls  
包含可以在Web服务器控件的类。Web控件在服务器上运行，并包括窗体控件（如按钮和文本框）以及特出用途的控件（如日历）。这使您能构以编程方式控制Web页上的这些元素。Web控件比HTML控件更抽象。它们的对象模型不一定反映HTML语法。  
System.Windows.Forms  
包含用于创建基于Windows的应用程序的类，这些应用程序可以充分利用Microsoft Windows操作系统中的丰富用户界面功能。  
System.Windows.Forms.Design  
包含可用于扩展Windows窗体设计时支持的类。  
System.Xml  
提供基于标准的XML架构（XSD）支持。  
System.Xml.Serialization  
包含用于将对象序列化为XML格式的文档或流的类。  
System.Xml.Xpath  
包含Xpath分析器和计算引擎。它支持W3C XML路径语言（XPath）1.0版建议。  
System.Xml.Xsl  
提供可扩展样式表转换（XSLT）支持。它支持W3C XSL转换（XSLT）1.0版建议。System 命名空间包含了定义数据类型、事件和事件处理程序等基本类；  
  
System.Data 命名空间包含了提供数据访问功能的命名空间和类；  
  
System.IO 命名空间包含了数据流读写相关功能的类；  
  
System.Windows.Forms 命名空间包含了Windows 窗体开发所需要的工具和窗体的类；  
  
System.Web 命名空间包含了所有网站开发相关的命名空间和类。  
  
Microsoft.Csharp  
包含支持用C#语言进行编译的代码  
Microsoft.Jscript  
包含支持用Jscript语言进行编译和代码生成的Jscript运行库和类  
Microsoft.VisualBasic  
包含Visual Basic.NET运行库。此运行库于Visual Basic.NET语言一起使用。此命名空间包含支持Visual Basic.NET语言进行编译和代码生成类。  
Microsoft.Win32  
提供两种类型的类：处理由操作系统引发的事件的类和对系统注册表进行操作的类。  
System  
最重要的类，包含用于定义常用值和引用数据类型、事件和事件处理程序、接口、属性、和处理异常的基础类和基类。  
System.CodeDom  
包含用于表示源代码文档的元素和结构的类  
System.CodeDom.Compiler  
包含源代码模型的结构，管理源代码所生成的编译的类。  
System.Collections  
包含定义各种对象集合（如列表、队列、位数组、散列表和词典）的接口和类。  
System.Collections.Specialized  
包含中用的强类型集合；例如，链接表词典、位向量以及值包含字符串的集合。  
System.CommponentModel  
提供用于实现组件和空间的运行时和设计时行为的类。此命名空间包括用于属性和类型转换器的事件、数据源绑定和组件授权的基类和接口。  
System.ComponentModel.Design  
使开发人员可以生成自定义用户控件，并将这些控件包括在设计时环境中以便与供应商空间一起使用。  
System.ComponentModel.Design.Serialization  
提供设计器所进行的组件序列化支持。此命名空间中的类可用于提哦能够自定义序列化程序、管理特定类型的序列化、管理设计器加载和设计器序列化，以及优化设计器重新加载。  
System.Configuration  
提供以变成方式访问.NET空间配置和处理配置文件（.config文件）中的错误的类和接口。  
System.Configuration.Assemblies  
包含用于配置程序集的类。  
System.Configuration.Install  
提供为组件编写自定义安装程序的类。Installer类是.NET框架中所有自定义安装程序的基类。  
System.Data  
基本上由构成ADO.NET结构的类组成。使用ADO.NET结构可以生成用于有效管理多个数据源中的数据组件。在断开连接的方案（如Internet）中，ADO.NET提供可以在多城系统中请求、更新和协调数据的工具。ADO.NET结构也可以在客户端应用程序或ASP.NET创建的HTML页中实现。  
System.Data.Common  
包含由.NET数据提供程序共享的类。.NET数据提供程序描述用于在托管空间中访问数据源的类的集合。  
System.Data.OleDb  
封装OLE DB.NET数据提供程序。.NET数据提供程序描述用于在托管空间中访问数据源的类的集合  
System.Data.SqlClient  
封装SQL Server.NET数据提供程序。.NET数据提供描述用于在托管空间中访问数据源的类的集合。  
System.Data.SqlTypes  
提供用于SQL Sserver中本机数据类型的类。这些类提供其他数据类型更安全、更快速的替代物。使用此命名空间中的类有助于防止在可能发生精度损失的情况中出现的类行转换错误。  
System.Diagnostics  
提供允许与系统进程、事件日志和性能计数器进行交互的类。此命名空间还提供可以调试应用程序和跟踪代码执行的类。  
System.Diagnostics.SymbolStore  
提供允许读取和写入调适符号信息的类。面向.NET框架的编译器可以将调试符号信息存储到程序员的数据库（PDB）文件中。调试器和代码分析器工具可以在运行时读取调试符号信息。  
System.DirectioyServices  
提供从托管代码轻松访问Active Directory的方法。  
System.Drawing  
提供对GDI+基本形功能的访问。System.Drawing.Drawing2D,System.Drawing.Imaging和System.Drawing.Text命名空间提供了更高级的功能  
System.Drawing.Design  
包含扩展设计时用户界面（UI）逻辑和绘制的类。可以进一步扩展此设计时功能，以创建自定义工具箱项、类型特定的值编辑器（可编辑和以图形方式表示所支持的类型的值）或类型转换器（可在特定类型之间转换值）。  
System.Drawing.Drawing2D  
提供高级的二维和向量图形功能。此命名空间包括渐变笔、Matrix类（用于定义几何转换）和GraphicsPath类。  
System.Drawing.Imaging  
提供高级的GDI+图象处理功能  
System.Drawing.Printing  
提供与打印相关的服务。  
System.Drawing.Text  
提供高级的GDI+版式功能。此命名空间中的类使用户可以创建和使用字体集合。  
System.EnterpriseServices  
为企业级应用程序提供重要的基础结构。COM+为企业级环境中部署的组件编程模型提供服务结构。此命名空间为.NET框架对象提供对COM+服务的访问，使.NET框架对象更适用于企业级应用程序。  
System.EnterpriseServices.CompensatingResourceManager  
提供在托管代码中使用补偿资源管理器（CRM）的类。CRM是由COM+提供的一项服务，使用户可以在Microsoft分布式事务处理协调器（DTC）事务中包括非事务对象。虽然CRM不提供完整资源管理器的功能，但它们却通过恢复日志提供事务性和持久性。  
System.Globalization  
包含定义区域性相关信息的类，这些信息包括语言、国家/地区、正在使用的日历、日期的格式模式、货币、数字以及字符串的排序顺序。  
System.IO  
包含允许对数据流和文件进行同步和异步读写的类型。  
System.IO.IsolatedStorage  
包含允许船舰和使用独立存储区的类型。通过使用这些存储区，可以读写信任度较低的代码无法访问的数据，防止公开可保存在文件系统其他位置的敏感信息。数据存储在独立于当前用户和代码所在的程序集的数据仓中。  
System.ManageMent  
提供对一组丰富的管理信息和管理事件（有关符合Windows管理规范（WMI）基础机构的系统、设备和应用程序的）的访问。  
System.Management.Instrumentation  
提供在规范应用程序管理并通过WMI向潜在用户公开管理信息和事件时必须的类。这样，Microsoft Application Center 或 Microsoft Operations Manager 等用户者就可以轻松地管理您的应用程序。而管理员脚本或其他应用程序（托管应用程序和非过关应用程序）也可以监视和配置您的应用程序。  
System.Messaging  
提供用户连接、监视和管理网络上的消息列队以及发送、和接受或查看消息的类。  
System.Net  
为当前网络采用的多种协议提供简单的编程接口。WebRequest和WebResponse类构成所谓的可插接式协议的基础，该协议式一种网络服务的实现，它使您可以开发使用Internet资源的应用程序，而不必考虑各个协议的具体细节。  
System.Net.Sockets  
为需要严格控制网络访问的开发人员提供Windows套接字（Winsock）接口的托管实现。  
System.Refection  
包含提供已加载类型、方法和字段的托管视图的类和接口，并具由动态创建和调用类型的能力。  
System.Reflection.Emit  
包含允许编译器或工具发出元数据和Micosoft中间语言（MSIL）并在磁盘上生成PE（可选）的类。这些类的主要客户端是脚本引擎和编译器。  
System.Runtime.CompilerServices  
为使用托管代码的编译编写器提供功能，以影响在公共语言运行库运行时行为的元数据中制定的属性。此命名空间中的类值用于编译器编译器。  
System.Runtime.InteropServices  
提供用于通过.NET访问COM对象和本机API的类的集合。此命名空间中的类型分为一下功能区：属性、异常、COM类型的托管定义、包装、类型转换器和Marshal类。  
System.Runtime.InteropServices.Expando  
包含Iexpando接口，此接口允许通过添加活移除对象的成员来修改对象。  
System.Runtime.Remoting  
提供允许开发人员创建和配置分布式应用程序的接口。  
System.Runtime.Remoting.Activation  
提供支持服务器和客户端远程对象激活的类和对象。  
System.Runtime.Remoting.Channels  
包含支持和处理信道和信道接收器的类，这些信道和信道接受器在客户端远程对象调用方法时用作传输媒介。  
System.Runtime.Remoting.Channels.Http  
包含使用HTTP协议与远程位置之间相互传输消息和对象的信道。默认情况下，HTTP信道以SOAP格式对对象和方法调用进行编码以便传输，但在信道的配置属性中也可以指定他编码和解码格式化程序接收器。  
System.Runtime.Remoting.Channels.Tcp  
包含使用TCP协议与远程位置之间相互传输消息和对象的信道。默认情况下，TCP信道以二进制格式对对象和方法调用进行编码以便传输，但在信道的配置属性中也可以指定其编码和解码格式化程序接收器。  
System.Runtime.Remoting.Contexts  
包含定义所有对象所驻留的上下文的对象。上下文时个有序的属性序列，用与定义其中的对象所处的环境。上下文时在对象的激活过程中创建的，这些对象被配置为要求某些自定服务，如同步，事务，实时（JIT）激活、安全性。多个对象可以存留在以个上下文内。  
System.Runtimg.Remoting.Lifetime  
包含管理远程对象生存期的类。传统上，分布式垃圾回收功能使用引用计数和Ping来控制对象的生存期、这种机制在每一项服务只有较少的客户端时可以正常工作，但是当每一项服务有几千个客户端时就不能正常工作了。远程处理生存期服务将每一项服务与一个租约关联，当租约到期时，就会删除该服务。生存期服务可以起到传统的分布式垃圾回收器的作用，并在每一项服务的客户端数量增加时很好的调整。  
System.Reuntime.Remoting.Metadata  
包含可用于为对象和字段定义SOAP的生成和处理的类和属性。此命名空间中的类可用于指示SOAPAction、类型输出、XML元素名和SML命名空间URL方法。  
System.Runtime.Remoting.Metadata.W3cXsd2001  
包含有WWW联合会（W3C）在2001年定义的XML架构定义（XSD）。W3C中的“XML Schema Part2:Data type”（XML架构第二布冯：数据类型）规范确定了各种数据类型的格式和行为。此命名空间包含符合W3C规范的数据类型的包装类。所有日期和时间类型都符合ISO标准规范。  
System.Runtime.Remoting.MetadataServices  
包含利用Soapsuds.exe命令行工具和用户代码在元数据和远程处理基础结构的XML架构之间相互转的类。  
System.Runtime.Remoting.Proxies  
包含控制和提供代理功能的类。代理时作为远程对象映像的本地对象。代理使客户端可以跨远程处理边界访问对象。  
System.Runtime.Remoting.Services  
包含为.NET框架提供功能的服务类。  
System.Runtime.Serialization  
包含可以用于序列化和反序列化对象的类。序列化是对象或对象图转换为线性的字节序列以存储或传输到其他位置的过程。反序列化是接受存储的信息并用这些信息重新建立对象的过程。  
System.Runtime.Serialization.Formatters  
提供由序列化格式化程序使用的通用枚举、接口和类。  
System.Runtime.Serialization.Formatters.Binary  
包含可以用二进制格式序列化和反序列化对象的BinaryFormatter类。  
System.Runtime.Serialization.Formatters.Soap  
包含可以用SOAP格式序列化和反序列化对象的SoapForamtter类。  
System.Security  
提供公共语言运行库安全系统的基础结构，包括权限的基类。  
System.Security.Cryptography  
提供加密服务，包括数据的安全编码和解码，以及其他许多操作，如散列处理、随即生成和消息身份验证。  
System.Security.Cryptography.X509Certificates  
包含Authenticode X.509 v.3证书的公共语言运行库实现。此证书用唯一明确标识证书持有者的私钥签名。  
System.Securiyt.Cryptography.XML  
包含在.NET框架安全系统中供独占使用的XML模型。此XML模型不用于常规用途。此模型允许对XML对象进行数字签名。  
System.Security.Permaissions  
定义根据策略控制操作和资源访问的类。  
System.Security.Policy  
包含代码组、成员条件和证据。这三种类性的类用于创建由公共语言运行库安全策略系统应用的规则。证据类是安全策略的输入，成员条件是开关；二者共同创建策略语句并确定授予的权限集。策略级别代码组是策略层次的结构。代码组是规则的封装并且在策略级别中分层排列。  
System.Security.Principal  
定义表示运行代码的安全上下文的用户对象。  
System.ServiceProcess  
提供使您可以实现、安装和控制Windows服务应用程序的类。服务是不需要用户界面、长期运行的可执行文件。实现服务的过程包括：从ServiceBase类继承，定义在传入启动、停止、暂定和继续命令时处理的特定行为，以及定义当前系统关闭时所采取的自定义行为和操作。  
System.Text  
包含表示ASCII、Unicode、UTF-7和UTF-8字符编码的类，用于在字符块和字节块之间相互转换的抽象基类，以及不需要创建字符串的中间实例就可以操作和格式化字符串对象的帮助器类。  
System.Text.RegularExpressions  
提供对.NET框架正则表达式引擎的访问的类。此命名空间提供可在Microsoft.NET框架上运行的任何平台或语言中使用的正则表达式功能。  
System.Threading  
提供支持多线程编程的类和接口。此命名空间包括管理线程组的ThreadPool类、允许在指定的一段时间后调用委托的Time类，以及用于同步互斥的线程的Mutex类。此命名空间还提供用于线程安排、等待通知和死锁的类。  
System.Timers  
提供允许以指定的间隔引发时间的Timer组件。  
System.Web  
提供支持浏览器/服务器通信的类和接口。此命名空间包括提供有关当前HTTP请求的大量信息的HTTPResponse类、管理HTTP到客户端的输出的HTTPResponse类，以及提供对服务器端实用工具和进程访问的HTTPServerUitility对象。System.Web还包括用于Cookie操作、文件传输、异常信息和输出缓存控制的类。  
System.Web.Caching  
提供用于在服务器上缓存常用资源的类。这些资源包括ASP.NET页、Web服务和用户空间。另外，缓存词典可以供您存储常用资源，如散列表和其他数据结构。  
System.Web.Configuration  
包含用于设置ASP.NET配置的类。  
System.Web.Hosting  
提供对Microsoft Internet Information Server(IIS)外部的托管应用程序承载ASP.NET应用程序功能。  
System.Web.Mail  
包含使用CDOSYS消息组建构造和发送消息的类。邮件消息通过Microsoft Windows 2000中内置的SMTP邮件服务或任意的SMTP服务器发送。此命名空间中的类可从ASP.NET或任何托管应用程序中使用。  
System.Web.Security  
包含用于在Web服务器应用程序中实现ASP.NET安全的类。  
System.Web.Services  
包含可以生成和使用Web服务的类。Web服务式驻留在Web服务器上并通过标准Internet协议公开的可编程实体（XML、SOAP）。  
System.Web.Services.Configuration  
由一些类组成，这些类配置用ASP.NET创建的XML Web services的运行方式。  
System.Web.Service.Description  
由一些类组成，这些类使您能够使用Web服务描述语言（WSDL）公开描述SML Web services。此命名空间中的每一个类对应于WSDL规范中的一个特定元素，并且类层次机构对应于有效的WSDL文档的XML结构。  
System.WebServices.Discovery  
由一些类组成，这些类允许XML Web Services客户端通过称为“XML Web services发现”的进程来定位Web服务其上可用的XML Web services。  
System.Web.Services.Protocols  
由一些类组成，这些类定义在通信期间通过网络在XML Web services 客户端和用ASP.NET创建的XML Web services之间传输数据的协议。  
System.Web.SessionState  
提供支持在服务器上存储特定于Web应用程序中的单个客户端的数据的类和接口。会话转台数据用于向客户端提供与应用程序的持久连接的外观。状态信息可以存储在本地进程内存中，或者，对于网络场配置来说，可以使用ASP.NET状态服务或SQL Server数据库状态信息存储进程外。  
System.Web.UI  
提供创建以Web页上用户界面形式出现在Web应用程序中的空间和页的类和接口。此命名空间包括Control类，该类为所有空间（不论使HTML空间、Web空间还是用户控件）提供一组通用功能。它还包括Page空间，每当对Web应用程序中的页发出请求时，都会自动生成此空间。另外还提供了一些类，这些类提供Web窗体服务器空间数据绑定功能、保存给定控件或页的视图状态的能力，以及对可编程控件和文本空间都适用的分析通能。  
System.Web.UI.Design  
包含用于扩展Web用户界面设计时的支持类。  
System.Web.UI.Design.WebControls  
包含用于扩展Web服务器控件设计时所支持的类。  
System.Web.UI.HtmlControls  
提供可以在Web页上创建HTML服务器控件的类。HTML服务器控件在服务器上运行，并直接映射到所有浏览器支持的标准HTML标记。这使您能够以编程方式控制Web页上的HTML元素。  
System.Web.UI.WebControls  
包含可以在Web服务器控件的类。Web控件在服务器上运行，并包括窗体控件（如按钮和文本框）以及特出用途的控件（如日历）。这使您能构以编程方式控制Web页上的这些元素。Web控件比HTML控件更抽象。它们的对象模型不一定反映HTML语法。  
System.Windows.Forms  
包含用于创建基于Windows的应用程序的类，这些应用程序可以充分利用Microsoft Windows操作系统中的丰富用户界面功能。  
System.Windows.Forms.Design  
包含可用于扩展Windows窗体设计时支持的类。  
System.Xml  
提供基于标准的XML架构（XSD）支持。  
System.Xml.Serialization  
包含用于将对象序列化为XML格式的文档或流的类。  
System.Xml.Xpath  
包含Xpath分析器和计算引擎。它支持W3C XML路径语言（XPath）1.0版建议。  
System.Xml.Xsl  
提供可扩展样式表转换（XSLT）支持。它支持W3C XSL转换（XSLT）1.0版建议。

## 9.0 [C#/.NET整数的三种强制类型转换（int）、Convert.ToInt32（）、int.Parse()的区别](http://www.cnblogs.com/xiesong/p/3536692.html)

这三种方式都是强制把内容转换为整数，但他们之间是有区别的，如下:

一、（int）适合简单数据类型之间的转换，C#的默认整型是int32（不支持bool型）。

二、int.Parse（string sParameter）是个构造函数，参数类型只支持string类型，Parse就是把String类型转换成int,char,double....等,也就是\*.Parse(string) 括号中的一定要是string类型。

三、Convert.ToInt32（）适合将Object类型转换为int型；Convert可以提供多种类型的转换,也就是Convert.\*()括号中可以为很多种类型(包括string)。

四、Convert.ToInt32（）和int.Parse()对于空值（null）的处理不同，Convert.ToInt32(null)会返回0而不会产生任何异常，但int.Parse(null)则会产生异常。

五、对于数字的精确度不同：

（1）、Convert.ToInt32(double value) 如果 value 为两个整数中间的数字，则返回二者中的偶数，即 4.5 转换为 4，而 5.5 转换为 6。

（2）、int(4.6) = 4，int转化其他数值类型为int时没有四舍五入，强制转换(截取整数部分)。

## 10.0 IntPtr和as

先来看看MSDN上说的:用于表示指针或句柄的平台特定类型。这个其实说出了这样两个事实，IntPtr 可以用来表示指针或句柄、它是一个平台特定类型。对于它的解释，这个哥们写的比较好：It's a class that wraps a pointer that is used when calling Windows API functions. The underlying pointer may be 32 bit or 64 bit, depending on the platform.

二：用在什么地方

（1）C#调用WIN32 API时

（2）C#调用C/C++写的DLL时（其实和1相同，只是这个一般是我们在和他人合作开发时经常用到）

三：怎样用

例如有一函数原型定义为：DLLDemo\_API int \_\_stdcall Inptr\_Test (LONG param1, HWND hWnd);那么我们在C#中引用时就要这样写：

 [DllImport("DllPlayer.dll", EntryPoint = "IP\_TPS\_OpenStream")]  
 public static extern int  Inptr\_Test (int param1, IntPtr hWnd);

在调用的时候就可以向Inptr\_Test 的第二个参数传入某一控件的Handle。这里涉及到C#类型与C++类型的对应关系，网上这种有很多，这里就不再赘述，只谈几个经常用到的和经常出错的。

      （1）一般对于char\* ,void\*这种可以直接对应IntPtr,比如在C#中，我们经常用string类型，其转换为IntPtr再传给char\*,void\*等，转换方法为

string txt="test"; Marshal.StringToCoTaskMemAuto(txt);

这里有时会用StringToCoTaskMemAnsi，不过StringToCoTaskMemAuto自动分配内存就可以了。这样就会将txt的内容复制到非托管的内存块中。

(2)对于结构体，比如有一结构体 StructText，将其转换为Intptr，尽量不要直接用Marshal.StructureToPtr，这样很容易出错。可以这样来用：

intsize = Marshal.SizeOf(StructText);//获取结构体占用空间大小

IntPtrintptr= Marshal.AllocHGlobal(size);//声明一个同样大小的空间

Marshal.StructureToPtr(StructText, intptr, true);//将结构体放到这个空间中

（3） 通过IntPtr访问内存

byte\* bytes = (byte\*)intPtr.ToPointer();

直接转换成类型指针（但前提是要允许使用不安全代码 unsafe code）

for (int index = 0; index < size; index++) 

{ 

     Console.Write(\*(bytes + index) + ","); 

}

        Byte[] aray = new Byte[data\_size];

 for(int i=0;i<data\_size;i++)  
          {  
               aray[i] = Marshal.ReadByte(p\_data, i);  
          }

       //Marshal.Copy(aray, 0, p\_data, data\_size);   将数据从托管堆拷贝到非托管堆使用

（4） 通过IntPtr 释放内存

Marshal.FreeHGlobal(p\_data);

as

在实际编码中有时会用到as关键字来将对象转换为指定类型，与is类型不同的是，is关键字是用于检查对象是否与给定类型兼容，如果兼容就返回true，如果不兼容就返回false。而as关键字会直接进行类型转换，如果转换成功会返回转换后的对象，如果转换不成功则不会抛出异常而是返回null。

## 11.0 字符相关拓展：

String：适用于少量的字符串操作的情况

StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

## 二、提升

## 1.C#项目解决方案管理器中将\*.Designer.cs文件放到\*.cs文件下

写C#项目时，会复用到以前项目中的.cs文件；在解决方案管理器中手动添加窗口文件后，\*.Designer.cs文件和\*.resx文件不会在.cs文件下。

  原项目中Form1的设计器文件和资源文件在Form1.cs文件下；当复制到新项目后，三个文件在同一级，在新项目中使用窗口设计器打开Form1，也显示不出原来的界面。

## 2.[C#中Trim()、TrimStart()、TrimEnd()的用法](http://www.cnblogs.com/carekee/articles/2094731.html)

**C#中Trim()、TrimStart()、TrimEnd()的用法：  
    这三个方法用于删除字符串头尾出现的某些字符。Trim()删除字符串头部及尾部出现的空格，删除的过程为从外到内，直到碰到一个非空格的字符为止，所以不管前后有多少个连续的空格都会被删除掉。TrimStart()只删除字符串的头部的空格。TrimEnd()只删除字符串尾部的空格。**

**如果这三个函数带上字符型数组的参数，则是删除字符型数组中出现的任意字符。如Trim("abcd".ToCharArray())就是删除字符串头部及尾部出现的a或b或c或d字符，删除的过程直到碰到一个既不是a也不是b也不是c也不是d的字符才结束。  
      这里最容易引起的误会就是以为删除的是"abcd"字符串。如下例：  
          string s = " from dual union all ";  
          s = s.Trim().TrimEnd("union all".ToCharArray());  
      可能有人以为上面s的最终结果是"from dual"，但真正的结果是"from d"。需要注意的是这种写法执行的删除对象是字符数组中出现的任意字符，而不是这些字符连在一起组成的字符串！**

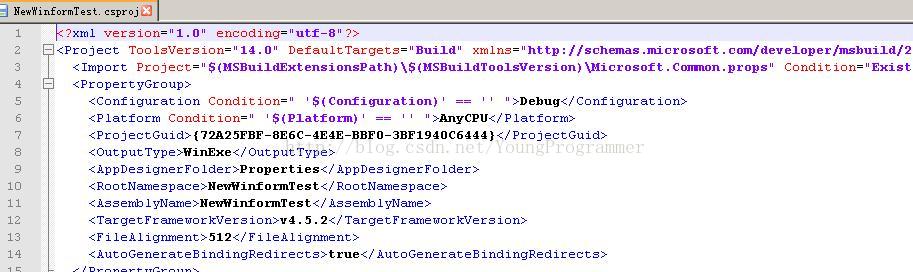
**一般TRIM函数用法：   
　　Trim() 　　功能删除字符串首部和尾部的空格。 　　语法Trim ( string ) 　　参数string：string类型，指定要删除首部和尾部空格的字符串返回值String。函数执行成功时返回删除了string字符串首部和尾部空格的字符串，发生错误时返回空字符串（""）。如果任何参数的值为NULL，Trim()函数返回NULL。 　　======================================================================== 　　SQL 中的 TRIM 函数是用来移除掉一个字串中的字头或字尾。最常见的用途是移除字首或字尾的空白。这个函数在不同的资料库中有不同的名称： 　　MySQL: TRIM(), RTRIM(), LTRIM() 　　Oracle: RTRIM(), LTRIM() 　　SQL Server: RTRIM(), LTRIM() 　　各种 trim 函数的语法如下： 　　TRIM([[位置] [要移除的字串] FROM ] 字串): [位置] 的可能值为 LEADING (起头), TRAILING (结尾), or BOTH (起头及结尾)。 这个函数将把 [要移除的字串] 从字串的起头、结尾，或是起头及结尾移除。如果我们没有列出 [要移除的字串] 是什么的话，那空白就会被移除。 　　LTRIM(字串): 将所有字串起头的空白移除。 　　RTRIM(字串): 将所有字串结尾的空白移除。**

## 3

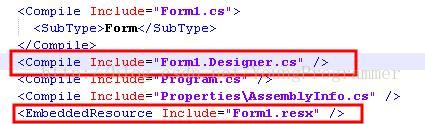
        

  为了让设计器能够正常使用，只有将\*.Designer.cs文件和\*.resx文件放到.cs文件下；具体做法如下：

1. 找到NewWinformTest项目的项目文件NewWinformTest.csproj，使用文本编辑器打开

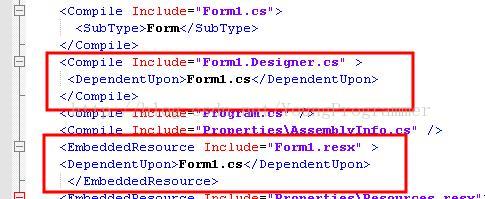


2.找到Form1.Designer.cs文件和Form1\*.resx文件的节点



3. 增加节点的DependentUpon属性，属性值上层文件的文件名

 <DependentUpon>Form1.cs</DependentUpon>



4.保存后重新打开项目\*.Designer.cs文件和\*.resx文件就在.cs文件下了！！

不光是窗口文件，其他所有的文件都能使用这种方式添加文件结构，让文件之间内容的关系通过结构体现出来，可以让代码结构显示更加清晰

注意：.csproj文件是项目文件，修改前最好备份，如果修改后xml文件格式不正确项目都打不开

自己不懂经常上网搜索，希望现在个人的一些小总结能够帮助到大家