甜蜜的语法糖

//SyntacticSugar – Program.cs

```
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SyntacticSugar
{
    class Program
    {
        static void Main(string[] args)
        {
            var myNum = 1;
            myNum++;

            Person p = new Person
            {
                Name = "张三",
                Age = 30
            };

            //int[] nums = new int[5];
            //for (int i = 0; i < nums.Length; i++)
            //{
            //    nums[i] = i;
            //}


            var nums = new int[] { 0, 1, 2, 3, 4 };

            var list = new List<int> { 1, 2, 3, 4, 5, 6 };

            var dict = new Dictionary<int, string>
            {
                { 1, "test1" }, { 2, "test2" }
```

```
            };

            var obj1 = new { Name = "张三", Age = 30 };
            var obj2 = new { Name = "李四", Age = 40 };

            //输出：true
            Console.WriteLine(obj1.GetType() == obj2.GetType());
            //输出：<>f__AnonymousType0`2[System.String,System.Int32]
            Console.WriteLine(obj1.GetType());
            //输出：Name:张三  Age:30
            Console.WriteLine("Name:{0} Age:{1}", obj1.Name, obj1.Age);



            dynamic test = "string";
            //输出：System.String
            Console.WriteLine(test.GetType());
            test = 100;
            //输出：System.Int32
            Console.WriteLine(test.GetType());

            dynamic myBag=new ExpandoObject();
            myBag.intValue = 100;
            myBag.message = "hello";
            Action<string> act= (str) => { Console.WriteLine(str); };
            myBag.say = act;

            //200
            Console.WriteLine("{0}",myBag.intValue*2);
            //HELLO
            Console.WriteLine(myBag.message.ToUpper());
            //Hello
            myBag.say("Hello");

            Console.ReadKey();
        }
}

class Person
```

```
{
        public int Age { get; set; }
        public string Name { get; set; }

}
namespace version1
{

        class MyClass
        {
                private int _value;
                public int GetValue()
                {
                        return _value;
                }
                public void SetValue(int value)
                {
                        _value = value;
                }
        }
}
namespace version2
{
        class MyClass
        {
                private int _value;
                public int Value
                {
                        get
                        {
                                return _value;
                        }
                        set
                        {
                                _value = value;
                        }
                }
        }
```

```
    }

    namespace version3
    {
        class MyClass
        {
            public int Value { get; set; }
        }
    }

}
```

TPL：基于任务的并行计算框架

//SequentialvsParalled – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

namespace SequentialvsParalled
{
    class Program
    {
        //要处理的数据项数
        const int DataSize = 1000000;

        //每个数据项进行的数据操作次数
        const int OperationCounterPerDataItem  = 100;

        static void Main(string[] args)
        {
            int[] data1 = new int[DataSize];
            Console.WriteLine("\n 敲任意键执行串行算法...\n");
            Console.ReadKey(true);
```

```
            Console.WriteLine("开始执行“纯”的顺序算法");
            Stopwatch sw = Stopwatch.StartNew();
            IncreaseNumberInSquence(data1,0,data1.Length);
            Console.WriteLine("顺序算法执行结束，使用了{0}毫秒
",sw.ElapsedMilliseconds);


            Console.WriteLine("\n 敲任意键执行并行算法一...\n");
            Console.ReadKey(true);

            Console.WriteLine("开始执行并行算法一");
            sw = Stopwatch.StartNew();
            IncreaseNumberInParallel(data1);
            Console.WriteLine("并行算法一执行结束，使用了{0}毫秒",
sw.ElapsedMilliseconds);

            Console.WriteLine("\n 敲任意键执行并行算法二...\n");
            Console.ReadKey(true);

            Console.WriteLine("开始执行并行算法二");
            sw = Stopwatch.StartNew();
            IncreaseNumberInParallel2(data1);
            Console.WriteLine("并行算法二执行结束，使用了{0}毫秒",
sw.ElapsedMilliseconds);

            Console.WriteLine("\n 敲任意键执行并行算法三...\n");
            Console.ReadKey(true);

            Console.WriteLine("开始执行并行算法三");
            sw = Stopwatch.StartNew();
            IncreaseNumberInParallel3(data1);
            Console.WriteLine("并行算法三执行结束，使用了{0}毫秒",
sw.ElapsedMilliseconds);

            Console.ReadKey(true);

        }
```

```
        //依次给一个数组中指定部分的元素(共 counter 个)执行
OperationCounterPerDataItem 次操作
        static void IncreaseNumberInSquence(int[]  arr,int startIndex,int counter)
        {
            for (int i = 0; i <counter;  i++)
                for (int j = 0; j < OperationCounterPerDataItem;  j++)
                    arr[startIndex+i]++;
        }

        //依据 CPU 核心数将任务划分为多个子任务，然后并行执行

        static void IncreaseNumberInParallel(int[] arr)
        {
            Console.WriteLine("手工创建并行任务，因为本机 CPU 为{0}核，所
以分为{0}个子任务交给 TPL 并行执行",Environment.ProcessorCount);
            //计算每个子任务要处理的数据项数，这里假设能正好整除
            int counter = DataSize / Environment.ProcessorCount;

            Parallel.For(0, Environment.ProcessorCount,  i =>
                {
                    int startIndex = i * counter;
                    IncreaseNumberInSquence(arr,  startIndex, counter);
                }
            );

        }


        static void IncreaseNumberInParallel2(int[] arr)
        {
            Console.WriteLine("针对原始数组执行并行循环，使用任务并行库
内部默认算法", arr.Length);
            Parallel.For(0, arr.Length, i =>
            {
                //直接在这里使用串行循环，则速度很快
                for (int j = 0; j < OperationCounterPerDataItem;  j++)
                    arr[i]++;
            }
```

```
            );
        }
        //任务划分过细，反而减慢运行速度
        static void IncreaseNumberInParallel3(int[] arr)
        {
            //为每个数据项创建一个任务
            Console.WriteLine("共创建{0}个子任务并行执行",arr.Length);
            Parallel.For(0, arr.Length, i =>
                {
                    //为每个数据项的每个操作建立一个并行任务，
                    //则并行算法与前述所有算法相比，慢得象蜗牛！
                    Parallel.For(0, OperationCounterPerDataItem, j => arr[i]++);
                }
            );
        }
    }
}
```

//IntroduceParallel – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StudyHelper;
using System.Threading;

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

            TestParallelInvoke();
            //TestParallelLoop();
            Console.ReadKey();

        }
```

```
        #region "并行执行"
        static void DoWork1()
        {
            Console.WriteLine("工作一，由线程{0}负责执行……",
Thread.CurrentThread.ManagedThreadId);
        }
        static void DoWork2()
        {
            Console.WriteLine("工作二，由线程{0}负责执行……",
Thread.CurrentThread.ManagedThreadId);
        }
        static void DoWork3()
        {
            Console.WriteLine("工作三，由线程{0}负责执行……",
Thread.CurrentThread.ManagedThreadId);
        }

        static void TestParallelInvoke()
        {
            Console.WriteLine("主线程{0}启动并行操作……",
Thread.CurrentThread.ManagedThreadId);
            Parallel.Invoke(
                () => DoWork1(),
                () => DoWork2(),
                () => DoWork3()
                );
            Console.WriteLine("并行操作结束，敲任意键退出主线程{0}……",
Thread.CurrentThread.ManagedThreadId);
        }
        #endregion

        #region "并行处理"

        static void VisitDataItem(int i)
        {
```

```
            Console.WriteLine("访问第{0}个数据项,由线程{1}负责执行", i,
Thread.CurrentThread.ManagedThreadId);
        }

        static void ProcessDataItem(int DataItem)
        {
            Console.WriteLine("处理数据项{0},由线程{1}负责执行", DataItem,
Thread.CurrentThread.ManagedThreadId);
        }
        private static void TestParallelLoop()
        {

            var intList = Enumerable.Range(1, 8);
            Console.WriteLine("原始数据");

            foreach (var item in intList)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("\n 主线程{0}启动并行操作……\n",
Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("For 并行执行后的数据:");
            Parallel.For(0, intList.Count(), VisitDataItem);
            Console.WriteLine("\nForEach 并行执行后的数据:");
            Parallel.ForEach(intList,ProcessDataItem);
            Console.WriteLine("\n 并行操作结束，敲任意键退出主线程
{0}……\n", Thread.CurrentThread.ManagedThreadId);
        }
        #endregion
    }
}
```

//FromThreadToTPL – frmMain.cs

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

```
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.Diagnostics;
using System.Threading.Tasks;


namespace CalculateVarianceOfPopulation
{
    public partial class frmMain : Form
    {
        public frmMain()
        {
            InitializeComponent();
        }
        /// <summary>
        ///  数据总个数（注意要能被 ThreadCount 整除)
        /// </summary>
        private const int DataSize = 90000000;


        /// <summary>
        ///  测试数据
        /// </summary>
        private double[] Data = new double[DataSize];


        #region "生成测试数据"
        private void btnGenerateData_Click(object sender, EventArgs e)
        {
            ShowInfo("正在生成测试数据，请稍候...");
            Thread th = new Thread(GeneratePopulation);
            th.IsBackground = true;
            th.Start();
        }

        /// <summary>
        ///  生成测试数据
```

```
/// </summary>
private void GeneratePopulation()
{
    Random ran = new Random();
    for (int i = 0; i < DataSize; i++)
        Data[i] = ran.NextDouble() * 100;

    string info = string.Format("\n 已生成{0}个测试数据\n", DataSize);
    ShowInfo(info);
    EnableDisableControl();
}

#endregion


#region "串行处理"
/// <summary>
///  计算数据平均值(串行算法)
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
private double CalcuateMeanInSequence(double[] data)
{
    double sum = 0;
    for (int i = 0; i < data.Length; i++)
        sum += data[i];
    return sum / data.Length;
}

/// <summary>
///  计算方差(串行算法）
/// </summary>
/// <param name="mean">要统计数据的平均值</param>
private double CalculateVarianceInSequence(double mean)
{

    double sum = 0;
    double SubtractionValue = 0;
```

```
        for (int i = 0; i < Data.Length; i++)
        {
            //保存数据项与平均值的差
            SubtractionValue = Data[i] - mean;
            //求平方和
            sum += SubtractionValue * SubtractionValue;
        }
        return sum / Data.Length;
    }

    private void btnUseSequence_Click(object sender, EventArgs e)
    {
        Thread th = new Thread(DoWorkInSequence);
        th.IsBackground = true;
        th.Start();
    }
    /// <summary>
    ///  串行完成整个工作
    /// </summary>
    private void DoWorkInSequence()
    {
        string str = "";
        Stopwatch sw = new Stopwatch();
        sw.Start();

        ShowInfo("\n=======串行算法========\n");
        ShowInfo("正在计算数据的平均值...\n");
        double mean = CalcuateMeanInSequence(Data);
        str = string.Format("测试数据的平均值为：{0}\n", mean);
        ShowInfo(str);
        str = "\n 开始计算方差...\n";
        ShowInfo(str);

        double variance = CalculateVarianceInSequence(mean);

        str = string.Format("测试数据的方差为:{0}\n", variance);
        ShowInfo(str);
        str = string.Format("\n 串行算法用时:{0}毫秒\n",
```

```
sw.ElapsedMilliseconds);
                ShowInfo(str);




            }

            #endregion


            #region "并行处理（使用线程）"



            /// <summary>
            /// 每个数据与平均值的差值的平方和
            /// </summary>
            private double SquareSumUsedByThread = 0;

            /// <summary>
            /// 用于等待工作线程运行结束
            /// </summary>
            private CountdownEvent counterForThread = new
CountdownEvent(Environment.ProcessorCount);

            /// <summary>
            /// 用于互斥访问线程共享变量 SquareSum
            /// </summary>
            private object SquareSumLockObject = new object();

            /// <summary>
            /// 线程安全的显示信息函数
            /// </summary>
            /// <param name="Info"></param>
            private void ShowInfo(string Info)
            {
```

```
            if (InvokeRequired)
            {
                Action<string> del = (str) => { rtfInfo.AppendText(str); };
                this.BeginInvoke(del, Info);
            }
            else
                rtfInfo.AppendText(Info);
        }

        /// <summary>
        /// 激活用于统计数据的控件，灰掉“生成数据”的控件
        /// </summary>
        private void EnableDisableControl()
        {
            Action del = () =>
            {
                groupBox1.Enabled = true;
                btnGenerateData.Enabled = false;
            };
            this.Invoke(del);
        }

        /// <summary>
        /// 使用多线程并行计算每个数据与平均值的差值的平方和
        /// </summary>
        /// <param name="ThreadArguObject"></param>

        private void CalculateSquareSumInParallelWithThread(object
ThreadArguObject)
        {
            ThreadArgu argu = ThreadArguObject as ThreadArgu;

            double sum = 0;
            double SubtractionValue = 0;
            for (int i = 0; i < argu.Count; i++)
            {
                //保存数据项与平均值的差
                SubtractionValue = Data[argu.StartIndex + i] - argu.Mean;
```

```
                //求平方和
                sum += SubtractionValue * SubtractionValue;
            }
            lock (SquareSumLockObject)
            {
                SquareSumUsedByThread += sum;
            };
            //通知别的线程自己已经完成工作
            counterForThread.Signal();
            string str = string.Format("\n 工作线程{0}已完成工作\n",
Thread.CurrentThread.ManagedThreadId);
            ShowInfo(str);

        }

        /// <summary>
        /// 使用线程并行计算方差
        /// </summary>
        private void DoWorkInParalleUseThread()
        {
            string str = "";
            Stopwatch sw = new Stopwatch();
            sw.Start();
            counterForThread.Reset();
            SquareSumUsedByThread = 0;
            ShowInfo("\n=======使用线程的并行算法========\n");
            ShowInfo("正在计算数据的平均值...\n");
            double mean = CalcuateMeanInSequence(Data);
            str = string.Format("测试数据的平均值为：{0}\n", mean);
            ShowInfo(str);
            //获取 CPU 核心数，作为并行线程数
            int ThreadCount = Environment.ProcessorCount;
            str = string.Format("\n 现在启动{0}个工作线程开始计算每个数据与
平均值的差值的平方和...\n", ThreadCount);

            //计算每个线程需要处理的数据项数
            int workload = DataSize / ThreadCount;
```

```
            //启动 ThreadCount 个工作线程并行执行工作
            for (int i = 0; i < ThreadCount; i++)
            {
                ThreadArgu argu = new ThreadArgu();
                argu.Count = workload;
                argu.Mean = mean;
                argu.StartIndex = workload * i;

                Thread th = new
Thread(CalculateSquareSumInParallelWithThread);
                th.IsBackground = true;
                th.Start(argu);

            }

            //主控线程等待工作线程完成工作
            counterForThread.Wait();

            str = "\n 所有工作线程都已经完成工作，现在可以计算方差...\n";
            ShowInfo(str);


            double variance = SquareSumUsedByThread / Data.Length;
            str = string.Format("测试数据的方差为:{0}\n", variance);
            ShowInfo(str);

            str = string.Format("\n 使用线程的并行算法用时:{0}毫秒\n",
sw.ElapsedMilliseconds);
            ShowInfo(str);

        }

        private void btnUseThread_Click(object sender, EventArgs e)
        {
            Thread th = new Thread(DoWorkInParalleUseThread);
            th.IsBackground = true;
            th.Start();
        }
```

```
        #endregion


        #region "并行处理（使用 TPL）"

        /// <summary>
        /// 针对特定索引范围内的元素，根据处理器个数分区，然后并行对每个分区并行执行一个数据处理函数
        /// </summary>
        /// <param name="fromInclusive"></param>
        /// <param name="toExclusive"></param>
        /// <param name="body"></param>
        /// <returns></returns>
        public static ParallelLoopResult ForRange(int fromInclusive, int toExclusive, Action<int, int> body)
        {

            // 依据本机所包容的处理器个数来决定并行处理的任务数
             int numberOfPartitions = System.Environment.ProcessorCount;

             // 获取要计算的数据范围
             int range = toExclusive - fromInclusive;

             //计算出每个并行任务要计算的数据个数
             int stride = range / numberOfPartitions;
             if (range == 0) numberOfPartitions = 0;

             //并行执行计算任务
             return Parallel.For(0, numberOfPartitions, i =>
             {
                 int start = i * stride;
                 int end = (i == numberOfPartitions - 1) ? toExclusive : start + stride;
                 body(start, end);
             });
        }
```

```
/// <summary>
/// 使用 TPL 并行计算方差
/// </summary>
private void DoWorkInParallelUseTPL()
{

    string str = "";
    Stopwatch sw = new Stopwatch();
    sw.Start();
    ShowInfo("\n=======并行算法（使用 TPL）========\n");
    SquareSumUsedByThread = 0;

    str = "\n 计算平均值\n";
    ShowInfo(str);
    double mean = CalcuateMeanInSequence(Data);
    str = string.Format("测试数据的平均值为：{0}\n", mean);
    ShowInfo(str);

    ForRange(0, Data.Length, (start, end) =>
    {
        double sum = 0;
        double temp = 0;
        for (int i = start; i < end; i++)
        {
            temp = Data[i] - mean;

            sum += temp * temp;
        }
        //保存结果
        lock (SquareSumLockObject)
        {
            SquareSumUsedByThread += sum;
        };
    });

    str = string.Format("\n 测试数据的方差为:{0}\n",
SquareSumUsedByThread / DataSize);
```

```
            ShowInfo(str);

            str = string.Format("\n 并行算法用时:{0}毫秒\n",
sw.ElapsedMilliseconds);
            ShowInfo(str);




        }

        private void btnUseTPL_Click(object sender, EventArgs e)
        {
            Thread th = new Thread(DoWorkInParallelUseTPL);
            th.IsBackground = true;
            th.Start();
        }

        #endregion

    }

    /// <summary>
    ///  向工作线程传入的参数
    /// </summary>
    public class ThreadArgu
    {
        /// <summary>
        ///  数组中元素的开始索引
        /// </summary>
        public int StartIndex
        {
            get;
            set;
        }
        /// <summary>
        ///  要计算的元素个数
```

```
        /// </summary>
        public int Count
        {
            get;
            set;
        }
        /// <summary>
        ///  数据的平均值
        /// </summary>
        public double Mean
        {
            get;
            set;
        }
    }


}
```

//MyImageProcessor – winMain.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.IO;
using System.Threading;
using WinForm = System.Windows.Forms;
using System.Threading.Tasks;
using System.Diagnostics;
```

```
namespace MyImageProcessor
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class winMain : Window
    {

        BitmapSource bmpSource = null;
        int stride = 0;
        byte[] ImagePixelData = null;   //用于保存图像的像素数据，将用于图像处理
        byte[] ImagePixelDataBackup = null; //图像的像素数据备份，用于还原
        WinForm.OpenFileDialog openFileDialog1 = null;

        public winMain()
        {
            InitializeComponent();
            openFileDialog1 = new WinForm.OpenFileDialog();
            openFileDialog1.Filter = "图像文件|*.jpg;*.gif;*.png;*.jpeg|所有文件|*.*";

        }


        /// <summary>
        ///  装入图像的像素数据到字节数组中。
        /// </summary>
        /// <param name="ImagePath"></param>
        private void LoadImage(string ImagePath)
        {
            bmpSource  = new BitmapImage(new Uri(ImagePath));
            stride = bmpSource.PixelWidth  * bmpSource.Format.BitsPerPixel  / 8;
            stride += 4 - stride % 4;
            int ImagePixelDataSize = stride * bmpSource.PixelHeight * bmpSource.Format.BitsPerPixel  / 8; ;
            ImagePixelData = new byte[ImagePixelDataSize];
```

```
        bmpSource.CopyPixels(ImagePixelData, stride, 0);
        //备份数据
        ImagePixelDataBackup = new byte[ImagePixelDataSize];
        bmpSource.CopyPixels(ImagePixelDataBackup, stride, 0);
    }



    /// <summary>
    ///  处理图像
    /// </summary>
    private void ProcessImageData()
    {
        if (ImagePixelData == null)
            return;
        long UsedTime = 0;




        if (chkSingleThread.IsChecked == true)
        {
            Stopwatch sw = new Stopwatch();
            sw.Start(); //启动计时


            for(int i=0;i<ImagePixelData.Length;i++)
            {
                byte value = ImagePixelData[i];
                ImagePixelData[i] = (byte)(Math.Sin(~value) * 255);
            }

            sw.Stop();//停止计时
            //获取算法执行时间
            UsedTime = sw.ElapsedMilliseconds;
        }
        else
```

```
            {
                Stopwatch sw = new Stopwatch();
                sw.Start(); //启动计时
                Parallel.For(0, ImagePixelData.Length, (i) =>
                    {
                        byte value = ImagePixelData[i];
                        ImagePixelData[i] = (byte)(Math.Sin(~value)*255);
                    });
                sw.Stop();//停止计时
                //获取算法执行时间
                UsedTime = sw.ElapsedMilliseconds;
            }
            ShowImageFromPixelData(ImagePixelData,image1);
            lblTime.Text = UsedTime.ToString();
        }

        /// <summary>
        ///  将字节数组中保存的图像像素数据显示在 Image 控件中
        /// </summary>
        /// <param name="imagePixelData">图像像素数据数组</param>
        /// <param name="imgControl">用于显示图像 Image 控件</param>
        private void ShowImageFromPixelData(byte[] imagePixelData, Image
imgControl)
        {
            BitmapSource newImageSource =
BitmapSource.Create(bmpSource.PixelWidth, bmpSource.PixelHeight,
                bmpSource.DpiX, bmpSource.DpiY, bmpSource.Format,
bmpSource.Palette,
                imagePixelData, stride);

            imgControl.Source = newImageSource;
        }

        private void btnLoadPicture_Click(object sender, RoutedEventArgs e)
        {
            if (openFileDialog1.ShowDialog() == WinForm.DialogResult.OK)
            {
                LoadImage(openFileDialog1.FileName);
```

```
                image1.Source = bmpSource;
            }
        }



        private void btnRestoreImage_Click(object sender, RoutedEventArgs e)
        {
            Array.Copy(ImagePixelDataBackup, ImagePixelData,
ImagePixelDataBackup.Length);
            ShowImageFromPixelData(ImagePixelData, image1);
        }

        private void btnProcessImage_Click(object sender, RoutedEventArgs e)
        {
            ProcessImageData();
        }
    }
}
```

//MyImageProcessor – winMain.xaml

```
<Window x:Class="MyImageProcessor.winMain"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="并行计算应用示例"   Height="300" Width="400"
Name="MainWindow">

    <DockPanel Margin="2" >
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal"
Width="350" >
            <Button Margin="10 5 10 5"    Width="100"
HorizontalAlignment="Center" Content="装入图像" Name="btnLoadPicture"
Click="btnLoadPicture_Click"></Button>
            <Button Margin="10 5 10 5"    Width="100"
HorizontalAlignment="Center" Content="处理图像" Name="btnProcessImage"
Click="btnProcessImage_Click" ></Button>
            <Button Margin="10 5 10 5"    Width="100"
HorizontalAlignment="Center" Content="恢复原图像" Name="btnRestoreImage"
Click="btnRestoreImage_Click"></Button>
```

```
            </StackPanel>
            <StackPanel DockPanel.Dock="Top" Orientation="Horizontal"
Width="300" >
                <RadioButton Margin="10 5 10 5"    HorizontalAlignment="Center"
VerticalAlignment="Center" Content="单线程" IsChecked="True"
Name="chkSingleThread" />
                <RadioButton Margin="10 5 10 5"    HorizontalAlignment="Center"
VerticalAlignment="Center" Content="多线程" Name="chkMultiThread" />
                <TextBlock Margin="10 5 10 5" HorizontalAlignment="Center"
FontSize="20"    FontWeight="Bold">
                    <Run Foreground="Red"  Name="lblTime">0</Run>
                    <Run>毫秒</Run>
                </TextBlock>
            </StackPanel>
            <Border BorderBrush="Black" BorderThickness="1" CornerRadius="10">
                <Image DockPanel.Dock="Top" Margin="10"    Stretch="Fill"
Name="image1"     />
    </Border>
     </DockPanel>

</Window>
```

//IntroduceTask – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace IntroduceTask
{
    class Program
    {
        static void Main(string[] args)
        {
            //UseTask();
            UseTaskDely();
```

```
            Console.WriteLine("演示结束，敲任意键退出...");
            Console.ReadKey(true);
        }
        //用于封装需要并行执行的代码
        static void DoSomeVeryImportantWork(int  id, int sleepTime)
        {
            Console.WriteLine("任务{0}正在执行……", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("任务{0}执行结束。", id);
        }


        static void UseTask()
        {
            Console.WriteLine("创建三个 Task 对象并启动其运行……");
            //任务方式一
            var t1 = new Task(() => DoSomeVeryImportantWork(1,  1500));
            t1.Start();

            //任务方式二
            var t2 = Task.Factory.StartNew(() => DoSomeVeryImportantWork(2,
3000));

            var t3 = Task.Run(() => DoSomeVeryImportantWork(3,  2000));

            Task.WaitAll(t1, t2, t3);
            Console.WriteLine("各任务的状态为：{0}，{1}，
{2}",t1.Status,t2.Status,t3.Status);
        }


        static void UseTaskDely()
        {
            Console.WriteLine("使用 Task.Delay()方法拖慢程序运行速度，仅供
演示！");
            Task.Run(() =>
            {
```

```
                for (int i = 1; i <= 10; i++)
                {
                    Console.WriteLine("{0}",i);
                    //在真正的程序中，多用 await Task.Delay(500);
                    Task.Delay(500).Wait();
                }
            }).Wait();

        }



    }
}
```

//getResultFromTaskUseThreadSync – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace GetResultFromTask
{
    class Program
    {

        /// <summary>
        /// 用于保存处理结果的共享资源
        /// </summary>
        static long result = 0;

        /// <summary>
        /// 用于通知启动任务的线程处理工作已完成
        /// </summary>
        private static    ManualResetEvent mre = new ManualResetEvent(false);
```

```
        static void Main(string[] args)
        {
            Action<object> TaskMethod = (end)=>
            {
                long sum=0;
                for (int i = 1; i <= (int)end; i++)
                    sum += i;
                //保存处理结果(使用 Interlocked 实现原子操作，无需加锁）
                Interlocked.Exchange(ref Program.result, sum);
                //通知调用者，工作已经完成，你可以取回结果了
                mre.Set();
            };

            //启动异步任务
            Task tsk = new Task(TaskMethod, 1000000);
            tsk.Start();

            //等待并行处理的完成以取回结果
            mre.WaitOne();

            Console.Write("程序运行结果为{0}", Program.result);
            Console.ReadKey();

        }
    }
}
```

//GetResultFromTaskResult – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace GetResultFromTaskTResult
{
    class Program
```

```
    {

        static void Main(string[] args)
        {
            Func<object,long> del = (end)=>
            {
                long sum = 0;
                for (int i = 1; i <= (int)end; i++)
                    sum += i;
                return sum;
            };
            Task<long> tsk = new Task<long>(del, 1000000);
            tsk.Start();
            Console.Write("程序运行结果为{0}", tsk.Result);
            Console.ReadKey();
        }
    }
}
```

//GetResultFromTaskWithoutThreadSync – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;


namespace GetResultFromTaskWithoutThreadSync
{
    class Program
    {

        static void Main(string[] args)
        {
            //需要并行执行的数据处理函数
            Func<object, long> ProcessData = (end) =>
            {
```

```
                long sum = 0;
                for (int i = 1; i <= (int)end; i++)
                {
                    sum += i;
                    //如果取消以下这句注释，会看到其异常被传播到后继
任务中。
                    throw new DivideByZeroException();
                }
                return sum;
            };
            //用于取回处理结果的函数
            Action<Task<long>> GetResult = (finishedTask) =>
            {
                //依据任务状态，决定后继处理工作
                if (finishedTask.IsFaulted)
                    Console.WriteLine("任务在执行时发生异常：{0}",
finishedTask.Exception);
                else
                    Console.Write("程序运行结果为{0}", finishedTask.Result);
            };

            //创建并行处理数据的任务对象
            Task<long> tskProcess = new Task<long>(ProcessData, 1000000);

            //当数据处理结束时，自动启动下一个工作任务，取回上一任务
的处理结果
            Task tskGetResult = tskProcess.ContinueWith(GetResult);

            //开始并行处理数据……
            tskProcess.Start();

            Console.ReadKey();

        }
    }
}
```

//TaskCooperation – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace TaskCooperation
{
    class Program
    {
        static void Main(string[] args)
        {
            //UseContinueWith();
            //UseConditionalContinueWith();
            //UseParentAndChildren();
            //UseParentAndChildren2();
            //UseWaitAll();
            //UseContinueWhenAll();
            Console.ReadKey(true);
        }

        #region "使用 ContinueWith"
        static void UseContinueWith()
        {
            Task.Run(() => DoStep1()).ContinueWith((prevTask) => DoStep2());
        }

        static void DoStep1()
        {
            Console.WriteLine("第一步");
        }
        static void DoStep2()
        {
            Console.WriteLine("第二步");
        }

        static void UseConditionalContinueWith()
```

```
        {

            Task<int> task = Task.Run(() =>
            {
                int value = new Random().Next(1, 100);
                //要测试出错情况，取消以下注释，Ctrl+F5 运行示例程序
                //throw new Exception("无效的数值");
                return value;
            });
            //正常运行结束，执行这句代码
            task.ContinueWith(prev =>
            {
                Console.WriteLine("前个任务传来的值为：{0}", prev.Result);
            }, TaskContinuationOptions.OnlyOnRanToCompletion);
            //出错了，执行以下这些代码
            task.ContinueWith(prev =>
            {
                Console.WriteLine("\n 任务在执行时出现未捕获异常，其信息为：\n{0}", prev.Exception);

            }, TaskContinuationOptions.OnlyOnFaulted);

            try
            {
                task.Wait();
                Console.WriteLine("工作结束");
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n 使用 try...catch 捕获 Wait()方法抛出的异常：\n{0}", ex);
            }


        }
        #endregion

        #region "一父多子类型的任务"
```

```
//第一种方式，父任务中创建子任务，然后等待其完成
static void UseParentAndChildren()
{
    Task tskParent = new Task(() =>
    {
        Console.WriteLine("父任务开始……");
        //父任务完成的工作……
        Console.WriteLine("父任务启动了两个子任务");
        //创建后继子任务并自动启动
        Task child1 = Task.Run(() =>
        {
            Console.WriteLine("子任务一在行动……");
            Task.Delay(1000).Wait();
            Console.WriteLine("子任务一结束");
        });
        Task child2 = Task.Run(() =>
        {
            Console.WriteLine("子任务二在行动……");
            Task.Delay(500).Wait();
            Console.WriteLine("子任务二结束");
        });
        //如果没有 WaitAll（）,那么，父任务将在子任务之前结束
        //可以试着注释掉以下这句，看看效果
        Task.WaitAll(child1, child2);
    });

    //启动父任务
    tskParent.Start();
    //等待整个任务树的完成
    tskParent.Wait();
    Console.WriteLine("父任务完成了自己的工作，功成身退。\n");
}
/// <summary>
///  方式二：不使用 Task.Run()创建子任务，而是使用
/// Task.Factory.StartNew()方法创建子任务，并
///  传给它一个 TaskCreationOptions.AttachedToParent 参数
///  从而无需在父任务中 WaitAll()
/// </summary>
```

```
static void UseParentAndChildren2()
{
    Task tskParent = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("父任务开始……");
        //父任务完成的工作……
        Console.WriteLine("父任务启动了两个子任务");
        //创建后继子任务并自动启动
        var child1 = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("子任务一在行动……");
            Task.Delay(1000).Wait();
            Console.WriteLine("子任务一结束");
        }, TaskCreationOptions.AttachedToParent);

        var child2 = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("子任务二在行动……");
            Task.Delay(500).Wait();
            Console.WriteLine("子任务二结束");
        }, TaskCreationOptions.AttachedToParent);


    });

    //等待整个任务树的完成
    tskParent.Wait();
    Console.WriteLine("父任务完成了自己的工作，功成身退。\n");

}
#endregion

#region"使用 WaitAll"

static void UseWaitAll()
{
    Console.WriteLine("启动三个并行任务……\n");
    var t1 = Task.Run(() => DoSomeVeryImportantWork(1,  3000));
```

```
            var t2 = Task.Run(() => DoSomeVeryImportantWork(2,  1000));
            var t3 = Task.Run(() => DoSomeVeryImportantWork(3,  300));
            Task.WaitAll(new Task[] { t1, t2, t3 });
            Console.WriteLine("\n 所有工作都执行完了。");
        }
        static void DoSomeVeryImportantWork(int  id, int sleepTime)
        {
            Console.WriteLine("任务{0}正在执行……", id);
            Thread.Sleep(sleepTime);
            Console.WriteLine("任务{0}执行结束。", id);
        }
        #endregion

        #region "使用 ContinueWhenAll"

        static void UseContinueWhenAll()
        {
            //创建“前期”任务数组
            Task[] tasks = new Task[]{
                Task.Run(() =>
                    {
                        Thread.Sleep(1000);   //模拟任务的延迟
                        Console.WriteLine("前期任务 1");
                    }),
                Task.Run(()=>
                    {
                        Console.WriteLine("前期任务 2");
                    })
            };
            //所有前期任务完成之后，启动下一个任务
            //to do:可以把 ContinueWhenAll 换成 ContinueWhenAny 进行试
验，看看结果有何不同？
            Task.Factory.ContinueWhenAll(tasks, prevTasks =>
            {
                Console.WriteLine("前期共有任务{0}个，这是收尾任务！",
prevTasks.Count());
            });
            //Task.Factory.ContinueWhenAny(tasks, prevTask =>
```

```
                //{
                //      Console.WriteLine("前期任务的状态是{0}，这是收尾任务！
", prevTask.Status);
                //});
            }
            #endregion
        }
}
```

//DownloadWebImages – winForm.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace DownloadWebImages
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
```

```
            InitializeComponent();
            Init();

        }
        //当前要下载的 Java 教学 PPT 数目为 49，这是依据教学网站上的 PPT 数目而定的
        private const int PPTCOUNT = 49;
        private void Init()
        {
            imageUrls = new List<string>();
            for (int i = 1; i <= PPTCOUNT; i++)
            {

imageUrls.Add(String.Format("http://www.jinxuliang.com/course/java/Thread/ThreadBasic/slidepic/Snap{0}.jpg", i));
            }
        }
        //用于保存 PPT 的下载链接
        private List<string> imageUrls = null;
        //用于记录已下载的 PPT
        private int DownloadCounter = 0;

        private ObservableCollection<ImageModel> images = new ObservableCollection<ImageModel>();

        private void btnDownload_Click(object sender, RoutedEventArgs e)
        {
            btnDownload.IsEnabled = false;
            DownloadCounter = 0;
            DownloadPPTsFromWeb();
        }
        /// <summary>
        ///  使用 TPL 并行下载 PPT
        /// </summary>
        private void DownloadPPTsFromWeb()
        {
            //为每个 PPT 图片创建一个并行下载的 Task 对象
            foreach (var imageUrl in imageUrls)
```

```
        {
            WebClient client = new WebClient();
            Uri imagUri = new Uri(imageUrl);
            Task.Run(() =>
            {
                return client.DownloadData(imagUri);
            }).ContinueWith((t) =>
            {
                //如果下载过程出错，显示出错信息，注意这是跨线程
更新 UI 控件
                if (t.Exception != null)
                {
                    Action info = () =>
                    {
                        btnDownload.IsEnabled = true;
                        tbInfo.Text = t.Exception.Flatten().Message;
                    };
                    Dispatcher.BeginInvoke(info);
                    return;
                }
                //下载成功，取出原始数据
                byte[] data = t.Result;
                //利用 WPF 的数据绑定机制，自动更新 UI 界面
                Action del = () =>
                {
                    images.Add(
                        new ImageModel
                        {
                            ImageSource =
ByteArrayToBitmapImage(data)
                        }
                        );
                    DownloadCounter++;
                    if (DownloadCounter == imageUrls.Count)
                    {
                        btnDownload.IsEnabled = true;
                    }
                    tbInfo.Text = String.Format("已下载{0},共{1}",
```

```
DownloadCounter, imageUrls.Count());
                        };
                        Dispatcher.BeginInvoke(del);
                    });
                }
            }

            //将图片原始数据，转换为 WPF 可以使用的位图对象
            public BitmapImage ByteArrayToBitmapImage(byte[] byteArray)
            {
                BitmapImage bmp = null;
                try
                {
                    bmp = new BitmapImage();
                    bmp.BeginInit();
                    bmp.StreamSource  = new MemoryStream(byteArray);
                    bmp.EndInit();
                }
                catch
                {
                    bmp = null;
                }
                return bmp;
            }

            private void Window_Loaded(object sender, RoutedEventArgs e)
            {
                //设定绑定数据源
                lstImages.Items.Clear();
                lstImages.ItemsSource = images;
            }
        }
        /// <summary>
        ///  设计一个支持 WPF 数据绑定的数据对象
        /// </summary>
        public class ImageModel : INotifyPropertyChanged
        {
```

```
        public event PropertyChangedEventHandler  PropertyChanged;

        private ImageSource imageSource;
        public ImageSource ImageSource
        {
            get { return imageSource; }
            set
            {
                imageSource = value;
                OnPropertyChanged("ImageSource");
            }
        }
        protected void OnPropertyChanged(string name)
        {
            var handler = PropertyChanged;
            if (null != handler)
            {
                handler(this, new PropertyChangedEventArgs(name));
            }
        }

    }
}
```

//DownloadWebImages – winForm.xaml

```
<Window x:Class="DownloadWebImages.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="使用 TPL 异步下载 PPT" Height="350" Width="525"
Loaded="Window_Loaded">
    <DockPanel >
        <StackPanel DockPanel.Dock="Top">
            <Button DockPanel.Dock="Top" x:Name="btnDownload"  Padding="5"
Content="从教学网站下载幻灯片" Margin="10" Click="btnDownload_Click"/>
            <TextBlock x:Name="tbInfo" Text="点击按钮开始从网站上下载 PPT"
TextAlignment="Center" Padding="2"/>
        </StackPanel>

        <ListBox Margin="10"    x:Name="lstImages"
```

```
HorizontalAlignment="Stretch"  VerticalAlignment="Stretch">
                <ListBox.ItemContainerStyle>
                    <Style TargetType="ListBoxItem">
                        <Setter Property="HorizontalContentAlignment"
Value="Stretch"></Setter>
                    </Style>
                </ListBox.ItemContainerStyle>
                <ListBox.ItemTemplate>
                    <DataTemplate>

                        <Border HorizontalAlignment="Center"
VerticalAlignment="Center" BorderBrush="Black" Margin="5" Padding="5"
CornerRadius="3">
                            <Image Source="{Binding ImageSource}" Margin="10"
Stretch="Fill"    Width="300" Height="200"/>
                        </Border>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>

        </DockPanel>
</Window>
```

//HandleTaskException – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HandleTaskException
{
    class Program
    {
        static void Main(string[] args)
        {
            //定义 3 个任务，每个任务都抛出一个异常
            Task task1 = new Task(() =>
```

```
        {
            throw new Exception();
        });
    Task task2= new Task(() =>
    {
        throw new IndexOutOfRangeException();
    });
    Task task3 = new Task(() =>
    {
        throw new DivideByZeroException();
    });
    //创建一个“父”任务，此任务包容着前面创建的 3 个子任务
    Task taskController = new Task(() =>
        {
            task1.Start();
            task2.Start();
            task3.Start();
            Task.WaitAll(task1, task2, task3);
        }
    );

    try
    {
        taskController.Start();
        taskController.Wait();
    }
    catch (AggregateException ae)
    {
        //“抹平”整个异常树，如果注释掉此句，则必须递归地遍历整个异常树，才能知道到底发生了哪些异常
        ae=ae.Flatten();
        Console.WriteLine("并行任务一共引发了{0}个异常",ae.InnerExceptions.Count);
        foreach (Exception ex in ae.InnerExceptions)
        {
            Console.WriteLine("{0}:{1}",ex.GetType(),ex.Message);
        }
    }
```

```
                Console.ReadKey();


            }
        }
}
```

//UseCancellationToken – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace UseCancelationToken
{

    public class ThreadFuncObject
    {
        //通过构造函数从外界传入取消令牌
        private CancellationToken _token;
        public ThreadFuncObject(CancellationToken token)
        {
            _token = token;
            _token.Register(() =>
            {
                Console.WriteLine("操作已被取消,这是操作取消时被回调的方
法");
            });
        }

        public void DoWork()     //将以多线程方式执行的函数
        {
            for (int i = 1; i <= 10; i++)
            {
```

```
                if (!_token.IsCancellationRequested)
                {
                    Thread.Sleep(500);
                    Console.WriteLine("正在工作：{0}", i);
                }
            }

        }
    }


class Program
{
    static void Main(string[] args)
    {
        CancellationTokenSource cts = new CancellationTokenSource();

        ThreadFuncObject threadObj = new ThreadFuncObject(cts.Token);
        Thread thread = new Thread(threadObj.DoWork);
        thread.Start();

        Console.WriteLine("敲任意键取消并行计算任务……");
        Console.ReadKey(true);
        cts.Cancel();
        //为方便观察，主线程在此阻塞等待工作线程执行结束
        thread.Join();
        Console.WriteLine("敲任意键退出");
        Console.ReadKey();
    }



    static void DoStep1WithCancelllationToken(CancellationToken token)
    {
        int sleepTime = new Random().Next(1, 5000);
        Console.WriteLine("第一步预计执行时间：{0}毫秒", sleepTime);
        Thread.Sleep(sleepTime);
        if (token.IsCancellationRequested)
```

```
        {
            Console.WriteLine("步骤一被取消");
            token.ThrowIfCancellationRequested();
        }
        Console.WriteLine( "第一步执行完毕");

    }
    static void DoStep2WithCancelllationToken(CancellationToken token)
    {
        int sleepTime = new Random().Next(1, 5000);
        Console.WriteLine("第二步预计执行时间：{0}毫秒", sleepTime);
        Thread.Sleep(sleepTime);
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("步骤二被取消");

            token.ThrowIfCancellationRequested();
        }
        Console.WriteLine("第二步执行完毕");

    }
    static void UseCancellationToken()
    {
        try
        {
            CancellationTokenSource cts = new CancellationTokenSource();
            Task.Factory.StartNew(() =>
DoStep1WithCancelllationToken(cts.Token)).ContinueWith((prevTask) =>
DoStep2WithCancelllationToken(cts.Token));
            Console.WriteLine("马上敲任意键取消执行");
            Console.ReadKey(true);
            cts.Cancel();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);

        }
```

```
            }
        }
}
```

//UnifiedModelForCancellation – winForm.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Collections.ObjectModel;
using System.Threading;

namespace UnifiedModelForCancellation
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();


            InitDemo();

        }
```

```
/// <summary>
///  初始化演示的相关参数
/// </summary>
private void InitDemo()
{
      threadObjs = new ObservableCollection<ThreadObject>();
      lstThreads.ItemsSource = threadObjs;

      tokenSource  = new CancellationTokenSource();

      btnNewThread.IsEnabled  = true;
      btnCancelThread.IsEnabled = false;
}

ObservableCollection<ThreadObject> threadObjs = null;


CancellationTokenSource tokenSource  = null;

private void btnNewThread_Click(object sender, RoutedEventArgs e)
{
    ThreadObject obj=new ThreadObject(tokenSource.Token);
      threadObjs.Add(obj);
      Thread th = new Thread(obj.DoWork);
      th.IsBackground = true;
      th.Start();
      btnCancelThread.IsEnabled = true;
}

private void btnCancelThread_Click(object sender, RoutedEventArgs e)
{
      tokenSource.Cancel();
      btnCancelThread.IsEnabled = false;
      btnNewThread.IsEnabled  = false;
}

private void btnRestart_Click(object sender, RoutedEventArgs e)
{
```

```
                InitDemo();
            }
        }
}
```

//UnifiedModelForCancellation – winMain.xaml

```
<Window x:Class="UnifiedModelForCancellation.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="统一线程取消模型示例" SizeToContent="WidthAndHeight"
ResizeMode="NoResize">
    <Window.Resources>
        <DataTemplate x:Key="ThreadObjectTemplate">
            <StackPanel >
                <ProgressBar Value="{Binding Path=Value}" Height="20"
Width="220" Margin="5" HorizontalAlignment="Center"
                                                />
            </StackPanel>

        </DataTemplate>
    </Window.Resources>
    <StackPanel>
        <Border Padding="2" BorderBrush="Blue" BorderThickness="1">
            <ListBox   ItemTemplate="{StaticResource ThreadObjectTemplate}"
BorderThickness="0"   Name="lstThreads" Margin="5" Height="200" Width="250"
HorizontalAlignment="Center">

        </ListBox>
</Border>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center"    >
            <Button Content="新建线程" Name="btnNewThread"
Click="btnNewThread_Click" Margin="5" Padding="5" Height="30"/>
            <Button Content="取消所有线程" Name="btnCancelThread"
Click="btnCancelThread_Click" Margin="5" Padding="5" Height="30"/>
            <Button Content="重新开始" Name="btnRestart"
Click="btnRestart_Click" Margin="5" Padding="5" Height="30"/>
        </StackPanel>
```

```
    </StackPanel>


</Window>
```

//UnifiedModelForCancellation – ThreadObject.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.ComponentModel;

namespace UnifiedModelForCancellation
{
    public class ThreadObject : INotifyPropertyChanged
    {
        private int _value;
        public int Value
        {
            get { return _value; }
            set
            {
                _value = value;
                OnPropertyChanged("Value");
            }
        }

        private CancellationToken _token ;

        public ThreadObject(CancellationToken token)
        {
            _token = token;
        }

        public void DoWork()
```

```
        {
            while (_token.IsCancellationRequested!=true)
            {
                if (Value + 5 > 100)
                    Value = 0;
                else
                    Value += 5;

                Thread.Sleep(200);
            }
        }
        protected void OnPropertyChanged(string name)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(name));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

//TaskCancel – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

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

```
var cts = new CancellationTokenSource();

//需要执行的任务函数
Action taskFunction = ()=>
{
    for (int i = 0; i < 100; i++)
    {
        cts.Token.ThrowIfCancellationRequested();
        Console.WriteLine(i);
        Thread.Sleep(200);
    }
};

Console.WriteLine("敲任意键发出取消任务请求...");

//请进行以下对比试验：

//(1) 让 Task 对象关联一个令牌对象，然后运行程序
//Task tsk = new Task(taskFunction, cts.Token);

//(2) 让 Task 对象不关联任何一个令牌对象，然后运行程序
Task tsk = new Task(taskFunction);

tsk.Start();
Console.ReadKey(true);
//发出异步取消任务请求
cts.Cancel();
Console.WriteLine("主线程已发出取消请求！");

//同步等待工作任务停止
try
{
    tsk.Wait();
}
catch (AggregateException ae)
{
    ae.Flatten(); //展平整个异常树
    foreach (Exception e in ae.InnerExceptions)
```

```
                {
                    if (e is TaskCanceledException)
                        Console.WriteLine("报告领导,您发给我的取消请求已经收到，任务已经取消！");
                    else
                    {
                        if (e is OperationCanceledException)
                            Console.WriteLine("报告领导,您发出的给其他人的取消命令已经收到，任务已经取消！");
                    }
                    Console.WriteLine("\n 捕获到的异常：\n{0}：{1}", e.GetType(), e.Message);
                }
            }


            Console.WriteLine("\nTask 对象的当前状态：{0}", tsk.Status.ToString());
            Console.ReadKey();
        }
    }
}
```

异步的魅力：.Net 异步编程指南

//IntroduceAsyncMethods – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

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

```
        //TestSyncInvoke();
        //TestAsyncInvokeUseAwait();
        TestAsyncInvokeUseTPL();
        Console.WriteLine("Main 方法：敲任意键退出。");
        Console.ReadKey(true);
    }
    /// <summary>
    /// 方法的同步调用，被调用方法不执行完毕，后续代码无法运行
    /// </summary>
    static void TestSyncInvoke()
    {
        DoLongJob();
        Console.WriteLine("DoLongJob 方法执行完毕");

    }
    /// <summary>
    /// 一个将要执行比较长时间的同步方法
    /// </summary>
    static void DoLongJob()
    {
        for (int i = 0; i < 10; i++)
        {
            Thread.Sleep(500);
            Console.WriteLine("正在工作：" + i);
        }
    }

    /// <summary>
    /// 测试方法的异步执行(使用 C#5 的 await 关键字）
    /// </summary>
    static async void TestAsyncInvokeUseAwait()
    {
        Console.WriteLine( "DoLongJob 方法正在后台执行……");
        await DoLongJobUseTask();
        Console.WriteLine("DoLongJob 方法执行完毕");
    }
    /// <summary>
    /// 使用 TPL 在另一个线程中运行代码
```

```
        /// </summary>
        /// <returns></returns>
        static Task DoLongJobUseTask()
        {
            return  Task.Run(() => DoLongJob());

        }

        /// <summary>
        ///  使用 TPL 版本实现的方法异步调用
        /// </summary>
        static void TestAsyncInvokeUseTPL()
        {
            Console.WriteLine("DoLongJob 方法正在后台执行……");

            Task.Run(() => DoLongJob())
                .ContinueWith(
                    (task) => {
                        Console.WriteLine("DoLongJob 方法执行完毕");
                    }
            );
        }
    }
}
```

//AsyncAndTread – Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

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

```
{
    Console.WriteLine("进入 Main()方法，执行线程 ID:{0},来自线程池?{1},是背景线程?{2}",
        Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread,
        Thread.CurrentThread.IsBackground);
    TestDoWorkAsync();
    Console.WriteLine("\n 返回 Main()方法，等待用户敲任意键退出。执行线程 ID:{0},来自线程池?{1},是背景线程?{2}",
        Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread,
        Thread.CurrentThread.IsBackground);
    Console.ReadKey();
}

private async static Task TestDoWorkAsync()
{
    Console.WriteLine("\n 进入 TestDoWorkAsync()方法，await 语句之前的代码执行线程 ID:{0}，来自线程池?{1},是背景线程?{2}",
        Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread,
        Thread.CurrentThread.IsBackground);
    await DoWork();
    Console.WriteLine("\n 退出 TestDoWorkAsync()方法，await 语句之后的代码执行线程 ID:{0}，来自线程池?{1},是背景线程?{2}",
        Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread,
        Thread.CurrentThread.IsBackground);
}

static Task DoWork()
{
    return Task.Run( () =>
    {
        Console.WriteLine("\n 使用 TPL 运行 DoWork 方法，负责执行的线程 ID:{0}，来自线程池？{1},是背景线程？{2}",
            Thread.CurrentThread.ManagedThreadId,
            Thread.CurrentThread.IsThreadPoolThread,
```

```
                Thread.CurrentThread.IsBackground);
            });
        }
    }


}
```

//AsyncvsTPLForWinForm – frmMain.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace AsyncVsTPLForWinForm
{
    public partial class frmMain : Form
    {
        public frmMain()
        {
            InitializeComponent();
        }

        private void btnLaunch_Click(object sender, EventArgs e)
        {
            //UseTPL();
            UseAsync();
        }
        #region "TPL 版本"
        private void UseTPL()
        {
            btnLaunch.Enabled = false;
            lblInfo.Text = "提示信息";
            //定义两个 Task 顺次工作
            Task.Factory.StartNew(
                //启动后台工作
                () => SayHelloTo("张三"))
```

```
                .ContinueWith(
                    task =>
                    {
                        lblInfo.Text = task.Result;
                        btnLaunch.Enabled = true;
                    },
                    //通知 TPL，需要捕获线程同步上下文
                    TaskScheduler.FromCurrentSynchronizationContext());
        }

        private string SayHelloTo(string person)
        {
            Thread.Sleep(2000);
            return person + ",你好！ ";
        }
        #endregion

        #region "async 版本"
        private async void UseAsync()
        {
            btnLaunch.Enabled = false;
            lblInfo.Text = "等待后台任务完成……";
            var result = await SayHelloToAsync("张三");
            lblInfo.Text = result;
            btnLaunch.Enabled = true;
        }
        private Task<String> SayHelloToAsync(string person)
        {
            return Task.Factory.StartNew(() => SayHelloTo(person));
        }
        #endregion
    }
}
```

//HowToWriteAsyncMethod – Program.cs

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

```
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace HowToWriteAsyncMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            //SayHelloAsync("张三");

            //GetTaskOfTResult();

            //FireAndForget();

            //UseTaskDelay();

            //TestAsyncLambda();

            Console.ReadKey();
        }

        #region "返回 Task 的异步方法"
        static Task SayHello(string name)
        {
            return  Task.Run(
                () =>
                {
                    Console.WriteLine("你好：{0}", name);
                });
        }


        static async void SayHelloAsync(string name)
        {
            await SayHello(name);
        }
```

```
#endregion

#region "返回 Task<T>的异步方法"
static Task<int> SumArray(int[] arr)
{
    return Task.Run(() => arr.Sum());
}

static async Task<int> GetSumAsync(int[] arr)
{
    int result = await SumArray(arr);
    return result;

}

static async void GetTaskOfTResult()
{
    int[] arr = Enumerable.Range(1, 100).ToArray();
    //可以通过 Task<T>.Result 属性阻塞等待提取结果
    Console.WriteLine("result={0}", GetSumAsync(arr).Result);
    //也可使用 await 使用“异步回调”方式取出返回值
    int result = await GetSumAsync(arr);
    Console.WriteLine("result={0}",result);
}

#endregion

#region "返回 void 的异步方法"
public static async void FireAndForget()
{
    var myTask = Task.Run(
    () =>
    {

        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Do work : {0}", i);
            Thread.Sleep(500);
```

```
            }
        });
        await myTask;
    }

    #endregion

    #region "Task.Delay"
    public static async void UseTaskDelay()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Do work : {0}", i);
            await Task.Delay(500);
        }

    }

    #endregion

    #region "异步 Lambda 表达式"

    static void TestAsyncLambda()
    {
        Action act = async () =>
        {
            for (int i = 0; i < 10; i++)
            {
                await Task.Delay(500);
                Console.WriteLine(i);
            }

        };
        act();
    }
    #endregion
}
```

```
}

//CancellationDemo – Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace CancellationDemo
{
    class Program
    {
        //一个支持取消的异步方法
        static async Task DoAsync(CancellationToken token)
        {
            for (int i = 0; i < 10; i++)
            {
                //当有取消请求时，抛出 OperationCanceledException
                token.ThrowIfCancellationRequested();
                Console.WriteLine(i);
                //用于“拖慢”程序运行速度
                await Task.Delay(500);
            }
        }
        static void Main(string[] args)
        {
            //TestTaskCancel();
            TestTimeOutCancel();
            Console.ReadKey();

        }

        static async void TestTaskCancel()
        {
            var cts = new CancellationTokenSource();
            //设置延迟 0.5 秒之后，发出取消请求
```

```
            Task.Delay(500).ContinueWith(t => cts.Cancel());

            Task task = DoAsync(cts.Token);
            try
            {
                await task;
                Console.WriteLine("异步方法结束,状态为：{0}", task.Status);
            }
            catch (Exception ex)
            {
                Console.WriteLine("异步方法被取消,任务状态为：{0}，异常类
型为：{1}，消息为：{2}",
                    task.Status,ex.GetType(),ex.Message);
            }

        }


        static async void TestTimeOutCancel()
        {
            //设置等待 0.5 秒，超时后，自动 Cancel
            var cts = new CancellationTokenSource(1500);
            Task task = DoAsync(cts.Token);
            try
            {
                await task;
                Console.WriteLine("异步方法结束,状态为：{0}", task.Status);
            }
            catch (Exception ex)
            {
                Console.WriteLine("异步方法因超时而被取消,任务状态为：
{0}，异常类型为：{1}，消息为：{2}",
                    task.Status, ex.GetType(), ex.Message);

            }
        }
```

```
    }
}
```

//ShowProgressAndCancell – frmMian.cs

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

namespace ShowProgressAndCancel
{
    public partial class frmMain : Form
    {
        /// <summary>
        /// 用于显示工作进度的组件
        /// </summary>
        Progress<int> progressReport = null;
        /// <summary>
        /// 用于设定要查找数的范围上限
        /// </summary>
        private int limit = 0;
        /// <summary>
        /// 用于提前取消操作
        /// </summary>
        CancellationTokenSource cts = null;

        public frmMain()
        {
            InitializeComponent();

            //以下两个初始化 Progress 组件的方式，可以任选一种
            InitProgressUseEvent();
```

```
            //InitProgressWithDelegate();


            limit = new Random().Next(Int32.MaxValue);

        }
        #region "初始化 Progresss 组件"
        /// <summary>
        ///  在构建 Progress 组件时直接指定 ProgressChanged 事件的回调方法
        /// </summary>
        private void InitProgressWithDelegate()
        {
            //显示进度的代码会自动推送到 UI 线程中执行
            progressReport = new Progress<int>((status) =>
            {
                //在此可以直接地编写代码访问 UI 控件
                lblInfo.Text = "已完成：" + status + "%";

            });
        }
        /// <summary>
        ///  基于事件响应的方式报告进度
        /// </summary>
        private void InitProgressUseEvent()
        {
            progressReport = new Progress<int>();
            progressReport.ProgressChanged +=
progressReport_ProgressChanged;
        }

        void progressReport_ProgressChanged(object sender, int e)
        {
            //在此可以直接地编写代码访问 UI 控件
            lblInfo.Text = "已完成：" + e + "%";
        }

        #endregion
```

```
        /// <summary>
        ///  计算指定范围内的偶数数量
        /// </summary>
        /// <param name="limit">要计算的区间上限</param>
        /// <param name="onProgressChanged">用于报告进度的组件</param>
        /// <param name="cancelToken">用于检测外界是否请求取消的取消令
牌</param>
        public async void ShowEvenNumbers(int  limit, IProgress<int>
onProgressChanged,
            CancellationToken cancelToken)
        {
            Func<int> calculateTask = () =>
            {
                int lastProgressValue = 0;
                int currentProgressValue = 0;
                int count = 0;
                for (int i = 1; i < limit; i++)
                {
                    if (i % 2 == 0)
                    {
                        count++;
                        currentProgressValue = (int)(((double)i / limit) * 100);
                        if (lastProgressValue != currentProgressValue)
                        {
                            //报造进度

onProgressChanged.Report(currentProgressValue);
                            lastProgressValue = currentProgressValue;
                        }
                    }
                    //检测外界是否发出了取消请求，如果发出了此请求，
                    //抛出 OperationCanceledException
                    cancelToken.ThrowIfCancellationRequested();
                }
                return count;

            };
            //允许被取消的计算任务
```

```
            Task<int> evenNumbersTask = Task.Run(calculateTask, cancelToken);
            try
            {
                //异步等待任务完成
                int evenNumberCount  = await evenNumbersTask;
                //可以直接访问 UI 控件，这是异步调用带给我们的方便
                lblInfo.Text = string.Format("计算结果：从{0}到{1}中共有偶数
{2}个。",
                        1, limit, evenNumberCount);

            }
            catch (OperationCanceledException)
            {
                lblInfo.Text = "计算任务已被取消。";
            }
            //允许重新启动一个新任务
            btnCancel.Enabled = false;
            btnStart.Enabled  = true;
        }

        private void btnStart_Click(object  sender, EventArgs e)
        {
            Start();
        }
        private void btnCancel_Click(object sender, EventArgs e)
        {
            Cancel();

        }
        /// <summary>
        ///  启动任务
        /// </summary>
        private void Start()
        {
            cts = new CancellationTokenSource();
            ShowEvenNumbers(limit,  progressReport, cts.Token);
            btnCancel.Enabled = true;
            lblInfo.Text = "计算任务已经启动并在后台运行，等其工作完成，
```

```
结果会自动显示";
            }
            /// <summary>
            ///  取消任务
            /// </summary>
            private void Cancel()
            {
                if (cts != null)
                {
                    cts.Cancel();
                }
            }


        }
}
```

//CatchException – Program.cs

```
using System;
using System.Threading.Tasks;

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

            TestExceptionThrowViaTaskResult();
            //TestExceptionThrowViaAwait();
            Console.ReadKey();
        }

        private static async void TestExceptionThrowViaAwait()
        {
            Console.WriteLine("使用 await 等待异步方法执行结束");
            Task<string> task = null;
            try
```

```
        {
            //当传入空串时，myAsyncMethod 将抛出
ArgumentNullException
            task = myAsyncMethod("");
            string result = await task;
            Console.WriteLine(result);
        }
        catch (Exception ex)
        {
            Console.WriteLine("\n 捕获到的异常，类型:{0}，信息：{1}",
                ex.GetType(), ex.Message);
            Console.WriteLine("\nTask.Status:{0}, \nTask.Exception:{1}",
                task.Status, task.Exception.GetType());
        }
    }

    private static void TestExceptionThrowViaTaskResult()
    {
        Console.WriteLine("使用 Task<T>.Result 等待异步方法执行结束");
        Task<String> task = null;
        try
        {

            //当传入空串时，myAsyncMethod 将抛出
ArgumentNullException
            task = myAsyncMethod("");
        }
        catch (Exception ex)
        {
            //这里的异常捕获代码是不会被执行的，myAsyncMethod 抛
出的异常，
            //将会延迟到访问 Task.Result 时
            Console.WriteLine("立即捕获的异常:{0}", ex.Message);
        }

        try
        {
            //因为访问了 Task.Result,所以在这里才抛出
```

ArgumentNullException 异常

```
                Console.WriteLine(task.Result);
            }
            catch (Exception ex)
            {
                Console.WriteLine("\n 捕获到的异常，类型:{0}，信息：{1}",
                    ex.GetType(), ex.Message);
                Console.WriteLine("\nTask.Status:{0}, \nTask.Exception:{1}",
                    task.Status, task.Exception.GetType());
            }
        }

        static async Task<string> myAsyncMethod(string info)
        {
            if (string.IsNullOrEmpty(info))
            {
                throw new ArgumentNullException();
            }
            return await Task.Run<string>(() => info.ToUpper());
        }
    }
}
```

//AsyncVoidDemo – frmMain.cs

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

namespace AsyncVoidDemo
{
    public partial class frmMain : Form
```

```
    {
        public frmMain()
        {
            InitializeComponent();
            ResetLabel();
        }

        private int left = 0;
        private int right = 0;
        private int result = 0;

        private Random ran = new Random();

        private void btnRun_Click(object sender, EventArgs e)
        {
            Console.WriteLine("UI Thread:{0}",
Thread.CurrentThread.ManagedThreadId);
            ResetLabel();

            if (rdoAsyncVoid.Checked)
            {
                TestAsyncVoid();
            }
            else
            {
                TestAsyncTask();
            }

        }

        private void ResetLabel()
        {
            lblResult.Text = "？ ";
            lblLeft.Text = "？ ";
            lblRight.Text = "？ ";
            //立即刷新
            lblResult.Refresh();
            lblLeft.Refresh();
```

```
        lblResult.Refresh();
    }

    #region "测试 async void 的异步方法"

    //生成第一个数
    private async void GeneratorLeftOperand()
    {
        await Task.Delay(new Random().Next(1, 1000));
        left = new Random().Next(1, 1000);
    }
    //生成第二个数
    private async void GeneratorRightOperand()
    {
        await Task.Delay(new Random().Next(1, 1000));
        right = new Random().Next(1, 1000);
    }
    //完成计算功能
    private async void Calculate()
    {
        await Task.Delay(new Random().Next(1, 1000));
        result = left + right;
    }

    //当异步方法返回 async void 时，由于它在独立的线程中执行，所以不但结果不对
    //就连显示都不一定正常
    private void TestAsyncVoid()
    {
        //(1)生成第一个数
        GeneratorLeftOperand();
        lblLeft.Text = left.ToString();
        //(2)生成第二个数
        GeneratorRightOperand();
        lblRight.Text = right.ToString();
        //(3)完成计算功能
        Calculate();
        lblResult.Text = result.ToString();
```

```
        }

        #endregion

        #region "使用 async Task"


        //生成第一个数
        private async Task GeneratorLeftOperandTask()
        {

            await Task.Run(async () =>
            {

                await Task.Delay(ran.Next(1, 1000));
                left = ran.Next(1, 1000);

            });
            lblLeft.Text = left.ToString();

        }
        //生成第二个数
        private async Task GeneratorRightOperandTask()
        {
            await Task.Run(async () =>
            {
                await Task.Delay(ran.Next(1, 1000));
                right = ran.Next(1, 1000);

            });
            lblRight.Text = right.ToString();
        }
        //完成计算
        private async Task CalculateTask()
        {
            //并行执行两个异步方法，生成两个数
            await Task.WhenAll(GeneratorLeftOperandTask(),
GeneratorRightOperandTask());
```

```
            //执行计算
            result = left + right;
        }

        //对于不需要返回值的异步方法，让其返回 async Task
        //在调用它的地方使用 await，就能保证工作正常
        private async void TestAsyncTask()
        {
            await CalculateTask();
            lblResult.Text = result.ToString();
        }

        #endregion
    }
}
```

//ThreadOverheadDemo – Program.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace ThreadOverheadDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadOverhead();
            Console.ReadKey();
        }
        private static void ThreadOverhead()
        {

            const int OneMB = 1024 * 1024;
```

```
                using (ManualResetEvent mre = new ManualResetEvent(false))
                {
                    int threadNum = 0;
                    long MemorySize = 0;
                    try
                    {
                        ParameterizedThreadStart ts = (mreWake) => (mreWake as
ManualResetEvent).WaitOne();

                        while (true)
                        {

                            Thread t = new Thread(ts);
                            t.Start(mre);

MemorySize=Process.GetCurrentProcess().VirtualMemorySize64;
                            Console.WriteLine("线程编号{0}:占用本进程的虚拟内
存{1}字节,{2}MB",++threadNum,
                                MemorySize,MemorySize/OneMB
                                );
                        }

                    }
                    catch (OutOfMemoryException)
                    {
                        Console.WriteLine("\n 内存不足。已创建线程：{0}个",
threadNum);
                        mre.Set();
                    }
                }
            }
        }
}
```