异步编程 - 学习笔记GUI(三)

序

主要介绍在 WinForm 中如何执行异步操作。

目录

一、在 WinForm 程序中执行异步操作

下面通过窗体示例演示以下操作-点击按钮后：

①将按钮禁用，并将标签内容改成：“Doing”（表示执行中）；

②线程挂起3秒（模拟耗时操作）；

③启用按钮，将标签内容改为：“Complete”（表示执行完成）。

```
1     public partial class Form1 : Form
2     {
3         public Form1()
4         {
5             InitializeComponent();
6         }
7 
8         private void btnDo_Click(object sender, EventArgs e)
9         {
10            btnDo.Enabled = false;
11            lblText.Text = @"Doing";
12
13            Thread.Sleep(3000);
14
15            btnDo.Enabled = true;
16            lblText.Text = @"Complete";
17        }
18    }
```

可是执行结果却是：

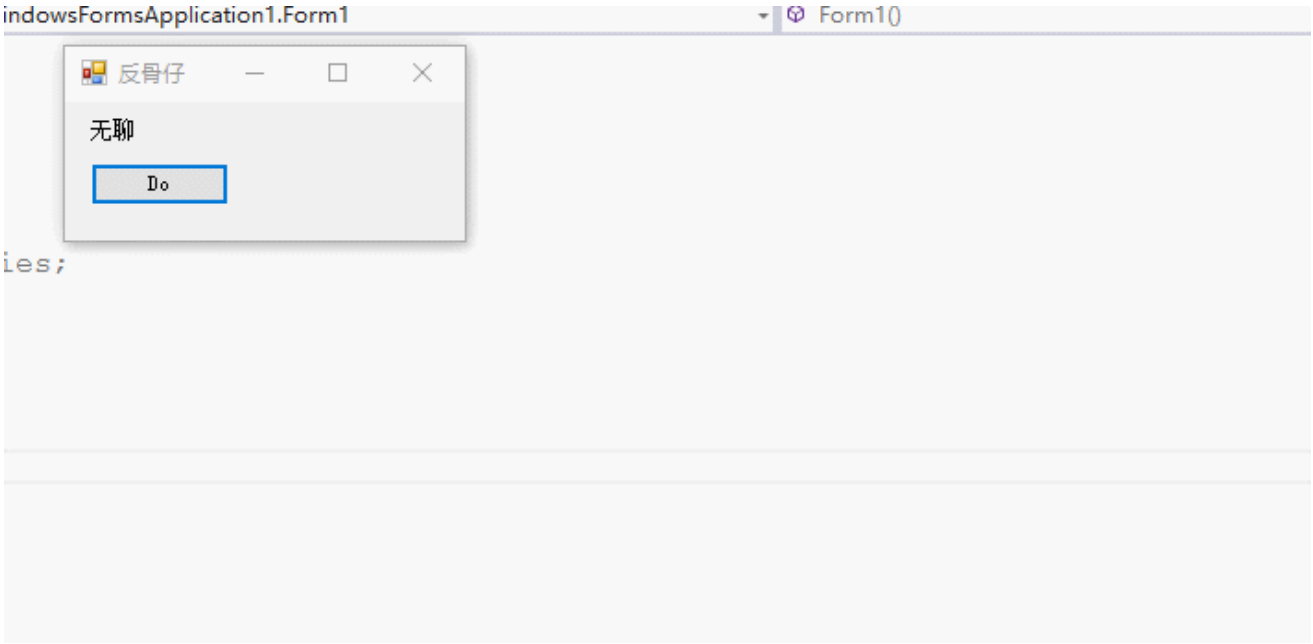

图1-1

【发现的问题】

①好像没有变成“Doing”?

②并且拖动窗口的时候卡住不动了?

③3秒后突然变到想拖动到的位置?

④同时文本变成“Complete”?

【分析】GUI 程序在设计中要求所有的显示变化都必须在主 GUI 线程中完成，如点击事件和移动窗体。Windows 程序时通过 消息来实现，消息放入消息泵管理的消息队列中。点击按钮时，按钮的Click消息放入消息队列。消息泵从队列中移除该消息，并开始处理点击事件的代码，即 btnDo_Click 事件的代码。

btnDo_Click 事件会将触发行为的消息放入队列，但在 btnDo_Click 时间处理程序完全退出前（线程挂起 3 秒退出前），消息都无法执行。（3 秒后）接着所有行为都发生了，但速度太快肉眼无法分辨才没有发现标签改成“Doing”。

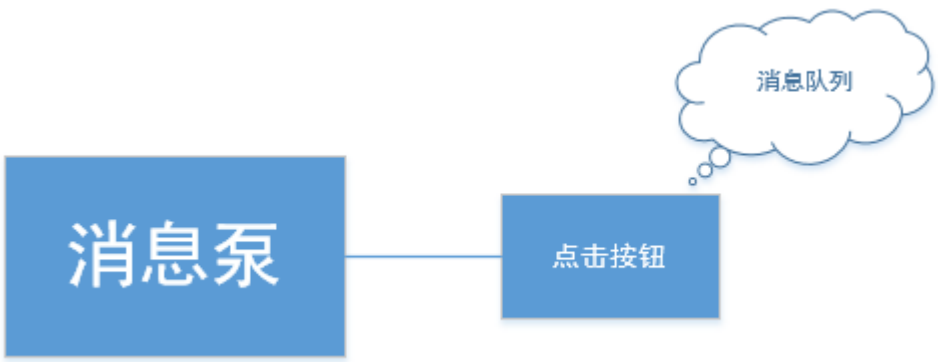

图1-2 点击事件

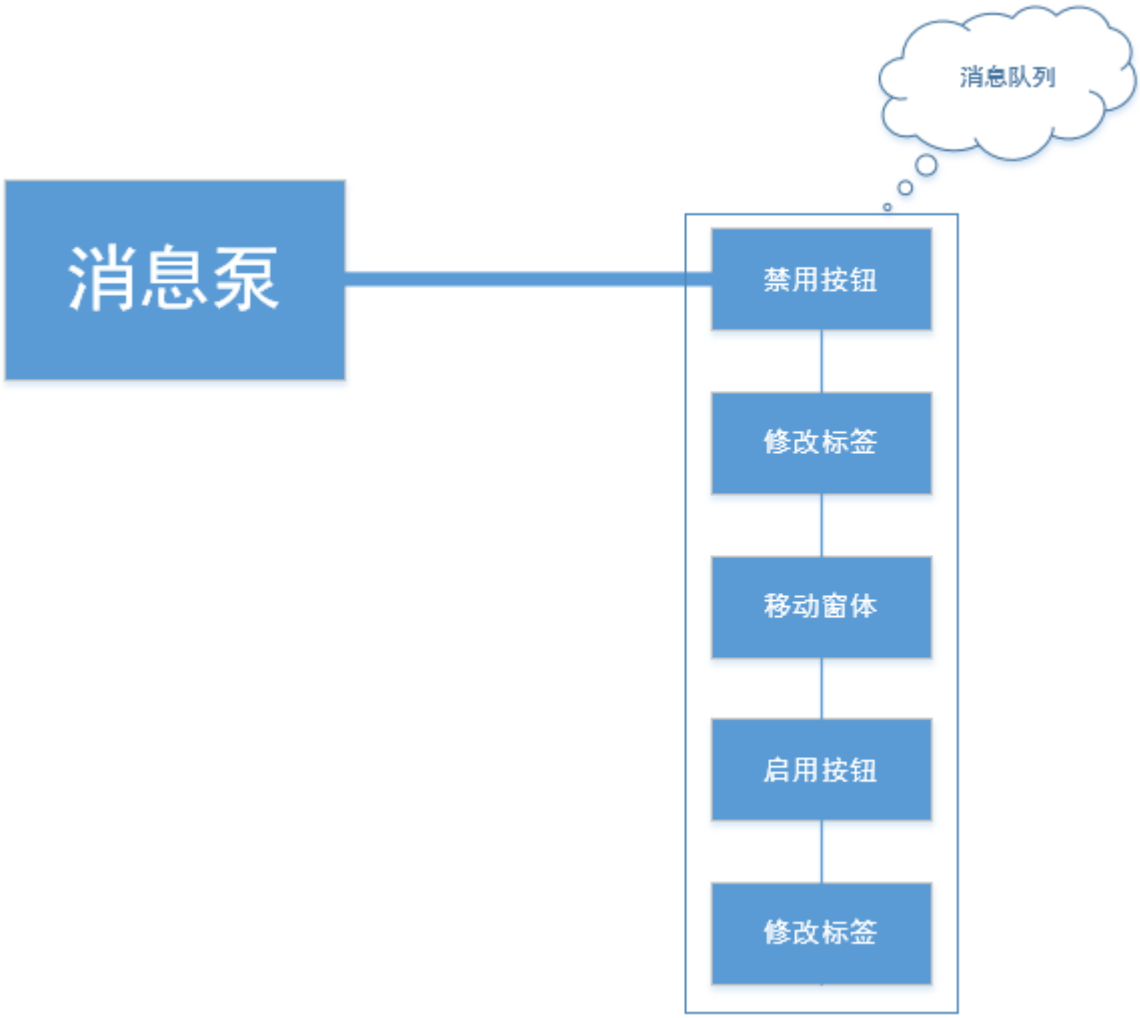

图1-3 点击事件具体执行过程

现在我们加入 async/await 特性。

```
1     public partial class Form1 : Form
2     {
3         public Form1()
4         {
5             InitializeComponent();
6         }
7 
8         private async void btnDo_Click(object sender, EventArgs e)
9         {
10            btnDo.Enabled = false;
11            lblText.Text = @"Doing";
12
13            await Task.Delay(3000);
14
15            btnDo.Enabled = true;
```

```
            lblText.Text = @"Complete";
        }
    }
```

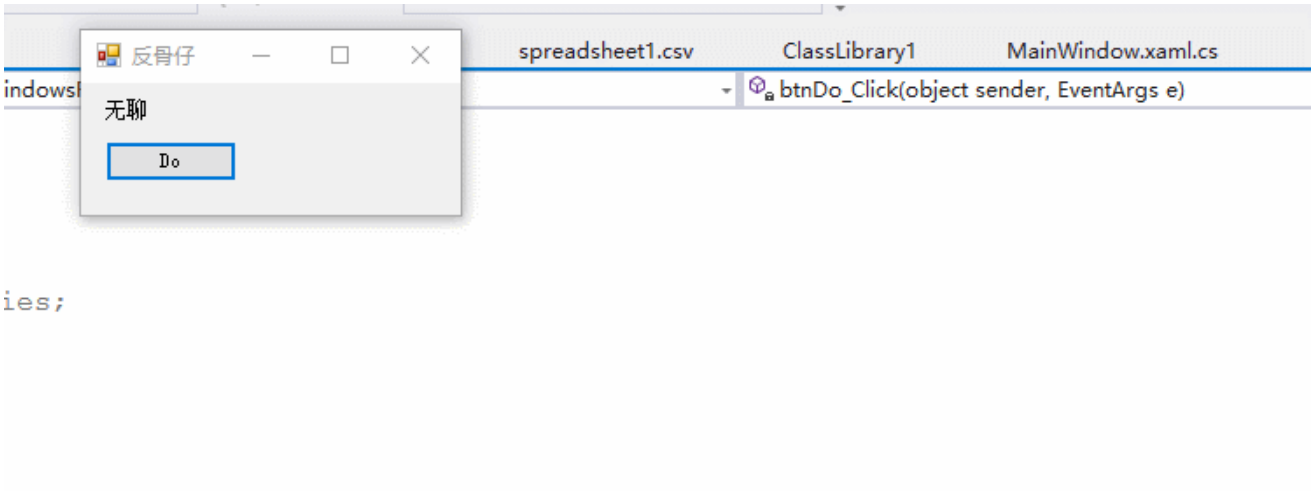

图1-4

现在，就是原先希望看到的效果。

【分析】btnDo_Click 事件处理程序先将前两条消息压入队列，然后将自己从处理器移出，在3秒后（等待空闲任务完成后 Task.Delay ）再将自己压入队列。这样可以保持响应，并保证所有的消息可以在线程挂起的时间内被处理。

1.1 Task.Yield

Task.Yield 方法创建一个立刻返回的 awaitable。等待一个Yield可以让异步方法在执行后续部分的同时返回到调用方法。可以将其理解为 离开当前消息队列，回到队列末尾，让 CPU 有时间处理其它任务。

```
    class Program
    {
        static void Main(string[] args)
        {
            const int num = 1000000;
            var t = DoStuff.Yield1000(num);

            Loop(num / 10);
            Loop(num / 10);
            Loop(num / 10);

            Console.WriteLine($"Sum: {t.Result}");

            Console.Read();
        }

        /// <summary>
        /// 循环
```

```
        /// </summary>
        /// <param name="num"></param>
        private static void Loop(int num)
        {
            for (var i = 0; i < num; i++) ;
        }
    }

    internal static class DoStuff
    {
        public static async Task<int> Yield1000(int n)
        {
            var sum = 0;
            for (int i = 0; i < n; i++)
            {
                sum += i;
                if (i % 1000 == 0)
                {
                    await Task.Yield(); //创建异步产生当前上下文的等待任务
                }
            }

            return sum;
        }
    }
```

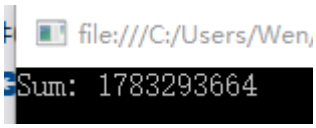

图1.1-1

上述代码每执行1000次循环就调用 Task.Yield 方法创建一个等待任务，让处理器有时间处理其它任务。该方法在 GUI 程序中是比较有用的。

二、在 WinForm 中使用异步 Lambda 表达式

将刚才的窗口程序的点击事件稍微改动一下。

```
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            //async (sender, e) 异步表达式
            btnDo.Click += async (sender, e) =>
            {
                Do(false, "Doing");
```

```
11
12                 await Task.Delay(3000);
13
14                 Do(true, "Finished");
15             };
16         }
17
18         private void Do(bool isEnable, string text)
19         {
20             btnDo.Enabled = isEnable;
21             lblText.Text = text;
22         }
23     }
```

还是原来的配方，还是熟悉的味道，还是原来哪个窗口，变的只是内涵。

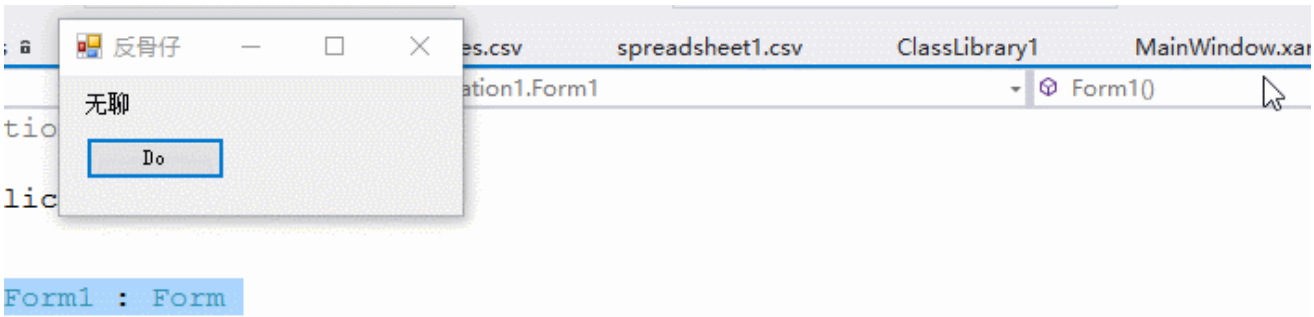

图2-1

三、一个完整的 WinForm 程序

现在在原来的基础上添加了进度条，以及取消按钮。

```
1     public partial class Form1 : Form
2     {
3         private CancellationTokenSource _source;
4         private CancellationToken _token;
5
6         public Form1()
7         {
8             InitializeComponent();
9         }
10
11        /// <summary>
12        /// Do 按钮事件
13        /// </summary>
14        /// <param name="sender"></param>
15        /// <param name="e"></param>
16        private async void btnDo_Click(object sender, EventArgs e)
```

```
17          {
18              btnDo.Enabled = false;
19
20              _source = new CancellationTokenSource();
21              _token = _source.Token;
22
23              var completedPercent = 0; //完成百分比
24              const int time = 10; //循环次数
25              const int timePercent = 100 / time; //进度条每次增加的进度值
26
27              for (var i = 0; i < time; i++)
28              {
29                  if (_token.IsCancellationRequested)
30                  {
31                      break;
32                  }
33
34                  try
35                  {
36                      await Task.Delay(500, _token);
37                      completedPercent = (i + 1) * timePercent;
38                  }
39                  catch (Exception)
40                  {
41                      completedPercent = i * timePercent;
42                  }
43                  finally
44                  {
45                      progressBar.Value = completedPercent;
46                  }
47              }
48
49              var msg = _token.IsCancellationRequested ?
                $"进度为：{completedPercent}% 已被取消！" : $"已经完成";
50
51              MessageBox.Show(msg, @"信息");
52
53              progressBar.Value = 0;
54              InitTool();
55          }
56
57          /// <summary>
58          /// 初始化窗体的工具控件
59          /// </summary>
60          private void InitTool()
61          {
62              progressBar.Value = 0;
63              btnDo.Enabled = true;
64              btnCancel.Enabled = true;
65          }
66
67          /// <summary>
68          /// 取消事件
```

```
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void btnCancel_Click(object sender, EventArgs e)
        {
            if (btnDo.Enabled) return;

            btnCancel.Enabled = false;
            _source.Cancel();
        }
    }
```

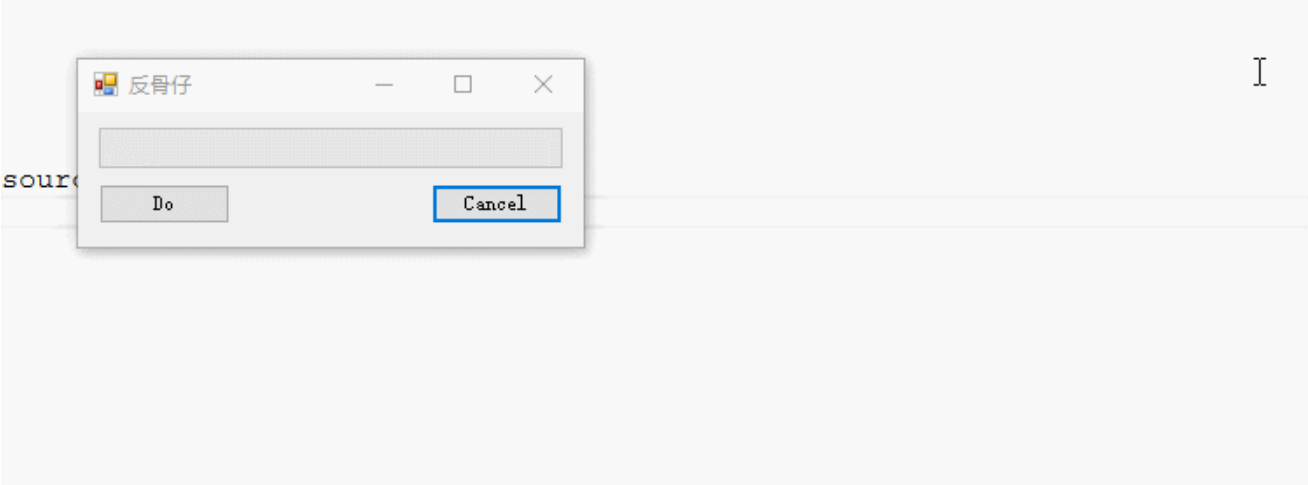

图3-1

四、另一种异步方式 - BackgroundWorker 类

与 async/await 不同的是，你有时候可能需要一个额外的线程，在后台持续完成某项任务，并不时与主线程通信，这时就需要用到 BackgroundWorker 类。主要用于 GUI 程序。

书中的千言万语不及一个简单的示例。

```
    public partial class Form2 : Form
    {
        private readonly BackgroundWorker _worker = new BackgroundWorker();

        public Form2()
        {
            InitializeComponent();

            //设置 BackgroundWorker 属性
            _worker.WorkerReportsProgress = true;    //能否报告进度更新
            _worker.WorkerSupportsCancellation = true;  //是否支持异步取消

            //连接 BackgroundWorker 对象的处理程序
            //开始执行后台操作时触发，即调用 BackgroundWorker.RunWorkerAsync 时触发
            _worker.DoWork += _worker_DoWork;
            //调用 BackgroundWorker.ReportProgress(System.Int32) 时触发
```

```
15              _worker.ProgressChanged += _worker_ProgressChanged;
                //当后台操作已完成、被取消或引发异常时触发
16              _worker.RunWorkerCompleted += _worker_RunWorkerCompleted;
17          }
18
19          /// <summary>
20          /// 当后台操作已完成、被取消或引发异常时发生
21          /// </summary>
22          /// <param name="sender"></param>
23          /// <param name="e"></param>
24          private void _worker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
25          {
26              MessageBox.Show(e.Cancelled ? $@"进程已被取消：{progressBar.Value}%" :
$@"进程执行完成：{progressBar.Value}%");
27              progressBar.Value = 0;
28          }
29
30          /// <summary>
31          /// 调用 BackgroundWorker.ReportProgress(System.Int32) 时发生
32          /// </summary>
33          /// <param name="sender"></param>
34          /// <param name="e"></param>
35          private void _worker_ProgressChanged(object sender, ProgressChangedEventArgs
e)
36          {
37              progressBar.Value = e.ProgressPercentage;   //异步任务的进度百分比
38          }
39
40          /// <summary>
41          /// 开始执行后台操作触发，即调用 BackgroundWorker.RunWorkerAsync 时发生
42          /// </summary>
43          /// <param name="sender"></param>
44          /// <param name="e"></param>
45          private static void _worker_DoWork(object sender, DoWorkEventArgs e)
46          {
47              var worker = sender as BackgroundWorker;
48              if (worker == null)
49              {
50                  return;
51              }
52
53              for (var i = 0; i < 10; i++)
54              {
55                  //判断程序是否已请求取消后台操作
56                  if (worker.CancellationPending)
57                  {
58                      e.Cancel = true;
59                      break;
60                  }
61                  //触发 BackgroundWorker.ProgressChanged 事件
62                  worker.ReportProgress((i + 1) * 10);
63                  Thread.Sleep(250);  //线程挂起 250 毫秒
```

```
64              }
65          }
66
67          private void btnDo_Click(object sender, EventArgs e)
68          {
69              //判断 BackgroundWorker 是否正在执行异步操作
70              if (!_worker.IsBusy)
71              {
72                  _worker.RunWorkerAsync();    //开始执行后台操作
73              }
74          }
75
76          private void btnCancel_Click(object sender, EventArgs e)
77          {
78              _worker.CancelAsync();  //请求取消挂起的后台操作
79          }
80      }
```