Windows Form UI 優化入門課 – 實作非同步作業

作者：李明儒

用 Visual Studio 開發 Windows Form 不是件難事，但我們很容易在使用過程瞧出程式師的功力！舉凡一執行長時間作業 UI 就會凍結沒反應的，肯定是菜鳥的作品。本文以 Windows Form 初學開發者爲對象，介紹如何利用非同步作業概念，改善使用者的操作經驗。

WinForm 菜鳥的考驗

一般人在學習程式開發的初期，思維模式會先以單一執行緒(Thread)爲主，A 指令跑完再執行 B 指令，接著再呼叫 C 函數，一切按步就班，條理清清楚楚。而對一般程式來說，單一執行緒便足以滿足”結果正確”的基本要求；大部分的情況下，引入多執行緒非同步執行，多半著眼於執行效率的提升或改善操作互動性。

在 ASP.NET 中，每一個 ASPX 網頁會由一條執行緒執行，當同時有多人存取網站時，便是不折不扣的多執行緒模式，不過這一段是 ASP.NET Framework 的職責。許多資深 ASP.NET 網頁設計師，即便享用多執行緒的好處多年，對於非同步作業及多執行緒卻仍然陌生，在踏入 Windows Form 領域後，仍直覺地只用單一執行緒解決問題，設計出來的 Windows Form，功能正確性有一定的水準，但不免被使用者抱怨，操作過程常會 Hang 住，用起來”卡卡”的。

在 Windows Form 中，需要一條執行緒(稱之爲 UI Thread)回應使用者點選、移動滑鼠、按鍵輸入等動作，同時還要負責更新畫面顯示。如果你用這條執行緒執行需要耗費大量時間的工作，即使是不耗用 CPU 運算能力，只是純粹等待，在沒有結束自訂邏輯，將控制權交還給 Windows 之前，整個 Form UI 是凍結的，對使用者的滑鼠、鍵盤動作亦不會有任何反應，即使你用迴圈不斷更新 Label.Text，在凍結期間也不會看到 Label 有任何變化。

這個特性導致程式在執行耗時作業時，UI 呈現凍結狀況，讓使用者叫天天不應，叫地地不靈，不知道程式是當掉了還是正在忙，不管是使用者或是開發者，大家都很清楚，沒人會喜歡這種愛搞自閉的兩光程式。

自閉版

我們先用一個簡單的範例來突顯這個問題。開啓一個統計資料夾檔案數與大小的 Windows Form 專案，放上 txtPath 輸入路徑，lblResult 顯示結果，btnOK 則是執行作業用。同時，爲了辨別 Form UI 是否仍正常反應，我們再放上一個 Timer，每隔 50ms 更新一次 StausBar 顯示現在的時間。當程式執行到耗時較久才能完成的 Method 時，UI 更新會暫停，察覺到 StatusBar 時間顯示停止更新，就知道程式已陷入沈思的自閉狀態。

主要程式碼如程式 1，執行畫面如圖 1。

程式 1

```
private void Form1_Load(object sender, EventArgs e)
{
    //預設爲暫存目錄
    txtPath.Text = Path.GetTempPath();
```

```
}

private int fileCount = 0;
private long totalSize = 0;

private void btnGO_Click(object sender, EventArgs e)
{
    //重設統計數據
    fileCount = 0;
    totalSize = 0;
    //統計檔案數及大小
    explore(txtPath.Text);
    //顯示結果
    lblResult.Text = string.Format("{0:#,##0} files, {1:#,##0} bytes",
        fileCount, totalSize);
}

private void explore(string path)
{
    //使用遞迴搜索所有目錄
    foreach (string dir in Directory.GetDirectories(path))
        explore(dir);
    foreach (string file in Directory.GetFiles(path))
    {
        fileCount++;
        //加總檔案大小
        FileInfo fi = new FileInfo(file);
        totalSize += fi.Length;
    }
}

private void timer1_Tick(object sender, EventArgs e)
{
    //每隔50ms顯示最新時間
    toolStripStatusLabel1.Text = DateTime.Now.ToString("HH:mm:ss.fff");
}
```

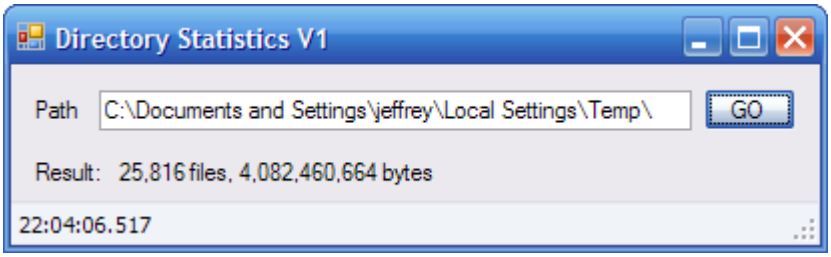

圖 1 程式執行結果

接著，我們故意讓程式出點糗。啓動程式，選擇一個檔案數較多的資料夾，按下"GO"鈕，程式開始搜索成千上萬個檔案，可能得耗上幾分鐘，此時狀態列上原本不斷跳動的時間會停滯，Form 的 UI 上的元素也不再對滑鼠鍵盤有任何回應。如果先開個 Notepad 故意將 Form 遮去一角，將 Notepad 關閉或移至下層後，被遮住的地方會呈現白色(如圖 2)，代表 Form 連最基本的視窗重畫功能都喪失了。

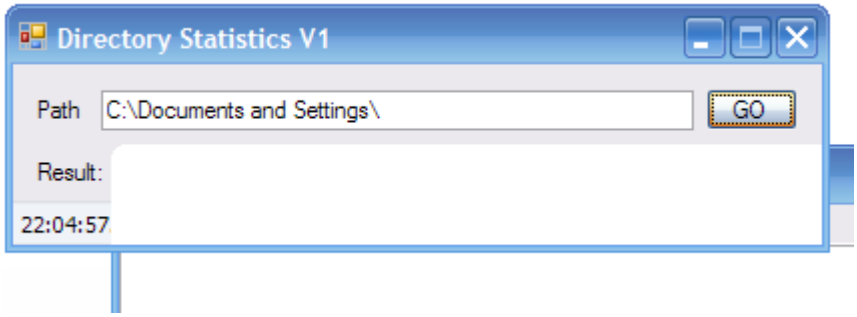

圖 2 程式 UI 停止回應，被其他視窗遮蔽的部分也不會重畫

如此的設計，一旦程式進入統計檔案流程，整個 Form UI 就完全癱瘓無反應，更不用提看不到任何執行進度的線索，讓人搞不清楚它究竟是當掉了，還是我們耐心不夠? 所有的使用者都會同意這樣的程式是很糟糕的! 那麼，身爲積極上進、奮發有爲的程式開發人員，要如何避免程式變成這等不入流的粗糙次級品呢?

改良版

托.NET 的福，不管是用 C#或 VB.NET，撰寫多執行緒程式已是件再簡單不過的小事了(不過要寫得好，卻還是有很多學問，就像寫得出可用的 ASP.NET，不代表它足以承載上千人同時使用)，有幾種選擇: 自己建立 Thread、利用 ThreadPool，或是借重.NET 2.0 的 BackgroundWorker。這裡，我們先由傳統的 ThreadPool 做法入手，對 Windows Form 的多執行緒議題建立一些基本觀念，對於未來處理與思考相關議題很有幫助。先練一點點基本功，再來體驗 BackgroundWorker 的懶人寫法，也會對於它的便利性更心存感激。

一般來說，ThreadPool 在應用上比自己建立 Thread 並管理其生命週期來得簡單些，同時還可確保在維持效率的前題下充分發揮多執行緒的威力(執行緒數目過多時，CPU 會耗費太多的資源處理 Context Switching，反而會拖累效能，而 ThreadPool 會依 CPU 數或 CPU 核心數決定適當的 Thread 數處理排入 Queue 的工作)，除非要精確管理 Thread 生命週期，建議多利用 ThreadPool 來達成多執行緒的目標。如程式 2，我們只需將 btnOK_Click 直接呼叫 explore 的寫法包入 startJob(object)中，以 startJob 建立一個 WaitCallback delegate，將它丟進 ThreadPool 的 Work Item Queue 即可。此時 Form 不需要等待 startJob 執行完成，就能跟平常一樣，繼續與使用者互動。實際執行程式，可發現按下"GO"按鈕後，statsBar 的時間顯示繼續跳動，

Form 也操作如常，不像自閉版會卡住不動。此時聽到硬碟聲大作，表示程式正在忙碌地清點檔案，但等待結果的愉快心情，稍後會因爲一個 Exception 而破滅。

程式 2

```
private void btnGO_Click(object sender, EventArgs e)
{
    //將統計作業排入ThreadPool Queue
    //Windows會另開Thread處理，UI Thread可以
    //繼續處理UI畫面顯示並與使用者互動
    ThreadPool.QueueUserWorkItem(new WaitCallback(startJob), txtPath.Text);
}

private void startJob(object path)
{
    fileCount = 0;
    totalSize = 0;
    explore(path.ToString());
    //***以下這行有點問題***
    lblResult.Text = string.Format("{0:#,##0} files, {1:#,##0} bytes",
        fileCount, totalSize);
}
```

UI Thread 限制

程式會在 startJob 中指定 lblResult.Text 時發生以下錯誤:

InvalidOperationException was unhandled

Cross-thread operation not valid: Control 'lblResult' accessed from a thread other than the thread it was created on.

在 Windows Form 的世界裡，有一條鐵律: (我喜歡稱它爲 UI Thread 限制)

"Thou shall not access UI controls from a thread other than the one that created the UI control in the first place"

“除了建立 UI Control 的那條執行緒外，你不可以用其他執行緒去存取該 UI Control!”

這條鐵律從 Windows SDK 時代就存在了，到.NET 時代依然不容撼動! 不過使用.NET 1.1 的朋友可能會抗議，因爲上面這段 Code 在.NET 1.1 下可以編譯、可以執行，也”未必”會出錯。注意哦! 是未必，代表運氣不好時，也許上線一陣子後，它有可能出現 Null Reference 之類讓你丈二金剛摸不著腦袋的錯誤，視各人造化不同，得花一段時間才能意會到問題出在違反了 UI Thread 限制。還要特別注意的是，某些程式碼對 UI Control 的變更是間接的，例如: 你在非 UI Thread 中變動了 DataTable 的內容，而該 DataTable 正好是某個 DataGrid 的 DataSource。變更 DataTable 時，觸發資料改變的非 UI Thread 繼續執行了相關的 Event 變動 DataGrid 的顯示。嗶嗶!!! 違規了，再加上運氣不好(也不是每次闖紅燈都會遇到條子)，就會跑出一堆難以聯想與 UI Thread 限制有關的錯誤。(經驗中以 Null Reference 爲主)

相形之下，.NET 2.0 Framework 在非 UI Thread 存取時立即開出罰單，並明確告知違規事由，會比.NET 1.1 時代來得容易察覺及修正，說起來更爲人性化。(在.NET 1.1 時代吃過苦頭的人，會覺得這張罰單真是可愛。)

由於 startJob 會由 UI Thread 以外的 Thread 執行，我們不能直接在 startJob 中存取 lblResult，要借用 Control.Invoke()或 Control.BeginInvoke()接呼叫更改 lblResult.Text 的程式碼。其中 Invoke()是同步呼叫，須等待 delegate 指向的程式執行完畢才會繼續向下走；BeginInvoke()則是非同步呼叫，呼叫後立即繼續向下執行。

在此，我們需要宣告一個 delegate，再用 Invoke 呼叫 delegate 執行另外宣告的 printResult。(程式 3) 透過 Invoke，我們可以確保 printResult 的程式碼會以 UI Thread 執行，就可以任意存取 UI 上的各項元素而不受限制。

【補充說明】透過 Invoke 方式呼叫的效率肯定比直接呼叫來得差，若 startJob 有可能以 UI Thread 執行也可能透過其他 Thread 執行時，可以利用 this.InvokeRequired 屬性進行測試，發現爲非 UI Thread 時用 Invoke，否則直接呼叫，以求效率的最佳化。

經過這番整修，現在按下 GO 之後，在程式統計檔案的漫長過程中，Form 仍可操作自如，統計結果也可順利顯示，我們終於擺脫動不動就卡死的自閉版時代。

程式 3

```
private void startJob(object path)
{
    fileCount = 0;
    totalSize = 0;
    explore(path.ToString());

    string text = string.Format("{0:#,##0} files, {1:#,##0} bytes", fileCount, totalSize);
    //透過Invoke，強制以UI Thread執行
    this.Invoke(
        new UpdateLableHandler(printResult),
        new object[] { text });
}
//宣告一個delegate
delegate void UpdateLableHandler(string text);
//透過Invoke，printReulst會以UI Thread執行
//故可隨意存取UI Control
private void printResult(string text)
{
    lblResult.Text = text;
}
```

豪華版

不過，光是解決自閉的問題，冗長的統計過程中無法得知進度，不能後悔取消(起手無回大丈夫嗎?)，還稱不上是一個高親和性(User-Friendly)的好程式。接下來，我們看看如何再加入這些特性。

如程式 4，我們改造了幾個地方，首先，新增一個 bool cancel，以便可以在途中剎車，放棄統計作業。而 btnOK 在按下後，顯示會由 GO 改成 Cancel，當場變身成取消按鈕。由於我們透過 cancel 旗標通知 explore()停下手中的工作，因此按下取消到實際取消間會有些時間差，這段時間要將 btnOK.Enabled 設爲 false 防止按鈕被重覆按下。

在 explore()中，我們加入兩個地方檢查 cancel 旗標，一旦發現使用者要求取消，就不再繼續執行。另外，在統計過程中仿效 startJob 用 this.Invoke 間接呼叫，每處理 100 個檔案，就更新 lblResult 顯示處理進度，使用者看著節節上升的數字不斷跳動，就不會因無法得知程式是當機還是忙碌而焦慮，獲得較佳的使用體驗。

startJob()執行 explore()完畢後則做了小小的修改，以區別是取消或執行完成，決定要顯示的文字內容。

程式 4

```
private void btnGO_Click(object sender, EventArgs e)
{
    if (btnGO.Text == "GO")
    {
        //按鈕功能變更爲取消
        btnGO.Text = "Cancel";
        cancel = false;
        ThreadPool.QueueUserWorkItem(new WaitCallback(startJob), txtPath.Text);
    }
    else
    {
        //設定取消旗標
        cancel = true;
        //在完全停止前，停用按鈕
        btnGO.Enabled = false;
    }
}
private void startJob(object path)
{
    fileCount = 0;
    totalSize = 0;
    explore(path.ToString());
    string text =
        //取消時顯示”Canceled”，不顯示統計
```

```
        (!cancel) ?
            string.Format("{0:#,##0} files, {1:#,##0} bytes", fileCount, totalSize) :
            "Canceled";
    this.Invoke(
        new UpdateLableHandler(printResult),
        new object[] { text });
}
delegate void UpdateLableHandler(string text);
private void printResult(string text)
{
    lblResult.Text = text;
    //作業完成，恢復按鈕功能
    btnGO.Text = "GO";
    btnGO.Enabled = true;
}
private void updateProgress(string text)
{
    lblResult.Text = text;
}

private bool cancel = false;

private void explore(string path)
{
    foreach (string dir in Directory.GetDirectories(path))
    {
        //使用者要求中止
        if (cancel) return;
        explore(dir);
    }
    foreach (string file in Directory.GetFiles(path))
    {
        //使用者要求中止
        if (cancel) return;

        //雖然在本例中非必要，但在此還是簡單示範利用
        //Interlocked.Increment增加fileCounter值的做法
        //防止多個Thread同時更動資料時發生衝突
        //另一種做法是用lock(C#)或SyncLock(VB.NET)
```

```
        //請參考MSDN獲得更詳細的介紹
        Interlocked.Increment(ref fileCount);

        //每100個檔案回報一下進度
        if (fileCount % 100 == 0)
            this.Invoke(
                new UpdateLableHandler(updateProgress),
                string.Format("{0:#,##0} files processed...", fileCount)
            );

        FileInfo fi = new FileInfo(file);
        totalSize += fi.Length;

    }
}
```

可以取消，又可即時掌握處理進度，現在程式又比改良版更邁進了一大步，堪稱豪華版了。

懶人式豪華版

爲了要寫出高親和度的程式，要懂的事還真不少，UI Thread、delegate、Invoke… 如先前的預告，.NET 2.0 裡出現了新的寶貝---BackgroundWorker! 可以讓你用一般的程式寫法，就做出前述豪華版的效果。

BackgroundWorker 被包裝成一個可視 Control，我們可以將其拖拉到 Form 上，並使用 Properties Window 設定它的屬性及事件。(如圖 2)

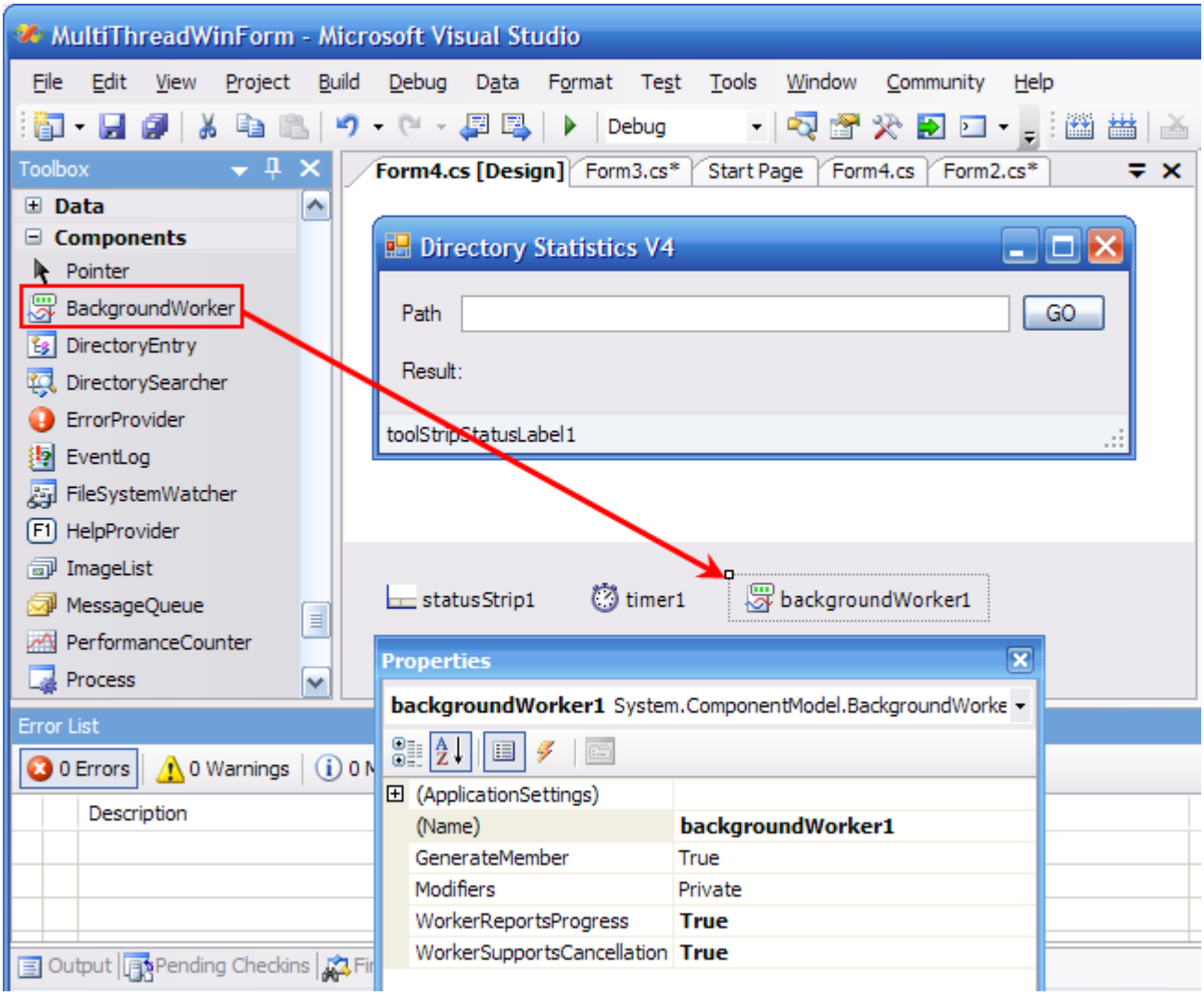

圖 2 在 VS 2005 中操作 BackgroundWorker

屬性的部分主要是設定執行中途是否要回報進度(WorkerReportsProgress)及是否支援取消(WorkerSupportsCancellation)。事件則有三個:

- DoWork: 由 BackgroundWorker. RunWorkerAsync ()觸發，事件中即爲要在背景執行的程式碼。由於會在另一個 Thread 中執行，一樣得遵守不可直接更動 UI Control 的原則，必要時可透過 DoWorkEventArgs 傳遞參數及結果。
- ProgressChanged: BackgroundWorker.ReportPrgress()時觸發，會在 UI Thread 中執行，因此可隨意存取 UI Control，與 DoWork 事件間則透過 ProgressChangedEventArgs 溝通。
- RunWorkerCompleted: DoWork 結束後觸發，亦在 UI Thread 中執行，透過 RunWorkerCompleteEventArgs 取得執行結果。

利用 BackgroudWorker，我們將程式修改如程式 5，在 btnOK_Click 時以 RunWorkerAsync()觸發 DoWork 事件，DoWork 再呼叫 explore()，呼叫時會將 BackgroundWorker 當成參數傳入，explore()過程中利用 BackgroundWorker. CancellationPending 檢查使用者是否要求取消決定是否停止作業。每統計 100 個檔案，利用 BackgroundWorker.ReportProgress()觸發 ProgressChanged 事件修改 lblResult。由於 ProgressChanged 事件是在 UI Thread 中執行，我們可以忘卻 Invoke、delegate，直接存取 UI Control，

顯然較 Invoke 做法來得易寫且單純。

DoWork 事件在工作完成後，將結果寫入 e.Cancel 或 e.Result，最後觸發在 UI Thread 執行的 RunWorkerCompleted 事件，將結果回應到 UI Control 上。

程式 5

```
private void btnGO_Click(object sender, EventArgs e)
{
    if (btnGO.Text == "GO")
    {
        btnGO.Text = "Cancel";
        fileCount = 0;
        totalSize = 0;
        //啓動非同步呼叫
        backgroundWorker1.RunWorkerAsync(txtPath.Text);
    }
    else
    {
        btnGO.Text = "GO";
        btnGO.Enabled = false;
        //取消非同步作業
        backgroundWorker1.CancelAsync();
    }

}

private void explore(string path, BackgroundWorker worker)
{
    foreach (string dir in Directory.GetDirectories(path))
    {
        //以CancellationPending判別是否要取消
        if (worker.CancellationPending) return;
        explore(dir, worker);
    }
    foreach (string file in Directory.GetFiles(path))
    {
        if (worker.CancellationPending) return;
        fileCount++;
        if (fileCount % 100 == 0)
            //利用ReportProgress回報進度
```

```
            backgroundWorker1.ReportProgress(
                0, //Progress在此未使用，設0
                string.Format("{0:#,##0} files processed...", fileCount)
            );

        FileInfo fi = new FileInfo(file);
        totalSize += fi.Length;
    }
}
//非同步作業
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    //e.Argument接入RunWorkerAsync傳入的參數
    explore((string)e.Argument, sender as BackgroundWorker);
    //執行完成，設執行定結果
    if (backgroundWorker1.CancellationPending)
        e.Cancel = true;
    else
        e.Result = string.Format("{0:#,##0} files, {1:#,##0} bytes", fileCount, totalSize);
}
//即時執行進度回報
private void backgroundWorker1_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    lblResult.Text = e.UserState.ToString();
}
//作業完畢處理
private void backgroundWorker1_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    lblResult.Text = (e.Cancelled) ? "Canceled" : //取消
        e.Result.ToString(); //結果
    //恢復按鈕功能
    btnGO.Text = "GO";
    btnGO.Enabled = true;
}
```

一模一樣的結果，卻完全不涉及 Invoke、delegate、ThreadPool、WaitCallback 這些需要花心思才能了解的術語與觀念，真正做到傻瓜都能寫多執行緒程式的境界。不過，如果講求更進階的應用，例如：同時開兩條以上的 Thread 並行、控制各 Thread 間的同步，就超出 BackgroundWorker 可以涵蓋的範圍，還是要回歸到自己管理 Thread 的根本做法。但若是單純的耗時作業，倒很適合交給 BackgroundWorker 處理，省時又省力。

結論

在本期文章中，我們討論了單執行緒 Windows Form 在 UI 操作上的嚴重缺陷，並介紹了如何運用多執行緒概念克服 UI 凍結的問題，.NET 2.0 的 BackgroundWorker 則讓多執行緒的實作變得出奇簡單。

不過，就算傻瓜相機可以輕鬆拍出好照片，真正要將攝影玩到淋漓盡致，曝光、景深這些觀念還是不可或缺的。這便是爲什麼這篇文章還是選擇了從傳統的做法出發，最後才導引到便利的現成元件，希望開發朋友們知與行兼顧，在處理問題上才會得心應手。

參考資料 1 Give Your .NET-based Application a Fast and Responsive UI with Multiple Threads
http://msdn.microsoft.com/msdnmag/issues/03/02/multithreading/