

Alpha 版

.NET 本事： 非同步程式設計



蔡煥麟 / 著

.NET 本事－非同步程式設計

Michael Tsai

This book is for sale at <http://leanpub.com/dotnet-async>

This version was published on 2016-09-12



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2016 Ministep Books

Contents

序	i
關於本書	ii
何處購買?	ii
書寫慣例	ii
需要準備的工具	iii
範例程式與補充資料	iii
關於作者	iv
1. 從零開始	1
1.1 話說從頭：處理序與執行緒	1
1.2 執行緒帶來的負擔	3
1.2.1 Context Switch	3
1.3 爭先恐後—關於優先順序	6
1.3.1 處理序的優先順序	7
1.3.2 執行緒的優先順序	8
1.4 並行、平行、非同步	9
1.5 本章回顧	10
2. .NET 非同步 API 概覽	11
2.1 專屬執行緒	11
2.1.1 建立與啓動執行緒	12
2.1.2 等待與暫停執行緒	15
2.1.3 共享變數	16
2.1.4 執行緒同步化	19
2.1.4.1 鎖定	19
2.1.5 前景執行緒 vs. 背景執行緒	21
2.1.6 使用專屬執行緒的時機	22
2.2 執行緒集區	23
2.2.1 執行緒集區的運作方式	23
2.2.2 執行緒集區的大小限制	24
2.2.3 工作執行緒與 I/O 執行緒	25
2.2.4 使用執行緒集區	25

CONTENTS

2.3	非同步程式設計模型（APM）	27
2.4	基於事件的非同步模式（EAP）	28
2.5	基於工作的非同步模式（TAP）	30
2.5.1	工作平行程式庫（TPL）	31
2.5.1.1	TPL 如何執行工作？	31
2.5.2	建立與起始非同步工作	31
2.6	重點回顧	35

序

還沒想好要說什麼....



這是預覽版的內容。

關於本書

這本書，筆者目前的打算是讓試閱章節包含全部的內容。換句話說，讀者可以免費下載與試閱整本書（這部分將來可能會調整）。若您覺得本書對您有幫助，或者想贊助作者多寫一點（或寫快一點），都歡迎付費購買，成為本書的正式讀者。每當書籍內容有更新，正式讀者都可以免費取得新版本。

何處購買？

您可以透過下列電子書平台購買本書：

- Leanpub: <https://leanpub.com/u/michaeltsai>
- Pubu: <http://www.pubu.com.tw/store/huanlin>

由於各平台的功能有一些差異，使得同一本書在不同書店的定價和閱讀體驗不太一致，這點還請讀者明察。基本上，Leanpub 的功能比較豐富，而 Pubu 則提供了更多種付款方式，而且中文介面用起來更直觀。

書寫慣例

技術書籍免不了夾雜一堆英文縮寫和不易翻譯成中文的術語，而且有些術語即使譯成中文也如隔靴搔癢，閱讀時反而會自動在腦袋裡轉成英文來理解。因此，對於這些比較麻煩的術語，有些會在第一次出現時採取中英並呈，有些則直接使用英文，例如 context switch。

書中不時會穿插一些與正文有關的補充資料，依不同性質，有的是以單純加框的側邊欄 (sidebar) 圈住，有的則會佐以不同的圖案。底下是書中常用的幾個圖案：



此區塊會提示當前範例原始碼的所在位置。



注意事項。



相關資訊。



通常是筆者的碎碎念、個人觀點（不見得完全正確或放諸四海皆準），或額外的補充說明。

需要準備的工具

本書範例皆以 C# 寫成，使用的開發工具是 Visual Studio 2013。為了能夠一邊閱讀、一邊練習，您的 Windows 作業環境至少需要安裝下列軟體：

- .NET Framework 4.5（一定要.NET 4.5 以上！）
- Visual Studio Community 2013 或 Professional 以上的版本

上述軟體皆可免費取得。其中 Visual Studio Community 2013 是微軟於 2014 年 11 月發布的社群版，可[免費下載¹](#)使用（若於企業內部使用，請留意限制條款）。

範例程式與補充資料

書中範例的完整原始碼都放在 Github 平台上，任何人皆可存取。網址是：<https://github.com/huanlin/async-book-support>

¹<http://www.visualstudio.com/products/visual-studio-community-vs>

關於作者

.NET 程式設計師，現任 C# MVP（2007 年至今），有幸曾站在恆逸講台上體會「好為人師」的滋味，也翻譯過幾本書。

近期著作：《.NET 相依性注入》²，2014。

陳年譯作：

- [軟體構築美學³](#)，2010（已絕版）。原著：Brownfield Application Development in .NET。
- [物件導向分析設計與應用⁴](#)，2009。原著：Object-Oriented Analysis and Design with Applications 3rd Edition。
- [ASP.NET AJAX 經典講座⁵](#)，2007。原著：Introducing Microsoft ASP.NET AJAX。
- [微軟解決方案框架精要⁶](#) 2007。原著：Microsoft Solution Framework Essentials。
- [軟體工程與 Microsoft Visual Studio Team System⁷](#)，2006。原著：Software Engineering and Microsoft Visual Studio Team System。

您可以透過下列管道得知作者的最新動態：

- 部落格：<http://huan-lin.blogspot.com/>
- Facebook 專頁：<https://www.facebook.com/huanlin.notes>
- Twitter：<https://twitter.com/huanlin>

²<https://leanpub.com/dinet>

³<http://www.books.com.tw/products/0010485217>

⁴<http://www.books.com.tw/products/0010427868>

⁵http://www.delightpress.com.tw/book.aspx?book_id=SKTP00007

⁶<http://www.books.com.tw/products/0010357719>

⁷<http://www.ithome.com.tw/node/40288>

1. 從零開始



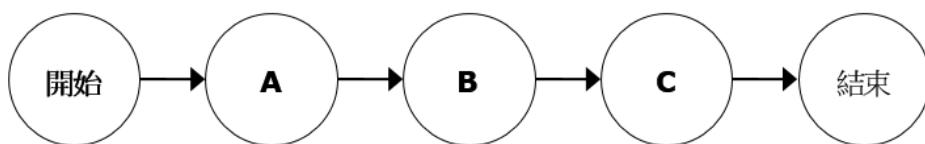
這是預覽版的內容。

大部分的程式碼都是以循序的方式執行，或者說，以同步（synchronous）的方式執行。一方面是因為我們的腦袋比較容易理解循序的執行程式碼，而較難想像同時併發、交錯執行的非同步程式碼實際上究竟是怎麼回事。另一方面，則是因為「交錯執行」所衍生的一些問題（例如鎖死 [deadlock]）需要學習額外的技巧來解決，使得非同步程式設計更加困難。

本章將從最基礎的概念開始談起，介紹處理序（process）、執行緒（thread），以及其他相關的名詞與基礎觀念，包括 context switch、捱餓（starvation）、執行緒的優先順序，以及並行（concurrency）、平行處理（parallel processing）、非同步處理（asynchronous processing）等等。

1.1 話說從頭：處理序與執行緒

從前從前，在還沒有執行緒（thread）這個概念的時候，作業系統本身與應用程式都是以類似接力賽跑的方式執行。也就是說，一件工作做完才接著做下一件；呼叫某個函式時，必須等該函式執行完畢，返回之後，呼叫端才能繼續下去。若將每一件工作用線段連接起來，結果就會像一條線，看起來像這樣：



此循序執行（後來又稱為同步執行）的方式有兩個問題。首先，對於需要跟使用者互動的應用程式來說，如果有某項工作要花很長的時間才能跑完，使用者就會在螢幕前發呆——這是很糟糕的使用者體驗。第二個問題，某個應用程式進入無窮迴圈將導致其他應用程式暫停，整個作業系統看起來就像當掉似的，使用者最終只好使出殺手鐗：強迫結束應用程式或重新開機——這當然也是很差的使用者體驗。

然後，Windows 作業系統有了「處理序」（process；又譯作「處理程序」）的概念，作為隔離應用程式的基本單位。當使用者開啓某應用程式，作業系統會將它載入記憶體並開始執行，這個載入記憶體中運行的應用程式實體（instance），便稱為處理序。一個處理序會在系統中佔據一個記憶區塊，此區塊是個獨立的虛擬位址空間，其中包含該應用程式的程式

碼以及相關資源，而且此空間只有該應用程式實體能夠存取，與別人互不相干。如此一來，運行中的各個處理序就不至於互相干擾，不會因為某個應用程式進入無窮迴圈而導致其他應用程式掛掉；同時，由於這些應用程式的處理序也被隔離於作業系統的核心程式碼之外，作業系統本身也更加穩固。

雖然應用程式與作業系統之間已經透過處理序來達到隔離和保護的效果，可是它們仍然有共用的資源：CPU。如果機器只有一顆CPU，那麼當某個應用程式進入無窮迴圈，那顆唯一的CPU就會忙著跑無窮迴圈而無暇照顧其他應用程式，形同鎖住。於是，使用者會發現每個應用程式都無法回應了，無論滑鼠點在哪裡都不起作用。為了解決這個CPU無法分身的問題，執行緒（thread）便應運而生。

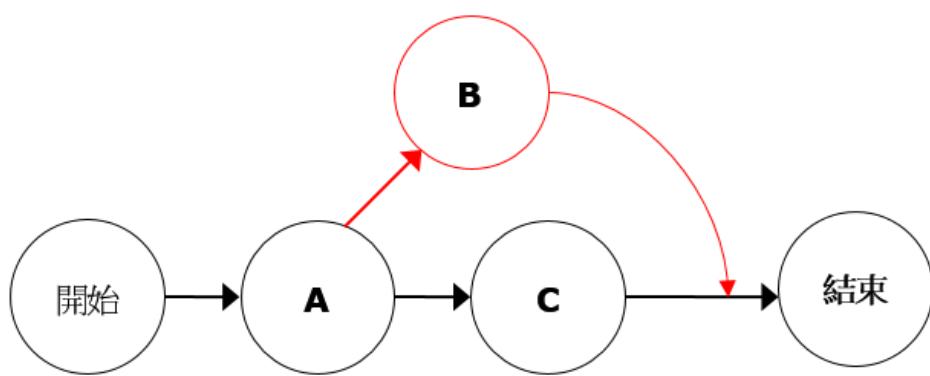
那麼，什麼是執行緒呢？

Jeffrey Richter 在《CLR via C#¹》中說：「執行緒是 Windows 作業系統用來虛擬化 CPU 的概念。」同時進一步解釋，「Windows 會給每一個處理序（process）配發一個專屬的執行緒（其功能近似於 CPU），而當某應用程式進入無窮迴圈，其所屬之處理序形同凍結，但其他處理序（擁有各自的執行緒）並未凍結；它們都還能夠繼續運行！」若要再解釋得更淺白些：執行緒就是用來切割 CPU 執行時間的基本單位，讓 CPU 好像有分身似的，可「同時」執行多項工作。或者更簡單些：執行緒就是一條獨立的程式碼序列（code sequence）。



所以我們常說「一個執行緒」，或「一條執行緒」。註：在撰寫本章內容時，我的主要參考資料是 Jeffery Richter 的《CLR Via C# 4th Edition》。

一個處理序裡面可以同時跑多條執行緒，於是原本只能單線循序執行的作業，現在變成可以向多頭馬車那樣分頭進行，如下圖所示。



¹<http://www.amazon.com/CLR-via-Edition-Developer-Reference/dp/0735667454>

1.2 執行緒帶來的負擔

執行緒解決了多個應用程式共用同一顆 CPU 所產生的問題，但也必須付出一點代價，包括空間（記憶體）與時間（執行效能）。Windows 每建立一條執行緒，須為它配置大約 1MB 左右的記憶體，其中包含執行緒核心物件、環境區塊（Thread Environment Block）、使用者模式堆疊、核心模式堆疊等等。這是記憶體空間的額外負擔。

執行緒所衍生的效能負擔包括兩個部分：與 unmanaged DLL 的互動，以及 context switch。前者涉及的底層細節，對多數讀者來說可能幫助不大，所以我把它放到底下的補充文字方塊裡面，可略過不看。後者（即 context switch）則有必要了解一下，於下一節說明。

執行緒與 unmanaged DLL

這裡說的 unmanaged DLL，你可以把它理解為傳統 Win32 DLL，也就是編譯成機器碼的 DLL。相對的，碰到 managed DLL 則可視為 .NET 組件。

每當 Windows 在某個處理序當中建立一條執行緒時，所有已經載入至該處理序的 unmanaged DLL 的 DllMain 函式都會被呼叫一遍，且呼叫時會傳入 DLL_THREAD_ATTACH 旗號。相對的，每當處理序中的執行緒釋放前，Windows 也會自動呼叫該處理序中所有 unmanaged DLL 的 DllMain 函式，並傳入 DLL_THREAD_DETACH 旗號。Windows 之所以有這個機制，主要是考慮到某些 unmanaged DLL 可能會需要在執行緒建立或摧毀時收到通知，以便進行初始化或資源清理的動作。有些應用程式運行時會載入很多 unmanaged DLL，以至於建立和摧毀執行緒的時候得多花一些時間。根據 Jeffrey Richter 在《CLR via C# 第四版》中所說，他的機器上所安裝的 Visual Studio 運行時所載入的 DLL 數量高達 470 個！剛才所提到的現象並不會發生在 C# 或 Visual Basic 程式所編譯成的 managed DLL，因為它們並沒有 DllMain。

1.2.1 Context Switch

Context Switch（環境切換、工作內容切換）其實是很貼近日常生活的概念。

比如說，當我們需要同時處理多項工作的時候，由於手邊的工作進行到一半，必須先把目前進度、待辦事項等相關資訊先記在某處，然後——有時可能還需要調整一下心情——再把另一件工作當時保存的相關資訊拿出來，讓記憶恢復一下，再繼續處理後續未完的事項。在開發軟體專案時，相信大家都有過類似的 context switch 經驗吧！



接下來要說明 Windows 作業系統的 context switch 程序時，會提到作業系統與計算機結構的幾個專有名詞，例如暫存器（register）、虛擬位址空間等等。

同樣的，對於只有一顆 CPU 的電腦而言，其實每次只能執行一件工作。故當作業系統同時載入執行多個應用程式時，Windows 就必須適當切割並分配 CPU 的運算時間給這些應

用程式的各個執行緒。於是，在某個瞬間會輪到某個執行緒擁有 CPU 資源一段短暫的時間（這段時間有個專有名詞，叫做 **quantum**）；等這短暫的時間一到，Windows 就會把 CPU 資源分配給另一個執行緒。像這樣從某個執行緒切換至另一個執行緒的程序就是 context switch，而每一次 context switch 都包含以下幾個動作：

1. 把 CPU 各個暫存器的值保存至目前執行緒的內部資料結構。
2. 挑選下一個幸運的執行緒。若該執行緒屬於另一個處理序，則在切換之前，Windows 還必須切換虛擬位址空間，這樣 CPU 才能存取到正確的程式碼和資料。
3. 從選中的執行緒之內部資料結構載入 CPU 暫存器的值。

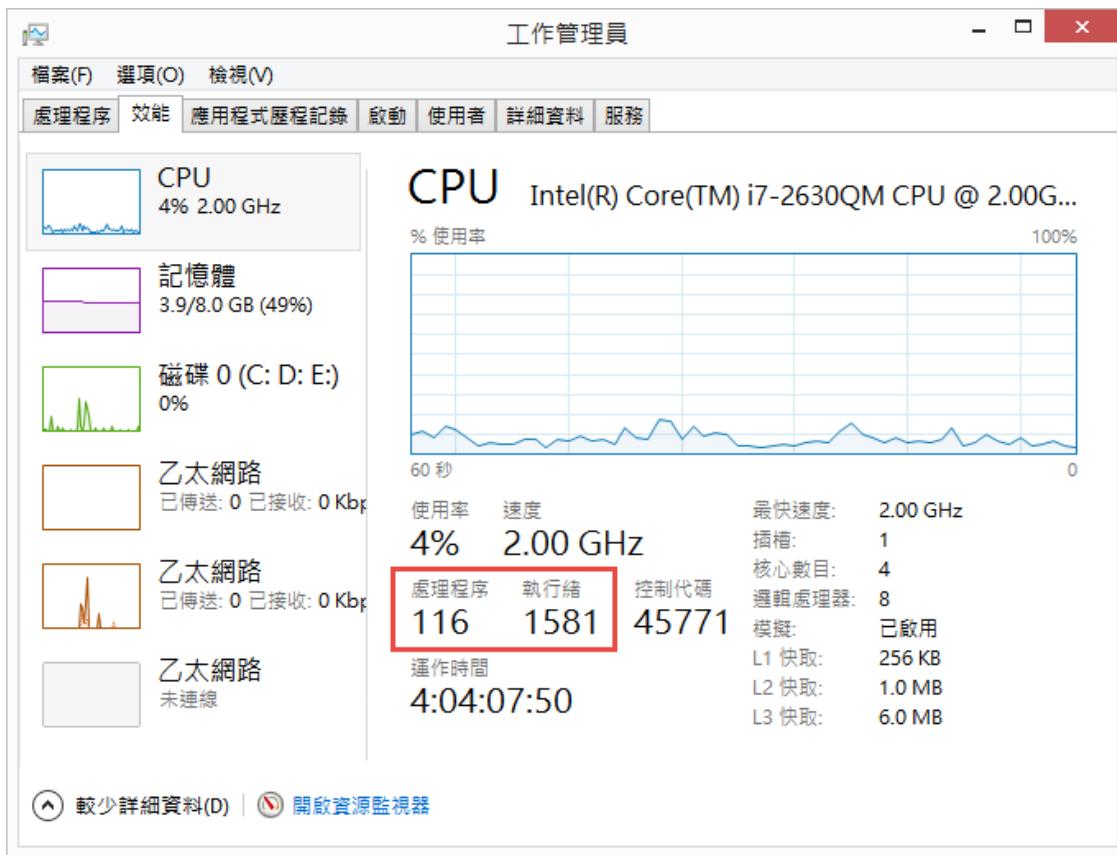
上述 context switch 動作完成後，CPU 便開始執行下一個選中的執行緒，直到分配給它的時間已過，接著又再一次切換執行緒。透過這種每隔一小段時間即切換執行緒的機制，就算某應用程式進入無窮迴圈，CPU 也不會被鎖在迴圈裡，而能夠繼續服務其他應用程式。

切換執行緒的動作會影響系統的執行效能，原因在於，CPU 本身雖然有內建快取（cache）來提升運算速度，但由於切換執行緒的緣故（Windows 大約每 30ms 切換一次），才剛剛載入快取的資料不一會兒就因為切換至另一個執行緒而又得載入新的資料，令快取形同虛設。

此外，每當 CLR 回收資源時，它會先暫停所有的執行緒，等到回收動作完成後才恢復。這表示如果應用程式能盡量減少執行緒的數量，就能改善 CLR 資源回收的效率。同樣的情形也發生在除錯時：每當除錯器碰到你設定的中斷點，Windows 會暫停該應用程式的所有執行緒，直到你再做一次單步除錯或繼續執行，那些執行緒才又「活過來」。

經過上述討論，我們知道建立、摧毀、和管理執行緒都得額外消耗一些記憶體空間，而執行緒切換也需要花一些時間。故可得出結論：在單一 CPU 的機器上，若無必要，應用程式應盡量避免建立額外的執行緒。

就拿我的電腦來說吧，下圖是 Windows 工作管理員呈現的系統效能數據。值得注意的是，處理序的數量為 116，執行緒數量為 1581，可是 CPU 整體負載卻只有 4%。這表示雖然系統目前已建立了許多執行緒，但是大多在背景閒置。



若切換至「詳細資料」頁籤，加入「執行緒」欄位，便能看到每個處理序的執行緒數量，如下圖。

The screenshot shows the Windows Task Manager with the '詳細資料' (Details) tab selected. A red box highlights the '執行緒' (Threads) column. The table lists various processes with their thread counts:

名稱	PID	狀態	CPU	執行緒	記憶體 (私人工作集)	描述
System	4	執行中	00	203	284 K	NT Kernel & System
sqlservr.exe	2428	執行中	00	73	11,964 K	SQL Server Windows NT - 64 Bit
nvstreamsvc.exe	3320	執行中	00	60	3,668 K	NVIDIA Streamer Service
explorer.exe	7240	執行中	00	53	49,516 K	Windows 檔案總管
Dropbox.exe	2620	執行中	00	47	75,908 K	Dropbox
chrome.exe	4516	執行中	00	38	131,552 K	Google Chrome
svchost.exe	1152	執行中	00	35	39,664 K	Windows Services 的主機處理程序
mqsvc.exe	2404	執行中	00	32	1,896 K	Message Queueing Service
MsMpEng.exe	3468	執行中	00	30	92,304 K	Antimalware Service Executable

從圖中可以看出，系統核心所建立的執行緒共有 203 個，SQL Server 有 73 個，連檔案總管都有 53 條執行緒！挺驚人的，不是嗎？我們不禁要擔心這些應用程式會不會過度使用執

行緒了。畢竟，閒置的執行緒仍會產生多餘的執行緒切換動作，而且還占用一些記憶體空間。

所幸目前市場上，多 CPU、超執行緒（hyperthreaded）CPU、或多核心（multi-core）CPU 的硬體架構已經非常普遍。在這些擁有多顆 CPU 或多核心的機器上跑多執行緒的應用程式時，前面提到的分時多工、輪流服務的情況可獲得大幅改善。這是因為 Windows 會為每一個 CPU 核心配給不同的執行緒，讓這些執行緒能夠真正地同時執行。當然了，在執行緒數量大於 CPU 數量的時候（幾乎都是這樣），每顆 CPU 內部還是會發生切換執行緒的情形。

1.3 爭先恐後—關於優先順序

既然切換執行緒在所難免，那麼對個別執行緒來說，能夠優先分配到更多 CPU 資源的，執行效能自然比較好。



如果你需要讓你的應用程式獲得更高的優先權，本節的內容會有一些幫助。若不需要，則可以先略過不讀。

Windows 作業系統把執行緒的優先順序分成 32 個等級，編號從最低的 0 至最高的 31，優先權愈高，愈能分到更多 CPU 時間。進一步說，當 Windows 要決定把 CPU 分給誰的時候，會先看看目前有沒有優先等級 31 的執行緒正在等候安排 CPU，若有，就會把 CPU 分給它一段時間。等它跑完配給的時間後，系統會把 CPU 分給其他同樣是優先等級 31 的執行緒。如果目前已經沒有優先等級 31 的執行緒在等待分配資源，才會輪到優先等級 30 的執行緒，然後是 29、28……；依此類推。

零頁執行緒

Windows 作業系統啓動時會建立一條特殊的執行緒，叫做「零頁執行緒」（zero page thread）。這條執行緒的優先等級是 0，而且整個系統當中也就只有它的優先等級是 0。換言之，應用程式的執行緒優先等級不可能為 0。

想像一群嗷嗷待哺的雛鳥，個個伸長了脖子張大了口等鳥媽媽餵食，但鳥媽媽卻偏愛其中一隻，只管餵牠。這樣下去，除非那隻受到特別關愛的雛鳥吃飽了，否則其他兄弟姊妹就只有捱餓的份。在 Windows 系統中，低優先等級的執行緒也會發生同樣的狀況，也叫做捱餓（starvation）。多 CPU（多核心）的機器能夠減少執行緒捱餓的機會，因為不同優先等級的執行緒可以同時分配給不同的 CPU。

還有一種狀況：有個優先等級 15 的執行緒幸運分配到一段 CPU 時間，可是才執行到一半，就出現另一個更高優先等級的執行緒；此時系統會立刻暫停較低優先的執行緒，並將 CPU 分配給較高優先的執行緒。用剛才的「鳥比喻」來說，就是：有隻雛鳥幸運分到食物，才剛咬幾口還沒吞下，就被鳥媽硬生生奪回，拿去餵另一隻更重要的雛鳥了。

之所以說「幸運分配到」，是因為我們無法精確指定或得知某執行緒究竟何時分配到 CPU，以及分配到多久的時間——這些完全由 Windows 作業系統來控制。我們能控制的，是藉由調整執行緒的優先等級來提高（或降低）執行緒獲得 CPU 資源的機會。

可是，優先等級共 32 級（若零頁執行緒專用的等級 0 不算則為 31 級），該如何決定哪些執行緒要用等級 2、5、12、還是 31 呢？為了簡化此問題，微軟用兩個條件的組合來決定執行緒的優先等級：處理序的優先順序類別（priority class），以及執行緒的優先順序。

1.3.1 處理序的優先順序

處理序的優先順序分成以下六種：

- 即時（RealTime）
- 高（High）
- 高於標準（Above Normal）
- 標準（Normal）
- 低於標準（Below Normal）
- 閑置（Idle）

預設的處理序優先順序是「標準」。應用程式應該只在真有必要時才用「高」優先類別，例如非關 I/O、執行時間短的處理序。至於「即時」優先類別則更應盡量避免，因為它的優先權極高，高到會影響作業系統的正常運作，例如干擾磁碟讀寫或網路傳輸，以及延遲鍵盤與滑鼠輸入的反應（使用者可能會以為系統當掉了）。總之，若無正當理由，最好別輕易調高應用程式的優先順序。

.NET Framework 的 `System.Diagnostics.ProcessPriorityClass` 列舉型別定義了處理序的優先順序。以下程式片段示範如何將目前處理序的優先順序類別設定為「高」：

```
1 var p = System.Diagnostics.Process.GetCurrentProcess();
2 p.PriorityClass = System.Diagnostics.ProcessPriorityClass.High;
```

此外，我們也可以利用 Windows 工作管理員來手動調整特定處理序的優先順序，如下圖所示：



了解處理序的優先等級之後，接著來看執行緒的優先等級。

1.3.2 執行緒的優先順序

Windows 提供七種執行緒優先順序：閒置 (Idle)、最低 (Lowest)、低於正常 (Below Normal)、正常 (Normal)、高於正常 (Above Normal)、最高 (Highest)、時間緊迫 (Time-Critical)。六種處理序優先順序類別搭配七種執行緒優先順序，便能決定執行緒最終的優先等級。參考下圖：

處理序優先順序 執行緒優先順序	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

舉例來說，若某處理序的優先順序類別為 Normal，而該處理序中的某個執行緒的優先順序為 Above Normal，則該執行緒的實際優先等級為 9。處理序優先順序類別若為 Realtime，則其中的執行緒優先等級最起碼為 16。

在 .NET Framework 中，`ThreadPriority` 列舉型別定義了執行緒的優先順序。以下程式片段即示範了如何設定執行緒的優先順序：

```
1 var t = new Thread(() => { Console.WriteLine("in worker thread"); });
2 t.Priority = ThreadPriority.Highest; // 設定成最高優先的執行緒。
```

其中的 `new Thread()` 語法會建立一條新的執行緒，其相關細節會在下一章進一步說明。

值得一提的是，.NET Framework 的 `ThreadPriority` 列舉型別僅定義了五種優先順序，缺了兩個：Idle 和 Time-Critical。為什麼 .NET 不提供這兩種執行緒優先順序呢？Jeffrey Richter 在他的《CLR via C#》中解釋：「如同 Windows 保留優先等級 0 和即時（real-time）等級給自己，.NET CLR 也保留了 Idle 和 Time-Critical 這兩種優先權給自己使用。」

Windows 市集應用程式

以上討論並不適用於 Windows 市集應用程式，因為這類應用程式既無法變更處理序的優先順序類別，也無法變更執行緒的優先順序。此外，當某個 Windows 市集應用程式從前景退居幕後成為背景程式時，Windows 會自動懸置該應用程式的所有執行緒。這麼做有兩個目的，一是避免背景應用程式拖慢前景應用程式的反應速度，讓使用者操作時更加流暢；二是減少 CPU 的負荷，從而節省電力耗損，提高電池的續航力。

1.4 並行、平行、非同步

結束本章前，有幾個與非同步程式設計有關的名詞挺容易混淆，需要先說明一下。

- 並行（**concurrency**）：一次處理多件工作。例如：使用者一邊輸入文字，應用程式在背後一邊執行拼字檢查。
- 多執行緒（**multithreading**）：特別指以多執行緒的方式來實現並行（concurrency）。
- 平行處理（**parallel processing**）：把工作切分成多個小單位，並分別交給多條執行緒來同時執行。
- 非同步處理（**asynchronous processing**）：是並行（concurrency）的一種形式，但不必然（甚至會避免）使用執行緒，而是採用「承諾」（**promise**）或回呼事件（**callback event**）的方式來達到並行的效果。

所謂的「承諾」（**promise**），或者說「未來」（**future**），指的是「將來一定會完成的某些操作」。例如 .NET 的 `Task` 和 `Task<TResult>` 類別都體現了「承諾」這個概念（後續章節會介紹 `Task` 類別的用法）。

進一步解釋，非同步程式設計的一個關鍵概念是：「非同步工作」(**asynchronous task**) 會在將來的某個時間點執行完畢，而在執行這些非同步工作時，它們並不會擋住(block)當前的執行緒；換言之，起始這些非同步操作的那一條執行緒仍能夠繼續執行，等到非同步操作完成時，便會透過「完成回呼事件」(completion callback event) 或所謂的 **promise/future** 來通知應用程式。

上面這段文字，第一次閱讀時可能不容易懂。沒關係，這裡只是先讓您對這些術語有個印象而已。在往後的章節中，我們會繼續討論非同步工作的相關細節。

1.5 本章回顧

讀完本章，您已經知道：

- 非同步程式設計的目的有二：一是提升應用程式的回應速度（尤其是對 UI 操作的回應），讓使用者在操作應用程式時不會老是覺得卡住。二是提升應用程式的整體執行效能；這也意味著多執行緒應用程式往往有較佳的使用者體驗，而且更能善用 CPU 的强大運算能力。
- 處理序 (process) 與執行緒 (thread) 的關係。
- 與執行緒有關的幾個基本概念，包括：執行緒時間片段 (thread quantum)、執行緒捱餓 (starvation)、context switch 等等。
- 「處理序優先順序類別」和「執行緒優先順序」只是用來簡化執行緒優先等級的設定，實際上 Windows 只會依執行緒的優先等級作出相應處置，而不會有「調整處理序優先順序」的動作。換言之，作用的對象是執行緒，不是處理序。
- 建立和摧毀執行緒都需要額外的成本，不是只要碰到效能不好的地方就一律建立新執行緒來處理。
- 非同步 (asynchrony) 是並行 (concurrency) 的一種形式，通常是透過 **future** 或 **callback event** 的方式來達到並行的效果，以盡量避免建立新的執行緒。

直接操控執行緒只是非同步程式設計的其中一種作法，而且通常不是最有效率的作法。本書後續章節將會陸續介紹幾種非同步程式設計的寫法。

2. .NET 非同步 API 概覽



這是預覽版的內容。

如上一章結束前提到的，直接操控執行緒並不是非同步程式設計的唯一方法，甚至不是最佳方法。打從.NET 1.x 開始就已經提供非同步 API (Application Programming Interface)，此後持續演進，在底層框架、模式、和語法方面都有逐步改進，並衍生出新的 API。本章將概略介紹.NET 非同步 API 的各種模式與寫法，包括直接建立執行緒（又稱為「建立專屬執行緒」）、執行緒集區（thread pool）、以及 APM (Asynchronous Programming Model) 和 EAP (Event-based Asynchronous Pattern)。當然，還有比較新的、目前建議使用的 TAP (Task-based Asynchronous Pattern) 以及 C# 為了支援 TAP 所增加的 `async` 和 `await` 關鍵字的用法。

在剛才提到的幾種非同步 API 寫法當中，專屬執行緒（2.1 節）在本章占有較大比重，而其他 API 只是蜻蜓點水般的粗淺介紹。這是因為，執行緒（thread）仍然是非同步程式設計的基本概念之一，了解其用法亦有助於學習其他非同步 API。不過，本書從下一章開始會把焦點放在比較新的、優先建議使用的 TAP 與 `async` 和 `await` 寫法，而且若非必要，將不再提及專屬執行緒或其他比較早期的 API，包括官方已明確表示不建議在新專案中使用的 APM 和 EAP¹。



本章範例程式的原始碼位置：

<https://github.com/huanlin/async-book-support> 裡面的 Examples/ch02 資料夾。

2.1 專屬執行緒

上一章提過，建立執行緒會產生一些額外負擔，包括作業系統的核心物件、堆疊空間、context switch 等等。儘管如此，它在某些場合仍有用處，特別是需要長時間執行的背景工作（本章稍後會進一步說明專屬執行緒的使用時機）。

為了與其他非同步程式設計模型有所區別，本書採用 Jeffrey Richter 在《CLR via C# 4th Edition》中的用詞「專屬執行緒」（dedicated thread）來指稱這種直接建立一條執行緒來專門執行某件工作的作法。

¹<http://msdn.microsoft.com/en-us/library/jj152938.aspx>

在 .NET Framework 中，用來操控專屬執行緒的類別是 `System.Threading.Thread`。在介紹此類別之前，有個名詞得先解釋一下：主執行緒（main thread）。

每個應用程式運行時都是有一條預設的執行緒，稱為「主執行緒」（**main thread**）。對於桌面應用程式來說，主執行緒通常也是負責處理使用者介面的執行緒，故有時也說「UI 執行緒」（UI thread）。

接著就來看 `Thread` 類別的一些基本用法。



Windows 市集應用程式無法使用 `System.Threading.Thread` 類別。

2.1.1 建立與啟動執行緒

底下是個測試多執行緒的簡單範例，示範如何建立一條執行緒來執行某件非同步工作。

```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6      static void Main(string[] args)
7      {
8          Thread t1 = new Thread(MyTask);
9          t1.Start();
10
11         for (int i = 0; i < 500; i++)
12         {
13             Console.Write(".");
14         }
15     }
16
17     static void MyTask()
18     {
19         for (int i = 0; i < 500; i++)
20         {
21             Console.Write("[" + Thread.CurrentThread.ManagedThreadId + "]");
22         }
23     }
24 }
```

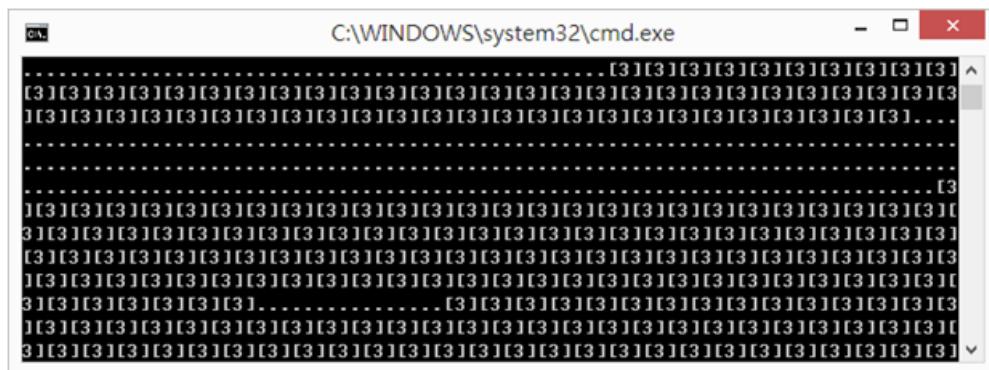


此範例程式的專案名稱：**Ex01_ThreadStart.csproj**。

程式說明：

- 使用 `System.Threading.Thread` 類別來建立執行緒物件，同時將一個委派方法 `MyTask` 傳入建構函式。這個委派方法將於該執行緒開始運行時被自動呼叫。
- 呼叫執行緒物件的 `Start` 方法，令執行緒開始運行，亦即在這個工作執行緒中呼叫 `MyTask` 方法。
- `Main` 函式開始一個迴圈，持續輸出「.」。這只是為了識別哪些文字是由主執行緒輸出，哪些是由工作執行緒輸出。
- `MyTask` 函式也有一個迴圈，持續輸出目前執行緒的編號。

下圖為此範例程式的執行結果：



從輸出結果可以看得出來，主執行緒跑了一段時間，切換至我們另外建立的工作執行緒。工作執行緒也同樣跑了一段時間之後，又切回主執行緒，如此反覆切換，直到兩個執行緒的迴圈結束為止。

建立 `Thread` 物件時，傳入建構函式的委派有兩種版本。一種是 `ThreadStart`，另一種是 `ParameterizedThreadStart`。以下是這兩種委派型別的宣告：

```
1 public delegate void ThreadStart();
2 public delegate void ParameterizedThreadStart(Object obj);
```

前述範例使用的是第一種，也就是不需要傳入參數的 `ThreadStart` 委派型別。如果在啓動工作執行緒時需要額外傳入一些資料，就可以使用第二種委派型別：`ParameterizedThreadStart`。參考以下範例：

```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6      static void Main(string[] args)
7      {
8          Thread t1 = new Thread(MyTask);
9          Thread t2 = new Thread(MyTask);
10         Thread t3 = new Thread(MyTask);
11
12         t1.Start("X");
13         t2.Start("Y");
14         t3.Start("Z");
15
16         for (int i = 0; i < 500; i++)
17         {
18             Console.Write(".");
19         }
20     }
21
22     static void MyTask(object param)
23     {
24         for (int i = 0; i < 500; i++)
25         {
26             Console.Write(param);
27         }
28     }
29 }
```

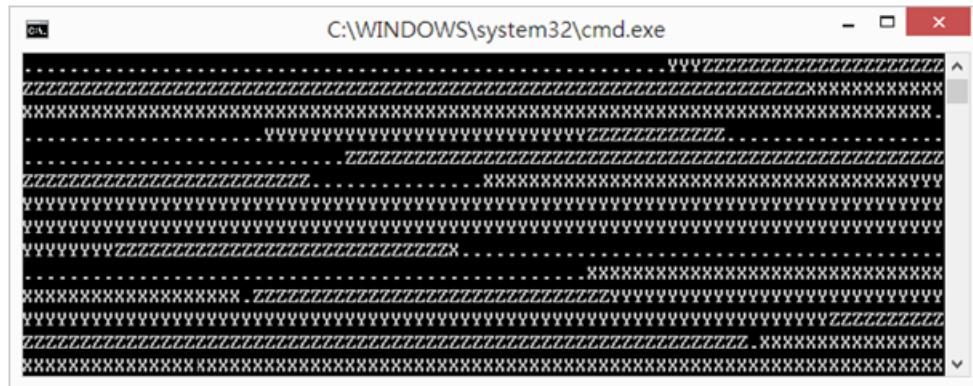


此範例程式的專案名稱：**Ex02_ParamThreadStart.csproj**。

程式說明：

- 首先建立三個執行緒物件，而且這三個執行緒都會執行同一項任務：MyTask。
- MyTask 方法需要傳入一個 object 型別的參數，而此參數的值是在啓動執行緒時傳入。在啓動三個執行緒物件時，我分別傳入了“X”、“Y”、“Z”，以便從輸出結果中觀察各執行緒輪流切換的情形。

執行結果：



2.1.2 等待與暫停執行緒

Thread 類別有個 `IsAlive` 屬性，代表執行緒是否正在運行。一旦呼叫執行緒物件的 `Start` 方法令它開始執行，其 `IsAlive` 屬性值就會等於 `true`，直到該執行緒的委派方法執行完畢，那條執行緒便隨之結束。因此，如果想要等待某執行緒的工作執行完畢才繼續處理其他工作，用一個迴圈來持續判斷執行緒物件的 `IsAlive` 屬性就能辦到。

還有一個更簡單的作法可以等待執行緒結束：呼叫 `Thread` 物件的 `Join` 方法。參考以下範例：

```
1  using System;
2  using System.Threading;
3
4  namespace Ex03_ThreadJoin
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Thread t1 = new Thread(MyTask);
11             Thread t2 = new Thread(MyTask);
12             Thread t3 = new Thread(MyTask);
13
14             t1.Start("T1");
15             t2.Start("T2");
16             t3.Start("T3");
17
18             t1.Join();
19             t2.Join();
20             t3.Join();
21
22             Console.ReadKey();
23     }
```

```
24
25     static void MyTask(object param)
26     {
27         Console.WriteLine("{0} 已開始執行 MyTask()", param);
28         Thread.Sleep(3000); // 令目前這條執行緒暫停三秒。
29         Console.WriteLine("{0} 即將完成工作", param);
30     }
31 }
32 }
```



此範例程式的專案名稱：**Ex03_ThreadJoin.csproj**。

說明：

- 在 Main 函式中，先起始三條執行緒，然後逐一呼叫它們的 Join 方法——這會令主執行緒依序等待 t1、t2、t3 執行完畢之後才繼續執行底下的程式碼。
- 此範例還用到了 Thread.Sleep 方法。此方法會令目前所在的執行緒休息一段指定的時間，時間單位是毫秒（millisecond）。Thread.Sleep 方法也常被用來模擬應用程式正在忙著處理某件工作而暫時無法回應其他請求。

執行結果如下圖：

```
C:\WINDOWS\system32\cmd.exe
T1 已開始執行 MyTask()
T2 已開始執行 MyTask()
T3 已開始執行 MyTask()
T1 即將完成工作
T2 即將完成工作
T3 即將完成工作
```

2.1.3 共享變數

理想情況下，各執行緒分頭進行，互不干涉，程式碼寫起來比較單純。但實務上，執行緒之間卻經常需要存取共享的資源或變數，這就產生了一些麻煩。



圖片來源: <http://goo.gl/lQ4qN0>

更明確地說，多條執行緒之間共享同一個變數時，如果都只是讀取變數值，並不至於有太大的問題。然而，如果有許多條執行緒會去修改共享變數的值，那就得運用一些技巧來避免數值錯亂的情形。看看底下這個範例：

```
1 class Program
2 {
3
4     static void Main(string[] args)
5     {
6         new SharedStateDemo().Run();
7         Console.ReadLine();
8     }
9 }
10
11 public class SharedStateDemo
12 {
13     private int itemCount = 0;    // 已加入購物車的商品數量。
14
15     public void Run()
16     {
17         var t1 = new Thread(AddToCart);
18         var t2 = new Thread(AddToCart);
19
20         t1.Start(300);
21         t2.Start(100);
22     }
23 }
```

```

24     private void AddToCart(object simulateDelay)
25     {
26         itemCount++;
27
28         /*
29          * 用 Thread.Sleep 來模擬這項工作所花的時間，時間長短
30          * 由呼叫端傳入的 simulateDelay 參數指定，以便藉由改變
31          * 此參數來觀察共享變數值的變化。
32         */
33         Thread.Sleep((int)simulateDelay);
34         Console.WriteLine("Items in cart: {0}", itemCount);
35     }
36 }
```

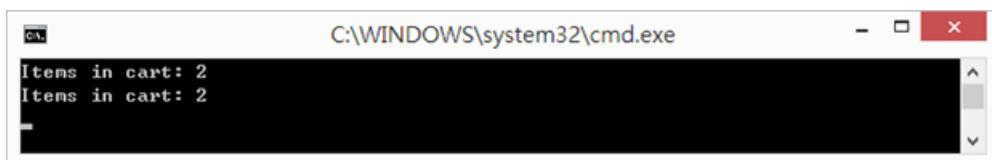


此範例程式的專案名稱：**Ex04_SharedState.csproj**。

程式說明：

- Main 函式會建立 SharedStateDemo 物件並呼叫其 Run 方法。此範例的重點在 SharedStateDemo 類別裡面，示範的情境為購物車。
- SharedStateDemo 類別有一個整數欄位：itemCount，代表已加入購物車的商品數量。此變數將作為執行緒之間共享的變數。
- SharedStateDemo 類別的 Run 方法會建立兩條執行緒，它們的工作都是呼叫 AddCart 方法，代表「加入購物車」的動作。
- AddCart 方法需要傳入一個參數，用來模擬每一次加入購物車的動作需要花多少時間。從 Run 方法的程式碼可以看得出來，我刻意讓第一條執行緒花比較多時間（延遲 300 毫秒）。

執行結果：



如果 t_1 和 t_2 這兩條執行緒是依照它們啓動的順序先後完成任務，執行結果的第一列所顯示的購物車商品數量應為 1，第二列的數量才是 2。可是現在卻全都是 2，這是因為 t_1 先啓動，進入 AddCart 函式之後，把 itemCount 加一，然後進入一段模擬長時間工作的延遲 (300ms)。由於此時 t_2 已經啓動了，也把 itemCount 加一了（其值為 2），然後也進入一段延遲 (100ms)。但由於 t_2 的延遲時間較短，比 t_1 更快執行完畢（後發而先至），因此執行結果畫面中的第一列文字其實是由執行緒 t_2 輸出的。接下來， t_1 也跑完了，但此時的 itemCount 已經被 t_2 改成了 2，所以輸出的結果自然就一樣了。

在我的機器上，即使呼叫 `t1.Start()` 時傳入 1（僅延遲 1 毫秒），輸出結果仍舊相同，並不因為 `t1` 的模擬延遲時間縮短成 1 毫秒而產生先 1 後 2 的結果。我想這是因為我的機器有多核心 CPU，於是 `t1` 才剛啓動，將 `itemCount` 遞增為 1，此時 `t2` 實際也已經（由另一個 CPU 核心）啓動了，`itemCount` 便遞增為 2。

不過，如果改成 `t1.Start(0)`，亦即令 `t1` 模擬延遲的時間為 0 毫秒，執行結果可能就會變成先 1 後 2 了（不見得每一次執行結果都一樣，視機器而定）。這是因為 `Thread.Sleep(0)` 完全沒有延遲的作用，故來得及在其他執行緒進入該程式區塊之前完成工作。

有時候，這種多條執行緒共同修改一個變數的情況可能會導致嚴重問題。比如說，當應用程式正在計算某員工的薪資，才處理到一半，還沒算完呢，又有其他執行緒修改了共享的薪資計算參數，可能原本的計算結果應該是 63,000，結果卻成了 59,000。

接著就來看看如何解決這個問題，讓此範例的執行結果顯示的商品數量變成先 1 後 2，而不是兩次都輸出 2。

2.1.4 執行緒同步化

剛才展示的多條執行緒修改同一變數所衍生之變數值錯亂的問題，有點像是很多人同時伸手搶一塊餅——很容易把餅給抓爛了。解決方法說來簡單，就是排隊。也就是說，原本以非同步執行的各條執行緒，碰到了要修改共享變數的時候，都要乖乖排隊，一個做完了才換下一個。這等於是暫時切換成同步執行的方式，如同在八線道的公路某處設下關卡，將道路限縮成單線道，只許一輛汽車通行；等車輛駛出關卡，前方又是一片開闊，任憑奔馳。

這種迫使多條執行緒從非同步暫時切換成同步執行的技巧，叫做執行緒同步化（**thread synchronization**）。

2.1.4.1 鎖定

執行緒同步化的技巧有很多種，這裡要示範的是以 C# 的 `lock` 敘述來建立獨佔鎖定（exclusive lock）的程式區塊，迫使各執行緒在進入特定程式碼區塊時乖乖排隊，以達到同步化的效果。也就是說，`lock` 可以把某程式碼區塊——而不是整個函式或整個類別——變成同時間只允許一個執行緒進入的「單線道」。



使用獨佔鎖定的技巧時，應注意避免兩條執行緒互相等待對方釋放鎖定而導致鎖死（deadlock）的情形。

只要稍微修改上一個範例的 `SharedStateDemo` 類別，輸出結果就會不同。底下是修改後的程式碼：

```
1 public class SharedStateDemo
2 {
3     private int itemCount = 0;
4     private object locker = new Object(); // 用於獨佔鎖定的物件
5
6     public void Run()
7     {
8         var t1 = new Thread(AddToCart);
9         var t2 = new Thread(AddToCart);
10
11         t1.Start(300);
12         t2.Start(100);
13     }
14
15     private void AddToCart(object simulateDelay)
16     {
17         Console.WriteLine("Enter thread {0}", // 顯示目前所在的執行緒編號
18                         Thread.CurrentThread.ManagedThreadId);
19         lock (locker) // 讓底下這個程式區塊變成同時間只允許一條執行緒進入。
20         {
21             itemCount++;
22
23             Thread.Sleep((int)simulateDelay);
24             Console.WriteLine("Items in cart: {0} on thread {1}",
25                               itemCount, Thread.CurrentThread.ManagedThreadId);
26         }
27     }
28 }
```

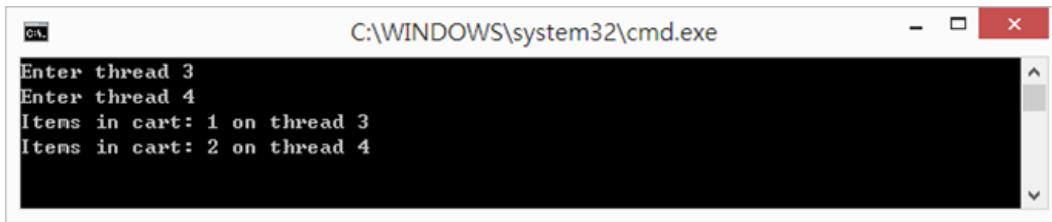


此範例程式的專案名稱：**Ex05_Lock.csproj**。

程式說明：

- 類別中多了一個型別為 `Object` 的私有成員：`locker`。此物件是用來作為獨佔鎖定之用，可以是任何參考型別。
- `AddCart` 函式中增加了 `lock` 敘述。當兩條執行緒同時爭搶同一個鎖定物件時，其中一條執行緒會被擋住，等到被鎖定的物件被先前搶到的執行緒釋放了，才能夠取得鎖定。如此便能夠確保以 `lock` 關鍵字包住的程式區塊在同一時間內只會有一條執行緒進入。

這次除了增加獨佔鎖定的程式敘述，還把執行緒編號也一併秀出來，方便確認。執行結果如下圖所示：



圖中可以看出，執行緒編號 3 和 4 都已分別啓動了，但是購物車的數量會依兩條執行緒的順序各自遞增一次，並顯示正確的結果。像這種有加上保護機制來避免多執行緒爭搶共用變數而致資料錯亂的程式寫法，我們說它是「執行緒安全的」(**thread-safe**)。如果你看到某些元件或類別庫宣稱它們是「執行緒安全的」，那就表示它們在設計時便已經考慮到多執行緒的環境。

其他同步化技巧

如果你需要了解其他同步化技巧，可上網搜尋以下類別的教學文件和範例：

- [Mutex](#)
- [SemaphoreSlim](#)
- [AutoResetEvent](#)
- [ManualResetEventSlim](#)
- [CountDownEvent](#)
- [Barrier](#)
- [ReaderWriterLockSlim](#)
- [SpinWait](#)

請注意上列以「Slim」結尾的類別，在 .NET Framework 裡面還有提供非「Slim」結尾的類別，例如 `ReaderWriterLock`。這些名稱以「Slim」結尾的是比較新的類別，除了更輕量，也更少發生鎖死（deadlock）的情況，故效能通常也比非“Slim”結尾的舊版類別更好。

2.1.5 前景執行緒 vs. 背景執行緒

在 .NET 中，有所謂的前景執行緒和背景執行緒。兩者的主要區別是：當某個應用程式中所有的前景執行緒都停止時，CLR 會停止該應用程式的所有背景執行緒（而且不會拋出任何異常），並結束應用程式。若只是停止背景執行緒，則不會造成應用程式結束。因此，我們通常會把那些一定要執行完畢的工作交給前景執行緒，而將比較不重要的、或者可以隨時中斷再接續進行的工作交給背景執行緒來處理。

預設情況下，新建立的執行緒皆為前景執行緒，但你可以透過 `Thread` 物件的 `IsBackground` 屬性來將它改成背景執行緒。參考以下範例：

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         Thread t = new Thread(MyTask);
6         t.IsBackground = true;
7         t.Start();
8
9         // 若 t 是前景執行緒，此應用程式不會結束，除非手動將它關閉；
10        // 若 t 是背景執行緒，此應用程式會立刻結束。
11    }
12
13    static void MyTask()
14    {
15        while (true)
16            ;
17    }
18 }
```



此範例程式的專案名稱：**Ex06_BackgroundThread.csproj**。

程式說明：

- 此範例程式在 `Main` 函式中建立一條新的執行緒之後，將它設定為背景執行緒，並令它開始執行。
- 接著 `Main` 就結束了，這表示前景執行緒結束了。因此就算 `MyTask` 函式仍在跑無窮迴圈，應用程式仍會立刻結束。若把 `t` 設定為前景執行緒（預設值），則 `Main` 函式結束之後，應用程式並不會結束，除非手動將它關閉。

2.1.6 使用專屬執行緒的時機

當你碰到以下幾種特殊場合，才應該考慮使用 `new Thread()` 這種建立專屬執行緒的方式來處理非同步工作：

- 欲執行的工作需要花較長時間才能執行完畢（例如 10 分鐘以上）。
- 你希望某些執行緒擁有特殊優先權（若無正當理由，不建議這麼做）。預設情況下，執行緒的優先權是「正常」等級。如果想要讓某執行緒擁有特權，則可以個別建立執行緒並修改其優先權。
- 你希望某些執行緒以前景執行緒的方式運作，以避免工作還沒完成，應用程式就被使用者或其他程序關閉。執行緒集區（於下一節介紹）裡面的執行緒永遠都是背景執行緒，它們有可能還沒完成任務就被 CLR 結束掉。

- 執行緒開始工作後，你可能需要在某些情況下提前終止執行緒（透過呼叫 `Thread` 類別的 `Abort` 方法）。

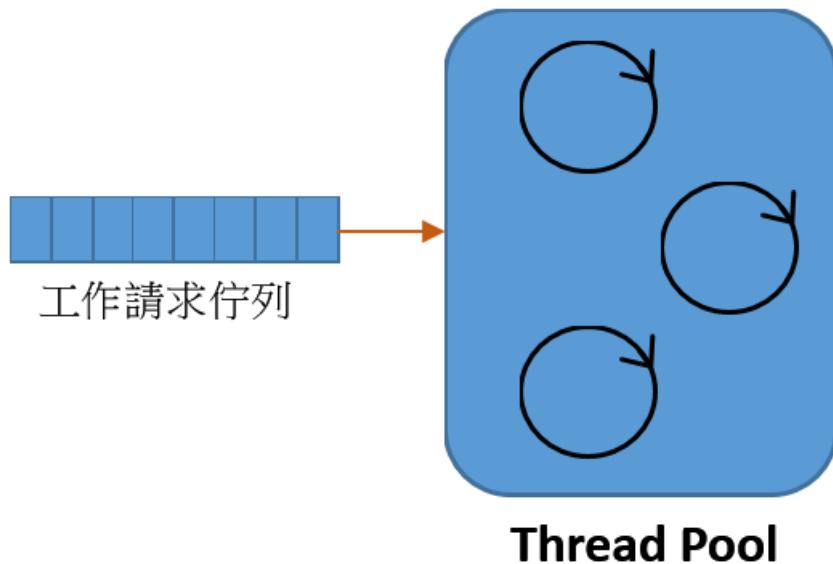
2.2 執行緒集區

如第 1 章提過的，建立執行緒需要付出額外成本，而頻繁地建立與摧毀執行緒，則是一種沒效率的資源運用方式，甚至可能影響效能。因此，.NET CLR 實作了集區（pool）的概念，讓應用程式可將已完成任務的執行緒丟進集區裡面待命，等到有其他工作需要以非同步方式執行，便可透過集區中閒置的執行緒來負責執行工作。簡單地說，執行緒集區就是一種重複使用執行緒的機制。

2.2.1 執行緒集區的運作方式

一般而言，每一個 .NET 應用程式都擁有一個自己專屬的執行緒集區。若要說得更精確些，按照 Jeffrey Richter 在《CLR via C# 第四版》中的說法是：「每一個 CLR 有一個執行緒集區，而且那個 CLR 管理的所有 App Domains 都會共享這個執行緒集區。如果一個應用程式會同時載入多個 CLR，則每一個 CLR 都有它自己的執行緒集區。」

執行緒集區本身有一個工作請求佇列，每當應用程式需要非同步操作時，便可呼叫特定 API 來將工作請求送進這個佇列。CLR 會從佇列中逐一取出請求（先到先服務），並查看集區裡面有沒有閒置的執行緒。由於 CLR 初始化時，其執行緒集區是空的，於是 CLR 會建立一條新的執行緒來負責執行任務。等到任務執行完畢，CLR 並不摧毀那個執行緒，而是將它放回執行緒集區，等待下一次任務指派。如此一來，就如前面所說，執行緒能夠重複使用，從而減少了反覆建立和摧毀執行緒所產生的效能損耗。下圖簡略描繪了執行緒集區的運作方式。



另一方面，集區中的執行緒在閒置一段時間之後若未再接到新任務，就會自動摧毀，以便將記憶體資源釋放出來。除了摧毀閒置的執行緒，CLR 還有一套演算法，會根據系統目前擁有的運算資源（CPU 核心的數量）和應用程式的負載等狀況來決定是否要建立更多執行緒來處理應用程式提出的工作請求。比如說，當 CLR 發現目前工作佇列中排隊等待的工作迅速增加，以至於集區中的執行緒數量來不及消化時，便會依內定的演算法來決定是否要加入新的執行緒至集區中。

2.2.2 執行緒集區的大小限制

CLR 的執行緒集區大小是有上限的——總不可能因為工作佇列突然湧進兩萬件工作，就讓集區裡面同時擠滿兩萬條執行緒吧？

在 .NET 1.0 時代，執行緒集區大小的預設上限是 25，亦即執行緒集區最多只能有 25 條執行緒。在此如此拮据的環境下，執行緒集區很容易出現「無兵可用」的窘境，連帶影響應用程式的效能。因此，在 .NET 版本的演進過程中，微軟便逐漸提高執行緒集區大小的上限，以及改善執行緒集區的效率。

到了 .NET 3.5，執行緒集區的大小限制改為每個 CPU 核心最多有 250 條執行緒；比如說，在四核心的機器上，執行緒集區最多可有 1000 條執行緒。然而，每一條執行緒大約要占用 1MB 的記憶體，而記憶體是珍貴的資源，因此 .NET 4.0 又進一步改善，會根據機器的記憶體大小來決定執行緒集區的上限，而此動態決定的上限對於大多數應用程式來說已經綽綽有餘——通常最多可有 1023 條工作執行緒，以及 1000 條 I/O 執行緒（下一節會說明這兩種執行緒的用途）。

2.2.3 工作執行緒與 I/O 執行緒

由 CLR 管理的執行緒集區有兩種：工作執行緒集區（**worker thread pool**）和輸入／輸出執行緒集區（**I/O thread pool**）。工作執行緒集區負責執行與 CPU 運算有關的工作，I/O 執行緒則專用來處理 I/O 操作（例如讀寫檔案、網路 I/O、資料庫處理等等）。其實這兩種集區裡面的執行緒都是同樣的東西，只是兩種集區在實作上採用了不同的演算法，以便更有效率地運用系統資源，並提升執行效能。

Windows 如何處理 I/O 操作

舉例來說，假設你的應用程式使用了 `File.ReadAllText()` 方法來讀取檔案內容；請注意這是個同步的（*synchronous*）方法。當應用程式執行到這個方法時，Windows 作業系統核心會起始一個檔案 I/O 操作，而當此 I/O 操作正在執行時，應用程式目前的執行緒便沒事可做，只能等待那個 I/O 操作完成，因此，Windows 會先讓那條執行緒進入休眠狀態，以免它閒閒沒事卻占用 CPU 時間。這樣雖然節省了時間（讓 CPU 把時間分配給其他執行緒），可是卻沒有節省到空間——執行緒雖然暫時休眠，但它仍然占著記憶體空間。此外，如果是 UI 類型的應用程式（例如 Windows Forms），主執行緒在休眠期間完全無法回應使用者的操作，使用者也就只能等待。

現在假設你改用 `StreamReader.ReadAsync()` 方法，以非同步 I/O 的方式讀取檔案。當應用程式執行到這個方法時，當前的執行緒並不會進入休眠，而是立刻返回，並且繼續執行後續的程式碼。如此一來，便改善了剛才提到的缺點。那麼，應用程式如何取得結果呢？非同步方法 `ReadAsync` 會傳回一個 `Task<int>`，你可以用這個物件來取得該方法的執行結果。關於 `Task` 類別的用法，本書後面還會進一步介紹。

大致了解執行緒集區的運作方式與相關概念之後，接著就來看看程式的寫法。

2.2.4 使用執行緒集區

欲利用集區中的執行緒來執行特定工作——這裡專指牽涉 CPU 運算的工作（compute-bound tasks）——可以用 .NET 的 `ThreadPool` 類別的靜態方法：`QueueUserWorkItem`。其實從方法的名稱也可以看得出來，此方法所使用的集區是工作執行緒集區，而不是 I/O 執行緒集區。

`QueueUserWorkItem` 方法有兩種版本：

```
1 static Boolean QueueUserWorkItem(WaitCallback callBack);
2 static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);
```

呼叫此方法時，它會將你指定的「工作項目」（work item）加入執行緒集區的工作請求佇列，然後立即返回呼叫端。所謂的工作項目，也就是輸入參數 `callBack` 所代表的回呼函式，此函式的宣告（回傳值與參數列）必須符合 `System.Threading.WaitCallback` 委派型別，如下所示：

```
delegate void WaitCallback(Object state);
```

當 CLR 從執行緒集區中取出一條執行緒來執行佇列中的任務時，就會呼叫那個預先指定的回呼函式。如需提供額外參數給回呼函式，在呼叫 `QueueUserWorkItem` 時可透過參數 `state` 來傳遞。

底下是個簡單範例：

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         ThreadPool.QueueUserWorkItem(new WaitCallback(MyTask));
6
7         for (int i = 0; i < 500; i++)
8         {
9             Console.Write(".");
10        }
11    }
12
13    static void MyTask(object state)
14    {
15        for (int i = 0; i < 500; i++)
16        {
17            Console.Write("[" + Thread.CurrentThread.ManagedThreadId + "]");
18        }
19    }
20 }
```



此範例程式的專案名稱：**Ex07_ThreadPool.csproj**。



雖然 `ThreadPool` 類別有提供 `SetMinThreads` 和 `SetMaxThreads` 方法來改變集區大小的下限與上限，但是最好還是別任意使用，因為更改集區大小的預設值往往只會讓效能更糟——除非你非常清楚目前使用的 CLR 版本所實作的執行緒集區的內部運作細節。

除了 `ThreadPool.QueueUserWorkItem()` 之外，另外還有兩種作法也是透過執行緒集區來執行非同步工作：

- `System.Threading.Timer`: 適用於定期執行特定背景工作的場合。
- `Asynchronous Programming Model (APM)`: 請接著看下一節。

2.3 非同步程式設計模型 (APM)

APM (Asynchronous Programming Model) 是 .NET 1.1 時代的產物（意思是這節跳過不讀也無妨），另一個通俗的稱呼是「Begin/End 模式」。這是因為，APM 的程式寫法慣例，都會在類別中額外提供一組以 Begin* 和 End* 開頭來命名的方法來支援非同步呼叫。

比如說，.NET 的 System.IO 命名空間的 FileStream 類別，針對「讀取檔案內容」這項操作，它提供了同步呼叫的 Read 方法，和基於 APM 的非同步呼叫的版本：BeginRead 和 EndRead 方法。

先來看 Read 方法的宣告：

```
1 public int Read(byte[] array, int offset, int count)
```

以及它的範例：

```
1 static void DemoSync()
2 {
3     using (var fs = new FileStream(@"C:\temp\foo.txt", FileMode.Open))
4     {
5         byte[] content = new byte[fs.Length];
6         fs.Read(content, 0, (int)fs.Length);      // 一次讀取整個檔案的內容。
7     }
8 }
```

再來看基於 APM 的非同步版本，也就是 BeginRead 和 EndRead 方法：

```
1 public IAsyncResult BeginRead(
2     byte[] array, int offset, int numBytes, // 這些是 `Read` 方法原本就有的參數。
3     AsyncCallback userCallback,           // 非同步呼叫作業完成時呼叫的函式。
4     Object stateObject                 // 你可以透過此參數傳遞額外資訊。
5 )
6
7 public int EndRead(IAsyncResult asyncResult)
```

依 APM 的命名慣例，非同步方法的 Begin* 方法所需要傳遞的參數，必定是同步方法所需傳遞的參數再加上兩個參數： AsyncCallback userCallback 和 Object stateObject。你可以比較一下剛才的 Read 方法和 BeginRead 方法的參數列，便可發現這個規則。在 .NET Framework 中，只要是基於 APM 來設計的非同步方法，都具有這樣的特徵。

現在把先前的同步呼叫範例改成非同步呼叫的版本，如下所示：

```

1 static void DemoAsync()
2 {
3     using (var fs = new FileStream(@"C:\temp\foo.txt", FileMode.Open))
4     {
5         byte[] content = new byte[fs.Length];
6         IAsyncResult ar = fs.BeginRead(content, 0, (int)fs.Length, null, null);
7
8         Console.WriteLine(" 控制流程回到主執行緒，執行其他工作... ");
9         Thread.Sleep(1500); // 模擬執行其他工作需要花費的時間。
10
11        // 等到需要取得結果時，呼叫 EndRead 方法（會 block 當前執行緒）
12        int bytesRead = fs.EndRead(ar);
13        Console.WriteLine(" 一共讀取了 {0} bytes。", bytesRead);
14    }
15 }
```

簡單起見，這裡在呼叫 `BeginRead` 方法時，並未傳入一個 callback 函式，因此最後兩個傳入的參數都是 `null`（第 6 行）。



此範例程式的專案名稱：**Ex08_APM.csproj**。

由於 APM 的寫法已經不建議使用，故簡單介紹到此。本節就以整理 APM 的幾個缺點來結尾：

- 非同步的程式碼寫法跟一般循序執行的程式碼差異頗大，不直觀，也不好理解（例如 `IAsyncResult` 的用法）。
- 無論是否需要取得非同步工作的執行結果，你都必須呼叫 `EndXXX` 方法，以確保非同步工作結束前釋放它所佔用的任何資源。

2.4 基於事件的非同步模式（EAP）

在 APM 之後，.NET 2.0 加入了新的非同步寫法，叫做「基於事件的非同步模式」（Event-based Asynchronous Pattern），簡稱 **EAP**。EAP 的特色是每一項非同步操作都會有兩個成員：一個是用來起始非同步工作的方法，另一個則是工作完成時觸發的事件，方便我們從事件處理常式的參數來直接取得非同步工作的結果。

以 `System.Net.WebClient` 的 `DownloadString` 為例，從方法名稱看得出來，它就是個普通的同步方法。以下是使此方法的使用範例：

```
1 static void DemoSync()
2 {
3     using (var client = new WebClient())
4     {
5         string result = client.DownloadString(new Uri("http://huan-lin.blogspot.com"));
6         Console.WriteLine(" 下載的網頁內容長度為 {0} 字元。 ", result.Length);
7     }
8 }
```

DownloadString 方法的 EAP 非同步版本是 DownloadStringAsync，同時搭配 DownloadStringCompleted 事件。於是，剛才的範例程式可以改成以下的非同步版本：

```
1 static void DemoAsync()
2 {
3     using (var client = new WebClient())
4     {
5         client.DownloadStringCompleted += WebDownloadStringCompleted;
6         client.DownloadStringAsync(new Uri("http://huan-lin.blogspot.com"));
7
8         Console.WriteLine(" 控制流程回到主執行緒，執行其他工作... ");
9         Thread.Sleep(2000); // 模擬執行其他工作需要花費的時間。
10    }
11 }
12
13 static void WebDownloadStringCompleted(
14     object sender, DownloadStringCompletedEventArgs e)
15 {
16     // 可以在這裡撰寫更新 UI 的程式碼，而無須額外撰寫切換至 UI 執行緒的程式碼。
17     if (e.Cancelled)
18     {
19         Console.WriteLine(" 非同步工作已取消! ");
20     }
21     else if (e.Error != null)
22     {
23         Console.WriteLine(" 非同步工作發生錯誤： " + e.Error.Message);
24     }
25     else
26     {
27         Console.WriteLine(" 下載的網頁內容長度為 {0} 字元。 ", e.Result.Length);
28     }
29 }
```

程式說明：

- 第 5 行：先設定好非同步工作完成時要 callback 的事件處理常式。此步驟必須在呼叫非同步方法之前進行。
- 第 6 行：呼叫非同步方法 `DownloadStringAsync`。此方法一進入之後，就會在內部起始一個非同步工作（通常意味著建立一條新的執行緒），並且立刻返回呼叫端；等到那個非同步工作完成時，便會主動去呼叫先前預先設定好的事件處理常式。
- 第 13 行：非同步工作執行完畢時觸發的事件處理常式。在此函式中，我們可以透過事件參數的 `Cancel` 屬性來判斷非同步工作是否已取消，以及透過 `Error` 屬性來判斷非同步工作的執行過程是否發生錯誤。如果沒有取消也沒發生錯誤，便可透過 `Result` 屬性來取得非同步工作的結果（此例的 `Result` 是 `string` 型別）。



如果你有用過上一節介紹的 APM 寫法，是不是覺得 EAP 的寫法更直觀、更好理解？

此範例程式還有一個值得特別留意的地方：我們可以在非同步工作完成時觸發的事件處理常式中直接更新 UI（使用者介面），而無須撰寫額外的程式碼來切換回 UI 執行緒。這是因為當此事件觸發時，`WebClient` 會在背後判斷是否需要切換至 UI 執行緒；如果你的應用程式是有 UI 的（例如 Windows Forms 應用程式），它就會切換至 UI 執行緒來觸發 `DownloadStringCompleted` 事件。但請注意，EAP 只是個模式，所以這個自動切回 UI 執行緒的功能，必得由設計元件的人負責實作，而不是說，所有按照 EAP 來命名的非同步方法都會自動擁有這項功能。



此範例程式的專案名稱：**Ex09_EAP.csproj**。

如果你想要確認 `WebClient` 在觸發 `DownloadStringCompleted` 事件時真的有幫你切回 UI 執行緒，可以實際跑跑看範例程式專案 `Ex09_EAP_WinForms.csproj`。

2.5 基於工作的非同步模式 (TAP)

TAP 是 Task-based Asynchronous Pattern 的縮寫，亦即「基於工作（任務）的非同步模式」。繼 APM (Asynchronous Programming Model) 與 EAP (Event-based Asynchronous Pattern) 之後，TAP 成為微軟建議的.NET 非同步程式設計模式。



在討論 task 時，我會交替使用幾個意思相近的名詞，包括：工作、任務，操作、或作業。例如：非同步作業。

2.5.1 工作平行程式庫 (TPL)

TAP 這個非同步模式需要倚賴 .NET 4.0 提供的一組新的 API，叫做 Task Parallel Library (工作平行程式庫)，簡稱 TPL。這組 API 的相關類別是放在 System.Threading 和 System.Threading.Tasks 命名空間裡。

進一步說，TAP 的基礎是 System.Threading.Tasks 命名空間中的 Task 和 Task<TResult> 類別。這兩個類別都是用來代表非同步工作——Task<TResult> 繼承自 Task，可用於需要取得非同步工作之執行結果的場合；而 Task 是用在無需返回工作結果的場合。

可以這麼說：TPL 的設計理念是要把「非同步工作」這個抽象概念統一用一個叫做 Task 的類別來表示，並且提供一組相應的 API 來輔助。每一個 Task 物件即封裝了一項非同步執行的工作。這有點像是委派 (delegate) 的概念——委派不也封裝了一項任務嗎？兩者的差別在於，委派是以同步的 (synchronous) 方式執行，而 Task 是以非同步的方式來執行其封裝的工作。

2.5.1.1 TPL 如何執行工作？

一個由 Task 所代表的非同步工作是由「工作排程器」(task scheduler) 來決定其執行策略。代表工作排程器的類別是 System.Threading.Tasks.TaskScheduler。若有特殊需求，你也可以撰寫自訂的工作排程器（本書不會介紹這個部分）。

預設的工作排程器是以執行緒集區 (thread pool) 為基礎。也就是說，當你利用 TPL 來建立非同步工作（待會馬上就會看到範例），預設情況下，工作排程器會向執行緒集區要求一個工作執行緒 (worker thread)。.NET 執行緒集區本身則內建了一些規則來判斷如何分配執行緒，以獲得更高的執行效能；它會判斷是否應該創建一個新的執行緒，還是重複使用已經結束工作的既有執行緒。

2.5.2 建立與起始非同步工作

底下這個範例程式與本章第一個範例程式 (Ex01_ThreadStart.csproj) 幾乎一樣，差別只在於把 Thread 類別換成了 Task。

```
1 static void Main()
2 {
3     var task = new Task(MyTask);
4     task.Start();
5
6     for (int i = 0; i < 500; i++)
7     {
8         Console.Write(".");
9     }
10 }
11
```

```

12 static void MyTask()
13 {
14     for (int i = 0; i < 500; i++)
15     {
16         Console.Write("[" + Thread.CurrentThread.ManagedThreadId + "]");
17     }
18 }
```

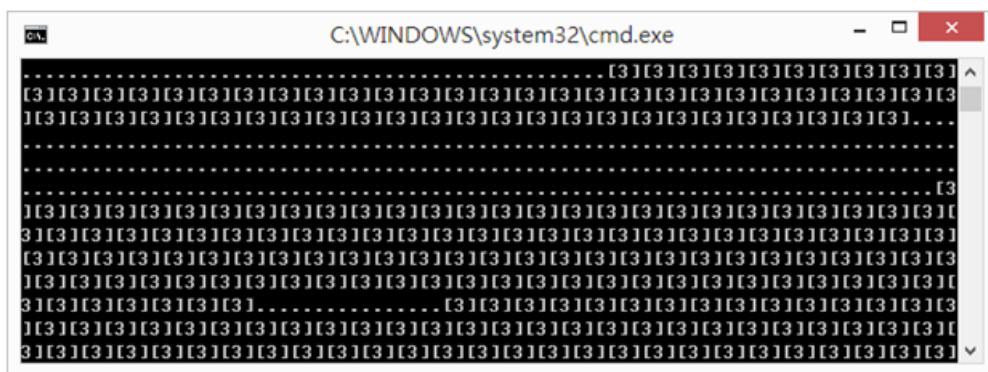


此範例程式的專案名稱：**Ex10_Task.csproj**。

程式說明：

- 使用 `System.Threading.Task` 類別來建立非同步工作，同時將一個符合 `Action` 委派的方法 `MyTask` 傳入建構函式。這個委派方法將於非同步工作開始執行時被自動呼叫。
- 呼叫 `Task` 物件的 `Start` 方法，以開始執行非同步工作。
- `Main` 函式開始一個迴圈，持續輸出「.」。這只是為了識別哪些文字是由主執行緒輸出，哪些是由工作執行緒輸出。
- `MyTask` 函式也有一個迴圈，持續輸出目前這個工作執行緒的編號。

下圖為此範例程式的執行結果：



前面提過，這裡簡短重複一次：預設的工作排程器會使用執行緒集區。換言之，當你使用 `Task` 類別來建立非同步工作時，就要意識到，這裡是會用到執行緒的（第 3 章介紹的 `async/await` 寫法則不見得會動用執行緒）。

從範例程式的輸出結果也可以看得出來，主執行緒跑了一段時間，切換至我們另外建立的工作執行緒。工作執行緒也同樣跑了一段時間之後，又切回主執行緒，如此反覆切換，直到兩個執行緒的迴圈結束為止。

順便一提，如果要等待某個非同步工作執行完畢，可呼叫 `Task` 類別的 `Wait` 方法。你可以試試上述範例程式的 `task.Start();` 之後緊接著加入一行 `task.Wait();`；看看執行結果有何不同。



你「可以」使用 `Task` 的 `Wait` 或其他方法來等待非同步工作的結果，但這不代表建議作法，也不代表你可以隨意使用。事實上，除非必要，否則應該儘量避免使用這類會令非同步工作流阻塞的 API。本書後面還會介紹非同步程式設計的一些建議寫法與陷阱。

剛才的範例是先建立一個 `Task` 物件，然後等到需要開始執行非同步工作時才呼叫該物件的 `Start` 方法來啓動工作。如果建立和起始非同步工作的操作不需要分開進行，那麼你也可以使用 `Task` 類別的靜態方法 `Run`。參考以下範例（`Ex10_TaskRun.csproj`）：

```
1 static void Main()
2 {
3     Task task = Task.Run(() =>
4     {
5         for (int i = 0; i < 500; i++)
6         {
7             Console.Write("[" + Thread.CurrentThread.ManagedThreadId + "]");
8         }
9     });
10
11    for (int i = 0; i < 500; i++)
12    {
13        Console.Write(".");
14    }
15
16    task.Wait(); // 確保非同步工作執行完畢之後才往下繼續執行。
17 }
```

這裡使用了 `Task.Run()` 來建立並起始一個非同步工作，並且使用 lambda 表達式來撰寫委派方法。另外還在應用程式結束之前呼叫 `Task` 物件的 `Wait` 方法來等待非同步工作執行完畢。



.NET 4.5 開始提供 `Task.Run()` 方法，它是 `Task.Factory.StartNew()` 的簡化形式。

目前的範例都只有一項非同步工作，當然，實務上通常有多項任務需要以非同步方式執行，而一種可能的情況是需要等待多項任務全部完成、或其中一項任務完成，然後才繼續往下執行。碰到這種情況，則可以呼叫靜態方法 `Task.WaitAll()` 或 `Task.WaitAny()`。

`Task` 類別的用法暫且簡單介紹到此，後續章節會在進一步介紹其他用法。本節內容主要在於點出，TAP 這個模式的核心概念就是由 `Task` 類別所封裝的工作，而 `Task` 及其相關操作（例如取消、等待、錯誤處理、進度回報等等）則是由 TPL 這組 API 所提供。至於 C# 5.0 的 `async` 與 `await` 語法，則是為了讓 TAP 程式碼寫起來更輕鬆，而且更容易閱讀和理解。

你的非同步程式設計觀念正確嗎？

Scott Hanselman 曾就非同步程式設計的議題訪問了 Damian Edwards 和 Levi Broderick。那次訪談的標題有點聳/動，叫做：「.NET 程式設計師所知道的非同步程式設計觀念都錯了」([Everything .NET Programmers Know About Asynchronous Programming is Wrong^a](#))。雖然是 2012 年 7 月的對話，但有些內容仍值得參考（主要是針對 ASP.NET 應用程式），故在此整理其中幾個重點。

三種適合非同步處理的場合

按照 Damian Edwards 的看法，在 ASP.NET web 應用程式中，只有三種情況才應該使用非同步處理，而且只有第一種情況最常用，另外兩種情況則很少用到。

第一種情況、也是最常見的情況，就是 I/O 處理，包括檔案存取、網路通訊、資料庫存取（通常包含檔案和網路）等等。比如說，常見的 web service 呼叫就是屬於網路 I/O。所以，可別以為在 ASP.NET web 應用程式中就不需要使用非同步呼叫喔。

第二種情況是類似儀錶板（dashboard）之類的頁面，它需要一次呈現許多區塊，而各區塊的內容又是透過其他檔案或遠端網路呼叫的方式取得。針對這種情況，由於一次要在網頁上呈現許多不同來源的資料，與其等到所有區塊的內容都取得之後才顯示整個頁面，不如以非同步的方式分頭進行，哪個區塊先取到資料，就先顯示那個區塊的內容。

第三種情況則是需要長時間處理的 HTTP 請求。比如說，收到某個 HTTP 請求時，必須啟動一個長時間的工作，並且等待那件工作執行完畢，才將工作的結果傳回用戶端（等待過程中，HTTP 連線仍是持續開著的）。

執行緒集區耗盡的問題

ASP.NET 是使用 CLR 的執行緒集區裡面的執行緒來處理用戶端發出的 HTTP 請求。如果你的 ASP.NET 應用程式會透過 `ThreadPool.QueueUserWorkItem` 方法來處理背景工作，就等於是跟 ASP.NET runtime 共用同一個執行緒集區——也就是說，你的應用程式用的執行緒數量越多，ASP.NET 能用來處理用戶端 HTTP 請求的執行緒數量就越少。不過，.NET 4.5 已針對此狀況做了改進：CLR 會偵測執行緒集區裡面的執行緒是否不夠用，並且視需要加入新的執行緒（當然，集區大小還是有上限的，這點前面已經提過）。

不要輕易調整預設參數值

早期的 ASP.NET 版本，每個 CPU 核心（core）預設能夠同時處理的最大請求數量是 12。這對許多應用程式來說，顯然是不夠的，所以當時有些人會想盡辦法調整系統預設組態來提升應用程式的效能。到了 .NET 4.0，這個預設值已經提升至每個 CPU 核心可分配到 5,000 個請求，這對大多數的 ASP.NET 應用程式來說，應該是綽綽有餘，也就不太需要去調整預設參數了。

^a<http://goo.gl/r4YWWn>

2.6 重點回顧

- 一個 CLR 有一個執行緒集區。故一般而言，一個 .NET 應用程式有一個自己專屬的執行緒集區（除非它會載入多個 CLR）。
- CLR 實作的執行緒集區分成兩種：工作執行緒集區和 I/O 執行緒集區。前者用來處理 CPU 運算類型的工作，後者專用於 I/O 操作。
- .NET 執行緒可分為兩種：前景執行緒和背景執行緒。兩者的主要區別是：當所有的前景執行緒停止時，應用程式就會結束，並且停止所有背景執行緒。若只是停止背景執行緒，則不會造成應用程式結束。此外，雖然結束應用程式時，.NET 會通知所有的背景執行緒停止，但比較保險的做法還是自行結束背景執行緒。
- 本章提及的幾種非同步程式設計方法，它們在 .NET 版本演進過程中出現的順序如下：
 - .NET 1.0: 專屬執行緒 (`Thread` 類別)、執行緒集區 (`ThreadPool` 類別)。
 - .NET 1.1: APM (Asynchronous Programming Model)。
 - .NET 2.0: EAP (Event-based Asynchronous Pattern)。
 - .NET 3.5: 改進執行緒集區的效能。
 - .NET 4.0: TPL (Task Parallel Library)。
 - .NET 4.5: TAP (Task-based Asynchronous Pattern)，C# 5 的 `async` 與 `await` 寫法。
- 使用 TPL 的 `Task` 類別來建立非同步工作時，往往代表背後會使用執行緒。因為 TPL 的預設工作排程器會透過執行緒集區來獲取工作執行緒。
- 並非所有的非同步寫法都必然會動用執行緒。

The End
