

C# 開發實戰

非同步程式開發技巧



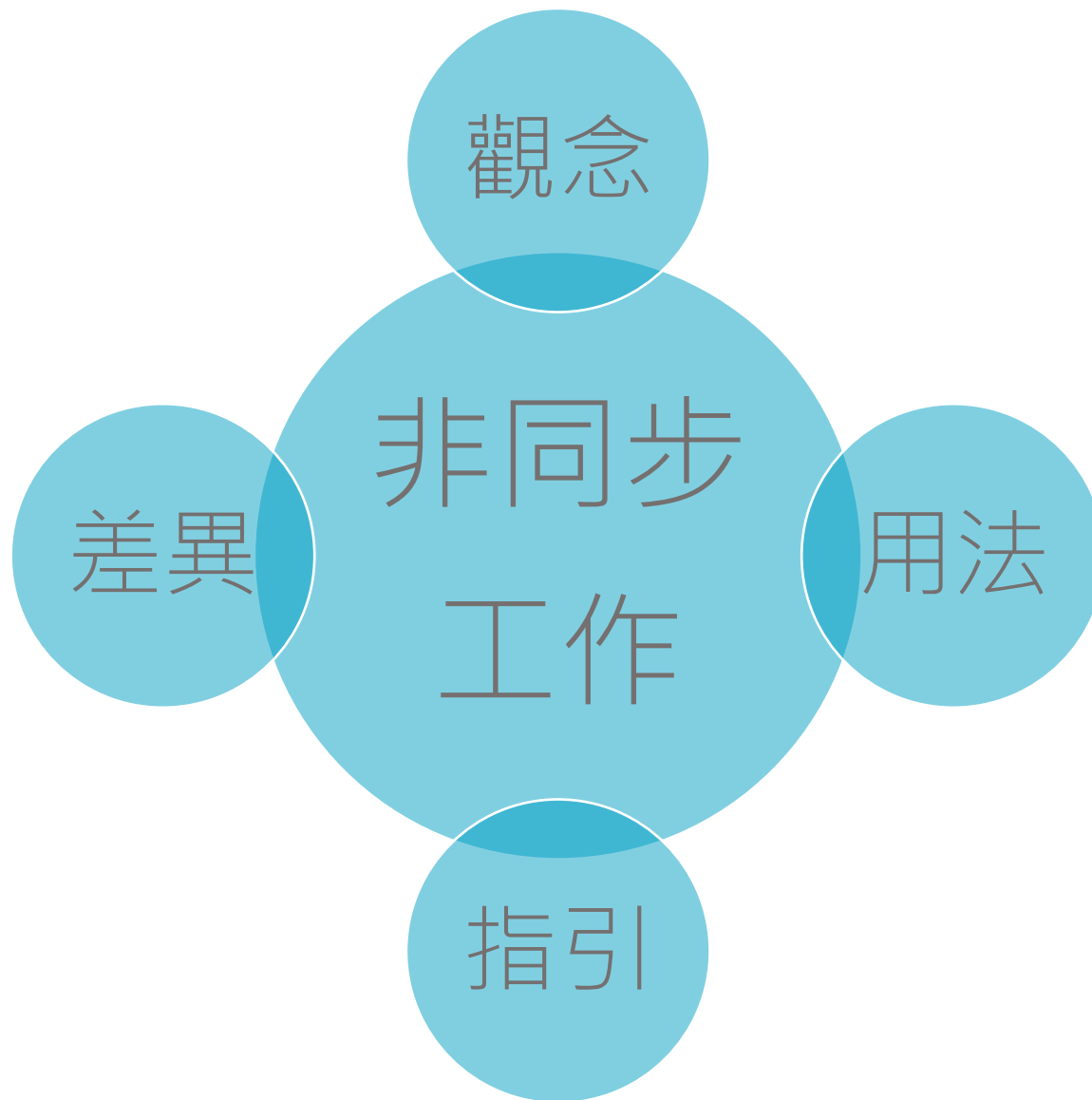
多奇數位創意有限公司

技術總監 黃保翕 (Will 保哥)

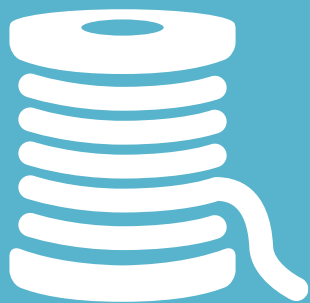
部落格：<http://blog.miniasp.com/>



你將會學到...



基本核心知識回顧



Thread
執行緒



Callback
回呼

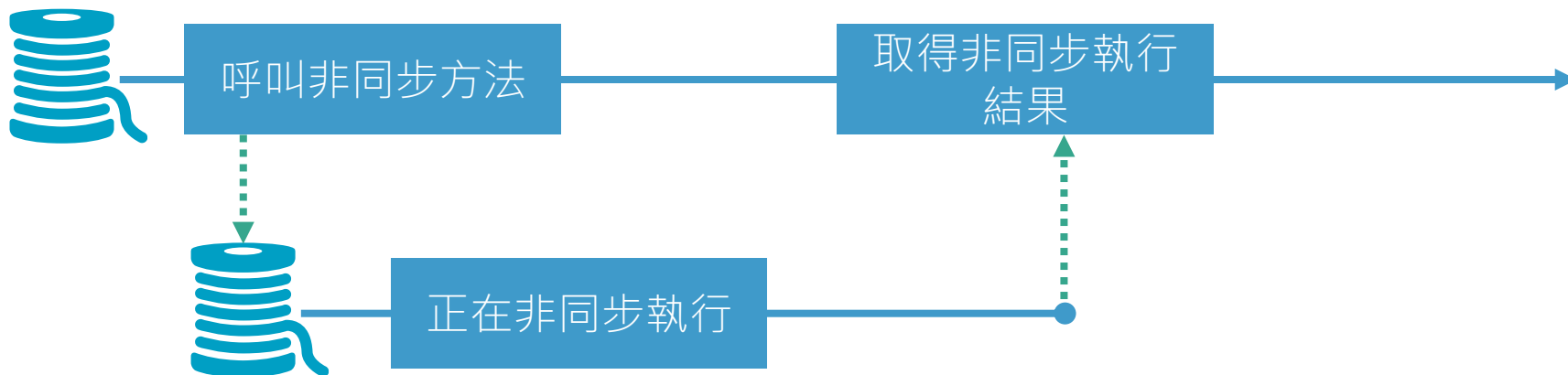


Task
工作



什麼是非同步程式設計

- 一種 並行 Concurrent 處理
 - 當呼叫**非同步方法**之後，將會**立即返回**呼叫端
 - 並**承諾**未來將一定會**完成這個方法**並且**得到結果**。
- 不一定需要透過**執行緒**才能夠做到**非同步作業**
 - 要看非同步作業類型屬於 **CPU Bound** 或者 **I/O Bound**



並行計算與平行計算

- Concurrent computing 並行計算

- a form of computing in which several computations are executed during **overlapping time periods** - **concurrently** - instead of sequentially
- 數個運算工作**同時**在特定時間區段內交替地完成，而非依序執行

只有一個
處理器



- Parallel computing 平行計算

- a type of computation in which many calculations or the execution of processes are carried out **simultaneously**
- 許多運算過程會**同時發生**

處理器 1



處理器 2



整個餐廳只有一個人在服務

- 餐廳



老闆無法服務更多客戶(因為他需要負責許多工作)並會造成客戶滿意度降低(點餐後餐點很久才會到、結帳速度慢)



餐廳只有老闆一人

老闆要負責招呼客人
老闆要接受點餐與送餐
老闆要烤吐司、煎蛋、煮咖啡
老闆要負責收銀櫃臺

烤吐司要手動，為避免吐司烤焦，所以，在烤吐司過程中，老闆要在土司機旁



整個餐廳有多人在服務

- 餐廳



餐廳增加收銀員、廚師、服務員

每個人專門負責自己的工作
更換可以定時烤麵包機

因此
可以服務更多客戶
提升客戶滿意度



一個典型的非同步作業範例 (說說您的看法)

正常運作分解

正常運作情況

失敗運作情況



一個典型的非同步作業範例 (說說您的看法)

- 若只有一個人來進行賽車維護，會有甚麼問題產生呢？
 - 設計**同步程式碼**比起**非同步程式碼**更加**簡單與容易**開發
- 因為**多人**同時一起完成一個需求，所以可以在很**短時間**內完成
 - 應用程式整體執行**效率**提升了
- **如何才能夠知道**整個過程都完成，因為賽車手要能夠繼續比賽
 - **等候**非同步的工作完成
- 任何一個人的**失敗**，將會造成輸了這場賽事，或者造成災害發生
 - 設計不當將會造成應用程式**崩潰**，並且很難進行除錯
- 每個維護人員負責**一件**維修事務
 - 使用執行緒來**單獨執行**這件事情
- 加入**更多**的維護人員是否可以更快的完成維修工作
 - 聘請更多人是**有成本**的，還必須確保彼此之間不會造成**同步干擾**



為什麼要學習 非同步程式設計？

提高整體執行效能和回應速度
Performance / Responsiveness

參考

GUI 應用程式 的同步需求處理

ASP.NET 應付請求與進行外部資源同步存取

非同步程式設計特色



若任何方法需要超過 **50ms** 以上時間才能夠完成，建議改成非同步方法



非同步程式設計就是 **I'll Call You Back**

- 同時進行處理多個需求，需要**等候**方法處理完成
- 非同步程式設計是一種 **並行 Concurrent** 程式設計，這些程式碼將會在應用程式也許以不同**執行緒**來執行，並且會通知呼叫執行緒的**完成、失敗、進度**狀態
- **可以改善應用程式的執行效能與回應速度**
- 有任何 **I/O Bound** 存取需求
 - 例如，從網路要求資料或存取資料庫
- 有任何 **CPU Bound** 計算需求
 - 例如執行耗費資源的計算



不良的設計與使用非同步程式設計方法將會帶給程式設計師**嚴重的噩夢**

如何精通非同步程式設計？

熟悉非同步的基礎知識

有助於寫出好的非同步程式碼

你是否也會經常遇到這些問題，不知如何因應呢？

- 為什麼**無法捕捉**到非同步方法的例外異常，而**閃退了**
- **TPL** (Task Parallel Library) 與 **TAP** (Task Asynchronous Pattern) 哪裡不同
- 為什麼有時候程式會**卡住不動了**(因為打死結了) (WPF)
- 因為呼叫了非同步方法導致無法存取資源 (**HttpContext**)
- **什麼時候**要來使用非同步方法？
- 想要理解 **async/await** 關鍵字的運作方式
- 非同步方法真的一定需要**使用執行緒**嗎？
- 如何**取消**一個或者多個非同步作業 (**時間逾期**並自動取消)
- 經常聽到在使用**非同步**方法的時候，**不要使用同步**方式設計，這是什麼意思呢？



相關重要名詞 (1)

- 多 CPU (Multi-CPU)
- 多核心 (Multi-Core)
- 超執行緒 (Hyper-Thread)
- 程式 (Program)
- 處理程序 (Process)
- 執行 (Execute)
- 執行緒 (Thread)
- 多執行緒 (Multi-Thread)
- 多工 (Multi-Tasking)
- 排程器 (Scheduler)
- 平行計算 (Parallel computing)
- 並行計算 (Concurrent computing)
- 同步 (Synchronous)
- 非同步 (Asynchronous)
- 內容交換 (Context Switching)
 - Windows 採用先佔式多工 (Preemptive Multitasking)，一個執行緒執行程式時間用完了，系統就會進行 context switch，把 CPU 分配給下一個執行緒，沒有一個程式能獨佔 CPU 時間

相關重要名詞 (2)

- 同步內容 (SynchronizationContext)
 - 允許執行緒透過將 工作單元(Work of Unit) 進行封裝 (Marshal) 之後，傳遞給其他執行緒
- 同步化 (Synchronization)
 - 當多執行緒需要同時存取共用資源的時候，所需要進行的管控機制，確保程式執行時可以得到預期結果！
 - 例如：lock, Mutex, SpinLock, Semaphore, ReaderWriterLock, ...
- 執行緒安全 (Thread-safe)
 - 在多執行緒環境下執行也可以確保得到預期的執行結果
 - 必要條件就是必須能夠提供 同步化 (Synchronization) 機制



相關重要名詞 (3)

- 關鍵區段 (Critical Section)
 - 是任意一段程式碼，**不允許多執行緒同時執行這段程式碼**
- 死結 (Deadlock)
 - 當兩個執行緒**都嘗試鎖定**另一個執行緒**已經鎖定的**資源時，就會發生死結。
 - 兩個執行緒都**不能繼續執行**。
- 競爭狀態 (Race Condition)
 - 競爭情形是一種錯誤，這種錯誤是指根據兩個或多個執行緒之中，哪一個先到達程式碼的特定區塊而決定程式的結果。
 - 執行程式**多次會產生不同的結果**，並且**無法預測**任何指定的執行結果。



相關重要名詞 (4)

- **APM** (Asynchronous Programming Model)
非同步程式設計模型
- **EAP** (Event-based Asynchronous Pattern)
事件架構非同步模式
- **TPL** (Task Parallel Library)
工作平行程式庫
- **TAP** (Task-based Asynchronous Pattern)
以工作為基礎的非同步模式



非同步程式運作示意圖

需要注意哪些事項呢？



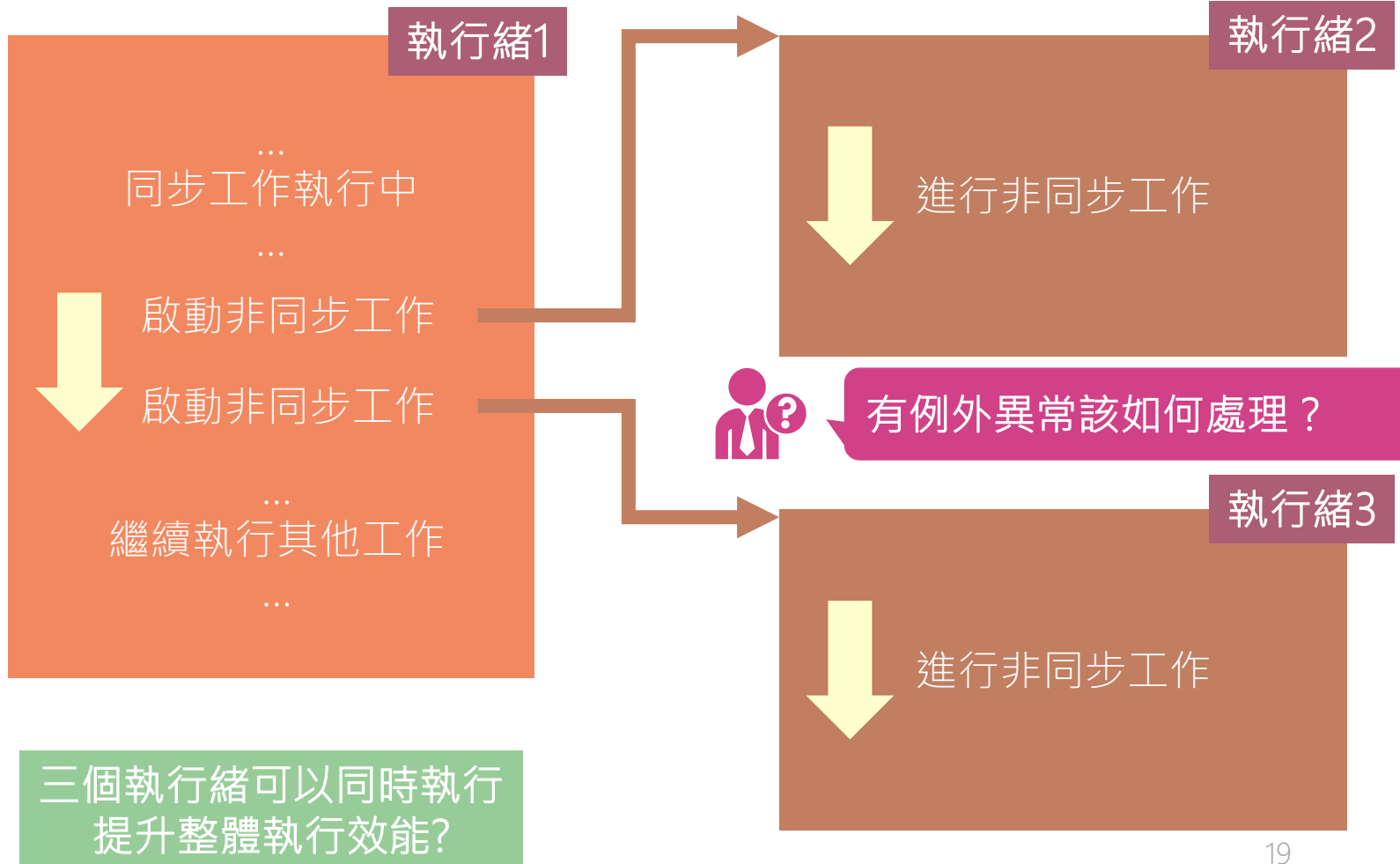
如何得知非同步工作何時完成？



如何取得非同步執行結果？



取得結果時，可以避免執行緒1等候嗎？



三個執行緒可以同時執行
提升整體執行效能？

多執行緒非同步作業與等候執行緒執行完成範例

- 建立兩個新執行緒
- 分別非同步執行
2秒鐘 與 3秒鐘
- 使用 **Join** 方法等候
執行緒執行完成
- 請說明執行結果？

```
Thread thread1 = new Thread(() =>
{
    Thread.Sleep(900);
    for (int i = 0; i < 20; i++)
    { Thread.Sleep(100); Console.Write("X"); }
});
Thread thread2 = new Thread(() =>
{
    Thread.Sleep(900);
    for (int i = 0; i < 20; i++)
    { Thread.Sleep(150); Console.Write("-"); }
});

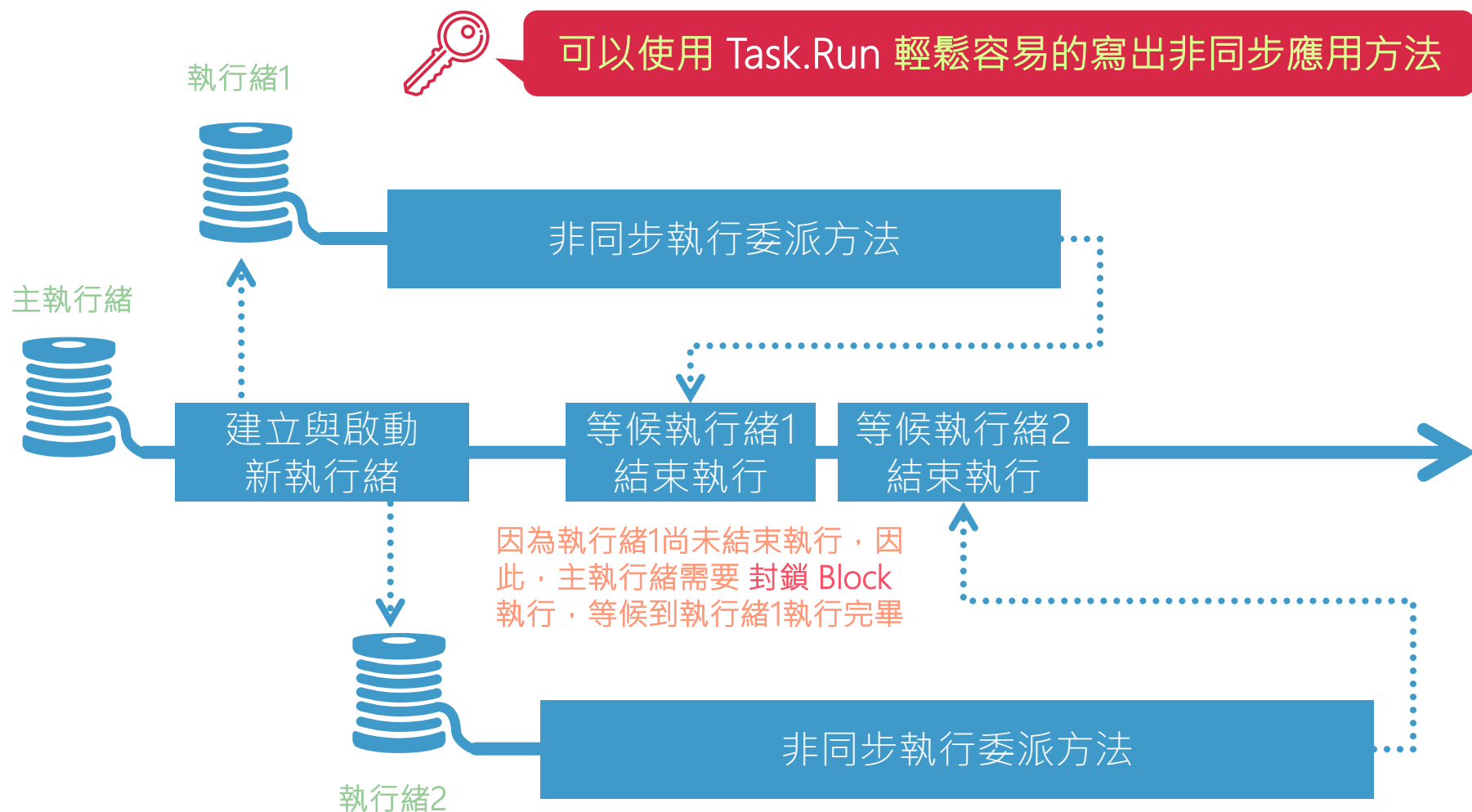
thread1.Start();thread2.Start();
thread1.Join();
thread2.Join();

Console.WriteLine("Press any key for continuing...");
Console.ReadKey();
```



使用 Thread.Join() 方法，將會造成主執行緒進入 封鎖(Block) 狀態

多執行緒非同步作業示意流程圖



主執行緒、執行緒1、執行緒2 將會同時並行在作業系統執行

多執行緒非同步作業與取得執行結果範例

- 取得執行緒執行結果內容做法
 - 使用共用變數來儲存執行結果
 - 使用 回呼 callback 委派方法取得執行結果
- 請說明執行結果？

```
static void Main(string[] args)
{
    int shareData = 0;
    Console.WriteLine($"建立新執行緒物件");
    Thread thread1 = new Thread(() =>
    {
        shareData = 0;
        Thread.Sleep(900);
        for (int i = 0; i < 20; i++)
        {
            Thread.Sleep(100); Console.Write("X");
            shareData += i;
        }
        Console.WriteLine();
    });
    thread1.Start();

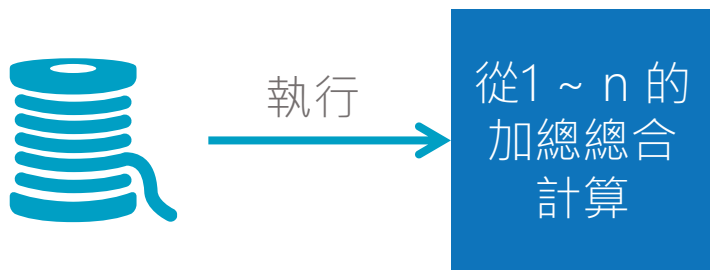
    thread1.Join();
    Console.WriteLine($"執行緒1 執行結果為 {shareData}");

    Console.WriteLine("Press any key for continuing...");
    Console.ReadKey();
}
```

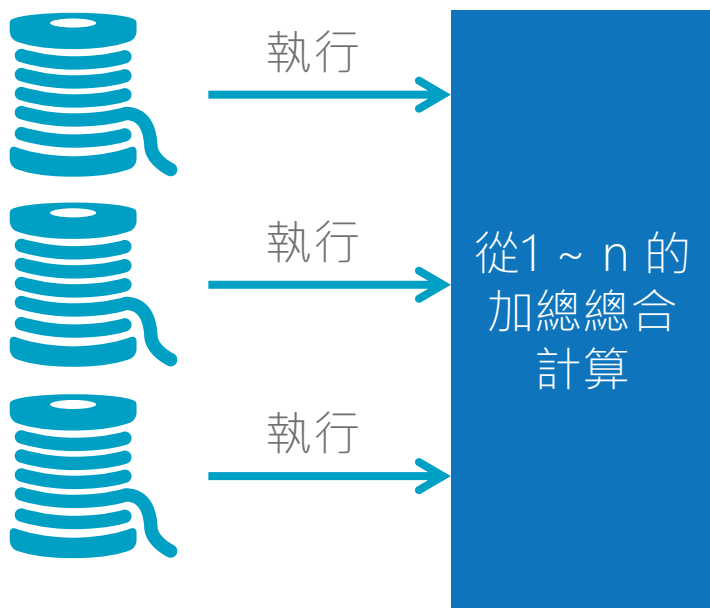


需要使用 Thread.Join() 方法，確認執行緒已經執行完畢

為什麼要有執行緒安全 (Thread Safe) ?



在同一個執行緒下執行，若執行多次，所得到的每次結果是否都會相同



在多個執行緒下同時執行，所得到的結果是否為 單一執行緒執行結果 \times N (執行緒數量)

執行緒安全 (Thread Safe)

- 若同一個方法程式碼，每次執行結果都是**可預期**的內容，則代表該方法為**執行緒安全**
- 當一個以上的執行緒同時競相存取共用資源時，就可能發生**競爭**情形。
- 當執行緒要操作共用資料的時候，必須要能夠保證同一個時間內，僅有一個執行緒可以操作共用資料。
- 可使用鎖定來保護程式碼的**關鍵區段 (Critical Section)** 不受競爭情形影響。
- .NET Framework 基礎類別庫 (BCL) 中，大多數類別都不會特別處理**執行緒安全**！



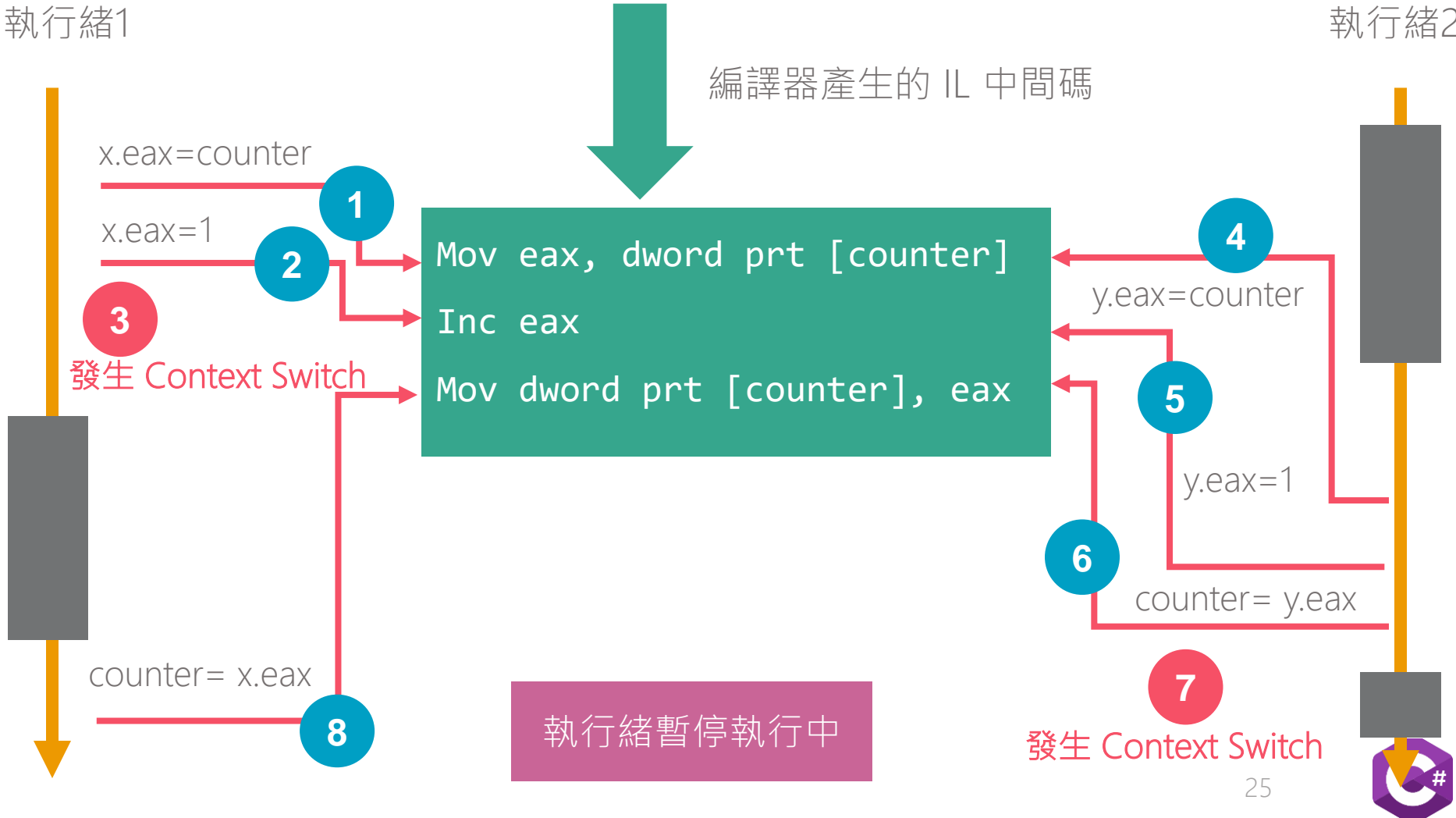
執行緒的競爭 時間序列說明

- C# 敘述 `counter = counter + 1;`

執行緒1

執行緒2

編譯器產生的 IL 中間碼



使用多執行緒同時計算總共跑多少次迴圈

- 累計統計數量將會儲存在**共用變數 counter** 內
- **單一**執行緒執行完的結果為 10,000,000
- 若同時啟動**兩個**執行緒，請說明執行結果？
 - 是 20,000,000 嗎？
- 如何解決**執行緒不安全**的問題呢？

```
class Program
{
    private static int counter = 0;
    public static void Main()
    {
        Thread thread1 = new Thread(非同步方法);
        Thread thread2 = new Thread(非同步方法);
        thread1.Start();
        thread2.Start();

        thread1.Join();
        thread2.Join();

        Console.WriteLine("兩個執行緒聯合計算結果是:");
        Console.WriteLine(counter);
    }

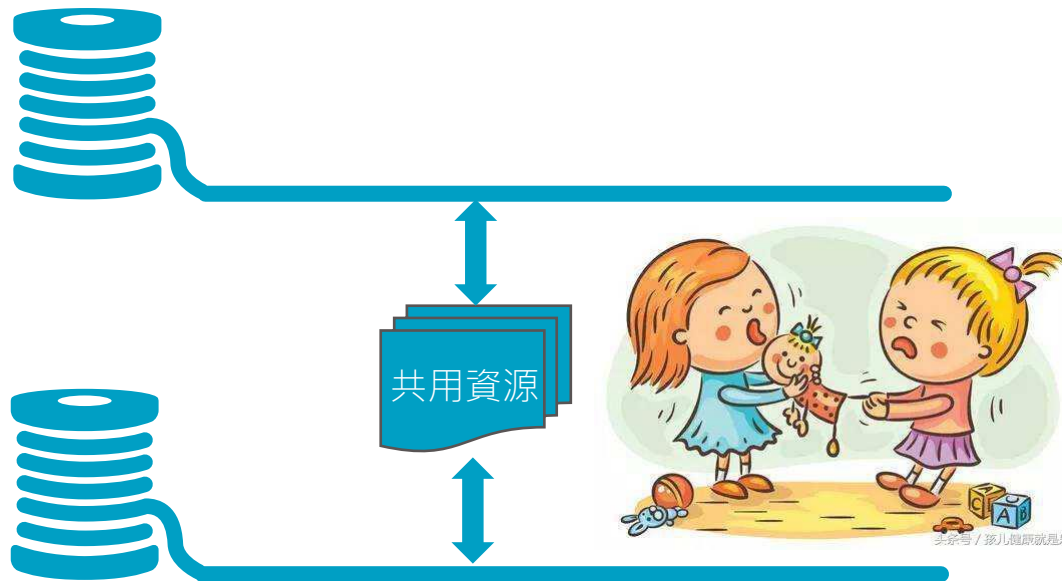
    private static void 非同步方法()
    {
        for (int index = 0; index < 10000000; index++)
        {
            counter++;
        }
    }
}
```



需要使用 Thread.Join() 方法，確認執行緒已經執行完畢

同步處理 (Synchronization)

- 確保多執行緒執行結果是**可以預測**的，不會每次執行**得到不同結果**
- .NET Framework 提供同步處理原始物件的範圍，以便控制執行緒的互動，及避免競爭情形。這些可以大致分為三大類：
 - 獨佔鎖定 (Exclusive locking)
 - 非獨佔鎖定 (Nonexclusive locking)
 - 信號 (Signaling)
- 請務必記得，在多執行緒需要存取共用資源時，若有一個執行緒**略過同步處理機制**並直接存取受保護的資源，則原本的**同步處理機制**便會失效！



非同步程式設計的注意要點

- 選擇適合的非同步運算方法 (I/O Bound, CPU Bound)
- 啟動與結束非同步方法
- 等候非同步運算完成
- 取消非同步運算
- 當有例外異常產生的時候，避免造成整個系統崩潰
- 如何得知非同步運算的處理進度
- 如何進行非同步工作的問題除錯

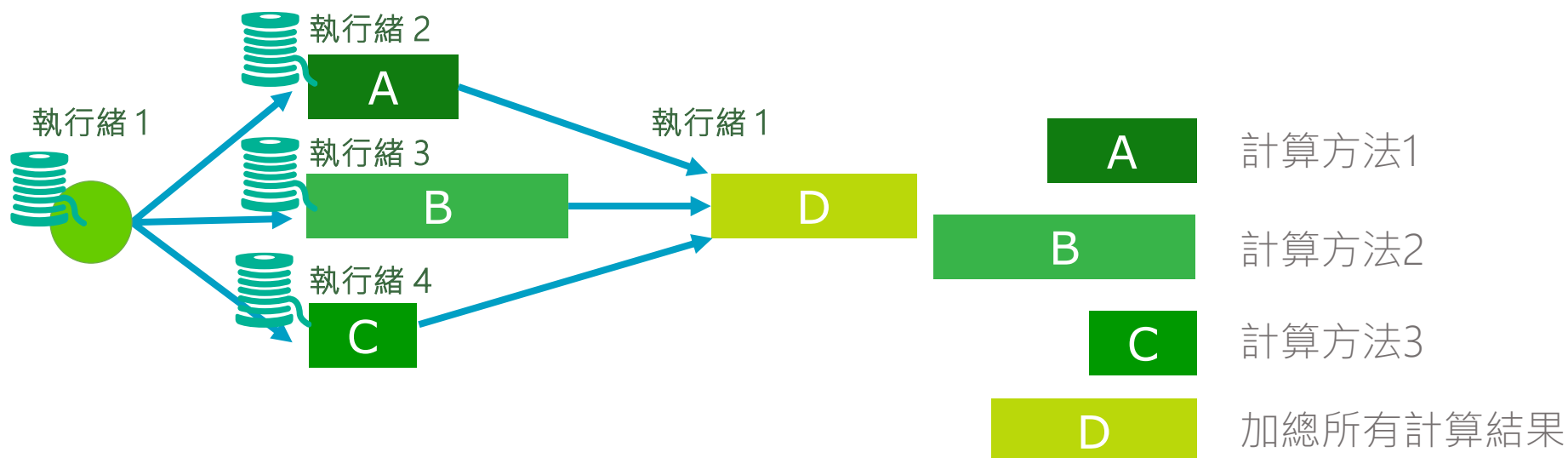


同步與非同步 (進行 CPU 相關的處理工作)

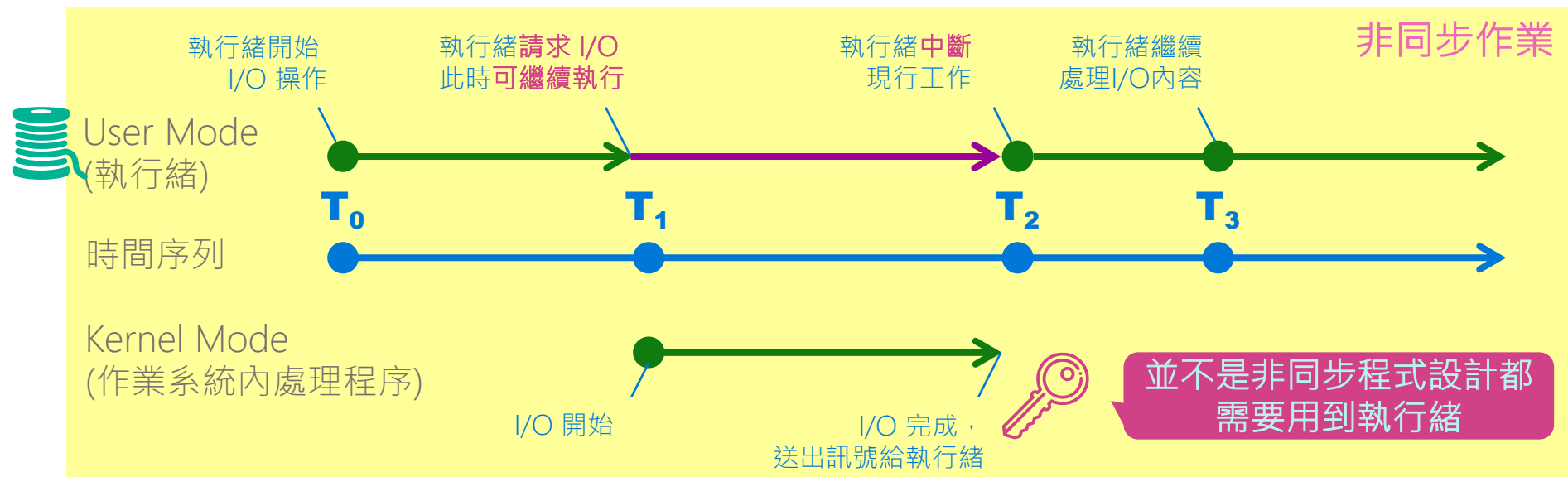
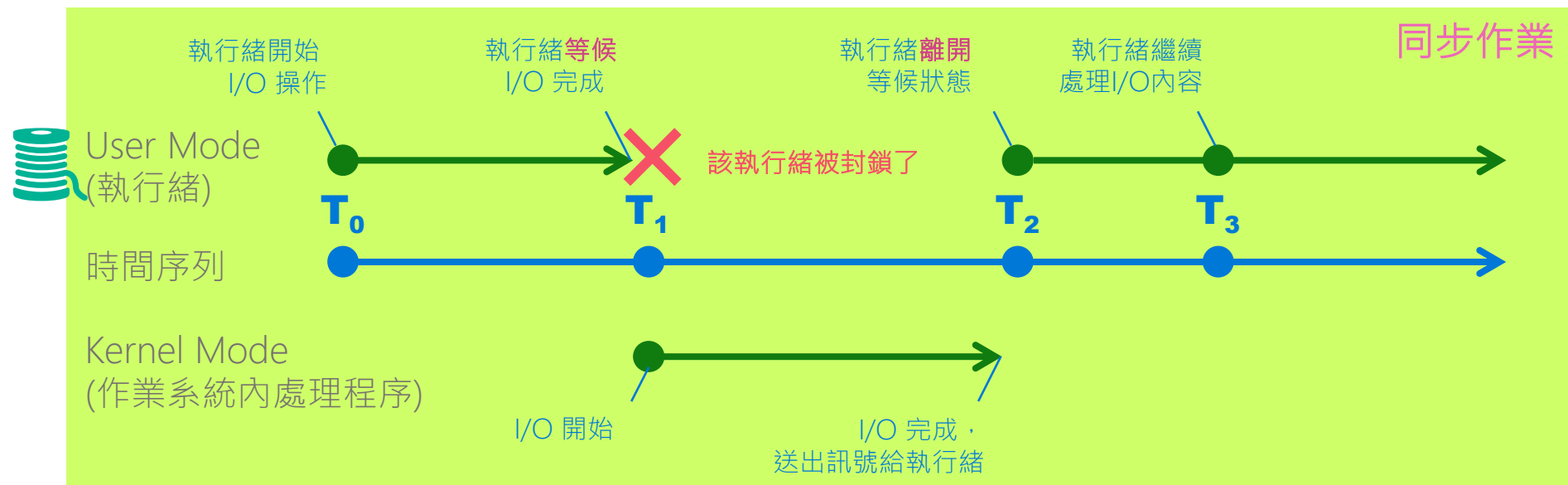
同步作業 (程式碼 依序 執行，若有程式碼需較多執行時間，只有等待完成)



非同步作業 (程式碼拆解不同區塊，使用 並行 執行，因此可以提升整體執行效能)



同步與非同步 (進行 I/O 相關的處理工作)



進行非同步 WriteAsync 作業的時候 可以不需要使用到執行緒作法說明

- 進行呼叫非同步 WriteAsync 方法
- OS 將會要求裝置驅動程式準備開始寫入資料
 - 寫入請求物件將會建立一個 IRP (I/O Request Packet)
 - 裝置驅動程式將會收到 IRP 要求，並且返回到 OS，接著返回到 WriteAsync 方法，此時，這是一個未完成的工作
- 當裝置驅動程式完成寫入
 - 將會產生一個中斷 (Interrupt) 通知給 OS
 - 此時，會將剛剛未完成的工作標記為完成狀態
 - 在 I/O 執行緒集區內執行緒將會通知這個非同步作業完成



常見的三種使用非同步的使用情境

- I/O

- 檔案 I/O 存取
- 網路、硬體資源
- Web 服務呼叫、資料庫存取、檔案讀寫

- Parallelism

- 平行處理多個同步或非同步 API (將問題切割成子問題)
- 可以提升系統反應時間
- 例如：可以平行取得儀表板上的各個元件項目

- Long-running event-driven

- 請求將會在休眠狀態，直到有相關事件發生
- 例如：WebSocket / SignalR / Stream

這些使用情境的共通特色：
都會超過 **50ms** 執行時間！

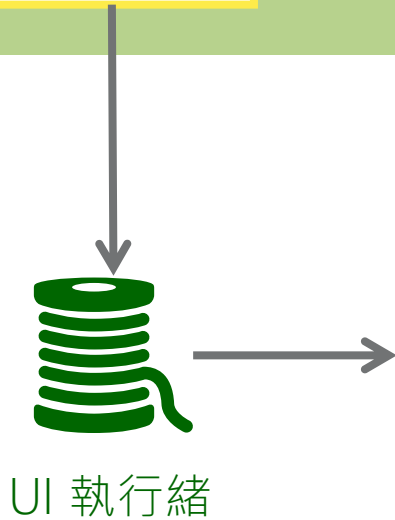
Windows Forms / WPF / Xamarin /
ASP.NET 開發框架下

究竟有什麼理由，需要使用到
非同步程式設計技術？

GUI 應用程式 的同步需求處理



1. 使用者點選 開始 按鈕
2. 因為 UI 執行緒上，同步進行長時間的工作
所以，整個應用程式的 UI 將會造成凍結現象
3. 當同步長時間工作處理完成後，應用程式便可繼續使用



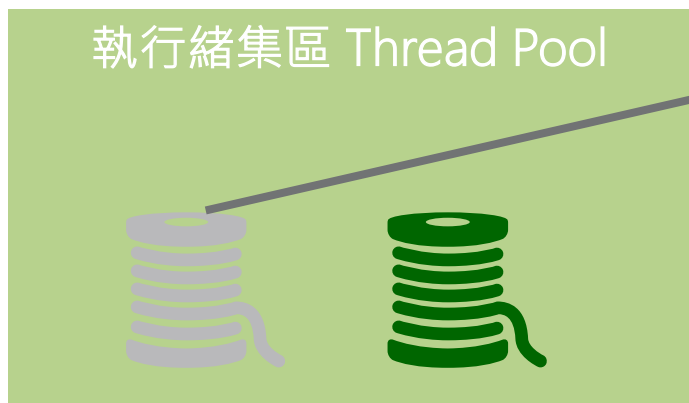
大量計算或長時間資源存取



UI 執行緒負責整個應用程式的使用者介面更新與顯示行為

ASP.NET 應付請求與進行外部資源同步存取

有外部請求 Request 進來，將會配置執行緒來執行



當有請求產生的時候



Http 請求 Request

1. ASP.NET 收到一個請求
2. 從執行緒集區取得一個執行緒
3. 該請求需要同步執行一個外部資源存取 (資料庫、Web API、檔案存取)
4. 現在執行緒將會被封鎖 (Block)
5. 外部資源存取完成後，該執行緒會繼續執行
6. 當該請求執行完畢後，將會把執行緒歸還到執行緒集區



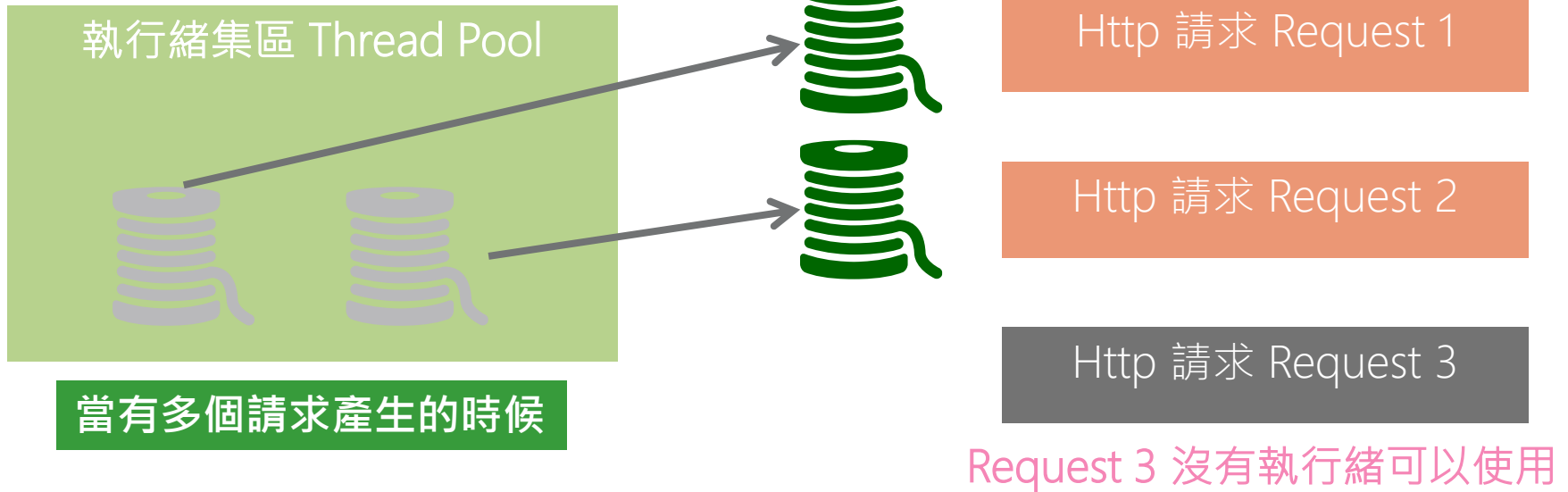
當請求執行完成的時候



HttpContext, HttpRequest 等等，只會確保在請求 Request 執行緒內可以安全存取這些物件

ASP.NET 應付多個請求與進行外部資源同步存取

假設該站台最多只有兩個執行緒可以提供服務



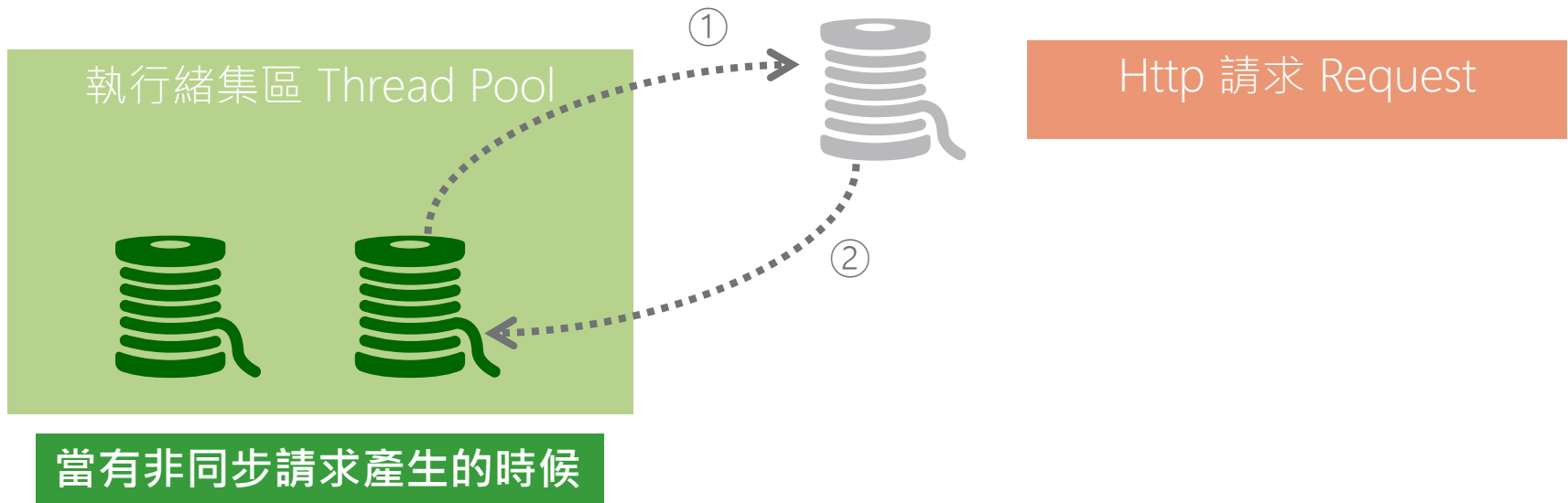
1. ASP.NET 收到一個請求
2. 從執行緒集區取得一個執行緒
3. 該請求需要同步執行一個外部資源存取
(資料庫、Web API、檔案存取)
4. 現在執行緒將會被封鎖 (Block)
5. 此時第二個請求也進來了，因此，如同步驟 1~4，該執行緒也被封鎖了 (Block)
6. ASP.NET 收到新的請求後，因為執行緒集區沒有足夠可用執行緒，將會等候，直到逾時
此時，將會回應 HTTP Error 503 (Service unavailable)
7. 可是，已經從執行緒集區取出的執行緒，也都沒有在做甚麼事情，
只是在等待外部資源存取工作完成，因為他們被封鎖了 (Block)



執行緒被 Block (封鎖) 這樣的行為，是浪費系統資源的動作，不建議使用

ASP.NET 應付請求與進行外部資源非同步存取

當呼叫非同步方法時會立即歸還執行緒給執行緒集區



1. ASP.NET 收到一個請求
2. 從執行緒集區取得一個執行緒
3. 該請求需要**非同步**執行一個外部資源存取 (資料庫、Web API、檔案存取)
4. 執行緒將會**歸還**給執行緒集區
5. 外部資源存取完成後，會從執行緒集區取得一個執行緒，透過該執行緒繼續執行該請求的其他程式碼
6. 當該請求執行完畢後，將會把執行緒歸還到執行緒集區

非同步程式設計模式

Asynchronous Programming Patterns



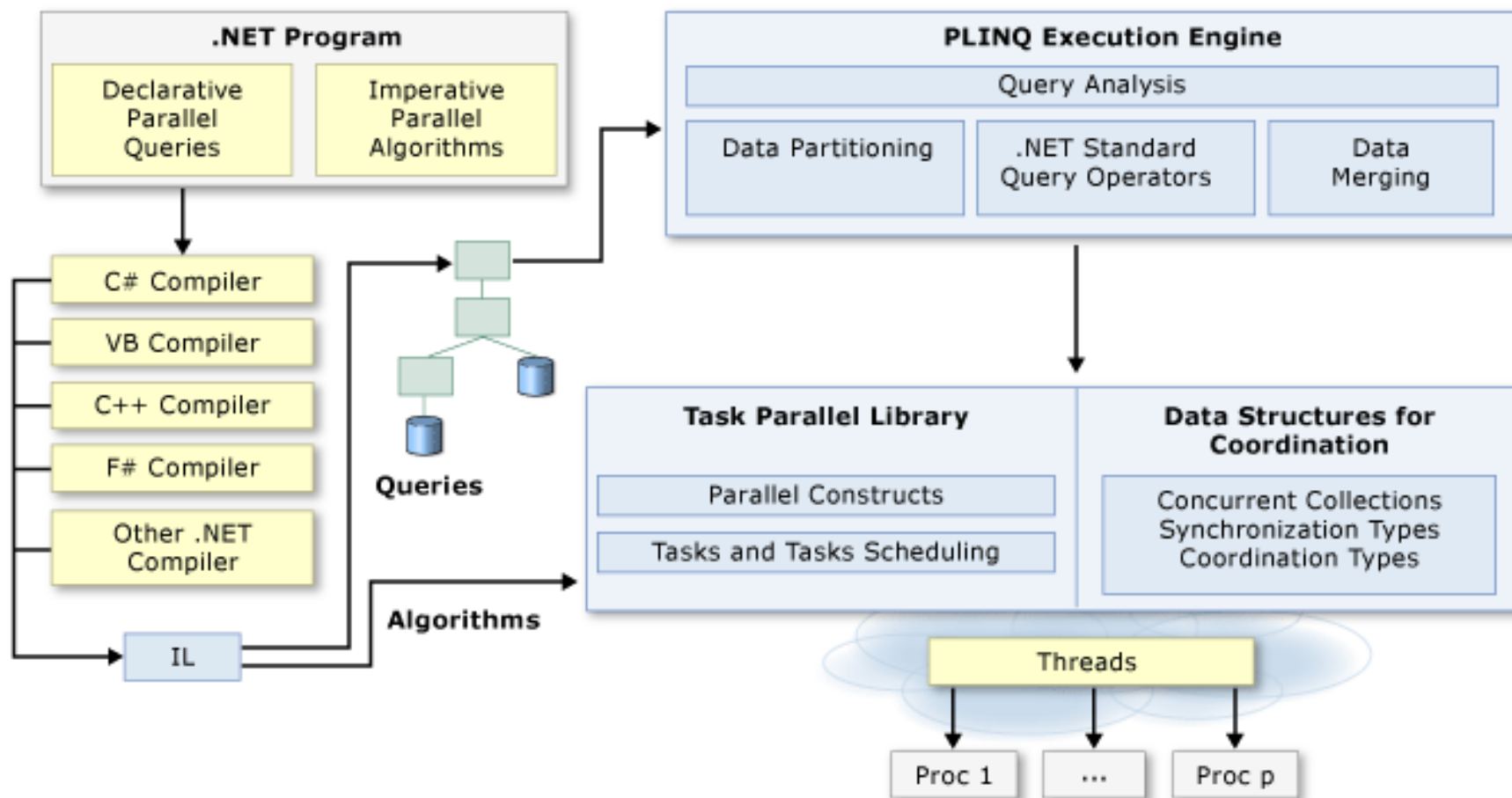
APM 非同步程式設計模型

EAP 事件架構非同步模式

TAP 以工作為基礎的非同步模式



.NET Framework 4 中平行程式設計架構



.NET 框架的非同步開發的歷史



非同步開發的歷史

執行緒 Thread 與 BackgroundWorker 和 Timer

- .NET Framework 1.1
 - **APM 非同步程式設計模型** (Asynchronous Programming Model)
- .NET Framework 2.0 / 3.5
 - **EAP 事件架構非同步模式** (Event-based Asynchronous Pattern)
 - **在不同執行緒上執行作業** (BackgroundWorker)
- .NET Framework 4.0
 - **TAP 以工作為基礎的非同步模式** (Task-based Asynchronous Pattern)
- .NET Framework 4.5 (C# 5.0+) / .NET Core 1.0+
 - 可透過 **async / await** 語法糖使用 **TAP 非同步模式**



同步與非同步程式碼範例

同步程式碼範例

```
public class MyClass
{
    public int Read(byte [] buffer, int offset, int count);
}
```

APM 非同步程式碼範例

```
public class MyClass
{
    public IAsyncResult BeginRead(byte [] buffer, int offset, int count,
                                     AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

EAP 非同步程式碼範例

```
public class MyClass {
    public void ReadAsync(byte [] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}
```

TAP 非同步程式碼範例

```
public class MyClass
{
    public Task<int> ReadAsync(byte [] buffer, int offset, int count);
}
```

非同步程式設計模式

- 相關技術背景介紹
- .NET 框架的非同步開發的歷史
- APM 非同步程式設計模型

Asynchronous Programming Model

- EAP 事件架構非同步模式

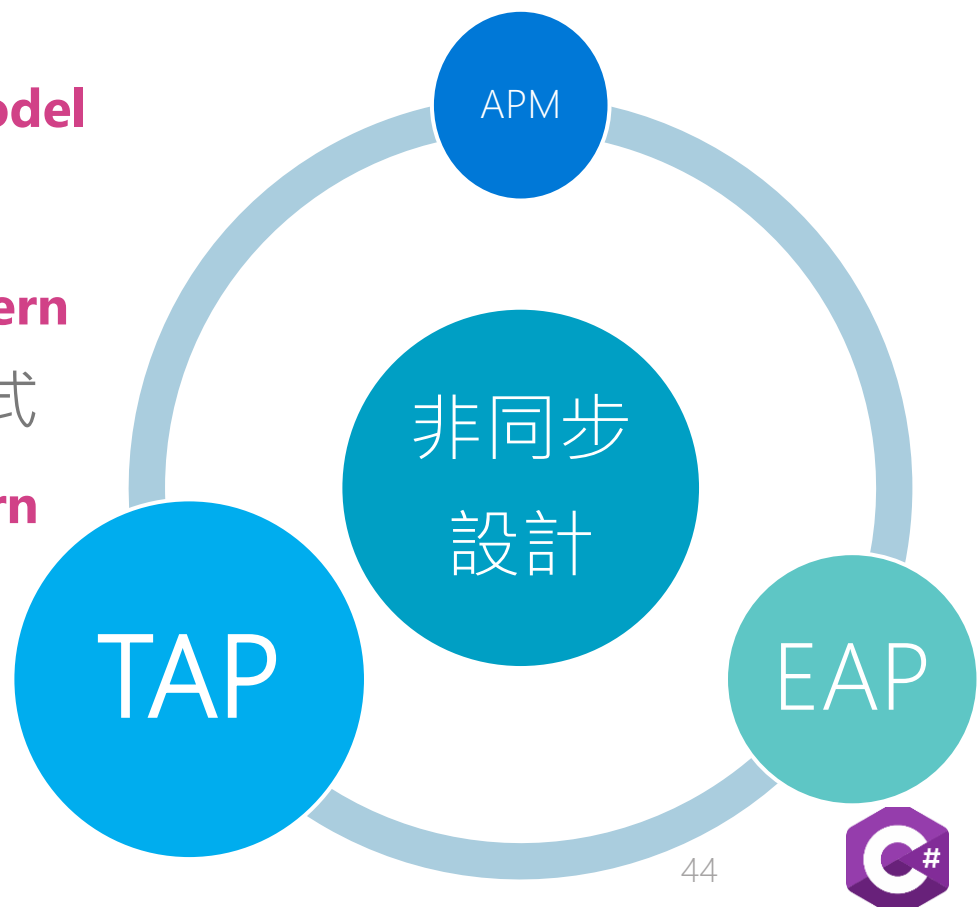
Event-based Asynchronous Pattern

- TAP 工作為基礎的非同步模式

Task-based Asynchronous Pattern



我要選擇哪一種非同步設計方法呢？



APM 非同步程式設計模型

Asynchronous Programming Model



.NET 1.1



非同步程式設計模型 (APM)



通稱為 Begin / End 模式

- 使用 **IAsyncResult** 設計模式的非同步作業會被實作成兩個方法，分別負責開始和結束非同步作業
OperationName
 - Begin**OperationName
 - 開始進行非同步執行作業，不影響現有執行緒執行
 - End**OperationName
 - 當非同步作業執行完畢，就可以呼叫此方法取得執行結果



當非同步作業尚未完成且呼叫 **EndXXX** 方法，會造成該執行緒被封鎖 Block



何時與如何呼叫 **EndXXX**，來取得執行結果？



執程式碼

BeginXXX

啟動非同步作業

EndXXX

取得非同步執行結果

執程式碼



APM 使用範例

同步方法

若尚未讀取資料完成，該Thread無法繼續執行下去

```
public class MyClass
{
    public int Read(byte[] buffer, int offset, int count);
}
```

APM方法

讀取資料完成後，會使用 AsyncCallback通知或者其他方法

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte[] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

需要使用 EndRead 取得所讀取到的資料



對於非同步方法的程式設計

如何得知非同步方法執行完畢

以便可以取得非同步方法執行結果

通常有 **4種** 作法

使用 APM 呼叫非同步取得結果四種方法

這是大部分非同步程式碼設計會用到的技術

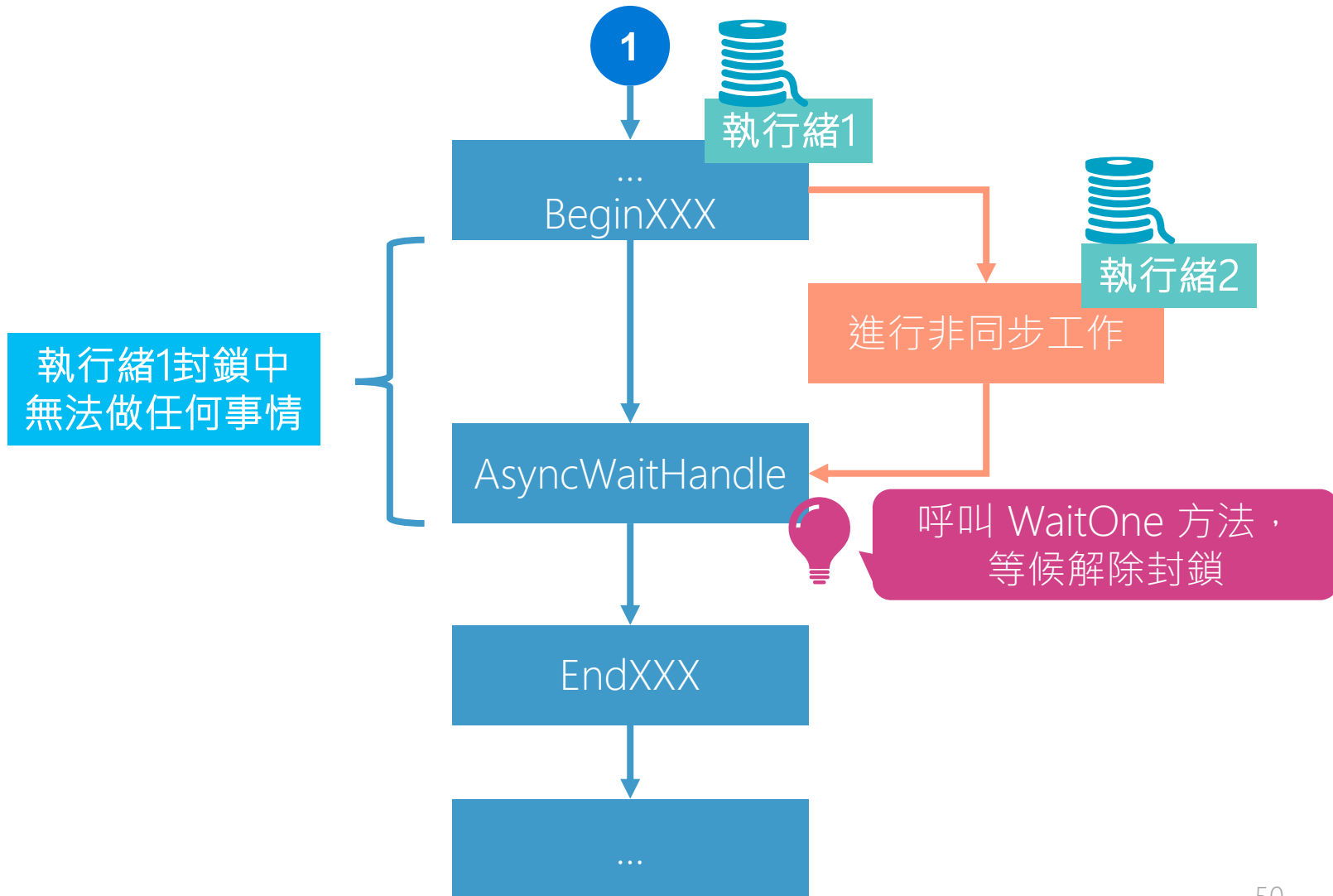
- ① 使用 AsyncWaitHandle 封鎖應用程式執行
 - 在非同步計算**結束前**，執行緒**停止執行**
- ② 結束非同步作業的方式封鎖應用程式執行
 - 在等候非同步**最後結果前**，執行緒**停止執行**
- ③ 輪詢非同步作業的狀態
 - 一直**持續詢問**非同步計算是否**已經完成**
- ④ 使用 AsyncCallback 委派結束非同步作業
 - 當非同步工作**完成後**，透過新**執行緒**執行**回呼 callback 委派**



哪種方式比較好呢？

使用 AsyncWaitHandle 圖例

封鎖式等待非同步作業結束



使用 APM 呼叫 BeginXXX 啟動非同步工作，但是採用同步程式碼設計

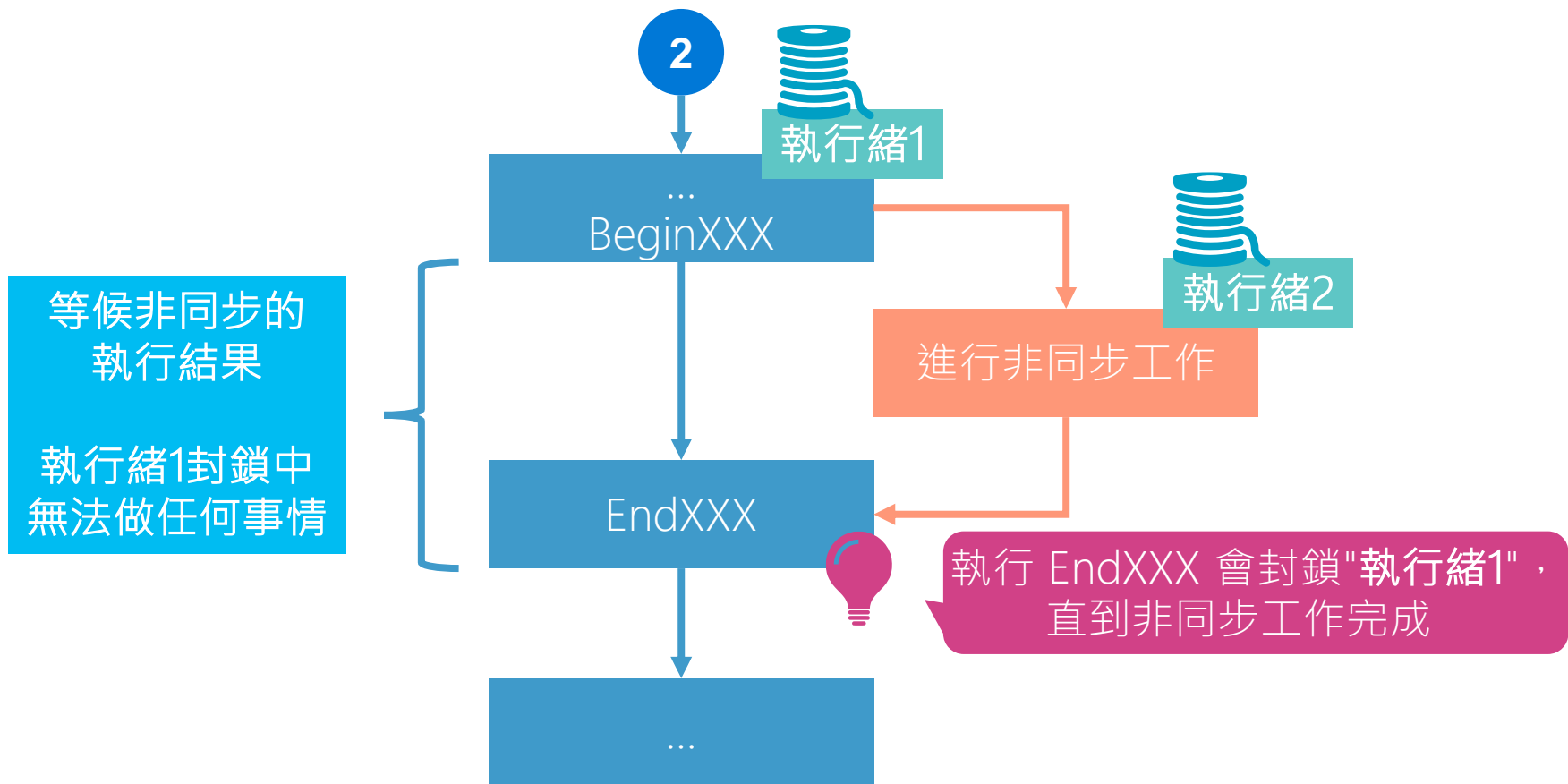
- 使用 APM 呼叫 BeginXXX 啟動非同步工作，但是，無須傳送 callback 委派方法
- 使用 **AsyncWaitHandle.WaitOne** 等候，直到整個處理程序完成(注意，此時，不是透過 callback 方法來處理後續工作)
- 由於非同步工作已經完成了，所以，我們在此呼叫了 EndXXX 方法，取得非同步工作的處理結果
- 請實際執行這個範例程式
 - 請比較與 APM + Callback 的用法差異與優缺點



使用 WebRequest.Create 工廠方法建立一個 HttpWebRequest 物件

結束非同步作業的方式

直接取得非同步作業執行結果(封鎖式等待)

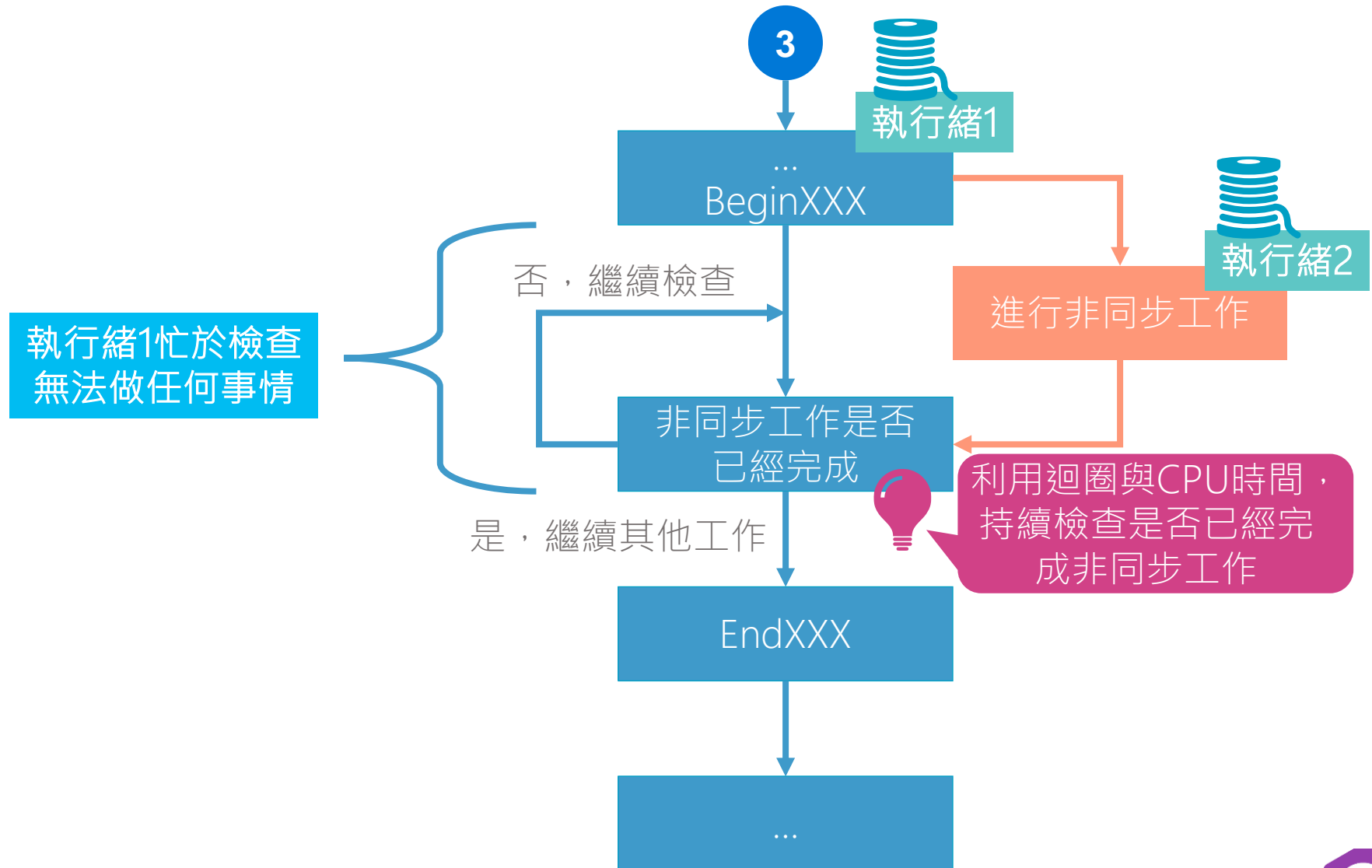


以結束非同步作業的方式封鎖應用程式執行

- 使用 APM 呼叫 BeginXXX 啟動非同步工作，但是，無須傳送 callback 委派方法
- 此時可以立即得到 IAsyncResult 型別物件，因為執行緒沒有被封鎖住，您可以在這裡處理其他工作
- **直接呼叫 EndXXX 方法**，直接等候非同步的處理最後結果 (也許是成功、也許是失敗)
- 不過此時的 Thread 是被鎖定的，也就是無法繼續執行其他工作
- 請實際執行這個範例程式
 - 請比較與 APM + Callback 的用法差異與優缺點

輪詢非同步作業的狀態 圖例

每隔一段時間，檢查非同步作業是否完成了



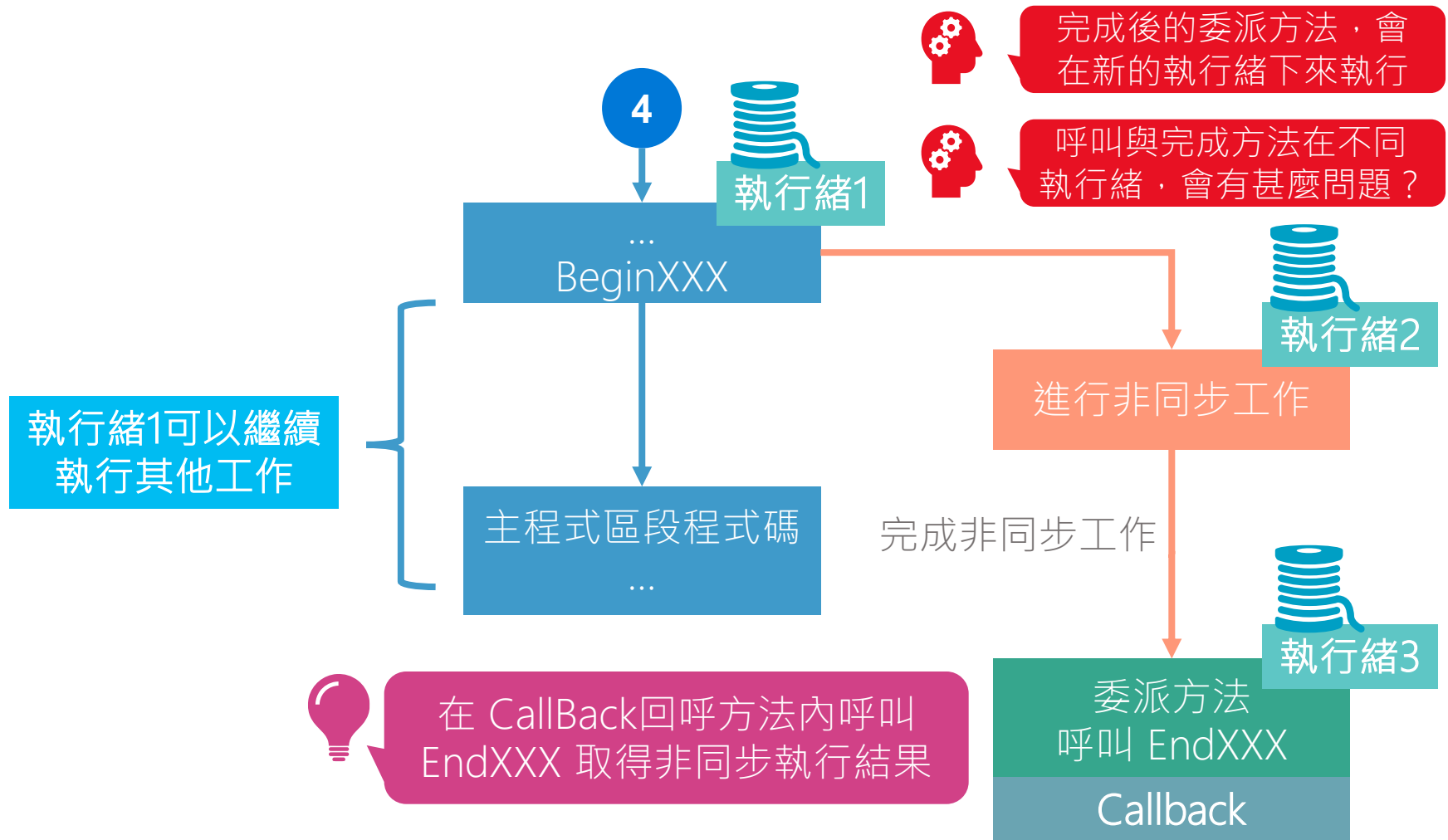
輪詢非同步作業的狀態，以便得知該非同步作業是否已經完成了

- 使用 APM 呼叫 BeginXXX 啟動非同步工作，但是，無須傳送 callback 委派方法
- 此時可以立即得到 IAsyncResult 型別物件，因為執行緒沒有被封鎖住，您可以在這裡處理其他工作
- 開始進行**輪詢非同步作業的狀態**，看看是否已經完成；我們會使用 . 句號輸出，表示這個應用程式還在執行中，並沒有封鎖起來
- 透過 EndXXX 告知非同步工作已經完成，並且取得非同步工作的最後執行結果內容
- 請實際執行這個範例程式
 - 請比較與 APM + Callback 的用法差異與優缺點

<https://docs.microsoft.com/zh-tw/dotnet/standard/asynchronous-programming-patterns/polling-for-the-status-of-an-asynchronous-operation>

使用 AsyncCallback 委派

不問經常來問，非同步作業完成後，會執行委派方法



BeginXXX & EndXXX 兩個方法在不同執行緒下執行，會有甚麼問題嗎？

使用 AsyncCallback 委派結束非同步作業

- 使用 APM 呼叫 BeginXXX 啟動非同步工作，但是，加入一個 **回呼 callback** 委派方法和 HttpWebRequest **狀態物件**
- 當非同步作業完成後，就會觸發 ResponseCallback 回呼 callback 委派方法
- 透過 callback 的參數 **IAsyncResult ar**，使用 **ar.AsyncState** as HttpWebRequest 取得狀態物件，便可以透過該狀態物件來呼叫 **EndGetResponse** 取得非同步作業執行結果。



EAP 事件架構非同步模式

Event-based Asynchronous Pattern

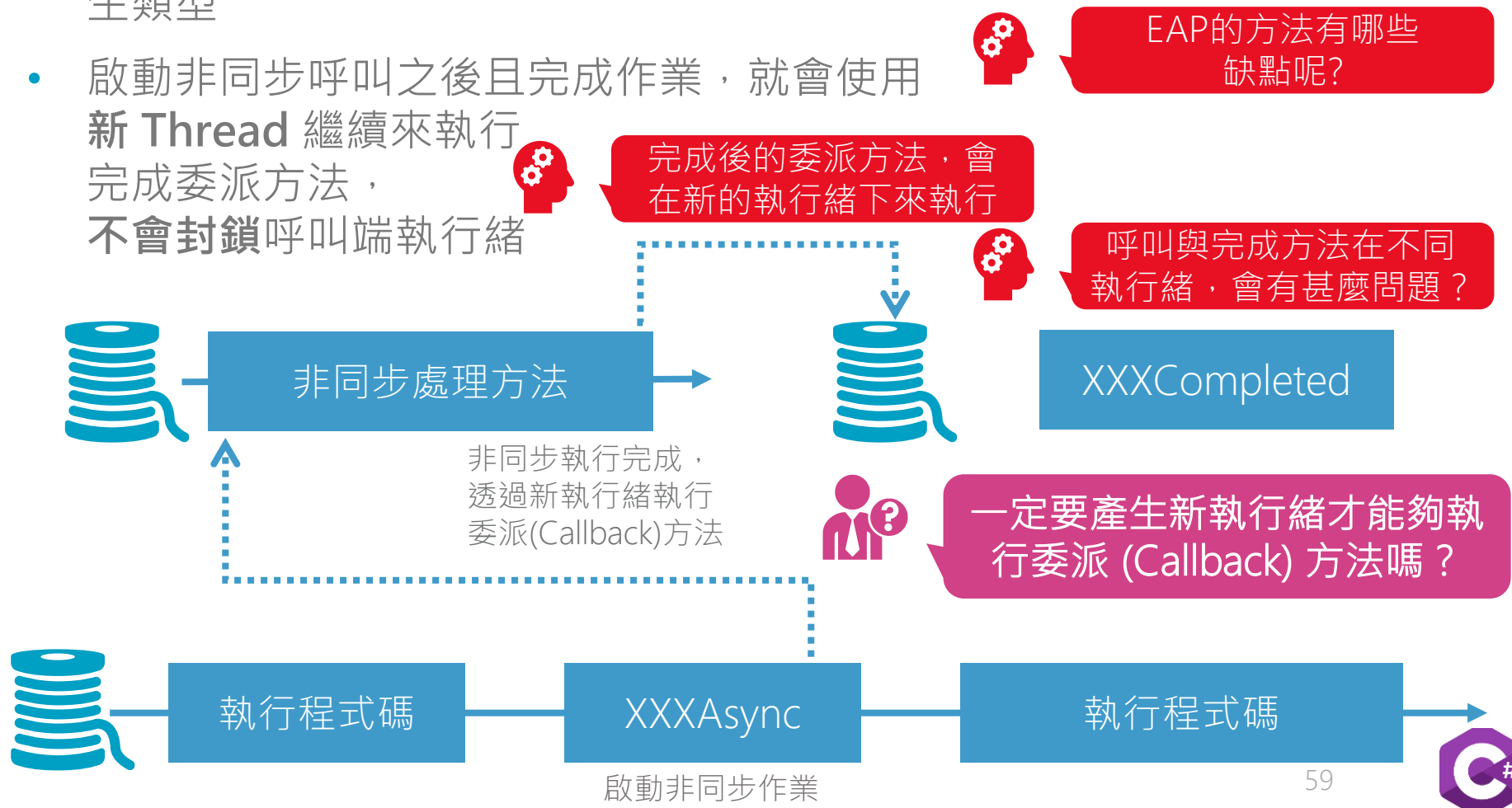


.NET 2.0



EAP 事件架構非同步模式

- 提供非同步行為的事件架構傳統模型。它需要一個具有 **Async** 尾碼的方法、一或多個事件，事件處理常式委派類型，以及 EventArgs 衍生類型
- 啟動非同步呼叫之後且完成作業，就會使用 **新 Thread** 繼續來執行完成委派方法，**不會封鎖** 呼叫端執行緒



EAP 使用範例

EAP方法

讀取資料完成後，會使用 ReadCompletedEventHandler通知作業完成

```
public class MyClass
{
    public void ReadAsync(byte[] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}
```



使用 WebClient 進行 EAP 非同步網頁存取呼叫

- 建立 WebClient 物件
- 訂閱 **WebClient.DownloadStringCompleted** (可以使用具名或者匿名 delegate / Lambda 委派方法來設計)
- 完成非同步呼叫之後，可以透過委派事件內的參數，取得此次非同步工作的失敗/成功狀態，完成結果內容
- 使用 **WebClient.DownloadStringAsync** 進行非同步方法呼叫
- 請實際執行這個範例程式
 - 請試著比較執行前後的執行緒有所不同



TAP 工作為基礎的非同步模式

Task-based Asynchronous Pattern



.NET 4.0



在 .NET Framework 4.0 以前，設計非同步方法是非常麻煩的！

- 僅有 執行緒、APM、EAP 技術可以選擇
- 無法使用同步設計邏輯來進行設計，不好維護
- 遇到 callback 內又有 callback 的情境
- 需要將程式碼從一個同步內容封裝到另一個同步內容內
- 面對例外異常將會不好處理
- 對於取得非同步執行結果需要額外進行處理



TAP 以工作為基礎的非同步模式

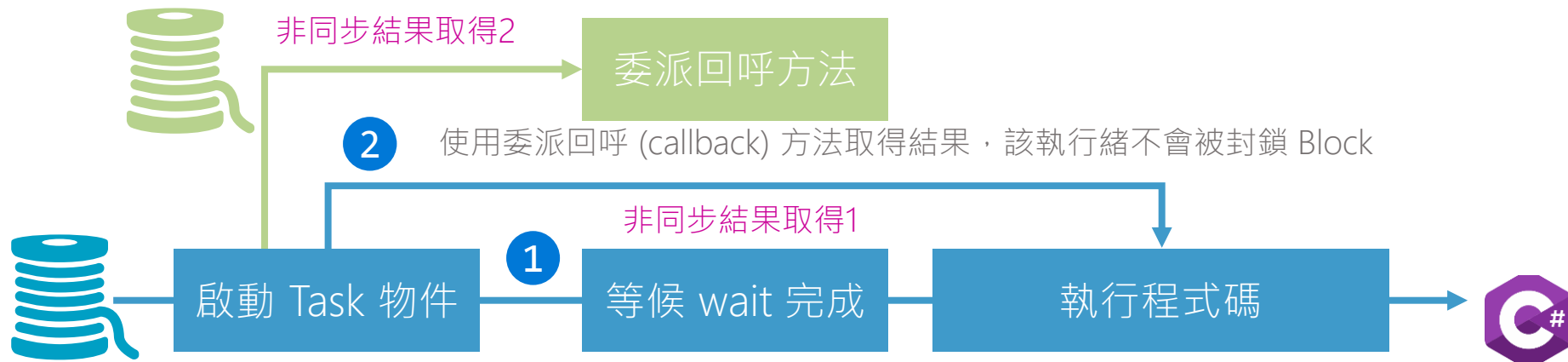


基於 TPL 所設計出來以工作為基礎的設計模式

- 使用**單一方法**表示非同步作業的啟始
 - APM 要有 **Begin** /執行、EAP 要有**啟動方法** 與 **完成事件**
- 也可以使用類似回呼 callback 委派事件
- 該模式要求用具有 **Async** 尾碼命名方法 (或者 **TaskAsync**)
- 配合 C# 5.0 **async / await** 關鍵字，可以使用**同步程式碼**設計方式來設計出非同步的應用程式碼
- 支援取消、進度、狀態、異常例外處理



非同步工作完成後，可以**強制等候**或者使用**委派回呼** Callback 接續處理



TAP 使用範例

TPL 方法

在讀取資料時候，Thread可以繼續執行其他工作，讀完後，繼續執行

```
public class MyClass
{
    public Task<int> ReadAsync(byte[] buffer, int offset, int count);
}
```



使用 HttpClient 進行 TAP 非同步網頁存取呼叫

- 透 HttpClient 非同步的方式取得網頁內容
 - `http://mocky.azurewebsites.net/api/delay/5000`
- 當要讀取網頁內容時候，使用 `httpClient.GetStringAsync(url)` 將會回傳 `Task<String>` 物件

以工作為基礎的非同步模式



什麼是「工作」(Task)



把非同步作業抽象化為 Task 類別，即 Task 物件封裝了一個非同步的作業(尚未完成作業)

- Task 類別的代表單一作業不會傳回值，而此通常以非同步方式執行
 - 但也會有例外，例如：該工作一執行，立即就結束了
 - Task<TResult> 類別代表有 TResult 型別的回傳值
- 表示有某些事情需要些時間完成
 - 計算機科學術語，代表 未來 future 與承諾 promise
 - 所謂未來與承諾，指的是將來一定會完成指定的作業
- 因為所執行的工作 Task 物件通常是從執行緒集區執行緒上取得執行緒來以非同步方式執行，而不是以同步方式在主應用程式執行緒中，您可以使用 Status 屬性，或者 IsCanceled，IsCompleted，及 IsFaulted 屬性，以判斷工作的現在執行結果狀態。
- 在某些方面，工作類似執行緒或 ThreadPool 工作項目，但為等級較高 抽象 封裝
 - 執行緒表示作業系統要處理工作的基本單元
- 比使用執行緒提供更多的程式設計控制能力與容易除錯



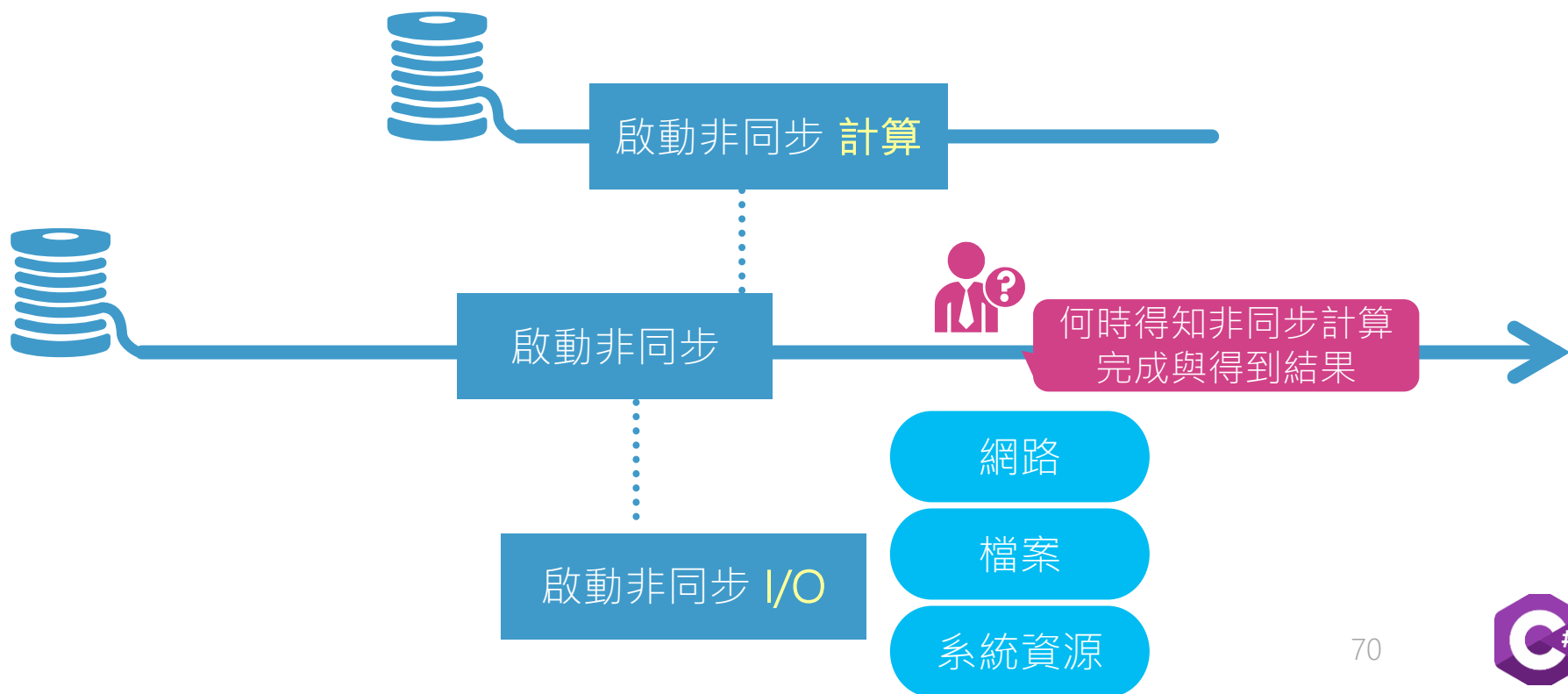
什麼是「工作」(Task)

- 最推薦的非同步開發方法
- 使用 **單一方法** 表示非同步作業**啟始**和**完成**
 - 其他模式比較
 - APM 使用 **Begin** 和 **End** 方法
 - EAP必須有一個或多個**方法**、事件處理常式**委派**類型，以及 **EventArgs** 衍生類型



為什麼要有 TAP 以工作為基礎的非同步模式

- 解決以往用執行緒與其他設計模式所存在的缺點與問題
 - 容易使用、完成，封鎖，除錯、測試、取消、例外異常、回報進度，同步處理 Synchronization
 - 讓非同步程式碼寫起來更輕鬆與容易和好維護



理解 Thread 與 Task 處理非同步的差異

- **執行緒 Thread** 代表作業系統內的執行緒，是作業系統配置處理器時間的基本單位，透過**排程器**決定執行優先順序；執行緒內容包括執行緒順暢繼續執行所需的所有資訊，包括執行緒的一組 **CPU 暫存器**和**堆疊與程式計數器 (Program Counter, PC)**
 - 使用執行緒來設計非同步工作需要注意很多細節
 - 不容易**除錯**和無法處理**例外異常**和**取得其他執行緒執行結果**
 - 使用執行緒會耗用記憶體資源(1MB)與CPU效能(內容交換)
 - 可以透過**執行緒集區(Thread Pool)** 來改善
- **工作 Task** 表示一個非同步操作物件，屬於 TPL 中的一部分
 - 工作可以使用非同步的方式來執行
 - 可以得知工作**何時結束**與**取得執行結果**
 - 工作完成之後可以設定**繼續執行**的指定程式碼
 - 例外異常發生時候將會包裝在工作內，**不會造成程式崩壞**
 - 可以**取消工作**與回報工作**處理進度**
 - 搭配 **async, await** 關鍵字會更容易寫出非同步應用程式碼



工作 Task 的 4 種用法

- 建立 HttpClient 物件，並且取得讀取網頁內容工作物件

```
var client = new HttpClient();  
Task<string> task = client.GetStringAsync(url);
```

此處將回傳一個工作物件

.NET Framework 4.0

- 使用同步方式取得非同步工作結果

```
string htmlSync = task.Result;
```

造成執行緒封鎖 Block

- 使用接續工作取得非同步工作結果

```
task.ContinueWith(ContinueTask =>  
{  
    string htmlContinue = task.Result;  
});
```

建立一個回呼 (Callback)

.NET Framework 4.5

- 使用等候 await 關鍵字來等候工作結果

```
string htmlAwait = await task;
```

此處將得到一個字串物件

由編譯器產生等候的最終程式碼



Task 的 9 種使用情境

- 產生 Creation
- 啟動 Start
- 傳入參數 Parameter
- 等候結束 Wait
- 繼續 Continuation
- 傳回值 Return
- 取消 Cancellation
- 進度 Progress
- 異常與除錯 Exception



工作的產生、啟動、傳入參數



工作的產生、啟動、傳入參數

執行緒的集區 - 自動執行

- `ThreadPool.QueueUserWorkItem(state => 我的工作());`

使用 Task 類別 - 手動啟動

- `var t = new Task(我的工作);`
 - `t.Start();`
-  工作冷啟動：建立工作後需要手動啟動工作執行

靜態方法產生工作 - 自動執行

- `Task.Factory.StartNew(我的工作);`

StartNew 簡單替代方案 - 自動執行

- `Task.Run(() => 我的工作());`

.NET 2 開始提供

- `ThreadPool.QueueUserWorkItem(state => 我的工作有參數(1));`

.NET 4 開始提供

- `var t = new Task(我的工作有參數,2);`
`t.Start();`

.NET 4 開始提供

- `Task.Factory.StartNew(我的工作有參數, 3);`

.NET 4.5 開始提供

- `Task.Run(() => 我的工作有參數(4));`



工作熱啟動：建立工作後，工作將會自動執行

練習情境

各種非同步工作的建立與啟動方法

D006. 建立和起始非同步工作

- 方法1
 - 使用**執行緒**的集區 .NET 2 開始提供，建立完成後，該非同步工作，就會自動啟動
- 方法2
 - 使用 **Task 類別**，在建立物件的時候，傳入非同步的委派方法 .NET 4 開始提供，產生完 Task 物件之後，需要呼叫 **Start()** 方法，來啟動這個非同步委派方法
- 方法3
 - 使用 **Task.Factory.StartNew** 工廠方法產生工作，這些 Factory 方法用於建立及設定 Task 和 Task<TResult> 執行個體。等同於建立藉由呼叫無參數的類別 TaskFactory.TaskFactory() 建構函式，使用 Task.Factory 靜態方法產生的非同步工作，會立即開始執行
- 方法4
 - 將指定在 **ThreadPool** 執行工作排入佇列，並傳回該工作的工作控制代碼 .NET 4.5 的 Task 類別新增了靜態方法 **Task.Run**



使用執行緒類別(Thread)來產生一個執行緒物件做法，請參考 [前面範例](#)

- 方法1
 - 使用**執行緒集區** 取得執行緒並傳入參數的用法
- 方法2
 - 使用 **Task 類別** 傳入參數的用法
- 方法3
 - 使用 **Task.Factory** 靜態方法產生工作 傳入參數的用法
- 方法4
 - **Task.Run** 靜態方法 傳入參數的用法



在非同步方法中，將會取得一個 Guid 物件參數

- 目的
 - 使用多個非同步工作，針對特定網站網址，進行壓力測試。
- 提示
 - 請使用 **Task.Run** 建立 10 個工作
 - 可以使用 `HttpClient.GetStringAsync(url).Result` 取得結果
 - 在每個工作使用 `HttpClient` 物件，開啟 HTTP 連線到指定網址
 - 每個工作開啟**兩次**該 URL
 - 列出 HTTP 開始時間、Web API 回傳內容、結束時間
 - 有錯誤，要列出提示訊息

工作的狀態：Task.Status 屬性之列舉值

- **Created**
 - 工作已初始化但尚未排程。
- **WaitingForActivation**
 - 工作正在等候由 .NET Framework 基礎結構從內部啟動並排程。
- **WaitingToRun**
 - 工作已排定執行，但尚未開始執行。
- **Running**
 - 工作正在執行，但尚未完成。
- **WaitingForChildrenToComplete**
 - 工作已完成執行，而且在暗中等候附加的子工作完成。
- **RanToCompletion**
 - 工作已成功完成執行。
- **Canceled**
 - 工作確認取消動作，不論是因為工作在語彙基元處於信號狀態時使用自己的 `CancellationToken` 擲回 `OperationCanceledException`，或是工作的 `CancellationToken` 信號在工作開始執行之前便已存在。如需詳細資訊，請參閱工作取消。
- **Faulted**
 - 工作因未處理的例外狀況而完成。



工作的狀態：工作物件的其他屬性

- Task.IsCompleted
 - 這個值表示工作是否**已經完成**。
- Task.IsCanceled
 - 這個值表示工作是否因**取消**才完成執行。
- Task.IsFaulted
 - 這個值表示工作是否因**未處理的例外狀況**才完成。



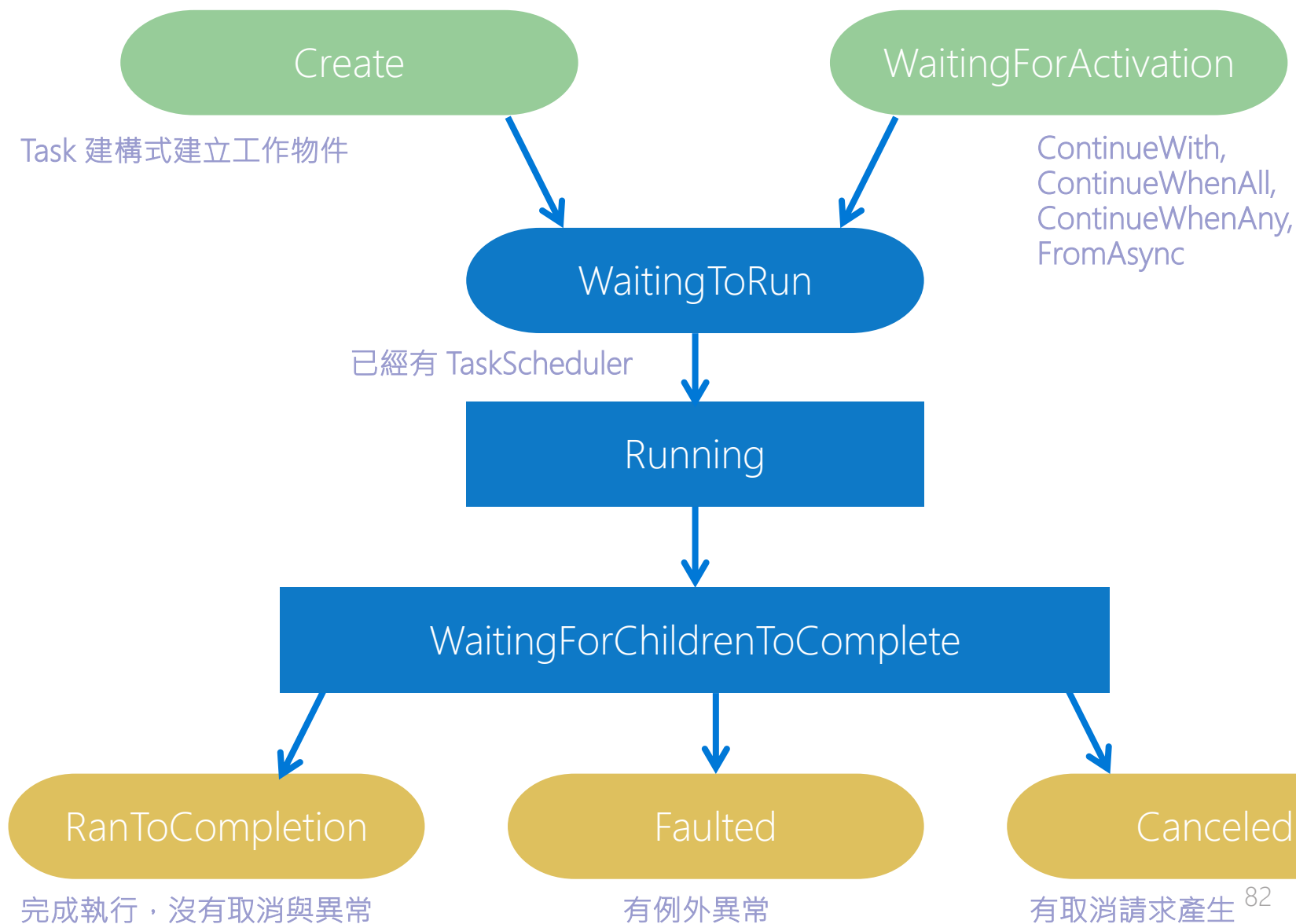
如需精準判斷非同步狀態，必須使用 Task.Status

- 意味著你不能使用 `async / await` 來判斷工作狀態

Status	IsCompleted	IsCanceled	IsFaulted
other	✗	✗	✗
RanToCompletion	✓	✗	✗
Canceled	✓	✓	✗
Faulted	✓	✗	✓

<https://blog.stephencleary.com/2014/06/a-tour-of-task-part-3-status.html>

委派類型的工作 (Delegate Tasks) 狀態

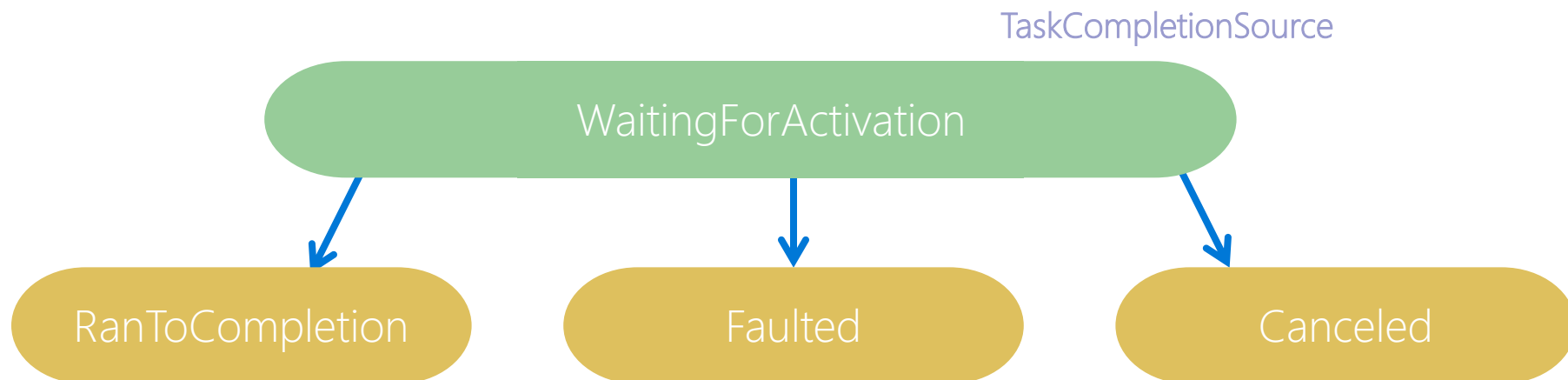


承諾類型的工作 (Promise Tasks) 狀態

`Task.IsCanceled` 為真，若且唯若 (if and only if)
`Task.Status` 等於 `TaskStatus.Canceled`.

`Task.IsFaulted` 為真，若且唯若 (if and only if)
`Task.Status` 等於 `TaskStatus.Faulted`。另外, `Task.Exception` 將會有例外異常物件。

`Task.IsCompleted` 為真，若且唯若 (if and only if)
`Task.Status` 等於這三種狀態的其中一種 `TaskStatus.RanToCompletion`,
`TaskStatus.Canceled`, `TaskStatus.Faulted`



了解各種工作物件的狀態列舉值變化情形

- 設定方案管理員要 顯示所有檔案
- 將 委派類型的工作.cs 檔案加入專案，並排除其他 .cs
 - 工作將會於3秒鐘後結束，看看輸出變化
 - 將 `Thread.Sleep(300);` 註解起來，看看輸出變化
 - 將 `throw new Exception("喔喔，發生例外異常");` 解除註解起來，看看輸出變化
- 將 承諾類型的工作.cs 檔案加入專案，並排除其他 .cs
 - 使用該敘述，觀察正常完成的狀態變化
`tcs.SetResult(null);`
 - 使用該敘述，觀察工作取消的狀態變化
`tcs.SetCanceled();`
 - 使用該敘述，觀察工作有例外異常的狀態變化
`tcs.SetException(new Exception("喔喔，發生例外異常"));`



工作的狀態將會不斷地變化中，讀取工作的狀態，不會造成 封鎖 Block

- 目的
 - 使用多個非同步工作對不同網址發出要求，列出失敗清單
- 提示
 - 請使用 **Task.Run** 建立 3 個工作
 - 可以使用 `HttpClient.GetStringAsync(url)` 取得結果
 - 請設定 3 個不同網址，兩個有效網址，一個無效網址
 - 請使用 `try/catch` 抓取工作例外
 - 請使用 `Task.WhenAll()` 等待工作結束
 - 請檢查所有工作中發生錯誤的詳細內容



工作等候結束與接續



工作等候結束與接續

- TPL

```
Task<int> 新的工作TPL =  
Task.Run() =  
{  
    return 42;  
});
```

會封鎖
執行緒

新的工作TPL.**Wait()**;

```
Console.WriteLine(新的工作  
TPL.Result);
```

會封鎖執行緒
如果上面沒有 .Wait()

- **await**關鍵字

```
Task<int> 新的工作Task =  
Task.Run() =>  
{  
    return 42;  
});
```

等候非同
步結果

int foo最後結果 = **await** 新的工
作Task;

```
Console.WriteLine(foo最後結果);
```

各種不同取得工作執行結果的用法

- 使用 `Task.Run` 建立非同步的工作
 - 模擬 3 秒鐘後才會回傳結果
- 呼叫 `Task.ContinueWith` 方法完成時的接續執行
- 透過 `Task.Result` 來取得非同步執行結果
- 使用 `Task.Wait` 方法與 `await` 關鍵字等候非同步作業完成
- 了解 `Task.Result` / `Task.Wait` / `await` 使用上的差異

等待 Wait 與取得結果之封鎖 Block 非同步用法

- `MethodAsync().Wait()`
 - 封鎖並等待 `MethodAsync` 方法執行完成
- `MethodAsync().Result`
 - 封鎖並等待 `MethodAsync` 方法執行完成後的回傳值
- `Task.WaitAll()` / `Task.WaitAny()`
 - 封鎖並等候 所有 / 任一 工作完成



```
thread.Join()  
task.Wait()  
task.Result  
tasks.WaitAll()  
task.WaitAny()
```

使用 封鎖 Block 方式來等候
非同步計算何時完成與回傳結果？

對於整個系統會有什麼影響嗎？

實作練習

D004. 同步設計披薩製作

請使用 **同步程式** 設計方法，設計一個披薩製作過程的程式

- 目的：您需要利用程式可以產生出一個披薩
- 步驟 1：烤箱預熱(30秒)
- 步驟 2：製作麵團(3秒)
- 步驟 3：發麵(8秒)
- 步驟 4：準備醬料(2秒)
- 步驟 5：準備配料(2秒)
- 步驟 6：製作披薩餅皮與
塗抹醬料和放置配料(3秒)
- 步驟 7：烤製披薩(6秒)
- 步驟 8：準備餐具與飲料(2秒)
- 步驟 9：披薩完成，開始食用(1秒)

```
D:\Vulcan\GitHub\NET-Async\NET非同步程式設計\D004.同步設計披薩製作\bin\Debug\D0...
開始進行製作披薩...
烤箱預熱中，預計 30 秒鐘 [Thread:1]
烤箱預熱 完成
製作麵團中，預計 3 秒鐘 [Thread:1]
麵團製作 完成
麵團發麵中，預計 8 秒鐘 [Thread:1]
麵團發麵 完成
準備醬料中，預計 2 秒鐘 [Thread:1]
準備醬料 完成
準備配料中，預計 2 秒鐘 [Thread:1]
準備配料 完成
製作披薩餅皮與塗抹醬料和放置配料中，預計 3 秒鐘 [Thread:1]
製作披薩餅皮與塗抹醬料和放置配料 完成
烤製披薩中，預計 6 秒鐘 [Thread:1]
烤製披薩 完成
準備餐具與飲料中，預計 2 秒鐘 [Thread:1]
準備餐具與飲料 完成
披薩完成 開始食用，預計 1 秒鐘 [Thread:1]
披薩完成 開始食用 完成
同步設計披薩製作共花費:57 秒
Press any key for continuing...
```



請利用 D004.同步設計披薩製作 專案來改成非同步運作程式碼

請使用 **非同步程式** TAP設計方法，設計一個披薩製作過程的程式

- **提示**：可以使用 **Task.Run** 靜態方法，執行一個非同步方法
- **提示**：使用工作物件的 **Wait** 方法等待非同步結束
- 注意非同步方法在哪個執行緒下執行
- 這個非同步設計**改善了**哪些**問題**
- 若披薩製作到一半，突然客戶說**不要買了**，這個時候**該如何處理**呢？

```
D:\Vulcan\GitHub\NET-Async\NET非同步程式設計\D005.非同步TAP設計披薩製作\bin\Debug\D0...
開始進行製作披薩...
製作麵團中，預計 3 秒鐘 [Thread:1]
烤箱預熱中，預計 30 秒鐘 [Thread:3]
麵團製作 完成
準備醬料中，預計 2 秒鐘 [Thread:1]
麵團發麵中，預計 8 秒鐘 [Thread:4]
準備醬料 完成
準備配料中，預計 2 秒鐘 [Thread:1]
準備配料 完成
麵團發麵 完成
製作披薩餅皮與塗抹醬料和放置配料中，預計 3 秒鐘 [Thread:1]
製作披薩餅皮與塗抹醬料和放置配料 完成
烤箱預熱 完成
準備餐具與飲料中，預計 2 秒鐘 [Thread:1]
烤製披薩中，預計 6 秒鐘 [Thread:4]
準備餐具與飲料 完成
烤製披薩 完成
披薩完成 開始食用，預計 1 秒鐘 [Thread:1]
披薩完成 開始食用 完成
同步設計披薩製作共花費:37 秒
Press any key for continuing...
```



若使用 APM 或者 EAP 來設計同樣需求，那又會如何呢？

await 的主要目的

- 是一個 C# 5.0 新增的關鍵字
- 套用至非同步方法中的工作，以在執行方法中插入暫停點，直到**等候 await**的工作完成為止
 - 這裡所謂的**等候**，指的是在遇到 **await** 關鍵字之後，該函數方法就直接 **Return** 了，而不是將這個執行緒**封鎖 Block**，等待 **wait** 非同步方法完成的結果。
- 將非同步程式**設計邏輯**，使用同步程式設計邏輯來進行
- 進行非同步計算呼叫，在執行緒等候結果期間**不會封鎖**
- **繁雜的工作** (在等待過程中會用到的各種程式碼)，**編譯器**都幫我們處理好了 → **編譯器會自動產生這些程式碼**



請說明 封鎖 Block 與 等候的差異

如何在 .NET Framework 4.0 專案內使用 await

- 安裝 Microsoft.Bcl.Async NuGet 套件

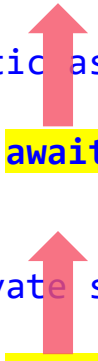


遇到 await 關鍵字，函數方法就直接 Return 的意義？

- 在這底下的例子，當執行 await MyAsyncMethod 方法後，會發生什麼結果
 - 因為該主執行緒因為遇到 await 關鍵字，就直接結束該主執行緒，也造成整個處理程序結束執行
- 所以，在 Console 類型專案中，不能這樣用 !!

```
class Program
{
    static async void Main(string[] args)
    {
        await MyAsyncMethod();
    }

    private static async Task MyAsyncMethod()
    {
        await Task.Run(() => Thread.Sleep(3000));
        return;
    }
}
```



請了解後面介紹的前景與背景執行緒的介紹與練習


主執行緒與背景執行緒特性


- .NET 處理程序 **Process** 啟動後，僅會有一個**前景執行緒**
- 執行緒可以**產生**出許多**前景**或者**背景**執行緒
- 在應用程式**結束**執行的時候
 - 所有前景執行緒**都要結束**執行，否則無法結束執行
 - 所有前景執行緒已經結束了，**不管**背景執行緒是否還有在執行，該處理程序**都會結束**
- **前景/背景**執行緒可以透過 **Thread** 類別產生物件來設定
- 從**執行緒集區**取得的執行緒，一定是**背景執行緒**
- **Task.Run** 從執行緒集區取得一個執行緒，因此是**背景執行緒**




- 這個範例中將會有四個執行緒
 - 主執行緒 → 前景執行緒：將會輸出 **M**
 - 使用 Thread 類別建立的 前景執行緒：將會輸出 **F**
 - 使用 Thread 類別建立的 背景執行緒：將會輸出 **B**
 - 使用執行緒集區取得的執行緒 → 背景執行緒：將會輸出 **P**
- 背景類型執行緒將會輸出 300000 次
- 前景類型執行緒將會輸出 600 次
- **前景執行緒執行完畢後，按下任一按鍵，整個程序將會結束**
 - 此時，就算**背景執行緒**尚未結束執行，也會被**強制結束**

async 的用法

- 當方法內有使用 **await** 關鍵字，方法需要標示 **async** 修飾詞
 - 使用 **async** 修飾詞可將方法、**Lambda 運算式**或**匿名方法**指定為**非同步**計算 (方法內要有 **await** 才會成立)
- **async Task MethodAsync()**
- **async Task<TResult> MethodAsync()**
- **async ()=>{ }**
- **async void MethodAsync()** 

這裡可以使用 await 嗎？
- 

僅適合用於事件之委派方法
- **async string MethodAsync()**


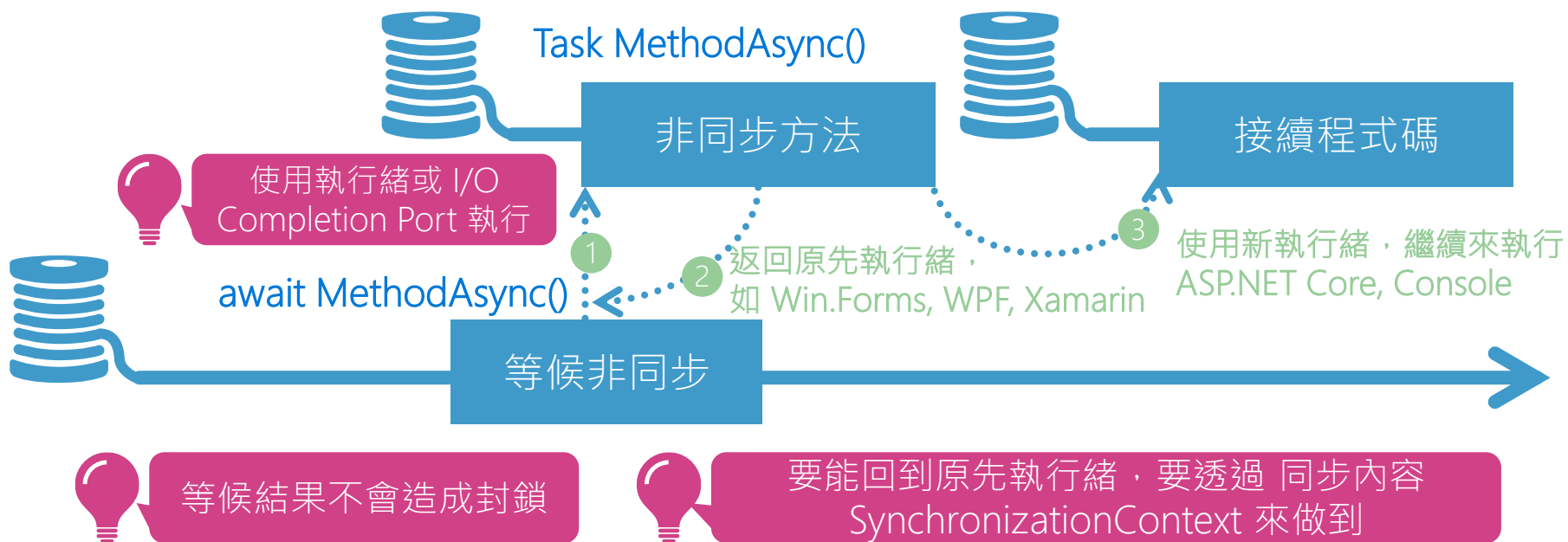
為什麼這樣的用法不正確呢？

async / await 示意流程圖



非同步工作完成後，要選擇回原執行緒或者從新執行緒來執行，可以依情境判斷

`await MethodAsync.ConfigureAwait(false)`



async / await

遇到 await 關鍵字，會先捕捉現在執行緒的 SynchronizationContext，當完成非同步計算之後，透過 SynchronizationContext 讓剩餘程式碼能夠在原先執行緒下繼續執行

1. **async** 修飾詞的非同步方法

- 可以使用 await 來指定暫停點

2. 使用 HttpClient 非同步方法，取得網頁內容

- 傳回 Task、Task<TResult> 物件

記住現在的位置

3. **await** 運算子會套用至非同步方法中的工作，以在執行方法中插入暫停點，直到等候的工作完成為止，並返回呼叫端；await 運算式不會封鎖其執行所在的執行緒。但是它會造成編譯器將其餘非同步方法註冊為所等候工作的接續。

取回剛剛記住的位置

4. 當工作完成時，它會叫用其接續，並從中斷處繼續執行非同步方法。可能是原有執行緒(UI)或新執行緒

GUI

ASP.NET / Console

```
static async Task<int> AccessTheWebAsync()  
{  
    string urlContents = await httpClient.GetStringAsync("https://host.com");  
    return urlContents.Length;  
}
```

透過編譯器重新編碼，
將程式碼拆成兩塊

可能是原有執行緒(UI)或新執行緒



編譯器對 `async` 和 `await` 做了甚麼事？

- 建立一個有限狀態機
 - 將 `async` 方法內的全部移植到有限狀態機
 - 在 `async` 方法內初始化與啟動有限狀態機
 - 呼叫 `MoveNext()` 開始運行
- 在有限狀態機內
 - 等候非同步工作完成的物件
 - 取得用來等候這個工作的`Getawaiter()`
 - 設定 `Taskawaiter` 物件的等待非同步工作完成事件
 - 在有限狀態機的`IAsyncStateMachine.MoveNext()`不斷運行，直到非同步工作處理完成



Async Await ， C# 編譯器做了些甚麼事情呢-1？



Async Await ， C# 編譯器做了些甚麼事情呢-2？



簡單理解與使用 `async / await`

- 在方法內若有使用到 `await` 關鍵字
 - 在方法簽名要有 `async` 修飾詞
- 方法簽名有 `async` 修飾詞
 - 該方法的回傳型別必須從
 - `void` 改成 `Task`
 - `TResult` 改成 `Task<TResult>`
- 在執行該方法內敘述，遇到 `await` 關鍵字時候
 - 非同步執行 `await` 等候的方法
 - 暫時 `Return` 該方法
- 當非同步作業完成後，會從 `await` 關鍵字繼續執行



Task 非同步工作 與 async 非同步方法

- Task 非同步工作
 - Task 類別定義於 TPL 中 (.NET Framework 4.0)
 - Task 可以有兩種模式
 - 委派 **Delegate** 工作 (CPU Bound)
 - 承諾 **Promise** 工作 (I/O Bound)
- async 非同步方法
 - 提供一個簡單的方式讓您建構出非同步的運算
 - 方法內需要**搭配 await 關鍵字**
 - 該方法內使用同步設計邏輯設計出一個非同步運算
 - 當**遇到非同步呼叫**，將會**回到呼叫端**
 - 當非同步工作完成運算之後，將會繼續執行剩下流程

了解『非同步方法』與使用『await 非同步方法』執行順序與執行緒的變化

- 設定方案管理員要 顯示所有檔案
- 將 使用同步的方式.cs 檔案加入專案，並排除其他 .cs
- 將 使用await與非同步方法.cs 檔案加入專案，並排除其他 .cs
- 同樣的程式碼，將入了 **await** 與 **Task** 將會有不同的輸出結果 (**注意執行緒的變化**)，您可以說明這**兩者的差異**嗎？

使用同步的方式

```
Checkpoint: 1, thread: 1
Checkpoint: 2, thread: 1
Checkpoint: 3, thread: 1
模擬寫到資料庫內，約2秒鐘
Checkpoint: 4, thread: 1
Checkpoint: 5, thread: 1
模擬主程式正在忙碌中，約1秒鐘
Checkpoint: 6, thread: 1
Press any key for continuing...
```

使用await與非同步方法

```
Checkpoint: 1, thread: 1
Checkpoint: 2, thread: 1
Checkpoint: 3, thread: 3
模擬寫到資料庫內，約2秒鐘
Checkpoint: 5, thread: 1
模擬主程式正在忙碌中，約1秒鐘
Checkpoint: 4, thread: 3
Checkpoint: 6, thread: 3
Press any key for continuing...
```



不是方法上加入了 async 就是一個非同步工作

想要使用 `async / await` 不會遇到相關
奇怪疑問，就需要充分了解什麼是
同步內容 `SynchronizationContext`

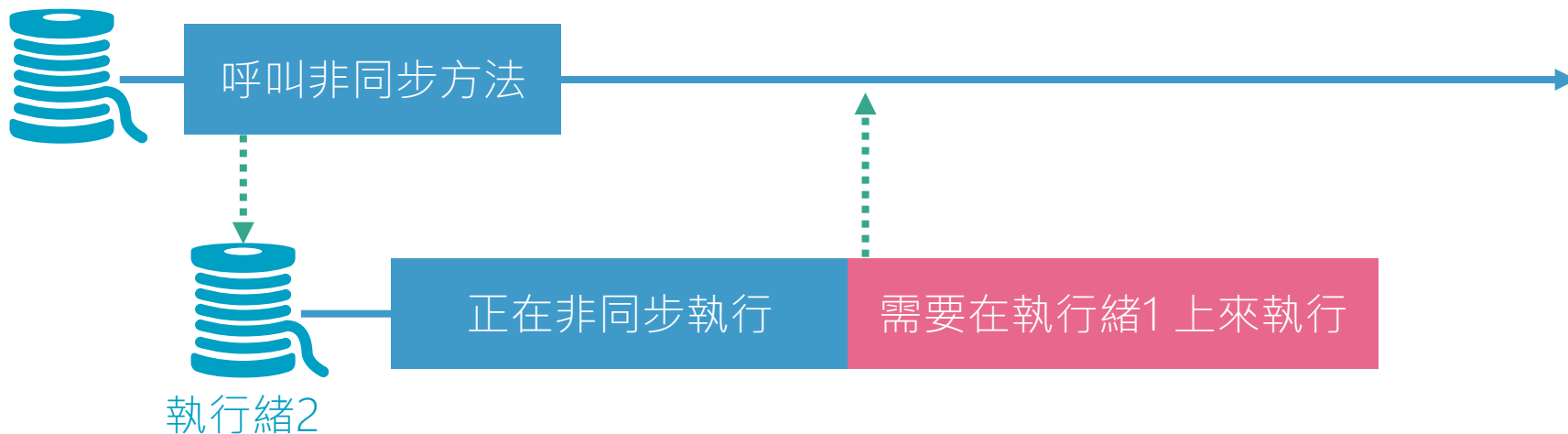


可是，`SynchronizationContext Class` 有看沒有懂

為什麼需要用到同步內容 圖解

- 執行緒1 啟動另外一個非同步執行緒2
- 執行緒2 完成後，接下來的程式碼想要在執行緒1上來執行

執行緒1



為什麼需要用到同步內容 (SynchronizationContext)

- 某些**工作需求**或者**資料物件**，僅能夠在**特定執行緒**下執行程式碼，才能夠存取與完成相關工作。
 - 也就是說，在其他**多執行緒**下，是**無法**處理這些工作與存取這些物件
- 所以，當**背景執行緒**或者**非同步作業**執行完成後，此時**執行緒**是**無法處理**接下來的工作與存取到相關物件(因為需要在**特定的執行緒**下才能夠成功運作)
 - 因此，需要一種機制，可以讓在背景執行緒下，指定一段**委派程式碼**，讓這些委派程式碼可以在**特定執行緒**下來執行
- 因此，解決方案就是

同步內容 (SynchronizationContext)



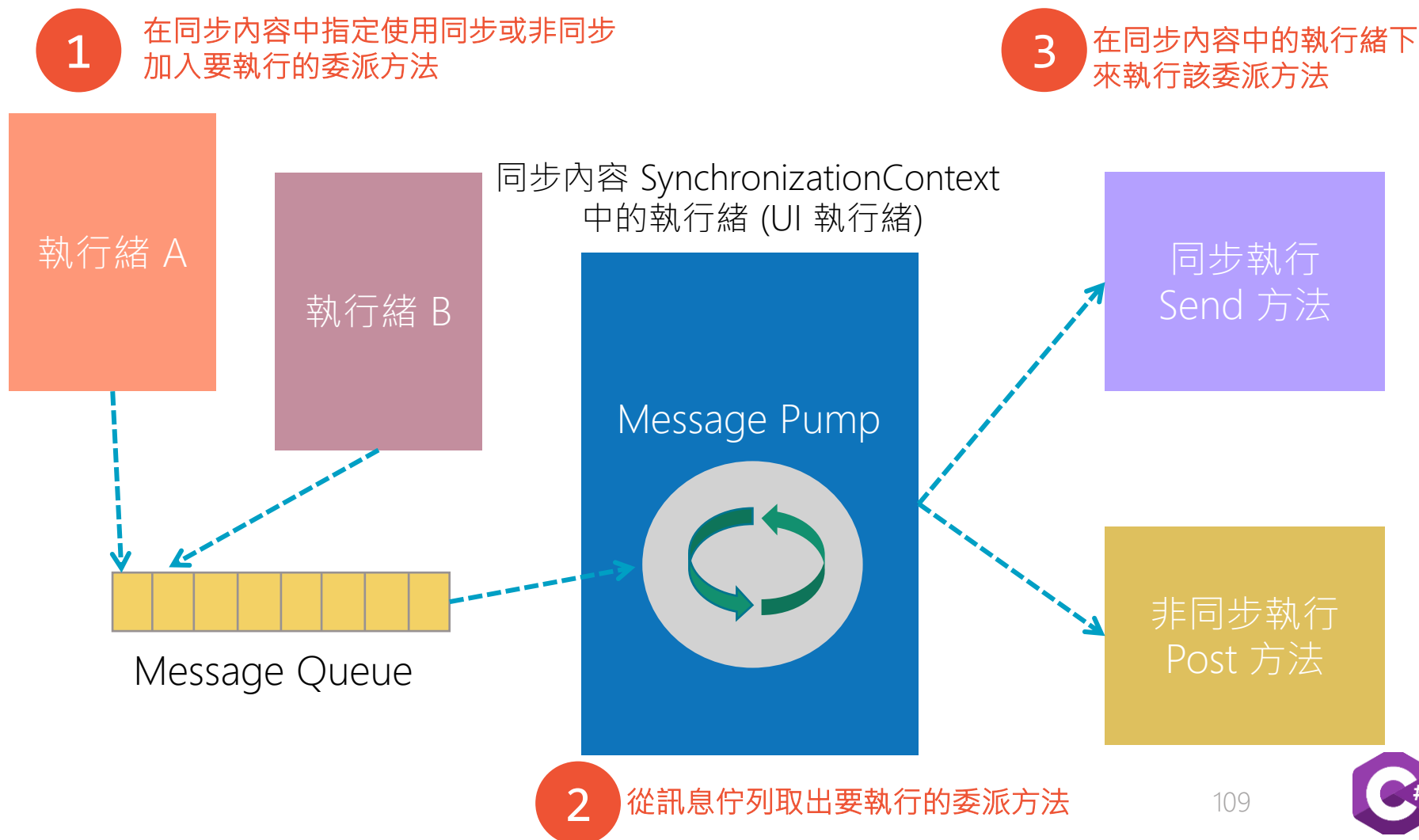
執行緒的同步內容 (SynchronizationContext)

- 同步內容是一種抽象概念，將會表示執行緒程式碼正在執行的位置
- 每個執行緒可以透過 `SynchronizationContext.Current` 屬性來取得當下執行緒的同步內容 (有可能是 `null`)
- 透過同步內容可以讓程式設計師在任何執行緒下，指定某段程式碼要在此同步內容的執行緒下來執行
 - 白話說，該同步內容屬於執行緒A，可以在執行緒B，指派某段程式碼，透過同步內容使其在執行緒A下來執行
 - 這些委派程式碼可以採用同步或者非同步方式來執行
 - 同步內容內使用 `Queue` 來儲存要執行的委派程式碼
- 上述內容以技術角度說明為
 - 允許執行緒透過將工作單元(unit of work)封裝 (Marshal)之後，傳遞給特定執行緒來執行



GUI 同步內容 SynchronizationContext 運作機制

- 運作流程圖



await 與 同步內容

遇到 await 關鍵字，會先捕捉現在執行緒的 SynchronizationContext，當完成非同步計算之後，透過 SynchronizationContext 讓剩下程式碼能夠在原先執行緒下繼續執行

- await 透過了 同步內容，讓非同步工作執行完成後，可以讓後續程式碼在同步內容的執行緒下來繼續執行
- 回顧剛剛說明的編譯將將 await 前後兩段程式碼拆開了
- 在步驟3 將會透過 SynchronizationContext.Current 屬性來取得當下執行緒的同步內容
- 當非同步工作完成後(此時執行緒將會是在非同步工作內使用的執行緒)，會透過剛剛記錄下來的同步內容物件，把底下黃色區塊的程式碼，在同步內容下的執行緒來繼續執行這些程式碼

透過編譯器重新編碼，將程式碼拆成兩塊

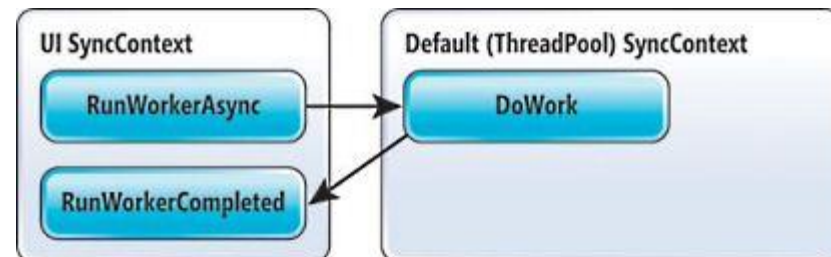
```
static async Task<int> AccessTheWebAsync()  
{  
    string urlContents = await httpClient.GetStringAsync("https://host.com");  
    return urlContents.Length;  
}
```


執行緒的同步內容 (SynchronizationContext)

- 在 .NET 的不同開發框架下，都有實作出同步內容類別，每個同步內容類別都有其特殊需求。
 - **GUI** 類型專案的同步內容將會為 **UI/Main 執行緒**
 - **ASP.NET** 類型專案的同步內容將會為當時**HTTP請求的執行緒**
- Windows Form
 - UI Thread (WindowsFormsSynchronizationContext)
- WPF
 - UI Thread (DispatcherSynchronizationContext)
- ASP.NET
 - AspNetSynchronizationContext
- **ASP.NET Core**
 - **無**

為什麼在 GUI開發下使用
BackgroundWorker的完成事件來更新 UI
控制項，不會有問題

當BackgroundWorker執行RunWorkerAsync，將會捕捉當前同步內容；當完成後便會在這個同步內容環境下執行RunWorkerCompleted事件委派方法



使用 SynchronizationContext 的優缺點

- 優點
 - 對於 GUI 類型的專案來說
 - 可以輕易地選用 UI 執行緒 (可以更新 UI 控制項)
 - 對於 ASP.NET 類型的專案來說
 - 多執行緒環境下更精準的選用每個 Request 使用的執行緒
- 缺點
 - 容易造成死結
 - 執行效能較差



不使用 SynchronizationContext 的優缺點

- 優點
 - 執行效能較好
 - 不容易造成死結
- 缺點
 - 對於 GUI 類型的專案來說
 - 修改 UI 控制項的屬性，只能夠在專屬執行緒下運行

```
private void Btn_Click(object sender, RoutedEventArgs e)
{
    ThreadPool.QueueUserWorkItem(x => {
        btn.Content = "I'm a button";
    });
}
```

透過非 UI 執行緒來更新 UI 控制項，將會得到底下例外異常

System.InvalidOperationException: '呼叫執行緒無法存取此物件，因為此物件屬於另一個執行緒。'

- 對於 ASP.NET 類型的專案來說 (除了 ASP.NET Core 以外)
 - 背景執行緒無法存取到 HttpContext 物件
 - HttpContext 僅會存在於當時處理的請求執行緒內

await 在不同開發框架下的運行說明

- 建立一個執行緒是相當耗費計算成本的(配置記憶體等等)
- 對於 **GUI** (Windows Forms / WPF) 專案
 - 透過 **同步內容 (SynchronizationContext)**，讓 **await** 後的程式碼在 UI 執行緒下能夠繼續執行
- 對於 **ASP.NET** 專案
 - 透過**同步內容**，讓 **await** 之後的程式碼，在新的執行緒下，可以繼續取得原有執行緒下的執行環境內容
- 對於 **Console** 專案
 - **沒有用到**同步內容，使得 **await** 之後的程式碼在新的執行緒下繼續執行
- 對於 **ASP.NET Core** 專案
 - 如同 Console 專案運作



ASP.NET 專案對於 同步內容 (SynchronizationContext) 的使用

- 請執行這個專案，了解執行後的輸出結果
- 為什麼在 Task.Run 內的委派方法，無法讀取到

HttpContext.Current

目前 HTTP 請求 Request 的 HttpContext 執行個體，
只能夠在這個請求的執行緒下才能夠讀取到

準備結束非同步工作 / 是否要回到原先執行緒的同步內文中 **True**執行緒 Thread → 7

工作排程 Task scheduler → System.Threading.Tasks.ThreadPoolTaskScheduler

同步內文 SynchronizationContext →

Http內文 Http Context →

因為與這次請求所使用的執行緒不同

非同步工作 已經完成

執行緒 Thread → 7

工作排程 Task scheduler → System.Threading.Tasks.ThreadPoolTaskScheduler

同步內文 SynchronizationContext → System.Web.AspNetSynchronizationContext

Http內文 Http Context → System.Web.HttpContext

- 修正 continueOnCapturedContext = false，再度執行，
了解到有何差異

非同步工作 已經完成

執行緒 Thread → 7

工作排程 Task scheduler → System.Threading.Tasks.ThreadPoolTaskScheduler

同步內文 SynchronizationContext →

Http內文 Http Context →

因為與這次請求所使用的執行緒不同

WPF await 接 續 封送 (Marshal) 處理測試

- 請分別針對這七個按鈕的執行結果，了解與分析
- 執行緒在前後的執行變化
- 輸入Log的輸出順序
- 與應用程式的執行結果

[1] 有await 非同步工作且會回到UI執行緒	[2] 有await 非同步工作且不會回到UI執行緒
[3] 沒有await 非同步工作且會回到UI執行緒	[4] 沒有await 非同步工作且不會回到UI執行緒
[5] 把非同步方法，當作同步方式呼叫(程式會凍結)	[6] 把非同步方法，當作同步方式呼叫(不會凍結)
	[7] 開始(UI執行緒忙碌中)



ConfigureAwait(true) 表示嘗試獲取當前的 SynchronizationContext，
將接續封送處理回原始擷取的内容



ConfigureAwait(false) 表示使用新執行緒，接續執行接下來的程式碼

- [1] 有await 非同步工作且會回到UI執行緒

```
await client.GetStringAsync("http://www.microsoft.com");
```

- [2] 有await 非同步工作且不會回到UI執行緒

```
await client.GetStringAsync("http://www.microsoft.com").ConfigureAwait(false);
```

- [3] 沒有await 非同步工作且會回到UI執行緒

```
await 非同步工作且會回到UI執行緒();  
非同步工作且會回到UI執行緒();
```

思考這兩行的差異

請解釋為什麼都會在 UI 執行緒下執行？

這裡使用沒有 await 的方式呼叫

- [4] 沒有await 非同步工作且不會回到UI執行緒

```
非同步工作且不會回到UI執行緒();
```

這兩行呼叫前後執行緒為何？

```
await client.GetStringAsync("http://www.microsoft.com")  
.ConfigureAwait(false);
```



- [5] 把非同步方法，當作同步方式呼叫(程式會凍結)

```
private void btn把非同步方法_當作同步方式呼叫  
_Click(object sender, RoutedEventArgs e)  
{ string result = 非同步工作且會回到UI執行緒().Result;}
```

為何螢幕會凍結

- [6] 把非同步方法，當作同步方式呼叫(不會凍結)

```
string result = 非同步工作且不會回到UI執行緒().Result;  
await client.GetStringAsync("http://www.microsoft.com")  
.ConfigureAwait(false);
```

為何螢幕不會凍結

- [7] 開始(UI執行緒忙碌中)

```
await 非同步工作且會回到UI執行緒();  
while (true) { Thread.Sleep(100); }
```


三種 Task 呼叫方式 使用 async await

在等候，
UI執行緒沒有被封鎖

```
private async void btn開始_Click(object sender, RoutedEventArgs e)
{
    Console.WriteLine("呼叫 btn開始_Click 前 : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
    string result = await DownloadContent();
    Console.WriteLine("呼叫 btn開始_Click 後 : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
}

public static async Task<string> DownloadContent()
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("呼叫 await 前 ) : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
        string 非同步工作結果1 = await client.GetStringAsync("http://www.microsoft.com");
        Console.WriteLine("呼叫 await 後 ) : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
    }
    return "";
}
```

3 4
因為呼叫了 await，所以，
會回到呼叫上層

三種 Task 呼叫方式 沒有使用 async await

遇到 await 就回到上一層，
DownloadContent()沒有
await，所以，會繼續執行

```
private void btn開始_Click(object sender, RoutedEventArgs e)
{
    Console.WriteLine("呼叫 btn開始_Click 前 : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
    DownloadContent();
    Console.WriteLine("呼叫 btn開始_Click 後 : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
}

public static async Task<string> DownloadContent()
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("呼叫 await 前 ) : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
        string 非同步工作結果1 = await client.GetStringAsync("http://www.microsoft.com");
        Console.WriteLine("呼叫 await 後 ) : Thread ID :{0}", Thread.CurrentThread.ManagedThreadId);
    }

    return "";
}
```

也就是，都是同步在處理

這個部分，經過編譯器處理，
都在有限狀態機內執行

三種 Task 呼叫方式

這個使用方式，發生了甚麼問題

因為同步程式執行，要等候 Result 取得後，才能繼續

1

```
private void btn會有問題_Click(object sender, RoutedEventArgs e)
{
    Console.WriteLine("呼叫 btn開始_Click 前 : Thread ID : {0}", Thread.CurrentThread.ManagedThreadId);
    string result = DownloadContent().Result;
    Console.WriteLine("呼叫 btn開始_Click 後 : Thread ID : {0}", Thread.CurrentThread.ManagedThreadId);
}

public static async Task<string> DownloadContent()
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("呼叫 await 前 ) : Thread ID : {0}", Thread.CurrentThread.ManagedThreadId);
        string 非同步工作結果1 = await client.GetStringAsync("http://www.microsoft.com");
        Console.WriteLine("呼叫 await 後 ) : Thread ID : {0}", Thread.CurrentThread.ManagedThreadId);
    }

    return "";
}
```

在此造成 UI 執行緒被封鎖了

當 DownloadContent() 非同步方法執行時候，會同步執行到 await，接著 client.GetStringAsync(“..”)會非同步在另外一個執行緒上執行。

2

3

這個需要同步在 UI 執行緒上執行

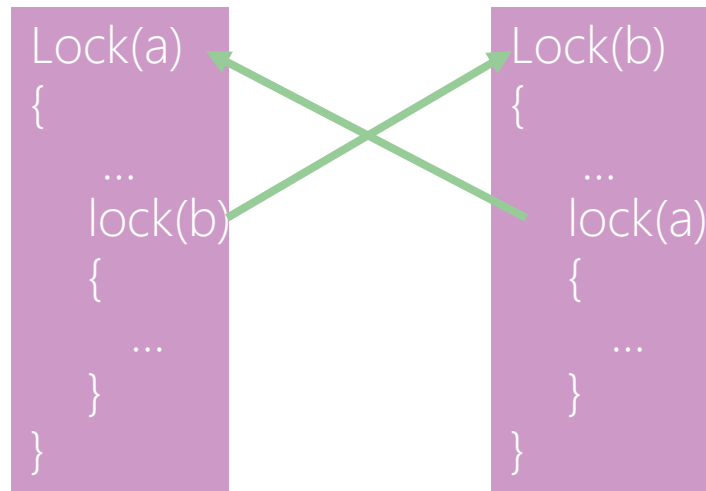
其執非同步工作行完成後，會再回到UI執行緒，不過，此時，UI執行緒，已經封鎖了(等候 Result)

不要混用同步與非同步方法

容易造成死結現象

什麼是 執行緒的死結

- 當執行緒都嘗試 封鎖 Block 另一個執行緒已經鎖定的資源時，就會發生死結。
- 結果就是：兩個執行緒都不能繼續進行。



不要混用同步與非同步方法

- 這是大部分人經常會遇到問題：**混用同步與非同步方法**
 - 接下來會有 GUI & ASP.NET 造成死結的範例
- 容易造成死結現象，底下程式碼會造成死結嗎？
- 解決死結問題的方法
 - 在非同步方法內，使用 **ConfigureAwait(false)**
 - 總是使用非同步呼叫**，避免封鎖 Block 的作法

```
private async void btnRunBlockForWait_Click(object sender, RoutedEventArgs e)
{
    var fooTask = GetRemoteResult1Async();
    var foo = fooTask.Result;
    tbkResult.Text = foo;
}
```



在這裡使用同步等待，造成**呼叫端執行緒**進入**封鎖狀態**，來取得非同步執行結果

```
public async Task<string> GetRemoteResult1Async()
{
    using (var client = new System.Net.Http.HttpClient())
    {
        var result = await client.GetStringAsync(
            "http://mocky.azurewebsites.net/api/delay/5000");
        return result;
    }
}
```



當 GetStringAsync **完成後**，將會透過 **SynchronizationContext** 繼續交由**呼叫端執行緒**來執行剩下程式碼，不過，因為**被封鎖**了，所以，**無法繼續執行**下去

練習情境 WPF 應用程式的造成死結範例

D001.GUI應用程式的造成死結範例

- 左邊按鈕將會造成程式無法反映，形同當掉
- 中間按鈕使用封鎖 Block 取得結果，只會短暫沒有反應
- 右邊按鈕全程使用非同步呼叫，執行相當順暢
- 下一頁有解說



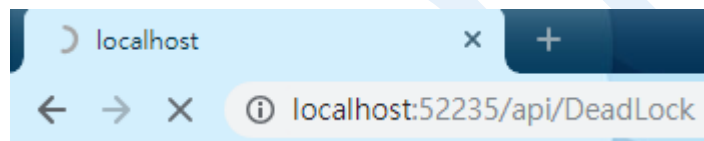
混用同步與非同步方法造成死結的過程說明

- 按鈕的事件中，呼叫 **GetRemoteResult1Async** 方法
- 在 **GetRemoteResult1Async** 方法內，使用 **HttpClient.GetStringAsync** 非同步呼叫遠端 **Web API**
- **GetStringAsync** 將會回傳一個尚未完成的工作給 **GetRemoteResult1Async**，並且將 **Context** 被捕捉下來，將於完成 **GetStringAsync** 工作後來使用
- 在按鈕事件中，使用 **GetRemoteResult1Async** 取得 **GetStringAsync** 回傳的未完成工作，並且想要透過屬性 **Result**，取得該工作的完成結果
 - 這將會造成這個**主執行 (UI) 緒被封鎖**
- 當 **GetStringAsync** 非同步作業完成後，準備要繼續執行下去
 - **GetStringAsync** 將會透過之前捕捉下來的 **Context**，在主執行緒下**繼續執行**下去 (回傳呼叫 **Web API** 的執行結果)
- 不過，因為主執行緒正在**處於封鎖狀態**，無法執行接下來繼續的程式碼，因此，造成了**死結**



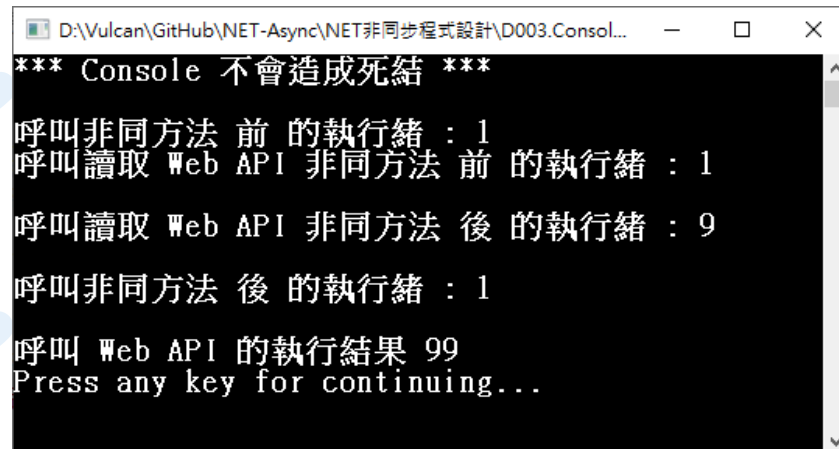
ASP.NET 應用程式的造成死結範例

- 請嘗試執行底下三個 URL，並了解哪個 URL 與為什麼會造成死結
 - <http://localhost:52235/api/NoDeadlock>
 - <http://localhost:52235/api/NoDeadlockByConfigureAwait>
 - <http://localhost:52235/api/Deadlock>



Console應用程式的不會造成死結範例

- 為什麼當在 Console 專案下，原先會造成打死結的程式碼，卻不會產生死結現象
- 因為當 Console 專案內，沒有 `SynchronizationContext`，因此，`GetStringAsync` 執行完畢後，會從執行緒集區取得新執行緒，接著往下執行



```
*** Console 不會造成死結 ***
呼叫非同方法 前 的執行緒 : 1
呼叫讀取 Web API 非同方法 前 的執行緒 : 1
呼叫讀取 Web API 非同方法 後 的執行緒 : 9
呼叫非同方法 後 的執行緒 : 1
呼叫 Web API 的執行結果 99
Press any key for continuing...
```

```
sb.Clear();
sb.Append($"*** Console 不會造成死結 ***{Environment.NewLine}{Environment.NewLine}");
sb.Append($"呼叫非同方法 前 的執行緒 : {Thread.CurrentThread.ManagedThreadId}{Environment.NewLine}");

var fooClientTask = GetRemoteResult1Async();
var foo = fooClientTask.Result;

sb.Append($"呼叫非同方法 後 的執行緒 : {Thread.CurrentThread.ManagedThreadId}{Environment.NewLine}{Environment.NewLine}");
sb.Append($"呼叫 Web API 的執行結果 {foo}");

Console.WriteLine(sb.ToString());

Console.WriteLine("Press any key for continuing...");
Console.ReadKey();
```



因為 `SynchronizationContext`
預設為執行緒集區



等候 await 的非同步工作用法

- `var result = await MethodAsync()`
 - 等候 `MethodAsync` 非同步方法執行完成並取得回傳值
- `var task = MethodAsync()`
 - 呼叫 `MethodAsync` 非同步方法，並取得工作物件
- `await MethodAsync()`
 - 等候 `MethodAsync` 非同步方法執行完成
- `await MethodAsync().ConfigureAwait(false)`
 - 等候 `MethodAsync` 執行完成後，使用新執行緒繼續工作
- `await Task.WhenAll()` / `await Task.WhenAny()`
 - 等候所有或任意工作完成
- `MethodAsync().ContinueWith()`
 - `MethodAsync` 執行完成後，使用回呼 callback 委派繼續



還有個使用 `async / await` 必須知道的
知識，那就是

射後不理 (Fire and Forget)

會造成無法捕捉到
例外異常與執行結果

正常情況，不要使用 `async void MethodAsync(){}`

什麼是 工作 的 Fire and Forget

- 呼叫一個方法，但是不需要關心何時結束與等候結果
- 使用 Thread 發生例外異常，應用程式會當掉 (Crash)
- 使用 Task 發生例外異常，你會不知道發生甚麼事情！
- 射後不理實作練習，了解各種用法的差異
 - 使用執行緒
 - `async Task` 方法
 - `async void` 方法



這裡若有例外異常，應用程式會崩潰嗎？



這裡若有例外異常，應用程式會崩潰嗎？



對於非同步工作 Fire and Forget 進行了解度測驗

- 請問底下程式碼會造成應用程式崩潰或者顯示出甚麼內容

```
static async Task Main(string[] args)
{
    var foo = FireException();
    Console.WriteLine("Press any key for continuing...");
    Console.ReadKey();
    if (foo.Exception?.InnerExceptions.Count > 0)
    {
        foreach (var item in foo.Exception.InnerExceptions)
        {
            Console.WriteLine($"發現到例外異常 {item.ToString()}");
        }
    }
    Console.WriteLine("Press any key for continuing...");
    Console.ReadKey();
}
```

這裡使用 try...catch 可以捕捉到例外異常嗎？



若把 Task 換成 void 又會如何？
你能解釋為什麼會造成這樣結果？

```
static public async Task FireException()
{
    Console.WriteLine("非同步工作休息5秒鐘");
    await Task.Delay(5000);
    Console.WriteLine("產生一個例外異常");
    throw new Exception("強制產生一個例外異常");
    return;
}
```



執行後五秒鐘 "內" 按下任一按鍵，
會出現甚麼結果



執行後五秒鐘 "後" 按下任一按鍵，
會出現甚麼結果



為何應用程式不會當掉呢？

- 設定方案管理員要 顯示所有檔案
- 將 要測試類型.cs 檔案加入專案，並排除其他 .cs
- 射後不理使用執行緒
 - 得到專案崩壞異常：System.Exception: '有例外異常發生了'

```
private static void OnDelegateWithException(object state)
{
    Thread.Sleep(5000);
    throw new Exception("有例外異常發生了");
}
```

在 Main 方法內，
使用執行緒來執行
這個委派方法

- 射後不理 async_void
 - 得到專案崩壞異常：System.Exception: '有例外異常發生了'

```
public static async void OnDelegateWithExceptionAsync()
{
    Console.WriteLine("OnDelegateWithExceptionAsync 開始執行了");
    await Task.Delay(5000);
    throw new Exception("有例外異常發生了");
}
```

在 Main 方法內，使用
OnDelegateWithExceptionAsync() 來執行這個
非同步方法(不等候)

- 射後不理 `async Task`
 - 專案正常結束，可是，對於發生異常的部分，為什麼沒有任何反應呢？

```
static async Task OnDelegateWithExceptionAsync()  
{  
    Console.WriteLine("OnDelegateWithExceptionAsync 開始執行了");  
    await Task.Delay(5000);  
    throw new Exception("有例外異常發生了");  
}
```

在 Main 方法內，使用
`var fooTask = OnDelegateWithExceptionAsync();`
來執行這個非同步方法。

→ 不過，為什麼會回傳 `Task` 物件呢？

→ 前一個 射後不理 `async void` 例子，為什麼沒回傳值？

多個工作的等待應用

Task.WaitAll、Task.WaitAny、
Task.WhenAll、Task.WhenAny

執行緒會進入 封鎖 Block，等待所有提供的 Task 物件完成執行

- 我們先啟動了**3個**非同步工作，分別需要花費 **1, 2, 3 秒鐘**的時間
- 接下來會等待 (使用 **Task.WaitAll**)，直到**所有工作都執行完成**，並且將工作的執行結果輸出到Console上
- 請問，執行三個非同步工作，**總共需要多少**時間才能完成？
- Task.**WaitAll**(tasks);
 - 此處會造成現在執行緒被**鎖定(Block)**，直到所有的工作都完成(也許是失敗、取消)

執行緒會進入 封鎖 Block，等候任一提供的 Task 物件完成執行

- 這個範例展示了：等待任一提供的 Task 物件完成執行，而不需等到所有工作都完成才需要繼續接下來的工作
- 我們先啟動了3個非同步工作，分別需要花費 1, 2, 3 秒鐘的時間
- 接下來會等待 (使用 **Task.WaitAll**)，直到任一工作執行完成，並且將工作的執行結果輸出到Console上
- `int i = Task.WaitAny(tasks);`
 - 此處會造成現在執行緒被**鎖定(Block)**，直到任一工作都完成(也許是失敗、取消)

使用 `await` 關鍵字，等候所有 Task 物件完成執行

- 我們先啟動了**3個**非同步工作，分別需要花費 **1, 2, 3 秒鐘**的時間
- 接下來會等待 (使用 `Task.WhenAll`)，直到**所有工作**都執行**完成**，並且將工作的執行結果輸出到Console上
- 請問，執行三個非同步工作，**總共需要多少**時間才能完成？
- `await Task.`**WhenAll**(tasks);
 - 此處會等候所有的工作完成
 - 請注意輸出內容的 **執行緒ID**

使用 `await` 關鍵字，等候任一 `Task` 物件完成執行

- 我們先啟動了**3個**非同步工作，分別需要花費 **1, 2, 3 秒鐘**的時間
- 接下來會等待 (使用 `Task.WhenAny`)，直到**所有工作都執行完成**，並且將工作的執行結果輸出到Console上
- 請問，執行三個非同步工作，**總共需要多少時間**才能完成？
- `await Task.`**WhenAny**`(tasks);`
 - 此處會等候任一的工作完成
 - 請注意輸出內容的 **執行緒ID**

使用 Task.WhenAll 來同時等待三個非同步工作的完成

- 目的
 - 使用多個非同步工作，呼叫多個 Web API，取得兩個整數相加值，最後顯示所有回傳結果的總和
- 提示
 - 請使用 HttpClient 分別呼叫這三個 URL (不同時間完成)
 - <http://mocky.azurewebsites.net/api/delay/1000>
 - <http://mocky.azurewebsites.net/api/delay/2000>
 - <http://mocky.azurewebsites.net/api/delay/3000>
 - 使用 Task.WhenAll 來同時等待這三個工作的完成
 - 預計在7秒鐘後，會顯示：計算總合為 458

接續工作 (Continue Tasks)

- 將資料從前項傳遞至接續
- 精確指定是否要叫用**接續的條件**
- 在接續啟動之前加以取消，或在它執行時以合作方式加以取消
- 提供有關如何排程接續的提示
- 從相同的前項叫用多個接續
- 在多個前項全部或有任何一個完成時，叫用一個接續
- 將接續逐一鏈結成任意長度
- 使用接續處理由前項所擲回的例外狀況



接續工作 (Continue Tasks) 撰寫風格

TPL

```
var t = Task.Run(() =>
{
    return 42;
})
.ContinueWith((t1) =>
{
    return t1.Result * 2;
});

t.Wait();
Console.WriteLine(t.Result);
```

await 關鍵字

```
var t = Task.Run(() =>
{
    return 42;
});

int result = await t;

result = result * 2;

Console.WriteLine(foo最後結果);
```



使用 ContinueWith 方法，來接續不同工作狀態並執行相關處理

- 切換 工作的接續動作Enum `fooTaskAction` 這個變數值
 - 體驗一個非同步工作正常結束、有例外異常、取消的時候，接下來該如何繼續執行下去
- 在非同步工作中，發出取消請求，可以使用
 - `CancellationTokenSource.Cancel()`
 - 在非同步工作內，需要執行底下方法
`CancellationToken.ThrowIfCancellationRequested`
- 在非同步工作中，使用底下敘述拋出例外異常
 - `throw new Exception("唉，真糟糕呀，真糟糕");`

使用 ContinueWith 與 await 的差異在哪裡

- **ContinueWith** 就像宣告 回呼 **callback** 委派方法
 - 當工作完成、有例外異常、取消會被呼叫
- **await** 是等候該工作完成
 - 工作**完成之後**，會繼續執行剩下來的程式碼
 - 當有取消、例外異常，會造成應用程式崩壞
- 兩者**都不會**造成呼叫端的**執行緒封鎖 Block**
- 兩者實際**運作方式**有些不太相同



Task 非同步方法的例外處理



執行緒例外異常

工作例外異常

await 例外異常

等待工作例外異常



關於 TAP 非同步樣式的例外處理原則

- 非同步方法應只有在回應**使用方式錯誤**時，才會從非同步方法拋出例外。
- **使用方式錯誤**一律不應發生在實際執行的程式碼中。
 - 您可以修改呼叫程式碼，確保絕不會傳遞 null 參考
- 對於所有其他錯誤，非同步方法執行時發生的例外狀況應該**指派給傳回的工作**，即使非同步方法剛好在工作傳回前同步完成也一樣。
- 通常，工作最多只能包含一個例外狀況。
 - 不過，如果工作表示多項作業 (例如 WhenAll)，則可能會有多個例外狀況與單一工作相關聯。
- [以工作為基礎的非同步模式 \(TAP\) | Microsoft Docs](#)



Thread 與 Task 例外狀況處理比較

- Thread (執行緒)
 - 無法自動捕捉例外異常
 - 開發者需要在執行緒內的程式碼手動進行捕捉例外 (請參考 **D030.非同步與例外異常** 的範例)
- Task (工作)
 - 將會自動捕捉例外異常
 - 例外異常將會儲存在 Task 物件內
 - 當等待**多個工作**完成時，可能會擲回多個例外狀況，此例外狀況會包裝在一個 `AggregateException` 執行個體，`AggregateException` 具有 `InnerExceptions` 屬性可被列舉來檢查所有擲回的原始例外狀況，並且個別處理 (或不處理) 每個例外狀況。



```

private void MoveNext()
{
    int num = <>1__state;
    string result;
    try
    {
        TaskAwaiter<string> awaiter;
        if (num != 0)
        {
            <myString>5__1 = "Begin";
            awaiter = new HttpClient().GetStringAsync("https://lobworkshop.azurewebsites.net/api/RemoteSource/Add/8/9/2").GetAwaiter();
            if (!awaiter.IsCompleted)
            {
                num = (<>1__state = 0);
                <>u__1 = awaiter;
                <MyMethodAsync>d__1 stateMachine = this;
                <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
                return;
            }
        }
        else
        {
            awaiter = <>u__1;
            <>u__1 = default(TaskAwaiter<string>);
            num = (<>1__state = -1);
        }
        <>s__3 = awaiter.GetResult();
        <result>5__2 = <>s__3;
        <>s__3 = null;
        <myString>5__1 = "Complete" + myInt + " " + <result>5__2;
        result = <myString>5__1;
    }
    catch (Exception exception)
    {
        <>1__state = -2;
        <>t__builder.SetException(exception);
        return;
    }
    <>1__state = -2;
    <>t__builder.SetResult(result);
}

```

TAP 非同步方法的例外異常發生狀況之歸納整理

- Thread
 - 執行緒可以拋出例外，但是無法捕捉到，因此會造成應用程式關閉！
- Task
 - 沒有透過「等待」的 Task 當例外發生時，並不會向外拋出例外！
 - 直接回傳 Task 的呼叫方式，等同於「沒有等待」的非同步呼叫，自然也無法透過 try/catch 捕捉到例外！
 - 有透過「等待」機制處理的 Task 才有機會透過 try/catch 捕捉例外！
 - `await someTask` 取得 Task 拋出的例外
 - `await Task.WhenAll` 取得第一個 Task 拋出的例外
 - `await someTask (取消請求)` `OperationCanceledException`
 - `await Task.WhenAny` 不會拋出例外 (回傳最近完成的工作) 
 - `Task.WaitAny` 不會拋出例外 (回傳最近完成工作索引)
 - `Task.WaitAll` `AggregateException`
 - `Task.Wait()` `AggregateException`
 - `Task.Wait() (取消請求)` `AggregateException`
 - `Task.Wait(millisecondsTimeout)` (未發生逾期但有例外異常) `AggregateException`
 - `Task.Wait(millisecondsTimeout)` (發生逾期但沒有例外拋出) 不會拋出例外 (回傳是否逾期)
 - [Exception handling \(Task Parallel Library\) | Microsoft Docs](#)

- 範例專案同時執行 9 個非同步工作時，使用 **Task.WhenAll** 來等候所有非同步工作完成
其中ID為 3, 6, 9 這三個工作，將會拋出例外異常
 - 是否只要有其中一個工作發生異常，就會立即結束 **Task.WhenAll** 的執行呢？
 - 如何捕 Task 執行過程中拋出的例外？
 - 你是否可以確認捕捉到的例外異常物件型別是什麼？
- 若執行非同步工作發生異常，必須判斷以下屬性：
 - **Task.IsFaulted**
- 若改成使用 **Task.WaitAll** 來等待所有非同步工作完成
 - 此時捕捉到的例外異常物件型別是什麼？

了解各種非同步作業中對於例外異常會產生何種情況

- await工作例外異常
- await工作取消例外異常
- 工作例外異常
- 工作取消例外異常
- 工作尚未啟動就發生例外異常
- 同步程式碼例外異常
- 執行緒的例外異常
- 工作WaitAll 例外異常
- 工作WaitAny 例外異常
- 工作Wait 例外異常
- 工作Wait 取消例外異常
- 工作Wait 逾期沒有例外異常
- 工作WhenAny 例外異常
- 工作WhentAll 例外異常

Task 取消的設計方法



取消權杖來源

呼叫端建立取消權杖來源物件，將 Token 屬性為引數

- CancellationTokenSource
 - 向 CancellationToken 發出訊號，表示應該將它取消
 - 使用 CancellationTokenSource.Cancel() 方法
 - 透過 CancellationTokenSource.Token 取得權杖
 - 權杖可用於多個非同步工作，也可以合併不同權杖成為一個新的取消權杖來源

CancellationTokenSource

Cancel

CancelAfter

CreateLinkedTokenSource

Token

IsCancellationRequested

取消權杖



被呼叫端使用取消權杖檢查是否有發出取消請求

- CancellationToken
 - 在非同步工作內透過方法參數取得
 - 透過該取消權杖的屬性檢查是否有需求要求發出
 - 透過 `CancellationTokenSource.Token` 取得
 - 散佈通知，表示作業應被取消

`CancellationToken`

`CanBeCanceled`

`IsCancellationRequested`

`CancellationToken.None`

`Register`

`ThrowIfCancellationRequested`



無法取消的取消語彙基元傳回此屬性;也就是說，其`CanBeCanceled`屬性是`false`



您也可以使用 `C# default(CancellationToken)` 陳述式來建立空的取消語彙基元

使用 CancellationTokenSource 來取消非同步工作的做法指引

- 在**呼叫**非同步工作端，建立 **CancellationTokenSource** 物件，使用 **CancellationTokenSource.Token** 這個屬性取得 **CancellationToken**
- 將 **CancellationToken** 傳入到非同步工作內
- 在非同步工作**方法**內，使用**輪詢 Polling** 方式檢查是否收到取消非同步工作通知
 - **CancellationToken.IsCancellationRequested**
 - **CancellationToken.ThrowIfCancellationRequested**
- 若收到取消通知，可以選擇在非同步工作內
 - 結束非同步工作執行
 - 拋出 **OperationCanceledException** 例外異常
- **Token 一旦取消之後，就無法再度被使用(要重新建立)**



CancellationTokenSource 與 CancellationToken 的設計練習

- 這個專案將會每 0.1 秒，使用輪詢 **Polling** 方式檢查是否有進入到取消狀態
 - 使用者可以按下 **Ctrl-C** 按鍵，就可以進入到取消狀態
- 若使用這個**屬性**來檢查是否有進入到取消狀態
CancellationToken.IsCancellationRequested
 - 可以結束該非同步執行程序，**清除**相關使用到的**資源**，並且**不會發出任何例外異常**
- 在每個輪詢點，執行
CancellationToken.ThrowIfCancellationRequested 方法
 - 若沒有進入到取消狀態，不會有任何動作
 - 若有進入到取消狀態，則會拋出
OperationCanceledException 例外異常

使用 CancellationSource 取消 HttpClient 網路存取練習

- 這個專案將會存取底下網址，並且於 10 秒後得到計算結果
 - `http://mocky.azurewebsites.net/api/delay/10000`
- 若在 **10 秒鐘**內按下 c 按鍵，則會取消此非同步網頁存取
- 當 HttpClient 物件收到取消訊號，便會拋出 **OperationCanceledException** 例外異常
- 記得要在 **await** 關鍵字使用 **try...catch** 捕捉例外異常

使用 CreateLinkedTokenSource 來組合多個 CancellationToken

- CreateLinkedTokenSource 將會組合多個 CancellationToken，建立新的 CancellationTokenSource 物件
- 任何一個 取消權杖來源 發出**取消要求**，將會進入到**取消狀態**



使用 CreateLinkedTokenSource 來組合兩個 CancellationToken

- 這個專案將會存取底下網址，並且於 **10 秒**後得到計算結果
 - <http://mocky.azurewebsites.net/api/delay/10000>
- 將會透過組合 **使用者要求** 取消與 **8秒** 內自動取消兩個 CancellationTokenSource
- CancellationTokenSource.CancelAfter(8000) 方法，將會設定在 8 秒鐘之後，自動進入到取消狀態
- 使用 CancellationTokenSource.IsCancellationRequested 可以判斷是否進入取消狀態

取得 Task 的執行進度



工作進度回報

- 在非同步工作內，接收參數 `IProgress<T>`
 - 定義進度更新的提供者 (可選擇 `Progress<T>` 實作)

`Progress<T>`

`OnReport(T)`

`ProgressChanged` 事件



`Progress<T>` 的明確介面 `IProgress<T>` 方法實作 `Report(T)`



`IProgress<T>` 使用 `Report` 回報，`Progress<T>` 使用 `OnReport` 回報

使用 `Progress<T>` 來設計非同步工作進度回報的做法指引

- 在呼叫非同步工作端，建立 `Progress<T>` 物件
- 在呼叫非同步工作端，訂閱 `Progress<T>.ProgressChanged` 事件，取得非同步工作中傳來的處理進度
- 在非同步工作方法內，提供 `IProgress<T>` 型別參數
- 在非同步工作方法內，透過 `IProgress<T>.Report` 方法，送出處理進度的通知



對於所用非同步工作有提供 `Iprogress<T>` 覆載，可以直接使用 `Progress<T>`

在 WPF 專案用 HttpClient 從網路下載大檔案，並可以更新下載進度

- 當執行 WPF 專案後，按下開始按鈕後，便會開始下載；下載的時候，進度狀態棒會根據下載數量更新完成百分比，當然，也可以按下取消按鈕，已取消這個非同步工作
- 在 HttpClient 內，使用
 - GetAsync 方法讀取 URL，但是使用 `HttpCompletionOption.ResponseHeadersRead` 引數值
 - `HttpResponseMessage.Content.Headers.ContentLength` 取得此次下載檔案的大小
- 了解處理進度 Progress 與取消權杖來源 CancellationTokenSource 的用法
 - 使用 `Progress<T>` 類別建立一個物件
 - 建構式要傳入一個委派方法 `Action<T>`，用來顯示處理進度
 - 將這個物件傳入到非同步方法內，使用這個參數型別 `IProgress<T>`
 - 在非同步方法內，使用 `IProgress<T>.Report` 回報進度

在 Console 專案用 HttpClient 從網路下載大檔案，並可以更新下載進度

- 當專案啟動後，便會開始下載檔案；下載的時候，將會回報下載進度完成百分比，當然，也可以按下 c 按鍵，以便取消這個非同步工作
- 在 HttpClient 內，使用
 - GetAsync 方法讀取 URL，但是使用 `HttpCompletionOption.ResponseHeadersRead` 引數值
 - `HttpResponseMessage.Content.Headers.ContentLength` 取得此次下載檔案的大小
- 了解處理進度 Progress 與取消權杖來源 CancellationTokentokenSource 的用法
 - 使用 `Progress<T>` 類別建立一個物件
 - 建構式要傳入一個委派方法 `Action<T>`，用來顯示處理進度
 - 將這個物件傳入到非同步方法內，使用這個參數型別 `IProgress<T>`
 - 在非同步方法內，使用 `IProgress<T>.Report` 回報進度

設計 Task 的方法



建立工作模式的方式

- 使用 `Task.Run` 或 `Task.TaskFactory.StartNew`
- 建立一個在新的非同步方法內**呼叫現有非同步方法**
 - 方法回傳值須為 `Task` 與 `Task<T>`
 - `Task` → 沒有需要回傳值
 - `Task<T>` → 需要有回傳值
- 使用 `TaskCompletionSource<T>` 來建立



使用 TaskCompletionSource 建立非同步工作

- 方法需要回傳 `T`
 - 建立 `TaskCompletionSource<T>` 物件
 - 回傳 `TaskCompletionSource.Task`
- 對於非同步工作部分，使用底下方法來決定工作狀態
 - `TaskCompletionSource<T>.SetResult(T)`
 - 用來設定非同步工作的回傳值
 - `TaskCompletionSource<T>.SetCanceled()`
 - 用來設定非同步工作接受到取消通知
 - `TaskCompletionSource<T>.SetException()`
 - 用來設定非同步工作有例外事件產生



CPU Bound 的非同步工作的設計方法

- 使用 TaskFactory.StartNew 或 Task.Run 來設計
- 使用 Task.ContinueWith 方法來繼續相關工作

```
public Task<int> ComputingAsync()  
{  
    return Task.Factory.StartNew(() =>  
    {  
        Thread.Sleep(1000);  
        return 15;  
    }));  
}
```



如何將 APM/EAP 轉換成為 TAP 方法

請參考後兩頁簡報的說明

I/O Bound 的非同步工作的設計方法

底下範例為將 EAP 改成 TAP 設計模式

- 使用 TaskCompletionSource
- 不需依賴 Thread Pool內的執行緒

```
public Task<string> DownloadStringAsync(Uri url,
                                       CancellationToken token)
{
    var tcs = new TaskCompletionSource<string>();
    var wc = new WebClient();
    wc.DownloadStringCompleted += (s, e) =>
    {
        if (e.Error != null) tcs.TrySetException(e.Error);
        else if (e.Cancelled) tcs.TrySetCanceled();
        else tcs.TrySetResult(e.Result);
    };
    token.Register(() => wc.CancelAsync());
    wc.DownloadStringAsync(url);
    return tcs.Task;
}
```



使用 TaskCompletionSource 類別，可以輕鬆地將 EAP 非同步設計改成 TAP 非同步設計

- 使用 `WebRequest.Create` 建立一個採用 APM 的物件
- 使用 `Task.Factory.FromAsync` 工廠方法，把 APM 的 `Begin / End` 方法，包裝成為一個 `Task` 物件
- 使用這個新建立的 `Task` 物件，進行非同步呼叫

```
WebRequest myHttpRequest1 = WebRequest.Create(url);
```

```
WebResponse webResponse = await Task.Factory.FromAsync(  
    myHttpRequest1.BeginGetResponse,  
    myHttpRequest1.EndGetResponse, null);
```

```
using (StreamReader reader =  
    new StreamReader(webResponse.GetResponseStream()))  
{  
    string result = await reader.ReadToEndAsync();  
    Console.WriteLine($"網頁執行結果:{result}");  
}
```

Task 偵錯方法



偵錯平行應用程式 檢視單一執行緒的呼叫堆疊

- [偵錯] [視窗] [執行緒]。
- [偵錯] [視窗] [呼叫堆疊]。
- 按兩下 [執行緒] 視窗中的執行緒，使它成為目前執行緒。
 - 目前執行緒具有黃色箭號。
 - 當您變更目前執行緒時，它的呼叫堆疊會出現在 [呼叫堆疊] 視窗中。



偵錯平行應用程式

檢視單一執行緒的呼叫堆疊

// 我們僅將最外圍的迴圈做平行計算，因為，裡面的兩個迴圈計算工作不多，若也採用平行計算方式，則會有額外的負擔產生

```
Parallel.For(0, matARows, i =>
{
    if (i == 100)
    {
        int a = 0;
    }
    for (int j = 0; j < matBCols; j++)
    {
        double temp = 0;
        for (int k = 0; k < matACols; k++)
        {
            temp += matA[i, k] * matB[k, j];
        }
        result[i, j] = temp;
    }
}); // Parallel.For
```

執行緒

ID	Managed ID	分類	名稱	位置
8012	10	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
17336	11	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
10800	12	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
17128	13	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
8660	14	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
16292	15	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
15980	16	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!
16876	17	背景工作執行緒	背景工作執行緒	計算兩個矩陣的!

呼叫堆疊

名稱	語言
計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計算 C#	C#
[外部程式碼]	

輸出 尋找結果 1 尋找符號結果 區域變數 監看到 1 執行緒 工作 呼叫堆疊 例外狀況設定 即時運算視窗

偵錯平行應用程式 檢查平行堆疊視窗

- [偵錯] [視窗] [平行堆疊]。
- 確定左上角的方塊中已選取 [執行緒]。
- 透過使用 [平行堆疊] 視窗，您可以在一個檢視中同時檢視多個呼叫堆疊。

	執行緒	堆疊框架
▼ ➡	8612 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	8660 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	10800 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	15980 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	16292 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	16876 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	17128 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計
▼	17336 (背景工作執行緒)	計算兩個矩陣的乘積.exe!計算兩個矩陣的乘積.Program.矩陣相乘之非同步平行計

Sleep & Delay 到底有何不同

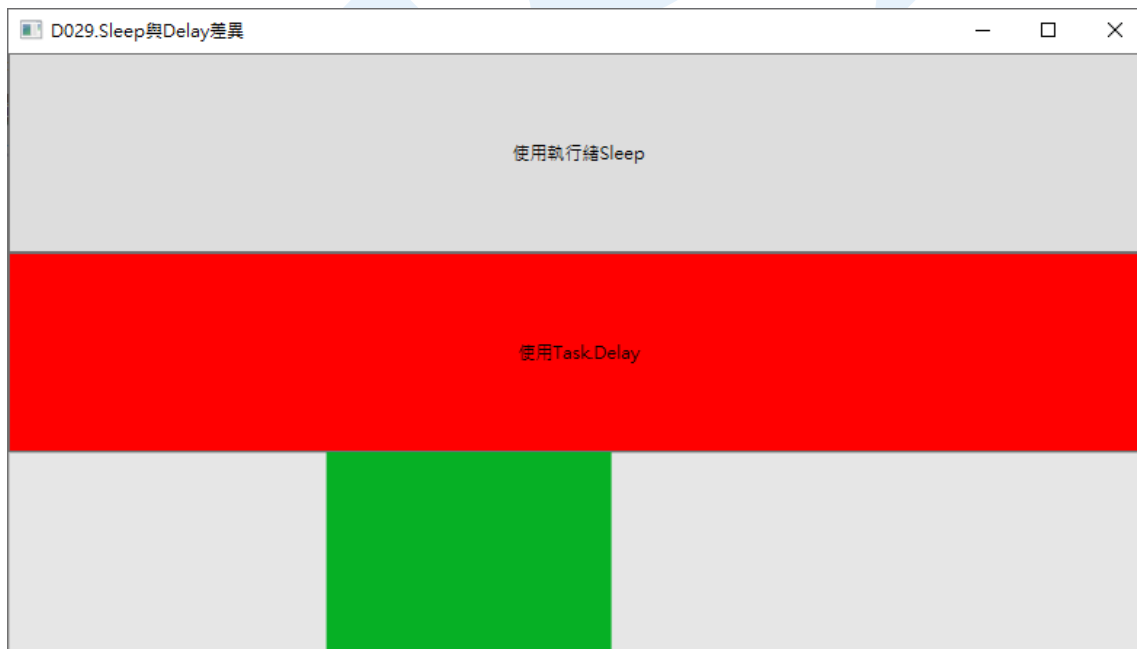
```
void MyMethod()  
{  
    // 在同步方法中，暫時停止1000毫秒  
    Thread.Sleep(1000);  
}
```

```
async Task MyMethodAsync()  
{  
    // 使用非同步的方式來暫停等待1000毫秒  
    await Task.Delay(1000);  
}
```



透過 GUI 應用程式，如 WPF，實際體驗 Sleep 與 Delay 的差異

- **Thread.Sleep** 會造成當時執行緒被封鎖(例如，UI執行緒無法執行任何指令，造成螢幕 **UI 控制項** 無法正常更新)
- **Task.Delay** 因為使用非同步等候方式，所以，當時的執行緒是不會被封鎖的，螢幕 **UI 控制項** 可以正常更新



非同步程式設計最佳實務



非同步程式設計中的最佳做法

執行以下操作...	替換以下方式...	使用這個
檢索背景工作的結果	<code>Task.Wait</code> or <code>Task.Result</code>	<code>await</code>
等待任何工作完成	<code>Task.WaitAny</code>	<code>await Task.WhenAny</code>
檢索多個工作的結果	<code>Task.WaitAll</code>	<code>await Task.WhenAll</code>
等待一段時間	<code>Thread.Sleep</code>	<code>await Task.Delay</code>



常見非同步問題的解決方案

問題	解決方案
建立工作以執行程式碼	<code>Task.Run</code> 或 <code>TaskFactory.StartNew</code> (不是 <code>Task</code> 建構函式或 <code>Task.Start</code>)
為操作或事件建立工作	<code>TaskFactory.FromAsync</code> <code>TaskCompletionSource<T></code>
支援取消	<code>CancellationTokenSource</code> <code>CancellationToken</code>
報告進度	<code>IProgress<T></code> 和 <code>Progress<T></code>



非同步程式設計不建議這麼做

- 使用 Thread 類別來自行實作非同步方法
- 在非同步程式碼中，執行同步程式呼叫 (常見錯誤) 
- 在方法前加上 async 就完成非同步程式設計 (常見錯誤)
- 非事件委派方法，使用 async void
- 使用 await 關鍵字，不使用 try...catch
- 使用射後不理 (Fire and forget) 來呼叫非同步方法
- 使用 Task.Wait 來等候工作結果
- 使用執行緒局部儲存 [ThreadStatic] 

支援非同步程式設計的方法

應用類型	支援非同步方法的類別
Web 存取	HttpClient 、 SyndicationClient
處理檔案	StorageFile 、 StreamWriter 、 StreamReader 、 XmlReader Stream 、 TextReader
處理影像	MediaCapture 、 BitmapEncoder 、 BitmapDecoder
資料庫存取	SqlClient 、 SqlCommand
Entity Framework	DbContext

從 APM/EAP 轉換為 TAP 非同步工作

- 可以使用底下方式進行轉換，便可以使用非同步方法來呼叫
 - APM模式
 - [Task.Factory.FromAsync](#)
 - EAP模式
 - [TaskCompletionSource](#)
 - 同步程式碼 (不做任何轉換)
 - [Task.CompletedTask](#) (沒有回傳值的 Task 回應)
 - [Task.FromResult](#) (含有回傳值的 Task 回應)



async / await 常見問題解答



async 相關疑問

- 為什麼需要編譯器幫助非同步程式設計
 - 任您寫非同步程式碼，使用同步方式來寫
 - 由**編譯器幫助**您產生非同步程式設計需要的程式碼
 - 處理必要轉換與避免執行緒封鎖
- 把同步方法放到 `Task.Run` 內，變成非同步
 - 改善 UI 執行緒回應 → Yes
 - 提高可擴展性 → 沒有任何實際意義
- 方法前加入 `async` 做了甚麼事情
 - 告訴編譯器，想在方法內部使用 `await` 關鍵字
 - 告訴編譯器，方法執行完的結果或可能發生的異常都將作為返回類型返回



async 相關疑問

- 方法前加入 `async`，就變成非同步方法嗎
 - 不會
 - 標記`async`的方法，沒有使用`await`，那麼該方法將以同步方式執行
- `async` 關鍵字導致新創建一個新執行緒嗎？
 - 不會
 - 方法就可以在`await` 處暫停並且在`await` 標記的工作完成後非同步喚醒
- `async` 關鍵字能標記任何方法**回傳型別**嗎
 - 不行，只有 `void, Task, Task<TResult>`



async & await 相關疑問

- async方法建立的工作，需要呼叫Start()嗎
 - 不用，TAP 方法取得工作，已經正在執行
- async方法建立的工作，需要Dispose()嗎
 - 不用
- await 是如何關聯到當前 執行緒的同步處理內容 (SynchronizationContext)
 - 正在等待的工作的默認行為是暫停前捕獲當前的 SynchronizationContext
 - 等工作完成，使用原先SynchronizationContext 的 Post方法，來執行完成的委派方法。



await 相關疑問

- await 關鍵字做了什麼
 - 告訴編譯器在 async 標記的方法中插入一個可能的暫停/恢復(suspension/resumption)點
- 哪些地方不能使用 await
 - 未標記async的方法或lambda 表達式中
 - 在屬性的get 存取子 和 set 存取子裡
 - 在lock/SyncLock 塊中
 - 在unsafe 區域中
 - 在catch 塊和finally 塊中
 - LINQ 中大部分查詢語法中



await 相關疑問

- await task 和 task.Wait 效果一樣嗎
 - task.Wait() 是一個同步呼叫，會封鎖執行緒
 - await task告訴編譯器在 async 標記的方法中插入一個可能的暫停/恢復點
 - 如果等待的task 沒有完成，非同步方法也會立馬返回到呼叫者
 - 當等待的工作完成時，喚醒它從恢復點處繼續執行



關於 Will 保哥

- Will 保哥的技術交流中心

<http://www.facebook.com/will.fans>

- Will 保哥的推特

https://twitter.com/Will_Huang

- The Will Will Web

記載著 Will 在網路世界的學習心得與技術分享

<http://blog.miniasp.com/>

- 台灣 .NET 技術愛好者俱樂部

<https://www.facebook.com/groups/DotNetUserGroupTaiwan/>

