kinanson的技術回憶

學習新技術是來幫忙解決問題，而不是來製造更多問題的

2017-05-03

[C#]TPL與Async Await的觀念還有使用方式

[C#]TPL與Async Await的使用方式和一些觀念

前言

其實TPL和Async Await都是比較新而且有點關聯性的東西，所以筆者想要把這兩個放在一起討論和解釋，到底有何不同，還有該怎麼使用，如何正確使用還有該是什麼狀況下要用哪一種，其實不管是執行緒或者TPL或Parallerl或async await都是為了要讓C#原本同步的執行方式，變成能多工執行的方式，在javascript就統稱都是非同步，在C#只把async await稱為非同步，不過雖然要做的事情都一樣，對機器底層的運作機制卻不太一樣，導致一樣要做多工的方式，你必須要取決於什麼狀況要如何處理，而不是一昧的用async await或執行緒，如果對執行緒有興趣可以參考(https://dotblogs.com.tw/kinanson/2017/04/25/083020)，如果對並行的方式有興趣可以參考(https://dotblogs.com.tw/kinanson/2017/04/28/100600)

導覽

TPL和ASYNC AWAIT的差異

其實以官網的說法，async和await只是一種模式，但是只要是官方制訂的api結尾有Async的，比如HttpClient的PostAsync方法，或者是lambda的extension比如FirstOrDefaultAsync()，都是不會額外使用到執行緒的也不會佔用CPU，而是使用I/O Bound的方式，但是我們自己使用TPL或Parallel都是以執行緒集區為基礎的，所以.net CLR會自動的幫你管理執行緒的狀況，而如果我們使用Thread則是自行需要管理，所以官方建議我們都使用TPL的方式來取代Thread的使用，但是Async Await其實也可以方便的搭配TPL使用，不過在此說明一下，如果我們自行使用TPL在包裝某個共用方式的時候，在方法的命名使用上不要使用Async為結尾的方式，以免讓人誤解以為我們的方法是不佔用執行緒的，畢竟比較適合IO的方式，大部份微軟也都幫我們做好了，甚至當我們使用dapper或stackexchange的時候，這些pakcage也都提供了async的api給我們使用，我們只要有正確的觀念，使用正確的方式使用就夠了，還記得嗎，執行緒集區總是使用背景執行緒哦，這個概念請必須謹記，如果我們需要前景執行緒的話，我們就必須得自行操作thread了，如果到這邊還是很難分清IO和執行緒的差異，其實可以了解一下Node.js的運行，因為Node.js就是單執行緒的，所以.net預設的一些Async的Api, 搭配上async await的方式，就是取經了node.js這種可以處理大量request的方式，

如何開始一個新的TPL任務

示例之前先寫一個void的方法，供後面調用和示例

```
void Task1()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Task1");
}
```

有三種方式，第一種方式開始後不會馬上執行，需要再自行調用start

```
Task task1=new Task(Task1);
task1.Start();
```

```
Task.Factory.StartNew(Task1);
Console.WriteLine("Task2");
```

第三種方式則是4.5之後提供，也是最建議方便的使用方式，總是使用一個lambda來開始

```
Task.Run(()=>Task1());
Console.WriteLine("Task2");
```

結果

```
Task2
Task1
```

使用ContinueWith來接續任務

其實我們可以把TPL想成就像javascript的promise，其實使用方式有很多部份都很像，比如目前講到的接續任務，就很像promise.then的方式

```
void Main()
{
        Task.Run(() => Task1())
                .ContinueWith(task => {
                        Task2();
                })
                .ContinueWith(task =>Task3()); //這邊是故意示範兩種寫法，如果我們只是
}

void Task1()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Task1");
}

void Task2()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task1");
}

void Task3()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task2");
}
```

結果

```
Task1
Containue from Task1
Containue from Task2
```

但是如果有錯誤的話，我們就要停止接下來所有的任務呢

```
void Main()
{
        Task.Run(() => Task1())
        .ContinueWith(task => Task2(), TaskContinuationOptions.OnlyOnRanToCompl
        .ContinueWith(task => Task3(), TaskContinuationOptions.OnlyOnRanToCompl
}

void Task1()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Task1");
```

```
}

void Task2()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task1");
}

void Task3()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task2");
}
```

結果

```
Task1
```

等待所有任務結束

等待所有任務結束有兩種方式，一種是waitAll另一種是whenAll，差異的點是waitAll執行但沒有任何回傳值，另一種whenAll有回傳值，可以後續做處理，我們先看一下正常狀況下，completed task會提早跑完。

```
void Main()
{
        var task1 = Task.Run(() => Task1());
        var task2 = Task.Run(() => Task2());
        var task3 = Task.Run(() => Task3());
        Console.WriteLine("completed task");
}

void Task1()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Task1");
}

void Task2()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task1");
}

void Task3()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task2");
}
```

結果

```
completed task
Containue from Task1
Containue from Task2
Task1
```

如果我們想要三個任務一起完成，才能繼續跑completed task的話，就可以使用waitAll的方式

```
void Main()
{
        var task1 = Task.Run(() => Task1());
        var task2 = Task.Run(() => Task2());
        var task3 = Task.Run(() => Task3());
        Task.WaitAll(task1,task2,task3); //在此等待所有任務結果，才繼續往下執行
        Console.WriteLine("completed task");
}
```

```
void Task1()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Task1");
}

void Task2()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task1");
}

void Task3()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Containue from Task2");
}
```

結果

```
Containue from Task2
Task1
Containue from Task1
completed task
```

等待任意一個任務結束

這邊就是只要任何一個任務結束，就會繼續執行下去了，也是有waitAny和wenAny。

```
void Main()
{
        var task1 = Task.Run(() => Task1());
        var task2 = Task.Run(() => Task2());
        Task.WaitAny(task1,task2);
        Console.WriteLine("completed task");
}

void Task1()
{
        Task.Delay(1000).Wait();
        Console.WriteLine("Task1");
}

void Task2()
{
        Task.Delay(2000).Wait();
        Console.WriteLine("Containue from Task1");
}
```

結果

```
Task1
completed task //因為Task1結束了，就繼續跑下去了
Containue from Task1
```

使用Async Await

接下來的案例，都會使用web api的方式來示例，先來看一下TPL如何搭配Async Await

TpiExample.cs

```
public class TplExample
{
    public async Task Task1()
    {
        await Task.Delay(1000);
```

```
    }

    public async Task Task2()
    {
        await Task.Delay(1000);
        Debug.WriteLine("Containue from Task1");
    }

}
```

TestController.cs

雖然能正確執行，但是卻不是真正的達成要使用非同步的目的，即然我們想要使用async的語法，就是希望使用多工進行的方式，而不用必須task1完成之後才跑task2，而是可以一起執行的方式，以上述的方式最後執行結果如下

```
Task1 //這條跑完需要一秒後才會執行下一個
Containue from Task1 //這條跑完才會執行completed task
Completed task
```

接著我們來改成不會一條卡一條的執行方式

```
    public class TestController : ApiController
    {
        public async Task<IHttpActionResult> Get()
        {
            TplExample tpl = new TplExample();
           var task1= tpl.Task1();
            var task2= tpl.Task2();
            Debug.WriteLine("Completed task");
            await task1;
            //在真正要輸出結果的時候才下await，就會等待task1結束再等待task2結束
            //不過因為我們的任務在最上面就同時開跑了，所以最後只是要確保再回傳結果前，所有任
            await task2;
            return Ok();
        }
    }
```

```
Completed task //明顯的已經先跑完最後一行
Containue from Task1
Task1
```

使用一些結尾Async的method，搭配Async Await

我們都知道，在HttpClient甚至Dapper都提供了Async的用法，但在我看過的好幾個專案中，幾乎都沒有看到使用Async的用法，有些是使用wait()或result()來強迫同步，有些則是有用了Async，但不一定有使用Await，很多奇奇怪怪的方式，打個比方HttpClient提供的幾乎都是Async的用法，但是反而造成了很多開發工程師的困擾的感覺，實際上就是非常多工程師對非同步都沒理解，很可惜微軟針對.net提供了很簡單的非同步的用法(跟es7很相像啊)，拿之前寫的例子

```
public class TestController : ApiController
{
    public async Task<IHttpActionResult> Get()
    {
       TplExample tpl = new TplExample();
       var task1= tpl.Task1();
       var task2= tpl.Task2();
       Debug.WriteLine("Completed task");
       await task1;
       await task2;
       return Ok();
    }

    //非常多人用這樣的寫法，不一定是在controller，有可能是充斥在各種類別裡面
    //public IHttpActionResult Get()
    //{
```

```
    //    tpl.Task1().Wait();
    //    tpl.Task2().Wait();
    //    Debug.WriteLine("Completed task");
    //    return Ok();
    //}
}
```

接下來示例一下去串接政府api的資料，來示例一下HttpClient的用法，在此我隨便抓了兩個台北市政府的資料，一個是氣象一個則是台北市府網站最新消息，先建立兩個dto物件

氣象

```
    public class TaipeiWeater
    {
        public Result result { get; set; }

        public class Result
        {
            public int offset { get; set; }
            public int limit { get; set; }
            public int count { get; set; }
            public string sort { get; set; }
            public Result1[] results { get; set; }
        }

        public class Result1
        {
            public string _id { get; set; }
            public string locationName { get; set; }
            public DateTime startTime { get; set; }
            public DateTime endTime { get; set; }
            public string parameterName1 { get; set; }
            public string parameterValue1 { get; set; }
            public string parameterName2 { get; set; }
            public string parameterUnit2 { get; set; }
            public string parameterName3 { get; set; }
            public string parameterUnit3 { get; set; }
        }
    }
```

台北最新消息資料

```
    public class TaipeiData
    {
        public Result result { get; set; }

        public class Result
        {
            public int offset { get; set; }
            public int limit { get; set; }
            public int count { get; set; }
            public string sort { get; set; }
            public Result1[] results { get; set; }
        }

        public class Result1
        {
            public string _id { get; set; }
            public string deptName { get; set; }
            public string data_id { get; set; }
            public string post_date { get; set; }
            public string news_from { get; set; }
            public string authority { get; set; }
            public string contact { get; set; }
            public string contact_phone { get; set; }
            public string news_class { get; set; }
            public string news_title { get; set; }
            public string news_content { get; set; }
```

```
            public string actstart_date { get; set; }
            public string actend_date { get; set; }
            public string s_time { get; set; }
            public string d_time { get; set; }
            public string regis { get; set; }
            public string county { get; set; }
            public string address { get; set; }
            public string cycle_type { get; set; }
            public string cycle_value { get; set; }
            public string news_scope { get; set; }
            public string keyword { get; set; }
            public string link { get; set; }
            public string file { get; set; }
            public string Extra1 { get; set; }
            public string fctupublic { get; set; }
            public string transKey { get; set; }
            public object vgroup { get; set; }
            public string longitude { get; set; }
            public string latitude { get; set; }
        }
    }
```

串接資料的邏輯

```
    public class HttpClientExample
    {
        public async Task<TaipeiWeater> GetWeater()
        {
            using (var httpClient = new HttpClient())
            {
                var result = await httpClient.GetAsync("http://data.taipei/oper
                return await result.Content.ReadAsAsync<TaipeiWeater>();
            }
        }

        public async Task<TaipeiData> GetTaipeiData()
        {
            using (var httpClient = new HttpClient())
            {
                var result = await httpClient.GetAsync("http://data.taipei/oper
                return await result.Content.ReadAsAsync<TaipeiData>();
            }
        }
    }
```

web api的程式碼

```
public async Task<IHttpActionResult> Get()
{
    Stopwatch totalTime = new Stopwatch();
    totalTime.Start();
    var httpClientExample = new HttpClientExample();
    var weaters =await httpClientExample.GetWeater();
    var taipeiData = await httpClientExample.GetTaipeiData();
    totalTime.Stop();
    return Ok(totalTime.ElapsedMilliseconds);
}
```

接著我們來看一下最後執行時間是多少

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<long xmlns="http://schemas.microsoft.com/2003/10/Serialization/">1684</long>
```

```
public async Task<IHttpActionResult> Get()
{
    Stopwatch totalTime = new Stopwatch();
    totalTime.Start();
    var httpClientExample = new HttpClientExample();
    var weaters = httpClientExample.GetWeater();
    var taipeiData =  httpClientExample.GetTaipeiData();
    await weaters;
    await taipeiData;
    totalTime.Stop();
    return Ok(totalTime.ElapsedMilliseconds);
}
```

執行時間明顯快了許多

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<long xmlns="http://schemas.microsoft.com/2003/10/Serialization/">753</long>
```

總是使用Async Await的方式

自從Async Await的方式出來之後，其實我們不管是使用TPL或者是使用一些Async結尾的方法，我們最好一律的都使用Async Await，然後避免的去使用Wait或Result的方式來達成我們想要變成同步的目的，在此需要特別說明一下，如果你使用Result或Wait的話，在沒搭配Async Await的時候，是沒有問題的，但是如果搭配的話，如果是在主控台或windows service的時候，因為在起始點不能使用async和task，所以你依然可以使用Result或Wait的方式，但是如果是在Mvc或Web Api或Win form的話，就會造成死鎖的問題，所以我們不如善加利用Async Await的語法糖，但是請遵照一個原則，就是從頭到尾的使用async await，不過請記得自己使用TPL做的共用方法請勿使用Async的關鍵名詞，以免造成誤解。

結論

其實不管是TPL或執行緒或最新的async await，都是為了多工並行處理的方式，C#因為歷史包袱還有技術會造成硬體負荷不同的緣故，所以會拆分的細一點，所以確實讓人很難理解，到底每種使用方式的差異點在哪邊，什麼時機適合使用呢?其實我個人取決的使用時機點也很簡單，如果第三方或微軟幫我們包裝好結尾命名有Async的，我就視為是I/O Bound，如果沒有現成方法我需要自行處理的，就是需要動用執行緒的方式，也算是一篇自己的記錄，因為我見過太多工程師用很奇怪的方式在操作async，甚至是不太想使用HttpClient來操作，而是寧願使用舊版的WebClient，就是覺得一定要使用async很麻煩，其實微軟已經把非同步包裝的很簡單，不止是使用簡單對效能也有很大的幫助，所以花點時間來了解一下是很有價值的，如果讀者看完覺得有什麼重要我沒寫到的，或者是有什麼錯誤請再指正一下筆者，也希望此篇文章對各位有幫助。

回首頁

系列文章

[c#]執行緒(thread)和執行緒集區(threadpool)的使用

[C#]TPL與Async Await的觀念還有使用方式

[C#]生產者vs消費者的典型案例(使用TPL來加快速度)

[C#]生產者vs消費者的典型案例(使用TPL的DataFlow來加快速度)

[C#]如何正確的不使用await的方式，把async的method改成同步的

[C#]使用rx.net來幫助完成主動通知和訂閱的功能

[C#]Multiple thread和非同步的差異，並正確自訂非同步的方式