عنوان: **متدهای الحاقی و ترکیب کننده‌های اعمال غیرهمزمان**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۵ ۱۴:۳۶
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

تعدادی متد جدید در دات نت 4.5 جهت ترکیب و کار با Taskها اضافه شده‌اند. نمونه‌ای از آن‌را در قسمت‌های قبل با معرفی متد WhenAll مشاهده کردید. در ادامه قصد داریم این متدها را بیشتر بررسی کنیم.

متد WhenAll

کار آن ترکیب تعدادی Task است و اجرای آن‌ها. تنها زمانی خاتمه می‌یابد که کلیه‌ی Taskهای معرفی شده به آن خاتمه یافته باشند. هدف از آن اجرای همزمان و مستقل چندین Task است. برای مثال دریافت چندین فایل به صورت همزمان از اینترنت.
همچنین باید دقت داشت که در اینجا، هر Task کاری به نتایج Taskهای دیگر ندارد و کاملا مستقل اجرا می‌شود. اگر نیاز است Taskها مستقل اجرا شوند، از همان روش سریالی اجرای Taskها، توسط معرفی هر کدام به کمک await استفاده کنید.
به علاوه اگر در این بین استثنایی وجود داشته باشد، تنها پس از پایان عملیات تمام Taskها بازگشت داده می‌شود. این استثناء نیز از نوع Aggregate Exception است.

```
using System.Linq;
using System.Threading.Tasks;

namespace Async07
{
    public class EggBoiler
    {
        private const int BoilingTimeMs = 200;

        private static Task boilEgg()
        {
            var bolingTask = Task.Run(() =>
            {
                Task.Delay(BoilingTimeMs);
            });
            return bolingTask;
        }

        public async Task BoilEggsSequentialAsync(int count)
        {
            for (var i = 0; i < count; i++)
            {
                await boilEgg();
            }
        }

        public async Task BoilEggsSimultaneousAsync(int count)
        {
            var tasksList = from egg in new[] { 1, 2, 3, 4, 5 }
                            select boilEgg();
            await Task.WhenAll(tasksList);
            // ...
        }
    }
}
```

در این مثال عمل پختن تخم مرغ را در یک مدت زمان مشخصی ملاحظه می‌کنید. در متد BoilEggsSequentialAsync، پختن تخم مرغ‌ها، ترتیبی است. ابتدا مورد اول انجام می‌شود و پس از پایان آن، مورد دوم و الی آخر. در اینجا اگر نیاز باشد، می‌توان از نتیجه‌ی عملیات قبلی، در عملیات بعدی استفاده کرد.
اما در متد BoilEggsSimultaneousAsync به علت بکارگیری Task.WhenAll پختن تمام تخم مرغ‌های مدنظر همزمان آغاز می‌شود و تا پایان عملیات (پخته شدن تمام تخم مرغ‌ها) صبر خواهد شد.

متد WhenAny

در حالت استفاده از متد WhenAny، هر کدام از Taskهای در حال پردازش که خاتمه یابند، کل عملیات خاتمه خواهد یافت. فرض

کنید نیاز دارید تا دمای کنونی هوای منطقه‌ی خاصی را از چند وب سرویس مختلف دریافت کنید. می‌توان در این حالت تمام این‌ها را توسط WhenAny ترکیب کرد و هر کدام که زودتر خاتمه یابد، عملیات را پایان خواهد داد.

```
public class Downloader
    {
        private Task<string> downloadTask(string url)
        {
            return new WebClient().DownloadStringTaskAsync(url);
        }

        public async Task<int> GetTemperature()
        {
            var sites = new[]
            {
                "http://www.site1.com/svc",
                "http://www.site2.com/svc",
                "http://www.site3.com/svc",
            };
            var tasksList = from site in sites
                            select downloadTask(site);
            try
            {
                var finishedTask = await Task.WhenAny(tasksList);
                var result = await finishedTask;

            }
            catch (Exception ex)
            {

            }

            // todo: process result, get temperature
            return 10; // for example.
        }
    }
```

در اینجا نحوه‌ی استفاده از WhenAny را مشاهده می‌کنید. نکته‌ی مهم این مثال، استفاده از await دوم بر روی Task بازگشت داده شده‌است. این مساله از این لحاظ مهم است که Task بازگشت داده شده الزامی ندارد که حتما با موفقیت پایان یافته باشد. فراخوانی await بر روی نتیجه‌ی آن سبب خواهد شد تا اگر استثنایی در این بین رخ داده باشد، قابل دریافت و پردازش شود. در این حالت اگر نیاز بود وضعیت سایر Taskها، مثلا در صورت شکست آن‌ها، بررسی شوند، می‌توان از یکی از دو قطعه کد زیر استفاده کرد:

```
foreach (var task in tasksList)
            {
                var ignored = task.ContinueWith(
                    t => Console.WriteLine(t.Exception), TaskContinuationOptions.OnlyOnFaulted);
            }

            // or
            foreach (var task in tasksList)
            {
                var ignored = task.ContinueWith(
                    t =>
                    {
                        if (t.IsFaulted)
                            Console.WriteLine(t.Exception);
                    });
            }
```

کاربرد دیگر WhenAny زمانی است که برای مثال می‌خواهید تعداد زیادی Url را پردازش کنید، اما نمی‌خواهید برای نمایش اطلاعات، تا پایان عملیات تمامی آن‌ها مانند WhenAll صبر کنید. می‌خواهید به محض پایان کار یکی از Taskها، عملیات نمایش نتیجه‌ی آن‌را انجام دهید:

```
public async Task ShowTemperatures()
        {
            var sites = new[]
            {
                "http://www.site1.com/svc",
                "http://www.site2.com/svc",
```

```
            "http://www.site3.com/svc",
        };
        var tasksList = sites.Select(site => downloadTask(site)).ToList();

        while (tasksList.Any())
        {
            try
            {
                var tempTask = await Task.WhenAny(tasksList);
                tasksList.Remove(tempTask);

                var result = await tempTask;
                //todo: show result
            }
            catch(Exception ex) { }
        }
    }
```

در اینجا در یک حلقه، هر Taskایی که زودتر پایان یابد، نمایش داده شده و سپس از لیست وظایف حذف می‌شود. در ادامه مجددا یک await روی آن انجام خواهد شد تا استثنای احتمالی آن بروز کند. سپس اگر مشکلی نبود، می‌توان نتیجه را نمایش داد.

کاربرد سوم WhenAny کنترل تعداد وظایف همزمان است. برای مثال اگر قرار است هزاران تصویر از اینترنت دریافت شوند، نباید تمام وظایف را یکجا راه اندازی کرد. شاید نیاز باشد هربار فقط 15 وظیفه‌ی همزمان عمل کنند و نه بیشتر. در این حالت، مثال قبلی دارای یک حلقه‌ی کنترل کننده tasksList ارائه شده خواهد شد. هر بار تعداد معینی وظیفه به tasksList اضافه و پردازش می‌شوند و این روند تا پایان کار تعداد Urlها ادامه خواهد یافت (یک Take و Skip است؛ مانند صفحه بندی اطلاعات).

متدهای Run و FromResult

متد Task.Run اضافه شده در دات نت 4.5 به این معنا است که می‌خواهید Task ایجاد شده بر روی Thread pool اجرا شود. پارامتر آن می‌تواند یک delegate یا عبارت lambda و یا حتی یک Task باشد. خروجی آن نیز یک Task است و به همین جهت با async و await سی شارپ 5 سازگاری بهتری دارد.
استفاده از Task.Run نسبت به عملیات Threading متداول کارآیی بهتری دارد، زیرا ایجاد Threadهای جدید زمانبر بوده و زمانیکه به صورت خودکار از Thread pool استفاده می‌شود، تا حد امکان، استفاده‌ی مجدد از تردهای بیکار در حال حاضر، مدنظر است.

متد Task.FromResult کار بازگشت یک Task را از نتایج متدهای مختلف فراهم می‌کند. فرض کنید یک متد async تعریف کرده‌اید که خروجی آن Task of T است. در اینجا اگر داخل متد، از یک متد معمولی که یک عدد int را ارائه می‌دهد استفاده کنیم، با استفاده از Task.FromResult بلافاصله می‌توان یک Task of int را بازگشت داد.

متد Delay

پیشتر برای به خواب فرو بردن یک ترد از متد Thread.Sleep استفاده می‌شد. کار Thread.Sleep بلاک کردن ترد جاری است. در دات نت 4.5، بجای آن باید از Task.Delay استفاده شود که یک مکانیزم غیر قفل کننده را جهت صبر کردن به همراه بازگشت یک Task، ارائه می‌دهد.
یکی از کاربردهای Delay منهای صبر کردن تا مدت زمانی مشخص، ایجاد مکانیزم timeout است. برای مثال حالت Task.WhenAny را درنظر بگیرید. اگر در اینجا timeout مدنظر ما 3 ثانیه باشد، می‌توان یکی از Taskها را Task.Delay با آرگومان مساوی 3000 معرفی کرد. اگر هر کدام از taskهای تعریف شده زودتر از 3 ثانیه پایان یافتند که بسیار خوب؛ در غیر اینصورت Task.Delay معرفی شده کار را تمام می‌کند.

متد Yield
متد Task.Yield بسیار شبیه به متد قدیمی DoEvents است که از آن برای اجازه دادن به سایر اعمال جهت اجرا، در بین یک عمل طولانی، استفاده می‌شد.

متد ConfigureAwait

به صورت پیش فرض ادامه یک عملیات همزمان، بر روی ترد ایجاد کننده‌ی آن اجرا می‌شود. برای نمونه اگر یک عملیات async در ترد UI آغاز شود، نتیجه‌ی آن نیز در همان ترد UI بازگشت داده می‌شود. به این ترتیب دیگر نیازی نخواهد بود تا نگرانی در مورد نحوه‌ی دسترسی به مقدار آن توسط عناصر UI داشته باشیم.
اگر به این مساله اهمیت نمی‌دهید، برای مثال اگر اعمال در حال انجام، کاری به عناصر UI ندارند، از متد ConfigureAwait با پارامتر false بر روی یک task پیش از فراخوانی await بر روی آن، استفاده کنید.

```
byte [] buffer = new byte[0x1000];
int numRead;
while((numRead = await source.ReadAsync(buffer, 0, buffer.Length).ConfigureAwait(false)) > 0)
{
  await source.WriteAsync(buffer, 0, numRead).ConfigureAwait(false);
}
```

این مثال در طی یک حلقه، هر بار مقدار کوچکی از منبع ارائه شده به آن را می‌خواند. در اینجا تعداد await cycles قابل توجهی وجود دارند. در هر سیکل نیز از دو فراخوانی async استفاده می‌شود؛ یکی برای انجام عملیات و دیگری برای بازگشت نتیجه به Synchronization Context آغاز کننده آن. با استفاده از ConfigureAwait false زمان اجرای این حلقه به شدت بهبود خواهد یافت و کوتاه‌تر خواهد شد؛ زیرا فاز هماهنگی آن با Synchronization Context حذف می‌شود.

به صورت خلاصه در سی شارپ 5

- بجای task.Wait قدیمی، از await task برای صبر کردن تا پایان یک task استفاده کنید.
- بجای task.Result جهت دریافت یک نتیجه‌ی یک task از await task کمک بگیرید.
- بجای Task.WaitAll از await Task.WhenAll و بجای Task.WaitAny از await Task.WhenAny استفاده نمائید.
- همچنین Thread.Sleep در اعمال async با await Task.Delay جایگزین شده‌است.
- در اعمال غیرهمزمان همیشه متد ConfigureAwait false را بکار بگیرید، مگر اینکه به Context نهایی آن واقعا نیاز داشته باشید. و برای ایجاد یک Task جدید از Task.Run یا TaskFactory.StartNew استفاده نمائید.