عنوان: **آشنایی با مدل برنامه نویسی TAP**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۲ ۱۳:۳۹
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

تاریخچه‌ی اعمال غیر همزمان در دات نت فریم ورک

دات نت فریم ورک، از زمان ارائه نگارش یک آن، از اعمال غیرهمزمان و API خاص آن پشتیبانی می‌کرده‌است. همچنین این مورد یکی از ویژگی‌های Win32 نیز می‌باشد. نوشتن کدهای همزمان متداول بسیار ساده است. در این نوع کدها هر عملیات خاص، پس از پایان عملیات قبلی انجام می‌شود.

```
public string TestNoneAsync()
        {
            var webClient = new WebClient();
            return webClient.DownloadString("http://www.google.com");
        }
```

در این مثال متداول، متد DownloadString به صورت همزمان یا synchronous عمل می‌کند. به این معنا که تا پایان عملیات دریافت اطلاعات از وب، منتظر مانده و ترد جاری را قفل می‌کند. مشکل از جایی آغاز می‌شود که مدت زمان دریافت اطلاعات، طولانی باشد. چون این عملیات در ترد UI در حال انجام است، کل رابط کاربری برنامه تا پایان عملیات نیز قفل شده و دیگر پاسخگوی سایر اعمال رسیده نخواهد بود. در این حالت عموما ویندوز در نوار عنوان برنامه، واژه‌های Not responding را نمایش می‌دهد.
این مورد همچنین در برنامه‌های سمت سرور نیز حائز اهمیت است. با قفل شدن تعداد زیادی ترد در حال اجرا، عملا قدرت پاسخ‌دهی سرور نیز کاهش می‌یابد. بنابراین در این نوع موارد، برنامه‌های چند ریسمانی هرچند در سمت کلاینت ممکن است مفید واقع شوند و برای مثال ترد UI را آزاد کنند، اما اثر آنچنانی بر روی برنامه‌های سمت سرور ندارند. زیرا در آن‌ها می‌توان هزاران ترد را ایجاد کرد که همگی دارای کدهای اصطلاحا blocking باشند. برای حل این مساله استفاده از API غیرهمزمان توصیه می‌شود.
برای نمونه کلاس WebClient توکار دات نت، دارای متدی به نام DownloadStringAsync نیز می‌باشد. این متد به محض فراخوانی، ترد جاری را آزاد می‌کند. به این معنا که فراخوانی آن سبب توقف ترد جاری برای دریافت نتیجه‌ی دریافت اطلاعات از وب نمی‌شود. به این نوع API، یک Asynchronous API گفته می‌شود؛ زیرا با سایر کدهای نوشته شده، هماهنگ و همزمان اجرا نمی‌شود.
هر چند این کد جدید مشکل عدم پاسخ دهی برنامه را برطرف می‌کند، اما مشکل دیگری را به همراه دارد؛ چگونه باید حاصل عملیات آن‌را پس از پایان کار دریافت کرد؟ چگونه باید خطاها و مشکلات احتمالی را مدیریت کرد؟
برای مدیریت این مساله، رخدادی به نام DownloadStringCompleted تعریف شده‌است. روال رویدادگردان آن پس از پایان کار دریافت اطلاعات از وب، فراخوانی می‌گردد.

```
public void TestAsync()
        {
            var webClient = new WebClient();
            webClient.DownloadStringAsync(new Uri("http://www.google.com"));
            webClient.DownloadStringCompleted += webClientDownloadStringCompleted;
        }

        void webClientDownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)
        {
            // use e.Result
        }
```

در اینجا همچنین توسط آرگومان DownloadStringCompletedEventArgs، موفقیت یا شکست عملیات نیز گزارش می‌شود و مقدار e.Result حاصل عملیات است.

مشکل! ما سادگی یک عملیات همزمان را از دست دادیم. متد TestNoneAsync از لحاظ پیاده سازی و همچنین خواندن و نگهداری آن در طول زمان، بسیار ساده‌تر است از نمونه‌ی TestAsync نوشته شده. در کدهای غیرهمزمان فوق، یک متد ساده، به دو متد مجزا خرد شده‌است و نتیجه‌ی نهایی، درون یک روال رخدادگردان بدست می‌آید.

به این مدل، EAP یا Event based asynchronous pattern نیز گفته می‌شود. EAP در دات نت 2 معرفی شد. روال‌های رخدادگردان در این حالت، در ترد اصلی برنامه اجرا می‌شوند. اما اگر به حالت اصلی اعمال غیرهمزمان موجود از دات نت یک کوچ کنیم، اینطور نیست. در WinForms و WPF برای به روز رسانی رابط کاربری نیاز است اطلاعات دریافت شده در همان تردی که رابط کاربری ایجاد شده است، تحویل گرفته شده و استفاده شوند. در غیراینصورت استثنایی صادر شده و برنامه خاتمه می‌یابد.

آشنایی با Synchronization Context

ابتدا یک برنامه‌ی WinForms ساده را آغاز کرده و یک دکمه‌ی جدید را به نام btnGetInfo و یک تکست باکس را به نام txtResults، به آن اضافه کنید. سپس کدهای فرم اصلی آن‌را به نحو ذیل تغییر دهید:

```
using System;
using System.Linq;
using System.Net;
using System.Windows.Forms;

namespace Async02
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnGetInfo_Click(object sender, EventArgs e)
        {
            var req = (HttpWebRequest)WebRequest.Create("http://www.google.com");
            req.Method = "HEAD";
            req.BeginGetResponse(
                asyncResult =>
                {
                    var resp = (HttpWebResponse)req.EndGetResponse(asyncResult);
                    var headersText = formatHeaders(resp.Headers);
                    txtResults.Text = headersText;
                }, null);
        }

        private string formatHeaders(WebHeaderCollection headers)
        {
            var headerString = headers.Keys.Cast<string>()
                                      .Select(header => string.Format("{0}:{1}", header,
headers[header]));
            return string.Join(Environment.NewLine, headerString.ToArray());
        }
    }
}
```

در اینجا از روش دیگری برای دریافت اطلاعات از وب استفاده کرده‌ایم. با استفاده از امکانات HttpWebRequest، کوئری‌های پیشرفته‌تری را می‌توان تهیه کرد. برای مثال می‌توان نوع متد را به HEAD تنظیم نمود؛ تا صرفا مقادیر هدر آدرس درخواستی از سرور، دریافت شوند.

همچنین در این مثال از متد غیرهمزمان BeginGetResponse نیز استفاده شده‌است. در این نوع API خاص، کار با BeginGetResponse آغاز شده و سپس در callback نهایی توسط EndGetResponse، نتیجه‌ی عملیات به دست می‌آید.

اگر برنامه را اجرا کنید، با استثنای زیر مواجه خواهید شد:

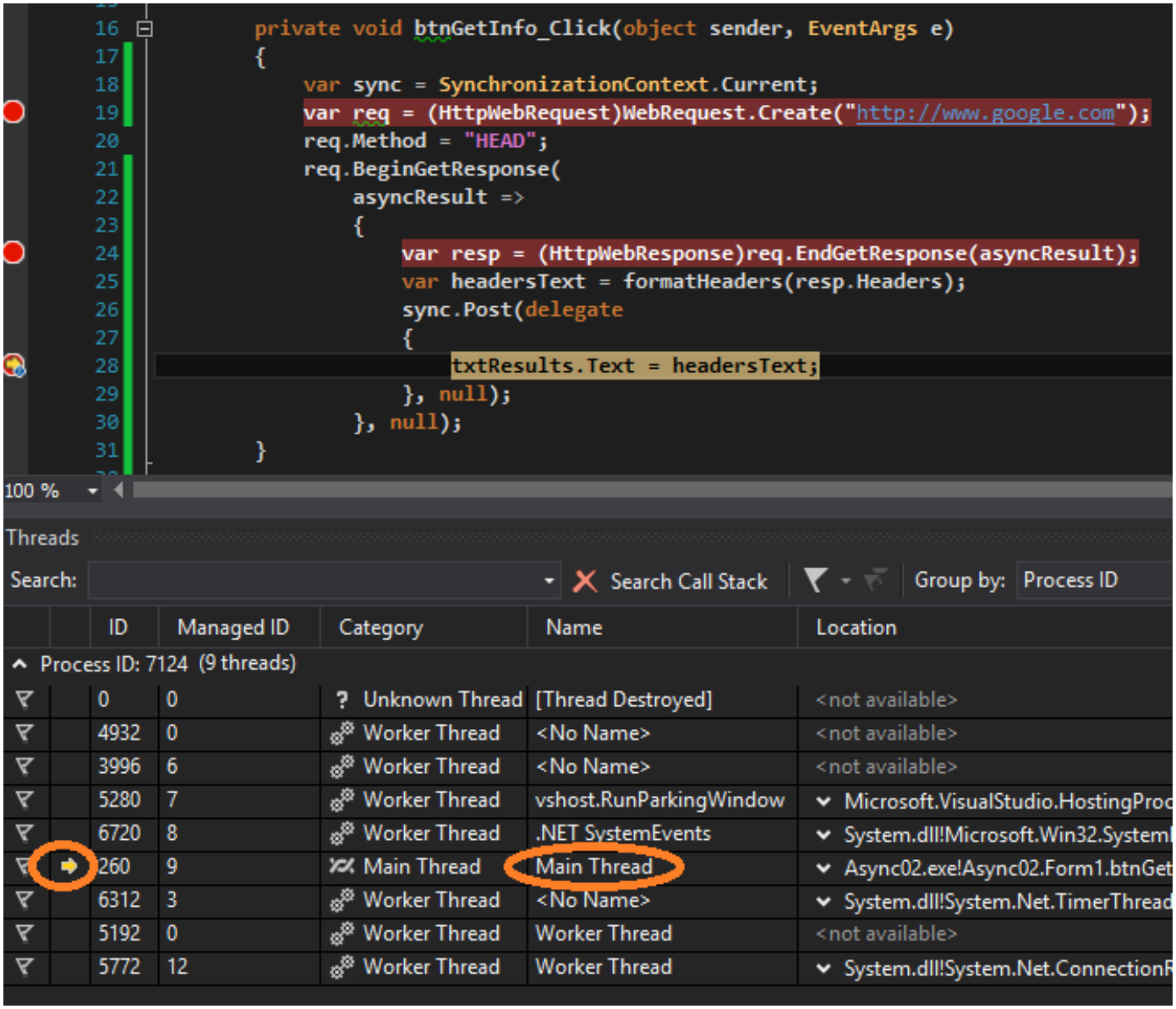
```
An exception of type 'System.InvalidOperationException' occurred in System.Windows.Forms.dll but was
not handled in user code
Additional information: Cross-thread operation not valid: Control 'txtResults' accessed from a thread
other than the thread it was created on.
```

علت اینجا است که asyncResult دریافتی، در تردی دیگر نسبت به ترد اصلی برنامه که UI را اداره می‌کند، اجرا می‌شود. یکی از راه حل‌های این مشکل و انتقال اطلاعات به ترد اصلی برنامه، استفاده از Synchronization Context است:

```
private void btnGetInfo_Click(object sender, EventArgs e)
        {
```

```
            var sync = SynchronizationContext.Current;
            var req = (HttpWebRequest)WebRequest.Create("http://www.google.com");
            req.Method = "HEAD";
            req.BeginGetResponse(
                asyncResult =>
                {
                    var resp = (HttpWebResponse)req.EndGetResponse(asyncResult);
                    var headersText = formatHeaders(resp.Headers);
                    sync.Post(delegate { txtResults.Text = headersText; }, null);
                }, null);
        }
```

SynchronizationContext.Current در اینجا چون در ابتدای متد دریافت اطلاعات اجرا می‌شود، به ترد UI، یا ترد اصلی برنامه اشاره می‌کند. به همین جهت این زمینه را نباید داخل Async callback دریافت کرد؛ زیرا ترد جاری آن، ترد UI مدنظر ما نیست. سپس همانطور که ملاحظه می‌کنید، توسط متد Post آن می‌توان اطلاعات را در زمینه‌ی تردی که SynchronizationContext به آن اشاره می‌کند اجرا کرد.



برای درک بهتر آن، سه break point را پیش از متد BeginGetResponse، داخل Async calback و داخل delegate متد Post قرار

دهید. پس از اجرای برنامه، از منوی دیباگ در VS.NET گزینه‌ی Windows و سپس Threads را انتخاب کنید.
در اینجا همانطور که مشخص است، کد داخل delegate تعریف شده، در ترد اصلی برنامه اجرا می‌شود و نه یکی از Worker threadهای ثانویه.
هر چند استفاده از متدهای تو در تو و lambda syntax، نیاز به تعریف چندین متد جداگانه را برطرف کرده‌است، اما باز هم کد ساده‌ای به نظر نمی‌رسد. در سی شارپ 5، برای مدیریت بهتر تمام مشکلات یاد شده، پشتیبانی توکاری از اعمال غیرهمزمان، به هسته‌ی زبان اضافه شده‌است.

Syntax ابتدایی یک متد Async

در ابتدا کلاس و متد Async زیر را در نظر بگیرید:

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

namespace Async01
{
    public class AsyncExample
    {
        public async Task DoWorkAsync(int parameter)
        {
            await Task.Delay(parameter);
            Console.WriteLine(parameter);
        }
    }
}
```

شیوه‌ی نگارش آن بر اساس راهنمای نوشتن برنامه‌های Async یا Task asynchronous programming model یا به اختصار TAP است:
- در مدل برنامه نویسی TAP، متدهای غیرهمزمان باید یک Task را بازگشت دهند؛ یا نمونه‌ی جنریک آن‌را. البته کامپایلر، async void را نیز پشتیبانی می‌کند ولی در قسمت‌های بعدی بررسی خواهیم کرد که چرا استفاده از آن مشکل‌زا است و باید از آن پرهیز شود.
- همچنین مطابق TAP، اینگونه متدها باید به پسوند Async ختم شوند تا استفاده کننده در حین کار با Intellisense، بتواند آن‌ها را از متدهای معمولی سریعتر تشخیص دهد.
- از واژه‌ی کلیدی async نیز استفاده می‌گردد تا کامپایلر از وجود اعمال غیر همزمان مطلع گردد.
- await به کامپایلر می‌گوید، عبارت پس از من، یک وظیفه‌ی غیرهمزمان است و ادامه‌ی کدهای نوشته شده، تنها زمانی باید اجرا شوند که عملیات غیرهمزمان معرفی شده، تکمیل گردد.

در متد DoWorkAsync، ابتدا به اندازه‌ای مشخص توقف حاصل شده و سپس سطر بعدی یعنی Console.WriteLine اجرا می‌شود.

یک اشتباه عمومی! استفاده از واژه‌های کلیدی async و await متد شما را async نمی‌کنند.

برخلاف تصور ابتدایی از بکارگیری واژه‌های کلیدی async و await، این کلمات نحوه‌ی اجرای متد شما را async نمی‌کنند. این کلمات صرفا برای تشکیل متدهایی که هم اکنون غیرهمزمان هستند، مفید می‌باشند. برای توضیح بیشتر آن به مثال ذیل دقت کنید:

```
public async Task<double> GetNumberAsync()
        {
            var generator = new Random();
            await Task.Delay(generator.Next(1000));

            return generator.NextDouble();
        }
```

در این متد با استفاده از Task.Delay، انجام یک عملیات طولانی شبیه سازی شده‌است؛ مثلا دریافت یک عدد یا نتیجه از یک وب سرویس. سپس در نهایت، عددی را بازگشت داده است. برای بازگشت یک خروجی double، در اینجا از نمونه‌ی جنریک Task استفاده شده‌است.

در ادامه برای استفاده از آن خواهیم داشت:

```
public async Task<double> GetSumAsync()
        {
            var leftOperand = await GetNumberAsync();
            var rightOperand = await GetNumberAsync();

            return leftOperand + rightOperand;
        }
```

خروجی این متد تنها زمانی بازگشت داده می‌شود که نتایج leftOperand و rightOperand از وب سرویس فرضی، دریافت شده باشند و در اختیار مصرف کننده قرارگیرند. بنابراین همانطور که ملاحظه می‌کنید از واژه‌ی کلیدی await جهت تشکیل یک عملیات غیرهمزمان و مدیریت ساده‌تر کدهای نهایی، شبیه به کدهای معمولی همزمان استفاده شده‌است.
در کدهای همزمان متداول، سطر اول ابتدا انجام می‌شود و بعد سطر دوم و الی آخر. با استفاده از واژه‌ی کلیدی await یک چنین عملکردی را با اعمال غیرهمزمان خواهیم داشت. پیش از این برای مدیریت اینگونه اعمال از یک سری callback و یا رخداد استفاده می‌شد. برای مثال ابتدا عملیات همزمانی شروع شده و سپس نتیجه‌ی آن در یک روال رخداد گردان جایی در کدهای برنامه دریافت می‌شد (مانند مثال ابتدای بحث). اکنون تصور کنید که قصد داشتید جمع نهایی حاصل دو عملیات غیرهمزمان را از دو روال رخدادگردان جدا از هم، جمع آوری کرده و بازگشت دهید. هرچند اینکار غیرممکن نیست، اما حاصل کار به طور قطع آنچنان زیبا نبوده و قابلیت نگهداری پایینی دارد. واژه‌ی کلیدی await، انجام اینگونه امور غیرهمزمان را طبیعی و همزمان جلوه می‌دهد. به این ترتیب بهتر می‌توان بر روی منطق و الگوریتم‌های مورد استفاده تمرکز داشت، تا اینکه مدام درگیر مکانیک اعمال غیرهمزمان بود.

امکان استفاده از واژه‌ی کلیدی await در هر جایی از کدها وجود دارد. برای نمونه در مثال زیر، برای ترکیب دو عملیات غیرهمزمان، از await در حین تشکیل عملیات ضرب نهایی، دقیقا در جایی که مقدار متد باید بازگشت داده شود، استفاده شده‌است:

```
public async Task<double> GetProductOfSumAsync()
        {
            var leftOperand = GetSumAsync();
            var rightOperand = GetSumAsync();

            return await leftOperand * await rightOperand;
        }
```

اگر await را از این مثال حذف کنیم، خطای کامپایل زیر را دریافت خواهیم کرد:

```
Operator '*' cannot be applied to operands of type 'System.Threading.Tasks.Task<double>' and
'System.Threading.Tasks.Task<double>'
```

خروجی متد GetSumAsync صرفا یک Task است و نه یک عدد. پس از استفاده از await، عملیات آن انجام شده و بازگشت داده می‌شود.

اگر متد DownloadString همزمان ابتدای بحث را نیز بخواهیم تبدیل به نمونه‌ی async سی‌شارپ 5 کنیم، می‌توان از متد الحاقی جدید آن به نام DownloadStringTaskAsync کمک گرفت:

```
public async Task<string> DownloadAsync()
        {
            var webClient = new WebClient();
            return await webClient.DownloadStringTaskAsync("http://www.google.com");
        }
```

نکته‌ی مهم این کد علاوه بر ساده سازی اعمال غیر همزمان، برای استفاده از نتیجه‌ی نهایی آن، نیازی به SynchronizationContext معرفی شده در تاریخچه‌ی ابتدای بحث نیست. نتیجه‌ی دریافتی از آن در ترد اصلی برنامه تحویل داده شده و به سادگی قابل استفاده است.

سؤال: آیا استفاده از await نیز ترد جاری را قفل می‌کند؟

اگر به کدها دقت کنید، استفاده از await به معنای صبر کردن تا پایان عملیات async است. پس اینطور به نظر می‌رسد که در اینجا نیز ترد اصلی، همانند قبل قفل شده‌است.

```
public void TestDownloadAsync()
        {
            Debug.WriteLine("Before DownloadAsync");
            DownloadAsync();
            Debug.WriteLine("After DownloadAsync");
        }
```

اگر این متد را اجرا کنید (در آن await بکار نرفته)، بلافاصله خروجی ذیل را مشاهده خواهید کرد:

```
Before DownloadAsync
After DownloadAsync
```

به این معنا که در اصل، همانند سایر روش‌های async موجود از دات نت یک، در اینجا نیز فراخوانی متد async ترد اصلی را بلافاصله آزاد می‌کند و ترد آن‌را قفل نخواهد کرد. استفاده از await نیز عملکرد کدها را تغییر نمی‌دهد. تنها کامپایلر در پشت صحنه همان کدهای لازم جهت مدیریت روال‌های رخدادگردان و callbackها را تولید می‌کند، به نحوی که صرفا نحوه‌ی کدنویسی ما همزمان به نظر می‌رسد، اما در پشت صحنه، نحوه‌ی اجرای آن غیرهمزمان است.

برنامه‌های Async و نگارش‌های مختلف دات نت

شاید در ابتدا به نظر برسد که قابلیت‌های جدید async و await صرفا متعلق هستند به دات نت 4.5 به بعد؛ اما خیر. اگر کامپایلری را داشته باشید که از این واژه‌های کلیدی را پشتیبانی کند، امکان استفاده از آن‌ها را با دات نت 4 نیز خواهید داشت. برای این منظور تنها کافی است از VS 2012 به بعد استفاده نمائید. سپس در کنسول پاورشل نیوگت دستور ذیل را اجرا نمائید (فقط برای برنامه‌های دات نت 4 البته):

```
PM> Install-Package Microsoft.Bcl.Async
```

این روال متداول VS.NET بوده است تا به امروز. برای مثال اگر VS 2010 را نصب کنید و سپس یک برنامه‌ی دات نت 3.5 را ایجاد کنید، امکان استفاده‌ی کامل از تمام امکانات سی‌شارپ 4، مانند آرگومان‌های نامدار و یا مقادیر پیش فرض آرگومان‌ها را در یک برنامه‌ی دات نت 3.5 نیز خواهید داشت. همین نکته در مورد async نیز صادق است. VS 2012 (یا نگارش‌های جدیدتر) را نصب کنید و سپس یک پروژه‌ی دات نت 4 را آغاز کنید. امکان استفاده از async و await را خواهید داشت. البته در این حالت دسترسی به متدهای الحاقی جدید را مانند DownloadStringTaskAsync نخواهید داشت. برای رفع این مشکل باید بسته‌ی Microsoft.Bcl.Async را نیز توسط نیوگت نصب کنید.

نظرات خوانندگان

نویسنده: علی رضایی
تاریخ: ۱۳۹۳/۰۱/۰۲ ۱۸:۶

سلام
بسیار بسیار تشکر برای آموزش این بحث جالب.
یک سوال:
موضوع کاربردی که من از این مطلب فهمیدم به شکل زیر است، لطفاً اگر اشتباه است بفرمایید:
در پروژهٔ واقعی که حجم دیتابیس زیاد میشود، ممکن است اندکی زمان برای ذخیره اطلاعات، جستجو و غیره لازم باشد که این باعث عدم پاسخ سرور به سایر درخواست‌ها میشود، حال با استفاده از Async مثلاً در زمان context.SaveChanges این مشکل رفع شده و پس از ثبت اطلاعات آی دی رکورد جدید برگشت داده میشود.
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۲ ۱۸:۳۱

استفاده از async به معنای خالی کردن ترد جاری کدهای مدیریت شده‌ی دات نت است و انجام سایر کارهای برنامه و صبر کردن برای دریافت پاسخی است که در سمت کدهای مدیریت شده نیازی به پردازش و محاسبه ندارد.
برای مثال در حالت کار با یک دیتابیس، این موتور بانک اطلاعاتی است که کوئری رسیده را پردازش می‌کند و برنامه‌ی ما صرفا درخواستی را به آن ارائه داده است. به این ترتیب در اینجا استفاده از async برای خالی کردن ترد جاری و صبر کردن جهت دریافت نتیجه‌ی اطلاعات از سرور مفید است و میزان پاسخ‌دهی برنامه را بالا می‌برد.
بنابراین استفاده از async در سمت کدهای دات نتی، تاثیری بر روی عملکرد یک بانک اطلاعاتی ندارد. فقط در سمت کدهای ما است که برنامه تا رسیدن و محاسبه‌ی درخواست توسط بانک اطلاعاتی، هنگ نمی‌کند.
در این حالت اگر برنامه‌ی شما ویندوزی است، ترد UI آن آزاد شده و برنامه مدام در حال هنگ به نظر نمی‌رسد. اگر برنامه‌ی وب است، ترد جاری آن آزاد شده و thread pool برنامه می‌تواند از این ترد آزاد شده، برای پردازش سایر درخواست‌های رسیده توسط کاربران استفاده کند. به این ترتیب بازدهی و اصطلاحا throughput سرور افزایش پیدا می‌کند.
در حال حاضر تمام APIهای جدید مایکروسافت نسخه‌ی async را هم اضافه کرده‌اند. برای مثال اگر از EF استفاده می‌کنید، از نسخه‌ی 6 آن به بعد، متدهایی مانند ToListAsync برای کوئری گرفتن معمولی غیرهمزمان و SaveChangesAsync برای ذخیره سازی اطلاعات به صورت غیرهمزمان، اضافه شده‌اند. یک مثال کامل در این مورد در اینجا
البته بدیهی است تمام ORMهای دات نتی در سطح پایین خودشان از ADO.NET استفاده می‌کنند. ADO.NET نیز Async API سازگار با دات نت 4.5 به بعد را مدتی است که اضافه کرده‌است. برای مثال متدهایی مانند GetFieldValueAsync ، ExecuteReaderAsync و ExecuteNonQueryAsync و امثال آن به زیر ساخت ADO.NET اضافه شده‌اند. اطلاعات بیشتر

نویسنده: شهروز جعفری
تاریخ: ۱۳۹۳/۰۱/۰۸ ۱۶:۴۷

من یکم گیج شدم:شما فرمودید که : - await به کامپایلر می‌گوید، عبارت پس از من، یک وظیفه‌ی غیرهمزمان است و ادامه‌ی کدهای نوشته شده، تنها زمانی باید اجرا شوند که عملیات غیرهمزمان معرفی شده، تکمیل گردد.
این آیا بدان معنا نیست که ترد اصلی برنامه باید قفل شود؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۸ ۱۷:۰

خیر. در پشت صحنه از یک ماشین حالت (state machine) برای پیاده سازی async استفاده می‌کند. کل سطرهای بعدی تبدیل به یک IEnumerator می‌شوند که هر دستور آن شامل یک yield return است. هر مرحله که تمام شد، این MoveNext IEnumerator فراخوانی می‌شود تا به مرحله‌ی بعدی برسد. به این روش استفاده از coroutines هم گفته می‌شود که در سی شارپ 5، کامپایلر

کار تولید کدهای آن‌را انجام می‌دهد. برای مطالعه بیشتر:
- انجام پی در پی اعمال Async به کمک Iterators - قسمت اول
- انجام پی در پی اعمال Async به کمک Iterators - قسمت دوم

نویسنده: مصطفی عسگری
تاریخ: ۱۳۹۳/۰۱/۰۹ ۱۳:۴۵

سلام
من متد DownloadStringAsync و رویداد مرتبط با آن یعنی DownloadStringCompleted رو تست کردم و به دلیل اینکه متد DownloadStringAsync را در UI Thread صدا میزدم رویداد DownloadStringCompleted نیز همیشه در UI Thread فراخوانی میشد.
من در یک پروژه یک کتابخانه درست کرده بودم که یکی از متدها باید کاری رو به صورت Async انجام میداد و وقتی که کار این متد تمام میشد نتیجه را با Raise کردن یک event به اطلاع استفاده کننده میرسوندم.
اما مشکل اینجا بود که به کنترل‌های روی فرم دسترسی نداشتم و داخل این رویداد ابتدا شرط InvokeRequired و سپس Invoke رو نوشته بودم.این کار مشکل رو حل کرده بود.
اما به نظر من این کار درست نیست.چون من در واقع یکسری API نوشته ام و در اختیار برنامه نویسان دیگر گذاشته ام و آنها باید بتوانند کدهای خود را بدون InvokeRequired درون رویداد بنویسند.
آیا راهی هست که بشه متد من در هر Thread یی اجرا شود رویداد اتمام آن نیز در همان Thread صدازننده ، فراخوانی شود؟
من در این کتابخانه از async و await استفاده نکرده بودم.
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۹ ۱۳:۵۵

- SynchronizationContext از دات نت 2 در دسترس است. بنابراین اجازه دهید مصرف کننده از متد Post آن در صورت صلاحدید، در هر جایی که لازم داشت برای ارسال نتیجه‌ی دریافتی به تردی خاص، مثلا ترد UI استفاده کند. در این مورد در مطلب « استفاده از Async و Await در برنامه‌های دسکتاپ » بیشتر بحث شده‌است.
- SynchronizationContext.Current را اگر پیش از آغاز ترد دریافت کنید، به ترد جاری فراخوان اشاره می‌کند. در پایان ترد، می‌توانید از متد Post آن برای بازگشت به ترد قبلی کمک بگیرید.

عنوان: **مدیریت استثناءها در حین استفاده از واژه‌های کلیدی async و await**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۳ ۱۴:۲۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

زمانیکه یک متد async، یک Task یا Task of T (نسخه‌ی جنریک Task) را باز می‌گرداند، کامپایلر سی‌شارپ به صورت خودکار تمام استثناءهای رخ داده درون متد را دریافت کرده و از آن برای تغییر حالت Task به اصطلاحا faulted state استفاده می‌کند. همچنین زمانیکه از واژه‌ی کلیدی await استفاده می‌شود، کدهایی که توسط کامپایلر تولید می‌شوند، عملا مباحث Continue موجود در TPL یا Task parallel library معرفی شده در دات نت 4 را پیاده سازی می‌کنند و نهایتا نتیجه‌ی Task را در صورت وجود، دریافت می‌کند. زمانیکه نتیجه‌ی یک Task مورد استفاده قرار می‌گیرد، اگر استثنایی وجود داشته باشد، مجددا صادر خواهد شد.
برای مثال اگر خروجی یک متد async از نوع Task of T باشد، امکان استفاده از خاصیتی به نام Result نیز برای دسترسی به نتیجه‌ی آن وجود دارد:

```
using System.Threading.Tasks;

namespace Async05
{
    class Program
    {
        static void Main(string[] args)
        {
            var res = doSomethingAsync().Result;
        }

        static async Task<int> doSomethingAsync()
        {
            await Task.Delay(1);
            return 1;
        }
    }
}
```

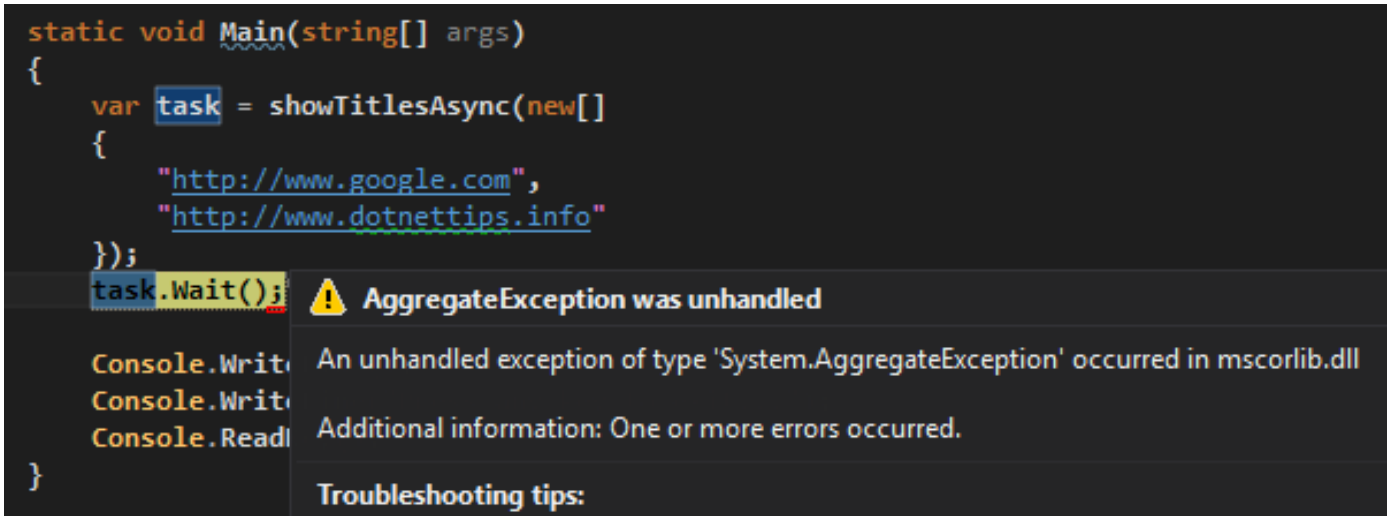
در این مثال یکی از روش‌های استفاده از متدهای async را در یک برنامه‌ی کنسول مشاهده می‌کنید. هر چند خروجی متد doSomethingAsync از نوع Task of int است، اما مستقیما یک int بازگشت داده شده است. تبدیلات نهایی در اینجا توسط کامپایلر انجام می‌شود. همچنین نحوه‌ی استفاده از خاصیت Result را نیز در متد Main مشاهده می‌کنید.
البته باید دقت داشت، زمانیکه از خاصیت Result استفاده می‌شود، این متد همزمان عمل خواهد کرد و نه غیرهمزمان (ترد جاری را بلاک می‌کند؛ یکی از موارد مجاز استفاده از آن در متد Main برنامه‌های کنسول است). همچنین اگر در متد doSomethingAsync استثنایی رخ داده باشد، این استثناء زمان استفاده از Result، به صورت یک AggregateException مجددا صادر خواهد شد. وجود کلمه‌ی Aggregate در اینجا به علت امکان استفاده‌ی تجمعی و ترکیب چندین Task باهم و داشتن چندین شکست و استثنای ممکن است.
همچنین اگر از کلمه‌ی کلیدی await بر روی یک faulted task استفاده کنیم، AggregateException صادر نمی‌شود. در این حالت کامپایلر AggregateException را بررسی کرده و آن‌را تبدیل به یک Exception متداول و معمول کدهای دات نت می‌کند. به عبارتی سعی شده‌است در این حالت، رفتار کدهای async را شبیه به رفتار کدهای متداول همزمان شبیه سازی کنند.

یک مثال

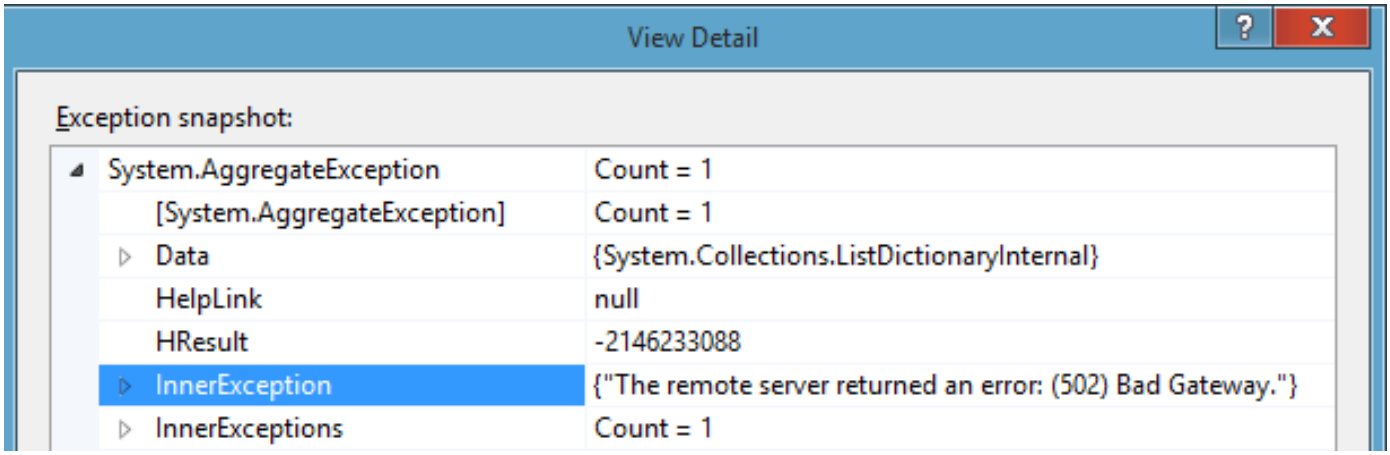
در اینجا توسط متد getTitleAsync، اطلاعات یک صفحه‌ی وب به صورت async دریافت شده و سپس عنوان آن استخراج می‌شود. در متد showTitlesAsync نیز از آن استفاده شده و در طی یک حلقه، چندین وب سایت مورد بررسی قرار خواهند گرفت. چون متد getTitleAsync از نوع async تعریف شده‌است، فراخوان آن نیز باید async تعریف شود تا بتوان از واژه‌ی کلیدی await برای کار با آن استفاده کرد.
نهایتا در متد Main برنامه، وظیفه‌ی غیرهمزمان showTitlesAsync اجرا شده و تا پایان عملیات آن صبر می‌شود. چون خروجی آن از نوع Task است و نه Task of T، در اینجا دیگر خاصیت Result قابل دسترسی نیست. متد Wait نیز ترد جاری را همانند خاصیت Result بلاک می‌کند.

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace Async05
{
    class Program
    {
        static void Main(string[] args)
        {
            var task = showTitlesAsync(new[]
            {
                "http://www.google.com",
                "http://www.dotnettips.info"
            });
            task.Wait();

            Console.WriteLine();
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        static async Task showTitlesAsync(IEnumerable<string> urls)
        {
            foreach (var url in urls)
            {
                var title = await getTitleAsync(url);
                Console.WriteLine(title);
            }
        }

        static async Task<string> getTitleAsync(string url)
        {
            var data = await new WebClient().DownloadStringTaskAsync(url);
            return getTitle(data);
        }

        private static string getTitle(string data)
        {
            const string patternTitle = @"(?s)<title>(.+?)</title>";
            var regex = new Regex(patternTitle);
            var mc = regex.Match(data);
            return mc.Groups.Count == 2 ? mc.Groups[1].Value.Trim() : string.Empty;
        }
    }
}
```

کلیه عملیات مبتنی برشبکه، همیشه مستعد به بروز خطا هستند. قطعی ارتباط یا حتی کندی آن می‌توانند سبب بروز استثناء شوند. برنامه را در حالت عدم اتصال به اینترنت اجرا کنید. استثنای صادر شده، در متد task.Wait ظاهر می‌شود (چون متدهای async ترد جاری را خالی کرده‌اند):



و اگر در اینجا بر روی لینک View details کلیک کنیم، در inner exception حاصل، خطای واقعی قابل مشاهده است:



همانطور که ملاحظه می‌کنید، استثنای صادر شده از نوع System.AggregateException است. به این معنا که می‌تواند حاوی چندین استثناء باشد که در اینجا تعداد آن‌ها با عدد یک مشخص شده‌است. بنابراین در این حالات، بررسی inner exception را فراموش نکنید.

در ادامه داخل حلقه‌ی foreach متد showTitlesAsync، یک try/catch قرار می‌دهیم:

```
static async Task showTitlesAsync(IEnumerable<string> urls)
        {
            foreach (var url in urls)
            {
                try
                {
                    var title = await getTitleAsync(url);
                    Console.WriteLine(title);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex);
                }
            }
        }
```

اینبار اگر برنامه را اجرا کنیم، خروجی ذیل را در صفحه می‌توان مشاهده کرد:

```
System.Net.WebException: The remote server returned an error: (502) Bad Gateway.
System.Net.WebException: The remote server returned an error: (502) Bad Gateway.

Press any key to exit...
```

در اینجا دیگر خبری از AggregateException نبوده و استثنای واقعی رخ داده در متد await شده بازگشت داده شده‌است. کار واژه‌ی کلیدی await در اینجا، بررسی استثنای رخ داده در متد async فراخوانی شده و بازگشت آن به جریان متداول متد جاری است؛ تا نتیجه‌ی عملیات همانند یک کد کامل همزمان به نظر برسد. به این ترتیب کامپایلر توانسته است رفتار بروز استثناءها را در کدهای همزمان و غیرهمزمان یک دست کند. دقیقا مانند حالتی که یک متد معمولی در این بین فراخوانی شده و استثنایی در آن رخ داده‌است.

مدیریت تمام inner exceptionهای رخ داده در پردازش‌های موازی

همانطور که عنوان شد، await تنها یک استثنای حاصل از Task در حال اجرا را به کد فراخوان بازگشت می‌دهد. در این حالت اگر این Task، چندین شکست را گزارش دهد، چطور باید برای دریافت تمام آن‌ها اقدام کرد؟ برای مثال استفاده از Task.WhenAll می‌تواند شامل چندین استثنای حاصل از چندین Task باشد، ولی await تنها اولین استثنای دریافتی را بازگشت می‌دهد. اما اگر از خاصیتی مانند Result یا متد Wait استفاده شود، یک AggregateException حاصل تمام استثناءها را دریافت خواهیم کرد. بنابراین هرچند await تنها اولین استثنای دریافتی را بازگشت می‌دهد، اما می‌توان به Taskهای مرتبط مراجعه کرد و سپس بررسی نمود که آیا استثناهای دیگری نیز وجود دارند یا خیر؟

برای نمونه در مثال فوق، حلقه‌ی foreach تشکیل شده آنچنان بهینه نیست. از این جهت که هر بار تنها یک سایت را بررسی می‌کند، بجای اینکه مانند مرورگرها چندین ترد را به یک یا چند سایت باز کرده و نتایج را دریافت کند.

البته انجام کارها به صورت موازی همیشه ایده‌ی خوبی نیست ولی حداقل در این حالت خاص که با یک یا چند سرور راه دور کار می‌کنیم، درخواست‌های همزمان دریافت اطلاعات، سبب کارآیی بهتر برنامه و بالا رفتن سرعت اجرای آن می‌شوند. اما مثلا در حالتیکه با سخت دیسک سیستم کار می‌کنیم، اجرای موازی کارها نه تنها کمکی نخواهد کرد، بلکه سبب خواهد شد تا مدام drive head در مکان‌های مختلفی مشغول به حرکت شده و در نتیجه کارآیی آن کاهش یابد.

برای ترکیب چندین Task، ویژگی خاصی به زبان سی‌شارپ اضافه نشده، زیرا نیازی نبوده است. برای این حالت تنها کافی است از متد Task.WhenAll، برای ساخت یک Task مرکب استفاده کرد. سپس می‌توان واژه‌ی کلیدی await را بر روی این Task مرکب فراخوانی کرد.

همچنین می‌توان از متد ContinueWith یک Task مرکب نیز برای جلوگیری از بازگشت صرفا اولین استثنای رخ داده توسط کامپایلر، استفاده کرد. در این حالت امکان دسترسی به خاصیت Result آن به سادگی میسر می‌شود که حاوی AggregateException کاملی است.

اعتبارسنجی آرگومان‌های ارسالی به یک متد async

زمان اعتبارسنجی آرگومان‌های ارسالی به متدهای async مهم است. بعضی از مقادیر را نمی‌توان بلافاصله اعتبارسنجی کرد؛ مانند مقادیری که نباید نال باشند. تعدادی دیگر نیز پس از انجام یک Task زمانبر مشخص می‌شوند که معتبر بوده‌اند یا خیر. همچنین فراخوان‌های این متدها انتظار دارند که متدهای async بلافاصله بازگشت داده شده و ترد جاری را خالی کنند. بنابراین اعتبارسنجی‌های آن‌ها باید با تاخیر انجام شود. در این حالات، دو نوع استثنای آنی و به تاخیر افتاده را شاهد خواهیم بود. استثنای آنی زمان شروع به کار متد صادر می‌شود و استثنای به تاخیر افتاده در حین دریافت نتایج از آن دریافت می‌گردد. باید دقت داشت کلیه استثناهای صادر شده در بدنه‌ی یک متد async، توسط کامپایلر به عنوان یک استثنای به تاخیر افتاده گزارش داده می‌شود. بنابراین اعتبارسنجی‌های آرگومان‌ها را بهتر است در یک متد سطح بالای غیر async انجام داد تا بلافاصله بتوان استثناءهای حاصل را دریافت نمود.

از دست دادن استثناءها

فرض کنید مانند مثال قسمت قبل، دو وظیفه‌ی async آغاز شده و نتیجه‌ی آن‌ها پس از await هر یک، با هم جمع زده می‌شوند. در این حالت اگر کل عملیات را داخل یک قطعه کد try/catch قرار دهیم، اولین await ایی که یک استثناء را صادر کند، صرفنظر از وضعیت await دوم، سبب اجرای بدنه‌ی catch می‌شود. همچنین انجام این عملیات بدین شکل بهینه نیست. زیرا ابتدا باید صبر کرد تا اولین Task تمام شود و سپس دومین Task شروع گردد و به این ترتیب پردازش موازی Taskها را از دست خواهیم داد. در یک چنین حالتی بهتر است از متد await Task.WhenAll استفاده شود. در اینجا دو Task مورد نیاز، تبدیل به یک Task مرکب می‌شوند. این Task مرکب تنها زمانی خاتمه می‌یابد که هر دوی Task اضافه شده به آن، خاتمه یافته باشند. به این ترتیب علاوه بر اجرای موازی Taskها، امکان دریافت استثناءهای هر کدام را نیز به صورت تجمعی خواهیم داشت.

مشکل! همانطور که پیشتر نیز عنوان شد، استفاده از await در اینجا سبب می‌شود تا کامپایلر تنها اولین استثنای دریافتی را بازگشت دهد و نه یک AggregateException نهایی را. روش حل آن‌را نیز عنوان کردیم. در این حالت بهتر است از متد ContinueWith و سپس استفاده از خاصیت Result آن برای دریافت کلیه استثناءها کمک گرفت.

حالت دوم از دست دادن استثناءها زمانی‌است که یک متد async void را ایجاد می‌کنید. در این حالات بهتر است از یک Task بجای بازگشت void استفاده شود. تنها علت وجودی async voidها، استفاده از آن‌ها در روال‌های رویدادگردان UI است (در سایر

حالات code smell درنظر گرفته می‌شود).

```
public async Task<double> GetSum2Async()
        {
            try
            {
                var task1 = GetNumberAsync();
                var task2 = GetNumberAsync();

                var compositeTask = Task.WhenAll(task1, task2);
                await compositeTask.ContinueWith(x => { });

                return compositeTask.Result[0] + compositeTask.Result[1];
            }
            catch (Exception ex)
            {
                //todo: log ex
                throw;
            }
        }
```

در مثال فوق، نحوه‌ی ترکیب دو Task را توسط Task.WhenAll جهت اجرای موازی و سپس اعمال نکته‌ی یک ContinueWith خالی و در ادامه استفاده از Result نهایی را جهت دریافت تمامی استثناءهای حاصل، مشاهده می‌کنید.
در این مثال دیگر مانند مثال قسمت قبل

```
public async Task<double> GetSumAsync()
        {
            var leftOperand = await GetNumberAsync();
            var rightOperand = await GetNumberAsync();

            return leftOperand + rightOperand;
        }
```

هر بار صبر نشده‌است تا یک Task تمام شود و سپس Task بعدی شروع گردد.
با کمک متد Task.WhenAll ترکیب آن‌ها ایجاد و سپس با فراخوانی await، سبب اجرای موازی چندین Task با هم شده‌ایم.

مدیریت خطاهای مدیریت نشده

ابتدا مثال زیر را در نظر بگیرید:

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

namespace Async01
{
    class Program
    {
        static void Main(string[] args)
        {
            Test2();
            Test();
            Console.ReadLine();

            GC.Collect();
            GC.WaitForPendingFinalizers();

            Console.ReadLine();
        }

        public static async Task Test()
        {
            throw new Exception();
        }

        public static async void Test2()
        {
            throw new Exception();
        }
    }
```

```
}
```

در این مثال دو متد که یکی async Task و دیگری async void است، تعریف شده‌اند.
اگر برنامه را کامپایل کنید، کامپایلر بر روی سطر فراخوانی متد Test اخطار زیر را صادر می‌کند. البته برنامه بدون مشکل کامپایل خواهد شد.

```
Warning  1  Because this call is not awaited, execution of the current method continues before the call
is completed.
Consider applying the 'await' operator to the result of the call.
```

اما چنین اخطاری در مورد async void صادر نمی‌شود. بنابراین ممکن است جایی در کدها، فراخوانی await فراموش شود. اگر خروجی متد شما ازنوع Task و مشتقات آن باشد، کامپایلر حتما اخطاری را جهت رفع آن گوشزد خواهد کرد؛ اما نه در مورد متدهای void که صرفا جهت کاربردهای UI و روال‌های رخدادگردان آن طراحی شده‌اند.
همچنین اگر برنامه را اجرا کنید استثنای صادر شده در متد async void سبب کرش برنامه می‌شود؛ اما نه استثنای صادر شده در متد async Task. متدهای async void چون دارای Synchronization Context نیستند، استثنای صادره را به Thread pool برنامه صادر می‌کنند. به همین جهت در همان لحظه نیز سبب کرش برنامه خواهند شد. اما در حالت async Task به این نوع استثناءها اصطلاحا Unobserved Task Exception گفته شده و سبب بروز faulted state در Task تعریف شده می‌گردند.
برای مدیریت آن‌ها در سطح برنامه باید در ابتدای کار و در متد Main، توسط TaskScheduler.UnobservedTaskException روال رخدادگردانی را برای مدیریت اینگونه استثناءها تدارک دید. زمانیکه GC شروع به آزاد سازی منابع می‌کند، این استثناءها نیز درنظر گرفته شده و سبب کرش برنامه خواهند شد. با استفاده از متد SetObserved همانند قطعه کد زیر، می‌توان از کرش برنامه جلوگیری کرد:

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

namespace Async01
{
    class Program
    {
        static void Main(string[] args)
        {
            TaskScheduler.UnobservedTaskException += TaskScheduler_UnobservedTaskException;

            //Test2();
            Test();
            Console.ReadLine();

            GC.Collect();
            GC.WaitForPendingFinalizers();

            Console.ReadLine();
        }

        private static void TaskScheduler_UnobservedTaskException(object sender,
UnobservedTaskExceptionEventArgs e)
        {
            e.SetObserved();
            Console.WriteLine(e.Exception);
        }

        public static async Task Test()
        {
            throw new Exception();
        }

        public static async void Test2()
        {
            throw new Exception();
        }
    }
}
```

البته لازم به ذکر است که این رفتار در دات نت 4.5 به این شکل تغییر کرده است تا کار با متدهای async ساده‌تر شود. در دات

نت 4، یک چنین استثناءهای مدیریت نشده‌ای،بلافاصله سبب بروز استثناء و کرش برنامه می‌شدند.
به عبارتی رفتار قطعه کد زیر در دات نت 4 و 4.5 متفاوت است:

```
Task.Factory.StartNew(() => { throw new Exception(); });

Thread.Sleep(100);
GC.Collect();
GC.WaitForPendingFinalizers();
```

در دات نت 4 اگر این برنامه را خارج از VS.NET اجرا کنیم، برنامه کرش می‌کند؛ اما در دات نت 4.5 خیر و آن‌ها به UnobservedTaskException یاد شده هدایت خواهند شد. اگر می‌خواهید این رفتار را به همان حالت دات نت 4 تغییر دهید، تنظیم زیر را به فایل config برنامه اضافه کنید:

```
<configuration>
    <runtime>
      <ThrowUnobservedTaskExceptions enabled="true"/>
    </runtime>
</configuration>
```

یک نکته‌ی تکمیلی: ممکن است عبارات lambda مورد استفاده، از نوع async void باشد.

همانطور که عنوان شد باید از async void منهای مواردی که کار مدیریت رویدادهای عناصر UI را انجام می‌دهند (مانند برنامه‌های ویندوز 8)، اجتناب کرد. چون پایان کار آن‌ها را نمی‌توان تشخیص داد و همچنین کامپایلر نیز اخطاری را در مورد استفاده ناصحیح از آن‌ها بدون await تولید نمی‌کند (چون نوع void اصطلاحا awaitable نیست). به علاوه بروز استثناء در آن‌ها، بلافاصله سبب خاتمه برنامه می‌شود. بنابراین اگر جایی در برنامه متد async void وجود دارد، قرار دادن try/catch داخل بدنه‌ی آن ضروری است.

```
protected override void LoadState(Object navigationParameter, Dictionary<String, Object> pageState)
{
    try
    {
        ClickMeButton.Tapped += async (sender, args) =>
        {
             throw new Exception();

        };
    }
    catch (Exception ex)
    {
        // This won’t catch exceptions!
        TextBlock1.Text = ex.Message;
    }
}
```

در این مثال خاص ویندوز 8، شاید به نظر برسد که try/catch تعریف شده سبب مهار استثنای صادر شده می‌شود؛ اما خیر!

```
public delegate void TappedEventHandler(object sender, TappedRoutedEventArgs e);
```

امضای متد TappedEventHandler از نوع delegate void است. بنابراین try/catch را باید داخل بدنه‌ی روال رویدادگردان تعریف شده قرار داد و نه خارج از آن.

نظرات خوانندگان

نویسنده: لیلا
تاریخ: ۱۳۹۳/۰۴/۲۱ ۲۰:۶

همانطور که در بالا اشاره کردید "در مثال فوق، نحوه‌ی ترکیب دو Task را توسط Task.WhenAll .... "در برخی موارد استفاده از async باعث افزایش کارآیی نیز می‌شود، آیا در موردی که مثلا من در یک اکشن برای انجام کاری نیاز به 4 درخواست مجزا به دیتابیس دارم و بعد از گرفتن نتیجه این 4 درخواست می‌توانم درخواست نهایی را به دیتابیس بفرستم، استفاده از async باعث افزایش کارایی نیز می‌شود ؟

برای تشریح بهتر من نتیجه تست خود را اضافه میکنم. من از mvc5 و EF6 database first استفاده کردم.

حالت sync :

```
var watch = Stopwatch.StartNew();

 int actionId = db.CF_AccessLevel.Where(a => a.Name.ToLower().Trim() == "Edit").Select(a =>
a.CF_AccessLevelId).Single();

 int moduleItemId = db.CF_ModuleItem.Where(m => m.Title.ToLower().Trim() ==
"license".ToLower().Trim()).Select(m => m.CF_ModuleItemId).Single();

 int groupRoleId = db.Users.Where(u => u.UserId == 1).Select(u => u.UserRoleId).Single();

watch.Stop();
 var elapsedMs = watch.ElapsedMilliseconds;
```

حالت async:

```
var watch = Stopwatch.StartNew();
 var something = Task<int>.Factory.StartNew(() => db.CF_AccessLevel.Where(a => a.Name.ToLower().Trim()
== "Edit").Select(a => a.CF_AccessLevelId).Single());
something.Wait();
int actionId = something.Result;

var something1 = Task<int>.Factory.StartNew(() => db.CF_ModuleItem.Where(m => m.Title.ToLower().Trim()
== "license".ToLower().Trim()).Select(m => m.CF_ModuleItemId).Single());
 something1.Wait();
 int moduleItemId = something1.Result;

var something2 = Task<int>.Factory.StartNew(() => db.Users.Where(u => u.UserId == 1).Select(u =>
u.UserRoleId).Single());
 something2.Wait();
 int groupRoleId = something2.Result;
 watch.Stop();
 var elapsedMs = watch.ElapsedMilliseconds;
```

در هر حالت بعد از انجام 3 درخواست ، درخواست نهایی را به سرور میفرستم (در کدهای بالا موجود نیست)
و نتیجه با جزئیات را در آخر اضافه کرده ام :
اما خلاصه میانگین روش sync 222 ms و روش async 191.75ms می‌باشد حدود 35.25ms تفات وجود دارد.
حال آیا تفاوت معنی دار می‌باشد؟ آیا کد async نوشته شده صحیح است؟ اگر صحیح نیست چه روشی صحیح میباشد؟ اگر نباید از async استفاده شود چه روشی بهتر است؟
همانطور که از کد مشخص است برای هدف authorization نوشتم، ولی اگر بخواهم به صورت async در فیلتر استفاده کنم امکانپذیر نیست ، آیا راهی وجود دارد برای استفاده از async در فیلتر سفارشی توی mvc5 ؟

221	202
226	179
208	198
219	197

221	202
245	188
207	195
217	193
220	187
212	171
215	227
312	177
222	187
227	191.75

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۲۱ ۲۰:۵۵

- در مورد EF و متدهای Async آن مطلب جداگانه‌ای تهیه شده: « پردازش‌های Async در Entity framework 6 »
- در مورد ASP.NET MVC و متدهای Async هم یک مطلب اختصاصی تهیه شده: « استفاده از Async و Await در برنامه‌های ASP.NET MVC »
- مثال دوم شما async نیست چون از متد Wait استفاده کرده‌اید (این متد، یک متد blocking است و ترد جاری را قفل می‌کند). این مثال با نمونه‌ی همزمان تقریبا یکسان عمل می‌کند.

همچنین در این مثال استفاده از Task.Factory.StartNew به معنای async تقلبی است و اصلا توصیه نمی‌شود. برای EF متدهای Async واقعی وجود دارند.

- هدف از بکارگیری متدهای async الزاما سریعتر کردن اجرای عملیات مورد نظر نیست. هدف خالی کردن ترد جاری و امکان استفاده‌ی مجدد از آن برای پاسخ دهی به یک کاربر دیگر است؛ با توجه به اینکه هزینه ایجاد تردهای جدید بالا است و همچنین نهایتا بر اساس مشخصات و منابع سرور، این تعداد محدود است. هدف بالا بردن میزان مقیاس پذیری یک برنامه است با تعداد کاربران بالا.

عنوان: **اعمال غیر همزمان و چند ریسمانی**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۳ ۲۳:۳۷
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

تصور عموم بر آن است که اعمال غیر همزمان با چند ریسمانی به یک معنا هستند. این مورد الزاما صحیح نیست. برای مثال دریافت غیرهمزمان یک فایل را از اینترنت درنظر بگیرید. شاید اینطور به نظر برسد که در اینجا یک ترد جدید ایجاد شده و در آن کل کار دریافت فایل آغاز می‌گردد؛ اما خیر. ایجاد یک ترد جدید تنها در قسمت‌های خاصی از یک پروسه انجام می‌شود. همچنین از لحاظ فنی امکان انجام کل کار در یک ترد، بدون بلاک کردن آن وجود دارد. از این جهت که بیشتر زمان، جهت صبر کردن دریافت پاسخی از سرور صرف می‌شود. زمانیکه کلاینت درخواستی را ارسال می‌کند، دیگر کار خاصی را نمی‌تواند انجام دهد تا اینکه پاسخی را دریافت کند.
زمانیکه از یک API غیرهمزمان برای مدیریت چنین عملیاتی استفاده می‌شود، ترد جاری را در این حالت در خواب فرو می‌برد. برای اینکه کار بیشتری برای انجام وجود ندارد. همچنین با اینکه کلاینت درخواستی را ارسال می‌کند یا پاسخی را دریافت، برای مدیریت کل عملیات در اکثر اوقات نیازی به تردها ندارد. این سخت افزار شبکه‌ی نصب شده در سیستم است که عمده‌ی کار را انجام می‌دهد و نه برنامه. زمانیکه برنامه درخواست ارسال اطلاعاتی را بر روی شبکه ارائه می‌دهد، درایور سخت افزار شبکه است که به سخت افزار مرتبط فرمان می‌دهد چه اطلاعاتی را باید ارسال کند. اکثر اینگونه سخت افزارها قادرند اطلاعات را خارج از حافظه‌ی اصلی سیستم دریافت کنند. در اینجا درایور تنها باید به سخت افزار عنوان کند، چه اطلاعاتی را و به کجا باید ارسال کند. بنابراین CPU تنها در طی ارسال این فرمان است که مشغول می‌باشد و نه خارج از آن و این زمان اصلا در مقایسه با زمان ارسال اطلاعات توسط سخت افزار مرتبط، طولانی نیست. CPU مجددا زمانی درگیر خواهد شد که سخت افزار شبکه، اطلاعاتی را دریافت کرده است و باز هم این زمان در مقایسه با زمان دریافت اطلاعات توسط سخت افزار شبکه بسیار کوتاه است.
اغلب کارهای IO به همین شکل هستند. شبیه به همین روند در حالت دسترسی به سخت دیسک وجود دارد. مدت زمانیکه CPU به دیسک کنترلر اعلام می‌کند چه اطلاعاتی را نیاز دارد در مقایسه با مدت زمانیکه دیسک کنترلر این اطلاعات را واقعا بارگذاری می‌کند، بسیار ناچیز است.
نمونه‌ی دیگر آن کار با بانک‌های اطلاعاتی است. در اغلب اوقات برنامه‌ی ما صرفا یک درخواست را به بانک اطلاعاتی ارائه می‌دهد و اصل عملیات در جایی دیگر و توسط موتور بانک اطلاعاتی، خارج از برنامه پردازش می‌گردد.
بنابراین جهت پردازش یک پروسه‌ی خاص، در بسیاری از مراحل آن تنها یک ترد کافی است و هدف اصلی اعمال غیرهمزمان، کاهش تعداد تردهایی است که برنامه جهت پردازش عملیاتی خاص، نیاز دارد. این نوع الگوریتم‌ها طوری طراحی شده‌اند تا تردها تنها زمانی بکار گرفته شود که واقعا CPU قرار است کار خاصی را انجام دهد و نه برای مثال زمانیکه دیسک کنترلر یا سخت افزار شبکه مشغول به کار هستند (و ویندوز به صورت توکار دارای یک چنین API ایی هست). این مساله در سمت کلاینت، سبب خواهد شد تا ترد UI آزاد شود و بتواند به درخواست‌های رسیده کاربر بهتر پاسخ دهد. همچنین این مساله در سمت سرور نیز بسیار مفید است، زیرا برنامه قادر خواهد شد تا به تعداد بیشتری از درخواست‌ها به صورت همزمان پاسخ دهد. زیرا با کاهش تعداد تردهای درگیر، مقیاس پذیری سیستم افزایش می‌یابد.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۳ ۱۱:۴۱

یک مطلب تکمیلی

توضیحات بیشتری در مورد اینکه پردازش اعمال غیرهمزمان واقعی پشتیبانی شده توسط ویندوز، نیازی به ترد اضافی ندارند:

There Is No Thread

عنوان: **تبدیل روش‌های قدیمی کدنویسی غیرهمزمان به async سی شارپ 5**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۴ ۱۷:۳۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

در قسمت اول این سری، با مدل برنامه نویسی Event based asynchronous pattern ارائه شده از دات نت 2 و همچنین APM یا Asynchronous programming model موجود از نگارش یک دات نت، آشنا شدیم (به آن الگوی IAsyncResult هم گفته می‌شود).
نکته‌ی مهم این الگوها، استفاده‌ی گسترده از آن‌ها در کدهای کلاس‌های مختلف دات نت فریم ورک است و برای بسیاری از آن‌ها هنوز async API سازگار با نگارش مبتنی بر Taskهای سی‌شارپ 5 ارائه نشده‌است. هرچند دات نت 4.5 سعی کرده‌است این خلاء را پوشش دهد، برای مثال متد الحاقی DownloadStringTaskAsync را به کلاس WebClient اضافه کرده‌است و امثال آن، اما هنوز بسیاری از کلاس‌های دیگر دات نتی هستند که معادل Task based API ایی برای آن‌ها طراحی نشده‌است. در ادامه قصد داریم بررسی کنیم چگونه می‌توان این الگوهای مختلف قدیمی برنامه نویسی غیرهمزمان را با استفاده از روش‌های جدیدتر ارائه شده بکار برد.

نگاشت APM به یک Task

در قسمت اول، نمونه مثالی را از APM، که در آن کار با BeginGetResponse آغاز شده و سپس در callback نهایی توسط EndGetResponse، نتیجه‌ی عملیات به دست می‌آید، مشاهده کردید. در ادامه می‌خواهیم یک محصور کننده‌ی جدید را برای این نوع API قدیمی تهیه کنیم، تا آن‌را به صورت یک Task ارائه دهد.

```
public static class ApmWrapper
    {
        public static Task<int> ReadAsync(this Stream stream, byte[] data, int offset, int count)
        {
            return Task<int>.Factory.FromAsync(stream.BeginRead, stream.EndRead, data, offset, count,
null);
        }
    }
```

همانطور که در این مثال مشاهده می‌کنید، یک چنین سناریوهایی در TPL یا کتابخانه‌ی Task parallel library پیش بینی شده‌اند.
در اینجا یک محصور کننده برای متدهای BeginRead و EndRead کلاس Stream دات نت ارائه شده‌است. به عمد نیز به صورت یک متد الحاقی تهیه شده‌است تا در حین استفاده از آن اینطور به نظر برسد که واقعا کلاس Stream دارای یک چنین متد Async ایی است. مابقی کار توسط متد Task.Factory.FromAsync انجام می‌شود. متد FromAsync دارای امضاهای متعددی است تا اکثر حالات APM را پوشش دهد.
در مثال فوق BeginRead و EndRead استفاده شده از نوع delegate هستند. چون خروجی EndRead از نوع int است، خروجی متد نیز از نوع Task of int تعیین شده‌است. همچنین سه پارامتر ابتدایی BeginRead ، دقیقا data، offset و count هستند. دو پارامتر آخر آن callback و state نام دارند. پارامتر callback توسط متد FromAsync فراهم می‌شود و state نیز در اینجا null درنظر گرفته شده‌است.
یک مثال استفاده از آن‌را در ادامه مشاهده می‌کنید:

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace Async06
{
    public static class ApmWrapper
    {
        public static Task<int> ReadAsync(this Stream stream, byte[] data, int offset, int count)
        {
            return Task<int>.Factory.FromAsync(stream.BeginRead, stream.EndRead, data, offset, count,
null);
        }
    }

    class Program
    {
        static void Main(string[] args)
```

```
        {
            using (var stream = File.OpenRead(@"..\..\program.cs"))
            {
                var data = new byte[10000];
                var task = stream.ReadAsync(data, 0, data.Length);
                Console.WriteLine("Read bytes: {0}", task.Result);
            }
        }
    }
}
```

File.OpenRead، خروجی از نوع استریم دارد. سپس متد الحاقی ReadAsync بر روی آن فراخوانی شده‌است و نهایتا تعداد بایت خوانده شده نمایش داده می‌شود.
البته همانطور که پیشتر نیز عنوان شد، استفاده از خاصیت Result، اجرای کد را بجای غیرهمزمان بودن، به حالت همزمان تبدیل می‌کند.
در اینجا چون خروجی متد ReadAsync یک Task است، می‌توان از متد ContinueWith نیز بر روی آن جهت دریافت نتیجه استفاده کرد:

```
using (var stream = File.OpenRead(@"..\..\program.cs"))
{
    var data = new byte[10000];
    var task = stream.ReadAsync(data, 0, data.Length);
    task.ContinueWith(t => Console.WriteLine("Read bytes: {0}", t.Result)).Wait();
}
```

یک نکته

پروژه‌ی سورس بازی به نام Async Generator در GitHub، سعی کرده‌است برای ساده سازی نوشتن محصور کننده‌های مبتنی بر Task روش APM، یک Code generator تولید کند. فایل‌های آن‌را از آدرس ذیل می‌توانید دریافت کنید:

https://github.com/chaliy/async-generator

نگاشت EAP به یک Task

نمونه‌ای از Event based asynchronous pattern یا EAP را در قسمت اول، زمانیکه روال رخدادگردان webClient.DownloadStringCompleted را بررسی کردیم، مشاهده نمودید. کار کردن با آن نسبت به APM بسیار ساده‌تر است و نتیجه‌ی نهایی عملیات غیرهمزمان را در یک روال رخدادگران، در اختیار استفاده کننده قرار می‌دهد. همچنین در روش EAP، اطلاعات در همان Synchronization Context ایی که عملیات شروع شده‌است، بازگشت داده می‌شود. به این ترتیب اگر آغاز کار در ترد UI باشد، نتیجه نیز در همان ترد دریافت خواهد شد. به این ترتیب دیگر نگران دسترسی به مقدار آن در کارهای UI نخواهیم بود؛ اما در APM چنین ضمانتی وجود ندارد.
متاسفانه TPL همانند روش FromAsync معرفی شده در ابتدای بحث، راه حل توکاری را برای محصور سازی متدهای روش EAP ارائه نداده‌است. اما با استفاده از امکانات TaskCompletionSource آن می‌توان چنین کاری را انجام داد. در ادامه سعی خواهیم کرد همان متد الحاقی توکار DownloadStringTaskAsync ارائه شده در دات نت 4.5 را از صفر بازنویسی کنیم.

```
public static class WebClientExtensions
    {
        public static Task<string> DownloadTextTaskAsync(this WebClient web, string url)
        {
            var tcs = new TaskCompletionSource<string>();

            DownloadStringCompletedEventHandler handler = null;
            handler = (sender, args) =>
            {
                web.DownloadStringCompleted -= handler;

                if (args.Cancelled)
                {
                    tcs.SetCanceled();
                }
                else if(args.Error!=null)
```

```
            {
                tcs.SetException(args.Error);
            }
            else
            {
                tcs.SetResult(args.Result);
            }
        };

        web.DownloadStringCompleted += handler;
        web.DownloadStringAsync(new Uri(url));

        return tcs.Task;
    }
}
```

روش انجام کار را در اینجا ملاحظه می‌کنید. ابتدا باید تعاریف delaget مرتبط با رخدادگردان Completed اضافه شوند. یکبار += را ملاحظه می‌کنید و بار دوم -= را. مورد دوم جهت آزاد سازی منابع و جلوگیری از نشتی حافظه‌ی روال رخدادگردان هنوز متصل، ضروری است.

سپس از TaskCompletionSource برای تبدیل این عملیات به یک Task کمک می‌گیریم. اگر args.Cancelled مساوی true باشد، یعنی عملیات دریافت فایل لغو شده‌است. بنابراین متد SetCanceled منبع Task ایجاد شده را فراخوانی خواهیم کرد. این مورد استثنایی را در کدهای فراخوان سبب می‌شود. به همین دلیل بررسی خطا با یک if else پس از آن انجام شده‌است. برای بازگشت خطای دریافت شده از متد SetException و برای بازگشت نتیجه‌ی واقعی دریافتی، از متد SetResult می‌توان استفاده کرد.

به این ترتیب متد الحاقی غیرهمزمان جدیدی را به نام DownloadTextTaskAsync برای محصور سازی متد EAP ایی به نام DownloadStringAsync و همچنین رخدادگران آن تهیه کردیم.

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.

عنوان: **لغو اعمال غیرهمزمان**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۶ ۱۵:۵۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

دات نت 4.5 روش عمومی را جهت لغو اعمال غیرهمزمان طولانی اضافه کرده‌است. برای مثال اگر نیاز است تا چندین عمل با هم انجام شوند تا کار مشخصی صورت گیرد و یکی از آن‌ها با شکست مواجه شود، ادامه‌ی عملیات با سایر وظایف تعریف شده، بی‌حاصل است. لغو اعمال در برنامه‌های دارای رابط کاربری نیز حائز اهمیت است. برای مثال یک کاربر ممکن است تصمیم بگیرد تا عملیاتی طولانی را لغو کند.

مدل لغو اعمال

پایه لغو اعمال، توسط مکانیزمی به نام CancellationToken پیاده سازی شده‌است و آن‌را به عنوان یکی از آرگومان‌های متدهایی که لغو اعمال را پشتیبانی می‌کنند، مشاهده خواهید کرد. به این ترتیب یک عمل خاص می‌تواند دریابد چه زمانی لغو آن درخواست شده‌است. البته باید دقت داشت که این عملیات بر مبنای ایده‌ی همه یا هیچ است. به این معنا که یک درخواست لغو را بار دیگر نمی‌توان لغو کرد.

یک مثال استفاده از CancellationToken

کدهای زیر، یک فایل حجیم را از مکانی به مکانی دیگر کپی می‌کنند. برای این منظور از متد CopyToAsync که در دات نت 4.5 اضافه شده‌است، استفاده کرده‌ایم؛ زیرا از مکانیزم لغو عملیات پشتیبانی می‌کند.

```
using System;
using System.IO;
using System.Threading;

namespace Async08
{
    class Program
    {
        static void Main(string[] args)
        {
            var source = @"c:\dir\file.bin";
            var target = @"d:\dir\file.bin";
            using (var inStream = File.OpenRead(source))
            {
                using (var outStream = File.OpenWrite(target))
                {
                    using (var cts = new CancellationTokenSource())
                    {
                        var task = inStream.CopyToAsync(outStream, bufferSize: 4059, cancellationToken:
cts.Token);
                        Console.WriteLine("Press 'c' to cancel.");
                        var key = Console.ReadKey().KeyChar;
                        if (key == 'c')
                        {
                            Console.WriteLine("Cancelling");
                            cts.Cancel();
                        }
                        Console.WriteLine("Wating...");
                        task.ContinueWith(t => { }).Wait();
                        Console.WriteLine("Status: {0}", task.Status);
                    }
                }
            }
        }
    }
}
```

کار با تعریف CancellationTokenSource شروع می‌شود. چون از نوع IDisposable است، نیاز است توسط عبارت using، جهت پاکسازی منابع آن، محصور گردد. سپس در اینجا اگر کاربر کلید c را فشار دهد، متد لغو توکن تعریف شده فراخوانی خواهد شد. این توکن نیز به عنوان آرگومان به متد CopyToAsync ارسال شده‌است.

علت استفاده از ContinueWith در اینجا این است که اگر یک task لغو شود، فراخوانی متد Wait بر روی آن سبب بروز استثناء می‌گردد. به همین جهت توسط ContinueWith یک Task خالی ایجاد شده و سپس بر روی آن Wait فراخوانی گردیده‌است.
همچنین باید دقت داشت که سازنده‌ی CancellationTokenSource امکان دریافت زمان timeout عملیات را نیز دارد. به علاوه متد CancelAfter نیز برای آن طراحی شده‌است. نمونه‌ی دیگری از تنظیم timeout را در قسمت قبل با معرفی متد Task.Delay و استفاده از آن با Task.WhenAny مشاهده کردید.

لغو ظاهری وظایفی که لغو پذیر نیستند

فرض کنید متدی به نام GetBitmapAsync با پارامتر cancellationToken طراحی نشده‌است. در این حالت کاربر قصد دارد با کلیک بر روی دکمه‌ی لغو، عملیات را خاتمه دهد. یک روش حل این مساله، استفاده از متد ذیل است:

```
public static class CancellationTokenExtensions
    {
        public static async Task UntilCompletionOrCancellation(Task asyncOp, CancellationToken ct)
        {
            var tcs = new TaskCompletionSource<bool>();
            using (ct.Register(() => tcs.TrySetResult(true)))
            {
                await Task.WhenAny(asyncOp, tcs.Task);
            }
        }
    }
```

در اینجا از روش Task.WhenAny استفاده شده‌است که در آن دو task ترکیب شده‌اند. Task اول همان وظیفه‌ای اصلی است و task دوم، از یک TaskCompletionSource حاصل شده‌است. اگر کاربر دستور لغو را صادر کند، callback ثبت شده توسط این توکن، اجرا خواهد شد. بنابراین در اینجا TrySetResult به true تنظیم شده و یکی از دو Task معرفی شده در WhenAny خاتمه می‌یابد.
این مورد هر چند task اول را واقعا لغو نمی‌کند، اما سبب خواهد شد تا کدهای پس از await UntilCompletionOrCancellation اجرا شوند.

طراحی متدهای غیرهمزمان لغو پذیر

کلاس زیر را در نظر بگیرید:

```
public class CancellationTokenTest
    {
        public static void Run()
        {
            var cts = new CancellationTokenSource();
            Task.Run(async () => await test(), cts.Token);
            Console.ReadLine();
            cts.Cancel();
            Console.WriteLine("Cancel...");
            Console.ReadLine();
        }

        private static async Task test()
        {
            while (true)
            {
                await Task.Delay(1000);
                Console.WriteLine("Test...");
            }
        }
    }
```

در اینجا cancellationToken متد Task.Run تنظیم شده‌است. همچنین پس از فراخوانی آن، اگر کاربر کلیدی را فشار دهد، متد Cancel این توکن فراخوانی خواهد شد. اما .... خروجی برنامه به صورت زیر است:

```
Test...
```

```
Test...
Test...

Cancel...
Test...
Test...
Test...
Test...
```

بله. وظیفه‌ی شروع شده، لغو شده‌است اما متد test آن هنوز مشغول به کار است.
روش اول حل این مشکل، معرفی پارامتر CancellationToken به متد test و سپس بررسی مداوم خاصیت IsCancellationRequested آن می‌باشد:

```
public class CancellationTokenTest
    {
        public static void Run()
        {
            var cts = new CancellationTokenSource();
            Task.Run(async () => await test(cts.Token), cts.Token);
            Console.ReadLine();
            cts.Cancel();
            Console.WriteLine("Cancel...");
            Console.ReadLine();
        }

        private static async Task test(CancellationToken ct)
        {
            while (true)
            {
                await Task.Delay(1000, ct);
                Console.WriteLine("Test...");

                if (ct.IsCancellationRequested)
                {
                    break;
                }
            }
            Console.WriteLine("Test cancelled");
        }
    }
```

در اینجا اگر متد cts.Cancel فراخوانی شود، مقدار خاصیت ct.IsCancellationRequested مساوی true شده و حلقه خاتمه می‌یابد.
روش دوم لغو عملیات، استفاده از متد Register است. هر زمان که توکن لغو شود، callback آن فراخوانی خواهد شد:

```
private static async Task test2(CancellationToken ct)
        {
            bool isRunning = true;

            ct.Register(() =>
            {
                isRunning = false;
                Console.WriteLine("Query cancelled");
            });

            while (isRunning)
            {
                await Task.Delay(1000, ct);
                Console.WriteLine("Test...");
            }
            Console.WriteLine("Test cancelled");
        }
```

این روش خصوصا برای حالت‌هایی مفید است که در آن‌ها از متدهایی استفاده می‌شود که خودشان امکان لغو شدن را نیز دارند. به این ترتیب دیگر نیازی نیست مدام بررسی کرد که آیا مقدار IsCancellationRequested مساوی true شده‌است یا خیر. هر زمان که callback ثبت شده در متد Register فراخوانی شد، یعنی عملیات باید خاتمه یابد.

عنوان: **گزارش درصد پیشرفت عملیات در اعمال غیرهمزمان**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۷ ۱۳:۴۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

گزارش درصد پیشرفت عملیات در اعمال طولانی، امکان لغو هوشمندانه‌تری را برای کاربر فراهم می‌کند. در دات نت 4.5 دو روش برای گزارش درصد پیشرفت عملیات اعمال غیرهمزمان تدارک دیده شده‌اند:
- اینترفیس جنریک IProgress واقع در فضای نام System
- کلاس جنریک Progress واقع در فضای نام System

در اینجا وهله‌ی از پیاده سازی اینترفیس IProgress به Task ارسال می‌شود. در این بین، عملیات در حال انجام با فراخوانی متد Report آن می‌تواند در هر زمانیکه نیاز باشد، درصد پیشرفت کار را گزارش کند.

```
namespace System
{
  public interface IProgress<in T>
  {
      void Report( T value );
  }
}
```

البته برای اینکه کار تعریف و پیاده سازی اینترفیس IProgress اندکی کاهش یابد، کلاس توکار Progress برای اینکار تدارک دیده شده‌است. نکته‌ی مهم آن استفاده از Synchronization Context برای ارائه گزارش پیشرفت در ترد UI است تا به سادگی بتوان از نتایج دریافتی، در رابط کاربری استفاده کرد.

```
namespace System
{
  public class Progress<T> : IProgress<T>
  {
    public Progress();
    public Progress( Action<T> handler );
    protected virtual void OnReport( T value );
  }
}
```

یک مثال از گزارش درصد پیشرفت عملیات به همراه پشتیبانی از لغو آن

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

namespace Async09
{
    public class TestProgress
    {
        public async Task DoProcessingReportProgress()
        {
            var progress = new Progress<int>(percent =>
            {
                Console.WriteLine(percent + "%");
            });

            var cts = new CancellationTokenSource();

            // call some where cts.Cancel();

            try
            {
                await doProcessing(progress, cts.Token);
            }
            catch (OperationCanceledException ex)
            {
```

```
                //todo: handle cancellations
                Console.WriteLine(ex);
            }

            Console.WriteLine("Done!");
        }

        private static async Task doProcessing(IProgress<int> progress, CancellationToken ct)
        {
            await Task.Run(async () =>
            {
                for (var i = 0; i != 100; ++i)
                {
                    await Task.Delay(100, ct);
                    if (progress != null)
                        progress.Report(i);

                    ct.ThrowIfCancellationRequested();
                }
            }, ct);
        }
    }
}
```

متد private static async Task doProcessing طوری طراحی شده‌است که از مفاهیم لغو یک عملیات غیرهمزمان و همچنین گزارش درصد پیشرفت آن توسط اینترفیس IProgress پشتیبانی می‌کند. در اینجا هر زمانیکه نیاز به گزارش درصد پیشرفت باشد، متد Report وهله‌ی ارسالی به آرگومان progress فراخوانی خواهد شد.

برای تدارک این وهله، از کلاس توکار Progress دات نت در متد public async Task DoProcessingReportProgress استفاده شده‌است.

این متد جنریک بوده و برای مثال نوع آن در اینجا int تعریف شده‌است. سازنده‌ی آن می‌تواند یک callback را قبول کند. هر زمانیکه متد Report در متد doProcessing فراخوانی گردد، این callback در سمت کدهای استفاده کننده، فراخوانی خواهد شد. مثلا توسط مقدار آن می‌توان یک Progress bar را نمایش داد.

به علاوه روش دیگری را در مورد لغو یک عملیات در اینجا ملاحظه می‌کنید. متد ThrowIfCancellationRequested نیز سبب خاتمه‌ی عملیات می‌گردد؛ البته اگر در کدهای برنامه در جایی متد Cancel توکن، فراخوانی گردد. برای مثال یک دکمه‌ی لغو عملیات در صفحه قرارگیرد و کار آن صرفا فراخوانی cts.Cancel باشد.

عنوان: **استفاده از Async و Await در برنامه‌های دسکتاپ**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۸ ۱۸:۳۳
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

امکان استفاده از قابلیت‌های غیرهمزمان دات نت 4.5 در برنامه‌های WPF نیز به روش‌های مختلفی میسر است که در ادامه دو روش مرسوم آن‌را بررسی خواهیم کرد.

تهیه مقدمات بحث

ابتدا یک برنامه‌ی WPF جدید را آغاز کنید. سپس کدهای MainWindow.xaml آن‌را به نحو ذیل تغییر دهید.

```
<Window x:Class="Async10.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <DockPanel>
        <DockPanel Dock="Top">
            <Button Name="BtnGo" Content="Go" Click="BtnGo_OnClick" />
            <ProgressBar Name="ProgressBar" IsIndeterminate="True" Visibility="Collapsed"/>
        </DockPanel>
        <TextBox Name="Results"/>
    </DockPanel>
</Window>
```

قصد داریم اطلاعاتی را از وب دریافت و سپس در TextBox قرار گرفته در صفحه نمایش دهیم.
در این مثال از کلاس جدید HttpClient نیز استفاده خواهیم کرد. برای استفاده از آن نیاز است ارجاعی را به اسمبلی استاندارد System.Net.Http.dll نیز به پروژه اضافه کنید.

روش اول

در ادامه کدهای فایل MainWindow.xaml.cs را به نحو ذیل تغییر داده و سپس برنامه را اجرا کنید.

```
using System.Net.Http;
using System.Windows;

namespace Async10
{
    public partial class MainWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void BtnGo_OnClick(object sender, RoutedEventArgs e)
        {
            BtnGo.IsEnabled = false;
            ProgressBar.Visibility = Visibility.Visible;

            var url = "http://www.dotnettips.info";
            var client = new HttpClient(); // make sure you have an assembly reference to
System.Net.Http.dll
            client.DefaultRequestHeaders.UserAgent.ParseAdd("Test Async");
            var task = client.GetStringAsync(url);
            task.ContinueWith(t =>
            {
                Results.Text = t.Result;

                BtnGo.IsEnabled = true;
                ProgressBar.Visibility = Visibility.Collapsed;
            });
        }
    }
}
```

روال رخدادگردان BtnGo_OnClick به نحو مرسوم آن نوشته شده است. بنابراین جهت دریافت نتیجه‌ی متد GetStringAsync می‌توان از متد ContinueWith بر روی task دریافت اطلاعات از وب، استفاده کرد. همچنین در اینجا مستقیما اطلاعات و نتیجه‌ی دریافتی را به عناصر UI انتساب داده‌ایم.
اگر پروژه را اجرا کنید، برنامه با استثنای زیر متوقف می‌شود:

```
The calling thread cannot access this object because a different thread owns it.
```

چون task آغاز شده در ترد دیگری نسبت به ترد UI اجرا می‌شود، مجوز تغییری را در کدهای UI ندارد. برای حل این مشکل می‌توان از دو روش ذیل استفاده کرد:

الف) با استفاده از SynchronizationContext.Current و متد Post آن

```
var context = SynchronizationContext.Current;
task.ContinueWith(t => context.Post(state =>
{
   Results.Text = t.Result;

   BtnGo.IsEnabled = true;
   ProgressBar.Visibility = Visibility.Collapsed;
}, null));
```

با این روش در قسمت اول آشنا شدید. SynchronizationContext.Current در اینجا چون در ابتدای متد و خارج از ContinueWith دریافت اطلاعات، اجرا می‌شود، به ترد UI یا ترد اصلی برنامه اشاره می‌کند. سپس همانطور که ملاحظه می‌کنید، توسط متد Post آن می‌توان اطلاعات را در زمینه‌ی تردی که SynchronizationContext به آن اشاره می‌کند اجرا کرد.

ب) با استفاده از امکانات TaskScheduler

```
var taskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
task.ContinueWith(t =>
{
   Results.Text = t.Result;

   BtnGo.IsEnabled = true;
   ProgressBar.Visibility = Visibility.Collapsed;
}, taskScheduler);
```

وقتی یک task اجرا می‌شود، TPL یا task parallel library نیاز دارد بداند، این task بر روی چه تردی و چه زمانی قرار است اجرا شود. به صورت پیش فرض از thread pool استفاده می‌کند، اما الزامی به آن نیست. با استفاده از TaskScheduler می‌توان بر روی نحوه‌ی رفتار تردهای TPL تاثیر گذاشت و یا حتی آن‌ها را سفارشی سازی کرد. متد FromCurrentSynchronizationContext، یک TaskScheduler جدید را در اختیار ما قرار می‌دهد که کدهای آن بر اساس SynchronizationContext.Current کار می‌کند؛ در اینجا Context به UI اشاره می‌کند و در یک برنامه‌ی وب، به یک درخواست رسیده.
برای مثال اگر در برنامه‌های وب یک Task جدید را اجرا کنید شاید اینطور به نظر برسد که به HttpContext دسترسی ندارید. این نقیصه را می‌توان توسط کار با SynchronizationContext جاری برطرف کرد.
در مثال فوق، چون taskScheduler پیش از فراخوانی متد ContinueWith ایجاد شده‌است، به ترد UI اشاره می‌کند. در این حالت برای نمایش اطلاعات در همان ترد اصلی برنامه کافی است این taskScheduler را به عنوان پارامتر متد ContinueWith معرفی کنیم.

روش دوم

در دات نت 4.5 می‌توان روال رخدادگردان تعریف شده را به صورت async نیز معرفی کرد (یعنی مجاز هستیم امضای متد پیش فرض تولید شده را تغییر دهیم):

```
private async void BtnGo_OnClick(object sender, RoutedEventArgs e)
```

سپس استفاده از await در کدهای برنامه میسر خواهد شد:

```
private async void BtnGo_OnClick(object sender, RoutedEventArgs e)
        {
            BtnGo.IsEnabled = false;
            ProgressBar.Visibility = Visibility.Visible;

            var url = "http://www.dotnettips.info";
            var client = new HttpClient(); // make sure you have an assembly reference to
System.Net.Http.dll
            client.DefaultRequestHeaders.UserAgent.ParseAdd("Test Async");
            Results.Text = await client.GetStringAsync(url);
            BtnGo.IsEnabled = true;
            ProgressBar.Visibility = Visibility.Collapsed;
        }
```

در این حالت دیگر نیازی به استفاده از ContinueWith و مباحث SynchronizationContext نیست. زیرا تمام آن‌ها به صورت توکار اعمال می‌شوند. به علاوه کدنهایی نیز بسیار خواناتر شده‌است.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۳ ۱۱:۳۰

یک نکته‌ی تکمیلی

اگر از الگوی MVVM استفاده می‌کنید، یک پیاده سازی AsyncCommand را در اینجا می‌توانید ملاحظه کنید:

Patterns for Asynchronous MVVM Applications: Commands

عنوان: استفاده از Async و Await در برنامه‌های ASP.NET MVC
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۰۹ ۱۴:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

از ASP.NET MVC 4 به بعد، امکان استفاده از اکشن متدهای async در ASP.NET MVC میسر شده‌است. البته همانطور که پیشتر نیز ذکر شد ، شرط استفاده از امکانات async در نگارش‌های پیش از دات نت 4.5، استفاده از کامپایلری است که بتواند کدهای async را تولید کند و این مورد تنها از VS 2012 به بعد ممکن شده‌است.

علت استفاده از اکشن متدهای async در ASP.NET MVC

اگر نیاز دارید که برنامه‌ی وبی، به شدت مقیاس پذیر را تولید کنید، باید بتوانید مجموعه تردهای سیستم را تا حد ممکن مشغول به کار و سرویس دهی نگه دارید. در برنامه‌های وب ASP.NET تنها تعداد مشخصی ترد، برای پاسخ دهی به درخواست‌های رسیده، همواره مشغول به کار می‌باشند. در اینجا اگر این تردها را برای مدت زمان زیادی جهت اعمال IO مشغول نگه داریم، دست آخر به سیستمی خواهیم رسید که تردهای مفید آن، جهت پایان عملیات مرتبط بیکار شده‌اند و دیگر ASP.NET قادر نیست از آن‌ها جهت پاسخ‌دهی به سایر درخواست‌های رسیده استفاده کند.
برای مثال یک اکشن متد را درنظر بگیرید که نیاز است با یک وب سرویس، برای دریافت نتیجه کار کند. اگر این عملیات اندکی طول بکشد، به همین میزان ترد جاری درحال پردازش این درخواست، بیکار شده و منتظر دریافت پاسخ خواهد ایستاد و اگر به همین ترتیب تعداد تردهای بیکار، بیشتر و بیشتر شوند، دیگر سیستم قادر نخواهد بود به درخواست‌های جدید رسیده پاسخ دهد و ASP.NET مجبور خواهد شد این درخواست‌ها را در صف قرار دهد تا بالاخره زمانی این تردها آزاد شده و قابل استفاده‌ی مجدد گردند. برای رفع این مشکل، استفاده از اعمال غیرهمزمان ابداع گردیدند تا در آن‌ها ترد مورد استفاده جهت پردازش درخواست رسیده را آزاد کرده و به این ترتیب دیگر نیازی نباشد تا تردجاری، تا پایان عملیات IO بلاک شده و بدون استفاده باقی بماند.
در ASP.NET MVC 3 برای نوشتن اکشن متدهای async می‌بایستی از روش قدیمی مدل‌های Async در دات نت مانند APM استفاده می‌شد و همچنین کنترلر جاری بجای ارث بری از کلاس Controller می‌بایستی از کلاس AsyncController مشتق می‌شد. به علت سخت بودن استفاده از آن، این روش و کنترلرهای async در نگاش 3 آن آنچنان مقبولیت و استفاده‌ی گسترده‌ای نیافتند. چون هر اکشن متد تبدیل می‌شد به دو قسمت Begin و End متداول روش‌های APM. سپس در کشن متد دومی، نتیجه‌ی این عملیات به View بازگشت داده می‌شد.
از ASP.NET MVC 4 به بعد، خالی کردن تردهای سیستم و استفاده‌ی مجدد و مشغول به کار نگه داشتن مداوم آن‌ها با استفاده از امکانات توکار زبان‌هایی مانند سی‌شارپ 5، ساده‌تر و خواناتر شده‌است.
البته باید دقت داشت که این بحث صرفا سمت سرور بوده و ارتباطی به مباحث غیرهمزمان سمت کلاینت، مانند Ajax ندارد (A در Ajax نیز به معنای Async است) و از دید مصرف کننده‌ی نهایی، نامرئی می‌باشد. کار Aajx در سمت کلاینت نیز خالی کردن ترد UI مرورگر است (و نه سرور) و در نهایت تهیه‌ی برنامه‌هایی با قابلیت پاسخ‌دهی بهتر.

نوشتن اکشن متدهای Async در ASP.NET MVC

اولین کاری که باید صورت گیرد، اندکی تغییر امضای اکشن متدهای متداول است:

```
public ActionResult Index()
```

این اکشن متد متداول، در یک ترد اجرا شده و این ترد تا پایان کار آن بلاک خواهد شد. برای مثال اگر قرار است مانند قسمت قبل، متد GetStringAsync در آن پردازش شود، تا پایان مدت زمان پردازش این متد، ترد جاری بلاک شده و سیستم قادر به استفاده‌ی مجدد از آن جهت پاسخ‌دهی به سایر درخواست‌های رسیده نخواهد بود. برای تبدیل آن به یک اکشن متد async باید به نحو ذیل عمل کرد:

```
public async Task<ActionResult> Index()
```

ابتدا واژه‌ی کلیدی async به ابتدای امضای متد اضافه می‌شود. سپس خروجی آن اینبار بجای ActionResult، نسخه‌ی جنریک

Task of T خواهد بود. همچنین دیگر نیازی نیست مانند MVC 3، کنترلر جاری از کلاس AsyncController مشتق شود.
زمانیکه به امضای متدی، async اضافه می‌شود، یعنی جایی در داخل بدنه‌ی آن باید await بکار رود:

```
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Mvc;

namespace Async11.Controllers
{
    public class HomeController : Controller
    {
        public async Task<ActionResult> Index()
        {
            var url = "http://www.dotnettips.info";
            var client = new HttpClient(); // make sure you have an assembly reference to
System.Net.Http.dll
            client.DefaultRequestHeaders.UserAgent.ParseAdd("Test Async");
            var result = await client.GetStringAsync(url);
            return View(result);
        }
    }
}
```

بنابراین اگر داخل اکشن متد جاری، جایی از await استفاده نمی‌شود، async کردن آن بی‌معنا است. این await است که سبب آزاد شدن ترد جاری جهت استفاده‌ی مجدد از آن برای پاسخ‌دهی به سایر درخواست‌های رسیده می‌شود.

یک نکته در مورد WCF 4.5

از WCF 4.5 به بعد، در صفحه‌ی معروف Add service references آن، با کلیک بر روی گزینه‌ی advanced و تنظیمات سرویس، امکان انتخاب گزینه‌ی Create task based operations نیز وجود دارد. این مورد دقیقا برای سهولت استفاده از آن با async و await سی‌شارپ 5 و مدل TAP آن طراحی شده‌است.

تعیین timeout در اکشن متدهای async

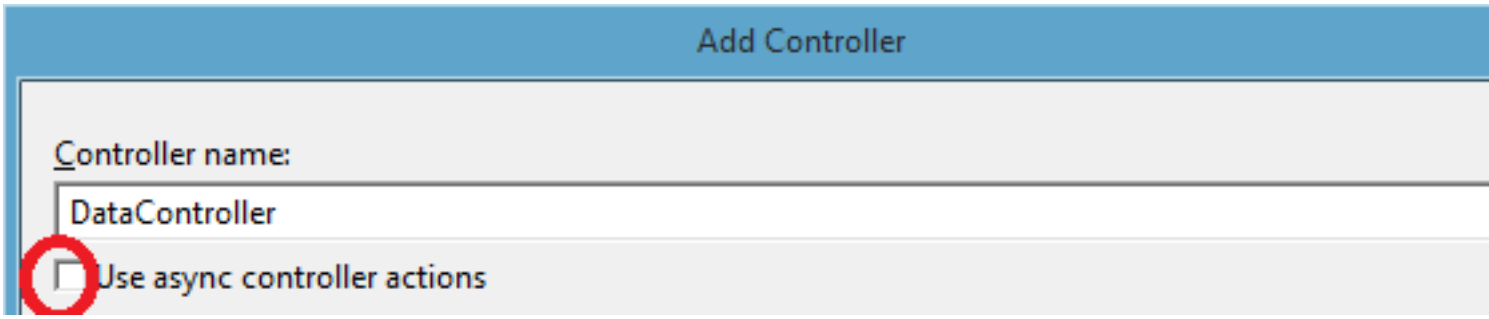
برای مشخص سازی صریح timeout در اکشن متدهای غیرهمزمان، می‌توان از ویژگی خاصی به نام AsyncTimeout به نحو ذیل استفاده کرد:

```
[AsyncTimeout(duration: 1200)]
  public async Task<ActionResult> Index(CancellationToken ct)
```

در مورد لغو اعمال غیرهمزمان پیشتر صحبت شد . در اینجا پارامتر CancellationToken توسط فریم ورک جاری تنظیم شده و می‌توان آن‌را به متدهایی که قادرند اعمال غیر همزمان خود را بر اساس درخواست رسیده CancellationToken لغو کنند، ارسال کرد.

استفاده از قابلیت‌های غیرهمزمان EF 6 به همراه ASP.NET MVC 5

EF 6 به همراه یک سری متد و همچنین متد الحاقی جدید است که اعمال Async را پشتیبانی می‌کنند. اگر در حین انتخاب گزینه‌ی ایجاد کنترلر جدید، گزینه‌ی MVC 5 Controller with views, using EF را انتخاب کنید، امکان تولید اکشن متدهای async نیز به صورت پیش فرض پیش بینی شده‌است:




```
public async Task<ActionResult> Index()
  {
     var model = await db.Books.ToListAsync();
     return View(model);
  }
```

در اینجا نیز امضای اکشن متد، همانند توضیحاتی است که در ابتدای بحث ارائه شد. فقط بجای متد ToList معمولی EF، از نگارش Async آن استفاده شده‌است و همچنین برای دریافت نتیجه‌ی آن از کلمه‌ی کلیدی await استفاده گردیده است.
به علاوه متد Find اکنون معادل FindAsync نیز دارد و همچنین SaveChanges دارای معادل غیرهمزمانی شده‌است به نام SaveChangesAsync .
البته باید دقت داشت که برای Where معادل Async ایی طراحی نشده‌است؛ زیرا نوع IQueryable صرفا یک عبارت است و اجرای آن تا زمانیکه First، ToList و امثال آن فراخوانی نشوند، به تعویق خواهد افتاد.

نظرات خوانندگان

نویسنده: فواد عبداللهی
تاریخ: ۱۳۹۳/۰۱/۱۶ ۱۱:۱۶

سلام؛ اگر

```
var model = await db.Books.ToListAsync();
```

همزمان اجرا میشه ولی بازم برای return باید منتظر پاسخ از db بمونه! پس اینجا فایده ای نداره؟
مشکل من اینجاست که فکر میکنم این روش تنها برای قسمت هایی بدرد میخوره که به هم وابسته نیستن. برای مثال وقتی یه فایل رو آپلود میکنی و بعد آدرس فایل رو ذخیره کنیم فایده نداره. چون تا فایل آپلود نشه ذخیره آدرس تو db بی معنیه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۶ ۱۱:۴۲

- پیشنیاز مطالعه قسمت جاری، مطالعه 6 قسمت اول این دوره است.
- «همزمان اجرا میشه»
خیر. متدهای Async واقعی مثل نمونه ارائه شده در EF غیرهمزمان اجرا می‌شوند. یعنی، ترد جاری را آزاد کرده و ASP.NET می‌تواند از آن ترد برای پاسخ دهی به یک درخواست رسیده دیگر استفاده کند.
- «باید منتظر پاسخ از db بمونه»
استفاده از await و async سبب بازنویسی بدنه متد توسط یک state machine در پشت صحنه می‌شوند. یعنی اینطور نیست که روش اجرای آن blocking است و تا رسیدن پاسخ از بانک اطلاعاتی، از این ترد دیگر نمی‌شود استفاده کرد. جایی که await فراخوانی می‌شود، ترد جاری برای استفاده بعدی آزاد خواهد شد. در ادامه مابقی کدها تبدیل به یک IEnumerator می‌شوند که هر دستور آن شامل یک yield return است. هر مرحله که تمام شد، MoveNext این IEnumerator فراخوانی می‌شود تا به مرحله‌ی بعدی برسد. به این روش استفاده از coroutines هم گفته می‌شود که در سی شارپ 5، کامپایلر کار تولید کدهای آن‌را انجام می‌دهد. برای مطالعه بیشتر:
- انجام پی در پی اعمال Async به کمک Iterators - قسمت اول
- انجام پی در پی اعمال Async به کمک Iterators - قسمت دوم
- «چون تا فایل آپلود نشه ذخیره آدرس تو db بی معنیه»
ذخیره آدرس هم یک قسمت از کار است و اتفاقا وابسته به سیستم جاری هم نیست. وابسته است به یک بانک اطلاعاتی که خارج از مرزهای سیستم، به صورت مستقل در حال فعالیت است (عموما البته؛ مثلا اگر از SQL Server استفاده می‌شود).
برای ذخیره فایل‌ها در سیستم هم متدهای Async به کلاس Stream در دات نت 4.5 اضافه شده‌اند؛ مثل WriteAsync . در این حالت هم می‌توان از await WriteAsync برای ذخیره اطلاعات و بازهم آزاد کردن ترد جاری استفاده کرد.

عنوان: **متدهای async تقلبی**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۰ ۱۸:۱۳
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

تا اینجا مشاهده کردیم که اگر یک چنین متد زمانبری را داشته باشیم که در آن عملیاتی طولانی انجام می‌شود،

```
class MyService
{
  public int CalculateXYZ()
  {
    // Tons of work to do in here!
    for (int i = 0; i != 10000000; ++i)
      ;
    return 42;
  }
}
```

برای نوشتن معادل async آن فقط کافی است که امضای متد را به async Task تغییر دهیم و سپس داخل آن از Task.Run استفاده کنیم:

```
class MyService
{
  public async Task<int> CalculateXYZAsync()
  {
    return await Task.Run(() =>
    {
      // Tons of work to do in here!
      for (int i = 0; i != 10000000; ++i)
        ;
      return 42;
    });
  }
}
```

و ... اگر از آن در یک کد UI استفاده کنیم، ترد آن‌را قفل نکرده و برنامه، پاسخگوی سایر درخواست‌های رسیده خواهد بود. اما ... به این روش اصطلاحا Fake Async گفته می‌شود؛ یا Async تقلبی!

کاری که در اینجا انجام شده، استفاده‌ی ناصحیح از Task.Run در حین طراحی یک متد و یک API است. عملیات انجام شده در آن واقعا غیرهمزمان نیست و در زمان انجام آن، باز هم ترد جدید اختصاص داده شده را تا پایان عملیات قفل می‌کند. اینجا است که باید بین CPU-bound operations و IO-bound operations تفاوت قائل شد. اگر Entity Framework 6 و یا کلاس WebClient و امثال آن، متدهایی Async را نیز ارائه داده‌اند، این‌ها به معنای واقعی کلمه، غیرهمزمان هستند و در آن‌ها کوچکترین CPU-bound operation ایی انجام نمی‌شود.

در حلقه‌ای که در مثال فوق در حال پردازش است و یا تمام اعمال انجام شده توسط CPU، از مرزهای سیستم عبور نمی‌کنیم. نه قرار است فایلی را ذخیره کنیم، نه با اینترنت سر و کار داشته باشیم و یا مثلا اطلاعاتی را از وب سرویسی دریافت کنیم و نه هیچگونه IO-bound operation خاصی قرار است صورت گیرد.

زمانیکه برنامه نویسی قرار است با API شما کار کند و به امضای async Task می‌رسد، فرضش بر این است که در این متد واقعا یک کار غیرهمزمان در حال انجام است. بنابراین جهت بالابردن کارآیی برنامه، این نسخه را نسبت به نمونه‌ی غیرهمزمان انتخاب می‌کند.

حال تصور کنید که استفاده کننده از این API یک برنامه‌ی دسکتاپ نیست، بلکه یک برنامه‌ی ASP.NET است. در اینجا Task.Run فراخوانی شده صرفا سبب خواهد شد عملیات مدنظر، بر روی یک ترد دیگر، نسبت به ترد اصلی اختصاص داده شده توسط ASP.NET برای فراخوانی و پردازش CalculateXYZAsync، صورت گیرد. این عملیات بهینه نیست. تمام پردازش‌های درخواست‌های ASP.NET در تردهای خاص خود انجام می‌شوند. وجود ترد دوم ایجاد شده توسط Task.Run در اینجا چه حاصلی را بجز سوئیچ بی‌جهت بین تردها و همچنین بالا بردن میزان کار Garbage collector دارد؟ در این حالت نه تنها سبب بالا بردن مقیاس پذیری سیستم نشده‌ایم، بلکه میزان کار Garbage collector و همچنین سوئیچ بین تردهای مختلف را در Thread pool برنامه به شدت افزایش داده‌ایم. همچنین یک چنین سیستمی برای تدارک تردهای بیشتر و مدیریت آن‌ها، مصرف حافظه‌ی بیشتری نیز خواهد داشت.

یک اصل مهم در طراحی کدهای Async

استفاده از Task.Run در پیاده سازی بدنه متدهای غیرهمزمان، یک code smell محسوب می‌شود.

چکار باید کرد؟

اگر در کدهای خود اعمال Async واقعی دارید که IO-bound هستند، از معادل‌های Async طراحی شده برای کار با آن‌ها، مانند متد SaveChangesAsync در EF، متد DownloadStringTaskAsync کلاس WebClient و یا متدهای جدید Async کلاس Stream برای خواندن و نوشتن اطلاعات استفاده کنید. در یک چنین حالتی ارائه متدهای async Task بسیار مفید بوده و در جهت بالابردن مقیاس پذیری سیستم بسیار مؤثر واقع خواهند شد.

اما اگر کدهای شما صرفا قرار است بر روی CPU اجرا شوند و تنها محاسباتی هستند، اجازه دهید مصرف کننده تصمیم بگیرد که آیا لازم است از Task.Run برای فراخوانی متد ارائه شده در کدهای خود استفاده کند یا خیر. اگر برنامه‌ی دسکتاپ است، این فراخوانی مفید بوده و سبب آزاد شدن ترد UI می‌شود. اگر برنامه‌ی وب است، به هیچ عنوان نیازی به Task.Run نبوده و فراخوانی متداول آن با توجه به اینکه درخواست‌های برنامه‌های ASP.NET در تردهای مجزایی اجرا می‌شوند، کفایت می‌کند.

به صورت خلاصه

از Task.Run در پیاده سازی بدنه متدهای API خود استفاده نکنید.

از Task.Run در صورت نیاز (مثلا در برنامه‌های دسکتاپ) در حین فراخوانی و استفاده از متدهای API ارائه شده استفاده نمائید:

```
private async void MyButton_Click(object sender, EventArgs e)
{
  await Task.Run(() => myService.CalculateXYZ());
}
```

در این مثال از همان نسخه‌ی غیرهمزمان متد محاسباتی استفاده شده‌است و اینبار مصرف کننده است که تصمیم گرفته در حین فراخوانی و استفاده نهایی، برای آزاد سازی ترد UI از await Task.Run استفاده کند (یا خیر).

بنابراین نوشتن یک چنین کدهایی در پیاده سازی یک API غیرهمزمان

```
await Task.Run(() =>
{
   for (int i = 0; i != 10000000; ++i)
     ;
});
```

صرفا خود را گول زدن است. کل این عملیات بر روی CPU انجام شده و هیچگاه از مرزهای IO سیستم عبور نمی‌کند.

برای مطالعه بیشتر

Should I expose asynchronous wrappers for synchronous methods

عنوان: **جلوگیری از deadlock در برنامه‌های async**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۱ ۱۵:۹
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

توضیح مطلب جاری نیاز به یک مثال دارد. به همین جهت یک برنامه‌ی WinForms یا WPF را آغاز کنید (تفاوتی نمی‌کند). سپس یک دکمه و یک برچسب را در صفحه قرار دهید. در ادامه کدهای فرم را به نحو ذیل تغییر دهید.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows.Forms;
using Newtonsoft.Json.Linq;

namespace Async13
{
    public static class JsonExt
    {
        public static async Task<JObject> GetJsonAsync(this Uri uri)
        {
            using (var client = new HttpClient())
            {
                var jsonString = await client.GetStringAsync(uri);
                return JObject.Parse(jsonString);
            }
        }
    }

    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnGo_Click(object sender, EventArgs e)
        {
            var url =
                "http://api.geonames.org/citiesJSON?north=44.1&south=-9.9&east=-
22.4&west=55.2&lang=de&username=demo";
            txtResult.Text = new Uri(url).GetJsonAsync().Result.ToString();
        }
    }
}
```

این کدها برای کامپایل نیاز به نصب بسته‌ی

```
PM> Install-Package Newtonsoft.Json
```

و همچنین افزودن ارجاعی به اسمبلی استاندارد System.Net.Http نیز دارند.
در اینجا قصد داریم اطلاعات JSON دریافتی را در یک TextBox نمایش دهیم. کاری که انجام شده، فراخوانی متد async ایی است به نام GetJsonAsync و سپس استفاده از خاصیت Result این Task برای صبر کردن تا پایان عملیات.
اگر برنامه را اجرا کنید و بر روی دکمه‌ی دریافت اطلاعات کلیک نمائید، برنامه قفل خواهد کرد. چرا؟
البته تفاوتی هم نمی‌کند که این یک برنامه‌ی دسکتاپ است یا یک برنامه‌ی وب. در هر دو حالت یک deadlock کامل را مشاهده خواهید کرد.

علت بروز deadlock در کدهای async چیست؟

همواره نتیجه‌ی await، در context فراخوان آن بازگشت داده می‌شود. اگر برنامه‌ی دسکتاپ است، این context همان ترد اصلی UI برنامه می‌باشد و اگر برنامه‌ی وب است، این context، زمینه‌ی درخواست در حال پردازش می‌باشد.
خاصیت Result و یا استفاده از متد Wait یک Task، به صورت همزمان عمل می‌کنند و نه غیرهمزمان. متد GetJsonAsync یک Task ناتمام را که فراخوان آن باید جهت پایان‌اش صبر کند، بازگشت می‌دهد. سپس در همینجا کد فراخوان، تردجاری را توسط

فراخوانی خاصیت Result قفل می‌کند. متد GetJsonAsync منتظر خواهد ایستاد تا این ترد آزاد شده و بتواند به کارش که بازگردان نتیجه‌ی عملیات به context جاری است، ادامه دهد.
به عبارتی، کدهای async منتظر پایان کار Result هستند تا نتیجه را بازگردانند. در همین لحظه کدهای همزمان برنامه نیز منتظر کدهای async هستند تا خاتمه یابند. نتیجه‌ی کار یک deadlock است.

روش‌های جلوگیری از deadlock در کدهای async؟

الف) در مورد متد ConfigureAwait در قسمت‌های قبل بحث شد و به عنوان یک best practice مطرح است:

```
public static class JsonExt
    {
        public static async Task<JObject> GetJsonAsync(this Uri uri)
        {
            using (var client = new HttpClient())
            {
                var jsonString = await
client.GetStringAsync(uri).ConfigureAwait(continueOnCapturedContext: false);
                return JObject.Parse(jsonString);
            }
        }
    }
```

با استفاده از ConfigureAwait false سبب خواهیم شد تا نتیجه‌ی عملیات به context جاری بازگشت داده نشود و نتیجه بر روی thread pool thread ادامه یابد. با اعمال این تغییر، کدهای متد btnGo_Click بدون مشکل اجرا خواهند شد.

ب) راه حل دوم، عدم استفاده از خواص و متدهای همزمان با متدهای غیر همزمان است:

```
private async void btnGo_Click(object sender, EventArgs e)
        {
            var url =
                "http://api.geonames.org/citiesJSON?north=44.1&south=-9.9&east=-
22.4&west=55.2&lang=de&username=demo";
            var data = await new Uri(url).GetJsonAsync();
            txtResult.Text = data.ToString();
        }
```

ابتدا امضای متد رویدادگردان را اندکی تغییر داده و واژه‌ی کلیدی async را به آن اضافه می‌کنیم. سپس از await برای صبر کردن تا پایان عملیات متد GetJsonAsync استفاده خواهیم کرد. صبر کردنی که در اینجا انجام شده، یک asynchronous waits است؛ برخلاف روش همزمان استفاده از خاصیت Result یا متد Wait.

خلاصه‌ی بحث

Await را با متدهای همزمان Wait یا خاصیت Result بلاک نکنید. در غیراینصورت در ترد اجرا کننده‌ی دستورات، یک deadlock رخ‌خواهد داد؛ زیرا نتیجه‌ی await باید به context جاری بازگشت داده شود اما این context توسط خواص یا متدهای همزمان فراخوانی شده بعدی، قفل شده‌است.

عنوان: **استفاده از async و await در برنامه‌های ASP.NET Web forms 4.5**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۲ ۱۲:۳۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

سؤال: چه زمانی از متدهای async و چه زمانی از متدهای همزمان بهتر است استفاده شود؟

از متدهای همزمان متداول برای انجام امور ذیل استفاده نمائید:
- جهت پردازش اعمالی ساده و سریع
- اعمال مدنظر بیشتر قرار است بر روی CPU اجرا شوند و از مرزهای IO سیستم عبور نمی‌کنند.

و از متدهای غیرهمزمان برای پردازش موارد زیر کمک بگیرید:
- از وب سرویس‌هایی استفاده می‌کنید که متدهای نگارش async را نیز ارائه داده‌اند.
- عمل مدنظر network-bound و یا I/O-bound است بجای CPU-bound. یعنی از مرزهای IO سیستم عبور می‌کند.
- نیاز است چندین عملیات را به موازات هم اجرا کرد.
- نیاز است مکانیزمی را جهت لغو یک عملیات طولانی ارائه دهید.

مزایای استفاده از متدهای async در ASP.NET

استفاده از await در ASP.NET، ساختار ذاتی پروتکل HTTP را که اساسا یک synchronous protocol، تغییر نمی‌دهد. کلاینت، درخواستی را ارسال می‌کند و باید تا زمان آماده شدن نتیجه و بازگشت آن از طرف سرور، صبر کند. نحوه‌ی تهیه‌ی این نتیجه، خواه async باشد و یا حتی همزمان، از دید مصرف کننده کاملا مخفی است. اکنون سؤال اینجا است که چرا باید از متدهای async استفاده کرد؟
- **پردازش موازی:** می‌توان چند Task را مثلا توسط Task.WhenAll به صورت موازی با هم پردازش کرده و در نهایت نتیجه را سریعتر به مصرف کننده بازگشت داد. اما باید دقت داشت که این Taskها اگر I/O bound باشند، ارزش پردازش موازی را دارند و اگر compute bound باشند (اعمال محاسباتی)، صرفا یک سری ترد را ایجاد و مصرف کرده‌اید که می‌توانسته‌اند به سایر درخواست‌های رسیده پاسخ دهند.
- **خالی کردن تردهای در حال انتظار:** در اعمالی که disk I/O و یا network I/O دارند، پردازش موازی و اعمال async به شدت مقیاس پذیری سیستم را بالا می‌برند. به این ترتیب worker thread جاری (که تعداد آن‌ها محدود است)، سریعتر آزاد شده و به worker pool بازگشت داده می‌شود تا بتواند به یک درخواست دیگر رسیده سرویس دهد. در این حالت می‌توان با منابع کمتری، درخواست‌های بیشتری را پردازش کرد.

ایجاد Asynchronous HTTP Handlers در ASP.Net 4.5

در نگارش‌های پیش از دات نت 4.5، برای نوشتن فایل‌های ashx غیرهمزمان می‌بایستی اینترفیس IHttpAsynchHandler پیاده سازی می‌شد که نحوه‌ی کار با آن از مدل APM پیروی می‌کرد؛ نیاز به استفاده از یک سری callback داشت و این عملیات باید طی دو متد پردازش می‌شد. اما در دات نت 4.5 و با معرفی امکانات async و await، نگارش سازگاری با پیاده سازی کلاس پایه HttpTaskAsyncHandler فراهم شده است.
برای آزمایش آن، یک برنامه‌ی جدید ASP.NET Web forms نگارش 4.5 یا بالاتر را ایجاد کنید. سپس از منوی پروژه، گزینه‌ی Add new item یک Generic handler به نام LogRequestHandler.ashx را به پروژه اضافه نمائید.
زمانیکه این فایل به پروژه اضافه می‌شود، یک چنین امضایی را دارد:

```
public class LogRequestHandler : IHttpHandler
```

IHttpHandler آن‌را اکنون به HttpTaskAsyncHandler تغییر دهید. سپس پیاده سازی ابتدایی آن به شکل زیر خواهد بود:

```
using System;
```

```
using System.Net;
using System.Text;
using System.Threading.Tasks;
using System.Web;

namespace Async14
{
    public class LogRequestHandler : HttpTaskAsyncHandler
    {
        public override async Task ProcessRequestAsync(HttpContext context)
        {
            string url = context.Request.QueryString["rssfeedURL"];
            if (string.IsNullOrWhiteSpace(url))
            {
                context.Response.Write("Rss feed URL is not provided");
            }

            using (var webClient = new WebClient {Encoding = Encoding.UTF8})
            {
                webClient.Headers.Add("User-Agent", "LogRequestHandler 1.0");
                var rssfeed = await webClient.DownloadStringTaskAsync(url);
                context.Response.Write(rssfeed);
            }
        }

        public override bool IsReusable
        {
            get { return true; }
        }

        public override void ProcessRequest(HttpContext context)
        {
            throw new Exception("The ProcessRequest method has no implementation.");
        }
    }
}
```

واژه‌ی کلیدی async را نیز جهت استفاده از await به نسخه‌ی غیرهمزمان آن اضافه کرده‌ایم.
در این مثال آدرس یک فید RSS از طریق کوئری استرینگ rssfeedURL دریافت شده و سپس محتوای آن به کمک متد DownloadStringTaskAsync دریافت و بازگشت داده می‌شود.
برای آزمایش آن، مسیر ذیل را درخواست دهید:

```
http://localhost:4207/LogRequestHandler.ashx?rssfeedURL=http://www.dotnettips.info/feed/latestchanges
```

کاربردهای فایل‌های ashx برای مثال ارائه فیدهای XML ایی یک سایت، ارائه منبع نمایش تصاویر پویا از بانک اطلاعاتی، ارائه JSON برای افزونه‌های auto complete جی‌کوئری و امثال آن است. مزیت آن‌ها سربار بسیار کم است؛ زیرا وارد چرخه‌ی طول عمر یک صفحه‌ی aspx معمولی نمی‌شوند.

صفحات async در ASP.NET 4.5

در قسمت‌های قبل مشاهده کردیم که در برنامه‌های دسکتاپ، به سادگی می‌توان امضای روال‌های رخداد گردان را به async تغییر داد و ... برنامه کار می‌کند. به علاوه از مزیت استفاده از واژه کلیدی await نیز در آن‌ها برخوردار خواهیم شد. اما ... هرچند این روش در وب فرم‌ها نیز صادق است (مثلا public void Page_Load را به public async void Page_Load می‌توان تبدیل کرد) اما اعضای تیم ASP.NET آن‌را در مورد برنامه‌های وب فرم توصیه نمی‌کنند:
Async void event handlers تنها در مورد تعداد کمی از روال‌های رخدادگردان ASP.NET Web forms کار می‌کنند و از آن‌ها تنها برای تدارک پردازش‌های ساده می‌توان استفاده کرد. اگر کار در حال انجام اندکی پیچیدگی دارد، «باید» از PageAsyncTask استفاده نمائید. علت اینجا است که Async void یعنی fire and forget (کاری را شروع کرده و فراموشش کنید). این روش در برنامه‌های دسکتاپ کار می‌کند، زیرا این برنامه‌ها مدل طول عمر متفاوتی داشته و تا زمانیکه برنامه از طرف OS خاتمه نیابد، مشکلی نخواهند داشت. اما برنامه‌های بدون حالت وب متفاوتند. اگر عملیات async پس از خاتمه‌ی طول عمر صفحه پایان یابد، دیگر نمی‌توان اطلاعات صحیحی را به کاربر ارائه داد. بنابراین تا حد ممکن از تعاریف async void در برنامه‌های وب خودداری کنید.

تبدیل روال‌های رخدادگردان متداول وب فرم‌ها به نسخه‌ی async شامل دو مرحله است:
الف) از متد جدید RegisterAsyncTask که در کلاس پایه Page قرار دارد برای تعریف یک PageAsyncTask استفاده کنید:

```
using System;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using System.Web.UI;

namespace Async14
{
    public partial class _default : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            RegisterAsyncTask(new PageAsyncTask(LoadSomeData));
        }

        public async Task LoadSomeData()
        {
            using (var webClient = new WebClient { Encoding = Encoding.UTF8 })
            {
                webClient.Headers.Add("User-Agent", "LogRequest 1.0");
                var rssfeed = await webClient.DownloadStringTaskAsync("url");

                //listcontacts.DataSource = rssfeed;
            }
        }
    }
}
```

با استفاده از System.Web.UI.PageAsyncTask می‌توان یک async Task را در روال‌های رخدادگردان ASP.NET مورد استفاده قرار داد.

ب) سپس در کدهای فایل aspx، نیاز است خاصیت async را نیز true نمائید:

```
<%@ Page Language="C#" AutoEventWireup="true"
Async="true"
  CodeBehind="default.aspx.cs" Inherits="Async14._default" %>
```

تغییر تنظیمات IIS برای بهره بردن از پردازش‌های Async

اگر از ویندوزهای 7، ویستا و یا 8 استفاده می‌کنید، IIS آن‌ها به صورت پیش فرض به 10 درخواست همزمان محدود است.
بنابراین تنظیمات ذیل مرتبط است به یک ویندوز سرور و نه یک work station :
به IIS manager مراجعه کنید. سپس برگه‌ی Application Pools آن‌را باز کرده و بر روی Application pool برنامه خود کلیک راست نمائید. در اینجا گزینه‌ی Advanced Settings را انتخاب کنید. در آن Queue Length را به مثلا عدد 5000 تغییر دهید. همچنین در دات نت 4.5 عدد 5000 برای MaxConcurrentRequestsPerCPU نیز مناسب است. به علاوه عدد
connectionManagement/maxconnection را نیز به 12 برابر تعداد هسته‌های موجود تغییر دهید.

عنوان: **استفاده از async و await در برنامه‌های کنسول و سرویس‌های ویندوز NT**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۸ ۰:۵۸
آدرس: www.dotnettips.info
گروه‌ها: C#, Asynchronous Programming

فرض کنید می‌خواهید از await در متد Main یک برنامه‌ی کنسول به نحو ذیل استفاده کنید:

```
using System;
using System.Net;

namespace Async15
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var webClient = new WebClient { })
            {
                webClient.Headers.Add("User-Agent", "AsyncContext 1.0");
                var data = await webClient.DownloadStringTaskAsync("http://www.dotnettips.info");
                Console.WriteLine(data);
            }
        }
    }
}
```

کامپایلر چنین اجازه‌ای را نمی‌دهد. زیرا از await جایی می‌توان استفاده کرد که متد فراخوان آن با async مزین شده باشد و همچنین دارای یک Context باشد تا نتیجه را بتواند دریافت کند. اگر در اینجا سعی کنید async را به امضای متد Main اضافه نمائید، کامپایلر مجددا خطای an entry point cannot be marked with the 'async' modifier را صادر می‌کند.
اضافه کردن واژه‌ی کلیدی async به روال‌های رخدادگردان void برنامه‌های دسکتاپ مجاز است؛ با توجه به اینکه متد async پیش از پایان کار به فراخوان بازگشت داده می‌شوند (ذات متدهای async به این نحو است). در برنامه‌های دسکتاپ، این بازگشت به UI event loop است؛ بنابراین برنامه بدون مشکل به کار خود ادامه خواهد داد. اما در اینجا، بازگشت متد Main، به معنای بازگشت به OS است و خاتمه‌ی برنامه. به همین جهت کامپایلر از async کردن آن ممانعت می‌کند.
برای حل این مشکل در برنامه‌های کنسول و همچنین برنامه‌های سرویس ویندوز NT که دارای یک async-compatible context نیستند، می‌توان از یک کتابخانه‌ی کمکی سورس باز به نام Nito AsyncEx استفاده کرد. برای نصب آن دستور ذیل را در کنسول پاورشل نیوگت وارد کنید:

```
PM> Install-Package Nito.AsyncEx
```

پس از نصب برای استفاده از آن خواهیم داشت:

```
using System;
using System.Net;
using Nito.AsyncEx;

namespace Async15
{
    class Program
    {
        static void Main(string[] args)
        {
            AsyncContext.Run(async () =>
            {
                using (var webClient = new WebClient())
                {
                    webClient.Headers.Add("User-Agent", "AsyncContext 1.0");
                    var data = await webClient.DownloadStringTaskAsync("http://www.dotnettips.info");
                    Console.WriteLine(data);
                }
            });
        }
    }
}
```

Context ارائه شده در اینجا برخلاف مثال‌های قسمت‌های قبل، نیازی به فراخوانی متد همزمان Wait و یا خاصیت Result که هر دو از نوع blocking هستند ندارد و یک فراخوانی async واقعی است. همچنین می‌شد یک متد async void را نیز در اینجا برای استفاده از DownloadStringTaskAsync تعریف کرد (تا برنامه کامپایل شود). اما پیشتر عنوان شد که هدف از این نوع متدهای خاص async void صرفا استفاده از آن‌ها در روال‌های رخدادگردان UI هستند. زیرا ماهیت آن‌ها fire and forget است و برای دریافت نتیجه‌ی نهایی به نحوی باید ترد اصلی را قفل کرد. برای مثال در یک برنامه‌ی کنسول متد Console.ReadLine را در انتهای کار فراخوانی کرد. اما با استفاده از AsyncContext.Run نیازی به این کارها نیست.

async lambda

در مثال فوق از یک async lambda، برای فراخوانی استفاده شده است که به همراه دات نت 4.5 ارائه شده‌اند:

```
Action, () => { }
Func<Task>, async () => { await Task.Yield(); }

Func<TResult>, () => { return 13; }
Func<Task<TResult>>, async () => { await Task.Yield(); return 13; }
```

آرگومان متد AsyncContext.Run از نوع Func of Task است. بنابراین برای مقدار دهی inline آن توسط lambda expressions مطابق مثال‌های فوق می‌توان از async lambda استفاده کرد.

روش دوم استفاده از AsyncContext.Run و مقدار دهی Func of Task، تعریف یک متد مستقل async Task دار، به نحو ذیل است:

```
class Program
{
  static async Task<int> AsyncMain()
  {
    ..
  }

  static int Main(string[] args)
  {
    return AsyncContext.Run(AsyncMain);
  }
}
```

رخدادهای مرتبط با طول عمر برنامه را async تعریف نکنید

همانند متد Main که async تعریف کردن آن سبب بازگشت آنی روال کار به OS می‌شود و برنامه خاتمه می‌یابد، روال‌های رخدادگردانی که با طول عمر یک برنامه‌ی UI سر و کار دارند مانند Application_Launching، Application_Closing، Application_Activated و Application_Deactivated (خصوصا در برنامه‌های ویندوز 8) نیز نباید async void تعریف شوند (چون مطابق ذات متدهای async، بلافاصله به برنامه اعلام می‌کنند که کار تمام شد). در این موارد خاص نیز می‌توان از متد AsyncContext.Run برای انجام اعمال async استفاده کرد.