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;
        }
    }
}
```

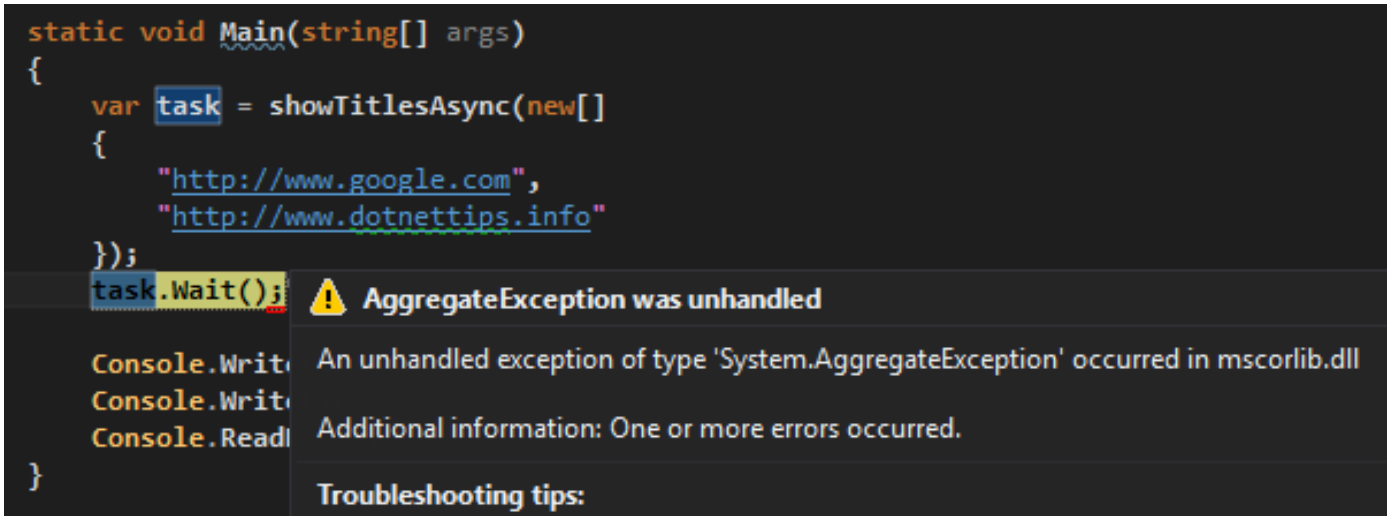
در این مثال یکی از روش‌های استفاده از متدهای async را در یک برنامه‌ی کنسول مشاهده می‌کنید. هر چند خروجی متد doSomethingAsync از نوع Task of int است، اما مستقیما یک int بازگشت داده شده است. تبدیلات نهایی در اینجا توسط کامپایلر انجام می‌شود. همچنین نحوه‌ی استفاده از خاصیت Result را نیز در متد Main مشاهده 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

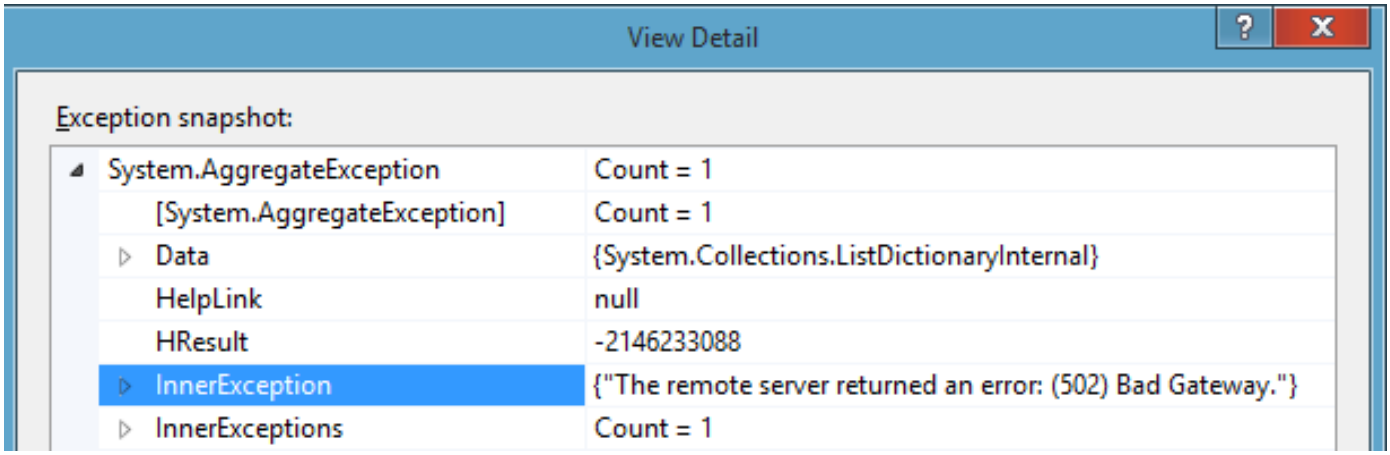
در اینجا توسط متد getTitleAsync، اطلاعات یک صفحه‌ی وب به صورت async دریافت شده و سپس عنوان آن استخراج می‌شود. در متد showTitlesAsync نیز از آن استفاده شده و در طی یک حلقه، چندین وب سایت مورد بررسی قرار خواهند گرفت. چون متد getTitleAsync از نوع async تعریف شده‌است، فراخوان آن نیز باید async تعریف شود 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.