

در این مطلب می‌خواهم روش استفاده از Async&Await رو براتون بگم. Async&Await خط و مشی جدید Microsoft برای تولید متدهای Async هستش که نوشتن این متدها رو خیلی جذاب کرده و کاربردهای خیلی زیادی هم داره. مثلاً هنگام استفاده از Web Api در برنامه‌های تحت ویندوز نظیر WPF این روش خیلی به ما کمک می‌کنه و در کل نوشتن Parallel Programming را خیلی جالب کرده.

برای اینکه بتونم قدرت و راحتی کار با این ابزار رو به خوبی نشون بدم ابتدا یک مثال رو به روشی قدیمی‌تر پیاده سازی می‌کنم. بعد پیاده سازی همین مثال رو به روش جدید بهتون نشون می‌دم.

می‌خواهم یک برنامه بنویسم که لیستی از محصولات رو به صورت Async در خروجی چاپ کنه. ابتدا کلاس مدل:

```
public class Product
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

حالا کلاس ProductService رو می‌نویسم:

```
public class ProductService
{
    public ProductService()
    {
        ListOfProducts = new List<Product>();
    }

    public List<Product> ListOfProducts
    {
        get;
        private set;
    }

    private void InitializeList( int toExclusive )
    {
        Parallel.For( 0 , toExclusive , ( int counter ) =>
        {
            ListOfProducts.Add( new Product()
            {
                Id = counter ,
                Name = "DefaultName" + counter.ToString()
            } );
        } );
    }

    public IAsyncResult BeginGetAll( AsyncCallback callback , object state )
    {
        var myTask = Task.Run<IEnumerable<Product>>( () =>
        {
            InitializeList( 100 );
            return ListOfProducts;
        } );
        return myTask.ContinueWith( x => callback( x ) );
    }

    public IEnumerable<Product> EndGetAll( IAsyncResult result )
    {
        return ( ( Task<IEnumerable<Product>> )result ).Result;
    }
}
```

در کلاس بالا دو متد مهم دارم. متد اول آن BeginGetAll است و همونطور که می‌بینید خروجی اون از نوع IAsyncResult است و باید هنگام استفاده، اونو به متد EndGetAll پاس بدم تا خروجی مورد نظر به دست بیاد. متد InitializeList به تعداد ورودی آیتم به لیست اضافه می‌کند و اونو به Callback می‌فرسته. در نهایت برای اینکه بتونم از این

کلاس ها استفاده کنیم باید به صورت زیر عمل بشه:

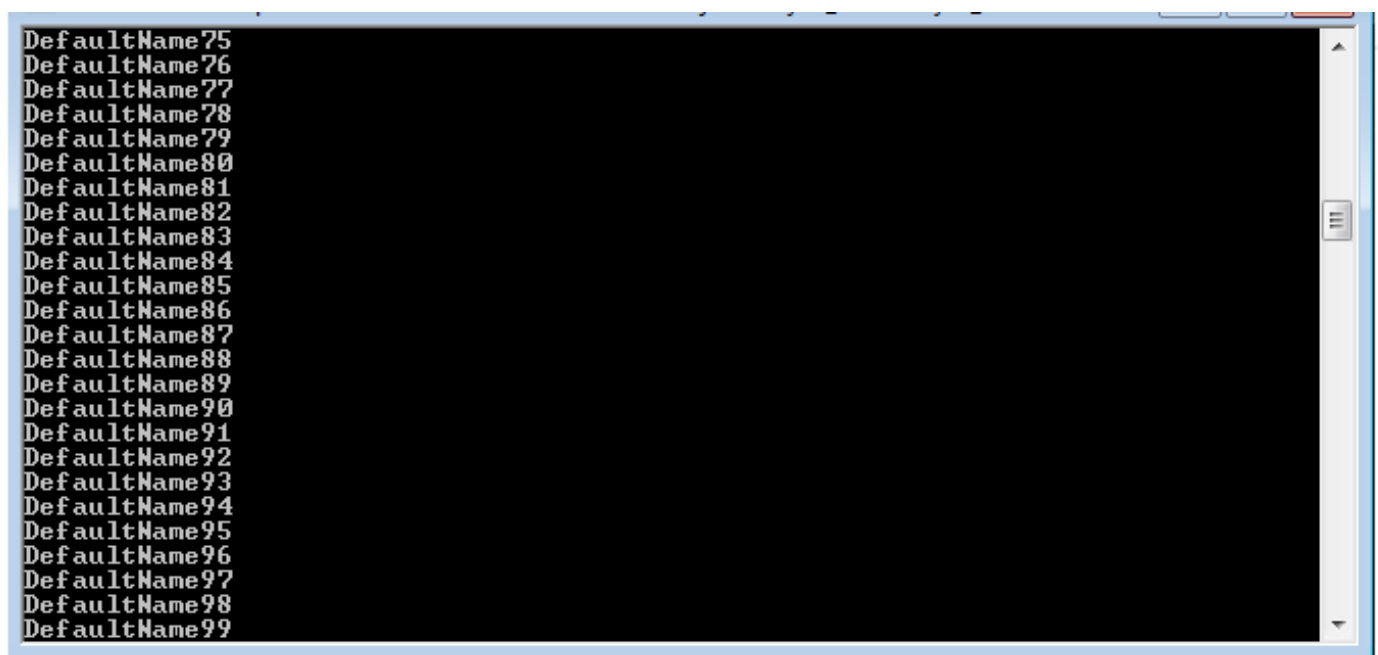
```
class Program
{
    static void Main( string[] args )
    {
        GetAllProducts().ToList().ForEach( ( Product item ) =>
        {
            Console.WriteLine( item.Name );
        } );

        Console.ReadLine();
    }

    public static IEnumerable<Product> GetAllProducts()
    {
        ProductService service = new ProductService();

        var output = Task.Factory.FromAsync<IEnumerable<Product>>( service.BeginGetAll ,
service.EndGetAll , TaskCreationOptions.None );
        return output.Result;
    }
}
```

خیلی راحت بود؛ درسته. خروجی مورد نظر رو می بینید:



```
DefaultName75
DefaultName76
DefaultName77
DefaultName78
DefaultName79
DefaultName80
DefaultName81
DefaultName82
DefaultName83
DefaultName84
DefaultName85
DefaultName86
DefaultName87
DefaultName88
DefaultName89
DefaultName90
DefaultName91
DefaultName92
DefaultName93
DefaultName94
DefaultName95
DefaultName96
DefaultName97
DefaultName98
DefaultName99
```

حالا همین کلاس بالا رو به روش Async&Await می نویسم:

```
public async Task<IEnumerable<Product>> GetAllAsync()
{
    var result = Task.Run( () =>
    {
        InitializeList( 100 );
        return ListOfProducts;
    } );
    return await result;
}
```

در متد بالا به جای استفاده از 2 متد از یک متد GetAllAsync استفاده کردم که خروجی آون از نوع async

Task<IEnumerable<Product>> بود و برای استفاده از اون کافیه در کلاس Program کد زیر رو بنویسم

```
class Program
{
    static void Main( string[] args )
    {
        GetAllProducts().Result.ToList().ForEach( ( Product item ) =>
        {
            Console.WriteLine( item.Name );
        } );

        Console.ReadLine();
    }

    public static async Task<IEnumerable<Product>> GetAllProducts()
    {
        ProductService service = new ProductService();

        return await service.GetAllAsync();
    }
}
```

فکر کنم همتون موافقید که روش Async&Await هم از نظر نوع کد نویسی و هم از نظر راحتی کار خیلی سرتیره. یکی از مزایای مهم این روش اینه که همین مراحل رو می تونید در هنگام استفاده از WCF در پروژه تکرار کنید. به خوبی کار می کنه.

عنوان:	دریافت زمانبندی شده به روز رسانی‌های آنتی ویروس Symantec به کمک کتابخانه‌های Quartz.NET و Html Agility Pack
نویسنده:	سیروان عقیقی
تاریخ:	۸:۳۰ ۱۳۹۲/۰۴/۰۳
آدرس:	www.dotnettips.info
برچسب‌ها:	Regular expressions, Quartz.NET, Html Agility Pack, Asynchronous Programming

در این رابطه آقای راد در دو قسمت به صورت مختصر و مفید این کتابخانه قدرتمند رو همراه با ارائه چندین مثال کاربردی معرفی کردند:

[قسمت اول](#)

[قسمت دوم](#)

در تکمیل قسمت‌های فوق بنده می‌خواهم مثالی رو در این رابطه براتون بذارم، هدف از ارائه این مثال اتوماتیک سازی یک فرآیند روتین می‌باشد، به این صورت که در جایی که بنده مشغول به کار هستم یک سری لایسنس آنتی ویروس برای کلاینت‌ها در یک شبکه با مقیاس متوسط تهیه گردیده است، حال یک نسخه رایگان نیز برای کاربرانی که قصد دارند آنتی ویروس را برای سیستم شخصی خود نصب کنند نیز موجود می‌باشد که نیاز به آپدیت دارد معمولاً آپدیت‌ها هر چند روز یکبار یا هر هفته در دو نسخه 64 و 32 بیتی ارائه می‌شوند، روال معمول برای دریافت آپدیت مراجعه به سایت و دانلود نسخه‌های مربوطه می‌باشد.

حال توسط کتابخانه قدرتمند [Quartz.NET](#) این فرآیند روتین را به صورت اتوماتیک می‌خواهیم انجام دهیم، استفاده از کتابخانه ذکر شده سخت نیست همانطور که در دو مطلب قبلی مرتبط ذکر گردیده، تنها پیاده سازی چندین اینترفیس است و بس.

```
namespace SymantecUpdateDownloader
{
    using System;
    using System.IO;
    using Quartz;
    using Quartz.Impl;
    using System.Globalization;
    public class TestJob : IJob
    {
        public void Execute(IJobExecutionContext context)
        {
            new Download().Scraping();
        }
    }
    public interface ISchedule
    {
        void Run();
    }
    public class TestSchedule : ISchedule
    {
        public void Run()
        {
            DateTimeOffset startTime = DateBuilder.FutureDate(2, IntervalUnit.Second);

            IJobDetail job = JobBuilder.Create<HelloJob>()
                .WithIdentity("job1")
                .Build();

            ITrigger trigger = TriggerBuilder.Create()
                .WithIdentity("trigger1")
                .StartAt(startTime)
                .WithDailyTimeIntervalSchedule(x =>
                    x.OnEveryDay().StartingDailyAt(new TimeOfDay(7, 0)).WithRepeatCount(0))
                .Build();

            ISchedulerFactory sf = new StdSchedulerFactory();
            IScheduler sc = sf.GetScheduler();
            sc.ScheduleJob(job, trigger);

            sc.Start();
        }
    }
}
```

در این کد که همانند کدهای پیشنهادی مطلب است، در خط 33 از متد `WithDailyTimeIntervalSchedule` استفاده شده است

و همانطور که مشخص است وظیفه تعیین شده و هر روز ساعت 7 اجرا میشود.

مورد بعدی عملیات دانلود فایل می‌باشد که در ادامه مشاهده خواهید کرد، [صفحه ایی](#) که لینک فایل‌های دانلود را ارائه داده است دو نسخه مد نظر ما را در ابتدا لیست کرده است و با استفاده از web scraping می‌توانیم موارد تعیین شده را استخراج کنیم برای این منظور از کتابخانه [htmlagilitypack](#) استفاده میکنیم، تطبیق دو مورد (لینک) اول جهت دریافت نسخه‌های 32 و 64 بیتی به کمک Regular Expression میسر است و همانطور که در شکل زیر مشاهده میکنید از سمت چپ تاریخ به صورت 8 رقم، سه رقم قسمت دوم و ارقام و حروف قسمت سوم است به اضافه پسوند فایل مشخص است :



```
public class Download
{
    static WebClient wc = new WebClient();
    static ManualResetEvent handle = new ManualResetEvent(true);

    private DateTime myDate = new DateTime();
    public void Scraping()
    {
        using (WebClient client = new WebClient())
        {
            client.Encoding = System.Text.Encoding.UTF8;
            var doc = new HtmlAgilityPack.HtmlDocument();
            ArrayList result = new ArrayList();

            doc.LoadHtml(client.DownloadString("https://www.symantec.com/security_response/definitions/download/detail.jsp?gid=savce"));
            var tasks = new List<Task>();
            foreach (var href in doc.DocumentNode.Descendants("a").Select(x =>
                x.Attributes["href"]))
            {
                if (href == null) continue;
                string s = href.Value;
                Match m = Regex.Match(s, @"http://definitions.symantec.com/defs/(\d{8}-\d{3}-v5i(32|64)\.exe)");
                if (m.Success)
                {
                    Match date = Regex.Match(m.Value, @"(\d{4})(\d{2})(\d{2})");
                    Match filename = Regex.Match(m.Value, @"(\d{8}-\d{3}-v5i(32|64)\.exe)");
                    int year = Int32.Parse(date.Groups[0].Value);
                    int month = Int32.Parse(date.Groups[1].Value);
                    int day = Int32.Parse(date.Groups[2].Value);

                    myDate = new DateTime(
                        Int32.Parse(date.Groups[1].Value),
                        Int32.Parse(date.Groups[2].Value),
                        Int32.Parse(date.Groups[3].Value));
                    if (myDate == DateTime.Today)
                    {
                        tasks.Add(DownloadUpdate(m.Value, filename.Value));
                    }
                    else
                    {
                        MessageBox.Show("امروز آپدیت موجود نیست");
                    }
                }
            }
            DownloadTask = Task.WhenAll(tasks);
        }
    }
    private static Task DownloadTask;
    private Task DownloadUpdate(string url, string fileName)
    {
        var wc = new WebClient();
        return wc.DownloadFileTaskAsync(new Uri(url), @"\\10.1.0.15\SymantecUpdate\\" + fileName);
    }
}
```

```
}
```

توضیح کدهای فوق :

ابتدا توسط متد LoadHtml خط 14 صفحه مورد نظر که حاوی لینک‌ها می‌باشد رو Load میکنیم، سپس توسط یک حلقه foreach خط 16 مقدار خصوصیت href تمام لینک‌های موجود در صفحه را استخراج میکنیم مثلا مقدار خصوصیت href در لینک‌ها به صورت زیر می‌باشد :

```
http://definitions.symantec.com/defs/20130622-007-v5i32.exe
```

```
http://definitions.symantec.com/defs/20130622-007-v5i64.exe
```

همانطور که مشخص است در دو مورد فوق تنها نام فایل متفاوت می‌باشد، همانطور که بحث شد برای نام فایل‌ها هم می‌توانیم یک Pattern را به صورت زیر داشته باشیم :

```
(\d{8}-\d{3}-v5i(32|64)\.exe)
```

در خط 20 نیز عملیات تطبیق تمام hrefهای موجود در صفحه را توسط Regular Expression فوق تطبیق می‌دهیم، اگر تطبیق با موفقیت انجام پذیرفت باید نام فایل و همچنین تاریخ موجود در نام فایل را نیز توسط دو Regular Expression استخراج کنیم(خط 23 و 24) در ادامه برای جدا کردن مقادیر سال ، ماه ، روز از امکان Groups در RegEx استفاده کرده ایم:

```
int year = Int32.Parse(date.Groups[0].Value);  
int month = Int32.Parse(date.Groups[1].Value);  
int day = Int32.Parse(date.Groups[3].Value);
```

در ادامه تاریخ استخراج شده را با تاریخ روز جاری مقایسه می‌کنیم اگر مساوی بود عملیات دانلود فایل‌ها توسط یک [Task](#) تعریف شده به صورت [همزمان](#) بر روی سرور مربوطه دانلود می‌شوند. البته لازم به ذکر است که کدهای فوق مسلما نیاز به Refactoring دارند منتها هدف از ارائه این مثال آشنایی بیشتر با کتابخانه‌های فوق می‌باشد.

نکته آخر اینکه برنامه فوق به حالت‌های مختلفی می‌تواند اجرا گردد مثل یک برنامه وب یا یک سرویس ویندوزی و ... ، بهترین حالت یک سرویس ویندوز می‌باشد، ولی در حالت خام در حال حاضر یک ویندوز اپلیکیشن ساده می‌باشد که بر روی سرور RUN شده است که در آینده به صورت یک سرویس ویندوز ارائه خواهد شد.

نظرات خوانندگان

نویسنده: افشین

تاریخ: ۱۳۹۲/۰۴/۱۵ ۸:۱

یه سؤال دارم که همیشه ذهنم رو مشغول کرده
مگه اینترفیس فقط امضا روال‌ها رو نداره؟ پس یک کلاس نیاز داره که بتونه اون متدها رو پیاده سازی کنه و ما ازش استفاده کنیم
غیر از اینه؟
پس در کد زیر

```
IJobDetail job = JobBuilder.Create<HelloJob>()
```

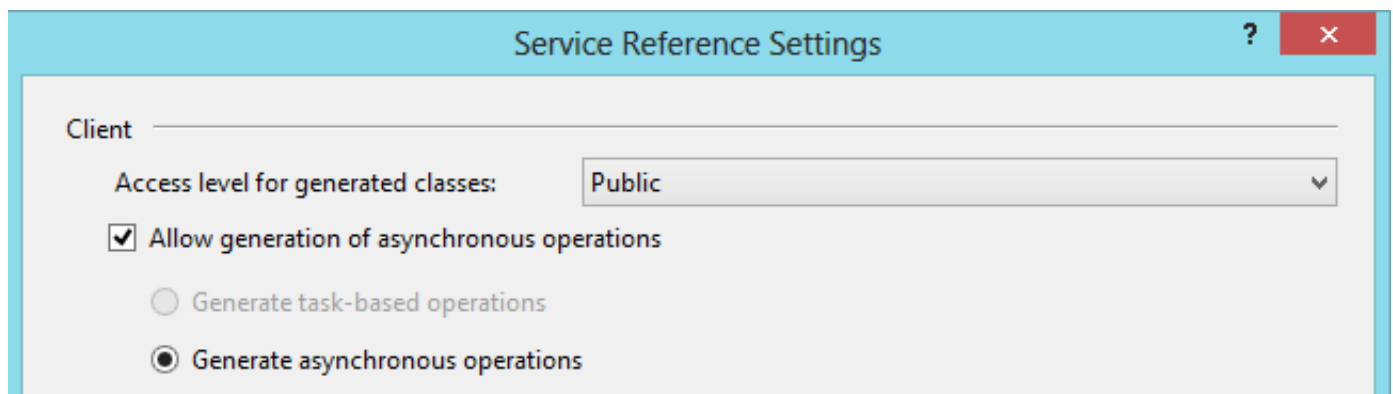
مجبوریم از اینترفیس به عنوان متغیر استفاده کنیم؟

نویسنده: سیروان عقیفی

تاریخ: ۱۳۹۲/۰۴/۱۵ ۱۲:۴۲

بله به همین صورته، این مطلب رو درباره [اینترفیس](#) و این مطلب رو درباره متدهای [Generic](#) بخونید،
متد Create یک متد Generic است که نام کلاسی رو که اینترفیس IJob و Implement کرده را قبول میکند، و در نهایت مقدار بازگشتی این متد از نوع IJobDetail است.

هنگام تولید و توسعه سیستم های مبتنی بر WCF حتما نیاز به سرویس هایی داریم که متدها را به صورت Async اجرا کنند. در دات نت 4.5 از Async&Await استفاده می کنیم (^). ولی در پروژه هایی که تحت دات نت 4 هستند این امکان وجود ندارد (البته می توانید Async&Await CTP رو برای دات نت 4 هم نصب کنید (^)). فرض کنید پروژه ای داریم تحت دات نت 3.5 یا 4 و قصد داریم یکی از متدهای سرویس WCF آن را به صورت Async پیاده سازی کنیم. ساده ترین روش این است که هنگام Add Service Reference از پنجره Advanced به صورت زیر عمل کنیم:



مهم ترین عیب این روش این است که در این حالت تمام متدهای این سرویس رو هم به صورت Sync و هم به صورت Async تولید می کنه در حالی که ما فقط نیاز به یک متد Async داریم .

در این پست قصد دارم پیاده سازی این متد رو بدون استفاده از Async&Await و Code Generation توکار دات نت شرح بدم که با دات نت 3.5 هم سازگار است.

ابتدا یک پروژه از نوع WCF Service Application ایجاد کنید.

یک ClassLibrary جدید به نام Model بسازید و کلاس زیر را به عنوان مدل در آن قرار دهید. (این اسمبلی باید هم به پروژه های کلاینت و هم به پروژه های سرور رفرنس داده شود)

```
[DataContract]
public class Book
{
    [DataMember]
    public int Code { get; set; }

    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public string Author { get; set; }
}
```

حال پیاده سازی سرویس و Contract مربوطه را شروع می کنیم.

Class Library به نام Contract بسازید. قصد داریم از این لایه به عنوان قراردادهای سمت کلاینت و سرور استفاده کنیم. اینترفیس زیر را به عنوان BookContract در آن بسازید.


```
[ServiceContract]
public interface IBookService
{
    [OperationContract( AsyncPattern = true )]
    IAsyncResult BeginGetAllBook( AsyncCallback callback, object state );

    IEnumerable<Book> EndGetAllBook( IAsyncResult asyncResult );
}
```

برای پیاده سازی متدهای Async به این روش باید دو متد داشته باشیم. یکی به عنوان شروع عملیات و دیگری اتمام. دقت کنید نام گذاری به صورت Begin و End کاملاً اختیاری است و برای خوانایی بهتر از این روش نام گذاری استفاده می‌کنم. متدی که به عنوان شروع عملیات استفاده می‌شود باید حتماً OperationContractAttribute رو داشته باشد و مقدار خاصیت AsyncPattern اون هم true باشد. همان طور که می‌بیند این متد دارای 2 آرگومان ورودی است. یکی از نوع AsyncCallback و دیگری از نوع object. تمام متدهای Async به این روش باید این دو آرگومان ورودی را حتماً داشته باشند. خروجی این متد حتماً باید از نوع IAsyncResult باشد. متد دوم که به عنوان اتمام عملیات استفاده می‌شود نباید OperationContractAttribute را داشته باشد. ورودی اون هم فقط یک آرگومان از نوع IAsyncResult است. خروجی اون هم هر نوعی که سمت کلاینت احتیاج دارید می‌تونه باشه. در صورت عدم رعایت نکات فوق، هنگام ساخت ChannelFactory یا خطا روبرو خواهید شد. اگر نیاز به پارامتر دیگری هم داشتید باید آن‌ها را قبل از این دو پارامتر قرار دهید. برای مثال:

```
[OperationContract]
IEnumerable<Book> GetAllBook(int code , AsyncCallback callback, object state );
```

قبل از پیاده سازی سرویس باید ابتدا یک AsyncResult سفارشی بسازیم. ساخت AsyncResult سفارشی بسیار ساده است. کافی است کلاسی بسازیم که اینترفیس IAsyncResult را به ارث ببرد.

```
public class CompletedAsyncResult<TEntity> : IAsyncResult where TEntity : class , new()
{
    public IList<TEntity> Result
    {
        get
        {
            return _result;
        }
        set
        {
            _result = value;
        }
    }
    private IList<TEntity> _result;

    public CompletedAsyncResult( IList<TEntity> data )
    {
        this.Result = data;
    }

    public object AsyncState
    {
        get
        {
            return ( IList<TEntity> )Result;
        }
    }

    public WaitHandle AsyncWaitHandle
    {
        get
        {
            throw new NotImplementedException();
        }
    }

    public bool CompletedSynchronously
    {
        get
        {
            return true;
        }
    }
}
```

```

    public bool IsCompleted
    {
        get
        {
            return true;
        }
    }
}

```

در کلاس بالا یک خاصیت به نام Result در نظر گرفتیم که لیستی از نوع TEntity است. TEntity به صورت generic تعریف شده و نوع ورودی آن هر نوع کلاس غیر abstract می تواند باشد. این کلاس برای تمام سرویس های Async یک پروژه مورد استفاده قرار خواهد گرفت برای همین ورودی آن به صورت generic در نظر گرفته شده است.

#پیاده سازی سرویس

```

public class BookService : IBookService
{
    public BookService()
    {
        ListOfBook = new List<Book>();
    }

    public List<Book> ListOfBook
    {
        get;
        private set;
    }

    private List<Book> CreateListOfBook()
    {
        Parallel.For( 0, 10000, ( int counter ) =>
        {
            ListOfBook.Add( new Book()
            {
                Code = counter,
                Title = String.Format( "Book {0}", counter ),
                Author = "Masoud Pakdel"
            } );
        } );

        return ListOfBook;
    }

    public IAsyncResult BeginGetAllBook( AsyncCallback callback, object state )
    {
        var result = CreateListOfBook();
        return new CompletedAsyncResult<Book>( result );
    }

    public IEnumerable<Book> EndGetAllBook( IAsyncResult asyncResult )
    {
        return ( ( CompletedAsyncResult<Book> )asyncResult ).Result;
    }
}

```

*در متد BeginGetAllBook ابتدا به تعداد 10,000 کتاب در یک لیست ساخته می شوند و بعد این لیست در کلاس CompletedAsyncResult که ساختیم به عنوان ورودی سازنده پاس داده می شوند. چون CompletedAsyncResult ارث برده است پس return آن به عنوان خروجی مانعی ندارد. با فراخوانی متد EndGetAllBook سمت کلاینت مقدار asyncResult به عنوان خروجی برگشت داده می شود. به عملیات casting برای دستیابی به مقدار Result در CompletedAsyncResult دقت کنید.

#کدهای سمت کلاینت:

اکثر برنامه نویسان با استفاده از روش AddServiceReference یک سرویس کلاینت در اختیار خواهند داشت که با وهله سازی از این کلاس یک ChannelFactory ایجاد می شود. در این پست به جای استفاده از Code Generation توکار دات نت برای ساخت ChannelFactory از روش دیگری استفاده خواهیم کرد. به عنوان برنامه نویس باید بدانیم که در پشت پرده عملیات ساخت ChannelFactory چگونه است.

روش AddServiceReference

بعد از اضافه شدن سرویس سمت کلاینت کدهای زیر برای سرویس Book به صورت زیر تولید می شود.

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
public partial class BookServiceClient :
System.ServiceModel.ClientBase<UI.BookService.IBookService>, UI.BookService.IBookService {

    public BookServiceClient() {
    }

    public BookServiceClient(string endpointConfigurationName) :
        base(endpointConfigurationName) {
    }

    public BookServiceClient(string endpointConfigurationName, string remoteAddress) :
        base(endpointConfigurationName, remoteAddress) {
    }

    public BookServiceClient(string endpointConfigurationName, System.ServiceModel.EndpointAddress
remoteAddress) :
        base(endpointConfigurationName, remoteAddress) {
    }

    public BookServiceClient(System.ServiceModel.Channels.Binding binding,
System.ServiceModel.EndpointAddress remoteAddress) :
        base(binding, remoteAddress) {
    }

    public UI.BookService.Book[] BeginGetAllBook() {
        return base.Channel.BeginGetAllBook();
    }
}
```

همانطور که می بینید سرویس بالا از کلاس ClientBase ارث برده است. ClientBase دارای خاصیتی به نام ChannelFactory است که فقط خواندنی می باشد. با استفاده از مقادیر EndPointConfiguration یک وهله از کلاس ChannelFactory با توجه به مقدار generic کلاس ClientBase ایجاد خواهد شد. در کد زیر مقدار TChannel برابر IBookService است:

```
System.ServiceModel.ClientBase<UI.BookService.IBookService>
```

وهله سازی از ChannelFactory به صورت دستی

یک پروژه ConsoleApplication سمت کلاینت ایجاد کنید. برای فراخوانی متدهای سرویس سمت سرور باید ابتدا تنظیمات EndPoint رو به درستی انجام دهید. سپس با استفاده از EndPoint به راحتی می توانیم Channel مربوطه را بسازیم. کلاسی به نام ServiceMapper ایجاد می کنیم که وظیفه آن ساخت ChannelFactory به ازای درخواست ها است.

```
public class ServiceMapper<TChannel>
{
    public static TChannel CreateChannel()
    {
        TChannel proxy;

        var endPointAddress = new EndpointAddress( "http://localhost:7000/" + typeof( TChannel
).Name.Remove( 0, 1 ) + ".svc" );

        var httpBinding = new BasicHttpBinding();

        ChannelFactory<TChannel> factory = new ChannelFactory<TChannel>( httpBinding,
endPointAddress );

        proxy = factory.CreateChannel();

        return proxy;
    }
}
```

در متد CreateChannel یک وهله از کلاس EndpointAddress ایجاد شده است. پارامتر ورودی آن آدرس سرویس هاست شده می باشد برای مثال:

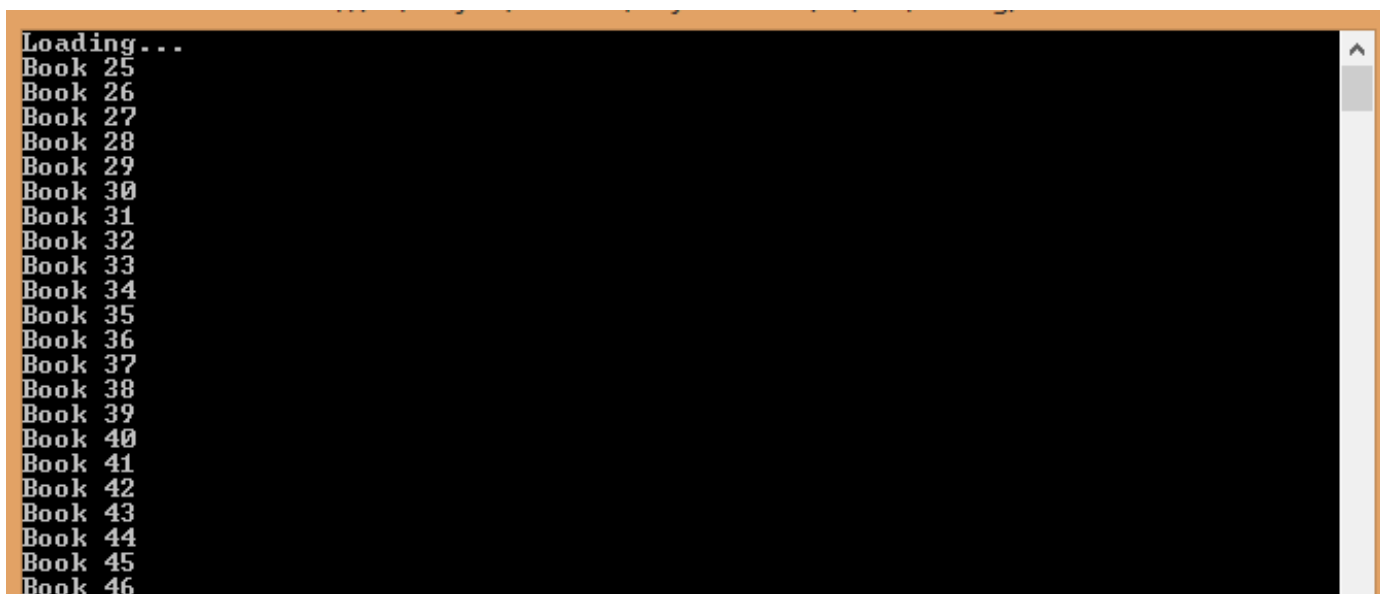
```
"http://localhost:7000/" + "BookService.svc"
```

دستور Remove برای حذف I از ابتدای نام سرویس است. پارامترهای ورودی برای سازنده کلاس ChannelFactory ابتدا یک نمونه از کلاس BasicHttpBinding می باشد. می توان از WSHttpBinding یا NetTcpBinding یا WSDLHttpBinding هم استفاده کرد. البته هر کدام از انواع Binding ها تنظیمات خاص خود را می طلبد که در مقاله ای جداگانه بررسی خواهیم کرد. پارامتر دوم هم EndPoint ساخته شده می باشد. در نهایت با دستور CreateChannel عملیات ساخت Channel به پایان می رسد.

بعد از اعمال تغییرات زیر در فایل Program پروژه Console و اجرای آن، خروجی به صورت زیر می باشد.

```
var channel = ServiceMapper<Contract.IBookService>.CreateChannel();
channel.BeginGetAllBook( new AsyncCallback( ( asyncResult ) =>
{
    channel.EndGetAllBook( asyncResult ).ToList().ForEach( _record =>
    {
        Console.WriteLine( _record.Title );
    } );
} ) , null );
Console.WriteLine( "Loading..." );
Console.ReadLine();
```

همان طور که می بینید ورودی متد BeginGtAllBook یک AsyncCallback است که در داخل آن متد EndGetAllBook صدا زده شده است. مقدار برگشتی متد EndGetAllBook خروجی مورد نظر ماست. خروجی :



```
Loading...
Book 25
Book 26
Book 27
Book 28
Book 29
Book 30
Book 31
Book 32
Book 33
Book 34
Book 35
Book 36
Book 37
Book 38
Book 39
Book 40
Book 41
Book 42
Book 43
Book 44
Book 45
Book 46
```

نکته: برای اینکه مطمئن شوید که سرویس مورد نظر در آدرس "http://localhost:7000" هاست شده است (یعنی همان آدرسی که در EndPointAddress از آن استفاده کردیم) کافیست از پنجره Project Properties برای پروژه سرویس وارد برگه Web شده و از بخش Servers گزینه Use Visual Studio Development Server و Specific Port 7000 رو انتخاب کنید.

نظرات خوانندگان

نویسنده: هیمن روحانی
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۲:۳۱

سلام؛ من این مثال رو با دات نت 3.5 تست کردم. سمت کلاینت، channel فقط یک تابع به نام GetAllBook دارد و دو تابعی که در سرویس تعریف شده اند نمایش داده نمی‌شود؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۳:۳۷

اگر از Add Service Reference برای اضافه کردن سرویس به کلاینت استفاده کردید باید از قسمت Advanced حتما تیک مربوط به Allow generation of asynchronous operation رو بزنید. ساخت متدهای Async در این روش به عهده code generator توکار دات است.

نویسنده: هیمن روحانی
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۳:۴۸

جدا از روش Add Service Reference چه روش دیگه‌ای هست؟ پس تابع CreateFactory فقط از روی همین Service Reference یک نمونه می‌سازه؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۴:۱۶

اگر از روش [ChannelFactory](#) استفاده کنید به دلیل دسترسی مستقیم به اسمبلی Service Contract تمام Operation Contract های تعریف شده در هر سرویس در دسترس خواهد بود. تابع CreateChannel با استفاده از تنظیمات Binding و EndpointAddress یک کانال به سرویس مربوطه خواهد ساخت و هیچ گونه نیازی به Add service reference در این روش نیست.

نویسنده: هیمن روحانی
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۵:۰۲

یعنی برای استفاده از [ChannelFactory](#) باید به پروژه کلاینت اسمبلی Service Contract رو به عنوان reference اضافه کنم؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۶:۱۸

بله. فقط به این نکته دقت داشته باشید که منظور از ServiceContract یعنی پروژه ای که فقط شامل اینترفیس هایی است که ServiceContractAttribute رو دارند. پیاده سازی این اینترفیس ها باید در یک پروژه دیگر برای مثال Service باشد که هیچ گونه رفرنسی به آن نیز نباید داده شود.

[Reactive extensions](#) یا به صورت خلاصه Rx، کتابخانه‌ی [سورس باز](#) تهیه شده‌ای توسط مایکروسافت است که اگر بخواهیم آن را به ساده‌ترین شکل ممکن تعریف کنیم، معنای Linq to events را می‌دهد و امکان مدیریت تعامل‌های پیچیده‌ی async را به صورت declaratively فراهم می‌کند. هدف آن بسط فضای نام System.Linq و تبدیل نتایج یک کوئری LINQ به یک مجموعه‌ی Observable است؛ به همراه مدیریت مسایل همزمانی آن. این افزونه جزو موفق‌ترین کتابخانه‌های دات نت مایکروسافت در سال‌های اخیر به شما می‌رود؛ تا حدی که معادل‌های بسیاری از آن برای زبان‌های دیگر مانند Java، JavaScript، Python، CPP و غیره نیز تهیه شده‌اند.

استفاده از Rx به همراه یک کوئری LINQ

یک برنامه‌ی کنسول جدید را ایجاد کنید. سپس برای نصب کتابخانه‌ی Rx، دستور ذیل را در کنسول پاورشل [نیوگت](#) اجرا نمایید:

```
PM> Install-Package Rx-Main
```

نصب آن از طریق نیوگت، به صورت خودکار کلیه وابستگی‌های مرتبط با آن را نیز به پروژه‌ی جاری اضافه می‌کند:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Rx-Core" version="2.2.4" targetFramework="net45" />
  <package id="Rx-Interfaces" version="2.2.4" targetFramework="net45" />
  <package id="Rx-Linq" version="2.2.4" targetFramework="net45" />
  <package id="Rx-Main" version="2.2.4" targetFramework="net45" />
  <package id="Rx-PlatformServices" version="2.2.4" targetFramework="net45" />
</packages>
```

سپس متد Main این برنامه را به نحو ذیل تغییر دهید:

```
using System;
using System.Linq;

namespace Rx01
{
    class Program
    {
        static void Main(string[] args)
        {
            var query = Enumerable.Range(1, 5).Select(number => number);
            foreach (var number in query)
            {
                Console.WriteLine(number);
            }
            finished();
        }

        private static void finished()
        {
            Console.WriteLine("Done!");
        }
    }
}
```

در اینجا یک سری عملیات متداول را مشاهده می‌کنید. بازه‌ای از اعداد توسط متد Enumerable.Range ایجاد شده و سپس به کمک یک حلقه، تمام آیتم‌های آن نمایش داده می‌شوند. همچنین در پایان کار نیز یک متد دیگر فراخوانی شده‌است. اکنون اگر بخواهیم همین عملیات را توسط Rx انجام دهیم، به شکل زیر خواهد بود:

```
using System;
```

```
using System.Linq;
using System.Reactive.Linq;

namespace Rx01
{
    class Program
    {
        static void Main(string[] args)
        {
            var query = Enumerable.Range(1, 5).Select(number => number);
            var observableQuery = query.ToObservable();
            observableQuery.Subscribe(onNext: number => Console.WriteLine(number), onCompleted: () =>
finished());
        }

        private static void finished()
        {
            Console.WriteLine("Done!");
        }
    }
}
```

ابتدا نیاز است تا کوثری متداول LINQ را تبدیل به نمونه‌ی Observable آن کرد. اینکار را توسط متد الحاقی ToObservable که در فضای نام System.Reactive.Linq تعریف شده‌است، انجام می‌دهیم. به این ترتیب، هر زمانیکه که عددی به query اضافه می‌شود، با استفاده از متد Subscribe می‌توان تغییرات آن را تحت کنترل قرار داد. برای مثال در اینجا هر بار که عددی در بازه‌ی 1 تا 5 تولید می‌شود، یکبار پارامتر onNext اجرا خواهد شد. برای نمونه در مثال فوق، از نتیجه‌ی آن برای نمایش مقدار دریافتی، استفاده شده‌است. سپس توسط پارامتر اختیاری onCompleted، در پایان کار، یک متد خاص را می‌توان فراخوانی کرد. خروجی برنامه در این حالت نیز به صورت ذیل است:

```
1
2
3
4
5
Done!
```

البته اگر قصد خلاصه نویسی داشته باشیم، سطر آخر متد Main، با سطر ذیل یکی است:

```
observableQuery.Subscribe(Console.WriteLine, finished);
```

در این مثال ساده صرفاً یک Syntax دیگر را نسبت به حلقه‌ی foreach متداول مشاهده کردیم که اندکی فشرده‌تر است. در هر دو حالت نیز عملیات انجام شده در تردجاری صورت گرفته‌اند. اما قابلیت‌ها و ارزش‌های واقعی Rx زمانی آشکار خواهند شد که پردازش موازی و پردازش در تردهای دیگر را در آن فعال کنیم.

الگوی Observer

Rx پیاده سازی کننده‌ی الگوی طراحی شیء‌گرایی به نام [Observer](#) است. برای توضیح آن یک لامپ و سوئیچ برق را در نظر بگیرید. زمانیکه لامپ مشاهده می‌کند سوئیچ برق در حالت روشن قرار گرفته‌است، روشن خواهد شد و برعکس. در اینجا به سوئیچ، subject و به لامپ، observer گفته می‌شود. هر زمان که حالت سوئیچ تغییر می‌کند، از طریق یک callback، وضعیت خود را به observer اعلام خواهد کرد. علت استفاده از callbackها، ارائه راه‌حل‌های عمومی است تا بتواند با انواع و اقسام اشیاء کار کند. به این ترتیب هر بار که شیء observer از نوع متفاوتی تعریف می‌شود (مثلاً بجای لامپ یک خودرو قرار گیرد)، نیازی نخواهد بود تا subject را تغییر داد.

در Rx دو اینترفیس معادل observer و subject تعریف شده‌اند. در اینجا اینترفیس IObservable معادل observer است و اینترفیس IObservable معادل subject می‌باشد:

```
class Subject : IObservable<int>
{
    public IDisposable Subscribe(IObserver<int> observer)
    {
```

```
}
}
```

کار متد Subscribe، اتصال به Observer است و برای این حالت نیاز به کلاسی دارد که اینترفیس IObservable را پیاده سازی کند.

```
class Observer : IObservable<int>
{
    public void OnCompleted()
    {
    }

    public void OnError(Exception error)
    {
    }

    public void OnNext(int value)
    {
    }
}
```

در اینجا OnCompleted زمانی اجرا می‌شود که پردازش مجموعه‌ای از اعداد int پایان یافته باشد. OnError در زمان وقوع استثنایی اجرا می‌شود و OnNext به ازای هر عدد موجود در مجموعه‌ای در حال پردازش، یکبار اجرا می‌شود. البته نیازی به پیاده سازی صریح این اینترفیس نیست و توسط متد توکار Observable.Create می‌توان به همین نتیجه رسید. مجموعه‌های Observable کلید کار با Rx هستند. در مثال قبل ملاحظه کردیم که با استفاده از متد الحاقی ToObservable بر روی یک کوئری LINQ و یا هر نوع IEnumerable ای، می‌توان یک مجموعه‌ی Observable را ایجاد کرد. خروجی کوئری حاصل از آن به صورت خودکار اینترفیس IObservable را پیاده سازی می‌کند که دارای یک متد به نام Subscribe است. در متد Subscribe کاری که به صورت خودکار صورت خواهد گرفت، ایجاد یک حلقه‌ی foreach بر روی مجموعه‌ی مورد آنالیز و سپس فراخوانی متد OnNext کلاس پیاده سازی کننده‌ی IObservable به ازای هر آیتیم موجود در مجموعه است (فراخوانی observer.OnNext). در پایان کار هم فقط return this در اینجا صورت خواهد گرفت. در حین پردازش حلقه، اگر خطایی رخ دهد، متد observer.OnError انجام می‌شود.

در مثال قبل، کوئری LINQ نوشته شده، خروجی از نوع IObservable ندارد. به کمک متد الحاقی ToObservable:

```
public static System.IObservable<TSource> ToObservable<TSource>(
    this System.Collections.Generic.IEnumerable<TSource> source,
    System.Reactive.Concurrency.IScheduler scheduler)
```

به صورت خودکار، IEnumerable حاصل از کوئری LINQ را تبدیل به یک IObservable کرده‌ایم. به این ترتیب اکنون کوئری LINQ ما همانند سوئیچ برق عمل می‌کند و با تغییر آیتیم‌های موجود در آن، مشاهده‌گرهایی که به آن متصل شده‌اند (از طریق فراخوانی متد Subscribe)، امکان دریافت سیگنال‌های تغییر وضعیت آن‌را خواهند داشت. البته استفاده از متد Subscribe به نحوی که در مثال قبل ذکر شد، خلاصه شده‌ی الگوی Observer است. اگر بخواهیم دقیقاً مانند الگو عمل کنیم، چنین شکلی را خواهد داشت:

```
var query = Enumerable.Range(1, 5).Select(number => number);
var observableQuery = query.ToObservable();
var observer = Observer.Create<int>(onNext: number => Console.WriteLine(number));
observableQuery.Subscribe(observer);
```

ابتدا توسط متد ToObservable یک IObservable (سوئیچ) را ایجاد کرده‌ایم. سپس توسط کلاس Observer موجود در فضای نام System.Reactive، یک IObservable (لامپ) را ایجاد کرده‌ایم. کار اتصال سوئیچ به لامپ در متد Subscribe انجام می‌شود. اکنون هر زمانیکه تغییری در وضعیت observableQuery حاصل شود، سیگنالی را به observer ارسال می‌کند. در اینجا callbacks کار مدیریت observer را انجام می‌دهند.

پردازش نتایج یک کوئری LINQ در تدری دیگر توسط Rx

برای اجرای نتایج متد Subscribe در یک ترد جدید، می‌توان پارامتر scheduler متد ToObservable را مقدار دهی کرد:

```
using System;
using System.Linq;
using System.Reactive.Concurrency;
using System.Reactive.Linq;
using System.Threading;

namespace Rx01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Thread-Id: {0}", Thread.CurrentThread.ManagedThreadId);
            var query = Enumerable.Range(1, 5).Select(number => number);
            var observableQuery = query.ToObservable(scheduler: NewThreadScheduler.Default);
            observableQuery.Subscribe(onNext: number =>
            {
                Console.WriteLine("number: {0}, on Thread-id: {1}", number,
                Thread.CurrentThread.ManagedThreadId);
            }, onCompleted: () => finished());
        }

        private static void finished()
        {
            Console.WriteLine("Done!");
        }
    }
}
```

خروجی این مثال به نحو ذیل است:

```
Thread-Id: 1
number: 1, on Thread-id: 3
number: 2, on Thread-id: 3
number: 3, on Thread-id: 3
number: 4, on Thread-id: 3
number: 5, on Thread-id: 3
Done!
```

پیش از آغاز کار و در متد Main، ترد آی دی ثبت شده مساوی 1 است. سپس هربار که callback متد Subscribe فراخوانی شده‌است، ملاحظه می‌کنید که ترد آی دی آن مساوی عدد 3 است. به این معنا که کلیه نتایج در یک ترد مشخص دیگر پردازش شده‌اند.

NewThreadScheduler.Default در فضای نام System.Reactive.Concurrency واقع شده‌است.

یک نکته

در نگارش‌های آغازین Rx، مقدار scheduler را می‌شد معادل Scheduler.NewThread نیز قرار داد که در نگارش‌های جدید منسوخ شده در نظر گرفته شده و به زودی حذف خواهد شد. معادل‌های جدید آن اکنون NewThreadScheduler.Default، ThreadPoolscheduler.Default و امثال آن هستند.

مدیریت خاتمه‌ی اعمال انجام شده‌ی در تردهای دیگر توسط Rx

یکی از مواردی که حین اجرای نتیجه‌ی callback‌های پردازش شده‌ی در تردهای دیگر نیاز است بدانیم، زمان خاتمه‌ی کار آن‌ها است. برای نمونه در مثال قبل، نمایش Done پس از پایان تمام callbacks انجام شده‌است. فرض کنید، callback پایان عملیات را حذف کرده و متد finished را پس از فراخوانی متد observableQuery.Subscribe قرار دهیم:

```
observableQuery.Subscribe(onNext: number =>
{
    Console.WriteLine("number: {0}, on Thread-id: {1}", number,
    Thread.CurrentThread.ManagedThreadId);
}/*, onCompleted: () => finished()*/);
```

```
finished();
```

اینبار اگر برنامه را اجرا کنیم به خروجی ذیل خواهیم رسید:

```
Thread-Id: 1
number: 1, on Thread-id: 3
Done!
number: 2, on Thread-id: 3
number: 3, on Thread-id: 3
number: 4, on Thread-id: 3
number: 5, on Thread-id: 3
```

این خروجی بدین معنا است که متد `observableQuery.Subscribe` در حین اجرا شدن در تردی دیگر، صبر نخواهد کرد تا عملیات مرتبط با آن خاتمه یابد و سپس سطر بعدی را اجرا کند. بنابراین برای حل این مشکل، تنها کافی است به آن اعلام کنیم که پس از پایان عملیات، `onCompleted` را اجرا کن.

مدیریت استثناهای رخ داده در حین پردازش مجموعه‌های واکنشگرا

متد `Subscribe` دارای چندین `overload` است. تا اینجا نمونه‌ای که دارای پارامترهای `onNext` و `onCompleted` بودند را بررسی کردیم. اگر بخواهیم مدیریت استثناءها را نیز در اینجا اضافه کنیم، فقط کافی است از `overload` دیگر آن که دارای پارامتر `onError` است، استفاده نمائیم:

```
observableQuery.Subscribe(
    onNext: number => Console.WriteLine(number),
    onError: exception => Console.WriteLine(exception.Message),
    onCompleted: () => finished());
```

اگر `callback` پارامتر `onError` اجرا شود، دیگر به `onCompleted` نخواهیم رسید. همچنین دیگر `onNext` ایی نیز اجرا نخواهد شد.

مدیریت ترد اجرای نتایج حاصل از Rx در یک برنامه‌ی دسکتاپ WPF یا WinForms

تا اینجا مشاهده کردیم که اجرای `callback`های `observer` در یک ترد دیگر، به سادگی تنظیم پارامتر `scheduler` متد `ToObservable` است. اما در برنامه‌های دسکتاپ برای به روز رسانی عناصر رابط کاربری، حتما باید در تردی قرار داشته باشیم که آن رابط کاربری در آن ایجاد شده است یا به عبارتی در ترد اصلی برنامه؛ در غیر اینصورت برنامه کرش خواهد کرد. مدیریت این مساله نیز در Rx بسیار ساده است. ابتدا نیاز است بسته‌ی [Rx-WPF](#) را نصب کرد:

```
PM> Install-Package Rx-WPF
```

سپس توسط متد `ObserveOn` می‌توان مشخص کرد که نتیجه‌ی عملیات باید بر روی کدام ترد اجرا شود:

```
observableQuery.ObserveOn(DispatcherScheduler.Current).Subscribe(...)
```

روش دیگر آن استفاده از متد `ObserveOnDispatcher` می‌باشد:

```
observableQuery.ObserveOnDispatcher().Subscribe(...)
```

بنابراین مشخص سازی پارامتر `scheduler` متد `ToObservable`، به معنای اجرای `query` آن در یک ترد دیگر و استفاده از متد `ObserveOn`، به معنای مشخص سازی ترد اجرای `callback`های مشاهده‌گر است.

و یا اگر از WinForms استفاده می‌کنید، ابتدا [بسته‌ی Rx خاص آن را](#) نصب کنید:

```
PM> Install-Package Rx-WinForms
```

و سپس ترد اجرای callback ها را `SynchronizationContext.Current` مشخص نمائید:

```
observableQuery.ObserveOn(SynchronizationContext.Current).Subscribe(...)
```

یک نکته

در Rx فرض می‌شود که کوثری شما زمانبر است و callback های مشاهده‌گر سریع عمل می‌کنند. بنابراین هدف از callback های آن، پردازش‌های سنگین نیست. جهت آزمایش این مساله، اینبار query ابتدایی برنامه را به شکل ذیل تغییر دهید که در آن بازگشت زمانبر یک سری داده شبیه سازی شده‌اند.

```
var query = Enumerable.Range(1, 5).Select(number =>
{
    Thread.Sleep(250);
    return number;
});
```

سپس با استفاده از متد `ToObservable`، ترد دیگری را برای اجرای واقعی آن مشخص کنید تا در حین اجرای آن برنامه در حالت هنگ به نظر نرسد و سپس نمایش آن را به کمک متد `ObserveOn`، بر روی ترد اصلی برنامه انجام دهید.

نظرات خوانندگان

نویسنده:

ژوپتر

تاریخ:

۱۱:۳۷ ۱۳۹۳/۰۲/۲۹

به نظرم باید نوع آرگومان اینجا مشخص باشه:

```
var observableQuery = query.ToObservable(scheduler: NewThreadScheduler.Default);
```

```
var observableQuery = query.ToObservable<int>(scheduler: NewThreadScheduler.Default);
```

نویسنده:

وحید نصیری

تاریخ:

۱۱:۵۲ ۱۳۹۳/۰۲/۲۹

زمانیکه از ریشارپر استفاده می‌کنید، این تعیین نوع صریح را به صورت کم رنگ (به معنای کد مرده یا زاید) معرفی می‌کند:

```
var observableQuery = query.ToObservable<int>(scheduler: NewThreadScheduler.Default);
//observableQuery.ObserveOn(Synchronizat
observableQuery/*.ObserveOnDispatcher()*.Type argument specification is redundant
current).Subscribe(
```

علت اینجا است که نوع آرگومان جنریک به صورت خودکار توسط نوع پارامتر ارسالی به متد قابل تشخیص است (در اینجا چون ToObservable یک متد الحاقی است، اولین پارامتر آن، عناصر توالی query هستند که از نوع IEnumerable of int تعریف شدند). برای مطالعه بیشتر مراجعه کنید به [Inference of type arguments part 25.6.4](#) C# specs (ECMA-334)

نویسنده:

ژوپتر

تاریخ:

۱۲:۱۵ ۱۳۹۳/۰۲/۲۹

حق با شماست. متأسفانه نمی‌دانم چرا ابتدا کامپایلر از این خط خطا می‌گرفت و می‌گفت باید نوع آرگومان تعیین شود.

در مطلب « [معرفی Reactive extensions](#) » با نحوه‌ی تبدیل IEnumerableها به نمونه‌های Observable آشنا شدیم. اما سایر حالات چگونه؟ آیا Rx صرفاً محدود است به کار با IEnumerableها؟ در ادامه نگاهی خواهیم داشت به نحوه‌ی تبدیل بسیاری از منابع داده دیگر به توالی‌های Observable قابل استفاده در Rx.

روش‌های متفاوت ایجاد توالی (sequence) در Rx

الف) استفاده از متدهای Factory

Observable.Create (1)

نمونه‌ای از استفاده از آن را در مطلب « [معرفی Reactive extensions](#) » مشاهده کردید.

```
var query = Enumerable.Range(1, 5).Select(number => number);
var observableQuery = query.ToObservable();
var observer = Observer.Create<int>(onNext: number => Console.WriteLine(number));
observableQuery.Subscribe(observer);
```

کار آن، تدارک delegate ایی است که توسط متد Subscribe، به ازای هربار پردازش مقدار موجود در توالی معرفی شده به آن، فراخوانی می‌گردد و هدف اصلی از آن این است که به صورت دستی اینترفیس IObservable را پیاده سازی نکنید (امکان پیاده سازی inline یک اینترفیس توسط Actionها). البته در این مثال فقط delegate مربوط به onNext را ملاحظه می‌کند. توسط سایر overloadهای آن امکان ذکر delegateهای onError/onCompleted نیز وجود دارد.

Observable.Return (2)

برای ایجاد یک خروجی Observable از یک مقدار مشخص، می‌توان از متد جنریک Observable.Return استفاده کرد. برای مثال:

```
var observableValue1 = Observable.Return("Value");
var observableValue2 = Observable.Return(2);
```

در ادامه نحوه‌ی پیاده سازی این متد را توسط Observable.Create مشاهده می‌کنید:

```
public static IObservable<T> Return<T>(T value)
{
    return Observable.Create<T>(o =>
    {
        o.OnNext(value);
        o.OnCompleted();
        return Disposable.Empty;
    });
}
```

البته دو سطر نوشته شده در اصل معادل هستند با سطرهای ذیل؛ که ذکر نوع جنریک آن‌ها ضروری نیست. زیرا به صورت خودکار از نوع آرگومان معرفی شده، تشخیص داده می‌شود:

```
var observableValue1 = Observable.Return<string>("Value");
var observableValue2 = Observable.Return<int>(2);
```

Observable.Empty (3)

برای بازگشت یک توالی خالی که تنها کار اطلاع رسانی onCompleted را انجام می‌دهد.

```
var emptyObservable = Observable.Empty<string>();
```

در کدهای ذیل، پیاده سازی این متد را توسط Observable.Create مشاهده می‌کنید:

```
public static IObservable<T> Empty<T>()
{
    return Observable.Create<T>(o =>
    {
        o.OnCompleted();
        return Disposable.Empty;
    });
}
```

Observable.Never (4)

برای بازگشت یک توالی بدون قابلیت اطلاع رسانی و notification

```
var neverObservable = Observable.Never<string>();
```

این متد به نحو زیر توسط Observable.Create پیاده سازی شده‌است:

```
public static IObservable<T> Never<T>()
{
    return Observable.Create<T>(o =>
    {
        return Disposable.Empty;
    });
}
```

Observable.Throw (5)

برای ایجاد یک توالی که صرفاً کار اطلاع رسانی OnError را توسط استثنای معرفی شده به آن انجام می‌دهد.

```
var throwObservable = Observable.Throw<string>(new Exception());
```

در ادامه نحوه‌ی پیاده سازی این متد را توسط Observable.Create مشاهده می‌کنید:

```
public static IObservable<T> Throws<T>(Exception exception)
{
    return Observable.Create<T>(o =>
    {
        o.OnError(exception);
        return Disposable.Empty;
    });
}
```

Observable.Range توسط (6)

به سادگی می‌توان بازه‌ی Observable ایی را ایجاد کرد:

```
var range = Observable.Range(10, 15);
range.Subscribe(Console.WriteLine, () => Console.WriteLine("Completed"));
```

Observable.Generate (7)

اگر بخواهیم عملیات Observable.Range را پیاده سازی کنیم، می‌توان از متد Observable.Generate استفاده کرد:

```
public static IObservable<int> Range(int start, int count)
{
    var max = start + count;
    return Observable.Generate(
        initialState: start,
```

```

        condition: value => value < max,
        iterate: value => value + 1,
        resultSelector: value => value);
    }

```

توسط پارامتر `initialState`، مقدار آغازین را دریافت می‌کند. پارامتر `condition`، مشخص می‌کند که توالی چه زمانی باید خاتمه یابد. در پارامتر `iterate`، مقدار جاری دریافت شده و مقدار بعدی تولید می‌شود. `resultSelector` کار تبدیل و بازگشت مقدار خروجی را به عهده دارد.

Observable.Interval (8)

عموماً از انواع و اقسام تایمرهای موجود در دات نت مانند `System.Timers.Timer`، `System.Threading.Timer` و `System.Windows.Threading.DispatcherTimer` برای ایجاد یک توالی از رخدادها استفاده می‌شود. تمام این‌ها را به سادگی می‌توان توسط متد `Observable.Interval`، که قابل انتقال به تمام پلتفرم‌هایی است که Rx برای آن‌ها تهیه شده‌است، جایگزین کرد:

```

var interval = Observable.Interval(period: TimeSpan.FromMilliseconds(250));
interval.Subscribe(Console.WriteLine, () => Console.WriteLine("completed"));

```

در اینجا تایمر تهیه شده، هر 450 میلی‌ثانیه یکبار اجرا می‌شود. برای خاتمه‌ی آن باید شیء `interval` را `Dispose` کنید. `Overload` دوم این متد، امکان معرفی `scheduler` و اجرای بر روی تردی دیگر را نیز میسر می‌کند.

Observable.Timer (9)

تفاوت `Observable.Timer` با `Observable.Interval` در مفهوم پارامتر ارسالی به آن‌ها است:

```

var timer = Observable.Timer(dueTime: TimeSpan.FromSeconds(1));
timer.Subscribe(Console.WriteLine, () => Console.WriteLine("completed"));

```

یکی `due time` دارد (مدت زمان صبر کردن تا تولید اولین خروجی) و دیگری `period` (به صورت متوالی تکرار می‌شود). خروجی `Observable.Interval` مثال زده شده به نحو زیر است و خاتمه‌ای ندارد:

0
1
2
3
4
5

اما خروجی `Observable.Timer` به نحو ذیل بوده و پس از یک ثانیه، خاتمه می‌یابد:

0

completed

متد `Observable.Timer` دارای هفت `overload` متفاوت است که توسط آن‌ها `dueTime` (مدت زمان صبر کردن تا تولید اولین خروجی)، `period` (کار `Observable.Timer` را به صورت متوالی در بازه‌ی زمانی مشخص شده تکرار می‌کند) و `scheduler` (تعیین ترد اجرایی عملیات) قابل مقدار دهی هستند.

اگر می‌خواهید `Observable.Timer` بلافاصله شروع به کار کند، مقدار `dueTime` آن‌را مساوی `TimeSpan.Zero` قرار دهید. به این ترتیب یک `Observable.Interval` را به وجود آورده‌اید که بلافاصله شروع به کار کرده است و تا مدت زمان مشخص شده‌ای جهت اجرای اولین `callback` خود صبر نمی‌کند.

ب) تبدیلگرهایی که خروجی `IObservable` ایجاد می‌کنند

برای تبدیل مدل‌های برنامه نویسی Async قدیمی دات نت مانند APM، رخدادها و امثال آن به معادل‌های Rx، متدهای الحاقی خاصی تهیه شده‌اند.

1) تبدیل delegates به معادل Observable

متد Observable.Start، امکان تبدیل یک Func یا Action زمانبر را به یک توالی observable میسر می‌کند. در این حالت به صورت پیش فرض، پردازش عملیات بر روی یکی از تردهای ThreadPool انجام می‌شود.

```
static void StartAction()
{
    var start = Observable.Start(() =>
    {
        Console.WriteLine("Observable.Start");
        for (int i = 0; i < 10; i++)
        {
            Thread.Sleep(100);
            Console.WriteLine(".");
        }
    });
    start.Subscribe(
        onNext: unit => Console.WriteLine("published"),
        onCompleted: () => Console.WriteLine("completed"));
}

static void StartFunc()
{
    var start = Observable.Start(() =>
    {
        Console.WriteLine("Observable.Start");
        for (int i = 0; i < 10; i++)
        {
            Thread.Sleep(100);
            Console.WriteLine(".");
        }
        return "value";
    });
    start.Subscribe(
        onNext: Console.WriteLine,
        onCompleted: () => Console.WriteLine("completed"));
}
```

در اینجا دو مثال از بکارگیری Action و Funcها را توسط Observable.Start مشاهده می‌کنید. زمانیکه از Func استفاده می‌شود، تابع یک خروجی را ارائه داده و سپس توالی خاتمه می‌یابد. اگر از Action استفاده شود، نوع Observable بازگشت داده شده از نوع Unit است که در برنامه نویسی functional معادل void است و هدف از آن مشخص سازی پایان عملیات Action می‌باشد. Unit دارای مقداری نبوده و صرفاً سبب اجرای اطلاع رسانی OnNext می‌شود. تفاوت مهم Observable.Start و Observable.Return در این است که Observable.Start مقدار تابع را به صورت تنبل (lazily) پردازش می‌کند، اما Observable.Return پردازش حریصانه‌ای (eagerly) را به همراه خواهد داشت. به این ترتیب Observable.Start بسیار شبیه به یک Task (پردازش‌های غیرهمزمان) عمل می‌کند. در اینجا شاید این سؤال مطرح شود که استفاده از قابلیت‌های Async سی‌شارپ 5 برای اینگونه کارها مناسب است یا Rx؟ قابلیت‌های Async بیشتر به اعمال مخصوص IO bound مانند کار با شبکه، دریافت فایل از اینترنت، کار با یک بانک اطلاعاتی خارج از مرزهای سیستم، مرتبط می‌شوند؛ اما اعمال CPU bound مانند محاسبات سنگین حاصل از توالی‌های observable را به خوبی می‌توان توسط Rx مدیریت کرد.

2) تبدیل Events به معادل Observable

دات نت از روزهای اول خود به همراه یک event driven programming model بوده‌است. Rx متدهایی را برای دریافت یک رخداد و تبدیل آن به یک توالی Observable ارائه داده‌است. برای نمونه ObservableCollection زیر را در نظر بگیرید

```
var items = new System.Collections.ObjectModel.ObservableCollection<string>
{
    "Item1", "Item2", "Item3"
```



```
};
```

اگر بخواهیم مانند روش‌های متداول، حذف شدن آیتم‌های آن‌را تحت نظر قرار دهیم، می‌توان نوشت:

```
items.CollectionChanged += (sender, ea) =>
{
    if (ea.Action == NotifyCollectionChangedEventArgs.Remove)
    {
        foreach (var oldItem in ea.OldItems.Cast<string>())
        {
            Console.WriteLine("Removed {0}", oldItem);
        }
    }
};
```

این نوع کدها در WPF زیاد کاربرد دارند. اکنون معادل کدهای فوق با Rx به صورت زیر هستند:

```
var removals =
    Observable.FromEventPattern<NotifyCollectionChangedEventArgs,
        NotifyCollectionChangedEventArgs>
    (
        addHandler: handler => items.CollectionChanged += handler,
        removeHandler: handler => items.CollectionChanged -= handler
    )
    .Where(e => e.EventArgs.Action == NotifyCollectionChangedEventArgs.Remove)
    .SelectMany(c => c.EventArgs.OldItems.Cast<string>());

var disposable = removals.Subscribe(onNext: item => Console.WriteLine("Removed {0}",
    item));
```

با استفاده از متد `Observable.FromEventPattern` می‌توان معادل `Observable` رخداد `CollectionChanged` را تهیه کرد. پارامتر اول جنریک آن، نوع رخداد است و پارامتر اختیاری دوم آن، `EventArgs` این رخداد. همچنین با توجه به قسمت `Where` نوشته شده، در این بین مواردی را که `Action` مساوی حذف شدن را دارا هستند، فیلتر کرده و نهایتاً لیست `Observable` آن‌ها بازگشت داده می‌شوند. اکنون می‌توان با استفاده از متد `Subscribe`، این تغییرات را دریافت کرد. برای مثال با فراخوانی

```
items.Remove("Item1");
```

بلافاصله خروجی `Removed item1` ظاهر می‌شود.

(3) تبدیل Task به معادل Observable

متد `ToObservable` واقع در فضای نام `System.Reactive.Threading.Tasks` را بر روی یک `Task` نیز می‌توان فراخوانی کرد:

```
var task = Task.Factory.StartNew(() => "Test");
var source = task.ToObservable();
source.Subscribe(Console.WriteLine, () => Console.WriteLine("completed"));
```

البته باید دقت داشت استفاده از `Task` دات نت 4.5 که بیشتر جهت پردازش‌های `async` اعمال `I/O-bound` طراحی شده‌است، بر `IObservable` مقدم است. صرفاً اگر نیاز است این `Task` را با سایر `observables` ادغام کنید از متد `ToObservable` برای کار با آن استفاده نمائید.

(4) تبدیل IEnumerable به معادل Observable

با این مورد [تاکنون](#) آشنا شده‌اید. فقط کافی است متد `ToObservable` را بر روی یک `IEnumerable`، جهت تهیه خروجی `Observable` فراخوانی کرد.

(5) تبدیل APM به معادل Observable

APM یا Asynchronous programming model، همان روش کار با متدهای Async با نام‌های BeginXXX و EndXXX است که از نگارش‌های آغازین دات نت به همراه آن بوده‌اند. کار کردن با آن مشکل است و مدیریت آن به همراه پراکندگی‌های بسیاری جهت کار با callbacks آن است. برای تبدیل این نوع روش برنامه نویسی به روش Rx نیز متدهایی پیش بینی شده‌است؛ مانند `Observable.FromAsyncPattern`.

یک نکته

کتابخانه‌ای به نام Rxx بسیاری از این محصور کننده‌ها را تهیه کرده‌است:

<http://Rxx.codeplex.com>

ابتدا بسته‌ی نیوگت آن را نصب کنید:

```
PM> Install-Package Rxx
```

سپس برای نمونه، برای کار با یک فایل استریم خواهیم داشت:

```
using (new FileStream("file.txt", FileMode.Open)
        .ReadToEndObservable()
        .Subscribe(x => Console.WriteLine(x.Length)))
{
    Console.ReadKey();
}
```

متد `ReadToEndObservable` یکی از متدهای الحاقی کتابخانه‌ی Rxx است.

پس از [معرفی](#) و مشاهده‌ی نحوه‌ی [ایجاد توالی‌ها در Rx](#) ، بهتر است با نمونه‌ای از نحوه‌ی استفاده از آن در یک برنامه‌ی WPF آشنا شویم.

بنابراین ابتدا دو بسته‌ی Rx-Main و Rx-WPF را توسط نیوگت، به یک برنامه‌ی جدید WPF اضافه کنید:

```
PM> Install-Package Rx-Main
PM> Install-Package Rx-WPF
```

فرض کنید قصد داریم محتوای یک فایل حجیم را به نحو ذیل خوانده و توسط Rx نمایش دهیم.

```
private static IEnumerable<string> readFile(string filename)
{
    using (TextReader reader = File.OpenText(filename))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            Thread.Sleep(100);
            yield return line;
        }
    }
}
```

در اینجا برای ایجاد یک توالی `IEnumerable` ، از `yield return` استفاده شده‌است. همچنین `Thread.Sleep` آن جهت بررسی قفل شدن رابط کاربری در حین خواندن فایل به عمد قرار گرفته است. UI برنامه نیز به نحو ذیل است:

```
<Window x:Class="WpfApplicationRxTests.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="450" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Button Grid.Row="0" Name="btnGenerateSequence" Click="btnGenerateSequence_Click">Generate
sequence</Button>
        <ListBox Grid.Row="1" Name="lstNumbers" />
        <Button Grid.Row="2" IsEnabled="False" Name="btnStop" Click="btnStop_Click">Stop</Button>
    </Grid>
</Window>
```

با این کدها

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.IO;
using System.Reactive.Concurrency;
using System.Reactive.Linq;
using System.Threading;
using System.Windows;

namespace WpfApplicationRxTests
{
    public partial class MainWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

```

    }

    private static IEnumerable<string> readFile(string filename)
    {
        using (TextReader reader = File.OpenText(filename))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                Thread.Sleep(100);
                yield return line;
            }
        }
    }

    private IDisposable _subscribe;
    private void btnGenerateSequence_Click(object sender, RoutedEventArgs e)
    {
        btnGenerateSequence.IsEnabled = false;
        btnStop.IsEnabled = true;

        var items = new ObservableCollection<string>();
        lstNumbers.ItemsSource = items;
        _subscribe = readFile("test.txt").ToObservable()
            .SubscribeOn(ThreadPoolScheduler.Instance)
            .ObserveOn(DispatcherScheduler.Current)
            .Finally(finallyAction: () =>
            {
                btnGenerateSequence.IsEnabled = true;
                btnStop.IsEnabled = false;
            })
            .Subscribe(onNext: line =>
            {
                items.Add(line);
            },
            onError: ex => { },
            onCompleted: () =>
            {
                //lstNumbers.ItemsSource = items;
            });
    }

    private void btnStop_Click(object sender, RoutedEventArgs e)
    {
        _subscribe.Dispose();
    }
}

```

توضیحات

حاصل متد `readFile` را که یک توالی معمولی `IEnumerable` را ایجاد می‌کند، توسط فراخوانی متد `ToObservable`، تبدیل به یک خروجی `IObservable` کرده‌ایم تا بتوانیم هربار که سطری از فایل مدنظر خوانده می‌شود، نسبت به آن واکنش نشان دهیم. متد `SubscribeOn` مشخص می‌کند که این توالی `Observable` باید بر روی چه تردی اجرا شود. در اینجا از `ThreadPoolScheduler.Instance` استفاده شده‌است تا در حین خواندن فایل، رابط کاربری در حالت هنگ به نظر نرسد و ترد جاری (ترد اصلی برنامه) به صورت خودکار آزاد گردد. از متد `ObserveOn` با پارامتر `DispatcherScheduler.Current` استفاده کرده‌ایم، تا نتیجه‌ی واکنش‌های به خوانده شدن سطرهای یک فایل مفروض، در ترد اصلی برنامه صورت گیرد. در غیر اینصورت امکان کار کردن با عناصر رابط کاربری در یک ترد دیگر وجود نخواهد داشت و برنامه کرش می‌کند. در قسمت‌های قبل، صرفاً متد `Subscribe` را مشاهده کرده بودید. در اینجا از متد `Finally` نیز استفاده شده‌است. علت اینجا است که اگر در حین خواندن فایل خطایی رخ دهد، قسمت `onError` متد `Subscribe` اجرا شده و دیگر به پارامتر `onCompleted` آن نخواهیم رسید. اما متد `Finally` آن همیشه در پایان عملیات اجرا می‌شود. خروجی حاصل از متد `Subscribe`، از نوع `IDisposable` است. `Rx` به صورت خودکار پس از پردازش آخرین عنصر توالی، این شیء را `Dispose` می‌کند. اینجا است که `callback` متد `Finally` یاد شده فراخوانی خواهد شد. اما اگر در حین خواندن یک فایل طولانی، کاربر علاقمند باشد تا عملیات را متوقف کند، تنها کافی است که به صورت صریح، این شیء را `Dispose` نماید. به همین جهت است که مشاهده می‌کنید، این خروجی به صورت یک فیلد تعریف شده‌است تا در متد `Stop` بتوانیم آن را در صورت نیاز

Dispose کنیم.

مثال فوق را از اینجا نیز می‌توانید دریافت کنید:

[WpfApplicationRxTests.zip](#)

به صورت پیش فرض، Rx هر بار تنها یک مقدار را بررسی می‌کند. اما گاهی از اوقات نیاز است تا در هر بار، بیشتر از یک مقدار دریافت و پردازش شوند. برای این منظور Rx متدهای الحاقی ویژه‌ای را به نام‌های Scan، Buffer و Window تدارک دیده‌است تا بتواند از یک توالی، چندین توالی را تولید کند (توالی توالی‌ها = Sequence of sequences).

متد Scan

فرض کنید قصد دارید تعدادی عدد را با هم جمع بزنید. برای اینکار عموماً عدد اول با عدد دوم جمع زده شده و سپس حاصل آن با عدد سوم جمع زده خواهد شد و به همین ترتیب تا آخر توالی. کار متد Scan نیز دقیقاً به همین نحو است. هر بار که قرار است توالی پردازش شود، حاصل عملیات مرحله‌ی قبل را در اختیار مصرف کننده قرار می‌دهد. در مثال ذیل، قصد داریم حاصل جمع اعداد موجود در آرایه‌ای را بدست بیاوریم:

```
var sequence = new[] { 12, 3, -4, 7 }.ToObservable();
var runningSum = sequence.Scan((accumulator, value) =>
{
    Console.WriteLine("accumulator {0}", accumulator);
    Console.WriteLine("value {0}", value);
    return accumulator + value;
});
runningSum.Subscribe(result => Console.WriteLine("result {0}\n", result));
```

با این خروجی

```
result 12
accumulator 12
value 3
result 15
accumulator 15
value -4
result 11
accumulator 11
value 7
result 18
```

در اولین بار اجرای متد Subscribe، کار مقدار دهی accumulator با اولین عنصر آرایه صورت می‌گیرد. در دفعات بعدی، مقدار این accumulator با عدد جاری جمع زده شده و حاصل این عملیات در تکرار آتی، مجدداً توسط accumulator قابل دسترسی خواهد بود.

یک نکته: اگر علاقمند باشیم که مقدار اولیه‌ی accumulator، اولین عنصر توالی نباشد، می‌توان آن را توسط پارامتر seed متد Scan مقدار دهی کرد:

```
var runningSum = sequence.Scan(seed: 10, accumulator: (accumulator, value) =>
```

متد Buffer

متد بافر، کار تقسیم یک توالی را به توالی‌های کوچکتر، بر اساس زمان، یا تعداد عنصر مشخص شده، انجام می‌دهد. برای مثال در برنامه‌های دسکتاپ شاید نیازی نباشد تا به ازای هر عنصر توالی، یکبار رابط کاربری را به روز کرد. عموماً بهتر است تا تعداد

مشخصی از عناصر یکجا پردازش شده و نتیجه‌ی این پردازش به تدریج نمایش داده شود.

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Buffer(count: 10);

sequence.Subscribe(onNext: numbers =>
{
    Console.WriteLine(numbers.Sum());
});
```

در اینجا نحوه‌ی استفاده از متد بافر را به همراه مشخص کردن تعداد اعضای بافر ملاحظه می‌کنید. هربار که `onNext` متد `Subscribe` فراخوانی شود، 10 عنصر از توالی را در اختیار خواهیم داشت (بجای یک عنصر حالت متداول بافر نشده). به این ترتیب می‌توان فشار حجم اطلاعات ورودی با فرکانس بالا را کنترل کرد و در نتیجه از منابع موجود بهتر استفاده نمود. برای مثال اگر می‌خواهید عملیات `bulk insert` را انجام دهید، می‌توان بر اساس یک `batch size` مشخص، گروه گروه اطلاعات را به بانک اطلاعاتی اضافه کرد تا فشار کار کاهش یابد.

همینکار را بر اساس زمان نیز می‌توان انجام داد:

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Buffer(TimeSpan.FromSeconds(2));
```

در مثال فوق هر 2 ثانیه یکبار، مجموعه‌ای از عناصر به متد `onNext` ارسال خواهند شد.

متد Window

متد `Window` نیز دقیقاً همان پارامترهای متد بافر را قبول می‌کند. با این تفاوت که هربار، یک توالی `observable` را به متد `onNext` ارسال می‌کند. نوع `numbers` پارامتر `onNext`، در حین بکارگیری متد بافر در مثال‌های فوق، `IList of int` است. اما اگر متدهای `Buffer` را تبدیل به متد `Window` کنیم، اینبار نوع `numbers`، معادل `IObservable of int` خواهد شد.

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Window(TimeSpan.FromSeconds(2));

sequence.Subscribe(onNext: numbers =>
{
    numbers.Subscribe(onNext: number => Console.WriteLine(number));
});
```

چه زمانی باید از Buffer استفاده کرد و چه زمانی از Window؟

در متد بافر، به ازای هر توالی که به پارامتر `onNext` ارسال می‌شود، یکبار وهله‌ی جدیدی از توالی مدنظر در حافظه ایجاد و ارسال خواهد شد. در متد `Window` صرفاً اشاره‌گرهایی به این توالی را در اختیار داریم؛ بنابراین مصرف حافظه‌ی کمتری را شاهد خواهیم بود. متد `Window` بسیار مناسب است برای اعمال `aggregation`. مثلاً اگر نیاز است جمع، میانگین، حداقل و حداکثر عناصر دریافتی محاسبه شوند، بهتر است از متد `Window` استفاده شود که نهایتاً قابلیت استفاده از متدهای الحاقی `Sum` و `Min` و `Max` را به همراه دارد. با این تفاوت که حاصل این‌ها نیز یک `IObservable` است که باید `Subscribe` آن‌را برای دریافت نتیجه فراخوانی کرد:

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Window(10);

sequence.Subscribe(onNext: numbers =>
{
    numbers.Sum().Subscribe(onNext: number => Console.WriteLine(number));
});
```

```
});
```

در این حالت متد Window، برخلاف متد Buffer، توالی numbers را هربار کش نمی‌کند و به این ترتیب می‌توان به مصرف حافظه‌ی کمتری رسید.

کاربردهای دنیای واقعی

در اینجا دو مثال از بکارگیری متد Buffer را جهت پردازش مجموعه‌های عظیمی از اطلاعات و نمایش همزمان آن‌ها در رابط کاربری ملاحظه می‌کنید.

مثال اول: فرض کنید قصد دارید تمام فایل‌های درایو C خود را توسط یک TreeView نمایش دهید. در این حالت نباید رابط کاربری برنامه در حالت هنگ به نظر برسد. همچنین به علت زیاد بودن تعداد فایل‌ها و نمایش همزمان آن‌ها در UI، نباید CPU Usage برنامه تا حدی باشد که در کار سایر برنامه‌ها اختلال ایجاد کند. در این مثال‌ها با استفاده از Rx و متد بافر آن، هربار مثلاً 1000 آیتم را بافر کرده و سپس یکجا در TreeView نمایش می‌دهند. به این ترتیب دو شرط یاد شده محقق می‌شوند.

[The Rx Framework By Example](#)

مثال دوم: خواندن تعداد زیادی رکورد از بانک اطلاعاتی به همراه نمایش همزمان آن‌ها در UI بدون اختلالی در کار سیستم و همچنین هنگ کردن برنامه.

[Using Reactive Extensions for Streaming Data from Database](#)

اجرای Async اعمال نسبتاً طولانی، در برنامه‌های مبتنی بر داده، عموماً این مزایا را به همراه دارد:

الف) مقیاس پذیری سمت سرور

در اعمال سمت سرور متداول، تردهای متعددی جهت پردازش درخواست‌های کلاینت‌ها تدارک دیده می‌شوند. هر زمانیکه یکی از این تردها، یک عملیات blocking را انجام می‌دهد (مانند دسترسی به شبکه یا اعمال I/O)، ترد مرتبط با آن تا پایان کار این عملیات معطل خواهد شد. با بالا رفتن تعداد کاربران یک برنامه و در نتیجه بیشتر شدن تعداد درخواست‌هایی که سرور باید پردازش کند، تعداد تردهای معطل مانده نیز به همین ترتیب بیشتر خواهند شد. مشکل اصلی اینجا است که نمونه سازی تردها بسیار هزینه بر است (با اختصاص 1MB of virtual memory space) و منابع سرور محدود. با زیاد شدن تعداد تردهای معطل اعمال I/O یا شبکه، سرور مجبور خواهد شد بجای استفاده مجدد از تردهای موجود، تردهای جدیدی را ایجاد کند. همین مساله سبب بالا رفتن بیش از حد مصرف منابع و حافظه برنامه می‌گردد. یکی از روش‌های رفع این مشکل بدون نیاز به بهبودهای سخت افزاری، تبدیل اعمال blocking نامبرده شده به نمونه‌های non-blocking است. به این ترتیب ترد پردازش کننده‌ی این اعمال Async بلافاصله آزاد شده و سرور می‌تواند از آن جهت پردازش درخواست دیگری استفاده کند؛ بجای اینکه ترد جدیدی را وهله سازی نماید.

ب) بالا بردن پاسخ دهی کلاینت‌ها

کلاینت‌ها نیز اگر مدام درخواست‌های blocking را به سرور جهت دریافت پاسخی ارسال کنند، به زودی به یک رابط کاربری غیرپاسخگو خواهند رسید. برای رفع این مشکل نیز می‌توان از [توانمندی‌های Async دات نت 4.5](#) جهت آزاد سازی ترد اصلی برنامه یا همان ترد UI استفاده کرد.

و ... تمام این‌ها یک شرط را دارند. نیاز است یک چنین API خاصی که اعمال Async واقعی را پشتیبانی می‌کنند، فراهم شده باشد. بنابراین صرفاً وجود متد Task.Run، به معنای اجرای واقعی Async یک متد خاص نیست. برای این منظور ADO.NET 4.5 به همراه متدهای Async ویژه کار با بانک‌های اطلاعاتی است و پس از آن Entity framework 6 از این زیر ساخت استفاده کرده‌است که در ادامه جزئیات آن‌را بررسی خواهیم کرد.

پیشنیازها

برای کار با امکانات جدید Async موجود در EF 6 نیاز است از VS 2012 به بعد که به همراه کامپایلری است که واژه‌های کلیدی async و await را پشتیبانی می‌کند و همچنین دات نت 4.5 استفاده کرد. چون ADO.NET 4.5 اعمال async واقعی را پشتیبانی می‌کند، دات نت 4 در اینجا قابل استفاده نخواهد بود.

متدهای الحاقی جدید Async در EF 6.x

جهت متدهای الحاقی متداول EF مانند ToList, Max, Min و غیره، نمونه‌های Async آن‌ها نیز اضافه شده‌اند:

```
QueryableExtensions:
AllAsync
AnyAsync
AverageAsync
ContainsAsync
CountAsync
FirstAsync
FirstOrDefaultAsync
ForEachAsync
LoadAsync
LongCountAsync
MaxAsync
```

```

MinAsync
SingleAsync
SingleOrDefaultAsync
SumAsync
ToArrayAsync
ToDictionaryAsync
ToListAsync

DbSet:
FindAsync

DbContext:
SaveChangesAsync

Database:
ExecuteSqlCommandAsync

```

بنابراین اولین قدم تبدیل کدهای قدیمی به Async، استفاده از متدهای الحاقی فوق است.

چند مثال

فرض کنید، مدل‌های برنامه، رابطه‌ی one-to-many ذیل را بین یک کاربر و مقالات او دارند:

```

public class User
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<BlogPost> BlogPosts { get; set; }
}

public class BlogPost
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    [ForeignKey("UserId")]
    public virtual User User { get; set; }
    public int UserId { get; set; }
}

```

همچنین Context برنامه نیز جهت در معرض دید قرار دادن این کلاس‌ها، به نحو ذیل تشکیل شده‌است:

```

public class MyContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<BlogPost> BlogPosts { get; set; }

    public MyContext()
        : base("Connection1")
    {
        this.Database.Log = sql => Console.WriteLine(sql);
    }
}

```

بر این اساس مثالی که دو رکورد را در جداول کاربران و مقالات به صورت async ثبت می‌کند، به نحو ذیل خواهد بود:

```

private async Task<User> addUserAsync(CancellationToken cancellationToken = default(CancellationToken))
{
    using (var context = new MyContext())
    {
        var user = context.Users.Add(new User
        {
            Name = "Vahid"
        });
        context.BlogPosts.Add(new BlogPost
        {
            Content = "Test",
            Title = "Test",

```

```

        User = user
    });
    await context.SaveChangesAsync(cancellationToken);
    return user;
}
}

```

چند نکته جهت یادآوری مباحث Async

- به امضای متد واژه‌ی کلیدی async اضافه شده‌است، زیرا در بدنه‌ی آن از کلمه‌ی کلیدی await استفاده کرده‌ایم (لازم و ملزوم هستند).
- به انتهای نام متد، کلمه‌ی Async اضافه شده‌است. این مورد ضروری نیست؛ اما به یک استاندارد و قرارداد تبدیل شده‌است.
- مدل Async دات نت 4.5 [مبتنی بر Taskها](#) است. به همین جهت اینبار خروجی‌های توابع نیاز است از نوع Task باشند و آرگومان جنریک آن‌ها، بیانگر نوع مقداری که باز می‌گردانند.
- تمام متدهای الحاقی جدیدی که نامبرده شدند، دارای پارامتر اختیاری [لغو عملیات](#) نیز هستند. این مورد را با مقدار دهی cancellationToken در کدهای فوق ملاحظه می‌کنید.
- نمونه‌ای از نحوه‌ی مقدار دهی این پارامتر [در ASP.NET MVC](#) به صورت زیر می‌تواند باشد:

```

[AsyncTimeout(8000)]
public async Task<ActionResult> Index(CancellationTokn cancellationToken)

```

- در اینجا به امضای اکشن متد جاری، async اضافه شده‌است و خروجی آن نیز به نوع Task تغییر یافته است. همچنین یک پارامتر cancellationToken نیز تعریف شده‌است. این پارامتر به صورت خودکار توسط ASP.NET MVC پس از زمانیکه توسط ویژگی AsyncTimeout تعیین شده‌است، تنظیم خواهد شد. به این ترتیب، اعمال async در حال اجرا به صورت خودکار لغو می‌شوند.
- برای اجرا و دریافت نتیجه‌ی متدهای Async دار EF، نیاز است از واژه‌ی کلیدی await استفاده گردد.

استفاده کننده نیز می‌تواند متد addUserAsync را به صورت زیر فراخوانی کند:

```

var user = await addUserAsync();
Console.WriteLine("user id: {0}", user.Id);

```

شبهه به همین اعمال را نیز جهت به روز رسانی و یا حذف اطلاعات خواهیم داشت:

```

private async Task<User> updateAsync(CancellationTokn cancellationToken = default(CancellationTokn))
{
    using (var context = new MyContext())
    {
        var user1 = await context.Users.FindAsync(cancellationToken, 1);
        if (user1 != null)
            user1.Name = "Vahid N.";

        await context.SaveChangesAsync(cancellationToken);
        return user1;
    }
}

private async Task<int> deleteAsync(CancellationTokn cancellationToken =
default(CancellationTokn))
{
    using (var context = new MyContext())
    {
        var user1 = await context.Users.FindAsync(cancellationToken, 1);
        if (user1 != null)
            context.Users.Remove(user1);

        return await context.SaveChangesAsync(cancellationToken);
    }
}

```

به قطعه کد ذیل دقت کنید:

```
public async Task<List<TEntity>> GetAllAsync()
{
    return await Task.Run(() => _tEntities.ToList());
}
```

این متد از یکی از Generic repository های فله‌ای رها شده در اینترنت انتخاب شده است. به این نوع متدها که از Task.Run برای فراخوانی متدهای همزمان قدیمی مانند ToList جهت Async جلوه دادن آن‌ها استفاده می‌شود، [کدهای Async تقلبی](#) می‌گویند! این عملیات هر چند در یک ترد دیگر انجام می‌شود اما هم سربار ایجاد یک ترد جدید را به همراه دارد و هم عملیات ToList آن کاملاً blocking است. معادل صحیح Async واقعی این عملیات را در ذیل مشاهده می‌کنید:

```
private async Task<List<User>> getUsersAsync(CancellationTokentoken cancellationToken =
default(CancellationTokentoken))
{
    using (var context = new MyContext())
    {
        return await context.Users.ToListAsync(cancellationToken);
    }
}
```

متد ToListAsync یک متد Async واقعی است و نه شبیه سازی شده توسط Task.Run. متدهای Async واقعی کار با شبکه و اعمال I/O، [از ترد استفاده نمی‌کنند](#) و توسط سیستم عامل به نحو بسیار بهینه‌ای اجرا می‌گردند. برای مثال پشت صحنه‌ی متد الحاقی SaveChangesAsync به یک چنین متدی ختم می‌شود:

```
internal override async Task<long> ExecuteAsync(
//...
rowsAffected = await
command.ExecuteNonQueryAsync(cancellationToken).ConfigureAwait(continueOnCapturedContext: false);
//...
```

متد ExecuteNonQueryAsync جزو متدهای ADO.NET 4.5 است و برای اجرا نیاز به هیچ ترد جدیدی ندارد. و یا برای شبیه سازی ToListAsync با ADO.NET 4.5 و استفاده از متدهای Async واقعی آن، به یک چنین کدهایی نیاز است:

```
var connectionString = ".....";
var sql = @".....";
var users = new List<User>();

using (var cnx = new SqlConnection(connectionString))
{
    using (var cmd = new SqlCommand(sql, cnx))
    {
        await cnx.OpenAsync();
        using (var reader = await cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection))
        {
            while (await reader.ReadAsync())
            {
                var user = new User
                {
                    Id = reader.GetInt32(0),
                    Name = reader.GetString(1),
                };
                users.Add(user);
            }
        }
    }
}
```

در متد ذیل، دو Task غیرهمزمان تعریف شده‌اند و سپس با `await Task.WhenAll` درخواست اجرای همزمان و موازی آن‌ها را کرده‌ایم:

```
// multiple operations
private static async Task loadAllAsync(CancellationTokentoken cancellationTokentoken =
default(CancellationTokentoken))
{
    using (var context = new MyContext())
    {
        var task1 = context.Users.ToListAsync(cancellationTokentoken);
        var task2 = context.BlogPosts.ToListAsync(cancellationTokentoken);

        await Task.WhenAll(task1, task2);
        // use task1.Result
    }
}
```

این متد ممکن است اجرا شود؛ یا در بعضی از مواقع با استثنای ذیل خاتمه یابد:

```
An unhandled exception of type 'System.NotSupportedException' occurred in mscorlib.dll
Additional information: A second operation started on this context before a previous asynchronous
operation completed.
Use 'await' to ensure that any asynchronous operations have completed before calling another method on
this context.
Any instance members are not guaranteed to be thread safe.
```

متن استثنای ارائه شده بسیار مفید است و توضیحات کامل را به همراه دارد. در EF در طی یک Context اگر عملیات Async شروع شده‌ای خاتمه نیافته باشد، مجاز به شروع یک عملیات Async دیگر، به موازت آن نخواهیم بود. برای رفع این مشکل یا باید از چندین Context استفاده کرد و یا `await Task.WhenAll` را حذف کرده و بجای آن واژه‌ی کلیدی `await` را همانند معمول، جهت صبر کردن برای دریافت نتیجه‌ی یک عملیات غیرهمزمان استفاده کنیم.

نظرات خوانندگان

نویسنده: میثم ثوامری
تاریخ: ۱۳۹۳/۰۳/۲۹ ۱:۰

با تشکر از شما.

میخواستم بدونم متدهای async چطور در یک repository استفاده کنم
بطور مثال:

```
public class ProductRepository<T> where T : class
{
    protected DbContext _context;

    public ProductRepository(DataContext context)
    {
        _context = context;
    }

    // GetAll
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۳/۲۹ ۱:۲۳

از متدهای الحاقی جدید Async که نامبرده شدند استفاده کنید (بجای متدهای قدیمی معادل) به همراه Set برای دستیابی به
موجودیت‌ها؛ مثلاً:

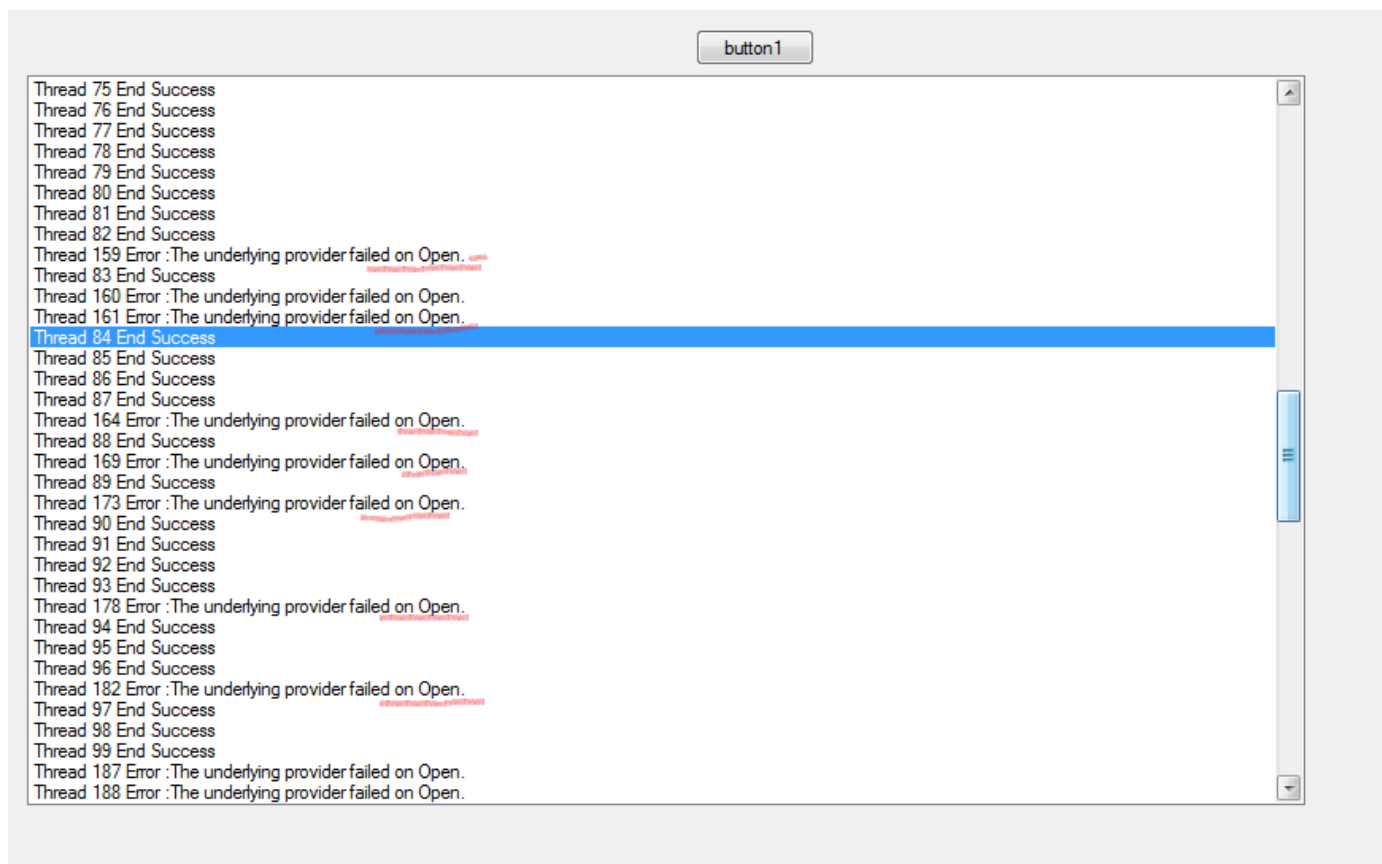
```
public async Task<List<T>> GetAllAsync()
{
    return await _dbContext.Set<T>().ToListAsync();
}
```

نویسنده: ج.زوسر
تاریخ: ۱۳۹۳/۰۴/۰۸ ۹:۴۹

مثلاً برای همچین کدی میشه از روش بالا استفاده کرد مشکل حل میشه ؟

```
{
    for (int i = 1; i <=500; i++)
    {
        ThreadPool.QueueUserWorkItem(Execute, i + 1);
    }
}

void Execute(Object obj)
{
    int thread = (int)obj;
    try
    {
        using (TestEntities ctx = new TestEntities())
        {
            int i = 1;
            foreach (var v in ctx.Customers)
            {
                Thread.Sleep(i * 1000);
                i++;
            }
            listBox1.Items.Add("Thread " + thread.ToString() + " End Success ");
        }
    }
    catch (Exception e)
    {
        listBox1.Items.Add("Thread " + thread.ToString() + " Error :" + e.Message);
    }
}
```



نویسنده:

وحید نصیری

تاریخ:

۱۰:۴۴ ۱۳۹۳/۰۴/۰۸

- بحث متدهای Async اضافه شده، ربطی به مباحث چند ریسمانی ندارد. «... متدهای Async واقعی کار با شبکه و اعمال I/O، [از ترد استفاده نمی‌کنند](#) ...» به همین جهت نسبت به حالت استفاده از تردها سربار کمتری دارند.

- در EF استثناءها چند سطحی هستند. نیاز است [inner exception](#) را جهت مشاهده‌ی اصل و علت واقعی خطا بررسی کرد. در مثال شما فقط سطح استثناء بررسی شده و نه اصل آن.

احتمالا خطای اصلی timeout است. این مورد به [مباحث قفل گذاری روی رکوردها](#) مرتبط است. تراکنش‌های طولانی همزمانی را آغاز کرده‌اید که دسترسی سایر کاربران را به جداول، تا پایان کار آن تراکنش‌ها، محدود می‌کنند.

- در کارهای چند ریسمانی برای دسترسی امن به عناصر UI، باید از روش‌های [Synchronization](#) استفاده کرد.

نویسنده:

هرمز

تاریخ:

۱۴:۵۶ ۱۳۹۳/۰۴/۰۸

سلام؛ متأسفانه من نمیتونم متود ToListAsync رو پیدا کنم. آیا باید ریفرنس خاصی اضافه کنم؟

نویسنده:

وحید نصیری

تاریخ:

۱۵:۳ ۱۳۹۳/۰۴/۰۸

- به روز رسانی خودکار وابستگی‌های پروژه:

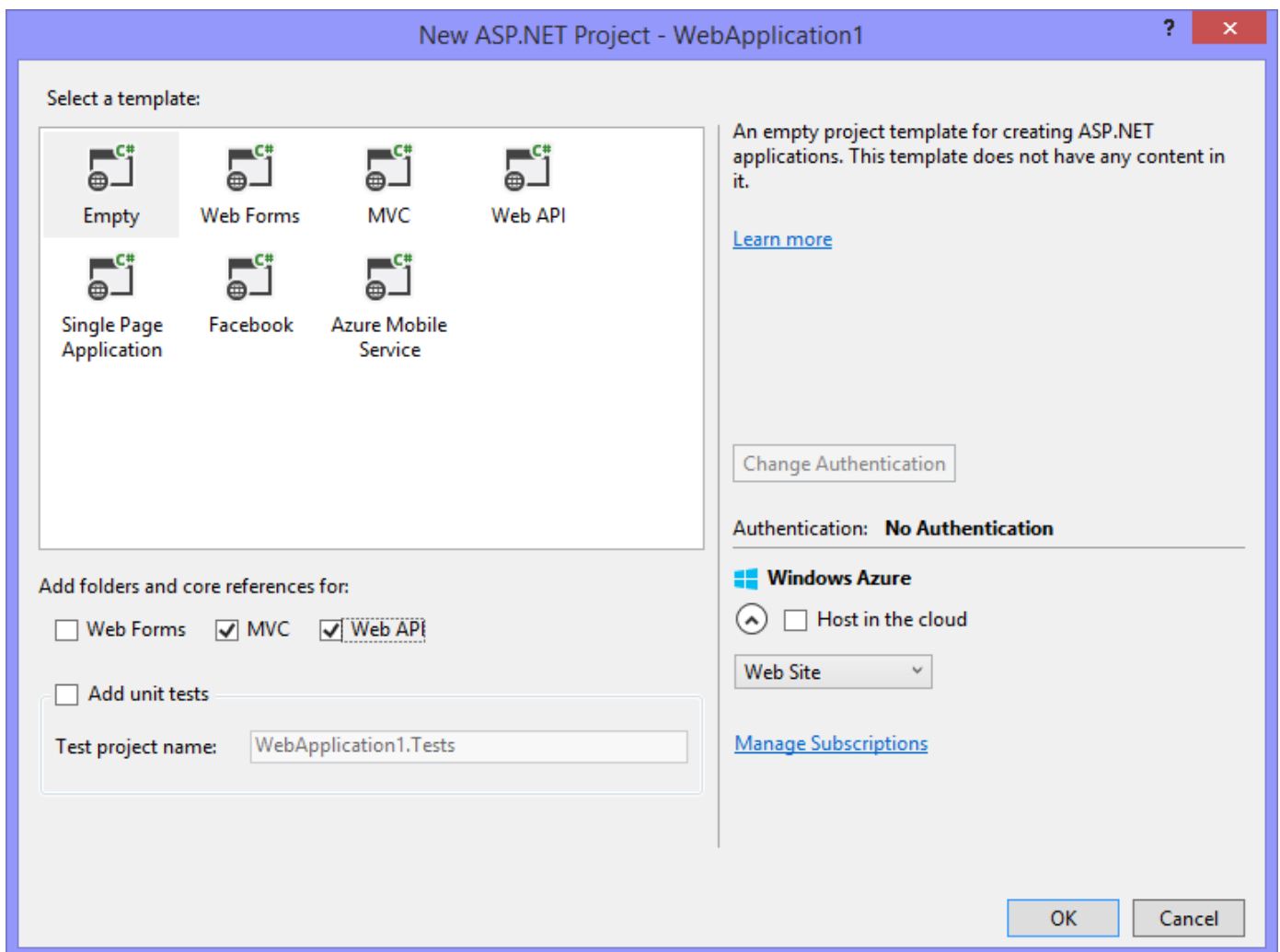
```
PM> update-package
```

- تعریف فضای نام مرتبط:

```
using System.Data.Entity;
```


فریم ورک ASP.NET Web API صرفاً برای ساخت سرویس‌های ساده‌ای که می‌شناسیم، نیست و در واقع مدل جدیدی برای برنامه نویسی HTTP است. کارهای بسیار زیادی را می‌توان توسط این فریم ورک انجام داد که در این مقاله به یکی از آنها می‌پردازم. فرض کنید می‌خواهیم یک فایل ویدیو را بصورت Asynchronous به کلاینت ارسال کنیم.

ابتدا پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب آن را MVC + Web API انتخاب کنید.



ابتدا به فایل `WebApiConfig.cs` در پوشه `App_Start` مراجعه کنید و مسیر پیش فرض را حذف کنید. برای مسیریابی سرویس‌ها از قابلیت جدید `Attribute Routing` استفاده خواهیم کرد. فایل مذکور باید مانند لیست زیر باشد.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Web API configuration and services

        // Web API routes
        config.MapHttpAttributeRoutes();
    }
}
```

```
}
}
```

حال در مسیر ریشه پروژه، پوشه جدیدی با نام *Videos* ایجاد کنید و یک فایل ویدیو نمونه بنام *sample.mp4* در آن کپی کنید. دقت کنید که فرمت فایل ویدیو در مثال جاری *mp4* در نظر گرفته شده اما به سادگی می‌توانید آن را تغییر دهید. سپس در پوشه *Models* کلاس جدیدی بنام *VideoStream* ایجاد کنید. این کلاس مسئول نوشتن داده فایل‌های ویدیویی در *OutputStream* خواهد بود. کد کامل این کلاس را در لیست زیر مشاهده می‌کنید.

```
public class VideoStream
{
    private readonly string _filename;
    private long _contentLength;

    public long FileLength
    {
        get { return _contentLength; }
    }

    public VideoStream(string videoPath)
    {
        _filename = videoPath;
        using (var video = File.Open(_filename, FileMode.Open, FileAccess.Read, FileShare.Read))
        {
            _contentLength = video.Length;
        }
    }

    public async void WriteToStream(Stream outputStream,
        HttpContext content, TransportContext context)
    {
        try
        {
            var buffer = new byte[65536];

            using (var video = File.Open(_filename, FileMode.Open, FileAccess.Read, FileShare.Read))
            {
                var length = (int)video.Length;
                var bytesRead = 1;

                while (length > 0 && bytesRead > 0)
                {
                    bytesRead = video.Read(buffer, 0, Math.Min(length, buffer.Length));
                    await outputStream.WriteAsync(buffer, 0, bytesRead);
                    length -= bytesRead;
                }
            }
        }
        catch (HttpException)
        {
            return;
        }
        finally
        {
            outputStream.Close();
        }
    }
}
```

شرح کلاس *VideoStream*

این کلاس ابتدا دو فیلد خصوصی تعریف می‌کند. یکی *_filename* که فقط-خواندنی است و نام فایل ویدیو درخواستی را نگهداری می‌کند. و دیگری *_contentLength* که سایز فایل ویدیو درخواستی را نگهداری می‌کند.

یک خاصیت عمومی بنام *FileLength* نیز تعریف شده که مقدار خاصیت *_contentLength* را بر می‌گرداند.

متد سازنده این کلاس پارامتری از نوع رشته بنام *videoPath* را می‌پذیرد که مسیر کامل فایل ویدیوی مورد نظر است. در این متد، متغیرهای *_filename* و *_contentLength* مقدار دهی می‌شوند. نکته‌ی قابل توجه در این متد استفاده از پارامتر *FileShare.Read* است که باعث می‌شود فایل مورد نظر هنگام باز شدن قفل نشود و برای پروسه‌های دیگر قابل دسترسی باشد.

در آخر متد *WriteToStream* را داریم که مسئول نوشتن داده فایل‌ها به *OutputStream* است. اول از همه دقت کنید که این متد از کلمه کلیدی *async* استفاده می‌کند بنابراین بصورت *asynchronous* اجرا خواهد شد. در بدنه این متد متغیری بنام *buffer* داریم که یک آرایه بایت با سایز 64KB را تعریف می‌کند. به بیان دیگر اطلاعات فایل‌ها را در پکیج‌های 64 کیلوبایتی برای کلاینت ارسال خواهیم کرد. در ادامه فایل مورد نظر را باز می‌کنیم (مجدداً با استفاده از *FileShare.Read*) و شروع به خواندن اطلاعات آن می‌کنیم. هر 64 کیلوبایت خوانده شده بصورت *async* در جریان خروجی نوشته می‌شود و تا هنگامی که به آخر فایل نرسیده ایم این روند ادامه پیدا می‌کند.

```
while (length > 0 && bytesRead > 0)
{
    bytesRead = video.Read(buffer, 0, Math.Min(length, buffer.Length));
    await outputStream.WriteAsync(buffer, 0, bytesRead);
    length -= bytesRead;
}
```

اگر دقت کنید تمام کد بدنه این متد در یک بلاک *try/catch* قرار گرفته است. در صورتی که با خطایی از نوع *HttpException* مواجه شویم (مثلاً هنگام قطع شدن کاربر) عملیات متوقف می‌شود و در آخر نیز جریان خروجی (*outputStream*) بسته خواهد شد. نکته دیگری که باید بدان اشاره کرد این است که کاربر حتی پس از قطع شدن از سرور می‌تواند ویدیو را تا جایی که دریافت کرده مشاهده کند. مثلاً ممکن است 10 پکیج از اطلاعات را دریافت کرده باشد و هنگام مشاهده پکیج دوم از سرور قطع شود. در این صورت امکان مشاهده ویدیو تا انتهای پکیج دهم وجود خواهد داشت.

حال که کلاس *VideoStream* را در اختیار داریم می‌توانیم پروژه را تکمیل کنیم. در پوشه کنترلرها کلاسی بنام *VideoController* بسازید. کد کامل این کلاس را در لیست زیر مشاهده می‌کنید.

```
public class VideoController : ApiController
{
    [Route("api/video/{ext}/{fileName}")]
    public HttpResponseMessage Get(string ext, string fileName)
    {
        string videoPath = HostingEnvironment.MapPath(string.Format("~/Videos/{0}.{1}", fileName,
ext));
        if (File.Exists(videoPath))
        {
            FileInfo fi = new FileInfo(videoPath);
            var video = new VideoStream(videoPath);

            var response = Request.CreateResponse();

            response.Content = new PushStreamContent((Action<Stream, HttpContext,
TransportContext>)video.WriteToStream,
                new MediaTypeHeaderValue("video/" + ext));

            response.Content.Headers.Add("Content-Disposition", "attachment;filename=" +
fi.Name.Replace(" ", ""));
            response.Content.Headers.Add("Content-Length", video.FileLength.ToString());

            return response;
        }
        else
        {
            return Request.CreateResponse(HttpStatusCode.NotFound);
        }
    }
}
```

شرح کلاس VideoController

همانطور که می‌بینید مسیر دستیابی به این کنترلر با استفاده از قابلیت *Attribute Routing* تعریف شده است.

```
[Route("api/video/{ext}/{fileName}")]
```

نمونه ای از یک درخواست که به این مسیر نگاشت می‌شود:

```
api/video/mp4/sample
```

بنابراین این مسیر فرمت و نام فایل مورد نظر را بدین شکل می‌پذیرد. در نمونه جاری ما فایل *sample.mp4* را درخواست کرده ایم.

متد *Get* این کنترلر دو پارامتر با نام‌های *ext* و *fileName* را می‌پذیرد که همان فرمت و نام فایل هستند. سپس با استفاده از کلاس *HostingEnvironment* سعی می‌کنیم مسیر کامل فایل درخواست شده را بدست آوریم.

```
string videoPath = HostingEnvironment.MapPath(string.Format("~/Videos/{0}.{1}", fileName, ext));
```

استفاده از این کلاس با *Server.MapPath* تفاوتی نمی‌کند. در واقع خود *Server.MapPath* نهایتاً همین کلاس *HostingEnvironment* را فراخوانی می‌کند. اما در کنترلرهای *Web Api* به کلاس *Server* دسترسی نداریم. همانطور که مشاهده می‌کنید فایل مورد نظر در پوشه *Videos* جستجو می‌شود، که در ریشه سایت هم قرار دارد. در ادامه اگر فایل درخواست شده وجود داشت و هله جدیدی از کلاس *VideoStream* می‌سازیم و مسیر کامل فایل را به آن پاس می‌دهیم.

```
var video = new VideoStream(videoPath);
```

سپس آبجکت پاسخ را و هله سازی می‌کنیم و با استفاده از کلاس *PushStreamContent* اطلاعات را به کلاینت می‌فرستیم.

```
var response = Request.CreateResponse();
response.Content = new PushStreamContent((Action<Stream, HttpContext,
TransportContext>)video.WriteToStream, new MediaTypeHeaderValue("video/" + ext));
```

کلاس *PushStreamContent* در فضای نام *System.Net.Http* وجود دارد. همانطور که می‌بینید امضای *Action* پاس داده شده، با امضای متد *WriteToStream* در کلاس *VideoStream* مطابقت دارد.

در آخر دو *Header* به پاسخ ارسالی اضافه می‌کنیم تا نوع داده ارسالی و سایز آن را مشخص کنیم.

```
response.Content.Headers.Add("Content-Disposition", "attachment;filename=" + fileName);
response.Content.Headers.Add("Content-Length", video.FileLength.ToString());
```

افزودن این دو مقدار مهم است. در صورتی که این *Header*ها را تعریف نکنید سایز فایل دریافتی و مدت زمان آن نامعلوم خواهد بود که تجربه کاربری خوبی بدست نمی‌دهد. نهایتاً هم آبجکت پاسخ را به کلاینت ارسال می‌کنیم. در صورتی هم که فایل مورد نظر در پوشه *Videos* پیدا نشود پاسخ *NotFound* را بر می‌گردانیم.

```
if(File.Exists(videoPath))
{
    // removed for bravery
}
else
{
    return Request.CreateResponse(HttpStatusCode.NotFound);
}
```

خوب، برای تست این مکانیزم نیاز به یک کنترلر *MVC* و یک *View* داریم. در پوشه کنترلرها کلاسی بنام *HomeController* ایجاد کنید که با لیست زیر مطابقت داشته باشد.

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index()
    {
```

```

    return View();
}
}

```

نمای این متد را بسازید (با کلیک راست روی متد Index و انتخاب گزینه Add View) و کد آن را مطابق لیست زیر تکمیل کنید.

```

<div>
  <div>
    <video width="480" height="270" controls="controls" preload="auto">
      <source src="/api/video/mp4/sample" type="video/mp4" />
      Your browser does not support the video tag.
    </video>
  </div>
</div>

```

همانطور که مشاهده می‌کنید یک المنت ویدیو تعریف کرده ایم که خواص طول، عرض و غیره آن نیز مقدار دهی شده اند. زیر تگ source متنی درج شده که در صورت لزوم به کاربر نشان داده می‌شود. گرچه اکثر مرورگرهای مدرن از المنت ویدیو پشتیبانی می‌کنند. تگ سورس فایل با مشخصات sample.mp4 را درخواست می‌کند و نوع آن را نیز video/mp4 مشخص کرده ایم.

اگر پروژه را اجرا کنید می‌بینید که ویدیو مورد نظر آماده پخش است. برای اینکه ببینید چطور داده‌های ویدیو در قالب پکیج‌های 64 کیلو بایتی دریافت می‌شوند از ابزار مرورگر تان استفاده کنید. مثلاً در گوگل کروم F12 را بزنید و به قسمت Network بروید. صفحه را یکبار مجدداً بارگذاری کنید تا ارتباطات شبکه مانیتور شود. اگر به المنت sample دقت کنید می‌بینید که با شروع پخش ویدیو پکیج‌های اطلاعات یکی پس از دیگری دریافت می‌شوند و اطلاعات ریز آن را می‌توانید مشاهده کنید.

پروژه نمونه به این مقاله ضمیمه شده است. قابلیت Package Restore فعال شده و برای صرفه جویی در حجم فایل، تمام پکیج‌ها و محتویات پوشه bin حذف شده اند. برای تست بیشتر می‌توانید فایل sample.mp4 را با فایل حجیم‌تر جایگزین کنید تا نحوه دریافت اطلاعات را با روشی که در بالا بدان اشاره شد مشاهده کنید.

[AsyncVideoStreaming.rar](#)

نظرات خوانندگان

نویسنده: علی

تاریخ: ۱۷:۵۵ ۱۳۹۳/۰۶/۱۰

سلام

امروز این مطلب رو دیدم و چند روز پیش خودم انجامش داده بودم. نکته‌ی عجیب اینه که وقتی از این حالت برای پخش ویدئو استفاده می‌کنیم، پلیر میزان فریم‌های بافر شده از ویدیو را نمایش نمیده، در واقع کاربر متوجه نمیشه که تا کجای فیلم از سرور دانلود شده (در صورتی که در حالت پخش مستقیم ویدیو از لینک مستقیم اینگونه نیست).

ممنون میشم اگر به این سه سوال پاسخ بدین :

- 1- مزیت این روش نسبت به روشی که از لینک مستقیم فایل ویدیو استفاد می‌کنیم چیه ؟
- 2- آیا استفاده از این روش باری بر روی پردازنده، رم و... سرور اضافه می‌کنه ؟
- 3- برای پخش ویدیو از این روش استفاده کنیم بهتره یا از لینک مستقیم ؟

با تشکر