عنوان: **معرفی Reactive extensions**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۲۳ ۱۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Asynchronous Programming, Reactive Extensions, Rx

Reactive extensions یا به صورت خلاصه Rx، کتابخانه‌ی سورس باز تهیه شده‌ای توسط مایکروسافت است که اگر بخواهیم آن‌را 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 نتی مایکروسافت در سال‌های اخیر به شما می‌رود؛ تا حدی که معادل‌های بسیاری از آن برای زبان‌های دیگر مانند CPP، Python، JavaScript، Java 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 نمائید:

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

نصب آن از طریق نیوگت، به صورت خودکار کلیه وابستگی‌های مرتبط با آن‌را 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 کوئری 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 آن‌را تحت کنترل قرار داد. برای مثال در اینجا هربار 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 صرفا 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 درنظر 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 (مثلا 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 IObserver 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 IObserver را پیاده سازی کند.

```
class Observer : IObserver<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 مجموعه‌ی در حال پردازش، یکبار اجرا می‌شود. البته نیازی به پیاده سازی صریح این اینترفیس نیست و توسط متد توکار Observer.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 ایی، 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 IObserver به ازای هر آیتم 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 آیتم‌های 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 دقیقا 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 IObserver (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 تردی 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 درنظر 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.Subscribeدر 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 callbackهای 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 شده‌است یا به عبارتی در ترد اصلی برنامه؛ در غیر اینصورت برنامه کرش خواهد کرد. مدیریت این مساله نیز در Rx بسیار ساده‌است. 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 callbackهای مشاهده‌گر است.

و یا اگر از WinForms استفاده می‌کنید، ابتدا بسته‌ی Rx خاص آن‌را نصب کنید:

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

و سپس ترد اجرای callbackها را SynchronizationContext.Current مشخص نمائید:

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

یک نکته

در Rx فرض می‌شود که کوئری شما زمانبر است و callbackهای مشاهده‌گر سریع عمل می‌کنند. بنابراین هدف از callbackهای 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 آن‌را به کمک متد ObserveOn، بر روی ترد اصلی برنامه انجام دهید.

نظرات خوانندگان

نویسنده: ژوپیتر
تاریخ: ۱۳۹۳/۰۲/۲۹ ۱۱:۳۷

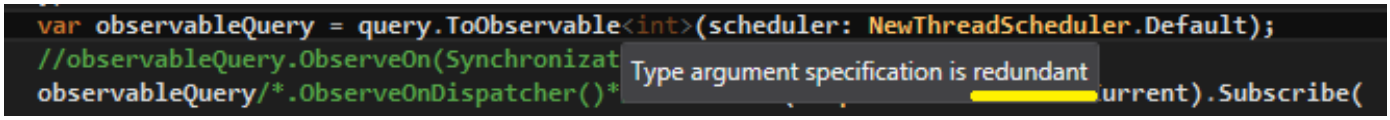
به نظرم باید نوع آرگومان اینجا مشخص باشه:

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

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

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۲۹ ۱۱:۵۲

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



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: ژوپیتر
تاریخ: ۱۳۹۳/۰۲/۲۹ ۱۲:۱۵

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