عنوان: مروری بر کاربردهای Action و Func - قسمت چهارم
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۰۵ ۲۳:۵۳
آدرس: www.dotnettips.info
برچسب‌ها: C#, Refactoring

طراحی API برنامه توسط Actionها

روش مرسوم طراحی Fluent interfaces ، جهت ارائه روش ساخت اشیاء مسطح به کاربران بسیار مناسب هستند. اما اگر سعی در تهیه API عمومی برای کار با اشیاء چند سطحی مانند معرفی فایل‌های XML توسط کلاس‌های سی شارپ کنیم، اینبار Fluent interfaces آنچنان قابل استفاده نخواهند بود و نمی‌توان این نوع اشیاء را به شکل روانی با کنار هم قرار دادن زنجیر وار متدها تولید کرد. برای حل این مشکل روش طراحی خاصی در نگارش‌های اخیر NHibernate معرفی شده است به نام loquacious interface که این روزها در بسیاری از APIهای جدید شاهد استفاده از آن هستیم و در ادامه با پشت صحنه و طرز تفکری که در حین ساخت این نوع API وجود دارد آشنا خواهیم شد.

در ابتدا کلاس‌های مدل زیر را در نظر بگیرید که قرار است توسط آن‌ها ساختار یک جدول از کاربر دریافت شود:

```
using System;
using System.Collections.Generic;

namespace Test
{
    public class Table
    {
        public Header Header { set; get; }
        public IList<Cell> Cells { set; get; }
        public float Width { set; get; }
    }

    public class Header
    {
        public string Title { set; get; }
        public DateTime Date { set; get; }
        public IList<Cell> Cells { set; get; }
    }

    public class Cell
    {
        public string Caption { set; get; }
        public float Width { set; get; }
    }
}
```

در روش طراحی loquacious interface به ازای هر کلاس مدل، یک کلاس سازنده ایجاد خواهد شد. اگر در کلاس جاری، خاصیتی از نوع کلاس یا لیست باشد، برای آن نیز کلاس سازنده خاصی درنظر گرفته می‌شود و این روند ادامه پیدا می‌کند تا به خواصی از انواع ابتدایی مانند int و string برسیم:

```
using System;
using System.Collections.Generic;

namespace Test
{
    public class TableApi
    {
        public Table CreateTable(Action<TableCreator> action)
        {
            var creator = new TableCreator();
            action(creator);
            return creator.TheTable;
        }
    }

    public class TableCreator
    {
        readonly Table _theTable = new Table();
        internal Table TheTable
        {
```

```
            get { return _theTable; }
        }

        public void Width(float value)
        {
            _theTable.Width = value;
        }

        public void AddHeader(Action<HeaderCreator> action)
        {
            _theTable.Header = ...
        }

        public void AddCells(Action<CellsCreator> action)
        {
            _theTable.Cells = ...
        }
    }
}
```

نقطه آغازین API ایی که در اختیار استفاده کنندگان قرار می‌گیرد با متد CreateTable ایی شروع می‌شود که ساخت شیء جدول را به ظاهر توسط یک Action به استفاده کننده واگذار کرده است، اما توسط کلاس TableCreator او را مقید و راهنمایی می‌کند که چگونه باید اینکار را انجام دهد.

همچنین در بدنه متد CreateTable، نکته نحوه دریافت خروجی از Action ایی که به ظاهر خروجی خاصی را بر نمی‌گرداند نیز قابل مشاهده است.

همانطور که عنوان شد کلاس‌های xyzCreator تا رسیدن به خواص معمولی و ابتدایی پیش می‌روند. برای مثال در سطح اول چون خاصیت عرض از نوع float است، صرفا با یک متد معمولی دریافت می‌شود. دو خاصیت دیگر نیاز به Creator دارند تا در سطحی دیگر برای آن‌ها سازنده‌های ساده‌تری را طراحی کنیم.

همچنین باید دقت داشت که در این طراحی تمام متدها از نوع void هستند. اگر قرار است خاصیتی را بین خود رد و بدل کنند، این خاصیت به صورت internal تعریف می‌شود تا در خارج از کتابخانه قابل دسترسی نباشد و در intellisense ظاهر نشود.

مرحله بعد، ایجاد دو کلاس HeaderCreator و CellsCreator است تا کلاس TableCreator تکمیل گردد:

```
using System;
using System.Collections.Generic;

namespace Test
{
    public class CellsCreator
    {
        readonly IList<Cell> _cells = new List<Cell>();
        internal IList<Cell> Cells
        {
            get { return _cells; }
        }

        public void AddCell(string caption, float width)
        {
            _cells.Add(new Cell { Caption = caption, Width = width });
        }
    }

    public class HeaderCreator
    {
        readonly Header _header = new Header();
        internal Header Header
        {
            get { return _header; }
        }

        public void Title(string title)
        {
            _header.Title = title;
        }

        public void Date(DateTime value)
        {
            _header.Date = value;
        }

        public void AddCells(Action<CellsCreator> action)
```

```
        {
            var creator = new CellsCreator();
            action(creator);
            _header.Cells = creator.Cells;
        }
    }
}
```

نحوه ایجاد کلاس‌های Builder و یا Creator این روش بسیار ساده و مشخص است:
مقدار هر خاصیت معمولی توسط یک متد ساده void دریافت خواهد شد.
هر خاصیتی که اندکی پیچیدگی داشته باشد، نیاز به یک Creator جدید خواهد داشت.
کار هر Creator بازگشت دادن مقدار یک شیء است یا نهایتا ساخت یک لیست از یک شیء. این مقدار از طریق یک خاصیت internal بازگشت داده می‌شود.

البته عموما بجای معرفی مستقیم کلاس‌های Creator از یک اینترفیس معادل آن‌ها استفاده می‌شود. سپس کلاس Creator را internal تعریف می‌کنند تا خارج از کتابخانه قابل دسترسی نباشد و استفاده کننده نهایی فقط با توجه به متدهای void تعریف شده در interface کار تعریف اشیاء را انجام خواهد داد.

در نهایت، مثال تکمیل شده ما به شکل زیر خواهد بود:

```
using System;
using System.Collections.Generic;

namespace Test
{
    public class TableCreator
    {
        readonly Table _theTable = new Table();
        internal Table TheTable
        {
            get { return _theTable; }
        }

        public void Width(float value)
        {
            _theTable.Width = value;
        }

        public void AddHeader(Action<HeaderCreator> action)
        {
            var creator = new HeaderCreator();
            action(creator);
            _theTable.Header = creator.Header;
        }

        public void AddCells(Action<CellsCreator> action)
        {
            var creator = new CellsCreator();
            action(creator);
            _theTable.Cells = creator.Cells;
        }
    }

    public class CellsCreator
    {
        readonly IList<Cell> _cells = new List<Cell>();
        internal IList<Cell> Cells
        {
            get { return _cells; }
        }

        public void AddCell(string caption, float width)
        {
            _cells.Add(new Cell { Caption = caption, Width = width });
        }
    }

    public class HeaderCreator
    {
        readonly Header _header = new Header();
```

```
        internal Header Header
        {
            get { return _header; }
        }

        public void Title(string title)
        {
            _header.Title = title;
        }

        public void Date(DateTime value)
        {
            _header.Date = value;
        }

        public void AddCells(Action<CellsCreator> action)
        {
            var creator = new CellsCreator();
            action(creator);
            _header.Cells = creator.Cells;
        }
    }
}
```

نحوه استفاده از این طراحی نیز جالب توجه است:

```
var data = new TableApi().CreateTable(table =>
            {
                table.Width(1);
                table.AddHeader(header=>
                {
                    header.Title("new rpt");
                    header.Date(DateTime.Now);
                    header.AddCells(cells=>
                    {
                        cells.AddCell("cell 1", 1);
                        cells.AddCell("cell 2", 2);
                    });
                });
                table.AddCells(tableCells=>
                {
                    tableCells.AddCell("c 1", 1);
                    tableCells.AddCell("c 2", 2);
                });
            });
```

این نوع طراحی مزیت‌های زیادی را به همراه دارد:

الف) ساده سازی طراحی اشیاء چند سطحی و تو در تو

ب) امکان درنظر گرفتن مقادیر پیش فرض برای خواص

ج) ساده‌تر سازی تعاریف لیست‌ها

د) استفاده کنندگان در حین استفاده نهایی و تعریف اشیاء به سادگی می‌توانند کدنویسی کنند (مثلا سلول‌ها را با یک حلقه اضافه کنند).

ه) امکان بهتر استفاده از امکانات Intellisense . برای مثال فرض کنید یکی از خاصیت‌هایی که قرار است برای آن Creator درست کنید یک interface را می‌پذیرد. همچنین در برنامه خود چندین پیاده سازی کمکی از آن نیز وجود دارد. یک روش این است که مستندات قابل توجهی را تهیه کنید تا این امکانات توکار را گوشزد کند؛ روش دیگر استفاده از طراحی فوق است. در اینجا در کلاس Creator ایجاد شده چون امکان معرفی متد وجود دارد، می‌توان امکانات توکار را توسط این متدها نیز معرفی کرد و به این ترتیب Intellisense تبدیل به راهنمای اصلی کتابخانه شما خواهد شد.

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۶/۰۶ ۱۷:۳۸

این تکنیک و مقاله، یکی از مطالب Must Read سال هست. به شخصه از این تکنیک در توسعه‌ی کامپوننت‌های ASP.NET MVC استفاده می‌کنم. کلاً تکنیک Fluent که برادر نصیری فعلاً در دو مقاله به اون پرداختند، انعطاف پذیری بسیاری به برنامه‌ها میده. مثلاً شبیه سازی روال RowDataBound کنترل GridView در Web Forms، در بستر MVC با استفاده از یک Action.
به نظر من کمبودی که ASP.NET MVC در حال حاضر داره، داشتن مجموعه ای غنی از کامپوننت‌های توکار هست که فکر می‌کنم در نسخه‌های آینده، مایکروسافت این نقیصه رو بر طرف می‌کنه، شاید با مشارکت شرکت‌های دیگه مثل Telerik.