عنوان: **اجرای متد از طریق Reflection**
نویسنده: سیدمجتبی حسینی
تاریخ: ۱۳۹۱/۰۷/۱۷ ۱۴:۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, Reflection

برای اجرای متد درون یک کلاس از طریق Reflection ابتدا نوع آن کلاس را به دست می‌آوریم و سپس از طریق کلاس Activator.CreateInstance یک نمونه از آن کلاس را ساخته و در متغیری از نوع object ذخیره کرده و با استفاده از GetMethod اطلاعات متد مورد نظر خود را در متغیری ذخیره کرده و سپس از طریق دستور Invoke آن متد را اجرا می‌کنیم. دستور Invoke دو سربارگذاری دارد که در یک نوع از آن، متغیر حاوی نمونه کلاس و پارامترهای متد مورد نظر، در قالب یک آرایه از نوع object، به عنوان آرگومان پذیرفته می‌شود. با امضای زیر

```
public Object Invoke(Object obj, Object[] parameters)
```

به مثال زیر که چگونگی این عملیات را شرح می‌دهد، توجه کنید:

```
public class TestMath
    {
        public int Squar(int i)
        {
            return i*i;
        }
    }

static void Main(string[] args)
        {
            Type type = typeof (TestMath);//به دست آوردن نوع کلاس
            object obj = Activator.CreateInstance(type);//ساختن نمونه‌ای از نوع مورد نظر
            MethodInfo methodInfo = type.GetMethod("Squar");//یافتن اطلاعات متد مورد نظر
            Console.WriteLine(methodInfo.Invoke(obj, new object[] { 100 }));// و نمایش نتیجه object[] ارسال عدد 100 به صورت
            Console.Read();
        }
```

توجه کنید که دو متد GetMethod و Invoke در فضای نام System.Reflection قرار دارند.

روش دیگر

در شیوه دیگر برای انجام این کار، نیازی به استفاده از GetMethod و Invoke نیست و فراخوانی متد مورد نظر بسیار شبیه فراخوانی عادی متدهاست و نیازی به ساخت متغیر ویژه‌ای از نوع object[] برای ارسال پارامترها نیست. برای انجام این کار فقط کافیست نوع متغیری که نوع نمونه‌سازی شده را نگه‌می‌دارد (در اینجا نمونه ای از کلاس را نگه‌می‌دارد) به صورت dynamic باشد:

```
static void Main(string[] args)
        {
            Type type = typeof (TestMath);
            dynamic obj = Activator.CreateInstance(type);
            Console.WriteLine(obj.Square(100));
            Console.Read();
        }
```

توجه کنید که بعد از تعریف obj، با درج نقطه در کنار آن، منوی Code Insight متد Square را شامل نمی‌شود اما کامپایلر آن را می‌پذیرد.

نظرات خوانندگان

نویسنده: KishIsland
تاریخ: ۱۳۹۱/۱۲/۱۵ ۱۹:۴۷

برای تست کد بالا من یک کلاس بشکل زیر تعریف کردم:

```
class Test
    {
        public void func1()
        {
            Console.WriteLine("Hello World!");
        }
    }
```

و در قسمت main برنامه فراخوانی رو بشکل زیر نوشتم مطابق روش دوم ولی برنامه Exception داد:

```
Type type = typeof(Test);
            dynamic obj = Activator.CreateInstance(type);
            Console.WriteLine(obj.func1());
```

علت بروز استثناء void بودن متد هست.می خواستم بدونم برای مواقعی که متد بصورت void تعریف شده چکار باید کرد؟
سپاس

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۱۲/۱۵ ۲۱:۲۰

در این حالت بجای

```
Console.WriteLine(obj.func1());
```

فقط کافی هست بنویسید

```
obj.func1();
```

عنوان: **چگونگی دسترسی به فیلد و خاصیت غیر عمومی**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۳/۱۵ ۱۶:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Reflection, read-only

یک از ابتدایی‌ترین مواردی که در یادگیری دات نت آموزش داده می‌شود مباحث مربوط به کپسوله سازی است. برای مثال فیلدها و خواص Private که به صورت خصوصی هستند یا Protected هستند از خارج کلاس قابل دسترسی نیستند. برای دسترسی به این کلاس‌ها باید از خواص یا متدهای عمومی استفاده کرد.

```
public class Book
    {
        private int code = 10;

        public int GetCode()
        {
            return code;
        }
    }
```

یا فیلدها و خواصی که به صورت فقط خواندنی هستند،(RealOnly) امکان تغییر مقدار برای اون‌ها وجود ندارد. برای مثال کد پایین کامپایل نخواهد شد.

```
public class Book
    {
        private readonly int code = 10;

        public int GetCode()
        {
            return code = 20;
        }
    }
```

اما در دات نت با استفاده از Reflectionها می‌تونیم تمام قوانین بالا رو نادیده بگیریم. یعنی می‌تونیم هم به خواص و فیلدهای غیر عمومی کلاس دسترسی پیدا کنیم و هم می‌تونیم مقدار فیلدهای فقط خواندنی رو تغییر بدیم. به مثال‌های زیر دقت کنید.
#مثال اول

```
using System.Reflection;

 public class Book
 {
        private int code = 10;
 }

 public class Program
 {
        static void Main( string[] args )
        {
            Book book = new Book();
            var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic |
BindingFlags.Instance );
            codeField.SetValue( book, 20 );
            var value = codeField.GetValue( book );
        }
    }
```

ابتدا یک کلاس که دارای یک متغیر به نام کد است ساخته ایم که مقدار 10 را دارد. فیلد به صورت private است. بعد از اجرا به راحتی مقدار Code را به دست می‌آوریم.

```
class Program
{
   static void Main( string[] args )
   {
      Book book = new Book();
      var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic | BindingFlags.Instance );
      var value = codeField.GetValue( book );
   }
}
```

value 10

حتی امکان تغییر مقدار فیلد private هم امکان پذیر است.

```
class Program
{
   static void Main( string[] args )
   {
      Book book = new Book();
      var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic | BindingFlags.Instance );
      codeField.SetValue( book, 100 );
      var value = codeField.GetValue( book );
   }
}
```

value 100

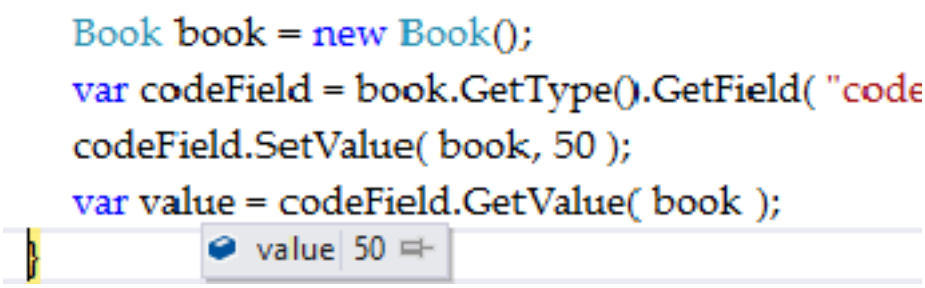
#مثال دوم.
در این مثال قصد داریم مقدار یک فیلد، از نوع فقط خواندنی رو تغییر دهیم.

```
using System.Reflection;

 public class Book
 {
        private readonly int code = 10;
 }

 public class Program
 {
        static void Main( string[] args )
        {
            Book book = new Book();
            var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic |
BindingFlags.Instance );
            codeField.SetValue( book, 50);
            var value = codeField.GetValue( book );
        }
    }
```

بعد از اجرا مقدار متغیر code به 50 تغییر می‌یابد.



مطالب تکمیلی

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۳/۱۷ ۱۰:۱۰

البته مباحث Reflection، تابع سطح دسترسی کد فراخوان است (همان لینک آخر بحث جهت تاکید بیشتر و همچنین تنویر مقدمه):

« Security Considerations for Reflection »

برای نمونه در حالت medium trust، گزینه ReflectionPermission غیرفعال است.

برای آزمایش این مسایل می‌شود از دو برنامه Permview و Permcalc استفاده کرد.

نویسنده: سالار خلیل زاده
تاریخ: ۱۳۹۲/۰۳/۱۸ ۹:۱۸

ReflectionMagic جهت همین کار طراحی شده

http://nuget.org/packages/ReflectionMagic

نویسنده: reza
تاریخ: ۱۳۹۳/۰۵/۲۵ ۱۷:۹

آیا می‌توان به کمک رفلکشن به خصوصیتی که مثلا Set ندارد مقدار دهی کرد. بعنوان مثال

```
class Program
{
  static void Main(string[] args)
  {
      Book book = new Book();
      var codeprop = book.GetType().GetProperty("Code", BindingFlags.Public | BindingFlags.Instance);
      codeprop.SetValue(book, 20, null);
      var value = codeprop.GetValue(book, null);
  }
}

public class Book
{
  public int code;
  public int Code
  {
      get { return code; }
  }
}
```

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۵/۲۶ ۱۴:۰

خیر! با یک ArgumentException و پیغام Property set method not found مواجه خواهید شد. اما در مثال بالا می‌توان مقدار فیلد code را تغییر داد که در نتیجه خاصیت Code نیز مقدار جدید را برگشت می‌دهد.

عنوان: **فراخوانی داینامیک اسمبلی‌های آفیس در برنامه**
نویسنده: مهران ر
تاریخ: ۱۳۹۲/۱۱/۰۷ ۱۶:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Reflection, Dynamic Programming

در بسیاری از پروژه‌های دات نت، نیاز به استفاده از فایلهای نرم افزار آفیس، از قبیل ورد و اکسل و ... وجود دارد. برای مثال گاهی لازم است اطلاعات یک گرید، یا هر منبع داده‌ای، در قالب اکسل به کاربر نمایش داده شود. بدین شکل که این فایلها در زمان اجرا ساخته شده و به کاربر نمایش داده شود .حال فرض کنید شما روی سیستم خودتان Office2007 را نصب کرده اید و به اسمبلی‌های این ورژن دسترسی دارید. البته بدون نیاز به نصب آفیس نیز میتوان به این توابع دسترسی داشت و از آنها در برنامه استفاده کرد که همان استفاده از Primary Interop Assemblies میباشد. مشکلی که ممکن است پیش آید این است که در کامپیوترهای کاربران ممکن است ورژن‌های مختلفی از آفیس نصب باشد مانند 2003 -2007-2010-2013 و اگر با ورژن اسمبلی‌هایی که فراخوانی‌های فایلهای اکسل از طریق آن انجام شده باشد متفاوت باشد، برنامه اجرا نمی‌شود .

در حالت معمول برای نمایش یک فایل آفیس مثل اکسل در برنامه، ابتدا اسمبلی مربوطه را (اکسل در این مثال) که به نام Microsoft.Office.Interop.Excel میباشد به اسمبلی‌های برنامه اضافه کرده (از طریق add reference) و برای نمایش یک فایل اکسل در زمان اجرا از کدهای زیر استفاده مینماییم :

```
try
            {
                var application =
(Microsoft.Office.Interop.Excel.Application)Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Appl
ication"));
               Workbook  wrkBook;
                var  wbk = application.Workbooks;
                 wrkBook = wbk.Add();
                wrkBook.Activate();
                application.Visible = true;
             }
            catch (Exception ex)
            {
                Error(ex.Message);
            }
```

حال اگر آفیس 2010 به عنوان مثال در سیستم ما نصب باشد، ورژن این اسمبلی 14 می‌باشد و اگر این برنامه را در کامپیوتر کلاینتی که آفیس 2007 بر روی آن نصب باشد انتشار دهیم اجرا نمیشود. برای حل این مشکل بنده با استفاده از روش dynamic این موضوع را حل کردم و بنظر می‌رسد راه‌های دیگری نیز برای حل آن وجود داشته باشد.

در این روش با توجه به ورژن آفیسی که بر روی سیستم کاربر نصب شده اسمبلی مربوطه را از سیستم کاربر لود کرده و فایلهای آفیس را اجرا مینماییم. در ابتدا تشخیص میدهیم چه ورژنی از آفیس بر روی سیستم کاربر نصب است :

```
string strVersion = null;
               dynamic objEApp =Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
               if (objEApp.Version == "12.0")
               {
                   strVersion = "2007";
               }
               else if (objEApp.Version == "14.0")
               {
                   strVersion = "2010";
               }
```

روش دیگر برای انجام اینکار استفاده از اطلاعات رجیستری ویندوز است :

```
string strEVersionSubKey = "\\Excel.Application\\CurVer";
            string strValue = null; //Value Present In Above Key
            string strVersion = null; //Determines Excel Version
            RegistryKey rkVersion = null; //Registry Key To Determine Excel Version
            rkVersion = Registry.ClassesRoot.OpenSubKey(strEVersionSubKey, false); //Open Registry Key
```

```
            if ((rkVersion != null)) //If Key Exists
            {
                strValue = (string)rkVersion.GetValue(string.Empty); //Get Value
                strValue = strValue.Substring(strValue.LastIndexOf(".") + 1); //Store Value
                switch (strValue) //Determine Version
                {
                    case "11":
                        strVersion = "2003";
                        break;

                    case "12":
                        strVersion = "2007";
                        break;

                    case "14":
                        strVersion = "2010";
                        break;

                }
```

حال با استفاده از تابع assembly.load() اسمبلی مورد نیاز را لود کرده و در برنامه استفاده مینماییم :

```
if (strVersion == "2007")
                {
                    string strAssemblyOff2007 =
                        "Microsoft.Office.Interop.Excel, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c";
                    try
                    {
                        Assembly xslExcelAssembly = Assembly.Load(strAssemblyOff2007); //Load Assembly
                        Type type = xslExcelAssembly.GetTypes().Single(t => t.Name ==
"ApplicationClass");
                        dynamic application = Activator.CreateInstance(type);
                        var workbooks = application.Workbooks;
                        var workbook = workbooks.Add();
                        var worksheet = workbook.Worksheets[1];
                        workbook.Activate();
                        application.Visible = true;
                    }
                    catch (Exception ex)
                    {
                    }
                }
```

در این حالت بدون اینکه بدانیم بر روی سیستم کاربر چه ورژنی از آفیس نصب است میتوان فایلهای آفیس را در زمان اجرا لود کرده و استفاده کرد .

نظرات خوانندگان

نویسنده: حسین
تاریخ: ۱۳۹۲/۱۱/۱۹ ۲۰:۳۳

با تشکر از مقاله مفید شما
من یه بار توی یه پروژه یک تمپلت ورد ایجاد کردم و توش انواع اقسام چارت‌ها و جدول‌ها رو توش رسم کردم و کلی هم روش کار کردم تا گزارش خوبی از کار در بیارد
واقعا اطلاع نداشتم با ورژن‌ها مختلف اجرا نمیشه!
الان عذاب وجدان گرفتم :)

عنوان: استفاده ازExpressionها جهت ایجاد Strongly typed view در ASP.NET MVC
نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۴/۱۵ ۰:۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Reflection, Kendo UI, jqGrid

مدل زیر را در نظر بگیرید:

```
/// <summary>
    ///
    /// </summary>
    public class CompanyModel
    {
        /// <summary>
        /// Table Identity
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// Company Name
        /// </summary>
        [DisplayName("نام شرکت")]
        public string CompanyName { get; set; }

        /// <summary>
        /// Company Abbreviation
        /// </summary>
        [DisplayName("نام اختصاری شرکت")]
        public string CompanyAbbr { get; set; }

    }
```

از View زیر جهت نمایش لیستی از شرکت‌ها متناظر با مدل جاری استفاده میشود:

```
@{
    const string viewTitle = "شرکت ها";
    ViewBag.Title = viewTitle;
    const string gridName = "companies-grid";
}
<div class="col-md-12">
    <div class="form-panel">
        <header>
            <div class="title">
                <i class="fa fa-book"></i>
                @viewTitle
            </div>
        </header>
        <div class="panel-body">
            <div id="@gridName">
            </div>
        </div>
    </div>
</div>
</div>
@section scripts
{
    <script type="text/javascript">
        $(document).ready(function () {
            $("#@gridName").kendoGrid({
                dataSource: {
                    type: "json",
                    transport: {
                        read: {
                            url: "@Html.Raw(Url.Action(MVC.Company.CompanyList()))",
                            type: "POST",
                            dataType: "json",
                            contentType: "application/json"
                        }
                    },
                    schema: {
                        data: "Data",
                        total: "Total",
                        errors: "Errors"
                        }
                    },
                    pageSize: 10,
```

```
                    serverPaging: true,
                    serverFiltering: true,
                    serverSorting: true
                },
                pageable: {
                    refresh: true
                },
                sortable: {
                    mode: "multiple",
                    allowUnsort: true
                },
                editable: false,
                filterable: false,
                scrollable: false,
                columns: [ {
                    field: "CompanyName",
                    title: "نام شرکت",
                    sortable: true,
                }, {
                    field: "CompanyAbbr",
                    title: "مخفف نام شرکت",
                    sortable: true
                }]
            });
        });
    </script>
}
```

مشکلی که در کد بالا وجود دارد این است که با تغییر نام هر یک از متغییر هایمان ، اطلاعات گرید در ستون مربوطه نمایش داده نمیشود.همچنین عناوین ستونها نیز از DisplayName مدل پیروی نمیکنند.توسط متدهای الحاقی زیر این مشکل برطرف شده است.

```
/// <summary>
    ///
    /// </summary>
    public static class PropertyExtensions
    {
        /// <summary>
        ///
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="expression"></param>
        /// <returns></returns>
        public static MemberInfo GetMember<T>(this Expression<Func<T, object>> expression)
        {
            var mbody = expression.Body as MemberExpression;

            if (mbody != null) return mbody.Member;
            //This will handle Nullable<T> properties.
            var ubody = expression.Body as UnaryExpression;
            if (ubody != null)
            {
                mbody = ubody.Operand as MemberExpression;
            }
            if (mbody == null)
            {
                throw new ArgumentException("Expression is not a MemberExpression", "expression");
            }
            return mbody.Member;
        }

        /// <summary>
        ///
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="expression"></param>
        /// <returns></returns>
        public static string PropertyName<T>(this Expression<Func<T, object>> expression)
        {
            return GetMember(expression).Name;
        }

        /// <summary>
        ///
        /// </summary>
```

```
        /// <typeparam name="T"></typeparam>
        /// <param name="expression"></param>
        /// <returns></returns>
        public static string PropertyDisplay<T>(this Expression<Func<T, object>> expression)
        {
            var propertyMember = GetMember(expression);
            var displayAttributes = propertyMember.GetCustomAttributes(typeof(DisplayNameAttribute),
true);
            return displayAttributes.Length == 1 ?
((DisplayNameAttribute)displayAttributes[0]).DisplayName : propertyMember.Name;
        }
    }
```

```
public static string PropertyName<T>(this Expression<Func<T, object>> expression)
```

جهت بدست آوردن نام متغییر هایمان استفاده مینماییم.

```
public static string PropertyDisplay<T>(this Expression<Func<T, object>> expression)
```

جهت بدست آوردن DisplayNameAttribute استفده میشود. درصورتیکه این DisplayNameAttribute یافت نشود نام متغییر بازگشت داده میشود.

بنابراین View مربوطه را اینگونه بازنویسی میکنیم:

```
@using Models
@{
    const string viewTitle = "شرکت ها";
    ViewBag.Title = viewTitle;
    const string gridName = "companies-grid";
}
<div class="col-md-12">
    <div class="form-panel">
        <header>
            <div class="title">
                <i class="fa fa-book"></i>
                @viewTitle
            </div>
        </header>
        <div class="panel-body">
            <div id="@gridName">
            </div>
        </div>
    </div>
</div>
</div>
@section scripts
{
    <script type="text/javascript">
        $(document).ready(function () {
            $("#@gridName").kendoGrid({
                dataSource: {
                    type: "json",
                    transport: {
                        read: {
                            url: "@Html.Raw(Url.Action(MVC.Company.CompanyList()))",
                            type: "POST",
                            dataType: "json",
                            contentType: "application/json"
                        }
                    },
                    schema: {
                        data: "Data",
                        total: "Total",
                        errors: "Errors"
                        }
```

```
                    },
                    pageSize: 10,
                    serverPaging: true,
                    serverFiltering: true,
                    serverSorting: true
                },
                pageable: {
                    refresh: true
                },
                sortable: {
                    mode: "multiple",
                    allowUnsort: true
                },
                editable: false,
                filterable: false,
                scrollable: false,
                columns: [ {
                    field: "@(PropertyExtensions.PropertyName<CompanyModel>(a => a.CompanyName))",
                    title: "@(PropertyExtensions.PropertyDisplay<CompanyModel>(a => a.CompanyName))",
                    sortable: true,
                }, {
                    field: "@(PropertyExtensions.PropertyName<CompanyModel>(a => a.CompanyAbbr))",
                    title: "@(PropertyExtensions.PropertyDisplay<CompanyModel>(a => a.CompanyAbbr))",
                    sortable: true
                }]
            });
        });
    </script>
}
```

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۱۶ ۱۲:۳۸

با تشکر از شما. حالت پیشرفته‌تر این مساله، کار با مدل‌های تو در تو هست. برای مثال:

```
public class CompanyModel
    {
        public int Id { get; set; }
        public string CompanyName { get; set; }
        public string CompanyAbbr { get; set; }

        public Product Product { set; get; }
    }

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

در اینجا اگر بخواهیم Product.Id را بررسی کنیم:

```
var data = PropertyExtensions.PropertyName<CompanyModel>(x => x.Product.Id);
```

فقط Id آن دریافت می‌شود.
راه حلی که از کدهای EF برای این مساله استخراج شده به صورت زیر است (نمونه‌اش متد Include تو در تو بر روی چند خاصیت):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;

namespace PropertyExtensionsApp
{
    public class PropertyHelper : ExpressionVisitor
    {
        private Stack<string> _stack;
        public string GetNestedPropertyPath(Expression expression)
        {
            _stack = new Stack<string>();
            Visit(expression);
            return _stack.Aggregate((s1, s2) => s1 + "." + s2);
        }

        protected override Expression VisitMember(MemberExpression expression)
        {
            if (_stack != null)
                _stack.Push(expression.Member.Name);
            return base.VisitMember(expression);
        }

        public string GetNestedPropertyName<TEntity>(Expression<Func<TEntity, object>> expression)
        {
            return GetNestedPropertyPath(expression);
        }
    }
}
```

در این حالت خواهیم داشت:

```
var name = new PropertyHelper().GetNestedPropertyName<CompanyModel>(x => x.Product.Id);
```

که خروجی Product.Id را بر می‌گرداند.

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۷/۲۶ ۱۸:۸

در نهایت این متد به این شکل اصلاح شود:

```
/// <summary>
        ///
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="expression"></param>
        /// <returns></returns>
        public static string PropertyName<T>(this Expression<Func<T, object>> expression)
        {
            return new PropertyHelper().GetNestedPropertyName(expression);
        }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۱۰ ۲۳:۵۷

روش دیگری در اینجا: « Strongly-Typed ID References to Razor-Generated Fields »

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۹/۱۱ ۱۴:۱

با تشکر
- نظر نویسنده مقاله تغییر کرده، بدلیل دوباره کاری انجام شده.(توضیحات بیشتر در کامنتهای مقاله ارجاعی)
- روش جاری وابسته به مدل ویو نیست و به همین دلیل محدودیت نداره، بطور مثال یک ویو شامل عملیات‌های اضافه و ویرایش و حذف و گرید لیست آنهاست.

عنوان: **کدام سلسله متدها، متد جاری را فراخوانی کرده‌اند؟**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۷/۰۶ ۹:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Reflection, Profiler, StackTrace

یکی از نیازهای نوشتن یک برنامه‌ی پروفایلر، نمایش اطلاعات متدهایی است که سبب لاگ شدن اطلاعاتی شده‌اند. برای مثال در طراحی interceptorهای EF 6 به یک چنین متدهایی می‌رسیم:

```
public void ScalarExecuted(DbCommand command,
                                    DbCommandInterceptionContext<object> interceptionContext)
        {
        }
```

سؤال: در زمان اجرای ScalarExecuted دقیقا در کجا قرار داریم؟ چه متدی در برنامه، در کدام کلاس، سبب رسیدن به این نقطه شده‌است؟
تمام این اطلاعات را در زمان اجرا توسط کلاس StackTrace می‌توان بدست آورد:

```
public static string GetCallingMethodInfo()
        {
            var stackTrace = new StackTrace(true);
            var frameCount = stackTrace.FrameCount;

            var info = new StringBuilder();
            var prefix = "-- ";
            for (var i = frameCount - 1; i >= 0; i--)
            {
                var frame = stackTrace.GetFrame(i);
                var methodInfo = getStackFrameInfo(frame);
                if (string.IsNullOrWhiteSpace(methodInfo))
                    continue;

                info.AppendLine(prefix + methodInfo);
                prefix = "-" + prefix;
            }

            return info.ToString();
        }
```

ایجاد یک نمونه جدید از کلاس StackTrace با پارامتر true به این معنا است که می‌خواهیم اطلاعات فایل‌های متناظر را نیز در صورت وجود دریافت کنیم.
خاصیت stackTrace.FrameCount مشخص می‌کند که در زمان فراخوانی متد GetCallingMethodInfo که اکنون برای مثال درون متد ScalarExecuted قرار گرفته‌است، از چند سطح بالاتر این فراخوانی صورت گرفته‌است. سپس با استفاده از متد stackTrace.GetFrame می‌توان به اطلاعات هر سطح دسترسی یافت.
در هر StackFrame دریافتی، با فراخوانی stackFrame.GetMethod می‌توان نام متد فراخوان را بدست آورد. متد stackFrame.GetFileLineNumber دقیقا شماره سطری را که فراخوانی از آن صورت گرفته، بازگشت می‌دهد و stackFrame.GetFileName نیز نام فایل مرتبط را مشخص می‌کند.

یک نکته:
شرط عمل کردن متدهای stackFrame.GetFileName و stackFrame.GetFileLineNumber در زمان اجرا، وجود فایل PDB اسمبلی در حال بررسی است. بدون آن اطلاعات محل قرارگیری فایل سورس مرتبط و شماره سطر فراخوان، قابل دریافت نخواهند بود.

اکنون بر اساس این اطلاعات، متد getStackFrameInfo چنین پیاده سازی را خواهد داشت:

```
private static string getStackFrameInfo(StackFrame stackFrame)
        {
            if (stackFrame == null)
                return string.Empty;

            var method = stackFrame.GetMethod();
```

```
            if (method == null)
                return string.Empty;

            if (isFromCurrentAsm(method) || isMicrosoftType(method))
            {
                return string.Empty;
            }

            var methodSignature = method.ToString();
            var lineNumber = stackFrame.GetFileLineNumber();
            var filePath = stackFrame.GetFileName();

            var fileLine = string.Empty;
            if (!string.IsNullOrEmpty(filePath))
            {
                var fileName = Path.GetFileName(filePath);
                fileLine = string.Format("[File={0}, Line={1}]", fileName, lineNumber);
            }

            var methodSignatureFull = string.Format("{0} {1}", methodSignature, fileLine);
            return methodSignatureFull;
        }
```

و خروجی آن برای مثال چنین شکلی را خواهد داشت:

```
Void Main(System.String[]) [File=Program.cs, Line=28]
```

که وجود file و line آن تنها به دلیل وجود فایل PDB اسمبلی مورد بررسی است.

در اینجا خروجی نهایی متد GetCallingMethodInfo به شکل زیر است که در آن چند سطح فراخوانی را می‌توان مشاهده کرد:

```
-- Void Main(System.String[]) [File=Program.cs, Line=28]
--- Void disposedContext() [File=Program.cs, Line=76]
---- Void Opened(System.Data.Common.DbConnection,
System.Data.Entity.Infrastructure.Interception.DbConnectionInterceptionContext)
[File=DatabaseInterceptor.cs,Line=157]
```

جهت تعدیل خروجی متد GetCallingMethodInfo، عموما نیاز است مثلا از کلاس یا اسمبلی جاری صرفنظر کرد یا اسمبلی‌های مایکروسافت نیز در این بین شاید اهمیتی نداشته باشند و بیشتر هدف بررسی سورس‌های موجود است تا فراخوانی‌های داخلی یک اسمبلی ثالث:

```
private static bool isFromCurrentAsm(MethodBase method)
        {
            return method.ReflectedType == typeof(CallingMethod);
        }

        private static bool isMicrosoftType(MethodBase method)
        {
            if (method.ReflectedType == null)
                return false;

            return method.ReflectedType.FullName.StartsWith("System.") ||
                   method.ReflectedType.FullName.StartsWith("Microsoft.");
        }
```

کد کامل CallingMethod.cs را از اینجا می‌توانید دریافت کنید:

CallingMethod.cs

نظرات خوانندگان

نویسنده: علیرضا
تاریخ: ۱۳۹۳/۰۷/۱۰ ۲۳:۳۸

چه موقعی GetMethod میتواند Null برگرداند؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۷/۱۱ ۰:۳۷

زمانیکه کامپایلر مباحث inlining متدها را جهت بهینه سازی اعمال کند.

عنوان: **ساخت Attribute های دلخواه یا خصوصی سازی شده**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۰۲ ۲۲:۴۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Validation, Reflection

در قسمت‌های مختلفی از منابع آموزشی این سایت از متادیتاها attributes استفاده شده و در برخی آموزش هایی چون EF و MVC حداقل یک قسمت کامل را به خود اختصاص داده‌اند. متادیتاها کلاس‌هایی هستند که به روشی سریع و کوتاه در بالای یک Type معرفی شده و ویژگی‌هایی را به آن اضافه می‌کنند. به عنوان مثال متادیتای زیر را ببینید. این متادیتا در بالای یک متد در یک کلاس تعریف شده است و این متد را منسوخ شده اعلام می‌کند و به برنامه نویس می‌گوید که در نسخه‌ی جاری کتابخانه، این متد که احتمال میرود در نسخه‌های پیشین کاربرد داشته است، الان کارآیی خوبی برای استفاده نداشته و بهتر است طبق مستندات آن کلاس، از یک متد جایگزین که برای آن فراهم شده است استفاده کند.

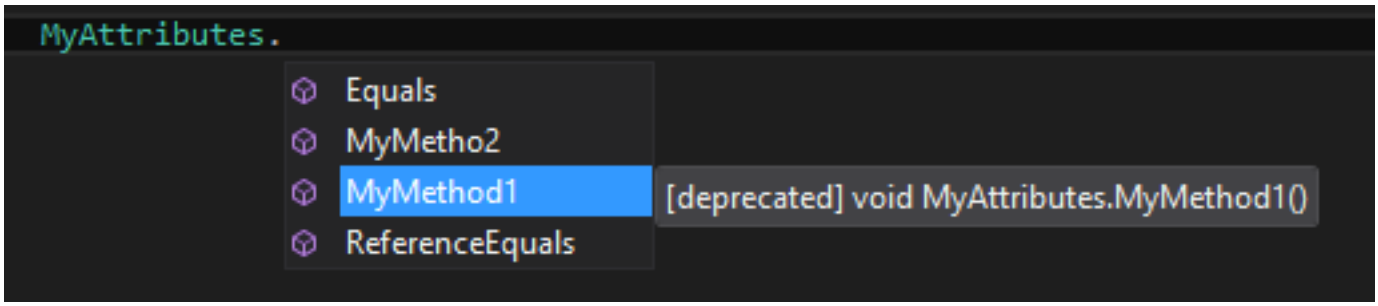
```
public static class  MyAttributes
    {
        [Obsolete]
        public static void MyMethod1()
        {

        }

        public static void MyMetho2()
        {

        }
    }
```

همانطور که ملاحظه می‌کنید می‌توانید اخطار آن را مشاهده کنید:



البته توصیه می‌کنم از ابزارهایی چون Resharper در کارهایتان استفاده کنید، تا طعم کدنویسی را بهتر بچشید. نحوه‌ی نمایش آن در Resharper به مراتب واضح‌تر و گویاتر است:

حال در این بین این سؤال پیش می‌آید که چگونه ما هم می‌توانیم متادیتاهایی را با سلیقه‌ی خود ایجاد کنیم.
برای تهیه‌ی یک متادیتا از کلاس system.attribute استفاده می‌کنیم:

```
public  class  MyMaxLength:Attribute
    {

    }
```

در چنین حالتی شما یک متادیتا ساخته‌اید که می‌توان از آن به شکل زیر استفاده کرد:

```
[MyMaxLength]
    public class GetCustomProperties
    {
//...
     }
```

ولی اگر بخواهید توسط این متادیتا اطلاعاتی را دریافت کنید، می‌توانید به روش زیر عمل کنید. در اینجا من دوست دارم یک متادیتا به اسم MyMaxLength را ایجاد کرده تا جایگزین MaxLength دات نت کنم، تا طبق میل من رفتار کند.

```
public  class  MyMaxLength:Attribute
    {
        private int max;
        public string ErrorText = "";

        public MyMaxLength(int max)
        {
            this.max = max;
            ErrorText = string.Format("max Length is {0} chars", max);
        }
    }
```

در کد بالا، یک متادیتا با یک پارامتر اجباری در سازنده تعریف شده است. این کلاس هم می‌تواند مثل سایر کلاس‌ها سازنده‌های مختلفی داشته باشد تا چندین شکل تعریف متادیتا داشته باشیم. متغیر ErrorText به عنوان یک پارامتر معرفی نشده، ولی از آن جا که public تعریف شده است می‌تواند مورد استفاده‌ی مستقیم قرار بگیرد و استفاده‌ی از آن نیز اختیاری است. نحوه‌ی معرفی این متادیتا نیز به صورت زیر است:

```
[MyMaxLength(30)]
    public class GetCustomProperties
    {
//...
     }

//or
 [MyMaxLength(30,ErrorText = "شما اجازه ندارید بیش از 30 کاراکتر وارد نمایید")]
    public class GetCustomProperties
    {
//...
     }
```

در حالت اول از آنجا که متغیر ErrorText اختیاری است، تعریف نشده‌است. پس در نتیجه با مقدار (x=max) Max length is chars پر خواهد شد ولی در حالت دوم برنامه نویس متن خطا را به خود کلاس واگذار نکرده است و آن را طبق میل خود تغییر داده است.

اجباری کردن Type

هر متادیتا می‌تواند مختص یک نوع Type باشد که این نوع می‌تواند یک کلاس، متد، پراپرتی یا ساختار و ... باشد. نحوه‌ی محدود سازی آن توسط یک متادیتا مشخص می‌شود:

```
[System.AttributeUsage(System.AttributeTargets.Class | System.AttributeTargets.Struct)]
    public  class  MyMaxLength:Attribute
    {
        private int max;
        public string ErrorText = "";

        public MyMaxLength(int max)
        {
            this.max = max;
            ErrorText = string.Format("max Length is {0} chars", max);
        }
    }
```

الان این کلاس توسط متادیتای AttributeUsage که پارامتر ورودی آن Enum است محدود به دو ساختار کلاس و Struct شده است. البته در ویژوال بیسیک با نام Structure معرفی شده است. اگر ساختار شمارشی AttributeTarget را مشاهده کنید، لیستی از نوع‌ها را چون All (همه موارد) ، دلیگیت، سازنده، متد و ... را مشاهده خواهید کرد و از آن جا که این متادیتای ما کاربردش در پراپرتی‌ها خلاصه می‌شود، از متادیتای زیر بر روی آن استفاده می‌کنیم:

```
[AttributeUsage(AttributeTargets.Property)]
```

```
public class User
    {
        [MyMaxLength(30, ErrorText = "شما اجازه ندارید بیش از 30 کاراکتر وارد نمایید")]
        public string Name { get; set; }
}
```

یکی دیگر از ویژگی‌های AttributeUsage خصوصیتی به اسم AllowMultiple است که اجازه می‌دهد بیش از یک بار این متادیتا، بر روی یک نوع استفاده شود:

```
[AttributeUsage(AttributeTargets.Property,AllowMultiple = true)]
    public  class  MyMaxLength:Attribute
    {
        //....
     }
```

که تعریف چندگانه آن به شکل زیر می‌شود:

```
[MyMaxLength(40, ErrorText = "شما اجازه ندارید بیش از 40 کاراکتر وارد نمایید")]
        [MyMaxLength(50, ErrorText = "شما اجازه ندارید بیش از 50 کاراکتر وارد نمایید")]
        [MyMaxLength(30, ErrorText = "شما اجازه ندارید بیش از 30 کاراکتر وارد نمایید")]
        public string Name { get; set; }
```

در این مثال ما فقط اجازه‌ی یکبار استفاده را خواهیم داد؛ پس مقدار این ویژگی را false قرار می‌دهم.

آخرین ویژگی که این متادیتا در دسترس ما قرار میدهد، استفاده از خصوصیت ارث بری است که به طور پیش فرض با True مقداردهی شده است. موقعی که شما یک متادیتا را به ویژگی ارث بری مزین کنید، در صورتی که آن کلاس که برایش متادیتا تعریف می‌کنید به عنوان والد مورد استفاده قرار بگیرد، فرزند آن هم به طور خودکار این متادیتا برایش منظور می‌گردد. به مثال‌های زیر دقت کنید:
دو عدد متادیتا تعریف شده که یکی از آن‌ها ارث بری در آن فعال شده و دیگری خیر.

```
public class MyAttribute : Attribute
{
    //...
}
```

```
[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
    //...
}
```

هر دو متادیتا بر سر یک متد در یک کلاسی که بعدا از آن ارث بری می‌شود تعریف شده اند.

```
public class MyClass
{
    [MyAttribute]
    [YourAttribute]
    public virtual void MyMethod()
    {
        //...
    }
}
```

در کد زیر کلاس بالا به عنوان والد معرفی شده و متد کلاس فرزند الان شامل متادیتایی به اسم MyAttribute است، ولی متادیتای YourAttribute بر روی آن تعریف نشده است.

```
public class YourClass : MyClass
{
    public override void MyMethod()
    {
        //...
    }

}
```

الان که با نحوه‌ی تعریف یکی از متادیتاها آشنا شدیم، این بحث پیش می‌آید که چگونه Type مورد نظر را تحت تاثیر این متادیتا قرار دهیم. الان چگونه میتوانم حداکثر متنی که یک پراپرتی می‌گیرد را کنترل کنم. در اینجا ما از مفهومی به نام Reflection استفاده می‌کنیم. با استفاده از این مفهوم ما میتوانیم به تمامی قسمت‌های یک Type دسترسی داشته باشیم. متاسفانه دسترسی مستقیمی از داخل کلاس متادیتا به نوع مورد نظر نداریم. کد زیر تمامی پراپرتی‌های یک کلاس را چک میکند و سپس ویژگی‌های هر پراپرتی را دنبال کرده و در صورتیکه متادیتای مورد نظر به آن پراپرتی ضمیمه شده باشد، حالا می‌توانید عملیات را انجام دهید. کد زیر میتواند در هر جایی نوشته شود. داخل کلاسی که به آن متادیتا ضمیمه می‌کنید یا داخل تابع Main در اپلیکشین‌ها و هر جای دیگر. مقدار True که به متد GetCustomAttributes پاس می‌شود باعث می‌شود تا متادیتاهای ارث بری شده هم لحاظ گردند.

```
Type type = typeof (User);

                foreach (PropertyInfo property in type.GetProperties())
                {
                    foreach (Attribute attribute in property.GetCustomAttributes(true))
                    {
                        MyMaxLength max = attribute as MyMaxLength;
                        if (max != null)
                        {
                            string Max = max.ErrorText;
                            //انجام عملیات
                        }
                    }
                }
```

البته یک ترفند جهت دسترسی به کلاس‌ها از داخل کلاس متادیتا وجود دارد و آن هم این هست که نوع را از طریق پارامتر به سمت متادیتا ارسال کنید. هر چند این کار زیبایی ندارد ولی به هر حال روش خوبی برای کنترل از داخل کلاس متادیتا و هچنین منظم سازی و دسته بندی و کم کردن کد دارد.

```
[MyMaxLength(30, typeof(User))]
```

عنوان: **عمومی سازی الگوریتم‌ها با استفاده از Reflection**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۰۳ ۱۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: MVC, Reflection

در این مقاله قصد داریم عملیات Reflection را بیشتر در انجام ساده‌تر عملیات ببینیم. عملیاتی که به همراه کار اضافه، تکراری و خسته کننده است و با استفاده از Reflection این کارها حذف شده و تعداد خطوط هم پایین می‌آید. حتی گاها ممکن است موجب استفاده‌ی مجدد از کدها شود که همگی این عوامل موجب بالا رفتن امتیاز Refactoring می‌شوند.
در مثال‌های زیر مجموعه‌ای از Reflectionهای ساده و کاملا کاربردی است که من با آن‌ها روبرو شده ام.

کوتاه سازی کدهای نمایش یک View در ASP.NET MVC با Reflection

یکی از قسمت‌هایی که مرتبا با آن سر و کار دارید، نمایش اطلاعات است. حتی یک جدول را هم که می‌خواهید بسازید، باید ستون‌های آن جدول را یک به یک معرفی کنید. ولی در عمل، یک Reflection ساده این کار به یک تابع چند خطی و سپس برای ترسیم هر ستون جدول از دو خط استفاده خواهید کرد ولی مزیتی که دارد این است که این تابع برای تمامی جدول‌ها کاربردی عمومی پیدا می‌کند. برای نمونه دوست داشتم برای بخش مدیر، قسمت پروفایلی را ایجاد کنم و در آن اطلاعاتی چون نام، نام خانوادگی، تاریخ تولد، تاریخ ایجاد و خیلی از اطلاعات دیگر را نمایش دهم. به جای اینکه بیایم برای هر قسمت یک خط partial ایجاد کنم، با استفاده از reflection و یک حلقه، تمامی اطلاعات را به آن پارشال پاس می‌کنم. مزیت این روش این است که اگر بخواهم در یک جای دیگر، اطلاعات یک محصول یا یک فاکتور را هم نمایش دهم، باز هم همین تابع برایم کاربرد خواهد داشت:

تصویر زیر را که برگرفته از یک قالب Bootstrap است، ملاحظه کنید. اصلا علاقه ندارم که برای یک به یک آن‌ها، یک سطر جدید را تعریف کنم و به View بگویم این پراپرتی را نشان بده؛ دوباره مورد بعدی هم به همین صورت و دوباره و دوباره و ... . دوست دارم یک تابع عمومی، همه‌ی این کارها را خودکار انجام دهد.

First Name	: Jonathan	Last Name	: Smith
Country	: Australia	Birthday	: 13 July 1983
Occupation	: UI Designer	Email	: jsmith@flatlab.com
Mobile	: (12) 03 4567890	Phone	: 88 (02) 123456

ساختار اطلاعاتی تصویر فوق به شرح زیر است:

```
<div>
                            <div>
                                <div>
                                    <p><span>First Name </span>: Jonathan</p>
                                </div>
   </div>
                        </div>
```

که به دو فایل پارشال تقسیم شده است _Bio و _BioRow که محتویات هر پارشال هم به شرح زیر است:

_BioRow

```
@model System.Web.UI.WebControls.ListItem
```

```
<div>
    <p><span>@Model.Text </span>: @Model.Value</p>
</div>
```

در پارشال بالا ورودی از نوع listItem است که یک متن دارد و یک مقدار.(شاید به نظر شبیه حالت جفت کلید و مقدار باشد ولی در این کلاس خبری از کلید نیست).

پارشال پایینی هم دربرگیرنده‌ی پارشال بالاست که قرار است چندین و چند بار پارشال بالا در خودش نمایش دهد.
_Bio

```
@using System.Web.UI.WebControls
@using ZekrWepApp.Filters
@model ZekrModel.Admin

    <div>
        <h1>Bio Graph</h1>
        <div>

            @{
                ListItemCollection collection = GetCustomProperties.Get(Model,exclude:new
string[]{"Poems","Id"});
                foreach (var item in collection)
                {
                    Html.RenderPartial(MVC.Shared.Views._BioRow, item);
                }
            }

                    </div>
    </div>
```

پارشال بالا یک مدل از کلاس Admin را می‌پذیرد که قرار است اطلاعات شخصی مدیر را نمایش دهد. در ابتدا متدی از یک کلاس ایستا وجود دارد که کدهای Reflection درون آن قرار دارند که یک مجموعه از ListItemها را بر می‌گرداند و سپس با یک حلقه، پارشال _BioRow را صدا می‌زند.

کد درون این کلاس ایستا را بررسی می‌کنیم؛ این کلاس دو متد دارد یکی عمومی و دیگری خصوصی است:

```
public class GetCustomProperties
    {
        private static PropertyInfo[] getObjectsInfos(object obj,string[] inclue,string[] exclude )
        {
            var list = obj.GetType().GetProperties();

            PropertyInfo[] outputPropertyInfos = null;

            if (inclue != null)
            {
                return list.Where(propertyInfo => inclue.Contains(propertyInfo.Name)).ToArray();
            }
            if (exclude != null)
            {
                return list.Where(propertyInfo => !exclude.Contains(propertyInfo.Name)).ToArray();
            }
            return list;
        }
    }
```

کد بالا که یک کد خصوصی است، سه پارامتر را می‌پذیرد. اولی مدل یا کلاسی است که به آن پاس کرده‌ایم. دو پارامتر بعدی اختیاری است و در کد پارشال بالا Exclude را تعریف کرده ایم و تنهای یکی از دو پارامتر بالا هم باید مورد استفاده قرار بگیرند و Include ارجحیت دارد. وظیفه‌ی این دو پارمتر این است که آرایه ای از رشته‌ها را دریافت می‌کنند که نام پراپرتی‌ها در آن‌ها ذکر شده است. آرایه Include می‌گوید که فقط این پراپرتی‌ها را برگردان ولی اگر دوست دارید همه‌ی پارامترها را نمایش دهید و تنها یکی یا چندتا از آن‌ها را حذف کنید، از آرایه Exclude استفاده کنید. در صورتی که این دو آرایه خالی باشند، همه‌ی پراپرتی‌ها

بازگشت داده می‌شوند و در صورتی که یکی از آن‌ها وارد شده باشد، طبق دستورات Linq بالا بررسی می‌کند که (Include) آیا اسامی مشترکی بین آن‌ها وجود دارد یا خیر؟ اگر وجود دارد آن را در لیست قرار داده و بر می‌گرداند و در حالت Exclude این مقایسه به صورت برعکس انجام می‌گیرد و باید لیستی برگردانده شود که اسامی، نکته مشترکی نداشته باشند.

متد عمومی که در این کلاس قرار دارد به شرح زیر است:

```
public static ListItemCollection Get(object obj,string[] inclue=null,string[] exclude=null)
        {
            var propertyInfos = getObjectsInfos(obj, inclue, exclude);
            if (propertyInfos == null) throw new ArgumentNullException("propertyInfos is null");

            var collection = new ListItemCollection();

            foreach (PropertyInfo propertyInfo in propertyInfos)
            {


                string name = propertyInfo.Name;

                foreach (Attribute attribute in propertyInfo.GetCustomAttributes(true))
                {
                    DisplayAttribute displayAttribute = attribute as DisplayAttribute;

                    if (displayAttribute != null)
                    {
                        name = displayAttribute.Name;
                        break;
                    }
                }

                string value = "";
                object objvalue = propertyInfo.GetValue(obj);
                if (objvalue != null) value = objvalue.ToString();

                collection.Add(new ListItem(name,value));
            }
            return collection;
        }
```

این متد سه پارامتر را از کاربر دریافت و به سمت متد خصوصی ارسال می‌کند. موقعی‌که پراپرتی‌ها بازگشت داده می‌شوند، دو قسمت آن مهم است؛ یکی عنوان پراپرتی و دیگری مقدار پراپرتی. از آن جا که نام پراپرتی‌ها طبق سلیقه‌ی برنامه نویس و با حروف انگلیسی نوشته می‌شوند، در صورتی که برنامه نویس از متادیتای Display در مدل بهره برده باشد، به جای نام پراپرتی مقداری را که به متادیتای Display داده‌ایم، بر می‌گردانیم.

کد بالا پراپرتی‌ها را دریافت و یک به یک متادیتاهای آن را بررسی کرده و در صورتی که از متادیتای Display استفاده کرده باشند، مقدار آن را جایگزین نام پراپرتی خواهد کرد. در مورد مقدار هم از آنجا که اگر پراپرتی با Null پر شده باشد، تبدیل به رشته‌ای با پیام خطای روبرو خواهد شد. در نتیجه بهتر است یک شرط احتیاط هم روی آن پیاده شود. در آخر هم از متن و مقدار، یک آیتم ساخته و درون Collection اضافه می‌کنیم و بعد از اینکه همه پراپرتی‌ها بررسی شدند، Collection را بر می‌گردانیم.

```
[Display(Name = "نام کاربری")]
public string UserName { get; set; }
```

کد کامل کلاس:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
using System.Web;
using System.Web.Mvc.Html;
using System.Web.UI.WebControls;
using Links;

namespace ZekrWepApp.Filters
```

```
{

    public class GetCustomProperties
    {
        public static ListItemCollection Get(object obj,string[] inclue=null,string[] exclude=null)
        {
            var propertyInfos = getObjectsInfos(obj, inclue, exclude);
            if (propertyInfos == null) throw new ArgumentNullException("propertyInfos is null");

            var collection = new ListItemCollection();

            foreach (PropertyInfo propertyInfo in propertyInfos)
            {
                string name = propertyInfo.Name;
                foreach (Attribute attribute in propertyInfo.GetCustomAttributes(true))
                {
                    DisplayAttribute displayAttribute = attribute as DisplayAttribute;

                    if (displayAttribute != null)
                    {
                        name = displayAttribute.Name;
                        break;
                    }
                }

                string value = "";
                object objvalue = propertyInfo.GetValue(obj);
                if (objvalue != null) value = objvalue.ToString();

                collection.Add(new ListItem(name,value));
            }
            return collection;
        }
        private static PropertyInfo[] getObjectsInfos(object obj,string[] include,string[] exclude )
        {
            var list = obj.GetType().GetProperties();

            PropertyInfo[] outputPropertyInfos = null;

            if (include != null)
            {
                return list.Where(propertyInfo => include.Contains(propertyInfo.Name)).ToArray();
            }
            if (exclude != null)
            {
                return list.Where(propertyInfo => !exclude.Contains(propertyInfo.Name)).ToArray();
            }
            return list;
        }
    }
}
```

لیستی از پارامترها با Reflection

مورد بعدی که ساده‌تر بوده و از کد بالا مختصرتر هم هست، این است که قرار بود برای یک درگاه، یک سری اطلاعات را با متد Post ارسال کنم که نحوه‌ی ارسال اطلاعات به شکل زیر بود:

```
amount=1000&orderId=452&Pid=xxx&....
```

کد زیر را من جهت ساخت قالب‌های این چنینی استفاده می‌کنم:

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

namespace Utils
{
    public class QueryStringParametersList
    {
        private string Symbol = "&";
        private List<KeyValuePair<string, string>> list { get; set; }
```

```
        public QueryStringParametersList()
        {
            list = new List<KeyValuePair<string, string>>();
        }
        public QueryStringParametersList(string symbol)
        {
            Symbol = symbol;
           list = new List<KeyValuePair<string, string>>();
        }

        public int Size
        {
            get { return list.Count; }
        }
        public void Add(string key, string value)
        {
            list.Add(new KeyValuePair<string, string>(key, value));
        }

        public string GetQueryStringPostfix()
        {
            return string.Join(Symbol, list.Select(p => Uri.EscapeDataString(p.Key) + "=" +
Uri.EscapeDataString(p.Value)));
        }
    }
}
```

یک متغیر به نام symbol دارد و در صورتی در شرایط متفاوت، قصد چسپاندن چیزی را به یکدیگر با علامتی خاص داشته باشید، این تابع می‌تواند کاربرد داشته باشد. این متد از یک لیست کلید و مقدار استفاده کرده و پارامترهایی را که به آن پاس می‌شود، نگهداری و سپس توسط متد GetQueryStringPostfix آن‌ها را با یکدیگر الحاق کرده و در قالب یک رشته بر می‌گرداند.
کاربرد Reflection در اینجا این است که من باید دوبار به شکل زیر، دو نوع اطلاعات متفاوت را پست کنم. یکی موقع ارسال به درگاه و دیگری موقع بازگشت از درگاه.

```
QueryStringParametersList queryparamsList = new QueryStringParametersList();

            ueryparamsList.Add("consumer_key", requestPayment.Consumer_Key);
            queryparamsList.Add("amount", requestPayment.Amount.ToString());
            queryparamsList.Add("callback", requestPayment.Callback);
            queryparamsList.Add("description", requestPayment.Description);
            queryparamsList.Add("email", requestPayment.Email);
            queryparamsList.Add("mobile", requestPayment.Mobile);
            queryparamsList.Add("name", requestPayment.Name);
            queryparamsList.Add("irderid", requestPayment.OrderId.ToString());
```

ولی با استفاده از کد Reflection که در بالاتر عنوان شد، باید نام و مقدار پراپرتی را گرفته و در یک حلقه آن‌ها را اضافه کنیم، بدین شکل:

```
private QueryStringParametersList ReadParams(object obj)
        {
            PropertyInfo[] propertyInfos = obj.GetType().GetProperties();

            QueryStringParametersList queryparamsList = new QueryStringParametersList();
            for (int i = 0; i < propertyInfos.Count(); i++)
            {

queryparamsList.Add(propertyInfos[i].Name.ToLower(),propertyInfos[i].GetValue(obj).ToString() );
            }
            return queryparamsList;
        }
```

در کد بالا هر بار پراپرتی‌های کلاس را خوانده و نام و مقدار آن‌ها را گرفته و به کلاس QueryString اضافه می‌کنیم. پارامتر ورودی این متد به این خاطر object در نظر گرفته شده است که تا هر کلاسی را بتوانیم به آن پاس کنیم که خودم در همین کلاس درگاه، دو کلاس را به آن پاس کردم.

عنوان: **ساده سازی و بالا بردن سرعت عملیات Reflection با استفاده از Dynamic Proxy**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۰۹ ۱۲:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Reflection, Performance, AOP

فرض کنید یک چنین کلاسی طراحی شده‌است:

```
public class NestedClass
{
    private int _field2;
    public NestedClass()
    {
        _field2 = 12;
    }
}

public class MyClass
{
    private int _field1;
    private NestedClass _nestedClass;

    public MyClass()
    {
        _field1 = 1;
        _nestedClass = new NestedClass();
    }

    private string GetData()
    {
        return "Test";
    }
}
```

می‌خواهیم از طریق Reflection مقادیر فیلدها و متدهای مخفی آن‌را بخوانیم.
حالت متداول دسترسی به فیلد خصوصی آن از طریق Reflection، یک چنین شکلی را دارد:

```
var myClass = new MyClass();

var field1Obj = myClass.GetType().GetField("_field1", BindingFlags.NonPublic | BindingFlags.Instance);
if (field1Obj != null)
{
    Console.WriteLine(Convert.ToInt32(field1Obj.GetValue(myClass)));
}
```

و یا دسترسی به مقدار خروجی متد خصوصی آن، به نحو زیر است:

```
var getDataMethod = myClass.GetType().GetMethod("GetData", BindingFlags.NonPublic |
BindingFlags.Instance);
if (getDataMethod != null)
{
    Console.WriteLine(getDataMethod.Invoke(myClass, null));
}
```

در اینجا دسترسی به مقدار فیلد مخفی NestedClass، شامل مراحل زیر است:

```
var nestedClassObj = myClass.GetType().GetField("_nestedClass", BindingFlags.NonPublic |
BindingFlags.Instance);
if (nestedClassObj != null)
{
    var nestedClassFieldValue = nestedClassObj.GetValue(myClass);
    var field2Obj = nestedClassFieldValue.GetType()
        .GetField("_field2", BindingFlags.NonPublic | BindingFlags.Instance);
    if (field2Obj != null)
    {
        Console.WriteLine(Convert.ToInt32(field2Obj.GetValue(nestedClassFieldValue)));
    }
}
```

البته این مقدار کد فقط برای دسترسی به دو سطح تو در تو بود.

چقدر خوب بود اگر می‌شد بجای این همه کد، نوشت:

```
myClass._field1
myClass._nestedClass._field2
myClass.GetData()
```

نه؟!
برای این مشکل راه حلی معرفی شده‌است به نام Dynamic Proxy که در ادامه به معرفی آن خواهیم پرداخت.

معرفی Dynamic Proxy

Dynamic Proxy یکی از مفاهیم AOP است. به این معنا که توسط آن یک محصور کننده‌ی نامرئی، اطراف یک شیء تشکیل خواهد شد. از این غشای نامرئی عموما جهت مباحث ردیابی اطلاعات، مانند پروکسی‌های Entity framework، همانجایی که تشخیص می‌دهد کدام خاصیت به روز شده‌است یا خیر، استفاده می‌شود و یا این غشای نامرئی کمک می‌کند که در حین دسترسی به خاصیت یا متدی، بتوان منطق خاصی را در این بین تزریق کرد. برای مثال فرآیند تکراری logging سیستم را به این غشای نامرئی منتقل کرد و به این ترتیب می‌توان به کدهای تمیزتری رسید.
یکی دیگر از کاربردهای این محصور کننده یا غشای نامرئی، ساده سازی مباحث Reflection است که نمونه‌ای از آن در پروژه‌ی EntityFramework.Extended بکار رفته‌است.
در اینجا، کار با محصور سازی نمونه‌ای از کلاس مورد نظر با Dynamic Proxy شروع می‌شود. سپس کل عملیات Reflection فوق در همین چند سطر ذیل به نحوی کاملا عادی و طبیعی قابل انجام است:

```
// Accessing a private field
dynamic myClassProxy = new DynamicProxy(myClass);
dynamic field1 = myClassProxy._field1;
Console.WriteLine((int)field1);

// Accessing a nested private field
dynamic field2 = myClassProxy._nestedClass._field2;
Console.WriteLine((int)field2);

// Accessing a private method
dynamic data = myClassProxy.GetData();
Console.WriteLine((string)data);
```

خروجی Dynamic Proxy از نوع dynamic دات نت 4 است. پس از آن می‌توان در اینجا هر نوع خاصیت یا متد دلخواهی را به شکل dynamic تعریف کرد و سپس به مقادیر آن‌ها دسترسی داشت.

بنابراین با استفاده از Dynamic Proxy فوق می‌توان به دو مهم دست یافت:
1) ساده سازی و زیبا سازی کدهای کار با Reflection
2) استفاده‌ی ضمنی از مباحث Fast Reflection . در کتابخانه‌ی Dynamic Proxy معرفی شده، دسترسی به خواص و متدها، توسط کدهای IL بهینه سازی شده‌است و در دفعات آتی کار با آن‌ها، دیگر شاهد سربار بالای Reflection نخواهیم بود.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:
DynamicProxyTests.zip