عنوان: **بررسی مقدماتی مراحل کامپایل یک قطعه کد سی‌شارپ و آشنایی با OpCodes**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۳ ۱۳:۸
آدرس: www.dotnettips.info
گروه‌ها: C#, CIL, CLR, IL, MSIL, Reflection

کامپایلر سی‌شارپ چگونه عمل می‌کند؟

کار یک کامپایلر ترجمه قطعه‌ای از اطلاعات به چیز دیگری است. کامپایلر سی‌شارپ، machine code معادل دستورات دات نتی را تهیه نمی‌کند. Machine code، کدی است که مستقیما بر روی CPU قابل اجرا است. در دات نت این مرحله به CLR یا Common language runtime واگذار شده است تا کار اجرای نهایی کدهای تهیه شده توسط کامپایلر سی‌شارپ را انجام دهد.
بنابراین زمانیکه در VS.NET سعی در اجرای یک قطعه کد می‌نمائیم، مراحل ذیل رخ می‌دهند:
الف) فایل‌های سی‌شارپ پروژه، توسط کامپایلر بارگذاری می‌شوند.
ب) کامپایلر کدهای این فایل‌ها را پردازش می‌کند.
ج) سپس چیزی را به نام MSIL تولید می‌کند.
د) در ادامه فایل خروجی نهایی، با افزودن PE Headers تولید می‌شود. توسط PE headers مشخص می‌شود که فایل تولیدی نهایی آیا اجرایی است، یا یک DLL می‌باشد و امثال آن.
ه) و در آخر، فایل اجرایی تولیدی توسط CLR بارگذاری و اجرا می‌شود.

MSIL چیست؟

MSIL مخفف Microsoft intermediate language است. به آن CIL یا Common intermediate language هم گفته می‌شود و این دقیقا همان کدی است که توسط CLR خوانده و اجرا می‌شود. MSIL یک زبان طراحی شده مبتنی بر پشته‌ها است و بسیار شبیه به سایر زبان‌های اسمبلی موجود می‌باشد.

یک سؤال: آیا قطعه کدهای ذیل، کدهای IL یکسانی را تولید می‌کنند؟

```
namespace FastReflectionTests
{
    public class Test
    {
        public void Method1()
        {
            var x = 10;
            var y = 20;
            if (x == 10)
            {
                if (y == 20)
                {

                }
            }
        }

        public void Method2()
        {
            var x = 10;
            var y = 20;
            if (x == 10 && y == 20)
            {

            }
        }
    }
}
```

برای یافتن کدهای MSIL یا IL یک برنامه کامپایل شده می‌توان از ابزارهایی مانند Reflector یا ILSpy استفاده کرد. برای نمونه اگر از برنامه ILSpy استفاده کنیم چنین خروجی IL معادلی را می‌توان مشاهده کرد:

```
.class public auto ansi beforefieldinit FastReflectionTests.Test
extends [mscorlib]System.Object
{
// Methods
.method public hidebysig
instance void Method1 () cil managed
{
// Method begins at RVA 0x3bd0
// Code size 17 (0x11)
.maxstack 2
.locals init (
[0] int32 x,
[1] int32 y
)

IL_0000: ldc.i4.s 10
IL_0002: stloc.0
IL_0003: ldc.i4.s 20
IL_0005: stloc.1
IL_0006: ldloc.0
IL_0007: ldc.i4.s 10
IL_0009: bne.un.s IL_0010

IL_000b: ldloc.1
IL_000c: ldc.i4.s 20
IL_000e: pop
IL_000f: pop

IL_0010: ret
} // end of method Test::Method1

.method public hidebysig
instance void Method2 () cil managed
{
// Method begins at RVA 0x3bf0
// Code size 17 (0x11)
.maxstack 2
.locals init (
[0] int32 x,
[1] int32 y
)

IL_0000: ldc.i4.s 10
IL_0002: stloc.0
IL_0003: ldc.i4.s 20
IL_0005: stloc.1
IL_0006: ldloc.0
IL_0007: ldc.i4.s 10
IL_0009: bne.un.s IL_0010

IL_000b: ldloc.1
IL_000c: ldc.i4.s 20
IL_000e: pop
IL_000f: pop

IL_0010: ret
} // end of method Test::Method2

.method public hidebysig specialname rtspecialname
instance void .ctor () cil managed
{
// Method begins at RVA 0x3c0d
// Code size 7 (0x7)
.maxstack 8

IL_0000: ldarg.0
IL_0001: call instance void [mscorlib]System.Object::.ctor()
IL_0006: ret
} // end of method Test::.ctor

} // end of class FastReflectionTests.Test
```

همانطور که مشاهده می‌کنید، کدهای IL با یک برچسب شروع می‌شوند مانند IL_0000. پس از آن OpCodes یا Operation codes قرار دارند. برای مثال ldc کار load constant را انجام می‌دهد. به این ترتیب مقدار ثابت 10 بارگذاری شده و بر روی پشته ارزیابی قرار داده می‌شود و نهایتا در سمت راست، مقادیر را ملاحظه می‌کنید؛ برای مثال مقادیری مانند 10 و 20.

این کدها در حالت کامپایل Release تهیه شده‌اند و در این حالت، کامپایلر یک سری بهینه سازی‌هایی را جهت بهبود سرعت و

کاهش تعداد OpCodes مورد نیاز برای اجرا برنامه، اعمال می‌کند.

بررسی OpCodes مقدماتی

الف) OpCodes ریاضی
مانند Add، Sub، Mul و Div

ب) OpCodes کنترل جریان برنامه
مانند Jmp، Beq، Bge، Ble، Bne، Call و Ret
برای پرش به یک برچسب، بررسی تساوی و بزرگتر یا کوچک بودن، فراخوانی متدها و بازگشت دادن مقادیر

ج) OpCodes مدیریت آرگومان‌ها
مانند Ldarg_0، Ldarg تا Ldc_I4 ، Ldarg_3 و Ldc_I4_1 تا Ldc_I4_8
برای بارگذاری آرگومان‌ها و همچنین بارگذاری مقادیر قرار گرفته شده بر روی پشته ارزیابی.

برای توضیحات بهتر این موارد می‌توان کدهای IL فوق را بررسی کرد:

```
IL_0000: ldc.i4.s 10
IL_0002: stloc.0
IL_0003: ldc.i4.s 20
IL_0005: stloc.1
IL_0006: ldloc.0
IL_0007: ldc.i4.s 10
IL_0009: bne.un.s IL_0010
IL_000b: ldloc.1
IL_000c: ldc.i4.s 20
IL_000e: pop
IL_000f: pop
```

در اینجا تعدادی مقدار بر روی پشته ارزیابی بارگذاری می‌شوند. تساوی آن‌ها بررسی شده و نهایتا متد خاتمه می‌یابد.

Stack چیست و MSIL چگونه عمل می‌کنید؟

Stack یکی از انواع بسیار متداول ساختار داده‌ها است و اگر بخواهیم خارج از دنیای رایانه‌ها مثالی را برای آن ارائه دهیم می‌توان به تعدادی برگه کاغذ که بر روی یکدیگر قرار گرفته‌اند، اشاره کرد. زمانیکه نیاز باشد تا برگه‌ای از این پشته برداشته شود، باید از بالاترین سطح آن شروع کرد که به آن LIFO یا Last in First out نیز گفته می‌شود. چیزی که آخر از همه بر روی پشته قرار می‌گیرد، در ابتدا برداشته و خارج خواهد شد.
در دات نت، برای قرار دادن اطلاعات بر روی Stack از متد Push و برای بازیابی از متد Pop استفاده می‌شود. استفاده از متد Pop، سبب خذف آن شیء از پشته نیز می‌گردد.
MSIL نیز یک Stack based language است. MSIL برای مدیریت یک سری از موارد از Stack استفاده می‌کند؛ مانند: پارامترهای متدها، مقادیر بازگشتی و انجام محاسبات در متدها. OpCodes کار قرار دادن و بازیابی مقادیر را از Stack به عهده دارند. به تمام این‌ها در MSIL، پشته ارزیابی یا Evaluation stack نیز می‌گویند.

یک مثال: فرض کنید می‌خواهید جمع 5+10 را توسط MSIL شبیه سازی کنیم.
الف) مقدار 5 بر روی پشته ارزیابی قرار داده می‌شود.
ب) مقدار 10 بر روی پشته ارزیابی قرار داده می‌شود. این مورد سبب می‌شود که 5 یک سطح به عقب رانده شود. به این ترتیب اکنون 10 بر روی پشته است و پس از آن 5 قرار خواهد داشت.
ج) سپس OpCode ایی مساوی Add فراخوانی می‌شود.
د) این OpCode سبب می‌شود تا دو مقدار موجود در پشته Pop شوند.
ه) سپس Add، حاصل عملیات را مجددا بر روی پشته قرار می‌دهد.

یک استثناء

در MSIL برای مدیریت متغیرهای محلی تعریف شده در سطح یک تابع، از Stack استفاده نمی‌شود. این مورد شبیه سایر زبان‌های اسمبلی است که در آن‌ها می‌توان مقادیر را در برچسب‌ها یا رجیسترهای خاصی نیز ذخیره کرد.

نظرات خوانندگان

نویسنده: میثم هوشمند
تاریخ: ۱۳۹۲/۰۵/۱۴ ۰:۳۱

سلام جناب نصیری
ممنون. این تیپ مقالات خیلی جذابند و البته عمق خوبی به بینش برنامه نویس می‌دهد .
یک سوال
یادم هست در فروم برنامه نویس چشمم به مطلب خورده که نوشته بود امکان دارد که کدهای دات نت تبدیل به کدهای ماشین کرد که دیگر نیازی به نصب دات نت فریم ورک بر روی سیستم مقصد نباشد
یعنی میتوان تمام نیازمندیهای برنامه را از دل فریم ورک بیرون کشید و به برنامه اضافه کرد و در نهایت یک فایل اجرایی قبال اجرا بدون نیاز به فریم

ممکنه توضیح بدهید در این خصوص؟
ممنون و متشکرم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۴ ۰:۵۹

یک نمونه از این پروژه‌ها، پروژه `Code Refractor` است. خلاصه کاری که انجام می‌دهد شامل مراحل زیر است:
- اسمبلی دات نتی را می‌خواند و `bytecodes/operations` آن‌را استخراج می‌کند.
- پس از آن، نتیجه را تبدیل به یک کد میانی خاص خودش می‌کند.
- این کد میانی خاص خودش را به `C++` ترجمه می‌کند.
- نهایتا از یک کامپایلر `C++` برای تولید فایل اجرایی نهایی استفاده خواهد کرد.
اطلاعات بیشتر

نویسنده: ابراهیم بیاگوی
تاریخ: ۱۳۹۲/۰۶/۰۲ ۱۲:۹

از اینکه این دوره را برای کسانی که در ۳۰ روز گذشته پستی نداشتند آزاد کردید تشکر می‌کنم.

عنوان: **مثال‌هایی بیشتر از عملکرد پشته‌ای MSIL**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۳ ۱۵:۱۶
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

بررسی عملکرد و کدهای IL یک متد

```
ldarg.0
        stloc_0
L_0000:
        ldloc_0
        ldc_i4 5
        add
        stloc_0
        ldloc_0
        ldc_i4 15
        blt L_0000
        ldloc_0
        ret
```

به کدهای IL فوق دقت کنید. در ادامه قصد داریم عملکرد این متد را بررسی کرده و سپس سعی کنیم تا معادل سی شارپ آن‌را حدس بزنیم. (البته سعی کنید طوری مطلب را مطالعه کنید که ادامه بحث را در ابتدا مشاهده نکنید!)
این متد، یک مقدار int را دریافت کرده و با انجام محاسباتی بر روی آن، مقدار int دیگری را بازگشت می‌دهد.
کار با ldarg.0 شروع می‌شود. به این ترتیب آرگومان موجود در ایندکس صفر، بر روی پشته بارگذاری خواهد شد. فرض کنید ورودی 5 را به این متد ارسال کرده‌ایم.
سپس stloc_0 این مقدار را از پشته pop کرده و در یک متغیر محلی ذخیره می‌کند.
در ادامه برچسب L_0000 تعریف شده است. از برچسب‌ها برای انتقال جریان اجرایی برنامه استفاده می‌کنیم.
ldloc_0 به معنای بارگذاری متغیر محلی از ایندکس صفر است. به این ترتیب عدد 5 بر روی پشته ارزیابی قرار می‌گیرد.
توسط ldc_i4 5، یک i4 یا 32 بیتی یا int ایی با 4 بایت، به عنوان یک عدد ثابت بارگذاری می‌شود. این عدد نیز بر روی پشته ارزیابی قرار می‌گیرد.
در ادامه با فراخوانی Add، دو مقدار قرار گرفته بر روی پشته pop شده و نتیجه 10؛ مجددا بر روی پشته قرار می‌گیرد.
stloc_0 سبب می‌شود تا این عدد 10 در یک متغیر محلی در ایندکس صفر ذخیره شود.
با فراخوانی ldloc_0، این متغیر محلی به پشته ارزیابی منتقل می‌شود.
به کمک ldc_i4 15، یک عدد صحیح 4 بایتی با مقدار ثابت 15 بارگذاری می‌شود.
در ادامه blt بررسی می‌کند که اگر 10 کوچکتر است از 15 ایی که بر روی پشته قرار گرفته، آنگاه جریان عملیات را به برچسب L_0000 منتقل می‌کند (پرش به برچسب صورت خواهد گرفت).
اگر با سایر زبان‌های اسمبلی کار کرده باشید با lt، gt و امثال آن به طور قطع آشنایی دارید. در اینجا blt به معنای branch less than equal است.
بنابراین در ادامه مجددا همین اعمال فوق تکرار خواهند شد تا به ارزیابی blt جهت دو مقدار 15 با 15 برسیم. از آنجائیکه اینبار 15 کوچکتر از 15 نیست، سطر پس از آن یعنی ldloc_0 اجرا می‌شود که معادل است با بارگذاری 15 به پشته ارزیابی و سپس فراخوانی return می‌شود تا این مقدار را بازگشت دهد.

خوب؛ آیا می‌توانید کدهای معادل سی‌شارپ آن‌را حدس بزنید؟!

```
public static int Calculate(int x)
        {
            for (; x < 15; x += 5)
            {
            }
            return x;
        }
```

بله. متد محاسباتی که در ابتدای بحث کدهای IL آن‌را ملاحظه نمودید، یک چنین معادل سی‌شارپی دارد.

فراخوانی متدها در MSIL

برای فراخوانی متدها در کدهای IL از OpCode ایی به نام call استفاده می‌شود:

```
ldstr "hello world"
call void [mscorlib]System.Console::WriteLine(string)
```

در این مثال توسط ldstr، یک رشته بارگذاری شده و سپس توسط call اطلاعات متدی که باید فراخوانی شود، ذکر می‌گردد. همانطور که ملاحظه می‌کنید، امضای کامل متد نیاز است ذکر گردد؛ متدی از نوع void قرار گرفته در mscorlib با ذکر فضای نام و نام متد مورد نظر.

بررسی یک الگوریتم بازگشتی در MSIL

ابتدا متد بازگشتی ذیل را درنظر بگیرید:

```
public static int Calculate(int n)
        {
            if (n <= 1) return n;
            return n * Calculate(n - 1);
        }
```

اگر کد دی‌کامپایل شده‌ی آن را در ILSpy بررسی کنیم، به دستورات ذیل خواهیم رسید:

```
.method public hidebysig static
int32 Calculate (
int32 n
) cil managed
{
// Method begins at RVA 0x3c0d
// Code size 17 (0x11)
.maxstack 8

IL_0000: ldarg.0
IL_0001: ldc.i4.1
IL_0002: bgt.s IL_0006

IL_0004: ldarg.0
IL_0005: ret

IL_0006: ldarg.0
IL_0007: ldarg.0
IL_0008: ldc.i4.1
IL_0009: sub
IL_000a: call int32 FastReflectionTests.Test::Calculate(int32)
IL_000f: mul
IL_0010: ret
} // end of method Test::Calculate
```

در اینجا ابتدا مقدار آرگومان متد، توسط ldarg بارگذاری می‌شود. سپس مقدار ثابت یک بارگذاری می‌شود. توسط bgt بررسی خواهد شد اگر مقدار آرگومان (عدد اول) بزرگتر است از مقدار عدد ثابت یک (عدد دوم)، آنگاه به برچسب IL_0006 پرش صورت گیرد و قسمت ضرب بازگشتی انجام شود. در غیراینصورت آرگومان متد بارگذاری شده و در سطر IL_0005 کار خاتمه می‌یابد.
در سطرهای IL_0006 و IL_0007، دوبار کار بارگذاری آرگومان متد انجام شده است؛ یکبار برای عملیات ضرب و بار دیگر برای عملیات تفریق از یک.
سپس عدد ثابت یک بارگذاری شده و از مقدار آرگومان، توسط sub کسر خواهد شد. در ادامه متد Calculate به صورت بازگشتی فراخونی می‌گردد. در این حالت دوباره به سطر IL_0000 هدایت خواهیم شد و عملیات ادامه می‌یابد.

عنوان: **تولید پویای کد در زمان اجرا توسط Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۳ ۱۹:۲۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

در ادامه قصد داریم توسط امکانات Reflection به همراه کدهای IL، اشیایی را در زمان اجرا ایجاد کنیم.

Reflection چیست؟

Reflection چیزهایی هستند که با نگاه در یک آینه قابل مشاهده‌اند. در این حالت شخص می‌تواند قسمت‌های مختلف ظاهر خود را برانداز کرده یا قسمتی را تغییر دهید. اما این مساله چه ربطی به دنیای دات نت دارد؟ در دات نت با استفاده از Reflection می‌توان به اطلاعات اشیاء یک برنامه‌ی در حال اجرا دسترسی یافت. برای مثال نام کلاس‌های مختلف آن چیست یا درون کلاسی خاص، چه متدهایی قرار دارند. همچنین با استفاده از Reflection می‌توان رفتارهای جدیدی را نیز به کلاس‌ها و اشیاء افزود یا آن‌ها را تغییر داد.
همواره عنوان می‌شود که از Reflection به دلیل سربار بالای آن پرهیز کنید و تنها از آن به عنوان آخرین راه حل موجود استفاده نمائید و این دقیقا موردی است که در مباحث جاری بیشتر از آن استفاده خواهد شد: ساخت اشیاء جدید در زمان اجرا به کمک کدهای IL و امکانات Reflection

نگاهی به امکانات متداول Reflection

در مثال بعد، نگاهی خواهیم داشت به امکانات متداول Reflection، مانند دسترسی به متدها و خواص یک کلاس و تعویض مقدار یا فراخوانی آن‌ها:

```
using System;

namespace FastReflectionTests
{
    class Person
    {
        public string Name { set; get; }

        public string Speak()
        {
            return string.Format("Hello, my name is {0}.", this.Name);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //روش متداول
            var vahid = new Person { Name = "Vahid" };
            Console.WriteLine(vahid.Speak());

            var type = vahid.GetType();

            //نمایش متدهای یک کلاس
            var methods = type.GetMethods();
            foreach (var method in methods)
            {
                Console.WriteLine(method.Name);
            }

            //تغییر مقدار یک خاصیت
            var setNameMethod = type.GetMethod("set_Name");
            setNameMethod.Invoke(obj: vahid, parameters: new[] { "Ali" });

            //فراخوانی یک متد
            var speakMethod = type.GetMethod("Speak");
            var result = speakMethod.Invoke(obj: vahid, parameters: null);
            Console.WriteLine(result);
        }
    }
}
```

با خروجی ذیل

```
Hello, my name is Vahid.
set_Name
get_Name
Speak
ToString
Equals
GetHashCode
GetType
Hello, my name is Ali.
```

توضیحات:

در اینجا یک کلاس شخص با خاصیت نام او تعریف شده است؛ به همراه متدی که رشته‌ای را نمایش خواهد داد.

در متد Main برنامه، ابتدا یک وهله جدید از این شخص ایجاد شده و سپس به روش متداول، متد Speak آن فراخوانی گردیده است. در ادامه کار از امکانات Reflection برای انجام همین امور کمک گرفته شده است.

کار با دریافت نوع یک وهله شروع می‌شود. برای نمونه در اینجا توسط vahid.GetType به نوع وهله ساخته شده دسترسی یافته‌ایم. سپس با داشتن این type، می‌توان به کلیه امکانات Reflection دسترسی یافت. برای مثال توسط GetMethods، لیست کلیه متدهای موجود در کلاس شخص بازگشت داده می‌شود.

اگر به خروجی فوق دقت کنید، پس از سطر اول، 7 سطر بعدی نمایانگر متدهای موجود در کلاس شخص هستند. شاید عنوان کنید که این کلاس به نظر یک متد بیشتر ندارد. اما در دات نت اشیاء از شیء Object مشتق می‌شوند و چهار متد ،ToString ،Equals GetHashCode و GetType متعلق به آن هستند. همچنین خواص تعریف شده نیز در اصل به دو متد set و get به صورت خودکار در کدهای IL برنامه ترجمه خواهند شد. از همین متد set_Name در ادامه برای مقدار دهی خاصیت نام وهله ایجاد شده استفاده شده است.

همانطور که ملاحظه می‌کنید برای فراخوانی یک وهله از طریق Reflection، ابتدا توسط متد type.GetMethod می‌توان به آن دسترسی یافت و سپس با فراخوانی متد Invoke، می‌توان متد مدنظر را بر روی یک شیء مهیا با پارامترهایی که ذکر می‌کنیم، فراخوانی کرد. اگر این متد پارامتری ندارد، آن‌را نال قرار خواهیم داد.

تا اینجا مقدمه‌ای را ملاحظه نمودید که بیشتر جهت تکمیل بحث، حفظ روابط منطقی قسمت‌های مختلف آن و یادآوری مباحث مرتبط با Reflection ذکر شدند.

ایجاد اشیاء در زمان اجرای برنامه

یکی از کلاس‌های مهم Reflection که در منابع مختلف کمتر به آن پرداخته شده است، کلاس DynamicMethod آن است که از آن می‌توان برای ایجاد اشیاء و یا متدهایی پویا در زمان اجرا استفاده کرد. این کلاس قرار گرفته در فضای نام System.Reflection.Emit، دارای یک ILGenerator است که می‌توان به آن OpCodeهایی را اضافه کرد. زمانیکه کار ایجاد این متدپویا به پایان رسید، با استفاده از Delegates امکان دسترسی و اجرای این متد پویا وجود خواهد داشت.

یک مثال کامل را در این زمینه در ادامه ملاحظه می‌نمائید:

```
using System;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        static double Divider(int a, int b)
        {
            return a / b;
        }

        delegate double DividerDelegate(int a, int b);
        static void Main(string[] args)
        {
            //روش متداول
            Console.WriteLine(Divider(10, 2));
```

```
            //تعریف امضای متد
            var myMethod = new DynamicMethod(
                                        name: "DividerMethod",
                                        returnType: typeof(double),
                                        parameterTypes: new[] { typeof(int), typeof(int) },
                                        m: typeof(Program).Module);
            //تعریف بدنه متد
            var il = myMethod.GetILGenerator();
            il.Emit(opcode: OpCodes.Ldarg_0); //بارگذاری پارامتر اول بر روی پشته ارزیابی
            il.Emit(opcode: OpCodes.Ldarg_1); //بارگذاری پارامتر دوم بر روی پشته ارزیابی
            il.Emit(opcode: OpCodes.Div); // دو پارامتر از پشته ارزیابی دریافت و تقسیم خواهند شد
            il.Emit(opcode: OpCodes.Ret); // دریافت نتیجه نهایی از پشته ارزیابی و بازگشت آن

            //فراخوانی متد پویا
            //روش اول
            var result = myMethod.Invoke(obj: null, parameters: new object[] { 10, 2 });
            Console.WriteLine(result);

            //روش دوم
            var method = (DividerDelegate)myMethod.CreateDelegate(delegateType:
typeof(DividerDelegate));
            Console.WriteLine(method(10, 2));
        }
    }
}
```

توضیحات

در ابتدای این مثال جدید یک متد متداول تقسیم کننده دو عدد را ملاحظه می‌کنید. در ادامه قصد داریم overload دیگری از این متد را توسط کدهای MSIL در زمان اجرا ایجاد کنیم که دو پارامتر int را قبول می‌کند.

کار با وهله سازی کلاس DynamicMethod موجود در فضای نام System.Reflection.Emit شروع می‌شود. در اینجا کار تعریف امضای متد جدید باید صورت گیرد. برای مثال نام آن چیست، نوع خروجی آن کدام است. نوع پارامترهای آن چیست و نهایتا این متدی که قرار است به صورت پویا به برنامه اضافه شود، باید در کجا قرار گیرد. برای اینکار از Module خود کلاس Program برنامه استفاده شده است.

پس از تعریف امضای متد پویا، نوبت به تعریف بدنه‌ی آن می‌رسد. کار با دریافت یک ILGenerator که می‌توان در آن کدهای IL را وارد کرد شروع می‌شود. مابقی آن تعریف کدهای IL توسط متد Emit است و پیشتر با مقدمات اسمبلی دات نت در قسمت‌های قبلی مبحث جاری آشنا شده‌ایم. ابتدا دو Ldarg فراخوانی شده‌اند تا دو پارامتر ورودی متد را دریافت کنند. سپس Div بر روی آن‌ها صورت گرفته و نهایتا نتیجه بازگشت داده شده است.

خوب؛ تا اینجا موفق شدیم اولین متد پویای خود را ایجاد نمائیم. برای اجرا آن حداقل دو روش وجود دارد:

الف) فراخوانی متد Invoke بر روی آن. با توجه به اینکه قرار نیست این متد بر روی وهله‌ی خاصی اجرا شود، اولین پارامتر آن null وارد شده است و سپس پارامترهای این متد پویا توسط آرگومان دوم متد Invoke وارد شده‌اند.

ب) می‌توان این عملیات را اندکی شکیل‌تر کرد. برای اینکار پیش از متد Main برنامه یک delegate به نام DividerDelegate تعریف شده است. سپس با استفاده از متد CreateDelegate، خروجی این متد پویا را تبدیل به یک delegate کرده‌ایم. اینبار فراخوانی متد پویا بسیار شبیه به متدهای معمولی می‌شود.

نظرات خوانندگان

نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۲ ۲۰:۶

زمانیکه یک کلاس همراه با یه سری property با استفاده از Reflection.Emit ایجاد کنیم آیا امکانش هست که از این کلاس یک نمونه ایجاد کنیم و به property های آن مقدار بدیم؟ ممنون میشم راهنمایی کنید

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۲ ۲۰:۹

بله. در ادامه بحث در مطلب « ایجاد یک کلاس جدید پویا و وهله‌ای از آن در زمان اجرا توسط Reflection.Emit » به آن پرداخته شده.

نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۱:۴۵

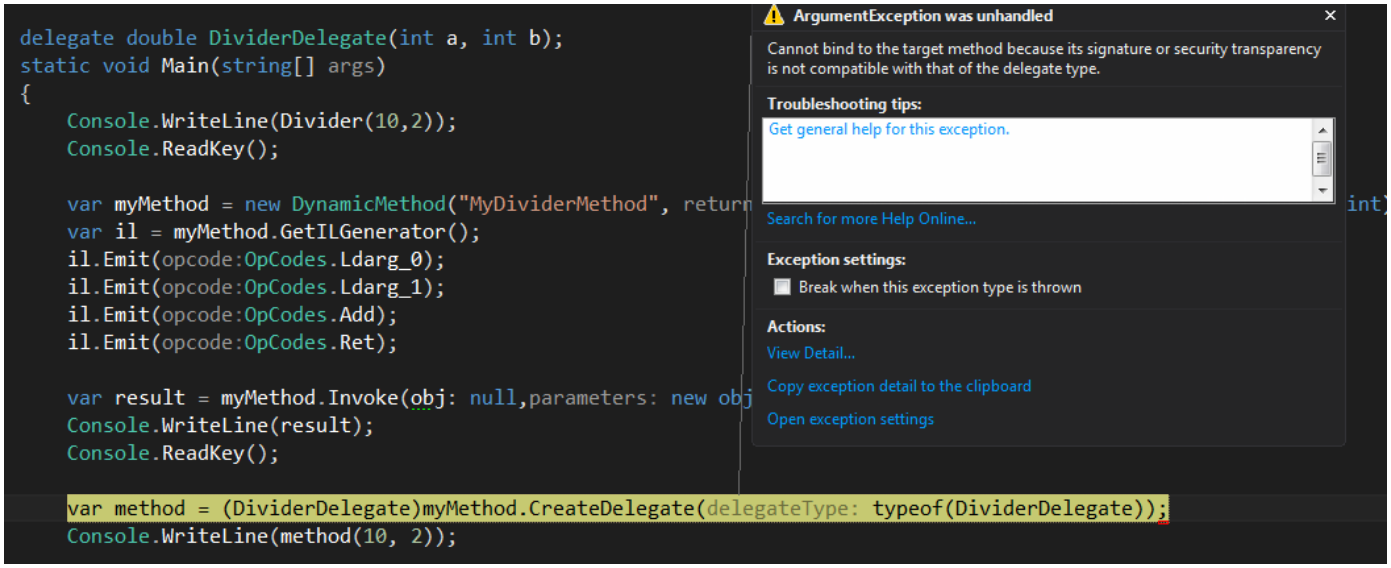
با سلام، من زمانی که می‌خواهم از روش دوم فراخوانی متد استفاده کنم با خطای زیر مواجه می‌شوم

```
var myMethod = new DynamicMethod("MyDividerMethod", returnType: typeof(int), parameterTypes: new[] {
typeof(int), typeof(int) }, m: typeof(Program).Module);
            var il = myMethod.GetILGenerator();
            il.Emit(opcode:OpCodes.Ldarg_0);
            il.Emit(opcode:OpCodes.Ldarg_1);
            il.Emit(opcode:OpCodes.Add);
            il.Emit(opcode:OpCodes.Ret);

            var result = myMethod.Invoke(obj: null,parameters: new object[] { 10, 2 });
            Console.WriteLine(result);
            Console.ReadKey();

            var method = (DividerDelegate)myMethod.CreateDelegate(delegateType:
typeof(DividerDelegate));
            Console.WriteLine(method(10, 2));
```

خطا



نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۱:۴۸

مشکل را متوجه شدم Signature تعریف Delegate من با متدی که تعریف کردم همخونی نداشت (int و double)
ممنونم

نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۱:۵۰

امکانش هست این قسمت را بیشتر توضیح بدید چون درست مفهومش رو متوجه نشدم
و **نهایتا این متدی که قرار است به صورت پویا به برنامه اضافه شود، باید در کجا قرار گیرد. برای اینکار از Module خود کلاس Program برنامه استفاده شده است.**
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۲:۵

تصویر قسمت‌ها و اجزای مختلف تشکیل دهنده یک اسمبلی، برای توضیحات بیشتر در مطلب « ایجاد یک اسمبلی جدید توسط Reflection.Emit » ارائه شده‌اند.

عنوان: **بررسی مثال‌ها و جزئیات بیشتر تولید کدهای پویا توسط Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۴ ۹:۴۲
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

نحوه معرفی متغیرهای محلی در Reflection.Emit

ابتدا مثال کامل ذیل را درنظر بگیرید:

```
using System;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        static int Calculate(int a, int b, int c)
        {
            var result = a * b;
            return result - c;
        }

        static void Main(string[] args)
        {
            //روش متداول
            Console.WriteLine(Calculate(10, 2, 3));

            //تعریف امضای متد
            var myMethod = new DynamicMethod(
                                        name: "CalculateMethod",
                                        returnType: typeof(int),
                                        parameterTypes: new[] { typeof(int), typeof(int), typeof(int)
},
                                        m: typeof(Program).Module);
            //تعریف بدنه متد
            var il = myMethod.GetILGenerator();

            il.Emit(opcode: OpCodes.Ldarg_0); // بارگذاری اولین آرگومان بر روی پشته ارزیابی
            il.Emit(opcode: OpCodes.Ldarg_1); // بارگذاری دومین آرگومان بر روی پشته ارزیابی
            il.Emit(opcode: OpCodes.Mul); // انجام عملیات ضرب
            il.Emit(opcode: OpCodes.Stloc_0); // ذخیره سازی نتیجه عملیات ضرب در یک متغیر محلی
            il.Emit(opcode: OpCodes.Ldloc_0); // متغیر محلی را بر روی پشته ارزیابی قرار می‌دهد تا در
عملیات بعدی قابل استفاده باشد
            il.Emit(opcode: OpCodes.Ldarg_2); // آرگومان سوم را بر روی پشته ارزیابی قرار می‌دهد
            il.Emit(opcode: OpCodes.Sub); // انجام عملیات تفریق
            il.Emit(opcode: OpCodes.Ret); // بازگشت نتیجه

            //فراخوانی متد پویا
            var method = (Func<int, int, int, int>)myMethod.CreateDelegate(typeof(Func<int, int, int,
int>));
            Console.WriteLine(method(10, 2, 3));

        }
    }
}
```

در این مثال سعی کرده‌ایم معادل متد Calculate را که در ابتدای برنامه ملاحظه می‌کنید، با کدهای IL تولید کنیم. روش کار مانند قسمت قبل است. ابتدا وهله‌ی جدیدی را از کلاس DynamicMethod جهت معرفی امضای متد پویای خود ایجاد می‌کنیم. در اینجا نوع خروجی را int و نوع سه پارامتر آن‌را به نحوی که مشخص شده است توسط آرایه‌ای از typeهای int معرفی خواهیم کرد. سپس محل قرارگیری کد تولیدی پویا مشخص می‌شود.

در ادامه توسط ILGenerator، آرگومان‌های دریافتی بارگذاری شده، در هم ضرب می‌شوند. سپس نتیجه در یک متغیر محلی ذخیره شده و سپس از آرگومان سوم کسر می‌گردد. در آخر هم این نتیجه بازگشت داده خواهد شد.

در اینجا روش سومی را برای کار با متدهای پویا مشاهده می‌کنید. بجای تعریف یک delegate به صورت صریح همانند قسمت قبل، از یک Func یا حتی Action نیز بنابر امضای متد مد نظر، می‌توان استفاده کرد. در اینجا از یک Func که سه پارامتر int را قبول کرده و خروجی int نیز دارد، استفاده شده است.

اگر برنامه را اجرا کنید ... کرش خواهد کرد! با استثنای ذیل:

```
System.InvalidProgramException was unhandled
Message=Common Language Runtime detected an invalid program.
```

علت اینجا است که در حین کار با System.Reflection.Emit، نیاز است نوع متغیر محلی مورد استفاده را نیز مشخص نمائیم. اینکار را توسط فراخوانی متد DeclareLocal که باید پس از فراخوانی GetILGenerator، درج گردد، می‌توان انجام داد:

```
il.DeclareLocal(typeof(int));
```

با این تغییر، برنامه بدون مشکل اجرا خواهد شد.

نحوه تعریف برچسب‌ها در Reflection.Emit

در ادامه قصد داریم یک مثال پیشرفته‌تر را بررسی کنیم.

```
static int Calculate(int x)
        {
            int result = 0;
            for (int i = 0; i < 10; i++)
            {
                result += i * x;
            }
            return result;
        }
```

در اینجا می‌خواهیم کدهای معادل متد محاسباتی فوق را توسط امکانات System.Reflection.Emit و کدهای IL تولید کنیم.

```
using System;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        static int Calculate(int x)
        {
            int result = 0;
            for (int i = 0; i < 10; i++)
            {
                result += i * x;
            }
            return result;
        }

        static void Main(string[] args)
        {
            //روش متداول
            Console.WriteLine(Calculate(10));

            //تعریف امضای متد
            var myMethod = new DynamicMethod(
                                        name: "CalculateMethod",
                                        returnType: typeof(int), // خروجی متد عدد صحیح است
                                        parameterTypes: new[] { typeof(int) }, // یک پارامتر عدد صحیح
دارد
                                        m: typeof(Program).Module);
            //تعریف بدنه متد
            var il = myMethod.GetILGenerator();

            // از برچسب‌ها برای انتقال کنترل استفاده می‌شود
            // در اینجا به دو برچسب برای تعریف ابتدای حلقه
            // و همچنین برای پرش به جایی که متد خاتمه می‌یابد نیاز داریم
            var loopStart = il.DefineLabel();
            var methodEnd = il.DefineLabel();

            // variable 0; result = 0
            il.DeclareLocal(typeof(int)); //  برای تعریف متغیر محلی نتیجه عملیات
            il.Emit(OpCodes.Ldc_I4_0); // عدد ثابت صفر را بر روی پشته ارزیابی قرار می‌دهد
            il.Emit(OpCodes.Stloc_0); // و نهایتا این عدد ثابت به متغیر محلی انتساب داده خواهد شد
```

```
            // variable 1; i = 0
            il.DeclareLocal(typeof(int)); // در اینجا کار تعریف و مقدار دهی متغیر حلقه انجام می‌شود
            il.Emit(OpCodes.Ldc_I4_0); // عدد ثابت صفر را بر روی پشته ارزیابی قرار می‌دهد
            il.Emit(OpCodes.Stloc_1); // و نهایتا این عدد ثابت به متغیر حلقه در ایندکس یک انتساب داده
خواهد شد

            // در اینجا کار تعریف بدنه حلقه شروع می‌شود
            il.MarkLabel(loopStart); // شروع حلقه را علامتگذاری می‌کنیم تا بعدا بتوانیم به این نقطه پرش
نمائیم
            il.Emit(OpCodes.Ldloc_1); // در ادامه می‌خواهیم بررسی کنیم که آیا مقدار متغیر حلقه از عدد 10
کوچکتر است یا خیر
            il.Emit(OpCodes.Ldc_I4, 10); // عدد ثابت ده را بر روی پشته ارزیابی قرار می‌دهد
            // برای انجام بررسی‌های تساوی یا کوچکتر یا بزرگتر نیاز است ابتدا دو متغیر مدنظر بر روی پشته قرار
گیرند
            il.Emit(OpCodes.Bge, methodEnd);  // اگر اینطور نیست و مقدار متغیر از 10 کمتر نیست، کنترل
برنامه را به انتهای متد هدایت خواهیم کرد

            // i * x
            il.Emit(OpCodes.Ldloc_1); // مقدار متغیر حلقه را بر روی پشته قرار می‌دهد
            il.Emit(OpCodes.Ldarg_0); // مقدار اولین آرگومان متد را بر روی پشته قرار می‌دهد
            il.Emit(OpCodes.Mul); // انجام عملیات ضرب
            // نتیجه این عملیات اکنون بر روی پشته قرار گرفته است

            // result +=
            il.Emit(OpCodes.Ldloc_0); // متغیر نتیجه را بر روی پشته قرار می‌دهد
            il.Emit(OpCodes.Add); // اکنون عملیات جمع بر روی نتیجه ضرب قسمت قبل که بر روی پشته قرار
دارد و همچنین متغیر نتیجه انجام می‌شود
            il.Emit(OpCodes.Stloc_0); // ذخیره سازی نتیجه در متغیر محلی

            // i++
            // در اینجا کار افزایش متغیر حلقه انجام می‌شود
            il.Emit(OpCodes.Ldloc_1); // مقدار متغیر حلقه بر روی پشته قرار می‌گیرد
            il.Emit(OpCodes.Ldc_I4_1); // عدد ثابت یک بر روی پشته قرار می‌گیرد
            il.Emit(OpCodes.Add); // سپس این دو عدد بارگذاری شده با هم جمع خواهند شد
            il.Emit(OpCodes.Stloc_1); // نتیجه در متغیر حلقه ذخیره خواهد شد

            // مرحله بعد شبیه سازی حلقه با پرش به ابتدای برچسب آن است
            il.Emit(OpCodes.Br, loopStart);

            //در اینجا انتهای متد علامتگذاری شده است
            il.MarkLabel(methodEnd);
            il.Emit(OpCodes.Ldloc_0); // مقدار نتیجه بر روی پشته قرار داده شده
            il.Emit(OpCodes.Ret); // و بازگشت داده می‌شود

            //فراخوانی متد پویا
            var method = (Func<int, int>)myMethod.CreateDelegate(typeof(Func<int, int>));
            Console.WriteLine(method(10));
        }
    }
}
```

کد کامل معادل را به همراه کامنت گذاری سطر به سطر آن، ملاحظه می‌کنید. در اینجا نکته‌های جدید، نحوه تعریف برچسب‌ها و انتقال کنترل برنامه به آن‌ها هستند؛ جهت شبیه سازی حلقه و همچنین خاتمه آن و انتقال کنترل به انتهای متد.

فراخوانی متدها توسط کدهای پویای Reflection.Emit

در ادامه کدهای کامل یک مثال متد پویا را که متد print را فراخوانی می‌کند، ملاحظه می‌کنید:

```
using System;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        public static void print(int i)
        {
            Console.WriteLine("i: {0}", i);
        }

        static void Main(string[] args)
        {
            روش متداول//
```

```
            print(10);

            //تعریف امضای متد
            var myMethod = new DynamicMethod(
                                    name: "myMethod",
                                    returnType: typeof(void),
                                    parameterTypes: null, // پارامتری ندارد
                                    m: typeof(Program).Module);
            //تعریف بدنه متد
            var il = myMethod.GetILGenerator();
            il.Emit(OpCodes.Ldc_I4, 10); // عدد ثابت 10 را بر روی پشته قرار می‌دهد
            // اکنون این مقدار بر روی پشته است و از آن می‌توان برای فراخوانی متد پرینت استفاده کرد
            il.Emit(OpCodes.Call, typeof(Program).GetMethod("print"));
            il.Emit(OpCodes.Ret);


            //فراخوانی متد پویا
            var method = (Action)myMethod.CreateDelegate(typeof(Action));
            method();
        }
    }
}
```

در اینجا از OpCode مخصوص فراخوانی متدها به نام Call که در قسمت‌های قبل در مورد آن بحث شد، استفاده گردیده است.
برای اینکه امضای دقیقی را در اختیار آن قرار دهیم، می‌توان از Reflection استفاده کرد که نمونه‌ای از آن‌را در اینجا ملاحظه می‌کنید.
به علاوه چون خروجی امضای متد ما از نوع void است، اینبار delegate تعریف شده را از نوع Action تعریف کرده‌ایم و نه از نوع Func.

فراخوانی متدهای پویای Reflection.Emit توسط سایر متدهای پویای Reflection.Emit

فراخوانی یک متد پویای مشخص از طریق متدهای پویای دیگر نیز همانند مثال قبل است:

```
using System;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        static void Main(string[] args)
        {
            //تعریف امضای متد
            var myMethod = new DynamicMethod(
                                    name: "mulMethod",
                                    returnType: typeof(int),
                                    parameterTypes: new[] { typeof(int) },
                                    m: typeof(Program).Module);
            //تعریف بدنه متد
            var il = myMethod.GetILGenerator();
            il.Emit(OpCodes.Ldarg_0); // اولین آرگومان متد را بر روی پشته قرار می‌دهد
            il.Emit(OpCodes.Ldc_I4, 42); // عدد ثابت 42 را بر روی پشته قرار می‌دهد
            il.Emit(OpCodes.Mul); // ضرب این دو در هم
            il.Emit(OpCodes.Ret); // بازگشت نتیجه

            //فراخوانی متد پویا
            var method = (Func<int, int>)myMethod.CreateDelegate(typeof(Func<int, int>));
            Console.WriteLine(method(10));

            // فراخوانی متد پویای فوق در یک متد پویای دیگر
            var callerMethod = new DynamicMethod(
                                    name: "callerMethod",
                                    returnType: typeof(int),
                                    parameterTypes: new[] { typeof(int), typeof(int) },
                                    m: typeof(Program).Module);
            //تعریف بدنه متد
            var callerMethodIL = callerMethod.GetILGenerator();
            callerMethodIL.Emit(OpCodes.Ldarg_0); // پارامتر اول متد را بر روی پشته قرار می‌دهد
            callerMethodIL.Emit(OpCodes.Ldarg_1); // پارامتر دوم متد را بر روی پشته قرار می‌دهد
            callerMethodIL.Emit(OpCodes.Mul); // ضرب این دو در هم
            //حاصل ضرب اکنون بر روی پشته است که در فراخوانی بعدی استفاده می‌شود
            callerMethodIL.Emit(OpCodes.Call, myMethod); // فراخوانی یک متد پویای دیگر
```

```
            callerMethodIL.Emit(OpCodes.Ret);

            //فراخوانی متد پویای جدید
            var method2 = (Func<int, int, int>)callerMethod.CreateDelegate(typeof(Func<int, int,
int>));
            Console.WriteLine(method2(10, 2));
        }
    }
}
```

در مثال فوق ابتدا یک متد پویای ضرب را تعریف کرده‌ایم که عددی صحیح را دریافت و آن‌را در 42 ضرب می‌کند و نتیجه را بازگشت می‌دهد.
سپس متد پویای دومی تعریف شده است که دو عدد صحیح را دریافت و این دو را در هم ضرب کرده و سپس نتیجه را به عنوان پارامتر به متد پویای اول ارسال می‌کند.
هنگام فراخوانی OpCodes.Call، پارامتر دوم باید از نوع MethodInfo باشد. نوع یک DynamicMethod نیز همان MethodInfo است. بنابراین برای فراخوانی آن، کار خاصی نباید انجام شود و صرفا ذکر نام متغیر مرتبط با مد پویای مدنظر کفایت می‌کند.

عنوان: **ایجاد یک کلاس جدید پویا و وهله‌ای از آن در زمان اجرا توسط Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۴ ۲۱:۴۷
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

توانایی‌های Reflection.Emit صرفا به ایجاد متدهایی کاملا جدید و پویا در زمان اجرا محدود نمی‌شود. برای نمونه کلاس ذیل را درنظر بگیرید:

```
public class Person
    {
        private string _name;
        public string Name
        {
            get { return _name; }
        }

        public Person(string name)
        {
            _name = name;
        }
    }
```

در ادامه قصد داریم معادل این کلاس را به همراه وهله‌ای از آن، به صورتی کاملا پویا در زمان اجرا ایجاد کرده (تصور کنید این کلاس در برنامه وجود خارجی نداشته و تنها جهت درک بهتر کدهای IL ادامه بحث، معرفی گردیده است) و سپس مقداری را به سازنده آن ارسال کنیم.
کدهای کامل و توضیحات این typeBuilder را در ادامه ملاحظه می‌کنید:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        static void Main(string[] args)
        {
            //اسمبلی محل قرارگیری کدهای پویای نهایی در اینجا تعیین می‌شود//
            //حالت دسترسی به آن اجرایی درنظر گرفته شده، امکان تعیین حالت‌های دیگری مانند ذخیره سازی نیز
وجود دارد
            var assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(
                                    name: new AssemblyName("Demo"), access:
AssemblyBuilderAccess.Run);

            // اکنون داخل این اسمبلی یک ماژول جدید را برای قرار دادن کلاس جدید خود تعریف می‌کنیم
            var moduleBuilder = assemblyBuilder.DefineDynamicModule(name: "PersonModule");

            // کار ساخت نوع و کلاس جدید شخص عمومی از اینجا شروع می‌شود
            var typeBuilder = moduleBuilder.DefineType(name: "Person", attr: TypeAttributes.Public);

            // افزودن فیلد خصوصی نام تعریف شده در سطح کلاس شخص
            var nameField = typeBuilder.DefineField(fieldName: "_name",
                                                    type: typeof(string),
                                                    attributes: FieldAttributes.Private);

            // تعریف سازنده عمومی کلاس شخص که دارای یک آرگومان رشته‌ای است
            var ctor = typeBuilder.DefineConstructor(
                                    attributes: MethodAttributes.Public,
                                    callingConvention: CallingConventions.Standard,
                                    parameterTypes: new[] { typeof(string) });
            // تعریف بدنه سازنده کلاس شخص
            // در اینجا فیلد خصوصی تعریف شده در سطح کلاس باید مقدار دهی شود
            var ctorIL = ctor.GetILGenerator();
            // نکته‌ای در مورد سازنده‌ها
            ctorIL.Emit(OpCodes.Ldarg_0); // اندیس صفر در سازنده کلاس به وهله‌ای از کلاس جاری اشاره
می‌کند
            ctorIL.Emit(OpCodes.Ldarg_1); // بارگذاری آرگومان سازنده و قرار دادن آن روی پشته
            // مقدار دهی فیلد خصوصی نام که به وهله‌ای از کلاس جاری و مقدار آرگومان دریافتی نیاز دارد
            ctorIL.Emit(OpCodes.Stfld, nameField);
            ctorIL.Emit(OpCodes.Ret); // پایان کار سازنده

            // تعریف خاصیت رشته‌ای نام در کلاس شخص
```

```
            var nameProperty = typeBuilder.DefineProperty(
                                            name: "Name",
                                            attributes: PropertyAttributes.HasDefault,
                                            returnType: typeof(string),
                                            parameterTypes: null); // خاصیت پارامتر ورودی ندارد

            var namePropertyGetMethod = typeBuilder.DefineMethod(
                                            name: "get_Name",
                                            attributes: MethodAttributes.Public |
                //متد ویژه‌ای است که توسط کامپایلر پردازش و تشخیص داده می‌شود
                                                        MethodAttributes.SpecialName |
                                                        MethodAttributes.HideBySig,
                                            returnType: typeof(string),
                                            parameterTypes: Type.EmptyTypes);
            // اتصال گت متد به خاصیت رشته‌ای نام که پیشتر تعریف شد
            nameProperty.SetGetMethod(namePropertyGetMethod);

            // بدنه گت متد در اینجا تعریف خواهد شد
            var namePropertyGetMethodIL = namePropertyGetMethod.GetILGenerator();
            namePropertyGetMethodIL.Emit(OpCodes.Ldarg_0); // بارگذاری اشاره‌گری به وهله‌ای از کلاس جاری
در پشته
            namePropertyGetMethodIL.Emit(OpCodes.Ldfld, nameField); // بارگذاری فیلد نام
            namePropertyGetMethodIL.Emit(OpCodes.Ret);

            var t = typeBuilder.CreateType(); // نهایی سازی کار ایجاد نوع جدید

            // ایجاد وهله‌ای از نوع جدید که پارامتری رشته‌ای به سازنده آن ارسال می‌شود
            var instance = Activator.CreateInstance(t, "Vahid");

            // دسترسی به خاصیت نام
            var nProperty = t.GetProperty("Name");
            // و دریافت مقدار آن برای نمایش
            var result = nProperty.GetValue(instance, null);

            Console.WriteLine(result);
        }
    }
}
```

در اینجا ایجاد یک کلاس جدید با ایجاد یک TypeBuilder واقع در فضای نام System.Reflection.Emit آغاز می‌شود. پیش از آن نیاز است یک اسمبلی پویا و ماژولی در آن‌را برای قرار دادن کدهای پویای این TypeBuilder ایجاد کنیم. توضیحات مرتبط با دستورات مختلف را به صورت کامنت در کدهای فوق ملاحظه می‌کنید. با استفاده از TypeBuilder و متد DefineField آن می‌توان یک فیلد در سطح کلاس ایجاد کرد و یا توسط متد DefineConstructor آن، سازنده کلاس را با امضایی ویژه تعریف نمود و سپس با دسترسی به ILGenerator آن، بدنه این سازنده را همانند متدهای پویا ایجاد کرد.

اگر به کدهای فوق دقت کرده باشید، متد get_Name به خاصیت Name انتساب داده شده است. علت را در قسمت معرفی اجمالی Reflection زمانیکه لیست متدهای کلاس Person را نمایش دادیم، ملاحظه کرده‌اید. تمام خواص Auto implemented در دات نت، هر چند ظاهر ساده‌ای دارند اما در عمل به دو متد get_Name و set_Name در کدهای IL توسط کامپایلر تبدیل می‌شوند. به همین جهت در اینجا نیاز بود تا get_Name را نیز تعریف کنیم.

چند مثال تکمیلی

Populating a PropertyGrid using Reflection.Emit

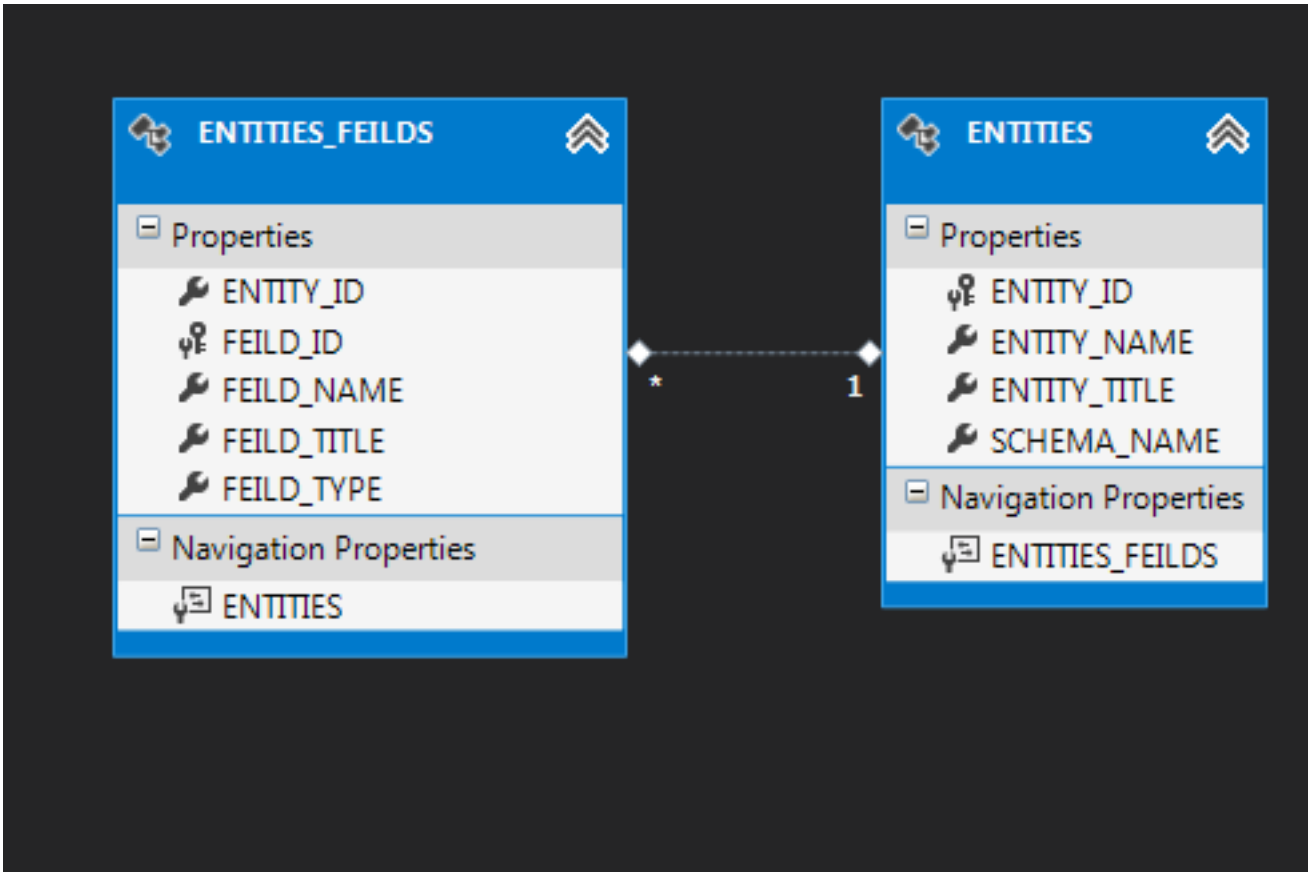
Dynamically adding RaisePropertyChanged to MVVM Light ViewModels using Reflection.Emit

نظرات خوانندگان

نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۴:۳۴

با سلام، من می‌خواهم در یکی از پروژه‌ها از این روش استفاده کنم و سناریویی که من روی اون کار می‌کنم به صورت زیر است:

درون دیتابیس، یک Table دارم که درون این Table نام تمام موجودیت‌های سیستم خودم رو نگه می‌دارم و در یک Table دیگر تمام فیلدهای موجودیت‌ها را همراه با نوع داده آنها ذخیره می‌کنم



برای یک سری شرایط خاص می‌خواهم کار زیر را انجام دهم:

یک فرم طراحی کردم که برای تمام موجودیت‌های تعریف شده درون جدول Entities کاربرد داره ، می‌خواهم زمانی که این فرم اجرا شده با توجه به اینکه این فرم برای کدام موجودیت فراخوانی شده است یک کلاس برای آن موجودیت ایجاد کنم و پس از آن یک لیست از کلاسی که ایجاد شده ، ایجاد بکنم و درون آن لیست مقادیری را قرار دهم (مقادیر را از دیتابیس خوانده می‌شود) و در آخر مقادیر لیست را در یک کنترل مثل gridview نمایش دهم

حال من برای انجام این کار به چند مشکل برخوردم . کدی که نوشتم به صورت زیر است

```
var ctx = new Entities();
            var Fields = ctx.ENTITIES_FEILDS.ToList();
            var assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(
                                    name: new AssemblyName("Demo"), access: AssemblyBuilderAccess.Run);

            var moduleBuilder = assemblyBuilder.DefineDynamicModule(name: "Module");

            var typeBuilder = moduleBuilder.DefineType(name:
```

```
Fields.First(c=>c.FEILD_ID==1).ENTITIES.ENTITY_NAME, attr: TypeAttributes.Public);

            foreach (var item in Fields)
            {

            }
```

در اینجا یک کلاس همنام با نام موجودیت ایجاد کردم و تمام فیلدهای این موجودیت را واکشی کردم حال می‌خواهم به ازای هر فیلد، یک Property ایجاد کنم. با توجه به مطلبی که در بالا فرمودید اگر ما تعداد فیلدهامون از قبل مشخص بود به راحتی می‌توانستیم این کار رو انجام بدیم ولی الان که مشخص نیست چگونه می‌توانیم Property خودمان را اضافه کنم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۴:۵۶

- در متن فوق جایی عنوان نشده که تنها اگر تعداد فیلدها از قبل مشخص بود، اینکار قابل انجام است. همچنین اگر به مثال بحث دقت کنید، پارامتر name رشته‌ای است. یعنی هر نام خاصیت دلخواهی قابل تعریف است. نوع آن نیز قابل مقدار دهی و تغییر است.
- در حلقه‌ای که نوشتید، کدهای «افزودن فیلد خصوصی» مثال بحث، «تعریف خاصیت رشته‌ای نام» ، «اتصال گت متد به خاصیت رشته‌ای نام» و «تعریف بدنه گت متد» باید به ازای هر خاصیت، تکرار شوند (پارامتر name را با نام خاصیت‌ها جایگزین کنید؛ نوع آن هم قابل تغییر است). اگر set هم دارد، علاوه بر متد گت، متد set_XYZ هم باید اضافه شود و روش کار یکی است.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۵:۸

برای ساده سازی و همچنین کپسوله کردن این عملیات، مراجعه کنید به مطالب زیر. در اینجا یک ClassGenerator با استفاده از Reflection Emit تهیه کرده‌اند:

Power of Reflection Emit

How to create a class with properties at run time

نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۵:۵۵

من کد رو به صورت زیر تغییر دادم

```
var ctx = new Entities();
            var Fields = ctx.ENTITIES_FEILDS.ToList();
            var assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(
                                    name: new AssemblyName("Demo"), access: AssemblyBuilderAccess.Run);

            var moduleBuilder = assemblyBuilder.DefineDynamicModule(name: "Module");

            var typeBuilder = moduleBuilder.DefineType(name: Fields.First(c => c.FEILD_ID ==
1).ENTITIES.ENTITY_NAME, attr: TypeAttributes.Public);

            foreach (var item in Fields)
            {
                switch (item.FEILD_TYPE)
                {
                    case 0://int
                        {
                            var intField = typeBuilder.DefineField(fieldName: string.Format("_{0}",
item.FEILD_NAME), type: typeof(string),
                                                   attributes: FieldAttributes.Private);

                            var intProperty = typeBuilder.DefineProperty(
                                                name: item.FEILD_NAME,
                                                attributes: PropertyAttributes.HasDefault,
                                                returnType: typeof(string),
                                                parameterTypes: null); // خاصیت پارامتر ورودی ندارد

                            //تعریف گت
                            var intpropertyGetMethod = typeBuilder.DefineMethod(
                                                               name: string.Format("get_{0}",
item.FEILD_NAME),
```

```
                                                    attributes: MethodAttributes.Public |
                           MethodAttributes.SpecialName |
                                                                MethodAttributes.HideBySig,
                                                    returnType: typeof(string),
                                                    parameterTypes: Type.EmptyTypes);

                                // اتصال گت متد به خاصیت عددی
                                intProperty.SetGetMethod(intpropertyGetMethod);


                                //تعریف ست
                                var propertySetMethod =
                                    typeBuilder.DefineMethod(name: string.Format("set_{0}",
item.FEILD_NAME),
                                    attributes: MethodAttributes.Public |
                                   MethodAttributes.SpecialName |
                                                                MethodAttributes.HideBySig,
                                                      returnType: typeof(void),
                                                    parameterTypes: new[] { typeof(string)
});
                                //اتصال ست متد
                                intProperty.SetSetMethod(propertySetMethod);


                                // بدنه گت متد در اینجا تعریف خواهد شد
                                var propertyGetMethodIL = intpropertyGetMethod.GetILGenerator();
                                propertyGetMethodIL.Emit(OpCodes.Ldarg_0); // بارگذاری اشاره‌گری به وهله‌ای از
کلاس جاری در پشته
                                propertyGetMethodIL.Emit(OpCodes.Ldfld, intField); // بارگذاری فیلد نام
                                propertyGetMethodIL.Emit(OpCodes.Ret);

                                //بدنه ست متد در اینجا تعریف شده است
                                var propertySetIL = propertySetMethod.GetILGenerator();
                                propertySetIL.Emit(OpCodes.Ldarg_0);
                                propertySetIL.Emit(OpCodes.Ldarg_1);
                                propertySetIL.Emit(OpCodes.Stfld, intField);
                                propertySetIL.Emit(OpCodes.Ret);

                            }

                            break;
                        case 1://string
                            {

                            } break;
                    }

                }
                var t = typeBuilder.CreateType();



                var instance = Activator.CreateInstance(t);

                var type = instance.GetType();

                //تغییر مقدار یک خاصیت
                var setNameMethod = type.GetMethod("set_CoOrder");
                setNameMethod.Invoke(obj: instance, parameters: new[] {"1"});


                // دسترسی به خاصیت نام
                var nProperty = t.GetProperty("CoOrder");
                // و دریافت مقدار آن برای نمایش
                var result = nProperty.GetValue(instance, null);

                Console.WriteLine(result);



                Console.ReadKey();
```

تا اینجا درست کار می‌کنه حال می‌خواهم از کلاسی که برای من ایجاد می‌کند یک لیست ایجاد کنم و بتونم بهش مقدار بدم. ولی هر چی تلاش کردم نتونستم کلاس خودم رو ایجاد کنم. ممنون میشم راهنمایی کنید

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۳ ۱۷:۴۹

الان لیست رو به صورت زیر ایجاد کنید

```
List<object> items = new List<object>();
```

هر آیتم این لیست، یک وهله از شیء پویایی خواهد بود که تهیه کردید.
گرید شما هم اگر حالت auto generate columns را پشتیبانی کند، بدون مشکل کار خواهد کرد.

نویسنده: پویا امینی
تاریخ: ۱۳۹۲/۰۵/۲۴ ۱۳:۵۱

یعنی اصلاً نمی‌توانم یه لیست از نوع کلاس ایجاد شده، تعریف کنم؟ چون گرید من auto generate columns نیست و من به صورت داینامیکcolumnها را مشخص می‌کنم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۴ ۱۴:۱۳

- وهله شیء تولیدی شما از نوع object است. آن‌را به لیست اضافه کنید و استفاده نمائید.
+ نوع جنریک در دات نت پویا نیست و نمی‌شود آن را به صورت یک متغیر تعریف کرد. مثلا حالت زیر مجاز نیست:

```
var myType = typeof(something);
List<myType> list = new List<myType>();
```

علت هم این است که هدف از نوع جنریک، compile time safety است و زمانیکه نوع در زمان کامپایل مشخص نباشد، این مساله قابل حصول نخواهد بود. تنها حالت پویای آن استفاده از نوع object است.
- البته می‌شود با استفاده Reflection نوع جنریک را به صورت متغیر تعریف کنید .

عنوان: **دسترسی سریع به مقادیر خواص توسط Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۵ ۱:۳
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

اگر پروژه‌های چندسال اخیر را مرور کرده باشید خصوصا در زمینه ORMها و یا Serializerها و کلا مواردی که با Reflection زیاد سروکار دارند، تعدادی از آن‌ها پیشوند fast را یدک می‌کشند و با ارائه نمودارهایی نشان می‌دهند که سرعت عملیات و کتابخانه‌های آن‌ها چندین برابر کتابخانه‌های معمولی است و ... سؤال مهم اینجا است که رمز و راز این‌ها چیست؟
فرض کنید تعاریف کلاس User به صورت زیر است:

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

همانطور که در قسمت‌های قبل نیز عنوان شد، خاصیت Id در کدهای IL نهایی به صورت متدهای get_Id و set_Id ظاهر می‌شوند.
حال اگر یک متد پویا ایجاد کنیم که بجای هر بار Reflection جهت دریافت مقدار Id، خود متد get_Id را مستقیما صدا بزند، چه خواهد شد؟
پیاده سازی این نکته را در ادامه ملاحظه می‌کنید:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Reflection;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    /// <summary>
    /// کلاسی برای اندازه گیری زمان اجرای عملیات
    /// </summary>
    public class Benchmark : IDisposable
    {
        Stopwatch _watch;
        string _name;

        public static Benchmark Start(string name)
        {
            return new Benchmark(name);
        }

        private Benchmark(string name)
        {
            _name = name;
            _watch = new Stopwatch();
            _watch.Start();
        }

        public void Dispose()
        {
            _watch.Stop();
            Console.WriteLine("{0} Total seconds: {1}"
                              , _name, _watch.Elapsed.TotalSeconds);
        }
    }

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

    class Program
    {
        public static Func<object, object> GetFastGetterFunc(string propertyName, Type ownerType)
        {
            var propertyInfo = ownerType.GetProperty(propertyName, BindingFlags.Instance |
BindingFlags.Public);
```

```
            if (propertyInfo == null)
                return null;

            var getter = ownerType.GetMethod("get_" + propertyInfo.Name,
                                             BindingFlags.Instance | BindingFlags.Public |
BindingFlags.FlattenHierarchy);
            if (getter == null)
                return null;

            var dynamicGetterMethod = new DynamicMethod(
                                              name: "_",
                                              returnType: typeof(object),
                                              parameterTypes: new[] { typeof(object) },
                                              owner: propertyInfo.DeclaringType,
                                              skipVisibility: true);
            var il = dynamicGetterMethod.GetILGenerator();

            il.Emit(OpCodes.Ldarg_0); // Load input to stack
            il.Emit(OpCodes.Castclass, propertyInfo.DeclaringType); // Cast to source type
            // نکته مهم در اینجا فراخوانی نهایی متد گت بدون استفاده از ریفلکشن است //
            il.Emit(OpCodes.Callvirt, getter); //calls its get method

            if (propertyInfo.PropertyType.IsValueType)
                il.Emit(OpCodes.Box, propertyInfo.PropertyType);//box

            il.Emit(OpCodes.Ret);

            return (Func<object, object>)dynamicGetterMethod.CreateDelegate(typeof(Func<object,
object>));
        }


        static void Main(string[] args)
        {
            //تهیه لیستی از داده‌ها جهت آزمایش
            var list = new List<User>();
            for (int i = 0; i < 1000000; i++)
            {
                list.Add(new User { Id = i });
            }

            // دسترسی به اطلاعات لیست به صورت متداول از طریق ریفلکشن معمولی
            var idProperty = typeof(User).GetProperty("Id");
            using (Benchmark.Start("Normal reflection"))
            {
                foreach (var item in list)
                {
                    var id = idProperty.GetValue(item, null);
                }
            }

            // دسترسی از طریق روش سریع دستیابی به اطلاعات خواص
            var fastIdProperty = GetFastGetterFunc("Id", typeof(User));
            using (Benchmark.Start("Fast Property"))
            {
                foreach (var item in list)
                {
                    var id = fastIdProperty(item);
                }
            }
        }
    }
}
```

توضیحات:

از کلاس Benchmark برای نمایش زمان انجام عملیات دریافت مقادیر Id از یک لیست، به دو روش Reflection متداول و روش صدا زدن مستقیم متد get_Id استفاده شده است.

در متد GetFastGetterFunc، ابتدا به متد get_Id خاصیت Id دسترسی پیدا خواهیم کرد. سپس یک متد پویا ایجاد می‌کنیم تا این get_Id را مستقیما صدا بزند. حاصل کار را به صورت یک delegate بازگشت می‌دهیم. شاید عنوان کنید که در اینجا هم حداقل در ابتدای کار متد، یک Reflection اولیه وجود دارد. پاسخ این است که مهم نیست؛ چون در یک برنامه واقعی، تهیه delegates در زمان آغاز برنامه انجام شده و حاصل کش می‌شود. بنابراین در زمان استفاده نهایی، به هیچ عنوان با سربار Reflection مواجه نخواهیم بود.

خروجی آزمایش فوق بر روی سیستم معمولی من به صورت زیر است:

```
Normal reflection Total seconds: 2.0054177
Fast Property Total seconds: 0.0552056
```

بله. نتیجه روش GetFastGetterFunc واقعا سریع و باور نکردنی است!

چند پروژه که از این روش استفاده می‌کنند

Dapper

AutoMapper

fastJson

در سورس این کتابخانه‌ها روش‌های فراخوانی مستقیم متدهای set نیز پیاده سازی شده‌اند که جهت تکمیل بحث می‌توان به آن‌ها مراجعه نمود.

ماخذ اصلی

این کشف و استفاده خاص، از اینجا شروع و عمومیت یافته است و پایه تمام کتابخانه‌هایی است که پیشوند fast را به خود داده‌اند:

faster using dynamic method calls 2000%

عنوان: **وهله سازی یک کلاس موجود توسط Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۵ ۲۱:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

در قسمت‌های قبل، نحوه ایجاد یک Type کاملا جدید را که در برنامه وجود خارجی ندارد، توسط Reflection.Emit بررسی کردیم. اکنون حالتی را در نظر بگیرید که کلاس مدنظر پیشتر در کدهای برنامه تعریف شده است، اما می‌خواهیم در یک DynamicMethod آن‌را وهله سازی کرده و حاصل را استفاده نمائیم.
کدهای کامل مثالی را در این زمینه در ادامه ملاحظه می‌کنید:

```
using System;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    public class Order
    {
        public string Name { set; get; }
        public Order()
        {
            Name = "Order01";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var myMethod = new DynamicMethod(name: "myMethod",
                                             returnType: typeof(Order),
                                             parameterTypes: Type.EmptyTypes,
                                             m: typeof(Program).Module);
            var il = myMethod.GetILGenerator();
            il.Emit(OpCodes.Newobj, typeof(Order).GetConstructor(Type.EmptyTypes));
            il.Emit(OpCodes.Ret);

            var getOrderMethod = (Func<Order>)myMethod.CreateDelegate(typeof(Func<Order>));

            Console.WriteLine(getOrderMethod().Name);
        }
    }
}
```

کار با ایجاد یک DynamicMethod شروع می‌شود. خروجی آن از نوع کلاس Order تعریف شده، پارامتری را نیز قبول نمی‌کند و برای تعریف آن از Type.EmptyTypes استفاده شده است.
سپس با دسترسی به ILGenerator سعی خواهیم کرد تا وهله جدیدی را از کلاس Order ایجاد کنیم. برای این منظور باید از OpCode جدیدی به نام Newobj استفاده کنیم که مخفف new object است. این OpCode برای عملکرد خود، نیاز به دریافت اشاره‌گری به سازنده کلاسی دارد که قرار است آن‌را وهله سازی کند. در اینجا با Ret، کار متد را خاتمه داده و در ادامه برای استفاده از آن تنها کافی است یک delegate را ایجاد نمائیم.

بنابراین به مجموعه متدهای سریع خود، متد ذیل را نیز می‌توان افزود:

```
public static Func<T> CreatFastObjectInstantiater<T>()
        {
            var t = typeof(T);
            var ctor = t.GetConstructor(Type.EmptyTypes);

            if (ctor == null)
                return null;

            var dynamicCtor = new DynamicMethod("_", t, Type.EmptyTypes, t, true);
            var il = dynamicCtor.GetILGenerator();
            il.Emit(OpCodes.Newobj, ctor);
            il.Emit(OpCodes.Ret);

            return (Func<T>)dynamicCtor.CreateDelegate(typeof(Func<T>));
        }
```

این نوع متدها که delegate بر می‌گردانند، باید یکبار در ابتدای برنامه ایجاد شده و نتیجه آن‌ها کش شوند. پس از آن به وهله سازی بسیار سریع دسترسی خواهیم داشت.

اگر علاقمند بودید که سرعت این روش را با روش متداول Activator.CreateInstance مقایسه کنید، مطلب زیر بسیار مفید است:

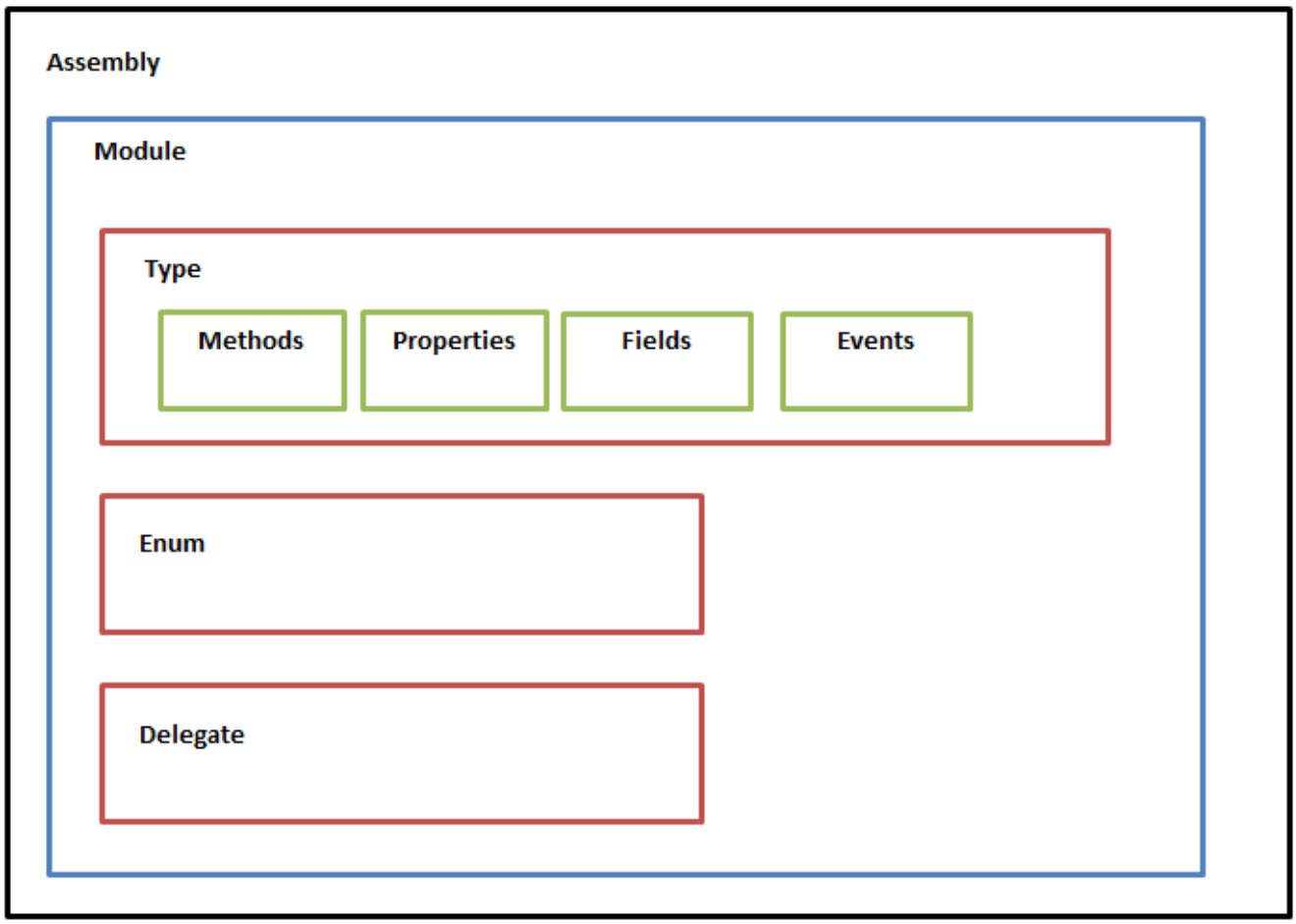
Creating objects - Perf implications

یک کاربرد مهم این مساله در نوشتن ORM مانندهایی است که قرار است لیستی جنریک را خیلی سریع تولید کنند؛ از این جهت که در حلقه DataReader آن‌ها مدام نیاز است یک وهله جدید از شیء مدنظر ایجاد و مقدار دهی شود:

Mapping Datareader to Objects Using Reflection.Emit

عنوان: **ایجاد یک اسمبلی جدید توسط Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۶ ۲۲:۵
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

مطابق استاندارد ECMA-335 قسمت دوم آن، یک اسمبلی از یک یا چند ماژول تشکیل می‌شود. هر ماژول از تعدادی نوع، enum و delegate تشکیل خواهد شد و هر نوع دارای تعدادی متد، فیلد، خاصیت و غیره می‌باشد. به همین جهت در حین کار با Reflection.Emit نیز این مراحل رعایت می‌شوند. ابتدا یک اسمبلی (AppDomain.DefineDynamicAssembly) ایجاد خواهد شد (یا از اسمبلی موجود استفاده می‌شود). سپس یک ماژول (AssemblyBuilder.DefineDynamicModule) را باید به آن اضافه کنیم (یا از ماژول اسمبلی جاری استفاده نمائیم). در ادامه یک Type باید به این ماژول اضافه شود (ModuleBuilder.DefineType) و اکنون می‌توان به این نوع جدید، سازنده (TypeBuilder.DefineConstructor)، متد (TypeBuilder.DefineMethod)، فیلد (TypeBuilder.DefineField)، خاصیت (TypeBuilder.DefineProperty) و رخداد (TypeBuilder.DefineEvent) اضافه کرد.




```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace FastReflectionTests
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            var name = "HelloWorld.exe";
            var assemblyName = new AssemblyName(name);
            // ایجاد یک اسمبلی جدید با قابلیت ذخیره سازی آن //
            var assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(
                                        name: assemblyName,
                                        access: AssemblyBuilderAccess.RunAndSave);
            // افزودن یک ماژول به اسمبلی //
            var moduleBuilder = assemblyBuilder.DefineDynamicModule(name);
            // تعریف یک کلاس در این ماژول //
            var programmClass = moduleBuilder.DefineType("Program", TypeAttributes.Public);
            // افزودن یک متد به این کلاس //
            // این متد خروجی ندارد اما ورودی آن شبیه به متد اصلی یک برنامه کنسول است //
            var mainMethod = programmClass.DefineMethod(name: "Main",
                                        attributes: MethodAttributes.Public |
MethodAttributes.Static,
                                        returnType: null,
                                        parameterTypes: new Type[] { typeof(string[]) });
            // تعیین بدنه متد اصلی برنامه //
            var il = mainMethod.GetILGenerator();
            il.Emit(OpCodes.Ldstr, "Hello World!");
            il.Emit(OpCodes.Call, (typeof(Console)).GetMethod("WriteLine", new Type[] { typeof(string)
}));
            il.Emit(OpCodes.Call, (typeof(Console)).GetMethod("ReadKey", new Type[0]));
            il.Emit(OpCodes.Pop);
            il.Emit(OpCodes.Ret);

            // تکمیل کار ایجاد نوع جدید //
            programmClass.CreateType();

            // تعیین نقطه شروع فایل اجرایی برنامه کنسول تهیه شده //
            assemblyBuilder.SetEntryPoint(((Type)programmClass).GetMethod("Main"));

            // ذخیره سازی این اسمبلی بر روی دیسک سخت //
            assemblyBuilder.Save(name);
        }
    }
}
```

مراحلی را که توضیح داده شد، در کدهای فوق ملاحظه می‌کنید. انتخاب حالت دسترسی AssemblyBuilderAccess.RunAndSave سبب می‌شود تا بتوان نتیجه حاصل را ذخیره کرد. فایل Exe نهایی را اگر در برنامه ILSpy باز کنیم چنین شکلی دارد:

```
using System;
public class Program
{
    public static void Main(string[] array)
    {
       Console.WriteLine("Hello World!");
       Console.ReadKey();
    }
}
```

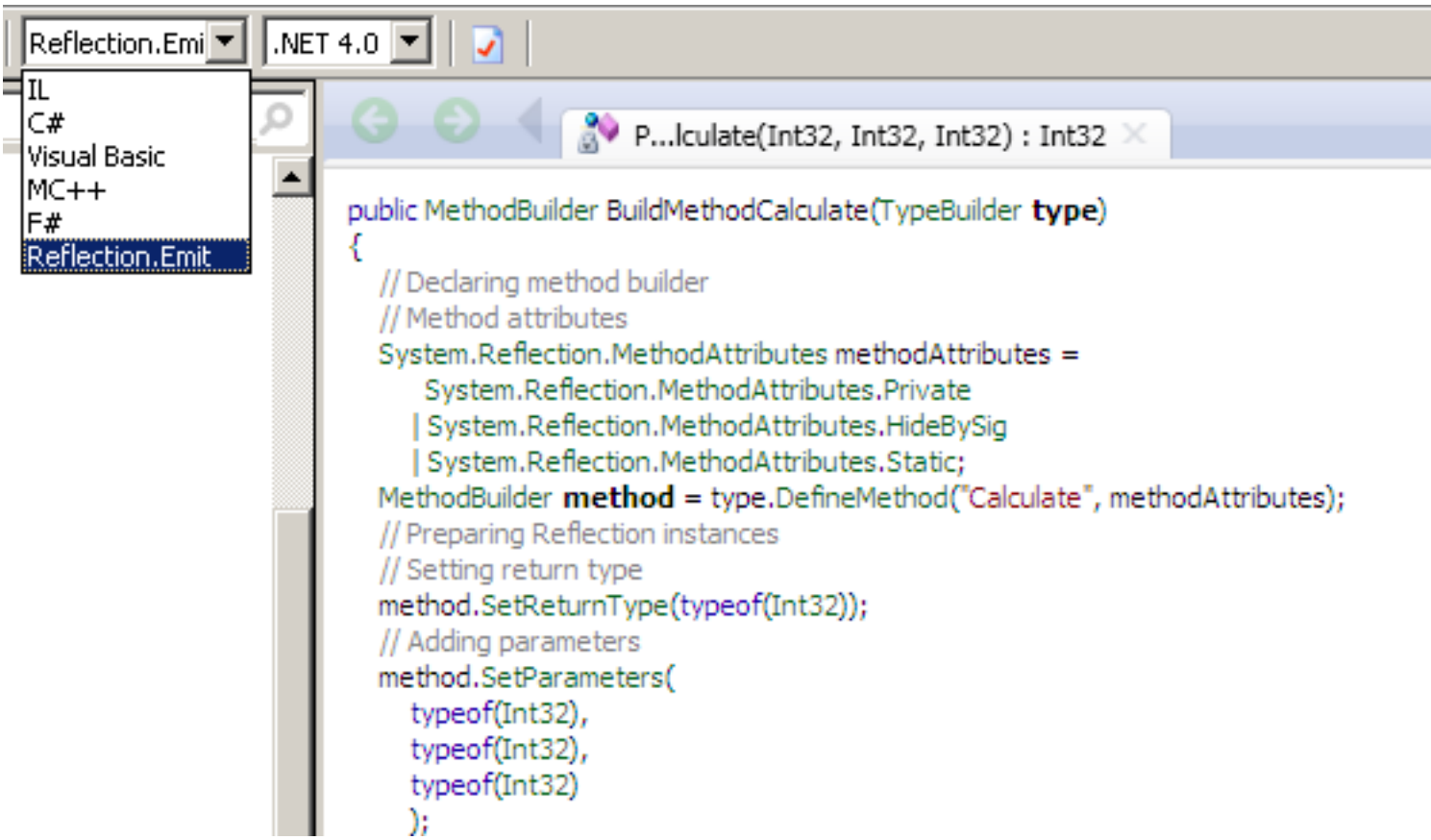
عنوان: **ابزاری برای تولید کدهای Reflection.Emit**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۷ ۱۰:۲۸
آدرس: www.dotnettips.info
برچسب‌ها: C#, CIL, CLR, IL, MSIL, Reflection

برنامه معروف Reflector دارای افزونه‌ای است به نام Reflector.ReflectionEmitLanguage که سورس آن از آدرس ذیل قابل دریافت است:

SourceControl/latest#Source/ReflectionEmitLanguage

مشخصات آن‌را نیز در آدرس زیر می‌توانید مشاهده نمائید:
ReflectionEmitLanguage

به این ترتیب به منوی انتخاب زبان‌های Reflector، یک زبان جدید به نام ReflectionEmit اضافه خواهد شد:



مشکل!
این افزونه مدت زیادی است که به روز نشده و با آخرین نگارش Reflector سازگار نیست. برای رفع این مشکل ابتدا سورس آن‌را از کدپلکس دریافت و سپس تغییرات ذیل را به آن اعمال کنید:
الف) به قسمت ارجاعات پروژه افزونه مراجعه و ارجاع به Reflector قدیمی آن‌را حذف و آدرس فایل exe برنامه Reflector جدید را به عنوان ارجاعی تازه، ثبت کنید.

ب) در فایل Visitor.cs آن باید تغییر کوچکی در متد ذیل به نحوی که مشاهده می‌کنید صورت گیرد:

```
public virtual void VisitOrderClause(IOrderClause value)
{
    this.VisitOrderClause(value);
}
```

پس از آن، پروژه را کامپایل کرده و فایل dll حاصل را در پوشه Addins نگارش جدید Reflector کپی کنید. سپس به منوی Tools و گزینه Addins در برنامه مراجعه کرده و آدرس فایل Reflector.ReflectionEmitLanguage.dll را برای معرفی به برنامه مشخص نمائید.
به این ترتیب نگارش قدیمی افزونه Reflector.ReflectionEmitLanguage.dll با نگارش جدید برنامه Reflector سازگار خواهد شد.
سورس تغییر یافته این افزونه را از اینجا نیز می‌توانید دریافت کنید:
ReflectionEmitLanguage.zip

بدیهی است به ازای هر نگارش جدید Reflector، یکبار باید قسمت الف توضیحات فوق تکرار شود.

عنوان: **موارد استفاده از Reflection.Emit در دنیای واقعی**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۱۸ ۱۳:۲۸
آدرس: www.dotnettips.info
گروه‌ها: C#, CIL, CLR, IL, MSIL, Reflection

1) در خود دات نت، Expression.Compile (موجود در فضای نام System.Linq.Expressions) در پشت صحنه از Reflection.Emit استفاده می‌کند.
2) چند مثال در قسمت‌های قبل مانند Dapper (که توسط نویسندگان Stack overflow تهیه شده) و fastJson ارائه شد که از Reflection.Emit برای دسترسی به متد get_XYZ یک خاصیت استفاده می‌کنند تا بجای Reflection، مستقیما به مقدار یک خاصیت دسترسی پیدا کنند و سرعت کار را به شدت بالا ببرند.
3) برای ایجاد dynamic proxies و مزین کردن کلاس‌ها و خواص آن‌ها در ORMهایی مانند NHibernate و یا حتی در پروژه Castle DynamicProxy و ... در فریم ورک‌های AOP.
4) اکثر کتابخانه‌های Mocking مانند Rhino Mocks و Moq از Reflection.Emit برای پیاده سازی خودکار اینترفیس‌ها و یا تهیه dynamic proxies استفاده می‌کنند.
5) DLR و اکثر زبان‌های مرتبط با آن استفاده گسترده‌ای از Reflection.Emit دارند.
6) برنامه معروف LINQPad از Reflection.Emit برای وهله سازی پویای اشیاء بهره می‌برد.

نظرات خوانندگان

نویسنده: ایزدی
تاریخ: ۱۳۹۳/۰۱/۲۷ ۸:۳۸

منظور از وهله سازی چیه ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۲۷ ۹:۱۹

نمونه سازی. instantiation

عنوان: **کتابخانه‌ی FastReflection**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۵/۳۰ ۱۴:۴۷
آدرس: www.dotnettips.info
گروه‌ها: C#, CIL, CLR, IL, MSIL, Reflection

در حین توسعه‌ی کتابخانه‌ی PdfReport نیاز به یک کتابخانه‌ی Reflection سریع با پشتیبانی از خواصی خصوصا تو در تو بود. حاصل مطلب « دسترسی سریع به مقادیر خواص توسط Reflection.Emit » تبدیل به کتابخانه‌ی FastReflection ذیل شد که هم اکنون در PdfReport مورد استفاده است:

FastReflection.zip

```
// کار با یک لیست جنریک تو در تو
            var list = new List<User>();
            for (int i = 0; i < 100; i++)
            {
                list.Add(new User
                {
                    Id = i+1,
                    Name = "name "+i,
                    Address = new Address
                    {
                        Address1 = "Addr1- "+i,
                        Address2 = "Addr2- "+i
                    }
                });
            }
            foreach (var item in list)
            {
                var propertyValues = new DumpNestedProperties().DumpPropertyValues(item, dumpLevel: 2);
                foreach (var result in propertyValues)
                {
                    Console.WriteLine(result.PropertyName + " -> " + result.PropertyValue);
                }
                Console.WriteLine();
            }
```

متد DumpPropertyValues ، توسط روش‌های Reflection.Emit تا تعداد سطحی را که مشخص می‌کنید، از شیء ارسالی به آن استخراج می‌کند. مباحث caching و استفاده مجدد از کدهای پویای تولید شده، در آن لحاظ شده و همچنین dumpLevel آن، از stack overflow در حین کار با پروکسی‌های پویای Entity framework جلوگیری می‌کند.