عنوان: **Roslyn #4**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۲۹ ۱۴:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

بررسی API کامپایل Roslyn

Compilation API، یک abstraction سطح بالا از فعالیت‌های کامپایل Roslyn است. برای مثال در اینجا می‌توان یک اسمبلی را از Syntax tree موجود، تولید کرد و یا جایگزین‌هایی را برای APIهای قدیمی CodeDOM و Reflection Emit ارائه داد. به علاوه این API امکان دسترسی به گزارشات خطاهای کامپایل را میسر می‌کند؛ به همراه دسترسی به اطلاعات Semantic analysis. در مورد تفاوت Syntax tree و Semantics در قسمت قبل بیشتر بحث شد.
با ارائه‌ی Roslyn، اینبار کامپایلرهای خط فرمان تولید شده مانند csc.exe، صرفا یک پوسته بر فراز Compilation API آن هستند. بنابراین دیگر نیازی به فراخوانی Process.Start بر روی فایل اجرایی csc.exe مانند یک سری کتابخانه‌های قدیمی نیست. در اینجا با کدنویسی، به تمام اجزاء و تنظیمات کامپایلر، دسترسی وجود دارد.

کامپایل پویای کد توسط Roslyn

برای کار با API کامپایل، سورس کد، به صورت یک رشته در اختیار کامپایلر قرار می‌گیرد؛ به همراه تنظیمات ارجاعاتی به اسمبلی‌هایی که نیاز دارد. سپس کار کامپایلر شروع خواهد شد و شامل مواردی است مانند تبدیل متن دریافتی به Syntax tree و همچنین تبدیل مواردی که اصطلاحا به آن‌ها Syntax sugars گفته می‌شود مانند خواص get و set دار به معادل‌های اصلی آن‌ها. در اینجا کار Semantic analysis هم انجام می‌شود و شامل تشخیص حوزه‌ی دید متغیرها، تشخیص overloadها و بررسی نوع‌های بکار رفته‌است. در نهایت کار تولید فایل باینری اسمبلی، از اطلاعات آنالیز شده صورت می‌گیرد. البته خروجی کامپایلر می‌تواند اسمبلی‌های exe یا dll، فایل XML مستندات اسمبلی و یا فایل‌های .netmudule و .winmdobj مخصوص WinRT هم باشد.
در ادامه، اولین مثال کار با Compilation API را مشاهده می‌کنید. پیشنیاز اجرای آن همان مواردی هستند که در قسمت قبل بحث شدند. یک برنامه‌ی کنسول ساده‌ی .NET 4.6 را آغاز کرده و سپس بسته‌ی نیوگت Microsoft.CodeAnalysis را در آن نصب کنید. در ادامه کدهای ذیل را به پروژه‌ی آماده شده اضافه کنید:

```
static void firstCompilation()
{
    var tree = CSharpSyntaxTree.ParseText("class Foo { void Bar(int x) {} }");

    var mscorlibReference = MetadataReference.CreateFromFile(typeof (object).Assembly.Location);

    var compilationOptions = new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary);

    var comp = CSharpCompilation.Create("Demo")
                                .AddSyntaxTrees(tree)
                                .AddReferences(mscorlibReference)
                                .WithOptions(compilationOptions);

    var res = comp.Emit("Demo.dll");

    if (!res.Success)
    {
        foreach (var diagnostic in res.Diagnostics)
        {
            Console.WriteLine(diagnostic.GetMessage());
        }
    }
}
```

در اینجا نحوه‌ی کامپایل پویای یک قطعه کد متنی سی‌شارپ را به DLL معادل آن مشاهده می‌کنید. مرحله‌ی اول اینکار، تولید Syntax tree از رشته‌ی متنی دریافتی است. سپس متد CSharpCompilation.Create یک وهله از Compilation API مخصوص C# را آغاز می‌کند. این API به صورت Fluent طراحی شده‌است و می‌توان سایر قسمت‌های آن‌را به همراه یک دات پس از ذکر متد، به طول زنجیره‌ی فراخوانی، اضافه کرد. برای نمونه در این مثال، نحوه‌ی افزودن ارجاعی را به اسمبلی mscorlib که System.Object در آن قرار دارد و همچنین ذکر نوع خروجی DLL یا DynamicallyLinkedLibrary را ملاحظه می‌کنید. اگر این تنظیم

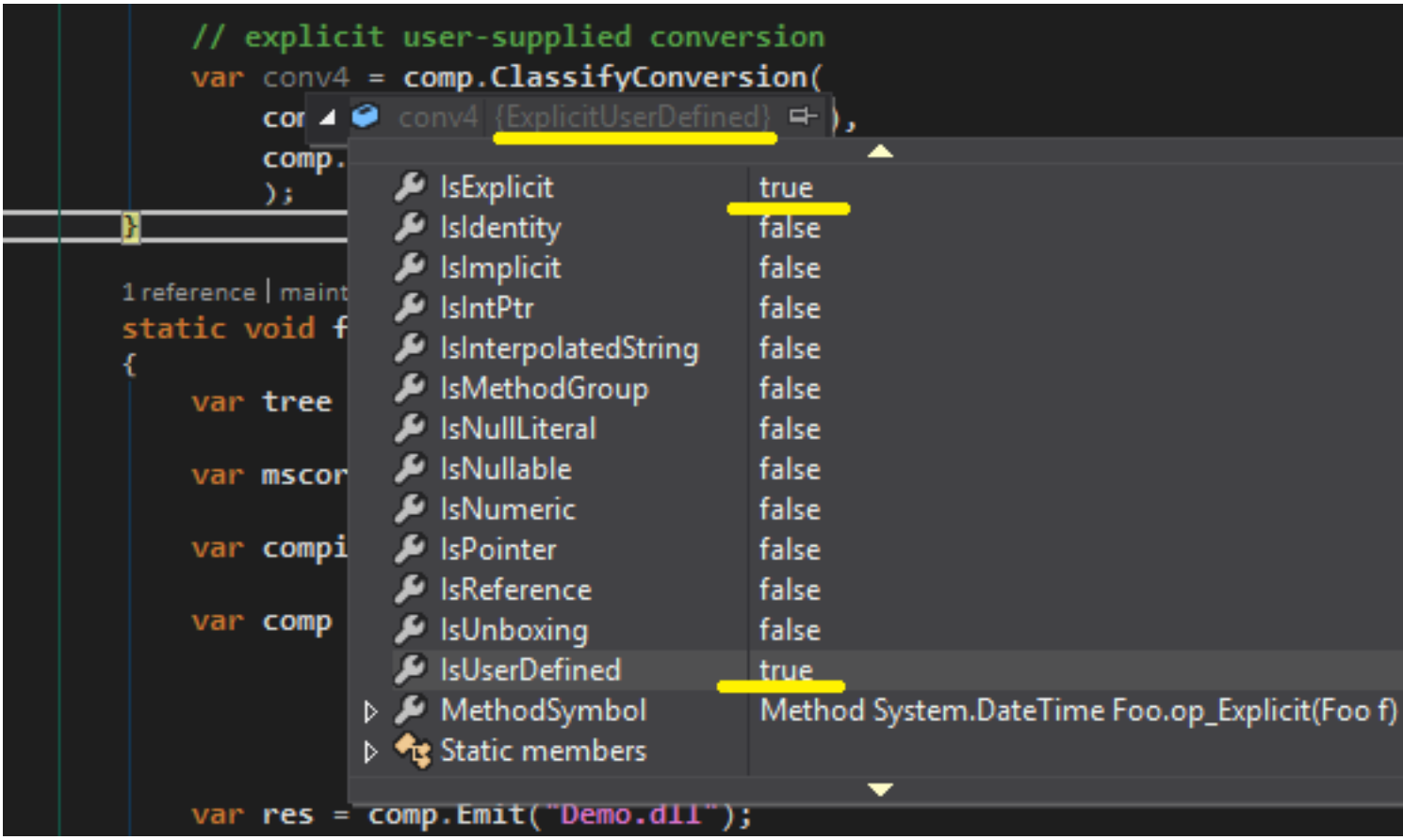
ذکر نشود، خروجی پیش فرض از نوع .exe خواهد بود و اگر mscorlib را اضافه نکنیم، نوع int سورس کد ورودی، شناسایی نشده و برنامه کامپایل نمی‌شود.
متدهای تعریف شده توسط Compilation API به یک s جمع، ختم می‌شوند؛ به این معنا که در اینجا در صورت نیاز، چندین Syntax tree یا ارجاع را می‌توان تعریف کرد.
پس از وهله سازی Compilation API و تنظیم آن، اکنون با فراخوانی متد Emit، کار تولید فایل اسمبلی نهایی صورت می‌گیرد. در اینجا اگر خطایی وجود داشته باشد، استثنایی را دریافت نخواهید کرد. بلکه باید خاصیت Success نتیجه‌ی آن‌را بررسی کرده و درصورت موفقیت آمیز نبودن عملیات، خطاهای دریافتی را از مجموعه‌ی Diagnostics آن دریافت کرد. کلاس Diagnostic، شامل اطلاعاتی مانند محل سطر و ستون وقوع مشکل و یا پیام متناظر با آن است.

معرفی مقدمات Semantic analysis

Compilation API به اطلاعات Semantics نیز دسترسی دارد. برای مثال آیا Type A قابل تبدیل به Type B هست یا اصلا نیازی به تبدیل ندارد و به صورت مستقیم قابل انتساب هستند؟ برای درک بهتر این مفهوم نیاز است یک مثال را بررسی کنیم:

```
static void semanticQuestions()
        {
            var tree = CSharpSyntaxTree.ParseText(@"
using System;

class Foo
{
    public static explicit operator DateTime(Foo f)
    {
        throw new NotImplementedException();
    }

    void Bar(int x)
    {
    }
}");

            var mscorlib = MetadataReference.CreateFromFile(typeof (object).Assembly.Location);
            var options = new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary);
            var comp =
CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib).WithOptions(options);
            // var res = comp.Emit("Demo.dll");

            // boxing
            var conv1 = comp.ClassifyConversion(
                comp.GetSpecialType(SpecialType.System_Int32),
                comp.GetSpecialType(SpecialType.System_Object)
                );

            // unboxing
            var conv2 = comp.ClassifyConversion(
                comp.GetSpecialType(SpecialType.System_Object),
                comp.GetSpecialType(SpecialType.System_Int32)
                );

            // explicit reference conversion
            var conv3 = comp.ClassifyConversion(
                comp.GetSpecialType(SpecialType.System_Object),
                comp.GetTypeByMetadataName("Foo")
                );

            // explicit user-supplied conversion
            var conv4 = comp.ClassifyConversion(
                comp.GetTypeByMetadataName("Foo"),
                comp.GetSpecialType(SpecialType.System_DateTime)
                );
        }
```

تا سطر CSharpCompilation.Create این مثال، مانند قبل است و تا اینجا به Compilation API دسترسی پیدا کرده‌ایم. پس از آن می‌خواهیم یک Semantic analysis مقدماتی را انجام دهیم. برای این منظور می‌توان از متد ClassifyConversion استفاده کرد. این متد یک نوع مبداء و یک نوع مقصد را دریافت می‌کند و بر اساس اطلاعاتی که از Compilation API بدست می‌آورد، می‌تواند مشخص کند که برای مثال آیا نوع کلاس Foo قابل تبدیل به DateTime هست یا خیر و اگر هست چه نوع تبدیلی را نیاز دارد؟



برای مثال نتیجه‌ی بررسی آخرین تبدیل انجام شده در تصویر فوق مشخص است. با توجه به تعریف `public static explicit operator DateTime` در سورس کد مورد آنالیز، این تبدیل `explicit` بوده و همچنین `user defined`. به علاوه متدی هم که این تبدیل را انجام می‌دهد، مشخص کرده‌است.