عنوان: **Roslyn #5**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۶/۳۰ ۱۵:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Roslyn

بررسی Semantic Models

همانطور که از قسمت قبل به‌خاطر دارید، برای دسترسی به اطلاعات semantics، نیاز به یک context مناسب که همان Compilation API است، می‌باشد. این context دارای اطلاعاتی مانند دسترسی به تمام نوع‌های تعریف شده‌ی توسط کاربر و متادیتاهای ارجاعی، مانند کلاس‌های پایه‌ی دات نت فریم‌ورک است. بنابراین پس از ایجاد وهله‌ای از Compilation API، کار با فراخوانی متد GetSemanticModel آن ادامه می‌یابد. در ادامه با مثال‌هایی، کاربرد این متد را بررسی خواهیم کرد.

ساختار جدید Optional

خروجی‌های تعدادی از متدهای Roslyn با ساختار جدیدی به نام Optional ارائه می‌شوند:

```
public struct Optional<T>
    {
        public bool HasValue { get; }
        public T Value { get; }
    }
```

این ساختار که بسیار شبیه است به ساختار قدیمی Nullable<T>، منحصر به Value types نیست و Reference types را نیز شامل می‌شود و بیانگر این است که آیا یک Reference type، واقعا مقدار دهی شده‌است یا خیر؟

دریافت مقادیر ثابت Literals

فرض کنید می‌خواهیم مقدار ثابت int x = 42 ; را دریافت کنیم. برای اینکار ابتدا باید syntax tree آن تشکیل شود و سپس نیاز به یک سری حلقه و if و else و همچنین بررسی نال بودن بسیاری از موارد است تا به نود مقدار ثابت 42 برسیم. سپس متد GetConstantValue مربوط به GetSemanticModel را بر روی آن فراخوانی می‌کنیم تا به مقدار واقعی آن که ممکن است در اثر محاسبات جاری تغییر کرده باشد، برسیم.
اما روش بهتر و توصیه شده، استفاده از CSharpSyntaxWalker است که در انتهای قسمت سوم معرفی شد:

```
class ConsoleWriteLineWalker : CSharpSyntaxWalker
{
    public ConsoleWriteLineWalker()
    {
        Arguments = new List<ExpressionSyntax>();
    }

    public List<ExpressionSyntax> Arguments { get; }

    public override void VisitInvocationExpression(InvocationExpressionSyntax node)
    {
        var member = node.Expression as MemberAccessExpressionSyntax;
        var type = member?.Expression as IdentifierNameSyntax;
        if (type != null && type.Identifier.Text == "Console" && member.Name.Identifier.Text ==
"WriteLine")
        {
            if (node.ArgumentList.Arguments.Count == 1)
            {
                var arg = node.ArgumentList.Arguments.Single().Expression;
                Arguments.Add(arg);
                return;
            }
        }

        base.VisitInvocationExpression(node);
    }
}
```

اگر به کدهای ادامه‌ی بحث دقت کنید، قصد داریم مقادیر ثابت آرگومان‌های Console.WriteLine را استخراج کنیم. به همین جهت در این SyntaxWalker، نوع Console و متد WriteLine آن مورد بررسی قرار گرفته‌اند. اگر این نود دارای یک تک آرگومان بود، آین آرگومان استخراج شده و به لیست آرگومان‌های خروجی این کلاس اضافه می‌شود.

در ادامه نحوه‌ی استفاده‌ی از این SyntaxWalker را ملاحظه می‌کنید. در اینجا ابتدا سورس کدی حاوی یک سری Console.WriteLine که دارای تک آرگومان‌های ثابتی هستند، تبدیل به syntax tree می‌شود. سپس از روی آن CSharpCompilation تولید می‌گردد تا بتوان به اطلاعات semantics دسترسی یافت:

```
static void getConstantValue()
{
    // Get the syntax tree.
    var code = @"
                using System;

                class Foo
                {
                    void Bar(int x)
                    {
                        Console.WriteLine(3.14);
                        Console.WriteLine(""qux"");
                        Console.WriteLine('c');
                        Console.WriteLine(null);
                        Console.WriteLine(x * 2 + 1);
                    }
                }
                ";

    var tree = CSharpSyntaxTree.ParseText(code);
    var root = tree.GetRoot();

    // Get the semantic model from the compilation.
    var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
    var comp = CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib);
    var model = comp.GetSemanticModel(tree);

    // Traverse the tree.
    var walker = new ConsoleWriteLineWalker();
    walker.Visit(root);


    // Analyze the constant argument (if any).
    foreach (var arg in walker.Arguments)
    {
        var val = model.GetConstantValue(arg);
        if (val.HasValue)
        {
            Console.WriteLine(arg + " has constant value " + (val.Value ?? "null") + " of type " +
(val.Value?.GetType() ?? typeof(object)));
        }
        else
        {
            Console.WriteLine(arg + " has no constant value");
        }
    }
}
```

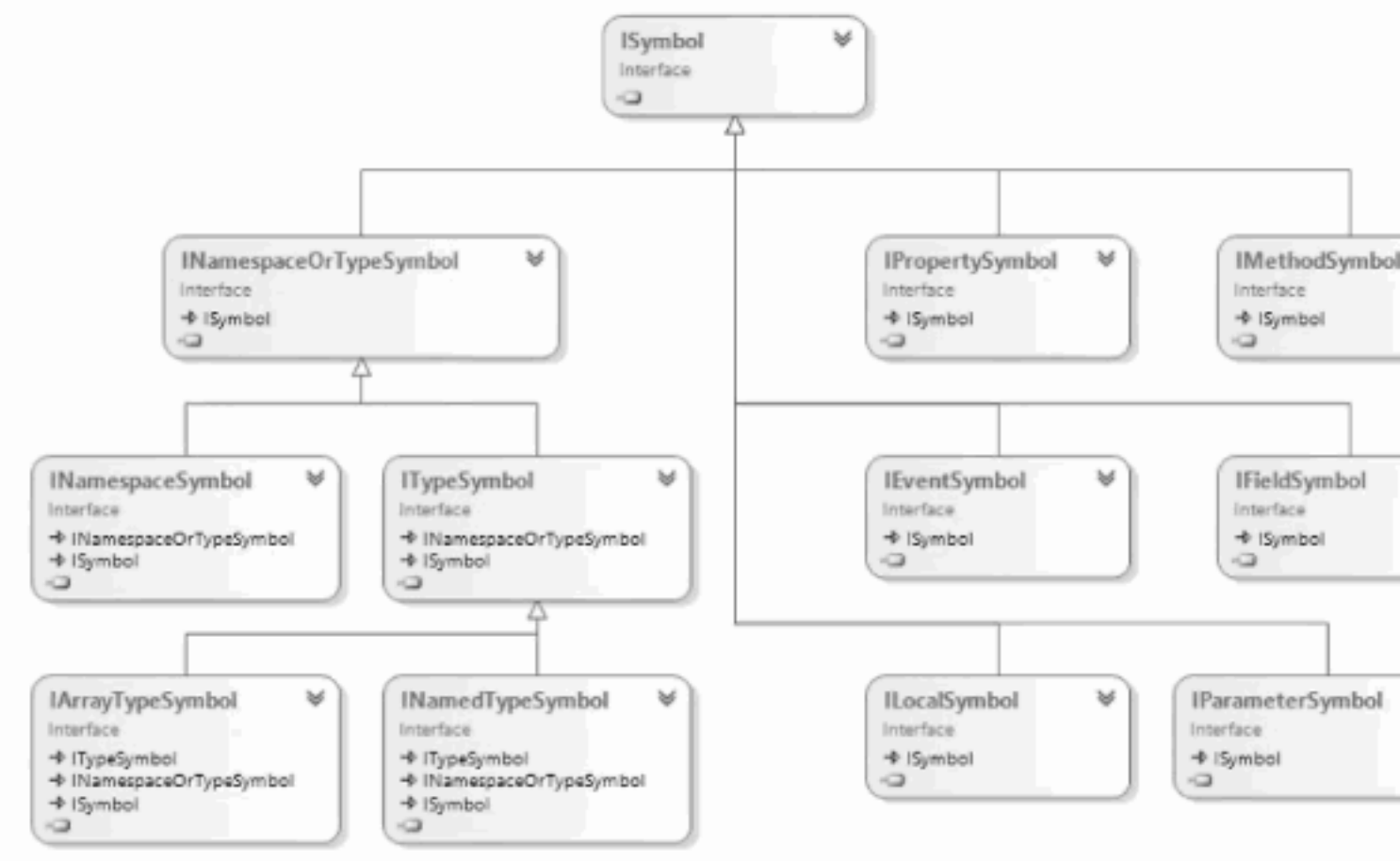
در ادامه با استفاده از CSharpCompilation و متد GetSemanticModel آن به SemanticModel جاری دسترسی خواهیم یافت. اکنون SyntaxWalker را وارد به حرکت بر روی ریشه‌ی syntax tree سورس کد آنالیز شده می‌کنیم. به این ترتیب لیست آرگومان‌های متدهای Console.WriteLine بدست می‌آیند. سپس با فراخوانی متد model.GetConstantValue بر روی هر آرگومان دریافتی، مقادیر آن‌ها با فرمت Optional<T> استخراج می‌شوند.

خروجی نمایش داده شده‌ی توسط برنامه به صورت ذیل است:

```
3.14 has constant value 3.14 of type System.Double
"qux" has constant value qux of type System.String
'c' has constant value c of type System.Char
null has constant value null of type System.Object
x * 2 + 1 has no constant value
```

درک مفهوم Symbols

اینترفیس ISymbol در Roslyn، ریشه‌ی تمام Symbolهای مختلف مدل سازی شده‌ی در آن است که تعدادی از آن‌ها را در تصویر ذیل مشاهده می‌کنید:



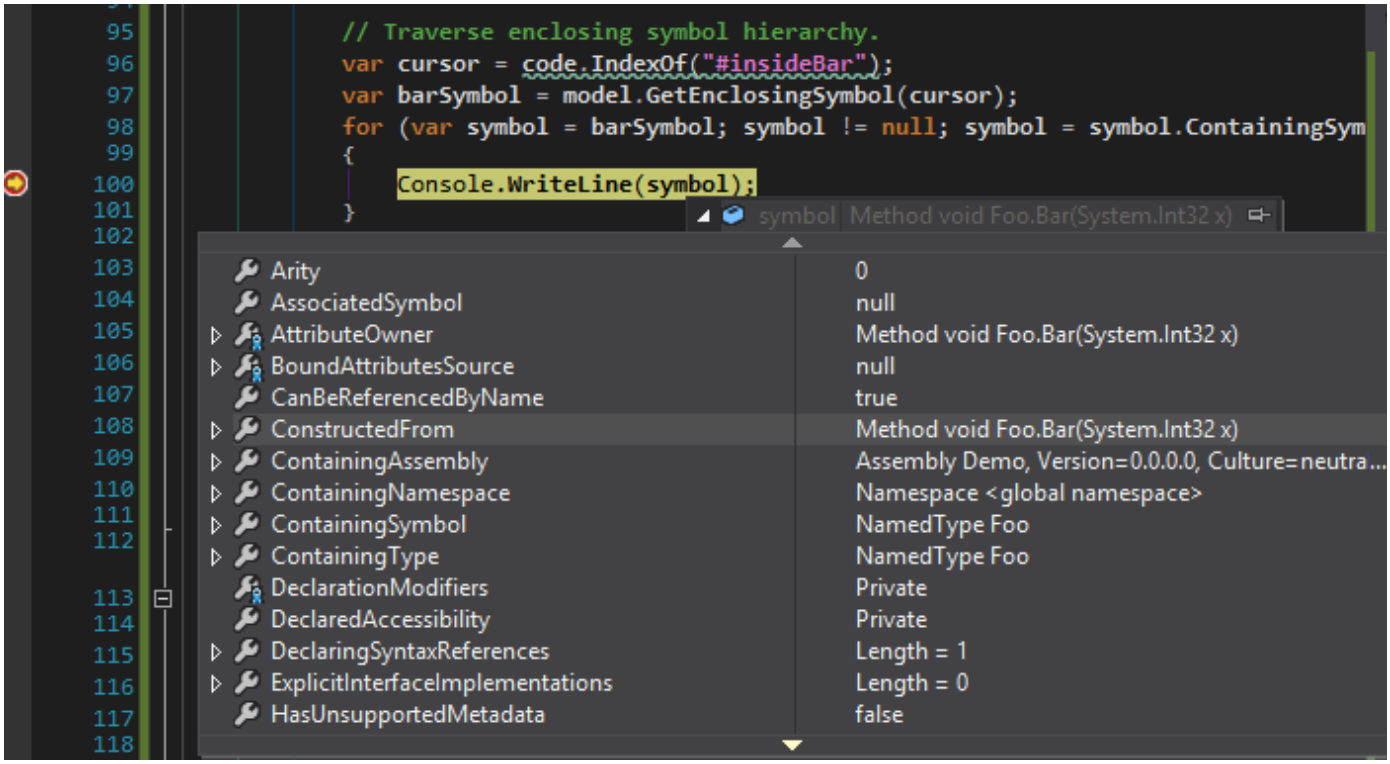
API کار با Symbols بسیار شبیه به API کار با Reflection است با این تفاوت که در زمان آنالیز کدها رخ می‌دهد و نه در زمان اجرای برنامه. همچنین در API Symbols امکان دسترسی به اطلاعاتی مانند labels, locals و امثال آن نیز وجود دارد که با استفاده از Reflection زمان اجرای برنامه قابل دسترسی نیستند. برای مثال فضاهای نام در Reflection صرفا به صورت رشته‌ای، با دات جدا شده از نوع‌های آنالیز شده‌ی توسط آن است؛ اما در اینجا مطابق تصویر فوق، یک اینترفیس مجزای خاص خود را دارد. جهت سهولت کار کردن با Symbols، الگوی Visitor با معرفی کلاس پایه‌ی SymbolVisitor نیز پیش بینی شده‌است.

```
static void workingWithSymbols()
{
    // Get the syntax tree.
    var code = @"
                using System;

                class Foo
                {
                    void Bar(int x)
                    {
                        // #insideBar
                    }
                }

                class Qux
                {
```

```
                    protected int Baz { get; set; }
                }
                ";

    var tree = CSharpSyntaxTree.ParseText(code);
    var root = tree.GetRoot();

    // Get the semantic model from the compilation.
    var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
    var comp = CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib);
    var model = comp.GetSemanticModel(tree);

    // Traverse enclosing symbol hierarchy.
    var cursor = code.IndexOf("#insideBar");
    var barSymbol = model.GetEnclosingSymbol(cursor);
    for (var symbol = barSymbol; symbol != null; symbol = symbol.ContainingSymbol)
    {
        Console.WriteLine(symbol);
    }

    // Analyze accessibility of Baz inside Bar.
    var bazProp = ((CompilationUnitSyntax)root)
        .Members.OfType<ClassDeclarationSyntax>()
        .Single(m => m.Identifier.Text == "Qux")
        .Members.OfType<PropertyDeclarationSyntax>()
        .Single();
    var bazSymbol = model.GetDeclaredSymbol(bazProp);
    var canAccess = model.IsAccessible(cursor, bazSymbol);
}
```

یکی از کاربردهای مهم Symbols API دریافت اطلاعات Symbols نقطه‌ای خاص از کدها می‌باشد. برای مثال در محل اشاره‌گر ادیتور، چه Symbols ایی تعریف شده‌اند و از آن‌ها در مباحث ساخت افزونه‌های آنالیز کدها زیاد استفاده می‌شود. نمونه‌ای از آن‌را در قطعه کد فوق ملاحظه می‌کنید. در اینجا با استفاده از متد GetEnclosingSymbol، سعی در یافتن Symbols قرار گرفته‌ی در ناحیه‌ی #insideBar کدهای فوق داریم؛ با خروجی ذیل که نام demo.exe آن از نام CSharpCompilation آن گرفته شده‌است:

```
Foo.Bar(int)
Foo
<global namespace>
Demo.exe
Demo, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```



همچنین در ادامه‌ی کد، توسط متد IsAccessible قصد داریم بررسی کنیم آیا Symbol قرار گرفته در محل کرسر، دسترسی به خاصیت protected کلاس Qux را دارد یا خیر؟ که پاسخ آن خیر است.

آشنایی با Binding symbols

یکی از مراحل کامپایل کد، binding نام دارد و در این مرحله است که اطلاعات Symbolic هر نود از Syntax tree دریافت می‌شود. برای مثال در اینجا مشخص می‌شود که این x، آیا یک متغیر محلی است، یا یک فیلد و یا یک خاصیت؟
مثال ذیل بسیار شبیه است به مثال getConstantValue ابتدای بحث، با این تفاوت که در حلقه‌ی آخر کار از متد GetSymbolInfo استفاده شده‌است:

```
static void bindingSymbols()
{
    // Get the syntax tree.
    var code = @"
                using System;

                class Foo
                {
                    private int y;

                    void Bar(int x)
                    {
                        Console.WriteLine(x);
                        Console.WriteLine(y);

                        int z = 42;
                        Console.WriteLine(z);

                        Console.WriteLine(a);
                    }
                }";

    var tree = CSharpSyntaxTree.ParseText(code);
    var root = tree.GetRoot();

    // Get the semantic model from the compilation.
    var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
    var comp = CSharpCompilation.Create("Demo").AddSyntaxTrees(tree).AddReferences(mscorlib);
```

```
    var model = comp.GetSemanticModel(tree);

    // Traverse the tree.
    var walker = new ConsoleWriteLineWalker();
    walker.Visit(root);

    // Bind the arguments.
    foreach (var arg in walker.Arguments)
    {
        var symbol = model.GetSymbolInfo(arg);
        if (symbol.Symbol != null)
        {
            Console.WriteLine(arg + " is bound to " + symbol.Symbol + " of type " +
symbol.Symbol.Kind);
        }
        else
        {
            Console.WriteLine(arg + " could not be bound");
        }
    }
}
```

با این خروجی:

```
x is bound to int of type Parameter
y is bound to Foo.y of type Field
z is bound to z of type Local
a could not be bound
```

در مثال فوق، با استفاده از Syntax Walker طراحی شده در ابتدای بحث که کار استخراج آرگومان‌های متدهای Console.WriteLine را انجام می‌دهد، قصد داریم بررسی کنیم، هر آرگومان به چه Symbol ایی بایند شده‌است و نوعش چیست؟ برای مثال Console.WriteLine اول که از پارامتر x استفاده می‌کند، نوع x مورد استفاده‌اش چیست؟ آیا فیلد است، متغیر محلی است یا یک پارامتر؟ این اطلاعات را با استفاده از متد model.GetSymbolInfo می‌توان استخراج کرد.