عنوان: **فعال سازی Multicore JIT**
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۳/۰۴/۲۸ ۱۷:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Performance, JIT, .NET 4.5, MultiCore JIT

Multicore JIT یکی از قابلیت‌های کلیدی در دات نت 4.5 می‌باشد که در واقع راه حلی برای بهبود سرعت اجرای برنامه‌های دات نتی است. قبل از معرفی این قابلیت ابتدا اجازه دهید نحوه کامپایل یک برنامه دات نتی را بررسی کنیم.

انواع compilation

در حالت کلی دو نوع فرآیند کامپایل داریم:

Explicit

در این حالت دستورات قبل از اجرای برنامه به زبان ماشین تبدیل می‌شوند. به این نوع کامپایلرها AOT یا Ahead Of Time گفته می‌شود. این نوع از کامپایلرها برای اطمینان از اینکه CPU بتواند قبل از انجام تعاملی تمام خطوط کد را تشخیص دهد، طراحی شده اند.

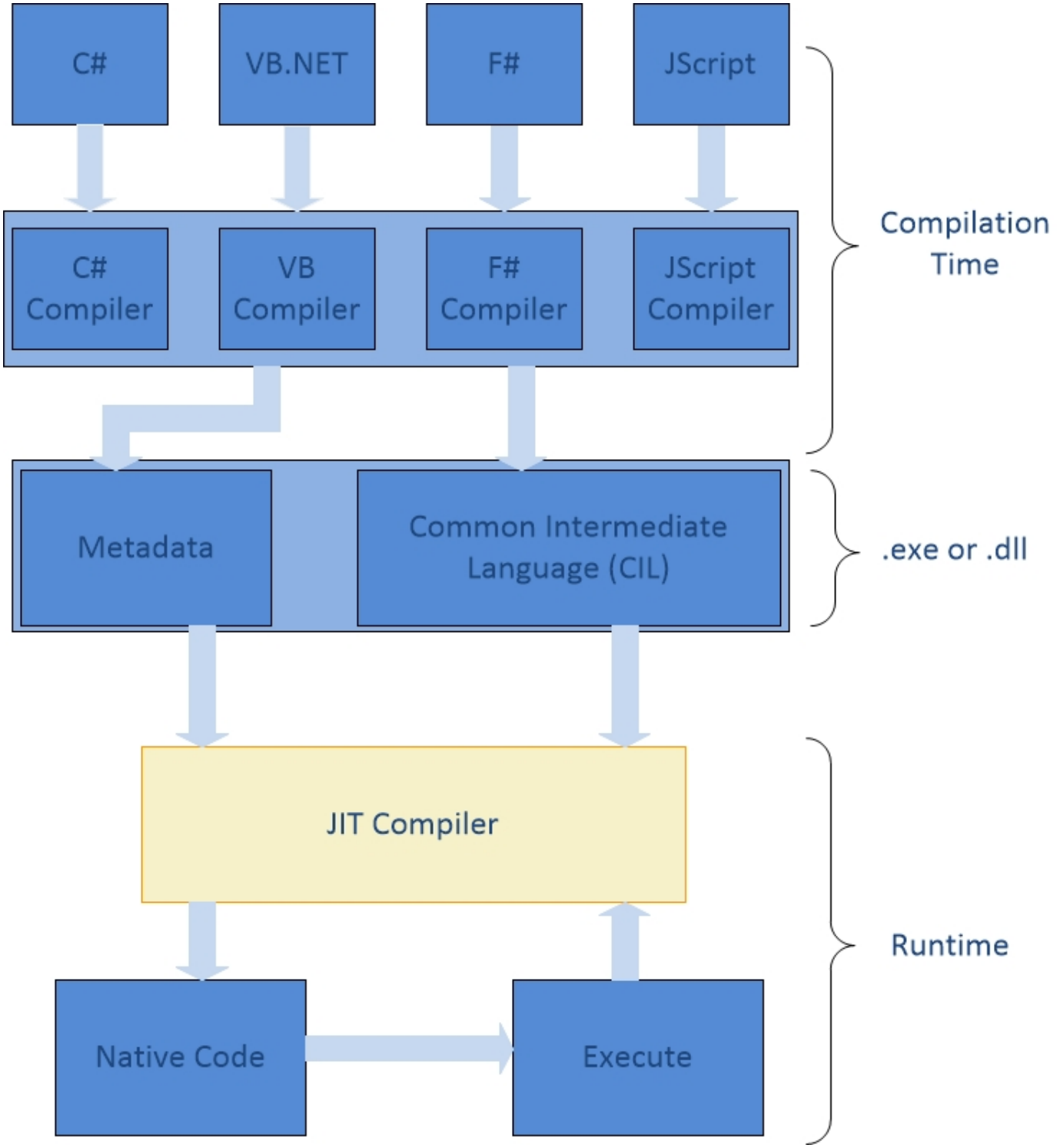
Implicit

این نوع compilation به صورت دو مرحله ایی صورت می‌گیرد. در اولین قدم سورس کد توسط یک کامپایلر به یک زبان سطح میانی(IL) تبدیل می‌شود. در مرحله بعدی کد IL به دستورات زبان ماشین تبدیل می‌شوند. در دات نت فریم ورک به این کامپایلر JIT یا Just-In-Time گفته می‌شود.

در حالت دوم قابلیت جابجایی برنامه به آسانی امکان پذیر است، زیرا اولین قدم از فرآیند به اصطلاح platform agnostic می‌باشد، یعنی قابلیت اجرا بر روی گستره وسیعی از پلت فرم‌ها را دارد.

کامپایلر JIT

JIT بخشی از Common Language Runtime یا CLR می‌باشد. CLR در واقع وظیفه مدیریت اجرای تمام برنامه‌های دات نتی را برعهده دارد.

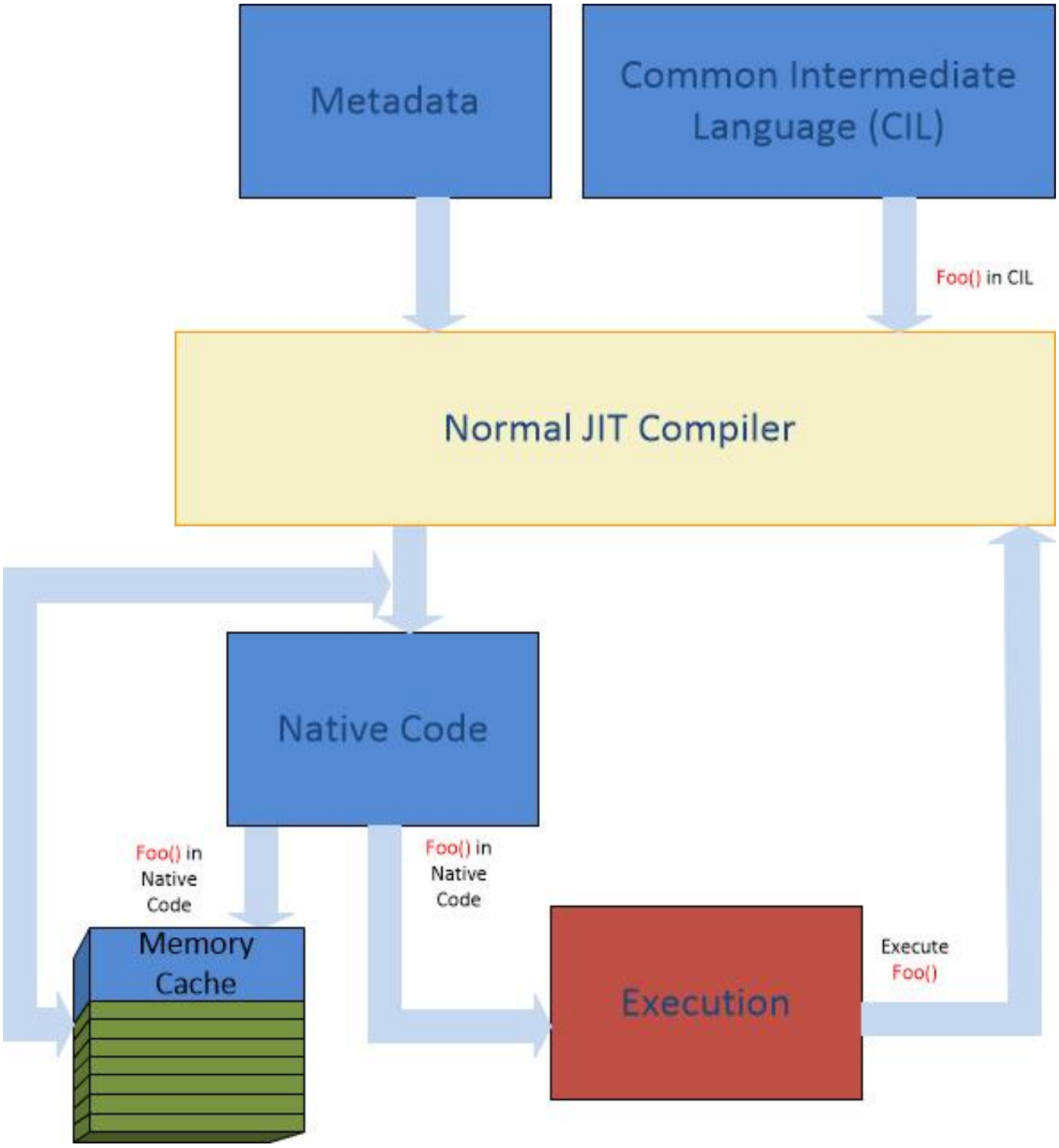


همانطور که در تصویر فوق مشاهده می‌کنید، سورس کد توسط کامپایلر دات نت به exe و یا dll کامپایل می‌شود. کامپایلر JIT تنها متدهایی را که در زمان اجرا(runtime) فراخوانی می‌شوند را کامپایل می‌کند. در دات نت فریم ورک سه نوع JIT Compilation داریم:

Normal JIT Compilation

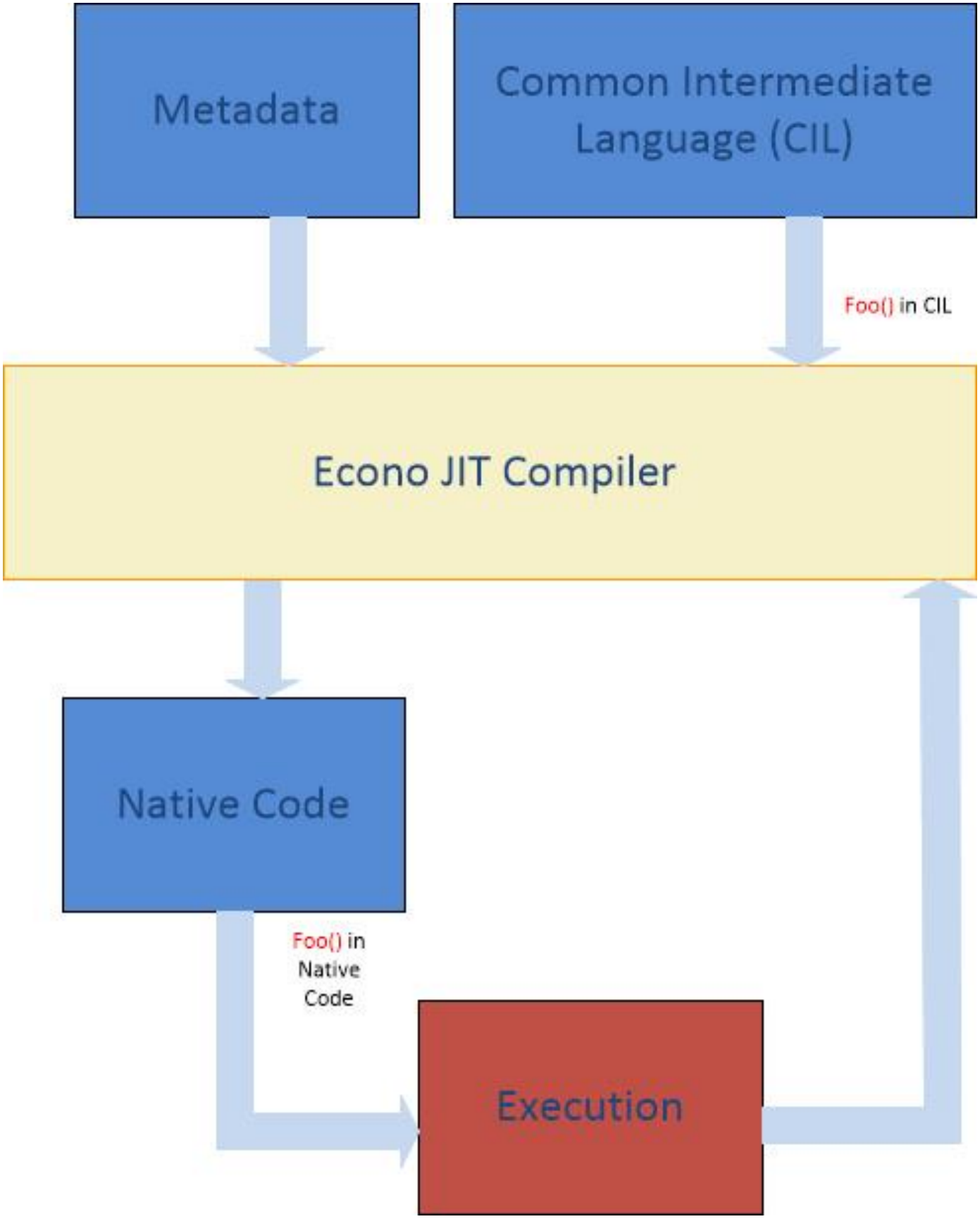
در این نوع کامپایل، متدها در زمان فراخوانی در زمان اجرا کامپایل می‌شوند. بعد از اجرا، متد داخل حافظه ذخیره می‌شود. به متدهای ذخیره شده در حافظه jitted گفته می‌شود. دیگر نیازی به کامپایل متد jit شده نیست. در فراخوانی بعدی، متد مستقیماً

از حافظه کش در دسترس خواهد بود.



Econo JIT Compilation

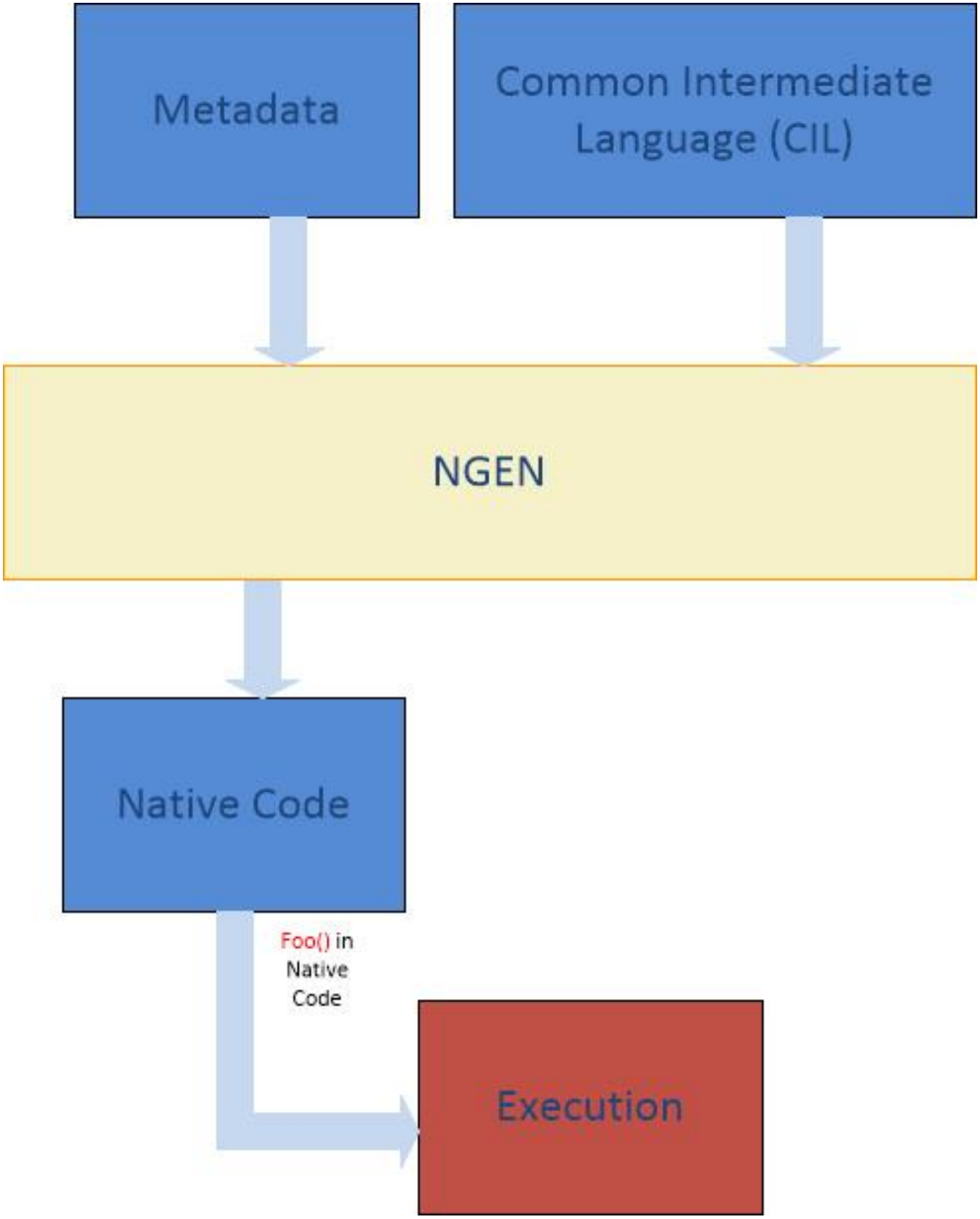
این نوع کامپایل شبیه به حالت Normal JIT است با این تفاوت که متدها بلافاصله بعد از اجرا از حافظه حذف می‌شوند.



Pre-JIT Compilation

یکی دیگر از حالت‌های کامپایل برنامه‌های دات نتی Pre-JIT Compilation می باشد. در این حالت به جای متدهای مورد استفاده،

کل اسمبلی کامپایل می‌شود. در دات نت می‌توان اینکار را توسط Ngen.exe یا (Native Image Generator) انجام داد. تمام دستورالعمل‌های CIL قبل از اجرا به کد محلی(Native Code) کامپایل می‌شوند. در این حالت runtime می‌تواند از native images به جای کامپایلر JIT استفاده کند. این نوع کامپایل عملیات تولید کد را در زمان اجرای برنامه به زمان Installation منتقل می‌کند، در اینصورت برنامه نیاز به یک Installer برای اینکار دارد.

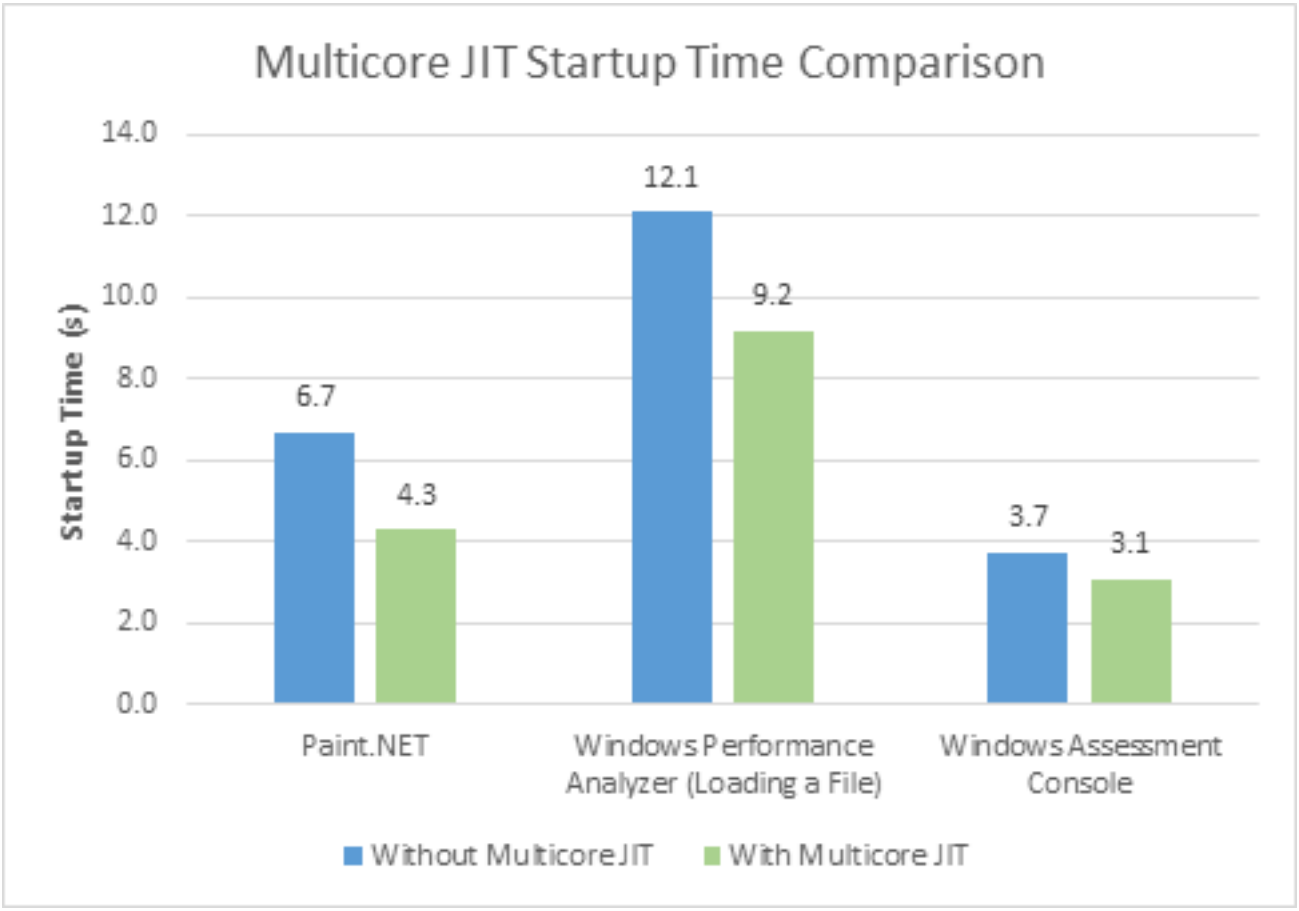


Multicore JIT

در دات نت فریم ورک 4.5 یک راه حل جایگزین دیگر برای بهینه سازی و بهبود سرعت اجرای برنامه‌های دات نت وجود دارد.

همانطور که عنوان شد Ngen.exe برای در دسترس بودن نیاز به Installer برای برنامه دارد. توسط Multicore JIT متدها بر روی دو هسته به صورت موازی کامپایل می‌شوند، در اینصورت می‌توانید تا 50 درصد از JIT Time صرفه جویی کنید.

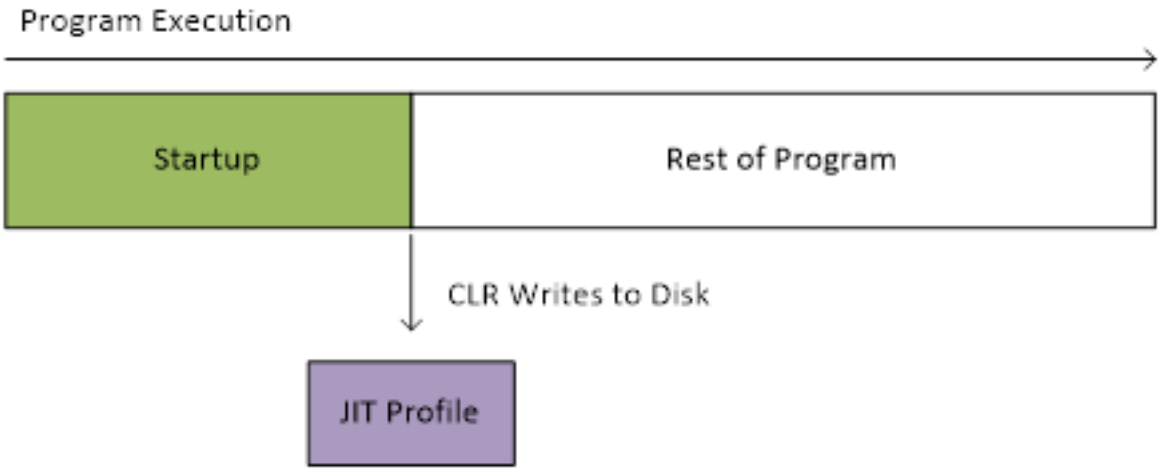
Multicore JIT همچنین می‌تواند باعث بهبود سرعت در برنامه‌های WPF شود. در نمودار زیر می‌توانید حالت‌های استفاده و عدم استفاده از Multicore JIT را در سه برنامه WPF نوشته شده مشاهده کنید.



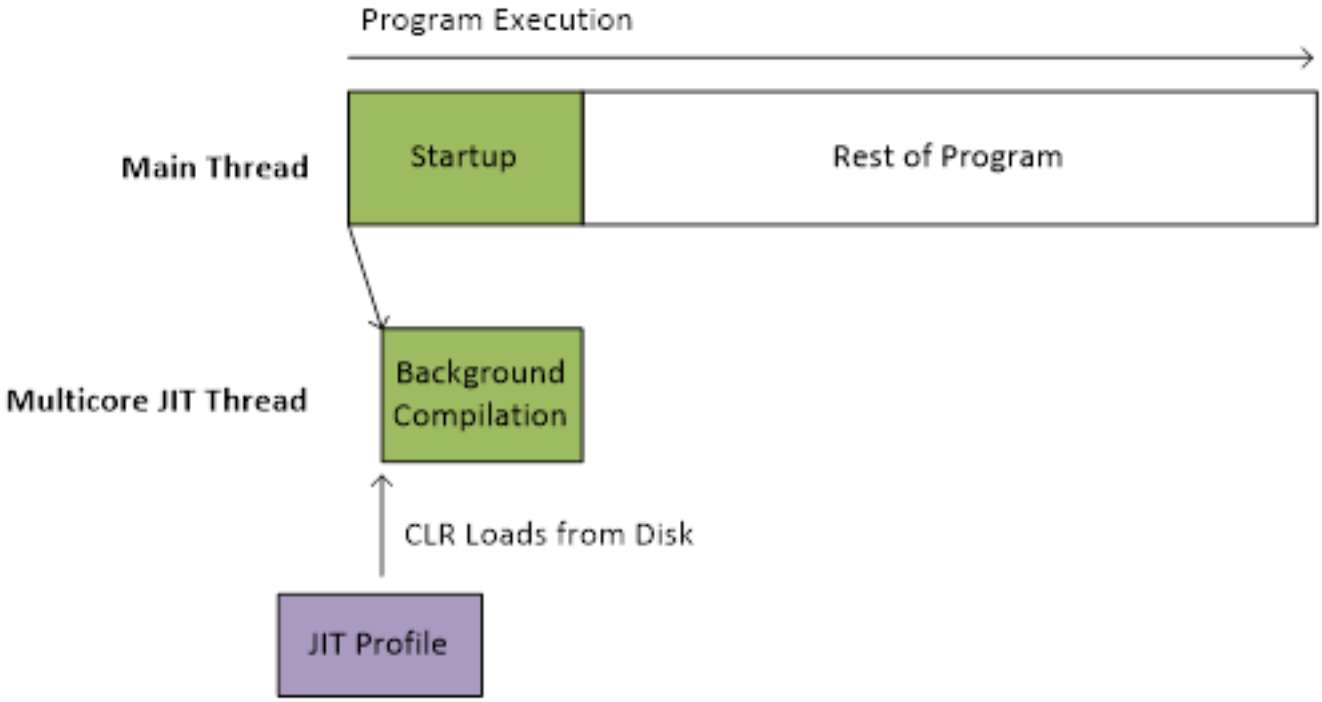
Multicore JIT در عمل

Multicore JIT از دو مد عملیاتی استفاده می‌کند: مد ثبت(Recording mode)، مد بازپخش(Playback mode)

در حالت ثبت کامپایلر JIT هر متدی که نیاز به کامپایل داشته باشد را رکورد می‌کند. بعد از اینکه CLR تعیین کند که اجرای برنامه به اتمام رسیده است، تمام متدهایی که اجرا شده اند را به صورت یک پروفایل بر روی دیسک ذخیره می‌کند.



هنگامیکه Multicore JIT فعال می‌شود، با اولین اجرای برنامه، حالت ثبت مورد استفاده قرار می‌گیرد. در اجراهای بعدی، از حالت بازپخش استفاده می‌شود. حالت بازپخش پروفایل را از طریق دیسک بارگیری کرده، و قبل از اینکه این اطلاعات توسط ترد اصلی مورد استفاده قرار گیرد، از آنها برای تفسیر (کامپایل) متدها در پیش‌زمینه استفاده می‌کند.



در نتیجه، ترد اصلی به کامپایل دیگری نیاز ندارد، در این حالت سرعت اجرای برنامه بیشتر می‌شود. حالت‌های ثبت و بازپخش تنها برای کامپیوترهایی با چندین هسته فعال می‌باشند.

استفاده از Multicore JIT

در برنامه‌های ASP.NET 4.5 و Silverlight 5 به صورت پیش فرض این ویژگی فعال می‌باشد. ازآنجائیکه این برنامه‌ها hosted application هستند؛ در نتیجه فضای مناسبی برای ذخیره سازی پروفایل در این نوع برنامه‌ها موجود می‌باشد. اما برای برنامه‌های

Desktop این ویژگی باید فعال شود. برای اینکار کافی است دو خط زیر را به نقطه شروع برنامه تان اضافه کنید:

```
public App()
{
    ProfileOptimization.SetProfileRoot(@"C:\MyAppFolder");
    ProfileOptimization.StartProfile("Startup.Profile");
}
```

توسط متد SetProfileRoot می‌توانیم مسیر ذخیره سازی پروفایل JIT را مشخص کنیم. در خط بعدی نیز توسط متد StartProfile نام پروفایل را برای فعال سازی Multicore JIT تعیین می‌کنیم. در این حالت در اولین اجرای برنامه پروفایلی وجود ندارد، Multicore JIT در حالت ثبت عمل می‌کند و پروفایل را در مسیر تعیین شده ایجاد می‌کند. در دومین بار اجرای برنامه CRL پروفایل را از اجرای قبلی برنامه بارگذاری می‌کند؛ در این حالت Multicore JIT به صورت بازپخش عمل می‌کند.

همانطور که عنوان شد در برنامه‌های ASP.NET 4.5 و Silverlight 5 قابلیت Multicore JIT به صورت پیش فرض فعال می‌باشد. برای غیر فعال سازی آن می‌توانید با تغییر فلگ profileGuidedOptimizations به None اینکار را انجام دهید:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <!-- ... -->
 <system.web>
 <compilation profileGuidedOptimizations="None" />
 <!-- ... -->
 </system.web>
</configuration>
```

عنوان: **کامپایل خودکار یک پروژه برای دو فریم ورک**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۵ ۱۸:۵
آدرس: www.dotnettips.info
گروه‌ها: MSBuild, .NET 4.5, Compile

فرض کنید می‌خواهید زمانیکه دکمه‌ی build در VS.NET فشرده شد، دو نسخه‌ی دات نت 4 و دات نت 4.5، از پروژه‌ی شما در پوشه‌های مجزایی کامپایل شده و قرار گیرند. در ادامه نحوه‌ی انجام این‌کار را بررسی خواهیم کرد.

پروژه نمونه

تنظیمات ذیل را بر روی یک پروژه از نوع class library دات نت 4 در VS 2013 اعمال خواهیم کرد.

ویرایش فایل پروژه برنامه

برای اینکه تنظیمات کامپایل خودکار مخصوص دات نت 4.5 را نیز به این پروژه دات نت 4 اضافه کنیم، نیاز است فایل csproj آن‌را مستقیما ویرایش نمائیم. این تغییرات شامل مراحل ذیل هستند:

الف) تعریف متغیر Framework

```
<PropertyGroup>
    <!-- ...-->
    <Framework Condition=" '$(Framework)' == '' ">NET40</Framework>
  </PropertyGroup>
```

به ابتدای فایل csproj در قسمت PropertyGroup آن یک متغیر جدید را به نام Framework اضافه کنید. از این متغیر در شرط‌های کامپایل استفاده خواهد شد.

ب) ویرایش مسیر خروجی تنظیمات کامپایل فعلی

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <!-- ...-->
    <OutputPath>bin\$(Configuration)\$(Framework)\</OutputPath>
  </PropertyGroup>
```

در حال حاضر حداقل تنظیمات کامپایل حالت debug، در فایل پروژه موجود است. مقدار OutputPath آن‌را به نحو فوق تغییر دهید تا خروجی نهایی را در پوشه‌ای مانند bin\Debug\NET40 ایجاد کند.
بدیهی است اگر حالت release هم وجود دارد، نیاز است مقدار OutputPath آن‌را نیز به همین ترتیب ویرایش کرد.

ج) افزودن تنظیمات کامپایل دات نت 4.5 به پروژه جاری

```
<PropertyGroup Condition=" '$(Framework)' == 'NET45' And '$(Configuration)|$(Platform)' ==
'Debug|AnyCPU'">
    <TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\$(Configuration)\$(Framework)\</OutputPath>
    <DefineConstants>DEBUG;TRACE;NET45</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Framework)' == 'NET45' And '$(Configuration)|$(Platform)' ==
'Release|AnyCPU' ">
    <TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
    <OutputPath>bin\$(Configuration)\$(Framework)\</OutputPath>
    <DefineConstants>TRACE;NET45</DefineConstants>
```

```
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
```

در اینجا تنظیمات حالت debug و release مخصوص دات نت 4.5 را مشاهده می‌کنید. برای نگارش‌های دیگر، تنها کافی است مقدار TargetFrameworkVersion را ویرایش کنید.
همچنین اگر به DefineConstants آن دقت کنید، مقدار NET45 نیز به آن اضافه شده‌است. این مورد سبب می‌شود که بتوانید در پروژه‌ی جاری، شرطی‌هایی را ایجاد کنید که کدهای آن فقط در حین کامپایل برای دات نت 4.5 به خروجی اسمبلی نهایی اضافه شوند:

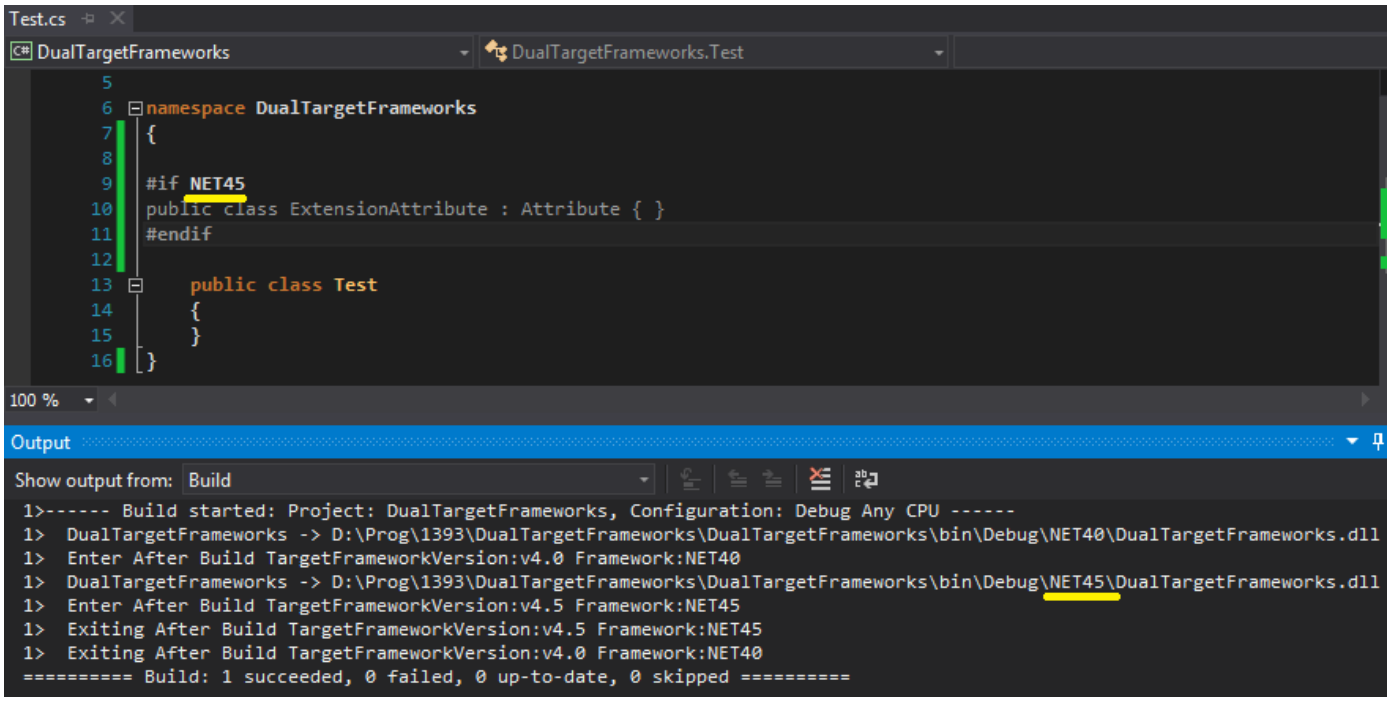
```
#if NET45
public class ExtensionAttribute : Attribute { }
#endif
```

د) افزودن تنظیمات پس از build

در انتهای فایل csproj قسمت AfterBuild به صورت کامنت شده موجود است. آن‌را به نحو ذیل تغییر دهید:

```
<Target Name="AfterBuild">
    <Message Text="Enter After Build TargetFrameworkVersion:$(TargetFrameworkVersion)
Framework:$(Framework)" Importance="high" />
    <MSBuild Condition=" '$(Framework)' != 'NET45'" Projects="$(MSBuildProjectFile)"
Properties="Framework=NET45" RunEachTargetSeparately="true" />
    <Message Text="Exiting After Build TargetFrameworkVersion:$(TargetFrameworkVersion)
Framework:$(Framework)" Importance="high" />
  </Target>
```

این تنظیم سبب می‌شود تا کامپایل مخصوص دات نت 4.5 نیز به صورت خودکار فعال گردد و خروجی آن در مسیر bin\Debug\NET45 به صورت جداگانه‌ای قرار گیرد.



برای آزمایش بیشتر، فایل csproj نهایی را از اینجا می‌توانید دریافت کنید:

DualTargetFrameworks.zip

نظرات خوانندگان

نویسنده: مهدی نقدی
تاریخ: ۱۳۹۳/۰۶/۳۰ ۲۰:۵۶

خیلی موضوع خوبی بود. واقعا خسته نباشید.
اگر از کتابخانه ای استفاده کنیم که DLL دات نت ۴ و ۴.۵ جداگانه ای رو ارائه داده باشه، چطور میشه این موضوع را پوشش داد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۳۰ ۲۲:۱

ویژگی‌های Condition ذکر شده در متن، در مورد ارجاعات هم قابل تنظیم است:

```
<Reference Include="MyAssembly"  Condition=".........">
  <SpecificVersion>False</SpecificVersion>
  <HintPath>path\to\MyAssembly.dll</HintPath>
</Reference>
```

همچنین در مورد فایل‌های سورس:

```
<Compile Include="Class20.cs" Condition=" ....." />
```

عنوان: **اضافه کردن Net 4.5. و بالاتر به پروژه ستاپ ویژوال استودیو 2010**
نویسنده: محبوبه محمدی
تاریخ: ۱۳۹۳/۰۷/۰۴ ۱۷:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Visual Studio, .NET 4.5, Setup

در این مطلب یک ترفند ساده و سریع برای دوستانی که می‌خواهند از ویژوال استودیو 2010 برای ساختن برنامه‌ی Setup پروژه‌های خود استفاده کنند، آورده می‌شود.
اگر برای ساخت برنامه‌های نصب خود بخواهید از ویژوال استودیو 2010 استفاده کنید و ورژن دات نت برنامه شما بالاتر از 4 باشد، متوجه خواهید شد که در قسمت prerequisites، ورژن دات نت مورد نظر شما وجود ندارد.
برای اضافه کردن net 4.5. و بالاتر به برنامه‌ی نصب خود باید یک Bootstrapper ایجاد کرده و به Bootstrapper Package های موجود در ویندوز اضافه نمایید. در اینجا نحوه ایجاد و استفاده از Bootstrapper توضیح داده شده است. اما برای اینکه نخواهید درگیر ساخت XML manifests برای Net 4.5. شوید، یک راه حل ساده‌تر وجود دارد و آ ن استفاده از Bootstrapper های ساخته شده هنگام نصب ورژن‌های بالاتر ویژوال استودیو که شامل ورژن‌های مورد نیاز از دات نت نیز هستند می‌باشد. برای این کار کافی است به مسیر زیر بر روی سیستم مراجعه نمایید:

```
C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\Bootstrapper\Packages
```

در مسیر فوق، فولدرهای DotNetFX45، DotNetFX451 و DotNetFX452 را مشاهده می‌کنید که شامل فایل‌های مورد نیاز برای اضافه کردن Bootstrapper به ویژوال استادیو 2010 است.
برای اینکار فولدر مربوطه را کپی نمایید و در مسیر زیر قرار دهید:

```
C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bootstrapper\Packages
```

فقط توجه داشته باشید که باید فایل نصب دات نت (مثلا برای دات نت 4.5 فایل dotNetFx45_Full_x86_x64.exe) را به آن اضافه نمایید .
حال اگر ویژوال استادیو 2010 را باز کنید و یک پروژه ستاپ ایجاد نمایید می‌بینید که ورژن مورد نظر به قسمت prerequisites اضافه شده است.

عنوان: .NET Just-In-Time Optimization
نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۱/۰۵ ۱۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Performance, JIT, .NET 4.5

هدف از توابع خطی(Inline)

استفاده از توابع، مقداری بر زمان اجرای برنامه می‌افزاید؛ هرچند که این زمان بسیار کم و در حد میلی ثانیه است، اما باری را بر روی برنامه قرار می‌دهد و علت این تاخیر زمانی این است که در فراخوانی و اعلان توابع، کامپایلر یک کپی از تابع مورد نظر را در حافظه قرار می‌دهد و در فراخوانی تابع، به آدرس مذکور مراجعه می‌کند و در عین حال آدرس موقعیت توقف دستورات در تابع main را نیز ذخیره می‌کند تا پس از پایان تابع، به آدرس قبل برگردد و ادامه‌ی دستورات را اجرا کند. در نتیجه این آدرس دهی‌ها و نقل و انتقالات بین آنها بار زمانی را در برنامه ایجاد می‌کند که در صورت زیاد بودن توابع در برنامه و تعداد فراخوانی‌های لازم، زمان قابل توجهی خواهد شد.

یکی از تکنیک‌های بهینه که برای کاهش زمان اجرای برنامه توسط کامپایلرها استفاده می‌شود استفاده از توابع خطی (inline) است. این امکان در زبان C با عنوان توابع ماکرو(Macro function) و در C++ با عنوان توابع خطی (inline function) وجود دارد.

در واقع توابع خطی به کامپایلر پیشنهاد می‌دهند، زمانی که سربار فراخوانی تابع بیشتر از سربار بدنه خود متد باشد، برای کاهش هزینه و زمان اجرای برنامه از تابع به صورت خطی استفاده کند و یک کپی از بنده‌ی تابع را در قسمتی که تابع ما فراخوانی شده است، قرار دهد که مورد آدرس دهی از میان خواهد رفت!

نمونه ای از پیاده سازی این تکنیک در زبان C++ :

```
inline type name(parameters)
{
    ...
}
```

بررسی متدهای خطی در سی شارپ به مثال زیر توجه کنید:

قسمت‌های getter و setter مربوط به پراپرتی‌ها سربار اضافی بر کلاس Vector می‌افزایند. این موضوع شاید آنچنان مسئله‌ی مهمی نباشد. ولی فرض کنید این پراپرتی‌ها به شکل زیر داخل حلقه‌ای طولانی قرار گیرند. اگر با استفاده از یک پروفایلر زمان اجرای برنامه را زیر نظر بگیرید، خواهید دید که بیش از 90 درصد آن صرف فراخوانی‌های متدهای بخش‌های get , set پراپرتی‌ها است. برای این منظور باید مطمئن شویم که فراخوانی این متدها، به صورت خطی صورت می‌گیرد!

```
public class Vector
  {
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }

    // ...
  }
```

برای این منظور آزمایشی را انجام می‌دهیم. فرض کنید کلاسی را به شکل زیر داشته باشیم:

```
public class MyClass
  {
    public int A { get; set; }
    public int C;
  }
```

و برای استفاده از آن به شکل زیر عمل کنیم:

```
static void Main()
{
    MyClass target = new MyClass();
    int a = target.A;
    Console.WriteLine("A = {0}", a);
    int c = target.C;
```

```
    Console.WriteLine("C = {0}", c);
  }
```

بعد از دیباگ برنامه و مشاهده‌ی کدهای ماشین مربوط به آن خواهیم دید که متد مربوط به getter پراپرتی A به صورت خطی فراخوانی نشده است:

```
int a = target.A;
  0000003e  mov         ecx,edi
  00000040  cmp         dword ptr [ecx],ecx
  00000042  call        dword ptr ds:[05FA29A8h]
  00000048  mov         esi,eax
  0000004a  mov         dword ptr [esp+4],esi
             int c = target.C;
  00000098  mov         edi,dword ptr [edi+4]
    MyClass.get_A() looks like this:
  00000000  push        esi
  00000001  mov         esi,ecx
  00000003  cmp         dword ptr ds:[03B701DCh],0
  0000000a  je          00000011
  0000000c  call        76BA6BA7
  00000011  mov         eax,dword ptr [esi+0Ch]
  00000014  pop         esi
  00000015  ret
```

چه اتفاقی افتاده است؟

کامپایلر سی شارپ در زمان کامپایل، کدهای برنامه را به کدهای IL تبدیل می‌کند و JITکامپایلر، این کدهای IL را گرفته و کد ساده‌ی ماشین را تولید می‌کند. لذا به دلیل اینکه JIT با معماری پردازنده آشنایی کافی دارد، مسئولیت تصمیم گیری اینکه کدام متد به صورت خطی فراخوانی شود برعهده‌ی آن است. در واقع این JIT است که تشخیص می‌دهد که آیا فراخونی متد به صورت خطی مناسب است یا نه و به صورت یک معاوضه کار بین خط لوله دستورالعمل‌ها و کش است.

اگر شما برنامه‌ی خود را با (F5) و همگام با دیباگ اجرا کنید، تمام بهینه سازی‌های JIT که Inline Method هم یکی از آنهاست، از کار خواهند افتاد. برای مشاهده‌ی کد بهینه شده باید با بدون دیباگ (CTRL+F5) برنامه خود را اجرا کنید که در آن صورت مشاهده خواهید کرد، متد getter مربوط به پراپرتی A به صورت خطی استفاده شده است.

```
int a = target.A;
00000024  mov         ebx,dword ptr [edi+0Ch]
```

JIT محدودیت هایی برای فراخونی به صورت خطی متدها دارد :

متد هایی که حجم کد IL آنها بیشتر از 32 بایت است.

متدهای بازگشتی.

متدهایی که با اتریبیوت MethodImpl علامتگذاری شدند و MethodImplOptions.NoInlining اعمال شده بر آن

متدهای virtual

متدهایی که دارای کد مدیریت خطا هستند

Methods that take a large value type as a parameter

Methods with complicated flowgraphs

برای اینکه در سی شارپ به کامپایلر اعلام کنیم تا متد مورد نظر به صورت خطی مورد استفاده قرار گیرد، در دات نت 4.5 توسط اتریبیوت MethodImpl و اعمال MethodImplOptions.AggressiveInlining که یک نوع شمارشی است می‌توان این کار را انجام داد. مثال:

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
class Program
{
    const int _max = 10000000;
    static void Main()
```

```
    {
         // ... Compile the methods.
        Method1();
        Method2();
        int sum = 0;
        var s1 = Stopwatch.StartNew();
        for (int i = 0; i < _max; i++)
           {
              sum += Method1();
           }
         s1.Stop();
         var s2 = Stopwatch.StartNew();
         for (int i = 0; i < _max; i++)
             {
                sum += Method2();
             }
         s2.Stop();
         Console.WriteLine(((double)(s1.Elapsed.TotalMilliseconds * 1000000) /
        _max).ToString("0.00 ns"));
        Console.WriteLine(((double)(s2.Elapsed.TotalMilliseconds * 1000000) /
      _max).ToString("0.00 ns"));
        Console.Read();
      }
      static int Method1()
      {
         // ... No inlining suggestion.
         return "one".Length + "two".Length + "three".Length +
              "four".Length + "five".Length + "six".Length +
              "seven".Length + "eight".Length + "nine".Length +
               "ten".Length;
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static int Method2()
    {
        // ... Aggressive inlining.
        return "one".Length + "two".Length + "three".Length +
           "four".Length + "five".Length + "six".Length +
           "seven".Length + "eight".Length + "nine".Length +
            "ten".Length;
    }
}
Output
7.34 ns    No options
0.32 ns    MethodImplOptions.AggressiveInlining
```

در واقع با اعمال این اتریبیوت، محدودیت شماره یک مبنی بر محدودیت حجم کد IL مربوط به متد، در نظر گرفته نخواهد شد .

مطالعه بیشتر: http://dotnet.dzone.com/news/aggressive-inlining-clr-45-jit

/http://www.ademiller.com/blogs/tech/2008/08/c-inline-methods-and-optimization

http://www.dotnetperls.com/aggressiveinlining

http://blogs.msdn.com/b/ericgu/archive/2004/01/29/64644.aspx

https://msdn.microsoft.com/en-us/library/ms973858.aspx#highperfmanagedapps_topic10

عنوان: **INPC استاندارد با بهره گیری از صفت CallerMemberName**
نویسنده: محسن افشین
تاریخ: ۱۳۹۴/۰۴/۱۷ ۱۵:۴۰
آدرس: www.dotnettips.info
گروه‌ها: WPF, .NET 4.5, C# 5.0

یکی از Attributeهای بسیار کاربردی که در سی شارپ 5 اضافه شد CallerMemberNameAttribute بود. این صفت به یک متد اجازه میدهد که از فراخواننده‌ی خود مطلع شود. این صفت را می‌توان بر روی یک پارامتر انتخابی که مقدار پیش‌فرضی دارد اعمال نمود.

استفاده از این صفت هم بسیار ساده است:

```
private void A ( [CallerMemberName] string callerName = "")
{
  Console.WriteLine("Caller is " + callerName);
}

private static void B()
{
        // let's call A
        A();
}
```

در کد فوق، متد A به راحتی می‌تواند بفهمد چه کسی آن را فراخوانی کرده است. از جمله کاربردهای این صفت در ردیابی و خطایابی است.

ولی یک استفاده‌ی بسیار کاربردی از این صفت، در پیاده سازی رابط INotifyPropertyChanged می‌باشد.

معمولا هنگام پیاده سازی INotifyPropertyChanged کدی شبیه به این را می‌نویسیم:

```
public class PersonViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }

        private string name;
        public string Name
        {
            get { return name; }
            set
            {
                this.name = value;
                OnPropertyChanged("Name");
            }
        }
    }
```

یعنی در Setter معمولا نام ویژگی ای را که تغییر کرده است، به متد OnPropertyChanged می‌فرستیم تا اطلاع رسانی‌های لازم انجام پذیرد. تا اینجای کار همه چیز خوب و آرام است. اما به محضی که کد شما کمی طولانی شود و شما به دلایلی نیاز به Refactor کردن کد و احیانا تغییر نام ویژگی‌ها را پیدا کنید، آن موقع مسائل جدیدی بروز پیدا می‌کند.

برای مثال فرض کنید پس از نوشتن کلاس PersonViewModel تصمیم می‌گیرد نام ویژگی Name را به FirstName تغییر دهید؛ چرا که می‌خواهید اجزای نام یک شخص را به صورت مجزا نگهداری و پردازش کنید. پس احتمالا با زدن کلید F2 روی فیلد name آن را به

firstName و ویژگی Name را به FirstName تغییر نام می‌دهید. همانند کد زیر:

```
private string firstName;
public string FirstName
{
            get { return firstName; }
            set
            {
                this.firstName = value;
                OnPropertyChanged("Name");
            }
}
```

برنامه را کامپایل کرده و در کمال تعجب می‌بینید که بخشی از برنامه درست رفتار نمی‌کند و تغییراتی که در نام کوچک شخص توسط کاربر ایجاد می‌شود به درستی بروزرسانی نمی‌شوند. علت ساده است: ما کد را به صورت اتوماتیک Refactor کرده ایم و گزینه‌ی Include String را در حین Refactor، در حالت پیشفرض غیرفعال رها کرده‌ایم. پس جای تعجبی ندارد که در هر جای کد که رشته‌ای به نام "Name" با ماهیت نام شخص داشته ایم، دست نخورده باقی مانده است. در واقع در کد تغییر یافته، هنگام تغییر FirstName، ما به سیستم گزارش می‌کنیم که ویژگی Name (که اصلا وجود ندارد) تغییر یافته است و این یعنی خطا.

حال احتمال بروز این خطا را در ViewModel هایی با ده‌ها ویژگی و ترکیب‌های مختلف در نظر بگیرید. پس کاملا محتمل است و برای خیلی از دوستان این اتفاق رخ داده است.

و اما راه حل چیست؟ به کارگیری صفت CallerMemberName

بهتر است که یک کلاس انتزاعی برای تمام ViewModelهای خود داشته باشیم و پیاده سازی جدید INPC را در درون آن قرار دهیم تا براحتی VMهای ما از آن مشتق شوند:

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string propertyName = "")
        {
            OnPropertyChangedExplicit(propertyName);
        }

        protected void OnPropertyChanged<TProperty>(Expression<Func<TProperty>> projection)
        {
            var memberExpression = (MemberExpression)projection.Body;
            OnPropertyChangedExplicit(memberExpression.Member.Name);
        }

        void OnPropertyChangedExplicit(string propertyName)
        {
            this.CheckPropertyName(propertyName);

            PropertyChangedEventHandler handler = this.PropertyChanged;

            if (handler != null)
            {
                var e = new PropertyChangedEventArgs(propertyName);
                handler(this, e);
            }
        }

        #region Check property name

        [Conditional("DEBUG")]
        [DebuggerStepThrough]
        public void CheckPropertyName(string propertyName)
        {
            if (TypeDescriptor.GetProperties(this)[propertyName] == null)
                throw new Exception(String.Format("Could not find property \"{0}\"", propertyName));
        }
```

```
        #endregion // Check property name
}
```

در این کلاس، ما پارامتر propertyName را از متد OnPropertyChanged، توسط صفت CallerMemberName حاشیه نویسی کرده‌ایم. این کار باعث می‌شود در Setterهای ویژگی‌ها، به راحتی بدون نوشتن نام ویژگی، عملیات اطلاع رسانی تغییرات را انجام دهیم. بدین صورت که کافیست متد OnPropertyChanged بدون هیچ آرگومانی در Setter فراخوانی شود و صفت CallerMemberName به صورت اتوماتیک نام ویژگی ای که فراخوانی از درون آن انجام شده است را درون پارامتر propertyName قرار می‌دهد.

پس کلاس PersonViewModel را به صورت زیر می‌توانیم اصلاح و تکمیل کنیم:

```
public class PersonViewModel : ViewModelBase
{
        private string firstName;
        public string FirstName
        {
            get { return firstName; }
            set
            {
                this.firstName = value;

                OnPropertyChanged();
                OnPropertyChanged(() => this.FullName);
            }
        }

        private string lastName;
        public string LastName
        {
            get { return lastName; }
            set
            {
                this.lastName = value;

                OnPropertyChanged();
                OnPropertyChanged(() => this.FullName);
            }
        }

        public string FullName
        {
            get { return string.Format("{0} {1}", FirstName, LastName); }
        }
}
```

همانطور که می‌بینید متد OnPropertyChanged بدون آرگومان فراخوانی میشود. اکنون اگر شما اقدام به Refactor کردن کد خود بکنید دیگر نگرانی از بابت تغییر نکردن رشته‌ها و کامنت‌ها نخواهید داشت و مطمئن هستید، نام ویژگی هر چیزی که باشد، به صورت خودکار به متد ارسال خواهد شد.

کلاس ViewModelBase یک پیاده سازی دیگر از OnPropetyChanged هم دارد که به شما اجازه می‌دهد با استفاده دستورات لامبدا، OnPropertyChanged را برای هر یک از اعضای دلخواه کلاس نیز فراخوانی کنید. همانطور که در مثال فوق می‌بینید، تغییرات نام خانوادگی در نام کامل شخص نیز اثرگذار است. در نتیجه به وسیله‌ی یک Func به راحتی بیان می‌کنیم که FullName هم تغییر کرده است و اطلاع رسانی برای آن نیز باید صورت پذیرد.

برای استفاده از صفت CallerMemberName باید دات نت هدف خود را 4.5 یا 4.6 قرار دهید.

ارجاع:

Raise INPC witout string name

نظرات خوانندگان

نویسنده: بهزاد
تاریخ: ۱۳۹۴/۰۴/۱۸ ۱۰:۲

با ایجاد Exception و خواندن StackTrace در نسخه‌های پایین‌تر هم میتوان به نام تابع فراخواننده پی برد