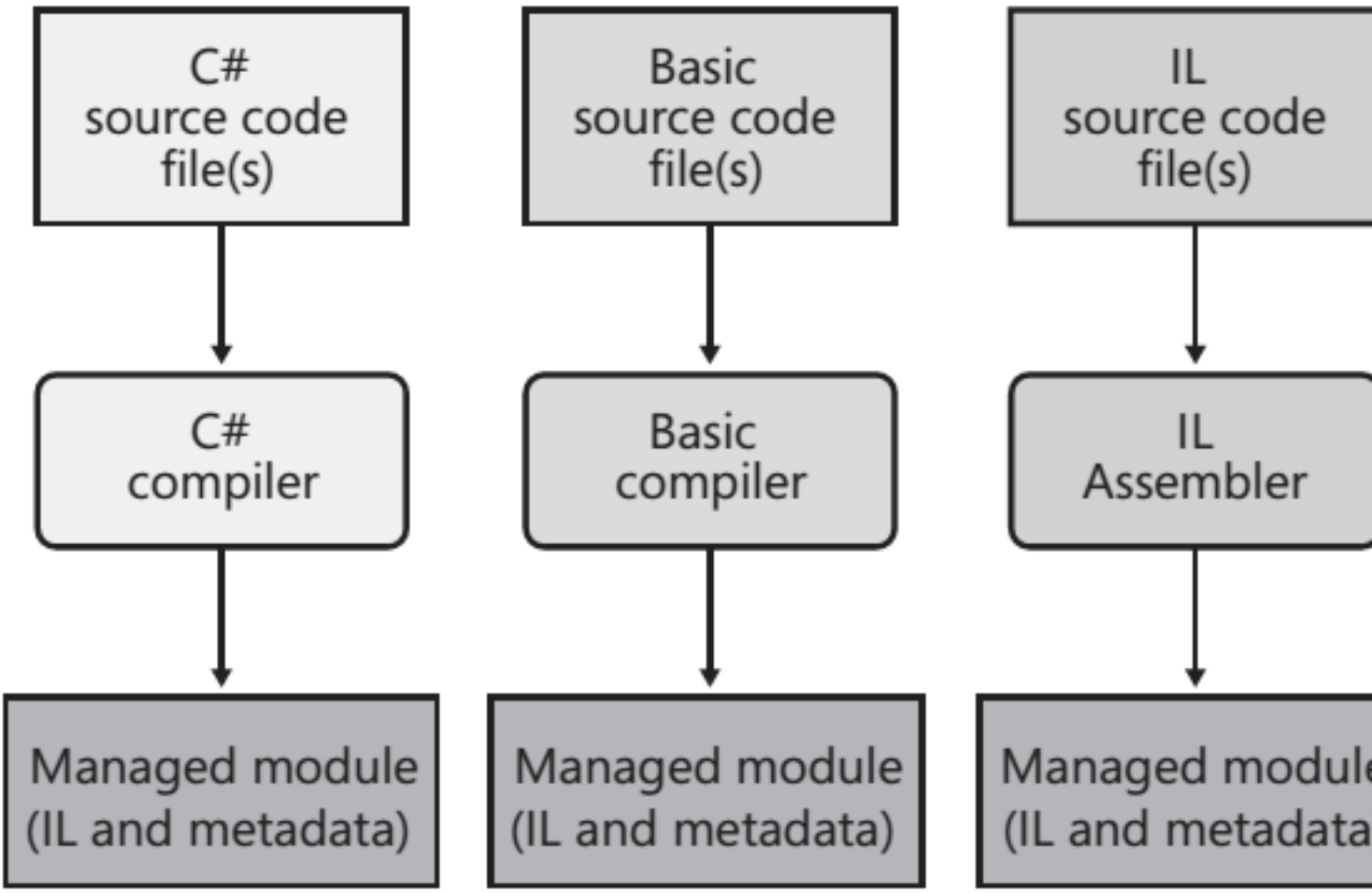
عنوان: **همه چیز در مورد CLR : قسمت اول**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۲ ۱۵:۴۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در حال حاضر من کتاب CLR Via Csharp ویرایش چهارم نوشته آقای جفری ریچر را مطالعه می‌کنم و نه قسمت از این مقالات، از بخش اول فصل اول آن به پایان رسیده که همگی آن‌ها را تا 9 روز آینده منتشر خواهم کرد. البته سعی شده که مقالات ترجمه صرف نباشند و منابع دیگری هم در کنار آن استفاده شده است. بعضی موارد را هم لینک کرده‌ام. تمام سعی خود را می‌کنم تا ادامه کتاب هم به مرور به طور مرتب ترجمه شود؛ تا شاید نسخه‌ی تقریبا کاملی از این کتاب را به زبان فارسی در اختیار داشته باشیم.
بعد از اینکه برنامه را تحلیل کردید و نیازمندی‌های یک برنامه را شناسایی کردید، وقت آن است که زبان برنامه نویسی خود را انتخاب کنید. هر زبان ویژگی‌های خاص و منحصر به فرد خود را دارد و این ممکن هست انتخاب شما را سخت کند. برای مثال شما در زبان‌های C/C++ unmanaged، کنترل بسیار زیادی روی امور سیستمی از قبیل حافظه و تردها دارید و به هر روشی که می‌خواهید می‌توانید آن‌ها را پیکربندی کنید. در زبان‌هایی چون Visual basic قدیم و مشابه‌های آن عموما اینگونه بود که طراحی یک اپلیکیشن از رابط کاربری گرفته تا اتصال به دیتابیس و اشیاء COM در آن ساده باشد؛ **ولی در زبان‌های CLR چطور؟**

در زبان‌های CLR شما دیگر وقت خود را به موضوعاتی چون مدیریت حافظه، هماهنگ سازی تردها و مباحث امنیتی و صدور استثناء در سطوح پایین‌تر نمی‌دهید و فرقی هم نمی‌کند که از چه زبانی استفاده می‌کنید. بلکه CLR هست که این امور را انجام می‌دهد و این مورد بین تمامی زبان‌های CLR مشترک است. برای مثال کاربری که قرار است در زمان اجرا استثناءها را صادر کند، در واقع مهم نیست که از چه زبانی برای آن استفاده می‌کند. بلکه آن CLR است که مدیریت آن را به عهده دارد و روال کار CLR برای همه زبان‌ها یکی است. پس این سوال پیش می‌آید که وقتی مبنا و زیر پایه‌ی همه زبان‌های CLR یکی است، چرا تعدد زبان دیده می‌شود و مزیت هر کدام بر دیگری چیست؟ اولین مورد syntax آن است. هر کاربر رو به چه زبانی کشیده می‌شود و شاید تجربه‌ی سابق در قدیم با یک برنامه‌ی مشابه بوده است که همچنان همان رویه سابق را ادامه می‌دهد و یا اینکه نحوه‌ی تحلیل و آنالیز کردن کدهای آن زبان است که کاربر را به سمت خود جذب کرده است. گاهی اوقات بعضی از زبان‌ها با تمرکز در انجام بعضی از کارها چون امور مالی یا ریاضیات، موارد فنی و ... باعث جذب کاربران آن گروه کاری به سمت خود می‌شوند. البته بعدا در آینده متوجه می‌شویم که بسیاری از زبان‌ها مثل سی شارپ و ویژوال بیسیک هر کدام قسمتی از امکانات CLR را پوشش می‌دهند نه تمام آن را.

زبان‌های CLR چگونه کار می‌کنند؟

در اولین گام بعد از نوشتن برنامه، کامپایلر آن زبان دست به کار شده و برنامه را برای شما کامپایل می‌کند. ولی اگر تصور می‌کنید که برنامه را به کد ماشین تبدیل می‌کند و از آن یک فایل اجرایی می‌سازد، سخت در اشتباه هستید. کامپایلر هر زبان CLR، کدها را به یک زبان میانی Intermediate Language به اختصار IL تبدیل می‌کند. فرقی نمی‌کند چه زبانی کار کرده‌اید، کد شما تبدیل شده است به یک زبان میانی مشترک. CLR نمی‌تواند برای تک تک زبان‌های شما یک مفسر داشته باشد. در واقع هر کمپایلر قواعد زبان خود را شناخته و آن را به یک زبان مشترک تبدیل می‌سازد و حالا CLR می‌تواند حرف تمامی زبان‌ها را بفهمد. به فایل ساخته شده managed module گویند و به زبان‌هایی که از این قواعد پیروی نمی‌کنند unmanaged گفته می‌شود؛ مثل زبان سی ++ که در دات نت هم managed و هم unmanaged داریم که اولی بدون فریم ورک دات نت کار می‌کند و مستقیما به کد ماشین تبدیل می‌شود و دومی نیاز به فریم ورک دات نت داشته و به زبان میانی کامپایل می‌شود. جدول زیر نشان می‌دهد که کد همه‌ی زبان‌ها تبدیل به یک نوع شده است.



فایل هایی که ساخته می‌شوند بر دو نوع هستند؛ یا بر اساس استاندارد windows Portable Executable 32bits برای سیستم‌های 32 بیتی و 64 بیتی هستند و یا بر اساس windows Portable Executable 64bits مختص سیستم‌های 64 بیتی هستند که به ترتیب PE32 و +PE32 نامیده می‌شوند که CLR بر اساس این اطلاعات آن‌ها را به کد اجرایی تبدیل می‌کند. زبان‌های CLR همیشه این مزیت را داشته‌اند که اصول امنیتی چون Data Execution Prevention یا DEP و همچنین ASLR یا Address Space Layout Randomization در آن‌ها لحاظ شده باشد.

نظرات خوانندگان

نویسنده: فلونی
تاریخ: ۱۳۹۴/۰۵/۳۱ ۱۲:۳۰

زبان‌های CLR همیشه این مزیت را داشته‌اند که اصول امنیتی چون DEP یا Data Execution Prevention و همچنین ASLR یا Address Space Layout Randomization در آن‌ها لحاظ شده باشد.

DEP و ASLR مکانیزم‌های امنیتی سیستم عامل‌ها هستند و ربطی به CLR و زبان برنامه نویسی ندارند .

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۳۱ ۱۲:۵۸

- به صورت خیلی خلاصه، کار DEP غیر میسر کردن اجرای داده‌ها به صورت کد است (مانند اجرای کدهای مخرب از طریق سر ریز بافر) و کار ASLR هم غیرقابل پیش بینی کردن محل قرارگرفتن بیت‌ها و داده‌های برنامه در حافظه‌است.

- ربطی به زبان برنامه نویسی ندارند؛ درست است. اما **R** CL یعنی **Runtime** Common Language . این محیط اجرایی و JIT آن ASLR را میسر می‌کنند . حتی اسمبلی‌های ngen شده هم از دات نت 3.5 به بعد دارای ASLR فعال هستند. همچنین DEP هم از طریق روشن کردن سوئیچ NXCOMPAT کامپایلر فراهم شده‌است (از زمان دات نت 2 به بعد). یعنی اگر OpenSSL را با دات نت می‌نوشتند، هیچ وقت مشکل heartbleed رخ نمی‌داد.

عنوان: **آشنایی با CLR: قسمت دوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۳ ۰:۵۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

متادیتاهای یک ماژول مدیریت شده Managed Module

در قسمت قبلی به اصل وجودی CLR پرداختیم. در این قسمت تا حدودی به بررسی ماژول مدیریت شده managed module که از زبان‌های دیگر، کامپایل شده و به زبان میانی تبدیل گشته است صحبت می‌کنیم.

یک ماژول مدیریت شده شامل بخش‌های زیر است:

نام بخش	توضیح
هدر PE32 یا PE32+	CLR باید بداند که برنامه‌ی نوشته شده قرار است روی چه پلتفرمی و با چه معماری، اجرا گردد. این برنامه یک برنامه‌ی 32 بیتی است یا 64 بیتی. همچنین این هدر اشاره می‌کند که نوع فایل از چه نوعی است؛ GUI,CUI یا DLL. به علاوه تاریخ ایجاد یا کامپایل فایل هم در آن ذکر شده است. در صورتیکه این فایل شامل کدهای بومی native CPU هم باشد، اطلاعاتی در مورد این نوع کدها نیز در این هدر ذکر می‌شود و اگر ماژول ارائه شده تنها شامل کد IL باشد، قسمت بزرگی از اطلاعات این هدر در نظر گرفته نمی‌شود.
CLR Header	اطلاعاتی را در مورد CLR ارائه می‌کند. اینکه برای اجرا به چه ورژنی از CLR نیاز دارد. منابع مورد استفاده. آدرس و اندازه جداول و فایل‌های متادیتا و جزئیات دیگر.
metadata	هر کد یا ماژول مدیریت شده‌ای، شامل جداول متادیتا است که این جداول بر دو نوع هستند. اول جداولی که نوع‌ها و اعضای تعریف شده در کد را توصیف می‌کنند و دومی جداولی که نوع‌ها و اعضایی را که در کد به آن ارجاع شده است، توصیف می‌کنند.
IL Code	اینجا محل قرار گیری کدهای میانی تبدیل شده است که در زمان اجرا، CLR آن‌ها را به کدهای بومی تبدیل می‌کند.

کامپایلرهایی که بر اساس CLR کار می‌کنند، وظیفه دارند جداول متادیتاها را به طور کامل ساخته و داخل فایل نهایی embed کنند. متادیتاها مجموعه‌ی کاملی از فناوری‌های قدیمی چون فایل‌های COM یا Component Object Model و همچنین IDL یا Interface Definition (Description) Language هستند. گفتیم که متادیتاها همیشه داخل فایل IL که ممکن است DLL باشد یا EXE، ترکیب یا Embed شده‌اند و جدایی آن‌ها غیر ممکن است. در واقع کامپایلر در یک زمان، هم کد IL و هم متادیتاها را تولید کرده و آن‌ها را به صورت یک نتیجه‌ی واحد در می‌آورد.

متادیتاها استفاده‌های زیادی دارند که در زیر به تعدادی از آنان اشاره می‌کنیم:
موقع کامپایل نیاز به هدرهای C و C++ از بین می‌رود؛ چرا که فایل نهایی شامل تمامی اطلاعات ارجاع شده می‌باشد. کامپایلرها می‌توانند مستقیما اطلاعات را از داخل متادیتاها بخوانند.
ویژوال استودیو از آن‌ها برای کدنویسی راحت‌تر بهره می‌گیرد. با استفاده از قابلیت IntelliSense، متادیتاها به شما خواهند گفت چه متدهایی، چه پراپرتی‌هایی، چه رویدادهایی و ... در دسترس شماست و هر متد انتظار چه پارامترهایی را از شما دارد.
CLR Code Verification از متادیتا برای اینکه اطمینان کسب کند که کدها تنها عملیات type Safe را انجام می‌دهند، استفاده می‌کند.

متادیتاها به فیلد یک شیء اجازه می‌دهند که خود را به داخل بلوک‌های حافظ انتقال داده و بعد از ارسال به یک ماشین دیگر، همان شیء را با همان وضعیت، ایجاد نماید.
متادیتاها به GC اجازه می‌دهند که طول عمر یک شیء را رصد کند. GC برای هر شیء موجود می‌تواند نوع هر شیء را تشخیص داده و از طریق متادیتاها می‌تواند تشخیص دهد که فیلدهای یک شیء به اشیاء دیگری هم متصل هستند.

در آینده بیشتر در مورد متادیتاها صحبت خواهیم کرد.

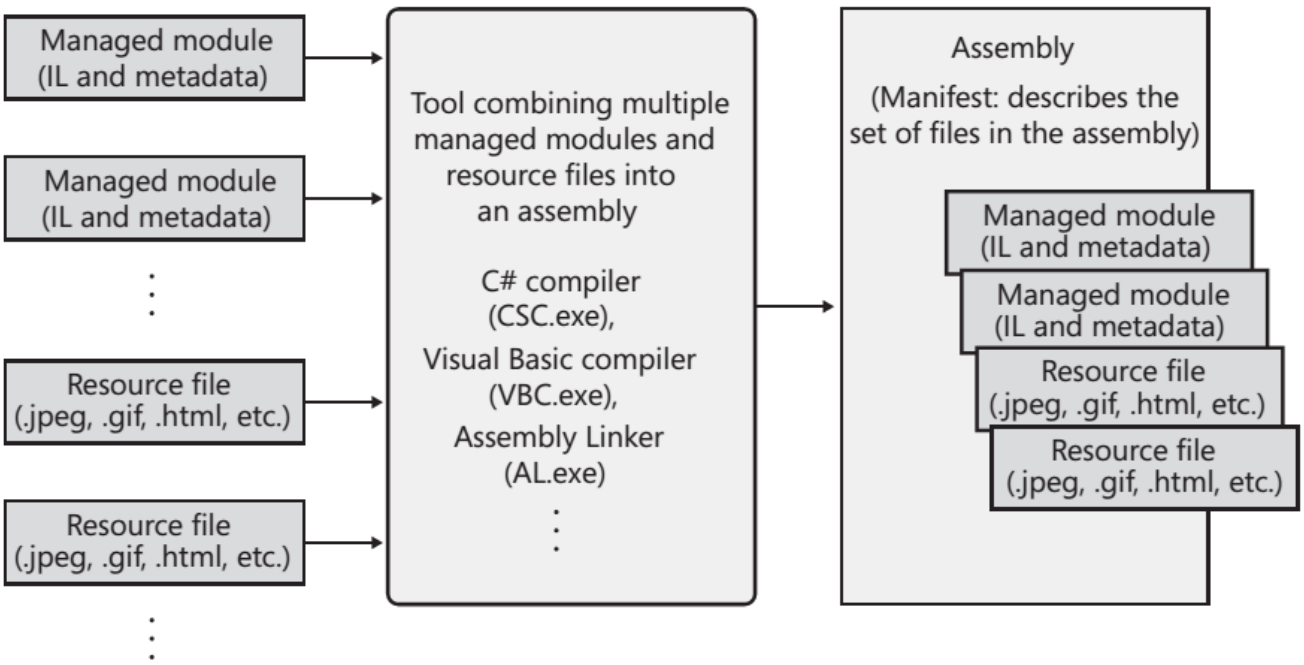
عنوان: **آشنایی با CLR: قسمت سوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۴ ۱:۱۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در اینجا ما زیاد بر روی جزئیات یک اسمبلی مانور نمی‌دهیم و آن را به آینده موکول می‌کنیم و فقط مقداری از مباحث اصلی را ذکر می‌کنیم.

ترکیب ماژول‌های مدیریت شده به یک اسمبلی

اگر حقیقت را بخواهید CLR نمی‌تواند با ماژول‌ها کار کند، بلکه با اسمبلی‌ها کار می‌کند. اسمبلی یک مفهوم انتزاعی است که به سختی میتوان برای بار اول آن را درک کرد.
اول از همه: اسمبلی یک گروه منطقی از یک یا چند ماژول یا فایل‌های ریسورس (منبع) است.
دوم: اسمبلی کوچکترین واحد استفاده مجدد، امنیت و نسخه بندی است.
بر اساس انتخابی که شما در استفاده از کامپایلرها و ابزارها کرده‌اید، نسخه‌ی نهایی شامل یک یا چند فایل اسمبلی خواهد شد. در دنیای CLR ما یک اسمبلی را کامپوننت صدا می‌زنیم.

شکل زیر در مورد اسمبلی‌ها توضیح می‌دهد. آنچه که شکل زیر توضیح می‌دهد تعدادی از ماژول‌های مدیریت شده به همراه فایل‌های منابع یا دیتا توسط ابزارهایی که مورد پردازش قرار گرفته‌اند به فایل‌های 32 یا 64 بیتی تبدیل شده‌اند که داخل یک گروه بندی منطقی از فایل‌ها قرار گرفته‌اند. آنچه که اتفاق می‌افتد این هست که این فایل‌های 32 یا 64 بیتی شامل بلوکی از داده‌هایی است که با نام manifest شناخته می‌شوند. manifest یک مجموعه دیگر از جداول متادیتاها است. این جداول به توصیف فایل‌های تشکیل دهنده اسمبلی می‌پردازد.



همه کارهای تولید اسمبلی به صورت خودکار اتفاق می‌افتد. ولی در صورتیکه قصد دارید فایلی را به اسمبلی به طور دستی اضافه کنید نیاز است که به دستورات و ابزارهای کامپایلر آشنایی داشته باشید.

یک اسمبلی به شما اجازه می‌دهد تا مفاهیم فیزیکی و منطقی کامپوننت را از هم جدا سازید. اینکه چگونه کد و منابع خود را از یکدیگر جدا کنید به خود شما بر می‌گردد. برای مثال اگر قصد دارید منابع یا نوع داده‌ای را که به ندرت مورد استفاده قرار می‌گیرد، در یک فایل جدا از اسمبلی نگهداری کنید، این فایل جدا میتواند بر اساس تقاضای کاربر در زمان اجرای برنامه از اینترنت دریافت شود. حال اگر همین فایل هیچگاه استفاده نشود، در زمان نصب برنامه و مقدار حافظه دیسک سخت صرفه جویی خواهد شد. اسمبلی‌ها به شما اجازه می‌دهند که فایل‌های توزیع برنامه را به چندین قسمت بشکنید، در حالی که همه‌ی آن‌ها متعلق به یک مجموعه هستند.

یک ماژول اسمبلی شامل اطلاعاتی در رابطه با ارجاعاتش است؛ به علاوه ورژن خود اسمبلی. این اطلاعات سبب می‌شوند که یک اسمبلی خود تعریف self-describing شود که به بیان ساده‌تر باعث می‌شود CLR وابستگی‌های یک اسمبلی را تشخیص داده تا ترتیب اجرای آن‌ها را پیدا کند. نه دیگر نیازی به اطلاعات اضافی در ریجستری است و نه در Active Directory Domain Service یا به اختصار ADDS.

از آنجایی که هیچ اطلاعاتی اضافی نیست، توزیع ماژول‌های مدیریت شده راحت‌تر از ماژول‌های مدیریت نشده است.

مطلب مشابهی نیز در وبلاگ آقای شهروز جعفری برای توصیف اسمبلی‌ها وجود دارد که خیلی خوب هست به قسمت مطالب مرتبط آن هم نگاهی داشته باشید.

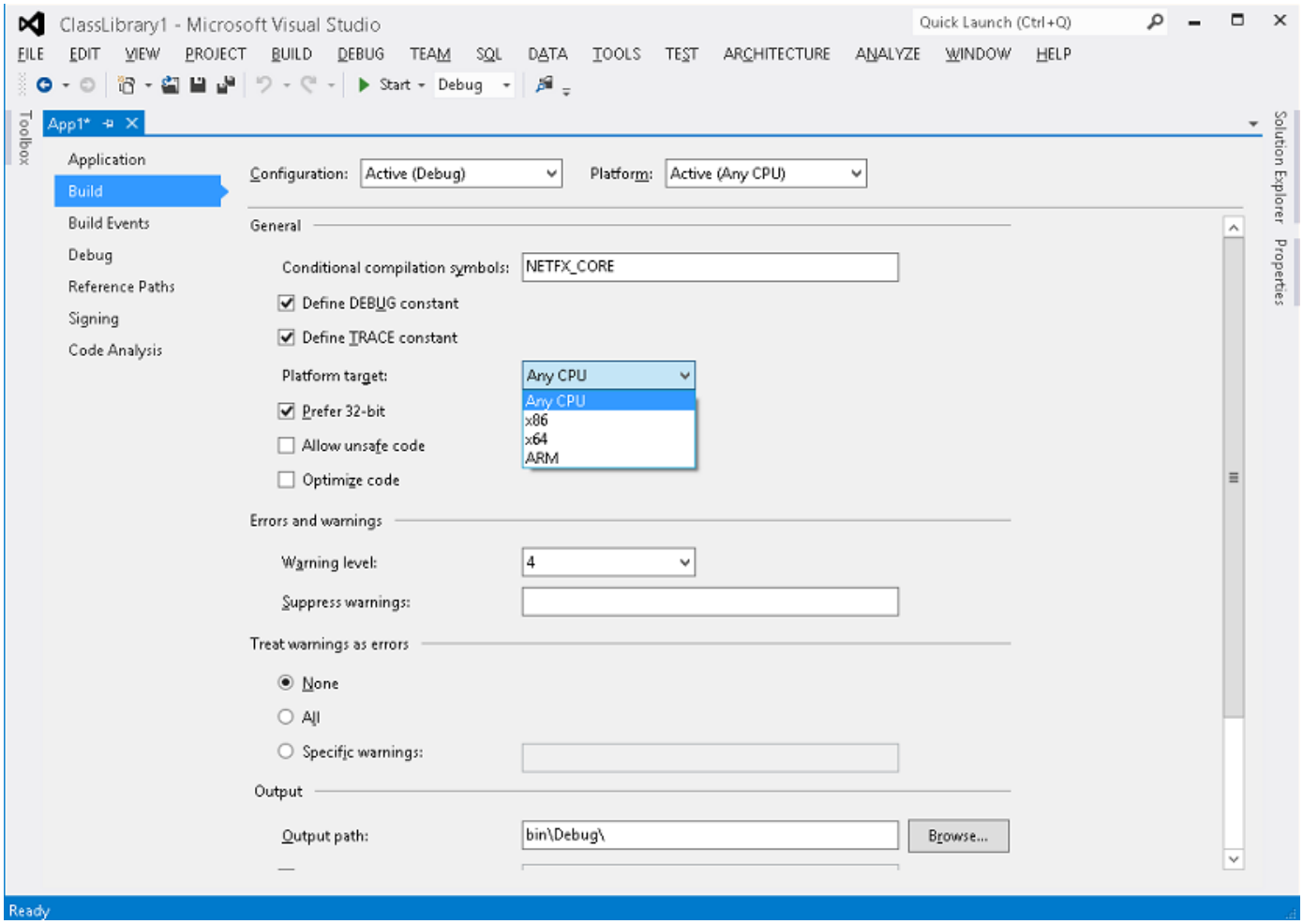
عنوان: **آشنایی با CLR: قسمت چهارم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۵ ۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در قسمت قبلی با اسمبلی‌ها تا حدی آشنا شدیم. امروز می‌خواهیم یاد بگیریم که چگونه اسمبلی‌ها در حافظه بارگذاری می‌شوند. همانطور که می‌دانید CLR مسئول اجرای کدهای داخل اسمبلی‌هاست. به همین دلیل یک نسخه‌ی دات نت فریم ورک هم باید در ماشین مقصد نصب باشد. به همین منظور مایکروسافت بسته‌های توزیع شونده‌ی دات نت فریمورک را فراهم کرده تا به سادگی بر روی سیستم مشتری نصب شوند و بعضی از ویندوزها نیز نسخه‌های متفاوتی از دات نت فریم ورک را شامل می‌شوند.
برای اینکه مطمئن شوید که آیا دات نت فریم ورک نصب شده است، می‌توانید در شاخه‌ی system32 سیستم، وجود فایل MSCorEE.dll را بررسی نمایید. البته بر روی یک سیستم می‌تواند نسخه‌های مختلفی از یک دات نت فریم ورک نصب باشد. برای آگاهی از اینکه چه نسخه‌هایی بر روی سیستم نصب است باید مسیرهای زیر را مورد بررسی قرار دهید:

```
%SystemRoot%\Microsoft.NET\Framework
%SystemRoot%\Microsoft.NET\Framework64
```

بسته‌ی دات نت فریمورک شامل ابزار خط فرمانی به نام CLRVer.exe می‌شود که همه‌ی نسخه‌های نصب شده را نشان می‌دهد. این ابزار با سوییچ all می‌تواند نشان دهد که چه پروسه‌هایی در حال حاضر دارند از یک نسخه‌ی خاص استفاده می‌کنند. یا اینکه ID یک پروسه را به آن داده و نسخه‌ی در حال استفاده را بیابیم.

قبل از اینکه پروسه‌ی بارگیری یک اسمبلی را بررسی کنیم، بهتر است به نسخه‌های 32 و 64 بیتی ویندوز، نگاهی بیندازیم:
یک برنامه در حالت عمومی بر روی تمامی نسخه‌ها قابل اجراست و نیازی نیست که توسعه دهنده کار خاصی انجام دهد. ولی اگر توسعه دهنده نیاز داشته باشد که برنامه را محدود به پلتفرم خاصی کند، باید از طریق برگه build در projectProperties در قسمت PlatformTarget معماری پردازنده را انتخاب کند:



موقعیکه گزینه برای روی anyCPU تنظیم شده باشد و تیک گزینه perfer 32-bit را زده باشید، به این معنی است که بر روی هر سیستمی قابل اجراست؛ ولی اجرا به شیوه‌ی 32 بیت اصلح است. به این معنی که در یک سیستم 64 بیت برنامه را به شکل 32 بیت بالا می‌آورد.

بسته به پلتفرمی که برای توزیع انتخاب می‌کنید، کامپایلر به ساخت اسمبلی‌های با هدرهای P32(+) می‌پردازد. مایکروسافت دو ابزار خط فرمان را به نام‌های exe. DumpBin و exe. CoreFlags در راستای آزمایش و بررسی هدرهای تولید شده توسط کامپایلر ارائه کرده است.

موقعی که شما یک فایل اجرایی را اجرا می‌کنید، ابتدا هدرها را خوانده و طبق اطلاعات موجود تصمیم می‌گیرد برنامه به چه شکلی اجرا شود. اگر دارای هدر p32 باشد قابل اجرا بر روی سیستم‌های 32 و 64 بیتی است و اگر PE32+ باشد روی سیستم‌های 64 بیتی قابل اجرا خواهد بود. همچنین به بررسی معماری پردازنده که در قسمت هدر embed شده، پرداخته تا اطمینان کسب کند که با خصوصیات پردازنده مقصد مطابقت می‌کند.

نسخه‌های 64 بیتی ارائه شده توسط مایکروسافت دارای فناوری به نام WOW64 یا Windows On Windows64 هستند که اجازه‌ی اجرای برنامه‌های 32 بیت را روی نسخه‌های 64 بیتی، می‌دهند.

جدول زیر اطلاعاتی را ارائه میکند که در حالت عادی برنامه روی چه سیستم‌هایی ارائه شده است و اگر آن‌را محدود به نسخه‌های 32 یا 64 بیتی کنیم، نحوه‌ی اجرا آن بر روی سایر پلتفرم‌ها چگونه خواهد بود.

/platform Switch	Resulting Managed Module	x86 Windows	x64 Windows	ARM Windows RT
anycpu (the default)	PE32/agnostic	Runs as a 32-bit application	Runs as a 64-bit application	Runs as a 32-bit application
anycpu32bitpreferred	PE32/agnostic	Runs as a 32-bit application	Runs as a 32-bit application	Runs as a 32-bit application
x86	PE32/x86	Runs as a 32-bit application	Runs as a WoW64 application	Doesn't run
x64	PE32+/x64	Doesn't run	Runs as a 64-bit application	Doesn't run
ARM	PE32/ARM	Doesn't run	Doesn't run	Runs as a 32-bit application

بعد از اینکه هدر مورد آزمایش قرار گرفت و متوجه شد چه نسخه‌ای از آن باید اجرا شود، بر اساس نسخه‌ی انتخابی، یک از نسخه‌های MSCorEE سی و دو بیتی یا 64 بیتی یا ARM را که در شاخه‌ی system32 قرار دارد، در حافظه بارگذاری می‌نماید. در نسخه‌های 64 بیتی ویندوز که نیاز به MSCorEE نسخه‌های 32 بیتی احساس می‌شود، در آدرس زیر قرار گرفته است:

```
%SystemRoot%\SysWow64
```

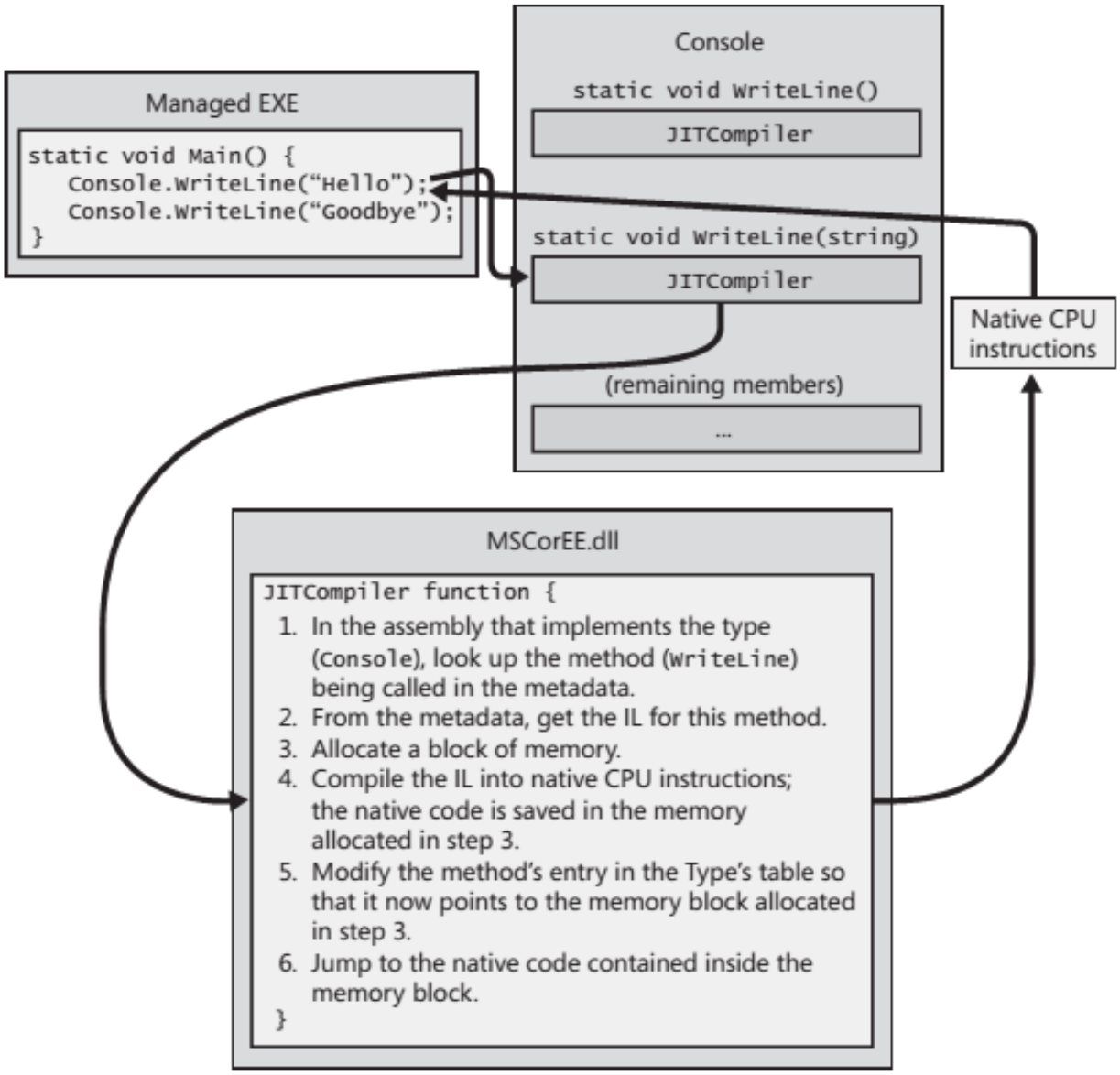
بعد از آن ترد اصلی پروسه، متدی را در MSCorEE صدا خواهد زد که موجب آماده سازی CLR بارگذاری اسمبلی اجرایی EXE در حافظه و صدا زدن مدخل ورودی برنامه یعنی متد Main می‌گردد. به این ترتیب برنامه‌ی مدیریت شده (managed) شما اجرا می‌گردد.

عنوان: **آشنایی با CLR: قسمت پنجم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۶ ۱:۱۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

اجرای کدهای اسمبلی

همانطور که قبلا ذکر کردیم یک اسمبلی شامل کدهای IL و متادیتا هاست. IL یک زبان غیر وابسته به معماری سی پی یو است که مایکروسافت پس از مشاوره‌های زیاد از طریق نویسندگان کامپایلر و زبان‌های آکادمی و تجاری آن را ایجاد کرده است. IL یک زبان کاملا سطح بالا نسبت به زبان‌های ماشین سی پی یو است. IL می‌تواند به انواع اشیاء دسترسی داشته و آن‌ها را دستکاری نماید و شامل دستورالعمل هایی برای ایجاد و آماده سازی اشیاست. صدا زدن متدهای مجازی بر روی اشیاء و دستکاری المان‌های یک آرایه به صورت مستقیم، از جمله کارهایی است که انجام می‌دهد. همچنین شامل دستوراتی برای صدور و کنترل استثناء هاست . شما می‌توانید IL را به عنوان یک زبان ماشین شیء گرایی تصور کنید.
معمولا برنامه نویس‌ها در یک زبان سطح بالا چون سی شارپ به نوشتن می‌پردازند و کمپایلر کد IL آن‌ها را ایجاد می‌کند و این کد IL می‌تواند به صورت اسمبلی نوشته شود. به همین علت مایکروسافت ابزار ILASM.exe و برای دی اسمبل کردن ILDASM.exe را ارائه کرده است.
این را همیشه به یاد داشته باشید که زبان‌های سطح بالا تنها به زیر قسمتی از قابلیت‌های CLR دسترسی دارند؛ ولی در IL Assembly توسعه دهنده به تمامی قابلیت‌های CLR دسترسی دارد. این انتخاب شما در زبان برنامه نویسی است که می‌خواهید تا چه حد به قابلیت‌های CLR دسترسی داشته باشید. البته یکپارچه بودن محیط در CLR باعث پیوند خوردن کدها به یکدیگر می‌شود. برای مثال می‌توانید قسمتی از یک پروژه که کار خواندن و نوشتن عملیات را به عهده دارد بر دوش C# قرار دهید و محاسبات امور مالی را به APL بسپارید.

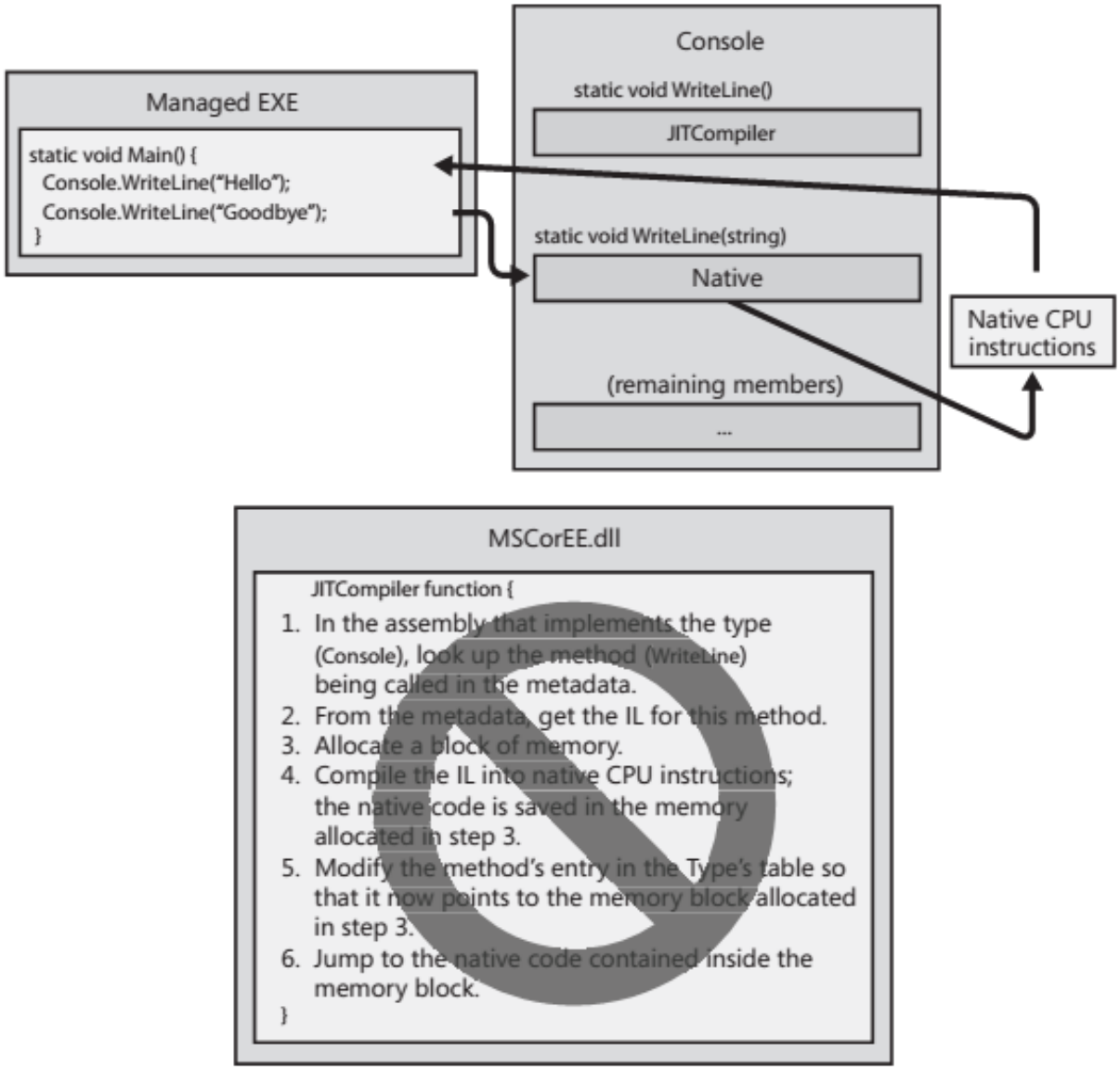
برای اجرا شدن کدهای IL، ابتدا CLR باید بر اساس معماری سی پی یو کد ماشین را به دست آورد که وظیفه‌ی تبدیل آن بر عهده Jit یا Just in Time است . شکل زیر نحوه انجام این کار را انجام می‌دهد:



قبل از اجرای متد Main، ابتدا CLR به دنبال ارجاعاتی می‌گردد که در این متد استفاده شده است تا یک ساختار داده داخلی، برای ارجاعات این متد در حافظه تشکیل شود. در شکل بالا یک ارجاع وجود دارد و آن هم شیء کنسول است. این ساختار داده داخلی شامل یک مدخل ورودی (آدرس آغاز در حافظه) به ازای هر متد تعریف شده در نوع کنسول است. هر مدخل ورودی شامل آدرسی است که متدها در آنجا پیاده سازی شده‌اند. موقعیکه این آماده سازی انجام می‌گیرد، آن‌ها را به سمت یک تابع مستند نشده در خود CLR به نام Jit Compiler ارسال می‌کند.
موقعیکه کنسول اولین متدش مثلا WriteLine را فراخوانی می‌کند، کامپایلر جیت صدا زده می‌شود. تابع کامپایلر جیت مسئولیت تبدیل کدهای IL را به کدهای بومی آن پلتفرم، به عهده دارد. از آنجایی که عمل کامپایل در همان لحظه یا در جا اتفاق می‌افتد (Just in time)، عموم این کامپایر را Jitter یا Jit Compiler می‌نامند.

موقعیکه صدا زدن آن متد به سمت jit انجام شد، جیت متوجه می‌شود که چه متدی درخواست شده و نحوه‌ی تعریف آن متد به چه صورتی است. جیت هم در متادیتای یک اسمبلی به جست و جو پرداخته و کدهای IL آن متد را دریافت می‌کند. سپس کدها را تایید و عملیات کامپایل به سمت کدهای بومی را آغاز می‌کند. در ادامه این کدهای بومی را در قطعه‌ای از حافظه ذخیره می‌کند.
سپس جیت به جایی بر می‌گردد که CLR از آنجا جیت را وارد کار کرده؛ یعنی مدخل ورودی متد writeline و سپس آدرس آن قطعه

حافظه را که شامل کد بومی است، بجای آن قطعه که به کد IL اشاره می‌کند، جابجا می‌کند و کد بومی شده را اجرا و نهایتا به محدوده‌ی main باز می‌گردد.
در شکل زیر مجددا همان متد صدا زده شده است. ولی از آنجا که قبلا کد کامپایل شده را به دست آوردیم، از همان استفاده می‌کنیم و دیگر تابع جیت را صدا نمی‌زنیم.



توجه داشته باشید، در متدهای چند ریختی که شکل‌های متفاوتی از پارامترها را دارند، هر کدام کمپایل جداگانه‌ای صورت می‌گیرد. یعنی برای متدهای زیر جیت برای هر کدام جداگانه فراخوانی می‌شود.

```
WriteLine("Hello");
WriteLine();
```

در مقاله‌ی آینده عملکرد جیت را بیشتر مورد بررسی قرار می‌دهیم و در مورد دیباگ کردن و به نظرم برتری CLR را نسبت به زبان‌های مدیریت نشده، بررسی می‌کنیم.

عنوان: **آشنایی با CLR: قسمت ششم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۷ ۰:۵۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در مقاله قبلی مبحث کامپایلر JIT را آغاز کردیم. در این قسمت قصد داریم مبحث کارآیی CLR و مباحث دیباگینگ را پیش بکشیم.
از آنجا که یک کد مدیریت نشده، مبحث کارهای JIT را ندارد، ولی CLR مجبور است وقتی را برای آن بگذارد، به نظر می‌رسد ما با یک نقص کوچک در کارآیی روبرو هستیم. گفتیم که جیت کدها را در حافظه‌ی پویا ذخیره می‌کند. به همین خاطر با terminate شدن یا خاتمه دادن به برنامه، این کدها از بین می‌روند یا اینکه اگر دو نمونه از برنامه را اجرا کنیم، هر کدام جداگانه کد را تولید می‌کنند و هر کدام برای خودشان حافظه‌ای بر خواهند داشت و اگر مقایسه‌ای با کدهای مدیریت نشده داشته باشید، در مورد مصرف حافظه یک مشکل ایجاد می‌کند. همچنین JIT در حین تبدیل به کدهای بومی یک بهینه سازی روی کد هم انجام میدهد که این بهینه سازی وقتی را به خود اختصاص می‌دهد ولی همین بهینه سازی کد موجب کارآیی بهتر برنامه می‌گردد.
در زبان سی شارپ دو سوئیچ وجود دارند که بر بهینه سازی کد تاثیر گذار هستند؛ سوئیچ‌های debug و optimize. در جدول زیر تاثیر هر یک از سوئیچ‌ها را بر کیفیت کد IL و JIT در تبدیل به کد بومی را نشان میدهد.

Compiler Switch Settings	C# IL Code Quality	JIT Native Code Quality
/optimize- /debug- (this is the default)	Unoptimized	Optimized
/optimize- /debug(+/full/pdbonly)	Unoptimized	Unoptimized
/optimize+ /debug(-/+/full/pdbonly)	Optimized	Optimized

موقعیکه از دستور optimize- استفاده می‌شود، کد IL تولید شده شامل تعداد زیادی از دستورات بدون دستورالعمل No Operation یا به اختصار NOP و پرش‌های شاخه‌ای به خط کد بعدی می‌باشد. این دستور العمل‌ها ما را قادر میسازند تا ویژگی edit & Continue را برای دیباگ کردن و یک سری دستورالعمل‌ها را برای کدنویسی راحت‌تر برای دیباگ کردن و ایجاد break pointها داشته باشیم.

موقعی که کد IL بهینه شده تولید شود، این خصوصیات اضافه حذف خواهند شد و دنبال کردن خط به خط کد، کار سختی می‌شود. ولی در عوض فایل نهایی exe یا dll، کوچکتر خواهد شد. بهینه سازی IL توسط JIT حذف خواهد شد و برای کسانی که دوست دارند کدهای IL را تحلیل و آنالیز کنند، خواندنش ساده‌تر و آسان‌تر خواهد بود.

نکته‌ی بعدی اینکه موقعیکه شما از سوئیچ debug(+/full/pdbonly/) استفاده می‌کنید، یک فایل PDB یا Program Database ایجاد می‌شود. این فایل به دیباگرها کمک می‌کند تا متغیرهای محلی را شناسایی و به کدهای IL متصل شوند. کلمه‌ی full بدین معنی است که JIT می‌تواند دستورات بومی را ردیابی کند تا مبداء آن کد را پیدا کند. سبب می‌شود که ویژوال استودیو به یک دیباگر متصل شده تا در حین اجرای پروسه، آن را دیباگ کند. در صورتی که این سوئیچ را استفاده نکنید، به طور پیش فرض پروسه اجرا و مصرف حافظه کمتر می‌شود. اگر شما پروسه‌ای را اجرا کنید که دیباگر به آن متصل شود، به طور اجباری JIT مجبور به انجام عملیات ردیابی خواهد شد؛ مگر اینکه گزینه‌ی suppress jit optimization on module load را غیرفعال کرده باشید.
موقعیکه در ویژوال استودیو دو حالت دیباگ و ریلیز را انتخاب می‌کنید، در واقع تنظیمات زیر را اجرا می‌کنید:

```
//debug

/optimize-
/debug:full

//=======================
```

```
//Release

/optimize+
/debug:pdbonly
```

احتمالا موارد بالا به شما می‌گویند که یک سیستم مبتنی بر CLR مشکلات زیادی دارد که یکی از آن‌ها، زمان‌بر بودن انجام عملیات فرآیند پردازش است و دیگری مصرف زیاد حافظه و عدم اشترک حافظه که در مورد کامپایل جیت به آن اشاره کردیم. ولی در بند بعدی قصد داریم نظرتان را عوض کنم.

اگر خیلی شک دارید که واقعا یک برنامه‌ی CLR کارآیی یک برنامه را پایین می‌آورد، بهتر هست به بررسی کارآیی چند برنامه غیر آزمایشی noTrial که حتی خود مایکروسافت آن برنامه‌ها را ایجاد کرده است بپردازید و آن‌ها را با یک برنامه‌ی unmanaged مقایسه کنید. قطعا باعث تعجب شما خواهد شد. این نکته دلایل زیادی دارد که در زیر تعدادی از آن‌ها را بررسی می‌کنیم.
اینکه CLR در محیط اجرا قصد کمپایل دارد، باعث آشنایی کامپایلر با محیط اجرا می‌گردد. از این رو تصمیماتی را که می‌گیرد، می‌تواند به کارآیی یک برنامه کمک کند. در صورتیکه یک برنامه‌ی unmanaged که قبلا کمپایل شده و با محیط‌های متفاوتی که روی آن‌ها اجرا میشود، هیچ آشنایی ندارد و نمیتواند از آن محیطها حداکثر بهره‌وری لازم را به عمل آورد.
برای آشنایی با این ویژگی‌ها توجه شما را به نکات ذیل جلب می‌کنم:

یک. JIT می‌تواند با نوع پردازنده آشنا شود که آیا این پردازنده از نسل پنتیوم 4 است یا نسل Core i. به همین علت می‌تواند از این مزیت استفاده کرده و دستورات اختصاصی آن‌ها را به کار گیرد، تا برنامه با performance بالاتری اجرا گردد. در صورتی که unmanaged باید حتما دستورات را در پایین‌ترین سطح ممکن و عمومی اجرا کند؛ در صورتیکه شاید یک دستور اختصاصی در یک سی پی یو خاص، در یک عملیات موجب 4 برابر، اجرای سریعتر شود.

دو. JIT میتواند بررسی هایی را که برابر false هستند، تشخیص دهد. برای فهم بهتر، کد زیر را در نظر بگیرید:

```
if (numberOfCPUs > 1) {
...
}
```

کد بالا در صورتیکه پردازنده تک هسته‌ای باشد یک کد بلا استفاده است که جیت باید وقتی را برای کامپایل آن اختصاص دهد؛ در صورتیکه JIT باهوش‌تر از این حرفاست و در کدی که تولید می‌کند، این دستورات حذف خواهند شد و باعث کوچکتر شدن کد و اجرای سریعتر می‌گردد.

سه. مورد بعدی که هنوز پیاده سازی نشده، ولی احتمال اجرای آن در آینده است، این است که یک کد می‌تواند جهت تصحیح بعضی موارد چون مسائل مربوط به دیباگ کردن و مرتب سازی‌های مجدد، عمل کامپایل را مجددا برای یک کد اعمال نماید.
دلایل بالا تنها قسمت کوچکی است که به ما اثبات می‌کند که چرا CLR می‌تواند کارآیی بهتری را نسبت به زبان‌های unmanaged امروزی داشته باشد. همچنین قول‌هایی از سازندگان برای بهبود کیفیت هر چه بیشتر این سیستم‌ها به گوش می‌رسد.

کارآیی بالاتر
اگر برنامه‌ای توسط شما بررسی شد و دیدید که نتایج مورد نیاز در مورد performance را نشان نمی‌دهد، می‌توانید از ابزار کمکی که مایکروسافت در بسته‌های فریمورک دات نت قرار داده است استفاده کنید. نام این ابزار Ngen.exe است و وظیفه‌ی آن این است که وقتی برنامه بر روی یک سیستم برای اولین مرتبه اجرا می‌گردد، کد همه‌ی اسمبلی‌ها را تبدیل کرده و آن‌ها روی دیسک ذخیره می‌کند. بدین ترتیب در دفعات بعدی اجرا، JIT بررسی می‌کند که آیا کد کامپایل شده‌ی اسمبلی از قبل موجود است یا خیر. در صورت وجود، عملیات کامپایل به کد بومی لغو شده و از کد ذخیره شده استفاده خواهد کرد.
نکته‌ای که باید در حین استفاده از این ابزار به آن دقت کنید این است که کد در محیط‌های واقعی اجرا چندان بهینه نیست. بعدا در مورد این ابزار به تفصیل صحبت می‌کنیم.

system.runtime.profileoptimization

کلاس بالا سبب می‌شود که CLR در یک فایل ثبت کند که چه متدهایی در حین اجرای برنامه کمپایل شوند تا در آینده در حین آغاز اجرای برنامه کامپایلر JIT بتواند همزمان این متدها را در ترد دیگری کامپایل کند. اگر برنامه‌ی شما روی یک پردازنده‌ی چند هسته‌ای اجرا می‌شود، در نتیجه اجرای سریعتری خواهید داشت. به این دلیل که چندین متد به طور همزمان در حال کمپایل شدن هستند و همزمان با آماده سازی برنامه برای اجرا اتفاق می‌افتد؛ به جای اینکه عمل کمپایل همزمان با تعامل کاربر با برنامه باشد.

عنوان: **آشنایی با CLR: قسمت هفتم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۸ ۱:۲۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

کدهای IL و تایید آن ها

ساختار استکی
IL از ساختار استک استفاده می‌کند. به این معنی که تمامی دستور العمل‌ها داخل آن push شده و نتیجه‌ی اجرای آن‌ها pop می‌شوند. از آنجا که IL به طور مستقیم ارتباطی با ثبات‌ها ندارد، ایجاد زبانهای برنامه نویسی جدید بر اساس CLR بسیار راحت‌تر هست و عمل کامپایل، تبدیل کردن به کدهای IL می‌باشد.

بدون نوع بودن(Typeless)
از دیگر مزیت‌های آن این است که کدهای IL بدون نوع هستند. به این معنی که موقع افزودن دستورالعملی به داخل استک، دو عملگر وارد می‌شوند و هیچ جداسازی در رابطه با سیستم‌های 32 یا 64 بیت صورت نمی‌گیرد و موقع اجرای برنامه است که تصمیم می‌گیرد از چه عملگرهایی باید استفاده شود.

Virtual Address Space
بزرگترین مزیت این سیستم‌ها امنیت و مقاومت آن هاست. موقعی که تبدیل کد IL به سمت کد بومی صورت می‌گیرد، CLR فرآیندی را با نام verification یا تاییدیه، اجرا می‌کند. این فرآیند تمامی کدهای IL را بررسی می‌کند تا از امنیت کدها اطمینان کسب کند. برای مثال بررسی می‌کند که هر متدی صدا زده می‌شود با تعدادی پارامترهای صحیح صدا زده شود و به هر پارامتر آن نوع صحیحش پاس شود و مقدار بازگشتی هر متد به درستی استفاده شود. متادیتا شامل اطلاعات تمامی پیاده سازی‌ها و متدها و نوع هاست که در انجام تاییدیه مورد استفاده قرار می‌گیرد.
در ویندوز هر پروسه، یک آدرس مجازی در حافظه دارد و این جدا سازی حافظه و ایجاد یک حافظه مجازی کاری لازم اجراست. شما نمی‌توانید به کد یک برنامه اعتماد داشته باشید که از حد خود تخطی نخواهد کرد و فرآیند برنامه‌ی دیگر را مختل نخواهد کرد. با خواندن و نوشتن در یک آدرس نامعتبر حافظه، ما این اطمینان را کسب می‌کنیم که هیچ گاه تخطی در حافظه صورت نمی‌گیرد.
قبلا به طور مفصل در این مورد ذخیره سازی در حافظه صحبت کرده ایم.

Hosting
از آنجا که پروسه‌های ویندوزی به مقدار زیادی از منابع سیستم عامل نیاز دارند که باعث کاهش منابع و محدودیت در آن می‌شوند و نهایت کارآیی سیستم را پایین می‌آورد، ولی با کاهش تعدادی برنامه‌های در حال اجرا به یک پروسه‌ی واحد می‌توان کارآیی سیستم را بهبود بخشید و منابع کمتری مورد استفاده قرار می‌گیرند که این یکی دیگر از مزایای کدهای managed نسبت به unmanaged است. CLR در حقیقت این قابلیت را به شما می‌دهد تا چند برنامه‌ی مدیریت شده را در قالب یک پروسه به اجرا درآورید. هر برنامه‌ی مدیریت شده به طور پیش فرض بر روی یک appDomain اجرا می‌گردد و هر فایل EXE روی حافظه‌ی مجازی مختص خودش اجرا می‌شود. هر چند پروسه‌هایی از قبیل IIS و SQL Server که پروسه‌های CLR را پشتیبانی یا هاست می‌کنند می‌توانند تصمیم بگیرند که آیا appDomainها را در یک پروسه‌ی واحد اجرا کنند یا خیر که در مقاله‌های آتی آن را بررسی می‌کنیم.

کد ناامن یا غیر ایمن UnSafe Code
به طور پیش فرض سی شارپ کدهای ایمنی را تولید می‌کند، ولی این اجازه را می‌دهد که اگر برنامه نویس بخواهد کدهای ناامن بزند، قادر به انجام آن باشد. این کدهای ناامن دسترسی مستقیم به خانه‌های حافظه و دستکاری بایت هاست. این مورد قابلیت قدرتمندی است که به توسعه دهنده اجازه می‌دهد که با کدهای مدیریت نشده ارتباط برقرار کند یا یک الگوریتم با اهمیت زمانی بالا را جهت بهبود کارآیی، اجرا کند.
هر چند یک کد ناامن سبب ریسک بزرگی می‌شود و می‌تواند وضعیت بسیاری از ساختارهای ذخیره شده در حافظه را به هم بزند و امنیت برنامه را تا حد زیادی کاهش دهد. به همین دلیل سی شارپ نیاز دارد تا تمامی متدهایی که شامل کد unsafe هستند را با کلمه کلیدی unsafe علامت گذاری کند. همچنین کمپایلر سی شارپ نیاز دارد تا شما این کدها را با سوئیچ unsafe/ کامپایل کنید.

موقعیکه جیت تلاش دارد تا یک کد ناامن را کامپایل کند، اسمبلی را بررسی می‌کند که آیا این متد اجازه و تاییدیه آن را دارد یا خیر. آیا System.Security.Permissions.SecurityPermission با فلگ SkipVerification مقدار دهی شده است یا خیر. اگر پاسخ مثبت بود JIT آن‌ها را کامپایل کرده و اجازه‌ی اجرای آن‌ها را می‌دهد. CLR به این کد اعتماد می‌کند و امیدوار است که آدرس دهی مستقیم و دستکاری بایت‌های حافظه موجب آسیبی نگردد. ولی اگر پاسخ منفی بود، یک استثناء از نوع System.InvalidProgramException یا System.Security.VerificationException را ایجاد می‌کند تا از اجرای این متد جلوگیری به عمل آید. در واقع کل برنامه خاتمه میابد ولی آسیبی به حافظه نمی‌زند.

پی نوشت: سیستم به اسمبلی هایی که از روی ماشین یا از طریق شبکه به اشتراک گذاشته می‌شوند اعتماد کامل میکند که این اعتماد شامل کدهای ناامن هم می‌شود ولی به طور پیش فرض به اسمبلی هایی از طریق اینترنت اجرا می‌شوند اجازه اجرای کدهای ناامن را نمی‌دهد و اگر شامل کدهای ناامن شود یکی از خطاهایی که در بالا به آن اشاره کردیم را صادر می‌کنند. در صورتی که مدیر یا کاربر سیستم اجازه اجرای آن را بدهد تمامی مسئولیت‌های این اجرا بر گردن اوست.

در این زمینه مایکروسافت ابزار سودمندی را با نام PEVerify.exe را معرفی کرده است که به بررسی تمامی متدهای یک اسمبلی پرداخته و در صورت وجود کد ناامن به شما اطلاع میدهد. بهتر است از این موضوع اطلاع داشته باشید که این ابزار نیاز دارد تا به متادیتاهای یک اسمبلی نیاز داشته باشید. باید این ابزار بتواند به تمامی ارجاعات آن دسترسی داشته باشد که در مورد عملیات بایندینگ در آینده بیشتر صحبت می‌کنیم.

IL و حقوق حق تالیف آن

بسیاری از توسعه دهندگان از اینکه IL هیچ شرایطی برای حفظ حق تالیف آن‌ها ایجاد نکرده است، ناراحت هستند. چرا که ابزارهای زیادی هستند که با انجام عملیات مهندسی معکوس می‌توانند به الگوریتم آنان دست پیدا کنند و میدانید که IL خیلی سطح پایین نیست و برگرداندن آن به شکل یک کد، کار راحت‌تری هست و بعضی ابزارها کدهای خوبی هم ارائه می‌کنند. از دست این ابزارها می‌توان به ILDisassembler و JustDecompile اشاره کرد.
اگر علاقمند هستید این عیب را برطرف کنید، می‌توانید از ابزارهای ثالث که به ابزارهای obfuscator (یک نمونه سورس باز) معروف هستند استفاده کنید تا با کمی پیچیدگی در متادیتاها، این مشکل را تا حدی برطرف کنند. ولی این ابزارها خیلی کامل نیستند، چرا که نباید به کامپایل کردن کار لطمه بزنند. پس اگر باز خیلی نگران این مورد هستید می‌توانید الگوریتم‌های حساس و اساسی خود را در قالب unmanaged code ارائه کنید که در بالا اشاراتی به آن کرده‌ایم.
برنامه‌های تحت وب به دلیل عدم دسترسی دیگران از امنیت کاملتری برخوردار هستند.

عنوان: **آشنایی با CLR: قسمت هشتم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۹ ۹:۱۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در قسمت پنجم در مورد ابزار Ngen کمی صحبت کردیم و در این قسمت هم در مورد آن صحبت هایی خواهیم کرد. گفتیم که این ابزار در زمان نصب، اسمبلی‌ها را کامپایل می‌کند تا در زمان اجرا JIT وقتی برای آن نگذارد. این کار دو مزیت به همراه دارد:

بهینه سازی زمان آغاز به کار برنامه
کاهش صفحات کاری برنامه: از آنجا که برنامه از قبل کامپایل شده، فراهم کردن صفحه بندی از ابتدای کار امر چندان دشواری نخواهد بود؛ لذا در این حالت صفحه بندی حافظه به صورت پویاتری انجام می‌گردد. شیوه‌ی کار به این صورت است که اسمبلی‌ها به چندین پروسه‌ی کاری کوچک‌تر تبدیل شده تا صفحه بندی هر کدام جدا صورت گیرد و محدوده‌ی صفحه بندی کوچکتر می‌شود. در نتیجه کمتر نقصی در صفحه بندی دیده شده یا کلا دیده نخواهد شد. نتیجه‌ی کار هم در یک فایل ذخیره می‌گردد که این فایل می‌تواند نگاشت به حافظه شود تا این قسمت از حافظه به طور اشتراکی مورد استفاده قرار گیرد و بدین صورت نیست که هر پروسه‌ای برای خودش قسمتی را گرفته باشد.
موقعی که اسمبلی، کد IL آن به کد بومی تبدیل می‌شود، یک اسمبلی جدید ایجاد شده که این فایل جدید در مسیر زیر قرار می‌گیرد:

```
%SystemRoot%\Assembly\NativeImages_v4.0.#####_64
```

نام دایرکتوری اطلاعاتی شامل نسخه CLR و اطلاعاتی مثل اینکه برنامه بر اساس چه نسخه‌ای 32 یا 64 بیت کامپایل شده است.

معایب
احتمالا شما پیش خود می‌گویید این مورد فوق العاده امکان جالبی هست. کدها از قبل تبدیل شده‌اند و دیگر فرآیند جیت صورت نمی‌گیرد. در صورتیکه ما تمامی امکانات یک CLR مثل مدیریت استثناءها و GC و ... را داریم، ولی غیر از این یک مشکلاتی هم به کارمان اضافه می‌شود که در زیر به آنها اشاره می‌کنیم:

عدم محافظت از کد در برابر بیگانگان: بعضی‌ها تصور می‌کنند که این کد را می‌توانند روی ماشین شخصی خود کامپایل کرده و فایل ngen را همراه با آن ارسال کنند. در این صورت کد IL نخواهد بود ولی موضوع این هست اینکار غیر ممکن است و هنوز استفاده از اطلاعات متادیتاها پابرجاست به خصوص در مورد اطلاعات چون reflection و serializationها . پس کد IL کماکان همراهش هست. نکته‌ی بعدی اینکه انتقال هم ممکن نیست؛ بنا به شرایطی که در مورد بعدی دلیل آن را متوجه خواهید شد.

از سینک با سیستم خارج میشوند: موقعیکه CLR، اسمبلی‌ها را به داخل حافظه بار می‌کند، یک سری خصوصیات محیط فعلی را با زمانیکه عملیات تبدیل IL به کد ماشین صورت گرفته است، چک می‌کند. اگر این خصوصیات هیچ تطابقی نداشته باشند، عملیات JIT همانند سابق انجام می‌گردد. خصوصیات و ویژگی‌هایی که چک می‌شوند به شرح زیر هستند:
ورژن CLR: در صورت تغییر، حتی با پچ‌ها و سرویس پک ها.
نوع پردازنده: در صورت تغییر پردازنده یا ارتقا سخت افزاری.
نسخه سیستم عامل : ارتقاء با سرویس پک ها.
MVID یا Assemblies Identity module Version Id: در صورت کامپایل مجدد تغییر می‌کند.
Referenced Assembly's version ID: در صورت کامپایل مجدد اسمبلی ارجاع شده.
تغییر مجوزها: در صورتی که تغییری نسبت به اولین بار رخ دهد؛ مثلا در قسمت قبلی در مورد اجازه نامه اجرای کدهای ناامن صحبت کردیم. برای نمونه اگر در همین اجازه نامه تغییری رخ دهد، یا هر نوع اجازه نامه دیگری، برنامه مثل سابق (جیت) اجرا خواهد شد.

پی نوشت: در آپدیت‌های دات نت فریم ورک به طور خودکار ابزار ngen صدا زده شده و اسمبلی‌ها مجددا کمپایل و دخیره میشوند و برنامه سینک و آپدیت باقی خواهد ماند.

کارایی پایین کد در زمان اجرا: استفاده از ngen از ابتدا قرار بود کارآیی را با حذف جیت بالا ببرد، ولی گاهی اوقات در بعضی شرایط ممکن نیست. کدهایی که ngen تولید می‌کند به اندازه‌ی جیت بهینه نیستند. برای مثال ngen نمی‌تواند بسیاری از دستورات خاص پردازنده را جز در زمان runtime مشخص کند. همچنین فیلدهایی چون static را از آنجا که نیاز است آدرس واقعی آن‌ها در زمان اجرا به دست بیاید، مجبور به تکنیک و ترفند میشود و موارد دیگری از این قبیل.
پس حتما نسخه‌ی ngen شده و غیر ngen را بررسی کنید و کارآیی هر دو را با هم مقایسه کنید. برای بسیاری از برنامه‌ها کاهش صفحه بندی یک مزیت و باعث بهبود کارآیی می‌شود. در نتیجه در این قسمت ngen برنده اعلام می‌شود.

توجه کنید برای سیستم‌هایی که در سمت سرور به فعالیت می‌پردازند، از آنجا که تنها اولین درخواست برای اولین کاربر کمی زمان می‌برد و برای باقی کاربران درخواست با سرعت بالاتری اجرا می‌گردد و اینکه برای بیشتر برنامه‌های تحت سرور از آنجا که تنها یک نسخه در حال اجراست، هیچ مزیت صفحه بندی را ngen ایجاد نمی‌کند.

برای بسیاری از برنامه‌های کلاینت که تجربه‌ی startup طولانی دارند، مایکروسافت ابزاری را به نام Managed Profile Guided Optimization Tool یا MPGO.exe دارد. این ابزار به تحلیل اجرای برنامه شما پرداخته و بررسی می‌کند که در زمان آغازین برنامه چه چیزهایی نیاز است. اطلاعات به دست آمده از تحلیل به سمت ngen فرستاده شده تا کد بومی بهینه‌تری تولید گردد. موقعیکه شما آماده ارائه برنامه خود هستید، برنامه را از طریق این تحلیل و اجرا کرده و با قسمت‌های اساسی برنامه کار کنید. با این کار اطلاعاتی در مورد اجرای برنامه در داخل یک پروفایل embed شده در اسمبلی، قرار گرفته و ngen موقع تولید کد، این پروفایل را جهت تولید کد بهینه مطالعه خواهد کرد.

در مقاله‌ی بعدی در مورد FCL صحبت‌هایی خواهیم کرد.

عنوان: **آشنایی با CLR: قسمت نهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۳۰ ۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

.net framework شامل Framework Class Library یا به اختصار FCL است. FCL مجموعه‌ای از dll اسمبلی‌هایی است که صدها و هزاران نوع در آن تعریف شده‌اند و هر نوع تعدادی کار انجام می‌دهد. همچنین مایکروسافت کتابخانه‌های اضافه‌تری را چون azure و Directx نیز ارائه کرده است که باز هر کدام شامل نوع‌های زیادی می‌شوند. این کتابخانه به طور شگفت آوری باعث سرعت و راحتی توسعه دهندگان در زمینه فناوری‌های مایکروسافت گشته است.

تعدادی از فناوری‌هایی که توسط این کتابخانه پشتیبانی می‌شوند در زیر آمده است:

Web Service : این فناوری اجازه‌ی ارسال و دریافت پیام‌های تحت شبکه را به خصوص بر روی اینترنت، فراهم می‌کند و باعث ارتباط جامع‌تر بین برنامه‌ها و فناوری‌های مختلف می‌گردد. در انواع جدیدتر WCF و Web Api نیز به بازار ارائه شده‌اند.

webform و MVC : فناوری‌های تحت وب که باعث سهولت در ساخت وب سایت‌ها می‌شوند که وب فرم رفته رفته به سمت منسوخ شدن پیش می‌رود و در صورتی که قصد دارید طراحی وب را آغاز کنید توصیه میکنم از همان اول به سمت MVC بروید.

Rich Windows GUI Application : برای سهولت در ایجاد برنامه‌های تحت وب حالا چه با فناوری WPF یا فناوری قدیمی و البته منسوخ شده Windows Form.
Windows Console Application : برای ایجاد برنامه‌های ساده و بدون رابط گرافیکی.
Windows Services : شما می‌توانید یک یا چند سرویس تحت ویندوز را که توسط Service Control Manager یا به اختصار SCM کنترل می‌شوند، تولید کنید.
Database stored Procedure : نوشتن stored procedure بر روی دیتابیس‌هایی چون sql server و اوراکل و ... توسط فریم ورک دات نت مهیاست.
Component Libraray : ساخت اسمبلی‌های واحدی که می‌توانند با انواع مختلفی از موارد بالا ارتباط برقرار کنند.
Portable Class Libary : این نوع پروژه‌ها شما را قادر می‌سازد تا کلاس‌هایی با قابلیت انتقال پذیری برای استفاده در سیلور لایت، ویندوز فون و ایکس باکس و فروشگاه ویندوز و ... تولید کنید.

ازآنجا که یک کتابخانه شامل زیادی نوع می‌گردد سعی شده است گروه بندی‌های مختلفی از آن در قالبی به اسم فضای نام namespace تقسیم بندی گردند که شما آشنایی با آن‌ها دارید. به همین جهت فقط تصویر زیر را که نمایشی از فضای نام‌های اساسی و مشترک و پرکاربرد هستند، قرار می‌دهم.

Namespace	Description of Contents
System	All of the basic types used by every application
System.Data	Types for communicating with a database and processing data
System.IO	Types for doing stream I/O and walking directories and files
System.Net	Types that allow for low-level network communications and working with some common Internet protocols
System.Runtime.InteropServices	Types that allow managed code to access unmanaged operating system platform facilities such as COM components and functions in Win32 or custom DLLs
System.Security	Types used for protecting data and resources
System.Text	Types to work with text in different encodings, such as ASCII and Unicode
System.Threading	Types used for asynchronous operations and synchronizing access to resources
System.Xml	Types used for processing Extensible Markup Language (XML) schemas and data

در CLR مفهومی به نام Common Type System یا CTS وجود دارد که توضیح می‌دهد نوع‌ها باید چگونه تعریف شوند و چگونه باید رفتار کنند که این قوانین از آنجایی که در ریشه‌ی CLR نهفته است، بین تمامی زبان‌های دات نت مشترک می‌باشد. تعدادی از مشخصات این CTS در زیر آورده شده است ولی در آینده بررسی بیشتری روی آنان خواهیم داشت:

فیلد
متد
پراپرتی
رویدادها

CTS همچنین شامل قوانین زیادی در مورد وضعیت کپسوله سازی برای اعضای یک نوع دارد:

private
public
Family یا در زبان‌هایی مثل سی ++ و سی شارپ با نام protected شناخته می‌شود.
family and assembly: این هم مثل بالایی است ولی کلاس مشتق شده باید در همان اسمبلی باشد. در زبان‌هایی چون سی شارپ و ویژوال بیسیک، چنین امکانی پیاده سازی نشده‌است و دسترسی به آن ممکن نیست ولی در IL Assembly چنین قابلیتی وجود دارد.
Assembly یا در بعضی زبان‌ها به نام internal شناخته می‌شود.
Family Or Assembly: که در سی شارپ با نوع Protected internal شناخته می‌شود. در این وضعیت هر عضوی در هر اسمبلی قابل ارث بری است و یک عضو فقط می‌تواند در همان اسمبلی مورد استفاده قرار بگیرد.

موارد دیگری که تحت قوانین CTS هستند مفاهیم ارث بری، متدهای مجازی، عمر اشیاء و .. است.

یکی دیگر از ویژگی‌های CTS این است که همه‌ی نوع‌ها از نوع شیء Object که در فضای نام system قرار دارد ارث بری کرده‌اند. به همین دلیل همه‌ی نوع‌ها حداقل قابلیت‌هایی را که یک نوع object ارئه میدهد، دارند که به شرح زیر هستند:

مقایسه‌ی دو شیء از لحاظ برابری.
به دست آوردن هش کد برای هر نمونه از یک شیء
ارائه‌ای از وضعیت شیء به صورت رشته ای
دریافت نوع شیء جاری

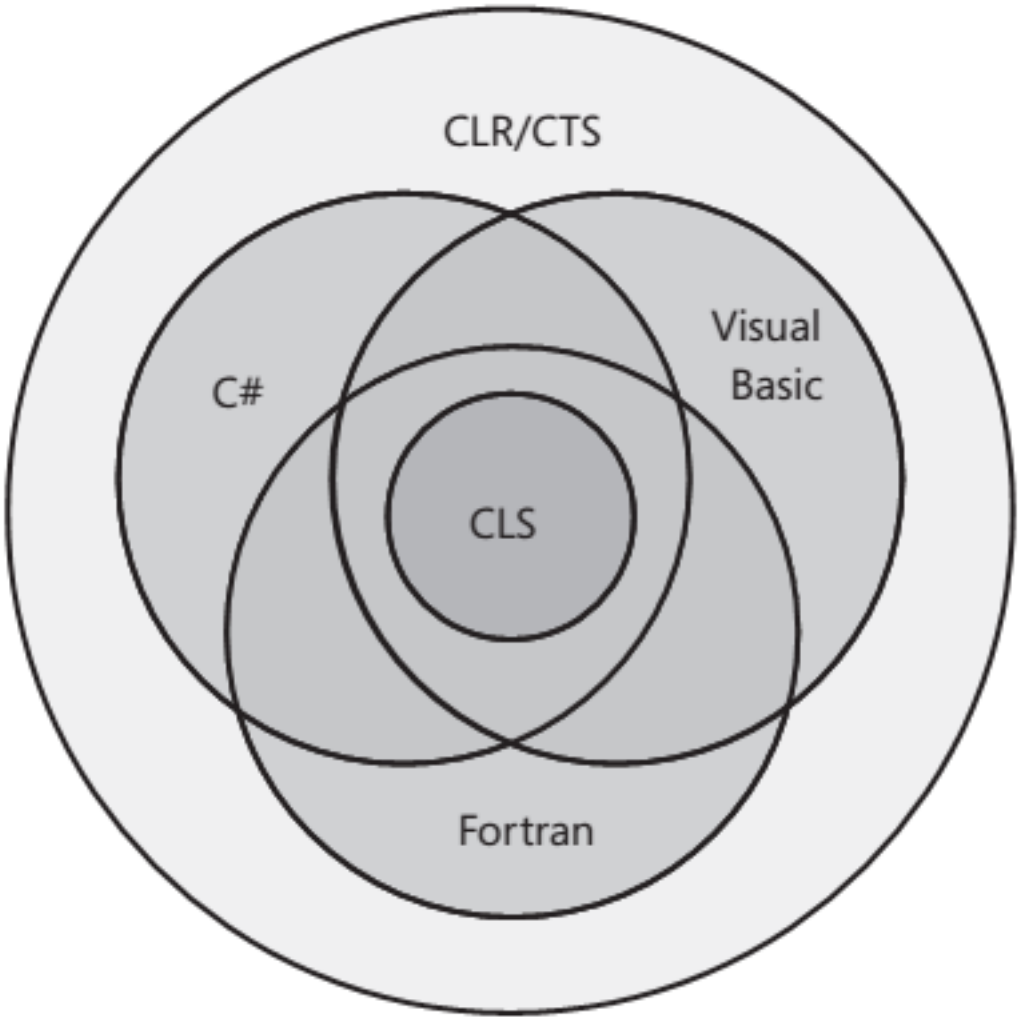
CLS

وجود COMها به دلیل ایجاد اشیاء در یک زبان متفاوت بود تا با زبان دیگر ارتباط برقرار کنند. در طرف دیگر CLR هم بین زبان‌های برنامه نویسی یکپارچگی ایجاد کرده است. یکپارچگی زبان‌های برنامه نویسی علل زیادی دارند. اول اینکه رسیدن به هدف یا یک الگوریتم خاص در زبان دیگر راحت‌تر از زبان پایه پروژه است. دوم در یک کار تیمی که افراد مختلف با دانش متفاوتی حضور دارند و ممکن است زبان هر یک متفاوت باشند.

برای ایجاد این یکپارچگی، مایکروسافت سیستم CLS یا Common Language Specification را راه اندازی کرد. این سیستم برای تولیدکنندگان کامپایلرها جزئیاتی را تعریف می‌کند که کامپایلر آن‌ها را باید با حداقل ویژگی‌های تعریف شده‌ی CLR، پشتیبانی کند.

CLR/CTS مجموعه‌ای از ویژگی‌ها را شامل می‌شود و گفتیم که هر زبانی بسیاری از این ویژگی‌ها را پشتیبانی می‌کند ولی نه کامل. به عنوان مثال برنامه نویسی که قصد کرده از IL Assembly استفاده کند، قادر است از تمامی این ویژگی‌هایی که CLR/CTS ارائه می‌دهند، استفاده کند ولی تعدادی دیگر از زبان‌ها مثل سی شارپ و فورترن و ویژوال بیسیک تنها بخشی از آن را استفاده می‌کنند و CLS *حداقل ویژگی که بین همه این زبان‌ها مشترک است را ارائه می‌کند.*

شکل زیر را نگاه کنید:



یعنی اگر شما دارید نوع جدیدی را در یک زبان ایجاد می‌کنید که قصد دارید در یک زبان دیگر استفاده شود، نباید از امتیازات ویژه‌ای که آن زبان در اختیار شما می‌گذارد و به بیان بهتر CLS آن‌ها را پشتیبانی نمی‌کند، استفاده کنید؛ چرا که کد شما ممکن است

در زبان دیگر مورد استفاده قرار نگیرد.

به کد زیر دقت کنید. تعدادی از کدها سازگاری کامل با CLS دارند که به آن‌ها CLS Compliant گویند و تعدادی از آن‌ها non-CLS-Compliant هستند یعنی با CLS سازگاری ندارند ولی استفاده از خاصیت [assembly: CLSCompliant(true)] باعث می‌شود که تا کامپایلر از پشتیبانی و سازگاری این کدها اطمینان کسب کند و در صورت وجود، از اجرای آن جلوگیری کند. با کمپایل کد زیر دو اخطار به ما میرسد.

```
using System;

// Tell compiler to check for CLS compliance
[assembly: CLSCompliant(true)]

namespace SomeLibrary {

// Warnings appear because the class is public
public sealed class SomeLibraryType {

// Warning: Return type of 'SomeLibrary.SomeLibraryType.Abc()'
// is not CLS-compliant
public UInt32 Abc() { return 0; }

// Warning: Identifier 'SomeLibrary.SomeLibraryType.abc()'
// differing only in case is not CLS-compliant
public void abc() { }

// No warning: this method is private
private UInt32 ABC() { return 0; }
}
}
```

اولین اخطار اینکه یکی از متدها یک عدد صحیح بدون علامت unsigned integer را بر می‌گرداند که همه‌ی زبان‌ها آن را پشتیبانی نمی‌کنند و خاص بعضی از زبان هاست.
دومین اخطار اینکه دو متد یکسان وجود دارند که در حروف بزرگ و کوچک تفاوت دارند. ولی زبان هایی چون ویژوال بیسیک نمی‌توانند تفاوتی بین دو متد abc و ABC بیابند.

نکته‌ی جالب اینکه اگر شما کلمه public را از جلوی نام کلاس بردارید تمامی این اخطارها لغو می‌شود. به این خاطر که این‌ها اشیای داخلی آن اسمبلی شناخته شده و قرار نیست از بیرون به آن دسترسی صورت بگیرد. عضو خصوصی کد بالا را ببینید؛ کامنت بالای آن می‌گوید که چون خصوصی است هشداری نمی‌گیرد، چون قرار نیست در زبان مقصد از آن به طور مستقیم استفاده کند.
برای دیدن قوانین CLS به این صفحه مراجعه فرمایید.

سازگاری با کدهای مدیریت نشده

در بالا در مورد یکپارچگی و سازگاری کدهای مدیریت شده توسط CLS صحبت کردیم ولی در مورد ارتباط با کدهای مدیریت نشده چطور؟
مایکروسافت موقعیکه CLR را ارئه کرد، متوجه این قضیه بود که بسیاری از شرکت‌ها توانایی اینکه کدهای خودشون را مجددا طراحی و پیاده سازی کنند، ندارند و خوب، سورس‌های مدیریت نشده‌ی زیادی هم موجود هست که توسعه دهندگان علاقه زیادی به استفاده از آن‌ها دارند. در نتیجه مایکروسافت طرحی را ریخت که CLR هر دو قسمت کدهای مدیریت شده و نشده را پشتیبانی کند. دو نمونه از این پشتیبانی را در زیر بیان می‌کنیم:
یک. کدهای مدیریت شده می‌توانند توابع مدیریت شده را در قالب یک dll صدا زده و از آن‌ها استفاده کنند.
دو. کدهای مدیریت شده می‌توانند از کامپوننت‌های COM استفاده کنند: بسیاری از شرکت‌ها از قبل بسیاری از کامپوننت‌های COM را ایجاد کرده بودند که کدهای مدیریت شده با راحتی با آن‌ها ارتباط برقرار می‌کنند. ولی اگر دوست دارید روی آن‌ها کنترل بیشتری داشته باشید و آن کدها را به معادل CLR تبدیل کنید؛ میتوانید از ابزار کمکی که مایکروسافت همراه فریم ورک دات نت ارائه کرده است استفاده کنید. نام این ابزار TLBIMP.exe می‌باشد که از Type Library Importer گرفته شده است.

سه. اگر کدهای مدیریت نشده‌ی زیادتری دارید شاید راحت‌تر باشد که برعکس کار کنید و کدهای مدیریت شده را در در یک برنامه‌ی مدیریت نشده اجرا کنید. این کدها می‌توانند برای مثال به یک Activex یا shell Extension تبدیل شده و مورد استفاده قرار گیرند. ابزارهای exe. TLBEXP و exe. RegAsm برای این منظور به همراه فریم ورک دات نت عرضه شده اند.
سورس کد Type Library Importer را میتوانید در کدپلکس بیابید.
در ویندوز 8 به بعد مایکروسافت API جدید را تحت عنوان WinsowsRuntime یا winRT ارائه کرده است . این api یک سیستم داخلی را از طریق کامپوننت‌های com ایجاد کرده و به جای استفاده از فایل‌های کتابخانه‌ای، کامپوننت‌ها api هایشان را از طریق متادیتاهایی بر اساس استاندارد ECMA که توسط تیم دات نت طراحی شده است معرفی می‌کنند.
زیبایی این روش اینست که کد نوشته شده در زبان‌های دات نت می‌تواند به طور مداوم با apiهای winrt ارتباط برقرار کند. یعنی همه‌ی کارها توسط CLR انجام می‌گیرد بدون اینکه لازم باشد از ابزار اضافی استفاده کنید. در آینده در مورد winRT بیشتر صحبت می‌کنیم.

سخن پایانی: ممنون از دوستان عزیز بابت پیگیری مطالب تا بدینجا. تا این قسمت فصل اول کتاب با عنوان **اصول اولیه CLR** بخش اول **مدل اجرای CLR** به پایان رسید.
ادامه‌ی مطالب بعد از تکمیل هر بخش در دسترس دوستان قرار خواهد گرفت.

عنوان: **آشنایی با CLR: قسمت دهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۲۵ ۲۰:۳۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, Windows, .NET Framework

در سلسله مقالات قبلی ما فصل اول از بخش اول را به پایان بردیم و مبحث آشنایی با CLR و نحوه‌ی اجرای برنامه را یاد گرفتیم. در این سلسله مقالات که مربوط به فصل دوم از بخش اول است، در مورد نحوه‌ی ساخت و توزیع برنامه صحبت می‌کنیم.

در طی این سال‌ها ویندوز به ناپایداری و پیچیدگی متهم شده است. صرف نظر از این که ویندوز شایستگی این اتهامات را دارد یاخیر، این اتهامات نتیجه‌ی چند عامل است:
اول از همه برنامه‌ها از dll هایی استفاده می‌کنند که بسیاری از آن‌ها نوشته‌ی برنامه نویسانشان نیست و توسط توسعه دهندگان دیگر ارائه شده‌اند و توسعه دهندگان مربوطه نمی‌توانند صد در صد مطمئن شوند که افراد دیگر، به چه نحوی از dll آن‌ها استفاده می‌کنند و در عمل ممکن هست باعث دردسرهای زیادی شود که البته این نوع مشکلات عموما از قبل خودشان را نشان نمی‌دهند، چرا که توسط سازنده‌ی برنامه تست و دیباگ شده‌اند.
موقعی کاربرها بیشتر دچار دردسر می‌گردند که برنامه‌های خودشان را به روز می‌کنند و عموما شرکت‌ها در آپدیت‌ها، فایل‌های جدید زیادی را روی سیستم کاربر منتقل می‌کنند که ممکن هست سازگاری با فایل‌های قبلی موجود نداشته باشند و از آنجا که همیشه تست این مورد برای توسعه دهنده امکان ندارد، به مشکلاتی بر می‌خورند و نمی‌توانند صد در صد مطمئن باشند که تغییرات جدید باعث تاثیر ناخوشایند نمی‌شود.
مطمئن هستم شما بسیاری از این مشکلات را دیده‌اید که کاربری یک برنامه را نصب می‌کند و شما متوجه می‌شوید که یک برنامه‌ی از قبل نصب شده به خاطر آن دچار مشکل می‌شود و این مورد به DLL hell مشهور هست. این مورد باعث ایجاد ترس و لرز برای کاربر شده تا با دقت بیشتری به نصب برنامه‌ها بپردازد.

دومین مورد مربوط به نصب برنامه‌ها است که متهم به پیچیدگی است. امروزه هر برنامه‌ای که روی سیستم نصب می‌شود، بر همه جای سیستم تاثیر می‌گذارد. یک برنامه را نصب می‌کنید و به هر دایرکتوری تعدادی فایل کپی می‌شود. تنظیمات ریجستری را آپدیت می‌کند، یک آیکن روی دسکتاپ و یکی هم start menu یا مترو را اضافه می‌کند. به این معنی که یک نصب کننده به عنوان یک موجودیت واحد شناخته نمی‌شود. شما نمی‌تونید راحت از یک برنامه بکاپ بگیرید. باید فایل‌های مختلفش را جمع آوری کنید و تنظیمات ریجیستری را ذخیره کنید. عدم امکان انتقال یک برنامه به یک سیستم دیگر هم وجود دارد که باید مجدد برنامه را نصب کنید و نکته‌ی نهایی، حذف برنامه که گاهی اوقات حذف کامل نیست و به شکل نامنظم و کثیفی اثراتش را به جا می‌گذارد.

سومین مورد امنیت هست. موقعی که کاربر برنامه‌ای را نصب می‌کند انواع فایل‌ها از شرکت و تولید کننده‌های مختلف روی سیستم نصب می‌شوند. گاهی اوقات برنامه‌ها بعضی از فایل هایشان را از روی اینترنت دریافت می‌کنند و کاربر اصلا متوجه موضوع نمی‌شود و این فایل‌ها می‌توانند هر کاری از حذف فایل از روی سیستم گرفته تا ارسال ایمیل را انجام بدهند که این موارد باعث وحشت کاربرها از نصب یک برنامه‌ی جدید می‌شود که این مورد را با قرار دادن یک سیستم امنیت داخلی با اجازه و عدم اجازه کاربر می‌شود تا حدی رفع کرد.

دات نت فریمورک هم این معضل را به طور عادی در زمینه‌ی DLL hell دارد که در فصل آتی حل آن بررسی خواهد شد. ولی بر خلاف COM، نوع‌های موجود در دات نت نیازی به ذخیره تنظیمات در ریجستری ندارند؛ ولی متاسفانه لینک‌های میانبر هنوز وجود دارند. در زمینه امنیت دات نت شامل یک مدل امنیتی به نام Code Access security می‌باشد؛ از آنجا که امنیت ویندوز بر اساس هویت کاربر تامین می‌شود. code access security به برنامه‌های میزبان مثل sql server اجازه می‌دهد که مجوز مربوطه را خودشان بدهند تا بدین صورت بر اعمال کامپونت‌های بار شده نظارت داشته باشند که البته این مجوزها در حد معمولی و اندک هست. ولی اگر برنامه خود میزبان که به طور محلی روی سیستم نصب می‌شوند، باشد دسترسی کاملب به مجوزها را دارد. پس بدین صورت کاربر این اجازه را دارد که بر آن چیزی که روی سیستم نصب یا اجرا می‌شود، نظارت داشته باشه تا کنترل سیستم به طور کامل در اختیار او باشد.
در قسمت بعدی با نحوه توزیع برنامه آشنا خواهیم شد.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۵/۲۵ ۲۰:۴۱

دات نت فریمورک هم یک معضل بزرگ در زمینه‌ی DLL hell دارد که برای حل مشکل در پخش کردن فایل‌ها در جای جای هارد دیسک راه درازی در پیش است.

GAC به همین منظور تدارک دیده شد. در GAC می‌توان چندین نگارش یک DLL دات نتی را ذخیره کرد، بدون اینکه برنامه‌های مختلف دات نتی با مشکل نصب یا ارتقاء مواجه شوند.

عنوان: **آشنایی با CLR: قسمت یازدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۲۶ ۱۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

انتشار نوع‌ها (Types) به یک ماژول

در این قسمت به نحوه‌ی تبدیل سورس به یک فایل قابل انتشار می‌پردازیم. کد زیر را به عنوان مثال در نظر بگیرید:

```
public sealed class Program {
  public static void Main() {
    System.Console.WriteLine("Hi");
  }
}
```

این کد یک ارجاع به نام کنسول دارد که این ارجاع، داخل فایلی به نام mscorlib.dll قرار دارد. پس برنامه‌ی ما نوعی را دارد، که آن نوع توسط شرکت دیگری پیاده سازی شده است. برای ساخت برنامه‌ی کد بالا، کدها را داخل فایلی با نام program.cs قرار داده و با دستور زیر در خط فرمان آن را کامپایل می‌کنیم:

```
csc.exe /out:Program.exe /t:exe /r:MSCorLib.dll Program.cs
```

کد بالا با سوئیچ اول می‌گوید که فایلی را با نام program.exe درست کن و با سوئیچ دوم می‌گوید که این برنامه از نوع کنسول هست.

موقعیکه کامپایلر فایل سورس را مورد بررسی قرار می‌دهد، متوجه متد writerline می‌گردد؛ ولی از آنجاکه این نوع توسط شما ایجاد نشده است و یک نوع خارجی است، شما باید یک مجموعه از ارجاعات را به کمپایلر داده تا آن نوع را در آن‌ها بیابد. ارائه این ارجاعات به کامپایلر توسط سوئیچ /r که در خط بالا استفاده شده است، صورت می‌گیرد.

mscorlib یک فایل سورس است که شامل همه‌ی نوع‌ها، از قبیل int,string,byte و خیلی از نوع‌های دیگر می‌شود. از آنجائیکه استفاده‌ی از این نوع‌ها به طور مکرر توسط برنامه نویس‌ها صورت می‌گیرد، کمپایلر به طور خودکار این کتابخانه را به لیست ارجاعات اضافه می‌کند. به بیان دیگر خط بالا به شکل زیر هم قابل اجراست:

```
csc.exe /out:Program.exe /t:exe Program.cs
```

به علاوه از آنجایی که بقیه‌ی سوئیچ‌ها هم مقدار پیش فرضی را دارند، خط زیر هم معادل خطوط بالاست:

```
csc.exe Program.cs
```

اگر به هر دلیلی دوست ندارید که سمت mscorlib ارجاعی صورت بگیرید، می‌توانید از دستور زیر استفاده کنید:

```
/nostdlib
```

مایکروسافت موقعی از این سوئیچ بالا استفاده کرده است که خواسته است خود mscorlib را بسازد. با اضافه کردن این سوئیچ، کد مثال که حاوی شیء یا نوع کنسول است به خطا برخواهد خورد چون تعریف آن در mscorlib صورت گرفته و شما با سوئیچ بالا دسترسی به آن را ممنوع اعلام کرده‌اید و از آنجاکه این نوع تعریف نشده، برنامه ازکامپایل بازخواهد ماند.

```
csc.exe /out:Program.exe /t:exe /nostdlib Program.cs
```

بیایید نگاه دقیقتری به فایل program.exe ساخته شده بیندازیم؛ دقیقا این فایل چه نوع فایلی است؟ برای بسیاری از مبتدیان، این یک فایل اجرایی است که در هر دو ماشین 32 و 64 بیتی قابل اجراست. ویندوز از سه نوع برنامه پشتیبانی می‌کند: CUI یا برنامه‌های تحت کنسول، برنامه‌هایی با رابط گرافیکی GUI و برنامه‌های مخصوص windows store که سوئیچ‌های آن به شرح زیر است:

```
//CUI
/t:exe

//GUI
/t:winexe

//Winsows store App
/t:appcontainerexe
```

قبل از اینکه بحث را در مورد سوئیچ‌ها به پایان برسانیم، اجازه دهید در مورد فایل‌های پاسخگو یا response file صحبت کنیم. یک فایل پاسخگو، فایلی است که شامل مجموعه‌ای از سوئیچ‌های خط فرمان می‌شود. موقعی‌که csc.exe اجرا می‌شود، به فایل پاسخگویی که شما به آن معرفی کرده‌اید مراجعه کرده و فرمان را با سوئیچ‌های داخل آن اجرا می‌کند. معرفی یک فایل پاسخگو به کامپایلر توسط علامت @ و سپس نام فایل صورت می‌گیرد و در این فایل هر خط، شامل یک سوئیچ است. مثلا فایل پاسخگوی response.rsp شامل سوئیچ‌های زیر است:

```
/out:MyProject.exe
/target:winexe
```

و برای در نظر گرفتن این سوئیچ‌ها فایل پاسخگو را به کامپایلر معرفی می‌کنیم:

```
csc.exe @MyProject.rsp CodeFile1.cs CodeFile2.cs
```

این فایل خیلی کار شما را راحت می‌کند و نمی‌گذارد در هر بار کامپایل، مرتب سوئیچ‌های آن را وارد کنید و کیفیت کار را بالا می‌برد. همچنین می‌توانید چندین فایل پاسخگو داشته باشید و هر کدام شامل سوئیچ‌های مختلفی تا اگر خواستید تنظیمات کامپایل را تغییر دهید، به راحتی تنها نام فایل پاسخگو را تغییر دهید. همچنین کامپایلر سی شارپ از چندین فایل پاسخگو هم پشتیبانی می‌کند و می‌توانید هر تعداد فایل پاسخگویی را به آن معرفی کنید. در صورتیکه فایل را با نام csc.rsp نامگذاری کرده باشید، نیازی به معرفی آن نیست چرا که کامپایلر در صورت وجود، آن را به طور خودکار خواهد خواند و به این فایل global response file یا فایل پاسخگوی عمومی گویند.
در صورتیکه چندین فایل پاسخگو که به آن فایل‌های محلی local می‌گویند، معرفی کنید که دستورات آن‌ها(سوئیچ) با دستورات داخل csc.rsp مقدار متفاوتی داشته باشند، فایل‌های محلی الویت بالاتری نسبت به فایل global داشته و تنظمیات آن‌ها روی فایل global رونوشت می‌گردند.

موقعی‌که شما دات نت فریمورک را نصب می‌کنید، فایل csc.rsp را با تنظیمات پیش فرض در مسیر زیر نصب می‌کند:

```
%SystemRoot%\
Microsoft.NET\Framework(64)\vX.X.X
```

حروف x نمایانگر نسخه‌ی دات نت فریمورکی هست که شما نصب کرده‌اید. آخرین ورژن از این فایل در زمان نگارش کتاب، شامل سوئیچ‌های زیر بوده است.

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.
# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
```

```
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.DataSetExtensions.dll
/r:System.Data.Linq.dll
/r:System.Data.OracleClient.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceModel.Web.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Extensions.Design.dll
/r:System.Web.Extensions.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.Workflow.Activities.dll
/r:System.Workflow.ComponentModel.dll
/r:System.Workflow.Runtime.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll
```

این فایل حاوی بسیاری از ارجاعات اسمبلی‌هایی است که بیشتر توسط توسعه دهندگان مورد استفاده قرار می‌گیرد و در صورتیکه برنامه‌ی شما به این اسمبلی‌ها محدود می‌گردد، لازم نیست که این اسمبلی‌ها را به کامپایلر معرفی کنید.
البته ارجاع کردن به این اسمبلی‌ها تا حد کمی باعث کند شدن صورت کامپایل می‌شوند؛ ولی تاثیری بر فایل نهایی و نحوه‌ی اجرای آن نمی‌گذارند.

نکته: در صورتی که قصد ارجاعی را دارید، می‌توانید آدرس مستقیم اسمبلی را هم ذکر کنید. ولی اگر تنها به نام اسمبلی اکتفا کنید، مسیرهای زیر جهت یافتن اسمبلی بررسی خواهند شد:
دایرکتوری برنامه
دایرکتوری که شامل فایل csc.exe می‌شود. که خود فایل mscorlib از همانجا خوانده می‌شود و مسیر آن شبیه مسیر زیر است:

```
%SystemRoot%\Microsoft.NET\Framework\v4.0.#####
```

هر دایرکتوری که توسط سوئیچ lib/ مشخص شده باشد.
هر دایرکتوری که توسط متغیر محلی lib مشخص شده باشد.

استفاده از سوئیچ noconfig/ هم باعث می‌شود که فایل‌های پاسخگوی از هر نوعی، چه عمومی و چه محلی، مورد استفاده قرار نگیرند. همچنین شما مجاز هستید که فایل csc.rsp را هم تغییر دهید؛ ولی این نکته را فراموش نکنید، در صورتی که برنامه‌ی شما به سیستمی دیگر منتقل شود، تنظیمات این فایل در آنجا متفاوت خواهد بود و بهتر هست یک فایل محلی را که همراه خودش هست استفاده کنید.
در قسمت بعدی نگاه دیگری بر متادیتا خواهیم داشت.

عنوان: **آشنایی با CLR: قسمت دوازدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۲۷ ۱:۴۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

متادیتاها شامل بلوکی از داده‌های باینری هستند که شامل چندین جدول شده و جدول‌ها نیز به سه دسته تقسیم می‌شوند:

جداول تعاریف Definition Table
جداول ارجاع References Table
جداول manifest

جداول تعریف
جدول زیر تعدادی از جداول تعریف‌ها را توضیح می‌دهد:

ModuleDef	شامل آدرس یا مدخلی است که ماژول در آن تعریف شده است. این آدرس شامل نام ماژول به همراه پسوند آن است؛ بدون ذکر مسیر. در صورتی که کامپایل به صورت GUID انجام گرفته باشد، Version ID ماژول هم همراه آن‌ها خواهد بود. در صورتیکه نام فایل تغییر کند، این جدول باز نام اصلی ماژول را به همراه خواهد داشت. هر چند تغییر نام فایل به شدت رد شده و ممکن است باعث شود CLR نتواند در زمان اجرا آن را پیدا کند.
TypeDef	شامل یک مدخل ورودی برای هر نوعی است که تعریف شده است. هر آدرس ورودی شامل نام نوع ، پرچمها (همان مجوزهای public و private و ...) می‌باشد. همچنین شامل اندیس هایی به متدها است که شامل جدول MethodDef می‌باشند یا فیلدهایی که شامل جدول FieldDef می‌باشند و الی آخر...
MethodDef	شامل آدرسی برای هر متد تعریف شده در ماژول است که شامل نام متد و پرچم هاست. همچنین شامل امضای متد و نقطه‌ی آغاز کد IL آن در ماژول هم می‌شود و آن آدرس هم میتواند ارجاعی به جدول ParamDef جهت شناسایی پارامترها باشد.
FieldDef	شامل اطلاعاتی در مورد فیلدهاست که این اطلاعات ، پرچم، نام و نوع فیلد را مشخص می‌کنند.
ParamDef	حاوی اطلاعات پارامتر متدهاست که این اطلاعات شامل پرچم‌ها (in , out ,retval) ، نوع و نام است.
PropertyDef	برای هر پراپرتی یا خصوصیت، شامل یک آدرس است که شامل نام، نوع و پرچم می‌شود.
EventDef	برای هر رویداد شامل یک آدرس است که این آدرس شامل نام و نوع است.

جداول ارجاعی
موقعی که کد شما کامپایل می‌شود، اگر شما به اسمبلی دیگری ارجاع داشته باشید، از جداول ارجاع کمک گرفته می‌شود که در

جدول زیر تعدادی از این جداول فهرست شده‌اند:

AssemblyRef	شامل آدرس اسمبلی است که ماژولی به آن ارجاع داده است و این آدرس شامل اطلاعات ضروری جهت اتصال به اسمبلی می‌شود و این اطلاعات شامل نام اسمبلی (بدون ذکر پسوند و مسیر)، شماره نسخه اسمبلی، سیستم فرهنگی و منطقه‌ای تعیین شده اسمبلی culture و یک کلید عمومی که عموما توسط ناشر ایجاد می‌گردد که هویت ناشر آن اسمبلی را مشخص می‌کند. هر آدرس شامل یک پرچم و یک کد هش هست که بری ارزیابی از صحت و بی خطا بودن بیت‌های اسمبلی ارجاع شده Checksum استفاده می‌شود.
ModuleRef	شامل یک آدرس ورودی به هدر PE ماژول است به نوع‌های پیاده سازی شده آن ماژول در آن اسمبلی. هر آدرس شامل نام فایل و پسوند آن بدون ذکر مسیر است. این جدول برای اتصال به نوع‌هایی استفاده می‌شود که در یک ماژول متفاوت از ماژول اسمبلی صدا زده شده پیاده سازی شده است.
TypeRef	شامل یک آدرس یا ورودی برای هر نوعی است که توسط ماژول ارجاع داده شده است. هر آدرس شامل نام نوع و آدرسی است که نوع در آن جا قرار دارد. اگر این نوع داخل نوع دیگری پیاده سازی شود، ارجاعات به سمت یک جدول TypeDef خواهد بود. اگر نوع داخل همان ماژول تعریف شده باشد، ارجاع به سمت جدول ModuleDef خواهد بود و اگر نوع در ماژول دیگری از آن اسمبلی پیاده سازی شده باشد، ارجاع به سمت یک جدول ModuleRef خواهد بود و اگر نوع در یک اسمبلی جداگانه تعریف شده باشد، ارجاع به جدول AssemblyRef خواهد بود.
MemberRef	شامل یک آدرس ورودی برای هر عضو (فیلد و متدها و حتی پراپرتی و رویدادها) است که توسط آن آن ماژول ارجاع شده باشد. هر آدرس شامل نام عضو، امضاء و یک اشاره‌گر به جدول TypeRef است، برای نوع‌هایی که به تعریف عضو پرداخته‌اند.

البته جداولی که در بالا هستند فقط تعدادی از آن جداول هستند، ولی قصد ما تنها یک آشنایی کلی با جداول هر قسمت بود. در مورد جداول manifest بعدتر صحبت می‌کنیم.
ابزارهای متنوع و زیادی هستند که برای بررسی و آزمایش متادیتاها استفاده می‌شوند. یکی از این ابزارها ILDasm.exe می‌باشد. برای دیدن متادیتاهای یک فایل اجرایی فرمان زیر را صادر کنید:

```
ILDasm Program.exe
```

صدور فرمان بالا باعث اجرای ILDasm و بارگزاری اسمبلی‌های program.exe می‌شود. برای مشاهده‌ی اطلاعات جداول متا به صورت شکیل و قابل خواندن برای انسان، در منوی برنامه، مسیر زیر را طی کنید:

```
View/MetaInfo/Show
```

با طی کردن گزینه‌های بالا، اطلاعات به صورت زیر نمایش داده می‌شوند:

```
==========================================================
ScopeName : Program.exe
```

```
MVID : {CA73FFE8-0D42-4610-A8D3-9276195C35AA}
============================================================
Global functions
------------------------------------------------------------
Global fields
------------------------------------------------------------
Global MemberRefs
------------------------------------------------------------
TypeDef #1 (02000002)
------------------------------------------------------------
TypDefName: Program (02000002)
Flags : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
[BeforeFieldInit] (00100101)
Extends : 01000001 [TypeRef] System.Object
Method #1 (06000001) [ENTRYPOINT]
------------------------------------------------------------
MethodName: Main (06000001)
Flags : [Public] [Static] [HideBySig] [ReuseSlot] (00000096)
RVA : 0x00002050
ImplFlags : [IL] [Managed] (00000000)
CallCnvntn: [DEFAULT]
ReturnType: Void
No arguments.
Method #2 (06000002)
------------------------------------------------------------
MethodName: .ctor (06000002)
Flags : [Public] [HideBySig] [ReuseSlot] [SpecialName]
[RTSpecialName] [.ctor] (00001886)
RVA : 0x0000205c
ImplFlags : [IL] [Managed] (00000000)
CallCnvntn: [DEFAULT]
hasThis
ReturnType: Void
No arguments.
TypeRef #1 (01000001)
------------------------------------------------------------
Token: 0x01000001
ResolutionScope: 0x23000001
TypeRefName: System.Object
MemberRef #1 (0a000004)
------------------------------------------------------------
Member: (0a000004) .ctor:
CallCnvntn: [DEFAULT]
hasThis
ReturnType: Void
No arguments.
TypeRef #2 (01000002)
------------------------------------------------------------
Token: 0x01000002
ResolutionScope: 0x23000001
TypeRefName: System.Runtime.CompilerServices.CompilationRelaxationsAttribute
MemberRef #1 (0a000001)
------------------------------------------------------------
Member: (0a000001) .ctor:
CallCnvntn: [DEFAULT]
hasThis
ReturnType: Void
1 Arguments
Argument #1: I4
TypeRef #3 (01000003)
------------------------------------------------------------
Token: 0x01000003
ResolutionScope: 0x23000001
TypeRefName: System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
MemberRef #1 (0a000002)
------------------------------------------------------------
Member: (0a000002) .ctor:
CallCnvntn: [DEFAULT]
hasThis
ReturnType: Void
No arguments.
TypeRef #4 (01000004)
------------------------------------------------------------
Token: 0x01000004
ResolutionScope: 0x23000001
TypeRefName: System.Console
MemberRef #1 (0a000003)
------------------------------------------------------------
Member: (0a000003) WriteLine:
CallCnvntn: [DEFAULT]
ReturnType: Void
```

```
1 Arguments
Argument #1: String
Assembly
-------------------------------------------------------
Token: 0x20000001
Name : Program
Public Key :
Hash Algorithm : 0x00008004
Version: 0.0.0.0
Major Version: 0x00000000
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [none] (00000000)
CustomAttribute #1 (0c000001)
-------------------------------------------------------
CustomAttribute Type: 0a000001
CustomAttributeName:
System.Runtime.CompilerServices.CompilationRelaxationsAttribute ::
instance void .ctor(int32)
Length: 8
Value : 01 00 08 00 00 00 00 00 > <
ctor args: (8)
CustomAttribute #2 (0c000002)
-------------------------------------------------------
CustomAttribute Type: 0a000002
CustomAttributeName: System.Runtime.CompilerServices.RuntimeCompatibilityAttribute ::
instance void .ctor()
Length: 30
Value : 01 00 01 00 54 02 16 57 72 61 70 4e 6f 6e 45 78 > T WrapNonEx<
: 63 65 70 74 69 6f 6e 54 68 72 6f 77 73 01 >ceptionThrows <
ctor args: ()
AssemblyRef #1 (23000001)
-------------------------------------------------------
Token: 0x23000001
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: mscorlib
Version: 4.0.0.0
Major Version: 0x00000004
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
User Strings
-------------------------------------------------------
70000001 : ( 2) L"Hi"
Coff symbol name overhead: 0
```

لازم نیست که تمامی اطلاعات بالا را به طور کامل بفهمید. همین که متوجه شوید برنامه شامل TypeDef است که نام آن Program است و این نوع به صورت یک کلاس عمومی sealed است که از نوع system.object ارث بری کرده است (یک نوع ارجاع از اسمبلی دیگر) و برنامه شامل دو متد main و یک سازنده ctor. است، کافی هست.

متد Main یک متد عمومی و ایستا static است که شامل کد IL است و هیچ خروجی ندارد و هیچ آرگومانی را نمی‌پذیرد. متد سازنده عمومی است و شامل کد IL است، سازنده هیچ نوع خروجی ندارد و هیچ آرگومانی هم نمی‌پذیرد و یک اشاره‌گر که به یک object در حافظه که موقع صدا زدن ساخته خواهد شد.

ابزار ILDasm امکاناتی بیشتری از آنچه که دیدید ارائه می‌کند. به عنوان نمونه اگر مسیر زیر را در منوها طی کنید:

```
View/statistics
```

اطلاعات آماری زیر نمایش داده می‌شود:

```
File size : 3584
PE header size : 512 (496 used) (14.29%)
PE additional info : 1411 (39.37%)
Num.of PE sections : 3
CLR header size : 72 ( 2.01%)
CLR meta-data size : 612 (17.08%)
CLR additional info : 0 ( 0.00%)
CLR method headers : 2 ( 0.06%)
```

```
Managed code : 20 ( 0.56%)
Data : 2048 (57.14%)
Unaccounted : -1093 (-30.50%)
Num.of PE sections : 3
.text - 1024
.rsrc - 1536
.reloc - 512
CLR meta-data size : 612
Module - 1 (10 bytes)
TypeDef - 2 (28 bytes) 0 interfaces, 0 explicit layout
TypeRef - 4 (24 bytes)
MethodDef - 2 (28 bytes) 0 abstract, 0 native, 2 bodies
MemberRef - 4 (24 bytes)
CustomAttribute- 2 (12 bytes)
Assembly - 1 (22 bytes)
AssemblyRef - 1 (20 bytes)
Strings - 184 bytes
Blobs - 68 bytes
UserStrings - 8 bytes
Guids - 16 bytes
Uncategorized - 168 bytes
CLR method headers : 2
Num.of method bodies - 2
Num.of fat headers - 0
Num.of tiny headers - 2
Managed code : 20
Ave method size - 10
```

اطلاعات بالا شامل نمایش حجم فایل به بایت و سایر قسمت‌های تشکیل دهنده فایل است...

توجه: ILDasm یک باگ دارد که بر نمایش اندازه‌ی فایل تاثیر می‌گذارد و باعث می‌شود شما نتوانید به اطلاعات ثبت شده اعتماد داشته باشید.

عنوان: **آشنایی با CLR: قسمت سیزدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۲۸ ۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

ترکیب ماژول‌ها به قالب یک اسمبلی

فایل Program.exe یک فایل PE با جداول متادیتا است که همچنین یک اسمبلی هم می‌باشد. یک اسمبلی مجموعه‌ای از یک یا چند فایل، شامل تعاریف نوع و منابع (ریسورس) می‌باشد و یکی از فایل‌های اسمبلی، برای نگهداری manifest انتخاب می‌شود. این جدول مجموعه‌ای است از جداول متادیتا که به طور کلی شامل نام فایل‌هایی است که قسمتی از اسمبلی را تشکیل می‌دهند. برای همین گفتیم که CLR با اسمبلی‌ها کار می‌کند. ابتدا جداول manifest را خوانده تا نام فایل‌ها را شناسایی کرده تا از آن‌ها را به حافظه بارگزاری کند. اسمبلی‌ها چند خصوصیت دارند که باید آن‌ها را بدانید:

- نوع‌های با قابلیت استفاده‌ی مجدد را تعریف می‌کنند.
- داری شماره‌ی نسخه version هستند.
- می‌توانند شامل اطلاعات امنیتی باشند.

این خواصی است که یک اسمبلی به همراه دارد و فایل‌هایی که شامل می‌شود، نمی‌توانند چنین خاصیتی را داشته باشند؛ مگر اینکه آن فایل‌ها در متای خود جدول manifest داشته باشند.

شما برای بسته بندی، شماره نسخه، مباحث امنیتی و استفاده از نوع‌ها، باید آن‌ها را داخل ماژولی قرار دهید که جزئی از اسمبلی است. یک فایل اسمبلی همانند program.exe به عنوان یک فایل واحد شناخته می‌شود. با اینکه یک اسمبلی از چند فایل تشکیل می‌شود، فایل‌های PE به همراه جداول متادیتای آن و تعدادی ریسورس مثل فایل‌های gif و jpg است که به شما کمک می‌کند به همه‌ی آن‌ها به عنوان یک فایل منطقی EXE یا dll نگاه کنید.

یکی از دلایلی که در قسمت سوم گفتیم این بود که می‌توانیم فایل‌هایی را که به ندرت استفاده می‌شوند، از طریق اینترنت مورد استفاده قرار دهیم. در حالتیکه نیاز به دسترسی به اسمبلی‌های روی اینترنت دارید، CLR ابتدا کش را بررسی می‌کند تا آیا فایل حاضر است یا خیر؟ اگر پاسخ مثبت بود، در حافظه قرار می‌گیرد. ولی اگر پاسخ منفی بود، CLR به آدرسی که اسمبلی در آن قرار دارد، رجوع کرده و آن را دانلود می‌کند و اگر فایل مد نظر یافت نشد، استثنای FileNotFound را در حین اجرا صادر خواهد کرد.

آقای جفری ریچر در کتاب خود سه تا از دلایل استفاده‌ی از اسمبلی‌های چند فایله را بر می‌شمارد:
جداسازی نوع‌ها در فایل‌های جداگانه که باعث کاهش حجم فایل از طریق اینترنت و بارگزاری حجم کمتر در حافظه می‌شوند.

استفاده از فایل‌های منبع و داده‌ها در اسمبلی: فرض کنید نیاز به محاسبه‌ی اطلاعات بیمه دارید و برای این کار به اطلاعات داخل یک جدول آماری احتیاج دارید. این جدول آماری می‌تواند یک فایل متنی ساده یا یک صفحه‌ی گسترده مثل اکسل یا در قالب ورد و هر چیز دیگری باشد که به جای embed شدن این جداول در سورس کد برنامه، آن‌ها را با استفاده از ابزاری مثل Assembly Linker -AL.exe می‌توانید جزئی از اسمبلی کنید و فقط نیاز است که بدانید چگونه آن فایل را پارس یا تبدیل کنید.

استفاده از انواع ایجاد شده در زبان‌های مختلف. در این حالت شما مقداری از کد را با استفاده از C# نوشته اید و مقداری از آن را با Visual Basic می‌نویسید و هر کدام در نهایت به یک ماژول جداگانه کامپایل خواهند شد. ولی تبدیل آن به یک واحد منطقی مثل اسمبلی ممکن است و از این نظر می‌توانید روی ماژول‌های یک دسته کنترل داشته باشید.

اگر چندین نوع دارید که شامل نسخه بندی و تنظیمات امنیتی مشترک هستند، بهتر است در یک اسمبلی قرار گیرند تا اینکه در اسمبلی‌های جداگانه‌ایی قرار بگیرند. دلیل این کار هم ایجاد performance یا کارآیی بهتر است. بارگذاری یک اسمبلی در حافظه زمانی را برای یافتن آن از CLR و ویندوز می‌گیرد و سپس وارد بارگیری آن‌ها در حافظه و آماده سازی می‌شود. پس هر چه تعداد اسمبلی‌ها کمتر باشد، کارآیی بهتری خواهید داشت، چون کمتر شدن بارگیری برابر با کاهش صفحات کاری است و پراکندگی fragmentation فضای آدرس دهی آن فرایند را کاهش خواهد داد. نهایتا Ngen می‌تواند در بهینه سازی فایل‌های بزرگتر موفق باشد.

برای ساخت اسمبلی، باید یکی از فایل‌های PE را برای نگهداری جدول manifest انتخاب کنید؛ یا خودتان یک فایل PE جدا درست کنید که تنها شامل جدول مانیفست شود. جدول زیر قالبی از جداول مانیفست هست که بابت ماژول‌های اضافه شده به یک

اسمبلی ایجاد می‌شوند.

AssemblyDef	شامل مدخل ورودی (آدرس شروع حافظه) برای اسمبلی‌هایی است که ماژول عضو آن است. این مدخل شامل نام اسمبلی (بدون مسیر و پسوند)، شماره نسخه یا ورژن، culture، فلگ، الگوریتم هش و کلید عمومی ناشر، که می‌تواند نال باشد، هست.
FileDef	شامل یک مدخل ورودی برای هر فایل PE و فایل‌های ریسورسی است که قسمتی از اسمبلی را تشکیل می‌دهند. این مدخل ورودی شامل نام و پسوند فایل (بدون ذکر مسیر)، فلگ و مقدار هش می‌شود. اگر تنها یک اسمبلی وجود داشته باشد، این جدول هیچ مدخلی نخواهد داشت.
ManifestResourceDef	شامل یک مدخل ورودی برای هر فایل ریسورس است. این مدخل شامل نام فایل ریسورس، فلگ و یک اندیس به جدول FileDef است که در آن اشاره‌ای به آن فایل ریسورس یا استریم است.
ExportedTypesDef	شامل یک مدخل ورودی برای هر نوع عمومی است که از همه ماژول‌های PE استخراج شده است. هر مدخل شامل نام نوع و اندیسی به جدول FileDef و یک اندیس دیگر به جدول TypeDef است. نکته: برای ذخیره سازی حافظه و کم حجم شدن فایل‌ها، نوع‌های استخراج شده از فایلی که شامل مانیفست است دیگر در جدول جاری نام نوع‌ها ذکر نمی‌گردد؛ چرا که این اطلاعات در جدول TypeDef اسمبلی جاری موجود است.

نکته: اسمبلی که شامل مانیفست است، شامل یک جدول AssemblyRef نیز می‌گردد که به تمام اسمبلی‌های ارجاع شده در آن اسمبلی اشاره می‌کند. با استفاده از ابزارهای موجود می‌توان اسمبلی مدنظر را باز کرده و به این ترتیب لیستی از اسمبلی‌های ارجاع شده را خواهید دید و بدین صورت این اسمبلی یک اسمبلی خود تعریف می‌شود.

کامپایلر سی شارپ با استفاده از سوئیچ‌های زیر یک اسمبلی را تولید می‌کند:

```
/t[arget]:exe, /t[arget]:winexe, /t[arget]: appcontainerexe, /t[arget]: library, or /t[arget]:winmdobj
```

سوئیچ‌های بالا باعث می‌شود که یک فایل PE با جدول مانیفست تولید گردد. در صورتیکه سوئیچ زیر را به کار ببرید، فایل تولید شده شامل جدول مانیفست نمی‌شود.

```
/t[arget]:module
```

این فایل PE تولید شده در قالب یک dll است که باید قبل از اینکه CLR به نوع‌های داخل آن دسترسی پیدا کند، به یک اسمبلی اضافه گردد. موقعی‌که شما از سوئیچ بالا استفاده می‌کنید، کامپایلر سی شارپ به طور پیش فرض از پسوند netmodule برای فایل خروجی استفاده می‌کند.

نکته‌ی پایانی: محیط توسعه ویژوال استادیو به طور پیش فرض از اسمبلی‌های چند فایل پشتیبانی نمی‌کند، اگر میخواهید که اسمبلی‌های چند فایله تولید کنید باید در سوئیچ‌های مورد استفاده آن تجدید نظری داشته باشید.

در مقاله آینده این روش‌ها را بررسی خواهیم کرد...

عنوان: **آشنایی با CLR: قسمت چهاردهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۲۹ ۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

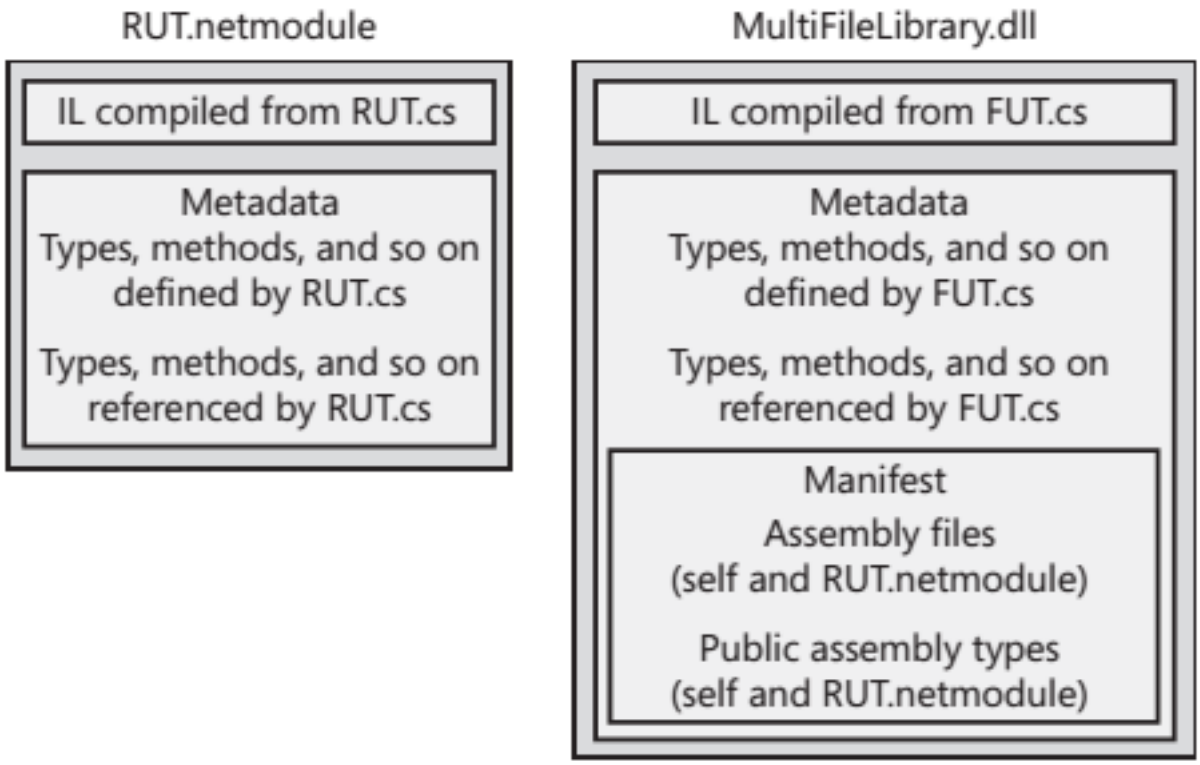
در ادامه قسمت قبلی روش‌های زیادی جهت اضافه شدن یک ماژول به یک اسمبلی وجود دارند. اگر شما از کامپایلر سی‌شارپ برای ساخت یک فایل PE با جدول مانیفست استفاده می‌کنید، می‌توانید از سوئیچ /AddModule استفاده کنید. برای اینکه بدانیم چگونه می‌توان یک اسمبلی چند فایله ساخت بیاید فرض کنیم که دو فایل سورس کد با مشخصات زیر داریم: RUT.cs: این سورس شامل کدهایی است که به ندرت در برنامه استفاده می‌شود. FUT.cs: این سورس شامل کدهایی است که به طور مکرر مورد استفاده قرار می‌گیرد.
ابتدا به صورت زیر کد سورسی را که به ندرت استفاده می‌شود، به عنوان یک ماژول جداگانه کامپایل می‌کنم:

```
csc /t:module RUT.cs
```

اجرای این خط سبب ایجاد یک فایل به نام RUT.netmodule می‌گردد که یک DLL استاندارد است؛ ولی CLR به تنهایی توانایی بارگیری آن را ندارد. دفعه‌ی بعد سورس کدی را که مکرر استفاده می‌شود، به صورت یک ماژول کامپایل می‌کنیم و از آنجائیکه این ماژول استفاده‌ی زیادی دارد، آن را نگهدارنده‌ی جدول مانیفست معرفی می‌کنیم و به این دلیل که این ماژول نماینده‌ی کل اسمبلی است، نام خروجی آن را به جای FUT.dll به MultiFileLibrary.dll تغییر می‌دهیم:

```
csc /out:MultiFileLibrary.dll /t:library /addmodule:RUT.netmodule FUT.cs
```

خط بالا به علت سوئیچ \t:library فایل MultiFileLibrary.dll را ایجاد می‌کند. این فایل شامل جدول متادیتای مانیفست می‌شود و سوئیچ به آن می‌گوید که باید ماژول RUT.netmodule را جزئی از اسمبلی بداند. این سوئیچ به کامپایلر اعلام می‌کند که ارجاع این فایل در جدول FileDef و ExportedTypesDef ثبت شود. بعد از اتمام عملیات کامپایل، مطابق شکل زیر دو فایل ایجاد می‌شود که فایل سمت راست شامل جدول مانیفست است. فایل RUT.netmodule شامل کد IL و جداول متادیتاهای مربوط به خواص و رویدادها و مواردی از این قبیل است که در این ماژول یافت می‌شود. فایل بعدی MultiFileLibrary.dll هست که شامل کد IL کد FUT.CS می‌شوذ بعلاوه جداول متادیتا مثل ماژول قبلی و جدول متادیتای مانیفست که باعث می‌شود به عنوان یک اسمبلی شناخته شود.



البته توجه داشته باشید که جدول مانیفست ارجاعی به نوع‌های عمومی استخراج شده داخل فایل خودش ندارد، زیرا که در جداول اختصاصی خودش موجود است و در ذخیره سازی صرفه جویی می‌گردد.
بعد از اینکه MultiFileLibrary.dll ساخته شد، به منظور آزمایش کردن جداول متادیتا می‌توانید از ابزار ILDasm.exe استفاده کنید تا ارجاع به فایل RUT.netmodule به شما ثابت شود. آنچه در زیر می‌بینید نمایی از جداول FileDef و ExportedTypesDef است:

```
File #1 (26000001)
-------------------------------------------------------
Token: 0x26000001
Name : RUT.netmodule
HashValue Blob : e6 e6 df 62 2c a1 2c 59 97 65 0f 21 44 10 15 96 f2 7e db c2
Flags : [ContainsMetaData] (00000000)


ExportedType #1 (27000001)
-------------------------------------------------------
Token: 0x27000001
Name: ARarelyUsedType
Implementation token: 0x26000001
TypeDef token: 0x02000002
Flags : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
[BeforeFieldInit](00100101)
```

همانطور که در بالا می‌بینید فایل RUT.netmodule با شناسه‌ی (توکن) 0x26000001 به عنوان بخشی از اسمبلی شناخته می‌شود و به نوع کد IL آن اشاره می‌کند.

قابل توجه افراد کنجکاو: توکن‌های جداول متا، مقادیر 4 بایتی است که بایت پر ارزش آن اشاره می‌کند که برای یافتن آن باید به چه جدولی ارجاع کرد. مقادیر زیر این نکته را روشن می‌کند که هر کد ابتدایی به چه جدولی اشاره می‌کند:

TypeRef	**0x01**
TypeDef	**0x02**
AssemblyRef	**0x23**
File *file definition*	**0x26**
ExportedType	**0x27**

برای دیدن لیست کاملی از این کدها فایل Corhdr.h را که به همراه فریم ورک دات نت نصب می‌شود، مطالعه فرمایید. سه بایت باقیمانده هم بر اساس جدولی که به آن ارجاع شده است مشخص می‌گردد؛ مثلا در مثال بالا کد 0x26000001 به اولین سطر جدول File اشاره می‌کند. برای اکثر جدول‌ها شماره گذاری سطرها از عدد 1 آغاز می‌شود نه صفر یا برای برای جداول TypeDef عموما از عدد 2 آغاز می‌شود.
برای اجرای اسمبلی، کامپایلر نیاز دارد که همه‌ی فایل‌های اسمبلی، نصب شده و قابل دسترس باشند و در صورتیکه شما فایل RUT.netmodule را حذف کنید کامپایلر سی شارپ خطای زیر را صادر می‌کند:

```
fatal error CS0009: Metadata file 'C:\ MultiFileLibrary.dll' could not be opened—'Error importing
module 'RUT.netmodule' of assembly 'C:\ MultiFileLibrary.dll'—The system cannot find the file
specified'
```

و این خطا بدین معنی است که برای ساخت اسمبلی باید تمامی فایل‌ها حاضر و مهیا باشند. هر کد کلاینتی که اجرا می‌شود آن متد را صدا می‌زنند. موقعی که یک متد برای اولین بار فراخوانی می‌شود، CLR عملیات شناسایی جهت شناسایی ارجاعات آن در

پارامترها، نوع خروجی متد و متغیرهای محلی آن اجرا می‌کند. سپس تلاش می‌کند تا فایل اسمبلی ارجاع شده را که شامل مانیفست هست، بار کند. اگر نوعی که لازم داریم در همین فایل متد وجود داشته باشد، اجرای عملیات را به سمت آن آغاز می‌کند ولی اگر جدول مانیفست ارجاع را به فایل دیگری بدهد، آن فایل در حافظه بار شده و سپس آن نوع را در دسترس قرار می‌دهد. خطوط بالا این نکته را روشن می‌کند که فایل‌های اسمبلی را تنها موقعی در حافظه بار میکند که ارجاعی از نوع موجود در آن صدا زده شده باشد؛ یعنی اینکه در زمان اجرای برنامه، لازم نیست که همه‌ی فایل‌ها حاضر و مهیا باشند.

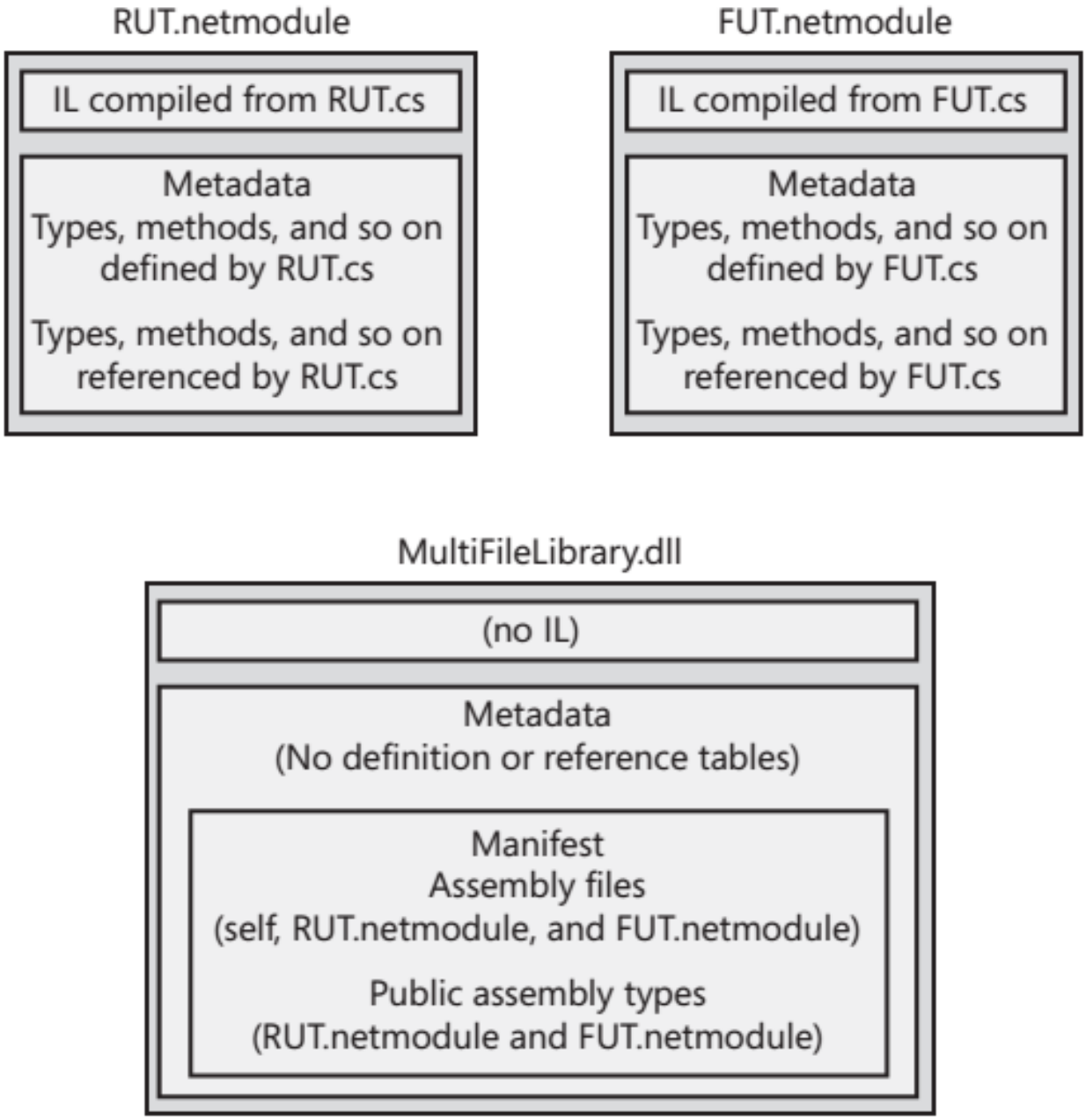
عنوان: **آشنایی با CLR: قسمت پانزدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۳۰ ۱:۵۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در قسمت قبلی نحوه‌ی ساخت اسمبلی را یاد گرفتیم. ولی ممکن است که بخواهید اسمبلی را از طریق Assembly Linker یا AL.exe ایجاد کنید. این روش موقعی سودمند است که بخواهید یک اسمبلی از ماژولها از کامپایلرهای مختلف را ایجاد کنید یا اینکه کامپایلر شما مانند کامپایلر سی شارپ از دستور یا سوئیچی مشابه addmodule استفاده نمی‌کند. یا حتی اینکه در زمان کامپایل هنوز اطلاعاتی از نیازمندی‌های اسمبلی‌ها ندارید و به بعد موکول می‌کنید. از AL همچنین می‌توانید در زمینه‌ی ساخت اسمبلی‌های فقط ریسورس هم استفاده کنید که می‌تواند جهت انجام localization به کار رود. AL می‌تواند یک فایل dll یا exe تولید کند که شامل یک فایل manifest بوده که اشاره به ماژول‌های تشکیل دهنده‌اش دارد.

نحوه‌ی ساخت اسمبلی با استفاده از ابزار AL :

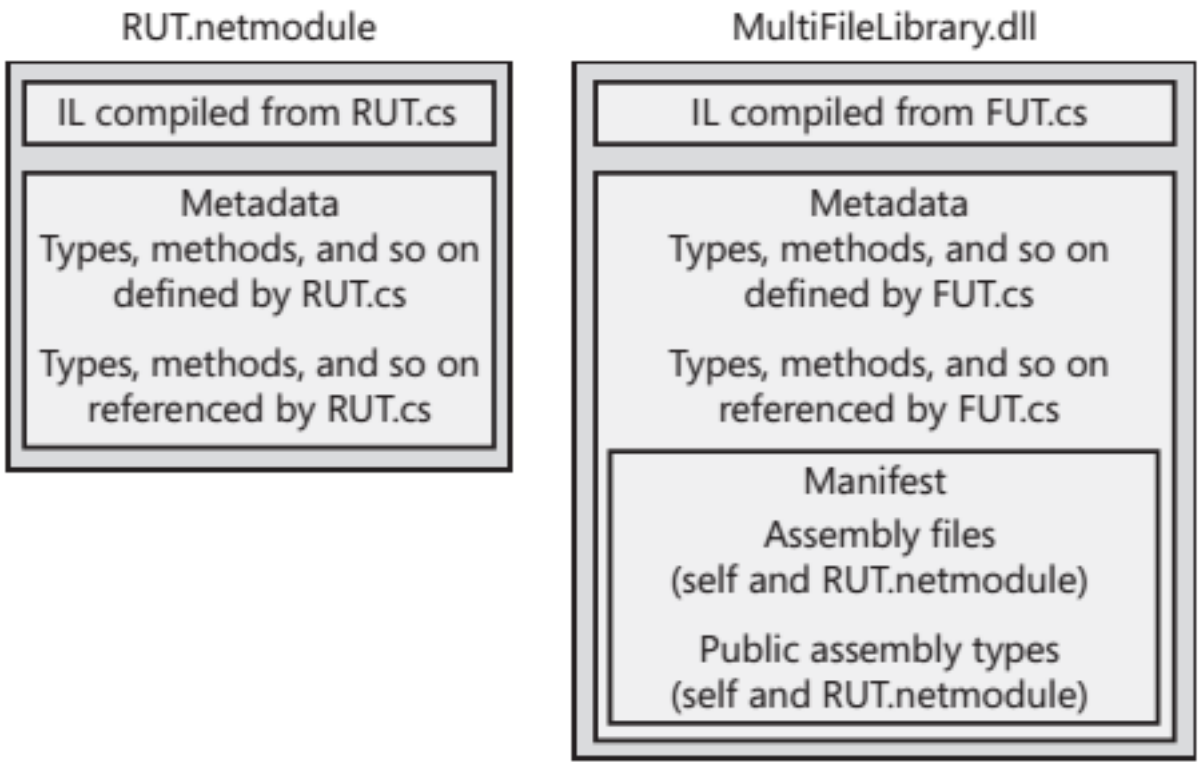
```
csc /t:module RUT.cs
csc /t:module FUT.cs
al /out: MultiFileLibrary.dll /t:library FUT.netmodule RUT.netmodule
```

تصویر زیر نتیجه‌ی دستور بالاست:



در این مثال ما دو ماژول جدا به نام‌های RUT.netmodule و FUT.netmodule را در یک اسمبلی ایجاد کرده‌ایم. داخل این اسمبلی‌ها جدول متادیتا یا بخش IL از ماژول‌ها به چشم نمی‌خورد. به این معنی که کد IL و جداول مربوطه به آن، هر کدام داخل ماژول یا فایل خودش بوده و در اسمبلی کدی وجود ندارد و تنها یک جدول مانیفست جهت شناسایی ماژول‌هایش دارد. شکل بالا گویای اطلاعات داخلی اسمبلی است که می‌توانید با تصویری که در قسمت قبلی درج شده مقایسه کنید.

تصویر قسمت قبلی جهت مقایسه:



در این حالت سه فایل تشکیل شده است که یکی از آن‌ها FUT.netmodule ، MultiFileLibrary.dll و RUT.netmodule است و در استفاده از این ابزار هیچ راهی برای داشتن یک تک فایل وجود ندارد.
این ابزار همچنین می‌تواند فایل‌های GUI, CUI و ... را با سوئیچ‌های زیر هم تولید کند:

```
/t[arget]:exe, /t[arget]:winexe, or /t[arget]:appcontainerexe
```

البته اینکار تا حدی غیر معمول است که یک فایل exe بخواهد کدهای IL ابتدایی را از ماژول‌های جداگانه بخواند. در صورتیکه چنین قصدی را دارید، باید یکی از ماژول‌ها را به عنوان مدخل ورودی Main تعریف کنید تا برنامه از آنجا آغاز به کار کند. نحوه‌ی ساخت یک فایل اجرایی و معرفی ماژول Main به شکل زیر است:

```
csc /t:module /r:MultiFileLibrary.dll Program.cs
al /out:Program.exe /t:exe /main:Program.Main Program.netmodule
```

در اولین خط مانند سابق فایل netmodule تهیه می‌گردد و در خط دوم، داخل اسمبلی قرار می‌گیرد. ولی به علت استفاده از سوئیچ main یک تابع عمومی global به نام __EntryPoint هم تعریف می‌گردد که کد IL آن به شرح زیر است:

```
.method privatescope static void __EntryPoint$PST06000001() cil managed
{
.entrypoint
// Code size 8 (0x8)
.maxstack 8
IL_0000: tail.
IL_0002: call void [.module 'Program.netmodule']Program::Main()
IL_0007: ret
} // end of method 'Global Functions'::__EntryPoint
```

کد بالا یک کد ساده است که می‌گوید داخل فایل Program.netmodule در نوع Program متدی وجود دارد به نام Main که محل آغازین برنامه است. البته این روش ایجاد فایل‌های EXE، بدین شکل توصیه چندانی نمی‌شود و ذکر این مطلب فقط اطلاع از وجود

چنین قابلیتی بود.

عنوان: **آشنایی با CLR: قسمت شانزدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۳۱ ۱۰:۳۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در مقاله قبلی بحث Assembly Linker را باز کردیم و یاد گرفتیم که چگونه می‌توان با استفاده از آن ماژول‌های مختلف را به یک اسمبلی اضافه کرد. در این قسمت از این سلسله مقالات قصد داریم فایل‌های منابع (Resource) مانند مواد چندرسانه‌ای، چند زبانه و .. را به آن اضافه کنیم. یک اسمبلی حتی میتواند تنها Resource باشد.

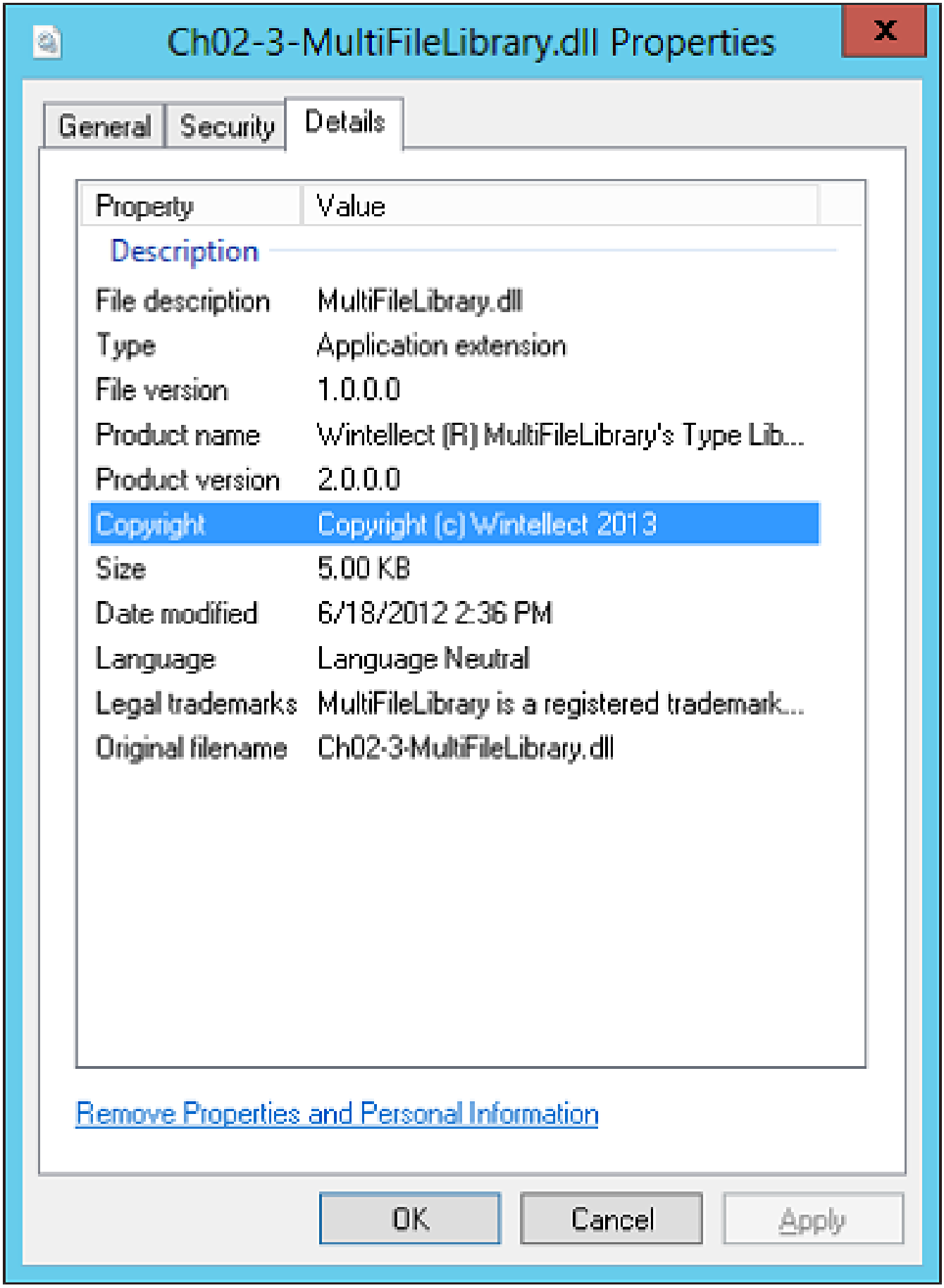
برای اضافه کردن یک فایل به عنوان منبع، از سوئیچ [embed[resource استفاده می‌شود. این سوئیچ محتوای هر نوع فایلی را که به آن پاس شود، به فایل PE اجرایی انتقال داده و جدول ManifestResourceDef را به روز می‌کند تا سیستم از وجود آن آگاه شود. سوئیچ [link[Resource هم برای الحاق کردن یک فایل به اسمبلی به کار می‌رود و دو جدول ManifestResourceDef و FileDef را جهت معرفی منبع جدید و شناسایی فایل اسمبلی که حاوی این منبع است، به روز می‌کند. در این حالت فایل منبع embed نشده و باید در کنار پروژه منتشر شود.
csc هم قابلیت‌های مشابهی را با استفاده از سوئیچ‌های resource/ و link/ دارد و به روز رسانی و دیگر اطلاعات تکمیلی آن مشابه موارد بالاست.

شما حتی می‌توانید منابع یک فایل win32 را خیلی راحت و آسان به اسمبلی معرفی کنید. شما به آسانی می‌توانید مسیر یک فایل res. را با استفاده از سوئیچ win32res/ در al یا csc مشخص کنید. یا برای embed کردن آیکن یک برنامه win32 از سوئیچ win32icon/ مسیر یک فایل ICO را مشخص کنید. در ویژوال استودیو این‌کار به صورت ویژوالی در پنجره تنظیمات پروژه و برگه‌ی Application امکان پذیر است. دلیل اصلی که آیکن برنامه‌ها به صورت embed ذخیره می‌شوند این است که این آیکن برای فایل اجرایی یک برنامه‌ی مدیریت شده هم به کار می‌رود.

فایل‌های اسمبلی Win32 شامل یک فایل مانیفست اطلاعاتی هستند که به طور خودکار توسط کمپایلر سی شارپ تولید می‌گردند. با استفاده از سوئیچ nowin32manifest/ میتوان از ایجاد این نوع فایل جلوگیری کرد. این اطلاعات به طور پیش فرض شبیه زیر است:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
<security>
<requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
<requestedExecutionLevel level="asInvoker" uiAccess="false"/>
</requestedPrivileges>
</security>
</trustInfo>
</assembly>
```

موقعیکه AL یا CSC یک فایل نهایی PE را ایجاد می‌کند، یک منبع نسخه بندی شده با استاندارد win32 نیز به آن Embed می‌شود که با راست کلیک روی فایل و انتخاب گزینه‌ی Properties و برگه‌ی Details این اطلاعات نمایش می‌یابد. در کدنویسی این اپلیکیشن هم می‌توانید از طریق فضای نام system.Diagnostics.FileVersionInfo و متد ایستای آن GetVersionInfo که *پارامتر ورودی آن مسیر فایل اسمبلی است* هم به این اطلاعات، در حین اجرای برنامه دست پیدا کنید.


Ch02-3-MultiFileLibrary.dll Properties
General
Security
Details
Property
Value
Description
File description
MultiFileLibrary.dll
Type
Application extension
File version
1.0.0.0
Product name
Wintellect (R) MultiFileLibrary's Type Lib...
Product version
2.0.0.0
Copyright
Copyright (c) Wintellect 2013
Size
5.00 KB
Date modified
6/18/2012 2:36 PM
Language
Language Neutral
Legal trademarks
MultiFileLibrary is a registered trademark...
Original filename
Ch02-3-MultiFileLibrary.dll
Remove Properties and Personal Information
OK
Cancel
Apply

موقعیکه شما یک اسمبلی می‌سازید باید فیلدهای منبع نسخه بندی را هم ذکر کنید. اینکار توسط خصوصیت‌ها (Attributes) در سطح کد انجام می‌گیرد. این خصوصیات شامل موارد زیر هستند که در فضای نام Reflection قرار گرفته‌اند.

```
using System.Reflection;

// FileDescription version information:
[assembly: AssemblyTitle("MultiFileLibrary.dll")]

// Comments version information:
[assembly: AssemblyDescription("This assembly contains MultiFileLibrary's types")]

// CompanyName version information:
[assembly: AssemblyCompany("Wintellect")]

// ProductName version information:
[assembly: AssemblyProduct("Wintellect (R) MultiFileLibrary's Type Library")]

// LegalCopyright version information:
[assembly: AssemblyCopyright("Copyright (c) Wintellect 2013")]

// LegalTrademarks version information:
[assembly:AssemblyTrademark("MultiFileLibrary is a registered trademark of Wintellect")]

// AssemblyVersion version information:
[assembly: AssemblyVersion("3.0.0.0")]

// FILEVERSION/FileVersion version information:
[assembly: AssemblyFileVersion("1.0.0.0")]

// PRODUCTVERSION/ProductVersion version information:
[assembly: AssemblyInformationalVersion("2.0.0.0")]

// Set the Language field (discussed later in the "Culture" section)
[assembly:AssemblyCulture("")]
```

جدول زیر اطلاعاتی در مورد سوئیچ‌های AL جهت مقداردهی این فیلدهای نسخه بندی دارد (کامپایلر سی شارپ این سوئیچ‌ها را ندارد و بهتر است از طریق همان خصوصیات در کدها اقدام کنید). بعضی از اطلاعات زیر با استفاده از سوئیچ‌ها قابل تغییر نیستند؛ چرا که این مقادیر یا ثابت هستند یا اینکه طبق شرایطی از بین چند مقدار ثابت، یکی از آن‌ها انتخاب می‌شود.

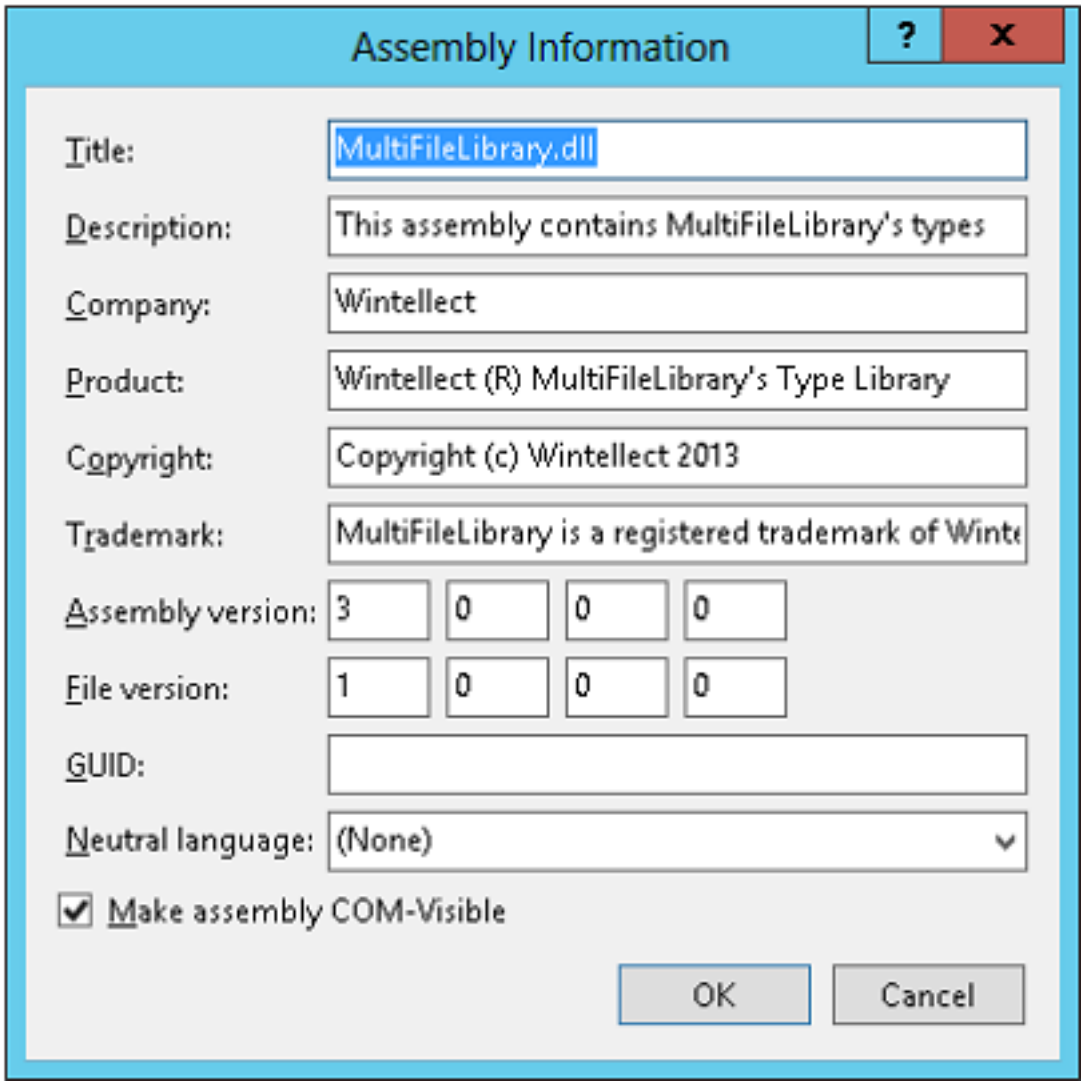
نسخه منبع	سوئیچ AL.exe	توصیف خصوصیت یا سوئیچ مربوطه
FILEVERSION	fileversion/	System.Reflection.AssemblyFileVersionAttribute .
PRODUCTVERSION	productversion/	System.Reflection. AssemblyInformationalVersionAttribute
FILEFLAGSMASK	-	Always set to VS_FFI_FILEFLAGSMASK (defined in WinVer.h as 0x0000003F).
FILEFLAGS	-	همیشه صفر است
FILEOS	-	در حال حاضر همیشه VOS__WINDOWS32 است
FILETYPE	target/	Set to VFT_APP if /target:exe or /target:winexe is specified; set to VFT_DLL if /target:library is specified.

نسخه منبع	سوئیچ AL.exe	توصیف خصوصیت یا سوئیچ مربوطه
FILESUBTYPE	-	Always set to VFT2_UNKNOWN . (This field has no meaning for VFT_APP and VFT_DLL .)
AssemblyVersion	version/	System.Reflection.AssemblyVersionAttribute
Comments	description/	System.Reflection.AssemblyDescriptionAttribute
CompanyName	company/	System.Reflection.AssemblyCompanyAttribute
FileDescription	title/	System.Reflection.AssemblyTitleAttribute
FileVersion	version/	System.Reflection.AssemblyFileVersionAttribute
InternalName	out/	ذکر نام فایل خروجی بدون پسوند.
LegalCopyright	copyright/	System.Reflection.AssemblyCopyrightAttribute
LegalTrademarks	trademark/	System.Reflection.AssemblyTrademarkAttribute
OriginalFilename	out	ذکر نام فایل خروجی بدون پسوند.
PrivateBuild	-	همیشه خالی است.
ProductName	product	System.Reflection.AssemblyProductAttribute
ProductVersion	productversion	System.Reflection.AssemblyInformationalVersionAttribute
SpecialBuild	-	همیشه خالی است.

موقعیکه شما یک پروژه‌ی سی شارپ را ایجاد می‌کنید، فایلی به نام AssebmlyInfo.cs در دایرکتوری Properties پروژه ایجاد می‌شود. این فایل شامل تمامی خصوصیت‌های نسخه بندی که در بالا ذکر شد، به‌علاوه یک سری خصوصیات دیگری که در آینده توضیح خواهیم داد، می‌باشد.

شما برای ویرایش این فایل می‌توانید به راحتی آن را باز کرده و اطلاعات داخل آن را تغییر دهید. ویژوال استودیو نیز برای ویرایش این فایل، امکانات GUI را نیز فراهم کرده است. برای استفاده از این امکان، پنجره‌ی properties را در سطح Solution

باز کرده و در تب Application روی Assembly Information کلیک کنید.



عنوان: **آشنایی با CLR: قسمت هفدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۶/۰۱ ۰:۴۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در مقاله قبلی در مورد افزودن منابع به اسمبلی صحبت‌هایی کردم که قسمتی از این منابع مربوط به اطلاعات نسخه بندی بود. در این مقاله قصد داریم این مسئله را بازتر کرده و در مورد نحوه‌ی نسخه بندی بیشتر صحبت کنیم.
در مقاله‌ی قبلی وقتی نسخه‌ی یک اسمبلی را مشخص می‌کردیم، از 4 عدد که با نقطه از هم جدا شده بودند، استفاده کردیم که در جدول زیر این 4 نامگذاری را مشاهده می‌کنید:

شماره اصلی Major Number	*شماره جزئی Minor Number*	*شماره ساخت Build Number*	*شماره بازبینی Revision Number*
2	5	719	2

اسمبلی بالا به ورژن یا نسخه‌ی 2.5.719.2 اشاره دارد که دو شماره‌ی اول (2.5) مثل تمامی برنامه‌ها به میزان تغییرات کارکردی یک اسمبلی اشاره دارد و عموم مردم هم نسخه یک نرم افزار را به همین دو عدد میشناسند. عدد سوم به این اشاره دارد که در شرکت، این ورژن از اسمبلی چندبار build شده است و شما باید به ازای هر Build این عدد را افزایش دهید. عدد آخری به این اشاره دارد که در طول روز انتشار، این چندمین Build بوده است. اگر در زمان ارائه‌ی این اسمبلی باگ مهمی در آن یافت شود، با هر بار Build آن در یک روز، باید این عدد افزایش یابد و برای روزهای آتی این مقدار مجددا آغاز می‌شود. مایکروسافت از این سیستم نسخه بندی استفاده می‌کند و بسیار توصیه می‌شود که توسعه دهندگان هم از این روش تبعیت کنند.

در جدول سابق شما متوجه شدید که سه نسخه بندی را می‌توان روی یک اسمبلی اعمال کرد که به شرح زیر است:

AssemblyFileVersion : این شماره نسخه در منابع اطلاعاتی Win32 ذخیره می‌گردد و کاربرد آن تنها جهت نمایش این اطلاعات است و CLR هیچ گونه ارجاع یا استفاده‌ای از آن ندارد. در حالت عادی، شما باید دو شماره اولی را جهت نمایش عمومی مشخص کنید. سپس با هر بار Build کردن، شماره‌های ساخت و بازبینی را هم به همان ترتیب افزایش می‌دهید. حالت ایده‌آل این است که ابزار AL یا CSC به طور خودکار با هر بار Build شدن، با توجه به زمان سیستم، به طور خودکار این دو شماره آخر را مشخص کنند ولی متاسفانه واقعیت این است که چنین کاری صورت نمی‌گیرد. این اعداد جهت نمایش و شناسایی اسمبلی برای اشکال زدایی مشکلات اسمبلی به کار می‌رود.

AssemblyInformationalVersion : این شماره نسخه هم در منابع اطلاعاتی Win32 ذخیره می‌گردد و تنها هدف اطلاعاتی دارد. مجددا اینکه CLR هیچ گونه اهمیتی به آن نمی‌دهد. این شماره نسخه به محصولی اشاره می‌کند که شامل این اسمبلی است.

به عنوان مثال ورژن 2 یک نرم افزار ممکن است شامل چند اسمبلی باشد که ورژن یکی از این اسمبلی‌ها یک است و دلیلش هم این است که این اسمبلی از نسخه‌ی 2 به بعد اضافه شده و در نسخه‌ی یک نرم افزار وجود نداشته است. به همین دلیل در این مدل از نسخه بندی شما دو شماره اول را به نسخه خود نرم افزار مقداردهی کرده و سپس مابقی اعداد را با هر بار پکیج شدن کل نرم افزار با توجه به زمان افزایش می‌دهید.

AssemblyVersion : این شماره نسخه در جدول متادیتای AssemblyDef ذخیره می‌گردد. CLR از این شماره نسخه جهت اتصال نام قوی Strongly Named به اسمبلی استفاده می‌کند (این مورد در فصل سه کتاب توضیح داده شده است). این شماره نسخه بسیار مهم بوده و به عنوان شناسه‌ی یکتا برای اسمبلی استفاده می‌شود.

موقعیکه شما شروع به توسعه‌ی یک اسمبلی می‌کنید، باید هر 4 شماره نسخه را مقداردهی کرده و تا زمانیکه توسعه‌ی نسخه بعدی

آن اسمبلی را آغاز نکرده‌اید، نباید آن را تغییر دهید. علت اصلی آن این است که موقعیکه اسمبلی «الف» با یک نام قوی به اسمبلی «ب» ارجاع می‌کند، نسخه‌ی اسمبلی «ب» در ورودی جدول AssemblyRef اسمبلی «الف» قرار گرفته است. این مورد باعث می‌شود زمانیکه CLR به بارگزاری اسمبلی «ب» احتیاج دارد، دقیقا می‌داند که چه نسخه‌ای با اسمبلی «الف» ساخته و تست شده است . البته این احتمال وجود دارد که CLR نسخه‌ای متفاوت از اسمبلی را با Binding Redirect بار کند که ادامه‌ی این مباحث در فصل سوم دنبال می‌شود.

عنوان: **آشنایی با CLR: قسمت هجدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۶/۰۲ ۱:۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در قسمت قبلی با نحوه‌ی نسخه بندی اسمبلی‌ها آشنا شدیم؛ ولی به غیر از نسخه بندی، فرهنگ (culture) هم قسمتی از عوامل شناسایی یک اسمبلی است. به عنوان نمونه من میتوانم یک اسمبلی داشته باشم که برای زبان آلمانی، انگلیسی آمریکایی، انگلیسی بریتانیایی و ... آماده شده است.
شناسایی فرهنگ یک اسمبلی از طریق یک رشته است که شامل یک تگ اصلی و ثانویه طبق استاندارد RFC1766 می‌باشد. جدول زیر تعدادی از این تگ‌ها را نمایش می‌دهد.

تگ اولیه	تگ ثانویه	فرهنگ مربوطه
De	-	آلمانی
De	AT	آلمانی اتریشی
De	CH	آلمانی سوئیسی
En	-	انگلیسی
En	GB	انگلیسی بریتانیا
En	US	انگلیسی آمریکایی

در حالت عادی که یک اسمبلی را که تنها شامل کد می‌شود، دارید برای آن culture تعریف نمی‌شود. چون مشخصه‌ی خاصی ندارد که به آن فرهنگ خاصی هم تعلق بگیرد. به این اسمبلی‌ها Culture Neutral یا خنثی می‌گویند. حال اگر شما در حال طراحی برنامه‌ای هستید که منابع Resources شامل مشخصه‌های فرهنگی و منطقه‌ای می‌شوند، مایکروسافت به شدت توصیه می‌کند که بحث کد را از منابع جدا کرده و اسمبلی هایشان جدا شوند. یک اسمبلی برای کدها و منابع مشترک استفاده شود که هیچ خصوصیت فرهنگی و منطقه‌ای خاصی ندارد. حال یک یا چند اسمبلی جداگانه برای منابع ساخته که هر کدام از آن‌ها به فرهنگ و منطقه‌ی خاصی اشاره می‌کنند. به این نوع اسمبلی‌ها Satellite Assembly یا اسمبلی ماهواره‌ای گویند. عموما از ابزار AL برای ساخت اسمبلی‌های ماهواره ای استفاده می‌شود. دلیل آن هم اینست که این اسمبلی‌ها چون عموما کدی را شامل نمی‌شوند، ساخت آن‌ها از طریق کامپایلر ممکن نیست. برای معرفی یک اسمبلی ماهواره‌ای باید از سوئیچ c یا culture استفاده کرد و به عنوان ورودی، این سوئیچ تگ‌ها را به آن نسبت داد.

```
/c[ulture]:En-US
```

بعد از اینکه اسمبلی ساخته شد در مسیر برنامه، در یک زیردایرکتوری که با همان شناسه‌ی تگ‌ها نام گرفته است، ذخیره می‌کنیم. به عنوان مثال اگر مسیر زیر، مسیر برنامه ما باشد:

```
C:\MyApp
```

اسمبلی ماهواره‌ای با مشخصات بالا باید در مسیر زیر قرار بگیرد:

```
C:\MyApp\en-US
```

در صورتی که قصد دارید در زمان اجرا به منابع یک اسمبلی ماهواره‌ای دسترسی پیدا کنید می‌توانید از کلاس زیر استفاده کنید:

```
System.Resources.ResourceManager
```

به هر حال اگر کدهای شما در فرهنگ و منطقه تاثیر دارند و دوست دارید اسمبلی کدها هم به عنوان یک اسمبلی ماهواره‌ای شناخته شوند از خصوصیت زیر برای معرفی اسمبلی خود استفاده کنید:

```
System.Reflection.AssemblyCultureAttribute
```

```
===============
[assembly:AssemblyCulture("de--CH")]
```

در AL هم از سوئیچ Culture/ استفاده می‌شود. به طور عادی شما نباید یک اسمبلی بسازید که به اسمبلی ماهواره ای اشاره می‌کند و جداول متادیتا اسمبلی‌ها باید به اسمبلی‌های خنثی اشاره کنند. اگر شما قصد دسترسی به اعضاء و خصوصیات یک اسمبلی ماهواره‌ای را دارید باید از طریق Reflection که در آینده در مورد آن صحبت خواهد شد اینکار را انجام دهید.(در کتاب جفری ریچر به *فصل بیست و سوم Assembly Loading and Reflection مراجعه شود)*

عنوان: **آشنایی با CLR: قسمت نوزدهم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۶/۰۳ ۱:۳۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در فصل دوم کتاب تا به الان یاد گرفتیم چگونه ماژول‌ها را کامپایل کنیم و چگونه آن‌ها را در یک اسمبلی قرار دهیم. حال وقت آن فرا رسیده است که با بسته بندی کردن (Package) و انتشار آن (Deploy) به طوری که کاربران بتوانند برنامه را اجرا کنند آشنا شویم.

نصب برنامه از طریق فروشگاه ویندوز

در فروشگاه ویندوز Windows Store Apps قوانین سخت و شدیدی برای بسته بندی کردن اسمبلی‌ها وجود دارد. ویژوال استودیو تمام اسمبلی‌های مورد نیاز برنامه را در یک فایل با پسوند appx قرار داده و آن را به سمت فروشگاه آپلود می‌کند. هر کاربری که این فایل appx را نصب کند، همه‌ی اسمبلی‌هایی را که در دایرکتوری مربوطه قرار گرفته است، توسط CLR بار شده و آیکن برنامه هم در صفحه‌ی start ویندوز قرار می‌گیرد و اگر دیگر کاربران همان سیستم هم این فایل appx را نصب کنند، از آنجا که قبلا روی سیستم موجود هست، تنها آیکن برنامه به صفحه‌ی start اضافه می‌گردد و برای حذف هم تنها آیکن برنامه از روی این صفحه حذف می‌شود؛ مگر اینکه تنها کاربری باشد که این برنامه را نصب کرده‌است که در آن صورت کلا همه‌ی اسمبلی‌های آن از روی سیستم حذف می‌شود.
در صورتیکه کاربرهای مختلف نسخه‌های مختلفی از همان برنامه را روی سیستم نصب کنند، برای اسمبلی‌ها هر کدام یک دایرکتوری ایجاد شده و به ازای نسخه‌ی نصب شده آن کاربر، یکی از این دایرکتوری‌ها مورد استفاده قرار می‌گیرند. کاربران مختلف می‌توانند روی سیستم به طور همزمان از نسخه‌های مختلف برنامه استفاده کنند.

روش‌های پکیج گذاری

برای برنامه‌های دسکتاپ که ربطی به فروشگاه ندارند و بین ایرانیان طرفدار زیادی دارد، نیازی به استفاده از هیچ روش خاصی نیست و یک کپی معمولی هم کفایت می‌کند. همه‌ی فایل‌های مثل اسمبلی، باید در یک دایرکتوری قرار گرفته و به روش کپی کردن آن را انتقال داد. یا برای بسته بندی از یک فایل batch کمک گرفت و آن را روی سیستم نصب کرد و نیازی به هیچ تغییری در رجیستری نیست. برای حذف برنامه هم، حذف معمولی دایرکتوری مربوطه کفایت می‌کند.
البته گزینه‌های دیگری هم برای پکیج کردن این نوع برنامه‌ها وجود دارند:
یکی از روش‌های پکیج کردن فایل‌ها به صورت cab هست که عموما برای سناریوهای اینترنتی و فشرده سازی و کاهش زمان دانلود به کار می‌رود.
روش دوم استفاده از پکیج MSI است که توسط سرویس نصب مایکروسافت *Microsoft Installer Service* یا **MSIExec.exe** انجام می‌گیرد. فایل‌های MSI به اسمبلی‌ها اجازه می‌دهند که بر اساس زمان تقاضای CLR برای بارگیری اولیه نصب شوند. البته این ویژگی جدیدی نیست و برای فایل‌های exe یا dll مدیریت نشده هم به کار می‌رود.

استفاده از نصاب سازها

بهتر هست که برای انتشار برنامه از برنامه‌های نصاب سازی استفاده کنید که با واسطی جذاب‌تر به نصب پرداخته و امکاناتی از قبیل shotrcutها، حذف و بازیابی و نصب و .. را هم به کاربر می‌دهند.
نصاب سازهای متفاوتی وجود دارند که در زیر به تعدادی از آنها اشاره می‌کنیم:
Install Shield (+) : این برنامه نسخه‌های متفاوتی را با قیمت‌های متفاوتی، عرضه می‌کند و در این زمینه، جزء بهترین‌ها نام برده می‌شود. حتی ویژگی‌های مخصوصی هم برای ویژوال استودیو دارد. شرکت سازنده، برنامه‌ی دیگری را هم اخیر تحت نام Install Anywhere عرضه کرده است که اجازه می‌دهد از روی یک برنامه برای پلتفرم‌های مختلف setup بسازد.

NSIS : این برنامه هم در زمینه‌ی ساخت setup محبوبیت زیادی دارد. این برنامه به صورت متن باز منتشر شده و رایگان است. امکانات این برنامه ساده است و برای راه اندازی سریع یک setup و اجرای راحت آن توسط کاربر، کاملا کاربردی است.

Tarma Installmate : این نرم افزار نسبت به InstallShield ساده‌تر و کم حجم‌تر است. حداقل برای برنامه‌های عادی امکانات

مناسبی دارد.

DeployMaster : یک برنامه‌ی دیگر با امکانات حرفه‌ای جهت انشار برنامه‌های دسکتاپ، که از ویندوز 98 تا 8.1 را در حال حاضر پشتیبانی می‌کند.

QSetup Installation Suite : یک برنامه‌ی نصب حرفه‌ای که فایل نهایی آن می‌تواند به دو فرمت exe یا MSI باشد و قابلیت‌هایی چون پشتیبانی از زبان فارسی، ورود لایسنس، سریال نرم افزار و ... را نیز پشتیبانی می‌کند.

Inno Setup : این برنامه هم امکانات خوبی را برای ساخت یک نصاب ساز دارد و همچنین از زبان پاسکال جهت اسکریپت نویسی جهت توسعه امکانات بهره می‌برد.

Visual Patch : وب سایت پی سی دانلود این برنامه را اینگونه توضیح می‌دهد:

نرم افزار Visual Patch یک ابزار توسعه یافته‌ی نرم افزاری برای ساخت پچ و آپدیت برنامه‌ها می‌باشد. این سازنده پچ باینری، استفاده از فشرده سازی داده DeltaMAX برای سریع‌تر کردن توسعه‌ی نرم افزار، یکپارچگی با نصب نرم افزار و ابزارهای مدیریت پچ از فروشندگانی نظیر Installshield, Lumension, Patchlink, Shavlik, Indigo Rose و ...، را به طور برجسته نمایان ساخته است.
با استفاده از این ابزار پچ کردن برنامه‌ها که برای توسعه دهندگان نرم افزار و برنامه نویسان طراحی شده است، توزیع نرم افزار و سیستم گسترش پچ بهبود می‌یابد. Visual Patch الگوریتم‌های فشرده سازی و state-of- the-art binary differencing را نمایان می‌سازد و این کمک می‌کند که شما به کوچکتر شدن و بهتر شدن پچ‌های نرم افزار اطمینان داشته باشید.
و ...

انتشار توسط ویژوال استودیو

ویژوال استودیو هم امکانات خوبی برای انتشار در بخش Properties پروژه، برگه‌ی publish ارائه می‌کند و فایل MSI نتیجه را به سمت وب سرور، FTP Server یا روی دیسک ارسال میکند. یکی از خصوصیات خوب این روش این است که میتواند پیش نیازهایی مانند فریم ورک دات نت یا sql server Express را به سیستم اضافه کنید؛ در نهایت با مزیت آپدیت و نصب تک کلیکی، کاربر، برنامه را بر روی سیستم نصب کند.

اسمبلی‌های انتشاریافته اختصاصی

در روش‌هایی که ذکر کردیم، از آنجا که اسمبلی‌ها در همان شاخه یا دایرکتوری برنامه قرار گرفته‌اند و نمی‌توان آن‌ها را با برنامه‌های دیگر به اشتراک گذاشت (مگر اینکه برنامه دیگری را هم در همان دایرکتوری قرار داد) به این روش *Privately Deployed Assemblies* می‌گویند. این روش برگ برنده بزرگی برای برنامه نویسان، کاربران و مدیران سیستم‌ها محسوب می‌شود. زیرا که جابجایی آنها راحت بوده و CLR در همانجا اسمبلی‌ها را در حافظه بار کرده و اجرا می‌کند. در این نوع برنامه‌ها عملیات نصب/جابجایی/ حذف به راحتی صورتی میگرد و نیازی به تنظیمات خارجی مانند رجیستری ندارد. یکی از خصوصیات مهمی که دارد این هست که جداول متادیتا به اسمبلی اشاره می‌کنند که برنامه بر پایه آن ساخته شده و با آن تست شده است؛ نه با اسمبلی موجود دیگر در سیستم که شاید نام نوع مورد استفاده آن یا اسمبلی آن به طور تصادفی با آن یکی است.
در مقالات آتی در مورد اشتراک گذاری اسمبلی‌ها بین چند برنامه مفصل‌تر صحبت خواهیم کرد.

عنوان: **آشنایی با CLR: قسمت بیستم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۶/۰۴ ۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

در قسمت قبلی با نحوه انتشار برنامه‌ها آشنا شدیم. در این قسمت نحوه پیکربندی یا تغییر پیکربندی برنامه را مشخص می‌کنیم. کاربر یا مدیر سیستم بهتر از هر کسی می‌تواند جنبه‌های مختلف اجرای برنامه را مشخص کند. به عنوان نمونه ممکن است مدیر سیستم بخواهد فایل‌های یک برنامه را سمت هارد دیسک سیستم کاربر انتقال دهد یا اطلاعات مانیفست یک اسمبلی را رونویسی کند و مباحث نسخه بندی که در آینده در مورد آن صحبت می‌کنیم.
با ارائه یک فایل پیکربندی در شاخه برنامه می‌توان به مدیر سسیتم اجازه داد تا کنترل بیشتر بر روی برنامه داشته باشد. ناشر برنامه می‌تواند این فایل را همراه دیگر فایل‌های برنامه پکیج کند تا در شاخه برنامه نصب شود تا بعدا مدیر یا کاربر سیستم بتوانند آن را تغییر و ویرایش کنند. CLR هم محتوای این فایل را تفسیر کرده و قوانین بارگیری اسمبلی‌ها و ... را تغییر می‌دهد. این فایل پیکربندی می‌تواند به صورت XML هم ارائه شود. مزیت قرار دادن یک فایل جداگانه نسبت به رجیستری این مزیت را دارد که هم قابل جابجایی و پشتیبانی گیری است و هم اینکه تغییر آن ساده‌تر است.
البته در آینده بیشتر در مورد این فایل صحبت می‌کنیم ولی در حال حاضر بهتر است اندکی طعم آن را بچشیم. فرض را بر این می‌گذاریم که ناشر می‌خواهد فایل‌های اسمبلی MultiFileLibrary را در دایرکتوری جداگانه‌ای قرار دهد و چیزی شبیه به ساختار زیر را در نظر دارد:

```
AppDir directory (contains the application’s assembly files)
Program.exe
Program.exe.config (discussed below)


AuxFiles subdirectory (contains MultiFileLibrary’s assembly files)
MultiFileLibrary.dll
FUT.netmodule
RUT.netmodule
```

حال با تنظیم بالا به دلیل اینکه CLR انتظار دارد این اسمبلی را در دایرکتوری برنامه بیابد و با این جابجایی قادر به انجام این کار نیست، استثنای زیر را صادر می‌کند:

```
System.IO.FileNotFoundException
```

برای حل این مشکل، ناشر یک فایل XML را ایجاد کرده و در مسیر دایرکتوری برنامه قرار می‌دهد. این فایل باید همنام اسمبلی اصلی برنامه با پسوند config. باشد. به عنوان مثال، نام فایل می‌شود: Program.exe.config و فایل پیکربندی هم چیزی شبیه فایل زیر می‌شود:

```
<configuration>
<runtime>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<probing privatePath="AuxFiles" />
</assemblyBinding>
</runtime>
</configuration>
```

حالا هر موقع CLR در جست و جوی یک اسمبلی باشد که نتواند آن را در دایرکتوری مربوطه بیابد مسیر Auxfiles را هم بررسی خواهد کرد. با قرار دادن کارکتر « , » هم می‌توان برای خصوصیت PrivatePath، دایرکتوری‌های زیادتری را معرفی کرد. البته مسیردهی این خصوصیت باید به طور نسبی باشد نه مطلق یا اینکه یک مسیر نسبی خارج از دایرکتوری برنامه. ایده اصلی اینکار این است که برنامه کنترل بیشتری روی دایرکتوری خود و دایرکتوری‌های زیرمجموعه داشته باشد نه خارج از آن.

نحوه عملکرد اسکن CLR برای بارگزاری اسمبلی ها

موقعی که CLR قصد بارگزاری اسمبلی‌های خنثی را دارد، شاخه‌های زیر را به طور خودکار اسکن خواهد نمود (مسیرهای FirstPrivatePath و SecondPrivatePath توسط فایل پیکربندی مشخص شده است)

```
AppDir\AsmName.dll
AppDir\AsmName\AsmName.dll
```

```
AppDir\firstPrivatePath\AsmName.dll
AppDir\firstPrivatePath\AsmName\AsmName.dll
AppDir\secondPrivatePath\AsmName.dll
AppDir\secondPrivatePath\AsmName\AsmName.dll
...
```

این نکته ضروری است که اگر اسمبلی شما در یک دایرکتوری همنام خودش (در مثال ما MultiFileLibrary) قرار بگیرد، نیازی نیست این مسیر را در فایل پیکربندی ذکر کنید؛ زیرا CLR در صورت نیافتن دایرکتوری با این نام را اسکن خواهد نمود. بعد از آن اگر به هر نحوی CLR نتواند اسمبلی را در هیچ کدام از دایرکتوری‌های گفته شده بیابد، با همان قوانین گفته شده اینبار به دنبال فایلی با پسوند exe خواهد بود و اگر باز هم جست و جوی آن نتیجه‌ای را در بر نداشته باشد، استثنای زیر را صادر می‌کند:

```
FileNotFoundException
```

برای اسمبلی‌های ماهواره‌ای همان قوانین بالا دنبال می‌شود؛ با این تفاوت که انتظار می‌رود اسمبلی داخل یک زیر دایرکتوری با تگ‌های RFC1766 مطابقت داشته باشد. به عنوان مثال اگر اسمبلی با فرهنگ و منطقه En-US مشخص شده باشد، دایرکتوری‌های زیر اسکن خواهند شد:

```
C:\AppDir\en-US\AsmName.dll
C:\AppDir\en-US\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.dll
C:\AppDir\en-US\AsmName.exe
C:\AppDir\en-US\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.exe
C:\AppDir\en\AsmName.dll
C:\AppDir\en\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.dll
C:\AppDir\en\AsmName.exe
C:\AppDir\en\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.exe
```

نحوه اسکن کردن CLR میتواند به ما بگوید که عمل اسکن می‌تواند گاهی اوقات با زمان زیادی روبرو شود (به خصوص که قابلیت اسکن روی شبکه را هم دارد). برای محدود کردن ناحیه یا نواحی اسکن می‌توانید یک یا چند المان culture را در فایل پیکربندی مشخص کنید. همچنین مایکروسافت ابزاری به نام FUSLogVw.exe را ارائه داده است که می‌تواند نواحی اسکن را در حین اجرای برنامه به ما گزارش دهد.

نام و محل فایل پیکربندی بسته به نوع برنامه می‌تواند متغیر باشد:
برای فایل‌های اجرایی EXE فایل پیکربندی باید در شاخه فایل اجرایی باشد و باید نام فایل پیکربندی همانند فایل exe بوده و یک پسوند config. را به آن اضافه کرد.

برای برنامه‌های وب فرم، فایل web.config موجود است که در ریشه شاخه مجازی وب اپلیکیشن قرار میگیرد و هر زیر دایرکتوری هم می‌تواند یک web.config جداگانه داشته باشد که میتواند از web.config ریشه هم تنظیماتش را ارث بری کند. برای نمونه آدرس زیر را در نظر بگیرد:

```
http://www.dotnettips.info/newsarchive
```

یک فایل کانفیگ در ریشه قرار می‌گیرد و یکی هم در زیر شاخه newsArchive می‌تواند قرار بگیرد.

فصل دوم « *نحوه ساخت و توزیع اسمبلی ها* » از بخش اول « *اصول اولیه CLR* » پایان یافت. فصل بعدی در مورد اسمبلی‌های اشتراکی است که بعد از آماده شدن این فصل، قسمت‌های بعدی در دسترس عزیزان قرار خواهد گرفت.

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: **توصیه هایی در استفاده از NGEN**
نویسنده: م منفرد
تاریخ: ۱۳۹۲/۱۱/۲۷ ۹:۰
آدرس: www.dotnettips.info
گروه‌ها: Performance, JIT, NGEN

یکی از مسائل ربز و فنی در دنیای .NET استفاده یا عدم استفاده از NGEN است. در مقاله کوتاه زیر بهترین مکان‌های استفاده و عدم استفاده از آن را در چند بند خلاصه می‌کنم:

کجا از NGEN استفاده کنیم؟

برنامه هایی که مقدار زیادی کد مدیریت شده قبل از نمایش فرم برنامه دارند. مانند برنامه Blend که مقدار زیادی کد در شروع برنامه دارد. استفاده از ngen می‌تواند باعث افزایش کارایی و سرعت اجرای برنامه شود

فریم ورک ها، dllها و کامپوننت‌های عمومی: کدهای تولید شده توسط JIT قابل اشتراک بین برنامه‌های مختلف نیستند ولی NGEN قابل اشتراک مابین برنامه‌های مختلف می‌باشد. بنابراین اگر کامپوننتی دارید که در بین برنامه‌های مختلف مشترک استفاده می‌شود، این کار می‌تواند سرعت شروع برنامه‌ها را بالا برده استفاده از منابع سیستم را کاهش دهد

برنامه هایی که در terminal serverها استفاده می‌شوند: توضیح فوق در مورد این برنامه‌ها نیز صادق است.

کجا از NGEN استفاده نکنیم؟

برنامه‌های کوچک: عملا سرعت JIT آن قدر بالا است که NGEN کار را کندتر خواهد کرد!

برنامه‌های سروری که سرعت شروع آن مهم نیست: برنامه‌ها یا dll هایی که سرعت شروع آنها مهم نیست، اگر NGEN نشوند سرعت بیشتری برای شما به ارمغان خواهند داشت چون JIT در هنگام اجرا، کد را بهینه می‌کند ولی NGEN این کار را انجام نمی‌دهد.

چند نکته دیگر که باید در نظر داشته باشید این است که قرار نیست NGEN مثل یک جادوگر کد شما را جادو کند که سریع‌تر اجرا شود. تنها کد را از قبل به کد native مربوط به معماری cpu شما کامپایل خواهد کرد که شروع اجرای آن سریع‌تر شود. البته این جادوگر :) قربانی هم می‌خواهد. قربانی آن optimizationهای داخلی JIT است که در برنامه شما اعمال نخواهد شد. بنابراین در رابطه با استفاده از NGEN نهایت دقت را به خرج دهید.

نظرات خوانندگان

نویسنده: شهروز جعفری
تاریخ: ۱۳۹۲/۱۱/۲۷ ۱۶:۲۲

بد نیست به این مطلب (NGEN.EXE در دات نت) یک سری بزنید.

نویسنده: ناصرنیازی
تاریخ: ۱۳۹۲/۱۱/۲۹ ۱۹:۲۸

می خواستم بپرسم آیا می‌توان با این برنامه فایل اجرایی برنامه را Standalone کرد مثل فایل‌های دلفی۷ که همه جا اجرا میشه ؟

نویسنده: م منفرد
تاریخ: ۱۳۹۲/۱۱/۳۰ ۰:۳۰

خیر. برای اجرای برنامه هنوز نیاز به فریمورک دات نت دارید.

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.

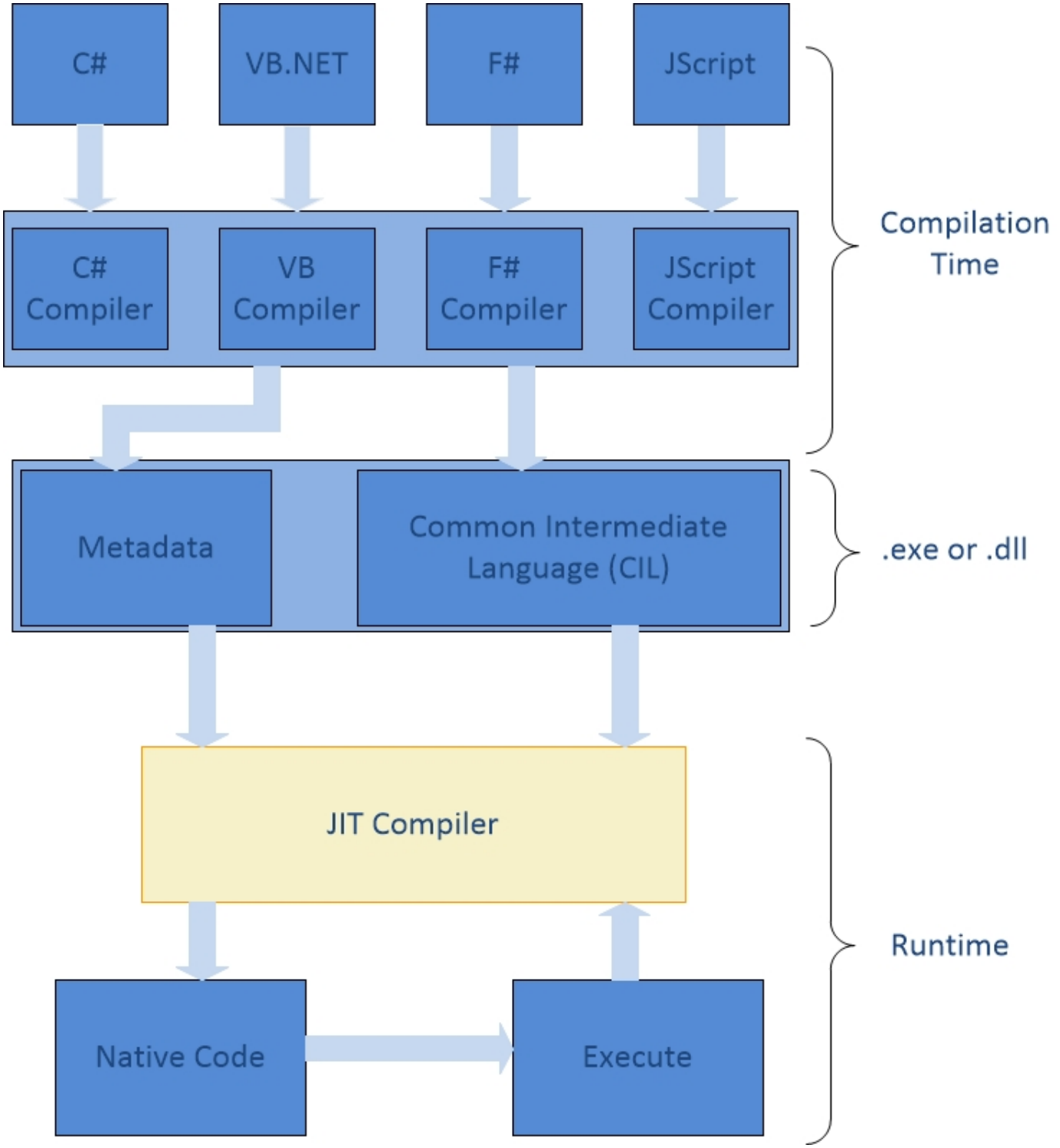
Implicit

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.

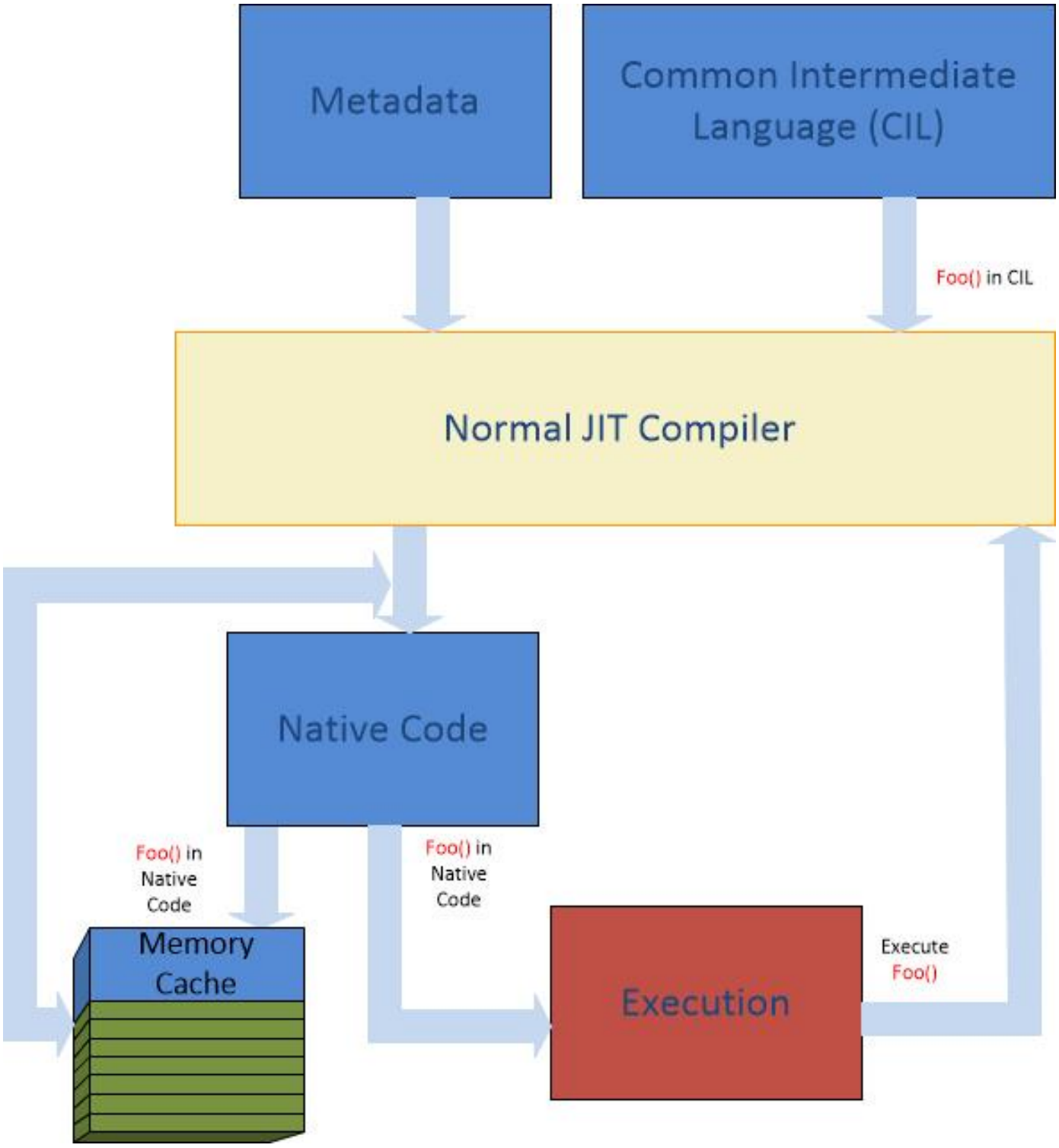


همانطور که در تصویر فوق مشاهده می‌کنید، سورس کد توسط کامپایلر دات نت به exe و یا dll کامپایل می‌شود. کامپایلر JIT تنها متدهایی را که در زمان اجرا(runtime) فراخوانی می‌شوند را کامپایل می‌کند. در دات نت فریم ورک سه نوع JIT Compilation داریم:

Normal JIT Compilation

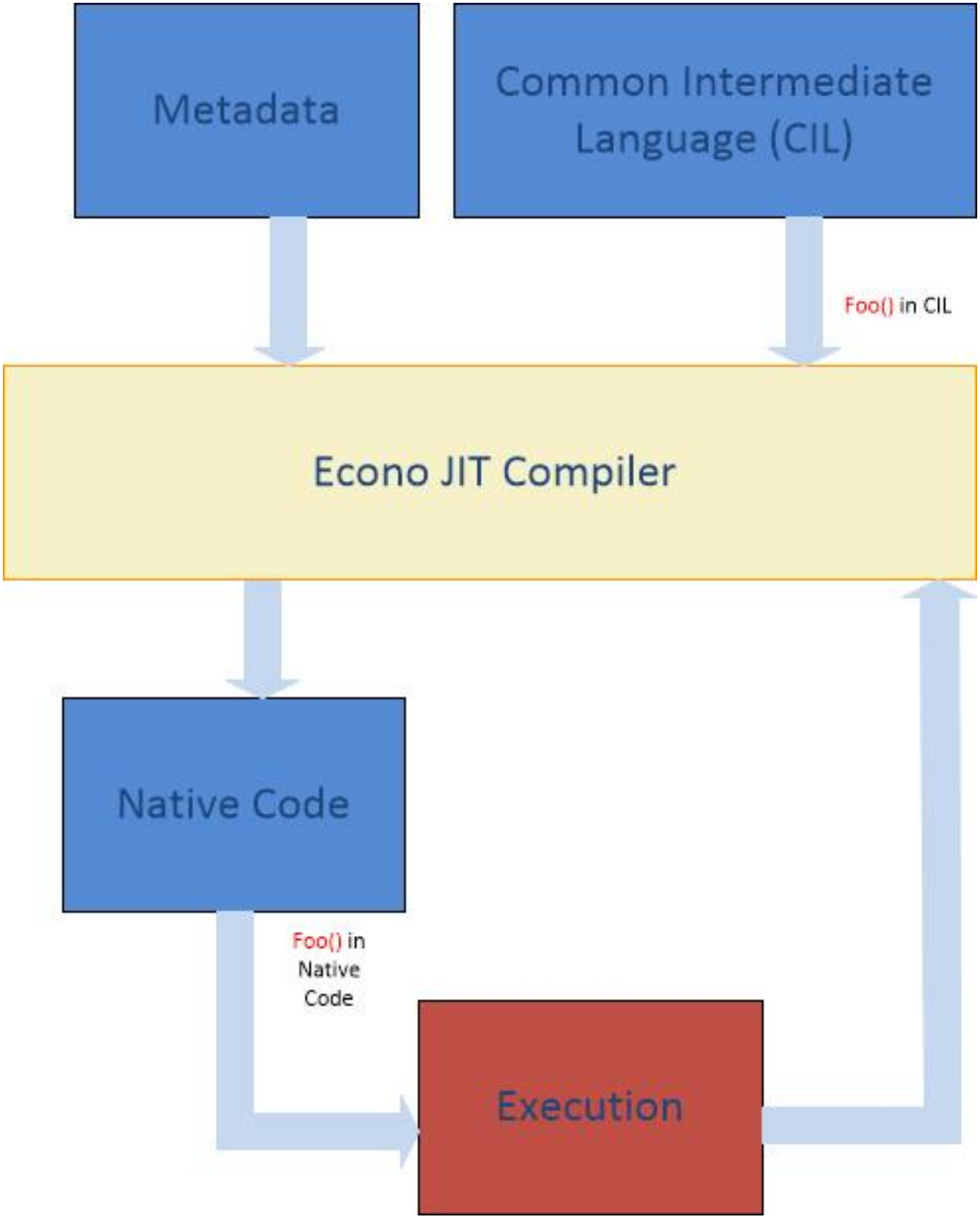
در این نوع کامپایل، متدها در زمان فراخوانی در زمان اجرا کامپایل می‌شوند. بعد از اجرا، متد داخل حافظه ذخیره می‌شود. به متدهای ذخیره شده در حافظه jitted گفته می‌شود. دیگر نیازی به کامپایل متد jit شده نیست. در فراخوانی بعدی، متد مستقیماً

از حافظه کش در دسترس خواهد بود.



Econo JIT Compilation

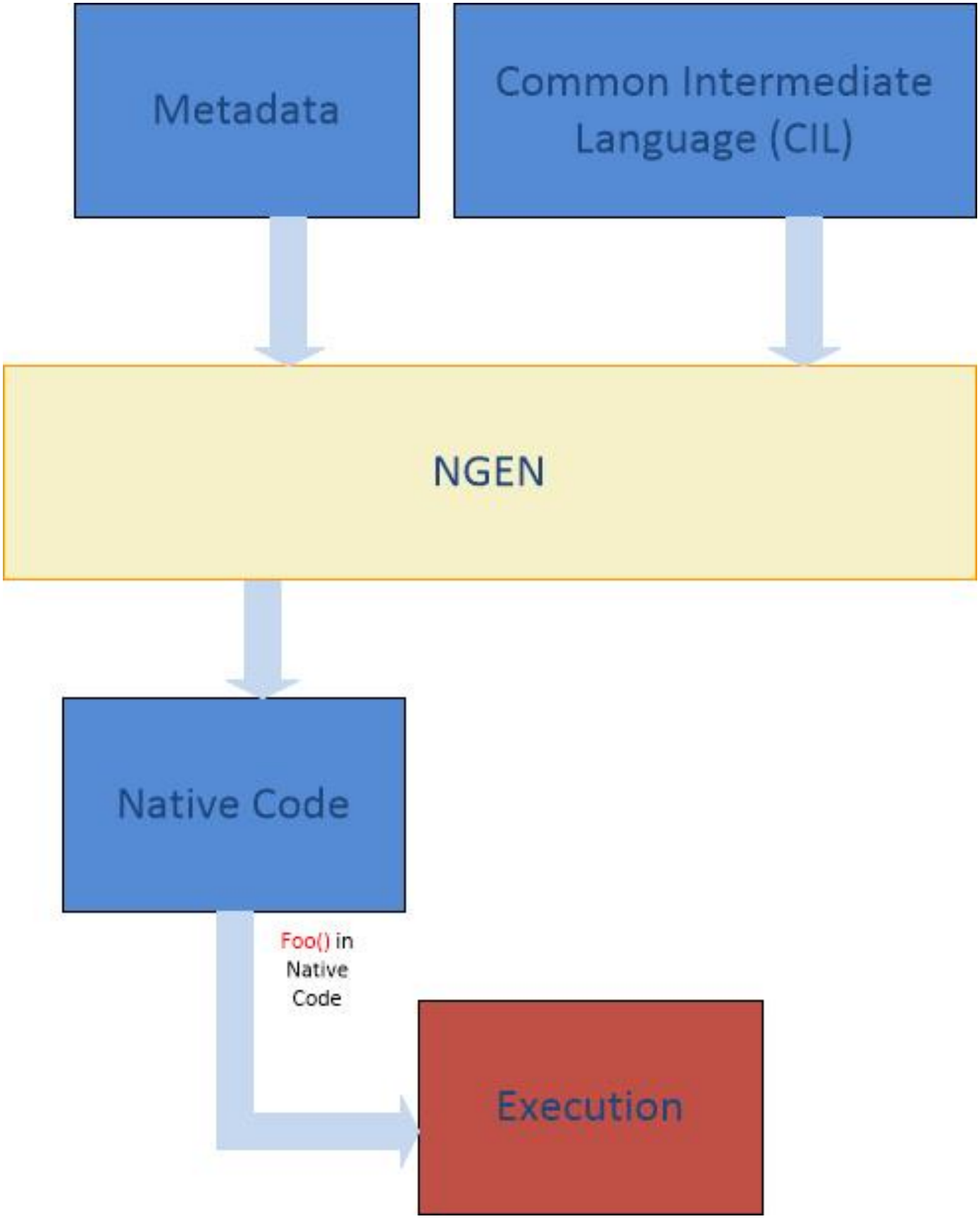
این نوع کامپایل شبیه به حالت Normal JIT است با این تفاوت که متدها بلافاصله بعد از اجرا از حافظه حذف می‌شوند.



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.

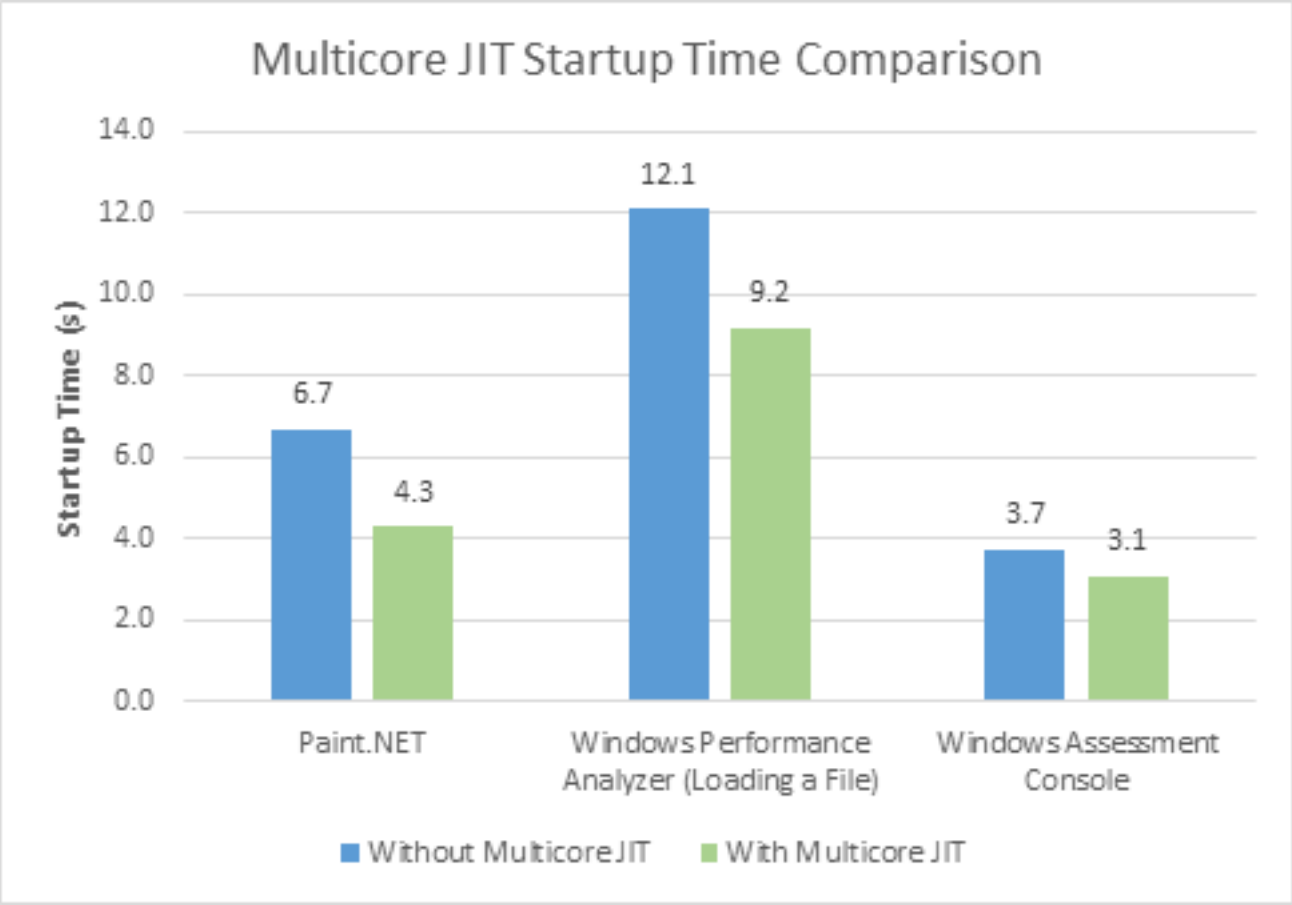


Multicore JIT

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.

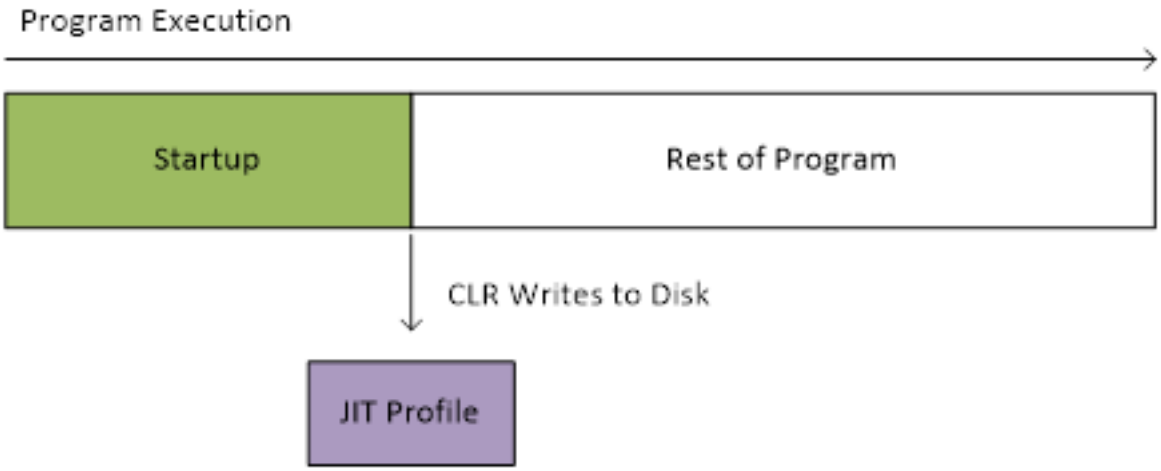
Multicore JIT همچنین می‌تواند باعث بهبود سرعت در برنامه‌های WPF شود. در نمودار زیر می‌توانید حالت‌های استفاده و عدم استفاده از Multicore JIT را در سه برنامه WPF نوشته شده مشاهده کنید.



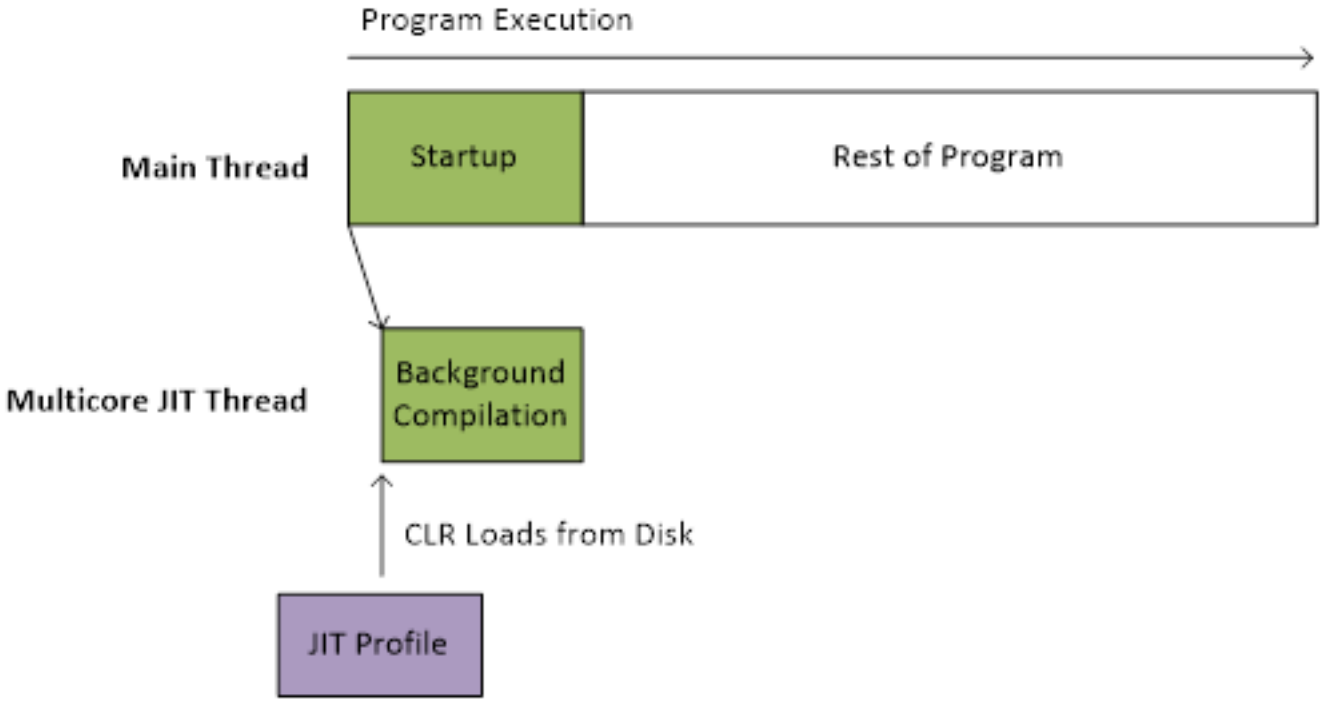
Multicore JIT در عمل

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>
```