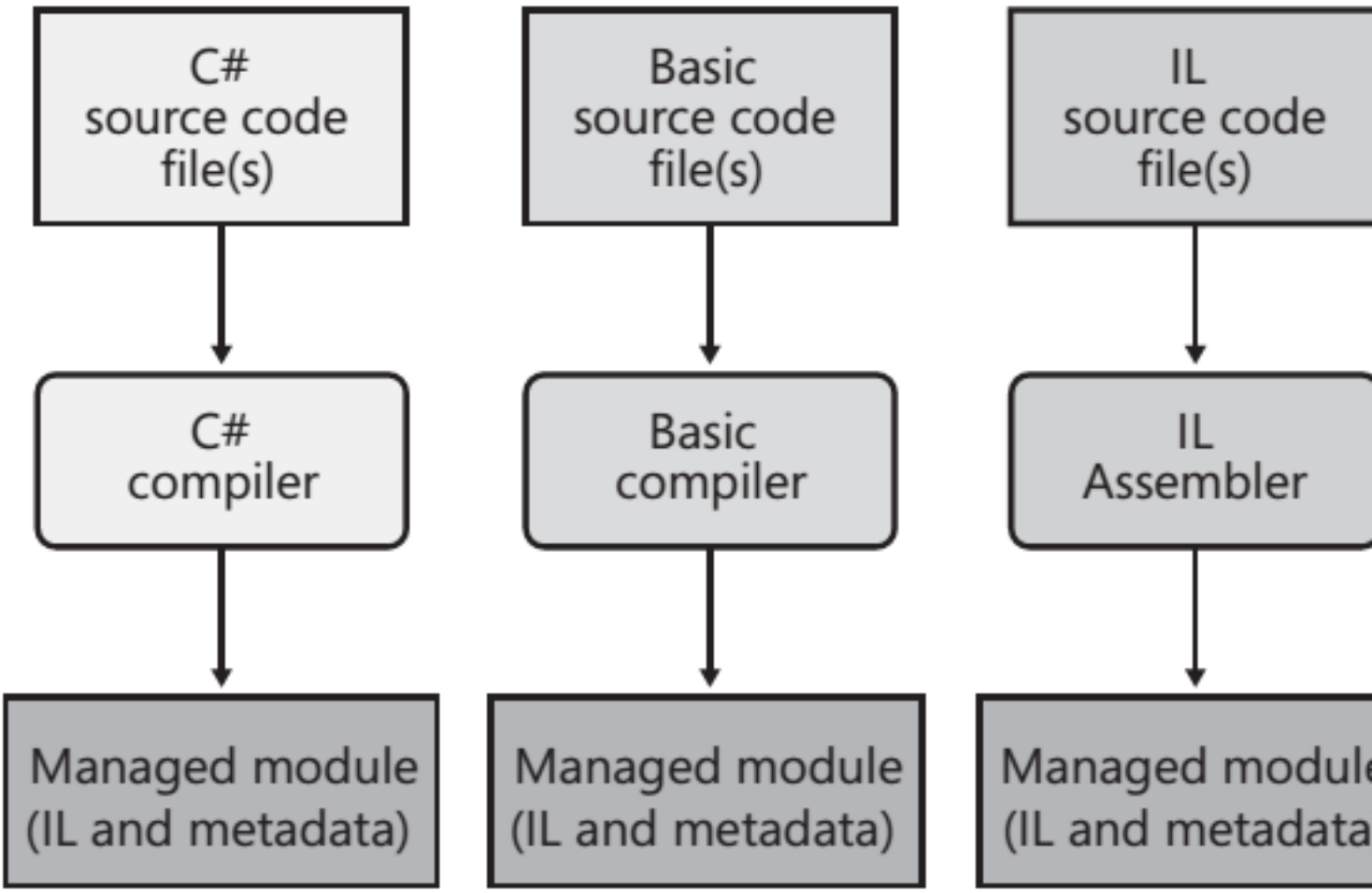
عنوان: **همه چیز در مورد CLR : قسمت اول**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۲ ۱۵:۴۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

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 DEP یا Data Execution Prevention و همچنین ASLR یا Address Space Layout Randomization در آن‌ها لحاظ شده باشد.

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.

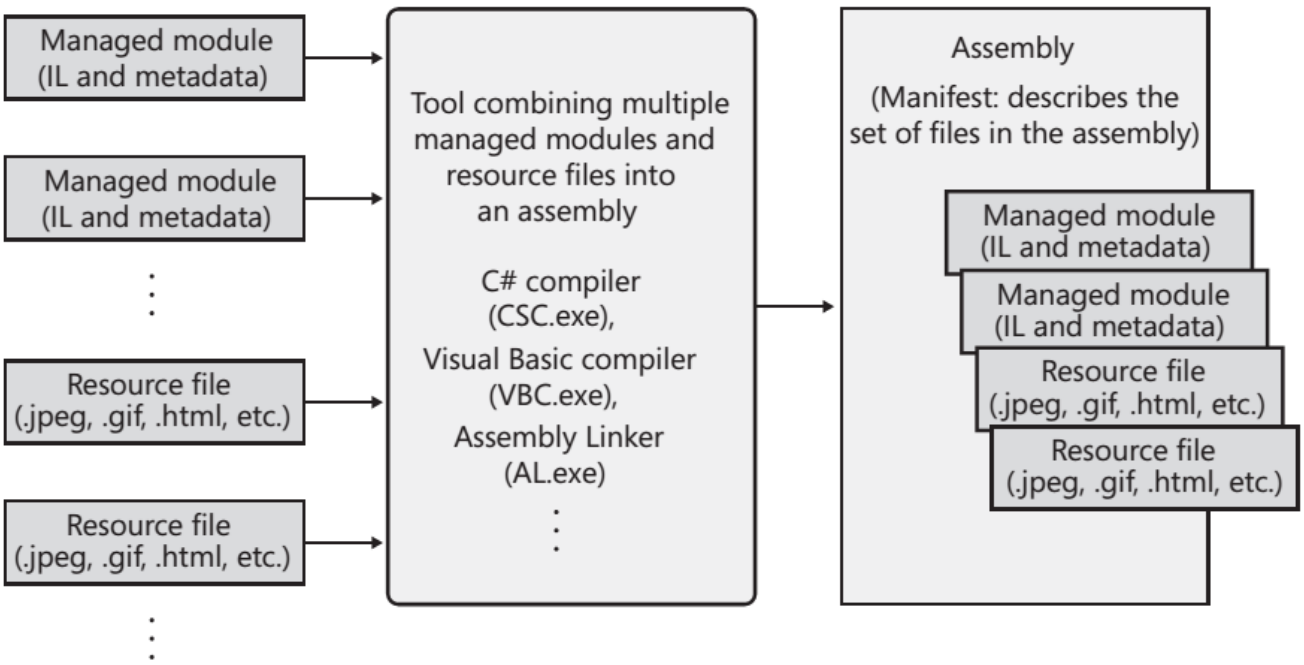
عنوان: **آشنایی با CLR: قسمت سوم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۴ ۱:۱۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

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.

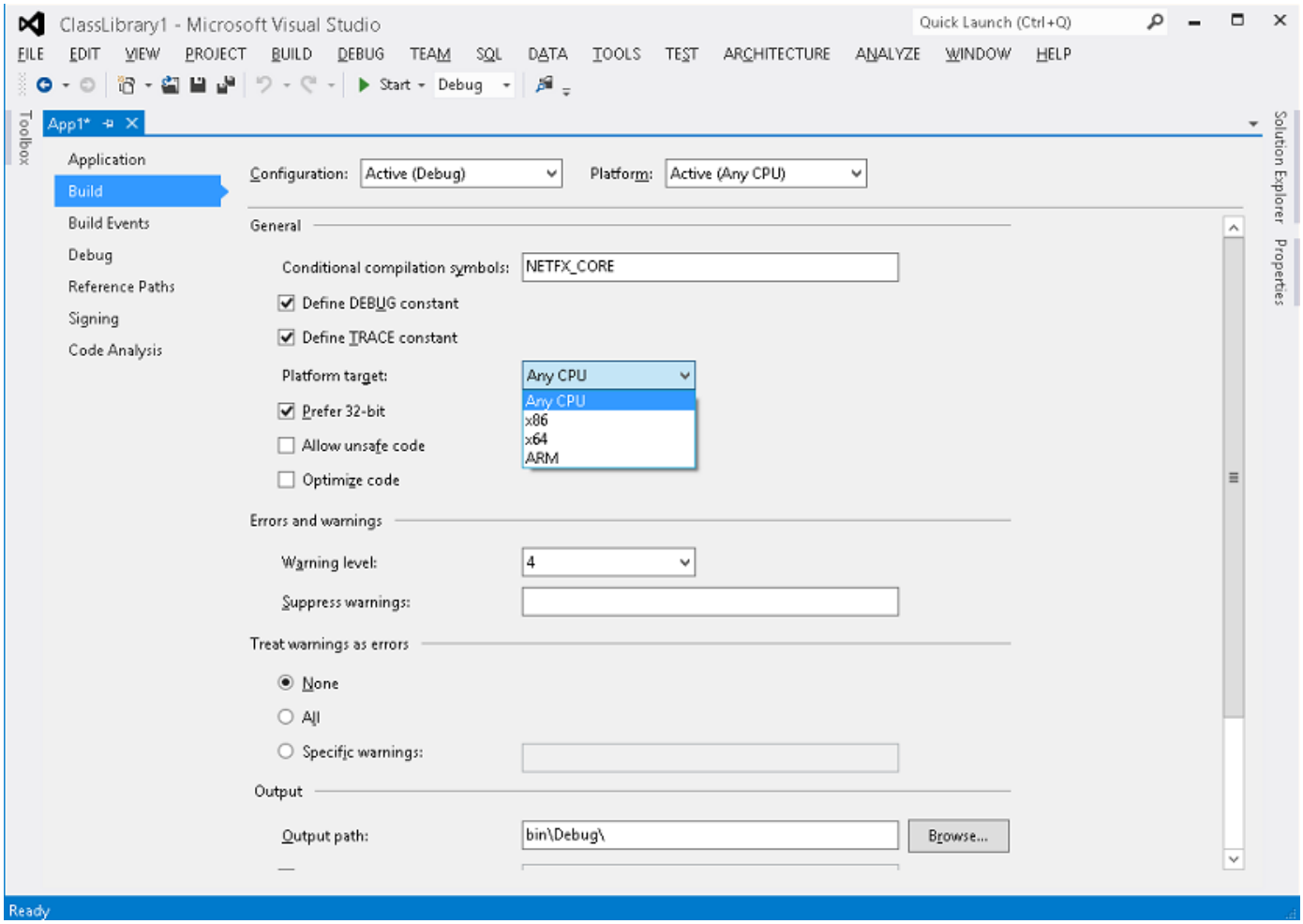
عنوان: **آشنایی با CLR: قسمت چهارم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۵ ۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

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
```

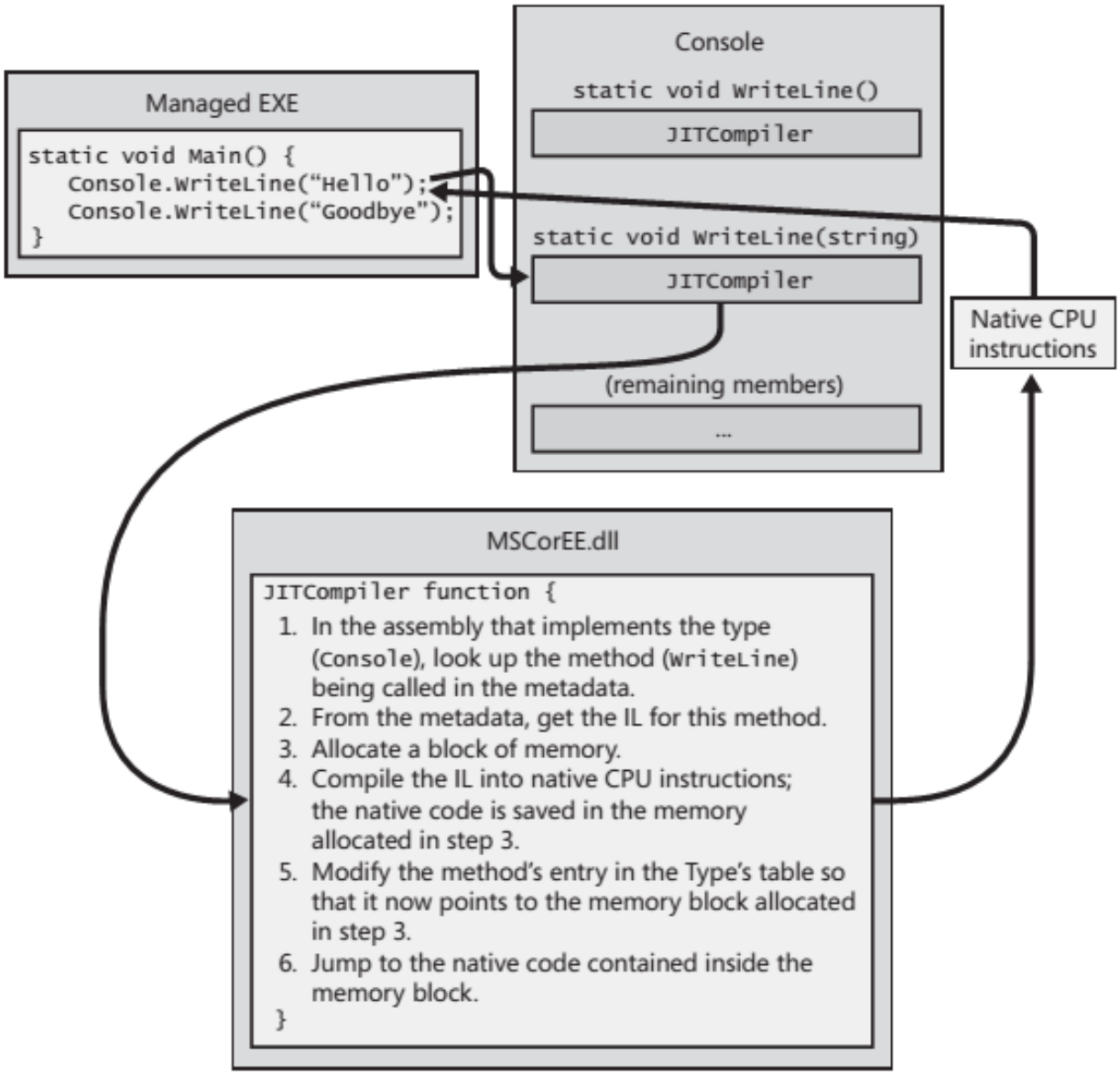
بعد از آن ترد اصلی پروسه، متدی را در MSCorEE صدا خواهد زد که موجب آماده سازی CLR بارگذاری اسمبلی اجرایی EXE در حافظه و صدا زدن مدخل ورودی برنامه یعنی متد Main می‌گردد. به این ترتیب برنامه‌ی مدیریت شده (managed) شما اجرا می‌گردد.

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:

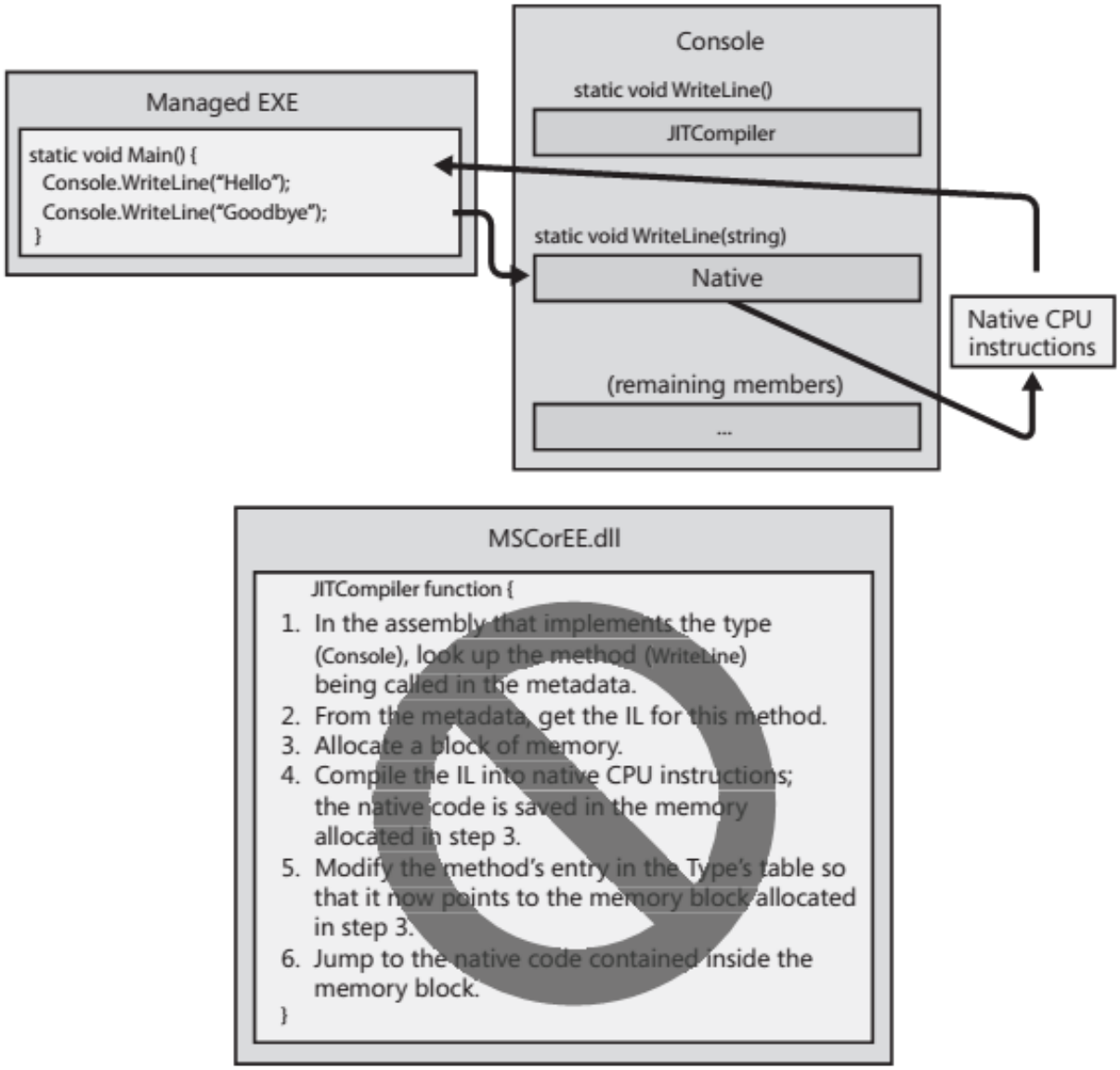


قبل از اجرای متد Main، ابتدا CLR به دنبال ارجاعاتی می‌گردد که در این متد استفاده شده است تا یک ساختار داده داخلی، برای ارجاعات این متد در حافظه تشکیل شود. در شکل بالا یک ارجاع وجود دارد و آن هم شیء کنسول است. این ساختار داده داخلی شامل یک مدخل ورودی (آدرس آغاز در حافظه) به ازای هر متد تعریف شده در نوع 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

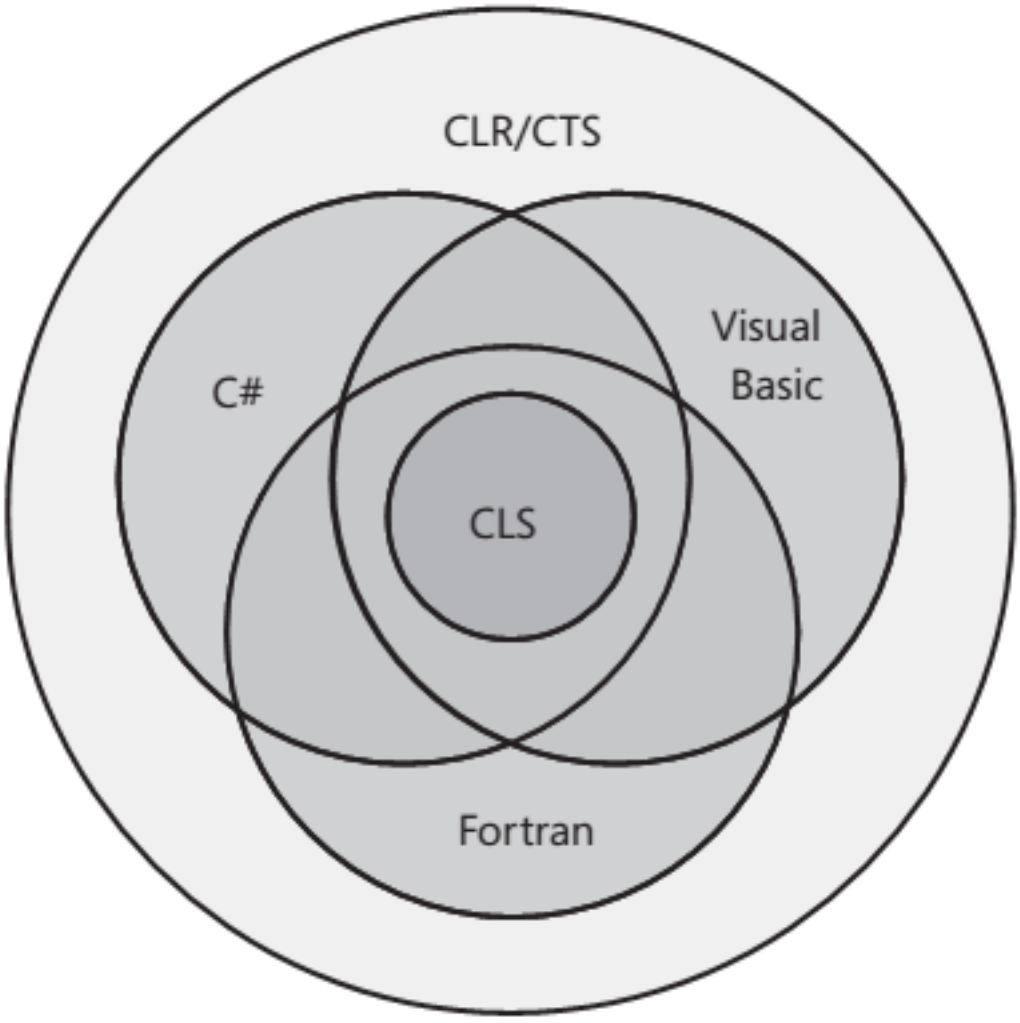
CLS

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.