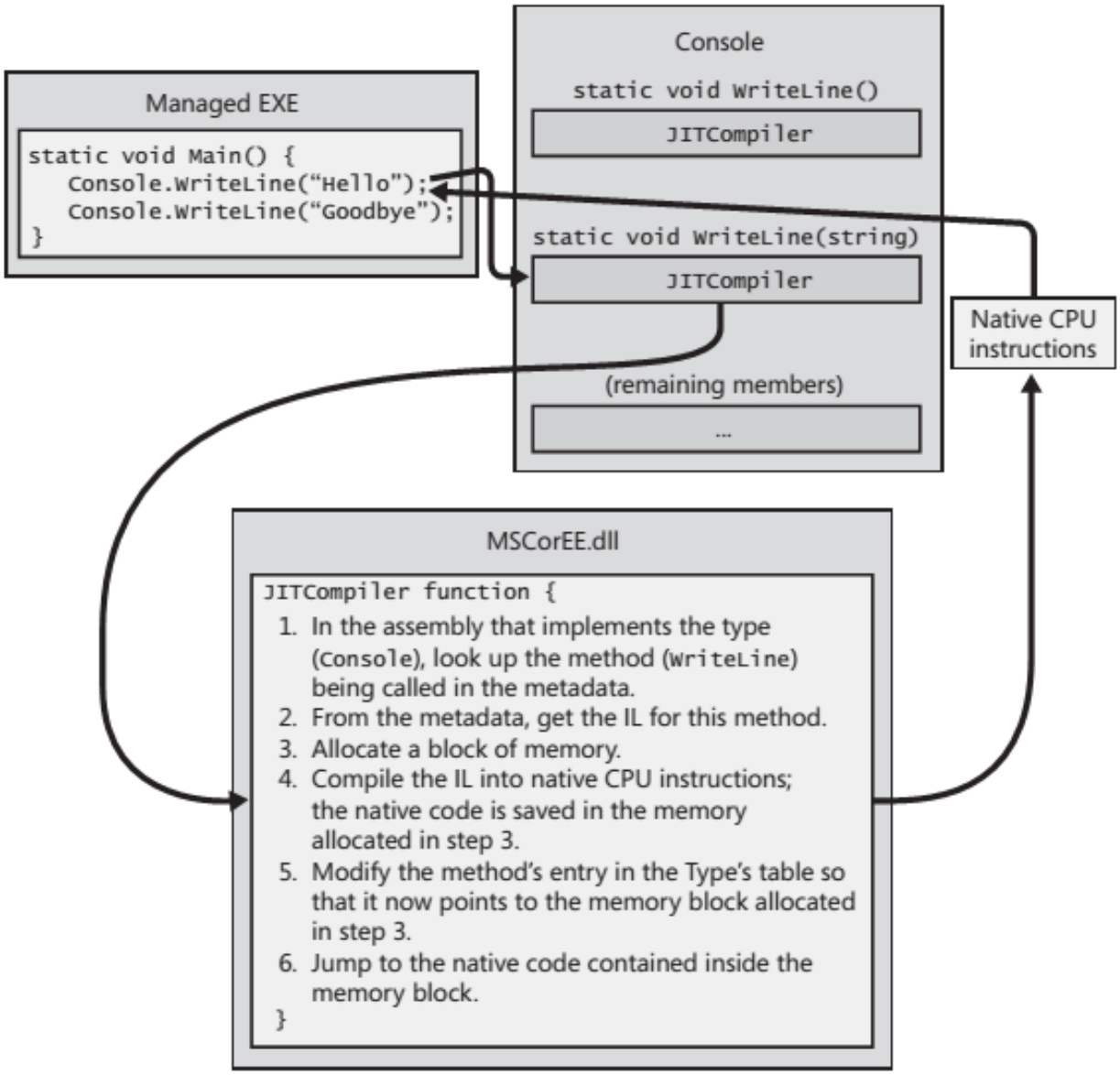
عنوان: **آشنایی با CLR: قسمت پنجم**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۳/۲۶ ۱:۱۰
آدرس: www.dotnettips.info
گروه‌ها: CLR, .NET Framework

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:

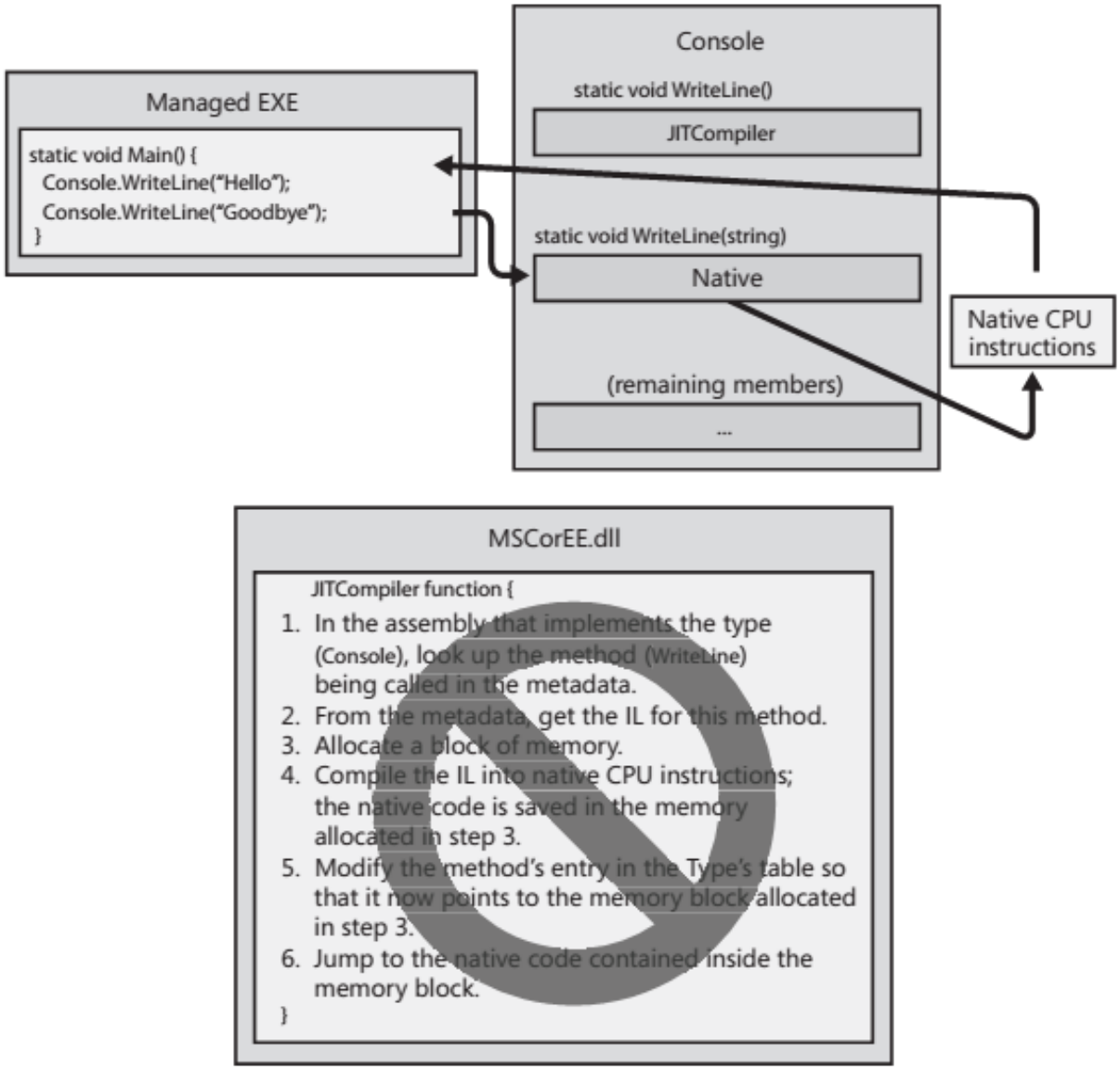


قبل از اجرای متد Main، ابتدا CLR به دنبال ارجاعاتی می‌گردد که در این متد استفاده شده است تا یک ساختار داده داخلی، برای ارجاعات این متد در حافظه تشکیل شود. در شکل بالا یک ارجاع وجود دارد و آن هم شیء کنسول است. این ساختار داده داخلی شامل یک مدخل ورودی (آدرس آغاز در حافظه) به ازای هر متد تعریف شده در نوع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