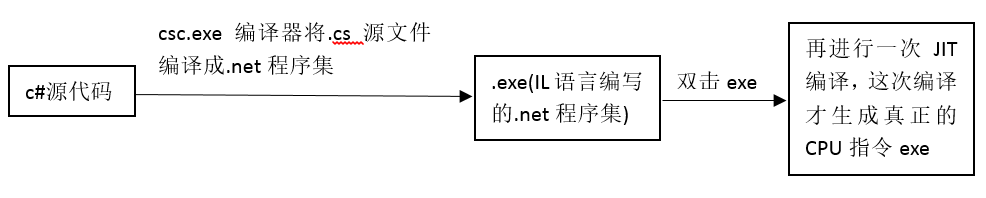
# c#编译执行过程

c#编译执行过程如下：



1. 编写C#源代码

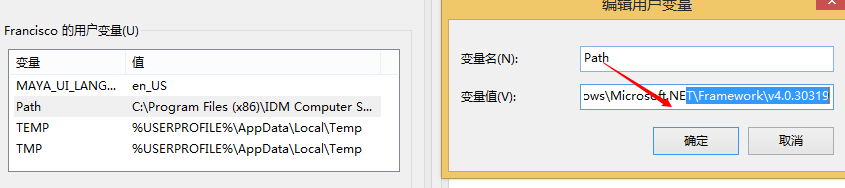
先用visual studio、ultraedit、sublime甚至记事本等文本编辑器，编写名为Program.cs的文本类型的源文件。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace 编译过程示例  {  class Program  {  static void Main(string[] args)  {  Console.WriteLine("Hello, World");  Console.ReadLine();  }  }  } |

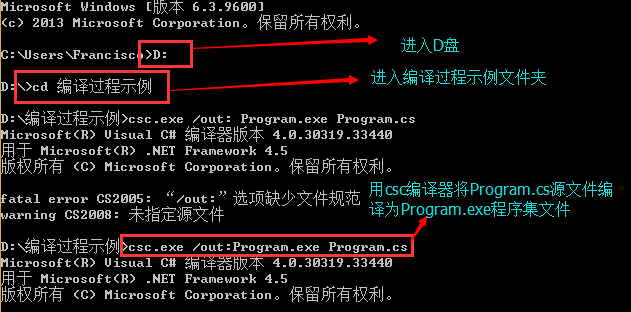
1. 编译成程序集文件

用csc.exe文件将源文件Program.cs（位于D盘的编译过程示例文件夹下）文件编译为名为Program.exe的.net程序集文件。

首先，把csc.exe添加到环境变量中，csc.exe的默认安装目录为：“C:\Windows\Microsoft.NET\Framework\v4.0.30319（这个是版本号）”。把目录添加到环境变量的path中即可。



之后打开cmd，输入以下的代码进行编译



这个编译好的exe文件并不是真正的操作系统可以理解的二进制文件，而是.net程序集，这个程序集实际上时把C#源代码编译成了MSIL源代码（IL语言）。

1. 双击生成的Program.exe运行。

当双击exe文件运行时，还会进行一次JIT即时编译（just in time），编译成CPU能识别的代码

1. 还可以用ngen.exe(位于于csc.exe相同的目录下)生成本机映像，并将它们安装到本地计算机上的本机映像缓存（C:\Windows\assembly\NativeImages\_v4.0.30319\_64）中。 运行时可从缓存中使用本机映像，而不必使用实时 (JIT) 编译器编译原始程序集。

使用方法：ngen.exe install Program.exe

# .net应用程序运行的背后机制

这节要探讨的问题是：当编译后的应用程序运行时，CLR是如何进行运作的。

## 1. 准备工作

程序清单Program.cs：

|  |
| --- |
| public sealed class Pragram{  public static void Main()  {  System.Console.WriteLine("Hi");  }  } |

在Developer Command Prompt for VS2013中编译上面的代码成为应用程序（程序集）

|  |
| --- |
| csc.exe Program.cs |

运行，程序开始执行。

|  |
| --- |
| Program.exe |

用ILDasm打开编译的程序集

|  |
| --- |
| ILDasm.exe Program.exe |

程序就可以运行了。

## 2. CLR加载并初始化自身

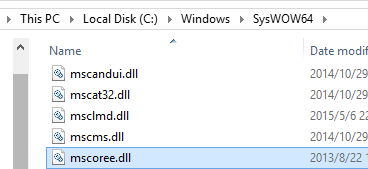
通过双击Program.exe或在CMD中运行程序时，Window会检查EXE文件头，决定创建32位还是64位进程之后，会在进程地址空间加载MSCorEE.dll的x86，x64或ARM版本。

如果操作系统是x86或ARM版本，MSCorEE.dll的x86在以下目录中

|  |
| --- |
| %SystemRoot%\System32 |

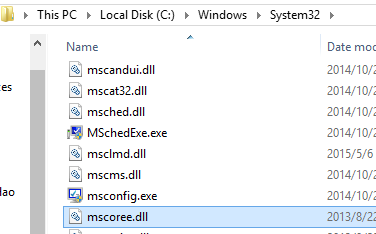
如果操作系统是x64，MSCorEE.dll的x86版本在以下目录中

|  |
| --- |
| %SystemRoot%\SysWow64 |



MSCorEE.dll的x64版本则在以下目录中

|  |
| --- |
| %SystemRoot%\System32 |



然后，进程的主线程调用MSCorEE.dll中定义的一个方法，这个方法会初始化CLR。之后权限就交给CLR了

## 从CLR头读取入口点

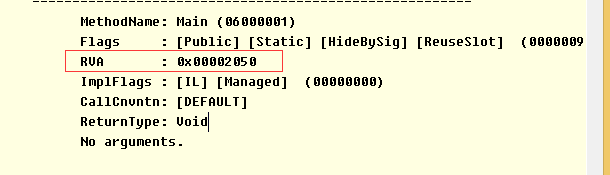
CLR初始化完成后，它会读取CLR头，查找应用程序入口标记

我们可以用ILDasm（view→headers）来查看程序入口标记。

|  |
| --- |
|  |

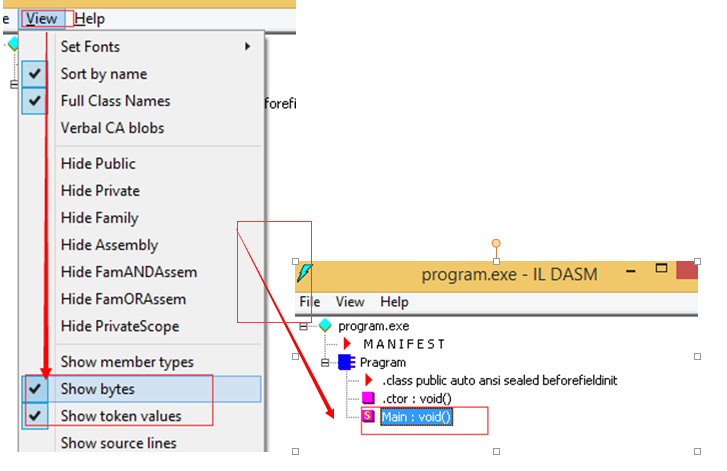
程序的入口点为：0x06000001。06表示标记的类型为MethodDef，000001表示是MethodDef表的第一行。之后通过这个方法定义表的标记，检索MethodDef元数据表。

在ILDasm，View→metaInfo→show打开元数据信息窗口



根据RVA就找到方法在IL代码中的偏移量。

下面我们查看一下Main函数的IL代码

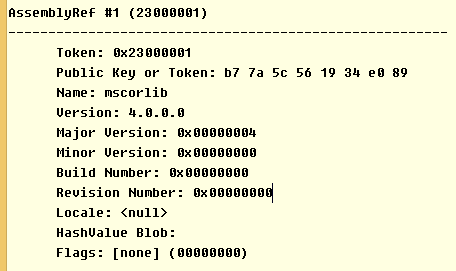


IL代码如下：



## CLR检测入口点方法的代码所引用的类型及方法

在Main执行前，会根据入口函数的所引用的类型和成员引用，加载它们的定义程序集（如果没有加载的话）。例如，上述IL代码包含对System.Console.WriteLine的引用。具体的说，IL call指令引用了元数据token 0A000003，该token表示MemberRef元数据表（表0A）中的记录项3。CLR检查该MembersRef记录项，发现它的字段引用了TypeRef表中的记录项01000004。根据这个TypeRef项，CLR被引导至一个AssemblyRef记录项（23000001）：



这时，就知道了它需要的是哪个程序集，接着，CLR就会定位被加载该程序集。

## 加载引用类型程序集并在内存中创建数据结构

CLR加载mscorlib.dll文件，并扫描元数据来定位Console类型。然后，CLR创建它的内部数据结构来表示类型。

在这个内部数据结构中，Console类型定义的方法每个方法都有一个对应的记录项。每一个记录项都有一个地址，根据这个地址可以找到方法的实现。在这个结构初始化时，每一个记录项被设置成指向CRL内部的一个函数JITCompiler（JIT编译器）

## JIT编译IL为本地代码

CLR创建完成引用类型的内部数据结构后，JIT编译器完成Mian方法的编译，Main方法开始执行。Main方法首次调用WriteLine时，JITComplier函数会被调用（因为WriteLine指向JITComplier函数）。JIT编译器知道要调用的是哪个方法，以及具体是什么类型定义了该方法。然后，JITComplier会在该类型所在的程序集的元数据中查找被调用方法的IL代码所在，接着JITCompliers验证IL代码，并将IL代码编译成本机CPU指令。本机CPU指令被保存到动态分配的内存中。然后，JITComplier回到CLR为类型创建的内部数据结构，找到与被调用的方法对应的那条记录，修改最初对JITComplier的引用，使其指向内存块（包含刚才编译好的本机CPU指令）的地址，最后，从JITComplier函数跳转到内存块中的代码，这些代码正是WriteLine方法的实现。代码执行完毕后并返回时，会回到Main中的代码，并像平常一样继续执行。

当第二次调用WriteLIne时（WriteLine执行的是内存块），这一次，由于已经对WriteLine的代码进行了验证和编译，所以会直接执行内存块中的代码，完全跳过JITComplier函数。Write函数执行完毕后，会再次回到Main继续执行。

## 程序的退出

JIT编译器将本机CPU指令存储到动态内存中。这意味着一旦应用程序终止，编译好的代码会被丢弃。所以，将来再次运行应用程序，或者同时启动应用程序的两个实例，JIT编译器必须再次将IL代码编译成本机指令。这可能显著增加内存耗用，但是，一般而言，JIT编译器造成的性能损失并不显著，因为大多数应用程序都反复调用相同的方法。程序运行期间。这些方法只会对性能造成一次性的影响。