元数据概述：元数据是一种二进制信息，用以对存储在公共语言运行库可移植可执行文件 (PE) 文件或存储在内存中的程序进行描述（总结就是元数据是对于特定文件的描述，而描述其实就是说明某一事物的特性的文字）。将您的**代码编译为 PE 文件**时，便会将元数据插入到该文件的一部分中，而将代码转换为 Microsoft 中间语言 (MSIL) 并将其插入到该文件的另一部分中。在模块或程序集中定义和引用的每个类型和成员都将在元数据中进行说明。当执行代码时，运行库将元数据加载到内存中，并引用它来发现有关代码的类、成员、继承等信息。

元数据以非特定语言的方式描述在代码中定义的每一类型和成员。元数据存储以下信息：

- 程序集的说明。
  - 标识（名称、版本、区域性、公钥）。
  - 导出的类型。
  - 该程序集所依赖的其他程序集。
  - 运行所需的安全权限。
- 类型的说明。
  - 名称、可见性、基类和实现的接口。
  - 成员（方法、字段、属性、事件、嵌套的类型）。
- 属性。
  - 修饰类型和成员的其他说明性元素。

元数据的优点

对于一种更简单的编程模型来说，元数据是关键，该模型不再需要接口定义语言 (IDL) 文件、头文件或任何外部组件引用方法。元数据允许 .NET 语言自动以非特定语言的方式对其自身进行描述，而这是开发人员和用户都无法看见的。另外，通过使用属性，可以对元数据进行扩展。元数据具有以下主要优点：

- 自描述文件

公共语言运行库模块和程序集是自描述的。模块的元数据包含与另一个模块进行交互所需的全部信息。元数据自动提供 COM 中 IDL 的功能，允许将一个文件同时用于定义和实现。运行库模块和程序集甚至不需要向操作系统注册。结果，运行库使用的说明始终反映编译文件中的实际代码，从而提高应用程序的可靠性。

- 语言互用性和更简单的基于组件的设计

元数据提供所有必需的有关已编译代码的信息，以供您从用不同语言编写的 PE 文件中继承类。您可以创建用任何托管语言（任何面向公共语言运行库的语言）编写的任何类的实例，而不用担心显式封送处理或使用自定义的互用代码。

- 属性

.NET Framework 允许您在编译文件中声明特定种类的元数据（称为属性）。在整个 .NET Framework 中到处都可以发现属性的存在，属性用于更精确地控制运行时您的程序如何工

作。另外，您可以通过用户定义的自定义属性向 .NET Framework 文件发出您自己的自定义元数据。有关更多信息，请参见利用属性扩展。

元数据和PE文件结构：

元数据存储在 .NET Framework 可移植可执行文件 (PE) 文件的一个部分中，而 Microsoft 中间语言 (MSIL) 则存储在 PE 文件的另一部分中。文件的**元数据部分包含一系列的表和堆数据结构**。MSIL 部分包含 MSIL 和引用 PE 文件元数据部分的元数据标记。当使用工具（例如，使用 MSIL 反汇编程序 (Ildasm.exe) 来查看代码的 MSIL 或使用运行库调试器 (Cordbg.exe) 来执行内存转储）时，您可能会遇到元数据标记。

元数据表和堆

每个元数据表都保留有关程序元素的信息。例如，一个元数据表说明代码中的类，另一个元数据表说明字段等。如果您的代码中有 10 个类，类表将有 10 行，每行一类。元数据表引用其他的表和堆。例如，类的元数据表引用方法表。

元数据还以四种堆结构存储信息：字符串、Blob、用户字符串和 GUID。所有用于对类型和成员进行命名的字符串都存储在字符串堆中。例如，方法表不直接存储特定方法的名称，而是指向存储在字符串堆中的方法的名称。

元数据标记

元数据标记在 PE 文件的 MSIL 部分中唯一确定每个元数据表的每一行。元数据标记在概念上和指针相似，永久驻留在 MSIL 中，引用特定的元数据表。

元数据标记是一个四个字节的数字。最高位字节表示特定标记（方法、类型等）引用的元数据表。剩下的三个字节指定与所说明的编程元素对应的元数据表中的行。如果您用 C# 定义一个方法并将其编译到 PE 文件，下面的元数据标记可能存在于 PE 文件的 MSIL 部分：

0x06000004	

其中最高位字节 (0x06) 表示这是一个 **MethodDef** 标记。低位的三个字节 (000004) 指示公共语言运行库在 **MethodDef** 表的第四行查找对该方法定义进行描述的信息。

PE 文件中的元数据

当为公共语言运行库编译程序时，该程序转换为由三部分组成的 PE 文件。下表说明了每部分的内容。

PE部分	**PE 部分的内容**
PE 标头	PE 文件主要部分的索引和入口点的地址。 运行库使用该信息确定该文件为 PE 文件并确定当将程序加载到内存时执行从何处开始。
MSIL 指令	组成代码的 Microsoft 中间语言指令 (MSIL)。许多 MSIL 指令带有元数据标记。
元数据	元数据表和堆。运行库使用该部分记录您的代码

	中每个类型和成员的信息。本部分还包括自定义属性和安全性信息。

元数据在运行时的作用：

要更好地理解元数据和它在公共语言运行库中的作用，构造一个简单的程序并说明元数据如何影响它的运行时情况可能很有帮助。下面的代码示例显示名为 MyApp 的类中的两种方法。Main 方法是程序入口点，而 Add 方法只返回两个整数参数的和。

当代码运行时，运行库将模块加载到内存并向元数据咨询该类的信息。加载后，运行库对方法的 Microsoft 中间语言 (MSIL) 流执行广泛的分析，将其转换为快速本机指令。运行库根据需要使用实时 (JIT) 编译器将 MSIL 指令转换为本机代码，每次转换一个方法。

下面的示例显示了从以前代码的 Main 功能生成的部分 MSIL。您可以使用 MSIL 反汇编程序 (Ildasm.exe) 从任何 .NET Framework 应用程序中查看 MSIL 和元数据。

JIT 编译器读取整个方法的 MSIL，对其进行彻底地分析，然后为该方法生成有效的本机指令。在 IL_000d 遇到 Add 方法 (/*06000003 */) 的元数据标记，运行库使用该标记参考 **MethodDef** 表的第三行。

下表显示了说明 Add 方法的元数据标记所引用的 **MethodDef** 表的一部分。虽然程序集中存在其他元数据表并具有它们自己唯一的值，但这里只讨论该表。

Row	相对虚拟地址 (RVA)	ImplFlags	Flags	Name （指向字符串堆。）	Signature（指向 Blob 堆）
1	0x00002050	IL Managed	Public ReuseSlot SpecialName RTSpecialName .ctor	.ctor（构造函数）	
2	0x00002058	IL Managed	Public Static ReuseSlot	Main	String
3	0x0000208c	IL Managed	Public Static ReuseSlot	Add	int, int, int

该表的每一列都包含有关代码的重要信息。**RVA** 列允许运行库计算定义该方法的 MSIL 的起始内存地址。**ImplFlags** 和 **Flags** 列包含说明该方法的位屏蔽（例如，该方法是公共

的还是私有的）。**Name** 列对来自字符串堆的方法的名称进行了索引。**Signature** 列对在 Blob 堆中的方法签名的定义进行了索引。

运行库在第三行的 **RVA** 列计算所需的偏移量地址并将该地址返回到 JIT 编译器，然后，JIT 编译器进入新地址。JIT 编译器继续在新地址处理 MSIL，直到它遇到另一个元数据标记，之后，重复该过程。

使用元数据，运行库可以访问加载代码并将其处理为本机指令所需的所有信息。以这种方式，元数据使自描述文件、公共类型系统和跨语言继承成为可能。