什么是“托管代码”?

使用 .NET 时，我们经常会遇到“托管代码”这个术语。 本文档解释这个术语的含义及其更多相关信息。

简而言之，**托管代码就是执行过程交由运行时(runtime)管理的代码。** 在这种情况下，相关的运行时称为公共语言运行时 (CLR)，不管使用的是哪种实现（例如 Mono、.NET Framework 或 .NET Core/.NET 5+）。 **CLR 负责提取托管代码、将其编译成机器代码，然后执行它。** 除此之外，运行时还提供多个重要服务，例如**自动内存管理、安全边界、类型安全**，等等。

相反，如果**运行 C/C++ 程序，则运行的代码也称为“非托管代码”。 在非托管环境中，程序员需要亲自负责处理相当多的事情。** 实际的程序在本质上是**操作系统 (OS) 载入内存，然后启动的二进制代码(这里的意思是正常的程序的工作只是负责读取内存中的二进制代码，而读取之后所有的东西都是程序来控制的）。** 其他任何工作 - **从内存管理到安全考虑因素 - 对于程序员来说是一个不小的负担。**

托管代码是使用可在 .NET 上运行的一种高级语言（例如 C#、Visual Basic、F# 等）编写的。 **使用相应的编译器编译以这些语言编写的代码时，无法获得机器代码， 而是获得 中间语言 代码，然后运行时会对其进行编译并将其执行。** C++ 是这条规则的一个例外，***因为它也能够生成可在 Windows 上运行的本机非托管二进制代码*。**

中间语言和执行

什么是“中间语言”（简称 IL）？ **中间语言是编译使用高级 .NET 语言编写的代码后获得的结果。** 对使用其中一种语言编写的代码进行编译后，即可获得 IL 所生成的二进制代码。 必须注意，**IL 独立于在运行时顶层运行的任何特定语言（IL是独立的，不管是c#,vb,F#，最终都会编译成IL，而不用在乎IL是什么语言生成的）**；行业甚至为它单独制定了规范，如果有需要，你可以阅读该规范。

从高级代码生成 IL 后，你很有可能想要运行它。 **CLR 此时将接管工作，启动 实时 (JIT) 编译过程，或者将代码从 IL 实时 编译成可以真正在 CPU 上运行的机器代码。** 这样，**CLR 就能确切地知道代码的作用，并可以有效地 *管理* 代码。**

中间语言有时也称为公共中间语言 (CIL) 或 Microsoft 中间语言 (MSIL)。

托管代码互操作性

当然，CLR 允许越过托管与非托管环境之间的边界，同时，即使在基类库中，也有很多代码可以做到这一点。 这称为 互操作性，简称 interop。 例如，使用这些机制可**以包装某个非托管库以及调用该库。** 但是，请务必注意，如果采取这种方法，当**代码越过运行时的边界时，实际的执行管理将再次交接到托管代码**，因而需要遵守相同的限制。

与此类似，**C# 语言可让你利用所谓的 不安全上下文（指定执行过程不由 CLR 管理的代码片段），在代码中直接使用非托管构造，例如指针。**

JIT 编译器和 IL

C# 等较高级的 .NET 语言编译为称为**中间语言 (IL) 的硬件无关性指令集（所谓的指令集就是一条条的指令）。** 应用运行时，**JIT 编译器将 IL 转换为处理器可理解的计算机代码（即就是JIT把IL转化成0和1这些计算机可以直接识别的代码）。** JIT 编译发生在要运行代码的同一台计算机上。

由于 **JIT 编译在应用程序的执行过程中发生，因此编译时间是运行时的一部分（这一段很明确的说明了JIT是CLR的一部分）**。 因此，JIT 编译器需要平衡优化代码所花费的时间与生成代码时可节约的时间。 但 **JIT 编译器知道实际硬件，这样开发人员就无需为不同平台提供不同的实现（即是JIT会根据不同的硬件情况来按照不同的方式来处理IL，而不用上层的开发人员专门去控制了）**。

.NET JIT 编译器可以执行分层编译，这意味着它可以在运行时重新编译各个方法。 通过此功能，它可以快速编译，同时仍然能够为常用方法生成高度优化的代码版本。

有关详细信息，请参阅托管执行过程和分层编译。