Managed Execution Process

Article • 09/15/2021

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. Choosing a compiler.

   To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

2. Compiling your code to MSIL.

   Compiling translates your source code into Microsoft intermediate language (MSIL) and generates the required metadata.

3. Compiling MSIL to native code.

   At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

4. Running code.

   The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

Choosing a Compiler

To obtain the benefits provided by the common language runtime (CLR), you must use one or more language compilers that target the runtime, such as Visual Basic, C#, Visual C++, F#, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.

Because it is a multilanguage execution environment, the runtime supports a wide variety of data types and language features. The language compiler you use determines which runtime features are available, and you design your code using those features. Your compiler, not the runtime, establishes the syntax your code must use. If your component must be completely usable by components written in other languages, your component's exported types must expose only language features that are included in the Common Language Specification (CLS). You can use the CLSCompliantAttribute

attribute to ensure that your code is CLS-compliant. For more information, see Language independence and language-independent components.

Back to top

Compiling to MSIL

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Before code can be run, MSIL must be converted to CPU-specific code, usually by a just-in-time (JIT) compiler. Because the common language runtime supplies one or more JIT compilers for each computer architecture it supports, the same set of MSIL can be JIT-compiled and run on any supported architecture.

When a compiler produces MSIL, it also produces metadata. Metadata describes the types in your code, including the definition of each type, the signatures of each type's members, the members that your code references, and other data that the runtime uses at execution time. The MSIL and metadata are contained in a portable executable (PE) file that is based on and that extends the published Microsoft PE and common object file format (COFF) used historically for executable content. This file format, which accommodates MSIL or native code as well as metadata, enables the operating system to recognize common language runtime images. The presence of metadata in the file together with MSIL enables your code to describe itself, which means that there is no need for type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

Back to top

Compiling MSIL to Native Code

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. .NET provides two ways to perform this conversion:

- A .NET just-in-time (JIT) compiler.
- Ngen.exe (Native Image Generator).

Compilation by the JIT Compiler

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, if your managed code calls platform-specific native APIs or a platform-specific class library, it will run only on that operating system.

JIT compilation takes into account the possibility that some code might never be called during execution. Instead of using time and memory to convert all the MSIL in a PE file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Therefore, subsequent calls to the JIT-compiled method go directly to the native code.

Install-Time Code Generation Using NGen.exe

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it affects performance adversely at run time. In most cases, that diminished performance is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the Ngen.exe (Native Image Generator) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

- It performs the conversion from MSIL to native code before running the application instead of while the application is running.
- It compiles an entire assembly at a time, instead of one method at a time.
- It persists the generated code in the Native Image Cache as a file on disk.

Code Verification

As part of its compilation to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and metadata to find out whether the code is type safe, which means that it accesses only the memory locations it is authorized to access. Type safety helps isolate objects from each other and helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

- A reference to a type is strictly compatible with the type being referenced.
- Only appropriately defined operations are invoked on an object.
- Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.

Back to top

Running Code

The common language runtime provides the infrastructure that enables managed execution to take place and services that can be used during execution. Before a method can be run, it must be compiled to processor-specific code. Each method for which MSIL has been generated is JIT-compiled when it is called for the first time, and then run. The next time the method is run, the existing JIT-compiled native code is run. The process of JIT-compiling and then running the code is repeated until execution is complete.

During execution, managed code receives services such as garbage collection, security, interoperability with unmanaged code, cross-language debugging support, and enhanced deployment and versioning support.

In Microsoft Windows Vista, the operating system loader checks for managed modules by examining a bit in the COFF header. The bit being set denotes a managed module. If the loader detects managed modules, it loads mscoree.dll, and `_CorValidateImage` and `_CorImageUnloading` notify the loader when the managed module images are loaded and unloaded. `_CorValidateImage` performs the following actions:

1. Ensures that the code is valid managed code.

2. Changes the entry point in the image to an entry point in the runtime.

On 64-bit Windows, `_CorValidateImage` modifies the image that is in memory by transforming it from PE32 to PE32+ format.

Back to top

See also

- Overview
- Language independence and language-independent components
- Metadata and Self-Describing Components
- Ilasm.exe (IL Assembler)
- Security
- Interoperating with Unmanaged Code
- Deployment
- Assemblies in .NET
- Application Domains