Profiling Overview - .NET Framework

By tommcdon

Archived: 2026-04-05 14:04:50 UTC

A profiler is a tool that monitors the execution of another application. A common language runtime (CLR) profiler is a dynamic link library (DLL) that consists of functions that receive messages from, and send messages to, the CLR by using the profiling API. The profiler DLL is loaded by the CLR at runtime.

Traditional profiling tools focus on measuring the execution of the application. That is, they measure the time that is spent in each function or the memory usage of the application over time. The profiling API targets a broader class of diagnostic tools such as code-coverage utilities and even advanced debugging aids. These uses are all diagnostic in nature. The profiling API not only measures but also monitors the execution of an application. For this reason, the profiling API should never be used by the application itself, and the application's execution should not depend on (or be affected by) the profiler.

Profiling a CLR application requires more support than profiling conventionally compiled machine code. This is because the CLR introduces concepts such as application domains, garbage collection, managed exception handling, just-in-time (JIT) compilation of code (converting common intermediate language, or CIL, code into native machine code), and similar features. Conventional profiling mechanisms cannot identify or provide useful information about these features. The profiling API provides this missing information efficiently, with minimal effect on the performance of the CLR and the profiled application.

JIT compilation at runtime provides good opportunities for profiling. The profiling API enables a profiler to change the in-memory CIL code stream for a routine before it is JIT-compiled. In this manner, the profiler can dynamically add instrumentation code to particular routines that need deeper investigation. Although this approach is possible in conventional scenarios, it is much easier to implement for the CLR by using the profiling API.

The Profiling API

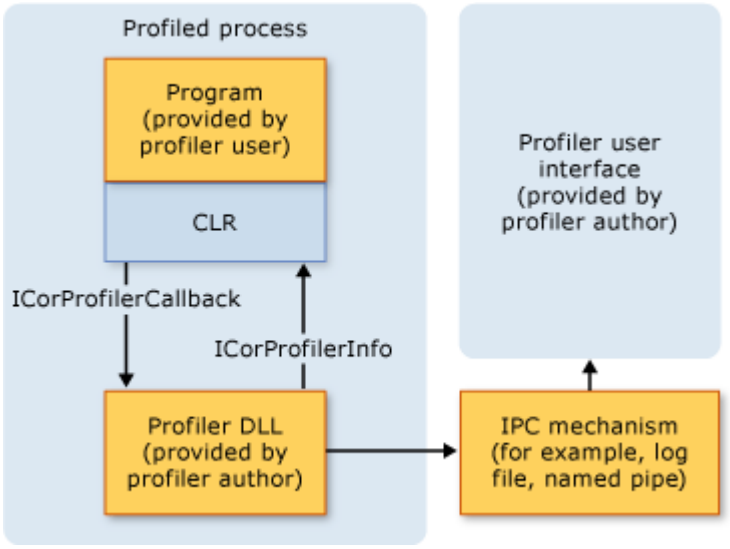
Typically, the profiling API is used to write a *code profiler*, which is a program that monitors the execution of a managed application.

The profiling API is used by a profiler DLL, which is loaded into the same process as the application that is being profiled. The profiler DLL implements a callback interface (ICorProfilerCallback in the .NET Framework version 1.0 and 1.1, ICorProfilerCallback2 in version 2.0 and later). The CLR calls the methods in that interface to notify the profiler of events in the profiled process. The profiler can call back into the runtime by using the methods in the ICorProfilerInfo and ICorProfilerInfo2 interfaces to obtain information about the state of the profiled application.

Note

Only the data-gathering part of the profiler solution should be running in the same process as the profiled application. All user interface and data analysis should be performed in a separate process.

The following illustration shows how the profiler DLL interacts with the application that is being profiled and the CLR.



The Notification Interfaces

ICorProfilerCallback and ICorProfilerCallback2 can be considered notification interfaces. These interfaces consist of methods such as ClassLoadStarted, ClassLoadFinished, and JITCompilationStarted. Each time the CLR loads or unloads a class, compiles a function, and so on, it calls the corresponding method in the profiler's `ICorProfilerCallback` or `ICorProfilerCallback2` interface.

For example, a profiler could measure code performance through two notification functions: FunctionEnter2 and FunctionLeave2. It just time-stamps each notification, accumulates results, and outputs a list that indicates which functions consumed the most CPU or wall-clock time during the execution of the application.

The Information Retrieval Interfaces

The other main interfaces involved in profiling are ICorProfilerInfo and ICorProfilerInfo2. The profiler calls these interfaces as required to obtain more information to help its analysis. For example, whenever the CLR calls the FunctionEnter2 function, it supplies a function identifier. The profiler can get more information about that function by calling the ICorProfilerInfo2::GetFunctionInfo2 method to discover the function's parent class, its name, and so on.

Supported Features

The profiling API provides information about a variety of events and actions that occur in the common language runtime. You can use this information to monitor the inner workings of processes and to analyze the performance of your .NET Framework application.

The profiling API retrieves information about the following actions and events that occur in the CLR:

- CLR startup and shutdown events.
- Application domain creation and shutdown events.
- Assembly loading and unloading events.
- Module loading and unloading events.
- COM vtable creation and destruction events.
- Just-in-time (JIT) compilation and code-pitching events.
- Class loading and unloading events.
- Thread creation and destruction events.
- Function entry and exit events.
- Exceptions.
- Transitions between managed and unmanaged code execution.
- Transitions between different runtime contexts.
- Information about runtime suspensions.
- Information about the runtime memory heap and garbage collection activity.

The profiling API can be called from any (non-managed) COM-compatible language.

The API is efficient with regard to CPU and memory consumption. Profiling does not involve changes to the profiled application that are significant enough to cause misleading results.

The profiling API is useful to both sampling and non-sampling profilers. A *sampling profiler* inspects the profile at regular clock ticks, say, at 5 milliseconds apart. A *non-sampling profiler* is informed of an event synchronously with the thread that causes the event.

Unsupported Functionality

The profiling API does not support the following functionality:

- Unmanaged code, which must be profiled using conventional Win32 methods. However, the CLR profiler includes transition events to determine the boundaries between managed and unmanaged code.
- Self-modifying applications that modify their own code for purposes such as aspect-oriented programming.
- Bounds checking, because the profiling API does not provide this information. The CLR provides intrinsic support for bounds checking of all managed code.
- Remote profiling, which is not supported for the following reasons:

- Remote profiling extends execution time. When you use the profiling interfaces, you must minimize execution time so that profiling results will not be unduly affected. This is especially true when execution performance is being monitored. However, remote profiling is not a limitation when the profiling interfaces are used to monitor memory usage or to obtain runtime information about stack frames, objects, and so on.
- The CLR code profiler must register one or more callback interfaces with the runtime on the local computer on which the profiled application is running. This limits the ability to create a remote code profiler.

Notification Threads

In most cases, the thread that generates an event also executes notifications. Such notifications (for example, FunctionEnter and FunctionLeave) do not need to supply the explicit `ThreadID`. Also, the profiler might decide to use thread-local storage to store and update its analysis blocks instead of indexing the analysis blocks in global storage, based on the `ThreadID` of the affected thread.

Note that these callbacks are not serialized. Users must protect their code by creating thread-safe data structures and by locking the profiler code where necessary to prevent parallel access from multiple threads. Therefore, in certain cases you can receive an unusual sequence of callbacks. For example, assume that a managed application is spawning two threads that are executing identical code. In this case, it is possible to receive a ICorProfilerCallback::JITCompilationStarted event for some function from one thread and a `FunctionEnter` callback from the other thread before receiving the ICorProfilerCallback::JITCompilationFinished callback. In this case, the user will receive a `FunctionEnter` callback for a function that may not have been fully just-in-time (JIT) compiled yet.

Security

A profiler DLL is an unmanaged DLL that runs as part of the common language runtime execution engine. As a result, the code in the profiler DLL is not subject to the restrictions of managed code access security. The only limitations on the profiler DLL are those imposed by the operating system on the user who is running the profiled application.

Profiler authors should take appropriate precautions to avoid security-related issues. For example, during installation, a profiler DLL should be added to an access control list (ACL) so that a malicious user cannot modify it.

Combining Managed and Unmanaged Code in a Code Profiler

An incorrectly written profiler can cause circular references to itself, resulting in unpredictable behavior.

A review of the CLR profiling API may create the impression that you can write a profiler that contains managed and unmanaged components that call each other through COM interop or indirect calls.

Although this is possible from a design perspective, the profiling API does not support managed components. A CLR profiler must be completely unmanaged. Attempts to combine managed and unmanaged code in a CLR profiler may cause access violations, program failure, or deadlocks. The managed components of the profiler will fire events back to their unmanaged components, which would subsequently call the managed components again, resulting in circular references.

The only location where a CLR profiler can call managed code safely is in the common intermediate language (CIL) body of a method. The recommended practice for modifying the CIL body is to use the JIT recompilation methods in the ICorProfilerCallback4 interface.

It is also possible to use the older instrumentation methods to modify CIL. Before the just-in-time (JIT) compilation of a function is completed, the profiler can insert managed calls in the CIL body of a method and then JIT-compile it (see the ICorProfilerInfo::GetILFunctionBody method). This technique can successfully be used for selective instrumentation of managed code, or to gather statistics and performance data about the JIT.

Alternatively, a code profiler can insert native hooks in the CIL body of every managed function that calls into unmanaged code. This technique can be used for instrumentation and coverage. For example, a code profiler could insert instrumentation hooks after every CIL block to ensure that the block has been executed. The modification of the CIL body of a method is a very delicate operation, and there are many factors that should be taken into consideration.

Profiling Unmanaged Code

The common language runtime (CLR) profiling API provides minimal support for profiling unmanaged code. The following functionality is provided:

- Enumeration of stack chains. This feature enables a code profiler to determine the boundary between managed code and unmanaged code.
- Determination whether a stack chain corresponds to managed code or native code.

In the .NET Framework versions 1.0 and 1.1, these methods are available through the in-process subset of the CLR debugging API. They are defined in the CorDebug.idl file.

In the .NET Framework 2.0 and later, you can use the ICorProfilerInfo2::DoStackSnapshot method for this functionality.

Using COM

Although the profiling interfaces are defined as COM interfaces, the common language runtime (CLR) does not actually initialize COM to use these interfaces. The reason is to avoid having to set the threading model by using the CoInitialize function before the managed application has had a chance to specify its desired threading model. Similarly, the profiler itself should not call `CoInitialize`, because it may pick a threading model that is incompatible with the application being profiled and may cause the application to fail.

Call Stacks

The profiling API provides two ways to obtain call stacks: a stack snapshot method, which enables sparse gathering of call stacks, and a shadow stack method, which tracks the call stack at every instant.

Stack Snapshot

A stack snapshot is a trace of the stack of a thread at an instant in time. The profiling API supports the tracing of managed functions on the stack, but it leaves the tracing of unmanaged functions to the profiler's own stack walker.

For more information about how to program the profiler to walk managed stacks, see the ICorProfilerInfo2::DoStackSnapshot method in this documentation set, and Profiler Stack Walking in the .NET Framework 2.0: Basics and Beyond.

Shadow Stack

Using the snapshot method too frequently can quickly create a performance issue. If you want to take stack traces frequently, your profiler should instead build a shadow stack by using the FunctionEnter2, FunctionLeave2, FunctionTailcall2, and ICorProfilerCallback2 exception callbacks. The shadow stack is always current and can quickly be copied to storage whenever a stack snapshot is needed.

A shadow stack may obtain function arguments, return values, and information about generic instantiations. This information is available only through the shadow stack and may be obtained when control is handed to a function. However, this information may not be available later during the run of the function.

Callbacks and Stack Depth

Profiler callbacks may be issued in very stack-constrained circumstances, and a stack overflow in a profiler callback will lead to an immediate process exit. A profiler should make sure to use as little stack as possible in response to callbacks. If the profiler is intended for use against processes that are robust against stack overflow, the profiler itself should also avoid triggering stack overflow.
