Improvements in native code interop in .NET 5.0

Elinor

September 1st, 2020

With .NET 5 scheduled to be released later this year, we thought it would be a good time to discuss some of the interop updates that went into the release and point out some items we are considering for the future.

As we start thinking about what comes next, we are looking for developers and consumers of any interop solutions to discuss their experiences. We are looking for feedback about interop scenarios in general – not just those related to .NET. If you have worked in the interop space, we'd love to hear from you on our GitHub issue.

Some items mentioned in this post are Windows-specific (COM and WinRT). In those cases, 'the runtime' refers only to CoreCLR.

Function pointers

C# function pointers will be coming to C# 9.0, enabling the declaration of function pointers to both managed and unmanaged functions. The runtime had some work to support and complement the interop-related parts of the feature.

UnmanagedCallersOnly

C# function pointers provide a performant way to call native functions from C#. It makes sense for the runtime to provide a symmetrical solution for calling managed functions from native code.

`UnmanagedCallersOnlyAttribute` indicates that a function will be called only from native code, allowing the runtime to reduce the cost of calling the managed function.

To limit the complexity of the scenario, use of this attribute is restricted to methods that must:

- Be `static`
- Only have blittable arguments
  - Removes reliance on any special marshalling logic
- Not be called from managed code
  - Limits the scenarios that need to be handled (e.g. no calls through reflection), allowing the focus to remain on reducing the cost of calling the managed function from native code

A basic usage scenario of passing a managed callback to a native function would, without `UnmanagedCallersOnlyAttribute`, look like:

```
public static int Callback(int i)
{
    // ...
}

private delegate void CallbackDelegate(int i);
private static CallbackDelegate s_callback = new CallbackDelegate(Callback);

[DllImport("NativeLib")]
private static extern void NativeFunctionWithCallback(IntPtr callback);

static void Main()
{
    IntPtr callback = Marshal.GetFunctionPointerForDelegate(s_callback);
    NativeFunctionWithCallback(callback);
}
```

The above requires the allocation of a delegate and the marshalling of that delegate to a function pointer. If the native function being called could hold on to the callback, we also need to ensure the delegate is not garbage collected. This detail is often missed, leading to intermittent “Callback on collected delegate” crashes.

With the combination of function pointers and `UnmanagedCallersOnlyAttribute`, this can be rewritten as:

```
[UnmanagedCallersOnly(CallConvs = new[] { typeof(CallConvCdecl) })]
public static int Callback(int i)
{
    // ...
}

[DllImport("NativeLib")]
private static extern void NativeFunctionWithCallback(delegate* cdecl<int,
int> callback);

static void Main()
{
    // The extra cast is a temporary workaround for Preview 8. It won't be
required in the final version.
    // The syntax will also be updated to use the 'unmanaged' keyword
    // delegate* unmanaged[Cdecl]<int, int> unmanagedPtr = &Callback;
    delegate* cdecl<int, int> unmanagedPtr = (delegate* cdecl<int, int>)
(delegate* <int, int>)&Callback;
    NativeFunctionWithCallback(unmanagedPtr);
}
```

The most obvious change is that the allocation of a delegate is no longer needed. By requiring that the function only have blittable arguments, the runtime does not need to do any marshalling, so the only requirement for entering the function is a GC transition to cooperative mode. The restriction of not allowing the function to be called from managed code means that the JIT-ed function itself can do the GC transition. The function pointer for `Callback` above actually points directly to the JIT-ed function. The extra error-prone code for keeping the delegate alive is no longer needed either.

`System.Private.CoreLib` has started using this attribute for some functions: dotnet/runtime#34270, dotnet/runtime#39082

The `UnmanagedCallersOnlyAttribute` is also supported by the .NET hosting APIs for calling a managed function from a native host.

Caveats:

- The x86 path is less optimized than others (dotnet/runtime#33582).
- Marking a P/Invoke with `UnmanagedCallersOnlyAttribute` is not supported

Resources:

- API: UnmanagedCallersOnlyAttribute
- Proposal: dotnet/runtime#32462
- Implementation: dotnet/runtime#33005, dotnet/runtime#35592
- Prototype of native exports (uses the .NET hosting APIs and `UnmanagedCallersOnlyAttribute` as building blocks): DNNE

Unmanaged calling convention

C# function pointers will allow declaration with an unmanaged calling convention using the `unmanaged` keyword (this syntax is not yet shipped, but will be in the final release). The following will use the platform-dependent default:

```
// Platform-dependent default calling convention
delegate* unmanaged<int, int>;
```

Since the unmanaged function may have a different calling convention from the platform default, the unmanaged calling convention can also be explicitly specified:

```
// cdecl calling convention
delegate* unmanaged[Cdecl] <int, int>;
```

Similarly, a function marked with `UnmanagedCallersOnlyAttribute` can rely on the platform-dependent default or explicitly specify its calling convention:

```
// Platform-dependent default calling convention
[UnmanagedCallersOnly]
public static int Callback(int i) { ... }

// cdecl calling convention
[UnmanagedCallersOnly(CallConvs = new[] { typeof(CallConvCdecl) })]
public static int Callback(int i) { ... }
```

The runtime recognizes the following calling conventions: `CallConvCdecl`, `CallConvFastcall`, `CallConvStdcall`, and `CallConvThiscall`.

As the Roslyn compiler and runtime teams were adding this support, extensibility was a major consideration. The metadata for a method signature has a `CallKind` value that identifies its calling convention (ECMA-335 II.15.3). The new `unmanaged` (0x9) calling convention value, rather than mapping directly to one specific calling convention, indicates that the calling convention can be encoded in the `modopt`s for the return type. To determine the actual calling convention, the runtime will check if the `modopt` values match known calling convention types and use the platform-dependent default if no values match.

With this mechanism in place, the runtime can add support for additional calling conventions in the future without using more values of the calling convention bit. It also allows for a way to encode modified behaviour such as `SuppressGCTransition` (dotnet/runtime#38134).

Resources:

- Proposal: dotnet/runtime#38133
- Implementation: dotnet/runtime#38357, dotnet/runtime#39030
- Method signature metadata: ECMA-335 II.15.3

Low-level APIs for interaction with the built-in interop system

The runtime has a built-in system that handles interop support such as P/Invokes, marshalling, and COM interactions. An underlying theme for interop in .NET 5 has been providing low-level building blocks that enable components outside of the runtime itself to better integrate with the built-in interop system. In .NET 5, we added some APIs that allow for more control over the interop system used in the runtime.

SuppressGCTransition

When executing a P/Invoke, the runtime switches the GC mode from cooperative to preemptive mode. Depending on the scenario, this transition, which also includes an additional frame, can lead to the setup of a P/Invoke being more expensive than the native function that is invoked.

`SuppressGCTransitionAttribute` provides a way for developers to indicate that a P/Invoke should avoid the GC transition. The ability to reduce this interop overhead enables high-performance P/Invoke calls in both runtime libraries and third-party libraries. This is similar in spirit to internal FCalls into the runtime itself.

This attribute effectively circumvents the safeguards normally provided by the runtime around memory management with P/Invokes. As such, it is important to abide by the conditions under which its usage is valid. The native function being called must:

- Always execute for a trivial amount of time (less than 1 microsecond)
- Not perform a blocking syscall (e.g. any type of I/O)
- Not call back into the runtime (e.g. Reverse P/Invoke)
- Not throw exceptions
- Not manipulate locks or other concurrency primitives

`System.Private.CoreLib` has started using this attribute and can be used as examples of valid use cases: dotnet/coreclr#27369, dotnet/runtime#37284, dotnet/runtime#39196, dotnet/runtime#39206.

Caveats:

- This attribute is intended for targeted scenarios. Invalid usage can have serious consequences; blocking operations can result in GC starvation and interactions with the runtime (such as calling back into the runtime or throwing exceptions) can lead to data corruption or runtime termination.
- Using mixed-mode debugging, it will not be possible to set breakpoints in or step into a P/Invoke that has been marked with this attribute.
- This attribute is ignored if the method is not also marked with `DllImport`

Resources:

- API: `SuppressGCTransitionAttribute`
- Proposal: dotnet/runtime#30741
- Implementation: dotnet/coreclr#26458

ComWrappers

On Windows, the Component Object Model (COM) defines a system by which binary components can be exposed and interact with other components and applications. The runtime has a built-in system for interoperating with COM objects, with standard wrapper classes – Runtime Callable Wrappers (RCW) and COM Callable Wrappers (CCW) – for handling the boundary between COM and the .NET runtime.

In .NET 5, we introduced `ComWrappers` as a mechanism for third parties to generate custom wrappers. `ComWrappers` is an abstract class that consumers can subclass in order create wrappers that integrate into the built-in runtime system’s management of object identity and lifetime. It is currently only supported on Windows.

The runtime distinguishes between COM objects by the pointer to the `IUnknown` instance exposed by each object. When getting an RCW for a COM object, the runtime will first check if an RCW already exists for that COM object identity. If an RCW already exists, that RCW will be reused; otherwise, a new RCW will be created. Similarly, the runtime maintains managed object identity. When getting a CCW for a managed object, the runtime will first check if there is already a CCW associated with that managed object. If a CCW already exists, that CCW will be used; otherwise, a new one will be created.

With the `ComWrappers` API, the runtime will continue to handle ensuring that object identity is respected while allowing for integration – through the overrides of `ComWrappers.CreateObject` and `ComWrappers.ComputeVtables` – at the point where it is determined a wrapper needs to be created. For example, if there is a subclass of `ComWrappers` named `MyComWrappers` being used for RCW and CCW creation:

```
var wrappers = new MyComWrappers();

object managedObj = ...
IntPtr ptr1 = wrappers.GetOrCreateComInterfaceForObject(managedObj,
CreateComInterfaceFlags.None);
IntPtr ptr2 = wrappers.GetOrCreateComInterfaceForObject(managedObj,
CreateComInterfaceFlags.None);

IntPtr comObj = ...
object obj1 = wrappers.GetOrCreateObjectForComInstance(comObj,
CreateObjectFlags.None);
object obj2 = wrappers.GetOrCreateObjectForComInstance(comObj,
CreateObjectFlags.None);
```

In the above, `ptr1` and `ptr2` are the same, as are `obj1` and `obj2`. The implementation of `MyComWrappers.ComputeVtables` is invoked only once – as part of the first call to `GetOrCreateComInterfaceForObject`. In the second call, the runtime determines that a CCW already exists for `managedObj` and does not create a new one. Likewise, the implementation of `MyComWrappers.CreateObject` is invoked only once – as part of the first call to `GetOrCreateObjectForComInstance`. In the second call, the runtime determines that an RCW already exists for `comObj` and does not create a new one. This enables the `MyComWrappers` implementation to provide custom wrapper creation while relying on the built-in runtime system for object identity.

The `Reference Tracker` API is an existing system used by the WinRT XAML runtime for managing object lifetime between itself and another runtime. `ComWrappers` provides support for these `Reference Tracker` scenarios to handle object lifetime coordination. When an RCW is created using with `CreateObjectFlags.TrackerObject`, the runtime will check if the COM object implements `IReferenceTracker`. If so, the runtime will get the object's `IReferenceTrackerManager` and update it with the `IReferenceTrackerHost` implemented by the runtime, thus enabling communication and coordination around garbage collection between the runtime and the third party that is implementing `IReferenceTrackerManager`. To handle creation of tracker targets, a `ComWrappers` instance can be registered as a global instance for tracker support through the `ComWrappers.RegisterForTrackerSupport` API. The runtime will then use that instance when its implementation of `IReferenceTrackerHost` receives requests to create a tracker target.

A `ComWrappers` instance can also be registered as a global instance for marshalling in the runtime through the `ComWrappers.RegisterForMarshalling` API. The registered instance is used for wrapper creation as part of COM-related `Marshal` APIs, P/Invokes with COM-related types, and COM activation. Since the registered instance is given priority across all these COM-related marshalling scenarios, it should take care to work well for any potential object – whether that is successfully handling the object or indicating that it cannot. The built-in wrappers will only be used if the registered instance returns a value indicating that it could not create the interface entries or managed object.

A sample demonstrating usage of `ComWrappers` for CCW creation can be found in the dotnet/samples repo. It shows a way to project .NET objects as `IDispatch` instances to a native consumer via an implementation of `ComWrappers`.

Resources:

- API: `ComWrappers`
- Proposal: dotnet/runtime#1845

- Implementation: dotnet/runtime#32091
- Sample: `ComWrappers` subclass for `IDispatch`

IDynamicInterfaceCastable

In .NET, the metadata for a type is static, so whether or not it is possible to cast one type to another type can be determined based on the metadata. The runtime does contain logic for handling special cases (e.g. COM objects) using information beyond the metadata, but there was no general mechanism for a class to participate in the type-cast logic.

`IDynamicInterfaceCastable` exposes a way to create a .NET class that supports interfaces which are not in its metadata. Implementing `IDynamicInterfaceCastable` allows a class to hook into two places:

- Casting (`isinst` and `castclass` instructions) to an interface
  - Calls `IDynamicInterfaceCastable.IsInterfaceImplemented`
- Interface dispatch (virtual stub dispatch)
  - Calls `IDynamicInterfaceCastable.GetInterfaceImplementation`

With the introduction of default implementations in interfaces in C# 8.0, we could require that the type returned by `IDynamicInterfaceCastable.GetInterfaceImplementation` be an interface. This restriction scopes down the breadth of issues around type safety, limiting them to interface dispatch (as opposed to field access on a class). Since interface dispatch goes through the two hooks mentioned above, the runtime can reasonably detect and error on cases where an operation is not supported. We further restricted the type that could be returned by requiring the interface have the `DynamicInterfaceCastableImplementation` attribute. This serves as a declaration of intent for the interface and a mechanism for the IL linker to handle `IDynamicInterfaceCastable` scenarios.

This support of type-casting beyond what is in a type's metadata has some nuance associated with it. Take the (contrived) implementation below:

```
public interface IGreet
{
    void Hello();
    void Goodbye();
}

public class DynamicCastable : IDynamicInterfaceCastable
{
    bool IDynamicInterfaceCastable.IsInterfaceImplemented(RuntimeTypeHandle
interfaceType, bool throwIfNotImplemented)
    {
        // Return true if casting to IGreet
        return interfaceType.Equals(typeof(IGreet).TypeHandle);
    }

    RuntimeTypeHandle
IDynamicInterfaceCastable.GetInterfaceImplementation(RuntimeTypeHandle
interfaceType)
    {
        // Return IGreetImpl type which has a default implementation of
IGreet.Hello
        return typeof(IGreetImpl).TypeHandle;
    }

    [DynamicInterfaceCastableImplementation]
    private interface IGreetImpl : IGreet
    {
        // This method will called based on DynamicCastable's implementation
of IDynamicInterfaceCastable
        void IGreet.Hello()
        {
            // The 'this' pointer here will be a DynamicCastable
            Console.WriteLine($"Hello World from {GetType()}");
        }
    }
}

static void Main()
{
    DynamicCastable obj = new DynamicCastable();

    // Since DynamicCastable implements IDynamicInterfaceCastable, the cast
calls IsInterfaceImplemented on 'obj'
    IGreet greet = (IGreet)obj;

    // Since DynamicCastable.GetInterfaceImplementation returns the IGreetImpl
type, this calls IGreetImpl.Hello()
    greet.Hello();
}
```

In the above program, the cast of `obj` to `IGreet` would succeed based on a call to `IsInterfaceImplemented`. The call to `Hello` would – based on a call to `GetInterfaceImplementation` – resolve the invocation to `IGreetImpl.Hello`. The resulting output would be `Hello World from DynamicCastable`.

The call to `IGreetImpl.Hello` would have a `this` pointer that is the `DynamicCastable` instance, but typed as `IGreetImpl`. The implementation of `IGreetImpl.Hello` could be updated to include:

```
this.Goodbye();
```

When dispatching that call, the runtime would call `GetInterfaceImplementation` and try to resolve the method on the returned `IGreetImpl` type. Since `IGreetImpl` does not have a default implementation of `Goodbye`, this would result in an

EntryPointNotFoundException at execution time.

Implementations of IDynamicInterfaceCastable control casting on a per-instance basis, but interface dispatch on a per-type basis. We can append the following to Main:

```
// Calls IsInterfaceImplemented on 'obj'
IGreet greetAgain = (IGreet)obj;
greetAgain.Hello();

DynamicCastable otherObj = new DynamicCastable();

// Call IsInterfaceImplemented on 'otherObj'
IGreet otherGreet = (IGreet)otherObj;

// Does *not* call GetInterfaceImplementation on 'otherObj'. The previous
resolution from 'obj' instance will be used.
otherGreet.Hello();
```

The result of a cast for an object implementing IDynamicInterfaceCastable is never cached, so the program would call the DynamicCastable implementation of IsInterfaceImplemented for every cast. It is up to the implementation to handle any desired caching for potentially expensive operations. The resolution of a dispatch is cached based on the type, so the resolution of the call to Hello for the otherObj instance of DynamicCastable would use the previous resolution from the obj instance of DynamicCastable. This means than an implementation of GetInterfaceImplementation for a particular type cannot return a different interface type for different instances. If an instance supports an interface, the implementation of that interface must be the same across all instances (that support the interface) of the same type.

Resources:

- API: IDynamicInterfaceCastable, DynamicInterfaceCastableImplementation
- Proposal: dotnet/runtime#36654
- Implementation: dotnet/runtime#37042
- Sample: IDynamicInterfaceCastable implementation

Support for WinRT

The APIs added above provided the basis for improvements to the way WinRT interop works with .NET. They enabled us to support WinRT APIs while de-coupling the WinRT interop system from the .NET runtime itself.

As previously announced, this meant we could remove the built-in support for WinRT interop in .NET 5 (dotnet/runtime#36715). The C#/WinRT tool chain takes advantage of the new APIs and serves as the replacement for that built-in support. This new model enables:

- Development and improvement of WinRT interop separate from the runtime.
- Symmetry with interop systems provided for other operating systems (e.g. iOS and Android).
- Use of NET features such as AOT and IL linking in the WinRT ecosystem.
- Simplification of the runtime codebase (~60k lines of code deleted).

COM objects with the `dynamic` keyword

In .NET Core 3.x and below, the dynamic keyword does not work with COM objects. While the support existed in .NET Framework, the amount of code was large and the logic was complex and specialized, so the support was not included in .NET Core. Thanks to the many developers that let us know how problematic this lack of functionality was for them, we knew we needed to add the support in .NET 5. Using the dynamic keywords for COM objects is now supported (dotnet/runtime#33060).

Marshalling of blittable generics

The runtime did not support marshalling of generic types. An attempt to do so would result in a `MarshalDirectiveException` indicating that generic types cannot be marshalled. In .NET 5, support was added for marshalling of blittable generics in P/Invokes (dotnet/runtime#103). Marshalling of non-blittable generics remains unsupported.

Beyond .NET 5

As we approach the release of .NET 5, we also wanted to provide a glimpse into some of the things we are considering for the future.

Code analyzers

Roslyn code analyzers allow for immediate feedback and guidance that is directly part of a user's development cycle. The .NET guide has information about native interoperability best practices, but the logic around marshalling and interop still remains complex and often confusing.

We expect to start investing more in code analyzers by adding rules for existing features and making sure new rules are considered for any new features.

Source generators

When handling the invocation of a P/Invoke, the runtime will create a stream of IL instructions that is JIT-ed, generating an IL stub. This model tries to be a magic box of marshalling logic that 'just works' and is generally opaque to the developer. However, it does have some significant drawbacks:

- The marshalling system is coupled to the runtime, such that any bug fixes require an update to the entire runtime.
- Since the marshalling code is generated at run time, it is not available for ahead-of-time (AOT) compiler scenarios.
- Debugging the auto-generated marshalling IL stub is difficult for runtime developers and close to impossible for consumers of P/Invokes.

To alleviate these issues, we are planning to use source generators to generate the necessary marshaling code for P/Invokes at compile time. This would allow for independent release and development from the runtime, compatibility with AOT scenarios, and an improved debugging experience. Our initial investigations have been started in dotnet/runtimelab.

Share your experiences

The interop space is as varied as it is complex. Whether you create or consume interop solutions, we are interested in your experiences. We would appreciate your thoughts and comments in our survey on GitHub.

Elinor Fung

Posted in .NET

Read next

ARM64 Performance in .NET 5
ARM64 performance work in .NET 5

Customizing Trimming in .NET 5

Kunal Pathak September 2, 2020

13 comments

This second post on app trimming goes into more detail about to annotate code to control the trimming process.

Sam Spencer September 2, 2020

7 comments

5 comments

Log in to join the discussion.

Tony Henrique September 1, 2020 2:19 pm

Lots of advanced stuff coming to .NET. Keep up the great work!

Log in to Reply

Paulo Pinto September 1, 2020 2:53 pm

Very welcomed improvements, many thanks making .NET into a runtime that can fully take advantage of the platform and reducing the use cases to look for help from C++.

Log in to Reply

Andrew Nosenko September 2, 2020 4:21 am

It's great to see COM gets lots of love with .NET 5.0 release, many thanks for that.
Also, as far as I can tell, *InterfaceType(ComInterfaceType.InterfaceIsIDispatch)* is no longer deprecated in 5.0, which is also a wise move.

Log in to Reply

Mark Adamson September 17, 2020 11:18 am

Will this mean that visual studio tools for office apps can be migrated to .net 5? We have some legacy tools that use it and they are currently stuck on .net framework 4

Log in to Reply

Elinor Fung September 21, 2020 1:01 pm

We have talked about it, but have no plans to upgrade those tools right now.

Log in to Reply

Relevant Links

.NET Download
.NET Hello World
.NET Meetup Events
.NET Documentation
.NET API Browser
.NET SDKs

.NET Application Architecture Guides

Web apps with ASP.NET Core
Mobile apps with Xamarin.Forms
Microservices with Docker Containers
Modernizing existing .NET apps to the cloud

Archive

October 2020
September 2020
August 2020
July 2020
June 2020
May 2020
April 2020
March 2020
February 2020
January 2020
December 2019

Topics

Dot.Net
.NET
.NET Core
.NET Framework
Entity Framework
C#
ML.NET
Visual Studio
F#
WPF
Machine Learning

Stay informed

What's new	Microsoft Store	Education	Enterprise	Developer	Company
Surface Duo	Account profile	Microsoft in education	Azure	Microsoft Visual Studio	Careers
Surface Laptop Go	Download Center	Office for students	AppSource	Windows Dev Center	About Microsoft
Surface Pro X	Microsoft Store support	Office 365 for schools	Automotive	Developer Center	Company news
Surface Go 2	Returns	Deals for students & parents	Government	Microsoft developer program	Privacy at Microsoft