Calling Native Functions from Managed Code

Article • 08/03/2021

The common language runtime provides Platform Invocation Services, or PInvoke, that enables managed code to call C-style functions in native dynamic-linked libraries (DLLs). The same data marshaling is used as for COM interoperability with the runtime and for the "It Just Works," or IJW, mechanism.

For more information, see:

- Using Explicit PInvoke in C++ (DllImport Attribute)
- Using C++ Interop (Implicit PInvoke)

The samples in this section just illustrate how `PInvoke` can be used. `PInvoke` can simplify customized data marshaling because you provide marshaling information declaratively in attributes instead of writing procedural marshaling code.

> **Note**
>
> The marshaling library provides an alternative way to marshal data between native and managed environments in an optimized way. See Overview of Marshaling in C++ for more information about the marshaling library. The marshaling library is usable for data only, and not for functions.

PInvoke and the DllImport Attribute

The following example shows the use of `PInvoke` in a Visual C++ program. The native function puts is defined in msvcrt.dll. The DllImport attribute is used for the declaration of puts.

C++

```
// platform_invocation_services.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("msvcrt", CharSet=CharSet::Ansi)]
extern "C" int puts(String ^);
```

```
int main() {
   String ^ pStr = "Hello World!";
   puts(pStr);
}
```

The following sample is equivalent to the previous sample, but uses IJW.

C++

```
// platform_invocation_services_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

#include <stdio.h>

int main() {
   String ^ pStr = "Hello World!";
   char* pChars =
(char*)Marshal::StringToHGlobalAnsi(pStr).ToPointer();
   puts(pChars);

   Marshal::FreeHGlobal((IntPtr)pChars);
}
```

Advantages of IJW

- There is no need to write `DLLImport` attribute declarations for the unmanaged APIs the program uses. Just include the header file and link with the import library.
- The IJW mechanism is slightly faster (for example, the IJW stubs do not need to check for the need to pin or copy data items because that is done explicitly by the developer).
- It clearly illustrates performance issues. In this case, the fact that you are translating from a Unicode string to an ANSI string and that you have an attendant memory allocation and deallocation. In this case, a developer writing the code using IJW would realize that calling `_putws` and using `PtrToStringChars` would be better for performance.
- If you call many unmanaged APIs using the same data, marshaling it once and passing the marshaled copy is much more efficient than re-marshaling every time.

Disadvantages of IJW

- Marshaling must be specified explicitly in code instead of by attributes (which often have appropriate defaults).
- The marshaling code is inline, where it is more invasive in the flow of the application logic.
- Because the explicit marshaling APIs return `IntPtr` types for 32-bit to 64-bit portability, you must use extra `ToPointer` calls.

The specific method exposed by C++ is the more efficient, explicit method, at the cost of some additional complexity.

If the application uses mainly unmanaged data types or if it calls more unmanaged APIs than .NET Framework APIs, we recommend that you use the IJW feature. To call an occasional unmanaged API in a mostly managed application, the choice is more subtle.

PInvoke with Windows APIs

PInvoke is convenient for calling functions in Windows.

In this example, a Visual C++ program interoperates with the MessageBox function that is part of the Win32 API.

C++

```
// platform_invocation_services_4.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;
typedef void* HWND;
[DllImport("user32", CharSet=CharSet::Ansi)]
extern "C" int MessageBox(HWND hWnd, String ^ pText, String ^ pCap-
tion, unsigned int uType);

int main() {
   String ^ pText = "Hello World! ";
   String ^ pCaption = "PInvoke Test";
   MessageBox(0, pText, pCaption, 0);
}
```

The output is a message box that has the title PInvoke Test and contains the text Hello World!.

The marshaling information is also used by PInvoke to look up functions in the DLL. In user32.dll there is in fact no MessageBox function, but CharSet=CharSet::Ansi enables PInvoke to use MessageBoxA, the ANSI version, instead of MessageBoxW, which is the

Unicode version. In general, we recommend that you use Unicode versions of unmanaged APIs because that eliminates the translation overhead from the native Unicode format of .NET Framework string objects to ANSI.

When Not to Use PInvoke

Using PInvoke is not appropriate for all C-style functions in DLLs. For example, suppose there is a function MakeSpecial in mylib.dll declared as follows:

```
char * MakeSpecial(char * pszString);
```

If we use PInvoke in a Visual C++ application, we might write something similar to the following:

```
C++

[DllImport("mylib")]
extern "C" String * MakeSpecial([MarshalAs(UnmanagedType::LPStr)]
String ^);
```

The difficulty here is that we cannot delete the memory for the unmanaged string returned by MakeSpecial. Other functions called through PInvoke return a pointer to an internal buffer that does not have to be deallocated by the user. In this case, using the IJW feature is the obvious choice.

Limitations of PInvoke

You cannot return the same exact pointer from a native function that you took as a parameter. If a native function returns the pointer that has been marshaled to it by PInvoke, memory corruption and exceptions may ensue.

```
C++

__declspec(dllexport)
char* fstringA(char* param) {
   return param;
}
```

The following sample exhibits this problem, and even though the program may seem to give the correct output, the output is coming from memory that had been freed.

```
C++
```

```
// platform_invocation_services_5.cpp
// compile with: /clr /c
using namespace System;
using namespace System::Runtime::InteropServices;
#include <limits.h>

ref struct MyPInvokeWrap {
public:
   [ DllImport("user32.dll", EntryPoint = "CharLower", CharSet =
CharSet::Ansi) ]
   static String^ CharLower([In, Out] String ^);
};

int main() {
   String ^ strout = "AabCc";
   Console::WriteLine(strout);
   strout = MyPInvokeWrap::CharLower(strout);
   Console::WriteLine(strout);
}
```

Marshaling Arguments

With `PInvoke`, no marshaling is needed between managed and C++ native primitive types with the same form. For example, no marshaling is required between Int32 and int, or between Double and double.

However, you must marshal types that do not have the same form. This includes char, string, and struct types. The following table shows the mappings used by the marshaler for various types:

Expand table

wtypes.h	Visual C++	Visual C++ with /clr	Common language runtime
HANDLE	void *	void *	IntPtr, UIntPtr
BYTE	unsigned char	unsigned char	Byte
SHORT	short	short	Int16
WORD	unsigned short	unsigned short	UInt16
INT	int	int	Int32
UINT	unsigned int	unsigned int	UInt32
LONG	long	long	Int32

wtypes.h	Visual C++	Visual C++ with /clr	Common language runtime
BOOL	long	bool	Boolean
DWORD	unsigned long	unsigned long	UInt32
ULONG	unsigned long	unsigned long	UInt32
CHAR	char	char	Char
LPSTR	char *	String ^ [in], StringBuilder ^ [in, out]	String ^ [in], StringBuilder ^ [in, out]
LPCSTR	const char *	String ^	String
LPWSTR	wchar_t *	String ^ [in], StringBuilder ^ [in, out]	String ^ [in], StringBuilder ^ [in, out]
LPCWSTR	const wchar_t *	String ^	String
FLOAT	float	float	Single
DOUBLE	double	double	Double

The marshaler automatically pins memory allocated on the runtime heap if its address is passed to an unmanaged function. Pinning prevents the garbage collector from moving the allocated block of memory during compaction.

In the example shown earlier in this topic, the CharSet parameter of DllImport specifies how managed Strings should be marshaled; in this case, they should be marshaled to ANSI strings for the native side.

You can specify marshaling information for individual arguments of a native function by using the MarshalAs attribute. There are several choices for marshaling a String * argument: BStr, ANSIBStr, TBStr, LPStr, LPWStr, and LPTStr. The default is LPStr.

In this example, the string is marshaled as a double-byte Unicode character string, LPWStr. The output is the first letter of Hello World! because the second byte of the marshaled string is null, and puts interprets this as the end-of-string marker.

C++

```
// platform_invocation_services_3.cpp
// compile with: /clr
using namespace System;
using namespace System::Runtime::InteropServices;

[DllImport("msvcrt", EntryPoint="puts")]
```

```
extern "C" int puts([MarshalAs(UnmanagedType::LPWStr)] String ^);

int main() {
   String ^ pStr = "Hello World!";
   puts(pStr);
}
```

The MarshalAs attribute is in the System::Runtime::InteropServices namespace. The attribute can be used with other data types such as arrays.

As mentioned earlier in the topic, the marshaling library provides a new, optimized method of marshaling data between native and managed environments. For more information, see Overview of Marshaling in C++.

Performance Considerations

PInvoke has an overhead of between 10 and 30 x86 instructions per call. In addition to this fixed cost, marshaling creates additional overhead. There is no marshaling cost between blittable types that have the same representation in managed and unmanaged code. For example, there is no cost to translate between int and Int32.

For better performance, have fewer PInvoke calls that marshal as much data as possible, instead of more calls that marshal less data per call.

See also

Native and .NET Interoperability