Understanding the marshaling flags: The free-threaded marshaler

devblogs.microsoft.com/oldnewthing/20220622-00

June 22, 2022

Raymond Chen

The free-threaded marshaler is a marshaler that simply provides direct access to the object from any apartment in the process. It is the "nop" marshaler that says "Sure, everybody can access the object directly from any thread without synchronization. Good luck with that!" Of course, the intent is that the free-threaded marshaler is used only by objects that are okay with that.

Whereas marshaling by value was all about dealing with content, the free-threaded marshaler is all about lifetime management. It's sort of the other half of the marshaling coin.

Note that the code I present below is not the *actual* free-threaded marshaler, but it is functionally equivalent and serves as a reasonable reference implementation.

```
bool CanUseFreeThreadedMarshaler(DWORD dwDestContext)
{
    return dwDestContext == MSHCTX_INPROC || dwDestContext == MSHCTX_CROSSCTX;
}
```

The free-threaded marshaler operates within a process, so we use it only if marshaling to a context that is part of the same process. We informally called this the "same-process" group, which consists of the `CROSSCTX` and `INPROC` marshaling contexts. If the context is not a supported one, then we fall back to the standard marshaler.

```
STDMETHODIMP GetUnmarshalClass(
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags,
    CLSID *clsid)
{
    if (CanUseFreeThreadedMarshaler(dwDestContext)) {
        *clsid = CLSID_FreeThreadedUnmarshaler;
        return S_OK;
    }

    ComPtr<IMarshal> marshal;
    RETURN_IF_FAILED(CoGetStandardMarshal(riid, nullptr, dwDestContext,
                                      pvDestContext, mshlflags, &marshal));
    return marshal->GetUnmarshalClass(riid, pv, dwDestContext,
                                  pvDestContext, mshlflags, clsid);
}
```

If the free-threaded marshaler can be used for the destination context, then we return the CLSID of the custom free-threaded unmarshaler class, which will be used to unmarshal the object from the stream.

The data for the free-threaded marshaler is just the raw pointer to the object. We know how to do that, since we did it when marshaling by value. So our initial outline goes like this:

```
STDMETHODIMP GetMarshalSizeMax(
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags,
    LPDWORD size)
{
    if (CanUseFreeThreadedMarshaler(dwDestContext)) {
        ⟦ we're not done yet ⟧
        *size = sizeof(void*);
        return S_OK;
    }

    ComPtr<IMarshal> marshal;
    RETURN_IF_FAILED(CoGetStandardMarshal(riid, CastToUnknown(), dwDestContext,
                                          pvDestContext, mshlflags, &marshal));
    RETURN_IF_FAILED(marshal->GetMarshalSizeMax(riid, pv, dwDestContext,
                                          pvDestContext, mshlflags, size));
    return S_OK;
}

STDMETHODIMP MarshalInterface(
    IStream* pstm,
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags)
{
    if (CanUseFreeThreadedMarshaler(dwDestContext)) {
        ⟦ we're not done yet ⟧
        void* pointer = this;
        RETURN_IF_FAILED(pstm->Write(&pointer, sizeof(pointer), nullptr));
        return S_OK;
    }

    ComPtr<IMarshal> marshal;
    RETURN_IF_FAILED(CoGetStandardMarshal(riid, CastToUnknown(), dwDestContext,
                                          pvDestContext, mshlflags, &marshal));
    RETURN_IF_FAILED(marshal->MarshalInterface(pstm, riid, pv, dwDestContext,
                                          pvDestContext, mshlflags));
    return S_OK;
}

STDMETHODIMP UnmarshalInterface(IStream* pstm, REFIID riid, void** ppv)
{
    *ppv = nullptr;
    ⟦ we're not done yet ⟧
    ULONG actual;
    IUnknown* punk;
    RETURN_IF_FAILED(pstm->Read(&punk, sizeof(punk), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(punk));
    *ppv = punk;
    return S_OK;
}
```

```
    STDMETHODIMP ReleaseMarshalData(IStream* pstm)
    {
        ⟦ we're not done yet ⟧
        ULONG actual;
        IUnknown* punk;
        RETURN_IF_FAILED(pstm->Read(&punk, sizeof(punk), &actual));
        RETURN_HR_IF(E_FAIL, actual != sizeof(*ppv));
        return S_OK;
    }

    STDMETHODIMP DisconnectObject(DWORD dwReserved)
    {
        return E_UNEXPECTED;
    }
```

This is pretty much the same as our marshal-by-value marshaler, except that I've left some spots marked as "we're not done yet" because, well, we're not done yet.

The implementation of `UnmarshalInterface` has a little different structure from what we saw with the marshal-by-value case. When marshaling by value, the object serves as its own unmarshaler, but in the case of the free-threaded marshaler, the unmarshaler is unmarshaling *some other object*. Therefore, we do not end with a `QueryInterface` of the unmarshaler, because we don't want to return the unmarshaler. We want to return the original object, which we put into the caller's `ppv` .

One of the rules for `UnmarshalInterface` is that it is called with the same interface that was originally passed to `MarshalInterface` , so we can just return the original pointer unmodified.

Sidebar: How does COM know what the original interface was? It saves it in the stream! Each block of marshal data is prefixed by some COM metadata that tells it which class to use as the unmarshaler, the interface that was originally marshaled, and some other bookkeeping. You can find the gory details in the specification. Specifically, the header is recorded in a structure known as an `OBJREF`. **End sidebar**.

Unlike the marshal-by-value marshaler, we have to deal with object lifetime and cleanup. Here's a table of object lifetimes based on the marshal flags:

	Weak	**Strong**	**Normal**
`Marshal-Interface`	Capture weak reference	Capture strong reference	Capture strong reference

`Unmarshal-Interface`	Return strong reference	Return strong reference	Return strong reference Release saved reference	
`Release-MarshalData`	Abandon saved reference	Release saved reference		Release saved reference

Let's ignore the *Normal* column for now, because despite the name, it's the abnormal one.

The weak and strong columns are pretty straightforward. To marshal, they capture a weak or strong reference. To unmarshal, they return a strong reference created from the captured reference. And to release, they clean up the weak or strong reference. Cleaning up a weak reference is just abandoning it, whereas cleaning up a strong reference is releasing it.

Okay, now on to the *Normal* column. Recall that for Normal marshaling, the sequence of operations is that the `MarshalInterface` is followed by *either* a call to `UnmarshalInterface` or `ReleaseMarshalData`, but not both. You can imagine that for a normal-marshaled interface, the `UnmarshalInterface` comes with an automatic `ReleaseMarshalData`, so you don't need to (and shouldn't) follow it with an explicit `ReleaseMarshalData`, because that would be a double-destruct.

Now, the `MarshalInterface` method is given the marshal flags, so it knows what kind of reference to write to the stream. But the `UnmarshalInterface` and `ReleaseMarshalData` methods do not receive those flags, so they don't know how to clean up the reference. What can we do?

We'll have to save the marshal flags in the stream as part of our own marshaling data.

Filling in those gaps, then, we have this:

```
bool CanUseFreeThreadedMarshaler(DWORD dwDestContext, DWORD mshlflags)
{
    return (dwDestContext == MSHCTX_INPROC ||
            dwDestContext == MSHCTX_CROSSCTX) &&
           (mshlflags == MSHLFLAGS_NORMAL ||
            mshlflags == MSHLFLAGS_TABLESTRONG ||
            mshlflags == MSHLFLAGS_TABLEWEAK);
}
```

We teach our validity filter about marshal flags and require that the marshal flags be one of the flags we understand.

```
STDMETHODIMP GetMarshalSizeMax(
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags,
    LPDWORD size)
{
    if (CanUseFreeThreadedMarshaler(dwDestContext, mshlflags)) {
        *size = sizeof(mshlflags) + sizeof(void*);
        return S_OK;
    }

    ComPtr<IMarshal> marshal;
    RETURN_IF_FAILED(CoGetStandardMarshal(riid, CastToUnknown(), dwDestContext,
                                          pvDestContext, mshlflags, &marshal));
    RETURN_IF_FAILED(marshal->GetMarshalSizeMax(riid, pv, dwDestContext,
                                          pvDestContext, mshlflags, size));
    return S_OK;
}
```

Our marshal size is now the size of the marshal flags plus the size of a pointer.

```
STDMETHODIMP MarshalInterface(
    IStream* pstm,
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags)
{
    if (CanUseFreeThreadedMarshaler(dwDestContext, mshlflags)) {
        RETURN_IF_FAILED(pstm->Write(&mshlflags, sizeof(mshlflags), nullptr));
        RETURN_IF_FAILED(pstm->Write(&pv, sizeof(pv), nullptr));
        if (mshlflags == MSHLFLAGS_TABLESTRONG || mshlflags == MSHLFLAGS_NORMAL) {
            ((IUnknown*)pv)->AddRef();
        }
        return S_OK;
    }

    ComPtr<IMarshal> marshal;
    RETURN_IF_FAILED(CoGetStandardMarshal(riid, CastToUnknown(), dwDestContext,
                                          pvDestContext, mshlflags, &marshal));
    RETURN_IF_FAILED(marshal->MarshalInterface(pstm, riid, pv, dwDestContext,
                                          pvDestContext, mshlflags));
    return S_OK;
}
```

When marshaling, we write the flags as well as the pointer. If marshaling strong, we take a strong reference. And since normal mode is basically “strong with auto-release”, we take a strong reference in the case of normal mode as well. That means that the only case that *doesn’t* take a strong reference is the weak marshaling, so we can collapse the test against two flags into a negated test against one:

```
STDMETHODIMP MarshalInterface(
    IStream* pstm,
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags)
{
    if (CanUseFreeThreadedMarshaler(dwDestContext, mshlflags)) {
        RETURN_IF_FAILED(pstm->Write(&mshlflags, sizeof(mshlflags), nullptr));
        RETURN_IF_FAILED(pstm->Write(&pv, sizeof(pv), nullptr));
        if (mshlflags != MSHLFLAGS_TABLEWEAK) {
            ((IUnknown*)pv)->AddRef();
        }
        return S_OK;
    }

    ComPtr<IMarshal> marshal;
    RETURN_IF_FAILED(CoGetStandardMarshal(riid, CastToUnknown(), dwDestContext,
                                          pvDestContext, mshlflags, &marshal));
    RETURN_IF_FAILED(marshal->MarshalInterface(pstm, riid, pv, dwDestContext,
                                          pvDestContext, mshlflags));
    return S_OK;
}
```

Next up is unmarshaling.

```
STDMETHODIMP UnmarshalInterface(IStream* pstm, REFIID riid, void** ppv)
{
    *ppv = nullptr;
    ULONG actual;
    DWORD mshlflags;
    RETURN_IF_FAILED(pstm->Read(&mshlflags, sizeof(mshlflags), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(mshlflags));
    IUnknown* punk;
    RETURN_IF_FAILED(pstm->Read(&punk, sizeof(punk), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(punk));
    punk->AddRef();
    if (mshlflags == MSHLFLAGS_NORMAL) {
        punk->Release();
    }
    *ppv = punk;
    return S_OK;
}
```

When unmarshaling, we read out the original marshal flags as well as the original raw pointer. Unmarshaling always produces a strong reference, so we call `AddRef()` on the pointer we are about to return. But in the case where the the interface was marshaled in normal mode, the unmarshaling also comes with an auto-release, so we release the pointer during unmarshaling (since there will be no `ReleaseMarshalData`).

And here is where the optimization for normal mode kicks in: In the case of normal mode, we are performing an `AddRef` immediately followed by a `Release`. These two operations cancel out, so we can bypass them.

```
STDMETHODIMP UnmarshalInterface(IStream* pstm, REFIID riid, void** ppv)
{
    *ppv = nullptr;
    ULONG actual;
    DWORD mshlflags;
    RETURN_IF_FAILED(pstm->Read(&mshlflags, sizeof(mshlflags), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(mshlflags));
    IUnknown* punk;
    RETURN_IF_FAILED(pstm->Read(&punk, sizeof(punk), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(punk));
    if (mshlflags != MSHLFLAGS_NORMAL) {
        punk->AddRef();
    }
    *ppv = punk;
    return S_OK;
}
```

The last operation is releasing the marshal data.

```
STDMETHODIMP ReleaseMarshalData(IStream* pstm)
{
    ULONG actual;
    DWORD mshlflags;
    RETURN_IF_FAILED(pstm->Read(&mshlflags, sizeof(mshlflags), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(mshlflags));
    IUnknown* punk;
    RETURN_IF_FAILED(pstm->Read(&punk, sizeof(punk), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(*ppv));
    if (mshlflags != MSHLFLAGS_TABLEWEAK) {
        punk->Release();
    }
    return S_OK;
}
```

If we are still holding onto a strong reference, then we need to release it. That will be the case if the interface was marshaled strong (in which case it remains strong until explicitly released), or if it was marshaled in normal mode (in which case it is still strong because it hasn't been unmarshaled). We use the same trick as we did when marshaling and testing the negation of the one remaining case, and just check that we aren't in weak mode.

The logic here is fairly straightforward once you understand the rules under which we are operating.

The normal mode optimization is the tricky one that comes into play when you are chasing down a marshaling issue. When the normal mode optimization is in play, unmarshaling the interface will not alter the reference count: Ownership of the strong reference is taken from the marshal data and transferred directly to the code requesting the unmarshaled interface. This means that breakpoints on `AddRef` and `Release` will not trigger, even though ownership of the reference is moving from one place to another. If you're matching up `AddRef` and `Release` calls, you'll see an `AddRef` coming from the marshaling code that seems to be leaked, and a `Release` from the consumer that appears to be an over-release.

Next time, we'll revisit the marshal-by-value marshaler to incorporate the free-threaded marshaler for more efficient intra-process marshaling.

Raymond Chen

Follow