Writing a marshal-by-value marshaler, part 1

devblogs.microsoft.com/oldnewthing/20220617-41

June 17, 2022

Raymond Chen

Last time, we created a skeleton marshaler that does default marshaling. By itself, it's not very interesting, but we can use it as a starting point for implementing a marshal-by-value object.

Marshaling by value is suitable for immutable objects, or at least objects which are logically immutable. The object may internally perform caching to avoid redundant computation, but the external behavior is as if the object were immutable. (Marshaling mutable objects by value can result in surprises when the client calls a mutating method, which causes the original object and its by-value-marshaled copy to fall out of sync.)

Even if your object is immutable, you will need to use the default marshal by reference if you rely on object identity. Marshaling by value creates a clone of the object, which will not be reference-identical with the original.

Okay, so you've decided that you want to marshal by value. For simplicity, let's say that the value in question is just a 32-bit integer.

```
bool ShouldMarshalByValue(DWORD dwDestContext)
{
    return dwDestContext == MSHCTX_CROSSCTX || dwDestContext == MSHCTX_INPROC ||
           dwDestContext == MSHCTX_LOCAL || dwDestContext == MSHCTX_NOSHAREDMEM;
}
```

We choose to use marshal by value for in-process marshaling as well as cross-process marshaling, but not for cross-machine marshaling because our unmarshaler may not be installed on the other machine.

First up is `GetUnmarshalClass` .

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

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

If we decide to marshal by value, then we return the CLSID of the unmarshaler. Here, as is common, the object is its own unmarshaler, so we just ask for another instance of ourselves to be created.

Next comes `GetMarshalSizeMax`. This one is easy because we don't have any variable-sized data.

```
STDMETHODIMP GetMarshalSizeMax(
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags,
    LPDWORD size)
{
    if (ShouldMarshalByValue(dwDestContext)) {
        *size = sizeof(m_value);
        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;
}
```

Marshaling the interface consists of just saving the 32-bit integer to the stream.

```
STDMETHODIMP MarshalInterface(
    IStream* pstm,
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags)
{
    if (ShouldMarshalByValue(dwDestContext)) {
        RETURN_IF_FAILED(pstm->Write(&m_value, sizeof(m_value), 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;
}
```

Note that in all of the above cases, we delegate any unwanted destination contexts to the standard marshaler. This is the recommended behavior, so that the system can add new destination contexts in the future. The documentation for `CoGetStandardMarshal` calls this out as a neat idea, but the documentation for `IMarshal::GetUnmarshalClass` calls it out as an imperative.

That takes care of the marshaling. Now comes the unmarshaling:

```
STDMETHODIMP UnmarshalInterface(IStream* pstm, REFIID riid, void** ppv)
{
    *ppv = nullptr;
    ULONG actual;
    RETURN_IF_FAILED(pstm->Read(&m_value, sizeof(m_value), &actual));
    RETURN_HR_IF(E_FAIL, actual != sizeof(m_value));
    return QueryInterface(riid, ppv);
}
```

We set up this object to be its own unmarshaler, so the unmarshaler reads the 32-bit integer from the stream into its internal state, and then returns the requested interface of itself. In the general case, the unmarshaler is permitted to create a new object or even reuse an existing one.

Exercise: Why don't we need to check `ShouldMarshalByValue` first?

The last group of functions is the cleanup functions.

```
STDMETHODIMP ReleaseMarshalData(IStream* pstm)
{
    RETURN_IF_FAILED(pstm->Seek({ sizeof(m_value), 0 }, STREAM_SEEK_CUR, nullptr));
    return S_OK;
}

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

We have no special state in our marshal data to clean up, so all we have to do is seek over it. The `ReleaseMarshalData` is expected to exit with the stream pointer pointing just past the marshal data. The reason for this is that serializing a data structure is inherently recursive, and COM needs to be able to move on to the next object to be released.

We'll take a digression into stream management before returning to the marshal-by-value marshaler.

Answer to exercise: In the cases where `ShouldMarshalByValue` is false, we delegate to the standard marshaler. The fact that our custom marshaler is active at all means that we must be in the case where `ShouldMarshalByValue` is true.

Raymond Chen