Writing a compound marshaler

devblogs.microsoft.com/oldnewthing/20220621-00

June 21, 2022

Raymond Chen

We left off our discussion of marshaling with a discussion of the recursive nature of marshaling. Let's demonstrate with a simple object that in turn contains other objects.

```
class CompoundObject : public IMarshal /* ... and other interfaces */
{
public:
  // QueryInterface, AddRef, and Release left as an exercise

private:
  int32_t m_value;
  ComPtr<IThing> thing;
};
```

Maybe you decide that this object should be marshaled by shallow copy, so you want to copy the `int32_t` and copy the reference to the `thing`. Therefore, the marshal size is `sizeof(m_value)` plus whatever the marshal size of `thing` turns out to be.[1]

```
  STDMETHODIMP GetMarshalSizeMax(
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD mshlflags,
    LPDWORD size)
  {
    if (ShouldMarshalByValue(dwDestContext)) {
      DWORD thingSize;
      RETURN_IF_FAILED(CoGetMarshalSizeMax(&thingSize, __uuidof(thing.Get()), thing.Get(),
                                           dwDestContext, pvDestContext, mshlflags));
      *size = sizeof(m_value) + thingSize;
      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 copies the integer and then marshals the `thing`:

```
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 CoMarshalInterface(pstm, __uuidof(thing.Get()), thing.Get(),
                              dwDestContext, pvDestContext, mshlflags);
  }

  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;
}
```

Unmarshaling the interface recovers the integer and then unmarshals the `thing`:

```
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_IF_FAILED(CoUnmarshalInterface(IID_PPV_ARGS(&thing));
  return QueryInterface(riid, ppv);
}
```

And releasing the marshal data skips over the integer and then releases the marshal data for the `thing`:

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

Each of the methods that operate on the marshal data must leave the stream pointer at the end of the current marshaler's data, so that the next method can resume where the previous one left off.

So far, we haven't been using the `mshlflags`. That will come into play when our marshal data requires cleanup. We'll investigate that next time.

[1] In practice, I probably would have avoided the temporary variable:

```
RETURN_IF_FAILED(CoGetMarshalSizeMax(size, __uuidof(thing.Get()), thing.Get(),
                                     dwDestContext, pvDestContext, mshlflags));
*size += sizeof(m_value);
```

For expository purposes, I calculated the size by calculate the size of each piece separately and adding them together at the end. This makes the code look a bit more consistent with the other cases that marshal and unmarshal the integer before the inner object.

Raymond Chen

Follow