COM asynchronous interfaces, part 3: Abandoning the operation after a timeout

devblogs.microsoft.com/oldnewthing/20220216-00

February 16, 2022

Raymond Chen

Last time, we learned how to abandon an asynchronous operation. But maybe we don't want to fire and forget so much as wait for a while before finally giving up.

You can check on the completion state of the asynchronous call by using the `ISynchronize` interface on the call object. Today, we're going to use the `Wait` method to wait for the call to complete, with a timeout.

Let's make these changes to our program.

```
int main(int, char**)
{
  winrt::init_apartment(winrt::apartment_type::multi_threaded);

  auto pipe = CreateSlowPipeOnOtherThread();

  winrt::com_ptr<::AsyncIPipeByte> call;
  auto factory = pipe.as<ICallFactory>();
  winrt::check_hresult(factory->CreateCall(
    __uuidof(::AsyncIPipeByte), nullptr,
    __uuidof(::AsyncIPipeByte),
    reinterpret_cast<::IUnknown**>(call.put())));

  printf("Beginning the Push\n");
  BYTE buffer[15] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                      11, 12, 13, 14, 15 };
  winrt::check_hresult(call->Begin_Push(buffer, 15));

  printf("Waiting up to 250ms...\n");
  auto sync = call.as<::ISynchronize>();
  if (sync->Wait(COWAIT_DEFAULT, 250) == S_OK) {
    auto hr = call->Finish_Push();
    printf("Pushed, result is %08x\n", hr);
  } else {
    printf("Took too long!\n");
    // abandon the operation
  }

  Sleep(2000); // so we can see the other thread finish
  return 0;
}
```

This time, instead of abandoning the operation immediately, we ask `ISynchronize::Wait` to wait up to 250ms for the call to complete. If it does, then we call `Finish_Push` to get the result of the `Push()` call. If it doesn't, then we just abandon the operation.

This works, but it could be better. Observe that even though we abandoned the operation, the `SlowPipe` still goes through with the `Push`. Instead of abandoning the operation, we can cancel it, to tell the server that it should stop doing any further work on this operation. The server can call `CoTestCancel()` periodically to see if the operation has been cancelled, and if so, stop and return `RPC_E_CALL_CANCELED`.

```
struct SlowPipe :
    winrt::implements<SlowPipe, ::IPipeByte, winrt::non_agile>
{
  // exit the STA thread when we destruct
  ~SlowPipe() {  PostQuitMessage(0); }

  STDMETHODIMP Pull(BYTE* buffer, ULONG size, ULONG* written)
  {
    HRESULT hr = S_OK;
    printf("Pulling %lu bytes...\n", size);
    ULONG index;
    for (index = 0; index < size / 2; index++) {
      if (CoTestCancel() == RPC_E_CALL_CANCELED) {
          hr = RPC_E_CALL_CANCELED;
          break;
      }
      Sleep(100);
      buffer[index] = 42;
      printf("Pulled byte %lu of %lu\n", index, size);
    }
    *written = index;
    printf("Finished pulling %lu% of %lu bytes, hr = %08x\n",
            index, size, hr);
    return hr;
  }

  STDMETHODIMP Push(BYTE* buffer, ULONG size)
  {
    HRESULT hr = S_OK;
    printf("Pushing %lu bytes...\n", size);
    ULONG index;
    for (index = 0; index < size; index++) {
      if (CoTestCancel() == RPC_E_CALL_CANCELED) {
          hr = RPC_E_CALL_CANCELED;
          break;
      }
      Sleep(100);
      printf("Pushed byte %08x\n", buffer[index]);
    }
    printf("Finished pushing %lu bytes, hr = %08x\n",
           size, hr);
    return hr;
  }
};
```

Our server now periodically checks whether the call was cancelled, and if so, it abandons the remainder of the operation. The `Pull` still reports the partial result, in case the caller cares.

Now we can issue a cancellation from the main thread and see how it alters the behavior of the server.

```
int main(int, char**)
{
  winrt::init_apartment(winrt::apartment_type::multi_threaded);

  auto pipe = CreateSlowPipeOnOtherThread();

  winrt::com_ptr<::AsyncIPipeByte> call;
  auto factory = pipe.as<ICallFactory>();
  winrt::check_hresult(factory->CreateCall(
    __uuidof(::AsyncIPipeByte), nullptr,
    __uuidof(::AsyncIPipeByte),
    reinterpret_cast<::IUnknown**>(call.put())));

  printf("Beginning the Push\n");
  BYTE buffer[15] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                      11, 12, 13, 14, 15 };
  winrt::check_hresult(call->Begin_Push(buffer, 15));

  printf("Waiting up to 250ms...\n");
  auto sync = call.as<::ISynchronize>();
  if (sync->Wait(COWAIT_DEFAULT, 250) == S_OK) {
    auto hr = call->Finish_Push();
    printf("Pushed, result is %08x\n", hr);
  } else {
    printf("Took too long!\n");
    call.as<::ICancelMethodCalls>()->Cancel(0);
  }

  Sleep(2000); // so we can see the other thread finish
  return 0;
}
```

This time, instead of abandoning the operation, we ask `ICancelMethodCalls::Cancel` to cancel it with a timeout of zero, which means "immediately." If you run this version of the program, you'll see that the slow pipe responds to the cancellation by abandoning the operation partway through.

At this point, we realize that we didn't need `ISynchronize` at all. We could just have gone straight to `ICancelMethodCalls::Cancel`, assuming we are willing to accept the fact that the `ICancelMethodCalls::Cancel` method takes the timeout in seconds rather than milliseconds.

```
int main(int, char**)
{
  winrt::init_apartment(winrt::apartment_type::multi_threaded);

  auto pipe = CreateSlowPipeOnOtherThread();

  winrt::com_ptr<::AsyncIPipeByte> call;
  auto factory = pipe.as<ICallFactory>();
  winrt::check_hresult(factory->CreateCall(
    __uuidof(::AsyncIPipeByte), nullptr,
    __uuidof(::AsyncIPipeByte),
    reinterpret_cast<::IUnknown**>(call.put())));

  printf("Beginning the Push\n");
  BYTE buffer[15] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                      11, 12, 13, 14, 15 };
  winrt::check_hresult(call->Begin_Push(buffer, 15));

  printf("Waiting up to 1 second...\n");
  call.as<::ICancelMethodCalls>()->Cancel(1);

  auto hr = call->Finish_Push();
  printf("Pushed, result is %08x\n", hr);

  Sleep(2000); // so we can see the other thread finish
  return 0;
}
```

The `Cancel()` method waits for the timeout, in case the operation completes in time. If not, then it issues a cancellation and returns immediately. It doesn't wait for the server to acknowledge the cancellation; it just issues the cancellation and marks the operation locally as having been cancelled, so that calling the `Finish_` method will return `RPC_E_CALL_CANCELED` immediately.

You can run the program again, but with a 2-second timeout to see the operation run to completion before the timeout elapses.

So far, we've just been sitting around doing nothing while waiting for the operation to complete. Next time, we'll try doing work in parallel.

Raymond Chen

Follow