CODE PROJECT For those who code

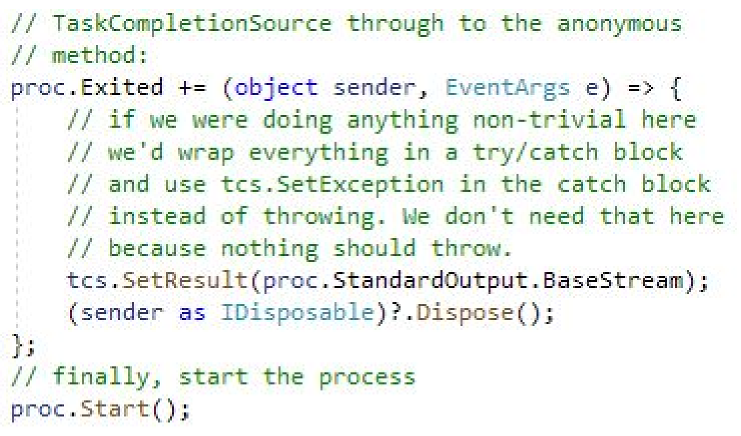
Adapting Event and Callback Based Asynchronicity to the Task Framework

honey the codewitch
20 Feb 2021 MIT

Use TaskCompletionSource to turn an event or callback based model into a Task based one

There are many ways to skin an asynchronous cat. We're going to adapt an event based model to a newer Task based model using the TaskCompletionSource object and just a little bit of witchcraft.

Download TaskCompletion.zip - 6.8 KB



```
// TaskCompletionSource through to the anonymous
// method:
proc.Exited += (object sender, EventArgs e) => {
    // if we were doing anything non-trivial here
    // we'd wrap everything in a try/catch block
    // and use tcs.SetException in the catch block
    // instead of throwing. We don't need that here
    // because nothing should throw.
    tcs.SetResult(proc.StandardOutput.BaseStream);
    (sender as IDisposable)?.Dispose();
};
// finally, start the process
proc.Start();
```

Introduction

.NET's TPL/Task API is a wonderful way to add asynchronicity to your application without a lot of hassle. The compiler support for it in C# makes it even better. However, some APIs still use other means, like callbacks or events to provide asynchronous notification of completion.

In this article we'll explore the `TaskCompletionSource<T>` which allows us to source a `Task<T>` object based on some other form of asynchronous notification.

Conceptualizing This Mess

What a Task<T> is and Isn't

A `Task` is in essence a contract that provides a standard mechanism to await the completion of work, to report errors from that work, and to signal cancelation of that work. That's all it is.

Tasks are not threads. Tasks are again, simply a contract used to signal an asynchronous unit of work. How that work is performed asynchronously is an implementation detail, if it is asynchronous at all. It is possible to do things like signal start and completion of work based on communication from some external hardware for example, which has nothing to do with tying up CPU resources, and may not involve additional threads at all.

Understanding TaskCompletionSource<T>

`TaskCompletionSource<T>` is maybe the most common way to produce a `Task<T>` object to return to a caller of your asynchronous method. These are lightweight objects that simply keep status information for an asynchronous operation in progress.

You use it simply by creating one, and then later when your work is complete you can call `SetResult(T result)`, or `SetException(Exception ex)` in the case of an error. There's no magic in this object. The only cleverness in how to use it comes from you. Your job is to somehow pass the `TaskCompletionSource<T>` around such that you can find it later when your work completes. We can use the hoisting feature available with anonymous methods to do the heavy lifting of passing our `TaskCompletionSource<T>` object to the completion handler. All of this will become clear when we get to the code in a bit. It's much simpler than it sounds.

Once we've created a `TaskCompletionSource<T>` we can use its `Task` property to get a `Task<T>` instance we can return to the caller of our asynchronous method. Of course, if we never call `SetResult()`, `SetException()` or `SetCanceled()` on the `TaskCompletionSource<T>` our task will never return when awaited on, so make sure you're calling one of these in every situation inside your task handling code.

Note: There is no corresponding non-genereric `TaskCompletionSource` object that goes with the non-generic `Task` object used for returning no result. You can simply use `bool` as a dummy `T` value. All `Task<T>` objects can be casted to `Task`. If you're concerned that someone might upcast your `Task` back to `Task<bool>` you can create a private dummy type to use as your `T` argument. In any case, when returning no result you're just going to use `SetResult(default(T))` to return your result, which will never be used.

This is one of those things that's easier to code than to describe. It's also probably easier to read the code than my hamfisted description. There are some caveats to this technique we'll go over after I cover the code, but first, more background.

Lo, a Use Case!

Thankfully, with the introduction of the task framework and the task based TPL model, Microsoft has wrapped most of their base class library with task based asynchronous methods already. However, this made it somewhat difficult for me to find a class to use to demonstrate this technique with that wasn't already wrapped.

Recently, I built on the techniques I outline here in order to adapt an event based asynchronous model in a commercial library to a TPL based one. However, as useful as that is, I don't want to saddle you with a commercial product you have to use in order to follow along.

After a bit of stewing on it, I did think of a class we could experiment on. The `System.Diagnostics.Process` class allows you to run a process and signals process exit (whether completion or error) using an event based model. We'll adapt it to a TPL based one.

It is sometimes useful to be able to execute a command line based process and capture the output in your program. You can do that currently with the `Process` class by either hooking the `Exited` event, or using the `WaitForExit()` method, each providing its own way to signal completion. Of the two, the former is more flexible, because the latter uncoditionally suspends the current thread until the process completes, effectively making it blocking - synchronous.

What we're going to do is provide a single asynchronous method that takes a `ProcessStartInfo` object and returns an (awaitable) `Task<Stream>`. The task completes when the process exists, at which point you can fetch the contents the process wrote to *stdout* as the result of the operation. The code returns a `Stream` rather than a `TextReader`, simply to allow for processes that return binary content like images.

Coding This Mess

Finally, we get to some code.

First, I've made a synchronous version of this method and an asynchronous version. Each uses a different method of signalling the process is complete. The synchronous one of course, blocks. The asynchronous one is awaitable. Other than that, they are (or at least should be) identical in behavior.

We'll cover the synchronous one first, because that way we can go over the basic steps of using a process before making a task based way to use it:

```
static Stream RunWithCapture(ProcessStartInfo startInfo)
{
    // create a process
    var proc = new Process();
    // fill it with what we need to capture
    proc.StartInfo = startInfo;
    proc.EnableRaisingEvents = false;
    startInfo.RedirectStandardOutput = true;
    startInfo.UseShellExecute = false;
    // start doesn't block
    proc.Start();
    // so we use WaitForExit() to block
    proc.WaitForExit();
    // grab our output stream
    var result =proc.StandardOutput.BaseStream;
    // close the process
    proc.Dispose();
    return result;
}
```

Notice the call to `WaitForExit()`, which causes the method to block until the process isn't running anymore.

You call `RunWithCapture()` like this:

```
var psi = new ProcessStartInfo() {
    FileName = "targetapp.exe"
};
using (var stream = RunWithCapture(psi))
{
    // use a streamreader because we want text
    var sr = new StreamReader(stream);
    // no need to dispose it because
    // we are disposing the stream
    Console.Write(sr.ReadToEnd());
}
```

Now let's get to the task based asynchronous one. We need to hook the `Exited` event here instead of calling `WaitForExit()`, since we can't block:

```
static Task<Stream> RunWithCaptureAsync(ProcessStartInfo startInfo)
{
    // we use this as a signal for our custom task
    // we can use SetResult, SetException, and even SetCancelled
    // to signal the completion of a task, with or without errors
    // do not throw exceptions in your async handlers.
    // Use SetException.
    var tcs = new TaskCompletionSource<Stream>();
    try
    {
        // create a process, and set it up for capturing
        // and raising events
        var proc = new Process();
        proc.StartInfo = startInfo;
        proc.EnableRaisingEvents = true;
        startInfo.RedirectStandardOutput = true;
        startInfo.UseShellExecute = false;
        // attach an event handler that signals completion
        // of our task - here Exited serves us well.
        // note we're using hoisting to pass the
        // TaskCompletionSource through to the anonymous
        // method:
        proc.Exited += (object sender, EventArgs e) => {
            // if we were doing anything non-trivial here
            // we'd wrap everything in a try/catch block
            // and use tcs.SetException() in the catch block
            // instead of throwing. We don't need that here
            // because nothing should throw.
            tcs.SetResult(proc.StandardOutput.BaseStream);
            (sender as IDisposable)?.Dispose();
        };
        // finally, start the process
        proc.Start();
    }
    catch (Exception ex)
    {
        // signal an exception
        tcs.SetException(ex);
    }
    // here we return the task to the caller
    return tcs.Task;
}
```

I've commented this heavily. The big deal here, if anything is the `Exited` event handler which we've hooked using an anonymous function. In that handler we simply call `tcs.SetResult()` to the output stream for the process and then `Dispose()` of the process - here represented by `sender`. We got `tcs` through the magic of hoisting, made possible because C# provides it when you use anonymous functions. We basically could not get it to the Exited handler otherwise, unless we used a static member variable, which is ugly. A preferable method to pass the task completion source down would be to use a `state` argument but that's something you typically have with callbacks, not events. Note that you *can* use this overall technique with callbacks as well, with some slight modification.

If you follow the method you'll note that the event handler comes first, but the code therein doesn't get executed until well after the containing method (`RunWithCaptureAsync()`) returns. Remember that `Start()` doesn't block. Instead, we're getting completion signalled not by blocking, but by an event.

Using it is nearly the same as before:

```
var psi = new ProcessStartInfo() {
  FileName = "targetapp.exe"
};
using (var stream = await RunWithCaptureAsync(psi))
{
    var sr = new StreamReader(stream);
    Console.Write(sr.ReadToEnd());
}
```

Caveats

You might wonder why these aren't extension methods for `Process`. The primary reason, at least in the asynchronous method's case is because the `Exited` handler complicates things. For starters, a `Process` object can be recycled, which means the `Exited` event can be fired several times over the life of the object, but our handler expects it only to be fired once. If you use `TrySetResult()` and `TrySetException()` it can at least fail silently on successive calls but it's still not nice at all. If you really need to hook a handler that might be called periodically instead of once, or might be hooked by others, you need to remove your handler's own delegate when your handler runs. This is much harder to do than it sounds, because your handler is an anonymous function, and it needs access to its own delegate. The solution is to use a member variable on the containing class to hold it, but this is ugly. Still, it allows us to call `Exited-=_myExitedHandler` with something meaningful. I did not do that here. Also doing that can cause problems if your code is accessed from multiple threads without synchronizing.

In short, this isn't easy to use for events that are signalled more than once, or if you have to share the object and its event handlers with others.

Hopefully, this technique can help you wrap some of that ugly old eventing or callback based asynchronous code with some more modern awaitable methods.

History

- 19th February, 2021 - Initial submission

License

About the Author

honey the codewitch
United States

Just a shiny lil monster. Casts spells in C++. Mostly harmless.

Comments and Discussions

1 message has been posted for this article Visit **https://www.codeproject.com/Tips/5295013/Adapting-Event-and-Callback-Based-Asynchronicity-t** to post and view comments on this article, or click **here** to get a print view with messages.