Serializing asynchronous operations in C#

devblogs.microsoft.com/oldnewthing/20220912-30

September 12, 2022

Raymond Chen

A serialization pattern for asynchronous operations is making sure only one operation is active at a time, usually because the operation is itself internally stateful, and you don't want multiple operations trying to mutate the state at the same time.

For example, turning a Widget on may require multiple steps, which need to be reversed when turning it off, so we can't just let "turn on" and "turn off" operations run simultaneously. Instead, you want the "turn off" operation to wait for the "turn on" to complete.

The common Windows Runtime libraries for asynchronous activities all use eager-started tasks. The C++ language infrastructure for coroutines supports both eager-started and lazy-started tasks, but the C++/WinRT library implements only eager-started tasks. C# tasks are also eager-started, but you could write your own lazy-started task library.

Let's say we aren't willing to write our own (or borrow an existing) lazy-started task library. We want to work with what we have in hand.

First, let's imagine what the code would be like if we had lazy-started tasks.

```
// C#
object m_lock = new object();
Task m_previousTask = Task.CompletedTask;

Task<T> QueueTaskAsync<T>(LazyTask<T> task)
{
    Task<T> currentTask;
    lock (m_lock) {
        var previousTask = m_previousTask;
        Func<Task<T>> maker = lazy async () => {
            try { await previousTask; } catch (Exception) {}
            return await task;
        };
        m_previousTask = currentTask = maker();
    }
    currentTask.Start();
    return currentTask;
}
```

The idea here is that we atomically add a new task to the previous task, but don't start it right away. That's because we don't want the incoming task running under the lock, which can happen if `previousTask` is an already-completed task.

The new task first awaits the completion of the previous task, and only after the previous task completes (either successfully or with an exception), does the inbound lazy task begin. The new task is itself lazy, and we don't start it until we exit the lock, to ensure that that the inbound lazy task starts outside the lock.

Now to try to translate this into eager tasks.

First of all, the inbound parameter cannot be a `Task`, because that has already been eager-started. One way is to pass a `Func<Task<T>>`: A function that produces a task on demand.

```
Task<T> QueueTaskAsync<T>(Func<Task<T>> taskSource)
{
    Task<T> currentTask;
    lock (m_lock) {
        var previousTask = m_previousTask;
        Func<Task<T>> maker = lazy async () => {
            try { await previousTask; } catch (Exception) {}
            return await taskSource();
        };
        m_previousTask = currentTask = maker();
    }
    currentTask.Start();
    return currentTask;
}
```

In order to create our lazy async task, we can have it immediately await something that we know is not ready, say, a task completion source. Setting the completion source then starts the task.

```
Task<T> QueueTaskAsync<T>(Func<Task<T>> taskSource)
{
    Task<T> currentTask;
    var lazyStart = new TaskCompletionSource<bool>();
    lock (m_lock) {
        var previousTask = m_previousTask;
        Func<Task<T>> maker = async () => {
            await lazyStart.Task;
            try { await previousTask; } catch (Exception) {}
            return await taskSource();
        };
        m_previousTask = currentTask = maker();
    }
    lazyStart.SetResult(true);
    return currentTask;
}
```

Personally, I feel kind of weird awaiting a task and then trying to catch all the exceptions because I worry that I may be catching exceptions that didn't originate from the task itself, but rather originated from the Task Parallel Library itself, such as "Was unable to allocate the memory needed to perform the await." In this case, the `previousTask` is *still running* because it is the await itself that failed, and we end up with two tasks running concurrently after all.

```
Task<T> QueueTaskAsync<T>(Func<Task<T>> taskSource)
{
    Task<T> currentTask;
    var lazyStart = new TaskCompletionSource<bool>();
    lock (m_lock) {
        var previousTask = m_previousTask;
        Func<Task<T>> maker = async () => {
            await lazyStart.Task;
            await previousTask.ContinueWith(t => {});
            return await taskSource();
        };
        m_previousTask = currentTask = maker();
    }
    lazyStart.SetResult(true);
    return currentTask;
}
```

Hooking up the continuation manually means that if any errors occur while trying to wait for the preceding task, it gets captured in the `currentTask` and prevents the `taskSource()` task from running, which is better than allowing two tasks to run simultaneously. Instead,

the caller gets a task which has failed.

With this helper function, you can make your tasks run in sequence, with each one waiting for the previous one to complete (either successfully or with an error) before starting on its own path.

```
Task StartAsync()
{
    return QueueTaskAsync(() => StartWorkerAsync());
}

async Task StartWorkerAsync()
{
    await SetAllDialsToZero();
    await ApplyPower();
    await SlowlyTurnDialsToDesiredLevel();
}

Task StopAsync()
{
    return QueueTaskAsync(() => StopWorkerAsync());
}

async Task StopWorkerAsync()
{
    await SlowlyTurnDialsToZero();
    await RemovePower();
}
```

One thing I was worried about is the fact that we hold onto the last `Task` indefinitely. Could that result in large resources held by the last task not being made available to garbage collection? My experiments suggest that this doesn't happen: When a `Task` completes, the resources held by the code inside the task are made available to GC, even if the `Task` itself remains alive.

Raymond Chen

Follow