On awaiting a task with a timeout in C#

devblogs.microsoft.com/oldnewthing/20220505-00

May 5, 2022

Raymond Chen

Say you have an awaitable object, and you want to await it, but with a timeout. How would you build that?

What you can do is use a `when_any`-like function in combination with a timeout coroutine. For C# this would be something like

```
await Task.WhenAny(
    DoSomethingAsync(),
    Task.Delay(TimeSpan.FromSeconds(1)));
```

The `WhenAny` method completes as soon as any of the passed-in tasks completes. It returns the winner, which you can use to detect whether the operation completed or timed out:

```
var somethingTask = DoSomethingAsync();
var winner = await Task.WhenAny(
    somethingTask,
    Task.Delay(TimeSpan.FromSeconds(1)));
if (winner == somethingTask)
{
    // hooray it worked
}
else
{
    // sad, it timed out
}
```

If the operation produced a result, you'll have to create a timeout task that completes with the same result type, even if you never actually use that result.

```
static async Task<T>
DelayedDummyResultTask<T>(TimeSpan delay)
{
    await Task.Delay(delay);
    return default(T);
}

var somethingTask = GetSomethingAsync();
var winner = await Task.WhenAny(
    somethingTask,
    DelayedDummyResultTask<Something>(TimeSpan.FromSeconds(1)));
if (winner == somethingTask)
{
    // hooray it worked
}
else
{
    // sad, it timed out
}
```

The purpose of the `DelayedDummyResultTask` is not to produce a result, but rather to provide a delay.

We can wrap this up in a helper:

```
static async Task<(bool, Task<T>)>
TaskWithTimeout<T>(
    Task<T> task,
    TimeSpan timeout)
{
    var winner = await Task.WhenAny(
        task, DelayedDummyResultTask<T>(timeout));
    return (winner == task, winner);
}

var (succeeded, task) = await TaskWithTimeout(
    GetProgramAsync(), TimeSpan.FromSeconds(1));
if (succeeded) {
    UseIt(task.Result);
} else {
    // Timed out
}
```

The usage pattern here is still rather clunky, though.

One common pattern is to call the method, but abandon it and return some fallback value instead (typically `false` or `null`):

```
static async Task<T>
DelayedResultTask<T>(TimeSpan delay, T result = default(T))
{
    await Task.Delay(delay);
    return result;
}

static async Task<T>
TaskWithTimeoutAndFallback<T>(
    Task<T> task,
    TimeSpan timeout,
    T fallback = default(T))
{
    return (await Task.WhenAny(
        task, DelayedResultTask<T>(timeout, fallback))).Result;
}
```

This time, our delayed dummy result is no longer a dummy result. If the task times out, then the result of `Task.WhenAny` is the timeout task, and *its* result is what becomes the result of the `TaskWithTimeoutAndFallback`.

Another way of writing the above would be

```
static async Task<T>
TaskWithTimeoutAndFallback<T>(
    Task<T> task,
    TimeSpan timeout,
    T fallback = default(T))
{
    return await await Task.WhenAny(
        task, DelayedResultTask<T>(timeout, fallback));
}
```

which you might choose if only because it give you a rare opportunity to write `await await`.

You could call the function like this:

```
var something = TaskWithTimeoutAndFallback(
    GetSomethingAsync(), TimeSpan.FromSeconds(1));
```

The value in `something` is the result of `GetSomethingAsync()` or `null`.

It might be that the fallback result is expensive to calculate. For example, it `GetSomething-Async` times out, maybe you want to query some alternate database to get the fallback value. So maybe we could have a version where the fallback value is generated lazily.

```
static async Task<T>
DelayedResultTask<T>(TimeSpan delay, Func<T> fallbackMaker)
{
    await Task.Delay(delay);
    return fallbackMaker();
}

static async Task<T>
TaskWithTimeoutAndFallback<T>(
    Task<T> task,
    TimeSpan timeout,
    Func<T> fallbackMaker)
{
    return await await Task.WhenAny(
        task, DelayedResultTask<T>(timeout, fallbackMaker));
}

var something = TaskWithTimeoutAndFallback(
    GetSomethingAsync(), TimeSpan.FromSeconds(1),
    () => LookupSomethingFromDatabase());
```

As a special case, you might want to raise a `TimeoutException` instead of a fallback value. You could do that by passing a lambda that just throws the `TimeoutException` instead of producing a fallback value.

```
var something = TaskWithTimeoutAndFallback(
    GetSomethingAsync(), TimeSpan.FromSeconds(1),
    () => throw TimeoutException());
```

This is probably a common enough pattern that we could provide a special helper for it.

```
static async Task<T>
DelayedTimeoutExceptionTask<T>(TimeSpan delay)
{
    await Task.Delay(delay);
    throw new TimeoutException();
}

static async Task<T>
TaskWithTimeoutAndException<T>(
    Task<T> task,
    TimeSpan timeout)
{
    return await await Task.WhenAny(
        task, DelayedTimeoutExceptionTask<T>(timeout));
}

// throws TimeoutException on timeout
var something = TaskWithTimeoutAndFallback(
    GetSomethingAsync(), TimeSpan.FromSeconds(1));
```

Note that in all of this, the task that timed out continues to run to completion. It's just that we're not paying attention to it any more. If you want to cancel the abandoned task, you need to hook up a task cancellation source when you create it, assuming that's even possible.

In the special case where the `Task` came from a Windows Runtime asynchronous action or operation, you can hook up the cancellation token yourself:

```
var source = new CancellationTokenSource();
var something = TaskWithTimeoutAndFallback(
    o.GetSomethingAsync().AsTask(source.token),
    TimeSpan.FromSeconds(1));
source.Cancel();
source.Dispose();

// see what's in the "something"
```

If you prefer to exit with an exception, then you need to cancel the operation in your timeout handler:

```
var source = new CancellationTokenSource();
try {
    var something = TaskWithTimeoutAndException(
        o.GetSomethingAsync().AsTask(source.token),
        TimeSpan.FromSeconds(1));
} catch (TimeoutException) {
    source.Cancel();
} finally {
    source.Dispose();
}
```

That was a very long discussion, and I haven't even gotten to the original purpose of writing about task cancellation with timeouts, which is to talk about how to do all of this in C++/WinRT. I'm tired, so we'll pick this up next time.

Bonus reading: Crafting a Task.TimeoutAfter Method.

Raymond Chen

Follow