Part of the "Why use F#?" series (more)

Asynchronous programming

Encapsulating a background task with the Async class

24 Apr 2012

Tweet

In this post we'll have a look at a few ways to write asynchronous code in F#, and a very brief example of parallelism as well.

Traditional asynchronous programming

As noted in the previous post, F# can directly use all the usual .NET suspects, such as `Thread` `AutoResetEvent`, `BackgroundWorker` and `IAsyncResult`.

Let's see a simple example where we wait for a timer event to go off:

```
open System

let userTimerWithCallback =
    // create an event to wait on
    let event = new System.Threading.AutoResetEvent(false)

    // create a timer and add an event handler that will signal the event
    let timer = new System.Timers.Timer(2000.0)
    timer.Elapsed.Add (fun _ -> event.Set() |> ignore )

    //start
    printfn "Waiting for timer at %O" DateTime.Now.TimeOfDay
    timer.Start()

    // keep working
    printfn "Doing something useful while waiting for event"

    // block on the timer via the AutoResetEvent
    event.WaitOne() |> ignore

    //done
    printfn "Timer ticked at %O" DateTime.Now.TimeOfDay
```

This shows the use of `AutoResetEvent` as a synchronization mechanism.

- A lambda is registered with the `Timer.Elapsed` event, and when the event is triggered, the AutoResetEvent is signalled.
- The main thread starts the timer, does something else while waiting, and then blocks until the event is triggered.
- Finally, the main thread continues, about 2 seconds later.

The code above is reasonably straightforward, but does require you to instantiate an AutoResetEvent, and could be buggy if the lambda is defined incorrectly.

Introducing asynchronous workflows

F# has a built-in construct called "asynchronous workflows" which makes async code much easier to write. These workflows are objects that encapsulate a background task, and provide a number of useful operations to manage them.

Here's the previous example rewritten to use one:

```
open System
//open Microsoft.FSharp.Control // Async.* is in this module.

let userTimerWithAsync =

    // create a timer and associated async event
    let timer = new System.Timers.Timer(2000.0)
    let timerEvent = Async.AwaitEvent (timer.Elapsed) |> Async.Ignore

    // start
    printfn "Waiting for timer at %O" DateTime.Now.TimeOfDay
    timer.Start()

    // keep working
    printfn "Doing something useful while waiting for event"

    // block on the timer event now by waiting for the async to complete
    Async.RunSynchronously timerEvent

    // done
    printfn "Timer ticked at %O" DateTime.Now.TimeOfDay
```

Here are the changes:

- the `AutoResetEvent` and lambda have disappeared, and are replaced by `let timerEvent = Control.Async.AwaitEvent (timer.Elapsed)` , which creates an `async` object directly from the event, without needing a lambda. The `ignore` is added to ignore the result.

- the `event.WaitOne()` has been replaced by `Async.RunSynchronously timerEvent` which blocks on the async object until it has completed.

That's it. Both simpler and easier to understand.

The async workflows can also be used with `IAsyncResult`, begin/end pairs, and other standard .NET methods.

For example, here's how you might do an async file write by wrapping the `IAsyncResult` generated from `BeginWrite`.

```
let fileWriteWithAsync =

    // create a stream to write to
    use stream = new System.IO.FileStream("test.txt",System.IO.FileMode.Create)

    // start
    printfn "Starting async write"
    let asyncResult = stream.BeginWrite(Array.empty,0,0,null,null)

        // create an async wrapper around an IAsyncResult
    let async = Async.AwaitIAsyncResult(asyncResult) |> Async.Ignore

    // keep working
    printfn "Doing something useful while waiting for write to complete"

    // block on the timer now by waiting for the async to complete
    Async.RunSynchronously async

    // done
    printfn "Async write completed"
```

Creating and nesting asynchronous workflows

Asynchronous workflows can also be created manually. A new workflow is created using the `async` keyword and curly braces. The braces contain a set of expressions to be executed in the background.

This simple workflow just sleeps for 2 seconds.

```
let sleepWorkflow  = async{
    printfn "Starting sleep workflow at %O" DateTime.Now.TimeOfDay
    do! Async.Sleep 2000
    printfn "Finished sleep workflow at %O" DateTime.Now.TimeOfDay
    }

Async.RunSynchronously sleepWorkflow
```

Note: the code `do! Async.Sleep 2000` is similar to `Thread.Sleep` but designed to work with asynchronous workflows.

Workflows can contain *other* async workflows nested inside them. Within the braces, the nested workflows can be blocked on by using the `let!` syntax.

```
let nestedWorkflow  = async{

    printfn "Starting parent"
    let! childWorkflow = Async.StartChild sleepWorkflow

    // give the child a chance and then keep working
    do! Async.Sleep 100
    printfn "Doing something useful while waiting "

    // block on the child
    let! result = childWorkflow

    // done
    printfn "Finished parent"
    }

// run the whole workflow
Async.RunSynchronously nestedWorkflow
```

Cancelling workflows

One very convenient thing about async workflows is that they support a built-in cancellation mechanism. No special code is needed.

Consider a simple task that prints numbers from 1 to 100:

```
let testLoop = async {
    for i in [1..100] do
        // do something
        printf "%i before.." i

        // sleep a bit
        do! Async.Sleep 10
        printfn "..after"
    }
```

We can test it in the usual way:

```
Async.RunSynchronously testLoop
```

Now let's say we want to cancel this task half way through. What would be the best way of doing it?

In C#, we would have to create flags to pass in and then check them frequently, but in F# this technique is built in, using the `CancellationToken` class.

Here an example of how we might cancel the task:

```
open System
open System.Threading

// create a cancellation source
let cancellationSource = new CancellationTokenSource()

// start the task, but this time pass in a cancellation token
Async.Start (testLoop,cancellationSource.Token)

// wait a bit
Thread.Sleep(200)

// cancel after 200ms
cancellationSource.Cancel()
```

In F#, any nested async call will check the cancellation token automatically!

In this case it was the line:

```
do! Async.Sleep(10)
```

As you can see from the output, this line is where the cancellation happened.

Composing workflows in series and parallel

Another useful thing about async workflows is that they can be easily combined in various ways: both in series and in parallel.

Let’s again create a simple workflow that just sleeps for a given time:

```
// create a workflow to sleep for a time
let sleepWorkflowMs ms = async {
    printfn "%i ms workflow started" ms
    do! Async.Sleep ms
    printfn "%i ms workflow finished" ms
    }
```

Here’s a version that combines two of these in series:

```
let workflowInSeries = async {
    let! sleep1 = sleepWorkflowMs 1000
    printfn "Finished one"
    let! sleep2 = sleepWorkflowMs 2000
    printfn "Finished two"
    }

#time
Async.RunSynchronously workflowInSeries
#time
```

And here’s a version that combines two of these in parallel:

```
// Create them
let sleep1 = sleepWorkflowMs 1000
let sleep2 = sleepWorkflowMs 2000

// run them in parallel
#time
[sleep1; sleep2]
    |> Async.Parallel
    |> Async.RunSynchronously
#time
```

Note: The #time command toggles the timer on and off. It only works in the interactive window, so this example must be sent to the interactive window in order to work corrrectly.

We’re using the `#time` option to show the total elapsed time, which, because they run in parallel, is 2 secs. If they ran in series instead, it would take 3 seconds.

Also you might see that the output is garbled sometimes because both tasks are writing to the console at the same time!

This last sample is a classic example of a “fork/join” approach, where a number of a child tasks are spawned and then the parent waits for them all to finish. As you can see, F# makes this very easy!

Example: an async web downloader

In this more realistic example, we’ll see how easy it is to convert some existing code from a non-asynchronous style to an asynchronous style, and the corresponding performance increase that can be achieved.

So here is a simple URL downloader, very similar to the one we saw at the start of the series:

```
open System.Net
open System
open System.IO

let fetchUrl url =
    let req = WebRequest.Create(Uri(url))
    use resp = req.GetResponse()
    use stream = resp.GetResponseStream()
    use reader = new IO.StreamReader(stream)
    let html = reader.ReadToEnd()
    printfn "finished downloading %s" url
```

And here is some code to time it:

```
// a list of sites to fetch
let sites = ["http://www.bing.com";
             "http://www.google.com";
             "http://www.microsoft.com";
             "http://www.amazon.com";
             "http://www.yahoo.com"]

#time                      // turn interactive timer on
sites                      // start with the list of sites
|> List.map fetchUrl       // loop through each site and download
#time                      // turn timer off
```

Make a note of the time taken, and let’s see if we can improve on it!

Obviously the example above is inefficient – only one web site at a time is visited. The program would be faster if we could visit them all at the same time.

So how would we convert this to a concurrent algorithm? The logic would be something like:

- Create a task for each web page we are downloading, and then for each task, the download logic would be something like:
    - Start downloading a page from a website. While that is going on, pause and let other tasks have a turn.
    - When the download is finished, wake up and continue on with the task
- Finally, start all the tasks up and let them go at it!

Unfortunately, this is quite hard to do in a standard C-like language. In C# for example, you have to create a callback for when an async task completes. Managing these callbacks is painful and creates a lot of extra support code that gets in the way of understanding the logic. There are some elegant solutions to this, but in general, the signal to noise ratio for concurrent programming in C# is very high*.

* As of the time of this writing. Future versions of C# will have the `await` keyword, which is similar to what F# has now.

But as you can guess, F# makes this easy. Here is the concurrent F# version of the downloader code:

```
open Microsoft.FSharp.Control.CommonExtensions
                                        // adds AsyncGetResponse

// Fetch the contents of a web page asynchronously
let fetchUrlAsync url =
    async {
        let req = WebRequest.Create(Uri(url))
        use! resp = req.AsyncGetResponse()  // new keyword "use!"
        use stream = resp.GetResponseStream()
        use reader = new IO.StreamReader(stream)
        let html = reader.ReadToEnd()
        printfn "finished downloading %s" url
        }
```

Note that the new code looks almost exactly the same as the original. There are only a few minor changes.

- The change from " `use resp =` " to " `use! resp =` " is exactly the change that we talked about above – while the async operation is going on, let other tasks have a turn.
- We also used the extension method `AsyncGetResponse` defined in the `CommonExtensions` namespace. This returns an async workflow that we can nest inside the main workflow.

- In addition the whole set of steps is contained in the " `async {...}` " wrapper which turns it into a block that can be run asynchronously.

And here is a timed download using the async version.

```
// a list of sites to fetch
let sites = ["http://www.bing.com";
             "http://www.google.com";
             "http://www.microsoft.com";
             "http://www.amazon.com";
             "http://www.yahoo.com"]

#time                      // turn interactive timer on
sites
|> List.map fetchUrlAsync  // make a list of async tasks
|> Async.Parallel          // set up the tasks to run in parallel
|> Async.RunSynchronously  // start them off
#time                      // turn timer off
```

The way this works is:

- `fetchUrlAsync` is applied to each site. It does not immediately start the download, but returns an async workflow for running later.
- To set up all the tasks to run at the same time we use the `Async.Parallel` function
- Finally we call `Async.RunSynchronously` to start all the tasks, and wait for them all to stop.

If you try out this code yourself, you will see that the async version is much faster than the sync version. Not bad for a few minor code changes! Most importantly, the underlying logic is still very clear and is not cluttered up with noise.

Example: a parallel computation

To finish up, let's have another quick look at a parallel computation again.

Before we start, I should warn you that the example code below is just to demonstrate the basic principles. Benchmarks from "toy" versions of parallelization like this are not meaningful, because any kind of real concurrent code has so many dependencies.

And also be aware that parallelization is rarely the best way to speed up your code. Your time is almost always better spent on improving your algorithms. I'll bet my serial version of quicksort against your parallel version of bubblesort any day! (For more details on how to improve performance, see the optimization series (/series/optimization.html))

Anyway, with that caveat, let's create a little task that chews up some CPU. We'll test this serially and in parallel.

```
let childTask() =
    // chew up some CPU.
    for i in [1..1000] do
        for i in [1..1000] do
            do "Hello".Contains("H") |> ignore
            // we don't care about the answer!

// Test the child task on its own.
// Adjust the upper bounds as needed
// to make this run in about 0.2 sec
#time
childTask()
#time
```

Adjust the upper bounds of the loops as needed to make this run in about 0.2 seconds.

Now let's combine a bunch of these into a single serial task (using composition), and test it with the timer:

```
let parentTask =
    childTask
    |> List.replicate 20
    |> List.reduce (>>)

//test
#time
parentTask()
#time
```

This should take about 4 seconds.

Now in order to make the `childTask` parallelizable, we have to wrap it inside an `async` :

```
let asyncChildTask = async { return childTask() }
```

And to combine a bunch of asyncs into a single parallel task, we use `Async.Parallel` .

Let's test this and compare the timings:

```
let asyncParentTask =
    asyncChildTask
    |> List.replicate 20
    |> Async.Parallel

//test
#time
asyncParentTask
|> Async.RunSynchronously
#time
```

On a dual-core machine, the parallel version is about 50% faster. It will get faster in proportion to the number of cores or CPUs, of course, but sublinearly. Four cores will be faster than one core, but not four times faster.

On the other hand, as with the async web download example, a few minor code changes can make a big difference, while still leaving the code easy to read and understand. So in cases where parallelism will genuinely help, it is nice to know that it is easy to arrange.

← 23. Concurrency (/posts/concurrency-intro/)

25. Messages and Agents → (/posts/concurrency-actor-model/)

Written by ScottW (/about/). Found a typo or error? (https://github.com/swlaschin/fsharpforfunandprofit.com) Follow me on Twitter (https://goo.gl/a1j5CS).

Filed under: Concurrency (/category/Concurrency/)

The "Why use F#?" series

Comments

Johan Sivertsen • 5 months ago

Using Fsharp Interactive 4.0 I get:

parentTask()
-> Real time ~ 14 s. CPU time ~ 14s

asyncParentTask |> Async.RunSynchronously |> ignore
-> Real time ~ 49 s. CPU time ~ 6 mins

Any idea what's going on ? Full code below:

let childTask() =
// chew up some CPU.
for i in [1..1000] do
for i in [1..1000] do
do "Hello".Contains("H") |> ignore
// we don't care about the answer!

let parentTask =
childTask

see more

Reply • Share ›

Angus McIntyre • a year ago

I was very confused about the C# task cancellation example. Cooperative cancellation via CancellationToken has been the defacto cancellation method in C# for a while!

Reply • Share ›

scottw Mod → Angus McIntyre • a year ago

Sorry for the confusion -- this post is from 2012 and really needs to be updated :)

Reply • Share ›

Angus McIntyre → scottw • a year ago

Ah, that makes sense. Thanks for the quick response!

Reply • Share ›

Mike Sigsworth • 3 years ago

I get the error **error FS0708: This control construct may only be used if the computation expression builder defines a 'Bind' method** when running this code:

```
let sleepWorkflow  = async{
    printfn "Starting sleep workflow at %O" DateTime.Now
    printfn "Starting sleep workflow at %O" DateTime.Now.
    do! Async.Sleep 2000
    printfn "Finished sleep workflow at %O" DateTime.Now.
}
Async.RunSynchronously sleepWorkflow
```

It doesn't like the `do! Async.Sleep 2000` line. How can I fix this?

^ | ⌄ • Reply • Share ›

scottw Mod ➦ Mike Sigsworth • 3 years ago

When I run that code in F# interactive I don't get any errors. Perhaps try resetting the F# interactive session (from the context menu)? That might clear out any junk in your environment that is causing an error. Hope that helps!

^ | ⌄ • Reply • Share ›

Mike Sigsworth ➦ scottw • 3 years ago

Thanks! Restarting Visual Studio did the trick. ~~(I couldn't find a reset option in the context menu).~~ Nevermind. I found it now. I think I just needed to finish my morning coffee...

1 ^ | ⌄ • Reply • Share ›

Sam Isaacson • 3 years ago

I am struggling with the following in my understanding of Async Workflows and feel it's something fundamental: In the async web downloader example above what would be the difference if you replace `use! resp = req.AsyncGetResponse()` in the `fetchUrlAsync` function with the following line: `use resp = req.GetResponse() `? Both are going to be "blocking things" until they're done at that point anyway so what's the difference? I do understand that the Async version will get run on a different thread, but why go to the hassle of creating it?

^ | ⌄ • Reply • Share ›

David Raab ➦ Sam Isaacson • 3 years ago

The most fundamental difference is that an "async" does not result in its own thread. By default async is a event-loop style programming. Meaning you can create thousands of async and all async can still be handled by a single-thread. The idea is that an async only has non-blocking operations. Instead of blocking a whole thread and let a thread "wait" the control is given to some other code that can run in control is given to some other code that can run in the meantime on the same thread.

Async "can" result in different threads, but they don't must be different threads. So with GetResponse() you block the whole thread. While with AsyncGetResponse() the event-loop could can run some other async on the same thread.

Async are important because they are even more "lighter" than threads. You can create thousands of asyncs without a problem. But creating thousand of threads would be a big problem, and probably even hurt performance instead of making things faster.

1 ^ | ˅ • Reply • Share ›

scottw Mod ➦ David Raab • 3 years ago

"You can create thousands of asyncs without a problem." Just to be clear -- only if they are awaiting IO asynchronously! You could easily create "async"s that aren't actually async, by using GetResponse rather than GetResponseAsync, say. Or Thread.Sleep rather than Async.Sleep. And then they would be just slow as "normal" code!

^ | ˅ • Reply • Share ›

David Raab ➦ scottw • 3 years ago

Yes, absolutely! I thought that was clear when i said "The idea is that an async only has non-blocking operations". GetResponse() or Thread.Sleep() are both blocking operations.

^ | ˅ • Reply • Share ›

scottw Mod ➦ Sam Isaacson • 3 years ago

Yes. this particular flow will be blocked until the response is available. The difference is that by using async, you free up the underlying thread to run other things while you're waiting for a response. And in fact, the response will probably be handled on a different thread. So it's not quite the same as creating a background thread.

Some reasons why this is important are: (a) if you have a web server you can handle more connections at one time (which is why the recent ASP.NET versions encourage async everywhere) (b) if you have 1000's of URLs to fetch, you can fetch them in parallel without creating 1000's of threads. Basically anything involving IO can be made better by using async and reducing load on the thread pool.

Here's a slide deck explaining this:
http://www.slideshare.net/t...
And as to why async is better than callbacks, see
http://tirania.org/blog/arc...

Thanks!

1 ^ | v • Reply • Share ›

Atm • 5 years ago

> the event.WaitOne() has been replaced by Async.RunSynchronously timerEvent which blocks on the async > object until it has completed.

> That's it. Both simpler and easier to understand.

So method called 'Run' doesn't actually run anything, it awaits. I can't understand that, but I can understand why WaitOne blocks thread.

^ | v • Reply • Share ›

George ➦ Atm • a year ago

Async.Run**Synchronously**. Synchronously means on the same time line as in after or next.

^ | v • Reply • Share ›

liammclennan • 5 years ago

The async web downloader is introduced as "a classic example of a "fork/join" approach" but it seems to me that it is missing the "join" part. Each async task downloads a web page, assigns the result to an identifier (html) and then forgets about it. Would this example be better if the results were collected and returned?

^ | v • Reply • Share ›

scottw Mod ➦ liammclennan • 5 years ago

My understanding of fork/join is just that the supervising process starts a bunch of subprocesses/threads/whatever and then waits for them all to finish before continuing. That is what Async.Parallel does, AFAIK.

^ | ˅ • Reply • Share ›

GET THE BOOKS

Reading offline? Download the ebook of this site. (/books/)

GREATEST HITS

New here? Try one of these:
Explore this site (/site-contents/)
Why use F#? (/posts/why-use-fsharp-intro/)
Tips for learning F# (/learning-fsharp/)
Troubleshooting F# (/troubleshooting-fsharp/)
Functional design patterns (talk) (/fppatterns/)
Thinking Functionally (/series/thinking-functionally.html)
Domain modeling with F# (talk) (/ddd/)
Designing with Types (/series/designing-with-types.html)
Railway Oriented Programming (talk) (/rop/) and the original post (/posts/recipe-part2/)
Thirteen ways of looking at a turtle (/posts/13-ways-of-looking-at-a-turtle/)
Understanding Parser Combinators (/parser/)
Roman Numerals Kata (/posts/roman-numeral-kata/)
Choosing properties for property-based testing (/posts/property-based-testing-2/)

RECENT POSTS

Why F# is the best enterprise language (/posts/fsharp-is-the-best-enterprise-language/)
Serializing your domain model (/posts/serializating-your-domain-model/)
Functional approaches to dependency injection (/posts/dependency-injection-1/)
» Archives (/archives/)

MORE

Follow @ScottWlaschin (https://goo.gl/a1j5CS)
About (/about/)
Newsletter (/subscribe.html)
Typos? (https://github.com/swlaschin/fsharpforfunandprofit.com)

SHARE THE LOVE

F# |>I❤ (https://www.dropbox.com/sh/odx68oqh5srur2n/AACc8tZ6-eDIo9YKo-Y5F4hNa?dl=0)

Back to top