How to Serialize Concurrent Operations

search

1

Async/Await

Nested Software Mar 5 *Updated on May 09, 2018*

#javascript **#callbacks** **#promises** **#await**

Overview

This article is about how to specify the order of concurrent operations in JavaScript.

Often we don't care about the order that concurrent operations are completed in. For example, let's say we have a Web server that is processing requests from clients. The time that each request takes can vary, and the order in which the responses are sent back doesn't matter.

However, it's not unusual for situations to arise where we do care about the ordering. Sometimes when we execute an asynchronous operation, we need for it to run to completion

post is about.

There are basically 3 ways to do this in modern JavaScript.

- The oldest way is to use only callbacks. This approach is perhaps conceptually the most pure, but it also can lead to so-called callback hell: A kind of spaghetti code that can be hard to understand and debug.
- Another approach is to use promises, which allows the sequence of operations to be specified in a more procedural manner.
- More recently, JavaScript has introduced `async` and `await`.

> These approaches are not mutually exclusive, but rather are complementary: `async/await` builds on promises, and promises use callbacks.

I'll show a simple example implemented in each of these 3 ways, first with callbacks, then with promises, and finally with async/await.

For this example, we have a hypothetical application that can automatically deploy some custom software to multiple computers concurrently. Let's say that each deployment has 3 steps:

- Install the OS
- Deploy our software
- Run tests

sequence, but they can be executed concurrently across targets (thanks to edA-qa for suggesting this practical example!).

Concurrent Execution

First let's look at some code that runs these tasks concurrently without serializing them at all (unserialized.js):

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const installOS = () => asyncTask("Install OS")

const deploySoftware = () => asyncTask("Deploy Software")

const runTests = () => asyncTask("Run Tests")

const taskDone = (name) => console.log(`Completed async "${name}"`)

const asyncTask = (name) =>  {
    console.log(`Started async "${name}"...`)
    setTimeout(() => taskDone(name), random(1,3) * 1000)
    console.log(`Returning from async "${name}"`)
}

const main = ()=> {
    installOS()
    deploySoftware()
    runTests()
}

main()
```

`setTimeout` to wait between 1 and 3 seconds before completing its task and calling `taskDone`.

Below is a typical output (the actual order will change each time this code is run):

```
C:\dev\asyncio>node unserialized.js
Started async "Install OS"...
Returning from async "Install OS"
Started async "Deploy Software"...
Returning from async "Deploy Software"
Started async "Run Tests"...
Returning from async "Run Tests"
Completed async "Deploy Software"
Completed async "Install OS"
Completed async "Run Tests"
```

As we can see, this is not so good: We deployed our software *before* the OS was even done installing!

Using Callbacks

All right, let's use callbacks to fix this problem (callbacks.js):

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const installOS = (nextTask) => asyncTask("Install OS", nextTask)

const deploySoftware = (nextTask) => asyncTask("Deploy Software", nextTask)

const runTests = () => asyncTask("Run Tests")

const taskDone = (name, nextTask) => {
    console.log(`Completed async "${name}"`)
```

```
    }
}

const asyncTask = (name, nextTask) =>  {
    console.log(`Started async "${name}"...`)
    setTimeout(() => taskDone(name, nextTask),
        random(1,3) * 1000)
    console.log(`Returning from async "${name}"`)
}

const main = ()=> {
    installOS(()=>deploySoftware(()=>runTests()))
}

main()
```

We call `installOS` with a callback that will run `deploySoftware` once `installOS` is done. Once `deploySoftware` is done, it will call its own callback, the `runTests` function.

Each time an operation is done, the `taskDone` function will log the operation as completed and start the next operation.

Let's see if it works:

```
C:\dev\asyncio>node callbacks.js
Started async "Install OS"...
Returning from async "Install OS"
Completed async "Install OS"
Started async "Deploy Software"...
Returning from async "Deploy Software"
Completed async "Deploy Software"
Started async "Run Tests"...
```

Good, we can see that each step happens in order.

However, there are still a number of issues with this code. Even with such a bare-bones example, I think that the code is a little difficult to read.

Error handling is also perhaps not as straightforward as it could be. For example, let's modify the `deploySoftware` function to throw an error:

```
const deploySoftware = (nextTask) => {
    throw new Error('deploying software failed')
    asyncTask("Deploy Software",
    nextTask)
}
```

And let's try to naively wrap our main call with an exception handler:

```
const main = ()=> {
    try {
        installOS(()=>deploySoftware(()=>runTests()))
    } catch (error) {
        console.log(`*** Error caught: '${error}' ***`)
    }
}
```

exception ends up popping the stack:

```
C:\dev\asyncio\callbacks.js:7
        throw new Error('deploying software failed')
        ^

Error: deploying software failed
    at deploySoftware (C:\dev\asyncio\callbacks.js:7:8)
    at installOS (C:\dev\asyncio\callbacks.js:30:17)
    at taskDone (C:\dev\asyncio\callbacks.js:17:3)
    at Timeout.setTimeout [as _onTimeout] (C:\dev\asyncio\callbacks.js:23:19
    at ontimeout (timers.js:458:11)
    at tryOnTimeout (timers.js:296:5)
    at Timer.listOnTimeout (timers.js:259:5)
```

The problem is that `installOS` has already returned by the time the error happens. Clearly some additional effort will have to go into dealing with errors. I'll leave that as an exercise for the reader. As we will see, promises will make the error handling easier.

Using Promises

Let's modify our code slightly to use promises (promises.js):

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const installOS = () => asyncTask("Install OS")

const deploySoftware = () => asyncTask("Deploy Software")

const runTests = () => asyncTask("Run Tests")
```

```
const asyncTask = (name) =>  {
    console.log(`Started async "${name}"...`)

    const promise = new Promise((resolve, reject) => {
        setTimeout(()=>resolve(name), random(1,3) * 1000)
    })

    console.log(`Returning from async "${name}"`)

    return promise
}

const main = ()=> {
    installOS().then(name=>{
        taskDone(name)
        return deploySoftware()
    }).then(name=>{
        taskDone(name)
        return runTests()
    }).then(taskDone)
}

main()
```

We can see that we've been able to remove the `nextTask` callback from our tasks. Now each task can run independently. The job of linking them together has been moved into `main`.

To accomplish this, we've modified `asyncTask` to return a promise.

How does this work? When the results from an asynchronous operation are ready, we call the promise's `resolve` callback. Promises have a method `then` which can be supplied with a callback as a parameter. When we trigger the `resolve`

`then` method.

This allows us to serialize our asynchronous operations. When `installOS` is done, we supply a callback to `then` that calls `deploySoftware`. The `deploySoftware` function returns another promise, which resolves by calling `runTests`. When `runTests` is done, we just supply a trivial callback that just logs the job as done.

By returning promise objects from our tasks, we can chain together the tasks that we want to complete one after the other.

> The details of how promises are implemented are beyond the scope of this post. If you're interested in this topic, I found an interesting article about the design of the Q promise library.

I think this code is easier to read than the callback example.

It also makes it easier to handle errors. Let's again modify `deploySoftware` to throw an error:

```
const deploySoftware = () => {
    throw new Error('"Deploy Software" failed')
    return asyncTask("Deploy Software")
}
```

append a `catch` method to the end of our promise chain:

```
const main = ()=> {
    installOS().then(name=>{
        taskDone(name)
        return deploySoftware()
    }).then(name=>{
        taskDone(name)
        return runTests()
    }).then(taskDone)
    .catch((error)=>console.log(`*** Error caught: '${error}' ***`))
}
```

If an error occurs while trying to resolve a promise, this `catch` method is called.

Let's see what happens when we run this code:

```
C:\dev\asyncio>node serialize_with_promises.js
Started async "Install OS"...
Returning from async "Install OS"
Completed async "Install OS"
*** Error caught: 'Error: "Deploy Software" failed' ***
```

Great, we caught our error! I think this looks much more straightforward than the pure callbacks example.

Using Async/Await

Aync/Await is the last example we'll look at. This syntax works together with promises to make serializing

Okay, no more waiting - let's modify our previous example to use async/await (async_await.js)!

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const installOS = () => asyncTask("Install OS")

const deploySoftware = () => asyncTask("Deploy Software")

const runTests = () => asyncTask("Run Tests")

const taskDone = (name) => console.log(`Completed async "${name}"`)

const asyncTask = (name) =>  {
    console.log(`Started async "${name}"...`)

    const promise = new Promise((resolve, reject) => {
        setTimeout(()=>resolve(name), random(1,3) * 1000)
    })

    console.log(`Returning from async "${name}"`)

    return promise
}

const main = async ()=> {
    const installOSResult = await installOS()
    taskDone(installOSResult)

    const deploySoftwareResult = await deploySoftware()
    taskDone(deploySoftwareResult)

    const runTestsResult = await runTests()
    taskDone(runTestsResult)
}

main()
```

What changes have we made? First, we've labelled `main` as an `async` function. Next, instead of a promise chain, we `await` the results of our asynchronous operations.

`await` will automatically wait for the promise returned by a function to resolve itself. It's non-blocking like all of the code we've looked at today, so other things can run concurrently while an expression is being awaited. However, the next line of code following an `await` won't run until the promise has been resolved. Any function that contains an `await` has to be marked as `async`.

Let's run this code and look at the results:

```
C:\dev\asyncio>async_await.js
Started async "Install OS"...
Returning from async "Install OS"
Completed async "Install OS"
Started async "Deploy Software"...
Returning from async "Deploy Software"
Completed async "Deploy Software"
Started async "Run Tests"...
Returning from async "Run Tests"
Completed async "Run Tests"
```

Great, it works!

We can again make a small change to cause `deploySoftware` to throw an error:

```
const deploySoftware = () => {
    throw new Error('"Deploy Software" failed')
```

Let's see how we can handle this:

```
const main = async ()=> {
    try {
        const installOSResult = await installOS()
        taskDone(installOSResult)

        const deploySoftwareResult = await deploySoftware()
        taskDone(deploySoftwareResult)

        const runTestsResult = await runTests()
        taskDone(runTestsResult)
    } catch(error) {
        console.log(`*** Error caught: '${error}' ***`)
    }
}
```

This works:

```
C:\dev\asyncio>node async_await.js
Started async "Install OS"...
Returning from async "Install OS"
Completed async "Install OS"
*** Error caught: 'Error: "Deploy Software" failed' ***
```

As we can see, async/await makes it possible to use standard synchronous syntax to handle any errors that are produced by our asynchronous code!

code to show that `await` really is non-blocking. Let's add a timer that will run concurrently with our other code:

```
const timer = () => setInterval(()=>console.log('tick'), 500)

const main = async ()=> {
    const t = timer()

    const installOSResult = await installOS()
    taskDone(installOSResult)

    const deploySoftwareResult = await deploySoftware()
    taskDone(deploySoftwareResult)

    const runTestsResult = await runTests()
    taskDone(runTestsResult)

    clearInterval(t)
}
```

Here's the result:

```
C:\dev\asyncio>node async_await.js
Started async "Install OS"...
Returning from async "Install OS"
tick
Completed async "Install OS"
Started async "Deploy Software"...
Returning from async "Deploy Software"
tick
tick
tick
tick
tick
tick
Completed async "Deploy Software"
Started async "Run Tests"...
```

```
tick
Completed async "Run Tests"
```

We can confirm that the timer continues to run while we `await` our tasks. Great!

When using `await`, I think it is helpful to keep in mind that it is roughly equivalent to getting a promise back from the asynchronous call and calling its `then` method.

> Some gotchas: You have to use `await` in a function marked with `async`. That means that you can't `await` something in the top level of JavaScript code. When writing top level code, you can use the `then` syntax of promises instead or wrap the code in a self-executing function that you mark as async.
>
> Also, if you need to run a bunch of tasks concurrently, and you just want to execute some code when *all* of them have finished, see Promise.all

Related:

- Lazy Evaluation in JavaScript with Generators, Map, Filter, and Reduce
- Careful Examination of JavaScript Await
- The Iterators Are Coming! [Symbol.iterator] and [Symbol.asyncIterator] in JavaScript
- Asynchronous Generators and Pipelines in JavaScript

Nested Software + FOLLOW

nestedsoftware

Add to the discussion

PREVIEW SUBMIT

Patrick Cole

Mar 12

Thanks for providing an outline on the progress of synchronous code execution in an asynchronous world! This was super helpful for someone like me who is just getting their head wrapped around async/await.

2

REPLY

Nested Software

Mar 12

Thank you. I really appreciate the positive feedback!

1

REPLY

code of conduct - report abuse

Classic DEV Post from May 20

work day?


Ben Halpern

...

52 36

READ POST SAVE FOR LATER

From one of our Community Sponsors

First Ever DEV Contest: Build a Realtime App with Pusher


Jess Lee

The theme of this contest is to create a real-time app or a hack that uses Pusher Channels real-time API.

281 77

READ POST SAVE FOR LATER

Another Post You Might Like

The Road To PWA - Part 1


Simon Hofmann

The Road To PWA - Progressive Web App Features

59 5

READ POST SAVE FOR LATER

moumni - Jun 1

Building a Pomodoro Timer with Vue.js on CodePen

Tori Pugh - May 31

Play With the React 🌀Router

Sai gowtham - Jun 1

Under the Hood of the Most Powerful Video JavaScript API

The JW Player Team - Jun 1