Ejecución sincrónica vs. ejecución asincrónica

Teoría

Introducción

Como ya vimos en el primer apunte de manejo de archivos, NodeJS nos ofrece múltiples funciones para realizar operaciones sobre archivos con mucha facilidad. Ya hemos visto las versiones **sincrónicas** y nos ha quedado pendiente ver las **asincrónicas**. Pero ¿Qué significa que una función sea *asincrónica*? Veamos un poco de qué se trata.

Ejecución sincrónica

Siempre que escribamos más de una instrucción en un programa, acostumbramos esperar que las instrucciones se ejecuten comenzando desde la primera línea, una por una, de arriba hacia abajo, hasta llegar al final del bloque de código. En el caso de que una de esas instrucciones sea una llamada a otra función, el orden de ejecución se pausa, y se procede a ejecutar las instrucciones dentro de esa función. Sólo una vez ejecutadas todas las instrucciones de esa función, es que el programa retomará con el flujo de instrucciones que venía ejecutando antes.

En todo momento, **sólo se están ejecutando las instrucciones de una sola de las funciones** a la vez. O sea, debe finalizar una función para poder continuar con la otra.

Siguiendo esta idea, el fin de una función marca el inicio de la siguiente, y el fin de ésta, el inicio de la que le sigue, y así sucesivamente, describiendo una secuencia que ocurre en *una única línea de tiempo*.

A está forma de ejecución se la conoce como ***sincrónica***.

Ejemplo

```
function funA(){
    console.log(1)
    funB()
    console.log(2)
}

function funB(){
    console.log(3)
    funC()
    console.log(4)
}
```

```
function funC(){
    console.log(5)
}
```

Al ejecutar la función `funA()` se muestra lo siguiente por pantalla:

```
1
3
5
4
2
```

Comportamiento de una función: bloqueante vs no-bloqueante

Cuando alguna de las instrucciones dentro de una función intente acceder a un recurso que se encuentre fuera del programa (por ejemplo, enviar un mensaje por la red, o leer un archivo del disco) nos encontraremos con dos maneras distintas de hacerlo: en forma bloqueante, o en forma no-bloqueante (*blocking* o *non-blocking*).

Operaciones bloqueantes

Este tipo de operaciones permiten que el programa se comporte de la manera más intuitiva, es decir, siguiendo las reglas establecidas en el punto anterior (ejecución sincrónica).

De esta manera si quisiéramos, por ejemplo, escribir un texto dentro de un archivo, podríamos hacer lo siguiente:

```
import fs from 'fs'

const arch = 'f1.txt'
const texto = 'hola mundo'

console.log('comenzando...')

fs.writeFileSync(arch, texto)
console.log(`${arch} grabado con éxito`)

console.log(`finalizado`)
```

Si lo ejecutamos, obtendremos un resultado similar a éste:

```
comenzando...
f1.txt grabado con éxito
finalizado
```

Operaciones no-bloqueantes

Ahora bien, en algunos casos esperar a que una operación termine para iniciar la siguiente puede no ser la mejor idea, ya que esto podría causar grandes demoras en la ejecución del programa. Es por eso que NodeJS ofrece una segunda opción: las operaciones no bloqueantes.
Este tipo de operaciones permite que, una vez iniciadas, el programa pueda continuar con la siguiente instrucción, sin esperar a que finalice la anterior. O sea, permite la ejecución de varias operaciones **en paralelo**, sucediendo al mismo tiempo. A este tipo de ejecución se la conoce como **asincrónica**.

Ejecución asincrónica

Para poder usar funciones que realicen operaciones no bloqueantes debemos aprender a usarlas adecuadamente, y no generar efectos adversos en forma accidental.

Cuando se trata de código que se ejecuta en forma sincrónica, establecer el orden de ejecución se vuelve tan fácil como decidir qué instrucción escribir primero. Sin embargo, cuando se trata de ejecución asincrónica, sólo sabemos en qué orden comenzarán su ejecución las instrucciones, pero no sabemos en qué momento ni en qué orden terminarán de ejecutarse.
Veamos el siguiente ejemplo, utilizando la versión no-bloqueante de la función de escritura en archivos:

```
import fs from 'fs'

const arch = 'f1.txt'
const texto = 'hola mundo'

console.log('comenzando...')

fs.writeFile(arch, texto, () => {
    console.log(`${arch} grabado con éxito`)
})

console.log(`finalizado`)
```

Si lo ejecutamos, obtendremos lo siguiente:

```
comenzando...
finalizado
f1.txt grabado con éxito
```

Qué pasó acá? Bueno, la explicación corta es la siguiente:
Dado que la operación de escritura es “no-bloqueante”, no bloquea la ejecución normal del programa (no frena la ejecución de las demás instrucciones hasta haberse completado), y permite que el programa se siga ejecutando. Entonces, la ejecución de la operación de escritura “comienza” e inmediatamente cede el control a la siguiente instrucción, que escribe por pantalla el mensaje de finalización. Finalmente, cuando la operación de escritura termina, entonces ejecuta el callback, que en caso de no haberse disparado algún error, informará por pantalla que la escritura se realizó con éxito.

Qué podemos hacer para evitar este comportamiento inesperado?

La solución es anidando las instrucciones dentro de los callbacks, de la siguiente manera:

```
...

fs.writeFile(arch, texto, () => {
    console.log(`${arch} grabado con éxito`)
    console.log(`finalizado`)
})
```

Esto funciona porque lo (único) que podemos controlar en este tipo de operaciones es que el callback *siempre* se ejecuta *luego de finalizar* todas las demás instrucciones involucradas en ese llamado.

Combinar comportamientos sincrónicos y asincrónicos, para exprimir al máximo el potencial del lenguaje, es un tema que abordaremos en las próximas clases.

Si quieren conocer más en profundidad sobre este tema (tan complejo como interesante!) el cual no abordaremos en más detalle durante la cursada, pueden buscar sobre el “event loop”, que es el sistema que administra el orden en que se ejecutan las instrucciones en NodeJS.