TEMA 11. Control asincronía. Callbacks, Promises, Async / Await ...

Desarrollo Web en Entorno Cliente.

Profesor: Juan José Gallego García

Índice :

Asincronía en JS

La asincronía es una de las características más importante en JavaScript para mejorar el rendimiento de una aplicación. En JS existe la posibilidad de obtener datos o ejecutar un determinado código de forma asíncrona mediante funciones que normalmente están basadas en operaciones de entrada/salida o en temporizadores, como por ejemplo leer un archivo, enviar peticiones AJAX, función setTimeout( ) etc..

Esta asincronía significa que JS sigue ejecutando las líneas de código secuencialmente sin esperar a otras funciones (asíncronas) que necesiten más tiempo para ser procesadas con lo que se consigue que la aplicación no quede bloqueada por estas últimas.

Veamos el siguiente código (**Ej1**) que usa un temporizador, y nos vamos a preguntar cuál es su comportamiento antes de ejecutarlo ...

Ej1.

```
<script>
    console.log("dato1");
    setTimeout(function fun1 (){console.log("dato2")},1000);
    console.log("dato3");
 </script>
```

1000 es el tiempo que permanece en la API

Tras ejecutarlo podemos observar que el comportamiento no es el esperado tal y como sería en otro lenguaje, es decir, primero se escribiría dato1 la segunda instrucción esperaría 1 segundo y se escribiría dato2 y finalmente dato3, pero esto no ocurre, dato2 se escribe en último lugar dado que pertenece a una función asíncrona, y JS continúa con la ejecución de las líneas de código sin esperar que se procese dicha función.

Es muy importante conocer cómo funciona el modelo JS para desarrollar aplicaciones no bloqueantes con ayuda de las Web APIs del navegador (llamadas a funciones asíncronas) que no se ejecutan en el mismo hilo que el programa principal (runtime).

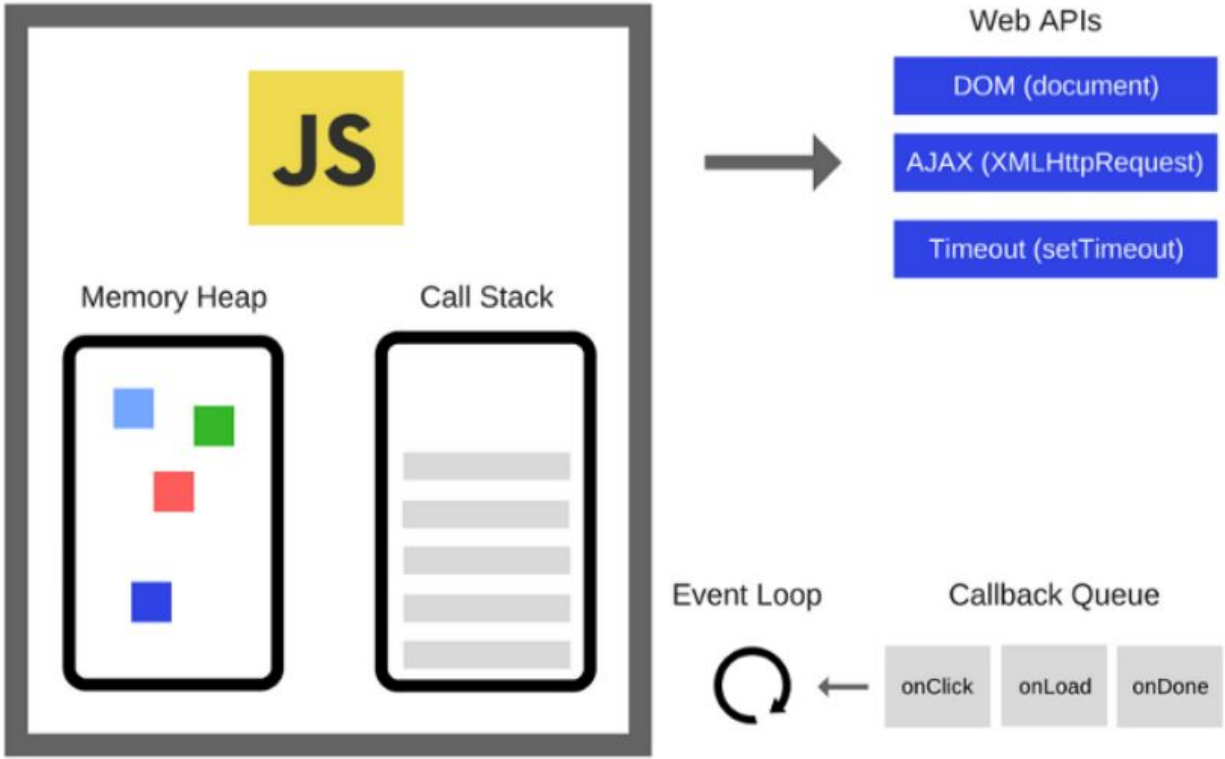
Modelo de JS

JS fue diseñado para navegadores, para interactuar con el usuario y para intercambiar datos a través de la red, todo ello de la forma más fluida posible. Para conseguir estos objetivos JS posee las siguientes características:

- Utiliza un modelo asíncrono, no bloqueante.
- Implementado en un sólo hilo de ejecución (thread).
- Altamente concurrente.

Por ejemplo para Chrome y Node.js el motor de JS es V8, que es un intérprete de código javascript escrito en C++.

A continuación podemos ver un diagrama que muestra los elementos que componen el sistema de ejecución principal (runtime) y las Web APIs usadas (para funciones asíncronas) que junto a los demás componentes forman el modelo de JS.



Memory Heap : *Región de **memoria** libre, normalmente de gran tamaño, dedicada al alojamiento dinámico de objetos.*

Call Stack : ***Pila** de llamadas, encargada de realizar un seguimiento de llamadas a funciones.*

Queue : *Es la **cola** que registra mensajes (eventos, temporizadores, etc..) asociados a sus callbacks (funciones de respuestas) para ser atendidos cuando marca el bucle de eventos.*

Event Loop : *Se encarga de controlar cuándo se procesan (de forma síncrona) los mensajes de la cola, siempre que la pila de llamadas esté vacía.*

Web APIs : *Son los interfaces del navegador que usa el modelo de JS, normalmente para* realizar operaciones asíncronas.

El funcionamiento del modelo sobre el ejemplo anterior **Ej1** :

a. Se carga en la pila (call stack), ***console.log("dato1")*** y se ejecuta.
b. Se carga en la pila **setTimeout (...)**, pero pasa automáticamente a llamar a la Web API (temporizador), esperando que transcurra el tiempo (en este caso 1 seg.), no se ejecuta **fun1( )** de momento.
c. Sin esperar al temporizador, se carga en la pila ***console.log("dato3")*** *y se ejecuta.*
d. *Seguidamente si ha transcurrido 1 segundo en el temporizador se produce un mensaje y la función* ***fun1( )*** *pasa a la cola, el bucle de eventos (Event Loop) comprueba que la pila está vacía, una vez en la cola y estando la pila vacía se ejecuta esta función lanzando* ***console.log("dato2")*** *y ejecutándose, finalizando el programa.*

Para ver una simulación del modelo para este programa podemos usar el siguiente enlace :

Simulación modelo JS

Callbacks

Ya sabemos que las operaciones síncronas son aquellas que se ejecutan una tras otra esperando que la anterior acabe por completo. Por el contrario, las operaciones asíncronas son aquellas que responden en tiempo indeterminado prosiguiendo así la ejecución del resto de operaciones sin esperar que se devuelva una respuesta hasta más tarde. Normalmente estas últimas van mezcladas con las primeras.

Pero, aunque sea en un “futuro”, queremos ejecutar alguna acción cuando tengamos respuesta asíncrona, para ello podemos usar los callbacks. Los callbacks son funciones pasadas como argumento a otra función, las cuales se ejecutarán más tarde cuando esta última haya realizado las operaciones necesarias. Se usan para controlar la programación asíncrona en JS.

Ejemplos a continuación →

En los siguientes ejemplos podemos ver cómo funcionan los callbacks al pasar como argumento una función.

```
<script>
  console.log("dato1");
  setTimeout(function fun1 (){console.log("dato2")},1000);
  console.log("dato3");
</script>
```

```
    <script>
    function saludar(nombre) {
        alert('Hola ' + nombre);
    }

    function procesarEntradaUsuario(callback) {
        var nombre = prompt('Por favor ingresa tu nombre.');
        callback(nombre);
    }
    procesarEntradaUsuario(saludar);
    </script>
```

Aunque con poco número de llamadas (callbacks) normalmente son suficiente para controlar la ejecución de operaciones asincrónicas, podría ser necesario tener que realizar varias llamadas anidadas de éstas, dando lugar a un código poco legible o entendible, por ejemplo :

Ej2:

```
<script>
setTimeout(function(){
  console.log("Etapa 1 completada");
  setTimeout(function(){
    console.log("Etapa 2 completada");
    setTimeout(function(){
      console.log("Etapa 3 completada");
      setTimeout(function(){
        console.log("Etapa 4 completada");
        // continuar indefinidamente ...
      }, 3000);
    }, 3000);
  }, 3000);
}, 3000);
</script>
```

A esta estructura de llamadas anidadas se le conoce como callbacks hell

Poca legibilidad que daremos solución a través de los objetos Promises y Async / Await

Promises (promesas)

Una promesa es un objeto que representa el resultado de una operación asíncrona. Están basados en callbacks pero facilitan el uso de las operaciones asíncronas devolviendo (normalmente) ahora o en un futuro dos estados (callbacks), uno se ha resuelto (resolved) con éxito y se ejecuta la función asociada, y otro se rechaza (rejected) con un error ejecutándose el código asociado. Ej:

```
<script>
var promise = new Promise(function(resolve, reject) {
  function hola() {
    resolve('Hola mundo')
  }
  setTimeout(hola, 2000)
  // reject('error');
})
promise
  .then(function(mensaje) {console.log(mensaje)})
  .catch(function(err) {console.error('ERROR: ' + err)})
</script>
```

Devuelve Hola Mundo si se ejecuta la promesa correctamente

En este ejemplo se crea una promesa en su forma más básica.

Se usa la función setTimeout para simular una petición asíncrona de la cual no sabemos qué tiempo de respuesta tendrá.

Como veremos en algunos ejemplos que vienen a continuación, y con la práctica comprobaremos que las promesas tienen las siguientes ventajas:

- Nos permite mantener un mejor control del programa, sin depender de la respuesta de una llamada asíncrona que podría acabar o no correctamente.
- Podemos controlar la respuesta si se lanza una excepción que se pasa al método catch( ).
- Podemos encadenar promesas que realizan operaciones asíncronas esperándose una tras otra.
- Lanzar simultáneamente operaciones asíncronas en “paralelo”, esperar que se resuelvan todas ó realizar alguna acción cuando alguna de ellas se resuelva .
- Todo ello hace que el código sea más legible y más fácil de mantener.

En el siguiente ejemplo se resuelve con promesas el código expuesto anteriormente que mostraba la estructura callbacks hell, el cual imprimía un mensaje cada cierto tiempo simulando llamadas asíncronas encadenadas.

callbacksHellConPromesas

En este otro, se usa el método Promise.all( ) para esperar a que se resuelvan **todas** las promesas. Acepta como parámetro un array de promesas, y devuelve otro array con los resultados de todas ellas.
promesasConAll

En este otro ejemplo, se usa el método Promise.race ( ) para devolver aquella promesa (operación asíncrona) que tarde menos en resolverse, se vuelve a usar el temporizador para simular distintos tiempos de resolución.
promesasConRace

Para este nuevo ejemplo vamos a usar el método fetch ( ) que está basado en promesas (devuelve una promesa) y que nos va a facilitar una petición asíncrona (es una mejora equivalente a usar el objeto de AJAX, XMLHttpRequest) . Accedemos a un archivo JSON e imprimimos el resultado solicitado.

```
<script>

fetch("carteras.json")
.then(result => result.json())
.then(objJson => console.log(objJson[0].nombre))
.catch(e => console.log(`Error capturado:  ${e}`));

</script>
```

Indicar que se están **usando funciones flechas** para reducir código, y como se puede ver usamos dos promesas encadenadas, en la primera devuelta por fetch( ) , se usa el método .json para devolver directamente objeto JSON,

y la segunda se resuelve para obtener el nombre de uno de los objetos del array.

En este otro ejemplo usamos fetch( ) en lugar del objeto XMLHttpRequest (visto en el tema anterior de AJAX) para simplificar la solicitud a un archivo PHP por GET, al cual se le envía varios parámetros y devuelve un objeto JSON.

```
<body>
    <button type="button" onclick="funAjax ()">Envía parámetros por GET y recibe valores</button>
    <div id="datos"> </div>
    <script>
    function funAjax() {
        fetch("jsonGET.php?nombre=Juan&ciudad=Ubrique")
        .then(result => result.json())
        .then(function(obj){document.getElementById("datos").innerHTML =
                        `Desde servidor ${obj.nombre} de ${obj.ciudad}`} )
        .catch(e => console.log(`Error capturado:  ${e}`));
        }
    </script>
</body>
```

En este otro, usamos de nuevo fetch( ) con POST, enviando parámetros en formato JSON y recibiendo respuesta en el mismo formato. Formado por los dos archivos (HTML y PHP) para solicitud y respuesta respectivamente.

fetchConPHPPostJson.html jsonPOSTJson.php

Para más información de cómo usar fetch( ), el siguiente enlace :

Uso de fetch ( )

Async / Await

En el apartado anterior hemos visto cómo las promesas facilitan el manejo de operaciones asíncronas. Con los últimos estándares se ha llegado más allá para facilitar aún más el control de esas operaciones.
Basada en las promesas, con async se declara una función asíncrona que devolverá una promesa automáticamente, y usando dentro de ella await esperará a que se resuelva la promesa sin bloquear código, de forma asíncrona. Ej:

```
<script>
async function busca(url)  {
  const response = await fetch(url);
  const objson= await response.json();
  console.log(objson[0].nombre)
}
busca("carteras.json");
</script>
```

Realiza la misma consulta al archivo JSON, pero en lugar de usar promesas usamos, async/await

Ahora veremos el mismo ejemplo anterior pero controlando posibles errores.

```
<script>
async function busca(url)  {
    try {
        const response = await fetch(url);
        const objJson= await response.json();
        console.log(objJson[0].nombre)
    }catch(error){
        console.log(error.message);
    }
}
busca("carteras.json");
</script>
```

A continuación dos nuevos ejemplos, el primero realiza varias llamadas a un temporizador simulando peticiones asíncronas, cada una de ellas debe acabar para resolver la siguiente (ejecución síncrona), y el segundo, lanza las peticiones en paralelo para resolverlas de forma concurrente. →

```
<script>
const pausa = t => new Promise(resolve => setTimeout(resolve, t));
async function espera() { // Ejecución de las pausas de forma síncrona ...
  await pausa(1000);
  await pausa(1000);
  console.log( "Tarda aprox., 2 segundos.");
};
espera();
</script>
```

```
<script>
const pausa = t => new Promise(resolve => setTimeout(resolve, t));
async function espera() { // Ejecución de las pausas de forma asíncrona ..
  const t1=pausa(1000);
  const t2=pausa(1000);
  await t1;
  await t2;
  console.log( "Tarda aprox., 1 segundo.");
}
espera();
</script>
```

Ejemplo import/export módulos

En este último ejemplo podemos ver cómo cargar módulos (ya sean variables, funciones, clases, etc..) de un archivo **.js** a otro **.js**, u otro HTML , primero estáticamente (obliga a que el nombre del módulo sea string literal) , y luego de forma dinámica usando **await** .

import.html exportSuma.js → de forma estática.

importDinami.html → de forma dinámica.

En este tema hemos visto cómo funciona el motor de JS, y cómo se hace más fluida la ejecución de código usando funciones asíncronas de la API Web del navegador, para ampliar información sobre desarrollo de código para ejecución paralela (varios hilos en segundo plano), podemos usar los llamados WebWorkers

Bibliografía

- LibrosWeb
- W3Schools
- Developer Mozilla Docs
- Lemoncode.net