PROGRAMACIÓN ASÍNCRONA

Contenido

1. PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA

Las funciones de tipo callback son uno de los elementos del lenguaje JavaScript que ayudan a entender por qué es un lenguaje asíncrono. Pero es momento ya de dedicar un tiempo a las bases de la programación asíncrona.

La mayoría de los lenguajes de programación son síncronos. Esto significa que, por ejemplo, la línea número 9 del código no se ejecuta si la tarea que realiza la línea 8 no ha terminado. La dificultad que tienen muchos desarrolladores, incluso experimentados, para entender y aplicar bien el lenguaje JavaScript se debe a esta circunstancia: a que antes han sido programadores de lenguajes síncronos. Que JavaScript sea asíncrono implica que una instrucción puede no haber terminado su labor cuando ya se está ejecutando la siguiente.

Pero la cuestión es por qué es interesante, incluso fundamental, que JavaScript trabaje de forma asíncrona. Un ejemplo típico es una página que necesita cargar dos elementos indepen- dientes que muestran información procedente de dos servicios en Internet. Por ejemplo, podría- mos crear una página que, en una capa, muestre la temperatura prevista para hoy, proporcionada por un servicio externo que proporciona las temperaturas, y en otra, un mapa que muestre cómo llegar a nuestra oficina, usando un servicio de mapas en Internet.

La temperatura y el mapa tardan en cargar. Si la programación fuera síncrona, el mapa no se carga si antes no ha llegado la información de la temperatura. No nos interesa este efecto, es más eficiente que ambos componentes se carguen de forma independiente. Si el segundo espera al primero, la carga es más lenta y el usuario de la aplicación lo notaría. Lo ideal es que la carga sea asíncrona.

En lenguaje síncronos la única manera de conseguir este tipo de efectos es crear varios hilos independientes en los que cada uno realiza una tarea. Así funciona, por ejemplo, el lenguaje Java para conseguir dos procesos que consigan resultados de forma independiente. JavaScript es un lenguaje de un único hilo, no se pueden programar dos hilos. Pero las operaciones sobre la red y otras operaciones de entrada/salida (como consultar bases de datos) se lanzan de forma independiente.

Pero también este efecto tiene sus problemas. ¿Y si deseamos colorear el mapa? El problema es que, aunque tengamos el código para colorear el mapa, debemos asegurarnos de que el mapa ha llegado antes de empezar a colorear. Esto requiere sincronización, algo del tipo: cuando el mapa llegue, coloreamos.

Solución a esto ya tenemos, puesto que conocemos las funciones callback. Mediante funciones callback podemos hacer que el código de colorear se lance justo cuando se ha cargado el mapa. Algo, como esto:

```
componente.haCargado(()=>{colorear()});
```

Se invoca a colorear cuando el componente se ha cargado.

2. CALLBACK HELL

Este término que podemos traducir como el infierno de los callback, se popularizó hace unos años cuando se percibió el problema de tener que sincronizar numerosas acciones. Esto provoca un exceso de funciones callback que van desplazando el código a su derecha haciéndole poco legible y mantenible.

Siguiendo con el ejemplo anterior del componente de mapas, supongamos que ahora deseamos que el mapa se cargue cuando hagamos clic a un botón concreto. Además, tras colorear el mapa, deseamos realizar una animación sobre el mismo. Pero, todo ello, solo si hemos podido realizar correctamente y sin error las tareas anteriores. El código podría ser:

```
boton.addEventListener('click', function (ev) {
  cargarMapa(componente, function (error) {
    if (error) {
      console.log('Error al cargar el mapa');
    } else {
      colorear(componente, function (error) {
        if (error) {
          console.log('Error al colorear');
        }
      });
    }
  });
});
```

Con tantas llamadas de tipo callback, el código se desplaza mucho a la derecha. En este código hemos hecho que las funciones callback reciban una variable que marca si hubo error en la operación. El hecho de valorar errores es fundamental para saber si el proceso se hizo bien o mal, es muy importante en labores de programación asíncrona. Pero esa valoración, que es habitual, desplaza aún más el código.

Ciertamente, no hay duda de que la invocación a ***animar(componente)*** realizará la animación tras colorear. Por otro lado, colorear se realiza tras la carga del mapa, y la carga no se inicia si no se ha hecho clic en el botón. El planteamiento es correcto, pero la legibilidad es terrible. Un infierno, como dice el título de este apartado.

3. PROMESAS

3.1. INTRODUCCIÓN A LAS PROMESAS

La solución al problema anterior es una estructura que permite controlar de forma más organizada las tareas asíncronas sin tener que usar tantas funciones callback anidadas. Esa nueva estructura es lo que se conoce como promesa.

Una promesa permite invocar a una función o tarea, cuya labor requiere una ejecución asíncrona. Podremos determinar lo que ocurre en el caso de que esa tarea concluya correctamente y lo que haremos en el caso de concluir mal. En las dos situaciones, podremos recoger información al respecto.

Las promesas pueden tener uno de estos estados:

- Cumplida (resolved o fulfdled). Si la tarea relacionada con la promesa se ha finalizado con éxito.
- Rechazada (rejected). Si la tarea no finaliza con éxito.
- Pendiente de finalizar (pending). En proceso de finalización.
- Finalizada (settled). Independiente de si se ha cumplido o no, la tarea relacionada con la promesa ha finalizado.

Las promesas son objetos de tipo Promise. Estos objetos son los que hacen la labor de relacionar la labor asíncrona con las acciones a tomar en caso de éxito o fracaso. Para ello, proporcionan varios métodos y, sobre todo, una función de construcción de objetos Promise que es donde se constituye realmente la promesa.

3.2. CREACIÓN DE PROMESAS

La creación de promesas sigue esta estructura formal:

```
let promesa = new Promise(function(resolver, rechazar) {

  código de la tarea asíncrona

  if (condición que valida que la tarea fue exitosa) {
    resolver(información sobre el éxito);
  }
  else {
    rechazar(información sobre el fracaso);
  }
});
```

Para explicar el código anterior veamos lo siguiente:

Las promesas son objetos que se construyen indicando una función de tipo callback.

Esa función callback acepta dos parámetros, los cuales son dos funciones. La primera (se la suele llamar resolve o resolver) se invoca cuando se ha verificado que la operación ha finalizado de forma correcta. La segunda (se suele llamar reject o rechazar) se invoca cuando se ha determinado que el proceso no ha finalizado correctamente.

Ambas funciones reciben parámetros. A cada una se la envía un objeto que contiene la información de la resolución en caso de éxito (función resolve) o de fracaso (reject). Para la función de rechazo el parámetro suele ser un objeto de error, ya que permite la gestión de errores de forma más conveniente.

La creación del objeto de promesa (la creación de la promesa) implica lanzar la tarea en segundo plano.

3.3. MÉTODO THEN

Pero la cuestión es cuándo recogemos los resultados del éxito o el fracaso. Ahí interviene el método then. Este método acepta una función callback, que **será invocada cuando la tarea de la promesa finalice con éxito**.

3.4. MÉTODO CATCH

Como ya hemos dicho, en el caso de la función rechazar, es habitual lanzar un error. Por eso, hay un método llamado catch que permite gestionar el rechazo en la promesa. Se asemeja a la estructura try..catch. Además, se permite encadenar ambos métodos, porque el resultado de los métodos then y catch es el propio objeto de la promesa (objeto Promise). La sintaxis completa es:

```
promesa.then(function(resultado){
    …
}).catch(function(resultado){
    …
});
```

3.5. EJEMPLOS DE PROMESAS

La idea es simple, pero es compleja de entender dada su novedosa sintaxis. Por ello, vamos a empezar haciendo una promesa simple. Haremos una promesa que consista en que, si dos variables valen lo mismo, por pantalla salga un mensaje diciéndolo, y si no así, lanzaremos un error. Evidentemente para una acción como esta, no hace falta usar promesas, pero usamos esta idea para entender el funcionamiento de las promesas.

```
var promesa = new Promise((resolver, rechazar) => {
  let n1 = 2;
  let n2 = 2;
  if (n1 == n2) resolver(' ¡Son iguales!');
  else rechazar(Error('Algo raro ha pasado'));
});

promesa
  .then((respuesta) => {
    console.log(respuesta);
  })
  .catch((error) => {
    console.log(error.message);
  });
```

Instantáneamente aparece el texto "¡Son iguales!" porque la condición es verdadera y eso provoca lanzar la función resolver enviando como parámetro el texto "¡Son iguales!". El método then lanza la función callback que usa como parámetro el texto anterior y ese texto

sale por pantalla, ya que la función callback del método then solo hace la labor de mostrar el parámetro que reciba por consola.

Si cambiamos los valores de n1 y n2 para que no sean iguales, se provocará una excepción que paraliza el programa y muestra el error. Para capturar el error y gestionarlo, es para lo que se usa catch.

3.6. ENCADENAMIENTO DE MÉTODOS

Se pueden encadenar los métodos then y catch las veces que haga falta. Eso es una buena manera de estructurar nuestras aplicaciones evitando un exceso de control de funciones callback.

Un ejemplo sencillo de esta idea sería este:

```
   cargarMapa()
  .then((mapa) => cargarPlantilla(mapa))
  .then((mapa) => colorear(mapa))
  .catch((error) => console.log("Error en la carga:", error));
```

No podemos colorear el mapa si no hemos podido cargarlo. Supongamos, además, que debemos cargar para colorear, una plantilla de colores de Internet. Sin esa plantilla no podemos colorear. Podríamos crear una promesa asociada al éxito de cargar o no el mapa. Si se carga de forma correcta el mapa, entonces lanzamos el segundo método de cuyo éxito depende el coloreado final.

3.7. MÉTODOS DEL OBJETO PROMISE

El objeto Promise aporta un método estático llamado **Promise.resolve**, que crea una nueva promesa resuelta y enviando (como si se hubiera invocado al método resolver en la creación de la promesa) el objeto que se envíe como parámetro.

Ejemplo de uso:

```
let promesa=Promise.resolve("Ha funcionado todo");
promesa.then((mensaje)=>{console.log(mensaje)});
```

Se muestra: Ha funcionado todo. Lo que hace este método es crear una promesa cumplida.

El método contrario es **Promise.reject**:

```
      let promesa = Promise.reject(Error('No ha funcionado nada'));
      promesa
        .then((mensaje) => {
          consolé.log(mensaje);
        })
        .catch((error) => {
          console.log(error.message);
        });
```

Muestra el mensaje "No ha funcionado nada". Lo que hace este método es crear una promesa rechazada.

Otro método interesante es **Promise.all**. Este método devuelve una promesa cumplida si todas las promesas, de la colección que recibe como parámetro, son cumplidas. Si alguna se rechaza, entonces, el método devuelve una promesa rechazada. Ejemplo:

```
let promesa1 = Promise.resolve('Estoy resuelta');
    let promesa2 = new Promise((resolver) => {
      setTimeout(() => {
        resolver('Resuelvo en 3s');
      }, 3000);
    });
    let promesa3 = new Promise((resolver) => {
      setTimeout(() => {
        resolver('Resuelvo en 6s');
      }, 6000);
    });
    let promesaConjunta = Promise.all([promesa1, promesa2, promesa3]);
    console.log('Empezando');
    promesaConjunta.then((resultados) => {
      let n = 1;
      for (let resultado of resultados) {
        console.log('Promesa num ${n}: Mensaje:${resultado}');
        n++;
      }
    });
```

Se crean tres promesas llamadas promesa1, promesa2 y promesa3. La primera genera una resolución positiva al instante, la siguiente tras tres segundos y la tercera tras seis segundos. Hay que recordar que setTimeout es asíncrona y realiza sus acciones en segundo plano.

La promesa conjunta, creada con el método Promise.all, exige del cumplimiento de las tres promesas para ser considerada una promesa resuelta. En este caso pasarán seis segundos antes de saber que las tres están resueltas. Lo cual significa que este código muestra el texto Empezando y seis segundos después, de golpe, muestra este texto:

```
Empezando
Promesa n2 1: Mensaje:Estoy resuelta
Promesa n2 2: Mensaje:Resuelvo en 3s
Promesa n2 3: Mensaje:Resuelvo en 6s
```

Hasta que no se resuelve la tercera, no podremos saber si las anteriores se han resuelto. **Esto significa que este método es fantástico para sincronizar acciones asíncronas.**

Es importante resaltar que, si alguna promesa se rechazara o provocara un error, instantánea- mente se generaría el rechazo, sin esperar al resto de promesas de la lista.

4. FUNCIONES ASYNC

Hay otras posibilidades en JavaScript de sincronización más avanzadas. En la norma ES2017 apareció una esperada mejora, conocida como **await/async**.

Hay funciones especiales que podemos declarar anteponiendo la palabra async. Esto declara un tipo de función especial, que internamente es un objeto de tipo AsyncFunction que devuelve una promesa implícita. Pero lo que nos interesa realmente, es que en las funciones de tipo async podemos utilizar la palabra clave await para poder sincronizar varios elementos asíncronos.

Es decir, la función puede requerir terminar un proceso antes de iniciar un segundo proceso que dependa de él. Hasta ahora como mecanismos disponíamos de las funciones callback y de los métodos then y catch de las promesas. No es que ya no necesitemos estos elementos, es que ahora los podemos integrar con un operador que aporta más legibilidad al código. El operador await requiere de una promesa, si se cumple, entonces la siguiente línea se ejecutará, si no, se mantiene en espera. Veamos un ejemplo:

```
let promesa1 = Promise.resolve('Estoy resuelta');
    let promesa2 = new Promise((resolver) => {
      setTimeout(() => {
        resolver('Resuelvo en 3s');
      }, 3000);
    });
    let promesa3 = new Promise((resolver) => {
      setTimeout(() => {
        resolver('Resuelvo en 6s');
      }, 6000);
    });

    async function esperarTiempos() {
      let mensaje1 = await promesa1;
      console.log(mensaje1);
      let mensaje2 = await promesa2;
      console.log(mensaje2);
      let mensaje3 = await promesa3;
      console.log(mensaje3);
    }

    esperarTiempos();
```

Las tres promesas iniciales son las que vimos en el apartado anterior para explicar el método Promise.all. La primera se resuelve al instante, la segunda tarda tres segundos y la tercera tarda seis segundos. La función esperarTiempos se marca con la palabra clave async y eso permite que haya tres variables (mensaje1, mensaje2 y mensaje3) que graben

el resultado del resolver de las promesas pero haciendo que se espere ese resultado antes de pasar a la siguiente línea de código. Eso provoca que el primer mensaje sale al instante, el segundo tres segundos desde que se inició el programa y **el tercer mensaje saldrá a los seis segundos**. Los segundos no se acumulan, porque las promesas se han creado previamente.

Diferente habría sido este otro código:

```
async function esperarTiempos() {
       let mensajel = await Promise.resolve('Estoy resuelta');
       console.log(mensajel);
       let mensaje2 = await new Promise((resolver) => {
         setTimeout(() => {
           resolver('Resuelvo en 3s');
         }, 3000);
       });
       console.log(mensaje2);
       let mensaje3 = await new Promise((resolver) => {
         setTimeout(() => {
           resolver('Resuelvo en 6s');
         }, 6000);
       });
       console.log(mensaje3);
     }
     esperarTiempos();
```

El código es similar, pero ahora cada promesa se genera tras esperar la finalización de la anterior. A la vista ocurre que el primer mensaje sale inmediatamente, el segundo tarda tres segundos desde que se mostró el primer mensaje y para ver el tercer mensaje hay que esperar seis segundos más, es decir**, el último mensaje aparece a los nueve segundos**.

Otra cuestión es qué pasa si alguna de las promesas es rechazada. ¿Cómo capturamos el rechazo? Lo lógico para ello es que el rechazo signifique lanzar un error, por lo que basta con que dispongamos una estructura de tipo try..catch.

```
  async function falla() {
       let resultado = await Promise.reject(Error('Promesa
rechazada')).catch(
         (error) => {
           console.log(error.message);
         }
       );
     }
     falla(); //Escribe: Promesa rechazada
```