Programación Asíncrona

Giancarlo Corzo Follow
May 26, 2017 · 3 min read

La programación asíncrona nos da la capacidad de "diferir" la ejecución de una función a la espera de que se complete una operación, normalmente de I/O (red, disco duro, …), y así evitar bloquear la ejecución hasta que se haya completado la tarea en cuestión. Esto es posible gracias a que las funciones son ciudadanos de primer nivel (first-class citizens) y pueden ser pasadas como argumentos de otras funciones tal cual lo haríamos con las variables.

Síncrono vs Asíncrono

Antes de poder hablar de programación asícrona debemos entender la diferencia entre ejecución síncrona y asícrona. Un código síncrono es aquel código donde cada instrucción espera a la anterior para ejecutarse mientras que un código asíncrono no espera a las instrucciones diferidas y continúa con su ejecución. Por lo general la asincronía permite tener una mejor respuesta en las aplicaciones y reduce el tiempo de espera del cliente.

Veamos un ejemplo:

Síncrono

Cada instrucción se ejecutará en secuencia hasta terminar.

```
1  console.log('Primero');
2  console.log('Segundo');
3  console.log('Tercero');
```

sincrno.js hosted with ♡ by GitHub view raw

Código síncrono

Asíncrono

En el caso asícrono, algunas de las instrucciones se ejecutarán a destiempo.

```
1  console.log('Primero');
2  setTimeout(_ => {
3    console.log('Segundo');
4  },10);
5  console.log('Tercero');
```

asincrono.js hosted with ♡ by GitHub view raw

Si ejecutamos este ejemplo veremos imprimirse ‘Primero’, ‘Tercero’, ‘Segundo’. Esto porque estamos usando la instrucción *setTimeout()* que difiere la ejecución x milisegundos.

Callbacks

Una función callback es una función de primer nivel que se pasa a otra función como variable y ésta es ejecutada en algún punto de la ejecución de la función que la recibe.

```
const callback = () => {
  console.log('Llamando a mi callback');
}

function otraFuncion(callback) {
  console.log('Ejecutando otra funcion');
  callback();
}

//Funcion anonima que funciona como callback
function otraFuncionAnonima(() => {
  console.log('Llamando a un callback anonimo');
});
```

callbacks.js hosted with ♡ by GitHub view raw

Callbacks

Veamos un ejemplo algo más complicado

```
const stations = [{id: 1, name: "Pardo"}, {id: 2, name: "Benavides"}];
// Map, reduce y Filter son clasicos ejemplos de callback,
// donde se delega la lógica detrás del map a una función externa haciéndo
// reutilizable el código
const stationNames = stations.map((station) => {
  return station.name;
});
```

map.js hosted with ♡ by GitHub view raw

Usando funciones anÓnimas en Mapas

Pues hasta ahí los callbacks parecen cool e inofensivos pero cuando abusamos de ellos se genera algo llamado el **Callback hell**

Callback Hell

Usar correctamente los callbacks a veces puede ser poco intuitivo y puede derivar en situaciones como las siguientes:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
```

```
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err)
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

callbackhell.is hosted with ♡ by GitHub view raw

Tomado de CallbackHell.com

Esto pasa cuando no se tiene buenas prácticas, pero ejemplos como estos se pueden mejorar rápidamente usando técnicas de modularización y manejo de errores. Pueden ver más sobre esto en callbackhell.com

Promesas (Promises)

¿Promise?

El objeto promesa es un proxy para un valor que no necesariamente se conoce al momento de creada la promesa. Permite asociar callbacks que se ejecutarán dependiendo del éxito o fracaso de la acción prometida. Las promesas pueden tener 3 estados definidos:

- *pendiente (pending)*: estado inicial, no cumplida o rechazada.
- *cumplida (fulfilled)*: significa que la operación se completó satisfactoriamente.
- *rechazada (rejected)*: significa que la operación falló.

Cuando creamos una promesa ésta recibe como parámetro una función con dos parámetros: el primero es una referencia a la función de éxito y el otro a una función de error.

```
var myPromise = new Promise((resolve, reject) => {

  //Instrucciones que se van a ejecutar

 if(/* Termino correctamente */) {
   resolve('Success!');
 } else {
    reject('Failure!');
 }
});

myPromise.then(function() {
  /* hacer algo mas cuando la promesa sea resuelta */
}).catch(function() {
  /* capturar el error */
```

```
16  })
```

promise.js hosted with ♡ by GitHub view raw

Ahora que conocemos la estructura básica de una promesa hagamos algo más complejo y real.

```
1   const get = url => {
2     return new Promise((resolve, reject) => {
3       var xhr = new XMLHttpRequest();
4       xhr.addEventListener('load', _ => {
5         if (xhr.status !== 200) {
6           reject(new Error(xhr.statusText));
7         }
8         resolve(xhr.response);
9       });
10
11      xhr.open('GET', url);
12      xhr.send();
13    }
14  }
15
16  get('story.json').then(response => {
17    console.log("Success!", response);
18  }).catch(error => {
19    console.error("Failed!", error);
20  });
```

getURL.js hosted with ♡ by GitHub view raw

get data usando promises

Conclusión

Los promises son una mejor forma de encapsular la lógica asíncrona logrando evitar tener que anidar callbacks y haciendo mucho más legible nuestro código y menos propenso a errores.

Referencias:

Promise

El objeto Promise (Promesa) es usado para computaciones asíncronas. Una promesa representa un valor que puede estar...

developer.mozilla.org

Callback Hell

Callback Hell *A guide to writing asynchronous JavaScript programs* ### What is "*callback hell*"? Asynchronous...

callbackhell.com

Thanks to Lupo Montero.

JavaScript Asynchronous Promises Callbacks

About Help Legal