Usar promesas

Una `Promise` (promesa en castellano) es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona. Una promesa puede ser creada usando su constructor. Sin embargo, la mayoría de la gente son consumidores de promesas ya creadas devueltas desde funciones. Esta guía explorará por lo tanto el consumo (uso) de promesas devueltas primero.

Esencialmente, una promesa es un objeto devuelto al cual enganchas las funciones callback, en vez de pasar funciones callback a una función.

Por ejemplo, en vez de una función del viejo estilo que espera dos funciones callback, y llama a una de ellas en caso de terminación o fallo:

```
function exitoCallback(resultado) {
  console.log("Tuvo éxito con " + resultado);
}

function falloCallback(error) {
  console.log("Falló con " + error);
}

hazAlgo(exitoCallback, falloCallback);
```

... las funciones modernas devuelven una promesa a la que puedes enganchar tus funciones de retorno:

```
let promesa = hazAlgo();
promesa.then(exitoCallback, falloCallback);
```

…o simplemente:

```
hazAlgo().then(exitoCallback, falloCallback);
```

Nosotros llamamos a esto una *llamada a función asíncrona*. Esta convención tiene varias ventajas. Exploraremos cada una de ellas.

Garantías

A diferencia de las funciones callback pasadas al viejo estilo, una promesa viene con algunas garantías:

- Las funciones callback nunca serán llamadas antes de la terminación de la ejecución actual del bucle de eventos de JavaScript.
- Las funciones callback añadidas con .then serán llamadas *después* del éxito o fracaso de la operación asíncrona, como arriba.
- Pueden ser añadidas múltiples funciones callback llamando a .then varias veces, para ser ejecutadas independientemente en el orden de inserción.

Pero el beneficio más inmediato de las promesas es el encadenamiento.

Encadenamiento

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo. Logramos esto creando una cadena de promesas.

Aquí está la magia: la función `then` devuelve una promesa nueva, diferente de la original:

```
const promesa = hazAlgo();
```

```
const promesa = hazAlgo();
const promesa2 = promesa.then(exitoCallback, falloCallback);
```

o

```
let promesa2 = hazAlgo().then(exitoCallback, falloCallback);
```

Esta segunda promesa representa la terminación no sólo de `hazAlgo()`, sino también de `exitoCallback` o `falloCallback` que pasaste, las cuales pueden ser otras funciones asíncronas devolviendo una promesa. Cuando ese es el caso, cualquier función callback añadida a `promesa2` se queda encolada detrás de la promesa devuelta por `exitoCallback` o `falloCallback`.

Básicamente, cada promesa representa la terminación de otro paso asíncrono en la cadena.

En los viejos días, hacer varias operaciones asíncronas en fila conduciría a la clásica pirámide maldita de funciones callback:

```
hazAlgo(function(resultado) {
  hazAlgoMas(resultado, function(nuevoResultado) {
    hazLaTerceraCosa(nuevoResultado, function(resultadoFinal) {
      console.log('Obtenido el resultado final: ' + resultadoFinal
    }, falloCallback);
  }, falloCallback);
}, falloCallback);
```

Con las funciones modernas, en cambio, enganchamos nuestras funciones callback a las promesas devueltas, formando una *cadena de promesas*:

```
hazAlgo().then(function(resultado) {
  return hazAlgoMas(resultado);
})
.then(function(nuevoResultado) {
  return hazLaTerceraCosa(nuevoResultado);
})
.then(function(resultadoFinal) {
  console.log('Obtenido el resultado final: ' + resultadoFinal);
})
.catch(falloCallback);
```

Los argumentos a `then` son opcionales, y `catch(falloCallBack)` es un atajo para `then(null, falloCallBack)`. Es posible que vea esto expresado con funciones de flecha en su lugar:

```
hazAlgo()
.then(resultado => hazAlgoMas(resultado))
.then(nuevoResultado => hazLaTerceraCosa(nuevoResultado))
.then(resultadoFinal => {
  console.log(`Obtenido el resultado final: ${resultadoFinal}`);
})
.catch(falloCallback);
```

Importante: Siempre devuelva promesas, de otra forma las funciones callback no se encadenarán, y los errores no sería capturados.

Encadenar después de una captura

Es posible encadenar después de un fallo, por ejemplo, un catch, lo que es útil para lograr nuevas acciones incluso después de una acción falló en la cadena. Lea el siguiente ejemplo:

```
new Promise((resolver, rechazar) => {
    console.log('Inicial');

    resolver();
})
.then(() => {
    throw new Error('Algo falló');

    console.log('Haz esto');
})
.catch(() => {
    console.log('Haz eso');
})
.then(() => {
    console.log('Haz esto sin que importe lo que sucedió antes');
});
```

Esto devolverá el siguiente texto:

```
Inicial
Haz eso
Haz esto sin que importe lo que sucedió antes
```

Note que el texto "Haz esto" no es escrito porque el error "Algo falló" causó un rechazo.

Propagación de errores

Usted podría recordar ver `falloCallback` tres veces en la pirámide maldita de antes, en comparación con sólo una vez al final de la cadena de promesas:

```
hazAlgo()
.then(resultado => hazAlgoMas(valor))
.then(nuevoResultado => hazLaTerceraCosa(nuevoResultado))
.then(resultadoFinal => console.log(`Obtenido el resultado final: ${resulta
.catch(falloCallback);
```

Básicamente, una cadena de promesas se para si hay una excepción, recorriendo la cadena por manejadores de captura en su lugar. Esto está modelado a la forma como trabaja el código síncrono:

```
try {
  let resultado = syncHazAlgo();
  let nuevoResultado = syncHazAlgoMas(resultado);

  let resultadoFinal = syncHazLaTerceraCosa(nuevoResultado);
  console.log(`Obtenido el resultado final: ${resultadoFinal}`);
} catch(error) {
  falloCallback(error);
}
```

Esta simetría con el código síncrono culmina con la mejora sintáctica async/await en ECMASCript 2017:

```
async function foo() {
```

```
  try {
    let resultado = await hazAlgo();
    let nuevoResultado = await hazAlgoMas(resultado);
    let resultadoFinal = await hazLaTerceraCosa(nuevoResultado);
    console.log(`Obtenido el resultado final: ${resultadoFinal}`);
  } catch(error) {
    falloCallback(error);
  }
}
```

Se construye sobre promesas, por ejemplo, `hazAlgo()` es la misma función que antes. Puedes leer más sobre la sintaxis aquí.

Las promesas resuelven un fallo fundamental de la pirámide maldita de funciones callback, capturando todos los errores, incluso excepciones lanzadas y errores de programación. Esto es esencial para la composición funcional de operaciones asíncronas.

Crear una promesa alrededor de una vieja API de callbacks

Una `Promise` puede ser creada desde cero usando su constructor. Esto debería ser sólo necesario para envolver viejas APIs.

En un mundo ideal, todas las funciones asíncronas ya devolverían promesas. Desafortunadamente, algunas APIs aún esperan que se les pase callbacks con resultado fallido/exitoso a la forma antigua. El ejemplo por excelencia es la función `setTimeout()`:

```
setTimeout(() => diAlgo("pasaron 10 segundos"), 10000);
```

Mezclar callbacks del viejo estilo y promesas es problemático. Si `diAlgo` falla o contiene un error de programación, nadie lo capturará.

Afortunadamente podemos envolverlas en una promesa. La mejor práctica es envolver las funciones problemáticas en el nivel más bajo posible, y después nunca llamarlas directamente de nuevo:

```
const espera = ms => new Promise(resuelve => setTimeout(resuelve, ms));
```

```
2
3 wait(10000).then(() => diAlgo("10 segundos")).catch(falloCallback);
```

Básicamente, el constructor de la promesa toma una función ejecutora que nos permite resolver o rechazar una promesa manualmente. Dado que `setTimeout` no falla realmente, descartamos el rechazo en este caso.

Composición

`Promise.resolve()` y `Promise.reject()` son atajos para crear manualmente una promesa resuelta o rechazada respectivamente. Esto puede ser útil a veces.

`Promise.all()` and `Promise.race()` son dos herramientas de composición para ejecutar operaciones asíncronas en paralelo.

La composición secuencial es posible usando algo de JavaScript inteligente:

```
1 [func1, func2].reduce((p, f) => p.then(f), Promise.resolve());
```

Básicamente, reducimos una matriz de funciones asíncronas a una cadena de promesas equivalente a: `Promise.resolve().then(func1).then(func2);`

Esto puede también ser hecho con una función de composición reutilizable, lo que es muy común en la programación funcional:

```
1 let applyAsync = (acc,val) => acc.then(val);
2 let composeAsync = (...funcs) => x => funcs.reduce(applyAsync, Promise.reso
```

La función composeAsync aceptará cualquier número de funciones como argumentos, y devolverá una nueva función que acepta un valor inicial que es pasado a través del conducto de composición. Esto es beneficioso porque cualquiera o todas las funciones pueden ser o asíncronas o síncronas y se garantiza que serán ejecutada en el orden correcto:

```
1 let transformData = composeAsync(func1, asyncFunc1, asyncFunc2, func2);
2 transformData(data);
```

En ECMAScript 2017, la composición secuencial puede ser realizada más simplemente con async/await:

```
for (let f of [func1, func2]) {
  await f();
}
```

Sincronización

Para evitar sorpresas, las funciones pasadas a `then` nunca serán llamadas sincrónicamente, incluso con una promesa ya resuelta:

```
Promise.resolve().then(() => console.log(2));
console.log(1); // 1, 2
```

En vez de ejecutarse inmediatamente, la función pasada es colocada en una cola de microtareas, lo que significa que se ejecuta más tarde cuando la cola es vaciada al final del actual ciclo de eventos de JavaScript, por ejemplo muy pronto:

```
const espera = ms => new Promise(resuelve => setTimeout(resuelve, ms));

espera().then(() => console.log(4));
Promise.resuelve().then(() => console.log(2)).then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

Vea también

- `Promise.then()`
- Promises/A+ specification

- Nolan Lawson: Tenemos un problema con las promesas — Errores comunes con las promesas