Async/await

Existe una sintaxis especial para trabajar con promesas de una forma más confortable, llamada “async/await”. Es sorprendentemente fácil de entender y usar.

Funciones async

Comencemos con la palabra clave `async`. Puede ser ubicada delante de una función como aquí:

```
async function f() {
return 1;
}
```

La palabra “async” ante una función significa solamente una cosa: que la función siempre devolverá una promesa. Otros valores serán envueltos y resueltos en una promesa automáticamente.
Por ejemplo, esta función devuelve una promesa resuelta con el resultado de `1`; Probémosla:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

...Podríamos explícitamente devolver una promesa, lo cual sería lo mismo:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Entonces, `async` se asegura de que la función devuelva una promesa, o envuelve las no promesas y las transforma en una. Bastante simple, ¿correcto? Pero no solo eso. Hay otra palabra, `await`, que solo trabaja dentro de funciones `async` y es muy interesante.

Await

```
// funciona solamente dentro de funciones async
let value = await promise;
```

`await` hace que JavaScript espere hasta que la promesa responda y devuelve su resultado.

Aquí hay un ejemplo con una promesa que resuelve en 1 segundo:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
setTimeout(() => resolve("¡Hecho!"), 1000)
  });

let result = await promise; // espera hasta que la promesa se resuelva (*)

alert(result); // "¡Hecho!"
}

f();
```

La ejecución de la función es pausada en la línea `(*)` y se reanuda cuando la promesa responde, con `result` volviéndose su resultado. Entonces el código arriba muestra "¡Hecho!" en un segundo.

Enfaticemos: `await` literalmente suspende la ejecución de la función hasta que se establezca la promesa, y luego la reanuda con el resultado de la promesa. Eso no cuesta ningún recurso de CPU, porque el motor de JavaScript puede hacer otros trabajos mientras tanto: ejecutar otros scripts, manejar eventos, etc.

Es simplemente una sintaxis más elegante para tener el resultado de una promesa que `promise.then`, es más fácil de leer y de escribir.

No se puede usar `await` en funciones comunes

Si tratamos de usar `await` en una función no async, tendremos un error de sintaxis:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

Es posible que obtengamos este error si olvidamos poner `async` antes de una función. Como se dijo, "await" solo funciona dentro de una función `async`.

Tomemos el ejemplo `showAvatar()` del capítulo Encadenamiento de promesas y rescribámoslo usando `async/await`:

1. Necesitaremos reemplazar los llamados `.then` con `await`.
2. También debemos hacer que la función sea `async` para que aquellos funcionen.

```
async function showAvatar() {

  // leer nuestro JSON
  let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

// leer usuario github
let githubResponse = await fetch(`https://api.github.com/users/${user.name}`)
let githubUser = await githubResponse.json();

  // muestra el avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // espera 3 segundos
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Bien limpio y fácil de leer, ¿no es cierto? Mucho mejor que antes.

Los navegadores modernos permiten `await` en el nivel superior de los módulos

En los navegadores modernos, `await` de nivel superior funciona, siempre que estamos dentro de un módulo. Cubriremos módulos en el artículo Módulos, introducción.

Por ejemplo:

```
// asumimos que este código se ejecuta en el nivel superior
dentro de un modulo
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

console.log(user);
```

Si no estamos usando módulos, o necesitamos soportar navegadores antiguos, tenemos una receta universal: envolverlos en una función async anónima.

Así:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
  ...
})();
```

await acepta "thenables"

Tal como `promise.then`, `await` nos permite el uso de objetos "thenable" (aquellos con el método `then`). La idea es que un objeto de terceras partes pueda no ser una promesa, sino compatible con una: si soporta `.then`, es suficiente para el uso con `await`.

Aquí hay una demostración de la clase `Thenable`; el `await` debajo acepta sus instancias:

```
class Thenable {
  constructor(num) {
this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // resuelve con this.num*2 después de 1000ms
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
}
}

async function f() {
  // espera durante 1 segundo, entonces el resultado se vuelve 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

Si `await` obtiene un objeto no-promesa con `.then`, llama tal método proveyéndole con las funciones incorporadas `resolve` y `reject` como argumentos (exactamente como lo hace con ejecutores `Promise` regulares). Entonce `await` espera hasta que une de ellos es llamado (en el ejemplo previo esto pasa en la línea `(*)`) y entonces procede con el resultado.

Métodos de clase Async

Para declarar un método de clase async, simplemente se le antepone `async`:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1 (lo mismo que (result => alert(result)))
```

El significado es el mismo: Asegura que el valor devuelto es una promesa y habilita `await`.

Manejo de Error

Si una promesa se resuelve normalmente, entonces `await promise` devuelve el resultado. Pero en caso de rechazo, dispara un error, tal como si hubiera una instrucción `throw` en aquella línea.

Este código:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

...es lo mismo que esto:

```
async function f() {
  throw new Error("Whoops!");
}
```

En situaciones reales, la promesa tomará algún tiempo antes del rechazo. En tal caso habrá un retardo antes de que `await` dispare un error.

Podemos atrapar tal error usando `try..catch`, de la misma manera que con un `throw` normal:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

En el caso de un error, el control salta al bloque `catch`. Podemos también envolver múltiples líneas:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
let user = await response.json();
} catch(err) {
// atrapa errores tanto en fetch como en response.json
    alert(err);
}
}

f();
```

```
f();
```

Si no tenemos `try..catch`, entonces la promesa generada por el llamado de la función async `f()` se vuelve rechazada. Podemos añadir `.catch` para manejarlo:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() se vuelve una promesa rechazada
f().catch(alert); // TypeError: failed to fetch // (*)
```

Si olvidamos añadir `.catch` allí, obtendremos un error de promesa no manejado (visible en consola). Podemos atrapar tales errores usando un manejador de evento global `unhandledrejection` como está descrito en el capítulo Manejo de errores con promesas.

`async/await` y `promise.then/catch`

Cuando usamos `async/await`, raramente necesitamos `.then`, porque `await` maneja la espera por nosotros. Y podemos usar un `try..catch` normal en lugar de `.catch`. Esto usualmente (no siempre) es más conveniente.

Pero en el nivel superior del código, cuando estamos fuera de cualquier función `async`, no estamos sintácticamente habilitados para usar `await`, entonces es práctica común agregar `.then/catch` para manejar el resultado final o errores que caigan a través, como en la línea `(*)` del ejemplo arriba.

`async/await` funciona bien con `Promise.all`

Cuando necesitamos esperar por múltiples promesas, podemos envolverlas en un `Promise.all` y luego `await`:

```
// espera por el array de resultados
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

En caso de error, se propaga como es usual, desde la promesa que falla a `Promise.all`, y entonces se vuelve una excepción que podemos atrapar usando `try..catch` alrededor del llamado.

Resumen

El comando `async` antes de una función tiene dos efectos:

1. Hace que siempre devuelva una promesa.
2. Permite que sea usado `await` dentro de ella.

El comando `await` antes de una promesa hace que JavaScript espere hasta que la promesa responda. Entonces:

1. Si es un error, la excepción es generada — lo mismo que si `throw error` fuera llamado en ese mismo lugar.
2. De otro modo, devuelve el resultado.

Juntos proveen un excelente marco para escribir código asincrónico que es fácil de leer y escribir.

Con `async/await` raramente necesitamos escribir `promise.then/catch`, pero aún no deberíamos olvidar que están basados en promesas porque a veces (ej. como en el nivel superior de código) tenemos que usar esos métodos. También `Promise.all` es adecuado cuando esperamos por varias tareas simultáneas.

Tareas

Rescribir usando async/await

Rescribir este código de ejemplo del capítulo Encadenamiento de promesas usando `async/await` en vez de `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('https://javascript.info/no-such-user.json')
.catch(alert); // Error: 404
```

solución

Reescribir "rethrow" con async/await

Debajo puedes encontrar el ejemplo “rethrow”. Rescríbelo usando `async/await` en vez de `.then/catch`.

Y deshazte de la recursión en favor de un bucle en `demoGithubUser`: con `async/await`, que se vuelve fácil de hacer.

```
class HttpError extends Error {
  constructor(response) {
```

```
    super(`${response.status} for ${response.url}`);
this.name = 'HttpError';
this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    });
}

// Pide nombres hasta que github devuelve un usuario válido
function demoGithubUser() {
  let name = prompt("Ingrese un nombre:", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Nombre completo: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No existe tal usuario, por favor reingrese.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

solución

Llamado async desde un non-async

Tenemos una función "regular" llamada `f`. ¿Cómo llamar la función `async`, `wait()` y usar su resultado dentro de `f`?

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

return 10;
```

```
}

function f() {
// ¿...qué escribir aquí?
// Necesitamos llamar async wait() y esperar a obtener 10
  // recuerda, no podemos usar "await"
}
```