6 de febrero de 2022

asíncrono/espera

Existe una sintaxis especial para trabajar con promesas de una manera más cómoda, llamada "async/await". Es sorprendentemente fácil de entender y utilizar.

Funciones asíncronas

Comencemos con la `async` palabra clave. Se puede colocar antes de una función, así:

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

La palabra "async" antes de una función significa una cosa simple: una función siempre devuelve una promesa. Otros valores se incluyen en una promesa resuelta automáticamente.

Por ejemplo, esta función devuelve una promesa resuelta con el resultado de `1` ; vamos a probarlo:

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

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

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

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

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

Por lo tanto, `async` garantiza que la función devuelva una promesa y envuelva las no promesas en ella. Bastante simple, ¿verdad? Pero no sólo eso. Hay otra palabra clave, `await` que funciona solo dentro de `async` funciones, y es genial.

Esperar

La sintaxis:

```
// works only inside async functions
let value = await promise;
```

La palabra clave `await` hace que JavaScript espere hasta que esa promesa se establezca y devuelva su resultado.

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

```
async function f() {

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

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

La ejecución de la función "se detiene" en la línea `(*)` y se reanuda cuando la promesa se cumple, convirtiéndose `result` en su resultado. Entonces el código anterior muestra "¡listo!" en un segundo.

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

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

No se puede usar `await` en funciones regulares.

Si intentamos utilizar `await` una función no asíncrona, habría un error de sintaxis:

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

Es posible que obtengamos este error si nos olvidamos de anteponer `async` una función. Como se indicó anteriormente, `await` solo funciona dentro de una `async` función.

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

1. Necesitaremos reemplazar `.then` las llamadas con `await` .
2. También deberíamos hacer la función `async` para que funcionen.

```
async function showAvatar() {

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

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

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

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Bastante limpio y fácil de leer, ¿verdad? Mucho mejor que antes.

`await` Los navegadores modernos permiten módulos de nivel superior

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

Por ejemplo:

```
1  // we assume this code runs at top level, inside a module
2  let response = await fetch('/article/promise-chaining/user.json');
3  let user = await response.json();
4
5  console.log(user);
```

Si no usamos módulos o debemos admitir navegadores más antiguos , existe una receta universal: incluir una función asíncrona anónima.

Como esto:

```
1  (async () => {
2    let response = await fetch('/article/promise-chaining/user.json');
3    let user = await response.json();
4    ...
5  })();
```

`await` acepta “thenables”

Like `promise.then`, `await` allows us to use thenable objects (those with a callable `then` method). The idea is that a third-party object may not be a promise, but promise-compatible: if it supports `.then`, that's enough to use it with `await`.

Here's a demo `Thenable` class; the `await` below accepts its instances:

```
1  class Thenable {
2    constructor(num) {
3      this.num = num;
4    }
5    then(resolve, reject) {
6      alert(resolve);
7      // resolve with this.num*2 after 1000ms
8      setTimeout(() => resolve(this.num * 2), 1000); // (*)
9    }
10 }
11
12 async function f() {
13   // waits for 1 second, then result becomes 2
14   let result = await new Thenable(1);
15   alert(result);
16 }
17
18 f();
```

If `await` gets a non-promise object with `.then`, it calls that method providing the built-in functions `resolve` and `reject` as arguments (just as it does for a regular `Promise` executor). Then `await` waits until one of them is called (in the example above it happens in the line `(*)`) and then proceeds with the result.

Async class methods

To declare an async class method, just prepend it with `async`:

```
1  class Waiter {
2    async wait() {
3      return await Promise.resolve(1);
4    }
5  }
6
7  new Waiter()
8    .wait()
9    .then(alert); // 1 (this is the same as (result => alert(result)))
```

The meaning is the same: it ensures that the returned value is a promise and enables `await`.

Error handling

If a promise resolves normally, then `await promise` returns the result. But in the case of a rejection, it throws the error, just as if there were a `throw` statement at that line.

This code:

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

...is the same as this:

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

In real situations, the promise may take some time before it rejects. In that case there will be a delay before `await` throws an error.

We can catch that error using `try..catch`, the same way as a regular `throw`:

```
async function f() {

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

f();
```

In the case of an error, the control jumps to the `catch` block. We can also wrap multiple lines:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}

```

```
12
    f();
```

If we don't have `try..catch`, then the promise generated by the call of the async function `f()` becomes rejected. We can append `.catch` to handle it:

```
1  async function f() {
2    let response = await fetch('http://no-such-url');
3  }
4
5  // f() becomes a rejected promise
6  f().catch(alert); // TypeError: failed to fetch // (*)
```

If we forget to add `.catch` there, then we get an unhandled promise error (viewable in the console). We can catch such errors using a global `unhandledrejection` event handler as described in the chapter Error handling with promises.

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

When we use `async/await`, we rarely need `.then`, because `await` handles the waiting for us. And we can use a regular `try..catch` instead of `.catch`. That's usually (but not always) more convenient.

But at the top level of the code, when we're outside any `async` function, we're syntactically unable to use `await`, so it's a normal practice to add `.then/catch` to handle the final result or falling-through error, like in the line `(*)` of the example above.

`async/await` works well with `Promise.all`

When we need to wait for multiple promises, we can wrap them in `Promise.all` and then `await`:

```
1  // wait for the array of results
2  let results = await Promise.all([
3    fetch(url1),
4    fetch(url2),
5    ...
6  ]);
```

In the case of an error, it propagates as usual, from the failed promise to `Promise.all`, and then becomes an exception that we can catch using `try..catch` around the call.

Summary

The `async` keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows `await` to be used in it.

La `await` palabra clave antes de una promesa hace que JavaScript espere hasta que esa promesa se cumpla y luego:

1. Si es un error, se genera una excepción, igual que si `throw error` nos llamaran en ese mismo lugar.
2. De lo contrario, devuelve el resultado.

Juntos proporcionan un excelente marco para escribir código asincrónico que sea fácil de leer y escribir.

`async/await` Rara vez necesitamos escribir , pero aún así no debemos olvidar que se basan en promesas, porque a veces (por ejemplo , `promise.then/catch` en el ámbito más externo) tenemos que usar estos métodos. También `Promise.all` es bueno cuando estamos esperando muchas tareas simultáneamente.

Tareas

Reescribir usando async/await

Vuelva a escribir este código de ejemplo del capítulo Encadenamiento de promesas usando `async/await` en lugar de `.then/catch` :

```
1  function loadJson(url) {
2    return fetch(url)
3      .then(response => {
4        if (response.status == 200) {
5          return response.json();
6        } else {
7          throw new Error(response.status);
8        }
9      });
10 }
11
12 loadJson('https://javascript.info/no-such-user.json')
13   .catch(alert); // Error: 404
```

solución

Reescribe "relanzar" con async/await

A continuación puede encontrar el ejemplo de "relanzar". Vuelva a escribirlo usando `async/await` en lugar de `.then/catch` .

Y deshacerse de la recursividad a favor de un bucle en `demoGithubUser` : con `async/await` eso se vuelve fácil de hacer.

```
1  class HttpError extends Error {
2    constructor(response) {
3      super(`${response.status} for ${response.url}`);
4      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);
      }
    });
}

// Ask for a user name until github returns a valid user
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

solución

Llamar asíncrono desde no asíncrono

Tenemos una función "normal" llamada `f`. ¿Cómo se puede llamar a la `async` función `wait()` y usar su resultado dentro de `f`?

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

  return 10;
}

function f() {
```

```
8    // ...what should you write here?
9    // we need to call async wait() and wait to get 10
10   // remember, we can't use "await"
11 }
```

PD: La tarea es técnicamente muy simple, pero la pregunta es bastante común para los desarrolladores nuevos en async/await.

solución

Previous lesson Next lesson

Share

Tutorial map

Comentarios

- Si tiene sugerencias sobre qué mejorar, envíe un problema de GitHub o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, explíquelo.
- Para insertar algunas palabras de código, use la `<code>` etiqueta, para varias líneas, envuélvalas en `<pre>` una etiqueta, para más de 10 líneas, use una caja de arena (plnkr , jsbin , codepen ...)

© 2007—2024 Ilya Kantorsobre el proyectoContáctenostérminos de uso política de privacidad