JavaScript
TUTORIAL

JavaScript async/await

Summary: in this tutorial, you will learn how to write asynchronous code using JavaScript `async` / `await` keywords.

Note that to understand how the `async` / `await` works, you need to know how promises work.

Introduction to JavaScript async / await keywords

In the past, to handle asynchronous operations, you used the callback functions. However, nesting many callback functions can make your code more difficult to maintain, resulting in a notorious issue known as callback hell.

Suppose that you need to perform three asynchronous operations in the following sequence:

1. Select a user from the database.
2. Get the user's services from an API.
3. Calculate the service cost based on the services from the server.

The following functions illustrate the three tasks. Note that we use the `setTimeout()` function to simulate the asynchronous operation.

```
function getUser(userId, callback) {
    console.log('Get user from the database.');
    setTimeout(() => {
        callback({
            userId: userId,
            username: 'john'
        });
    }, 1000);
}

function getServices(user, callback) {
    console.log(`Get services of  ${user.username} from the API.`);
    setTimeout(() => {
        callback(['Email', 'VPN', 'CDN']);
    }, 2 * 1000);
}

function getServiceCost(services, callback) {
    console.log(`Calculate service costs of ${services}.`);
    setTimeout(() => {
        callback(services.length * 100);
    }, 3 * 1000);
}
```

The following shows the nested callback functions:

```
getUser(100, (user) => {
    getServices(user, (services) => {
        getServiceCost(services, (cost) => {
            console.log(`The service cost is ${cost}`);
        });
    });
});
```

Output:

```
Get user from the database.
Get services of  john from the API.
Calculate service costs of Email,VPN,CDN.
The service cost is 300
```

To avoid this callback hell issue, ES6 introduced the promises that allow you to write asynchronous code in more manageable ways.

First, you need to return a `Promise` in each function:

```
function getUser(userId) {
    return new Promise((resolve, reject) => {
        console.log('Get user from the database.');
        setTimeout(() => {
            resolve({
                userId: userId,
                username: 'john'
            });
        }, 1000);
    })
}

function getServices(user) {
    return new Promise((resolve, reject) => {
        console.log(`Get services of  ${user.username} from the API.`);
        setTimeout(() => {
            resolve(['Email', 'VPN', 'CDN']);
        }, 2 * 1000);
    });
}

function getServiceCost(services) {
    return new Promise((resolve, reject) => {
        console.log(`Calculate service costs of ${services}.`);
        setTimeout(() => {
            resolve(services.length * 100);
        }, 3 * 1000);
    });
}
```

Then, you chain the promises:

```
getUser(100)
    .then(getServices)
    .then(getServiceCost)
    .then(console.log);
```

ES2017 introduced the `async` / `await` keywords that build on top of promises, allowing you to write asynchronous code that looks more like synchronous code and is more readable. Technically speaking, the `async` / `await` is syntactic sugar for promises.

If a function returns a Promise, you can place the `await` keyword in front of the function call, like this:

```
let result = await f();
```

The `await` will wait for the `Promise` returned from the `f()` to settle. The `await` keyword can be used only inside the `async` functions.

The following defines an `async` function that calls the three asynchronous operations in sequence:

```
async function showServiceCost() {
    let user = await getUser(100);
    let services = await getServices(user);
    let cost = await getServiceCost(services);
    console.log(`The service cost is ${cost}`);
}

showServiceCost();
```

As you can see, the asynchronous code now looks like the synchronous code.

Let's dive into the async / await keywords.

The async keyword

The `async` keyword allows you to define a function that handles asynchronous operations.

To define an `async` function, you place the `async` keyword in front of the function keyword as follows:

```
async function sayHi() {
    return 'Hi';
}
```

Asynchronous functions execute asynchronously via the event loop. It always returns a `Promise`.

In this example, because the `sayHi()` function returns a `Promise`, you can consume it, like this:

```
sayHi().then(console.log);
```

You can also explicitly return a `Promise` from the `sayHi()` function as shown in the following code:

```
async function sayHi() {
    return Promise.resolve('Hi');
}
```

The effect is the same.

Besides the regular functions, you can use the `async` keyword in the function expressions:

```
let sayHi = async function () {
    return 'Hi';
}
```

arrow functions:

```
let sayHi = async () => 'Hi';
```

and methods of classes:

```
class Greeter {
    async sayHi() {
        return 'Hi';
    }
}
```

The await keyword

You use the `await` keyword to wait for a `Promise` to settle either in a resolved or rejected state. You can use the `await` keyword only inside an `async` function:

```
async function display() {
    let result = await sayHi();
    console.log(result);
}
```

In this example, the `await` keyword instructs the JavaScript engine to wait for the `sayHi()` function to complete before displaying the message.

Note that if you use the `await` operator outside of an `async` function, you will get an error.

Error handling

If a promise resolves, the `await promise` returns the result. However, when the promise is rejected, the `await promise` will throw an error as if there were a `throw` statement.

The following code:

```
async function getUser(userId) {
     await Promise.reject(new Error('Invalid User Id'));
}
```

... is the same as this:

```
async function getUser(userId) {
    throw new Error('Invalid User Id');
}
```

In a real scenario, it will take a while for the promise to throw an error.

You can catch the error by using the `try...catch` statement, the same way as a regular `throw` statement:

```
async function getUser(userId) {
    try {
       const user = await Promise.reject(new Error('Invalid User Id'));
    } catch(error) {
        console.log(error);
    }
}
```

It's possible to catch errors caused by one or more `await promise` 's:

```
async function showServiceCost() {
    try {
        let user = await getUser(100);
        let services = await getServices(user);
        let cost = await getServiceCost(services);
        console.log(`The service cost is ${cost}`);
    } catch(error) {
        console.log(error);
    }
}
```

In this tutorial, you have learned how to use the JavaScript `async` / `await` keyword to write asynchronous code looks like synchronous code.