Javascript Asynchronous programming

written by Hong Le Nguyen 08.2024

terminology

asynchronous operations ---
are operations supported by *asynchronous functions*.
Some long-running operations (such as making Http requests, reading files) are asynchronous.

asynchronous functions ---
provided by the JavaScript environment, whose execution *does not block the main thread*.
JavaScript environment delegates asynchronous functions to background processes (in Web APIs) and upon completion, a callback function is triggered to process the result.

Callback function
is a regular function, passed as an argument to asynchronous function. *All callbacks are executed after the top-level synchronous code has finished*.

more "How Does Synchronous JavaScript environment Work ?" at the end of the document. =>See 5)

Note that *any function that calls a asynchronous function must itself be asynchronous*, using a callback to deliver its result.

synchronous code ---
JavaScript code that does not use any asynchronous functions will execute in a synchronous manner — one at a time, sequentially.

asynchronous code ---
Asynchronous functions enable asynchronous code.

asynchronous programming ---
writing code to handle and manage *asynchronous operations in sequence*, and improve overall performance.
JavaScript provides several mechanisms for writing asynchronous code, including *callbacks, promises, and async/await*.

An Example of asynchronous function ---

The asynchronous function setTimeout() does not block the main thread, so the (synchronous) function nextTask() is executed before the setTimeout() function completes and the callback is called.

```
function callback() {
   console.log("async operation done, callback run..")
}

function nextTask() {
   console.log("next task done")
}

// Executes the functions
setTimeout(callback, 1000) // Delays for 1s then executes callback
nextTask()
```

```
// output
next task done
async operation done, callback run..
```

1)
implement asynchronous operation using asynchronous function and its callback

Asynchronous function produces a result or error, which is then processed by the callback function.

```
function customAsyncFunction(callback, param) {
   // perform an asynchronous operation here...
   const result = ...

   // call callback to process result of asynchronous operation
   callback(result)
}
```

An Example ---

The function getData() calls an asynchronous function setTimeout(), must itself be asynchronous, does not block the execution of the function nextTask().
The callback logResult(result) processes the result of asynchronous operation.

```
function logResult(result) { // callback function
   console.log(`async operation done, callback run.. result=${result}`)
}

function getData(callback, URL) {
   // Simulating fetching user data from an API
   setTimeout(function() {
     const data = "abc"
     callback(data)  // callback processes result of asynchronous operation
   }, 1000)
}

function nextTask() {
   console.log("next task done")
}

getData(logResult, "myhost/data")
nextTask()

// output
next task done
async operation done, callback run.. result=abc
```

callback hell ---

Callback hell typically occurs when dealing with a series of asynchronous operations, such as making API requests, where one operation depends on the result of another or previous one.
The callback itself has to call functions that accept a callback. As a result, they are nested.

An example of what callback hell might look like:

```
asyncOperation1(function(result1) { // callback level 1
   asyncOperation2(result1, function(result2) { // callback level 2
            asyncOperation3(result2, function(result3) { // callback level 3
              // ...
            });
   });
});
```

With promises and async/await, you can avoid the clutches of callback and make code more readable, maintainable.

2)
promise

advantages ---

Promises provide a more structured and elegant way to handle asynchronous operations.
Promises simplify error handling*, eliminate callback nesting*, and improve code readability.
A promise represents the eventual completion or failure of an asynchronous operation, allowing you to chain operations using methods like then() and catch().

implement asynchronous operation using promise and callback

Create a Promise ---
A promise represents the eventual completion or failure of an asynchronous operation

```
function fetchData(URL) {
   let promise = new Promise(function(resolve, reject) {  // executor function

            // asynchronous operation
            // produces result or error

            // let success = true; // false
            // let result  =
            // let error   =

            if (success) {
               resolve(result);
            } else {
               reject(error);
            }

   });

   return promise;
}

fetchData(URL)
```

The constructor function takes a function as an argument called *the executor function*.
The executor function takes two arguments, resolve and reject. Your logic goes inside the executor function that runs automatically when a new Promise is created.
The returned promise object should be capable of informing when the execution has been started (Pending), completed (resolved) or retuned with error (rejected).

Consume a Promise ---

Once you have a reference, you can *attach callback function* by using the .then and .catch methods.
Once the promise is resolved, the .then callback method will be called with the resolved value.
And if the promise is rejected, the .catch method will be called with an error message.

```
fetchData(URL)
.then(function(result) { // callback function
   // This runs if the promise is fulfilled
   console.log(result);
})
.catch(function(error) {
   // This runs if the promise is rejected
   console.log(error);
});
```

implement *(asynchronous) operations in sequence* using Promises Chain

Chain Promises ---

Sometimes, you need to do several operations one after another, like load some data, process it, and then display it.

The linking of the promises is achieved by chaining then methods:

then() method uses a callback function as an argument and returns a new promise.
The new promise is then resolved with the value returned by the callback function.

```
fetchData(URL)
.then(function(response) {
   // Suppose this is another function that returns a promise
   return processData(response); // ==> processedData in next then-callback
})
.then(function(processedData) {
   console.log("Processed data:", processedData);
})
.catch(function(error) {
   console.log("Error:", error);
});
```

fetchData(URL) promise is resolved with value response,
1.then() return a new promise, the new promise is resolved with value processedData, returned from 1.then-callback, and so ...

An Example of Promises chaining ---

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
.then(response => response.json())
.then(data => console.log(data.title))
.catch((error) => {
  console.log("Error:", error);
});
```

3)
Async function

Async/await is provides a way to write asynchronous code in a synchronous style.

Async/Await, it's just syntactic sugar for Promises. In other words, *any existing Promises can be converted to use Async/Await syntax* instead.

What Async/Await allows us to do is *instead of having to access the results of our Promises inside a then method, we're able to assign the result of an asynchronous operation directly to a variable*. Even though this code looks synchronous, it behaves asynchronously.

await keyword is used *inside a* ***async*** *function* (function that contains asynchronous code), and therefore we have to put the async keyword before the function definition.

```
function doSomeAsyncStuff = async function () {
   try {
         let x = await someAsyncFunction () ;
   } catch (error) {
         ...
   }
}
```

an Example of Promises chaining using Async/Await ---

```
async function getData(URL) {
  try {
        const response = await fetch(URL);
        const data = await response.json();
        console.log(data.title);
  } catch (error) {
        console.log("Error:", error);
  }
}

getData('https://jsonplaceholder.typicode.com/posts/1');
```

4)
using axios library
Chaining Promises ---

```
axios.get('<https://api.example.com/data>')  // a promise
  .then((response) => {
    if (response.status !== 200) {
      throw new Error('Request failed with status: ' + response.status);
    }
    return response.data;
  })
  .then((data) => {
    // Process the data from the first request
    console.log(data);
    // Return a new Axios request
    return axios.get('<https://api.example.com/other-data>');
  })
  .then((otherDataResponse) => {
    // Process the data from the second request
    console.log(otherDataResponse.data);
  })
  .catch((error) => {
    // Handle errors that occurred in the chain
    console.error(error);
  });
```

Handling multiple Promises with Async/Await ---

```
async function fetchData() {
  try {
    // Initial Axios request
    const response1 = await axios.get('https://api.example.com/data');

    if (response1.status !== 200) {
      throw new Error('Request failed with status: ' + response1.status);
    }

    const data1 = response1.data;
    console.log(data1);

    // Second Axios request
    const response2 = await axios.get('https://api.example.com/other-data');
    const data2 = response2.data;
    console.log(data2);

    // Continue with more operations if needed...

  } catch (error) {
    // Handle errors that occurred in the try block
    console.error(error);
  }
}

// Call the async function
fetchData();
```

5)
How Does Synchronous JavaScript Work ?

How does JavaScript handle asynchronous functions ? ----------
https://www.codemancers.com/blog/2022-10-26-how-asynchronous-js-works/

When an *asynchronous function* is loaded to the ***call stack***, it is executed and starts a task *outside of the JavaScript engine* in the Web API. At this point, the asynchronous function has completed execution and is immediately popped from the call stack. Since it has been popped from the call stack, *it does not block code execution*. When the async function returns and is ready to be executed, it is placed in the ***callback queue***. The callback queue is like a data structure that holds all the callbacks that are going to be executed.

The ***Event Loop*** checks the call stack, and when the call stack is empty, it loads the first callback from the callback queue to the call stack.
This is why *the JavaScript Engine executes callbacks only after everything in the call stack has been processed*. Hence making asynchronous functions and non-blocking behaviour possible in JavaScript.

Promises and the micro-tasks queue ------
https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff

In JavaScript 6, the results of all asynchronous functions and DOM events are pushed to the callback queue. However, the results of promises are added to a different queue called the ***mirco-tasks queue***.

--promise-then-methods are executed before callbacks ----
The difference between the callback queue and the mirco-tasks queue is that *the mirco-tasks queue has a higher priority than the callback queue*, which means that promise jobs inside the micro-tasks queue will be executed before the callbacks inside the callback queue.

While the event loop is executing the tasks in the micro-tasks queue and in that time if another promise is resolved, it will be added to the end of the same micro-tasks queue, and it will be executed before the callbacks inside the callback queue no matter for how much time the callback is waiting to be executed.

Responsive Layout basiert auf (flexibel-)Grid

written by Hong Le Nguyen 08.2024

Grid Definition
  n gleich-breiten Columns
  n möglichen “Column-Span”s
  beliebig vielen Rows

Page Layout Gestaltung je nach Screen-Width
  m Rows
  Row-Gestaltung (= wie “Column-Span”s in Row sich ordnen)
  Zuordnung der Elementen zu Column-Spans innerhalb dem Row (= den Elementen mit den oben definierten Selectors und Styles markieren)

Ein Beispiel -------

```
// grid.html
<!DOCTYPE html>
<html>
<head>
   <title>Responsive Grid</title>
   <meta name="viewport" content="width=device-width">
   <link rel="stylesheet" type="text/css" href="grid.css">
</head>
<body>
<div class="row header">
   <div class="sm-col-span-2 lg-col-span-4"><h1>Blog</h1></div>
   <div class="sm-col-span-2 lg-col-span-4">
      <nav>
         <ul>
            <li><a href="#">Latest posts</a></li>
            <li><a href="#">Popular posts</a></li>
         </ul>
      </nav>
   </div>
</div>
<div class="row">
   <div class="sm-col-span-4 lg-col-span-3">
      <h2>Article title</h2>
      <p>02/12/2013</p>
      <p>Praesent commodo cursus magna, vel scelerisque nisl consectetur et. Cras justo odio... </p>
   </div>
   <div class="sm-col-span-4 lg-col-span-1">
      <h2>Related Articles</h2>
      <ul>
         <li><a href="#">Article item 1</a></li>
         <li><a href="#">Article item 2</a></li>
         <li><a href="#">Article item 3</a></li>
      </ul>
   </div>
</div>
</body>
</html>
```

```
/*  grid.css  4-Columns and n-Rows Grid */
.row:after {
   content: ' ';
   clear: both;
   display: block;
}

[class*="col-span"] {
   float: left;
   box-sizing: border-box;
   padding-left: 15px;
   padding-right: 15px;
}

.sm-col-span-1 { width: 25%; }
.sm-col-span-2 { width: 50%; }
.sm-col-span-3 { width: 75%; }
.sm-col-span-4 { width: 100%; }

@media screen and (min-width: 768px) {
   .lg-col-span-1 { width: 25%; }
   .lg-col-span-2 { width: 50%; }
   .lg-col-span-3 { width: 75%; }
   .lg-col-span-4 { width: 100%; }
}

/*Site specific*/
body {
   margin: 0;
   padding: 0;
   font-size: 14px;
   line-height: 18px;
}

.header {
   background: #304480;
   padding-top: 10px;
   padding-bottom: 10px;
   color: #fff;
}

nav { text-align: right; }
nav a { color: #fff; }
ul { padding: 0px; margin: 0px; }
ul li { list-style: none; }

@media screen and (min-width: 768px) {
   nav {
      text-align: center;
      padding-top: 10px;
      margin-top: 10px;
   }
   nav li { display: inline-block; }
}
```