JavaScript

setInterval, setTimeout, Promises

Introduction

Sometimes you may want to perform a action not right now but after a certain time. It is called **scheduling**.

There are two methods for it:

- **setTimeout** allows us to run a function once after the interval of time.
- **setInterval** allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In particular, they are supported in all browsers and Node.js.

setTimeout

Syntax:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Parameters:

Func|code: Function or a string of code to execute. Usually, that's a function. For historical reasons, **a string of code can be passed, but that's not recommended**.

Delay: The delay before run, in milliseconds (1000 ms = 1 second), by default 0.

arg1, arg2...: Arguments for the function

setTimeout

For instance, this code calls sayHi() after one second:

```
1  function sayHi() {
2    alert('Hello');
3  }
4
5  setTimeout(sayHi, 1000);
```

With arguments:

```
1  function sayHi(phrase, who) {
2    alert( phrase + ', ' + who );
3  }
4
5  setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Cancelling with clearTimeout

A call to setTimeout returns a “timer identifier” timerId that we can use to cancel the execution.

The syntax to cancel:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

setInterval

The setInterval method has the same syntax as setTimeout

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

All arguments have the same meaning. But unlike setTimeout it runs the function not only once, but regularly after the given interval of time.

To stop further calls, we should call **clearInterval(timerId)**.

The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Execution of setTimeout and setInterval

This schedules the execution of func as soon as possible. But the scheduler will invoke it only after the currently executing script is complete.

So the function is scheduled to run “right after” the current script.

For instance, this outputs “Hello”, then immediately “World”:

```
1  setTimeout(() => alert("World"));
2
3  alert("Hello");
```

Promises

Introduction

Asynchronous JavaScript refers to a programming approach that enables non-blocking execution of tasks, allowing for concurrent operations and improved responsiveness in web applications. JavaScript is a single-threaded and synchronous language, meaning that the code is executed in order, one line at a time

However, asynchronous JavaScript can be achieved using methods such as **callbacks and Promises**.

Promise

Imagine that you're a top singer, and fans ask day and night for your upcoming song.

To get some relief, you promise to send it to them when it's published. You give your fans a list. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, a fire in the studio, so that you can't publish the song, they will still be notified.

Everyone is happy: you, because the people don't crowd you anymore, and fans, because they won't miss the song.

Promise

This is a real-life analogy for things we often have in programming:

1. A **“producing code”** that does something and takes time. For instance, some code that loads the data over a network. That’s a “singer”.

2. A **“consuming code”** that wants the result of the “producing code” once it’s ready. Many functions may need that result. These are the “fans”.

3. **A promise is a special JavaScript object** that l**inks the “producing code” and the “consuming code” together.**

   In terms of our analogy: this is the “subscription list”. The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

Promise - Syntax (Producing Code)

The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

The function passed to new Promise is called the executor

When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result.

Its arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor

When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

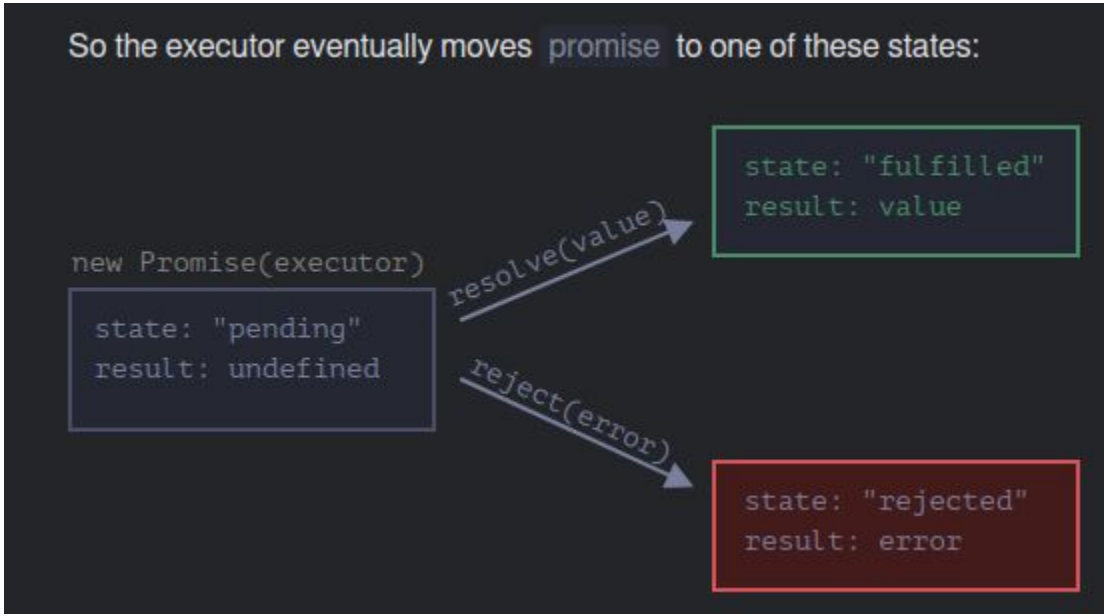
- resolve(value) — if the job is finished successfully, with result value.
- reject(error) — if an error has occurred, error is the error object.

Promise - Syntax (Producing Code)

The promise object returned by the new Promise constructor has these internal properties:

- state — initially "**pending**", then changes to either "**fulfilled**" when resolve is called or "**rejected**" when reject is called.
- result — **initially undefined**, then changes to value when resolve(value) is called or error when reject(error) is called.

Promise - Syntax (Producing Code)



Consumer - then/catch

A Promise object serves as a link between the executor (the “producing code” or “singer”) and the consuming functions (the “fans”), which will receive the result or error. Consuming functions can be registered (subscribed) using the methods .then and .catch.

The most important, fundamental one is .then.

The syntax is:

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

Consumer - then/catch

1. The first argument of .then is a function that runs when the promise is resolved and receives the result.

2. The second argument of .then is a function that runs when the promise is rejected and receives the error.

If we're interested only in successful completions, then we can provide only one function argument to .then:

Consumer - then/catch

Catch

If we're interested only in errors, then we can use null as the first argument: **.then(null, errorHandlingFunction)**

Or

we can use **.catch(errorHandlingFunction)**, which is exactly the same

Cleanup - Finally

Just like there's a finally clause in a regular try {...} catch {...}, there's finally in promises.

The call .finally(f) is similar to .then(f, f) in the sense that f runs always, when the promise is settled: be it resolve or reject.

The idea of finally is to set up a handler for performing cleanup/finalizing after the previous operations are complete.

E.g. stopping loading indicators, closing no longer needed connections, etc.

Cleanup - Finally

A finally handler has no arguments. In finally we don't know whether the promise is successful or not. That's all right, as our task is usually to perform "general" finalizing procedures.

```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve or maybe reject */
})
 // runs when the promise is settled, doesn't matter successfully or not
 .finally(() => stop loading indicator)
 // so the loading indicator is always stopped before we go on
 .then(result => show result, err => show error)
```

Please take a look at the example above: as you can see, the finally handler has no arguments, and the promise outcome is handled by the next handler.

A finally handler "passes through" the result or error to the next suitable handler.

For instance, here the result is passed through finally to then