1st November 2019

Scheduling: setTimeout and setInterval

We may decide to execute a function not right now, but at a certain time later. That's called "scheduling a call".

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

The 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 (not supported in IE9-)

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

```
function sayHi() {
  alert('Hello');
}

setTimeout(sayHi, 1000);
```

With arguments:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

If the first argument is a string, then JavaScript creates a function from it.

So, this will also work:

```
setTimeout("alert('Hello')", 1000);
```

But using strings is not recommended, use arrow functions instead of them, like this:

```
setTimeout(() => alert('Hello'), 1000);
```

Pass a function, but don't run it

Novice developers sometimes make a mistake by adding brackets `()` after the function:

```
// wrong!
setTimeout(sayHi(), 1000);
```

That doesn't work, because `setTimeout` expects a reference to a function. And here `sayHi()` runs the function, and the *result of its execution* is passed to `setTimeout`. In our case the result of `sayHi()` is `undefined` (the function returns nothing), so nothing is scheduled.

Canceling 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);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier


```

```
clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

As we can see from `alert` output, in a browser the timer identifier is a number. In other environments, this can be something else. For instance, Node.js returns a timer object with additional methods.

Again, there is no universal specification for these methods, so that's fine.

For browsers, timers are described in the timers section of HTML5 standard.

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);
```

Time goes on while `alert` is shown

In most browsers, including Chrome and Firefox the internal timer continues "ticking" while showing `alert/confirm/prompt`.

So if you run the code above and don't dismiss the `alert` window for some time, then in the next `alert` will be shown immediately as you do it. The actual interval between alerts will be shorter than 2 seconds.

Nested setTimeout

There are two ways of running something regularly.

One is `setInterval`. The other one is a nested `setTimeout`, like this:

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');

```

```
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

The `setTimeout` above schedules the next call right at the end of the current one `(*)`.

The nested `setTimeout` is a more flexible method than `setInterval`. This way the next call may be scheduled differently, depending on the results of the current one.

For instance, we need to write a service that sends a request to the server every 5 seconds asking for data, but in case the server is overloaded, it should increase the interval to 10, 20, 40 seconds…

Here's the pseudocode:

```
let delay = 5000;

let timerId = setTimeout(function request() {
  ...send request...

  if (request failed due to server overload) {
    // increase the interval to the next run
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

And if the functions that we're scheduling are CPU-hungry, then we can measure the time taken by the execution and plan the next call sooner or later.

Nested `setTimeout` allows to set the delay between the executions more precisely than `setInterval`.

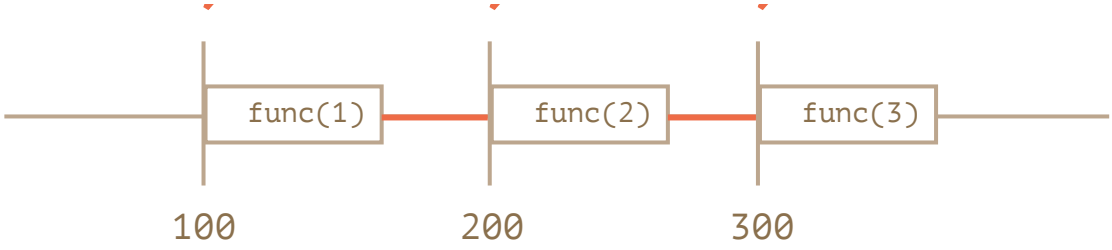
Let's compare two code fragments. The first one uses `setInterval`:

```
let i = 1;
setInterval(function() {
  func(i++);
}, 100);
```

The second one uses nested `setTimeout`:

```
let i = 1;
setTimeout(function run() {
  func(i++);
  setTimeout(run, 100);
}, 100);
```

For `setInterval` the internal scheduler will run `func(i++)` every 100ms:



Did you notice?

The real delay between `func` calls for `setInterval` is less than in the code!

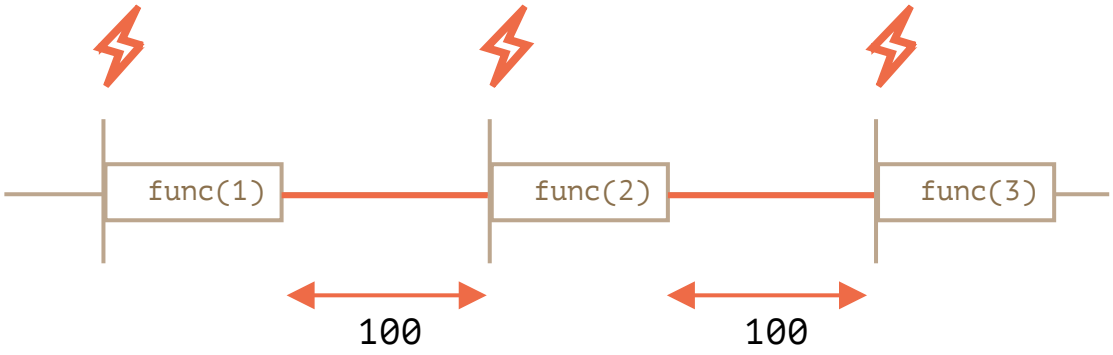
That's normal, because the time taken by `func` 's execution "consumes" a part of the interval.

It is possible that `func` 's execution turns out to be longer than we expected and takes more than 100ms.

In this case the engine waits for `func` to complete, then checks the scheduler and if the time is up, runs it again *immediately*.

In the edge case, if the function always executes longer than `delay` ms, then the calls will happen without a pause at all.

And here is the picture for the nested `setTimeout` :



The nested `setTimeout` guarantees the fixed delay (here 100ms).

That's because a new call is planned at the end of the previous one.

Garbage collection and setInterval/setTimeout callback

When a function is passed in `setInterval/setTimeout` , an internal reference is created to it and saved in the scheduler. It prevents the function from being garbage collected, even if there are no other references to it.

```
1  // the function stays in memory until the scheduler calls it
2  setTimeout(function() {...}, 100);
```

For `setInterval` the function stays in memory until `clearInterval` is called.

There's a side-effect. A function references the outer lexical environment, so, while it lives, outer variables live too. They may take much more memory than the function itself. So when we don't need the scheduled function anymore, it's better to cancel it, even if it's very small.

Zero delay setTimeout

There's a special use case: `setTimeout(func, 0)`, or just `setTimeout(func)`.

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");
```

The first line "puts the call into calendar after 0ms". But the scheduler will only "check the calendar" after the current script is complete, so `"Hello"` is first, and `"World"` – after it.

There are also advanced browser-related use cases of zero-delay timeout, that we'll discuss in the chapter Event loop: microtasks and macrotasks.

Zero delay is in fact not zero (in a browser)

In the browser, there's a limitation of how often nested timers can run. The HTML5 standard says: "after five nested timers, the interval is forced to be at least 4 milliseconds.".

Let's demonstrate what it means with the example below. The `setTimeout` call in it re-schedules itself with zero delay. Each call remembers the real time from the previous one in the `times` array. What do the real delays look like? Let's see:

```
1  let start = Date.now();
2  let times = [];
3
4  setTimeout(function run() {
5    times.push(Date.now() - start); // remember delay from the previous cal
6
7    if (start + 100 < Date.now()) alert(times); // show the delays after 10
8    else setTimeout(run); // else re-schedule
9  });
10
11 // an example of the output:
12 // 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

First timers run immediately (just as written in the spec), and then we see `9, 15, 20, 24...`. The 4+ ms obligatory delay between invocations comes into play.

The similar thing happens if we use `setInterval` instead of `setTimeout`: `setInterval(f)` runs `f` few times with zero-delay, and afterwards with 4+ ms delay.

That limitation comes from ancient times and many scripts rely on it, so it exists for historical reasons.

For server-side JavaScript, that limitation does not exist, and there exist other ways to schedule an immediate asynchronous job, like setImmediate for Node.js. So this note is browser-specific.

Summary

- Methods `setTimeout(func, delay, ...args)` and `setInterval(func, delay, ...args)` allow us to run the `func` once/regularly after `delay` milliseconds.
- To cancel the execution, we should call `clearTimeout/clearInterval` with the value returned by `setTimeout/setInterval`.
- Nested `setTimeout` calls are a more flexible alternative to `setInterval`, allowing us to set the time *between* executions more precisely.
- Zero delay scheduling with `setTimeout(func, 0)` (the same as `setTimeout(func)`) is used to schedule the call “as soon as possible, but after the current script is complete”.
- The browser limits the minimal delay for five or more nested call of `setTimeout` or for `setInterval` (after 5th call) to 4ms. That’s for historical reasons.

Please note that all scheduling methods do not *guarantee* the exact delay.

For example, the in-browser timer may slow down for a lot of reasons:

- The CPU is overloaded.
- The browser tab is in the background mode.
- The laptop is on battery.

All that may increase the minimal timer resolution (the minimal delay) to 300ms or even 1000ms depending on the browser and OS-level performance settings.

Tasks

Output every second

importance: 5

Write a function `printNumbers(from, to)` that outputs a number every second, starting from `from` and ending with `to`.

Make two variants of the solution.

1. Using `setInterval`.
2. Using nested `setTimeout`.

solution

What will setTimeout show?

importance: 5

In the code below there’s a `setTimeout` call scheduled, then a heavy calculation is run, that takes more than 100ms to finish.

When will the scheduled function run?

1. After the loop.
2. Before the loop.
3. In the beginning of the loop.

What is `alert` going to show?

```
let i = 0;

setTimeout(() => alert(i), 100); // ?

// assume that the time to execute this function is >100ms
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)