Understanding timers: setTimeout and setInterval

Ilya Kantor

Tweet

Like 23

Browser provides a built-in scheduler which allows to setup function calls for execution after given period of time.

setTimeout

The syntax is:

`var timerId = setTimeout(func|code, delay)`

func|code
    Function variable or the string of code to execute.

delay
    The delay in microseconds, 1000 microseconds = 1 second.

The execution will occur after the given delay.

For example, the code below calls `alert('hi')` after one second:

R
```
function f() {
  alert('Hi')
}
setTimeout(f, 1000)
```

If the first argument is a string, the interpreter creates an anonymous function at place from it.

So, this call works the same.

R
```
setTimeout("alert('Hi')", 1000)
```

Using a string version is not recommended, it causes problems with minificators. It's here mainly for compatibility with older JavaScript code.

Use anonymous functions instead:

R
```
setTimeout(function() { alert('Hi') }, 1000)
```

Cancelling the execution

The returned `timerId` can be used to cancel the action.

The syntax is: `clearTimeout(timerId)`.

In the example below the timeout is set and then cleared, so nothing happens.

R
```
var timerId = setTimeout(function() { alert(1) }, 1000)

```

```
clearTimeout(timerId)
```

`setInterval`

The `setInterval(func|code, delay)` method has same features as `setTimeout`.

It schedules the repeating execution after every `delay` microseconds, which can be stopped by `clearInterval` call.

The following example will give you an alert every 2 seconds until you press stop. Run it to see in action.

s

```
<input type="button" onclick="clearInterval(timer)" value="stop">

<script>
  var i = 1
  var timer = setInterval(function() { alert(i++) }, 2000)
</script>
```

Create the clock red-green-blue clock as given below **using `setInterval`**:

15:33:35
Start Stop

The starting code is here: tutorial/advanced/timing/clock-interval-src/index.html .

Open solution

Create the clock red-green-blue clock as given below **using `setTimeout`**:

15:34:05
Start Stop

The starting code is here: tutorial/advanced/timing/clock-timeout-src/index.html .

Open solution

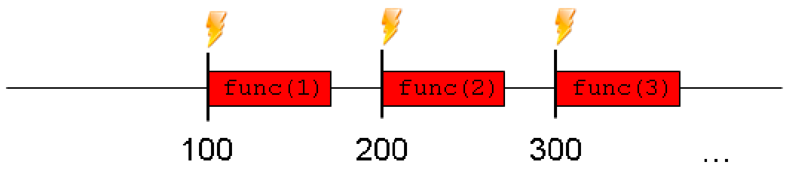
The real delay of `setInterval`

`setInterval(func, delay)` tries to execute the `func` every `delay` ms.

Real delay for `setInterval` is actually less than given.

Let's take a `setInterval(function() { func(i++) }, 100)` as an example. It executes `func` every 100 ms, increasing the counter every run.

In the picture below, the red brick is `func` execution time. The time between bricks the time between executions is actually less than the delay:

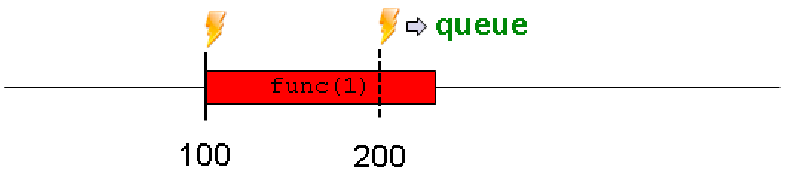


Interesting things start to happen when `func` takes more than the delay

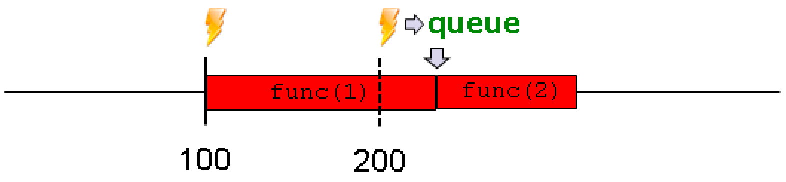
If the execution is impossible, it is queued.

The picture below shows an example of a long-running function.

The execution of `setInterval` gets queued and runs immediately when possible.

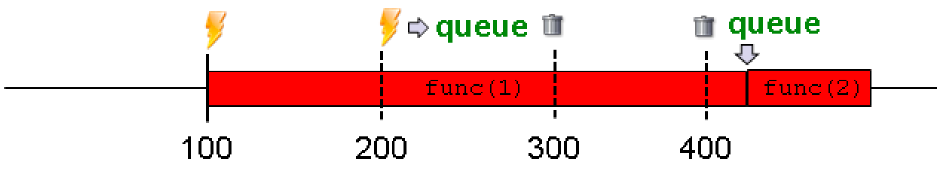


The real time between executions here is much more than `delay` ms.



There are cases when `func` takes more than *several* scheduled runs.
If the browser is busy, and the execuion is already queued, `setInterval` skips it.

On the picture below, `setInterval` tries to execute at 200 ms and queues the run. At 300 ms and 400 ms it wakes up again, but does nothing.



Only one execution can be queued.

Let's see how that affects real code. Run the example and await for an `alert`. Note that while the `alert` is shown, JavaScript execution is blocked, so we have a long-running function. Wait a while and press OK.

s
```
<input type="button" onclick="clearInterval(timer)" value="stop">

<script>
  var i = 1
  timer = setInterval(function() { alert(i++) }, 2000)
</script>
```

1. The browser runs the function every 2 seconds
2. **When you press alert** - the execution gets blocked and remains blocked while the alert is shown.
3. If you wait long enough, the browser queues next execution and ignores those after it.
4. **When you press OK and release alert** - the queued execution triggers immediately.
5. The next execution triggers with shorter delay. That's because scheduler wakes up each 2000ms. So, if the alert is released at 3500ms, then the next run is in 500ms.

`setInterval(func, delay)` does not guarantee a given `delay` between executions.

There are cases when the real delay is more or less than given.

In fact, it doesn't guarantee that there be any delay at all.

Repeating with known delay

When we need a fixed delay, rescheduling with `setTimeout` is used.

Here is the example which gives alert every two seconds with `setTimeout` instead of `setInterval`:

s
```
<input type="button" onclick="clearTImeout(timer)" value="stop">

<script>
  var i = 1
  var timer = setTimeout(function() {
    alert(i++)
    timer = setTimeout(arguments.callee, 2000)
  }, 2000)
```

```
</script>
```

The trick is to reschedule the new execution every call.

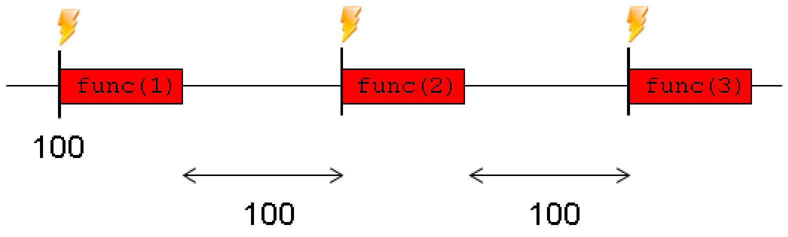
The example below demonstrates the syntax for a named function:

s
```
<input type="button" onclick="clearTImeout(timer)" value="stop">

<script>
  var i = 1

  function func() {
    alert(i++)
    timer = setTimeout(func, 2000)
  }
  var timer = setTimeout(func, 2000)
</script>
```

The execution timeline will have fixed delays between executions (picture for 100 ms):



The minimal delay

The timer resolution is limited. Actually, the minimal timer tick varies between 1ms and 15ms for modern browsers and can be larger for older ones.

If the timer resolution is 10, then there is no difference between `setTimeout(..,1)` and `setTimeout(..,9)`.

Let's see that in the next example. It runs several timers, from 2 to 20. Each timer increases the length of the corresponding `DIV`.

Run it in different browsers and notice that for most of them, several first `DIVs` animate identically. That's exactly because **the timer doesn't differ between too small values**.

s
```
<style> div { height: 18px; margin: 1px; background-color:green; }
</style>

<input type="button" value="Click to stop">

<script>
onload = function() {
  for(var i=0; i<=20; i+=2) {
    var div = document.createElement('div')
    div.innerHTML = i
    document.body.appendChild(div)
    animateDiv(div, i)
  }
}

function animateDiv(div, speed) {
  var timer = setInterval(function() {
    div.style.width = (parseInt(div.style.width||0)+2)%400 + 'px'
  }, speed)

  var stop = document.getElementsByTagName('input')[0]
  var prev = stop.onclick
  stop.onclick = function() {
    clearInterval(timer)
    prev && prev()
  }
}
</script>
```

```
Click to stop
0
2
4
6
8
10
12
14
16
18
```

Behavior of `setTimeout` and `setInterval` with `0 delay` is slightly different.

- In Opera, `setTimeout(.. ,0)`, is same as `setTimeout(.. ,10)`. It will execute *less* often than `setTimeout(.. ,2)`.
- In Internet Explorer, zero delay `setInterval(.., 0)` doesn't work. Changing the delay from 0 to 1 or using `setTimeout` helps.

Open the animation with `setTimeout`

In real world animations, it is not recommended to have many `setInterval/setTimeout` at the same time. They eat CPU.

It is much more preferred that a single `setInterval/setTimeout` call manages animation for multiple elements.

The execution context, `this`

The button below should change it's value to 'OK', but it doesn't work. Why?

```
<input type="button"
  onclick="setTimeout(function() { this.value='OK' }, 100)"
  value="Click me"
>
```

Click me

The reason is *wrong* `this`. **Functions executed by `setInterval/setTimeout` have `this=window`**, or `this=undefined` in ES5 strict mode.

The reference is usually passed through closure. The following is fine:

```
<input id="click-ok" type="button" value="Click me">
<script>
  document.getElementById('click-ok').onclick = function() {
    var self = this
    setTimeout(function() { self.value='OK' }, 100)
  }
</script>
```

Click me

Method call scheduling

In object-oriented code, a scheduled method call may not work:

R
```
function User(login) {
  this.login = login
  this.sayHi = function() {
    alert(this.login)
  }
}

var user = new User('John')

setTimeout(user.sayHi, 1000)
```

The `user.sayHi` indeed executes, but **`setTimeout` executes a function without the context**. It takes the bare function and schedules it.

To put it clear, these two `setTimeout` do the same:

```
setTimeout(user.sayHi, 1000)

var f = user.sayHi
setTimeout(f, 1000)
```

There are two ways to solve the problem. The first is to create an intermediate function:

R

```
function User(name) {
  this.name = name
  this.sayHi = function() {
    alert(this.name)
  }
}

var user = new User('John')

setTimeout(function() {
  user.sayHi()
}, 1000)
```

The second way is to bind `sayHi` to the object by referencing it through closure instead of using `this`:

R

```
function User(name) {
  this.name = name

  var self = this

  this.sayHi = function() {
    alert(self.name)
  }
}

var user = new User('John')

setTimeout(user.sayHi, 1000)
```

‹ Date/Time functions

Events and timing in-depth 深入事件和定时机制 ›

The content of this site is available under the terms of CC BY-NC-SA.

© Ilya Kantor, 2011.

"JavaScript™ is a registered trademark of Oracle corp. This site is not affiliated with Oracle corp.