**JavaScript的程序调度的两个函数**

有时我们并不想立即执行一个函数，而是等待特定一段时间之后再执行。这就是所谓的“计划调用（scheduling a call）”。

目前有两种方式可以实现：

* setTimeout 允许我们将函数推迟到一段时间间隔之后再执行。
* setInterval 允许我们重复运行一个函数，从一段时间间隔之后开始运行，之后以该时间间隔连续重复运行该函数。

这两个方法并不在 JavaScript 的规范中。但是大多数运行环境都有内建的调度程序，并且提供了这些方法。目前来讲，所有浏览器以及 Node.js 都支持这两个方法。

**一、setTimeout**

语法：

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

参数说明：

func|code想要执行的函数或代码字符串。 一般传入的都是函数。由于某些历史原因，支持传入代码字符串，但是不建议这样做。delay执行前的延时，以毫秒为单位（1000 毫秒 = 1 秒），默认值是 0；arg1，arg2…要传入被执行函数（或代码字符串）的参数列表（IE9 以下不支持）

例如，在下面这个示例中，sayHi() 方法会在 1 秒后执行：

function sayHi() {

alert('Hello');

}

setTimeout(sayHi, 1000);

带参数的情况：

function sayHi(phrase, who) {

alert( phrase + ', ' + who );

}

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

如果第一个参数位传入的是字符串，JavaScript 会自动为其创建一个函数。

所以这么写也是可以的：

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

但是，不建议使用字符串，我们可以使用箭头函数代替它们，如下所示：

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

**传入一个函数，但不要执行它**

新手开发者有时候会误将一对括号 () 加在函数后面：

// 错的！

setTimeout(sayHi(), 1000);

这样不行，因为 setTimeout 期望得到一个对函数的引用。而这里的 sayHi() 很明显是在执行函数，所以实际上传入 setTimeout 的是 **函数的执行结果**。在这个例子中，sayHi() 的执行结果是 undefined（也就是说函数没有返回任何结果），所以实际上什么也没有调度。

**二、用 clearTimeout 来取消调度**

setTimeout 在调用时会返回一个“定时器标识符（timer identifier）”，在我们的例子中是 timerId，我们可以使用它来取消执行。

取消调度的语法：

let timerId = setTimeout(...);

clearTimeout(timerId);

在下面的代码中，我们对一个函数进行了调度，紧接着取消了这次调度（中途反悔了）。所以最后什么也没发生：

let timerId = setTimeout(() => alert("never happens"), 1000);

alert(timerId); // 定时器标识符

clearTimeout(timerId);

alert(timerId); // 还是这个标识符（并没有因为调度被取消了而变成 null）

从 alert 的输出来看，在浏览器中，定时器标识符是一个数字。在其他环境中，可能是其他的东西。例如 Node.js 返回的是一个定时器对象，这个对象包含一系列方法。

再重申一遍，这些方法没有统一的规范定义，所以这没什么问题。

针对浏览器环境，定时器在 HTML5 的标准中有详细描述，详见 timers section。

**三、setInterval**

setInterval 方法和 setTimeout 的语法相同：

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

所有参数的意义也是相同的。不过与 setTimeout 只执行一次不同，setInterval 是每间隔给定的时间周期性执行。

想要阻止后续调用，我们需要调用 clearInterval(timerId)。

下面的例子将每间隔 2 秒就会输出一条消息。5 秒之后，输出停止：

// 每 2 秒重复一次

let timerId = setInterval(() => alert('tick'), 2000);

// 5 秒之后停止

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);

alert 弹窗显示的时候计时器依然在进行计时

在大多数浏览器中，包括 Chrome 和 Firefox，在显示 alert/confirm/prompt 弹窗时，内部的定时器仍旧会继续“嘀嗒”。

所以，在运行上面的代码时，如果在一定时间内没有关掉 alert 弹窗，那么在你关闭弹窗后，下一个 alert 会立即显示。两次 alert 之间的时间间隔将小于 2 秒。

**四、嵌套的 setTimeout**

周期性调度有两种方式。

一种是使用 setInterval，另外一种就是嵌套的 setTimeout，就像这样：

/\*\* instead of:

let timerId = setInterval(() => alert('tick'), 2000);

\*/

let timerId = setTimeout(function tick() {

alert('tick');

timerId = setTimeout(tick, 2000); // (\*)

}, 2000);

上面这个 setTimeout 在当前这一次函数执行完时 (\*) 立即调度下一次调用。

嵌套的 setTimeout 要比 setInterval 灵活得多。采用这种方式可以根据当前执行结果来调度下一次调用，因此下一次调用可以与当前这一次不同。

例如，我们要实现一个服务（server），每间隔 5 秒向服务器发送一个数据请求，但如果服务器过载了，那么就要降低请求频率，比如将间隔增加到 10、20、40 秒等。

以下是伪代码：

let delay = 5000;

let timerId = setTimeout(function request() {

...发送请求...

if (request failed due to server overload) {

// 下一次执行的间隔是当前的 2 倍

delay \*= 2;

}

timerId = setTimeout(request, delay);

}, delay);

并且，如果我们调度的函数占用大量的 CPU，那么我们可以测量执行所需要花费的时间，并安排下次调用是应该提前还是推迟。

**嵌套的 setTimeout 能够精确地设置两次执行之间的延时，而 setInterval 却不能。**

下面来比较这两个代码片段。第一个使用的是 setInterval：

let i = 1;

setInterval(function() {

func(i++);

}, 100);

第二个使用的是嵌套的 setTimeout：

let i = 1;

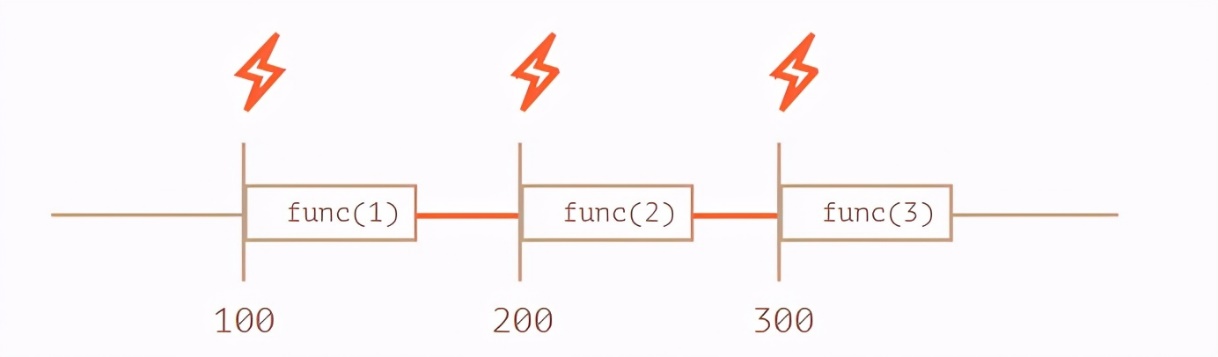
setTimeout(function run() {

func(i++);

setTimeout(run, 100);

}, 100);

对 setInterval 而言，内部的调度程序会每间隔 100 毫秒执行一次 func(i++)：



注意到了吗？

**使用 setInterval 时，func 函数的实际调用间隔要比代码中设定的时间间隔要短！**

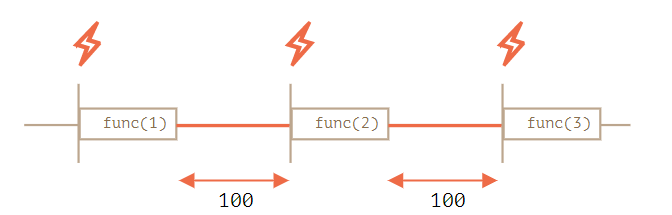
这也是正常的，因为 func 的执行所花费的时间“消耗”了一部分间隔时间。

也可能出现这种情况，就是 func 的执行所花费的时间比我们预期的时间更长，并且超出了 100 毫秒。

在这种情况下，JavaScript 引擎会等待 func 执行完成，然后检查调度程序，如果时间到了，则 **立即** 再次执行它。

极端情况下，如果函数每次执行时间都超过 delay 设置的时间，那么每次调用之间将完全没有停顿。

这是嵌套的 setTimeout 的示意图：



**嵌套的 setTimeout 就能确保延时的固定（这里是 100 毫秒）。**

这是因为下一次调用是在前一次调用完成时再调度的。

**垃圾回收和 setInterval/setTimeout 回调（callback）**

当一个函数传入 setInterval/setTimeout 时，将为其创建一个内部引用，并保存在调度程序中。这样，即使这个函数没有其他引用，也能防止垃圾回收器（GC）将其回收。

// 在调度程序调用这个函数之前，这个函数将一直存在于内存中

setTimeout(function() {...}, 100);

对于 setInterval，传入的函数也是一直存在于内存中，直到 clearInterval 被调用。

这里还要提到一个副作用。如果函数引用了外部变量（译注：闭包），那么只要这个函数还存在，外部变量也会随之存在。它们可能比函数本身占用更多的内存。因此，当我们不再需要调度函数时，最好取消它，即使这是个（占用内存）很小的函数。

**五、零延时的 setTimeout**

这儿有一种特殊的用法：setTimeout(func, 0)，或者仅仅是 setTimeout(func)。

这样调度可以让 func 尽快执行。但是只有在当前正在执行的脚本执行完成后，调度程序才会调用它。

也就是说，该函数被调度在当前脚本执行完成“之后”立即执行。

例如，下面这段代码会先输出 “Hello”，然后立即输出 “World”：

setTimeout(() => alert("World"));

alert("Hello");

第一行代码“将调用安排到日程（calendar）0 毫秒处”。但是调度程序只有在当前脚本执行完毕时才会去“检查日程”，所以先输出 "Hello"，然后才输出 "World"。

此外，还有与浏览器相关的 0 延时 timeout 的高级用例，我们将在 事件循环：微任务和宏任务 一章中详细讲解。

**零延时实际上不为零（在浏览器中）**

在浏览器环境下，嵌套定时器的运行频率是受限制的。根据 HTML5 标准 所讲：“经过 5 重嵌套定时器之后，时间间隔被强制设定为至少 4 毫秒”。

让我们用下面的示例来看看这到底是什么意思。其中 setTimeout 调用会以零延时重新调度自身的调用。每次调用都会在 times 数组中记录上一次调用的实际时间。那么真正的延迟是什么样的？让我们来看看：

let start = Date.now();

let times = [];

setTimeout(function run() {

times.push(Date.now() - start); // 保存前一个调用的延时

if (start + 100 < Date.now()) alert(times); // 100 毫秒之后，显示延时信息

else setTimeout(run); // 否则重新调度

});

// 输出示例：

// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100

第一次，定时器是立即执行的（正如规范里所描述的那样），接下来我们可以看到 9, 15, 20, 24...。两次调用之间必须经过 4 毫秒以上的强制延时。（译注：这里作者没说清楚，timer 数组里存放的是每次定时器运行的时刻与 start 的差值，所以数字只会越来越大，实际上前后调用的延时是数组值的差值。示例中前几次都是 1，所以延时为 0）

如果我们使用 setInterval 而不是 setTimeout，也会发生类似的情况：setInterval(f) 会以零延时运行几次 f，然后以 4 毫秒以上的强制延时运行。

这个限制来自“远古时代”，并且许多脚本都依赖于此，所以这个机制也就存在至今。

对于服务端的 JavaScript，就没有这个限制，并且还有其他调度即时异步任务的方式。例如 Node.js 的 setImmediate。因此，这个提醒只是针对浏览器环境的。

**六、总结**

* setTimeout(func, delay, ...args) 和 setInterval(func, delay, ...args) 方法允许我们在 delay 毫秒之后运行 func 一次或以 delay 毫秒为时间间隔周期性运行 func。
* 要取消函数的执行，我们应该调用 clearInterval/clearTimeout，并将 setInterval/setTimeout 返回的值作为入参传入。
* 嵌套的 setTimeout 比 setInterval 用起来更加灵活，允许我们更精确地设置两次执行之间的时间。
* 零延时调度 setTimeout(func, 0)（与 setTimeout(func) 相同）用来调度需要尽快执行的调用，但是会在当前脚本执行完成后进行调用。
* 浏览器会将 setTimeout 或 setInterval 的五层或更多层嵌套调用（调用五次之后）的最小延时限制在 4ms。这是历史遗留问题。

请注意，所有的调度方法都不能 **保证** 确切的延时。

例如，浏览器内的计时器可能由于许多原因而变慢：

* CPU 过载。
* 浏览器页签处于后台模式。
* 笔记本电脑用的是电池供电（译注：使用电池供电会以降低性能为代价提升续航）。

所有这些因素，可能会将定时器的最小计时器分辨率（最小延迟）增加到 300ms 甚至 1000ms，具体以浏览器及其设置为准。