ph_piter 17 октября 2018 в 10:32

Таймеры JavaScript: все что нужно знать

Автор оригинала: Samer Buna

Блог компании Издательский дом «Питер», JavaScript, Программирование, Node.JS

Перевод

Здравствуйте, коллеги. Давным-давно на Хабре уже переводилась статья под авторством Джона Резига как раз на эту тему. Прошло уж 10 лет, а тема по-прежнему требует разъяснений. Поэтому предлагаем интересующимся почитать статью Самера Буны, в которой дается не только теоретический обзор таймеров в JavaScript (в контексте Node.js), но и задачи на них.

Несколько недель назад я опубликовал в Твиттере следующий вопрос с одного собеседования:

> «Где находится исходный код функций setTimeout и setInterval? Где бы вы его искали? Погуглить нельзя :)»

***Ответьте на него для себя, а потом читайте дальше ***

Примерно половина ответов на этот твит были неверными. Нет, дело НЕ СВЯЗАНО с V8 (или другими VM)!!! Функции вроде `setTimeout` и `setInterval`, гордо именуемые «Таймерами JavaScript», не входят ни в одну спецификацию ECMAScript или в реализацию движка JavaScript. Функции-таймеры реализуются на уровне браузера, поэтому в разных браузерах их реализации отличаются. Также таймеры нативно реализуются в самой среде исполнения Node.js.

В браузерах основные функции-таймеры относятся к интерфейсу `Window`, также связанному с некоторыми другими функциями и объектами. Этот интерфейс предоставляет ко всем своим элементам глобальный доступ в главной области видимости JavaScript. Вот почему функцию `setTimeout` можно выполнять непосредственно в консоли браузера.

В Node таймеры входят в состав объекта `global`, который устроен подобно браузерному интерфейсу `Window`. Исходный код таймеров в Node показан здесь.

Кому-то может показаться, что это просто плохой вопрос с собеседования – какой вообще прок знать подобное?! Я, как JavaScript-разработчик, думаю так: предполагается, что вы должны это знать, поскольку обратное может свидетельствовать, что вы не вполне понимаете, как V8 (и другие виртуальные машины) взаимодействует с браузерами и Node.

Рассмотрим несколько примеров и решим парочку задач на таймеры, давайте?

Для запуска примеров из этой статьи можно воспользоваться командой node. Большинство рассмотренных здесь примеров фигурируют в моем курсе Getting Started with Node.js на Pluralsight.

Отложенное выполнение функции

Таймеры – это функции высшего порядка, при помощи которых можно откладывать или повторять выполнение других функций (таймер получает такую функцию в качестве первого аргумента).

Вот пример отложенного выполнения:

```
// example1.js
setTimeout(
  () => {
    console.log('Hello after 4 seconds');
  },
  4 * 1000
);
```

В этом примере при помощи `setTimeout` вывод приветственного сообщения откладывается на 4 секунды. Второй аргумент `setTimeout` — это задержка (в мс). Я умножаю 4 на 1000, чтобы получилось 4 секунды.

Первый аргумент `setTimeout` – функция, выполнение которой будет откладываться.
Если выполнить файл `example1.js` командой node, Node приостановится на 4 секунды, а затем выведет приветственное сообщение (после чего последует выход).

Обратите внимание: первый аргумент `setTimeout` — это всего лишь **ссылка на функцию**. Она не должна быть встроенной функцией – такой, как `example1.js`. Вот тот же самый пример без использования встроенной функции:

```
const func = () => {
  console.log('Hello after 4 seconds');
};
setTimeout(func, 4 * 1000);
```

Передача аргументов

Если функция, для задержки которой используется `setTimeout`, принимает какие-либо аргументы, то можно использовать оставшиеся аргументы самой функции `setTimeout` (после тех 2, которые мы уже успели изучить) для переброски значений аргументов к отложенной функции.

```
// Для: func(arg1, arg2, arg3, ...)
// Можно использовать: setTimeout(func, delay, arg1, arg2, arg3, ...)
```

Вот пример:

```
// example2.js
const rocks = who => {
  console.log(who + ' rocks');
};
setTimeout(rocks, 2 * 1000, 'Node.js');
```

Вышеприведенная функция `rocks`, отложенная на 2 секунды, принимает аргумент `who`, и вызов `setTimeout` передает ей значение “Node.js” в качестве такого аргумента `who`.

При выполнении `example2.js` командой `node` фраза “Node.js rocks” будет выведена на экран через 2 секунды.

Задача на таймеры #1

Итак, опираясь на уже изученный материал о `setTimeout`, выведем 2 следующих сообщения после соответствующих задержек.

- Сообщение “Hello after 4 seconds” выводим через 4 секунды.
- Сообщение “Hello after 8 seconds” выводим через 8 секунд.

Ограничение

В вашем решении можно определить всего одну функцию, содержащую встроенные функции. Это означает, что множество вызовов `setTimeout` должны будут использовать одну и ту же функцию.

Решение

Вот как я бы решил эту задачу:

```
// solution1.js
const theOneFunc = delay => {
  console.log('Hello after ' + delay + ' seconds');
};
setTimeout(theOneFunc, 4 * 1000, 4);
setTimeout(theOneFunc, 8 * 1000, 8);
```

У меня `theOneFunc` получает аргумент `delay` и использует значение данного аргумента `delay` в сообщении, выводимом на экран. Таким образом, функция может выводить разные сообщения в зависимости от того, какое значение задержки мы ей сообщим.

Затем я использовал `theOneFunc` в двух вызовах `setTimeout`, причем, первый вызов срабатывает через 4 секунды, а второй – через 8 секунд. Оба эти вызова `setTimeout` также получают 3-й аргумент, представляющий аргумент `delay` для `theOneFunc`.

Выполнив файл `solution1.js` командой node, мы выведем на экран требования задачи, причем, первое сообщение появится через 4 секунды, а второе — через 8 секунд.

Повторяем выполнение функции

А что, если бы я задал вам выводить сообщение каждые 4 секунды, неограниченно долго?
Конечно, можно заключить `setTimeout` в цикл, но в API таймеров также предлагается функция `setInterval`, при помощи которой можно запрограммировать «вечное» выполнение какой-либо операции.

Вот пример `setInterval`:

```
// example3.js
setInterval(
  () => console.log('Hello every 3 seconds'),
  3000
);
```

Этот код будет выводить сообщение каждые 3 секунды. Если выполнить `example3.js` командой `node`, то Node будет выводить

эту команду до тех пор, пока вы принудительно не завершите процесс (CTRL+C).

Отмена таймеров

Поскольку при вызове функции таймера назначается действие, это действие также можно отменить, прежде, чем он будет выполнен.

Вызов `setTimeout` возвращает ID таймера, и можно использовать этот ID таймера при вызове `clearTimeout`, чтобы отменить таймер. Вот пример:

```
// example4.js
const timerId = setTimeout(
  () => console.log('You will not see this one!'),
  0
);
clearTimeout(timerId);
```

Этот простой таймер должен срабатывать через 0 мс (то есть, сразу же), но этого не произойдет, поскольку мы захватываем значение `timerId` и немедленно отменяем этот таймер при помощи вызова `clearTimeout`.

При выполнении `example4.js` командой `node`, Node ничего не напечатает — процесс просто сразу же завершится.

Кстати, в Node.js предусмотрен и другой способ задать `setTimeout` со значением 0 мс. В API таймеров Node.js есть еще одна функция под названием `setImmediate`, и она в принципе делает то же самое, что и `setTimeout` со значением 0 мс, но в данном случае задержку можно не указывать:

```
setImmediate(
  () => console.log('I am equivalent to setTimeout with 0 ms'),
);
```

Функция `setImmediate` поддерживается не во всех браузерах. Не используйте ее в клиентском коде.

Наряду с `clearTimeout` есть функция `clearInterval`, которая делает то же самое, но с вызовами `setInerval`, а также есть вызов `clearImmediate`.

Задержка таймера – вещь не гарантированная

Вы заметили, что в предыдущем примере при выполнении операции с `setTimeout` после 0 мс эта операция происходит не сразу же (после `setTimeout`), а только после того, как будет целиком выполнен весь код скрипта (в том числе, вызов `clearTimeout`)?

Позвольте мне пояснить этот момент на примере. Вот простой вызов `setTimeout`, который должен бы сработать через полсекунды — но этого не происходит:

```
// example5.js
setTimeout(
  () => console.log('Hello after 0.5 seconds. MAYBE!'),
  500,
);
for (let i = 0; i < 1e10; i++) {
  // Синхронно блокируем операции
}
```

Сразу после определения таймера в данном примере мы синхронно блокируем среду времени выполнения большим циклом `for`. Значение `1e10` равно 1 с 10 нулями, поэтому цикл длится 10 миллиардов процессорных тактов (в принципе, так имитируется перегруженный процессор). Node ничего не может сделать, пока этот цикл не завершится.

Разумеется, на практике так делать очень плохо, но данный пример помогает понять, что задержка `setTimeout` – это не гарантированное, а, скорее, **минимальное значение**. Величина 500 мс означает, что задержка продлится минимум 500 мс. На самом деле, скрипту потребуется гораздо больше времени для вывода приветственной строки на экран. Сначала ему придется

дождаться, пока завершится блокирующий цикл.

Задача на таймеры #2

Напишите скрипт, который будет выводить сообщение “Hello World” раз в секунду, но всего 5 раз. После 5 итераций скрипт должен вывести сообщение “Done”, после чего процесс Node завершится.

Ограничение: при решении данной задачи нельзя вызывать `setTimeout`.

Подсказка: нужен счетчик.

Решение

Вот как я бы решил эту задачу:

```
let counter = 0;
const intervalId = setInterval(() => {
  console.log('Hello World');
  counter += 1;
if (counter === 5) {
    console.log('Done');
    clearInterval(intervalId);
  }
}, 1000);
```

В качестве исходного значения `counter` я задал 0, а затем вызвал `setInterval`, берущий его id.

Отложенная функция будет выводить сообщение и всякий раз при этом увеличивать счетчик на единицу. Внутри отложенной функции у нас инструкция if, которая будет проверять, не прошло ли уже 5 итераций. По истечении 5 итераций программа выведет “Done” и очистит значение интервала, воспользовавшись захваченной константой `intervalId`. Задержка интервала — 1000 мс.

«Кто» именно вызывает отложенные функции?

При использовании ключевого слова JavaScript `this` внутри обычной функции, вот так например:

```
function whoCalledMe() {
  console.log('Caller is', this);
}
```

значение в ключевом слове `this` будет соответствовать **вызывающей стороне**. Если определить вышеупомянутую функцию внутри Node REPL, то вызывать ее будет объект `global`. Если определить функцию в консоли браузера, то вызывать ее будет объект `window`.

Давайте определим функцию как свойство объекта, чтобы стало немного понятнее:

```
const obj = {
  id: '42',
  whoCalledMe() {
    console.log('Caller is', this);
  }
};
// Теперь ссылка на функцию такова: obj.whoCallMe
```

Теперь, когда при работе с функцией `obj.whoCallMe` мы будем напрямую использовать ссылку на нее, в качестве вызывающей стороны будет выступать объект `obj` (идентифицируемый по своему `id`):

```
~ $
~ $ node
> const obj = {
...   id: '42',
...   whoCalledMe() {
.....     console.log('Caller is ', this);
.....   }
... };
undefined
>
> obj.whoCalledMe()
Caller is  { id: '42', whoCalledMe: [Function: whoCalledMe] }
undefined
>
```

А теперь вопрос: кто будет вызывающей стороной, если передать ссылку на `obj.whoCallMe` вызову `setTimetout`?

```
// Какой текст будет выведен в данном случае??
setTimeout(obj.whoCalledMe, 0);
```

Кто в данном случае вызывающий?

Ответ будет отличаться в зависимости от того, где выполняется функция таймера. В данном случае просто недопустима зависимость от того, кто — вызывающая сторона. Вы утратите контроль над вызывающей стороной, поскольку именно от реализации таймера будет зависеть, кто в данном случае вызывает вашу функцию. Если протестировать этот код в Node REPL, то вызывающей стороной окажется объект `Timeout`:

```
> Caller is  Timeout {
  _called: true,
```

Обратите внимание: это важно лишь в случае, когда ключевое слово JavaScript `this` используется внутри обычных функций. При использовании стрелочных функций вызывающая сторона вас вообще не должна беспокоить.

Задача на таймеры #3

Напишите скрипт, который будет непрерывно выводить сообщение “Hello World” с варьирующимися задержками. Начните с односекундной задержки, после чего на каждой итерации увеличивайте ее на секунду. На второй итерации задержка будет 2 секунды. На третьей — три, и так далее.

Включите задержку в выводимое сообщение. У вас должен получиться примерно такой вывод:

```
Hello World. 1
Hello World. 2
Hello World. 3
...
```

Ограничения: переменные можно определять только при помощи const. При помощи let или var — нельзя.

Решение

Поскольку длительность задержки в данной задаче – величина переменная, использовать `setInterval` здесь нельзя, но можно вручную настроить интервальное выполнение при помощи `setTimeout` внутри рекурсивного вызова. Первая выполненная функция с `setTimeout` будет создавать следующий таймер, и так далее.

Кроме того, поскольку нельзя использовать `let`/`var`, у нас не может быть счетчика для приращения задержки при каждом рекурсивном вызове; вместо этого можно воспользоваться аргументами рекурсивной функции, чтобы выполнять приращение

во время рекурсивного вызова.

Вот как можно было бы решить эту задачу:

```
const greeting = delay =>
  setTimeout(() => {
    console.log('Hello World. ' + delay);
    greeting(delay + 1);
  }, delay * 1000);
greeting(1);
```

Задача на таймеры #4

Напишите скрипт, который будет выводить сообщение “Hello World” с такой же структурой задержек, как и в задаче #3, но на этот раз группами по 5 сообщений, а в группах будет основной интервал задержки. Для первой группы из 5 сообщений выбираем исходную задержку в 100 мс, для следующей – 200 мс, для третьей – 300 мс и так далее.

Вот как должен работать этот скрипт:

- На отметке 100 мс скрипт впервые выводит “Hello World”, и делает так 5 раз с интервалом, нарастающим по 100 мс. Первое сообщение появится через 100 мс, второе через 200 мс и т.д.
- После первых 5 сообщений скрипт должен увеличивать основную задержку уже на 200 мс. Таким образом, 6-е сообщение будет выведено через 500 мс + 200 мс (700 мс), 7-е — 900 мс, 8-е сообщение – через 1100 мс, и так далее.
- После 10 сообщений скрипт должен увеличивать основной интервал задержки на 300 мс. 11-е сообщение должно быть выведено через 500 мс + 1000 мс + 300 мс (18000 мс). 12-е сообщение должно быть выведено через 2100 мс, и т.д.

По такому принципу программа должна работать неограниченно долго.

Включите задержку в выводимое сообщение. У вас должен получиться примерно такой вывод (без комментариев):

```
Hello World. 100 // При 100 мс
Hello World. 100 // При 200 мс
Hello World. 100 // При 300 мс
Hello World. 100 // При 400 мс
Hello World. 100 // При 500 мс
Hello World. 200 // При 700 мс
Hello World. 200 // При 900 мс
Hello World. 200 // При 1100 мс
...
```

Ограничения: Можно использовать лишь вызовы `setInterval` (а не `setTimeout`) и только ОДНУ инструкцию `if`.

Решение

Поскольку мы можем работать только с вызовами `setInterval`, здесь нам потребуется использовать рекурсию, а также увеличивать задержку следующего вызова `setInterval`. Кроме того, инструкция `if` понадобится нам, чтобы это стало происходить лишь после 5 вызовов этой рекурсивной функции.

Вот возможное решение:

```
let lastIntervalId, counter = 5;
const greeting = delay => {
  if (counter === 5) {
    clearInterval(lastIntervalId);
    lastIntervalId = setInterval(() => {
      console.log('Hello World. ', delay);
      greeting(delay + 100);
    }, delay);
    counter = 0;
  }
```

```
counter += 1;
};
greeting(100);
```

Спасибо всем, кто дочитал.

Теги: javascript, node.js, front-end, web-программирование, задачи для программистов

Хабы: Блог компании Издательский дом «Питер», JavaScript, Программирование, Node.JS

ПОХОЖИЕ ПУБЛИКАЦИИ

14 мая 2015 в 12:09
Фреймворки JavaScript. Как изучить их по-быстрому
+10 62k 452 10

8 октября 2014 в 16:49
Node.js в действии
+17 26,4k 143 5

10 февраля 2014 в 11:47
Разработка Backbone.js приложений
+4 17,9k 27 27

Комментарии 12

XopHeT 17 октября 2018 в 10:42 0

del

Ответить

ph_piter 17 октября 2018 в 10:48 +1

Спасибо, поправили

Ответить

webdevium 17 октября 2018 в 11:55 –2

Я понимаю, что это перевод, но все же есть маленькое замечание:
в первом примере и во всех последующих, выполнение откладывается не на X секунд, а не менее чем на X секунд. Такова природа JS.

Ответить