Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

- `setTimeout` позволяет вызвать функцию **один раз** через определённый интервал времени.
- `setInterval` позволяет вызывать функцию **регулярно**, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

setTimeout

Синтаксис:

```
let timerId = setTimeout(func|code, [delay], [arg1], [a
```

Параметры:

`func|code`

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

`delay`

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

`arg1, arg2 …`

Аргументы, передаваемые в функцию (не поддерживается в IE9-)

Например, данный код вызывает `sayHi()` спустя одну секунду:

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

setTimeout(sayHi, 1000);
```

С аргументами:

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

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Д
```

Если первый аргумент является строкой, то JavaScript создаст из неё функцию.

Это также будет работать:


```
1  setTimeout("alert('Привет')", 1000);
```

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

```
1  setTimeout(() => alert('Привет'), 1000);
```

Передавайте функцию, но не запускайте её

Начинающие разработчики иногда ошибаются, добавляя скобки `()` после функции:

```
1  // не правильно!
2  setTimeout(sayHi(), 1000);
```

Это не работает, потому что `setTimeout` ожидает ссылку на функцию. Здесь `sayHi()` запускает выполнение функции, и *результат выполнения* отправляется в `setTimeout`. В нашем случае результатом выполнения `sayHi()` является `undefined` (так как функция ничего не возвращает), поэтому ничего не планируется.

Отмена через clearTimeout

Вызов `setTimeout` возвращает «идентификатор таймера» `timerId`, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

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

В коде ниже планируем вызов функции и затем отменяем его (просто передумали). В результате ничего не происходит:

```
1  let timerId = setTimeout(() => alert("ничего не происхо
2  alert(timerId); // идентификатор таймера
3
4  clearTimeout(timerId);
5  alert(timerId); // тот же идентификатор (не принимает з
```

Как мы видим из вывода `alert`, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, Node.js возвращает объект таймера с дополнительными методами.

Повторюсь, что нет единой спецификации на эти методы, поэтому такое поведение является нормальным.

Для браузеров таймеры описаны в разделе таймеров стандарта HTML5.

setInterval

Метод `setInterval` имеет такой же синтаксис как `setTimeout`:

```
1  let timerId = setInterval(func|code, [delay], [arg1], [
```

Все аргументы имеют такое же значение. Но отличие этого метода от `setTimeout` в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать `clearInterval(timerId)`.

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

```
// повторить с интервалом 2 секунды
let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд
setTimeout(() => { clearInterval(timerId); alert('stop'
```

Во время показа `alert` время тоже идёт

В большинстве браузеров, включая Chrome и Firefox, внутренний счётчик продолжает тикать во время показа `alert/confirm/prompt`.

Так что если вы запустите код выше и подождёте с закрытием `alert` несколько секунд, то следующий `alert` будет показан сразу, как только вы закроете предыдущий. Интервал времени между сообщениями `alert` будет короче, чем 2 секунды.

Рекурсивный setTimeout

Есть два способа запускать что-то регулярно.

Один из них `setInterval`. Другим является рекурсивный `setTimeout`. Например:

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

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

Метод `setTimeout` выше планирует следующий вызов прямо после окончания текущего `(*)`.

Рекурсивный `setTimeout` – более гибкий метод, чем `setInterval`. С его помощью последующий вызов может быть задан по-разному в зависимости от результатов предыдущего.

Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд... Вот псевдокод:

```
let delay = 5000;

let timerId = setTimeout(function request() {
  ...отправить запрос...

  if (ошибка запроса из-за перегрузки сервера) {
    // увеличить интервал для следующего запроса
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

А если функции, которые мы планируем, ресурсоёмкие и требуют времени, то мы можем измерить время, затраченное на выполнение, и спланировать следующий вызов раньше или позже.

Рекурсивный `setTimeout` позволяет задать задержку между выполнениями более точно, чем `setInterval`.

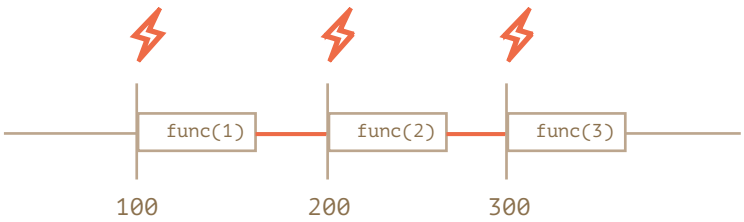
Сравним два фрагмента кода. Первый использует `setInterval`:

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

Второй использует рекурсивный `setTimeout`:

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

Для `setInterval` внутренний планировщик будет выполнять `func(i)` каждые 100 мс:



Обратили внимание?

Реальная задержка между вызовами `func` с помощью `setInterval` меньше, чем указано в коде!

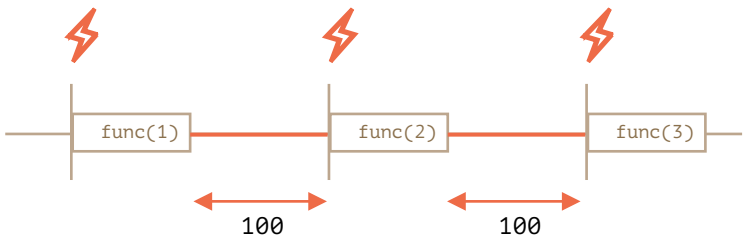
Это нормально, потому что время, затраченное на выполнение `func`, использует часть заданного интервала времени.

Вполне возможно, что выполнение `func` будет дольше, чем мы ожидали, и займёт более 100 мс.

В данном случае движок ждёт окончания выполнения `func` и затем проверяет планировщик и, если время истекло, *немедленно* запускает его снова.

В крайнем случае, если функция всегда выполняется дольше, чем задержка `delay`, то вызовы будут выполняться без задержек вообще.

Ниже представлено изображение, показывающее процесс работы рекурсивного `setTimeout`:



Рекурсивный `setTimeout` гарантирует фиксированную задержку (здесь 100 мс).

Это потому, что новый вызов планируется в конце предыдущего.

Сборка мусора и колбэк setTimeout/setInterval

Когда функция передаётся в `setInterval/setTimeout`, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик мусора, даже если на неё нет других ссылок.

```
// функция остаётся в памяти до тех пор, пока план
setTimeout(function() {...}, 100);
```

Для `setInterval` функция остаётся в памяти до тех пор, пока не будет вызван `clearInterval`.

Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

setTimeout с нулевой задержкой

Особый вариант использования: `setTimeout(func, 0)` или просто `setTimeout(func)`.

Это планирует вызов `func` настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода.

Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

```
setTimeout(() => alert("Мир"));

alert("Привет");
```

Первая строка помещает вызов в «календарь» через 0 мс. Но планировщик проверит «календарь» только после того, как текущий код завершится. Поэтому `"Привет"` выводится первым, а `"Мир"` – после него.

Есть и более продвинутые случаи использования нулевой задержки в браузерах, которые мы рассмотрим в главе Событийный цикл: микрозадачи и макрозадачи.

Минимальная задержка вложенных таймеров в браузере

В браузере есть ограничение на то, как часто внутренние счётчики могут выполняться. В стандарте HTML5 говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.».

Продемонстрируем в примере ниже, что это означает. Вызов `setTimeout` повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время от предыдущего вызова в массиве `times`. Какова реальная задержка? Посмотрим:

```
let start = Date.now();
let times = [];

setTimeout(function run() {
  times.push(Date.now() - start); // запоминаем за

  if (start + 100 < Date.now()) alert(times); // r
  else setTimeout(run); // если нужно ещё запланир
});

// пример вывода:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,7
```

Первый таймер запускается сразу (как и указано в спецификации), а затем задержка вступает в игру, и мы видим `9, 15, 20, 24...`.

Аналогичное происходит при использовании `setInterval` вместо `setTimeout`: `setInterval(f)` запускает `f` несколько раз с нулевой задержкой, а затем с задержкой 4+ мс.

Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам.

Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, setImmediate для Node.js. Так что это ограничение относится только к браузерам.

Итого

- Методы `setInterval(func, delay, ...args)` и `setTimeout(func, delay, ...args)` позволяют выполнять `func` регулярно или только один раз после задержки `delay`, заданной в мс.
- Для отмены выполнения необходимо вызвать `clearInterval/clearTimeout` со значением, которое возвращают методы `setInterval/setTimeout`.
- Вложенный вызов `setTimeout` является более гибкой альтернативой `setInterval`. Также он позволяет более точно задать интервал между выполнениями.
- Планирование с нулевой задержкой `setTimeout(func,0)` или, что то же самое, `setTimeout(func)` используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.
- Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами `setTimeout`, а также для `setInterval`, начиная с 5-го вызова.

Обратим внимание, что все методы планирования *не гарантируют* точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

- Перегружен процессор.
- Вкладка браузера в фоновом режиме.
- Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

Поделиться

Редактировать на GitHub

Задачи

Вывод каждую секунду

важность: 5

Напишите функцию `printNumbers(from, to)`, которая выводит число каждую секунду, начиная от `from` и заканчивая `to`.

Сделайте два варианта решения.

1. Используя `setInterval`.
2. Используя рекурсивный `setTimeout`.

решение

Что покажет setTimeout?

важность: 5

В приведённом ниже коде запланирован вызов `setTimeout`, а затем выполняется сложное вычисление, для завершения которого требуется более 100 мс.

Когда будет выполнена запланированная функция?

1. После цикла.
2. Перед циклом.
3. В начале цикла.

Что покажет `alert`?

```
let i = 0;

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

// предположим, что время выполнения этой функции >100 
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

решение

Комментарии

перед тем как писать...