JavaScript асинхронность - начало

Планирование: setTimeout и setInterval;
Сетевые запросы;
Событийный цикл: микро и макро задачи.

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 определенный 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

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

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

Типичная ошибка - Передавайте функцию, но не запускайте её

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

// неправильно!

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.

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

```
let timerId = setTimeout(() => alert("ничего не происходит"), 1000);
alert(timerId); // идентификатор таймера

clearTimeout(timerId);
alert(timerId); // тот же идентификатор (не принимает значение null после отмены)
```

setInterval

Метод setInterval имеет такой же синтаксис как setTimeout:

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).

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

Вложенный setTimeout

Есть два способа запускать что-то регулярно.

Один из них setInterval. Другим является вложенный setTimeout

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

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

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

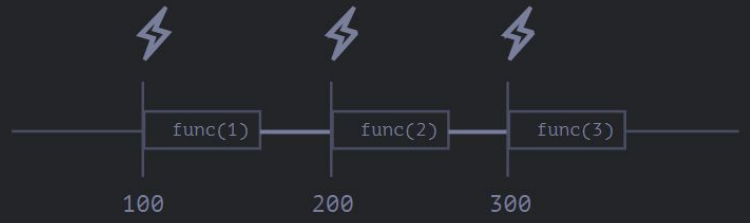
```
let delay = 5000;

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);
```

Вложенный setTimeout гарантирует фиксированную задержку

Сравним два фрагмента кода. Первый использует `setInterval`:

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

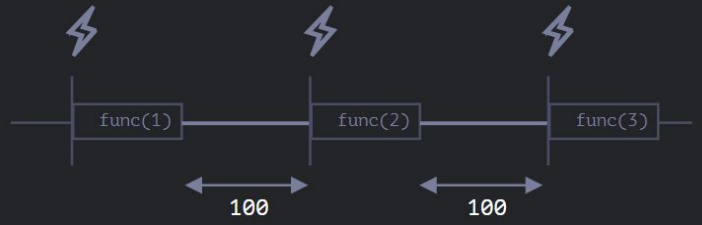
Для `setInterval` внутренний планировщик будет выполнять `func(i)` каждые 100 мс:



Второй использует вложенный `setTimeout`:

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

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



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.

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.

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

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.

Событийный цикл: микро и макро задачи

Поток выполнения в браузере, равно как и в Node.js, основан на событийном цикле.

Понимание работы событийного цикла важно для оптимизаций, иногда для правильной архитектуры.

Идея событийного цикла очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

Общий алгоритм движка:

- Пока есть задачи:
  - выполнить их, начиная с самой старой
- Бездействовать до появления новой задачи, а затем перейти к пункту 1

очередью макрозадач

Задачи поступают на выполнение – движок выполняет их – затем ожидает новые задачи (во время ожидания практически не нагружая процессор компьютера)

Может так случиться, что задача поступает, когда движок занят чем-то другим, тогда она ставится в очередь.

Очередь, которую формируют такие задачи, называют «очередью макрозадач»

Отметим две детали:

- Рендеринг (отрисовка страницы) никогда не происходит во время выполнения задачи движком. Не имеет значения, сколь долго выполняется задача. Изменения в DOM отрисовываются только после того, как задача выполнена.
- Если задача выполняется очень долго, то браузер не может выполнять другие задачи, обрабатывать пользовательские события, поэтому спустя некоторое время браузер предлагает «убить» долго выполняющуюся задачу. Такое возможно, когда в скрипте много сложных вычислений или ошибка, ведущая к бесконечному циклу.

Пример 1: разбиение «тяжелой» задачи.

Если вы запустите код ниже, движок «зависнет» на некоторое время. Для серверного JS это будет явно заметно, а если вы будете выполнять этот код в браузере, то попробуйте нажимать другие кнопки на странице – вы заметите, что никакие другие события не обрабатываются до завершения функции счета.

Браузер может даже показать сообщение «скрипт выполняется слишком долго».

Давайте разобьем задачу на части, воспользовавшись вложенным setTimeout.

```
let i = 0;

let start = Date.now();

function count() {

  // делаем тяжёлую работу
  for (let j = 0; j < 1e9; j++) {
    i++;
  }

  alert("Done in " + (Date.now() - start) + 'ms');
}

count();
```

Решение

Теперь интерфейс браузера полностью работоспособен во время выполнения «счёта».

Один вызов count делает часть работы (*), а затем, если необходимо, планирует свой очередной запуск (**):

- Первое выполнение производит счёт: i=1…1000000.
- Второе выполнение производит счёт: i=1000001…2000000.
- …и так далее.

Теперь если новая сторонняя задача (например, событие onclick) появляется, пока движок занят выполнением 1-й части, то она становится в очередь, и затем выполняется, когда 1-я часть завершена, перед следующей частью. Периодические возвраты в событийный цикл между запусками count дают движку достаточно «воздуха», чтобы сделать что-то ещё, отреагировать на действия пользователя.

Отметим, что оба варианта – с разбиением задачи с помощью setTimeout и без – сопоставимы по скорости выполнения. Нет большой разницы в общем времени счета.

```
let i = 0;

let start = Date.now();

function count() {

  // делаем часть тяжёлой работы (*)
  do {
    i++;
  } while (i % 1e6 != 0);

  if (i == 1e9) {
    alert("Done in " + (Date.now() - start) + 'ms');
  } else {
    setTimeout(count); // планируем новый вызов (**)
  }

}

count();
```

Оптимизация

Теперь, когда мы начинаем выполнять count() и видим, что потребуется выполнить count() ещё раз, мы планируем этот вызов немедленно, перед выполнением работы.

Если вы запустите этот код, то легко заметите, что он требует значительно меньше времени.

Почему?

Всё просто: как вы помните, в браузере есть минимальная задержка в 4 миллисекунды при множестве вложенных вызовов setTimeout. Даже если мы указываем задержку 0, на самом деле она будет равна 4 мс (или чуть больше). Поэтому чем раньше мы запланируем выполнение – тем быстрее выполнится код.

Итак, мы разбили ресурсоемкую задачу на части – теперь она не блокирует пользовательский интерфейс, причем почти без потерь в общем времени выполнения.

```
let i = 0;

let start = Date.now();

function count() {

  // перенесём планирование очередного вызова в начало
  if (i < 1e9 - 1e6) {
    setTimeout(count); // запланировать новый вызов
  }

  do {
    i++;
  } while (i % 1e6 != 0);

  if (i == 1e9) {
    alert("Done in " + (Date.now() - start) + 'ms');
  }

}

count();
```

Пример 2: индикация прогресса

Еще одно преимущество разделения на части крупной задачи в браузерных скриптах – это возможность показывать индикатор выполнения.

Обычно браузер отрисовывает содержимое страницы после того, как заканчивается выполнение текущего кода. Не имеет значения, насколько долго выполняется задача. Изменения в DOM отображаются только после ее завершения.

С одной стороны, это хорошо, потому что наша функция может создавать много элементов, добавлять их по одному в документ и изменять их стили – пользователь не увидит «промежуточного», незаконченного состояния.

изменения i не будут заметны, пока функция не завершится, поэтому мы увидим только последнее значение i:

```
<div id="progress"></div>

<script>

  function count() {
    for (let i = 0; i < 1e6; i++) {
      i++;
      progress.innerHTML = i;
    }
  }

  count();
</script>
```

Решение:

Но, возможно, мы хотим что-нибудь показать во время выполнения задачи, например, индикатор выполнения.

Если мы разобьем тяжёлую задачу на части, используя setTimeout, то изменения индикатора будут отрисованы в промежутках между частями.

Теперь показывает растущее значение i – это своего рода индикатор выполнения.

```
<div id="progress"></div>

<script>
  let i = 0;

  function count() {

    // сделать часть крупной задачи (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 != 0);

    if (i < 1e7) {
      setTimeout(count);
    }

  }

  count();
</script>
```

Макрозадачи и Микрозадачи

Помимо макрозадач, описанных в этой части, существуют микрозадачи.

Микрозадачи приходят только из кода. Обычно они создаются промисами: выполнение обработчика .then/catch/finally становится микрозадачей. Микрозадачи также используются «под капотом» await, т.к. это форма обработки промиса.

Также есть специальная функция queueMicrotask(func), которая помещает func в очередь микрозадач.

Сразу после каждой макрозадачи движок исполняет все задачи из очереди микрозадач перед тем, как выполнить следующую макрозадачу или отобразить изменения на странице, или сделать что-то ещё.

Event loop

- code появляется первым, т.к. это обычный синхронный вызов.
- promise появляется вторым, потому что .then проходит через очередь микрозадач и выполняется после текущего синхронного кода.
- timeout появляется последним, потому что это макрозадача.

Все микрозадачи завершаются до обработки каких-либо событий или рендеринга, или перехода к другой макрозадаче.

Сетевые запросы - Fetch

JavaScript может отправлять сетевые запросы на сервер и подгружать новую информацию по мере необходимости.

Есть несколько способов делать сетевые запросы и получать информацию с сервера.

Метод fetch() — современный и очень мощный, поэтому начнём с него. Он не поддерживается старыми (можно использовать полифил), но поддерживается всеми современными браузерами.

Базовый синтаксис:

let promise = fetch(url, [options])

- url – URL для отправки запроса.
- options – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url.

Браузер сразу же начинает запрос и возвращает промис, который внешний код использует для получения результата.

Процесс получения ответа обычно происходит в два этапа.

```
let response = await fetch(url);

if (response.ok) { // если HTTP-статус в диапазоне 200-299
  // получаем тело ответа (см. про этот метод ниже)
  let json = await response.json();
} else {
  alert("Ошибка HTTP: " + response.status);
}
```

Для сетевых запросов из JavaScript есть широко известный термин «AJAX» (аббревиатура от Asynchronous JavaScript And XML). XML мы использовать не обязаны, просто термин старый, поэтому в нём есть это слово.

Во-первых, promise выполняется с объектом встроенного класса Response в качестве результата, как только сервер пришлет заголовки ответа.

На этом этапе мы можем проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

Промис завершается с ошибкой, если fetch не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

Мы можем увидеть HTTP-статус в свойствах ответа:

status – код статуса HTTP-запроса, например 200.

ok – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.

Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода.

Response предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:

- response.text() – читает ответ и возвращает как обычный текст,
- response.json() – декодирует ответ в формате JSON,
- response.formData() – возвращает ответ как объект FormData (разберём его в следующей главе),
- response.blob() – возвращает объект как Blob (бинарные данные с типом),
- response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),
- помимо этого, response.body – это объект ReadableStream, с помощью которого можно считывать тело запроса по частям. Мы рассмотрим и такой пример несколько позже.

Важно:

Мы можем выбрать только один метод чтения ответа.

Если мы уже получили ответ с response.text(), тогда response.json() не сработает, так как данные уже были обработаны.

```
let text = await response.text(); // тело ответа обработано
let parsed = await response.json(); // ошибка (данные уже были обработаны)
```

Заголовки ответа

Заголовки ответа хранятся в похожем на Map объекте response.headers.

Это не совсем Map, но мы можем использовать такие же методы, как с Map, чтобы получить заголовок по его имени или перебрать заголовки в цикле:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

// получить один заголовок
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// перебрать все заголовки
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

Заголовки запроса

Для установки заголовка запроса в fetch мы можем использовать опцию headers. Она содержит объект с исходящими заголовками.

Эти заголовки обеспечивают достоверность данных и корректную работу протокола HTTP, поэтому они контролируются исключительно браузером.

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

POST-запросы

Для отправки POST-запроса или запроса с другим методом, нам необходимо использовать fetch параметры:

- method – HTTP метод, например POST,
- body – тело запроса, одно из списка:
  - строка (например, в формате JSON),
  - объект FormData для отправки данных как form/multipart,
  - Blob/BufferSource для отправки бинарных данных,
  - URLSearchParams для отправки данных в кодировке x-www-form-urlencoded, используется редко.

Чаще всего используется JSON.

```
let user = {
  name: 'John',
  surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Отправка изображения

Мы можем отправить бинарные данные при помощи fetch, используя объекты Blob или BufferSource.

В этом примере есть элемент <canvas>, на котором мы можем рисовать движением мыши. При нажатии на кнопку «Отправить» изображение отправляется на сервер:

Заметим, что здесь нам не нужно вручную устанавливать заголовок Content-Type, потому что объект Blob имеет встроенный тип (image/png, заданный в toBlob). При отправке объектов Blob он автоматически становится значением Content-Type.

Функция submit() может быть переписана без async/await, например, так:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

  <input type="button" value="Отправить" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
      let response = await fetch('/article/fetch/post/image', {
        method: 'POST',
        body: blob
      });

      // сервер ответит подтверждением и размером изображения
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```

```
function submit() {
  canvasElem.toBlob(function(blob) {
    fetch('/article/fetch/post/image', {
      method: 'POST',
      body: blob
    })
      .then(response => response.json())
      .then(result => alert(JSON.stringify(result, null, 2)))
  }, 'image/png');
}
```

Итого

Типичный запрос с помощью `fetch` состоит из двух операторов `await`:

```
let response = await fetch(url, options); // завершается с заголовками ответа
let result = await response.json(); // читать тело ответа в формате JSON
```

Или, без `await`:

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* обрабатываем результат */)
```

Параметры ответа:

- response.status – HTTP-код ответа,
- response.ok – true, если статус ответа в диапазоне 200-299.
- response.headers – похожий на Map объект с HTTP-заголовками.

Методы для получения тела ответа:

- response.text() – возвращает ответ как обычный текст,
- response.json() – декодирует ответ в формате JSON,
- response.formData() – возвращает ответ как объект FormData (кодировка form/multipart, см. следующую главу),
- response.blob() – возвращает объект как Blob (бинарные данные с типом),
- response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневые бинарные данные),

Опции fetch, которые мы изучили на данный момент:

- method – HTTP-метод,
- headers – объект с запрашиваемыми заголовками (не все заголовки разрешены),
- body – данные для отправки (тело запроса) в виде текста, FormData, BufferSource, Blob или UrlSearchParams.

Ресурсы

Планирование: setTimeout и setInterval - ТЫК

Событийный цикл: микрозадачи и макрозадачи - ТЫК

Сетевые запросы Fetch - ТЫК

JSONPlaceHolder (учебные API для имитации работы с сервером) - ТЫК