нетология

Асинхронность

Евгений
Шек

Евгений Шек

Frontend-разработчик

Aviata.kz / Chocotravel.kz

Евгений Шек

План занятия

1. Понятие асинхронности
2. setTimeout
3. setInterval
4. Работа с HTTP
5. Оптимизация вычислений
6. bind

Понятие асинхронности

Синхронное выполнение

До настоящего момента весь код, что вы писали, и что мы проходили на лекциях, был **синхронным**.

Принцип синхронного выполнения прост: код выполняется сразу.

Иногда для выполнения работы коду требуется значительное время, но суть та же: программа начинает выполняться **незамедлительно:**

```
const factorial = n => {
  for (var i = 1; n > 1; i *= n--);
  return i;
}
// 1 * 2 * 3 * ... * 50
factorial(50);
```

Асинхронное выполнение

При разработке ПО бывают и специфические **задачи**:

- **функция** выполняется очень долго, и хочется параллельно выполнять другой код. Например, пока ждём новых данных с сервера, можно работать со старыми
- **функция** должна выполняться в определённое время. Например, будильник или рассылка email-уведомлений
- неизвестно точно, когда **функция** выполнится. Например, экран телефона необходимо включить только тогда, когда пользователь нажмёт на кнопку Power

Асинхронное выполнение

Типичный случай из непрограммирования. В данном примере цвет кнопки изменится **только при наведении** на неё. Когда это может произойти и произойдёт ли вообще, зависит только от посетителя:

```
<button>Нажми на меня</button>
```

```
button:hover {
  background: red;
}
```

setTimeout

setTimeout

Для отложенного вызова функций в браузере есть функция *setTimeout(fn, ms)*, где *fn* — колбэк-функция, которую нужно запустить через *ms* миллисекунд *(1000 миллисекунд = 1 секунда)*. Например, этот код через 1 секунду добавит в консоль приветствие:

```
const showGreeting = () =>
  console.log(
    "Поздравляем, вы стали обладателем $1000! Но мы их вам не дадим!"
  );

setTimeout(showGreeting, 1000);
```

Неточность вызова

Тем не менее, *showGreeting* из прошлого примера **выполнится** не обязательно через 1 секунду.

Для наглядности 10 раз вызовем *setTimeout* с одним и тем же параметром:

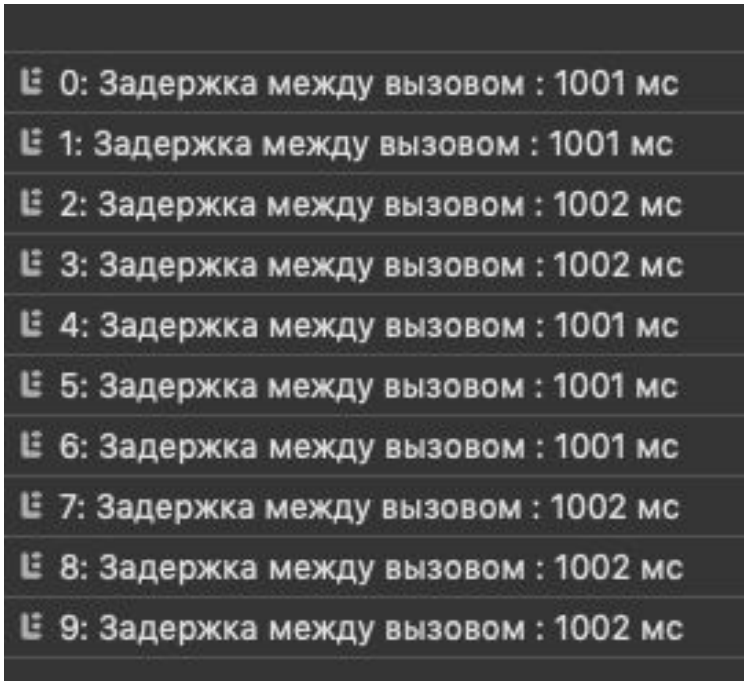
```
const checkDelay = (index, delay) => {
  const start = new Date();
  setTimeout(() => {
    const end = new Date();
    const delay = end - start;

    console.log(`${index}: Задержка между вызовом : ${delay} мс`);
  }, delay);
};

for (let i = 0; i < 10; i++) checkDelay(i, 1000);
```

Неточность вызова

Каждый раз **вывод** в консоль может немного отличаться. Вот пример вызова:



Один поток

Данная особенность связана с тем, что код JavaScript **выполняется в одном потоке**.

В нашем случае *showGreeting* выполнится, только если в настоящий момент не выполняется вообще ничего.

Если же параллельно с *showGreeting* выполняется какой-либо другой код, интерпретатор JavaScript **сначала закончит** выполнять его и только **потом возьмётся** за *showGreeting*

Один поток

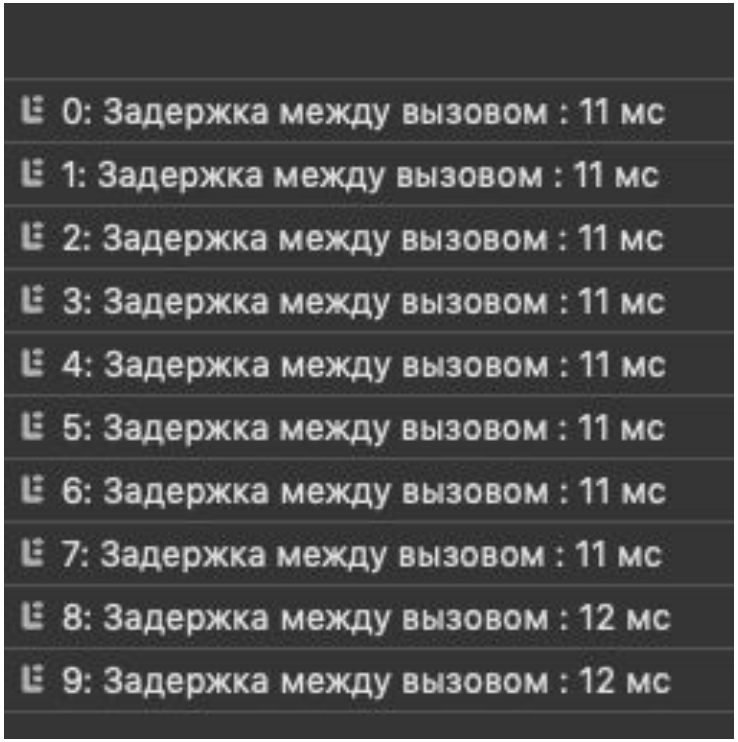
Для демонстрации нам необходимо выполнить два действия:

- **уменьшить задержку** для наглядности
- **нагрузить интерпретатор** долгими вычислениями:

```
for (let i = 0; i < 10; i++) {
  checkDelay(i, 10);
}
```

Один поток

Вот эталонный результат, который тоже может немного **отличаться** от вызова к вызову:



Один поток

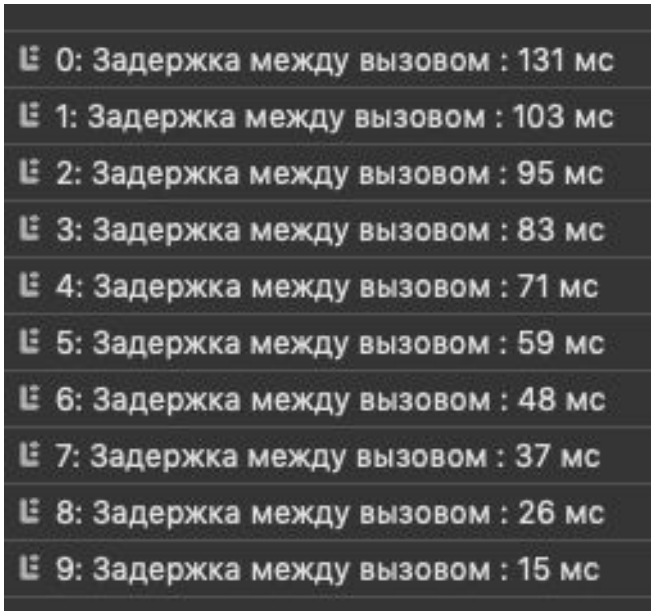
Теперь нагрузим JavaScript **бесполезной долгой работой**:

```
// делает много бесполезной работы. Главное, чтобы долго!
const idle = (n) => {
  let sum = 0;
  for (let i = 0; i < n; i++) {
    sum += i;
  }
};

for (let i = 0; i < 10; i++) {
  checkDelay(i, 10);
  idle(1000000);
}
```

Один поток

И результат существенно изменится:



Это значит, что **функция** в *setTimeout* будет вызвана, только если у неё будет такая возможность.

JavaScript не несёт никакой ответственности за то, что *setTimeout* сработает вовремя, но обещает это сделать как можно раньше в рамках **заданной задержки**

Минимальная задержка

Вы можете вызывать *setTimeout* и **без параметра** временной задержки. В таком случае функция обратного вызова сработает как можно скорее *и без каких-либо обязательств*.

Пример 1:

```
const showGreeting = () => console.log("Добрый день, я консольный бог!");

setTimeout(showGreeting);
```

Минимальная задержка

Пример 2:

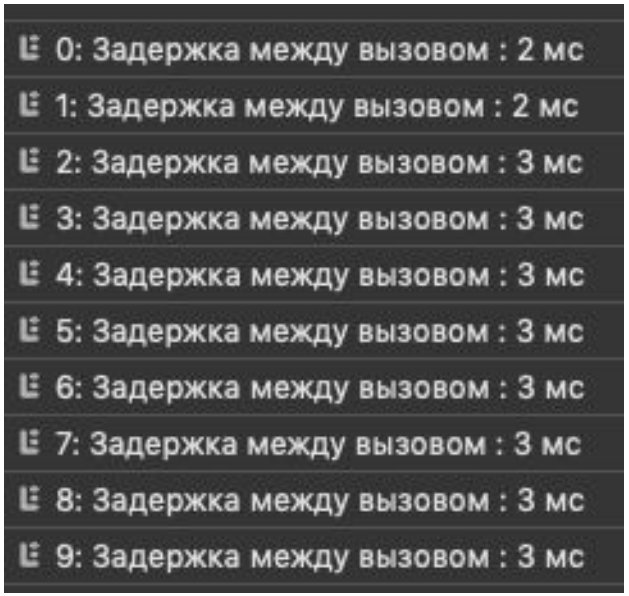
```
const checkDelay = (index, delay) => {
  const start = new Date();
  setTimeout(() => {
    const end = new Date();
    const delay = end - start;

    console.log(`${index}: Задержка между вызовом: ${delay} мс`);
  }, delay);
};

for (let i = 0; i < 10; i++) {
  // второй параметр в данном случае будет равен undefined
  checkDelay(i);
}
```

Минимальная задержка

Пример вызова:



Вызов без второго аргумента равносилен указанию значения 0.

У каждого браузера всё равно есть минимальный **порог**, меньше которого задержка не будет

Минимальная задержка

```
const showGreeting = () => console.log("Добрый день, я консольный бог!");

// можно было и не указывать 2 параметр
setTimeout(showGreeting, 0);
```

clearTimeout

Запланированную задачу можно отменить с помощью **функции** *clearTimeout*. Для этого необходимо передать идентификатор таймаута, возвращаемый от *setTimeout*:

```
// сообщение в консоль не будет выведено
const id = setTimeout(() => console.log("Я хочу жить :("), 500);

clearTimeout(id);
```

clearTimeout

Запланированную задачу можно отменить с помощью **функции** *clearTimeout*. Для этого необходимо передать идентификатор таймаута, возвращаемый от *setTimeout:*

```
// сообщение в консоль не будет выведено
const sendPayment = () => console.log("Вам начислена зарплата!");
const isCrisis = true;
const id = setTimeout(sendPayment, 1000);

// не выплачиваем сотрудникам деньги. У нас кризис!
if (isCrisis) {
  clearTimeout(id);
}
```

setInterval

setInterval: **демо**

Аналогично с *setTimeout*, *setInterval* запускает функцию обратного вызова.

Но делает это постоянно:

```
const areWeHome = () => console.log("Мы уже приехали?");
// будет запускать areWeHome до посинения
setInterval(areWeHome, 1000);
```

Если для удаления таймаута используется *clearTimeout*, то для **удаления интервала** используется *clearInterval*

setInterval: демо

Функция *clearInterval* принимает *id* интервала и удаляет его, *если он существует*

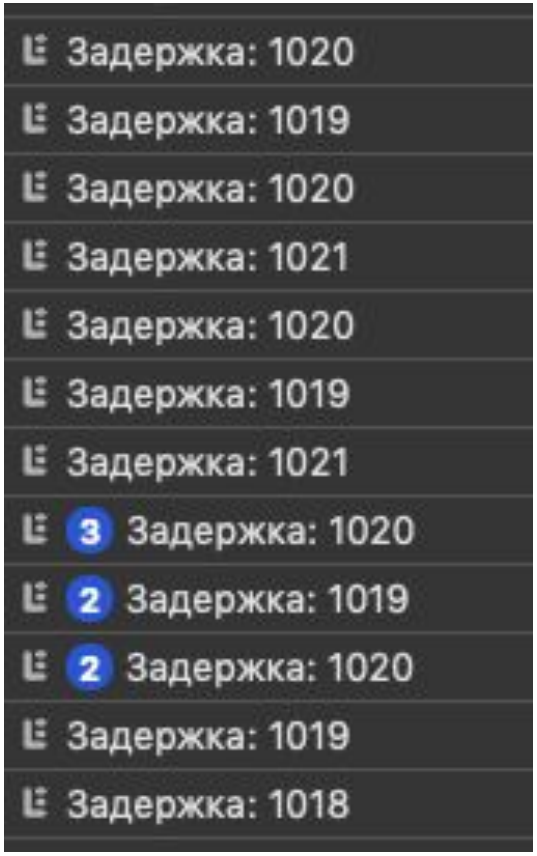
Болезнь запуска

Как и *setTimeout*, *setInterval* также болеет непредсказуемым временем запуска **функций обратного вызова:**

```
let start = new Date();
setInterval(() => {
  const end = new Date();
  const delay = end - start;
  console.log(`Задержка: ${delay}`);
  start = new Date();
}, 1000);
```

Болезнь запуска

Пример выполнения:



Передача аргументов

Все **аргументы**, которые передаются в *setTimeout* и *setInterval* после второго, становятся аргументами **callback-функции** в момент вызова:

```
// выведет сообщение «Блиц, скорость без границ!» через 1с.
setTimeout(console.log, 1000, "Блиц, скорость без границ!");

const sum = (a, b) => a + b;

// Выведет 29 через 1с.
setTimeout(sum, 1000, 10, 19);
```

Работа с HTTP

Работа с HTTP

Наиболее распространённый случай работы с **асинхронным кодом**

— запросы по HTTP: https://repl.it/repls/CourteousPoisedBrace

Рассмотрим этот код

```
// 1. Создаём запрос
var xhr = new XMLHttpRequest();

// 2. Определяем функцию обратного вызова
xhr.onreadystatechange = processResponse;

// Этот код выполнится, запрос кода будет в пути
function processResponse(e) {
  if (xhr.readyState === 4) {    // Запрос выполнился!
    console.log(xhr.responseText);
  } else {     // Запрос ещё выполняется
    console.log("Загружаем ...");
  }
}

// 3. Определяем, куда и как отправлять запрос
xhr.open("GET", "employees.json", true);

// 4. Отправляем запрос
xhr.send();

console.log("Другая важная работа ...");
```

Результаты работы – ответ в консоли:

```
Загружаем ...
Другая важная работа ...
Загружаем ...
Загружаем ...
[
  {
    "name": "Jim",
    "inOffice": false
  },
  {
    "name": "Joe",
    "inOffice": true
  },
  {
    "name": "John",
    "inOffice": true
  }
]
```

Оптимизация вычислений

Оптимизация вычислений

Проблема долгих вычислений. Второй вид функций выполняется очень долго. При больших значениях *n* страница будет существенно подвисать:

```
const sum = (n) => {
  let sum = n;
  for (let i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
};

sum(100000);
```

Оптимизация вычислений

Решение. **Интервалы** и **таймеры** — идеальное решение для долгих или рекурсивных вычислений. С помощью них мы можем проделывать полезную работу небольшими порциями, не нагружая страницу

Оптимизация вычислений

```
// вычисляем сумму от 0 до n, по itemsPerStep элементов за раз
const sumStep = (n, itemsPerStep, onload) => {
  const size = Math.ceil(n / itemsPerStep); // количество шагов
  let index = 0; // текущий шаг
  let sum = 0; // сумма вычислений

  // эта функция будет вызываться каждые 500 мс
  return () => {
    // окончание вычислений
    if (index === size) {
      onload(sum);
      return;
    }
...
```

Вторая часть кода на следующей странице

Оптимизация вычислений

```
…
    // начальные и конечные значения шага
    const start = index * itemsPerStep;
    const end = Math.min((index + 1) * itemsPerStep, n + 1);

    // сами вычисления
    for (let i = start; i < end; i++) {
      sum += i;
    }

    console.log(`Шаг ${index}: ${sum}`);
    index++;
    // планируем новый шаг
    scheduleStep();
  };
};
```

Третья часть кода на следующей странице

Оптимизация вычислений

```
...
// функция обратного вызова для вывода результата
const onload = (result) => console.log(`Результат вычислений: ${result}`);

// задаем начальные настройки
const step = sumStep(1000000, 1000, onload);
const scheduleStep = () => setTimeout(step, 500);

scheduleStep();
```

Тайм-ауты в интерфейсах

Использование **тайм-аутов** и **интервалов** в JavaScript — довольно распространённое явление. На сайтах вы наверняка встречались со следующими примерами:

- появление всплывающих **окон** в заданное время
- **таймеры** обратного отсчёта на странице*: «До конца акции осталось...»*
- показ **подсказок** при поиске только по окончании ввода в текстовое поле
- простая **анимация**
- автоматическая **смена слайдов** в фотогалерее
- **проверка** в простых чатах новых сообщений

Вызов *setTimeout* для метода объекта

Представим, что у нас есть **объект,** и мы хотим вызвать его **метод** через определённое время:

```
const user = {
  firstName: "Antonio",
  sayName() {
    console.log("My name is " + this.firstName);
  },
};
user.sayName(); // работает
setTimeout(user.sayName, 500); // "My name is undefined"
```

Почему так? Рассмотрим другой пример

Вызов *setTimeout* для метода объекта

```
let func = user.sayName;
setTimeout(func, 500); // "My name is undefined"
func(); // "My name is undefined"
```

Причина в том, что контекст **определяется во время вызова**, *а не объявления,* по правилу «объект перед точкой». *func,* равная *user.sayName,* вызывается с *this = window*.

Таким образом, для *this.firstName* пытаемся получить *window.firstName*, которого не существует

Как исправить

Существует несколько способов исправить данную ситуацию. С одним из них вы уже знакомы из лекции «Функции-декораторы». Это *func.call*:

```
//1
setTimeout(function () {
  func.call(user); // установить this = user при вызове
}, 500);

//2
setTimeout(function () {
  user.sayName();
}, 500);

//3 (наиболее распространенный способ)
setTimeout(
  () => user.sayName(), // аналогично предыдущему, стрелочная функция
  500
);
```

Bind

Bind

Ещё одним выходом является **привязка функции** к контексту с помощью *let newFunc = oldFunc.bind(newThis)*:

```
user.sayNameBinded = user.sayName.bind(user);
// или даже так, так как старая функция нам больше не нужна
user.sayName = user.sayName.bind(user);
// или так
let func = user.sayName.bind(user);
```

Bind

Bind из старой функции *user.sayName* создаёт новую с привязанным контекстом:

```
setTimeout(user.sayName); // "My name is Antonio"
setTimeout(func); // "My name is Antonio"
```

Вызов *setTimeout* в методе объекта

```
const user = {
  firstName: "Antonio",
  sayName() {
    setTimeout(function () {
      console.log("My name is " + this.firstName);
    });
  },
};
user.sayName(); // "My name is undefined
```

Причина та же. Внутри *setTimeout* *this = window*

Как исправить?

```
const user = {
  firstName: "Antonio",
  sayName() {
    setTimeout(() => {
      console.log("My name is " + this.firstName);
    });
  },
};
user.sayName(); // "My name is Antonio
```

Так как стрелочная функция не имеет собственного *this*, то *this* будет взято из **замыкания** — *лексического окружения*

Как исправить?

Этот код аналогичен следующему:

```
const user = {
  firstName: "Antonio",
  sayName() {
    let savedThis = this;
    setTimeout(function () {
      console.log("My name is " + savedThis.firstName);
    });
  },
};
user.sayName(); // "My name is Antonio
```

Итоги

Чему мы научились?

- **Разобрались** с основами таймеров и интервалов
- **Научились** работать с датами
- **Узнали** о болезни запуска *setTimeout* и *setInterval*
- **Научились** оптимизировать долгие и рекурсивные вычисления
- **Узнали** разницу между синхронной и асинхронной работой
- **Разобрались,** как отправлять HTTP-запросы и обрабатывать результаты
- **Познакомились** с новым способом привязки контекста *bind*

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Евгений Шек

Евгений Шек