hillel IT school

Lesson 21

План заняття

- Планування виконання коду;
- Event loop;
- Callbacks;

setTimeout та setInterval

Ми можемо вирішити виконати функцію не зараз, а через певний час пізніше. Це називається “планування виклику”.

Для цього існує два методи:

1. setTimeout дозволяє нам запускати функцію один раз через певний інтервал часу.
2. setInterval дозволяє нам запускати функцію багаторазово, починаючи через певний інтервал часу, а потім постійно повторюючи у цьому інтервалі.

Ці методи не є частиною специфікації JavaScript. Але більшість середовищ виконання JS-коду мають внутрішній планувальник і надають ці методи. Зокрема, вони підтримуються у всіх браузерах та Node.js.

Специфікація

setTimeout

Синтаксис:

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

Аргументи:

- func|code - Функція або рядок коду для виконання.
- delay - Затримка перед запуском, у мілісекундах (1000 мс = 1 секунда), типове значення – 0. Але 0 === 4ms
- arg1, arg2 - Аргументи, які передаються у функцію
- timerId - ідентифікатор таймера

clearTimeout

clearTimeout(timerId) - скасує таймер.

setInterval

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

Цей метод запускає функцію не один раз, але регулярно через заданий проміжок часу.

Щоб припинити подальші виклики, нам слід викликати clearInterval (timerId).

Вкладений setTimeout

Існує два способи запускати щось регулярно.

Один із них – setInterval. Інший – це вкладений setTimeout

Вкладений setTimeout є більш гнучким методом, ніж setInterval. Таким чином, наступний виклик може бути запланований по-різному, залежно від результатів поточного.

Вкладений setTimeout дозволяє більш точно встановити затримку між виконанням, ніж setInterval.

setTimeout з нульовою затримкою

Існує особливий варіант використання: setTimeout(func, 0) або просто setTimeout(func).

Це планує виконання func якнайшвидше. Але планувальник викликає його лише після завершення виконання поточного скрипту. Таким чином, функція планується до запуску “відразу після” поточного скрипту.

У браузері є обмеження щодо того, як часто можуть запускатися вкладені таймери. Згідно зі HTML Living Standard: “після п’яти вкладених таймерів інтервал змушений бути не менше 4 мілісекунд.”.

Цикл подій (event loop)

Потік виконання JavaScript в браузері, так само як і в Node.js, базується на циклі подій (event loop).

Концепція циклу подій дуже проста. Існує нескінченний цикл, в якому рушій JavaScript очікує завдання, виконує їх, а потім переходить в режим очікування нових завдань.

Загальний алгоритм рушія:

- Поки є завдання виконати їх, починаючи з найстарішого.
- Очікувати поки завдання не з'явиться, потім перейти до пункту 1.

Рушій JavaScript більшість часу не робить нічого, він працює лише коли спрацьовує скрипт, обробник подій чи подія.

З'являються задачі для виконання – рушій виконує їх – потім очікує нових завдань (майже не навантажуючи процесор в режимі очікування).

Може трапитись так, що завдання приходить тоді, коли рушій вже зайнятий, тоді це завдання стає в чергу.

Чергу з таких завдань називають "чергою макрозавдань" ("macrotask queue", термін v8)

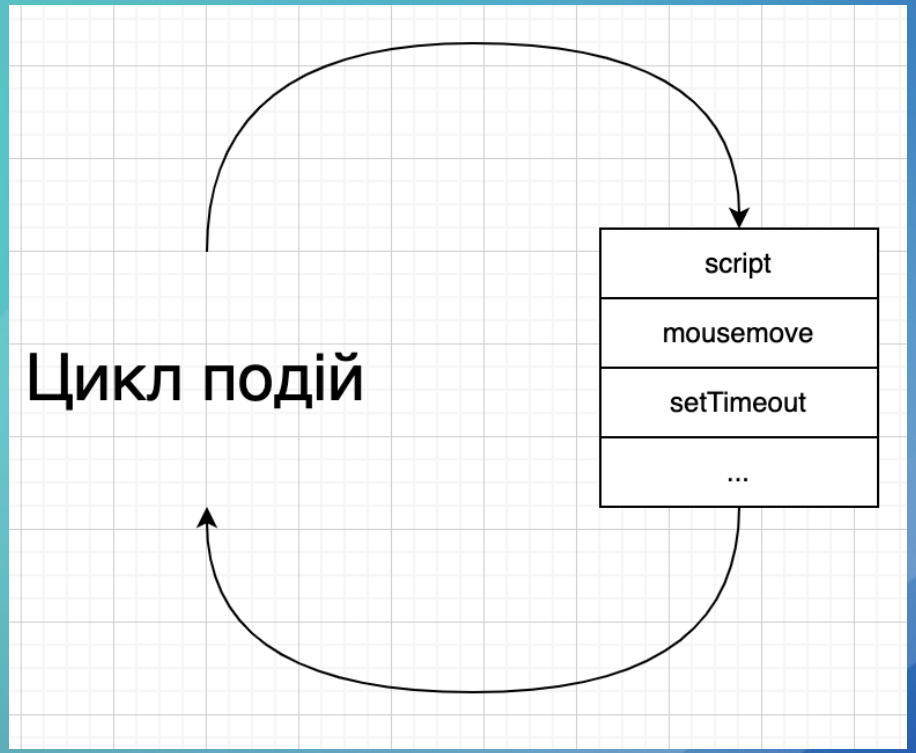
macrotask

Макротаски створюють:

- setTimeout, setInterval, setImidiat.
- All Events. Mouse events, ui events, script, завантаження зображень.
- Браузерні подіїБ наприклад рендер і т.д.

Наприклад, поки рушій виконує script, користувач може порухати мишкою, що спричинить появу події mousemove, та може вийти час, запрограмований в setTimeout і так далі. Ці завдання сформують чергу.

Задачі з черги виконуються за правилом “перший прийшов – перший пішов”. Коли рушій браузера закінчить виконання script, він обробить подію mousemove, потім виконає обробник setTimeout тощо.



macrotask

Запам’ятайте:

1. Рендеринг ніколи не відбувається поки рушій виконує завдання. Не має значення наскільки довго виконується завдання. Зміни в DOM будуть відмальовані лише після завершення завдання.
2. Якщо виконання завдання займає надто багато часу, браузер не зможе виконувати інші завдання, наприклад, обробляти користувацькі події. Тож після недовгого часу “зависання” з’явиться оповіщення “Сторінка не відповідає” і пропозиція вбити процес виконання завдання разом з цілою сторінкою. Таке трапляється коли код містить багато складних обрахунків або виникає програмна помилка, що створює нескінченний цикл.

microtask

Макротаски створюють:

- Проміси. .then/catch/finally
- Використовуються “під капотом” у await, так як це форма обробки проміса.
- queueMicrotask(func) - ставить func в чергу мікрозавдань. Використовуємо якщо хочемо виконати функцію асинхронно (після поточного коду), але до відображення змін чи обробки нових подій
- MutationObserver

Одразу після кожного макрозавдання, рушій виконує всі завдання з черги мікрозавдань перед тим як виконати якесь макрозавдання чи рендеринг чи виконати щось іще.

Всі мікрозавдання завершуються до обробки будь-яких подій чи рендерингу чи виконання інших макрозавдань.

Це важливо, тому що це гарантує, що середовище застосунку залишається незмінним між мікрозадачами (не змінились координати мишки, не з’явились нові дані через мережу тощо).

Обробка асинхронного коду

Багато функцій надаються середовищами JavaScript, які дозволяють планувати асинхронні дії. Тобто дії, які ми починаємо зараз, але закінчуємо пізніше.

Наприклад:

- Функція setTimeout.
- Завантаження скриптів і модулів.
- Отримання даних з сервера.
- тощо...

колбеки

Ідея така:

1. перший аргумент - це ресурс або тип
2. другий аргумент – у функції це функція (зазвичай анонімна), яка запускається після завершення дії.

Обробка помилок

Домовленість така:

1. Перший аргумент callback зарезервовано для помилки, якщо вона виникає. В такому випадку викликається callback(err).
2. Другий аргумент (і наступні, якщо потрібно) – для успішного результату. В такому випадку викликається callback(null, result1, result2…).

Таким чином, єдина callback-функція використовується як для повідомлення про помилки, так і для повернення результатів.

Дякую за увагу