JavaScript - асинхронность

Асинхронность
event-loop (событийный цикл)
Микро-задачи (micro Tasks)
Макро-задачи (macro Tasks)
Многопоточность
Web Worker
Service Worker

Асинхронность

Асинхронность — это концепция, которая позволяет выполнять несколько задач одновременно, не блокируя основной процесс или поток выполнения. Это особенно полезно для задач, которые могут занять неопределенное время, например, сетевых запросов или операций ввода/вывода

Принципы асинхронности:

- Не блокирующее выполнение: Асинхронные операции позволяют программе продолжать выполнение других задач, не дожидаясь завершения текущей задачи. Это предотвращает "замораживание" программы, когда выполняются длительные или ресурсоемкие операции.
- Обратные вызовы (Callbacks): В асинхронном программировании часто используются обратные вызовы (callbacks), которые представляют собой функции, вызываемые по завершении асинхронной операции. Это позволяет продолжить выполнение программы, когда операция завершится.
- Промисы (Promises): Промисы представляют собой более структурированный способ обработки асинхронных операций. Промис хранит результат асинхронной операции и позволяет обрабатывать успешное завершение или ошибку.
- Async/Await: Async/await — это современный синтаксис для работы с промисами, который упрощает написание асинхронного кода, делая его похожим на синхронный. Это облегчает чтение и отладку кода.

Виды асинхронности

Асинхронность может реализовываться различными способами, и основные типы включают отложенные вызовы функций и многопоточность.

Отложенные вызовы функций

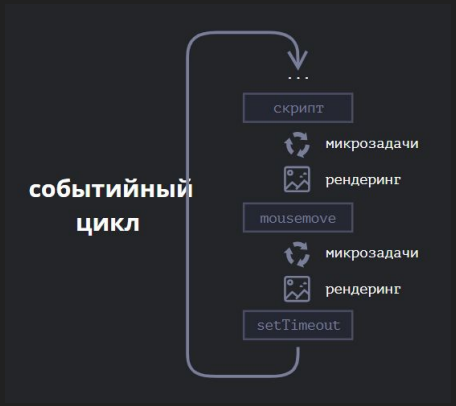
Отложенные вызовы (или задержки вызовов) подразумевают выполнение функции в будущем, через определенный интервал времени. Это не совсем то же самое, что многопоточность, но важно понимать, как это работает в контексте асинхронного программирования.

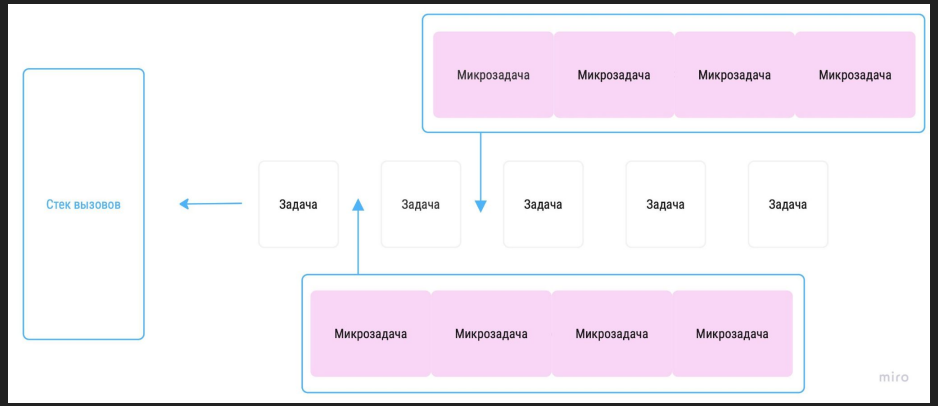
Многопоточность

Многопоточность позволяет выполнять несколько потоков выполнения параллельно, что может улучшить производительность, особенно для задач, требующих значительных ресурсов или времени.

Event-loop (событийный цикл)

Event Loop в JavaScript - это механизм, который позволяет JavaScript выполнять асинхронный код и управлять событиями. Он играет ключевую роль в не-блокирующей модели выполнения JavaScript, которая позволяет программе оставаться отзывчивой и эффективной даже при выполнении долгих операций.





Heap memory (куча)

Heap memory (куча) — это область памяти, используемая для динамического распределения памяти в программировании. В JavaScript, когда создаются объекты, массивы или функции, память для них выделяется в куче. Эта память не освобождается автоматически, и за её управление отвечает сборщик мусора (garbage collector).

Когда вы создаете объекты, массивы или функции, память для них выделяется в куче:

В этом примере person, numbers и sayHello хранятся в куче.

```
// Создание объекта
let person = {
  name: "John",
  age: 30
};

// Создание массива
let numbers = [1, 2, 3, 4, 5];

// Создание функции
function sayHello() {
  console.log("Hello, world!");
}
```

Основные компоненты Event Loop:

Call Stack: Это структура данных, которая управляет функциями, которые нужно выполнить. Когда вызывается функция, она добавляется в стек вызовов. Когда функция завершается, она удаляется из стека.

Web APIs: Это браузерные API, такие как setTimeout, DOM events, fetch, и другие, которые позволяют выполнять асинхронные операции. Когда эти API выполняют свои задачи, они передают результаты в соответствующие очереди.

Callback Queue (Task Queue): Это очередь, в которую попадают обратные вызовы (callbacks), когда асинхронные операции завершены. Callbacks ждут своей очереди на выполнение.

Microtask Queue: Это отдельная очередь, в которой хранятся микрозадачи (promises, MutationObserver). Она имеет приоритет перед Callback Queue и выполняется после текущего стека вызовов, но до выполнения любых задач из Callback Queue.

Как работает Event Loop:

Инициализация: JavaScript начинает с выполнения глобального кода, что означает, что он добавляет функции в Call Stack.

Выполнение кода: Функции в Call Stack выполняются синхронно, по одной за раз.

Асинхронные задачи: Когда вызываются асинхронные функции (например, setTimeout), они делегируются Web APIs, которые затем добавляют соответствующие callback-функции в Callback Queue после завершения своих операций.

Event Loop: После того, как Call Stack пуст, Event Loop начинает проверять очереди задач. Сначала он проверяет Microtask Queue и выполняет все микрозадачи. Затем он переходит к Callback Queue и выполняет одну задачу из этой очереди.

Повторение: Этот процесс повторяется бесконечно, что позволяет JavaScript выполнять асинхронный код и оставаться отзывчивым.

Особенности Event Loop:

Синхронность и асинхронность: Event Loop позволяет JavaScript выполнять код асинхронно, не блокируя выполнение других задач.

Микрозадачи и макрозадачи: Микрозадачи (promises) имеют приоритет перед макрозадачами (callback-ами из setTimeout, setInterval, I/O операции). Это означает, что все микрозадачи будут выполнены перед любой макрозадачей.

Однопоточность: JavaScript однопоточен, что означает, что в любой момент времени может выполняться только одна операция. Event Loop управляет этим, обеспечивая выполнение асинхронных задач в правильном порядке.

Пример

Порядок выполнения:

1. **Start:** Выполняется синхронный код и выводится в консоль.
2. **End:** Выполняется оставшийся синхронный код и выводится в консоль.
3. **Promise callback:** Микрозадача выполняется перед макрозадачей.
4. **Timeout callback:** Выполняется callback из setTimeout.

```
console.log('Start');

setTimeout(() => {
  console.log('Timeout callback');
}, 0);

Promise.resolve().then(() => {
  console.log('Promise callback');
});

console.log('End');
```

Макро задачи (Macro Tasks)

Макро задачи представляют собой крупные задачи, которые включают в себя, например, обработку событий, таймеров и сетевых запросов. Они выполняются последовательно в основном потоке, и каждая макро задача выполняется полностью перед началом следующей.

Особенности макро задач:

- Выполняются последовательно.
- Каждая макро-задача блокирует выполнение следующих макро-задач до тех пор, пока она не завершится.

Порождение макро-задачи

setTimeout:

```
setTimeout(() => {
    console.log('Macrotask 1');
}, 0);
```

setInterval:

```
setInterval(() => {
    console.log('Macrotask 2');
}, 1000);
```

Обработчики событий:

```
document.addEventListener('click', () => {
    console.log('Event handler');
});
```

Микро задачи (Micro Tasks)

Микро задачи выполняются после завершения текущего макро-задачи, но перед началом следующей макро-задачи. Они позволяют более детально управлять выполнением кода и могут быть использованы для обработки результатов промисов и других асинхронных операций.

Особенности микро задач:

- Выполняются сразу после завершения текущей макро-задачи.
- Микро задачи могут быть добавлены в очередь во время выполнения других микро задач.

Порождение микро-задачи

Промисы (Promise) - Промисы позволяют добавлять задачи в очередь микро задач. Коллбэки .then, .catch и .finally выполняются как микро задачи.

```
Promise.resolve().then(() => {
    console.log('Executed as a micro task');
});
```

MutationObserver - позволяет наблюдать за изменениями в DOM и выполнять функции как микро задачи.

```
const observer = new MutationObserver(() => {
    console.log('MutationObserver executed as a micro task');
});

observer.observe(document.body, { attributes: true });
```

queueMicrotask - Эта функция позволяет явно добавить задачу в очередь микро задач. Это эквивалентно Promise.resolve().then(...), но более прямолинейно.

```
queueMicrotask(() => {
    console.log('Executed as a micro task using queueMicrotask');
});
```

MutationObserver: наблюдатель за изменениями

MutationObserver - это встроенный объект в JavaScript, который позволяет наблюдать за изменениями в DOM-дереве. Это может быть полезно для отслеживания изменений, добавления или удаления элементов, изменения атрибутов и т.д.

Зачем использовать MutationObserver?

Обнаружение изменений в DOM: Автоматически реагировать на изменения в DOM, например, когда пользователь добавляет или удаляет элементы.

Отслеживание динамического контента: Веб-приложения часто обновляют контент динамически, и MutationObserver помогает управлять этими изменениями.

Оптимизация производительности: Вместо того, чтобы постоянно проверять DOM на изменения, MutationObserver позволяет реагировать только тогда, когда изменения действительно происходят.

Как пользоваться MutationObserver?

Создание MutationObserver: Для начала нужно создать объект MutationObserver, передав в него коллбэк-функцию, которая будет вызываться при обнаружении изменений.

Настройка наблюдателя: Нужно настроить, за какими изменениями наблюдать (например, за изменениями атрибутов, добавлением или удалением элементов и т.д.).

Подключение наблюдателя к элементу: Запустить наблюдателя для конкретного элемента.

Отключение наблюдателя: Когда больше не нужно наблюдать за изменениями, наблюдателя можно отключить.

методы и свойства

MutationObserver.observe(target, options): Начинает наблюдение за изменениями на элементе target с заданными опциями options.

MutationObserver.disconnect(): Останавливает наблюдение за изменениями.

MutationObserver.takeRecords(): Возвращает список всех обнаруженных изменений.

Опции наблюдения

- attributes: Наблюдение за изменениями атрибутов.
- childList: Наблюдение за добавлением или удалением дочерних элементов.
- subtree: Наблюдение за изменениями не только в указанном элементе, но и во всех его дочерних элементах.
- attributeOldValue: Запоминает старое значение атрибута до его изменения.
- characterData: Наблюдение за изменениями текстового содержимого элемента.
- characterDataOldValue: Запоминает старое значение текстового содержимого до его изменения.

Пример

```
// Шаг 1: Создаем MutationObserver и передаем коллбэк-функцию
const observer = new MutationObserver((mutationsList, observer) => {
  for (let mutation of mutationsList) {
    if (mutation.type === 'childList') {
      console.log('Добавлены или удалены дочерние элементы.');
    } else if (mutation.type === 'attributes') {
      console.log(`Изменен атрибут: ${mutation.attributeName}`);
    }
  }
});

// Шаг 2: Определяем настройки наблюдателя
const config = {
  attributes: true,    // Наблюдение за изменениями атрибутов
  childList: true,     // Наблюдение за добавлением или удалением дочерних элементов
  subtree: true        // Наблюдение за всеми дочерними элементами, а не только непосредственными
};

// Шаг 3: Указываем элемент для наблюдения
const targetNode = document.getElementById('target');
observer.observe(targetNode, config);

// Шаг 4: Пример добавления нового элемента
const newElement = document.createElement('div');
newElement.textContent = 'Новый элемент';
targetNode.appendChild(newElement);

// Шаг 5: Отключение наблюдателя при необходимости
observer.disconnect();
```

Многопоточность

Многопоточность — это способность программного обеспечения выполнять несколько потоков (threads) одновременно. Поток — это наименьшая единица обработки, которая может быть запланирована операционной системой. Основные преимущества многопоточности включают параллельное выполнение задач, улучшение производительности и более эффективное использование ресурсов.

Многопоточность в JavaScript

JavaScript изначально был разработан как однопоточный язык, что означает, что он выполняет только один поток кода в любой момент времени. Однако, с помощью некоторых механизмов и API, JavaScript может эффективно управлять асинхронными задачами и выполнять параллельные вычисления.

Основные механизмы для реализации многопоточности в JavaScript:

- Web Workers
- Service Workers
- Worker Threads (в Node.js)

Web Workers

Web Workers — это технология в JavaScript, которая позволяет выполнять скрипты в фоновом потоке, параллельно основному потоку (главному потоку). Это полезно для выполнения ресурсоемких операций без блокировки основного потока, что позволяет сохранять отзывчивость интерфейса.

Создание и работа

Web Worker создается через конструктор Worker, принимающий URL скрипта, который будет выполнен в фоновом потоке.

```
// Создание нового web worker
const worker = new Worker('worker.js');
```

Передача данных

Взаимодействие с Web Worker осуществляется через механизм обмена сообщениями. Основной поток и worker могут отправлять друг другу сообщения с помощью метода postMessage и прослушивать события message.

```
// В основном потоке
worker.postMessage('Hello, worker!');

worker.onmessage = function(event) {
    console.log('Сообщение от worker: ', event.data);
};

// В worker.js
self.onmessage = function(event) {
    console.log('Сообщение от основного потока: ', event.data);
    self.postMessage('Hello, main thread!');
};
```

Ограничения

Web Workers не имеют доступа к DOM, что означает, что они не могут напрямую манипулировать элементами на странице. Однако, они могут выполнять задачи, такие как обработка данных, математические вычисления, работа с таймерами и пр.

Завершение работы

Worker может быть остановлен из основного потока с помощью метода terminate.

```
worker.terminate();
```

Работа с файлами и библиотеками

Web Workers могут импортировать скрипты и работать с файлами с помощью метода importScripts.

```
// Внутри worker.js
importScripts('script1.js', 'script2.js');
```

Shared Workers

Существуют также Shared Workers, которые могут быть использованы несколькими скриптами (из разных вкладок или фреймов) одновременно.

```
// Создание Shared Worker
const sharedWorker = new SharedWorker('sharedWorker.js');

// Обмен сообщениями
sharedWorker.port.postMessage('Hello, shared worker!');
sharedWorker.port.onmessage = function(event) {
    console.log('Сообщение от shared worker: ', event.data);
};
```

Пример

Для решения задачи с подсчетом чисел Фибоначчи на Web Worker, необходимо создать основной скрипт, который будет взаимодействовать с worker, и сам worker, который будет выполнять вычисления в фоновом потоке.

Основной скрипт (main.js)

Этот скрипт создает Web Worker, отправляет ему сообщение с индексом числа Фибоначчи и получает результат.

```
// Создание нового Web Worker
const worker = new Worker('worker.js');

// Обработчик получения сообщения от worker
worker.onmessage = function(event) {
    console.log('Результат:', event.data);
};

// Отправка сообщения worker с индексом числа Фибоначчи
const index = 40; // Индекс числа Фибоначчи для вычисления
worker.postMessage(index);
```

Скрипт Web Worker (worker.js)

Этот скрипт выполняет вычисление числа Фибоначчи для заданного индекса и отправляет результат обратно в основной поток.

```
// Функция для вычисления числа Фибоначчи
function fibonacci(n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Обработчик получения сообщения от основного потока
self.onmessage = function(event) {
    const index = event.data;
    const result = fibonacci(index);
    self.postMessage(result);
};
```

Вывод

При запуске этого примера основной поток отправит Web Worker индекс числа Фибоначчи для вычисления, worker выполнит вычисление в фоновом режиме и отправит результат обратно в основной поток, который выведет его в консоль.

Service Worker

serviceWorker в JavaScript — это скрипт, который браузер запускает в фоновом режиме, отдельно от веб-страницы. Он позволяет осуществлять контроль над поведением сети, кэширования и получения пуш-уведомлений, обеспечивая тем самым возможность создания оффлайн-приложений и улучшая производительность и пользовательский опыт.

Функции и преимущества serviceWorker:

- Кэширование и оффлайн-доступ: serviceWorker позволяет кэшировать ресурсы (HTML, CSS, JavaScript, изображения и т.д.), чтобы пользователь мог взаимодействовать с веб-приложением даже без интернет-соединения.
- Управление сетевыми запросами: serviceWorker может перехватывать и управлять сетевыми запросами, решая, когда доставлять ресурсы из кэша, а когда обращаться к сети.
- Фоновая синхронизация: serviceWorker может выполнять фоновую синхронизацию данных, обеспечивая обновление данных приложения, когда устройство пользователя снова подключится к интернету.
- Пуш-уведомления: serviceWorker позволяет отправлять пользователям пуш-уведомления, даже если веб-приложение не активно.

Основные этапы работы с serviceWorker

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js').then(registration => {
      console.log('ServiceWorker зарегистрирован: ', registration);
    }).catch(error => {
      console.log('Регистрация ServiceWorker не удалась: ', error);
    });
  });
}
```

```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => {
        return response || fetch(event.request);
      })
  );
});
```

```
self.addEventListener('install', event => {
  // Логика установки, например, кэширование ресурсов
  event.waitUntil(
    caches.open('my-cache')
      .then(cache => cache.addAll([
        '/',
        '/index.html',
        '/styles.css',
        '/script.js',
        '/image.png'
      ]))
  );
});

self.addEventListener('activate', event => {
  // Логика активации, например, удаление старых кэшей
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.filter(cacheName => {
          return cacheName !== 'my-cache';
        }).map(cacheName => {
          return caches.delete(cacheName);
        })
      );
    })
  );
});
```

Ресурсы

Микро-задачи (micro-tasks) - ТЫК

Событийный цикл (event-loop) - ТЫК

MutationObserver: наблюдатель за изменениями - ТЫК

Пример web-worker-а - ТЫК

Пример service-worker-а - ТЫК

Свой event-loop на JavaScript - ТЫК

Визуализация event-loop-а - ТЫК