Основы веб-технологий

Асинхронный JavaScript

Асинхронный JavaScript

По умолчанию JavaScript является синхронным однопоточным языком программирования.

Синхронный код выполняется последовательно, каждая инструкция ожидает пока выполнится предыдущая. Асинхронный код выполняется не дожидаясь выполнения предыдущих инструкций.

Асинхронное программирование — это паттерн программирования, который позволяет множеству задач обрабатываться параллельно без блокировки друг-друга или основного потока. В языке JavaScript это чаще всего достигается применением обратных вызовов(callbacks), обещаний(promises) и ключевых слов async/await.

Синхронный код и его проблемы

Чтобы выполнить код, нам нужен JavaScript Engine (движок) — программа, которая «читает и выполняет» то, что мы написали. Самый распространённый движок среди всех — это V8, он используется в Google Chrome и Node.js.
Выполнение JS-кода — однопоточное. Это значит, что в конкретный момент времени движок может выполнять не более одной строки кода. То есть вторая строка не будет выполнена, пока не выполнится первая.
Такое выполнение кода (строка за строкой) называется синхронным.

Пример проблемного кода:

```
console.log("start");
function hello() {
    // действия, выполнение которых занимает значительное время
    // ....
    console.log("hello");
}

hello();
console.log("end");
```

Если мы запустили синхронную долгую функцию, то движок бы ничем другим заниматься в это время не мог. Выполнение синхронного кода — строка за строкой. То есть, пока долгая функция не выполнится до конца, к следующей строке интерпретатор не перейдёт. А это значит, что пока не пройдёт время и функция не выполнится, мы вообще ничего сделать не сможем: ни вывести что-то в консоль ещё, ни выполнить другие функции. Такие операции, которые не дают выполнять ничего кроме них самих, пока они не завершатся, называются **блокирующими выполнение.**

Асинхронный код

Теперь попробуем решить задачу, но так, чтобы наш код не блокировал выполнение. Для этого мы воспользуемся функцией таймером setTimeout() с нулевой задержкой:

```
console.log("start");
setTimeout(function hello() {
    // действия, выполнение которых занимает значительное время
    // ....
    console.log("hello");
}, 0)
console.log("end");
```

«hello» будет выведено в консоль с задержкой, однако в это время мы можем выполнять другие действия, даже несмотря на то, что задержка таймера нулевая.

Таймеры

Таймеры в JavaScript позволяют запустить код через нужное время. Чтобы запустить таймер, вам нужно указать время задержки. Время задержки отсчитывается в миллисекундах. Далее нужно указать код, который следует запустить после указанного времени. Таймер начинает работу сразу же после его настройки.

После завершения отсчета таймер запускает код JavaScript и исчезает. Можно создать многоразовый таймер, который будет запускать код через определенные промежутки времени, пока вы его не остановите. Также можно использовать одноразовый таймер.

Одноразовые таймеры запускаются при помощи метода `setTimeout()`.

```
setTimeout(()=>{ console.log("Привет!") }, 2000);
```

У метода два аргумента. В первом аргументе указывается код, который запускается **после** прекращения работы таймера. Во втором – время задержки в миллисекундах.

Задание таймера, который должен срабатывать через определенные промежутки времени, осуществляется с помощью метода `setInterval()`. Его синтаксис похож на предыдущий метод. В результате заданный код будет запускаться много раз.

```
// повторить с интервалом 2 секунды
const timerId = setInterval(()=>{ console.log("tick"); }, 1000);
// остановить вывод через 10 секунд
setTimeout(()=>{  clearInterval(timerId); console.log("stop"); }, 10000);
```

Реальная частота срабатывания счетчика

У браузерного таймера есть минимальная возможная задержка. В современных браузерах она меняется от примерно 0мс до 4мс. В более старых она может быть больше и достигать 15мс. По стандарту, минимальная задержка составляет 4мс. Так что разницы между setTimeout(..., 1) и setTimeout(..., 4) нет.

В ряде ситуаций таймер будет срабатывать реже, чем обычно. Задержка между вызовами setInterval(..., 4) может быть не 4мс, а 30мс или даже 1000мс. При слишком большой загрузке процессора некоторые запуски функций-интервалов будут пропущены.

Большинство браузеров, в первую очередь десктопных, продолжают выполнять setTimeout и setInterval, даже если вкладка неактивна. При этом ряд из них снижают минимальную частоту таймера до 1 раза в секунду. Получается, что в фоновой вкладке будет срабатывать таймер, но редко.

Колбэки

Пример с setTimeout(), показывает, как работают функции обратного вызова — колбэки.

Callback (колбэк, функция обратного вызова) — функция, которая вызывается в ответ на совершение некоторого события. В целом, событием может быть что угодно:

- ответ от сервера;
- завершение какой-то длительной вычислительной задачи;
- действие пользователя в браузере и пр.

Таким образом колбэк — это первый способ обработать какое-либо асинхронное действие.

Изначально колбэки были единственным способом работать с асинхронным кодом в JavaScript.

Асинхронные колбэки — это функции, которые определяются как аргументы при вызове функции, которая начнёт выполнение кода в фоновом режиме. Когда код в фоновом режиме завершает свою работу, он вызывает колбэк-функцию, оповещающую, что работа сделана, либо оповещающую о трудностях в завершении работы. Обратные вызовы — немного устаревшая практика, но они все ещё употребляются в некоторых старомодных, но часто используемых API.

Пример использования колбэка в addEventListener()

Подписка на событие через addEventListener

```
element.addEventListener(event, handler, [options]);
```

Первый параметр — тип обрабатываемого события, второй параметр — колбэк-функция, вызываемая при срабатывании события.

При передаче колбэк-функции как аргумента в другую функцию, мы передаём только ссылку на функцию как аргумент, следовательно колбэк-функция не выполняется мгновенно. Она вызывается асинхронно внутри тела, содержащего функцию. Эта функция должна выполнять колбэк-функцию в нужный момент.

```
const btn = document.querySelector(".btn");
btn.addEventListener("click", ()=>{
    console.log("click");
})
```

Пример использования колбэка при загрузке скрипта

```
function loadScript(src, callbackF){
    const newScript = document.createElement("script");
    newScript.setAttribute("src", src);
    newScript.onload = () => callbackF(null, newScript);
    newScript.onerror = () => callbackF(new Error(`Не удалось загрузить скрипт  ${src}`),
newScript);
    document.body.appendChild(newScript);
}

loadScript("/path/to/script.js", function (error, script) {
    if (error) {
        // обработка ошибки
    } else {
        newFunction();
        // действия в случае успешной загрузки скрипта
    };
});
```

Ад колбэков (Callback-hell)

У колбэков есть неприятный минус, так называемый ад колбэков (callback hell), он возникает, когда есть ряд асинхронных задач, которые зависят друг от друга: то есть первая задача запускает по завершении вторую, вторая — третью и т. д.

Пример 1

```
setTimeout(() => {
    setTimeout(() => {
      setTimeout(() => {
        setTimeout(() => {
          console.log('Hello!');
        }, 5000);
      }, 5000);
    }, 5000);
  }, 5000);
```

Пример 2

```
loadScript("/path/to/script.js", function (error, script) {
    if (error) {
        // обработка ошибки
    } else {
        newFunction();
        // действия в случае успешной загрузки скрипта
        loadScript("/path/to/script1.js", function (error, script) {
            if (error) {
                // обработка ошибки
            } else {
                newFunction1();
                // действия в случае успешной загрузки скрипта
                loadScript("/path/to/script2.js", function (error, script) {
                    if (error) {
                        // обработка ошибки
                    } else {
                        newFunction2();
                        // действия в случае успешной загрузки скрипта
                    };
                });
            };
        });
    };
});
```

Ад колбэков (Callback-hell)

Чем больше вложенных вызовов, тем наш код будет иметь всё большую вложенность, которую сложно поддерживать, особенно если вместо ... у нас код, содержащий другие цепочки вызовов, условия и т.д.

Пирамида вложенных вызовов растёт вправо с каждым асинхронным действием. В итоге вы сами будете путаться, где что есть.

Такой подход к написанию кода не приветствуется. Вместо вложенных колбеков нужно использовать промисы.

Промисы (Promise)

Promise (обещание, промис) — объект, представляющий текущее состояние асинхронной операции. Удобный способ организации асинхронного кода.

У промиса есть 2 состояния:
- Pending — ожидание, исходное состояние при создании промиса.
- Settled — выполнен, которое в свою очередь делится на две категории: fullfilled — выполнено успешно и rejected — выполнено с ошибкой.

Вначале промис находится в состоянии ожидания (pending), после чего он может выполнится **успешно (fulfilled) или с ошибкой (rejected).** Когда промис переходит в состояние выполнен (settled), с результатом или ошибкой – это навсегда. Грубо говоря, промис – это болванка для данных, значение которых мы не знаем в момент его создания.

Способ использования:
- Код, которому надо сделать что-то асинхронно, создаёт обещание и возвращает его.
- Внешний код, получив обещание, навешивает на него обработчики.
- По завершении процесса асинхронный код переводит обещание в состояние fulfilled или rejected. При этом автоматически вызываются обработчики во внешнем коде.

Отличия промиса и callback-функции:
- Коллбэки – это функции, обещания это объекты.
- Коллбэки передаются в качестве аргументов из внешнего кода во внутренний, обещания возвращаются из внутреннего кода во внешний.
- Коллбэки обрабатывают успешное или неуспешное завершение, обещания ничего не обрабатывают.
- Коллбэки могут обрабатывать несколько событий, обещания связаны только с одним событием.
- Обещания (помимо читабельности, которая является побочным положительным эффектом) используются в основном для композиции, цепочки вызовов и обработки результата и ошибок в отдельных логических ветках.

Промисы (Promise)

```
const promise = new Promise((resolve, reject) => {
        /* Эта функция будет вызвана автоматически. В ней можно выполнять любые асинхронные операции. Когда они завершатся — нужно
         * вызвать одно из: resolve(результат) при успешном выполнении, или reject(ошибка) при ошибке. */
        setTimeout( () => {
            //resolve("all ok");
            reject("not ok");
        }, 2000);
});

promise.then(
    data => {console.log(data)},  // Будет вызвана, если обещание выполнится успешно
    error => {console.log(error)}     // Будет вызвана, если обещание выполнится с ошибкой
);

promise.then( data => {                 // Будет вызвана, если обещание выполнится успешно
        console.log(data)
    }).catch(error => {                  // Будет вызвана, если обещание выполнится с ошибкой
        console.log(error)
    })

promise
    .then( data => {                    // Будет вызвана, если обещание выполнится успешно
        console.log(data)
    })
    .catch(error => {                   // Будет вызвана, если обещание выполнится с ошибкой
        console.log(error)
    })
    .finally( () => {                   // Будет выполнено всегда
        console.log("finished")
    });
```

Промисы (Promise)

```
function loadScript(src){
    return new Promise( (resolve, reject) =>{
        const newScript = document.createElement("script");
        newScript.setAttribute("src", src);
        document.body.appendChild(newScript);
        newScript.onload = () => resolve(newScript);
        newScript.onerror = () => reject(new Error(`Не удалось загрузить скрипт  ${src}`));
    })
}

const promise = loadScript("/path/to/script.js");
promise.then( data => {
    newFunction();
}).catch( error =>{
    console.log(error)
})
```

`Promise.resolve(value)` создаёт успешно выполненный промис с результатом `value`. Этот метод используют для совместимости: когда ожидается, что функция возвратит именно промис.
Аналог для:

```
const promise = new Promise(resolve => resolve(value));
```

Цепочки промисов

Возможность строить асинхронные цепочки из промисов – одна из основных причин существования и активного использования промисов.
Каждый метод then, результатом своего выполнения, возвращает промис. Его значением будет то, что возвращается из callback-функции onResolve.

```
asyncFn(...)
  .then(...)
  .then(...)
  .then(...)
  .catch(...);
```

```
loadScript("/path/to/script.js")
.then( data => loadScript("/path/to/script1.js") )
.then( data => loadScript("/path/to/script2.js") )
.then( data =>{
    newFunction();   //from script
    newFunction1();  //from script1
    newFunction2();  //from script2
})
.catch( error => {
    console.log(error)
});
```

```
loadScript("/path/to/script.js")
.then( data => {
    //some code
    return loadScript("/path/to/script1.js")
})
.then( data => {
    //some code
    return loadScript("/path/to/script2.js")
})
.then( data =>{
    newFunction();   //from script
    newFunction1();  //from script1
    newFunction2();  //from script2
})
.catch( error => {
    console.log(error)
});
```

Асинхронные функции

Существует специальный синтаксис для работы с промисами, который называется «async/await».

Асинхронные функции — функции, которые ВСЕГДА возвращают промисы.

Асинхронная функция помечается специальным ключевым словом async. **У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.**

Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

Все асинхронные функции внутри вызываются с await — таким образом промис, который функция возвращает, автоматически разворачивается, и мы получаем значение, которое было внутри промиса.

Определение асинхронных функций

```
async function request() {}
const req = async () => {}
```

Пример

```
async function showInfo(){
    const response = await fetch('/path/to/data.json');
    const user = await response.json();

    const divUser = document.createElement("div");
    divUser.innerHTML = `<h1>${user.surname} ${user.name}</h1>`
    divUser.innerHTML += `<p>${user.email}</p>`;
    document.body.append(divUser);
}
showInfo()
```

С помощью функции fetch() можно отправлять сетевые запросы на сервер — как получать, так и отправлять данные. Метод возвращает промис с объектом ответа, где находится дополнительная информация (статус ответа, заголовки) и ответ на запрос.

Асинхронные функции

Можно также обрабатывать ошибки с try-catch. Как и с синхронным кодом, обработка ошибок сводится к оборачиванию опасных операций в try-catch:

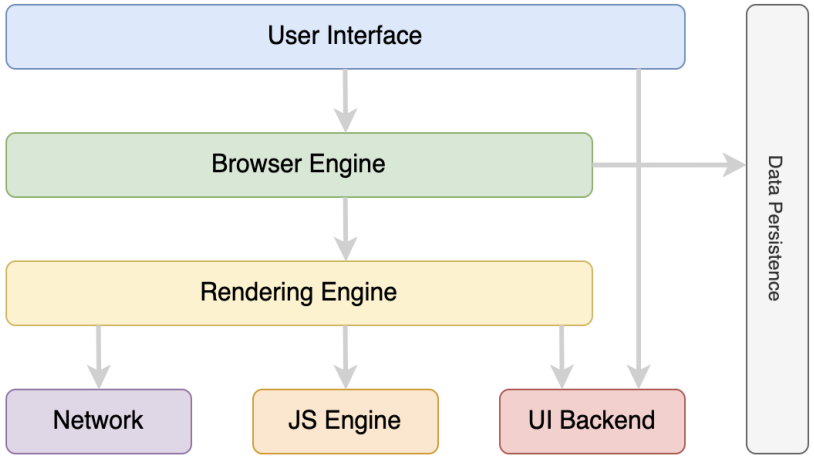
```
async function showInfo(){
    try{
        const response = await fetch('/path/to/data.json');
        const user = await response.json();

        const divUser = document.createElement("div");
        divUser.innerHTML = `<h1>${user.surname} ${user.name}</h1>`
        divUser.innerHTML += `<p>${user.email}</p>`;
        document.body.append(divUser);
    } catch (error) {
        console.log(error.message)
    }
}

showInfo()
```

Архитектура браузера

Браузер состоит из нескольких компонентов, каждый из которых выполняет определенные функции. Эти компоненты включают в себя:



Пользовательский интерфейс (User Interface). Этот компонент браузера включает в себя адресную строку, кнопки «Вперёд» и «Назад», команды для работы с закладками, и так далее. В целом, это всё то, что выводит на экран браузер — за исключением той области его окна, где находится отображаемая им веб-страница.

Браузерный движок – занимается поддержкой взаимодействия между пользовательским интерфейсом и движком рендеринга.

- Blink (представляет собой форк движка WebKit): используется в браузерах Google Chrome, Opera, Microsoft Edge (начиная с версии 79) и других.
- WebKit: используется в браузерах Safari.
- Gecko: используется в браузере Mozilla Firefox.

Рендеринговый движок (Render Engine) – это компонент браузера, который отображает веб-страницы на экране.

JavaScript движок (JS Engine) – это программное обеспечение, которое обрабатывает и исполняет скрипты на веб-страницах. Скрипты могут выполнять различные действия на веб-странице, такие как изменение содержимого страницы, обновление данных, отправка запросов на сервер и многое другое.

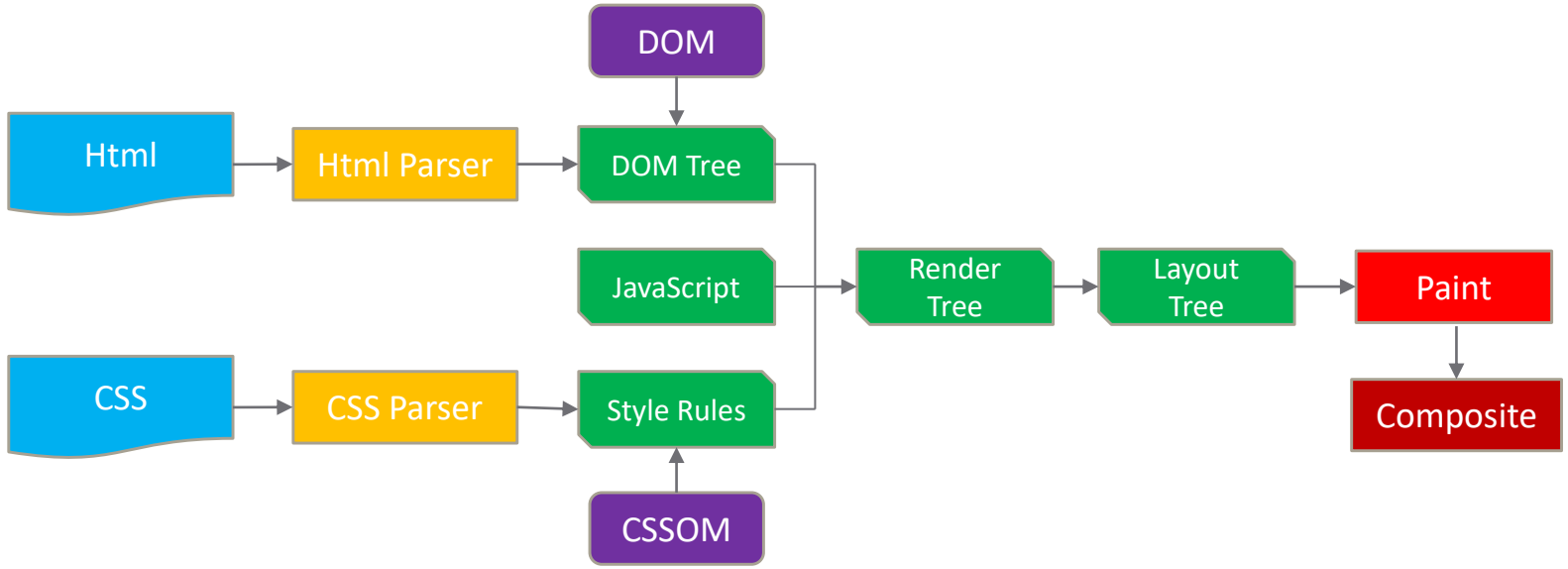
Сетевой слой (Network) – это компонент браузера, который управляет сетевыми запросами и ответами. Он использует протокол HTTP (Hypertext Transfer Protocol) для отправки запросов на сервер и получения ответов.

База данных браузера (DataPersistance) – это хранилище данных, которое используется браузером для хранения истории браузинга, кэша веб-страниц и другой информации.

Критический путь рендеринга

Парсинг
Построение DOM
Получение StyleRules
Выполнение JavaScript

Дерево рендеринга
Деревья, построенные на этапе парсинга, объединяются в дерево рендеринга. Его используют для вычисления расположения всех видимых элементов, которые будут выведены на экран в конце. Результат этих действий — дерево рендеринга со всеми видимыми узлами с содержимым и стилями.



Компоновка (Layout)
В тот момент, когда дерево рендера (render tree) построено, становится возможным этап компоновки (layout). Компоновка зависит от размеров экрана. Этот этап определяет, где и как на странице будут спозиционированы элементы и каковы связи между элементами. Мета тэг viewport, который вы можете указать в Head страницы, определяет ширину видимой области и влияет на компоновку. Без этого тэга браузеры используют ширину "по умолчанию", которая обычно составляет 960px.

Отрисовка (Paint)
Когда дерево рендера (render tree) создано, компоновка (layout) произошла, пиксели могут быть отрисованы. При первичной загрузке документа (onload) весь экран будет отрисован. После этого будут перерисовываться только необходимые к обновлению части экрана, так как браузер старается оптимизировать процесс отрисовки, избегая ненужной работы.

Движок JS

Движки JavaScript, как правило, продукт деятельности разработчиков веб-браузеров. Самые популярные браузеры — Chrome, Safari, Edge и Firefox. У каждого из них свой движок JavaScript:

- V8. Высокопроизводительный движок JavaScript компании Google. Он написан на C++ и используется, в частности, в браузере Chrome и платформе Node.js. Реализует стандарты ECMA-262, ECMA-402 и WebAssembly.
- SpiderMonkey. Движок JavaScript и WebAssembly компании Mozilla. Он написан на C++, JavaScript и Rust и используется в Firefox, Servo и других проектах.
- JavaScriptCore. Встроенный движок JavaScript для WebKit, на котором работает Safari, Mail и другие приложения на macOS. В настоящее время он реализует спецификацию ECMA-262.
- Chakra. Движок JavaScript, разработанный компанией Microsoft для браузера Microsoft Edge и других приложений Windows. Он реализует ECMA-262 версии 5.1 и частично поддерживает версию 6.

В начале движки JavaScript были простыми интерпретаторами. Современные браузеры проводят так называемую компиляцию Just-In-Time (JIT), сочетание компиляции и интерпретации, код преобразуется во время выполнения

Основные задачи, которые решает движок JS:

1. Работа с кучей (heap) и стеком вызовов (call stack)
2. Работа с памятью (выделение памяти и сбор мусора)
3. Компиляция JS в машинный код
4. Оптимизация (кеш, скрытые классы и пр.)

WEB API

Интерфейс прикладного программирования (Application Programming Interfaces, APIs) — это готовые конструкции языка программирования, позволяющие разработчику строить сложную функциональность с меньшими усилиями. Они "скрывают" более сложный код от программиста, обеспечивая простоту использования.
Для JavaScript на стороне клиента, в частности, существует множество API. Они не являются частью языка, а построены с помощью встроенных функций JavaScript для того, чтобы увеличить ваши возможности при написании кода. Их можно разделить на две категории – **API браузера** (встроены в веб-браузер и способны использовать данные браузера и компьютерной среды для осуществления более сложных действий с этими данными) и **сторонние API** (не встроены в браузер по умолчанию).

Перечень интерфейсов WEB API https://developer.mozilla.org/ru/docs/Web/API .

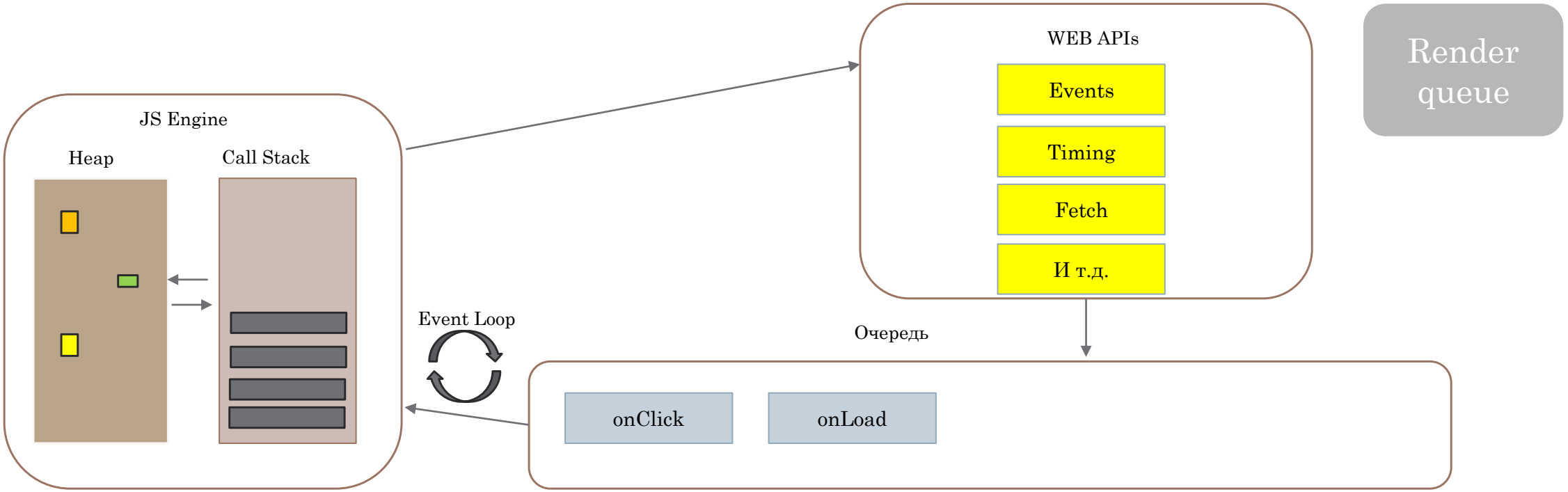
В частности, к наиболее часто используемым категориям API относятся :

- API для работы с документами, загруженными в браузер (DOM API).
- API, принимающие данные от сервера (например XMLHttpRequest и Fetch API).
- API для работы с графикой (например, Canvas и WebGL).
- Аудио и Видео API (например, HTMLMediaElement, Web Audio API, и WebRTC)
- API устройств — в основном, это API для обработки и считывания данных с современных устройств удобным для работы веб-приложений образом. Например, Geolocation API позволяет считать данные о местоположении устройства.
- API хранения данных на стороне пользователя (например,Web Storage API или IndexedDB API).

Цикл событий Event Loop

Цикл событий не является частью движков JS, он предоставляется средой окружения, например браузером или NodeJS. Устройство цикла событий отличается для браузеров и NodeJS.

Для работы сайта браузер выделяет один-единственный поток, который должен успевать одновременно делать две важные задачи: выполнять код и обновлять интерфейс. Но один поток в один момент времени может совершать только одно действие. Поэтому поток выполняет эти задачи по очереди, чтобы создать иллюзию параллельного выполнения. Это и есть цикл событий (event loop).



Цикл событий

- Web API хранит в себе асинхронные операции и не является частью движка JS, но предоставляет окружение, набор API, предоставляемых движку JS для его взаимодействия с вебом.
- Очередь состоит из коллбэков – операций, которые были отложены по времени ввиду асинхронного поведения JS.
- Движок – сердце JS, без которого исполнение кода невозможно. Куча (heap) содержит в себе ссылочные элементы. На взаимодействие с ними тратится больше ресурсов, чем на те, что в стеке, в котором содержатся статичные элементы. Стек помимо хранения примитивных данных выполняет поступающие к нему инструкции.
- Цикл событий (event loop) играет роль дирижёра, который делает корректным выполнение программы.
- Когда стек пустой, ивент луп прокручивается, смотрит в очередь, обнаруживает, есть ли там колбэк, который можно было бы прокинуть в пустой стек. Если есть, то делает это.

Синхронные действия

```
console.log("start");
function f3(){
    console.log("in f3")
}
function f2(){
    f3()
}
function f1(){
    f2();
}
f1();
console.log("finish")
```

Выполнение кода происходит в стеке вызовов.
Стек — это структура данных, в которой элементы упорядочены так, что последний элемент, который попадает в стек, выходит из него первым (LIFO: last in, first out). Стек похож на стопку книг: та книга, которую мы кладём последней, находится сверху. В стеке вызовов хранятся функции, до которых дошёл интерпретатор, и которые надо выполнить.
Если функция внутри себя вызывает другую функцию, то вызов её самой приостанавливается до тех пор, пока выполняется другая функция, — таким образом получается эдакий стек вызовов. Как только все операции будут выполнены и стек опустеет, цикл событий может поместить в стек ещё какой-нибудь код для выполнения. В синхронном коде в стеке хранится вся цепочка вызовов.

Стек вызовов										
					console.log					
				f3	f3	f3				
			f2	f2	f2	f2	f2			
console.log		f1	f1	f1	f1	f1	f1	f1		console.log

Цикл событий

```
console.log("start");
setTimeout(hello = () => {
    console.log("in timer");
}, 1000);
console.log("finish")
```

Очередь задач											
							hello				
Web API											
			Timer hello	Timer hello	Timer hello	Timer hello					
Стек вызовов											
									console.log("in timer")		
console.log ("start")		setTimeout hello	setTimeout hello		console.log ("finish")			hello	hello	hello	

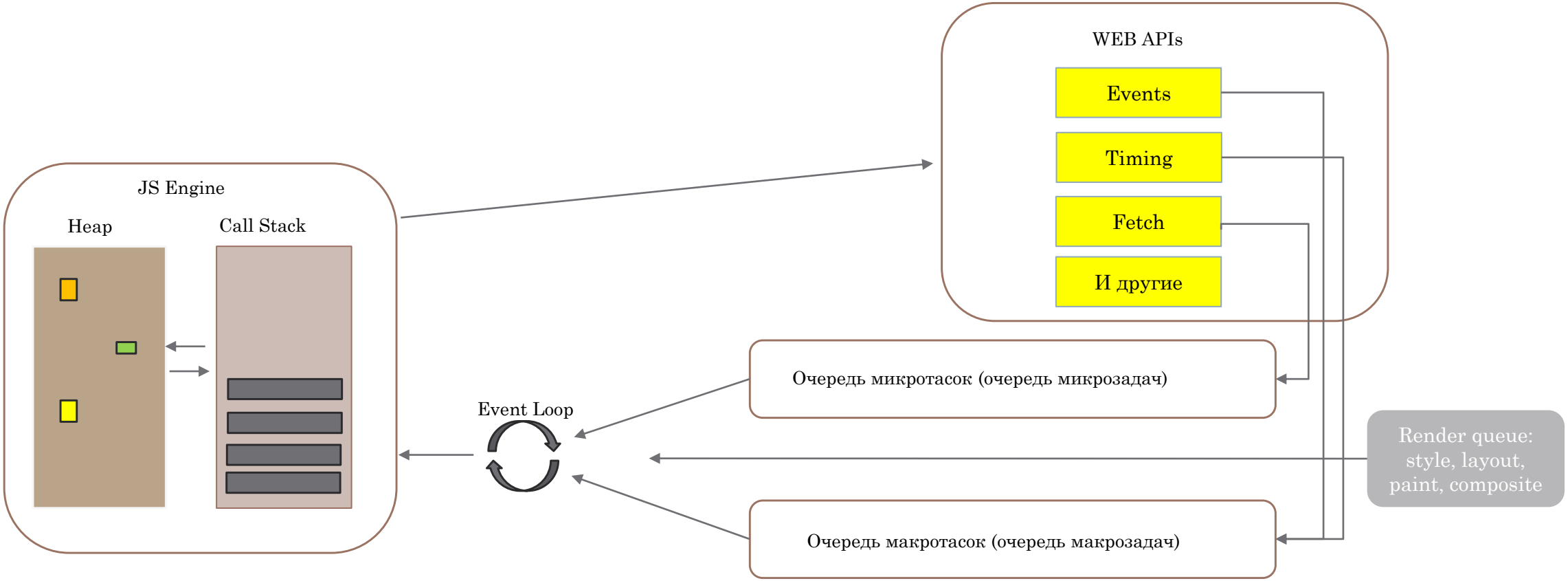
Цикл событий

```
btn1.addEventListener("click", function (){
    console.log("click1");
})
btn2.addEventListener("click", function (){
    console.log("click2");
})
```

Очередь задач											
						console.log("click 1");	console.log("click 2");				
Web API											
	Button1	Button1	Button1	Button1 Button2	Button1 Button2	Button1 Button2	Button1 Button2	Button1 Button2	Button1 Button2	Button1 Button2	Button1 Button2
Стек вызовов											
btn1.addEventListener	btn1.addEventListener		btn2.addEventListener	btn1.addEventListener			console.log("click 1");		console.log("click 2");		

Микротаски и макротаски

Существуют две очереди асинхронных операций: очереди микро- и макротасок. Микротаски имеют более высокий приоритет исполнения. Поэтому когда event loop прокручивается, ему не важно, сколько в очереди макротасок – он возьмёт микротаску, даже если она в очереди единственная, а макротасок 100.
Все микрозадачи завершаются до обработки каких-либо событий или рендеринга, или перехода к другой макрозадаче.



Микротаски и макротаски

Примеры **макрозадач**:

- Когда загружается внешний скрипт <script src="...">, то задача – это выполнение этого скрипта.
- Когда пользователь двигает мышь, задача – сгенерировать событие mousemove и выполнить его обработчики.
- Когда истечёт таймер, установленный с помощью setTimeout(func, ...), задача – это выполнение функции func
- и пр.

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.

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.

Спасибо за внимание!