Event Loop

Как работает JavaScript

JavaScript-движки

JavaScript является одновременно компилируемым и интерпретируемым языком. На самом деле JavaScript-движки компилируют ваш код за микросекунды до его исполнения.Это называется **JIT (Just in time compilation).**

И да JavaScript компилируется не браузером а JavaScript движком который в каждом браузере свой. Самыми популярными движками являются V8 в Google Chrome и Node.js, SpiderMonkey в Firefox, JavaScriptCore в Safari/WebKit.

JavaScript-движки

Давайте посмотрим на следующий код:

```
var val = 5;

function sum(num) {

    return num + num;

}
```

Как вы думаете как обрабатывается этот код?

JavaScript-движки

Код считывает не браузер, а движок. JavaScript-движок считывает код, и как только он определяет первую строку, то кладёт ссылки в глобальную память.

Глобальная память (называется кучей (**heap**)) — это область, в которой JavaScript-движок хранит переменные и объявления функций. И когда он прочитает приведённый выше код, то в глобальной памяти появятся два биндинга

Global Memory

val = 5

sum = function

JavaScript-движки

Так как JavaScript исполняется в какой то среде: в браузере или Node.js. В таких средах есть много заранее существующих функций и переменных, которые называют глобальными. Поэтому глобальная память будет содержать гораздо больше данных, чем просто val и sum

JavaScript-движки

Давайте теперь попробуем вызвать нашу функцию.

Что произойдет?

JavaScript-движок выделит два раздела:

1. Глобальный контекст исполнения (Global Execution Context)
2. Стек вызовов (Call Stack)

```
var val = 5;


function sum(num) {

    return num + num;

}


sum(val);
```

Global Execution Context & Call Stack

Итак мы знаем что JavaScript-движок читает переменные и объявления функций и записывает их в глобальную память (кучу).

Но теперь мы исполняем JavaScript-функцию, и движок должен с этим что-то сделать. У каждого JavaScript-движка есть важный компонент, который называется стек вызовов (call stack).

В call stack элементы могут добавляться сверху, но они не могут исключаться из структуры, пока над ними есть другие элементы. Именно так устроены JavaScript-функции. При исполнении они не могут покинуть стек вызовов, если в нём присутствует другая функция.

Global Execution Context & Call Stack

CALL STACK

sum()

GLOBAL MEMORY

val = 5

sum = function

Global Execution Context & Call Stack

Call stack можно представить в виде стопки чипсов Pringles. Мы не можем съесть чипс снизу стопки, пока не съедим те что сверху.

В то же самое время движок размещает в памяти глобальный контекст исполнения, это глобальная среда, в которой исполняется JavaScript-код.

Global Execution Context & Call Stack

CALL STACK

sum()

GLOBAL EX. CONTEXT

sum

GLOBAL MEMORY

val = 5

sum = function

Global Execution Context & Call Stack

Даже в простом случае, как показано ниже, JavaScript-движок создаёт локальный контекст исполнения:

```
var val = 2;

function sum(num) {

    var x = 10;

    return num + x;

}

sum(val);
```

Global Execution Context & Call Stack

Обратите внимание, что я добавил в функцию **sum** переменную **x**. В этом случае локальный контекст исполнения будет содержать раздел для **x**.

И так для каждой вложенной функции внутри вложенной функции движок создаёт другие локальные контексты исполнения. Все эти разделы-прямоугольники появляются очень быстро! Как матрёшка!

Global Execution Context & Call Stack

CALL STACK

sum()

GLOBAL EX. CONTEXT

sum

Local context
x = 10

GLOBAL MEMORY

val = 5

sum = function

Однопоточность

Мы говорим, что JavaScript является однопоточным, потому что наши функции обрабатывает лишь один стек вызовов. Напомню, что функции не могут покинуть стек вызовов, если исполнения ожидают другие функции.

Это не проблема, если мы работаем с синхронным кодом. К примеру, сложение двух чисел является синхронным и вычисляется за микросекунды.

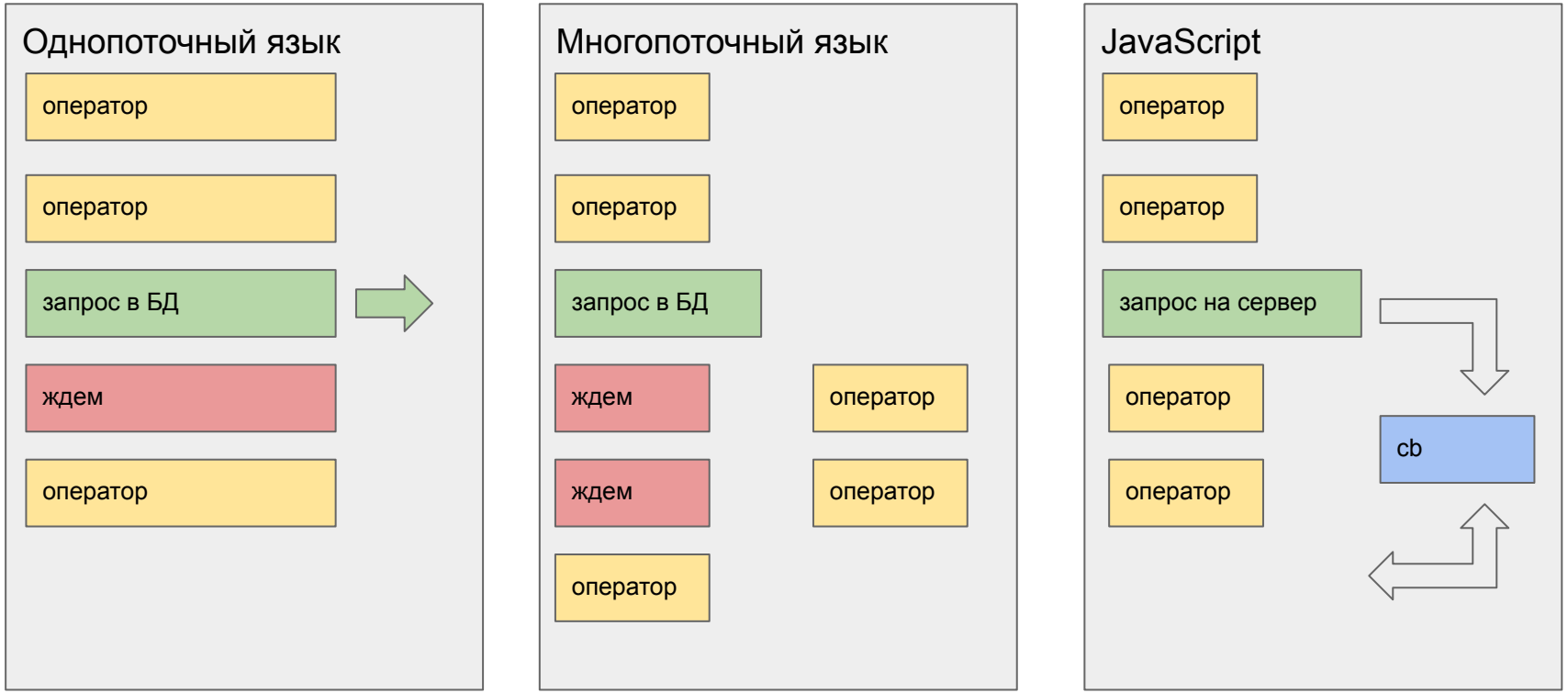
А как быть с сетевыми запросами, setTimeout и т.д

Однопоточность

Давайте подытожим что мы прошли. когда браузер загружает какой-то JavaScript-код, движок считывает этот код строка за строкой и выполняет следующие шаги:

- Помещает в глобальную память (кучу) переменные и объявления функций.
- Отправляет вызов каждой функции в стек вызовов.
- Создает глобальный контекст исполнения, в котором исполняются глобальные функции.
- Создает много маленьких локальных контекстов исполнения (если есть внутренние переменные или вложенные функции).

Асинхронный JavaScript



Асинхронный JavaScript

Благодаря глобальной памяти, контексту исполнения и стеку вызовов синхронный JavaScript-код исполняется в наших браузерах. Но что происходит, если нужно исполнить какую-нибудь асинхронную функцию?

Под асинхронной функцией я подразумеваю запросы к API или таймеры Это все асинхронные действия и на их выполнение могут уйти секунды. Благодаря имеющимся в движке элементам мы можем обрабатывать такие функции без блокирования стека вызовов и браузера. Не забывайте, стек вызовов может исполнять одновременно только одну функцию, и даже одна блокирующая функция может буквально остановить браузер. К счастью, JavaScript-движки «умны», и с небольшой помощью

Асинхронный JavaScript

Уверен, что хоть вы и видели setTimeout уже сотни раз, однако можете не знать, что эта функция не встроена в JavaScript. Вот так, когда JavaScript появился, в нём не было функции setTimeout. По сути, она является частью так называемых браузерных API, коллекции удобных инструментов, которые нам предоставляет браузер. Чудесно! Но что это означает на практике? Поскольку setTimeout относится к браузерным API, эта функция исполняется самим браузером (на мгновение она появляется в стеке вызовов, но сразу оттуда удаляется).

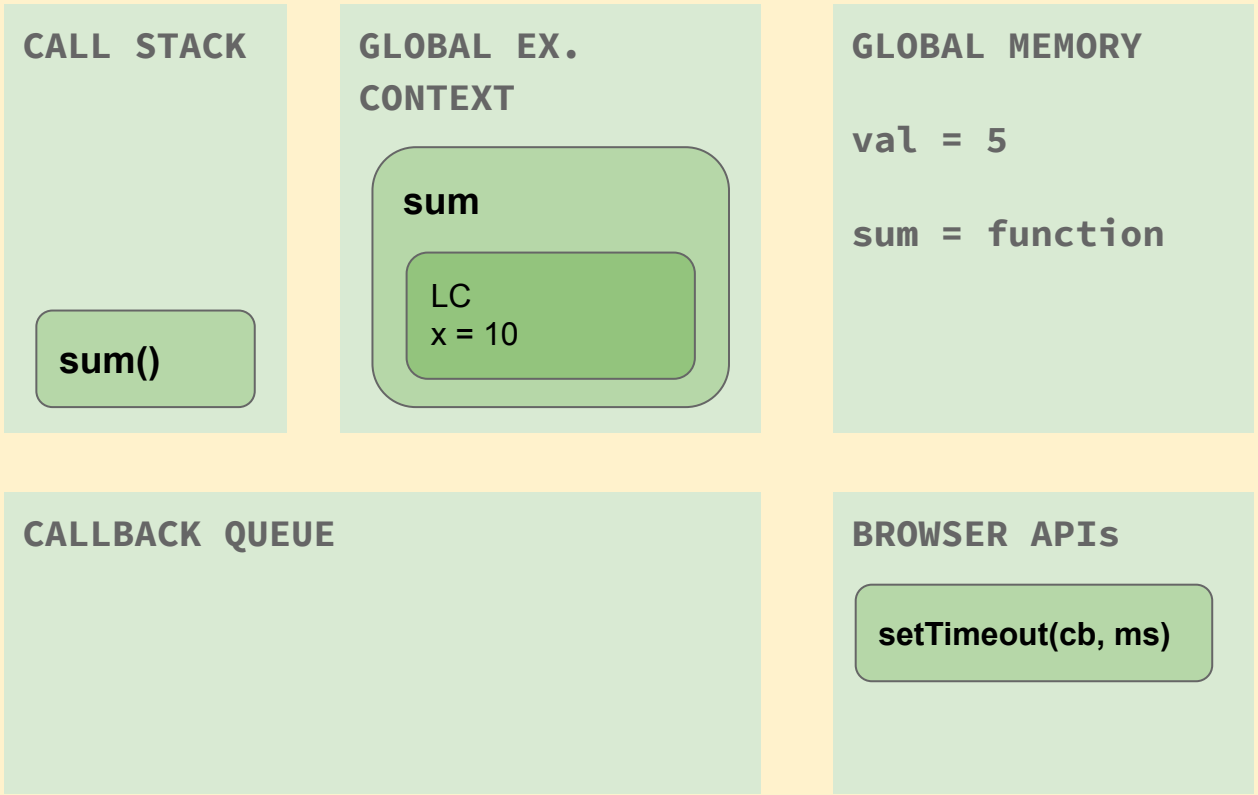
Асинхронный JavaScript

Давайте посмотрим на следующий код.

Через 10 секунд браузер берёт callback-функцию, которую мы ему передали, и кладёт её в очередь обратных вызовов. В данный момент в JavaScript-движке появилось ещё два раздела это Browser APIs и Callback Queue.

```
var val = 2;

function sum(num) {

    return num + num;

}

sum(val);

setTimeout(callback, 10000);

function callback(){

    console.log('hello timer!');

}
```

Асинхронный JavaScript



Асинхронный JavaScript

setTimeout исполняется внутри контекста браузера. Через 10 секунд таймер запускается и callback-функция готова к исполнению. Но для начала она должна пройти через очередь обратных вызовов. Это структура данных в виде очереди, и, как свидетельствует её название, представляет собой упорядоченную очередь из функций.

Каждая асинхронная функция должна пройти через очередь обратных вызовов, прежде чем попасть в стек вызовов. Но кто отправляет функции дальше? Это делает компонент под названием цикл событий (event loop).

Асинхронный JavaScript

Пока что цикл событий занимается только одним: проверяет, пуст ли стек вызовов. Если в очереди обратных вызовов есть какая-нибудь функция и если стек вызовов свободен, тогда пора отправлять callback в стек вызовов.

После этого функция считается исполненной. Так выглядит общая схема обработки асинхронного и синхронного кода JavaScript-движком:

CALL STACK

sum()

GLOBAL EX. CONTEXT

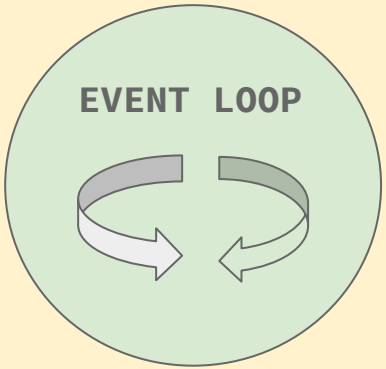
sum

LC
x = 10

GLOBAL MEMORY

val = 5

sum = function



CALLBACK QUEUE

callback()

BROWSER APIs

setTimeout(cb, ms)

Допустим, callback() готова к исполнению. После завершения исполнения sum() стек вызовов освобождается и цикл событий отправляет в него callback().
Помните: браузерные API, очередь обратных вызовов и цикл событий являются столпами асинхронного JavaScript.
Важно. Источником синхронизации является функция и setTimeout гарантирует выполнение функции не раньше чем через 10сек но это может произойти позже.

Асинхронный JavaScript

Материалы:

https://www.youtube.com/watch?v=8aGhZQkoFbQ

https://habr.com/ru/company/oleg-bunin/blog/417461/

https://tylermcginnis.com/javascript-visualizer/