Event Loop

Рассмотрим такой код:

console.log ('1 log');

console.log ('2 log');

setTimeout (function () {

console.log ('3 log');

}, 4000);

console.log ('4 log');

setTimeout (function () {

console.log ('end');

}, 1500);

Все, что вывелось консоль, не в заданном порядке. Дело в том, что есть концепция *event loop* – интерпретатор проходит и выполняет команды, но если команда *асинхронна*, т.е. вызовется в независимости от потока и времени, то она кинется в стек и там уже после выполнения всех *синхронных* операций, интерпретатор начнет разбирать асинхронные операции.

Все, что вывелось консоль, не в заданном порядке. Дело в том, что существуют два типа команд *синхронные* и *асинхронные*.

**Синхронные** операции вызываются сразу в тот момент, когда интерпретатор обрабатывает их. **Асинхронные** же операции могут вызваться в любом потоке и времени, к примеру, событие. Примеры асинхронности:

* setTimeout и setInterval
* события
* работа с сервером

Все работает, так как есть понятие call stack, web apis и callback queue. Это места, в которых хранятся команды.

Первое это **call stack** – это место куда интерпретатор, кидает все синхронные команды. Если же команда асинхронная, то она попадет в зону **web apis**, которая начнется выполняться, только тогда когда call stack станет пустым.

Когда начнется web apis, то он начнет отправлять в **callback queue** синхронные команды, например, когда протечет время, вызовется событие и т.д.

Уже в самой callback queue зоне создается **event loops** – цикл, который перебирает присланные команды и отправляет их в call stack и там уже интерпретатор выполняет их.

Получается асинхронные команды – это команды (далее как вызовы) выполняются в любом потоке и времени в программе. Получается – это необходимость и ее нужно продумать, как организовать. Есть два способа и один неполноценный, но который используется чаще всего.

Пирамида вызовов

Т.к. асинхронные команды не выполняются по порядку, то нужно организовать их работу. Допустим у нас функция, которая добавляет на сайт новый скрипт и из него нужно вызвать метод.

Код, который выполнится с ошибкой, ведь загрузка скрипта – это асинхронная команда:

function loadStript(path) {

let script = document.createElement ('script');

script.src = path;

document.head.append (script);

}

loadStript('t2.js');

someFuncFromScript();

Как вариант – это добавить callback функцию:

function loadStript(path, callback) {

let script = document.createElement('script');

script.src = path;

script.onload = () => callback (path);

document.head.append (script);

}

loadStript ('t2.js', (src) => {

someFuncFromScript();

});

Достоинство в том, что можно сразу организовать несколько вызовов:

loadStript ('t2.js', (src) => {

someFuncFromScript ();

loadStript ('t3.js', (src => {

// ...

}))

});

Но если будет уже более трех, то пирамида будет слишком громоздкая и ее можно назвать **адская пирами вызовов** и ее будет еще труднее поддерживать и добавлять.

Промисы

**Промисы** – это механизм языка, встроенный в сам язык программирования и служит для организации асинхронного кода.

В основу лег класс Promise, которому передается callback. Содержащий асинхронный код:

let p =

new Promise((resolve, reject) => {

setTimeout (() => {

console.log ('Подготовка данных')

const data = {

server: 'asw',

port: 2000,

states: 'working'

}

resolve(data);

}, 2000);

})

Этот код можно назвать **создающим кодом** и у него может быть **потребители**, те, которые должны выполняться после создающего кода.

Создающий код содержит две функции resolve и reject, которые обозначают, как выполнен код успешно или не успешно код соотвественно. Первому нужно передать один объект результат работы, а второму объект ошибки.

Для создания потребителей используется три метода Promise:

1. then – выполнятся, если в потребительском коде вызван resolve. Нужно передать callback с одним значением.
2. catch – выполняется, если в потребительском коде вызван reject. Нужно передать callback с одним значением.
3. finally – выполнится в любом случаи и используется, в качестве роли finally в обработке ошибок. Нужно передать callback без параметров. К примеру, остановить преолайдер.

Все эти три метода создают так называемую**цепочку промисов**.

p

.then (value => {

return new Promise((resolve, reject) => {

setTimeout (() => {

let str = '';

for (let k in value) str += `${k}: ${value[k]}\n`;

console.log ('Обрабатываем результат')

resolve (str)

}, 2400);

})

})

.then (value => {

console.log (value);

})

.catch (error => {

console.error (error)

})

.finally (() => {

console.log('Останавливаем процесс');

})

У класса Promise есть мелкое API для работы с промисами. Метод race, который вернет, тот промис, который выполнится раньше всех. Метод all, который дождется выполнения всех переданных промисов, и вернем массив с их значениями.

Пример их использования:

function sleep(ms) {

return new Promise(resolve => {

setTimeout (() => {

resolve(ms / 1000)

}, ms);

})

}

Promise.all ([

sleep(2000),

sleep(4000),

sleep(1000)

])

.then(console.log);

Promise.race ([

sleep(3000),

sleep(2000),

sleep(7000)

])

.then(value => console.log('Первым пришел со временем ' + value + ' sc'));

Async / Await

Синтаксический сахар async / await – это упрощение then, где await позволяет остановиться на время для выполнения промиса и получения его результата, но все это делается, только async-функциях.

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout (() => resolve("готово!"), 1000)

});

let result = await promise;

console.log (result);

}

f()

Функция f здесь неявно вернет Promise и возврат из функции здесь будет resolve(то, что вернули).