Event Loop (цикл событий) - является частью ядра Node.js и отвечает за управление асинхронными операциями, такими как операции ввода/вывода (I/O), таймеры и события.

Принцип работы Event Loop в Node.js можно описать следующим образом:

1. **Инициализация**: Когда Node.js приложение запускается, Event Loop начинает свою работу. Он инициализирует глобальные объекты и регистрирует обработчики событий.
2. **Выполнение синхронного кода**: Event Loop начинает выполнение синхронного кода вашей программы. Это код, который выполняется сразу при старте приложения и не является асинхронным.
3. **Ожидание асинхронных операций**: Когда Event Loop достигает асинхронной операции (например, чтение файла, сетевой запрос), он делегирует выполнение этой операции библиотеке Node.js, такой как libuv. Event Loop продолжает выполнение следующего кода, не блокируя основной поток.
4. **Регистрация колбэков**: При начале асинхронной операции, вы можете зарегистрировать колбэк (функцию обратного вызова), который будет вызван после завершения операции. Этот колбэк будет добавлен в очередь событий Event Loop.
5. **Обработка событий**: Event Loop постоянно проверяет очередь событий на наличие колбэков, которые готовы к выполнению. Как только такой колбэк найден, Event Loop вызывает его.
6. **Возврат к ожиданию**: Если все асинхронные операции завершены и нет других событий для обработки, Event Loop возвращается к ожиданию новых событий.
7. **Таймеры и интервалы**: Event Loop также управляет таймерами и интервалами, позволяя выполнять код по расписанию. Если у вас есть задержка или интервалы в вашем коде, Event Loop следит за их выполнением.
8. **Выход из приложения**: Когда основной код программы завершает выполнение, Event Loop завершает работу, и приложение завершает выполнение.

<https://nodejsdev.ru/guides/webdraftt/event-loop/>

1. **Структура calback-очереди**



1. **Event Loop: макрозадачи** - выполняются по одной за один проход цикла; **микрозадачи** - на каждом проходе цикл выполняет все накопившееся.
2. **Макрозадачи:** setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering
3. **Микрозадачи:** process.nextTick, Object.observe, Promises
4. **Микрозадачи:** специальная функция **queueMicrotask (func**) - ставит func в очередь на выполнение в очереди микрозадач.

**18. Основные глобальные объекты Node.js и их назначение:**

Node.js предоставляет несколько глобальных объектов, которые могут быть использованы в любом модуле вашего приложения. Некоторые из основных глобальных объектов в Node.js включают:

* **global**: Это корневой объект, который представляет глобальную область видимости в Node.js. Все глобальные переменные и функции доступны как свойства этого объекта. Однако, его использование не рекомендуется, и стоит избегать глобальных переменных во избежание конфликтов и нежелательного воздействия.
* **process**: Этот объект представляет текущий процесс Node.js и предоставляет доступ к его свойствам и методам. Например, **process.env** используется для доступа к переменным окружения, а **process.exit()** используется для завершения процесса.
* **console**: Объект **console** предоставляет методы для вывода информации в консоль. Это удобно для отладки и регистрации информации.
* **require**: Это функция, используемая для импорта модулей в вашем приложении. Вы можете использовать **require** для подключения внешних библиотек и модулей Node.js.
* buffer - класс Buffer – предназначен для работы с двоичными данными.

**19. Асинхронная функция:**

Асинхронная функция (асинхронный вызов) - это функция, которая выполняется асинхронно, то есть она не блокирует выполнение программы и позволяет продолжать работу с другими операциями, не ожидая завершения данной функции.

Основная особенность асинхронных функций заключается в том, что они используют колбэки, промисы, асинхронные/ожидающие операции ввода-вывода или асинхронные API для выполнения задачи. Примерами асинхронных операций являются чтение файла, сетевые запросы, таймеры и другие.

Пример асинхронной функции с использованием колбэка:

javascriptCopy code

function fetchData(callback) { setTimeout(() => { const data = 'Данные из асинхронной функции'; callback(data); }, 1000); } fetchData((result) => { console.log(result); });

**20. Стандартные системные потоки:**

Стандартные системные потоки в Node.js представляют собой три стандартных потока ввода/вывода, которые связаны с процессом Node.js:

* **process.stdin**: Стандартный входной поток, который обычно связан с клавиатурой и позволяет программе принимать ввод от пользователя.
* **process.stdout**: Стандартный поток вывода, который используется для отправки данных на экран или в другие места, например, в консоль.
* **process.stderr**: Стандартный поток ошибок, предназначен для вывода ошибок и диагностических сообщений.

Стандартные потоки позволяют взаимодействовать с окружающей средой, отправлять и получать данные, а также обрабатывать ошибки. Они часто используются в Node.js для взаимодействия с консолью и другими процессами.

**21. Функции process.nextTick и setImmediate, и их разница:**

* **process.nextTick**: Функция **process.nextTick** позволяет добавить колбэк в начало очереди микрозадач (microtask queue) Event Loop. Это означает, что колбэк будет выполнен как можно быстрее, после завершения текущей операции Event Loop, до начала выполнения следующей фазы Event Loop. Она предоставляет высший приоритет выполнения.

Пример:

javascriptCopy code

process.nextTick(() => { console.log('Этот код выполнится максимально быстро'); });

* **setImmediate**: Функция **setImmediate** планирует выполнение колбэка в очереди таймеров Event Loop, после текущей фазы Event Loop. Она предоставляет более низкий приоритет выполнения по сравнению с **process.nextTick**.

Пример:

javascriptCopy code

setImmediate(() => { console.log('Этот код выполнится после текущей фазы Event Loop'); });

Разница между ними заключается в приоритете выполнения. **process.nextTick** выполняется сразу после завершения текущей операции Event Loop, в то время как **setImmediate** будет выполнен после того, как текущая фаза Event Loop завершится и начнется следующая фаза. Иногда это может иметь значение в порядке выполнения вашего кода, особенно при решении проблем с глубокой рекурсией или производительностью.