GeekBrains

Курс по Node.js

Цикл событий. События в Node.js

[Node.js v14.x]

На этом уроке

1. Поговорим о внутреннем устройстве Node.js: движке V8, библиотеке libuv.
2. Узнаем, что такое «цикл событий» и зачем он нужен.
3. Рассмотрим порядок по выполнению операций в цикле событий.
4. Разберём функции setTimeout, setInterval и setImmediate.
5. Узнаем, что такое события.
6. Рассмотрим, как события применяются в Node.js.
7. Ознакомимся со стандартным модулем Events и его базовым инструментарием.

Оглавление

Теория урока

В прошлом уроке мы говорили, что Node.js состоит из движка V8, библиотеки libuv и стандартных модулей.

1. Движок V8 отвечает за выполнение JS-кода.
2. Библиотека libuv, написанная на C++, отвечает за взаимодействие с возможностями операционной системы.
3. Стандартные модули покрывают базовые необходимости программной среды.

В этом уроке мы подробнее изучим, что именно происходит внутри Node.js, когда программист запускает свой код.

Введение

JavaScript-код выполняется в однопоточном режиме. Это значит, что в каждый момент времени выполняется какая-то одна конкретная операция. Но если сравнивать Node.js с PHP, например, то может показаться, что однопоточность — это недостаток.

Интуитивно кажется, что чем больше потоков поддерживает среда, тем выше производительность. Однако, это не всегда так. И в случае с Node.js, однопоточность — это преимущество. Своим появлением Node.js обязан стремлению обойти проблемы многопоточности.

Многопоточные серверы

Такие серверы, принимая запрос от клиента, создают под него отдельный поток, то есть выделяют какие-то серверные ресурсы (CPU, память и т. д.) для обработки этого запроса. Количество потоков, которые сервер может обрабатывать — конечно. Обозначим это количество буквой N. Когда на сервер придёт запрос N + 1 — сервер не сможет на него ответить из-за обработки предыдущих запросов. Запрос будет ожидать, когда для него освободятся ресурсы. А потом появятся запросы N+2, N+3 и другие.

Эту проблему не получится решить простым добавлением ресурсов к серверу. Ресурсы конечны, а проблема рано или поздно появится снова.

Блокирующие операции ввода-вывода

Ограниченное число одновременно обрабатываемых сервером потоков — не единственная проблема. Есть ещё проблема нерационального использования ресурсов этих потоков. Это связано с так называемыми блокирующими операциями ввода-вывода.

Всё, что работает с вводом-выводом данных на диск, в память, по сети, работает гораздо медленнее, чем простой код. Поэтому поток вынужден простаивать, ожидая чтения или записи файла, выборки из базы данных или запроса в сторонний ресурс по сети.

Для простоты приведём пример. Студент заходит в личный кабинет на учебном портале и открывает страницу с уроками, чтобы посмотреть пройденные и оставшиеся уроки. Происходит следующее:

1. На сервер приходит запрос на страницу с уроками для конкретного авторизованного студента.
2. Открывается поток, который ищет подходящий запросу обработчик — участок кода, предназначенный для обработки запросов такого типа. Поток работает.
3. Подходящий код найден, начинается обработка запроса, то есть выполнение соответствующих инструкций. Поток работает.
4. В файл записываются логи приложения — пришёл такой-то запрос, с такими-то параметрами, в такое-то время, от такого-то клиента. Пока идёт запись в файл — поток ждёт.
5. Запись логов завершена, продолжается обработка запроса. Поток работает.
6. Теперь требуется сделать выборку из базы данных, чтобы узнать, какие уроки студент уже прошёл, а какие — нет. Поток ждёт.
7. Данные получены, запускается рендеринг HTML-страницы с уроками студента. Поток работает.
8. Сервер отдаёт страницу клиенту, и поток закрывается.

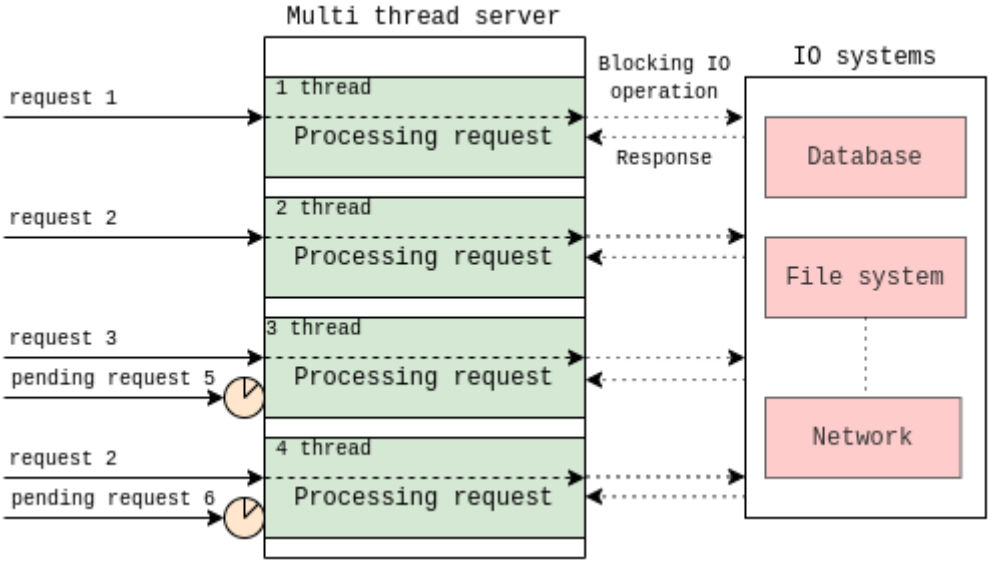

Рис. 1. Принцип работы многопоточного сервера

В этом примере два раза поток простаивал, ожидая выполнения долгой операции. При этом мы ещё не логировали ответ, не обращались к сторонним API за дополнительными данными и не читали файлы.

Ниже в таблице приводятся трудозатраты процессора на различные задачи:

Операция	**Количество тактов CPU**
CPU Registers	3 такта
L1 Cache	8 тактов
L2 Cache	12 тактов
RAM	150 тактов
Disk	30 000 000 тактов
Network	250 000 000 тактов

По мере развития компьютерных технологий это время сокращается, но разница между вычислениями и операциями ввода-вывода очевидна.

Проблема C10k

С английского C10k расшифровывается как 10k connections — проблема 10 тысяч соединений. Понятие возникло примерно в 2000 году, когда широко стояла задача по созданию и конфигурированию сервера, который мог бы одновременно обрабатывать 10 тысяч входящих соединений. И если зачастую были машины, технически позволяющие обрабатывать такое

количество запросов, то проблемы возникали с программными возможностями. Отдельные потоки для каждого запроса потребляли много ресурсов, при этом огромная их часть расходовалась впустую.

Сейчас, с ростом технологий, большинство серверов и программных фреймворков умеют справляться с проблемой C10k.

Асинхронный однопоточный сервер

Для решения вышеописанных проблем появился Node.js, который работает в асинхронном однопоточном режиме. Исключение — случай, когда программисты сами добавляют поддержку многопоточности своему серверу. Для этого используется встроенный в Node.js инструментарий модуля worker_threads, который в этом курсе не рассматривается.

Итак, Node.js по умолчанию принимает все запросы к серверу в один поток. При этом операции ввода-вывода он запускает в асинхронном неблокирующем режиме. Это значит, что при запуске таких операций система не простаивает, ожидая, пока они завершатся, а выполняет другие инструкции.

На рисунке 2 приводится схема, которая наглядно объясняет логику работы асинхронного сервера на Node.js.

Рассмотрим её пошагово.

1. Все запросы, приходящие на сервер, попадают в один поток, в котором их обрабатывает цикл событий.
2. При обработке запроса цикл событий выгружает в операционную систему задачи, связанные с вводом-выводом.
3. В ожидании ответа по задаче ввода-вывода система обрабатывает другие запросы.
4. Как только ответ пришёл — срабатывает так называемый callback — функция, которая обрабатывает данные, возвращаемые асинхронной операцией. Запрос обрабатывается дальше по описанным выше шагам, и в конце ответ возвращается клиенту.

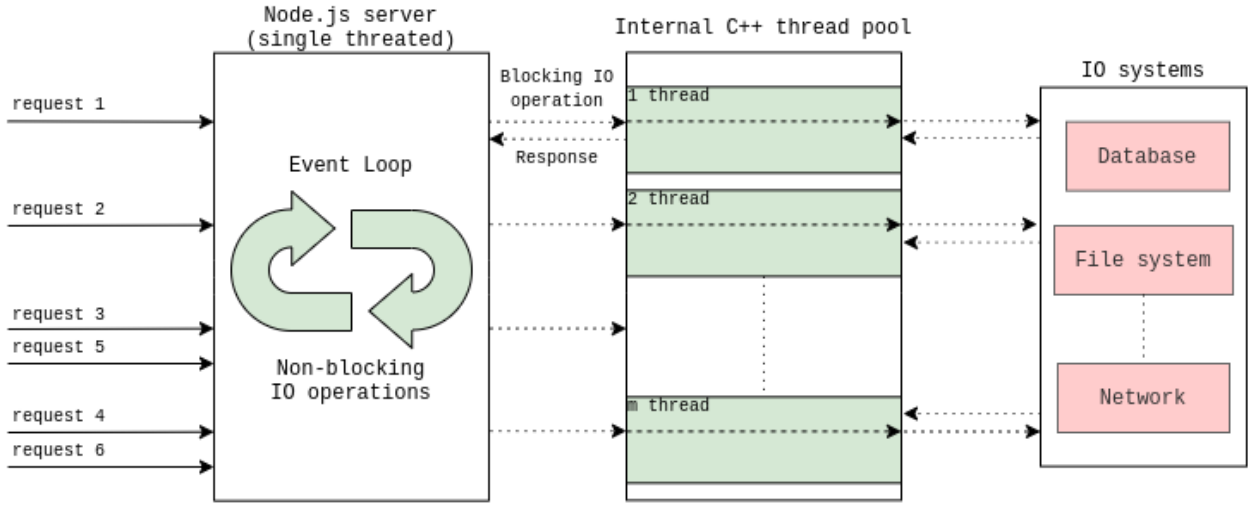

Рис. 2. Асинхронный однопоточный Node.js-сервер

Важно! *Такой цикл, как while(true) в многопоточном сервере заблокирует только один поток. В однопоточном асинхронном сервере блокируются основной и единственный поток. Это надо учитывать при написании кода, особенно при использовании рекурсии.*

Если бы сервер в предыдущем примере был написан на Node.js, то в моменты записи логов в файл и выборки из базы данных он бы не простаивал, ожидая результаты, а обрабатывал другие запросы. Узнаем, благодаря чему.

Цикл событий (Event Loop)

Цикл событий (Event Loop) позволяет Node.js запускать операции ввода-вывода в асинхронном режиме, не блокируя основной поток. Это происходит благодаря тому, что в цикле событий такие операции выгружаются в ядро системы и выполняются там. Их принято называть асинхронными операциями. Node.js остаётся только дождаться результата выполнения асинхронной операции и продолжить с ним работать. При этом в процессе ожидания результата Node.js может выполнять другой JS-код. А так как все современные ядра — многопоточные, у Node.js появляется возможность выгружать в систему не одну операцию ввода-вывода, а несколько.

По сравнению с многопоточными системами такой подход позволяет избавить программиста от решения проблем по типу состязания за ресурсы, синхронизации потоков и других. Нужно лишь сосредоточиться на том, что именно выполняет написанный код, и избегать блокировки потока. То есть не запускать такие задачи, которые «займут» его на долгое время. Например, запускать цикл while (true) — это плохая идея.

Структура цикла событий

При запуске Node.js в стандартном режиме, не в REPL, происходит инициализация цикла событий, и запускается обработка всех инструкций, которые в него попадают. Таким образом, через цикл событий

проходит весь написанный программистом код. Пока есть что выполнять, цикл событий работает бесконечно.

При этом у цикла событий есть свой порядок выполнения операций, или так называемые фазы цикла событий. Всего фаз 6, и они изображены на рисунке ниже. Рассмотрим их все.

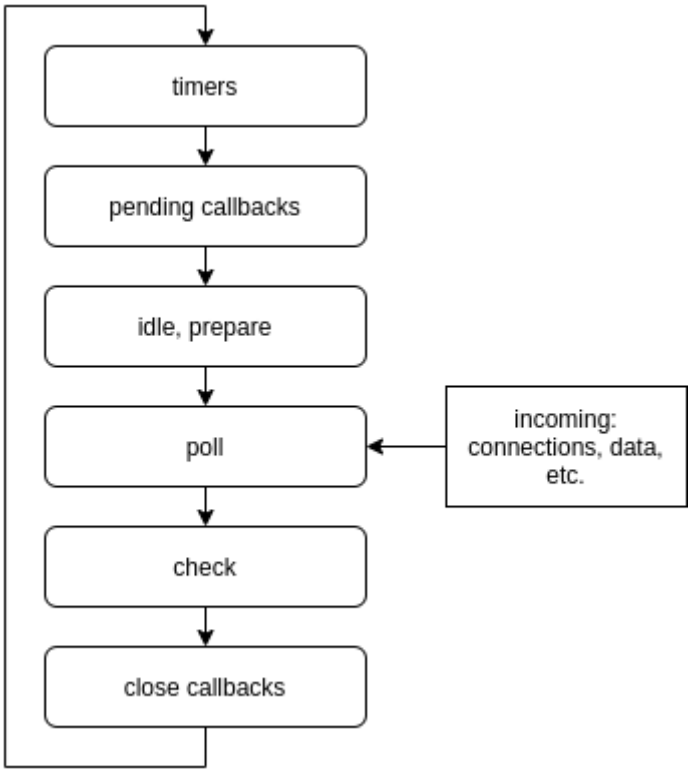

Рис. 3. Структура цикла событий

Программно структуру на рисунке выше реализует библиотека libuv, которую мы упоминали в прошлом уроке. Эта библиотека написана на C. Она позволяет Node.js работать с операционной системой. Учитывая, что все операционные системы устроены по-разному, библиотека libuv также выполняет задачу единого интерфейса для работы Node.js со всеми операционными системами.

Timers

На этой фазе цикла событий выполняются колбэки функций setTimeout и setInterval.

Функция setTimeout позволяет запускать функцию, переданную в качестве колбэка, по прошествии определённого временного промежутка.

```
const helloWorld = () => console.log('Hello World!');

setTimeout(helloWorld, 1000);
```

В примере выше функция helloWorld будет взята в работу по прошествии одной секунды, если для этого реализуется первая возможность. Что это значит?

Через одну секунду колбэк добавится в очередь колбэков для выполнения на фазе таймеров. Однако в это время цикл событий может быть занят выполнением других задач на стадиях опроса или проверки. В таком случае функция helloWorld станет ожидать завершения этих задач, что займёт некоторое время.

Функция setInterval выполняет практически то же самое, что и setTimeout, однако запуск функции колбэка — повторяющийся. Таким образом, в примере выше, где используется setTimeout, запись выведется в консоль только один раз. А в примере ниже, где используется setInterval, запись будет выводиться примерно каждую секунду.

```
const helloWorld = () => console.log('Hello World!');

setInterval(helloWorld, 1000);
```

Важно! *Если открыть исходный код этих двух функций, то можно увидеть, что разница между ними буквально в одной булевой переменной isRepeat. У функции setInterval эта переменная выставлена в true, а у setTimeout — в false. В остальном же реализация совпадает и заключается в вызове внутреннего таймера с соответствующими настройками.*

Pending callbacks

Эта фаза включает в себя выполнение колбэков для некоторых системных операций, например, ошибки TCP-сокетов, которые ожидает система, и некоторые другие.

Idle, prepare

Это фаза для внутренних целей и задач. Разработчик на Node.js никак на неё не влияет.

Poll

Фаза, на которой происходит получение новых событий ввода или вывода. Другими словами, выполняется написанный программистом код. Здесь может заблокироваться поток Node.js, если написанный код содержит какую-то тяжёлую операцию вычисления, либо вызывает синхронное чтение или запись большого файла.

Фаза Poll — самая сложная во всём цикле событий. В этом курсе мы рассмотрим упрощённую версию. Чтобы понять, как работает цикл событий во всех деталях, рекомендуется изучить исходный код библиотеки libuv.

Если в момент запуска фазы Poll в системе обнаруживаются запланированные таймеры, то для фазы, выполнения задач в ней, а также их ожидания будет выделено время.

Когда при запуске фазы Poll таймеры не запланированы:

1. Если в очереди задач фазы есть задачи, они будут выполняться последовательно в синхронном режиме, пока не закончится очередь, или не наступит предел количества обрабатываемых зараз колбэков (зависит от системы).
2. Если очередь пустая или закончилась, то происходит проверка на запланированные задачи setImmediate:
    a. Если такие задачи есть — цикл завершит фазу Poll и перейдёт к фазе Check.
    b. Если таких задач нет — цикл событий начнёт проверять таймеры. А если какие-то из них уже готовы — цикл событий перейдёт к фазе Timers.
    c. Если готовых таймеров нет — цикл будет ожидать новых колбэков в своей очереди, чтобы незамедлительно их выполнить.

Check

Здесь выполняются колбэки функции setImmediate. Если в фазе опроса нет задач в очереди, обычно она переходит в режим ожидания новых задач. А если колбэки, установленные функцией setImmediate, есть, то вместо ожидания новых задач в фазе Poll цикл событий переходит в фазу Check для выполнения таких колбэков.

Таким образом, колбэк из функции setImmediate будет выполняться всегда раньше колбэков из таймеров, если они запланированы в цикле ввода-вывода, а не в основном модуле.

Close callbacks

Колбэки событий close. Что такое «события», мы разберём чуть позже, но для понимания приведём пример: когда веб-сокет закрывает соединение, он сообщает об этом событием close. Обработчик любого события — колбэк события. Эта функция вызывается, когда событие «случается».

Важно! *Каждое выполнение всех 6 фаз — это один tick.*

Микрозадачи

Функция process.nextTick() технически не относится к циклу событий, так как это часть движка V8.

Для таких колбэков есть отдельная очередь — очередь микрозадач. Если в эту очередь попадает один или несколько колбэков, то после завершения текущей фазы сначала выполняются они, а затем продолжается работа цикла событий. К микрозадачам также относятся методы, предоставляемые Promise (then, catch, finally).

К макрозадачам относятся рассмотренные ранее методы setTimeout, setInterval, I/O-операции и т. д.

Важно! *С этой возможностью следует быть осторожным, так как рекурсивный вызов process.nextTick способен блокировать цикл событий и не давать ему достичь фазы опроса.*

Чтобы понять, как всё работает на практике, разберём задачу:

```
console.log('Record 1');

Promise.resolve().then(() => console.log('Record 2'));

setTimeout(() => {
  console.log('Record 3');
  Promise.resolve().then(() => {
    console.log('Record 4');
  });
});

Promise.resolve().then(() => {
  Promise.resolve().then(() => {
    console.log('Record 5');
  });
  console.log('Record 6');
});

console.log('Record 7');
```

Эта одна из множества типовых задач, которые часто встречаются на собеседованиях. Смысл задачи в том, чтобы правильно ответить, в каком порядке цифры будут выводиться в консоль.

При запуске этого скрипта интерпретатор JavaScript начинает обрабатывать строчку за строчкой. Представим, что мы интерпретатор:

1. Во время запуска программы в нашем цикле событий нет ни таймеров, чьи колбэки пора выполнять, ни каких-то других асинхронных операций, например, операций ввода-вывода. Поэтому цикл событий быстро проходит все первые фазы и оказывается на фазе Poll. В этой фазе начинает исполняться написанный нами код.
2. На строчке 1 располагается простая синхронная операция вывода данных в консоль. Она выполняется моментально, и в консоли появляется цифра «1».
3. На 3 строчке мы видим, что в нашем коде появляется микрозадача — разрешение промиса. Наш долг как интерпретатора — добавить вызов функции then() в очередь микрозадач. Именно добавить, а не выполнить, так как выполнение колбэков из очереди микрозадач происходит по завершении текущей фазы.
4. На 5 строке кода регистрируем таймер. Так как второго аргумента у таймера здесь нет, то его колбэк войдёт в очередь колбэков для фазы таймеров настолько быстро, насколько это будет технически возможно.

5. На 12 строке снова обнаруживается микрозадача, и её мы также кладём в очередь микрозадач. Одна там уже ожидает нас.
6. На 19 строке снова синхронный console.log — выполняем и выводим в консоль цифру «7»;
7. Так как 19 строка — последняя в нашей программе, цикл событий завершает фазу Poll. Однако, если в очереди микрозадач есть задачи, они будут выполнены, и только потом цикл событий продолжит работу. Сделаем это:
    a. Первый в очереди — промис, расположенный на 3 строке кода. Он разрешается, выполняется колбэк, и в консоли появляется цифра «2»;
    b. Затем настаёт очередь промиса с 12-й строчки кода. Он разрешается, однако в колбэке этого промиса в очередь микрозадач добавляется ещё одна функция.
    c. Потом выполняется синхронная команда вывода в консоль цифры «6».
    d. В очереди микрозадач остаётся последний промис, который также разрешается, и его колбэк выводит в консоль цифру «5».
8. На этом обработка задач из очереди микрозадач заканчивается, и цикл событий переходит на следующую фазу. Колбэки функции setImmediate в этом коде не запланированы, поэтому следующая фаза — фаза таймеров.
9. Здесь в очереди колбэков таймеров уже лежит колбэк нашего единственного таймера. Начинаем его выполнять:
    a. Синхронно выводим в консоль цифру «3».
    b. Добавляем в очередь микрозадач промис.
10. На этом фаза таймеров закончена, однако перед переходом к следующей фазе требуется выполнить все задачи из очереди микрозадач:
    a. Здесь разрешается промис, и в консоль выводится цифра «4».
11. Очередь микрозадач пуста. Цикл событий проходит через фазу колбэков ожидания, техническую фазу подготовки, и снова доходит до фазы Poll.
12. Однако весь код программы уже выполнен, цикл событий больше не ожидает ни таймеров, ни операций ввода-вывода, ничего, поэтому программа завершает свою работу.

Итого, в терминале мы видим результат выполнения этой программы:

```
Record 1
Record 7
Record 2
Record 6
Record 5
Record 3
Record 4
```

События в Node.js

В прошлом уроке мы упоминали, что большая часть Node.js основана на асинхронной событийно-ориентированной архитектуре. В этой части урока рассмотрим это подробнее, а также познакомимся со стандартным модулем Events. На базе этого модуля строятся разные виды инструментария — работа с HTTP-запросами, ответами, потоками, веб-сокетами и т. д. Этот модуль даёт возможность использовать события для каких-либо нужд программиста, например, при создании собственных модулей.

Событийно-ориентированная архитектура базируется на двух вещах — эмиттерах событий и обработчиках (слушателях) событий.

Эмиттеры событий (от англ. emit — излучать) — это специальные объекты, которые генерируют различные события.

Обработчики (слушатели) событий — это специальные функции, которые реагируют на факт генерации события в соответствии с заданными им инструкциями.

Эмиттеры событий выполняют две ключевые функции:

1. Генерирование событий.
2. Регистрация и дерегистрация обработчиков событий.

По своей сути генерирование какого-либо события — сигнал, что определённое условие соблюдается. Обычно это означает изменение состояния объекта-эмиттера события.

Стандартный модуль events предоставляет класс EventEmitter, который предназначен для работы с событиями в Node.js. Он хранит в себе все зарегистрированные имена событий, а также массивы функций-обработчиков всех этих событий. Все эмиттеры событий — экземпляры класса EventEmitter.

Модуль events и класс EventEmitter есть только в Node.js, так как представляют собой его части, соответственно, в браузере их нет.

Создание событий

Подключается модуль events так же, как и любой другой стандартный модуль:

```
const EventEmitter = require('events');
```

Представим, что наша задача — смоделировать электронную очередь посетителей в окна какого-нибудь МФЦ. В условиях такой задачи у нас есть следующие сущности:

1. Посетители, которые приходят друг за другом через какие-то промежутки времени.

2. Типы запросов, с которыми приходят посетители.

3. Система, которая обрабатывает запрос в зависимости от его типа.

Определимся, с какими типами запросов мы ожидаем посетителей. Для этого создадим массив объектов, описывающих различные типы запросов:

```
const requestTypes = [
  {
    type: 'send',
    payload: 'to send a document'
  },
  {
    type: 'receive',
    payload: 'to receive a document'
  },
  {
    type: 'sign',
    payload: 'to sign a document'
  }
];
```

У каждого объекта есть поле type — название типа запроса, а также поле payload — некоторое сообщение, характерное для посетителя с таким типом запроса.

Теперь требуется написать инструментарий генерирования нового посетителя. Первым шагом создадим класс Customer. У него будут свойства type и payload, соответствующие типу запроса каждого посетителя.

```
class Customer {
  constructor(params) {
    this.type = params.type;
    this.payload = params.payload;
  }
}
```

Далее опишем собственно процесс генерирования нового пользователя. Пусть это будет система, которая генерирует нового посетителя с интервалом, расположенным в диапазоне от 1 секунды до 5. А чтобы стало интереснее, этот интервал будет не фиксированным, а псевдослучайным. Для этого используем метод Math.random. Подробнее о генераторах псевдослучайных чисел — здесь.

Для начала напишем функцию, генерирующую псевдослучайное число в заданном диапазоне. При этом для удобства работы включим границы.

```
const generateIntInRange = (min, max) => {
  min = Math.ceil(min);
```

```
  max = Math.floor(max);

  return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

Подробнее о функции Math.random и её использовании — здесь.

Эту функцию мы используем как для генерации псевдослучайной задержки нового посетителя, так и для «случайного» выбора запроса, с которым он пришёл.

Для задержки при появлении нового посетителя нам пригодится функция delay. Она резолвит промис после истечения таймера.

```
const delay = (ms) => {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
};
```

Теперь напишем функцию, которая будет эмулировать приход нового посетителя с какой-то случайной задержкой.

```
const generateNewCustomer = () => {
  const intervalValue = generateIntInRange(1, 5) * 1000;
  const params = requestTypes[generateIntInRange(0, 2)];

  return delay(intervalValue).then(() => new Customer(params));
}
```

Сначала генерируем псевдослучайное значение задержки в секундах. Диапазон — от 1 до 5 секунд. Но так как функция setTimeout использует значение в миллисекундах, задержку потребуется умножить на 1000.

Аналогичным способом выбираем псевдослучайное значение в диапазоне от 0 до 2, включая границы. Этот диапазон соответствует разбросу индексов массива requestTypes, который содержит в себе 3 объекта.

Затем создаём новый объект посетителя, передавая в конструктор параметры его запроса. При этом объект посетителя появится только по истечении времени задержки (см. функцию delay).

Для создания новых посетителей у нас всё готово. Теперь нужно написать логику, которая будет обрабатывать посетителей в зависимости от их запросов. Это реализуется через конструкции if/else или switch, а также используя события. Последний вариант мы и выберем.

Сначала создадим класс Handler, который обработает запросы в зависимости от их типа.

```
class Handler {
  static send(payload) {
    console.log('Send request');
    console.log(`Customer need ${payload}`);
  }
  static receive(payload) {
    console.log('Receive request');
    console.log(`Customer need ${payload}`);
  }
  static sign(payload) {
    console.log('Sign request');
    console.log(`Customer need ${payload}`);
  }
}
```

У этого класса есть несколько статических методов, предназначенных для обработки различных запросов.

Здесь мы вплотную подошли к работе с событиями. Для генерации событий и создания их обработчиков используется специальный объект — эмиттер событий. Для него требуется создать класс-наследник класса EventEmitter, затем сделать его экземпляр:

```
class MyEmitter extends EventEmitter {};

const emitterObject = new MyEmitter();
```

Появление нового посетителя с запросом какого-то типа — новое событие. Это событие происходит тогда, когда посетитель заходит в помещение, подходит к терминалу и выбирает свой запрос. Далее система проверяет тип запроса и выполняет соответствующие инструкции — обработка события.

Логично, что перед тем как начать создавать события, требуется сделать их обработчики. В качестве названий событий мы используем названия соответствующих типов запросов. Для создания одного обработчика у объекта-эмиттера событий нужно вызвать метод on. Далее следует передать в него название события и функцию, которая будет вызываться каждый раз, когда генерируется соответствующее событие. В нашем случае для всех 4 типов запросов это будет выглядеть следующим образом:

```
emitterObject.on('send', Handler.send);

emitterObject.on('receive', Handler.receive);

emitterObject.on('sign', Handler.sign);
```

Разберём первую строчку. Она означает, что при появлении события send его будет обрабатывать функция send() класса Handler. То же самое и для остальных трёх типов событий.

Теперь мы готовы создать первого посетителя. Для этого создадим нового пользователя через функцию generateNewCustomer и сгенерируем событие:

```
generateNewCustomer().then(
  customer => emitterObject.emit(customer.type, customer.payload)
);
```

Для генерации события используем метод emit() объекта-эмиттера событий. Первым аргументом в него передадим название события. Второй аргумент (необязательный) — это данные, которые мы хотим передать обработчику события. Эти данные поступают в соответствующую функцию-обработчик события класса Handler и доступны там в переменной payload. Если потребуется, можно передавать не один, а несколько аргументов с данными. Они станут доступны в соответствующем обработчике в том же порядке, в каком и переданы.

Теперь если запустить программу, она создаст одного случайного посетителя со случайной задержкой и выведет в терминал что-нибудь вроде:

```
Send request
Customer need to send a document
```

Однако посетитель будет только один, а наша задача — смоделировать постоянный поток посетителей.

Это легко сделать, если создать специальную функцию для запуска программы и добавить в неё рекурсию — когда функция вызывает саму себя. Для этого заменим предыдущую строчку кода на следующий блок:

```
const run = async () => {
  const customer = await generateNewCustomer();
  emitterObject.emit(customer.type, customer.payload);

  run();
};

run();
```

Здесь для удобства написания кода и его красоты вместо конструкции промиса then применяется конструкция async/await. После первого запуска функции run и генерации нового посетителя эта

функция вызывает саму себя. Таким образом, создаётся модель бесконечного потока посетителей. В терминал будут выводиться данные по каждому посетителю:

```
Receive request
Customer need to receive a document
Send request
Customer need to send a document
Send request
Customer need to send a document
```

Важно! *Рекурсия — функция, которая вызывает саму себя. Это очень мощный и опасный инструмент. Он может привести к утечкам памяти, блокировке приложения и его падению. В этом примере рекурсия используется только для сохранения простоты примера и фокусирования на работе с событиями.*

Итак, мы смоделировали обработку постоянно приходящих посетителей МФЦ с различными запросами. Рассмотрим некоторые нюансы работы с событиями.

Порядок срабатывания обработчиков

Все обработчики событий срабатывают синхронно в том порядке, в котором зарегистрированы. Вызов метода on объекта-эмиттера событий называют регистрацией обработчика событий.

Допустим, посетителям, которые пришли с запросом отправки документа send, после обработки этого запроса требуется ещё оплатить отправку документа через кассу. Это реализуется несколькими способами. И один из них — добавить дополнительный обработчик:

```
emitterObject.on('send', Handler.send);
emitterObject.on('send', Handler.pay);
```

Важно не забыть добавить соответствующую функцию в класс Handler:

```
static pay() {
  console.log(`Customer needs to pay for the services`);
}
```

Теперь если придёт посетитель с запросом send, то в терминале появится следующий результат обработки такого запроса:

```
Send request
Customer needs to send a document
Customer needs to pay for the services
```

При этом обработчики будут срабатывать каждый раз, когда приходит посетитель с соответствующим запросом.

Класс EventEmitter также даёт возможность выполнить обработчик один раз, а потом автоматически снять его с регистрации. Для этого он предоставляет метод once, используемый вместо on:

```
emitterObject.on('send', Handler.send);
emitterObject.once('send', Handler.pay);
```

Тогда заплатить придётся только первому посетителю, который пришёл с запросом send.

Удаление обработчика

Зарегистрированный обработчик также удаляется вручную. Для этого класс EventEmitter предоставляет метод removeListener. Например, если отправка документов вдруг стала бесплатной, то потребуется убрать обработчик Handler.pay:

```
emitterObject.removeListener('send', Handler.pay);
```

Этот метод удаляет за один раз не более одного обработчика.

Важно! *Если событие уже сгенерировано, вызов метода removeListener не повлияет на обработчиков, пока последний из них не отработает. Для следующих событий указанный обработчик уже запускаться не будет.*

Максимальное количество обработчиков

По умолчанию при создании более 10 обработчиков для одного события класс EventEmitter покажет предупреждение. Это количество можно изменить для конкретного экземпляра класса посредством метода setMaxListeners. Если в него передать 0, то количество будет неограниченным:

```
emitterObject.setMaxListeners(0);
```

Обработка ошибок

Для обработки ошибок предусматривается специальное событие error. Оно, как и другие события, генерируется самостоятельно. Если для события error не будет ни одного обработчика, программа выбросит исключение, покажет stack trace и завершит работу.

Например, генерирование события без подключённого обработчика:

```
emitterObject.emit('error', new Error('Что-то пошло не так!'));
```

Завершит программу примерно с таким сообщением:

```
events.js:174
      throw er; // Unhandled 'error' event
      ^

Error: Что-то пошло не так!
    at Object.<anonymous> (/home/user/Documents/test.js:8:29)
    at Module._compile (internal/modules/cjs/loader.js:778:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:623:3)
Emitted 'error' event at:
    at Object.<anonymous> (/home/user/Documents/test.js:8:15)
    at Module._compile (internal/modules/cjs/loader.js:778:30)
    [... lines matching original stack trace ...]
    at bootstrapNodeJSCore (internal/bootstrap/node.js:623:3)
```

Важно! *Всегда обрабатываем события ошибок.*

Практическое задание

1. Решите задачу по выводу данных в консоль.

```
console.log('Record 1');

setTimeout(() => {
  console.log('Record 2');
  Promise.resolve().then(() => {
    setTimeout(() => {
      console.log('Record 3');
      Promise.resolve().then(() => {
        console.log('Record 4');
      });
    });
  });
});

console.log('Record 5');

Promise.resolve().then(() => Promise.resolve().then(() => console.log('Record
6')));
```

2. Напишите программу, которая будет принимать на вход несколько аргументов: дату и время в формате «час-день-месяц-год». Задача программы — создавать для каждого аргумента таймер с обратным отсчётом: посекундный вывод в терминал состояния таймеров (сколько осталось). По истечении какого-либо таймера, вместо сообщения о том, сколько осталось, требуется показать сообщение о завершении его работы. Важно, чтобы работа программы основывалась на событиях.

Глоссарий

Блокирующие операции ввода-вывода — операции ввода-вывода, блокирующие поток выполнения программы до тех пор, пока операция не закончится.

Макрозадачи — задачи вроде setTimeout, setInterval, операции ввода-вывода и т. д.

Микрозадачи — задачи вроде методов объектов Promise (then, catch, finally) или process.nextTick. Не относится к циклу событий, а принадлежит к движку V8. Выполняются в отдельной очереди задач, сразу после завершения текущей фазы цикла событий.

Многопоточный сервер создаёт новый поток для каждого входящего запроса от клиента.

Неблокирующие операции ввода-вывода — операции ввода-вывода, которые не блокируют поток выполнения программы из-за выгрузки таких операций в ядро системы.

Обработчик события — это специальная функция, которая реагирует на факт генерации события в соответствии с заданными в ней инструкциями.

Однопоточный сервер принимает все входящие запросы от клиентов в один поток.

Рекурсия — это функция, вызывающая саму себя.

Событийно-ориентированная архитектура — паттерн, лежащий в основе большей части Node.js-модулей. Заключается в активном использовании концепции событий — когда специальные объекты генерируют события, а специальные функции-обработчики реагируют на них, выполняя заданные инструкции.

Цикл событий — механизм на основе библиотеки libuv, который позволяет Node.js выполнять неблокирующие операции ввода-вывода, возвращающие по мере выполнения данные, через систему колбэков.

Эмиттер события (emit — излучать) — это специальный объект, генерирующий различные события.

Libuv — библиотека, написанная на C. Реализует цикл событий, выгрузку асинхронных операций ввода-вывода в систему и отвечает за кросс-платформенность для неблокирующей работы с операциями ввода-вывода.

Tick — выполнение каждой из 6 фаз цикла событий.

Дополнительные материалы

1. Официальная документация.
2. Исходный код библиотеки libuv.
3. Генератор псевдослучайных чисел.
4. Метод Math.random().

Используемые источники

1. Официальная документация.
2. Проблема C10k.
3. Документация по JavaScript от Mozilla.