GeekBrains

Курс по Node.js

Работа с файловой системой. Класс Buffer. Модуль Streams

[Node.js v14.x]

На этом уроке

1. Познакомимся с файловой системой.
2. Поговорим о кодировке.
3. Рассмотрим класс Buffer.
4. Узнаем, что такое чтение файла.
5. Выясним, что такое запись файла.
6. Узнаем, что такое поток.
7. Рассмотрим виды потоков.
8. Ознакомимся с чтением и записью файла через поток.
9. Поговорим о модуле Stream.

Оглавление

Теория урока

Файловая система

Файловая система — это порядок, который определяет способ организации, хранения и именования данных (файлов) на цифровых носителях информации.

Обычно с этим понятием сталкиваются при форматировании флешки или SD-карты, установке операционной системы или восстановлении данных.

Файловая система отвечает за всё, что касается работы с файлами — присвоение имён, интерфейс работы с файлами для приложений, хранение параметров файлов и другое.

Таким образом, когда мы работаем с файлами на своём компьютере, то взаимодействуем с файловой системой.

Приходится работать с файловыми системами и в процессе написания программ и приложений. Порой требуется считывать данные из какого-то файла, а также записывать или обрабатывать.

В Node.js для работы с файловой системой предназначен стандартный модуль fs. Но, прежде чем перейти к изучению взаимодействия с файлами в Node.js, познакомимся с кодировкой.

Что такое кодировка

Текст, содержащийся в файле, хранится там не так, как мы его привыкли видеть. В файле нет кириллических букв (или любых других), цифр, знаков и т. д. На самом деле текст в файле закодирован.

Вообще, вся информация в компьютере (любой файл) хранится в виде последовательности единиц и нулей. Причина этого кроется в физической реализации устройств, хранящих и обрабатывающих информацию. Тонкости физики работы этих устройств мы не будем разбирать в рамках этого курса. Однако ознакомиться с этим можно в статье на «Википедии».

Как понять, какой набор единиц и нулей соответствует, например, букве «б»? Ведь это не число, которое можно перевести из его системы счисления в двоичную. Этот аспект и определяет кодировка.

Кодировка — числовой код, который присваивается каждому символу (цифре, букве, знаку). Эти числовые коды хранятся в специальных кодовых таблицах. При открытии файла в каком-либо редакторе происходит декодирование, и на экране появляется соответствующий текст.

Этим объясняются, например, различные проблемы при открытии неизвестных файлов. Когда редактор не поддерживает кодировку файла, или в настройках редактора установлена по умолчанию неправильная кодировка — пользователь вместо текста на экране видит набор непонятных символов.

Однако требуется выбрать правильную кодировку в настройках, чтобы непонятные символы превратились в текст.

Сегодня наиболее распространена кодировка UTF-8. Она поддерживается практически всеми редакторами. Если у нас есть файл, и мы не знаем, какая у него кодировка — сначала используем UTF-8.

Подробнее о кодировках — в дополнительных материалах в конце урока.

Класс Buffer

Перед тем как перейти к работе с файлами, поговорим о классе Buffer.

Это глобальный класс, позволяющий работать с потоками двоичных данных. Он доступен без подключения соответствующего модуля.

Компьютер умеет работать только с 0 и 1, любые типы данных представляются в этом виде.

О потоках мы поговорим чуть позже. Здесь только кратко упомянем, что это некая последовательность данных, которые перемещаются из одного места в другое. Главная идея потоков — возможность работать с данными «порционно», оптимизируя, таким образом, взаимодействие с памятью.

Работа с потоками двоичных данных, для которой предназначается класс Buffer, заключается в обработке перемещения данных файлов, например, при их чтении для дальнейшей обработки и передачи.

Что именно делает Buffer?

Важно вспомнить, что скорость обработки данных компьютером конечна. Соответственно, есть какой-то минимальный и максимальный набор данных, которые он может обработать за некий промежуток времени. Если в процессе перемещения данные быстрее поступают, чем обрабатываются — данные, которые ещё не попали в обработку будут где-то ждать своей очереди. И наоборот, если данные поступают медленнее, чем обрабатываются — поступившие данные должны где-то ожидать дополнительных данных, до того как система их обработает. Место, где данные ожидают обработки, и называется буфером.

Физически буфер чаще всего представляет собой некий участок оперативной памяти, где накапливаются данные, ожидающие своей очереди.

Проведём аналогию с залом ожидания аэропорта — пассажиры туда прибывают, а оттуда убывают. Там они ожидают рейса или ждут пересадки.

Доступ к этим данным и предоставляет класс Buffer. Он позволяет их считывать и обрабатывать. Этот класс также позволяет создавать собственные буфера и определять их характеристики, например, размер в байтах. В следующем разделе мы увидим, как это выглядит на практике.

Работа с файлами

Теперь после краткого экскурса в кодировку текста в файлах, а также в класс Buffer перейдём к работе с ними. Для этого в Node.js есть стандартный модуль fs. Он позволяет работать с файловой системой — создавать и удалять директории и файлы, читать файлы, изменять их. Особенность модуля в том, что многие из операций имеют два варианта выполнения — синхронный и асинхронный. У синхронных методов в названии есть слово Sync. Асинхронные варианты функций после выполнения операции вызывают callback.

Модуль fs подключается в файл через уже знакомую нам функцию require.

```
const fs = require('fs');
```

Чтение файла

Допустим, нам надо прочитать файл логов с сервера. Файл назовём access.log и запишем в него следующие данные:

```
127.0.0.1 - - [30/Jan/2021:11:10:15 -0300] "GET /sitemap.xml HTTP/1.1" 200 0 "-"
"curl/7.47.0"
```

В модуле fs для чтения файла есть два метода:

1. readFile(path[, options], callback) — асинхронное чтение файла. По завершении чтения вызывается переданный в метод callback, где и будут доступны считанные данные.
2. readFileSync(path[, options]) — синхронное чтение файла. Возвращает данные, полученные при чтении. Могут быть представлены либо строкой либо специальным классом Buffer, о нём поговорим немного позже.

Рассмотрим параметры, которые передаются в эти методы.

1. path — путь до файла.
2. options — объект или строка. Включает в себя следующие параметры:
    a. encoding — кодировка. Значение по умолчанию — null.
    b. flag — флаг файловой системы. Флаг задаёт режим работы с файлом. Например, можно создать файл, если его нет, а также открыть файл только для чтения или для чтения и записи. Значение по умолчанию — ‘r’ — открыть файл для чтения с выбросом исключения, если файла нет. Полный список флагов — по ссылке в официальной документации Node.js.

    c. signal — метод специального класса AbortSignal. Позволяет прерывать чтение файла. Используется только в асинхронной функции чтения.
3. callback — функция, которая вызывается после завершения асинхронной функцией процесса по чтению файла. Принимает в себя два аргумента:
    a. err — объект ошибки, если она случилась.
    b. data — данные, полученные при чтении файла. Обычно представляются строкой или специальным классом Buffer.

Чем отличается синхронное чтение файла от асинхронного? Для этого обратимся к материалам прошлого урока. Во время асинхронного чтения программа выполняет другие задачи, тогда как при синхронном чтении она будет ожидать завершения операции прежде, чем пойти дальше. Это стоит учитывать, особенно при чтении больших файлов, так как это прямо отражается на производительности программы.

Например, если пользователь запрашивает с сервера большой файл, то чтение его синхронным способом заблокирует основной поток, и сервер не выполнит никаких функций, пока не прочитает этот большой файл. Если же читать его асинхронным способом, то в процессе ожидания, пока завершится чтение, сервер выполнит другие функции, например, обработает другие запросы.

Асинхронное чтение файла

Для начала прочитаем наш файл асинхронным методом. Сам модуль fs мы уже подключали выше, теперь осталось только вызвать соответствующий метод.

```
fs.readFile('./access.log', (err, data) => console.log(data));
```

После запуска такого скрипта в терминале появится следующее:

```
<Buffer 31 32 37 2e 30 2e 30 2e 31 20 2d 20 2d 20 5b 33 30 2f 4a 75 6e 2f 32 30
31 36 3a 31 34 3a 31 30 3a 31 35 20 2d 30 34 30 30 5d 20 22 47 45 54 20 2f 65
... 43 more bytes>
```

Это и есть данные, представленные классом Buffer. Заметим, что содержимое каждого байта представлено числом в шестнадцатеричной системе счисления. Подробнее об этой системе — здесь.

Преобразовываются эти двоичные данные в нормальный текст двумя способами.

Первый способ — вызвать метод toString у полученного объекта, который представлен классом Buffer, и указать кодировки:

```
fs.readFile('./access.log', (err, data) => console.log(data.toString('utf8')));
```

В результате выполнения такой функции в терминале появится то содержимое файла, которые мы туда записали:

```
127.0.0.1 - - [30/Jan/2021:11:10:15 -0300] "GET /sitemap.xml HTTP/1.1" 200 0 "-"
"curl/7.47.0"
```

Второй способ предоставлен методом чтения файла readFile — передача через options названия кодировки, которая используется для преобразования двоичных файлов:

```
fs.readFile('./access.log','utf8', (err, data) => console.log(data));
```

Результат в терминале будет аналогичен прошлому.

Синхронное чтение файла

В общем случае предпочтительно использовать асинхронное чтение файла, так как оно не блокирует поток программы. Однако если по какой-то причине программисту понадобится использовать синхронное чтение, то модуль fs предоставит такую возможность посредством метода readFileSync. Например, если мы пишем для себя какую-то небольшую консольную утилиту для работы с файлами, то можно воспользоваться простотой метода синхронного чтения.

```
const data = fs.readFileSync('./access.log');

console.log(data);
```

При запуске такого кода в терминале появится уже знакомый нам формат данных, представленный классом Buffer. Аналогично прошлому примеру, мы можем использовать кодировку, название которой передаём в метод readFileSync.

```
const data = fs.readFileSync('./access.log', 'utf8');
```

Мы научились считывать данные из файла логов, теперь запишем туда дополнительные логи.

Запись файла

Допустим, наш сервер принял ещё два запроса и ему требуется записать их в файл логов. Один запрос будет на запись данных:

```
127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0"
```

И ещё один — на получение:

```
127.0.0.1 - - [30/Jan/2021:11:11:25 -0300] "GET /boo HTTP/1.1" 404 0 "-"
"curl/7.47.0"
```

Как и в случае чтения файлов, модуль fs предоставляет для записи асинхронный и синхронный инструменты.

1. writeFile(file, data[, options], callback) — асинхронная запись файла. По завершении записи вызывается переданный в метод коллбэк, где будет доступен объект ошибки, если она произошла в процессе записи файла.
2. writeFileSync(file, data[, options]) — синхронная запись файла. Возвращает undefined.

Параметры, передающиеся в эти методы:

1. file — имя файла, которое также включает и путь.
2. data — данные для записи. Обычно представляются в формате строки, Buffer, объекта и специальных типов TypedArray и DataView. Прочитать об этих типах можно здесь и здесь.
3. Options — объект или строка. Как правило, включает в себя следующие параметры:
    a. encoding — кодировка. Значение по умолчанию — 'utf8'.
    b. mode — этот параметр позволяет устанавливать специальные свойства файла, например, только для чтения. Значение по умолчанию — ‘0o666’ — файл доступен для чтения и записи любым пользователям.
    c. flag — флаг файловой системы. Разные флаги используются, чтобы создать файл, которого ещё нет, открыть файл только для чтения или для чтения и записи. Значение по умолчанию — ‘w’ — открыть файл для записи. Если файла нет, то он создастся, а, наоборот — перезапишется.
    d. signal — метод специального класса AbortSignal. Позволяет прерывать запись файла. Используется только в асинхронной функции записи.
4. callback — функция, которая вызывается после завершения асинхронной функцией процесса записи файла. Принимает в себя только один аргумент:
    a. err — объект ошибки, если она случилась в процессе записи файла.

Протестируем эти методы на практике.

Асинхронная запись файла

Для начала создадим константы, которые будут хранить логи наших запросов.

```
const log1 = '127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1"
200 0 "-" "curl/7.47.0"';
const log2 = '127.0.0.1 - - [30/Jan/2021:11:11:25 -0300] "GET /boo HTTP/1.1" 404
0 "-" "curl/7.47.0"';
```

Запишем лог первого запроса в файл, который читали ранее:

```
fs.writeFile('./access.log', log1, (err) => console.log(err));
```

Если всё прошло успешно, то в терминале появится следующее:

```
null
```

Теперь откроем файл access.log и посмотрим на его содержимое:

```
127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0"
```

Мы видим, что содержимое файла полностью перезаписалось. Но ведь если теперь мы попытаемся записать лог следующего запроса, он «перезатрёт» текущий. Таким образом, в файле всегда будет только один лог, что не очень логично. Ведь смысл файла с логами — хранить все логи приложения, чтобы понимать происходящее.

Как вариант, можно считать данные из файла, добавить к ним новые логи и снова записать в файл. И так каждый раз. Это работает на одной строчке, на двух, но на больших объёмах данных это работать не будет. С каждой новой записью количество данных будет увеличиваться, и, соответственно, время на операции чтения и записи тоже.

Чтобы добавить данные в файл вместо перезаписи, в options надо передать параметр flag = ‘a’. Узнать, какие бывают флаги и их предназначения, можно в официальной документации.

```
fs.writeFile('./access.log', log1, { flag: 'a' }, (err) => console.log(err));
```

Результат в файле будет следующим:

```
127.0.0.1 - - [30/Jan/2021:11:10:15 -0300] "GET /sitemap.xml HTTP/1.1" 200 0 "-"
"curl/7.47.0"127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200
0 "-" "curl/7.47.0"
```

Ура! Мы дописали данные в конец файла, и это успех. Осталось научиться добавлять данные так, чтобы их удобно было читать. В нашем файле мы видим, что новая строчка записалась вплотную к старой. Так произошло, потому что функция дозаписала данные в конец файла. Чтобы этого избежать, надо между двумя записями вставить символ переноса строки.

В общем случае символ переноса строки — ‘\n’. Однако, если мы работаем в Windows, наша программа для чтения текстовых файлов может не воспринять такой перенос строки. Это связано с особенностями работы Windows. В таком случае для корректной работы к символу переноса строки требуется добавить символ возврата каретки — ‘\r’.

Теперь, когда мы знаем, что делать, выберем вариант реализации. В нашей задаче есть два варианта добавления символа, относящиеся к переносу строки:

1. Добавить его в начало записываемого лога и всё вместе записать в файл.
2. Разделить операции и записать перенос строки отдельной командой.

Для наглядности выберем вариант №2.

```
fs.writeFile('./access.log', '\n', { flag: 'a' }, (err) => console.log(err));
fs.writeFile('./access.log', log1, { flag: 'a' }, (err) => console.log(err));
```

Результат в файле будет следующим:

```
127.0.0.1 - - [30/Jan/2021:11:10:15 -0300] "GET /sitemap.xml HTTP/1.1" 200 0 "-"
"curl/7.47.0"127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200
0 "-" "curl/7.47.0"
127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0"
```

Синхронная запись файла

В некоторых случаях требуется синхронная запись. Например, надо убедиться, что данные записались, и только после этого выполнить какие-то следующие инструкции.

Синхронная запись осуществляется методом writeFileSync. Запишем второй лог в файл:

```
fs.writeFileSync('./access.log', log2);
```

Проверяем файл и видим, что и этот метод перезаписывает файл полностью. Как и в случае асинхронной записи, всё решается выставлением специального флага.

Модуль fs предоставляет специальные методы для дополнения файлов, вместо перезаписывания. В некоторых случаях их использование более предпочтительно, так как упрощает читаемость кода.

Есть два метода — асинхронный и синхронный:

1. appendFile(path, data[, options], callback) — асинхронное дополнение данных в файле. Если файла нет — он создаётся. По завершении записи вызывается переданный в метод коллбэк, где будет доступен объект ошибки, если она произошла в процессе записи файла.
2. appendFileSync(path, data[, options]) — синхронное дополнение данных в файле. Как и в асинхронной версии, файл создаётся, если его нет.

Эти методы поддерживают те же параметры, что и writeFile/writeFileSync.

Потоки

Коротко о потоках мы упоминали в начале урока. Главная идея потоков — возможность работать порционно с данными, которые перемещаются из одного места в другое.

В Node.js потоки — это абстрактный интерфейс для работы с такими данными. Его «абстрактность» заключается в предоставлении единого интерфейса для работы с данными в оперативной памяти, на диске и с данными из сети. Этот интерфейс позволяет обрабатывать данные небольшими порциями, не загружая полного объёма в оперативную память сразу.

Потоки используются во многих встроенных Node.js-модулях:

- http — для работы с обработкой запросов и ответов;
- fs — чтение или запись файлов;
- zlib — преобразование данных;
- в модулях шифрования;
- и других.

Без использования потоков практически невозможно обрабатывать большие файлы.

Это крайне полезная возможность. Если рассматривать наш пример с логами запросов к серверу, то обычно такие файлы имеют очень большой объём. Однако логи собираются на сервере для дальнейшего анализа и расследования инцидентов, если они случаются. А перед этим информацию из этих файлов требуется считать в память.

Стандартный объём памяти, выделяемый для Node.js-приложения:

- 700 Мб — для 32-битных систем;
- 1400 Мб — для 64-битных систем.

Таким образом, если мы разом прочитаем, синхронно или асинхронно, файл объёмом 2 Гб, то при стандартных настройках, на любой машине наша программа упадёт с ошибкой JavaScript heap out of memory. В подобных задачах и используются потоки, которые позволяют обрабатывать такие данные порционно.

В Node.js есть всего 4 вида потоков:

1. Readable — поток на чтение.

2. Writable — поток на запись.
3. Duplex — поток на чтение и запись.
4. Transform — подвид Duplex-потока, который изменяет данные.

Для работы с потоками в Node.js есть стандартный модуль streams. Каждый вид потока в Node.js представлен соответствующим классом в модуле streams.

Вид потока	Класс
Readable	stream.Readable
Writable	stream.Writable
Duplex	stream.Duplex
Transform	stream.Transform

Примечательно, что все потоки — экземпляры класса EventEmitter, о котором мы говорили в прошлый раз. Благодаря этому, у каждого потока есть свой набор событий, и обработка этих событий позволяет нам работать с данными.

Поток на чтение

Как было сказано выше, инструментарий потоков используется также в модуле fs для чтения и записи данных в файл. Использование потоков вместо прямых методов чтения или записи позволяет:

- более гибко подходить к решению связанных с этим задач;
- управлять использованием памяти;
- записывать и считывать большие объёмы данных, не теряя в производительности.

Для чтения данных из файла, используя потоки, в модуле fs есть класс fs.ReadStream и метод fs.createReadStream. Причём метод fs.createReadStream — это функция-обёртка для создания экземпляра класса fs.ReadStream. В документации так и написано: Returns: <fs.ReadStream>. Убедиться в этом можно, заглянув в исходный код:

```
fs.createReadStream = function(path, options) {
  return new ReadStream(path, options);
};
```

Таким образом, поток для чтения файла создаётся двумя способами:

1. Создать вручную экземпляр класса fs.ReadStream.
2. Вызвать функцию fs.createReadStream, которая сделает это за нас.

Разберём, какие параметры требуется передать в конструктор.

1. path — путь к файлу, который надо прочитать.
2. options — необязательный параметр. Если он передаётся, то может быть строкой или объектом. Строкой указывается только кодировка. Если же передавать объект, то подключаются следующие параметры:
    a. flags — знакомые по методам чтения или записи флаги для работы с файловой системой. Значение по умолчанию — ‘r’. Это означает открыть файл только для чтения, бросить исключение, если файла нет.
    b. encoding — кодировка содержимого файла. По умолчанию — null.
    c. fd — файловый дескриптор (неотрицательное целое число) или экземпляр класса-обёртки над файловым дескриптором FileHandle. Его отличие от файлового дескриптора в том, что он предоставляет API для работы с файлом. По умолчанию — null. Здесь мы не рассматриваем понятие «файловый дескриптор», но подробнее об этом — здесь.
    d. mode — этот параметр позволяет устанавливать специальные свойства файла, например, разрешение только для чтения. Значение по умолчанию — ‘0o666’ — файл доступен для чтения и записи любым пользователем.
    e. autoClose — это флаг. Если он выставлен в false, то при ошибке или событии ‘end’ дескриптор файла не закроется, и это станет обязанностью приложения. По умолчанию — true.
    f. emitClose — флаг, отвечающий за генерирование события ‘close’ при закрытии потока.
    g. start — неотрицательное целое число, обозначающее байт, с которого надо начать чтение включительно. Если он не определён, чтение станет происходить с текущей позиции файла.
    h. end — неотрицательное целое число, обозначающее байт, до которого нужно читать файл включительно. По умолчанию — Infinity.
    i. highWaterMark — параметр, влияющий на размер одной «порции» данных в потоке. По умолчанию — 64 Кб (64 * 1024).
    j. fs — объект, в котором передаются методы ‘open’, ‘read’ и ‘close’ для переопределения текущих методов ‘open’, ‘read’ и ‘close’. По умолчанию — null.

Посмотрим, как это работает в минимальной конфигурации. Создадим поток на чтение файла логов.

```
const readStream = fs.createReadStream('./access.log');
```

При создании объекта потока он подключается к источнику данных (к файлу ‘access.log’) и начинает считывать данные из него. Как только первая порция данных готова — генерируется событие ‘data’, и в него передаются данные. Таким образом, чтобы взаимодействовать с данными, требуется создать обработчик этого события.

```
readStream.on('data', (chunk) => {
```

```
      console.log('Chunk');
      console.log(chunk);
});
```

В этом обработчике выводятся считанные данные в консоль. Чтобы наглядно показать те самые «порции» данных, мы добавили console.log со словом ‘chunk’ (с англ. — часть).

При запуске программы появится сообщение в терминале:

```
Chunk
<Buffer 31 32 37 2e 30 2e 30 2e 31 20 2d 20 2d 20 5b 33 30 2f 4a 61 6e 2f 32 30
32 31 3a 31 31 3a 31 30 3a 31 35 20 2d 30 33 30 30 5d 20 22 47 45 54 20 2f 73
... 118 more bytes>
```

Мы видим только один chunk, потому что в файле слишком мало данных, и все они считываются одной «порцией». Чтобы увидеть несколько «порций», увеличим размер файла благодаря добавлению новых логов, относящихся к запросам.

После добавления всех требуемых записей ещё раз запустим программу. Представим, что сервер, чьи логи мы пишем, работает уже давно.

В терминале появится следующее:

```
Chunk
<Buffer 31 22 20 32 30 30 20 30 20 22 2d 22 0a 31 32 37 2e 30 2e 30 2e 31 20 2d
20 2d 20 5b 33 30 2f 4a 61 6e 2f 32 30 32 31 3a 31 31 3a 31 31 3a 32 30 20 2d
... 65486 more bytes>
Chunk
<Buffer 3a 31 35 20 2d 30 33 30 30 5d 20 22 47 45 54 20 2f 73 69 74 65 6d 61 70
2e 78 6d 6c 20 48 54 54 50 2f 31 2e 31 22 20 32 30 30 20 30 20 22 2d 22 0a 31
... 65486 more bytes>
Chunk
<Buffer 2e 30 22 0a 31 32 37 2e 30 2e 30 2e 31 20 2d 20 2d 20 5b 33 30 2f 4a 61
6e 2f 32 30 32 31 3a 31 31 3a 31 30 3a 31 35 20 2d 30 33 30 30 5d 20 22 47 45
... 10433 more bytes>
```

Теперь мы видим, что поток читает данные порционно.

Данные снова представлены классом Buffer. Чтобы превратить их в человеческий текст, достаточно указать кодировку при создании объекта потока на чтение.

```
const readStream = fs.createReadStream('./access.log', 'utf8');
```

Если чтение источника данных закончилось, то генерируется событие ‘end’. Его также можно обработать:

```
readStream.on('end', () => console.log('File reading finished'));
```

Если в процессе чтения файла произошла ошибка, то поток генерирует событие ошибки. В Node.js хорошей практикой считается обработка ошибок. Обработаем и мы.

```
readStream.on('error', () => console.log(err));
```

Итоговый код:

```
const fs = require('fs');

const readStream = fs.createReadStream('./access.log', 'utf8');

readStream.on('data', (chunk) => {
      console.log('Chunk');
      console.log(chunk);
});

readStream.on('end', () => console.log('File reading finished'));
readStream.on('error', () => console.log(err));
```

В этом примере мы написали программу, считывающую данные из файлов любого объёма. Теперь пора научится эти данные записывать.

Поток на запись

Аналогично потоку на чтение данных, функциональность потока на запись в модуле fs представлена классом fs.WriteStream и методом fs.createWriteStream. Метод fs.createWriteStream также считается функцией-обёрткой для создания экземпляра класса fs.WriteStream. Чтобы убедиться в этом, заглянем в [исходный код]:

```
fs.createWriteStream = function(path, options) {
  return new WriteStream(path, options);
};
```

Параметры, которые передаются в конструктор, в целом аналогичны параметрам при создании потока на чтение:

1. path — путь к файлу, в который требуется записать данные.

2. options — необязательный параметр. Если передаётся — может быть строкой или объектом. Строкой указывается только кодировка. Если же передавать объект, то подключаются следующие параметры:
    a. flags — флаги для работы с файловой системой. Значение по умолчанию — ‘w’. Означает открыть файл для записи. Если файла нет — он будет создан, а, наоборот — перезаписан.
    b. encoding — кодировка содержимого файла. По умолчанию — ‘utf8’.
    c. fd — файловый дескриптор (неотрицательное целое число) или экземпляр класса-обёртки над файловым дескриптором FileHandle. Его отличие от файлового дескриптора в том, что он предоставляет API для работы с файлом. По умолчанию — null. Здесь мы не рассматриваем понятие «файловый дескриптор», но подробнее об этом — здесь.
    d. mode — этот параметр позволяет устанавливать специальные свойства файла, например, разрешение только для чтения. Значение по умолчанию — ‘0o666’. То есть файл доступен для чтения и записи любым пользователям.
    e. autoClose — это флаг. Если выставлен в false, то при ошибке или событии ‘end’ дескриптор файла не закроется, и это станет обязанностью приложения. По умолчанию — true.
    f. emitClose — флаг, отвечающий за генерирование события ‘close’ при закрытии потока.
    g. start — неотрицательное целое число, обозначающее байт, с которого требуется начать чтение включительно. Если он не определён — чтение будет происходить с текущей позиции файла.
    h. fs — объект, в котором передаются методы ‘open’, ‘write’, ‘writev’ и ‘close’ для переопределения текущих методов ‘open’, ‘read’ и ‘close’. По умолчанию — null.

Добавим лог ещё одного запроса к серверу в файл, используя поток на запись.

Сначала создадим объект потока на запись, сразу указав кодировку:

```
const writeStream = fs.createWriteStream('./access.log', 'utf8');
```

Чтобы записать данные, используем метод write потока на запись:

```
writeStream.write(log1);
```

Наш файл снова перезаписался, и в нём опять только одна запись. Вспомним о флаге, который позволяет добавлять данные в файл, а не перезаписывать его полностью. Сразу добавим перенос строки.

```
const writeStream = fs.createWriteStream('./access.log',  { flags: 'a',
```

```
encoding: 'utf8' });

writeStream.write('\n');
writeStream.write(log1);
```

Теперь результат соответствует ожиданиям:

```
127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0"
127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0"
```

После окончания записи данных в файл закрываем поток:

```
writeStream.end(() => console.log('File writing finished'));
```

Итоговый код:

```
const fs = require('fs');

const log1 = '127.0.0.1 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1"
200 0 "-" "curl/7.47.0"';

const writeStream = fs.createWriteStream('./access.log',  { flags: 'a',
encoding: 'utf8' });

writeStream.write('\n');
writeStream.write(log1);

writeStream.end(() => console.log('File writing finished'));
```

Поток на чтение и запись

До этого момента мы работали с потоками на чтение и запись, которые реализованы через стандартный модуль для работы с файловой системой fs.

Однако в Node.js есть отдельный модуль для работы с потоками — stream.

Duplex-поток реализует интерфейсы потока на чтение и потока на запись. Такие потоки используются:

- в TCP-сокетах — транспортный протокол передачи данных по сети;
- в потоках сжатия данных, шифрования данных и т. д.

Мы рассмотрим один из видов потока на чтение и запись (и самый распространённый) — поток Transform. Он предназначен для преобразования порции считанных или полученных данных и отправки их дальше по цепочке.

Допустим, в одном файле мы храним неизменные, необработанные логи в том виде, в котором их получаем. Но эти данные требуются нам и в обработанном виде. Для решения такой задачи вполне подойдёт поток Transform.

Разобьём задачу на шаги:

1. Считать данные из файла.
2. Преобразовать данные.
3. Вывести данные в терминал.

Соединить все три шага мы можем посредством метода pipe(), который доступен в объекте любого потока. Этот метод предназначен для создания цепочек потоков и передачи по ней данных.

Читать данные из файла, используя потоки, мы уже умеем.

```
const fs = require('fs');
const readStream = new fs.ReadStream('./access.log', 'utf8');
```

Теперь создадим и настроим поток Transform.

```
const { Transform } = require('stream');

const transformStream = new Transform({
   transform(chunk, encoding, callback) {
       const transformedChunk = chunk.toString().replace(/127.0.0.1/g, '');

       this.push(transformedChunk);

       callback();
   }
});
```

Здесь мы создаём экземпляр класса Transfrom и передаём в конструктор метод transform. Этот метод и будет выполнять основную работу — преобразовывать данные и отправлять их дальше.

Допустим, нам надо удалить из логов любое упоминание об localhost. Делаем это посредством метода replace и простого регулярного выражения.

Вызовом метода this.push мы фиксируем изменения, а затем через вызов callback отправляем данные на выход потока.

Функция callback также фиксирует изменения и сразу отправляет данные на выход. Для этого надо передать в неё эти данные вторым аргументом. Первым аргументом она принимает объект ошибки.

```
callback(null, transformedChunk);
```

Однако если фиксировать изменения методом this.push, то у нас появится возможность делать это несколько раз перед отправкой данных на выход. Это позволит, например, продублировать какую-то часть данных или изменять их поэтапно.

Теперь осталось вывести изменённые данные в терминал. Для этого используем вышеупомянутый метод pipe(). Соединяем поток на чтение с трансформирующим потоком, затем отправляем данные в поток на вывод process.stdout:

```
readStream.pipe(transformStream).pipe(process.stdout);
```

Здесь появляется незнакомая нам сущность — process.stdout. Это интерфейс стандартного модуля process для работы с потоком вывода stdout. В нашем случае, отправляя данные в такой поток, мы выводим их в терминал. Кроме стандартного потока вывода, есть также стандартный поток ввода stdin и стандартный поток вывода ошибок stderr. Подробнее о потоках — в этой статье.

Итоговый код:

```
const fs = require('fs');

const readStream = new fs.ReadStream('./access.log', 'utf8');

const { Transform } = require('stream');

const transformStream = new Transform({
   transform(chunk, encoding, callback) {
       const transformedChunk = chunk.toString().replace(/127.0.0.1/g, '');

       this.push(transformedChunk);

       callback();
   }
});

readStream.pipe(transformStream).pipe(process.stdout);
```

Итак, в этом уроке мы рассмотрели много важных понятий, касающихся не только разработки на Node.js, но и веб-разработки в целом. Работа с файлами, кодировки, двоичные данные, потоки — это лишь малая часть того интересного мира, который мы изучаем. Важно тренироваться и

экспериментировать с пройденным материалом, пытаться адаптировать предоставленные знания под конкретные задачи. Только закрепляя теоретические знания на практике, можно сделать их устойчивой и неотъемлемой частью профессионализма.

Практическое задание

По ссылке вы найдёте файл с логами запросов к серверу весом более 2 Гб. Напишите программу, которая находит в этом файле все записи с ip-адресами 89.123.1.41 и 34.48.240.111, а также сохраняет их в отдельные файлы с названием %ip-адрес%_requests.log.

Глоссарий

Буфер — участок памяти, в котором данные хранятся в ожидании обработки.

Класс Buffer — это глобальный класс, позволяющий работать с потоками двоичных данных.

Кодировка — это числовой код, который присваивается каждому символу (цифре, букве, знаку).

Потоки — это абстракция для «порционной» работы с данными, которые перемещаются из одного места в другое.

Файловый дескриптор — это неотрицательное целое число, которое возвращает ядро системы процессу при создании нового потока ввода-вывода.

Файловая система — это порядок, который определяет способ организации, хранения и именования данных на цифровых носителях информации.

Дополнительные материалы

1. Официальная документация.
2. Статья «Компьютерная память».
3. Статья «О кодировках и кодовых страницах».
4. Статья «И снова о Unicode».
5. Класс Buffer в официальной документации.
6. Флаги для работы с файловой системой.
7. Статья «Что такое файловый дескриптор простыми словами».
8. Статья «Стандартные потоки ввода-вывода».

Используемые источники

1. Официальная документация.
2. Статья «И снова о Unicode».
3. Статья Do you want a better understanding of Buffer in Node.js? Check this out.