Прототипы

Курс: JavaScript, часть 2: прототипы и асинхронность

18 февраля 2018 г.

Оглавление

Глава 4

Node.js

4.1. Блокирующий ввод/вывод

Любая программа потребляет те или иные виды ресурсов в том или ином объеме. Например, если ваша программа выполняет какие-то сложные вычисления, она в значительной степени потребляет время центрального процессора. Возможно, она хочет хранить какие-то промежуточные вычисления в оперативной памяти, таким образом потребляя и этот ресурс. А, может быть, для вычислений необходимо считать какой-то массив данных с диска или, например, с удаленного хранилища. В этом случае она обращается к системам ввода-вывода для чтения с локального диска или для обращения по сети.
Например, вычисления числа Фибоначчи. Эта операция, в основном, потребляет время центрального процессора. Мы можем сказать, что она ограничена производительностью этого процессора, и если мы поставим процессор мощнее, мы сможем вычислить это число быстрее.
Второй пример: подсчет количества строк в файле. Здесь уже не так много вычислений, нужно лишь увеличивать счетчик. И большую часть времени данная операция будет потреблять ресурсы систем ввода-вывода, ведь файл необходимо вначале прочитать, прежде чем подсчитать количество строк в нем.
Посмотрим на основные операции, из которых состоит работа веб-сервера:

- прочитать запрос от пользователя: операция, которая потребляет ресурсы системы ввода-вывода;
- получив запрос, мы должны его разобрать: вычисления, потребляем время центрального процессора;

- для подготовки ответа для пользователя нам необходимо сходить в базу данных или сделать запрос какому-то удаленному API: ресурсы системы ввода-вывода;
- на основе этих данных генерируем HTML: вычисления;
- отправляем этот полученный HTML пользователю: ввод-вывод.

Наш веб-сервер запускается операционной системой в потоке. В рамках одного потока может выполняться одновременно только одна операция, независимо от того, вычислительная она, или, например, мы читаем файл с диска. Для обращения к системам ввода-вывода приложение активно взаимодействует с операционной системой. И по умолчанию данное взаимодействие — **блокирующее**.
Например, мы выполняем вычисления, и для ответа нам понадобилось прочитать какие-то данные из локального файла. Мы делаем запрос в операционную систему и начинаем ждать данные из этого файла. Так как ввод-вывод у нас блокирующий, на это время наше приложение замирает и не может больше выполнять никаких операций. В этот самый момент к нам может прийти второй пользователь, и мы не сможем его обслужить. Таким образом, мы для каждой операции ввода-вывода блокируем поток выполнения нашей программы.
Блокирующий ввод-вывод может накладывать существенные ограничения на производительность нашего веб-сервера. Допустим, мы читаем 1 Кб данных с твердотельного накопителя. Это занимает 0,0014 миллисекунды. За это время стандартное ядро двухгигагерцового процессора позволяет выполнить 28 тысяч циклов, то есть примерно 28 тысяч операци. Но пока мы ждем этот килобайт данных, наш поток приложения блокируется и мы не можем выполнить эти 28 тысяч циклов.

4.2. Многопоточность и неблокирующий ввод/-вывод

4.2.1. Многопоточность

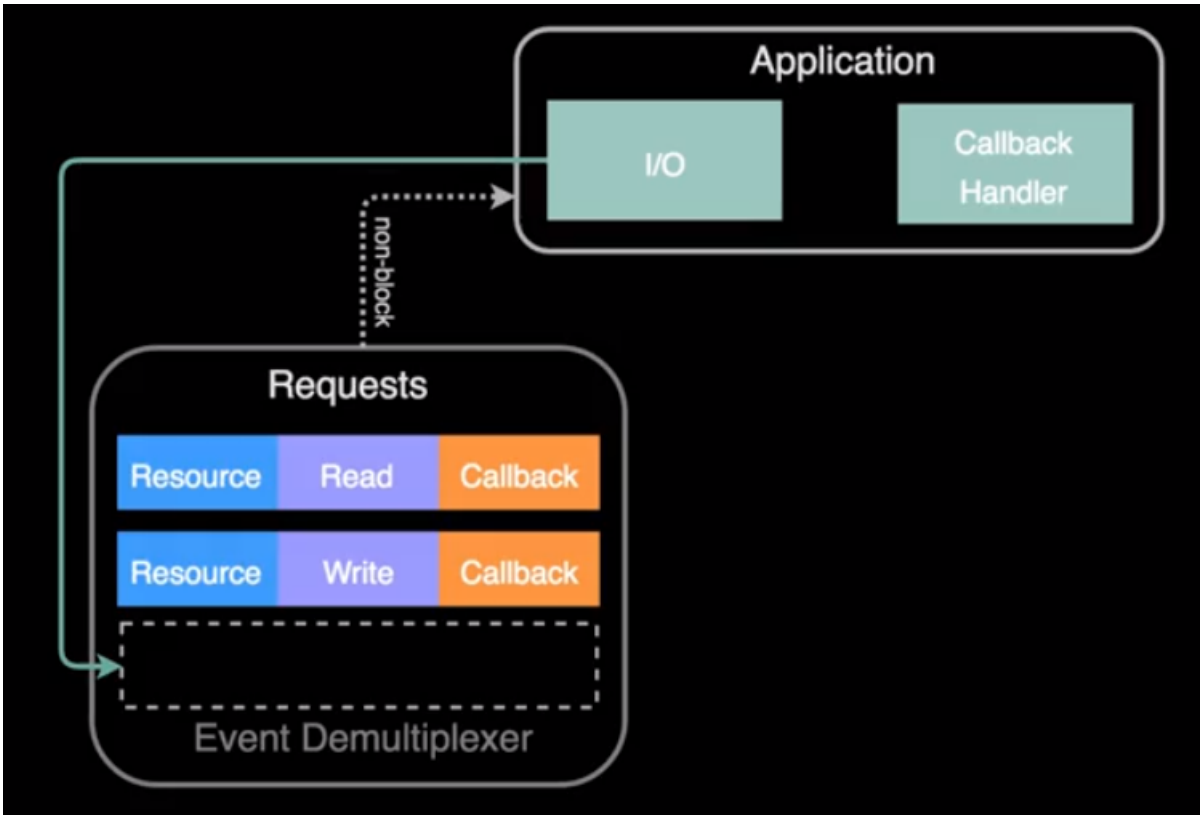
Мы поднимаем еще один экземпляр нашего приложения в отдельном потоке. И пока первый поток занят ожиданием операции ввода/вывода, второй поток в этот момент может без ограничений обслужить следующего пользователя.
Ограничения метода:

- Поднятие потока — это не бесплатная операция, но это несущественно, т.к. можем использовать пул подготовленных потоков;
- есть определенный лимит на количество поднимаемых потоков в операционной системе, и если количество пользователей, которое нам необходимо обслужить, будет заметно превышать этот лимит, мы снова столкнемся с проблемами с производительностью;
- каждый поток требует дополнительной памяти для своей работы.

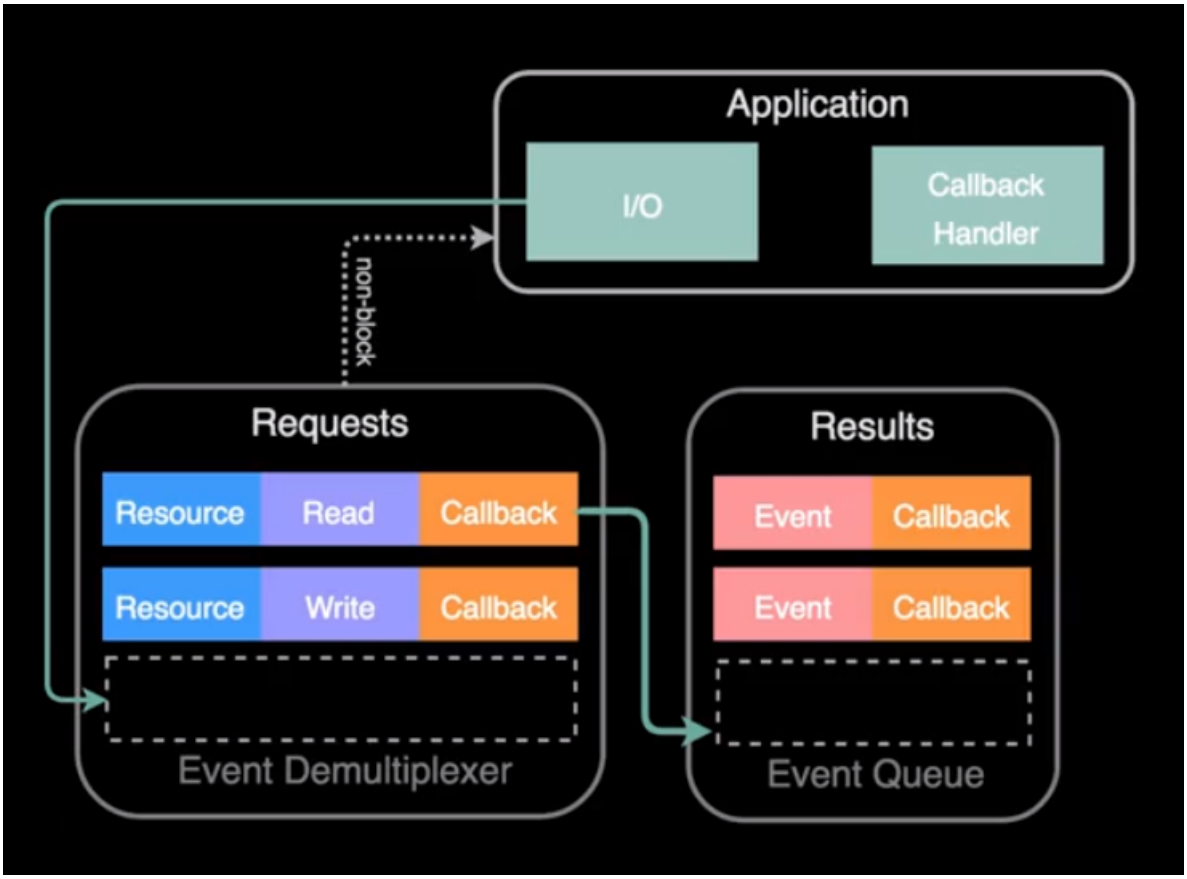
.

4.2.2. Паттерн `reactor`

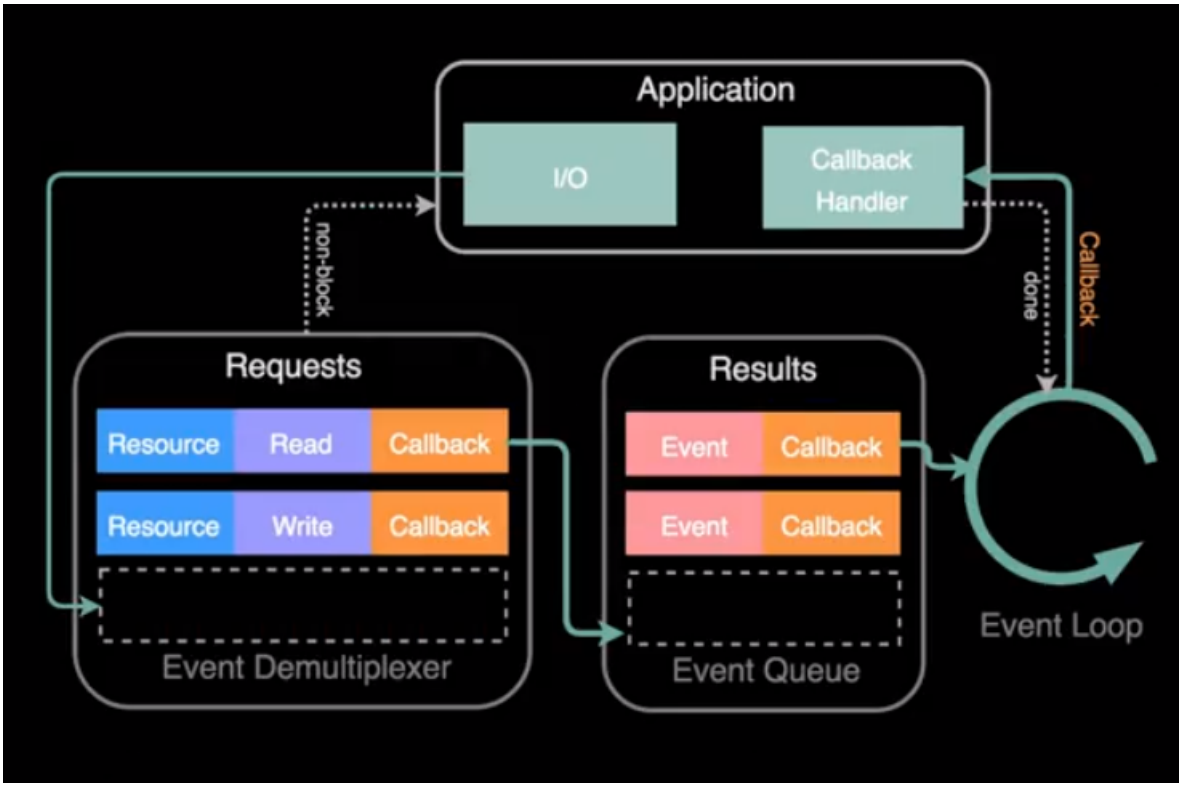
Как только мы запрашиваем какой-то ресурс, например, локальный файл или пытаемся сделать запрос по сети, мы делаем это со специальным флагом. Как только мы делаем запрос, операционная система его регистрирует и в тот же самый момент возвращает управление в поток нашего приложения. И наше приложение может заниматься другой полезной работой. Как только операционная система подготовит ресурс для обработки, она уведомит нас об этом, и наше приложение получит это уведомление и может с этим файлом что-то сделать.
Давайте подробнее рассмотрим, как наше приложение работает с неблокирующей системой ввода/вывода. Например, для продолжения работы нашего приложения ей необходимо прочитать некоторый файл. Для этого она регистрирует необходимый ресурс и операцию над ним в специальном механизме ОС — демультиплексере событий.



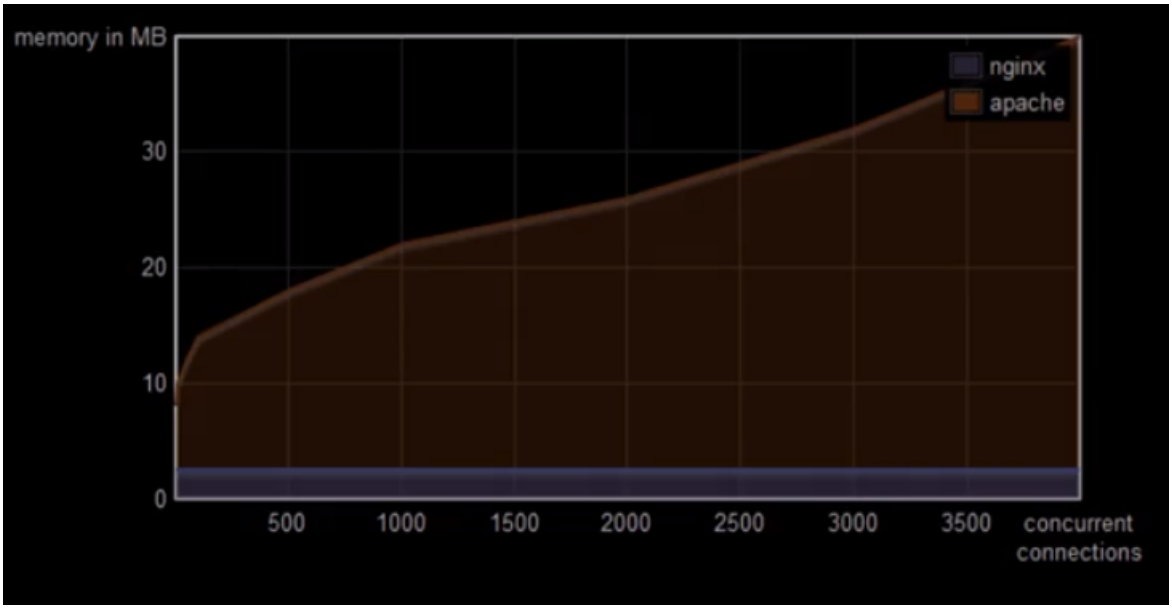
Помимо этого, она в запросе указывает ссылку на обработчик/`callback`, который необходимо выполнить, как только ресурс будет готов. После регистрации запроса на ресурс управление передается обратно в приложение, и наше приложение не блокируется и может продолжать заниматься полезными делами. Как только ресурс будет готов, демультиплексер событий положит в очередь событий сообщение о том, что он готов. С каждым событием будет связан обработчик, который необходимо вызвать.

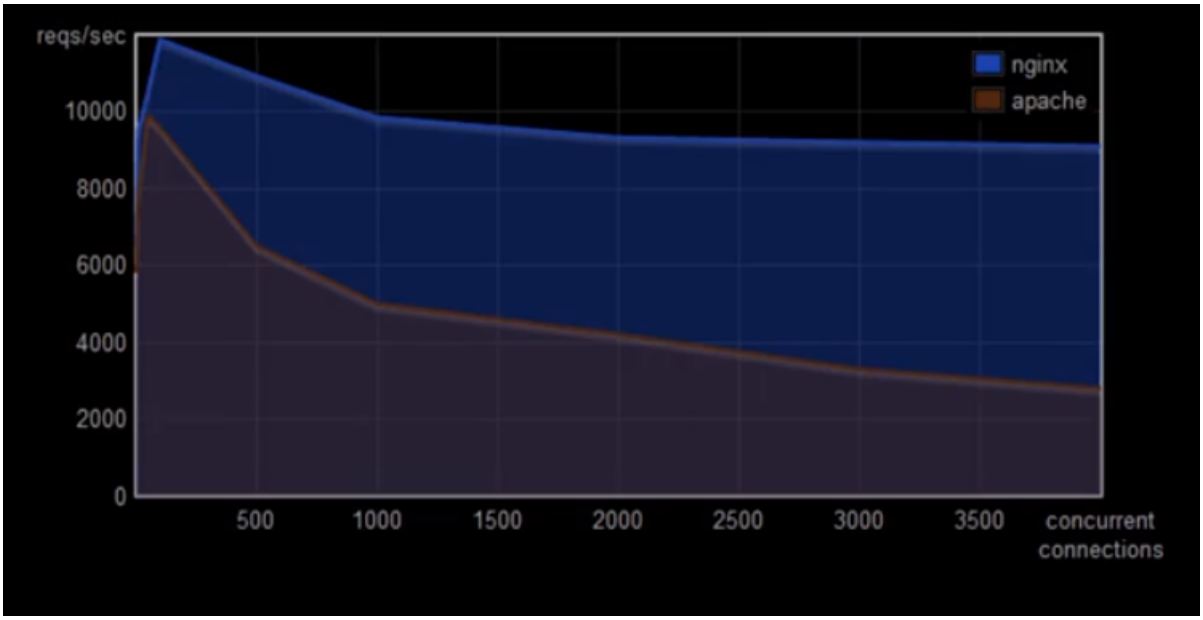


Очередь событий шаг за шагом разбирает в бесконечном цикле обработчик событий. Он вызывает обработчики, они выполняются на стороне приложения, и управление обратно возвращается обработчику событий, он берет следующее событие, вызывает следующий обработчик. В момент выполнения обработчика на стороне приложения может возникнуть потребность запросить еще ресурсов, и тогда мы снова пойдем по кругу.



Весь этот набор механизмов и взаимодействий описан в паттерне `reactor`. Мы рассмотрели два способа решения проблемы блокирующего ввода/вывода. Сравнение двух этих подходов наиболее ярко прослеживается в противостоянии двух известных серверов: Apache и Nginx. Изначально Apache использовал первый подход, а именно многопоточность: на каждый запрос он поднимал отдельный поток. В то же самое время Nginx использовал уже паттерн `reactor` для обработки входящих запросов.

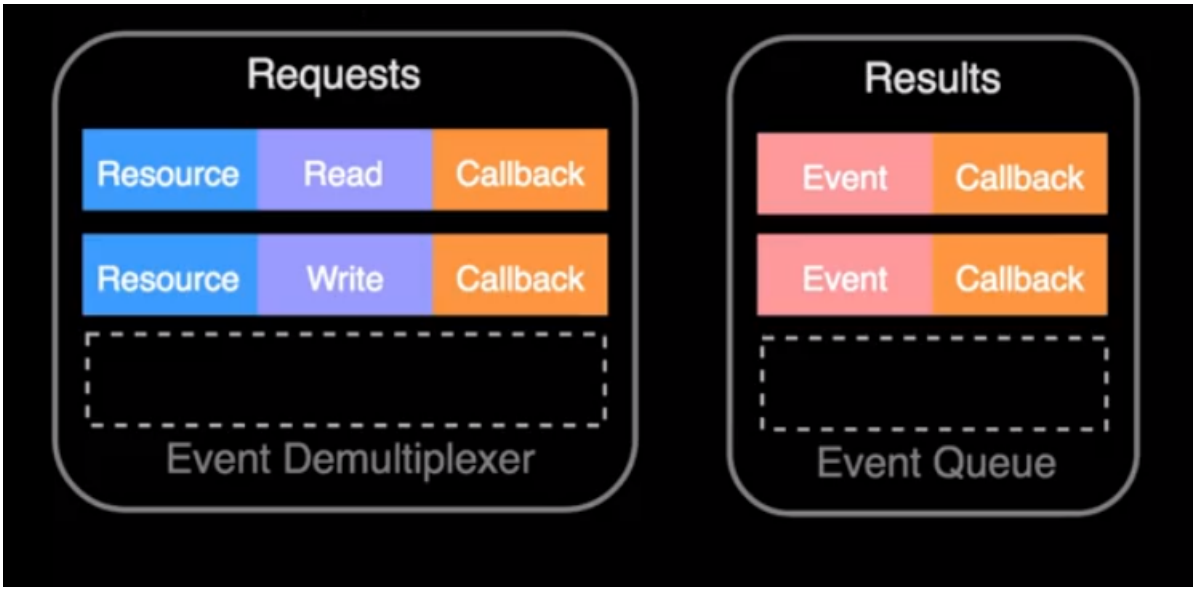




В то время, как для обработки все большего количества одновременных запросов серверу Apache понадобилось все больше и больше памяти, потребление памяти сервером Nginx с увеличением количества одновременных запросов оставалось на одном уровне.

4.3. Архитектура Node.js

Преимущество Nginx на Apache вдохновило разработчика Райана Дала на создание платформы Node.js.
В первую очередь, Райану Далу необходимо было подружить свою программную платформу с демультиплексером событий, но в разных операционных системах данный механизм реализован по-разному. И Райан Дал написал небольшую обертку libuv над тремя реализациями этих систем.
Здесь стоит сделать паузу и подробнее рассмотреть особенности этой библиотеки. Несмотря на то, что все современные операционные системы предоставляют нам неблокирующий интерфейс для работы с вводом-выводом, это не всегда так. Например, в Linux-системах, операции над локальными файлами до сих пор всегда блокирующие, в отличие от тех же сетевых операций. Поэтому для эмуляции неблокирующего поведения libuv под капотом использует первый подход многопоточности для решения этой проблемы.
По умолчанию, эта библиотека поднимает пул из четырех потоков, но этот параметр мы можем поменять. Это значает по умолчанию, что четыре долгих операции над локальными файлами заблокируют нам всё приложение. И об этом важно помнить.
Если вам захочется узнать подробнее об этой библиотеке я рекомендую вам начать с видео Берта Белдера. Далее по ссылке1 и ссылке2 вы можете познакомиться с другими нюансами в работе с этой библиотекой.
Итак, для того чтобы ваша программа могла взаимодействовать с механизмами неблокирующего ввода-вывода при регистрации запроса на какой-либо ресурс, она должна указывать еще и ссылку на обработчик результата `Callback`, который будет вызван, как только ресурс будет готов.



Для этого нашу программу в синхронном стиле мы должны написать в асинхронном: вызовы синхронных методов заменить на асинхронные, и для обработки результатов передавать дополнительно в качестве последнего аргумента функцию «Обработчик». Этот паттерн называется паттерном `Callback` и он следует двум соглашениям:

- они всегда идут последним аргументом;
- если мы не смогли получить какой-то ресурс, ошибка об этом должна приходить в обработчик в качестве первого аргумента.

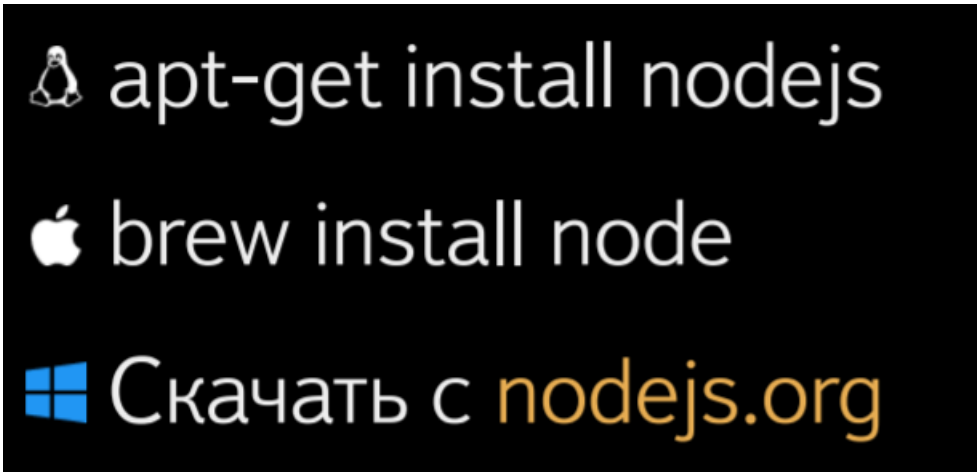
```
var data = request('https://api.github.com/');
var result = writeFile(file, data);
console.info(result);

request('https://api.github.com/', function (err, data) {
    writeFile(file, data, function (err, result) {
        console.info(result);
    });
}
```

Для написания таких программ как нельзя лучше подходит язык программирования JavaScript:

- данный язык из коробки предлагает нам функции первого класса и замыкание, а это значит, что мы можем использовать функции в качестве аргументов;
- язык уже готов к работе с EventLoop — обработчиком событий, благодаря тому, что он прекрасно взаимодействовал с этим механизмом в браузерах;
- язык имеет большое комьюнити.

В качестве интерпретатора был выбран движок V8 от компании Google как наиболее быстрый на тот момент.
Существует API для работы с файловой системой, для осуществления запросов по http и для логирования. Всё это называется Core JavaScript API. Но JS — это Open-source разработка, соответственно, весь исходный код располагается на github.
Всё, что осталось сделать — это объединить библиотеку LIBUV с интерпретатором V8 и с удобным JavaScript API для разработчиков. И это стало возможным благодаря написанию bindings, которые позволили связать код, написанный на C++ и C с кодом, написанным на JavaScript. Все эти библиотеки и механизмы были связаны в единую платформу, которая получила название Node.js.
Я рекомендую вам ознакомиться с оригинальной презентацией этой платформы от Райана Дала, а также по второй и третьей ссылке вы можете изучить более подробно архитектуру этой платформы, познакомиться с тем, как она работает с обработчиком событий.
Начать работу с этой программой-платформой очень просто: достаточно её установить.



На текущий момент разработчики node.js поддерживают две ветки версии: чётные версии — это версии с длительной поддержкой. Вы можете использовать приложение, написанное на платформе этой версии в бою без каких-либо опасений. Нечётные версии — это нестабильные версии, но они включают, как правило, все самые новые возможности.

4.4. Модули

> **Модуль — это текстовый файл, в котором написан код на языке JavaScript. Все Node.js-приложения состоят из набора таких модулей. Если какой-либо модуль реиспользуется между разными приложениями, такие модули называют пакетами.**

В основе синтаксиса этих модулей лежит спецификация, разработанная проектом CommonJS, и по ссылке вы с ней сможете подробней ознакомиться.
Попробуем создать простой модуль для вычисления гипотенузы. В нем есть основная функция и дополнительная для вычисления квадрата числа.

```
// hypotenuse.js
function square(n) {
    return n * n;
}
function calculateHypo(a, b) {
    return Math.sqrt(square(a) + square(b));
}
```

Чтобы наш модуль стал полезным, нам необходимо экспортировать из него основную функцию. Для этого Node.js, перед тем как интерпретировать код нашего модуля, добавляет в него специальный объект с метаинформацией. Помимо прочего в нем содержится, например, полный путь до файла, где этот модуль описан, а также специальное поле exports, которое представляет собой объект. Если мы в этот объект поместим нашу функцию для вычисления гипотенузы, она будет доступна для других модулей.

```
// hypotenuse.js
// module = {
// filename: '/absolute/path/to/hypotenuse.js'
,
// exports: {}
// }
function square(n) {
    return n * n;
}
module.exports.calculate = function (a, b) {
    return Math.sqrt(square(a) + square(b));
}
// return module.exports;
```

Написав простой модуль и экспортировав из него функцию вычисления гипотенузы, мы можем импортировать этот модуль в другом файле при помощи специальной функции `require`.

```
// index.js
var hypotenuse = require('./hypotenuse.js')
hypotenuse.calculate(3, 4); // 5
```

Мы можем экспортировать не только объекты, но и функции. Так, в нашем модуле для вычисления гипотенузы у нас есть одна основная функция, и мы можем сразу экспортировать ее, поместив в специальное поле `exports`.

```
// hypotenuse.js
function square(n) {
    return n * n;
}
module.exports = function (a, b) {
    return Math.sqrt(square(a) + square(b));
}
// index.js
var hypotenuse = require('./hypotenuse.js');
hypotenuse(3, 4); // 5
```

Мы не ограничены экспортом только функции или объектов, и можем экспортировать другие типы данных, например число или конструктор, или экспортировать объект, созданный на основе этого конструктора.

```
module.exports = 42; // Число

function Student(name) {
    this.name = name;
}
Student.prototype.getName = function() {
    return this.name;
};
module.exports = Student; // Конструктор

module.exports = new Student('Billy'); // Объект
```

Существует более лаконичная форма записи экспорта. Так, наряду со специальным объектом модуль, в котором есть поле `exports`, Node.js создает специальную переменную, которая хранит ссылку на это поле, и мы можем записать наш экспорт чуть лаконичнее.

```
// hypotenuse.js
function square(n) {
    return n * n;
}

exports.calculate = function (a, b) {
    return Math.sqrt(square(a) + square(b));
}
```

Но с этой возможностью стоит обращаться немного с осторожностью, так как если мы попробуем записать в эту переменную функцию, то с удивлением обнаружим, что никакую функцию мы на самом деле не экспортировали, так как на самом деле Node.js возвращает всегда то, что находится в поле exports объекта module. А в данном случае мы просто перезаписали нашу переменную exports и связь между этой переменной и полем exports просто потерялась.

```
// hypotenuse.js
function square(n) {}
exports = function (a, b) {
return Math.sqrt(square(a) + square(b));
}
// index.js
var hypotenuse = require('./hypotenuse.js');
hypotenuse(3, 4); // hypotenuse is not a function
```

Помимо собственных модулей мы можем импортировать встроенные в Node.js, и так их достаточно много. Например, мы можем импортировать модуль url. Это модуль предоставляет ряд функций для работы с адресами. Так, мы можем разобрать адресно составные части при помощи функции parse и получить объект с полями, каждое из которых хранит какую-либо из частей адреса.

```
var url = require('url');
url.parse('https://yandex.ru/');
{

    protocol: 'https:',
    host: 'yandex.ru',
    port: null,
    path: '/'

}
```

Если встроенных в Node.js модулей будет недостаточно, мы можем импортировать модули других разработчиков. Для этого мы воспользуемся той же самой функцией `require`, но передадим туда уже идентификатор модуля, который для нее обозначил сторонний разработчик. Так мы можем импортировать самый популярный на данный момент модуль `lodash`. Данный модуль предоставляет несколько полезных методов для работы, например с объектами, или массивами.

```
var lodash = require('lodash');

lodash.shuffle([1, 2, 3, 4]);
// [4, 1, 3, 2]

lodash.uniq([2, 1, 2]);
// [2, 1]
```

Давайте рассмотрим важную особенность импорта модулей. Итак, допустим, у нас есть модуль, в котором есть счетчик (`counter`), и его начальное значение 1. Мы экспортируем из этого модуля функцию, которая ничего не делает, кроме того как увеличивает этот счетчик на 1.

```
var counter = 1;
module.exports = function() {
    return counter++;
};
```

Далее мы попробуем использовать этот модуль, но подключим его дважды, создадим два счетчика: `counter` и `anotherCounter`. Попробуем вызвать вначале первый счетчик. Вызвав первый счетчик, мы получим его начальное значение, и затем оно увеличивается на 1. Вызвав его второй раз, мы получим увеличенное на 1 значение — 2.

```
var counter = require('./counter');
var anotherCounter = require('./counter');
console.info(counter()); // 1
console.info(counter()); // 2
```

Что произойдет, если мы попробуем вызвать другой счетчик? Отсчет продолжится. Почему это так? На самом деле модули импортируются один раз, и после того как функция `require` отработала, экспорт кешируется. Кешируется он в специальном свойстве этой функции — `cache`. Выглядит она примерно так.

```
{
'/absolute/path/to/filename.js': {
filename: '...'
,
exports: {},
}
}
```

Это простой объект, в качестве ключей которых абсолютный путь до файла, где расположен модуль. А далее объект, описывающий этот модуль, включая специальное поле `exports`, где хранятся все экспортированные функции.
Каким же образом Node.js понимает, какой из типов модулей необходимо импортировать прямо сейчас?

- вначале она смотрит, есть ли встроенный модуль с тем именем, который мы передали в функцию `require`: если есть, импортирует его;
- если в функцию `require` мы передали путь до модуля, то импортируется модуль по указанному пути;
- в противном случае модуль или пакет ищется в специальной директории `node_modules`, начиная с текущей директории, где находится файл с нашим исходным кодом.

Полный алгоритм несколько сложнее, и вы сможете ознакомиться с ним по ссылке.

4.5. Пакетный менеджер NPM

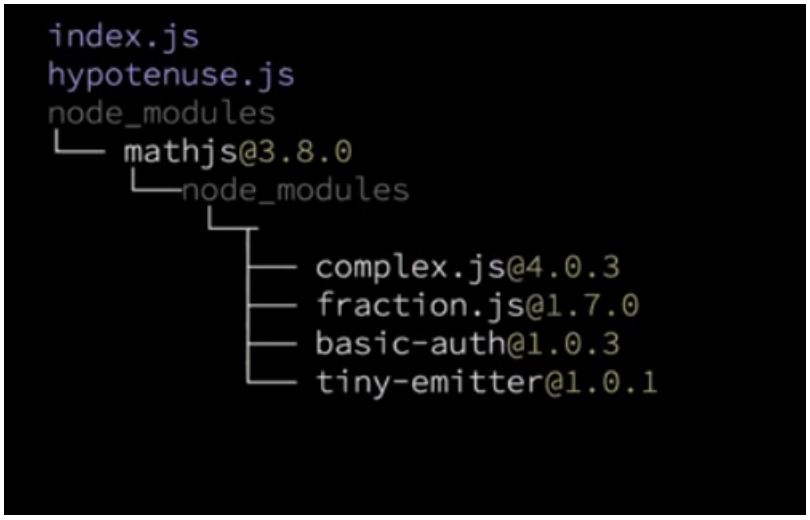
Чтобы находить и устанавливать модули, вместе с Node.js устанавливается специальный инструмент командной строки. Именно он и позволяет находить и устанавливать модули. Этот инструмент взаимодействует с глобальным хранилищем модулей. Вы можете посмотреть информацию об этом хранилище через веб-интерфейс здесь.

Первая команда, которая вам понадобится — команда `init`. После ее запуска она задаст вам несколько простых вопросов: как будет называться ваш модуль, кто ее автор и т.д. После этого она создаст специальный файл-манифест с названием `package` и с расширением json, который будет описывать ваш модуль согласно вашим ответам. Можно сказать, что модуль плюс этот файл-манифест и есть используемый пакет.

Допустим, нашему приложению необходим пакет с математическими функциями. Для того чтобы найти такой пакет, мы можем воспользоваться следующей командой — командой `search`. Она ищет пакет в хранилище по имени и находит все подходящие и выводит в виде списка.

Альтернативный способ поиска пакетов — использование веб-интерфейса. Вы можете воспользоваться официальным сайтом этого пакетного менеджера npmjs.com или его альтернативной версией npms.io. Выбрав подходящий пакет, вы можете посмотреть информацию о нем при помощи команды `show`.

Если вас пакет устраивает, можете установить его при помощи команды `install`, передав ей имя пакета. Данная команда устанавливает пакет в отдельную директорию в качестве зависимости и помещает ее в директорию node_modules на уровне ваших файлов. Если у этой зависимости есть подзависимости, они установятся в ту же самую директорию, но уже у зависимости. Это легко представить в виде дерева файлов.



Если мы посмотрим на результат выполнения этой команды, то увидим, что на уровне наших файлов создалась директория node_modules, внутри которой есть директория с нашей зависимостью. А внутри директории с зависимостью создалась еще одна папка node_modules, внутри которой есть подзависимости для нашей зависимости.

Если ваше приложение совместимо с определенной версией зависимости, вы можете указать ее дополнительно в команде install. Разработчикам пакетов рекомендуется следовать специальному соглашению о версионировании своих пакетов. Это соглашение предлагает следующий формат — разбивать версию пакета на три: мажорную (новые возможности без обратной совместимости с предыдущими версиями), минорную (новые возможности + обратная совместимость) и патчевую (исправление ошибок или рефракторинг), и разделять эти три числа точками.

Мы можем зафиксировать нашу зависимость в файле-манифесте package.json, передав в команду `install` специальный флаг `save`.

Первая группа зависимостей — та, что требуется для работы нашего приложения напрямую, а вторая группа зависимостей — которая необходима для тестирования нашего пакета, сборки и другого типа обслуживания. Для того чтобы установить пакет и записать его во вторую группу, необходимо передать другой флаг в команду `install`, а именно `save-dev`. Зависимости фиксируются в специальный раздел файла-манифеста "dependencies". Мы можем зафиксировать там весь набор зависимостей и, более того, заполнить этот раздел вручную. И тогда установка всего набора будет проходить очень просто: `install` без каких-либо параметров, и она установит за нас все зависимости.

Зафиксировать зависимость мы можем с разной версией, указывая ее в разном формате. Самый простой формат — просто указать версию. Если мы хотим, чтобы при установке нашей зависимости `npm` всегда брал самую свежую версию из определенного диапазона, мы можем указать их в виде диапазона так или таким образом.

```
{

"dependencies": {
"express": "1.2.3",
"express": ">1.2.3",
"express": ">=1.2.3",
}
```

Если мы готовы автоматически устанавливать только свежие патч-версии, но не готовы автоматически устанавливать свежие минор-версии нашей зависимости, тогда перед версией мы добавляем значок «тильда».
Если же мы готовы устанавливать и свежие и минорные версии — добавляем другой значок. Вместо одной из версий мы можем указать «звездочку» или x, что означает «любое число». Более того, вместо версии мы можем указывать определенные теги. Например, тег `latest` означает, что мы хотим устанавливать всегда самую последнюю версию этой зависимости.

```
{
"dependencies": {
"express": "~1.2.3", // >=1.2.3 <1.3.0
"express": "^1.2.3", // >=1.2.3 <2.0.0
"express": "1.2.*",
"express": "latest",
}
```

Более того, `npm` умеет устанавливать зависимости не только из глобального хранилища, но и например с github. Для этого мы в качестве зависимости указываем специальный git URL. Также мы можем указать тег или даже ветку, а может быть и конкретный коммит.

```
{
"dependencies": {
"express": "git://github.com/expressjs/express.git",
"express": "git://github.com/expressjs/express.git#4.13.4",
"express": "git://github.com/expressjs/express.git#master",
"express": "git://github.com/expressjs/express.git#f3d99a4",
}
```

Это далеко не полный набор вариантов, как мы можем фиксировать зависимости. С полным набором вы можете ознакомиться вот по этой ссылке.
Вы можете управлять поведением данной командной строки, записав в файл npmrc ряд опций. Рассмотрим самые полезные из них. Так, вы можете указать опцию `save` со значением `true`, и тогда команда `install` будет всегда фиксировать зависимость в файл-манифест, независимо от того, передали ли вы флаг save или нет. Более того, вы всегда можете фиксировать зависимость строго.

```
save=true // Всегда фиксировать зависимость
save-exact=true // Строго фиксировать версию
```

Я очень рекомендую этот параметр, так как не все разработчики пакетов следуют соглашению о версионировании: даже новая патч-версия вашей зависимости может нарушить работу вашего приложения.
Подробнее прочитать про файл-манифест можно по следующей ссылке, а узнать некоторые трюки и советы по работе с данной командной утилитой — из презентации.

4.6. http-клиент и http-сервер на Node.js

Начнем мы с решения задачи организации веб-сервера: приложения, которое будет принимать запросы от пользователей и отвечать на них. Для решения этой задачи нам понадобится модуль http. Давайте подключим его и попробуем создать объект нашего веб-сервера. Для этого воспользуемся конструктором Server и создадим такой объект. Далее нам необходимо при помощи метода `on` подписать наш сервер на событие `request`, которое будет вызываться каждый раз, когда к нам будет приходить новый запрос от пользователя. Далее нам

необходимо запустить наш веб-сервер. Мы сделаем это при помощи вызова метода `listen` и передадим туда номер порта, который будет слушать наш веб-сервер.

```
var http = require('http');

var server = new http.Server();

server.on('request', function (req, res) {
    res.end('Hello, User!');
});

server.listen(8080);
```

Как только в наш веб-сервер придет новый запрос от пользователя, произойдет событие `request`, и будет вызван обработчик, связанный с этим событием. В качестве первого аргумента в обработчик придет объект, экземпляр конструктора `IncomingMessage`. Данный объект поможет нам получить информацию о запросе: http-метод, с которым был сделан этот запрос, а также заголовки или путь до ресурса, который мы запрашиваем у веб-сервера.

```
server.on('request', function (req, res) {
    console.info(req.method); // GET
});

req.headers; // {'accept-encoding': 'gzip'}

req.url; // /favicon.ico
```

В качестве второго аргумента в наш обработчик придет экземпляр другого конструктора, `ServerResponse`. Здесь можем посмотреть текущий статус ответа или поменять его. Также мы можем поменять заголовки ответа и, наконец, отправить в ответе какое-то сообщение при помощи метода `write`. Мы можем вызывать этот метод несколько раз и отправить несколько пачек ответа. Как только мы поймем, что все, что мы хотели отправить пользователю, отправили, мы завершаем наш ответ методом `end`.

```
server.on('request', function (req, res) {
    console.info(res.statusCode); // 200
});

res.setHeader('content-type', 'text/html');

res.write('<strong>Hello!</strong>');

res.end();
```

Данный модуль позволяет легко сделать запрос к нашему веб-серверу и прочитать ответ от него. Для этого мы вновь подключаем этот модуль, но в этот раз конструируем объект запроса при помощи другого метода, метода request и передаем в него параметры запроса. Мы хотим обратиться к веб-серверу, который у нас запущен локально, по тому самому порту, который этот веб-сервер обслуживает.

```
var http = require('http');

var req = http.request({
    hostname: 'localhost',
    port: 8080
});
```

Далее мы подписываем наш объект запроса на событие «ответ»: происходит каждый раз, как только к нам приходит ответ от веб-сервера. Мы назначаем для этого события обработчик и внутри события начинаем собирать ответ по частям, так как веб-сервер нам его отправляет тоже по частям. Каждый раз, когда веб-сервер будет отправлять нам часть ответа при помощи метода `write`, будет вызвано событие `data`, и мы получаем частичку ответа в отдельную переменную `body`. Как только веб-сервер нам даст сигнал о том, что он закончил с ответом, мы получим событие `end`, и в обработчике этого события мы уже можем вывести ответ. Как только мы навесим наш обработчик на объект запроса, мы можем этот запрос выполнить при помощи метода `end`.

```
req.on('response', function (response) {
    var body = '';

    response.on('data', function (chunk) {
        body += chunk; // res.write(chunk);
    });

    response.on('end', function () {
        console.info(body); // res.end();
    });
});
req.end();
```

Мы работаем с нашим веб-сервером в асинхронном стиле, благодаря этому не блокируем поток выполнения всего приложения. Мы подписываем объект запроса на специальное событие `response` — ответ от сервера. Это становится возможным благодаря тому, что объект запроса в своей основе имеет механизм генератора событий или механизм `EventEmitter`. Данный механизм лежит в основе многих модулей платформы Node.js, и мы сможем познакомиться с ним подробнее, если создадим экземпляр конструктора `EventEmitter`, который лежит в модуле `events`.
Данный механизм позволяет подписывать объекты на события и назначать для этих событий обработчики. В данном случае мы подписываем наш объект на событие `log` и вызываем обработчик, который выводит на экран все, что придет в качестве данных этого события. Далее `EventEmitter` может помочь нам сгенерировать это событие и передать данные, связанные с ним, которые попадут в обработчик.

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

emitter.on('log', console.info);
emitter.emit('log','Hello!'); // Hello!

emitter.emit('unknown event'); // Do nothing
emitter.emit('error'); // Uncaught, unspecified "error" event.
```

Если мы попробуем сгенерировать событие, с которым не связан ни один обра-

ботчик, Node.js просто проигнорирует. Но из этого правила есть одно исключение. Если мы попробуем сгенерировать событие ошибки, в этом случае Node.js не проигнорирует, а наоборот бросит нам ошибку и скажет, что с этим событием не связан ни один обработчик, как бы подталкивая нас всегда назначать обработчики в случае ошибок.
Следующие два модуля помогают нам наладить работу нашего веб-сервера.
Первый из них — это модуль `url`. Этот модуль позволяет нам работать с url-адресами: например, методом `parse`, который позволяет разобрать url-адрес на составляющие. У него есть парный метод, метод `format`, который наоборот позволяет собрать адрес из составляющих в строку.

```
1  url.parse('https://yandex.ru/');
2  // {
3  //      protocol: 'https:',
4  //      host: 'yandex.ru',
5  //      path: '/',
6  // ...
7  // }
8
9  url.format({
10     protocol: 'https:',
11     host: 'yandex.ru'
12 });
13 // https://yandex.ru/
```

И последний модуль `querystring`. Данный модуль позволяет разобрать строку, содержащую GET-параметры, в объект с этими параметрами для дальнейшей обработки. Также есть в нем и парный метод, который позволяет этот объект собрать обратно в строку.

```
querystring.parse('foo=bar&arr=a&arr=b');
// {
// foo: 'bar',
// arr: ['a','b']
// }
querystring.stringify({
    foo: 'bar',
    arr: ['a','b']
});
// foo=bar&arr=a&arr=b
```

4.7. Работа с локальной файловой системой

Модуль для работы с локальными файлами носит название fs и, в частности, предлагает нам метод для асинхронного чтения содержимого файлов, не блокируя при этом поток выполнения приложения.
При помощи метода `readFile` мы можем прочитать этот файл, передав в качестве первого аргумента абсолютный или относительный путь до файла. А в качестве второго аргумента — обработчик, который будет вызван, как только операционная система подготовит для нас этот файл и уведомит нас об этом. В данном случае мы попытаемся прочитать текстовый файл с текущим исходным кодом. Для этого передадим в качестве первого аргумента переменную `fileName`, которая хранит абсолютный путь до текущего файла.

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

fs.readFile(__filename, function (err, content) {
    console.info(content);
});
```

Как только файл будет прочитан, будет вызван обработчик. Если произойдет какая-то ошибка, в качестве первого аргумента мы сможем получить доступ к ней. Если ошибка не возникнет, и файл будет прочитан успешно, то его содержимое придет нам в качестве второго аргумента: объект конструктора `Buffer`, который представляет из себя на экране набор чисел в 16-ричной системе счисления.

Во-первых, этот объект предназначен для работы с бинарными данными. Его можно рассматривать как массив чисел от 0 до 255, где каждое число представляет собой байт.
Создадим для примера другой объект при помощи конструктора `Buffer` из одноименного модуля. В качестве параметра передадим туда массив с кодом символа — буквой B. Для того чтобы преобразовать полученный буфер к строке, достаточно просто вызвать метод `toString`. Более того, мы можем передать в качестве первого аргумента в метод toString кодировку этого символа.

```
1  var letterB = new Buffer([98]);
2
3  console.info(letterB.toString()); // b
4
5  console.info(letterB.toString('utf-8')); // b
```

Допустим, у нас есть некий буфер, и мы не знаем, в какой кодировке написана эта строка. Мы пробуем просто преобразовать его к строке, по умолчанию будет использована кодировка utf8, и мы на выходе получим не совсем желаемый результат.

```
1  var msg = new Buffer([0x2f, 0x04, 0x3d, 0x04, 0x34, 0x04, 0x35, 0x04,
    ↪  0x3a, 0x04, 0x41, 0x04]);
2
3  msg.toString(); // Default: utf8
4
5  // \u0004=\u00044\u00045\u0004:\u0004A\u0004
6
7  msg.toString('ucs2'); // 'Яндекс'
```

Преобразовать буфер в строку можно другим способом, передав параметр, указывающий кодировку методу readFile.

```
fs.readFile(__filename, function (err, data) {
    console.info(data.toString('utf-8'));
});

fs.readFile(__filename, 'utf-8', function (err, data) {
    console.info(data);
});
```

Помимо метода чтения файла, этот модуль, конечно, предлагает и много других возможностей. Мы можем добавлять содержимое файла, перезаписывать его, удалять файлы, и даже работать с директориями.

```
fs.appendFile();

fs.writeFile();

fs.unlink();

fs.mkdir();

fs.stat(__filename, function (stats) {
    console.info(stat.isDirectory()); // false
});
```

Метод `watch` позволяет следить за содержимым файла. Мы можем подписаться на изменения файла и получать уведомления каждый раз, когда этот файл будет изменен. В обработчик будет приходить первым аргументом событие: изменен этот файл или переименован. Таким же образом мы можем следить не только за файлами, но и за директориями.

```
fs.watch();

fs.watch(__filename, function (event, filename) {
    console.info(event); // change or rename
});

fs.watch(__dirname, function (event, filename) {
    console.info(event); // change or rename
});
```

Такой метод часто используется в программах, отвечающих за сборку файлов из какого-то исходного формата в какой-то конечный в автоматическом режиме.
Модуль fs также предлагает для всех методов синхронные способы их вызова. Использовать их следует с осторожностью, так как они будут блокировать поток выполнения вашей программы.

```
fs.readFileSync(__filename);

fs.writeFileSync(__filename, data);

fs.mkdirSync('/games/diablo3');
```