NODE.JS*, C++*, БЛОГ КОМПАНИИ MAIL.RU GROUP

Вы не знаете Node: краткий обзор основных возможностей

ПЕРЕВОД

dedokOne 11 мая 2016 в 11:04 58,2k

Оригинал: Azat Mardan

> **Ремарка от автора**
> Это статья новая, но она не о новых возможностях. Она о core, то есть о платформе и о том что многие кто просто используют grunt, или webpack могут не подозревать, так сказать about fundamentals.

> Более подробно читайте:
>
> комментарии rumkin:
> habrahabr.ru/company/mailru/blog/283228/#comment_8890604
>
> комментарии Aiditz:
> habrahabr.ru/company/mailru/blog/283228/#comment_8890476
>
> комментарии Suvitruf:
> habrahabr.ru/company/mailru/blog/283228/#comment_8890430

Идея этой публикации была навеяна серией книг Кайла Симпсона «Вы не знаете JavaScript». Они являются хорошим началом для изучения основ этого языка. А Node — это практически тот же JavaScript, за исключением небольших отличий, о которых я расскажу в этой статье. Весь код, приведённый ниже, вы можете скачать из репозитория, из папки `code`.

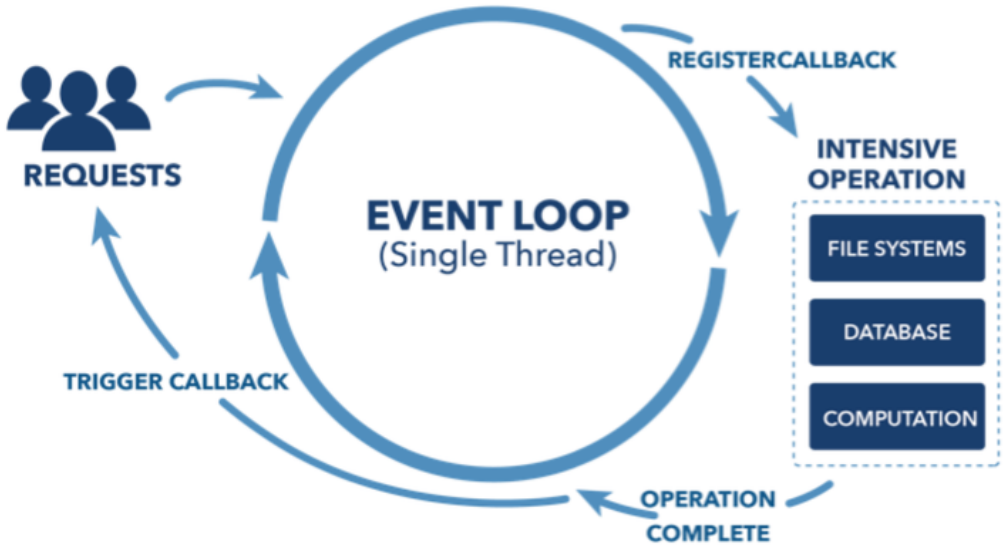
Зачем вообще переживать насчёт Node? Node — это JavaScript, а JavaScript используется почти везде! Мир был бы лучше, если бы большинство разработчиков в совершенстве владели Node. Чем лучше приложения, тем лучше жизнь!

Эта статья представляет собой реалистичный взгляд на наиболее интересные основные возможности Node. Ключевые моменты статьи:

1. **Цикл событий:** освежаем ключевую концепцию, позволяющую реализовать неблокирующие операции ввода/вывода.
2. **Глобальный объект и процесс:** как получить больше информации.
3. **Эмиттеры событий:** интенсивное введение в событийную модель (event-based pattern)
4. **Stream’ы и буферы:** эффективный способ работы с данными
5. **Кластеры:** форкай процессы как профессионал
6. **Обработка асинхронный ошибок:** AsyncWrap, Domain и uncaughtException
7. **Аддоны на C++:** внесение своих наработок в ядро и написание собственных аддонов на C++

Цикл событий

Начнём с цикла событий, лежащего в основе Node.



Неблокирующие операции ввода/вывода в Node.js

Цикл позволяет нам работать с другими задачами параллельно с выполнение операций ввода/вывода. Сравните Nginx и Apache. Именно благодаря циклу событий Node работает очень быстро и эффективно, поскольку блокирующие операции ввода/вывода недёшевы!

Взгляните на этот простой пример отложенной функции `println` в Java:

```
System.out.println("Step: 1");
System.out.println("Step: 2");
```

```
Thread.sleep(1000);
System.out.println("Step: 3");
```

Это сравнимо (хотя и не совсем) с кодом Node:

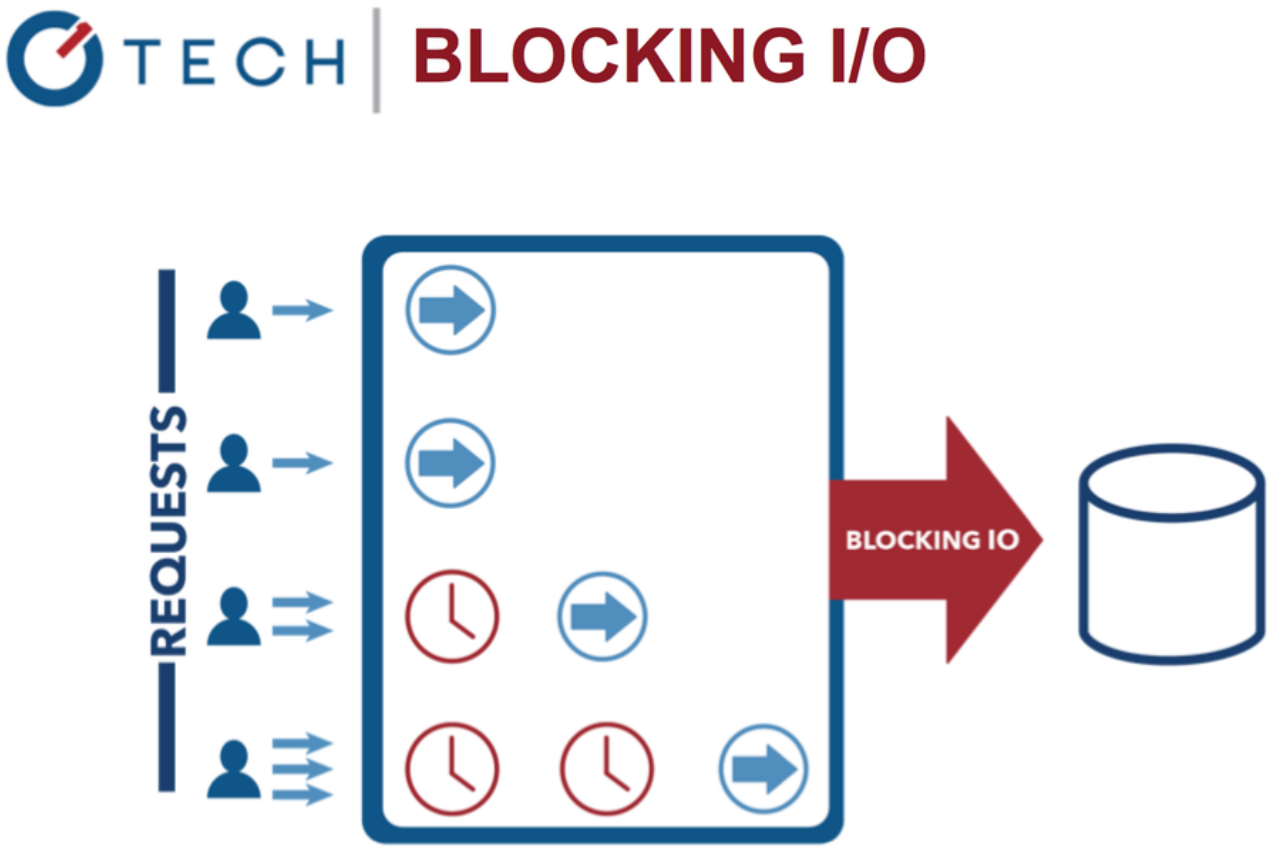
```
console.log('Step: 1')
setTimeout(function () {
  console.log('Step: 3')
}, 1000)
console.log('Step: 2')
```

Это не то же самое. Начните думать с точки зрения асинхронной работы. Выходные данные Node-скрипта — 1, 2, 3; но если бы после “Step 2” у нас было больше выражений, то сначала выполнялись бы они, а уже потом callback функции `setTimeout`. Взгляните на этот фрагмент:

```
console.log('Step: 1')
setTimeout(function () {
  console.log('Step: 3')
  console.log('Step 5')
}, 1000);
console.log('Step: 2')
console.log('Step 4')
```

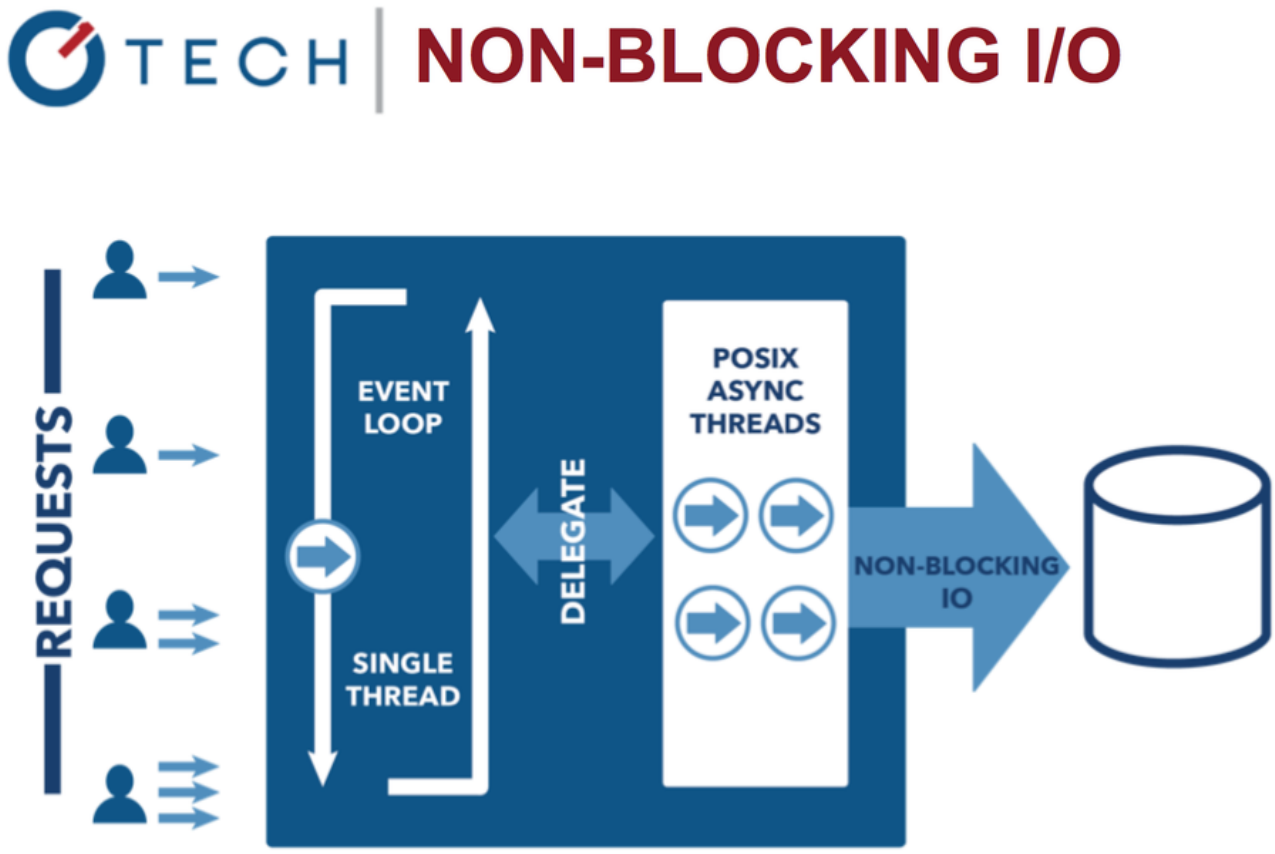
Результатом его работы будет очерёдность 1, 2, 4, 3, 5. Причина в том, что `setTimeout` помещает свой callback в будущие периоды цикла событий.

Можно воспринимать цикл событий как бесконечный цикл наподобие `for … while`. Он останавливается только тогда, когда выполнять больше нечего, ни сейчас, ни в будущем.



Блокирующие операции ввода/вывода: многопоточная Java

Цикл событий позволяет системе работать более эффективно, приложение может сделать что-то еще пока ждет завершение дорогих операций ввода/вывода.



Неблокирующие операции ввода/вывода в Node.js

Это контрастирует с более распространённой сегодня моделью параллельной обработки (concurrency model), при которой задействуются thread'ы операционной системы. Сетевая потоковая модель (thread-based networking) достаточно неэффективна и очень трудна в использовании. Более того, пользователи Node могут не опасаться полного блокирования процессов — здесь отсутствуют lock'и.

К слову, в Node.js всё же можно написать блокирующий код. Присмотритесь к этому простому фрагменту:

```
console.log('Step: 1')
var start = Date.now()
```

```
for (var i = 1; i<1000000000; i++) {
  // This will take 100-1000ms depending on your machine
}
var end = Date.now()
console.log('Step: 2')
console.log(end-start)
```

Конечно, обычно в нашем коде отсутствуют пустые циклы. При использовании чужих модулей может быть труднее выявлять синхронный, а значит блокирующий код. К примеру, основной модуль `fs` (файловая система) идёт с двумя наборами методов. Каждая пара делает одно и то же, но разными способами. Блокирующие методы в модуле `fs` имеют в названиях слово `Sync`:

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

var contents = fs.readFileSync('accounts.txt','utf8')
console.log(contents)
console.log('Hello Ruby\n')

var contents = fs.readFileSync('ips.txt','utf8')
console.log(contents)
console.log('Hello Node!')
```

Результат выполнения этого кода совершенно предсказуем даже для новичков в Node/JavaScript:

```
data1->Hello Ruby->data2->Hello NODE!
```

Но всё меняется, когда мы переходим на асинхронные методы. Вот пример неблокирующего кода:

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

var contents = fs.readFile('accounts.txt','utf8',
function(err,contents){
   console.log(contents);
});
console.log('Hello Python\n');

var contents = fs.readFile('ips.txt','utf8', function(err,contents){
   console.log(contents);
});
console.log("Hello Node!");
```

`contents` выводятся на экран последними, потому что их выполнение занимает какое-то время, они же находятся в callback'ах. Цикл событий перейдёт к ним по окончании чтения файла:

```
Hello Python->Hello Node->data1->data2
```

В общем, цикл событий и неблокирующие операции ввода/вывода — вещи очень мощные, но вам придётся писать асинхронный код, к чему многие не привыкли.

Глобальный объект

Когда разработчики переходят с браузерного JavaScript или иного языка на Node.js, то у них возникают следующие вопросы:

- Где хранить пароли?
- Как создавать глобальные переменные (в Node нет `window`)?

- Как обращаться к входным данным CLI, ОС, платформе, памяти, версиям и т.д.?

Для этого существует глобальный объект, у которого есть определённые свойства. Вот некоторые из них:

- `global.process`: процесс, система, информация об окружении (вы можете обратиться к входным данным CLI, к переменным окружения с паролями, к памяти т.д.)
- `global.__filename`: имя файла и путь к выполняемому в данный момент скрипту, в котором находится это выражение.
- `global.__dirname`: полный путь к выполняемому в данный момент скрипту.
- `global.module`: объект для экспорта кода, создающего модуль из этого файла.
- `global.require()`: метод для импорта модулей, JSON-файлов и папок.

Также есть обычные подозреваемые — методы из браузерного JavaScript:

- `global.console()`
- `global.setInterval()`
- `global.setTimeout()`

К каждому из глобальных свойств можно обратиться как с помощью набранного заглавными буквами имени `GLOBAL`, так и вообще без имени, просто написав `process` вместо

`global.process`.

Процесс

Объект процесса заслуживает отдельной главы, потому что содержит массу информации. Вот некоторые из его свойств:

- `process.pid`: ID процесса этого экземпляра Node.
- `process.versions`: разные версии Node, V8 и других компонентов
- `process.arch`: архитектура системы
- `process.argv`: аргументы CLI
- `process.env`: переменные окружения

Некоторые методы:

- `process.uptime()`: получает время работы
- `process.memoryUsage()`: получает объём потребляемой памяти
- `process.cwd()`: получает текущую рабочую папку. Не путать с `__dirname`, не зависящим от места, из которого был запущен процесс.
- `process.exit()`: выходит из текущего процесс. К примеру, можно передать код 0 или 1.
- `process.on()`: прикрепляет на событие, например, `on(‘uncaughtException’)

Трудный вопрос: кому нравятся и кто понимает суть callback'ов?

Кто-то так сильно в них «влюблён», что создал http://callbackhell.com. Если этот термин вам не знаком, то вот иллюстрация:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height +
'x' + height)
            this.resize(width, height).write(dest + 'w' + width +
'_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

«Ад callback'ов» труден для чтения, и здесь легко можно допустить ошибки. Так как же нам разделять на модули и организовывать асинхронный код, если не с помощью callback'ов, которые не слишком-то удобны для масштабирования с точки зрения разработки.

Эмиттеры событий

Чтобы справиться с адом callback'ов, или пирамидой погибели (pyramid of doom), применяются эмиттеры событий. С их помощью можно реализовать асинхронный код с использованием событий.

Если кратко, то эмиттер событий — это триггер для события, которое может прослушать кто угодно. В Node.js за каждым событием закреплено строковое имя, на которое эмиттером может быть повешен callback.

Для чего нужны эмиттеры:

- В Node события обрабатываются с использованием шаблона "observer".
- Событие (или субъект) отслеживает все связанные с ним функции.
- Эти функции — observer'ы— исполняются при активизации данного события.

Для использования эмиттеров нужно импортировать модуль и создать экземпляр объекта:

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

Далее можно прикрепить получателей событий и активировать/передавать события:

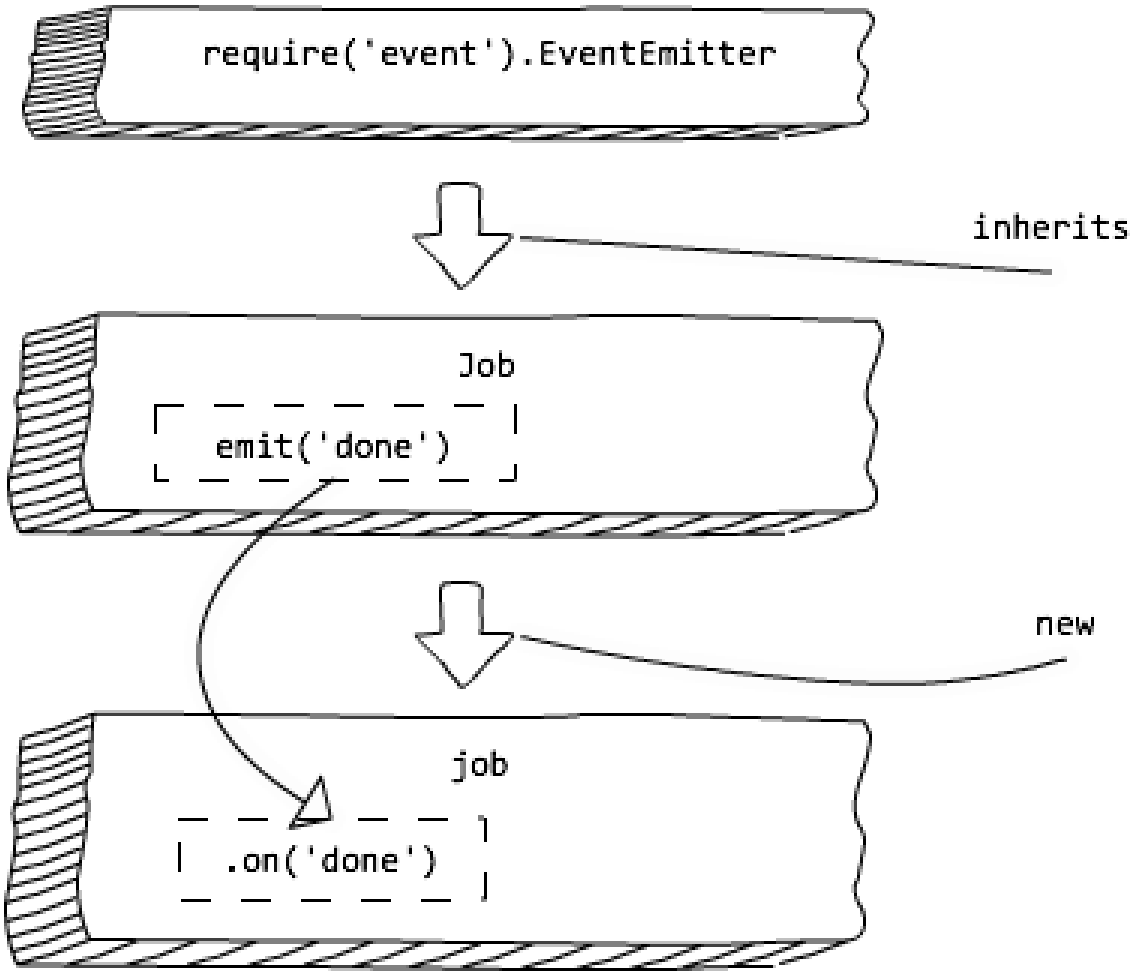
```
emitter.on('knock', function() {
  console.log('Who\'s there?')
})

emitter.on('knock', function() {
  console.log('Go away!')
})

emitter.emit('knock')
```

Давайте с помощью `EventEmitter` сделаем что-нибудь полезное, унаследовав это от него. Допустим, вам регулярно нужно реализовывать какой-то класс — ежемесячно, еженедельно или каждый день. Этот класс должен быть достаточно гибким, чтобы другие разработчики могли кастомизировать финальный результат. Иными словами, по окончании вашей работы любой человек должен иметь возможность поместить в класс какую-то свою логику.

На этой схеме показано, как мы с помощью наследования от модуля событий создаём `Job`, а затем используем получателя событий `done` для изменения поведения класса `Job`:



Эмиттеры событий в Node.js: шаблон “observer”

Класс `Job` сохранит свои свойства, но в то же время получит и события. По окончании процесса нам нужно лишь запустить событие `done`:

```
// job.js
var util = require('util')
var Job = function Job() {
  var job = this
  // ...
  job.process = function() {
    // ...
    job.emit('done', { completedOn: new Date() })
  }
}
```

```
util.inherits(Job, require('events').EventEmitter)
module.exports = Job
```

В завершение изменим поведение `Job`. Раз он передаёт `done`, значит мы можем прикрепить получателя событий:

```
// weekly.js
var Job = require('./job.js')
var job = new Job()

job.on('done', function(details){
  console.log('Job was completed at', details.completedOn)
  job.removeAllListeners()
})

job.process()
```

У эмиттеров есть и другие возможности:

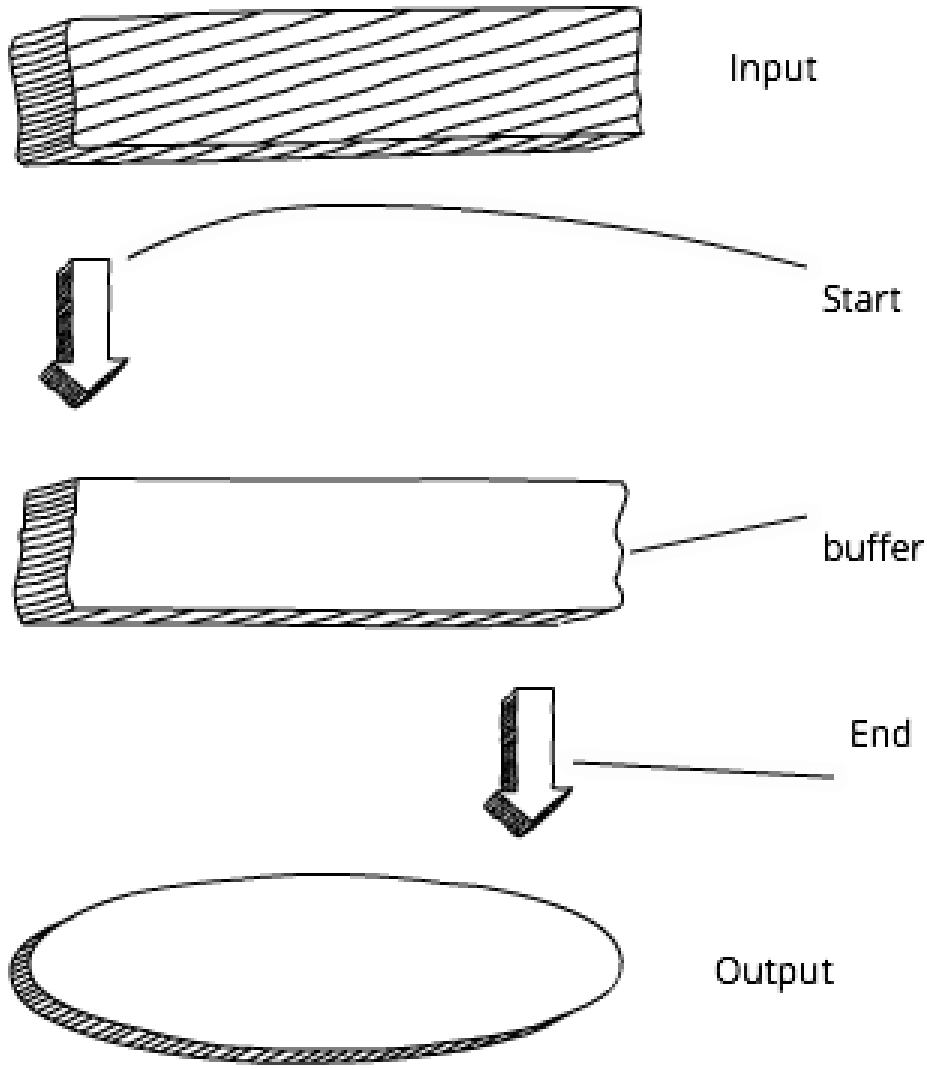
- `emitter.listeners(eventName)`: формирует список всех получателей для данного события.
- `emitter.once(eventName, listener)`: прикрепляет одноразового получателя событий.
- `emitter.removeListener(eventName, listener)`: удаляет получателя событий.

В Node везде используется событийный шаблон, особенно в основных модулях. Так что если вы будете грамотно использовать события, то сэкономите кучу времени.

Stream’ы

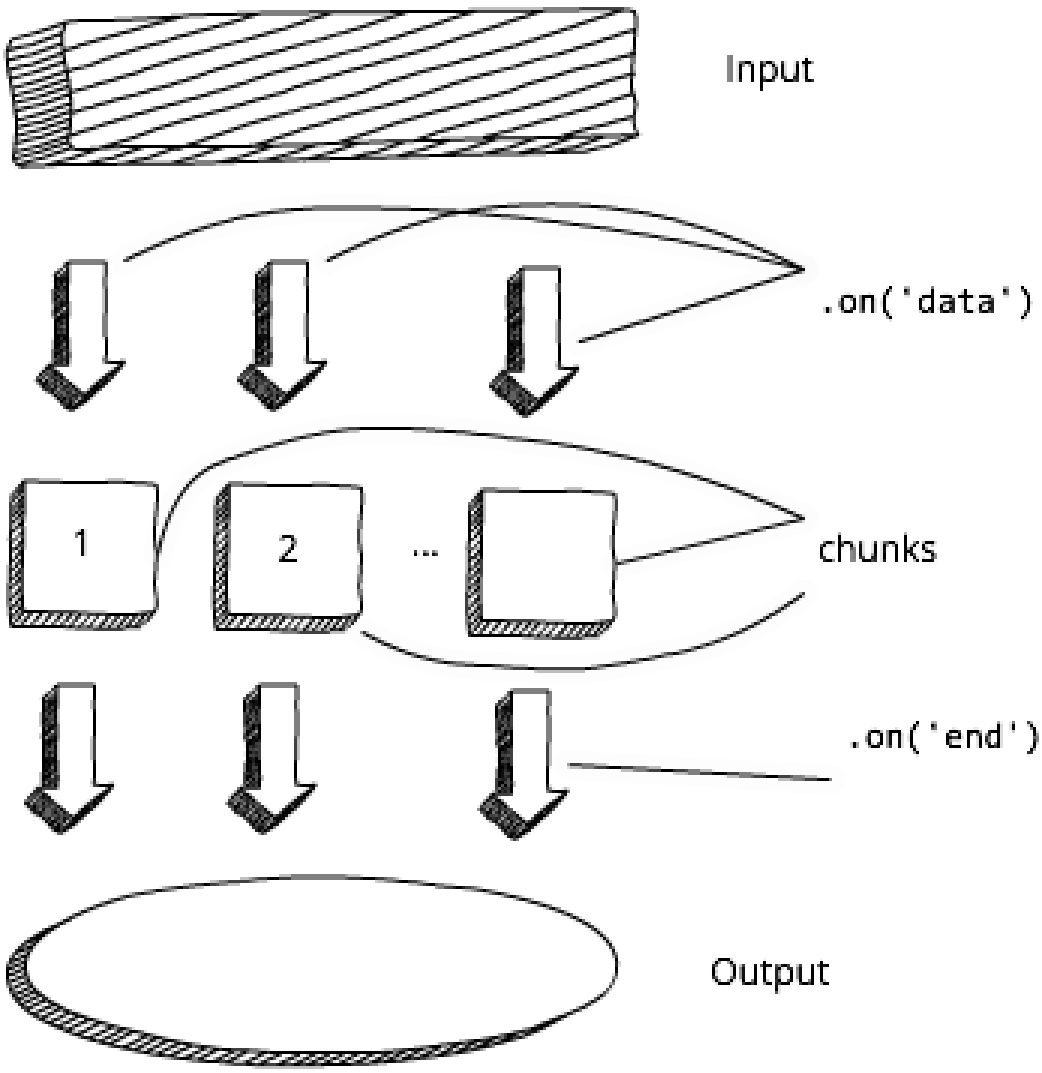
При работе с большими объёмами данных в Node возникает несколько проблем. Производительность может быть низкой, а размер буфера ограничен примерно 1 Гб. Кроме того, как работать в условиях бесконечного ресурса, который создавался из расчёта, что он никогда не закончится? В этих ситуациях нам помогут stream’ы.

Stream’ы в Node являются абстракцией, обозначающей непрерывное разбиение данных на фрагменты. Иными словами, вам не нужно ждать полной загрузки ресурса. На диаграмме показан стандартный подход к буферизации:



Подход к буферизации в Node.js

Прежде чем начать обработку данных и/или выводить их, нам приходится ждать полной загрузки буфера. А теперь сравните это со схемой работы stream’ов. В этом случае мы можем сразу начать обрабатывать данные и/или выводить их, как только получим первый чанк:



Поточный подход в Node.js

В Node есть четыре типа stream’ов:

- Читаемые (readable): из них можно читать.
- Записываемые (writable): в них можно писать.
- Дуплексные (duplex): можно и писать, и читать.
- Stream’ы преобразования (transform): их можно использовать для преобразования данных.

Виртуально stream’ы используются в Node повсеместно. Наиболее востребованные реализации stream’ов:

- HTTP-запросы и отклики.
- Стандартные операции ввода/вывода.
- Чтение из файлов и запись в них.

Для обеспечения шаблона “observer”, stream’ы — события — наследуют от объекта «эмиттер событий». Мы можем использовать это для реализации stream’ов.

Пример читаемого stream’а

В качестве примера можно привести `process.stdin`, являющийся стандартным stream’ом ввода. В нём содержатся данные, которые поступают в приложение. Обычно это информация с клавиатуры, используемая для запуска процесса.

Для считывания данных из `stdin` применяются события `data` и `end`. Callback события `data` в качестве аргумента будет иметь `chunk`:

```
process.stdin.resume()
process.stdin.setEncoding('utf8')

process.stdin.on('data', function (chunk) {
  console.log('chunk: ', chunk)
})

process.stdin.on('end', function () {
  console.log('--- END ---')
})
```

Далее `chunk` скармливается программе в качестве входных данных. Это событие может быть активировано несколько раз, в

зависимости от общего объёма входящей информации. О завершении stream'а необходимо сигнализировать с помощью события `end`.

Обратите внимание: `stdin` по умолчанию ставится на паузу, из которой его надо вывести прежде, чем считывать из него данные.

У читаемых stream'ов есть синхронно работающий интерфейс `read()`. По завершении stream'а он возвращает `chunk` или `null`. Мы можем воспользоваться этим поведением, положив в условие `while` конструкцию `null !== (chunk = readable.read())`:

```
var readable = getReadableStreamSomehow()
readable.on('readable', () => {
  var chunk
  while (null !== (chunk = readable.read())) {
    console.log('got %d bytes of data', chunk.length)
  }
})
```

В идеале, нам хотелось бы как можно чаще писать в Node асинхронный код, чтобы избегать блокирования thread'а. Но благодаря маленькому размеру чанков можно не волноваться насчёт того, что синхронный `readable.read()` заблокирует thread.

Пример записываемого stream'а

В качестве примера можно привести `process.stdout`, являющийся стандартным stream'ом вывода. В нём содержатся данные, которые покидают приложение. Записывать в stream

можно с помощью операции `write`.

```
process.stdout.write('A simple message\n')
```

Записанные в стандартный stream вывода данные отображаются в командной строке, как если бы мы воспользовались `console.log()`.

Pipe

В Node есть альтернатива вышеописанным событиям — метод `pipe()`. В следующем примере происходит чтение данных из файла, сжатие с помощью GZip и запись результата в файл:

```
var r = fs.createReadStream('file.txt')
var z = zlib.createGzip()
var w = fs.createWriteStream('file.txt.gz')
r.pipe(z).pipe(w)
```

`Readable.pipe()` берёт поток данных и пропускает через все stream'ы, поэтому мы можем создавать цепочки из методов `pipe()`.

Так что при использовании stream'ов вы можете применять события или pipe.

HTTP-stream'ы

Большинство из нас используют Node для создания веб-

приложений: традиционных (серверных) или на основе REST API (клиентских). А что насчёт HTTP-запросов? Их можно стримить? Однозначно!

Запросы и отклики представляют из себя читаемые и записываемые stream'ы, наследуемые от эмиттеров событий. Можно прикрепить получателя событий `data` и принимать `chunk` в его callback'е, который можно сразу преобразовывать, не дожидаясь получения всего отклика. В следующем примере мы конкатенируем `body` и парсим его в callback события `end`:

```
const http = require('http')
var server = http.createServer( (req, res) => {
  var body = ''
  req.setEncoding('utf8')
  req.on('data', (chunk) => {
    body += chunk
  })
  req.on('end', () => {
    var data = JSON.parse(body)
    res.write(typeof data)
    res.end()
  })
})

server.listen(1337)
```

Обратите внимание: согласно ES6, `()=>{}` является новым синтаксисом для анонимных функций, а `const` — новый оператор. Если вы ещё не знакомы с особенностями и синтаксисом ES6/ES2015, то можете изучить статью *Top 10 свойств ES6, о которых должен знать каждый занятой JavaScript-разработчик*.

Давайте теперь воспользуемся Express.js, чтобы наш сервер стал

менее оторванным от реальной жизни. Возьмём огромное изображение (около 8 Мб) и два набора Express routes `/stream` и `/non-stream`.

server-stream.js:

```
app.get('/non-stream', function(req, res) {
  var file = fs.readFile(largeImagePath, function(error, data){
    res.end(data)
  })
})
app.get('/stream', function(req, res) {
  var stream = fs.createReadStream(largeImagePath)
  stream.pipe(res)
})
```

Также у меня есть альтернативная реализация `/stream2` с событиями, и синхронная реализация `/non-stream2`. Они делают всё то же самое, но в них использован другой синтаксис и стиль. В данном случае синхронные методы работают быстрее, потому что мы отправляем лишь один запрос, а не несколько конкурирующих.

Запустить это код можно через терминал:

```
$ node server-stream
```

Теперь откройте в Chrome http://localhost:3000/stream и http://localhost:3000/non-stream. Обратите внимание на заголовки во вкладке Сеть в инструментах разработчика, сравнив `X-Response-Time`. В моём случае `/stream` и `/stream2` различались на порядки: 300 мсек. и 3–5 сек. У вас могут быть

другие значения, но идея понятна: в случае со `/stream` пользователи/клиенты раньше начнут получать данные. Stream'ы в Node — очень мощный инструмент! Вы можете научиться хорошо управлять ресурсами stream'ов, став в вашей команде экспертом в этой области.

С помощью npm можете установить себе Stream Handbook и stream-adventure:

```
$ sudo npm install -g stream-adventure
$ stream-adventure
```

Буферы

Какие типы мы можем использовать для бинарных данных? Если помните, в браузере JavaScript нет бинарного типа данных, а в Node есть. Это называется буфером. Он представляет собой глобальный объект, поэтому нет нужды импортировать его в виде модуля.

Можно использовать одно из этих выражений для создания бинарного типа:

- `new Buffer(size)`
- `new Buffer(array)`
- `new Buffer(buffer)`
- `new Buffer(str[, encoding])`

Полный список методов и кодировок доступен в документации по

буферу. Чаще всего используется кодировка `utf8`.

Обычно содержимое буфера выглядит абракадаброй, поэтому, чтобы его можно было прочитать человеку, нужно сначала конвертировать его в строковое представление с помощью `toString()`. Создадим буфер с алфавитом с помощью цикла `for`:

```
let buf = new Buffer(26)
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97 // 97 is ASCII a
}
```

Если не конвертировать буфер в строковое представление, то он будет выглядеть как массив чисел:

```
console.log(buf) // <Buffer 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
6e 6f 70 71 72 73 74 75 76 77 78 79 7a>
```

Осуществим конвертацию:

```
buf.toString('utf8') // outputs: abcdefghijklmnopqrstuvwxyz
buf.toString('ascii') // outputs: abcdefghijklmnopqrstuvwxyz
```

Если нам нужно лишь часть строки (sub string), то метод берёт начальное число и конечную позицию нужного отрезка:

```
buf.toString('ascii', 0, 5) // outputs: abcde
buf.toString('utf8', 0, 5) // outputs: abcde
buf.toString(undefined, 0, 5) // encoding defaults to 'utf8',
outputs abcde
```

Помните `fs`? По умолчанию значение `data` тоже является буфером:

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) return console.error(err)
  console.log(data)
});
```

`data` выполняет роль буфера при работе с файлами.

Кластеры

Противники Node часто приводят аргумент, что он может масштабироваться, поскольку обладает лишь одним thread'ом. Однако с помощью основного модуля `cluster` (вам не нужно его устанавливать, это часть платформы) мы можем использовать все ресурсы процессора на любой машине. Иными словами, благодаря кластерам мы можем вертикально масштабировать Node-приложения.

Код очень прост: импортируем модуль, создаём одного мастера и несколько работников (worker). Обычно создают по одному процессу на каждый ЦПУ, но это не является незыблемым правилом. Вы можете наделать столько процессов, сколько пожелаете, но с определённого момента прирост производительности прекратится, согласно закону убывания доходности.

Код мастера и работника находится в одном файле. Работник может прослушивать тот же порт, отправляя мастеру сообщения посредством событий. Мастер может слушать события, и в случае необходимости перезапускать кластеры. Для мастера используется `cluster.isMaster()`, для работника — `cluster.isWorker()`. Большая часть серверного кода будет расположена в работнике (`isWorker()`).

```
// cluster.js
var cluster = require('cluster')
if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork()
  }
} else if (cluster.isWorker) {
  // ваш серверный код
})
```

В этом примере мой сервер выдаёт ID процессов, поэтому можно наблюдать, как разные работники обрабатывают разные запросы. Похоже на балансировщика нагрузки, но это лишь впечатление, потому что нагрузка не будет распределяться равномерно. Например, по PID вы увидите, как один из процессов может обрабатывать гораздо больше запросов.

Чтобы посмотреть, как разные работники обслуживают разные запросы, воспользуйтесь нагрузочным тестовым инструментом `loadtest` на базе Node:

1. Установите `loadtest` с помощью `npm: $ npm install -g loadtest`

2. Запустите `code/cluster.js` с помощью node (`$ node cluster.js`); пусть сервер продолжает работать.
3. Запустите в новом окне нагрузочное тестирование: `$ loadtest http://localhost:3000 -t 20 -c 10`.
4. Проанализируйте результаты из серверного терминала и из терминала `loadtest`.
5. По окончании тестирования в серверном терминале нажмите Ctrl+C. Вы увидите разные PID.

В команде `loadtest` кусок `-t 20 -c 10` означает, что будет выполнено 10 конкурентных запросов в течение не более чем 20 секунд.

Кластер — это часть ядра, и это практически единственное его преимущество. Когда ваш проект будет готов к развёртыванию, вам может понадобиться воспользоваться более продвинутым диспетчером процессов. Хорошим выбором могут быть:

- `strong-cluster-control` (https://github.com/strongloop/strong-cluster-control) или `$ slc run`
- `pm2` (https://github.com/Unitech/pm2)

pm2

Давайте рассмотрим инструмент `pm2`, поскольку это один из лучших способов вертикального масштабирования вашего Node-приложения. Кроме того, `pm2` повышает производительность на стадии production.

Преимущества `pm2`:

- Балансировщик нагрузок и ряд других полезных возможностей.
- Отсутствие даунтайма при перезагрузке, то есть он работает постоянно.
- Хорошее покрытие тестами.

Документацию можно найти в https://github.com/Unitech/pm2 и http://pm2.keymetrics.io.

В качестве примера использования pm2 можно привести экспресс-сервер `server.js`. Хорошо, что здесь нет шаблонного кода `isMaster()`, потому что вам не придётся модифицировать исходный код, как мы это делали в случае с `cluster`. Здесь достаточно лишь логгировать `pid` и сохранять по ним статистику.

```
var express = require('express')
var port = 3000
global.stats = {}
console.log('worker (%s) is now listening to http://localhost:%s',
 process.pid, port)
var app = express()
app.get('*', function(req, res) {
  if (!global.stats[process.pid]) global.stats[process.pid] = 1
  else global.stats[process.pid] += 1;
  var l ='cluser '
    + process.pid
    + ' responded \n';
  console.log(l, global.stats)
  res.status(200).send(l)
})
app.listen(port)
```

Для запуска этого примера введите `pm2 start server.js`. Можете размножить экземпляры/процессы, введя необходимое количество (`-i 0` означает, что процессов должно быть столько же, сколько ЦПУ, в моём случае их 4). Для сохранения лога в файл используйте опцию `-l log.txt`:

```
$ pm2 start server.js -i 0 -l ./log.txt
```

Приятно то, что pm2 работает в фоновом режиме. Для просмотра исполняемых в данный момент процессов введите:

```
$ pm2 list
```

Теперь воспользуйтесь `loadtest`, как мы это делали в примере с `cluster`. В новом окне выполните:

```
$ loadtest  http://localhost:3000 -t 20 -c 10
```

У вас могут быть другие результаты, но у меня в `log.txt` распределение получилось более-менее равномерным:

```
cluser 67415 responded
 { '67415': 4078 }
cluser 67430 responded
 { '67430': 4155 }
cluser 67404 responded
 { '67404': 4075 }
```

```
cluser 67403 responded
 { '67403': 4054 }
```

Сравниваем Spawn, Fork и Exec

В примере `cluter.js` для создания нового экземпляра Node-сервера был использован `fork()`. Но нужно сказать, что запустить внешний процесс из Node.js можно тремя способами: `spawn()`, `fork()` и `exec()`. Причём все они берутся из основного модуля `child_process`. Разница между этими тремя способами заключается в следующем:

- `require('child_process').spawn()`: используется для больших объёмов данных; поддерживает stream'ы; может применяться с любыми командами; не создаёт новый экземпляр V8.
- `require('child_process').fork()`: создаёт новый экземпляр V8 и экземпляры работников; работает только со скриптами Node.js (команда `node`).
- `require('child_process').exec()`: неудобен для больших объёмов данных, потому что использует буфер; работает асинхронно, чтобы одновременно получать все данные в callback'е; может применяться с любой командой, не только `node`.

Посмотрите на следующий пример: мы исполняем `node program.js`, но можно запустить и любые другие команды или скрипты — bash, Python, Ruby и т.д. Если нужно передать команде дополнительные аргументы, просто положите их в виде

аргументов массива, который является параметром `spawn()`. Данные попадут в обработчик события `data` в виде stream'а:

```
var fs = require('fs')
var process = require('child_process')
var p = process.spawn('node', 'program.js')
p.stdout.on('data', function(data)) {
  console.log('stdout: ' + data)
})
```

С точки зрения команды `node program.js`, `data` является стандартным способом вывода, например, в терминал.

Синтаксис `fork()` почти аналогичен синтаксису метода `spawn()`, за одним исключением: здесь не нужна команда, потому что `fork()` предполагает, что все процессы относятся к Node.js:

```
var fs = require('fs')
var process = require('child_process')
var p = process.fork('program.js')
p.stdout.on('data', function(data)) {
  console.log('stdout: ' + data)
})
```

Наконец, `exec()`. Этот метод отличается от предыдущих тем, что использует не шаблон событий, а одиночный callback. Внутри него есть error, standard output и параметры стандартной ошибки:

```
var fs = require('fs')
var process = require('child_process')
var p = process.exec('node program.js', function (error, stdout,
stderr) {
  if (error) console.log(error.code)
})
```

Разница между `error` и `stderr` заключается в том, что первый мы получаем из `exec()` (скажем, `program.js` отказано в доступе), а второй — из вывода ошибки запущенной вами команды (например, `program.js` не смог подключиться к базе данных).

Обработка асинхронных ошибок

Для обработки ошибок в Node.js и почти всех остальных языках программирования используется `try/catch`. С синхронными ошибками эта конструкция работает замечательно.

```
try {
  throw new Error('Fail!')
} catch (e) {
  console.log('Custom Error: ' + e.message)
}
```

Модули и функции кидают ошибки, а мы их потом ловим. Так работает Java и *синхронный* Node. Но в Node.js лучше всего писать *асинхронный* код, чтобы не блокировать thread.

Благодаря циклу событий система может делегировать и применять расписание для кода, который должен быть выполнен в будущем, по завершении ресурсозатратных задач ввода/вывода. Но тут у нас возникает проблема с асинхронными ошибками, потому что контекст ошибки утрачивается.

К примеру, `setTimeout()` работает асинхронно, откладывая на

будущее вызов callback'а. Аналогично поведёт себя и асинхронная функция, делающая HTTP-запрос, читающая из БД или пишущая в файл:

```
try {
  setTimeout(function () {
    throw new Error('Fail!')
  }, Math.round(Math.random()*100))
} catch (e) {
  console.log('Custom Error: ' + e.message)
}
```

Когда выполняется callback и приложение падает, то здесь уже нет `try/catch`. Конечно, можно поместить в callback другую конструкцию `try/catch`, которая поймает ошибку, но это плохое решение. Все эти назойливые асинхронные ошибки труднее вылавливать и отлаживать. Для асинхронного кода `try/catch` не лучший вариант.

Итак, приложение упало. Что нам с этим делать? Вы уже видели, что в большинстве callback'ов есть аргумент `error`. В каждом случае вам нужно проверять и вытаскивать его: отвергнуть цепочку callback'а или вывести пользователю сообщение об ошибке.

```
  if (error) return callback(error)
  //  or
  if (error) return console.error(error)
```

Другой хороший способ обработки асинхронных ошибок заключается в следующем:

- Прослушиваем все сообщения об ошибках (on error).
- Прослушиваем `uncaughtException`.
- Используем `domain` (несколько устарел) или AsyncWrap.
- Логгируем и отслеживаем.
- Уведомляем (по желанию).
- Выходим и перезапускаем процесс.

on('error')

Прослушивайте события `on('error')`, генерируемые большинством основных объектов в Node.js, в особенности `http`. Также `error` генерирует всё, наследуется от или создаёт экземпляры Express.js, LoopBack, Sails, Hapi и т.д., потому что эти фреймворки расширяют `http`.

```
js
server.on('error', function (err) {
console.error(err)
console.error(err)
process.exit(1)
})
```

uncaughtException

Всегда прослушивайте `uncaughtException` в объекте `process`! `uncaughtException` — это очень грубый механизм обработки ошибок. Если ошибка не обработана, то ваше приложение — а значит и Node.js — находится в неопределённом состоянии.

An unhandled exception means your application – and by extension

Node.js itself – is in an undefined state. Если возобновить работу вслепую, то может произойти что угодно.

```
process.on('uncaughtException', function (err) {
  console.error('uncaughtException: ', err.message)
  console.error(err.stack)
  process.exit(1)
})
```

или

```
process.addListener('uncaughtException', function (err) {
  console.error('uncaughtException: ', err.message)
  console.error(err.stack)
  process.exit(1)
```

Domain

У `domain` нет ничего общего с сетевыми доменами в браузере. Это основной модуль Node.js для обработки асинхронных ошибок. Он сохраняет контекст, в котором реализован асинхронный код. Стандартный подход: создать копию `domain` и поместить код с ошибкой внутрь callback'а `run()`:

```
var domain = require('domain').create()
domain.on('error', function(error){
  console.log(error)
})
domain.run(function(){
  throw new Error('Failed!')
})
```

Начиная с версии 4.0 `domain` считается устаревшим, поэтому

разработчики Node наверняка отделят его от платформы. Но на сегодняшний день в ядре Node нет альтернативы `domain`. Кроме того, благодаря серьёзной поддержке и широкому использованию, `domain` будет существовать в виде отдельного npm-модуля, поэтому вы сможете легко переключаться с основного модуля на npm. Так что с `domain` мы не прощаемся.

Давайте сделаем асинхронную ошибку с помощью того же `setTimeout()`:

```
// domain-async.js:
var d = require('domain').create()
d.on('error', function(e) {
   console.log('Custom Error: ' + e)
})
d.run(function() {
  setTimeout(function () {
    throw new Error('Failed!')
  }, Math.round(Math.random()*100))
});
```

Код не упадёт! От принадлежащего `domain` обработчика событий `error` мы получим красивое сообщение “Custom Error”, а не типичную для Node трассировку стека.

Аддоны на C++

Популярность Node среди разработчиков железа, IoT, дронов, роботов и умных гаджетов заключается в том, что он позволяет забавляться с более низкоуровневым кодом на C/C++. Как же можно писать свои C/C++ биндинги?

Это последняя из основных возможностей в этой статье. Большинство новичков в Node даже не представляют о том, что здесь можно писать собственные аддоны на C++! Это настолько просто, что мы прямо сейчас напишем аддон с нуля.

Сначала создадим файл `hello.cc`, в начале которого поместим несколько шаблонных импортов. Затем определим метод, возвращающий строковое и экспортирующий себя.

```
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "capital
one")); // String
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method); // Exporting
}

NODE_MODULE(addon, init)

}
```

Даже если вы не знаток C, всё равно можете легко понять, что здесь происходит, ведь синтаксис не слишком сильно отличается

от JavaScript. Строковым будет `capital one`:

```
args.GetReturnValue().Set(String::NewFromUtf8(isolate, "capital
one"));
```

А после экспортирования будет присвоено имя `hello`:

```
void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}
```

Файл `hello.cc` готов, идём дальше. Создадим `binding.gyp`, содержащий имя файла с исходным кодом и именем аддона:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

Сохраним `binding.gyp` в одной папке с `hello.cc`, а затем установим node-gyp:

```
$ npm install -g node-gyp
```

Теперь в той же папке, где лежат `hello.cc` и `binding.gyp`, выполните эти команды конфигурирования и сборки:

```
$ node-gyp configure
$ node-gyp build
```

Будет создана папка `build`. Зайдите в `build/Release/` и проверьте наличие файлов `.node`. Наконец, напишите на Node.js создающий скрипт `hello.js`, включив в него аддон на C++:

```
var addon = require('./build/Release/addon')
console.log(addon.hello()) // 'capital one'
```

Чтобы запустить скрипт и увидеть на экране содержимое `capital one`, введите:

```
$ node hello.js
```

Другие примеры аддонов на C++ можно найти здесь: https://github.com/nodejs/node-addon-examples.

Заключение

Весь вышеприведённый код можно скачать с GitHub. Если вас заинтересовали шаблоны Node.js, callback'и и Node-соглашения, можете почитать мою статью Шаблоны Node: от callback'ов до observer'а.

Краткое содержание текущей статьи:

1. **Цикл событий:** механизм, лежащий в основе неблокирующих операций ввода/вывода в Node.
2. **Глобал и процесс:** глобальные объекты и системная информация.
3. **Эмиттеры событий:** шаблон “observer” в Node.js.
4. **Потоки:** шаблон работы с большими объёмами данных.
5. **Буферы:** тип двоичных данных.
6. **Кластеры:** вертикальное масштабирование.
7. **Domain:** обработка асинхронных ошибок.
8. **Аддоны на C++:** низкоуровневые аддоны.

Большая часть Node представляет собой тот же JavaScript, за исключением некоторых основных возможностей, по большей части относящихся к системному доступу, глобальным объектам, внешним процессам и низкоуровневому коду. Если вы разобрались со всеми этими концепциями, то вскоре сможете стать настоящим сэнсэем Node.js.

Похожие публикации

Добро Mail.ru и Нетология запускают проект «Безграничные возможности»

6

Soboleva • 21 марта 2014 в 16:50

Рейтинг Mail.Ru: новости с полей

12

gornal • 7 ноября 2013 в 13:59

Форум Технологий Mail.Ru Group: v5.0

6

plaksa • 4 марта 2013 в 15:02

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов

20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE

24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить 85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе 139

marks • вчера в 14:19

Дела шпионские (часть 1)

TashaFridrih • вчера в 13:16

16