Node.js

Часть 2

В предыдущей лекции

- Были кратко рассмотрены основные понятия лежащие в основе Node.js: Событийно-ориентированная асинхронная модель программирования, неблокирующий ввод-вывод.
- Два важных понятия лежащих в основе JavaScript: замыкания и ООП в JavaScript.
- Рассмотрены примеры создания простого веб-сервера и веб-сайта с использованием Node.js

В этой лекции

- Node Core
- Как все работает? Event loop в Node.js
- Глобальные объекты (Globals)
- Процессы
- События Объект EventEmitter
- Модули
- ~~Работа с файлами~~

Node Core

- . Даже в простом движке плоского сайта мы использовали почти десяток модулей, каждый из которых отвечает за что-то свое. Это нормальный принцип построения Node-приложений. Но, разумеется, этим и многим другим модулям просто не с чем было бы работать, если бы не основа системы - ядро Node.js, содержащее объекты и методы, доступные всем модулям, в глобальном пространстве имен. Именно с этого мы и начнем изучение Node, попутно освоив базовые понятия и элементы системы.

Event loop в Node.js

- В основе Node.js лежит библиотека libev, реализующая цикл событий (event loop). Libev - это написанная на C библиотека событийно-ориентированной обработки данных, предназначен-ная для упрощения асинхронного неблокирующего ввода/вывода.

Event loop в Node.js

При каждой итерации цикла происходят следующие события (причем именно в таком порядке):

- выполняются функции, установленные на предыдущей итерации цикла с помощью особого метода - process.nextTick( ), обрабатывающего события libev, в том числе таймеров;
- выполняется опрос libeio (библиотеки для создания пула потоков -thread pool) для завершения операций ввода/вывода и выполнения установленных для них кэллбеков.
- Если ни одно из вышеперечисленных действий не потребовалось (то есть все очереди, таймеры и т. д. оказались пусты), Node.js завершает работу.

Global

- Самый главный в иерархии глобальных объектов так и называется Global.
- В браузере при инициализации переменной на верхнем уровне вложенности она автоматически становится глобальной в смысле области видимости.
- В Node.js переменная, объявленная в любом месте модуля, будет определена только в этом модуле.
- Чтобы переменная стала глобальной, необходимо объявить её как свойство объекта global.

Console.log(global)

Объект Console

- может принимать два аргумента, аналогично функции языка Си printf( ):

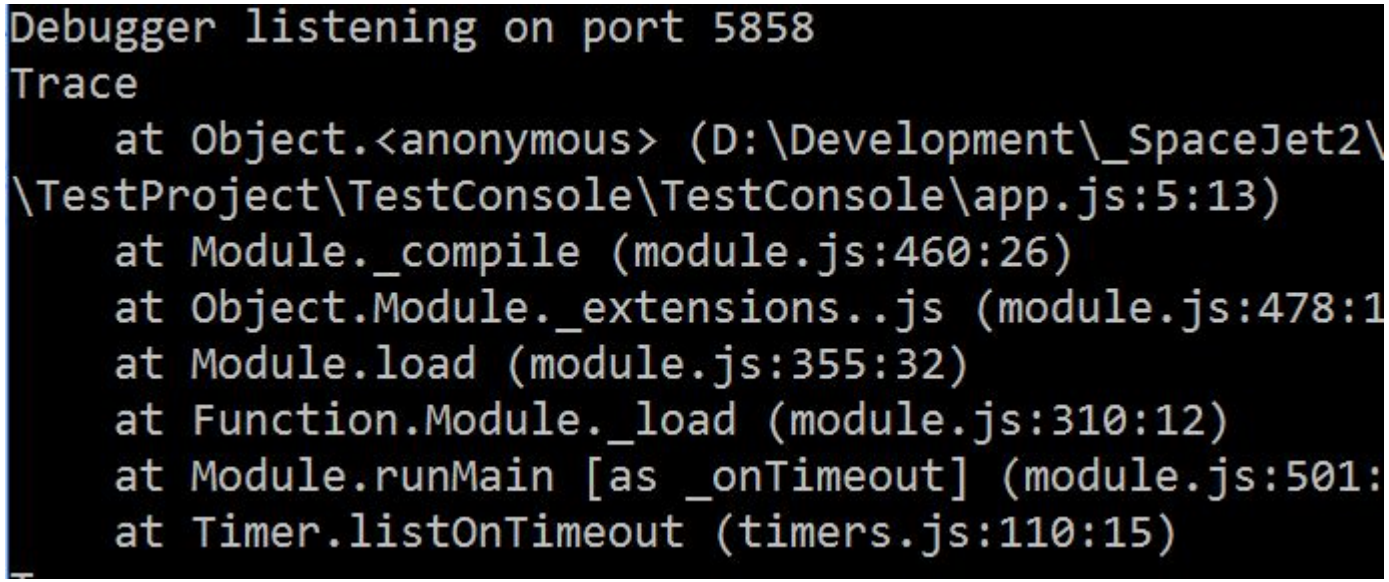
```
console.log('Price: %d', bar); // Price: 7
```

- Для stderr (стандартного потока вывода ошибок) существует метод console.error
- console.time() и console.timeEnd() позволяют отслеживать время исполнения программы
- Для отладки также полезен метод console.trace(), выводящий в консоль стек вызовов для текущей инструкции

Объект Console

```
console.time('items');
for (var i = 0; i < 1000000000; i++) { /* что-нибудь делаем*/ }
console.timeEnd('items'); //2114ms
```

```
console.trace();
```



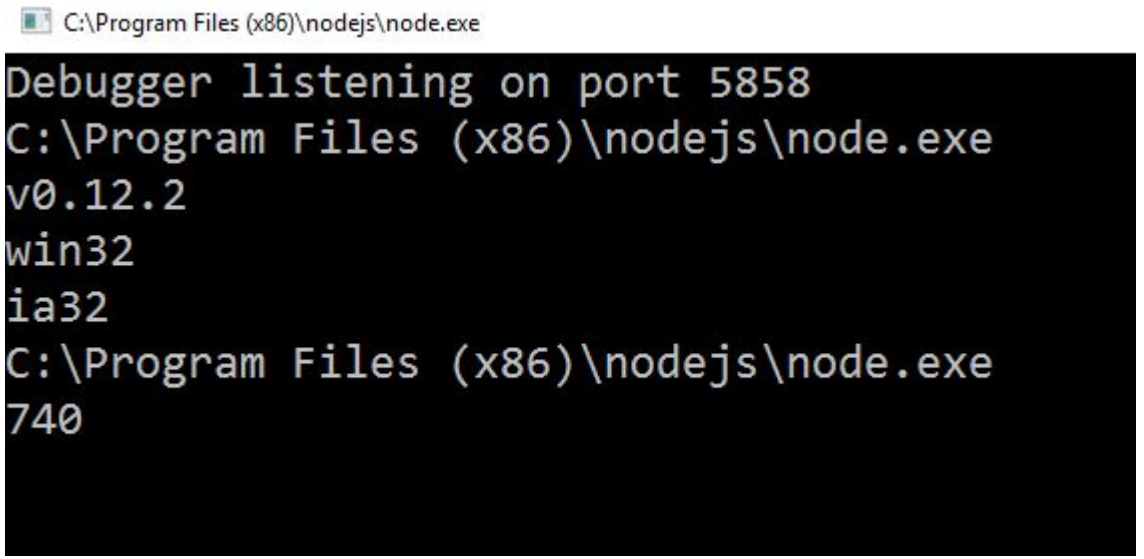
Require и псевдоглобальные объекты

- require.cache отвечает за кэширование модулей

Процессы

Каждое Node.js-приложение - это экземпляр объекта Process и наследует его свойства. Это свойства и методы, несущие информацию о приложении и контексте его исполнения.

```
console.log(process.execPath);
console.log(process.version);
console.log(process.platform);
console.log(process.arch);
console.log(process.title);
console.log(process.pid);
```



Process mouleLoadList, argv

- Свойство process.moduleLoadList показывает информацию о загруженных модулях, а process.argv содержит массив аргументов командной строки:

- C:\Node>node process.js foo bar 1
- О - node
- 1 - C:\Node\process.js
- 2 - foo
- З - bar
- 4 - 1

Process.Exit

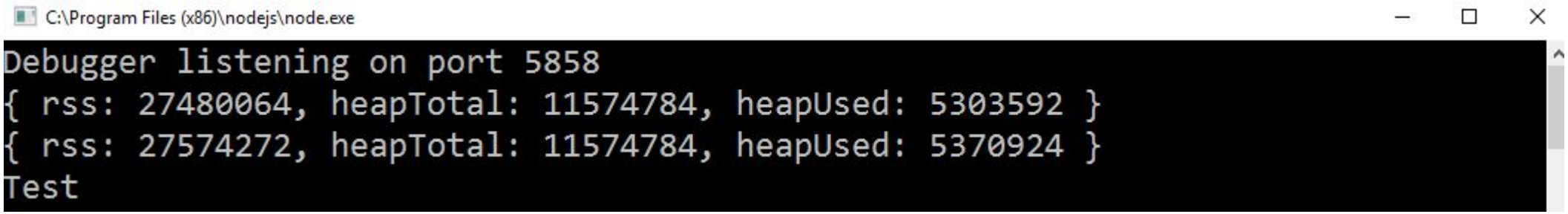
- Команда process.exit() завершает процесс.
- Чтобы выйти с ошибкой, следует выполнить process.exit(1).

```
process.on('exit', function (code) {
    setTimeout(function () {
        console.log('This will not run');

    }, 0);
    console.log('Exit with code:' + code);
});
```

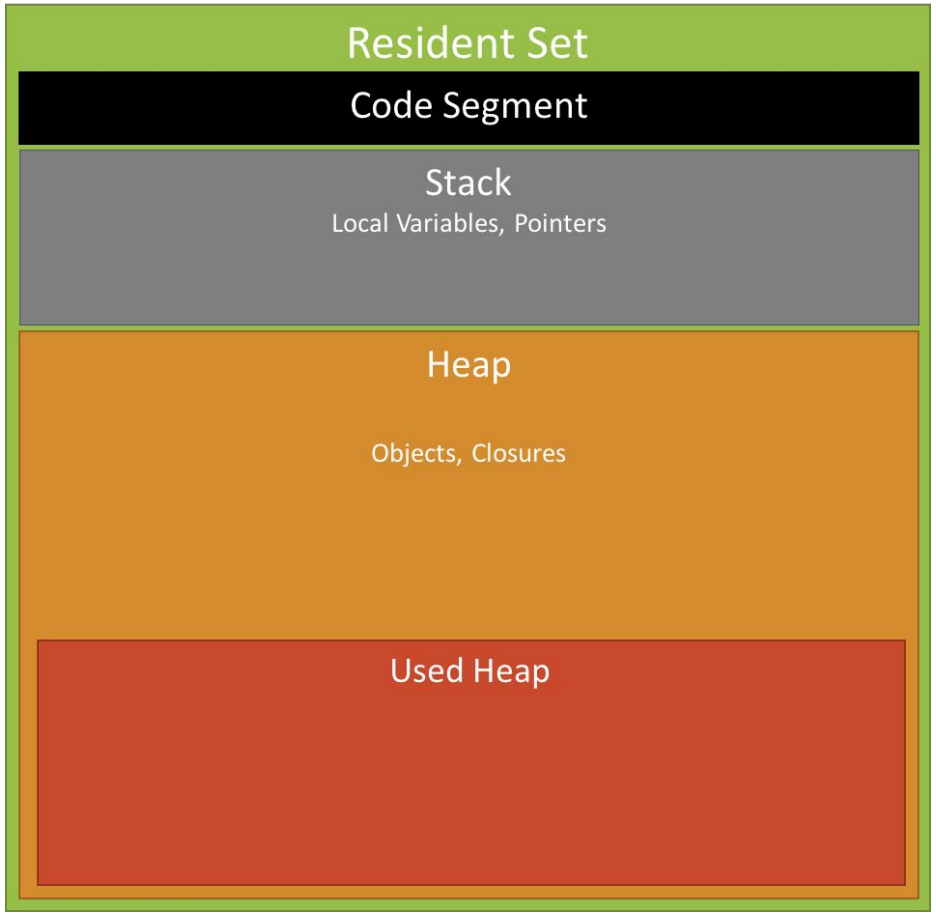
Process Kill

- Метод process.kill(), аргументами которого служат идентификатор процесса и команда, делает то же, что и одноименная команда операционной системы, то есть посылает сигнал процессу.
- process.memoryUsage(), возвращает объект, описывающий потребление памяти процессом Node.

```
console.log(process.memoryUsage());
process.nextTick(function () { console.log('Test'); });
```



RSS, HeapTotal, Heap Used



rss: Resident Set Size
heapTotal: Total Size of the Heap
heapUsed: Heap actually Used

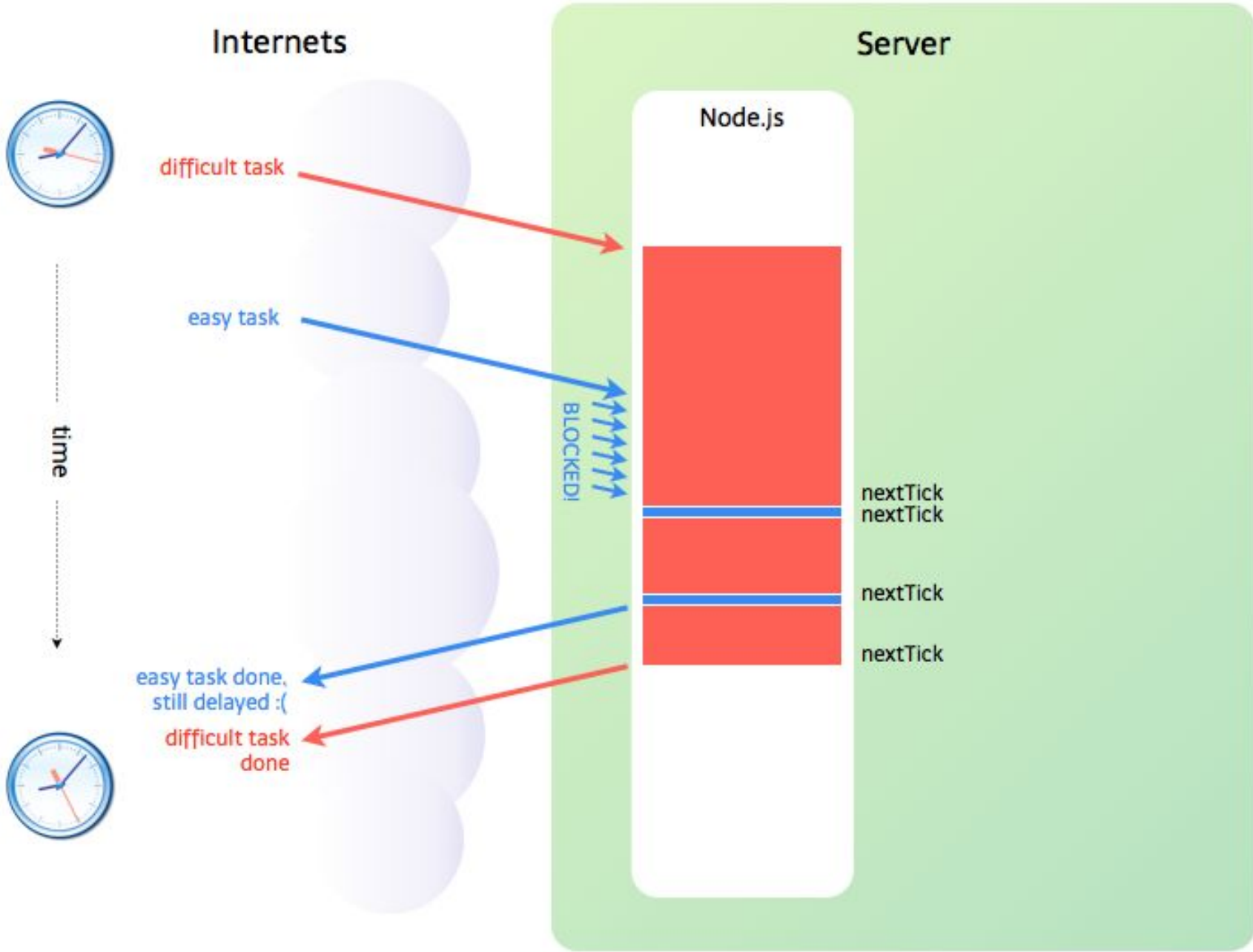
A running program is always represented through some space allocated in memory. This space is called **Resident Set**

Метод process.nextTick()

- NextTick() назначает функцию, служащую ему аргументом, к исполнению при следующей итерации цикла событий. Это такой своеобразный event handler, специфичный для node.js.

- На каждом витке event loop в первую очередь идёт выполнение функций, установленных на предыдущем витке цикла с помощью process.nextTick(). Далее идёт обработка событий libev, в частности событий таймеров. В последнюю очередь идёт опрос libeio для завершения операций ввода/вывода и выполнения установленных для них функций обратного вызова. В случае если ни одной операции не назначено, нет работающих таймеров и очереди запросов в libev и libeio пусты, node завершает работу.

Single-tasking Node.js server, using process.nextTick()



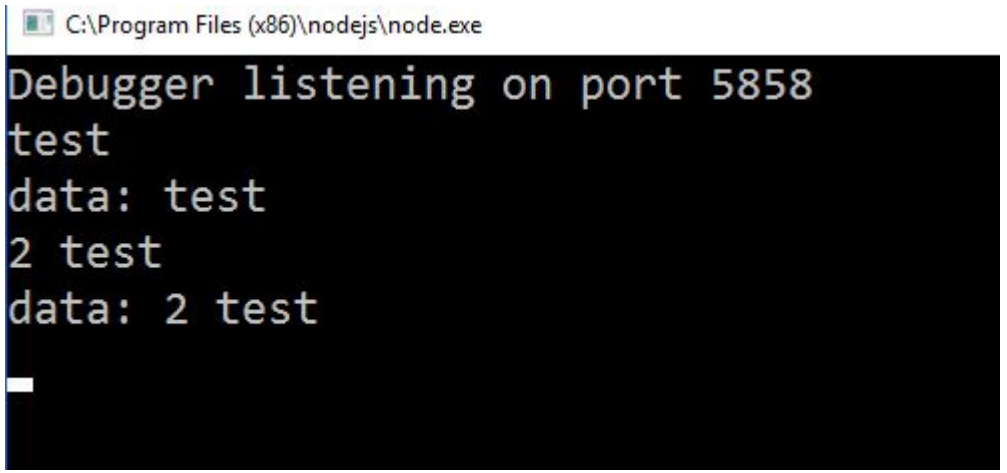
Процессы ввода/вывода

- Стандартные процессы ввода/вывода операционной системы -stdin, stdout и stderror - также имеют свое воплощение в объекте process.
- Метод process.stdin.resume(), возобновляет работу потока process.stdin (он по умолчанию приостановлен).
- Поток process.stderr представляет стандартный поток ошибок stderr. Следует принимать во внимание важное отличие его от предыдущих потоков, операция записи в него всегда является блокирующей.

Сервер эхо-печати

Сервер эхо-печати, возвращающий в консоль вводимый текст.

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

process.stdin.on('data', function (chunk) {
    if (chunk != "end\n") {
        process.stdout.write('data: ' + chunk);
    }
    else{
        process.kill();
    }
});
```



Signal Events

- У объекта Process, как и у всех порядочных javaScript-объектов, есть свои события, с которыми можно связать необходимые обработчики.
- Signal Events генерируются при получении процессом сигнала.
- Сигналы могут быть стандартные, POSIX-ов SIGINT, SIGUSR1, SIGTSTP и др.
- Note: Windows does not support sending signals, but Node.js offers some emulation with process.kill(), and ChildProcess.kill(). Sending signal 0 can be used to test for the existence of a process. Sending SIGINT, SIGTERM, and SIGKILL cause the unconditional termination of the target process.

Signal Events

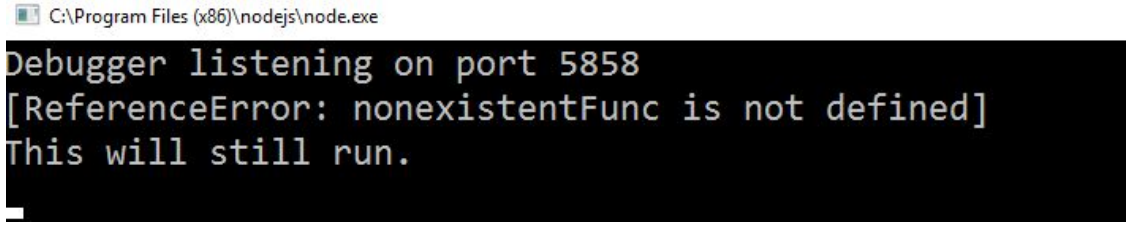
```
process.on('SIGHUP', function () {
    console.log('Got a SIGHUP');
});
setInterval(function () {
        console.log(' Running');
    }, 10000);
console.log('PID: ', process.pid);
```

Теперь в другой консоли, зная PID запущенного процесса, можно послать ему требуемый сигнал: kill -s SIGHUP 5772

Результат > ‘Got a SIGHUP’

UncaughtException handler

```
process.on('uncaughtException', function (err) {
    console.log(err);
});
setTimeout(function() {
    console.log('This will still run.');
}, 500);
// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```



Child Process

Вот некоторые возможности child _process:

позволяет запускать команды shell;

дает возможность запускать дочерние процессы, исполняемые. параллельно;

позволяет процессам обмениваться сообщениями.

Простой, но действенный метод этого модуля, child_process. exec(), позволяет запустить shell-команду на выполнение и сохранить результат в буфер. Ценность команды состоит в возможности дальнейших действий с полученными результатами.

Вот пример его работы:

```
var exec = require('child_process').exec;
exec('node -v', function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);
    if (error !== null) {
        console.log('exec error: ' + error);
    }
});
```

Child Process Spawn

- child_process.spawn() запускает команду в новом процессе, дескриптор которого становится доступным

```
var spawn = require('child_process').spawn
var cp = spawn('cmd', ['/c', 'dir']);
cp.stdout.on("data", function (data) {
    console.log(data.toString());
});

cp.stderr.on("data", function (data) {
    console.error(data.toString());
});

cp.on('close', function (code) { console.log('child process exited with code ' +
code); })
```

Spawn vs Exec

- Exec возвращает буфер из дочернего процесса, после его завершения. По умолчанию размер буффера 200K. После чего будет ошибка maximumBuffer exceded.
- Spawn возвращает поток данных немедленно после запуска дочернего процесса (используется когда дочерний процесс должен вернуть большой объем данных).

Fork

- Следующий метод child_process.fork() запускает дочерним процессом процесс, порожденный самой Node.js. Продемонстрировать его работу можно следующим простым кодом (файл main.js):

```
var cp = require('child_process');
var child1 = cp.fork('sub.js');
var child2 = cp.fork('sub2.js');

while (1)
{
    console.log("running main");
}
```

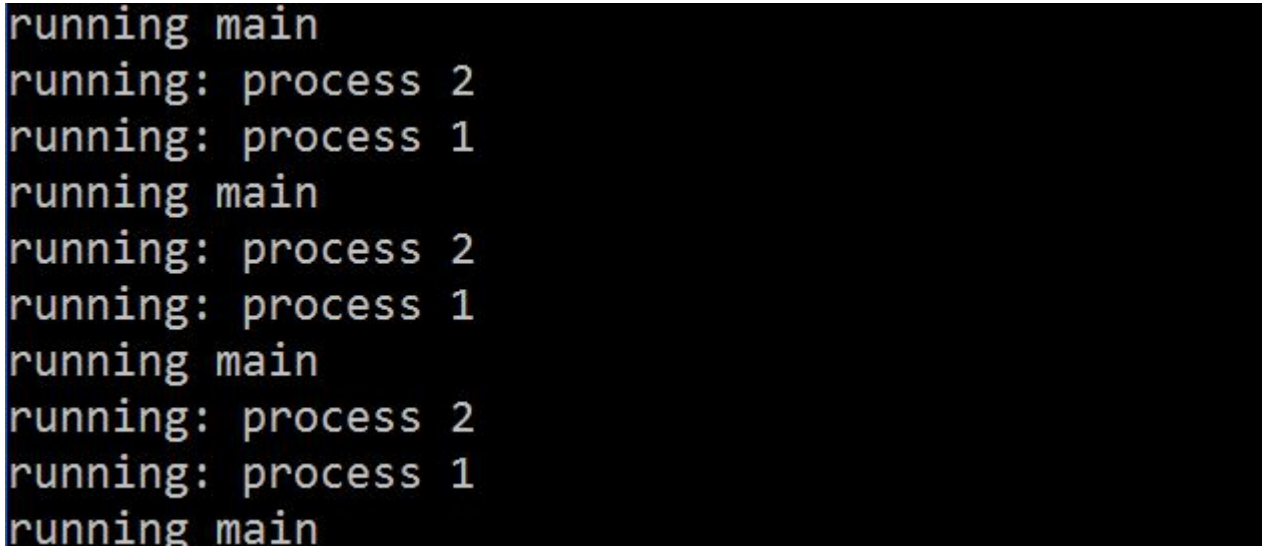
Fork

Sub.js

```
while (1) { console.log("running: process 1"); }
```

Sub2.js

```
while (1) { console.log("running: process 2"); }
```



Сообщения

```
const cp = require('child_process');
const n = cp.fork('sub.js');
n.on('message', function (m) {
    console.log('PARENT got message:', m);
});
n.send({ hello: 'world' });
//sub.js
process.on('message', function(m) {
    console.log('CHILD got message:', m);
});

process.send({ foo: 'bar' });
```

```
PARENT got message: { foo: 'bar' }
CHILD got message: { hello: 'world' }
```

Понятие буфера

- В любой системе, претендующей на роль серверной платформы в веб-среде, необходимы средства для полноценной работы с потоками двоичных данных, одним plain-текстом сыт не будешь. В классическом JavaScript подобные средства отсутствовали (если не считать недавно появившихся типов File API, ArrayBuffer, относящихся не к самому языку, а к объектной модели браузера). В Node.js для решения подобных задач существует объект Buffer. Бинарные данные хранятся в экземплярах этого класса, с ним ассоциирована область памяти, выделенная вне стандартной кучи V8.

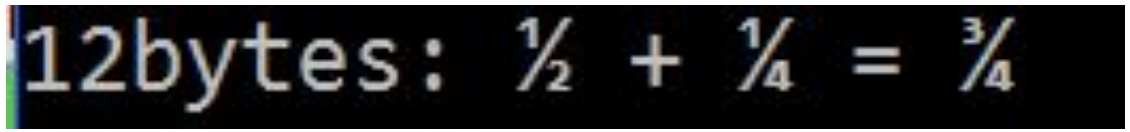
Кодировки

Node поддерживает следующие кодировки для строк:

- 'ascii' - только для 7-битных ASCII-строк. Этот метод кодирования очень быстрый, он сбрасывает старший бит символа;
- 'utf8' - Unicode-символы UTF-8;
- 'base64' -строка, закодированная в системе Base64;
- 'hex' -кодирует каждый байт как два шестнадцатеричных символа.

Работа с буфером

```
buf = new Buffer(256, 'utf8');
text = '\u00bd + \u00bc = \u00be';
len = buf.write(text); //buf.write(text, 0, text.length);
console.log(len + "bytes: " + buf.toString('utf8', 0,
len));
```



Важная особенность, призванная немного сохранить психику разработчиков, заключается в том, что, несмотря на то, что запись данных прекращается при превышении размера буфера, символы юникода не будут записаны «частично», по первому байту, или целиком, или никак.

toJSON

Для полного счастья программистов предусмотрен еще метод buffer.toJSON()

- `buf = new Buffer('test');`
- `console.log(buf.toJSON())`
- `console.log(buf);`
- `console.log(buf[3]);`

```
{ type: 'Buffer', data: [ 116, 101, 115, 116 ] }
<Buffer 74 65 73 74>
116
```

Buffer прочее

- Метод buffer.length(), возвращающий размер данных в буфере, имеет одну особенность - это общий объем зарезервированного пространства под данные, он может не совпадать с объёмом самих данных.
- Еще один способ задания буфера - непосредственная передача конструктору массива байтов (то есть восьмибитных данных):

```
buf  = new Buffer([01,02,03,04,05));
```

buffer.slice()

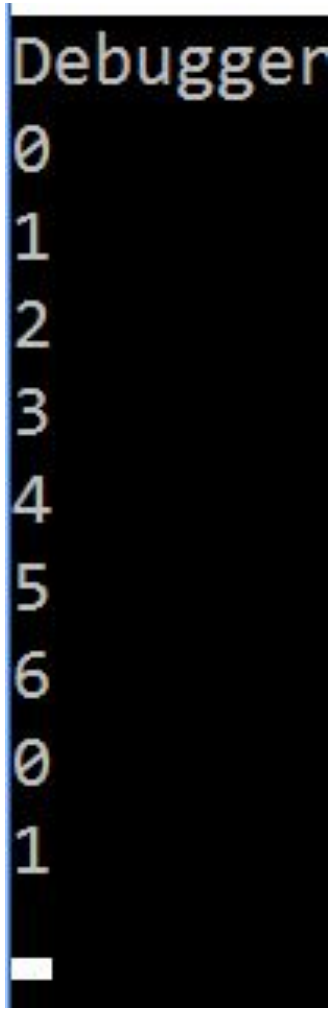
- Метод buffer.slice() возвращает новый буфер, представляющий собой срез старого. При этом надо понимать, что вновь сознанный объект указывает на ту же область памяти, что и предыдущий, соответственно, любые изменения нового буфера коснутся и буфера источника.

Таймеры

- Таймеры Node.js представлены несколькими жизненно необходимыми глобальными функциями, хорошо знакомыми по классическому JavaScript. Прежде всего это setTimeout(), позволяющая выполнить переданный ей в качестве аргумента код через заданное количество миллисекунд. Функция возвращает ID тайм-аута. clearTimeout() обнуляет счетчики по заданному идентификатору.

SetTimeout и ClearTimeout

```
var tid;
function toConsole(n)
{
    console.log(n);
    tid = setTimeout(toConsole, 1000, n + 1);
        if (n > 5) {
            clearTimeout(tid);
            toConsole(0);
        };
}
toConsole(0);
```



SetTimeout и ClearTimeout

- Тут следует обратить внимание на то, что все дополнительные аргументы setTimeout() превращаются в аргументы функции обратного вызова. То обстоятельство, что в консоль проникла цифра 6, объясняется тем, что таймер успевает отработать перед уничтожением. Впрочем, аналогичная задача решается без всякой рекурсии двумя другими таймер-функциями setInterval() и clearInterval(), устанавливающими и сбрасывающими (соответственно) так называемые интервальные таймеры, то есть таймеры, срабатывающие периодически, через заданный интервал.

SetInterval и ClearInterval

```
var tid;
function toConsole(){
    console.log(n); n++;
    if (n > 51) {
        clearInterval(tid);

    }
}
var n = 0;
tid = setInterval(toConsole, 10);
```

События

Обработка событий - основа работы с Node.js. События генерируют практически все объекты. В явном и неявном виде мы уже использовали их.

За события в Node.js отвечает специальный модуль - events. Назначать объекту обработчик события следует методом addListener(event, listener), аналогичным имеющемуся в обычном «браузерном» JavaScript. Аргументами для него служат имя события (строка, обычно в camelCase-стиле: connect, messages, messageBegin) и функция обратного вызова - обработчик события.

События

Для особо ленивых разработчиков, привыкших к удобствам jQuery, для этого метода существует синоним - просто on():

```
server.on('connection', function () {
console.log('connected!'); });
```

У метода on() есть чрезвычайно полезная модификация - once(), назначающая однократный обработчик события. То есть код

```
server.once('connection', function () {
console.log('connected!'); });
```

сработает только при первом соединении с сервером.

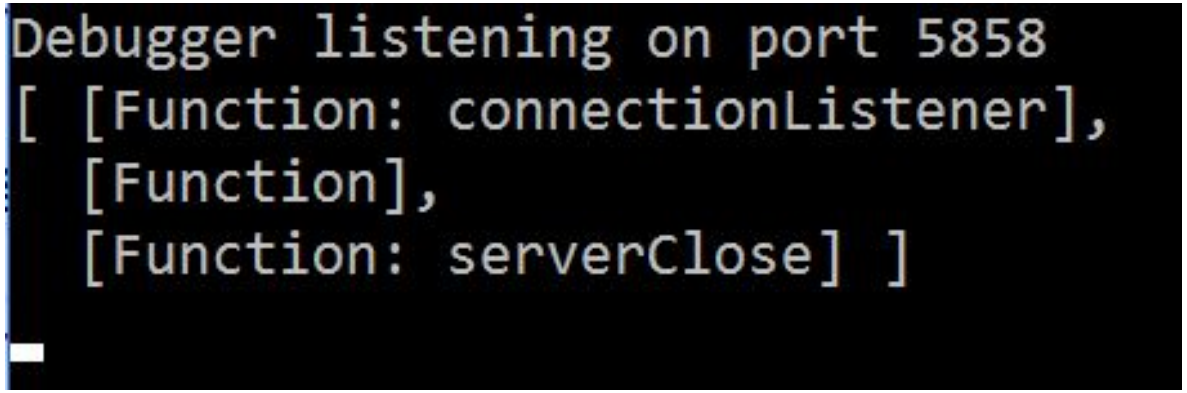
SetMaxListeners

- Теоретически с одним объектом можно связать сколько угодно обработчиков, но по умолчанию их количество ограничено 10. Это сделано для предотвращения утечек памяти. Ограничение преодолевается методом setMaxListeners(n), где n - требуемое максимально допустимое количество обработчиков.

Метод listeners

Посмотреть все обработчики объекта, связанные с конкретным событием, можно с помощью метода listeners:

```
var http = require('http');
var server = http.createServer(function (request, response) {
}).listen(8080);

function serverClose() { server.close() };
server.on('connection', function ()
{
    console.log('Connected ! ');
});
server.on('connection', serverClose);
console.log(server.listeners('connection'));
```



RemoveListener

- Удалить обработчик можно можно методом removeListener(event, Listener). Как видно из сигнатуры метода, желательно, чтобы функция-обработчик была именована или присвоена именованной переменной

```
var callback = function () {
   console.log('Connected! '); }
server.on('connection', callback); // ...
server.removeListener('connection', callback);
```

Emit

- Наконец, метод emit(event, [args]) позволяет назначенным обработчикам срабатывать, как если бы связанное событие случилось. Причем событие, переданное emit(), не обязательно должно вообще существовать.

```
var http = require('http');
var util = require('util');
server.on('someevent', function (arg) {
console.log('event ' + arg); });
server.emit('someevent', '! ! !');
console.log(util.inspect(server.listeners('someevent')));
```

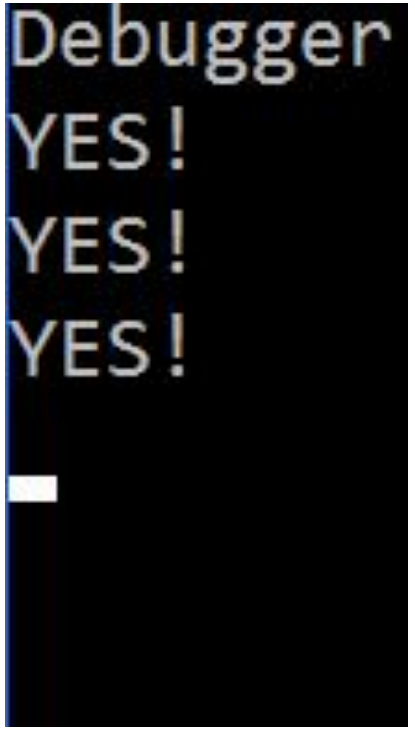
```
Debugger listening on port 5858
event ! ! !
[ [Function] ]
```

Объект EventEmitter

- EventEmitter - это основной объект, реализующий работу обработчиков событий в Node.js. Любой объект, являющийся источником событий, наследует от класса EventEmitter, и, все методы, о которых мы говорили, принадлежат этому классу.
- Оперировать событиями посредством EventEmiter можно и напрямую, явным образом, создав объект этого класса.

Объект EventEmitter

```
var EventEmitter = require('events').EventEmitter;
var emiter = new EventEmitter();
emiter.on('myEvent', function (ray) {
console.log(ray); });
setInterval(function (){
    emiter.emit('myEvent', 'YES!');
}, 1000);
```



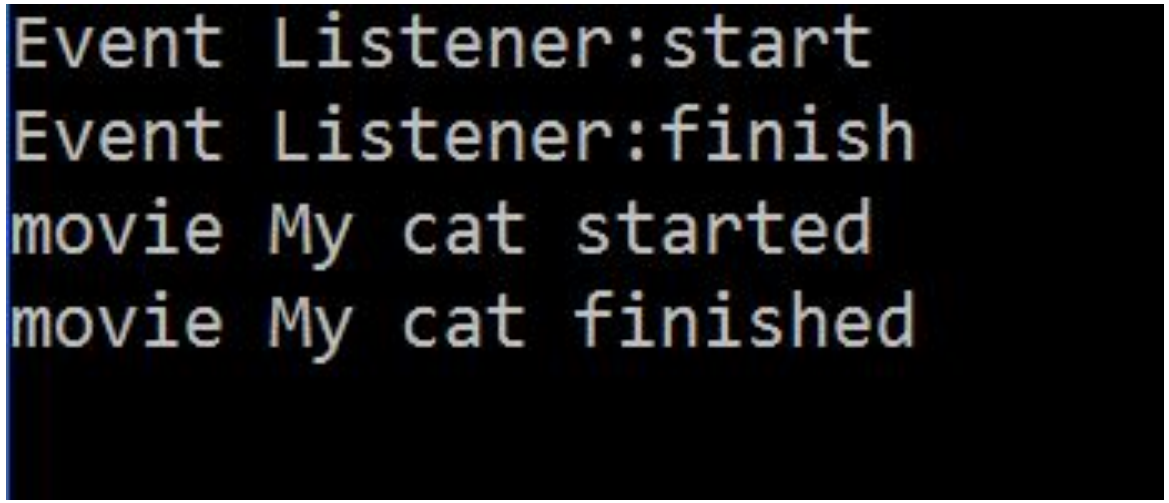
Наследование от EventEmiter

Для того чтобы добавить методы EventEmitter к произвольному (например, созданному нами) объекту, достаточно унаследовать EvetntEmitter с помощью метода inherits из модуля utils:

```
var util = require("util");
var EventEmitter = require('events').EventEmitter;
var VideoPlayer = function (movie) {
    var self = this;
    setTimeout(function () { self.emit('start', movie); }, 0);
    setTimeout(function () { self.emit('finish', movie); }, 5000);
    this.on('newListener',
        function (listener) {
            console.log('Event Listener:' + listener);
        });
}
```

Наследование от EventEmiter

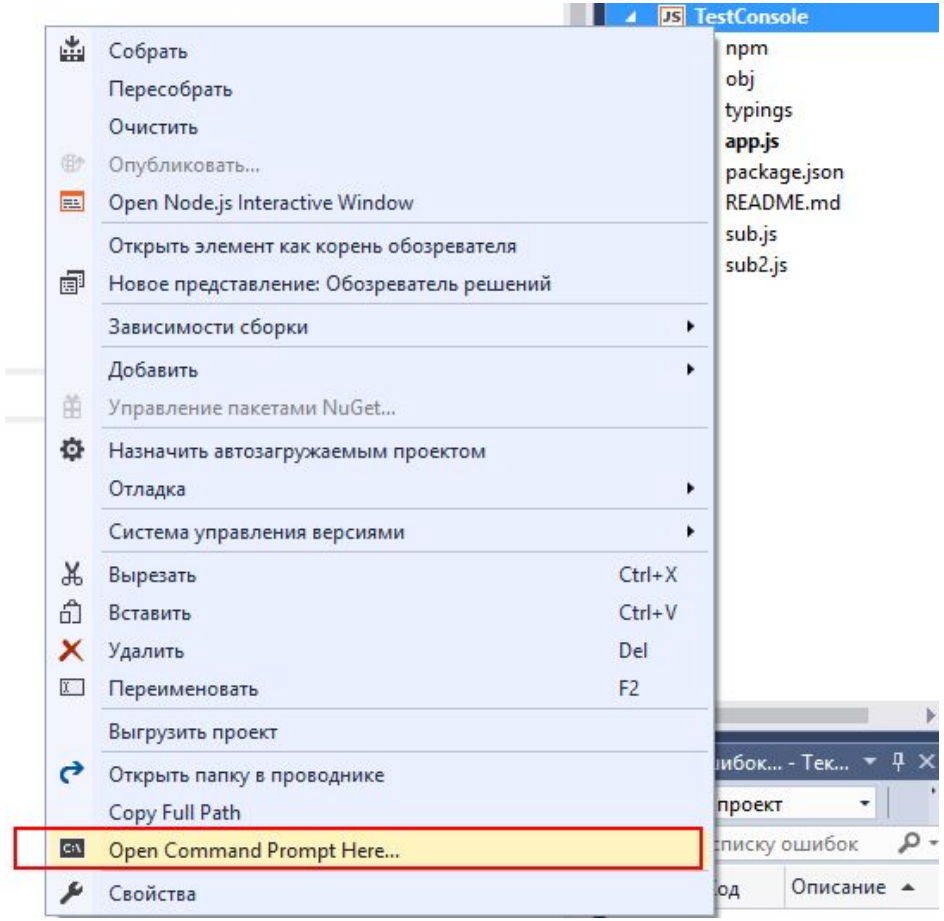
```
util.inherits(VideoPlayer, EventEmitter);

var movie = { name: 'My cat' };
var myPlayer = new VideoPlayer(movie);

myPlayer.on('start', function (movie)
    { console.log('movie ' + movie.name + ' started'); });
myPlayer.on('finish', function (movie)
    { console.log('movie ' + movie.name + ' finished'); });
```



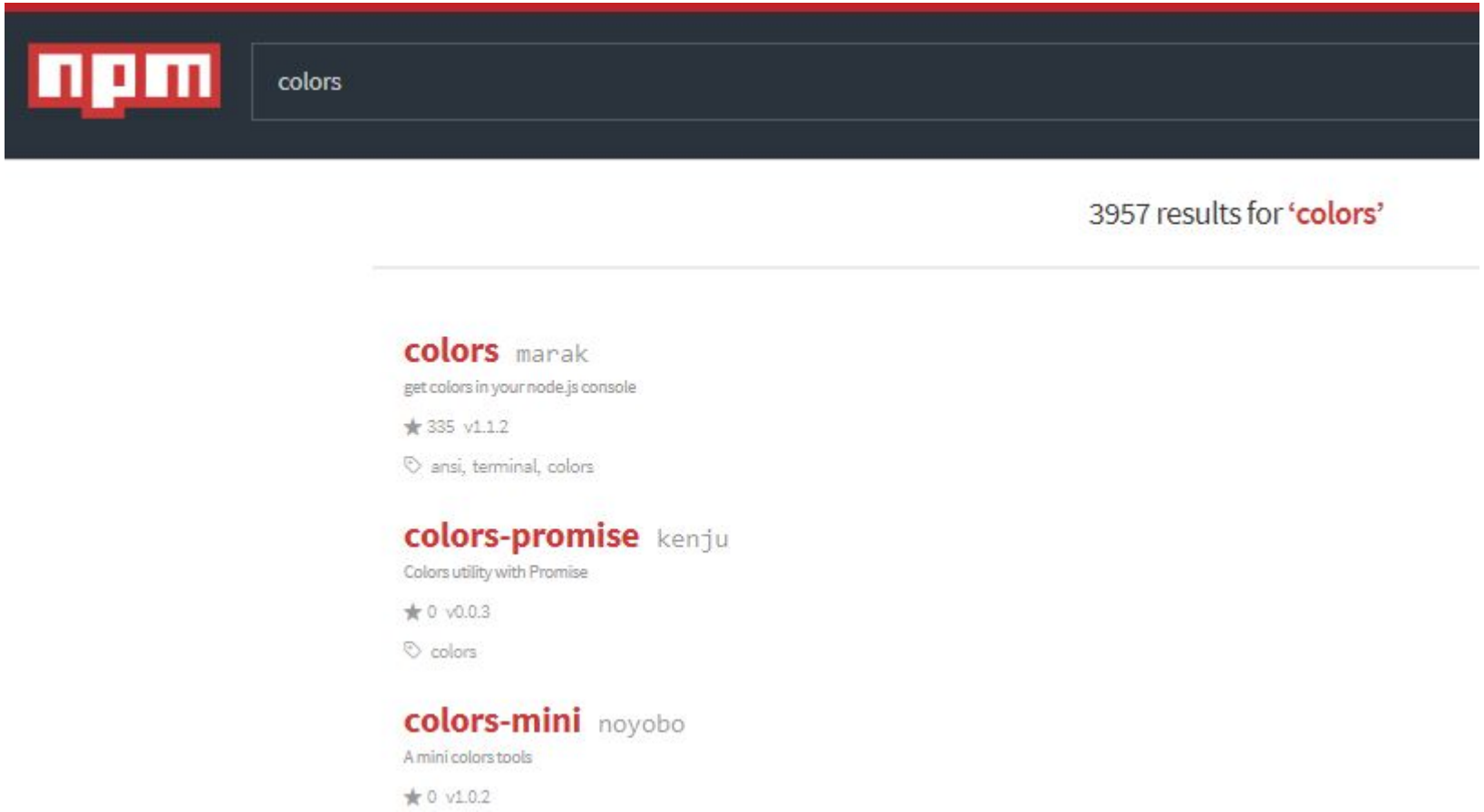
Модули

- Да, все (весьма впечатляюще) возможности платформы Node.js реализованы (и продолжают реализовываться) в модулях. Мы уже неоднократно применяли этот механизм расширений - модули util, http, fs, event и др. Кроме набора модулей, входящего в стандартную поставку Node.js, существует великое множество расширений, реализующих самую разную функциональность. Это могут быть модули для работы с разными базами данных, с протоколом WebSockets, с Apache Hadoop или хэш-таблицами memcached. Замечательный MVC-фреймворк Express, фреймворк Connect, шаблонный движок Jade, все остальные инструменты Node.js - все это тоже модули.
- Водятся модули на просторах репозитория GitHub.

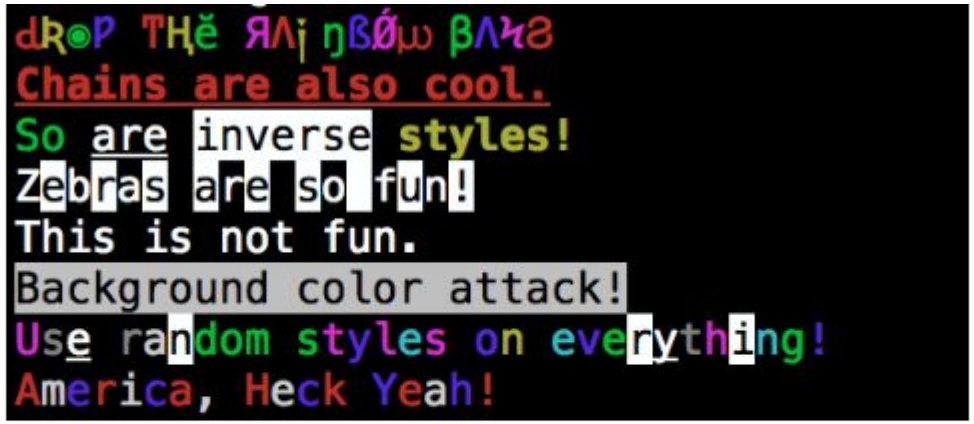
Command Prompt



Поиск пакета https://www.npmjs.com/



Страница пакета



Installation

```
npm install colors
```

Установка

```
stConsole>npm install colors
npm WARN package.json test-console@0.0.0 No repository field.
colors@1.1.2 node_modules\colors
```

```
stConsole>npm ls
test-console@0.0.0 D:\Develop
ject\TestConsole\TestConsole
└── colors@1.1.2 extraneous
```

Использование

```
var colors = require('colors');
console.log('rainbows rasing!'.rainbow);
console.log('background color!'.grey.blueBG)
console.log(colors.bold(colors.red('Chains are also
cool ... ')));
```

Создаем собственный модуль

выделим наш объект в отдельный файл (band.js):

```
var band = function (name) { this.name = name; };
band.prototype.getName = function () {
console.log(this.name); }
```

И подключим его с помощью уже хорошо знакомого нам метода require:

```
require('./band.js');
var myBand1 = new band("The Beatles");
var myBand2 = new band("The Rolling Stones");
myBand1.getName();
myBand2.getName();
```

Создаем собственный модуль

- Все? Нет, не все. Такая конструкция работать не будет. Мы уже упоминали, что, в отличие от подключаемых скриптов на веб-странице, код на Node.js не имеет глобальных объектов.
- Можно использовать уже знакомый нам объект global, переведя band в данное пространство имен.

```
var band = function (name) { this.name = name; };
band.prototype.getName = function () {
console.log(this.name); }
global.band = band;
```

Создаем собственный модуль

- Так все будет работать, но мы лишаем себя одного из преимуществ модульности - использования собственного пространства имен.
- Node.js предлагает способ лучше -метод exports() глобального объекта module, наследником которого автоматически стал наш модуль.

```
var band = function (name) { this.name = name; };

band.prototype.getName = function () { console.log(this.name); }

exports.band = band;
```

Создаем собственный модуль

Можем пользоваться всеми преимуществами изолированного пространства имен:

```
var item = require('./band.js');
var myBand1 = new item.band("The Beatles");
var myBand2 = new item.band("The Rolling Stones");

myBand1.getName();
myBand2.getName();
```

Выводы

- Были рассмотрены принципы работы Event loop в Node.js.
- Переменные объявленные на верхнем уровне автоматически не становятся глобальными.
- Были рассмотрены процессы и работа с ними, отправка сообщений, порождение дочерних процессов с помощью методов exec, spawn, fork.
- Бинарные данные хранятся в экземплярах класса Buffer, с ним ассоциирована область памяти, выделенная вне стандартной кучи V8.
- Для работы с таймерами используются методы SetTimer, ClearTimer, SetInterval, ClearInterval.
- За события в Node.js отвечает специальный модуль – events.
- EventEmitter - это основной объект, реализующий работу обработчиков событий в Node.js. Любой объект, являющийся источником событий, наследует от класса EventEmitter.
- На базовом уровне рассмотрена работа с модулями.

В следующей лекции

- Работа с файлами
- **Создаем TCP-сервер**
- **IncomingMessage -входящий HTTP-запрос**
- **Server Response**
- **WebSockets**
- **Node.js control-flow**
- Async - берем поток исполнения в свои руки
- Node.js и данные. Базы данных .

Список литературы

- Сухов К. К. Node.js. Путеводитель по технологии. - М.: ДМК Пресс, 2015. 416 с.: ил.
- http://www.codingdefined.com/2014/08/difference-between-fork-spawn-and-exec.html
- https://nodejs.org/api/child_process.html
- https://ru.wikipedia.org/wiki/Node.js
- Пауэрс Ш. Изучаем Node.js. - СПб.: Питер, 2014. - 400 с: ил. - (Серия "Бестселлеры O'Reilly").