***Приёмы работы с модулями***

Нашей следующей темой будет изучение объекта module.

Объект module – это основополагающий объект в понимании модулей. В нем есть множество важных свойств, которые понадобятся нам в более сложных сценариях поведения, к которым мы с вами скоро перейдем.

Объект module является переменной, которая существует в каждом модуле. Выведем ее.

console.log(module)

Итак, запускаем. Переменный модуль есть в каждом файле и содержит информацию об объект данного модуля, который, по мере того, как Node.js обрабатывает файл, постепенно заполняется. Кратко пройдемся по его основным свойствам.

1. **id**

id, как правило, содержит полный путь к файлу. Если операционная и файловая системы поддерживают символические ссылки, то все они будут здесь отрисованы. Если вы не знаете, что такое символические ссылки, то ничего страшного. Непонимание здесь некритично. id используется внутри Node.js. Как правило, мы сами его использовать не будем.

1. **exports**

Это то, что выдается наружу.

1. **parent**

Это ссылка на родительский модуль, который рекваерит данные.

1. **filename**

Это полное имя файла с учетом адресловленных ссылок.

1. **loaded**

Показывает, загрузился ли модуль. На момент, когда мы вводим в *console.log(…)* модуль, этот модуль до конца еще не обработан, поэтому *loaded: false.*

1. **children**

Это те модули, которые данный модуль подключил через require. В данном случае только один модуль, который тоже загрузился *loaded: true*.

1. **paths**

Это тоже внутренняя переменная, как и id. Мы ее не будем использовать. Я расскажу о ней немного позже, когда мы будем разбирать порядок поиска модулей с учетом путей.

1. Из этого обширного списка нам важны два свойства, которые на практике используются более часто. Это свойства **parent** и **exports.** Поэтому о них мы поговорим более подробно.

Первый, можно сказать, разминочный прием, который мы с вами обсудим, это использование module.parent. Бывает так, что какой-то js файл может работать, как режим прямого запуска через Node, например, Node server.js, так и в качестве модуля для чего-то другого. Например, модуль server.js будет запускать этот функционал, только в том случае, если он запущен явно.

|  |  |  |
| --- | --- | --- |
| var user = require('./user'); | | |
|  | | |  |
| var vasya = new user.User("Вася"); | | |
| var petya = new user.User("Петя"); | | |
|  | | |
| vasya.hello(petya);  А если нет? Если какой-то другой модуль его подключил, то пусть он этот функционал экспортирует. Разделить эти два случая можно при помощи проверки.  if (module.parent) {  Если *module.parent* есть – это означает, что server.js кто-то подключил. В этом случае имеет смысл экспортировать весь этот функционал, например, заключив его в функцию *run*. Так мы и сделаем. | | |
| var user = require('./user'); | | |
|  |  | |
| function run() { | |
| var vasya = new user.User("Вася"); | |
| var petya = new user.User("Петя"); | |
|  | |
| vasya.hello(petya); | |
| } | |
|  | |
| if (module.parent) {  exports.run = run;  А если модуля *parent* нет – это означает, что сервер запущен сам по себе. В этом случае, давайте запустим эту функцию прямо сейчас.   |  | | --- | | var user = require('./user'); | |  |  | | | function run() { | | | var vasya = new user.User("Вася"); | | | var petya = new user.User("Петя"); | | |  | | | vasya.hello(petya); | | | } | | |  | | | if (module.parent) { | | | exports.run = run; | | | } else { | | | run(); | | | } | | | |

Проверяем. Он работает? Работает! Допустим, я сделаю новый модуль, назову его *app.js.* подключу в нем сервер.

var server = require('./server');

Попробую запустить node app.js. Обратите внимание, ничего такого не произошло. То есть, *user* она подключила, что происходит в любом случае, но *run* не запустилась, потому что есть *module.parent.* Он работает только в режиме модуля. Уберу лишний *console.log*

console.log("user.js is required!");

Соответственно, если в app.js я захочу вызвать сервер, я его вызову.

server.run();

Как правило, такой прием используется в тех случаях, когда пишется какая-то консольная утилита или какое-то независимое приложение, которое может работать, как часть чего-то другого в том числе.

Следующий прием по работе с модулями, который мы изучим, касается правильного использования *module.exports*.

Когда мы записываем свойства в *exports* для того, чтобы вынести из модуля, на самом деле, мы пишем их в *module.exports.* Это и есть то самое истинное свойство объекта модуль, которое наружу выходит, а *exports* и *this* в контексте модуля являются ссылками на него.

// *module.exports = exports = this*

Поэтому, я могу написать

exports.User = User;

а могу написать

this.User = User;

Разницы не будет. Могу написать полностью

module.exports.User = User;

Обычно используют *exports* по двум причинам. Первая – это то, что короче, чем *module.exports,* а второе – это то, что *this* еще короче, но он менее универсален, потому что *this* на уровне модуля – это то же самое, что и *exports,* а вот *this* функции уже будет другим. Поэтому для унификации используют везде обычно *exports.* А по какой-либо причине используют *module.exports.* А какая эта особая причина? Если внимательно посмотреть на код, то можно увидеть некую несуразность.

|  |
| --- |
| var user = require('./user'); |
|  |  | |
| function run() { | |
| var vasya = new user.User("Вася"); | |
| var petya = new user.User("Петя"); | |
|  | |
| vasya.hello(petya); | |
| } | |
|  | |
| if (module.parent) { | |
| exports.run = run; | |
| } else { | |
| run(); | |
| } | |

На самом деле, мы из этого модуля хотим наружу выдать только функцию *user.*

function User(name){

Мы не хотим выдать что-то еще, поэтому объект нам не нужен. Нам бы хотелось так:

var User = require('./user'); - получили функцию, использовали

|  |
| --- |
| var vasya = new User("Вася"); |
| var petya = new User("Петя"); |

без всякого промежуточного объекта. Это возможно, если записать напрямую:

module.exports = User;

Экспортируемый объект как раз и будет нашей функцией *user.* Обратите внимание, записывать только так, а в случае записи:

exports = User; - было бы невозможно, это не работает, потому что у нас есть *module.exports,* а *exports* – это всего лишь ссылка на него. Если заменить эту ссылку, то на *module.exports* это не повлияет. Так работают объекты в JavaScript. Если я меняю что-то по ссылке, то оно изменится внутри объекта, а если я заменяю саму ссылку, то ничего не произойдет с другими ссылками. Итак, проверяем. Все хорошо! Наш код стал еще проще.

|  |
| --- |
| var user = require('./user'); |
|  |
| function run() { |
| var vasya = new User("Вася"); |
| var petya = new User("Петя"); |
|  |
| vasya.hello(petya); |
| } |
|  |
| if (module.parent) { |
| exports.run = run; |
| } else { |
| run(); |
| } |

Следующим этапом мы добавим к проекту базу данных. Пока что это будет неполноценная база данных, о ней мы поговорим позже, а просто на уровне структуры поговорим, как это реализуется и почему работает. База данных будет файлом *db.js*. Здесь у нас для базы данных есть прямой кандидат – это фразы *ru.json.* Сразу поместим фразы в базу данных, для этого я сделаю каталог *db*, и здесь уже будет не *db.js,* а *index.js*.

Что есть у объекта базы? Есть метод *connect*

exports.connect = function() {

При вызове этого метода пусть она загружает фразы

|  |
| --- |
| phrases = require('./ru'); |
| }; |

Сейчас это просто так, а в реальной жизни это будет подсоединено к базе данных, из которой потом можно будет делать запросы. Как? Пусть у нас будет метод

|  |
| --- |
| exports.getPhrase = function(name) { |
| if (!phrases[name]) { |
| throw new Error("Нет такой фразы: " + name); |
| } |

Этот метод будет возвращать соответствующую фразу, а если нет, то выдавать ошибку.

Теперь используем эту базу в объекте *user.* Фразы теперь будут получатьcя из базы данных.

var phrases = require('../db');

База данных на уровень выше находится. И вместо *phrases* в строчке с Hello мы делаем

console.log(db.getPhrases("Hallo") + ", " + who.name);

Вот так. Запускаем. Ошибка вышла. Какая ошибка? Ага, я не загрузил ничего из базы. Давайте загружу.

db.connect();

Действительно, когда мы обращаемся к базе, нужно сначала *connect*, а потом все остальное. Запускаю еще раз. И сейчас уже все хорошо. А теперь, так как эта база глобальная для всего нашего проекта, давайте ею воспользуюсь и в сервере тоже. Подключаю объект базы

var db = require('../db');

в server.js и выведу

console.log(db.getPhrases("Run successful"));

Успешный запуск. Добавлю эту фразу в базу. Теперь несколько слов о том, как это все будет работать. Когда Node.js первый раз загружает модуль, он полностью создает соответствующий объект *module* с учетом *parent, exports* и аналогичных свойств. Запоминает его у себя. Модуль id, тот, который является обычно полным путем к файлу, служит идентификатором для внутреннего кеша. Node.js как бы запоминает: файл такой-то, для него создан объект модуля такой-то. В следующий раз, когда мы получаем тот же файл, а файл, по сути дела, одинаковый, просто пути разные к нему, но абсолютный путь одинаковый. Node.js обращается к кешу и берет все тот же объект. Получается, что в *server.js* и *index.js* будет использован один и тот же объект базы данных. Соответственно, прием здесь такой: первый раз, когда подключается модуль, он инициализуется. В данном случае используется *connect* в базе данных. В дальнейшем модуль уже инициализован, поэтому, просто его берем и пользуемся.

Теперь сделаем следующий логичный шаг. Дело в том, что в *server.js* *db* мы получаем из текущей директории, а в *index.js* из родительской:

var db = require('../db');

Давайте подумаем, что произойдет, когда мы *user* будем развивать, и у директора *user* будут поддиректории. Получается, что в *index.js* я делаю *require* вот так:

var db = require('../db');

а в поддиректории мне надо будет делать *require* вот так:

var db = require('../../db');

А если я буду переносить файл, то мне нужно уследить, чтобы пути автоматически правильно обновлялись. Конечно, ide нам поможет в этом, но тем не менее.

Как бы нам сделать такую простую штуку, чтобы база подключалась просто так? Зачем мне указывать явно путь к базе, если мы знаем, что имеется в виду одна, которая в корне, главная? Для того, чтобы так сделать, нам нужно понимать порядок поиска модулей в Node.js. Для этого обратимся к документации. Ищем *node module*. И первая ссылка даст нам документацию *modules.html* по модулям в Node.js. Здесь, в дальнейшем, я буду по возможности именно комментировать документацию, которая уже есть, разъяснять всякие тонкие моменты, нежели, чем писать свои какие-то выводы. На документацию нужно ориентироваться, ее нужно понимать. В данном случае, здесь есть описание ряда свойств, о которых мы говорили. Нас интересует порядок поиска модуля.

require(X) from module at path Y

1. If X is a core module,

a. return the core module

b. STOP

2. If X begins with './' or '/' or '../'

a. LOAD\_AS\_FILE(Y + X)

b. LOAD\_AS\_DIRECTORY(Y + X)

3. LOAD\_NODE\_MODULES(X, dirname(Y))

4. THROW "not found"

LOAD\_AS\_FILE(X)

1. If X is a file, load X as JavaScript text. STOP

2. If X.js is a file, load X.js as JavaScript text. STOP

3. If X.json is a file, parse X.json to a JavaScript Object. STOP

4. If X.node is a file, load X.node as binary addon. STOP

LOAD\_AS\_DIRECTORY(X)

1. If X/package.json is a file,

a. Parse X/package.json, and look for "main" field.

b. let M = X + (json main field)

c. LOAD\_AS\_FILE(M)

2. If X/index.js is a file, load X/index.js as JavaScript text. STOP

3. If X/index.json is a file, parse X/index.json to a JavaScript object. STOP

4. If X/index.node is a file, load X/index.node as binary addon. STOP

LOAD\_NODE\_MODULES(X, START)

1. let DIRS=NODE\_MODULES\_PATHS(START)

2. for each DIR in DIRS:

a. LOAD\_AS\_FILE(DIR/X)

b. LOAD\_AS\_DIRECTORY(DIR/X)

NODE\_MODULES\_PATHS(START)

1. let PARTS = path split(START)

2. let I = count of PARTS - 1

3. let DIRS = []

4. while I >= 0,

a. if PARTS[I] = "node\_modules" CONTINUE

c. DIR = path join(PARTS[0 .. I] + "node\_modules")

b. DIRS = DIRS + DIR

c. let I = I - 1

5. return DIRS

Вот так. Это то, что происходит, когда вызывают *require.* Скорее всего, если вы видите это первый раз, то не очень понятно. Я постараюсь разъяснить. Итак, *require* модуль. Вообще в Node.js много встроенных модулей, например, модуль по работе с файловой системой *fs.* Если есть такой встоенный модуль, то *require* сработает тут же и все, готово.

Если я указал путь к *require,* в данном случае это ('./db') скажем

var db = require('./db');

тогда Node.js поищет файл по этому пути, попытается либо найти данный файл, либо попытается получить этот файл, как директорию. Здесь есть упоминание package.json, мы позже поговорим про это. В нашем случае он возьмет *db index.js*, это и будет файлом модуля. Ну, и наконец, третий пункт LOAD\_NODE\_MODULES(X, dirname(Y)). Он сработает по алгоритму ниже только в том случае, если я не указал путь, и, при этом, это не встроенный модуль. Тогда Node.js будет его искать. Но как? Есть специальное название директории, которое называется "node\_modules". Он поищет эту директорию сначала в текущем местоположении, то есть, в *server.js,* вот такую – *start node\_modules*. Если он ее найдет, то попытается взять модуль из нее. Проверим. Запускаю. Все сработало. Оно ее нашло в директории *node\_modules.*

Либо, если в директории node\_modules нету, то оно поищет директорию *node\_modules* выше. Там не найдет – ищет еще выше и т.д. Помните, мы смотрели module pevs? Модуль певс – это как раз те пути, по которым она будет искать. Иначе говоря, это просто текущий путь и все выше него. Это нужно затем, что можно делать директорию *node\_modules*, в нее ставятся внешние пакеты, внешние модули, и потом они доступны. Эта директория может быть выше, выше, на любом уровне. Может быть несколько директорий – первая найденная, где этот модуль есть, послужит точкой остановки. Таким образом, я могу использовать вот такой вот путь, если *db* есть в *node\_modules*.

А что если я не хочу помещать ее в *node\_modules?* Зачем усложнять? Обычно *db, node\_modules* совершенно ни к чему. Можно сделать и так, потому что после того, как оно поищет все эти *node\_modules* и ничего не найдет, оно использует еще одно место для поиска. Это переменная NODE\_PATH. В ней можно указать несколько путей, по которым оно еще будет искать. В данном случае, давайте я укажу. Я переместил *db* и запущу. Не срабатывает, не находит модуль. Но если указать

NODE\_PATH = . переменное окружение.

Если вы работаете под Windows, то именно окружение ставится при помощи *set*-команды, если вы с командной строки:

Windows:

>set NODE\_PATH = .

>node server.js

Либо переменное окружение вы можете поправить в настройках вашего файлового менеджера или с места, откуда вы запускаете. Соответственно NODE\_PATH = . Все хорошо. Потому что текущий путь добавился в список тех, по которым происходит поиск. И, наконец, кроме NODE\_PATH, по историческим причинам происходит поиск еще в таких директориях:

* 1: $HOME/.node\_modules
* 2: $HOME/.node\_libraries
* 3: $PREFIX/lib/node

Вообще, на это можно не ориентироваться, так как в реальной жизни мы этим пользоваться не будем, нам это не нужно. Когда-то давно было по-другому, и с того времени остались эти пути.

Итак, что мы получили? Мы получили следующие дополнительные приемы работы с модулями.

Кеширование модулей.

Подключаем модуль один раз, инициализуем и в дальнейшем пользуемся уже объектом. То есть, заново файл модуля никогда не читается. Хотя, существуют такие специальные команды, которые позволяют сделать пустым кеш Node.js, то есть, убрать модуль с кеша. Можно поэкспериментировать, если хотите, в документации. Но обычно этого никто не делает. Итак, просчитали модуль, закешировали, идем дальше.

Если мы хотим, чтобы модуль у нас был глобальным, то есть, искал себе пути, то он должен быть в *node\_modules,* либо по NODE\_PATH должен искаться.

Следующий прием, который мы рассмотрим, называется модуль-фабрика. Он используется для того, чтобы передавать модулю параметры. Посмотрим этот прием на практическом примере, а именно в процессе подключения логгера к нашему приложению. Логгер – это отдельный модуль, который мы назовем logger.js. Мы ничего пока сюда записывать не будем, а сосредоточимся на том, что нужно для того, чтобы им пользоваться. А именно:

var log = require('logger')

Что мы хотим? Мы хотим, чтобы, когда я вызываю log,то вводилась соответствующая строка, и перед ней было название модуля, который ее вызывает:

log(db.getPhrase("Hello") + ", " + who.name);

Соответственно получается, для того, чтобы мне логгер выводил здесь именно название текущего модуля, мне нужно ему передать текущий модуль. Давайте я это и сделаю. Могу передать через запятую:

var log = require('logger',module);

Я могу получить логгер и, таким образом, создать нужную функцию. То есть, внутри логгера я делаю следующее:

module.exports = function(module) {

function(module) - фабричная функция, которая получает название модуля, который нужно логгировать и который, исходя из этого модуля, делает функцию логгер.

return function(/\* ... \*/) {

Эта функция получает какие-то параметры, и давайте она будет передавать их *console.log:*

console.log.apply(console, args);

Конечно, файлы можно вводить в базу и куда угодно. Мы передадим все аргументы в *console.log.* Давайте, не просто их передадим, а еще кое-что добавим:

var args = [module.filename].concat([].slice.call(arguments));

Вначале мы прибавили имя файла и передали все аргументы в консоль. Давайте проверим, работает ли оно. Сейчас я сделаю разбивку по вертикали. Запускаю. Теперь каждый модуль мы видим:

|  |
| --- |
| // var log = require('logger')(module); |
|  |  | |
| module.exports = function(module) { | |
|  | |
| return function(/\* ... \*/) { | |
| var args = [module.filename].concat([].slice.call(arguments)); | |
| console.log.apply(console, args); | |
| }; | |
| }; | |

Можно, конечно же, прибавлять не названия модуля, а, например, только последние два элемента. Поле для экспериментов тут большое. В дальнейшем мы поговорим о некоторых уже готовых логгерах, которые уже можно поставить и использовать.

Итак, на этом занятии мы поговорили о различных приемах работы с модулями: о том, что такое объект module; как запустить модуль в различных режимах приложения или компонента; как экспортировать то, что нам надо, а не обязательно объект; как работает кеширование модулей (кстати, именно это кеширование позволяет избежать глобальных перемен); как происходит поиск модулей и как реализуется передача параметров модулей при помощи модуль-фабрики.

На нашем следующем занятии мы поговорим о пакетном менеджере NPM и обсудим некоторые модули, которые используем в дальнейшем в процессе разработки.