Mastering Node

Node is an exciting new platform developed by *Ryan Dahl*, allowing JavaScript developers to create extremely high performance servers by leveraging Google's V8 JavaScript engine, and asynchronous I/O. In *Mastering Node* we will discover how to write high concurrency web servers, utilizing the CommonJS module system, node's core libraries, third party modules, high level web development and more.

Installing Node

In this chapter we will be looking at the installation and compilation of node. Although there are several ways we may install node, we will be looking at homebrew, nDistro, and the most flexible method, of course - compiling from source.

Homebrew

Homebrew is a package management system for *OSX* written in Ruby, is extremely well adopted, and easy to use. To install node via the `brew` executable simply run:

```
$ brew install node.js
```

nDistro

nDistro is a distribution toolkit for node, which allows creation and installation of node distros within seconds. An *nDistro* is simply a dotfile named *.ndistro* which defines module and node binary version dependencies. In the example below we specify the node binary version *0.1.102*, as well as several 3rd party modules.

```
node 0.1.102
module senchalabs connect
module visionmedia express 1.0.0beta2
module visionmedia connect-form
module visionmedia connect-redis
module visionmedia jade
module visionmedia ejs
```

Any machine that can run a shell script can install distributions, and keeps dependencies defined to a single directory structure, making it easy to maintain an deploy. nDistro uses pre-compiled node binaries making them extremely fast to install, and module tarballs which are fetched from GitHub via *wget* or *curl* (auto detected).

To get started we first need to install nDistro itself, below we *cd* to our bin directory of choice, *curl* the shell script, and pipe the response to *sh* which will install nDistro to the current directory:

```
$ cd /usr/local/bin && curl http://github.com/visionmedia/ndistro/raw/master/install | sh
```

Next we can place the contents of our example in *./.ndistro*, and execute *ndistro* with no arguments, prompting the program to load the config, and start installing:

```
$ ndistro
```

Installation of the example took less than 17 seconds on my machine, and outputs the following *stdout* indicating success. Not bad for an entire stack!

```
... installing node-0.1.102-i386
... installing connect
... installing express 1.0.0beta2
... installing bin/express
... installing connect-form
... installing connect-redis
... installing jade
... installing bin/jade
... installing ejs
... installation complete
```

Building From Source

To build and install node from source, we first need to obtain the code. The first method of doing so is via `git`, if you have git installed you can execute:

```
$ git clone http://github.com/ry/node.git && cd node
```

For those without *git*, or who prefer not to use it, we can also download the source via *curl*, *wget*, or similar:

```
$ curl -# http://nodejs.org/dist/node-v0.1.99.tar.gz > node.tar.gz
$ tar -zxf node.tar.gz
```

Now that we have the source on our machine, we can run `./configure` which discovers which libraries are available for node to utilize such as *OpenSSL* for transport security support, C and C++ compilers, etc. `make` which builds node, and finally `make install` which will install node.

```
$ ./configure && make && make install
```

Globals

As we have learnt, node's module system discourages the use of globals; however node provides a few important globals for use to utilize. The first and most important is the `process` global, which exposes process manipulation such as signalling, exiting, the process id (pid), and more. Other globals, such as the `console` object, are provided to those used to writing JavaScript for web browsers.

console

The `console` object contains several methods which are used to output information to *stdout* or *stderr*. Let's take a look at what each method does:

console.log()

The most frequently used console method is `console.log()`, which simply writes to *stdout* and appends a line feed (`\n`). Currently aliased as `console.info()`.

```
console.log('wahoo');
// => wahoo

console.log({ foo: 'bar' });
// => [object Object]
```

console.error()

Identical to `console.log()`, however writes to *stderr*. Aliased as `console.warn()` as well.

```
console.error('database connection failed');
```

console.dir()

Utilizes the *sys* module's `inspect()` method to pretty-print the object to *stdout*.

```
console.dir({ foo: 'bar' });
// => { foo: 'bar' }
```

console.assert()

Asserts that the given expression is truthy, or throws an exception.

```
console.assert(connected, 'Database connection failed');
```

process

The `process` object is plastered with goodies. First we will take a look at some properties that provide information about the node process itself:

process.version

The node version string, for example "v0.1.103".

process.installPrefix

The installation prefix. In my case "*/usr/local*", as node's binary was installed to "*/usr/local/bin/node*".

process.execPath

The path to the executable itself "*/usr/local/bin/node*".

process.platform

The platform you are running on. For example, "darwin".

process.pid

The process id.

process.cwd()

Returns the current working directory. For example:

```
cd ~ && node
node> process.cwd()
"/Users/tj"
```

process.chdir()

Changes the current working directory to the path passed.

```
process.chdir('/foo');
```

process.getuid()

Returns the numerical user id of the running process.

process.setuid()

Sets the effective user id for the running process. This method accepts both a numerical id, as well as a string. For example both `process.setuid(501)`, and `process.setuid('tj')` are valid.

process.getgid()

Returns the numerical group id of the running process.

process.setgid()

Similar to `process.setuid()` however operates on the group, also accepting a numerical value or string representation. For example, `process.setgid(20)` or `process.setgid('www')`.

process.env

An object containing the user's environment variables. For example:

```
{ PATH: '/Users/tj/.gem/ruby/1.8/bin:/Users/tj/.nvm/current/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X1
, PWD: '/Users/tj/ebooks/masteringnode'
, EDITOR: 'mate'
, LANG: 'en_CA.UTF-8'
, SHLVL: '1'
, HOME: '/Users/tj'
, LOGNAME: 'tj'
, DISPLAY: '/tmp/launch-YCkT03/org.x:0'
, _: '/usr/local/bin/node'
, OLDPWD: '/Users/tj'
}
```

process.argv

When executing a file with the `node` executable `process.argv` provides access to the argument vector, the first value being the node executable, second being the filename, and remaining values being the arguments passed.

For example, our source file *./src/process/misc.js* can be executed by running:

```
$ node src/process/misc.js foo bar baz
```

in which we call `console.dir(process.argv)`, outputting the following:

```
[ 'node'
, '/Users/tj/EBooks/masteringnode/src/process/misc.js'
, 'foo'
, 'bar'
, 'baz'
]
```

process.exit()

The `process.exit()` method is synonymous with the C function `exit()`, in which an exit code > 0 is passed to indicate failure, or 0 is passed to indicate success. When invoked, the *exit* event is emitted, allowing a short time for arbitrary processing to occur before `process.reallyExit()` is called with the given status code.

process.on()

The process itself is an `EventEmitter`, allowing you to do things like listen for uncaught exceptions via the *uncaughtException* event:

```
process.on('uncaughtException', function(err){
    console.log('got an error: %s', err.message);
    process.exit(1);
});

setTimeout(function(){
    throw new Error('fail');
}, 100);
```

process.kill()

`process.kill()` method sends the signal passed to the given *pid*, defaulting to **SIGINT**. In the example below, we send the **SIGTERM** signal to the same node process to illustrate signal trapping, after which we output "terminating" and exit. Note that the second timeout of 1000 milliseconds is never reached.

```
process.on('SIGTERM', function(){
    console.log('terminating');
    process.exit(1);
});

setTimeout(function(){
    console.log('sending SIGTERM to process %d', process.pid);
    process.kill(process.pid, 'SIGTERM');
}, 500);

setTimeout(function(){
    console.log('never called');
}, 1000);
```

errno

The `process` object is host of the error numbers, which reference what you would find in C-land. For example, `process.EPERM` represents a permission based error, while `process.ENOENT` represents a missing file or directory. Typically these are used within bindings to bridge the gap between C++ and JavaScript, but they're useful for handling exceptions as well:

```
if (err.errno === process.ENOENT) {
    // Display a 404 "Not Found" page
} else {
    // Display a 500 "Internal Server Error" page
}
```

Events

The concept of an "event" is crucial to node, and is used heavily throughout core and 3rd-party modules. Node's core module *events* supplies us with a single constructor, *EventEmitter*.

Emitting Events

Typically an object inherits from *EventEmitter*, however our small example below illustrates the API. First we create an `emitter`, after which we can define any number of callbacks using the `emitter.on()` method, which accepts the *name* of the event and arbitrary objects passed as data. When `emitter.emit()` is called, we are only required to pass the event *name*, followed by any number of arguments (in this case the `first` and `last` name strings).

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

var emitter = new EventEmitter;

emitter.on('name', function(first, last){
    console.log(first + ', ' + last);
});

emitter.emit('name', 'tj', 'holowaychuk');
emitter.emit('name', 'simon', 'holowaychuk');
```

Inheriting From EventEmitter

A more practical and common use of `EventEmitter` is to inherit from it. This means we can leave `EventEmitter`'s prototype untouched while utilizing its API for our own means of world domination!

To do so, we begin by defining the `Dog` constructor, which of course will bark from time to time (also known as an *event*).

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

function Dog(name) {
    this.name = name;
}
```

Here we inherit from `EventEmitter` so we can use the methods it provides, such as `EventEmitter#on()` and `EventEmitter#emit()`. If the `__proto__` property is throwing you off, don't worry, we'll be coming back to this later.

```
Dog.prototype.__proto__ = EventEmitter.prototype;
```

Now that we have our `Dog` set up, we can create... Simon! When Simon barks, we can let *stdout* know by calling `console.log()` within the callback. The callback itself is called in the context of the object (aka `this`).

```
var simon = new Dog('simon');

simon.on('bark', function(){
    console.log(this.name + ' barked');
});
```

Bark twice per second:

```
setInterval(function(){
    simon.emit('bark');
}, 500);
```

Removing Event Listeners

As we have seen, event listeners are simply functions which are called when we `emit()` an event. We can remove these listeners by calling the `removeListener(type, callback)` method, although this isn't seen often. In the example below we emit the *message* "foo bar" every `300` milliseconds, which has a callback of `console.log()`. After 1000 milliseconds, we call `removeListener()` with the same arguments that we passed to `on()` originally. We could also have used `removeAllListeners(type)`, which removes all listeners registered to the given *type*.

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

var emitter = new EventEmitter;

emitter.on('message', console.log);

setInterval(function(){
    emitter.emit('message', 'foo bar');
}, 300);

setTimeout(function(){
    emitter.removeListener('message', console.log);
}, 1000);
```

Buffers

To handle binary data, node provides us with the global `Buffer` object. `Buffer` instances represent memory allocated independently of V8's heap. There are several ways to construct a `Buffer` instance, and many ways you can manipulate its data.

The simplest way to construct a `Buffer` from a string is to simply pass a string as the first argument. As you can see in the log output, we now have a buffer object containing 5 bytes of data represented in hexadecimal.

```
var hello = new Buffer('Hello');

console.log(hello);
// => <Buffer 48 65 6c 6c 6f>

console.log(hello.toString());
// => "Hello"
```

By default, the encoding is "utf8", but this can be overridden by passing a string as the second argument. For example, the ellipsis below will be printed to stdout as the "&" character when in "ascii" encoding.

```
var buf = new Buffer('â–¦');
console.log(buf.toString());
// => â–¦

var buf = new Buffer('â–¦', 'ascii');
console.log(buf.toString());
// => &
```

An alternative (but in this case functionality equivalent) method is to pass an array of integers representing the octet stream.

```
var hello = new Buffer([0x48, 0x65, 0x6c, 0x6c, 0x6f]);
```

Buffers can also be created with an integer representing the number of bytes allocated, after which we can call the `write()` method, providing an optional offset and encoding. Below, we provide an offset of 2 bytes to our second call to `write()` (buffering "Hel") and then write another two bytes with an offset of 3 (completing "Hello").

```
var buf = new Buffer(5);
buf.write('He');
buf.write('l', 2);
buf.write('lo', 3);
console.log(buf.toString());
// => "Hello"
```

The `.length` property of a buffer instance contains the byte length of the stream, as opposed to native strings, which simply return the number of characters. For example, the ellipsis character 'â—¦' consists of three bytes, so the buffer will respond with the byte length (3), and not the character length (1).

```
var ellipsis = new Buffer('â–¦', 'utf8');

console.log('â–¦ string length: %d', 'â–¦'.length);
// => â–¦ string length: 1

console.log('â–¦ byte length: %d', ellipsis.length);
// => â–¦ byte length: 3

console.log(ellipsis);
```

```
// => <Buffer e2 80 a6>
```

To determine the byte length of a native string, pass it to the `Buffer.byteLength()` method.

The API is written in such a way that it is String-like. For example, we can work with "slices" of a `Buffer` by passing offsets to the `slice()` method:

```
var chunk = buf.slice(4, 9);
console.log(chunk.toString());
// => "some"
```

Alternatively, when expecting a string, we can pass offsets to `Buffer#toString()`:

```
var buf = new Buffer('just some data');
console.log(buf.toString('ascii', 4, 9));
// => "some"
```

Streams

Streams are an important concept in node. The stream API is a unified way to handle stream-like data. For example, data can be streamed to a file, streamed to a socket to respond to an HTTP request, or streamed from a read-only source such as *stdin*. For now, we'll concentrate on the API, leaving stream specifics to later chapters.

Readable Streams

Readable streams such as an HTTP request inherit from `EventEmitter` in order to expose incoming data through events. The first of these events is the *data* event, which is an arbitrary chunk of data passed to the event handler as a `Buffer` instance.

```
req.on('data', function(buf){
    // Do something with the Buffer
});
```

As we know, we can call `toString()` on a buffer to return a string representation of the binary data. Likewise, we can call `setEncoding()` on a stream, after which the *data* event will emit strings.

```
req.setEncoding('utf8');
req.on('data', function(str){
    // Do something with the String
});
```

Another important event is *end*, which represents the ending of *data* events. For example, here's an HTTP echo server, which simply "pumps" the request body data through to the response. So if we POST "hello world", our response will be "hello world".

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

http.createServer(function(req, res){
    res.writeHead(200);
    req.on('data', function(data){
        res.write(data);
    });
    req.on('end', function(){
        res.end();
    });
}).listen(3000);
```

The *sys* module actually has a function designed specifically for this "pumping" action, aptly named `sys.pump()`. It accepts a read stream as the first argument, and write stream as the second.

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

http.createServer(function(req, res){
    res.writeHead(200);
    sys.pump(req, res);
}).listen(3000);
```

File System

To work with the filesystem, node provides the "fs" module. The commands emulate the POSIX operations, and most methods work synchronously or asynchronously. We will look at how to use both, then establish which is the better option.

Working with the filesystem

Lets start with a basic example of working with the filesystem. This example creates a directory, creates a file inside it, then writes the contents of the file to console:

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

fs.mkdir('./helloDir',0777, function (err) {
  if (err) throw err;

  fs.writeFile('./helloDir/message.txt', 'Hello Node', function (err) {
    if (err) throw err;
    console.log('file created with contents:');

    fs.readFile('./helloDir/message.txt','UTF-8' ,function (err, data) {
      if (err) throw err;
      console.log(data);
    });
  });
});
```

As evident in the example above, each callback is placed in the previous callback — these are referred to as chainable callbacks. This pattern should be followed when using asynchronous methods, as there's no guarantee that the operations will be completed in the order they're created. This could lead to unpredictable behavior.

The example can be rewritten to use a synchronous approach:

```
fs.mkdirSync('./helloDirSync',0777);
fs.writeFileSync('./helloDirSync/message.txt', 'Hello Node');
var data = fs.readFileSync('./helloDirSync/message.txt','UTF-8');
console.log('file created with contents:');
console.log(data);
```

It is better to use the asynchronous approach on servers with a high load, as the synchronous methods will cause the whole process to halt and wait for the operation to complete. This will block any incoming connections or other events.

File information

The fs.Stats object contains information about a particular file or directory. This can be used to determine what type of object we're working with. In this example, we're getting all the file objects in a directory and displaying whether they're a file or a directory object.

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

fs.readdir('/etc/', function (err, files) {
  if (err) throw err;

  files.forEach( function (file) {
    fs.stat('/etc/' + file, function (err, stats) {
      if (err) throw err;

      if (stats.isFile()) {
        console.log("%s is file", file);
      }
      else if (stats.isDirectory ()) {
      console.log("%s is a directory", file);
      }
    console.log('stats:  %s',JSON.stringify(stats));
    });
  });
});
```

Watching files

The fs.watchfile method monitors a file and fires an event whenever the file is changed.

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

fs.watchFile('./testFile.txt', function (curr, prev) {
  console.log('the current mtime is: ' + curr.mtime);
  console.log('the previous mtime was: ' + prev.mtime);
});

fs.writeFile('./testFile.txt', "changed", function (err) {
  if (err) throw err;

  console.log("file write complete");
});
```

A file can also be unwatched using the fs.unwatchFile method call. This should be used once a file no longer needs to be monitored.

Nodejs Docs for further reading

The node API docs are very detailed and list all the possible filesystem commands available when working with Nodejs.

TCP

...

TCP Servers

...

TCP Clients

...

HTTP

...

HTTP Servers

...

HTTP Clients

...

Connect

Connect is a ...

Express

Express is a ...

Testing

...

Expresso

...

Vows

...

Deployment

...