The NodeJS Cluster Module

Antonio Santiago

The NodeJS Cluster Module

Antonio Santiago

This book is for sale at http://leanpub.com/thenodejsclustermodule

This version was published on 2017-11-19

Contents

Understanding the NodeJS cluster module **1**
Introducing the cluster module 1
A basic example 2
Comunicating master and worker processes 4
Conclusion 6

Using cluster module with HTTP servers **7**
How cluster module works with network connections ? 8
Other alternatives to cluster module load balancing 9
Conclusions 9

Using PM2 to manage NodeJS cluster **10**
Introducing PM2 10
Conclusions 13
References 13

Graceful shutdown NodeJS HTTP server when using PM2 . . . . **14**
Starting a HTTP server 14
How to shutdown properly an HTTP server ? 15
Graceful shutdown/restart with and without PM2 15
Conclusions 16

Understanding the NodeJS cluster module

NodeJS processes runs on a single process, which means it does not take advantage from multi-core systems by default. If you have an 8 core CPU and run a NodeJS program via `$ node app.js` it will run in a single process, wasting the rest of CPUs.

Hopefully for us NodeJS offers the cluster[1] module that contains a set of functions and properties that help us to create programs that uses all the CPUs. Not a surprise the mechanism the cluster module uses to maximize the CPU usage was via forking processes, similar to the old fork()[2] system call Unix systems.

Introducing the cluster module

The cluster module is a NodeJS module that contains a set of functions and properties that help us forking processes to take advantage of multi-core systems. It is propably the first level of scalability you must take care in your node application, specifally if you are working in a HTTP server application, before going to a higher scalability levels (I mean scaling vertically and horizontally in different machines).

With the cluster module a *parent/master* process can be forked in any number of *child/worker* processes and communicate with them sending messages via IPC[3] communication. **Remember there is no shared memory among processes**.

Next lines are a compilation of sentences from the NodeJS documentation I have taken the liberty to copy&pasta to put it in a way I think can help you understand thw whole thing in a few lines.

> A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node.js processes to handle the load.
>
> The cluster module allows easy creation of child processes that all share server ports.
>
> The worker (child) processes are spawned using the `child_proces.fork()` method, so that they can communicate with the parent via IPC and pass server handles back and forth. The `child_process.fork()` method is a special case of `child_process.spawn()` used specifically to spawn new Node.js processes. Like `child_process.spawn()`, a

[1] https://nodejs.org/api/cluster.html

[2] http://www.includehelp.com/c-programs/c-fork-function-linux-example.aspx

[3] https://en.wikipedia.org/wiki/Inter-process_communication

> `ChildProcess` object is returned. The returned `ChildProcess` will have an additional communication channel built-in that allows messages to be passed back and forth between the parent and child, through the `send()` method. See `subprocess.sen()` for details.
>
> It is important to keep in mind that spawned Node.js child processes are independent of the parent with exception of the IPC communication channel that is established between the two. Each process has its own memory, with their own V8 instances. Because of the additional resource allocations required, spawning a large number of child Node.js processes is not recommended.

So, most of the magic is done by the child_process[4] module, which is resposible to spawn new process and help communicate among them, for example, creating pipes. You can find a great article at Node.js Child Processes: Everything you need to know[5].

A basic example

Stop talking and lets see a real exampe. Next we show a very basic code that:

- Creates a master process that retrives the number of CPUs and forks a worker process for each CPU, and
- Each child process prints a message in console and exit.

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  masterProcess();
} else {
  childProcess();
}

function masterProcess() {
  console.log(`Master ${process.pid} is running`);

  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);
    cluster.fork();
  }
```

[4] https://nodejs.org/api/child_process.html
[5] https://medium.freecodecamp.org/node-js-child-processes-everything-you-need-to-know-e69498fe970a

```

  process.exit();
}

function childProcess() {
  console.log(`Worker ${process.pid} started and finished`);

  process.exit();
}
```

Save the code in `app.js` file and run executing: `$ node app.js`. The output should be something similar to:

```
$ node app.js

Master 8463 is running
Forking process number 0...
Forking process number 1...
Forking process number 2...
Forking process number 3...
Worker 8464 started and finished
Worker 8465 started and finished
Worker 8467 started and finished
Worker 8466 started and finished
```

Code explanation

When we run the `app.js` program an OS process is created that starts running our code. At the beginning the cluster mode is imported `const cluster = require('cluster')` and in the `if` sentence we check if the `isMaster` property.

Because the process is the *first* process the `isMaster` property is `true` and then we run the code of `masterProcess` function. This function has not much secret, it loops depending on the number of CPUs of your machine and forks the current process using the `cluster.fork()` method.

What the `fork()` really does is to create a new node process, like if you run it via command line with `$node app.js`, that is you have many processes running your `app.js` program.

When a child process is created and executed, it does the same as the master, that is, imports the cluster module and executes the `if` statement. Once of the differences is for the child process the value of `cluster.isMaster` is `false`, so they ends running the `childProcess` function.

Note, we explicitly terminate the master and worker processes with `process.exit()`, which by default return value of zero.

> NOTE: NodeJS also offers the Child Processes[6] module that simplifies the creation and comunication with other processes. For example we can spawn the `ls -l` terminal command and pipe with another process that handles the results.

Comunicating master and worker processes

When a worker process is created, an IPC channel is created among the worker and the master, allowing us to communicated between them with the `send()` method, which accepts a JavaScript object as parameter. Remember they are different processes (not threads) so we can't use shared memory as a way of communcation.

From the master process, we can send a message to a worker process using the process reference, i.e. `someChild.send({ ... })`, and within the worker process we can messages to the master simply using the current `process` reference, i.e. `process.send()`.

We have updated slighly the previous code to allow master send and receive messages from/to the workers and also the workers receive and send messages from the master process:

```
function childProcess() {
  console.log(`Worker ${process.pid} started`);

  process.on('message', function(message) {
    console.log(`Worker ${process.pid} recevies message '${JSON.stringify(messag\
e)}'`);
  });

  console.log(`Worker ${process.pid} sends message to master...`);
  process.send({ msg: `Message from worker ${process.pid}` });

  console.log(`Worker ${process.pid} finished`);
}
```

The worker process is simply to understand. First we listen for the `message` event registering a listener with the `process.on('message', handler)` method. Later we send a messages with `process.send({...})`. Note the message is a plain JavaScript object.

[6] https://nodejs.org/api/child_process.html

```
let workers = [];

function masterProcess() {
  console.log(`Master ${process.pid} is running`);

  // Fork workers
  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);

    const worker = cluster.fork();
    workers.push(worker);

    // Listen for messages from worker
    worker.on('message', function(message) {
      console.log(`Master ${process.pid} recevies message '${JSON.stringify(mess\
age)}' from worker ${worker.process.pid}`);
    });
  }

  // Send message to the workers
  workers.forEach(function(worker) {
    console.log(`Master ${process.pid} sends message to worker ${worker.process.\
pid}...`);
    worker.send({ msg: `Message from master ${process.pid}` });
  }, this);
}
```

The `masterProcess` function has been divided in two parts. In the first loop we fork as much workers as CPUs we have. The `cluster.fork()` returns a `worker` object representing the worker process, we store the reference in an array and register a listener to receive messages that comes from that worker instance.

Later, we loop over the array of workers and send a message from the master process to that concrete worker.

If you run the code the output will be something like:

```
$ node app.js

Master 4045 is running
Forking process number 0...
Forking process number 1...
Master 4045 sends message to worker 4046...
Master 4045 sends message to worker 4047...
Worker 4047 started
Worker 4047 sends message to master...
Worker 4047 finished
Master 4045 recevies message '{"msg":"Message from worker 4047"}' from worker 40\
47
Worker 4047 recevies message '{"msg":"Message from master 4045"}'
Worker 4046 started
Worker 4046 sends message to master...
Worker 4046 finished
Master 4045 recevies message '{"msg":"Message from worker 4046"}' from worker 40\
46
Worker 4046 recevies message '{"msg":"Message from master 4045"}'
```

Here we are not terminating the process with `process.exit()` so to close the application you need to use `ctrl+c`.

Conclusion

The cluster module[7] offers to NodeJS the needed capabilities to use the whole power of a CPU. Although not seen in this post, the cluster module is complemented with the child process[8] module that offers plenty of tools to work with processes: start, stop and pipe input/out, etc.

Cluster module allow us to easily create worker processes. In addition it **magically** creates an IPC channel to communicate the master and worker process passing JavaScript objects.

In my next post I will show how important is the cluster module when working in an HTTP server, no matter if an API or web server working with ExpressJS. The cluster module can increase performance of our application having as many worker processes as CPUs cores.

[7] https://nodejs.org/api/cluster.html
[8] https://nodejs.org/api/child_process.html

Using cluster module with HTTP servers

The cluster[9] module allow us improve performance of our application in multicore CPU systems. This is specially important no matter if working on an APIs or an, i.e. ExpressJS based, web servers, what we desire is to take advantage of all the CPUs on each machine our NodeJS application is running.

The cluster module allow us to load balance the incoming request among a set of worker processes and, because of this, improving the throughput of our application.

In the previous section I have introduced the cluster module and shown some basic usages of it to create worker processes and comunicate them with the master process. In this post we are going to see how to use the cluster module when creating HTTP servers, both using plain HTTP[10] module and with ExpressJS.

Lets go to see how we can create a really basic HTTP server that takes profit of the cluster module.

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  masterProcess();
} else {
  childProcess();
}

function masterProcess() {
  console.log(`Master ${process.pid} is running`);

  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);
    cluster.fork();
  }
}

```

[9]https://nodejs.org/api/cluster.html
[10]https://nodejs.org/api/http.html

```
function childProcess() {
  console.log(`Worker ${process.pid} started...`);

  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello World');
  }).listen(3000);
}
```

We have diveded the code in two parts, the one corresponding to the master process and the one where we initialize the worker processes. This way the `masterProcess` function forks a worker process per CPU code. On the other hand the `childProcess` simply creates an HTTP server listenen on port 3000 and returning a nice `Hello World` text string with a 200 status code.

If you run the code the output must show something like:

```
$ node app.js

Master 1859 is running
Forking process number 0...
Forking process number 1...
Forking process number 2...
Forking process number 3...
Worker 1860 started...
Worker 1862 started...
Worker 1863 started...
Worker 1861 started...
```

Basically our initial process (the master) is spawing a new worker process per CPU that runs an HTTP server that handle requests. As you can see this can improve a lot your server performance because it is not the same having one processing attending one million of requests than having four processes attending one millun requests.

How cluster module works with network connections ?

The previous example is simple but hides something tricky, some *magic* NodeJS make to simplify our live as developer.

In any OS a process can use a port to communicate with other systems and, that means, the given port can only be used by that process. So, the question is, **how can the forked worker processes use the same port?**

The answer, the simplified answer, is the master process is the one who listens in the given port and load balances the requests among all the child/worker processes. From the offical documentation:

> The worker processes are spawned using the `child_process.fork()` method, so that they can communicate with the parent via IPC and pass server handles back and forth.
>
> The cluster module supports two methods of distributing incoming connections.
>
> - The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.
>
> - The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.
>
> **As long as there are some workers still alive, the server will continue to accept connections. If no workers are alive, existing connections will be dropped and new connections will be refused.**

Other alternatives to cluster module load balancing

Cluster module allow the master process to receive request and load balance it among all the worker processes. This is a way to improve performance but it is not the only one.

In the post Node.js process load balance performance: comparing cluster module, iptables and Nginx[11] you can find a performance comparison among: node cluster module, iptables and nginx reverse proxy.

Conclusions

Nowadays performance is mandatory on any web applications, we need to support high throughput and serve data fast.

The cluster module is one possible solution, it allows us to have one master process and create a worker processes for each core, so that they run an HTTP server. The cluster module offers two great features:

- simplifies communication among master and workers, by creating an IPC channel and allowing send messages with `process.send()`,

- allow worker processes share the same port. This is done making the master process the one which receives requests and multiplexe them among workers.

[11]https://medium.com/@fermads/node-js-process-load-balancing-comparing-cluster-iptables-and-nginx-6746aaf38272

Using PM2 to manage NodeJS cluster

The cluster module allows us to create worker processes to improve our NodeJS applications performance. This is specially important in web applications, where a master process receives all the requests and load balances them among the worker processes.

But all this power comes with the cost that must be the application who manages all the complexity associated with process managements: what happens if a worker process exists unexpectedly, how exit gracefully the worker processes, what if you need to restart all your workers, etc.

In this post we present PM2[12] tool. although it is a general process manager, that means it can manage any kind of process like python, ruby, … and not only NodeJS processes, the tool is specially designed to manage NodeJS applications that want to work with the cluster module.

Introducing PM2

As said previously, PM2 is a general process manager, that is, a program that controls the execution of other process (like a python program that check if you have new emails) and does things like: check your process is running, re-execute your process if for some reason it exits unexpectedly, log its output, etc.

The most important thing for us is PM2 simplifies the execution of NodeJS applications to run as a cluster. Yes, you write your application without worrying about cluster module and is PM2 who creates a given number of worker processes to run your application.

The hard part of cluster module

Lets see an example where we create a very basic HTTP server using the cluster module. The master process will spawn as many workers as CPUs and will take care if any of the workers exists to spawn a new worker.

[12] http://pm2.keymetrics.io

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  masterProcess();
} else {
  childProcess();
}

function masterProcess() {
  console.log(`Master ${process.pid} is running`);

  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);

    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died`);
    console.log(`Forking a new process...`);

    cluster.fork();
  });
}

function childProcess() {
  console.log(`Worker ${process.pid} started...`);

  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello World');

    process.exit(1);
  }).listen(3000);
}
```

The worker process is a very simple HTTP server listening on port 3000 and programmed to return a `Hello World` and exit (to simulate a failure).

If we run the program with `$ node app.js` the output will show something like:

```
$ node app.js

Master 2398 is running
Forking process number 0...
Forking process number 1...
Worker 2399 started...
Worker 2400 started...
```

If we go to browser at URL `http://localhost:3000` we will get a `Hello World` and in the console see something like:

```
Worker 2400 died
Forking a new process...
Worker 2401 started...
```

That's very nice, now lets go to see how PM2 can simplify our application.

The PM2 way

Before continue, you need to instal PM2 on your system. Typically it is installed as a global module with `$ npm install pm2 -g` or `$ yarn global add pm2`.

When using PM2 we can forget the part of the code related with the master process, that will responsibility of PM2, so our very basic HTTP server can be rewritten as:

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

console.log(`Worker ${process.pid} started...`);

http.createServer((req, res) => {
  res.writeHead(200);
  res.end('Hello World');

  process.exit(1);
}).listen(3000);
```

Now run PM2 with `$ pm2 start app.js -i 3` and you will see an output similar to:

> Note the option `-i` that is used to indicate the number of instances to create. The idea is that number be the same as your number of CPU cores. If you don't know them you can set `-i 0` to leave PM2 detect it automatically.

```
$ pm2 start app.js -i 3

[PM2] Starting /Users/blablabla/some-project/app.js in cluster_mode (3 instances)
[PM2] Done.

| Name       | mode    | status | □ | cpu | memory     |
| ----------|---------|--------|---|-----|-----------|
| app       | cluster | online | 0 | 23% | 27.1 MB   |
| app       | cluster | online | 0 | 26% | 27.3 MB   |
| app       | cluster | online | 0 | 14% | 25.1 MB   |
```

We can see the application logs running `$ pm2 log`. Now when accessing the the `http://localhost:3000` URL we will see logs similar to:

```
PM2         | App name:app id:0 disconnected
PM2         | App [app] with id [0] and pid [1299], exited with code [1] via sign\
al [SIGINT]
PM2         | Starting execution sequence in -cluster mode- for app name:app id:0
PM2         | App name:app id:0 online
0|app       | Worker 1489 started...
```

We can see how PM2 process detects one of our workers has exit and automatically starts a new instance.

Conclusions

Although the NodeJS cluster module is a powerful mechanism to improve performance it comes at the cost of complexity required to manage all the situations an application can found: what happens if a worker exists, how can we reload the application cluster without down time, etc.

PM2 is a process manager specially designed to work with NodeJS clusters. It allow to cluster an application, restart or reload, without the required code complexity in addition to offer tools to see log outputs, monitorization, etc.

References

Node.js clustering made easy with PM2[13]

[13] https://keymetrics.io/2015/03/26/pm2-clustering-made-easy/

Graceful shutdown NodeJS HTTP server when using PM2

So you have created a NodeJS server that receives tons of requests and you are really happy but, as every piece of software, you found a bug or add a new feature to it. It is clear you will need to shutdown your NodeJS process/es and restart again so that the new code takes place. The question is: **how can you do that in a graceful way that allows continue serving incoming requests?**

Starting a HTTP server

Before see how we must shutdown a HTTP server lets see how usually create one. Next code shows a very basic code with an ExpressJS service that will return `Hello World !!!` when accessing the `/hello` path. You can also pass a path param, i.e. `/hello/John` with a name so it returns `Hello John !!!`.

```
const express = require('express')

const expressApp = express()

// Responds with Hello World or optionally the name you pass as path param
expressApp.get('/hello/:name?', function (req, res) {
  const name = req.params.name

  if (name) {
    return res.send(`Hello ${name}!!!`)
  }

  return res.send('Hello World !!!')
})

// Start server
expressApp.listen(3000, function () {
  console.log('App listening on port 3000!')
})
```

What `app.listen()` function does is start a new HTTP server using the core `http` module and return a reference to the HTTP server object. In concrete, the source code of the `listen()` is as follows:

```
app.listen = function listen() {
  var server = http.createServer(this);
  return server.listen.apply(server, arguments);
};
```

> NOTE: Another way to create an express server is pass our `expressApp` reference directly to the `http. createServer()`, something like: `const server = http.createServer(app).listen(300`

How to shutdown properly an HTTP server ?

The proper way to shutdown an HTTP server is to invoke the `server.close()` function, this will stop server from accepting new connections while keeps existing ones until response them.

Next code presents a new `/close` endpoint that once invoked will stop the HTTP server and exit the applications (stopping the nodejs process):

```
app.get('/close', (req, res) => {
  console.log('Closing the server...')

  server.close(() => {
    console.log('--> Server call callback run !!')

    process.exit()
  })
})
```

It is clear shutting down a server through an endpoint is not the right way to it.

Graceful shutdown/restart with and without PM2

The goal of a graceful shutdown is to close the incoming connections to a server without killing the current ones we are handling.

When using a process manager like PM2, we manage a cluster of processes each one acting as a HTTP server. The way PM2 achieves the graceful restart is:

- sending a `SIGNINT` signal to each worker process,
- the worker are responsible to catch the signal, cleanup or free any used resource and finish the its process,
- finally PM2 manager spawns a new process

Because this is done sequentially with our cluster processes customers must not be affected by the restart because there will always be some processes working and attending requests.

This is very useful when we deploy new code and want to restart our servers so the new changes take effect without risk for incoming requests. We can achieve this putting next code in out app:

```
// Graceful shutdown
process.on('SIGINT', () => {
  const cleanUp = () => {
    // Clean up other resources like DB connections
  }

  console.log('Closing server...')

  server.close(() => {
    console.log('Server closed !!! ')

    cleanUp()
    process.exit()
  })

  // Force close server after 5secs
  setTimeout((e) => {
    console.log('Forcing server close !!!', e)

    cleanUp()
    process.exit(1)
  }, 5000)
})
```

When the `SINGINT` signal it catch we invoke the `server.close()` to avoid accepting more requests and once it is closed we clean up any resource used by our app, like close database connection, close opened files, etc invoking the `cleanUp()` function and, finally, we exits the process with `process.exit()`. In addition, if for some reason our code spends too much time to close the server we force it running a very similar code within a `setTimeout()`.

Conclusions

When creating a HTTP server, no matter if a web server to serve pages or an API, we need to take into account the fact it will be updated in time with new features and bug fixes, so we need to think in a way to minimize the impact on customers.

Running nodejs processes in cluster mode is a common way to improve our applications performance and we need to think on how to graceful shutdown all them to not affect incoming requests.

Terminating a node process with `process.exit()` is not enough when working with an HTTP server because it will terminate abruptly all the communications, we need to first stop accepting new connections, free any resource used by our application and, finally, stop the process.