Node.js v5.6.0 Documentation

Index | View on single page | View as JSON

Table of Contents

Cluster

Stability: 2 - Stable

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 you to easily create child processes that all share server ports.

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

```
if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);
}
```

Running Node.js will now share port 8000 between the workers:

```
$ NODE_DEBUG=cluster node server.js
23521,Master Worker 23524 online
23521,Master Worker 23526 online
23521,Master Worker 23523 online
23521,Master Worker 23528 online
```

Please note that, on Windows, it is not yet possible to set up a named pipe server in a worker.

How It Works

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.

The second approach should, in theory, give the best performance. In practice however, distribution tends to be very unbalanced due to operating system scheduler vagaries. Loads have been observed where over 70% of all connections ended up in just two processes, out of a total of eight.

Because `server.listen()` hands off most of the work to the master process, there are three cases where the behavior between a normal Node.js process and a cluster worker differs:

1. `server.listen({fd: 7})` Because the message is passed to the master, file descriptor 7 **in the parent** will be listened on, and the handle passed to the worker, rather than listening to the worker's idea of what the number 7 file descriptor references.
2. `server.listen(handle)` Listening on handles explicitly will cause the worker to

use the supplied handle, rather than talk to the master process. If the worker already has the handle, then it's presumed that you know what you are doing.

3. `server.listen(0)` Normally, this will cause servers to listen on a random port. However, in a cluster, each worker will receive the same "random" port each time they do `listen(0)`. In essence, the port is random the first time, but predictable thereafter. If you want to listen on a unique port, generate a port number based on the cluster worker ID.

There is no routing logic in Node.js, or in your program, and no shared state between the workers. Therefore, it is important to design your program such that it does not rely too heavily on in-memory data objects for things like sessions and login.

Because workers are all separate processes, they can be killed or re-spawned depending on your program's needs, without affecting other workers. 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. Node.js does not automatically manage the number of workers for you, however. It is your responsibility to manage the worker pool for your application's needs.

Class: Worker

A Worker object contains all public information and method about a worker. In the master it can be obtained using `cluster.workers`. In a worker it can be obtained using `cluster.worker`.

Event: 'disconnect'

Similar to the `cluster.on('disconnect')` event, but specific to this worker.

```
cluster.fork().on('disconnect', () => {
  // Worker has disconnected
});
```

Event: 'error'

This event is the same as the one provided by `child_process.fork()`.

In a worker you can also use `process.on('error')`.

Event: 'exit'

- `code` Number the exit code, if it exited normally.
- `signal` String the name of the signal (eg. `'SIGHUP'`) that caused the process to be killed.

Similar to the `cluster.on('exit')` event, but specific to this worker.

```
const worker = cluster.fork();
worker.on('exit', (code, signal) => {
  if( signal ) {
    console.log(`worker was killed by signal: ${signal}`);
  } else if( code !== 0 ) {
    console.log(`worker exited with error code: ${code}`);
  } else {
    console.log('worker success!');
  }
});
```

Event: 'listening'

- `address` Object

Similar to the `cluster.on('listening')` event, but specific to this worker.

```
cluster.fork().on('listening', (address) => {
  // Worker is listening
});
```

It is not emitted in the worker.

Event: 'message' #

- `message` Object

Similar to the `cluster.on('message')` event, but specific to this worker.

This event is the same as the one provided by `child_process.fork()`.

In a worker you can also use `process.on('message')`.

As an example, here is a cluster that keeps count of the number of requests in the master process using the message system:

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

if (cluster.isMaster) {

  // Keep track of http requests
  var numReqs = 0;
  setInterval(() => {
    console.log('numReqs =', numReqs);
  }, 1000);

  // Count requests
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd == 'notifyRequest') {
      numReqs += 1;
    }
  }

  // Start workers and listen for messages containing notifyRequest
  const numCPUs = require('os').cpus().length;
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  Object.keys(cluster.workers).forEach((id) => {
    cluster.workers[id].on('message', messageHandler);
  });

} else {

  // Worker processes have a http server.
  http.Server((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
```

```
    // notify master about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```

Event: 'online'

Similar to the `cluster.on('online')` event, but specific to this worker.

```
cluster.fork().on('online', () => {
  // Worker is online
});
```

It is not emitted in the worker.

worker.disconnect()

In a worker, this function will close all servers, wait for the `'close'` event on those servers, and then disconnect the IPC channel.

In the master, an internal message is sent to the worker causing it to call `.disconnect()` on itself.

Causes `.suicide` to be set.

Note that after a server is closed, it will no longer accept new connections, but connections may be accepted by any other listening worker. Existing connections will be allowed to close as usual. When no more connections exist, see [server.close()][], the IPC channel to the worker will close allowing it to die gracefully.

The above applies *only* to server connections, client connections are not automatically closed by workers, and disconnect does not wait for them to close before exiting.

Note that in a worker, `process.disconnect` exists, but it is not this function, it is `disconnect`.

Because long living server connections may block workers from disconnecting, it may be useful to send a message, so application specific actions may be taken to close them. It also may be useful to implement a timeout, killing a worker if the `'disconnect'` event has not been emitted after some time.

```
if (cluster.isMaster) {
  var worker = cluster.fork();
  var timeout;

  worker.on('listening', (address) => {
    worker.send('shutdown');
    worker.disconnect();
    timeout = setTimeout(() => {
      worker.kill();
    }, 2000);
  });

  worker.on('disconnect', () => {
    clearTimeout(timeout);
  });
```

```
} else if (cluster.isWorker) {
  const net = require('net');
  var server = net.createServer((socket) => {
    // connections never end
  });

  server.listen(8000);

  process.on('message', (msg) => {
    if(msg === 'shutdown') {
      // initiate graceful close of any connections to server
    }
  });
}
```

worker.id

- Number

Each new worker is given its own unique id, this id is stored in the `id`.

While a worker is alive, this is the key that indexes it in cluster.workers

worker.isConnected()

This function returns `true` if the worker is connected to its master via its IPC channel, `false` otherwise. A worker is connected to its master after it's been created. It is disconnected after the `'disconnect'` event is emitted.

worker.isDead()

This function returns `true` if the worker's process has terminated (either because of exiting or being signaled). Otherwise, it returns `false`.

worker.kill([signal='SIGTERM'])

- `signal` String Name of the kill signal to send to the worker process.

This function will kill the worker. In the master, it does this by disconnecting the `worker.process`, and once disconnected, killing with `signal`. In the worker, it does it by disconnecting the channel, and then exiting with code `0`.

Causes `.suicide` to be set.

This method is aliased as `worker.destroy()` for backwards compatibility.

Note that in a worker, `process.kill()` exists, but it is not this function, it is `kill`.

worker.process

- ChildProcess object

All workers are created using `child_process.fork()`, the returned object from this function is stored as `.process`. In a worker, the global `process` is stored.

See: Child Process module

Note that workers will call `process.exit(0)` if the `'disconnect'` event occurs on `process` and `.suicide` is not `true`. This protects against accidental disconnection.

worker.send(message[, sendHandle][, callback])

- `message` Object
- `sendHandle` Handle object

- `callback` Function
- Return: Boolean

Send a message to a worker or master, optionally with a handle.

In the master this sends a message to a specific worker. It is identical to `ChildProcess.send()`.

In a worker this sends a message to the master. It is identical to `process.send()`.

This example will echo back all messages from the master:

```
if (cluster.isMaster) {
  var worker = cluster.fork();
  worker.send('hi there');

} else if (cluster.isWorker) {
  process.on('message', (msg) => {
    process.send(msg);
  });
}
```

worker.suicide

- Boolean

Set by calling `.kill()` or `.disconnect()`, until then it is `undefined`.

The boolean `worker.suicide` lets you distinguish between voluntary and accidental exit, the master may choose not to respawn a worker based on this value.

```
cluster.on('exit', (worker, code, signal) => {
  if (worker.suicide === true) {
    console.log('Oh, it was just suicide\' – no need to worry').
  }
});

// kill worker
worker.kill();
```

Event: 'disconnect'

- `worker` Worker object

Emitted after the worker IPC channel has disconnected. This can occur when a worker exits gracefully, is killed, or is disconnected manually (such as with worker.disconnect()).

There may be a delay between the `'disconnect'` and `'exit'` events. These events can be used to detect if the process is stuck in a cleanup or if there are long-living connections.

```
cluster.on('disconnect', (worker) => {
  console.log(`The worker #${worker.id} has disconnected`);
});
```

Event: 'exit'

- `worker` Worker object

- `code` Number the exit code, if it exited normally.
- `signal` String the name of the signal (eg. `'SIGHUP'`) that caused the process to be killed.

When any of the workers die the cluster module will emit the `'exit'` event.

This can be used to restart the worker by calling `.fork()` again.

```
cluster.on('exit', (worker, code, signal) => {
  console.log('worker %d died (%s). restarting...',
    worker.process.pid, signal || code);
  cluster.fork();
});
```

See child_process event: 'exit'.

Event: 'fork'

- `worker` Worker object

When a new worker is forked the cluster module will emit a `'fork'` event. This can be used to log worker activity, and create your own timeout.

```
var timeouts = [];
function errorMsg() {
  console.error('Something must be wrong with the connection ...');
}

cluster.on('fork', (worker) => {
  timeouts[worker.id] = setTimeout(errorMsg, 2000);
});
cluster.on('listening', (worker, address) => {
  clearTimeout(timeouts[worker.id]);
});
cluster.on('exit', (worker, code, signal) => {
  clearTimeout(timeouts[worker.id]);
  errorMsg();
});
```

Event: 'listening'

- `worker` Worker object
- `address` Object

After calling `listen()` from a worker, when the `'listening'` event is emitted on the server, a `'listening'` event will also be emitted on `cluster` in the master.

The event handler is executed with two arguments, the `worker` contains the worker object and the `address` object contains the following connection properties: `address`, `port` and `addressType`. This is very useful if the worker is listening on more than one address.

```
cluster.on('listening', (worker, address) => {
  console.log(
    `A worker is now connected to ${address.address}:${address.port}`);
});
```

The `addressType` is one of:

- `4` (TCPv4)
- `6` (TCPv6)
- `-1` (unix domain socket)
- `"udp4"` or `"udp6"` (UDP v4 or v6)

Event: 'message'

- `worker` Worker object
- `message` Object

Emitted when any worker receives a message.

See child_process event: 'message'.

Event: 'online'

- `worker` Worker object

After forking a new worker, the worker should respond with an online message. When the master receives an online message it will emit this event. The difference between `'fork'` and `'online'` is that fork is emitted when the master forks a worker, and 'online' is emitted when the worker is running.

```
cluster.on('online', (worker) => {
  console.log('Yay, the worker responded after it was forked');
});
```

Event: 'setup'

- `settings` Object

Emitted every time `.setupMaster()` is called.

The `settings` object is the `cluster.settings` object at the time `.setupMaster()` was called and is advisory only, since multiple calls to `.setupMaster()` can be made in a single tick.

If accuracy is important, use `cluster.settings`.

cluster.disconnect([callback])

- `callback` Function called when all workers are disconnected and handles are closed

Calls `.disconnect()` on each worker in `cluster.workers`.

When they are disconnected all internal handles will be closed, allowing the master process to die gracefully if no other event is waiting.

The method takes an optional callback argument which will be called when finished.

This can only be called from the master process.

cluster.fork([env])

- `env` Object Key/value pairs to add to worker process environment.
- return Worker object

Spawn a new worker process.

This can only be called from the master process.

cluster.isMaster

- Boolean

True if the process is a master. This is determined by the `process.env.NODE_UNIQUE_ID`. If `process.env.NODE_UNIQUE_ID` is undefined, then `isMaster` is `true`.

cluster.isWorker

- Boolean

True if the process is not a master (it is the negation of `cluster.isMaster`).

cluster.schedulingPolicy

The scheduling policy, either `cluster.SCHED_RR` for round-robin or `cluster.SCHED_NONE` to leave it to the operating system. This is a global setting and effectively frozen once you spawn the first worker or call `cluster.setupMaster()`, whatever comes first.

`SCHED_RR` is the default on all operating systems except Windows. Windows will change to `SCHED_RR` once libuv is able to effectively distribute IOCP handles without incurring a large performance hit.

`cluster.schedulingPolicy` can also be set through the `NODE_CLUSTER_SCHED_POLICY` environment variable. Valid values are `"rr"` and `"none"`.

cluster.settings

- Object
  - `execArgv` Array list of string arguments passed to the Node.js executable. (Default=`process.execArgv`)
  - `exec` String file path to worker file. (Default=`process.argv[1]`)
  - `args` Array string arguments passed to worker. (Default=`process.argv.slice(2)`)
  - `silent` Boolean whether or not to send output to parent's stdio. (Default=`false`)
  - `uid` Number Sets the user identity of the process. (See setuid(2).)
  - `gid` Number Sets the group identity of the process. (See setgid(2).)

After calling `.setupMaster()` (or `.fork()`) this settings object will contain the settings, including the default values.

It is effectively frozen after being set, because `.setupMaster()` can only be called once.

This object is not supposed to be changed or set manually, by you.

cluster.setupMaster([settings])

- `settings` Object
  - `exec` String file path to worker file. (Default=`process.argv[1]`)
  - `args` Array string arguments passed to worker. (Default=`process.argv.slice(2)`)
  - `silent` Boolean whether or not to send output to parent's stdio. (Default=`false`)

`setupMaster` is used to change the default 'fork' behavior. Once called, the settings will be present in `cluster.settings`.

Note that:

- any settings changes only affect future calls to `.fork()` and have no effect on workers that are already running
- The *only* attribute of a worker that cannot be set via `.setupMaster()` is the `env` passed to `.fork()`
- the defaults above apply to the first call only, the defaults for later calls is the current value at the time of `cluster.setupMaster()` is called

Example:

```
const cluster = require('cluster');
cluster.setupMaster({
  exec: 'worker.js',
  args: ['--use', 'https'],
  silent: true
});
cluster.fork(); // https worker
cluster.setupMaster({
  args: ['--use', 'http']
});
cluster.fork(); // http worker
```

This can only be called from the master process.

cluster.worker

- Object

A reference to the current worker object. Not available in the master process.

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

if (cluster.isMaster) {
  console.log('I am master');
  cluster.fork();
  cluster.fork();
} else if (cluster.isWorker) {
  console.log(`I am worker #${cluster.worker.id}`);
}
```

cluster.workers

- Object

A hash that stores the active worker objects, keyed by `id` field. Makes it easy to loop through all the workers. It is only available in the master process.

A worker is removed from cluster.workers after the worker has disconnected *and* exited. The order between these two events cannot be determined in advance. However, it is guaranteed that the removal from the cluster.workers list happens before last `'disconnect'` or `'exit'` event is emitted.

```
// Go through all workers
function eachWorker(callback) {
  for (var id in cluster.workers) {
    callback(cluster.workers[id]);
```

```
  }
}
eachWorker((worker) => {
  worker.send('big announcement to all workers');
});
```

Should you wish to reference a worker over a communication channel, using the worker's unique id is the easiest way to find the worker.

```
socket.on('data', (id) => {
  var worker = cluster.workers[id];
});
```