Cluster

Stability: 2 - Stable

Clusters of Node.js processes can be used to run multiple instances of Node.js that can distribute workloads among their application threads. When process isolation is not needed, use the `worker_threads` module instead, which allows running multiple application threads within a single Node.js instance.

The cluster module allows easy creation of child processes that all share server ports.

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

const numCPUs = cpus().length;

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let 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);

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

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

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);
```

```
  // Fork workers.
  for (let 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);

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

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

```
$ node server.js
Primary 3596 is running
Worker 4324 started
Worker 4520 started
Worker 6056 started
Worker 5644 started
```

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 primary 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 primary 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 primary 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 primary, 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 primary process.
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. To listen on a unique port, generate a port number based on the cluster worker ID.

Node.js does not provide routing logic. It is therefore important to design an application 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 a 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, however. It is the application's responsibility to manage the worker pool based on its own needs.

Although a primary use case for the `cluster` module is networking, it can also be used for other use cases requiring worker processes.

Class: `Worker`

- Extends: {EventEmitter}

A `Worker` object contains all public information and method about a worker. In the primary 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()`.

Within a worker, `process.on('error')` may also be used.

Event: `'exit'`

- `code` {number} The exit code, if it exited normally.
- `signal` {string} The name of the signal (e.g. `'SIGHUP'`) that caused the process to be killed.

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

```
import cluster from 'cluster';

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!');
  }
});
```

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

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.

```
import cluster from 'cluster';

cluster.fork().on('listening', (address) => {
```

```
  // Worker is listening
});

const cluster = require('cluster');

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

It is not emitted in the worker.

Event: `'message'`

- `message` {Object}
- `handle` {undefined|Object}

Similar to the `'message'` event of `cluster`, but specific to this worker.

Within a worker, `process.on('message')` may also be used.

See `process` event: `'message'`.

Here is an example using the message system. It keeps a count in the primary process of the number of HTTP requests received by the workers:

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

if (cluster.isPrimary) {

  // Keep track of http requests
  let 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 = cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
```

```
  }

  for (const id in cluster.workers) {
    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 primary about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```

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

if (cluster.isPrimary) {

  // Keep track of http requests
  let 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 (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  for (const id in cluster.workers) {
    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 primary 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()`

- Returns: {cluster.Worker} A reference to `worker`.

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

Causes `.exitedAfterDisconnect` to be set.

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.

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.isPrimary) {
  const worker = cluster.fork();
  let 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');
  const 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.exitedAfterDisconnect`

- {boolean}

This property is `true` if the worker exited due to `.disconnect()`. If the worker exited any other way, it is `false`. If the worker has not exited, it is `undefined`.

The boolean `worker.exitedAfterDisconnect` allows distinguishing between voluntary and accidental exit, the primary may choose not to respawn a worker based on this value.

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

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

`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 primary via its IPC channel, `false` otherwise. A worker is connected to its primary after it has 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`.

```
import cluster from 'cluster';
import http from 'http';
import { cpus } from 'os';
import process from 'process';

const numCPUs = cpus().length;

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('fork', (worker) => {
    console.log('worker is dead:', worker.isDead());
  });

  cluster.on('exit', (worker, code, signal) => {
    console.log('worker is dead:', worker.isDead());
  });
} else {
  // Workers can share any TCP connection. In this case, it is an HTTP server.
  http.createServer((req, res) => {
```

```
    res.writeHead(200);
    res.end(`Current process\n ${process.pid}`);
    process.kill(process.pid);
  }).listen(8000);
}
```

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

if (cluster.isPrimary) {
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('fork', (worker) => {
    console.log('worker is dead:', worker.isDead());
  });

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

`worker.kill([signal])`

- `signal` {string} Name of the kill signal to send to the worker process. **Default:** `'SIGTERM'`

This function will kill the worker. In the primary worker, it does this by disconnecting the `worker.process`, and once disconnected, killing with `signal`. In the worker, it does it by killing the process with `signal`.

The `kill()` function kills the worker process without waiting for a graceful disconnect, it has the same behavior as `worker.process.kill()`.

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

In a worker, `process.kill()` exists, but it is not this function; it is `kill()`.

`worker.process`

- {ChildProcess}

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.

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

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

- `message` {Object}
- `sendHandle` {Handle}
- `options` {Object} The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:
  - `keepOpen` {boolean} A value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. **Default:** `false`.
- `callback` {Function}
- Returns: {boolean}

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

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

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

This example will echo back all messages from the primary:

```
if (cluster.isPrimary) {
  const worker = cluster.fork();
  worker.send('hi there');

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

Event: `'disconnect'`

- `worker` {cluster.Worker}

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` {cluster.Worker}
- `code` {number} The exit code, if it exited normally.
- `signal` {string} The name of the signal (e.g. `'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` {cluster.Worker}

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

```
const 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` {cluster.Worker}
- `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 primary.

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'` (UDPv4 or UDPv6)

Event: `'message'`

- `worker` {cluster.Worker}
- `message` {Object}
- `handle` {undefined|Object}

Emitted when the cluster primary receives a message from any worker.

See `child_process` event: `'message'`.

Event: `'online'`

- `worker` {cluster.Worker}

After forking a new worker, the worker should respond with an online message. When the primary receives an online message it will emit this event. The

difference between `'fork'` and `'online'` is that fork is emitted when the primary 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 `.setupPrimary()` is called.

The `settings` object is the `cluster.settings` object at the time `.setupPrimary()` was called and is advisory only, since multiple calls to `.setupPrimary()` 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 primary 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 primary process.

`cluster.fork([env])`

- `env` {Object} Key/value pairs to add to worker process environment.
- Returns: {cluster.Worker}

Spawn a new worker process.

This can only be called from the primary process.

`cluster.isMaster`

Deprecated alias for `cluster.isPrimary`.

`cluster.isPrimary`

- {boolean}

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

`cluster.isWorker`

- {boolean}

True if the process is not a primary (it is the negation of `cluster.isPrimary`).

`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 either the first worker is spawned, or `.setupPrimary()` is called, whichever 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` {string[]} 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` {string[]} String arguments passed to worker. **Default:** `process.argv.slice(2)`.
  - `cwd` {string} Current working directory of the worker process. **Default:** `undefined` (inherits from parent process).
  - `serialization` {string} Specify the kind of serialization used for sending messages between processes. Possible values are `'json'` and `'advanced'`. See Advanced serialization for `child_process` for more details. **Default:** `false`.
  - `silent` {boolean} Whether or not to send output to parent's stdio. **Default:** `false`.
  - `stdio` {Array} Configures the stdio of forked processes. Because the cluster module relies on IPC to function, this configuration must contain an `'ipc'` entry. When this option is provided, it overrides `silent`.
  - `uid` {number} Sets the user identity of the process. (See setuid(2).)
  - `gid` {number} Sets the group identity of the process. (See setgid(2).)
  - `inspectPort` {number|Function} Sets inspector port of worker. This can be a number, or a function that takes no arguments and returns a

number. By default each worker gets its own port, incremented from the primary's `process.debugPort`.
  - `windowsHide` {boolean} Hide the forked processes console window that would normally be created on Windows systems. **Default:** `false`.

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

This object is not intended to be changed or set manually.

`cluster.setupMaster([settings])`

Deprecated alias for `.setupPrimary()`.

`cluster.setupPrimary([settings])`

- `settings` {Object} See `cluster.settings`.

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

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 `.setupPrimary()` is the `env` passed to `.fork()`.

The defaults above apply to the first call only; the defaults for later calls are the current values at the time of `cluster.setupPrimary()` is called.

```
import cluster from 'cluster';

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

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

cluster.setupPrimary({
  exec: 'worker.js',
  args: ['--use', 'https'],
```

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

This can only be called from the primary process.

`cluster.worker`

- {Object}

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

```
import cluster from 'cluster';

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

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

if (cluster.isPrimary) {
  console.log('I am primary');
  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. This makes it easy to loop through all the workers. It is only available in the primary 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 the last `'disconnect'` or `'exit'` event is emitted.

```mjs
import cluster from 'cluster';

for (const worker of Object.values(cluster.workers)) {
  worker.send('big announcement to all workers');
}
```

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

for (const worker of Object.values(cluster.workers)) {
  worker.send('big announcement to all workers');
}
```