/\*\*

\* 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.

\*

\* ```js

\* 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`);

\* }

\* ```

\*

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

\*

\* ```console

\* $ 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.

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/cluster.js)

\*/

declare module 'cluster' {

import \* as child from 'node:child\_process';

import EventEmitter = require('node:events');

import \* as net from 'node:net';

export interface ClusterSettings {

execArgv?: string[] | undefined; // default: process.execArgv

exec?: string | undefined;

args?: string[] | undefined;

silent?: boolean | undefined;

stdio?: any[] | undefined;

uid?: number | undefined;

gid?: number | undefined;

inspectPort?: number | (() => number) | undefined;

}

export interface Address {

address: string;

port: number;

addressType: number | 'udp4' | 'udp6'; // 4, 6, -1, "udp4", "udp6"

}

/\*\*

\* 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`.

\* @since v0.7.0

\*/

export class Worker extends EventEmitter {

/\*\*

\* 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`.

\* @since v0.8.0

\*/

id: number;

/\*\*

\* 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.

\* @since v0.7.0

\*/

process: child.ChildProcess;

/\*\*

\* 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:

\*

\* ```js

\* if (cluster.isPrimary) {

\* const worker = cluster.fork();

\* worker.send('hi there');

\*

\* } else if (cluster.isWorker) {

\* process.on('message', (msg) => {

\* process.send(msg);

\* });

\* }

\* ```

\* @since v0.7.0

\* @param options The `options` argument, if present, is an object used to parameterize the sending of certain types of handles. `options` supports the following properties:

\*/

send(message: child.Serializable, callback?: (error: Error | null) => void): boolean;

send(message: child.Serializable, sendHandle: child.SendHandle, callback?: (error: Error | null) => void): boolean;

send(message: child.Serializable, sendHandle: child.SendHandle, options?: child.MessageOptions, callback?: (error: Error | null) => void): boolean;

/\*\*

\* This function will kill the worker. In the primary, 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`.

\*

\* Because `kill()` attempts to gracefully disconnect the worker process, it is

\* susceptible to waiting indefinitely for the disconnect to complete. For example,

\* if the worker enters an infinite loop, a graceful disconnect will never occur.

\* If the graceful disconnect behavior is not needed, use `worker.process.kill()`.

\*

\* Causes `.exitedAfterDisconnect` to be set.

\*

\* This method is aliased as `worker.destroy()` for backward compatibility.

\*

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

\* it is `kill()`.

\* @since v0.9.12

\* @param [signal='SIGTERM'] Name of the kill signal to send to the worker process.

\*/

kill(signal?: string): void;

destroy(signal?: string): void;

/\*\*

\* 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.

\*

\* ```js

\* 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

\* }

\* });

\* }

\* ```

\* @since v0.7.7

\* @return A reference to `worker`.

\*/

disconnect(): void;

/\*\*

\* 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.

\* @since v0.11.14

\*/

isConnected(): boolean;

/\*\*

\* This function returns `true` if the worker's process has terminated (either

\* because of exiting or being signaled). Otherwise, it returns `false`.

\*

\* ```js

\* 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);

\* }

\* ```

\* @since v0.11.14

\*/

isDead(): boolean;

/\*\*

\* This property is `true` if the worker exited due to `.kill()` or`.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.

\*

\* ```js

\* 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();

\* ```

\* @since v6.0.0

\*/

exitedAfterDisconnect: boolean;

/\*\*

\* events.EventEmitter

\* 1. disconnect

\* 2. error

\* 3. exit

\* 4. listening

\* 5. message

\* 6. online

\*/

addListener(event: string, listener: (...args: any[]) => void): this;

addListener(event: 'disconnect', listener: () => void): this;

addListener(event: 'error', listener: (error: Error) => void): this;

addListener(event: 'exit', listener: (code: number, signal: string) => void): this;

addListener(event: 'listening', listener: (address: Address) => void): this;

addListener(event: 'message', listener: (message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

addListener(event: 'online', listener: () => void): this;

emit(event: string | symbol, ...args: any[]): boolean;

emit(event: 'disconnect'): boolean;

emit(event: 'error', error: Error): boolean;

emit(event: 'exit', code: number, signal: string): boolean;

emit(event: 'listening', address: Address): boolean;

emit(event: 'message', message: any, handle: net.Socket | net.Server): boolean;

emit(event: 'online'): boolean;

on(event: string, listener: (...args: any[]) => void): this;

on(event: 'disconnect', listener: () => void): this;

on(event: 'error', listener: (error: Error) => void): this;

on(event: 'exit', listener: (code: number, signal: string) => void): this;

on(event: 'listening', listener: (address: Address) => void): this;

on(event: 'message', listener: (message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

on(event: 'online', listener: () => void): this;

once(event: string, listener: (...args: any[]) => void): this;

once(event: 'disconnect', listener: () => void): this;

once(event: 'error', listener: (error: Error) => void): this;

once(event: 'exit', listener: (code: number, signal: string) => void): this;

once(event: 'listening', listener: (address: Address) => void): this;

once(event: 'message', listener: (message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

once(event: 'online', listener: () => void): this;

prependListener(event: string, listener: (...args: any[]) => void): this;

prependListener(event: 'disconnect', listener: () => void): this;

prependListener(event: 'error', listener: (error: Error) => void): this;

prependListener(event: 'exit', listener: (code: number, signal: string) => void): this;

prependListener(event: 'listening', listener: (address: Address) => void): this;

prependListener(event: 'message', listener: (message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

prependListener(event: 'online', listener: () => void): this;

prependOnceListener(event: string, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'disconnect', listener: () => void): this;

prependOnceListener(event: 'error', listener: (error: Error) => void): this;

prependOnceListener(event: 'exit', listener: (code: number, signal: string) => void): this;

prependOnceListener(event: 'listening', listener: (address: Address) => void): this;

prependOnceListener(event: 'message', listener: (message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

prependOnceListener(event: 'online', listener: () => void): this;

}

export interface Cluster extends EventEmitter {

disconnect(callback?: () => void): void;

fork(env?: any): Worker;

/\*\* @deprecated since v16.0.0 - use isPrimary. \*/

readonly isMaster: boolean;

readonly isPrimary: boolean;

readonly isWorker: boolean;

schedulingPolicy: number;

readonly settings: ClusterSettings;

/\*\* @deprecated since v16.0.0 - use setupPrimary. \*/

setupMaster(settings?: ClusterSettings): void;

/\*\*

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

\*/

setupPrimary(settings?: ClusterSettings): void;

readonly worker?: Worker | undefined;

readonly workers?: NodeJS.Dict<Worker> | undefined;

readonly SCHED\_NONE: number;

readonly SCHED\_RR: number;

/\*\*

\* events.EventEmitter

\* 1. disconnect

\* 2. exit

\* 3. fork

\* 4. listening

\* 5. message

\* 6. online

\* 7. setup

\*/

addListener(event: string, listener: (...args: any[]) => void): this;

addListener(event: 'disconnect', listener: (worker: Worker) => void): this;

addListener(event: 'exit', listener: (worker: Worker, code: number, signal: string) => void): this;

addListener(event: 'fork', listener: (worker: Worker) => void): this;

addListener(event: 'listening', listener: (worker: Worker, address: Address) => void): this;

addListener(event: 'message', listener: (worker: Worker, message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

addListener(event: 'online', listener: (worker: Worker) => void): this;

addListener(event: 'setup', listener: (settings: ClusterSettings) => void): this;

emit(event: string | symbol, ...args: any[]): boolean;

emit(event: 'disconnect', worker: Worker): boolean;

emit(event: 'exit', worker: Worker, code: number, signal: string): boolean;

emit(event: 'fork', worker: Worker): boolean;

emit(event: 'listening', worker: Worker, address: Address): boolean;

emit(event: 'message', worker: Worker, message: any, handle: net.Socket | net.Server): boolean;

emit(event: 'online', worker: Worker): boolean;

emit(event: 'setup', settings: ClusterSettings): boolean;

on(event: string, listener: (...args: any[]) => void): this;

on(event: 'disconnect', listener: (worker: Worker) => void): this;

on(event: 'exit', listener: (worker: Worker, code: number, signal: string) => void): this;

on(event: 'fork', listener: (worker: Worker) => void): this;

on(event: 'listening', listener: (worker: Worker, address: Address) => void): this;

on(event: 'message', listener: (worker: Worker, message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

on(event: 'online', listener: (worker: Worker) => void): this;

on(event: 'setup', listener: (settings: ClusterSettings) => void): this;

once(event: string, listener: (...args: any[]) => void): this;

once(event: 'disconnect', listener: (worker: Worker) => void): this;

once(event: 'exit', listener: (worker: Worker, code: number, signal: string) => void): this;

once(event: 'fork', listener: (worker: Worker) => void): this;

once(event: 'listening', listener: (worker: Worker, address: Address) => void): this;

once(event: 'message', listener: (worker: Worker, message: any, handle: net.Socket | net.Server) => void): this; // the handle is a net.Socket or net.Server object, or undefined.

once(event: 'online', listener: (worker: Worker) => void): this;

once(event: 'setup', listener: (settings: ClusterSettings) => void): this;

prependListener(event: string, listener: (...args: any[]) => void): this;

prependListener(event: 'disconnect', listener: (worker: Worker) => void): this;

prependListener(event: 'exit', listener: (worker: Worker, code: number, signal: string) => void): this;

prependListener(event: 'fork', listener: (worker: Worker) => void): this;

prependListener(event: 'listening', listener: (worker: Worker, address: Address) => void): this;

// the handle is a net.Socket or net.Server object, or undefined.

prependListener(event: 'message', listener: (worker: Worker, message: any, handle?: net.Socket | net.Server) => void): this;

prependListener(event: 'online', listener: (worker: Worker) => void): this;

prependListener(event: 'setup', listener: (settings: ClusterSettings) => void): this;

prependOnceListener(event: string, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'disconnect', listener: (worker: Worker) => void): this;

prependOnceListener(event: 'exit', listener: (worker: Worker, code: number, signal: string) => void): this;

prependOnceListener(event: 'fork', listener: (worker: Worker) => void): this;

prependOnceListener(event: 'listening', listener: (worker: Worker, address: Address) => void): this;

// the handle is a net.Socket or net.Server object, or undefined.

prependOnceListener(event: 'message', listener: (worker: Worker, message: any, handle: net.Socket | net.Server) => void): this;

prependOnceListener(event: 'online', listener: (worker: Worker) => void): this;

prependOnceListener(event: 'setup', listener: (settings: ClusterSettings) => void): this;

}

const cluster: Cluster;

export default cluster;

}

declare module 'node:cluster' {

export \* from 'cluster';

export { default as default } from 'cluster';

}