# Node.js v0.8.0 Manual & Documentation

PDF created by Mirco Zeiss
Content by node.js and its creators

June 27, 2012

**Abstract**

I sometimes prefer manuals on paper to online references. Taking notes and writing down little hints is much easier. That's why I created this pdf version of the current node.js manual & documentation. However think about nature before printing too many documents!

All the content is taken from the node.js homepage / its github repository.

I used pandoc to convert the .markdown files to .tex files. They were then converted into a .pdf file with pdfLaTeX.

You can find the project on github.

# Contents

# About this Documentation

The goal of this documentation is to comprehensively explain the Node.js API, both from a reference as well as a conceptual point of view. Each section describes a built-in module or high-level concept.

Where appropriate, property types, method arguments, and the arguments provided to event handlers are detailed in a list underneath the topic heading.

Every `.html` document has a corresponding `.json` document presenting the same information in a structured manner. This feature is experimental, and added for the benefit of IDEs and other utilities that wish to do programmatic things with the documentation.

Every `.html` and `.json` file is generated based on the corresponding `.markdown` file in the `doc/api/` folder in node's source tree. The documentation is generated using the `tools/doc/generate.js` program. The HTML template is located at `doc/template.html`.

## Stability Index

Throughout the documentation, you will see indications of a section's stability. The Node.js API is still somewhat changing, and as it matures, certain parts are more reliable than others. Some are so proven, and so relied upon, that they are unlikely to ever change at all. Others are brand new and experimental, or known to be hazardous and in the process of being redesigned.

The notices look like this:

```
Stability: 1 Experimental
```

The stability indices are as follows:

- **0 - Deprecated** This feature is known to be problematic, and changes are planned. Do not rely on it. Use of the feature may cause warnings. Backwards compatibility should not be expected.
- **1 - Experimental** This feature was introduced recently, and may change or be removed in future versions. Please try it out and provide feedback. If it addresses a use-case that is important to you, tell the node core team.
- **2 - Unstable** The API is in the process of settling, but has not yet had sufficient real-world testing to be considered stable. Backwards-compatibility will be maintained if reasonable.
- **3 - Stable** The API has proven satisfactory, but cleanup in the underlying code may cause minor changes. Backwards-compatibility is guaranteed.
- **4 - API Frozen** This API has been tested extensively in production and is unlikely to ever have to change.
- **5 - Locked** Unless serious bugs are found, this code will not ever change. Please do not suggest changes in this area; they will be refused.

## JSON Output

```
Stability: 1 - Experimental
```

Every HTML file in the markdown has a corresponding JSON file with the same data.

This feature is new as of node v0.6.12. It is experimental.

## Synopsis

An example of a web server written with Node which responds with 'Hello World':

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

To run the server, put the code into a file called `example.js` and execute it with the node program

```
> node example.js
Server running at http://127.0.0.1:8124/
```

All of the examples in the documentation can be run similarly.

# Global Objects

These objects are available in all modules. Some of these objects aren't actually in the global scope but in the module scope - this will be noted.

## global

- {Object} The global namespace object.

In browsers, the top-level scope is the global scope. That means that in browsers if you're in the global scope `var something` will define a global variable. In Node this is different. The top-level scope is not the global scope; `var something` inside a Node module will be local to that module.

## process

- {Object}

The process object. See the process object section.

## console

- {Object}

Used to print to stdout and stderr. See the stdio section.

## Class: Buffer

- {Function}

Used to handle binary data. See the buffer section

## require()

- {Function}

To require modules. See the Modules section. `require` isn't actually a global but rather local to each module.

### require.resolve()

Use the internal `require()` machinery to look up the location of a module, but rather than loading the module, just return the resolved filename.

### require.cache

- {Object}

Modules are cached in this object when they are required. By deleting a key value from this object, the next `require` will reload the module.

### require.extensions

- {Array}

Instruct `require` on how to handle certain file extensions.

Process files with the extension `.sjs` as `.js`:

```
require.extensions['.sjs'] = require.extensions['.js'];
```

Write your own extension handler:

```
require.extensions['.sjs'] = function(module, filename) {
  var content = fs.readFileSync(filename, 'utf8');
  // Parse the file content and give to module.exports
  module.exports = content;
};
```

### __filename

- {String}

The filename of the code being executed. This is the resolved absolute path of this code file. For a main program this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.

Example: running `node example.js` from `/Users/mjr`

```
console.log(__filename);
// /Users/mjr/example.js
```

`__filename` isn't actually a global but rather local to each module.

### __dirname

- {String}

The name of the directory that the currently executing script resides in.

Example: running `node example.js` from `/Users/mjr`

```
console.log(__dirname);
// /Users/mjr
```

`__dirname` isn't actually a global but rather local to each module.

### module

- {Object}

A reference to the current module. In particular `module.exports` is the same as the `exports` object. `module` isn't actually a global but rather local to each module.

See the module system documentation for more information.

### exports

An object which is shared between all instances of the current module and made accessible through `require()`. `exports` is the same as the `module.exports` object. `exports` isn't actually a global but rather local to each module.

See the module system documentation for more information.

See the module section for more information.

### setTimeout(cb, ms)

### clearTimeout(t)

### setInterval(cb, ms)

### clearInterval(t)

The timer functions are global variables. See the timers section.

# console

```
Stability: 4 - API Frozen
```

- {Object}

For printing to stdout and stderr. Similar to the console object functions provided by most web browsers, here the output is sent to stdout or stderr.

## console.log([data], [...])

Prints to stdout with newline. This function can take multiple arguments in a `printf()`-like way. Example:

```
console.log('count: %d', count);
```

If formatting elements are not found in the first string then `util.inspect` is used on each argument. See util.format() for more information.

## console.info([data], [...])

Same as `console.log`.

## console.error([data], [...])

Same as `console.log` but prints to stderr.

## console.warn([data], [...])

Same as `console.error`.

## console.dir(obj)

Uses `util.inspect` on `obj` and prints resulting string to stdout.

## console.time(label)

Mark a time.

## console.timeEnd(label)

Finish timer, record output. Example:

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
  ;
}
console.timeEnd('100-elements');
```

## console.trace(label)

Print a stack trace to stderr of the current position.

## console.assert(expression, [message])

Same as assert.ok() where if the `expression` evaluates as `false` throw an AssertionError with `message`.

# Timers

```
Stability: 5 - Locked
```

All of the timer functions are globals. You do not need to `require()` this module in order to use them.

## setTimeout(callback, delay, [arg], [...])

To schedule execution of a one-time `callback` after `delay` milliseconds. Returns a `timeoutId` for possible use with `clearTimeout()`. Optionally you can also pass arguments to the callback.

It is important to note that your callback will probably not be called in exactly `delay` milliseconds - Node.js makes no guarantees about the exact timing of when the callback will fire, nor of the ordering things will fire in. The callback will be called as close as possible to the time specified.

## clearTimeout(timeoutId)

Prevents a timeout from triggering.

## setInterval(callback, delay, [arg], [...])

To schedule the repeated execution of `callback` every `delay` milliseconds. Returns a `intervalId` for possible use with `clearInterval()`. Optionally you can also pass arguments to the callback.

## clearInterval(intervalId)

Stops a interval from triggering.

# Modules

```
Stability: 5 - Locked
```

Node has a simple module loading system. In Node, files and modules are in one-to-one correspondence. As an example, `foo.js` loads the module `circle.js` in the same directory.

The contents of `foo.js`:

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
           + circle.area(4));
```

The contents of `circle.js`:

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

exports.circumference = function (r) {
  return 2 * PI * r;
};
```

The module `circle.js` has exported the functions `area()` and `circumference()`. To export an object, add to the special `exports` object.

Variables local to the module will be private. In this example the variable `PI` is private to `circle.js`.

The module system is implemented in the `require("module")` module.

## Cycles

When there are circular `require()` calls, a module might not be done being executed when it is returned.

Consider this situation:

`a.js`:

```
console.log('a starting');
exports.done = false;
var b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

`b.js`:

```
console.log('b starting');
exports.done = false;
var a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

`main.js`:

```
console.log('main starting');
var a = require('./a.js');
var b = require('./b.js');
console.log('in main, a.done=%j, b.done=%j', a.done, b.done);
```

When `main.js` loads `a.js`, then `a.js` in turn loads `b.js`. At that point, `b.js` tries to load `a.js`. In order to prevent an infinite loop an **unfinished copy** of the `a.js` exports object is returned to the `b.js` module. `b.js` then finishes loading, and its exports object is provided to the `a.js` module.

By the time `main.js` has loaded both modules, they're both finished. The output of this program would thus be:

```
$ node main.js
main starting
a starting
b starting
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done=true, b.done=true
```

If you have cyclic module dependencies in your program, make sure to plan accordingly.

## Core Modules

Node has several modules compiled into the binary. These modules are described in greater detail elsewhere in this documentation.

The core modules are defined in node's source in the `lib/` folder.

Core modules are always preferentially loaded if their identifier is passed to `require()`. For instance, `require('http')` will always return the built in HTTP module, even if there is a file by that name.

## File Modules

If the exact filename is not found, then node will attempt to load the required filename with the added extension of `.js`, `.json`, and then `.node`.

`.js` files are interpreted as JavaScript text files, and `.json` files are parsed as JSON text files. `.node` files are interpreted as compiled addon modules loaded with `dlopen`.

A module prefixed with `'/'` is an absolute path to the file. For example, `require('/home/marco/foo.js')` will load the file at `/home/marco/foo.js`.

A module prefixed with `'./'` is relative to the file calling `require()`. That is, `circle.js` must be in the same directory as `foo.js` for `require('./circle')` to find it.

Without a leading '/' or './' to indicate a file, the module is either a "core module" or is loaded from a `node_modules` folder.

If the given path does not exist, `require()` will throw an Error with its `code` property set to `'MODULE_NOT_FOUND'`.

## Loading from `node_modules` Folders

If the module identifier passed to `require()` is not a native module, and does not begin with `'/'`, `'../'`, or `'./'`, then node starts at the parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location.

If it is not found there, then it moves to the parent directory, and so on, until the root of the tree is reached.

For example, if the file at `'/home/ry/projects/foo.js'` called `require('bar.js')`, then node would look in the following locations, in this order:

- `/home/ry/projects/node_modules/bar.js`
- `/home/ry/node_modules/bar.js`
- `/home/node_modules/bar.js`
- `/node_modules/bar.js`

This allows programs to localize their dependencies, so that they do not clash.

## Folders as Modules

It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to that library. There are three ways in which a folder may be passed to `require()` as an argument.

The first is to create a `package.json` file in the root of the folder, which specifies a `main` module. An example package.json file might look like this:

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

If this was in a folder at `./some-library`, then `require('./some-library')` would attempt to load `./some-library/lib/some-library.js`.

This is the extent of Node's awareness of package.json files.

If there is no package.json file present in the directory, then node will attempt to load an `index.js` or `index.node` file out of that directory. For example, if there was no package.json file in the above example, then `require('./some-library')` would attempt to load:

- `./some-library/index.js`
- `./some-library/index.node`

## Caching

Modules are cached after the first time they are loaded. This means (among other things) that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.

Multiple calls to `require('foo')` may not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

If you want to have a module execute code multiple times, then export a function, and call that function.

### Module Caching Caveats

Modules are cached based on their resolved filename. Since modules may resolve to a different filename based on the location of the calling module (loading from `node_modules` folders), it is not a *guarantee* that `require('foo')` will always return the exact same object, if it would resolve to different files.

## The `module` Object

- {Object}

In each module, the `module` free variable is a reference to the object representing the current module. In particular `module.exports` is the same as the `exports` object. `module` isn't actually a global but rather local to each module.

### module.exports

- {Object}

The `exports` object is created by the Module system. Sometimes this is not acceptable, many want their module to be an instance of some class. To do this assign the desired export object to `module.exports`. For example suppose we were making a module called `a.js`

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

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
```

```
setTimeout(function() {
  module.exports.emit('ready');
}, 1000);
```

Then in another file we could do

```
var a = require('./a');
a.on('ready', function() {
  console.log('module a is ready');
});
```

Note that assignment to `module.exports` must be done immediately. It cannot be done in any callbacks. This does not work:

x.js:

```
setTimeout(function() {
  module.exports = { a: "hello" };
}, 0);
```

y.js:

```
var x = require('./x');
console.log(x.a);
```

#### module.require(id)

- `id` {String}
- Return: {Object} `exports` from the resolved module

The `module.require` method provides a way to load a module as if `require()` was called from the original module.

Note that in order to do this, you must get a reference to the `module` object. Since `require()` returns the `exports`, and the `module` is typically *only* available within a specific module's code, it must be explicitly exported in order to be used.

#### module.id

- {String}

The identifier for the module. Typically this is the fully resolved filename.

#### module.filename

- {String}

The fully resolved filename to the module.

#### module.loaded

- {Boolean}

Whether or not the module is done loading, or is in the process of loading.

#### module.parent

- {Module Object}

The module that required this one.

**module.children**

- {Array}

The module objects required by this one.

## All Together...

To get the exact filename that will be loaded when `require()` is called, use the `require.resolve()` function.

Putting together all of the above, here is the high-level algorithm in pseudocode of what require.resolve does:

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
3. LOAD_NODE_MODULES(X, dirname(Y))
4. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.node is a file, load X.node as binary addon.  STOP

LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
2. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
3. If X/index.node is a file, load X/index.node as binary addon.  STOP

LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let ROOT = index of first instance of "node_modules" in PARTS, or 0
3. let I = count of PARTS - 1
4. let DIRS = []
5. while I > ROOT,
   a. if PARTS[I] = "node_modules" CONTINUE
   c. DIR = path join(PARTS[0 .. I] + "node_modules")
   b. DIRS = DIRS + DIR
   c. let I = I - 1
6. return DIRS
```

## Loading from the global folders

If the `NODE_PATH` environment variable is set to a colon-delimited list of absolute paths, then node will search those paths for modules if they are not found elsewhere. (Note: On Windows, `NODE_PATH` is delimited by semicolons instead of colons.)

Additionally, node will search in the following locations:

- 1: `$HOME/.node_modules`

- 2: `$HOME/.node_libraries`
- 3: `$PREFIX/lib/node`

Where `$HOME` is the user's home directory, and `$PREFIX` is node's configured `node_prefix`.

These are mostly for historic reasons. You are highly encouraged to place your dependencies locally in `node_modules` folders. They will be loaded faster, and more reliably.

### Accessing the main module

When a file is run directly from Node, `require.main` is set to its `module`. That means that you can determine whether a file has been run directly by testing

```
require.main === module
```

For a file `foo.js`, this will be `true` if run via `node foo.js`, but `false` if run by `require('./foo')`.

Because `module` provides a `filename` property (normally equivalent to `__filename`), the entry point of the current application can be obtained by checking `require.main.filename`.

### Addenda: Package Manager Tips

The semantics of Node's `require()` function were designed to be general enough to support a number of sane directory structures. Package manager programs such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from Node modules without modification.

Below we give a suggested directory structure that could work:

Let's say that we wanted to have the folder at `/usr/lib/node/<some-package>/<some-version>` hold the contents of a specific version of a package.

Packages can depend on one another. In order to install package `foo`, you may have to install a specific version of package `bar`. The `bar` package may itself have dependencies, and in some cases, these dependencies may even collide or form cycles.

Since Node looks up the `realpath` of any modules it loads (that is, resolves symlinks), and then looks for their dependencies in the `node_modules` folders as described above, this situation is very simple to resolve with the following architecture:

- `/usr/lib/node/foo/1.2.3/` - Contents of the `foo` package, version 1.2.3.
- `/usr/lib/node/bar/4.3.2/` - Contents of the `bar` package that `foo` depends on.
- `/usr/lib/node/foo/1.2.3/node_modules/bar` - Symbolic link to `/usr/lib/node/bar/4.3.2/`.
- `/usr/lib/node/bar/4.3.2/node_modules/*` - Symbolic links to the packages that `bar` depends on.

Thus, even if a cycle is encountered, or if there are dependency conflicts, every module will be able to get a version of its dependency that it can use.

When the code in the `foo` package does `require('bar')`, it will get the version that is symlinked into `/usr/lib/node/foo/1.2.3/node_modules/bar`. Then, when the code in the `bar` package calls `require('quux')`, it'll get the version that is symlinked into `/usr/lib/node/bar/4.3.2/node_modules/quux`.

Furthermore, to make the module lookup process even more optimal, rather than putting packages directly in `/usr/lib/node`, we could put them in `/usr/lib/node_modules/<name>/<version>`. Then node will not bother looking for missing dependencies in `/usr/node_modules` or `/node_modules`.

In order to make modules available to the node REPL, it might be useful to also add the `/usr/lib/node_modules` folder to the `$NODE_PATH` environment variable. Since the module lookups using `node_modules` folders are all relative, and based on the real path of the files making the calls to `require()`, the packages themselves can be anywhere.

# Addons

Addons are dynamically linked shared objects. They can provide glue to C and C++ libraries. The API (at the moment) is rather complex, involving knowledge of several libraries:

- V8 JavaScript, a C++ library. Used for interfacing with JavaScript: creating objects, calling functions, etc. Documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node source tree), which is also available online.
- libuv, C event loop library. Anytime one needs to wait for a file descriptor to become readable, wait for a timer, or wait for a signal to received one will need to interface with libuv. That is, if you perform any I/O, libuv will need to be used.
- Internal Node libraries. Most importantly is the `node::ObjectWrap` class which you will likely want to derive from.
- Others. Look in `deps/` for what else is available.

Node statically compiles all its dependencies into the executable. When compiling your module, you don't need to worry about linking to any of these libraries.

## Hello world

To get started let's make a small Addon which is the C++ equivalent of the following JavaScript code:

```
exports.hello = function() { return 'world'; };
```

First we create a file `hello.cc`:

```
#include <node.h>
#include <v8.h>

using namespace v8;

Handle<Value> Method(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("world"));
}

void init(Handle<Object> target) {
  target->Set(String::NewSymbol("hello"),
      FunctionTemplate::New(Method)->GetFunction());
}
NODE_MODULE(hello, init)
```

Note that all Node addons must export an initialization function:

```
void Initialize (Handle<Object> target);
NODE_MODULE(module_name, Initialize)
```

There is no semi-colon after `NODE_MODULE` as it's not a function (see `node.h`).

The `module_name` needs to match the filename of the final binary (minus the .node suffix).

The source code needs to be built into `hello.node`, the binary Addon. To do this we create a file called `binding.gyp` which describes the configuration to build your module in a JSON-like format. This file gets compiled by node-gyp.

```
{
  "targets": [
    {
      "target_name": "hello",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

The next step is to generate the appropriate project build files for the current platform. Use `node-gyp configure` for that.

Now you will have either a `Makefile` (on Unix platforms) or a `vcxproj` file (on Windows) in the `build/` directory. Next invoke the `node-gyp build` command.

Now you have your compiled `.node` bindings file! The compiled bindings end up in `build/Release/`.

You can now use the binary addon in a Node project `hello.js` by pointing `require` to the recently built `hello.node` module:

```
var addon = require('./build/Release/hello');

console.log(addon.hello()); // 'world'
```

Please see patterns below for further information or https://github.com/arturadib/node-qt for an example in production.

## Addon patterns

Below are some addon patterns to help you get started. Consult the online v8 reference for help with the various v8 calls, and v8's Embedder's Guide for an explanation of several concepts used such as handles, scopes, function templates, etc.

In order to use these examples you need to compile them using `node-gyp`. Create the following `binding.gyp` file:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "addon.cc" ]
    }
  ]
}
```

In cases where there is more than one `.cc` file, simply add the file name to the `sources` array, e.g.:

```
"sources": ["addon.cc", "myexample.cc"]
```

Now that you have your `binding.gyp` ready, you can configure and build the addon:

```
$ node-gyp configure build
```

### Function arguments

The following pattern illustrates how to read arguments from JavaScript function calls and return a result. This is the main and only needed source `addon.cc`:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>

using namespace v8;

Handle<Value> Add(const Arguments& args) {
  HandleScope scope;

  if (args.Length() < 2) {
    ThrowException(Exception::TypeError(String::New("Wrong number of arguments")));
    return scope.Close(Undefined());
  }

  if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    ThrowException(Exception::TypeError(String::New("Wrong arguments")));
    return scope.Close(Undefined());
```

```
  }

  Local<Number> num = Number::New(args[0]->NumberValue() +
      args[1]->NumberValue());
  return scope.Close(num);
}

void Init(Handle<Object> target) {
  target->Set(String::NewSymbol("add"),
      FunctionTemplate::New(Add)->GetFunction());
}

NODE_MODULE(addon, Init)
```

You can test it with the following JavaScript snippet:

```
var addon = require('./build/Release/addon');

console.log( 'This should be eight:', addon.add(3,5) );
```

### Callbacks

You can pass JavaScript functions to a C++ function and execute them from there. Here's `addon.cc`:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>

using namespace v8;

Handle<Value> RunCallback(const Arguments& args) {
  HandleScope scope;

  Local<Function> cb = Local<Function>::Cast(args[0]);
  const unsigned argc = 1;
  Local<Value> argv[argc] = { Local<Value>::New(String::New("hello world")) };
  cb->Call(Context::GetCurrent()->Global(), argc, argv);

  return scope.Close(Undefined());
}

void Init(Handle<Object> target) {
  target->Set(String::NewSymbol("runCallback"),
      FunctionTemplate::New(RunCallback)->GetFunction());
}

NODE_MODULE(addon, Init)
```

To test it run the following JavaScript snippet:

```
var addon = require('./build/Release/addon');

addon.runCallback(function(msg){
  console.log(msg); // 'hello world'
});
```

### Object factory

You can create and return new objects from within a C++ function with this `addon.cc` pattern, which returns an object with property `msg` that echoes the string passed to `createObject()`:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
```

```
using namespace v8;

Handle<Value> CreateObject(const Arguments& args) {
  HandleScope scope;

  Local<Object> obj = Object::New();
  obj->Set(String::NewSymbol("msg"), args[0]->ToString());

  return scope.Close(obj);
}

void Init(Handle<Object> target) {
  target->Set(String::NewSymbol("createObject"),
      FunctionTemplate::New(CreateObject)->GetFunction());
}

NODE_MODULE(addon, Init)
```

To test it in JavaScript:

```
var addon = require('./build/Release/addon');

var obj1 = addon.createObject('hello');
var obj2 = addon.createObject('world');
console.log(obj1.msg+' '+obj2.msg); // 'hello world'
```

### Function factory

This pattern illustrates how to create and return a JavaScript function that wraps a C++ function:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>

using namespace v8;

Handle<Value> MyFunction(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("hello world"));
}

Handle<Value> CreateFunction(const Arguments& args) {
  HandleScope scope;

  Local<FunctionTemplate> tpl = FunctionTemplate::New(MyFunction);
  Local<Function> fn = tpl->GetFunction();
  fn->SetName(String::NewSymbol("theFunction")); // omit this to make it anonymous

  return scope.Close(fn);
}

void Init(Handle<Object> target) {
  target->Set(String::NewSymbol("createFunction"),
      FunctionTemplate::New(CreateFunction)->GetFunction());
}

NODE_MODULE(addon, Init)
```

To test:

```
var addon = require('./build/Release/addon');
```

```
var fn = addon.createFunction();
console.log(fn()); // 'hello world'
```

### Wrapping C++ objects

Here we will create a wrapper for a C++ object/class `MyObject` that can be instantiated in JavaScript through the `new` operator. First prepare the main module `addon.cc`:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

void InitAll(Handle<Object> target) {
  MyObject::Init(target);
}

NODE_MODULE(addon, InitAll)
```

Then in `myobject.h` make your wrapper inherit from `node::ObjectWrap`:

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Handle<v8::Object> target);

 private:
  MyObject();
  ~MyObject();

  static v8::Handle<v8::Value> New(const v8::Arguments& args);
  static v8::Handle<v8::Value> PlusOne(const v8::Arguments& args);
  double counter_;
};

#endif
```

And in `myobject.cc` implement the various methods that you want to expose. Here we expose the method `plusOne` by adding it to the constructor's prototype:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

MyObject::MyObject() {};
MyObject::~MyObject() {};

void MyObject::Init(Handle<Object> target) {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(New);
  tpl->SetClassName(String::NewSymbol("MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);
  // Prototype
  tpl->PrototypeTemplate()->Set(String::NewSymbol("plusOne"),
      FunctionTemplate::New(PlusOne)->GetFunction());
```

```
  Persistent<Function> constructor = Persistent<Function>::New(tpl->GetFunction());
  target->Set(String::NewSymbol("MyObject"), constructor);
}

Handle<Value> MyObject::New(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = new MyObject();
  obj->counter_ = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
  obj->Wrap(args.This());

  return args.This();
}

Handle<Value> MyObject::PlusOne(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.This());
  obj->counter_ += 1;

  return scope.Close(Number::New(obj->counter_));
}
```

Test it with:

```
var addon = require('./build/Release/addon');

var obj = new addon.MyObject(10);
console.log( obj.plusOne() ); // 11
console.log( obj.plusOne() ); // 12
console.log( obj.plusOne() ); // 13
```

### Factory of wrapped objects

This is useful when you want to be able to create native objects without explicitly instantiating them with the `new` operator in JavaScript, e.g.

```
var obj = addon.createObject();
// instead of:
// var obj = new addon.Object();
```

Let's register our `createObject` method in `addon.cc`:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

Handle<Value> CreateObject(const Arguments& args) {
  HandleScope scope;
  return scope.Close(MyObject::NewInstance(args));
}

void InitAll(Handle<Object> target) {
  MyObject::Init();

  target->Set(String::NewSymbol("createObject"),
      FunctionTemplate::New(CreateObject)->GetFunction());
}

NODE_MODULE(addon, InitAll)
```

In `myobject.h` we now introduce the static method `NewInstance` that takes care of instantiating the object (i.e. it does the job of `new` in JavaScript):

```
#define BUILDING_NODE_EXTENSION
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init();
  static v8::Handle<v8::Value> NewInstance(const v8::Arguments& args);

 private:
  MyObject();
  ~MyObject();

  static v8::Persistent<v8::Function> constructor;
  static v8::Handle<v8::Value> New(const v8::Arguments& args);
  static v8::Handle<v8::Value> PlusOne(const v8::Arguments& args);
  double counter_;
};

#endif
```

The implementation is similar to the above in `myobject.cc`:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

MyObject::MyObject() {};
MyObject::~MyObject() {};

Persistent<Function> MyObject::constructor;

void MyObject::Init() {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(New);
  tpl->SetClassName(String::NewSymbol("MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);
  // Prototype
  tpl->PrototypeTemplate()->Set(String::NewSymbol("plusOne"),
      FunctionTemplate::New(PlusOne)->GetFunction());

  constructor = Persistent<Function>::New(tpl->GetFunction());
}

Handle<Value> MyObject::New(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = new MyObject();
  obj->counter_ = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
  obj->Wrap(args.This());

  return args.This();
}

Handle<Value> MyObject::NewInstance(const Arguments& args) {
  HandleScope scope;
```

```
  const unsigned argc = 1;
  Handle<Value> argv[argc] = { args[0] };
  Local<Object> instance = constructor->NewInstance(argc, argv);

  return scope.Close(instance);
}

Handle<Value> MyObject::PlusOne(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.This());
  obj->counter_ += 1;

  return scope.Close(Number::New(obj->counter_));
}
```

Test it with:

```
var addon = require('./build/Release/addon');

var obj = addon.createObject(10);
console.log( obj.plusOne() ); // 11
console.log( obj.plusOne() ); // 12
console.log( obj.plusOne() ); // 13

var obj2 = addon.createObject(20);
console.log( obj2.plusOne() ); // 21
console.log( obj2.plusOne() ); // 22
console.log( obj2.plusOne() ); // 23
```

### Passing wrapped objects around

In addition to wrapping and returning C++ objects, you can pass them around by unwrapping them with Node's `node::ObjectWrap::Unwrap` helper function. In the following `addon.cc` we introduce a function `add()` that can take on two `MyObject` objects:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

Handle<Value> CreateObject(const Arguments& args) {
  HandleScope scope;
  return scope.Close(MyObject::NewInstance(args));
}

Handle<Value> Add(const Arguments& args) {
  HandleScope scope;

  MyObject* obj1 = node::ObjectWrap::Unwrap<MyObject>(
      args[0]->ToObject());
  MyObject* obj2 = node::ObjectWrap::Unwrap<MyObject>(
      args[1]->ToObject());

  double sum = obj1->Val() + obj2->Val();
  return scope.Close(Number::New(sum));
}

void InitAll(Handle<Object> target) {
  MyObject::Init();
```

```
  target->Set(String::NewSymbol("createObject"),
      FunctionTemplate::New(CreateObject)->GetFunction());

  target->Set(String::NewSymbol("add"),
      FunctionTemplate::New(Add)->GetFunction());
}

NODE_MODULE(addon, InitAll)
```

To make things interesting we introduce a public method in `myobject.h` so we can probe private values after unwrapping the object:

```
#define BUILDING_NODE_EXTENSION
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init();
  static v8::Handle<v8::Value> NewInstance(const v8::Arguments& args);
  double Val() const { return val_; }

 private:
  MyObject();
  ~MyObject();

  static v8::Persistent<v8::Function> constructor;
  static v8::Handle<v8::Value> New(const v8::Arguments& args);
  double val_;
};

#endif
```

The implementation of `myobject.cc` is similar as before:

```
#define BUILDING_NODE_EXTENSION
#include <node.h>
#include "myobject.h"

using namespace v8;

MyObject::MyObject() {};
MyObject::~MyObject() {};

Persistent<Function> MyObject::constructor;

void MyObject::Init() {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(New);
  tpl->SetClassName(String::NewSymbol("MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  constructor = Persistent<Function>::New(tpl->GetFunction());
}

Handle<Value> MyObject::New(const Arguments& args) {
  HandleScope scope;

  MyObject* obj = new MyObject();
  obj->val_ = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
```

```
  obj->Wrap(args.This());

  return args.This();
}

Handle<Value> MyObject::NewInstance(const Arguments& args) {
  HandleScope scope;

  const unsigned argc = 1;
  Handle<Value> argv[argc] = { args[0] };
  Local<Object> instance = constructor->NewInstance(argc, argv);

  return scope.Close(instance);
}
```

Test it with:

```
var addon = require('./build/Release/addon');

var obj1 = addon.createObject(10);
var obj2 = addon.createObject(20);
var result = addon.add(obj1, obj2);

console.log(result); // 30
```

## process

The `process` object is a global object and can be accessed from anywhere. It is an instance of EventEmitter.

### Event: 'exit'

Emitted when the process is about to exit. This is a good hook to perform constant time checks of the module's state (like for unit tests). The main event loop will no longer be run after the 'exit' callback finishes, so timers may not be scheduled.

Example of listening for `exit`:

```
process.on('exit', function () {
  process.nextTick(function () {
   console.log('This will not run');
  });
  console.log('About to exit.');
});
```

### Event: 'uncaughtException'

Emitted when an exception bubbles all the way back to the event loop. If a listener is added for this exception, the default action (which is to print a stack trace and exit) will not occur.

Example of listening for `uncaughtException`:

```
process.on('uncaughtException', function (err) {
  console.log('Caught exception: ' + err);
});

setTimeout(function () {
  console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

Note that `uncaughtException` is a very crude mechanism for exception handling. Using try / catch in your program will give you more control over your program's flow. Especially for server programs that are designed to stay running forever, `uncaughtException` can be a useful safety mechanism.

### Signal Events

Emitted when the processes receives a signal. See sigaction(2) for a list of standard POSIX signal names such as SIGINT, SIGUSR1, etc.

Example of listening for `SIGINT`:

```
// Start reading from stdin so we don't exit.
process.stdin.resume();

process.on('SIGINT', function () {
  console.log('Got SIGINT.  Press Control-D to exit.');
});
```

An easy way to send the `SIGINT` signal is with `Control-C` in most terminal programs.

### process.stdout

A `Writable Stream` to `stdout`.

Example: the definition of `console.log`

```
console.log = function (d) {
  process.stdout.write(d + '\n');
};
```

`process.stderr` and `process.stdout` are unlike other streams in Node in that writes to them are usually blocking. They are blocking in the case that they refer to regular files or TTY file descriptors. In the case they refer to pipes, they are non-blocking like other streams.

### process.stderr

A writable stream to stderr.

`process.stderr` and `process.stdout` are unlike other streams in Node in that writes to them are usually blocking. They are blocking in the case that they refer to regular files or TTY file descriptors. In the case they refer to pipes, they are non-blocking like other streams.

### process.stdin

A `Readable Stream` for stdin. The stdin stream is paused by default, so one must call `process.stdin.resume()` to read from it.

Example of opening standard input and listening for both events:

```
process.stdin.resume();
process.stdin.setEncoding('utf8');

process.stdin.on('data', function (chunk) {
  process.stdout.write('data: ' + chunk);
});

process.stdin.on('end', function () {
  process.stdout.write('end');
});
```

### process.argv

An array containing the command line arguments. The first element will be 'node', the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments.

```
// print process.argv
process.argv.forEach(function (val, index, array) {
  console.log(index + ': ' + val);
});
```

This will generate:

```
$ node process-2.js one two=three four
0: node
1: /Users/mjr/work/node/process-2.js
2: one
3: two=three
4: four
```

### process.execPath

This is the absolute pathname of the executable that started the process.

Example:

```
/usr/local/bin/node
```

### process.abort()

This causes node to emit an abort. This will cause node to exit and generate a core file.

### process.chdir(directory)

Changes the current working directory of the process or throws an exception if that fails.

```
console.log('Starting directory: ' + process.cwd());
try {
  process.chdir('/tmp');
  console.log('New directory: ' + process.cwd());
}
catch (err) {
  console.log('chdir: ' + err);
}
```

### process.cwd()

Returns the current working directory of the process.

```
console.log('Current directory: ' + process.cwd());
```

### process.env

An object containing the user environment. See environ(7).

### process.exit([code])

Ends the process with the specified `code`. If omitted, exit uses the 'success' code `0`.

To exit with a 'failure' code:

```
process.exit(1);
```

The shell that executed node should see the exit code as 1.

### process.getgid()

Note: this function is only available on POSIX platforms (i.e. not Windows)

Gets the group identity of the process. (See getgid(2).) This is the numerical group id, not the group name.

```
if (process.getgid) {
  console.log('Current gid: ' + process.getgid());
}
```

### process.setgid(id)

Note: this function is only available on POSIX platforms (i.e. not Windows)

Sets the group identity of the process. (See setgid(2).) This accepts either a numerical ID or a groupname string. If a groupname is specified, this method blocks while resolving it to a numerical ID.

```
if (process.getgid && process.setgid) {
  console.log('Current gid: ' + process.getgid());
  try {
    process.setgid(501);
    console.log('New gid: ' + process.getgid());
  }
  catch (err) {
    console.log('Failed to set gid: ' + err);
  }
}
```

### process.getuid()

Note: this function is only available on POSIX platforms (i.e. not Windows)

Gets the user identity of the process. (See getuid(2).) This is the numerical userid, not the username.

```
if (process.getuid) {
  console.log('Current uid: ' + process.getuid());
}
```

### process.setuid(id)

Note: this function is only available on POSIX platforms (i.e. not Windows)

Sets the user identity of the process. (See setuid(2).) This accepts either a numerical ID or a username string. If a username is specified, this method blocks while resolving it to a numerical ID.

```
if (process.getuid && process.setuid) {
  console.log('Current uid: ' + process.getuid());
  try {
    process.setuid(501);
    console.log('New uid: ' + process.getuid());
  }
  catch (err) {
    console.log('Failed to set uid: ' + err);
  }
}
```

### process.version

A compiled-in property that exposes `NODE_VERSION`.

```
console.log('Version: ' + process.version);
```

### process.versions

A property exposing version strings of node and its dependencies.

```
console.log(process.versions);
```

Will output:

```
{ node: '0.4.12',
  v8: '3.1.8.26',
  ares: '1.7.4',
  ev: '4.4',
  openssl: '1.0.0e-fips' }
```

### process.config

An Object containing the JavaScript representation of the configure options that were used to compile the current node executable. This is the same as the "config.gypi" file that was produced when running the `./configure` script.

An example of the possible output looks like:

```
{ target_defaults:
   { cflags: [],
     default_configuration: 'Release',
     defines: [],
     include_dirs: [],
     libraries: [] },
  variables:
   { host_arch: 'x64',
     node_install_npm: 'true',
     node_install_waf: 'true',
     node_prefix: '',
     node_shared_v8: 'false',
     node_shared_zlib: 'false',
     node_use_dtrace: 'false',
     node_use_openssl: 'true',
     node_shared_openssl: 'false',
     strict_aliasing: 'true',
     target_arch: 'x64',
     v8_use_snapshot: 'true' } }
```

### process.kill(pid, [signal])

Send a signal to a process. `pid` is the process id and `signal` is the string describing the signal to send. Signal names are strings like 'SIGINT' or 'SIGUSR1'. If omitted, the signal will be 'SIGTERM'. See kill(2) for more information.

Note that just because the name of this function is `process.kill`, it is really just a signal sender, like the `kill` system call. The signal sent may do something other than kill the target process.

Example of sending a signal to yourself:

```
process.on('SIGHUP', function () {
  console.log('Got SIGHUP signal.');
});

setTimeout(function () {
  console.log('Exiting.');
  process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

### process.pid

The PID of the process.

```
console.log('This process is pid ' + process.pid);
```

### process.title

Getter/setter to set what is displayed in 'ps'.

### process.arch

What processor architecture you're running on: `'arm'`, `'ia32'`, or `'x64'`.

```
console.log('This processor architecture is ' + process.arch);
```

### process.platform

What platform you're running on: `'darwin'`, `'freebsd'`, `'linux'`, `'solaris'` or `'win32'`

```
console.log('This platform is ' + process.platform);
```

### process.memoryUsage()

Returns an object describing the memory usage of the Node process measured in bytes.

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

console.log(util.inspect(process.memoryUsage()));
```

This will generate:

```
{ rss: 4935680,
  heapTotal: 1826816,
  heapUsed: 650472 }
```

`heapTotal` and `heapUsed` refer to V8's memory usage.

### process.nextTick(callback)

On the next loop around the event loop call this callback. This is *not* a simple alias to `setTimeout(fn, 0)`, it's much more efficient.

```
process.nextTick(function () {
  console.log('nextTick callback');
});
```

### process.umask([mask])

Sets or reads the process's file mode creation mask. Child processes inherit the mask from the parent process. Returns the old mask if `mask` argument is given, otherwise returns the current mask.

```
var oldmask, newmask = 0644;

oldmask = process.umask(newmask);
console.log('Changed umask from: ' + oldmask.toString(8) +
            ' to ' + newmask.toString(8));
```

### process.uptime()

Number of seconds Node has been running.

### process.hrtime()

Returns the current high-resolution real time in a `[seconds, nanoseconds]` tuple Array. It is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals.

You may pass in the result of a previous call to `process.hrtime()` to get a diff reading, useful for benchmarks and measuring intervals:

```
var t = process.hrtime();
// [ 1800216, 927643717 ]

setTimeout(function () {
  t = process.hrtime(t);
  // [ 1, 6962306 ]

  console.log('benchmark took %d seconds and %d nanoseconds', t[0], t[1]);
  // benchmark took 1 seconds and 6962306 nanoseconds
}, 1000);
```

# util

```
Stability: 5 - Locked
```

These functions are in the module `'util'`. Use `require('util')` to access them.

## util.format(format, [...])

Returns a formatted string using the first argument as a `printf`-like format.

The first argument is a string that contains zero or more *placeholders*. Each placeholder is replaced with the converted value from its corresponding argument. Supported placeholders are:

- `%s` - String.
- `%d` - Number (both integer and float).
- `%j` - JSON.
- `%%` - single percent sign (`'%'`). This does not consume an argument.

If the placeholder does not have a corresponding argument, the placeholder is not replaced.

```
util.format('%s:%s', 'foo'); // 'foo:%s'
```

If there are more arguments than placeholders, the extra arguments are converted to strings with `util.inspect()` and these strings are concatenated, delimited by a space.

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

If the first argument is not a format string then `util.format()` returns a string that is the concatenation of all its arguments separated by spaces. Each argument is converted to a string with `util.inspect()`.

```
util.format(1, 2, 3); // '1 2 3'
```

## util.debug(string)

A synchronous output function. Will block the process and output `string` immediately to `stderr`.

```
require('util').debug('message on stderr');
```

## util.error([...])

Same as `util.debug()` except this will output all arguments immediately to `stderr`.

## util.puts([...])

A synchronous output function. Will block the process and output all arguments to `stdout` with newlines after each argument.

## util.print([...])

A synchronous output function. Will block the process, cast each argument to a string then output to `stdout`. Does not place newlines after each argument.

## util.log(string)

Output with timestamp on `stdout`.

```
require('util').log('Timestamped message.');
```

### util.inspect(object, [showHidden], [depth], [colors])

Return a string representation of `object`, which is useful for debugging.

If `showHidden` is `true`, then the object's non-enumerable properties will be shown too. Defaults to `false`.

If `depth` is provided, it tells `inspect` how many times to recurse while formatting the object. This is useful for inspecting large complicated objects.

The default is to only recurse twice. To make it recurse indefinitely, pass in `null` for `depth`.

If `colors` is `true`, the output will be styled with ANSI color codes. Defaults to `false`.

Example of inspecting all properties of the `util` object:

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

console.log(util.inspect(util, true, null));
```

### util.isArray(object)

Returns `true` if the given "object" is an `Array`. `false` otherwise.

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

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

### util.isRegExp(object)

Returns `true` if the given "object" is a `RegExp`. `false` otherwise.

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

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

### util.isDate(object)

Returns `true` if the given "object" is a `Date`. `false` otherwise.

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

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

### util.isError(object)

Returns `true` if the given "object" is an `Error`. `false` otherwise.

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

util.isError(new Error())
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

### util.pump(readableStream, writableStream, [callback])

Experimental

Read the data from `readableStream` and send it to the `writableStream`. When `writableStream.write(data)` returns `false` `readableStream` will be paused until the `drain` event occurs on the `writableStream`. `callback` gets an error as its only argument and is called when `writableStream` is closed or when an error occurs.

### util.inherits(constructor, superConstructor)

Inherit the prototype methods from one constructor into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.

As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.

```
var util = require("util");
var events = require("events");

function MyStream() {
    events.EventEmitter.call(this);
}

util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
    this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); // true
console.log(MyStream.super_ === events.EventEmitter); // true

stream.on("data", function(data) {
    console.log('Received data: "' + data + '"');
})
stream.write("It works!"); // Received data: "It works!"
```

# Events

```
Stability: 4 - API Frozen
```

Many objects in Node emit events: a `net.Server` emits an event each time a peer connects to it, a `fs.readStream` emits an event when the file is opened. All objects which emit events are instances of `events.EventEmitter`. You can access this module by doing: `require("events");`

Typically, event names are represented by a camel-cased string, however, there aren't any strict restrictions on that, as any string will be accepted.

Functions can then be attached to objects, to be executed when an event is emitted. These functions are called *listeners*.

## Class: events.EventEmitter

To access the EventEmitter class, `require('events').EventEmitter`.

When an `EventEmitter` instance experiences an error, the typical action is to emit an `'error'` event. Error events are treated as a special case in node. If there is no listener for it, then the default action is to print a stack trace and exit the program.

All EventEmitters emit the event `'newListener'` when new listeners are added.

### emitter.addListener(event, listener)

### emitter.on(event, listener)

Adds a listener to the end of the listeners array for the specified event.

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
```

### emitter.once(event, listener)

Adds a **one time** listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.

```
server.once('connection', function (stream) {
  console.log('Ah, we have our first user!');
});
```

### emitter.removeListener(event, listener)

Remove a listener from the listener array for the specified event. **Caution**: changes array indices in the listener array behind the listener.

```
var callback = function(stream) {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

### emitter.removeAllListeners([event])

Removes all listeners, or those of the specified event.

Note that this will **invalidate** any arrays that have previously been returned by `emitter.listeners(event)`.

### emitter.setMaxListeners(n)

By default EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.

### emitter.listeners(event)

Returns an array of listeners for the specified event.

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection'))); // [ [Function] ]
```

This array **may** be a mutable reference to the same underlying list of listeners that is used by the event subsystem. However, certain actions (specifically, removeAllListeners) will invalidate this reference.

If you would like to get a copy of the listeners at a specific point in time that is guaranteed not to change, make a copy, for example by doing `emitter.listeners(event).slice(0)`.

In a future release of node, this behavior **may** change to always return a copy, for consistency. In your programs, please do not rely on being able to modify the EventEmitter listeners using array methods. Always use the 'on' method to add new listeners.

### emitter.emit(event, [arg1], [arg2], [...])

Execute each of the listeners in order with the supplied arguments.

### Event: 'newListener'

- `event` {String} The event name
- `listener` {Function} The event handler function

This event is emitted any time someone adds a new listener.

# Domain

```
Stability: 1 - Experimental
```

Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an `error` event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the `process.on('uncaughtException')` handler, or causing the program to exit with an error code.

This feature is new in Node version 0.8. It is a first pass, and is expected to change significantly in future versions. Please use it and provide feedback.

Due to their experimental nature, the Domains features are disabled unless the `domain` module is loaded at least once. No domains are created or registered by default. This is by design, to prevent adverse effects on current programs. It is expected to be enabled by default in future Node.js versions.

## Additions to Error objects

Any time an Error object is routed through a domain, a few extra fields are added to it.

- `error.domain` The domain that first handled the error.
- `error.domain_emitter` The event emitter that emitted an 'error' event with the error object.
- `error.domain_bound` The callback function which was bound to the domain, and passed an error as its first argument.
- `error.domain_thrown` A boolean indicating whether the error was thrown, emitted, or passed to a bound callback function.

## Implicit Binding

If domains are in use, then all new EventEmitter objects (including Stream objects, requests, responses, etc.) will be implicitly bound to the active domain at the time of their creation.

Additionally, callbacks passed to lowlevel event loop requests (such as to fs.open, or other callback-taking methods) will automatically be bound to the active domain. If they throw, then the domain will catch the error.

In order to prevent excessive memory usage, Domain objects themselves are not implicitly added as children of the active domain. If they were, then it would be too easy to prevent request and response objects from being properly garbage collected.

If you *want* to nest Domain objects as children of a parent Domain, then you must explicitly add them, and then dispose of them later.

Implicit binding routes thrown errors and `'error'` events to the Domain's `error` event, but does not register the EventEmitter on the Domain, so `domain.dispose()` will not shut down the EventEmitter. Implicit binding only takes care of thrown errors and `'error'` events.

## Explicit Binding

Sometimes, the domain in use is not the one that ought to be used for a specific event emitter. Or, the event emitter could have been created in the context of one domain, but ought to instead be bound to some other domain.

For example, there could be one domain in use for an HTTP server, but perhaps we would like to have a separate domain to use for each request.

That is possible via explicit binding.

For example:

```
// create a top-level domain for the server
var serverDomain = domain.create();

serverDomain.run(function() {
  // server is created in the scope of serverDomain
  http.createServer(function(req, res) {
    // req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    var reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', function(er) {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
        res.on('close', function() {
          // forcibly shut down any other things added to this domain
          reqd.dispose();
        });
      } catch (er) {
        console.error('Error sending 500', er, req.url);
        // tried our best.  clean up anything remaining.
        reqd.dispose();
      }
    });
  }).listen(1337);
});
```

## domain.create()

- return: {Domain}

Returns a new Domain object.

## Class: Domain

The Domain class encapsulates the functionality of routing errors and uncaught exceptions to the active Domain object.

Domain is a child class of EventEmitter. To handle the errors that it catches, listen to its `error` event.

### domain.run(fn)

- `fn` {Function}

Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowlevel requests that are created in that context.

This is the most basic way to use a domain.

Example:

```
var d = domain.create();
d.on('error', function(er) {
  console.error('Caught error!', er);
});
d.run(function() {
  process.nextTick(function() {
    setTimeout(function() { // simulating some various async stuff
      fs.open('non-existent file', 'r', function(er, fd) {
```

```
        if (er) throw er;
        // proceed...
      });
    }, 100);
  });
});
```

In this example, the `d.on('error')` handler will be triggered, rather than crashing the program.

### domain.members

- {Array}

An array of timers and event emitters that have been explicitly added to the domain.

### domain.add(emitter)

- `emitter` {EventEmitter | Timer} emitter or timer to be added to the domain

Explicitly adds an emitter to the domain. If any event handlers called by the emitter throw an error, or if the emitter emits an `error` event, it will be routed to the domain's `error` event, just like with implicit binding.

This also works with timers that are returned from `setInterval` and `setTimeout`. If their callback function throws, it will be caught by the domain 'error' handler.

If the Timer or EventEmitter was already bound to a domain, it is removed from that one, and bound to this one instead.

### domain.remove(emitter)

- `emitter` {EventEmitter | Timer} emitter or timer to be removed from the domain

The opposite of `domain.add(emitter)`. Removes domain handling from the specified emitter.

### domain.bind(cb)

- `cb` {Function} The callback function
- return: {Function} The bound function

The returned function will be a wrapper around the supplied callback function. When the returned function is called, any errors that are thrown will be routed to the domain's `error` event.

```
var d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, d.bind(function(er, data) {
    // if this throws, it will also be passed to the domain
    return cb(er, JSON.parse(data));
  }));
}

d.on('error', function(er) {
  // an error occurred somewhere.
  // if we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

**Example**

### domain.intercept(cb)

- `cb` {Function} The callback function
- return: {Function} The intercepted function

This method is almost identical to `domain.bind(cb)`. However, in addition to catching thrown errors, it will also intercept `Error` objects sent as the first argument to the function.

In this way, the common `if (er) return cb(er);` pattern can be replaced with a single error handler in a single place.

```
var d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, d.intercept(function(data) {
    // note, the first argument is never passed to the
    // callback since it is assumed to be the 'Error' argument
    // and thus intercepted by the domain.

    // if this throws, it will also be passed to the domain
    // so the error-handling logic can be moved to the 'error'
    // event on the domain instead of being repeated throughout
    // the program.
    return cb(null, JSON.parse(data));
  }));
}

d.on('error', function(er) {
  // an error occurred somewhere.
  // if we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

**Example**

### domain.dispose()

The dispose method destroys a domain, and makes a best effort attempt to clean up any and all IO that is associated with the domain. Streams are aborted, ended, closed, and/or destroyed. Timers are cleared. Explicitly bound callbacks are no longer called. Any error events that are raised as a result of this are ignored.

The intention of calling `dispose` is generally to prevent cascading errors when a critical part of the Domain context is found to be in an error state.

Once the domain is disposed the `dispose` event will emit.

Note that IO might still be performed. However, to the highest degree possible, once a domain is disposed, further errors from the emitters in that set will be ignored. So, even if some remaining actions are still in flight, Node.js will not communicate further about them.

# Buffer

```
Stability: 3 - Stable
```

Pure JavaScript is Unicode friendly but not nice to binary data. When dealing with TCP streams or the file system, it's necessary to handle octet streams. Node has several strategies for manipulating, creating, and consuming octet streams.

Raw data is stored in instances of the `Buffer` class. A `Buffer` is similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. A `Buffer` cannot be resized.

The `Buffer` class is a global, making it very rare that one would need to ever `require('buffer')`.

Converting between Buffers and JavaScript string objects requires an explicit encoding method. Here are the different string encodings.

- `'ascii'` - for 7 bit ASCII data only. This encoding method is very fast, and will strip the high bit if set. Note that this encoding converts a null character (`'\0'` or `'\u0000'`) into `0x20` (character code of a space). If you want to convert a null character into `0x00`, you should use `'utf8'`.
- `'utf8'` - Multibyte encoded Unicode characters. Many web pages and other document formats use UTF-8.
- `'utf16le'` - 2 or 4 bytes, little endian encoded Unicode characters. Surrogate pairs (U+10000 to U+10FFFF) are supported.
- `'ucs2'` - Alias of `'utf16le'`.
- `'base64'` - Base64 string encoding.
- `'binary'` - A way of encoding raw binary data into strings by using only the first 8 bits of each character. This encoding method is deprecated and should be avoided in favor of `Buffer` objects where possible. This encoding will be removed in future versions of Node.
- `'hex'` - Encode each byte as two hexadecimal characters.

## Class: Buffer

The Buffer class is a global type for dealing with binary data directly. It can be constructed in a variety of ways.

### new Buffer(size)

- `size` Number

Allocates a new buffer of `size` octets.

### new Buffer(array)

- `array` Array

Allocates a new buffer using an `array` of octets.

### new Buffer(str, [encoding])

- `str` String - string to encode.
- `encoding` String - encoding to use, Optional.

Allocates a new buffer containing the given `str`. `encoding` defaults to `'utf8'`.

### buf.write(string, [offset], [length], [encoding])

- `string` String - data to be written to buffer
- `offset` Number, Optional, Default: 0
- `length` Number, Optional, Default: `buffer.length - offset`
- `encoding` String, Optional, Default: 'utf8'

Writes `string` to the buffer at `offset` using the given encoding. `offset` defaults to `0`, `encoding` defaults to `'utf8'`. `length` is the number of bytes to write. Returns number of octets written. If `buffer` did not contain enough space to fit the entire string, it will write a partial amount of the string. `length` defaults to `buffer.length - offset`. The method will not write partial characters.

```
buf = new Buffer(256);
len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(len + " bytes: " + buf.toString('utf8', 0, len));
```

The number of characters written (which may be different than the number of bytes written) is set in `Buffer._charsWritten` and will be overwritten the next time `buf.write()` is called.

### buf.toString([encoding], [start], [end])

- `encoding` String, Optional, Default: 'utf8'
- `start` Number, Optional, Default: 0
- `end` Number, Optional, Default: `buffer.length`

Decodes and returns a string from buffer data encoded with `encoding` (defaults to `'utf8'`) beginning at `start` (defaults to `0`) and ending at `end` (defaults to `buffer.length`).

See `buffer.write()` example, above.

### buf[index]

Get and set the octet at `index`. The values refer to individual bytes, so the legal range is between `0x00` and `0xFF` hex or `0` and `255`.

Example: copy an ASCII string into a buffer, one byte at a time:

```
str = "node.js";
buf = new Buffer(str.length);

for (var i = 0; i < str.length ; i++) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf);

// node.js
```

### Class Method: Buffer.isBuffer(obj)

- `obj` Object
- Return: Boolean

Tests if `obj` is a `Buffer`.

### Class Method: Buffer.byteLength(string, [encoding])

- `string` String
- `encoding` String, Optional, Default: 'utf8'

- Return: Number

Gives the actual byte length of a string. `encoding` defaults to `'utf8'`. This is not the same as `String.prototype.length` since that returns the number of *characters* in a string.

Example:

```
str = '\u00bd + \u00bc = \u00be';

console.log(str + ": " + str.length + " characters, " +
  Buffer.byteLength(str, 'utf8') + " bytes");

// Ã¯ + Ã = Ã¿: 9 characters, 12 bytes
```

### Class Method: Buffer.concat(list, [totalLength])

- `list` {Array} List of Buffer objects to concat
- `totalLength` {Number} Total length of the buffers when concatenated

Returns a buffer which is the result of concatenating all the buffers in the list together.

If the list has no items, or if the totalLength is 0, then it returns a zero-length buffer.

If the list has exactly one item, then the first item of the list is returned.

If the list has more than one item, then a new Buffer is created.

If totalLength is not provided, it is read from the buffers in the list. However, this adds an additional loop to the function, so it is faster to provide the length explicitly.

### buf.length

- Number

The size of the buffer in bytes. Note that this is not necessarily the size of the contents. `length` refers to the amount of memory allocated for the buffer object. It does not change when the contents of the buffer are changed.

```
buf = new Buffer(1234);

console.log(buf.length);
buf.write("some string", 0, "ascii");
console.log(buf.length);

// 1234
// 1234
```

### buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd])

- `targetBuffer` Buffer object - Buffer to copy into
- `targetStart` Number, Optional, Default: 0
- `sourceStart` Number, Optional, Default: 0
- `sourceEnd` Number, Optional, Default: `buffer.length`

Does copy between buffers. The source and target regions can be overlapped. `targetStart` and `sourceStart` default to `0`. `sourceEnd` defaults to `buffer.length`.

Example: build two Buffers, then copy `buf1` from byte 16 through byte 19 into `buf2`, starting at the 8th byte in `buf2`.

```
buf1 = new Buffer(26);
buf2 = new Buffer(26);
```

```
for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
  buf2[i] = 33; // ASCII !
}

buf1.copy(buf2, 8, 16, 20);
console.log(buf2.toString('ascii', 0, 25));

// !!!!!!!!qrst!!!!!!!!!!!!
```

### buf.slice([start], [end])

- `start` Number, Optional, Default: 0
- `end` Number, Optional, Default: `buffer.length`

Returns a new buffer which references the same memory as the old, but offset and cropped by the `start` (defaults to `0`) and `end` (defaults to `buffer.length`) indexes.

**Modifying the new buffer slice will modify memory in the original buffer!**

Example: build a Buffer with the ASCII alphabet, take a slice, then modify one byte from the original Buffer.

```
var buf1 = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
}

var buf2 = buf1.slice(0, 3);
console.log(buf2.toString('ascii', 0, buf2.length));
buf1[0] = 33;
console.log(buf2.toString('ascii', 0, buf2.length));

// abc
// !bc
```

### buf.readUInt8(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads an unsigned 8 bit integer from the buffer at the specified offset.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

for (ii = 0; ii < buf.length; ii++) {
  console.log(buf.readUInt8(ii));
}

// 0x3
// 0x4
```

```
// 0x23
// 0x42
```

### buf.readUInt16LE(offset, [noAssert])

### buf.readUInt16BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads an unsigned 16 bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

console.log(buf.readUInt16BE(0));
console.log(buf.readUInt16LE(0));
console.log(buf.readUInt16BE(1));
console.log(buf.readUInt16LE(1));
console.log(buf.readUInt16BE(2));
console.log(buf.readUInt16LE(2));

// 0x0304
// 0x0403
// 0x0423
// 0x2304
// 0x2342
// 0x4223
```

### buf.readUInt32LE(offset, [noAssert])

### buf.readUInt32BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads an unsigned 32 bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;
```

```
console.log(buf.readUInt32BE(0));
console.log(buf.readUInt32LE(0));

// 0x03042342
// 0x42230403
```

### buf.readInt8(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a signed 8 bit integer from the buffer at the specified offset.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Works as `buffer.readUInt8`, except buffer contents are treated as two's complement signed values.

### buf.readInt16LE(offset, [noAssert])

### buf.readInt16BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a signed 16 bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Works as `buffer.readUInt16*`, except buffer contents are treated as two's complement signed values.

### buf.readInt32LE(offset, [noAssert])

### buf.readInt32BE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a signed 32 bit integer from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Works as `buffer.readUInt32*`, except buffer contents are treated as two's complement signed values.

### buf.readFloatLE(offset, [noAssert])

### buf.readFloatBE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a 32 bit float from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(4);

buf[0] = 0x00;
buf[1] = 0x00;
buf[2] = 0x80;
buf[3] = 0x3f;

console.log(buf.readFloatLE(0));

// 0x01
```

### buf.readDoubleLE(offset, [noAssert])

### buf.readDoubleBE(offset, [noAssert])

- `offset` Number
- `noAssert` Boolean, Optional, Default: false
- Return: Number

Reads a 64 bit double from the buffer at the specified offset with specified endian format.

Set `noAssert` to true to skip validation of `offset`. This means that `offset` may be beyond the end of the buffer. Defaults to `false`.

Example:

```
var buf = new Buffer(8);

buf[0] = 0x55;
buf[1] = 0x55;
buf[2] = 0x55;
buf[3] = 0x55;
buf[4] = 0x55;
buf[5] = 0x55;
buf[6] = 0xd5;
buf[7] = 0x3f;

console.log(buf.readDoubleLE(0));

// 0.3333333333333333
```

### buf.writeUInt8(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset. Note, `value` must be a valid unsigned 8 bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);

// <Buffer 03 04 23 42>
```

### buf.writeUInt16LE(value, offset, [noAssert])

### buf.writeUInt16BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note, `value` must be a valid unsigned 16 bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);

// <Buffer de ad be ef>
// <Buffer ad de ef be>
```

### buf.writeUInt32LE(value, offset, [noAssert])

### buf.writeUInt32BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note, `value` must be a valid unsigned 32 bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeUInt32BE(0xfeedface, 0);
```

```
console.log(buf);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);

// <Buffer fe ed fa ce>
// <Buffer ce fa ed fe>
```

### buf.writeInt8(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset. Note, `value` must be a valid signed 8 bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Works as `buffer.writeUInt8`, except value is written out as a two's complement signed integer into `buffer`.

### buf.writeInt16LE(value, offset, [noAssert])

### buf.writeInt16BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note, `value` must be a valid signed 16 bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Works as `buffer.writeUInt16*`, except value is written out as a two's complement signed integer into `buffer`.

### buf.writeInt32LE(value, offset, [noAssert])

### buf.writeInt32BE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note, `value` must be a valid signed 32 bit integer.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Works as `buffer.writeUInt32*`, except value is written out as a two's complement signed integer into `buffer`.

### buf.writeFloatLE(value, offset, [noAssert])

### buf.writeFloatBE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note, `value` must be a valid 32 bit float.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(4);
buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);

// <Buffer 4f 4a fe bb>
// <Buffer bb fe 4a 4f>
```

### buf.writeDoubleLE(value, offset, [noAssert])

### buf.writeDoubleBE(value, offset, [noAssert])

- `value` Number
- `offset` Number
- `noAssert` Boolean, Optional, Default: false

Writes `value` to the buffer at the specified offset with specified endian format. Note, `value` must be a valid 64 bit double.

Set `noAssert` to true to skip validation of `value` and `offset`. This means that `value` may be too large for the specific function and `offset` may be beyond the end of the buffer leading to the values being silently dropped. This should not be used unless you are certain of correctness. Defaults to `false`.

Example:

```
var buf = new Buffer(8);
buf.writeDoubleBE(0xdeadbeefcafebabe, 0);

console.log(buf);

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);

// <Buffer 43 eb d5 b7 dd f9 5f d7>
// <Buffer d7 5f f9 dd b7 d5 eb 43>
```

### buf.fill(value, [offset], [end])

- `value`

- `offset` Number, Optional
- `end` Number, Optional

Fills the buffer with the specified value. If the `offset` (defaults to `0`) and `end` (defaults to `buffer.length`) are not given it will fill the entire buffer.

```
var b = new Buffer(50);
b.fill("h");
```

## buffer.INSPECT_MAX_BYTES

- Number, Default: 50

How many bytes will be returned when `buffer.inspect()` is called. This can be overridden by user modules.

Note that this is a property on the buffer module returned by `require('buffer')`, not on the Buffer global, or a buffer instance.

## Class: SlowBuffer

This class is primarily for internal use. JavaScript programs should use Buffer instead of using SlowBuffer.

In order to avoid the overhead of allocating many C++ Buffer objects for small blocks of memory in the lifetime of a server, Node allocates memory in 8Kb (8192 byte) chunks. If a buffer is smaller than this size, then it will be backed by a parent SlowBuffer object. If it is larger than this, then Node will allocate a SlowBuffer slab for it directly.

# Stream

```
Stability: 2 - Unstable
```

A stream is an abstract interface implemented by various objects in Node. For example a request to an HTTP server is a stream, as is stdout. Streams are readable, writable, or both. All streams are instances of EventEmitter

You can load up the Stream base class by doing `require('stream')`.

## Readable Stream

A `Readable Stream` has the following methods, members, and events.

### Event: 'data'

```
function (data) { }
```

The `'data'` event emits either a `Buffer` (by default) or a string if `setEncoding()` was used.

Note that the **data will be lost** if there is no listener when a `Readable Stream` emits a `'data'` event.

### Event: 'end'

```
function () { }
```

Emitted when the stream has received an EOF (FIN in TCP terminology). Indicates that no more `'data'` events will happen. If the stream is also writable, it may be possible to continue writing.

### Event: 'error'

```
function (exception) { }
```

Emitted if there was an error receiving data.

### Event: 'close'

```
function () { }
```

Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.

### stream.readable

A boolean that is `true` by default, but turns `false` after an `'error'` occurred, the stream came to an `'end'`, or `destroy()` was called.

### stream.setEncoding([encoding])

Makes the `'data'` event emit a string instead of a `Buffer`. `encoding` can be `'utf8'`, `'utf16le'` (`'ucs2'`), `'ascii'`, or `'hex'`. Defaults to `'utf8'`.

### stream.pause()

Issues an advisory signal to the underlying communication layer, requesting that no further data be sent until `resume()` is called.

Note that, due to the advisory nature, certain streams will not be paused immediately, and so `'data'` events may be emitted for some indeterminate period of time even after `pause()` is called. You may wish to buffer such `'data'` events.

#### stream.resume()

Resumes the incoming `'data'` events after a `pause()`.

#### stream.destroy()

Closes the underlying file descriptor. Stream will not emit any more events.

#### stream.pipe(destination, [options])

This is a `Stream.prototype` method available on all `Stream`s.

Connects this read stream to `destination` WriteStream. Incoming data on this stream gets written to `destination`. The destination and source streams are kept in sync by pausing and resuming as necessary.

This function returns the `destination` stream.

Emulating the Unix `cat` command:

```
process.stdin.resume();
process.stdin.pipe(process.stdout);
```

By default `end()` is called on the destination when the source stream emits `end`, so that `destination` is no longer writable. Pass `{ end: false }` as `options` to keep the destination stream open.

This keeps `process.stdout` open so that "Goodbye" can be written at the end.

```
process.stdin.resume();

process.stdin.pipe(process.stdout, { end: false });

process.stdin.on("end", function() {
  process.stdout.write("Goodbye\n");
});
```

### Writable Stream

A `Writable Stream` has the following methods, members, and events.

#### Event: 'drain'

`function () { }`

After a `write()` method returned `false`, this event is emitted to indicate that it is safe to write again.

#### Event: 'error'

`function (exception) { }`

Emitted on error with the exception `exception`.

#### Event: 'close'

`function () { }`

Emitted when the underlying file descriptor has been closed.

#### Event: 'pipe'

`function (src) { }`

Emitted when the stream is passed to a readable stream's pipe method.

### stream.writable

A boolean that is `true` by default, but turns `false` after an `'error'` occurred or `end()` / `destroy()` was called.

### stream.write(string, [encoding], [fd])

Writes `string` with the given `encoding` to the stream. Returns `true` if the string has been flushed to the kernel buffer. Returns `false` to indicate that the kernel buffer is full, and the data will be sent out in the future. The `'drain'` event will indicate when the kernel buffer is empty again. The `encoding` defaults to `'utf8'`.

If the optional `fd` parameter is specified, it is interpreted as an integral file descriptor to be sent over the stream. This is only supported for UNIX streams, and is silently ignored otherwise. When writing a file descriptor in this manner, closing the descriptor before the stream drains risks sending an invalid (closed) FD.

### stream.write(buffer)

Same as the above except with a raw buffer.

### stream.end()

Terminates the stream with EOF or FIN. This call will allow queued write data to be sent before closing the stream.

### stream.end(string, encoding)

Sends `string` with the given `encoding` and terminates the stream with EOF or FIN. This is useful to reduce the number of packets sent.

### stream.end(buffer)

Same as above but with a `buffer`.

### stream.destroy()

Closes the underlying file descriptor. Stream will not emit any more events. Any queued write data will not be sent.

### stream.destroySoon()

After the write queue is drained, close the file descriptor. `destroySoon()` can still destroy straight away, as long as there is no data left in the queue for writes.

# Crypto

```
Stability: 3 - Stable
```

Use `require('crypto')` to access this module.

The crypto module requires OpenSSL to be available on the underlying platform. It offers a way of encapsulating secure credentials to be used as part of a secure HTTPS net or http connection.

It also offers a set of wrappers for OpenSSL's hash, hmac, cipher, decipher, sign and verify methods.

## crypto.createCredentials(details)

Creates a credentials object, with the optional details being a dictionary with keys:

- `pfx` : A string or buffer holding the PFX or PKCS12 encoded private key, certificate and CA certificates
- `key` : A string holding the PEM encoded private key
- `passphrase` : A string of passphrase for the private key or pfx
- `cert` : A string holding the PEM encoded certificate
- `ca` : Either a string or list of strings of PEM encoded CA certificates to trust.
- `crl` : Either a string or list of strings of PEM encoded CRLs (Certificate Revocation List)
- `ciphers`: A string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format.

If no 'ca' details are given, then node.js will use the default publicly trusted list of CAs as given in http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt.

## crypto.createHash(algorithm)

Creates and returns a hash object, a cryptographic hash with the given algorithm which can be used to generate hash digests.

`algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha1'`, `'md5'`, `'sha256'`, `'sha512'`, etc. On recent releases, `openssl list-message-digest-algorithms` will display the available digest algorithms.

Example: this program that takes the sha1 sum of a file

```
var filename = process.argv[2];
var crypto = require('crypto');
var fs = require('fs');

var shasum = crypto.createHash('sha1');

var s = fs.ReadStream(filename);
s.on('data', function(d) {
  shasum.update(d);
});

s.on('end', function() {
  var d = shasum.digest('hex');
  console.log(d + '  ' + filename);
});
```

## Class: Hash

The class for creating hash digests of data.

Returned by `crypto.createHash`.

#### hash.update(data, [input_encoding])

Updates the hash content with the given `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. Defaults to `'binary'`. This can be called many times with new data as it is streamed.

#### hash.digest([encoding])

Calculates the digest of all of the passed data to be hashed. The `encoding` can be `'hex'`, `'binary'` or `'base64'`. Defaults to `'binary'`.

Note: `hash` object can not be used after `digest()` method been called.

### crypto.createHmac(algorithm, key)

Creates and returns a hmac object, a cryptographic hmac with the given algorithm and key.

`algorithm` is dependent on the available algorithms supported by OpenSSL - see createHash above. `key` is the hmac key to be used.

### Class: Hmac

Class for creating cryptographic hmac content.

Returned by `crypto.createHmac`.

#### hmac.update(data)

Update the hmac content with the given `data`. This can be called many times with new data as it is streamed.

#### hmac.digest([encoding])

Calculates the digest of all of the passed data to the hmac. The `encoding` can be `'hex'`, `'binary'` or `'base64'`. Defaults to `'binary'`.

Note: `hmac` object can not be used after `digest()` method been called.

### crypto.createCipher(algorithm, password)

Creates and returns a cipher object, with the given algorithm and password.

`algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent releases, `openssl list-cipher-algorithms` will display the available cipher algorithms. `password` is used to derive key and IV, which must be a `'binary'` encoded string or a buffer.

### crypto.createCipheriv(algorithm, key, iv)

Creates and returns a cipher object, with the given algorithm, key and iv.

`algorithm` is the same as the argument to `createCipher()`. `key` is the raw key used by the algorithm. `iv` is an initialization vector.

`key` and `iv` must be `'binary'` encoded strings or buffers.

### Class: Cipher

Class for encrypting data.

Returned by `crypto.createCipher` and `crypto.createCipheriv`.

#### cipher.update(data, [input_encoding], [output_encoding])

Updates the cipher with `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. Defaults to `'binary'`.

The `output_encoding` specifies the output format of the enciphered data, and can be `'binary'`, `'base64'` or `'hex'`. Defaults to `'binary'`.

Returns the enciphered contents, and can be called many times with new data as it is streamed.

#### cipher.final([output_encoding])

Returns any remaining enciphered contents, with `output_encoding` being one of: `'binary'`, `'base64'` or `'hex'`. Defaults to `'binary'`.

Note: `cipher` object can not be used after `final()` method been called.

#### cipher.setAutoPadding(auto_padding=true)

You can disable automatic padding of the input data to block size. If `auto_padding` is false, the length of the entire input data must be a multiple of the cipher's block size or `final` will fail. Useful for non-standard padding, e.g. using `0x0` instead of PKCS padding. You must call this before `cipher.final`.

### crypto.createDecipher(algorithm, password)

Creates and returns a decipher object, with the given algorithm and key. This is the mirror of the createCipher() above.

### crypto.createDecipheriv(algorithm, key, iv)

Creates and returns a decipher object, with the given algorithm, key and iv. This is the mirror of the createCipheriv() above.

### Class: Decipher

Class for decrypting data.

Returned by `crypto.createDecipher` and `crypto.createDecipheriv`.

#### decipher.update(data, [input_encoding], [output_encoding])

Updates the decipher with `data`, which is encoded in `'binary'`, `'base64'` or `'hex'`. Defaults to `'binary'`.

The `output_decoding` specifies in what format to return the deciphered plaintext: `'binary'`, `'ascii'` or `'utf8'`. Defaults to `'binary'`.

#### decipher.final([output_encoding])

Returns any remaining plaintext which is deciphered, with `output_encoding` being one of: `'binary'`, `'ascii'` or `'utf8'`. Defaults to `'binary'`.

Note: `decipher` object can not be used after `final()` method been called.

#### decipher.setAutoPadding(auto_padding=true)

You can disable auto padding if the data has been encrypted without standard block padding to prevent `decipher.final` from checking and removing it. Can only work if the input data's length is a multiple of the ciphers block size. You must call this before streaming data to `decipher.update`.

## crypto.createSign(algorithm)

Creates and returns a signing object, with the given algorithm. On recent OpenSSL releases, `openssl list-public-key-algorithms` will display the available signing algorithms. Examples are `'RSA-SHA256'`.

## Class: Signer

Class for generating signatures.

Returned by `crypto.createSign`.

### signer.update(data)

Updates the signer object with data. This can be called many times with new data as it is streamed.

### signer.sign(private_key, [output_format])

Calculates the signature on all the updated data passed through the signer. `private_key` is a string containing the PEM encoded private key for signing.

Returns the signature in `output_format` which can be `'binary'`, `'hex'` or `'base64'`. Defaults to `'binary'`.

Note: `signer` object can not be used after `sign()` method been called.

## crypto.createVerify(algorithm)

Creates and returns a verification object, with the given algorithm. This is the mirror of the signing object above.

## Class: Verify

Class for verifying signatures.

Returned by `crypto.createVerify`.

### verifier.update(data)

Updates the verifier object with data. This can be called many times with new data as it is streamed.

### verifier.verify(object, signature, [signature_format])

Verifies the signed data by using the `object` and `signature`. `object` is a string containing a PEM encoded object, which can be one of RSA public key, DSA public key, or X.509 certificate. `signature` is the previously calculated signature for the data, in the `signature_format` which can be `'binary'`, `'hex'` or `'base64'`. Defaults to `'binary'`.

Returns true or false depending on the validity of the signature for the data and public key.

Note: `verifier` object can not be used after `verify()` method been called.

## crypto.createDiffieHellman(prime_length)

Creates a Diffie-Hellman key exchange object and generates a prime of the given bit length. The generator used is `2`.

## crypto.createDiffieHellman(prime, [encoding])

Creates a Diffie-Hellman key exchange object using the supplied prime. The generator used is `2`. Encoding can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

## Class: DiffieHellman

The class for creating Diffie-Hellman key exchanges.

Returned by `crypto.createDiffieHellman`.

### diffieHellman.generateKeys([encoding])

Generates private and public Diffie-Hellman key values, and returns the public key in the specified encoding. This key should be transferred to the other party. Encoding can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

### diffieHellman.computeSecret(other_public_key, [input_encoding], [output_encoding])

Computes the shared secret using `other_public_key` as the other party's public key and returns the computed shared secret. Supplied key is interpreted using specified `input_encoding`, and secret is encoded using specified `output_encoding`. Encodings can be `'binary'`, `'hex'`, or `'base64'`. The input encoding defaults to `'binary'`. If no output encoding is given, the input encoding is used as output encoding.

### diffieHellman.getPrime([encoding])

Returns the Diffie-Hellman prime in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

### diffieHellman.getGenerator([encoding])

Returns the Diffie-Hellman prime in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

### diffieHellman.getPublicKey([encoding])

Returns the Diffie-Hellman public key in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

### diffieHellman.getPrivateKey([encoding])

Returns the Diffie-Hellman private key in the specified encoding, which can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

### diffieHellman.setPublicKey(public_key, [encoding])

Sets the Diffie-Hellman public key. Key encoding can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

### diffieHellman.setPrivateKey(public_key, [encoding])

Sets the Diffie-Hellman private key. Key encoding can be `'binary'`, `'hex'`, or `'base64'`. Defaults to `'binary'`.

## crypto.getDiffieHellman(group_name)

Creates a predefined Diffie-Hellman key exchange object. The supported groups are: `'modp1'`, `'modp2'`, `'modp5'` (defined in RFC 2412) and `'modp14'`, `'modp15'`, `'modp16'`, `'modp17'`, `'modp18'` (defined in RFC 3526). The returned object mimics the interface of objects created by crypto.createDiffieHellman() above, but will not allow to change the keys (with diffieHellman.setPublicKey() for example). The advantage of using this routine is that the parties don't have to generate nor exchange group modulus beforehand, saving both processor and communication time.

Example (obtaining a shared secret):

```
var crypto = require('crypto');
var alice = crypto.getDiffieHellman('modp5');
var bob = crypto.getDiffieHellman('modp5');

alice.generateKeys();
bob.generateKeys();

var alice_secret = alice.computeSecret(bob.getPublicKey(), 'binary', 'hex');
var bob_secret = bob.computeSecret(alice.getPublicKey(), 'binary', 'hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

### crypto.pbkdf2(password, salt, iterations, keylen, callback)

Asynchronous PBKDF2 applies pseudorandom function HMAC-SHA1 to derive a key of given length from the given password, salt and iterations. The callback gets two arguments `(err, derivedKey)`.

### crypto.randomBytes(size, [callback])

Generates cryptographically strong pseudo-random data. Usage:

```
// async
crypto.randomBytes(256, function(ex, buf) {
  if (ex) throw ex;
  console.log('Have %d bytes of random data: %s', buf.length, buf);
});

// sync
try {
  var buf = crypto.randomBytes(256);
  console.log('Have %d bytes of random data: %s', buf.length, buf);
} catch (ex) {
  // handle error
}
```

# TLS (SSL)

```
Stability: 3 - Stable
```

Use `require('tls')` to access this module.

The `tls` module uses OpenSSL to provide Transport Layer Security and/or Secure Socket Layer: encrypted stream communication.

TLS/SSL is a public/private key infrastructure. Each client and each server must have a private key. A private key is created like this

```
openssl genrsa -out ryans-key.pem 1024
```

All severs and some clients need to have a certificate. Certificates are public keys signed by a Certificate Authority or self-signed. The first step to getting a certificate is to create a "Certificate Signing Request" (CSR) file. This is done with:

```
openssl req -new -key ryans-key.pem -out ryans-csr.pem
```

To create a self-signed certificate with the CSR, do this:

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

Alternatively you can send the CSR to a Certificate Authority for signing.

(TODO: docs on creating a CA, for now interested users should just look at `test/fixtures/keys/Makefile` in the Node source code)

To create .pfx or .p12, do this:

```
openssl pkcs12 -export -in agent5-cert.pem -inkey agent5-key.pem \
    -certfile ca-cert.pem -out agent5.pfx
```

- `in`: certificate
- `inkey`: private key
- `certfile`: all CA certs concatenated in one file like `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

## Client-initiated renegotiation attack mitigation

The TLS protocol lets the client renegotiate certain aspects of the TLS session. Unfortunately, session renegotiation requires a disproportional amount of server-side resources, which makes it a potential vector for denial-of-service attacks.

To mitigate this, renegotiations are limited to three times every 10 minutes. An error is emitted on the CleartextStream instance when the threshold is exceeded. The limits are configurable:

- `tls.CLIENT_RENEG_LIMIT`: renegotiation limit, default is 3.
- `tls.CLIENT_RENEG_WINDOW`: renegotiation window in seconds, default is 10 minutes.

Don't change the defaults unless you know what you are doing.

To test your server, connect to it with `openssl s_client -connect address:port` and tap `R<CR>` (that's the letter `R` followed by a carriage return) a few times.

## NPN and SNI

NPN (Next Protocol Negotiation) and SNI (Server Name Indication) are TLS handshake extensions allowing you:

- NPN - to use one TLS server for multiple protocols (HTTP, SPDY)
- SNI - to use one TLS server for multiple hostnames with different SSL certificates.

### tls.createServer(options, [secureConnectionListener])

Creates a new tls.Server. The `connectionListener` argument is automatically set as a listener for the secureConnection event. The `options` object has these possibilities:

- `pfx`: A string or `Buffer` containing the private key, certificate and CA certs of the server in PFX or PKCS12 format. (Mutually exclusive with the `key`, `cert` and `ca` options.)
- `key`: A string or `Buffer` containing the private key of the server in PEM format. (Required)
- `passphrase`: A string of passphrase for the private key or pfx.
- `cert`: A string or `Buffer` containing the certificate key of the server in PEM format. (Required)
- `ca`: An array of strings or `Buffer`s of trusted certificates. If this is omitted several well known "root" CAs will be used, like VeriSign. These are used to authorize connections.
- `crl` : Either a string or list of strings of PEM encoded CRLs (Certificate Revocation List)
- `ciphers`: A string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format. To mitigate [BEAST attacks] (http://blog.ivanristic.com/2011/10/mitigating-the-beast-attack-on-tls.html), it is recommended that you use this option in conjunction with the `honorCipherOrder` option described below to prioritize the RC4 algorithm, since it is a non-CBC cipher. A recommended cipher list follows: `ECDHE-RSA-AES256-SHA:AES256-SHA:RC4-SHA:RC4:HIGH:!MD5:!aNULL:!EDH:!AESGCM`
- `honorCipherOrder` : When choosing a cipher, use the server's preferences instead of the client preferences. Note that if SSLv2 is used, the server will send its list of preferences to the client, and the client chooses the cipher. Although, this option is disabled by default, it is *recommended* that you use this option in conjunction with the `ciphers` option to mitigate BEAST attacks.
- `requestCert`: If `true` the server will request a certificate from clients that connect and attempt to verify that certificate. Default: `false`.
- `rejectUnauthorized`: If `true` the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if `requestCert` is `true`. Default: `false`.
- `NPNProtocols`: An array or `Buffer` of possible NPN protocols. (Protocols should be ordered by their priority).
- `SNICallback`: A function that will be called if client supports SNI TLS extension. Only one argument will be passed to it: `servername`. And `SNICallback` should return SecureContext instance. (You can use `crypto.createCredentials(...).context` to get proper SecureContext). If `SNICallback` wasn't provided - default callback with high-level API will be used (see below).
- `sessionIdContext`: A string containing a opaque identifier for session resumption. If `requestCert` is `true`, the default is MD5 hash value generated from command-line. Otherwise, the default is not provided.

Here is a simple example echo server:

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

var options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),

  // This is necessary only if using the client certificate authentication.
  requestCert: true,

  // This is necessary only if the client uses the self-signed certificate.
  ca: [ fs.readFileSync('client-cert.pem') ]
};

var server = tls.createServer(options, function(cleartextStream) {
  console.log('server connected',
              cleartextStream.authorized ? 'authorized' : 'unauthorized');
  cleartextStream.write("welcome!\n");
```

```
  cleartextStream.setEncoding('utf8');
  cleartextStream.pipe(cleartextStream);
});
server.listen(8000, function() {
  console.log('server bound');
});
```

Or

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

var options = {
  pfx: fs.readFileSync('server.pfx'),

  // This is necessary only if using the client certificate authentication.
  requestCert: true,

};

var server = tls.createServer(options, function(cleartextStream) {
  console.log('server connected',
              cleartextStream.authorized ? 'authorized' : 'unauthorized');
  cleartextStream.write("welcome!\n");
  cleartextStream.setEncoding('utf8');
  cleartextStream.pipe(cleartextStream);
});
server.listen(8000, function() {
  console.log('server bound');
});
```

You can test this server by connecting to it with `openssl s_client`:

```
openssl s_client -connect 127.0.0.1:8000
```

### tls.connect(options, [secureConnectListener])

### tls.connect(port, [host], [options], [secureConnectListener])

Creates a new client connection to the given `port` and `host` (old API) or `options.port` and `options.host`. (If `host` is omitted, it defaults to `localhost`.) `options` should be an object which specifies:

- `host`: Host the client should connect to
- `port`: Port the client should connect to
- `socket`: Establish secure connection on a given socket rather than creating a new socket. If this option is specified, `host` and `port` are ignored.
- `pfx`: A string or `Buffer` containing the private key, certificate and CA certs of the server in PFX or PKCS12 format.
- `key`: A string or `Buffer` containing the private key of the client in PEM format.
- `passphrase`: A string of passphrase for the private key or pfx.
- `cert`: A string or `Buffer` containing the certificate key of the client in PEM format.
- `ca`: An array of strings or `Buffer`s of trusted certificates. If this is omitted several well known "root" CAs will be used, like VeriSign. These are used to authorize connections.
- `rejectUnauthorized`: If `true`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails. Default: `false`.
- `NPNProtocols`: An array of string or `Buffer` containing supported NPN protocols. `Buffer` should have following format: `0x05hello0x05world`, where first byte is next protocol name's length. (Passing array should usually be much simpler: `['hello', 'world']`.)

- `servername`: Servername for SNI (Server Name Indication) TLS extension.

The `secureConnectListener` parameter will be added as a listener for the 'secureConnect' event.

`tls.connect()` returns a CleartextStream object.

Here is an example of a client of echo server as described previously:

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

var options = {
  // These are necessary only if using the client certificate authentication
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // This is necessary only if the server uses the self-signed certificate
  ca: [ fs.readFileSync('server-cert.pem') ]
};

var cleartextStream = tls.connect(8000, options, function() {
  console.log('client connected',
              cleartextStream.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(cleartextStream);
  process.stdin.resume();
});
cleartextStream.setEncoding('utf8');
cleartextStream.on('data', function(data) {
  console.log(data);
});
cleartextStream.on('end', function() {
  server.close();
});
```

Or

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

var options = {
  pfx: fs.readFileSync('client.pfx')
};

var cleartextStream = tls.connect(8000, options, function() {
  console.log('client connected',
              cleartextStream.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(cleartextStream);
  process.stdin.resume();
});
cleartextStream.setEncoding('utf8');
cleartextStream.on('data', function(data) {
  console.log(data);
});
cleartextStream.on('end', function() {
  server.close();
});
```

### tls.createSecurePair([credentials], [isServer], [requestCert], [rejectUnauthorized])

Creates a new secure pair object with two streams, one of which reads/writes encrypted data, and one reads/writes cleartext data. Generally the encrypted one is piped to/from an incoming encrypted data stream, and the cleartext one is used as a replacement for the initial encrypted stream.

- `credentials`: A credentials object from crypto.createCredentials( ... )

- `isServer`: A boolean indicating whether this tls connection should be opened as a server or a client.
- `requestCert`: A boolean indicating whether a server should request a certificate from a connecting client. Only applies to server connections.
- `rejectUnauthorized`: A boolean indicating whether a server should automatically reject clients with invalid certificates. Only applies to servers with `requestCert` enabled.

`tls.createSecurePair()` returns a SecurePair object with [cleartext][] and `encrypted` stream properties.

## Class: SecurePair

Returned by tls.createSecurePair.

### Event: 'secure'

The event is emitted from the SecurePair once the pair has successfully established a secure connection.

Similarly to the checking for the server 'secureConnection' event, pair.cleartext.authorized should be checked to confirm whether the certificate used properly authorized.

## Class: tls.Server

This class is a subclass of `net.Server` and has the same methods on it. Instead of accepting just raw TCP connections, this accepts encrypted connections using TLS or SSL.

### Event: 'secureConnection'

```
function (cleartextStream) {}
```

This event is emitted after a new connection has been successfully handshaked. The argument is a instance of CleartextStream. It has all the common stream methods and events.

`cleartextStream.authorized` is a boolean value which indicates if the client has verified by one of the supplied certificate authorities for the server. If `cleartextStream.authorized` is false, then `cleartextStream.authorizationError` is set to describe how authorization failed. Implied but worth mentioning: depending on the settings of the TLS server, you unauthorized connections may be accepted. `cleartextStream.npnProtocol` is a string containing selected NPN protocol. `cleartextStream.servername` is a string containing servername requested with SNI.

### Event: 'clientError'

```
function (exception) { }
```

When a client connection emits an 'error' event before secure connection is established - it will be forwarded here.

### server.listen(port, [host], [callback])

Begin accepting connections on the specified `port` and `host`. If the `host` is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`).

This function is asynchronous. The last parameter `callback` will be called when the server has been bound.

See `net.Server` for more information.

### server.close()

Stops the server from accepting new connections. This function is asynchronous, the server is finally closed when the server emits a `'close'` event.

#### server.address()

Returns the bound address, the address family name and port of the server as reported by the operating system. See net.Server.address() for more information.

#### server.addContext(hostname, credentials)

Add secure context that will be used if client request's SNI hostname is matching passed `hostname` (wildcards can be used). `credentials` can contain `key`, `cert` and `ca`.

#### server.maxConnections

Set this property to reject connections when the server's connection count gets high.

#### server.connections

The number of concurrent connections on the server.

### Class: tls.CleartextStream

This is a stream on top of the *Encrypted* stream that makes it possible to read/write an encrypted data as a cleartext data.

This instance implements a duplex Stream interfaces. It has all the common stream methods and events.

A ClearTextStream is the `clear` member of a SecurePair object.

#### Event: 'secureConnect'

This event is emitted after a new connection has been successfully handshaked. The listener will be called no matter if the server's certificate was authorized or not. It is up to the user to test `cleartextStream.authorized` to see if the server certificate was signed by one of the specified CAs. If `cleartextStream.authorized === false` then the error can be found in `cleartextStream.authorizationError`. Also if NPN was used - you can check `cleartextStream.npnProtocol` for negotiated protocol.

#### cleartextStream.authorized

A boolean that is `true` if the peer certificate was signed by one of the specified CAs, otherwise `false`

#### cleartextStream.authorizationError

The reason why the peer's certificate has not been verified. This property becomes available only when `cleartextStream.authorized === false`.

#### cleartextStream.getPeerCertificate()

Returns an object representing the peer's certificate. The returned object has some properties corresponding to the field of the certificate.

Example:

```
{ subject:
   { C: 'UK',
     ST: 'Acknack Ltd',
     L: 'Rhys Jones',
     O: 'node.js',
     OU: 'Test TLS Certificate',
     CN: 'localhost' },
```

```
  issuer:
   { C: 'UK',
     ST: 'Acknack Ltd',
     L: 'Rhys Jones',
     O: 'node.js',
     OU: 'Test TLS Certificate',
     CN: 'localhost' },
  valid_from: 'Nov 11 09:52:22 2009 GMT',
  valid_to: 'Nov  6 09:52:22 2029 GMT',
  fingerprint: '2A:7A:C2:DD:E5:F9:CC:53:72:35:99:7A:02:5A:71:38:52:EC:8A:DF' }
```

If the peer does not provide a certificate, it returns `null` or an empty object.

#### cleartextStream.getCipher()

Returns an object representing the cipher name and the SSL/TLS protocol version of the current connection.

Example: { name: 'AES256-SHA', version: 'TLSv1/SSLv3' }

See SSL_CIPHER_get_name() and SSL_CIPHER_get_version() in http://www.openssl.org/docs/ssl/ssl.html#DEALING
for more information.

#### cleartextStream.address()

Returns the bound address, the address family name and port of the underlying socket as reported by the operating system. Returns an object with three properties, e.g. `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

#### cleartextStream.remoteAddress

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.

#### cleartextStream.remotePort

The numeric representation of the remote port. For example, `443`.

# StringDecoder

```
Stability: 3 - Stable
```

To use this module, do `require('string_decoder')`. StringDecoder decodes a buffer to a string. It is a simple interface to `buffer.toString()` but provides additional support for utf8.

```
var StringDecoder = require('string_decoder').StringDecoder;
var decoder = new StringDecoder('utf8');

var cent = new Buffer([0xC2, 0xA2]);
console.log(decoder.write(cent));

var euro = new Buffer([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

## Class: StringDecoder

Accepts a single argument, `encoding` which defaults to `utf8`.

### StringDecoder.write(buffer)

Returns a decoded string.

# File System

```
Stability: 3 - Stable
```

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

The asynchronous form always take a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Here is an example of the asynchronous version:

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

fs.unlink('/tmp/hello', function (err) {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Here is the synchronous version:

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

fs.unlinkSync('/tmp/hello')
console.log('successfully deleted /tmp/hello');
```

With the asynchronous methods there is no guaranteed ordering. So the following is prone to error:

```
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', function (err, stats) {
  if (err) throw err;
  console.log('stats: ' + JSON.stringify(stats));
});
```

It could be that `fs.stat` is executed before `fs.rename`. The correct way to do this is to chain the callbacks.

```
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  fs.stat('/tmp/world', function (err, stats) {
    if (err) throw err;
    console.log('stats: ' + JSON.stringify(stats));
  });
});
```

In busy processes, the programmer is *strongly encouraged* to use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete–halting all connections.

Relative path to filename can be used, remember however that this path will be relative to `process.cwd()`.

## fs.rename(oldPath, newPath, [callback])

Asynchronous rename(2). No arguments other than a possible exception are given to the completion callback.

## fs.renameSync(oldPath, newPath)

Synchronous rename(2).

### fs.truncate(fd, len, [callback])

Asynchronous ftruncate(2). No arguments other than a possible exception are given to the completion callback.

### fs.truncateSync(fd, len)

Synchronous ftruncate(2).

### fs.chown(path, uid, gid, [callback])

Asynchronous chown(2). No arguments other than a possible exception are given to the completion callback.

### fs.chownSync(path, uid, gid)

Synchronous chown(2).

### fs.fchown(fd, uid, gid, [callback])

Asynchronous fchown(2). No arguments other than a possible exception are given to the completion callback.

### fs.fchownSync(fd, uid, gid)

Synchronous fchown(2).

### fs.lchown(path, uid, gid, [callback])

Asynchronous lchown(2). No arguments other than a possible exception are given to the completion callback.

### fs.lchownSync(path, uid, gid)

Synchronous lchown(2).

### fs.chmod(path, mode, [callback])

Asynchronous chmod(2). No arguments other than a possible exception are given to the completion callback.

### fs.chmodSync(path, mode)

Synchronous chmod(2).

### fs.fchmod(fd, mode, [callback])

Asynchronous fchmod(2). No arguments other than a possible exception are given to the completion callback.

### fs.fchmodSync(fd, mode)

Synchronous fchmod(2).

### fs.lchmod(path, mode, [callback])

Asynchronous lchmod(2). No arguments other than a possible exception are given to the completion callback.

### fs.lchmodSync(path, mode)

Synchronous lchmod(2).

### fs.stat(path, [callback])

Asynchronous stat(2). The callback gets two arguments `(err, stats)` where `stats` is a fs.Stats object. See the fs.Stats section below for more information.

### fs.lstat(path, [callback])

Asynchronous lstat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `lstat()` is identical to `stat()`, except that if `path` is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

### fs.fstat(fd, [callback])

Asynchronous fstat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `fstat()` is identical to `stat()`, except that the file to be stat-ed is specified by the file descriptor `fd`.

### fs.statSync(path)

Synchronous stat(2). Returns an instance of `fs.Stats`.

### fs.lstatSync(path)

Synchronous lstat(2). Returns an instance of `fs.Stats`.

### fs.fstatSync(fd)

Synchronous fstat(2). Returns an instance of `fs.Stats`.

### fs.link(srcpath, dstpath, [callback])

Asynchronous link(2). No arguments other than a possible exception are given to the completion callback.

### fs.linkSync(srcpath, dstpath)

Synchronous link(2).

### fs.symlink(destination, path, [type], [callback])

Asynchronous symlink(2). No arguments other than a possible exception are given to the completion callback. `type` argument can be either `'dir'`, `'file'`, or `'junction'` (default is `'file'`). It is only used on Windows (ignored on other platforms). Note that Windows junction points require the destination path to be absolute. When using `'junction'`, the `destination` argument will automatically be normalized to absolute path.

### fs.symlinkSync(destination, path, [type])

Synchronous symlink(2).

### fs.readlink(path, [callback])

Asynchronous readlink(2). The callback gets two arguments `(err, linkString)`.

### fs.readlinkSync(path)

Synchronous readlink(2). Returns the symbolic link's string value.

### fs.realpath(path, [cache], callback)

Asynchronous realpath(2). The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd` to resolve relative paths. `cache` is an object literal of mapped paths that can be used to force a specific path resolution or avoid additional `fs.stat` calls for known real paths.

Example:

```
var cache = {'/etc':'/private/etc'};
fs.realpath('/etc/passwd', cache, function (err, resolvedPath) {
  if (err) throw err;
  console.log(resolvedPath);
});
```

### fs.realpathSync(path, [cache])

Synchronous realpath(2). Returns the resolved path.

### fs.unlink(path, [callback])

Asynchronous unlink(2). No arguments other than a possible exception are given to the completion callback.

### fs.unlinkSync(path)

Synchronous unlink(2).

### fs.rmdir(path, [callback])

Asynchronous rmdir(2). No arguments other than a possible exception are given to the completion callback.

### fs.rmdirSync(path)

Synchronous rmdir(2).

### fs.mkdir(path, [mode], [callback])

Asynchronous mkdir(2). No arguments other than a possible exception are given to the completion callback. `mode` defaults to `0777`.

### fs.mkdirSync(path, [mode])

Synchronous mkdir(2).

### fs.readdir(path, [callback])

Asynchronous readdir(3). Reads the contents of a directory. The callback gets two arguments `(err, files)` where `files` is an array of the names of the files in the directory excluding `'.'` and `'..'`.

### fs.readdirSync(path)

Synchronous readdir(3). Returns an array of filenames excluding `'.'` and `'..'`.

### fs.close(fd, [callback])

Asynchronous close(2). No arguments other than a possible exception are given to the completion callback.

### fs.closeSync(fd)

Synchronous close(2).

### fs.open(path, flags, [mode], [callback])

Asynchronous file open. See open(2). `flags` can be:

- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs'` - Open file for reading in synchronous mode. Instructs the operating system to bypass the local file system cache.

This is primarily useful for opening files on NFS mounts as it allows you to skip the potentially stale local cache. It has a very real impact on I/O performance so don't use this mode unless you need it.

Note that this doesn't turn `fs.open()` into a synchronous blocking call. If that's what you want then you should be using `fs.openSync()`

- `'rs+'` - Open file for reading and writing, telling the OS to open it synchronously. See notes for `'rs'` about using this with caution.
- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` - Like `'w'` but opens the file in exclusive mode.
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` - Like `'w+'` but opens the file in exclusive mode.
- `'a'` - Open file for appending. The file is created if it does not exist.
- `'ax'` - Like `'a'` but opens the file in exclusive mode.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` - Like `'a+'` but opens the file in exclusive mode.

`mode` defaults to `0666`. The callback gets two arguments `(err, fd)`.

Exclusive mode (`O_EXCL`) ensures that `path` is newly created. `fs.open()` fails if a file by that name already exists. On POSIX systems, symlinks are not followed. Exclusive mode may or may not work with network file systems.

### fs.openSync(path, flags, [mode])

Synchronous open(2).

### fs.utimes(path, atime, mtime, [callback])

### fs.utimesSync(path, atime, mtime)

Change file timestamps of the file referenced by the supplied path.

### fs.futimes(fd, atime, mtime, [callback])

### fs.futimesSync(fd, atime, mtime)

Change the file timestamps of a file referenced by the supplied file descriptor.

### fs.fsync(fd, [callback])

Asynchronous fsync(2). No arguments other than a possible exception are given to the completion callback.

### fs.fsyncSync(fd)

Synchronous fsync(2).

### fs.write(fd, buffer, offset, length, position, [callback])

Write `buffer` to the file specified by `fd`.

`offset` and `length` determine the part of the buffer to be written.

`position` refers to the offset from the beginning of the file where this data should be written. If `position` is `null`, the data will be written at the current position. See pwrite(2).

The callback will be given three arguments `(err, written, buffer)` where `written` specifies how many *bytes* were written from `buffer`.

Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

### fs.writeSync(fd, buffer, offset, length, position)

Synchronous version of buffer-based `fs.write()`. Returns the number of bytes written.

### fs.writeSync(fd, str, position, [encoding])

Synchronous version of string-based `fs.write()`. `encoding` defaults to `'utf8'`. Returns the number of *bytes* written.

### fs.read(fd, buffer, offset, length, position, [callback])

Read data from the file specified by `fd`.

`buffer` is the buffer that the data will be written to.

`offset` is offset within the buffer where writing will start.

`length` is an integer specifying the number of bytes to read.

`position` is an integer specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position.

The callback is given the three arguments, `(err, bytesRead, buffer)`.

### fs.readSync(fd, buffer, offset, length, position)

Synchronous version of buffer-based `fs.read`. Returns the number of `bytesRead`.

### fs.readSync(fd, length, position, encoding)

Legacy synchronous version of string-based `fs.read`. Returns an array with the data from the file specified and number of bytes read, `[string, bytesRead]`.

### fs.readFile(filename, [encoding], [callback])

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

### fs.readFileSync(filename, [encoding])

Synchronous version of `fs.readFile`. Returns the contents of the `filename`.

If `encoding` is specified then this function returns a string. Otherwise it returns a buffer.

### fs.writeFile(filename, data, [encoding], [callback])

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer. The `encoding` argument is ignored if `data` is a buffer. It defaults to `'utf8'`.

Example:

```
fs.writeFile('message.txt', 'Hello Node', function (err) {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

### fs.writeFileSync(filename, data, [encoding])

The synchronous version of `fs.writeFile`.

### fs.appendFile(filename, data, encoding='utf8', [callback])

Asynchronously append data to a file, creating the file if it not yet exists. `data` can be a string or a buffer. The `encoding` argument is ignored if `data` is a buffer.

Example:

```
fs.appendFile('message.txt', 'data to append', function (err) {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

### fs.appendFileSync(filename, data, encoding='utf8')

The synchronous version of `fs.appendFile`.

### fs.watchFile(filename, [options], listener)

```
Stability: 2 - Unstable.  Use fs.watch instead, if available.
```

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

The second argument is optional. The `options` if provided should be an object containing two members a boolean, `persistent`, and `interval`. `persistent` indicates whether the process should continue to run as long as files are being watched. `interval` indicates how often the target should be polled, in milliseconds. The default is `{ persistent: true, interval: 5007 }`.

The `listener` gets two arguments the current stat object and the previous stat object:

```
fs.watchFile('message.text', function (curr, prev) {
  console.log('the current mtime is: ' + curr.mtime);
  console.log('the previous mtime was: ' + prev.mtime);
});
```

These stat objects are instances of `fs.Stat`.

If you want to be notified when the file was modified, not just accessed you need to compare `curr.mtime` and `prev.mtime`.

## fs.unwatchFile(filename)

```
Stability: 2 - Unstable.  Use fs.watch instead, if available.
```

Stop watching for changes on `filename`.

## fs.watch(filename, [options], [listener])

```
Stability: 2 - Unstable.  Not available on all platforms.
```

Watch for changes on `filename`, where `filename` is either a file or a directory. The returned object is a fs.FSWatcher.

The second argument is optional. The `options` if provided should be an object containing a boolean member `persistent`, which indicates whether the process should continue to run as long as files are being watched. The default is `{ persistent: true }`.

The listener callback gets two arguments `(event, filename)`. `event` is either 'rename' or 'change', and `filename` is the name of the file which triggered the event.

### Caveats

The `fs.watch` API is not 100% consistent across platforms, and is unavailable in some situations.

**Availability** This feature depends on the underlying operating system providing a way to be notified of filesystem changes.

- On Linux systems, this uses `inotify`.
- On BSD systems (including OS X), this uses `kqueue`.
- On SunOS systems (including Solaris and SmartOS), this uses `event ports`.
- On Windows systems, this feature depends on `ReadDirectoryChangesW`.

If the underlying functionality is not available for some reason, then `fs.watch` will not be able to function. You can still use `fs.watchFile`, which uses stat polling, but it is slower and less reliable.

**Filename Argument** Providing `filename` argument in the callback is not supported on every platform (currently it's only supported on Linux and Windows). Even on supported platforms `filename` is not always guaranteed to be provided. Therefore, don't assume that `filename` argument is always provided in the callback, and have some fallback logic if it is null.

```
fs.watch('somedir', function (event, filename) {
  console.log('event is: ' + event);
  if (filename) {
    console.log('filename provided: ' + filename);
  } else {
    console.log('filename not provided');
  }
});
```

### fs.exists(path, [callback])

Test whether or not the given path exists by checking with the file system. Then call the `callback` argument with either true or false. Example:

```
fs.exists('/etc/passwd', function (exists) {
  util.debug(exists ? "it's there" : "no passwd!");
});
```

### fs.existsSync(path)

Synchronous version of `fs.exists`.

### Class: fs.Stats

Objects returned from `fs.stat()`, `fs.lstat()` and `fs.fstat()` and their synchronous counterparts are of this type.

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()` (only valid with `fs.lstat()`)
- `stats.isFIFO()`
- `stats.isSocket()`

For a regular file `util.inspect(stats)` would return a string very similar to this:

```
{ dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT }
```

Please note that `atime`, `mtime` and `ctime` are instances of Date object and to compare the values of these objects you should use appropriate methods. For most general uses getTime() will return the number of milliseconds elapsed since *1 January 1970 00:00:00 UTC* and this integer should be sufficient for any comparison, however there additional methods which can be used for displaying fuzzy information. More details can be found in the MDN JavaScript Reference page.

### fs.createReadStream(path, [options])

Returns a new ReadStream object (See `Readable Stream`).

`options` is an object with the following defaults:

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0666,
  bufferSize: 64 * 1024
}
```

`options` can include `start` and `end` values to read a range of bytes from the file instead of the entire file. Both `start` and `end` are inclusive and start at 0. The `encoding` can be `'utf8'`, `'ascii'`, or `'base64'`.

An example to read the last 10 bytes of a file which is 100 bytes long:

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

### Class: fs.ReadStream

`ReadStream` is a Readable Stream.

#### Event: 'open'

- `fd` {Integer} file descriptor used by the ReadStream.

Emitted when the ReadStream's file is opened.

### fs.createWriteStream(path, [options])

Returns a new WriteStream object (See `Writable Stream`).

`options` is an object with the following defaults:

```
{ flags: 'w',
  encoding: null,
  mode: 0666 }
```

`options` may also include a `start` option to allow writing data at some position past the beginning of the file. Modifying a file rather than replacing it may require a `flags` mode of `r+` rather than the default mode `w`.

### fs.WriteStream

`WriteStream` is a Writable Stream.

#### Event: 'open'

- `fd` {Integer} file descriptor used by the WriteStream.

Emitted when the WriteStream's file is opened.

#### file.bytesWritten

The number of bytes written so far. Does not include data that is still queued for writing.

### Class: fs.FSWatcher

Objects returned from `fs.watch()` are of this type.

### watcher.close()

Stop watching for changes on the given `fs.FSWatcher`.

### Event: 'change'

- `event` {String} The type of fs change
- `filename` {String} The filename that changed (if relevant/available)

Emitted when something changes in a watched directory or file. See more details in fs.watch.

### Event: 'error'

- `error` {Error object}

Emitted when an error occurs.

# Path

```
Stability: 3 - Stable
```

This module contains utilities for handling and transforming file paths. Almost all these methods perform only string transformations. The file system is not consulted to check whether paths are valid.

Use `require('path')` to use this module. The following methods are provided:

## path.normalize(p)

Normalize a string path, taking care of `'..'` and `'.'` parts.

When multiple slashes are found, they're replaced by a single one; when the path contains a trailing slash, it is preserved. On windows backslashes are used.

Example:

```
path.normalize('/foo/bar//baz/asdf/quux/..')
// returns
'/foo/bar/baz/asdf'
```

## path.join([path1], [path2], [...])

Join all arguments together and normalize the resulting path. Non-string arguments are ignored.

Example:

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
// returns
'/foo/bar/baz/asdf'

path.join('foo', {}, 'bar')
// returns
'foo/bar'
```

## path.resolve([from ...], to)

Resolves `to` to an absolute path.

If `to` isn't already absolute `from` arguments are prepended in right to left order, until an absolute path is found. If after using all `from` paths still no absolute path is found, the current working directory is used as well. The resulting path is normalized, and trailing slashes are removed unless the path gets resolved to the root directory. Non-string arguments are ignored.

Another way to think of it is as a sequence of `cd` commands in a shell.

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

Is similar to:

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

The difference is that the different paths don't need to exist and may also be files.

Examples:

```
path.resolve('/foo/bar', './baz')
// returns
'/foo/bar/baz'
```

```
path.resolve('/foo/bar', '/tmp/file/')
// returns
'/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
// if currently in /home/myself/node, it returns
'/home/myself/node/wwwroot/static_files/gif/image.gif'
```

### path.relative(from, to)

Solve the relative path from `from` to `to`.

At times we have two absolute paths, and we need to derive the relative path from one to the other. This is actually the reverse transform of `path.resolve`, which means we see that:

```
path.resolve(from, path.relative(from, to)) == path.resolve(to)
```

Examples:

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb')
// returns
'..\\..\\impl\\bbb'

path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb')
// returns
'../../impl/bbb'
```

### path.dirname(p)

Return the directory name of a path. Similar to the Unix `dirname` command.

Example:

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns
'/foo/bar/baz/asdf'
```

### path.basename(p, [ext])

Return the last portion of a path. Similar to the Unix `basename` command.

Example:

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns
'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns
'quux'
```

### path.extname(p)

Return the extension of the path, from the last '.' to end of string in the last portion of the path. If there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string. Examples:

```
path.extname('index.html')
// returns
'.html'

path.extname('index.')
```

```
// returns
'.'

path.extname('index')
// returns
''
```

### path.sep

The platform-specific file separator. `'\\'` or `'/'`.

An example on linux:

```
'foo/bar/baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

An example on windows:

```
'foo\\bar\\baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

# net

```
Stability: 3 - Stable
```

The `net` module provides you with an asynchronous network wrapper. It contains methods for creating both servers and clients (called streams). You can include this module with `require('net');`

## net.createServer([options], [connectionListener])

Creates a new TCP server. The `connectionListener` argument is automatically set as a listener for the 'connection' event.

`options` is an object with the following defaults:

```
{ allowHalfOpen: false
}
```

If `allowHalfOpen` is `true`, then the socket won't automatically send a FIN packet when the other end of the socket sends a FIN packet. The socket becomes non-readable, but still writable. You should call the `end()` method explicitly. See 'end' event for more information.

Here is an example of a echo server which listens for connections on port 8124:

```
var net = require('net');
var server = net.createServer(function(c) { //'connection' listener
  console.log('server connected');
  c.on('end', function() {
    console.log('server disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.listen(8124, function() { //'listening' listener
  console.log('server bound');
});
```

Test this by using `telnet`:

```
telnet localhost 8124
```

To listen on the socket `/tmp/echo.sock` the third line from the last would just be changed to

```
server.listen('/tmp/echo.sock', function() { //'listening' listener
```

Use `nc` to connect to a UNIX domain socket server:

```
nc -U /tmp/echo.sock
```

## net.connect(options, [connectionListener])

## net.createConnection(options, [connectionListener])

Constructs a new socket object and opens the socket to the given location. When the socket is established, the 'connect' event will be emitted.

For TCP sockets, `options` argument should be an object which specifies:

- `port`: Port the client should connect to (Required).
- `host`: Host the client should connect to. Defaults to `'localhost'`.
- `localAddress`: Local interface to bind to for network connections.

For UNIX domain sockets, `options` argument should be an object which specifies:

- `path`: Path the client should connect to (Required).

Common options are:

- `allowHalfOpen`: if `true`, the socket won't automatically send a FIN packet when the other end of the socket sends a FIN packet. Defaults to `false`. See 'end' event for more information.

The `connectListener` parameter will be added as an listener for the 'connect' event.

Here is an example of a client of echo server as described previously:

```
var net = require('net');
var client = net.connect({port: 8124},
    function() { //'connect' listener
  console.log('client connected');
  client.write('world!\r\n');
});
client.on('data', function(data) {
  console.log(data.toString());
  client.end();
});
client.on('end', function() {
  console.log('client disconnected');
});
```

To connect on the socket `/tmp/echo.sock` the second line would just be changed to

```
var client = net.connect({path: '/tmp/echo.sock'},
```

### net.connect(port, [host], [connectListener])

### net.createConnection(port, [host], [connectListener])

Creates a TCP connection to `port` on `host`. If `host` is omitted, `'localhost'` will be assumed. The `connectListener` parameter will be added as an listener for the 'connect' event.

### net.connect(path, [connectListener])

### net.createConnection(path, [connectListener])

Creates unix socket connection to `path`. The `connectListener` parameter will be added as an listener for the 'connect' event.

### Class: net.Server

This class is used to create a TCP or UNIX server. A server is a `net.Socket` that can listen for new incoming connections.

#### server.listen(port, [host], [backlog], [listeningListener])

Begin accepting connections on the specified `port` and `host`. If the `host` is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`). A port value of zero will assign a random port.

Backlog is the maximum length of the queue of pending connections. The actual length will be determined by your OS through sysctl settings such as `tcp_max_syn_backlog` and `somaxconn` on linux. The default value of this parameter is 511 (not 512).

This function is asynchronous. When the server has been bound, 'listening' event will be emitted. The last parameter `listeningListener` will be added as an listener for the 'listening' event.

One issue some users run into is getting `EADDRINUSE` errors. This means that another server is already running on the requested port. One way of handling this would be to wait a second and then try again. This can be done with

```
server.on('error', function (e) {
  if (e.code == 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(function () {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

(Note: All sockets in Node set `SO_REUSEADDR` already)

### server.listen(path, [listeningListener])

Start a UNIX socket server listening for connections on the given `path`.

This function is asynchronous. When the server has been bound, 'listening' event will be emitted. The last parameter `listeningListener` will be added as an listener for the 'listening' event.

### server.listen(handle, [listeningListener])

- `handle` {Object}
- `listeningListener` {Function}

The `handle` object can be set to either a server or socket (anything with an underlying `_handle` member), or a `{fd: <n>}` object.

This will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket.

Listening on a file descriptor is not supported on Windows.

This function is asynchronous. When the server has been bound, 'listening' event will be emitted. the last parameter `listeningListener` will be added as an listener for the 'listening' event.

### server.close([cb])

Stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a `'close'` event. Optionally, you can pass a callback to listen for the `'close'` event.

### server.address()

Returns the bound address, the address family name and port of the server as reported by the operating system. Useful to find which port was assigned when giving getting an OS-assigned address. Returns an object with three properties, e.g. `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

Example:

```
var server = net.createServer(function (socket) {
  socket.end("goodbye\n");
});

// grab a random port.
server.listen(function() {
  address = server.address();
  console.log("opened server on %j", address);
});
```

Don't call `server.address()` until the `'listening'` event has been emitted.

#### server.maxConnections

Set this property to reject connections when the server's connection count gets high.

It is not recommended to use this option once a socket has been sent to a child with `child_process.fork()`.

#### server.connections

The number of concurrent connections on the server.

This becomes `null` when sending a socket to a child with `child_process.fork()`.

`net.Server` is an EventEmitter with the following events:

#### Event: 'listening'

Emitted when the server has been bound after calling `server.listen`.

#### Event: 'connection'

- {Socket object} The connection object

Emitted when a new connection is made. `socket` is an instance of `net.Socket`.

#### Event: 'close'

Emitted when the server closes. Note that if connections exist, this event is not emitted until all connections are ended.

#### Event: 'error'

- {Error Object}

Emitted when an error occurs. The `'close'` event will be called directly following this event. See example in discussion of `server.listen`.

### Class: net.Socket

This object is an abstraction of a TCP or UNIX socket. `net.Socket` instances implement a duplex Stream interface. They can be created by the user and used as a client (with `connect()`) or they can be created by Node and passed to the user through the `'connection'` event of a server.

#### new net.Socket([options])

Construct a new socket object.

`options` is an object with the following defaults:

```
{ fd: null
  type: null
  allowHalfOpen: false
}
```

`fd` allows you to specify the existing file descriptor of socket. `type` specified underlying protocol. It can be `'tcp4'`, `'tcp6'`, or `'unix'`. About `allowHalfOpen`, refer to `createServer()` and `'end'` event.

### socket.connect(port, [host], [connectListener])

### socket.connect(path, [connectListener])

Opens the connection for a given socket. If `port` and `host` are given, then the socket will be opened as a TCP socket, if `host` is omitted, `localhost` will be assumed. If a `path` is given, the socket will be opened as a unix socket to that path.

Normally this method is not needed, as `net.createConnection` opens the socket. Use this only if you are implementing a custom Socket or if a Socket is closed and you want to reuse it to connect to another server.

This function is asynchronous. When the 'connect' event is emitted the socket is established. If there is a problem connecting, the `'connect'` event will not be emitted, the `'error'` event will be emitted with the exception.

The `connectListener` parameter will be added as an listener for the 'connect' event.

### socket.bufferSize

`net.Socket` has the property that `socket.write()` always works. This is to help users get up and running quickly. The computer cannot always keep up with the amount of data that is written to a socket - the network connection simply might be too slow. Node will internally queue up the data written to a socket and send it out over the wire when it is possible. (Internally it is polling on the socket's file descriptor for being writable).

The consequence of this internal buffering is that memory may grow. This property shows the number of characters currently buffered to be written. (Number of characters is approximately equal to the number of bytes to be written, but the buffer may contain strings, and the strings are lazily encoded, so the exact number of bytes is not known.)

Users who experience large or growing `bufferSize` should attempt to "throttle" the data flows in their program with `pause()` and `resume()`.

### socket.setEncoding([encoding])

Set the encoding for the socket as a Readable Stream. See stream.setEncoding() for more information.

### socket.write(data, [encoding], [callback])

Sends data on the socket. The second parameter specifies the encoding in the case of a string–it defaults to UTF8 encoding.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is again free.

The optional `callback` parameter will be executed when the data is finally written out - this may not be immediately.

### socket.end([data], [encoding])

Half-closes the socket. i.e., it sends a FIN packet. It is possible the server will still send some data.

If `data` is specified, it is equivalent to calling `socket.write(data, encoding)` followed by `socket.end()`.

### socket.destroy()

Ensures that no more I/O activity happens on this socket. Only necessary in case of errors (parse error or so).

### socket.pause()

Pauses the reading of data. That is, `'data'` events will not be emitted. Useful to throttle back an upload.

### socket.resume()

Resumes reading after a call to `pause()`.

### socket.setTimeout(timeout, [callback])

Sets the socket to timeout after `timeout` milliseconds of inactivity on the socket. By default `net.Socket` do not have a timeout.

When an idle timeout is triggered the socket will receive a `'timeout'` event but the connection will not be severed. The user must manually `end()` or `destroy()` the socket.

If `timeout` is 0, then the existing idle timeout is disabled.

The optional `callback` parameter will be added as a one time listener for the `'timeout'` event.

### socket.setNoDelay([noDelay])

Disables the Nagle algorithm. By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting `true` for `noDelay` will immediately fire off data each time `socket.write()` is called. `noDelay` defaults to `true`.

### socket.setKeepAlive([enable], [initialDelay])

Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket. `enable` defaults to `false`.

Set `initialDelay` (in milliseconds) to set the delay between the last data packet received and the first keepalive probe. Setting 0 for initialDelay will leave the value unchanged from the default (or previous) setting. Defaults to `0`.

### socket.address()

Returns the bound address, the address family name and port of the socket as reported by the operating system. Returns an object with three properties, e.g. `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

### socket.remoteAddress

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.

### socket.remotePort

The numeric representation of the remote port. For example, `80` or `21`.

### socket.bytesRead

The amount of received bytes.

### socket.bytesWritten

The amount of bytes sent.

`net.Socket` instances are EventEmitter with the following events:

### Event: 'connect'

Emitted when a socket connection is successfully established. See `connect()`.

#### Event: 'data'

- {Buffer object}

Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by `socket.setEncoding()`. (See the Readable Stream section for more information.)

Note that the **data will be lost** if there is no listener when a `Socket` emits a `'data'` event.

#### Event: 'end'

Emitted when the other end of the socket sends a FIN packet.

By default (`allowHalfOpen == false`) the socket will destroy its file descriptor once it has written out its pending write queue. However, by setting `allowHalfOpen == true` the socket will not automatically `end()` its side allowing the user to write arbitrary amounts of data, with the caveat that the user is required to `end()` their side now.

#### Event: 'timeout'

Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.

See also: `socket.setTimeout()`

#### Event: 'drain'

Emitted when the write buffer becomes empty. Can be used to throttle uploads.

See also: the return values of `socket.write()`

#### Event: 'error'

- {Error object}

Emitted when an error occurs. The `'close'` event will be called directly following this event.

#### Event: 'close'

- `had_error` {Boolean} true if the socket had a transmission error

Emitted once the socket is fully closed. The argument `had_error` is a boolean which says if the socket was closed due to a transmission error.

### net.isIP(input)

Tests if input is an IP address. Returns 0 for invalid strings, returns 4 for IP version 4 addresses, and returns 6 for IP version 6 addresses.

### net.isIPv4(input)

Returns true if input is a version 4 IP address, otherwise returns false.

### net.isIPv6(input)

Returns true if input is a version 6 IP address, otherwise returns false.

# UDP / Datagram Sockets

```
Stability: 3 - Stable
```

Datagram sockets are available through `require('dgram')`.

## dgram.createSocket(type, [callback])

- `type` String. Either 'udp4' or 'udp6'
- `callback` Function. Attached as a listener to `message` events. Optional
- Returns: Socket object

Creates a datagram Socket of the specified types. Valid types are `udp4` and `udp6`.

Takes an optional callback which is added as a listener for `message` events.

Call `socket.bind` if you want to receive datagrams. `socket.bind()` will bind to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). You can then retrieve the address and port with `socket.address().address` and `socket.address().port`.

## Class: Socket

The dgram Socket class encapsulates the datagram functionality. It should be created via `dgram.createSocket(type, [callback])`.

### Event: 'message'

- `msg` Buffer object. The message
- `rinfo` Object. Remote address information

Emitted when a new datagram is available on a socket. `msg` is a `Buffer` and `rinfo` is an object with the sender's address information and the number of bytes in the datagram.

### Event: 'listening'

Emitted when a socket starts listening for datagrams. This happens as soon as UDP sockets are created.

### Event: 'close'

Emitted when a socket is closed with `close()`. No new `message` events will be emitted on this socket.

### Event: 'error'

- `exception` Error object

Emitted when an error occurs.

### dgram.send(buf, offset, length, port, address, [callback])

- `buf` Buffer object. Message to be sent
- `offset` Integer. Offset in the buffer where the message starts.
- `length` Integer. Number of bytes in the message.
- `port` Integer. destination port
- `address` String. destination IP
- `callback` Function. Callback when message is done being delivered. Optional.

For UDP sockets, the destination port and IP address must be specified. A string may be supplied for the `address` parameter, and it will be resolved with DNS. An optional callback may be specified to detect any DNS errors and when `buf` may be re-used. Note that DNS lookups will delay the time that a send takes place, at least until the next tick. The only way to know for sure that a send has taken place is to use the callback.

If the socket has not been previously bound with a call to `bind`, it's assigned a random port number and bound to the "all interfaces" address (0.0.0.0 for `udp4` sockets, ::0 for `udp6` sockets).

Example of sending a UDP packet to a random port on `localhost`;

```
var dgram = require('dgram');
var message = new Buffer("Some bytes");
var client = dgram.createSocket("udp4");
client.send(message, 0, message.length, 41234, "localhost", function(err, bytes) {
  client.close();
});
```

### A Note about UDP datagram size

The maximum size of an `IPv4/v6` datagram depends on the `MTU` (*Maximum Transmission Unit*) and on the `Payload Length` field size.

- The `Payload Length` field is `16 bits` wide, which means that a normal payload cannot be larger than 64K octets including internet header and data (65,507 bytes = 65,535 âĹŠ 8 bytes UDP header âĹŠ 20 bytes IP header); this is generally true for loopback interfaces, but such long datagrams are impractical for most hosts and networks.
- The `MTU` is the largest size a given link layer technology can support for datagrams. For any link, `IPv4` mandates a minimum `MTU` of `68` octets, while the recommended `MTU` for IPv4 is `576` (typically recommended as the `MTU` for dial-up type applications), whether they arrive whole or in fragments.

For `IPv6`, the minimum `MTU` is `1280` octets, however, the mandatory minimum fragment reassembly buffer size is `1500` octets. The value of `68` octets is very small, since most current link layer technologies have a minimum `MTU` of `1500` (like Ethernet).

Note that it's impossible to know in advance the MTU of each link through which a packet might travel, and that generally sending a datagram greater than the (receiver) `MTU` won't work (the packet gets silently dropped, without informing the source that the data did not reach its intended recipient).

### dgram.bind(port, [address])

- `port` Integer
- `address` String, Optional

For UDP sockets, listen for datagrams on a named `port` and optional `address`. If `address` is not specified, the OS will try to listen on all addresses.

Example of a UDP server listening on port 41234:

```
var dgram = require("dgram");

var server = dgram.createSocket("udp4");

server.on("message", function (msg, rinfo) {
  console.log("server got: " + msg + " from " +
    rinfo.address + ":" + rinfo.port);
});

server.on("listening", function () {
  var address = server.address();
  console.log("server listening " +
      address.address + ":" + address.port);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

#### dgram.close()

Close the underlying socket and stop listening for data on it.

#### dgram.address()

Returns an object containing the address information for a socket. For UDP sockets, this object will contain `address` , `family` and `port`.

#### dgram.setBroadcast(flag)

- `flag` Boolean

Sets or clears the `SO_BROADCAST` socket option. When this option is set, UDP packets may be sent to a local interface's broadcast address.

#### dgram.setTTL(ttl)

- `ttl` Integer

Sets the `IP_TTL` socket option. TTL stands for "Time to Live," but in this context it specifies the number of IP hops that a packet is allowed to go through. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded. Changing TTL values is typically done for network probes or when multicasting.

The argument to `setTTL()` is a number of hops between 1 and 255. The default on most systems is 64.

#### dgram.setMulticastTTL(ttl)

- `ttl` Integer

Sets the `IP_MULTICAST_TTL` socket option. TTL stands for "Time to Live," but in this context it specifies the number of IP hops that a packet is allowed to go through, specifically for multicast traffic. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded.

The argument to `setMulticastTTL()` is a number of hops between 0 and 255. The default on most systems is 1.

#### dgram.setMulticastLoopback(flag)

- `flag` Boolean

Sets or clears the `IP_MULTICAST_LOOP` socket option. When this option is set, multicast packets will also be received on the local interface.

#### dgram.addMembership(multicastAddress, [multicastInterface])

- `multicastAddress` String
- `multicastInterface` String, Optional

Tells the kernel to join a multicast group with `IP_ADD_MEMBERSHIP` socket option.

If `multicastInterface` is not specified, the OS will try to add membership to all valid interfaces.

**dgram.dropMembership(multicastAddress, [multicastInterface])**

- `multicastAddress` String
- `multicastInterface` String, Optional

Opposite of `addMembership` - tells the kernel to leave a multicast group with `IP_DROP_MEMBERSHIP` socket option. This is automatically called by the kernel when the socket is closed or process terminates, so most apps will never need to call this.

If `multicastInterface` is not specified, the OS will try to drop membership to all valid interfaces.

# DNS

```
Stability: 3 - Stable
```

Use `require('dns')` to access this module. All methods in the dns module use C-Ares except for `dns.lookup` which uses `getaddrinfo(3)` in a thread pool. C-Ares is much faster than `getaddrinfo` but the system resolver is more constant with how other programs operate. When a user does `net.connect(80, 'google.com')` or `http.get({ host: 'google.com' })` the `dns.lookup` method is used. Users who need to do a large number of look ups quickly should use the methods that go through C-Ares.

Here is an example which resolves `'www.google.com'` then reverse resolves the IP addresses which are returned.

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

dns.resolve4('www.google.com', function (err, addresses) {
  if (err) throw err;

  console.log('addresses: ' + JSON.stringify(addresses));

  addresses.forEach(function (a) {
    dns.reverse(a, function (err, domains) {
      if (err) {
        throw err;
      }

      console.log('reverse for ' + a + ': ' + JSON.stringify(domains));
    });
  });
});
```

## dns.lookup(domain, [family], callback)

Resolves a domain (e.g. `'google.com'`) into the first found A (IPv4) or AAAA (IPv6) record. The `family` can be the integer `4` or `6`. Defaults to `null` that indicates both Ip v4 and v6 address family.

The callback has arguments `(err, address, family)`. The `address` argument is a string representation of a IP v4 or v6 address. The `family` argument is either the integer 4 or 6 and denotes the family of `address` (not necessarily the value initially passed to `lookup`).

On error, `err` is an `Error` object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOENT'` not only when the domain does not exist but also when the lookup fails in other ways such as no available file descriptors.

## dns.resolve(domain, [rrtype], callback)

Resolves a domain (e.g. `'google.com'`) into an array of the record types specified by rrtype. Valid rrtypes are `'A'` (IPV4 addresses, default), `'AAAA'` (IPV6 addresses), `'MX'` (mail exchange records), `'TXT'` (text records), `'SRV'` (SRV records), `'PTR'` (used for reverse IP lookups), `'NS'` (name server records) and `'CNAME'` (canonical name records).

The callback has arguments `(err, addresses)`. The type of each item in `addresses` is determined by the record type, and described in the documentation for the corresponding lookup methods below.

On error, `err` is an `Error` object, where `err.code` is one of the error codes listed below.

## dns.resolve4(domain, callback)

The same as `dns.resolve()`, but only for IPv4 queries (`A` records). `addresses` is an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).

### dns.resolve6(domain, callback)

The same as `dns.resolve4()` except for IPv6 queries (an `AAAA` query).

### dns.resolveMx(domain, callback)

The same as `dns.resolve()`, but only for mail exchange queries (`MX` records).

`addresses` is an array of MX records, each with a priority and an exchange attribute (e.g. `[{'priority': 10, 'exchange': 'mx.example.com'},...]`).

### dns.resolveTxt(domain, callback)

The same as `dns.resolve()`, but only for text queries (`TXT` records). `addresses` is an array of the text records available for `domain` (e.g., `['v=spf1 ip4:0.0.0.0 ~all']`).

### dns.resolveSrv(domain, callback)

The same as `dns.resolve()`, but only for service records (`SRV` records). `addresses` is an array of the SRV records available for `domain`. Properties of SRV records are priority, weight, port, and name (e.g., `[{'priority': 10, {'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]`).

### dns.resolveNs(domain, callback)

The same as `dns.resolve()`, but only for name server records (`NS` records). `addresses` is an array of the name server records available for `domain` (e.g., `['ns1.example.com', 'ns2.example.com']`).

### dns.resolveCname(domain, callback)

The same as `dns.resolve()`, but only for canonical name records (`CNAME` records). `addresses` is an array of the canonical name records available for `domain` (e.g., `['bar.example.com']`).

### dns.reverse(ip, callback)

Reverse resolves an ip address to an array of domain names.

The callback has arguments `(err, domains)`.

On error, `err` is an `Error` object, where `err.code` is one of the error codes listed below.

### Error codes

Each DNS query can return one of the following error codes:

- `dns.NODATA`: DNS server returned answer with no data.
- `dns.FORMERR`: DNS server claims query was misformatted.
- `dns.SERVFAIL`: DNS server returned general failure.
- `dns.NOTFOUND`: Domain name not found.
- `dns.NOTIMP`: DNS server does not implement requested operation.
- `dns.REFUSED`: DNS server refused query.
- `dns.BADQUERY`: Misformatted DNS query.
- `dns.BADNAME`: Misformatted domain name.
- `dns.BADFAMILY`: Unsupported address family.

- `dns.BADRESP`: Misformatted DNS reply.
- `dns.CONNREFUSED`: Could not contact DNS servers.
- `dns.TIMEOUT`: Timeout while contacting DNS servers.
- `dns.EOF`: End of file.
- `dns.FILE`: Error reading file.
- `dns.NOMEM`: Out of memory.
- `dns.DESTRUCTION`: Channel is being destroyed.
- `dns.BADSTR`: Misformatted string.
- `dns.BADFLAGS`: Illegal flags specified.
- `dns.NONAME`: Given hostname is not numeric.
- `dns.BADHINTS`: Illegal hints flags specified.
- `dns.NOTINITIALIZED`: c-ares library initialization not yet performed.
- `dns.LOADIPHLPAPI`: Error loading iphlpapi.dll.
- `dns.ADDRGETNETWORKPARAMS`: Could not find GetNetworkParams function.
- `dns.CANCELLED`: DNS query cancelled.

# HTTP

```
Stability: 3 - Stable
```

To use the HTTP server and client one must `require('http')`.

The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses–the user is able to stream data.

HTTP message headers are represented by an object like this:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

Keys are lowercased. Values are not modified.

In order to support the full spectrum of possible HTTP applications, Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.

## http.STATUS_CODES

- {Object}

A collection of all the standard HTTP response status codes, and the short description of each. For example, `http.STATUS_CODES[404] === 'Not Found'`.

## http.createServer([requestListener])

Returns a new web server object.

The `requestListener` is a function which is automatically added to the `'request'` event.

## http.createClient([port], [host])

This function is **deprecated**; please use http.request() instead. Constructs a new HTTP client. `port` and `host` refer to the server to be connected to.

## Class: http.Server

This is an EventEmitter with the following events:

### Event: 'request'

```
function (request, response) { }
```

Emitted each time there is a request. Note that there may be multiple requests per connection (in the case of keep-alive connections). `request` is an instance of `http.ServerRequest` and `response` is an instance of `http.ServerResponse`

### Event: 'connection'

```
function (socket) { }
```

When a new TCP stream is established. `socket` is an object of type `net.Socket`. Usually users will not want to access this event. The `socket` can also be accessed at `request.connection`.

#### Event: 'close'

```
function () { }
```

Emitted when the server closes.

#### Event: 'checkContinue'

```
function (request, response) { }
```

Emitted each time a request with an http Expect: 100-continue is received. If this event isn't listened for, the server will automatically respond with a 100 Continue as appropriate.

Handling this event involves calling `response.writeContinue` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g., 400 Bad Request) if the client should not continue to send the request body.

Note that when this event is emitted and handled, the `request` event will not be emitted.

#### Event: 'connect'

```
function (request, socket, head) { }
```

Emitted each time a client requests a http CONNECT method. If this event isn't listened for, then clients requesting a CONNECT method will have their connections closed.

- `request` is the arguments for the http request, as it is in the request event.
- `socket` is the network socket between the server and client.
- `head` is an instance of Buffer, the first packet of the tunneling stream, this may be empty.

After this event is emitted, the request's socket will not have a `data` event listener, meaning you will need to bind to it in order to handle data sent to the server on that socket.

#### Event: 'upgrade'

```
function (request, socket, head) { }
```

Emitted each time a client requests a http upgrade. If this event isn't listened for, then clients requesting an upgrade will have their connections closed.

- `request` is the arguments for the http request, as it is in the request event.
- `socket` is the network socket between the server and client.
- `head` is an instance of Buffer, the first packet of the upgraded stream, this may be empty.

After this event is emitted, the request's socket will not have a `data` event listener, meaning you will need to bind to it in order to handle data sent to the server on that socket.

#### Event: 'clientError'

```
function (exception) { }
```

If a client connection emits an 'error' event - it will forwarded here.

### server.listen(port, [hostname], [backlog], [callback])

Begin accepting connections on the specified port and hostname. If the hostname is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`).

To listen to a unix socket, supply a filename instead of port and hostname.

Backlog is the maximum length of the queue of pending connections. The actual length will be determined by your OS through sysctl settings such as `tcp_max_syn_backlog` and `somaxconn` on linux. The default value of this parameter is 511 (not 512).

This function is asynchronous. The last parameter `callback` will be added as a listener for the 'listening' event. See also net.Server.listen(port).

#### server.listen(path, [callback])

Start a UNIX socket server listening for connections on the given `path`.

This function is asynchronous. The last parameter `callback` will be added as a listener for the 'listening' event. See also net.Server.listen(path).

#### server.listen(handle, [listeningListener])

- `handle` {Object}
- `listeningListener` {Function}

The `handle` object can be set to either a server or socket (anything with an underlying `_handle` member), or a `{fd: <n>}` object.

This will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket.

Listening on a file descriptor is not supported on Windows.

This function is asynchronous. The last parameter `callback` will be added as a listener for the 'listening' event. See also net.Server.listen().

#### server.close([cb])

Stops the server from accepting new connections. See net.Server.close().

#### server.maxHeadersCount

Limits maximum incoming headers count, equal to 1000 by default. If set to 0 - no limit will be applied.

### Class: http.ServerRequest

This object is created internally by a HTTP server – not by the user – and passed as the first argument to a `'request'` listener.

The request implements the Readable Stream interface. This is an EventEmitter with the following events:

#### Event: 'data'

```
function (chunk) { }
```

Emitted when a piece of the message body is received. The chunk is a string if an encoding has been set with `request.setEncoding()`, otherwise it's a Buffer.

Note that the **data will be lost** if there is no listener when a `ServerRequest` emits a `'data'` event.

#### Event: 'end'

```
function () { }
```

Emitted exactly once for each request. After that, no more `'data'` events will be emitted on the request.

#### Event: 'close'

```
function () { }
```

Indicates that the underlaying connection was terminated before `response.end()` was called or able to flush.

Just like `'end'`, this event occurs only once per request, and no more `'data'` events will fire afterwards.

Note: `'close'` can fire after `'end'`, but not vice versa.

### request.method

The request method as a string. Read only. Example: `'GET'`, `'DELETE'`.

### request.url

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

Then `request.url` will be:

```
'/status?name=ryan'
```

If you would like to parse the URL into its parts, you can use `require('url').parse(request.url)`. Example:

```
node> require('url').parse('/status?name=ryan')
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status' }
```

If you would like to extract the params from the query string, you can use the `require('querystring').parse` function, or pass `true` as the second argument to `require('url').parse`. Example:

```
node> require('url').parse('/status?name=ryan', true)
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: { name: 'ryan' },
  pathname: '/status' }
```

### request.headers

Read only.

### request.trailers

Read only; HTTP trailers (if present). Only populated after the 'end' event.

### request.httpVersion

The HTTP protocol version as a string. Read only. Examples: `'1.1'`, `'1.0'`. Also `request.httpVersionMajor` is the first integer and `request.httpVersionMinor` is the second.

### request.setEncoding([encoding])

Set the encoding for the request body. See stream.setEncoding() for more information.

### request.pause()

Pauses request from emitting events. Useful to throttle back an upload.

### request.resume()

Resumes a paused request.

#### request.connection

The `net.Socket` object associated with the connection.

With HTTPS support, use request.connection.verifyPeer() and request.connection.getPeerCertificate() to obtain the client's authentication details.

### Class: http.ServerResponse

This object is created internally by a HTTP server–not by the user. It is passed as the second parameter to the `'request'` event.

The response implements the Writable Stream interface. This is an EventEmitter with the following events:

#### Event: 'close'

```
function () { }
```

Indicates that the underlaying connection was terminated before `response.end()` was called or able to flush.

#### response.writeContinue()

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the 'checkContinue' event on `Server`.

#### response.writeHead(statusCode, [reasonPhrase], [headers])

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, are the response headers. Optionally one can give a human-readable `reasonPhrase` as the second argument.

Example:

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain' });
```

This method must only be called once on a message and it must be called before `response.end()` is called.

If you call `response.write()` or `response.end()` before calling this, the implicit/mutable headers will be calculated and call this function for you.

Note: that Content-Length is given in bytes not characters. The above example works because the string `'hello world'` contains only single byte characters. If the body contains higher coded characters then `Buffer.byteLength()` should be used to determine the number of bytes in a given encoding. And Node does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

#### response.statusCode

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be send to the client when the headers get flushed.

Example:

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

### response.setHeader(name, value)

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here if you need to send multiple headers with the same name.

Example:

```
response.setHeader("Content-Type", "text/html");
```

or

```
response.setHeader("Set-Cookie", ["type=ninja", "language=javascript"]);
```

### response.sendDate

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

### response.getHeader(name)

Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive. This can only be called before headers get implicitly flushed.

Example:

```
var contentType = response.getHeader('content-type');
```

### response.removeHeader(name)

Removes a header that's queued for implicit sending.

Example:

```
response.removeHeader("Content-Encoding");
```

### response.write(chunk, [encoding])

If this method is called and `response.writeHead()` has not been called, it will switch to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`.

**Note**: This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time `response.write()` is called, it will send the buffered header information and the first body to the client. The second time `response.write()` is called, Node assumes you're going to be streaming data, and sends that separately. That is, the response is buffered up to the first chunk of body.

### response.addTrailers(headers)

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Trailers will **only** be emitted if chunked encoding is used for the response; if it is not (e.g., if the request was HTTP/1.0), they will be silently discarded.

Note that HTTP requires the `Trailer` header to be sent if you intend to emit trailers, with a list of the header fields in its value. E.g.,

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': "7895bf4b8828b55ceaf47747b4bca667"});
response.end();
```

#### response.end([data], [encoding])

This method signals to the server that all of the response headers and body has been sent; that server should consider this message complete. The method, `response.end()`, MUST be called on each response.

If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `response.end()`.

### http.request(options, callback)

Node maintains several connections per server to make HTTP requests. This function allows one to transparently issue requests.

`options` can be an object or a string. If `options` is a string, it is automatically parsed with url.parse().

Options:

- `host`: A domain name or IP address of the server to issue the request to. Defaults to `'localhost'`.
- `hostname`: To support `url.parse()` `hostname` is preferred over `host`
- `port`: Port of remote server. Defaults to 80.
- `localAddress`: Local interface to bind for network connections.
- `socketPath`: Unix Domain Socket (use one of host:port or socketPath)
- `method`: A string specifying the HTTP request method. Defaults to `'GET'`.
- `path`: Request path. Defaults to `'/'`. Should include query string if any. E.G. `'/index.html?page=12'`
- `headers`: An object containing request headers.
- `auth`: Basic authentication i.e. `'user:password'` to compute an Authorization header.
- `agent`: Controls Agent behavior. When an Agent is used request will default to `Connection: keep-alive`. Possible values:
- `undefined` (default): use global Agent for this host and port.
- `Agent` object: explicitly use the passed in `Agent`.
- `false`: opts out of connection pooling with an Agent, defaults request to `Connection: close`.

`http.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.

Example:

```
var options = {
  host: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST'
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
```

```
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// write data to request body
req.write('data\n');
req.write('data\n');
req.end();
```

Note that in the example `req.end()` was called. With `http.request()` one must always call `req.end()` to signify that you're done with the request - even if there is no data being written to the request body.

If any error is encountered during the request (be that with DNS resolution, TCP level errors, or actual HTTP parse errors) an `'error'` event is emitted on the returned request object.

There are a few special headers that should be noted.

- Sending a 'Connection: keep-alive' will notify Node that the connection to the server should be persisted until the next request.
- Sending a 'Content-length' header will disable the default chunked encoding.
- Sending an 'Expect' header will immediately send the request headers. Usually, when sending 'Expect: 100-continue', you should both set a timeout and listen for the `continue` event. See RFC2616 Section 8.2.3 for more information.
- Sending an Authorization header will override using the `auth` option to compute basic authentication.

### http.get(options, callback)

Since most requests are GET requests without bodies, Node provides this convenience method. The only difference between this method and `http.request()` is that it sets the method to GET and calls `req.end()` automatically.

Example:

```
http.get("http://www.google.com/index.html", function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

### Class: http.Agent

In node 0.5.3+ there is a new implementation of the HTTP Agent which is used for pooling sockets used in HTTP client requests.

Previously, a single agent instance helped pool for a single host+port. The current implementation now holds sockets for any number of hosts.

The current HTTP Agent also defaults client requests to using Connection:keep-alive. If no pending HTTP requests are waiting on a socket to become free the socket is closed. This means that node's pool has the benefit of keep-alive when under load but still does not require developers to manually close the HTTP clients using keep-alive.

Sockets are removed from the agent's pool when the socket emits either a "close" event or a special "agentRemove" event. This means that if you intend to keep one HTTP request open for a long time and don't want it to stay in the pool you can do something along the lines of:

```
http.get(options, function(res) {
  // Do stuff
}).on("socket", function (socket) {
  socket.emit("agentRemove");
});
```

Alternatively, you could just opt out of pooling entirely using `agent:false`:

```
http.get({host:'localhost', port:80, path:'/', agent:false}, function (res) {
  // Do stuff
})
```

#### agent.maxSockets

By default set to 5. Determines how many concurrent sockets the agent can have open per host.

#### agent.sockets

An object which contains arrays of sockets currently in use by the Agent. Do not modify.

#### agent.requests

An object which contains queues of requests that have not yet been assigned to sockets. Do not modify.

### http.globalAgent

Global instance of Agent which is used as the default for all http client requests.

### Class: http.ClientRequest

This object is created internally and returned from `http.request()`. It represents an *in-progress* request whose header has already been queued. The header is still mutable using the `setHeader(name, value)`, `getHeader(name)`, `removeHeader(name)` API. The actual header will be sent along with the first data chunk or when closing the connection.

To get the response, add a listener for `'response'` to the request object. `'response'` will be emitted from the request object when the response headers have been received. The `'response'` event is executed with one argument which is an instance of `http.ClientResponse`.

During the `'response'` event, one can add listeners to the response object; particularly to listen for the `'data'` event. Note that the `'response'` event is called before any part of the response body is received, so there is no need to worry about racing to catch the first part of the body. As long as a listener for `'data'` is added during the `'response'` event, the entire body will be caught.

```
// Good
request.on('response', function (response) {
  response.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

// Bad - misses all or part of the body
request.on('response', function (response) {
  setTimeout(function () {
    response.on('data', function (chunk) {
      console.log('BODY: ' + chunk);
    });
  }, 10);
});
```

Note: Node does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

The request implements the Writable Stream interface. This is an EventEmitter with the following events:

### Event 'response'

`function (response) { }`

Emitted when a response is received to this request. This event is emitted only once. The `response` argument will be an instance of `http.ClientResponse`.

Options:

- `host`: A domain name or IP address of the server to issue the request to.
- `port`: Port of remote server.
- `socketPath`: Unix Domain Socket (use one of host:port or socketPath)

### Event: 'socket'

`function (socket) { }`

Emitted after a socket is assigned to this request.

### Event: 'connect'

`function (response, socket, head) { }`

Emitted each time a server responds to a request with a CONNECT method. If this event isn't being listened for, clients receiving a CONNECT method will have their connections closed.

A client server pair that show you how to listen for the `connect` event.

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

// Create an HTTP tunneling proxy
var proxy = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
proxy.on('connect', function(req, cltSocket, head) {
  // connect to an origin server
  var srvUrl = url.parse('http://' + req.url);
  var srvSocket = net.connect(srvUrl.port, srvUrl.hostname, function() {
    cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
                    'Proxy-agent: Node-Proxy\r\n' +
                    '\r\n');
    srvSocket.write(head);
    srvSocket.pipe(cltSocket);
    cltSocket.pipe(srvSocket);
  });
});

// now that proxy is running
proxy.listen(1337, '127.0.0.1', function() {

  // make a request to a tunneling proxy
  var options = {
    port: 1337,
    host: '127.0.0.1',
    method: 'CONNECT',
    path: 'www.google.com:80'
  };

  var req = http.request(options);
  req.end();
```

```
  req.on('connect', function(res, socket, head) {
    console.log('got connected!');

    // make a request over an HTTP tunnel
    socket.write('GET / HTTP/1.1\r\n' +
                 'Host: www.google.com:80\r\n' +
                 'Connection: close\r\n' +
                 '\r\n');
    socket.on('data', function(chunk) {
      console.log(chunk.toString());
    });
    socket.on('end', function() {
      proxy.close();
    });
  });
});
```

#### Event: 'upgrade'

`function (response, socket, head) { }`

Emitted each time a server responds to a request with an upgrade. If this event isn't being listened for, clients receiving an upgrade header will have their connections closed.

A client server pair that show you how to listen for the `upgrade` event.

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

// Create an HTTP server
var srv = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
srv.on('upgrade', function(req, socket, head) {
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
               'Upgrade: WebSocket\r\n' +
               'Connection: Upgrade\r\n' +
               '\r\n');

  socket.pipe(socket); // echo back
});

// now that server is running
srv.listen(1337, '127.0.0.1', function() {

  // make a request
  var options = {
    port: 1337,
    host: '127.0.0.1',
    headers: {
      'Connection': 'Upgrade',
      'Upgrade': 'websocket'
    }
  };

  var req = http.request(options);
  req.end();

  req.on('upgrade', function(res, socket, upgradeHead) {
    console.log('got upgraded!');
    socket.end();
```

```
    process.exit(0);
  });
});
```

#### Event: 'continue'

```
function () { }
```

Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

### request.write(chunk, [encoding])

Sends a chunk of the body. By calling this method many times, the user can stream a request body to a server–in that case it is suggested to use the `['Transfer-Encoding', 'chunked']` header line when creating the request.

The `chunk` argument should be a Buffer or a string.

The `encoding` argument is optional and only applies when `chunk` is a string. Defaults to `'utf8'`.

### request.end([data], [encoding])

Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream. If the request is chunked, this will send the terminating `'0\r\n\r\n'`.

If `data` is specified, it is equivalent to calling `request.write(data, encoding)` followed by `request.end()`.

### request.abort()

Aborts a request. (New since v0.3.8.)

### request.setTimeout(timeout, [callback])

Once a socket is assigned to this request and is connected socket.setTimeout() will be called.

### request.setNoDelay([noDelay])

Once a socket is assigned to this request and is connected socket.setNoDelay() will be called.

### request.setSocketKeepAlive([enable], [initialDelay])

Once a socket is assigned to this request and is connected socket.setKeepAlive() will be called.

## http.ClientResponse

This object is created when making a request with `http.request()`. It is passed to the `'response'` event of the request object.

The response implements the Readable Stream interface. This is an EventEmitter with the following events:

#### Event: 'data'

```
function (chunk) { }
```

Emitted when a piece of the message body is received.

Note that the **data will be lost** if there is no listener when a `ClientResponse` emits a `'data'` event.

#### Event: 'end'

```
function () { }
```

Emitted exactly once for each message. No arguments. After emitted no other events will be emitted on the response.

#### Event: 'close'

```
function (err) { }
```

Indicates that the underlaying connection was terminated before `end` event was emitted. See http.ServerRequest's `'close'` event for more information.

#### response.statusCode

The 3-digit HTTP response status code. E.G. `404`.

#### response.httpVersion

The HTTP version of the connected-to server. Probably either `'1.1'` or `'1.0'`. Also `response.httpVersionMajor` is the first integer and `response.httpVersionMinor` is the second.

#### response.headers

The response headers object.

#### response.trailers

The response trailers object. Only populated after the 'end' event.

#### response.setEncoding([encoding])

Set the encoding for the response body. See stream.setEncoding() for more information.

#### response.pause()

Pauses response from emitting events. Useful to throttle back a download.

#### response.resume()

Resumes a paused response.

# HTTPS

```
Stability: 3 - Stable
```

HTTPS is the HTTP protocol over TLS/SSL. In Node this is implemented as a separate module.

## Class: https.Server

This class is a subclass of `tls.Server` and emits events same as `http.Server`. See `http.Server` for more information.

## https.createServer(options, [requestListener])

Returns a new HTTPS web server object. The `options` is similar to tls.createServer(). The `requestListener` is a function which is automatically added to the `'request'` event.

Example:

```
// curl -k https://localhost:8000/
var https = require('https');
var fs = require('fs');

var options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

Or

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

var options = {
  pfx: fs.readFileSync('server.pfx')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

## https.request(options, callback)

Makes a request to a secure web server. All options from http.request() are valid.

Example:

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

var options = {
  host: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET'
};
```

```
var req = https.request(options, function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

The options argument has the following options

- host: IP or domain of host to make request to. Defaults to `'localhost'`.
- port: port of host to request to. Defaults to 443.
- path: Path to request. Default `'/'`.
- method: HTTP request method. Default `'GET'`.
- `host`: A domain name or IP address of the server to issue the request to. Defaults to `'localhost'`.
- `hostname`: To support `url.parse()` `hostname` is preferred over `host`
- `port`: Port of remote server. Defaults to 443.
- `method`: A string specifying the HTTP request method. Defaults to `'GET'`.
- `path`: Request path. Defaults to `'/'`. Should include query string if any. E.G. `'/index.html?page=12'`
- `headers`: An object containing request headers.
- `auth`: Basic authentication i.e. `'user:password'` to compute an Authorization header.
- `agent`: Controls Agent behavior. When an Agent is used request will default to `Connection: keep-alive`. Possible values:
- `undefined` (default): use globalAgent for this host and port.
- `Agent` object: explicitly use the passed in `Agent`.
- `false`: opts out of connection pooling with an Agent, defaults request to `Connection: close`.

The following options from tls.connect() can also be specified. However, a globalAgent silently ignores these.

- `pfx`: Certificate, Private key and CA certificates to use for SSL. Default `null`.
- `key`: Private key to use for SSL. Default `null`.
- `passphrase`: A string of passphrase for the private key or pfx. Default `null`.
- `cert`: Public x509 certificate to use. Default `null`.
- `ca`: An authority certificate or array of authority certificates to check the remote host against.
- `ciphers`: A string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format.
- `rejectUnauthorized`: If `true`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails. Verification happens at the connection level, *before* the HTTP request is sent. Default `false`.

In order to specify these options, use a custom `Agent`.

Example:

```
var options = {
  host: 'encrypted.google.com',
  port: 443,
  path: '/',
```

```
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};
options.agent = new https.Agent(options);

var req = https.request(options, function(res) {
  ...
}
```

Or does not use an `Agent`.

Example:

```
var options = {
  host: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

var req = https.request(options, function(res) {
  ...
}
```

### https.get(options, callback)

Like `http.get()` but for HTTPS.

Example:

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

https.get({ host: 'encrypted.google.com', path: '/' }, function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });

}).on('error', function(e) {
  console.error(e);
});
```

### Class: https.Agent

An Agent object for HTTPS similar to http.Agent. See [https.request()][] for more information.

### https.globalAgent

Global instance of https.Agent for all HTTPS client requests.

# URL

Stability: 3 - Stable

This module has utilities for URL resolution and parsing. Call `require('url')` to use it.

Parsed URL objects have some or all of the following fields, depending on whether or not they exist in the URL string. Any parts that are not in the URL string will not be in the parsed object. Examples are shown for the URL

`'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- `href`: The full URL that was originally parsed. Both the protocol and host are lowercased.

  Example: `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- `protocol`: The request protocol, lowercased.

  Example: `'http:'`

- `host`: The full lowercased host portion of the URL, including port information.

  Example: `'host.com:8080'`

- `auth`: The authentication information portion of a URL.

  Example: `'user:pass'`

- `hostname`: Just the lowercased hostname portion of the host.

  Example: `'host.com'`

- `port`: The port number portion of the host.

  Example: `'8080'`

- `pathname`: The path section of the URL, that comes after the host and before the query, including the initial slash if present.

  Example: `'/p/a/t/h'`

- `search`: The 'query string' portion of the URL, including the leading question mark.

  Example: `'?query=string'`

- `path`: Concatenation of `pathname` and `search`.

  Example: `'/p/a/t/h?query=string'`

- `query`: Either the 'params' portion of the query string, or a querystring-parsed object.

  Example: `'query=string'` or `{'query':'string'}`

- `hash`: The 'fragment' portion of the URL including the pound-sign.

  Example: `'#hash'`

The following methods are provided by the URL module:

## url.parse(urlStr, [parseQueryString], [slashesDenoteHost])

Take a URL string, and return an object.

Pass `true` as the second argument to also parse the query string using the `querystring` module. Defaults to `false`.

Pass `true` as the third argument to treat `//foo/bar` as `{ host: 'foo', pathname: '/bar' }` rather than `{ pathname: '//foo/bar' }`. Defaults to `false`.

### url.format(urlObj)

Take a parsed URL object, and return a formatted URL string.

- `href` will be ignored.
- `protocol`is treated the same with or without the trailing `:` (colon).
- The protocols `http`, `https`, `ftp`, `gopher`, `file` will be postfixed with `://` (colon-slash-slash).
- All other protocols `mailto`, `xmpp`, `aim`, `sftp`, `foo`, etc will be postfixed with `:` (colon)
- `auth` will be used if present.
- `hostname` will only be used if `host` is absent.
- `port` will only be used if `host` is absent.
- `host` will be used in place of `hostname` and `port`
- `pathname` is treated the same with or without the leading `/` (slash)
- `search` will be used in place of `query`
- `query` (object; see `querystring`) will only be used if `search` is absent.
- `search` is treated the same with or without the leading `?` (question mark)
- `hash` is treated the same with or without the leading `#` (pound sign, anchor)

### url.resolve(from, to)

Take a base URL, and a href URL, and resolve them as a browser would for an anchor tag.

# Query String

```
Stability: 3 - Stable
```

This module provides utilities for dealing with query strings. It provides the following methods:

## querystring.stringify(obj, [sep], [eq])

Serialize an object to a query string. Optionally override the default separator (`'&'`) and assignment (`'='`) characters.

Example:

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })
// returns
'foo=bar&baz=qux&baz=quux&corge='

querystring.stringify({foo: 'bar', baz: 'qux'}, ';', ':')
// returns
'foo:bar;baz:qux'
```

## querystring.parse(str, [sep], [eq], [options])

Deserialize a query string to an object. Optionally override the default separator (`'&'`) and assignment (`'='`) characters.

Options object may contain `maxKeys` property (equal to 1000 by default), it'll be used to limit processed keys. Set it to 0 to remove key count limitation.

Example:

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge')
// returns
{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }
```

## querystring.escape

The escape function used by `querystring.stringify`, provided so that it could be overridden if necessary.

## querystring.unescape

The unescape function used by `querystring.parse`, provided so that it could be overridden if necessary.

# Readline

```
Stability: 2 - Unstable
```

To use this module, do `require('readline')`. Readline allows reading of a stream (such as `process.stdin`) on a line-by-line basis.

Note that once you've invoked this module, your node program will not terminate until you've closed the interface. Here's how to allow your program to gracefully exit:

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

var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question("What do you think of node.js? ", function(answer) {
  // TODO: Log the answer in a database
  console.log("Thank you for your valuable feedback:", answer);

  rl.close();
});
```

## readline.createInterface(options)

Creates a readline `Interface` instance. Accepts an "options" Object that takes the following values:

- `input` - the readable stream to listen to (Required).
- `output` - the writable stream to write readline data to (Required).
- `completer` - an optional function that is used for Tab autocompletion. See below for an example of using this.
- `terminal` - pass `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. Defaults to checking `isTTY` on the `output` stream upon instantiation.

The `completer` function is given a the current line entered by the user, and is supposed to return an Array with 2 entries:

1. An Array with matching entries for the completion.
2. The substring that was used for the matching.

Which ends up looking something like: `[[substr1, substr2, ...], originalsubstring]`.

Example:

```
function completer(line) {
  var completions = '.help .error .exit .quit .q'.split(' ')
  var hits = completions.filter(function(c) { return c.indexOf(line) == 0 })
  // show all completions if none found
  return [hits.length ? hits : completions, line]
}
```

Also `completer` can be run in async mode if it accepts two arguments:

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```

`createInterface` is commonly used with `process.stdin` and `process.stdout` in order to accept user input:

```
var readline = require('readline');
var rl = readline.createInterface({
  input: process.stdin,
```

```
  output: process.stdout
});
```

Once you have a readline instance, you most commonly listen for the `"line"` event.

If `terminal` is `true` for this instance then the `output` stream will get the best compatibility if it defines an `output.columns` property, and fires a `"resize"` event on the `output` if/when the columns ever change (`process.stdout` does this automatically when it is a TTY).

## Class: Interface

The class that represents a readline interface with an input and output stream.

### rl.setPrompt(prompt, length)

Sets the prompt, for example when you run `node` on the command line, you see `>`, which is node's prompt.

### rl.prompt([preserveCursor])

Readies readline for input from the user, putting the current `setPrompt` options on a new line, giving the user a new spot to write. Set `preserveCursor` to `true` to prevent the cursor placement being reset to `0`.

This will also resume the `input` stream used with `createInterface` if it has been paused.

### rl.question(query, callback)

Prepends the prompt with `query` and invokes `callback` with the user's response. Displays the query to the user, and then invokes `callback` with the user's response after it has been typed.

This will also resume the `input` stream used with `createInterface` if it has been paused.

Example usage:

```
interface.question('What is your favorite food?', function(answer) {
  console.log('Oh, so your favorite food is ' + answer);
});
```

### rl.pause()

Pauses the readline `input` stream, allowing it to be resumed later if needed.

### rl.resume()

Resumes the readline `input` stream.

### rl.close()

Closes the `Interface` instance, relinquishing control on the `input` and `output` streams. The "close" event will also be emitted.

### rl.write(data, [key])

Writes `data` to `output` stream. `key` is an object literal to represent a key sequence; available if the terminal is a TTY.

This will also resume the `input` stream if it has been paused.

Example:

```
rl.write('Delete me!');
// Simulate ctrl+u to delete the line written previously
rl.write(null, {ctrl: true, name: 'u'});
```

## Events

### Event: 'line'

`function (line) {}`

Emitted whenever the `input` stream receives a `\n`, usually received when the user hits enter, or return. This is a good hook to listen for user input.

Example of listening for `line`:

```
rl.on('line', function (cmd) {
  console.log('You just typed: '+cmd);
});
```

### Event: 'pause'

`function () {}`

Emitted whenever the `input` stream is paused.

Also emitted whenever the `input` stream is not paused and receives the `SIGCONT` event. (See events `SIGTSTP` and `SIGCONT`)

Example of listening for `pause`:

```
rl.on('pause', function() {
  console.log('Readline paused.');
});
```

### Event: 'resume'

`function () {}`

Emitted whenever the `input` stream is resumed.

Example of listening for `resume`:

```
rl.on('resume', function() {
  console.log('Readline resumed.');
});
```

### Event: 'close'

`function () {}`

Emitted when `close()` is called.

Also emitted when the `input` stream receives its "end" event. The `Interface` instance should be considered "finished" once this is emitted. For example, when the `input` stream receives `^D`, respectively known as `EOT`.

This event is also called if there is no `SIGINT` event listener present when the `input` stream receives a `^C`, respectively known as `SIGINT`.

#### Event: 'SIGINT'

`function () {}`

Emitted whenever the `input` stream receives a `^C`, respectively known as `SIGINT`. If there is no `SIGINT` event listener present when the `input` stream receives a `SIGINT`, `pause` will be triggered.

Example of listening for `SIGINT`:

```
rl.on('SIGINT', function() {
  rl.question('Are you sure you want to exit?', function(answer) {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

#### Event: 'SIGTSTP'

`function () {}`

**This does not work on Windows.**

Emitted whenever the `input` stream receives a `^Z`, respectively known as `SIGTSTP`. If there is no `SIGTSTP` event listener present when the `input` stream receives a `SIGTSTP`, the program will be sent to the background.

When the program is resumed with `fg`, the `pause` and `SIGCONT` events will be emitted. You can use either to resume the stream.

The `pause` and `SIGCONT` events will not be triggered if the stream was paused before the program was sent to the background.

Example of listening for `SIGTSTP`:

```
rl.on('SIGTSTP', function() {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

#### Event: 'SIGCONT'

`function () {}`

**This does not work on Windows.**

Emitted whenever the `input` stream is sent to the background with `^Z`, respectively known as `SIGTSTP`, and then continued with `fg(1)`. This event only emits if the stream was not paused before sending the program to the background.

Example of listening for `SIGCONT`:

```
rl.on('SIGCONT', function() {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

### Example: Tiny CLI

Here's an example of how to use all these together to craft a tiny command line interface:

```
var readline = require('readline'),
    rl = readline.createInterface(process.stdin, process.stdout);

rl.setPrompt('OHAI> ');
rl.prompt();

rl.on('line', function(line) {
```

```
  switch(line.trim()) {
    case 'hello':
      console.log('world!');
      break;
    default:
      console.log('Say what? I might have heard `' + line.trim() + '`');
      break;
  }
  rl.prompt();
}).on('close', function() {
  console.log('Have a great day!');
  process.exit(0);
});
```

# REPL

A Read-Eval-Print-Loop (REPL) is available both as a standalone program and easily includable in other programs. The REPL provides a way to interactively run JavaScript and see the results. It can be used for debugging, testing, or just trying things out.

By executing `node` without any arguments from the command-line you will be dropped into the REPL. It has simplistic emacs line-editing.

```
mjr:~$ node
Type '.help' for options.
> a = [ 1, 2, 3];
[ 1, 2, 3 ]
> a.forEach(function (v) {
...   console.log(v);
...   });
1
2
3
```

For advanced line-editors, start node with the environmental variable `NODE_NO_READLINE=1`. This will start the main and debugger REPL in canonical terminal settings which will allow you to use with `rlwrap`.

For example, you could add this to your bashrc file:

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

## repl.start(options)

Returns and starts a `REPLServer` instance. Accepts an "options" Object that takes the following values:

- `prompt` - the prompt and `stream` for all I/O. Defaults to `>`.
- `input` - the readable stream to listen to. Defaults to `process.stdin`.
- `output` - the writable stream to write readline data to. Defaults to `process.stdout`.
- `terminal` - pass `true` if the `stream` should be treated like a TTY, and have ANSI/VT100 escape codes written to it. Defaults to checking `isTTY` on the `output` stream upon instantiation.
- `eval` - function that will be used to eval each given line. Defaults to an async wrapper for `eval()`. See below for an example of a custom `eval`.
- `useColors` - a boolean which specifies whether or not the `writer` function should output colors. If a different `writer` function is set then this does nothing. Defaults to the repl's `terminal` value.
- `useGlobal` - if set to `true`, then the repl will use the `global` object, instead of running scripts in a separate context. Defaults to `false`.
- `ignoreUndefined` - if set to `true`, then the repl will not output the return value of command if it's `undefined`. Defaults to `false`.
- `writer` - the function to invoke for each command that gets evaluated which returns the formatting (including coloring) to display. Defaults to `util.inspect`.

You can use your own `eval` function if it has following signature:

```
function eval(cmd, context, filename, callback) {
  callback(null, result);
}
```

Multiple REPLs may be started against the same running instance of node. Each will share the same global object but will have unique I/O.

Here is an example that starts a REPL on stdin, a Unix socket, and a TCP socket:

```
var net = require("net"),
    repl = require("repl");
```

```
connections = 0;

repl.start({
  prompt: "node via stdin> ",
  input: process.stdin,
  output: process.stdout
});

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via Unix socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  })
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via TCP socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  });
}).listen(5001);
```

Running this program from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet` is useful for connecting to TCP sockets, and `socat` can be used to connect to both Unix and TCP sockets.

By starting a REPL from a Unix socket-based server instead of stdin, you can connect to a long-running node process without restarting it.

For an example of running a "full-featured" (`terminal`) REPL over a `net.Server` and `net.Socket` instance, see: https://gist.github.com/2209310

For an example of running a REPL instance over `curl(1)`, see: https://gist.github.com/2053342

#### Event: 'exit'

```
function () {}
```

Emitted when the user exits the REPL in any of the defined ways. Namely, typing `.exit` at the repl, pressing Ctrl+C twice to signal SIGINT, or pressing Ctrl+D to signal "end" on the `input` stream.

Example of listening for `exit`:

```
r.on('exit', function () {
  console.log('Got "exit" event from repl!');
  process.exit();
});
```

### REPL Features

Inside the REPL, Control+D will exit. Multi-line expressions can be input. Tab completion is supported for both global and local variables.

The special variable `_` (underscore) contains the result of the last expression.

```
> [ "a", "b", "c" ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
4
```

The REPL provides access to any variables in the global scope. You can expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`. For example:

```
// repl_test.js
var repl = require("repl"),
    msg = "message";

repl.start().context.m = msg;
```

Things in the `context` object appear as local within the REPL:

```
mjr:~$ node repl_test.js
> m
'message'
```

There are a few special REPL commands:

- `.break` - While inputting a multi-line expression, sometimes you get lost or just don't care about completing it. `.break` will start over.
- `.clear` - Resets the `context` object to an empty object and clears any multi-line expression.
- `.exit` - Close the I/O stream, which will cause the REPL to exit.
- `.help` - Show this list of special commands.
- `.save` - Save the current REPL session to a file >.save ./file/to/save.js
- `.load` - Load a file into the current REPL session. >.load ./file/to/load.js

The following key combinations in the REPL have these special effects:

- `<ctrl>C` - Similar to the `.break` keyword. Terminates the current command. Press twice on a blank line to forcibly exit.
- `<ctrl>D` - Similar to the `.exit` keyword.

# Executing JavaScript

```
Stability: 2 - Unstable. See Caveats, below.
```

You can access this module with:

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

JavaScript code can be compiled and run immediately or compiled, saved, and run later.

## Caveats

The `vm` module has many known issues and edge cases. If you run into issues or unexpected behavior, please consult the open issues on GitHub. Some of the biggest problems are described below.

### Sandboxes

The `sandbox` argument to `vm.runInNewContext` and `vm.createContext`, along with the `initSandbox` argument to `vm.createContext`, do not behave as one might normally expect and their behavior varies between different versions of Node.

The key issue to be aware of is that V8 provides no way to directly control the global object used within a context. As a result, while properties of your `sandbox` object will be available in the context, any properties from the `prototypes` of the `sandbox` may not be available. Furthermore, the `this` expression within the global scope of the context evaluates to the empty object (`{}`) instead of to your sandbox.

Your sandbox's properties are also not shared directly with the script. Instead, the properties of the sandbox are copied into the context at the beginning of execution, and then after execution, the properties are copied back out in an attempt to propagate any changes.

### Globals

Properties of the global object, like `Array` and `String`, have different values inside of a context. This means that common expressions like `[] instanceof Array` or `Object.getPrototypeOf([]) === Array.prototype` may not produce expected results when used inside of scripts evaluated via the `vm` module.

Some of these problems have known workarounds listed in the issues for `vm` on GitHub. for example, `Array.isArray` works around the example problem with `Array`.

## vm.runInThisContext(code, [filename])

`vm.runInThisContext()` compiles `code`, runs it and returns the result. Running code does not have access to local scope. `filename` is optional, it's used only in stack traces.

Example of using `vm.runInThisContext` and `eval` to run the same code:

```
var localVar = 123,
    usingscript, evaled,
    vm = require('vm');

usingscript = vm.runInThisContext('localVar = 1;',
  'myfile.vm');
console.log('localVar: ' + localVar + ', usingscript: ' +
  usingscript);
evaled = eval('localVar = 1;');
console.log('localVar: ' + localVar + ', evaled: ' +
  evaled);

// localVar: 123, usingscript: 1
// localVar: 1, evaled: 1
```

`vm.runInThisContext` does not have access to the local scope, so `localVar` is unchanged. `eval` does have access to the local scope, so `localVar` is changed.

In case of syntax error in `code`, `vm.runInThisContext` emits the syntax error to stderr and throws an exception.

### vm.runInNewContext(code, [sandbox], [filename])

`vm.runInNewContext` compiles `code`, then runs it in `sandbox` and returns the result. Running code does not have access to local scope. The object `sandbox` will be used as the global object for `code`. `sandbox` and `filename` are optional, `filename` is only used in stack traces.

Example: compile and execute code that increments a global variable and sets a new one. These globals are contained in the sandbox.

```
var util = require('util'),
    vm = require('vm'),
    sandbox = {
      animal: 'cat',
      count: 2
    };

vm.runInNewContext('count += 1; name = "kitty"', sandbox, 'myfile.vm');
console.log(util.inspect(sandbox));

// { animal: 'cat', count: 3, name: 'kitty' }
```

Note that running untrusted code is a tricky business requiring great care. To prevent accidental global variable leakage, `vm.runInNewContext` is quite useful, but safely running untrusted code requires a separate process.

In case of syntax error in `code`, `vm.runInNewContext` emits the syntax error to stderr and throws an exception.

### vm.runInContext(code, context, [filename])

`vm.runInContext` compiles `code`, then runs it in `context` and returns the result. A (V8) context comprises a global object, together with a set of built-in objects and functions. Running code does not have access to local scope and the global object held within `context` will be used as the global object for `code`. `filename` is optional, it's used only in stack traces.

Example: compile and execute code in a existing context.

```
var util = require('util'),
    vm = require('vm'),
    initSandbox = {
      animal: 'cat',
      count: 2
    },
    context = vm.createContext(initSandbox);

vm.runInContext('count += 1; name = "CATT"', context, 'myfile.vm');
console.log(util.inspect(context));

// { animal: 'cat', count: 3, name: 'CATT' }
```

Note that `createContext` will perform a shallow clone of the supplied sandbox object in order to initialize the global object of the freshly constructed context.

Note that running untrusted code is a tricky business requiring great care. To prevent accidental global variable leakage, `vm.runInContext` is quite useful, but safely running untrusted code requires a separate process.

In case of syntax error in `code`, `vm.runInContext` emits the syntax error to stderr and throws an exception.

### vm.createContext([initSandbox])

`vm.createContext` creates a new context which is suitable for use as the 2nd argument of a subsequent call to `vm.runInContext`. A (V8) context comprises a global object together with a set of build-in objects and functions. The optional argument `initSandbox` will be shallow-copied to seed the initial contents of the global object used by the context.

### vm.createScript(code, [filename])

`createScript` compiles `code` but does not run it. Instead, it returns a `vm.Script` object representing this compiled code. This script can be run later many times using methods below. The returned script is not bound to any global object. It is bound before each run, just for that run. `filename` is optional, it's only used in stack traces.

In case of syntax error in `code`, `createScript` prints the syntax error to stderr and throws an exception.

### Class: Script

A class for running scripts. Returned by vm.createScript.

#### script.runInThisContext()

Similar to `vm.runInThisContext` but a method of a precompiled `Script` object. `script.runInThisContext` runs the code of `script` and returns the result. Running code does not have access to local scope, but does have access to the `global` object (v8: in actual context).

Example of using `script.runInThisContext` to compile code once and run it multiple times:

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

globalVar = 0;

var script = vm.createScript('globalVar += 1', 'myfile.vm');

for (var i = 0; i < 1000 ; i += 1) {
  script.runInThisContext();
}

console.log(globalVar);

// 1000
```

#### script.runInNewContext([sandbox])

Similar to `vm.runInNewContext` a method of a precompiled `Script` object. `script.runInNewContext` runs the code of `script` with `sandbox` as the global object and returns the result. Running code does not have access to local scope. `sandbox` is optional.

Example: compile code that increments a global variable and sets one, then execute this code multiple times. These globals are contained in the sandbox.

```
var util = require('util'),
    vm = require('vm'),
    sandbox = {
      animal: 'cat',
      count: 2
    };

var script = vm.createScript('count += 1; name = "kitty"', 'myfile.vm');

for (var i = 0; i < 10 ; i += 1) {
```

```
  script.runInNewContext(sandbox);
}

console.log(util.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

Note that running untrusted code is a tricky business requiring great care. To prevent accidental global variable leakage, `script.runInNewContext` is quite useful, but safely running untrusted code requires a separate process.

# Child Process

```
Stability: 3 - Stable
```

Node provides a tri-directional `popen(3)` facility through the `child_process` module.

It is possible to stream data through a child's `stdin`, `stdout`, and `stderr` in a fully non-blocking way.

To create a child process use `require('child_process').spawn()` or `require('child_process').fork()`. The semantics of each are slightly different, and explained below.

## Class: ChildProcess

`ChildProcess` is an EventEmitter.

Child processes always have three streams associated with them. `child.stdin`, `child.stdout`, and `child.stderr`. These may be shared with the stdio streams of the parent process, or they may be separate stream objects which can be piped to and from.

The ChildProcess class is not intended to be used directly. Use the `spawn()` or `fork()` methods to create a Child Process instance.

### Event: 'exit'

- `code` {Number} the exit code, if it exited normally.
- `signal` {String} the signal passed to kill the child process, if it was killed by the parent.

This event is emitted after the child process ends. If the process terminated normally, `code` is the final exit code of the process, otherwise `null`. If the process terminated due to receipt of a signal, `signal` is the string name of the signal, otherwise `null`.

Note that the child process stdio streams might still be open.

See `waitpid(2)`.

### Event: 'close'

This event is emitted when the stdio streams of a child process have all terminated. This is distinct from 'exit', since multiple processes might share the same stdio streams.

### Event: 'disconnect'

This event is emitted after using the `.disconnect()` method in the parent or in the child. After disconnecting it is no longer possible to send messages. An alternative way to check if you can send messages is to see if the `child.connected` property is `true`.

### Event: 'message'

- `message` {Object} a parsed JSON object or primitive value
- `sendHandle` {Handle object} a Socket or Server object

Messages send by `.send(message, [sendHandle])` are obtained using the `message` event.

### child.stdin

- {Stream object}

A `Writable Stream` that represents the child process's `stdin`. Closing this stream via `end()` often causes the child process to terminate.

If the child stdio streams are shared with the parent, then this will not be set.

### child.stdout

- {Stream object}

A `Readable Stream` that represents the child process's `stdout`.

If the child stdio streams are shared with the parent, then this will not be set.

### child.stderr

- {Stream object}

A `Readable Stream` that represents the child process's `stderr`.

If the child stdio streams are shared with the parent, then this will not be set.

### child.pid

- {Integer}

The PID of the child process.

Example:

```
var spawn = require('child_process').spawn,
    grep  = spawn('grep', ['ssh']);

console.log('Spawned child pid: ' + grep.pid);
grep.stdin.end();
```

### child.kill([signal])

- `signal` {String}

Send a signal to the child process. If no argument is given, the process will be sent `'SIGTERM'`. See `signal(7)` for a list of available signals.

```
var spawn = require('child_process').spawn,
    grep  = spawn('grep', ['ssh']);

grep.on('exit', function (code, signal) {
  console.log('child process terminated due to receipt of signal '+signal);
});

// send SIGHUP to process
grep.kill('SIGHUP');
```

Note that while the function is called `kill`, the signal delivered to the child process may not actually kill it. `kill` really just sends a signal to a process.

See `kill(2)`

### child.send(message, [sendHandle])

- `message` {Object}
- `sendHandle` {Handle object}

When using `child_process.fork()` you can write to the child using `child.send(message, [sendHandle])` and messages are received by a `'message'` event on the child.

For example:

```
var cp = require('child_process');

var n = cp.fork(__dirname + '/sub.js');

n.on('message', function(m) {
  console.log('PARENT got message:', m);
});

n.send({ hello: 'world' });
```

And then the child script, `'sub.js'` might look like this:

```
process.on('message', function(m) {
  console.log('CHILD got message:', m);
});

process.send({ foo: 'bar' });
```

In the child the `process` object will have a `send()` method, and `process` will emit objects each time it receives a message on its channel.

There is a special case when sending a `{cmd: 'NODE_foo'}` message. All messages containing a `NODE_` prefix in its `cmd` property will not be emitted in the `message` event, since they are internal messages used by node core. Messages containing the prefix are emitted in the `internalMessage` event, you should by all means avoid using this feature, it is subject to change without notice.

The `sendHandle` option to `child.send()` is for sending a TCP server or socket object to another process. The child will receive the object as its second argument to the `message` event.

**send server object**

Here is an example of sending a server:

```
var child = require('child_process').fork('child.js');

// Open up the server object and send the handle.
var server = require('net').createServer();
server.on('connection', function (socket) {
  socket.end('handled by parent');
});
server.listen(1337, function() {
  child.send('server', server);
});
```

And the child would the recive the server object as:

```
process.on('message', function(m, server) {
  if (m === 'server') {
    server.on('connection', function (socket) {
      socket.end('handled by child');
    });
  }
});
```

Note that the server is now shared between the parent and child, this means that some connections will be handled by the parent and some by the child.

**send socket object**

Here is an example of sending a socket. It will spawn two childs and handle connections with the remote address `74.125.127.100` as VIP by sending the socket to a "special" child process. Other sockets will go to a "normal" process.

```
var normal = require('child_process').fork('child.js', ['normal']);
var special = require('child_process').fork('child.js', ['special']);

// Open up the server and send sockets to child
```

```
var server = require('net').createServer();
server.on('connection', function (socket) {

  // if this is a VIP
  if (socket.remoteAddress === '74.125.127.100') {
    special.send('socket', socket);
    return;
  }
  // just the usual dudes
  normal.send('socket', socket);
});
server.listen(1337);
```

The `child.js` could look like this:

```
process.on('message', function(m, socket) {
  if (m === 'socket') {
    socket.end('You where handled as a ' + process.argv[2] + ' person');
  }
});
```

Note that once a single socket has been sent to a child the parent can no longer keep track of when the socket is destroyed. To indicate this condition the `.connections` property becomes `null`. It is also recomended not to use `.maxConnections` in this condition.

#### child.disconnect()

To close the IPC connection between parent and child use the `child.disconnect()` method. This allows the child to exit gracefully since there is no IPC channel keeping it alive. When calling this method the `disconnect` event will be emitted in both parent and child, and the `connected` flag will be set to `false`. Please note that you can also call `process.disconnect()` in the child process.

### child_process.spawn(command, [args], [options])

- `command` {String} The command to run
- `args` {Array} List of string arguments
- `options` {Object}
- `cwd` {String} Current working directory of the child process
- `stdio` {Array|String} Child's stdio configuration. (See below)
- `customFds` {Array} **Deprecated** File descriptors for the child to use for stdio. (See below)
- `env` {Object} Environment key-value pairs
- `detached` {Boolean} The child will be a process group leader. (See below)
- return: {ChildProcess object}

Launches a new process with the given `command`, with command line arguments in `args`. If omitted, `args` defaults to an empty Array.

The third argument is used to specify additional options, which defaults to:

```
{ cwd: undefined,
  env: process.env
}
```

`cwd` allows you to specify the working directory from which the process is spawned. Use `env` to specify environment variables that will be visible to the new process.

Example of running `ls -lh /usr`, capturing `stdout`, `stderr`, and the exit code:

```
var util  = require('util'),
    spawn = require('child_process').spawn,
    ls    = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

ls.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

ls.on('exit', function (code) {
  console.log('child process exited with code ' + code);
});
```

Example: A very elaborate way to run 'ps ax | grep ssh'

```
var util  = require('util'),
    spawn = require('child_process').spawn,
    ps    = spawn('ps', ['ax']),
    grep  = spawn('grep', ['ssh']);

ps.stdout.on('data', function (data) {
  grep.stdin.write(data);
});

ps.stderr.on('data', function (data) {
  console.log('ps stderr: ' + data);
});

ps.on('exit', function (code) {
  if (code !== 0) {
    console.log('ps process exited with code ' + code);
  }
  grep.stdin.end();
});

grep.stdout.on('data', function (data) {
  console.log(data);
});

grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

grep.on('exit', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

Example of checking for failed exec:

```
var spawn = require('child_process').spawn,
    child = spawn('bad_command');

child.stderr.setEncoding('utf8');
child.stderr.on('data', function (data) {
  if (/^execvp\(\)/.test(data)) {
    console.log('Failed to start child process.');
  }
});
```

Note that if spawn receives an empty options object, it will result in spawning the process with an empty environment rather than using `process.env`. This due to backwards compatibility issues with a deprecated API.

The 'stdio' option to `child_process.spawn()` is an array where each index corresponds to a fd in the child. The value is one of the following:

1. `'pipe'` - Create a pipe between the child process and the parent process. The parent end of the pipe is exposed to the parent as a property on the `child_process` object as `ChildProcess.stdio[fd]`. Pipes created for fds 0 - 2 are also available as ChildProcess.stdin, ChildProcess.stdout and ChildProcess.stderr, respectively.
2. `'ipc'` - Create an IPC channel for passing messages/file descriptors between parent and child. A ChildProcess may have at most *one* IPC stdio file descriptor. Setting this option enables the ChildProcess.send() method. If the child writes JSON messages to this file descriptor, then this will trigger ChildProcess.on('message'). If the child is a Node.js program, then the presence of an IPC channel will enable process.send() and process.on('message').
3. `'ignore'` - Do not set this file descriptor in the child. Note that Node will always open fd 0 - 2 for the processes it spawns. When any of these is ignored node will open `/dev/null` and attach it to the child's fd.
4. `Stream` object - Share a readable or writable stream that refers to a tty, file, socket, or a pipe with the child process. The stream's underlying file descriptor is duplicated in the child process to the fd that corresponds to the index in the `stdio` array.
5. Positive integer - The integer value is interpreted as a file descriptor that is is currently open in the parent process. It is shared with the child process, similar to how `Stream` objects can be shared.
6. `null`, `undefined` - Use default value. For stdio fds 0, 1 and 2 (in other words, stdin, stdout, and stderr) a pipe is created. For fd 3 and up, the default is `'ignore'`.

As a shorthand, the `stdio` argument may also be one of the following strings, rather than an array:

- `ignore` - `['ignore', 'ignore', 'ignore']`
- `pipe` - `['pipe', 'pipe', 'pipe']`
- `inherit` - `[process.stdin, process.stdout, process.stderr]` or `[0,1,2]`

Example:

```
var spawn = require('child_process').spawn;

// Child will use parent's stdios
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs present a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

If the `detached` option is set, the child process will be made the leader of a new process group. This makes it possible for the child to continue running after the parent exits.

By default, the parent will wait for the detached child to exit. To prevent the parent from waiting for a given `child`, use the `child.unref()` method, and the parent's event loop will not include the child in its reference count.

Example of detaching a long-running process and redirecting its output to a file:

```
 var fs = require('fs'),
     spawn = require('child_process').spawn,
     out = fs.openSync('./out.log', 'a'),
     err = fs.openSync('./out.log', 'a');
```

```
var child = spawn('prg', [], {
  detached: 'true',
  stdio: [ 'ignore', out, err ]
});

child.unref();
```

When using the `detached` option to start a long-running process, the process will not stay running in the background unless it is provided with a `stdio` configuration that is not connected to the parent. If the parent's `stdio` is inherited, the child will remain attached to the controlling terminal.

There is a deprecated option called `customFds` which allows one to specify specific file descriptors for the stdio of the child process. This API was not portable to all platforms and therefore removed. With `customFds` it was possible to hook up the new process' `[stdin, stdout, stderr]` to existing streams; `-1` meant that a new stream should be created. Use at your own risk.

There are several internal options. In particular `stdinStream`, `stdoutStream`, `stderrStream`. They are for INTERNAL USE ONLY. As with all undocumented APIs in Node, they should not be used.

See also: `child_process.exec()` and `child_process.fork()`

## child_process.exec(command, [options], callback)

- `command` {String} The command to run, with space-separated arguments
- `options` {Object}
- `cwd` {String} Current working directory of the child process
- `stdio` {Array|String} Child's stdio configuration. (See above)
- `customFds` {Array} **Deprecated** File descriptors for the child to use for stdio. (See above)
- `env` {Object} Environment key-value pairs
- `encoding` {String} (Default: 'utf8')
- `timeout` {Number} (Default: 0)
- `maxBuffer` {Number} (Default: 200*1024)
- `killSignal` {String} (Default: 'SIGTERM')
- `callback` {Function} called with the output when process terminates
- `error` {Error}
- `stdout` {Buffer}
- `stderr` {Buffer}
- Return: ChildProcess object

Runs a command in a shell and buffers the output.

```
var util = require('util'),
    exec = require('child_process').exec,
    child;

child = exec('cat *.js bad_file | wc -l',
  function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);
    if (error !== null) {
      console.log('exec error: ' + error);
    }
});
```

The callback gets the arguments `(error, stdout, stderr)`. On success, `error` will be `null`. On error, `error` will be an instance of `Error` and `err.code` will be the exit code of the child process, and `err.signal` will be set to the signal that terminated the process.

There is a second optional argument to specify several options. The default options are

```
{ encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null }
```

If `timeout` is greater than 0, then it will kill the child process if it runs longer than `timeout` milliseconds. The child process is killed with `killSignal` (default: `'SIGTERM'`). `maxBuffer` specifies the largest amount of data allowed on stdout or stderr - if this value is exceeded then the child process is killed.

### child_process.execFile(file, args, options, callback)

- `file` {String} The filename of the program to run
- `args` {Array} List of string arguments
- `options` {Object}
- `cwd` {String} Current working directory of the child process
- `stdio` {Array|String} Child's stdio configuration. (See above)
- `customFds` {Array} **Deprecated** File descriptors for the child to use for stdio. (See above)
- `env` {Object} Environment key-value pairs
- `encoding` {String} (Default: 'utf8')
- `timeout` {Number} (Default: 0)
- `maxBuffer` {Number} (Default: 200*1024)
- `killSignal` {String} (Default: 'SIGTERM')
- `callback` {Function} called with the output when process terminates
- `error` {Error}
- `stdout` {Buffer}
- `stderr` {Buffer}
- Return: ChildProcess object

This is similar to `child_process.exec()` except it does not execute a subshell but rather the specified file directly. This makes it slightly leaner than `child_process.exec`. It has the same options.

### child_process.fork(modulePath, [args], [options])

- `modulePath` {String} The module to run in the child
- `args` {Array} List of string arguments
- `options` {Object}
- `cwd` {String} Current working directory of the child process
- `env` {Object} Environment key-value pairs
- `encoding` {String} (Default: 'utf8')
- `timeout` {Number} (Default: 0)
- Return: ChildProcess object

This is a special case of the `spawn()` functionality for spawning Node processes. In addition to having all the methods in a normal ChildProcess instance, the returned object has a communication channel built-in. See `child.send(message, [sendHandle])` for details.

By default the spawned Node process will have the stdout, stderr associated with the parent's. To change this behavior set the `silent` property in the `options` object to `true`.

These child Nodes are still whole new instances of V8. Assume at least 30ms startup and 10mb memory for each new Node. That is, you cannot create many thousands of them.

# Assert

```
Stability: 5 - Locked
```

This module is used for writing unit tests for your applications, you can access it with `require('assert')`.

### assert.fail(actual, expected, message, operator)

Throws an exception that displays the values for `actual` and `expected` separated by the provided operator.

### assert(value, message), assert.ok(value, [message])

Tests if value is truthy, it is equivalent to `assert.equal(true, !!value, message);`

### assert.equal(actual, expected, [message])

Tests shallow, coercive equality with the equal comparison operator ( `==` ).

### assert.notEqual(actual, expected, [message])

Tests shallow, coercive non-equality with the not equal comparison operator ( `!=` ).

### assert.deepEqual(actual, expected, [message])

Tests for deep equality.

### assert.notDeepEqual(actual, expected, [message])

Tests for any deep inequality.

### assert.strictEqual(actual, expected, [message])

Tests strict equality, as determined by the strict equality operator ( `===` )

### assert.notStrictEqual(actual, expected, [message])

Tests strict non-equality, as determined by the strict not equal operator ( `!==` )

### assert.throws(block, [error], [message])

Expects `block` to throw an error. `error` can be constructor, regexp or validation function.

Validate instanceof using constructor:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  Error
);
```

Validate error message using RegExp:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /value/
);
```

Custom error validation:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  function(err) {
    if ( (err instanceof Error) && /value/.test(err) ) {
      return true;
    }
  },
  "unexpected error"
);
```

### assert.doesNotThrow(block, [error], [message])

Expects `block` not to throw an error, see assert.throws for details.

### assert.ifError(value)

Tests if value is not a false value, throws if it is a true value. Useful when testing the first argument, `error` in callbacks.

# TTY

```
Stability: 2 - Unstable
```

The `tty` module houses the `tty.ReadStream` and `tty.WriteStream` classes. In most cases, you will not need to use this module directly.

When node detects that it is being run inside a TTY context, then `process.stdin` will be a `tty.ReadStream` instance and `process.stdout` will be a `tty.WriteStream` instance. The preferred way to check if node is being run in a TTY context is to check `process.stdout.isTTY`:

```
$ node -p -e "Boolean(process.stdout.isTTY)"
true
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

## tty.isatty(fd)

Returns `true` or `false` depending on if the `fd` is associated with a terminal.

## tty.setRawMode(mode)

Deprecated. Use `tty.ReadStream#setRawMode()` (i.e. `process.stdin.setRawMode()`) instead.

## Class: ReadStream

A `net.Socket` subclass that represents the readable portion of a tty. In normal circumstances, `process.stdin` will be the only `tty.ReadStream` instance in any node program (only when `isatty(0)` is true).

### rs.isRaw

A `Boolean` that is initialized to `false`. It represents the current "raw" state of the `tty.ReadStream` instance.

### rs.setRawMode(mode)

`mode` should be `true` or `false`. This sets the properties of the `tty.ReadStream` to act either as a raw device or default. `isRaw` will be set to the resulting mode.

## Class WriteStream

A `net.Socket` subclass that represents the writable portion of a tty. In normal circumstances, `process.stdout` will be the only `tty.WriteStream` instance ever created (and only when `isatty(1)` is true).

### ws.columns

A `Number` that gives the number of columns the TTY currently has. This property gets updated on "resize" events.

### ws.rows

A `Number` that gives the number of rows the TTY currently has. This property gets updated on "resize" events.

**Event: 'resize'**

`function () {}`

Emitted by `refreshSize()` when either of the `columns` or `rows` properties has changed.

```
process.stdout.on('resize', function() {
  console.log('screen size has changed!');
  console.log(process.stdout.columns + 'x' + process.stdout.rows);
});
```

# Zlib

```
Stability: 3 - Stable
```

You can access this module with:

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

This provides bindings to Gzip/Gunzip, Deflate/Inflate, and DeflateRaw/InflateRaw classes. Each class takes the same options, and is a readable/writable Stream.

## Examples

Compressing or decompressing a file can be done by piping an fs.ReadStream into a zlib stream, then into an fs.WriteStream.

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

Compressing or decompressing data in one step can be done by using the convenience methods.

```
var input = '.................................';
zlib.deflate(input, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString('base64'));
  }
});

var buffer = new Buffer('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString());
  }
});
```

To use this module in an HTTP client or server, use the accept-encoding on requests, and the content-encoding header on responses.

**Note: these examples are drastically simplified to show the basic concept.** Zlib encoding can be expensive, and the results ought to be cached. See Memory Usage Tuning below for more information on the speed/memory/compression tradeoffs involved in zlib usage.

```
// client request example
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
var request = http.get({ host: 'izs.me',
                         path: '/',
                         port: 80,
                         headers: { 'accept-encoding': 'gzip,deflate' } });
request.on('response', function(response) {
  var output = fs.createWriteStream('izs.me_index.html');

  switch (response.headers['content-encoding']) {
    // or, just use zlib.createUnzip() to handle both cases
    case 'gzip':
      response.pipe(zlib.createGunzip()).pipe(output);
      break;
    case 'deflate':
      response.pipe(zlib.createInflate()).pipe(output);
```

```
      break;
    default:
      response.pipe(output);
      break;
  }
});

// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
http.createServer(function(request, response) {
  var raw = fs.createReadStream('index.html');
  var acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  // Note: this is not a conformant accept-encoding parser.
  // See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
  if (acceptEncoding.match(/\bdeflate\b/)) {
    response.writeHead(200, { 'content-encoding': 'deflate' });
    raw.pipe(zlib.createDeflate()).pipe(response);
  } else if (acceptEncoding.match(/\bgzip\b/)) {
    response.writeHead(200, { 'content-encoding': 'gzip' });
    raw.pipe(zlib.createGzip()).pipe(response);
  } else {
    response.writeHead(200, {});
    raw.pipe(response);
  }
}).listen(1337);
```

### zlib.createGzip([options])

Returns a new Gzip object with an options.

### zlib.createGunzip([options])

Returns a new Gunzip object with an options.

### zlib.createDeflate([options])

Returns a new Deflate object with an options.

### zlib.createInflate([options])

Returns a new Inflate object with an options.

### zlib.createDeflateRaw([options])

Returns a new DeflateRaw object with an options.

### zlib.createInflateRaw([options])

Returns a new InflateRaw object with an options.

### zlib.createUnzip([options])

Returns a new Unzip object with an options.

### Class: zlib.Gzip

Compress data using gzip.

### Class: zlib.Gunzip

Decompress a gzip stream.

### Class: zlib.Deflate

Compress data using deflate.

### Class: zlib.Inflate

Decompress a deflate stream.

### Class: zlib.DeflateRaw

Compress data using deflate, and do not append a zlib header.

### Class: zlib.InflateRaw

Decompress a raw deflate stream.

### Class: zlib.Unzip

Decompress either a Gzip- or Deflate-compressed stream by auto-detecting the header.

## Convenience Methods

All of these take a string or buffer as the first argument, and call the supplied callback with `callback(error, result)`. The compression/decompression engine is created using the default settings in all convenience methods. To supply different options, use the zlib classes directly.

### zlib.deflate(buf, callback)

Compress a string with Deflate.

### zlib.deflateRaw(buf, callback)

Compress a string with DeflateRaw.

### zlib.gzip(buf, callback)

Compress a string with Gzip.

### zlib.gunzip(buf, callback)

Decompress a raw Buffer with Gunzip.

### zlib.inflate(buf, callback)

Decompress a raw Buffer with Inflate.

### zlib.inflateRaw(buf, callback)

Decompress a raw Buffer with InflateRaw.

### zlib.unzip(buf, callback)

Decompress a raw Buffer with Unzip.

## Options

Each class takes an options object. All options are optional. (The convenience methods use the default settings for all options.)

Note that some options are only relevant when compressing, and are ignored by the decompression classes.

- chunkSize (default: 16*1024)
- windowBits
- level (compression only)
- memLevel (compression only)
- strategy (compression only)
- dictionary (deflate/inflate only, empty dictionary by default)

See the description of `deflateInit2` and `inflateInit2` at http://zlib.net/manual.html#Advanced for more information on these.

## Memory Usage Tuning

From `zlib/zconf.h`, modified to node's usage:

The memory requirements for deflate are (in bytes):

```
(1 << (windowBits+2)) +  (1 << (memLevel+9))
```

that is: 128K for windowBits=15 + 128K for memLevel = 8 (default values) plus a few kilobytes for small objects.

For example, if you want to reduce the default memory requirements from 256K to 128K, set the options to:

```
{ windowBits: 14, memLevel: 7 }
```

Of course this will generally degrade compression (there's no free lunch).

The memory requirements for inflate are (in bytes)

```
1 << windowBits
```

that is, 32K for windowBits=15 (default value) plus a few kilobytes for small objects.

This is in addition to a single internal output slab buffer of size `chunkSize`, which defaults to 16K.

The speed of zlib compression is affected most dramatically by the `level` setting. A higher level will result in better compression, but will take longer to complete. A lower level will result in less compression, but will be much faster.

In general, greater memory usage options will mean that node has to make fewer calls to zlib, since it'll be able to process more data in a single `write` operation. So, this is another factor that affects the speed, at the cost of memory usage.

## Constants

All of the constants defined in zlib.h are also defined on `require('zlib')`. In the normal course of operations, you will not need to ever set any of these. They are documented here so that their presence is not surprising. This section is taken almost directly from the zlib documentation. See http://zlib.net/manual.html#Constants for more details.

Allowed flush values.

- `zlib.Z_NO_FLUSH`
- `zlib.Z_PARTIAL_FLUSH`
- `zlib.Z_SYNC_FLUSH`
- `zlib.Z_FULL_FLUSH`
- `zlib.Z_FINISH`
- `zlib.Z_BLOCK`
- `zlib.Z_TREES`

Return codes for the compression/decompression functions. Negative values are errors, positive values are used for special but normal events.

- `zlib.Z_OK`
- `zlib.Z_STREAM_END`
- `zlib.Z_NEED_DICT`
- `zlib.Z_ERRNO`
- `zlib.Z_STREAM_ERROR`
- `zlib.Z_DATA_ERROR`
- `zlib.Z_MEM_ERROR`
- `zlib.Z_BUF_ERROR`
- `zlib.Z_VERSION_ERROR`

Compression levels.

- `zlib.Z_NO_COMPRESSION`
- `zlib.Z_BEST_SPEED`
- `zlib.Z_BEST_COMPRESSION`
- `zlib.Z_DEFAULT_COMPRESSION`

Compression strategy.

- `zlib.Z_FILTERED`
- `zlib.Z_HUFFMAN_ONLY`
- `zlib.Z_RLE`
- `zlib.Z_FIXED`
- `zlib.Z_DEFAULT_STRATEGY`

Possible values of the data_type field.

- `zlib.Z_BINARY`
- `zlib.Z_TEXT`
- `zlib.Z_ASCII`

- `zlib.Z_UNKNOWN`

The deflate compression method (the only one supported in this version).

- `zlib.Z_DEFLATED`

For initializing zalloc, zfree, opaque.

- `zlib.Z_NULL`

## os

```
Stability: 4 - API Frozen
```

Provides a few basic operating-system related utility functions.

Use `require('os')` to access this module.

### os.tmpDir()

Returns the operating system's default directory for temp files.

### os.hostname()

Returns the hostname of the operating system.

### os.type()

Returns the operating system name.

### os.platform()

Returns the operating system platform.

### os.arch()

Returns the operating system CPU architecture.

### os.release()

Returns the operating system release.

### os.uptime()

Returns the system uptime in seconds.

### os.loadavg()

Returns an array containing the 1, 5, and 15 minute load averages.

### os.totalmem()

Returns the total amount of system memory in bytes.

### os.freemem()

Returns the amount of free system memory in bytes.

### os.cpus()

Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of CPU ticks spent in: user, nice, sys, idle, and irq).

Example inspection of os.cpus:

```
[ { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 252020,
       nice: 0,
       sys: 30340,
       idle: 1070356870,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 306960,
       nice: 0,
       sys: 26980,
       idle: 1071569080,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 248450,
       nice: 0,
       sys: 21750,
       idle: 1070919370,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 256880,
       nice: 0,
       sys: 19430,
       idle: 1070905480,
       irq: 20 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 511580,
       nice: 20,
       sys: 40900,
       idle: 1070842510,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 291660,
       nice: 0,
       sys: 34360,
       idle: 1070888000,
       irq: 10 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 308260,
       nice: 0,
       sys: 55410,
       idle: 1071129970,
```

```
      irq: 880 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 266450,
       nice: 1480,
       sys: 34920,
       idle: 1072572010,
       irq: 30 } } ]
```

### os.networkInterfaces()

Get a list of network interfaces:

```
{ lo0:
   [ { address: '::1', family: 'IPv6', internal: true },
     { address: 'fe80::1', family: 'IPv6', internal: true },
     { address: '127.0.0.1', family: 'IPv4', internal: true } ],
  en1:
   [ { address: 'fe80::cabc:c8ff:feef:f996', family: 'IPv6',
       internal: false },
     { address: '10.0.1.123', family: 'IPv4', internal: false } ],
  vmnet1: [ { address: '10.99.99.254', family: 'IPv4', internal: false } ],
  vmnet8: [ { address: '10.88.88.1', family: 'IPv4', internal: false } ],
  ppp0: [ { address: '10.2.0.231', family: 'IPv4', internal: false } ] }
```

### os.EOL

A constant defining the appropriate End-of-line marker for the operating system.

# Debugger

```
Stability: 3 - Stable
```

V8 comes with an extensive debugger which is accessible out-of-process via a simple TCP protocol. Node has a built-in client for this debugger. To use this, start Node with the `debug` argument; a prompt will appear:

```
% node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
debug>
```

Node's debugger client doesn't support the full range of commands, but simple step and inspection is possible. By putting the statement `debugger;` into the source code of your script, you will enable a breakpoint.

For example, suppose `myscript.js` looked like this:

```
// myscript.js
x = 5;
setTimeout(function () {
  debugger;
  console.log("world");
}, 1000);
console.log("hello");
```

Then once the debugger is run, it will break on line 4.

```
% node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
debug> cont
< hello
break in /home/indutny/Code/git/indutny/myscript.js:3
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
  4   console.log("world");
  5 }, 1000);
debug> next
break in /home/indutny/Code/git/indutny/myscript.js:4
  2 setTimeout(function () {
  3   debugger;
  4   console.log("world");
  5 }, 1000);
  6 console.log("hello");
debug> repl
Press Ctrl + C to leave debug repl
> x
5
> 2+2
4
debug> next
< world
break in /home/indutny/Code/git/indutny/myscript.js:5
  3   debugger;
  4   console.log("world");
```

```
  5 }, 1000);
  6 console.log("hello");
  7
debug> quit
%
```

The `repl` command allows you to evaluate code remotely. The `next` command steps over to the next line. There are a few other commands available and more to come. Type `help` to see others.

## Watchers

You can watch expression and variable values while debugging your code. On every breakpoint each expression from the watchers list will be evaluated in the current context and displayed just before the breakpoint's source code listing.

To start watching an expression, type `watch("my_expression")`. `watchers` prints the active watchers. To remove a watcher, type `unwatch("my_expression")`.

## Commands reference

### Stepping

- `cont`, `c` - Continue execution
- `next`, `n` - Step next
- `step`, `s` - Step in
- `out`, `o` - Step out
- `pause` - Pause running code (like pause button in Developer TOols)

### Breakpoints

- `setBreakpoint()`, `sb()` - Set breakpoint on current line
- `setBreakpoint(line)`, `sb(line)` - Set breakpoint on specific line
- `setBreakpoint('fn()')`, `sb(...)` - Set breakpoint on a first statement in functions body
- `setBreakpoint('script.js', 1)`, `sb(...)` - Set breakpoint on first line of script.js
- `clearBreakpoint`, `cb(...)` - Clear breakpoint

### Info

- `backtrace`, `bt` - Print backtrace of current execution frame
- `list(5)` - List scripts source code with 5 line context (5 lines before and after)
- `watch(expr)` - Add expression to watch list
- `unwatch(expr)` - Remove expression from watch list
- `watchers` - List all watchers and their values (automatically listed on each breakpoint)
- `repl` - Open debugger's repl for evaluation in debugging script's context

### Execution control

- `run` - Run script (automatically runs on debugger's start)
- `restart` - Restart script
- `kill` - Kill script

**Various**

- `scripts` - List all loaded scripts
- `version` - Display v8's version

## Advanced Usage

The V8 debugger can be enabled and accessed either by starting Node with the `--debug` command-line flag or by signaling an existing Node process with `SIGUSR1`.

# Cluster

```
Stability: 1 - Experimental
```

A single instance of Node runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node processes to handle the load.

The cluster module allows you to easily create a network of processes that all share server ports.

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

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

  cluster.on('exit', function(worker, code, signal) {
    console.log('worker ' + worker.pid + ' died');
  });
} else {
  // Workers can share any TCP connection
  // In this case its a HTTP server
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");
  }).listen(8000);
}
```

Running node will now share port 8000 between the workers:

```
% node server.js
Worker 2438 online
Worker 2437 online
```

This feature was introduced recently, and may change in future versions. Please try it out and provide feedback.

Also 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.

When you call `server.listen(...)` in a worker, it serializes the arguments and passes the request to the master process. If the master process already has a listening server matching the worker's requirements, then it passes the handle to the worker. If it does not already have a listening server matching that requirement, then it will create one, and pass the handle to the child.

This causes potentially surprising behavior in three edge cases:

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 case 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.

When multiple processes are all `accept()`ing on the same underlying resource, the operating system load-balances across them very efficiently. 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. Node 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.

### cluster.settings

- {Object}
- `exec` {String} file path to worker file. (Default=`__filename`)
- `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`)

All settings set by the `.setupMaster` is stored in this settings object. This object is not supposed to be change or set manually, by you.

### 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}

This boolean flag is true if the process is a worker forked from a master. If the `process.env.NODE_UNIQUE_ID` is set to a value, then `isWorker` is `true`.

### 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 you own timeout.

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

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

### Event: 'online'

- `worker` {Worker object}

After forking a new worker, the worker should respond with a online message. When the master receives a online message it will emit such event. The difference between 'fork' and 'online' is that fork is emitted when the master tries to fork a worker, and 'online' is emitted when the worker is being executed.

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

### Event: 'listening'

- `worker` {Worker object}
- `address` {Object}

When calling `listen()` from a worker, a 'listening' event is automatically assigned to the server instance. When the server is listening a message is send to the master where the 'listening' event is emitted.

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', function(worker, address) {
  console.log("A worker is now connected to " + address.address + ":" + address.port);
});
```

### Event: 'disconnect'

- `worker` {Worker object}

When a workers IPC channel has disconnected this event is emitted. This will happen when the worker dies, usually after calling `.destroy()`.

When calling `.disconnect()`, there may be a delay between the `disconnect` and `exit` events. This event can be used to detect if the process is stuck in a cleanup or if there are long-living connections.

```
cluster.on('disconnect', function(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', function(worker, code, signal) {
  var exitCode = worker.process.exitCode;
  console.log('worker ' + worker.pid + ' died ('+exitCode+'). restarting...');
  cluster.fork();
});
```

### Event: 'setup'

- `worker` {Worker object}

When the `.setupMaster()` function has been executed this event emits. If `.setupMaster()` was not executed before `fork()` this function will call `.setupMaster()` with no arguments.

### cluster.setupMaster([settings])

- `settings` {Object}
- `exec` {String} file path to worker file. (Default=`__filename`)
- `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`)

The `setupMaster` is used to change the default 'fork' behavior. It takes one option object argument.

Example:

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

### cluster.fork([env])

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

Spawn a new worker process. This can only be called from the master process.

### cluster.settings

- {Object}
- `exec` {String} file path to worker file. Default: `__filename`
- `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`)

All settings set by the `.setupMaster` is stored in this settings object. This object is not supposed to be change or set manually.

### cluster.disconnect([callback])

- `callback` {Function} called when all workers are disconnected and handlers are closed

When calling this method, all workers will commit a graceful suicide. When they are disconnected all internal handlers will be closed, allowing the master process to die graceful if no other event is waiting.

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

### cluster.workers

- {Object}

In the cluster all living worker objects are stored in this object by there `id` as the key. This makes it easy to loop through all living workers.

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

## 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`.

### worker.id

- {String}

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.process

- {ChildProcess object}

All workers are created using `child_process.fork()`, the returned object from this function is stored in process.

See: Child Process module

### worker.suicide

- {Boolean}

This property is a boolean. It is set when a worker dies after calling `.destroy()` or immediately after calling the `.disconnect()` method. Until then it is `undefined`.

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

- `message` {Object}
- `sendHandle` {Handle object}

This function is equal to the send methods provided by `child_process.fork()`. In the master you should use this function to send a message to a specific worker. However in a worker you can also use `process.send(message)`, since this is the same function.

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', function(msg) {
    process.send(msg);
  });
}
```

### worker.destroy()

This function will kill the worker, and inform the master to not spawn a new worker. The boolean `suicide` lets you distinguish between voluntary and accidental exit.

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

// destroy worker
worker.destroy();
```

### worker.disconnect()

When calling this function the worker will no longer accept new connections, but they will be handled by any other listening worker. Existing connection will be allowed to exit as usual. When no more connections exist, the IPC channel to the worker will close allowing it to die graceful. When the IPC channel is closed the `disconnect` event will emit, this is then followed by the `exit` event, there is emitted when the worker finally die.

Because there might be long living connections, it is useful to implement a timeout. This example ask the worker to disconnect and after 2 seconds it will destroy the server. An alternative wound be to execute `worker.destroy()` after 2 seconds, but that would normally not allow the worker to do any cleanup if needed.

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

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

  worker.on('disconnect', function() {
    clearTimeout(timeout);
  });

} else if (cluster.isWorker) {
  var net = require('net');
  var server = net.createServer(function(socket) {
    // connection never end
  });

  server.listen(8000);

  server.on('close', function() {
    // cleanup
  });

  process.on('message', function(msg) {
    if (msg === 'force kill') {
      server.destroy();
    }
  });
}
```

### Event: 'message'

- `message` {Object}

This event is the same as the one provided by `child_process.fork()`. In the master you should use this event, however 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:

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

if (cluster.isMaster) {

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

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

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

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

} else {

  // Worker processes have a http server.
  http.Server(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");

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

### Event: 'online'

Same as the `cluster.on('online')` event, but emits only when the state change on the specified worker.

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

### Event: 'listening'

- `address` {Object}

Same as the `cluster.on('listening')` event, but emits only when the state change on the specified worker.

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

#### Event: 'disconnect'

Same as the `cluster.on('disconnect')` event, but emits only when the state change on the specified worker.

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

#### 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.

Emitted by the individual worker instance, when the underlying child process is terminated. See child_process event: 'exit'.

```
var worker = cluster.fork();
worker.on('exit', function(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!");
  }
};
```

# Appendix 1 - Third Party Modules

There are many third party modules for Node. At the time of writing, August 2010, the master repository of modules is the wiki page.

You can also find third party modules via npm search.

This appendix is intended as a SMALL guide to new-comers to help them quickly find what are considered to be quality modules. It is not intended to be a complete list. There may be better more complete modules found elsewhere.

- Module Installer: npm
- HTTP Middleware: Connect
- Web Framework: Express
- Web Sockets: Socket.IO
- HTML Parsing: HTML5
- mDNS/Zeroconf/Bonjour
- RabbitMQ, AMQP
- mysql
- Serialization: msgpack
- Scraping: Apricot
- Debugger: ndb is a CLI debugger inspector is a web based tool.
- pcap binding
- ncurses
- Testing/TDD/BDD: vows, mocha, mjsunit.runner

Patches to this list are welcome.