Node.js Manual &Documentation

Table Of Contents

Synopsis

An example of a [web server](#) written with Node which responds with 'Hello World':

```javascript
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 object are available in the global scope and can be accessed from anywhere.

global

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

The process object. See the 'process object' section.

require()

To require modules. See the 'Modules' section.

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.paths

An array of search paths for `require()`. This array can be modified to add custom paths.

Example: add a new path to the beginning of the search list

```
require.paths.unshift('/usr/local/node');
```

__filename

The filename of the script being executed. This is the absolute path, and not necessarily the same filename passed in as a command line argument.

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

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

__dirname

The dirname of the script being executed.

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

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

module

A reference to the current module. In particular `module.exports` is the same as the `exports` object. See `src/node.js` for more information.

Timers

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

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

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

Node uses the CommonJS module system.

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`.

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`, and then `.node`.
`.js` files are interpreted as JavaScript text files, and `.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.

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 either the module is found, or 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.

Optimizations to the `node_modules` Lookup Process

When there are many levels of nested dependencies, it is possible for these file trees to get fairly long. The following optimizations are thus made to the process.

First, `/node_modules` is never appended to a folder already ending in `/node_modules`.
Second, if the file calling `require()` is already inside a `node_modules` hierarchy, then the top-most `node_modules` folder is treated as the root of the search tree.
For example, if the file at `'/home/ry/projects/foo/node_modules/bar/node_modules/baz/quux.js'` called `require('asdf.js')`, then node would search the following locations:

- `/home/ry/projects/foo/node_modules/bar/node_modules/baz/node_modules/asdf.js`

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

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 `ndex.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.

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)
```

1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with `./` 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. LOAD_AS_FILE(X/index)

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
6. return DIRS
```

Loading from the `require.paths` Folders

In node, `require.paths` is an array of strings that represent paths to be searched for modules when they are not prefixed with `'/'`, `'./'`, or `'../'`. For example, if require.paths were set to:

```
[ '/home/micheil/.node_modules',
  '/usr/local/lib/node_modules' ]
```

Then calling `require('bar/baz.js')` would search the following locations:

- 1: `'/home/micheil/.node_modules/bar/baz.js'`
- 2: `'/usr/local/lib/node_modules/bar/baz.js'`

The `require.paths` array can be mutated at run time to alter this behavior. It is set initially from the `NODE_PATH` environment variable, which is a colon-delimited list of absolute paths. In the previous example, the `NODE_PATH` environment variable might have been set to:

```
/home/micheil/.node_modules:/usr/local/lib/node_modules
```

Loading from the `require.paths` locations is only performed if the module could not be found using the `node_modules` algorithm above. Global modules are lower priority than bundled dependencies.

Note: Please Avoid Modifying `require.paths`

For compatibility reasons, `require.paths` is still given first priority in the module lookup process. However, it may disappear in a future release. While it seemed like a good idea at the time, and enabled a lot of useful experimentation, in practice a mutable `require.paths` list is often a troublesome source of confusion and headaches.

Setting `require.paths` to some other value does nothing.

This does not do what one might expect:

```
require.paths = [ '/usr/lib/node' ];
```

All that does is lose the reference to the *actual* node module lookup paths, and create a new reference to some other thing that isn't used for anything.

Putting relative paths in `require.paths` is... weird.

If you do this:

```
require.paths.push('./lib');
```

then it does *not* add the full resolved path to where `./lib` is on the filesystem. Instead, it literally adds `'./lib'`, meaning that if you do `require('y.js')` in `/a/b/x.js`, then it'll look in `/a/b/lib/y.js`. If you then did `require('y.js')` in `/l/m/n/o/p.js`, then it'd look in `/l/m/n/o/lib/y.js`.
In practice, people have used this as an ad hoc way to bundle dependencies, but this technique is brittle.

Zero Isolation

There is (by regrettable design), only one `require.paths` array used by all modules.
As a result, if one node program comes to rely on this behavior, it may permanently and subtly alter the behavior of all other node programs in the same process. As the application stack grows, we tend to assemble functionality, and it is a problem with those parts interact in ways that are difficult to predict.

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).
- libev, 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 libev. That is, if you perform any I/O, libev will need to be used. Node uses the `EV_DEFAULT` event loop. Documentation can be found [here](#).
- libeio, C thread pool library. Used to execute blocking POSIX system calls asynchronously. Mostly wrappers already exist for such calls, in `src/file.cc` so you will probably not need to use it. If you do need it, look at the header file `deps/libeio/eio.h`.
- 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.

To get started let's make a small Addon which does the following except in C++:

```
exports.hello = 'world';
```

To get started we create a file `hello.cc`:

```
#include <v8.h>

using namespace v8;

extern "C" void
init (Handle<Object> target)
{
  HandleScope scope;
```

```
  target->Set(String::New("hello"), String::New("world"));
}
```

This source code needs to be built into hello.node, the binary Addon. To do this we create a file called wscript which is python code and looks like this:

```
srcdir = '.'
blddir = 'build'
VERSION = '0.0.1'

def set_options(opt):
  opt.tool_options('compiler_cxx')

def configure(conf):
  conf.check_tool('compiler_cxx')
  conf.check_tool('node_addon')

def build(bld):
  obj = bld.new_task_gen('cxx', 'shlib', 'node_addon')
  obj.target = 'hello'
  obj.source = 'hello.cc'
```

Running node-waf configure build will create a file build/default/hello.node which is our Addon. node-waf is just [WAF](), the python-based build system. node-waf is provided for the ease of users.
All Node addons must export a function called init with this signature:

```
extern 'C' void init (Handle<Object> target)
```

For the moment, that is all the documentation on addons. Please see [https://github.com/ry/node_postgres]() for a real example.

process

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

Event: 'exit'

`function () {}`
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'

`function (err) { }`
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

function () {}
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

A writable stream to stderr. Writes on this stream are blocking.

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.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=0)

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()

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

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

process.setgid(id)

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.

```
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()

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

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

process.setuid(id)

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.

```
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.installPrefix

A compiled-in property that exposes NODE_PREFIX.

```
console.log('Prefix: ' + process.installPrefix);
```

process.kill(pid, signal='SIGTERM')

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.platform

What platform you're running on. 'linux2', 'darwin', etc.

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

process.memoryUsage()

Returns an object describing the memory usage of the Node process.

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

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

This will generate:

```
{ rss: 4935680,
  vsize: 41893888,
  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));
```

util

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

util.debug(string)

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

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

util.log(string)

Output with timestamp on stdout.

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

util.inspect(object, showHidden=false, depth=2)

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.
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.

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

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

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

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

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*.

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. The listener is invoked only the first 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 from the listener array for the specified event.

emitter.setMaxListeners(n)

By default EventEmitters will print a warning if more than 10 listeners are added to it. 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. This array can be manipulated, e.g. to remove listeners.

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

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

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

Event: 'newListener'

```
function (event, listener) { }
```
This event is emitted any time someone adds a new listener.

Buffers

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` object is global.

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.
- `'utf8'` - Multi byte encoded Unicode characters. Many web pages and other document formats use UTF-8.
- `'ucs2'` - 2-bytes, little endian encoded Unicode characters. It can encode only BMP(Basic Multilingual Plane, U+0000 - U+FFFF).
- `'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.

new Buffer(size)

Allocates a new buffer of `size` octets.

new Buffer(array)

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

new Buffer(str, encoding='utf8')

Allocates a new buffer containing the given `str`.

buffer.write(string, offset=0, encoding='utf8')

Writes `string` to the buffer at `offset` using the given encoding. 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. In the case of` utf8` encoding, the method will not write partial characters.

Example: write a utf8 string into a buffer, then print it

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

// 12 bytes: 陆 + 录 = 戮
```

buffer.toString(encoding, start=0, end=buffer.length)

Decodes and returns a string from buffer data encoded with encoding beginning at start and ending at end.
See buffer.write() example, above.

buffer[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
```

Buffer.isBuffer(obj)

Tests if `obj` is a `Buffer`.

Buffer.byteLength(string, encoding='utf8')

Gives the actual byte length of a string. 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");

// ½ + ¼ = ¾: 9 characters, 12 bytes
```

buffer.length

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

// 1234
// 1234
```

buffer.copy(targetBuffer, targetStart=0, sourceStart=0, sourceEnd=buffer.length)

Does a memcpy() between buffers.

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!!!!!!!!!!!!!
```

buffer.slice(start, end=buffer.length)

Returns a new buffer which references the same memory as the old, but offset and cropped by the start and end 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
```

Streams

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.

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.

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 file descriptor has been closed. Not all streams will emit this. (For example, an incoming HTTP request will not emit `'close'`.)

Event: 'fd'

```
function (fd) { }
```
Emitted when a file descriptor is received on the stream. Only UNIX streams support this functionality; all others will simply never emit this event.

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'`, `'ascii'`, or `'base64'`.

stream.pause()

Pauses the incoming `'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.destroySoon()

After the write queue is drained, close the file descriptor.

stream.pipe(destination, [options])

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

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.

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");
});
```

NOTE: If the source stream does not support `pause()` and `resume()`, this function adds simple definitions which simply emit `'pause'` and `'resume'` events on the source stream.

Writable Stream

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

Event: 'drain'

```
function () { }
```
Emitted after a `write()` method was called that returned `false` 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='utf8', [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.

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.

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

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:

- `key` : a string holding the PEM encoded private key
- `cert` : a string holding the PEM encoded certificate
- `ca` : either a string or list of strings of PEM encoded CA certificates to trust.

If no 'ca' details are given, then node.js will use the default publicly trusted list of CAs as given

inhttp://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.

hash.update(data)

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

hash.digest(encoding='binary')

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

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.

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='binary')

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

crypto.createCipher(algorithm, key)

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

`algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent releases, `openssl list-cipher-algorithms` will display the available cipher algorithms.

cipher.update(data, input_encoding='binary', output_encoding='binary')

Updates the cipher with `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`,`'ascii'` or `'binary'`.
The `output_encoding` specifies the output format of the enciphered data, and can be` 'binary'`, `'base64'` or `'hex'`.
Returns the enciphered contents, and can be called many times with new data as it is streamed.

cipher.final(output_encoding='binary')

Returns any remaining enciphered contents, with `output_encoding` being one of: `'binary'`,`'ascii'` or` 'utf8'`.

crypto.createDecipher(algorithm, key)

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

decipher.update(data, input_encoding='binary', output_encoding='binary')

Updates the decipher with `data`, which is encoded in `'binary'`, `'base64'` or `'hex'`. The `output_decoding`specifies in what format to return the deciphered plaintext: `'binary'`,`'ascii'` or `'utf8'`.

decipher.final(output_encoding='binary')

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

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'.

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='binary')

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'`.

crypto.createVerify(algorithm)

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

verifier.update(data)

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

verifier.verify(cert, signature, signature_format='binary')

Verifies the signed data by using the `cert` which is a string containing the PEM
encoded public key, and `signature`, which is the previously calculates signature

for the data, in the `signature_format` which can
be`'binary'`,`'hex'` or `'base64'`.
Returns true or false depending on the validity of the signature for the data and
public key.

TLS (SSL)

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)

s = tls.connect(port, [host], [options], callback)

Creates a new client connection to the given `port` and `host`. (If `host` defaults to `localhost`.) `options` should be an object which specifies

- `key`: A string or `Buffer` containing the private key of the server in PEM format. (Required)
- `cert`: A string or `Buffer` containing the certificate key of the server 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.

`tls.connect()` returns a cleartext `CryptoStream` object.

After the TLS/SSL handshake the `callback` is called. The `callback` will be called no matter if the server's certificate was authorized or not. It is up to the user to test `s.authorized` to see if the server certificate was signed by one of the specified CAs. If `s.authorized === false` then the error can be found in `s.authorizationError`.

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.

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')
};

tls.createServer(options, function (s) {
  s.write("welcome!\n");
  s.pipe(s);
}).listen(8000);
```

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

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

tls.createServer(options, secureConnectionListener)

This is a constructor for the `tls.Server` class. The options object has these possibilities:

- key: A string or `Buffer` containing the private key of the server in PEM format. (Required)
- 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.
- 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`.

Event: 'secureConnection'

function (cleartextStream) {}
This event is emitted after a new connection has been successfully handshaked. The argument is a duplex instance of `stream.Stream`. 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.

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.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.

File System

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`.

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.

fs.rename(path1, path2, [callback])

Asynchronous rename(2). No arguments other than a possible exception are given to the completion callback.

fs.renameSync(path1, path2)

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.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.stat(path, [callback])

Asynchronous stat(2). The callback gets two arguments (err, stats) where stats is a fs.Stats object. It looks like this:

```
{ dev: 2049,
  ino: 305352,
  mode: 16877,
```

```
    nlink: 12,
    uid: 1000,
    gid: 1000,
    rdev: 0,
    size: 4096,
    blksize: 4096,
    blocks: 8,
    atime: '2009-06-29T11:11:55Z',
    mtime: '2009-06-29T11:11:40Z',
    ctime: '2009-06-29T11:11:40Z' }
```

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.

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(linkdata, path, [callback])

Asynchronous symlink(2). No arguments other than a possible exception are given to the completion callback.

fs.symlinkSync(linkdata, path)

Synchronous symlink(2).

fs.readlink(path, [callback])

Asynchronous readlink(2). The callback gets two arguments (err, resolvedPath).

fs.readlinkSync(path)

Synchronous readlink(2). Returns the resolved path.

fs.realpath(path, [callback])

Asynchronous realpath(2). The callback gets two arguments (err, resolvedPath).

fs.realpathSync(path)

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.

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', 'r+', 'w', 'w+', 'a', or 'a+'. mode defaults to 0666. The callback gets two arguments (err, fd).

fs.openSync(path, flags, [mode])

Synchronous open(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 isnull, the data will be written at the current position. See pwrite(2).
The callback will be given two arguments (err, written) where written specifies how many *bytes* were written.
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='utf8')

Synchronous version of string-based `fs.write()`. 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 two arguments, (`err`, `bytesRead`).

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)

Synchronous version of string-based `fs.read`. Returns the number of `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='utf8', [callback])

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer.
Example:

```
fs.writeFile('message.txt', 'Hello Node', function (err) {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

fs.writeFileSync(filename, data, encoding='utf8')

The synchronous version of `fs.writeFile`.

fs.watchFile(filename, [options], listener)

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`, a polling value in milliseconds. The default is `{ persistent: true, interval: 0 }`.
The `listener` gets two arguments the current stat object and the previous stat object:

```
fs.watchFile(f, 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)

Stop watching for changes on `filename`.

fs.Stats

Objects returned from `fs.stat()` and `fs.lstat()` are of this type.

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()` (only valid with `fs.lstat()`)
- `stats.isFIFO()`
- `stats.isSocket()`

fs.ReadStream

`ReadStream` is a `Readable Stream`.

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. When used, both the limits must be specified always.

An example to read the last 10 bytes of a file which is 100 bytes long:

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

fs.WriteStream

`WriteStream` is a `Writable Stream`.

Event: 'open'

```
function (fd) { }
```
`fd` is the file descriptor used by the WriteStream.

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 }
```

Path

This module contains utilities for dealing with file paths.

Use `require('path')` to use it. It provides the following methods:

path.normalize(p)

Normalize a string path, taking care of '..' and '.' parts.
When multiple slashes are found, they're replaces 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.

Example:

```
node> require('path').join(
...   '/foo', 'bar', 'baz/asdf', 'quux', '..')
'/foo/bar/baz/asdf'
```

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.
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.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. Everything after the last '.' in the last portion of the path. If there is no '.' in the last portion of the path or the only '.' is the first character, then it returns an empty string. Examples:

```
path.extname('index.html')
// returns
'.html'

path.extname('index')
// returns
''
```

path.exists(p, [callback])

Test whether or not the given path exists. Then, call the `callback` argument with either true or false. Example:

```
path.exists('/etc/passwd', function (exists) {
  util.debug(exists ? "it's there" : "no passwd!");
});
```

path.existsSync(p)

Synchronous version of `path.exists`.

net

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 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.

net.createConnection(arguments...)

Construct a new socket object and opens a socket to the given location. When the socket is established the` connect'` event will be emitted.
The arguments for this method change the type of connection:

- `net.createConnection(port, [host])`
  Creates a TCP connection to `port` on `host`. If `host` is omitted, `localhost` will be assumed.
- `net.createConnection(path)`
  Creates unix socket connection to `path`

---

net.Server

This class is used to create a TCP or UNIX server.

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) {
  c.write('hello\r\n');
  c.pipe(c);
});
server.listen(8124, 'localhost');
```

Test this by using `telnet`:

```
telnet localhost 8124
```

To listen on the socket `/tmp/echo.sock` the last line would just be changed to

```
server.listen('/tmp/echo.sock');
```

Use `nc` to connect to a UNIX domain socket server:

```
nc -U /tmp/echo.sock
```

`net.Server` is an `EventEmitter` with the following events:

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.
One issue some users run into is getting `EADDRINUSE` errors. Meaning another server is already running on the requested port. One way of handling this would be to wait a second and the 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 are set SO_REUSEADDR already)

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 called when the server has been bound.

server.listenFD(fd)

Start a server listening for connections on the given file descriptor.

This file descriptor must have already had the `bind(2)` and `listen(2)` system calls invoked on it.

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 of the server as seen by the operating system. Useful to find which port was assigned when giving getting an OS-assigned address

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);
});
```

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.

Event: 'connection'

```
function (socket) {}
```
Emitted when a new connection is made. `socket` is an instance of `net.Socket`.

Event: 'close'

```
function () {}
```
Emitted when the server closes.

---

net.Socket

This object is an abstraction of 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.

`net.Socket` instances are EventEmitters with the following events:

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], [callback])

socket.connect(path, [callback])

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 `callback` 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 an running quickly. The computer cannot necessarily 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=null)

Sets the encoding (either `'ascii'`, `'utf8'`, or `'base64'`) for data that is received.

socket.setSecure()

This function has been removed in v0.3. It used to upgrade the connection to SSL/TLS. See the TLS for the new API.

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.write(data, [encoding], [fileDescriptor], [callback])

For UNIX sockets, it is possible to send a file descriptor through the socket. Simply add the `fileDescriptor` argument and listen for the `'fd'` event on the other end.

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=true)

Disables the Nagle algorithm. By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting `noDelay` will immediately fire off data each time `socket.write()` is called.

socket.setKeepAlive(enable=false, [initialDelay])

Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket. 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.

socket.remoteAddress

The string representation of the remote IP address. For
example, `'74.125.127.100'` or `'2001:4860:a005::68'`.
This member is only present in server-side connections.

Event: 'connect'

`function () { }`
Emitted when a socket connection successfully is established. See `connect()`.

Event: 'data'

`function (data) { }`
Emitted when data is received. The argument `data` will be a `Buffer` or `String`.
Encoding of data is set by `socket.setEncoding()`. (See the section
on `Readable Socket` for more information.)

Event: 'end'

`function () { }`
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'

`function () { }`
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'

`function () { }`
Emitted when the write buffer becomes empty. Can be used to throttle uploads.

Event: 'error'

`function (exception) { }`
Emitted when an error occurs. The `'close'` event will be called directly
following this event.

Event: 'close'

`function (had_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

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.

DNS

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

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) {
        console.log('reverse for ' + a + ' failed: ' +
          err.message);
      } else {
        console.log('reverse for ' + a + ': ' +
          JSON.stringify(domains));
      }
    });
  });
});
```

dns.lookup(domain, family=null, callback)

Resolves a domain (e.g. `'google.com'`) into the first found A (IPv4) or AAAA (IPv6) record.

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`).

dns.resolve(domain, rrtype='A', callback)

Resolves a domain (e.g. `google.com`) into an array of the record types specified by rrtype. Valid rrtypes are A(IPV4 addresses), AAAA (IPV6 addresses), MX (mail exchange records), TXT (text records), SRV (SRV records), and PTR (used for reverse IP lookups).

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` would be an instanceof `Error` object, where `err.errno` is one of the error codes listed below and `err.message` is a string describing the error in English.

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.reverse(ip, callback)

Reverse resolves an ip address to an array of domain names.

The callback has arguments (err, domains).
If there an an error, err will be non-null and an instanceof the Error object.
Each DNS query can return an error code.

- dns.TEMPFAIL: timeout, SERVFAIL or similar.
- dns.PROTOCOL: got garbled reply.
- dns.NXDOMAIN: domain does not exists.
- dns.NODATA: domain exists but no data of reqd type.
- dns.NOMEM: out of memory while processing.
- dns.BADQUERY: the query is malformed.

UDP / Datagram Sockets

Datagram sockets are available through require('dgram'). Datagrams are most commonly handled as IP/UDP messages but they can also be used over Unix domain sockets.

Event: 'message'

function (msg, rinfo) { }
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'

function () { }
Emitted when a socket starts listening for datagrams. This happens as soon as UDP sockets are created. Unix domain sockets do not start listening until calling bind() on them.

Event: 'close'

function () { }
Emitted when a socket is closed with close(). No new message events will be emitted on this socket.

dgram.createSocket(type, [callback])

Creates a datagram socket of the specified types. Valid types are: udp4, udp6, and unix_dgram.

Takes an optional callback which is added as a listener for message events.

dgram.send(buf, offset, length, path, [callback])

For Unix domain datagram sockets, the destination address is a pathname in the filesystem. An optional callback may be supplied that is invoked after the sendto call is completed by the OS. It is not safe to re-use buf until the callback is invoked. Note that unless the socket is bound to a pathname with bind() there is no way to receive messages on this socket.

Example of sending a message to syslogd on OSX via Unix domain socket /var/run/syslog:

```
var dgram = require('dgram');
var message = new Buffer("A message to log.");
var client = dgram.createSocket("unix_dgram");
client.send(message, 0, message.length, "/var/run/syslog",
  function (err, bytes) {
    if (err) {
      throw err;
    }
    console.log("Wrote " + bytes + " bytes to socket.");
});
```

dgram.send(buf, offset, length, port, address, [callback])

For UDP sockets, the destination port and IP address must be specified. A string may be supplied for theaddress 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.

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");
client.close();
```

dgram.bind(path)

For Unix domain datagram sockets, start listening for incoming datagrams on a socket specified by path. Note that clients may send() without bind(), but no datagrams will be received without a bind().

Example of a Unix domain datagram server that echoes back all messages it receives:

```
var dgram = require("dgram");
var serverPath = "/tmp/dgram_server_sock";
var server = dgram.createSocket("unix_dgram");

server.on("message", function (msg, rinfo) {
  console.log("got: " + msg + " from " + rinfo.address);
  server.send(msg, 0, msg.length, rinfo.address);
});

server.on("listening", function () {
  console.log("server listening " + server.address().address);
})

server.bind(serverPath);
```

Example of a Unix domain datagram client that talks to this server:

```
var dgram = require("dgram");
var serverPath = "/tmp/dgram_server_sock";
var clientPath = "/tmp/dgram_client_sock";

var message = new Buffer("A message at " + (new Date()));

var client = dgram.createSocket("unix_dgram");

client.on("message", function (msg, rinfo) {
  console.log("got: " + msg + " from " + rinfo.address);
});

client.on("listening", function () {
  console.log("client listening " + client.address().address);
  client.send(message, 0, message.length, serverPath);
});

client.bind(clientPath);
```

dgram.bind(port, [address])

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");
var messageToSend = new Buffer("A message to send");

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. UDP sockets automatically listen for messages, even if they did not call bind().

dgram.address()

Returns an object containing the address information for a socket. For UDP sockets, this object will containaddress and port. For Unix domain sockets, it will contain only address.

dgram.setBroadcast(flag)

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)

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)

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 64.

dgram.setMulticastLoopback(flag)

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])

Tells the kernel to join a multicast group with `IP_ADD_MEMBERSHIP` socket option.

If `multicastAddress` is not specified, the OS will try to add membership to all valid interfaces.

dgram.dropMembership(multicastAddress, [multicastInterface])

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 `multicastAddress` is not specified, the OS will try to drop membership to all valid interfaces.

HTTP

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.Server

This is an `EventEmitter` with the following events:

Event: 'request'

`function (request, response) { }`
`request` is an instance of `http.ServerRequest` and `response` is an instance of `http.ServerResponse`

Event: 'connection'

`function (stream) { }`
When a new TCP stream is established. `stream` is an object of type `net.Stream`. Usually users will not want to access this event. The `stream` can also be accessed at `request.connection`.

Event: 'close'

`function (errno) { }`
Emitted when the server closes.

Event: 'request'

`function (request, response) {}`
Emitted each time there is request. Note that there may be multiple requests per connection (in the case of keep-alive connections).

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: '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.

http.createServer(requestListener)

Returns a new web server object.

The `requestListener` is a function which is automatically added to the `'request'` event.

server.listen(port, [hostname], [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.

This function is asynchronous. The last parameter `callback` will be called when the server has been bound to the 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 called when the server has been bound.

server.close()

Stops the server from accepting new connections.

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.
This is an `EventEmitter` with the following events:

Event: 'data'

`function (chunk) { }`
Emitted when a piece of the message body is received.

Example: A chunk of the body is given as the single argument. The transfer-encoding has been decoded. The body chunk is a string. The body encoding is set with `request.setBodyEncoding()`.

Event: 'end'

`function () { }`
Emitted exactly once for each message. No arguments. After emitted no other events will be emitted on the request.

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=null)

Set the encoding for the request body. Either `'utf8'` or `'binary'`. Defaults to `null`, which means that the `'data'` event will emit a `Buffer` object..

request.pause()

Pauses request from emitting events. Useful to throttle back an upload.

request.resume()

Resumes a paused request.

request.connection

The `net.Stream` object associated with the connection.
With HTTPS support, use request.connection.verifyPeer() and
request.connection.getPeerCertificate() to obtain the client's authentication
details.

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. It is a `Writable Stream`.

response.writeContinue()

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request
body should be sent. See the 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.

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;
```

response.setHeader(name, value)

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, it's 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.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='utf8')

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': 'TraceInfo' });
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:

- `host`: A domain name or IP address of the server to issue the request to.
- `port`: Port of remote server.
- `method`: A string specifying the HTTP request method. Possible values: `'GET'` (default), `'POST'`, `'PUT'`, and `'DELETE'`.
- `path`: Request path. Should include query string and fragments if any. E.G. `'/index.html?page=12'`
- `headers`: An object containing request headers.

`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);
  });
});

// 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.

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:

```
var options = {
  host: 'www.google.com',
  port: 80,
  path: '/index.html'
};

http.get(options, function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

http.Agent

http.getAgent(host, port)

`http.request()` uses a special `Agent` for managing multiple connections to an HTTP server. Normally `Agent` instances should not be exposed to user code, however in certain situations it's useful to check the status of the agent. The `http.getAgent()` function allows you to access the agents.

Event: 'upgrade'

`function (request, 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.

See the description of the `upgrade` event for `http.Server` for further details.

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.

agent.maxSockets

By default set to 5. Determines how many concurrent sockets the agent can have open.

agent.sockets

An array of sockets currently in use by the Agent. Do not modify.

agent.queue

A queue of requests waiting to be sent to sockets.

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);
});
```

This is a `Writable Stream`.
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`.

request.write(chunk, encoding='utf8')

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 an array of integers or a string.
The `encoding` argument is optional and only applies when `chunk` is a string.

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.)

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.

Event: 'data'

`function (chunk) {}`
Emitted when a piece of the message body is received.

Event: 'end'

`function () {}`
Emitted exactly once for each message. No arguments. After emitted no other events will be emitted on the response.

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=null)

Set the encoding for the response body. Either `'utf8'`, `'ascii'`, or `'base64'`. Defaults to `null`, which means that the `'data'` event will emit a `Buffer` object..

response.pause()

Pauses response from emitting events. Useful to throttle back a download.

response.resume()

Resumes a paused response.

HTTPS

HTTPS is the HTTP protocol over TLS/SSL. In Node this is implemented as a separate module.

https.Server

https.createServer

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);
```

https.request(options, callback)

Makes a request to a secure web server. Similar options to `http.request()`.
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'`.
- key: Private key to use for SSL. 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.

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);
});
```

URL

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.
  Example: `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`
- `protocol`: The request protocol.
  Example: `'http:'`
- `host`: The full host portion of the URL, including port and authentication information.
  Example: `'user:pass@host.com:8080'`
- `auth`: The authentication information portion of a URL.
  Example: `'user:pass'`
- `hostname`: Just the 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'`
- `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=false)

Take a URL string, and return an object. Pass `true` as the second argument to also parse the query string using the `querystring` module.

url.format(urlObj)

Take a parsed URL object, and return a formatted URL string.

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

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'})
// returns
'foo=bar'

querystring.stringify({foo: 'bar', baz: 'bob'}, ';', ':')
// returns
'foo:bar;baz:bob'
```

querystring.parse(str, sep='&', eq='=')

Deserialize a query string to an object. Optionally override the default separator and assignment characters.

Example:

```
querystring.parse('a=b&b=c')
// returns
{ a: 'b', b: 'c' }
```

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.

REPL

A Read-Eval-Print-Loop (REPL) is available both as a standalone program and easily includable in other programs. 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 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(prompt='> ', stream=process.stdin)

Starts a REPL with `prompt` as the prompt and `stream` for all I/O. `prompt` is optional and defaults to `>` . `stream` is optional and defaults to `process.stdin`. 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("node via stdin> ");

net.createServer(function (socket) {
  connections += 1;
  repl.start("node via Unix socket> ", socket);
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
  connections += 1;
  repl.start("node via TCP socket> ", socket);
}).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.

REPL Features

Inside the REPL, Control+D will exit. Multi-line expressions can be input.

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.

Child Processes

Node provides a tri-directional `popen(3)` facility through the `ChildProcess` class.
It is possible to stream data through the child's `stdin`, `stdout`, and `stderr` in a fully non-blocking way.
To create a child process use `require('child_process').spawn()`.
Child processes always have three streams associated with them. `child.stdin`, `child.stdout`, and`child.stderr`.
`ChildProcess` is an `EventEmitter`.

Event: 'exit'

`function (code, signal) {}`
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`.
See `waitpid(2)`.

child.stdin

A `Writable Stream` that represents the child process's `stdin`. Closing this stream via `end()` often causes the child process to terminate.

child.stdout

A `Readable Stream` that represents the child process's `stdout`.

child.stderr

A `Readable Stream` that represents the child process's `stderr`.

child.pid

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_process.spawn(command, args=[], [options])

Launches a new process with the given command, with command line arguments in args. If omitted, argsdefaults to an empty Array.
The third argument is used to specify additional options, which defaults to:

```
{ cwd: undefined,
  env: process.env,
  customFds: [-1, -1, -1],
  setsid: false
}
```

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. With customFds it is possible to hook up the new process' [stdin, stout, stderr] to existing streams; -1 means that a new stream should be created. setsid, if set true, will cause the subprocess to be run in a new session.
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.');
  }
});
```

See also: child_process.exec()

child_process.exec(command, [options], callback)

High-level way to execute a command as a child process, buffer the output, and return it all in a callback.

```
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.kill(signal='SIGTERM')

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)`

Assert

This module is used for writing unit tests for your applications, you can access it with `require('assert')`.

assert.fail(actual, expected, message, operator)

Tests if `actual` is equal to `expected` using the operator provided.

assert.ok(value, [message])

Tests if value is a `true` value, 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

Use `require('tty')` to access this module.
Example:

```
var tty = require('tty');
tty.setRawMode(true);
process.stdin.resume();
process.stdin.on('keypress', function(char, key) {
  if (key && key.ctrl && key.name == 'c') {
    console.log('graceful exit');
    process.exit()
  }
});
```

tty.open(path, args=[])

Spawns a new process with the executable pointed to by `path` as the session leader to a new pseudo terminal.
Returns an array `[slaveFD, childProcess]`. `slaveFD` is the file descriptor of the slave end of the pseudo terminal. `childProcess` is a child process object.

tty.isatty(fd)

Returns `true` or `false` depending on if the `fd` is associated with a terminal.

tty.setRawMode(mode)

`mode` should be `true` or `false`. This sets the properties of the current process's stdin fd to act either as a raw device or default.

tty.setWindowSize(fd, row, col)

`ioctl`s the window size settings to the file descriptor.

tty.getWindowSize(fd)

Returns `[row, col]` for the TTY associated with the file descriptor.

os Module

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

os.hostname()

Returns the hostname of the operating system.

os.type()

Returns the operating system name.

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 } } ]
```

Debugger

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
debug>
```

At this point `myscript.js` is not yet running. To start the script, enter the command `run`. If everything works okay, the output should look like this:

```
% node debug myscript.js
debug> run
debugger listening on port 5858
connecting...ok
```

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
debug> run
debugger listening on port 5858
connecting...ok
hello
break in #<an Object>._onTimeout(), myscript.js:4
  debugger;
  ^
debug> next
break in #<an Object>._onTimeout(), myscript.js:5
  console.log("world");
  ^
debug> print x
5
debug> print 2+2
4
debug> next
world
break in #<an Object>._onTimeout() returning undefined, myscript.js:6
}, 1000);
^
debug> quit
A debugging session is active. Quit anyway? (y or n) y
%
```

The print command allows you to evaluate variables. 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.

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.

Appendixes

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](#).

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](#), [expresso](#), [mjsunit.runner](#)

Patches to this list are welcome.