Node v0.1.100

    - require
    - require.paths
    - __filename
    - __dirname
    - module
* +process
    - Event: 'exit'
    - Event: 'uncaughtException'
    - Signal Events
    - process.stdout
    - process.openStdin
    - process.argv
    - process.execPath
    - process.chdir
    - process.compile
    - process.cwd
    - process.env
    - process.exit
    - process.getgid
    - process.getuid
    - process.version
    - process.installPrefix
    - process.kill
    - process.pid
    - process.platform
    - process.memoryUsage
    - process.nextTick
    - process.umask
* +sys
    - sys.print
    - sys.debug
    - sys.log
    - sys.inspect
    - sys.pump
* +Timers
    - setTimeout
    - clearTimeout
    - setInterval
    - clearInterval
* +Child Processes
    - Event: 'exit'
    - child_process.spawn
    - child.kill
    - child.pid
    - child.stdin.write
    - child.stdin.end
    - child_process.exec
* +Script
    - Script.runInThisContext

    - Script.runInNewContext
    - new Script
    - script.runInThisContext
    - script.runInNewContext
* +File System
    - fs.rename
    - fs.renameSync
    - fs.truncate
    - fs.truncateSync
    - fs.chmod
    - fs.chmodSync
    - fs.stat
    - fs.statSync
    - fs.link
    - fs.linkSync
    - fs.symlink
    - fs.symlinkSync
    - fs.readlink
    - fs.readlinkSync
    - fs.realpath
    - fs.realpathSync
    - fs.unlink
    - fs.unlinkSync
    - fs.rmdir
    - fs.rmdirSync
    - fs.mkdir
    - fs.mkdirSync
    - fs.readdir
    - fs.readdirSync
    - fs.close
    - fs.closeSync
    - fs.open
    - fs.openSync
    - fs.write
    - fs.writeSync
    - fs.read
    - fs.readSync
    - fs.readFile
    - fs.readFileSync
    - fs.writeFile
    - fs.writeFileSync
    - fs.watchFile
    - fs.unwatchFile
* fs.Stats
* fs.ReadStream
    - fs.createReadStream
    - readStream.readable
    - readStream.pause
    - readStream.resume

    o readStream.destroy
* +fs.WriteStream
    o fs.createWriteStream
    o writeStream.writeable
    o writeStream.write
    o writeStream.end
    o writeStream.destroy
* HTTP
* http.Server
    o Event: 'request'
    o Event: 'connection'
    o Event: 'close'
    o http.createServer
    o Event: 'request'
    o Event: 'upgrade'
    o Event: 'clientError'
    o server.listen
    o server.listen
    o server.setSecure
    o server.close
* +http.ServerRequest
    o Event: 'data'
    o Event: 'end'
    o request.method
    o request.url
    o request.headers
    o request.httpVersion
    o request.setEncoding
    o request.pause
    o request.resume
    o request.connection
* +http.ServerResponse
    o response.writeHead
    o response.write
    o response.end
* +http.Client
    o Event: 'upgrade'
    o http.createClient
    o client.request
    o client.verifyPeer
    o client.getPeerCertificate
* +http.ClientRequest
    o Event 'response'
    o request.write
    o request.end
* +http.ClientResponse
    o Event: 'data'
    o Event: 'end'
    o response.statusCode

    - response.httpVersion
    - response.headers
    - response.setEncoding
    - response.pause
    - response.resume
    - response.client
* +net.Server
    - Event: 'connection'
    - Event: 'close'
    - net.createServer
    - server.listen
    - server.listen
    - server.listenFD
    - server.close
* +net.Stream
    - Event: 'connect'
    - Event: 'secure'
    - Event: 'data'
    - Event: 'end'
    - Event: 'timeout'
    - Event: 'drain'
    - Event: 'error'
    - Event: 'close'
    - net.createConnection
    - stream.connect
    - stream.remoteAddress
    - stream.readyState
    - stream.setEncoding
    - stream.setSecure
    - stream.verifyPeer
    - stream.getPeerCertificate
    - stream.write
    - stream.end
    - stream.destroy
    - stream.pause
    - stream.resume
    - stream.setTimeout
    - stream.setNoDelay
    - stream.setKeepAlive
* +Crypto
    - crypto.createCredentials
    - crypto.createHash
    - hash.update
    - hash.digest
    - crypto.createHmac
    - hmac.update
    - hmac.digest
    - crypto.createCipher
    - cipher.update

    - cipher.final
    - crypto.createDecipher
    - decipher.update
    - decipher.final
    - crypto.createSign
    - signer.update
    - signer.sign
    - crypto.createVerify
    - verifier.update
    - verifier.verify
* +DNS
    - dns.resolve
    - dns.resolve4
    - dns.resolve6
    - dns.resolveMx
    - dns.resolveTxt
    - dns.resolveSrv
    - dns.reverse
* +Assert
    - assert.fail
    - assert.ok
    - assert.equal
    - assert.notEqual
    - assert.deepEqual
    - assert.notDeepEqual
    - assert.strictEqual
    - assert.notStrictEqual
    - assert.throws
    - assert.doesNotThrow
    - assert.ifError
* +Path
    - path.join
    - path.normalizeArray
    - path.normalize
    - path.dirname
    - path.basename
    - path.extname
    - path.exists
* +URL
    - url.parse
    - url.format
    - url.resolve
* +Query String
    - querystring.stringify
    - querystring.parse
    - querystring.escape
    - querystring.unescape
* +REPL
    - repl.start

        o REPL Features
    * Modules
    * Addons

node(1)

  1. node(1)
  2.
  3. node(1)

NAME

node -- evented I/O for V8 JavaScript
Synopsis

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

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

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

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

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

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

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

Node comes with a number of modules that are compiled in to the process, most of which are documented below. The most common way to use these modules is with require('name') and then assigning the return value to a local variable with the same name as the module.

Example:

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

It is possible to extend node with other modules. See 'Modules'
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. Access the class with require('buffer').Buffer.

Converting between Buffers and JavaScript string objects requires an explicit encoding method. Node supports 3 string encodings: UTF-8 ('utf8'), ASCII ('ascii'), and Binary ('binary').

*

'ascii' - for 7 bit ASCII data only. This encoding method is very fast, and will strip the high bit if set.

*

'utf8' - Unicode characters. Many web pages and other document formats use UTF-8.

*

'binary' - A legacy encoding. Used to store raw binary data in a string by only using the first 8 bits of every character. Don't use this.

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

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

// 12 bytes: ½ + ¼ = ¾
```

buffer.toString(encoding, start, end)

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:

```
var Buffer = require('buffer').Buffer,
  str = "node.js",
  buf = new Buffer(str.length),
  i;

for (i = 0; i < str.length ; i += 1) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf);

// node.js
```

Buffer.byteLength(string, encoding)

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:

```
var Buffer = require('buffer').Buffer,
  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.

```
var Buffer = require('buffer').Buffer,
  buf = new Buffer(1234);

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

// 1234
// 1234
```

buffer.copy(targetBuffer, targetStart, sourceStart, sourceEnd)

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.

```
var Buffer = require('buffer').Buffer,
  buf1 = new Buffer(26),
  buf2 = new Buffer(26),
  i;

for (i = 0 ; i < 26 ; i += 1) {
  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)

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 Buffer = require('buffer').Buffer,
  buf1 = new Buffer(26), buf2,
  i;

for (i = 0 ; i < 26 ; i += 1) {
  buf1[i] = i + 97; // 97 is ASCII a
}

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
```

EventEmitter

Many objects in Node emit events: a TCP server emits an event each time there is a stream, a child process emits an event when it exits. All objects which emit events are instances of events.EventEmitter.

Events are represented by a camel-cased string. Here are some examples: 'stream', 'data', 'messageBegin'.

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

require('events').EventEmitter to access the EventEmitter class.

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

When an EventEmitter experiences an error, the typical action is to emit an 'error' event. Error events are special--if there is no handler for them they will print a stack trace and exit the program.
Event: 'newListener'

function (event, listener) { }

This event is made any time someone adds a new listener.
Event: 'error'

function (exception) { }

If an error was encountered, then this event is emitted. This event is special - when there are no listeners to receive the error Node will terminate execution and display the exception's stack trace.
emitter.addListener(event, listener)

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

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

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.
emitter.removeAllListeners(event)

Removes all listeners from the listener array for the specified event.
emitter.listeners(event)

Returns an array of listeners for the specified event. This array can be manipulated, e.g. to remove listeners.
emitter.emit(event, arg1, arg2, ...)

Execute each of the listeners in order with the supplied arguments.
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 be 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.setEncoding(encoding)

Makes the data event emit a string instead of a Buffer. encoding can be 'utf8', 'ascii', or 'binary'.
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.
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.
stream.write(string, encoding, [fd])

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

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

Same as the above except with a raw buffer.
stream.end()

Terminates the stream with EOF or FIN.
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.
Global Objects

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

global

The global namespace object.
process

The process object. Most stuff lives in here. See the 'process object' section.
require()

To require modules. See the 'Modules' section.
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');
console.log(require.paths);
// /usr/local/node,/Users/mjr/.node_libraries
```

__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.
__dirname

The dirname of the script being executed.

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

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

module

A reference to the current module (of type process.Module). In particular module.exports is the same as the exports object. See src/process.js for more information.
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.addListener('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.addListener('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:

```
var stdin = process.openStdin();

process.addListener('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.openStdin()

Opens the standard input stream, returns a readable stream.

Example of opening standard input and listening for both events:

```
var stdin = process.openStdin();

stdin.setEncoding('utf8');

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

stdin.addListener('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.
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.compile(code, filename)

Similar to eval except that you can specify a filename for better error reporting and the code cannot see the local scope. The value of filename will be used as a filename if a stack trace is generated by the compiled code.

Example of using process.compile and eval to run the same code:

```
var localVar = 123,
    compiled, evaled;

compiled = process.compile('localVar = 1;', 'myfile.js');
console.log('localVar: ' + localVar + ', compiled: ' + compiled);
evaled = eval('localVar = 1;');
console.log('localVar: ' + localVar + ', evaled: ' + evaled);

// localVar: 123, compiled: 1
// localVar: 1, evaled: 1
```

process.compile does not have access to the local scope, so localVar is unchanged. eval does have access to the local scope, so localVar is changed.

In case of syntax error in code, process.compile exits node.

See also: Script
process.cwd()

Returns the current working directory of the process.

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

process.env

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

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

To exit with a 'failure' code:

process.exit(1);

The shell that executed node should see the exit code as 1.
process.getgid(), process.setgid(id)

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

```
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(), process.setuid(id)

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

```
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)

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 'SIGINT'. 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.addListener('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.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 sys = require('sys');

console.log(sys.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 read 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));
```

sys

These functions are in the module 'sys'. Use require('sys') to access them.
sys.print(string)

Like console.log() but without the trailing newline.

```
require('sys').print('String with no newline');
```

sys.debug(string)

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

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

sys.log(string)

Output with timestamp on stdout.

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

sys.inspect(object, showHidden, depth)

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 sys object:

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

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

sys.pump(readableStream, writeableStream, [callback])

Experimental

Read the data from readableStream and send it to the writableStream. When writeableStream.write(data) returns false readableStream will be paused until the drain event occurs on the writableStream. callback is called when writableStream is closed.

Timers

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

To schedule execution of callback after delay milliseconds. Returns a timeoutId for possible use with clearTimeout().

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.

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.

After this event is emitted, the 'output' and 'error' callbacks will no longer be made.

See waitpid(2).
child_process.spawn(command, args, env)

Launches a new process with the given command, command line arguments, and environment variables. If omitted, args defaults to an empty Array, and env defaults to process.env.

Example of running ls -lh /usr, capturing stdout, stderr, and the exit code:

```
var sys   = require('sys'),
    spawn = require('child_process').spawn,
    ls    = spawn('ls', ['-lh', '/usr']);

ls.stdout.addListener('data', function (data) {
  sys.print('stdout: ' + data);
});

ls.stderr.addListener('data', function (data) {
  sys.print('stderr: ' + data);
});

ls.addListener('exit', function (code) {
  console.log('child process exited with code ' + code);
});
```

Example of checking for failed exec:

```
var spawn = require('child_process').spawn,
    child = spawn('bad_command');

child.stderr.addListener('data', function (data) {
  if (/^execvp\(\)/.test(data.asciiSlice(0,data.length))) {
    console.log('Failed to start child process.');
  }
});
```

See also: child_process.exec()
child.kill(signal)

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.addListener('exit', function (code, signal) {
  console.log('child process terminated due to receipt of signal '+signal);
});

// send SIGHUP to process
grep.kill('SIGHUP');
```

Note that while the function is called kill, the signal delivered to the child process may not actually kill it. kill really just sends a signal to a process.

See kill(2)
child.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.stdin.write(data, encoding)

Write data to the child process's stdin. The second argument is optional and specifies the encoding: possible values are 'utf8', 'ascii', and 'binary'.

Example: A very elaborate way to run 'ps ax | grep ssh'

```
var sys   = require('sys'),
    spawn = require('child_process').spawn,
    ps    = spawn('ps', ['ax']),
    grep  = spawn('grep', ['ssh']);

ps.stdout.addListener('data', function (data) {
  grep.stdin.write(data);
});

ps.stderr.addListener('data', function (data) {
  sys.print('ps stderr: ' + data);
});

ps.addListener('exit', function (code) {
  if (code !== 0) {
```

```
    console.log('ps process exited with code ' + code);
  }
  grep.stdin.end();
});

grep.stdout.addListener('data', function (data) {
  sys.print(data);
});

grep.stderr.addListener('data', function (data) {
  sys.print('grep stderr: ' + data);
});

grep.addListener('exit', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});

child.stdin.end()
```

Closes the child process's stdin stream. This often causes the child process to terminate.

Example:

```
var spawn = require('child_process').spawn,
    grep  = spawn('grep', ['ssh']);

grep.addListener('exit', function (code) {
  console.log('child process exited with code ' + code);
});

grep.stdin.end();
```

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 sys   = require('sys'),
    exec  = require('child_process').exec,
    child;

child = exec('cat *.js bad_file | wc -l',
  function (error, stdout, stderr) {
    sys.print('stdout: ' + stdout);
    sys.print('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: 'SIGKILL'
}
```

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: 'SIGKILL'). maxBuffer specifies the largest amount of data allowed on stdout or stderr - if this value is exceeded then the child process is killed.

Script

Script class compiles and runs JavaScript code. You can access this class with:

```
var Script = process.binding('evals').Script;
```

New JavaScript code can be compiled and run immediately or compiled, saved, and run later.

Script.runInThisContext(code, filename)

Similar to process.compile. Script.runInThisContext compiles code as if it were loaded from filename, runs it and returns the result. Running code does not have access to local scope. filename is optional.

Example of using Script.runInThisContext and eval to run the same code:

```
var localVar = 123,
    usingscript, evaled,
    Script = process.binding('evals').Script;

usingscript = Script.runInThisContext('localVar = 1;',
  'myfile.js');
console.log('localVar: ' + localVar + ', usingscript: ' +
  usingscript);
evaled = eval('localVar = 1;');
console.log('localVar: ' + localVar + ', evaled: ' +
  evaled);

// localVar: 123, usingscript: 1
// localVar: 1, evaled: 1
```

Script.runInThisContext does not have access to the local scope, so localVar is unchanged. eval does have access to the local scope, so localVar is changed.

In case of syntax error in code, Script.runInThisContext emits the syntax error to stderr and throws.an exception.
Script.runInNewContext(code, sandbox, filename)

Script.runInNewContext compiles code to run in sandbox as if it were loaded from filename, then runs it and returns the result. Running code does not have access to local scope and the object sandbox will be used as the global object for code. sandbox and filename are optional.

Example: compile and execute code that increments a global variable and sets a new one. These globals are contained in the sandbox.

```
var sys = require('sys'),
    Script = process.binding('evals').Script,
    sandbox = {
      animal: 'cat',
      count: 2
    };

Script.runInNewContext(
  'count += 1; name = "kitty"', sandbox, 'myfile.js');
console.log(sys.inspect(sandbox));

// { animal: 'cat', count: 3, name: 'kitty' }
```

Note that running untrusted code is a tricky business requiring great care. To prevent accidental global variable leakage, Script.runInNewContext is quite useful, but safely running untrusted code requires a separate process.

In case of syntax error in code, Script.runInThisContext emits the syntax error to stderr and throws an exception.
new Script(code, filename)

new Script compiles code as if it were loaded from filename, but does not run it. Instead, it returns a Script object representing this compiled code. This script can be run later many times using methods below. The returned script is not bound to any global object. It is bound before each run, just for that run. filename is optional.

In case of syntax error in code, new Script emits the syntax error to stderr and throws an exception.
script.runInThisContext()

Similar to Script.runInThisContext (note capital 'S'), but now being a method of a precompiled Script object. script.runInThisContext runs the code of script and returns the result. Running code does not have access to local scope, but does have access to the global object (v8: in actual context).

Example of using script.runInThisContext to compile code once and run it multiple times:

```
var Script = process.binding('evals').Script,
    scriptObj, i;
```

```
globalVar = 0;

scriptObj = new Script('globalVar += 1', 'myfile.js');

for (i = 0; i < 1000 ; i += 1) {
  scriptObj.runInThisContext();
}

console.log(globalVar);

// 1000
```

script.runInNewContext(sandbox)

Similar to Script.runInNewContext (note capital 'S'), but now being a method of a precompiled Script object. script.runInNewContext runs the code of script with sandbox as the global object and returns the result. Running code does not have access to local scope. sandbox is optional.

Example: compile code that increments a global variable and sets one, then execute this code multiple times. These globals are contained in the sandbox.

```
var sys = require('sys'),
    Script = process.binding('evals').Script,
    scriptObj, i,
    sandbox = {
      animal: 'cat',
      count: 2
    };

scriptObj = new Script(
    'count += 1; name = "kitty"', 'myfile.js');

for (i = 0; i < 10 ; i += 1) {
  scriptObj.runInNewContext(sandbox);
}

console.log(sys.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

Note that running untrusted code is a tricky business requiring great care. To prevent accidental global variable leakage, script.runInNewContext is quite useful, but safely running untrusted code requires a separate process.

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), fs.lstat(path, callback), fs.fstat(fd, callback)

Asynchronous stat(2), lstat(2) or fstat(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.statSync(path), fs.lstatSync(path), fs.fstatSync(fd)

Synchronous stat(2), lstat(2) or 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(dstpath, srcpath)

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+'. 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 is null, 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.
fs.writeSync(fd, data, position, encoding)

Synchronous version of 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 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. 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 changes.

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.
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.

options is an object with the following defaults:

```
{ 'flags': 'r'
, 'encoding': 'binary'
, 'mode': 0666
, 'bufferSize': 4 * 1024
}
```

readStream.readable

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

Stops the stream from reading further data. No 'data' event will be fired until the stream is resumed.
readStream.resume()

Resumes the stream. Together with pause() this useful to throttle reading.
readStream.destroy()

Allows to close the stream before the 'end' is reached. No more events other than 'close' will be fired after this method has been called.
fs.WriteStream

WriteStream is a writable stream.
fs.createWriteStream(path, [options])

Returns a new WriteStream object. options is an object with the following defaults:

```
{ 'flags': 'w'
, 'encoding': 'binary'
, 'mode': 0666
}
```

writeStream.writeable

A boolean that is true by default, but turns false after an 'error' occurred or end() / destroy() was called.
writeStream.write(data, encoding='utf8')

Returns true if the data was flushed to the kernel, and false if it was queued up for being written later. A 'drain' will fire after all queued data has been written.

The second optional parameter specifies the encoding of for the string.
writeStream.end()

Closes the stream right after all queued write() calls have finished.
writeStream.destroy()

Allows to close the stream regardless of its current state.
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'
, 'stream': '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.

HTTPS is supported if OpenSSL is available on the underlying platform.
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.
http.createServer(requestListener, [options])

Returns a new web server object.

The options argument is optional. The options argument accepts the same values as the options argument for net.Server.

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

function (request, socket, head)

Emitted each time a client requests a http upgrade. If this event isn't listened for, then clients requesting an upgrade will have their connections closed.

* request is the arguments for the http request, as it is in the request event.
* socket is the network socket between the server and client.
* head is an instance of Buffer, the first packet of the upgraded stream, this may be empty.

After this event is emitted, the request's socket will not have a data event listener, meaning you will need to bind to it in order to handle data sent to the server on that socket.
Event: 'clientError'

function (exception) {}

If a client connection emits an 'error' event - it will forwarded here.
server.listen(port, hostname=null, callback=null)

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=null)

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.setSecure(credentials)

Enables HTTPS support for the server, with the crypto module credentials specifying the private key and certificate of the server, and optionally the CA certificates for use in client authentication.

If the credentials hold one or more CA certificates, then the server will request for the client to submit a client certificate as part of the HTTPS connection handshake. The validity and content of this can be accessed via verifyPeer() and getPeerCertificate() from the server's request.connection.
server.close()

Stops the server from accepting new connections.
http.ServerRequest
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.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='binary')

Set the encoding for the request body. Either 'utf8' or 'binary'. Defaults to 'binary'.
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 assocated 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.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.
response.write(chunk, encoding)

This method must be called after writeHead was called. It sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

If chunk is a string, the second parameter specifies how to encode it into a byte stream. By default the encoding is 'ascii'.

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.end()

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.
http.Client

An HTTP client is constructed with a server address as its argument, the returned handle is then used to issue one or more requests. Depending on the server connected to, the client might pipeline the requests or reestablish the stream after each stream. Currently the implementation does not pipeline requests.

Example of connecting to google.com:

```
var http = require('http');
var google = http.createClient(80, 'www.google.com');
var request = google.request('GET', '/',
  {'host': 'www.google.com'});
request.end();
request.addListener('response', function (response) {
  console.log('STATUS: ' + response.statusCode);
  console.log('HEADERS: ' + JSON.stringify(response.headers));
```

```
  response.setEncoding('utf8');
  response.addListener('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});
```

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.
http.createClient(port, host, secure, credentials)

Constructs a new HTTP client. port and host refer to the server to be connected to. A stream is not established until a request is issued.

secure is an optional boolean flag to enable https support and credentials is an optional credentials object from the crypto module, which may hold the client's private key, certificate, and a list of trusted CA certificates.

If the connection is secure, but no explicit CA certificates are passed in the credentials, then node.js will default to the publicly trusted list of CA certificates, as given in http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt
client.request([method], path, [request_headers])

Issues a request; if necessary establishes stream. Returns a http.ClientRequest instance.

method is optional and defaults to 'GET' if omitted.

request_headers is optional. Additional request headers might be added internally by Node. Returns a ClientRequest object.

Do remember to include the Content-Length header if you plan on sending a body. If you plan on streaming the body, perhaps set Transfer-Encoding: chunked.

NOTE: the request is not complete. This method only sends the header of the request. One needs to call request.end() to finalize the request and retrieve the response. (This sounds convoluted but it provides a chance for the user to stream a body to the server with request.write().)
client.verifyPeer()

Returns true or false depending on the validity of the server's certificate in the context of the defined or default list of trusted CA certificates.
client.getPeerCertificate()

Returns a JSON structure detailing the server's certificate, containing a dictionary with keys for the certificate 'subject', 'issuer', 'valid_from' and 'valid_to'

http.ClientRequest

This object is created internally and returned from the request() method of a http.Client. It represents an in-progress request whose header has already been sent.

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.addListener('response', function (response) {
  response.addListener('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

// Bad - misses all or part of the body
request.addListener('response', function (response) {
  setTimeout(function () {
    response.addListener('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='ascii')

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. The encoding argument should be either 'utf8' or 'ascii'. By default the body uses ASCII encoding, as it is faster.

request.end()

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'.
http.ClientResponse

This object is created when making a request with http.Client. 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.

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

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.
response.setEncoding(encoding)

Set the encoding for the response body. Either 'utf8' or 'binary'. Defaults to 'binary'.
response.pause()

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

Resumes a paused response.
response.client

A reference to the http.Client that this response belongs to.

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 (stream) {
  stream.setEncoding('utf8');
  stream.addListener('connect', function () {
    stream.write('hello\r\n');
  });
  stream.addListener('data', function (data) {
    stream.write(data);
  });
  stream.addListener('end', function () {
    stream.write('goodbye\r\n');
    stream.end();
  });
});
server.listen(8124, 'localhost');
```

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

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

This is an EventEmitter with the following events:
Event: 'connection'

```
function (stream) {}
```

Emitted when a new connection is made. stream is an instance of net.Stream.
Event: 'close'

```
function () {}
```

Emitted when the server closes.
net.createServer(connectionListener)

Creates a new TCP server. The connection_listener argument is automatically set as a listener for the 'connection' event.
server.listen(port, host=null, callback=null)

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.
server.listen(path, callback=null)

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.
net.Stream

This object is an abstraction of of a TCP or UNIX socket. net.Stream instance 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.Stream instances are an EventEmitters with the following events:
Event: 'connect'

function () { }

Emitted when a stream connection successfully is established. See connect().
Event: 'secure'

function () { }

Emitted when a stream connection successfully establishes a HTTPS handshake with its peer.
Event: 'data'

function (data) { }

Emitted when data is received. The argument data will be a Buffer or String. Encoding of data is set by stream.setEncoding(). (See the section on Readable Streams for more infromation.)
Event: 'end'

function () { }

Emitted when the other end of the stream sends a FIN packet. After this is emitted the readyState will be 'writeOnly'. One should probably just call stream.end() when this event is emitted.
Event: 'timeout'

function () { }

Emitted if the stream times out from inactivity. This is only to notify that the stream has been idle. The user must manually close the connection.

See also: stream.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 () { }

Emitted once the stream is fully closed. The argument had_error is a boolean which says if the stream was closed due to a transmission error.
net.createConnection(port, host='127.0.0.1')

Construct a new stream object and opens a stream to the specified port and host. If the second parameter is omitted, localhost is assumed.

When the stream is established the 'connect' event will be emitted.
stream.connect(port, host='127.0.0.1')

Opens a stream to the specified port and host. createConnection() also opens a stream; normally this method is not needed. Use this only if a stream is closed and you want to reuse the object to connect to another server.

This function is asynchronous. When the 'connect' event is emitted the stream is established. If there is a problem connecting, the 'connect' event will not be emitted, the 'error' event will be emitted with the exception.
stream.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.
stream.readyState

Either 'closed', 'open', 'opening', 'readOnly', or 'writeOnly'.
stream.setEncoding(encoding)

Sets the encoding (either 'ascii', 'utf8', or 'binary') for data that is received.
stream.setSecure(credentials)

Enables HTTPS support for the stream, with the crypto module credentials specifying the private key and certificate of the stream, and optionally the CA certificates for use in peer authentication.

If the credentials hold one ore more CA certificates, then the stream will request for the peer to submit a client certificate as part of the HTTPS connection handshake. The validity and content of this can be accessed via verifyPeer() and getPeerCertificate().
stream.verifyPeer()

Returns true or false depending on the validity of the peers's certificate in the context of the defined or default list of trusted CA certificates.
stream.getPeerCertificate()

Returns a JSON structure detailing the peer's certificate, containing a dictionary with keys for the certificate 'subject', 'issuer', 'valid_from' and 'valid_to'
stream.write(data, encoding='ascii')

Sends data on the stream. The second parameter specifies the encoding in the case of a string--it defaults to ASCII because encoding to UTF8 is rather slow.

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.
stream.end()

Half-closes the stream. I.E., it sends a FIN packet. It is possible the server will still send some data. After calling this readyState will be 'readOnly'.
stream.destroy()

Ensures that no more I/O activity happens on this stream. Only necessary in case of errors (parse error or so).
stream.pause()

Pauses the reading of data. That is, 'data' events will not be emitted. Useful to throttle back an upload.
stream.resume()

Resumes reading after a call to pause().
stream.setTimeout(timeout)

Sets the stream to timeout after timeout milliseconds of inactivity on the stream. By default net.Stream do not have a timeout.

When an idle timeout is triggered the stream will receive a 'timeout' event but the connection will not be severed. The user must manually end() or destroy() the stream.

If timeout is 0, then the existing idle timeout is disabled.
stream.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 stream.write() is called.
stream.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 stream. 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.

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 in http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt

crypto.createHash(algorithm)

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

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

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)

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)

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, output_encoding)

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)

Returns any remaining enciphered contents, with output_encoding as update above.
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, output_encoding)

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 - either 'binary', 'ascii' or 'utf8'.
decipher.final(output_encoding)

Returns any remaining plaintext which is deciphered, with `output_encoding' as update above.
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)

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 verifyer object with data. This can be called many times with new data as it is streamed.
verifier.verify(public_key, signature, signature_format)

Verifies the signed data by using the public_key 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.
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));

  for (var i = 0; i < addresses.length; i++) {
    var a = addresses[i];
    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.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.

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.
assert.doesNotThrow(block, error, message)

Expects block not to throw an error.
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.
Path

This module contains utilities for dealing with file paths. Use require('path') to use it. It provides the following methods:
path.join(/ path1, path2, ... /)

Join all arguments together and resolve the resulting path. Example:

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

path.normalizeArray(arr)

Normalize an array of path parts, taking care of '..' and '.' parts. Example:

```
path.normalizeArray(['',
  'foo', 'bar', 'baz', 'asdf', 'quux', '..'])
// returns
[ '', 'foo', 'bar', 'baz', 'asdf' ]
```

path.normalize(p)

Normalize a string path, taking care of '..' and '.' parts. Example:

```
path.normalize('/foo/bar/baz/asdf/quux/..')
// returns
'/foo/bar/baz/asdf'
```

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) {
  sys.debug(exists ? "it's there" : "no passwd!");
});
```

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='=', munge=true)

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'
```

By default, this function will perform PHP/Rails-style parameter mungeing for arrays and objects used as values within obj. Example:

```
querystring.stringify({foo: 'bar', foo: 'baz', foo: 'boz'})
```

```
// returns
'foo=boz'

querystring.stringify({foo: {bar: 'baz'}})
// returns
'foo[bar]=baz'
```

If you wish to disable the array mungeing (e.g. when generating parameters for a Java servlet), you can set the munge argument to false. Example:

```
querystring.stringify({foo: 'bar', foo: 'baz', foo: 'boz'}, '&', '=', false)
// returns
'foo=bar&foo=baz&foo=boz'
```

Note that when munge is false, parameter names with object values will still be munged.
querystring.parse(str, sep='&', eq='=')

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

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

This function can parse both munged and unmunged query strings (see stringify for details).
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-editting.

```
mjr:~$ node
Type '.help' for options.
node> a = [ 1, 2, 3];
[ 1, 2, 3 ]
node> 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)
```

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

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.

```
node> [ "a", "b", "c" ]
[ 'a', 'b', 'c' ]
node> _.length
3
node> _ += 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 scope object associated with each REPLServer. For example:

```
// repl_test.js
var repl = require("repl"),
    msg = "message";

repl.start().scope.m = msg;
```

Things in the scope object appear as local within the REPL:

```
mjr:~$ node repl_test.js
node> 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 scope 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.

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');
```

```
console.log( 'The area of a circle of radius 4 is '
        + circle.area(4));
```

The contents of circle.js:

```
var PI = 3.14;

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. (Alternatively, one can use this instead of exports.) Variables local to the module will be private. In this example the variable PI is private to circle.js. The function puts() comes from the module 'sys', which is a built-in module. Modules which are not prefixed by './' are built-in module--more about this later.

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 the leading './', like require('assert') the module is searched for in the require.paths array. require.paths on my system looks like this:

```
[ '/home/ryan/.node_libraries' ]
```

That is, when require('assert') is called Node looks for:

* 1: /home/ryan/.node_libraries/assert.js
* 2: /home/ryan/.node_libraries/assert.node
* 3: /home/ryan/.node_libraries/assert/index.js
* 4: /home/ryan/.node_libraries/assert/index.node

interrupting once a file is found. Files ending in '.node' are binary Addon Modules; see 'Addons' below. 'index.js' allows one to package a module as a directory.

require.paths can be modified at runtime by simply unshifting new paths onto it, or at startup with the NODE_PATH environmental variable (which should be a list of paths, colon separated).

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 http:/cvs.schmorp.de/libev/ev.html[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 http://code.google.com/p/waf/[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 http://github.com/ry/node_postgres for a real example.