

Node.js v6.3.0 Documentation

[Index](#) | [View on single page](#) | [View as JSON](#)

Table of Contents

- [About this Documentation](#)
 - [Stability Index](#)
 - [JSON Output](#)
 - [Syscalls and man pages](#)
- [Usage](#)
 - [Example](#)
- [Addons](#)
 - [Hello world](#)
 - [Building](#)
 - [Linking to Node.js' own dependencies](#)
 - [Loading Addons using require\(\)](#)
 - [Native Abstractions for Node.js](#)
 - [Addon examples](#)
 - [Function arguments](#)
 - [Callbacks](#)
 - [Object factory](#)
 - [Function factory](#)
 - [Wrapping C++ objects](#)
 - [Factory of wrapped objects](#)
 - [Passing wrapped objects around](#)
 - [AtExit hooks](#)
 - `void AtExit(callback, args)`
- [Assert](#)
 - `assert(value[, message])`
 - `assert.deepEqual(actual, expected[, message])`
 - `assert.deepStrictEqual(actual, expected[, message])`
 - `assert.doesNotThrow(block[, error][, message])`

- `assert.equal(actual, expected[, message])`
- `assert.fail(actual, expected, message, operator)`
- `assert.ifError(value)`
- `assert.notDeepEqual(actual, expected[, message])`
- `assert.notDeepStrictEqual(actual, expected[, message])`
- `assert.notEqual(actual, expected[, message])`
- `assert.notStrictEqual(actual, expected[, message])`
- `assert.ok(value[, message])`
- `assert.strictEqual(actual, expected[, message])`
- `assert.throws(block[, error][, message])`
- Buffer
 - `Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()`
 - The --zero-fill-buffers command line option
 - What makes `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` "unsafe"?
 - Buffers and Character Encodings
 - Buffers and `TypedArray`
 - Buffers and ES6 iteration
 - Class: `Buffer`
 - `new Buffer(array)`
 - `new Buffer(buffer)`
 - `new Buffer(arrayBuffer[, byteOffset [, length]])`
 - `new Buffer(size)`
 - `new Buffer(str[, encoding])`
 - Class Method: `Buffer.alloc(size[, fill[, encoding]])`
 - Class Method: `Buffer.allocUnsafe(size)`
 - Class Method: `Buffer.allocUnsafeSlow(size)`
 - Class Method: `Buffer.byteLength(string[, encoding])`
 - Class Method: `Buffer.compare(buf1, buf2)`
 - Class Method: `Buffer.concat(list[, totalLength])`
 - Class Method: `Buffer.from(array)`
 - Class Method: `Buffer.from(arrayBuffer[, byteOffset[, length]])`
 - Class Method: `Buffer.from(buffer)`

- Class Method: `Buffer.from(str[, encoding])`
- Class Method: `Buffer.isBuffer(obj)`
- Class Method: `Buffer.isEncoding(encoding)`
- `buf[index]`
- `buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])`
- `buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])`
- `buf.entries()`
- `buf.equals(otherBuffer)`
- `buf.fill(value[, offset[, end]][], encoding)`
- `buf.indexOf(value[, byteOffset][], encoding)`
- `buf.includes(value[, byteOffset][], encoding)`
- `buf.keys()`
- `buf.lastIndexOf(value[, byteOffset][], encoding)`
- `buf.length`
- `buf.readDoubleBE(offset[, noAssert])`
- `buf.readDoubleLE(offset[, noAssert])`
- `buf.readFloatBE(offset[, noAssert])`
- `buf.readFloatLE(offset[, noAssert])`
- `buf.readInt8(offset[, noAssert])`
- `buf.readInt16BE(offset[, noAssert])`
- `buf.readInt16LE(offset[, noAssert])`
- `buf.readInt32BE(offset[, noAssert])`
- `buf.readInt32LE(offset[, noAssert])`
- `buf.readIntBE(offset, byteLength[, noAssert])`
- `buf.readIntLE(offset, byteLength[, noAssert])`
- `buf.readUInt8(offset[, noAssert])`
- `buf.readUInt16BE(offset[, noAssert])`
- `buf.readUInt16LE(offset[, noAssert])`
- `buf.readUInt32BE(offset[, noAssert])`
- `buf.readUInt32LE(offset[, noAssert])`
- `buf.readUIntBE(offset, byteLength[, noAssert])`
- `buf.readUIntLE(offset, byteLength[, noAssert])`
- `buf.slice([start[, end]])`

- `buf.swap16()`
- `buf.swap32()`
- `buf.swap64()`
- `buf.toString([encoding[, start[, end]]])`
- `buf.toJSON()`
- `buf.values()`
- `buf.write(string[, offset[, length]][], encoding)`
- `buf.writeDoubleBE(value, offset[, noAssert])`
- `buf.writeDoubleLE(value, offset[, noAssert])`
- `buf.writeFloatBE(value, offset[, noAssert])`
- `buf.writeFloatLE(value, offset[, noAssert])`
- `buf.writeInt8(value, offset[, noAssert])`
- `buf.writeInt16BE(value, offset[, noAssert])`
- `buf.writeInt16LE(value, offset[, noAssert])`
- `buf.writeInt32BE(value, offset[, noAssert])`
- `buf.writeInt32LE(value, offset[, noAssert])`
- `buf.writeIntBE(value, offset, byteLength[, noAssert])`
- `buf.writeIntLE(value, offset, byteLength[, noAssert])`
- `buf.writeUInt8(value, offset[, noAssert])`
- `buf.writeUInt16BE(value, offset[, noAssert])`
- `buf.writeUInt16LE(value, offset[, noAssert])`
- `buf.writeUInt32BE(value, offset[, noAssert])`
- `buf.writeUInt32LE(value, offset[, noAssert])`
- `buf.writeUIntBE(value, offset, byteLength[, noAssert])`
- `buf.writeUIntLE(value, offset, byteLength[, noAssert])`
- `buffer.INSPECT_MAX_BYTES`
- Class: `SlowBuffer`
 - `new SlowBuffer(size)`
- Child Process
 - Asynchronous Process Creation
 - Spawning .bat and .cmd files on Windows
 - `child_process.exec(command[, options][, callback])`
 - `child_process.execFile(file[, args][, options][, callback])`

- `child_process.fork(modulePath[, args][, options])`
- `child_process.spawn(command[, args][, options])`
 - `options.detached`
 - `options.stdio`
- Synchronous Process Creation
 - `child_process.execFileSync(file[, args][, options])`
 - `child_process.execSync(command[, options])`
 - `child_process.spawnSync(command[, args][, options])`
- Class: ChildProcess
 - `Event: 'close'`
 - `Event: 'disconnect'`
 - `Event: 'error'`
 - `Event: 'exit'`
 - `Event: 'message'`
 - `child.connected`
 - `child.disconnect()`
 - `child.kill([signal])`
 - `child.pid`
 - `child.send(message[, sendHandle[, options]][, callback])`
 - Example: sending a server object
 - Example: sending a socket object
 - `child.stderr`
 - `child.stdin`
 - `child.stdio`
 - `child.stdout`
- `maxBuffer` and `Unicode`
- Cluster
 - How It Works
 - Class: Worker
 - `Event: 'disconnect'`
 - `Event: 'error'`
 - `Event: 'exit'`
 - `Event: 'listening'`

- Event: 'message'
- Event: 'online'
- worker.disconnect()
- worker.exitedAfterDisconnect
- worker.id
- worker.isConnected()
- worker.isDead()
- worker.kill([signal='SIGTERM'])
- worker.process
- worker.send(message[, sendHandle][, callback])
- worker.suicide
- Event: 'disconnect'
- Event: 'exit'
- Event: 'fork'
- Event: 'listening'
- Event: 'message'
- Event: 'online'
- Event: 'setup'
- cluster.disconnect([callback])
- cluster.fork([env])
- cluster.isMaster
- cluster.isWorker
- cluster.schedulingPolicy
- cluster.settings
- cluster.setupMaster([settings])
- cluster.worker
- cluster.workers
- Command Line Options
 - Synopsis
 - Options
 - -v, --version
 - -h, --help
 - -e, --eval "script"

- `-p, --print "script"`
- `-c, --check`
- `-i, --interactive`
- `-r, --require module`
- `--no-deprecation`
- `--trace-deprecation`
- `--throw-deprecation`
- `--no-warnings`
- `--trace-warnings`
- `--trace-sync-io`
- `--zero-fill-buffers`
- `--preserve-symlinks`
- `--track-heap-objects`
- `--prof-process`
- `--v8-options`
- `--tls-cipher-list=list`
- `--enable-fips`
- `--force-fips`
- `--icu-data-dir=file`
- Environment Variables
 - `NODE_DEBUG=module[,...]`
 - `NODE_PATH=path[:...]`
 - `NODE_DISABLE_COLORS=1`
 - `NODE_ICU_DATA=file`
 - `NODE_REPL_HISTORY=file`
- Console
 - Asynchronous vs Synchronous Consoles
 - Class: `Console`
 - `new Console(stdout[, stderr])`
 - `console.assert(value[, message][, ...])`
 - `console.dir(obj[, options])`

- `console.error([data][, ...])`
- `console.info([data][, ...])`
- `console.log([data][, ...])`
- `console.time(label)`
- `console.timeEnd(label)`
- `console.trace(message[, ...])`
- `console.warn([data][, ...])`

- **Crypto**

- Determining if crypto support is unavailable
- Class: Certificate
 - `new crypto.Certificate()`
 - `certificate.exportChallenge(spkac)`
 - `certificate.exportPublicKey(spkac)`
 - `certificate.verifySpkac(spkac)`
- Class: Cipher
 - `cipher.final([output_encoding])`
 - `cipher.setAAD(buffer)`
 - `cipher.getAuthTag()`
 - `cipher.setAutoPadding(auto_padding=true)`
 - `cipher.update(data[, input_encoding][, output_encoding])`
- Class: Decipher
 - `decipher.final([output_encoding])`
 - `decipher.setAAD(buffer)`
 - `decipher.setAuthTag(buffer)`
 - `decipher.setAutoPadding(auto_padding=true)`
 - `decipher.update(data[, input_encoding][, output_encoding])`
- Class: DiffieHellman
 - `diffieHellman.computeSecret(other_public_key[, input_encoding][, output_encoding])`
 - `diffieHellman.generateKeys([encoding])`
 - `diffieHellman.getGenerator([encoding])`
 - `diffieHellman.getPrime([encoding])`
 - `diffieHellman.getPrivateKey([encoding])`
 - `diffieHellman.getPublicKey([encoding])`

- `diffieHellman.setPrivateKey(private_key[, encoding])`
- `diffieHellman.setPublicKey(public_key[, encoding])`
- `diffieHellman.verifyError`
- Class: ECDH
 - `ecdh.computeSecret(other_public_key[, input_encoding][, output_encoding])`
 - `ecdh.generateKeys([encoding[, format]])`
 - `ecdh.getPrivateKey([encoding])`
 - `ecdh.getPublicKey([encoding[, format]])`
 - `ecdh.setPrivateKey(private_key[, encoding])`
 - `ecdh.setPublicKey(public_key[, encoding])`
- Class: Hash
 - `hash.digest([encoding])`
 - `hash.update(data[, input_encoding])`
- Class: Hmac
 - `hmac.digest([encoding])`
 - `hmac.update(data[, input_encoding])`
- Class: Sign
 - `sign.sign(private_key[, output_format])`
 - `sign.update(data[, input_encoding])`
- Class: Verify
 - `verifier.update(data[, input_encoding])`
 - `verifier.verify(object, signature[, signature_format])`
- crypto module methods and properties
- crypto.constants
 - `crypto.DEFAULT_ENCODING`
 - `crypto.fips`
 - `crypto.createCipher(algorithm, password)`
 - `crypto.createCipheriv(algorithm, key, iv)`
 - `crypto.createCredentials(details)`
 - `crypto.createDecipher(algorithm, password)`
 - `crypto.createDecipheriv(algorithm, key, iv)`
 - `crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding])`

- `crypto.createDiffieHellman(prime_length[, generator])`
- `crypto.createECDH(curve_name)`
- `crypto.createHash(algorithm)`
- `crypto.createHmac(algorithm, key)`
- `crypto.createSign(algorithm)`
- `crypto.createVerify(algorithm)`
- `crypto.getCiphers()`
- `crypto.getCurves()`
- `crypto.getDiffieHellman(group_name)`
- `crypto.getHashes()`
- `crypto.pbkdf2(password, salt, iterations, keylen, digest, callback)`
- `crypto.pbkdf2Sync(password, salt, iterations, keylen, digest)`
- `crypto.privateDecrypt(private_key, buffer)`
- `crypto.privateEncrypt(private_key, buffer)`
- `crypto.publicDecrypt(public_key, buffer)`
- `crypto.publicEncrypt(public_key, buffer)`
- `crypto.randomBytes(size[, callback])`
- `crypto.setEngine(engine[, flags])`
- Notes
 - Legacy Streams API (pre Node.js v0.10)
 - Recent ECDH Changes
 - Support for weak or compromised algorithms
- Crypto Constants
 - OpenSSL Options
 - OpenSSL Engine Constants
 - Other OpenSSL Constants
 - Node.js Crypto Constants
- Debugger
 - Watchers
 - Command reference
 - Stepping
 - Breakpoints
 - Information

- Execution control
- Various
- Advanced Usage
- V8 Inspector Integration for Node.js
- UDP / Datagram Sockets
 - Class: `dgram.Socket`
 - Event: 'close'
 - Event: 'error'
 - Event: 'listening'
 - Event: 'message'
 - `socket.addMembership(multicastAddress[, multicastInterface])`
 - `socket.address()`
 - `socket.bind([port][, address][, callback])`
 - `socket.bind(options[, callback])`
 - `socket.close([callback])`
 - `socket.dropMembership(multicastAddress[, multicastInterface])`
 - `socket.send(msg, [offset, length,] port, address[, callback])`
 - `socket.setBroadcast(flag)`
 - `socket.setMulticastLoopback(flag)`
 - `socket.setMulticastTTL(ttl)`
 - `socket.setTTL(ttl)`
 - `socket.ref()`
 - `socket.unref()`
 - Change to asynchronous `socket.bind()` behavior
 - `dgram` module functions
 - `dgram.createSocket(options[, callback])`
 - `dgram.createSocket(type[, callback])`
- DNS
 - `dns.getServers()`
 - `dns.lookup(hostname[, options], callback)`
 - Supported `getaddrinfo` flags
 - `dns.lookupService(address, port, callback)`
 - `dns.resolve(hostname[, rrtype], callback)`

- `dns.resolve4(hostname, callback)`
- `dns.resolve6(hostname, callback)`
- `dns.resolveCname(hostname, callback)`
- `dns.resolveMx(hostname, callback)`
- `dns.resolveNaptr(hostname, callback)`
- `dns.resolveNs(hostname, callback)`
- `dns.resolveSoa(hostname, callback)`
- `dns.resolveSrv(hostname, callback)`
- `dns.resolvePtr(hostname, callback)`
- `dns.resolveTxt(hostname, callback)`
- `dns.reverse(ip, callback)`
- `dns.setServers(servers)`
- Error codes
- Implementation considerations
 - `dns.lookup()`
 - `dns.resolve()`, `dns.resolve*()` and `dns.reverse()`
- Domain
 - Warning: Don't Ignore Errors!
 - Additions to Error objects
 - Implicit Binding
 - Explicit Binding
 - `domain.create()`
 - Class: Domain
 - `domain.run(fn[, arg][, ...])`
 - `domain.members`
 - `domain.add(emitter)`
 - `domain.remove(emitter)`
 - `domain.bind(callback)`
 - Example
 - `domain.intercept(callback)`
 - Example
 - `domain.enter()`
 - `domain.exit()`

- domain.dispose()
- Errors
 - Error Propagation and Interception
 - Node.js style callbacks
 - Class: Error
 - new Error(message)
 - Error.captureStackTrace(targetObject[, constructorOpt])
 - Error.stackTraceLimit
 - error.message
 - error.stack
 - Class: RangeError
 - Class: ReferenceError
 - Class: SyntaxError
 - Class: TypeError
 - Exceptions vs. Errors
 - System Errors
 - Class: System Error
 - error.code
 - error.errno
 - error.syscall
 - Common System Errors
- Events
 - Passing arguments and this to listeners
 - Asynchronous vs. Synchronous
 - Handling events only once
 - Error events
 - Class: EventEmitter
 - Event: 'newListener'
 - Event: 'removeListener'
 - EventEmitter.listenerCount(emitter, eventName)
 - EventEmitter.defaultMaxListeners
 - emitter.addListener(eventName, listener)
 - emitter.emit(eventName[, arg1][, arg2][, ...])

- `emitter.eventNames()`
- `emitter.getMaxListeners()`
- `emitter.listenerCount(eventName)`
- `emitter.listeners(eventName)`
- `emitter.on(eventName, listener)`
- `emitter.once(eventName, listener)`
- `emitter.prependListener(eventName, listener)`
- `emitter.prependOnceListener(eventName, listener)`
- `emitter.removeAllListeners([eventName])`
- `emitter.removeListener(eventName, listener)`
- `emitter.setMaxListeners(n)`

- **File System**

- **Buffer API**
- **Class: fs.FSWatcher**
 - `Event: 'change'`
 - `Event: 'error'`
 - `watcher.close()`
- **Class: fs.ReadStream**
 - `Event: 'open'`
 - `Event: 'close'`
 - `readStream.path`
- **Class: fs.Stats**
 - `Stat Time Values`
- **Class: fs.WriteStream**
 - `Event: 'open'`
 - `Event: 'close'`
 - `writeStream.bytesWritten`
 - `writeStream.path`
- `fs.access(path[, mode], callback)`
- `fs.accessSync(path[, mode])`
- `fs.appendFile(file, data[, options], callback)`
- `fs.appendFileSync(file, data[, options])`
- `fs.chmod(path, mode, callback)`

- `fs.chmodSync(path, mode)`
- `fs.chown(path, uid, gid, callback)`
- `fs.chownSync(path, uid, gid)`
- `fs.close(fd, callback)`
- `fs.closeSync(fd)`
- `fs.constants`
- `fs.createReadStream(path[, options])`
- `fs.createWriteStream(path[, options])`
- `fs.exists(path, callback)`
- `fs.existsSync(path)`
- `fs.fchmod(fd, mode, callback)`
- `fs.fchmodSync(fd, mode)`
- `fs.fchown(fd, uid, gid, callback)`
- `fs.fchownSync(fd, uid, gid)`
- `fs.fdatasync(fd, callback)`
- `fs.fdatasyncSync(fd)`
- `fs.fstat(fd, callback)`
- `fs.fstatSync(fd)`
- `fs.fsync(fd, callback)`
- `fs.fsyncSync(fd)`
- `fs.ftruncate(fd, len, callback)`
- `fs.ftruncateSync(fd, len)`
- `fs.futimes(fd, atime, mtime, callback)`
- `fs.futimesSync(fd, atime, mtime)`
- `fs.lchmod(path, mode, callback)`
- `fs.lchmodSync(path, mode)`
- `fs.lchown(path, uid, gid, callback)`
- `fs.lchownSync(path, uid, gid)`
- `fs.link(srcpath, dstpath, callback)`
- `fs.linkSync(srcpath, dstpath)`
- `fs.lstat(path, callback)`
- `fs.lstatSync(path)`
- `fs.mkdir(path[, mode], callback)`

- `fs.mkdirSync(path[, mode])`
- `fs.mkdtemp(prefix, callback)`
- `fs.mkdtempSync(prefix)`
- `fs.open(path, flags[, mode], callback)`
- `fs.openSync(path, flags[, mode])`
- `fs.read(fd, buffer, offset, length, position, callback)`
- `fs.readdir(path[, options], callback)`
- `fs.readdirSync(path[, options])`
- `fs.readFile(file[, options], callback)`
- `fs.readFileSync(file[, options])`
- `fs.readlink(path[, options], callback)`
- `fs.readlinkSync(path[, options])`
- `fs.readSync(fd, buffer, offset, length, position)`
- `fs.realpath(path[, options], callback)`
- `fs.realpathSync(path[, options])`
- `fs.rename(oldPath, newPath, callback)`
- `fs.renameSync(oldPath, newPath)`
- `fs.rmdir(path, callback)`
- `fs.rmdirSync(path)`
- `fs.stat(path, callback)`
- `fs.statSync(path)`
- `fs.symlink(target, path[, type], callback)`
- `fs.symlinkSync(target, path[, type])`
- `fs.truncate(path, len, callback)`
- `fs.truncateSync(path, len)`
- `fs.unlink(path, callback)`
- `fs.unlinkSync(path)`
- `fs.unwatchFile(filename[, listener])`
- `fs.utimes(path, atime, mtime, callback)`
- `fs.utimesSync(path, atime, mtime)`
- `fs.watch(filename[, options][, listener])`

- **Caveats**

- **Availability**

- Inodes
- Filename Argument
- fs.watchFile(filename[, options], listener)
- fs.write(fd, buffer, offset, length[, position], callback)
- fs.write(fd, data[, position[, encoding]], callback)
- fs.writeFileSync(file, data[, options], callback)
- fs.writeFileSync(file, data[, options])
- fs.writeFileSync(fd, buffer, offset, length[, position])
- fs.writeFileSync(fd, data[, position[, encoding]])
- FS Constants
 - File Access Constants
 - File Open Constants
 - File Type Constants
 - File Mode Constants
- [AHAFS]: https://www.ibm.com/developerworks/aix/library/au-aix_event_infrastructure/
- Global Objects
 - Class: Buffer
 - _dirname
 - _filename
 - clearImmediate(immediateObject)
 - clearInterval(intervalObject)
 - clearTimeout(timeoutObject)
 - console
 - exports
 - global
 - module
 - process
 - require()
 - require.cache
 - require.extensions
 - require.resolve()
 - setImmediate(callback[, arg][, ...])
 - setInterval(callback, delay[, arg][, ...])

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

- **HTTP**

- **Class: http.Agent**

- `new Agent([options])`
 - `agent.createConnection(options[, callback])`
 - `agent.destroy()`
 - `agent.freeSockets`
 - `agent.getName(options)`
 - `agent.maxFreeSockets`
 - `agent.maxSockets`
 - `agent.requests`
 - `agent.sockets`

- **Class: http.ClientRequest**

- `Event: 'abort'`
 - `Event: 'aborted'`
 - `Event: 'checkExpectation'`
 - `Event: 'connect'`
 - `Event: 'continue'`
 - `Event: 'response'`
 - `Event: 'socket'`
 - `Event: 'upgrade'`
 - `request.abort()`
 - `request.end([data][, encoding][, callback])`
 - `request.flushHeaders()`
 - `request.setNoDelay([noDelay])`
 - `request.setSocketKeepAlive([enable][, initialDelay])`
 - `request.setTimeout(timeout[, callback])`
 - `request.write(chunk[, encoding][, callback])`

- **Class: http.Server**

- `Event: 'checkContinue'`
 - `Event: 'clientError'`
 - `Event: 'close'`
 - `Event: 'connect'`

- Event: 'connection'
- Event: 'request'
- Event: 'upgrade'
- `server.close([callback])`
- `server.listen(handle[, callback])`
- `server.listen(path[, callback])`
- `server.listen(port[, hostname][, backlog][, callback])`
- `server.listening`
- `server.maxHeadersCount`
- `server.setTimeout(msecs, callback)`
- `server.timeout`
- Class: `http.ServerResponse`
 - Event: 'close'
 - Event: 'finish'
 - `response.addTrailers(headers)`
 - `response.end([data][, encoding][, callback])`
 - `response.finished`
 - `response.getHeader(name)`
 - `response.headersSent`
 - `response.removeHeader(name)`
 - `response.sendDate`
 - `response.setHeader(name, value)`
 - `response.setTimeout(msecs, callback)`
 - `response.statusCode`
 - `response.statusMessage`
 - `response.write(chunk[, encoding][, callback])`
 - `response.writeContinue()`
 - `response.writeHead(statusCode[, statusMessage][, headers])`
- Class: `http.IncomingMessage`
 - Event: 'aborted'
 - Event: 'close'
 - `message.destroy([error])`
 - `message.headers`

- `message.httpVersion`
- `message.method`
- `message.rawHeaders`
- `message.rawTrailers`
- `message.setTimeout(msecs, callback)`
- `message.statusCode`
- `message.statusMessage`
- `message.socket`
- `message.trailers`
- `message.url`
- `http.METHODS`
- `http.STATUS_CODES`
- `http.createClient([port][, host])`
- `http.createServer([requestListener])`
- `http.get(options[, callback])`
- `http.globalAgent`
- `http.request(options[, callback])`
- **HTTPS**
 - Class: `https.Agent`
 - Class: `https.Server`
 - `server.setTimeout(msecs, callback)`
 - `server.timeout`
 - `https.createServer(options[, requestListener])`
 - `server.close([callback])`
 - `server.listen(handle[, callback])`
 - `server.listen(path[, callback])`
 - `server.listen(port[, host][, backlog][, callback])`
 - `https.get(options, callback)`
 - `https.globalAgent`
 - `https.request(options, callback)`
- **Modules**
 - Accessing the main module
 - Addenda: Package Manager Tips

- All Together...
- Caching
 - Module Caching Caveats
- Core Modules
- Cycles
- File Modules
- Folders as Modules
- Loading from node_modules Folders
- Loading from the global folders
- The module wrapper
- The module Object
 - module.children
 - module.exports
 - exports alias
 - module.filename
 - module.id
 - module.loaded
 - module.parent
 - module.require(id)
- net
 - Class: net.Server
 - Event: 'close'
 - Event: 'connection'
 - Event: 'error'
 - Event: 'listening'
 - server.address()
 - server.close([callback])
 - server.connections
 - server.getConnections(callback)
 - server.listen(handle[, backlog][, callback])
 - server.listen(options[, callback])
 - server.listen(path[, backlog][, callback])
 - server.listen(port[, hostname][, backlog][, callback])

- `server.listening`
- `server.maxConnections`
- `server.ref()`
- `server.unref()`
- Class: `net.Socket`
 - `new net.Socket([options])`
 - `Event: 'close'`
 - `Event: 'connect'`
 - `Event: 'data'`
 - `Event: 'drain'`
 - `Event: 'end'`
 - `Event: 'error'`
 - `Event: 'lookup'`
 - `Event: 'timeout'`
 - `socket.address()`
 - `socket.bufferSize`
 - `socket.bytesRead`
 - `socket.bytesWritten`
 - `socket.connect(options[, connectListener])`
 - `socket.connect(path[, connectListener])`
 - `socket.connect(port[, host][, connectListener])`
 - `socket.connecting`
 - `socket.destroy([exception])`
 - `socket.destroyed`
 - `socket.end([data][, encoding])`
 - `socket.localAddress`
 - `socket.localPort`
 - `socket.pause()`
 - `socket.ref()`
 - `socket.remoteAddress`
 - `socket.remoteFamily`
 - `socket.remotePort`
 - `socket.resume()`

- `socket.setEncoding([encoding])`
- `socket.setKeepAlive([enable][, initialDelay])`
- `socket.setNoDelay([noDelay])`
- `socket.setTimeout(timeout[, callback])`
- `socket.unref()`
- `socket.write(data[, encoding][, callback])`
- `net.connect(options[, connectListener])`
- `net.connect(path[, connectListener])`
- `net.connect(port[, host][, connectListener])`
- `net.createConnection(options[, connectListener])`
- `net.createConnection(path[, connectListener])`
- `net.createConnection(port[, host][, connectListener])`
- `net.createServer([options][, connectionListener])`
- `net.isIP(input)`
- `net.isIPv4(input)`
- `net.isIPv6(input)`
- OS
 - `os.EOL`
 - `os.arch()`
 - `os.constants`
 - `os.cpus()`
 - `os.endianness()`
 - `os.freemem()`
 - `os.homedir()`
 - `os.hostname()`
 - `os.loadavg()`
 - `os.networkInterfaces()`
 - `os.platform()`
 - `os.release()`
 - `os.tmpdir()`
 - `os.totalmem()`
 - `os.type()`
 - `os.uptime()`

- `os.userInfo([options])`
- OS Constants
 - Signal Constants
 - Error Constants
 - POSIX Error Constants
 - Windows Specific Error Constants
 - libuv Constants
- Path
 - Windows vs. POSIX
 - `path.basename(path[, ext])`
 - `path.delimiter`
 - `path.dirname(path)`
 - `path.extname(path)`
 - `path.format(pathObject)`
 - `path.isAbsolute(path)`
 - `path.join([path[, ...]])`
 - `path.normalize(path)`
 - `path.parse(path)`
 - `path.posix`
 - `path.relative(from, to)`
 - `path.resolve([path[, ...]])`
 - `path.sep`
 - `path.win32`
- process
 - Process Events
 - Event: 'beforeExit'
 - Event: 'disconnect'
 - Event: 'exit'
 - Event: 'message'
 - Event: 'rejectionHandled'
 - Event: 'uncaughtException'
 - Warning: Using 'uncaughtException' correctly
 - Event: 'unhandledRejection'

- Event: 'warning'
 - Emitting custom warnings
 - Emitting custom deprecation warnings
- Signal Events
- `process.abort()`
- `process.arch`
- `process.argv`
- `process.chdir(directory)`
- `process.config`
- `process.connected`
- `process.cpuUsage([previousValue])`
- `process.cwd()`
- `process.disconnect()`
- `process.env`
- `process.emitWarning(warning[, name][, ctor])`
 - Avoiding duplicate warnings
- `process.execArgv`
- `process.execPath`
- `process.exit([code])`
- `process.exitCode`
- `process.getegid()`
- `process.geteuid()`
- `process.getgid()`
- `process.getgroups()`
- `process.getuid()`
- `process.hrtime([time])`
- `process.initgroups(user, extra_group)`
- `process.kill(pid[, signal])`
- `process.mainModule`
- `process.memoryUsage()`
- `process.nextTick(callback[, arg][, ...])`
- `process.pid`
- `process.platform`

- `process.release`
- `process.send(message[, sendHandle[, options]][], callback)`
- `process.setegid(id)`
- `process.seteuid(id)`
- `process.setgid(id)`
- `process.setgroups(groups)`
- `process.setuid(id)`
- `process.stderr`
- `process.stdin`
- `process.stdout`
 - TTY Terminals and `process.stdout`
- `process.title`
- `process.umask([mask])`
- `process.uptime()`
- `process.version`
- `process.versions`
- Exit Codes
- `punycode`
 - `punycode.decode(string)`
 - `punycode.encode(string)`
 - `punycode.toASCII(domain)`
 - `punycode.toUnicode(domain)`
 - `punycode.ucs2`
 - `punycode.ucs2.decode(string)`
 - `punycode.ucs2.encode(codePoints)`
 - `punycode.version`
- `Query String`
 - `querystring.escape(str)`
 - `querystring.parse(str[, sep[, eq[, options]]])`
 - `querystring.stringify(obj[, sep[, eq[, options]]])`
 - `querystring.unescape(str)`
- `Readline`
 - Class: Interface

- Event: 'close'
 - Event: 'line'
 - Event: 'pause'
 - Event: 'resume'
 - Event: 'SIGCONT'
 - Event: 'SIGINT'
 - Event: 'SIGTSTP'
 - rl.close()
 - rl.pause()
 - rl.prompt([preserveCursor])
 - rl.question(query, callback)
 - rl.resume()
 - rl.setPrompt(prompt)
 - rl.write(data[, key])
 - readline.clearLine(stream, dir)
 - readline.clearScreenDown(stream)
 - readline.createInterface(options)
 - Use of the completer Function
 - readline.cursorTo(stream, x, y)
 - readline.emitKeypressEvents(stream[, interface])
 - readline.moveCursor(stream, dx, dy)
 - Example: Tiny CLI
 - Example: Read File Stream Line-by-Line
- REPL
 - Design and Features
 - Commands and Special Keys
 - Default Evaluation
 - JavaScript Expressions
 - Global and Local Scope
 - Accessing Core Node.js Modules
 - Assignment of the _ (underscore) variable
 - Custom Evaluation Functions
 - Recoverable Errors

- Customizing REPL Output
- Class: REPLServer
 - Event: 'exit'
 - Event: 'reset'
 - `replServer.defineCommand(keyword, cmd)`
 - `replServer.displayPrompt([preserveCursor])`
- `repl.start([options])`
- The Node.js REPL
 - Environment Variable Options
 - Persistent History
 - `NODE_REPL_HISTORY_FILE`
 - Using the Node.js REPL with advanced line-editors
 - Starting multiple REPL instances against a single running instance
- Stream
 - Organization of this document
 - Types of Streams
 - Object Mode
 - Buffering
 - API for Stream Consumers
 - Writable Streams
 - Class: `stream.Writable`
 - Event: 'close'
 - Event: 'drain'
 - Event: 'error'
 - Event: 'finish'
 - Event: 'pipe'
 - Event: 'unpipe'
 - `writable.cork()`
 - `writable.end([chunk][, encoding][, callback])`
 - `writable.setDefaultEncoding(encoding)`
 - `writable.uncork()`
 - `writable.write(chunk[, encoding][, callback])`
 - Readable Streams

- Two Modes
- Three States
- Choose One
- Class: `stream.Readable`
 - Event: 'close'
 - Event: 'data'
 - Event: 'end'
 - Event: 'error'
 - Event: 'readable'
 - `readable.isPaused()`
 - `readable.pause()`
 - `readable.pipe(destination[, options])`
 - `readable.read([size])`
 - `readable.resume()`
 - `readable.setEncoding(encoding)`
 - `readable.unpipe([destination])`
 - `readable.unshift(chunk)`
 - `readable.wrap(stream)`
- Duplex and Transform Streams
 - Class: `stream.Duplex`
 - Class: `stream.Transform`
- API for Stream Implementers
 - Simplified Construction
 - Implementing a Writable Stream
 - Constructor: `new stream.Writable([options])`
 - `writable._write(chunk, encoding, callback)`
 - `writable._writev(chunks, callback)`
 - Errors While Writing
 - An Example Writable Stream
 - Implementing a Readable Stream
 - `new stream.Readable([options])`
 - `readable._read(size)`
 - `readable.push(chunk[, encoding])`

- Errors While Reading
- An Example Counting Stream
- Implementing a Duplex Stream
 - `new stream.Duplex(options)`
 - An Example Duplex Stream
 - Object Mode Duplex Streams
- Implementing a Transform Stream
 - `new stream.Transform([options])`
 - Events: 'finish' and 'end'
 - `transform._flush(callback)`
 - `transform._transform(chunk, encoding, callback)`
 - Class: `stream.PassThrough`
- Additional Notes
 - Compatibility with Older Node.js Versions
 - `readable.read(0)`
 - `readable.push('')`
- StringDecoder
 - Class: `new StringDecoder([encoding])`
 - `stringDecoder.end([buffer])`
 - `stringDecoder.write(buffer)`
- Timers
 - Class: Immediate
 - Class: Timeout
 - `timeout.ref()`
 - `timeout.unref()`
 - Scheduling Timers
 - `setImmediate(callback[, ...arg])`
 - `setInterval(callback, delay[, ...arg])`
 - `setTimeout(callback, delay[, ...arg])`
 - Cancelling Timers
 - `clearImmediate(immediate)`
 - `clearInterval(timeout)`
 - `clearTimeout(timeout)`

- TLS (SSL)

- TLS/SSL Concepts
 - Perfect Forward Secrecy
 - ALPN, NPN and SNI
 - Client-initiated renegotiation attack mitigation
- Modifying the Default TLS Cipher suite
- Class: `tls.Server`
 - Event: `'tlsClientError'`
 - Event: `'newSession'`
 - Event: `'OCSPRequest'`
 - Event: `'resumeSession'`
 - Event: `'secureConnection'`
 - `server.addContext(hostname, context)`
 - `server.address()`
 - `server.close([callback])`
 - `server.connections`
 - `server.getTicketKeys()`
 - `server.listen(port[, hostname][, callback])`
 - `server.setTicketKeys(keys)`
- Class: `tls.TLSSocket`
 - `new tls.TLSSocket(socket[, options])`
 - Event: `'OCSPResponse'`
 - Event: `'secureConnect'`
 - `tlsSocket.address()`
 - `tlsSocket.authorized`
 - `tlsSocket.authorizationError`
 - `tlsSocket.encrypted`
 - `tlsSocket.getCipher()`
 - `tlsSocket.getEphemeralKeyInfo()`
 - `tlsSocket.getPeerCertificate([detailed])`
 - `tlsSocket.getProtocol()`
 - `tlsSocket.getSession()`
 - `tlsSocket.getTLSTicket()`

- `tlsSocket.localAddress`
- `tlsSocket.localPort`
- `tlsSocket.remoteAddress`
- `tlsSocket.remoteFamily`
- `tlsSocket.remotePort`
- `tlsSocket.renegotiate(options, callback)`
- `tlsSocket.setMaxSendFragment(size)`
- `tls.connect(options[, callback])`
- `tls.connect(port[, host][, options][, callback])`
- `tls.createSecureContext(options)`
- `tls.createServer(options[, secureConnectionListener])`
- `tls.getCiphers()`
- **Deprecated APIs**
 - **Class: CryptoStream**
 - `cryptoStream.bytesWritten`
 - **Class: SecurePair**
 - **Event: 'secure'**
 - `tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])`
- **TTY**
 - **Class: tty.ReadStream**
 - `readStream.isRaw`
 - `readStream.setRawMode(mode)`
 - **Class: tty.WriteStream**
 - **Event: 'resize'**
 - `writeStream.columns`
 - `writeStream.rows`
 - `tty.isatty(fd)`
- **URL**
 - **URL Strings and URL Objects**
 - `urlObject.href`
 - `urlObject.protocol`
 - `urlObject.slashes`
 - `urlObject.host`

- `urlObject.auth`
- `urlObject.hostname`
- `urlObject.port`
- `urlObject.pathname`
- `urlObject.search`
- `urlObject.path`
- `urlObject.query`
- `urlObject.hash`
- `url.format(urlObject)`
- `url.parse(urlString[, parseQueryString[, slashesDenoteHost]])`
- `url.resolve(from, to)`
- Escaped Characters
- util
 - `util.debuglog(section)`
 - `util.deprecate(function, string)`
 - `util.format(format[, ...])`
 - `util.inherits(constructor, superConstructor)`
 - `util.inspect(object[, options])`
 - Customizing `util.inspect` colors
 - Custom `inspect()` function on Objects
 - Deprecated APIs
 - `util.debug(string)`
 - `util.error([...])`
 - `util.isArray(object)`
 - `util.isBoolean(object)`
 - `util.isBuffer(object)`
 - `util.isDate(object)`
 - `util.isError(object)`
 - `utilisFunction(object)`
 - `util.isNull(object)`
 - `util.isNullOrUndefined(object)`
 - `util.isNumber(object)`
 - `utilisObject(object)`

- `util.isPrimitive(object)`
- `util.isRegExp(object)`
- `util.isString(object)`
- `util.isSymbol(object)`
- `util.isUndefined(object)`
- `util.log(string)`
- `util.print([...])`
- `util.puts([...])`
- `util._extend(obj)`

- **V8**

- `v8.getHeapStatistics()`
- `v8.getHeapSpaceStatistics()`
- `v8.setFlagsFromString(string)`

- **Executing JavaScript**

- **Class: `vm.Script`**
 - `new vm.Script(code, options)`
 - `script.runInContext(contextifiedSandbox[, options])`
 - `script.runInNewContext([sandbox][, options])`
 - `script.runInThisContext([options])`
- `vm.createContext([sandbox])`
- `vm.isContext(sandbox)`
- `vm.runInContext(code, contextifiedSandbox[, options])`
- `vm.runInDebugContext(code)`
- `vm.runInNewContext(code[, sandbox][, options])`
- `vm.runInThisContext(code[, options])`
- **Example: Running an HTTP Server within a VM**
- **What does it mean to "contextify" an object?**

- **Zlib**

- **Compressing HTTP requests and responses**
- **Memory Usage Tuning**
- **Flushing**
- **Constants**
- **Class Options**

- Class: zlib.Deflate
- Class: zlib.DeflateRaw
- Class: zlib.Gunzip
- Class: zlib.Gzip
- Class: zlib.Inflate
- Class: zlib.InflateRaw
- Class: zlib.Unzip
- Class: zlib.Zlib
 - `zlib.flush([kind], callback)`
 - `zlib.params(level, strategy, callback)`
 - `zlib.reset()`
- `zlib.createDeflate([options])`
- `zlib.createDeflateRaw([options])`
- `zlib.createGunzip([options])`
- `zlib.createGzip([options])`
- `zlib.createInflate([options])`
- `zlib.createInflateRaw([options])`
- `zlib.createUnzip([options])`
- Convenience Methods
 - `zlib.deflate(buf[, options], callback)`
 - `zlib.deflateSync(buf[, options])`
 - `zlib.deflateRaw(buf[, options], callback)`
 - `zlib.deflateRawSync(buf[, options])`
 - `zlib.gunzip(buf[, options], callback)`
 - `zlib.gunzipSync(buf[, options])`
 - `zlib.gzip(buf[, options], callback)`
 - `zlib.gzipSync(buf[, options])`
 - `zlib.inflate(buf[, options], callback)`
 - `zlib.inflateSync(buf[, options])`
 - `zlib.inflateRaw(buf[, options], callback)`
 - `zlib.inflateRawSync(buf[, options])`
 - `zlib.unzip(buf[, options], callback)`
 - `zlib.unzipSync(buf[, options])`

About this Documentation

#

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

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

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

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

If you find an error in this documentation, please [submit an issue](#) or see the [contributing guide](#) for directions on how to submit a patch.

Stability Index

#

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

The stability indices are as follows:

Stability: 0 - Deprecated

This feature is known to be problematic, and changes are planned. Do not rely on it. Use of the feature may cause warnings. Backwards compatibility should not be expected.

Stability: 1 - Experimental

This feature is subject to change, and is gated by a command line flag. It may change or be removed in future versions.

Stability: 2 - Stable

The API has proven satisfactory. Compatibility with the npm ecosystem is a high priority, and will not be broken unless absolutely necessary.

Stability: 3 - Locked

Only fixes related to security, performance, or bug fixes will be accepted. Please do not suggest API changes in this area; they will be refused.

JSON Output

Stability: 1 - Experimental

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

This feature was added in Node.js v0.6.12. It is experimental.

Syscalls and man pages

System calls like `open(2)` and `read(2)` define the interface between user programs and the underlying operating system. Node functions which simply wrap a syscall, like `fs.open()`, will document that. The docs link to the corresponding man pages (short for manual pages) which describe how the syscalls work.

Caveat: some syscalls, like [lchown\(2\)](#), are BSD-specific. That means, for example, that `fs.lchown()` only works on Mac OS X and other BSD-derived systems, and is not available on Linux.

Most Unix syscalls have Windows equivalents, but behavior may differ on Windows relative to Linux and OS X. For an example of the subtle ways in which it's sometimes impossible to replace Unix syscall semantics on Windows, see [Node issue 4760](#).

Usage

```
node [options] [v8 options] [script.js | -e "script"] [arguments]
```

Please see the [Command Line Options](#) document for information about different options and ways to run scripts with Node.js.

Example

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

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

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

```
$ node example.js
Server running at http://127.0.0.1:3000/
```

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

Addons

#

Node.js Addons are dynamically-linked shared objects, written in C or C++, that can be loaded into Node.js using the `require()` function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

At the moment, the method for implementing Addons is rather complicated, involving knowledge of several components and APIs :

- **V8:** the C++ library Node.js currently uses to provide the JavaScript implementation. V8 provides the mechanisms for creating objects, calling functions, etc. V8's API is documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node.js source tree), which is also available [online](#).
- **libuv:** The C library that implements the Node.js event loop, its worker threads and all of the asynchronous behaviors of the platform. It also serves as a cross-platform abstraction library, giving easy, POSIX-like access across all major operating systems to many common system tasks, such as interacting with the filesystem, sockets, timers and system events. libuv also provides a pthreads-like threading abstraction that may be used to power more sophisticated asynchronous Addons that need to move beyond the standard event loop. Addon authors are encouraged to think about how to avoid blocking the event loop with I/O or other time-intensive tasks by off-loading work via libuv to non-

blocking system operations, worker threads or a custom use of libuv's threads.

- Internal Node.js libraries. Node.js itself exports a number of C/C++ APIs that Addons can use – the most important of which is the `node::ObjectWrap` class.
- Node.js includes a number of other statically linked libraries including OpenSSL. These other libraries are located in the `deps/` directory in the Node.js source tree. Only the V8 and OpenSSL symbols are purposefully re-exported by Node.js and may be used to various extents by Addons. See [Linking to Node.js' own dependencies](#) for additional information.

All of the following examples are available for [download](#) and may be used as a starting-point for your own Addon.

Hello world

#

This "Hello world" example is a simple Addon, written in C++, that is the equivalent of the following JavaScript code:

```
module.exports.hello = () => 'world';
```

First, create the file `hello.cc`:

```
// hello.cc
#include <node.h>

namespace demo {

    using v8::FunctionCallbackInfo;
    using v8::Isolate;
    using v8::Local;
    using v8::Object;
    using v8::String;
```

```
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}

void init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(addon, init)

} // namespace demo
```

Note that all Node.js Addons must export an initialization function following the pattern:

```
void Initialize(Local<Object> exports);
NODE_MODULE(module_name, Initialize)
```

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

The `module_name` must match the filename of the final binary (excluding the .node suffix).

In the `hello.cc` example, then, the initialization function is `init` and the Addon module name is `addon`.

Building

#

Once the source code has been written, it must be compiled into the binary `addon.node` file. To do so, create a file called `binding.gyp` in the top-level of the project describing the build configuration of your module using a JSON-like format.

This file is used by `node-gyp` -- a tool written specifically to compile Node.js Addons.

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

Note: A version of the `node-gyp` utility is bundled and distributed with Node.js as part of `npm`. This version is not made directly available for developers to use and is intended only to support the ability to use the `npm install` command to compile and install Addons. Developers who wish to use `node-gyp` directly can install it using the command `npm install -g node-gyp`. See the `node-gyp` [installation instructions](#) for more information, including platform-specific requirements.

Once the `binding.gyp` file has been created, use `node-gyp configure` to generate the appropriate project build files for the current platform. This will generate either a `Makefile` (on Unix platforms) or a `vcxproj` file (on Windows) in the `build/` directory.

Next, invoke the `node-gyp build` command to generate the compiled `addon.node` file. This will be put into the `build/Release/` directory.

When using `npm install` to install a Node.js Addon, npm uses its own bundled version of `node-gyp` to perform this same set of actions, generating a compiled version of the Addon for the user's platform on demand.

Once built, the binary Addon can be used from within Node.js by pointing `require()` to the built `addon.node` module:

```
// hello.js
```

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

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

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

Because the exact path to the compiled Addon binary can vary depending on how it is compiled (i.e. sometimes it may be in `./build/Debug/`), Addons can use the `bindings` package to load the compiled module.

Note that while the `bindings` package implementation is more sophisticated in how it locates Addon modules, it is essentially using a try-catch pattern similar to:

```
try {
  return require('./build/Release/addon.node');
} catch (err) {
  return require('./build/Debug/addon.node');
}
```

Linking to Node.js' own dependencies

Node.js uses a number of statically linked libraries such as V8, libuv and OpenSSL. All Addons are required to link to V8 and may link to any of the other dependencies as well. Typically, this is as simple as including the appropriate `#include <...>` statements (e.g. `#include <v8.h>`) and `node-gyp` will locate the appropriate headers automatically. However, there are a few caveats to be aware of:

- When `node-gyp` runs, it will detect the specific release version of Node.js and download either the full source tarball or just the headers. If the full source is downloaded, Addons will have complete access to the full set of Node.js dependencies. However, if only the Node.js headers are downloaded, then only the symbols exported by Node.js will be available.

- `node-gyp` can be run using the `--nodedir` flag pointing at a local Node.js source image. Using this option, the Addon will have access to the full set of dependencies.

Loading Addons using require()

#

The filename extension of the compiled Addon binary is `.node` (as opposed to `.dll` or `.so`). The `require()` function is written to look for files with the `.node` file extension and initialize those as dynamically-linked libraries.

When calling `require()`, the `.node` extension can usually be omitted and Node.js will still find and initialize the Addon. One caveat, however, is that Node.js will first attempt to locate and load modules or JavaScript files that happen to share the same base name. For instance, if there is a file `addon.js` in the same directory as the binary `addon.node`, then `require('addon')` will give precedence to the `addon.js` file and load it instead.

Native Abstractions for Node.js

#

Each of the examples illustrated in this document make direct use of the Node.js and V8 APIs for implementing Addons. It is important to understand that the V8 API can, and has, changed dramatically from one V8 release to the next (and one major Node.js release to the next). With each change, Addons may need to be updated and recompiled in order to continue functioning. The Node.js release schedule is designed to minimize the frequency and impact of such changes but there is little that Node.js can do currently to ensure stability of the V8 APIs.

The `Native Abstractions for Node.js` (or `nan`) provide a set of tools that Addon developers are recommended to use to keep compatibility between past and future releases of V8 and Node.js. See the `nan` `examples` for an illustration of how it can be used.

Addon examples

#

Following are some example Addons intended to help developers get started. The examples make use of the V8 APIs. Refer to the online `V8 reference` for help with

the various V8 calls, and V8's [Embedder's Guide](#) for an explanation of several concepts used such as handles, scopes, function templates, etc.

Each of these examples using the following `binding.gyp` file:

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

In cases where there is more than one `.cc` file, simply add the additional filename to the `sources` array. For example:

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

Once the `binding.gyp` file is ready, the example Addons can be configured and built using `node-gyp`:

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

Function arguments

Addons will typically expose objects and functions that can be accessed from JavaScript running within Node.js. When functions are invoked from JavaScript, the input arguments and return value must be mapped to and from the C/C++ code.

The following example illustrates how to read function arguments passed from JavaScript and how to return a result:

```
// addon.cc

#include <node.h>

namespace demo {

using v8::Exception;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// This is the implementation of the "add" method
// Input arguments are passed using the
// const FunctionCallbackInfo<Value>& args struct
void Add(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    // Check the number of arguments passed.
    if (args.Length() < 2) {
        // Throw an Error that is passed back to JavaScript
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate, "Wrong number of arguments")));
        return;
    }

    // Check the argument types
    if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
        isolate->ThrowException(Exception::TypeError(
            String::NewFromUtf8(isolate, "Wrong arguments")));
        return;
    }

    // Add the numbers
    Local<Number> result = Number::New(isolate, args[0].Value() + args[1].Value());
    args.GetReturnValue().Set(result);
}
```

```

}

// Perform the operation

double value = args[0]->NumberValue() + args[1]->NumberValue();
Local<Number> num = Number::New(isolate, value);

// Set the return value (using the passed in
// FunctionCallbackInfo<Value>&)
args.GetReturnValue().Set(num);

}

void Init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, Init)

} // namespace demo

```

Once compiled, the example Addon can be required and used from within Node.js:

```

// test.js
const addon = require('./build/Release/addon');

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

```

Callbacks

#

It is common practice within Addons to pass JavaScript functions to a C++ function and execute them from there. The following example illustrates how to invoke such callbacks:

```

// addon.cc

#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Null;
using v8::Object;
using v8::String;
using v8::Value;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    Local<Function> cb = Local<Function>::Cast(args[0]);
    const unsigned argc = 1;
    Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world");
        cb->Call(Null(isolate), argc, argv);
    }
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", RunCallback);
}

NODE_MODULE(addon, Init)

} // namespace demo

```

Note that this example uses a two-argument form of `Init()` that receives the full `module` object as the second argument. This allows the Addon to completely

overwrite `exports` with a single function instead of adding the function as a property of `exports`.

To test it, run the following JavaScript:

```
// test.js
const addon = require('./build/Release/addon');

addon((msg) => {
  console.log(msg); // 'hello world'
});
```

Note that, in this example, the callback function is invoked synchronously.

Object factory

Addons can create and return new objects from within a C++ function as illustrated in the following example. An object is created and returned with a property `msg` that echoes the string passed to `CreateObject()`:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
```

```

Local<Object> obj = Object::New(isolate);
obj->Set(String::NewFromUtf8(isolate, "msg"), args[0]->ToString());

args.GetReturnValue().Set(obj);

}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(addon, Init)

} // namespace demo

```

To test it in JavaScript:

```

// test.js
const addon = require('./build/Release/addon');

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

```

Function factory

Another common scenario is creating JavaScript functions that wrap C++ functions and returning those back to JavaScript:

```

// addon.cc
#include <node.h>

```

```
namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();
    args.GetReturnValue().Set(String::NewFromUtf8(isolate, "hello world"))
}

void CreateFunction(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
    Local<Function> fn = tpl->GetFunction();

    // omit this to make it anonymous
    fn->SetName(String::NewFromUtf8(isolate, "theFunction"));

    args.GetReturnValue().Set(fn);
}

void Init(Local<Object> exports, Local<Object> module) {
    NODE_SET_METHOD(module, "exports", CreateFunction);
}

NODE_MODULE(addon, Init)
```

```
} // namespace demo
```

To test:

```
// test.js
const addon = require('./build/Release/addon');

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

Wrapping C++ objects

It is also possible to wrap C++ objects/classes in a way that allows new instances to be created using the JavaScript `new` operator:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Local;
using v8::Object;

void InitAll(Local<Object> exports) {
    MyObject::Init(exports);
}

NODE_MODULE(addon, InitAll)

} // namespace demo
```

Then, in `myobject.h`, the wrapper class inherits from `node::ObjectWrap`:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

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

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
    static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
    static v8::Persistent<v8::Function> constructor;
    double value_;

};

} // namespace demo

#endif
```

In `myobject.cc`, implement the various methods that are to be exposed. Below, the method `plusOne()` is exposed by adding it to the constructor's prototype:

```
// myobject.cc
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {}

MyObject::~MyObject() {}

void MyObject::Init(Local<Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);
}
```

```

// Prototype

NODE_SET_PROTOTYPE_METHOD(tp, "plusOne", PlusOne);

constructor.Reset(isolate, tpl->GetFunction());
exports->Set(String::NewFromUtf8(isolate, "MyObject"),
              tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Context> context = isolate->GetCurrentContext();
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> result =
      cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(result);
  }
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());

```

```
obj->value_ += 1;

args.GetReturnValue().Set(Number::New(isolate, obj->value_));

}

} // namespace demo
```

To build this example, the `myobject.cc` file must be added to the `binding.gyp`:

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

Test it with:

```
// test.js
const addon = require('./build/Release/addon');

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

Factory of wrapped objects

Alternatively, it is possible to use a factory pattern to avoid explicitly creating object instances using the JavaScript `new` operator:

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

First, the `createObject()` method is implemented in `addon.cc`:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
    MyObject::NewInstance(args);
}

void InitAll(Local<Object> exports, Local<Object> module) {
    MyObject::Init(exports->GetIsolate());

    NODE_SET_METHOD(module, "exports", CreateObject);
}
```

```
NODE_MODULE(addon, InitAll)
```

```
} // namespace demo
```

In `myobject.h`, the static method `NewInstance()` is added to handle instantiating the object. This method takes the place of using `new` in JavaScript:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
public:
    static void Init(v8::Isolate* isolate);
    static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
    static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
    static v8::Persistent<v8::Function> constructor;
    double value_;

};

} // namespace demo
```

```
#endif
```

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

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {}

MyObject::~MyObject() {}

void MyObject::Init(Isolate* isolate) {
    // Prepare constructor template
```

```
Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
tpl->InstanceTemplate()->SetInternalFieldCount(1);

// Prototype
NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
Isolate* isolate = args.GetIsolate();

if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
} else {
    // Invoked as plain function `MyObject(...)`, turn into construct call
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Context> context = isolate->GetCurrentContext();
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(instance);
}
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
Isolate* isolate = args.GetIsolate();
```

```

const unsigned argc = 1;
Local<Value> argv[argc] = { args[0] };
Local<Function> cons = Local<Function>::New(isolate, constructor);
Local<Context> context = isolate->GetCurrentContext();
Local<Object> instance =
    cons->NewInstance(context, argc, argv).ToLocalChecked();

args.GetReturnValue().Set(instance);
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

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

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

} // namespace demo

```

Once again, to build this example, the `myobject.cc` file must be added to the `binding.gyp`:

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

```
    }  
]  
}
```

Test it with:

```
// test.js  
const createObject = require('./build/Release/addon');  
  
var obj = createObject(10);  
console.log(obj.plusOne()); // 11  
console.log(obj.plusOne()); // 12  
console.log(obj.plusOne()); // 13  
  
var obj2 = createObject(20);  
console.log(obj2.plusOne()); // 21  
console.log(obj2.plusOne()); // 22  
console.log(obj2.plusOne()); // 23
```

Passing wrapped objects around

In addition to wrapping and returning C++ objects, it is possible to pass wrapped objects around by unwrapping them with the Node.js helper function `node::ObjectWrap::Unwrap`. The following examples shows a function `add()` that can take two `MyObject` objects as input arguments:

```
// addon.cc  
#include <node.h>  
#include <node_object_wrap.h>  
#include "myobject.h"  
  
namespace demo {
```

```
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
    MyObject::NewInstance(args);
}

void Add(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

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

    double sum = obj1->value() + obj2->value();
    args.GetReturnValue().Set(Number::New(isolate, sum));
}

void InitAll(Local<Object> exports) {
    MyObject::Init(exports->GetIsolate());

    NODE_SET_METHOD(exports, "createObject", CreateObject);
    NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, InitAll)
```

```
} // namespace demo
```

In `myobject.h`, a new public method is added to allow access to private values after unwrapping the object.

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
public:
    static void Init(v8::Isolate* isolate);
    static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);
    inline double value() const { return value_; }

private:
    explicit MyObject(double value = 0);
    ~MyObject();

    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
    static v8::Persistent<v8::Function> constructor;
    double value_;

};

} // namespace demo

#endif
```

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

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {}

MyObject::~MyObject() {}

void MyObject::Init(Isolate* isolate) {
    // Prepare constructor template
    Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
    tpl->InstanceTemplate()->SetInternalFieldCount(1);
```

```
constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    if (args.IsConstructCall()) {
        // Invoked as constructor: `new MyObject(...)`
        double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
        MyObject* obj = new MyObject(value);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
    } else {
        // Invoked as plain function `MyObject(...)` , turn into construct call
        const int argc = 1;
        Local<Value> argv[argc] = { args[0] };
        Local<Context> context = isolate->GetCurrentContext();
        Local<Function> cons = Local<Function>::New(isolate, constructor);
        Local<Object> instance =
            cons->NewInstance(context, argc, argv).ToLocalChecked();
        args.GetReturnValue().Set(instance);
    }
}
```

```
void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    const unsigned argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Context> context = isolate->GetCurrentContext();
    Local<Object> instance =
```

```
    cons->NewInstance(context, argc, argv).ToLocalChecked();

    args.GetReturnValue().Set(instance);
}

} // namespace demo
```

Test it with:

```
// test.js
const addon = require('./build/Release/addon');

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

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

AtExit hooks

An "AtExit" hook is a function that is invoked after the Node.js event loop has ended but before the JavaScript VM is terminated and Node.js shuts down. "AtExit" hooks are registered using the `node::AtExit` API.

void AtExit(callback, args)

- `callback : void (*)(void*)` - A pointer to the function to call at exit.
- `args : void*` - A pointer to pass to the callback at exit.

Registers exit hooks that run after the event loop has ended but before the VM is killed.

AtExit takes two parameters: a pointer to a callback function to run at exit, and a pointer to untyped context data to be passed to that callback.

Callbacks are run in last-in first-out order.

The following `addon.cc` implements AtExit:

```
// addon.cc
#define NDEBUG

#include <assert.h>
#include <stdlib.h>
#include <node.h>

namespace demo {

using node::AtExit;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;

static char cookie[] = "yum yum";
static int at_exit_cb1_called = 0;
static int at_exit_cb2_called = 0;

static void at_exit_cb1(void* arg) {
    Isolate* isolate = static_cast<Isolate*>(arg);
    HandleScope scope(isolate);
    Local<Object> obj = Object::New(isolate);
    assert(!obj.IsEmpty()); // assert VM is still alive
    assert(obj->IsObject());
    at_exit_cb1_called++;
}

static void at_exit_cb2(void* arg) {
    assert(arg == static_cast<void*>(cookie));
```

```
at_exit_cb2_called++;

}

static void sanity_check(void*) {
    assert(at_exit_cb1_called == 1);
    assert(at_exit_cb2_called == 2);
}

void init(Local<Object> exports) {
    AtExit(sanity_check);
    AtExit(at_exit_cb2, cookie);
    AtExit(at_exit_cb2, cookie);
    AtExit(at_exit_cb1, exports->GetIsolate());
}

NODE_MODULE(addon, init);

} // namespace demo
```

Test in JavaScript by running:

```
// test.js
const addon = require('./build/Release/addon');
```

Assert

Stability: 3 - Locked

The `assert` module provides a simple set of assertion tests that can be used to test invariants. The module is intended for internal use by Node.js, but can be used in application code via `require('assert')`. However, `assert` is not a testing

framework, and is not intended to be used as a general purpose assertion library.

The API for the `assert` module is **Locked**. This means that there will be no additions or changes to any of the methods implemented and exposed by the module.

assert(value[, message])

Added in: v0.5.9

An alias of `assert.ok()`.

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

assert(true); // OK
assert(1);    // OK
assert(false);
              // throws "AssertionError: false == true"
assert(0);
              // throws "AssertionError: 0 == true"
assert(false, 'it\'s false');
              // throws "AssertionError: it's false"
```

assert.deepEqual(actual, expected[, message])

Added in: v0.1.21

Tests for deep equality between the `actual` and `expected` parameters. Primitive values are compared with the equal comparison operator (`==`).

Only enumerable "own" properties are considered. The `deepEqual()` implementation does not test object prototypes, attached symbols, or non-enumerable properties. This can lead to some potentially surprising results. For example, the following example does not throw an `AssertionError` because the properties on the `Error` object are non-enumerable:

```
// WARNING: This does not throw an AssertionError!
assert.deepEqual(Error('a'), Error('b'));
```

"Deep" equality means that the enumerable "own" properties of child objects are evaluated also:

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

const obj1 = {
  a : {
    b : 1
  }
};

const obj2 = {
  a : {
    b : 2
  }
};

const obj3 = {
  a : {
    b : 1
  }
};

const obj4 = Object.create(obj1);

assert.deepEqual(obj1, obj1);
// OK, object is equal to itself

assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }
// values of b are different
```

```
assert.deepEqual(obj1, obj3);
  // OK, objects are equal

assert.deepEqual(obj1, obj4);
  // AssertionError: { a: { b: 1 } } deepEqual {}
  // Prototypes are ignored
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.deepEqual(actual, expected[, message])

Added in: v1.2.0

Generally identical to `assert.deepEqual()` with two exceptions. First, primitive values are compared using the strict equality operator (`==`). Second, object comparisons include a strict equality check of their prototypes.

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

assert.deepEqual({a:1}, {a:'1'});
  // OK, because 1 == '1'

assert.deepStrictEqual({a:1}, {a:'1'});
  // AssertionError: { a: 1 } deepStrictEqual { a: '1' }
  // because 1 !== '1' using strict equality
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.doesNotThrow(block[, error][, message])

Asserts that the function `block` does not throw an error. See [assert.throws\(\)](#) for more details.

When `assert.doesNotThrow()` is called, it will immediately call the `block` function.

If an error is thrown and it is the same type as that specified by the `error` parameter, then an `AssertionError` is thrown. If the error is of a different type, or if the `error` parameter is undefined, the error is propagated back to the caller.

The following, for instance, will throw the `TypeError` because there is no matching error type in the assertion:

```
assert.doesNotThrow(  
  () => {  
    throw new TypeError('Wrong value');  
  },  
  SyntaxError  
);
```

However, the following will result in an `AssertionError` with the message 'Got unwanted exception (TypeError).':

```
assert.doesNotThrow(  
  () => {  
    throw new TypeError('Wrong value');  
  },  
  TypeError  
);
```

If an `AssertionError` is thrown and a value is provided for the `message` parameter, the value of `message` will be appended to the `AssertionError`

message:

```
assert.doesNotThrow(  
  () => {  
    throw new TypeError('Wrong value');  
  },  
  TypeError,  
  'Whoops'  
);  
// Throws: AssertionError: Got unwanted exception (TypeError). Whoops
```

assert.equal(actual, expected[, message])

Added in: v0.1.21

Tests shallow, coercive equality between the `actual` and `expected` parameters using the equal comparison operator (`==`).

```
const assert = require('assert');  
  
assert.equal(1, 1);  
  // OK, 1 == 1  
assert.equal(1, '1');  
  // OK, 1 == '1'  
  
assert.equal(1, 2);  
  // AssertionError: 1 == 2  
assert.equal({a: {b: 1}}, {a: {b: 1}});  
  //AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

If the values are not equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

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

Added in: v0.1.21

Throws an `AssertionError`. If `message` is falsy, the error message is set as the values of `actual` and `expected` separated by the provided `operator`. Otherwise, the error message is the value of `message`.

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

assert.fail(1, 2, undefined, '>');
// AssertionError: 1 > 2

assert.fail(1, 2, 'whoops', '>');
// AssertionError: whoops
```

assert.ifError(value)

Added in: v0.1.97

Throws `value` if `value` is truthy. This is useful when testing the `error` argument in callbacks.

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

assert.ifError(0); // OK
assert.ifError(1); // Throws 1
assert.ifError('error') // Throws 'error'
assert.ifError(new Error()); // Throws Error
```

assert.notDeepEqual(actual, expected[, message])

Added in: v0.1.21

Tests for any deep inequality. Opposite of `assert.deepEqual()`.

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

const obj1 = {
  a : {
    b : 1
  }
};

const obj2 = {
  a : {
    b : 2
  }
};

const obj3 = {
  a : {
    b : 1
  }
}

const obj4 = Object.create(obj1);

assert.notDeepEqual(obj1, obj1);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj2);
// OK, obj1 and obj2 are not deeply equal

assert.notDeepEqual(obj1, obj3);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj4);
// OK, obj1 and obj2 are not deeply equal
```

If the values are deeply equal, an `AssertionError` is thrown with a message

property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.notDeepStrictEqual(actual, expected[, message])

Added in: v1.2.0

Tests for deep strict inequality. Opposite of `assert.deepStrictEqual()`.

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

assert.notDeepEqual({a:1}, {a:'1'});
// AssertionError: { a: 1 } notDeepEqual { a: '1' }

assert.notDeepStrictEqual({a:1}, {a:'1'});
// OK
```

If the values are deeply and strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.notEqual(actual, expected[, message])

Added in: v0.1.21

Tests shallow, coercive inequality with the not equal comparison operator (`!=`).

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

assert.notEqual(1, 2);
// OK

assert.notEqual(1, 1);
// AssertionError: 1 != 1
```

```
assert.notEqual(1, '1');
// AssertionError: 1 != '1'
```

If the values are equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.notStrictEqual(actual, expected[, message])

Added in: v0.1.21

Tests strict inequality as determined by the strict not equal operator (`!==`).

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

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
// AssertionError: 1 != 1

assert.notStrictEqual(1, '1');
// OK
```

If the values are strictly equal, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.ok(value[, message])

Added in: v0.1.21

Tests if `value` is truthy. It is equivalent to `assert.equal(!value, true, message)`.

If `value` is not truthy, an `AssertionError` is thrown with a `message` property set equal to the value of the `message` parameter. If the `message` parameter is `undefined`, a default error message is assigned.

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

assert.ok(true); // OK
assert.ok(1); // OK
assert.ok(false);
  // throws "AssertionError: false == true"
assert.ok(0);
  // throws "AssertionError: 0 == true"
assert.ok(false, 'it\'s false');
  // throws "AssertionError: it's false"
```

assert.strictEqual(actual, expected[, message])

Added in: v0.1.21

Tests strict equality as determined by the strict equality operator (`====`).

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

assert.strictEqual(1, 2);
  // AssertionError: 1 === 2

assert.strictEqual(1, 1);
  // OK

assert.strictEqual(1, '1');
  // AssertionError: 1 === '1'
```

If the values are not strictly equal, an `AssertionError` is thrown with a `message`

property set equal to the value of the `message` parameter. If the `message` parameter is undefined, a default error message is assigned.

assert.throws(block[, error][, message])

Added in: v0.1.21

Expects the function `block` to throw an error.

If specified, `error` can be a constructor, [RegExp](#), or validation function.

If specified, `message` will be the message provided by the [AssertionError](#) if the block fails to throw.

Validate instanceof using constructor:

```
assert.throws(  
  () => {  
    throw new Error('Wrong value');  
  },  
  Error  
);
```

Validate error message using [RegExp](#):

```
assert.throws(  
  () => {  
    throw new Error('Wrong value');  
  },  
  /value/  
);
```

Custom error validation:

```
assert.throws(  
  () => {  
    throw new Error('Wrong value');  
  },  
  function(err) {  
    if (err instanceof Error) && /value/.test(err) {  
      return true;  
    }  
  },  
  'unexpected error'  
);
```

Note that `error` can not be a string. If a string is provided as the second argument, then `error` is assumed to be omitted and the string will be used for `message` instead. This can lead to easy-to-miss mistakes:

```
// THIS IS A MISTAKE! DO NOT DO THIS!  
assert.throws(myFunction, 'missing foo', 'did not throw with expected me  
  
// Do this instead.  
assert.throws(myFunction, /missing foo/, 'did not throw with expected me
```

Buffer

Stability: 2 - Stable

Prior to the introduction of `TypedArray` in ECMAScript 2015 (ES6), the JavaScript language had no mechanism for reading or manipulating streams of binary data. The `Buffer` class was introduced as part of the Node.js API to make it possible to interact with octet streams in the context of things like TCP streams and file system

operations.

Now that `TypedArray` has been added in ES6, the `Buffer` class implements the `Uint8Array` API in a manner that is more optimized and suitable for Node.js' use cases.

Instances of the `Buffer` class are similar to arrays of integers but correspond to fixed-sized, raw memory allocations outside the V8 heap. The size of the `Buffer` is established when it is created and cannot be resized.

The `Buffer` class is a global within Node.js, making it unlikely that one would need to ever use `require('buffer').Buffer`.

```
const buf1 = Buffer.alloc(10);
// Creates a zero-filled Buffer of length 10.

const buf2 = Buffer.alloc(10, 1);
// Creates a Buffer of length 10, filled with 0x01.

const buf3 = Buffer.allocUnsafe(10);
// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using either fill() or write().

const buf4 = Buffer.from([1,2,3]);
// Creates a Buffer containing [01, 02, 03].

const buf5 = Buffer.from('test');
// Creates a Buffer containing ASCII bytes [74, 65, 73, 74].
```



```
const buf6 = Buffer.from('tést', 'utf8');
// Creates a Buffer containing UTF8 bytes [74, c3, a9, 73, 74].
```

Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe()

#

In versions of Node.js prior to v6, `Buffer` instances were created using the `Buffer` constructor function, which allocates the returned `Buffer` differently based on what arguments are provided:

- Passing a number as the first argument to `Buffer()` (e.g. `new Buffer(10)`), allocates a new `Buffer` object of the specified size. The memory allocated for such `Buffer` instances is *not initialized* and *can contain sensitive data*. Such `Buffer` objects *must* be initialized *manually* by using either `buf.fill(0)` or by writing to the `Buffer` completely. While this behavior is *intentional* to improve performance, development experience has demonstrated that a more explicit distinction is required between creating a fast-but-uninitialized `Buffer` versus creating a slower-but-safer `Buffer`.
- Passing a string, array, or `Buffer` as the first argument copies the passed object's data into the `Buffer`.
- Passing an `ArrayBuffer` returns a `Buffer` that shares allocated memory with the given `ArrayBuffer`.

Because the behavior of `new Buffer()` changes significantly based on the type of value passed as the first argument, applications that do not properly validate the input arguments passed to `new Buffer()`, or that fail to appropriately initialize newly allocated `Buffer` content, can inadvertently introduce security and reliability issues into their code.

To make the creation of `Buffer` objects more reliable and less error prone, the various forms of the `new Buffer()` constructor have been **deprecated** and replaced by separate `Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()` methods.

Developers should migrate all existing uses of the `new Buffer()` constructors to one of these new APIs.

- `Buffer.from(array)` returns a new `Buffer` containing a *copy* of the provided octets.

- `Buffer.from(arrayBuffer[, byteOffset [, length]])` returns a new `Buffer` that *shares* the same allocated memory as the given `ArrayBuffer`.
- `Buffer.from(buffer)` returns a new `Buffer` containing a *copy* of the contents of the given `Buffer`.
- `Buffer.from(str[, encoding])` returns a new `Buffer` containing a *copy* of the provided string.
- `Buffer.alloc(size[, fill[, encoding]])` returns a "filled" `Buffer` instance of the specified size. This method can be significantly slower than `Buffer.allocUnsafe(size)` but ensures that newly created `Buffer` instances never contain old and potentially sensitive data.
- `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` each return a new `Buffer` of the specified `size` whose content *must* be initialized using either `buf.fill(0)` or written to completely.

`Buffer` instances returned by `Buffer.allocUnsafe(size)` *may* be allocated off a shared internal memory pool if `size` is less than or equal to half `Buffer.poolSize`. Instances returned by `Buffer.allocUnsafeSlow(size)` *never* use the shared internal memory pool.

The `--zero-fill-buffers` command line option

Added in: v5.10.0

Node.js can be started using the `--zero-fill-buffers` command line option to force all newly allocated `Buffer` instances created using either `new Buffer(size)`, `Buffer.allocUnsafe(size)`, `Buffer.allocUnsafeSlow(size)` or `new SlowBuffer(size)` to be *automatically zero-filled* upon creation. Use of this flag *changes the default behavior* of these methods and *can have a significant impact* on performance. Use of the `--zero-fill-buffers` option is recommended only when absolutely necessary to enforce that newly allocated `Buffer` instances cannot contain potentially sensitive data.

```
$ node --zero-fill-buffers
> Buffer.allocUnsafe(5);
<Buffer 00 00 00 00 00>
```

What makes `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` "unsafe"?

When calling `Buffer.allocUnsafe()` (and `Buffer.allocUnsafeSlow()`), the segment of allocated memory is *uninitialized* (it is not zeroed-out). While this design makes the allocation of memory quite fast, the allocated segment of memory might contain old data that is potentially sensitive. Using a `Buffer` created by `Buffer.allocUnsafe()` without completely overwriting the memory can allow this old data to be leaked when the `Buffer` memory is read.

While there are clear performance advantages to using `Buffer.allocUnsafe()`, extra care *must* be taken in order to avoid introducing security vulnerabilities into an application.

Buffers and Character Encodings

Buffers are commonly used to represent sequences of encoded characters such as UTF8, UCS2, Base64 or even Hex-encoded data. It is possible to convert back and forth between Buffers and ordinary JavaScript string objects by using an explicit encoding method.

```
const buf = Buffer.from('hello world', 'ascii');
console.log(buf.toString('hex'));
// prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// prints: aGVsbG8gd29ybGQ=
```

The character encodings currently supported by Node.js include:

- `'ascii'` - for 7-bit ASCII data only. This encoding method is very fast and will strip the high bit if set.
- `'utf8'` - Multibyte encoded Unicode characters. Many web pages and other document formats use UTF-8.

- `'utf16le'` - 2 or 4 bytes, little-endian encoded Unicode characters. Surrogate pairs (U+10000 to U+10FFFF) are supported.
- `'ucs2'` - Alias of `'utf16le'`.
- `'base64'` - Base64 string encoding. When creating a buffer from a string, this encoding will also correctly accept "URL and Filename Safe Alphabet" as specified in [RFC 4648, Section 5](#).
- `'binary'` - A way of encoding the buffer into a one-byte (`latin-1`) encoded string. The string `'latin-1'` is not supported. Instead, pass `'binary'` to use `'latin-1'` encoding.
- `'hex'` - Encode each byte as two hexadecimal characters.

Buffers and TypedArray

#

Buffers are also `Uint8Array` TypedArray instances. However, there are subtle incompatibilities with the TypedArray specification in ECMAScript 2015. For instance, while `ArrayBuffer#slice()` creates a copy of the slice, the implementation of `Buffer#slice()` creates a view over the existing Buffer without copying, making `Buffer#slice()` far more efficient.

It is also possible to create new TypedArray instances from a `Buffer` with the following caveats:

1. The `Buffer` object's memory is copied to the TypedArray, not shared.
2. The `Buffer` object's memory is interpreted as an array of distinct elements, and not as a byte array of the target type. That is, `new Uint32Array(Buffer.from([1,2,3,4]))` creates a 4-element `Uint32Array` with elements `[1,2,3,4]`, not a `Uint32Array` with a single element `[0x1020304]` or `[0x4030201]`.

It is possible to create a new `Buffer` that shares the same allocated memory as a TypedArray instance by using the TypeArray object's `.buffer` property:

```
const arr = new Uint16Array(2);
arr[0] = 5000;
arr[1] = 4000;

const buf1 = Buffer.from(arr); // copies the buffer
const buf2 = Buffer.from(arr.buffer); // shares the memory with arr;

console.log(buf1);
// Prints: <Buffer 88 a0>, copied buffer has only two elements
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;
console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>
```

Note that when creating a `Buffer` using the `TypedArray`'s `.buffer`, it is possible to use only a portion of the underlying `ArrayBuffer` by passing in `byteOffset` and `length` parameters:

```
const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);
console.log(buf.length);
// Prints: 16
```

The `Buffer.from()` and `TypedArray.from()` (e.g. `Uint8Array.from()`) have different signatures and implementations. Specifically, the `TypedArray` variants accept a second argument that is a mapping function that is invoked on every element of the typed array:

- `TypedArray.from(source[, mapFn[, thisArg]])`

The `Buffer.from()` method, however, does not support the use of a mapping function:

- `Buffer.from(array)`
- `Buffer.from(buffer)`
- `Buffer.from(arrayBuffer[, byteOffset [, length]])`
- `Buffer.from(str[, encoding])`

Buffers and ES6 iteration

Buffers can be iterated over using the ECMAScript 2015 (ES6) `for..of` syntax:

```
const buf = Buffer.from([1, 2, 3]);

for (var b of buf)
    console.log(b)

// Prints:
// 1
// 2
// 3
```

Additionally, the `buf.values()`, `buf.keys()`, and `buf.entries()` methods can be used to create iterators.

Class: Buffer

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

`new Buffer(array)`

Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use [Buffer.from\(array\)](#) instead.

- `array <Array>`

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

```
const buf = new Buffer([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);  
// creates a new Buffer containing ASCII bytes  
// ['b', 'u', 'f', 'f', 'e', 'r']
```

`new Buffer(buffer)`

Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use [Buffer.from\(buffer\)](#) instead.

- `buffer <Buffer>`

Copies the passed `buffer` data onto a new `Buffer` instance.

```
const buf1 = new Buffer('buffer');  
const buf2 = new Buffer(buf1);  
  
buf1[0] = 0x61;  
console.log(buf1.toString());  
// 'auffer'  
console.log(buf2.toString());  
// 'buffer' (copy is not changed)
```

`new Buffer(arrayBuffer[, byteOffset [, length]])`

Stability: 0 - Deprecated: Use `Buffer.from(arrayBuffer[, byteOffset [, length]])` instead.

- `arrayBuffer <ArrayBuffer>` The `.buffer` property of a `TypedArray` or a `new ArrayBuffer()`
- `byteOffset <Number>` Default: `0`
- `length <Number>` Default: `arrayBuffer.length - byteOffset`

When passed a reference to the `.buffer` property of a `TypedArray` instance, the newly created Buffer will share the same allocated memory as the `TypedArray`.

The optional `byteOffset` and `length` arguments specify a memory range within the `arrayBuffer` that will be shared by the `Buffer`.

```
const arr = new Uint16Array(2);
arr[0] = 5000;
arr[1] = 4000;

const buf = new Buffer(arr.buffer); // shares the memory with arr;

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// changing the TypdArray changes the Buffer also
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

Stability: 0 - Deprecated: Use [`Buffer.alloc\(size\[, fill\[, encoding\]\]\)`](#) instead (also see [`Buffer.allocUnsafe\(size\)`](#)).

- **size** <Number>

Allocates a new `Buffer` of `size` bytes. The `size` must be less than or equal to the value of `require('buffer').kMaxLength` (on 64-bit architectures, `kMaxLength` is `(231)-1`). Otherwise, a `RangeError` is thrown. A zero-length Buffer will be created if a `size` less than or equal to 0 is specified.

Unlike `ArrayBuffers`, the underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of a newly created `Buffer` are unknown and could contain sensitive data. Use `buf.fill(0)` to initialize a `Buffer` to zeroes.

```
const buf = new Buffer(5);
console.log(buf);
// <Buffer 78 e0 82 02 01>
// (octets will be different, every time)
buf.fill(0);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

`new Buffer(str[, encoding])`

#

Deprecated since: v6.0.0

Stability: 0 - Deprecated:
Use [`Buffer.from\(str\[, encoding\]\)`](#) instead.

- **str** <String> string to encode.
- **encoding** <String> Default: `'utf8'`

Creates a new Buffer containing the given JavaScript string `str`. If provided, the `encoding` parameter identifies the strings character encoding.

```
const buf1 = new Buffer('this is a tést');
console.log(buf1.toString());
  // prints: this is a tést
console.log(buf1.toString('ascii'));
  // prints: this is a tC)st

const buf2 = new Buffer('7468697320697320612074c3a97374', 'hex');
console.log(buf2.toString());
  // prints: this is a tést
```

Class Method: Buffer.alloc(size[, fill[, encoding]])

#

Added in: v5.10.0

- `size` <Number>
- `fill` <Value> Default: `undefined`
- `encoding` <String> Default: `utf8`

Allocates a new `Buffer` of `size` bytes. If `fill` is `undefined`, the `Buffer` will be *zero-filled*.

```
const buf = Buffer.alloc(5);
console.log(buf);
  // <Buffer 00 00 00 00 00>
```

The `size` must be less than or equal to the value of `require('buffer').kMaxLength` (on 64-bit architectures, `kMaxLength` is $(2^{31})-1$). Otherwise, a `RangeError` is thrown. A zero-length Buffer will be created if a `size` less than or equal to 0 is specified.

If `fill` is specified, the allocated `Buffer` will be initialized by calling

`buf.fill(fill)`. See `buf.fill()` for more information.

```
const buf = Buffer.alloc(5, 'a');
console.log(buf);
// <Buffer 61 61 61 61 61>
```

If both `fill` and `encoding` are specified, the allocated `Buffer` will be initialized by calling `buf.fill(fill, encoding)`. For example:

```
const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');
console.log(buf);
// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

Calling `Buffer.alloc(size)` can be significantly slower than the alternative `Buffer.allocUnsafe(size)` but ensures that the newly created `Buffer` instance contents will *never contain sensitive data*.

A `TypeError` will be thrown if `size` is not a number.

Class Method: `Buffer.allocUnsafe(size)`

#

Added in: v5.10.0

- `size` `<Number>`

Allocates a new *non-zero-filled* `Buffer` of `size` bytes. The `size` must be less than or equal to the value of `require('buffer').kMaxLength` (on 64-bit architectures, `kMaxLength` is `(231) - 1`). Otherwise, a `RangeError` is thrown. A zero-length `Buffer` will be created if a `size` less than or equal to 0 is specified.

The underlying memory for `Buffer` instances created in this way is *not initialized*. The contents of the newly created `Buffer` are unknown and *may contain sensitive data*. Use `buf.fill(0)` to initialize such `Buffer` instances to zeroes.

```
const buf = Buffer.allocUnsafe(5);
```

```
console.log(buf);
  // <Buffer 78 e0 82 02 01>
  // (octets will be different, every time)
buf.fill(0);
console.log(buf);
  // <Buffer 00 00 00 00 00>
```

A `TypeError` will be thrown if `size` is not a number.

Note that the `Buffer` module pre-allocates an internal `Buffer` instance of size `Buffer.poolSize` that is used as a pool for the fast allocation of new `Buffer` instances created using `Buffer.allocUnsafe(size)` (and the deprecated `new Buffer(size)` constructor) only when `size` is less than or equal to `Buffer.poolSize >> 1` (floor of `Buffer.poolSize` divided by two). The default value of `Buffer.poolSize` is `8192` but can be modified.

Use of this pre-allocated internal memory pool is a key difference between calling `Buffer.alloc(size, fill)` vs. `Buffer.allocUnsafe(size).fill(fill)`. Specifically, `Buffer.alloc(size, fill)` will never use the internal Buffer pool, while `Buffer.allocUnsafe(size).fill(fill)` will use the internal Buffer pool if `size` is less than or equal to half `Buffer.poolSize`. The difference is subtle but can be important when an application requires the additional performance that `Buffer.allocUnsafe(size)` provides.

Class Method: `Buffer.allocUnsafeSlow(size)`

#

Added in: v5.10.0

- `size` `<Number>`

Allocates a new *non-zero-filled* and non-pooled `Buffer` of `size` bytes. The `size` must be less than or equal to the value of `require('buffer').kMaxLength` (on 64-bit architectures, `kMaxLength` is `(231-1)`). Otherwise, a `RangeError` is thrown. A zero-length Buffer will be created if a `size` less than or equal to 0 is specified.

The underlying memory for `Buffer` instances created in this way is *not initialized*.

The contents of the newly created `Buffer` are unknown and *may contain sensitive data*. Use `buf.fill(0)` to initialize such `Buffer` instances to zeroes.

When using `Buffer.allocUnsafe()` to allocate new `Buffer` instances, allocations under 4KB are, by default, sliced from a single pre-allocated `Buffer`. This allows applications to avoid the garbage collection overhead of creating many individually allocated Buffers. This approach improves both performance and memory usage by eliminating the need to track and cleanup as many `Persistent` objects.

However, in the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may be appropriate to create an un-pooled Buffer instance using `Buffer.allocUnsafeSlow()` then copy out the relevant bits.

```
// need to keep around a few small chunks of memory
const store = [];

socket.on('readable', () => {
  const data = socket.read();
  // allocate for retained data
  const sb = Buffer.allocUnsafeSlow(10);
  // copy the data into the new allocation
  data.copy(sb, 0, 0, 10);
  store.push(sb);
});
```

Use of `Buffer.allocUnsafeSlow()` should be used only as a last resort *after* a developer has observed undue memory retention in their applications.

A `TypeError` will be thrown if `size` is not a number.

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

- `string <String> | <Buffer> | <TypedArray> | <DataView> |`

<ArrayBuffer>

- **encoding** <String> Default: 'utf8'
- Return: <Number>

Returns 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:

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

console.log(`#${str}: ${str.length} characters, ` +
  `#${Buffer.byteLength(str, 'utf8')} bytes`);

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

When `string` is a `Buffer / DataView / TypedArray / ArrayBuffer`, returns the actual byte length.

Otherwise, converts to `String` and returns the byte length of string.

Class Method: Buffer.compare(buf1, buf2)

Added in: v0.11.13

- **buf1** <Buffer>
- **buf2** <Buffer>
- Return: <Number>

Compares `buf1` to `buf2` typically for the purpose of sorting arrays of Buffers. This is equivalent is calling `buf1.compare(buf2)`.

```
const arr = [Buffer.from('1234'), Buffer.from('0123')];
arr.sort(Buffer.compare);
```

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

Added in: v0.7.11

- `list` <Array> List of Buffer objects to concat
- `totalLength` <Number> Total length of the Buffers in the list when concatenated
- Return: <Buffer>

Returns a new Buffer which is the result of concatenating all the Buffers in the `list` together.

If the list has no items, or if the `totalLength` is 0, then a new zero-length Buffer is returned.

If `totalLength` is not provided, it is calculated from the Buffers in the `list`. This, however, adds an additional loop to the function, so it is faster to provide the length explicitly.

Example: build a single Buffer from a list of three Buffers:

```
const buf1 = Buffer.alloc(10);
const buf2 = Buffer.alloc(14);
const buf3 = Buffer.alloc(18);
const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength);
const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);
console.log(bufA);
console.log(bufA.length);

// 42
// <Buffer 00 00 00 00 ...>
// 42
```

Class Method: Buffer.from(array)

Added in: v3.0.0

- `array <Array>`

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

```
const buf = Buffer.from([0x62,0x75,0x66,0x66,0x65,0x72]);
// creates a new Buffer containing ASCII bytes
// ['b','u','f','f','e','r']
```

A `TypeError` will be thrown if `array` is not an `Array`.

Class Method: Buffer.from(arrayBuffer[, byteOffset[, length]])

Added in: v5.10.0

- `arrayBuffer <ArrayBuffer>` The `.buffer` property of a `TypedArray` or a `new ArrayBuffer()`
- `byteOffset <Number>` Default: `0`
- `length <Number>` Default: `arrayBuffer.length - byteOffset`

When passed a reference to the `.buffer` property of a `TypedArray` instance, the newly created `Buffer` will share the same allocated memory as the `TypedArray`.

```
const arr = new Uint16Array(2);
arr[0] = 5000;
arr[1] = 4000;

const buf = Buffer.from(arr.buffer); // shares the memory with arr;

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// changing the TypedArray changes the Buffer also
arr[1] = 6000;
```

```
console.log(buf);
  // Prints: <Buffer 88 13 70 17>
```

The optional `byteOffset` and `length` arguments specify a memory range within the `arrayBuffer` that will be shared by the `Buffer`.

```
const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);
console.log(buf.length);
  // Prints: 2
```

A `TypeError` will be thrown if `arrayBuffer` is not an `ArrayBuffer`.

Class Method: `Buffer.from(buffer)`

Added in: v3.0.0

- `buffer <Buffer>`

Copies the passed `buffer` data onto a new `Buffer` instance.

```
const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;
console.log(buf1.toString());
  // 'auffer'
console.log(buf2.toString());
  // 'buffer' (copy is not changed)
```

A `TypeError` will be thrown if `buffer` is not a `Buffer`.

Class Method: `Buffer.from(str[, encoding])`

- `str` `<String>` String to encode.
- `encoding` `<String>` Encoding to use, Default: `'utf8'`

Creates a new `Buffer` containing the given JavaScript string `str`. If provided, the `encoding` parameter identifies the character encoding. If not provided, `encoding` defaults to `'utf8'`.

```
const buf1 = Buffer.from('this is a tést');
console.log(buf1.toString());
  // prints: this is a tést
console.log(buf1.toString('ascii'));
  // prints: this is a tC)st

const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');
console.log(buf2.toString());
  // prints: this is a tést
```

A `TypeError` will be thrown if `str` is not a string.

Class Method: `Buffer.isBuffer(obj)`

- `obj` `<Object>`
- Return: `<Boolean>`

Returns 'true' if `obj` is a Buffer.

Class Method: `Buffer.isEncoding(encoding)`

Added in: v0.9.1

- `encoding` `<String>` The encoding string to test
- Return: `<Boolean>`

Returns true if the `encoding` is a valid encoding argument, or false otherwise.

`buf[index]`

The index operator `[index]` can be used to get and set the octet at position `index` in the Buffer. The values refer to individual bytes, so the legal value range is between `0x00` and `0xFF` (hex) or `0` and `255` (decimal).

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

```
const str = "Node.js";
const buf = Buffer.allocUnsafe(str.length);

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

console.log(buf.toString('ascii'));
// Prints: Node.js
```

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

Added in: v0.11.13

- `target` `<Buffer>`
- `targetStart` `<Integer>` The offset within `target` at which to begin comparison. default = `0`.
- `targetEnd` `<Integer>` The offset with `target` at which to end comparison. Ignored when `targetStart` is `undefined`. default = `target.byteLength`.
- `sourceStart` `<Integer>` The offset within `buf` at which to begin comparison. Ignored when `targetStart` is `undefined`. default = `0`
- `sourceEnd` `<Integer>` The offset within `buf` at which to end comparison. Ignored when `targetStart` is `undefined`. default = `buf.byteLength`.
- Return: `<Number>`

Compares two Buffer instances and returns a number indicating whether `buf` comes before, after, or is the same as the `target` in sort order. Comparison is based on the actual sequence of bytes in each Buffer.

- `0` is returned if `target` is the same as `buf`
- `1` is returned if `target` should come *before* `buf` when sorted.
- `-1` is returned if `target` should come *after* `buf` when sorted.

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('BCD');
const buf3 = Buffer.from('ABCD');

console.log(buf1.compare(buf1));
  // Prints: 0

console.log(buf1.compare(buf2));
  // Prints: -1

console.log(buf1.compare(buf3));
  // Prints: 1

console.log(buf2.compare(buf1));
  // Prints: 1

console.log(buf2.compare(buf3));
  // Prints: 1

[buf1, buf2, buf3].sort(Buffer.compare);
  // produces sort order [buf1, buf3, buf2]
```

The optional `targetStart`, `targetEnd`, `sourceStart`, and `sourceEnd` arguments can be used to limit the comparison to specific ranges within the two `Buffer` objects.

```
const buf1 = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8, 9]);
const buf2 = Buffer.from([5, 6, 7, 8, 9, 1, 2, 3, 4]);

console.log(buf1.compare(buf2, 5, 9, 0, 4));
  // Prints: 0

console.log(buf1.compare(buf2, 0, 6, 4));
```

```
// Prints: -1  
console.log(buf1.compare(buf2, 5, 6, 5));  
// Prints: 1
```

A `RangeError` will be thrown if: `targetStart < 0`, `sourceStart < 0`, `targetEnd > target.byteLength` or `sourceEnd > source.byteLength`.

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

- `targetBuffer` `<Buffer>` Buffer to copy into
- `targetStart` `<Number>` Default: 0
- `sourceStart` `<Number>` Default: 0
- `sourceEnd` `<Number>` Default: `buffer.length`
- Return: `<Number>` The number of bytes copied.

Copies data from a region of this Buffer to a region in the target Buffer even if the target memory region overlaps with the source.

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

```
const buf1 = Buffer.allocUnsafe(26);  
const buf2 = Buffer.allocUnsafe(26).fill('!');  
  
for (let i = 0 ; i < 26 ; i++) {  
    buf1[i] = i + 97; // 97 is ASCII a  
}  
  
buf1.copy(buf2, 8, 16, 20);  
console.log(buf2.toString('ascii', 0, 25));  
// Prints: !!!!!!!qrst!!!!!!!!!!
```

Example: Build a single Buffer, then copy data from one region to an overlapping

region in the same Buffer

```
const buf = Buffer.allocUnsafe(26);

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

buf.copy(buf, 0, 4, 10);
console.log(buf.toString());

// efghijhijklmnopqrstuvwxyz
```

buf.entries()

#

Added in: v1.1.0

- Return: <Iterator>

Creates and returns an iterator of [index, byte] pairs from the Buffer contents.

```
const buf = Buffer.from('buffer');
for (var pair of buf.entries()) {
  console.log(pair);
}

// prints:
//  [0, 98]
//  [1, 117]
//  [2, 102]
//  [3, 102]
//  [4, 101]
//  [5, 114]
```

buf.equals(otherBuffer)

Added in: v1.0.0

- `otherBuffer <Buffer>`
- Return: `<Boolean>`

Returns a boolean indicating whether `this` and `otherBuffer` have exactly the same bytes.

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));
  // Prints: true
console.log(buf1.equals(buf3));
  // Prints: false
```

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

Added in: v0.5.0

- `value <String> | <Buffer> | <Number>`
- `offset <Number>` Default: 0
- `end <Number>` Default: `buf.length`
- `encoding <String>` Default: 'utf8'
- Return: `<Buffer>`

Fills the Buffer with the specified value. If the `offset` (defaults to 0) and `end` (defaults to `buf.length`) are not given the entire buffer will be filled. The method returns a reference to the Buffer, so calls can be chained. This is meant as a small simplification to creating a Buffer. Allowing the creation and fill of the Buffer to be done on a single line:

`encoding` is only relevant if `value` is a string. Otherwise it is ignored. `value` is coerced to a `uint32` value if it is not a String or Number.

The `fill()` operation writes bytes into the Buffer dumbly. If the final write falls in between a multi-byte character then whatever bytes fit into the buffer are written.

```
Buffer(3).fill('\u0222');  
// Prints: <Buffer c8 a2 c8>
```

`buf.indexOf(value[, byteOffset][, encoding])`

Added in: v1.5.0

- `value` `<String> | <Buffer> | <Number>`
 - `byteOffset` `<Number>` Default: 0
 - `encoding` `<String>` Default: 'utf8'
 - Return: `<Number>`

Operates similar to `Array#indexOf()` in that it returns either the starting index position of `value` in Buffer or `-1` if the Buffer does not contain `value`. The `value` can be a String, Buffer or Number. Strings are by default interpreted as UTF8. Buffers will use the entire Buffer (to compare a partial Buffer use `buf.slice()`). Numbers can range from 0 to 255.

```
const buf = Buffer.from('this is a buffer');

buf.indexOf('this');
// returns 0

buf.indexOf('is');
```

```

    // returns 2
buf.indexOf(Buffer.from('a buffer'));
    // returns 8
buf.indexOf(97); // ascii for 'a'
    // returns 8
buf.indexOf(Buffer.from('a buffer example'));
    // returns -1
buf.indexOf(Buffer.from('a buffer example')).slice(0,8));
    // returns 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'ucs2'

utf16Buffer.indexOf('\u03a3', 0, 'ucs2');
    // returns 4
utf16Buffer.indexOf('\u03a3', -4, 'ucs2');
    // returns 6

```

buf.includes(value[, byteOffset][, encoding])

#

Added in: v5.3.0

- `value` `<String> | <Buffer> | <Number>`
- `byteOffset` `<Number>` Default: 0
- `encoding` `<String>` Default: `'utf8'`
- Return: `<Boolean>`

Operates similar to `Array#include()`. The `value` can be a String, Buffer or Number. Strings are interpreted as UTF8 unless overridden with the `encoding` argument. Buffers will use the entire Buffer (to compare a partial Buffer use `buf.slice()`). Numbers can range from 0 to 255.

The `byteOffset` indicates the index in `buf` where searching begins.

```
const buf = Buffer.from('this is a buffer');
```

```
buf.includes('this');
  // returns true
buf.includes('is');
  // returns true
buf.includes(Buffer.from('a buffer'));
  // returns true
buf.includes(97); // ascii for 'a'
  // returns true
buf.includes(Buffer.from('a buffer example'));
  // returns false
buf.includes(Buffer.from('a buffer example').slice(0,8));
  // returns true
buf.includes('this', 4);
  // returns false
```

buf.keys()

#

Added in: v1.1.0

- Return: <Iterator>

Creates and returns an **iterator** of Buffer keys (indices).

```
const buf = Buffer.from('buffer');
for (var key of buf.keys()) {
  console.log(key);
}
// prints:
//  0
//  1
//  2
//  3
//  4
```

buf.lastIndexOf(value[, byteOffset][, encoding])

#

Added in: v6.0.0

- `value` <String> | <Buffer> | <Number>
- `byteOffset` <Number> Default: `buf.length`
- `encoding` <String> Default: 'utf8'
- Return: <Number>

Identical to `Buffer#indexOf()`, but searches the Buffer from back to front instead of front to back. Returns the starting index position of `value` in Buffer or `-1` if the Buffer does not contain `value`. The `value` can be a String, Buffer or Number. Strings are by default interpreted as UTF8. If `byteOffset` is provided, will return the last match that begins at or before `byteOffset`.

```
const buf = new Buffer('this buffer is a buffer');
```

```
buf.lastIndexOf('this');
  // returns 0

buf.lastIndexOf('buffer');
  // returns 17

buf.lastIndexOf(new Buffer('buffer'));
  // returns 17

buf.lastIndexOf(97); // ascii for 'a'
  // returns 15

buf.lastIndexOf(new Buffer('yolo'));
  // returns -1

buf.lastIndexOf('buffer', 5)
  // returns 5

buf.lastIndexOf('buffer', 4)
  // returns -1
```

```
const utf16Buffer = new Buffer('\u039a\u0391\u03a3\u03a3\u0395', 'ucs2')

utf16Buffer.lastIndexOf('\u03a3', null, 'ucs2');
  // returns 6
utf16Buffer.lastIndexOf('\u03a3', -5, 'ucs2');
  // returns 4
```

buf.length

- <Number>

Returns the amount of memory allocated for the Buffer in number of bytes. Note that this does not necessarily reflect the amount of usable data within the Buffer. For instance, in the example below, a Buffer with 1234 bytes is allocated, but only 11 ASCII bytes are written.

```
const buf = Buffer.alloc(1234);

console.log(buf.length);
  // Prints: 1234

buf.write('some string', 0, 'ascii');
console.log(buf.length);
  // Prints: 1234
```

While the `length` property is not immutable, changing the value of `length` can result in undefined and inconsistent behavior. Applications that wish to modify the length of a Buffer should therefore treat `length` as read-only and use `buf.slice()` to create a new Buffer.

```
var buf = Buffer.allocUnsafe(10);

buf.write('abcdefghijklm', 0, 'ascii');
console.log(buf.length);
```

```
// Prints: 10  
buf = buf.slice(0,5);  
console.log(buf.length);  
// Prints: 5
```

buf.readDoubleBE(offset[, noAssert]) #

buf.readDoubleLE(offset[, noAssert]) #

- `offset <Number>` `0 <= offset <= buf.length - 8`
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads a 64-bit double from the Buffer at the specified `offset` with specified endian format (`readDoubleBE()` returns big endian, `readDoubleLE()` returns little endian).

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

```
const buf = Buffer.from([1,2,3,4,5,6,7,8]);  
  
buf.readDoubleBE();  
  // Returns: 8.20788039913184e-304  
buf.readDoubleLE();  
  // Returns: 5.447603722011605e-270  
buf.readDoubleLE(1);  
  // throws RangeError: Index out of range  
  
buf.readDoubleLE(1, true); // Warning: reads passed end of buffer!  
  // Segmentation fault! don't do this!
```

buf.readFloatBE(offset[, noAssert]) #

buf.readFloatLE(offset[, noAssert])

- `offset <Number>` $0 \leq offset \leq buf.length - 4$
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads a 32-bit float from the Buffer at the specified `offset` with specified endian format (`readFloatBE()` returns big endian, `readFloatLE()` returns little endian).

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

```
const buf = Buffer.from([1,2,3,4]);

buf.readFloatBE();
  // Returns: 2.387939260590663e-38
buf.readFloatLE();
  // Returns: 1.539989614439558e-36
buf.readFloatLE(1);
  // throws RangeError: Index out of range

buf.readFloatLE(1, true); // Warning: reads past end of buffer!
  // Segmentation fault! don't do this!
```

buf.readInt8(offset[, noAssert])

- `offset <Number>` $0 \leq offset \leq buf.length - 1$
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads a signed 8-bit integer from the Buffer at the specified `offset`.

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

Integers read from the Buffer are interpreted as two's complement signed values.

```
const buf = Buffer.from([1,-2,3,4]);  
  
buf.readInt8(0);  
  // returns 1  
buf.readInt8(1);  
  // returns -2
```

buf.readInt16BE(offset[, noAssert])

#

buf.readInt16LE(offset[, noAssert])

#

- `offset <Number>` $0 \leq offset \leq buf.length - 2$
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads a signed 16-bit integer from the Buffer at the specified `offset` with the specified endian format (`readInt16BE()` returns big endian, `readInt16LE()` returns little endian).

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

Integers read from the Buffer are interpreted as two's complement signed values.

```
const buf = Buffer.from([1,-2,3,4]);  
  
buf.readInt16BE();  
  // returns 510  
buf.readInt16LE(1);  
  // returns 1022
```

buf.readInt32BE(offset[, noAssert])

#

buf.readInt32LE(offset[, noAssert])

#

- offset <Number> $0 \leq \text{offset} \leq \text{buf.length} - 4$
- noAssert <Boolean> Default: false
- Return: <Number>

Reads a signed 32-bit integer from the Buffer at the specified offset with the specified endian format (`readInt32BE()` returns big endian, `readInt32LE()` returns little endian).

Setting noAssert to true skips validation of the offset. This allows the offset to be beyond the end of the Buffer.

Integers read from the Buffer are interpreted as two's complement signed values.

```
const buf = Buffer.from([1,-2,3,4]);  
  
buf.readInt32BE();  
  // returns 33424132  
buf.readInt32LE();  
  // returns 67370497  
buf.readInt32LE(1);  
  // throws RangeError: Index out of range
```

buf.readIntBE(offset, byteLength[, noAssert])

#

buf.readIntLE(offset, byteLength[, noAssert])

#

Added in: v1.0.0

- offset <Number> $0 \leq \text{offset} \leq \text{buf.length} - \text{byteLength}$
- byteLength <Number> $0 < \text{byteLength} \leq 6$
- noAssert <Boolean> Default: false
- Return: <Number>

Reads `byteLength` number of bytes from the Buffer at the specified `offset` and interprets the result as a two's complement signed value. Supports up to 48 bits of accuracy. For example:

```
const buf = Buffer.allocUnsafe(6);
buf.writeUInt16LE(0x90ab, 0);
buf.writeUInt32LE(0x12345678, 2);
buf.readIntLE(0, 6).toString(16); // Specify 6 bytes (48 bits)
// Returns: '1234567890ab'

buf.readIntBE(0, 6).toString(16);
// Returns: -546f87a9cbee
```

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

buf.readUInt8(offset[, noAssert])

- `offset <Number>` $0 \leqslant \text{offset} \leqslant \text{buf.length} - 1$
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads an unsigned 8-bit integer from the Buffer at the specified `offset`.

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

```
const buf = Buffer.from([1,-2,3,4]);

buf.readUInt8(0);
  // returns 1
buf.readUInt8(1);
  // returns 254
```

buf.readUInt16BE(offset[, noAssert])

#

buf.readUInt16LE(offset[, noAssert])

#

- offset <Number> $0 \leq \text{offset} \leq \text{buf.length} - 2$
- noAssert <Boolean> Default: false
- Return: <Number>

Reads an unsigned 16-bit integer from the Buffer at the specified `offset` with specified endian format (`readUInt16BE()` returns big endian, `readUInt16LE()` returns little endian).

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

Example:

```
const buf = Buffer.from([0x3, 0x4, 0x23, 0x42]);

buf.readUInt16BE(0);
  // Returns: 0x0304

buf.readUInt16LE(0);
  // Returns: 0x0403

buf.readUInt16BE(1);
  // Returns: 0x0423

buf.readUInt16LE(1);
  // Returns: 0x2304

buf.readUInt16BE(2);
  // Returns: 0x2342

buf.readUInt16LE(2);
  // Returns: 0x4223
```

buf.readUInt32BE(offset[, noAssert])

#

buf.readUInt32LE(offset[, noAssert])

#

- `offset <Number>` $0 \leq offset \leq buf.length - 4$
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads an unsigned 32-bit integer from the Buffer at the specified `offset` with specified endian format (`readUInt32BE()` returns big endian, `readUInt32LE()` returns little endian).

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

Example:

```
const buf = Buffer.from([0x3, 0x4, 0x23, 0x42]);

buf.readUInt32BE(0);
  // Returns: 0x03042342
console.log(buf.readUInt32LE(0));
  // Returns: 0x42230403
```

`buf.readUIntBE(offset, byteLength[, noAssert])` #

`buf.readUIntLE(offset, byteLength[, noAssert])` #

Added in: v1.0.0

- `offset <Number>` $0 \leq offset \leq buf.length - byteLength$
- `byteLength <Number>` $0 < byteLength \leq 6$
- `noAssert <Boolean>` Default: false
- Return: `<Number>`

Reads `byteLength` number of bytes from the Buffer at the specified `offset` and interprets the result as an unsigned integer. Supports up to 48 bits of accuracy. For example:

```
const buf = Buffer.allocUnsafe(6);
buf.writeUInt16LE(0x90ab, 0);
buf.writeUInt32LE(0x12345678, 2);
buf.readUIntLE(0, 6).toString(16); // Specify 6 bytes (48 bits)
// Returns: '1234567890ab'

buf.readUIntBE(0, 6).toString(16);
// Returns: ab9078563412
```

Setting `noAssert` to `true` skips validation of the `offset`. This allows the `offset` to be beyond the end of the Buffer.

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

- `start <Number>` Default: 0
- `end <Number>` Default: `buffer.length`
- Return: `<Buffer>`

Returns a new Buffer that references the same memory as the original, but offset and cropped by the `start` and `end` indices.

Note that modifying the new Buffer slice will modify the memory in the original Buffer because the allocated memory of the two objects overlap.

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

```
const buf1 = Buffer.allocUnsafe(26);

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

const buf2 = buf1.slice(0, 3);
```

```
buf2.toString('ascii', 0, buf2.length);
  // Returns: 'abc'
buf1[0] = 33;
buf2.toString('ascii', 0, buf2.length);
  // Returns : '!bc'
```

Specifying negative indexes causes the slice to be generated relative to the end of the Buffer rather than the beginning.

```
const buf = Buffer.from('buffer');

buf.slice(-6, -1).toString();
  // Returns 'buffe', equivalent to buf.slice(0, 5)
buf.slice(-6, -2).toString();
  // Returns 'buff', equivalent to buf.slice(0, 4)
buf.slice(-5, -2).toString();
  // Returns 'uff', equivalent to buf.slice(1, 4)
```

buf.swap16()

#

Added in: v5.10.0

- Return: <Buffer>

Interprets the `Buffer` as an array of unsigned 16-bit integers and swaps the byte-order *in-place*. Throws a `RangeError` if the `Buffer` length is not a multiple of 16 bits. The method returns a reference to the Buffer, so calls can be chained.

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);
console.log(buf);
  // Prints <Buffer 01 02 03 04 05 06 07 08>
buf.swap16();
console.log(buf);
  // Prints <Buffer 02 01 04 03 06 05 08 07>
```

buf.swap32()

#

Added in: v5.10.0

- Return: <Buffer>

Interprets the `Buffer` as an array of unsigned 32-bit integers and swaps the byte-order *in-place*. Throws a `RangeError` if the `Buffer` length is not a multiple of 32 bits. The method returns a reference to the Buffer, so calls can be chained.

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);
console.log(buf);
  // Prints <Buffer 01 02 03 04 05 06 07 08>
buf.swap32();
console.log(buf);
  // Prints <Buffer 04 03 02 01 08 07 06 05>
```

buf.swap64()

#

Added in: v6.3.0

- Return: <Buffer>

Interprets the `Buffer` as an array of 64-bit numbers and swaps the byte-order *in-place*. Throws a `RangeError` if the `Buffer` length is not a multiple of 64 bits. The method returns a reference to the Buffer, so calls can be chained.

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);
console.log(buf);
  // Prints <Buffer 01 02 03 04 05 06 07 08>
buf.swap64();
console.log(buf);
  // Prints <Buffer 08 07 06 05 04 03 02 01>
```

Note that JavaScript cannot encode 64-bit integers. This method is intended for working with 64-bit floats.

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

- `encoding` <String> Default: 'utf8'
- `start` <Number> Default: 0
- `end` <Number> Default: `buffer.length`
- Return: <String>

Decodes and returns a string from the Buffer data using the specified character set `encoding`.

```
const buf = Buffer.allocUnsafe(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97; // 97 is ASCII a
}
buf.toString('ascii');
  // Returns: 'abcdefghijklmnopqrstuvwxyz'
buf.toString('ascii',0,5);
  // Returns: 'abcde'
buf.toString('utf8',0,5);
  // Returns: 'abcde'
buf.toString(undefined,0,5);
  // Returns: 'abcde', encoding defaults to 'utf8'
```

buf.toJSON()

Added in: v0.9.2

- Return: <Object>

Returns a JSON representation of the Buffer instance. `JSON.stringify()` implicitly calls this function when stringifying a Buffer instance.

Example:

```
const buf = Buffer.from('test');
const json = JSON.stringify(buf);

console.log(json);
// Prints: '{"type":"Buffer","data":[116,101,115,116]}'

const copy = JSON.parse(json, (key, value) => {
    return value && value.type === 'Buffer'
        ? Buffer.from(value.data)
        : value;
});

console.log(copy.toString());
// Prints: 'test'
```

buf.values()

#

Added in: v1.1.0

- Return: <Iterator>

Creates and returns an **iterator** for Buffer values (bytes). This function is called automatically when the Buffer is used in a `for .. of` statement.

```
const buf = Buffer.from('buffer');
for (var value of buf.values()) {
    console.log(value);
}

// prints:
// 98
// 117
// 102
// 102
// 101
```

```
// 114
```

```
for (var value of buf) {
    console.log(value);
}
// prints:
// 98
// 117
// 102
// 102
// 101
// 114
```

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

- `string <String>` Bytes to be written to buffer
- `offset <Number>` Default: 0
- `length <Number>` Default: `buffer.length - offset`
- `encoding <String>` Default: 'utf8'
- Return: <Number> Numbers of bytes written

Writes `string` to the Buffer at `offset` using the given `encoding`. The `length` parameter is the number of bytes to write. If the Buffer did not contain enough space to fit the entire string, only a partial amount of the string will be written however, it will not write only partially encoded characters.

```
const buf = Buffer.allocUnsafe(256);
const len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(` ${len} bytes: ${buf.toString('utf8', 0, len)} `);
// Prints: 12 bytes: ½ + ¼ = ¾
```

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

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

- `value` <Number> Bytes to be written to Buffer
- `offset` <Number> $0 \leqslant \text{offset} \leqslant \text{buf.length} - 8$
- `noAssert` <Boolean> Default: false
- Return: <Number> The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` with specified endian format (`writeDoubleBE()` writes big endian, `writeDoubleLE()` writes little endian). The `value` argument *should* be a valid 64-bit double. Behavior is not defined when `value` is anything other than a 64-bit double.

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

Example:

```
const buf = Buffer.allocUnsafe(8);
buf.writeDoubleBE(0xdeadbeefcafebabe, 0);

console.log(buf);
// Prints: <Buffer 43 eb d5 b7 dd f9 5f d7>

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);
// Prints: <Buffer d7 5f f9 dd b7 d5 eb 43>
```

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

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

- `value` <Number> Bytes to be written to Buffer

- `offset <Number>` $0 \leq offset \leq buf.length - 4$
- `noAssert <Boolean>` Default: false
- Return: `<Number>` The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` with specified endian format (`writeFloatBE()` writes big endian, `writeFloatLE()` writes little endian).

Behavior is not defined when `value` is anything other than a 32-bit float.

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

Example:

```
const buf = Buffer.allocUnsafe(4);
buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer bb fe 4a 4f>
```

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

- `value <Number>` Bytes to be written to Buffer
- `offset <Number>` $0 \leq offset \leq buf.length - 1$
- `noAssert <Boolean>` Default: false
- Return: `<Number>` The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset`. The `value` should be a valid

signed 8-bit integer. Behavior is not defined when `value` is anything other than a signed 8-bit integer.

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

The `value` is interpreted and written as a two's complement signed integer.

```
const buf = Buffer.allocUnsafe(2);
buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);
console.log(buf);
// Prints: <Buffer 02 fe>
```

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

#

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

#

- `value` <Number> Bytes to be written to Buffer
- `offset` <Number> $0 \leqslant \text{offset} \leqslant \text{buf.length} - 2$
- `noAssert` <Boolean> Default: false
- Return: <Number> The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` with specified endian format (`writeInt16BE()` writes big endian, `writeInt16LE()` writes little endian). The `value` should be a valid signed 16-bit integer. Behavior is not defined when `value` is anything other than a signed 16-bit integer.

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

The `value` is interpreted and written as a two's complement signed integer.

```
const buf = Buffer.allocUnsafe(4);
buf.writeInt16BE(0x0102, 0);
buf.writeInt16LE(0x0304, 2);
console.log(buf);
// Prints: <Buffer 01 02 04 03>
```

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

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

- `value` <Number> Bytes to be written to Buffer
- `offset` <Number> $0 \leq \text{offset} \leq \text{buf.length} - 4$
- `noAssert` <Boolean> Default: false
- Return: <Number> The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` with specified endian format (`writeInt32BE()` writes big endian, `writeInt32LE()` writes little endian). The `value` should be a valid signed 32-bit integer. Behavior is not defined when `value` is anything other than a signed 32-bit integer.

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

The `value` is interpreted and written as a two's complement signed integer.

```
const buf = Buffer.allocUnsafe(8);
buf.writeInt32BE(0x01020304, 0);
buf.writeInt32LE(0x05060708, 4);
console.log(buf);
// Prints: <Buffer 01 02 03 04 08 07 06 05>
```

`buf.writeIntBE(value, offset, byteLength[, noAssert])`

#

`buf.writeIntLE(value, offset, byteLength[, noAssert])`

#

Added in: v1.0.0

- `value <Number>` Bytes to be written to Buffer
- `offset <Number>` $0 \leq \text{offset} \leq \text{buf.length} - \text{byteLength}$
- `byteLength <Number>` $0 < \text{byteLength} \leq 6$
- `noAssert <Boolean>` Default: false
- Return: `<Number>` The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` and `byteLength`. Supports up to 48 bits of accuracy. For example:

```
const buf1 = Buffer.allocUnsafe(6);
buf1.writeUIntBE(0x1234567890ab, 0, 6);
console.log(buf1);
// Prints: <Buffer 12 34 56 78 90 ab>
```

```
const buf2 = Buffer.allocUnsafe(6);
buf2.writeUIntLE(0x1234567890ab, 0, 6);
console.log(buf2);
// Prints: <Buffer ab 90 78 56 34 12>
```

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

Behavior is not defined when `value` is anything other than an integer.

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

#

- `value <Number>` Bytes to be written to Buffer

- `offset <Number>` $0 \leq offset \leq buf.length - 1$
- `noAssert <Boolean>` Default: false
- Return: `<Number>` The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset`. The `value` should be a valid unsigned 8-bit integer. Behavior is not defined when `value` is anything other than an unsigned 8-bit integer.

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

Example:

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

console.log(buf);
// Prints: <Buffer 03 04 23 42>
```

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

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

- `value <Number>` Bytes to be written to Buffer
- `offset <Number>` $0 \leq offset \leq buf.length - 2$
- `noAssert <Boolean>` Default: false
- Return: `<Number>` The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` with specified endian format (`writeUInt16BE()` writes big endian, `writeUInt16LE()` writes little endian). The

`value` should be a valid unsigned 16-bit integer. Behavior is not defined when `value` is anything other than an unsigned 16-bit integer.

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

Example:

```
const buf = Buffer.allocUnsafe(4);
buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>

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

console.log(buf);
// Prints: <Buffer ad de ef be>
```

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

#

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

#

- `value` <Number> Bytes to be written to Buffer
- `offset` <Number> $0 \leq \text{offset} \leq \text{buf.length} - 4$
- `noAssert` <Boolean> Default: false
- Return: <Number> The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` with specified endian format (`writeUInt32BE()` writes big endian, `writeUInt32LE()` writes little endian). The

`value` should be a valid unsigned 32-bit integer. Behavior is not defined when `value` is anything other than an unsigned 32-bit integer.

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

Example:

```
const buf = Buffer.allocUnsafe(4);
buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer fe ed fa ce>

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>
```

`buf.writeUIntBE(value, offset, byteLength[, noAssert])` #

`buf.writeUIntLE(value, offset, byteLength[, noAssert])` #

- `value` <Number> Bytes to be written to Buffer
- `offset` <Number> $0 \leq offset \leq buf.length - byteLength$
- `byteLength` <Number> $0 < byteLength \leq 6$
- `noAssert` <Boolean> Default: false
- Return: <Number> The offset plus the number of written bytes

Writes `value` to the Buffer at the specified `offset` and `byteLength`. Supports up to 48 bits of accuracy. For example:

```
const buf = Buffer.allocUnsafe(6);
buf.writeUIntBE(0x1234567890ab, 0, 6);
console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

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

Behavior is not defined when `value` is anything other than an unsigned integer.

buffer.INSPECT_MAX_BYTES

#

- <Number> Default: 50

Returns the maximum number of bytes that will be returned when `buffer.inspect()` is called. This can be overridden by user modules. See [util.inspect\(\)](#) for more details on `buffer.inspect()` behavior.

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

Class: SlowBuffer

#

Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use [Buffer.allocUnsafeSlow\(size\)](#) instead.

Returns an un-pooled `Buffer`.

In order to avoid the garbage collection overhead of creating many individually allocated Buffers, by default allocations under 4KB are sliced from a single larger allocated object. This approach improves both performance and memory usage

since v8 does not need to track and cleanup as many `Persistent` objects.

In the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may be appropriate to create an unpooled Buffer instance using `SlowBuffer` then copy out the relevant bits.

```
// need to keep around a few small chunks of memory
const store = [];

socket.on('readable', () => {
  var data = socket.read();
  // allocate for retained data
  var sb = SlowBuffer(10);
  // copy the data into the new allocation
  data.copy(sb, 0, 0, 10);
  store.push(sb);
});
```

Use of `SlowBuffer` should be used only as a last resort *after* a developer has observed undue memory retention in their applications.

`new SlowBuffer(size)`

#

Deprecated since: v6.0.0

Stability: 0 - Deprecated: Use
[Buffer.allocUnsafeSlow\(size\)](#) instead.

- `size` Number

Allocates a new `SlowBuffer` of `size` bytes. The `size` must be less than or equal to the value of `require('buffer').kMaxLength` (on 64-bit architectures, `kMaxLength` is `(231) - 1`). Otherwise, a `RangeError` is thrown. A zero-length Buffer will be created if a `size` less than or equal to 0 is specified.

The underlying memory for `SlowBuffer` instances is *not initialized*. The contents of a newly created `SlowBuffer` are unknown and could contain sensitive data. Use `buf.fill(0)` to initialize a `SlowBuffer` to zeroes.

```
const SlowBuffer = require('buffer').SlowBuffer;
const buf = new SlowBuffer(5);
console.log(buf);
// <Buffer 78 e0 82 02 01>
// (octets will be different, every time)
buf.fill(0);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

Child Process

Stability: 2 - Stable

The `child_process` module provides the ability to spawn child processes in a manner that is similar, but not identical, to `popen(3)`. This capability is primarily provided by the `child_process.spawn()` function:

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

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});
```

```
ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

By default, pipes for `stdin`, `stdout` and `stderr` are established between the parent Node.js process and the spawned child. It is possible to stream data through these pipes in a non-blocking way. Note, however, that some programs use line-buffered I/O internally. While that does not affect Node.js, it can mean that data sent to the child process may not be immediately consumed.

The `child_process.spawn()` method spawns the child process asynchronously, without blocking the Node.js event loop. The `child_process.spawnSync()` function provides equivalent functionality in a synchronous manner that blocks the event loop until the spawned process either exits or is terminated.

For convenience, the `child_process` module provides a handful of synchronous and asynchronous alternatives to `child_process.spawn()` and `child_process.spawnSync()`. Note that each of these alternatives are implemented on top of `child_process.spawn()` or `child_process.spawnSync()`.

- `child_process.exec()`: spawns a shell and runs a command within that shell, passing the `stdout` and `stderr` to a callback function when complete.
- `child_process.execFile()`: similar to `child_process.exec()` except that it spawns the command directly without first spawning a shell.
- `child_process.fork()`: spawns a new Node.js process and invokes a specified module with an IPC communication channel established that allows sending messages between parent and child.
- `child_process.execSync()`: a synchronous version of `child_process.exec()` that will block the Node.js event loop.
- `child_process.execFileSync()`: a synchronous version of `child_process.execFile()` that will block the Node.js event loop.

For certain use cases, such as automating shell scripts, the `synchronous counterparts` may be more convenient. In many cases, however, the synchronous

methods can have significant impact on performance due to stalling the event loop while spawned processes complete.

Asynchronous Process Creation

The `child_process.spawn()`, `child_process.fork()`, `child_process.exec()`, and `child_process.execFile()` methods all follow the idiomatic asynchronous programming pattern typical of other Node.js APIs.

Each of the methods returns a `ChildProcess` instance. These objects implement the Node.js `EventEmitter` API, allowing the parent process to register listener functions that are called when certain events occur during the life cycle of the child process.

The `child_process.exec()` and `child_process.execFile()` methods additionally allow for an optional `callback` function to be specified that is invoked when the child process terminates.

Spawning .bat and .cmd files on Windows

The importance of the distinction between `child_process.exec()` and `child_process.execFile()` can vary based on platform. On Unix-type operating systems (Unix, Linux, OSX) `child_process.execFile()` can be more efficient because it does not spawn a shell. On Windows, however, `.bat` and `.cmd` files are not executable on their own without a terminal, and therefore cannot be launched using `child_process.execFile()`. When running on Windows, `.bat` and `.cmd` files can be invoked using `child_process.spawn()` with the `shell` option set, with `child_process.exec()`, or by spawning `cmd.exe` and passing the `.bat` or `.cmd` file as an argument (which is what the `shell` option and `child_process.exec()` do).

```
// On Windows Only ...
const spawn = require('child_process').spawn;
const bat = spawn('cmd.exe', ['/c', 'my.bat']);
```

```

bat.stdout.on('data', (data) => {
  console.log(data);
});

bat.stderr.on('data', (data) => {
  console.log(data);
});

bat.on('exit', (code) => {
  console.log(`Child exited with code ${code}`);
});

// OR...

const exec = require('child_process').exec;
exec('my.bat', (err, stdout, stderr) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(stdout);
});

```

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

#

Added in: v0.1.90

- **command** `<String>` The command to run, with space-separated arguments
- **options** `<Object>`
 - **cwd** `<String>` Current working directory of the child process
 - **env** `<Object>` Environment key-value pairs
 - **encoding** `<String>` (Default: `'utf8'`)
 - **shell** `<String>` Shell to execute the command with (Default: `'/bin/sh'` on UNIX, `'cmd.exe'` on Windows, The shell should understand the `-c` switch on UNIX or `/s /c` on Windows. On Windows,

command line parsing should be compatible with `cmd.exe`.)

- `timeout <Number>` (Default: `0`)
- `maxBuffer <Number>` largest amount of data (in bytes) allowed on `stdout` or `stderr` - if exceeded child process is killed (Default: `200*1024`)
- `killSignal <String>` (Default: `'SIGTERM'`)
- `uid <Number>` Sets the user identity of the process. (See `setuid(2)`.)
- `gid <Number>` Sets the group identity of the process. (See `setgid(2)`.)
- `callback <Function>` called with the output when process terminates
 - `error <Error>`
 - `stdout <String> | <Buffer>`
 - `stderr <String> | <Buffer>`
- Return: `<ChildProcess>`

Spawns a shell then executes the `command` within that shell, buffering any generated output.

```
const exec = require('child_process').exec;
exec('cat *.js bad_file | wc -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

If a `callback` function is provided, it is called with the arguments `(error, stdout, stderr)`. On success, `error` will be `null`. On error, `error` will be an instance of `Error`. The `error.code` property will be the exit code of the child process while `error.signal` will be set to the signal that terminated the process. Any exit code other than `0` is considered to be an error.

The `stdout` and `stderr` arguments passed to the callback will contain the `stdout` and `stderr` output of the child process. By default, Node.js will decode the output as UTF-8 and pass strings to the callback. The `encoding` option can be used to specify the character encoding used to decode the `stdout` and `stderr` output. If `encoding` is `'buffer'`, or an unrecognized character encoding, `Buffer` objects will be passed to the callback instead.

The `options` argument may be passed as the second argument to customize how the process is spawned. The default options are:

```
{  
  encoding: 'utf8',  
  timeout: 0,  
  maxBuffer: 200*1024,  
  killSignal: 'SIGTERM',  
  cwd: null,  
  env: null  
}
```

If `timeout` is greater than `0`, the parent will send the signal identified by the `killSignal` property (the default is `'SIGTERM'`) if the child runs longer than `timeout` milliseconds.

Note: Unlike the `exec(3)` POSIX system call, `child_process.exec()` does not replace the existing process and uses a shell to execute the command.

`child_process.execFile(file[, args][, options][, callback])`

#

Added in: v0.1.91

- `file` `<String>` The name or path of the executable file to run
- `args` `<Array>` List of string arguments
- `options` `<Object>`
 - `cwd` `<String>` Current working directory of the child process
 - `env` `<Object>` Environment key-value pairs

- `encoding` <String> (Default: 'utf8')
- `timeout` <Number> (Default: 0)
- `maxBuffer` <Number> largest amount of data (in bytes) allowed on stdout or stderr - if exceeded child process is killed (Default: 200*1024)
- `killSignal` <String> (Default: 'SIGTERM')
- `uid` <Number> Sets the user identity of the process. (See `setuid(2)`.)
- `gid` <Number> Sets the group identity of the process. (See `setgid(2)`.)
- `callback` <Function> called with the output when process terminates
 - `error` <Error>
 - `stdout` <String> | <Buffer>
 - `stderr` <String> | <Buffer>
- Return: <ChildProcess>

The `child_process.execFile()` function is similar to `child_process.exec()` except that it does not spawn a shell. Rather, the specified executable `file` is spawned directly as a new process making it slightly more efficient than `child_process.exec()`.

The same options as `child_process.exec()` are supported. Since a shell is not spawned, behaviors such as I/O redirection and file globbing are not supported.

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (error, stdout, stderr) =>
  if (error) {
    throw error;
  }
  console.log(stdout);
});
```

The `stdout` and `stderr` arguments passed to the callback will contain the stdout and stderr output of the child process. By default, Node.js will decode the output as UTF-8 and pass strings to the callback. The `encoding` option can be used to specify

the character encoding used to decode the stdout and stderr output. If `encoding` is `'buffer'`, or an unrecognized character encoding, `Buffer` objects will be passed to the callback instead.

`child_process.fork(modulePath[, args][, options])`

#

Added in: v0.5.0

- `modulePath` `<String>` The module to run in the child
- `args` `<Array>` List of string arguments
- `options` `<Object>`
 - `cwd` `<String>` Current working directory of the child process
 - `env` `<Object>` Environment key-value pairs
 - `execPath` `<String>` Executable used to create the child process
 - `execArgv` `<Array>` List of string arguments passed to the executable
(Default: `process.execArgv`)
 - `silent` `<Boolean>` If `true`, `stdin`, `stdout`, and `stderr` of the child will be piped to the parent, otherwise they will be inherited from the parent, see the `'pipe'` and `'inherit'` options for `child_process.spawn()`'s `stdio` for more details (Default: `false`)
 - `uid` `<Number>` Sets the user identity of the process. (See `setuid(2)`.)
 - `gid` `<Number>` Sets the group identity of the process. (See `setgid(2)`.)
- Return: `<ChildProcess>`

The `child_process.fork()` method is a special case of `child_process.spawn()` used specifically to spawn new Node.js processes. Like `child_process.spawn()`, a `ChildProcess` object is returned. The returned `ChildProcess` will have an additional communication channel built-in that allows messages to be passed back and forth between the parent and child. See `child.send()` for details.

It is important to keep in mind that spawned Node.js child processes are independent of the parent with exception of the IPC communication channel that is established between the two. Each process has its own memory, with their own V8 instances. Because of the additional resource allocations required, spawning a large number of child Node.js processes is not recommended.

By default, `child_process.fork()` will spawn new Node.js instances using the `process.execPath` of the parent process. The `execPath` property in the `options` object allows for an alternative execution path to be used.

Node.js processes launched with a custom `execPath` will communicate with the parent process using the file descriptor (fd) identified using the environment variable `NODE_CHANNEL_FD` on the child process. The input and output on this fd is expected to be line delimited JSON objects.

Note: Unlike the `fork(2)` POSIX system call, `child_process.fork()` does not clone the current process.

`child_process.spawn(command[, args][, options])`

#

Added in: v0.1.90

- `command` `<String>` The command to run
- `args` `<Array>` List of string arguments
- `options` `<Object>`
 - `cwd` `<String>` Current working directory of the child process
 - `env` `<Object>` Environment key-value pairs
 - `stdio` `<Array>` | `<String>` Child's stdio configuration. (See `options.stdio`)
 - `detached` `<Boolean>` Prepare child to run independently of its parent process. Specific behavior depends on the platform, see `options.detached`)
 - `uid` `<Number>` Sets the user identity of the process. (See `setuid(2)`.)
 - `gid` `<Number>` Sets the group identity of the process. (See `setgid(2)`.)
 - `shell` `<Boolean>` | `<String>` If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on UNIX, and `'cmd.exe'` on Windows. A different shell can be specified as a string. The shell should understand the `-c` switch on UNIX, or `/s /c` on Windows. Defaults to `false` (no shell).
- return: `<ChildProcess>`

The `child_process.spawn()` method spawns a new process using the given

`command`, with command line arguments in `args`. If omitted, `args` defaults to an empty array.

A third argument may be used to specify additional options, with these defaults:

```
{  
  cwd: undefined,  
  env: process.env  
}
```

Use `cwd` to specify the working directory from which the process is spawned. If not given, the default is to inherit the current working directory.

Use `env` to specify environment variables that will be visible to the new process, the default is `process.env`.

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

```
const spawn = require('child_process').spawn;  
const ls = spawn('ls', ['-lh', '/usr']);  
  
ls.stdout.on('data', (data) => {  
  console.log(`stdout: ${data}`);  
});  
  
ls.stderr.on('data', (data) => {  
  console.log(`stderr: ${data}`);  
});  
  
ls.on('close', (code) => {  
  console.log(`child process exited with code ${code}`);  
});
```

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

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

ps.stdout.on('data', (data) => {
  grep.stdin.write(data);
});

ps.stderr.on('data', (data) => {
  console.log(`ps stderr: ${data}`);
});

ps.on('close', (code) => {
  if (code !== 0) {
    console.log(`ps process exited with code ${code}`);
  }
  grep.stdin.end();
});

grep.stdout.on('data', (data) => {
  console.log(`${data}`);
});

grep.stderr.on('data', (data) => {
  console.log(`grep stderr: ${data}`);
});

grep.on('close', (code) => {
  if (code !== 0) {
    console.log(`grep process exited with code ${code}`);
  }
})
```

```
});
```

Example of checking for failed exec:

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

child.on('error', (err) => {
  console.log('Failed to start child process.');
});
```

options.detached

Added in: v0.7.10

On Windows, setting `options.detached` to `true` makes it possible for the child process to continue running after the parent exits. The child will have its own console window. *Once enabled for a child process, it cannot be disabled.*

On non-Windows platforms, if `options.detached` is set to `true`, the child process will be made the leader of a new process group and session. Note that child processes may continue running after the parent exits regardless of whether they are detached or not. See [setsid\(2\)](#) for more information.

By default, the parent will wait for the detached child to exit. To prevent the parent from waiting for a given `child`, use the `child.unref()` method. Doing so will cause the parent's event loop to not include the child in its reference count, allowing the parent to exit independently of the child, unless there is an established IPC channel between the child and parent.

When using the `detached` option to start a long-running process, the process will not stay running in the background after the parent exits unless it is provided with a `stdio` configuration that is not connected to the parent. If the parent's `stdio` is inherited, the child will remain attached to the controlling terminal.

Example of a long-running process, by detaching and also ignoring its parent `stdio` file descriptors, in order to ignore the parent's termination:

```
const spawn = require('child_process').spawn;

const child = spawn(process.argv[0], ['child_program.js'], {
  detached: true,
  stdio: ['ignore']
});

child.unref();
```

Alternatively one can redirect the child process' output into files:

```
const fs = require('fs');
const spawn = require('child_process').spawn;
const out = fs.openSync('./out.log', 'a');
const err = fs.openSync('./out.log', 'a');

const child = spawn('prg', [], {
  detached: true,
  stdio: [ 'ignore', out, err ]
});

child.unref();
```

options.stdio

#

Added in: v0.7.10

The `options.stdio` option is used to configure the pipes that are established between the parent and child process. By default, the child's `stdin`, `stdout`, and `stderr` are redirected to corresponding `child.stdin`, `child.stdout`, and `child.stderr` streams on the `ChildProcess` object. This is equivalent to setting the

`options.stdio` equal to `['pipe', 'pipe', 'pipe']`.

For convenience, `options.stdio` may be one of the following strings:

- `'pipe'` - equivalent to `['pipe', 'pipe', 'pipe']` (the default)
- `'ignore'` - equivalent to `['ignore', 'ignore', 'ignore']`
- `'inherit'` - equivalent to `[process.stdin, process.stdout, process.stderr]` or `[0,1,2]`

Otherwise, the value of `options.stdio` is an array where each index corresponds to an fd in the child. The fds 0, 1, and 2 correspond to stdin, stdout, and stderr, respectively. Additional fds can be specified to create additional pipes between the parent and child. The value is one of the following:

1. `'pipe'` - Create a pipe between the child process and the parent process. The parent end of the pipe is exposed to the parent as a property on the `child_process` object as `child.stdio[fd]`. Pipes created for fds 0 - 2 are also available as `child.stdin`, `child.stdout` and `child.stderr`, respectively.
2. `'ipc'` - Create an IPC channel for passing messages/file descriptors between parent and child. A `ChildProcess` may have at most one IPC stdio file descriptor. Setting this option enables the `child.send()` method. If the child writes JSON messages to this file descriptor, the `child.on('message')` event handler will be triggered in the parent. If the child is a Node.js process, the presence of an IPC channel will enable `process.send()`, `process.disconnect()`, `process.on('disconnect')`, and `process.on('message')` within the child.
3. `'ignore'` - Instructs Node.js to ignore the fd in the child. While Node.js will always open fds 0 - 2 for the processes it spawns, setting the fd to `'ignore'` will cause Node.js to open `/dev/null` and attach it to the child's fd.
4. `<Stream>` object - Share a readable or writable stream that refers to a tty, file, socket, or a pipe with the child process. The stream's underlying file descriptor is duplicated in the child process to the fd that corresponds to the index in the `stdio` array. Note that the stream must have an underlying descriptor (file streams do not until the `'open'` event has occurred).
5. Positive integer - The integer value is interpreted as a file descriptor that is currently open in the parent process. It is shared with the child process, similar

to how `<Stream>` objects can be shared.

6. `null`, `undefined` - Use default value. For stdio fds 0, 1 and 2 (in other words, `stdin`, `stdout`, and `stderr`) a pipe is created. For fd 3 and up, the default is '`ignore`'.

Example:

```
const spawn = require('child_process').spawn;

// Child will use parent's stdios
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs presenting a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

It is worth noting that when an IPC channel is established between the parent and child processes, and the child is a Node.js process, the child is launched with the IPC channel unreferenced (using `unref()`) until the child registers an event handler for the `process.on('disconnect')` event. This allows the child to exit normally without the process being held open by the open IPC channel.

See also: `child_process.exec()` and `child_process.fork()`

Synchronous Process Creation

The `child_process.spawnSync()`, `child_process.execSync()`, and `child_process.execFileSync()` methods are **synchronous** and **WILL** block the Node.js event loop, pausing execution of any additional code until the spawned process exits.

Blocking calls like these are mostly useful for simplifying general purpose scripting

tasks and for simplifying the loading/processing of application configuration at startup.

child_process.execFileSync(file[, args][, options])

#

Added in: v0.11.12

- `file` `<String>` The name or path of the executable file to run
- `args` `<Array>` List of string arguments
- `options` `<Object>`
 - `cwd` `<String>` Current working directory of the child process
 - `input` `<String>` | `<Buffer>` The value which will be passed as stdio to the spawned process
 - supplying this value will override `stdio[0]`
 - `stdio` `<Array>` Child's stdio configuration. (Default: `'pipe'`)
 - `stderr` by default will be output to the parent process' stderr unless `stdio` is specified
 - `env` `<Object>` Environment key-value pairs
 - `uid` `<Number>` Sets the user identity of the process. (See [setuid\(2\)](#).)
 - `gid` `<Number>` Sets the group identity of the process. (See [setgid\(2\)](#).)
 - `timeout` `<Number>` In milliseconds the maximum amount of time the process is allowed to run. (Default: `undefined`)
 - `killSignal` `<String>` The signal value to be used when the spawned process will be killed. (Default: `'SIGTERM'`)
 - `maxBuffer` `<Number>` largest amount of data (in bytes) allowed on stdout or stderr - if exceeded child process is killed
 - `encoding` `<String>` The encoding used for all stdio inputs and outputs. (Default: `'buffer'`)
- return: `<Buffer>` | `<String>` The stdout from the command

The `child_process.execFileSync()` method is generally identical to `child_process.execFile()` with the exception that the method will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely

exited. Note that if the child process intercepts and handles the `SIGTERM` signal and does not exit, the parent process will still wait until the child process has exited.

If the process times out, or has a non-zero exit code, this method **will** throw. The `Error` object will contain the entire result from `child_process.spawnSync()`

`child_process.execSync(command[, options])`

#

Added in: v0.11.12

- `command` `<String>` The command to run
- `options` `<Object>`
 - `cwd` `<String>` Current working directory of the child process
 - `input` `<String>` | `<Buffer>` The value which will be passed as `stdin` to the spawned process
 - supplying this value will override `stdio[0]`
 - `stdio` `<Array>` Child's `stdio` configuration. (Default: `'pipe'`)
 - `stderr` by default will be output to the parent process' `stderr` unless `stdio` is specified
 - `env` `<Object>` Environment key-value pairs
 - `shell` `<String>` Shell to execute the command with (Default: `'/bin/sh'` on UNIX, `'cmd.exe'` on Windows, The shell should understand the `-c` switch on UNIX or `/s /c` on Windows. On Windows, command line parsing should be compatible with `cmd.exe`.)
 - `uid` `<Number>` Sets the user identity of the process. (See `setuid(2)`.)
 - `gid` `<Number>` Sets the group identity of the process. (See `setgid(2)`.)
 - `timeout` `<Number>` In milliseconds the maximum amount of time the process is allowed to run. (Default: `undefined`)
 - `killSignal` `<String>` The signal value to be used when the spawned process will be killed. (Default: `'SIGTERM'`)
 - `maxBuffer` `<Number>` largest amount of data (in bytes) allowed on `stdout` or `stderr` - if exceeded child process is killed
 - `encoding` `<String>` The encoding used for all `stdio` inputs and outputs. (Default: `'buffer'`)

- return: `<Buffer> | <String>` The stdout from the command

The `child_process.execSync()` method is generally identical to `child_process.exec()` with the exception that the method will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited. *Note that if the child process intercepts and handles the `SIGTERM` signal and doesn't exit, the parent process will wait until the child process has exited.*

If the process times out, or has a non-zero exit code, this method **will** throw. The `Error` object will contain the entire result from `child_process.spawnSync()`

`child_process.spawnSync(command[, args][, options])`

#

Added in: v0.11.12

- `command` `<String>` The command to run
- `args` `<Array>` List of string arguments
- `options` `<Object>`
 - `cwd` `<String>` Current working directory of the child process
 - `input` `<String> | <Buffer>` The value which will be passed as stdin to the spawned process
 - supplying this value will override `stdio[0]`
 - `stdio` `<Array>` Child's stdio configuration.
 - `env` `<Object>` Environment key-value pairs
 - `uid` `<Number>` Sets the user identity of the process. (See `setuid(2)`.)
 - `gid` `<Number>` Sets the group identity of the process. (See `setgid(2)`.)
 - `timeout` `<Number>` In milliseconds the maximum amount of time the process is allowed to run. (Default: `undefined`)
 - `killSignal` `<String>` The signal value to be used when the spawned process will be killed. (Default: `'SIGTERM'`)
 - `maxBuffer` `<Number>` largest amount of data (in bytes) allowed on stdout or stderr - if exceeded child process is killed
 - `encoding` `<String>` The encoding used for all stdio inputs and outputs. (Default: `'buffer'`)

- `shell` `<Boolean> | <String>` If `true`, runs `command` inside of a shell. Uses `'/bin/sh'` on UNIX, and `'cmd.exe'` on Windows. A different shell can be specified as a string. The shell should understand the `-c` switch on UNIX, or `/s /c` on Windows. Defaults to `false` (no shell).
- return: `<Object>`
 - `pid` `<Number>` Pid of the child process
 - `output` `<Array>` Array of results from stdio output
 - `stdout` `<Buffer> | <String>` The contents of `output[1]`
 - `stderr` `<Buffer> | <String>` The contents of `output[2]`
 - `status` `<Number>` The exit code of the child process
 - `signal` `<String>` The signal used to kill the child process
 - `error` `<Error>` The error object if the child process failed or timed out

The `child_process.spawnSync()` method is generally identical to `child_process.spawn()` with the exception that the function will not return until the child process has fully closed. When a timeout has been encountered and `killSignal` is sent, the method won't return until the process has completely exited. Note that if the process intercepts and handles the `SIGTERM` signal and doesn't exit, the parent process will wait until the child process has exited.

Class: ChildProcess

Added in: v2.2.0

Instances of the `ChildProcess` class are `EventEmitters` that represent spawned child processes.

Instances of `ChildProcess` are not intended to be created directly. Rather, use the `child_process.spawn()`, `child_process.exec()`, `child_process.execFile()`, or `child_process.fork()` methods to create instances of `ChildProcess`.

Event: 'close'

Added in: v0.7.7

- `code` `<Number>` the exit code if the child exited on its own.

- `signal` <String> the signal by which the child process was terminated.

The '`close`' event is emitted when the stdio streams of a child process have been closed. This is distinct from the '`exit`' event, since multiple processes might share the same stdio streams.

Event: 'disconnect'

#

Added in: v0.7.2

The '`disconnect`' event is emitted after calling the `child.disconnect()` method in parent process or `process.disconnect()` in child process. After disconnecting it is no longer possible to send or receive messages, and the `child.connected` property is `false`.

Event: 'error'

#

- `err` <Error> the error.

The '`error`' event is emitted whenever:

1. The process could not be spawned, or
2. The process could not be killed, or
3. Sending a message to the child process failed.

Note that the '`exit`' event may or may not fire after an error has occurred. If you are listening to both the '`exit`' and '`error`' events, it is important to guard against accidentally invoking handler functions multiple times.

See also `child.kill()` and `child.send()`.

Event: 'exit'

#

Added in: v0.1.90

- `code` <Number> the exit code if the child exited on its own.
- `signal` <String> the signal by which the child process was terminated.

The '`exit`' event is emitted after the child process ends. If the process exited, `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`. One of the two will always be non-null.

Note that when the `'exit'` event is triggered, child process stdio streams might still be open.

Also, note that Node.js establishes signal handlers for `SIGINT` and `SIGTERM` and Node.js processes will not terminate immediately due to receipt of those signals. Rather, Node.js will perform a sequence of cleanup actions and then will re-raise the handled signal.

See [waitpid\(2\)](#).

Event: 'message'

Added in: v0.5.9

- `message` `<Object>` a parsed JSON object or primitive value.
- `sendHandle` `<Handle>` a `net.Socket` or `net.Server` object, or `undefined`.

The `'message'` event is triggered when a child process uses `process.send()` to send messages.

child.connected

Added in: v0.7.2

- `<Boolean>` Set to `false` after `child.disconnect()` is called

The `child.connected` property indicates whether it is still possible to send and receive messages from a child process. When `child.connected` is `false`, it is no longer possible to send or receive messages.

child.disconnect()

Added in: v0.7.2

Closes the IPC channel between parent and child, allowing the child to exit gracefully once there are no other connections keeping it alive. After calling this method the `child.connected` and `process.connected` properties in both the parent and child (respectively) will be set to `false`, and it will be no longer possible to pass messages between the processes.

The 'disconnect' event will be emitted when there are no messages in the process of being received. This will most often be triggered immediately after calling `child.disconnect()`.

Note that when the child process is a Node.js instance (e.g. spawned using `child_process.fork()`), the `process.disconnect()` method can be invoked within the child process to close the IPC channel as well.

child.kill([signal])

#

Added in: v0.1.90

- `signal <String>`

The `child.kill()` methods sends a signal to the child process. If no argument is given, the process will be sent the 'SIGTERM' signal. See `signal(7)` for a list of available signals.

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

grep.on('close', (code, signal) => {
  console.log(`child process terminated due to receipt of signal ${signal}`);
});

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

The `ChildProcess` object may emit an 'error' event if the signal cannot be delivered. Sending a signal to a child process that has already exited is not an error but may have unforeseen consequences. Specifically, if the process identifier (PID) has been reassigned to another process, the signal will be delivered to that process instead which can have unexpected results.

Note that while the function is called `kill`, the signal delivered to the child process

may not actually terminate the process.

See [kill\(2\)](#) for reference.

Also note: on Linux, child processes of child processes will not be terminated when attempting to kill their parent. This is likely to happen when running a new process in a shell or with use of the `shell` option of `ChildProcess`, such as in this example:

```
'use strict';

const spawn = require('child_process').spawn;

let child = spawn('sh', ['-c',
  `node -e "setInterval(() => {
    console.log(process.pid + ' is alive')
  }, 500);`],
  {
    stdio: ['inherit', 'inherit', 'inherit']
});

setTimeout(() => {
  child.kill(); // does not terminate the node process in the shell
}, 2000);
```

child.pid

#

Added in: v0.1.90

- <Number> Integer

Returns the process identifier (PID) of the child process.

Example:

```
const spawn = require('child_process').spawn;
```

```
const grep = spawn('grep', ['ssh']);

console.log(`Spawned child pid: ${grep.pid}`);
grep.stdin.end();
```

child.send(message[, sendHandle[, options]][], callback)

Added in: v0.5.9

- `message` <Object>
- `sendHandle` <Handle>
- `options` <Object>
- `callback` <Function>
- Return: <Boolean>

When an IPC channel has been established between the parent and child (i.e. when using `child_process.fork()`), the `child.send()` method can be used to send messages to the child process. When the child process is a Node.js instance, these messages can be received via the `process.on('message')` event.

For example, in the parent script:

```
const cp = require('child_process');
const n = cp.fork(`$__dirname__/sub.js`);

n.on('message', (m) => {
  console.log('PARENT got message:', m);
});

n.send({ hello: 'world' });
```

And then the child script, `'sub.js'` might look like this:

```
process.on('message', (m) => {
  console.log('CHILD got message:', m);
});

process.send({ foo: 'bar' });
```

Child Node.js processes will have a `process.send()` method of their own that allows the child to send messages back to the parent.

There is a special case when sending a `{cmd: 'NODE_foo'}` message. All messages containing a `NODE_` prefix in its `cmd` property are considered to be reserved for use within Node.js core and will not be emitted in the child's `process.on('message')` event. Rather, such messages are emitted using the `process.on('internalMessage')` event and are consumed internally by Node.js. Applications should avoid using such messages or listening for `'internalMessage'` events as it is subject to change without notice.

The optional `sendHandle` argument that may be passed to `child.send()` is for passing a TCP server or socket object to the child process. The child will receive the object as the second argument passed to the callback function registered on the `process.on('message')` event. Any data that is received and buffered in the socket will not be sent to the child.

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

- `keepOpen` - A Boolean value that can be used when passing instances of `net.Socket`. When `true`, the socket is kept open in the sending process. Defaults to `false`.

The optional `callback` is a function that is invoked after the message is sent but before the child may have received it. The function is called with a single argument: `null` on success, or an `Error` object on failure.

If no `callback` function is provided and the message cannot be sent, an `'error'`

event will be emitted by the `ChildProcess` object. This can happen, for instance, when the child process has already exited.

`child.send()` will return `false` if the channel has closed or when the backlog of unsent messages exceeds a threshold that makes it unwise to send more. Otherwise, the method returns `true`. The `callback` function can be used to implement flow control.

Example: sending a server object

#

The `sendHandle` argument can be used, for instance, to pass the handle of a TCP server object to the child process as illustrated in the example below:

```
const child = require('child_process').fork('child.js');

// Open up the server object and send the handle.
const server = require('net').createServer();
server.on('connection', (socket) => {
  socket.end('handled by parent');
});

server.listen(1337, () => {
  child.send('server', server);
});
```

The child would then receive the server object as:

```
process.on('message', (m, server) => {
  if (m === 'server') {
    server.on('connection', (socket) => {
      socket.end('handled by child');
    });
  }
});
```

Once the server is now shared between the parent and child, some connections can be handled by the parent and some by the child.

While the example above uses a server created using the `net` module, `dgram` module servers use exactly the same workflow with the exceptions of listening on a `'message'` event instead of `'connection'` and using `server.bind()` instead of `server.listen()`. This is, however, currently only supported on UNIX platforms.

Example: sending a socket object

Similarly, the `sendHandler` argument can be used to pass the handle of a socket to the child process. The example below spawns two children that each handle connections with "normal" or "special" priority:

```
const normal = require('child_process').fork('child.js', ['normal']);
const special = require('child_process').fork('child.js', ['special']);

// Open up the server and send sockets to child
const server = require('net').createServer();
server.on('connection', (socket) => {

    // If this is special priority
    if (socket.remoteAddress === '74.125.127.100') {
        special.send('socket', socket);
        return;
    }

    // This is normal priority
    normal.send('socket', socket);
});

server.listen(1337);
```

The `child.js` would receive the socket handle as the second argument passed to the event callback function:

```
process.on('message', (m, socket) => {
  if (m === 'socket') {
    socket.end(`Request handled with ${process.argv[2]} priority`);
  }
});
```

Once a socket has been passed to a child, the parent is no longer capable of tracking when the socket is destroyed. To indicate this, the `.connections` property becomes `null`. It is recommended not to use `.maxConnections` when this occurs.

Note: this function uses `JSON.stringify()` internally to serialize the message.

child.stderr

Added in: v0.1.90

- `<Stream>`

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

If the child was spawned with `stdio[2]` set to anything other than `'pipe'`, then this will be `undefined`.

`child.stderr` is an alias for `child.stdio[2]`. Both properties will refer to the same value.

child.stdin

Added in: v0.1.90

- `<Stream>`

A `Writable Stream` that represents the child process's `stdin`.

Note that if a child process waits to read all of its input, the child will not continue until this stream has been closed via `end()`.

If the child was spawned with `stdio[0]` set to anything other than `'pipe'`, then this will be `undefined`.

`child.stdin` is an alias for `child.stdio[0]`. Both properties will refer to the same value.

child.stdio

#

Added in: v0.7.10

- <Array>

A sparse array of pipes to the child process, corresponding with positions in the `stdio` option passed to `child_process.spawn()` that have been set to the value `'pipe'`. Note that `child.stdio[0]`, `child.stdio[1]`, and `child.stdio[2]` are also available as `child.stdin`, `child.stdout`, and `child.stderr`, respectively.

In the following example, only the child's fd `1` (stdout) is configured as a pipe, so only the parent's `child.stdio[1]` is a stream, all other values in the array are `null`.

```
const assert = require('assert');
const fs = require('fs');
const child_process = require('child_process');

const child = child_process.spawn('ls', {
  stdio: [
    0, // Use parents stdin for child
    'pipe', // Pipe child's stdout to parent
    fs.openSync('err.out', 'w') // Direct child's stderr to a file
  ]
});

assert.equal(child.stdio[0], null);
assert.equal(child.stdio[0], child.stdin);

assert(child.stdout);
assert.equal(child.stdio[1], child.stdout);
```

```
assert.equal(child.stdio[2], null);
assert.equal(child.stdio[2], child.stderr);
```

child.stdout

Added in: v0.1.90

- <Stream>

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

If the child was spawned with `stdio[1]` set to anything other than `'pipe'`, then this will be `undefined`.

`child.stdout` is an alias for `child.stdio[1]`. Both properties will refer to the same value.

maxBuffer and Unicode

It is important to keep in mind that the `maxBuffer` option specifies the largest number of octets allowed on `stdout` or `stderr`. If this value is exceeded, then the child process is terminated. This particularly impacts output that includes multibyte character encodings such as UTF-8 or UTF-16. For instance, the following will output 13 UTF-8 encoded octets to `stdout` although there are only 4 characters:

```
console.log('中文测试');
```

Cluster

Stability: 2 - Stable

A single instance of Node.js runs in a single thread. To take advantage of multi-core

systems the user will sometimes want to launch a cluster of Node.js processes to handle the load.

The cluster module allows you to easily create child processes that all share server ports.

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

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

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

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

```
$ NODE_DEBUG=cluster node server.js
23521,Master Worker 23524 online
23521,Master Worker 23526 online
```

23521,Master Worker 23523 online
23521,Master Worker 23528 online

Please note that, on Windows, it is not yet possible to set up a named pipe server in a worker.

How It Works

The worker processes are spawned using the `child_process.fork()` method, so that they can communicate with the parent via IPC and pass server handles back and forth.

The cluster module supports two methods of distributing incoming connections.

The first one (and the default one on all platforms except Windows), is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers in a round-robin fashion, with some built-in smarts to avoid overloading a worker process.

The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

The second approach should, in theory, give the best performance. In practice however, distribution tends to be very unbalanced due to operating system scheduler vagaries. Loads have been observed where over 70% of all connections ended up in just two processes, out of a total of eight.

Because `server.listen()` hands off most of the work to the master process, there are three cases where the behavior between a normal Node.js process and a cluster worker differs:

1. `server.listen({fd: 7})` Because the message is passed to the master, file descriptor 7 in the parent will be listened on, and the handle passed to the worker, rather than listening to the worker's idea of what the number 7 file descriptor references.

2. `server.listen(handle)` Listening on handles explicitly will cause the worker to use the supplied handle, rather than talk to the master process. If the worker already has the handle, then it's presumed that you know what you are doing.
3. `server.listen(0)` Normally, this will cause servers to listen on a random port. However, in a cluster, each worker will receive the same "random" port each time they do `listen(0)`. In essence, the port is random the first time, but predictable thereafter. If you want to listen on a unique port, generate a port number based on the cluster worker ID.

There is no routing logic in Node.js, or in your program, and no shared state between the workers. Therefore, it is important to design your program such that it does not rely too heavily on in-memory data objects for things like sessions and login.

Because workers are all separate processes, they can be killed or re-spawned depending on your program's needs, without affecting other workers. As long as there are some workers still alive, the server will continue to accept connections. If no workers are alive, existing connections will be dropped and new connections will be refused. Node.js does not automatically manage the number of workers for you, however. It is your responsibility to manage the worker pool for your application's needs.

Class: Worker

#

A Worker object contains all public information and method about a worker. In the master it can be obtained using `cluster.workers`. In a worker it can be obtained using `cluster.worker`.

Event: 'disconnect'

#

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

```
cluster.fork().on('disconnect', () => {
  // Worker has disconnected
});
```

Event: 'error'

#

This event is the same as the one provided by `child_process.fork()`.

In a worker you can also use `process.on('error')`.

Event: 'exit'

#

- `code <Number>` the exit code, if it exited normally.
- `signal <String>` the name of the signal (eg. `'SIGHUP'`) that caused the process to be killed.

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

```
const worker = cluster.fork();
worker.on('exit', (code, signal) => {
  if (signal) {
    console.log(`worker was killed by signal: ${signal}`);
  } else if (code !== 0) {
    console.log(`worker exited with error code: ${code}`);
  } else {
    console.log('worker success!');
  }
});
```

Event: 'listening'

#

- `address <Object>`

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

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

It is not emitted in the worker.

- message <Object>
- handle <undefined> | <Object>

Similar to the `cluster.on('message')` event, but specific to this worker. In a worker you can also use `process.on('message')`.

See [process event: 'message'](#).

As an example, here is a cluster that keeps count of the number of requests in the master process using the message system:

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

if (cluster.isMaster) {

    // Keep track of http requests
    var numReqs = 0;
    setInterval(() => {
        console.log('numReqs =', numReqs);
    }, 1000);

    // Count requests
    function messageHandler(msg) {
        if (msg.cmd && msg.cmd == 'notifyRequest') {
            numReqs += 1;
        }
    }

    // Start workers and listen for messages containing notifyRequest
    const numCPUs = require('os').cpus().length;
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
}
```

```
}

Object.keys(cluster.workers).forEach((id) => {
  cluster.workers[id].on('message', messageHandler);
});

} else {

  // Worker processes have a http server.
  http.Server((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');

    // notify master about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}

}
```

Event: 'online'

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

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

It is not emitted in the worker.

worker.disconnect()

In a worker, this function will close all servers, wait for the `'close'` event on those servers, and then disconnect the IPC channel.

In the master, an internal message is sent to the worker causing it to call

`.disconnect()` on itself.

Causes `.exitedAfterDisconnect` to be set.

Note that after a server is closed, it will no longer accept new connections, but connections may be accepted by any other listening worker. Existing connections will be allowed to close as usual. When no more connections exist, see `server.close()`, the IPC channel to the worker will close allowing it to die gracefully.

The above applies *only* to server connections, client connections are not automatically closed by workers, and disconnect does not wait for them to close before exiting.

Note that in a worker, `process.disconnect` exists, but it is not this function, it is `disconnect`.

Because long living server connections may block workers from disconnecting, it may be useful to send a message, so application specific actions may be taken to close them. It also may be useful to implement a timeout, killing a worker if the '`disconnect`' event has not been emitted after some time.

```
if (cluster.isMaster) {  
  var worker = cluster.fork();  
  var timeout;  
  
  worker.on('listening', (address) => {  
    worker.send('shutdown');  
    worker.disconnect();  
    timeout = setTimeout(() => {  
      worker.kill();  
    }, 2000);  
  });  
  
  worker.on('disconnect', () => {
```

```
    clearTimeout(timeout);
});

} else if (cluster.isWorker) {
  const net = require('net');
  var server = net.createServer((socket) => {
    // connections never end
  });
}

server.listen(8000);

process.on('message', (msg) => {
  if (msg === 'shutdown') {
    // initiate graceful close of any connections to server
  }
});
}
```

worker.exitedAfterDisconnect

- <Boolean>

Set by calling `.kill()` or `.disconnect()`. Until then, it is `undefined`.

The boolean `worker.exitedAfterDisconnect` lets you distinguish between voluntary and accidental exit, the master may choose not to respawn a worker based on this value.

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

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

worker.id

- <Number>

Each new worker is given its own unique id, this id is stored in the `id`.

While a worker is alive, this is the key that indexes it in `cluster.workers`

worker.isConnected()

This function returns `true` if the worker is connected to its master via its IPC channel, `false` otherwise. A worker is connected to its master after it's been created. It is disconnected after the `'disconnect'` event is emitted.

worker.isDead()

This function returns `true` if the worker's process has terminated (either because of exiting or being signaled). Otherwise, it returns `false`.

worker.kill([signal='SIGTERM'])

- `signal` <String> Name of the kill signal to send to the worker process.

This function will kill the worker. In the master, it does this by disconnecting the `worker.process`, and once disconnected, killing with `signal`. In the worker, it does it by disconnecting the channel, and then exiting with code `0`.

Causes `.exitedAfterDisconnect` to be set.

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

Note that in a worker, `process.kill()` exists, but it is not this function, it is `kill`.

worker.process

- <ChildProcess>

All workers are created using `child_process.fork()`, the returned object from this function is stored as `.process`. In a worker, the global `process` is stored.

See: [Child Process module](#)

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

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

- `message <Object>`
- `sendHandle <Handle>`
- `callback <Function>`
- Return: Boolean

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

In the master this sends a message to a specific worker. It is identical to [ChildProcess.send\(\)](#).

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

This example will echo back all messages from the master:

```
if (cluster.isMaster) {  
    var worker = cluster.fork();  
    worker.send('hi there');  
  
} else if (cluster.isWorker) {  
    process.on('message', (msg) => {  
        process.send(msg);  
    });  
}
```

Stability: 0 - Deprecated: Use [worker.exitedAfterDisconnect](#) instead.

An alias to `worker.exitedAfterDisconnect`.

Set by calling `.kill()` or `.disconnect()`. Until then, it is `undefined`.

The boolean `worker.suicide` lets you distinguish between voluntary and accidental exit, the master may choose not to respawn a worker based on this value.

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

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

This API only exists for backwards compatibility and will be removed in the future.

Event: 'disconnect'

#

- `worker <cluster.Worker>`

Emitted after the worker IPC channel has disconnected. This can occur when a worker exits gracefully, is killed, or is disconnected manually (such as with `worker.disconnect()`).

There may be a delay between the `'disconnect'` and `'exit'` events. These events can be used to detect if the process is stuck in a cleanup or if there are long-living connections.

```
cluster.on('disconnect', (worker) => {
  console.log(`The worker ${worker.id} has disconnected`);
});
```

Event: 'exit'

#

- `worker <cluster.Worker>`
- `code <Number>` the exit code, if it exited normally.
- `signal <String>` the name of the signal (eg. `'SIGHUP'`) that caused the process to be killed.

When any of the workers die the cluster module will emit the `'exit'` event.

This can be used to restart the worker by calling `.fork()` again.

```
cluster.on('exit', (worker, code, signal) => {
  console.log(`worker ${worker.pid} died (${signal}). restarting...`,
    worker.process.pid, signal || code);
  cluster.fork();
});
```

See `child_process` event: `'exit'`.

Event: 'fork'

#

- `worker <cluster.Worker>`

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

```
var timeouts = [];
function errorMsg() {
  console.error('Something must be wrong with the connection ...');
```

```
cluster.on('fork', (worker) => {
  timeouts[worker.id] = setTimeout(errorMsg, 2000);
});

cluster.on('listening', (worker, address) => {
  clearTimeout(timeouts[worker.id]);
});

cluster.on('exit', (worker, code, signal) => {
  clearTimeout(timeouts[worker.id]);
  errorMsg();
});
```

Event: 'listening'

#

- `worker <cluster.Worker>`
- `address <Object>`

After calling `listen()` from a worker, when the `'listening'` event is emitted on the server, a `'listening'` event will also be emitted on `cluster` in the master.

The event handler is executed with two arguments, the `worker` contains the worker object and the `address` object contains the following connection properties: `address`, `port` and `addressType`. This is very useful if the worker is listening on more than one address.

```
cluster.on('listening', (worker, address) => {
  console.log(
    `A worker is now connected to ${address.address}:${address.port}`);
});
```

The `addressType` is one of:

- 4 (TCPv4)

- 6 (TCPv6)
- -1 (unix domain socket)
- "udp4" or "udp6" (UDP v4 or v6)

Event: 'message'

#

- worker <cluster.Worker>
- message <Object>
- handle <undefined> | <Object>

Emitted when any worker receives a message.

See [child_process event: 'message'](#).

Before Node.js v6.0, this event emitted only the message and the handle, but not the worker object, contrary to what the documentation stated.

If you need to support older versions and don't need the worker object, you can work around the discrepancy by checking the number of arguments:

```
cluster.on('message', function(worker, message, handle) {
  if (arguments.length === 2) {
    handle = message;
    message = worker;
    worker = undefined;
  }
  // ...
});
```

Event: 'online'

#

- worker <cluster.Worker>

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

between 'fork' and 'online' is that fork is emitted when the master forks a worker, and 'online' is emitted when the worker is running.

```
cluster.on('online', (worker) => {
  console.log('Yay, the worker responded after it was forked');
});
```

Event: 'setup'

#

- `settings <Object>`

Emitted every time `.setupMaster()` is called.

The `settings` object is the `cluster.settings` object at the time `.setupMaster()` was called and is advisory only, since multiple calls to `.setupMaster()` can be made in a single tick.

If accuracy is important, use `cluster.settings`.

cluster.disconnect([callback])

#

- `callback <Function>` called when all workers are disconnected and handles are closed

Calls `.disconnect()` on each worker in `cluster.workers`.

When they are disconnected all internal handles will be closed, allowing the master process to die gracefully if no other event is waiting.

The method takes an optional callback argument which will be called when finished.

This can only be called from the master process.

cluster.fork([env])

#

- `env <Object>` Key/value pairs to add to worker process environment.

- return `<cluster.Worker>`

Spawn a new worker process.

This can only be called from the master process.

cluster.isMaster

#

- `<Boolean>`

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

cluster.isWorker

#

- `<Boolean>`

True if the process is not a master (it is the negation of `cluster.isMaster`).

cluster.schedulingPolicy

#

The scheduling policy, either `cluster.SCHED_RR` for round-robin or `cluster.SCHED_NONE` to leave it to the operating system. This is a global setting and effectively frozen once you spawn the first worker or call `cluster.setupMaster()`, whatever comes first.

`SCHED_RR` is the default on all operating systems except Windows. Windows will change to `SCHED_RR` once libuv is able to effectively distribute IOCP handles without incurring a large performance hit.

`cluster.schedulingPolicy` can also be set through the `NODE_CLUSTER_SCHED_POLICY` environment variable. Valid values are `"rr"` and `"none"`.

cluster.settings

#

- `<Object>`

- `execArgv` `<Array>` list of string arguments passed to the Node.js executable. (Default= `process.execArgv`)
- `exec` `<String>` file path to worker file. (Default= `process.argv[1]`)
- `args` `<Array>` string arguments passed to worker. (Default= `process.argv.slice(2)`)
- `silent` `<Boolean>` whether or not to send output to parent's stdio. (Default= `false`)
- `uid` `<Number>` Sets the user identity of the process. (See `setuid(2)`.)
- `gid` `<Number>` Sets the group identity of the process. (See `setgid(2)`.)

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

This object is not supposed to be changed or set manually, by you.

cluster.setupMaster([settings])

#

- `settings` `<Object>`
 - `exec` `<String>` file path to worker file. (Default= `process.argv[1]`)
 - `args` `<Array>` string arguments passed to worker. (Default= `process.argv.slice(2)`)
 - `silent` `<Boolean>` whether or not to send output to parent's stdio. (Default= `false`)

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

Note that:

- any settings changes only affect future calls to `.fork()` and have no effect on workers that are already running
- The *only* attribute of a worker that cannot be set via `.setupMaster()` is the `env` passed to `.fork()`
- the defaults above apply to the first call only, the defaults for later calls is the current value at the time of `cluster.setupMaster()` is called

Example:

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

This can only be called from the master process.

cluster.worker

#

- <Object>

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

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

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

cluster.workers

#

- <Object>

A hash that stores the active worker objects, keyed by `id` field. Makes it easy to loop through all the workers. It is only available in the master process.

A worker is removed from `cluster.workers` after the worker has disconnected and exited. The order between these two events cannot be determined in advance. However, it is guaranteed that the removal from the `cluster.workers` list happens before last `'disconnect'` or `'exit'` event is emitted.

```
// Go through all workers
function eachWorker(callback) {
  for (var id in cluster.workers) {
    callback(cluster.workers[id]);
  }
}
eachWorker((worker) => {
  worker.send('big announcement to all workers');
});
```

Should you wish to reference a worker over a communication channel, using the worker's unique id is the easiest way to find the worker.

```
socket.on('data', (id) => {
  var worker = cluster.workers[id];
});
```

Command Line Options

#

Node.js comes with a variety of CLI options. These options expose built-in debugging, multiple ways to execute scripts, and other helpful runtime options.

To view this documentation as a manual page in your terminal, run `man node`.

Synopsis

#

```
node [options] [v8 options] [script.js | -e "script"] [arguments]
```

```
node debug [script.js | -e "script" | <host>:<port>] ...
```

```
node --v8-options
```

Execute without arguments to start the [REPL](#).

For more info about `node debug`, please see the [debugger](#) documentation.

Options

#

-v, --version

#

Added in: v0.1.3

Print node's version.

-h, --help

#

Added in: v0.1.3

Print node command line options. The output of this option is less detailed than this document.

-e, --eval "script"

#

Added in: v0.5.2

Evaluate the following argument as JavaScript. The modules which are predefined in the REPL can also be used in `script`.

-p, --print "script"

#

Added in: v0.6.4

Identical to `-e` but prints the result.

-c, --check #

Added in: v5.0.0

Syntax check the script without executing.

-i, --interactive #

Added in: v0.7.7

Opens the REPL even if stdin does not appear to be a terminal.

-r, --require module #

Added in: v1.6.0

Preload the specified module at startup.

Follows `require()`'s module resolution rules. `module` may be either a path to a file, or a node module name.

--no-deprecation #

Added in: v0.8.0

Silence deprecation warnings.

--trace-deprecation #

Added in: v0.8.0

Print stack traces for deprecations.

--throw-deprecation #

Added in: v0.11.14

Throw errors for deprecations.

--no-warnings #

Added in: v6.0.0

Silence all process warnings (including deprecations).

--trace-warnings

Added in: v6.0.0

Print stack traces for process warnings (including deprecations).

--trace-sync-io

Added in: v2.1.0

Prints a stack trace whenever synchronous I/O is detected after the first turn of the event loop.

--zero-fill-buffers

Added in: v6.0.0

Automatically zero-fills all newly allocated `Buffer` and `SlowBuffer` instances.

--preserve-symlinks

Added in: v6.3.0

Instructs the module loader to preserve symbolic links when resolving and caching modules.

By default, when Node.js loads a module from a path that is symbolically linked to a different on-disk location, Node.js will dereference the link and use the actual on-disk "real path" of the module as both an identifier and as a root path to locate other dependency modules. In most cases, this default behavior is acceptable. However, when using symbolically linked peer dependencies, as illustrated in the example below, the default behavior causes an exception to be thrown if `moduleA` attempts to require `moduleB` as a peer dependency:

```
{appDir}
  └── app
    ├── index.js
    └── node_modules
      └── moduleA → {appDir}/moduleA
          └── moduleB
```

```
|   └── index.js  
|       └── package.json  
└── moduleA  
    ├── index.js  
    └── package.json
```

The `--preserve-symlinks` command line flag instructs Node.js to use the symlink path for modules as opposed to the real path, allowing symbolically linked peer dependencies to be found.

Note, however, that using `--preserve-symlinks` can have other side effects. Specifically, symbolically linked *native* modules can fail to load if those are linked from more than one location in the dependency tree (Node.js would see those as two separate modules and would attempt to load the module multiple times, causing an exception to be thrown).

--track-heap-objects

#

Added in: v2.4.0

Track heap object allocations for heap snapshots.

--prof-process

#

Added in: v6.0.0

Process v8 profiler output generated using the v8 option `--prof`.

--v8-options

#

Added in: v0.1.3

Print v8 command line options.

Note: v8 options allow words to be separated by both dashes (`-`) or underscores (`_`).

For example, `--stack-trace-limit` is equivalent to `--stack_trace_limit`.

--tls-cipher-list=list #

Added in: v4.0.0

Specify an alternative default TLS cipher list. (Requires Node.js to be built with crypto support. (Default))

--enable-fips #

Added in: v6.0.0

Enable FIPS-compliant crypto at startup. (Requires Node.js to be built with

./configure --openssl-fips)

--force-fips #

Added in: v6.0.0

Force FIPS-compliant crypto on startup. (Cannot be disabled from script code.)

(Same requirements as --enable-fips)

--icu-data-dir=file #

Added in: v0.11.15

Specify ICU data load path. (overrides NODE_ICU_DATA)

Environment Variables

NODE_DEBUG=module[,...] #

Added in: v0.1.32

' , ' -separated list of core modules that should print debug information.

NODE_PATH=path[:...] #

Added in: v0.1.32

' : ' -separated list of directories prefixed to the module search path.

Note: on Windows, this is a ' ; ' -separated list instead.

NODE_DISABLE_COLORS=1

#

Added in: v0.3.0

When set to `1` colors will not be used in the REPL.

NODE_ICU_DATA=file

#

Added in: v0.11.15

Data path for ICU (Intl object) data. Will extend linked-in data when compiled with small-icu support.

NODE_REPL_HISTORY=file

#

Added in: v5.0.0

Path to the file used to store the persistent REPL history. The default path is `~/.node_repl_history`, which is overridden by this variable. Setting the value to an empty string (`""` or `" "`) disables persistent REPL history.

Console

#

Stability: 2 - Stable

The `console` module provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

The module exports two specific components:

- A `Console` class with methods such as `console.log()`, `console.error()` and `console.warn()` that can be used to write to any Node.js stream.
- A global `console` instance configured to write to `stdout` and `stderr`. Because this object is global, it can be used without calling `require('console')`.

Example using the global `console`:

```
console.log('hello world');
  // Prints: hello world, to stdout
console.log('hello %s', 'world');
  // Prints: hello world, to stdout
console.error(new Error('Whoops, something bad happened'));
  // Prints: [Error: Whoops, something bad happened], to stderr

const name = 'Will Robinson';
console.warn(`Danger ${name}! Danger!`);
  // Prints: Danger Will Robinson! Danger!, to stderr
```

Example using the `Console` class:

```
const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new console.Console(out, err);

myConsole.log('hello world');
  // Prints: hello world, to out
myConsole.log('hello %s', 'world');
  // Prints: hello world, to out
myConsole.error(new Error('Whoops, something bad happened'));
  // Prints: [Error: Whoops, something bad happened], to err

const name = 'Will Robinson';
myConsole.warn(`Danger ${name}! Danger!`);
  // Prints: Danger Will Robinson! Danger!, to err
```

While the API for the `Console` class is designed fundamentally around the browser `console` object, the `Console` in Node.js is not intended to duplicate the browser's functionality exactly.

Asynchronous vs Synchronous Consoles

#

The console functions are usually asynchronous unless the destination is a file. Disks are fast and operating systems normally employ write-back caching; it should be a very rare occurrence indeed that a write blocks, but it is possible.

Additionally, console functions are blocking when outputting to TTYs (terminals) on OS X as a workaround for the OS's very small, 1kb buffer size. This is to prevent interleaving between `stdout` and `stderr`.

Class: Console

#

The `Console` class can be used to create a simple logger with configurable output streams and can be accessed using either `require('console').Console` or `console.Console`:

```
const Console = require('console').Console;
const Console = console.Console;
```

`new Console(stdout[, stderr])`

#

Creates a new `Console` by passing one or two writable stream instances. `stdout` is a writable stream to print log or info output. `stderr` is used for warning or error output. If `stderr` isn't passed, warning and error output will be sent to `stdout`.

```
const output = fs.createWriteStream('./stdout.log');
const errorOutput = fs.createWriteStream('./stderr.log');
// custom simple logger
const logger = new Console(output, errorOutput);
// use it like console
var count = 5;
logger.log('count: %d', count);
// in stdout.log: count 5
```

The global `console` is a special `Console` whose output is sent to `process.stdout` and `process.stderr`. It is equivalent to calling:

```
new Console(process.stdout, process.stderr);
```

console.assert(value[, message][, ...])

Added in: v0.1.101

A simple assertion test that verifies whether `value` is truthy. If it is not, an `AssertionError` is thrown. If provided, the error `message` is formatted using `util.format()` and used as the error message.

```
console.assert(true, 'does nothing');
// OK

console.assert(false, 'Whoops %s', 'didn\'t work');
// AssertionError: Whoops didn't work
```

Note: the `console.assert()` method is implemented differently in Node.js than the `console.assert()` method available in browsers.

Specifically, in browsers, calling `console.assert()` with a falsy assertion will cause the `message` to be printed to the console without interrupting execution of subsequent code. In Node.js, however, a falsy assertion will cause an `AssertionError` to be thrown.

Functionality approximating that implemented by browsers can be implemented by extending Node.js' `console` and overriding the `console.assert()` method.

In the following example, a simple module is created that extends and overrides the default behavior of `console` in Node.js.

```
'use strict';
```

```
// Creates a simple extension of console with a
// new impl for assert without monkey-patching.
const myConsole = Object.setPrototypeOf({
  assert(assertion, message, ...args) {
    try {
      console.assert(assertion, message, ...args);
    } catch (err) {
      console.error(err.stack);
    }
  }
}, console);

module.exports = myConsole;
```

This can then be used as a direct replacement for the built in console:

```
const console = require('./myConsole');
console.assert(false, 'this message will print, but no error thrown');
console.log('this will also print');
```

console.dir(obj[, options])

#

Added in: v0.1.101

Uses `util.inspect()` on `obj` and prints the resulting string to `stdout`. This function bypasses any custom `inspect()` function defined on `obj`. An optional `options` object may be passed to alter certain aspects of the formatted string:

- `showHidden` - if `true` then the object's non-enumerable and symbol properties will be shown too. Defaults to `false`.
- `depth` - tells `util.inspect()` how many times to recurse while formatting the object. This is useful for inspecting large complicated objects. Defaults to `2`. To make it recurse indefinitely, pass `null`.

- `colors` - if `true`, then the output will be styled with ANSI color codes. Defaults to `false`. Colors are customizable; see [customizing util.inspect\(\) colors](#).

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

Added in: v0.1.100

Prints to `stderr` with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to `printf(3)` (the arguments are all passed to [util.format\(\)](#)).

```
const code = 5;
console.error('error #%d', code);
// Prints: error #5, to stderr
console.error('error', code);
// Prints: error 5, to stderr
```

If formatting elements (e.g. `%d`) are not found in the first string then [util.inspect\(\)](#) is called on each argument and the resulting string values are concatenated. See [util.format\(\)](#) for more information.

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

Added in: v0.1.100

The `console.info()` function is an alias for [console.log\(\)](#).

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

Added in: v0.1.100

Prints to `stdout` with newline. Multiple arguments can be passed, with the first used as the primary message and all additional used as substitution values similar to `printf(3)` (the arguments are all passed to [util.format\(\)](#)).

```
var count = 5;
console.log('count: %d', count);
```

```
// Prints: count: 5, to stdout
console.log('count: ', count);
// Prints: count: 5, to stdout
```

If formatting elements (e.g. `%d`) are not found in the first string then `util.inspect()` is called on each argument and the resulting string values are concatenated. See `util.format()` for more information.

console.time(label)

#

Added in: v0.1.104

Starts a timer that can be used to compute the duration of an operation. Timers are identified by a unique `label`. Use the same `label` when you call `console.timeEnd()` to stop the timer and output the elapsed time in milliseconds to `stdout`. Timer durations are accurate to the sub-millisecond.

console.timeEnd(label)

#

Added in: v0.1.104

Stops a timer that was previously started by calling `console.time()` and prints the result to `stdout`:

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
    ;
}
console.timeEnd('100-elements');
// prints 100-elements: 225.438ms
```

Note: As of Node.js v6.0.0, `console.timeEnd()` deletes the timer to avoid leaking it. On older versions, the timer persisted. This allowed `console.timeEnd()` to be called multiple times for the same label. This functionality was unintended and is no longer supported.

console.trace(message[, ...])

#

Added in: v0.1.104

Prints to `stderr` the string `'Trace : '`, followed by the `util.format()` formatted message and stack trace to the current position in the code.

```
console.trace('Show me');

// Prints: (stack trace will vary based on where trace is called)
// Trace: Show me
//     at repl:2:9
//     at REPLServer.defaultEval (repl.js:248:27)
//     at bound (domain.js:287:14)
//     at REPLServer.runBound [as eval] (domain.js:300:12)
//     at REPLServer.<anonymous> (repl.js:412:12)
//     at emitOne (events.js:82:20)
//     at REPLServer.emit (events.js:169:7)
//     at REPLServer.Interface._onLine (readline.js:210:10)
//     at REPLServer.Interface._line (readline.js:549:8)
//     at REPLServer.Interface._ttyWrite (readline.js:826:14)
```

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

#

Added in: v0.1.100

The `console.warn()` function is an alias for `console.error()`.

Crypto

#

Stability: 2 - Stable

The `crypto` module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions.

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

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

const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
    .update('I love cupcakes')
    .digest('hex');

console.log(hash);
// Prints:
//   c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea6ea959d6658e
```

Determining if crypto support is unavailable

It is possible for Node.js to be built without including support for the `crypto` module. In such cases, calling `require('crypto')` will result in an error being thrown.

```
var crypto;
try {
  crypto = require('crypto');
} catch (err) {
  console.log('crypto support is disabled!');
}
```

Class: Certificate

SPKAC is a Certificate Signing Request mechanism originally implemented by Netscape and now specified formally as part of [HTML5's keygen element](#).

The `crypto` module provides the `Certificate` class for working with SPKAC data. The most common usage is handling output generated by the [HTML5 <keygen>](#) element. Node.js uses [OpenSSL's SPKAC implementation](#) internally.

`new crypto.Certificate()`

Instances of the `Certificate` class can be created using the `new` keyword or by calling `crypto.Certificate()` as a function:

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

const cert1 = new crypto.Certificate();
const cert2 = crypto.Certificate();
```

`certificate.exportChallenge(spkac)`

The `spkac` data structure includes a public key and a challenge. The `certificate.exportChallenge()` returns the challenge component in the form of a Node.js `Buffer`. The `spkac` argument can be either a string or a `Buffer`.

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const challenge = cert.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints the challenge as a UTF8 string
```

`certificate.exportPublicKey(spkac)`

The `spkac` data structure includes a public key and a challenge. The `certificate.exportPublicKey()` returns the public key component in the form of a Node.js `Buffer`. The `spkac` argument can be either a string or a `Buffer`.

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const publicKey = cert.exportPublicKey(spkac);
console.log(publicKey);
// Prints the public key as <Buffer ...>
```

certificate.verifySpkac(spkac)

#

Returns `true` if the given `spkac` data structure is valid, `false` otherwise. The `spkac` argument must be a Node.js `Buffer`.

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(Buffer.from(spkac)));
// Prints true or false
```

Class: Cipher

#

Instances of the `Cipher` class are used to encrypt data. The class can be used in one of two ways:

- As a `stream` that is both readable and writable, where plain unencrypted data is written to produce encrypted data on the readable side, or
- Using the `cipher.update()` and `cipher.final()` methods to produce the encrypted data.

The `crypto.createCipher()` or `crypto.createCipheriv()` methods are used to create `Cipher` instances. `Cipher` objects are not to be created directly using the `new` keyword.

Example: Using `Cipher` objects as streams:

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

var encrypted = '';
cipher.on('readable', () => {
  var data = cipher.read();
  if (data)
```

```
        encrypted += data.toString('hex');

    });

cipher.on('end', () => {
    console.log(encrypted);
    // Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d8
});

cipher.write('some clear text data');
cipher.end();
```

Example: Using `Cipher` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const cipher = crypto.createCipher('aes192', 'a password');

const input = fs.createReadStream('test.js');
const output = fs.createWriteStream('test.enc');

input.pipe(cipher).pipe(output);
```

Example: Using the `cipher.update()` and `cipher.final()` methods:

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

var encrypted = cipher.update('some clear text data', 'utf8', 'hex');
encrypted += cipher.final('hex');
console.log(encrypted);
// Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d8
```

```
cipher.final([output_encoding]) #  
Returns any remaining enciphered contents. If output_encoding parameter is one  
of 'binary', 'base64' or 'hex', a string is returned. If an output_encoding is  
not provided, a Buffer is returned.
```

Once the `cipher.final()` method has been called, the `Cipher` object can no longer be used to encrypt data. Attempts to call `cipher.final()` more than once will result in an error being thrown.

```
cipher.setAAD(buffer) #
```

When using an authenticated encryption mode (only `GCM` is currently supported), the `cipher.setAAD()` method sets the value used for the *additional authenticated data* (AAD) input parameter.

```
cipher.getAuthTag() #
```

When using an authenticated encryption mode (only `GCM` is currently supported), the `cipher.getAuthTag()` method returns a `Buffer` containing the *authentication tag* that has been computed from the given data.

The `cipher.getAuthTag()` method should only be called after encryption has been completed using the `cipher.final()` method.

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

When using block encryption algorithms, the `Cipher` class will automatically add padding to the input data to the appropriate block size. To disable the default padding call `cipher.setAutoPadding(false)`.

When `auto_padding` is `false`, the length of the entire input data must be a multiple of the cipher's block size or `cipher.final()` will throw an Error. Disabling automatic padding is useful for non-standard padding, for instance using `0x0` instead of PKCS padding.

The `cipher.setAutoPadding()` method must be called before `cipher.final()`.

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

Updates the cipher with `data`. If the `input_encoding` argument is given, its value must be one of `'utf8'`, `'ascii'`, or `'binary'` and the `data` argument is a string using the specified encoding. If the `input_encoding` argument is not given, `data` must be a `Buffer`. If `data` is a `Buffer` then `input_encoding` is ignored.

The `output_encoding` specifies the output format of the enciphered data, and can be `'binary'`, `'base64'` or `'hex'`. If the `output_encoding` is specified, a string using the specified encoding is returned. If no `output_encoding` is provided, a `Buffer` is returned.

The `cipher.update()` method can be called multiple times with new data until `cipher.final()` is called. Calling `cipher.update()` after `cipher.final()` will result in an error being thrown.

Class: Decipher

Instances of the `Decipher` class are used to decrypt data. The class can be used in one of two ways:

- As a `stream` that is both readable and writable, where plain encrypted data is written to produce unencrypted data on the readable side, or
- Using the `decipher.update()` and `decipher.final()` methods to produce the unencrypted data.

The `crypto.createDecipher()` or `crypto.createDecipheriv()` methods are used to create `Decipher` instances. `Decipher` objects are not to be created directly using the `new` keyword.

Example: Using `Decipher` objects as streams:

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

var decrypted = '';
decipher.on('readable', () => {
```

```
var data = decipher.read();
if (data)
  decrypted += data.toString('utf8');
};

decipher.on('end', () => {
  console.log(decrypted);
  // Prints: some clear text data
});

var encrypted = 'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed';
decipher.write(encrypted, 'hex');
decipher.end();
```

Example: Using `Decipher` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const decipher = crypto.createDecipher('aes192', 'a password');

const input = fs.createReadStream('test.enc');
const output = fs.createWriteStream('test.js');

input.pipe(decipher).pipe(output);
```

Example: Using the `decipher.update()` and `decipher.final()` methods:

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

var encrypted = 'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed';
var decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
```

```
console.log(decrypted);
// Prints: some clear text data
```

decipher.final([output_encoding])

Returns any remaining deciphered contents. If `output_encoding` parameter is one of `'binary'`, `'base64'` or `'hex'`, a string is returned. If an `output_encoding` is not provided, a `Buffer` is returned.

Once the `decipher.final()` method has been called, the `Decipher` object can no longer be used to decrypt data. Attempts to call `decipher.final()` more than once will result in an error being thrown.

decipher.setAAD(buffer)

When using an authenticated encryption mode (only `GCM` is currently supported), the `cipher.setAAD()` method sets the value used for the *additional authenticated data* (AAD) input parameter.

decipher.setAuthTag(buffer)

When using an authenticated encryption mode (only `GCM` is currently supported), the `decipher.setAuthTag()` method is used to pass in the received *authentication tag*. If no tag is provided, or if the cipher text has been tampered with, `decipher.final()` will throw, indicating that the cipher text should be discarded due to failed authentication.

decipher.setAutoPadding(auto_padding=true)

When data has been encrypted without standard block padding, calling `decipher.setAutoPadding(false)` will disable automatic padding to prevent `decipher.final()` from checking for and removing padding.

Turning auto padding off will only work if the input data's length is a multiple of the ciphers block size.

The `decipher.setAutoPadding()` method must be called before

`decipher.update()`.

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

Updates the decipher with `data`. If the `input_encoding` argument is given, its value must be one of `'binary'`, `'base64'`, or `'hex'` and the `data` argument is a string using the specified encoding. If the `input_encoding` argument is not given, `data` must be a `Buffer`. If `data` is a `Buffer` then `input_encoding` is ignored.

The `output_encoding` specifies the output format of the enciphered data, and can be `'binary'`, `'ascii'` or `'utf8'`. If the `output_encoding` is specified, a string using the specified encoding is returned. If no `output_encoding` is provided, a `Buffer` is returned.

The `decipher.update()` method can be called multiple times with new data until `decipher.final()` is called. Calling `decipher.update()` after `decipher.final()` will result in an error being thrown.

Class: DiffieHellman

The `DiffieHellman` class is a utility for creating Diffie-Hellman key exchanges.

Instances of the `DiffieHellman` class can be created using the `crypto.createDiffieHellman()` function.

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createDiffieHellman(2048);
const alice_key = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createDiffieHellman(alice.getPrime(), alice.getGenera
const bob_key = bob.generateKeys();
```

```
// Exchange and generate the secret...
const alice_secret = alice.computeSecret(bob_key);
const bob_secret = bob.computeSecret(alice_key);

// OK
assert.equal(alice_secret.toString('hex'), bob_secret.toString('hex'));
```

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

Computes the shared secret using `other_public_key` as the other party's public key and returns the computed shared secret. The supplied key is interpreted using the specified `input_encoding`, and secret is encoded using specified `output_encoding`. Encodings can be `'binary'`, `'hex'`, or `'base64'`. If the `input_encoding` is not provided, `other_public_key` is expected to be a `Buffer`.

If `output_encoding` is given a string is returned; otherwise, a `Buffer` is returned.

diffieHellman.generateKeys([encoding])

Generates private and public Diffie-Hellman key values, and returns the public key in the specified `encoding`. This key should be transferred to the other party. Encoding can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

diffieHellman.getGenerator([encoding])

Returns the Diffie-Hellman generator in the specified `encoding`, which can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

diffieHellman.getPrime([encoding])

Returns the Diffie-Hellman prime in the specified `encoding`, which can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned;

otherwise a **Buffer** is returned.

diffieHellman.getPrivateKey([encoding])

#

Returns the Diffie-Hellman private key in the specified `encoding`, which can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a **Buffer** is returned.

diffieHellman.getPublicKey([encoding])

#

Returns the Diffie-Hellman public key in the specified `encoding`, which can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a **Buffer** is returned.

diffieHellman.setPrivateKey(private_key[, encoding])

#

Sets the Diffie-Hellman private key. If the `encoding` argument is provided and is either `'binary'`, `'hex'`, or `'base64'`, `private_key` is expected to be a string. If no `encoding` is provided, `private_key` is expected to be a **Buffer**.

diffieHellman.setPublicKey(public_key[, encoding])

#

Sets the Diffie-Hellman public key. If the `encoding` argument is provided and is either `'binary'`, `'hex'` or `'base64'`, `public_key` is expected to be a string. If no `encoding` is provided, `public_key` is expected to be a **Buffer**.

diffieHellman.verifyError

#

A bit field containing any warnings and/or errors resulting from a check performed during initialization of the `DiffieHellman` object.

The following values are valid for this property (as defined in `constants` module):

- `DH_CHECK_P_NOT_SAFE_PRIME`
- `DH_CHECK_P_NOT_PRIME`
- `DH_UNABLE_TO_CHECK_GENERATOR`
- `DH_NOT_SUITABLE_GENERATOR`

Class: ECDH

The `ECDH` class is a utility for creating Elliptic Curve Diffie-Hellman (ECDH) key exchanges.

Instances of the `ECDH` class can be created using the `crypto.createECDH()` function.

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createECDH('secp521r1');
const alice_key = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createECDH('secp521r1');
const bob_key = bob.generateKeys();

// Exchange and generate the secret...
const alice_secret = alice.computeSecret(bob_key);
const bob_secret = bob.computeSecret(alice_key);

assert(alice_secret, bob_secret);
// OK
```

`ecdh.computeSecret(other_public_key[, input_encoding][, output_encoding])` #

Computes the shared secret using `other_public_key` as the other party's public key and returns the computed shared secret. The supplied key is interpreted using specified `input_encoding`, and the returned secret is encoded using the specified `output_encoding`. Encodings can be `'binary'`, `'hex'`, or `'base64'`. If the `input_encoding` is not provided, `other_public_key` is expected to be a `Buffer`.

If `output_encoding` is given a string will be returned; otherwise a `Buffer` is returned.

`ecdh.generateKeys([encoding[, format]])`

Generates private and public EC Diffie-Hellman key values, and returns the public key in the specified `format` and `encoding`. This key should be transferred to the other party.

The `format` argument specifies point encoding and can be `'compressed'`, `'uncompressed'`, or `'hybrid'`. If `format` is not specified, the point will be returned in `'uncompressed'` format.

The `encoding` argument can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`ecdh.getPrivateKey([encoding])`

Returns the EC Diffie-Hellman private key in the specified `encoding`, which can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned.

`ecdh.getPublicKey([encoding[, format]])`

Returns the EC Diffie-Hellman public key in the specified `encoding` and `format`.

The `format` argument specifies point encoding and can be `'compressed'`, `'uncompressed'`, or `'hybrid'`. If `format` is not specified the point will be returned in `'uncompressed'` format.

The `encoding` argument can be `'binary'`, `'hex'`, or `'base64'`. If `encoding` is specified, a string is returned; otherwise a `Buffer` is returned.

`ecdh.setPrivateKey(private_key[, encoding])`

Sets the EC Diffie-Hellman private key. The `encoding` can be `'binary'`, `'hex'` or `'base64'`. If `encoding` is provided, `private_key` is expected to be a string; otherwise `private_key` is expected to be a `Buffer`. If `private_key` is not valid

for the curve specified when the `ECDH` object was created, an error is thrown. Upon setting the private key, the associated public point (key) is also generated and set in the `ECDH` object.

`ecdh.setPublicKey(public_key[, encoding])`

#

Stability: 0 - Deprecated

Sets the EC Diffie-Hellman public key. Key encoding can be `'binary'`, `'hex'` or `'base64'`. If `encoding` is provided `public_key` is expected to be a string; otherwise a `Buffer` is expected.

Note that there is not normally a reason to call this method because `ECDH` only requires a private key and the other party's public key to compute the shared secret. Typically either `ecdh.generateKeys()` or `ecdh.setPrivateKey()` will be called. The `ecdh.setPrivateKey()` method attempts to generate the public point/key associated with the private key being set.

Example (obtaining a shared secret):

```
const crypto = require('crypto');
const alice = crypto.createECDH('secp256k1');
const bob = crypto.createECDH('secp256k1');

// Note: This is a shortcut way to specify one of Alice's previous private
// keys. It would be unwise to use such a predictable private key in a real
// application.
alice.setPrivateKey(
  crypto.createHash('sha256').update('alice', 'utf8').digest()
);

// Bob uses a newly generated cryptographically strong
// pseudorandom key pair bob.generateKeys();
```

```
const alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

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

Class: Hash

The `Hash` class is a utility for creating hash digests of data. It can be used in one of two ways:

- As a `stream` that is both readable and writable, where data is written to produce a computed hash digest on the readable side, or
- Using the `hash.update()` and `hash.digest()` methods to produce the computed hash.

The `crypto.createHash()` method is used to create `Hash` instances. `Hash` objects are not to be created directly using the `new` keyword.

Example: Using `Hash` objects as streams:

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.on('readable', () => {
  var data = hash.read();
  if (data)
    console.log(data.toString('hex'));
  // Prints:
  //   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e5
});

hash.write('some data to hash');
hash.end();
```

Example: Using `Hash` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const hash = crypto.createHash('sha256');

const input = fs.createReadStream('test.js');
input.pipe(hash).pipe(process.stdout);
```

Example: Using the `hash.update()` and `hash.digest()` methods:

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.update('some data to hash');
console.log(hash.digest('hex'));

// Prints:
// 6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
```

hash.digest([encoding])

Calculates the digest of all of the data passed to be hashed (using the `hash.update()` method). The `encoding` can be `'hex'`, `'binary'` or `'base64'`. If `encoding` is provided a string will be returned; otherwise a `Buffer` is returned.

The `Hash` object can not be used again after `hash.digest()` method has been called. Multiple calls will cause an error to be thrown.

hash.update(data[, input_encoding])

Updates the hash content with the given `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a

Buffer then `input_encoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Hmac

#

The `Hmac` Class is a utility for creating cryptographic HMAC digests. It can be used in one of two ways:

- As a `stream` that is both readable and writable, where data is written to produce a computed HMAC digest on the readable side, or
- Using the `hmac.update()` and `hmac.digest()` methods to produce the computed HMAC digest.

The `crypto.createHmac()` method is used to create `Hmac` instances. `Hmac` objects are not to be created directly using the `new` keyword.

Example: Using `Hmac` objects as streams:

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.on('readable', () => {
  var data = hmac.read();
  if (data)
    console.log(data.toString('hex'));
  // Prints:
  //   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77
});

hmac.write('some data to hash');
hmac.end();
```

Example: Using `Hmac` and piped streams:

```
const crypto = require('crypto');
const fs = require('fs');
const hmac = crypto.createHmac('sha256', 'a secret');

const input = fs.createReadStream('test.js');
input.pipe(hmac).pipe(process.stdout);
```

Example: Using the `hmac.update()` and `hmac.digest()` methods:

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.update('some data to hash');
console.log(hmac.digest('hex'));
// Prints:
// 7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
```

`hmac.digest([encoding])`

#

Calculates the HMAC digest of all of the data passed using `hmac.update()`. The `encoding` can be `'hex'`, `'binary'` or `'base64'`. If `encoding` is provided a string is returned; otherwise a `Buffer` is returned;

The `Hmac` object can not be used again after `hmac.digest()` has been called. Multiple calls to `hmac.digest()` will result in an error being thrown.

`hmac.update(data[, input_encoding])`

#

Updates the `Hmac` content with the given `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer` then `input_encoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Sign

#

The `Sign` Class is a utility for generating signatures. It can be used in one of two ways:

- As a writable `stream`, where data to be signed is written and the `sign.sign()` method is used to generate and return the signature, or
- Using the `sign.update()` and `sign.sign()` methods to produce the signature.

The `crypto.createSign()` method is used to create `Sign` instances. `Sign` objects are not to be created directly using the `new` keyword.

Example: Using `Sign` objects as streams:

```
const crypto = require('crypto');
const sign = crypto.createSign('RSA-SHA256');

sign.write('some data to sign');
sign.end();

const private_key = getPrivateKeySomehow();
console.log(sign.sign(private_key, 'hex'));
// Prints the calculated signature
```

Example: Using the `sign.update()` and `sign.sign()` methods:

```
const crypto = require('crypto');
const sign = crypto.createSign('RSA-SHA256');

sign.update('some data to sign');
```

```
const private_key = getPrivateKeySomehow();
console.log(sign.sign(private_key, 'hex'));
    // Prints the calculated signature
```

A `Sign` instance can also be created by just passing in the digest algorithm name, in which case OpenSSL will infer the full signature algorithm from the type of the PEM-formatted private key, including algorithms that do not have directly exposed name constants, e.g. 'ecdsa-with-SHA256'.

Example: signing using ECDSA with SHA256

```
const crypto = require('crypto');
const sign = crypto.createSign('sha256');

sign.update('some data to sign');

const private_key = '-----BEGIN EC PRIVATE KEY-----\n' +
'MHcCAQEEIF+jnWY1D5kbVYDNvxxo/Y+ku2uJPDwS0r/VuPZQrjjVoAoGCCqGSM4
'AwEHoUQDQgAEur0xfSxmqIRYzJVagdZfMMSjRNNhB8i3mXyIMq704m2m52FdfKZ
'pQhByd5eyj3lgZ7m7jbchtdgyOF8Io/1ng==\n' +
'-----END EC PRIVATE KEY-----\n';

console.log(sign.sign(private_key).toString('hex'));
```

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

Calculates the signature on all the data passed through using either `sign.update()` or `sign.write()`.

The `private_key` argument can be an object or a string. If `private_key` is a string, it is treated as a raw key with no passphrase. If `private_key` is an object, it is interpreted as a hash containing two properties:

- `key` : `<String>` - PEM encoded private key

- `passphrase` : `<String>` - passphrase for the private key

The `output_format` can specify one of `'binary'`, `'hex'` or `'base64'`. If `output_format` is provided a string is returned; otherwise a `Buffer` is returned.

The `Sign` object can not be again used after `sign.sign()` method has been called. Multiple calls to `sign.sign()` will result in an error being thrown.

`sign.update(data[, input_encoding])`

#

Updates the `Sign` content with the given `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer` then `input_encoding` is ignored.

This can be called many times with new data as it is streamed.

Class: Verify

#

The `Verify` class is a utility for verifying signatures. It can be used in one of two ways:

- As a writable `stream` where written data is used to validate against the supplied signature, or
- Using the `verify.update()` and `verify.verify()` methods to verify the signature.

The `crypto.createSign()` method is used to create `Sign` instances. `Sign` objects are not to be created directly using the `new` keyword.

Example: Using `Verify` objects as streams:

```
const crypto = require('crypto');
const verify = crypto.createVerify('RSA-SHA256');

verify.write('some data to sign');
verify.end();
```

```
const public_key = getPublicKeySomehow();
const signature = getSignatureToVerify();
console.log(verify.verify(public_key, signature));
// Prints true or false
```

Example: Using the `verify.update()` and `verify.verify()` methods:

```
const crypto = require('crypto');
const verify = crypto.createVerify('RSA-SHA256');

verify.update('some data to sign');

const public_key = getPublicKeySomehow();
const signature = getSignatureToVerify();
console.log(verify.verify(public_key, signature));
// Prints true or false
```

`verifier.update(data[, input_encoding])`

#

Updates the `Verify` content with the given `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. If `encoding` is not provided, and the `data` is a string, an encoding of `'utf8'` is enforced. If `data` is a `Buffer` then `input_encoding` is ignored.

This can be called many times with new data as it is streamed.

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

#

Verifies the provided data using the given `object` and `signature`. The `object` argument is a string containing a PEM encoded object, which can be one an RSA public key, a DSA public key, or an X.509 certificate. The `signature` argument is the previously calculated signature for the data, in the `signature_format` which can be `'binary'`, `'hex'` or `'base64'`. If a `signature_format` is specified, the

`signature` is expected to be a string; otherwise `signature` is expected to be a `Buffer`.

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

The `verifier` object can not be used again after `verify.verify()` has been called. Multiple calls to `verify.verify()` will result in an error being thrown.

crypto module methods and properties

crypto.constants

Returns an object containing commonly used constants for crypto and security related operations. The specific constants currently defined are described in [Crypto Constants](#).

crypto.DEFAULT_ENCODING

The default encoding to use for functions that can take either strings or `buffers`. The default value is `'buffer'`, which makes methods default to `Buffer` objects.

The `crypto.DEFAULT_ENCODING` mechanism is provided for backwards compatibility with legacy programs that expect `'binary'` to be the default encoding.

New applications should expect the default to be `'buffer'`. This property may become deprecated in a future Node.js release.

crypto.fips

Property for checking and controlling whether a FIPS compliant crypto provider is currently in use. Setting to true requires a FIPS build of Node.js.

crypto.createCipher(algorithm, password)

Creates and returns a `Cipher` object that uses the given `algorithm` and `password`.

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

The `password` is used to derive the cipher key and initialization vector (IV). The value must be either a `'binary'` encoded string or a `Buffer`.

The implementation of `crypto.createCipher()` derives keys using the OpenSSL function `EVP_BytesToKey` with the digest algorithm set to MD5, one iteration, and no salt. The lack of salt allows dictionary attacks as the same password always creates the same key. The low iteration count and non-cryptographically secure hash algorithm allow passwords to be tested very rapidly.

In line with OpenSSL's recommendation to use pbkdf2 instead of `EVP_BytesToKey` it is recommended that developers derive a key and IV on their own using `crypto.pbkdf2()` and to use `crypto.createCipheriv()` to create the `Cipher` object.

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

#

Creates and returns a `Cipher` object, with the given `algorithm`, `key` and initialization vector (`iv`).

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

The `key` is the raw key used by the `algorithm` and `iv` is an `initialization vector`. Both arguments must be `'binary'` encoded strings or `buffers`.

`crypto.createCredentials(details)`

#

Stability: 0 - Deprecated: Use `tls.createSecureContext()` instead.

The `crypto.createCredentials()` method is a deprecated alias for creating and returning a `tls.SecureContext` object. The `crypto.createCredentials()`

method should not be used.

The optional `details` argument is a hash object with keys:

- `pfx` : `<String> | <Buffer>` - PFX or PKCS12 encoded private key, certificate and CA certificates
- `key` : `<String>` - PEM encoded private key
- `passphrase` : `<String>` - passphrase for the private key or PFX
- `cert` : `<String>` - PEM encoded certificate
- `ca` : `<String> | <Array>` - Either a string or array of strings of PEM encoded CA certificates to trust.
- `crl` : `<String> | <Array>` - Either a string or array of strings of PEM encoded CRLs (Certificate Revocation List)
- `ciphers` : `<String>` using the [OpenSSL cipher list format](#) describing the cipher algorithms to use or exclude.

If no 'ca' details are given, Node.js will use Mozilla's default [publicly trusted list of CAs](#).

`crypto.createDecipher(algorithm, password)`

#

Creates and returns a `Decipher` object that uses the given `algorithm` and `password` (key).

The implementation of `crypto.createDecipher()` derives keys using the OpenSSL function [EVP_BytesToKey](#) with the digest algorithm set to MD5, one iteration, and no salt. The lack of salt allows dictionary attacks as the same password always creates the same key. The low iteration count and non-cryptographically secure hash algorithm allow passwords to be tested very rapidly.

In line with OpenSSL's recommendation to use pbkdf2 instead of [EVP_BytesToKey](#) it is recommended that developers derive a key and IV on their own using `crypto.pbkdf2()` and to use `crypto.createDecipheriv()` to create the `Decipher` object.

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

#

Creates and returns a `Decipher` object that uses the given `algorithm`, `key` and initialization vector (`iv`).

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

The `key` is the raw key used by the `algorithm` and `iv` is an `initialization vector`. Both arguments must be '`'binary'`' encoded strings or `buffers`.

```
crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding]) #
```

Creates a `DiffieHellman` key exchange object using the supplied `prime` and an optional specific `generator`.

The `generator` argument can be a number, string, or `Buffer`. If `generator` is not specified, the value `2` is used.

The `prime_encoding` and `generator_encoding` arguments can be '`'binary'`', '`'hex'`', or '`'base64'`'.

If `prime_encoding` is specified, `prime` is expected to be a string; otherwise a `Buffer` is expected.

If `generator_encoding` is specified, `generator` is expected to be a string; otherwise either a number or `Buffer` is expected.

```
crypto.createDiffieHellman(prime_length[, generator]) #
```

Creates a `DiffieHellman` key exchange object and generates a prime of `prime_length` bits using an optional specific numeric `generator`. If `generator` is not specified, the value `2` is used.

```
crypto.createECDH(curve_name) #
```

Creates an Elliptic Curve Diffie-Hellman (`ECDH`) key exchange object using a predefined curve specified by the `curve_name` string. Use `crypto.getCurves()` to

obtain a list of available curve names. On recent OpenSSL releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve.

crypto.createHash(algorithm)

Creates and returns a `Hash` object that can be used to generate hash digests using the given `algorithm`.

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

Example: generating the sha256 sum of a file

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

const hash = crypto.createHash('sha256');

const input = fs.createReadStream(filename);
input.on('readable', () => {
  var data = input.read();
  if (data)
    hash.update(data);
  else {
    console.log(`#${hash.digest('hex')} ${filename}`);
  }
});
```

crypto.createHmac(algorithm, key)

Creates and returns an `Hmac` object that uses the given `algorithm` and `key`.

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

The `key` is the HMAC key used to generate the cryptographic HMAC hash.

Example: generating the sha256 HMAC of a file

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

const hmac = crypto.createHmac('sha256', 'a secret');

const input = fs.createReadStream(filename);
input.on('readable', () => {
  var data = input.read();
  if (data)
    hmac.update(data);
  else {
    console.log(`#${hmac.digest('hex')} ${filename}`);
  }
});
```

`crypto.createSign(algorithm)`

#

Creates and returns a `Sign` object that uses the given `algorithm`. On recent OpenSSL releases, `openssl list-public-key-algorithms` will display the available signing algorithms. One example is `'RSA-SHA256'`.

`crypto.createVerify(algorithm)`

#

Creates and returns a `Verify` object that uses the given algorithm. On recent OpenSSL releases, `openssl list-public-key-algorithms` will display the available signing algorithms. One example is '`RSA-SHA256`'.

crypto.getCiphers()

Returns an array with the names of the supported cipher algorithms.

Example:

```
const ciphers = crypto.getCiphers();
console.log(ciphers); // ['aes-128-cbc', 'aes-128-ccm', ...]
```

crypto.getCurves()

Returns an array with the names of the supported elliptic curves.

Example:

```
const curves = crypto.getCurves();
console.log(curves); // ['secp256k1', 'secp384r1', ...]
```

crypto.getDiffieHellman(group_name)

Creates a predefined `DiffieHellman` key exchange object. The supported groups are: '`modp1`' , '`modp2`' , '`modp5`' (defined in [RFC 2412](#), but see [Caveats](#)) and '`modp14`' , '`modp15`' , '`modp16`' , '`modp17`' , '`modp18`' (defined in [RFC 3526](#)).

The returned object mimics the interface of objects created by `crypto.createDiffieHellman()`, but will not allow changing the keys (with `diffieHellman.setPublicKey()` for example). The advantage of using this method is that the parties do not have to generate nor exchange a group modulus beforehand, saving both processor and communication time.

Example (obtaining a shared secret):

```
const crypto = require('crypto');
const alice = crypto.getDiffieHellman('modp14');
const bob = crypto.getDiffieHellman('modp14');

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

const alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex')
const bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

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

crypto.getHashes()

#

Returns an array with the names of the supported hash algorithms.

Example:

```
const hashes = crypto.getHashes();
console.log(hashes); // ['sha', 'sha1', 'sha1WithRSAEncryption', ...]
```

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

#

Provides an asynchronous Password-Based Key Derivation Function 2 (PBKDF2) implementation. A selected HMAC digest algorithm specified by `digest` is applied to derive a key of the requested byte length (`keylen`) from the `password`, `salt` and `iterations`.

The supplied `callback` function is called with two arguments: `err` and `derivedKey`. If an error occurs, `err` will be set; otherwise `err` will be null. The successfully generated `derivedKey` will be passed as a `Buffer`.

The `iterations` argument must be a number set as high as possible. The higher the number of iterations, the more secure the derived key will be, but will take a longer amount of time to complete.

The `salt` should also be as unique as possible. It is recommended that the salts are random and their lengths are greater than 16 bytes. See [NIST SP 800-132](#) for details.

Example:

```
const crypto = require('crypto');
crypto.pbkdf2('secret', 'salt', 100000, 512, 'sha512', (err, key) => {
  if (err) throw err;
  console.log(key.toString('hex')); // 'c5e478d...1469e50'
});
```

An array of supported digest functions can be retrieved using `crypto.getHashes()`.

`crypto.pbkdf2Sync(password, salt, iterations, keylen, digest)` #

Provides a synchronous Password-Based Key Derivation Function 2 (PBKDF2) implementation. A selected HMAC digest algorithm specified by `digest` is applied to derive a key of the requested byte length (`keylen`) from the `password`, `salt` and `iterations`.

If an error occurs an Error will be thrown, otherwise the derived key will be returned as a [Buffer](#).

The `iterations` argument must be a number set as high as possible. The higher the number of iterations, the more secure the derived key will be, but will take a longer amount of time to complete.

The `salt` should also be as unique as possible. It is recommended that the salts are random and their lengths are greater than 16 bytes. See [NIST SP 800-132](#) for

details.

Example:

```
const crypto = require('crypto');
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 512, 'sha512');
console.log(key.toString('hex')); // 'c5e478d...1469e50'
```

An array of supported digest functions can be retrieved using `crypto.getHashes()`.

crypto.privateDecrypt(private_key, buffer) #

Decrypts `buffer` with `private_key`.

`private_key` can be an object or a string. If `private_key` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_OAEP_PADDING`. If `private_key` is an object, it is interpreted as a hash object with the keys:

- `key` : `<String>` - PEM encoded private key
- `passphrase` : `<String>` - Optional passphrase for the private key
- `padding` : An optional padding value, one of the following:
 - `crypto.constants.RSA_NO_PADDING`
 - `crypto.constants.RSA_PKCS1_PADDING`
 - `crypto.constants.RSA_PKCS1_OAEP_PADDING`

All paddings are defined in `crypto.constants`.

crypto.privateEncrypt(private_key, buffer) #

Encrypts `buffer` with `private_key`.

`private_key` can be an object or a string. If `private_key` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_PADDING`. If `private_key` is an object, it is interpreted as a hash object with the keys:

- `key` : `<String>` - PEM encoded private key
- `passphrase` : `<String>` - Optional passphrase for the private key
- `padding` : An optional padding value, one of the following:
 - `crypto.constants.RSA_NO_PADDING`
 - `crypto.constants.RSA_PKCS1_PADDING`
 - `crypto.constants.RSA_PKCS1_OAEP_PADDING`

All paddings are defined in `crypto.constants`.

`crypto.publicDecrypt(public_key, buffer)`

Decrypts `buffer` with `public_key`.

`public_key` can be an object or a string. If `public_key` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_PADDING`. If `public_key` is an object, it is interpreted as a hash object with the keys:

- `key` : `<String>` - PEM encoded public key
- `passphrase` : `<String>` - Optional passphrase for the private key
- `padding` : An optional padding value, one of the following:
 - `crypto.constants.RSA_NO_PADDING`
 - `crypto.constants.RSA_PKCS1_PADDING`
 - `crypto.constants.RSA_PKCS1_OAEP_PADDING`

Because RSA public keys can be derived from private keys, a private key may be passed instead of a public key.

All paddings are defined in `crypto.constants`.

`crypto.publicEncrypt(public_key, buffer)`

Encrypts `buffer` with `public_key`.

`public_key` can be an object or a string. If `public_key` is a string, it is treated as the key with no passphrase and will use `RSA_PKCS1_OAEP_PADDING`. If `public_key` is an object, it is interpreted as a hash object with the keys:

- `key` : `<String>` - PEM encoded public key
- `passphrase` : `<String>` - Optional passphrase for the private key
- `padding` : An optional padding value, one of the following:
 - `crypto.constants.RSA_NO_PADDING`
 - `crypto.constants.RSA_PKCS1_PADDING`
 - `crypto.constants.RSA_PKCS1_OAEP_PADDING`

Because RSA public keys can be derived from private keys, a private key may be passed instead of a public key.

All paddings are defined in `crypto.constants`.

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

Generates cryptographically strong pseudo-random data. The `size` argument is a number indicating the number of bytes to generate.

If a `callback` function is provided, the bytes are generated asynchronously and the `callback` function is invoked with two arguments: `err` and `buf`. If an error occurs, `err` will be an Error object; otherwise it is null. The `buf` argument is a `Buffer` containing the generated bytes.

```
// Asynchronous
const crypto = require('crypto');
crypto.randomBytes(256, (err, buf) => {
  if (err) throw err;
  console.log(`#${buf.length} bytes of random data: ${buf.toString('hex')}`);
});
```

If the `callback` function is not provided, the random bytes are generated synchronously and returned as a `Buffer`. An error will be thrown if there is a problem generating the bytes.

```
// Synchronous
```

```
const buf = crypto.randomBytes(256);
console.log(`\${buf.length} bytes of random data: \${buf.toString('hex')}`);
```

The `crypto.randomBytes()` method will block until there is sufficient entropy. This should normally never take longer than a few milliseconds. The only time when generating the random bytes may conceivably block for a longer period of time is right after boot, when the whole system is still low on entropy.

`crypto.setEngine(engine[, flags])` #

Load and set the `engine` for some or all OpenSSL functions (selected by flags).

`engine` could be either an id or a path to the engine's shared library.

The optional `flags` argument uses `ENGINE_METHOD_ALL` by default. The `flags` is a bit field taking one of or a mix of the following flags (defined in `crypto.constants`):

- `crypto.constants.ENGINE_METHOD_RSA`
- `crypto.constants.ENGINE_METHOD_DSA`
- `crypto.constants.ENGINE_METHOD_DH`
- `crypto.constants.ENGINE_METHOD_RAND`
- `crypto.constants.ENGINE_METHOD_ECDH`
- `crypto.constants.ENGINE_METHOD_ECDSA`
- `crypto.constants.ENGINE_METHOD_CIPHERS`
- `crypto.constants.ENGINE_METHOD_DIGESTS`
- `crypto.constants.ENGINE_METHOD_STORE`
- `crypto.constants.ENGINE_METHOD_PKEY_METHS`
- `crypto.constants.ENGINE_METHOD_PKEY_ASN1_METHS`
- `crypto.constants.ENGINE_METHOD_ALL`
- `crypto.constants.ENGINE_METHOD_NONE`

Legacy Streams API (pre Node.js v0.10)

#

The Crypto module was added to Node.js before there was the concept of a unified Stream API, and before there were `Buffer` objects for handling binary data. As such, the many of the `crypto` defined classes have methods not typically found on other Node.js classes that implement the `streams` API (e.g. `update()`, `final()`, or `digest()`). Also, many methods accepted and returned `'binary'` encoded strings by default rather than Buffers. This default was changed after Node.js v0.8 to use `Buffer` objects by default instead.

Recent ECDH Changes

#

Usage of `ECDH` with non-dynamically generated key pairs has been simplified. Now, `ecdh.setPrivateKey()` can be called with a preselected private key and the associated public point (key) will be computed and stored in the object. This allows code to only store and provide the private part of the EC key pair.
`ecdh.setPrivateKey()` now also validates that the private key is valid for the selected curve.

The `ecdh.setPublicKey()` method is now deprecated as its inclusion in the API is not useful. Either a previously stored private key should be set, which automatically generates the associated public key, or `ecdh.generateKeys()` should be called. The main drawback of using `ecdh.setPublicKey()` is that it can be used to put the ECDH key pair into an inconsistent state.

Support for weak or compromised algorithms

#

The `crypto` module still supports some algorithms which are already compromised and are not currently recommended for use. The API also allows the use of ciphers and hashes with a small key size that are considered to be too weak for safe use.

Users should take full responsibility for selecting the crypto algorithm and key size according to their security requirements.

Based on the recommendations of NIST SP 800-131A:

- MD5 and SHA-1 are no longer acceptable where collision resistance is required such as digital signatures.
- The key used with RSA, DSA and DH algorithms is recommended to have at least 2048 bits and that of the curve of ECDSA and ECDH at least 224 bits, to be safe to use for several years.
- The DH groups of `modp1`, `modp2` and `modp5` have a key size smaller than 2048 bits and are not recommended.

See the reference for other recommendations and details.

Crypto Constants

The following constants exported by `crypto.constants` apply to various uses of the `crypto`, `tls`, and `https` modules and are generally specific to OpenSSL.

OpenSSL Options

Constant	Description
<code>SSL_OP_ALL</code>	Applies multiple bug fixes. See https://www.openssl.org/docs/man1.1.1/man3/SSL_OP_ALL.html for detail.
<code>SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION</code>	Allows legacy insecure renegotiations between clients or servers. See https://www.openssl.org/docs/man1.1.1/man3/SSL_OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION.html .
<code>SSL_OP_CIPHER_SERVER_PREFERENCE</code>	Uses the server's preferred cipher. See https://www.openssl.org/docs/man1.1.1/man3/SSL_OP_CIPHER_SERVER_PREFERENCE.html .
<code>SSL_OP_CISCO_ANYCONNECT</code>	Instructs OpenSSL to support Cisco AnyConnect.
<code>SSL_OP_COOKIE_EXCHANGE</code>	Instructs OpenSSL to support cookie exchange.
<code>SSL_OP_CRYPTOPRO_TLSEXT_BUG</code>	Instructs OpenSSL to support the cryptopro draft.

SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	Instructs OpenSSL not to insert empty fragments, added in OpenSSL 1.1.1.
SSL_OP_EPHEMERAL_RSA	Instructs OpenSSL to use ephemeral RSA keys for operations.
SSL_OP_LEGACY_SERVER_CONNECT	Allow initial connection to legacy servers.
SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER	
SSL_OP_MICROSOFT_SESS_ID_BUG	
SSL_OP_MSIE_SSLV2_RSA_PADDING	Instructs OpenSSL to use Microsoft's SSLv2 RSA padding, protocol-version vulnerability.
SSL_OP_NETSCAPE_CA_DN_BUG	
SSL_OP_NETSCAPE_CHALLENGE_BUG	
SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG	
SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG	
SSL_OP_NO_COMPRESSION	Instructs OpenSSL not to use compression.
SSL_OP_NO_QUERY_MTU	
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	Instructs OpenSSL not to resume sessions during renegotiation.
SSL_OP_NO_SSLv2	Instructs OpenSSL not to support SSLv2.
SSL_OP_NO_SSLv3	Instructs OpenSSL not to support SSLv3.
SSL_OP_NO_TICKET	Instructs OpenSSL not to use session tickets.

SSL_OP_NO_TLSv1	Instructs OpenSSL to not use TLSv1.
SSL_OP_NO_TLSv1_1	Instructs OpenSSL to not use TLSv1.1.
SSL_OP_NO_TLSv1_2	Instructs OpenSSL to not use TLSv1.2.
SSL_OP_PKCS1_CHECK_1	
SSL_OP_PKCS1_CHECK_2	
SSL_OP_SINGLE_DH_USE	Instructs OpenSSL to use temporary/ephemeral DH parameters.
SSL_OP_SINGLE_ECDH_USE	Instructs OpenSSL to use temporary/ephemeral ECDH parameters.
SSL_OP_SSLEAY_080_CLIENT_DH_BUG	
SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG	
SSL_OP_TLS_BLOCK_PADDING_BUG	
SSL_OP_TLS_D5_BUG	
SSL_OP_TLS_ROLLBACK_BUG	Instructs OpenSSL to not use TLSv1.2 or later.

OpenSSL Engine Constants

#

Constant	Description
ENGINE_METHOD_RSA	Limit engine usage to RSA
ENGINE_METHOD_DSA	Limit engine usage to DSA

ENGINE_METHOD_DH	Limit engine usage to DH
ENGINE_METHOD RAND	Limit engine usage to RAND
ENGINE_METHOD_ECDH	Limit engine usage to ECDH
ENGINE_METHOD_ECDSA	Limit engine usage to ECDSA
ENGINE_METHOD_CIPHERS	Limit engine usage to CIPHERS
ENGINE_METHOD_DIGESTS	Limit engine usage to DIGESTS
ENGINE_METHOD_STORE	Limit engine usage to STORE
ENGINE_METHOD_PKEY_METHS	Limit engine usage to PKEY_METHDS
ENGINE_METHOD_PKEY ASN1_METHS	Limit engine usage to PKEY ASN1_METHS
ENGINE_METHOD_ALL	
ENGINE_METHOD_NONE	

Other OpenSSL Constants

#

Constant	Description
DH_CHECK_P_NOT_SAFE_PRIME	
DH_CHECK_P_NOT_PRIME	
DH_UNABLE_TO_CHECK_GENERATOR	

DH_NOT_SUITABLE_GENERATOR	
NPN_ENABLED	
ALPN_ENABLED	
RSA_PKCS1_PADDING	
RSA_SSLV23_PADDING	
RSA_NO_PADDING	
RSA_PKCS1_OAEP_PADDING	
RSA_X931_PADDING	
RSA_PKCS1_PSS_PADDING	
POINT_CONVERSION_COMPRESSED	
POINT_CONVERSION_UNCOMPRESSED	
POINT_CONVERSION_HYBRID	

Node.js Crypto Constants

Constant	Description
defaultCoreCipherList	Specifies the built-in default cipher list used by Node.js.
defaultCipherList	Specifies the active default cipher list used by the current Node.js process.

Debugger

#

Stability: 2 - Stable

Node.js includes a full-featured out-of-process debugging utility accessible via a simple [TCP-based protocol](#) and built-in debugging client. To use it, start Node.js with the `debug` argument followed by the path to the script to debug; a prompt will be displayed indicating successful launch of the debugger:

```
$ node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(() => {
  3   debugger;
debug>
```

Node.js's debugger client is not a full-featured debugger, but simple step and inspection are possible.

Inserting the statement `debugger;` into the source code of a script will enable a breakpoint at that position in the code:

```
// myscript.js
x = 5;
setTimeout(() => {
  debugger;
  console.log('world');
}, 1000);
console.log('hello');
```

Once the debugger is run, a breakpoint will occur at line 4:

```
$ node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
1 x = 5;
2 setTimeout(() => {
3   debugger;
debug> cont
< hello
break in /home/indutny/Code/git/indutny/myscript.js:3
1 x = 5;
2 setTimeout(() => {
3   debugger;
4   console.log('world');
5 }, 1000);
debug> next
break in /home/indutny/Code/git/indutny/myscript.js:4
2 setTimeout(() => {
3   debugger;
4   console.log('world');
5 }, 1000);
6 console.log('hello');
debug> repl
Press Ctrl + C to leave debug repl
> x
5
> 2+2
4
debug> next
< world
break in /home/indutny/Code/git/indutny/myscript.js:5
```

```
3 debugger;  
4 console.log('world');  
5 }, 1000);  
6 console.log('hello');  
7  
debug> quit
```

The `repl` command allows code to be evaluated remotely. The `next` command steps to the next line. Type `help` to see what other commands are available.

Pressing `enter` without typing a command will repeat the previous debugger command.

Watchers

It is possible to watch expression and variable values while debugging. On every breakpoint, each expression from the watchers list will be evaluated in the current context and displayed immediately before the breakpoint's source code listing.

To begin watching an expression, type `watch('my_expression')`. The command `watchers` will print the active watchers. To remove a watcher, type `unwatch('my_expression')`.

Command reference

Stepping

- `cont` , `c` - Continue execution
- `next` , `n` - Step next
- `step` , `s` - Step in
- `out` , `o` - Step out
- `pause` - Pause running code (like pause button in Developer Tools)

Breakpoints

- `setBreakpoint()`, `sb()` - Set breakpoint on current line
- `setBreakpoint(line)`, `sb(line)` - Set breakpoint on specific line
- `setBreakpoint('fn()')`, `sb(...)` - Set breakpoint on a first statement in functions body
- `setBreakpoint('script.js', 1)`, `sb(...)` - Set breakpoint on first line of script.js
- `clearBreakpoint('script.js', 1)`, `cb(...)` - Clear breakpoint in script.js on line 1

It is also possible to set a breakpoint in a file (module) that isn't loaded yet:

```
$ ./node debug test/fixtures/break-in-module/main.js
< debugger listening on port 5858
connecting to port 5858... ok
break in test/fixtures/break-in-module/main.js:1
  1 var mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();

debug> setBreakpoint('mod.js', 23)
Warning: script 'mod.js' was not loaded yet.

  1 var mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();

debug> c
break in test/fixtures/break-in-module/mod.js:23
  21
  22 exports.hello = () => {
  23   return 'hello from module';
  24 };
  25

debug>
```

- `backtrace`, `bt` - Print backtrace of current execution frame
- `list(5)` - List scripts source code with 5 line context (5 lines before and after)
- `watch(expr)` - Add expression to watch list
- `unwatch(expr)` - Remove expression from watch list
- `watchers` - List all watchers and their values (automatically listed on each breakpoint)
- `repl` - Open debugger's repl for evaluation in debugging script's context
- `exec expr` - Execute an expression in debugging script's context

Execution control

#

- `run` - Run script (automatically runs on debugger's start)
- `restart` - Restart script
- `kill` - Kill script

Various

#

- `scripts` - List all loaded scripts
- `version` - Display V8's version

Advanced Usage

#

An alternative way of enabling and accessing the debugger is to start Node.js with the `--debug` command-line flag or by signaling an existing Node.js process with `SIGUSR1`.

Once a process has been set in debug mode this way, it can be inspected using the Node.js debugger by either connecting to the `pid` of the running process or via URI reference to the listening debugger:

- `node debug -p <pid>` - Connects to the process via the `pid`
- `node debug <URI>` - Connects to the process via the URI such as `localhost:5858`

V8 Inspector Integration for Node.js

#

NOTE: This is an experimental feature.

V8 Inspector integration allows attaching Chrome DevTools to Node.js instances for debugging and profiling.

V8 Inspector can be enabled by passing the `--inspect` flag when starting a Node.js application. It is also possible to supply a custom port with that flag, e.g. `--inspect=9222` will accept DevTools connections on port 9222.

To break on the first line of the application code, provide the `--debug-brk` flag in addition to `--inspect`.

UDP / Datagram Sockets

#

Stability: 2 - Stable

The `dgram` module provides an implementation of UDP Datagram sockets.

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
```

```
var address = server.address();
console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

Class: dgram.Socket

The `dgram.Socket` object is an `EventEmitter` that encapsulates the datagram functionality.

New instances of `dgram.Socket` are created using `dgram.createSocket()`. The `new` keyword is not to be used to create `dgram.Socket` instances.

Event: 'close'

The `'close'` event is emitted after a socket is closed with `close()`. Once triggered, no new `'message'` events will be emitted on this socket.

Event: 'error'

- `exception <Error>`

The `'error'` event is emitted whenever any error occurs. The event handler function is passed a single Error object.

Event: 'listening'

The `'listening'` event is emitted whenever a socket begins listening for datagram messages. This occurs as soon as UDP sockets are created.

Event: 'message'

- `msg <Buffer>` - The message
- `rinfo <Object>` - Remote address information

The `'message'` event is emitted when a new datagram is available on a socket. The

event handler function is passed two arguments: `msg` and `rinfo`. The `msg` argument is a `Buffer` and `rinfo` is an object with the sender's address information provided by the `address`, `family` and `port` properties:

```
socket.on('message', (msg, rinfo) => {
  console.log('Received %d bytes from %s:%d\n',
    msg.length, rinfo.address, rinfo.port);
});
```

socket.addMembership(multicastAddress[, multicastInterface])

Added in: v0.6.9

- `multicastAddress` `<String>`
- `multicastInterface` `<String>`, Optional

Tells the kernel to join a multicast group at the given `multicastAddress` using the `IP_ADD_MEMBERSHIP` socket option. If the `multicastInterface` argument is not specified, the operating system will try to add membership to all valid networking interfaces.

socket.address()

Returns an object containing the address information for a socket. For UDP sockets, this object will contain `address`, `family` and `port` properties.

socket.bind([port][, address][, callback])

- `port` `<Number>` - Integer, Optional
- `address` `<String>`, Optional
- `callback` `<Function>` with no parameters, Optional. Called when binding is complete.

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address`. If `port` is not specified, the operating system will attempt to bind to a random port. If `address` is not specified, the operating

system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

Note that specifying both a `'listening'` event listener and passing a `callback` to the `socket.bind()` method is not harmful but not very useful.

A bound datagram socket keeps the Node.js process running to receive datagram messages.

If binding fails, an `'error'` event is generated. In rare case (e.g. attempting to bind with a closed socket), an `Error` may be thrown.

Example of a UDP server listening on port 41234:

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
  console.log(`server error:\n${err.stack}`);
  server.close();
});

server.on('message', (msg, rinfo) => {
  console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
  var address = server.address();
  console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

socket.bind(options[, callback])

- `options <Object>` - Required. Supports the following properties:
 - `port <Number>` - Required.
 - `address <String>` - Optional.
 - `exclusive <Boolean>` - Optional.
- `callback <Function>` - Optional.

For UDP sockets, causes the `dgram.Socket` to listen for datagram messages on a named `port` and optional `address` that are passed as properties of an `options` object passed as the first argument. If `port` is not specified, the operating system will attempt to bind to a random port. If `address` is not specified, the operating system will attempt to listen on all addresses. Once binding is complete, a `'listening'` event is emitted and the optional `callback` function is called.

The `options` object may contain an additional `exclusive` property that is used when using `dgram.Socket` objects with the [`cluster`] module. When `exclusive` is set to `false` (the default), cluster workers will use the same underlying socket handle allowing connection handling duties to be shared. When `exclusive` is `true`, however, the handle is not shared and attempted port sharing results in an error.

An example socket listening on an exclusive port is shown below.

```
socket.bind({  
    address: 'localhost',  
    port: 8000,  
    exclusive: true  
});
```

socket.close([callback])

Close the underlying socket and stop listening for data on it. If a callback is provided, it is added as a listener for the `'close'` event.

socket.dropMembership(multicastAddress[, multicastInterface])

Added in: v0.6.9

- `multicastAddress` `<String>`
- `multicastInterface` `<String>`, Optional

Instructs the kernel to leave a multicast group at `multicastAddress` using the `IP_DROP_MEMBERSHIP` socket option. This method is automatically called by the kernel when the socket is closed or the process terminates, so most apps will never have reason to call this.

If `multicastInterface` is not specified, the operating system will attempt to drop membership on all valid interfaces.

socket.send(msg, [offset, length,] port, address[, callback])

- `msg` `<Buffer>` | `<String>` | `<Array>` Message to be sent
- `offset` `<Number>` Integer. Optional. Offset in the buffer where the message starts.
- `length` `<Number>` Integer. Optional. Number of bytes in the message.
- `port` `<Number>` Integer. Destination port.
- `address` `<String>` Destination hostname or IP address.
- `callback` `<Function>` Called when the message has been sent. Optional.

Broadcasts a datagram on the socket. The destination `port` and `address` must be specified.

The `msg` argument contains the message to be sent. Depending on its type, different behavior can apply. If `msg` is a `Buffer`, the `offset` and `length` specify the offset within the `Buffer` where the message begins and the number of bytes in the message, respectively. If `msg` is a `String`, then it is automatically converted to a `Buffer` with `'utf8'` encoding. With messages that contain multi-byte characters, `offset` and `length` will be calculated with respect to `byte length` and not the character position. If `msg` is an array, `offset` and `length` must not be specified.

The `address` argument is a string. If the value of `address` is a host name, DNS will be used to resolve the address of the host. If the `address` is not specified or is an empty string, '`127.0.0.1`' or '`::1`' will be used instead.

If the socket has not been previously bound with a call to `bind`, the socket is assigned a random port number and is bound to the "all interfaces" address (`'0.0.0.0'` for `udp4` sockets, `::0` for `udp6` sockets.)

An optional `callback` function may be specified to as a way of reporting DNS errors or for determining when it is safe to reuse the `buf` object. Note that DNS lookups delay the time to send for at least one tick of the Node.js event loop.

The only way to know for sure that the datagram has been sent is by using a `callback`. If an error occurs and a `callback` is given, the error will be passed as the first argument to the `callback`. If a `callback` is not given, the error is emitted as an '`error`' event on the `socket` object.

Offset and length are optional, but if you specify one you would need to specify the other. Also, they are supported only when the first argument is a `Buffer`.

Example of sending a UDP packet to a random port on `localhost`;

```
const dgram = require('dgram');
const message = Buffer.from('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});
```

Example of sending a UDP packet composed of multiple buffers to a random port on `localhost`;

```
const dgram = require('dgram');
const buf1 = Buffer.from('Some ');
```

```
const buf2 = Buffer.from('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, 'localhost', (err) => {
  client.close();
});
```

Sending multiple buffers might be faster or slower depending on your application and operating system: benchmark it. Usually it is faster.

A Note about UDP datagram size

The maximum size of an IPv4/v6 datagram depends on the MTU (Maximum Transmission Unit) and on the Payload Length field size.

- The Payload Length field is 16 bits wide, which means that a normal payload exceed 64K octets *including* the internet header and data (65,507 bytes = 65,535 – 8 bytes UDP header – 20 bytes IP header); this is generally true for loopback interfaces, but such long datagram messages are impractical for most hosts and networks.
- The MTU is the largest size a given link layer technology can support for datagram messages. For any link, IPv4 mandates a minimum MTU of 68 octets, while the recommended MTU for IPv4 is 576 (typically recommended as the MTU for dial-up type applications), whether they arrive whole or in fragments.

For IPv6, the minimum MTU is 1280 octets, however, the mandatory minimum fragment reassembly buffer size is 1500 octets. The value of 68 octets is very small, since most current link layer technologies, like Ethernet, have a minimum MTU of 1500.

It is impossible to know in advance the MTU of each link through which a packet might travel. Sending a datagram greater than the receiver MTU will not work because the packet will get silently dropped without informing the source that the data did not reach its intended recipient.

socket.setBroadcast(flag)

Added in: v0.6.9

- flag <Boolean>

Sets or clears the `SO_BROADCAST` socket option. When set to `true`, UDP packets may be sent to a local interface's broadcast address.

socket.setMulticastLoopback(flag)

- flag <Boolean>

Sets or clears the `IP_MULTICAST_LOOP` socket option. When set to `true`, multicast packets will also be received on the local interface.

socket.setMulticastTTL(ttl)

- ttl <Number> Integer

Sets the `IP_MULTICAST_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel 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 passed to `socket.setMulticastTTL()` is a number of hops between 0 and 255. The default on most systems is `1` but can vary.

socket.setTTL(ttl)

- ttl <Number> Integer

Sets the `IP_TTL` socket option. While TTL generally stands for "Time to Live", in this context it specifies the number of IP hops that a packet is allowed to travel 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 `socket.setTTL()` is a number of hops between 1 and 255. The default on most systems is 64 but can vary.

socket.ref()

#

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active. The `socket.ref()` method adds the socket back to the reference counting and restores the default behavior.

Calling `socket.ref()` multiples times will have no additional effect.

The `socket.ref()` method returns a reference to the socket so calls can be chained.

socket.unref()

#

By default, binding a socket will cause it to block the Node.js process from exiting as long as the socket is open. The `socket.unref()` method can be used to exclude the socket from the reference counting that keeps the Node.js process active, allowing the process to exit even if the socket is still listening.

Calling `socket.unref()` multiple times will have no addition effect.

The `socket.unref()` method returns a reference to the socket so calls can be chained.

Change to asynchronous `socket.bind()` behavior

#

As of Node.js v0.10, `dgram.Socket#bind()` changed to an asynchronous execution model. Legacy code that assumes synchronous behavior, as in the following example:

```
const s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

Must be changed to pass a callback function to the `dgram.Socket#bind()` function:

```
const s = dgram.createSocket('udp4');
s.bind(1234, () => {
  s.addMembership('224.0.0.114');
});
```

dgram module functions

dgram.createSocket(options[, callback])

- `options` <Object>
- `callback` <Function> Attached as a listener to 'message' events.
- Returns: <dgram.Socket>

Creates a `dgram.Socket` object. The `options` argument is an object that should contain a `type` field of either `udp4` or `udp6` and an optional boolean `reuseAddr` field.

When `reuseAddr` is `true` `socket.bind()` will reuse the address, even if another process has already bound a socket on it. `reuseAddr` defaults to `false`. An optional `callback` function can be passed specified which is added as a listener for 'message' events.

Once the socket is created, calling `socket.bind()` will instruct the socket to begin listening for datagram messages. When `address` and `port` are not passed to `socket.bind()` the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). The bound address and port can be retrieved using `socket.address().address` and `socket.address().port`.

dgram.createSocket(type[, callback])

- `type` <String> - Either 'udp4' or 'udp6'
- `callback` <Function> - Attached as a listener to 'message' events.
Optional

- Returns: `<dgram.Socket>`

Creates a `dgram.Socket` object of the specified `type`. The `type` argument can be either `udp4` or `udp6`. An optional `callback` function can be passed which is added as a listener for 'message' events.

Once the socket is created, calling `socket.bind()` will instruct the socket to begin listening for datagram messages. When `address` and `port` are not passed to `socket.bind()` the method will bind the socket to the "all interfaces" address on a random port (it does the right thing for both `udp4` and `udp6` sockets). The bound address and port can be retrieved using `socket.address().address` and `socket.address().port`.

DNS

#

Stability: 2 - Stable

The `dns` module contains functions belonging to two different categories:

1) Functions that use the underlying operating system facilities to perform name resolution, and that do not necessarily perform any network communication. This category contains only one function: `dns.lookup()`. Developers looking to perform name resolution in the same way that other applications on the same operating system behave should use `dns.lookup()`.

For example, looking up `nodejs.org`.

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

dns.lookup('nodejs.org', (err, addresses, family) => {
  console.log('addresses:', addresses);
});
```

2) Functions that connect to an actual DNS server to perform name resolution, and that *always* use the network to perform DNS queries. This category contains all functions in the `dns` module except `dns.lookup()`. These functions do not use the same set of configuration files used by `dns.lookup()` (e.g. `/etc/hosts`). These functions should be used by developers who do not want to use the underlying operating system's facilities for name resolution, and instead want to *always* perform DNS queries.

Below is an example that resolves '`'nodejs.org'`' then reverse resolves the IP addresses that are returned.

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

dns.resolve4('nodejs.org', (err, addresses) => {
  if (err) throw err;

  console.log(`addresses: ${JSON.stringify(addresses)}`);

  addresses.forEach((a) => {
    dns.reverse(a, (err, hostnames) => {
      if (err) {
        throw err;
      }
      console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
    });
  });
});
```

There are subtle consequences in choosing one over the other, please consult the [Implementation considerations section](#) for more information.

`dns.getServers()`

#

Added in: v0.11.3

Returns an array of IP address strings that are being used for name resolution.

dns.lookup(hostname[, options], callback)

#

Added in: v0.1.90

Resolves a hostname (e.g. `'nodejs.org'`) into the first found A (IPv4) or AAAA (IPv6) record. `options` can be an object or integer. If `options` is not provided, then IPv4 and IPv6 addresses are both valid. If `options` is an integer, then it must be `4` or `6`.

Alternatively, `options` can be an object containing these properties:

- `family <Number>` - The record family. If present, must be the integer `4` or `6`. If not provided, both IP v4 and v6 addresses are accepted.
- `hints : <Number>` - If present, it should be one or more of the supported `getaddrinfo` flags. If `hints` is not provided, then no flags are passed to `getaddrinfo`. Multiple flags can be passed through `hints` by logically `OR` ing their values. See [supported getaddrinfo flags](#) for more information on supported flags.
- `all : <Boolean>` - When `true`, the callback returns all resolved addresses in an array, otherwise returns a single address. Defaults to `false`.

All properties are optional. An example usage of options is shown below.

```
{  
  family: 4,  
  hints: dns.ADDRCONFIG | dns.V4MAPPED,  
  all: false  
}
```

The `callback` function has arguments `(err, address, family)`. `address` is a string representation of an IPv4 or IPv6 address. `family` is either the integer `4` or `6` and denotes the family of `address` (not necessarily the value initially passed to `lookup`).

With the `all` option set to `true`, the arguments change to `(err, addresses)`, with `addresses` being an array of objects with the properties `address` and `family`.

On error, `err` is an [Error](#) object, where `err.code` is the error code. Keep in mind that `err.code` will be set to `'ENOENT'` not only when the hostname does not exist but also when the lookup fails in other ways such as no available file descriptors.

`dns.lookup()` does not necessarily have anything to do with the DNS protocol. The implementation uses an operating system facility that can associate names with addresses, and vice versa. This implementation can have subtle but important consequences on the behavior of any Node.js program. Please take some time to consult the [Implementation considerations section](#) before using `dns.lookup()`.

Supported getaddrinfo flags

The following flags can be passed as hints to `dns.lookup()`.

- `dns.ADDRCONFIG` : Returned address types are determined by the types of addresses supported by the current system. For example, IPv4 addresses are only returned if the current system has at least one IPv4 address configured. Loopback addresses are not considered.
- `dns.V4MAPPED` : If the IPv6 family was specified, but no IPv6 addresses were found, then return IPv4 mapped IPv6 addresses. Note that it is not supported on some operating systems (e.g FreeBSD 10.1).

`dns.lookupService(address, port, callback)`

Added in: v0.11.14

Resolves the given `address` and `port` into a hostname and service using the operating system's underlying `getnameinfo` implementation.

If `address` is not a valid IP address, a `TypeError` will be thrown. The `port` will be coerced to a number. If it is not a legal port, a `TypeError` will be thrown.

The callback has arguments `(err, hostname, service)`. The `hostname` and `service` arguments are strings (e.g. `'localhost'` and `'http'` respectively).

On error, `err` is an `Error` object, where `err.code` is the error code.

```
const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
  console.log(hostname, service);
  // Prints: localhost ssh
});
```

dns.resolve(hostname[, rrtype], callback)

#

Added in: v0.1.27

Uses the DNS protocol to resolve a hostname (e.g. `'nodejs.org'`) into an array of the record types specified by `rrtype`.

Valid values for `rrtype` are:

- `'A'` - IPV4 addresses, default
- `'AAAA'` - IPV6 addresses
- `'MX'` - mail exchange records
- `'TXT'` - text records
- `'SRV'` - SRV records
- `'PTR'` - PTR records
- `'NS'` - name server records
- `'CNAME'` - canonical name records
- `'SOA'` - start of authority record
- `'NAPTR'` - name authority pointer record

The `callback` function has arguments `(err, addresses)`. When successful, `addresses` will be an array. The type of each item in `addresses` is determined by the record type, and described in the documentation for the corresponding lookup methods.

On error, `err` is an `Error` object, where `err.code` is one of the error codes listed [here](#).

dns.resolve4(hostname, callback)

#

Added in: v0.1.16

Uses the DNS protocol to resolve a IPv4 addresses (`A` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).

dns.resolve6(hostname, callback)

#

Added in: v0.1.16

Uses the DNS protocol to resolve a IPv6 addresses (`AAAA` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of IPv6 addresses.

dns.resolveCname(hostname, callback)

#

Added in: v0.3.2

Uses the DNS protocol to resolve `CNAME` records for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of canonical name records available for the `hostname` (e.g. `['bar.example.com']`).

dns.resolveMx(hostname, callback)

#

Added in: v0.1.27

Uses the DNS protocol to resolve mail exchange records (`MX` records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of objects containing both a `priority` and `exchange` property (e.g. `[{priority: 10, exchange: 'mx.example.com'}, ...]`).

dns.resolveNaptr(hostname, callback)

#

Uses the DNS protocol to resolve regular expression based records (NAPTR records) for the `hostname`. The `callback` function has arguments `(err, addresses)`. The `addresses` argument passed to the `callback` function will contain an array of objects with the following properties:

- `flags`
- `service`
- `regexp`
- `replacement`
- `order`
- `preference`

For example:

```
{
  flags: 's',
  service: 'SIP+D2U',
  regexp: '',
  replacement: '_sip._udp.example.com',
  order: 30,
  preference: 100
}
```

dns.resolveNs(hostname, callback)

#

Added in: v0.1.90

Uses the DNS protocol to resolve name server records (NS records) for the `hostname`. The `addresses` argument passed to the `callback` function will contain an array of name server records available for `hostname` (e.g., `['ns1.example.com', 'ns2.example.com']`).

dns.resolveSoa(hostname, callback)

Added in: v0.11.10

Uses the DNS protocol to resolve a start of authority record (SOA record) for the `hostname`. The `addresses` argument passed to the `callback` function will be an object with the following properties:

- `nsname`
- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{  
    nsname: 'ns.example.com',  
    hostmaster: 'root.example.com',  
    serial: 2013101809,  
    refresh: 10000,  
    retry: 2400,  
    expire: 604800,  
    minttl: 3600  
}
```

dns.resolveSrv(hostname, callback)

Added in: v0.1.27

Uses the DNS protocol to resolve service records (SRV records) for the `hostname`. The `addresses` argument passed to the `callback` function will be an array of objects with the following properties:

- priority
- weight
- port
- name

```
{  
  priority: 10,  
  weight: 5,  
  port: 21223,  
  name: 'service.example.com'  
}
```

dns.resolvePtr(hostname, callback)

Added in: v6.0.0

Uses the DNS protocol to resolve pointer records (PTR records) for the hostname . The addresses argument passed to the callback function will be an array of strings containing the reply records.

dns.resolveTxt(hostname, callback)

Added in: v0.1.27

Uses the DNS protocol to resolve text queries (TXT records) for the hostname . The addresses argument passed to the callback function is a two-dimentional array of the text records available for hostname (e.g., [['v=spf1 ip4:0.0.0.0', '~all']]). Each sub-array contains TXT chunks of one record. Depending on the use case, these could be either joined together or treated separately.

dns.reverse(ip, callback)

Added in: v0.1.16

Performs a reverse DNS query that resolves an IPv4 or IPv6 address to an array of

hostnames.

The `callback` function has arguments `(err, hostnames)`, where `hostnames` is an array of resolved hostnames for the given `ip`.

On error, `err` is an [Error](#) object, where `err.code` is one of the [DNS error codes](#).

dns.setServers(servers)

Added in: v0.11.3

Sets the IP addresses of the servers to be used when resolving. The `servers` argument is an array of IPv4 or IPv6 addresses.

If a port specified on the address it will be removed.

An error will be thrown if an invalid address is provided.

The `dns.setServers()` method must not be called while a DNS query is in progress.

Error codes

Each DNS query can return one of the following error codes:

- `dns.NODATA` : DNS server returned answer with no data.
- `dns.FORMERR` : DNS server claims query was misformatted.
- `dns.SERVFAIL` : DNS server returned general failure.
- `dns.NOTFOUND` : Domain name not found.
- `dns.NOTIMP` : DNS server does not implement requested operation.
- `dns.REFUSED` : DNS server refused query.
- `dns.BADQUERY` : Misformatted DNS query.
- `dns.BADNAME` : Misformatted hostname.
- `dns.BADFAMILY` : Unsupported address family.
- `dns.BADRESP` : Misformatted DNS reply.

- `dns.CONNREFUSED` : Could not contact DNS servers.
- `dns.TIMEOUT` : Timeout while contacting DNS servers.
- `dns.EOF` : End of file.
- `dns.FILE` : Error reading file.
- `dns.NOMEM` : Out of memory.
- `dns.DESTRATION` : Channel is being destroyed.
- `dns.BADSTR` : Misformatted string.
- `dns.BADFLAGS` : Illegal flags specified.
- `dns.NONAME` : Given hostname is not numeric.
- `dns.BADHINTS` : Illegal hints flags specified.
- `dns.NOTINITIALIZED` : c-ares library initialization not yet performed.
- `dns.LOADIPHLPAPI` : Error loading iphlpapi.dll.
- `dns.ADDRGETNETWORKPARAMS` : Could not find GetNetworkParams function.
- `dns.CANCELLED` : DNS query cancelled.

Implementation considerations

Although `dns.lookup()` and the various `dns.resolve*()/dns.reverse()` functions have the same goal of associating a network name with a network address (or vice versa), their behavior is quite different. These differences can have subtle but significant consequences on the behavior of Node.js programs.

`dns.lookup()`

Under the hood, `dns.lookup()` uses the same operating system facilities as most other programs. For instance, `dns.lookup()` will almost always resolve a given name the same way as the `ping` command. On most POSIX-like operating systems, the behavior of the `dns.lookup()` function can be modified by changing settings in `nsswitch.conf(5)` and/or `resolv.conf(5)`, but note that changing these files will change the behavior of *all other programs running on the same operating system*.

Though the call to `dns.lookup()` will be asynchronous from JavaScript's

perspective, it is implemented as a synchronous call to `getaddrinfo(3)` that runs on libuv's threadpool. Because libuv's threadpool has a fixed size, it means that if for whatever reason the call to `getaddrinfo(3)` takes a long time, other operations that could run on libuv's threadpool (such as filesystem operations) will experience degraded performance. In order to mitigate this issue, one potential solution is to increase the size of libuv's threadpool by setting the `'UV_THREADPOOL_SIZE'` environment variable to a value greater than `4` (its current default value). For more information on libuv's threadpool, see [the official libuv documentation](#).

`dns.resolve()`, `dns.resolve*`(`)` and `dns.reverse()`

#

These functions are implemented quite differently than `dns.lookup()`. They do not use `getaddrinfo(3)` and they *always* perform a DNS query on the network. This network communication is always done asynchronously, and does not use libuv's threadpool.

As a result, these functions cannot have the same negative impact on other processing that happens on libuv's threadpool that `dns.lookup()` can have.

They do not use the same set of configuration files than what `dns.lookup()` uses. For instance, *they do not use the configuration from `/etc/hosts`*.

Domain

#

Stability: 0 - Deprecated

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most end users should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an `'error'`

event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the `process.on('uncaughtException')` handler, or causing the program to exit immediately with an error code.

Warning: Don't Ignore Errors!

Domain error handlers are not a substitute for closing down your process when an error occurs.

By the very nature of how `throw` works in JavaScript, there is almost never any way to safely "pick up where you left off", without leaking references, or creating some other sort of undefined brittle state.

The safest way to respond to a thrown error is to shut down the process. Of course, in a normal web server, you might have many connections open, and it is not reasonable to abruptly shut those down because an error was triggered by someone else.

The better approach is to send an error response to the request that triggered the error, while letting the others finish in their normal time, and stop listening for new requests in that worker.

In this way, `domain` usage goes hand-in-hand with the cluster module, since the master process can fork a new worker when a worker encounters an error. For Node.js programs that scale to multiple machines, the terminating proxy or service registry can take note of the failure, and react accordingly.

For example, this is not a good idea:

```
// XXX WARNING!  BAD IDEA!

var d = require('domain').create();
d.on('error', (er) => {
  // The error won't crash the process, but what it does is worse!
  // Though we've prevented abrupt process restarting, we are leaking
```

```
// resources like crazy if this ever happens.  
// This is no better than process.on('uncaughtException')!  
console.log('error, but oh well', er.message);  
});  
d.run(() => {  
  require('http').createServer((req, res) => {  
    handleRequest(req, res);  
  }).listen(PORT);  
});
```

By using the context of a domain, and the resilience of separating our program into multiple worker processes, we can react more appropriately, and handle errors with much greater safety.

```
// Much better!  
  
const cluster = require('cluster');  
const PORT = +process.env.PORT || 1337;  
  
if (cluster.isMaster) {  
  // In real life, you'd probably use more than just 2 workers,  
  // and perhaps not put the master and worker in the same file.  
  //  
  // You can also of course get a bit fancier about logging, and  
  // implement whatever custom logic you need to prevent DoS  
  // attacks and other bad behavior.  
  //  
  // See the options in the cluster documentation.  
  //  
  // The important thing is that the master does very little,  
  // increasing our resilience to unexpected errors.  
  
  cluster.fork();
```

```
cluster.fork();

cluster.on('disconnect', (worker) => {
    console.error('disconnect!');
    cluster.fork();
});

} else {
    // the worker
    //
    // This is where we put our bugs!

const domain = require('domain');

// See the cluster documentation for more details about using
// worker processes to serve requests. How it works, caveats, etc.

const server = require('http').createServer((req, res) => {
    var d = domain.create();
    d.on('error', (er) => {
        console.error('error', er.stack);

        // Note: we're in dangerous territory!
        // By definition, something unexpected occurred,
        // which we probably didn't want.
        // Anything can happen now! Be very careful!

    try {
        // make sure we close down within 30 seconds
        var killtimer = setTimeout(() => {
            process.exit(1);
        }, 3000);
        // But don't keep the process open just for that!
    }
});
```

```
killtimer.unref();

    // stop taking new requests.
    server.close();

    // Let the master know we're dead. This will trigger a
    // 'disconnect' in the cluster master, and then it will fork
    // a new worker.
    cluster.worker.disconnect();

    // try to send an error to the request that triggered the problem
    res.statusCode = 500;
    res.setHeader('content-type', 'text/plain');
    res.end('Oops, there was a problem!\n');

} catch (er2) {
    // oh well, not much we can do at this point.
    console.error('Error sending 500!', er2.stack);
}

});

// Because req and res were created before this domain existed,
// we need to explicitly add them.
// See the explanation of implicit vs explicit binding below.
d.add(req);
d.add(res);

// Now run the handler function in the domain.
d.run(() => {
    handleRequest(req, res);
});

server.listen(PORT);

}
```

```
// This part isn't important. Just an example routing thing.  
// You'd put your fancy application logic here.  
  
function handleRequest(req, res) {  
  switch(req.url) {  
    case '/error':  
      // We do some async stuff, and then...  
      setTimeout(() => {  
        // Whoops!  
        flerb.bark();  
      });  
      break;  
    default:  
      res.end('ok');  
  }  
}
```

Additions to Error objects

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

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

Implicit Binding

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

their creation.

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

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

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

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

Explicit Binding

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

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

That is possible via explicit binding.

For example:

```
// create a top-level domain for the server
const domain = require('domain');
const http = require('http');
const serverDomain = domain.create();
```

```
serverDomain.run(() => {
  // server is created in the scope of serverDomain
  http.createServer((req, res) => {
    // req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    var reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', (er) => {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
      } catch (er) {
        console.error('Error sending 500', er, req.url);
      }
    });
  }).listen(1337);
});
```

domain.create()

#

- return: <Domain>

Returns a new Domain object.

Class: Domain

#

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

Domain is a child class of [EventEmitter](#). To handle the errors that it catches, listen

to its 'error' event.

domain.run(fn[, arg][, ...])

#

- `fn` <Function>

Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowlevel requests that are created in that context. Optionally, arguments can be passed to the function.

This is the most basic way to use a domain.

Example:

```
const domain = require('domain');
const fs = require('fs');
const d = domain.create();
d.on('error', (er) => {
  console.error('Caught error!', er);
});
d.run(() => {
  process.nextTick(() => {
    setTimeout(() => { // simulating some various async stuff
      fs.open('non-existent file', 'r', (er, fd) => {
        if (er) throw er;
        // proceed...
      });
    }, 100);
  });
});
```

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

domain.members

#

- <Array>

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

domain.add(emitter)

- `emitter` <EventEmitter> | <Timer> emitter or timer to be added to the domain

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

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

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

domain.remove(emitter)

- `emitter` <EventEmitter> | <Timer> emitter or timer to be removed from the domain

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

domain.bind(callback)

- `callback` <Function> The callback function
- `return: <Function>` The bound function

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

Example

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

```
function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.bind((er, data) => {
    // if this throws, it will also be passed to the domain
    return cb(er, data ? JSON.parse(data) : null);
  }));
}

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

domain.intercept(callback)

#

- `callback <Function>` The callback function
- `return: <Function>` The intercepted function

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

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

Example

#

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

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.intercept((data) => {
    // note, the first argument is never passed to the
    // callback since it is assumed to be the 'Error' argument
  }));
}
```

```
// and thus intercepted by the domain.

// if this throws, it will also be passed to the domain
// so the error-handling logic can be moved to the 'error'
// event on the domain instead of being repeated throughout
// the program.

return cb(null, JSON.parse(data));

});

}

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

});
```

domain.enter()

The `enter` method is plumbing used by the `run`, `bind`, and `intercept` methods to set the active domain. It sets `domain.active` and `process.domain` to the domain, and implicitly pushes the domain onto the domain stack managed by the domain module (see `domain.exit()` for details on the domain stack). The call to `enter` delimits the beginning of a chain of asynchronous calls and I/O operations bound to a domain.

Calling `enter` changes only the active domain, and does not alter the domain itself. `enter` and `exit` can be called an arbitrary number of times on a single domain.

If the domain on which `enter` is called has been disposed, `enter` will return without setting the domain.

domain.exit()

The `exit` method exits the current domain, popping it off the domain stack. Any time execution is going to switch to the context of a different chain of asynchronous

calls, it's important to ensure that the current domain is exited. The call to `exit` delimits either the end of or an interruption to the chain of asynchronous calls and I/O operations bound to a domain.

If there are multiple, nested domains bound to the current execution context, `exit` will exit any domains nested within this domain.

Calling `exit` changes only the active domain, and does not alter the domain itself. `enter` and `exit` can be called an arbitrary number of times on a single domain.

If the domain on which `exit` is called has been disposed, `exit` will return without exiting the domain.

domain.dispose()

#

Stability: 0 - Deprecated. Please recover from failed IO actions explicitly via error event handlers set on the domain.

Once `dispose` has been called, the domain will no longer be used by callbacks bound into the domain via `run`, `bind`, or `intercept`, and a '`'dispose'`' event is emitted.

Errors

#

Applications running in Node.js will generally experience four categories of errors:

- Standard JavaScript errors such as:
 - `<EvalError>` : thrown when a call to `eval()` fails.
 - `<SyntaxError>` : thrown in response to improper JavaScript language syntax.
 - `<RangeError>` : thrown when a value is not within an expected range
 - `<ReferenceError>` : thrown when using undefined variables
 - `<TypeError>` : thrown when passing arguments of the wrong type
 - `<URIError>` : thrown when a global URI handling function is misused.

- System errors triggered by underlying operating system constraints such as attempting to open a file that does not exist, attempting to send data over a closed socket, etc;
- And User-specified errors triggered by application code.
- Assertion Errors are a special class of error that can be triggered whenever Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

All JavaScript and System errors raised by Node.js inherit from, or are instances of, the standard JavaScript `<Error>` class and are guaranteed to provide *at least* the properties available on that class.

Error Propagation and Interception

Node.js supports several mechanisms for propagating and handling errors that occur while an application is running. How these errors are reported and handled depends entirely on the type of Error and the style of the API that is called.

All JavaScript errors are handled as exceptions that *immediately* generate and throw an error using the standard JavaScript `throw` mechanism. These are handled using the `try / catch` construct provided by the JavaScript language.

```
// Throws with a ReferenceError because z is undefined
try {
  const m = 1;
  const n = m + z;
} catch (err) {
  // Handle the error here.
}
```

Any use of the JavaScript `throw` mechanism will raise an exception that *must* be handled using `try / catch` or the Node.js process will exit immediately.

With few exceptions, *Synchronous APIs* (any blocking method that does not accept a `callback` function, such as `fs.readFileSync`), will use `throw` to report errors.

Errors that occur within Asynchronous APIs may be reported in multiple ways:

- Most asynchronous methods that accept a `callback` function will accept an `Error` object passed as the first argument to that function. If that first argument is not `null` and is an instance of `Error`, then an error occurred that should be handled.

```
const fs = require('fs');
fs.readFile('a file that does not exist', (err, data) => {
  if (err) {
    console.error('There was an error reading the file!', e)
    return;
  }
  // Otherwise handle the data
});
```

- When an asynchronous method is called on an object that is an `EventEmitter`, errors can be routed to that object's `'error'` event.

```
const net = require('net');
const connection = net.connect('localhost');

// Adding an 'error' event handler to a stream:
connection.on('error', (err) => {
  // If the connection is reset by the server, or if it can
  // connect at all, or on any sort of error encountered by
  // the connection, the error will be sent here.
  console.error(err);
});

connection.pipe(process.stdout);
```

- A handful of typically asynchronous methods in the Node.js API may still use the `throw` mechanism to raise exceptions that must be handled using `try / catch`. There is no comprehensive list of such methods; please refer to the documentation of each method to determine the appropriate error handling mechanism required.

The use of the '`'error'`' event mechanism is most common for **stream-based** and **event emitter-based** APIs, which themselves represent a series of asynchronous operations over time (as opposed to a single operation that may pass or fail).

For *all* `EventEmitter` objects, if an '`'error'`' event handler is not provided, the error will be thrown, causing the Node.js process to report an unhandled exception and crash unless either: The `domain` module is used appropriately or a handler has been registered for the `process.on('uncaughtException')` event.

```
const EventEmitter = require('events');
const ee = new EventEmitter();

setImmediate(() => {
  // This will crash the process because no 'error' event
  // handler has been added.
  ee.emit('error', new Error('This will crash'));
});
```

Errors generated in this way *cannot* be intercepted using `try / catch` as they are thrown *after* the calling code has already exited.

Developers must refer to the documentation for each method to determine exactly how errors raised by those methods are propagated.

Node.js style callbacks

#

Most asynchronous methods exposed by the Node.js core API follow an idiomatic pattern referred to as a "Node.js style callback". With this pattern, a callback function is passed to the method as an argument. When the operation either

completes or an error is raised, the callback function is called with the Error object (if any) passed as the first argument. If no error was raised, the first argument will be passed as `null`.

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

function nodeStyleCallback(err, data) {
  if (err) {
    console.error('There was an error', err);
    return;
  }
  console.log(data);
}

fs.readFile('/some/file/that/does-not-exist', nodeStyleCallback);
fs.readFile('/some/file/that/does-exist', nodeStyleCallback)
```

The JavaScript `try / catch` mechanism **cannot** be used to intercept errors generated by asynchronous APIs. A common mistake for beginners is to try to use `throw` inside a Node.js style callback:

```
// THIS WILL NOT WORK:
const fs = require('fs');

try {
  fs.readFile('/some/file/that/does-not-exist', (err, data) => {
    // mistaken assumption: throwing here...
    if (err) {
      throw err;
    }
  });
} catch(err) {
```

```
// This will not catch the throw!
console.log(err);
}
```

This will not work because the callback function passed to `fs.readFile()` is called asynchronously. By the time the callback has been called, the surrounding code (including the `try {} catch(err) {}` block) will have already exited. Throwing an error inside the callback can crash the Node.js process in most cases. If `domains` are enabled, or a handler has been registered with `process.on('uncaughtException')`, such errors can be intercepted.

Class: Error

A generic JavaScript `Error` object that does not denote any specific circumstance of why the error occurred. `Error` objects capture a "stack trace" detailing the point in the code at which the `Error` was instantiated, and may provide a text description of the error.

All errors generated by Node.js, including all System and JavaScript errors, will either be instances of, or inherit from, the `Error` class.

`new Error(message)`

Creates a new `Error` object and sets the `error.message` property to the provided text message. If an object is passed as `message`, the text message is generated by calling `message.toString()`. The `error.stack` property will represent the point in the code at which `new Error()` was called. Stack traces are dependent on [V8's stack trace API](#). Stack traces extend only to either (a) the beginning of *synchronous code execution*, or (b) the number of frames given by the property `Error.stackTraceLimit`, whichever is smaller.

`Error.captureStackTrace(targetObject[, constructorOpt])`

Creates a `.stack` property on `targetObject`, which when accessed returns a string representing the location in the code at which

Error.captureStackTrace() was called.

```
const myObject = {};
Error.captureStackTrace(myObject);
myObject.stack // similar to `new Error().stack`
```

The first line of the trace, instead of being prefixed with `ErrorType: message`, will be the result of calling `targetObject.toString()`.

The optional `constructorOpt` argument accepts a function. If given, all frames above `constructorOpt`, including `constructorOpt`, will be omitted from the generated stack trace.

The `constructorOpt` argument is useful for hiding implementation details of error generation from an end user. For instance:

```
function MyError() {
  Error.captureStackTrace(this, MyError);
}

// Without passing MyError to captureStackTrace, the MyError
// frame would show up in the .stack property. By passing
// the constructor, we omit that frame and all frames above it.
new MyError().stack
```

Error.stackTraceLimit

The `Error.stackTraceLimit` property specifies the number of stack frames collected by a stack trace (whether generated by `new Error().stack` or `Error.captureStackTrace(obj)`).

The default value is `10` but may be set to any valid JavaScript number. Changes will affect any stack trace captured *after* the value has been changed.

If set to a non-number value, or set to a negative number, stack traces will not capture any frames.

error.message

Returns the string description of error as set by calling `new Error(message)`. The `message` passed to the constructor will also appear in the first line of the stack trace of the `Error`, however changing this property after the `Error` object is created *may not* change the first line of the stack trace.

```
const err = new Error('The message');
console.log(err.message);
// Prints: The message
```

error.stack

Returns a string describing the point in the code at which the `Error` was instantiated.

For example:

```
Error: Things keep happening!
at /home/gbusey/file.js:525:2
at Frobnicator.refrobulate (/home/gbusey/business-logic.js:424:21)
at Actor.<anonymous> (/home/gbusey/actors.js:400:8)
at increaseSynergy (/home/gbusey/actors.js:701:6)
```

The first line is formatted as `<error class name>: <error message>`, and is followed by a series of stack frames (each line beginning with "at"). Each frame describes a call site within the code that lead to the error being generated. V8 attempts to display a name for each function (by variable name, function name, or object method name), but occasionally it will not be able to find a suitable name. If V8 cannot determine a name for the function, only location information will be displayed for that frame. Otherwise, the determined function name will be displayed

with location information appended in parentheses.

It is important to note that frames are **only** generated for JavaScript functions. If, for example, execution synchronously passes through a C++ addon function called `cheetahify`, which itself calls a JavaScript function, the frame representing the `cheetahify` call will **not** be present in the stack traces:

```
const cheetahify = require('./native-binding.node');

function makeFaster() {
    // cheetahify *synchronously* calls speedy.
    cheetahify(function speedy() {
        throw new Error('oh no!');
    });
}

makeFaster(); // will throw:
// /home/gbusey/file.js:6
//     throw new Error('oh no!');
//           ^
// Error: oh no!
//     at speedy (/home/gbusey/file.js:6:11)
//     at makeFaster (/home/gbusey/file.js:5:3)
//     at Object.<anonymous> (/home/gbusey/file.js:10:1)
//     at Module._compile (module.js:456:26)
//     at Object.Module._extensions..js (module.js:474:10)
//     at Module.load (module.js:356:32)
//     at Function.Module._load (module.js:312:12)
//     at Function.Module.runMain (module.js:497:10)
//     at startup (node.js:119:16)
//     at node.js:906:3
```

The location information will be one of:

- `native`, if the frame represents a call internal to V8 (as in `[] .forEach`).
- `plain-filename.js:line:column`, if the frame represents a call internal to Node.js.
- `/absolute/path/to/file.js:line:column`, if the frame represents a call in a user program, or its dependencies.

The string representing the stack trace is lazily generated when the `error.stack` property is accessed.

The number of frames captured by the stack trace is bounded by the smaller of `Error.stackTraceLimit` or the number of available frames on the current event loop tick.

System-level errors are generated as augmented `Error` instances, which are detailed [here](#).

Class: RangeError

A subclass of `Error` that indicates that a provided argument was not within the set or range of acceptable values for a function; whether that is a numeric range, or outside the set of options for a given function parameter.

For example:

```
require('net').connect(-1);
// throws RangeError, port should be > 0 && < 65536
```

Node.js will generate and throw `RangeError` instances *immediately* as a form of argument validation.

Class: ReferenceError

A subclass of `Error` that indicates that an attempt is being made to access a variable that is not defined. Such errors commonly indicate typos in code, or an otherwise broken program.

While client code may generate and propagate these errors, in practice, only V8 will do so.

```
doesNotExist;  
// throws ReferenceError, doesNotExist is not a variable in this program
```

`ReferenceError` instances will have an `error.arguments` property whose value is an array containing a single element: a string representing the variable that was not defined.

```
const assert = require('assert');  
try {  
  doesNotExist;  
} catch(err) {  
  assert(err.arguments[0], 'doesNotExist');  
}
```

Unless an application is dynamically generating and running code, `ReferenceError` instances should always be considered a bug in the code or its dependencies.

Class: SyntaxError

A subclass of `Error` that indicates that a program is not valid JavaScript. These errors may only be generated and propagated as a result of code evaluation. Code evaluation may happen as a result of `eval`, `Function`, `require`, or `vm`. These errors are almost always indicative of a broken program.

```
try {  
  require('vm').runInThisContext('binary ! isNotOk');  
} catch(err) {  
  // err will be a SyntaxError
```

```
}
```

`SyntaxError` instances are unrecoverable in the context that created them – they may only be caught by other contexts.

Class: TypeError

#

A subclass of `Error` that indicates that a provided argument is not an allowable type. For example, passing a function to a parameter which expects a string would be considered a `TypeError`.

```
require('url').parse() => { };  
// throws TypeError, since it expected a string
```

Node.js will generate and throw `TypeError` instances *immediately* as a form of argument validation.

Exceptions vs. Errors

#

A JavaScript exception is a value that is thrown as a result of an invalid operation or as the target of a `throw` statement. While it is not required that these values are instances of `Error` or classes which inherit from `Error`, all exceptions thrown by Node.js or the JavaScript runtime *will* be instances of `Error`.

Some exceptions are *unrecoverable* at the JavaScript layer. Such exceptions will *always* cause the Node.js process to crash. Examples include `assert()` checks or `abort()` calls in the C++ layer.

System Errors

#

System errors are generated when exceptions occur within the program's runtime environment. Typically, these are operational errors that occur when an application violates an operating system constraint such as attempting to read a file that does not exist or when the user does not have sufficient permissions.

System errors are typically generated at the syscall level: an exhaustive list of error codes and their meanings is available by running `man 2 intro` or `man 3 errno` on most Unices; or [online](#).

In Node.js, system errors are represented as augmented `Error` objects with added properties.

Class: System Error

`error.code` #

`error.errno` #

Returns a string representing the error code, which is always `E` followed by a sequence of capital letters, and may be referenced in `man 2 intro`.

The properties `error.code` and `error.errno` are aliases of one another and return the same value.

`error.syscall` #

Returns a string describing the [syscall](#) that failed.

Common System Errors

This list is **not exhaustive**, but enumerates many of the common system errors encountered when writing a Node.js program. An exhaustive list may be found [here](#).

- `EACCES` (Permission denied): An attempt was made to access a file in a way forbidden by its file access permissions.
- `EADDRINUSE` (Address already in use): An attempt to bind a server ([net](#), [http](#), or [https](#)) to a local address failed due to another server on the local system already occupying that address.
- `ECONNREFUSED` (Connection refused): No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

- **ECONNRESET** (Connection reset by peer): A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or reboot. Commonly encountered via the `http` and `net` modules.
- **EEXIST** (File exists): An existing file was the target of an operation that required that the target not exist.
- **EISDIR** (Is a directory): An operation expected a file, but the given pathname was a directory.
- **EMFILE** (Too many open files in system): Maximum number of `file descriptors` allowable on the system has been reached, and requests for another descriptor cannot be fulfilled until at least one has been closed. This is encountered when opening many files at once in parallel, especially on systems (in particular, OS X) where there is a low file descriptor limit for processes. To remedy a low limit, run `ulimit -n 2048` in the same shell that will run the Node.js process.
- **ENOENT** (No such file or directory): Commonly raised by `fs` operations to indicate that a component of the specified pathname does not exist -- no entity (file or directory) could be found by the given path.
- **ENOTDIR** (Not a directory): A component of the given pathname existed, but was not a directory as expected. Commonly raised by `fs.readdir`.
- **ENOTEMPTY** (Directory not empty): A directory with entries was the target of an operation that requires an empty directory -- usually `fs.unlink`.
- **EPERM** (Operation not permitted): An attempt was made to perform an operation that requires elevated privileges.
- **EPIPE** (Broken pipe): A write on a pipe, socket, or FIFO for which there is no process to read the data. Commonly encountered at the `net` and `http` layers, indicative that the remote side of the stream being written to has been closed.
- **ETIMEDOUT** (Operation timed out): A connect or send request failed because

the connected party did not properly respond after a period of time. Usually encountered by `http` or `net` -- often a sign that a `socket.end()` was not properly called.

Events

#

Stability: 2 - Stable

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called.

For instance: a `net.Server` object emits an event each time a peer connects to it; a `fs.ReadStream` emits an event when the file is opened; a `stream` emits an event whenever data is available to be read.

All objects that emit events are instances of the `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more Functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

When the `EventEmitter` object emits an event, all of the Functions attached to that specific event are called *synchronously*. Any values returned by the called listeners are *ignored* and will be discarded.

The following example shows a simple `EventEmitter` instance with a single listener. The `eventEmitter.on()` method is used to register listeners, while the `eventEmitter.emit()` method is used to trigger the event.

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}
```

```
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

Passing arguments and `this` to listeners

#

The `eventEmitter.emit()` method allows an arbitrary set of arguments to be passed to the listener functions. It is important to keep in mind that when an ordinary listener function is called by the `EventEmitter`, the standard `this` keyword is intentionally set to reference the `EventEmitter` to which the listener is attached.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this);
  // Prints:
  //   a b MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined }
});
myEmitter.emit('event', 'a', 'b');
```

It is possible to use ES6 Arrow Functions as listeners, however, when doing so, the `this` keyword will no longer reference the `EventEmitter` instance:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  console.log(a, b, this);
```

```
// Prints: a b {}  
});  
myEmitter.emit('event', 'a', 'b');
```

Asynchronous vs. Synchronous

#

The `EventEmitter` calls all listeners synchronously in the order in which they were registered. This is important to ensure the proper sequencing of events and to avoid race conditions or logic errors. When appropriate, listener functions can switch to an asynchronous mode of operation using the `setImmediate()` or `process.nextTick()` methods:

```
const myEmitter = new MyEmitter();  
myEmitter.on('event', (a, b) => {  
  setImmediate(() => {  
    console.log('this happens asynchronously');  
  });  
});  
myEmitter.emit('event', 'a', 'b');
```

Handling events only once

#

When a listener is registered using the `eventEmitter.on()` method, that listener will be invoked *every time* the named event is emitted.

```
const myEmitter = new MyEmitter();  
var m = 0;  
myEmitter.on('event', () => {  
  console.log(++m);  
});  
myEmitter.emit('event');  
// Prints: 1
```

```
myEmitter.emit('event');
// Prints: 2
```

Using the `eventEmitter.once()` method, it is possible to register a listener that is unregistered before it is called.

```
const myEmitter = new MyEmitter();
var m = 0;
myEmitter.once('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
myEmitter.emit('event');
// Ignored
```

Error events

When an error occurs within an `EventEmitter` instance, the typical action is for an `'error'` event to be emitted. These are treated as a special case within Node.js.

If an `EventEmitter` does *not* have at least one listener registered for the `'error'` event, and an `'error'` event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.

```
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));
// Throws and crashes Node.js
```

To guard against crashing the Node.js process, developers can either register a listener for the `process.on('uncaughtException')` event or use the `domain` module (*Note, however, that the `domain` module has been deprecated*).

```
const myEmitter = new MyEmitter();

process.on('uncaughtException', (err) => {
  console.log('whoops! there was an error');
});

myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an error
```

As a best practice, developers should always register listeners for the `'error'` event:

```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
  console.log('whoops! there was an error');
});
myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an error
```

Class: EventEmitter

The `EventEmitter` class is defined and exposed by the `events` module:

```
const EventEmitter = require('events');
```

All EventEmitters emit the event `'newListener'` when new listeners are added and `'removeListener'` when a listener is removed.

Event: 'newListener'

- `eventName <String> | <Symbol>` The name of the event being listened for
- `listener <Function>` The event handler function

The `EventEmitter` instance will emit it's own '`'newListener'`' event before a listener is added to it's internal array of listeners.

Listeners registered for the '`'newListener'`' event will be passed the event name and a reference to the listener being added.

The fact that the event is triggered before adding the listener has a subtle but important side effect: any *additional* listeners registered to the same `name` *within* the '`'newListener'`' callback will be inserted *before* the listener that is in the process of being added.

```
const myEmitter = new MyEmitter();
// Only do this once so we don't loop forever
myEmitter.once('newListener', (event, listener) => {
  if (event === 'event') {
    // Insert a new listener in front
    myEmitter.on('event', () => {
      console.log('B');
    });
  }
});
myEmitter.on('event', () => {
  console.log('A');
});
myEmitter.emit('event');
// Prints:
//  B
//  A
```

- ## Event: `'removeListener'` #
- `eventName` `<String> | <Symbol>` The event name
 - `listener` `<Function>` The event handler function

The 'removeListener' event is emitted *after* a listener is removed.

EventEmitter.listenerCount(emitter, eventName)

#

Stability: 0 - Deprecated: Use [emitter.listenerCount\(\)](#) instead.

A class method that returns the number of listeners for the given `eventName` registered on the given `emitter`.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {});
myEmitter.on('event', () => {});
console.log(EventEmitter.listenerCount(myEmitter, 'event'));
// Prints: 2
```

EventEmitter.defaultMaxListeners

#

By default, a maximum of `10` listeners can be registered for any single event. This limit can be changed for individual `EventEmitter` instances using the `emitter.setMaxListeners(n)` method. To change the default for *all* `EventEmitter` instances, the `EventEmitter.defaultMaxListeners` property can be used.

Take caution when setting the `EventEmitter.defaultMaxListeners` because the change effects *all* `EventEmitter` instances, including those created before the change is made. However, calling `emitter.setMaxListeners(n)` still has precedence over `EventEmitter.defaultMaxListeners`.

Note that this is not a hard limit. The `EventEmitter` instance will allow more listeners to be added but will output a trace warning to stderr indicating that a possible `EventEmitter memory leak` has been detected. For any single `EventEmitter`, the `emitter.getMaxListeners()` and `emitter.setMaxListeners()` methods can be used to temporarily avoid this

warning:

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', () => {
  // do stuff
  emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1, 0));
});
```

emitter.addListener(eventName, listener) #

Alias for `emitter.on(eventName, listener)`.

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

Synchronously calls each of the listeners registered for the event named `eventName`, in the order they were registered, passing the supplied arguments to each.

Returns `true` if the event had listeners, `false` otherwise.

emitter.eventNames() #

Returns an array listing the events for which the emitter has registered listeners. The values in the array will be strings or Symbols.

```
const EventEmitter = require('events');
const myEE = new EventEmitter();
myEE.on('foo', () => {});
myEE.on('bar', () => {});

const sym = Symbol('symbol');
myEE.on(sym, () => {});

console.log(myEE.eventNames());
// Prints [ 'foo', 'bar', Symbol(symbol) ]
```

emitter.getMaxListeners()

#

Returns the current max listener value for the `EventEmitter` which is either set by `emitter.setMaxListeners(n)` or defaults to `EventEmitter.defaultMaxListeners`.

emitter.listenerCount(eventName)

#

- `eventName` <Value> The name of the event being listened for

Returns the number of listeners listening to the event named `eventName`.

emitter.listeners(eventName)

#

Returns a copy of the array of listeners for the event named `eventName`.

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

emitter.on(eventName, listener)

#

- `eventName` <string> | <Symbol> The name of the event.
- `listener` <Function> The callback function

Adds the `listener` function to the end of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

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

```
});
```

Returns a reference to the `EventEmitter` so calls can be chained.

By default, event listeners are invoked in the order they are added. The `emitter.prependListener()` method can be used as an alternative to add the event listener to the beginning of the listeners array.

```
const myEE = new EventEmitter();
myEE.on('foo', () => console.log('a'));
myEE.prependListener('foo', () => console.log('b'));
myEE.emit('foo');

// Prints:
//   b
//   a
```

emitter.once(eventName, listener)

- `eventName` `<string> | <Symbol>` The name of the event.
- `listener` `<Function>` The callback function

Adds a **one time** `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

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

Returns a reference to the `EventEmitter` so calls can be chained.

By default, event listeners are invoked in the order they are added. The `emitter.prependOnceListener()` method can be used as an alternative to add the event listener to the beginning of the listeners array.

```
const myEE = new EventEmitter();
myEE.once('foo', () => console.log('a'));
myEE.prependOnceListener('foo', () => console.log('b'));
myEE.emit('foo');

// Prints:
//   b
//   a
```

emitter.prependListener(eventName, listener)

- `eventName` `<string> | <Symbol>` The name of the event.
- `listener` `<Function>` The callback function

Adds the `listener` function to the *beginning* of the listeners array for the event named `eventName`. No checks are made to see if the `listener` has already been added. Multiple calls passing the same combination of `eventName` and `listener` will result in the `listener` being added, and called, multiple times.

```
server.prependListener('connection', (stream) => {
  console.log('someone connected!');
});
```

Returns a reference to the `EventEmitter` so calls can be chained.

emitter.prependOnceListener(eventName, listener)

- `eventName` `<string> | <Symbol>` The name of the event.
- `listener` `<Function>` The callback function

Adds a **one time** `listener` function for the event named `eventName` to the *beginning* of the listeners array. The next time `eventName` is triggered, this listener is removed, and then invoked.

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

Returns a reference to the `EventEmitter` so calls can be chained.

emitter.removeAllListeners([eventName])

Removes all listeners, or those of the specified `eventName`.

Note that it is bad practice to remove listeners added elsewhere in the code, particularly when the `EventEmitter` instance was created by some other component or module (e.g. sockets or file streams).

Returns a reference to the `EventEmitter` so calls can be chained.

emitter.removeListener(eventName, listener)

Removes the specified `listener` from the listener array for the event named `eventName`.

```
var callback = (stream) => {
  console.log('someone connected!');
};

server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener` will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified `eventName`, then `removeListener` must be called multiple times to remove each instance.

Note that once an event has been emitted, all listeners attached to it at the time of emitting will be called in order. This implies that any `removeListener()` or

`removeAllListeners()` calls *after* emitting and *before* the last listener finishes execution will not remove them from `emit()` in progress. Subsequent events will behave as expected.

```
const myEmitter = new MyEmitter();

var callbackA = () => {
    console.log('A');
    myEmitter.removeListener('event', callbackB);
};

var callbackB = () => {
    console.log('B');
};

myEmitter.on('event', callbackA);

myEmitter.on('event', callbackB);

// callbackA removes listener callbackB but it will still be called.
// Internal listener array at time of emit [callbackA, callbackB]
myEmitter.emit('event');

// Prints:
//   A
//   B

// callbackB is now removed.
// Internal listener array [callbackA]
myEmitter.emit('event');

// Prints:
//   A
```

Because listeners are managed using an internal array, calling this will change the

position indices of any listener registered *after* the listener being removed. This will not impact the order in which listeners are called, but it will mean that any copies of the listener array as returned by the `emitter.listeners()` method will need to be recreated.

Returns a reference to the `EventEmitter` so calls can be chained.

`emitter.setMaxListeners(n)`

#

By default EventEmitters will print a warning if more than `10` listeners are added for a particular event. This is a useful default that helps finding memory leaks. Obviously, not all events should be limited to just 10 listeners. The `emitter.setMaxListeners()` method allows the limit to be modified for this specific `EventEmitter` instance. The value can be set to `Infinity` (or `0`) to indicate an unlimited number of listeners.

Returns a reference to the `EventEmitter` so calls can be chained.

File System

#

Stability: 2 - Stable

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Here is an example of the asynchronous version:

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

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Here is the synchronous version:

```
const 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', (err) => {
  if (err) throw err;
  console.log('renamed complete');
});

fs.stat('/tmp/world', (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', (err) => {
  if (err) throw err;
```

```
fs.stat('/tmp/world', (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.

The relative path to a filename can be used. Remember, however, that this path will be relative to `process.cwd()`.

Most fs functions let you omit the callback argument. If you do, a default callback is used that rethrows errors. To get a trace to the original call site, set the `NODE_DEBUG` environment variable:

```
$ cat script.js
function bad() {
  require('fs').readFile('/');
}
bad();

$ env NODE_DEBUG=fs node script.js
fs.js:88
    throw backtrace;
^

Error: EISDIR: illegal operation on a directory, read
<stack trace.>
```

Buffer API

#

Added in: v6.0.0

`fs` functions support passing and receiving paths as both strings and Buffers. The latter is intended to make it possible to work with filesystems that allow for non-UTF-8 filenames. For most typical uses, working with paths as Buffers will be unnecessary, as the string API converts to and from UTF-8 automatically.

Note that on certain file systems (such as NTFS and HFS+) filenames will always be encoded as UTF-8. On such file systems, passing non-UTF-8 encoded Buffers to `fs` functions will not work as expected.

Class: `fs.FSWatcher`

#

Added in: v0.5.8

Objects returned from `fs.watch()` are of this type.

Event: 'change'

#

Added in: v0.5.8

- `event <String>` The type of fs change
- `filename <String> | <Buffer>` The filename that changed (if relevant/available)

Emitted when something changes in a watched directory or file. See more details in `fs.watch()`.

The `filename` argument may not be provided depending on operating system support. If `filename` is provided, it will be provided as a `Buffer` if `fs.watch()` is called with its `encoding` option set to `'buffer'`, otherwise `filename` will be a string.

```
fs.watch('./tmp', {encoding: 'buffer'}, (event, filename) => {
  if (filename)
    console.log(filename);
    // Prints: <Buffer ...>
});
```

Event: 'error'

#

Added in: v0.5.8

- `error <Error>`

Emitted when an error occurs.

watcher.close()

#

Added in: v0.5.8

Stop watching for changes on the given `fs.FSWatcher`.

Class: `fs.ReadStream`

#

Added in: v0.1.93

`ReadStream` is a `Readable Stream`.

Event: 'open'

#

Added in: v0.1.93

- `fd <Integer>` Integer file descriptor used by the `ReadStream`.

Emitted when the `ReadStream`'s file is opened.

Event: 'close'

#

Added in: v0.1.93

Emitted when the `ReadStream`'s underlying file descriptor has been closed using the `fs.close()` method.

readStream.path

#

Added in: v0.1.93

The path to the file the stream is reading from as specified in the first argument to `fs.createReadStream()`. If `path` is passed as a string, then `readStream.path` will be a string. If `path` is passed as a `Buffer`, then `readStream.path` will be a `Buffer`.

Class: fs.Stats

Added in: v0.1.21

Objects returned from `fs.stat()`, `fs.lstat()` and `fs.fstat()` and their synchronous counterparts are of this type.

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

For a regular file `util.inspect(stats)` would return a string very similar to this:

```
{  
  dev: 2114,  
  ino: 48064969,  
  mode: 33188,  
  nlink: 1,  
  uid: 85,  
  gid: 100,  
  rdev: 0,  
  size: 527,  
  blksize: 4096,  
  blocks: 8,  
  atime: Mon, 10 Oct 2011 23:24:11 GMT,  
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,  
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,  
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT  
}
```

Please note that `atime`, `mtime`, `birthtime`, and `ctime` are instances of [Date](#) object and to compare the values of these objects you should use appropriate methods. For most general uses `getTime()` will return the number of milliseconds elapsed since *1 January 1970 00:00:00 UTC* and this integer should be sufficient for any comparison, however there are additional methods which can be used for displaying fuzzy information. More details can be found in the [MDN JavaScript Reference](#) page.

Stat Time Values

#

The times in the stat object have the following semantics:

- `atime` "Access Time" - Time when file data last accessed. Changed by the `mknod(2)`, `utimes(2)`, and `read(2)` system calls.
- `mtime` "Modified Time" - Time when file data last modified. Changed by the `mknod(2)`, `utimes(2)`, and `write(2)` system calls.
- `ctime` "Change Time" - Time when file status was last changed (inode data modification). Changed by the `chmod(2)`, `chown(2)`, `link(2)`, `mknod(2)`, `rename(2)`, `unlink(2)`, `utimes(2)`, `read(2)`, and `write(2)` system calls.
- `birthtime` "Birth Time" - Time of file creation. Set once when the file is created. On filesystems where `birthtime` is not available, this field may instead hold either the `ctime` or `1970-01-01T00:00Z` (ie, unix epoch timestamp `0`). Note that this value may be greater than `atime` or `mtime` in this case. On Darwin and other FreeBSD variants, also set if the `atime` is explicitly set to an earlier value than the current `birthtime` using the `utimes(2)` system call.

Prior to Node v0.12, the `ctime` held the `birthtime` on Windows systems. Note that as of v0.12, `ctime` is not "creation time", and on Unix systems, it never was.

Class: `fs.WriteStream`

#

Added in: v0.1.93

`WriteStream` is a [Writable Stream](#).

Event: 'open'

#

Added in: v0.1.93

- `fd` <Integer> Integer file descriptor used by the WriteStream.

Emitted when the WriteStream's file is opened.

Event: 'close'

Added in: v0.1.93

Emitted when the `WriteStream`'s underlying file descriptor has been closed using the `fs.close()` method.

writeStream.bytesWritten

Added in: v0.4.7

The number of bytes written so far. Does not include data that is still queued for writing.

writeStream.path

Added in: v0.1.93

The path to the file the stream is writing to as specified in the first argument to `fs.createWriteStream()`. If `path` is passed as a string, then `writeStream.path` will be a string. If `path` is passed as a `Buffer`, then `writeStream.path` will be a `Buffer`.

fs.access(path[, mode], callback)

Added in: v0.11.15

- `path` <String> | <Buffer>
- `mode` <Integer>
- `callback` <Function>

Tests a user's permissions for the file or directory specified by `path`. The `mode` argument is an optional integer that specifies the accessibility checks to be performed. The following constants define the possible values of `mode`. It is possible to create a mask consisting of the bitwise OR of two or more values.

- `fs.constants.F_OK` - `path` is visible to the calling process. This is useful for

determining if a file exists, but says nothing about `rwx` permissions. Default if no `mode` is specified.

- `fs.constants.R_OK` - `path` can be read by the calling process.
- `fs.constants.W_OK` - `path` can be written by the calling process.
- `fs.constants.X_OK` - `path` can be executed by the calling process. This has no effect on Windows (will behave like `fs.constants.F_OK`).

The final argument, `callback`, is a callback function that is invoked with a possible error argument. If any of the accessibility checks fail, the error argument will be populated. The following example checks if the file `/etc/passwd` can be read and written by the current process.

```
fs.access('/etc/passwd', fs.constants.R_OK | fs.constants.W_OK, (err) =>
  console.log(err ? 'no access!' : 'can read/write');
});
```

fs.accessSync(path[, mode])

#

Added in: v0.11.15

- `path` `<String>` | `<Buffer>`
- `mode` `<Integer>`

Synchronous version of `fs.access()`. This throws if any accessibility checks fail, and does nothing otherwise.

fs.appendFile(file, data[, options], callback)

#

Added in: v0.6.7

- `file` `<String>` | `<Buffer>` | `<Number>` filename or file descriptor
- `data` `<String>` | `<Buffer>`
- `options` `<Object>` | `<String>`
 - `encoding` `<String>` | `<Null>` default = `'utf8'`
 - `mode` `<Integer>` default = `0o666`

- `flag` `<String>` default = `'a'`
- `callback` `<Function>`

Asynchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a buffer.

Example:

```
fs.appendFile('message.txt', 'data to append', (err) => {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

If `options` is a string, then it specifies the encoding. Example:

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback);
```

Any specified file descriptor has to have been opened for appending.

Note: Specified file descriptors will not be closed automatically.

fs.appendFileSync(file, data[, options])

Added in: v0.6.7

- `file` `<String> | <Buffer> | <Number>` filename or file descriptor
- `data` `<String> | <Buffer>`
- `options` `<Object> | <String>`
 - `encoding` `<String> | <Null>` default = `'utf8'`
 - `mode` `<Integer>` default = `0o666`
 - `flag` `<String>` default = `'a'`

The synchronous version of `fs.appendFile()`. Returns `undefined`.

fs.chmod(path, mode, callback)

Added in: v0.1.30

- `path` <String> | <Buffer>
- `mode` <Integer>
- `callback` <Function>

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

fs.chmodSync(path, mode)

Added in: v0.6.7

- `path` <String> | <Buffer>
- `mode` <Integer>

Synchronous `chmod(2)`. Returns `undefined`.

fs.chown(path, uid, gid, callback)

Added in: v0.1.97

- `path` <String> | <Buffer>
- `uid` <Integer>
- `gid` <Integer>
- `callback` <Function>

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

fs.chownSync(path, uid, gid)

Added in: v0.1.97

- `path` <String> | <Buffer>
- `uid` <Integer>
- `gid` <Integer>

Synchronous `chown(2)`. Returns `undefined`.

fs.close(fd, callback)

#

Added in: v0.0.2

- `fd` <Integer>
- `callback` <Function>

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

fs.closeSync(fd)

#

Added in: v0.1.21

- `fd` <Integer>

Synchronous `close(2)`. Returns `undefined`.

fs.constants

#

Returns an object containing commonly used constants for file system operations. The specific constants currently defined are described in [FS Constants](#).

fs.createReadStream(path[, options])

#

Added in: v0.1.31

- `path` <String> | <Buffer>
- `options` <String> | <Object>
 - `flags` <String>
 - `encoding` <String>
 - `fd` <Integer>
 - `mode` <Integer>
 - `autoClose` <Boolean>
 - `start` <Integer>

- `end <Integer>`

Returns a new `ReadStream` object. (See [Readable Stream](#)).

Be aware that, unlike the default value set for `highWaterMark` on a readable stream (16 kb), the stream returned by this method has a default value of 64 kb for the same parameter.

`options` is an object or string with the following defaults:

```
{  
  flags: 'r',  
  encoding: null,  
  fd: null,  
  mode: 0o666,  
  autoClose: true  
}
```

`options` can include `start` and `end` values to read a range of bytes from the file instead of the entire file. Both `start` and `end` are inclusive and start at 0. The `encoding` can be any one of those accepted by [Buffer](#).

If `fd` is specified, `ReadStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. Note that `fd` should be blocking; non-blocking `fd`s should be passed to [net.Socket](#).

If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is your responsibility to close it and make sure there's no file descriptor leak. If `autoClose` is set to true (default behavior), on `error` or `end` the file descriptor will be closed automatically.

`mode` sets the file mode (permission and sticky bits), but only if the file was created.

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

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

If `options` is a string, then it specifies the encoding.

fs.createWriteStream(path[, options])

Added in: v0.1.31

- `path` `<String> | <Buffer>`
- `options` `<String> | <Object>`
 - `flags` `<String>`
 - `defaultEncoding` `<String>`
 - `fd` `<Integer>`
 - `mode` `<Integer>`
 - `autoClose` `<Boolean>`
 - `start` `<Integer>`

Returns a new `WriteStream` object. (See [Writable Stream](#)).

`options` is an object or string with the following defaults:

```
{  
  flags: 'w',  
  defaultEncoding: 'utf8',  
  fd: null,  
  mode: 0o666,  
  autoClose: true  
}
```

`options` may also include a `start` option to allow writing data at some position past the beginning of the file. Modifying a file rather than replacing it may require a `flags` mode of `r+` rather than the default mode `w`. The `defaultEncoding` can be any one of those accepted by `Buffer`.

If `autoClose` is set to true (default behavior) on `error` or `end` the file descriptor will be closed automatically. If `autoClose` is false, then the file descriptor won't be closed, even if there's an error. It is your responsibility to close it and make sure there's no file descriptor leak.

Like `ReadStream`, if `fd` is specified, `WriteStream` will ignore the `path` argument and will use the specified file descriptor. This means that no `'open'` event will be emitted. Note that `fd` should be blocking; non-blocking `fd`s should be passed to `net.Socket`.

If `options` is a string, then it specifies the encoding.

fs.exists(path, callback)

#

Added in: v0.0.2 Deprecated since: v1.0.0

Stability: 0 - Deprecated: Use `fs.stat()` or `fs.access()` instead.

- `path` `<String> | <Buffer>`
- `callback` `<Function>`

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

```
fs.exists('/etc/passwd', (exists) => {
  console.log(exists ? 'it\'s there' : 'no passwd!');
});
```

`fs.exists()` should not be used to check if a file exists before calling `fs.open()`. Doing so introduces a race condition since other processes may change the file's state between the two calls. Instead, user code should call `fs.open()` directly and handle the error raised if the file is non-existent.

fs.existsSync(path)

#

Stability: 0 - Deprecated: Use [fs.statSync\(\)](#) or [fs.accessSync\(\)](#) instead.

- `path` <String> | <Buffer>

Synchronous version of [fs.exists\(\)](#). Returns `true` if the file exists, `false` otherwise.

fs.fchmod(fd, mode, callback)

#

Added in: v0.4.7

- `fd` <Integer>
- `mode` <Integer>
- `callback` <Function>

Asynchronous [fchmod\(2\)](#). No arguments other than a possible exception are given to the completion callback.

fs.fchmodSync(fd, mode)

#

Added in: v0.4.7

- `fd` <Integer>
- `mode` <Integer>

Synchronous [fchmod\(2\)](#). Returns `undefined`.

fs.fchown(fd, uid, gid, callback)

#

Added in: v0.4.7

- `fd` <Integer>
- `uid` <Integer>
- `gid` <Integer>
- `callback` <Function>

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

fs.fchownSync(fd, uid, gid)

#

Added in: v0.4.7

- `fd` <Integer>
- `uid` <Integer>
- `gid` <Integer>

Synchronous `fchown(2)`. Returns `undefined`.

fs.fdatasync(fd, callback)

#

Added in: v0.1.96

- `fd` <Integer>
- `callback` <Function>

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

fs.fdatasyncSync(fd)

#

Added in: v0.1.96

- `fd` <Integer>

Synchronous `fdatasync(2)`. Returns `undefined`.

fs.fstat(fd, callback)

#

Added in: v0.1.95

- `fd` <Integer>
- `callback` <Function>

Asynchronous `fstat(2)`. The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. `fstat()` is identical to `stat()`, except that the file to

be stat-ed is specified by the file descriptor `fd`.

fs.fstatSync(fd)

#

Added in: v0.1.95

- `fd` <Integer>

Synchronous `fstat(2)`. Returns an instance of `fs.Stats`.

fs.fsync(fd, callback)

#

Added in: v0.1.96

- `fd` <Integer>
- `callback` <Function>

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

fs.fsyncSync(fd)

#

Added in: v0.1.96

- `fd` <Integer>

Synchronous `fsync(2)`. Returns `undefined`.

fs.ftruncate(fd, len, callback)

#

Added in: v0.8.6

- `fd` <Integer>
- `len` <Integer>
- `callback` <Function>

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

fs.ftruncateSync(fd, len)

#

- `fd` <Integer>
- `len` <Integer>

Synchronous `ftruncate(2)`. Returns `undefined`.

fs.futimes(fd, atime, mtime, callback)

Added in: v0.4.2

- `fd` <Integer>
- `atime` <Integer>
- `mtime` <Integer>
- `callback` <Function>

Change the file timestamps of a file referenced by the supplied file descriptor.

fs.futimesSync(fd, atime, mtime)

Added in: v0.4.2

- `fd` <Integer>
- `atime` <Integer>
- `mtime` <Integer>

Synchronous version of `fs.futimes()`. Returns `undefined`.

fs.lchmod(path, mode, callback)

Deprecated since: v0.4.7

- `path` <String> | <Buffer>
- `mode` <Integer>
- `callback` <Function>

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

fs.lchmodSync(path, mode)

#

Deprecated since: v0.4.7

- `path` `<String> | <Buffer>`
- `mode` `<Integer>`

Synchronous [lchmod\(2\)](#). Returns `undefined`.

fs.lchown(path, uid, gid, callback)

#

Deprecated since: v0.4.7

- `path` `<String> | <Buffer>`
- `uid` `<Integer>`
- `gid` `<Integer>`
- `callback` `<Function>`

Asynchronous [lchown\(2\)](#). No arguments other than a possible exception are given to the completion callback.

fs.lchownSync(path, uid, gid)

#

Deprecated since: v0.4.7

- `path` `<String> | <Buffer>`
- `uid` `<Integer>`
- `gid` `<Integer>`

Synchronous [lchown\(2\)](#). Returns `undefined`.

fs.link(srcpath, dstpath, callback)

#

Added in: v0.1.31

- `srcpath` `<String> | <Buffer>`
- `dstpath` `<String> | <Buffer>`

- `callback` <Function>

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

fs.linkSync(srcpath, dstpath)

Added in: v0.1.31

- `srcpath` <String> | <Buffer>
- `dstpath` <String> | <Buffer>

Synchronous `link(2)`. Returns `undefined`.

fs.lstat(path, callback)

Added in: v0.1.30

- `path` <String> | <Buffer>
- `callback` <Function>

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.lstatSync(path)

Added in: v0.1.30

- `path` <String> | <Buffer>

Synchronous `lstat(2)`. Returns an instance of `fs.Stats`.

fs.mkdir(path[, mode], callback)

Added in: v0.1.8

- `path` <String> | <Buffer>
- `mode` <Integer>
- `callback` <Function>

Asynchronous `mkdir(2)`. No arguments other than a possible exception are given to the completion callback. `mode` defaults to `0o777`.

fs.mkdirSync(path[, mode])

#

Added in: v0.1.21

- `path` `<String>` | `<Buffer>`
- `mode` `<Integer>`

Synchronous `mkdir(2)`. Returns `undefined`.

fs.mkdtemp(prefix, callback)

#

Added in: v5.10.0

Creates a unique temporary directory.

Generates six random characters to be appended behind a required `prefix` to create a unique temporary directory.

The created folder path is passed as a string to the callback's second parameter.

Example:

```
fs.mkdtemp('/tmp/foo-', (err, folder) => {
  console.log(folder);
  // Prints: /tmp/foo-itXde2
});
```

Note: The `fs.mkdtemp()` method will append the six randomly selected characters directly to the `prefix` string. For instance, given a directory `/tmp`, if the intention is to create a temporary directory *within* `/tmp`, the `prefix` must end with a trailing platform-specific path separator (`require('path').sep`).

```
// The parent directory for the new temporary directory
```

```

const tmpDir = '/tmp';

// This method is *INCORRECT*:
fs.mkdtemp(tmpDir, (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Will print something similar to `/tmpabc123`.
  // Note that a new temporary directory is created
  // at the file system root rather than *within*
  // the /tmp directory.
});

// This method is *CORRECT*:
const path = require('path');
fs.mkdtemp(tmpDir + path.sep, (err, folder) => {
  if (err) throw err;
  console.log(folder);
  // Will print something similar to `/tmp/abc123`.
  // A new temporary directory is created within
  // the /tmp directory.
});

```

fs.mkdtempSync(prefix)

#

Added in: v5.10.0

The synchronous version of `fs.mkdtemp()`. Returns the created folder path.

fs.open(path, flags[, mode], callback)

#

Added in: v0.0.2

- `path` <String> | <Buffer>
- `flags` <String> | <Number>
- `mode` <Integer>

- `callback <Function>`

Asynchronous file open. See `open(2)`. `flags` can be:

- `'r'` - Open file for reading. An exception occurs if the file does not exist.
- `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs+'` - Open file for reading and writing in synchronous mode. Instructs the operating system to bypass the local file system cache.

This is primarily useful for opening files on NFS mounts as it allows you to skip the potentially stale local cache. It has a very real impact on I/O performance so don't use this flag unless you need it.

Note that this doesn't turn `fs.open()` into a synchronous blocking call. If that's what you want then you should be using `fs.openSync()`

- `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` - Like `'w'` but fails if `path` exists.
- `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` - Like `'w+'` but fails if `path` exists.
- `'a'` - Open file for appending. The file is created if it does not exist.
- `'ax'` - Like `'a'` but fails if `path` exists.
- `'a+'` - Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` - Like `'a+'` but fails if `path` exists.

`mode` sets the file mode (permission and sticky bits), but only if the file was created.

It defaults to `0666`, readable and writable.

The callback gets two arguments `(err, fd)`.

The exclusive flag `'x'` (`O_EXCL` flag in [open\(2\)](#)) ensures that `path` is newly created. On POSIX systems, `path` is considered to exist even if it is a symlink to a non-existent file. The exclusive flag may or may not work with network file systems.

`flags` can also be a number as documented by [open\(2\)](#); commonly used constants are available from `fs.constants`. On Windows, flags are translated to their equivalent ones where applicable, e.g. `O_WRONLY` to `FILE_GENERIC_WRITE`, or `O_EXCL|O_CREAT` to `CREATE_NEW`, as accepted by `CreateFileW`.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

Note: The behavior of `fs.open()` is platform specific for some flags. As such, opening a directory on OS X and Linux with the `'a+'` flag - see example below - will return an error. In contrast, on Windows and FreeBSD, a file descriptor will be returned.

```
// OS X and Linux
fs.open('<directory>', 'a+', (err, fd) => {
  // => [Error: EISDIR: illegal operation on a directory, open <directory>]
})

// Windows and FreeBSD
fs.open('<directory>', 'a+', (err, fd) => {
  // => null, <fd>
})
```

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

Added in: v0.1.21

- `path` `<String> | <Buffer>`
- `flags` `<String> | <Number>`
- `mode` `<Integer>`

Synchronous version of `fs.open()`. Returns an integer representing the file descriptor.

`fs.read(fd, buffer, offset, length, position, callback)`

Added in: v0.0.2

- `fd` `<Integer>`
- `buffer` `<String> | <Buffer>`
- `offset` `<Integer>`
- `length` `<Integer>`
- `position` `<Integer>`
- `callback` `<Function>`

Read data from the file specified by `fd`.

`buffer` is the buffer that the data will be written to.

`offset` is the offset in the buffer to start writing at.

`length` is an integer specifying the number of bytes to read.

`position` is an integer specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position.

The callback is given the three arguments, `(err, bytesRead, buffer)`.

`fs.readdir(path[, options], callback)`

Added in: v0.1.8

- `path` `<String> | <Buffer>`
- `options` `<String> | <Object>`

- encoding <String> default = 'utf8'
- callback <Function>

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 `'..'`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames passed to the callback. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `Buffer` objects.

fs.readdirSync(path[, options])

#

Added in: v0.1.21

- path <String> | <Buffer>
- options <String> | <Object>
 - encoding <String> default = 'utf8'

Synchronous `readdir(3)`. Returns an array of filenames excluding `'.'` and `'..'`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames passed to the callback. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `Buffer` objects.

fs.readFile(file[, options], callback)

#

Added in: v0.1.29

- file <String> | <Buffer> | <Integer> filename or file descriptor
- options <Object> | <String>
 - encoding <String> | <Null> default = `null`
 - flag <String> default = `'r'`
- callback <Function>

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', (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.

If `options` is a string, then it specifies the encoding. Example:

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

Any specified file descriptor has to support reading.

Note: Specified file descriptors will not be closed automatically.

fs.readFileSync(file[, options])

#

Added in: v0.1.8

- `file` `<String> | <Buffer> | <Integer>` filename or file descriptor
- `options` `<Object> | <String>`
 - `encoding` `<String> | <Null>` default = `null`
 - `flag` `<String>` default = `'r'`

Synchronous version of `fs.readFile`. Returns the contents of the `file`.

If the `encoding` option is specified then this function returns a string. Otherwise it returns a buffer.

fs.readlink(path[, options], callback)

#

Added in: v0.1.31

- `path` `<String> | <Buffer>`
- `options` `<String> | <Object>`
 - `encoding` `<String>` default = `'utf8'`
- `callback` `<Function>`

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

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path passed to the callback. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fs.readlinkSync(path[, options])

Added in: v0.1.31

- `path` `<String> | <Buffer>`
- `options` `<String> | <Object>`
 - `encoding` `<String>` default = `'utf8'`

Synchronous `readlink(2)`. Returns the symbolic link's string value.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the link path passed to the callback. If the `encoding` is set to `'buffer'`, the link path returned will be passed as a `Buffer` object.

fs.readSync(fd, buffer, offset, length, position)

Added in: v0.1.21

- `fd` `<Integer>`
- `buffer` `<String> | <Buffer>`
- `offset` `<Integer>`
- `length` `<Integer>`
- `position` `<Integer>`

Synchronous version of `fs.read()`. Returns the number of `bytesRead`.

fs.realpath(path[, options], callback)

Added in: v0.1.31

- `path` `<String> | <Buffer>`
- `options` `<String> | <Object>`
 - `encoding` `<String>` default = `'utf8'`
- `callback` `<Function>`

Asynchronous `realpath(3)`. The `callback` gets two arguments `(err, resolvedPath)`. May use `process.cwd` to resolve relative paths.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

fs.realpathSync(path[, options])

Added in: v0.1.31

- `path` `<String> | <Buffer>;`
- `options` `<String> | <Object>`
 - `encoding` `<String>` default = `'utf8'`

Synchronous `realpath(3)`. Returns the resolved path.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the path passed to the callback. If the `encoding` is set to `'buffer'`, the path returned will be passed as a `Buffer` object.

fs.rename(oldPath, newPath, callback)

Added in: v0.0.2

- `oldPath` `<String> | <Buffer>`

- `newPath` `<String> | <Buffer>`
- `callback` `<Function>`

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

fs.renameSync(oldPath, newPath)

Added in: v0.1.21

- `oldPath` `<String> | <Buffer>`
- `newPath` `<String> | <Buffer>`

Synchronous `rename(2)`. Returns `undefined`.

fs.rmdir(path, callback)

Added in: v0.0.2

- `path` `<String> | <Buffer>`
- `callback` `<Function>`

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

fs.rmdirSync(path)

Added in: v0.1.21

- `path` `<String> | <Buffer>`

Synchronous `rmdir(2)`. Returns `undefined`.

fs.stat(path, callback)

Added in: v0.0.2

- `path` `<String> | <Buffer>`
- `callback` `<Function>`

Asynchronous `stat(2)`. The callback gets two arguments `(err, stats)` where

`stats` is a `fs.Stats` object. See the `fs.Stats` section for more information.

fs.statSync(path)

#

Added in: v0.1.21

- `path` `<String> | <Buffer>`

Synchronous `stat(2)`. Returns an instance of `fs.Stats`.

fs.symlink(target, path[, type], callback)

#

Added in: v0.1.31

- `target` `<String> | <Buffer>`
- `path` `<String> | <Buffer>`
- `type` `<String>`
- `callback` `<Function>`

Asynchronous `symlink(2)`. No arguments other than a possible exception are given to the completion callback. The `type` argument can be set to `'dir'`, `'file'`, or `'junction'` (default is `'file'`) and is only available on Windows (ignored on other platforms). Note that Windows junction points require the destination path to be absolute. When using `'junction'`, the `target` argument will automatically be normalized to absolute path.

Here is an example below:

```
fs.symlink('./foo', './new-port');
```

It creates a symbolic link named "new-port" that points to "foo".

fs.symlinkSync(target, path[, type])

#

Added in: v0.1.31

- `target` `<String> | <Buffer>`

- `path` `<String> | <Buffer>`
- `type` `<String>`

Synchronous `symlink(2)`. Returns `undefined`.

fs.truncate(path, len, callback)

Added in: v0.8.6

- `path` `<String> | <Buffer>`
- `len` `<Integer>`
- `callback` `<Function>`

Asynchronous `truncate(2)`. No arguments other than a possible exception are given to the completion callback. A file descriptor can also be passed as the first argument. In this case, `fs.ftruncate()` is called.

fs.truncateSync(path, len)

Added in: v0.8.6

- `path` `<String> | <Buffer>`
- `len` `<Integer>`

Synchronous `truncate(2)`. Returns `undefined`.

fs.unlink(path, callback)

Added in: v0.0.2

- `path` `<String> | <Buffer>`
- `callback` `<Function>`

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

fs.unlinkSync(path)

Added in: v0.1.21

- `path` `<String> | <Buffer>`

Synchronous `unlink(2)`. Returns `undefined`.

fs.unwatchFile(filename[, listener])

Added in: v0.1.31

- `filename` `<String> | <Buffer>`
- `listener` `<Function>`

Stop watching for changes on `filename`. If `listener` is specified, only that particular listener is removed. Otherwise, *all* listeners are removed and you have effectively stopped watching `filename`.

Calling `fs.unwatchFile()` with a filename that is not being watched is a no-op, not an error.

Note: `fs.watch()` is more efficient than `fs.watchFile()` and `fs.unwatchFile()`. `fs.watch()` should be used instead of `fs.watchFile()` and `fs.unwatchFile()` when possible.

fs.utimes(path, atime, mtime, callback)

Added in: v0.4.2

- `path` `<String> | <Buffer>`
- `atime` `<Integer>`
- `mtime` `<Integer>`
- `callback` `<Function>`

Change file timestamps of the file referenced by the supplied path.

Note: the arguments `atime` and `mtime` of the following related functions does follow the below rules:

- If the value is a numberable string like `'123456789'`, the value would get converted to corresponding number.

- If the value is `Nan` or `Infinity`, the value would get converted to `Date.now()`.

fs.utimesSync(path, atime, mtime)

Added in: v0.4.2

- `path` `<String> | <Buffer>`
- `atime` `<Integer>`
- `mtime` `<Integer>`

Synchronous version of `fs.utimes()`. Returns `undefined`.

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

Added in: v0.5.10

- `filename` `<String> | <Buffer>`
- `options` `<String> | <Object>`
 - `persistent` `<Boolean>` Indicates whether the process should continue to run as long as files are being watched. default = `true`
 - `recursive` `<Boolean>` Indicates whether all subdirectories should be watched, or only the current directory. This applies when a directory is specified, and only on supported platforms (See [Caveats](#)). default = `false`
 - `encoding` `<String>` Specifies the character encoding to be used for the filename passed to the listener. default = `'utf8'`
- `listener` `<Function>`

Watch for changes on `filename`, where `filename` is either a file or a directory.

The returned object is a `fs.FSWatcher`.

The second argument is optional. If `options` is provided as a string, it specifies the `encoding`. Otherwise `options` should be passed as an object.

The listener callback gets two arguments `(event, filename)`. `event` is either `'rename'` or `'change'`, and `filename` is the name of the file which triggered the event.

Caveats

The `fs.watch` API is not 100% consistent across platforms, and is unavailable in some situations.

The recursive option is only supported on OS X and Windows.

Availability

This feature depends on the underlying operating system providing a way to be notified of filesystem changes.

- On Linux systems, this uses `inotify`
- On BSD systems, this uses `kqueue`
- On OS X, this uses `kqueue` for files and `FSEvents` for directories.
- On SunOS systems (including Solaris and SmartOS), this uses `event ports`.
- On Windows systems, this feature depends on `ReadDirectoryChangesW`.
- On Aix systems, this feature depends on [`AHAFS`], which must be enabled.

If the underlying functionality is not available for some reason, then `fs.watch` will not be able to function. For example, watching files or directories can be unreliable, and in some cases impossible, on network file systems (NFS, SMB, etc), or host file systems when using virtualization software such as Vagrant, Docker, etc.

You can still use `fs.watchFile`, which uses stat polling, but it is slower and less reliable.

Inodes

On Linux and OS X systems, `fs.watch()` resolves the path to an `inode` and watches the inode. If the watched path is deleted and recreated, it is assigned a new inode. The watch will emit an event for the delete but will continue watching the *original* inode. Events for the new inode will not be emitted. This is expected behavior.

Filename Argument

Providing `filename` argument in the callback is only supported on Linux and

Windows. Even on supported platforms, `filename` is not always guaranteed to be provided. Therefore, don't assume that `filename` argument is always provided in the callback, and have some fallback logic if it is null.

```
fs.watch('somedir', (event, filename) => {
  console.log(`event is: ${event}`);
  if (filename) {
    console.log(`filename provided: ${filename}`);
  } else {
    console.log('filename not provided');
  }
});
```

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

#

Added in: v0.1.31

- `filename` `<String> | <Buffer>`
- `options` `<Object>`
 - `persistent` `<Boolean>`
 - `interval` `<Integer>`
- `listener` `<Function>`

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.

The `options` argument may be omitted. If provided, it should be an object. The `options` object may contain a boolean named `persistent` that indicates whether the process should continue to run as long as files are being watched. The `options` object may specify an `interval` property indicating how often the target should be polled in milliseconds. The default is `{ persistent: true, interval: 5007 }`.

The `listener` gets two arguments the current stat object and the previous stat object:

```
fs.watchFile('message.text', (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`.

Note: when an `fs.watchFile` operation results in an `ENOENT` error, it will invoke the listener once, with all the fields zeroed (or, for dates, the Unix Epoch). In Windows, `blksize` and `blocks` fields will be `undefined`, instead of zero. If the file is created later on, the listener will be called again, with the latest stat objects. This is a change in functionality since v0.10.

Note: `fs.watch()` is more efficient than `fs.watchFile` and `fs.unwatchFile`. `fs.watch` should be used instead of `fs.watchFile` and `fs.unwatchFile` when possible.

`fs.write(fd, buffer, offset, length[, position], callback)`

Added in: v0.0.2

- `fd` `<Integer>`
- `buffer` `<String> | <Buffer>`
- `offset` `<Integer>`
- `length` `<Integer>`
- `position` `<Integer>`
- `callback` `<Function>`

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 `typeof position !== 'number'`, the data will be written at the current position. See [pwrite\(2\)](#).

The callback will be given three arguments `(err, written, buffer)` where `written` specifies how many *bytes* were written from `buffer`.

Note that it is unsafe to use `fs.write` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

`fs.write(fd, data[, position[, encoding]], callback)`

Added in: v0.11.5

- `fd` <Integer>
- `data` <String> | <Buffer>
- `position` <Integer>
- `encoding` <String>
- `callback` <Function>

Write `data` to the file specified by `fd`. If `data` is not a Buffer instance then the value will be coerced to a string.

`position` refers to the offset from the beginning of the file where this data should be written. If `typeof position !== 'number'` the data will be written at the current position. See [pwrite\(2\)](#).

`encoding` is the expected string encoding.

The callback will receive the arguments `(err, written, string)` where `written` specifies how many *bytes* the passed string required to be written. Note

that bytes written is not the same as string characters. See [Buffer.byteLength](#).

Unlike when writing `buffer`, the entire string must be written. No substring may be specified. This is because the byte offset of the resulting data may not be the same as the string offset.

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.

On Linux, positional writes don't work when the file is opened in append mode. The kernel ignores the position argument and always appends the data to the end of the file.

fs.writeFile(file, data[, options], callback)

Added in: v0.1.29

- `file` `<String> | <Buffer> | <Integer>` filename or file descriptor
- `data` `<String> | <Buffer>`
- `options` `<Object> | <String>`
 - `encoding` `<String> | <Null>` default = `'utf8'`
 - `mode` `<Integer>` default = `0o666`
 - `flag` `<String>` default = `'w'`
- `callback` `<Function>`

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

The `encoding` option is ignored if `data` is a buffer. It defaults to `'utf8'`.

Example:

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

```
});
```

If `options` is a string, then it specifies the encoding. Example:

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

Any specified file descriptor has to support writing.

Note that it is unsafe to use `fs.writeFile` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

Note: Specified file descriptors will not be closed automatically.

fs.writeFileSync(file, data[, options])

Added in: v0.1.29

- `file` `<String> | <Buffer> | <Integer>` filename or file descriptor
- `data` `<String> | <Buffer>`
- `options` `<Object> | <String>`
 - `encoding` `<String> | <Null>` default = `'utf8'`
 - `mode` `<Integer>` default = `0o666`
 - `flag` `<String>` default = `'w'`

The synchronous version of `fs.writeFile()`. Returns `undefined`.

fs.writeSync(fd, buffer, offset, length[, position])

Added in: v0.1.21

- `fd` `<Integer>`
- `buffer` `<String> | <Buffer>`
- `offset` `<Integer>`
- `length` `<Integer>`

- `position` <Integer>

fs.writeFileSync(fd, data[, position[, encoding]])

#

Added in: v0.11.5

- `fd` <Integer>
- `data` <String> | <Buffer>
- `position` <Integer>
- `encoding` <String>

Synchronous versions of `fs.write()`. Returns the number of bytes written.

FS Constants

#

The following constants are exported by `fs.constants`. Note: Not every constant will be available on every operating system.

File Access Constants

#

The following constants are meant for use with `fs.access()`.

Constant	Description
F_OK	Flag indicating that the file is visible to the calling process.
R_OK	Flag indicating that the file can be read by the calling process.
W_OK	Flag indicating that the file can be written by the calling process.
X_OK	Flag indicating that the file can be executed by the calling process.

File Open Constants

#

The following constants are meant for use with `fs.open()`.

Constant	Description
O_RDONLY	Flag indicating to open a file for read-only access.
O_WRONLY	Flag indicating to open a file for write-only access.
O_RDWR	Flag indicating to open a file for read-write access.
O_CREAT	Flag indicating to create the file if it does not already exist.
O_EXCL	Flag indicating that opening a file should fail if the O_CREAT flag is set and the file already exists.
O_NOCTTY	Flag indicating that if path identifies a terminal device, opening the path shall not cause that terminal to become the controlling terminal for the process (if the process does not already have one).
O_TRUNC	Flag indicating that if the file exists and is a regular file, and the file is opened successfully for write access, its length shall be truncated to zero.
O_APPEND	Flag indicating that data will be appended to the end of the file.
O_DIRECTORY	Flag indicating that the open should fail if the path is not a directory.
O_NOATIME	Flag indicating reading accesses to the file system will no longer result in an update to the atime information associated with the file. This flag is available on Linux operating systems only.
O_NOFOLLOW	Flag indicating that the open should fail if the path is a symbolic link.
O_SYNC	Flag indicating that the file is opened for synchronous I/O.

O_SYMLINK	Flag indicating to open the symbolic link itself rather than the resource it is pointing to.
O_DIRECT	When set, an attempt will be made to minimize caching effects of file I/O.
O_NONBLOCK	Flag indicating to open the file in nonblocking mode when possible.

File Type Constants

The following constants are meant for use with the `fs.Stats` object's `mode` property for determining a file's type.

Constant	Description
S_IFMT	Bit mask used to extract the file type code.
S_IFREG	File type constant for a regular file.
S_IFDIR	File type constant for a directory.
S_IFCHR	File type constant for a character-oriented device file.
S_IFBLK	File type constant for a block-oriented device file.
S_IFIFO	File type constant for a FIFO/pipe.
S_IFLNK	File type constant for a symbolic link.
S_IFSOCK	File type constant for a socket.

File Mode Constants

The following constants are meant for use with the `fs.Stats` object's `mode` property for determining the access permissions for a file.

Constant	Description
S_IRWXU	File mode indicating readable, writable and executable by owner.
S_IRUSR	File mode indicating readable by owner.
S_IWUSR	File mode indicating writable by owner.
S_IXUSR	File mode indicating executable by owner.
S_IRWXG	File mode indicating readable, writable and executable by group.
S_IRGRP	File mode indicating readable by group.
S_IWGRP	File mode indicating writable by group.
S_IXGRP	File mode indicating executable by group.
S_IRWXO	File mode indicating readable, writable and executable by others.
S_IROTH	File mode indicating readable by others.
S_IWOTH	File mode indicating writable by others.
S_IXOTH	File mode indicating executable by others.

<<<<< HEAD

[AHAFS]:

#

https://www.ibm.com/developerworks/aix/aix_event_infrastructure/

```
dcccbfd...
src: refactor
require('constants')
```

Global Objects

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

The objects listed here are specific to Node.js. There are a number of [built-in objects](#) that are part of the JavaScript language itself, which are also globally accessible.

Class: Buffer

- [<Function>](#)

Used to handle binary data. See the [buffer section](#).

__dirname

- [<String>](#)

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

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

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

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

For instance, given two modules: `a` and `b`, where `b` is a dependency of `a` and

there is a directory structure of:

- /Users/mjr/app/a.js
- /Users/mjr/app/node_modules/b/b.js

References to `__dirname` within `b.js` will return `/Users/mjr/app/node_modules/b` while references to `__dirname` within `a.js` will return `/Users/mjr/app`.

`__filename`

- `<String>`

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

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

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

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

`clearImmediate(immediateObject)`

`clearImmediate` is described in the `timers` section.

`clearInterval(intervalObject)`

`clearInterval` is described in the `timers` section.

`clearTimeout(timeoutObject)`

`clearTimeout` is described in the `timers` section.

console

#

- <Object>

Used to print to stdout and stderr. See the [console](#) section.

exports

#

A reference to the `module.exports` that is shorter to type. See [module system documentation](#) for details on when to use `exports` and when to use `module.exports`.

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

See the [module system documentation](#) for more information.

global

#

- <Object> The global namespace object.

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

module

#

- <Object>

A reference to the current module. In particular `module.exports` is used for defining what a module exports and makes available through `require()`.

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

See the [module system documentation](#) for more information.

process

#

- <Object>

The process object. See the [process object](#) section.

require()

#

- <Function>

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

require.cache

#

- <Object>

Modules are cached in this object when they are required. By deleting a key value from this object, the next `require` will reload the module. Note that this does not apply to [native addons](#), for which reloading will result in an Error.

require.extensions

#

Stability: 0 - Deprecated

- <Object>

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

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

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

Deprecated In the past, this list has been used to load non-JavaScript modules into Node.js by compiling them on-demand. However, in practice, there are much better ways to do this, such as loading modules via some other Node.js program, or compiling them to JavaScript ahead of time.

Since the Module system is locked, this feature will probably never go away.

However, it may have subtle bugs and complexities that are best left untouched.

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.

setImmediate(callback[, arg][, ...])

#

`setImmediate` is described in the [timers](#) section.

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

#

`setInterval` is described in the [timers](#) section.

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

#

`setTimeout` is described in the [timers](#) section.

HTTP

#

Stability: 2 - Stable

To use the HTTP server and client one must `require('http')`.

The HTTP interfaces in Node.js 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',
'host': 'mysite.com',
'accept': '*/*' }
```

Keys are lowercased. Values are not modified.

In order to support the full spectrum of possible HTTP applications, Node.js'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.

See `message.headers` for details on how duplicate headers are handled.

The raw headers as they were received are retained in the `rawHeaders` property, which is an array of `[key, value, key2, value2, ...]`. For example, the previous message header object might have a `rawHeaders` list like the following:

```
[ 'Content-Length', '123456',
  'content-LENGTH', '123',
  'content-type', 'text/plain',
  'CONNECTION', 'keep-alive',
  'Host', 'mysite.com',
  'accepT', '*/*' ]
```

Class: http.Agent

Added in: v0.3.4

The HTTP Agent is used for pooling sockets used in HTTP client requests.

The HTTP Agent also defaults client requests to using `Connection:keep-alive`. If no pending HTTP requests are waiting on a socket to become free the socket is closed. This means that Node.js's pool has the benefit of `keep-alive` when under load but still does not require developers to manually close the HTTP clients using `KeepAlive`.

If you opt into using HTTP KeepAlive, you can create an Agent object with that flag set to `true`. (See the [constructor options](#).) Then, the Agent will keep unused sockets in a pool for later use. They will be explicitly marked so as to not keep the Node.js process running. However, it is still a good idea to explicitly `destroy()` KeepAlive agents when they are no longer in use, so that the Sockets will be shut down.

Sockets are removed from the agent's pool when the socket emits either a `'close'` event or a special `'agentRemove'` event. This means that if you intend to keep one HTTP request open for a long time and don't want it to stay in the pool you can do something along the lines of:

```
http.get(options, (res) => {
  // Do stuff
}).on('socket', (socket) => {
  socket.emit('agentRemove');
});
```

Alternatively, you could just opt out of pooling entirely using `agent:false`:

```
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false // create a new agent just for this one request
}, (res) => {
  // Do stuff with response
})
```

[new Agent\(options\)](#)

#

Added in: v0.3.4

- `options` `<Object>` Set of configurable options to set on the agent. Can have the following fields:
 - `keepAlive` `<Boolean>` Keep sockets around in a pool to be used by other requests in the future. Default = `false`
 - `keepAliveMsecs` `<Integer>` When using HTTP KeepAlive, how often to send TCP KeepAlive packets over sockets being kept alive. Default = `1000`. Only relevant if `keepAlive` is set to `true`.
 - `maxSockets` `<Number>` Maximum number of sockets to allow per host. Default = `Infinity`.
 - `maxFreeSockets` `<Number>` Maximum number of sockets to leave open in a free state. Only relevant if `keepAlive` is set to `true`. Default = `256`.

The default `http.globalAgent` that is used by `http.request()` has all of these values set to their respective defaults.

To configure any of them, you must create your own `http.Agent` object.

```
const http = require('http');
var keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

`agent.createConnection(options[, callback])`

#

Added in: v0.11.4

Produces a socket/stream to be used for HTTP requests.

By default, this function is the same as `net.createConnection()`. However, custom Agents may override this method in case greater flexibility is desired.

A socket/stream can be supplied in one of two ways: by returning the socket/stream from this function, or by passing the socket/stream to `callback`.

`callback` has a signature of `(err, stream)`.

agent.destroy()

#

Added in: v0.11.4

Destroy any sockets that are currently in use by the agent.

It is usually not necessary to do this. However, if you are using an agent with KeepAlive enabled, then it is best to explicitly shut down the agent when you know that it will no longer be used. Otherwise, sockets may hang open for quite a long time before the server terminates them.

agent.freeSockets

#

Added in: v0.11.4

An object which contains arrays of sockets currently awaiting use by the Agent when HTTP KeepAlive is used. Do not modify.

agent.getName(options)

#

Added in: v0.11.4

Get a unique name for a set of request options, to determine whether a connection can be reused. In the http agent, this returns `host:port:localAddress`. In the https agent, the name includes the CA, cert, ciphers, and other HTTPS/TLS-specific options that determine socket reusability.

Options:

- `host` : A domain name or IP address of the server to issue the request to.
- `port` : Port of remote server.
- `localAddress` : Local interface to bind for network connections when issuing the request.

agent.maxFreeSockets

#

Added in: v0.11.7

By default set to 256. For Agents supporting HTTP KeepAlive, this sets the maximum number of sockets that will be left open in the free state.

agent.maxSockets

#

Added in: v0.3.6

By default set to Infinity. Determines how many concurrent sockets the agent can have open per origin. Origin is either a 'host:port' or 'host:port:localAddress' combination.

agent.requests

#

Added in: v0.5.9

An object which contains queues of requests that have not yet been assigned to sockets. Do not modify.

agent.sockets

#

Added in: v0.3.6

An object which contains arrays of sockets currently in use by the Agent. Do not modify.

Class: http.ClientRequest

#

Added in: v0.1.17

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.IncomingMessage`.

During the '`'response'`' event, one can add listeners to the response object; particularly to listen for the '`'data'`' event.

If no '`'response'`' handler is added, then the response will be entirely discarded.

However, if you add a '`'response'` event handler, then you **must** consume the data from the response object, either by calling `response.read()` whenever there is a '`'readable'`' event, or by adding a '`'data'`' handler, or by calling the `.resume()` method. Until the data is consumed, the '`'end'`' event will not fire. Also, until the data is read it will consume memory that can eventually lead to a 'process out of memory' error.

Note: Node.js does not check whether `Content-Length` and the length of the body which has been transmitted are equal or not.

The request implements the [Writable Stream](#) interface. This is an [EventEmitter](#) with the following events:

Event: 'abort'#

Added in: v1.4.1

```
function () { }
```

Emitted when the request has been aborted by the client. This event is only emitted on the first call to `abort()`.

Event: 'aborted'#

Added in: v0.3.8

```
function () { }
```

Emitted when the request has been aborted by the server and the network socket has closed.

Event: 'checkExpectation'#

Added in: v5.5.0

```
function (request, response) { }
```

Emitted each time a request with an http Expect header is received, where the value is not 100-continue. If this event isn't listened for, the server will automatically respond with a 417 Expectation Failed as appropriate.

Note that when this event is emitted and handled, the `request` event will not be emitted.

Event: 'connect'

Added in: v0.7.0

```
function (response, socket, head) { }
```

Emitted each time a server responds to a request with a `CONNECT` method. If this event isn't being listened for, clients receiving a `CONNECT` method will have their connections closed.

A client server pair that show you how to listen for the `'connect'` event.

```
const http = require('http');
const net = require('net');
const url = require('url');

// Create an HTTP tunneling proxy
var proxy = http.createServer( (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});

proxy.on('connect', (req, cltSocket, head) => {
  // connect to an origin server
  var srvUrl = url.parse(`http://${req.url}`);
  var srvSocket = net.connect(srvUrl.port, srvUrl.hostname, () => {
    cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
      'Proxy-agent: Node.js-Proxy\r\n' +
      '\r\n');
    srvSocket.write(head);
    srvSocket.pipe(cltSocket);
    cltSocket.pipe(srvSocket);
  });
});
```

```
// now that proxy is running
proxy.listen(1337, '127.0.0.1', () => {

    // make a request to a tunneling proxy
    var options = {
        port: 1337,
        hostname: '127.0.0.1',
        method: 'CONNECT',
        path: 'www.google.com:80'
    };

    var req = http.request(options);
    req.end();

    req.on('connect', (res, socket, head) => {
        console.log('got connected!');

        // make a request over an HTTP tunnel
        socket.write('GET / HTTP/1.1\r\n' +
            'Host: www.google.com:80\r\n' +
            'Connection: close\r\n' +
            '\r\n');

        socket.on('data', (chunk) => {
            console.log(chunk.toString());
        });

        socket.on('end', () => {
            proxy.close();
        });
    });
});
```

Event: 'continue'

Added in: v0.3.2

```
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.

Event: 'response'

Added in: v0.1.0

```
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.IncomingMessage](#).

Event: 'socket'

Added in: v0.5.3

```
function (socket) { }
```

Emitted after a socket is assigned to this request.

Event: 'upgrade'

Added in: v0.1.94

```
function (response, socket, head) { }
```

Emitted each time a server responds to a request with an upgrade. If this event isn't being listened for, clients receiving an upgrade header will have their connections closed.

A client server pair that show you how to listen for the '`'upgrade'` event.

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

```
// Create an HTTP server
```

```
var srv = http.createServer( (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});

srv.on('upgrade', (req, socket, head) => {
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
    'Upgrade: WebSocket\r\n' +
    'Connection: Upgrade\r\n' +
    '\r\n');

  socket.pipe(socket); // echo back
});

// now that server is running
srv.listen(1337, '127.0.0.1', () => {

  // make a request
  var options = {
    port: 1337,
    hostname: '127.0.0.1',
    headers: {
      'Connection': 'Upgrade',
      'Upgrade': 'websocket'
    }
  };

  var req = http.request(options);
  req.end();

  req.on('upgrade', (res, socket, upgradeHead) => {
    console.log('got upgraded!');
    socket.end();
    process.exit(0);
  });
});
```

```
});
```

```
});
```

request.abort()

#

Added in: v0.3.8

Marks the request as aborting. Calling this will cause remaining data in the response to be dropped and the socket to be destroyed.

request.end([data][, encoding][, callback])

#

Added in: v0.1.90

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 `response.write(data, encoding)` followed by `request.end(callback)`.

If `callback` is specified, it will be called when the request stream is finished.

request.flushHeaders()

#

Added in: v1.6.0

Flush the request headers.

For efficiency reasons, Node.js normally buffers the request headers until you call `request.end()` or write the first chunk of request data. It then tries hard to pack the request headers and data into a single TCP packet.

That's usually what you want (it saves a TCP round-trip) but not when the first data isn't sent until possibly much later. `request.flushHeaders()` lets you bypass the optimization and kickstart the request.

request.setNoDelay([noDelay])

#

Added in: v0.5.9

Once a socket is assigned to this request and is connected `socket.setNoDelay()`

will be called.

request.setSocketKeepAlive([enable][, initialDelay])

Added in: v0.5.9

Once a socket is assigned to this request and is connected
`socket.setKeepAlive()` will be called.

request.setTimeout(timeout[, callback])

Added in: v0.5.9

Once a socket is assigned to this request and is connected `socket.setTimeout()` will be called.

- `timeout` `<Number>` Milliseconds before a request is considered to be timed out.
- `callback` `<Function>` Optional function to be called when a timeout occurs. Same as binding to the `timeout` event.

request.write(chunk[, encoding][, callback])

Added in: v0.1.29

Sends a chunk of the body. By calling this method many times, the user can stream a request body to a server--in that case it is suggested to use the `['Transfer-Encoding', 'chunked']` header line when creating the request.

The `chunk` argument should be a `Buffer` or a string.

The `encoding` argument is optional and only applies when `chunk` is a string. Defaults to `'utf8'`.

The `callback` argument is optional and will be called when this chunk of data is flushed.

Returns `request`.

Class: http.Server

This class inherits from `net.Server` and has the following additional events:

Event: 'checkContinue'

#

Added in: v0.3.0

```
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: 'clientError'

#

Added in: v0.1.94

```
function (exception, socket) { }
```

If a client connection emits an `'error'` event, it will be forwarded here. Listener of this event is responsible for closing/destroying the underlying socket. For example, one may wish to more gracefully close the socket with an HTTP '400 Bad Request' response instead of abruptly severing the connection.

Default behavior is to destroy the socket immediately on malformed request.

`socket` is the `net.Socket` object that the error originated from.

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

```
const server = http.createServer((req, res) => {
  res.end();
});

server.on('clientError', (err, socket) => {
  socket.end('HTTP/1.1 400 Bad Request\r\n\r\n');
});

server.listen(8000);
```

When the `'clientError'` event occurs, there is no `request` or `response` object, so any HTTP response sent, including response headers and payload, *must* be written directly to the `socket` object. Care must be taken to ensure the response is a properly formatted HTTP response message.

Event: 'close'

Added in: v0.1.4

```
function () { }
```

Emitted when the server closes.

Event: 'connect'

Added in: v0.7.0

```
function (request, socket, head) { }
```

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

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

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

on that socket.

Event: 'connection'

#

Added in: v0.1.0

```
function (socket) { }
```

When a new TCP stream is established. `socket` is an object of type `net.Socket`. Usually users will not want to access this event. In particular, the socket will not emit '`'readable'`' events because of how the protocol parser attaches to the socket. The `socket` can also be accessed at `request.connection`.

Event: 'request'

#

Added in: v0.1.0

```
function (request, response) { }
```

Emitted each time there is a request. Note that there may be multiple requests per connection (in the case of keep-alive connections). `request` is an instance of `http.IncomingMessage` and `response` is an instance of `http.ServerResponse`.

Event: 'upgrade'

#

Added in: v0.1.94

```
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.

server.close([callback])

#

Added in: v0.1.90

Stops the server from accepting new connections. See [net.Server.close\(\)](#).

server.listen(handle[, callback])

#

Added in: v0.5.10

- `handle <Object>`
- `callback <Function>`

The `handle` object can be set to either a server or socket (anything with an underlying `_handle` member), or a `{fd: <n>}` object.

This will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket.

Listening on a file descriptor is not supported on Windows.

This function is asynchronous. The last parameter `callback` will be added as a listener for the `'listening'` event. See also [net.Server.listen\(\)](#).

Returns `server`.

server.listen(path[, callback])

#

Added in: v0.1.90

Start a UNIX socket server listening for connections on the given `path`.

This function is asynchronous. The last parameter `callback` will be added as a listener for the `'listening'` event. See also [net.Server.listen\(path\)](#).

server.listen(port[, hostname][, backlog][, callback])

#

Added in: v0.1.90

Begin accepting connections on the specified `port` and `hostname`. If the `hostname` is omitted, the server will accept connections on any IPv6 address (`::`) when IPv6 is available, or any IPv4 address (`0.0.0.0`) otherwise. Use a port value

of `0` to have the operating system assign an available port.

To listen to a unix socket, supply a filename instead of port and hostname.

Backlog is the maximum length of the queue of pending connections. The actual length will be determined by your OS through sysctl settings such as `tcp_max_syn_backlog` and `somaxconn` on linux. The default value of this parameter is 511 (not 512).

This function is asynchronous. The last parameter `callback` will be added as a listener for the `'listening'` event. See also [net.Server.listen\(port\)](#).

server.listening

#

Added in: v5.7.0

A Boolean indicating whether or not the server is listening for connections.

server.maxHeadersCount

#

Added in: v0.7.0

Limits maximum incoming headers count, equal to 1000 by default. If set to 0 - no limit will be applied.

server.setTimeout(msecs, callback)

#

Added in: v0.9.12

- `msecs <Number>`
- `callback <Function>`

Sets the timeout value for sockets, and emits a `'timeout'` event on the Server object, passing the socket as an argument, if a timeout occurs.

If there is a `'timeout'` event listener on the Server object, then it will be called with the timed-out socket as an argument.

By default, the Server's timeout value is 2 minutes, and sockets are destroyed automatically if they time out. However, if you assign a callback to the Server's `'timeout'` event, then you are responsible for handling socket timeouts.

Returns `server`.

server.timeout

#

Added in: v0.9.12

- <Number> Default = 120000 (2 minutes)

The number of milliseconds of inactivity before a socket is presumed to have timed out.

Note that the socket timeout logic is set up on connection, so changing this value only affects *new* connections to the server, not any existing connections.

Set to 0 to disable any kind of automatic timeout behavior on incoming connections.

Class: http.ServerResponse

#

Added in: v0.1.17

This object is created internally by a HTTP server--not by the user. It is passed as the second parameter to the '`request`' event.

The response implements, but does not inherit from, the [Writable Stream](#) interface. This is an [EventEmitter](#) with the following events:

Event: 'close'

#

Added in: v0.6.7

```
function () { }
```

Indicates that the underlying connection was terminated before `response.end()` was called or able to flush.

Event: 'finish'

#

Added in: v0.3.6

```
function () { }
```

Emitted when the response has been sent. More specifically, this event is emitted

when the last segment of the response headers and body have been handed off to the operating system for transmission over the network. It does not imply that the client has received anything yet.

After this event, no more events will be emitted on the response object.

response.addTrailers(headers)

Added in: v0.3.0

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Trailers will **only** be emitted if chunked encoding is used for the response; if it is not (e.g., if the request was HTTP/1.0), they will be silently discarded.

Note that HTTP requires the `Trailer` header to be sent if you intend to emit trailers, with a list of the header fields in its value. E.g.,

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                           'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667'});
response.end();
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

response.end([data][, encoding][, callback])

Added in: v0.1.90

This method signals to the server that all of the response headers and body have 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(callback)`.

If `callback` is specified, it will be called when the response stream is finished.

response.finished

#

Added in: v0.0.2

Boolean value that indicates whether the response has completed. Starts as `false`.

After `response.end()` executes, the value will be `true`.

response.getHeader(name)

#

Added in: v0.4.0

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.headersSent

#

Added in: v0.9.3

Boolean (read-only). True if headers were sent, false otherwise.

response.removeHeader(name)

#

Added in: v0.4.0

Removes a header that's queued for implicit sending.

Example:

```
response.removeHeader('Content-Encoding');
```

response.sendDate

#

Added in: v0.7.5

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

response.setHeader(name, value)

#

Added in: v0.4.0

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here if you need to send multiple headers with the same name.

Example:

```
response.setHeader('Content-Type', 'text/html');
```

or

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

Attempting to set a header field name or value that contains invalid characters will result in a [TypeError](#) being thrown.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// returns content-type = text/plain
const server = http.createServer((req,res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
```

```
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('ok');
});
```

response.setTimeout(msecs, callback)

Added in: v0.9.12

- `msecs <Number>`
- `callback <Function>`

Sets the Socket's timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then sockets are destroyed when they time out. If you assign a handler on the request, the response, or the server's `'timeout'` events, then it is your responsibility to handle timed out sockets.

Returns `response`.

response.statusCode

Added in: v0.4.0

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

Example:

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

response.statusMessage

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status message that will be sent to the client when the headers get flushed. If this is left as `undefined` then the standard message for the status code will be used.

Example:

```
response.statusMessage = 'Not found';
```

After response header was sent to the client, this property indicates the status message which was sent out.

`response.write(chunk[, encoding][, callback])`

#

Added in: v0.1.29

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'`. The last parameter `callback` will be called when this chunk of data is flushed.

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.js 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.

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 free again.

response.writeContinue()

Added in: v0.3.0

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the '`'checkContinue'`' event on `Server`.

response.writeHead(statusCode[, statusMessage][, headers])

Added in: v0.1.30

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 `statusMessage` as the second argument.

Example:

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': Buffer.byteLength(body),
  '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.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// returns content-type = text/plain
const server = http.createServer((req,res) => {
```

```
res.setHeader('Content-Type', 'text/html');
res.setHeader('X-Foo', 'bar');
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('ok');

});
```

Note that Content-Length is given in bytes not characters. The above example works because the string `'hello world'` contains only single byte characters. If the body contains higher coded characters then `Buffer.byteLength()` should be used to determine the number of bytes in a given encoding. And Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

Class: http.IncomingMessage

Added in: v0.1.17

An `IncomingMessage` object is created by `http.Server` or `http.ClientRequest` and passed as the first argument to the `'request'` and `'response'` event respectively. It may be used to access response status, headers and data.

It implements the `Readable Stream` interface, as well as the following additional events, methods, and properties.

Event: 'aborted'

Added in: v0.3.8

```
function () { }
```

Emitted when the request has been aborted by the client and the network socket has closed.

Event: 'close'

function () { }

Indicates that the underlying connection was closed. Just like `'end'`, this event occurs only once per response.

message.destroy([error])

#

Added in: v0.3.0

- `error <Error>`

Calls `destroy()` on the socket that received the `IncomingMessage`. If `error` is provided, an `'error'` event is emitted and `error` is passed as an argument to any listeners on the event.

message.headers

#

Added in: v0.1.5

The request/response headers object.

Key-value pairs of header names and values. Header names are lower-cased.

Example:

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

Duplicates in raw headers are handled in the following ways, depending on the header name:

- Duplicates of `age`, `authorization`, `content-length`, `content-type`, `etag`, `expires`, `from`, `host`, `if-modified-since`, `if-unmodified-since`, `last-modified`, `location`, `max-forwards`, `proxy-`

`authorization`, `referer`, `retry-after`, or `user-agent` are discarded.

- `set-cookie` is always an array. Duplicates are added to the array.
- For all other headers, the values are joined together with ','.

message.httpVersion

Added in: v0.1.1

In case of server request, the HTTP version sent by the client. In the case of client response, the HTTP version of the connected-to server. Probably either '`1.1`' or '`1.0`'.

Also `message.httpVersionMajor` is the first integer and `message.httpVersionMinor` is the second.

message.method

Added in: v0.1.1

Only valid for request obtained from `http.Server`.

The request method as a string. Read only. Example: '`GET`', '`DELETE`'.

message.rawHeaders

Added in: v0.11.6

The raw request/response headers list exactly as they were received.

Note that the keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values.

Header names are not lowercased, and duplicates are not merged.

```
// Prints something like:  
//  
// [ 'user-agent',  
//   'this is invalid because there can be only one',
```

```
//  'User-Agent',
//  'curl/7.22.0',
//  'Host',
//  '127.0.0.1:8000',
//  'ACCEPT',
//  '/*/*' ]
console.log(request.rawHeaders);
```

message.rawTrailers

Added in: v0.11.6

The raw request/response trailer keys and values exactly as they were received.
Only populated at the `'end'` event.

message.setTimeout(msecs, callback)

Added in: v0.5.9

- `msecs <Number>`
- `callback <Function>`

Calls `message.connection.setTimeout(msecs, callback)`.

Returns `message`.

message.statusCode

Added in: v0.1.1

Only valid for response obtained from `http.ClientRequest`.

The 3-digit HTTP response status code. E.G. `404`.

message.statusMessage

Added in: v0.11.10

Only valid for response obtained from `http.ClientRequest`.

The HTTP response status message (reason phrase). E.G. `OK` or `Internal Server`

message.socket

Added in: v0.3.0

The `net.Socket` object associated with the connection.

With HTTPS support, use `request.socket.getPeerCertificate()` to obtain the client's authentication details.

message.trailers

Added in: v0.3.0

The request/response trailers object. Only populated at the `'end'` event.

message.url

Added in: v0.1.90

Only valid for request obtained from `http.Server`.

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'
}
```

http.METHODS

Added in: v0.11.8

- `<Array>`

A list of the HTTP methods that are supported by the parser.

http.STATUS_CODES

Added in: v0.1.22

- `<Object>`

A collection of all the standard HTTP response status codes, and the short

description of each. For example, `http.STATUS_CODES[404] === 'Not Found'`.

http.createClient([port][, host])

Added in: v0.1.13 Deprecated since: v0.3.6

Stability: 0 - **Deprecated:** Use [http.request\(\)](#) instead.

Constructs a new HTTP client. `port` and `host` refer to the server to be connected to.

http.createServer([requestListener])

Added in: v0.1.13

Returns a new instance of [http.Server](#).

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

http.get(options[, callback])

Added in: v0.3.6

Since most requests are GET requests without bodies, Node.js provides this convenience method. The only difference between this method and [http.request\(\)](#) is that it sets the method to GET and calls `req.end()` automatically.

Example:

```
http.get('http://www.google.com/index.html', (res) => {
  console.log(`Got response: ${res.statusCode}`);
  // consume response body
  res.resume();
}).on('error', (e) => {
```

```
    console.log(`Got error: ${e.message}`);
});
```

http.globalAgent

Added in: v0.5.9

Global instance of Agent which is used as the default for all http client requests.

http.request(options[, callback])

Added in: v0.3.6

Node.js maintains several connections per server to make HTTP requests. This function allows one to transparently issue requests.

`options` can be an object or a string. If `options` is a string, it is automatically parsed with `url.parse()`.

Options:

- `protocol` : Protocol to use. Defaults to `'http:'`.
- `host` : A domain name or IP address of the server to issue the request to. Defaults to `'localhost'`.
- `hostname` : Alias for `host`. To support `url.parse()` `hostname` is preferred over `host`.
- `family` : IP address family to use when resolving `host` and `hostname`. Valid values are `4` or `6`. When unspecified, both IP v4 and v6 will be used.
- `port` : Port of remote server. Defaults to 80.
- `localAddress` : Local interface to bind for network connections.
- `socketPath` : Unix Domain Socket (use one of host:port or socketPath).
- `method` : A string specifying the HTTP request method. Defaults to `'GET'`.
- `path` : Request path. Defaults to `'/'`. Should include query string if any. E.G. `'/index.html?page=12'`. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may

change in the future.

- `headers` : An object containing request headers.
- `auth` : Basic authentication i.e. '`'user:password'`' to compute an Authorization header.
- `agent` : Controls `Agent` behavior. When an Agent is used request will default to `Connection: keep-alive`. Possible values:
 - `undefined` (default): use `http.globalAgent` for this host and port.
 - `Agent` object: explicitly use the passed in `Agent`.
 - `false` : opts out of connection pooling with an Agent, defaults request to `Connection: close`.
- `createConnection` : A function that produces a socket/stream to use for the request when the `agent` option is not used. This can be used to avoid creating a custom Agent class just to override the default `createConnection` function. See `agent.createConnection()` for more details.

The optional `callback` parameter will be added as a one time listener for the '`response`' event.

`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 postData = querystring.stringify({
  'msg' : 'Hello World!'
});
```

```
var options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
```

```

'Content-Type': 'application/x-www-form-urlencoded',
'Content-Length': Buffer.byteLength(postData)

}

};

var req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(`BODY: ${chunk}`);
  });
  res.on('end', () => {
    console.log('No more data in response.')
  })
});

req.on('error', (e) => {
  console.log(`problem with request: ${e.message}`);
});

// write data to request body
req.write(postData);
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. As with all `'error'` events, if no listeners are registered the error will be thrown.

There are a few special headers that should be noted.

- Sending a 'Connection: keep-alive' will notify Node.js that the connection to the server should be persisted until the next request.
- Sending a 'Content-length' header will disable the default chunked encoding.
- Sending an 'Expect' header will immediately send the request headers. Usually, when sending 'Expect: 100-continue', you should both set a timeout and listen for the 'continue' event. See RFC2616 Section 8.2.3 for more information.
- Sending an Authorization header will override using the `auth` option to compute basic authentication.

HTTPS

Stability: 2 - Stable

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

Class: https.Agent

Added in: v0.4.5

An Agent object for HTTPS similar to `http.Agent`. See `https.request()` for more information.

Class: https.Server

Added in: v0.3.4

This class is a subclass of `tls.Server` and emits events same as `http.Server`. See `http.Server` for more information.

server.setTimeout(msecs, callback)

Added in: v0.11.2

See [http.Server#setTimeout\(\)](#).

server.timeout

#

Added in: v0.11.2

See [http.Server#timeout](#).

https.createServer(options[, requestListener])

#

Added in: v0.3.4

Returns a new HTTPS web server object. The `options` is similar to [tls.createServer\(\)](#). The `requestListener` is a function which is automatically added to the `'request'` event.

Example:

```
// curl -k https://localhost:8000/
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Or

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

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

const options = {
  pfx: fs.readFileSync('server.pfx')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

server.close([callback])

Added in: v0.1.90

See [http.close\(\)](#) for details.

server.listen(handle[, callback])

server.listen(path[, callback])

server.listen(port[, host][, backlog][, callback])

See [http.listen\(\)](#) for details.

https.get(options, callback)

Added in: v0.3.6

Like [http.get\(\)](#) but for HTTPS.

`options` can be an object or a string. If `options` is a string, it is automatically parsed with [url.parse\(\)](#).

Example:

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

```
https.get('https://encrypted.google.com/', (res) => {
  console.log('statusCode: ', res.statusCode);
  console.log('headers: ', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });

}).on('error', (e) => {
  console.error(e);
});
```

https.globalAgent

Added in: v0.5.9

Global instance of `https.Agent` for all HTTPS client requests.

https.request(options, callback)

Added in: v0.3.6

Makes a request to a secure web server.

`options` can be an object or a string. If `options` is a string, it is automatically parsed with `url.parse()`.

All options from `http.request()` are valid.

Example:

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

var options = {
  hostname: 'encrypted.google.com',
  port: 443,
```

```

path: '/',
method: 'GET'

};

var req = https.request(options, (res) => {
  console.log('statusCode: ', res.statusCode);
  console.log('headers: ', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });
});

req.end();

req.on('error', (e) => {
  console.error(e);
});

```

The options argument has the following options

- `host` : A domain name or IP address of the server to issue the request to. Defaults to `'localhost'` .
- `hostname` : Alias for `host`. To support `url.parse()` `hostname` is preferred over `host` .
- `family` : IP address family to use when resolving `host` and `hostname` . Valid values are `4` or `6` . When unspecified, both IP v4 and v6 will be used.
- `port` : Port of remote server. Defaults to `443`.
- `localAddress` : Local interface to bind for network connections.
- `socketPath` : Unix Domain Socket (use one of host:port or socketPath).
- `method` : A string specifying the HTTP request method. Defaults to `'GET'` .
- `path` : Request path. Defaults to `'/'` . Should include query string if any. E.G. `'/index.html?page=12'` . An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may

change in the future.

- `headers` : An object containing request headers.
- `auth` : Basic authentication i.e. '`'user:password'`' to compute an Authorization header.
- `agent` : Controls `Agent` behavior. When an Agent is used request will default to `Connection: keep-alive`. Possible values:
 - `undefined` (default): use `globalAgent` for this host and port.
 - `Agent` object: explicitly use the passed in `Agent`.
 - `false` : opts out of connection pooling with an Agent, defaults request to `Connection: close`.

The following options from `tls.connect()` can also be specified. However, a `globalAgent` silently ignores these.

- `pfx` : Certificate, Private key and CA certificates to use for SSL. Default `null`.
- `key` : Private key to use for SSL. Default `null`.
- `passphrase` : A string of passphrase for the private key or pfx. Default `null`.
- `cert` : Public x509 certificate to use. Default `null`.
- `ca` : A string, `Buffer` or array of strings or `Buffers` of trusted certificates in PEM format. If this is omitted several well known "root" CAs will be used, like VeriSign. These are used to authorize connections.
- `ciphers` : A string describing the ciphers to use or exclude. Consult <https://www.openssl.org/docs/apps/ciphers.html#CIPHER-LIST-FORMAT> for details on the format.
- `rejectUnauthorized` : If `true`, the server certificate is verified against the list of supplied CAs. An '`error`' event is emitted if verification fails. Verification happens at the connection level, *before* the HTTP request is sent. Default `true`.
- `secureProtocol` : The SSL method to use, e.g. `SSLv3_method` to force SSL version 3. The possible values depend on your installation of OpenSSL and are defined in the constant `SSL_METHODS`.
- `servername` : Servername for SNI (Server Name Indication) TLS extension.

In order to specify these options, use a custom `Agent`.

Example:

```
var options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};
options.agent = new https.Agent(options);

var req = https.request(options, (res) => {
  ...
})
```

Alternatively, opt out of connection pooling by not using an `Agent`.

Example:

```
var options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

var req = https.request(options, (res) => {
  ...
})
```

Modules

#

Stability: 3 - Locked

Node.js has a simple module loading system. In Node.js, 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`:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

The contents of `circle.js`:

```
const PI = Math.PI;

exports.area = (r) => PI * r * r;

exports.circumference = (r) => 2 * PI * r;
```

The module `circle.js` has exported the functions `area()` and `circumference()`. To add functions and objects to the root of your module, you can add them to the special `exports` object.

Variables local to the module will be private, because the module is wrapped in a function by Node.js (see [module wrapper](#)). In this example, the variable `PI` is private to `circle.js`.

If you want the root of your module's export to be a function (such as a constructor) or if you want to export a complete object in one assignment instead of building it

one property at a time, assign it to `module.exports` instead of `exports`.

Below, `bar.js` makes use of the `square` module, which exports a constructor:

```
const square = require('./square.js');
var mySquare = square(2);
console.log(`The area of my square is ${mySquare.area()}`);
```

The `square` module is defined in `square.js`:

```
// assigning to exports will not modify module, must use module.exports
module.exports = (width) => {
  return {
    area: () => width * width
  };
}
```

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

Accessing the main module

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

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

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

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

Addenda: Package Manager Tips

#

The semantics of Node.js's `require()` function were designed to be general enough to support a number of reasonable directory structures. Package manager programs such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from Node.js 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.js 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 [here](#), 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.js will not bother looking for missing dependencies in `/usr/node_modules` or `/node_modules`.

In order to make modules available to the Node.js 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.

All Together...

#

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

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

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

```
LOAD_AS_FILE(X)
```

1. If X is a file, load X as JavaScript text. STOP
2. If X.js is a file, load X.js as JavaScript text. STOP

3. If X.json is a file, parse X.json to a JavaScript Object. STOP
4. If X.node is a file, load X.node as binary addon. STOP

LOAD_AS_DIRECTORY(X)

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

LOAD_NODE_MODULES(X, START)

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

NODE_MODULES_PATHS(START)

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

Caching

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

returned, if it would resolve to the same file.

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

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

Module Caching Caveats

#

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

Additionally, on case-insensitive file systems or operating systems, different resolved filenames can point to the same file, but the cache will still treat them as different modules and will reload the file multiple times. For example, `require('./foo')` and `require('./F00')` return two different objects, irrespective of whether or not `./foo` and `./F00` are the same file.

Core Modules

#

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

The core modules are defined within Node.js's source and are located 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.

Cycles

#

When there are circular `require()` calls, a module might not have finished executing when it is returned.

Consider this situation:

`a.js`:

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

`b.js`:

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

`main.js`:

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

When `main.js` loads `a.js`, then `a.js` in turn loads `b.js`. At that point, `b.js` tries to load `a.js`. In order to prevent an infinite loop, an **unfinished copy** of the

`a.js` exports object is returned to the `b.js` module. `b.js` then finishes loading, and its `exports` object is provided to the `a.js` module.

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

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

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

File Modules

If the exact filename is not found, then Node.js will attempt to load the required filename with the added extensions: `.js`, `.json`, and finally `.node`.

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

A required 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 required 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 must either be a core module or is loaded from a `node_modules` folder.

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

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.js's awareness of `package.json` files.

Note: If the file specified by the `"main"` entry of `package.json` is missing and can not be resolved, Node.js will report the entire module as missing with the default error:

```
Error: Cannot find module 'some-library'
```

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

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

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.js starts at the parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location. Node will not append `node_modules` to a path already ending in `node_modules`.

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

For example, if the file at `'/home/ry/projects/foo.js'` called `require('bar.js')`, then Node.js 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.

You can require specific files or sub modules distributed with a module by including a path suffix after the module name. For instance `require('example-module/path/to/file')` would resolve `path/to/file` relative to where `example-module` is located. The suffixed path follows the same module resolution semantics.

Loading from the global folders

If the `NODE_PATH` environment variable is set to a colon-delimited list of absolute paths, then Node.js will search those paths for modules if they are not found

elsewhere. (Note: On Windows, `NODE_PATH` is delimited by semicolons instead of colons.)

`NODE_PATH` was originally created to support loading modules from varying paths before the current [module resolution](#) algorithm was frozen.

`NODE_PATH` is still supported, but is less necessary now that the Node.js ecosystem has settled on a convention for locating dependent modules. Sometimes deployments that rely on `NODE_PATH` show surprising behavior when people are unaware that `NODE_PATH` must be set. Sometimes a module's dependencies change, causing a different version (or even a different module) to be loaded as the `NODE_PATH` is searched.

Additionally, Node.js will search in the following locations:

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

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

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

The module wrapper

Before a module's code is executed, Node.js will wrap it with a function wrapper that looks like the following:

```
(function (exports, require, module, __filename, __dirname) {  
  // Your module code actually lives in here  
});
```

By doing this, Node.js achieves a few things:

- It keeps top-level variables (defined with `var`, `const` or `let`) scoped to the module rather than the global object.
- It helps to provide some global-looking variables that are actually specific to the module, such as:
 - The `module` and `exports` objects that the implementor can use to export values from the module.
 - The convenience variables `__filename` and `__dirname`, containing the module's absolute filename and directory path.

The `module` Object

- `<Object>`

In each module, the `module` free variable is a reference to the object representing the current module. For convenience, `module.exports` is also accessible via the `exports` module-global. `module` isn't actually a global but rather local to each module.

`module.children`

- `<Array>`

The module objects required by this one.

`module.exports`

- `<Object>`

The `module.exports` object is created by the Module system. Sometimes this is not acceptable; many want their module to be an instance of some class. To do this, assign the desired export object to `module.exports`. Note that assigning the desired object to `exports` will simply rebind the local `exports` variable, which is probably not what you want to do.

For example suppose we were making a module called `a.js`

```
const EventEmitter = require('events');

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(() => {
  module.exports.emit('ready');
}, 1000);
```

Then in another file we could do

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

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

x.js:

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

y.js:

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

exports alias

The `exports` variable that is available within a module starts as a reference to `module.exports`. As with any variable, if you assign a new value to it, it is no longer bound to the previous value.

To illustrate the behavior, imagine this hypothetical implementation of `require()`:

```
function require(...) {
  // ...
  ((module, exports) => {
    // Your module code here
    exports = some_func;           // re-assigns exports, exports is no lon
                                  // a shortcut, and nothing is exported.
    module.exports = some_func; // makes your module export 0
  })(module, module.exports);
  return module;
}
```

As a guideline, if the relationship between `exports` and `module.exports` seems like magic to you, ignore `exports` and only use `module.exports`.

module.filename

- `<String>`

The fully resolved filename to the module.

module.id

- `<String>`

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

module.loaded

- `<Boolean>`

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

module.parent #

- <Object> Module object

The module that first required this one.

module.require(id) #

- id <String>
- Return: <Object> module.exports from the resolved module

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

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

net

Stability: 2 - Stable

The `net` module provides you with an asynchronous network wrapper. It contains functions for creating both servers and clients (called streams). You can include this module with `require('net');`.

Class: net.Server

Added in: v0.1.90

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

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

Event: 'close'

Added in: v0.5.0

Emitted when the server closes. Note that if connections exist, this event is not emitted until all connections are ended.

Event: 'connection'

#

Added in: v0.1.90

- `<net.Socket>` The connection object

Emitted when a new connection is made. `socket` is an instance of `net.Socket`.

Event: 'error'

#

Added in: v0.1.90

- `<Error>`

Emitted when an error occurs. The '`close`' event will be called directly following this event. See example in discussion of `server.listen`.

Event: 'listening'

#

Added in: v0.1.90

Emitted when the server has been bound after calling `server.listen`.

server.address()

#

Added in: v0.1.90

Returns the bound address, the address family name, and port of the server as reported by the operating system. Useful to find which port was assigned when getting an OS-assigned address. Returns an object with `port`, `family`, and `address` properties: `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

Example:

```
var server = net.createServer((socket) => {
  socket.end('goodbye\n');
}).on('error', (err) => {
  // handle errors here
```

```
    throw err;  
});  
  
// grab a random port.  
server.listen(() => {  
  address = server.address();  
  console.log('opened server on %j', address);  
});
```

Don't call `server.address()` until the '`'listening'`' event has been emitted.

server.close([callback])

Added in: v0.1.90

Stops the server from accepting new connections and keeps existing connections. This function is asynchronous, the server is finally closed when all connections are ended and the server emits a '`'close'`' event. The optional `callback` will be called once the '`'close'`' event occurs. Unlike that event, it will be called with an Error as its only argument if the server was not open when it was closed.

server.connections

Added in: v0.2.0 Deprecated since: v0.9.7

Stability: 0 - Deprecated: Use [server.getConnections\(\)](#) instead.

The number of concurrent connections on the server.

This becomes `null` when sending a socket to a child with `child_process.fork()`. To poll forks and get current number of active connections use asynchronous `server.getConnections` instead.

server.getConnections(callback)

Added in: v0.9.7

Asynchronously get the number of concurrent connections on the server. Works

when sockets were sent to forks.

Callback should take two arguments `err` and `count`.

`server.listen(handle[, backlog][, callback])`

#

Added in: v0.5.10

- `handle` `<Object>`
- `backlog` `<Number>`
- `callback` `<Function>`

The `handle` object can be set to either a server or socket (anything with an underlying `_handle` member), or a `{fd: <n>}` object.

This will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket.

Listening on a file descriptor is not supported on Windows.

This function is asynchronous. When the server has been bound, '`listening`' event will be emitted. The last parameter `callback` will be added as a listener for the '`listening`' event.

The parameter `backlog` behaves the same as in `server.listen(port[, hostname][, backlog][, callback])`.

`server.listen(options[, callback])`

#

Added in: v0.11.14

- `options` `<Object>` - Required. Supports the following properties:
 - `port` `<Number>` - Optional.
 - `host` `<String>` - Optional.
 - `backlog` `<Number>` - Optional.
 - `path` `<String>` - Optional.
 - `exclusive` `<Boolean>` - Optional.

- `callback` `<Function>` - Optional.

The `port`, `host`, and `backlog` properties of `options`, as well as the optional `callback` function, behave as they do on a call to `server.listen(port[, hostname][, backlog][, callback])`. Alternatively, the `path` option can be used to specify a UNIX socket.

If `exclusive` is `false` (default), then cluster workers will use the same underlying handle, allowing connection handling duties to be shared. When `exclusive` is `true`, the handle is not shared, and attempted port sharing results in an error. An example which listens on an exclusive port is shown below.

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

`server.listen(path[, backlog][, callback])`

#

Added in: v0.1.90

- `path` `<String>`
- `backlog` `<Number>`
- `callback` `<Function>`

Start a local socket server listening for connections on the given `path`.

This function is asynchronous. When the server has been bound, '`listening`' event will be emitted. The last parameter `callback` will be added as a listener for the '`listening`' event.

On UNIX, the local domain is usually known as the UNIX domain. The path is a filesystem path name. It gets truncated to `sizeof(sockaddr_un.sun_path)` bytes, decreased by 1. It varies on different operating system between 91 and 107 bytes. The typical values are 107 on Linux and 103 on OS X. The path is subject to the same

naming conventions and permissions checks as would be done on file creation, will be visible in the filesystem, and will *persist until unlinked*.

On Windows, the local domain is implemented using a named pipe. The path *must* refer to an entry in `\?\pipe\` or `\.\pipe\`. Any characters are permitted, but the latter may do some processing of pipe names, such as resolving `..` sequences. Despite appearances, the pipe name space is flat. Pipes will *not persist*, they are removed when the last reference to them is closed. Do not forget JavaScript string escaping requires paths to be specified with double-backslashes, such as:

```
net.createServer().listen(  
  path.join('\\\\?\\\\\\pipe', process.cwd(), 'myctl'))
```

The parameter `backlog` behaves the same as in `server.listen(port[, hostname][, backlog][, callback])`.

`server.listen(port[, hostname][, backlog][, callback])`

Added in: v0.1.90

Begin accepting connections on the specified `port` and `hostname`. If the `hostname` is omitted, the server will accept connections on any IPv6 address (`::`) when IPv6 is available, or any IPv4 address (`0.0.0.0`) otherwise. Use a port value of `0` to have the operating system assign an available port.

Backlog is the maximum length of the queue of pending connections. The actual length will be determined by the OS through sysctl settings such as `tcp_max_syn_backlog` and `somaxconn` on Linux. The default value of this parameter is 511 (not 512).

This function is asynchronous. When the server has been bound, '`listening`' event will be emitted. The last parameter `callback` will be added as a listener for the '`listening`' event.

One issue some users run into is getting `EADDRINUSE` errors. This means that another server is already running on the requested port. One way of handling this

would be to wait a second and then try again:

```
server.on('error', (e) => {
  if (e.code === 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

(Note: All sockets in Node.js are set `SO_REUSEADDR`.)

server.listening

Added in: v5.7.0

A Boolean indicating whether or not the server is listening for connections.

server.maxConnections

Added in: v0.2.0

Set this property to reject connections when the server's connection count gets high.

It is not recommended to use this option once a socket has been sent to a child with `child_process.fork()`.

server.ref()

Added in: v0.9.1

Opposite of `unref`, calling `ref` on a previously `unref`d server will not let the program exit if it's the only server left (the default behavior). If the server is `ref`d calling `ref` again will have no effect.

Returns `server`.

server.unref()

#

Added in: v0.9.1

Calling `unref` on a server will allow the program to exit if this is the only active server in the event system. If the server is already `unref`d calling `unref` again will have no effect.

Returns `server`.

Class: net.Socket

#

Added in: v0.3.4

This object is an abstraction of a TCP or local 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.js and passed to the user through the `'connection'` event of a server.

new net.Socket(options)

#

Added in: v0.3.4

Construct a new socket object.

`options` is an object with the following defaults:

```
{  
  fd: null,  
  allowHalfOpen: false,  
  readable: false,  
  writable: false  
}
```

`fd` allows you to specify the existing file descriptor of socket. Set `readable` and/or `writable` to `true` to allow reads and/or writes on this socket (NOTE: Works only when `fd` is passed). About `allowHalfOpen`, refer to `createServer()` and `'end'` event.

`net.Socket` instances are [EventEmitter](#) with the following events:

Event: 'close'

#

Added in: v0.1.90

- `had_error` <Boolean> `true` if the socket had a transmission error.

Emitted once the socket is fully closed. The argument `had_error` is a boolean which says if the socket was closed due to a transmission error.

Event: 'connect'

#

Added in: v0.1.90

Emitted when a socket connection is successfully established. See [connect\(\)](#).

Event: 'data'

#

Added in: v0.1.90

- <Buffer>

Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by `socket.setEncoding()`. (See the [Readable Stream](#) section for more information.)

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

Event: 'drain'

#

Added in: v0.1.90

Emitted when the write buffer becomes empty. Can be used to throttle uploads.

See also: the return values of `socket.write()`

Event: 'end'

#

Added in: v0.1.90

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: 'error'

#

Added in: v0.1.90

- `<Error>`

Emitted when an error occurs. The `'close'` event will be called directly following this event.

Event: 'lookup'

#

Added in: v0.11.3

Emitted after resolving the hostname but before connecting. Not applicable to UNIX sockets.

- `err` `<Error> | <Null>` The error object. See `dns.lookup()`.
- `address` `<String>` The IP address.
- `family` `<String> | <Null>` The address type. See `dns.lookup()`.
- `host` `<String>` The hostname.

Event: 'timeout'

#

Added in: v0.1.90

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()`

socket.address()

#

Added in: v0.1.90

Returns the bound address, the address family name and port of the socket as

reported by the operating system. Returns an object with three properties, e.g. {
port: 12346, family: 'IPv4', address: '127.0.0.1' }

socket.bufferSize

#

Added in: v0.3.8

`net.Socket` has the property that `socket.write()` always works. This is to help users get up and running quickly. The computer cannot always keep up with the amount of data that is written to a socket - the network connection simply might be too slow. Node.js 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.bytesRead

#

Added in: v0.5.3

The amount of received bytes.

socket.bytesWritten

#

Added in: v0.5.3

The amount of bytes sent.

socket.connect(options[, connectListener])

#

Added in: v0.1.90

Opens the connection for a given socket.

For TCP sockets, `options` argument should be an object which specifies:

- `port` : Port the client should connect to (Required).
- `host` : Host the client should connect to. Defaults to `'localhost'`.
- `localAddress` : Local interface to bind to for network connections.
- `localPort` : Local port to bind to for network connections.
- `family` : Version of IP stack. Defaults to `4`.
- `hints` : `dns.lookup()` hints. Defaults to `0`.
- `lookup` : Custom lookup function. Defaults to `dns.lookup`.

For local domain sockets, `options` argument should be an object which specifies:

- `path` : Path the client should connect to (Required).

Normally this method is not needed, as `net.createConnection` opens the socket. Use this only if you are implementing a custom Socket.

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

The `connectListener` parameter will be added as a listener for the '`connect`' event.

```
socket.connect(path[, connectListener]) #
```

```
socket.connect(port[, host][, connectListener]) #
```

Added in: v0.1.90

As `socket.connect(options[, connectListener])`, with options either as either `{port: port, host: host}` or `{path: path}`.

```
socket.connecting #
```

Added in: v6.1.0

If `true` - `socket.connect(options[, connectListener])` was called and haven't yet finished. Will be set to `false` before emitting `connect` event and/or calling `socket.connect(options[, connectListener])`'s callback.

`socket.destroy([exception])`

#

Added in: v0.1.90

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

If `exception` is specified, an '`error`' event will be emitted and any listeners for that event will receive `exception` as an argument.

`socket.destroyed`

#

A Boolean value that indicates if the connection is destroyed or not. Once a connection is destroyed no further data can be transferred using it.

`socket.end([data][, encoding])`

#

Added in: v0.1.90

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.localAddress`

#

Added in: v0.9.6

The string representation of the local IP address the remote client is connecting on. For example, if you are listening on `'0.0.0.0'` and the client connects on `'192.168.1.1'`, the value would be `'192.168.1.1'`.

`socket.localPort`

#

Added in: v0.9.6

The numeric representation of the local port. For example, `80` or `21`.

socket.pause() #

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

socket.ref() #

Added in: v0.9.1

Opposite of `unref`, calling `ref` on a previously `unref`d socket will *not* let the program exit if it's the only socket left (the default behavior). If the socket is `ref`d calling `ref` again will have no effect.

Returns `socket`.

socket.remoteAddress #

Added in: v0.5.10

The string representation of the remote IP address. For example, '`74.125.127.100`' or '`2001:4860:a005::68`'. Value may be `undefined` if the socket is destroyed (for example, if the client disconnected).

socket.remoteFamily #

Added in: v0.11.14

The string representation of the remote IP family. '`IPv4`' or '`IPv6`'.

socket.remotePort #

Added in: v0.5.10

The numeric representation of the remote port. For example, `80` or `21`.

socket.resume() #

Resumes reading after a call to `pause()`.

socket.setEncoding([encoding]) #

Added in: v0.1.90

Set the encoding for the socket as a [Readable Stream](#). See `stream.setEncoding()`

for more information.

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

Added in: v0.1.92

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

Set `initialDelay` (in milliseconds) to set the delay between the last data packet received and the first keepalive probe. Setting 0 for `initialDelay` will leave the value unchanged from the default (or previous) setting. Defaults to `0`.

Returns `socket`.

socket.setNoDelay([noDelay])

Added in: v0.1.90

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

Returns `socket`.

socket.setTimeout(timeout[, callback])

Added in: v0.1.90

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.

Returns `socket`.

`socket.unref()`

#

Added in: v0.9.1

Calling `unref` on a socket will allow the program to exit if this is the only active socket in the event system. If the socket is already `unref`d calling `unref` again will have no effect.

Returns `socket`.

`socket.write(data[, encoding][, callback])`

#

Added in: v0.1.90

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.

`net.connect(options[, connectListener])`

#

Added in: v0.7.0

A factory function, which returns a new `net.Socket` and automatically connects with the supplied `options`.

The options are passed to both the `net.Socket` constructor and the `socket.connect` method.

The `connectListener` parameter will be added as a listener for the '`'connect'`' event once.

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

```
const net = require('net');
const client = net.connect({port: 8124}, () => {
  // 'connect' listener
  console.log('connected to server!');
  client.write('world!\r\n');
});
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
client.on('end', () => {
  console.log('disconnected from server');
});
```

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

```
const client = net.connect({path: '/tmp/echo.sock'});
```

net.connect(path[, connectListener])

Added in: v0.1.90

A factory function, which returns a new unix `net.Socket` and automatically connects to the supplied `path`.

The `connectListener` parameter will be added as a listener for the '`connect`' event once.

net.connect(port[, host][, connectListener])

Added in: v0.1.90

A factory function, which returns a new `net.Socket` and automatically connects to

the supplied `port` and `host`.

If `host` is omitted, '`localhost`' will be assumed.

The `connectListener` parameter will be added as a listener for the '`connect`' event once.

net.createConnection(options[, connectListener])

Added in: v0.1.90

A factory function, which returns a new `net.Socket` and automatically connects with the supplied `options`.

The options are passed to both the `net.Socket` constructor and the `socket.connect` method.

The `connectListener` parameter will be added as a listener for the '`connect`' event once.

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

```
const net = require('net');
const client = net.createConnection({port: 8124}, () => {
  // 'connect' listener
  console.log('connected to server!');
  client.write('world!\r\n');
});
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
client.on('end', () => {
  console.log('disconnected from server');
});
```

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

```
const client = net.connect({path: '/tmp/echo.sock'});
```

net.createConnection(path[, connectListener])

Added in: v0.1.90

A factory function, which returns a new unix `net.Socket` and automatically connects to the supplied `path`.

The `connectListener` parameter will be added as a listener for the '`connect`' event once.

net.createConnection(port[, host][, connectListener])

Added in: v0.1.90

A factory function, which returns a new `net.Socket` and automatically connects to the supplied `port` and `host`.

If `host` is omitted, '`localhost`' will be assumed.

The `connectListener` parameter will be added as a listener for the '`connect`' event once.

net.createServer([options][, connectionListener])

Added in: v0.5.0

Creates a new server. The `connectionListener` argument is automatically set as a listener for the '`connection`' event.

`options` is an object with the following defaults:

```
{  
  allowHalfOpen: false,  
  pauseOnConnect: false  
}
```

If `allowHalfOpen` is `true`, then the socket won't automatically send a FIN packet when the other end of the socket sends a FIN packet. The socket becomes non-readable, but still writable. You should call the `end()` method explicitly. See '`end`' event for more information.

If `pauseOnConnect` is `true`, then the socket associated with each incoming connection will be paused, and no data will be read from its handle. This allows connections to be passed between processes without any data being read by the original process. To begin reading data from a paused socket, call `resume()`.

Here is an example of an echo server which listens for connections on port 8124:

```
const net = require('net');  
  
const server = net.createServer((c) => {  
  // 'connection' listener  
  console.log('client connected');  
  c.on('end', () => {  
    console.log('client disconnected');  
  });  
  c.write('hello\r\n');  
  c.pipe(c);  
});  
server.on('error', (err) => {  
  throw err;  
});  
server.listen(8124, () => {  
  console.log('server bound');  
});
```

Test this by using `telnet`:

```
telnet localhost 8124
```

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

```
server.listen('/tmp/echo.sock', () => {
  console.log('server bound');
});
```

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

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

net.isIP(input)

#

Added in: v0.3.0

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)

#

Added in: v0.3.0

Returns true if input is a version 4 IP address, otherwise returns false.

net.isIPv6(input)

#

Added in: v0.3.0

Returns true if input is a version 6 IP address, otherwise returns false.

Stability: 2 - Stable

The `os` module provides a number of operating system-related utility methods. It can be accessed using:

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

os.EOL

#

Added in: v0.7.8

A string constant defining the operating system-specific end-of-line marker:

- `\n` on POSIX
- `\r\n` on Windows

os.arch()

#

Added in: v0.5.0

The `os.arch()` method returns a string identifying the operating system CPU architecture *for which the Node.js binary was compiled*.

The current possible values are: `'arm'`, `'arm64'`, `'ia32'`, `'mips'`, `'mipsel'`, `'ppc'`, `'ppc64'`, `'s390'`, `'s390x'`, `'x32'`, `'x64'`, and `'x86'`.

Equivalent to `process.arch`.

os.constants

#

Returns an object containing commonly used operating system specific constants for error codes, process signals, and so on. The specific constants currently defined are described in [OS Constants](#).

os.cpus()

Added in: v0.3.3

The `os.cpus()` method returns an array of objects containing information about each CPU/core installed.

The properties included on each object include:

- `model <String>`
- `speed <number> (in MHz)`
- `times <Object>`
 - `user <number>` The number of milliseconds the CPU has spent in user mode.
 - `nice <number>` The number of milliseconds the CPU has spent in nice mode.
 - `sys <number>` The number of milliseconds the CPU has spent in sys mode.
 - `idle <number>` The number of milliseconds the CPU has spent in idle mode.
 - `irq <number>` The number of milliseconds the CPU has spent in irq mode.

For example:

```
[  
 {  
   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  
  }  
}  
]  
}
```

Note: Because `nice` values are UNIX-specific, on Windows the `nice` values of all processors are always 0.

os.endianness()

#

Added in: v0.9.4

The `os.endianness()` method returns a string identifying the endianness of the CPU *for which the Node.js binary was compiled*.

Possible values are:

- `'BE'` for big endian
- `'LE'` for little endian.

os.freemem()

#

Added in: v0.3.3

The `os.freemem()` method returns the amount of free system memory in bytes as an integer.

os.homedir()

#

The `os.homedir()` method returns the home directory of the current user as a string.

os.hostname()

#

Added in: v0.3.3

The `os.hostname()` method returns the hostname of the operating system as a string.

os.loadavg()

#

Added in: v0.3.3

The `os.loadavg()` method returns an array containing the 1, 5, and 15 minute load averages.

The load average is a measure of system activity, calculated by the operating system and expressed as a fractional number. As a rule of thumb, the load average should ideally be less than the number of logical CPUs in the system.

The load average is a UNIX-specific concept with no real equivalent on Windows platforms. On Windows, the return value is always `[0, 0, 0]`.

os.networkInterfaces()

#

Added in: v0.6.0

The `os.networkInterfaces()` method returns an object containing only network interfaces that have been assigned a network address.

Each key on the returned object identifies a network interface. The associated value is an array of objects that each describe an assigned network address.

The properties available on the assigned network address object include:

- `address` `<String>` The assigned IPv4 or IPv6 address

- `netmask` <String> The IPv4 or IPv6 network mask
- `family` <String> Either `IPv4` or `IPv6`
- `mac` <String> The MAC address of the network interface
- `internal` <boolean> `true` if the network interface is a loopback or similar interface that is not remotely accessible; otherwise `false`
- `scopeid` <number> The numeric IPv6 scope ID (only specified when `family` is `IPv6`)

```
{
  lo: [
    {
      address: '127.0.0.1',
      netmask: '255.0.0.0',
      family: 'IPv4',
      mac: '00:00:00:00:00:00',
      internal: true
    },
    {
      address: '::1',
      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
      family: 'IPv6',
      mac: '00:00:00:00:00:00',
      internal: true
    }
  ],
  eth0: [
    {
      address: '192.168.1.108',
      netmask: '255.255.255.0',
      family: 'IPv4',
      mac: '01:02:03:0a:0b:0c',
      internal: false
    },
    {
      address: 'fe80::10a:0b0c%eth0',
      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
      family: 'IPv6',
      mac: '01:02:03:0a:0b:0c',
      internal: false
    }
  ]
}
```

```
{  
    address: 'fe80::a00:27ff:fe4e:66a1',  
    netmask: 'ffff:ffff:ffff:ffff::',  
    family: 'IPv6',  
    mac: '01:02:03:0a:0b:0c',  
    internal: false  
}  
]  
}
```

os.platform()

#

Added in: v0.5.0

The `os.platform()` method returns a string identifying the operating system platform as set during compile time of Node.js.

Currently possible values are:

- `'aix'`
- `'darwin'`
- `'freebsd'`
- `'linux'`
- `'openbsd'`
- `'sunos'`
- `'win32'`

Equivalent to `process.platform`.

Note: The value `'android'` may also be returned if the Node.js is built on the Android operating system. However, Android support in Node.js is considered to be experimental at this time.

os.release()

#

Added in: v0.3.3

The `os.release()` method returns a string identifying the operating system release.

Note: On POSIX systems, the operating system release is determined by calling `uname(3)`. On Windows, `GetVersionExW()` is used. Please see <https://en.wikipedia.org/wiki/Uname#Examples> for more information.

os.tmpdir()

#

Added in: v0.9.9

The `os.tmpdir()` method returns a string specifying the operating system's default directory for temporary files.

os.totalmem()

#

Added in: v0.3.3

The `os.totalmem()` method returns the total amount of system memory in bytes as an integer.

os.type()

#

Added in: v0.3.3

The `os.type()` method returns a string identifying the operating system name as returned by `uname(3)`. For example '`Linux`' on Linux, '`Darwin`' on OS X and '`Windows_NT`' on Windows.

Please see <https://en.wikipedia.org/wiki/Uname#Examples> for additional information about the output of running `uname(3)` on various operating systems.

os.uptime()

#

Added in: v0.3.3

The `os.uptime()` method returns the system uptime in number of seconds.

Note: Within Node.js' internals, this number is represented as a `double`. However, fractional seconds are not returned and the value can typically be treated as an integer.

os.userInfo(options)

#

Added in: v6.0.0

- `options <Object>`
 - `encoding <String>` Character encoding used to interpret resulting strings. If `encoding` is set to `'buffer'`, the `username`, `shell`, and `homedir` values will be `Buffer` instances. (Default: `'utf8'`)

The `os.userInfo()` method returns information about the currently effective user -- on POSIX platforms, this is typically a subset of the password file. The returned object includes the `username`, `uid`, `gid`, `shell`, and `homedir`. On Windows, the `uid` and `gid` fields are `-1`, and `shell` is `null`.

The value of `homedir` returned by `os.userInfo()` is provided by the operating system. This differs from the result of `os.homedir()`, which queries several environment variables for the home directory before falling back to the operating system response.

OS Constants

#

The following constants are exported by `os.constants`. Note: Not all constants will be available on every operating system.

Signal Constants

#

The following signal constants are exported by `os.constants.signals`:

Constant	Description
SIGHUP	Sent to indicate when a controlling terminal is closed or a parent process exits.
	Sent to indicate when a user wishes to interrupt a process

SIGINT	((Ctrl+C)).
SIGQUIT	Sent to indicate when a user wishes to terminate a process and perform a core dump.
SIGILL	Sent to a process to notify that it has attempted to perform an illegal, malformed, unknown or privileged instruction.
SIGTRAP	Sent to a process when an exception has occurred.
SIGABRT	Sent to a process to request that it abort.
SIGIOT	Synonym for SIGABRT
SIGBUS	Sent to a process to notify that it has caused a bus error.
SIGFPE	Sent to a process to notify that it has performed an illegal arithmetic operation.
SIGKILL	Sent to a process to terminate it immediately.
SIGUSR1 SIGUSR2	Sent to a process to identify user-defined conditions.
SIGSEGV	Sent to a process to notify of a segmentation fault.
SIGPIPE	Sent to a process when it has attempted to write to a disconnected pipe.
SIGALRM	Sent to a process when a system timer elapses.
SIGTERM	Sent to a process to request termination.
SIGCHLD	Sent to a process when a child process terminates.

SIGSTKFLT	Sent to a process to indicate a stack fault on a coprocessor.
SIGCONT	Sent to instruct the operating system to continue a paused process.
SIGSTOP	Sent to instruct the operating system to halt a process.
SIGTSTP	Sent to a process to request it to stop.
SIGBREAK	Sent to indicate when a user wishes to interrupt a process.
SIGTTIN	Sent to a process when it reads from the TTY while in the background.
SIGTTOU	Sent to a process when it writes to the TTY while in the background.
SIGURG	Sent to a process when a socket has urgent data to read.
SIGXCPU	Sent to a process when it has exceeded its limit on CPU usage.
SIGXFSZ	Sent to a process when it grows a file larger than the maximum allowed.
SIGVTALRM	Sent to a process when a virtual timer has elapsed.
SIGPROF	Sent to a process when a system timer has elapsed.
SIGWINCH	Sent to a process when the controlling terminal has changed its size.
SIGIO	Sent to a process when I/O is available.
SIGPOLL	Synonym for SIGIO

SIGLOST	Sent to a process when a file lock has been lost.
SIGPWR	Sent to a process to notify of a power failure.
SIGINFO	Synonym for SIGPWR
SIGSYS	Sent to a process to notify of a bad argument.
SIGUNUSED	Synonym for SIGSYS

Error Constants

The following error constants are exported by `os.constants.errno`:

POSIX Error Constants

Constant	Description
E2BIG	Indicates that the list of arguments is longer than expected.
EACCES	Indicates that the operation did not have sufficient permissions.
EADDRINUSE	Indicates that the network address is already in use.
EADDRNOTAVAIL	Indicates that the network address is currently unavailable for use.
EAFNOSUPPORT	Indicates that the network address family is not supported.
EAGAIN	Indicates that there is currently no data available and to try the operation again later.
EALREADY	Indicates that the socket already has a pending connection in progress.

EBADF	Indicates that a file descriptor is not valid.
EBADMSG	Indicates an invalid data message.
EBUSY	Indicates that a device or resource is busy.
ECANCELED	Indicates that an operation was canceled.
ECHILD	Indicates that there are no child processes.
ECONNABORTED	Indicates that the network connection has been aborted.
ECONNREFUSED	Indicates that the network connection has been refused.
ECONNRESET	Indicates that the network connection has been reset.
EDEADLK	Indicates that a resource deadlock has been avoided.
EDESTADDRREQ	Indicates that a destination address is required.
EDOM	Indicates that an argument is out of the domain of the function.
EDQUOT	Indicates that the disk quota has been exceeded.
EEXIST	Indicates that the file already exists.
EFAULT	Indicates an invalid pointer address.
EFBIG	Indicates that the file is too large.

EHOSTUNREACH	Indicates that the host is unreachable.
EIDRM	Indicates that the identifier has been removed.
EILSEQ	Indicates an illegal byte sequence.
EINPROGRESS	Indicates that an operation is already in progress.
EINTR	Indicates that a function call was interrupted.
EINVAL	Indicates that an invalid argument was provided.
EIO	Indicates an otherwise unspecified I/O error.
EISCONN	Indicates that the socket is connected.
EISDIR	Indicates that the path is a directory.
ELLOOP	Indicates too many levels of symbolic links in a path.
EMFILE	Indicates that there are too many open files.
EMLINK	Indicates that there are too many hard links to a file.
EMSGSIZE	Indicates that the provided message is too long.
EMULTIHOP	Indicates that a multihop was attempted.
ENAMETOOLONG	Indicates that the filename is too long.
ENETDOWN	Indicates that the network is down.
ENETRESET	Indicates that the connection has been aborted by the network.

ENETUNREACH	Indicates that the network is unreachable.
ENFILE	Indicates too many open files in the system.
ENOBUFS	Indicates that no buffer space is available.
ENODATA	Indicates that no message is available on the stream head read queue.
ENODEV	Indicates that there is no such device.
ENOENT	Indicates that there is no such file or directory.
ENOEXEC	Indicates an exec format error.
ENOLCK	Indicates that there are no locks available.
ENOLINK	Indications that a link has been severed.
ENOMEM	Indicates that there is not enough space.
ENOMSG	Indicates that there is no message of the desired type.
ENOPROTOOPT	Indicates that a given protocol is not available.
ENOSPC	Indicates that there is no space available on the device.
ENOSR	Indicates that there are no stream resources available.
ENOSTR	Indicates that a given resource is not a stream.

ENOSYS	Indicates that a function has not been implemented.
ENOTCONN	Indicates that the socket is not connected.
ENOTDIR	Indicates that the path is not a directory.
ENOTEMPTY	Indicates that the directory is not empty.
ENOTSOCK	Indicates that the given item is not a socket.
ENOTSUP	Indicates that a given operation is not supported.
ENOTTY	Indicates an inappropriate I/O control operation.
ENXIO	Indicates no such device or address.
EOPNOTSUPP	Indicates that an operation is not supported on the socket. Note that while ENOTSUP and EOPNOTSUPP have the same value on Linux, according to POSIX.1 these error values should be distinct.)
EOVERFLOW	Indicates that a value is too large to be stored in a given data type.
EPERM	Indicates that the operation is not permitted.
EPIPE	Indicates a broken pipe.
EPROTO	Indicates a protocol error.
EPROTONOSUPPORT	Indicates that a protocol is not supported.
EPROTOTYPE	Indicates the wrong type of protocol for a socket.
	Indicates that the results are too large.

ERANGE	
EROFS	Indicates that the file system is read only.
ESPIPE	Indicates an invalid seek operation.
ESRCH	Indicates that there is no such process.
ESTALE	Indicates that the file handle is stale.
ETIME	Indicates an expired timer.
ETIMEDOUT	Indicates that the connection timed out.
ETXTBSY	Indicates that a text file is busy.
EWOULDBLOCK	Indicates that the operation would block.
EXDEV	Indicates an improper link.

Windows Specific Error Constants

#

The following error codes are specific to the Windows operating system:

Constant	Description
WSAEINTR	Indicates an interrupted function call.
WSAEBADF	Indicates an invalid file handle.
WSAEACCES	Indicates insufficient permissions to complete the operation.
WSAEFAULT	Indicates an invalid pointer address.

WSAEINVAL	Indicates that an invalid argument was passed.
WSAEMFILE	Indicates that there are too many open files.
WSAEWOULDBLOCK	Indicates that a resource is temporarily unavailable.
WSAEINPROGRESS	Indicates that an operation is currently in progress.
WSAEALREADY	Indicates that an operation is already in progress.
WSAENOTSOCK	Indicates that the resource is not a socket.
WSAEDESTADDRREQ	Indicates that a destination address is required.
WSAEMSGSIZE	Indicates that the message size is too long.
WSAEPROTOTYPE	Indicates the wrong protocol type for the socket.
WSAENOPROTOOPT	Indicates a bad protocol option.
WSAEPROTONOSUPPORT	Indicates that the protocol is not supported.
WSAESOCKTNOSUPPORT	Indicates that the socket type is not supported.
WSAEOPNOTSUPP	Indicates that the operation is not supported.
WSAEPFNOSUPPORT	Indicates that the protocol family is not supported.

WSAEAFNOSUPPORT	Indicates that the address family is not supported.
WSAEADDRINUSE	Indicates that the network address is already in use.
WSAEADDRNOTAVAIL	Indicates that the network address is not available.
WSAENETDOWN	Indicates that the network is down.
WSAENETUNREACH	Indicates that the network is unreachable.
WSAENETRESET	Indicates that the network connection has been reset.
WSAECONNABORTED	Indicates that the connection has been aborted.
WSAECONNRESET	Indicates that the connection has been reset by the peer.
WSAENOBUFS	Indicates that there is no buffer space available.
WSAEISCONN	Indicates that the socket is already connected.
WSAENOTCONN	Indicates that the socket is not connected.
WSAESHUTDOWN	Indicates that data cannot be sent after the socket has been shutdown.
WSAETOOMANYREFS	Indicates that there are too many references.
WSAETIMEDOUT	Indicates that the connection has timed out.

WSAECONNREFUSED	Indicates that the connection has been refused.
WSAELoop	Indicates that a name cannot be translated.
WSAENAMETOOLONG	Indicates that a name was too long.
WSAEHOSTDOWN	Indicates that a network host is down.
WSAEHOSTUNREACH	Indicates that there is no route to a network host.
WSAENOTEMPTY	Indicates that the directory is not empty.
WSAEPROCLIM	Indicates that there are too many processes.
WSAEUSERS	Indicates that the user quota has been exceeded.
WSAEDQUOT	Indicates that the disk quota has been exceeded.
WSAESTALE	Indicates a stale file handle reference.
WSAEREMOTE	Indicates that the item is remote.
WSASYSNOTREADY	Indicates that the network subsystem is not ready.
WSAVERNOTSUPPORTED	Indicates that the winsock.dll version is out of range.
WSANOTINITIALISED	Indicates that successful WSAStartup has not yet been performed.

WSAEDISCON	Indicates that a graceful shutdown is in progress.
WSAENOMORE	Indicates that there are no more results.
WSAECANCELLED	Indicates that an operation has been canceled.
WSAEINVALIDPROCTABLE	Indicates that the procedure call table is invalid.
WSAEINVALIDPROVIDER	Indicates an invalid service provider.
WSAEPROVIDERFAILEDINIT	Indicates that the service provider failed to initialized.
WSASYSCALLFAILURE	Indicates a system call failure.
WSASERVICE_NOT_FOUND	Indicates that a service was not found.
WSATYPE_NOT_FOUND	Indicates that a class type was not found.
WSA_E_NO_MORE	Indicates that there are no more results.
WSA_E_CANCELLED	Indicates that the call was canceled.
WSAEREFUSED	Indicates that a database query was refused.

libuv Constants

#

Constant	Description
UV_UDP_REUSEADDR	

Stability: 2 - Stable

The `path` module provides utilities for working with file and directory paths. It can be accessed using:

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

Windows vs. POSIX

#

The default operation of the `path` module varies based on the operating system on which a Node.js application is running. Specifically, when running on a Windows operating system, the `path` module will assume that Windows-style paths are being used.

For example, using the `path.basename()` function with the Windows file path `C:\\temp\\myfile.html`, will yield different results when running on POSIX than when run on Windows:

On POSIX:

```
path.basename('C:\\temp\\myfile.html');
// returns 'C:\\temp\\myfile.html'
```

On Windows:

```
path.basename('C:\\temp\\myfile.html');
// returns 'myfile.html'
```

To achieve consistent results when working with Windows file paths on any

operating system, use `path.win32`:

On POSIX and Windows:

```
path.win32.basename('C:\\temp\\myfile.html');
  // returns 'myfile.html'
```

To achieve consistent results when working with POSIX file paths on any operating system, use `path.posix`:

On POSIX and Windows:

```
path.posix.basename('/tmp/myfile.html');
  // returns 'myfile.html'
```

path.basename(path[, ext])

#

Added in: v0.1.25

- `path` `<String>`
- `ext` `<String>` An optional file extension

The `path.basename()` methods returns the last portion of a `path`, similar to the Unix `basename` command.

For example:

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

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
  // returns 'quux'
```

A `TypeError` is thrown if `path` is not a string or if `ext` is given and is not a string.

path.delimiter

#

Added in: v0.9.3

Provides the platform-specific path delimiter:

- ; for Windows
- : for POSIX

For example, on POSIX:

```
console.log(process.env.PATH)
// '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env PATH.split(path.delimiter)
// returns ['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

On Windows:

```
console.log(process.env.PATH)
// 'C:\Windows\system32;C:\Windows;C:\Program Files\node\' 

process.env PATH.split(path.delimiter)
// returns ['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\']
```

path.dirname(path)

#

Added in: v0.1.16

- `path <String>`

The `path.dirname()` method returns the directory name of a `path`, similar to the Unix `dirname` command.

For example:

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

A `TypeError` is thrown if `path` is not a string.

path.basename(path)

#

Added in: v0.1.25

- `path <String>`

The `path.basename()` method returns the extension of the `path`, from the last occurrence of the `.` (period) character to end of string in the last portion of the `path`. If there is no `.` in the last portion of the `path`, or if the first character of the basename of `path` (see `path.basename()`) is `.`, then an empty string is returned.

For example:

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

```
path.basename('index.coffee.md')
// returns '.md'
```

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

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

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

A `TypeError` is thrown if `path` is not a string.

path.format(pathObject)

Added in: v0.11.15

- `pathObject <Object>`
 - `dir <String>`
 - `root <String>`
 - `base <String>`
 - `name <String>`
 - `ext <String>`

The `path.format()` method returns a path string from an object. This is the opposite of `path.parse()`.

The following process is used when constructing the path string:

- `output` is set to an empty string.
- If `pathObject.dir` is specified, `pathObject.dir` is appended to `output` followed by the value of `path.sep`;
- Otherwise, if `pathObject.root` is specified, `pathObject.root` is appended to `output`.
- If `pathObject.base` is specified, `pathObject.base` is appended to `output`;
- Otherwise:
 - If `pathObject.name` is specified, `pathObject.name` is appended to `output`
 - If `pathObject.ext` is specified, `pathObject.ext` is appended to `output`.
- Return `output`

For example, on POSIX:

```
// If `dir` and `base` are provided,
// `${dir}${path.sep}${base}`
// will be returned.

path.format({
```

```
dir: '/home/user/dir',
base: 'file.txt'

});

// returns '/home/user/dir/file.txt'

// `root` will be used if `dir` is not specified.
// If only `root` is provided or `dir` is equal to `root` then the
// platform separator will not be included.
path.format({
  root: '/',
  base: 'file.txt'
});

// returns '/file.txt'

// `name` + `ext` will be used if `base` is not specified.
path.format({
  root: '/',
  name: 'file',
  ext: '.txt'
});

// returns '/file.txt'

// `base` will be returned if `dir` or `root` are not provided.
path.format({
  base: 'file.txt'
});

// returns 'file.txt'
```

On Windows:

```
path.format({
  root : "C:\\",
  dir : "C:\\path\\dir",
```

```
base : "file.txt",
ext : ".txt",
name : "file"
})
// returns 'C:\\path\\dir\\file.txt'
```

path.isAbsolute(path)

#

Added in: v0.11.2

- `path <String>`

The `path.isAbsolute()` method determines if `path` is an absolute path.

If the given `path` is a zero-length string, `false` will be returned.

For example on POSIX:

```
path.isAbsolute('/foo/bar') // true
path.isAbsolute('/baz/..') // true
path.isAbsolute('qux/') // false
path.isAbsolute('..') // false
```

On Windows:

```
path.isAbsolute('//server') // true
path.isAbsolute('C:/foo/..') // true
path.isAbsolute('bar\\baz') // false
path.isAbsolute('..') // false
```

A `TypeError` is thrown if `path` is not a string.

path.join([path[, ...]])

#

Added in: v0.1.16

- `[path[, ...]] <String>` A sequence of path segments

The `path.join()` method join all given `path` segments together using the platform specific separator as a delimiter, then normalizes the resulting path.

Zero-length `path` segments are ignored. If the joined path string is a zero-length string then `'.'` will be returned, representing the current working directory.

For example:

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '...')  
// returns '/foo/bar/baz/asdf'  
  
path.join('foo', {}, 'bar')  
// throws TypeError: Arguments to path.join must be strings
```

A `TypeError` is thrown if any of the path segments is not a string.

path.normalize(path)

#

Added in: v0.1.23

- `path <String>`

The `path.normalize()` method normalizes the given `path`, resolving `'..'` and `'.'` segments.

When multiple, sequential path segment separation characters are found (e.g. `/` on POSIX and `\` on Windows), they are replaced by a single instance of the platform specific path segment separator. Trailing separators are preserved.

If the `path` is a zero-length string, `'.'` is returned, representing the current working directory.

For example on POSIX:

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

```
// returns '/foo/bar/baz/asdf'
```

On Windows:

```
path.normalize('C:\\temp\\\\\\foo\\bar\\..\\');  
// returns 'C:\\temp\\foo\\'
```

A `TypeError` is thrown if `path` is not a string.

path.parse(path)

#

Added in: v0.11.15

- `path` `<String>`

The `path.parse()` method returns an object whose properties represent significant elements of the `path`.

The returned object will have the following properties:

- `root` `<String>`
- `dir` `<String>`
- `base` `<String>`
- `ext` `<String>`
- `name` `<String>`

For example on POSIX:

```
path.parse('/home/user/dir/file.txt')  
// returns  
// {  
//   root : "/",  
//   dir : "/home/user/dir",  
//   base : "file.txt",
```

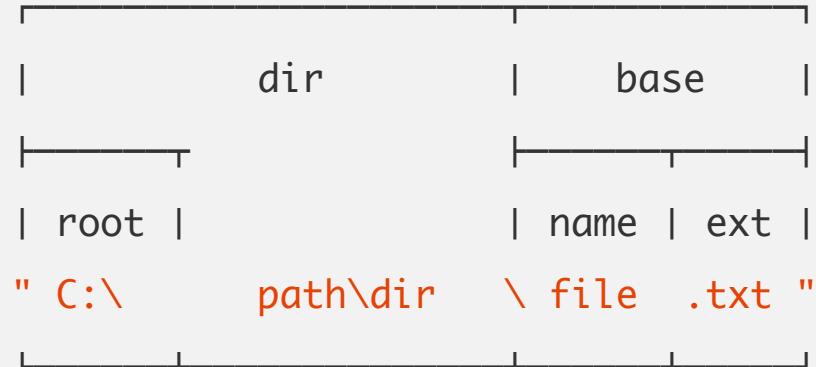
```
//   ext : ".txt",
//   name : "file"
// }
```



(all spaces in the "" line should be ignored -- they're purely for format)

On Windows:

```
path.parse('C:\\path\\dir\\file.txt')
// returns
// {
//   root : "C:\\",
//   dir : "C:\\path\\dir",
//   base : "file.txt",
//   ext : ".txt",
//   name : "file"
// }
```



(all spaces in the "" line should be ignored -- they're purely for format)

A `TypeError` is thrown if `path` is not a string.

path.posix

Added in: v0.11.15

The `path.posix` property provides access to POSIX specific implementations of the `path` methods.

path.relative(from, to)

Added in: v0.5.0

- `from <String>`
- `to <String>`

The `path.relative()` method returns the relative path from `from` to `to`. If `from` and `to` each resolve to the same path (after calling `path.resolve()` on each), a zero-length string is returned.

If a zero-length string is passed as `from` or `to`, the current working directory will be used instead of the zero-length strings.

For example on POSIX:

```
path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb')
// returns '../..../impl/bbb'
```

On Windows:

```
path.relative('C:\\\\orandea\\\\test\\\\aaa', 'C:\\\\orandea\\\\impl\\\\bbb')
// returns '...\\\\..\\\\impl\\\\bbb'
```

A `TypeError` is thrown if neither `from` nor `to` is a string.

path.resolve([path[, ...]])

#

Added in: v0.3.4

- [path[, ...]] <String> A sequence of paths or path segments

The `path.resolve()` method resolves a sequence of paths or path segments into an absolute path.

The given sequence of paths is processed from right to left, with each subsequent `path` prepended until an absolute path is constructed. For instance, given the sequence of path segments: `/foo`, `/bar`, `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz`.

If after processing all given `path` segments an absolute path has not yet been generated, the current working directory is used.

The resulting path is normalized and trailing slashes are removed unless the path is resolved to the root directory.

Zero-length `path` segments are ignored.

If no `path` segments are passed, `path.resolve()` will return the absolute path of the current working directory.

For example:

```
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 the current working directory is /home/myself/node,
// this returns '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

A [TypeError](#) is thrown if any of the arguments is not a string.

path.sep

#

Added in: v0.7.9

Provides the platform-specific path segment separator:

- \ on Windows
- / on POSIX

For example on POSIX:

```
'foo/bar/baz'.split(path.sep)  
// returns ['foo', 'bar', 'baz']
```

On Windows:

```
'foo\\bar\\baz'.split(path.sep)  
// returns ['foo', 'bar', 'baz']
```

path.win32

#

Added in: v0.11.15

The `path.win32` property provides access to Windows-specific implementations of the `path` methods.

process

#

The `process` object is a `global` that provides information about, and control over, the current Node.js process. As a global, it is always available to Node.js applications without using `require()`.

Process Events

Event: 'beforeExit'

#

Added in: v0.11.12

The '`beforeExit`' event is emitted when Node.js empties its event loop and has no additional work to schedule. Normally, the Node.js process will exit when there is no work scheduled, but a listener registered on the '`beforeExit`' event can make asynchronous calls, and thereby cause the Node.js process to continue.

The listener callback function is invoked with the value of `[process.exitCode][]` passed as the only argument.

The '`beforeExit`' event is *not* emitted for conditions causing explicit termination, such as calling `process.exit()` or uncaught exceptions.

The '`beforeExit`' should *not* be used as an alternative to the '`exit`' event unless the intention is to schedule additional work.

Event: 'disconnect'

#

Added in: v0.7.7

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the '`disconnect`' event will be emitted when the IPC channel is closed.

Event: 'exit'

#

Added in: v0.1.7

The '`exit`' event is emitted when the Node.js process is about to exit as a result of either:

- The `process.exit()` method being called explicitly;
- The Node.js event loop no longer having any additional work to perform.

There is no way to prevent the exiting of the event loop at this point, and once all '`exit`' listeners have finished running the Node.js process will terminate.

The listener callback function is invoked with the exit code specified either by the `[process.exitCode][][]` property, or the `exitCode` argument passed to the `process.exit()` method, as the only argument.

For example:

```
process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});
```

Listener functions **must** only perform **synchronous** operations. The Node.js process will exit immediately after calling the `'exit'` event listeners causing any additional work still queued in the event loop to be abandoned. In the following example, for instance, the timeout will never occur:

```
process.on('exit', (code) => {
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
});
```

Event: 'message'

Added in: v0.5.10

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `'message'` event is emitted whenever a message sent by a parent process using `childprocess.send()` is received by the child process.

The listener callback is invoked with the following arguments:

- `message <Object>` a parsed JSON object or primitive value
- `sendHandle <Handle object>` a `net.Socket` or `net.Server` object, or `undefined`.

Added in: v1.4.1

The '`rejectionHandled`' event is emitted whenever a `Promise` has been rejected and an error handler was attached to it (using `promise.catch()`, for example) later than one turn of the Node.js event loop.

The listener callback is invoked with a reference to the rejected `Promise` as the only argument.

The `Promise` object would have previously been emitted in an '`unhandledRejection`' event, but during the course of processing gained a rejection handler.

There is no notion of a top level for a `Promise` chain at which rejections can always be handled. Being inherently asynchronous in nature, a `Promise` rejection can be handled at a future point in time — possibly much later than the event loop turn it takes for the '`unhandledRejection`' event to be emitted.

Another way of stating this is that, unlike in synchronous code where there is an ever-growing list of unhandled exceptions, with Promises there can be a growing-and-shrinking list of unhandled rejections.

In synchronous code, the '`uncaughtException`' event is emitted when the list of unhandled exceptions grows.

In asynchronous code, the '`unhandledRejection`' event is emitted when the list of unhandled rejections grows, and the '`rejectionHandled`' event is emitted when the list of unhandled rejections shrinks.

For example:

```
const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, p) => {
  unhandledRejections.set(p, reason);
});
```

```
process.on('rejectionHandled', (p) => {
  unhandledRejections.delete(p);
});
```

In this example, the `unhandledRejections` `Map` will grow and shrink over time, reflecting rejections that start unhandled and then become handled. It is possible to record such errors in an error log, either periodically (which is likely best for long-running application) or upon process exit (which is likely most convenient for scripts).

Event: 'uncaughtException'

Added in: v0.1.18

The `'uncaughtException'` event is emitted when an exception bubbles all the way back to the event loop. By default, Node.js handles such exceptions by printing the stack trace to `stderr` and exiting. Adding a handler for the `'uncaughtException'` event overrides this default behavior.

The listener function is called with the `Error` object passed as the only argument.

For example:

```
process.on('uncaughtException', (err) => {
  console.log(`Caught exception: ${err}`);
});

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

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

Warning: Using 'uncaughtException' correctly

Note that 'uncaughtException' is a crude mechanism for exception handling intended to be used only as a last resort. The event *should not* be used as an equivalent to `On Error Resume Next`. Unhandled exceptions inherently mean that an application is in an undefined state. Attempting to resume application code without properly recovering from the exception can cause additional unforeseen and unpredictable issues.

Exceptions thrown from within the event handler will not be caught. Instead the process will exit with a non-zero exit code and the stack trace will be printed. This is to avoid infinite recursion.

Attempting to resume normally after an uncaught exception can be similar to pulling out of the power cord when upgrading a computer -- nine out of ten times nothing happens - but the 10th time, the system becomes corrupted.

The correct use of 'uncaughtException' is to perform synchronous cleanup of allocated resources (e.g. file descriptors, handles, etc) before shutting down the process. It is not safe to resume normal operation after 'uncaughtException' .

Event: 'unhandledRejection'

Added in: v1.4.1

The 'unhandledRejection' event is emitted whenever a `Promise` is rejected and no error handler is attached to the promise within a turn of the event loop. When programming with Promises, exceptions are encapsulated as "rejected promises". Rejections can be caught and handled using `promise.catch()` and are propagated through a `Promise` chain. The 'unhandledRejection' event is useful for detecting and keeping track of promises that were rejected whose rejections have not yet been handled.

The listener function is called with the following arguments:

- `reason <Error> | <any>` The object with which the promise was rejected (typically an `Error` object).
- `p` the `Promise` that was rejected.

For example:

```
process.on('unhandledRejection', (reason, p) => {
  console.log("Unhandled Rejection at: Promise ", p, " reason: ", reason);
  // application specific logging, throwing an error, or other logic here
});

somePromise.then((res) => {
  return reportToUser(JSON.parse(res)); // note the typo (`pasre`)
}); // no `.catch` or `.then`
```

The following will also trigger the `'unhandledRejection'` event to be emitted:

```
function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

var resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

In this example case, it is possible to track the rejection as a developer error as would typically be the case for other `'unhandledRejection'` events. To address such failures, a non-operational `.catch(() => {})` handler may be attached to `resource.loaded`, which would prevent the `'unhandledRejection'` event from being emitted. Alternatively, the `'rejectionHandled'` event may be used.

Event: 'warning'

#

Added in: v6.0.0

The `'warning'` event is emitted whenever Node.js emits a process warning.

A process warning is similar to an error in that it describes exceptional conditions

that are being brought to the user's attention. However, warnings are not part of the normal Node.js and JavaScript error handling flow. Node.js can emit warnings whenever it detects bad coding practices that could lead to sub-optimal application performance, bugs or security vulnerabilities.

The listener function is called with a single `warning` argument whose value is an `Error` object. There are three key properties that describe the warning:

- `name <String>` The name of the warning (currently `Warning` by default).
- `message <String>` A system-provided description of the warning.
- `stack <String>` A stack trace to the location in the code where the warning was issued.

```
process.on('warning', (warning) => {
  console.warn(warning.name);    // Print the warning name
  console.warn(warning.message); // Print the warning message
  console.warn(warning.stack);   // Print the stack trace
});
```

By default, Node.js will print process warnings to `stderr`. The `--no-warnings` command-line option can be used to suppress the default console output but the `'warning'` event will still be emitted by the `process` object.

The following example illustrates the warning that is printed to `stderr` when too many listeners have been added to an event

```
$ node
> event.defaultMaxListeners = 1;
> process.on('foo', () => {});
> process.on('foo', () => {});
> (node:38638) Warning: Possible EventEmitter memory leak detected. 2 fo
... listeners added. Use emitter.setMaxListeners() to increase limit
```

In contrast, the following example turns off the default warning output and adds a custom handler to the 'warning' event:

```
$ node --no-warnings
> var p = process.on('warning', (warning) => console.warn('Do not do tha
> event.defaultMaxListeners = 1;
> process.on('foo', () => {});
> process.on('foo', () => {});
> Do not do that!
```

The `--trace-warnings` command-line option can be used to have the default console output for warnings include the full stack trace of the warning.

Emitting custom warnings

The `process.emitWarning()` method can be used to issue custom or application specific warnings.

```
// Emit a warning using a string...
process.emitWarning('Something happened!');
// Prints: (node 12345) Warning: Something happened!

// Emit a warning using an object...
process.emitWarning('Something Happened!', 'CustomWarning');
// Prints: (node 12345) CustomWarning: Something happened!

// Emit a warning using a custom Error object...
class CustomWarning extends Error {
  constructor(message) {
    super(message);
    this.name = 'CustomWarning';
    Error.captureStackTrace(this, CustomWarning);
  }
}
```

```
}  
const myWarning = new CustomWarning('Something happened!');  
process.emitWarning(myWarning);  
// Prints: (node 12345) CustomWarning: Something happened!
```

Emitting custom deprecation warnings

Custom deprecation warnings can be emitted by setting the `name` of a custom warning to `DeprecationWarning`. For instance:

```
process.emitWarning('This API is deprecated', 'DeprecationWarning');
```

Or,

```
const err = new Error('This API is deprecated');  
err.name = 'DeprecationWarning';  
process.emitWarning(err);
```

Launching Node.js using the `--throw-deprecation` command line flag will cause custom deprecation warnings to be thrown as exceptions.

Using the `--trace-deprecation` command line flag will cause the custom deprecation to be printed to `stderr` along with the stack trace.

Using the `--no-deprecation` command line flag will suppress all reporting of the custom deprecation.

The `*-deprecation` command line flags only affect warnings that use the name `DeprecationWarning`.

Signal Events

Signal events will be emitted when the Node.js process receives a signal. Please refer to [sigaction\(2\)](#) for a listing of standard POSIX signal names such as `SIGINT`,

SIGHUP , etc.

The name of each event will be the uppercase common name for the signal (e.g. 'SIGINT' for SIGINT signals).

For example:

```
// Begin reading from stdin so the process does not exit.  
process.stdin.resume();  
  
process.on('SIGINT', () => {  
  console.log('Received SIGINT. Press Control-D to exit.');//  
});
```

Note: An easy way to send the SIGINT signal is with <Ctrl>-C in most terminal programs.

It is important to take note of the following:

- SIGUSR1 is reserved by Node.js to start the debugger. It's possible to install a listener but doing so will not stop the debugger from starting.
- SIGTERM and SIGINT have default handlers on non-Windows platforms that resets the terminal mode before exiting with code 128 + signal number . If one of these signals has a listener installed, its default behavior will be removed (Node.js will no longer exit).
- SIGPIPE is ignored by default. It can have a listener installed.
- SIGHUP is generated on Windows when the console window is closed, and on other platforms under various similar conditions, see [signal\(7\)](#) . It can have a listener installed, however Node.js will be unconditionally terminated by Windows about 10 seconds later. On non-Windows platforms, the default behavior of SIGHUP is to terminate Node.js, but once a listener has been installed its default behavior will be removed.
- SIGTERM is not supported on Windows, it can be listened on.
- SIGINT from the terminal is supported on all platforms, and can usually be generated with CTRL+C (though this may be configurable). It is not generated

when terminal raw mode is enabled.

- `SIGBREAK` is delivered on Windows when `<Ctrl>+<Break>` is pressed, on non-Windows platforms it can be listened on, but there is no way to send or generate it.
- `SIGWINCH` is delivered when the console has been resized. On Windows, this will only happen on write to the console when the cursor is being moved, or when a readable tty is used in raw mode.
- `SIGKILL` cannot have a listener installed, it will unconditionally terminate Node.js on all platforms.
- `SIGSTOP` cannot have a listener installed.

Note: Windows does not support sending signals, but Node.js offers some emulation with `process.kill()`, and `ChildProcess.kill()`. Sending signal `0` can be used to test for the existence of a process. Sending `SIGINT`, `SIGTERM`, and `SIGKILL` cause the unconditional termination of the target process.

process.abort()

Added in: v0.7.0

The `process.abort()` method causes the Node.js process to exit immediately and generate a core file.

process.arch

Added in: v0.5.0

The `process.arch` property returns a String identifying the processor architecture that the Node.js process is currently running on. For instance `'arm'`, `'ia32'`, or `'x64'`.

```
console.log(`This processor architecture is ${process.arch}`);
```

process.argv

Added in: v0.1.27

The `process.argv` property returns an array containing the command line arguments passed when the Node.js process was launched. The first element will be `process.execPath`. The second element will be the path to the JavaScript file being executed. The remaining elements will be any additional command line arguments.

For example, assuming the following script for `process-args.js`:

```
// print process.argv
process.argv.forEach((val, index) => {
  console.log(`#${index}: ${val}`);
});
```

Launching the Node.js process as:

```
$ node process-2.js one two=three four
```

Would generate the output:

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

process.chdir(directory)

#

Added in: v0.1.17

- `directory <String>`

The `process.chdir()` method changes the current working directory of the Node.js process or throws an exception if doing so fails (for instance, if the specified `directory` does not exist).

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

process.config

#

Added in: v0.7.7

The `process.config` property returns an Object containing the JavaScript representation of the configure options used to compile the current Node.js executable. This is the same as the `config.gypi` file that was produced when running the `./configure` script.

An example of the possible output looks like:

```
{
  target_defaults:
  { cflags: [],
    default_configuration: 'Release',
    defines: [],
    include_dirs: [],
    libraries: [] },
  variables:
  {
    host_arch: 'x64',
    node_install_npm: 'true',
    node_prefix: '',
    node_shared_cares: 'false',
```

```
    node_shared_http_parser: 'false',
    node_shared_libuv: 'false',
    node_shared_zlib: 'false',
    node_use_dtrace: 'false',
    node_use_openssl: 'true',
    node_shared_openssl: 'false',
    strict_aliasing: 'true',
    target_arch: 'x64',
    v8_use_snapshot: 'true'

}
```

}

Note: The `process.config` property is **not** read-only and there are existing modules in the ecosystem that are known to extend, modify, or entirely replace the value of `process.config`.

process.connected

Added in: v0.7.2

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `process.connected` property will return `true` so long as the IPC channel is connected and will return `false` after `process.disconnect()` is called.

Once `process.connected` is `false`, it is no longer possible to send messages over the IPC channel using `process.send()`.

process.cpuUsage([previousValue])

Added in: v6.1.0

- `previousValue <Array>` A previous return value from calling `process.cpuUsage()`

The `process.cpuUsage()` method returns the user and system CPU time usage of the current process, in an object with properties `user` and `system`, whose values

are microsecond values (millionth of a second). These values measure time spent in user and system code respectively, and may end up being greater than actual elapsed time if multiple CPU cores are performing work for this process.

The result of a previous call to `process.cpuUsage()` can be passed as the argument to the function, to get a diff reading.

```
const startUsage = process.cpuUsage();
// { user: 38579, system: 6986 }

// spin the CPU for 500 milliseconds
const now = Date.now();
while (Date.now() - now < 500);

console.log(process.cpuUsage(startUsage));
// { user: 514883, system: 11226 }
```

process.cwd()

#

Added in: v0.1.8

The `process.cwd()` method returns the current working directory of the Node.js process.

```
console.log(`Current directory: ${process.cwd()}`);
```

process.disconnect()

#

Added in: v0.7.2

If the Node.js process is spawned with an IPC channel (see the [Child Process](#) and [Cluster](#) documentation), the `process.disconnect()` method will close the IPC channel to the parent process, allowing the child process to exit gracefully once there are no other connections keeping it alive.

The effect of calling `process.disconnect()` is that same as calling the parent process's `ChildProcess.disconnect()`.

If the Node.js process was not spawned with an IPC channel, `process.disconnect()` will be `undefined`.

process.env

#

Added in: v0.1.27

The `process.env` property returns an object containing the user environment. See [environ\(7\)](#).

An example of this object looks like:

```
{  
  TERM: 'xterm-256color',  
  SHELL: '/usr/local/bin/bash',  
  USER: 'maciej',  
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',  
  PWD: '/Users/maciej',  
  EDITOR: 'vim',  
  SHLVL: '1',  
  HOME: '/Users/maciej',  
  LOGNAME: 'maciej',  
  _: '/usr/local/bin/node'  
}
```

It is possible to modify this object, but such modifications will not be reflected outside the Node.js process. In other words, the following example would not work:

```
$ node -e 'process.env.foo = "bar"' && echo $foo
```

While the following will:

```
process.env.foo = 'bar';
console.log(process.env.foo);
```

Assigning a property on `process.env` will implicitly convert the value to a string.

Example:

```
process.env.test = null;
console.log(process.env.test);
// => 'null'
process.env.test = undefined;
console.log(process.env.test);
// => 'undefined'
```

Use `delete` to delete a property from `process.env`.

Example:

```
process.env.TEST = 1;
delete process.env.TEST;
console.log(process.env.TEST);
// => undefined
```

process.emitWarning(warning[, name][, ctor])

Added in: v6.0.0

- `warning` `<String>` | `<Error>` The warning to emit.
- `name` `<String>` When `warning` is a String, `name` is the name to use for the warning. Default: `Warning`.
- `ctor` `<Function>` When `warning` is a String, `ctor` is an optional function used to limit the generated stack trace. Default `process.emitWarning`

The `process.emitWarning()` method can be used to emit custom or application specific process warnings. These can be listened for by adding a handler to the `process.on('warning')` event.

```
// Emit a warning using a string...
process.emitWarning('Something happened!');
// Emits: (node: 56338) Warning: Something happened!
```

```
// Emit a warning using a string and a name...
process.emitWarning('Something Happened!', 'CustomWarning');
// Emits: (node:56338) CustomWarning: Something Happened!
```

In each of the previous examples, an `Error` object is generated internally by `process.emitWarning()` and passed through to the `process.on('warning')` event.

```
process.on('warning', (warning) => {
  console.warn(warning.name);
  console.warn(warning.message);
  console.warn(warning.stack);
});
```

If `warning` is passed as an `Error` object, it will be passed through to the `process.on('warning')` event handler unmodified (and the optional `name` and `ctor` arguments will be ignored):

```
// Emit a warning using an Error object...
const myWarning = new Error('Warning! Something happened!');
myWarning.name = 'CustomWarning';

process.emitWarning(myWarning);
```

```
// Emits: (node:56338) CustomWarning: Warning! Something Happened!
```

A `TypeError` is thrown if `warning` is anything other than a string or `Error` object.

Note that while process warnings use `Error` objects, the process warning mechanism is **not** a replacement for normal error handling mechanisms.

The following additional handling is implemented if the warning `name` is `DeprecationWarning`:

- If the `--throw-deprecation` command-line flag is used, the deprecation warning is thrown as an exception rather than being emitted as an event.
- If the `--no-deprecation` command-line flag is used, the deprecation warning is suppressed.
- If the `--trace-deprecation` command-line flag is used, the deprecation warning is printed to `stderr` along with the full stack trace.

Avoiding duplicate warnings

As a best practice, warnings should be emitted only once per process. To do so, it is recommended to place the `emitWarning()` behind a simple boolean flag as illustrated in the example below:

```
var warned = false;
function emitMyWarning() {
  if (!warned) {
    process.emitWarning('Only warn once!');
    warned = true;
  }
}
emitMyWarning();
// Emits: (node: 56339) Warning: Only warn once!
emitMyWarning();
// Emits nothing
```

process.execArgv

#

Added in: v0.7.7

The `process.execArgv` property returns the set of Node.js-specific command-line options passed when the Node.js process was launched. These options do not appear in the array returned by the `process.argv` property, and do not include the Node.js executable, the name of the script, or any options following the script name. These options are useful in order to spawn child processes with the same execution environment as the parent.

For example:

```
$ node --harmony script.js --version
```

Results in `process.execArgv`:

```
['--harmony']
```

And `process.argv`:

```
['/usr/local/bin/node', 'script.js', '--version']
```

process.execPath

#

Added in: v0.1.100

The `process.execPath` property returns the absolute pathname of the executable that started the Node.js process.

For example:

process.exit([code])

Added in: v0.1.13

- `code <Integer>` The exit code. Defaults to `0`.

The `process.exit()` method instructs Node.js to terminate the process as quickly as possible with the specified exit `code`. If the `code` is omitted, `exit` uses either the 'success' code `0` or the value of `process.exitCode` if specified.

To exit with a 'failure' code:

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

The shell that executed Node.js should see the exit code as `1`.

It is important to note that calling `process.exit()` will force the process to exit as quickly as possible *even if there are still asynchronous operations pending* that have not yet completed fully, *including I/O operations to process.stdout and process.stderr*.

In most situations, it is not actually necessary to call `process.exit()` explicitly. The Node.js process will exit on its own *if there is no additional work pending* in the event loop. The `process.exitCode` property can be set to tell the process which exit code to use when the process exits gracefully.

For instance, the following example illustrates a *misuse* of the `process.exit()` method that could lead to data printed to `stdout` being truncated and lost:

```
// This is an example of what *not* to do:  
if (someConditionNotMet()) {  
    printUsageToStdout();  
}
```

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

The reason this is problematic is because writes to `process.stdout` in Node.js are usually *non-blocking* and may occur over multiple ticks of the Node.js event loop. Calling `process.exit()`, however, forces the process to exit *before* those additional writes to `stdout` can be performed.

Rather than calling `process.exit()` directly, the code *should* set the `process.exitCode` and allow the process to exit naturally by avoiding scheduling any additional work for the event loop:

```
// How to properly set the exit code while letting
// the process exit gracefully.
if (someConditionNotMet()) {
  printUsageToStdout();
  process.exitCode = 1;
}
```

If it is necessary to terminate the Node.js process due to an error condition, throwing an *uncaught* error and allowing the process to terminate accordingly is safer than calling `process.exit()`.

process.exitCode

#

Added in: v0.11.8

A number which will be the process exit code, when the process either exits gracefully, or is exited via `process.exit()` without specifying a code.

Specifying a code to `process.exit(code)` will override any previous setting of `process.exitCode`.

process.getegid()

#

Added in: v2.0.0

The `process.getegid()` method returns the numerical effective group identity of the Node.js process. (See [getegid\(2\)](#).)

```
if (process.getegid) {  
  console.log(`Current gid: ${process.getegid()}`);  
}
```

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

process.geteuid()

#

Added in: v2.0.0

The `process.geteuid()` method returns the numerical effective user identity of the process. (See [geteuid\(2\)](#).)

```
if (process.geteuid) {  
  console.log(`Current uid: ${process.geteuid()}`);  
}
```

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

process.getgid()

#

Added in: v0.1.31

The `process.getgid()` method returns the numerical group identity of the process. (See [getgid\(2\)](#).)

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

```
}
```

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

process.getgroups()

#

Added in: v0.9.4

The `process.getgroups()` method returns an array with the supplementary group IDs. POSIX leaves it unspecified if the effective group ID is included but Node.js ensures it always is.

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

process.getuid()

#

Added in: v0.1.28

The `process.getuid()` method returns the numeric user identity of the process. (See [getuid\(2\)](#).)

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

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

process.hrtime([time])

#

Added in: v0.7.6

The `process.hrtime()` method returns the current high-resolution real time in a `[seconds, nanoseconds]` tuple Array. `time` is an optional parameter that must

be the result of a previous `process.hrtime()` call (and therefore, a real time in a `[seconds, nanoseconds]` tuple Array containing a previous time) to diff with the current time. These times are relative to an arbitrary time in the past, and not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals.

Passing in the result of a previous call to `process.hrtime()` is useful for calculating an amount of time passed between calls:

```
var time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  var diff = process.hrtime(time);
  // [ 1, 552 ]

  console.log(`Benchmark took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
  // benchmark took 1000000527 nanoseconds
}, 1000);
```

Constructing an array by some method other than calling `process.hrtime()` and passing the result to `process.hrtime()` will result in undefined behavior.

process.initgroups(user, extra_group)

#

Added in: v0.9.4

- `user` `<String>` | `<number>` The user name or numeric identifier.
- `extra_group` `<String>` | `<number>` A group name or numeric identifier.

The `process.initgroups()` method reads the `/etc/group` file and initializes the group access list, using all groups of which the user is a member. This is a privileged operation that requires that the Node.js process either have `root` access or the `CAP_SETGID` capability.

Note that care must be taken when dropping privileges. Example:

```
console.log(process.getgroups());          // [ 0 ]
process.initgroups('bnoordhuis', 1000);    // switch user
console.log(process.getgroups());          // [ 27, 30, 46, 1000, 0 ]
process.setgid(1000);                     // drop root gid
console.log(process.getgroups());          // [ 27, 30, 46, 1000 ]
```

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

process.kill(pid[, signal])

#

Added in: v0.0.6

- `pid` <number> A process ID
- `signal` <String> | <number> The signal to send, either as a string or number. Defaults to `'SIGTERM'`.

The `process.kill()` method sends the `signal` to the process identified by `pid`.

Signal names are strings such as `'SIGINT'` or `'SIGHUP'`. See [Signal Events](#) and [kill\(2\)](#) for more information.

This method will throw an error if the target `pid` does not exist. As a special case, a signal of `0` can be used to test for the existence of a process. Windows platforms will throw an error if the `pid` is used to kill a process group.

Note: Even though 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.

For example:

```
process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
```

```
});  
  
setTimeout(() => {  
  console.log('Exiting.');//  
  process.exit(0);  
}, 100);  
  
process.kill(process.pid, 'SIGHUP');
```

Note: When `SIGUSR1` is received by a Node.js process, Node.js will start the debugger, see [Signal Events](#).

process.mainModule

Added in: v0.1.17

The `process.mainModule` property provides an alternative way of retrieving `require.main`. The difference is that if the main module changes at runtime, `require.main` may still refer to the original main module in modules that were required before the change occurred. Generally it's safe to assume that the two refer to the same module.

As with `require.main`, `process.mainModule` will be `undefined` if there is no entry script.

process.memoryUsage()

Added in: v0.1.16

The `process.memoryUsage()` method returns an object describing the memory usage of the Node.js process measured in bytes.

For example, the code:

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

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

Will generate:

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

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

process.nextTick(callback[, arg][,...])

Added in: v0.1.26

- `callback` <Function>
- `[, arg][, ...]` <any> Additional arguments to pass when invoking the `callback`

The `process.nextTick()` method adds the `callback` to the "next tick queue". Once the current turn of the event loop turns to completion, all callbacks currently in the next tick queue will be called.

This is *not* a simple alias to `setTimeout(fn, 0)`, it's much more efficient. It runs before any additional I/O events (including timers) fire in subsequent ticks of the event loop.

```
console.log('start');  
process.nextTick(() => {  
  console.log('nextTick callback');  
});  
console.log('scheduled');  
// Output:
```

```
// start  
// scheduled  
// nextTick callback
```

This is important when developing APIs in order to give users the opportunity to assign event handlers *after* an object has been constructed but before any I/O has occurred:

```
function MyThing(options) {  
  this.setupOptions(options);  
  
  process.nextTick(() => {  
    this.startDoingStuff();  
  });  
}  
  
var thing = new MyThing();  
thing.getReadyForStuff();  
  
// thing.startDoingStuff() gets called now, not before.
```

It is very important for APIs to be either 100% synchronous or 100% asynchronous. Consider this example:

```
// WARNING! DO NOT USE! BAD UNSAFE HAZARD!  
function maybeSync(arg, cb) {  
  if (arg) {  
    cb();  
    return;  
  }  
  
  fs.stat('file', cb);
```

```
}
```

This API is hazardous because in the following case:

```
maybeSync(true, () => {
  foo();
});
bar();
```

It is not clear whether `foo()` or `bar()` will be called first.

The following approach is much better:

```
function definitelyAsync(arg, cb) {
  if (arg) {
    process.nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

Note: the next tick queue is completely drained on each pass of the event loop **before** additional I/O is processed. As a result, recursively setting `nextTick` callbacks will block any I/O from happening, just like a `while(true);` loop.

process.pid

#

Added in: v0.1.15

The `process.pid` property returns the PID of the process.

```
console.log(`This process is pid ${process.pid}`);
```

process.platform

#

Added in: v0.1.16

The `process.platform` property returns a string identifying the operating system platform on which the Node.js process is running. For instance `'darwin'`, `'freebsd'`, `'linux'`, `'sunos'` or `'win32'`

```
console.log(`This platform is ${process.platform}`);
```

process.release

#

Added in: v3.0.0

The `process.release` property returns an Object containing metadata related to the current release, including URLs for the source tarball and headers-only tarball.

`process.release` contains the following properties:

- `name <String>` A value that will always be `'node'` for Node.js. For legacy io.js releases, this will be `'io.js'`.
- `sourceUrl <String>` an absolute URL pointing to a `_.tar.gz_` file containing the source code of the current release.
- `headersUrl <String>` an absolute URL pointing to a `_.tar.gz_` file containing only the source header files for the current release. This file is significantly smaller than the full source file and can be used for compiling Node.js native add-ons.
- `libUrl <String>` an absolute URL pointing to a `_node.lib_` file matching the architecture and version of the current release. This file is used for compiling Node.js native add-ons. *This property is only present on Windows builds of Node.js and will be missing on all other platforms.*
- `lts <String>` a string label identifying the [LTS][] label for this release. If the Node.js release is not an LTS release, this will be `undefined`.

For example:

```
{  
  name: 'node',  
  lts: 'Argon',  
  sourceUrl: 'https://nodejs.org/download/release/v4.4.5/node-v4.4.5.tar'  
  headersUrl: 'https://nodejs.org/download/release/v4.4.5/node-v4.4.5-headers.tar'  
  libUrl: 'https://nodejs.org/download/release/v4.4.5/win-x64/node.lib'  
}
```

In custom builds from non-release versions of the source tree, only the `name` property may be present. The additional properties should not be relied upon to exist.

process.send(message[, sendHandle[, options]][, callback])

Added in: v0.5.9

- `message` <Object>
- `sendHandle` <Handle object>
- `options` <Object>
- `callback` <Function>
- Return: <Boolean>

If Node.js is spawned with an IPC channel, the `process.send()` method can be used to send messages to the parent process. Messages will be received as a '`message`' event on the parent's `ChildProcess` object.

If Node.js was not spawned with an IPC channel, `process.send()` will be `undefined`.

Note: This function uses `JSON.stringify()` internally to serialize the `message`.*

process.setegid(id)

Added in: v2.0.0

- `id` <String> | <number> A group name or ID

The `process.setegid()` method sets the effective group identity of the process. (See [setegid\(2\)](#).) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated a numeric ID.

```
if (process.getegid && process.setegid) {  
    console.log(`Current gid: ${process.getegid()}`);  
    try {  
        process.setegid(501);  
        console.log(`New gid: ${process.getegid()}`);  
    }  
    catch (err) {  
        console.log(`Failed to set gid: ${err}`);  
    }  
}
```

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

process.seteuid(id)

#

Added in: v2.0.0

- `id` <String> | <number> A user name or ID

The `process.seteuid()` method sets the effective user identity of the process. (See [seteuid\(2\)](#).) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

```
if (process.geteuid && process.seteuid) {  
    console.log(`Current uid: ${process.geteuid()}`);  
    try {
```

```
process.seteuid(501);
console.log(`New uid: ${process.geteuid()}`);
}
catch (err) {
  console.log(`Failed to set uid: ${err}`);
}
}
```

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

process.setgid(id)

#

Added in: v0.1.31

- `id <String> | <number>` The group name or ID

The `process.setgid()` method sets the group identity of the process. (See `setgid(2)`.) The `id` can be passed as either a numeric ID or a group name string. If a group name is specified, this method blocks while resolving the associated numeric ID.

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

Note: This function is only available on POSIX platforms (i.e. not Windows or

process.setgroups(groups)

Added in: v0.9.4

- `groups <Array>`

The `process.setgroups()` method sets the supplementary group IDs for the Node.js process. This is a privileged operation that requires the Node.js process to have `root` or the `CAP_SETGID` capability.

The `groups` array can contain numeric group IDs, group names or both.

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

process.setuid(id)

Added in: v0.1.28

The `process.setuid(id)` method sets the user identity of the process. (See `setuid(2)`.) The `id` can be passed as either a numeric ID or a username string. If a username is specified, the method blocks while resolving the associated numeric ID.

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

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

process.stderr

#

The `process.stderr` property returns a [Writable](#) stream equivalent to or associated with `stderr` (fd `2`).

`process.stderr` and `process.stdout` are unlike other streams in Node.js in that they cannot be closed (calling `end()` will throw an Error), they never emit the '`'finish'`' event, and writes can block when output is redirected to a file (although disks are fast and operating systems normally employ write-back caching so it should be a very rare occurrence indeed.)

Additionally, `process.stderr` and `process.stdout` are blocking when outputting to TTYs (terminals) on OS X as a workaround for the OS's very small, 1kb buffer size. This is to prevent interleaving between `stdout` and `stderr`.

process.stdin

#

The `process.stdin` property returns a [Readable](#) stream equivalent to or associated with `stdin` (fd `0`).

For example:

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

process.stdin.on('readable', () => {
  var chunk = process.stdin.read();
  if (chunk !== null) {
    process.stdout.write(`data: ${chunk}`);
  }
});

process.stdin.on('end', () => {
```

```
process.stdout.write('end');
});
```

As a **Readable** stream, `process.stdin` can also be used in "old" mode that is compatible with scripts written for Node.js prior to v0.10. For more information see [Stream compatibility](#).

Note: In "old" streams mode the `stdin` stream is paused by default, so one must call `process.stdin.resume()` to read from it. Note also that calling `process.stdin.resume()` itself would switch stream to "old" mode.

process.stdout

The `process.stdout` property returns a **Writable** stream equivalent to or associated with `stdout` (fd `1`).

For example:

```
console.log = (msg) => {
  process.stdout.write(` ${msg}\n`);
};
```

`process.stderr` and `process.stdout` are unlike other streams in Node.js in that they cannot be closed (calling `end()` will throw an Error), they never emit the '**'finish'**' event and that writes can block when output is redirected to a file (although disks are fast and operating systems normally employ write-back caching so it should be a very rare occurrence indeed.)

To check if Node.js is being run in a TTY context, read the `isTTY` property on `process.stderr`, `process.stdout`, or `process.stdin`:

TTY Terminals and process.stdout

The `process.stderr` and `process.stdout` streams are blocking when

outputting to TTYs (terminals) on OS X as a workaround for the operating system's small, 1kb buffer size. This is to prevent interleaving between `stdout` and `stderr`.

To check if Node.js is being run in a TTY context, check the `isTTY` property on `process.stderr`, `process.stdout`, or `process.stdin`.

For instance:

```
$ node -p "Boolean(process.stdin.isTTY)"  
true  
$ echo "foo" | node -p "Boolean(process.stdin.isTTY)"  
false  
  
$ node -p "Boolean(process.stdout.isTTY)"  
true  
$ node -p "Boolean(process.stdout.isTTY)" | cat  
false
```

See the TTY documentation for more information.

process.title

Added in: v0.1.104

The `process.title` property returns the current process title (i.e. returns the current value of `ps`). Assigning a new value to `process.title` modifies the current value of `ps`.

Note: When a new value is assigned, different platforms will impose different maximum length restrictions on the title. Usually such restrictions are quite limited. For instance, on Linux and OS X, `process.title` is limited to the size of the binary name plus the length of the command line arguments because setting the `process.title` overwrites the `argv` memory of the process. Node.js v0.8 allowed for longer process title strings by also overwriting the `environ` memory but that was potentially insecure and confusing in some (rather obscure) cases.

process.umask([mask])

#

Added in: v0.1.19

- `mask <number>`

The `process.umask()` method sets or returns the Node.js process's file mode creation mask. Child processes inherit the mask from the parent process. The old mask is returned if the `mask` argument is given, otherwise returns the current mask.

```
const newmask = 0o022;
const oldmask = process.umask(newmask);
console.log(
  `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`);
);
```

process.uptime()

#

Added in: v0.5.0

The `process.uptime()` method returns the number of seconds the current Node.js process has been running.

process.version

#

Added in: v0.1.3

The `process.version` property returns the Node.js version string.

```
console.log(`Version: ${process.version}`);
```

process.versions

#

Added in: v0.2.0

The `process.versions` property returns an object listing the version strings of Node.js and its dependencies.

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

Will generate output similar to:

```
{  
  http_parser: '2.3.0',  
  node: '1.1.1',  
  v8: '4.1.0.14',  
  uv: '1.3.0',  
  zlib: '1.2.8',  
  ares: '1.10.0-DEV',  
  modules: '43',  
  icu: '55.1',  
  openssl: '1.0.1k'  
}
```

Exit Codes

Node.js will normally exit with a `0` status code when no more async operations are pending. The following status codes are used in other cases:

- **1 Uncaught Fatal Exception** - There was an uncaught exception, and it was not handled by a domain or an '`uncaughtException`' event handler.
- **2** - Unused (reserved by Bash for builtin misuse)
- **3 Internal JavaScript Parse Error** - The JavaScript source code internal in Node.js's bootstrapping process caused a parse error. This is extremely rare, and generally can only happen during development of Node.js itself.
- **4 Internal JavaScript Evaluation Failure** - The JavaScript source code internal in Node.js's bootstrapping process failed to return a function value when evaluated. This is extremely rare, and generally can only happen during development of Node.js itself.
- **5 Fatal Error** - There was a fatal unrecoverable error in V8. Typically a

message will be printed to stderr with the prefix `FATAL ERROR`.

- **6 Non-function Internal Exception Handler** - There was an uncaught exception, but the internal fatal exception handler function was somehow set to a non-function, and could not be called.
- **7 Internal Exception Handler Run-Time Failure** - There was an uncaught exception, and the internal fatal exception handler function itself threw an error while attempting to handle it. This can happen, for example, if a `'uncaughtException'` or `domain.on('error')` handler throws an error.
- **8** - Unused. In previous versions of Node.js, exit code 8 sometimes indicated an uncaught exception.
- **9 - Invalid Argument** - Either an unknown option was specified, or an option requiring a value was provided without a value.
- **10 Internal JavaScript Run-Time Failure** - The JavaScript source code internal in Node.js's bootstrapping process threw an error when the bootstrapping function was called. This is extremely rare, and generally can only happen during development of Node.js itself.
- **12 Invalid Debug Argument** - The `--debug` and/or `--debug-brk` options were set, but an invalid port number was chosen.
- **>128 Signal Exits** - If Node.js receives a fatal signal such as `SIGKILL` or `SIGHUP`, then its exit code will be `128` plus the value of the signal code. This is a standard Unix practice, since exit codes are defined to be 7-bit integers, and signal exits set the high-order bit, and then contain the value of the signal code.

punycode

#

Stability: 2 - Stable

The `punycode` module is a bundled version of the `Punycode.js` module. It can be accessed using:

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

`Punycode` is a character encoding scheme defined by RFC 3492 that is primarily

intended for use in Internationalized Domain Names. Because host names in URLs are limited to ASCII characters only, Domain Names that contain non-ASCII characters must be converted into ASCII using the Punycode scheme. For instance, the Japanese character that translates into the English word, 'example' is '例'. The Internationalized Domain Name, '例.com' (equivalent to 'example.com') is represented by Punycode as the ASCII string 'xn--fsq.com'.

The `punycode` module provides a simple implementation of the Punycode standard.

Note: The `punycode` module is a third-party dependency used by Node.js and made available to developers as a convenience. Fixes or other modifications to the module must be directed to the [Punycode.js](#) project.

punycode.decode(string)

Added in: v0.5.1

- `string <String>`

The `punycode.decode()` method converts a [Punycode](#) string of ASCII-only characters to the equivalent string of Unicode codepoints.

```
punycode.decode('maana-pta'); // 'mañana'  
punycode.decode('--dqe34k'); // '–'
```

punycode.encode(string)

Added in: v0.5.1

- `string <String>`

The `punycode.encode()` method converts a string of Unicode codepoints to a [Punycode](#) string of ASCII-only characters.

```
punycode.encode('mañana'); // 'maana-pta'  
punycode.encode('–'); // '--dqe34k'
```

punycode.toASCII(domain)

#

Added in: v0.6.1

- domain <String>

The `punycode.toASCII()` method converts a Unicode string representing an Internationalized Domain Name to [Punycode](#). Only the non-ASCII parts of the domain name will be converted. Calling `punycode.toASCII()` on a string that already only contains ASCII characters will have no effect.

```
// encode domain names
punycode.toASCII('mañana.com'); // 'xn--maana-pta.com'
punycode.toASCII('☃-⌘.com'); // 'xn----dgo34k.com'
punycode.toASCII('example.com'); // 'example.com'
```

punycode.toUnicode(domain)

#

Added in: v0.6.1

- domain <String>

The `punycode.toUnicode()` method converts a string representing a domain name containing [Punycode](#) encoded characters into Unicode. Only the [Punycode](#) encoded parts of the domain name are be converted.

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn----dgo34k.com'); // '☃-⌘.com'
punycode.toUnicode('example.com'); // 'example.com'
```

punycode.ucs2

#

Added in: v0.7.0

punycode.ucs2.decode(string)

#

Added in: v0.7.0

- `string <String>`

The `punycode.ucs2.decode()` method returns an array containing the numeric codepoint values of each Unicode symbol in the string.

```
punycode.ucs2.decode('abc'); // [0x61, 0x62, 0x63]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

punycode.ucs2.encode(codePoints)

#

Added in: v0.7.0

- `codePoints <Array>`

The `punycode.ucs2.encode()` method returns a string based on an array of numeric code point values.

```
punycode.ucs2.encode([0x61, 0x62, 0x63]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.version

#

Added in: v0.6.1

Returns a string identifying the current `Punycode.js` version number.

Query String

#

Stability: 2 - Stable

The `querystring` module provides utilities for parsing and formatting URL query

strings. It can be accessed using:

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

querystring.escape(str)

#

Added in: v0.1.25

- `str` `<String>`

The `querystring.escape()` method performs URL percent-encoding on the given `str` in a manner that is optimized for the specific requirements of URL query strings.

The `querystring.escape()` method is used by `querystring.stringify()` and is generally not expected to be used directly. It is exported primarily to allow application code to provide a replacement percent-encoding implementation if necessary by assigning `querystring.escape` to an alternative function.

querystring.parse(str[, sep[, eq[, options]]])

#

Added in: v0.1.25

- `str` `<String>` The URL query string to parse
- `sep` `<String>` The substring used to delimit key and value pairs in the query string. Defaults to `'&'`.
- `eq` `<String>`. The substring used to delimit keys and values in the query string. Defaults to `'='`.
- `options` `<Object>`
 - `decodeURIComponent` `<Function>` The function to use when decoding percent-encoded characters in the query string. Defaults to `querystring.unescape()`.
 - `maxKeys` `<number>` Specifies the maximum number of keys to parse. Defaults to `1000`. Specify `0` to remove key counting limitations.

The `querystring.parse()` method parses a URL query string (`str`) into a

collection of key and value pairs.

For example, the query string '`'foo=bar&abc=xyz&abc=123'`' is parsed into:

```
{  
  foo: 'bar',  
  abc: ['xyz', '123']  
}
```

Note: The object returned by the `querystring.parse()` method *does not* prototypically extend from the JavaScript `Object`. This means that the typical `Object` methods such as `obj.toString()`, `obj.hashOwnProperty()`, and others are not defined and *will not work*.

By default, percent-encoded characters within the query string will be assumed to use UTF-8 encoding. If an alternative character encoding is used, then an alternative `decodeURIComponent` option will need to be specified as illustrated in the following example:

```
// Assuming gbkDecodeURIComponent function already exists...  
  
querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null,  
  { decodeURIComponent: gbkDecodeURIComponent })
```

querystring.stringify(obj[, sep[, eq[, options]]])

Added in: v0.1.25

- `obj` `<Object>` The object to serialize into a URL query string
- `sep` `<String>` The substring used to delimit key and value pairs in the query string. Defaults to `'&'`.
- `eq` `<String>`. The substring used to delimit keys and values in the query string. Defaults to `'='`.

- `options`
 - `encodeURIComponent <Function>` The function to use when converting URL-unsafe characters to percent-encoding in the query string. Defaults to `querystring.escape()`.

The `querystring.stringify()` method produces a URL query string from a given `obj` by iterating through the object's "own properties".

For example:

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })
// returns 'foo=bar&baz=qux&baz=quux&corge='

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

By default, characters requiring percent-encoding within the query string will be encoded as UTF-8. If an alternative encoding is required, then an alternative `encodeURIComponent` option will need to be specified as illustrated in the following example:

```
// Assuming gbkEncodeURIComponent function already exists,
querystring.stringify({ w: '中文', foo: 'bar' }, null, null,
{ encodeURIComponent: gbkEncodeURIComponent })
```

querystring.unescape(str)

#

Added in: v0.1.25

- `str <String>`

The `querystring.unescape()` method performs decoding of URL percent-encoded characters on the given `str`.

The `querystring.unescape()` method is used by `querystring.parse()` and is generally not expected to be used directly. It is exported primarily to allow application code to provide a replacement decoding implementation if necessary by assigning `querystring.unescape` to an alternative function.

By default, the `querystring.unescape()` method will attempt to use the JavaScript built-in `decodeURIComponent()` method to decode. If that fails, a safer equivalent that does not throw on malformed URLs will be used.

Readline

Stability: 2 - Stable

The `readline` module provides an interface for reading data from a `Readable` stream (such as `process.stdin`) one line at a time. It can be accessed using:

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

The following simple example illustrates the basic use of the `readline` module.

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('What do you think of Node.js? ', (answer) => {
  // TODO: Log the answer in a database
  console.log('Thank you for your valuable feedback:', answer);

  rl.close();
});
```

```
});
```

Note Once this code is invoked, the Node.js application will not terminate until the `readline.Interface` is closed because the interface waits for data to be received on the `input` stream.

Class: Interface

Added in: v0.1.104

Instances of the `readline.Interface` class are constructed using the `readline.createInterface()` method. Every instance is associated with a single `input` `Readable` stream and a single `output` `Writable` stream. The `output` stream is used to print prompts for user input that arrives on, and is read from, the `input` stream.

Event: 'close'

Added in: v0.1.98

The '`close`' event is emitted when one of the following occur:

- The `rl.close()` method is called and the `readline.Interface` instance has relinquished control over the `input` and `output` streams;
- The `input` stream receives its '`end`' event;
- The `input` stream receives `<ctrl>-D` to signal end-of-transmission (EOT);
- The `input` stream receives `<ctrl>-C` to signal `SIGINT` and there is no `SIGINT` event listener registered on the `readline.Interface` instance.

The listener function is called without passing any arguments.

The `readline.Interface` instance should be considered to be "finished" once the '`close`' event is emitted.

Event: 'line'

Added in: v0.1.98

The '`line`' event is emitted whenever the `input` stream receives an end-of-line

`input(\n, \r, or \r\n)`. This usually occurs when the user presses the `<Enter>`, or `<Return>` keys.

The listener function is called with a string containing the single line of received input.

For example:

```
rl.on('line', (input) => {
  console.log(`Received: ${input}`);
});
```

Event: 'pause'

Added in: v0.7.5

The `'pause'` event is emitted when one of the following occur:

- The `input` stream is paused.
- The `input` stream is not paused and receives the `SIGCONT` event. (See events `SIGTSTP` and `SIGCONT`)

The listener function is called without passing any arguments.

For example:

```
rl.on('pause', () => {
  console.log('Readline paused.');
});
```

Event: 'resume'

Added in: v0.7.5

The `'resume'` event is emitted whenever the `input` stream is resumed.

The listener function is called without passing any arguments.

```
rl.on('resume', () => {
  console.log('Readline resumed.');
});
```

Event: 'SIGCONT'

Added in: v0.7.5

The '**SIGCONT**' event is emitted when a Node.js process previously moved into the background using **<ctrl>-Z** (i.e. **SIGTSTP**) is then brought back to the foreground using **fg(1)**.

If the **input** stream was paused *before* the **SIGSTP** request, this event will not be emitted.

The listener function is invoked without passing any arguments.

For example:

```
rl.on('SIGCONT', () => {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

Note: The '**SIGCONT**' event is *not* supported on Windows.

Event: 'SIGINT'

Added in: v0.3.0

The '**SIGINT**' event is emitted whenever the **input** stream receives a **<ctrl>-C** input, known typically as **SIGINT**. If there are no '**SIGINT**' event listeners registered when the **input** stream receives a **SIGINT**, the '**'pause'**' event will be emitted.

The listener function is invoked without passing any arguments.

For example:

```
rl.on('SIGINT', () => {
  rl.question('Are you sure you want to exit?', answer) => {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

Event: 'SIGTSTP'

Added in: v0.7.5

The 'SIGTSTP' event is emitted when the `input` stream receives a `<ctrl>-Z` input, typically known as `SIGTSTP`. If there are no `SIGTSTP` event listeners registered when the `input` stream receives a `SIGTSTP`, the Node.js process will be sent to the background.

When the program is resumed using `fg(1)`, the 'pause' and `SIGCONT` events will be emitted. These can be used to resume the `input` stream.

The 'pause' and 'SIGCONT' events will not be emitted if the `input` was paused before the process was sent to the background.

The listener function is invoked without passing any arguments.

For example:

```
rl.on('SIGTSTP', () => {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

Note: The 'SIGTSTP' event is not supported on Windows.

rl.close()

#

Added in: v0.1.98

The `rl.close()` method closes the `readline.Interface` instance and relinquishes control over the `input` and `output` streams. When called, the `'close'` event will be emitted. Closes the `Interface` instance, relinquishing control on the `input` and `output` streams. The `'close'` event will also be emitted.

rl.pause()

#

Added in: v0.3.4

The `rl.pause()` method pauses the `input` stream, allowing it to be resumed later if necessary.

Calling `rl.pause()` does not immediately pause other events (including `'line'`) from being emitted by the `readline.Interface` instance.

rl.prompt([preserveCursor])

#

Added in: v0.1.98

- `preserveCursor` <boolean> If `true`, prevents the cursor placement from being reset to `0`.

The `rl.prompt()` method writes the `readline.Interface` instances configured `prompt` to a new line in `output` in order to provide a user with a new location at which to provide input.

When called, `rl.prompt()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the prompt is not written.

rl.question(query, callback)

#

Added in: v0.3.3

- `query <String>` A statement or query to write to `output`, prepended to the prompt.
- `callback <Function>` A callback function that is invoked with the user's input in response to the `query`.

The `rl.question()` method displays the `query` by writing it to the `output`, waits for user input to be provided on `input`, then invokes the `callback` function passing the provided input as the first argument.

When called, `rl.question()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `query` is not written.

Example usage:

```
rl.question('What is your favorite food?', (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});
```

Note: The `callback` function passed to `rl.question()` does not follow the typical pattern of accepting an `Error` object or `null` as the first argument. The `callback` is called with the provided answer as the only argument.

rl.resume()

Added in: v0.3.4

The `rl.resume()` method resumes the `input` stream if it has been paused.

rl.setPrompt(prompt)

Added in: v0.1.98

- `prompt <String>`

The `rl.setPrompt()` method sets the prompt that will be written to `output` whenever `rl.prompt()` is called.

rl.write(data[, key])

Added in: v0.1.98

- `data` `<String>`
- `key` `<Object>`
 - `ctrl` `<boolean>` `true` to indicate the `<ctrl>` key.
 - `meta` `<boolean>` `true` to indicate the `<Meta>` key.
 - `shift` `<boolean>` `true` to indicate the `<Shift>` key.
 - `name` `<String>` The name of the a key.

The `rl.write()` method will write either `data` or a key sequence identified by `key` to the `output`. The `key` argument is supported only if `output` is a `TTY` text terminal.

If `key` is specified, `data` is ignored.

When called, `rl.write()` will resume the `input` stream if it has been paused.

If the `readline.Interface` was created with `output` set to `null` or `undefined` the `data` and `key` are not written.

For example:

```
rl.write('Delete this!');  
// Simulate Ctrl+u to delete the line written previously  
rl.write(null, {ctrl: true, name: 'u'});
```

readline.clearLine(stream, dir)

Added in: v0.7.7

- `stream` `<Writable>`
- `dir` `<number>`
 - `-1` - to the left from cursor
 - `1` - to the right from cursor

- `0` - the entire line

The `readline.clearLine()` method clears current line of given TTY stream in a specified direction identified by `dir`.

readline.clearScreenDown(stream)

Added in: v0.7.7

- `stream <Writable>`

The `readline.clearScreenDown()` method clears the given TTY stream from the current position of the cursor down.

readline.createInterface(options)

Added in: v0.1.98

- `options <Object>`
 - `input <Readable>` The Readable stream to listen to. This option is *required*.
 - `output <Writable>` The Writable stream to write readline data to.
 - `completer <Function>` An optional function used for Tab completion.
 - `terminal <boolean>` `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. Defaults to checking `isTTY` on the `output` stream upon instantiation.
 - `historySize <number>` maximum number of history lines retained. To disable the history set this value to `0`. Defaults to `30`. This option makes sense only if `terminal` is set to `true` by the user or by an internal `output` check, otherwise the history caching mechanism is not initialized at all.
 - `prompt` - the prompt string to use. Default: '`>` '

The `readline.createInterface()` method creates a new `readline.Interface` instance.

For example:

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

Once the `readline.Interface` instance is created, the most common case is to listen for the `'line'` event:

```
rl.on('line', (line) => {
  console.log(`Received: ${line}`);
});
```

If `terminal` is `true` for this instance then the `output` stream will get the best compatibility if it defines an `output.columns` property and emits a `'resize'` event on the `output` if or when the columns ever change (`process.stdout` does this automatically when it is a TTY).

Use of the `completer` Function

When called, the `completer` function is provided the current line entered by the user, and is expected to return an Array with 2 entries:

- An Array with matching entries for the completion.
- The substring that was used for the matching.

For instance: `[[substr1, substr2, ...], originalsubstring]`.

```
function completer(line) {
  var completions = '.help .error .exit .quit .q'.split(' ')
  var hits = completions.filter((c) => { return c.indexOf(line) == 0 })
  // show all completions if none found
  return [hits.length ? hits : completions, line]
```

```
}
```

The `completer` function can be called asynchronously if it accepts two arguments:

```
function completer(linePartial, callback) {
  callback(null, [['123']], linePartial);
}
```

readline.cursorTo(stream, x, y)

#

Added in: v0.7.7

- `stream <Writable>`
- `x <number>`
- `y <number>`

The `readline.cursorTo()` method moves cursor to the specified position in a given `TTY stream`.

readline.emitKeypressEvents(stream[, interface])

#

Added in: v0.7.7

- `stream <Readable>`
- `interface <readline.Interface>`

The `readline.emitKeypressEvents()` method causes the given `Writable stream` to begin emitting '`keypress`' events corresponding to received input.

Optionally, `interface` specifies a `readline.Interface` instance for which autocompletion is disabled when copy-pasted input is detected.

If the `stream` is a `TTY`, then it must be in raw mode.

```
readline.emitKeypressEvents(process.stdin);
if (process.stdin.isTTY)
```

```
process.stdin.setRawMode(true);
```

readline.moveCursor(stream, dx, dy)

Added in: v0.7.7

- `stream <Writable>`
- `dx <number>`
- `dy <Number>`

The `readline.moveCursor()` method moves the cursor *relative* to its current position in a given `TTY stream`.

Example: Tiny CLI

The following example illustrates the use of `readline.Interface` class to implement a small command-line interface:

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  prompt: 'OHAI> '
});

rl.prompt();

rl.on('line', (line) => {
  switch(line.trim()) {
    case 'hello':
      console.log('world!');
      break;
    default:
      console.log(`Say what? I might have heard '${line.trim()}'`);
  }
});
```

```
        break;  
    }  
  
    rl.prompt();  
}).on('close', () => {  
    console.log('Have a great day!');  
    process.exit(0);  
});
```

Example: Read File Stream Line-by-Line

A common use case for `readline` is to consume input from a filesystem `Readable` stream one line at a time, as illustrated in the following example:

```
const readline = require('readline');  
const fs = require('fs');  
  
const rl = readline.createInterface({  
    input: fs.createReadStream('sample.txt')  
});  
  
rl.on('line', (line) => {  
    console.log('Line from file:', line);  
});
```

REPL

Stability: 2 - Stable

The `repl` module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includable in other applications. It can be accessed using:

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

#

Design and Features

The `repl` module exports the `repl.REPLServer` class. While running, instances of `repl.REPLServer` will accept individual lines of user input, evaluate those according to a user-defined evaluation function, then output the result. Input and output may be from `stdin` and `stdout`, respectively, or may be connected to any Node.js [stream](#).

Instances of `repl.REPLServer` support automatic completion of inputs, simplistic Emacs-style line editing, multi-line inputs, ANSI-styled output, saving and restoring current REPL session state, error recovery, and customizable evaluation functions.

Commands and Special Keys

#

The following special commands are supported by all REPL instances:

- `.break` - When in the process of inputting a multi-line expression, entering the `.break` command (or pressing the `<ctrl>-C` key combination) will abort further input or processing of that expression.
- `.clear` - Resets the REPL `context` to an empty object and clears any multi-line expression currently being input.
- `.exit` - Close the I/O stream, causing the REPL to exit.
- `.help` - Show this list of special commands.
- `.save` - Save the current REPL session to a file: > `.save ./file/to/save.js`
- `.load` - Load a file into the current REPL session. > `.load ./file/to/load.js`

The following key combinations in the REPL have these special effects:

- `<ctrl>-C` - When pressed once, has the same effect as the `.break` command. When pressed twice on a blank line, has the same effect as the `.exit` command.
- `<ctrl>-D` - Has the same effect as the `.exit` command.

- <tab> - When pressed on a blank line, displays global and local(scope) variables. When pressed while entering other input, displays relevant autocomplete options.

Default Evaluation

#

By default, all instances of `repl.REPLServer` use an evaluation function that evaluates JavaScript expressions and provides access to Node.js' built-in modules. This default behavior can be overridden by passing in an alternative evaluation function when the `repl.REPLServer` instance is created.

JavaScript Expressions

#

The default evaluator supports direct evaluation of JavaScript expressions:

```
> 1 + 1
2
> var m = 2
undefined
> m + 1
3
```

Unless otherwise scoped within blocks (e.g. `{ ... }`) or functions, variables declared either implicitly or using the `var` keyword are declared at the `global` scope.

Global and Local Scope

#

The default evaluator provides access to any variables that exist in the global scope. It is possible to expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`. For example:

```
const repl = require('repl');
var msg = 'message';

repl.start('> ').context.m = msg;
```

Properties in the `context` object appear as local within the REPL:

```
$ node repl_test.js
> m
'message'
```

It is important to note that context properties are *not* read-only by default. To specify read-only globals, context properties must be defined using `Object.defineProperty()`:

```
const repl = require('repl');
var msg = 'message';

const r = repl.start('> ');
Object.defineProperty(r, 'm', {
  configurable: false,
  enumerable: true,
  value: msg
});
```

Accessing Core Node.js Modules

The default evaluator will automatically load Node.js core modules into the REPL environment when used. For instance, unless otherwise declared as a global or scoped variable, the input `fs` will be evaluated on-demand as `global.fs = require('fs')`.

```
> fs.createReadStream('./some/file');
```

Assignment of the `_` (underscore) variable

The default evaluator will, by default, assign the result of the most recently evaluated expression to the special variable `_` (underscore).

```
> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
4
```

Explicitly setting `_` to a value will disable this behavior.

Custom Evaluation Functions

When a new `repl.REPLServer` is created, a custom evaluation function may be provided. This can be used, for instance, to implement fully customized REPL applications.

The following illustrates a hypothetical example of a REPL that performs translation of text from one language to another:

```
const repl = require('repl');
const Translator = require('translator').Translator;

const myTranslator = new Translator('en', 'fr');

function myEval(cmd, context, filename, callback) {
  callback(null, myTranslator.translate(cmd));
}

repl.start({prompt: '> ', eval: myEval});
```

Recoverable Errors

As a user is typing input into the REPL prompt, pressing the `<enter>` key will send the current line of input to the `eval` function. In order to support multi-line input, the eval function can return an instance of `repl.Recoverable` to the provided callback function:

```
function eval(cmd, context, filename, callback) {
  var result;
  try {
    result = vm.runInThisContext(cmd);
  } catch (e) {
    if (isRecoverableError(e)) {
      return callback(new repl.Recoverable(e));
    }
  }
  callback(null, result);
}

function isRecoverableError(error) {
  if (error.name === 'SyntaxError') {
    return /^(Unexpected end of input|Unexpected token)/.test(error.message);
  }
  return false;
}
```

Customizing REPL Output

By default, `repl.REPLServer` instances format output using the `util.inspect()` method before writing the output to the provided Writable stream (`process.stdout` by default). The `useColors` boolean option can be specified at construction to instruct the default writer to use ANSI style codes to colorize the output from the `util.inspect()` method.

It is possible to fully customize the output of a `repl.REPLServer` instance by

passing a new function in using the `writer` option on construction. The following example, for instance, simply converts any input text to upper case:

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

const r = repl.start({prompt: '>', eval: myEval, writer: myWriter});

function myEval(cmd, context, filename, callback) {
  callback(null, cmd);
}

function myWriter(output) {
  return output.toUpperCase();
}
```

Class: REPLServer

Added in: v0.1.91

The `repl.REPLServer` class inherits from the `readline.Interface` class. Instances of `repl.REPLServer` are created using the `repl.start()` method and *should not* be created directly using the JavaScript `new` keyword.

Event: 'exit'

Added in: v0.7.7

The `'exit'` event is emitted when the REPL is exited either by receiving the `.exit` command as input, the user pressing `<ctrl>-C` twice to signal `SIGINT`, or by pressing `<ctrl>-D` to signal `'end'` on the input stream. The listener callback is invoked without any arguments.

```
replServer.on('exit', () => {
  console.log('Received "exit" event from repl!');
  process.exit();
```

```
});
```

Event: 'reset'

#

Added in: v0.11.0

The `'reset'` event is emitted when the REPL's context is reset. This occurs whenever the `.clear` command is received as input *unless* the REPL is using the default evaluator and the `repl.REPLServer` instance was created with the `useGlobal` option set to `true`. The listener callback will be called with a reference to the `context` object as the only argument.

This can be used primarily to re-initialize REPL context to some pre-defined state as illustrated in the following simple example:

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

function initializeContext(context) {
  context.m = 'test';
}

var r = repl.start({prompt: '>'});
initializeContext(r.context);

r.on('reset', initializeContext);
```

When this code is executed, the global `'m'` variable can be modified but then reset to its initial value using the `.clear` command:

```
$ ./node example.js
>m
'test'
>m = 1
```

```
1  
>m  
1  
>.clear  
Clearing context...  
>m  
'test'  
>
```

replServer.defineCommand(keyword, cmd)

Added in: v0.3.0

- `keyword` `<String>` The command keyword (*without* a leading `.` character).
- `cmd` `<Object>` | `<Function>` The function to invoke when the command is processed.

The `replServer.defineCommand()` method is used to add new `.`-prefixed commands to the REPL instance. Such commands are invoked by typing a `.` followed by the `keyword`. The `cmd` is either a Function or an object with the following properties:

- `help` `<String>` Help text to be displayed when `.help` is entered (Optional).
- `action` `<Function>` The function to execute, optionally accepting a single string argument.

The following example shows two new commands added to the REPL instance:

```
const repl = require('repl');  
  
var replServer = repl.start({prompt: '> '});  
replServer.defineCommand('sayhello', {  
  help: 'Say hello',  
  action: function(name) {  
    this.lineParser.reset();
```

```
this.bufferedCommand = '';
console.log(`Hello, ${name}!`);
this.displayPrompt();
}

});

replServer.defineCommand('saybye', function() {
  console.log('Goodbye!');
  this.close();
});
```

The new commands can then be used from within the REPL instance:

```
> .sayhello Node.js User
Hello, Node.js User!
> .saybye
Goodbye!
```

replServer.displayPrompt([preserveCursor])

Added in: v0.1.91

- `preserveCursor` `<Boolean>`

The `replServer.displayPrompt()` method readies the REPL instance for input from the user, printing the configured `prompt` to a new line in the `output` and resuming the `input` to accept new input.

When multi-line input is being entered, an ellipsis is printed rather than the 'prompt'.

When `preserveCursor` is `true`, the cursor placement will not be reset to `0`.

The `replServer.displayPrompt` method is primarily intended to be called from within the action function for commands registered using the `replServer.defineCommand()` method.

repl.start(options)

#

Added in: v0.1.91

- `options <Object>`
 - `prompt <String>` The input prompt to display. Defaults to `>`.
 - `input <Readable>` The Readable stream from which REPL input will be read. Defaults to `process.stdin`.
 - `output <Writable>` The Writable stream to which REPL output will be written. Defaults to `process.stdout`.
 - `terminal <boolean>` If `true`, specifies that the `output` should be treated as a TTY terminal, and have ANSI/VT100 escape codes written to it. Defaults to checking the value of the `isTTY` property on the `output` stream upon instantiation.
 - `eval <Function>` The function to be used when evaluating each given line of input. Defaults to an async wrapper for the JavaScript `eval()` function. An `eval` function can error with `repl.Recoverable` to indicate the input was incomplete and prompt for additional lines.
 - `useColors <boolean>` If `true`, specifies that the default `writer` function should include ANSI color styling to REPL output. If a custom `writer` function is provided then this has no effect. Defaults to the REPL instances `terminal` value.
 - `useGlobal <boolean>` If `true`, specifies that the default evaluation function will use the JavaScript `global` as the context as opposed to creating a new separate context for the REPL instance. Defaults to `false`.
 - `ignoreUndefined <boolean>` If `true`, specifies that the default writer will not output the return value of a command if it evaluates to `undefined`. Defaults to `false`.
 - `writer <Function>` The function to invoke to format the output of each command before writing to `output`. Defaults to `util.inspect()`.
 - `replMode` - A flag that specifies whether the default evaluator executes all JavaScript commands in strict mode, default mode, or a hybrid mode ("magic" mode.) Acceptable values are:
 - `repl.REPL_MODE_SLOPPY` - evaluates expressions in sloppy mode.
 - `repl.REPL_MODE_STRICT` - evaluates expressions in strict mode. This

is equivalent to prefacing every repl statement with 'use strict'.

- `repl.REPL_MODE_MAGIC` - attempt to evaluate expressions in default mode. If expressions fail to parse, re-try in strict mode.
- `breakEvalOnSigint` - Stop evaluating the current piece of code when `SIGINT` is received, i.e. `Ctrl+C` is pressed. This cannot be used together with a custom `eval` function. Defaults to `false`.

The `repl.start()` method creates and starts a `repl.REPLServer` instance.

The Node.js REPL

Node.js itself uses the `repl` module to provide its own interactive interface for executing JavaScript. This can be used by executing the Node.js binary without passing any arguments (or by passing the `-i` argument):

```
$ node
> a = [1, 2, 3];
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
... });
1
2
3
```

Environment Variable Options

Various behaviors of the Node.js REPL can be customized using the following environment variables:

- `NODE_REPL_HISTORY` - When a valid path is given, persistent REPL history will be saved to the specified file rather than `.node_repl_history` in the user's home directory. Setting this value to `""` will disable persistent REPL history. Whitespace will be trimmed from the value.

- `NODE_REPL_HISTORY_SIZE` - Defaults to `1000`. Controls how many lines of history will be persisted if history is available. Must be a positive number.
- `NODE_REPL_MODE` - May be any of `sloppy`, `strict`, or `magic`. Defaults to `magic`, which will automatically run "strict mode only" statements in strict mode.

Persistent History

By default, the Node.js REPL will persist history between `node` REPL sessions by saving inputs to a `.node_repl_history` file located in the user's home directory. This can be disabled by setting the environment variable `NODE_REPL_HISTORY=""`.

`NODE_REPL_HISTORY_FILE`

Added in: v2.0.0 Deprecated since: v3.0.0

Stability: 0 - Deprecated: Use `NODE_REPL_HISTORY` instead.

Previously in Node.js/io.js v2.x, REPL history was controlled by using a `NODE_REPL_HISTORY_FILE` environment variable, and the history was saved in JSON format. This variable has now been deprecated, and the old JSON REPL history file will be automatically converted to a simplified plain text format. This new file will be saved to either the user's home directory, or a directory defined by the `NODE_REPL_HISTORY` variable, as documented in the [Environment Variable Options](#).

Using the Node.js REPL with advanced line-editors

For advanced line-editors, start Node.js with the environmental variable `NODE_NO_READLINE=1`. This will start the main and debugger REPL in canonical terminal settings which will allow you to use with `rlwrap`.

For example, you could add this to your `bashrc` file:

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

Starting multiple REPL instances against a single running instance

It is possible to create and run multiple REPL instances against a single running instance of Node.js that share a single `global` object but have separate I/O interfaces.

The following example, for instance, provides separate REPLs on `stdin`, a Unix socket, and a TCP socket:

```
const net = require('net');
const repl = require('repl');
var connections = 0;

repl.start({
  prompt: 'Node.js via stdin> ',
  input: process.stdin,
  output: process.stdout
});

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via Unix socket> ',
    input: socket,
    output: socket
  }).on('exit', () => {
    socket.end();
  })
}).listen('/tmp/node-repl-sock');

net.createServer((socket) => {
  connections += 1;
  repl.start({
    prompt: 'Node.js via TCP socket> ',
    input: socket,
    output: socket
  })
}).listen(1337);
```

```
    output: socket
}).on('exit', () => {
  socket.end();
});
}).listen(5001);
```

Running this application from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet`, for instance, is useful for connecting to TCP sockets, while `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, it is possible to connect to a long-running Node.js process without restarting it.

For an example of running a "full-featured" (`terminal`) REPL over a `net.Server` and `net.Socket` instance, see: <https://gist.github.com/2209310>

For an example of running a REPL instance over `curl(1)`, see:
<https://gist.github.com/2053342>

Stream

#

Stability: 2 - Stable

A stream is an abstract interface for working with streaming data in Node.js. The `stream` module provides a base API that makes it easy to build objects that implement the stream interface.

There are many stream objects provided by Node.js. For instance, a `request` to an `HTTP` server and `process.stdout` are both stream instances.

Streams can be readable, writable, or both. All streams are instances of `EventEmitter`.

The `stream` module can be accessed using:

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

While it is important for all Node.js users to understand how streams works, the `stream` module itself is most useful for developer's that are creating new types of stream instances. Developer's who are primarily *consuming* stream objects will rarely (if ever) have need to use the `stream` module directly.

Organization of this document

This document is divided into two primary sections and third section for additional notes. The first section explains the elements of the stream API that are required to *use* streams within an application. The second section explains the elements of the API that are required to *implement* new types of streams.

Types of Streams

There are four fundamental stream types within Node.js:

- **Readable** - streams from which data can be read (for example `fs.createReadStream()`).
- **Writable** - streams to which data can be written (for example `fs.createWriteStream()`).
- **Duplex** - streams that are both Readable and Writable (for example `net.Socket`).
- **Transform** - Duplex streams that can modify or transform the data as it is written and read (for example `zlib.createDeflate()`).

Object Mode

All streams created by Node.js APIs operate exclusively on strings and `Buffer` objects. It is possible, however, for stream implementations to work with other types of JavaScript values (with the exception of `null` which serves a special purpose within streams). Such streams are considered to operate in "object mode".

Stream instances are switched into object mode using the `objectMode` option when the stream is created. Attempting to switch an existing stream into object mode is not safe.

Buffering

Both `Writable` and `Readable` streams will store data in an internal buffer that can be retrieved using `writable._writableState.getBuffer()` or `readable._readableState.buffer`, respectively.

The amount of data potentially buffered depends on the `highWaterMark` option passed into the streams constructor. For normal streams, the `highWaterMark` option specifies a total number of bytes. For streams operating in object mode, the `highWaterMark` specifies a total number of objects.

Data is buffered in Readable streams when the implementation calls `stream.push(chunk)`. If the consumer of the Stream does not call `stream.read()`, the data will sit in the internal queue until it is consumed.

Once the total size of the internal read buffer reaches the threshold specified by `highWaterMark`, the stream will temporarily stop reading data from the underlying resource until the data currently buffered can be consumed (that is, the stream will stop calling the internal `readable._read()` method that is used to fill the read buffer).

Data is buffered in Writable streams when the `writable.write(chunk)` method is called repeatedly. While the total size of the internal write buffer is below the threshold set by `highWaterMark`, calls to `writable.write()` will return `true`. Once the the size of the internal buffer reaches or exceeds the `highWaterMark`, `false` will be returned.

A key goal of the `stream` API, an in particular the `stream.pipe()` method, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

Because `Duplex` and `Transform` streams are both Readable and Writable, each

maintain two separate internal buffers used for reading and writing, allowing each side to operate independently of the other while maintaining an appropriate and efficient flow of data. For example, `net.Socket` instances are **Duplex** streams whose Readable side allows consumption of data received *from* the socket and whose Writable side allows writing data *to* the socket. Because data may be written to the socket at a faster or slower rate than data is received, it is important each side operate (and buffer) independently of the other.

API for Stream Consumers

Almost all Node.js applications, no matter how simple, use streams in some manner. The following is an example of using streams in a Node.js application that implements an HTTP server:

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

const server = http.createServer( (req, res) => {
  // req is an http.IncomingMessage, which is a Readable Stream
  // res is an http.ServerResponse, which is a Writable Stream

  var body = '';
  // Get the data as utf8 strings.
  // If an encoding is not set, Buffer objects will be received.
  req.setEncoding('utf8');

  // Readable streams emit 'data' events once a listener is added
  req.on('data', (chunk) => {
    body += chunk;
  });

  // the end event indicates that the entire body has been received
  req.on('end', () => {
    try {
```

```
    const data = JSON.parse(body);
} catch (er) {
    // uh oh! bad json!
    res.statusCode = 400;
    return res.end(`error: ${er.message}`);
}

// write back something interesting to the user:
res.write(typeof data);
res.end();
});

});

server.listen(1337);

// $ curl localhost:1337 -d '{}'
// object
// $ curl localhost:1337 -d '"foo"'
// string
// $ curl localhost:1337 -d 'not json'
// error: Unexpected token o
```

Writable streams (such as `res` in the example) expose methods such as `write()` and `end()` that are used to write data onto the stream.

Readable streams use the `EventEmitter` API for notifying application code when data is available to be read off the stream. That available data can be read from the stream in multiple ways.

Both `Writable` and `Readable` streams use the `EventEmitter` API in various ways to communicate the current state of the stream.

Duplex and Transform streams are both `Writable` and `Readable`.

Applications that are either writing data to or consuming data from a stream are not required to implement the stream interfaces directly and will generally have no reason to call `require('stream')`.

Developers wishing to implement new types of streams should refer to the section [API for Stream Implementers](#).

Writable Streams

Writable streams are an abstraction for a *destination* to which data is written.

Examples of [Writable](#) streams include:

- [HTTP requests, on the client](#)
- [HTTP responses, on the server](#)
- [fs write streams](#)
- [zlib streams](#)
- [crypto streams](#)
- [TCP sockets](#)
- [child process stdin](#)
- [`process.stdout`, `process.stderr`](#)

Note: Some of these examples are actually [Duplex](#) streams that implement the [Writable](#) interface.

All [Writable](#) streams implement the interface defined by the `stream.Writable` class.

While specific instances of [Writable](#) streams may differ in various ways, all Writable streams follow the same fundamental usage pattern as illustrated in the example below:

```
const myStream = getWritableStreamSomehow();
myStream.write('some data');
myStream.write('some more data');
```

```
myStream.end('done writing data');
```

Class: stream.Writable

#

Event: 'close'

#

The `'close'` event is emitted when the stream and any of its underlying resources (a file descriptor, for example) have been closed. The event indicates that no more events will be emitted, and no further computation will occur.

Not all Writable streams will emit the `'close'` event.

Event: 'drain'

#

If a call to `stream.write(chunk)` returns `false`, the `'drain'` event will be emitted when it is appropriate to resume writing data to the stream.

```
// Write the data to the supplied writable stream one million times.  
// Be attentive to back-pressure.  
  
function writeOneMillionTimes(writer, data, encoding, callback) {  
  var i = 1000000;  
  writer.write(data, encoding, callback);  
  write();  
  function write() {  
    var ok = true;  
    do {  
      i--;  
      if (i === 0) {  
        // last time!  
        writer.write(data, encoding, callback);  
      } else {  
        // see if we should continue, or wait  
        // don't pass the callback, because we're not done yet.  
        ok = writer.write(data, encoding);  
      }  
    } while (ok);  
  }  
}
```

```
    } while (i > 0 && ok);
    if (i > 0) {
        // had to stop early!
        // write some more once it drains
        writer.once('drain', write);
    }
}
```

Event: 'error'

#

- <Error>

The 'error' event is emitted if an error occurred while writing or piping data. The listener callback is passed a single Error argument when called.

Note: The stream is not closed when the 'error' event is emitted.

Event: 'finish'

#

The 'finish' event is emitted after the stream.end() method has been called, and all data has been flushed to the underlying system.

```
const writer = getWritableStreamSomehow();
for (var i = 0; i < 100; i++) {
    writer.write(`hello, #${i}!\n`);
}
writer.end('This is the end\n');
writer.on('finish', () => {
    console.error('All writes are now complete.');
});
```

Event: 'pipe'

#

- `src` <`stream.Readable`> source stream that is piping to this writable

The '`pipe`' event is emitted when the `stream.pipe()` method is called on a readable stream, adding this writable to its set of destinations.

```
const writer = getWritableStreamSomehow();
const reader = getReadableStreamSomehow();
writer.on('pipe', (src) => {
  console.error('something is piping into the writer');
  assert.equal(src, reader);
});
reader.pipe(writer);
```

Event: 'unpipe'

#

- `src` <`Readable Stream`> The source stream that unpiped this writable

The '`unpipe`' event is emitted when the `stream.unpipe()` method is called on a `Readable` stream, removing this `Writable` from its set of destinations.

```
const writer = getWritableStreamSomehow();
const reader = getReadableStreamSomehow();
writer.on('unpipe', (src) => {
  console.error('Something has stopped piping into the writer.');
  assert.equal(src, reader);
});
reader.pipe(writer);
reader.unpipe(writer);
```

writable.cork()

#

The `writable.cork()` method forces all written data to be buffered in memory. The buffered data will be flushed when either the `stream.uncork()` or `stream.end()` methods are called.

The primary intent of `writable.cork()` is to avoid a situation where writing many small chunks of data to a stream do not cause an backup in the internal buffer that would have an adverse impact on performance. In such situations, implementations that implement the `writable._writev()` method can perform buffered writes in a more optimized manner.

writable.end([chunk][, encoding][, callback])

#

- `chunk <String> | <Buffer> | <any>` Optional data to write. For streams not operating in object mode, `chunk` must be a string or a `Buffer`. For object mode streams, `chunk` may be any JavaScript value other than `null`.
- `encoding <String>` The encoding, if `chunk` is a String
- `callback <Function>` Optional callback for when the stream is finished

Calling the `writable.end()` method signals that no more data will be written to the `Writable`. The optional `chunk` and `encoding` arguments allow one final additional chunk of data to be written immediately before closing the stream. If provided, the optional `callback` function is attached as a listener for the '`finish`' event.

Calling the `stream.write()` method after calling `stream.end()` will raise an error.

```
// write 'hello, ' and then end with 'world!'
const file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// writing more now is not allowed!
```

writable.setDefaultEncoding(encoding)

#

- `encoding <String>` The new default encoding
- Return: `this`

The `writable.setDefaultEncoding()` method sets the default `encoding` for a

writable.uncork()

The `writable.uncork()` method flushes all data buffered since `stream.cork()` was called.

When using `writable.cork()` and `writable.uncork()` to manage the buffering of writes to a stream, it is recommended that calls to `writable.uncork()` be deferred using `process.nextTick()`. Doing so allows batching of all `writable.write()` calls that occur within a given Node.js event loop phase.

```
stream.cork();
stream.write('some ');
stream.write('data ');
process.nextTick(() => stream.uncork());
```

If the `writable.cork()` method is called multiple times on a stream, the same number of calls to `writable.uncork()` must be called to flush the buffered data.

```
stream.cork();
stream.write('some ');
stream.cork();
stream.write('data ');
process.nextTick(() => {
  stream.uncork();
  // The data will not be flushed until uncork() is called a second time
  stream.uncork();
});
```

writable.write(chunk[, encoding][, callback])

- `chunk <String> | <Buffer>` The data to write

- `encoding` `<String>` The encoding, if `chunk` is a String
- `callback` `<Function>` Callback for when this chunk of data is flushed
- Returns: `<Boolean>` `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

The `writable.write()` method writes some data to the stream, and calls the supplied `callback` once the data has been fully handled. If an error occurs, the `callback` *may or may not* be called with the error as its first argument. To reliably detect write errors, add a listener for the `'error'` event.

The return value indicates whether the written `chunk` was buffered internally and the buffer has exceeded the `highWaterMark` configured when the stream was created. If `false` is returned, further attempts to write data to the stream should be paused until the `'drain'` event is emitted.

A Writable stream in object mode will always ignore the `encoding` argument.

Readable Streams

Readable streams are an abstraction for a source from which data is consumed.

Examples of Readable streams include:

- [HTTP responses, on the client](#)
- [HTTP requests, on the server](#)
- [fs read streams](#)
- [zlib streams](#)
- [crypto streams](#)
- [TCP sockets](#)
- [child process stdout and stderr](#)
- [process.stdin](#)

All [Readable](#) streams implement the interface defined by the `stream.Readable` class.

Two Modes

Readable streams effectively operate in one of two modes: flowing and paused.

When in flowing mode, data is read from the underlying system automatically and provided to an application as quickly as possible using events via the `EventEmitter` interface.

In paused mode, the `stream.read()` method must be called explicitly to read chunks of data from the stream.

All `Readable` streams begin in paused mode but can be switched to flowing mode in one of the following ways:

- Adding a '`data`' event handler.
- Calling the `stream.resume()` method.
- Calling the `stream.pipe()` method to send the data to a `Writable`.

The Readable can switch back to paused mode using one of the following:

- If there are no pipe destinations, by calling the `stream.pause()` method.
- If there are pipe destinations, by removing any '`data`' event handlers, and removing all pipe destinations by calling the `stream.unpipe()` method.

The important concept to remember is that a Readable will not generate data until a mechanism for either consuming or ignoring that data is provided. If the consuming mechanism is disabled or taken away, the Readable will *attempt* to stop generating the data.

Note: For backwards compatibility reasons, removing '`data`' event handlers will **not** automatically pause the stream. Also, if there are piped destinations, then calling `stream.pause()` will not guarantee that the stream will *remain* paused once those destinations drain and ask for more data.

Note: If a `Readable` is switched into flowing mode and there are no consumers available handle the data, that data will be lost. This can occur, for instance, when the `readable.resume()` method is called without a listener attached to the '`data`' event, or when a '`data`' event handler is removed from the stream.

Three States

The "two modes" of operation for a Readable stream are a simplified abstraction for the more complicated internal state management that is happening within the Readable stream implementation.

Specifically, at any given point in time, every Readable is in one of three possible states:

- `readable._readableState.flowing = null`
- `readable._readableState.flowing = false`
- `readable._readableState.flowing = true`

When `readable._readableState.flowing` is `null`, no mechanism for consuming the streams data is provided so the stream will not generate its data.

Attaching a listener for the `'data'` event, calling the `readable.pipe()` method, or calling the `readable.resume()` method will switch `readable._readableState.flowing` to `true`, causing the Readable to begin actively emitting events as data is generated.

Calling `readable.pause()`, `readable.unpipe()`, or receiving "back pressure" will cause the `readable._readableState.flowing` to be set as `false`, temporarily halting the flowing of events but *not* halting the generation of data.

While `readable._readableState.flowing` is `false`, data may be accumulating within the streams internal buffer.

Choose One

The Readable stream API evolved across multiple Node.js versions and provides multiple methods of consuming stream data. In general, developers should choose *one* of the methods of consuming data and *should never* use multiple methods to consume data from a single stream.

Use of the `readable.pipe()` method is recommended for most users as it has been implemented to provide the easiest way of consuming stream data. Developers that require more fine-grained control over the transfer and generation of data can

use the `EventEmitter` and `readable.pause()` / `readable.resume()` APIs.

Class: stream.Readable

#

Event: 'close'

#

The `'close'` event is emitted when the stream and any of its underlying resources (a file descriptor, for example) have been closed. The event indicates that no more events will be emitted, and no further computation will occur.

Not all `Readable` streams will emit the `'close'` event.

Event: 'data'

#

- `chunk <Buffer> | <String> | <any>` The chunk of data. For streams that are not operating in object mode, the chunk will be either a string or `Buffer`. For streams that are in object mode, the chunk can be any JavaScript value other than `null`.

The `'data'` event is emitted whenever the stream is relinquishing ownership of a chunk of data to a consumer. This may occur whenever the stream is switched in flowing mode by calling `readable.pipe()`, `readable.resume()`, or by attaching a listener callback to the `'data'` event. The `'data'` event will also be emitted whenever the `readable.read()` method is called and a chunk of data is available to be returned.

Attaching a `'data'` event listener to a stream that has not been explicitly paused will switch the stream into flowing mode. Data will then be passed as soon as it is available.

The listener callback will be passed the chunk of data as a string if a default encoding has been specified for the stream using the `readable.setEncoding()` method; otherwise the data will be passed as a `Buffer`.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
```

```
    console.log(`Received ${chunk.length} bytes of data.`);
});
```

Event: 'end'

#

The `'end'` event is emitted when there is no more data to be consumed from the stream.

Note: The `'end'` event **will not be emitted** unless the data is completely consumed. This can be accomplished by switching the stream into flowing mode, or by calling `stream.read()` repeatedly until all data has been consumed.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
readable.on('end', () => {
  console.log('There will be no more data.');
});
```

Event: 'error'

#

- <Error>

The `'error'` event may be emitted by a Readable implementation at any time. Typically, this may occur if the underlying stream is unable to generate data due to an underlying internal failure, or when a stream implementation attempts to push an invalid chunk of data.

The listener callback will be passed a single `Error` object.

Event: 'readable'

#

The `'readable'` event is emitted when there is data available to be read from the

stream. In some cases, attaching a listener for the 'readable' event will cause some amount of data to be read into an internal buffer.

```
const readable = getReadableStreamSomehow();
readable.on('readable', () => {
  // there is some data to read now
});
```

The 'readable' event will also be emitted once the end of the stream data has been reached but before the 'end' event is emitted.

Effectively, the 'readable' event indicates that the stream has new information: either new data is available or the end of the stream has been reached. In the former case, `stream.read()` will return the available data. In the latter case, `stream.read()` will return `null`. For instance, in the following example, `foo.txt` is an empty file:

```
const fs = require('fs');
const rr = fs.createReadStream('foo.txt');
rr.on('readable', () => {
  console.log('readable:', rr.read());
});
rr.on('end', () => {
  console.log('end');
});
```

The output of running this script is:

```
$ node test.js
readable: null
end
```

Note: In general, the `readable.pipe()` and '`data`' event mechanisms are preferred over the use of the '`readable`' event.

readable.isPaused()

#

- Return: `<Boolean>`

The `readable.isPaused()` method returns the current operating state of the Readable. This is used primarily by the mechanism that underlies the `readable.pipe()` method. In most typical cases, there will be no reason to use this method directly.

```
const readable = new stream.Readable

readable.isPaused() // === false
readable.pause()
readable.isPaused() // === true
readable.resume()
readable.isPaused() // === false
```

readable.pause()

#

- Return: `this`

The `readable.pause()` method will cause a stream in flowing mode to stop emitting '`data`' events, switching out of flowing mode. Any data that becomes available will remain in the internal buffer.

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
  readable.pause();
  console.log('There will be no additional data for 1 second.');
  setTimeout(() => {
```

```
    console.log('Now data will start flowing again.');
    readable.resume();
}, 1000);
});
```

readable.pipe(destination[, options])

#

- `destination` `<stream.Writable>` The destination for writing data
- `options` `<Object>` Pipe options
 - `end` `<Boolean>` End the writer when the reader ends. Defaults to `true`.

The `readable.pipe()` method attaches a `Writable` stream to the `readable`, causing it to switch automatically into flowing mode and push all of its data to the attached `Writable`. The flow of data will be automatically managed so that the destination `Writable` stream is not overwhelmed by a faster Readable stream.

The following example pipes all of the data from the `readable` into a file named `file.txt`:

```
const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt'
readable.pipe(writable);
```

It is possible to attach multiple `Writable` streams to a single `Readable` stream.

The `readable.pipe()` method returns a reference to the *destination* stream making it possible to set up chains of piped streams:

```
const r = fs.createReadStream('file.txt');
const z = zlib.createGzip();
const w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);
```

By default, `stream.end()` is called on the destination Writable stream when the source Readable stream emits '`end`', so that the destination is no longer writable. To disable this default behavior, the `end` option can be passed as `false`, causing the destination stream to remain open, as illustrated in the following example:

```
reader.pipe(writer, { end: false });
reader.on('end', () => {
  writer.end('Goodbye\n');
});
```

One important caveat is that if the Readable stream emits an error during processing, the Writable destination *is not closed* automatically. If an error occurs, it will be necessary to *manually* close each stream in order to prevent memory leaks.

Note: The `process.stderr` and `process.stdout` Writable streams are never closed until the Node.js process exits, regardless of the specified options.

readable.read([size])

- `size <Number>` Optional argument to specify how much data to read.
- Return `<String> | <Buffer> | <Null>`

The `readable.read()` method pulls some data out of the internal buffer and returns it. If no data available to be read, `null` is returned. By default, the data will be returned as a `Buffer` object unless an encoding has been specified using the `readable.setEncoding()` method or the stream is operating in object mode.

The optional `size` argument specifies a specific number of bytes to read. If `size` bytes are not available to be read, `null` will be returned *unless* the stream has ended, in which case all of the data remaining in the internal buffer will be returned (*even if it exceeds size bytes*).

If the `size` argument is not specified, all of the data contained in the internal buffer will be returned.

The `readable.read()` method should only be called on Readable streams operating in paused mode. In flowing mode, `readable.read()` is called automatically until the internal buffer is fully drained.

```
const readable = getReadableStreamSomehow();
readable.on('readable', () => {
  var chunk;
  while (null !== (chunk = readable.read())) {
    console.log(`Received ${chunk.length} bytes of data.`);
  }
});
```

In general, it is recommended that developers avoid the use of the `'readable'` event and the `readable.read()` method in favor of using either `readable.pipe()` or the `'data'` event.

A Readable stream in object mode will always return a single item from a call to `readable.read(size)`, regardless of the value of the `size` argument.

Note: If the `readable.read()` method returns a chunk of data, a `'data'` event will also be emitted.

Note: Calling `stream.read([size])` after the `'end'` event has been emitted will return `null`. No runtime error will be raised.

readable.resume()

#

- Return: `this`

The `readable.resume()` method causes an explicitly paused Readable stream to resume emitting `'data'` events, switching the stream into flowing mode.

The `readable.resume()` method can be used to fully consume the data from a stream without actually processing any of that data as illustrated in the following example:

```
getReadableStreamSomehow()
  .resume()
  .on('end', () => {
    console.log('Reached the end, but did not read anything.');
  });
});
```

readable.setEncoding(encoding)

- `encoding <String>` The encoding to use.
- Return: `this`

The `readable.setEncoding()` method sets the default character encoding for data read from the Readable stream.

Setting an encoding causes the stream data to be returned as string of the specified encoding rather than as `Buffer` objects. For instance, calling `readable.setEncoding('utf8')` will cause the output data will be interpreted as UTF-8 data, and passed as strings. Calling `readable.setEncoding('hex')` will cause the data to be encoded in hexadecimal string format.

The Readable stream will properly handle multi-byte characters delivered through the stream that would otherwise become improperly decoded if simply pulled from the stream as `Buffer` objects.

Encoding can be disabled by calling `readable.setEncoding(null)`. This approach is useful when working with binary data or with large multi-byte strings spread out over multiple chunks.

```
const readable = getReadableStreamSomehow();
readable.setEncoding('utf8');
readable.on('data', (chunk) => {
  assert.equal(typeof chunk, 'string');
  console.log('got %d characters of string data', chunk.length);
});
```

readable.unpipe([destination])

#

- `destination <stream.Writable>` Optional specific stream to unpipe

The `readable.unpipe()` method detaches a Writable stream previously attached using the `stream.pipe()` method.

If the `destination` is not specified, then *all* pipes are detached.

If the `destination` is specified, but no pipe is set up for it, then the method does nothing.

```
const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt',
// but only for the first second
readable.pipe(writable);
setTimeout(() => {
  console.log('Stop writing to file.txt');
  readable.unpipe(writable);
  console.log('Manually close the file stream');
  writable.end();
}, 1000);
```

readable.unshift(chunk)

#

- `chunk <Buffer> | <String>` Chunk of data to unshift onto the read queue

The `readable.unshift()` method pushes a chunk of data back into the internal buffer. This is useful in certain situations where a stream is being consumed by code that needs to "un-consume" some amount of data that it has optimistically pulled out of the source, so that the data can be passed on to some other party.

Note: The `stream.unshift(chunk)` method cannot be called after the 'end'

event has been emitted or a runtime error will be thrown.

Developers using `stream.unshift()` often should consider switching to use of a **Transform** stream instead. See the [API for Stream Implementers](#) section for more information.

```
// Pull off a header delimited by \n\n
// use unshift() if we get too much
// Call the callback with (error, header, stream)
const StringDecoder = require('string_decoder').StringDecoder;
function parseHeader(stream, callback) {
  stream.on('error', callback);
  stream.on('readable', onReadable);
  const decoder = new StringDecoder('utf8');
  var header = '';
  function onReadable() {
    var chunk;
    while (null !== (chunk = stream.read())) {
      var str = decoder.write(chunk);
      if (str.match(/\n\n/)) {
        // found the header boundary
        var split = str.split(/\n\n/);
        header += split.shift();
        const remaining = split.join('\n\n');
        const buf = Buffer.from(remaining, 'utf8');
        if (buf.length)
          stream.unshift(buf);
        stream.removeListener('error', callback);
        stream.removeListener('readable', onReadable);
        // now the body of the message can be read from the stream.
        callback(null, header, stream);
      } else {
        // still reading the header.
      }
    }
  }
}
```

```
    header += str;
}
}
}
}
```

Note: Unlike `stream.push(chunk)`, `stream.unshift(chunk)` will not end the reading process by resetting the internal reading state of the stream. This can cause unexpected results if `readable.unshift()` is called during a read (i.e. from within a `stream._read()` implementation on a custom stream). Following the call to `readable.unshift()` with an immediate `stream.push('')` will reset the reading state appropriately, however it is best to simply avoid calling `readable.unshift()` while in the process of performing a read.

readable.wrap(stream)

- `stream <Stream>` An "old style" readable stream

Versions of Node.js prior to v0.10 had streams that did not implement the entire `stream` module API as it is currently defined. (See [Compatibility](#) for more information.)

When using an older Node.js library that emits '`'data'`' events and has a `stream.pause()` method that is advisory only, the `readable.wrap()` method can be used to create a [Readable](#) stream that uses the old stream as its data source.

It will rarely be necessary to use `readable.wrap()` but the method has been provided as a convenience for interacting with older Node.js applications and libraries.

For example:

```
const OldReader = require('./old-api-module.js').OldReader;
const Readable = require('stream').Readable;
const oreader = new OldReader;
```

```
const myReader = new Readable().wrap(coreader);

myReader.on('readable', () => {
  myReader.read(); // etc.
});
```

Duplex and Transform Streams

Class: stream.Duplex

Duplex streams are streams that implement both the **Readable** and **Writable** interfaces.

Examples of Duplex streams include:

- **TCP sockets**
- **zlib streams**
- **crypto streams**

Class: stream.Transform

Transform streams are **Duplex** streams where the output is in some way related to the input. Like all **Duplex** streams, Transform streams implement both the **Readable** and **Writable** interfaces.

Examples of Transform streams include:

- **zlib streams**
- **crypto streams**

API for Stream Implementers

The `stream` module API has been designed to make it possible to easily implement streams using JavaScript's prototypical inheritance model.

First, a stream developer would declare a new JavaScript class that extends one of

the four basic stream classes (`stream.Writable`, `stream.Readable`, `stream.Duplex`, or `stream.Transform`), making sure to call the appropriate parent class constructor:

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    super(options);
  }
}
```

The new stream class must then implement one or more specific methods, depending on the type of stream being created, as detailed in the chart below:

Use-case	Class	Method(s) to implement
Reading only	Readable	<code>_read</code>
Writing only	Writable	<code>_write</code> , <code>_writerv</code>
Reading and writing	Duplex	<code>_read</code> , <code>_write</code> , <code>_writerv</code>
Operate on written data, then read the result	Transform	<code>_transform</code> , <code>_flush</code>

Note: The implementation code for a stream should *never* call the "public" methods of a stream that are intended for use by consumers (as described in the [API for Stream Consumers](#) section). Doing so may lead to adverse side effects in application code consuming the stream.

Simplified Construction

For many simple cases, it is possible to construct a stream without relying on inheritance. This can be accomplished by directly creating instances of the `stream.Writable`, `stream.Readable`, `stream.Duplex` or `stream.Transform` objects and passing appropriate methods as constructor options.

For example:

```
const Writable = require('stream').Writable;

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  }
});
```

Implementing a Writable Stream

The `stream.Writable` class is extended to implement a `Writable` stream.

Custom Writable streams *must* call the `new stream.Writable([options])` constructor and implement the `writable._write()` method. The `writable._writev()` method *may* also be implemented.

Constructor: `new stream.Writable(options)`

- `options <Object>`
 - `highWaterMark <Number>` Buffer level when `stream.write()` starts returning `false`. Defaults to `16384` (16kb), or `16` for `objectMode` streams.
 - `decodeStrings <Boolean>` Whether or not to decode strings into Buffers before passing them to `stream._write()`. Defaults to `true`.
 - `objectMode <Boolean>` Whether or not the `stream.write(anyObj)` is a valid operation. When set, it becomes possible to write JavaScript values other than string or `Buffer` if supported by the stream implementation.

Defaults to `false`

- `write` <Function> Implementation for the `stream._write()` method.
- `writev` <Function> Implementation for the `stream._writev()` method.

For example:

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    // Calls the stream.Writable() constructor
    super(options);
  }
}
```

Or, when using pre-ES6 style constructors:

```
const Writable = require('stream').Writable;
const util = require('util');

function MyWritable(options) {
  if (!(this instanceof MyWritable))
    return new MyWritable(options);
  Writable.call(this, options);
}

util.inherits(MyWritable, Writable);
```

Or, using the Simplified Constructor approach:

```
const Writable = require('stream').Writable;

const myWritable = new Writable({
```

```
write(chunk, encoding, callback) {  
  // ...  
},  
writev(chunks, callback) {  
  // ...  
}  
});
```

writable._write(chunk, encoding, callback)

- `chunk <Buffer> | <String>` The chunk to be written. Will **always** be a buffer unless the `decodeStrings` option was set to `false`.
- `encoding <String>` If the chunk is a string, then `encoding` is the character encoding of that string. If `chunk` is a `Buffer`, or if the stream is operating in object mode, `encoding` may be ignored.
- `callback <Function>` Call this function (optionally with an error argument) when processing is complete for the supplied chunk.

All Writable stream implementations must provide a `writable._write()` method to send data to the underlying resource.

Note: Transform streams provide their own implementation of the `writable._write()`.

Note: This function **MUST NOT** be called by application code directly. It should be implemented by child classes, and called only by the internal Writable class methods only.

The `callback` method must be called to signal either that the write completed successfully or failed with an error. The first argument passed to the `callback` must be the `Error` object if the call failed or `null` if the write succeeded.

It is important to note that all calls to `writable.write()` that occur between the time `writable._write()` is called and the `callback` is called will cause the written data to be buffered. Once the `callback` is invoked, the stream will emit a '`'drain'`' event. If a stream implementation is capable of processing multiple

chunks of data at once, the `writable._writev()` method should be implemented.

If the `decodeStrings` property is set in the constructor options, then `chunk` may be a string rather than a Buffer, and `encoding` will indicate the character encoding of the string. This is to support implementations that have an optimized handling for certain string data encodings. If the `decodeStrings` property is explicitly set to `false`, the `encoding` argument can be safely ignored, and `chunk` will always be a Buffer .

The `writable._write()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

writable._writev(chunks, callback)

#

- `chunks <Array>` The chunks to be written. Each chunk has following format:
`{ chunk: ..., encoding: ... }`.
- `callback <Function>` A callback function (optionally with an error argument) to be invoked when processing is complete for the supplied chunks.

Note: This function **MUST NOT** be called by application code directly. It should be implemented by child classes, and called only by the internal Writable class methods only.

The `writable._writev()` method may be implemented in addition to `writable._write()` in stream implementations that are capable of processing multiple chunks of data at once. If implemented, the method will be called with all chunks of data currently buffered in the write queue.

The `writable._writev()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

Errors While Writing

#

It is recommended that errors occurring during the processing of the `writable._write()` and `writable._writev()` methods are reported by invoking the callback and passing the error as the first argument. This will cause an

'error' event to be emitted by the Writable. Throwing an Error from within `writable._write()` can result in expected and inconsistent behavior depending on how the stream is being used. Using the callback ensures consistent and predictable handling of errors.

```
const Writable = require('stream').Writable;

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'))
    } else {
      callback()
    }
  }
});
```

An Example Writable Stream

The following illustrates a rather simplistic (and somewhat pointless) custom Writable stream implementation. While this specific Writable stream instance is not of any real particular usefulness, the example illustrates each of the required elements of a custom `Writable` stream instance:

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    super(options);
  }

  _write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
```

```
        callback(new Error('chunk is invalid'))
    } else {
        callback()
    }
}
```

Implementing a Readable Stream

The `stream.Readable` class is extended to implement a `Readable` stream.

Custom Readable streams *must* call the `new stream.Readable([options])` constructor and implement the `readable._read()` method.

`new stream.Readable(options)`

- `options <Object>`
 - `highWaterMark <Number>` The maximum number of bytes to store in the internal buffer before ceasing to read from the underlying resource. Defaults to `16384` (16kb), or `16` for `objectMode` streams
 - `encoding <String>` If specified, then buffers will be decoded to strings using the specified encoding. Defaults to `null`
 - `objectMode <Boolean>` Whether this stream should behave as a stream of objects. Meaning that `stream.read(n)` returns a single value instead of a Buffer of size n. Defaults to `false`
 - `read <Function>` Implementation for the `stream._read()` method.

For example:

```
const Readable = require('stream').Readable;

class MyReadable extends Readable {
    constructor(options) {
        // Calls the stream.Readable(options) constructor
        super(options);
```

```
 }  
 }
```

Or, when using pre-ES6 style constructors:

```
const Readable = require('stream').Readable;  
const util = require('util');  
  
function MyReadable(options) {  
  if (!(this instanceof MyReadable))  
    return new MyReadable(options);  
  Readable.call(this, options);  
}  
util.inherits(MyReadable, Readable);
```

Or, using the Simplified Constructor approach:

```
const Readable = require('stream').Readable;  
  
const myReadable = new Readable({  
  read(size) {  
    // ...  
  }  
});
```

readable._read(size) #

- **size** <Number> Number of bytes to read asynchronously

Note: This function MUST NOT be called by application code directly. It should be implemented by child classes, and called only by the internal Readable class methods only.

All Readable stream implementations must provide an implementation of the `readable._read()` method to fetch data from the underlying resource.

When `readable._read()` is called, if data is available from the resource, the implementation should begin pushing that data into the read queue using the `this.push(dataChunk)` method. `_read()` should continue reading from the resource and pushing data until `readable.push()` returns `false`. Only when `_read()` is called again after it has stopped should it resume pushing additional data onto the queue.

Note: Once the `readable._read()` method has been called, it will not be called again until the `readable.push()` method is called.

The `size` argument is advisory. For implementations where a "read" is a single operation that returns data can use the `size` argument to determine how much data to fetch. Other implementations may ignore this argument and simply provide data whenever it becomes available. There is no need to "wait" until `size` bytes are available before calling `stream.push(chunk)`.

The `readable._read()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

Readable.push(chunk[, encoding])

#

- `chunk <Buffer> | <Null> | <String>` Chunk of data to push into the read queue
- `encoding <String>` Encoding of String chunks. Must be a valid Buffer encoding, such as `'utf8'` or `'ascii'`
- Returns `<Boolean>` `true` if additional chunks of data may be pushed; `false` otherwise.

When `chunk` is a `Buffer` or `string`, the `chunk` of data will be added to the internal queue for users of the stream to consume. Passing `chunk` as `null` signals the end of the stream (EOF), after which no more data can be written.

When the Readable is operating in paused mode, the data added with

`readable.push()` can be read out by calling the `readable.read()` method when the '`readable`' event is emitted.

When the Readable is operating in flowing mode, the data added with `readable.push()` will be delivered by emitting a '`data`' event.

The `readable.push()` method is designed to be as flexible as possible. For example, when wrapping a lower-level source that provides some form of pause/resume mechanism, and a data callback, the low-level source can be wrapped by the custom Readable instance as illustrated in the following example:

```
// source is an object with readStop() and readStart() methods,  
// and an `ondata` member that gets called when it has data, and  
// an `onend` member that gets called when the data is over.  
  
class SourceWrapper extends Readable {  
  constructor(options) {  
    super(options);  
  
    this._source = getLowlevelSourceObject();  
  
    // Every time there's data, push it into the internal buffer.  
    this._source.ondata = (chunk) => {  
      // if push() returns false, then stop reading from source  
      if (!this.push(chunk))  
        this._source.readStop();  
    };  
  
    // When the source ends, push the EOF-signaling `null` chunk  
    this._source.onend = () => {  
      this.push(null);  
    };  
  }  
  
  // _read will be called when the stream wants to pull more data in
```

```
// the advisory size argument is ignored in this case.  
_read(size) {  
    this._source.readStart();  
}  
}
```

Note: The `readable.push()` method is intended be called only by Readable Implementers, and only from within the `readable._read()` method.

Errors While Reading

It is recommended that errors occurring during the processing of the `readable._read()` method are emitted using the `'error'` event rather than being thrown. Throwing an Error from within `readable._read()` can result in expected and inconsistent behavior depending on whether the stream is operating in flowing or paused mode. Using the `'error'` event ensures consistent and predictable handling of errors.

```
const Readable = require('stream').Readable;  
  
const myReadable = new Readable({  
    read(size) {  
        if (checkSomeErrorCondition()) {  
            process.nextTick(() => this.emit('error', err));  
            return;  
        }  
        // do some work  
    }  
});
```

An Example Counting Stream

The following is a basic example of a Readable stream that emits the numerals from 1 to 1,000,000 in ascending order, and then ends.

```
const Readable = require('stream').Readable;

class Counter extends Readable {
  constructor(opt) {
    super(opt);
    this._max = 1000000;
    this._index = 1;
  }

  _read() {
    var i = this._index++;
    if (i > this._max)
      this.push(null);
    else {
      var str = '' + i;
      var buf = Buffer.from(str, 'ascii');
      this.push(buf);
    }
  }
}
```

Implementing a Duplex Stream

A **Duplex** stream is one that implements both **Readable** and **Writable**, such as a TCP socket connection.

Because Javascript does not have support for multiple inheritance, the `stream.Duplex` class is extended to implement a **Duplex** stream (as opposed to extending the `stream.Readable` and `stream.Writable` classes).

Note: The `stream.Duplex` class prototypically inherits from `stream.Readable` and parasitically from `stream.Writable`.

Custom Duplex streams *must* call the `new stream.Duplex([options])` constructor and implement both the `readable._read()` and `writable._write()` methods.

`new stream.Duplex(options)`

#

- `options <Object>` Passed to both Writable and Readable constructors. Also has the following fields:
 - `allowHalfOpen <Boolean>` Defaults to `true`. If set to `false`, then the stream will automatically end the readable side when the writable side ends and vice versa.
 - `readableObjectMode <Boolean>` Defaults to `false`. Sets `objectMode` for readable side of the stream. Has no effect if `objectMode` is `true`.
 - `writableObjectMode <Boolean>` Defaults to `false`. Sets `objectMode` for writable side of the stream. Has no effect if `objectMode` is `true`.

For example:

```
const Duplex = require('stream').Duplex;

class MyDuplex extends Duplex {
  constructor(options) {
    super(options);
  }
}
```

Or, when using pre-ES6 style constructors:

```
const Duplex = require('stream').Duplex;
const util = require('util');

function MyDuplex(options) {
  if (!(this instanceof MyDuplex))
    return new MyDuplex(options);
  Duplex.call(this, options);
```

```
}

util.inherits(MyDuplex, Duplex);
```

Or, using the Simplified Constructor approach:

```
const Duplex = require('stream').Duplex;

const myDuplex = new Duplex({
  read(size) {
    // ...
  },
  write(chunk, encoding, callback) {
    // ...
  }
});
```

An Example Duplex Stream

The following illustrates a simple example of a Duplex stream that wraps a hypothetical lower-level source object to which data can be written, and from which data can be read, albeit using an API that is not compatible with Node.js streams. The following illustrates a simple example of a Duplex stream that buffers incoming written data via the [Writable](#) interface that is read back out via the [Readable](#) interface.

```
const Duplex = require('stream').Duplex;
const kSource = Symbol('source');

class MyDuplex extends Duplex {
  constructor(source, options) {
    super(options);
    this[kSource] = source;
  }
}
```

```
_write(chunk, encoding, callback) {
  // The underlying source only deals with strings
  if (Buffer.isBuffer(chunk))
    chunk = chunk.toString(encoding);
  this[kSource].writeSomeData(chunk, encoding);
  callback();
}

_read(size) {
  this[kSource].fetchSomeData(size, (data, encoding) => {
    this.push(Buffer.from(data, encoding));
  });
}
}
```

The most important aspect of a Duplex stream is that the Readable and Writable sides operate independently of one another despite co-existing within a single object instance.

Object Mode Duplex Streams

For Duplex streams, `objectMode` can be set exclusively for either the Readable or Writable side using the `readableObjectMode` and `writableObjectMode` options respectively.

In the following example, for instance, a new Transform stream (which is a type of Duplex stream) is created that has an object mode Writable side that accepts JavaScript numbers that are converted to hexadecimal strings on the Readable side.

```
const Transform = require('stream').Transform;

// All Transform streams are also Duplex Streams
const myTransform = new Transform({
```

```
writableObjectMode: true,  
  
transform(chunk, encoding, callback) {  
  // Coerce the chunk to a number if necessary  
  chunk |= 0;  
  
  // Transform the chunk into something else.  
  const data = chunk.toString(16);  
  
  // Push the data onto the readable queue.  
  callback(null, '0'.repeat(data.length % 2) + data);  
}  
});  
  
myTransform.setEncoding('ascii');  
myTransform.on('data', (chunk) => console.log(chunk));  
  
myTransform.write(1);  
  // Prints: 01  
myTransform.write(10);  
  // Prints: 0a  
myTransform.write(100);  
  // Prints: 64
```

Implementing a Transform Stream

#

A **Transform** stream is a **Duplex** stream where the output is computed in some way from the input. Examples include **zlib** streams or **crypto** streams that compress, encrypt, or decrypt data.

Note: There is no requirement that the output be the same size as the input, the same number of chunks, or arrive at the same time. For example, a Hash stream will only ever have a single chunk of output which is provided when the input is ended. A

`zlib` stream will produce output that is either much smaller or much larger than its input.

The `stream.Transform` class is extended to implement a `Transform` stream.

The `stream.Transform` class prototypically inherits from `stream.Duplex` and implements its own versions of the `writable._write()` and `readable._read()` methods. Custom Transform implementations *must* implement the `transform._transform()` method and *may* also implement the `transform._flush()` method.

Note: Care must be taken when using Transform streams in that data written to the stream can cause the Writable side of the stream to become paused if the output on the Readable side is not consumed.

`new stream.Transform(options)`

- `options <Object>` Passed to both Writable and Readable constructors. Also has the following fields:
 - `transform <Function>` Implementation for the `stream._transform()` method.
 - `flush <Function>` Implementation for the `stream._flush()` method.

For example:

```
const Transform = require('stream').Transform;

class MyTransform extends Transform {
  constructor(options) {
    super(options);
  }
}
```

Or, when using pre-ES6 style constructors:

```
const Transform = require('stream').Transform;
const util = require('util');

function MyTransform(options) {
  if (!(this instanceof MyTransform))
    return new MyTransform(options);
  Transform.call(this, options);
}

util.inherits(MyTransform, Transform);
```

Or, using the Simplified Constructor approach:

```
const Transform = require('stream').Transform;

const myTransform = new Transform({
  transform(chunk, encoding, callback) {
    // ...
  }
});
```

Events: 'finish' and 'end'

#

The '`finish`' and '`end`' events are from the `stream.Writable` and `stream.Readable` classes, respectively. The '`finish`' event is emitted after `stream.end()` is called and all chunks have been processed by `stream._transform()`. The '`end`' event is emitted after all data has been output, which occurs after the callback in `transform._flush()` has been called.

`transform._flush(callback)`

#

- `callback <Function>` A callback function (optionally with an error argument) to be called when remaining data has been flushed.

Note: This function MUST NOT be called by application code directly. It should be

implemented by child classes, and called only by the internal Readable class methods only.

In some cases, a transform operation may need to emit an additional bit of data at the end of the stream. For example, a `zlib` compression stream will store an amount of internal state used to optimally compress the output. When the stream ends, however, that additional data needs to be flushed so that the compressed data will be complete.

Custom `Transform` implementations *may* implement the `transform._flush()` method. This will be called when there is no more written data to be consumed, but before the '`end`' event is emitted signaling the end of the `Readable` stream.

Within the `transform._flush()` implementation, the `readable.push()` method may be called zero or more times, as appropriate. The `callback` function must be called when the flush operation is complete.

The `transform._flush()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

`transform._transform(chunk, encoding, callback)`

- `chunk <Buffer> | <String>` The chunk to be transformed. Will **always** be a buffer unless the `decodeStrings` option was set to `false`.
- `encoding <String>` If the chunk is a string, then this is the encoding type. If chunk is a buffer, then this is the special value - 'buffer', ignore it in this case.
- `callback <Function>` A callback function (optionally with an error argument and data) to be called after the supplied `chunk` has been processed.

Note: This function **MUST NOT** be called by application code directly. It should be implemented by child classes, and called only by the internal Readable class methods only.

All Transform stream implementations must provide a `_transform()` method to accept input and produce output. The `transform._transform()` implementation handles the bytes being written, computes an output, then passes that output off to

the readable portion using the `readable.push()` method.

The `transform.push()` method may be called zero or more times to generate output from a single input chunk, depending on how much is to be output as a result of the chunk.

It is possible that no output is generated from any given chunk of input data.

The `callback` function must be called only when the current chunk is completely consumed. The first argument passed to the `callback` must be an `Error` object if an error occurred while processing the input or `null` otherwise. If a second argument is passed to the `callback`, it will be forwarded on to the `readable.push()` method. In other words the following are equivalent:

```
transform.prototype._transform = function (data, encoding, callback) {  
    this.push(data);  
    callback();  
};  
  
transform.prototype._transform = function (data, encoding, callback) {  
    callback(null, data);  
};
```

The `transform._transform()` method is prefixed with an underscore because it is internal to the class that defines it, and should never be called directly by user programs.

Class: `stream.PassThrough`

#

The `stream.PassThrough` class is a trivial implementation of a `Transform` stream that simply passes the input bytes across to the output. Its purpose is primarily for examples and testing, but there are some use cases where `stream.PassThrough` is useful as a building block for novel sorts of streams.

Additional Notes

#

In versions of Node.js prior to v0.10, the Readable stream interface was simpler, but also less powerful and less useful.

- Rather than waiting for calls the `stream.read()` method, '`data`' events would begin emitting immediately. Applications that would need to perform some amount of work to decide how to handle data were required to store read data into buffers so the data would not be lost.
- The `stream.pause()` method was advisory, rather than guaranteed. This meant that it was still necessary to be prepared to receive '`data`' events even when the stream was in a paused state.

In Node.js v0.10, the `Readable` class was added. For backwards compatibility with older Node.js programs, Readable streams switch into "flowing mode" when a '`data`' event handler is added, or when the `stream.resume()` method is called. The effect is that, even when not using the new `stream.read()` method and '`readable`' event, it is no longer necessary to worry about losing '`data`' chunks.

While most applications will continue to function normally, this introduces an edge case in the following conditions:

- No '`data`' event listener is added.
- The `stream.resume()` method is never called.
- The stream is not piped to any writable destination.

For example, consider the following code:

```
// WARNING! BROKEN!
net.createServer((socket) => {

    // we add an 'end' method, but never consume the data
    socket.on('end', () => {
        // It will never get here.
        socket.end('The message was received but was not processed.\n');
    });
})
```

```
}).listen(1337);
```

In versions of Node.js prior to v0.10, the incoming message data would be simply discarded. However, in Node.js v0.10 and beyond, the socket remains paused forever.

The workaround in this situation is to call the `stream.resume()` method to begin the flow of data:

```
// Workaround
net.createServer((socket) => {

  socket.on('end', () => {
    socket.end('The message was received but was not processed.\n');
  });

  // start the flow of data, discarding it.
  socket.resume();

}).listen(1337);
```

In addition to new Readable streams switching into flowing mode, pre-v0.10 style streams can be wrapped in a Readable class using the `readable.wrap()` method.

readable.read(0)

There are some cases where it is necessary to trigger a refresh of the underlying readable stream mechanisms, without actually consuming any data. In such cases, it is possible to call `readable.read(0)`, which will always return `null`.

If the internal read buffer is below the `highWaterMark`, and the stream is not currently reading, then calling `stream.read(0)` will trigger a low-level `stream._read()` call.

While most applications will almost never need to do this, there are situations within Node.js where this is done, particularly in the Readable stream class internals.

readable.push('')

#

Use of `readable.push('')` is not recommended.

Pushing a zero-byte string or `Buffer` to a stream that is not in object mode has an interesting side effect. Because it is a call to `readable.push()`, the call will end the reading process. However, because the argument is an empty string, no data is added to the readable buffer so there is nothing for a user to consume.

StringDecoder

#

Stability: 2 - Stable

The `string_decoder` module provides an API for decoding `Buffer` objects into strings in a manner that preserves encoded multi-byte UTF-8 and UTF-16 characters. It can be accessed using:

```
const StringDecoder = require('string_decoder').StringDecoder;
```

The following example shows the basic use of the `StringDecoder` class.

```
const StringDecoder = require('string_decoder').StringDecoder;
const decoder = new StringDecoder('utf8');

const cent = Buffer.from([0xC2, 0xA2]);
console.log(decoder.write(cent));

const euro = Buffer.from([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

When a `Buffer` instance is written to the `StringDecoder` instance, an internal buffer is used to ensure that the decoded string does not contain any incomplete multibyte characters. These are held in the buffer until the next call to `stringDecoder.write()` or until `stringDecoder.end()` is called.

In the following example, the three UTF-8 encoded bytes of the European Euro symbol (€) are written over three separate operations:

```
const StringDecoder = require('string_decoder').StringDecoder;
const decoder = new StringDecoder('utf8');

decoder.write(Buffer.from([0xE2]));
decoder.write(Buffer.from([0x82]));
console.log(decoder.end(Buffer.from([0xAC])));
```

Class: new `StringDecoder([encoding])`

#

Added in: v0.1.99

- `encoding <string>` The character encoding the `StringDecoder` will use. Defaults to `'utf8'`.

Creates a new `StringDecoder` instance.

`stringDecoder.end(buffer)`

#

Added in: v0.9.3

- `buffer <Buffer>` A `Buffer` containing the bytes to decode.

Returns any remaining input stored in the internal buffer as a string. Bytes representing incomplete UTF-8 and UTF-16 characters will be replaced with substitution characters appropriate for the character encoding.

If the `buffer` argument is provided, one final call to `stringDecoder.write()` is performed before returning the remaining input.

`stringDecoder.write(buffer)`

#

- `buffer <Buffer>` A `Buffer` containing the bytes to decode.

Returns a decoded string, ensuring that any incomplete multibyte characters at the end of the `Buffer` are omitted from the returned string and stored in an internal buffer for the next call to `stringDecoder.write()` or `stringDecoder.end()`.

Timers

Stability: 3 - Locked

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call `require('timers')` to use the API.

The timer functions within Node.js implement a similar API as the timers API provided by Web Browsers but use a different internal implementation that is built around the Node.js Event Loop.

Class: Immediate

This object is created internally and is returned from `setImmediate()`. It can be passed to `clearImmediate()` in order to cancel the scheduled actions.

Class: Timeout

This object is created internally and is returned from `setTimeout()` and `setInterval()`. It can be passed to `clearTimeout` or `clearInterval` (respectively) in order to cancel the scheduled actions.

By default, when a timer is scheduled using either `setTimeout` or `setInterval`, the Node.js event loop will continue running as long as the timer is active. Each of the `Timeout` objects returned by these functions export both `timeout.ref()` and `timeout.unref()` functions that can be used to control this default behavior.

timeout.ref()

#

Added in: v0.9.1

When called, requests that the Node.js event loop *not* exit so long as the `Timeout` is active. Calling `timeout.ref()` multiple times will have no effect.

Note: By default, all `Timeout` objects are "ref'd", making it normally unnecessary to call `timeout.ref()` unless `timeout.unref()` had been called previously.

Returns a reference to the `Timeout`.

timeout.unref()

#

Added in: v0.9.1

When called, the active `Timeout` object will not require the Node.js event loop to remain active. If there is no other activity keeping the event loop running, the process may exit before the `Timeout` object's callback is invoked. Calling `timout.unref()` multiple times will have no effect.

Note: Calling `timout.unref()` creates an internal timer that will wake the Node.js event loop. Creating too many of these can adversely impact performance of the Node.js application.

Returns a reference to the `Timeout`.

Scheduling Timers

#

A timer in Node.js is an internal construct that calls a given function after a certain period of time. When a timer's function is called varies depending on which method was used to create the timer and what other work the Node.js event loop is doing.

setImmediate(callback[, ...arg])

#

Added in: v0.9.1

- `callback <Function>` The function to call at the end of this turn of `the Node.js Event Loop`
- `[, ...arg]` Optional arguments to pass when the `callback` is called.

Schedules the "immediate" execution of the `callback` after I/O events' callbacks and before timers created using `setTimeout()` and `setInterval()` are triggered. Returns an `Immediate` for use with `clearImmediate()`.

When multiple calls to `setImmediate()` are made, the `callback` functions are queued for execution in the order in which they are created. The entire callback queue is processed every event loop iteration. If an immediate timer is queued from inside an executing callback, that timer will not be triggered until the next event loop iteration.

If `callback` is not a function, a `TypeError` will be thrown.

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

#

Added in: v0.0.1

- `callback <Function>` The function to call when the timer elapses.
- `delay <number>` The number of milliseconds to wait before calling the `callback`.
- `[, ...arg]` Optional arguments to pass when the `callback` is called.

Schedules repeated execution of `callback` every `delay` milliseconds. Returns a `Timeout` for use with `clearInterval()`.

When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`.

If `callback` is not a function, a `TypeError` will be thrown.

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

#

Added in: v0.0.1

- `callback <Function>` The function to call when the timer elapses.
- `delay <number>` The number of milliseconds to wait before calling the `callback`.
- `[, ...arg]` Optional arguments to pass when the `callback` is called.

Schedules execution of a one-time `callback` after `delay` milliseconds. Returns a

Timeout for use with `clearTimeout()`.

The `callback` will likely not be invoked in precisely `delay` milliseconds. Node.js makes no guarantees about the exact timing of when callbacks will fire, nor of their ordering. The callback will be called as close as possible to the time specified.

Note: When `delay` is larger than `2147483647` or less than `1`, the `delay` will be set to `1`.

If `callback` is not a function, a `TypeError` will be thrown.

Cancelling Timers

The `setImmediate()`, `setInterval()`, and `setTimeout()` methods each return objects that represent the scheduled timers. These can be used to cancel the timer and prevent it from triggering.

`clearImmediate(immediate)`

Added in: v0.9.1

- `immediate` <Immediate> An `Immediate` object as returned by `setImmediate()`.

Cancels an `Immediate` object created by `setImmediate()`.

`clearInterval(timeout)`

Added in: v0.0.1

- `timeout` <Timeout> A `Timeout` object as returned by `setInterval()`.

Cancels a `Timeout` object created by `setInterval()`.

`clearTimeout(timeout)`

Added in: v0.0.1

- `timeout` <Timeout> A `Timeout` object as returned by `setTimeout()`.

Cancels a `Timeout` object created by `setTimeout()`.

Stability: 2 - Stable

The `tls` module provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL. The module can be accessed using:

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

TLS/SSL Concepts

#

The TLS/SSL is a public/private key infrastructure (PKI). For most common cases, each client and server must have a *private key*.

Private keys can be generated in multiple ways. The example below illustrates use of the OpenSSL command-line interface to generate a 2048-bit RSA private key:

```
openssl genrsa -out ryans-key.pem 2048
```

With TLS/SSL, all servers (and some clients) must have a *certificate*. Certificates are *public keys* that correspond to a private key, and that are digitally signed either by a Certificate Authority or by the owner of the private key (such certificates are referred to as "self-signed"). The first step to obtaining a certificate is to create a *Certificate Signing Request (CSR)* file.

The OpenSSL command-line interface can be used to generate a CSR for a private key:

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

Once the CSR file is generated, it can either be sent to a Certificate Authority for signing or used to generate a self-signed certificate.

Creating a self-signed certificate using the OpenSSL command-line interface is illustrated in the example below:

```
openssl x509 -req -in ryan-csr.pem -signkey ryan-key.pem -out ryan-ce
```

Once the certificate is generated, it can be used to generate a `.pfx` or `.p12` file:

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

Where:

- `in` : is the signed certificate
- `inkey` : is the associated private key
- `certfile` : is a concatenation of all Certificate Authority (CA) certs into a single file, e.g. `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

Perfect Forward Secrecy

The term "**Forward Secrecy**" or "Perfect Forward Secrecy" describes a feature of key-agreement (i.e., key-exchange) methods. That is, the server and client keys are used to negotiate new temporary keys that are used specifically and only for the current communication session. Practically, this means that even if the server's private key is compromised, communication can only be decrypted by eavesdroppers if the attacker manages to obtain the key-pair specifically generated for the session.

Perfect Forward Secrecy is achieved by randomly generating a key pair for key-agreement on every TLS/SSL handshake (in contrast to using the same key for all sessions). Methods implementing this technique are called "ephemeral".

Currently two methods are commonly used to achieve Perfect Forward Secrecy (note the character "E" appended to the traditional abbreviations):

- **DHE** - An ephemeral version of the Diffie Hellman key-agreement protocol.
- **ECDHE** - An ephemeral version of the Elliptic Curve Diffie Hellman key-agreement protocol.

Ephemeral methods may have some performance drawbacks, because key generation is expensive.

To use Perfect Forward Secrecy using **DHE** with the **tls** module, it is required to generate Diffie-Hellman parameters. The following illustrates the use of the OpenSSL command-line interface to generate such parameters:

```
openssl dhparam -outform PEM -out dhparam.pem 2048
```

If using Perfect Foward Secrecy using **ECDHE**, Diffie-Hellman parameters are not required and a default ECDHE curve will be used. The **ecdheCurve** property can be used when creating a TLS Server to specify the name of an alternative curve to use.

ALPN, NPN and SNI

ALPN (Application-Layer Protocol Negotiation Extension), NPN (Next Protocol Negotiation) and, SNI (Server Name Indication) are TLS handshake extensions:

- ALPN/NPN - Allows the use of one TLS server for multiple protocols (HTTP, SPDY, HTTP/2)
- SNI - Allows the use of one TLS server for multiple hostnames with different SSL certificates.

Note: Use of ALPN is recommended over NPN. The NPN extension has never been formally defined or documented and generally not recommended for use.

Client-initiated renegotiation attack mitigation

The TLS protocol allows clients to renegotiate certain aspects of the TLS session. Unfortunately, session renegotiation requires a disproportionate amount of server-

side resources, making it a potential vector for denial-of-service attacks.

To mitigate the risk, renegotiation is limited to three times every ten minutes. An 'error' event is emitted on the `tls.TLSSocket` instance when this threshold is exceeded. The limits are configurable:

- `tls.CLIENT_RENEG_LIMIT <number>` Specifies the number of renegotiation requests. Defaults to 3.
- `tls.CLIENT_RENEG_WINDOW <number>` Specifies the time renegotiation window in seconds. Defaults to 600 (10 minutes).

Note: The default renegotiation limits should not be modified without a full understanding of the implications and risks.

To test the renegotiation limits on a server, connect to it using the OpenSSL command-line client (`openssl s_client -connect address:port`) then input R<CR> (i.e., the letter R followed by a carriage return) multiple times.

Modifying the Default TLS Cipher suite

Node.js is built with a default suite of enabled and disabled TLS ciphers. Currently, the default cipher suite is:

```
ECDHE-RSA-AES128-GCM-SHA256:  
ECDHE-ECDSA-AES128-GCM-SHA256:  
ECDHE-RSA-AES256-GCM-SHA384:  
ECDHE-ECDSA-AES256-GCM-SHA384:  
DHE-RSA-AES128-GCM-SHA256:  
ECDHE-RSA-AES128-SHA256:  
DHE-RSA-AES128-SHA256:  
ECDHE-RSA-AES256-SHA384:  
DHE-RSA-AES256-SHA384:  
ECDHE-RSA-AES256-SHA256:  
DHE-RSA-AES256-SHA256:  
HIGH:
```

```
!aNULL:  
!eNULL:  
!EXPORT:  
!DES:  
!RC4:  
!MD5:  
!PSK:  
!SRP:  
!CAMELLIA
```

This default can be replaced entirely using the `--tls-cipher-list` command line switch. For instance, the following makes `ECDHE-RSA-AES128-GCM-SHA256:!RC4` the default TLS cipher suite:

```
node --tls-cipher-list="ECDHE-RSA-AES128-GCM-SHA256:!RC4"
```

Note: The default cipher suite included within Node.js has been carefully selected to reflect current security best practices and risk mitigation. Changing the default cipher suite can have a significant impact on the security of an application. The `--tls-cipher-list` switch should be used only if absolutely necessary.

Class: `tls.Server`

#

Added in: v0.3.2

The `tls.Server` class is a subclass of `net.Server` that accepts encrypted connections using TLS or SSL.

Event: 'tlsClientError'

#

Added in: v6.0.0

The '`'tlsClientError'`' event is emitted when an error occurs before a secure connection is established. The listener callback is passed two arguments when called:

- `exception` `<Error>` The `Error` object describing the error
- `tlsSocket` `<tls.TLSSocket>` The `tls.TLSSocket` instance from which the error originated.

Event: 'newSession'

#

Added in: v0.9.2

The `'newSession'` event is emitted upon creation of a new TLS session. This may be used to store sessions in external storage. The listener callback is passed three arguments when called:

- `sessionId` - The TLS session identifier
- `sessionData` - The TLS session data
- `callback` `<Function>` A callback function taking no arguments that must be invoked in order for data to be sent or received over the secure connection.

Note: Listening for this event will have an effect only on connections established after the addition of the event listener.

Event: 'OCSPRequest'

#

Added in: v0.11.13

The `'OCSPRequest'` event is emitted when the client sends a certificate status request. The listener callback is passed three arguments when called:

- `certificate` `<Buffer>` The server certificate
- `issuer` `<Buffer>` The issuer's certificate
- `callback` `<Function>` A callback function that must be invoked to provide the results of the OCSP request.

The server's current certificate can be parsed to obtain the OCSP URL and certificate ID; after obtaining an OCSP response, `callback(null, resp)` is then invoked, where `resp` is a `Buffer` instance containing the OCSP response. Both `certificate` and `issuer` are `Buffer` DER-representations of the primary and issuer's certificates. These can be used to obtain the OCSP certificate ID and OCSP endpoint URL.

Alternatively, `callback(null, null)` may be called, indicating that there was no OCSP response.

Calling `callback(err)` will result in a `socket.destroy(err)` call.

The typical flow of an OCSP Request is as follows:

1. Client connects to the server and sends an '`OCSPRequest`' (via the status info extension in `ClientHello`).
2. Server receives the request and emits the '`OCSPRequest`' event, calling the listener if registered.
3. Server extracts the OCSP URL from either the `certificate` or `issuer` and performs an `OCSP request` to the CA.
4. Server receives `OCSPResponse` from the CA and sends it back to the client via the `callback` argument
5. Client validates the response and either destroys the socket or performs a handshake.

Note: The `issuer` can be `null` if the certificate is either self-signed or the issuer is not in the root certificates list. (An issuer may be provided via the `ca` option when establishing the TLS connection.)

Note: Listening for this event will have an effect only on connections established after the addition of the event listener.

Note: An npm module like `asn1.js` may be used to parse the certificates.

Event: 'resumeSession'

#

Added in: v0.9.2

The '`resumeSession`' event is emitted when the client requests to resume a previous TLS session. The listener callback is passed two arguments when called:

- `sessionId` - The TLS/SSL session identifier
- `callback <Function>` A callback function to be called when the prior session has been recovered.

When called, the event listener may perform a lookup in external storage using the

given `sessionId` and invoke `callback(null, sessionId)` once finished. If the session cannot be resumed (i.e., doesn't exist in storage) the callback may be invoked as `callback(null, null)`. Calling `callback(err)` will terminate the incoming connection and destroy the socket.

Note: Listening for this event will have an effect only on connections established after the addition of the event listener.

The following illustrates resuming a TLS session:

```
const tlsSessionStore = {};
server.on('newSession', (id, data, cb) => {
  tlsSessionStore[id.toString('hex')] = data;
  cb();
});
server.on('resumeSession', (id, cb) => {
  cb(null, tlsSessionStore[id.toString('hex')] || null);
});
```

Event: 'secureConnection'

Added in: v0.3.2

The `'secureConnection'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback is passed a single argument when called:

- `tlsSocket <tls.TLSSocket>` The established TLS socket.

The `tlsSocket.authorized` property is a `boolean` indicating whether the client has been verified by one of the supplied Certificate Authorities for the server. If `tlsSocket.authorized` is `false`, then `socket.authorizationError` is set to describe how authorization failed. Note that depending on the settings of the TLS server, unauthorized connections may still be accepted.

The `tlsSocket.npnProtocol` and `tlsSocket.alpnProtocol` properties are

strings that contain the selected NPN and ALPN protocols, respectively. When both NPN and ALPN extensions are received, ALPN takes precedence over NPN and the next protocol is selected by ALPN.

When ALPN has no selected protocol, `tlsSocket.alpnProtocol` returns `false`.

The `tlsSocket.servername` property is a string containing the server name requested via SNI.

server.addContext(hostname, context)

Added in: v0.5.3

- `hostname` `<string>` A SNI hostname or wildcard (e.g. `'*'`)
- `context` `<Object>` An object containing any of the possible properties from the `tls.createSecureContext()` `options` arguments (e.g. `key`, `cert`, `ca`, etc).

The `server.addContext()` method adds a secure context that will be used if the client request's SNS hostname matches the supplied `hostname` (or wildcard).

server.address()

Added in: v0.6.0

Returns the bound address, the address family name, and port of the server as reported by the operating system. See `net.Server.address()` for more information.

server.close([callback])

Added in: v0.3.2

- `callback` `<Function>` An optional listener callback that will be registered to listen for the server instance's `'close'` event.

The `server.close()` method stops the server from accepting new connections.

This function operates asynchronously. The `'close'` event will be emitted when the the server is finally closed.

server.connections

#

Added in: v0.3.2

Returns the current number of concurrent connections on the server.

server.getTicketKeys()

#

Added in: v3.0.0

Returns a `Buffer` instance holding the keys currently used for encryption/decryption of the [TLS Session Tickets](#)

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

#

Added in: v0.3.2

- `port` <number> The TCP/IP port on which to begin listening for connections. A value of `0` (zero) will assign a random port.
- `hostname` <string> The hostname, IPv4, or IPv6 address on which to begin listening for connections. If `undefined`, the server will accept connections on any IPv6 address (`::`) when IPv6 is available, or any IPv4 address (`0.0.0.0`) otherwise.
- `callback` <Function> A callback function to be invoked when the server has begun listening the the `port` and `hostname`.

The `server.listen()` methods instructs the server to begin accepting connections on the specified `port` and `hostname`.

This function operates asynchronously. If the `callback` is given, it will be called when the server has started listening.

See `net.Server` for more information.

server.setTicketKeys(keys)

#

Added in: v3.0.0

- `keys` <Buffer> The keys used for encryption/decryption of the [TLS Session Tickets](#).

Updates the keys for encryption/decryption of the [TLS Session Tickets](#).

Note: The key's `Buffer` should be 48 bytes long. See `ticketKeys` option in `tls.createServer` for more information on how it is used.

Note: Changes to the ticket keys are effective only for future server connections. Existing or currently pending server connections will use the previous keys.

Class: `tls.TLSSocket`

#

Added in: v0.11.4

The `tls.TLSSocket` is a subclass of `net.Socket` that performs transparent encryption of written data and all required TLS negotiation.

Instances of `tls.TLSSocket` implement the duplex `Stream` interface.

Note: Methods that return TLS connection metadata (e.g.

`tls.TLSSocket.getPeerCertificate()` will only return data while the connection is open.

`new tls.TLSSocket(socket[, options])`

#

Added in: v0.11.4

- `socket <net.Socket>` An instance of `net.Socket`
- `options <Object>`
 - `secureContext` : An optional TLS context object from `tls.createSecureContext()`
 - `isServer` : If `true` the TLS socket will be instantiated in server-mode. Defaults to `false`.
 - `server <net.Server>` An optional `net.Server` instance.
 - `requestCert` : Optional, see `tls.createServer()`
 - `rejectUnauthorized` : Optional, see `tls.createServer()`
 - `NPNProtocols` : Optional, see `tls.createServer()`
 - `ALPNProtocols` : Optional, see `tls.createServer()`
 - `SNICallback` : Optional, see `tls.createServer()`
 - `session <Buffer>` An optional `Buffer` instance containing a TLS

session.

- `requestOCSP` <boolean> If `true`, specifies that the OCSP status request extension will be added to the client hello and an '`OCSPResponse`' event will be emitted on the socket before establishing a secure communication

Construct a new `tls.TLSSocket` object from an existing TCP socket.

Event: 'OCSPResponse'

#

Added in: v0.11.13

The '`OCSPResponse`' event is emitted if the `requestOCSP` option was set when the `tls.TLSSocket` was created and an OCSP response has been received. The listener callback is passed a single argument when called:

- `response` <Buffer> The server's OCSP response

Typically, the `response` is a digitally signed object from the server's CA that contains information about server's certificate revocation status.

Event: 'secureConnect'

#

Added in: v0.11.4

The '`secureConnect`' event is emitted after the handshaking process for a new connection has successfully completed. The listener callback will be called regardless of whether or not the server's certificate has been authorized. It is the client's responsibility to check the `tlsSocket.authorized` property to determine if the server certificate was signed by one of the specified CAs. If `tlsSocket.authorized === false`, then the error can be found by examining the `tlsSocket.authorizationError` property. If either ALPN or NPN was used, the `tlsSocket.alpnProtocol` or `tlsSocket.npnProtocol` properties can be checked to determine the negotiated protocol.

`tlsSocket.address()`

#

Added in: v0.11.4

Returns the bound address, the address family name, and port of the underlying socket as reported by the operating system. Returns an object with three properties, e.g., `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

`tlsSocket.authorized`

Added in: v0.11.4

Returns `true` if the peer certificate was signed by one of the CAs specified when creating the `tls.TLSSocket` instance, otherwise `false`.

`tlsSocket.authorizationError`

Added in: v0.11.4

Returns the reason why the peer's certificate was not been verified. This property is set only when `tlsSocket.authorized === false`.

`tlsSocket.encrypted`

Added in: v0.11.4

Always returns `true`. This may be used to distinguish TLS sockets from regular `net.Socket` instances.

`tlsSocket.getCipher()`

Added in: v0.11.4

Returns an object representing the cipher name and the SSL/TLS protocol version that first defined the cipher.

For example: `{ name: 'AES256-SHA', version: 'TLSv1/SSLv3' }`

See `SSL_CIPHER_get_name()` and `SSL_CIPHER_get_version()` in https://www.openssl.org/docs/manmaster/ssl/SSL_CIPHER_get_name.html for more information.

`tlsSocket.getEphemeralKeyInfo()`

Added in: v5.0.0

Returns an object representing the type, name, and size of parameter of an ephemeral key exchange in **Perfect Forward Secrecy** on a client connection. It returns an empty object when the key exchange is not ephemeral. As this is only supported on a client socket; `null` is returned if called on a server socket. The supported types are `'DH'` and `'ECDH'`. The `name` property is available only when

type is 'ECDH'.

For Example: { type: 'ECDH', name: 'prime256v1', size: 256 }

tlsSocket.getPeerCertificate([detailed])

#

Added in: v0.11.4

- `detailed` <boolean> Specify `true` to request that the full certificate chain with the `issuer` property be returned; `false` to return only the top certificate without the `issuer` property.

Returns an object representing the peer's certificate. The returned object has some properties corresponding to the fields of the certificate.

For example:

```
{ subject:  
  { C: 'UK',  
   ST: 'Acknack Ltd',  
   L: 'Rhys Jones',  
   O: 'node.js',  
   OU: 'Test TLS Certificate',  
   CN: 'localhost' },  
  
  issuerInfo:  
  { C: 'UK',  
   ST: 'Acknack Ltd',  
   L: 'Rhys Jones',  
   O: 'node.js',  
   OU: 'Test TLS Certificate',  
   CN: 'localhost' },  
  
  issuer:  
  { ... another certificate ... },  
  raw: < RAW DER buffer >,  
  valid_from: 'Nov 11 09:52:22 2009 GMT',  
  valid_to: 'Nov 6 09:52:22 2029 GMT',
```

```
fingerprint: '2A:7A:C2:DD:E5:F9:CC:53:72:35:99:7A:02:5A:71:38:52:EC:8A  
serialNumber: 'B9B0D332A1AA5635' }
```

If the peer does not provide a certificate, `null` or an empty object will be returned.

tlsSocket.getProtocol()

Added in: v5.7.0

Returns a string containing the negotiated SSL/TLS protocol version of the current connection. The value `'unknown'` will be returned for connected sockets that have not completed the handshaking process. The value `null` will be returned for server sockets or disconnected client sockets.

Example responses include:

- `SSLv3`
- `TLSv1`
- `TLSv1.1`
- `TLSv1.2`
- `unknown`

See https://www.openssl.org/docs/manmaster/ssl/SSL_get_version.html for more information.

tlsSocket.getSession()

Added in: v0.11.4

Returns the ASN.1 encoded TLS session or `undefined` if no session was negotiated. Can be used to speed up handshake establishment when reconnecting to the server.

tlsSocket.getTLSTicket()

Added in: v0.11.4

Returns the TLS session ticket or `undefined` if no session was negotiated.

Note: This only works with client TLS sockets. Useful only for debugging, for session

reuse provide `session` option to `tls.connect()`.

`tlsSocket.localAddress`

#

Added in: v0.11.4

Returns the string representation of the local IP address.

`tlsSocket.localPort`

#

Added in: v0.11.4

Returns the numeric representation of the local port.

`tlsSocket.remoteAddress`

#

Added in: v0.11.4

Returns the string representation of the remote IP address. For example,

`'74.125.127.100'` or `'2001:4860:a005::68'`.

`tlsSocket.remoteFamily`

#

Added in: v0.11.4

Returns the string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

`tlsSocket.remotePort`

#

Added in: v0.11.4

Returns the numeric representation of the remote port. For example, `443`.

`tlsSocket.renegotiate(options, callback)`

#

Added in: v0.11.8

- `options` `<Object>`
 - `rejectUnauthorized` `<boolean>`
 - `requestCert`
- `callback` `<Function>` A function that will be called when the renegotiation request has been completed.

The `tlsSocket.renegotiate()` method initiates a TLS renegotiation process.

Upon completion, the `callback` function will be passed a single argument that is either an `Error` (if the request failed) or `null`.

Note: This method can be used to request a peer's certificate after the secure connection has been established.

Note: When running as the server, the socket will be destroyed with an error after `handshakeTimeout` timeout.

`tlsSocket.setMaxSendFragment(size)`

#

Added in: v0.11.11

- `size` <number> The maximum TLS fragment size. Defaults to `16384`. The maximum value is `16384`.

The `tlsSocket.setMaxSendFragment()` method sets the maximum TLS fragment size. Returns `true` if setting the limit succeeded; `false` otherwise.

Smaller fragment sizes decrease the buffering latency on the client: larger fragments are buffered by the TLS layer until the entire fragment is received and its integrity is verified; large fragments can span multiple roundtrips and their processing can be delayed due to packet loss or reordering. However, smaller fragments add extra TLS framing bytes and CPU overhead, which may decrease overall server throughput.

`tls.connect(options[, callback])`

#

Added in: v0.11.3

- `options` <Object>
 - `host` <string> Host the client should connect to.
 - `port` <number> Port the client should connect to.
 - `socket` <net.Socket> Establish secure connection on a given socket rather than creating a new socket. If this option is specified, `host` and `port` are ignored.
 - `path` <string> Creates unix socket connection to path. If this option is specified, `host` and `port` are ignored.
 - `pfx` <string> | <Buffer> A string or `Buffer` containing the private key,

certificate, and CA certs of the client in PFX or PKCS12 format.

- `key` `<string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s containing the private key of the client in PEM format.
- `passphrase` `<string>` A string containing the passphrase for the private key or pfx.
- `cert` `<string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s containing the certificate key of the client in PEM format.
- `ca` `<string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s of trusted certificates in PEM format. If this is omitted several well known "root" CAs (like VeriSign) will be used. These are used to authorize connections.
- `ciphers` `<string>` A string describing the ciphers to use or exclude, separated by `:`. Uses the same default cipher suite as `tls.createServer()`.
- `rejectUnauthorized` `<boolean>` If `true`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. Defaults to `true`.
- `NPNProtocols` `<string[]> | <Buffer[]>` An array of strings or `Buffer`'s containing supported NPN protocols. `Buffer`'s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g. `['hello', 'world']`.
- `ALPNProtocols` : `<string[]> | <Buffer[]>` An array of strings or `Buffer`'s containing the supported ALPN protocols. `Buffer`'s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler: `['hello', 'world']`.)
- `servername` : `<string>` Server name for the SNI (Server Name Indication) TLS extension.
- `checkServerIdentity(servername, cert)` `<Function>` A callback function to be used when checking the server's hostname against the certificate. This should throw an error if verification fails. The method

should return `undefined` if the `servername` and `cert` are verified.

- `secureProtocol` `<string>` The SSL method to use, e.g., `SSLv3_method` to force SSL version 3. The possible values depend on the version of OpenSSL installed in the environment and are defined in the constant `SSL_METHODS`.
- `secureContext` `<object>` An optional TLS context object as returned by from `tls.createSecureContext(...)`. It can be used for caching client certificates, keys, and CA certificates.
- `session` `<Buffer>` A `Buffer` instance, containing TLS session.
- `minDHSize` `<number>` Minimum size of the DH parameter in bits to accept a TLS connection. When a server offers a DH parameter with a size less than `minDHSize`, the TLS connection is destroyed and an error is thrown. Defaults to `1024`.
- `callback` `<Function>`

Creates a new client connection to the given `options.port` and `options.host`. If `options.host` is omitted, it defaults to `localhost`.

The `callback` function, if specified, will be added as a listener for the '`secureConnect`' event.

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

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

#

Added in: v0.11.3

- `port` `<number>`
- `host` `<string>`
- `options` `<Object>`
 - `host` `<string>` Host the client should connect to.
 - `port` `<number>` Port the client should connect to.
 - `socket` `<net.Socket>` Establish secure connection on a given socket rather than creating a new socket. If this option is specified, `host` and `port` are ignored.

- `path <string>` Creates unix socket connection to path. If this option is specified, `host` and `port` are ignored.
- `pfx <string> | <Buffer>` A string or `Buffer` containing the private key, certificate, and CA certs of the client in PFX or PKCS12 format.
- `key <string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s containing the private key of the client in PEM format.
- `passphrase <string>` A string containing the passphrase for the private key or pfx.
- `cert <string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s containing the certificate key of the client in PEM format.
- `ca <string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s of trusted certificates in PEM format. If this is omitted several well known "root" CAs (like VeriSign) will be used. These are used to authorize connections.
- `ciphers <string>` A string describing the ciphers to use or exclude, separated by `:`. Uses the same default cipher suite as `tls.createServer()`.
- `rejectUnauthorized <boolean>` If `true`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. Defaults to `true`.
- `NPNProtocols <string[]> | <Buffer[]>` An array of strings or `Buffer`'s containing supported NPN protocols. `Buffer`'s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g. `['hello', 'world']`.
- `ALPNProtocols :<string[]> | <Buffer[]>` An array of strings or `Buffer`'s containing the supported ALPN protocols. `Buffer`'s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler: `['hello', 'world']`.)
- `servername :<string>` Server name for the SNI (Server Name Indication) TLS extension.

- `checkServerIdentity(servername, cert)` <Function> A callback function to be used when checking the server's hostname against the certificate. This should throw an error if verification fails. The method should return `undefined` if the `servername` and `cert` are verified.
- `secureProtocol` <string> The SSL method to use, e.g., `SSLv3_method` to force SSL version 3. The possible values depend on the version of OpenSSL installed in the environment and are defined in the constant `SSL_METHODS`.
- `secureContext` <object> An optional TLS context object as returned by from `tls.createSecureContext(...)`. It can be used for caching client certificates, keys, and CA certificates.
- `session` <Buffer> A `Buffer` instance, containing TLS session.
- `minDHSize` <number> Minimum size of the DH parameter in bits to accept a TLS connection. When a server offers a DH parameter with a size less than `minDHSize`, the TLS connection is destroyed and an error is thrown. Defaults to `1024`.
- `callback` <Function>

Creates a new client connection to the given `port` and `host` or `options.port` and `options.host`. (If `host` is omitted, it defaults to `localhost`.)

The `callback` function, if specified, will be added as a listener for the '`'secureConnect'`' event.

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

The following implements a simple "echo server" example:

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

const options = {
  // Necessary only if using the client certificate authentication
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),
```

```
// Necessary only if the server uses the self-signed certificate
ca: [ fs.readFileSync('server-cert.pem') ]

};

const socket = tls.connect(8000, options, () => {
  console.log('client connected',
    socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});

socket.setEncoding('utf8');
socket.on('data', (data) => {
  console.log(data);
});
socket.on('end', () => {
  server.close();
});
```

Or

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

const options = {
  pfx: fs.readFileSync('client.pfx')
};

const socket = tls.connect(8000, options, () => {
  console.log('client connected',
    socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
```

```
});  
socket.setEncoding('utf8');  
socket.on('data', (data) => {  
  console.log(data);  
});  
socket.on('end', () => {  
  server.close();  
});
```

tls.createSecureContext(options)

Added in: v0.11.13

- **options** `<Object>`
 - **pfx** `<string> | <Buffer>` A string or `Buffer` holding the PFX or PKCS12 encoded private key, certificate, and CA certificates.
 - **key** `<string> | <string[]> | <Buffer> | <Object[]>` The private key of the server in PEM format. To support multiple keys using different algorithms, an array can be provided either as an array of key strings or as an array of objects in the format `{pem: key, passphrase: passphrase}`. This option is *required* for ciphers that make use of private keys.
 - **passphrase** `<string>` A string containing the passphrase for the private key or pfx.
 - **cert** `<string>` A string containing the PEM encoded certificate
 - **ca** `<string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s of trusted certificates in PEM format. If omitted, several well known "root" CAs (like VeriSign) will be used. These are used to authorize connections.
 - **crl** `<string> | <string[]>` Either a string or array of strings of PEM encoded CRLs (Certificate Revocation List).
 - **ciphers** `<string>` A string describing the ciphers to use or exclude. Consult <https://www.openssl.org/docs/apps/ciphers.html#CIPHER-LIST-FORMAT> for details on the format.
 - **honorCipherOrder** `<boolean>` If `true`, when a cipher is being selected,

the server's preferences will be used instead of the client preferences.

The `tls.createSecureContext()` method creates a `credentials` object.

If the 'ca' option is not 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>.

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

Added in: v0.3.2

- `options <Object>`
 - `pfx <string> | <Buffer>` A string or `Buffer` containing the private key, certificate and CA certs of the server in PFX or PKCS12 format. (Mutually exclusive with the `key`, `cert`, and `ca` options.)
 - `key <string> | <string[]> | <Buffer> | <Object[]>` The private key of the server in PEM format. To support multiple keys using different algorithms an array can be provided either as a plain array of key strings or an array of objects in the format `{pem: key, passphrase: passphrase}`. This option is *required* for ciphers that make use of private keys.
 - `passphrase <string>` A string containing the passphrase for the private key or pfx.
 - `cert <string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s containing the certificate key of the server in PEM format. (Required)
 - `ca <string> | <string[]> | <Buffer> | <Buffer[]>` A string, `Buffer`, array of strings, or array of `Buffer`'s of trusted certificates in PEM format. If this is omitted several well known "root" CAs (like VeriSign) will be used. These are used to authorize connections.
 - `crl <string> | <string[]>` Either a string or array of strings of PEM encoded CRLs (Certificate Revocation List).
 - `ciphers <string>` A string describing the ciphers to use or exclude, separated by `:`.
 - `ecdhCurve <string>` A string describing a named curve to use for ECDH

key agreement or `false` to disable ECDH. Defaults to `prime256v1` (NIST P-256). Use `crypto.getCurves()` to obtain a list of available curve names. On recent releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve.

- `dhparam <string> | <Buffer>` A string or `Buffer` containing Diffie Hellman parameters, required for **Perfect Forward Secrecy**. Use `openssl dhparam` to create the parameters. The key length must be greater than or equal to 1024 bits, otherwise an error will be thrown. It is strongly recommended to use 2048 bits or larger for stronger security. If omitted or invalid, the parameters are silently discarded and DHE ciphers will not be available.
- `handshakeTimeout <number>` Abort the connection if the SSL/TLS handshake does not finish in the specified number of milliseconds. Defaults to `120` seconds. A '`clientError`' is emitted on the `tls.Server` object whenever a handshake times out.
- `honorCipherOrder <boolean>` When choosing a cipher, use the server's preferences instead of the client preferences. Defaults to `true`.
- `requestCert <boolean>` If `true` the server will request a certificate from clients that connect and attempt to verify that certificate. Defaults to `false`.
- `rejectUnauthorized <boolean>` 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`. Defaults to `false`.
- `NPNProtocols <string[]> | <Buffer>` An array of strings or a `Buffer` naming possible NPN protocols. (Protocols should be ordered by their priority.)
- `ALPNProtocols <string[]> | <Buffer>` An array of strings or a `Buffer` naming possible ALPN protocols. (Protocols should be ordered by their priority.) When the server receives both NPN and ALPN extensions from the client, ALPN takes precedence over NPN and the server does not send an NPN extension to the client.
- `SNICallback(servername, cb) <Function>` A function that will be called if the client supports SNI TLS extension. Two arguments will be passed when called: `servername` and `cb`. `SNICallback` should invoke `cb(null, ctx)`, where `ctx` is a `SecureContext` instance. (`tls.createSecureContext(...)` can be used to get a proper

SecureContext.) If `SNICallback` wasn't provided the default callback with high-level API will be used (see below).

- `sessionTimeout <number>` An integer specifying the number of seconds after which the TLS session identifiers and TLS session tickets created by the server will time out. See `SSL_CTX_set_timeout` for more details.
- `ticketKeys` : A 48-byte `Buffer` instance consisting of a 16-byte prefix, a 16-byte HMAC key, and a 16-byte AES key. This can be used to accept TLS session tickets on multiple instances of the TLS server. Note that this is automatically shared between `cluster` module workers.
- `sessionIdContext <string>` A string containing an opaque identifier for session resumption. If `requestCert` is `true`, the default is a 128 bit truncated SHA1 hash value generated from the command-line. Otherwise, a default is not provided.
- `secureProtocol <string>` The SSL method to use, e.g., `SSLv3_method` to force SSL version 3. The possible values depend on the version of OpenSSL installed in the environment and are defined in the constant `SSL_METHODS`.
- `secureConnectionListener <Function>`

Creates a new `Tls.Server`. The `secureConnectionListener`, if provided, is automatically set as a listener for the '`'secureConnection'` event.

For the `ciphers` option, the default cipher suite is:

ECDHE-RSA-AES128-GCM-SHA256:

ECDHE-ECDSA-AES128-GCM-SHA256:

ECDHE-RSA-AES256-GCM-SHA384:

ECDHE-ECDSA-AES256-GCM-SHA384:

DHE-RSA-AES128-GCM-SHA256:

ECDHE-RSA-AES128-SHA256:

DHE-RSA-AES128-SHA256:

ECDHE-RSA-AES256-SHA384:

DHE-RSA-AES256-SHA384:

ECDHE-RSA-AES256-SHA256:

DHE-RSA-AES256-SHA256:

```
HIGH:  
!aNULL:  
!eNULL:  
!EXPORT:  
!DES:  
!RC4:  
!MD5:  
!PSK:  
!SRP:  
!CAMELLIA
```

The default cipher suite prefers GCM ciphers for [Chrome's 'modern cryptography' setting](#) and also prefers ECDHE and DHE ciphers for Perfect Forward Secrecy, while offering *some* backward compatibility.

128 bit AES is preferred over 192 and 256 bit AES in light of [specific attacks affecting larger AES key sizes](#).

Old clients that rely on insecure and deprecated RC4 or DES-based ciphers (like Internet Explorer 6) cannot complete the handshaking process with the default configuration. If these clients *must* be supported, the [TLS recommendations](#) may offer a compatible cipher suite. For more details on the format, see the [OpenSSL cipher list format documentation](#).

The following illustrates a simple echo server:

```
const tls = require('tls');  
const fs = require('fs');  
  
const options = {  
  key: fs.readFileSync('server-key.pem'),  
  cert: fs.readFileSync('server-cert.pem'),  
  // This is necessary only if using the client certificate authentication
```

```
requestCert: true,  
  
    // This is necessary only if the client uses the self-signed certificate  
    ca: [ fs.readFileSync('client-cert.pem') ]  
};  
  
const server = tls.createServer(options, (socket) => {  
    console.log('server connected',  
        socket.authorized ? 'authorized' : 'unauthorized');  
    socket.write('welcome!\n');  
    socket.setEncoding('utf8');  
    socket.pipe(socket);  
});  
server.listen(8000, () => {  
    console.log('server bound');  
});
```

Or

```
const tls = require('tls');  
const fs = require('fs');  
  
const options = {  
    pfx: fs.readFileSync('server.pfx'),  
  
    // This is necessary only if using the client certificate authentication  
    requestCert: true,  
};  
  
const server = tls.createServer(options, (socket) => {  
    console.log('server connected',  
        socket.authorized ? 'authorized' : 'unauthorized');
```

```
socket.write('welcome!\n');
socket.setEncoding('utf8');
socket.pipe(socket);
});
server.listen(8000, () => {
  console.log('server bound');
});
```

This server can be tested by connecting to it using `openssl s_client`:

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

tls.getciphers()

#

Added in: v0.10.2

Returns an array with the names of the supported SSL ciphers.

For example:

```
console.log(tls.getciphers()); // ['AES128-SHA', 'AES256-SHA', ...]
```

Deprecated APIs

#

Class: CryptoStream

#

Added in: v0.3.4 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use [tls.TLSSocket](#) instead.

The `tls.CryptoStream` class represents a stream of encrypted data. This class has been deprecated and should no longer be used.

cryptoStream.bytesWritten

#

Added in: v0.3.4 Deprecated since: v0.11.3

The `cryptoStream.bytesWritten` property returns the total number of bytes written to the underlying socket *including* the bytes required for the implementation of the TLS protocol.

Class: SecurePair

#

Added in: v0.3.2 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use [`tls.TLSSocket`](#) instead.

Returned by `tls.createSecurePair()`.

Event: 'secure'

#

Added in: v0.3.2 Deprecated since: v0.11.3

The '`secure`' event is emitted by the `SecurePair` object once a secure connection has been established.

As with checking for the server `secureConnection` event, `pair.cleartext.authorized` should be inspected to confirm whether the certificate used is properly authorized.

`tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])`

#

Added in: v0.3.2 Deprecated since: v0.11.3

Stability: 0 - Deprecated: Use [`tls.TLSSocket`](#) instead.

- `context` `<Object>` A secure context object as returned by `tls.createSecureContext()`
- `isServer` `<boolean>` `true` to specify that this TLS connection should be opened as a server.

- `requestCert` <boolean> `true` to specify whether a server should request a certificate from a connecting client. Only applies when `isServer` is `true`.
- `rejectUnauthorized` <boolean> `true` to specify whether a server should automatically reject clients with invalid certificates. Only applies when `isServer` is `true`.
- `options`
 - `secureContext` : An optional TLS context object from `tls.createSecureContext()`
 - `isServer` : If `true` the TLS socket will be instantiated in server-mode. Defaults to `false`.
 - `server` <`net.Server`> An optional `net.Server` instance
 - `requestCert` : Optional, see `tls.createServer()`
 - `rejectUnauthorized` : Optional, see `tls.createServer()`
 - `NPNProtocols` : Optional, see `tls.createServer()`
 - `ALPNProtocols` : Optional, see `tls.createServer()`
 - `SNICallback` : Optional, see `tls.createServer()`
 - `session` <`Buffer`> An optional `Buffer` instance containing a TLS session.
 - `requestOCSP` <boolean> If `true`, specifies that the OCSP status request extension will be added to the client hello and an '`OCSPResponse`' event will be emitted on the socket before establishing a secure communication

Creates a new secure pair object with two streams, one of which reads and writes the encrypted data and the other of which reads and writes the cleartext data. Generally, the encrypted stream is piped to/from an incoming encrypted data stream and the cleartext one is used as a replacement for the initial encrypted stream.

`tls.createSecurePair()` returns a `tls.SecurePair` object with `cleartext` and `encrypted` stream properties.

Note: `cleartext` has the same API as `tls.TLSSocket`.

Note: The `tls.createSecurePair()` method is now deprecated in favor of

`tls.TLSSocket()`. For example, the code:

```
pair = tls.createSecurePair( ... );
pair.encrypted.pipe(socket);
socket.pipe(pair.encrypted);
```

can be replaced by:

```
secure_socket = tls.TLSSocket(socket, options);
```

where `secure_socket` has the same API as `pair.cleartext`.

TTY

Stability: 2 - Stable

The `tty` module provides the `tty.ReadStream` and `tty.WriteStream` classes. In most cases, it will not be necessary or possible to use this module directly. However, it can be accessed using:

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

When Node.js detects that it is being run inside a text terminal ("TTY") context, the `process.stdin` will, by default, be initialized as an instance of `tty.ReadStream` and both `process.stdout` and `process.stderr` will, by default be instances of `tty.WriteStream`. The preferred method of determining whether Node.js is being run within a TTY context is to check that the value of the `process.stdout.isTTY` property is `true`:

```
$ node -p -e "Boolean(process.stdout.isTTY)"
```

```
true
```

```
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

In most cases, there should be little to no reason for an application to create instances of the `tty.ReadStream` and `tty.WriteStream` classes.

Class: `tty.ReadStream`

#

Added in: v0.5.8

The `tty.ReadStream` class is a subclass of `net.Socket` that represents the readable side of a TTY. In normal circumstances `process.stdin` will be the only `tty.ReadStream` instance in a Node.js process and there should be no reason to create additional instances.

`readStream.isRaw`

#

Added in: v0.7.7

A `boolean` that is `true` if the TTY is currently configured to operate as a raw device. Defaults to `false`.

`readStream.setRawMode(mode)`

#

Added in: v0.7.7

- `mode <boolean>` If `true`, configures the `tty.ReadStream` to operate as a raw device. If `false`, configures the `tty.ReadStream` to operate in its default mode. The `readStream.isRaw` property will be set to the resulting mode.

Class: `tty.WriteStream`

#

Added in: v0.5.8

The `tty.WriteStream` class is a subclass of `net.Socket` that represents the writable side of a TTY. In normal circumstances, `process.stdout` and `process.stderr` will be the only `tty.WriteStream` instances created for a Node.js process and there should be no reason to create additional instances.

Event: 'resize'

Added in: v0.7.7

The `'resize'` event is emitted whenever either of the `writeStream.columns` or `writeStream.rows` properties have changed. No arguments are passed to the listener callback when called.

```
process.stdout.on('resize', () => {
  console.log('screen size has changed!');
  console.log(`>${process.stdout.columns}x${process.stdout.rows}`);
});
```

writeStream.columns

Added in: v0.7.7

A `number` specifying the number of columns the TTY currently has. This property is updated whenever the `'resize'` event is emitted.

writeStream.rows

Added in: v0.7.7

A `number` specifying the number of rows the TTY currently has. This property is updated whenever the `'resize'` event is emitted.

tty.isatty(fd)

Added in: v0.5.8

- `fd` <number> A numeric file descriptor

The `tty.isatty()` method returns `true` if the given `fd` is associated with a TTY and `false` if it is not.

URL

The `url` module provides utilities for URL resolution and parsing. It can be accessed using:

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

URL Strings and URL Objects

#

A URL string is a structured string containing multiple meaningful components. When parsed, a URL object is returned containing properties for each of these components.

The following details each of the components of a parsed URL. The example `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'` is used to illustrate each.

href						
protocol	auth	host	path			
		hostname	port	pathname	search	
					query	
<code>" http://user:pass @ host.com : 8080 /p/a/t/h ? query=string</code>						

All spaces in the `""` line should be ignored -- they're purely for formality.

The `href` property is the full URL string that was parsed with both the `protocol` and `host` components converted to lower-case.

For example: '`http://user:pass@host.com:8080/p/a/t/h?query=string#hash`'

`urlObject.protocol`

The `protocol` property identifies the URL's lower-cased protocol scheme.

For example: '`http:`'

`urlObject.slashes`

The `slashes` property is a `boolean` with a value of `true` if two ASCII forward-slash characters (/) are required following the colon in the `protocol`.

`urlObject.host`

The `host` property is the full lower-cased host portion of the URL, including the `port` if specified.

For example: '`host.com:8080`'

`urlObject.auth`

The `auth` property is the username and password portion of the URL, also referred to as "userinfo". This string subset follows the `protocol` and double slashes (if present) and precedes the `host` component, delimited by an ASCII "at sign" (@). The format of the string is `{username}[:{password}]`, with the `[:{password}]` portion being optional.

For example: '`user:pass`'

`urlObject.hostname`

The `hostname` property is the lower-cased host name portion of the `host` component *without* the `port` included.

For example: 'host.com'

urlObject.port

The port property is the numeric port portion of the host component.

For example: '8080'

urlObject.pathname

The pathname property consists of the entire path section of the URL. This is everything following the host (including the port) and before the start of the query or hash components, delimited by either the ASCII question mark (?) or hash (#) characters.

For example '/p/a/t/h'

No decoding of the path string is performed.

urlObject.search

The search property consists of the entire "query string" portion of the URL, including the leading ASCII question mark (?) character.

For example: '?query=string'

No decoding of the query string is performed.

urlObject.path

The path property is a concatenation of the pathname and search components.

For example: '/p/a/t/h?query=string'

No decoding of the path is performed.

urlObject.query

The query property is either the "params" portion of the query string (everything except the leading ASCII question mark (?), or an object returned by the

`querystring` module's `parse()` method:

For example: `'query=string'` or `{'query': 'string'}`

If returned as a string, no decoding of the query string is performed. If returned as an object, both keys and values are decoded.

urlObject.hash

The `hash` property consists of the "fragment" portion of the URL including the leading ASCII hash (`#`) character.

For example: `'#hash'`

url.format(urlObject)

Added in: v0.1.25

- `urlObject <Object> | <String>` A URL object (as returned by `url.parse()` or constructed otherwise). If a string, it is converted to an object by passing it to `url.parse()`.

The `url.format()` method returns a formatted URL string derived from `urlObject`.

If `urlObject` is not an object or a string, `url.parse()` will throw a `TypeError`.

The formatting process operates as follows:

- A new empty string `result` is created.
- If `urlObject.protocol` is a string, it is appended as-is to `result`.
- Otherwise, if `urlObject.protocol` is not `undefined` and is not a string, an `Error` is thrown.
- For all string values of `urlObject.protocol` that do not end with an ASCII colon (`:`) character, the literal string `:` will be appended to `result`.
- If either the `urlObject.slashes` property is true, `urlObject.protocol` begins with one of `http`, `https`, `ftp`, `gopher`, or `file`, or `urlObject.protocol` is `undefined`, the literal string `//` will be appended to

result .

- If the value of the `urlObject.auth` property is truthy, and either `urlObject.host` or `urlObject.hostname` are not `undefined`, the value of `urlObject.auth` will be coerced into a string and appended to `result` followed by the literal string `@`.
- If the `urlObject.host` property is `undefined` then:
 - If the `urlObject.hostname` is a string, it is appended to `result`.
 - Otherwise, if `urlObject.hostname` is not `undefined` and is not a string, an **Error** is thrown.
 - If the `urlObject.port` property value is truthy, and `urlObject.hostname` is not `undefined` :
 - The literal string `:` is appended to `result`, and
 - The value of `urlObject.port` is coerced to a string and appended to `result`.
- Otherwise, if the `urlObject.host` property value is truthy, the value of `urlObject.host` is coerced to a string and appended to `result` .
- If the `urlObject.pathname` property is a string that is not an empty string:
 - If the `urlObject.pathname` does not start with an ASCII forward slash `(/)`, then the literal string `'/` is appended to `result` .
 - The value of `urlObject.pathname` is appended to `result` .
- Otherwise, if `urlObject.pathname` is not `undefined` and is not a string, an **Error** is thrown.
- If the `urlObject.search` property is `undefined` and if the `urlObject.query` property is an `Object` , the literal string `?` is appended to `result` followed by the output of calling the `querystring` module's `stringify()` method passing the value of `urlObject.query` .
- Otherwise, if `urlObject.search` is a string:
 - If the value of `urlObject.search` does not start with the ASCII question mark `(?)` character, the literal string `?` is appended to `result` .
 - The value of `urlObject.search` is appended to `result` .
- Otherwise, if `urlObject.search` is not `undefined` and is not a string, an **Error** is thrown.

- If the `urlObject.hash` property is a string:
 - If the value of `urlObject.hash` does not start with the ASCII hash (#) character, the literal string # is appended to `result`.
 - The value of `urlObject.hash` is appended to `result`.
- Otherwise, if the `urlObject.hash` property is not `undefined` and is not a string, an `Error` is thrown.
- `result` is returned.

`url.parse(urlString[, parseQueryString[, slashesDenoteHost]])`

Added in: v0.1.25

- `urlString` <string> The URL string to parse.
- `parseQueryString` <boolean> If `true`, the `query` property will always be set to an object returned by the `querystring` module's `parse()` method. If `false`, the `query` property on the returned URL object will be an unparsed, undecoded string. Defaults to `false`.
- `slashesDenoteHost` <boolean> If `true`, the first token after the literal string `//` and preceding the next `/` will be interpreted as the `host`. For instance, given `//foo/bar`, the result would be `{host: 'foo', pathname: '/bar'}` rather than `{pathname: '//foo/bar'}`. Defaults to `false`.

The `url.parse()` method takes a URL string, parses it, and returns a URL object.

`url.resolve(from, to)`

Added in: v0.1.25

- `from` <string> The Base URL being resolved against.
- `to` <string> The HREF URL being resolved.

The `url.resolve()` method resolves a target URL relative to a base URL in a manner similar to that of a Web browser resolving an anchor tag HREF.

For example:

```
url.resolve('/one/two/three', 'four')           // '/one/two/four'  
url.resolve('http://example.com/', '/one')      // 'http://example.com/one'  
url.resolve('http://example.com/one', '/two')    // 'http://example.com/two'
```

Escaped Characters

URLs are only permitted to contain a certain range of characters. Spaces (' ') and the following characters will be automatically escaped in the properties of URL objects:

```
< > " ` \r \n \t { } | \ ^ '
```

For example, the ASCII space character (' ') is encoded as `%20`. The ASCII forward slash (/) character is encoded as `%3C`.

util

Stability: 2 - Stable

The `util` module is primarily designed to support the needs of Node.js' own internal APIs. However, many of the utilities are useful for application and module developers as well. It can be accessed using:

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

util.debuglog(section)

- `section` `<String>` A string identifying the portion of the application for which the `debuglog` function is being created.
- Returns: `<Function>` The logging function

The `util.debuglog()` method is used to create a function that conditionally writes debug messages to `stderr` based on the existence of the `NODE_DEBUG` environment variable. If the `section` name appears within the value of that environment variable, then the returned function operates similar to `console.error()`. If not, then the returned function is a no-op.

For example:

```
const util = require('util');
const debuglog = util.debuglog('foo');

debuglog('hello from foo [%d]', 123);
```

If this program is run with `NODE_DEBUG=foo` in the environment, then it will output something like:

```
F00 3245: hello from foo [123]
```

where `3245` is the process id. If it is not run with that environment variable set, then it will not print anything.

Multiple comma-separated `section` names may be specified in the `NODE_DEBUG` environment variable. For example: `NODE_DEBUG=fs,net,tls`.

util.deprecate(function, string)

#

The `util.deprecate()` method wraps the given `function` in such a way that it is marked as deprecated.

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

exports.puts = util.deprecate(() => {
  for (var i = 0, len = arguments.length; i < len; ++i) {
```

```
    process.stdout.write(arguments[i] + '\n');
}
}, 'util.puts: Use console.log instead');
```

When called, `util.deprecate()` will return a function that will emit a `DeprecationWarning` using the `process.on('warning')` event. By default, this warning will be emitted and printed to `stderr` exactly once, the first time it is called. After the warning is emitted, the wrapped `function` is called.

If either the `--no-deprecation` or `--no-warnings` command line flags are used, or if the `process.noDeprecation` property is set to `true` prior to the first deprecation warning, the `util.deprecate()` method does nothing.

If the `--trace-deprecation` or `--trace-warnings` command line flags are set, or the `process.traceDeprecation` property is set to `true`, a warning and a stack trace are printed to `stderr` the first time the deprecated function is called.

If the `--throw-deprecation` command line flag is set, or the `process.throwDeprecation` property is set to `true`, then an exception will be thrown when the deprecated function is called.

The `--throw-deprecation` command line flag and `process.throwDeprecation` property take precedence over `--trace-deprecation` and `process.traceDeprecation`.

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

- `format <string>` A `printf`-like format string.

The `util.format()` method returns a formatted string using the first argument as a `printf`-like format.

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

- `%s` - String.
- `%d` - Number (both integer and float).
- `%j` - JSON. Replaced with the string '`[Circular]`' if the argument contains circular references.
- `%%` - single percent sign (`'%'`). This does not consume an argument.

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

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

If there are more arguments passed to the `util.format()` method than the number of placeholders, the extra arguments are coerced into strings (for objects and symbols, `util.inspect()` is used) then concatenated to the returned string, each delimited by a space.

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

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

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

util.inherits(constructor, superConstructor)

#

Note: usage of `util.inherits()` is discouraged. Please use the ES6 `class` and `extends` keywords to get language level inheritance support. Also note that the two styles are *semantically incompatible*.

- `constructor` `<Function>`

- `superConstructor` <Function>

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.

```
const util = require('util');
const EventEmitter = require('events');

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

util.inherits(MyStream, EventEmitter);

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

const stream = new MyStream();

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

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

util.inspect(object[, options])

#

- `object` <any> Any JavaScript primitive or Object.

- `options <Object>`
 - `showHidden <boolean>` If `true`, the `object`'s non-enumerable symbols and properties will be included in the formatted result. Defaults to `false`.
 - `depth <number>` Specifies the number of times to recurse while formatting the `object`. This is useful for inspecting large complicated objects. Defaults to `2`. To make it recurse indefinitely pass `null`.
 - `colors <boolean>` If `true`, the output will be styled with ANSI color codes. Defaults to `false`. Colors are customizable, see [Customizing `util.inspect colors`](#).
 - `customInspect <boolean>` If `false`, then custom `inspect(depth, opts)` functions exported on the `object` being inspected will not be called. Defaults to `true`.
 - `showProxy <boolean>` If `true`, then objects and functions that are `Proxy` objects will be introspected to show their `target` and `handler` objects. Defaults to `false`.
 - `maxArrayLength <number>` Specifies the maximum number of array and `TypedArray` elements to include when formatting. Defaults to `100`. Set to `null` to show all array elements. Set to `0` or negative to show no array elements.
 - `breakLength <number>` The length at which an object's keys are split across multiple lines. Set to `Infinity` to format an object as a single line. Defaults to `60` for legacy compatibility.

The `util.inspect()` method returns a string representation of `object` that is primarily useful for debugging. Additional `options` may be passed that alter certain aspects of the formatted string.

The following example inspects all properties of the `util` object:

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

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

Values may supply their own custom `inspect(depth, opts)` functions, when

called these receive the current `depth` in the recursive inspection, as well as the `options` object passed to `util.inspect()`.

Customizing `util.inspect` colors

Color output (if enabled) of `util.inspect` is customizable globally via the `util.inspect.styles` and `util.inspect.colors` properties.

`util.inspect.styles` is a map associating a style name to a color from `util.inspect.colors`.

The default styles and associated colors are:

- `number` - `yellow`
- `boolean` - `yellow`
- `string` - `green`
- `date` - `magenta`
- `regexp` - `red`
- `null` - `bold`
- `undefined` - `grey`
- `special` - `cyan` (only applied to functions at this time)
- `name` - (no styling)

The predefined color codes are: `white`, `grey`, `black`, `blue`, `cyan`, `green`, `magenta`, `red` and `yellow`. There are also `bold`, `italic`, `underline` and `inverse` codes.

Color styling uses ANSI control codes that may not be supported on all terminals.

Custom `inspect()` function on Objects

Objects may also define their own `inspect(depth, opts)` function that `util.inspect()` will invoke and use the result of when inspecting the object:

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

```
const obj = { name: 'nate' };
obj.inspect = function(depth) {
  return ` ${this.name} `;
};

util.inspect(obj);
// "{nate}"
```

Custom `inspect(depth, opts)` functions typically return a string but may return a value of any type that will be formatted accordingly by `util.inspect()`.

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

const obj = { foo: 'this will not show up in the inspect() output' };
obj.inspect = function(depth) {
  return { bar: 'baz' };
};

util.inspect(obj);
// "{ bar: 'baz' }"
```

Deprecated APIs

The following APIs have been deprecated and should no longer be used. Existing applications and modules should be updated to find alternative approaches.

util.debug(string)

Stability: 0 - Deprecated: Use `console.error()` instead.

- `string <string>` The message to print to `stderr`

Deprecated predecessor of `console.error`.

util.error([...])

#

Stability: 0 - Deprecated: Use `console.error()` instead.

- `string <string>` The message to print to `stderr`

Deprecated predecessor of `console.error`.

util.isArray(object)

#

Stability: 0 - Deprecated

- `object <any>`

Internal alias for `Array.isArray`.

Returns `true` if the given `object` is an `Array`. Otherwise, returns `false`.

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

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

util.isBoolean(object)

#

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `Boolean`. Otherwise, returns `false`.

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

util.isBoolean(1);
  // false
util.isBoolean(0);
  // false
util.isBoolean(false);
  // true
```

`util.isBuffer(object)`

#

Stability: 0 - Deprecated: Use [Buffer.isBuffer\(\)](#) instead.

- `object <any>`

Returns `true` if the given `object` is a `Buffer`. Otherwise, returns `false`.

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

util.isBuffer({ length: 0 });
  // false
util.isBuffer([]);
  // false
util.isBuffer(Buffer.from('hello world'));
  // true
```

`util.isDate(object)`

#

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `Date`. Otherwise, returns `false`.

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

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

util.isError(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is an `Error`. Otherwise, returns `false`.

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

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

Note that this method relies on `Object.prototype.toString()` behavior. It is possible to obtain an incorrect result when the `object` argument manipulates `@@toStringTag`.

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

const obj = { name: 'Error', message: 'an error occurred' };

util.isError(obj);
// false

obj[Symbol.toStringTag] = 'Error';
util.isError(obj);
// true
```

util.isFunction(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `Function`. Otherwise, returns `false`.

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

function Foo() {}
const Bar = function() {};

util.isFunction({});
// false

util.isFunction(Foo);
// true

util.isFunction(Bar);
// true
```

util.isNull(object)

#

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is strictly `null`. Otherwise, returns `false`.

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

util.isNull(0);
  // false
util.isNull(undefined);
  // false
util.isNull(null);
  // true
```

util.isNullOrUndefined(object)

#

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is `null` or `undefined`. Otherwise, returns `false`.

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

util.isNullOrUndefined(0);
  // false
util.isNullOrUndefined(undefined);
  // true
util.isNullOrUndefined(null);
```

```
// true
```

util.isNumber(object)

#

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `Number`. Otherwise, returns `false`.

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

util.isNumber(false);
  // false
util.isNumber(Infinity);
  // true
util.isNumber(0);
  // true
util.isNumber(NaN);
  // true
```

utilisObject(object)

#

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is strictly an `Object` and not a `Function`. Otherwise, returns `false`.

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

```
util.isObject(5);
  // false
util.isObject(null);
  // false
util.isObject({});
  // true
util.isObject(function(){});
  // false
```

util.isPrimitive(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a primitive type. Otherwise, returns `false`.

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

util.isPrimitive(5);
  // true
util.isPrimitive('foo');
  // true
util.isPrimitive(false);
  // true
util.isPrimitive(null);
  // true
util.isPrimitive(undefined);
  // true
util.isPrimitive({});
  // false
util.isPrimitive(function() {});
  // false
```

```
util.isPrimitive(/^\$/);
// false

util.isPrimitive(new Date());
// false
```

util.isRegExp(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `RegExp`. Otherwise, returns `false`.

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

util.isRegExp(/some regexp/);
// true

util.isRegExp(new RegExp('another regexp'));
// true

util.isRegExp({});
// false
```

util.isString(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `string`. Otherwise, returns `false`.

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

```
util.isString('');  
  // true  
util.isString('foo');  
  // true  
util.isString(String('foo'));  
  // true  
util.isString(5);  
  // false
```

util.isSymbol(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is a `Symbol`. Otherwise, returns `false`.

```
const util = require('util');  
  
util.isSymbol(5);  
  // false  
util.isSymbol('foo');  
  // false  
util.isSymbol(Symbol('foo'));  
  // true
```

util.isUndefined(object)

Stability: 0 - Deprecated

- `object <any>`

Returns `true` if the given `object` is `undefined`. Otherwise, returns `false`.

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

const foo = undefined;
util.isUndefined(5);
// false

util.isUndefined(foo);
// true

util.isUndefined(null);
// false
```

util.log(string)

Stability: 0 - Deprecated: Use a third party module instead.

- `string <string>`

The `util.log()` method prints the given `string` to `stdout` with an included timestamp.

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

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

util.print([...])

Stability: 0 - Deprecated: Use `console.log()` instead.

Deprecated predecessor of `console.log`.

`util.puts(...)`

#

Stability: 0 - Deprecated: Use [console.log\(\)](#) instead.

Deprecated predecessor of `console.log`.

`util._extend(obj)`

#

Stability: 0 - Deprecated: Use `Object.assign()` instead.

The `util._extend()` method was never intended to be used outside of internal Node.js modules. The community found and used it anyway.

It is deprecated and should not be used in new code. JavaScript comes with very similar built-in functionality through `Object.assign()`.

V8

#

The `v8` module exposes APIs that are specific to the version of `V8` built into the Node.js binary. It can be accessed using:

```
const v8 = require('v8');
```

Note: The APIs and implementation are subject to change at any time.

v8.getHeapStatistics()

#

Added in: v1.0.0

Returns an object with the following properties:

- `total_heap_size` <number>
- `total_heap_size_executable` <number>
- `total_physical_size` <number>

- `total_available_size` <number>
- `used_heap_size` <number>
- `heap_size_limit` <number>

For example:

```
{  
    total_heap_size: 7326976,  
    total_heap_size_executable: 4194304,  
    total_physical_size: 7326976,  
    total_available_size: 1152656,  
    used_heap_size: 3476208,  
    heap_size_limit: 1535115264  
}
```

v8.getHeapSpaceStatistics()

#

Added in: v6.0.0

Returns statistics about the V8 heap spaces, i.e. the segments which make up the V8 heap. Neither the ordering of heap spaces, nor the availability of a heap space can be guaranteed as the statistics are provided via the V8 [GetHeapSpaceStatistics](#) function and may change from one V8 version to the next.

The value returned is an array of objects containing the following properties:

- `space_name` <string>
- `space_size` <number>
- `space_used_size` <number>
- `space_available_size` <number>
- `physical_space_size` <number>

For example:

```
[  
  {  
    "space_name": "new_space",  
    "space_size": 2063872,  
    "space_used_size": 951112,  
    "space_available_size": 80824,  
    "physical_space_size": 2063872  
  },  
  {  
    "space_name": "old_space",  
    "space_size": 3090560,  
    "space_used_size": 2493792,  
    "space_available_size": 0,  
    "physical_space_size": 3090560  
  },  
  {  
    "space_name": "code_space",  
    "space_size": 1260160,  
    "space_used_size": 644256,  
    "space_available_size": 960,  
    "physical_space_size": 1260160  
  },  
  {  
    "space_name": "map_space",  
    "space_size": 1094160,  
    "space_used_size": 201608,  
    "space_available_size": 0,  
    "physical_space_size": 1094160  
  },  
  {  
    "space_name": "large_object_space",  
    "space_size": 0,  
    "space_used_size": 0,
```

```
"space_available_size": 1490980608,  
"physical_space_size": 0  
}  
]
```

v8.setFlagsFromString(string)

Added in: v1.0.0

The `v8.setFlagsFromString()` method can be used to programmatically set V8 command line flags. This method should be used with care. Changing settings after the VM has started may result in unpredictable behavior, including crashes and data loss; or it may simply do nothing.

The V8 options available for a version of Node.js may be determined by running `node --v8-options`. An unofficial, community-maintained list of options and their effects is available [here](#).

Usage:

```
// Print GC events to stdout for one minute.  
const v8 = require('v8');  
v8.setFlagsFromString('--trace_gc');  
setTimeout(function() { v8.setFlagsFromString('--notrace_gc'); }, 60e3);
```

Executing JavaScript

Stability: 2 - Stable

The `vm` module provides APIs for compiling and running code within V8 Virtual Machine contexts. It can be accessed using:

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

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

Class: `vm.Script`

#

Added in: v0.3.1

Instances of the `vm.Script` class contain precompiled scripts that can be executed in specific sandboxes (or "contexts").

`new vm.Script(code, options)`

#

Added in: v0.3.1

- `code` <string> The JavaScript code to compile.
- `options`
 - `filename` <string> Specifies the filename used in stack traces produced by this script.
 - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
 - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
 - `displayErrors` <boolean> When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
 - `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.
 - `cachedData` <Buffer> Provides an optional `Buffer` with V8's code cache data for the supplied source. When supplied, the `cachedDataRejected` value will be set to either `true` or `false` depending on acceptance of the data by V8.
 - `produceCachedData` <boolean> When `true` and no `cachedData` is present, V8 will attempt to produce code cache data for `code`. Upon

success, a `Buffer` with V8's code cache data will be produced and stored in the `cachedData` property of the returned `vm.Script` instance. The `cachedDataProduced` value will be set to either `true` or `false` depending on whether code cache data is produced successfully.

Creating a new `vm.Script` object compiles `code` but does not run it. The compiled `vm.Script` can be run later multiple times. It is important to note that the `code` is not bound to any global object; rather, it is bound before each run, just for that run.

`script.runInContext(contextifiedSandbox[, options])`

#

Added in: v0.3.1

- `contextifiedSandbox <Object>` A `contextified` object as returned by the `vm.createContext()` method.
- `options <Object>`
 - `filename <string>` Specifies the filename used in stack traces produced by this script.
 - `lineOffset <number>` Specifies the line number offset that is displayed in stack traces produced by this script.
 - `columnOffset <number>` Specifies the column number offset that is displayed in stack traces produced by this script.
 - `displayErrors <boolean>` When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
 - `timeout <number>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.
 - `breakOnSigint` : if `true`, the execution will be terminated when `SIGINT` (`Ctrl+C`) is received. Existing handlers for the event that have been attached via `process.on("SIGINT")` will be disabled during script execution, but will continue to work after that. If execution is terminated, an `Error` will be thrown.

Runs the compiled code contained by the `vm.Script` object within the given `contextifiedSandbox` and returns the result. Running code does not have access to local scope.

The following example compiles code that increments a global variable, sets the value of another global variable, then execute the code multiple times. The globals are contained in the `sandbox` object.

```
const util = require('util');
const vm = require('vm');

const sandbox = {
  animal: 'cat',
  count: 2
};

const script = new vm.Script('count += 1; name = "kitty";');

const context = new vm.createContext(sandbox);
for (var i = 0; i < 10; ++i) {
  script.runInContext(context);
}

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

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

script.runInNewContext([sandbox][, options])

Added in: v0.3.1

- `sandbox` `<Object>` An object that will be `contextified`. If `undefined`, a new object will be created.
- `options` `<Object>`
 - `filename` `<string>` Specifies the filename used in stack traces produced by this script.
 - `lineOffset` `<number>` Specifies the line number offset that is displayed in stack traces produced by this script.

- `columnOffset <number>` Specifies the column number offset that is displayed in stack traces produced by this script.
- `displayErrors <boolean>` When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
- `timeout <number>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.

First contextifies the given `sandbox`, runs the compiled code contained by the `vm.Script` object within the created sandbox, and returns the result. Running code does not have access to local scope.

The following example compiles code that sets a global variable, then executes the code multiple times in different contexts. The globals are set on and contained within each individual `sandbox`.

```
const util = require('util');
const vm = require('vm');

const script = new vm.Script('globalVar = "set"');

const sandboxes = [ {}, {}, {} ];
sandboxes.forEach((sandbox) => {
  script.runInNewContext(sandbox);
});

console.log(util.inspect(sandboxes));

// [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'set' }]
```

script.runInThisContext(options)

#

Added in: v0.3.1

- `options <Object>`
 - `filename <string>` Specifies the filename used in stack traces produced by this script.
 - `lineOffset <number>` Specifies the line number offset that is displayed in stack traces produced by this script.
 - `columnOffset <number>` Specifies the column number offset that is displayed in stack traces produced by this script.
 - `displayErrors <boolean>` When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
 - `timeout <number>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.

Runs the compiled code contained by the `vm.Script` within the context of the current `global` object. Running code does not have access to local scope, but *does* have access to the current `global` object.

The following example compiles code that increments a `global` variable then executes that code multiple times:

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

global.globalVar = 0;

const script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' })

for (var i = 0; i < 1000; ++i) {
  script.runInThisContext();
}

console.log(globalVar);

// 1000
```

vm.createContext([sandbox])

#

Added in: v0.3.1

- `sandbox <Object>`

If given a `sandbox` object, the `vm.createContext()` method will **prepare that sandbox** so that it can be used in calls to `vm.runInContext()` or `script.runInContext()`. Inside such scripts, the `sandbox` object will be the global object, retaining all of its existing properties but also having the built-in objects and functions any standard **global object** has. Outside of scripts run by the `vm` module, `sandbox` will remain unchanged.

If `sandbox` is omitted (or passed explicitly as `undefined`), a new, empty **contextified** `sandbox` object will be returned.

The `vm.createContext()` method is primarily useful for creating a single `sandbox` that can be used to run multiple scripts. For instance, if emulating a web browser, the method can be used to create a single `sandbox` representing a window's global object, then run all `<script>` tags together within the context of that `sandbox`.

vm.isContext(sandbox)

#

Added in: v0.11.7

- `sandbox <Object>`

Returns `true` if the given `sandbox` object has been **contextified** using `vm.createContext()`.

vm.runInContext(code, contextifiedSandbox[, options])

#

- `code <string>` The JavaScript code to compile and run.
- `contextifiedSandbox <Object>` The **contextified** object that will be used as the `global` when the `code` is compiled and run.
- `options`
 - `filename <string>` Specifies the filename used in stack traces produced

by this script.

- `lineOffset <number>` Specifies the line number offset that is displayed in stack traces produced by this script.
- `columnOffset <number>` Specifies the column number offset that is displayed in stack traces produced by this script.
- `displayErrors <boolean>` When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
- `timeout <number>` Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.

The `vm.runInContext()` method compiles `code`, runs it within the context of the `contextifiedSandbox`, then returns the result. Running code does not have access to the local scope. The `contextifiedSandbox` object *must* have been previously `contextified` using the `vm.createContext()` method.

The following example compiles and executes different scripts using a single `contextified` object:

```
const util = require('util');
const vm = require('vm');

const sandbox = { globalVar: 1 };
vm.createContext(sandbox);

for (var i = 0; i < 10; ++i) {
  vm.runInContext('globalVar *= 2;', sandbox);
}
console.log(util.inspect(sandbox));

// { globalVar: 1024 }
```

vm.runInDebugContext(code)

#

Added in: v0.11.14

- `code` <string> The JavaScript code to compile and run.

The `vm.runInDebugContext()` method compiles and executes `code` inside the V8 debug context. The primary use case is to gain access to the V8 `Debug` object:

```
const vm = require('vm');
const Debug = vm.runInDebugContext('Debug');
console.log(Debug.findScript(process.emit).name); // 'events.js'
console.log(Debug.findScript(process.exit).name); // 'internal/process.'
```

Note: The debug context and object are intrinsically tied to V8's debugger implementation and may change (or even be removed) without prior warning.

The `Debug` object can also be made available using the V8-specific `--expose_debug_as=` [command line option][cli.md].

vm.runInNewContext(code[, sandbox][, options])

#

Added in: v0.3.1

- `code` <string> The JavaScript code to compile and run.
- `sandbox` <Object> An object that will be `contextified`. If `undefined`, a new object will be created.
- `options`
 - `filename` <string> Specifies the filename used in stack traces produced by this script.
 - `lineOffset` <number> Specifies the line number offset that is displayed in stack traces produced by this script.
 - `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
 - `displayErrors` <boolean> When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the

stack trace.

- `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.

The `vm.runInContext()` first contextifies the given `sandbox` object (or creates a new `sandbox` if passed as `undefined`), compiles the `code`, runs it within the context of the created context, then returns the result. Running code does not have access to the local scope.

The following example compiles and executes code that increments a global variable and sets a new one. These globals are contained in the `sandbox`.

```
const util = require('util');
const vm = require('vm');

const sandbox = {
  animal: 'cat',
  count: 2
};

vm.runInNewContext('count += 1; name = "kitty"', sandbox);
console.log(util.inspect(sandbox));

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

vm.runInThisContext(code[, options])

#

Added in: v0.3.1

- `code` <string> The JavaScript code to compile and run.
- `options`
 - `filename` <string> Specifies the filename used in stack traces produced by this script.
 - `lineOffset` <number> Specifies the line number offset that is displayed in

stack traces produced by this script.

- `columnOffset` <number> Specifies the column number offset that is displayed in stack traces produced by this script.
- `displayErrors` <boolean> When `true`, if an `Error` error occurs while compiling the `code`, the line of code causing the error is attached to the stack trace.
- `timeout` <number> Specifies the number of milliseconds to execute `code` before terminating execution. If execution is terminated, an `Error` will be thrown.

`vm.runInThisContext()` compiles `code`, runs it within the context of the current `global` and returns the result. Running code does not have access to local scope, but does have access to the current `global` object.

The following example illustrates using both `vm.runInThisContext()` and the JavaScript `eval()` function to run the same code:

```
const vm = require('vm');
var localVar = 'initial value';

const vmResult = vm.runInThisContext('localVar = "vm";');
console.log('vmResult: ', vmResult);
console.log('localVar: ', localVar);

const evalResult = eval('localVar = "eval";');
console.log('evalResult: ', evalResult);
console.log('localVar: ', localVar);

// vmResult: 'vm', localVar: 'initial value'
// evalResult: 'eval', localVar: 'eval'
```

Because `vm.runInThisContext()` does not have access to the local scope, `localVar` is unchanged. In contrast, `eval()` does have access to the local scope, so the value `localVar` is changed. In this way `vm.runInThisContext()` is much like

an indirect eval() call, e.g. (0,eval)('code') .

Example: Running an HTTP Server within a VM

When using either `script.runInThisContext()` or `vm.runInThisContext()`, the code is executed within the current V8 global context. The code passed to this VM context will have its own isolated scope.

In order to run a simple web server using the `http` module the code passed to the context must either call `require('http')` on its own, or have a reference to the `http` module passed to it. For instance:

```
'use strict';
const vm = require('vm');

let code =
`(function(require) {

    const http = require('http');

    http.createServer( (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/');
})`;

vm.runInThisContext(code)(require);
```

Note: The `require()` in the above case shares the state with context it is passed from. This may introduce risks when untrusted code is executed, e.g. altering objects from the calling thread's context in unwanted ways.

What does it mean to "contextify" an object?

All JavaScript executed within Node.js runs within the scope of a "context".

According to the [V8 Embedder's Guide](#):

In V8, a context is an execution environment that allows separate, unrelated, JavaScript applications to run in a single instance of V8. You must explicitly specify the context in which you want any JavaScript code to be run.

When the method `vm.createContext()` is called, the `sandbox` object that is passed in (or a newly created object if `sandbox` is `undefined`) is associated internally with a new instance of a V8 Context. This V8 Context provides the `code` run using the `vm` modules methods with an isolated global environment within which it can operate. The process of creating the V8 Context and associating it with the `sandbox` object is what this document refers to as "contextifying" the `sandbox`.

Zlib

Stability: 2 - Stable

The `zlib` module provides compression functionality implemented using Gzip and Deflate/Inflate. It can be accessed using:

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

Compressing or decompressing a stream (such as a file) can be accomplished by piping the source stream data through a `zlib` stream into a destination stream:

```
const gzip = zlib.createGzip();
const fs = require('fs');
```

```
const inp = fs.createReadStream('input.txt');
const out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

It is also possible to compress or decompress data in a single step:

```
const input = '.....';
zlib.deflate(input, (err, buffer) => {
  if (!err) {
    console.log(buffer.toString('base64'));
  } else {
    // handle error
  }
});

const buffer = Buffer.from('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, (err, buffer) => {
  if (!err) {
    console.log(buffer.toString());
  } else {
    // handle error
  }
});
```

Compressing HTTP requests and responses

The `zlib` module can be used to implement support for the `gzip` and `deflate` content-encoding mechanisms defined by [HTTP](#).

The HTTP [Accept-Encoding](#) header is used within an http request to identify the compression encodings accepted by the client. The [Content-Encoding](#) header is used to identify the compression encodings actually applied to a message.

Note: the examples given below are drastically simplified to show the basic concept. Using `zlib` encoding can be expensive, and the results ought to be cached. See [Memory Usage Tuning](#) for more information on the speed/memory/compression tradeoffs involved in `zlib` usage.

```
// client request example
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
const request = http.get({ host: 'example.com',
                           path: '/',
                           port: 80,
                           headers: { 'Accept-Encoding': 'gzip,deflate' } });
request.on('response', (response) => {
  var output = fs.createWriteStream('example.com_index.html');

  switch (response.headers['content-encoding']) {
    // or, just use zlib.createUnzip() to handle both cases
    case 'gzip':
      response.pipe(zlib.createGunzip()).pipe(output);
      break;
    case 'deflate':
      response.pipe(zlib.createInflate()).pipe(output);
      break;
    default:
      response.pipe(output);
      break;
  }
});

// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
```

```

const zlib = require('zlib');
const http = require('http');
const fs = require('fs');

http.createServer((request, response) => {
  var raw = fs.createReadStream('index.html');

  var acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  // Note: this is not a conformant accept-encoding parser.
  // See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
  if (acceptEncoding.match(/\bdeflate\b/)) {
    response.writeHead(200, { 'Content-Encoding': 'deflate' });
    raw.pipe(zlib.createDeflate()).pipe(response);
  } else if (acceptEncoding.match(/\bgzip\b/)) {
    response.writeHead(200, { 'Content-Encoding': 'gzip' });
    raw.pipe(zlib.createGzip()).pipe(response);
  } else {
    response.writeHead(200, {});
    raw.pipe(response);
  }
}).listen(1337);

```

By default, the `zlib` methods with throw an error when decompressing truncated data. However, if it is known that the data is incomplete, or the desire is to inspect only the beginning of a compressed file, it is possible to suppress the default error handling by changing the flushing method that is used to compressed the last chunk of input data:

```

// This is a truncated version of the buffer from the above examples
const buffer = Buffer.from('eJzT0yMA', 'base64');

```

```
zlib.unzip(buffer, { finishFlush: zlib.Z_SYNC_FLUSH }, (err, buffer) =>
  if (!err) {
    console.log(buffer.toString());
  } else {
    // handle error
  }
});
```

This will not change the behavior in other error-throwing situations, e.g. when the input data has an invalid format. Using this method, it will not be possible to determine whether the input ended prematurely or lacks the integrity checks, making it necessary to manually check that the decompressed result is valid.

Memory Usage Tuning

From `zlib/zconf.h`, modified to node.js's usage:

The memory requirements for deflate are (in bytes):

```
(1 << (windowBits+2)) + (1 << (memLevel+9))
```

That is: 128K for windowBits=15 + 128K for memLevel = 8 (default values) plus a few kilobytes for small objects.

For example, to reduce the default memory requirements from 256K to 128K, the options should be set to:

```
{ windowBits: 14, memLevel: 7 }
```

This will, however, generally degrade compression.

The memory requirements for inflate are (in bytes)

```
1 << windowBits
```

That is, 32K for `windowBits=15` (default value) plus a few kilobytes for small objects.

This is in addition to a single internal output slab buffer of size `chunkSize`, which defaults to 16K.

The speed of `zlib` compression is affected most dramatically by the `level` setting. A higher level will result in better compression, but will take longer to complete. A lower level will result in less compression, but will be much faster.

In general, greater memory usage options will mean that Node.js has to make fewer calls to `zlib` because it will be able to process more data on each `write` operation. So, this is another factor that affects the speed, at the cost of memory usage.

Flushing

#

Calling `.flush()` on a compression stream will make `zlib` return as much output as currently possible. This may come at the cost of degraded compression quality, but can be useful when data needs to be available as soon as possible.

In the following example, `flush()` is used to write a compressed partial HTTP response to the client:

```
const zlib = require('zlib');
const http = require('http');

http.createServer((request, response) => {
  // For the sake of simplicity, the Accept-Encoding checks are omitted.
  response.writeHead(200, { 'content-encoding': 'gzip' });
  const output = zlib.createGzip();
  output.pipe(response);

  setInterval(() => {
```

```
    output.write(`The current time is ${Date()}\n`, () => {
        // The data has been passed to zlib, but the compression algorithm
        // have decided to buffer the data for more efficient compression.
        // Calling .flush() will make the data available as soon as the cl
        // is ready to receive it.
        output.flush();
    });
}, 1000);
}).listen(1337);
```

Constants

Added in: v0.5.8

All of the constants defined in `zlib.h` are also defined on `require('zlib')`. In the normal course of operations, it will not be necessary to use these constants. They are documented so that their presence is not surprising. This section is taken almost directly from the [zlib documentation](#). See <http://zlib.net/manual.html#Constants> for more details.

Allowed flush values.

- `zlib.Z_NO_FLUSH`
- `zlib.Z_PARTIAL_FLUSH`
- `zlib.Z_SYNC_FLUSH`
- `zlib.Z_FULL_FLUSH`
- `zlib.Z_FINISH`
- `zlib.Z_BLOCK`
- `zlib.Z_TREES`

Return codes for the compression/decompression functions. Negative values are errors, positive values are used for special but normal events.

- `zlib.Z_OK`

- `zlib.Z_STREAM_END`
- `zlib.Z_NEED_DICT`
- `zlib.Z_ERRNO`
- `zlib.Z_STREAM_ERROR`
- `zlib.Z_DATA_ERROR`
- `zlib.Z_MEM_ERROR`
- `zlib.Z_BUF_ERROR`
- `zlib.Z_VERSION_ERROR`

Compression levels.

- `zlib.Z_NO_COMPRESSION`
- `zlib.Z_BEST_SPEED`
- `zlib.Z_BEST_COMPRESSION`
- `zlib.Z_DEFAULT_COMPRESSION`

Compression strategy.

- `zlib.Z_FILTERED`
- `zlib.Z_HUFFMAN_ONLY`
- `zlib.Z_RLE`
- `zlib.Z_FIXED`
- `zlib.Z_DEFAULT_STRATEGY`

Possible values of the `data_type` field.

- `zlib.Z_BINARY`
- `zlib.Z_TEXT`
- `zlib.Z_ASCII`
- `zlib.Z_UNKNOWN`

The deflate compression method (the only one supported in this version).

- `zlib.Z_DEFLATED`

For initializing `zalloc`, `zfree`, `opaque`.

- `zlib.Z_NULL`

Class Options

Added in: v0.11.1

Each class takes an `options` object. All options are optional.

Note that some options are only relevant when compressing, and are ignored by the decompression classes.

- `flush` (default: `zlib.Z_NO_FLUSH`)
- `finishFlush` (default: `zlib.Z_FINISH`)
- `chunkSize` (default: `16*1024`)
- `windowBits`
- `level` (compression only)
- `memLevel` (compression only)
- `strategy` (compression only)
- `dictionary` (deflate/inflate only, empty dictionary by default)

See the description of `deflateInit2` and `inflateInit2` at
<http://zlib.net/manual.html#Advanced> for more information on these.

Class: zlib.Deflate

Added in: v0.5.8

Compress data using deflate.

Class: zlib.DeflateRaw

Added in: v0.5.8

Compress data using deflate, and do not append a `zlib` header.

Class: zlib.Gunzip

Added in: v0.5.8

Decompress a gzip stream.

Class: zlib.Gzip

Added in: v0.5.8

Compress data using gzip.

Class: zlib.Inflate

Added in: v0.5.8

Decompress a deflate stream.

Class: zlib.InflateRaw

Added in: v0.5.8

Decompress a raw deflate stream.

Class: zlib.Unzip

Added in: v0.5.8

Decompress either a Gzip- or Deflate-compressed stream by auto-detecting the header.

Class: zlib.Zlib

Added in: v0.5.8

Not exported by the `zlib` module. It is documented here because it is the base class of the compressor/decompressor classes.

`zlib.flush([kind], callback)`

Added in: v0.5.8

`kind` defaults to `zlib.Z_FULL_FLUSH`.

Flush pending data. Don't call this frivolously, premature flushes negatively impact the effectiveness of the compression algorithm.

Calling this only flushes data from the internal `zlib` state, and does not perform flushing of any kind on the streams level. Rather, it behaves like a normal call to `.write()`, i.e. it will be queued up behind other pending writes and will only produce output when data is being read from the stream.

`zlib.params(level, strategy, callback)`

Added in: v0.11.4

Dynamically update the compression level and compression strategy. Only applicable to deflate algorithm.

`zlib.reset()`

Added in: v0.7.0

Reset the compressor/decompressor to factory defaults. Only applicable to the inflate and deflate algorithms.

`zlib.createDeflate(options)`

Added in: v0.5.8

Returns a new `Deflate` object with an `options`.

`zlib.createDeflateRaw(options)`

Added in: v0.5.8

Returns a new `DeflateRaw` object with an `options`.

`zlib.createGunzip(options)`

Added in: v0.5.8

Returns a new `Gunzip` object with an `options`.

`zlib.createGzip(options)`

#

Added in: v0.5.8

Returns a new `Gzip` object with an `options`.

`zlib.createInflate(options)`

#

Added in: v0.5.8

Returns a new `Inflate` object with an `options`.

`zlib.createInflateRaw(options)`

#

Added in: v0.5.8

Returns a new `InflateRaw` object with an `options`.

`zlib.createUnzip(options)`

#

Added in: v0.5.8

Returns a new `Unzip` object with an `options`.

Convenience Methods

#

All of these take a `Buffer` or string as the first argument, an optional second argument to supply options to the `zlib` classes and will call the supplied callback with `callback(error, result)`.

Every method has a `*Sync` counterpart, which accept the same arguments, but without a callback.

`zlib.deflate(buf[, options], callback)`

#

Added in: v0.6.0

`zlib.deflateSync(buf[, options])`

#

Added in: v0.11.12

Compress a Buffer or string with Deflate.

zlib.deflateRaw(buf[, options], callback)

#

Added in: v0.6.0

zlib.deflateRawSync(buf[, options])

#

Added in: v0.11.12

Compress a Buffer or string with DeflateRaw.

zlib.gunzip(buf[, options], callback)

#

Added in: v0.6.0

zlib.gunzipSync(buf[, options])

#

Added in: v0.11.12

Decompress a Buffer or string with Gunzip.

zlib.gzip(buf[, options], callback)

#

Added in: v0.6.0

zlib.gzipSync(buf[, options])

#

Added in: v0.11.12

Compress a Buffer or string with Gzip.

zlib.inflate(buf[, options], callback)

#

Added in: v0.6.0

zlib.inflateSync(buf[, options])

#

Added in: v0.11.12

Decompress a Buffer or string with Inflate.

zlib.inflateRaw(buf[, options], callback)

#

Added in: v0.6.0

zlib.inflateRawSync(buf[, options])

#

Added in: v0.11.12

Decompress a Buffer or string with InflateRaw.

zlib.unzip(buf[, options], callback)

#

Added in: v0.6.0

zlib.unzipSync(buf[, options])

#

Added in: v0.11.12

Decompress a Buffer or string with Unzip.