Utility Modules

There are several utility modules available in Node.js module library. These modules are very common and are frequently used while developing any Node based application.

|  |  |
| --- | --- |
| Sr.No. | Module Name & Description |
| 1 | OS Module Provides basic operating-system related utility functions. |
| 2 | Path Module Provides utilities for handling and transforming file paths. |
| 3 | Net Module Provides both servers and clients as streams. Acts as a network wrapper. |
| 4 | DNS Module Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities. |
| 5 | Domain Module Provides ways to handle multiple different I/O operations as a single group. |

OS Module

Node.js os module provides a few basic operating-system related utility functions.

This module can be imported using the following syntax.

var os = require("os")

# Methods

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | os.tmpdir() Returns the operating system's default directory for temp files. |
| 2 | os.endianness()  Returns the endianness of the CPU. Possible values are "BE" or "LE". |
| 3 | os.hostname()  Returns the hostname of the operating system. |
| 4 | os.type()  Returns the operating system name. |
| 5 | os.platform()  Returns the operating system platform. |
| 6 | os.arch()  Returns the operating system CPU architecture. Possible values are "x64", "arm" and "ia32". |
| 7 | os.release()  Returns the operating system release. |
| 8 | os.uptime()  Returns the system uptime in seconds. |
| 9 | os.loadavg()  Returns an array containing the 1, 5, and 15 minute load averages. |
| 10 | os.totalmem()  Returns the total amount of system memory in bytes. |
| 11 | os.freemem()  Returns the amount of free system memory in bytes. |
| 12 | os.cpus()  Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the CPU/core spent in: user, nice, sys, idle, and irq). |
| 13 | os.networkInterfaces()  Get a list of network interfaces. |

# Properties

|  |  |
| --- | --- |
| Sr.No. | Property & Description |
| 1 | os.EOL  A constant defining the appropriate End-of-line marker for the operating system. |

**Example**

The following example demonstrates a few OS methods. Create a js file named main.js with the following code.

var os = require("os");

// Endianness  
console.log('endianness : ' + os.endianness());

// OS type  
console.log('type : ' + os.type());

// OS platform  
console.log('platform : ' + os.platform());

// Total system memory  
console.log('total memory : ' + os.totalmem() + " bytes.");

// Total free memory  
console.log('free memory : ' + os.freemem() + " bytes.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

endianness : LE  
type : Linux  
platform : linux  
total memory : 25103400960 bytes.  
free memory : 20676710400 bytes.

Path Module

Node.js path module is used for handling and transforming file paths.

This module can be imported using the following syntax.

var path = require("path")

# Methods

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | path.normalize(p)  Normalize a string path, taking care of '..' and '.' parts. |
| 2 | path.join([path1][, path2][, ...])  Join all the arguments together and normalize the resulting path. |
| 3 | path.resolve([from ...], to)  Resolves to an absolute path. |
| 4 | path.isAbsolute(path)  Determines whether the path is an absolute path. An absolute path will always resolve to the same location, regardless of the working directory. |
| 5 | path.relative(from, to)  Solve the relative path from from to to. |
| 6 | path.dirname(p)  Return the directory name of a path. Similar to the Unix dirname command. |
| 7 | path.basename(p[, ext])  Return the last portion of a path. Similar to the Unix basename command. |
| 8 | path.extname(p)  Return the extension of the path, from the last '.' to end of string in the last portion of the path. If there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string. |
| 9 | path.parse(pathString)  Returns an object from a path string. |
| 10 | path.format(pathObject)  Returns a path string from an object, the opposite of path.parse above. |

# Properties

|  |  |
| --- | --- |
| Sr.No. | Property & Description |
| 1 | path.sep  The platform-specific file separator. '\\' or '/'. |
| 2 | path.delimiter  The platform-specific path delimiter, ; or ':'. |
| 3 | path.posix  Provide access to aforementioned path methods but always interact in a posix compatible way. |
| 4 | path.win32  Provide access to aforementioned path methods but always interact in a win32 compatible way. |

**Example**

Create a js file named main.js with the following code −

var path = require("path");

// Normalization  
console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

// Join  
console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..'));

// Resolve  
console.log('resolve : ' + path.resolve('main.js'));

// extName  
console.log('ext name : ' + path.extname('main.js'));

Now run the main.js to see the result −

$ node main.js

Verify the Output.

normalization : /test/test1/2slashes/1slash

joint path : /test/test1/2slashes/1slash

resolve : /web/com/1427176256\_27423/main.js

ext name : .js

Net Module

Node.js net module is used to create both servers and clients.

This module provides an asynchronous network wrapper and it can be imported using the following syntax.

var net = require("net")

# Methods

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | **net.createServer([options][, connectionListener])**  Creates a new TCP server.  The connectionListener argument is automatically set as a listener for the 'connection' event. |
| 2 | **net.connect(options[, connectionListener])**  A factory method,  which returns a new 'net.Socket' and connects to the supplied address and port. |
| 3 | **net.createConnection(options[, connectionListener])**  A factory method,  which returns a new 'net.Socket' and connects to the supplied address and port. |
| 4 | **net.connect(port[, host][, connectListener])**  Creates a TCP connection to port on host.  If host is omitted, 'localhost' will be assumed.  The connectListener parameter will be added as a listener for the 'connect' event.  It is a factory method which returns a new 'net.Socket'. |
| 5 | **net.createConnection(port[, host][, connectListener])**  Creates a TCP connection to port on host.  If host is omitted, 'localhost' will be assumed.  The connectListener parameter will be added as a listener for the 'connect' event.  It is a factory method which returns a new 'net.Socket'. |
| 6 | **net.connect(path[, connectListener])**  Creates Unix socket connection to path.  The connectListener parameter will be added as a listener for the 'connect' event.  It is a factory method which returns a new 'net.Socket'. |
| 7 | **net.createConnection(path[, connectListener])**  Creates Unix socket connection to path.  The connectListener parameter will be added as a listener for the 'connect' event.  It is a factory method which returns a new 'net.Socket'. |
| 8 | **net.isIP(input)**  Tests if the input is an IP address.  Returns 0 for invalid strings,  4 for IP version 4 addresses,  and 6 for IP version 6 addresses. |
| 9 | **net.isIPv4(input)**  Returns true if the input is a version 4 IP address,  otherwise returns false. |
| 10 | **net.isIPv6(input)**  Returns true if the input is a version 6 IP address,  otherwise returns false. |

# Class - net.Server

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

## Methods

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | **server.listen(port[, host][, backlog][, callback])**  Begin accepting connections on the specified port and host.  If the host is omitted,  the server will accept connections directed to any IPv4 address (INADDR\_ANY).  A port value of zero will assign a random port. |
| 2 | **server.listen(path[, callback])**  Start a local socket server listening for connections on the given path. |
| 3 | **server.listen(handle[, callback])**  The handle object can be set to either a server  or socket (anything with an underlying \_handle member),  or a {fd: <n>} object.  It 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. |
| 4 | **server.listen(options[, callback])**  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, [host], [backlog], [callback]) . Alternatively, the path option can be used to specify a UNIX socket. |
| 5 | **server.close([callback])**  Finally closed when all connections are ended and the server emits a 'close' event. |
| 6 | **server.address()**  Returns the bound address,  the address family name  and port of the server as reported by the operating system. |
| 7 | **server.unref()**  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 unrefd, then calling unref again will have no effect. |
| 8 | **server.ref()**  Opposite of unref, calling ref on a previously unrefd server will not let the program exit  if it's the only server left (the default behavior).  If the server is refd, then calling the ref again will have no effect. |
| 9 | **server.getConnections(callback)**  Asynchronously get the number of concurrent connections on the server.  Works when sockets were sent to forks (Entwicklungszweige).  Callback should take two arguments err and count. |

## Events

|  |  |
| --- | --- |
| Sr.No. | Events & Description |
| 1 | **listening**  Emitted when the server has been bound after calling server.listen. |
| 2 | **connection**  Emitted when a new connection is made.  Socket object, the connection object is available to event handler.  Socket is an instance of net.Socket. |
| 3 | **close**  Emitted when the server closes.  Note that if connections exist,  this event is not emitted until all the connections are ended. |
| 4 | **error**  Emitted when an error occurs.  The 'close' event will be called directly following this event. |

# Class - net.Socket

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 and passed to the user through the 'connection' event of a server.

## Events

net.Socket is an eventEmitter and it emits the following events.

|  |  |
| --- | --- |
| Sr.No. | Events & Description |
| 1 | **Lookup**  Emitted after resolving the hostname but before connecting.  Not applicable to UNIX sockets. |
| 2 | **Connect**  Emitted when a socket connection is successfully established. |
| 3 | **Data**  Emitted when data is received.  The argument data will be a Buffer or String.  Encoding of data is set by socket.setEncoding(). |
| 4 | **End**  Emitted when the other end of the socket sends a FIN packet. |
| 5 | **Timeout**  Emitted if the socket times out from inactivity.  This is only to notify that the socket has been idle.  The user must manually close the connection. |
| 6 | **Drain**  Emitted when the write buffer becomes empty.  Can be used to throttle uploads. |
| 7 | **Error**  Emitted when an error occurs.  The 'close' event will be called directly following this event. |
| 8 | **Close**  Emitted once the socket is fully closed.  The argument had\_error is a boolean which indicates  if the socket was closed due to a transmission error. |

## Properties

net.Socket provides many useful properties to get better control over socket interactions.

|  |  |
| --- | --- |
| Sr.No. | Property & Description |
| 1 | **socket.bufferSize**  This property shows the number of characters currently buffered to be written. |
| 2 | **socket.remoteAddress**  The string representation of the remote IP address. For example, '74.125.127.100' or '2001:4860:a005::68'. |
| 3 | **socket.remoteFamily**  The string representation of the remote IP family. 'IPv4' or 'IPv6'. |
| 4 | **socket.remotePort**  The numeric representation of the remote port. For example, 80 or 21. |
| 5 | **socket.localAddress**  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'. |
| 6 | **socket.localPort**  The numeric representation of the local port. For example, 80 or 21. |
| 7 | **socket.bytesRead**  The amount of received bytes. |
| 8 | **socket.bytesWritten**  The amount of bytes sent. |

## Methods

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | **new net.Socket([options])**  Construct a new socket object. |
| 2 | **socket.connect(port[, host][, connectListener])**  Opens the connection for a given socket.  If port and host are given, then the socket will be opened as a TCP socket,  if host is omitted, localhost will be assumed.  If a path is given, the socket will be opened as a Unix socket to that path. |
| 3 | **socket.connect(path[, connectListener])**  Opens the connection for a given socket.  If port and host are given, then the socket will be opened as a TCP socket,  if host is omitted, localhost will be assumed.  If a path is given, the socket will be opened as a Unix socket to that path. |
| 4 | **socket.setEncoding([encoding])**  Set the encoding for the socket as a Readable Stream. |
| 5 | **socket.write(data[, encoding][, callback])**  Sends data on the socket. The second parameter specifies the encoding in the case of a string--it defaults to UTF8 encoding. |
| 6 | **socket.end([data][, encoding])**  Half-closes the socket,  i.e.,  it sends a FIN packet.  It is possible the server will still send some data. |
| 7 | **socket.destroy()**  Ensures that no more I/O activity happens on this socket.  Necessary only in case of errors (parse error or so). |
| 8 | **socket.pause()**  Pauses the reading of data.  That is, 'data' events will not be emitted. Useful to throttle back an upload. |
| 9 | **socket.resume()**  Resumes reading after a call to pause(). |
| 10 | **socket.setTimeout(timeout[, callback])**  Sets the socket to timeout after timeout milliseconds of inactivity on the socket.  By default, net.Socket does not have a timeout. |
| 11 | **socket.setNoDelay([noDelay])**  Disables the Nagle algorithm.  By default, TCP connections use the Nagle algorithm, they buffer data before sending it off.  Setting true for noDelay will immediately fire off data each time socket.write() is called.  noDelay defaults to true. |
| 12 | **socket.setKeepAlive([enable][, initialDelay])**  Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket.  enable defaults to false. |
| 13 | **socket.address()**  Returns the bound address, the address family name, and the 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' }. |
| 14 | **socket.unref()**  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 unrefd, then calling unref again will have no effect. |
| 15 | **socket.ref()**  Opposite of unref,  calling ref on a previously unrefd socket will not let the program exit if it's the only socket left (the default behavior).  If the socket is refd, then calling ref again will have no effect. |

**Example**

Create a js file named server.js with the following code − File: **server.js**

var net = require('net');

var server = net.createServer(function(connection) {   
 console.log('client connected');

connection.on('end', function() {  
 console.log('client disconnected');  
 });

connection.write('Hello World!\r\n');  
 connection.pipe(connection);  
});

server.listen(8080, function() {   
 console.log('server is listening');  
});

Now run the server.js to see the result −

$ node server.js

Verify the Output.

server is listening

Create a js file named client.js with the following code − File: **client.js**

var net = require('net');

var client = net.connect({port: 8080}, function() {  
 console.log('connected to server!');   
});

client.on('data', function(data) {  
 console.log(data.toString());  
 client.end();  
});

client.on('end', function() {   
 console.log('disconnected from server');  
});

Now run the client.js from another terminal to see the result −

$ node client.js

Verify the Output.

connected to server!  
Hello World!  
disconnected from server

Verify the Output on the terminal where server.js is running.

server is listening  
client connected  
client disconnected