All Built-in Modules: NodeJs

|  |  |
| --- | --- |
| **Module** | **Description** |
| assert | Provides a set of assertion tests |
| buffer | To handle binary data |
| child\_process | To run a child process |
| cluster | To split a single Node process into multiple processes |
| crypto | To handle OpenSSL cryptographic functions |
| dgram | Provides implementation of UDP datagram sockets |
| dns | To do DNS lookups and name resolution functions |
| domain | Deprecated. To handle unhandled errors |
| events | To handle events |
| fs | To handle the file system |
| http | To make Node.js act as an HTTP server |
| https | To make Node.js act as an HTTPS server. |
| net | To create servers and clients |
| os | Provides information about the operation system |
| path | To handle file paths |
| punycode | Deprecated. A character encoding scheme |
| querystring | To handle URL query strings |
| readline | To handle readable streams one line at the time |
| stream | To handle streaming data |
| string\_decoder | To decode buffer objects into strings |
| timers | To execute a function after a given number of milliseconds |
| tls | To implement TLS and SSL protocols |
| tty | Provides classes used by a text terminal |
| url | To parse URL strings |
| util | To access utility functions |
| v8 | To access information about V8 (the JavaScript engine) |
| vm | To compile JavaScript code in a virtual machine |
| zlib | To compress or decompress files |

**Node.js - Utility Modules**

|  |  |
| --- | --- |
| **Module Name** | **Description** |
| OS Module | Provides basic operating-system related utility functions. |
| Path Module | Provides utilities for handling and transforming file paths. |
| Net Module | Provides both servers and clients as streams. Acts as a network wrapper. |
| DNS Module | Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities. |
| Domain Module | Provides ways to handle multiple different I/O operations as a single group. |

**Node.js - OS Module:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| os.tmpdir() | Returns the operating system's default directory for temp files. |
| os.endianness() | Returns the endianness of the CPU. Possible values are "BE" or "LE". |
| os.hostname() | Returns the hostname of the operating system. |
| os.type() | Returns the operating system name. |
| os.platform() | Returns the operating system platform. |
| os.arch() | Returns the operating system CPU architecture. Possible values are "x64", "arm" and "ia32". |
| os.release() | Returns the operating system release. |
| os.uptime() | Returns the system uptime in seconds. |
| os.loadavg() | Returns an array containing the 1, 5, and 15 minute load averages. |
| os.totalmem() | Returns the total amount of system memory in bytes. |
| os.freemem() | Returns the amount of free system memory in bytes. |
| os.cpus() | Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the CPU/core spent in: user, nice, sys, idle, and irq). |
| os.networkInterfaces() | Get a list of network interfaces. |
| **os.EOL** | A constant defining the appropriate End-of-line marker for the operating system. |

**Node.js - Path Module**

|  |  |
| --- | --- |
| **Method / Property** | **Description** |
| path.normalize(p) | Normalize a string path, taking care of '..' and '.' parts. |
| path.join([path1][, path2][, ...]) | Join all the arguments together and normalize the resulting path. |
| path.resolve([from ...], to) | Resolves to an absolute path. |
| 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. |
| path.relative(from, to) | Solve the relative path from from to to. |
| path.dirname(p) | Return the directory name of a path. Similar to the Unix dirname command. |
| path.basename(p[, ext]) | Return the last portion of a path. Similar to the Unix basename command. |
| 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. |
| path.parse(pathString) | Returns an object from a path string. |
| path.format(pathObject) | Returns a path string from an object, the opposite of path.parse above. |
| path.sep | The platform-specific file separator. '\\' or '/'. |
| path.delimiter | The platform-specific path delimiter, ; or ':'. |
| path.posix | Provide access to aforementioned path methods but always interact in a posix compatible way. |
| path.win32 | Provide access to aforementioned path methods but always interact in a win32 compatible way. |

**Node.js - Net Module**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **net.createServer([options][, connectionListener])** | Creates a new TCP server. The connectionListener argument is automatically set as a listener for the 'connection' event. |
| **net.connect(options[, connectionListener])** | A factory method, which returns a new 'net.Socket' and connects to the supplied address and port. |
| **net.createConnection(options[, connectionListener])** | A factory method, which returns a new 'net.Socket' and connects to the supplied address and port. |
| **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'. |
| **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'. |
| **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'. |
| **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'. |
| **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. |
| **net.isIPv4(input)** | Returns true if the input is a version 4 IP address, otherwise returns false. |
| **net.isIPv6(input)** | Returns true if the input is a version 6 IP address, otherwise returns false. |

**Class - net.Server**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **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.** |
| **server.listen(path[, callback])** | **Start a local socket server listening for connections on the given path.** |
| **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.** |
| **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.** |
| **server.close([callback])** | **Finally closed when all connections are ended and the server emits a 'close' event.** |
| **server.address()** | **Returns the bound address, the address family name and port of the server as reported by the operating system.** |
| **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.** |
| **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.** |
| **server.getConnections(callback)** | **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.** |

**Class - net.Server**

|  |  |
| --- | --- |
| **Events** | **Description** |
| **listening** | **Emitted when the server has been bound after calling server.listen.** |
| **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.** |
| **close** | **Emitted when the server closes. Note that if connections exist, this event is not emitted until all the connections are ended.** |
| **error** | **Emitted when an error occurs. The 'close' event will be called directly following this event.** |

**Class - net.Socket**

|  |  |
| --- | --- |
| **Events** | **Description** |
| **lookup** | **Emitted after resolving the hostname but before connecting. Not applicable to UNIX sockets.** |
| **connect** | **Emitted when a socket connection is successfully established.** |
| **data** | **Emitted when data is received. The argument data will be a Buffer or String. Encoding of data is set by socket.setEncoding().** |
| **end** | **Emitted when the other end of the socket sends a FIN packet.** |
| **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.** |
| **drain** | **Emitted when the write buffer becomes empty. Can be used to throttle uploads.** |
| **error** | **Emitted when an error occurs. The 'close' event will be called directly following this event.** |
| **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.** |

|  |  |
| --- | --- |
| **Property** | **Description** |
| **socket.bufferSize** | **This property shows the number of characters currently buffered to be written.** |
| **socket.remoteAddress** | **The string representation of the remote IP address. For example, '74.125.127.100' or '2001:4860:a005::68'.** |
| **socket.remoteFamily** | **The string representation of the remote IP family. 'IPv4' or 'IPv6'.** |
| **socket.remotePort** | **The numeric representation of the remote port. For example, 80 or 21.** |
| **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'.** |
| **socket.localPort** | **The numeric representation of the local port. For example, 80 or 21.** |
| **socket.bytesRead** | **The amount of received bytes.** |
| **socket.bytesWritten** | **The amount of bytes sent.** |

|  |  |
| --- | --- |
| **Method** | **Description** |
| **new net.Socket([options])** | **Construct a new socket object.** |
| **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.** |
| **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.** |
| **socket.setEncoding([encoding])** | **Set the encoding for the socket as a Readable Stream.** |
| **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.** |
| **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.** |
| **socket.destroy()** | **Ensures that no more I/O activity happens on this socket. Imp only in case of errors (parse error or so).** |
| **socket.pause()** | **Pauses the reading of data. That is, 'data' events will not be emitted. Useful to throttle back an upload.** |
| **socket.resume()** | **Resumes reading after a call to pause().** |
| **socket.setTimeout(timeout[, callback])** | **Sets the socket to timeout after timeout milliseconds of inactivity on the socket. By default, net.Socket does not have a timeout.** |
| **socket.setNoDelay([noDelay])** | **Disables the Nagle algorithm. By default, TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting true for noDelay will immediately fire off data each time socket.write() is called. noDelay defaults to true.** |
| **socket.setKeepAlive([enable][, initialDelay])** | **Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket. enable defaults to false.** |
| **socket.address()** | **Returns the bound address, the address family name, and the port of 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.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.** |
| **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.** |

**Node.js - DNS Module**

**Node.js dns module is used to do actual DNS lookup as well as to use underlying operating system name resolution functionalities. This module provides an asynchronous network wrapper**

* **rrtypes**
* **Following is the list of valid rrtypes used by dns.resolve() method −**
  + **A − IPV4 addresses, default**
  + **AAAA − IPV6 addresses**
  + **MX − mail exchange records**
  + **TXT − text records**
  + **SRV − SRV records**
  + **PTR − used for reverse IP lookups**
  + **NS − name server records**
  + **CNAME − canonical name records**
  + **SOA − start of authority record**

**Node.js - DNS Module**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **dns.lookup(hostname[, options], callback)** | **Resolves a hostname (e.g. 'google.com') into the first found A (IPv4) or AAAA (IPv6) record. options can be an object or integer. If options is not provided, then IP v4 and v6 addresses are both valid. If options is an integer, then it must be 4 or 6.** |
| **dns.lookupService(address, port, callback)** | **Resolves the given address and port into a hostname and service using getnameinfo.** |
| **dns.resolve(hostname[, rrtype], callback)** | **Resolves a hostname (e.g. 'google.com') into an array of the record types specified by rrtype.** |
| **dns.resolve4(hostname, callback)** | **The same as dns.resolve(), but only for IPv4 queries (A records). addresses is an array of IPv4 addresses (e.g. ['74.125.79.104', '74.125.79.105', '74.125.79.106']).** |
| **dns.resolve6(hostname, callback)** | **The same as dns.resolve4() except for IPv6 queries (an AAAA query).** |
| **dns.resolveMx(hostname, callback)** | **The same as dns.resolve(), but only for mail exchange queries (MX records).** |
| **dns.resolveTxt(hostname, callback)** | **The same as dns.resolve(), but only for text queries (TXT records). addresses is an 2-d 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, they could be either joined together or treated separately.** |
| **dns.resolveSrv(hostname, callback)** | **The same as dns.resolve(), but only for service records (SRV records). addresses is an array of the SRV records available for hostname. Properties of SRV records are priority, weight, port, and name (e.g., [{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]).** |
| **dns.resolveSoa(hostname, callback)** | **The same as dns.resolve(), but only for start of authority record queries (SOA record).** |
| **dns.resolveNs(hostname, callback)** | **The same as dns.resolve(), but only for name server records (NS records). addresses is an array of the name server records available for hostname (e.g., ['ns1.example.com', 'ns2.example.com']).** |
| **dns.resolveCname(hostname, callback)** | **The same as dns.resolve(), but only for canonical name records (CNAME records). addresses is an array of the canonical name records available for hostname (e.g., ['bar.example.com']).** |
| **dns.reverse(ip, callback)** | **Reverse resolves an ip address to an array of hostnames.** |
| **dns.getServers()** | **Returns an array of IP addresses as strings that are currently being used for resolution.** |
| **dns.setServers(servers)** | **Given an array of IP addresses as strings, set them as the servers to use for resolving.** |

* **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.DESTRUCTION − Channel is being destroyed.**
  + **dns.BADSTR − Misformatted string.**
  + **dns.BADFLAGS − Illegal flags specified.**
  + **dns.NONAME − Given hostname is not numeric.**
  + **dns.BADHINTS − Illegal hints flags specified.**
  + **dns.NOTINITIALIZED − c-ares library initialization not yet performed.**
  + **dns.LOADIPHLPAPI − Error loading iphlpapi.dll.**
  + **dns.ADDRGETNETWORKPARAMS − Could not find GetNetworkParams function.**
  + **dns.CANCELLED − DNS query cancelled.**

**Node.js - Domain Module**

|  |  |
| --- | --- |
| **Method / Property** | **Description** |
| **domain.run(function)** | **Run the supplied function in the context of the domain, implicitly binding all event emitters, timers, and lowl evel requests that are created in that context. This is the most basic way to use a domain.** |
| **domain.add(emitter)** | **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.** |
| **domain.remove(emitter)** | **The opposite of domain.add(emitter). Removes domain handling from the specified emitter.** |
| **domain.bind(callback)** | **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.** |
| **domain.intercept(callback)** | **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.** |
| **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.** |
| **domain.exit()** | **The exit method exits the current domain, popping it off the domain stack. Whenever the execution switches 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.** |
| **domain.dispose()** | **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 emit** |
| **domain.members** | **An array of timers and event emitters that have been explicitly added to the domain.** |