Node.js v0.12.2 Manual & Documentation

Table of Contents

Stream

```
Stability: 2 - Unstable
```

A stream is an abstract interface implemented by various objects in Node. For example a **request to an HTTP server** is a stream, as is **stdout**. Streams are readable, writable, or both. All streams are instances of **EventEmitter**

You can load the Stream base classes by doing `require('stream')`. There are base classes provided for **Readable** streams, **Writable** streams, **Duplex** streams, and **Transform** streams.

This document is split up into 3 sections. The first explains the parts of the API that you need to be aware of to use streams in your programs. If you never implement a streaming API yourself, you can stop there.

The second section explains the parts of the API that you need to use if you implement your own custom streams yourself. The API is designed to make this easy for you to do.

The third section goes into more depth about how streams work, including some of the internal mechanisms and functions that you should probably not modify unless you definitely know what you are doing.

API for Stream Consumers

Streams can be either **Readable**, **Writable**, or both (**Duplex**).

All streams are EventEmitters, but they also have other custom methods and properties depending on whether they are Readable, Writable, or Duplex.

If a stream is both Readable and Writable, then it implements all of the methods and events below. So, a **Duplex** or **Transform** stream is fully described by this API, though their implementation may be somewhat different.

It is not necessary to implement Stream interfaces in order to consume streams in your programs. If you **are** implementing streaming interfaces in your own program, please also refer to **API for Stream Implementors** below.

Almost all Node programs, no matter how simple, use Streams in some way. Here is an example of using Streams in a Node program:

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

var server = http.createServer(function (req, res) {
  // req is an http.IncomingMessage, which is a Readable Stream
  // res is an http.ServerResponse, which is a Writable Stream

  var body = '';
  // we want to get the data as utf8 strings
  // If you don't set an encoding, then you'll get Buffer objects
  req.setEncoding('utf8');

  // Readable streams emit 'data' events once a listener is added
  req.on('data', function (chunk) {
    body += chunk;
  });

  // the end event tells you that you have entire body
  req.on('end', function () {
    try {
      var 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
```

Class: stream.Readable

The Readable stream interface is the abstraction for a *source* of data that you are reading from. In other words, data comes *out* of a Readable stream.

A Readable stream will not start emitting data until you indicate that you are ready to receive it.

Readable streams have two "modes": a **flowing mode** and a **paused mode**. When in flowing mode, data is read from the underlying system and provided to your program as fast as possible. In paused mode, you must explicitly call `stream.read()` to get chunks of data out. Streams start out in paused mode.

Note: If no data event handlers are attached, and there are no **`pipe()`** destinations, and the stream is switched into flowing mode, then data will be lost.

You can switch to flowing mode by doing any of the following:

- Adding a **`'data'` event** handler to listen for data.
- Calling the **`resume()`** method to explicitly open the flow.
- Calling the **`pipe()`** method to send the data to a **Writable**.

You can switch back to paused mode by doing either of the following:

- If there are no pipe destinations, by calling the **`pause()`** method.
- If there are pipe destinations, by removing any **`'data'` event** handlers, and removing all pipe destinations by calling the **`unpipe()`** method.

Note that, for backwards compatibility reasons, removing `'data'` event handlers will **not** automatically pause the stream. Also, if there are piped destinations, then calling `pause()` will not guarantee that the stream will *remain* paused once those destinations drain and ask for more data.

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**

Event: 'readable'#

When a chunk of data can be read from the stream, it will emit a `'readable'` event.

In some cases, listening for a `'readable'` event will cause some data to be read into the internal buffer from the underlying system, if it hadn't already.

```
var readable = getReadableStreamSomehow();
readable.on('readable', function() {
  // there is some data to read now
});
```

Once the internal buffer is drained, a `readable` event will fire again when more data is available.

Event: 'data'#

- `chunk` Buffer | String The chunk of data.

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.

If you just want to get all the data out of the stream as fast as possible, this is the best way to do so.

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
});
```

Event: 'end'#

This event fires when there will be no more data to read.

Note that the `end` event **will not fire** unless the data is completely consumed. This can be done by switching into flowing mode, or by calling `read()` repeatedly until you get to the end.

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
});
readable.on('end', function() {
  console.log('there will be no more data.');
});
```

Event: 'close'#

Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.

Event: 'error'#

- Error Object

Emitted if there was an error receiving data.

readable.read([size])#

- `size` Number Optional argument to specify how much data to read.
- Return String | Buffer | null

The `read()` method pulls some data out of the internal buffer and returns it. If there is no data available, then it will return `null`.

If you pass in a `size` argument, then it will return that many bytes. If `size` bytes are not available, then it will return `null`.

If you do not specify a `size` argument, then it will return all the data in the internal buffer.

This method should only be called in paused mode. In flowing mode, this method is called automatically until the internal buffer is drained.

```
var readable = getReadableStreamSomehow();
readable.on('readable', function() {
  var chunk;
  while (null !== (chunk = readable.read())) {
    console.log('got %d bytes of data', chunk.length);
  }
});
```

If this method returns a data chunk, then it will also trigger the emission of a **`'data'` event**.

readable.setEncoding(encoding)#

- `encoding` String The encoding to use.
- Return: `this`

Call this function to cause the stream to return strings of the specified encoding instead of Buffer objects. For example, if you do `readable.setEncoding('utf8')`, then the output data will be interpreted as UTF-8 data, and returned as strings. If you do `readable.setEncoding('hex')`, then the data will be encoded in hexadecimal string format.

This properly handles multi-byte characters that would otherwise be potentially mangled if you simply pulled the Buffers directly and called `buf.toString(encoding)` on them. If you want to read the data as strings, always use this method.

```
var readable = getReadableStreamSomehow();
readable.setEncoding('utf8');
readable.on('data', function(chunk) {
  assert.equal(typeof chunk, 'string');
  console.log('got %d characters of string data', chunk.length);
});
```

readable.resume()#

- Return: `this`

This method will cause the readable stream to resume emitting `data` events.

This method will switch the stream into flowing mode. If you do *not* want to consume the data from a stream, but you *do* want to get to its `end` event, you can call **`readable.resume()`** to open the flow of data.

```
var readable = getReadableStreamSomehow();
readable.resume();
readable.on('end', function(chunk) {
  console.log('got to the end, but did not read anything');
});
```

readable.pause()#

- Return: `this`

This 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.

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
  readable.pause();
  console.log('there will be no more data for 1 second');
  setTimeout(function() {
    console.log('now data will start flowing again');
    readable.resume();
  }, 1000);
});
```

readable.isPaused()#

- Return: `Boolean`

This method returns whether or not the `readable` has been **explicitly** paused by client code (using `readable.pause()` without a corresponding `readable.resume()`).

```
var readable = new stream.Readable

readable.isPaused() // === false
readable.pause()
readable.isPaused() // === true
readable.resume()
readable.isPaused() // === false
```

readable.pipe(destination[, options])#

- `destination` **Writable** Stream The destination for writing data
- `options` Object Pipe options
  - `end` Boolean End the writer when the reader ends. Default = `true`

This method pulls all the data out of a readable stream, and writes it to the supplied destination, automatically managing the flow so that the destination is not overwhelmed by a fast readable stream.

Multiple destinations can be piped to safely.

```
var readable = getReadableStreamSomehow();
var writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt'
readable.pipe(writable);
```

This function returns the destination stream, so you can set up pipe chains like so:

```
var r = fs.createReadStream('file.txt');
var z = zlib.createGzip();
var w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);
```

For example, emulating the Unix `cat` command:

```
process.stdin.pipe(process.stdout);
```

By default **end()** is called on the destination when the source stream emits `end`, so that `destination` is no longer writable. Pass `{ end: false }` as `options` to keep the destination stream open.

This keeps `writer` open so that "Goodbye" can be written at the end.

```
reader.pipe(writer, { end: false });
reader.on('end', function() {
  writer.end('Goodbye\n');
});
```

Note that `process.stderr` and `process.stdout` are never closed until the process exits, regardless of the specified options.

readable.unpipe([destination])#

- destination **Writable** Stream Optional specific stream to unpipe

This method will remove the hooks set up for a previous pipe() call.

If the destination is not specified, then all pipes are removed.

If the destination is specified, but no pipe is set up for it, then this is a no-op.

```
var readable = getReadableStreamSomehow();
var writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt',
// but only for the first second
readable.pipe(writable);
setTimeout(function() {
  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

This is useful in certain cases where a stream is being consumed by a parser, which needs to "un-consume" some data that it has optimistically pulled out of the source, so that the stream can be passed on to some other party.

If you find that you must often call stream.unshift(chunk) in your programs, consider implementing a **Transform** stream instead. (See API for Stream Implementors, below.)

```
// Pull off a header delimited by \n\n
// use unshift() if we get too much
// Call the callback with (error, header, stream)
var StringDecoder = require('string_decoder').StringDecoder;
function parseHeader(stream, callback) {
  stream.on('error', callback);
  stream.on('readable', onReadable);
  var 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();
        var remaining = split.join('\n\n');
        var buf = new Buffer(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;
      }
    }
  }
}
```

readable.wrap(stream)#

- `stream` Stream An "old style" readable stream

Versions of Node prior to v0.10 had streams that did not implement the entire Streams API as it is today. (See "Compatibility" below for more information.)

If you are using an older Node library that emits `'data'` events and has a **`pause()`** method that is advisory only, then you can use the `wrap()` method to create a **Readable** stream that uses the old stream as its data source.

You will very rarely ever need to call this function, but it exists as a convenience for interacting with old Node programs and libraries.

For example:

```
var OldReader = require('./old-api-module.js').OldReader;
var oreader = new OldReader;
var Readable = require('stream').Readable;
var myReader = new Readable().wrap(oreader);

myReader.on('readable', function() {
  myReader.read(); // etc.
});
```

Class: stream.Writable

The Writable stream interface is an abstraction for a *destination* that you are writing data *to*.

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**

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 True if the data was handled completely.

This method writes some data to the underlying system, and calls the supplied callback once the data has been fully handled.

The return value indicates if you should continue writing right now. If the data had to be buffered internally, then it will return `false`. Otherwise, it will return `true`.

This return value is strictly advisory. You MAY continue to write, even if it returns `false`. However, writes will be buffered in memory, so it is best not to do this excessively. Instead, wait for the `drain` event before writing more data.

Event: 'drain'#

If a **`writable.write(chunk)`** call returns false, then the `drain` event will indicate when it is appropriate to begin writing more data to the stream.

```
// Write the data to the supplied writable stream 1MM times.
// Be attentive to back-pressure.
```

```
function writeOneMillionTimes(writer, data, encoding, callback) {
  var i = 1000000;
  write();
  function write() {
    var ok = true;
    do {
      i -= 1;
      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 (i > 0 && ok);
    if (i > 0) {
      // had to stop early!
      // write some more once it drains
      writer.once('drain', write);
    }
  }
}
```

writable.cork()#

Forces buffering of all writes.

Buffered data will be flushed either at `.uncork()` or at `.end()` call.

writable.uncork()#

Flush all data, buffered since `.cork()` call.

writable.setDefaultEncoding(encoding)#

- `encoding` String The new default encoding
- Return: `Boolean`

Sets the default encoding for a writable stream. Returns `true` if the encoding is valid and is set. Otherwise returns `false`.

writable.end([chunk][, encoding][, callback])#

- `chunk` String | Buffer Optional data to write
- `encoding` String The encoding, if `chunk` is a String
- `callback` Function Optional callback for when the stream is finished

Call this method when no more data will be written to the stream. If supplied, the callback is attached as a listener on the `finish` event.

Calling **`write()`** after calling **`end()`** will raise an error.

```
// write 'hello, ' and then end with 'world!'
var file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// writing more now is not allowed!
```

Event: 'finish'#

When the **`end()`** method has been called, and all data has been flushed to the underlying system, this event is emitted.

```
var writer = getWritableStreamSomehow();
for (var i = 0; i < 100; i ++) {
  writer.write('hello, #' + i + '!\n');
}
writer.end('this is the end\n');
writer.on('finish', function() {
  console.error('all writes are now complete.');
});
```

Event: 'pipe'#

- `src` **Readable** Stream source stream that is piping to this writable

This is emitted whenever the `pipe()` method is called on a readable stream, adding this writable to its set of destinations.

```
var writer = getWritableStreamSomehow();
var reader = getReadableStreamSomehow();
writer.on('pipe', function(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

This is emitted whenever the **`unpipe()`** method is called on a readable stream, removing this writable from its set of destinations.

```
var writer = getWritableStreamSomehow();
var reader = getReadableStreamSomehow();
writer.on('unpipe', function(src) {
  console.error('something has stopped piping into the writer');
  assert.equal(src, reader);
});
reader.pipe(writer);
reader.unpipe(writer);
```

Event: 'error'#

- Error object

Emitted if there was an error when writing or piping data.

Class: stream.Duplex

Duplex streams are streams that implement both the **Readable** and **Writable** interfaces. See above for usage.

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 computed from the input. They implement both the **Readable** and **Writable** interfaces. See above for usage.

Examples of Transform streams include:

- **zlib streams**
- **crypto streams**

API for Stream Implementors

To implement any sort of stream, the pattern is the same:

1. Extend the appropriate parent class in your own subclass. (The `util.inherits` method is particularly helpful for this.)
2. Call the appropriate parent class constructor in your constructor, to be sure that the internal mechanisms are set up properly.
3. Implement one or more specific methods, as detailed below.

The class to extend and the method(s) to implement depend on the sort of stream class you are writing:

Use-case	Class	Method(s) to implement
Reading only	**Readable**	**_read**
Writing only	**Writable**	**_write**
Reading and writing	**Duplex**	**_read**, **_write**
Operate on written data, then read the result	**Transform**	`_transform`, `_flush`

In your implementation code, it is very important to never call the methods described in **API for Stream Consumers** above. Otherwise, you can potentially cause adverse side effects in programs that consume your streaming interfaces.

Class: stream.Readable

`stream.Readable` is an abstract class designed to be extended with an underlying implementation of the **`_read(size)`** method.

Please see above under **API for Stream Consumers** for how to consume streams in your programs. What follows is an explanation of how to implement Readable streams in your programs.

Example: A Counting Stream#

This is a basic example of a Readable stream. It emits the numerals from 1 to 1,000,000 in ascending order, and then ends.

```
var Readable = require('stream').Readable;
var util = require('util');
util.inherits(Counter, Readable);

function Counter(opt) {
  Readable.call(this, opt);
  this._max = 1000000;
  this._index = 1;
}

Counter.prototype._read = function() {
  var i = this._index++;
  if (i > this._max)
    this.push(null);
  else {
    var str = '' + i;
    var buf = new Buffer(str, 'ascii');
    this.push(buf);
  }
};
```

Example: SimpleProtocol v1 (Sub-optimal)#

This is similar to the `parseHeader` function described above, but implemented as a custom stream. Also, note that this implementation does not convert the incoming data to a string.

However, this would be better implemented as a **Transform** stream. See below for a better implementation.

```
// A parser for a simple data protocol.
// The "header" is a JSON object, followed by 2 \n characters, and
// then a message body.
//
// NOTE: This can be done more simply as a Transform stream!
// Using Readable directly for this is sub-optimal.  See the
// alternative example below under the Transform section.

var Readable = require('stream').Readable;
var util = require('util');

util.inherits(SimpleProtocol, Readable);

function SimpleProtocol(source, options) {
  if (!(this instanceof SimpleProtocol))
    return new SimpleProtocol(source, options);

  Readable.call(this, options);
  this._inBody = false;
  this._sawFirstCr = false;

  // source is a readable stream, such as a socket or file
  this._source = source;

  var self = this;
  source.on('end', function() {
    self.push(null);
  });

  // give it a kick whenever the source is readable
  // read(0) will not consume any bytes
  source.on('readable', function() {
    self.read(0);
  });

  this._rawHeader = [];
  this.header = null;
}

SimpleProtocol.prototype._read = function(n) {
  if (!this._inBody) {
    var chunk = this._source.read();

    // if the source doesn't have data, we don't have data yet.
    if (chunk === null)
      return this.push('');
    // check if the chunk has a \n\n
    var split = -1;
    for (var i = 0; i < chunk.length; i++) {
      if (chunk[i] === 10) { // '\n'
        if (this._sawFirstCr) {
          split = i;
          break;
        } else {
          this._sawFirstCr = true;
        }
      } else {
        this._sawFirstCr = false;
      }
    }

    if (split === -1) {
      // still waiting for the \n\n
      // stash the chunk, and try again.
      this._rawHeader.push(chunk);
      this.push('');
    } else {
      this._inBody = true;
      var h = chunk.slice(0, split);
      this._rawHeader.push(h);
      var header = Buffer.concat(this._rawHeader).toString();
      try {
        this.header = JSON.parse(header);
      } catch (er) {
        this.emit('error', new Error('invalid simple protocol data'));
        return;
      }
      // now, because we got some extra data, unshift the rest
      // back into the read queue so that our consumer will see it.
      var b = chunk.slice(split);
      this.unshift(b);

      // and let them know that we are done parsing the header.
      this.emit('header', this.header);
    }
  } else {
    // from there on, just provide the data to our consumer.
    // careful not to push(null), since that would indicate EOF.
    var chunk = this._source.read();
    if (chunk) this.push(chunk);
  }
};

// Usage:
// var parser = new SimpleProtocol(source);
// Now parser is a readable stream that will emit 'header'
// with the parsed header data.
```

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. Default=16kb, or 16 for `objectMode` streams
  - `encoding` String If specified, then buffers will be decoded to strings using the specified encoding. Default=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. Default=false

In classes that extend the Readable class, make sure to call the Readable constructor so that the buffering settings can be properly initialized.

readable._read(size)#

- `size` Number Number of bytes to read asynchronously

Note: **Implement this function, but do NOT call it directly.**

This function should NOT be called directly. It should be implemented by child classes, and only called by the internal Readable class methods.

All Readable stream implementations must provide a `_read` method to fetch data from the underlying resource.

This method is prefixed with an underscore because it is internal to the class that defines it, and should not be called directly by user programs. However, you **are** expected to override this method in your own extension classes.

When data is available, put it into the read queue by calling `readable.push(chunk)`. If `push` returns false, then you should stop reading. When `_read` is called again, you should start pushing more data.

The `size` argument is advisory. Implementations where a "read" is a single call that returns data can use this to know how much data to fetch. Implementations where that is not relevant, such as TCP or TLS, may ignore this argument, and simply provide data whenever it becomes available. There is no need, for example to "wait" until `size` bytes are available before calling **`stream.push(chunk)`**.

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'`
- return Boolean Whether or not more pushes should be performed

Note: **This function should be called by Readable implementors, NOT by consumers of Readable streams.**

The `_read()` function will not be called again until at least one `push(chunk)` call is made.

The `Readable` class works by putting data into a read queue to be pulled out later by calling the `read()` method when the `'readable'` event fires.

The `push()` method will explicitly insert some data into the read queue. If it is called with `null` then it will signal the end of the data (EOF).

This API is designed to be as flexible as possible. For example, you may be wrapping a lower-level source which has some sort of pause/resume mechanism, and a data callback. In those cases, you could wrap the low-level source object by doing something like this:

```
// 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.

util.inherits(SourceWrapper, Readable);

function SourceWrapper(options) {
  Readable.call(this, options);

  this._source = getLowlevelSourceObject();
  var self = this;

  // Every time there's data, we push it into the internal buffer.
  this._source.ondata = function(chunk) {
    // if push() returns false, then we need to stop reading from source
    if (!self.push(chunk))
```

```
      self._source.readStop();
    };

    // When the source ends, we push the EOF-signaling `null` chunk
    this._source.onend = function() {
      self.push(null);
    };
  }

  // _read will be called when the stream wants to pull more data in
  // the advisory size argument is ignored in this case.
  SourceWrapper.prototype._read = function(size) {
    this._source.readStart();
  };
```

Class: stream.Writable

`stream.Writable` is an abstract class designed to be extended with an underlying implementation of the **`_write(chunk, encoding, callback)`** method.

Please see above under **API for Stream Consumers** for how to consume writable streams in your programs. What follows is an explanation of how to implement Writable streams in your programs.

new stream.Writable([options])#

- `options` Object
  - `highWaterMark` Number Buffer level when **`write()`** starts returning false. Default=16kb, or 16 for `objectMode` streams
  - `decodeStrings` Boolean Whether or not to decode strings into Buffers before passing them to **`_write()`**. Default=true
  - `objectMode` Boolean Whether or not the `write(anyObj)` is a valid operation. If set you can write arbitrary data instead of only `Buffer` / `String` data. Default=false

In classes that extend the Writable class, make sure to call the constructor so that the buffering settings can be properly initialized.

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 this is the encoding type. Ignore if chunk is a buffer. Note that chunk will **always** be a buffer unless the `decodeStrings` option is explicitly set to `false`.
- `callback` Function Call this function (optionally with an error argument) when you are done processing the supplied chunk.

All Writable stream implementations must provide a **`_write()`** method to send data to the underlying resource.

Note: **This function MUST NOT be called directly.** It should be implemented by child classes, and called by the internal Writable class methods only.

Call the callback using the standard `callback(error)` pattern to signal that the write completed successfully or with an error.

If the `decodeStrings` flag is set in the constructor options, then `chunk` may be a string rather than a Buffer, and `encoding` will indicate the sort of string that it is. This is to support implementations that have an optimized handling for certain string data encodings. If you do not explicitly set the `decodeStrings` option to `false`, then you can safely ignore the `encoding` argument, and assume that `chunk` will always be a Buffer.

This method is prefixed with an underscore because it is internal to the class that defines it, and should not be called directly by user programs. However, you **are** expected to override this method in your own extension classes.

writable._writev(chunks, callback)

- `chunks` Array The chunks to be written. Each chunk has following format: `{ chunk: ..., encoding: ... }`.
- `callback` Function Call this function (optionally with an error argument) when you are done processing the supplied chunks.

Note: **This function MUST NOT be called directly.** It may be implemented by child classes, and called by the internal Writable class methods only.

This function is completely optional to implement. In most cases it is unnecessary. If implemented, it will be called with all the chunks that are buffered in the write queue.

Class: stream.Duplex

A "duplex" stream is one that is both Readable and Writable, such as a TCP socket connection.

Note that `stream.Duplex` is an abstract class designed to be extended with an underlying implementation of the `_read(size)` and **`_write(chunk, encoding, callback)`** methods as you would with a Readable or Writable stream class.

Since JavaScript doesn't have multiple prototypal inheritance, this class prototypally inherits from Readable, and then parasitically from Writable. It is thus up to the user to implement both the lowlevel `_read(n)` method as well as the lowlevel **`_write(chunk, encoding, callback)`** method on extension duplex classes.

new stream.Duplex(options)#

- `options` Object Passed to both Writable and Readable constructors. Also has the following fields:
  - `allowHalfOpen` Boolean Default=true. If set to `false`, then the stream will automatically end the readable side when the writable side ends and vice versa.
  - `readableObjectMode` Boolean Default=false. Sets `objectMode` for readable side of the stream. Has no effect if `objectMode` is `true`.
  - `writableObjectMode` Boolean Default=false. Sets `objectMode` for writable side of the stream. Has no effect if `objectMode` is `true`.

In classes that extend the Duplex class, make sure to call the constructor so that the buffering settings can be properly initialized.

Class: stream.Transform

A "transform" stream is a duplex stream where the output is causally connected in some way to the input, such as a **zlib** stream or a **crypto** stream.

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.

Rather than implement the **`_read()`** and **`_write()`** methods, Transform classes must implement the `_transform()` method, and may optionally also implement the `_flush()` method. (See below.)

new stream.Transform([options])#

- `options` Object Passed to both Writable and Readable constructors.

In classes that extend the Transform class, make sure to call the constructor so that the buffering settings can be properly initialized.

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. (Ignore if `decodeStrings` chunk is a buffer.)
- `callback` Function Call this function (optionally with an error argument and data) when you are done processing the supplied chunk.

Note: **This function MUST NOT be called directly.** It should be implemented by child classes, and called by the internal Transform class methods only.

All Transform stream implementations must provide a `_transform` method to accept input and produce output.

`_transform` should do whatever has to be done in this specific Transform class, to handle the bytes being written, and pass them off to the readable portion of the interface. Do asynchronous I/O, process things, and so on.

Call `transform.push(outputChunk)` 0 or more times to generate output from this input chunk, depending on how much data you want to output as a result of this chunk.

Call the callback function only when the current chunk is completely consumed. Note that there may or may not be output as a result of any particular input chunk. If you supply as the second argument to the it will be passed to 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);
}
```

This method is prefixed with an underscore because it is internal to the class that defines it, and should not be called directly by user programs. However, you **are** expected to override this method in your own extension classes.

transform._flush(callback)#

- `callback` Function Call this function (optionally with an error argument) when you are done flushing any remaining data.

Note: **This function MUST NOT be called directly.** It MAY be implemented by child classes, and if so, will be called by the internal Transform class methods only.

In some cases, your transform operation may need to emit a bit more data at the end of the stream. For example, a `Zlib` compression stream will store up some internal state so that it can optimally compress the output. At the end, however, it needs to do the best it can with what is left, so that the data will be complete.

In those cases, you can implement a `_flush` method, which will be called at the very end, after all the written data is consumed, but before emitting `end` to signal the end of the readable side. Just like with `_transform`, call `transform.push(chunk)` zero or more times, as appropriate, and call `callback` when the flush operation is complete.

This method is prefixed with an underscore because it is internal to the class that defines it, and should not be called directly by user programs. However, you **are** expected to override this method in your own extension classes.

Events: 'finish' and 'end'#

The **`finish`** and **`end`** events are from the parent Writable and Readable classes respectively. The `finish` event is fired after `.end()` is called and all chunks have been processed by `_transform`, `end` is fired after all data has been output which is after the callback in `_flush` has been called.

Example: `SimpleProtocol` parser v2#

The example above of a simple protocol parser can be implemented simply by using the higher level **Transform** stream class, similar to the `parseHeader` and `SimpleProtocol v1` examples above.

In this example, rather than providing the input as an argument, it would be piped into the parser, which is a more idiomatic Node stream approach.

```
var util = require('util');
var Transform = require('stream').Transform;
util.inherits(SimpleProtocol, Transform);

function SimpleProtocol(options) {
  if (!(this instanceof SimpleProtocol))
    return new SimpleProtocol(options);

  Transform.call(this, options);
  this._inBody = false;
  this._sawFirstCr = false;
  this._rawHeader = [];
  this.header = null;
}

SimpleProtocol.prototype._transform = function(chunk, encoding, done) {
  if (!this._inBody) {
    // check if the chunk has a \n\n
    var split = -1;
    for (var i = 0; i < chunk.length; i++) {
      if (chunk[i] === 10) { // '\n'
        if (this._sawFirstCr) {
          split = i;
          break;
        } else {
          this._sawFirstCr = true;
        }
      } else {
        this._sawFirstCr = false;
      }
    }

    if (split === -1) {
      // still waiting for the \n\n
      // stash the chunk, and try again.
      this._rawHeader.push(chunk);
    } else {
      this._inBody = true;
      var h = chunk.slice(0, split);
      this._rawHeader.push(h);
      var header = Buffer.concat(this._rawHeader).toString();
      try {
        this.header = JSON.parse(header);
      } catch (er) {
        this.emit('error', new Error('invalid simple protocol data'));
        return;
      }
      // and let them know that we are done parsing the header.
      this.emit('header', this.header);

      // now, because we got some extra data, emit this first.
      this.push(chunk.slice(split));
    }
  } else {
    // from there on, just provide the data to our consumer as-is.
    this.push(chunk);
  }
  done();
};

// Usage:
// var parser = new SimpleProtocol();
// source.pipe(parser)
// Now parser is a readable stream that will emit 'header'
// with the parsed header data.
```

Class: stream.PassThrough

This is a trivial implementation of a **Transform** stream that simply passes the input bytes across to the output. Its purpose is mainly for examples and testing, but there are occasionally use cases where it can come in handy as a building block for novel sorts of streams.

Streams: Under the Hood

Buffering

Both Writable and Readable streams will buffer data on an internal object called `_writableState.buffer` or `_readableState.buffer`, respectively.

The amount of data that will potentially be buffered depends on the `highWaterMark` option which is passed into the constructor.

Buffering in Readable streams happens when the implementation calls **`stream.push(chunk)`**. If the consumer of the Stream does not call `stream.read()`, then the data will sit in the internal queue until it is consumed.

Buffering in Writable streams happens when the user calls **`stream.write(chunk)`** repeatedly, even when `write()` returns `false`.

The purpose of streams, especially with the `pipe()` method, is to limit the buffering of data to acceptable levels, so that sources and destinations of varying speed will not overwhelm the available memory.

`stream.read(0)`

There are some cases where you want to trigger a refresh of the underlying readable stream mechanisms, without actually consuming any data. In that case, you can call `stream.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 `read(0)` will trigger a low-level `_read` call.

There is almost never a need to do this. However, you will see some cases in Node's internals where this is done, particularly in the Readable stream class internals.

`stream.push('')`

Pushing a zero-byte string or Buffer (when not in **Object mode**) has an interesting side effect. Because it *is* a call to **`stream.push()`**, it will end the `reading` process. However, it does *not* add any data to the readable buffer, so there's nothing for a user to consume.

Very rarely, there are cases where you have no data to provide now, but the consumer of your stream (or, perhaps, another bit of your own code) will know when to check again, by calling `stream.read(0)`. In those cases, you *may* call `stream.push('')`.

So far, the only use case for this functionality is in the **tls.CryptoStream** class, which is deprecated in Node v0.12. If you find that you have to use `stream.push('')`, please consider another approach, because it almost certainly indicates that something is horribly wrong.

Compatibility with Older Node Versions

In versions of Node prior to v0.10, the Readable stream interface was simpler, but also less powerful and less useful.

- Rather than waiting for you to call the `read()` method, `'data'` events would start emitting immediately. If you needed to do some I/O to decide how to handle data, then you had to store the chunks in some kind of buffer so that they would not be lost.
- The **`pause()`** method was advisory, rather than guaranteed. This meant that you still had to be prepared to receive `'data'` events even when the stream was in a paused state.

In Node v0.10, the Readable class described below was added. For backwards compatibility with older Node programs, Readable streams switch into "flowing mode" when a `'data'` event handler is added, or when the **`resume()`** method is called. The effect is that, even if you are not using the new `read()` method and `'readable'` event, you no longer have to worry about losing `'data'` chunks.

Most programs will continue to function normally. However, this introduces an edge case in the following conditions:

- No **`'data'` event** handler is added.
- The **`resume()`** method is never called.
- The stream is not piped to any writable destination.

For example, consider the following code:

```
// WARNING!  BROKEN!
net.createServer(function(socket) {

  // we add an 'end' method, but never consume the data
  socket.on('end', function() {
    // It will never get here.
    socket.end('I got your message (but didnt read it)\n');
  });

}).listen(1337);
```

In versions of node prior to v0.10, the incoming message data would be simply discarded. However, in Node v0.10 and beyond, the socket will remain paused forever.

The workaround in this situation is to call the `resume()` method to start the flow of data:

```
// Workaround
net.createServer(function(socket) {

  socket.on('end', function() {
    socket.end('I got your message (but didnt read it)\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 `wrap()` method.

Object Mode

Normally, Streams operate on Strings and Buffers exclusively.

Streams that are in **object mode** can emit generic JavaScript values other than Buffers and Strings.

A Readable stream in object mode will always return a single item from a call to `stream.read(size)`, regardless of what the size argument is.

A Writable stream in object mode will always ignore the `encoding` argument to `stream.write(data, encoding)`.

The special value `null` still retains its special value for object mode streams. That is, for object mode readable streams, `null` as a return value from `stream.read()` indicates that there is no more data, and **`stream.push(null)`** will signal the end of stream data (`EOF`).

No streams in Node core are object mode streams. This pattern is only used by userland streaming libraries.

You should set `objectMode` in your stream child class constructor on the options object. Setting `objectMode` mid-stream is not safe.

For Duplex streams `objectMode` can be set exclusively for readable or writable side with `readableObjectMode` and `writableObjectMode` respectively. These options can be used to implement parsers and serializers with Transform streams.

```
var util = require('util');
var StringDecoder = require('string_decoder').StringDecoder;
var Transform = require('stream').Transform;
util.inherits(JSONParseStream, Transform);

// Gets \n-delimited JSON string data, and emits the parsed objects
function JSONParseStream() {
  if (!(this instanceof JSONParseStream))
    return new JSONParseStream();

  Transform.call(this, { readableObjectMode : true });

  this._buffer = '';
  this._decoder = new StringDecoder('utf8');
}

JSONParseStream.prototype._transform = function(chunk, encoding, cb) {
  this._buffer += this._decoder.write(chunk);
  // split on newlines
  var lines = this._buffer.split(/\r?\n/);
  // keep the last partial line buffered
  this._buffer = lines.pop();
  for (var l = 0; l < lines.length; l++) {
    var line = lines[l];
    try {
      var obj = JSON.parse(line);
    } catch (er) {
      this.emit('error', er);
      return;
    }
    // push the parsed object out to the readable consumer
    this.push(obj);
  }
  cb();
};

JSONParseStream.prototype._flush = function(cb) {
  // Just handle any leftover
  var rem = this._buffer.trim();
  if (rem) {
    try {
      var obj = JSON.parse(rem);
    } catch (er) {
      this.emit('error', er);
      return;
    }
    // push the parsed object out to the readable consumer
    this.push(obj);
  }
  cb();
};
```