3 Using Streams

This chapter covers the following topics

- Handling data larger than fits in memory
- Decoupling I/O from modules
- Reducing latency in our apps
- Composing pipelines

Introduction

Streams are one of the best features in Node. They have been a big part of the ecosystem since the early days of Node and today thousands of modules exists on npm that help us compose all kinds of great stream based apps. They allow us to work with large volumes of data in environments with limited resources. In addition to that they help us decouple our applications by supplying a generic abstraction that most I/O patterns work with.

In this chapter we're going to explore why streams are such a valuable abstraction, how to safely compose streams together in a production environment, and convenient utilities to stream creation and management.

Processing big data

Let's dive right into it by looking at a classic Node problem, counting all Node modules available on npm. The npm registry exposes an HTTP endpoint where we can get the entire contents of the npm registry content as JSON.

Using the command line tool `curl` which is included (or at least installable) on most operating systems we can try it out.

```
$ curl https://skimdb.npmjs.com/registry/_changes?include_docs=true
```

This will prints a new line delimited JSON stream of all modules.

The JSON stream returned by the registry contains a JSON object for each module

stored on npm followed by a new line character.

A simple Node program that counts all modules could look like this:

```
var request = require('request')
var registryUrl = 'https://skimdb.npmjs.com/registry/_changes?include_d

request(registryUrl, function (err, data) {
  if (err) throw err
  var numberOfLines = data.split('\n').length + 1
  console.log('Total modules on npm: ' + numberOfLines)
})
```

If we try and run the above program we'll notice a couple of things.

First of all this program takes quite a long time to run. Second, depending on the machine we are using, there is a very good chance the program will crash with an "out of memory" error.

Why is this happening?

The npm registry stores a very large amount of JSON data, and it takes quite a bit of memory to buffer it all. Let us investigate how we can use streams to improve our program.

Getting Ready

Let's create a folder called `self-read` with an `index.js` file.

How to do it

A good way to start understanding how streams work is to look at how Node core uses them.

The core `fs` module has a `createReadStream` method, let's use that to make a read stream:

```
const rs = fs.createReadStream(__filename)
```

The `__filename` variable is provided by Node, it holds the absolute path of the file currently being executed (in our case it will point to the `index.js` file in the `self-read` folder).

The first thing to notice is that this method is synchronous.

Normally when we work with I/O in Node we have to provide a callback.

Streams abstract this away by returning an object instance that represents the entire contents of the file. How do we get the file data out of this abstraction?

We can extract data from the stream using the `data` event.

Let's attach a data listener that will be called every time a new small chunk of the file has been read.

```
rs.on('data', (data) => {
  console.log('Read chunk:', data)
})

rs.on('end', () => {
  console.log('No more data')
})
```

When we are done reading the file the stream will emit an `end` event.

Let's try this out

```
$ node index.js
```

How it works

Streams are bundled with Node core as a core module (the `streams`) module. Other parts of core such as `fs` rely on the `streams` module for their higher level interfaces. The two main stream abstractions are a readable stream and a writable stream.

In our case we use a readable stream (as provided by the `fs` module), to read our source file (`index.js`) a chunk at a time. Since our file is smaller than the maximum size per chunk (16KB), only one chunk is read.

The `data` event is therefore only emitted once, and then the `end` event is emitted.

There's more

For more information about the different stream base classes checkout the Node

stream docs.

Types of Stream

If we want to make a stream that provides data for other users to read we need to make a *Readable stream*. An example of a readable stream could be a stream that reads data from a file stored on disk.

If we want to make a stream others users can write data to, we need to make a *Writable stream*. An example of a writable stream could be a stream that writes data to a file stored on disk.

> **Inspecting all core stream interfaces** 
>
> Node core provides base implementations of all these variations of streams that we can extend to support various use cases. We can use the `node -p "require('stream')"` as a convenient way to take look at available stream implementations

Sometimes you want to make a stream that is both readable and writable at the same time. We call these *Duplex streams*. An example of a duplex stream could be a TCP network stream that both allows us to read data from the network and write data back at the same time.

A special case of a duplex stream is a stream that transforms the data being written to it and makes the transformed data available to read out of the stream. We call these *Transform streams*. An example of a transform stream could be a gzip stream that compresses the input data written to it.

Processing infinite amounts of data

Using the `data` event we can process the file a small chunk of the time instead without using a lot of memory. For example, we may wish to count the number of bytes in a file.

Let's create a new folder called `infinite-read` with a `index.js`.

Assuming we are using a Unix-like machine we can try to tweak this example to count the number of bytes in `/dev/urandom`. This is an infinite file that contains random data.

Let's write the following into `index.js`:

```
const rs = fs.createReadStream('/dev/urandom')
const size = 0

rs.on('data', (data) => {
  size += data.length
  console.log('File size:', size)
})
```

Now we can run our program:

```
$ node index.js
```

Notice that the program does not crash even though the file is infinite. It just keeps counting bytes!

Scalability is one of the best features about streams in general as most of the programs written using streams will scale well with any input size.

Understanding stream events

All streams inherit from EventEmitter and emit a series of different events. When working with streams it is a good idea to understand some of the more important events being emitted. Knowing what each event means will make debugging streams a lot easier.

- `data`. Emitted when new data is read from a readable stream. The data is provided as the first argument to the event handler. Beware that unlike other event handlers attaching a data listener has side effects. When the first data listener is attached your stream will be unpaused. You should never emit `data` yourself. Always use the `.push()` function instead.
- `end`. Emitted when a readable stream has no more data available AND all available data has been read. You should never emit `end` yourself. Use `.push(null)` instead.
- `finish`. Emitted when a writable stream has been ended AND all pending writes has been completed. Similar to the above events you should never emit `finish` yourself. Use `.end()` to trigger finish manually pipe a readable stream to it.
- `close`. Loosely defined in the stream docs, `close` is usually emitted when

the stream is fully closed. Contrary to `end` and `finish` a stream is *not* guaranteed to emit this event. It is fully up to the implementer to do this.

- `error`. Emitted when a stream has experienced an error. Tends to followed by a `close` event although, again, no guarantees that this will happen.
- `pause`. Emitted when a readable stream has been paused. Pausing will happen when either backpressure happens or if the `.pause` method is explicitly called. For most use cases you can just ignore this event although it is useful to listen for, for debugging purposes sometimes.
- `resume`. Emitted when a readable stream goes from being paused to being resumed again. Will happen when the writable stream you are piping to has been drained or if `.resume` has been explicitly called.

See also

- TBD

Using the `pipe` method

A pipe is used to connect streams together. DOS and Unix-like shells use the vertical bar (|) to pipe the output of one program to another; we can chain several pipes together to process and massage data in number of ways.

Likewise, the Streams API affords us the `pipe` method to channel data through multiple streams. Every readable stream has a pipe method that expects a writable stream (the destination) as its first parameter.

In this recipe we're going to pipe several streams together.

Getting Ready

Let's create a folder called `piper`, initialize it as a package, and install `tar-map-stream`, and create an `index.js` file:

```
$ mkdir piper
$ cd piper
$ npm init -y
$ npm install tar-map-stream
$ touch index.js
```

How to do it

In our `index.js` file let's begin by requiring the dependencies we'll be using to create various streams:

```
const zlib = require('zlib')
const map = require('tar-map-stream')
```

Let's imagine we want to take the gzipped tarball of the very first available version of Node, and change all the file paths in that tarball, as well as altering the `uname` (owner user) and `mtime` (modified time) fields of each file.

Let's create some streams we'll be using to do that:

```
const decompress = zlib.createGunzip()
const whoami = process.env.USER || process.env.USERNAME
const convert = map((header) => {
  header.uname = whoami
  header.mtime = new Date()
  header.name = header.name.replace('node-v0.1.100', 'edon-v0.0.0')
  return header
})
const compress = zlib.createGzip()
```

Finally we'll set up the pipeline:

```
process.stdin
  .pipe(decompress)
  .pipe(convert)
  .pipe(compress)
  .pipe(process.stdout)
```

> ### Don't use `pipe` in production! 
>
> For most cases, `pipe` should be avoided in a production server context. Instead we recommend `pump`, see the next recipe in this chapter for more.

We can use our program like so:

```
$ curl https://nodejs.org/dist/v0.1.100/node-v0.1.100.tar.gz | node ind
```

We can list the contents of the tar archive to ensure the paths and stats are updated like so:

```
$ tar -tvf edon.tar.gz
```

How it works

The `pipe` method attaches a `data` event listener to the source stream (the stream on which `pipe` is called), which writes incoming data to the destination stream (the stream that was passed into `pipe`).

When we string several streams together with the `pipe` method we're essentially instructing Node to shuffle data through those streams.

Using `pipe` is safer than using `data` events and then writing to another stream directly, because it also handles back pressure for free. Back pressure has to be applied to source streams that process data faster than destination streams, so that the destination streams memory doesn't grow out of control due to a data back log.

Our recipe uses five streams, and creates three of them. The `process.stdin` and `process.stdout` streams connect with the terminal STDIN and STDOUT interfaces respectively. This is what allows us to pipe from the `curl` command to our program and the redirect output to the `edon.tar.gz` file.

The `compress` and `decompress` streams are created with the core `zlib` module, using the `createGunzip` and `createGzip` methods, which return transform streams. A transform stream has both readable and writable interfaces, and will mutate the data in some way as it flows through the pipeline.

The final `convert` stream is also a transform stream that's generated by the `tar-map-stream` module - which we assigned to `map`. When we call `map` it returns a stream that can parse a tar archive and call a function with the header information of each file in the archive. Whatever we return from from the function supplied to `map` will become the new header information for the tar archive.

So when we use `curl` to fetch the first available version of Node, we use a Unix pipe (`|`) to shuffle the data from `curl` into our program. This data comes in through the `process.stdin` stream, and is passed on to the `decompress` stream. The `decompress` stream understands the GZIP format and deflates the content

accordingly. It propagates each decompressed chunk to the next stream: our `convert` stream. The `convert` stream incrementally parses the `tar` archive, calling our function every time a header is encountered, and then outputs content in the same tar format with our modified headers. The `compress` stream gzips our new tar and then passes the data through the `process.stdout` stream. Back on the command line we've used the IO redirect syntax (`>`) to write the data into the `edon.tar.gz` file.

There's more

Let's take a look at the one option which can be passed to the `pipe` method.

Keeping Piped Streams Alive

By default, when one stream is piped to another, the stream being piped to (the destination), is ended when the stream being piped from (the source) has ended.

Sometimes, we may want to make additional writes to a stream when a source stream is complete.

Let's create a folder called `pipe-without-end` , with two files, `broken.js` and `index.js` :

```
$ mkdir pipe-without-end
$ cd pipe-without-end
$ touch broken.js
$ touch index.js
```

Let's put the following in `broken.js` :

```
const net = require('net')
const fs = require('fs')

net.createServer((socket) => {
  const content = fs.createReadStream(__filename)
  content.pipe(socket)
  content.on('end', () => {
    socket.end('\n======= Footer =======\n')
  })
}).listen(3000)
```

Now let's start our broken server:

```
$ node broken.js
```

We can try out the TCP server in several ways, such as `telnet localhost 3000` or with netcat `nc localhost 3000`, but even navigating a browser to `http://localhost:3000`, or using curl will work. Let's use `curl`:

```
$ curl http://localhost:3000
```

This will cause our `broken.js` server to crash, with the error "Error: write after end". This is because when the `content` stream ended, it also ended the `socket` stream. But we want to append a footer to the content when the `content` stream is ended.

Let's make our `index.js` look like this:

```
const net = require('net')
const fs = require('fs')

net.createServer((socket) => {
  const content = fs.createReadStream(__filename)
  content.pipe(socket, {end: false})
  content.on('end', () => {
    socket.end('\n======= Footer =======\n')
  })
}).listen(3000)
```

Notice the second argument passed to pipe is an object with `end` set to `false`. This instructs the `pipe` method to avoid ending the destination stream when a source stream ends.

If we start our fixed server:

```
$ node index.js
```

And hit it with `curl`:

```
$ curl http://localhost:3000
```

We'll see our content, along with the footer, and the server stays alive.

See also

- TBD

Piping streams in production

The `pipe` method is one of the most well known features of streams, it allows us to compose advanced streaming pipelines as a single line of code.

As a part of Node core, we discussed the `pipe` method in the previous recipe, and it can be useful for cases where process uptime isn't important (such as CLI tools).

Unfortunately, however, it lacks a very important feature: error handling.

If one of the streams in a pipeline composed with `pipe` fails, the pipeline is simply "unpiped". It is up to us to detect the error and then afterwards destroy the remaining streams so they do not leak any resources. This can easily lead to memory leaks.

Let's consider the following example:

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

const server = http.createServer((req, res) => {
  fs.createReadStream('big.file').pipe(res)
})

server.listen(8080)
```

A simple, straight forward, HTTP server that serves a big file to its users.

Since this server is using `pipe` to send back the file there is a big chance that this server will produce memory and file descriptor leaks while running.

If the HTTP response were to close before the file has been fully streamed to the user (for instance, when the user closes their browser), we will leak a file descriptor and a piece of memory used by the file stream. The file stream stays in memory because it's never closed.

We have to handle `error` and `close` events, and destroy other streams in the pipeline. This adds a lot of boilerplate, and can be difficult to cover all cases.

In this recipe we're going to explore the `pump` module, which is built specifically to solve this problem.

Getting Ready

Let's create a folder called `big-file-server`, with an `index.js`.

We'll need to initialize the folder as a package, and install the `pump` module:

```
$ mkdir big-file-server
$ cd big-file-server
$ npm init -y
$ npm install --save pump
```

We'll also need a big file, so let's create that quickly with `dd`:

```
$ TODO    > big.file
```

How to do it

We'll begin by requiring the `http` and `pump` modules:

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

Now let's create our HTTP server and `pump` instead of `pipe` our big file stream to our response stream:

```
const server = http.createServer((req, res) => {
  const stream = fs.createReadStream('big.file')
  pump(stream, response, (err) => {
    if (err) {
      return console.error('File was not fully streamed to the user', e
    }
    console.log('File was fully streamed to the user')
  })
})

server.listen(8080)
```

Piping many streams with `pump` 

> If our pipeline has more than two streams we simply pass all of them to `pump`: `pump(stream1, stream2, stream3, ...)`

Now let's run our server

```
$ node index.js
```

If we use curl and hit Ctrl+C before finishing the download, we should be able to trigger the error state, with the server logging that the file was not fully streamed to the user.

```
$ curl http://localhost:8080 # hit Ctrl + C before finish
```

How it works

Every stream we pass into the `pump` function will be piped to the next (as per order of arguments passed into `pump`). If the last argument past to `pump` is a function the `pump` module will call that function when all streams have finished (or one has errored).

Internally, `pump` attaches `close` and `error` handlers, and also covers other esoteric cases where a stream in a pipeline may close without notifying other streams.

If one of the streams close, the other streams are destroyed and the callback passed to `pump` is called.

It is possible to handle this manually, but the boilerplate overhead and potential for missed cases is generally unacceptable for production code.

For instance, here's our specific case from the recipe altered to handle the response closing:

```
const server = http.createServer((req, res) => {
  const stream = fs.createReadStream('big.file')
  stream.pipe(res)
  res.on('close', () => {
    stream.destroy()
  })
```

```
})
```

If we multiply that by every stream in a pipeline, and then multiply it again by every possible case (mostly `close` and `error` but also esoteric cases) we end up with an extraordinary amount of boilerplate.

There are very few use cases where we want to use `pipe` (sometimes we want to apply manual error handling) instead of `pump` so for production purposes it is a lot safer to always use `pump` instead `pipe`.

There's more

Here's some other usual things we can do with `pump`.

Use `pumpify` to expose pipelines

When writing pipelines, especially as part of module, we might want to expose these pipelines to a user. So how do we do that? As described above a pipeline consists of a series of transform streams. We write data to the first stream in the pipeline and the data flows through it until it is written to the final stream.

```
function pipeline () {
  pump(stream1, stream2, stream3, stream4)
}
```

If we were to expose the above pipeline to a user we would need to both return `stream1` and `stream4`. `stream1` is the stream a user should write the pipeline data to and `stream4` is the stream the user should read the pipeline results from. Since we only need to write to `stream1` and only read from `stream4` we could just combine to two streams into a new duplex stream that would then represent the entire pipeline.

The npm module pumpify does *exactly* this

```
var pipe = pipeline()

pipe.write('hello') // written to stream1

pipe.on('data', function (data) {
  console.log(data) // read from stream4
})
```

```
pipe.on('finish', function () {
  // all data was succesfully flushe to stream4
})

function pipeline () {
  return pumpify(stream1, stream2, stream3, stream4)
}
```

See also

- TBD

Creating transform streams

Streams allow for asynchronous functional programming, The most common stream is the transform stream, it's a black box that takes input and produce output asynchronously.

In this recipe, we'll look at creating a transform stream with the `through2` module, in the **There's More** section we'll look at how to create streams with the core `streams` module.

Getting Ready

Let's create a folder called `through-streams` with an `index.js`, initialize the folder as a package and install `through2`:

```
$ mkdir through-streams
$ cd through-streams
$ npm init -y
$ npm install through2
$ touch index.js
```

> ### Why the 2? 
>
> The `through2` module is a successor to the `through` module. The `through` module was built against an earlier Node core streams API (retrospectively called Streams 1 API). Later versions of Node introduced Streams 2 (and indeed 3). The `through2` module was written to use the superior Streams 2 API (and is still relevant for the Streams 3 API, there's no need for a

> `through3`!). In fact, any streams utility module on npm suffixed with the number 2 is named as such for the same reasons (such as `from2`, `to2`, `split2` and so forth)

How to do it

First we'll require `through2`:

```
const through = require('through2')
```

Next we'll use it to create a stream that upper cases incoming data:

```
const upper = through((chunk, enc, cb) => {
  cb(null, chunk.toString().toUpperCase())
})
```

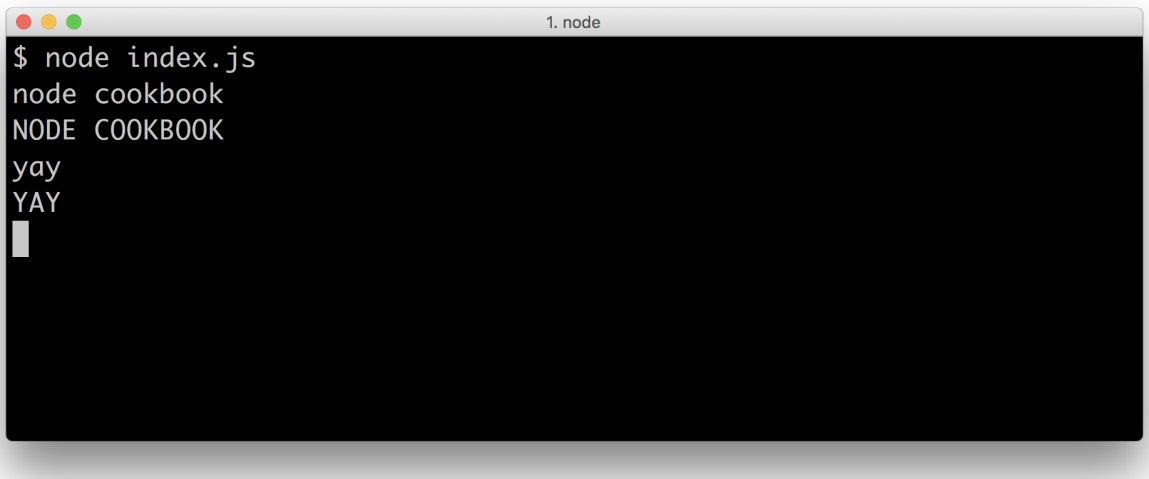
Finally we'll create a pipeline from the terminals STDIN through our `upper` stream to the terminals STDOUT:

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

Now if we start our program:

```
$ node index.js
```

Each line we type into the terminal will be uppercased, as demonstrated in the following image:



How it works

The `through2` module provides a thin layer over the core streams `Transform` constructor. It ultimately attaches the function we provide to as the `_transform` method of a stream instance which inherits from the `Transform` constructor.

When we create our `upper` stream, we call `through` and pass it a function. This is called the transform function. Each piece of data that the stream recieves will be passed to this function. The first `chunk` is the data being received, the `enc` parameter indicates the encoding of the data, and the `cb` parameter is a callback function which we call to indicate we've finished processing the data, and pass our transformed data through.

There are a couple of benefits of using the `through2` module over core primitives. Primarily, it's typically less noisy, easier for human reading and uses the `readable-stream` module. The `readable-stream` module is the core stream module, but published to npm as the latest streams implementation. This keeps behavior consistent across Node versions, using `through2` implicitly grants this advantage and we don't have to think about it.

There's more

How would we go about creating core transform streams, also let's explore object streams.

Transform streams with Node's core `stream` module

Let's create a folder called `core-transform-streams` with a `prototypal.js`, `classical`, `modern.js` and `index.js` files:

```
$ mkdir core-transform-streams
$ touch prototypal.js classical.js modern.js index.js
```

We'll use these files to explore the evolution of stream creation.

Let's write the following in `prototypal.js`:

```
const stream = require('stream')
const util = require('util')

function MyTransform(opts) {
```

```
  stream.Transform.call(this, opts)
}

util.inherits(MyTransform, stream.Transform)

MyTransform.prototype._transform = function (chunk, enc, cb) {
  cb(null, chunk.toString().toUpperCase())
}

const upper = new MyTransform()

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

In earlier version of Node this was the canonical way to create streams, with the advent of EcmaScript 2015 (ES6) classes, there's a slightly less noisy approach.

Let's make the `classical.js` file look as follows:

```
const {Transform} = require('stream')

class MyTransform extends Transform {
  _transform (chunk, enc, cb) {
    cb(null, chunk.toString().toUpperCase())
  }
}

const upper = new MyTransform()

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

Still applying the abstract method paradigm with an underscored namespace is esoteric for JavaScript, and the use of classes is generally discouraged by the authors since, to be clear, ES6 classes are not classes - which leads to confusion.

In Node 4, support for the `transform` option was added, this allows for a more functional approach (similar to `through2`), let's make `modern.js` look as follows:

```
const {Transform} = require('stream')

const upper = Transform({
  transform: (chunk, enc, cb) => {
    cb(null, chunk.toString().toUpperCase())
  }
})

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

The `Transform` constructor doesn't require `new` invocation, so we can call it as a function. We can pass our transform function as the `transform` property on the options object passed to the `Transform` function.

For our final mutation, let's initialize the folder as a package and install `readable-stream`:

```
$ npm init -y
$ npm install readable-stream
```

To have complete parity with the `through2` module, we need to use `readable-stream` instead of the core `stream` module.

Let's make `index.js` look as follows:

```
const {Transform} = require('readable-stream')

const upper = Transform({
  transform: (chunk, enc, cb) => {
    cb(null, chunk.toString().toUpperCase())
  }
})

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

This of course limits us to using Node 4 or above, so isn't a recommended pattern for public modules, the prototypal approach is still most appropriate for modules we intend to publish to npm.

Creating Object mode transform streams

If our stream is not returning serializable data (a Buffer or a string) we need to make it use "object mode". Object mode just means that the values returned are generic objects and the only different is how much data is buffered. Per default when not using object mode the stream will buffer around 16kb of data before pausing. When using object mode it will start pausing when 16 objects have been buffered.

Let's create folder called `object-streams`, initialize it as a package, install `through2` and `ndjson` and create an `index.js` file:

```
$ mkdir object-streams
```

```
$ cd object-streams
$ npm init -y
$ npm install through2 ndjson
$ touch index.js
```

Let's make `index.js` look like this:

```
const through = require('through2')
const {serialize} = require('ndjson')

const xyz = through.obj(({x, y}, enc, cb) => {
  cb(null, {z: x + y})
})

xyz.pipe(serialize()).pipe(process.stdout)

xyz.write({x: 199, y: 3})

xyz.write({x: 10, y: 12})
```

We can create an object stream with `through2` using the `obj` method. The behavior of `through.obj` is the same as `through`, except instead of data chunks our transform function receives and responds with objects.

We use the `ndjson` module's `serialize` function to create a serializer stream which converts streamed objects into newline delimited JSON. The serializer stream is a hybrid stream where the writable side is in object mode, but the readable side isn't. Objects go in, buffers come out.

With core streams we pass an `objectMode` option to create an object stream instead. Let's create a `core.js` file in the same folder,

```
$ touch core.js
```

Now we'll fill it with the following code:

```
const {Transform} = require('stream')
const {serialize} = require('ndjson')

const xyz = Transform({
  objectMode: true,
  transform: ({x, y}, enc, cb) => { cb(null, {z: x + y}) }
})
```

```
xyz.pipe(serialize()).pipe(process.stdout)

xyz.write({x: 199, y: 3})

xyz.write({x: 10, y: 12})
```

See also

- TBD

Creating Readable and Writable Streams

Readable streams allow us to do things like representing infinite data series and reading out data that does not necessarily fit in memory, and much more. Writable streams can be created to connect with outputs that operate at the C level to control hardware (such as sockets), to wrap around other objects that aren't streams but nevertheless have a some form of API to where data is pushed to them, or to collect chunks together and potentially process them in batch.

In this recipe we're going create Readable and Writable streams using the `from2` and `to2` modules, in the **There's More** section we'll discover how to do the equivalent with Node's core streams module.

Getting Ready

Let's create a folder called `from2-to2-streams`, initialize it as a package, install the `from2` and `to2` modules and create an `index.js` file:

```
$ mkdir from2-to2-streams
$ cd from2-to2-streams
$ npm init -y
$ npm install --save from2 to2
$ touch index.js
```

How to do it

We'll start of by requiring `from2` and `to2`:

```
const from = require('from2')
const to = require('to2')
```

Next let's create our read stream:

```
const rs = from(() => {
  rs.push(Buffer('Hello, World!'))
  rs.push(null)
})
```

To consume data from the stream we either need to attach a `data` listener or pipe the stream to a writable stream.

As an intermediate step to check our stream, we can add a data listener like so:

```
rs.on('data', (data) => {
  console.log(data.toString())
})
```

Now let's try running our program:

```
$ node index.js
```

We should see the readable stream print out the `Hello, World!` message, via the data event listener.

But we're not done! Let's comment out the `data` handler, like so:

```
// rs.on('data', (data) => {
//   console.log(data.toString())
// })
```

We're going to create a writable stream that can we can pipe our read stream to.

```
const ws = to((data, enc, cb) => {
  console.log(`Data written: ${data.toString()}`)
  cb()
})
```

Finally we add the following line to our `index.js` file:

```
rs.pipe(ws)
```

Now if we run our program, again:

```
$ node index.js
```

We should see "Data written: Hello, World!"

How it works

The `from2` module wraps the `stream.Readable` base constructor and creates the stream for us. It also adds some extra benefits, such as a `destroy` function to cleanly free up stream resources and the ability to perform asynchronous pushing (see the **There's More** section for more).

> ### Object Mode
>
> Like `through2`, both the `from2` and `to2` modules have `obj` methods which allow for convenient creation of object streams. See the **There's More** section of the **Creating transform streams** recipe for more.

The `to2` module is actually an alias for the `flush-write-stream` module, which similarly supplies a `destroy` function, and the ability to supply a function (the flush function) which supplies final writes to the stream before it finishes.

When we `pipe` the `rs` stream to the `ws` stream, the "Hello World" string pushed (with `rs.push`) inside the read function passed to `from2` is emitted as a `data` event which the `pipe` method has hooked into so that the event causes a write to our `ws` stream. The write function (as supplied to the `to` call), dutifully logs out the "Data written: Hello World" message, and then calls `cb` to indicate it's ready for the next piece of data. The `null` primitive is supplied to the second call to `rs.push` inside the function supplied to the `from` invocation. This indicates that the stream has finished, and it triggers it's own `end` event. Internally, an `end` event listener calls the `end` method on the destination stream (the stream passed to `pipe`, in our case `ws`).

At this point our process has nothing left to do, and the program finishes.

There's more

How do we achieve with just the core stream module? Does using core have any

drawbacks (other than the additional syntax?)

Readable and Writable streams with Node's core `stream` module

If we wanted our own readable stream we would need the `stream.Readable` base constructor.

This base class will call a special method called `_read`. It's up to us to implement the `_read` method. Since Node 4, we can also supply a `read` property to an options object which will the supplied function to be added as the `_read` method of the returned instance.

Whenever this method is called the stream expects us to provide more data available that can be consumed by the stream. We can add data to the stream by calling the `push` method with a new chunk of data.

> ### Using `readable-stream` instead of `stream` 
>
> To allow universal behavior across Node modules, if we ever use the core `stream` module to create streams, we should actually use the `readable-stream` module available on npm. This an up to date and multi-version compatible representation of the core streams module and ensures consistency.

Let's create a folder called `core-streams` and create an `index.js` file inside.

At the top of `index.js` we write:

```
const {Readable, Writable} = require('stream')

const rs = Readable({
  read: () => {
    rs.push(Buffer('Hello, World!'))
    rs.push(null)
  }
})
```

Each call to `push` sends data through the stream. When we pass `null` to `push` we're informing the `stream.Readable` interface that there is no more data available.

The use of the `read` option instead of attaching a `_read` method is only appropriate for scenarios where our code is expected to be used by Node 4 and above (the same goes for the use of destructing context and fat arrow lambda functions).

To create a writable stream we need the `stream.Writable` base class. When data is written to the stream the writable base class will buffer the data internally and call the `_write` method that it expects us to implement. Likewise from Node 4 we can use the `write` option for a nicer syntax. Again this approach isn't appropriate for modules which are intended to be made publicly available, since it doesn't cater to legacy Node users.

Now to the bottom of our `index.js` file let's add the following:

```
const ws = Writable({
  write: (data, enc, cb) => {
    console.log(`Data written: ${data.toString()}`)
    cb()
  }
})
```

To write data to the stream we can either do it manually using the `write` method or we can pipe a readable stream to it.

If we want to move the data from a readable to a writable stream the `pipe` method available on readable streams is a much more elegant solution than using the `data` event on the readable stream and calling `write` on the writable stream (but remember we should use `pump` in production).

Let's add this final line to our `index.js` file:

```
rs.pipe(ws)
```

Now we can run our program:

```
$ node index.js
```

This should print out "Data written: Hello, World!".

Core Readable Streams flow control issue

The `_read` method on readable streams does not accept a callback. Since a stream usually contains more than just a single buffer of data the stream needs to call the `_read` method more than once.

The way it does this is by waiting for us to call `push` and then calling `_read` again if the internal buffer of the stream has available space.

A problem with this approach is that if we want to call `push` more than once, in an asynchronous way this becomes problematic.

Let's create a folder called `readable-flow-control`, with a file called `undefined-behavior.js` containing the following:

```
// WARNING: DOES NOT WORK AS EXPECTED
const {Readable} = require('stream')
const rs = Readable({
  read: () => {
    setTimeout(() => {
      rs.push('Data 0')
      setTimeout(() => {
        rs.push('Data 1')
      }, 50)
    }, 100)
  }
})

rs.on('data', (data) => {
  console.log(data.toString())
})
```

If we run that:

```
$ node undefined-behavior.js
```

We might expect it to produce a stream of alternating `Data 0`, `Data 1` buffers but in reality it has undefined behavior.

Luckily as we show in this recipe, there are more user friendly modules available (such as as `from2`) to make all of this easier.

Let's install `from2` into our folder and create a file called `expected-behavior.js`:

```
$ npm init -y
$ npm install --save from2
```

```
$ touch expected-behavior.js
```

We make the `expected-behavior.js` contain the following content:

```
const from = require('from2')
const rs = from((size, cb) => {
  setTimeout(() => {
    rs.push('Data 0')
    setTimeout(() => {
      rs.push('Data 1')
      cb()
    }, 50)
  }, 100)
})

rs.on('data', (data) => {
  console.log(data.toString())
})
```

Now if we run that

```
$ node expected-behavior.js
```

We'll see alternating messages, as expected.

Stream destruction

As opposed to using the `stream.Readable` constructor in Node core to create your own readable streams `from2` adds another essential feature. It adds a way to stop or destroy the stream prematurely.

Core streams in Node actually *do not* document a way to do this in general but most used streams support a `destroy` method that will destroy a stream before it emits all of its data. When using the `destroy` method that `from2` provides the stream will stop emitting data and emit a `close` event to indicate that no more data will be emitted. It won't necessarily emit an `end` in this case.

To showcase the `destroy` method, we'll create an infinite stream (a fun sub-genre of readable streams, that allow for infinite data with finite memory).

Let's create a folder called `stream-destruction`, initialize it as a package, install `from2` and create an `index.js` file:

```
$ mkdir stream-destruction
$ cd stream-destruction
$ npm init -y
$ npm install --save from2
$ touch index.js
```

At the top of `index.js` we write:

```
const from = require('from2')

function createInfiniteTickStream () {
  var tick = 0
  return from.obj((size, cb) => {
    setImmediate(() => cb(null, {tick: tick++}))
  })
}
```

Let's create the stream and log each `data` event:

```
const stream = createInfiniteTickStream()

stream.on('data', (data) => {
  console.log(data)
})
```

Let's run our program so far:

```
$ node index.js
```

We'll notice that it just floods the console as it never ends.

Since an infinite stream won't end by itself we need to have a mechanism for which we can tell it from the outside that it should stop. We need this incase we are consuming the stream and one of the downstream dependents experiences an error which makes us wanting to shutdown the pipeline.

Now let's add the following to our `index.js` file:

```
stream.on('close', () => {
  console.log('(stream destroyed)')
})
```

```
setTimeout(() => {
  stream.destroy()
}, 2000)
```

Running the above code will make the tick stream flood the console for about 2s and then stop, while a final message "(stream destroyed)" is printed to the console before the program exits.

The `destroy` method is extremely useful in many applications and more or less essential when doing any kind of stream error handling.

For this reason using `from2` (and other stream modules described in this book) is highly recommended over using the core stream module.

Composing duplex streams

A duplex stream is a stream with a readable and writable interface. We can take a readable stream and a writeable stream and join them as a duplex stream using the `duplexify` module.

Let's create a folder called `composing-duplex-streams`, initialize as a package, install `from2`, `to2` and `duplexify` and create an an `index.js` file:

```
$ mkdir composing-duplex-streams
$ cd composing-duplex-streams
$ npm init -y
$ npm install --save from2 to2 duplexify
$ touch index.js
```

Then in our `index.js` file we'll write:

```
const from = require('from2')
const to = require('to2')
const duplexify = require('duplexify')

const rs = from(() => {
  rs.push(Buffer('Hello, World!'))
  rs.push(null)
})

const ws = to((data, enc, cb) => {
  console.log(`Data written: ${data.toString()}`)
  cb()
})
```

```
const stream = duplexify(ws, rs)

stream.pipe(stream)
```

We're using the same readable and writable streams from the main recipe (`rs` and `ws`), however we create the `stream` assignment by passing `ws` and `rs` to `duplexify` . Now instead of piping `rs` to `ws` we can pipe `stream` to itself.

This can be a very useful API pattern, when we want to return or export two streams that are interrelated in some way.

See also

- TBD

Decoupling I/O

Getting Ready

How to do it

How it works

There's more

See also