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 thousdans 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.

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.

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

// request.pipe(decrompress).pipe(parse).pipe(analyse)

Getting Ready

How to do it

How it works

There's more

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. Since it's part of Node core we discuss in the previous recipe.

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.

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

Creating streams

// through2, from2

Getting Ready

How to do it

How it works

There's more

Creating streams with Node's core `stream` module

Node core provides base implementations of all these variations of streams that we can extend to support various use cases.

We can access the core base implementations by requiring the stream module in node.

We can easily take a look at these with the following command

```
$ node -p "require('stream')"
```

This will print the various base interfaces supplied by the `stream` module.

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

This base class will call a special method called `_read`. It's up to us to implement the `_read` method. 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 `readable.js` file inside.

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

```
const stream = require('stream')
const rs = new stream.Readable()

rs._read = function () {
  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.

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

Let's add this to our `readable.js` file:

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

Now let's try running our program:

```
$ node readable.js
```

We should see the readable stream print out the `Hello, World!` message.

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.

Let's copy our `readable.js` file and call it `index.js`.

We need to remove the `data` handler, let's comment it out like so:

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

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

```
const ws = new stream.Writable()

ws._write = function (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!".

As you may have noticed, creating our own streams is a little bit cumbersome. We need to override methods and there is a lot of ways to implement poorly. For example 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 the same `_read` context things become tricky.

Here is an example:

```
// THIS EXAMPLE DOES NOT WORK AS EXPECTED
var rs = new stream.Readable()

rs._read = function () {
  setTimeout(function () {
    rs.push('Data 0')
    setTimeout(function () {
      rs.push('Data 1')
    }, 50)
  }, 100)
}

rs.on('data', function (data) {
  console.log(data.toString())
```

```
})
```

Try running this example. 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 `through2`) to make all of this easier.

Creating read and write streams

- from2
- to2
- discussion on on maybe just using through2

Composing duplex streams

- duplexify

See also

Decoupling I/O

Getting Ready

How to do it

How it works

There's more

See also

Streams in the browser

Getting Ready

How to do it

How it works

There's more

See also