3 Coordinating I/O

This chapter covers the following topics

- Interfacing with standard I/O
- Working with files
- Fetching meta-data
- Watching files and directories
- Communicating over sockets

Introduction

Operationally, Node.js is C/C++ with JavaScript's clothes on. Just like C and other low-level environments, Node interacts with the Operating System at a fundamental level: Input and Output.

In this chapter we'll explore some core API's provided by Node, along with a few third party utilities that allow us to interact with standard I/O, the file system, and the network stack.

Interfacing with standard I/O

Standard I/O relates to the predefined input, output and error data channels that connect a process to a shell terminal, commonly known as STDIN, STDOUT and STDERR. Of course these can be redirected and piped to other programs for further processing, storage and so on.

Node provides access to standard I/O on the global `process` object.

In this recipe we're going to take some input, use it to form some data which we'll send to STDOUT, while simultaneously logging to STDERR.

Getting Ready

Let's create a file called `base64.js`, we're going to write a tiny program that converts input into its base64 equivalent.

How to do it

First, we'll listen for a `data` event on `process.stdin`,

```
process.stdin.on('data', data => {

})
```

If we run our file now:

```
$ node base64.js
```

We should notice that the process does not exit, and allows keyboard input.

Now let's do something with the incoming data, we'll modify our code thusly:

```
process.stdin.on('data', data => {
  process.stderr.write(`Converting: "${data}" to base64\n`)
  process.stdout.write(data.toString('base64') + '\n')
})
```

We can test our program like so

```
$ echo -e "hi\nthere" | node base64.js
```

Which should output:

```
Converting: "hi
there
" to base64
aGkKdGhlcmUK
```

We can also simply run our program, and each line of input will be converted:

```
$ node base64.js
<keyboard input>
```

Of course if we want to filter out logs to standard error, we can do the following:

```
$ echo -e "hi\nthere" | node base64.js 2> /dev/null
```

Which outputs:

```
aGkKdGhlcmUK
```

How it works

The standard I/O channels are implemented using Node.js Streams.

We'll find out more about these in **Chapter 4 Using Streams**, we also have an example in the **There's more** section of this recipe of using the standard I/O channels as streams.

Suffice it to say, that Node `Stream` instances (instantiated from Node's core `stream` module) inherit from `EventEmitter` (from Node's core `events` module), and emit a `data` event for every chunk of data received.

When in interactive mode (that is, when inputting via the keyboard), each data chunk is determined by a newline. When piping data through to the process, each data chunk is determined by the maximum memory consumption allowable for the stream (this determined by the `highWaterMark`, but we'll learn more about this in Chapter 3).

We listen to the `data` event, which provides a `Buffer` object (also called `data`) holding a binary representation of the input.

We write to standard error first, simply by passing a string `process.stderr.write`. The string contains the `data` `Buffer` object, which is automatically converted to a string representation when interpolated into (or concatanated with) another string.

Next we write to standard output using `process.stdout.write`. `Buffer` objects have a `toString` method that can be passed an encoding option. We specify an encoding of `'base64'` when calling `toString` on the buffer object, to convert the input from raw binary data to a base64 string representation.

There's more

Let's take a look at how Node Streams wrap Standard I/O channels, and how to detect whether an I/O channel is connected directly to a terminal or not.

Piping

As mentioned in the main recipe, the standard I/O channels available on the global `process` object are implementations of a core Node abstraction: streams. We'll be covering these in much greater detail in **Chapter 4 Using Streams** but for now let's see how we could achieve an equivalent effect using Node Streams' `pipe` method.

For this example we need the third party `base64-encode-stream` module, so let's open a terminal and run the following commands:

```
$ mkdir piping
$ cd piping
$ npm init -y
$ npm install --save base64-encode-stream
```

We just created a folder, used `npm init` to create a `package.json` file for us, and then installed the `base64-encode-stream` dependency.

Now let's create a fresh `base64.js` file in the `piping` folder, and write the following:

```
const encode = require('base64-encode-stream')
process.stdin.pipe(encode()).pipe(process.stdout)
```

We can try out our code like so:

```
$ echo -e "hi\nthere" | node base64.js
aGkKdGhlcmUK
```

In this case, we didn't write to standard error, but we did set up a pipeline from standard input to standard output, transforming any data that passes through the pipeline.

TTY Detection

As a concept, Standard I/O is decoupled from terminals and devices. However, it can be useful to know whether a program is directly connected to a terminal or whether it's I/O is being redirected.

We can check with the `isTTY` flag on each I/O channel.

For instance, let's try the following command:

```
$ node -p "process.stdin.isTTY"
true
```

> ### The `-p` flag
>
> The `-p` flag will evaluate a supplied string and output the final result. There's also the related `-e` flag which only evaluates a supplied command line string, but doesn't output it's return value.

We're running `node` directly, so our STDIN is correctly identified as a TTY input.

Now let's try the following:

```
$ echo "hi" | node -p "process.stdin.isTTY"
undefined
```

This time `isTTY` is `undefined`, this is because the our program is executed inside a shell pipeline. It's important to note `isTTY` is `undefined` and not `false` as this can lead to bugs (for instance, when checking for a `false` value instead of a falsey result).

The `isTTY` flag is `undefined` instead of `false` in the second case because the standard I/O channels are internally initialised from different constructors depending on scenario. So when the process is directly connected to a terminal, `process.stdin` is created using the core `tty` modules `ReadStream` constructor, which has the `isTTY` flag. However, when I/O is redirected the channels are created from the `net` module's `Socket` constructor, which does not have an `isTTY` flag.

Knowing whether a process is directly connected to a terminal can be useful in certain cases - for instance when determining whether to output plain text or text decorated with ANSI Escape codes for colouring, boldness and so forth.

See also

- *Creating a Node.js WebSocket client* in the *There's More* section of

Communicating with WebSockets in **Chapter 5 Wielding Web Protocols**
- *Using the pipe method* in **Chapter 4 Using Streams**
- *Getting symlink information* in the *There's More* section of *Fetching meta-data* in this chapter

Working with files

The ability to read and manipulate the file system is fundamental to server side programming.

Node's `fs` module provides this ability.

In this recipe we'll learn how to read, write and append to files in a synchronous manner. In the **There's More** section we'll explore how to perform the same operations asynchronously and incrementally.

Getting Ready

We'll need a file to read.

We can use the following to populate a file with 1MB of data:

```
$ node -p "Buffer.allocUnsafe(1e6).toString()" > file.dat
```

> ### Allocating Buffers
>
> When the `Buffer` constructor is passed a number, the specified amount of bytes will be allocated from *deallocated memory*, data in RAM that was previously discarded. This means the buffer could contain anything. From Node v6 and above, passing a number to `Buffer` is deprecated. Instead we use `Buffer.allocUnsafe` to achieve the same effect, or just `Buffer.alloc` to have a zero-filled buffer (but at the cost of slower instantiation). To state the obvious, the file we generated for ourselves (`file.dat`) should not be shared with anyone else.

We'll also want to create a source file, let's call it `null-byte-remover.js`.

How to do it

We're going to write simple program that strips all null bytes (bytes with a value of zero) from our file, and saves it in a new file called `clean.dat`.

First we'll require our dependencies

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

Now let's load our generated file into the process:

```
const cwd = process.cwd()
const bytes = fs.readFileSync(path.join(cwd, 'file.dat'))
```

> ### Synchronous Operations
>
> Always think twice before using a synchronous API in JavaScript. JavaScript executes in a single threaded event-loop, a long-lasting synchronous operation will delay concurrent logic. We'll explore this more in the **There's more** section.

Next, we'll remove null bytes and save:

```
const clean = bytes.filter(n => n)
fs.writeFileSync(path.join(cwd, 'clean.dat'), clean)
```

Finally, let's append to a log file so we can keep a record:

```
fs.appendFileSync(
  path.join(cwd, 'log.txt'),
  (new Date) + ' ' + (bytes.length - clean.length) + ' bytes removed\n'
)
```

How it works

Both `fs` and `path` are core modules, so there's no need to install these dependencies.

The `path.join` method is a useful utility that normalizes paths across platforms, since Windows using back slashes (`\`) whilst others use forward slashes ('/') to denote path segments.

We use this three times, all with the `cwd` reference which we obtain by calling `process.cwd()` to fetch the current working directory.

`fs.readFileSync` will *synchronously* read the entire file into process memory.

This means that any queued logic is blocked until the entire file is read, thus ruining any capacity for concurrent operations (like, serving web requests).

That's why synchronous operations are usually explicit in Node Core (for instance, the `Sync` in `readFileSync`).

For our current purposes it doesn't matter, since we're interested only in processing a single set of sequential actions in series.

So we read the contents of `file.dat` using `fs.readFileSync` and assign the resulting buffer to `bytes`.

Next we remove the zero-bytes, using a `filter` method. By default `fs.readFileSync` returns a `Buffer` object which is a container for the binary data. `Buffer` objects inherit from native `UInt8Array` (part of the EcmaScript 6 specification), which in turn inherits from the native JavaScript `Array` constructor.

This means we can call functional methods like `filter` (or `map`, or `reduce`) on `Buffer` objects!

The `filter` method is passed a predicate function, which simply returns the value passed into it. If the value is 0 the byte will be removed from the bytes array because the number `0` is coerced to `false` in boolean checking contexts.

The filter bytes are assigned to `clean`, which is then written synchronously to a file named `clean.dat`, using `fs.writeFileSync`. Again, because the operation is synchronous nothing else can happen until the write is complete.

Finally, we use `fs.appendFileSync` to record the date and amount of bytes removed to a `log.txt` file. If the `log.txt` file doesn't exist it will be automatically created and written to.

There's more

Let's explore asynchronous I/O.

Asynchronous file operations

Suppose we wanted some sort of feedback, to show that the process was doing something.

We could use an interval to write a dot to `process.stdout` every 10 milliseconds.

If we add the following to top of the file:

```
setInterval(() => process.stdout.write('.'), 10).unref()
```

This should queue a function every 10 milliseconds that writes a dot to Standard Out.

The `unref` method may seem alien if we're used to using timers in the browser.

Browser timers (`setTimeout` and `setInterval`) return numbers, for ID's that can be passed into the relevant clear function. Node timers return objects, which also act as ID's in the same manner, but additionally have this handy `unref` method.

Simply put, the `unref` method prevents the timer from keeping the process alive.

When we run our code, we won't see any dots being written to the console.

This is because the synchronous operations all occur in the same tick of the event loop, then there's nothing else to do and the process exits. A much worse scenario is where a synchronous operation occurs in several ticks, and delays a timer (or an HTTP response).

Now that we want something to happen alongside our operations, we need to switch to using asynchronous file methods.

Let's rewrite out source file like so:

```
setInterval(() => process.stdout.write('.'), 10).unref()

const fs = require('fs')
const path = require('path')
const cwd = process.cwd()

fs.readFile(path.join(cwd, 'file.dat'), (err, bytes) => {
  if (err) { console.error(err); process.exit(1); }
  const clean = bytes.filter(n => n)
  fs.writeFile(path.join(cwd, 'clean.dat'), clean, (err) => {
```

```
    if (err) { console.error(err); process.exit(1); }
    fs.appendFile(
      path.join(cwd, 'log.txt'),
      (new Date) + ' ' + (bytes.length - clean.length) + ' bytes remove
    )
  })
})
```

Now when we run our file, we'll see a few dots printed to the console. Still not as many as expected - the process takes around 200ms to complete, there should be more than 2-3 dots! This is because the `filter` operation is unavoidably synchronous and quite intensive, it's delaying queued intervals.

Incremental Processing

How could we mitigate the intense byte-stripping operation from blocking other important concurrent logic?

Node's core abstraction for incremental asynchronous processing has already appeared in the previous recipe, but it's merits bear repetition.

Streams to the rescue!

We're going to convert our recipe once more, this time to using a streaming abstraction.

First we'll need the third-party `strip-bytes-stream` package:

```
$ npm init -y # create package.json if we don't have it
$ npm install --save strip-bytes-stream
```

Now let's alter our code like so:

```
setInterval(() => process.stdout.write('.'), 10).unref()

const fs = require('fs')
const path = require('path')
const cwd = process.cwd()

const sbs = require('strip-bytes-stream')

fs.createReadStream(path.join(cwd, 'file.dat'))
  .pipe(sbs((n) => n))
  .on('end', function () { log(this.total) })
  .pipe(fs.createWriteStream(path.join(cwd, 'clean.dat')))
```

```
function log(total) {
  fs.appendFile(
    path.join(cwd, 'log.txt'),
    (new Date) + ' ' + total + ' bytes removed\n'
  )
}
```

This time we should see around 15 dots, which over roughly 200ms of execution time is much fairer.

This is because the file is read into the process in chunks, each chunk is stripped of null bytes, and written to file, the old chunk and stripped result is discarded, whilst the next chunk enters process memory. This all happens over multiple ticks of the event loop, allowing room for processing of the interval timer queue.

We'll be delving much deeper into Streams in **Chapter 4 Using Streams**, but for the time being we can see that `fs.createReadStream` and `fs.createWriteStream` are, more often that not, the most suitable way to read and write to files.

See also

- *Receiving POST Data* in **Chapter 5 Wielding Web protocols**
- *Deploying a full system* in **Chapter 11 Deploying Node.js**
- *Multipart POST uploads* in **Chapter 5 Wielding Web protocols**
- *Processing big data* in **Chapter 4 Using Streams**
- *Creating an SMTP server* in **Chapter 5 Wielding Web protocols**
- *Fetching meta-data* in this chapter

Fetching meta-data

Reading directory listings, fetching permissions, getting time of creation and modification. These are all essential pieces of the file system tool kit.

> ### `fs` and POSIX
>
> Most of the `fs` module methods are light wrappers around POSIX operations (with shims for Windows), so many of the concepts and names should be similar if we've indulged in any system programming or even shell scripting.

In this recipe, we're going to write a CLI tool that supplies in depth information about files and directories for a given path.

Getting Ready

To get started, let's create a new folder called `fetching-meta-data`, containing a file called `meta.js`:

```
$ mkdir fetching-meta-data
$ cd fetching-meta-data
$ touch meta.js
```

Now let's use `npm` to create `package.json` file,

```
$ npm init -y
```

We're going to display tabulated and styled metadata in the terminal, instead of manually writing ANSI codes and tabulating code, we'll simply be using the third party `tableaux` module.

We can install it like so

```
$ npm install --save tableaux
```

Finally we'll create a folder structure that we can check our program against:

```
$ mkdir -p my-folder/my-subdir/my-subsubdir
$ cd my-folder
$ touch my-file my-private-file
$ chmod 000 my-private-file
$ echo "my edit" > my-file
$ ln -s my-file my-symlink
$ touch my-subdir/another-file
$ touch my-subdir/my-subsubdir/too-deep
```

How to do it

Let's open `meta.js`, and begin by loading the modules we'll be using

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

```
const path = require('path')
const tableaux = require('tableaux')
```

Next we'll initialize tableaux with some table headers, which in turn will supply a `write` function which we'll be using shortly:

```
const write = tableaux(
  {name: 'Name', size: 20},
  {name: 'Created', size: 30},
  {name: 'DeviceId', size: 10},
  {name: 'Mode', size: 8},
  {name: 'Lnks', size: 4},
  {name: 'Size', size:  6}
)
```

Now let's sketch out a `print` function

```
function print(dir) {
  fs.readdirSync(dir)
    .map((file) => ({file, dir}))
    .map(toMeta)
    .forEach(output)
  write.newline()
}
```

The `print` function wont work yet, not until we define the `toMeta` and `output` functions.

The `toMeta` function is going to take an object with a `file` property, `stat` the file in order to obtain information about it, then return that data, like so:

```
function toMeta({file, dir}) {
  const stats = fs.statSync(path.join(dir, file))
  var {birthtime, ino, mode, nlink, size} = stats
  birthtime = birthtime.toUTCString()
  mode = mode.toString(8)
  size += 'B'
  return {
    file,
    dir,
    info: [birthtime, ino, mode, nlink, size],
    isDir: stats.isDirectory()
  }
}
```

The `output` function is going to output the information supplied by `toMeta`, and in cases where a given entity is a directory, it will query and output a summary of the directory contents. Our `output` functions looks like the following:

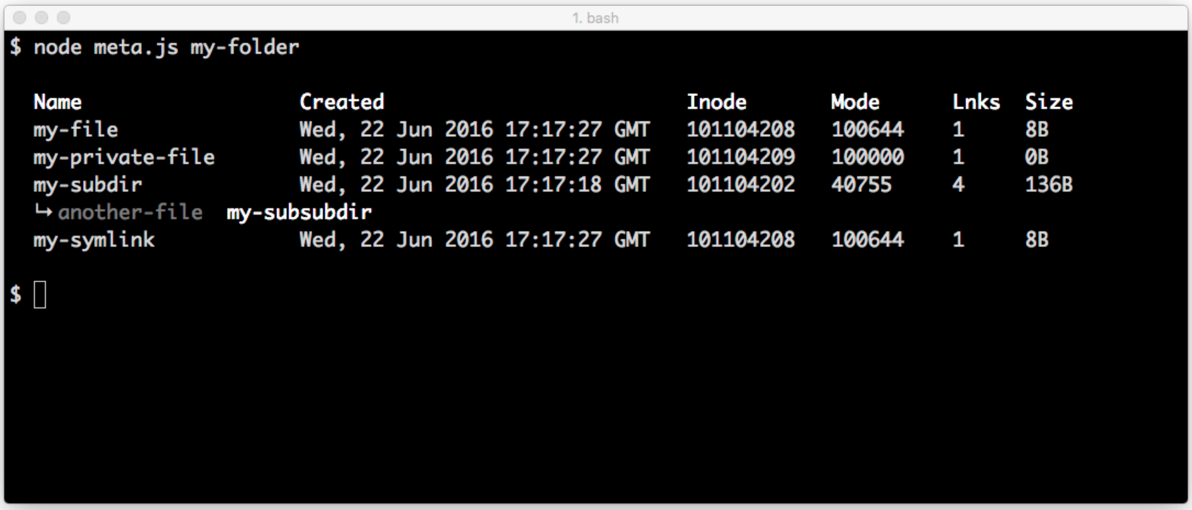
```
function output({file, dir, info, isDir}) {
  write(file, ...info)
  if (!isDir) { return }
  const p = path.join(dir, file)
  write.arrow()
  fs.readdirSync(p).forEach((f) => {
    const stats = fs.statSync(path.join(p, f))
    const style = stats.isDirectory() ? 'bold' : 'dim'
    write[style](f)
  })
  write.newline()
}
```

Finally we can call the `print` function:

```
print(process.argv[2] || '.')
```

Now let's run our program, assuming our current working directory is the `fetching-meta-data` folder, we should be able to successfully run the following:

```
$ node meta.js my-folder
```



Our program output should look similar to this

How it works

When we call `print` we pass in `process.argv[2]`, if it's value is falsey, then we alternatively pass a dot (meaning current working directory).

The `argv` property on `process` is an array of command line arguments, including the call to `node` (at `process.argv[0]`) and the file being executed (at `process.argv[1]`).

When we ran `node meta.js my-folder`, `process.argv[2]` had the value `'my-folder'`.

Our `print` function uses `fs.readdirSync` to get an array of all the files and folders in the specified `dir` (in our case, the `dir` was `'my-folder'`).

> ### Functional Programming
>
> We use a functional approach in this recipe (and mostly throughout this book) If this is unfamiliar, check out the functional programming workshop on http://nodeschool.io

We call `map` on the returned array to create a new array of objects containing both the `file` and the `dir` (we need to keep a reference to the `dir` so it can eventually be passed to the `output` function).

We call `map` again, on the previously mapped array of objects, this time passing the `toMeta` function as the transformer function.

The `toMeta` function uses the ES2015 destructuring syntax to accept an object and break its `file` and `dir` property into variables that are local to the function. Then `toMeta` passes the `file` and `dir` into `path.join` (to assemble a complete cross-platform to the file), which in turn is passed into `fs.statSync`. The `fs.statSync` method (and its asynchronous counter `fs.stat`) is a light wrapper around the POSIX `stat` operation.

It supplies an object with following information about a file or directory:

Information Point	Namespace
ID of device holding file	`dev`
Inode number	`ino`
Access Permissions	`mode`
Number of hard links contained	`nlink`
User ID	`uid`

Group ID	`gid`
Device ID of special device file	`rdev`
Total bytes	`size`
Filesystem block size	`blksize`
Number of 512byte blocks allocated for file	`blocks`
Time of last access	`atime`
Time of last modification	`mtime`
Time of last status change	`ctime`
Time of file creation	`birthtime`

We use assignment destructuring in out `toMeta` function to grab the `birthtime` , `ino` , `mode` , `nlink` and `size` values. We ensure `birthtime` is a standard UTC string (instead of local time), and convert the `mode` from a decimal to the more familiar octal permissions. representation.

The `stats` object supplies some methods:

- `isFile`
- `isDirectory`
- `isBlockDevice`
- `isCharacterDevice`
- `isFIFO`
- `isSocket`
- `isSymbolicLink`

> ### `isSymbolicLink`
>
> The `isSymbolicLink` method is only available when the file stats have been retrieved with `fs.lstat` (not with `fs.stat`), see the **There's More** section for an example where we use `fs.lstat` .

In the object returned from `toMeta` , we add an `isDir` property, the value of which is determined by the `stats.isDirectory()` call.

We also add the `file` and `dir` use ES2015 property shorthand , and an array of our selected stats on the `info` property.

> ### ES2015 Property Shorthand

> ES2015 (or ES6) defines a host of convenient syntax extensions for JavaScript, 96% of which is supported in Node v6. One such extension is property shorthand, where a variable can be placed in an object without specifying property name or value. Under the rules of property shorthand the property name corresponds to the variable name, the value to the value referenced by the variable.

Once the second `map` call in our `print` function has looped over every element, passing each to the `toMeta` function, we are left with a new array composed of objects as returned from `toMeta` - containing `file`, `dir`, `info`, and `isDir` properties.

The `forEach` method is called on this array of metadata, and it's passed the `output` function. This means that each piece of metadata is processed by the `output` function.

In similar form to the `toMeta` function, the `output` function likewise deconstructs the passed in object into `file`, `dir`, `info`, and `isDir` references. We then pass the `file` string and all of elements in the `info` array, using the ES2015 spread operator, to our `write` function (as supplied by the third party `tableaux` module).

If we're only dealing with a file, (that is, if `isDir` is not `true`), we exit the `output` function by returning early.

If, however, `isDir` is true then we call `write.arrow` (which writes a unicode arrow to the terminal), read the directory, call `forEach` on the returned array of directory contents, and call `fs.statSync` on each item in the directory.

We then check whether the item is a directory (that is, a subdirectory) using the returned stats object, if it is we write the directory name to the terminal in bold, if it isn't we write a in a dulled down white color.

There's more

Let's find out how to examine symlinks, check whether files exists and see how to actually alter file system metadata.

Getting symlink information

There are other types of stat calls, one such call is `lstat` (the 'l' stands for link).

When an `lstat` command comes across a symlink, it stats the symlink itself, rather than the file it points to.

Let's modify our `meta.js` file to recognize and resolve symbolic links.

First we'll modify the `toMeta` function to use `fs.lstatSync` instead of `fs.statSync`, and then add an `isSymLink` property to the returned object:

```
function toMeta({file, dir}) {
  const stats = fs.lstatSync(path.join(dir, file))
  var {birthtime, ino, mode, nlink, size} = stats
  birthtime = birthtime.toUTCString()
  mode = mode.toString(8)
  size += 'B'
  return {
    file,
    dir,
    info: [birthtime, ino, mode, nlink, size],
    isDir: stats.isDirectory(),
    isSymLink: stats.isSymbolicLink()
  }
}
```

Now let's add a new function, called `outputSymlink`:

```
function outputSymlink(file, dir, info) {
  write('\u001b[33m' + file + '\u001b[0m', ...info)
  process.stdout.write('\u001b[33m')
  write.arrow(4)
  write.bold(fs.readlinkSync(path.join(dir, file)))
  process.stdout.write('\u001b[0m')
  write.newline()
}
```

Our `outputSymlink` function using terminal ANSI escape codes to color the symlink name, arrow and file target, yellow.

Next in the `output` function, we'll check whether the file is a symbolic link, and delegate to `outputSymlink` if it is.

Additionally when we're querying subdirectories, we'll switch to `fs.lstatSync` so we can colour an symbolic links in the subdirectories a dim yellow as well.

```
function output({file, dir, info, isDir, isSymLink}) {
  if (isSymLink) {
```

```
    outputSymlink(file, dir, info)
    return
  }
  write(file, ...info)
  if (!isDir) { return }
  const p = path.join(dir, file)
  write.arrow()
  fs.readdirSync(p).forEach((f) => {
    const stats = fs.lstatSync(path.join(p, f))
    const style = stats.isDirectory() ? 'bold' : 'dim'
    if (stats.isSymbolicLink()) { f = '\u001b[33m' + f + '\u001b[0m'}
    write[style](f)
  })
  write.newline()
}
```

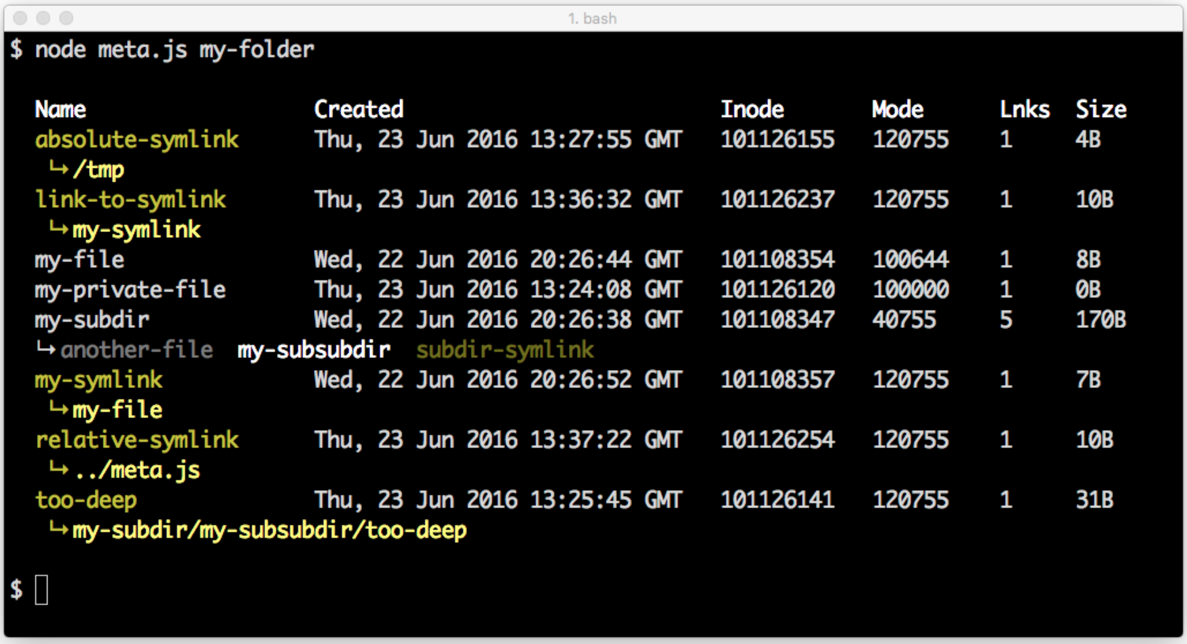
Now when we run

```
$ node meta.js my-folder
```

We should see the `my-symlink` file in a pretty yellow color.

Let's finish up by adding some extra symlinks and seeing how they render:

```
$ cd my-folder
$ ln -s /tmp absolute-symlink
$ ln -s my-symlink link-to-symlink
$ ln -s ../meta.js relative-symlink
$ ln -s my-subdir/my-subsubdir/too-deep too-deep
$ cd my-subdir
$ ln -s another-file subdir-symlink
$ cd ../..
$ node meta.js my-folder
```



Symlinks in glorious yellow

Checking file existence

A fairly common task in systems and server side programming, is checking whether a file exists or not.

There is a method, `fs.exists` (and its sync cousin) `fs.existsSync`, which allows us perform this very action. However, it has been deprecated since Node version 4, and therefore isn't future-safe.

A better practice is to use `fs.access` (added when `fs.exists` was deprecated).

By default `fs.access` checks purely for "file visibility", which is essentially the equivalent of checking for existence.

Let's write a file called `check.js`, we can pass it a file and it will tell us whether the file exists or not:

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

const exists = (file) => new Promise((resolve, reject) => {
  fs.access(file, (err) => {
    if (err) {
      if (err.code !== 'ENOENT') { return reject(err) }
      return resolve({file, exists: false})
    }
    resolve({file, exists: true})
  })
```

```
})

exists(process.argv[2])
  .then(({file, exists}) => console.log(`"${file}" does${exists ? '' :
  .catch(console.error)
```

> ### Promises
>
> For extra fun here (because the paradigm fits well in this case), we used the ES2015 native `Promise` abstraction. Find out more about promises at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise In general we tend to use a minimal subset of ES2015 (ES6) throughout so we can focus more on using Node and less on syntax, avoiding either extra discourse or potential confusion. We should also note Promises are currently a poor choice for implementing production server logic in Node, due to (standards specified) opaque behaviors (error swallowing, asynchronous stack unwinding) leading to difficulties in production root cause analysis.

Now if we run the following:

```
$ node check.js non-existent-file
"non-existent-file" does not exist
```

But if we run:

```
$ node check.js check.js
"check.js" does exist
```

The `fs.access` method is more versatile than `fs.exists`, it can be passed different modes (`fs.F_OK` (default), `fs.R_OK`, `fs.W_OK` and `fs.X_OK`) to alter access check being made. The mode is a number, and the constants of `fs` (ending in `_OK`) are numbers, allowing for a bitmask approach.

For instance, here's how we can check if have permissions to read, write and execute a file (this time with `fs.accessSync`):

```
fs.access('/usr/local/bin/node', fs.R_OK | fs.W_OK | fs.X_OK, console.l
```

If there's a problem accessing, an error will be logged, if not `null` will be logged.

> ### Modes and Bitmasks
>
> For more on `fs.access` see the docs at https://nodejs.org/api/fs.html#fs_fs_access_path_mode_callback, to learn about bitmasks check out https://abdulapopoola.com/2016/05/30/understanding-bit-masks/

Manipulating metadata

By now we have learned how to fetch information about a file or directory, but how do we alter specific qualities.

Let's create a small program that creates a file, sets the UID and GID to `nobody` and sets access permissions to `000` (not readable, writeable or executable).

```
const fs = require('fs')
const { execSync } = require('child_process')

const file = process.argv[2]
if (!file) {
  console.error('specify a file')
  process.exit(1)
}
try {
  fs.accessSync(file)
  console.error('file already exists')
  process.exit(1)
} catch (e) {
  makeIt()
}

function makeIt() {
  const nobody = Number(execSync('id -u nobody').toString().trim())
  fs.writeFileSync(file, '')
  fs.chownSync(file, nobody, nobody)
  fs.chmodSync(file, 0)
  console.log(file + ' created')
}
```

We used `fs.accessSync` to synchronously check for file existence, using a `try/catch` since `fs.accessSync` throws when a file does not exist.

> ### try/catch
>
> In this particular context, a try/catch is fine. However as a general rule we

> should avoid try/catch when possible. While Node 6 and above successfully handle the performance implications of try/catch there are other points to be aware of. See How to know when (not) to throw for more details.

If the file does not exist, we call our `makeIt` function.

This uses the `execSync` function from the `child_process` module to get the numerical ID of the `nobody` user on our system.

Next we use `fs.writeFileSync` to create an empty file, then use `fs.chownSync` to set the user and group to nobody, use `fs.chmodSync` to set the permissions to their minimum possible value, and finally log out a confirmation message.

We can improve our approach a little here. Each operation has to access the file separately, instead of retaining a reference to it throughout. We can make this a little more efficient by using file handles.

Let's rewrite out `makeIt` function like so:

```
function makeIt() {
  const nobody = Number(execSync('id -u nobody').toString().trim())
  const fd = fs.openSync(file, 'w')
  fs.fchmodSync(fd, 0)
  fs.fchownSync(fd, nobody, nobody)
  console.log(file + ' created')
}
```

This achieve the same result, but directly manages the file handle (an OS-level reference to the file).

We use `fs.openSync` to create the file and get a file descriptor (`fd`), then instead of `fs.chmodSync` and `fs.chownSync` both of which expect a file path, we use `fs.fchmodSync` and `fs.fchownSync` which take a file descriptor.

See also

- *Watching files and directories* in this chapter
- *Receiving POST Data* in **Chapter 5 Wielding Web protocols**

Watching files and directories

The ability to receive notifications when a file is added, removed or updated can be extremely useful. Node's `fs` module supplies this functionality cross-platform, however as we'll explore the functionality across operating systems can be patch.

In this recipe, we'll write a program that watches a file and outputs some data about the file when it changes. In the *There's More* section, we'll explore the limitation of Node's watch functionality along with a third-party module that wraps the core functionality to make it more consistent.

Getting Ready

Let's create a new folder called `watching-files-and-directories`, create a `package.json` in the folder and then install the third party `human-time` module for nicely formatted time outputs.

```
$ mkdir watching-files-and-directories
$ cd watching-files-and-directories
$ npm init -y
$ npm install --save human-time
```

We'll also create a file to watch:

```
$ echo "some content" > my-file.txt
```

Finally we want to create a file called `watcher.js` (inside the `watching-files-and-directories` folder) and open it in our favorite editor.

How to do it

Let's start by loading the dependencies we'll be needing:

```
const fs = require('fs')
const human = require('human-time')
```

Next we'll set up some references:

```
const interval = 5007
const file = process.argv[2]
var exists = false
```

Do a quick check to make sure we've been supplied a file:

```
if (!file) {
  console.error('supply a file')
  process.exit(1)
}
```

Now we'll set up some utility functions, which will help us interpret the file change event:

```
const created = ({birthtime})
  => !exists && (Date.now() - birthtime) < interval

const missing = ({birthtime, mtime, atime, ctime}) =>
  !(birthtime|mtime|atime|ctime)

const updated = (cur, prv) => cur.mtime !== prv.mtime
```

Finally we use `fs.watchFile` to being polling the specified file and then log out activity from the listener function supplied to `fs.watchFile`.

Like so:

```
fs.watchFile(file, {interval}, (cur, prv) => {
  if (missing(cur)) {
    const msg = exists ? 'removed' : 'doesn\'t exist'
    exists = false
    return console.log(`${file} ${msg}`)
  }

  if (created(cur)) {
    exists = true
    return console.log(`${file} created ${human((cur.birthtime))}`)
  }

  exists = true

  if (updated(cur, prv)) {
    return console.log(`${file} updated ${human((cur.mtime))}`)
  }

  console.log(`${file} modified ${human((cur.mtime))}`)
})
```

We should now be able to test our watcher.

In one terminal we can run:

```
$ node watcher my-file.txt
```

And in another we can make a change

```
$ echo "more content" >> my-file.txt
```

Or remove the file

```
$ rm my-file.txt
```

And recreate it

```
$ echo "back again" > my-file.txt
```

We should be seeing results similar to this

How it works

The `fs` module has two watch methods, `fs.watch` and `fs.watchFile`.

Whilst `fs.watch` is more responsive and can watch entire directories, recursively, it has various consistency and operational issues on different platforms (for instance, inability to report filenames on OS X, may report events twice, or not report them at all).

Instead of using an OS relevant notification subsystem (like `fs.watch`) the `fs.watchFile` function polls the file at a specified interval (defaulting to 5007 milliseconds).

The listener function (supplied as the last argument to `fs.watchFile`), is called every time the file is altered in some way. The listener takes two arguments. The first argument is a stats object (as provided by `fs.stat`, see Fetching MetaData) of the file it its current state, the second argument is a stats object of the file in its previous state.

We use these objects along with our three lambda functions, `created`, `missing` and `updated` to infer how the file has been altered.

The `created` function checks whether the `birthtime` (time of file creation) is less than the polling interval, then it's likely the file was created.

We introduce certainty by setting an `exists` variable and tracking the file existence in our listener function. So our `created` function check this variable first, if the file is known to exist then it can't have been created. This caters to situations where a file is updated multiple times within the polling interval period and ensures the first file alteration event is interpreted as a change, whilst subsequent triggers are not (unless the file was detected as removed).

When `fs.watchFile` attempts to poll a non-existent (or at least, inaccessible), file, it signals this eventuality by setting the `birthtime`, `mtime`, `atime` and `ctime` to zero (the Unix epoch). Our `missing` function checks for this by bitwise ORing all four dates, this implicitly converts the dates to numerical values and will result either in `0` or some other number (if any of the four values is non-zero). This in turn is converted to a boolean, if the result is `0` missing returns `true` else it returns false.

The `mtime` is the time since file data was last changed. Comparing the `mtime` of the file before and after the event allows us to differentiate between a change where the file content was updated, and a change where file metadata was altered.

The `updated` function compares the `mtime` on the previous and current stat objects, if they're not the same then the file content must have been changed, if they are the same then file was modified in some other way (for instance, a chmod).

Our listener function, checks these utility functions and then updates the `exists` variable and logs out messages accordingly.

There's more

The core watching functionality is often too basic, let's take a look at the third party alternative, `chokidar`

Watching directories with `chokidar`

The `fs.watchFile` method is slow, CPU intensive and only watches an individual file.

The `fs.watch` method is unreliable.

Enter chokidar. Chokidar wraps the core watching functionality to make it more reliable across platforms, more configurable and less CPU intensive. It also watches entire directories recursively.

Let's create a new watcher that watches a whole directory tree.

Let's make a new folder, 'watching-with-chokidar', with a subdirectory called `my-folder`, which in turn has another subfolder called `my-subfolder`

```
$ mkdir -p watching-with-chokidar/my-folder/my-subfolder
```

In our `watching-with-chokidar` folder we'll automatically create a new `package.json` and install dependencies with `npm`:

```
$ cd watching-with-chokidar
$ npm init -y
$ npm install --save chokidar human-time
```

Now let's create our new `watcher.js` file.

First we'll require the dependencies and create a chokidar `watcher` instance:

```
const chokidar = require('chokidar')
const human = require('human-time')
const watcher = chokidar.watch(process.argv[2] || '.', {
  alwaysStat: true
})
```

Now we'll listen for the `ready` event (meaning that chokidar has scanned directory contents), and then listen for various change events.

```
watcher.on('ready', () => {
  watcher
    .on('add', (file, stat) => {
      console.log(`${file} created ${human((stat.birthtime))}`)
```

```
    })
    .on('unlink', (file) => {
      console.log(`${file} removed`)
    })
    .on('change', (file, stat) => {
      const msg = (+stat.ctime === +stat.mtime) ? 'updated' : 'modified
      console.log(`${file} ${msg} ${human((stat.ctime))}`)
    })
    .on('addDir', (dir, stat) => {
      console.log(`${dir} folder created ${human((stat.birthtime))}`)
    })
    .on('unlinkDir', (dir) => {
      console.log(`${dir} folder removed`)
    })
})
```

Now we should be able to spin up our watcher, point it at `my-folder` and being making observable changes.

In one terminal we do:

```
$ node watcher.js my-folder
```

In another terminal:

```
cd my-folder
echo "me again" > my-file.txt
chmod 700 my-file.txt
echo "more" >> my-file.txt
rm my-file.txt
cd my-subfolder
echo "deep" > deep.txt
rm deep.txt
cd ..
rm -fr my-subfolder
mkdir my-subfolder
```

Watching a directory tree

See also

- *Fetching meta-data* in this chapter
- *Setting up a development environment* in **Chapter 10 Building Microservice systems**

Communicating over sockets

One way to look at a socket is as a special file. Like a file it's a readable and writable data container. On some Operating Systems network sockets are literally a special type of file whereas on others the implementation is more abstract.

At any rate, the concept of a socket has changed our lives because it allows machines to communicate, to co-ordinate I/O across a network. Sockets are the backbone of distributed computing.

In this recipe we'll build a TCP client and server.

Getting Ready

Let's create two files `client.js` and `server.js` and open them in our favorite editor.

How to do it

First, we'll create our server.

In `server.js`, let's write the following:

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

net.createServer((socket) => {
  console.log('-> client connected')
  socket.on('data', name => {
    socket.write(`Hi ${name}!`)
  })
  socket.on('close', () => {
    console.log('-> client disconnected')
  })
}).listen(1337, 'localhost')
```

Now for the client, our `client.js` should look like this:

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

const socket = net.connect(1337, 'localhost')
const name = process.argv[2] || 'Dave'

socket.write(name)

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

socket.on('close', () => {
  console.log('-> disconnected by server')
})
```

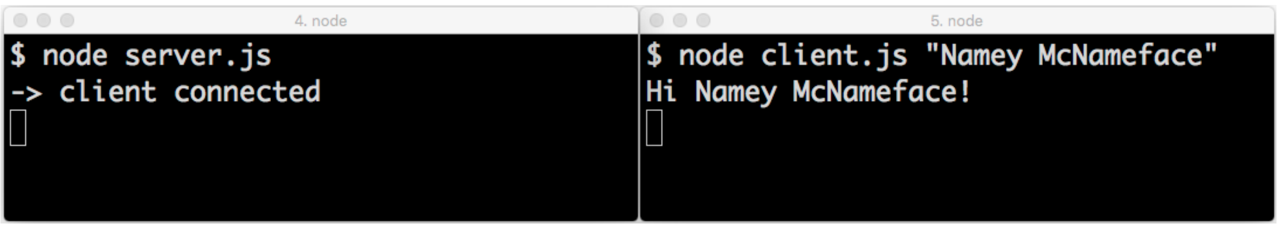
We should be able to start our server and connect to it with our client.

In one terminal:

```
$ node server.js
```

In another:

```
$ node client.js "Namey McNameface"
```



Client server interaction

Further if we kill the client with `Ctrl+C` the server will output:

```
-> client disconnected
```

But if we kill the server, the client will output:

```
-> disconnected by server
```

How it works

Our server uses `net.createServer` to instantiate a TCP server.

This returns an object with a `listen` method which is called with two arguments, `1337` and `localhost` which instructors our server to listen on port `1337` on the local loop network interface.

The `net.createServer` method is passed a connection handler function, which is called every time a new connection to the server is established.

This function receives a single argument, the `socket`.

We listen for a `data` event on the `socket`, and then send the data back to the client embedded inside a greeting message, by passing this greeting to the `socket.write` method.

We also listen for a `close` event, which will detect when the client closes the connection and log a message if it does.

Our client uses the `net.connect` method passing it the same port and hostname as defined in our server, which in turn returns a `socket`.

We immediately write the `name` to the socket, and attach a `data` listener in order to receive a response from the server. When we get a response we simply log it to the terminal, we have to call the `toString` method on incoming data because sockets deliver raw binary data in the form of Node buffers (this string conversion happens implicitly on our server when we embed the Buffer into the greeting string).

Finally our client also listens for a `close` event, which will trigger in cases where the server ends the connection.

There's more

Let's learn a little more about sockets, and the different types of sockets that are available.

`net` sockets are streams

Previous recipes in this chapter have alluded to streams, we'll be studying these in depth in **Chapter 4 Using Streams**.

However we would be remiss if we didn't mention that TCP sockets implement the streams interface.

In our main recipe, the `client.js` file contains the following code:

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

We can write this more succinctly like so:

```
socket.pipe(process.stdout)
```

Here we pipe from the socket to Standard out (see the first recipe of this chapter, *Interfacing with standard I/O*)

In fact sockets are both readable and writable (known as duplex streams).

We can even create an echo server in one line of code:

```
require('net').createServer((socket) => socket.pipe(socket)).listen(133
```

The readable interface pipes directly back to the writable interface so all incoming data is immediately written back out.

Likewise, we can create a client for our echo server in one line:

```
process.stdin.pipe(require('net').connect(1338)).pipe(process.stdout)
```

We pipe Standard input through a socket that's connected to our echo server and then pipe anything that comes through the socket to Standard out.

Unix Sockets

The `net` module also allows us to communicate across Unix sockets, these are special files that can be place on the file system.

All we have to do is listen on and connect to a file path instead of a port number and hostname.

In `client.js` we modify the following:

```
const socket = net.connect(1337, 'localhost')
```

To this:

```
const socket = net.connect('/tmp/my.socket')
```

The last line of `server.js` looks like so:

```
}).listen(1337, 'localhost')
```

We simply change it to the following:

```
}).listen('/tmp/my.socket')
```

Now our client and server can talk over a Unix socket instead of the network.

> **IPC**
>
> Unix sockets are primarily useful for low level IPC (Inter Process Communication), however for general IPC needs the `child_process` module supplies a more convenient high level abstraction.

UDP Sockets

Whilst TCP is a protocol built for reliability, UDP is minimalistic and more suited to use cases where speed is more important than consistency (for instance gaming or media streaming).

Node supplies UDP via the `dgram` module (UDP stands for User Datagram Protocol).

Let's reimplement our recipe with UDP.

First we'll rewrite `client.js`:

```
const dgram = require('dgram')

const socket = dgram.createSocket('udp4')
const name = process.argv[2] || 'Dave'
```

```
socket.bind(1400)
socket.send(name, 1339)

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

Notice that we're no longer listening for a `close` event, this is because it's now pointless to do so because our server (as we'll see) is incapable of closing the client connection.

Let's implement the `server.js` file:

```
const dgram = require('dgram')

const socket = dgram.createSocket('udp4')
socket.bind(1339)

socket.on('message', (name) => {
  socket.send(`Hi ${name}!`, 1400)
})
```

Now, the server looks much more like a client than server.

This is because there's no real concept of server-client architecture with UDP - that's implemented by the TCP layer.

There is only sockets, that bind to a specific port and listen.

We cannot bind two processes to the same port, so to get similar functionality we actually have to bind to two ports. There is a way to have multiple processes bind to the same port (using the `reuseAddr` option), but then we would have to deal with both processes receiving the same packets. Again, this is something TCP usually deals with.

Our client binds to port `1400`, and sends a message to port `1399`, whereas our server binds to port `1339` (so it can receive the clients message) but sends a message to port `1400` (which the client will receive).

Notice we use a `send` method instead of a `write` method as in the main recipe. The `write` method is part of the streams API, UDP sockets are not streams (the paradigm doesn't fit because they're not reliable nor persistent).

Likewise, we no longer listen for a `data` event, but a `message` event. Again the `data` event belongs to the streams API, whereas `message` is part of the `dgram` module.

We'll notice that the server (like the client) no longer listens for a `close` event, this is because the sockets are bound to different ports so there's not way (without a higher level protocol like TCP) of triggering a close from the other side.

See also

- *Interfacing with standard I/O* in this chapter
- *Setting up a development environment* in **Chapter 10 Building Microservice systems**
- *Using the pipe method* in **Chapter 4 Using Streams**
- *Decoupling I/O* in **Chapter 4 Using Streams**
- *Pattern Routing* in the *There's More* section of *Standardizing service boilerplate* in **Chapter 10 Building Microservice Systems**