Chapter 16. Server-Side JavaScript with Node

Node is JavaScript with bindings to the underlying operating system, making it possible to write JavaScript programs that read and write files, execute child processes, and communicate over the network. This makes Node useful as a:

- Modern alternative to shell scripts that does not suffer from the arcane syntax of bash and other Unix shells.
- General-purpose programming language for running trusted programs, not subject to the security constraints imposed by web browsers on untrusted code.
- Popular environment for writing efficient and highly concurrent web servers.

The defining feature of Node is its single-threaded event-based concurrency enabled by an asynchronous-by-default API. If you have programmed in other languages but have not done much JavaScript coding, or if you're an experienced client-side JavaScript programmer used to writing code for web browers, using Node will be a bit of an adjustment, as is any new programming language or environment. This chapter begins by explaining the Node programming model, with an emphasis on concurrency, Node's API for working with streaming data, and Node's Buffer type for working with binary data. These initial sections are followed by sections that highlight and demonstrate some of the most important Node APIs, including those for working with files, networks, processes, and threads.

One chapter is not enough to document all of Node's APIs, but my hope is that this chapter will explain enough of the fundamentals to make you productive with Node, and confident that you can master any new APIs you need.

INSTALLING NODE

Node is open source software. Visit *https://nodejs.org* to download and install Node for Windows and MacOS. On Linux, you may be able to install Node with your normal package manager, or you can visit *https://nodejs.org/en/download* to download the binaries directly. If you work on containerized software, you can find official Node Docker images at *https://hub.docker.com*.

In addition to the Node executable, a Node installation also includes npm, a package manager that enables easy access to a vast ecosystem of JavaScript tools and libraries. The examples in this chapter will use only Node's built-in packages and will not require npm or any external libraries.

Finally, do not overlook the official Node documentation, available at *https://nodejs.org/api* and *https://nodejs.org/docs/guides*. I have found it to be well organized and well written.

16.1 Node Programming Basics

We'll begin this chapter with a quick look at how Node programs are structured and how they interact with the operating system.

16.1.1 Console Output

If you are used to JavaScript programming for web browsers, one of the minor surprises about Node is that `console.log()` is not just for debugging, but is Node's easiest way to display a message to the user, or, more generally, to send output to the stdout stream. Here's the classic "Hello World" program in Node:

```
console.log("Hello World!");
```

There are lower-level ways to write to stdout, but no fancier or more official way than simply calling `console.log()`.

In web browsers, `console.log()`, `console.warn()`, and `console.error()` typically display little icons next to their output in the developer console to indicate the variety of the log message. Node does not do this, but output displayed with `console.error()` is distinguished from output displayed with `console.log()` because `console.error()` writes to the stderr stream. If you're using Node to write a program that is designed to have stdout redirected to a file or a pipe, you can use `console.error()` to display text to the console where the user will see it, even though text printed with `console.log()` is hidden.

16.1.2 Command-Line Arguments and Environment Variables

If you have previously written Unix-style programs designed to be invoked from a terminal or other command-line interface, you know that these programs typically get their input primarily from command-line arguments and secondarily from environment variables.

Node follows these Unix conventions. A Node program can read its command-line arguments from the array of strings `process.argv`. The first element of this array is always the path to the Node executable. The second argument is the path to the file of JavaScript code that Node is executing. Any remaining elements in this array are the space-separated arguments that you passed on the command-line when you invoked Node.

For example, suppose you save this very short Node program to the file *argv.js*:

```
console.log(process.argv);
```

You can then execute the program and see output like this:

```
$ node --trace-uncaught argv.js --arg1 --arg2 filename
[
  '/usr/local/bin/node',
  '/private/tmp/argv.js',
  '--arg1',
  '--arg2',
  'filename'
]
```

There are a couple of things to note here:

- The first and second elements of `process.argv` will be fully qualified filesystem paths to the Node executable and the file of JavaScript that is being executed, even if you did not type them that way.
- Command-line arguments that are intended for and interpreted by the Node executable itself are consumed by the Node executable and do not appear in `process.argv`. (The `--trace-uncaught` command-line argument isn't actually doing anything useful in the previous example; it is just there to demonstrate that it does not appear in the output.) Any arguments (such as `--arg1` and `filename`) that appear after the name of the JavaScript file will appear in `process.argv`.

Node programs can also take input from Unix-style environment variables. Node makes these available though the `process.env` object. The property names of this object are environment variable names, and the property values (always strings) are the values of those variables.

Here is a partial list of environment variables on my system:

```
$ node -p -e 'process.env'
{
  SHELL: '/bin/bash',
  USER: 'david',
  PATH: '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',
  PWD: '/tmp',
  LANG: 'en_US.UTF-8',
  HOME: '/Users/david',
}
```

You can use `node -h` or `node --help` to find out what the `-p` and `-e` command-line arguments do. However, as a hint, note that you could rewrite the line above as `node --eval 'process.env' --print`.

16.1.3 Program Life Cycle

The `node` command expects a command-line argument that specifies the file of JavaScript code to be run. This initial file typically imports other modules of JavaScript code, and may also define its own classes and functions. Fundamentally, however, Node executes the JavaScript code in the specified file from top to bottom. Some Node programs exit when they are done executing the last line of code in the file. Often, however, a Node program will keep running long after the initial file has been executed. As we'll discuss in the following sections, Node programs are often asynchronous and based on callbacks and event handlers. Node programs do not exit until they are done running the initial file and until all callbacks have been called and there are no more pending events. A Node-based server program that listens for incoming network connections will theoretically run forever because it will always be waiting for more events.

A program can force itself to exit by calling `process.exit()`. Users can usually terminate a Node program by typing Ctrl-C in the terminal window where the program is running. A program can ignore Ctrl-C by registering a signal handler function with `process.on("SIGINT", ()=>{})`.

If code in your program throws an exception and no `catch` clause catches it, the program will print a stack trace and exit. Because of Node's asynchronous nature, exceptions that occur in callbacks or event handlers must be handled locally or not handled at all, which means that handling exceptions that occur in the asynchronous parts of your program can be a difficult problem. If you don't want these exceptions to cause your program to completely crash, register a global handler function that will be invoked instead of crashing:

```
process.setUncaughtExceptionCaptureCallback(e => {
    console.error("Uncaught exception:", e);
});
```

A similar situation arises if a Promise created by your program is rejected and there is no `.catch()` invocation to handle it. As of Node 13, this is not a fatal error that causes your program to exit, but it does print a verbose error message to the console. In some future version of Node, unhandled Promise rejections are expected to become fatal errors. If you do not want unhandled rejections, to print error messages or terminate your program, register a global handler function:

```
process.on("unhandledRejection", (reason, promise) => {
    // reason is whatever value would have been passed to a .catch() functio
    // promise is the Promise object that rejected
});
```

16.1.4 Node Modules

Chapter 10 documented JavaScript module systems, covering both Node modules and ES6 modules. Because Node was created before JavaScript had a module system, Node had to create its own. Node's module system uses the `require()` function to import values into a module and the `exports` object or the `module.exports` property to export values from a module. These are a fundamental part of the Node programming model, and they are covered in detail in §10.2.

Node 13 adds support for standard ES6 modules as well as require-based modules (which Node calls "CommonJS modules"). The two module systems are not fully compatible, so this is somewhat tricky to do. Node needs to know—before it loads a module—whether that module will be using `require()` and `module.exports` or if it will be using `import` and `export`. When Node loads a file of JavaScript code as a CommonJS module, it automatically defines the `require()` function along with identifiers `exports` and `module`, and it does not enable the `import` and `export` keywords. On the other hand, when Node loads a file of code as an ES6 module, it must enable the `import` and `export` declarations, and it must *not* define extra identifiers like `require`, `module`, and `exports`.

The simplest way to tell Node what kind of module it is loading is to encode this information in the file extension. If you save your JavaScript code in a file that ends with *.mjs*, then Node will always load it as an ES6 module, will expect it to use `import` and `export`, and will not provide a `require()` function. And if you save your code in a file that ends with *.cjs*, then Node will always treat it as a CommonJS module, will provide a `require()` function, and will throw a SyntaxError if you use `import` or `export` declarations.

For files that do not have an explicit *.mjs* or *.cjs* extension, Node looks for a file named *package.json* in the same directory as the file and then in each of the containing directories. Once the nearest *package.json* file is found, Node checks for a top-level `type` property in the JSON object. If the value of the `type` property is "module", then Node loads the file as an ES6 module. If the value of that property is "commonjs", then Node loads the file as a CommonJS module. Note that you do not need to have a *package.json* file to run Node programs: when no such file is found (or when the file is found but it does not have a `type` property), Node defaults to using CommonJS modules. This *package.json* trick only becomes necessary if you want to use ES6 modules with Node and do not want to use the *.mjs* file extension.

Because there is an enormous amount of existing Node code written using CommonJS module format, Node allows ES6 modules to load CommonJS modules using the `import` keyword. The reverse is not true, however: a CommonJS module cannot use `require()` to load an ES6 module.

16.1.5 The Node Package Manager

When you install Node, you typically get a program named npm as well. This is the Node Package Manager, and it helps you download and manage libraries that your program depends on. npm keeps track of those dependencies (as well as other information about your program) in a file named *package.json* in the root directory of your project. This *package.json* file created by npm is where you would add `"type":"module"` if you wanted to use ES6 modules for your project.

This chapter does not cover npm in any detail (but see §17.4 for a little more depth). I'm mentioning it here because unless you write programs that do not use any external libraries, you will almost certainly be using npm or a tool like it. Suppose, for example, that you are going to be developing a web server and plan to use the Express framework (*https://expressjs.com*) to simplify the task. To get started, you might create a directory for your project, and then, in that directory type `npm init`. npm will ask you for your project name, version number, etc., and will then create an initial *package.json* file based on your responses.

Now to start using Express, you'd type `npm install express`. This tells npm to download the Express library along with all of its dependencies and install all the packages in a local *node_modules/* directory:

```
$ npm install express
npm notice created a lockfile as package-lock.json. You should commit this f
npm WARN my-server@1.0.0 No description
npm WARN my-server@1.0.0 No repository field.

+ express@4.17.1
added 50 packages from 37 contributors and audited 126 packages in 3.058s
found 0 vulnerabilities
```

When you install a package with npm, npm records this dependency—that your project depends on Express—in the *package.json* file. With this dependency recorded in *package.json*, you could give another programmer a copy of your code and your *package.json*, and they could simply type `npm install` to automatically download and install all of the libraries that your program needs in order to run.

16.2 Node Is Asynchronous by Default

JavaScript is a general-purpose programming language, so it is perfectly possible to write CPU-intensive programs that multiply large matrices or perform complicated statistical analyses. But Node was designed and optimized for programs—like network servers—that are I/O intensive. And in particular, Node was designed to make it possible to easily implement highly concurrent servers that can handle many requests at the same time.

Unlike many programming languages, however, Node does not achieve concurrency with threads. Multithreaded programming is notoriously hard to do correctly, and difficult to debug. Also, threads are a relatively heavyweight abstraction and if you want to write a server that can handle hundreds of concurrent requests, using hundreds of threads may require a prohibitive amount of memory. So Node adopts the single-threaded JavaScript programming model that the web uses, and this turns out to be a vast simplification that makes the creation of network servers a routine skill rather than an arcane one.

TRUE PARALLELISM WITH NODE

Node programs can run multiple operating system processes, and Node 10 and later support Worker objects (§16.11), which are a kind of thread borrowed from web browsers. If you use multiple processes or create one or more Worker threads and run your program on a system with more than one CPU, then your program will no longer be single-threaded and your program will truly be executing multiple streams of code in parallel. These techniques can be valuable for CPU-intensive operations but are not commonly used for I/O-intensive programs like servers.

It is worth noting, however, that Node's processes and Workers avoid the typical complexity of multithreaded programming because interprocess and inter-Worker communication is via message passing and they cannot easily share memory with each other.

Node achieves high levels of concurrency while maintaining a single-threaded programming model by making its API asynchronous and non-blocking by default. Node takes its nonblocking approach very seriously and to an extreme that may surprise you. You probably expect functions that read from and write to the network to be asynchronous, but Node goes further and defines nonblocking asynchronous functions for reading and writing files from the local filesystem. This makes sense, when you think about it: the Node API was designed in the days when spinning hard drives were still the norm and there really were milliseconds of blocking "seek time" while waiting for the disc to spin around before a file operation could begin. And in modern datacenters, the "local" filesystem may actually be across the network somewhere with network latencies on top of drive latencies. But even if reading a file asynchronously seems normal to you, Node takes it still further: the default functions for initiating a network connection or looking up a file modification time, for example, are also nonblocking.

Some functions in Node's API are synchronous but nonblocking: they run to completion and return without ever needing to block. But most of the interesting functions perform some kind of input or output, and these are asynchronous functions so they can avoid even the tiniest amount of blocking. Node was created before JavaScript had a Promise class, so asynchronous Node APIs are callback-based. (If you have not yet read or have already forgotten Chapter 13, this would be a good time to skip back to that chapter.) Generally, the last argument you pass to an asynchronous Node function is a callback. Node uses *error-first callbacks*, which are typically invoked with two arguments. The first argument to an error-first callback is normally `null` in the case where no error occurred, and the second argument is whatever data or response was produced by the original asynchronous function you called. The reason for putting the error argument first is to make it impossible for you to omit it, and you should always check for a non-null value in this argument. If it is an Error object, or even an integer error code or string error message, then something went wrong. In this case, the second argument to your callback function is likely to be `null`.

The following code demonstrates how to use the nonblocking `readFile()` function to read a configuration file, parse it as JSON, and then pass the parsed configuration object to another callback:

```
const fs = require("fs");  // Require the filesystem module
```

```
// Read a config file, parse its contents as JSON, and pass the
// resulting value to the callback. If anything goes wrong,
// print an error message to stderr and invoke the callback with null
function readConfigFile(path, callback) {
    fs.readFile(path, "utf8", (err, text) => {
        if (err) {    // Something went wrong reading the file
            console.error(err);
            callback(null);
            return;
        }
        let data = null;
        try {
            data = JSON.parse(text);
        } catch(e) {  // Something went wrong parsing the file contents
            console.error(e);
        }
        callback(data);
    });
}
```

Node predates standardized promises, but because it is fairly consistent about its error-first callbacks, it is easy to create Promise-based variants of its callback-based APIs using the `util.promisify()` wrapper. Here's how we could rewrite the `readConfigFile()` function to return a Promise:

```
const util = require("util");
const fs = require("fs");   // Require the filesystem module
const pfs = {               // Promise-based variants of some fs functions
    readFile: util.promisify(fs.readFile)
};

function readConfigFile(path) {
    return pfs.readFile(path, "utf-8").then(text => {
        return JSON.parse(text);
    });
}
```

We can also simpify the preceding Promise-based function using `async` and `await` (again, if you have not yet read through Chapter 13, this would be a good time to do so):

```
async function readConfigFile(path) {
    let text = await pfs.readFile(path, "utf-8");
```

```
        return JSON.parse(text);
    }
```

The `util.promisify()` wrapper can produce a Promise-based version of many Node functions. In Node 10 and later, the `fs.promises` object has a number of predefined Promise-based functions for working with the filesystem. We'll discuss them later in this chapter, but note that in the preceding code, we could replace `pfs.readFile()` with `fs.promises.readFile()`.

We had said that Node's programming model is async-by-default. But for programmer convenience, Node does define blocking, synchronous variants of many of its functions, especially in the filesystem module. These functions typically have names that are clearly labeled with `Sync` at the end.

When a server is first starting up and is reading its configuration files, it is not handling network requests yet, and little or no concurrency is actually possible. So in this situation, there is really no need to avoid blocking, and we can safely use blocking functions like `fs.readFileSync()`. We can drop the `async` and `await` from this code and write a purely synchronous version of our `readConfigFile()` function. Instead of invoking a callback or returning a Promise, this function simply returns the parsed JSON value or throws an exception:

```
const fs = require("fs");
function readConfigFileSync(path) {
    let text = fs.readFileSync(path, "utf-8");
    return JSON.parse(text);
}
```

In addition to its error-first two-argument callbacks, Node also has a number of APIs that use event-based asynchrony, typically for handling streaming data. We'll cover Node events in more detail later.

Now that we've discussed Node's aggressively nonblocking API, let's turn back to the topic of concurrency. Node's built-in nonblocking functions work using the operating system's version of callbacks and event handlers. When you call one of these functions, Node takes action to get the operation started, then registers some kind of event handler with the operating system so that it will be notified when the operation is complete. The callback you passed to the Node function gets stored internally so that Node can invoke your callback when the operating system sends the appropriate event to Node.

This kind of concurrency is often called event-based concurrency. At its core, Node has a single thread that runs an "event loop." When a Node program starts, it runs whatever code you've told it to run. This code presumably calls at least one nonblocking function causing a callback or event handler to be registered with the operating system. (If not, then you've written a synchronous Node program, and Node simply exits when it reaches the end.) When Node reaches the end of your program, it blocks until an event happens, at which time the OS starts it running again. Node maps the OS event to the JavaScript callback you registered and then invokes that function. Your callback function may invoke more nonblocking Node functions, causing more OS event handlers to be registered. Once your callback function is done running, Node goes back to sleep again and the cycle repeats.

For web servers and other I/O-intensive applications that spend most of their time waiting for input and output, this style of event-based concurrency is efficient and effective. A web server can concurrently handle requests from 50 different clients without needing 50 different threads as long as it uses nonblocking APIs and there is some kind of internal mapping from network sockets to JavaScript functions to invoke when activity occurs on those sockets.

16.3 Buffers

One of the datatypes you're likely to use frequently in Node—especially when reading data from files or from the network—is the Buffer class. A Buffer is a lot like a string, except that it is a sequence of bytes instead of a sequence of characters. Node was created before core JavaScript supported typed arrays (see §11.2) and there was no Uint8Array to represent an array of unsigned bytes. Node defined the Buffer class to fill that need. Now that Uint8Array is part of the JavaScript language, Node's Buffer class is a subclass of Uint8Array.

What distinguishes Buffer from its Uint8Array superclass is that it is designed to interoperate with JavaScript strings: the bytes in a buffer can be initialized from character strings or converted to character strings. A character encoding maps each character in some set of characters to an integer. Given a string of text and a character encoding, we can *encode* the characters in the string into a sequence of bytes. And given a (properly encoded) sequence of bytes and a character encoding, we can *decode* those bytes into a sequence of characters. Node's Buffer class has methods that perform both encoding and decoding, and you can recognize these methods because they expect an `encoding` argument that specifies the encoding to be used.

Encodings in Node are specified by name, as strings. The supported encodings are:

`"utf8"`

> This is the default when no encoding is specified, and is the Unicode encoding you are most likely to use.

`"utf16le"`

> Two-byte Unicode characters, with little-endian ordering. Codepoints above `\uffff` are encoded as a pair of two-byte sequences. Encoding `"ucs2"` is an alias.

`"latin1"`

> The one-byte-per-character ISO-8859-1 encoding that defines a character set suitable for many Western European languages. Because there is a one-to-one mapping between bytes and latin-1 characters, this encoding is also known as `"binary"`.

`"ascii"`

> The 7-bit English-only ASCII encoding, a strict subset of the `"utf8"` encoding.

`"hex"`

> This encoding converts each byte to a pair of ASCII hexadecimal digits.

`"base64"`

> This encoding converts each sequence of three bytes into a sequence of four ascii characters.

Here is some example code that demonstrates how to work with Buffers and how to convert to and from strings:

```
let b = Buffer.from([0x41, 0x42, 0x43]);           // <Buffer 41 42 43>
b.toString()                                       // => "ABC"; default "utf8
b.toString("hex")                                  // => "414243"

let computer = Buffer.from("IBM3111", "ascii");    // Convert string to Buffe
for(let i = 0; i < computer.length; i++) {         // Use Buffer as byte arra
    computer[i]--;                                 // Buffers are mutable
}
computer.toString("ascii")                         // => "HAL2000"
computer.subarray(0,3).map(x=>x+1).toString()      // => "IBM"

// Create new "empty" buffers with Buffer.alloc()
let zeros = Buffer.alloc(1024);                    // 1024 zeros
let ones = Buffer.alloc(128, 1);                   // 128 ones
let dead = Buffer.alloc(1024, "DEADBEEF", "hex"); // Repeating pattern of by

// Buffers have methods for reading and writing multi-byte values
// from and to a buffer at any specified offset.
dead.readUInt32BE(0)       // => 0xDEADBEEF
dead.readUInt32BE(1)       // => 0xADBEEFDE
dead.readBigUInt64BE(6)    // => 0xBEEFDEADBEEFDEADn
dead.readUInt32LE(1020)    // => 0xEFBEADDE
```

If you write a Node program that actually manipulates binary data, you may find yourself using the Buffer class extensively. On the other hand, if you are just working with text that is read from or written to a file or the network, then you may only encounter Buffer as an intermediate representation of your data. A number of Node APIs can take input or return output as either strings or Buffer objects. Typically, if you pass a string, or expect a string to be returned, from one of these APIs, you'll need to specify the name of the text encoding you want to use. And if you do this, then you may not need to use a Buffer object at all.

16.4 Events and EventEmitter

As described, all of Node's APIs are asynchronous by default. For many of them, this asynchrony takes the form of two-argument error-first callbacks that are invoked when the requested operation is complete. But some of the more complicated APIs are event-based instead. This is typically the case when the API is designed around an object rather than a function, or when a callback function needs to be invoked multiple times, or when there are multiple types of callback functions that may be required. Consider the `net.Server` class, for example: an object of this type is a server socket that is used to accept incoming connections from clients. It emits a "listening" event when it first starts listening for connections, a "connection" event every time a client connects, and a "close" event when it has been closed and is no longer listening.

In Node, objects that emit events are instances of EventEmitter or a subclass of EventEmitter:

```
const EventEmitter = require("events"); // Module name does not match class
const net = require("net");
let server = new net.Server();          // create a Server object
server instanceof EventEmitter          // => true: Servers are EventEmitter
```

The main feature of EventEmitters is that they allow you to register event handler functions with the `on()` method. EventEmitters can emit multiple types of events, and event types are identified by name. To register an event handler, call the `on()` method, passing the name of the event type and the function that should be invoked when an event of that type occurs. EventEmitters can invoke handler functions with any number of arguments, and you need to read the documentation for a specific kind of event from a specific EventEmitter to know what arguments you should expect to be passed:

```
const net = require("net");
let server = new net.Server();          // create a Server object
server.on("connection", socket => {     // Listen for "connection" events
    // Server "connection" events are passed a socket object
    // for the client that just connected. Here we send some data
    // to the client and disconnect.
    socket.end("Hello World", "utf8");
});
```

If you prefer more explicit method names for registering event listeners, you can also use `addListener()`. And you can remove a previously registered event listener with `off()` or `removeListener()`. As a special case, you can register an event listener that will be automatically removed after it is triggered for the first time by calling `once()` instead of `on()`.

When an event of a particular type occurs for a particular EventEmitter object, Node invokes all of the handler functions that are currently registered on that EventEmitter for events of that type. They are invoked in order from the first registered to the last registered. If there is more than one handler function, they are invoked sequentially on a single thread: there is no parallelism in Node, remember. And, importantly, event handling functions are invoked synchronously, not asynchronously. What this means is that the `emit()` method does not queue up event handlers to be invoked at some later time. `emit()` invokes all the registered handlers, one after the other, and does not return until the last event handler has returned.

What this means, in effect, is that when one of the built-in Node APIs emits an event, that API is basically blocking on your event handlers. If you write an event handler that calls a blocking function like `fs.readFileSync()`, no further event handling will happen until your synchronous file read is complete. If your program is one—like a network server—that needs to be responsive, then it is important that you keep your event handler functions nonblocking and fast. If you need to do a lot of computation when an event occurs, it is often best to use the handler to schedule that computation asynchronously using `setTimeout()` (see §11.10). Node also defines `setImmediate()`, which schedules a function to be invoked immediately after all pending callbacks and events have been handled.

The EventEmitter class also defines an `emit()` method that causes the registered event handler functions to be invoked. This is useful if you are defining your own event-based API, but is not commonly used when you're just programming with existing APIs. `emit()` must be invoked with the name of the event type as its first argument. Any additional arguments that are passed to `emit()` become arguments to the registered event handler functions. The handler functions are also invoked with the `this` value set to the EventEmitter object itself, which is often convenient. (Remember, though, that arrow functions always use the `this` value of the context in which they are defined, and they cannot be invoked with any other `this` value. Nevertheless, arrow functions are often the most convenient way to write event handlers.)

Any value returned by an event handler function is ignored. If an event handler function throws an exception, however, it propagates out from the `emit()` call and prevents the execution of any handler functions that were registered after the one that threw the exception.

Recall that Node's callback-based APIs use error-first callbacks, and it is important that you always check the first callback argument to see if an error occurred. With event-based APIs, the equivalent is “error” events. Since event-based APIs are often used for networking and other forms of streaming I/O, they are vulnerable to unpredictable asynchronous errors, and most EventEmitters define an “error” event that they emit when an error occurs. Whenever you use an event-based API, you should make it a habit to register a handler for “error” events. “Error” events get special treatment by the EventEmitter class. If `emit()` is called to emit an “error” event, and if there are no handlers registered for that event type, then an exception will be thrown. Since this occurs asynchronously, there is no way for you to handle the exception in a `catch` block, so this kind of error typically causes your program to exit.

16.5 Streams

When implementing an algorithm to process data, it is almost always easiest to read all the data into memory, do the processing, and then write the data out. For example, you could write a Node function to copy a file like this.[1]

```
const fs = require("fs");

// An asynchronous but nonstreaming (and therefore inefficient) function.
function copyFile(sourceFilename, destinationFilename, callback) {
    fs.readFile(sourceFilename, (err, buffer) => {
        if (err) {
            callback(err);
        } else {
            fs.writeFile(destinationFilename, buffer, callback);
        }
    });
}
```

This `copyFile()` function uses asynchronous functions and callbacks, so it does not block and is suitable for use in concurrent programs like servers. But notice that it must allocate enough memory to hold the entire contents of the file in memory at once. This may be fine in some use cases, but it starts to fail if the files to be copied are very large, or if your program is highly concurrent and there may be many files being copied at the same time. Another shortcoming of this `copyFile()` implementation is that it cannot start writing the new file until it has finished reading the old file.

The solution to these problems is to use streaming algorithms where data “flows” into your program, is processed, and then flows out of your program. The idea is that your algorithm processes the data in small chunks and the full dataset is never held in memory at once. When streaming solutions are possible, they are more memory efficient and can also be faster. Node’s networking APIs are stream-based and Node’s filesystem module defines streaming APIs for reading and writing files, so you are likely to use a streaming API in many of the Node programs that you write. We’ll see a streaming version of the `copyFile()` function in “Flowing mode”.

Node supports four basic stream types:

Readable

Readable streams are sources of data. The stream returned by `fs.createReadStream()`, for example, is a stream from which the content of a specified file can be read. `process.stdin` is another Readable stream that returns data from standard input.

Writable

Writable streams are sinks or destinations for data. The return value of `fs.createWriteStream()`, for example, is a Writable stream: it allows data to be written to it in chunks, and outputs all of that data to a specified file.

Duplex

Duplex streams combine a Readable stream and a Writable stream into one object. The Socket objects returned by `net.connect()` and other Node networking APIs, for example, are Duplex streams. If you write to a socket, your data is sent across the network to whatever computer the socket is connected to. And if you read from a socket, you access the data written by that other computer.

Transform

Transform streams are also readable and writable, but they differ from Duplex streams in an important way: data written to a Transform stream becomes readable—usually in some transformed form—from the same stream. The `zlib.createGzip()` function, for example, returns a Transform stream that compresses (with the *gzip* algorithm) the data written to it. In a similar way, the `crypto.createCipheriv()` function returns a Transform stream that encrypts or decrypts data that is written to it.

By default, streams read and write buffers. If you call the `setEncoding()` method of a Readable stream, it will return decoded strings to you instead of Buffer objects. And if you write a string to a Writable buffer, it will be automatically encoded using the buffer's default encoding or whatever encoding you specify. Node's stream API also supports an "object mode" where streams read and write objects more complex than buffers and strings. None of Node's core APIs use this object mode, but you may encounter it in other libraries.

Readable streams have to read their data from somewhere, and Writable streams have to write their data to somewhere, so every stream has two ends: an input and an output or a source and a destination. The tricky thing about stream-based APIs is that the two ends of the stream will almost always flow at different speeds. Perhaps the code that reads from a stream wants to read and process data more quickly than the data is actually being written into the stream. Or the reverse: perhaps data is written to a stream more quickly than it can be read and pulled out of the stream on the other end. Stream implementations almost always include an internal buffer to hold data that has been written but not yet read. Buffering helps to ensure that there is data available to read when it's requested, and that there is space to hold data when it is written. But neither of these things can ever be guaranteed, and it is the nature of stream-based programming that readers will sometimes have to wait for data to be written (because the stream buffer is empty), and writers will sometimes have to wait for data to be read (because the stream buffer is full).

In programming environments that use thread-based concurrency, stream APIs typically have blocking calls: a call to read data does not return until data arrives in the stream and a call to write data blocks until there is enough room in the stream's internal buffer to accommodate the new data. With an event-based concurrency model, however, blocking calls do not make sense, and Node's stream APIs are event- and callback-based. Unlike other Node APIs, there are not "Sync" versions of the methods that will be described later in this chapter.

The need to coordinate stream readability (buffer not empty) and writability (buffer not full) via events makes Node's stream APIs somewhat complicated. This is compounded by the fact that these APIs have evolved and changed over the years: for Readable streams, there are two completely distinct APIs that you can use. Despite the complexity, it is worth understanding and mastering Node's streaming APIs because they enable high-throughput I/O in your programs.

The subsections that follow demonstrate how to read and write from Node's stream classes.

16.5.1 Pipes

Sometimes, you need to read data from a stream simply to turn around and write that same data to another stream. Imagine, for example, that you are writing a simple HTTP server that serves a directory of static files. In this case, you will need to read data from a file input stream and write it out to a network socket. But instead of writing your own code to handle the reading and writing, you can instead simply connect the two sockets together as a "pipe" and let Node handle the complexities for you. Simply pass the Writable stream to the `pipe()` method of the Readable stream:

```
const fs = require("fs");

function pipeFileToSocket(filename, socket) {
    fs.createReadStream(filename).pipe(socket);
}
```

The following utility function pipes one stream to another and invokes a callback when done or when an error occurs:

```
function pipe(readable, writable, callback) {
    // First, set up error handling
    function handleError(err) {
        readable.close();
        writable.close();
        callback(err);
    }

    // Next define the pipe and handle the normal termination case
    readable
        .on("error", handleError)
        .pipe(writable)
        .on("error", handleError)
        .on("finish", callback);
}
```

Transform streams are particularly useful with pipes, and create pipelines that involve more than two streams. Here's an example function that compresses a file:

```
const fs = require("fs");
const zlib = require("zlib");

function gzip(filename, callback) {
    // Create the streams
    let source = fs.createReadStream(filename);
    let destination = fs.createWriteStream(filename + ".gz");
    let gzipper = zlib.createGzip();

    // Set up the pipeline
    source
        .on("error", callback)   // call callback on read error
        .pipe(gzipper)
        .pipe(destination)
        .on("error", callback)   // call callback on write error
        .on("finish", callback); // call callback when writing is complete
}
```

Using the `pipe()` method to copy data from a Readable stream to a Writable stream is easy, but in practice, you often need to process the data somehow as it streams through your program. One way to do this is to implement your own Transform stream to do that processing, and this approach allows you to avoid manually reading and writing the streams. Here, for example, is a function that works like the Unix `grep` utility: it reads lines of text from an input stream, but writes only the lines that match a specified regular expression:

```
const stream = require("stream");

class GrepStream extends stream.Transform {
    constructor(pattern) {
        super({decodeStrings: false});// Don't convert strings back to buffe
        this.pattern = pattern;       // The regular expression we want to m
        this.incompleteLine = "";     // Any remnant of the last chunk of da
    }

    // This method is invoked when there is a string ready to be
    // transformed. It should pass transformed data to the specified
    // callback function. We expect string input so this stream should
    // only be connected to readable streams that have had
    // setEncoding() called on them.
    _transform(chunk, encoding, callback) {
        if (typeof chunk !== "string") {
            callback(new Error("Expected a string but got a buffer"));
            return;
        }
        // Add the chunk to any previously incomplete line and break
        // everything into lines
        let lines = (this.incompleteLine + chunk).split("\n");

        // The last element of the array is the new incomplete line
        this.incompleteLine = lines.pop();

        // Find all matching lines
        let output = lines                        // Start with all complete li
            .filter(l => this.pattern.test(l)) // filter them for matches,
            .join("\n");                       // and join them back up.

        // If anything matched, add a final newline
        if (output) {
            output += "\n";
        }

        // Always call the callback even if there is no output
        callback(null, output);
    }

    // This is called right before the stream is closed.
    // It is our chance to write out any last data.
    _flush(callback) {
        // If we still have an incomplete line, and it matches
        // pass it to the callback
        if (this.pattern.test(this.incompleteLine)) {
            callback(null, this.incompleteLine + "\n");
        }
    }
}

// Now we can write a program like 'grep' with this class.
let pattern = new RegExp(process.argv[2]); // Get a RegExp from command line
process.stdin                              // Start with standard input,
    .setEncoding("utf8")                   // read it as Unicode strings,
    .pipe(new GrepStream(pattern))         // pipe it to our GrepStream,
    .pipe(process.stdout)                  // and pipe that to standard out.
    .on("error", () => process.exit());    // Exit gracefully if stdout clos
```

16.5.2 Asynchronous Iteration

In Node 12 and later, Readable streams are asynchronous iterators, which means that within an `async` function you can use a `for/await` loop to read string or Buffer chunks from a stream using code that is structured like synchronous code would be. (See §13.4 for more on asynchronous iterators and `for/await` loops.)

Using an asynchronous iterator is almost as easy as using the `pipe()` method, and is probably easier when you need to process each chunk you read in some way. Here's how we could rewrite the `grep` program in the previous section using an `async` function and a `for/await` loop:

```
// Read lines of text from the source stream, and write any lines
// that match the specified pattern to the destination stream.
async function grep(source, destination, pattern, encoding="utf8") {
    // Set up the source stream for reading strings, not Buffers
    source.setEncoding(encoding);

    // Set an error handler on the destination stream in case standard
    // output closes unexpectedly (when piping output to `head`, e.g.)
    destination.on("error", err => process.exit());

    // The chunks we read are unlikely to end with a newline, so each will
    // probably have a partial line at the end. Track that here
    let incompleteLine = "";

    // Use a for/await loop to asynchronously read chunks from the input str
    for await (let chunk of source) {
        // Split the end of the last chunk plus this one into lines
        let lines = (incompleteLine + chunk).split("\n");
        // The last line is incomplete
        incompleteLine = lines.pop();
        // Now loop through the lines and write any matches to the destinati
        for(let line of lines) {
            if (pattern.test(line)) {
                destination.write(line + "\n", encoding);
            }
        }
    }
    // Finally, check for a match on any trailing text.
    if (pattern.test(incompleteLine)) {
        destination.write(incompleteLine + "\n", encoding);
    }
}

let pattern = new RegExp(process.argv[2]);   // Get a RegExp from command li
grep(process.stdin, process.stdout, pattern) // Call the async grep() functi
```

```
        .catch(err => {                              // Handle asynchronous exceptio
            console.error(err);
            process.exit();
        });
```

16.5.3 Writing to Streams and Handling Backpressure

The async `grep()` function in the preceding code example demonstrated how to use a Readable stream as an asynchronous iterator, but it also demonstrated that you can write data to a Writable stream simply by passing it to the `write()` method. The `write()` method takes a buffer or string as the first argument. (Object streams expect other kinds of objects, but are beyond the scope of this chapter.) If you pass a buffer, the bytes of that buffer will be written directly. If you pass a string, it will be encoded to a buffer of bytes before being written. Writable streams have a default encoding that is used when you pass a string as the only argument to `write()`. The default encoding is typically "utf8," but you can set it explicitly by calling `setDefaultEncoding()` on the Writable stream. Alternatively, when you pass a string as the first argument to `write()` you can pass an encoding name as the second argument.

`write()` optionally takes a callback function as its third argument. This will be invoked when the data has actually been written and is no longer in the Writable stream's internal buffer. (This callback may also be invoked if an error occurs, but this is not guaranteed. You should register an "error" event handler on the Writable stream to detect errors.)

The `write()` method has a very important return value. When you call `write()` on a stream, it will always accept and buffer the chunk of data you have passed. It then returns `true` if the internal buffer is not yet full. Or, if the buffer is now full or overfull, it returns `false`. This return value is advisory, and you can ignore it—Writable streams will enlarge their internal buffer as much as needed if you keep calling `write()`. But remember that the reason to use a streaming API in the first place is to avoid the cost of keeping lots of data in memory at once.

A return value of `false` from the `write()` method is a form of *backpressure*: a message from the stream that you have written data more quickly than it can be handled. The proper response to this kind of backpressure is to stop calling `write()` until the stream emits a "drain" event, signaling that there is once again room in the buffer. Here, for example, is a function that writes to a stream, and then invokes a callback when it is OK to write more data to the stream:

```
function write(stream, chunk, callback) {
    // Write the specified chunk to the specified stream
    let hasMoreRoom = stream.write(chunk);

    // Check the return value of the write() method:
    if (hasMoreRoom) {                  // If it returned true, then
        setImmediate(callback);         // invoke callback asynchronously.
    } else {                            // If it returned false, then
        stream.once("drain", callback); // invoke callback on drain event.
    }
}
```

The fact that it is sometimes OK to call `write()` multiple times in a row and sometimes you have to wait for an event between writes makes for awkward algorithms. This is one of the reasons that using the `pipe()` method is so appealing: when you use `pipe()`, Node handles backpressure for you automatically.

If you are using `await` and `async` in your program, and are treating Readable streams as asynchronous iterators, it is straightforward to implement a Promise-based version of the `write()` utility function above to properly handle backpressure. In the async `grep()` function we just looked at, we did not handle backpressure. The async `copy()` function in the following example demonstrates how it can be done correctly. Note that this function just copies chunks from a source stream to a destination stream and calling `copy(source, destination)` is much like calling `source.pipe(destination)`:

```
// This function writes the specified chunk to the specified stream and
// returns a Promise that will be fulfilled when it is OK to write again.
// Because it returns a Promise, it can be used with await.
function write(stream, chunk) {
    // Write the specified chunk to the specified stream
    let hasMoreRoom = stream.write(chunk);

    if (hasMoreRoom) {                      // If buffer is not full, return
        return Promise.resolve(null);       // an already resolved Promise ob
    } else {
        return new Promise(resolve => {     // Otherwise, return a Promise th
            stream.once("drain", resolve); // resolves on the drain event.
        });
    }
}

// Copy data from the source stream to the destination stream
// respecting backpressure from the destination stream.
// This is much like calling source.pipe(destination).
async function copy(source, destination) {
    // Set an error handler on the destination stream in case standard
    // output closes unexpectedly (when piping output to `head`, e.g.)
    destination.on("error", err => process.exit());

    // Use a for/await loop to asynchronously read chunks from the input str
    for await (let chunk of source) {
        // Write the chunk and wait until there is more room in the buffer.
        await write(destination, chunk);
    }
}

// Copy standard input to standard output
copy(process.stdin, process.stdout);
```

Before we conclude this discussion of writing to streams, note again that failing to respond to backpressure can cause your program to use more memory than it should when the internal buffer of a Writable stream overflows and grows larger and larger. If you are writing a network server, this can be a remotely exploitable security issue. Suppose you write an HTTP server that delivers files over the network, but you didn't use `pipe()` and you didn't take the time to handle backpressure from the `write()` method. An attacker could write an HTTP client that initiates requests for large files (such as images) but never actually reads the body of the request. Since the client is not reading the data over the network, and the server isn't responding to backpressure, buffers on the server are going to overflow. With enough concurrent connections from the attacker, this can turn into a denial-of-service attack that slows your server down or even crashes it.

16.5.4 Reading Streams with Events

Node's readable streams have two modes, each of which has its own API for reading. If you can't use pipes or asynchronous iteration in your program, you will need to pick one of these two event-based APIs for handling streams. It is important that you use only one or the other and do not mix the two APIs.

Flowing mode

In *flowing mode*, when readable data arrives, it is immediately emitted in the form of a "data" event. To read from a stream in this mode, simply register an event handler for "data" events, and the stream will push chunks of data (buffers or strings) to you as soon as they becomes available. Note that there is no need to call the `read()` method in flowing mode: you only need to handle "data" events. Note that newly created streams do not start off in flowing mode. Registering a "data" event handler switches a stream into flowing mode. Conveniently, this means that a stream does not emit "data" events until you register the first "data" event handler.

If you are using flowing mode to read data from a Readable stream, process it, then write it to a Writable stream, then you may need to handle backpressure from the Writable stream. If the `write()` method returns `false` to indicate that the write buffer is full, you can call `pause()` on the Readable stream to temporarily stop `data` events. Then, when you get a "drain" event from the Writable stream, you can call `resume()` on the Readable stream to start the "data" events flowing again.

A stream in flowing mode emits an "end" event when the end of the stream is reached. This event indicates that no more "data" events will ever be emitted. And, as with all streams, an "error" event is emitted if an error occurs.

At the beginning of this section on streams, we showed a nonstreaming `copyFile()` function and promised a better version to come. The following code shows how to implement a streaming `copyFile()` function that uses the flowing mode API and handles backpressure. This would have been easier to implement with a `pipe()` call, but it serves here as a useful demonstration of the multiple event handlers that are used to coordinate data flow from one stream to the other.

```
const fs = require("fs");

// A streaming file copy function, using "flowing mode".
// Copies the contents of the named source file to the named destination fil
// On success, invokes the callback with a null argument. On error,
// invokes the callback with an Error object.
function copyFile(sourceFilename, destinationFilename, callback) {
    let input = fs.createReadStream(sourceFilename);
```

```
        let output = fs.createWriteStream(destinationFilename);

        input.on("data", (chunk) => {          // When we get new data,
            let hasRoom = output.write(chunk); // write it to the output stream.
            if (!hasRoom) {                    // If the output stream is full
                input.pause();                 // then pause the input stream.
            }
        });
        input.on("end", () => {                // When we reach the end of input
            output.end();                      // tell the output stream to end.
        });
        input.on("error", err => {             // If we get an error on the inpu
            callback(err);                     // call the callback with the err
            process.exit();                    // and quit.
        });

        output.on("drain", () => {             // When the output is no longer f
            input.resume();                    // resume data events on the inpu
        });
        output.on("error", err => {            // If we get an error on the outp
            callback(err);                     // call the callback with the err
            process.exit();                    // and quit.
        });
        output.on("finish", () => {            // When output is fully written
            callback(null);                    // call the callback with no erro
        });
    }

    // Here's a simple command-line utility to copy files
    let from = process.argv[2], to = process.argv[3];
    console.log(`Copying file ${from} to ${to}...`);
    copyFile(from, to, err => {
        if (err) {
            console.error(err);
        } else {
            console.log("done.");
        }
    });
```

Paused mode

The other mode for Readable streams is “paused mode.” This is the mode that streams start in. If you never register a “data” event handler and never call the `pipe()` method, then a Readable stream remains in paused mode. In paused mode, the stream does not push data to you in the form of “data” events. Instead, you pull data from the stream by explicitly calling its `read()` method. This is not a blocking call, and if there is no data available to read on the stream, it will return `null`. Since there is not a synchronous API to wait for data, the paused mode API is also event-based. A Readable stream in paused mode emits "readable" events when data becomes available to read on the stream. In response, your code should call the `read()` method to read that data. You must do this in a loop, calling `read()` repeatedly until it returns `null`. It is necessary to completely drain the stream's buffer like this in order to trigger a new "readable" event in the future. If you stop calling `read()` while there is still readable data, you will not get another "readable" event and your program is likely to hang.

Streams in paused mode emit "end" and "error" events just like flowing mode streams do. If you are writing a program that reads data from a Readable stream and writes it to a Writable stream, then paused mode may not be a good choice. In order to properly handle backpressure, you only want to read when the input stream is readable and the output stream is not backed up. In paused mode, that means reading and writing until `read()` returns `null` or `write()` returns `false`, and then starting reading or writing again on a `readable` or `drain` event. This is inelegant, and you may find that flowing mode (or pipes) is easier in this case.

The following code demonstrates how you can compute a SHA256 hash for the contents of a specified file. It uses a Readable stream in paused mode to read the contents of a file in chunks, then passes each chunk to the object that computes the hash. (Note that in Node 12 and later, it would be simpler to write this function using a `for/await` loop.)

```
const fs = require("fs");
const crypto = require("crypto");

// Compute a sha256 hash of the contents of the named file and pass the
// hash (as a string) to the specified error-first callback function.
function sha256(filename, callback) {
    let input = fs.createReadStream(filename); // The data stream.
    let hasher = crypto.createHash("sha256");  // For computing the hash.

    input.on("readable", () => {          // When there is data ready to read
        let chunk;
        while(chunk = input.read()) {     // Read a chunk, and if non-null,
            hasher.update(chunk);         // pass it to the hasher,
        }                                 // and keep looping until not readal
    });
    input.on("end", () => {                 // At the end of the stream,
        let hash = hasher.digest("hex"); // compute the hash,
        callback(null, hash);            // and pass it to the callback.
    });
    input.on("error", callback);          // On error, call callback
}

// Here's a simple command-line utility to compute the hash of a file
sha256(process.argv[2], (err, hash) => { // Pass filename from command line.
    if (err) {                              // If we get an error
        console.error(err.toString());   // print it as an error.
    } else {                                // Otherwise,
        console.log(hash);               // print the hash string.
    }
});
```

16.6 Process, CPU, and Operating System Details

The global Process object has a number of useful properties and functions that generally relate to the state of the currently running Node process. Consult the Node documentation for complete details, but here are some properties and functions you should be aware of:

```
process.argv            // An array of command-line arguments.
process.arch            // The CPU architecture: "x64", for example.
process.cwd()           // Returns the current working directory.
process.chdir()         // Sets the current working directory.
process.cpuUsage()      // Reports CPU usage.
process.env             // An object of environment variables.
process.execPath        // The absolute filesystem path to the node executab
process.exit()          // Terminates the program.
process.exitCode        // An integer code to be reported when the program e
process.getuid()        // Return the Unix user id of the current user.
process.hrtime.bigint() // Return a "high-resolution" nanosecond timestamp.
process.kill()          // Send a signal to another process.
process.memoryUsage()   // Return an object with memory usage details.
process.nextTick()      // Like setImmediate(), invoke a function soon.
process.pid             // The process id of the current process.
process.ppid            // The parent process id.
process.platform        // The OS: "linux", "darwin", or "win32", for exampl
process.resourceUsage() // Return an object with resource usage details.
process.setuid()        // Sets the current user, by id or name.
process.title            // The process name that appears in `ps` listings.
process.umask()          // Set or return the default permissions for new fil
process.uptime()         // Return Node's uptime in seconds.
process.version          // Node's version string.
process.versions         // Version strings for the libraries Node depends on
```

The “os” module (which, unlike `process`, needs to be explicitly loaded with `require()`) provides access to similarly low-level details about the computer and operating system that Node is running on. You may never need to use any of these features, but it is worth knowing that Node makes them available:

```
const os = require("os");
os.arch()               // Returns CPU architecture. "x64" or "arm", for exam
os.constants            // Useful constants such as os.constants.signals.SIGI
os.cpus()               // Data about system CPU cores, including usage times
os.endianness()         // The CPU's native endianness "BE" or "LE".
os.EOL                  // The OS native line terminator: "\n" or "\r\n".
os.freemem()            // Returns the amount of free RAM in bytes.
os.getPriority()        // Returns the OS scheduling priority of a process.
os.homedir()            // Returns the current user's home directory.
os.hostname()           // Returns the hostname of the computer.
os.loadavg()            // Returns the 1, 5, and 15-minute load averages.
os.networkInterfaces()  // Returns details about available network. connectio
os.platform()           // Returns OS: "linux", "darwin", or "win32", for exa
os.release()            // Returns the version number of the OS.
os.setPriority()        // Attempts to set the scheduling priority for a proc
os.tmpdir()             // Returns the default temporary directory.
os.totalmem()           // Returns the total amount of RAM in bytes.
os.type()               // Returns OS: "Linux", "Darwin", or "Windows_NT", e.
os.uptime()             // Returns the system uptime in seconds.
os.userInfo()           // Returns uid, username, home, and shell of current
```

16.7 Working with Files

Node’s “fs” module is a comprehensive API for working with files and directories. It is complemented by the “path” module, which defines utility functions for working with file and directory names. The “fs” module contains a handful of high-level functions for easily reading, writing, and copying files. But most of the functions in the module are low-level JavaScript bindings to Unix system calls (and their equivalents on Windows). If you have worked with low-level filesystem calls before (in C or other languages), then the Node API will be familiar to you. If not, you may find parts of the “fs” API to be terse and unintuitive. The function to delete a file, for example, is called `unlink()`.

The “fs” module defines a large API, mainly because there are usually multiple variants of each fundamental operation. As discussed at the beginning of the chapter, most functions such as `fs.readFile()` are nonblocking, callback-based, and asynchronous. Typically, though, each of these functions has a synchronous blocking variant, such as `fs.readFileSync()`. In Node 10 and later, many of these functions also have a Promise-based asynchronous variant such as `fs.promises.readFile()`. Most “fs” functions take a string as their first argument, specifying the path (filename plus optional directory names) to the file that is to be operated on. But a number of these functions also support a variant that takes an integer “file descriptor” as the first argument instead of a path. These variants have names that begin with the letter “f.” For example, `fs.truncate()` truncates a file specified by path, and `fs.ftruncate()` truncates a file specified by file descriptor. There is a Promise-based `fs.promises.truncate()` that expects a path and another Promise-based version that is implemented as a method of a FileHandle object. (The FileHandle class is the equivalent of a file descriptor in the Promise-based API.) Finally, there are a handful of functions in the “fs” module that have variants whose names are prefixed with the letter “l.” These “l” variants are like the base function but do not follow symbolic links in the filesystem and instead operate directly on the symbolic links themselves.

16.7.1 Paths, File Descriptors, and FileHandles

In order to use the “fs” module to work with files, you first need to be able to name the file you want to work with. Files are most often specified by *path*, which means the name of the file itself, plus the hierarchy of directories in which the file appears. If a path is *absolute*, it means that directories all the way up to the filesystem root are specified. Otherwise, the path is *relative* and is only meaningful in relation to some other path, usually the *current working directory*. Working with paths can be a little tricky because different operating systems use different characters to separate directory names, it is easy to accidentally double those separator characters when concatenating paths, and because `../` parent directory path segments need special handling. Node’s “path” module and a couple of other important Node features help:

```
// Some important paths
process.cwd()      // Absolute path of the current working directory.
__filename         // Absolute path of the file that holds the current code.
__dirname          // Absolute path of the directory that holds __filename.
os.homedir()       // The user's home directory.

const path = require("path");

path.sep                          // Either "/" or "\" depending on your OS

// The path module has simple parsing functions
let p = "src/pkg/test.js";        // An example path
path.basename(p)                  // => "test.js"
path.extname(p)                   // => ".js"
path.dirname(p)                   // => "src/pkg"
path.basename(path.dirname(p))    // => "pkg"
path.dirname(path.dirname(p))     // => "src"

// normalize() cleans up paths:
path.normalize("a/b/c/../d/")     // => "a/b/d/": handles ../ segments
path.normalize("a/./b")           // => "a/b": strips "./" segments
path.normalize("//a//b//")        // => "/a/b/": removes duplicate /

// join() combines path segments, adding separators, then normalizes
path.join("src", "pkg", "t.js")   // => "src/pkg/t.js"

// resolve() takes one or more path segments and returns an absolute
// path. It starts with the last argument and works backward, stopping
// when it has built an absolute path or resolving against process.cwd().
path.resolve()                    // => process.cwd()
path.resolve("t.js")              // => path.join(process.cwd(), "t.js")
path.resolve("/tmp", "t.js")      // => "/tmp/t.js"
path.resolve("/a", "/b", "t.js") // => "/b/t.js"
```

Note that `path.normalize()` is simply a string manipulation function that has no access to the actual filesystem. The `fs.realpath()` and `fs.realpathSync()` functions perform filesystem-aware canonicalization: they resolve symbolic links and interpret relative pathnames relative to the current working directory.

In the previous examples, we assumed that the code is running on a Unix-based OS and `path.sep` is “/.” If you want to work with Unix-style paths even when on a Windows system, then use `path.posix` instead of `path`. And conversely, if you want to work with Windows paths even when on a Unix system, `path.win32`. `path.posix` and `path.win32` define the same properties and functions as `path` itself.

Some of the "fs" functions we'll be covering in the next sections expect a *file descriptor* instead of a file name. File descriptors are integers used as OS-level references to "open" files. You obtain a descriptor for a given name by calling the `fs.open()` (or `fs.openSync()`) function. Processes are only allowed to have a limited number of files open at one time, so it is important that you call `fs.close()` on your file descriptors when you are done with them. You need to open files if you want to use the lowest-level `fs.read()` and `fs.write()` functions that allow you to jump around within a file, reading and writing bits of it at different times. There are other functions in the "fs" module that use file descriptors, but they all have name-based versions, and it only really makes sense to use the descriptor-based functions if you were going to open the file to read or write anyway.

Finally, in the Promise-based API defined by `fs.promises`, the equivalent of `fs.open()` is `fs.promises.open()`, which returns a Promise that resolves to a FileHandle object. This FileHandle object serves the same purpose as a file descriptor. Again, however, unless you need to use the lowest-level `read()` and `write()` methods of a FileHandle, there is really no reason to create one. And if you do create a FileHandle, you should remember to call its `close()` method once you are done with it.

16.7.2 Reading Files

Node allows you to read file content all at once, via a stream, or with the low-level API.

If your files are small, or if memory usage and performance are not the highest priority, then it is often easiest to read the entire content of a file with a single call. You can do this synchronously, with a callback, or with a Promise. By default, you'll get the bytes of the file as a buffer, but if you specify an encoding, you'll get a decoded string instead.

```
const fs = require("fs");
let buffer = fs.readFileSync("test.data");      // Synchronous, returns buff
let text = fs.readFileSync("data.csv", "utf8"); // Synchronous, returns stri

// Read the bytes of the file asynchronously
fs.readFile("test.data", (err, buffer) => {
    if (err) {
        // Handle the error here
    } else {
        // The bytes of the file are in buffer
    }
});

// Promise-based asynchronous read
fs.promises
    .readFile("data.csv", "utf8")
    .then(processFileText)
    .catch(handleReadError);

// Or use the Promise API with await inside an async function
async function processText(filename, encoding="utf8") {
    let text = await fs.promises.readFile(filename, encoding);
    // ... process the text here...
}
```

If you are able to process the contents of a file sequentially and do not need to have the entire content of the file in memory at the same time, then reading a file via a stream may be the most efficient approach. We've covered streams extensively: here is how you might use a stream and the `pipe()` method to write the contents of a file to standard output:

```
function printFile(filename, encoding="utf8") {
    fs.createReadStream(filename, encoding).pipe(process.stdout);
}
```

Finally, if you need low-level control over exactly what bytes you read from a file and when you read them, you can open a file to get a file descriptor and then use `fs.read()`, `fs.readSync()`, or `fs.promises.read()` to read a specified number of bytes from a specified source location of the file into a specified buffer at the specified destination position:

```
const fs = require("fs");

// Reading a specific portion of a data file
fs.open("data", (err, fd) => {
    if (err) {
        // Report error somehow
        return;
    }
    try {
        // Read bytes 20 through 420 into a newly allocated buffer.
        fs.read(fd, Buffer.alloc(400), 0, 400, 20, (err, n, b) => {
            // err is the error, if any.
            // n is the number of bytes actually read
            // b is the buffer that they bytes were read into.
        });
    }
    finally {           // Use a finally clause so we always
        fs.close(fd);   // close the open file descriptor
    }
});
```

The callback-based `read()` API is awkward to use if you need to read more than one chunk of data from a file. If you can use the synchronous API (or the Promise-based API with `await`), it becomes easy to read multiple chunks from a file:

```
const fs = require("fs");

function readData(filename) {
    let fd = fs.openSync(filename);
    try {
        // Read the file header
        let header = Buffer.alloc(12); // A 12 byte buffer
        fs.readSync(fd, header, 0, 12, 0);

        // Verify the file's magic number
        let magic = header.readInt32LE(0);
        if (magic !== 0xDADAFEED) {
            throw new Error("File is of wrong type");
        }

        // Now get the offset and length of the data from the header
        let offset = header.readInt32LE(4);
        let length = header.readInt32LE(8);

        // And read those bytes from the file
        let data = Buffer.alloc(length);
        fs.readSync(fd, data, 0, length, offset);
        return data;
    } finally {
        // Always close the file, even if an exception is thrown above
        fs.closeSync(fd);
    }
}
```

16.7.3 Writing Files

Writing files in Node is a lot like reading them, with a few extra details that you need to know about. One of these details is that the way you create a new file is simply by writing to a filename that does not already exist.

As with reading, there are three basic ways to write files in Node. If you have the entire content of the file in a string or a buffer, you can write the entire thing in one call with `fs.writeFile()` (callback-based), `fs.writeFileSync()` (synchronous), or `fs.promises.writeFile()` (Promise-based):

```
fs.writeFileSync(path.resolve(__dirname, "settings.json"),
                 JSON.stringify(settings));
```

If the data you are writing to the file is a string, and you want to use an encoding other than “utf8,” pass the encoding as an optional third argument.

The related functions `fs.appendFile()`, `fs.appendFileSync()`, and `fs.promises.appendFile()` are similar, but when the specified file already exists, they append their data to the end rather than overwriting the existing file content.

If the data you want to write to a file is not all in one chunk, or if it is not all in memory at the same time, then using a Writable stream is a good approach, assuming that you plan to write the data from beginning to end without skipping around in the file:

```
const fs = require("fs");
let output = fs.createWriteStream("numbers.txt");
for(let i = 0; i < 100; i++) {
    output.write(`${i}\n`);
}
output.end();
```

Finally, if you want to write data to a file in multiple chunks, and you want to be able to control the exact position within the file at which each chunk is written, then you can open the file with `fs.open()`, `fs.openSync()`, or `fs.promises.open()` and then use the resulting file descriptor with the `fs.write()` or `fs.writeSync()` functions.

These functions come in different forms for strings and buffers. The string variant takes a file descriptor, a string, and the file position at which to write that string (with an encoding as an optional fourth argument). The buffer variant takes a file descriptor, a buffer, an offset, and a length that specify a chunk of data within the buffer, and a file position at which to write the bytes of that chunk. And if you have an array of Buffer objects that you want to write, you can do this with a single `fs.writev()` or `fs.writevSync()`. Similar low-level functions exist for writing buffers and strings using `fs.promises.open()` and the FileHandle object it produces.

FILE MODE STRINGS

We saw the `fs.open()` and `fs.openSync()` methods before when using the low-level API to read files. In that use case, it was sufficient to just pass the filename to the open function. When you want to write a file, however, you must also specify a second string argument that specifies how you intend to use the file descriptor. Some of the available flag strings are as follows:

"w"

Open the file for writing

"w+"

Open for writing and reading

"wx"

Open for creating a new file; fails if the named file already exists

"wx+"

Open for creation, and also allow reading; fails if the named file already exists

"a"

Open the file for appending; existing content won't be overwritten

"a+"

Open for appending, but also allow reading

If you do not pass one of these flag strings to `fs.open()` or `fs.openSync()`, they use the default "r" flag, making the file descriptor read-only. Note that it can also be useful to pass these flags to other file-writing methods:

```
// Write to a file in one call, but append to anything that is already there
// This works like fs.appendFileSync()
fs.writeFileSync("messages.log", "hello", { flag: "a" });

// Open a write stream, but throw an error if the file already exists.
// We don't want to accidentally overwrite something!
// Note that the option above is "flag" and is "flags" here
fs.createWriteStream("messages.log", { flags: "wx" });
```

You can chop off the end of a file with `fs.truncate()`, `fs.truncateSync()`, or `fs.promises.truncate()`. These functions take a path as their first argument and a length as their second, and modify the file so that it has the specified length. If you omit the length, zero is used and the file becomes empty. Despite the name of these functions, they can also be used to extend a file: if you specify a length that is longer than the current file size, the file is extended with zero bytes to the new size. If you have already opened the file you wish to modify, you can use `ftruncate()` or `ftruncateSync()` with the file descriptor or FileHandle.

The various file-writing functions described here return or invoke their callback or resolve their Promise when the data has been "written" in the sense that Node has handed it off to the operating system. But this does not necessarily mean that the data has actually been written to persistent storage yet: at least some of your data may still be buffered somewhere in the operating system or in a device driver waiting to be written to disk. If you call `fs.writeSync()` to synchronously write some data to a file, and if there is a power outage immediately after the function returns, you may still lose data. If you want to force your data out to disk so you know for sure that it has been safely saved, use `fs.fsync()` or `fs.fsyncSync()`. These functions only work with file descriptors: there is no path-based version.

16.7.4 File Operations

The preceding discussion of Node's stream classes included two examples of `copyFile()` functions. These are not practical utilities that you would actually use because the "fs" module defines its own `fs.copyFile()` method (and also `fs.copyFileSync()` and `fs.promises.copyFile()`, of course).

These functions take the name of the original file and the name of the copy as their first two arguments. These can be specified as strings or as URL or Buffer objects. An optional third argument is an integer whose bits specify flags that control details of the `copy` operation. And for the callback-based `fs.copyFile()`, the final argument is a callback function that will be called with no arguments when the copy is complete, or that will be called with an error argument if something fails. Following are some examples:

```
// Basic synchronous file copy.
fs.copyFileSync("ch15.txt", "ch15.bak");

// The COPYFILE_EXCL argument copies only if the new file does not already
// exist. It prevents copies from overwriting existing files.
fs.copyFile("ch15.txt", "ch16.txt", fs.constants.COPYFILE_EXCL, err => {
    // This callback will be called when done. On error, err will be non-nul
});

// This code demonstrates the Promise-based version of the copyFile function
// Two flags are combined with the bitwise OR opeartor |. The flags mean tha
// existing files won't be overwritten, and that if the filesystem supports
// it, the copy will be a copy-on-write clone of the original file, meaning
// that no additional storage space will be required until either the origin
// or the copy is modified.
fs.promises.copyFile("Important data",
                     `Important data ${new Date().toISOString()}"
                     fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICL
    .then(() => {
        console.log("Backup complete");
    });
    .catch(err => {
        console.error("Backup failed", err);
    });
```

The `fs.rename()` function (along with the usual synchronous and Promise-based variants) moves and/or renames a file. Call it with the current path to the file and the desired new path to the file. There is no flags argument, but the callback-based version takes a callback as the third argument:

```
fs.renameSync("ch15.bak", "backups/ch15.bak");
```

Note that there is no flag to prevent renaming from overwriting an existing file. Also keep in mind that files can only be renamed within a filesystem.

The functions `fs.link()` and `fs.symlink()` and their variants have the same signatures as `fs.rename()` and behave something like `fs.copyFile()` except that they create hard links and symbolic links, respectively, rather than creating a copy.

Finally, `fs.unlink()`, `fs.unlinkSync()`, and `fs.promises.unlink()` are Node's functions for deleting a file. (The unintuitive naming is inherited from Unix where deleting a file is basically the opposite of creating a hard link to it.) Call this function with the string, buffer, or URL path to the file to be deleted, and pass a callback if you are using the callback-based version:

```
fs.unlinkSync("backups/ch15.bak");
```

16.7.5 File Metadata

The `fs.stat()`, `fs.statSync()`, and `fs.promises.stat()` functions allow you to obtain metadata for a specified file or directory. For example:

```
const fs = require("fs");
let stats = fs.statSync("book/ch15.md");
stats.isFile()         // => true: this is an ordinary file
stats.isDirectory()    // => false: it is not a directory
stats.size             // file size in bytes
stats.atime            // access time: Date when it was last read
stats.mtime            // modification time: Date when it was last written
stats.uid              // the user id of the file's owner
stats.gid              // the group id of the file's owner
stats.mode.toString(8) // the file's permissions, as an octal string
```

The returned Stats object contains other, more obscure properties and methods, but this code demonstrates those that you are most likely to use.

`fs.lstat()` and its variants work just like `fs.stat()`, except that if the specified file is a symbolic link, Node will return metadata for the link itself rather than following the link.

If you have opened a file to produce a file descriptor or a FileHandle object, then you can use `fs.fstat()` or its variants to get metadata information for the opened file without having to specify the filename again.

In addition to querying metadata with `fs.stat()` and all of its variants, there are also functions for changing metadata.

`fs.chmod()`, `fs.lchmod()`, and `fs.fchmod()` (along with synchronous and Promise-based versions) set the “mode” or permissions of a file or directory. Mode values are integers in which each bit has a specific meaning and are easiest to think about in octal notation. For example, to make a file read-only to its owner and inaccessible to everyone else, use `0o400`:

```
fs.chmodSync("ch15.md", 0o400);  // Don't delete it accidentally!
```

`fs.chown()`, `fs.lchown()`, and `fs.fchown()` (along with synchronous and Promise-based versions) set the owner and group (as IDs) for a file or directory. (These matter because they interact with the file permissions set by `fs.chmod()`.)

Finally, you can set the access time and modification time of a file or directory with `fs.utimes()` and `fs.futimes()` and their variants.

16.7.6 Working with Directories

To create a new directory in Node, use `fs.mkdir()`, `fs.mkdirSync()`, or `fs.promises.mkdir()`. The first argument is the path of the directory to be created. The optional second argument can be an integer that specifies the mode (permissions bits) for the new directory. Or you can pass an object with optional `mode` and `recursive` properties. If `recursive` is `true`, then this function will create any directories in the path that do not already exist:

```
// Ensure that dist/ and dist/lib/ both exist.
fs.mkdirSync("dist/lib", { recursive: true });
```

`fs.mkdtemp()` and its variants take a path prefix you provide, append some random characters to it (this is important for security), create a directory with that name, and return (or pass to a callback) the directory path to you.

To delete a directory, use `fs.rmdir()` or one of its variants. Note that directories must be empty before they can be deleted:

```
// Create a random temporary directory and get its path, then
// delete it when we are done
let tempDirPath;
try {
    tempDirPath = fs.mkdtempSync(path.join(os.tmpdir(), "d"));
    // Do something with the directory here
} finally {
    // Delete the temporary directory when we're done with it
```

```
        fs.rmdirSync(tempDirPath);
    }
```

The “fs” module provides two distinct APIs for listing the contents of a directory. First, `fs.readdir()`, `fs.readdirSync()`, and `fs.promises.readdir()` read the entire directory all at once and give you an array of strings or an array of Dirent objects that specify the names and types (file or directory) of each item. Filenames returned by these functions are just the local name of the file, not the entire path. Here are examples:

```
let tempFiles = fs.readdirSync("/tmp");  // returns an array of strings

// Use the Promise-based API to get a Dirent array, and then
// print the paths of subdirectories
fs.promises.readdir("/tmp", {withFileTypes: true})
    .then(entries => {
        entries.filter(entry => entry.isDirectory())
            .map(entry => entry.name)
            .forEach(name => console.log(path.join("/tmp/", name)));
    })
    .catch(console.error);
```

If you anticipate needing to list directories that might have thousands of entries, you might prefer the streaming approach of `fs.opendir()` and its variants. These functions return a Dir object representing the specified directory. You can use the `read()` or `readSync()` methods of the Dir object to read one Dirent at a time. If you pass a callback function to `read()`, it will call the callback. And if you omit the callback argument, it will return a Promise. When there are no more directory entries, you’ll get `null` instead of a Dirent object.

The easiest way to use Dir objects is as async iterators with a `for/await` loop. Here, for example, is a function that uses the streaming API to list directory entries, calls `stat()` on each entry, and prints file and directory names and sizes:

```
const fs = require("fs");
const path = require("path");

async function listDirectory(dirpath) {
    let dir = await fs.promises.opendir(dirpath);
    for await (let entry of dir) {
        let name = entry.name;
        if (entry.isDirectory()) {
            name += "/";  // Add a trailing slash to subdirectories
        }
        let stats = await fs.promises.stat(path.join(dirpath, name));
        let size = stats.size;
        console.log(String(size).padStart(10), name);
    }
}
```

16.8 HTTP Clients and Servers

Node's "http," "https," and "http2" modules are full-featured but relatively low-level implementations of the HTTP protocols. They define comprehensive APIs for implementing HTTP clients and servers. Because the APIs are relatively low-level, there is not room in this chapter to cover all the features. But the examples that follow demonstrate how to write basic clients and servers.

The simplest way to make a basic HTTP GET request is with `http.get()` or `https.get()`. The first argument to these functions is the URL to fetch. (If it is an `http://` URL, you must use the "http" module, and if it is an `https://` URL you must use the "https" module.) The second argument is a callback that will be invoked with an IncomingMessage object when the server's response has started to arrive. When the callback is called, the HTTP status and headers are available, but the body may not be ready yet. The IncomingMessage object is a Readable stream, and you can use the techniques demonstrated earlier in this chapter to read the response body from it.

The `getJSON()` function at the end of §13.2.6 used the `http.get()` function as part of a demonstration of the `Promise()` constructor. Now that you know about Node streams and the Node programming model more generally, it is worth revisiting that example to see how `http.get()` is used.

`http.get()` and `https.get()` are slightly simplified variants of the more general `http.request()` and `https.request()` functions. The following `postJSON()` function demonstrates how to use `https.request()` to make an HTTPS POST request that includes a JSON request body. Like the `getJSON()` function of Chapter 13, it expects a

JSON response and returns a Promise that fulfills to the parsed version of that response:

```
const https = require("https");

/*
 * Convert the body object to a JSON string then HTTPS POST it to the
 * specified API endpoint on the specified host. When the response arrives,
 * parse the response body as JSON and resolve the returned Promise with
 * that parsed value.
 */
function postJSON(host, endpoint, body, port, username, password) {
    // Return a Promise object immediately, then call resolve or reject
    // when the HTTPS request succeeds or fails.
    return new Promise((resolve, reject) => {
        // Convert the body object to a string
        let bodyText = JSON.stringify(body);

        // Configure the HTTPS request
        let requestOptions = {
            method: "POST",       // Or "GET", "PUT", "DELETE", etc.
            host: host,           // The host to connect to
            path: endpoint,       // The URL path
            headers: {            // HTTP headers for the request
                "Content-Type": "application/json",
                "Content-Length": Buffer.byteLength(bodyText)
            }
        };

        if (port) {                          // If a port is specified,
            requestOptions.port = port;      // use it for the request.
        }
        // If credentials are specified, add an Authorization header.
        if (username && password) {
            requestOptions.auth = `${username}:${password}`;
        }

        // Now create the request based on the configuration object
        let request = https.request(requestOptions);

        // Write the body of the POST request and end the request.
        request.write(bodyText);
        request.end();

        // Fail on request errors (such as no network connection)
        request.on("error", e => reject(e));
        // Handle the response when it starts to arrive.
        request.on("response", response => {
            if (response.statusCode !== 200) {
                reject(new Error(`HTTP status ${response.statusCode}`));
                // We don't care about the response body in this case, but
                // we don't want it to stick around in a buffer somewhere, s
                // we put the stream into flowing mode without registering
                // a "data" handler so that the body is discarded.
                response.resume();
                return;
            }

            // We want text, not bytes. We're assuming the text will be
            // JSON-formatted but aren't bothering to check the
            // Content-Type header.
            response.setEncoding("utf8");

            // Node doesn't have a streaming JSON parser, so we read the
            // entire response body into a string.
            let body = "";
            response.on("data", chunk => { body += chunk; });

            // And now handle the response when it is complete.
            response.on("end", () => {          // When the response is done
                try {                           // try to parse it as JSON
                    resolve(JSON.parse(body));  // and resolve the result.
                } catch(e) {                    // Or, if anything goes wron
                    reject(e);                  // reject with the error
                }
            });
        });
    });
}
```

In addition to making HTTP and HTTPS requests, the “http” and “https” modules also allow you to write servers that respond to those requests. The basic approach is as follows:

- Create a new Server object.
- Call its `listen()` method to begin listening for requests on a specified port.
- Register an event handler for “request” events, use that handler to read the client’s request (particularly the `request.url` property), and write your response.

The code that follows creates a simple HTTP server that serves static files from the local filesystem and also implements a debugging endpoint that responds to a client's request by echoing that request.

```
// This is a simple static HTTP server that serves files from a specified
// directory. It also implements a special /test/mirror endpoint that
// echoes the incoming request, which can be useful when debugging clients.
const http = require("http");   // Use "https" if you have a certificate
const url = require("url");     // For parsing URLs
const path = require("path");   // For manipulating filesystem paths
const fs = require("fs");       // For reading files

// Serve files from the specified root directory via an HTTP server that
// listens on the specified port.
function serve(rootDirectory, port) {
    let server = new http.Server();  // Create a new HTTP server
    server.listen(port);             // Listen on the specified port
    console.log("Listening on port", port);

    // When requests come in, handle them with this function
    server.on("request", (request, response) => {
        // Get the path portion of the request URL, ignoring
        // any query parameters that are appended to it.
        let endpoint = url.parse(request.url).pathname;

        // If the request was for "/test/mirror", send back the request
        // verbatim. Useful when you need to see the request headers and bod
        if (endpoint === "/test/mirror") {
            // Set response header
            response.setHeader("Content-Type", "text/plain; charset=UTF-8");

            // Specify response status code
            response.writeHead(200);  // 200 OK

            // Begin the response body with the request
            response.write(`${request.method} ${request.url} HTTP/${
                                   request.httpVersion
                               }\r\n`);

            // Output the request headers
            let headers = request.rawHeaders;
            for(let i = 0; i < headers.length; i += 2) {
                response.write(`${headers[i]}: ${headers[i+1]}\r\n`);
            }

            // End headers with an extra blank line
            response.write("\r\n");
            // Now we need to copy any request body to the response body
            // Since they are both streams, we can use a pipe
            request.pipe(response);
        }
        // Otherwise, serve a file from the local directory.
        else {
            // Map the endpoint to a file in the local filesystem
            let filename = endpoint.substring(1); // strip leading /
            // Don't allow "../" in the path because it would be a security
            // hole to serve anything outside the root directory.
            filename = filename.replace(/\.\.\//g, "");
            // Now convert from relative to absolute filename
            filename = path.resolve(rootDirectory, filename);

            // Now guess the type file's content type based on extension
            let type;
            switch(path.extname(filename))  {
            case ".html":
            case ".htm": type = "text/html"; break;
            case ".js":  type = "text/javascript"; break;
            case ".css": type = "text/css"; break;
            case ".png": type = "image/png"; break;
            case ".txt": type = "text/plain"; break;
            default:     type = "application/octet-stream"; break;
            }

            let stream = fs.createReadStream(filename);
            stream.once("readable", () => {
                // If the stream becomes readable, then set the
                // Content-Type header and a 200 OK status. Then pipe the
                // file reader stream to the response. The pipe will
                // automatically call response.end() when the stream ends.
                response.setHeader("Content-Type", type);
                response.writeHead(200);
                stream.pipe(response);
            });

            stream.on("error", (err) => {
                // Instead, if we get an error trying to open the stream
                // then the file probably does not exist or is not readable.
                // Send a 404 Not Found plain-text response with the
                // error message.
                response.setHeader("Content-Type", "text/plain; charset=UTF-
                response.writeHead(404);
                response.end(err.message);
            });
        }
    });
}

// When we're invoked from the command line, call the serve() function
serve(process.argv[2] || "/tmp", parseInt(process.argv[3]) || 8000);
```

Node's built-in modules are all you need to write simple HTTP and HTTPS servers. Note, however, that production servers are not typically built directly on top of these modules. Instead, most nontrivial servers are implemented using external libraries—such as the Express framework—that provide "middleware" and other higher-level utilities that backend web developers have come to expect.

16.9 Non-HTTP Network Servers and Clients

Web servers and clients have become so ubiquitous that it is easy to forget that it is possible to write clients and servers that do not use HTTP. Even though Node has a reputation as a good environment for writing web servers, Node also has full support for writing other types of network servers and clients.

If you are comfortable working with streams, then networking is relatively simple, because network sockets are simply a kind of Duplex stream. The "net" module defines Server and Socket classes. To create a server, call `net.createServer()`, then call the `listen()` method of the resulting object to tell the server what port to listen on for connections. The Server object will generate "connection" events when a client connects on that port, and the value passed to the event listener will be a Socket object. The Socket object is a Duplex stream, and you can use it to read data from the client and write data to the client. Call `end()` on the Socket to disconnect.

Writing a client is even easier: pass a port number and hostname to `net.createConnection()` to create a socket to communicate with whatever server is running on that host and listening on that port. Then use that socket to read and write data from and to the server.

The following code demonstrates how to write a server with the "net" module. When the client connects, the server tells a knock-knock joke:

```
// A TCP server that delivers interactive knock-knock jokes on port 6789.
// (Why is six afraid of seven? Because seven ate nine!)
const net = require("net");
const readline = require("readline");

// Create a Server object and start listening for connections
let server = net.createServer();
server.listen(6789, () => console.log("Delivering laughs on port 6789"));

// When a client connects, tell them a knock-knock joke.
server.on("connection", socket => {
    tellJoke(socket)
        .then(() => socket.end())  // When the joke is done, close the socke
        .catch((err) => {
            console.error(err);    // Log any errors that occur,
            socket.end();          // but still close the socket!
        });
});

// These are all the jokes we know.
const jokes = {
    "Boo": "Don't cry...it's only a joke!",
    "Lettuce": "Let us in! It's freezing out here!",
    "A little old lady": "Wow, I didn't know you could yodel!"
};

// Interactively perform a knock-knock joke over this socket, without blocki
async function tellJoke(socket) {
    // Pick one of the jokes at random
    let randomElement = a => a[Math.floor(Math.random() * a.length)];
    let who = randomElement(Object.keys(jokes));
    let punchline = jokes[who];

    // Use the readline module to read the user's input one line at a time.
    let lineReader = readline.createInterface({
        input: socket,
        output: socket,
        prompt: ">> "
    });

    // A utility function to output a line of text to the client
    // and then (by default) display a prompt.
    function output(text, prompt=true) {
        socket.write(`${text}\r\n`);
        if (prompt) lineReader.prompt();
    }

    // Knock-knock jokes have a call-and-response structure.
```

```
        // We expect different input from the user at different stages and
        // take different action when we get that input at different stages.
        let stage = 0;

        // Start the knock-knock joke off in the traditional way.
        output("Knock knock!");

        // Now read lines asynchronously from the client until the joke is done.
        for await (let inputLine of lineReader) {
            if (stage === 0) {
                if (inputLine.toLowerCase() === "who's there?") {
                    // If the user gives the right response at stage 0
                    // then tell the first part of the joke and go to stage 1.
                    output(who);
                    stage = 1;
                } else  {
                    // Otherwise teach the user how to do knock-knock jokes.
                    output('Please type "Who\'s there?".');
                }
            } else if (stage === 1) {
                if (inputLine.toLowerCase() === `${who.toLowerCase()} who?`) {
                    // If the user's response is correct at stage 1, then
                    // deliver the punchline and return since the joke is done.
                    output(`${punchline}`, false);
                    return;
                } else {
                    // Make the user play along.
                    output(`Please type "${who} who?".`);
                }
            }
        }
    }
```

Simple text-based servers like this do not typically need a custom client. If the `nc` ("netcat") utility is installed on your system, you can use it to communicate with this server as follows:

```
$ nc localhost 6789
Knock knock!
>> Who's there?
A little old lady
>> A little old lady who?
Wow, I didn't know you could yodel!
```

On the other hand, writing a custom client for the joke server is easy in Node. We just connect to the server, then pipe the server's output to stdout and pipe stdin to the server's input:

```
// Connect to the joke port (6789) on the server named on the command line
let socket = require("net").createConnection(6789, process.argv[2]);
socket.pipe(process.stdout);              // Pipe data from the socket to st
process.stdin.pipe(socket);               // Pipe data from stdin to the soc
socket.on("close", () => process.exit()); // Quit when the socket closes.
```

In addition to supporting TCP-based servers, Node's "net" module also supports interprocess communication over "Unix domain sockets" that are identified by a filesystem path rather than by a port number. We are not going to cover that kind of socket in this chapter, but the Node documentation has details. Other Node features that we don't have space to cover here include the "dgram" module for UDP-based clients and servers and the "tls" module that is to "net" as "https" is to "http." The `tls.Server` and `tls.TLSSocket` classes allow the creation of TCP servers (like the knock-knock joke server) that use SSL-encrypted connections like HTTPS servers do.

16.10 Working with Child Processes

In addition to writing highly concurrent servers, Node also works well for writing scripts that execute other programs. In Node the "child_process" module defines a number of functions for running other programs as child processes. This section demonstrates some of those functions, starting with the simplest and moving to the more complicated.

16.10.1 execSync() and execFileSync()

The easiest way to run another program is with `child_process.execSync()`. This function takes the command to run as its first argument. It creates a child process, runs a shell in that process, and uses the shell to execute the command you passed. Then it blocks until the command (and the shell) exit. If the command exits with an error, then `execSync()` throws an exception. Otherwise, `execSync()` returns whatever output the command writes to its stdout stream. By default this return value is a buffer, but you can specify an encoding in an optional second argument to get a string instead. If the command writes any output to stderr, that output just gets passed through to the parent process's stderr stream.

So, for example, if you are writing a script and performance is not a concern, you might use `child_process.execSync()` to list a directory with a familiar Unix shell command rather than using the `fs.readdirSync()` function:

```
const child_process = require("child_process");
let listing = child_process.execSync("ls -l web/*.html", {encoding: "utf8"})
```

The fact that `execSync()` invokes a full Unix shell means that the string you pass to it can include multiple semicolon-separated commands, and can take advantage of shell features such as filename wildcards, pipes, and output redirection. This also means that you must be careful to never pass a command to `execSync()` if any portion of that command is user input or comes from a similar untrusted source. The complex syntax of shell commands can be easily subverted to allow an attacker to run arbitrary code.

If you don't need the features of a shell, you can avoid the overhead of starting a shell by using `child_process.execFileSync()`. This function executes a program directly, without invoking a shell. But since no shell is involved, it can't parse a command line, and you must pass the executable as the first argument and an array of command-line arguments as the second argument:

```
let listing = child_process.execFileSync("ls", ["-l", "web/"],
                                         {encoding: "utf8"});
```

CHILD PROCESS OPTIONS

`execSync()` and many of the other `child_process` functions have a second or third optional argument that specifies additional details about how the child process is to run. The `encoding` property of this object was used earlier to specify that we'd like the command output to be delivered as a string rather than as a buffer. Other important properties that you can specify include the following (note that not all options are available to all child process functions):

- `cwd` specifies the working directory for the child process. If you omit this, then the child process inherits the value of `process.cwd()`.
- `env` specifies the environment variables that the child process will have access to. By default, child processes simply inherit `process.env`, but you can specify a different object if you want.
- `input` specifies a string or buffer of input data that should be used as the standard input to the child process. This option is only available to the synchronous functions that do not return a ChildProcess object.
- `maxBuffer` specifies the maximum number of bytes of output that will be collected by the `exec` functions. (It does not apply to `spawn()` and `fork()`, which use streams.) If a child process produces more output than this, it will be killed and will exit with an error.
- `shell` specifies the path to a shell executable or `true`. For child process functions that normally execute a shell command, this option allows you to specify which shell to use. For functions that do not normally use a shell, this option allows you to specify that a shell should be used (by setting the property to `true`) or to specify exactly which shell to use.
- `timeout` specifies the maximum number of milliseconds that the child process should be allowed to run. If it has not exited before this time elapses, it will be killed and will exit with an error. (This option applies to the `exec` functions but not to `spawn()` or `fork()`.)
- `uid` specifies the user ID (a number) under which the program should be run. If the parent process is running in a privileged account, it can use this option to run the child with reduced privileges.

16.10.2 exec() and execFile()

The `execSync()` and `execFileSync()` functions are, as their names indicate, synchronous: they block and do not return until the child process exits. Using these functions is a lot like typing Unix commands in a terminal window: they allow you to run a sequence of commands one at a time. But if you're writing a program that needs to accomplish a number of tasks, and those tasks don't depend on each other in any way, then you may want to parallelize them and run multiple commands at the same time. You can do this with the asynchronous functions `child_process.exec()` and `child_process.execFile()`.

`exec()` and `execFile()` are like their synchronous variants except that they return immediately with a ChildProcess object that represents the running child process, and they take an error-first callback as their final argument. The callback is invoked when the child process exits, and it is actually called with three arguments. The first is the error, if any; it will be `null` if the process terminated normally. The second argument is the collected output that was sent to the child's standard output stream. And the third argument is any output that was sent to the child's standard error stream.

The ChildProcess object returned by `exec()` and `execFile()` allows you to terminate the child process, and to write data to it (which it can then read from its standard input). We'll cover ChildProcess in more detail when we discuss the `child_process.spawn()` function.

If you plan to execute multiple child processes at the same time, then it may be easiest to use the "promisified" version of `exec()` which returns a Promise object which, if the child process exits without error, resolves to an object with `stdout` and `stderr` properties. Here, for example, is a function that takes an array of shell commands as its input and returns a Promise that resolves to the result of all of those commands:

```
const child_process = require("child_process");
const util = require("util");
const execP = util.promisify(child_process.exec);

function parallelExec(commands) {
    // Use the array of commands to create an array of Promises
    let promises = commands.map(command => execP(command, {encoding: "utf8"}
    // Return a Promise that will fulfill to an array of the fulfillment
    // values of each of the individual promises. (Instead of returning obje
    // with stdout and stderr properties we just return the stdout value.)
    return Promise.all(promises)
        .then(outputs => outputs.map(out => out.stdout));
}

module.exports = parallelExec;
```

16.10.3 spawn()

The various `exec` functions described so far—both synchronous and asynchronous—are designed to be used with child processes that run quickly and do not produce a lot of output. Even the asynchronous `exec()` and `execFile()` are nonstreaming: they return the process output in a single batch, only after the process has exited.

The `child_process.spawn()` function allows you streaming access to the output of the child process, while the process is still running. It also allows you to write data to the child process (which will see that data as input on its standard input stream): this means it is possible to dynamically interact with a child process, sending it input based on the output it generates.

`spawn()` does not use a shell by default, so you must invoke it like `execFile()` with the executable to be run and a separate array of command-line arguments to pass to it. `spawn()` returns a ChildProcess object like `execFile()` does, but it does not take a callback argument. Instead of using a callback function, you listen to events on the ChildProcess object and on its streams.

The ChildProcess object returned by `spawn()` is an event emitter. You can listen for the "exit" event to be notified when the child process exits. A ChildProcess object also has three stream properties. `stdout` and `stderr` are Readable streams: when the child process writes to its stdout and its stderr streams, that output becomes readable through the ChildProcess streams. Note the inversion of the names here. In the child process, "stdout" is a Writable output stream, but in the parent process, the `stdout` property of a ChildProcess object is a Readable input stream.

Similarly, the `stdin` property of the ChildProcess object is a Writeable stream: anything you write to this stream becomes available to the child process on its standard input.

The ChildProcess object also defines a `pid` property that specifies the process id of the child. And it defines a `kill()` method that you can use to terminate a child process.

16.10.4 fork()

`child_process.fork()` is a specialized function for running a module of JavaScript code in a child Node process. `fork()` expects the same arguments as `spawn()`, but the first argument should specify the path to a file of JavaScript code instead of an executable binary file.

A child process created with `fork()` can communicate with the parent process via its standard input and standard output streams, as described in the previous section for `spawn()`. But in addition, `fork()` enables another, much easier, communication channel between the parent and child processes.

When you create a child process with `fork()`, you can use the `send()` method of the returned ChildProcess object to send a copy of an object to the child process. And you can listen for the "message" event on the ChildProcess to receive messages from the child. The code running in the child process can use `process.send()` to send a message to the parent and can listen for "message" events on `process` to receive messages from the parent.

Here, for example, is some code that uses `fork()` to create a child process, then sends that child a message and waits for a response:

```
const child_process = require("child_process");

// Start a new node process running the code in child.js in our directory
let child = child_process.fork(`${__dirname}/child.js`);

// Send a message to the child
child.send({x: 4, y: 3});

// Print the child's response when it arrives.
child.on("message", message => {
    console.log(message.hypotenuse); // This should print "5"
    // Since we only send one message we only expect one response.
    // After we receive it we call disconnect() to terminate the connection
    // between parent and child. This allows both processes to exit cleanly.
    child.disconnect();
});
```

And here is the code that runs in the child process:

```
// Wait for messages from our parent process
process.on("message", message => {
    // When we receive one, do a calculation and send the result
```

```
        // back to the parent.
        process.send({hypotenuse: Math.hypot(message.x, message.y)});
    });
```

Starting child processes is an expensive operation, and the child process would have to be doing orders of magnitude more computation before it would make sense to use `fork()` and interprocess communication in this way. If you are writing a program that needs to be very responsive to incoming events and also needs to perform time-consuming computations, then you might consider using a separate child process to perform the computations so that they don't block the event loop and reduce the responsiveness of the parent process. (Though a thread—see §16.11—may be a better choice than a child process in this scenario.)

The first argument to `send()` will be serialized with `JSON.stringify()` and deserialized in the child process with `JSON.parse()`, so you should only include values that are supported by the JSON format. `send()` has a special second argument, however, that allows you to transfer Socket and Server objects (from the "net" module) to a child process. Network servers tend to be IO-bound rather than compute-bound, but if you have written a server that needs to do more computation than a single CPU can handle, and if you're running that server on a machine with multiple CPUs, then you could use `fork()` to create multiple child processes for handling requests. In the parent process, you might listen for "connection" events on your Server object, then get the Socket object from that "connection" event and `send()` it—using the special second argument—to one of the child processes to be handled. (Note that this is an unlikely solution to an uncommon scenario. Rather than writing a server that forks child processes, it is probably simpler to keep your server single-threaded and deploy multiple instances of it in production to handle the load.)

16.11 Worker Threads

As explained at the beginning of this chapter, Node's concurrency model is single-threaded and event-based. But in version 10 and later, Node does allow true multithreaded programming, with an API that closely mirrors the Web Workers API defined by web browsers (§15.13). Multithreaded programming has a well-deserved reputation for being difficult. This is almost entirely because of the need to carefully synchronize access by threads to shared memory. But JavaScript threads (in both Node and browsers) do not share memory by default, so the dangers and difficulties of using threads do not apply to these “workers” in JavaScript.

Instead of using shared memory, JavaScript’s worker threads communicate by message passing. The main thread can send a message to a worker thread by calling the `postMessage()` method of the Worker object that represents that thread. The worker thread can receive messages from its parent by listening for “message” events. And workers can send messages to the main thread with their own version of `postMessage()`, which the parent can receive with its own “message” event handler. The example code will make it clear how this works.

There are three reasons why you might want to use worker threads in a Node application:

- If your application actually needs to do more computation than one CPU core can handle, then threads allow you to distribute work across the multiple cores, which have become commonplace on computers today. If you’re doing scientific computing or machine learning or graphics processing in Node, then you may want to use threads simply to throw more computing power at your problem.
- Even if your application is not using the full power of one CPU, you may still want to use threads to maintain the responsiveness of the main thread. Consider a server that handles large but relatively infrequent requests. Suppose it gets only one request a second, but needs to spend about half a second of (blocking CPU-bound) computation to process each request. On average, it will be idle 50% of the time. But when two requests arrive within a few milliseconds of each other, the server will not even be able to begin a response to the second request until the computation of the first response is complete. Instead, if the server uses a worker thread to perform the computation, the server can begin the response to both requests immediately and provide a better experience for the server’s clients. Assuming the server has more than one CPU core, it can also compute the body of both responses in parallel, but even if there is only a single core, using workers still improves the responsiveness.
- In general, workers allow us to turn blocking synchronous operations into nonblocking asynchronous operations. If you are writing a program that depends on legacy code that is unavoidably synchronous, you may be able to use workers to avoid blocking when you need to call that legacy code.

Worker threads are not nearly as heavyweight as child processes, but they are not lightweight. It does not generally make sense to create a worker unless you have significant work for it to do. And, generally speaking, if your program is not CPU-bound and is not having responsiveness problems, then you probably do not need worker threads.

16.11.1 Creating Workers and Passing Messages

The Node module that defines workers is known as "worker_threads." In this section we'll refer to it with the identifier `threads`:

```
const threads = require("worker_threads");
```

This module defines a Worker class to represent a worker thread, and you can create a new thread with the `threads.Worker()` constructor. The following code demonstrates using this constructor to create a worker, and shows how to pass messages from main thread to worker and from worker to main thread. It also demonstrates a trick that allows you to put the main thread code and the worker thread code in the same file.[2]

```
const threads = require("worker_threads");

// The worker_threads module exports the boolean isMainThread property.
// This property is true when Node is running the main thread and it is
// false when Node is running a worker. We can use this fact to implement
// the main and worker threads in the same file.
if (threads.isMainThread) {
    // If we're running in the main thread, then all we do is export
    // a function. Instead of performing a computationally intensive
    // task on the main thread, this function passes the task to a worker
    // and returns a Promise that will resolve when the worker is done.
    module.exports = function reticulateSplines(splines) {
        return new Promise((resolve,reject) => {
            // Create a worker that loads and runs this same file of code.
            // Note the use of the special __filename variable.
            let reticulator = new threads.Worker(__filename);

            // Pass a copy of the splines array to the worker
            reticulator.postMessage(splines);

            // And then resolve or reject the Promise when we get
            // a message or error from the worker.
            reticulator.on("message", resolve);
```

```
                reticulator.on("error", reject);
            });
        };
    } else {
        // If we get here, it means we're in the worker, so we register a
        // handler to get messages from the main thread. This worker is designed
        // to only receive a single message, so we register the event handler
        // with once() instead of on(). This allows the worker to exit naturally
        // when its work is complete.
        threads.parentPort.once("message", splines => {
            // When we get the splines from the parent thread, loop
            // through them and reticulate all of them.
            for(let spline of splines) {
                // For the sake of example, assume that spline objects usually
                // have a reticulate() method that does a lot of computation.
                spline.reticulate ? spline.reticulate() : spline.reticulated = t
            }

            // When all the splines have (finally!) been reticulated
            // pass a copy back to the main thread.
            threads.parentPort.postMessage(splines);
        });
    }
```

The first argument to the `Worker()` constructor is the path to a file of JavaScript code that is to run in the thread. In the preceding code, we used the predefined `__filename` identifier to create a worker that loads and runs the same file as the main thread. In general, though, you will be passing a file path. Note that if you specify a relative path, it is relative to `process.cwd()`, not relative to the currently running module. If you want a path relative to the current module, use something like `path.resolve(__dirname, 'workers/reticulator.js')`.

The `Worker()` constructor can also accept an object as its second argument, and the properties of this object provide optional configuration for the worker. We'll cover a number of these options later, but for now note that if you pass `{eval: true}` as the second argument, then the first argument to `Worker()` is interpreted as a string of JavaScript code to be evaluated instead of a filename:

```
new threads.Worker(`
    const threads = require("worker_threads");
    threads.parentPort.postMessage(threads.isMainThread);
`, {eval: true}).on("message", console.log);  // This will print "false"
```

Node makes a copy of the object passed to `postMessage()` rather than sharing it directly with the worker thread. This prevents the worker thread and the main thread from sharing memory. You might expect that this copying would be done with `JSON.stringify()` and `JSON.parse()` (§11.6). But in fact, Node borrows a more robust technique known as the structured clone algorithm from web browsers.

The structured clone algorithm enables serialization of most JavaScript types, including Map, Set, Date, and RegExp objects and typed arrays, but it cannot, in general, copy types defined by the Node host environment, such as sockets and streams. Note, however, that Buffer objects are partially supported: if you pass a Buffer to `postMessage()` it will be received as a Uint8Array, and can be converted back into a Buffer with `Buffer.from()`. Read more about the structured clone algorithm in "The Structured Clone Algorithm".

16.11.2 The Worker Execution Environment

For the most part, JavaScript code in a Node worker thread runs just like it would in Node's main thread. There are a few differences that you should be aware of, and some of these differences involve properties of the optional second argument to the `Worker()` constructor:

- As we've seen, `threads.isMainThread` is `true` in the main thread but is always `false` in any worker thread.
- In a worker thread, you can use `threads.parentPort.postMessage()` to send a message to the parent thread and `threads.parentPort.on` to register event handlers for messages from the parent thread. In the main thread, `threads.parentPort` is always `null`.
- In a worker thread, `threads.workerData` is set to a copy of the `workerData` property of the second argument to the `Worker()` constructor. In the main thread, this property is always `null`. You can use this `workerData` property to pass an initial message to the worker that will be available as soon as it starts so that the worker does not have to wait for a "message" event before it can start doing work.
- By default, `process.env` in a worker thread is a copy of `process.env` in the parent thread. But the parent thread can specify a custom set of environment variables by setting the `env` property of the second argument to the `Worker()` constructor. As a special (and

potentially dangerous) case, the parent thread can set the `env` property to `threads.SHARE_ENV`, which will cause the two threads to share a single set of environment variables so that a change in one thread is visible in the other.

- By default, the `process.stdin` stream in a worker never has any readable data on it. You can change this default by passing `stdin: true` in the second argument to the `Worker()` constructor. If you do that, then the `stdin` property of the Worker object is a Writable stream. Any data that the parent writes to `worker.stdin` becomes readable on `process.stdin` in the worker.
- By default, the `process.stdout` and `process.stderr` streams in the worker are simply piped to the corresponding streams in the parent thread. This means, for example, that `console.log()` and `console.error()` produce output in exactly the same way in a worker thread as they do in the main thread. You can override this default by passing `stdout:true` or `stderr:true` in the second argument to the `Worker()` constructor. If you do this, then any output the worker writes to those streams becomes readable by the parent thread on the `worker.stdout` and `worker.stderr` threads. (There is a potentially confusing inversion of stream directions here, and we saw the same thing with with child processes earlier in the chapter: the output streams of a worker thread are input streams for the parent thread, and the input stream of a worker is an output stream for the parent.)
- If a worker thread calls `process.exit()`, only the thread exits, not the entire process.
- Worker threads are not allowed to change shared state of the process they are part of. Functions like `process.chdir()` and `process.setuid()` will throw exceptions when invoked from a worker.
- Operating system signals (like `SIGINT` and `SIGTERM`) are only delivered to the main thread; they cannot be received or handled in worker threads.

16.11.3 Communication Channels and MessagePorts

When a new worker thread is created, a communication channel is created along with it that allows messages to be passed back and forth between the worker and the parent thread. As we've seen, the worker thread uses `threads.parentPort` to send and receive messages to and from the parent thread, and the parent thread uses the Worker object to send and receive messages to and from the worker thread.

The worker thread API also allows the creation of custom communication channels using the MessageChannel API defined by web browsers and covered in §15.13.5. If you have read that section, much of what follows will sound familiar to you.

Suppose a worker needs to handle two different kinds of messages sent by two different modules in the main thread. These two different modules could both share the default channel and send messages with `worker.postMessage()`, but it would be cleaner if each module has its own private channel for sending messages to the worker. Or consider the case where the main thread creates two independent workers. A custom communication channel can allow the two workers to communicate directly with each other instead of having to send all their messages via the parent.

Create a new message channel with the `MessageChannel()` constructor. A MessageChannel object has two properties, named `port1` and `port2`. These properties refer to a pair of MessagePort objects. Calling `postMessage()` on one of the ports will cause a "message" event to be generated on the other with a structured clone of the Message object:

```
const threads = require("worker_threads");
let channel = new threads.MessageChannel();
channel.port2.on("message", console.log);  // Log any messages we receive
channel.port1.postMessage("hello");        // Will cause "hello" to be print
```

You can also call `close()` on either port to break the connection between the two ports and to signal that no more messages will be exchanged. When `close()` is called on either port, a "close" event is delivered to both ports.

Note that the code example above creates a pair of MessagePort objects and then uses those objects to transmit a message within the main thread. In order to use custom communication channels with workers, we must transfer one of the two ports from the thread in which it is created to the thread in which it will be used. The next section explains how to do this.

16.11.4 Transferring MessagePorts and Typed Arrays

The `postMessage()` function uses the structured clone algorithm, and as we've noted, it cannot copy objects like SSockets and Streams. It can handle MessagePort objects, but only as a special case using a special technique. The `postMessage()` method (of a Worker object, of `threads.parentPort`, or of any MessagePort object) takes an optional second argument. This argument (called `transferList`) is an array of objects that are to be transferred between threads rather than being copied.

A MessagePort object cannot be copied by the structured clone algorithm, but it can be transferred. If the first argument to `postMessage()` has included one or more MessagePorts (nested arbitrarily deeply within the Message object), then those MessagePort objects must also appear as members of the array passed as the second argument. Doing this tells Node that it does not need to make a copy of the MessagePort, and can instead just give the existing object to the other thread. The key thing to understand, however, about transferring values between threads is that once a value is transferred, it can no longer be used in the thread that called `postMessage()`.

Here is how you might create a new MessageChannel and transfer one of its MessagePorts to a worker:

```
// Create a custom communication channel
const threads = require("worker_threads");
let channel = new threads.MessageChannel();

// Use the worker's default channel to transfer one end of the new
// channel to the worker. Assume that when the worker receives this
// message it immediately begins to listen for messages on the new channel.
worker.postMessage({ command: "changeChannel", data: channel.port1 },
                   [ channel.port1 ]);

// Now send a message to the worker using our end of the custom channel
channel.port2.postMessage("Can you hear me now?");

// And listen for responses from the worker as well
channel.port2.on("message", handleMessagesFromWorker);
```

MessagePort objects are not the only ones that can be transferred. If you call `postMessage()` with a typed array as the message (or with a message that contains one or more typed arrays nested arbitrarily deep within the message), that typed array (or those typed arrays) will simply be copied by the structured clone algorithm. But typed arrays can be large; for example, if you are using a worker thread to do image processing on millions of pixels. So for efficiency, `postMessage()` also gives us the option to transfer typed arrays rather than copying them. (Threads share memory by default. Worker threads in JavaScript generally avoid shared memory, but when we allow this kind of controlled transfer, it can be done very efficiently.) What makes this safe is that when a typed array is transferred to another thread, it becomes unusable in the thread that transferred it. In the image-processing scenario, the main thread could transfer the pixels of an image to the worker thread, and then the worker thread could transfer the processed pixels back to the main thread when it was done. The memory would not need to be copied, but it would never be accessible by two threads at once.

To transfer a typed array instead of copying it, include the ArrayBuffer that backs the array in the second argument to `postMessage()`:

```
let pixels = new Uint32Array(1024*1024);  // 4 megabytes of memory

// Assume we read some data into this typed array, and then transfer the
// pixels to a worker without copying. Note that we don't put the array
// itself in the transfer list, but the array's Buffer object instead.
worker.postMessage(pixels, [ pixels.buffer ]);
```

As with transferred MessagePorts, a transferred typed array becomes unusable once transferred. No exceptions are thrown if you attempt to use a MessagePort or typed array that has been transferred; these objects simply stop doing anything when you interact with them.

16.11.5 Sharing Typed Arrays Between Threads

In addition to transferring typed arrays between threads, it is actually possible to share a typed array between threads. Simply create a SharedArrayBuffer of the desired size and then use that buffer to create a typed array. When a typed array that is backed by a SharedArrayBuffer is passed via `postMessage()`, the underlying memory will be shared between the threads. You should not include the shared buffer in the second argument to `postMessage()` in this case.

You really should not do this, however, because JavaScript was never designed with thread safety in mind and multithreaded programming is very difficult to get right. (And this is why SharedArrayBuffer was not covered in §11.2: it is a niche feature that is difficult to get right.) Even the simple `++` operator is not thread-safe because it needs to read a value, increment it, and write it back. If two threads are incrementing a value at the same time, it will often only be incremented once, as the following code demonstrates:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    // In the main thread, we create a shared typed array with
    // one element. Both threads will be able to read and write
    // sharedArray[0] at the same time.
    let sharedBuffer = new SharedArrayBuffer(4);
    let sharedArray = new Int32Array(sharedBuffer);

    // Now create a worker thread, passing the shared array to it with
    // as its initial workerData value so we don't have to bother with
    // sending and receiving a message
    let worker = new threads.Worker(__filename, { workerData: sharedArray })

    // Wait for the worker to start running and then increment the
    // shared integer 10 million times.
    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;

        // Once we're done with our increments, we start listening for
        // message events so we know when the worker is done.
        worker.on("message", () => {
            // Although the shared integer has been incremented
            // 20 million times, its value will generally be much less.
            // On my computer the final value is typically under 12 million.
            console.log(sharedArray[0]);
        });
    });
} else {
    // In the worker thread, we get the shared array from workerData
    // and then increment it 10 million times.
    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;
    // When we're done incrementing, let the main thread know
    threads.parentPort.postMessage("done");
}
```

One scenario in which it might be reasonable to use a SharedArrayBuffer is when the two threads operate on entirely separate sections of the shared memory. You might enforce this by creating two typed arrays that serve as views of nonoverlapping regions of the shared buffer, and then have your two threads use those two separate typed arrays. A parallel merge sort could be done like this: one thread sorts the bottom half of an array and the other thread sorts the top half, for example. Or some kinds of image-processing algorithms are also suitable for this approach: multiple threads working on disjoint regions of the image.

If you really must allow multiple threads to access the same region of a shared array, you can take one step toward thread safety with the functions defined by the Atomics object. Atomics was added to JavaScript when SharedArrayBuffer was to define atomic operations on the elements of a shared array. For example, the `Atomics.add()` function reads the specified element of a shared array, adds a specified value to it, and writes the sum back into the array. It does this atomically as if it was a single operation, and ensures that no other thread can read or write the value while the operation is taking place. `Atomics.add()` allows us to rewrite the parallel increment code we just looked at and get the correct result of 20 million increments of a shared array element:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    let sharedBuffer = new SharedArrayBuffer(4);
    let sharedArray = new Int32Array(sharedBuffer);
    let worker = new threads.Worker(__filename, { workerData: sharedArray })

    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) {
            Atomics.add(sharedArray, 0, 1);  // Threadsafe atomic increment
        }

        worker.on("message", (message) => {
            // When both threads are done, use a threadsafe function
            // to read the shared array and confirm that it has the
            // expected value of 20,000,000.
            console.log(Atomics.load(sharedArray, 0));
        });
    });
} else {
    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) {
        Atomics.add(sharedArray, 0, 1);       // Threadsafe atomic increment
    }
    threads.parentPort.postMessage("done");
}
```

This new version of the code correctly prints the number 20,000,000. But it is about nine times slower than the incorrect code it replaces. It would be much simpler and much faster to just do all 20 million increments in one thread. Also note that atomic operations may be able to ensure thread safety for image-processing algorithms for which each array element is a value entirely independent of all other values. But in most real-world programs, multiple array elements are often related to one another and some kind of higher-level thread synchronization is required. The low-level `Atomics.wait()` and `Atomics.notify()` function can help with this, but a discussion of their use is out of scope for this book.

16.12 Summary

Although JavaScript was created to run in web browsers, Node has made JavaScript into a general-purpose programming language. It is particularly popular for implementing web servers, but its deep bindings to the operating system mean that it is also a good alternative to shell scripts.

The most important topics covered in this long chapter include:

- Node's asynchronous-by-default APIs and its single-threaded, callback, and event-based style of concurrency.
- Node's fundamental datatypes, buffers, and streams.
- Node's “fs” and “path” modules for working with the filesystem.
- Node's “http” and “https” modules for writing HTTP clients and servers.
- Node's “net” module for writing non-HTTP clients and servers.
- Node's “child_process” module for creating and communicating with child processes.
- Node's “worker_threads” module for true multithreaded programming using message-passing instead of shared memory.

[1] Node defines a `fs.copyFile()` function that you would actually use in practice.

2 It is often cleaner and simpler to define the worker code in a separate file. But this trick of having two threads run different sections of the same file blew my mind when I first encountered it for the Unix `fork()` system call. And I think it is worth demonstrating this technique simply for its strange elegance.