NodeJs流以难以使用而闻名，甚至更难理解。 好吧，我给你带来了好消息 - 现在已经不是这样了。

多年来，开发人员创建了许多软件包，其唯一目的是为了更容易的使用流。 但是，在本文中，我将重点介绍NodJs 原生的流API。

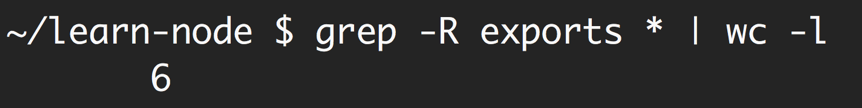
“Streams are Node’s best and most misunderstood idea.”

— Dominic Tarr

Streams究竟是什么？

流是数据的集合 - 就像数组或字符串一样。 不同之处在于流可能无法一次全部可用，并且它们不必适合内存。(译者注：流可以分片处理数据，所以不是一次全部可用，也不用担心数据太大，内存不够) 这使得流在处理大量数据时非常强大，或者一次来自外部源的数据。

但是，流不仅仅是处理大数据。 它们还为我们提供了代码中可组合性的强大功能。 就像我们可以通过管道联结其他较小的Linux命令来组合强大的Linux命令一样，通过使用流，我们在Node中也可以这样做。



const grep = ... // A stream for the grep output

const wc = ... // A stream for the wc input

grep.pipe(wc)

Node中的许多内置模块实现了流接口：

|  |  |
| --- | --- |
| Readable Streams | Writable Streams |
| HTTP response, on the client | HTTP requests, on the client |
| HTTP requests, on the server | HTTP responses, on the server |
| fs read streams | fs write streams |
| zlib streams | zlib streams |
| crypto streams | crypto streams |
| TCP sockets | TCP sockets |
| child process stdout & stderr | |  | | --- | | child process stdin | |
| process.stdin | process.stdout, process.stderr |

上面的列表包含一些原生Node对象的示例，这些对象是可读或可写的流。 其中一些对象是可读写的流，如TCP套接字，zlib和加密流。

请注意，对象也是密切相关的。 虽然HTTP响应是客户端上的可读流，但它是服务器上的可写流。 这是因为在HTTP情况下，我们基本上从一个对象（http.IncomingMessage）读取并写入另一个对象（http.ServerResponse）。

还要注意stdio流（stdin，stdout，stderr）在子进程方面如何具有反向流类型。 这允许使用主进程stdio流以非常简单的方式管理这些子进程stdio流。

一个Streams实例

理论很棒，但往往不是100％令人信服。 让我们看一个示例，演示在内存消耗方面使用流可以产生的差异。

让我们先创建一个大文件：

const fs = require('fs');

const file = fs.createWriteStream('./big.file');

for(let i=0; i<= 1e6; i++) {

file.write('Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.\n');

}

file.end();

看看我用来创建那个大文件的东西。 一个可写的流！

fs模块可用于使用流接口读取和写入文件。 在上面的例子中，我们通过可写流向一个big.file循环写了一百万行。

运行上面的脚本会生成大约约400 MB的文件。

这是一个简单的Node Web服务器，专门用于提供big.file：

const fs = require('fs');

const server = require('http').createServer();

server.on('request', (req, res) => {

fs.readFile('./big.file', (err, data) => {

if (err) throw err;

res.end(data);

});

});

server.listen(8000);

当服务器收到请求时，它将使用异步方法fs.readFile为大文件提供服务。 但是，嘿，这不像我们要阻止事件循环或任何事情。 一切都很棒，对吧？ 对？

好吧，让我们看看当我们运行服务器，连接到它并监视内存时会发生什么。

当我运行服务器时，它开始时具有正常的内存量，8.7 MB：

当我连接上服务器，注意看内存的消耗量，内存使用量达到了400多兆

在将它们写入响应对象之前，我们基本上将整个big.file内容放在内存中。 这是非常低效的。

HTTP响应对象（上面代码中的res）也是可写流。 这意味着如果我们有一个表示big.file内容的可读流，我们可以将这两个流对象通过管道连接，并实现大致相同的结果，而不会消耗~400 MB的内存。

Node的fs模块可以使用createReadStream方法为任何文件提供可读的流。 我们可以将它传递给响应对象：

const fs = require('fs');

const server = require('http').createServer();

server.on('request', (req, res) => {

const src = fs.createReadStream('./big.file');

src.pipe(res);

});

server.listen(8000);

现在，当你连接到这个服务器时，会发生一件神奇的事情（看一下内存消耗）：

发生了什么？

当客户端请求该大文件时，我们一次流一个块，这意味着我们根本不将它缓冲在内存中。 内存使用量增长了大约25 MB，就是这样。

您可以将此示例推到极限。 使用500万行而不是仅仅100万行重新生成big.file，这将使文件超过2 GB，并且实际上大于Node中的默认缓冲区限制。

如果您尝试使用fs.readFile提供该文件，则默认情况下不能（您可以更改限制）。 但是使用fs.createReadStream，向请求者传输2 GB数据是没有问题的，最重要的是，进程内存使用情况大致相同。

准备好学习流了吗？

流101

Node中有四种基本流类型：可读，可写，双工和转换流。

可读流是可以从中消耗数据的源的抽象。一个例子是fs.createReadStream方法。

可写流是可以写入数据的目标的抽象。一个例子是fs.createWriteStream方法。

双工流是可读和可写的。一个例子是TCP套接字。

变换流基本上是双工流，可用于在数据写入和读取时修改或转换数据。一个例子是使用gzip压缩数据的zlib.createGzip流。您可以将转换流视为一个函数，其中输入是可写流部分，输出是可读流部分。您可能还会听到称为“通过流”(passthrough)的变换流。

所有流都是EventEmitter的实例。它们发出可用于读取和写入数据的事件。但是，我们可以使用管道方法以更简单的方式使用流数据。

管道方法

这是你需要记住的一行魔法代码：

readableSrc.pipe（writableDest）;

在这个简单的行中，我们将可读流的输出(数据源) - 作为可写流的输入(作为目标)。源必须是可读流，目标必须是可写的。当然，它们也可以是双工/变换流。事实上，如果我们pipe的目标是双工流，我们可以像在Linux中一样链接管道调用：

readableSrc

.pipe(transformStream1)

.pipe(transformStream2)

.pipe(finalWrtitableDest)

pipe方法返回目标流，这使我们能够进行上面的链接。对于流a（可读），b和c（双工）以及d（可写），我们可以：a.pipe(b).pipe(c).pipe(d)

//这相当于：

# Which is equivalent to:

a.pipe(b)

b.pipe(c)

c.pipe(d)

# Which, in Linux, is equivalent to:

$ a | b | c | d

管道方法是消费流的最简单方法。通常建议使用管道方法或使用事件消耗流，但避免混合这两者。通常，当您使用管道方法时，您不需要使用事件，但如果您需要以更自定义的方式使用流，则事件将是可行的方法。

流事件

除了从可读流源读取并写入可写目标之外，管道方法还会自动管理一些事情。 例如，它处理错误，文件结束以及一个流比另一个流更慢或更快的情况。

但是，流也可以直接与事件一起使用。 这是管道方法主要用于读取和写入数据的简化事件等效代码：

# readable.pipe(writable)

readable.on('data', (chunk) => {

writable.write(chunk);

});

readable.on('end', () => {

writable.end();

});

以下是可与可读写流一起使用的重要事件和方法的列表：

|  |  |  |
| --- | --- | --- |
|  | Readable Streams | Writable Streams |
| ****Events**** | data, end, error, close, readable | drain, finish, error, close, pipe, unpipe |
| ****Methods**** | pipe(), unpipe(), wrap(), destroy() | write(), destroy(), end() |
|  | read(), unshift(), resume(), pause(), isPaused(), setEncoding() | cork(), uncork(), setDefaultEncoding() |

上面列表中的事件和方法在某种程度上是相关的，因为它们通常一起使用。

可读流上最重要的事件是：

数据事件，只要流将一大块数据传递给使用者，就会发出该事件

结束事件，当没有更多数据要从流中消耗时发出。

可写流上最重要的事件是：

排空事件，是可写流可以接收更多数据的信号。

完成事件，在将所有数据刷新到底层系统时发出。

可以组合事件和功能以实现流的自定义和优化使用。 要使用可读流，我们可以使用pipe / unpipe方法或read / unshift / resume方法。 要使用可写流，我们可以将它作为pipe / unpipe的目标，或者只使用write方法写入它，并在完成后调用end方法。

Events and functions can be combined to make for a custom and optimized use of streams. To consume a readable stream, we can use the pipe/unpipe methods, or the read/unshift/resume methods. To consume a writable stream, we can make it the destination of pipe/unpipe, or just write to it with the write method and call the end method when we’re done.

Paused and Flowing Modes of Readable Streams

Readable streams have two main modes that affect the way we can consume them:

They can be either in the paused mode

Or in the flowing mode

Those modes are sometimes referred to as pull and push modes.

All readable streams start in the paused mode by default but they can be easily switched to flowing and back to paused when needed. Sometimes, the switching happens automatically.

When a readable stream is in the paused mode, we can use the read() method to read from the stream on demand, however, for a readable stream in the flowing mode, the data is continuously flowing and we have to listen to events to consume it.

In the flowing mode, data can actually be lost if no consumers are available to handle it. This is why, when we have a readable stream in flowing mode, we need a data event handler. In fact, just adding a data event handler switches a paused stream into flowing mode and removing the data event handler switches the stream back to paused mode. Some of this is done for backward compatibility with the older Node streams interface.

To manually switch between these two stream modes, you can use the resume() and pause() methods.

Screenshot captured from my Pluralsight course — Advanced Node.js

When consuming readable streams using the pipe method, we don’t have to worry about these modes as pipe manages them automatically.

Implementing Streams

When we talk about streams in Node.js, there are two main different tasks:

The task of implementing the streams.

The task of consuming them.

So far we’ve been talking about only consuming streams. Let’s implement some!

Stream implementers are usually the ones who require the stream module.

Implementing a Writable Stream

To implement a writable stream, we need to to use the Writable constructor from the stream module.

const { Writable } = require('stream');

We can implement a writable stream in many ways. We can, for example, extend the Writable constructor if we want

class myWritableStream extends Writable {

}

However, I prefer the simpler constructor approach. We just create an object from the Writable constructor and pass it a number of options. The only required option is a write function which exposes the chunk of data to be written.

const { Writable } = require('stream');

const outStream = new Writable({

write(chunk, encoding, callback) {

console.log(chunk.toString());

callback();

}

});

process.stdin.pipe(outStream);

This write method takes three arguments.

The chunk is usually a buffer unless we configure the stream differently.

The encoding argument is needed in that case, but usually we can ignore it.

The callback is a function that we need to call after we’re done processing the data chunk. It’s what signals whether the write was successful or not. To signal a failure, call the callback with an error object.

In outStream, we simply console.log the chunk as a string and call the callback after that without an error to indicate success. This is a very simple and probably not so useful echo stream. It will echo back anything it receives.

To consume this stream, we can simply use it with process.stdin, which is a readable stream, so we can just pipe process.stdin into our outStream.

When we run the code above, anything we type into process.stdin will be echoed back using the outStream console.log line.

This is not a very useful stream to implement because it’s actually already implemented and built-in. This is very much equivalent to process.stdout. We can just pipe stdin into stdout and we’ll get the exact same echo feature with this single line:

process.stdin.pipe(process.stdout);

Implement a Readable Stream

To implement a readable stream, we require the Readable interface, and construct an object from it, and implement a read() method in the stream’s configuration parameter:

const { Readable } = require('stream');

const inStream = new Readable({

read() {}

});

There is a simple way to implement readable streams. We can just directly push the data that we want the consumers to consume.

const { Readable } = require('stream');

const inStream = new Readable({

read() {}

});

inStream.push('ABCDEFGHIJKLM');

inStream.push('NOPQRSTUVWXYZ');

inStream.push(null); // No more data

inStream.pipe(process.stdout);

When we push a null object, that means we want to signal that the stream does not have any more data.

To consume this simple readable stream, we can simply pipe it into the writable stream process.stdout.

When we run the code above, we’ll be reading all the data from inStream and echoing it to the standard out. Very simple, but also not very efficient.

We’re basically pushing all the data in the stream before piping it to process.stdout. The much better way is to push data on demand, when a consumer asks for it. We can do that by implementing the read() method in the configuration object:

const inStream = new Readable({

read(size) {

// there is a demand on the data... Someone wants to read it.

}

});

When the read method is called on a readable stream, the implementation can push partial data to the queue. For example, we can push one letter at a time, starting with character code 65 (which represents A), and incrementing that on every push:

const inStream = new Readable({

read(size) {

this.push(String.fromCharCode(this.currentCharCode++));

if (this.currentCharCode > 90) {

this.push(null);

}

}

});

inStream.currentCharCode = 65;

inStream.pipe(process.stdout);

While the consumer is reading a readable stream, the read method will continue to fire, and we’ll push more letters. We need to stop this cycle somewhere, and that’s why an if statement to push null when the currentCharCode is greater than 90 (which represents Z).

This code is equivalent to the simpler one we started with but now we’re pushing data on demand when the consumer asks for it. You should always do that.

Implementing Duplex/Transform Streams

With Duplex streams, we can implement both readable and writable streams with the same object. It’s as if we inherit from both interfaces.

Here’s an example duplex stream that combines the two writable and readable examples implemented above:

const { Duplex } = require('stream');

const inoutStream = new Duplex({

write(chunk, encoding, callback) {

console.log(chunk.toString());

callback();

},

read(size) {

this.push(String.fromCharCode(this.currentCharCode++));

if (this.currentCharCode > 90) {

this.push(null);

}

}

});

inoutStream.currentCharCode = 65;

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

By combining the methods, we can use this duplex stream to read the letters from A to Z and we can also use it for its echo feature. We pipe the readable stdin stream into this duplex stream to use the echo feature and we pipe the duplex stream itself into the writable stdout stream to see the letters A through Z.

It’s important to understand that the readable and writable sides of a duplex stream operate completely independently from one another. This is merely a grouping of two features into an object.

A transform stream is the more interesting duplex stream because its output is computed from its input.

For a transform stream, we don’t have to implement the read or write methods, we only need to implement a transform method, which combines both of them. It has the signature of the write method and we can use it to push data as well.

Here’s a simple transform stream which echoes back anything you type into it after transforming it to upper case format:

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

const upperCaseTr = new Transform({

transform(chunk, encoding, callback) {

this.push(chunk.toString().toUpperCase());

callback();

}

});

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

In this transform stream, which we’re consuming exactly like the previous duplex stream example, we only implemented a transform() method. In that method, we convert the chunk into its upper case version and then push that version as the readable part.

Streams Object Mode

By default, streams expect Buffer/String values. There is an objectMode flag that we can set to have the stream accept any JavaScript object.

Here’s a simple example to demonstrate that. The following combination of transform streams makes for a feature to map a string of comma-separated values into a JavaScript object. So “a,b,c,d” becomes {a: b, c: d}.

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

const commaSplitter = new Transform({

readableObjectMode: true,

transform(chunk, encoding, callback) {

this.push(chunk.toString().trim().split(','));

callback();

}

});

const arrayToObject = new Transform({

readableObjectMode: true,

writableObjectMode: true,

transform(chunk, encoding, callback) {

const obj = {};

for(let i=0; i < chunk.length; i+=2) {

obj[chunk[i]] = chunk[i+1];

}

this.push(obj);

callback();

}

});

const objectToString = new Transform({

writableObjectMode: true,

transform(chunk, encoding, callback) {

this.push(JSON.stringify(chunk) + '\n');

callback();

}

});

process.stdin

.pipe(commaSplitter)

.pipe(arrayToObject)

.pipe(objectToString)

.pipe(process.stdout)

We pass the input string (for example, “a,b,c,d”) through commaSplitter which pushes an array as its readable data ([“a”, “b”, “c”, “d”]). Adding the readableObjectMode flag on that stream is necessary because we’re pushing an object there, not a string.

We then take the array and pipe it into the arrayToObject stream. We need a writableObjectMode flag to make that stream accept an object. It’ll also push an object (the input array mapped into an object) and that’s why we also needed the readableObjectMode flag there as well. The last objectToString stream accepts an object but pushes out a string, and that’s why we only needed a writableObjectMode flag there. The readable part is a normal string (the stringified object).

Usage of the example above

Node’s built-in transform streams

Node has a few very useful built-in transform streams. Namely, the zlib and crypto streams.

Here’s an example that uses the zlib.createGzip() stream combined with the fs readable/writable streams to create a file-compression script:

const fs = require('fs');

const zlib = require('zlib');

const file = process.argv[2];

fs.createReadStream(file)

.pipe(zlib.createGzip())

.pipe(fs.createWriteStream(file + '.gz'));

You can use this script to gzip any file you pass as the argument. We’re piping a readable stream for that file into the zlib built-in transform stream and then into a writable stream for the new gzipped file. Simple.

The cool thing about using pipes is that we can actually combine them with events if we need to. Say, for example, I want the user to see a progress indicator while the script is working and a “Done” message when the script is done. Since the pipe method returns the destination stream, we can chain the registration of events handlers as well:

const fs = require('fs');

const zlib = require('zlib');

const file = process.argv[2];

fs.createReadStream(file)

.pipe(zlib.createGzip())

.on('data', () => process.stdout.write('.'))

.pipe(fs.createWriteStream(file + '.zz'))

.on('finish', () => console.log('Done'));

So with the pipe method, we get to easily consume streams, but we can still further customize our interaction with those streams using events where needed.

What’s great about the pipe method though is that we can use it to compose our program piece by piece, in a much readable way. For example, instead of listening to the data event above, we can simply create a transform stream to report progress, and replace the .on() call with another .pipe() call:

const fs = require('fs');

const zlib = require('zlib');

const file = process.argv[2];

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

const reportProgress = new Transform({

transform(chunk, encoding, callback) {

process.stdout.write('.');

callback(null, chunk);

}

});

fs.createReadStream(file)

.pipe(zlib.createGzip())

.pipe(reportProgress)

.pipe(fs.createWriteStream(file + '.zz'))

.on('finish', () => console.log('Done'));

This reportProgress stream is a simple pass-through stream, but it reports the progress to standard out as well. Note how I used the second argument in the callback() function to push the data inside the transform() method. This is equivalent to pushing the data first.

The applications of combining streams are endless. For example, if we need to encrypt the file before or after we gzip it, all we need to do is pipe another transform stream in that exact order that we needed. We can use Node’s crypto module for that:

const crypto = require('crypto');

// ...

fs.createReadStream(file)

.pipe(zlib.createGzip())

.pipe(crypto.createCipher('aes192', 'a\_secret'))

.pipe(reportProgress)

.pipe(fs.createWriteStream(file + '.zz'))

.on('finish', () => console.log('Done'));

The script above compresses and then encrypts the passed file and only those who have the secret can use the outputted file. We can’t unzip this file with the normal unzip utilities because it’s encrypted.

To actually be able to unzip anything zipped with the script above, we need to use the opposite streams for crypto and zlib in a reverse order, which is simple:

fs.createReadStream(file)

.pipe(crypto.createDecipher('aes192', 'a\_secret'))

.pipe(zlib.createGunzip())

.pipe(reportProgress)

.pipe(fs.createWriteStream(file.slice(0, -3)))

.on('finish', () => console.log('Done'));

Assuming the passed file is the compressed version, the code above will create a read stream from that, pipe it into the crypto createDecipher() stream (using the same secret), pipe the output of that into the zlib createGunzip() stream, and then write things out back to a file without the extension part.

That’s all I have for this topic. Thanks for reading! Until next time!

Learning React or Node? Checkout my books:

React Beyond the Basics

Node Beyond the Basics