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。

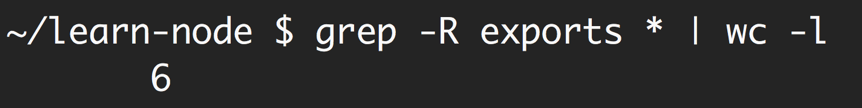
“Streams are Node’s best and most misunderstood idea.”

— 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。

暂停和流动模式

可读流有两种主要模式影响我们使用它们的方式：

它们可以处于暂停模式

或者在流动模式下

这些模式有时被称为pull和push模式。

默认情况下，所有可读流都以暂停模式启动，但可以在需要时轻松切换为流动并返回暂停状态。有时，切换会自动进行。

当可读流处于暂停模式时，我们可以使用read（）方法根据需要从流中读取。但是，对于流动模式下的可读流，数据不断流动，我们必须监听事件以消耗它。

在流动模式下，如果没有消费者可以处理数据，实际上可能会丢失数据。这就是为什么当我们在流动模式下有可读流时，我们需要一个数据事件处理程序。实际上，只需添加数据事件处理程序即可将暂停的流切换为流动模式，并删除数据事件处理程序会将流切换回暂停模式。其中一些是为了与旧的Node流接口向后兼容而完成的。

要在这两种流模式之间手动切换，可以使用resume（）和pause（）方法。

TIP: 当使用管道方法消耗可读流时，我们不必担心这些模式，因为管道会自动管理它们。

实现可写流

要实现可写流，我们需要使用流模块中的Writable构造函数。

const {Writable} = require（“stream”）;

我们可以通过多种方式实现可写流。例如，如果我们需要，我们可以扩展Writable构造函数：

class myWritableStream extends Writable {}

但是，我更喜欢更简单的构造方法。我们只是从Writable构造函数创建一个对象，并传递了许多选项。唯一需要的选项是write函数，它公开要写入的数据块。

const {Writable} = require（“stream”）;

const outStream = new Writable（{

  write（chunk，encoding，callback）{

    console.log（chunk.toString（））;

    callback();  }

}）;

process.stdin.pipe（outStream）;

这个write方法有三个参数。

除非我们以不同方式配置流，否则块通常是缓冲区。

在这种情况下需要编码参数，但我们通常可以忽略它。

回调是我们在处理完数据块后需要调用的函数。这就是写入是否成功的信号。要发出故障信号，请使用错误对象调用回调。

在outStream中，我们只需将chunk控制为字符串，然后在没有错误的情况下调用回调以指示成功。这是一个非常简单且可能不那么有用的回声流。它将回应它收到的任何东西。

要使用这个流，我们可以简单地将它与process.stdin一起使用，这是一个可读的流，所以我们可以将process.stdin管道输入到我们的outStream中。

当我们运行上面的代码时，我们在process.stdin中键入的任何内容都将使用outStream console.log行回显。

这不是一个非常有用的实现流，因为它实际上已经实现和内置。这与process.stdout非常相同。我们可以将stdin管道输入stdout，我们将使用这一行获得完全相同的echo功能：

process.stdin.pipe（process.stdout）;

实现可读流

要实现可读流，我们需要Readable接口并从中构造一个对象：

const {Readable} = require（“stream”）;

const inStream = new Readable（{}）;

有一种实现可读流的简单方法。我们可以直接推送我们希望消费者使用的数据。

const {Readable} = require（“stream”）;

const inStream = new Readable（）;

inStream.push（ “ABCDEFGHIJKLM”）;

inStream.push（ “NOPQRSTUVWXYZ”）;

inStream.push（NULL）; //没有更多数据

inStream.pipe（process.stdout）;

当我们推送一个空对象时，这意味着我们想要发信号通知该流没有任何更多的数据。

要使用这个简单的可读流，我们可以简单地将其传递到可写流process.stdout中。

当我们运行上面的代码时，我们将从inStream读取所有数据并将其回显到标准输出。很简单，但也不是很有效率。

我们基本上推送流中的所有数据，然后将其传递给process.stdout。当消费者要求时，更好的方法是按需推送数据。我们可以通过在可读流配置中实现read（）方法来实现：

const inStream = new Readable({

read(size) {

// there is a demand on the data...

// Someone wants to read it.

}

});

当在可读流上调用read方法时，实现可以将部分数据推送到队列。例如，我们可以一次推送一个字母，从字符代码65（代表A）开始，并在每次推送时递增代码：

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);

当消费者正在阅读可读流时，read方法将继续触发，我们将推送更多字母。我们需要在某个地方停止这个循环，这就是为什么我在currentCharCode大于90（表示Z）时使用if语句来推送null。

这段代码相当于我们开始使用的代码更简单，但现在我们在消费者要求时按需推送数据。你应该永远这样做。

实现双工/转换流

使用Duplex流，我们可以使用相同的对象实现可读和可写流。就像我们从两个接口继承一样。

这是一个示例双工流，它结合了上面实现的两个可写和可读示例：

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);

通过组合这些方法，我们可以使用此双工流来读取从A到Z的字母，我们也可以将其用于其回声功能。我们将可读的stdin流传输到此双工流中以使用echo功能，我们将双工流本身传输到可写的stdout流中以查看字母A到Z.

重要的是要理解双工流的可读和可写侧完全独立地操作。这仅仅是将两个特征分组到一个对象中。

变换流是更有趣的双工流，因为其输出是根据其输入计算的。

对于转换流，我们不必实现读取或写入方法，我们只需要实现一个转换方法，它将两者结合起来。它具有write方法的签名，我们也可以用它来“推送”数据。

这是一个简单的变换流，它将您转换为大写格式后输入的任何内容回送回来：

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);

在这个我们正在消耗的变换流中，就像前面的双工流示例一样，我们只实现了transform（）方法。在该方法中，我们将块转换为其大写版本，然后将该版本作为可读部分推送。

流对象模式

默认情况下，流期望缓冲区/字符串值。我们可以设置一个objectMode标志，让流接受任何JavaScript对象。

这是一个简单的例子来证明这一点。以下转换流组合创建了一个将逗号分隔值字符串映射到JavaScript对象的功能。所以“a，b，c，d”变成{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();

}

});

然后，使用这些流：

process.stdin

.pipe(commaSplitter)

.pipe(arrayToObject)

.pipe(objectToString)

.pipe(process.stdout);

我们通过commaSplitter传递输入字符串（例如，“a，b，c，d”），它将数组推送为可读数据（[“a”，“b”，“c”，“d”]）。在该流上添加readableObjectMode标志是必要的，因为我们在那里推送一个对象，而不是字符串。

然后我们获取数组并将其传递到arrayToObject流中。我们需要一个writableObjectMode标志来使该流接受一个对象。它还会推送一个对象（映射到对象的输入数组），这就是为什么我们也需要那里的readableObjectMode标志。最后一个objectToString流接受一个对象，但是推出一个字符串，这就是为什么我们只需要一个writableObjectMode标志。可读部分是普通字符串（字符串化对象）。

内置变换流

Node有一些非常有用的内置变换流，例如zlib`和`crypto streams。

这是一个使用zlib.createGzip（）流与fs可读/可写流组合创建文件压缩脚本的示例：

const fs = require("fs");

const zlib = require("zlib");

const file = process.argv[2];

fs.createReadStream(file)

.pipe(zlib.createGzip())

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

您可以使用此脚本将您传递的作为参数的任何文件进行gzip压缩。我们将该文件的可读流传输到zlib内置转换流中，然后传输到新gzip压缩文件的可写流中。简单。

使用管道的一个很酷的事情是，如果需要，我们实际上可以将它们与事件结合起来。比如说，我希望用户在脚本运行时看到进度指示器，在脚本完成时看到“完成”消息。由于pipe方法返回目标流，我们也可以链接事件处理程序的注册：

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"));****

使用管道方法，我们可以轻松地使用流，但我们仍然可以使用需要的事件进一步自定义与这些流的交互。

然而，管道方法的优点是我们可以用一种可读的方式逐个编写我们的程序。例如，我们可以简单地创建一个转换流来报告进度，并用另一个.pipe（）调用替换.on（）调用，而不是监听上面的数据事件：

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"));

此reportPressress流是一个简单的传递流，但它也会将进度报告给标准输出。注意我是如何使用callback（）函数中的第二个参数来推送transform（）方法中的数据的。这相当于首先推送数据。

组合流的应用是无止境的。例如，如果我们需要在gzip之前或之后加密文件，我们需要做的就是按照我们需要的确切顺序管道另一个转换流。我们可以使用Node的加密模块：

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"));

上面的脚本压缩然后加密它作为参数接收的文件。只有拥有秘密的人才能使用输出的文件。我们无法使用普通的解压缩实用程序解压缩此文件，因为它已加密。

要实际能够解压缩上面脚本压缩的任何内容，我们需要以相反的顺序使用相反的流加密和zlib，这很简单：

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"));

假设使用的文件是压缩版本，上面的代码将从中创建一个读取流，将其传递到crypto createDecipher（）流（使用相同的秘密），将其输出传递到zlib createGunzip（）流中，并且然后把东西写回没有扩展部分的文件。