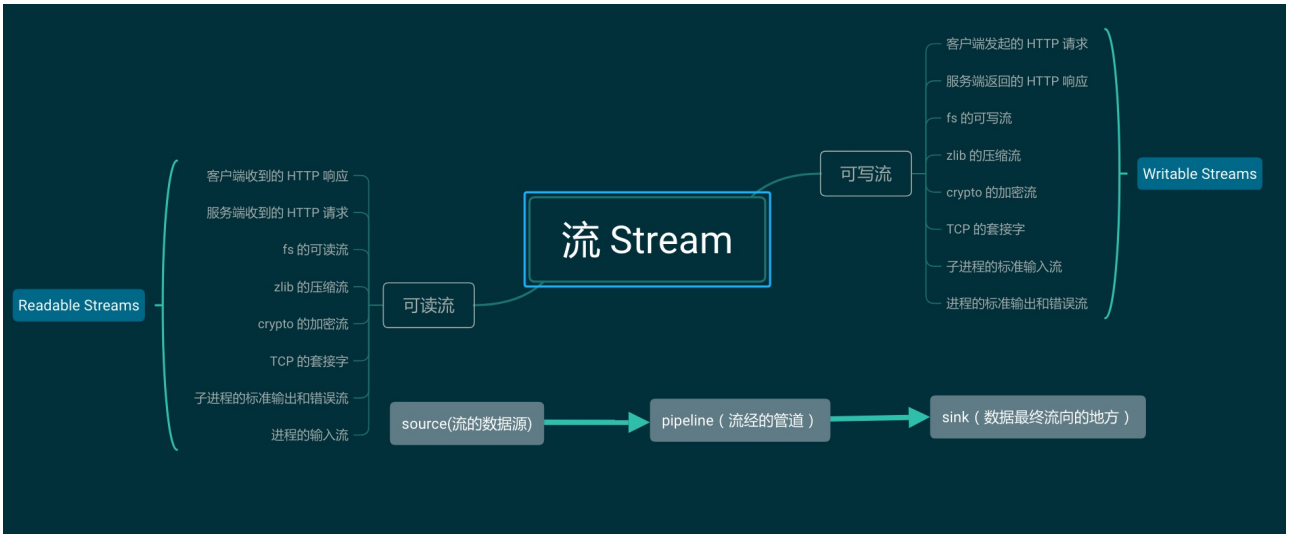
[视频流转 MP3 工具] Node 数据流与管道 - Stream/pipe

本节目标：【实现一个视频流转 MP3 工具】 - 数据界的顺丰，把一个一个数据包运往九州各地。



Stream 流

流是 UNIX 系统中的一个标准概念，很多场景都用到了流，比如标准输入输出，比如管道符命令 `cat *.js | grep fs`，来打印出所有的 js 文件，然后交给 grep 来过滤出包含 fs 的文件内容，这个竖线就是 unix 的管道。

我们已经知道，Buffer 是一个临时的内存缓冲区，用来保存原始二进制数据，而流就是移动数据，它俩通常结合起来用，我们要拷贝一份文件，比如上面例子里面，拷贝一个小的 logo 图片，这种方式是把文件内容全部读入内存中，然后再写入到文件，对于小体积的文件是 ok 的，但是对于体积较大的文件，比如视频，如果使用这种方法，内存可能就不够用了，如果此时有多个请求都在请求和文件，服

务器需要不断的读取某个文件，每个请求都会有一份内存保存文件，内存也很容易爆掉，所以最好是能做到边读边写，读一部分写一部分而不是一口吃成大胖子，这就要借助流来完成。

流是一个抽象接口，在 Nodejs 有很多模块都用到流，比如：

- fs 文件系统模块：可读可写流，如 createReadStream/createWriteStream
- net 底层的网络通信模块：处理通信的双工流 如 connect/tcp/socket
- crypto 加密解密模块：各种算法类的加密流，如 Hmac/Cipher/Hash
- http 网络模块：请求响应流，如 request/response
- process 进程模块：输入输出流，如 stdin/stuout/stderr
- zlib 压缩模块：各种压缩解压流，如 createGzip/createGunzip/createDeflate/createInflate

流的应用范围这么广，所以我们有必要了解流的概念以及使用，这对于我们理解 HTTP，以及运用 Nodejs 很有帮助。

四大流王

流里面也有不同的分工，我们先看下四大流王都是什么和能做什么?

Node 里面的流有四种，分别是 readable/writable/duplextransform，它们的意义如下：

- Readable Stream 是可读流，用来提供数据，外部来源的数据会被存储到内部的 Buffer 数组内缓存起来，可读流有两个模式，分别是 pause（暂停）和 resume（流动），顾名思义，流动模式会源源不断把数据读进来缓存，暂停则按兵不动，不去获取数据也不会积累缓存。
- Writeable Stream 是可写流，用来消费数据，从可读流中获取数据，对拿到的 Buffer 数据进行处理消耗，把它写入目标对

象，它有一个 drain 事件，来判定是否当前的缓存数据写入完毕。
- Duplex streams 也叫双工流，文武双全可直可弯，即是 Readable 也是 Writable，比如 Tcp socket 是可读可写的双工流，另外 zlib/crypto 也都实现了双工流。
- Transform streams 是转换流，它本身也是双工流，只是输出和输入有一定的关联关系，它通常不保存数据，只负责处理和加工流经它的数据，可以把它想象成一个水管的阀门控制器或者消毒器这样的中间件，在 Node 里面，zlib/crypto 实现了转换流。

四大流都有各自不同的能力，但总体上的特征类似，都是对缓冲的数据进行进出处理，他们结合起来使用会非常编写，而把他们结合起来，我们通常的做法是通过 pipe 管道来连接或者反转，管道的部分我们得会来讲，先看下流本身具备的一些状态，也就是跟事件的结合。

流与事件

我们上一节有尝试借助 Buffer 来复制图片的代码，再用同步的方式改造一下：

```
const source = fs.readFileSync('img.png')

fs.writeFileSync('img_copy.png', source)
```

当然也可以通过 fs.copyFile 来实现，只不过这样过于简单粗暴，实际上数据在流动过程中，应该有一些精细的传输阶段或者状态的，这些状态呢，是通过 EventEmitter 控制，而流 Stream 是 EventEmitter 的实例，它是基于事件机制运作的，也就是说在流对象上，可以监听事件可以触发事件，流在各个阶段的变化我们都可以实时监听到，从而实现更精细的控制，比如暂停和恢复，我们再来上一段读取 mp3 的代码：

```
const fs = require('fs')
// 创建一个可读流，把内容从目标文件里一块一块抠出来缓存
const rs = fs.createReadStream('./myfile.mp3')
let n = 0
rs
  // 数据正在传递时，触发该事件（以 chunk 数据块为对象）
  // 每次 chunk 块最大是 64kb，如果凑不够 64kb，会缩小为 32kb
  .on('data', (chunk) => {
    // 记录一共获取到了多少次 chunk
    n++
    console.log(chunk.byteLength)
    console.log(Buffer.isBuffer(chunk))
    // console.log('data emits')
    // console.log(chunk.toString('utf8'))
    // 我们可以每次都暂停数据读取，做一些数据中间处理（比如压缩）后再继续读取数据
    rs.pause()
    console.log('暂停获取....')
    setTimeout(() => {
      console.log('继续获取....', n + 1)
      rs.resume()
    }, 100)
  })
  // 数据传递完成后，会触发 'end' 事件
  .on('end', () => {
    console.log(`传输结束，共收到 ${n} 个 Buffer 块`)
  })
  // 整个流传输结束关闭的时候会触发 close
  .on('close', () => {
    console.log('传输关闭')
  })
  // 异常中断或者出错时的回调处理
```

```
  .on('error', (e) => {
     console.log('传输出错' + e)
  })
```

打印的结果类似这样：

```
65536
true
暂停获取....
继续获取.... 2
65536
true
...
暂停获取....
继续获取.... 107
65536
true
暂停获取....
继续获取.... 108
23428
true
暂停获取....
传输结束，共收到 108 个 Buffer 块
传输关闭
继续获取.... 109
```

发现我本地的这个 mp3 文件，读取过程中，一共读取到 109 块缓冲，一开始每次都是 64kb，最后一次剩余了不到 20kb 的数据，每次读进来的都是 Buffer，而且读取过程中可以暂停也可以再恢复。

流速控制

了解了 stream 的暂停能力和事件特征，我们可以再次重构下复制图片的代码：

```
const rs = fs.createReadStream('./logo.png')
const ws =
fs.createWriteStream('./logo_write.png')

rs.on('data', (chunk) => {
  // 当有数据流出时，写入数据
  ws.write(chunk)
})
rs.on('end', () => {
  // 当没有数据时，关闭数据流
  ws.end()
})
```

这是基于流机制实现的文件拷贝，它存在这样一个问题，如果读的快，写的慢，因为磁盘 IO 的读写速度并不是一致的，如果读的快，写得慢，积压的内存缓冲越来越多，内存可能会爆仓，那应该怎么办呢?

幸运的是，在 stream 里面，流的 write 方法会有一个返回值，它告诉我们传入的数据是否依然停留在缓存区，再根据 drain 事件判断是否缓存数据写入目标了，就可以继续恢复，来写入下一个数据缓存了，这样来改造下：

```
const fs = require('fs')
const rs = fs.createReadStream('./logo.png')
const ws =
fs.createWriteStream('./logo_write_safe.png')

rs.on('data', (chunk) => {
  // 看看是否缓冲数据被写入，写入是 true，未写入是 false
  if (ws.write(chunk) === false) {
    console.log('still cached')
    rs.pause()
  }
})
rs.on('end', () => {
  // 当没有数据再消耗后，关闭数据流
  ws.end()
})
ws.on('drain', () => {
  console.log('数据被消耗后，继续启动读数据')
  rs.resume()
})
```

这样就简陋的实现了防爆仓，越是大的文件越需要优雅的处理。

流的数据管道 - pipe

无论是哪一种流，都会使用 pipe() 方法来实现输入和输出，pipe 的左边是是流，右边也是流，左边读出的数据，经过 pipe 输送给右边的目标流，目标流经过处理后，可以继续往下不断的 pipe，从而形成一个 pipe 链条，小水管就全部串起来了。

对于 pipe 方法，我们来举两个例子，第一个例子是依然是复制图片，我们可以这样做：

```
fs.createReadStream('./logo.png')
  .pipe(fs.createWriteStream('./logo-pipe.png'))
```

一句代码就能搞定复制，非常强大，再来看第二个例子，从浏览器向服务器请求一个非常大的文本文件，大家可以在本地存一个大于 5MB 的文本文件，然后跑如下代码：

```
// request-txt.js
const fs = require('fs')
const http = require('http')

http.createServer((req, res) => {
  res.writeHeader(200, {'Context-Type':
'text/html'})
  // 1. 把文件内容全部读入内存
  fs.readFile('./big.txt', (err, data) => {
    // 2. 通过 res 批量返回
    res.end(data)
  })
}).listen(5000)
```

会发现客户端需要等待一段时间才能看到数据，我们用 pipe 改写下再观察下页面内容呈现的速度：

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

http.createServer((req, res) => {
  res.writeHeader(200, {'Context-Type':
'text/html'})
  fs.createReadStream('./big.txt').pipe(res)
}).listen(5000)
```

明显这个展现速度加快许多，内容是一片片出来的，原因就在于 pipe 会自动监听 data 和 end 事件，文件中的每一小段数据都会源源不断的发送给客户端，pipe 方法 还可以自动控制后端压力，在客户端连接缓慢的时候 Node 可以将尽可能少的缓存放到内存中，通过对内存空间的调度，就能自动控制流量从而避免目标被快速读取的可读流所淹没，并且，数据在 pipe 的时候，只有 pipe 链末端的目标流真正需要数据的时候，数据才会从源头被取出来，然后顺着管子一路走下去，属于被动消费，那么整体表现就会更优异一些。

OK, 关于 pipe，有许多有趣的玩法，我们对流和 pipe 简单总结一下：

- 可读流负责获取外部数据，并把外部数据缓存到内部 Buffer 数组
- 可写流负责消费数据，从可读流中获取到数据，然后对得到的 chunk
- 数据块进行处理，至于如何处理，就取决于于这个可写流内部 write 方法如何实现
- pipe 会自动控制数据的读取速度，来帮助数据以一种比较合理的速度，源源不断的输送给目的地

定制流

Node 除了提供各种流，还提供了流的接口，来定制我们自己的流方案，这些接口实例也拥有各种和流交互的方法，比如：

```
// 获取流很简单，require 即可
const Readable = require('stream').Readable
const Writeble = require('stream').Writable
// require 后，可以来创建流实例
const rs = new Readable()
const ws = new Writeble()

// 流实例创建后，比如是 stream
// 可以往流里面推送一个 chunk 数据
stream.push()
// 推送 null 来告诉流可以 close 了
stream.push(null)
// 流异常时候发出一个异常事件
stream.emit('error', error)

// 告诉流可以继续消费数据了
stream.resume()
// 告诉流先暂停
stream.pause()
// 每次有数据过来，都会流经这个回调函数
stream.on('data', data => {})

// 监听流异常事件，调用回调函数
stream.on('error', (err) => {})
// 监听流关闭事件，调用回调函数
stream.on('close', () => {})
// 监听流完成事件，调用回调函数
stream.on('finish', () => {})
```

流实例的交互能力很完整了，我们来实现一个搬运字符串的小例子：

```
// 拿到 stream 里面的可读可写流接口
const Readable = require('stream').Readable
const Writeble = require('stream').Writable
const rs = new Readable()
const ws = new Writeble()
let n = 0

// 一次次往流里面推数据
rs.push('I ')
rs.push('Love ')
rs.push('Juejin!\n')
rs.push(null)

// 每一次 push 的内容在 pipe 的时候
// 都会走到 _write 方法，在 _write 里面可以再做处理
ws._write = function(chunk, ev, cb) {
  n++
  console.log('chunk' + n + ': ' +
chunk.toString())
  // chunk1: I
  // chunk2: Love
  // chunk3: Juejin!
  cb()
}

// pipe 将两者连接起来，实现数据的持续传递，我们可以不去关
心内部数据如何流动
rs.pipe(ws)
```

这个案例可以改的再复杂一些，来加深印象，我们把转换流也加进去，实现它的内置接口 _transform 和 _flush：

```
const stream = require('stream')
```

```
class ReadStream extends stream.Readable {
  constructor() {
    super()
  }

  _read () {
    this.push('I ')
    this.push('Love ')
    this.push('Juejin!\n')
    this.push(null)
  }
}

class WriteStream extends stream.Writable {
  constructor() {
    super()
    this._storage = Buffer.from('')
  }

  _write (chunk, encode, cb) {
    console.log(chunk.toString())
    cb()
  }
}

class TransformStream extends stream.Transform {
  constructor() {
    super()
    this._storage = Buffer.from('')
  }

  _transform (chunk, encode, cb) {
```

```
    this.push(chunk)
    cb()
  }

  _flush (cb) {
    this.push('Oh Yeah!')
    cb()
  }
}

const rs = new ReadStream()
const ws = new WriteStream()
const ts = new TransformStream()

rs.pipe(ts).pipe(ws)
```

编程练习 - 实现一个 MP4 转 MP3 工具

最后，我们来基于对流的理解，在本地实现一个下载 MP4 和从 MP4 里面导出 MP3 的小工具，本地操作视频流，我们可以借助于 FFMPEG (http://ffmpeg.org/) , ffmpeg 是一个跨平台的流媒体库，可以记录和转换音视频，有非常强大的多媒体处理能力，大家可以前往看文档，结合自己的操作系统来安装，在 Mac 上安装特别简单，首先保证已经安装过 homebrew (https://brew.sh/)，可能安装需要梯子，安装后，直接执行：

```
brew install ffmpeg
```

然后代码实现上，我们通过 ffmpeg 的流来把一个远端的 MP4 文件中的音频流存储为 mp3，或者干脆下载这个 mp4，源文件我们就使用一个豆瓣预告片视频好了：

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

```

```
//const https = require('https')
const http = require('http')
const request = require('request')
const child_process = require('child_process')
const EventEmitter =
require('events').EventEmitter

const spawn = child_process.spawn
const mp3Args = ['-i', 'pipe:0', '-f', 'mp3', '-
ac', '2', '-ab', '128k', '-acodec', 'libmp3lame',
'pipe:1']
const mp4Args = ['-i', 'pipe:0', '-c', 'copy', '-
bsf:a', 'aac_adtstoasc', 'pipe:1']

class VideoTool extends EventEmitter {
  constructor (url, filename) {
    super()
    this.url = url
    this.filename = filename
  }

  mp3 () {
    // 创建 FFMPEG 进程
    this.ffmpeg = spawn('ffmpeg', mp3Args)

    // 拿到 Stream 流
    http.get(this.url, (res) => {
      res.pipe(this.ffmpeg.stdin)
    })

    // 把拿到的流 pipe 到文件中

this.ffmpeg.stdout.pipe(fs.createWriteStream(this
```

```
.filename))

    this.ffmpeg.on('exit', () => {
      console.log('Finished:', this.filename)
    })
  }

  mp4 () {
    let stream =
fs.createWriteStream(this.filename)
    request
      .get(this.url, {
        headers: {
          'Content-Type': 'video/mpeg4',
          'User-Agent': 'Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_13_4) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/70.0.3538.102
Safari/537.36'
        }
      })
      .pipe(stream)
      .on('open', () => {
        console.log('start download')
      })
      .on('close', () => {
        console.log('download finished')
      })
  }
}

const video =
'http://vt1.doubanio.com/201810291353/4d7bcf6af73
0df6d9b4da321aa6d7faa/view/movie/M/402380210.mp4'
```

```
const m1 = new VideoTool(video, 'audio.mp3')
const m2 = new VideoTool(video, __dirname +
'/video.mp4')

m1.mp3()
m2.mp4()
```