Stream

前端精进(后端方向) - Node.js 全解

版权声明

联系方式

如果你想要购买本课程
请微信联系 xiedaimala02 或 xiedaimala03

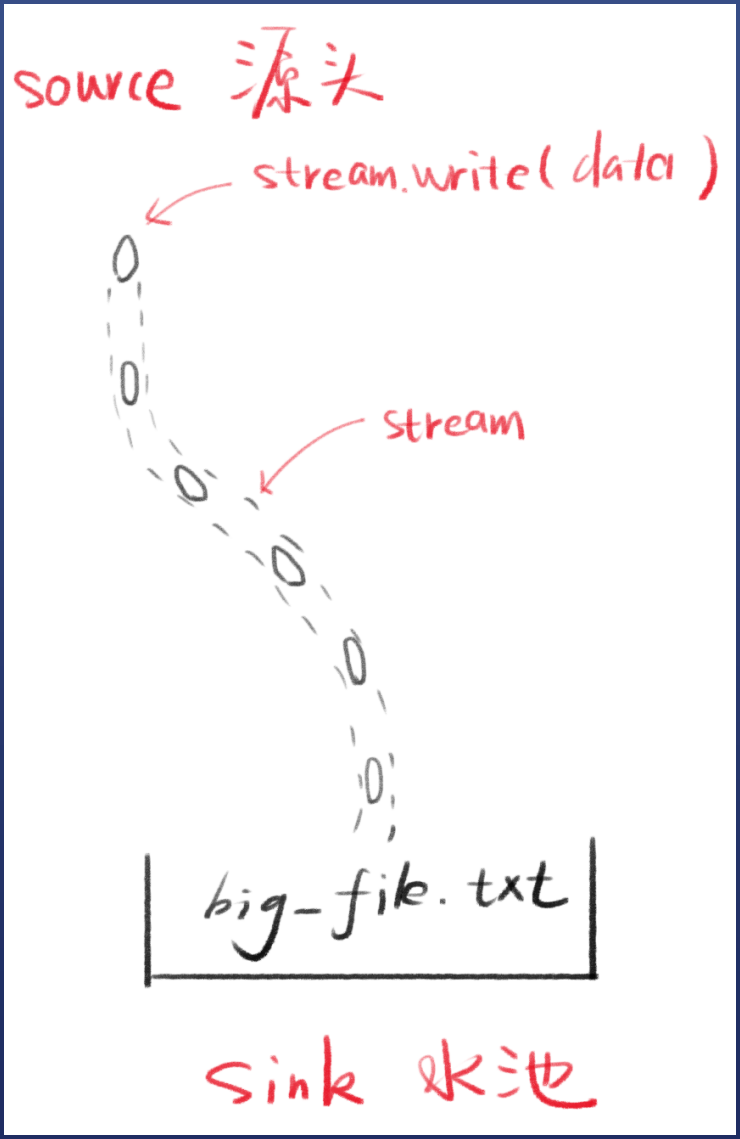
如果你发现有人盗用本课程
请微信联系 xiedaimala02 或 xiedaimala03

第一个 Stream 例子

```
const fs = require('fs')
const stream = fs.createWriteStream('./big_file.txt')
for(let i=0; i<10000; i++){
  stream.write(`这是第 ${i} 行内容，我们需要很多很多内容，要不停地写文件啊啊啊啊啊啊回车\n`)
}
stream.end() // 别忘了关掉 stream
console.log('done')
```

- 分析
  - ✓ 打开流，多次往里面塞内容，关闭流
  - ✓ 看起来就是可以多次写嘛，没什么大不了的
  - ✓ 最终我们得到一个 128 兆左右的文件

Stream - 流



释义

stream 是水流，但默认没有水
stream.write 可以让水流中有水（数据）
每次写的小数据叫做 chunk（块）
产生数据的一段叫做 source（源头）
得到数据的一段叫做 sink（水池）

第二个例子

```
// 请先引入 fs 和 http
const server = http.createServer()
server.on('request', (request, response)=>{
  fs.readFile('./big_file.txt', (error,
data)=>{
    if(err) throw err
    response.end(data)
    console.log('done')
  })
})
server.listen(8888)
```

- 分析
  - ✓ 用任务管理器看看 Node.js 内存占用，大概 130Mb

第三个例子

- 用 Stream 改写第二个例子

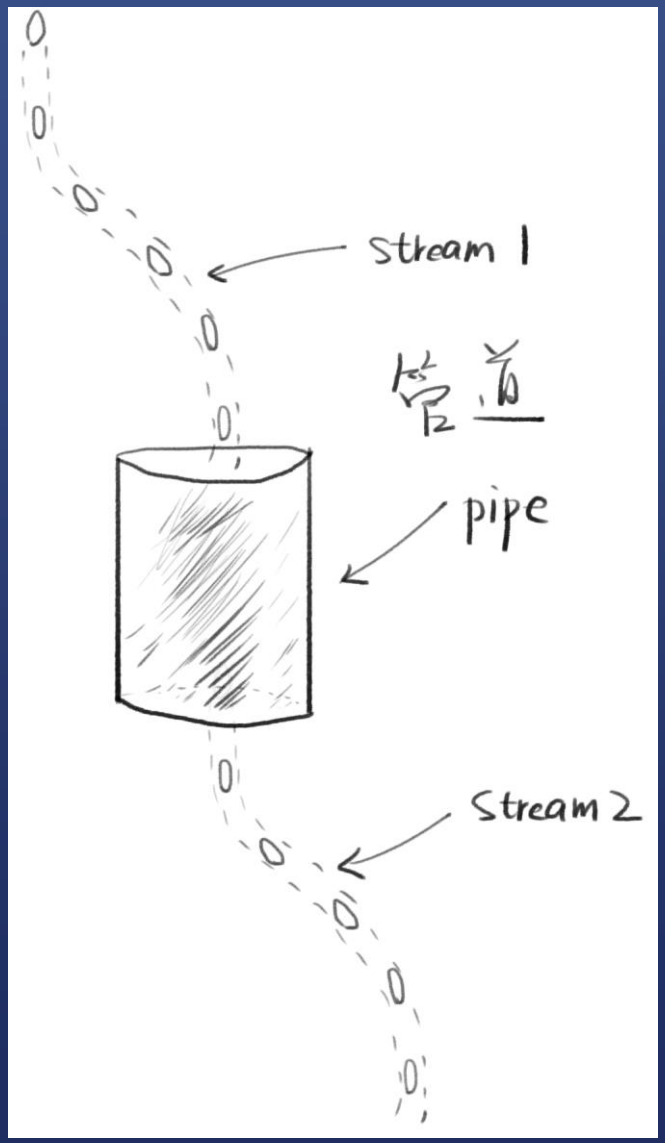
```
// 请先引入 fs 和 http
const server = http.createServer()
server.on('request', (request, response)=>{
  const stream =
    fs.createReadStream('./big_file.txt')
  stream.pipe(response)
})

server.listen(8888)
```

- 分析

✓ 查看 Node.js 内存占用，基本不会高于 30 Mb
✓ 文件 stream 和 response stream 通过管道相连

管道



释义

两个流可以用一个管道相连
stream1 的末尾连接上 stream2 的开端
只要 stream1 有数据，就会流到 stream2

常用代码

```
stream1.pipe(stream2)
```

链式操作

```
a.pipe(b).pipe(c)
// 等价于
a.pipe(b)
b.pipe(c)
```

管道续

- 管道可以通过事件实现

```
// stream1 一有数据就塞给 stream2
stream1.on('data', (chunk)=>{
  stream2.write(chunk)
})
// stream1 停了，就停掉 stream2
stream1.on('end', ()=>{
  stream2.end()
})
```

Stream 对象的原型链

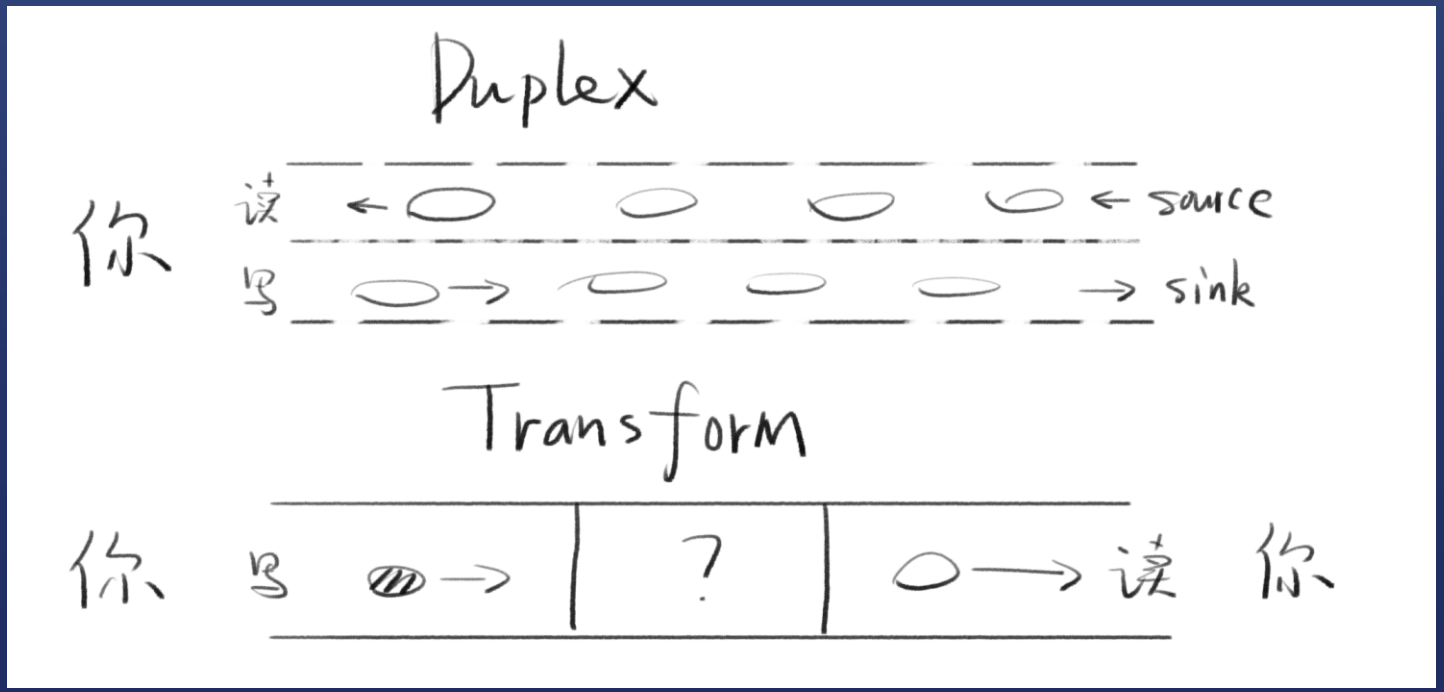
- s = fs.createReadStream(path)
  - ✓ 那么它的对象层级为
  - ✓ 自身属性（由 fs.ReadStream 构造）
  - ✓ 原型：stream.Readable.prototype
  - ✓ 二级原型：stream.Stream.prototype
  - ✓ 三级原型：events.EventEmitter.prototype
  - ✓ 四级原型：Object.prototype
- Stream 对象都继承了 EventEmitter

支持的事件和方法

	Readable Stream	Writable Stream
事件	data, end, error, close, readable	drain, finish, error, close, pipe, unpipe
方法	pipe() unpipe() wrap() destroy() read() unshift() resume() pause() isPaused() setEncoding()	write() destroy() end() cork() uncork() setDefaultEncoding()

Stream 分类

名称	特点
Readable	可读
Writable	可写
Duplex	可读可写（双向）
Transform	可读可写（变化）



Readable Stream

- 静止态 paused 和流动态 flowing

- ✓ 默认处于 paused 态
- ✓ 添加 data 事件监听，它就变为 flowing 态
- ✓ 删掉 data 事件监听，它就变为 paused 态
- ✓ pause() 可以将它变为 paused
- ✓ resume() 可以将它变为 flowing

Writable Stream

- drain 流干了事件
  - 表示可以加点水了
  - 我们调用 stream.write(chunk) 的时候，可能会得到 false
  - false 的意思是你写太快了，数据积压了
  - 这个时候我们就不能再 write 了，要监听 drain
  - 等 drain 事件触发了，我们才能继续 write
  - 听不懂？看文档中的代码就懂了
- finish 事件
  - 调用 stream.end() 之后，而且
  - 缓冲区数据都已经传给底层系统之后，
  - 触发 finish 事件

之前讲的内容

都是在使用 Stream

怎么创建自己的流，给别人用

创建一个 Writable Stream

```
const { Writable } = require("stream");

const outStream = new Writable({
  write(chunk, encoding, callback) {
    console.log(chunk.toString());
    callback();
  }
});

process.stdin.pipe(outStream);
// 保存文件为 writable.js 然后用 node 运行
// 不管你输入什么，都会得到相同的结果
```

创建一个 Readable Stream

```
const { Readable } = require("stream");

const inStream = new Readable();

inStream.push("ABCDEFGHIJKLM");
inStream.push("NOPQRSTUVWXYZ");

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

inStream.pipe(process.stdout);
// 保存文件为 readable.js 然后用 node 运行
// 我们先把所有数据都 push 进去了，然后 pipe
```

续

```
const { Readable } = require("stream");

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)
// 保存文件为 readable2.js 然后用 node 运行
// 这次的数据是按需供给的，对方调用 read 我们才会给一次数据
```

Duplex Stream

```
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);
// 源代码来源见参考链接
```

Transform Stream

```
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);
// 源代码来源见参考链接
```

内置的 Transform Stream

```
const fs = require("fs");
const zlib = require("zlib");
const file = process.argv[2];

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream(file + ".gz"));
```

续

```
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"));
```

续2

```
// 略
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"));
```

续3

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

// ..

fs.createReadStream(file)
   .pipe(zlib.createGzip())
   .pipe(crypto.createCipher("aes192", "123456"))
   .pipe(reportProgress)
   .pipe(fs.createWriteStream(file + ".zz"))
   .on("finish", () => console.log("Done"));

// 有 bug，无法解压
// 应该先加密，再压缩
```

Stream 用途非常广

在 Node.js 里随处可见

Node.js 中的 Steam

Readable Stream	Writable Stream
HTTP Response - 客户端	HTTP Request - 客户端
HTTP Request - 服务端	HTTP Response - 服务端
fs read stream	fs write stream
zlib stream	zlib stream
TCP sockets	TCP sockets
child process stdout & stderr	child process stdin
process.stdin	process.stdout, process.stderr
其他	其他

数据流中的积压问题

背压 Back Pressure

参考

- Node's Streams
  - 链接，非常推荐阅读（如果英文好的话）
- Node.js Stream 文档
  - 英文链接
  - 中文链接
  - 文档里面有你想要知道的所有细节
- 面试题
  - 面试高级前端工程师必问之流-stream