link: null
title: 珠峰架构师成长计划
description: 实现了stream.Readable接口的对象,将对象数据读取为流数据,当监听data事件后,开始发射数据
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=80 sentences=409, words=2110

1. 流的概念

- 流是一组有序的，有起点和终点的字节数据传输手段
- 它不关心文件的整体内容，只关注是否从文件中读到了数据，以及读到数据之后的处理
- 流是一个抽象接口，被 Node 中的很多对象所实现。比如HTTP 服务器request和response对象都是流。

2.可读流createReadStream

实现了 stream.Readable接口的对象,将对象数据读取为流数据,当监听data事件后,开始发射数据

```
fs.createReadStream = function(path, options) {
  return new ReadStream(path, options);
};
util.inherits(ReadStream, Readable);
```

```
var rs = fs.createReadStream(path,[options]);
```

> 如果指定utf8编码highWaterMark要大于3个字节

流切换到流动模式,数据会被尽可能快的读出

```
rs.on('data', function (data) {
    console.log(data);
});
```

该事件会在读完数据后被触发

```
rs.on('end', function () {
    console.log('读取完成');
});
```

```
rs.on('error', function (err) {
    console.log(err);
});
```

```
rs.on('open', function () {
    console.log(err);
});
```

```
rs.on('close', function () {
    console.log(err);
});
```

与指定{encoding:'utf8'}效果相同，设置编码

```
rs.setEncoding('utf8');
```

通过pause()方法和resume()方法

```
rs.on('data', function (data) {
    rs.pause();
    console.log(data);
});
setTimeout(function () {
    rs.resume();
},2000);
```

3.可写流createWriteStream

实现了stream.Writable接口的对象来将流数据写入到对象中

```
fs.createWriteStream = function(path, options) {
  return new WriteStream(path, options);
};

util.inherits(WriteStream, Writable);
```

```
var ws = fs.createWriteStream(path,[options]);
```

```
ws.write(chunk,[encoding],[callback]);
```

> 返回值为布尔值，系统缓存区满时为false,未满时为true

```
ws.end(chunk,[encoding],[callback]);
```

> 表明接下来没有数据要被写入 Writable 通过传入可选的 chunk 和 encoding 参数，可以在关闭流之前再写入一段数据 如果传入了可选的 callback 函数，它将作为 'finish' 事件的回调函数

- 当一个流不处在 drain 的状态， 对 write() 的调用会缓存数据块， 并且返回 false。 一旦所有当前所有缓存的数据块都排空了（被操作系统接受来进行输出）， 那么 'drain' 事件就会被触发
- 建议， 一旦 write() 返回 false， 在 'drain' 事件触发前， 不能写入任何数据块

```
let fs = require('fs');
let ws = fs.createWriteStream('./2.txt',{
  flags:'w',
  encoding:'utf8',
  highWaterMark:3
});
let i = 10;
function write(){
 let  flag = true;
 while(i&&flag){
      flag = ws.write("1");
      i--;
     console.log(flag);
 }
}
write();
ws.on('drain',()=>{
  console.log("drain");
  write();
});
```

在调用了 stream.end() 方法，且缓冲区数据都已经传给底层系统之后， 'finish' 事件将被触发。

```
var writer = fs.createWriteStream('./2.txt');
for (let i = 0; i < 100; i++) {
  writer.write(`hello, ${i}!\n`);
}
writer.end('&#x7ED3;&#x675F;\n');
writer.on('finish', () => {
  console.error('&#x6240;&#x6709;&#x7684;&#x5199;&#x5165;&#x5DF2;&#x7ECF;&#x5B8C;&#x6210;!');
});
```

4.pipe方法

```
var fs = require('fs');
var ws = fs.createWriteStream('./2.txt');
var rs = fs.createReadStream('./1.txt');
rs.on('data', function (data) {
    var flag = ws.write(data);
    if(!flag)
    rs.pause();
});
ws.on('drain', function () {
    rs.resume();
});
rs.on('end', function () {
    ws.end();
});
```

```
readStream.pipe(writeStream);
var from = fs.createReadStream('./1.txt');
var to = fs.createWriteStream('./2.txt');
from.pipe(to);
```

> 将数据的滞留量限制到一个可接受的水平，以使得不同速度的来源和目标不会淹没可用内存。

- readable.unpipe()方法将之前通过stream.pipe()方法绑定的流分离
- 如果 destination 没有传入, 则所有绑定的流都会被分离.

```
let fs = require('fs');
var from = fs.createReadStream('./1.txt');
var to = fs.createWriteStream('./2.txt');
from.pipe(to);
setTimeout(() => {
console.log('关闭向2.txt的写入');
from.unpipe(writable);
console.log('手工关闭文件流');
to.end();
}, 1000);
```

调用 writable.cork() 方法将强制所有写入数据都存放到内存中的缓冲区里。 直到调用 stream.uncork() 或 stream.end() 方法时，缓冲区里的数据才会被输出。

writable.uncork()将输出在 `stream.cork()` 方法被调用之后缓冲在内存中的所有数据。

```
stream.cork();
stream.write('1');
stream.write('2');
process.nextTick(() => stream.uncork());
```

5. 简单实现

```
let fs = require('fs');
let ReadStream = require('./ReadStream');
let rs = ReadStream('./1.txt', {
    flags: 'r',
    encoding: 'utf8',
    start: 3,
    end: 7,
    highWaterMark: 3
});
rs.on('open', function () {
    console.log("open");
});
rs.on('data', function (data) {
    console.log(data);
});
rs.on('end', function () {
    console.log("end");
});
rs.on('close', function () {
    console.log("close");
});
/**
 open
 456
 789
 end
 close
 **/
```

```
let fs = require('fs');
let EventEmitter = require('events');

class WriteStream extends EventEmitter {
    constructor(path, options) {
        super(path, options);
        this.path = path;
        this.fd = options.fd;
        this.flags = options.flags || 'r';
        this.encoding = options.encoding;
        this.start = options.start || 0;
        this.pos = this.start;
        this.end = options.end;
        this.flowing = false;
        this.autoClose = true;
        this.highWaterMark = options.highWaterMark || 64 * 1024;
        this.buffer = Buffer.alloc(this.highWaterMark);
        this.length = 0;
        this.on('newListener', (type, listener) => {
            if (type == 'data') {
                this.flowing = true;
                this.read();
            }
        });
        this.on('end', () => {
            if (this.autoClose) {
                this.destroy();
            }
        });
        this.open();
    }

    read() {
        if (typeof this.fd != 'number') {
            return this.once('open', () => this.read());
        }
        let n = this.end ? Math.min(this.end - this.pos, this.highWaterMark) : this.highWaterMark;
        fs.read(this.fd,this.buffer,0,n,this.pos,(err,bytesRead)=>{
            if(err){
             return;
            }
            if(bytesRead){
                let data = this.buffer.slice(0,bytesRead);
                data = this.encoding?data.toString(this.encoding):data;
                this.emit('data',data);
                this.pos += bytesRead;
                if(this.end && this.pos > this.end){
                  return this.emit('end');
                }
                if(this.flowing)
                    this.read();
            }else{
                this.emit('end');
            }
        })
    }

    open() {
        fs.open(this.path, this.flags, this.mode, (err, fd) => {
            if (err) return this.emit('error', err);
            this.fd = fd;
            this.emit('open', fd);
        })
    }

    end() {
        if (this.autoClose) {
            this.destroy();
        }
    }

    destroy() {
        fs.close(this.fd, () => {
            this.emit('close');
        })
    }

}

module.exports = WriteStream;
```

```
let fs = require('fs');
let FileWriteStream = require('./FileWriteStream');
let ws = FileWriteStream('./2.txt',{
    flags:'w',
    encoding:'utf8',
    highWaterMark:3
});
let i = 10;
function write(){
    let  flag = true;
    while(i&&flag){
        flag = ws.write("1",'utf8',(function(i){
            return function(){
                console.log(i);
            }
        })(i));
        i--;
        console.log(flag);
    }
}
write();
ws.on('drain',()=>{
    console.log("drain");
    write();
});
/**
 10
 9
 8
 drain
 7
 6
 5
 drain
 4
 3
 2
 drain
 1
 **/
```

```
let fs = require('fs');
let EventEmitter = require('events');
class WriteStream extends  EventEmitter{
    constructor(path, options) {
        super(path, options);
        this.path = path;
        this.fd = options.fd;
        this.flags = options.flags || 'w';
        this.mode = options.mode || 0o666;
        this.encoding = options.encoding;
        this.start = options.start || 0;
        this.pos = this.start;
        this.writing = false;
        this.autoClose = true;
        this.highWaterMark = options.highWaterMark || 16 * 1024;
        this.buffers = [];
        this.length = 0;
        this.open();
    }

    open() {
        fs.open(this.path, this.flags, this.mode, (err, fd) => {
            if (err) return this.emit('error', err);
            this.fd = fd;
            this.emit('open', fd);
        })
    }

    write(chunk, encoding, cb) {
        if (typeof encoding == 'function') {
            cb = encoding;
            encoding = null;
        }

        chunk = Buffer.isBuffer(chunk) ? chunk : Buffer.from(chunk, this.encoding || 'utf8');
        let len = chunk.length;
        this.length += len;
        let ret = this.length < this.highWaterMark;
        if (this.writing) {
            this.buffers.push({
                chunk,
                encoding,
                cb,
            });
        } else {
            this.writing = true;
            this._write(chunk, encoding,this.clearBuffer.bind(this));
        }
        return ret;
    }

    _write(chunk, encoding, cb) {
        if (typeof this.fd != 'number') {
            return this.once('open', () => this._write(chunk, encoding, cb));
        }
        fs.write(this.fd, chunk, 0, chunk.length, this.pos, (err, written) => {
            if (err) {
                if (this.autoClose) {
                    this.destroy();
                }
                return this.emit('error', err);
            }
            this.length -= written;
            this.pos += written;
            cb && cb();
        });
    }

    clearBuffer() {
        let data = this.buffers.shift();
        if (data) {
            this._write(data.chunk, data.encoding, this.clearBuffer.bind(this))
        } else {
            this.writing = false;
            this.emit('drain');
        }
    }

    end() {
        if (this.autoClose) {
            this.emit('end');
            this.destroy();
        }
    }

    destroy() {
        fs.close(this.fd, () => {
            this.emit('close');
        })
    }

}

module.exports = WriteStream;
```

```
let fs = require('fs');
let ReadStream = require('./ReadStream');
let rs = ReadStream('./1.txt', {
    flags: 'r',
    encoding: 'utf8',
    highWaterMark: 3
});
let FileWriteStream = require('./WriteStream');
let ws = FileWriteStream('./2.txt',{
    flags:'w',
    encoding:'utf8',
    highWaterMark:3
});
rs.pipe(ws);
```

```
ReadStream.prototype.pipe = function (dest) {
    this.on('data', (data)=>{
        let flag = dest.write(data);
        if(!flag){
            this.pause();
        }
    });
    dest.on('drain', ()=>{
        this.resume();
    });
    this.on('end', ()=>{
        dest.end();
    });
}
ReadStream.prototype.pause = function(){
    this.flowing = false;

}
ReadStream.prototype.resume = function(){
    this.flowing = true;
    this.read();
}
```

5.4 暂停模式

```
let fs =require('fs');
let ReadStream2 = require('./ReadStream2');
let rs = new ReadStream2('./1.txt',{
    start:3,
    end:8,
    encoding:'utf8',
    highWaterMark:3
});
rs.on('readable',function () {
    console.log('readable');
    console.log('rs.buffer.length',rs.length);
    let d = rs.read(1);
    console.log(d);
    console.log('rs.buffer.length',rs.length);

    setTimeout(()=>{
        console.log('rs.buffer.length',rs.length);
    },500)
});
```

```
let fs = require('fs');
let EventEmitter = require('events');
class ReadStream extends EventEmitter {
    constructor(path, options) {
        super(path, options);
        this.path = path;
        this.highWaterMark = options.highWaterMark || 64 * 1024;
        this.buffer = Buffer.alloc(this.highWaterMark);
        this.flags = options.flags || 'r';
        this.encoding = options.encoding;
        this.mode = options.mode || 0o666;
        this.start = options.start || 0;
        this.end = options.end;
        this.pos = this.start;
        this.autoClose = options.autoClose || true;
        this.bytesRead = 0;
        this.closed = false;
        this.flowing;
        this.needReadable = false;
        this.length = 0;
        this.buffers = [];
        this.on('end', function () {
            if (this.autoClose) {
                this.destroy();
            }
        });
        this.on('newListener', (type) => {
            if (type == 'data') {
                this.flowing = true;
                this.read();
            }
            if (type == 'readable') {
                this.read(0);
            }
        });
        this.open();
    }

    open() {
        fs.open(this.path, this.flags, this.mode, (err, fd) => {
            if (err) {
                if (this.autoClose) {
                    this.destroy();
```

```
                    return this.emit('error', err);
                }
            }
            this.fd = fd;
            this.emit('open');
        });
    }

    read(n) {
        if (typeof this.fd != 'number') {
            return this.once('open', () => this.read());
        }
        n = parseInt(n,10);
        if(n != n){
            n = this.length;
        }
        if(this.length ==0)
            this.needReadable = true;
        let ret;
        if (0this.length) {
            ret = Buffer.alloc(n);
            let b ;
            let index = 0;
            while(null != (b = this.buffers.shift())){
                for(let i=0;iif(index == ret.length){
                        this.length -= n;
                        b = b.slice(i+1);
                        this.buffers.unshift(b);
                        break;
                    }
                }
            }
            if (this.encoding) ret = ret.toString(this.encoding);
        }

        let _read = () => {
            let m = this.end ? Math.min(this.end - this.pos + 1, this.highWaterMark) : this.highWaterMark;
            fs.read(this.fd, this.buffer, 0, m, this.pos, (err, bytesRead) => {
                if (err) {
                    return
                }
                let data;
                if (bytesRead > 0) {
                    data = this.buffer.slice(0, bytesRead);
                    this.pos += bytesRead;
                    this.length += bytesRead;
                    if (this.end && this.pos > this.end) {
                        if(this.needReadable){
                            this.emit('readable');
                        }

                        this.emit('end');
                    } else {
                        this.buffers.push(data);
                        if(this.needReadable){
                            this.emit('readable');
                            this.needReadable = false;
                        }

                    }
                } else {
                    if(this.needReadable) {
                        this.emit('readable');
                    }
                    return this.emit('end');
                }
            })
        }
        if (this.length == 0 || (this.length < this.highWaterMark)) {
            _read(0);
        }
        return ret;
    }

    destroy() {
        fs.close(this.fd, (err) => {
            this.emit('close');
        });
    }

    pause() {
        this.flowing = false;
    }

    resume() {
        this.flowing = true;
        this.read();
    }

    pipe(dest) {
        this.on('data', (data) => {
            let flag = dest.write(data);
            if (!flag) this.pause();
        });
        dest.on('drain', () => {
            this.resume();
        });
        this.on('end', () => {
            dest.end();
        });
    }

}
```

```
module.exports = ReadStream;
```