link: null
title: 珠峰架构师成长计划
description: 在任意时刻，任意可读流应确切处于下面三种状态之一：
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=31 sentences=132, words=675

1. Node.js 中有四种基本的流类型：

- Readable - 可读的流 (例如 fs.createReadStream()).
- Writable - 可写的流 (例如 fs.createWriteStream()).
- Duplex - 可读写的流 (例如 net.Socket).
- Transform - 在读写过程中可以修改和变换数据的 Duplex 流 (例如 zlib.createDeflate()).

2. 流中的数据有两种模式,二进制模式和对象模式.

- 二进制模式, 每个分块都是buffer或者string对象.
- 对象模式, 流内部处理的是一系列普通对象.

> 所有使用 Node.js API 创建的流对象都只能操作 strings 和 Buffer对象。但是，通过一些第三方流的实现，你依然能够处理其它类型的 JavaScript 值 (除了 null，它在流处理中有特殊意义)。 这些流被认为是工作在 "对象模式"（object mode）。 在创建流的实例时，可以通过 objectMode 选项使流的实例切换到对象模式。试图将已经存在的流切换到对象模式是不安全的。

3. 可读流的两种模式

- 可读流事实上工作在下面两种模式之一： `flowing` 和 `paused`
- 在 flowing 模式下， 可读流自动从系统底层读取数据，并通过 EventEmitter 接口的事件尽快将数据提供给应用。
- 在 paused 模式下，必须显式调用 stream.read() 方法来从流中读取数据片段。
- 所有初始工作模式为 paused 的 Readable 流，可以通过下面三种途径切换到 flowing 模式：
  - 监听 'data' 事件
  - 调用 stream.resume() 方法
  - 调用 stream.pipe() 方法将数据发送到 Writable
- 可读流可以通过下面途径切换到 paused 模式：
  - 如果不存在管道目标（pipe destination），可以通过调用 stream.pause() 方法实现。
  - 如果存在管道目标，可以通过取消 'data' 事件监听，并调用 stream.unpipe() 方法移除所有管道目标来实现。

> 如果 Readable 切换到 flowing 模式，且没有消费者处理流中的数据，这些数据将会丢失。 比如， 调用了 readable.resume() 方法却没有监听 'data' 事件，或是取消了 'data' 事件监听，就有可能出现这种情况。

4.缓存区

- Writable 和 Readable 流都会将数据存储到内部的缓冲器（buffer）中。这些缓冲器可以 通过相应的 writable._writableState.getBuffer() 或 readable._readableState.buffer 来获取。
- 缓冲器的大小取决于传递给流构造函数的 highWaterMark 选项。 对于普通的流， highWaterMark 选项指定了总共的字节数。对于工作在对象模式的流， highWaterMark 指定了对象的总数。
- 当可读流的实现调用 `stream.push(chunk)` 方法时，数据被放到缓冲器中。如果流的消费者没有调用 `stream.read()` 方法， 这些数据会始终存在于内部队列中，直到被消费。
- 当内部可读缓冲器的大小达到 highWaterMark 指定的阈值时，流会暂停从底层资源读取数据，直到当前 缓冲器的数据被消费 (也就是说， 流会在内部停止调用 readable._read() 来填充可读缓冲器)。
- 可写流通过反复调用 writable.write(chunk) 方法将数据放到缓冲器。 当内部可写缓冲器的总大小小于 highWaterMark 指定的阈值时， 调用 writable.write() 将返回true。 一旦内部缓冲器的大小达到或超过 highWaterMark ，调用 writable.write() 将返回 false 。
- stream API 的关键目标， 尤其对于 stream.pipe() 方法， 就是限制缓冲器数据大小，以达到可接受的程度。这样，对于读写速度不匹配的源头和目标，就不会超出可用的内存大小。
- Duplex 和 Transform 都是可读写的。 在内部，它们都维护了 两个 相互独立的缓冲器用于读和写。 在维持了合理高效的数据流的同时，也使得对于读和写可以独立进行而互不影响。

5. 可读流的三种状态

在任意时刻，任意可读流应确切处于下面三种状态之一：

- readable._readableState.flowing = null
- readable._readableState.flowing = false
- readable._readableState.flowing = true
- 若 readable._readableState.flowing 为 null，由于不存在数据消费者，可读流将不会产生数据。 在这个状态下，监听 'data' 事件，调用 readable.pipe() 方法，或者调用 readable.resume() 方法，readable._readableState.flowing 的值将会变为 true 。这时，随着数据生成，可读流开始频繁触发事件。
- 调用 readable.pause() 方法， readable.unpipe() 方法， 或者接收 "背压"（back pressure）， 将导致 readable._readableState.flowing 值变为 false。 这将暂停事件流，但 不会 暂停数据生成。 在这种情况下，为 'data' 事件设置监听函数不会导致 readable._readableState.flowing 变为 true。
- 当 readable._readableState.flowing 值为 false 时， 数据可能堆积到流的内部缓存中。

6.readable

'readable' 事件将在流中有数据可供读取时触发。在某些情况下，为 'readable' 事件添加回调将会导致一些数据被读取到内部缓存中。

```
const readable = getReadableStreamSomehow();
readable.on('readable', () => {
  // &#x6709;&#x4E00;&#x4E9B;&#x6570;&#x636E;&#x53EF;&#x8BFB;&#x4E86;
});
```

- 当到达流数据尾部时， 'readable' 事件也会触发。触发顺序在 'end' 事件之前。
- 事实上， 'readable' 事件表明流有了新的动态：要么是有了新的数据，要么是到了流的尾部。 对于前者， stream.read() 将返回可用的数据。而对于后者， stream.read() 将返回 null。

```
let fs =require('fs');
let rs = fs.createReadStream('./1.txt',{
  start:3,
  end:8,
  encoding:'utf8',
  highWaterMark:3
});
rs.on('readable',function () {
  console.log('readable');
  console.log('rs._readableState.buffer.length',rs._readableState.length);
  let d = rs.read(1);
  console.log('rs._readableState.buffer.length',rs._readableState.length);
  console.log(d);
  setTimeout(()=>{
      console.log('rs._readableState.buffer.length',rs._readableState.length);
  },500)
});
```

7.流的经典应用

7.1 行读取器

7.1.1 换行和回车

- 以前的打印要每秒可以打印10个字符，换行城要0.2秒，正要可以打印2个字符。
- 研制人员就是在每行后面加两个表示结束的字符。一个叫做"回车"，告诉打字机把打印头定位在左边界；另一个叫做"换行"，告诉打字机把纸向下移一行。
- Unix系统里，每行结尾只有换行"(line feed)"，即"\n",
- Windows系统里面，每行结尾是"
- Mac系统里，每行结尾是"回车"(carriage return)，即"\r"
- 在ASCII码里
    - 换行 \n 10 0A
    - 回车 \r 13 0D

ASCII (http://ascii.911cha.com/)

7.1.2 代码

```
let fs = require('fs');
let EventEmitter = require('events');
let util = require('util');
util.inherits(LineReader, EventEmitter)
fs.readFile('./1.txt',function (err,data) {
    console.log(data);
})
function LineReader(path) {
    EventEmitter.call(this);
    this._rs = fs.createReadStream(path);
    this.RETURN = 0x0D;// \r 13
    this.NEW_LINE = 0x0A;// \n 10
    this.on('newListener', function (type, listener) {
        if (type == 'newLine') {
            let buffer = [];
            this._rs.on('readable', () => {
                let bytes;
                while (null != (bytes = this._rs.read(1))) {
                    let ch = bytes[0];
                    switch (ch) {
                        case this.RETURN:
                            this.emit('newLine', Buffer.from(buffer));
                            buffer.length = 0;
                            let nByte = this._rs.read(1);
                            if (nByte && nByte[0] != this.NEW_LINE) {
                                buffer.push(nByte[0]);
                            }
                            break;
                        case this.NEW_LINE:
                            this.emit('newLine', Buffer.from(buffer));
                            buffer.length = 0;
                            break;
                        default:
                            buffer.push(bytes[0]);
                            break;
                    }
                }
            });
            this._rs.on('end', () => {
                if (buffer.length > 0) {
                    this.emit('newLine', Buffer.from(buffer));
                    buffer.length = 0;
                    this.emit('end');
                }
            })
        }
    });
}

var lineReader = new LineReader('./1.txt');
lineReader.on('newLine', function (data) {
    console.log(data.toString());
}).on('end', function () {
    console.log("end");
})
```