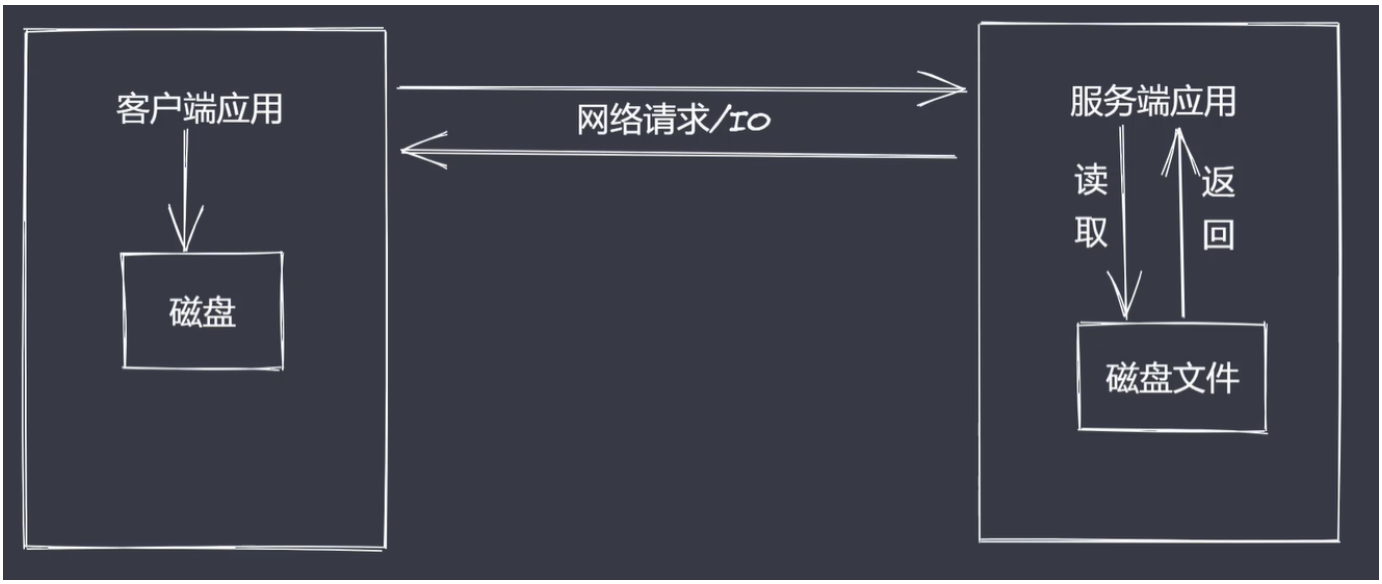
第八节 Nodejs 核心模块stream

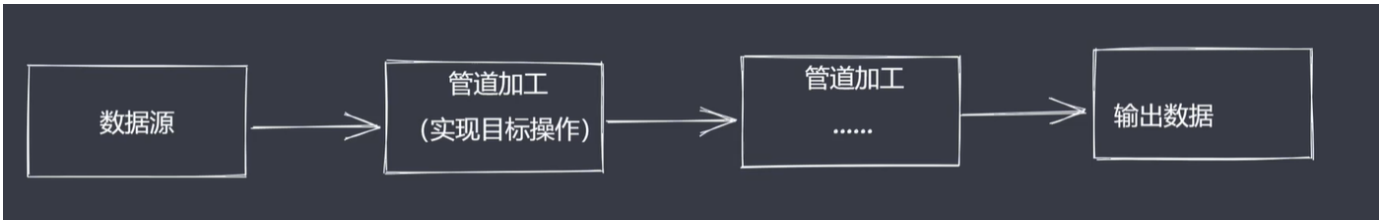
1.基本概念

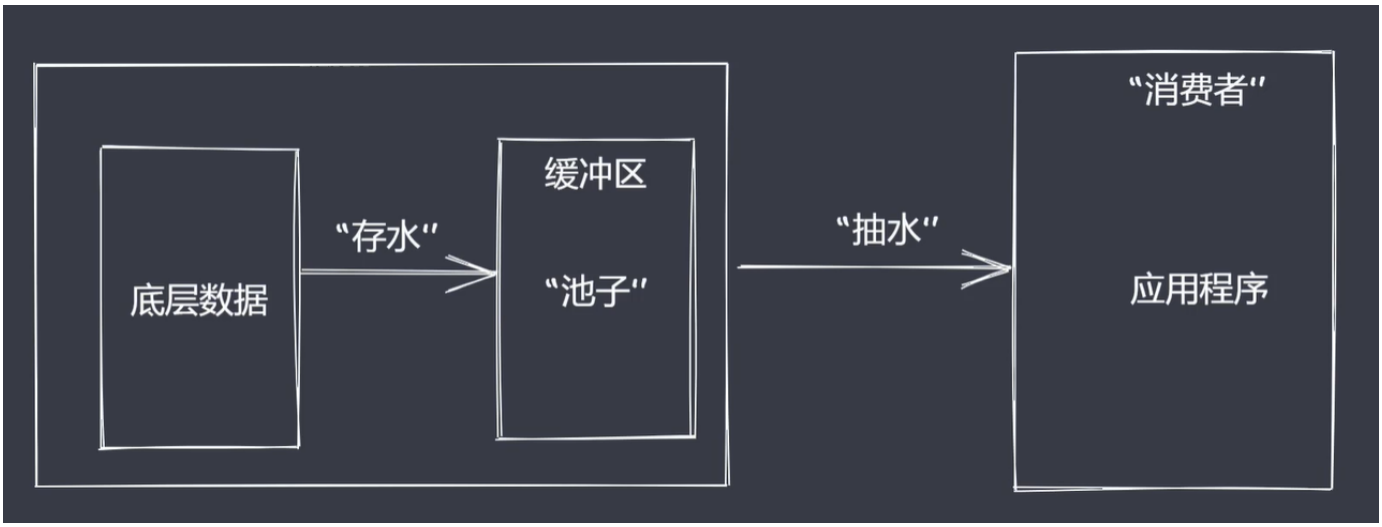
- Nodejs诞生之初就是为了提高IO性能
- 文件操作系统和网络模块实现了流接口
- Nodejs中的流就是处理流式数据的抽象接口

2.应用程序为什么使用流来处理数据?

- 同步读取资源文件， 用户需要等待数据读取完成
- 资源文件最终一次性加载至内存， 开销较大
- 时间效率： 流的分段处理可以同时操作多个数据chunk
- 空间效率： 同一个时间流无须占据大内存空间
- 使用方便： 流配合管理，扩展程序变简单







3. stream模块， 实现了流操作

3.1 流的分类

- Readable: 可读流，能够实现数据的读取
- Writeable： 可写流， 能够实现数据的写操作
- Duplex: 双工流，既可以读，又可以写
- Transform: 转换流，可读可写， 还能实现数据转换

3.2 Nodejs流特点

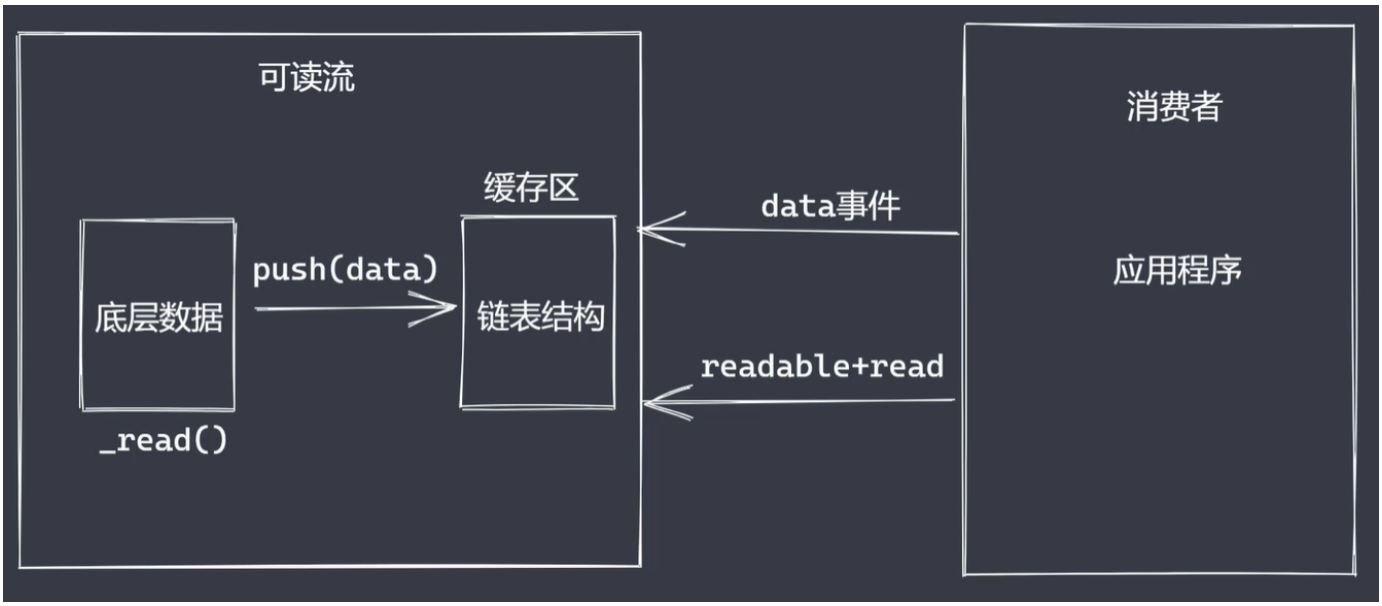
- Stream模块实现了四个具体的抽象
- 所有流都继承了EventEmitter

3.3 stream之可读流

- 生产供程序消费数据的流

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

let rs = fs.createReadStream(path.resolve(__dirname,'test.txt'))
let ws = fs.createWriteStream(path.resolve(__dirname,'test_temp.tx
t'))

rs.pipe(ws)
```

- 自定义可读流
  - 继承stream里的Readable
  - 重写_read方法调用push产出数据
- 消费数据
  - readable事件： 当流中存在可读数据时触发
  - data事件： 当流中数据块传给消费者后触发
- 总结
  - 明确数据生产与消费流程
  - 利用API实现自定义的可读流
  - 明确数据消费的事件使用




```
const { Readable } = require('stream')

// 模拟底层数据
let source = ['lg', 'zce', 'syy']

class MyReadable extends Readable {
  constructor(source) {
    super()
    this.source = source
  }
  _read() {
    let data = this.source.shift() || null
    this.push(data)
  }
}

// 实例化
let myReadable = new MyReadable(source)
myReadable.on('readable', () => {
  let data = null
  while((data = myReadable.read(2)) != null) {
    console.log(data.toString())
  }
})
```

```
25
26 myReadable.on('data', (chunk) => {
27   console.log(chunk.toString())
28 })
```

3.4 stream之可写流

用于消费数据的流

```
1 const fs = require('fs')
2 const path = require('path')
3
4 let rs = fs.createReadStream(path.resolve(__dirname,'test.txt'))
5 let ws = fs.createWriteStream(path.resolve(__dirname,'test_temp.tx
  t'))
6
7 rs.pipe(ws)
```

- 自定义可以写流
  - 继承stream模块的Writeable
  - 重写_write方法，调用write执行写入
- 可写流事件
  - pipe事件：可读流调用pipe方法时触发
  - unpipe事件， 可读流调用unpipe方法时触发

```
1 const {Writable} = require('stream')
2
3 class MyWriteable extends Writable {
4   constructor() {
5     super()
6   }
7   _write(chunk,en, done) {
8     process.stdout.write(chunk,toString() + '<---')
9     process.nextTick(done)
10  }
11 }
12
```

```
let myWriteable = new MyWriteable()

myWriteable.write('test', 'utf-8', () => {
  console.log('end')
})
```

3.5 stream之双工流和转换流

- Duplex是双工流， 既能生成又能消费
  - 自定义双工流
    - 继承Duplex
    - 重写_read方法， 调用push生产数据
    - 重写_write方法， 使用write消费数据

```
let {Duplex} = require('stream')

class MyDuplex extends Duplex{
  constructor(source) {
    super()
    this.source = source
  }
  _read() {
    let data = this.source.shift() || null
    this.push(data)
  }
  _write(chunk, en, next) {
    process.stdout.write(chunk)
    process.nextTick(next)
  }
}

let source = ['a', 'b', 'c']
let myDuplex = new MyDuplex(source)

/* myDuplex.on('data', (chunk) => {
  console.log(chunk.toString())
}) */
myDuplex.write('test', () => {
  console.log(1111)
```

```
26 })
```

- Transform也是一个双工流
  - 自定义转换流
    - 继承Transform
    - 重写_transform方法， 调用push和callback
    - 重写_flush方法， 处理剩余数据

```
1  let {Transform} = require('stream')
2
3
4  class MyTransform extends Transform{
5    constructor() {
6      super()
7    }
8    _transform(chunk, en, cb) {
9      this.push(chunk.toString().toUpperCase())
10     cb(null)
11   }
12 }
13
14 let t = new MyTransform()
15
16 t.write('a')
17
18 t.on('data', (chunk) => {
19   console.log(chunk.toString())
20 })
```

3.6 文件可读流

```
1  const fs = require('fs')
2
3  let rs = fs.createReadStream('test.txt', {
4    flags: 'r',
5    encoding: null,
6    fd: null,
```

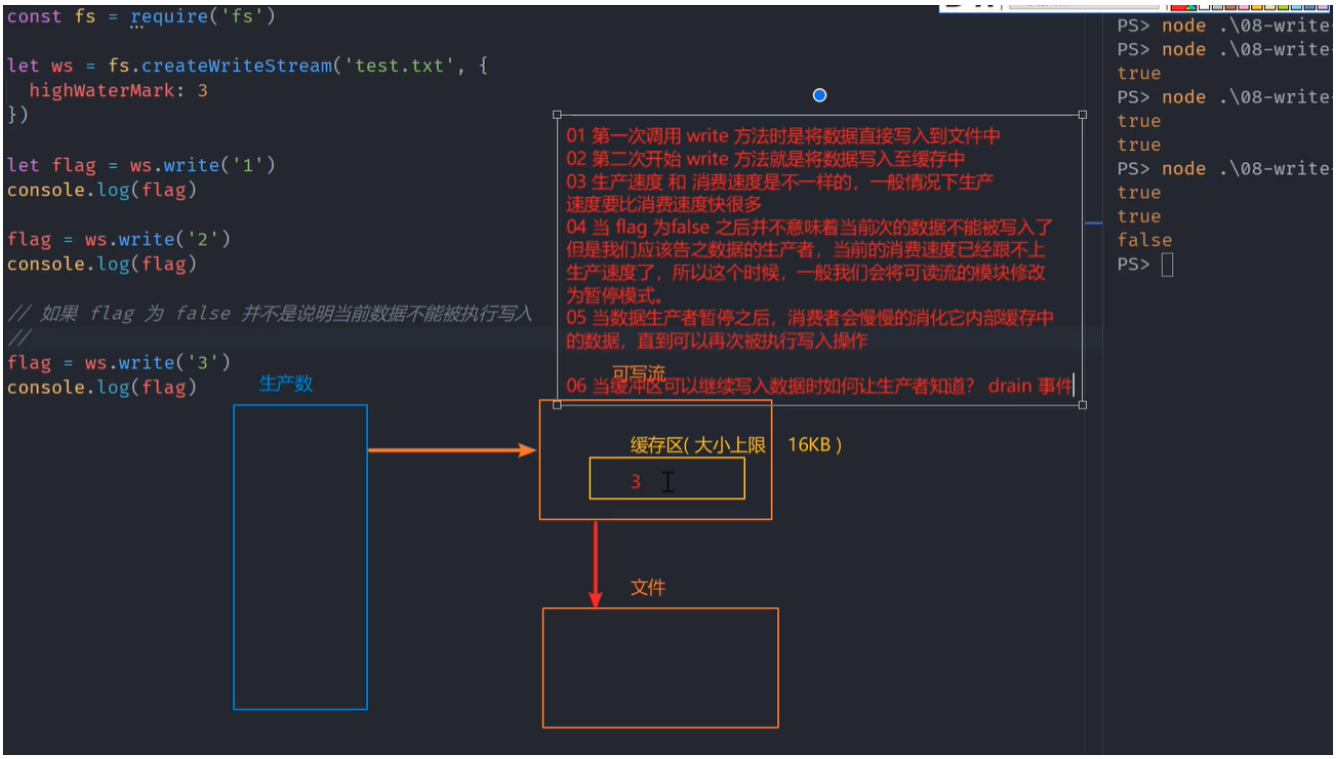
```
  mode: 438,
  autoClose: true,
  start: 0,
  // end: 3,
  highWaterMark: 4
})

/* rs.on('data', (chunk) => {
  console.log(chunk.toString())
  rs.pause()
  setTimeout(() => {
    rs.resume()
  }, 1000)
}) */

/* rs.on('readable', () => {
  let data = rs.read()
  console.log(data)
  let data
  while((data = rs.read(1)) !== null) {
    console.log(data.toString())
    console.log('----------', rs._readableState.length)
  }
}) */

rs.on('open', (fd) => {
  console.log(fd, '文件打开了')
})

rs.on('close', () => {
  console.log('文件关闭了')
})
let bufferArr = []
rs.on('data', (chunk) => {
  bufferArr.push(chunk)
})

rs.on('end', () => {
  console.log(Buffer.concat(bufferArr).toString())
  console.log('当数据被清空之后')
```

```
})

rs.on('error', (err) => {
  console.log('出错了')
})
```

3.7 文件可写流

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

const ws = fs.createWriteStream('test.txt', {
  flags: 'w',
  mode: 438,
  fd: null,
  encoding: "utf-8",
  start: 0,
  highWaterMark: 3
})

let buf = Buffer.from('abc')

// 字符串 或者  buffer ===》 fs rs
/* ws.write(buf, () => {
  console.log('ok2')
}) */

/* ws.write('tes', () => {
  console.log('ok1')
}) */

/* ws.on('open', (fd) => {
  console.log('open', fd)
}) */

ws.write("2")

// close 是在数据写入操作全部完成之后再执行
/* ws.on('close', () => {
```

```
  console.log('文件关闭了')
}) */

// end 执行之后就意味着数据写入操作完成
ws.end('test')


// error
ws.on('error', (err) => {
  console.log('出错了')
})

```

- drain事件
- 

- 写入速度

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

let ws = fs.createWriteStream(path.resolve(__dirname, 'test.txt')
, {
  highWaterMark: 3
```

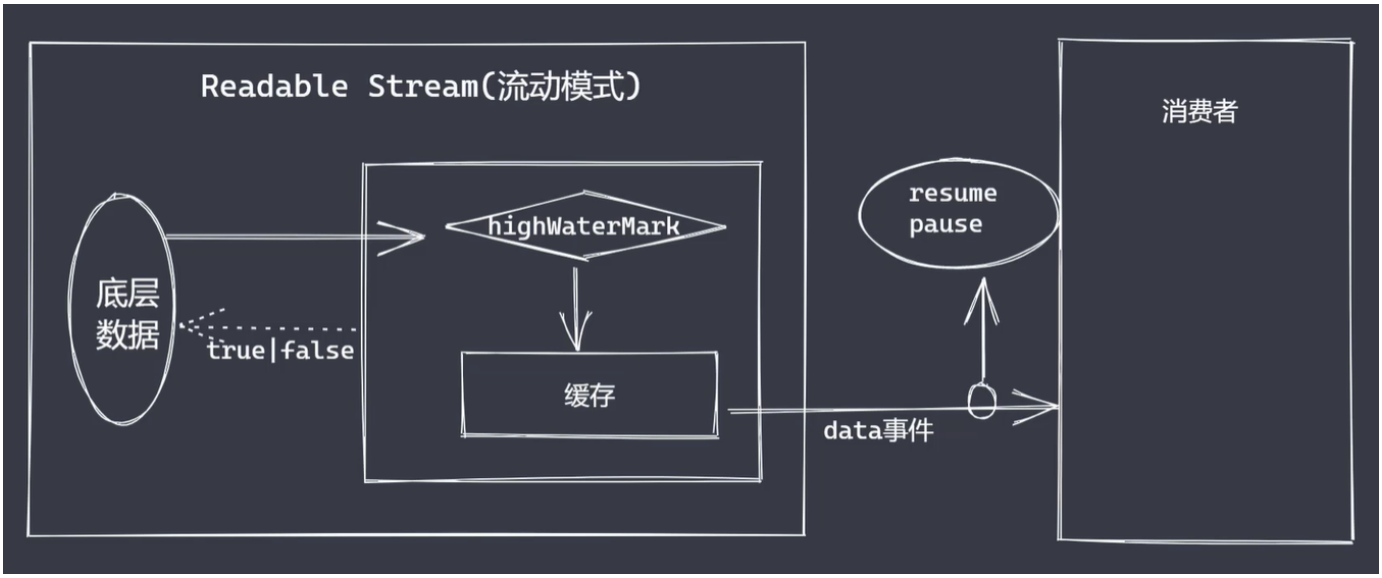
```
})

const source = '测试数据'.split('')

let num = 0
let flag = true

function executeWrite() {
  flag = true
  while(num !=  source.length && flag) {
    flag = ws.write(source[num])
    num++
  }
}

executeWrite()

ws.on('drain', () => {
  executeWrite()
})
```

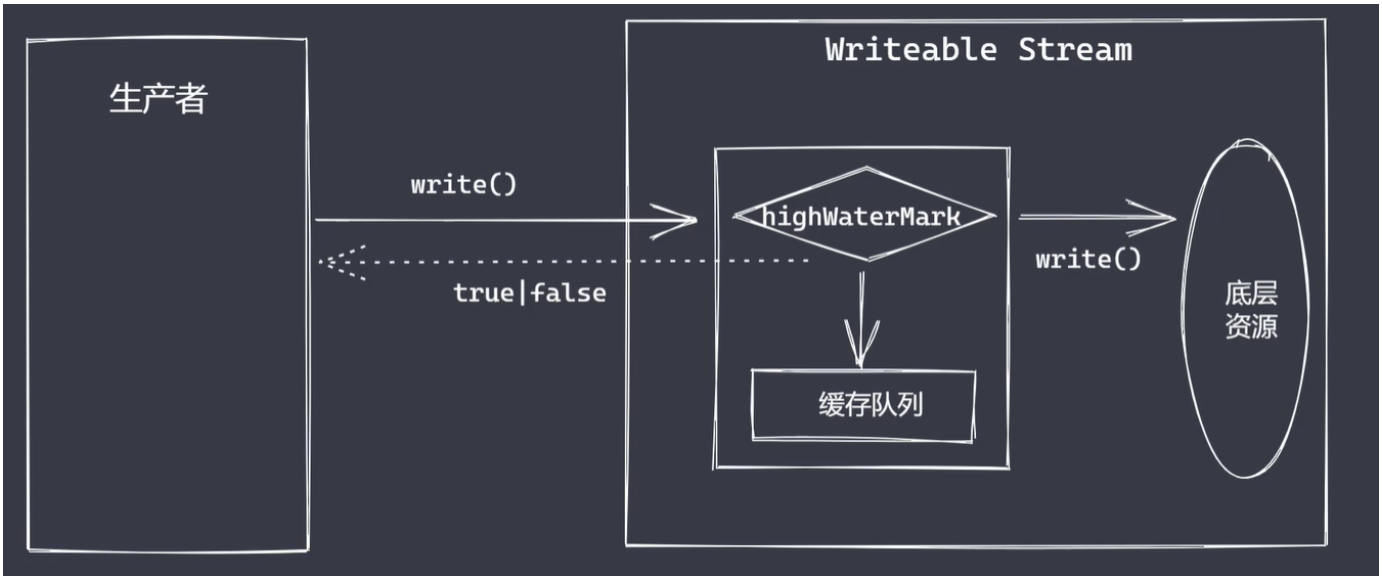
- 背压机制

  数据从磁盘里面被读取出来的速度 远远大于被 写入磁盘的速度

  消费者的速度远远跟不上生产者的速度， 就产生产能过剩

  当前不能被消化掉的数据，先缓存到一个队列里面，但是队列的大小是有上限的，如果不去实现一个**背压机制**很有可能出现内存溢出、gc频繁调用， 其他进程变慢， 需要让数据的生成者和消费者之间平滑流动的机制，这个就是背压机制


Readable Stream(流动模式)
消费者
resume
pause
highWaterMark
底层
数据
true|false
缓存
data事件


Writeable Stream
生产者
write()
highWaterMark
write()
底层
资源
true|false
缓存队列