Node.js中的流

fs模块中几种流的读写方法的区别

用途	异步	同步
将文件作为 整体 读入缓存区	readFile	readFileSync
将文件 部分 读入缓存区	read	readSync
将数据 完整 写入文件	writeFile	writeFileSync
将缓存区的 部分 内容写入文件	write	writeSync

read&readSync读取文件

- **一小块** 一小块读入缓存区
- 最后从从缓存区读取 **完整** 文件内容

write&writeSync写入文件

- 将需要书写的数据写到一个 **系统缓存区**
- 待缓存区写满后再将缓存区写到文件中
- 重复执行上面二个步骤

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
- 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。比如对一个 HTTP 服务器的请求对象request是一个流，stdout 也是一个流。
- 流是可读、可写或兼具两者的。所有流都是 **EventEmitter** 的实例。

> “ 几乎所有 Node 程序，无论多简单，都在某种途径用到了流

stream.Readable可读流

使用实现了stream.Readable接口的对象来将对象数据读取为流数据,在您表明您准备好接收之前，Readable 流并不会开始发射数据。

对象	用途
fs.ReadStream	读取文件
http.IncomingMessage	客户端的请求或服务器端的响应
net.Socket	tcp连接中的socket对象
process.stdin	标准输入流
Gzip	数据压缩

- 内部有flowing(**流动**)模式和非flowing(**暂停**)模式来读取数据
- `flowing` 模式使用操作系统的内部IO机制来读取数据，并尽可能 **快** 地提供给您
- 非 `flowing` 模式时流默认处于 **暂停** 模式,必须显式调用 `read` 方法来读取数据

> 注意：如果没有绑定 data 事件处理器，并且没有 pipe() 目标，同时流被切换到流动模式，那么数据会流失。

- 如何切换到流动模式
  - 添加一个 `data` 事件处理器来监听数据。
  - 调用 `resume()` 方法来明确开启数据流。
  - 调用 `pipe()` 方法将数据发送到一个 **Writable** 可写流。
- 切换回暂停模式
  - 如果没有导流目标，调用 `pause()` 方法。
  - 如果有导流目标，移除所有 data 事件处理器，调用 `unpipe()`

ReadStream文件可读流

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

- path 读取的 **文件路径**
- options
  - flags 对文件采取何种操作,默认为 'r'
  - encoding 指定 **编码** ，默认为null
  - autoClose 读完数据后是否关闭文件描述符
  - start 用整数表示文件 **开始** 读取的字节数的索引位置
  - end 用整数表示文件 **结束** 读取的字节数的索引位置(**包括end位置**)
  - highWaterMark 最高水位线，停止从底层资源读取前内部缓冲区最多能存放的字节数。缺省为 64kb

可读流触发的事件

事件	用途
readable	监听 `readable` 会使数据从底层读到系统缓存区，读到数据后或者排空后如果再读到数据，会触发 `readable` 事件
data	绑定一个 `data` 事件监听器到会将流切换到 **流动模式** ，数据会被尽可能快的读出
end	该事件会在 **读完** 数据后被触发
error	当数据接收时发生 **错误** 时触发
close	当底层数据源（比如，源头的文件描述符）被 **关闭** 时触发。并不是所有流都会触发这个事件

可读流的方法

方法	描述
read	在 `readable` 事件触发时的 **回调函数** 里读取数据
setEncoding	指定 **编码**
pause	通知对象 **停止** 触发data事件
resume	通知对象 **恢复** 触发data事件
pipe	设置 **管道** ，将可读流里的内容导入到参数指定的 **可写流** 里
unpipe	**取消** 数据通道
unshift	把数据块 **插回** 队列开头

可读流触发的事件

事件	用途
drain	`write` 返回 **false** 后触发，表示操作系统缓存区中的数据已经全部 **输出** 到目标对象中。
error	写入时 **错误** 时触发

Writable可写流

使用各种实现stream.Writable接口的对象来将流数据**写入**到对象中

对象	用途
fs.writeStream	写入文件
http.ClientRequest	客户端请求对象
http.ServerResponse	http中的响应对象
net.Socket	TCP中的socket对象
process.stdout	标准输出
process.stderr	错误输出
Gunzip	解压

可写流的方法

方法	描述
write	写入数据
end	结束写入 数据时触发。迫使缓存区中的数据立即写入目标对象，调用后不能再写入

WriteStream

在fs模块中使用`createWriteStream`方法创建一个将流数据写入文件中的`WriteStream`对象

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

- path 读取的文件路径
- options
  - flags 对文件采取何种**操作**,默认为 'w'
  - encoding 指定**编码** ，默认为null
  - autoClose 是否**关闭**文件描述符
  - start 用整数表示文件**开始**字节数的写入位置
  - highWaterMark 最高水位线，write()开始返回 false 的**缓冲大小**。缺省为 16kb

write方法

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

- 参数
  - chunk 要**写入**的数据，Buffer或字符串对象，必须指定
  - encoding 写入**编码**，chunk为字符串时有用，可选
  - callback 写入成功后的**回调**
- 返回值为布尔值，系统缓存区定满时为false,未满时为true

end方法

在写入文件时，当不再需要写入数据时可调用该方法关闭文件。 迫使系统缓存区的数据立即写入文件中。

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

大文件读取流程

1. 从文件读入 **缓存区** 并填满
2. 把缓存区中的数据写入目标文件，同时读取剩余数据到 **内存** 中，`write` 返回 `false`
3. 缓存区中的数据全部写入后触发 `drain` 事件
4. 先将内存中的数据写入缓存区，再读取文件剩余数据到缓存区直到填满
5. **持续** 上述步骤，直到读取完成

pipe

流，尤其是 pipe() 方法的初衷，是将数据的 **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。

```
readStream.pipe(writeStream,[options]);
```

- readStream可读流对象
- writeStream可写流对象
- options
    - end 为true时表示数据读取完毕后立刻将缓存区中的数据写入目标文件并 **关闭** 文件

自定义流

无论实现任何形式的流，模式都是一样的：

- 子类 **继承** 适合的父类(util.inherits)
- 在您的构造函数中调用 **父类的构造函数** ，以确保内部的机制被正确初始化。
- 自定义流的类别

使用情景	类	需要实现的方法
只读	Readable	_read
只写	Writable	_write
可读可写	Duplex	_read, _write
转换数据	Transform	_transform

readable可读流

- stream.Readable 是一个可被继承的、实现了底层方法 `_read(size)` 的抽象类
- 需要重写 `_read` 方法来从底层资源抓取数据
- options {Object}
  - highWaterMark {Number} 停止从底层资源读取前 **内部缓冲区** 最多能存放的字节数。缺省为 64kb
  - encoding {String} 若给出，则 Buffer 会被解码成所给 **编码** 的字符串。缺省为 null
  - objectMode {Boolean} 该流是否应该表现为 **对象** 的流。意思是说 stream.read(n) 返回一个 **单独的对象** ，而不是大小为 n 的 Buffer
- push() 方法会明确地向 **读取队列** 中插入一些数据
- 如果调用它时传入了 null 参数，那么它会触发数据 **结束** 信号。

Writable

stream.Writable 是一个可被继承的、实现了底层方法 `_write(chunk, encoding, callback)` 的抽象类。

```
writable._write(chunk, encoding, callback)
```

- chunk {Buffer | String} 要被写入的 **数据块**
- encoding {String} 如果数据块是字符串，则这里指定它的 **编码** 类型。
- callback {Function} 当您 **处理完** 所给数据块时调用此函数,使用标准的 `callback(error)` 形式来调用回调以表明写入成功完成或遇到错误。

stream.Duplex双工流

- “双工”（duplex）流同时兼具 **可读** 和 **可写** 特性，比如一个 TCP 的 **socket**。
- 实现了底层方法 `_read(size)` 和 `_write(chunk, encoding, callback)` 的抽象类。

stream.Transform转换流

- “转换”（ **transform** ）流实际上是一个输出与输入存在 **因果** 关系的双工流，比如 **zlib** 流或 **crypto** 流。
- 转换类必须实现 `_transform()` 方法

```
transform._transform(chunk, encoding, callback)
```

  - chunk {Buffer | String} 要被转换的 **数据块** 。
  - encoding {String} 如果数据块是一个字符串，那么这就是它的 **编码** 类型。
  - callback {Function} 当您 **处理完** 所提供的数据块时调用此函数。

stream.PassThrough(路过流?)

这是 Transform 流的一个简单实现，将**输入的字节简单地传递给输出**