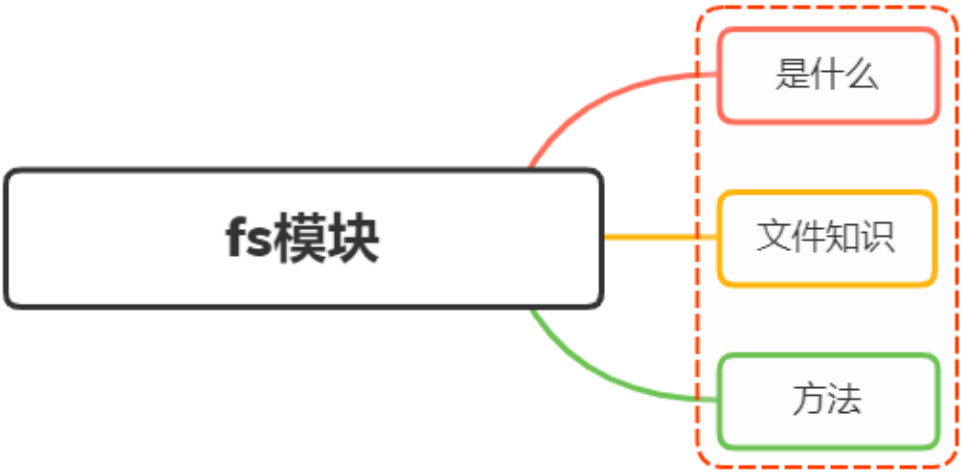
- 与 websocket 配合，开发长连接的实时交互应用程序

具体场景可以表现为如下：

- 第一大类：用户表单收集系统、后台管理系统、实时交互系统、考试系统、联网软件、高并发量的web应用程序
- 第二大类：基于web、canvas等多人联网游戏
- 第三大类：基于web的多人实时聊天客户端、聊天室、图文直播
- 第四大类：单页面浏览器应用程序
- 第五大类：操作数据库、为前端和移动端提供基于 `json` 的API

其实， `Nodejs` 能实现几乎一切的应用，只考虑适不适合使用它

2. 说说对 Node 中的 fs模块的理解？有哪些常用方法



2.1. 是什么

fs（filesystem），该模块提供本地文件的读写能力，基本上是 `POSIX` 文件操作命令的简单包装

可以说，所有与文件的操作都是通过 `fs` 核心模块实现

导入模块如下：

```
JavaScript | 复制代码
1  const fs = require('fs');
```

这个模块对所有文件系统操作提供异步（不具有 `sync` 后缀）和同步（具有 `sync` 后缀）两种操作方式，而供开发者选择

2.1.1. 文件知识

在计算机中有关于文件的知识：

- 权限位 mode
- 标识位 flag
- 文件描述为 fd

2.1.2. 权限位 mode

权限分配	文件所有者			文件所属组			其他用户		
权限项	读	写	执行	读	写	执行	读	写	执行
字符表示	r	w	x	r	w	x	r	w	x
数字表示	4	2	1	4	2	1	4	2	1

针对文件所有者、文件所属组、其他用户进行权限分配，其中类型又分成读、写和执行，具备权限位4、2、1，不具备权限为0

如在 `linux` 查看文件权限位：

JavaScript | 复制代码

```
drwxr-xr-x 1 PandaShen 197121 0 Jun 28 14:41 core
-rw-r--r-- 1 PandaShen 197121 293 Jun 23 17:44 index.md
```

在开头前十位中， `d` 为文件夹， `-` 为文件，后九位就代表当前用户、用户所属组和其他用户的权限位，按每三位划分，分别代表读（r）、写（w）和执行（x），– 代表没有当前位对应的权限

2.1.3. 标识位

标识位代表着对文件的操作方式，如可读、可写、即可读又可写等等，如下表所示：

符号	含义
r	读取文件，如果文件不存在则抛出异常。
r+	读取并写入文件，如果文件不存在则抛出异常。
rs	读取并写入文件，指示操作系统绕开本地文件系统缓存。

w	写入文件，文件不存在会被创建，存在则清空后写入。
wx	写入文件，排它方式打开。
w+	读取并写入文件，文件不存在则创建文件，存在则清空后写入。
wx+	和 w+ 类似，排他方式打开。
a	追加写入，文件不存在则创建文件。
ax	与 a 类似，排他方式打开。
a+	读取并追加写入，不存在则创建。
ax+	与 a+ 类似，排他方式打开。

2.1.4. 文件描述为 fd

操作系统会为每个打开的文件分配一个名为文件描述符的数值标识，文件操作使用这些文件描述符来识别与追踪每个特定的文件

`Window` 系统使用了一个不同但概念类似的机制来追踪资源，为方便用户， `NodeJS` 抽象了不同操作系统间的差异，为所有打开的文件分配了数值的文件描述符

在 `NodeJS` 中，每操作一个文件，文件描述符是递增的，文件描述符一般从 `3` 开始，因为前面有 `0` 、 `1` 、 `2` 三个比较特殊的描述符，分别代表 `process.stdin` （标准输入）、 `process.stdout` （标准输出）和 `process.stderr` （错误输出）

2.2. 方法

下面针对 `fs` 模块常用的方法进行展开：

- 文件读取
- 文件写入
- 文件追加写入
- 文件拷贝
- 创建目录

2.2.1. 文件读取

2.2.1.1. fs.readFileSync

同步读取，参数如下：

- 第一个参数为读取文件的路径或文件描述符
- 第二个参数为 options，默认值为 null，其中有 encoding（编码，默认为 null）和 flag（标识位，默认为 r），也可直接传入 encoding

结果为返回文件的内容

JavaScript 复制代码

```javascript
const fs = require("fs");

let buf = fs.readFileSync("1.txt");
let data = fs.readFileSync("1.txt", "utf8");

console.log(buf); // <Buffer 48 65 6c 6c 6f>
console.log(data); // Hello
```

2.2.1.2. fs.readFile

异步读取方法 `readFile` 与 `readFileSync` 的前两个参数相同，最后一个参数为回调函数，函数内有两个参数 `err`（错误）和 `data`（数据），该方法没有返回值，回调函数在读取文件成功后执行

JavaScript 复制代码

```javascript
const fs = require("fs");

fs.readFile("1.txt", "utf8", (err, data) => {
    if(!err){
        console.log(data); // Hello
    }
});
```

2.2.2. 文件写入

2.2.2.1. writeFileSync

同步写入，有三个参数：

- 第一个参数为写入文件的路径或文件描述符
- 第二个参数为写入的数据，类型为 String 或 Buffer
- 第三个参数为 options，默认值为 null，其中有 encoding（编码，默认为 utf8）、 flag（标识位，默认为 w）和 mode（权限位，默认为 0o666），也可直接传入 encoding

JavaScript 复制代码

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

fs.writeFileSync("2.txt", "Hello world");
let data = fs.readFileSync("2.txt", "utf8");

console.log(data); // Hello world
```

2.2.2.2. writeFile

异步写入，`writeFile` 与 `writeFileSync` 的前三个参数相同，最后一个参数为回调函数，函数内有一个参数 `err` （错误），回调函数在文件写入数据成功后执行

JavaScript 复制代码

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

fs.writeFile("2.txt", "Hello world", err => {
    if (!err) {
        fs.readFile("2.txt", "utf8", (err, data) => {
            console.log(data); // Hello world
        });
    }
});
```

2.2.3. 文件追加写入

2.2.3.1. appendFileSync

参数如下：

- 第一个参数为写入文件的路径或文件描述符
- 第二个参数为写入的数据，类型为 String 或 Buffer
- 第三个参数为 options，默认值为 null，其中有 encoding（编码，默认为 utf8）、 flag（标识位，

默认为 a）和 mode（权限位，默认为 0o666），也可直接传入 encoding

```JavaScript
const fs = require("fs");

fs.appendFileSync("3.txt", " world");
let data = fs.readFileSync("3.txt", "utf8");
```

2.2.3.2. appendFile

异步追加写入方法 `appendFile` 与 `appendFileSync` 的前三个参数相同，最后一个参数为回调函数，函数内有一个参数 `err`（错误），回调函数在文件追加写入数据成功后执行

```JavaScript
const fs = require("fs");

fs.appendFile("3.txt", " world", err => {
    if (!err) {
        fs.readFile("3.txt", "utf8", (err, data) => {
            console.log(data); // Hello world
        });
    }
});
```

2.2.4. 文件拷贝

2.2.4.1. copyFileSync

同步拷贝

```JavaScript
const fs = require("fs");

fs.copyFileSync("3.txt", "4.txt");
let data = fs.readFileSync("4.txt", "utf8");

console.log(data); // Hello world
```

2.2.4.2. copyFile

异步拷贝

```JavaScript
const fs = require("fs");

fs.copyFile("3.txt", "4.txt", () => {
    fs.readFile("4.txt", "utf8", (err, data) => {
        console.log(data); // Hello world
    });
});
```

2.2.5. 创建目录

2.2.5.1. mkdirSync

同步创建，参数为一个目录的路径，没有返回值，在创建目录的过程中，必须保证传入的路径前面的文件目录都存在，否则会抛出异常

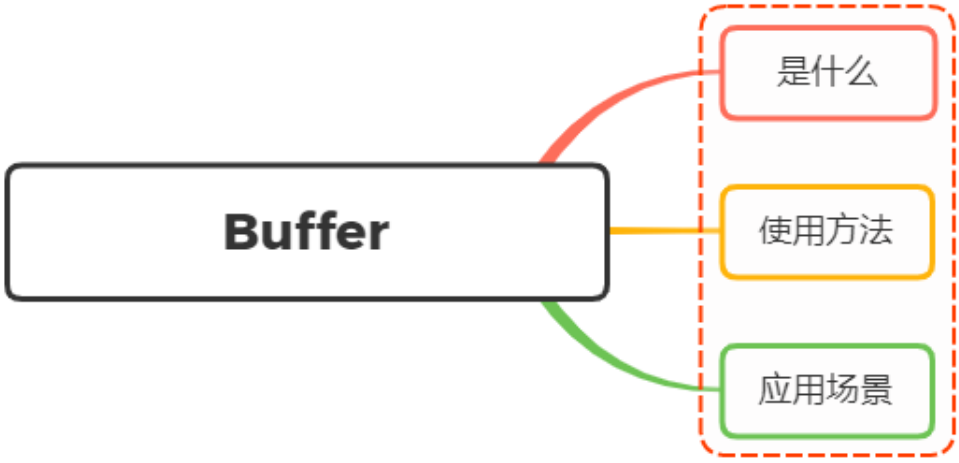
```JavaScript
// 假设已经有了 a 文件夹和 a 下的 b 文件夹
fs.mkdirSync("a/b/c")
```

2.2.5.2. mkdir

异步创建，第二个参数为回调函数

```JavaScript
fs.mkdir("a/b/c", err => {
    if (!err) console.log("创建成功");
});
```

3. 说说对 Node 中的 Buffer 的理解？应用场景？



3.1. 是什么

在 `Node` 应用中，需要处理网络协议、操作数据库、处理图片、接收上传文件等，在网络流和文件的操作中，要处理大量二进制数据，而 `Buffer` 就是在内存中开辟一片区域（初次初始化为8KB），用来存放二进制数据

在上述操作中都会存在数据流动，每个数据流动的过程中，都会有一个最小或最大数据量

如果数据到达的速度比进程消耗的速度快，那么少数早到达的数据会处于等待区等候被处理。反之，如果数据到达的速度比进程消耗的数据慢，那么早先到达的数据需要等待一定量的数据到达之后才能被处理

这里的等待区就指的缓冲区（Buffer），它是计算机中的一个小物理单位，通常位于计算机的 `RAM` 中

简单来讲， `Nodejs` 不能控制数据传输的速度和到达时间，只能决定何时发送数据，如果还没到发送时间，则将数据放在 `Buffer` 中，即在 `RAM` 中，直至将它们发送完毕

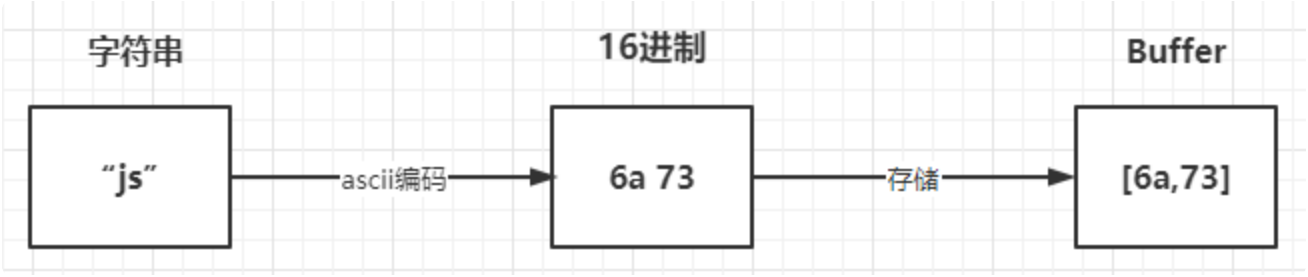
上面讲到了 `Buffer` 是用来存储二进制数据，其的形式可以理解成一个数组，数组中的每一项，都可以保存8位二进制： `00000000` ，也就是一个字节

例如：

JavaScript | 复制代码

```
1  const buffer = Buffer.from("why")
```

其存储过程如下图所示：



3.2. 使用方法

`Buffer` 类在全局作用域中，无须 `require` 导入

创建 `Buffer` 的方法有很多种，我们讲讲下面的两种常见的形式：

- Buffer.from()
- Buffer.alloc()

3.2.1. Buffer.from()

JavaScript | 复制代码

```
const b1 = Buffer.from('10');
const b2 = Buffer.from('10', 'utf8');
const b3 = Buffer.from([10]);
const b4 = Buffer.from(b3);

console.log(b1, b2, b3, b4); // <Buffer 31 30> <Buffer 31 30> <Buffer 0a> <Buffer 0a>
```

3.2.2. Buffer.alloc()

JavaScript | 复制代码

```
const bAlloc1 = Buffer.alloc(10); // 创建一个大小为 10 个字节的缓冲区
const bAlloc2 = Buffer.alloc(10, 1); // 建一个长度为 10 的 Buffer,其中全部填充了值为 `1` 的字节
console.log(bAlloc1); // <Buffer 00 00 00 00 00 00 00 00 00 00>
console.log(bAlloc2); // <Buffer 01 01 01 01 01 01 01 01 01 01>
```

在上面创建 `buffer` 后，则能够 `toString` 的形式进行交互，默认情况下采取 `utf8` 字符编码形式，如下

```JavaScript
const buffer = Buffer.from("你好");
console.log(buffer);
// <Buffer e4 bd a0 e5 a5 bd>
const str = buffer.toString();
console.log(str);
// 你好
```

如果编码与解码不是相同的格式则会出现乱码的情况，如下：

```JavaScript
const buffer = Buffer.from("你好","utf-8 ");
console.log(buffer);
// <Buffer e4 bd a0 e5 a5 bd>
const str = buffer.toString("ascii");
console.log(str);
// d= e%=
```

当设定的范围导致字符串被截断的时候，也会存在乱码情况，如下：

```JavaScript
const buf = Buffer.from('Node.js 技术栈', 'UTF-8');

console.log(buf)          // <Buffer 4e 6f 64 65 2e 6a 73 20 e6 8a 80 e6 9c af e6 a0 88>
console.log(buf.length)   // 17

console.log(buf.toString('UTF-8', 0, 9))  // Node.js 
console.log(buf.toString('UTF-8', 0, 11)) // Node.js 技
```

所支持的字符集有如下：

- ascii：仅支持 7 位 ASCII 数据，如果设置去掉高位的话，这种编码是非常快的
- utf8：多字节编码的 Unicode 字符，许多网页和其他文档格式都使用 UTF-8
- utf16le：2 或 4 个字节，小字节序编码的 Unicode 字符，支持代理对（U+10000至 U+10FFFF）
- ucs2，utf16le 的别名
- base64：Base64 编码
- latin：一种把 Buffer 编码成一字节编码的字符串的方式
- binary：latin1 的别名，
- hex：将每个字节编码为两个十六进制字符

3.3. 应用场景

`Buffer` 的应用场景常常与流的概念联系在一起，例如有如下：

- I/O操作
- 加密解密
- zlib.js

3.3.1. I/O操作

通过流的形式，将一个文件的内容读取到另外一个文件

JavaScript | 复制代码

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

const inputStream = fs.createReadStream('input.txt'); // 创建可读流
const outputStream = fs.createWriteStream('output.txt'); // 创建可写流

inputStream.pipe(outputStream); // 管道读写
```

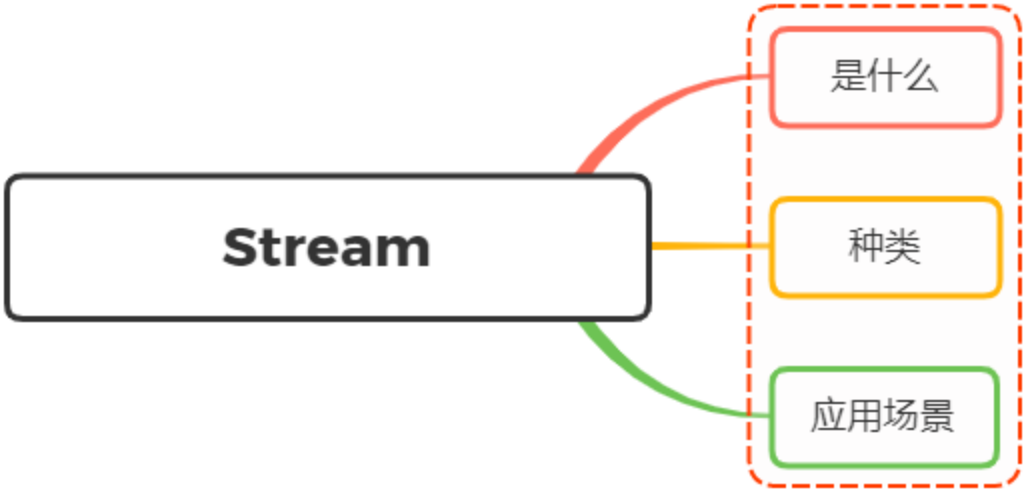
3.3.2. 加解密

在一些加解密算法中会遇到使用 `Buffer`，例如 `crypto.createCipheriv` 的第二个参数 `key` 为 `string` 或 `Buffer` 类型

3.3.3. zlib.js

`zlib.js` 为 `Node.js` 的核心库之一，其利用了缓冲区（`Buffer`）的功能来操作二进制数据流，提供了压缩或解压功能

4. 说说对 Node 中的 Stream 的理解？应用场景？



4.1. 是什么

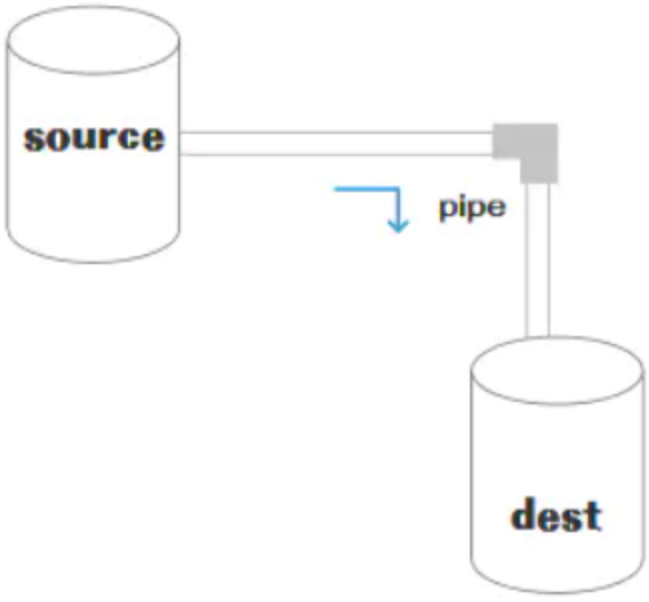
流（Stream），是一个数据传输手段，是端到端信息交换的一种方式，而且是有顺序的,是逐块读取数据、处理内容，用于顺序读取输入或写入输出

`Node.js` 中很多对象都实现了流，总之它是会冒数据（以 `Buffer` 为单位）

它的独特之处在于，它不像传统的程序那样一次将一个文件读入内存，而是逐块读取数据、处理其内容，而不是将其全部保存在内存中

流可以分成三部分： `source` 、 `dest` 、 `pipe`

在 `source` 和 `dest` 之间有一个连接的管道 `pipe` ,它的基本语法是 `source.pipe(dest)` ， `source` 和 `dest` 就是通过pipe连接，让数据从 `source` 流向了 `dest` ，如下图所示：



4.2. 种类

在 `NodeJS` ，几乎所有的地方都使用到了流的概念，分成四个种类：

- 可写流：可写入数据的流。例如 fs.createWriteStream() 可以使用流将数据写入文件
- 可读流： 可读取数据的流。例如fs.createReadStream() 可以从文件读取内容
- 双工流： 既可读又可写的流。例如 net.Socket
- 转换流： 可以在数据写入和读取时修改或转换数据的流。例如，在文件压缩操作中，可以向文件写入压缩数据，并从文件中读取解压数据

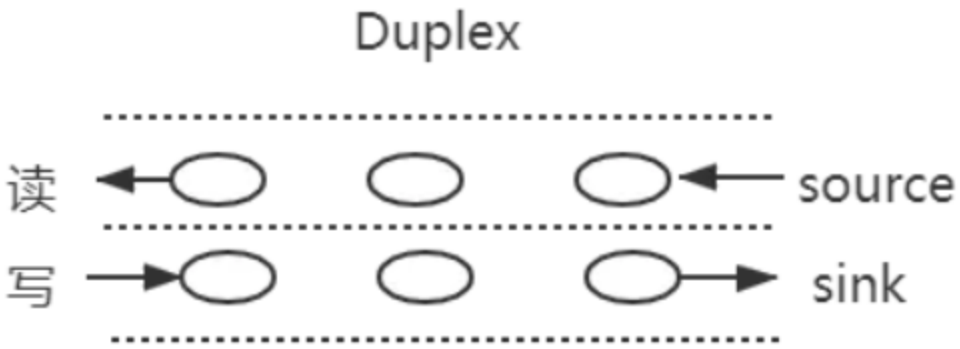
在 `NodeJS` 中 `HTTP` 服务器模块中， `request` 是可读流， `response` 是可写流。还有 `fs` 模块，能同时处理可读和可写文件流

可读流和可写流都是单向的，比较容易理解，而另外两个是双向的

4.2.1. 双工流

之前了解过 `websocket` 通信，是一个全双工通信，发送方和接受方都是各自独立的方法，发送和接收都没有任何关系

如下图所示：



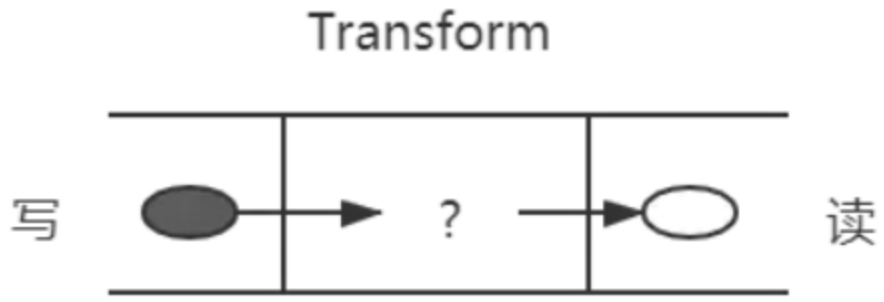
基本代码如下：

```JavaScript
const { Duplex } = require('stream');

const myDuplex = new Duplex({
  read(size) {
    // ...
  },
  write(chunk, encoding, callback) {
    // ...
  }
});
```

4.2.2. 双工流

双工流的演示图如下所示：



除了上述压缩包的例子，还比如一个 `babel`，把 `es6` 转换为，我们在左边写入 `es6`，从右边读取 `es5`

基本代码如下所示：

```JavaScript
const { Transform } = require('stream');

const myTransform = new Transform({
  transform(chunk, encoding, callback) {
    // ...
  }
});
```

4.3. 应用场景

`stream` 的应用场景主要就是处理 `IO` 操作，而 `http` 请求和文件操作都属于 `IO` 操作

试想一下，如果一次 `IO` 操作过大，硬件的开销就过大，而将此次大的 `IO` 操作进行分段操作，让数据像水管一样流动，直到流动完成

常见的场景有：

- get请求返回文件给客户端
- 文件操作
- 一些打包工具的底层操作

4.3.1. get请求返回文件给客户端

使用 `stream` 流返回文件， `res` 也是一个 `stream` 对象，通过 `pipe` 管道将文件数据返回

JavaScript | 复制代码

```
const server = http.createServer(function (req, res) {
    const method = req.method; // 获取请求方法
    if (method === 'GET') { // get 请求
        const fileName = path.resolve(__dirname, 'data.txt');
        let stream = fs.createReadStream(fileName);
        stream.pipe(res); // 将 res 作为 stream 的 dest
    }
});
server.listen(8000);
```

4.3.2. 文件操作

创建一个可读数据流 `readStream` ，一个可写数据流 `writeStream` ，通过 `pipe` 管道把数据流转过去

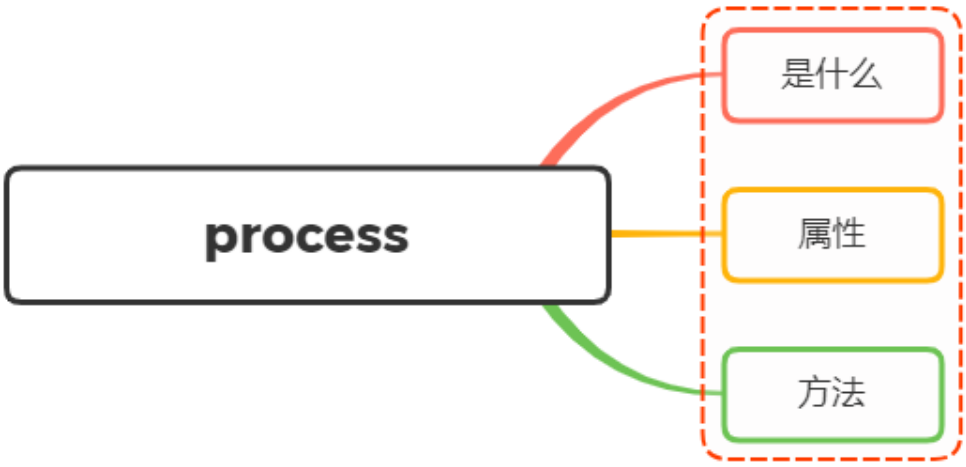
```javascript
const fs = require('fs')
const path = require('path')

// 两个文件名
const fileName1 = path.resolve(__dirname, 'data.txt')
const fileName2 = path.resolve(__dirname, 'data-bak.txt')
// 读取文件的 stream 对象
const readStream = fs.createReadStream(fileName1)
// 写入文件的 stream 对象
const writeStream = fs.createWriteStream(fileName2)
// 通过 pipe执行拷贝，数据流转
readStream.pipe(writeStream)
// 数据读取完成监听，即拷贝完成
readStream.on('end', function () {
    console.log('拷贝完成')
})
```

4.3.3. 一些打包工具的底层操作

目前一些比较火的前端打包构建工具，都是通过 `node.js` 编写的，打包和构建的过程肯定是文件频繁操作的过程，离不来 `stream` ，如 `gulp`

5. 说说对 Node 中的 process 的理解？有哪些常用方法？



5.1. 是什么

`process` 对象是一个全局变量，提供了有关当前 `Node.js` 进程的信息并对其进行控制，作为一个全局变量

我们都知道，进程计算机系统进行资源分配和调度的基本单位，是操作系统结构的基础，是线程的容器

当我们启动一个 `js` 文件，实际就是开启了一个服务进程，每个进程都拥有自己的独立空间地址、数据栈，像另一个进程无法访问当前进程的变量、数据结构，只有数据通信后，进程之间才可以数据共享

由于 `JavaScript` 是一个单线程语言，所以通过 `node xxx` 启动一个文件后，只有一条主线程

5.2. 属性与方法

关于 `process` 常见的属性有如下：

- process.env：环境变量，例如通过 `process.env.NODE_ENV 获取不同环境项目配置信息
- process.nextTick：这个在谈及 `EventLoop` 时经常为会提到
- process.pid：获取当前进程id
- process.ppid：当前进程对应的父进程
- process.cwd()：获取当前进程工作目录，
- process.platform：获取当前进程运行的操作系统平台
- process.uptime()：当前进程已运行时间，例如：pm2 守护进程的 uptime 值
- 进程事件： process.on('uncaughtException',cb) 捕获异常信息、 process.on('exit',cb) 进程推出监听
- 三个标准流： process.stdout 标准输出、 process.stdin 标准输入、 process.stderr 标准错误输出
- process.title 指定进程名称，有的时候需要给进程指定一个名称

下面再稍微介绍下某些方法的使用：

5.2.1. process.cwd()

返回当前 `Node` 进程执行的目录

一个 `Node` 模块 `A` 通过 NPM 发布，项目 `B` 中使用了模块 `A` 。在 `A` 中需要操作 `B` 项目下的文件时，就可以用 `process.cwd()` 来获取 `B` 项目的路径

5.2.2. process.argv

在终端通过 Node 执行命令的时候，通过 `process.argv` 可以获取传入的命令行参数，返回值是一个数组：

- 0: Node 路径（一般用不到，直接忽略）
- 1: 被执行的 JS 文件路径（一般用不到，直接忽略）
- 2~n: 真实传入命令的参数

所以，我们只要从 `process.argv[2]` 开始获取就好了

JavaScript | 复制代码

```
const args = process.argv.slice(2);
```

5.2.3. process.env

返回一个对象，存储当前环境相关的所有信息，一般很少直接用到。

一般我们会在 `process.env` 上挂载一些变量标识当前的环境。比如最常见的用 `process.env.NODE_ENV` 区分 `development` 和 `production`

在 `vue-cli` 的源码中也经常会看到 `process.env.VUE_CLI_DEBUG` 标识当前是不是 `DEBUG` 模式

5.2.4. process.nextTick()

我们知道 `NodeJs` 是基于事件轮询，在这个过程中，同一时间只会处理一件事情

在这种处理模式下， `process.nextTick()` 就是定义出一个动作，并且让这个动作在下一个事件轮询的时间点上执行

例如下面例子将一个 `foo` 函数在下一个时间点调用

JavaScript | 复制代码

```
function foo() {
    console.error('foo');
}

process.nextTick(foo);
console.error('bar');
```

输出结果为 `bar` 、 `foo`

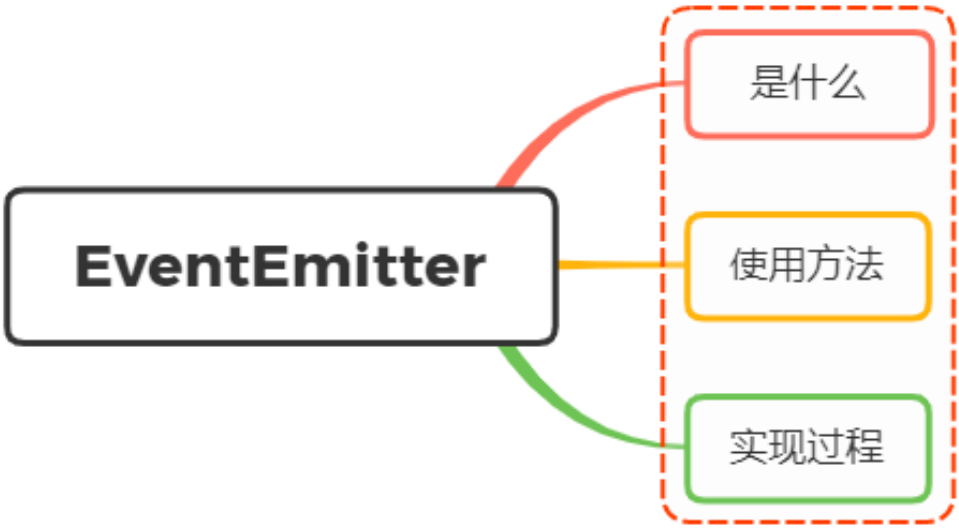
虽然下述方式也能实现同样效果：

```
JavaScript | 复制代码
1  setTimeout(foo, 0);
2  console.log('bar');
```

两者区别在于：

- process.nextTick()会在这一次event loop的call stack清空后（下一次event loop开始前）再调用callback
- setTimeout()是并不知道什么时候call stack清空的，所以何时调用callback函数是不确定的

6. 说说Node中的EventEmitter? 如何实现一个EventEmitter?



6.1. 是什么

我们了解到， `Node` 采用了事件驱动机制，而 `EventEmitter` 就是 `Node` 实现事件驱动的基础

在 `EventEmitter` 的基础上， `Node` 几乎所有的模块都继承了这个类，这些模块拥有了自己的事件，可以绑定 / 触发监听器，实现了异步操作

`Node.js` 里面的许多对象都会分发事件，比如 fs.readStream 对象会在文件被打开的时候触发一个事件

这些产生事件的对象都是 events.EventEmitter 的实例，这些对象有一个 eventEmitter.on() 函数，用于将一个或多个函数绑定到命名事件上

6.2. 使用方法

`Node` 的 `events` 模块只提供了一个 `EventEmitter` 类，这个类实现了 `Node` 异步事件驱动架构的基本模式——观察者模式

在这种模式中，被观察者(主体)维护着一组其他对象派来(注册)的观察者，有新的对象对主体感兴趣就注册观察者，不感兴趣就取消订阅，主体有更新的话就依次通知观察者们

基本代码如下所示：

JavaScript | 复制代码

```
const EventEmitter = require('events')

class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter()

function callback() {
    console.log('触发了event事件! ')
}
myEmitter.on('event', callback)
myEmitter.emit('event')
myEmitter.removeListener('event', callback);
```

通过实例对象的 `on` 方法注册一个名为 `event` 的事件，通过 `emit` 方法触发该事件，而 `removeListener` 用于取消事件的监听

关于其常见的方法如下：

- emitter.addListener/on(eventName, listener) ： 添加类型为 eventName 的监听事件到事件数组尾部
- emitter.prependListener(eventName, listener)：添加类型为 eventName 的监听事件到事件数组头部
- emitter.emit(eventName[, ...args])：触发类型为 eventName 的监听事件
- emitter.removeListener/off(eventName, listener)：移除类型为 eventName 的监听事件
- emitter.once(eventName, listener)：添加类型为 eventName 的监听事件，以后只能执行一次并删除
- emitter.removeAllListeners([eventName])： 移除全部类型为 eventName 的监听事件

6.3. 实现过程

通过上面的方法了解， `EventEmitter` 是一个构造函数，内部存在一个包含所有事件的对象

```JavaScript
class EventEmitter {
    constructor() {
        this.events = {};
    }
}
```

其中 `events` 存放的监听事件的函数的结构如下：

```JavaScript
{
  "event1": [f1,f2,f3],
  "event2": [f4,f5],
  ...
}
```

然后开始一步步实现实例方法，首先是 `emit` ，第一个参数为事件的类型，第二个参数开始为触发事件函数的参数，实现如下：

```JavaScript
emit(type, ...args) {
    this.events[type].forEach((item) => {
        Reflect.apply(item, this, args);
    });
}
```

当实现了 `emit` 方法之后，然后实现 `on` 、 `addListener` 、 `prependListener` 这三个实例方法，都是添加事件监听触发函数，实现也是大同小异

```JavaScript
on(type, handler) {
    if (!this.events[type]) {
        this.events[type] = [];
    }
    this.events[type].push(handler);
}

addListener(type,handler){
    this.on(type,handler)
}

prependListener(type, handler) {
    if (!this.events[type]) {
        this.events[type] = [];
    }
    this.events[type].unshift(handler);
}
```

紧接着就是实现事件监听的方法 `removeListener/on`

```JavaScript
removeListener(type, handler) {
    if (!this.events[type]) {
        return;
    }
    this.events[type] = this.events[type].filter(item => item !== handler)
;
}

off(type,handler){
    this.removeListener(type,handler)
}
```

最后再来实现 `once` 方法， 再传入事件监听处理函数的时候进行封装，利用闭包的特性维护当前状态，通过 `fired` 属性值判断事件函数是否执行过

JavaScript | 复制代码

```
once(type, handler) {
    this.on(type, this._onceWrap(type, handler, this));
  }

  _onceWrap(type, handler, target) {
    const state = { fired: false, handler, type , target};
    const wrapFn = this._onceWrapper.bind(state);
    state.wrapFn = wrapFn;
    return wrapFn;
  }

  _onceWrapper(...args) {
    if (!this.fired) {
      this.fired = true;
      Reflect.apply(this.handler, this.target, args);
      this.target.off(this.type, this.wrapFn);
    }
  }
```

完整代码如下：

JavaScript | 复制代码

```javascript
class EventEmitter {
    constructor() {
        this.events = {};
    }

    on(type, handler) {
        if (!this.events[type]) {
            this.events[type] = [];
        }
        this.events[type].push(handler);
    }

    addListener(type,handler){
        this.on(type,handler)
    }

    prependListener(type, handler) {
        if (!this.events[type]) {
            this.events[type] = [];
        }
        this.events[type].unshift(handler);
    }

    removeListener(type, handler) {
        if (!this.events[type]) {
            return;
        }
        this.events[type] = this.events[type].filter(item => item !== handler);
    }

    off(type,handler){
        this.removeListener(type,handler)
    }

    emit(type, ...args) {
        this.events[type].forEach((item) => {
            Reflect.apply(item, this, args);
        });
    }

    once(type, handler) {
        this.on(type, this._onceWrap(type, handler, this));
    }

```

```
    _onceWrap(type, handler, target) {
        const state = { fired: false, handler, type , target};
        const wrapFn = this._onceWrapper.bind(state);
        state.wrapFn = wrapFn;
        return wrapFn;
    }

    _onceWrapper(...args) {
        if (!this.fired) {
            this.fired = true;
            Reflect.apply(this.handler, this.target, args);
            this.target.off(this.type, this.wrapFn);
        }
    }
}
```

测试代码如下：

JavaScript | 复制代码

```
const ee = new EventEmitter();

// 注册所有事件
ee.once('wakeUp', (name) => { console.log(`${name} 1`); });
ee.on('eat', (name) => { console.log(`${name} 2`) });
ee.on('eat', (name) => { console.log(`${name} 3`) });
const meetingFn = (name) => { console.log(`${name} 4`) };
ee.on('work', meetingFn);
ee.on('work', (name) => { console.log(`${name} 5`) });

ee.emit('wakeUp', 'xx');
ee.emit('wakeUp', 'xx');          // 第二次没有触发
ee.emit('eat', 'xx');
ee.emit('work', 'xx');
ee.off('work', meetingFn);         // 移除事件
ee.emit('work', 'xx');            // 再次工作
```

7. 说说 Node 文件查找的优先级以及 Require 方法的文件查找策略?