Node.js

1.常见命令行指令

```
node -v  //查看node版本号(检测node是否安装成功)
npm -v   // npm 版本号
dir  //查看当前路径所有文件夹
cd  // 进入文件夹
cd .. //返回上一层文件夹
cls  // 清楚控制台记录
```

2.commonJS导入与导出

- **main.js**

```
//导入a.js
const a = require('./a.js')  //后缀名可以省略
const {a_name:LH} = require('./a.js'); //冒号后面取别名
const b = require('./b.js')
const {name,className} = require('./c.js');
console.log("a:",a);
console.log("a_name of LH",LH) //打印"李四"
console.log("b:",b);
```

- **a.js**

```
console.log('I am a file');
const obj = {
    name:"暖心",
    age: 22
}

// 1.module.exports 可以使用命名导出
  module.exports.a_name = "张";
  module.exports.a_className = "1班";
  module.exports.a_score = 99;

```

```
// 2.module.exports 可以直接使用等号
// 注意 :一旦使用等号方式导出,那么会覆盖之前所有命名导出的内容
// module.exports 直接使用等号做好写一次,node只认为最后一次导出
   module.exports = obj;  // 会覆盖
   module.exports = "b中的函数"; //会显示"b中的函数"

// 3.如果你在前面使用,module.exports = xxx 这种,那么你可以在它之后继续使用命名导出
   module.exports.a_name ="李四"; //不会覆盖
```

- **b.js**

```
  //1.命名导出
  exports.name = "wang"
  exports.className = "2班"

  //2.exports 不能直接使用等号 不起作用
  exports = 100;
  exports = {
        name:"123"
  }
  exports = ()⇒{};
```

- **总结**
  - **一切都指向module.exports**

- **解构赋值**

```
const c = 3;
const obj = {
        a: 1,
        b: 2,
        c
}

console.log(obj.a);  //1
console.log(obj['a']); //1
const key = 'a';
console.log(obj[key]); //1

const new_obj  = obj;
console.log(new_obj.a,new_obj.b); // 1 2
const {a,b,c:c_val} = obj;  // 对象解构
console.log(a,b,c_val); // 1 2 3
```

- **配置文件**
  - 命令

```
npm init -y;
```

  - “scripts”

```
 "dev": "node main.js"
// 运行main.js
 npm run dev;
"start":"node main.js"
//运行main.js
 npm start;
```

  - “type”

```
默认“commonjs” //commonJS
设置“module” // ESM
```

3. CommonJS解决依赖地域

- **main.js**

```
const a = require('./a.js');
const b = require('./b.js');

console.log("a→",JSON.stringify(a,null,2));
console.log("b→",JSON.stringify(b,null,2));

```

- **a.js**

```
exports.loaded = false;
const b = require('./b.js');
module.exports = {
    loaded: true,
    b
};

```

- **b.js**

```
exports.loaded = false;
const a = require('./a.js');
// console.log("在b文件中,获取的a文件是已执行过的a文件的内容," +
//  "有可能不是全部",a);
module.exports = {
    loaded: true,
    a
};
/*
  commonJS 的解决"依赖地域"的方案
  1.可以解决(不会报错),但解决的方案并不完美
  2.执行结果,取决require 的导入顺序
*/
```

- **打印结果**

```
a→ {
  "loaded": true,
  "b": {
    "loaded": true,
    "a": {
      "loaded": false
    }
  }
}
b→ {
  "loaded": true,
  "a": {
    "loaded": false
  }
```

4.ESM导入与导出

- ```
  package.json 中的type 如果是module 则不能使用commonjs语法
  ESM 中导入或加载文件必须写后缀名,而且后缀名必须是js
  ESM语法
     import  导入模块
     export  导出模块
     export default  导出默认模块(一个文件只能有一个默认模块)
  ```

- **main.js**

```
import './a.js';

import a from "./a.js";

//const {name:b_name,arr} = require("./b.js"); 这种是commonJS 起别名写法
import {name as b_name,arr} from './b.js'

import c ,{name as c_name } from './c.js'
/*
   ESM中的通配符*, 代表导入文件中的所有导出变量(export和export default)
   export default 导出的内容包含default属性里面
*/
import d,* as d_all from './d.js'
/*
   解构赋值时,只能获取export 导出的内容,
   export default 导出的不能直接通过default 获取,因为语法会报错
*/

console.log("a",a); //a { name: 'ESM', obj: { a: 1, b: 2 }, sum: [Function: sum] }

console.log("b的解构赋值",b_name); // b is a file
console.log("b的解构赋值",arr);//[1,2,3]

console.log("c",c); //c { name: 'c file,默认导出', age: 18, arr: [ 1, 2, 3 ] }
console.log("c的解构赋值",c_name); // c file
```
```

```
26
27 console.log("d的通配符",d_all); //{ name: 'd file', arr:
   [ 1, 2, 3 ] }
28 console.log("d的通配符",d_all.sub1(10,20)); //30
29 console.log("d的通配符",d_all.sub2(30,50));//-20
30 console.log("d的通配符
   add",d_all.default.add1(10,20));//30
31 console.log("d add",d.add1(10,20));//30
32 console.log("d add",d.add2(30,50));//-20
33
```

- **a.js**

```
1  console.log("a file");
2
3  const name = "ESM";
4
5  const obj  ={
6      a:1,
7      b:2
8  }
9
10 const sum  = (a,b) => a+b;
11
```

- **b.js**

```
1 export const name = 'b is a file';
2
3 export const arr = [1, 2, 3];
```

- **c.js**

```
1 export const name = 'c file';
2
3 export default {
4     name: 'c file,默认导出',
5     age: 18,
6     arr: [1,2,3]
7 }
```

- **d.js**

```
export const sub1 = (a, b) => a - b;
export const sub2 = (a, b) => a - b;

export const name = 'd file';

export default {
    add1: (a, b) => a + b,
    add2: (a, b) => a + b,
    add3: (a, b) => a + b,
}
```

5.CommonJS与ESM区别

> ESM 导入/导出的内容是只读的,如果你想修改的话,只能在声明文件中修改,如果修改成功,那么其他引入该内容的文件的值也相应修改
> ESM 导入是静态的(先分析项目中所有的import,然后在按顺序执行import的文件,最后执行入口文件)
> commonJS 导入/导出的内容不是原内容而是原内容的副本,也就是说我们可以在require导入之后,自由修改变量,不用担心影响到其他文件
> commonJS 导入是动态的,根据'依赖地域'来理解

- **CommonJS**
  - **main.js**

```
console.log("加载main.js");
const {count,msg,add,updateMsg} = require ('./a.js');
require ('./b.js');

add('main');

console.log("index文件中的count:",count);
console.log("index文件中的msg:",msg);
console.log("加载完毕main.js");
```

  - **a.js**

```
console.log('加载a.js'); // 加载a.js
let count = 0;
exports.count = count;

exports.msg = '初始化'

exports.add = (who) => {
    count++;
    console.log(`${who}调用了我修改count`)
};
exports.updateMsg = () => msg = "修改了msg";

console.log('a.js加载完毕'); // a.js加载完毕
```

- **b.js**

```
console.log('加载b.js'); // 加载b.js
const {count,msg,add,updateMsg} = require('./a.js');

console.log("b文件中的count:",count); //0
console.log("b文件中的msg:",msg); //初始化
console.log("加载完毕b.js"); //加载完毕b.js
```

- **ESM**
  - **main.js**

```
console.log("加载main.js");
import {count,msg,add,updateMsg} from './a.js';
import './b.js';

add('main');

console.log("index文件中的count:",count); //1
console.log("index文件中的msg:",msg); //初始化
console.log("加载完毕main.js"); //加载完毕main.js
```

  - **a.js**

```
console.log('加载a.js');

export let count = 0;

export let msg = '初始化'

export const add = (who) => {
    count++;
    console.log(`${who}调用了我修改count`)
};
export const updateMsg = () => msg = "修改了msg"

console.log('a.js加载完毕')// a.js加载完毕
```

- **b.js**

```
console.log('加载b.js');
import {count,msg,add,updateMsg} from './a.js';

console.log("b文件中的count:",count); //0
console.log("b文件中的msg:",msg); //初始化
console.log("加载完毕b.js"); //加载完毕b.js
```

- **结果对比图**

CommonJS

```
加载main.js
加载a.js
a.js加载完毕
加载b.js
b文件中的count: 0
b文件中的msg: 初始化
加载完毕b.js
main调用了我修改count
index文件中的count: 0
index文件中的msg: 初始化
加载完毕main.js
```

ESM

```
加载a.js
a.js加载完毕
加载b.js
b文件中的count: 0
b文件中的msg: 初始化
加载完毕b.js
加载main.js
main调用了我修改count
index文件中的count: 1
index文件中的msg: 初始化
加载完毕main.js
```

6. 核心模块

- **全局变量**
  - **process**

> **process.argv是命令行参数数组，第一个元素是 node，第二个元素是脚本文件名，从第三个元素开始每个元素是一个运行参数。**

```
/*
   progress.argv 返回一个数组
   ['node的安装路径',"当前运行文件的绝对路径","参数1","参数2","参数3" ... ]
*/

console.log("process.argv:", process.argv); // 返回一个数组
console.log("获取参数:",process.argv.slice(2)); //参数数组
console.log("获取参数:",process.argv[2] ? process.argv : "没有参数"); //参数数组

// 命令行输入 node process_argv.js a=1 b=2;
const args_arr = process.argv.slice(2);
const args_obj = {};
args_arr.forEach(item => {
    const arr = item.split("=");
    args_obj[arr[0]] = arr[1]; // 参数名称为键，参数值为值
});
console.log("args_obj:", args_obj); //{a:'1' , b:'2'}
console.log(args_obj.a + args_obj.b); //12 字符串连接
```

> **process_stdout 就是console.log 的底层实现**

```
console.log("hello world");
process.stdout.write("hello world\n");
```

> **process.stdin 标准输入流,监控用户在命令行中输入的内容**
>
> **初始时它是被暂停的，要想从标准输入读取数据，你必须恢复流，并手动编写流的事件响应函数。**

```
//可以理解为不马上结束程序,而是等待用户输入内容,然后结束程序
process.stdin.resume(); //恢复输入流

process.stdin.on("data",(msg)⇒{
        // console.log(msg.toString());
    process.stdout.write(`获取标准输入流信息:${msg.toString()}`)
});

```

```
console.log()
console.error() // 打印错误信息
console.trace() //错误信息追踪(准确定位哪行出现错误)
```

- **文件系统 fs**

> 读取文件内容的函数:异步的 **fs.readFile()**
>
> 同步的 **fs.readFileSync()**

  - **fs.readfile**

```
fs.readFile(filename,[encoding],[callback(err,data)])
          必写:文件名 可选:字符编码 回调函数 : 用于接受文件的内容
```

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

//不写字符编码，它会以Buffer形式表示二进制数据
fs.readFile("./console.js", (err, data) ⇒ {
        if (err) {
                console.log(err);
        } else {
                console.log(data);
        }
});


fs.readFile("./console.txt","utf-8",(err,data)⇒{
        if(err) return

        console.log(data);
})
```

- **fs.readFileSync**

> **fs.readFileSync(filename, [encoding])**
>
> 读取到的文件内容会以函数返回值的形式返回
>
> 如果有错误发生，fs 将会抛出异常，你需要使用 **try** 和 **catch** 捕捉并处理异常。

```
import fs,{readFileSync} from 'fs';
let res1;
try{
            res1 = readFileSync('./a.txt','utf-8')
}catch(e){
    // console.log(1000,e); // 捕捉异常信息
}
console.log("a.txt",res1);


const res2 = fs.readFileSync('./b.txt','utf-8');
console.log(res2);


```

- **node_modules**

```
//创建一个文件index.js
require('m1.js') //引入m1.js文件
//输出：我是一个简单的模块
// m1.js
console.log("我是一个简单的模块");


 require('m1'); 引入一个 m1 的模块
```

> **首先它会找所在目录里面有没有node_modules 文件夹，有的话，会在里面找到 m1 文件夹，在m1文件夹里面找 index.js 执行文件 。**
>
> **如果所在目录里面没有找到，它会往上一层寻找，一层一层的寻找，会一直找到所在磁盘根目录，还是没有，它不会放弃，最后去node安装路径里面寻找，如果找到，会执行刚才操作。**

- **writeFile**

```
fs.writeFile(filename,data,[第三个参数 可选],[callback(err)])
```

```
// fs.writeFile 异步方法
// 如果文件不存在会帮我们自动创建在写入
// fs.writeFile 会帮我们创建不存在的文件，不会创建不存在的文件夹

const fs = require('fs');
//data 写入的数据
const str = `
  你好,暖心
     很高兴见到你!
`

//第三个参数 : config string / object
/* {
      encoding : "utf-8",
      mode:"权限",
      flag:
           "w" -write -覆盖写入 默认值
           "a" -append -追加
      signal : 忽略
   }
*/
fs.writeFile('./target1.txt',str,e⇒{
    return e ? console.trace("写入错误") : console.log("写入成功");
}) 默认参数 flag: 'w'    每次写入都会覆盖之前的

fs.writeFile("./target2.txt",str,{flag:'a'},e⇒{
    if(e){
        console.trace("写入错误",e);
    }
    console.log("写入成功");
})    // 设置了第三个参数 flag: 'a' 追加写入，不覆盖
```

- **writeFileSync**

```
import fs from "fs"; //ESM 写法

const str = "你好,writeFileSync 同步方法\n\r";

try{
    fs.writeFileSync('./t.txt',str,{flag:'a'})
    console.log("写入成功");
}catch (e) {
    console.log("写入错误",e);
}

```

- **open，read，close**

```
fs.open(path, flags, [mode], [callback(err, fd)])
        路径          权限
fs.read(fd, buffer, offset, length, position,
[callback(err, bytesRead, buffer)])
```

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

exports.myOpenRead = (filename) =>{
    fs.open(filename,"r",666,(err,fd)=>{
        if(err){
            console.log("打开失败",err);
            return
        }
        // console.log("fd:",fd);
        const buffer = Buffer.alloc(300); //定义多少个字节
        const offset = 0; // 申请空间的开始位置

        const len = buffer.length; //读取的长度
        const pos = 0; //从哪开始读

        fs.read(fd,buffer,offset,len,pos,
(e,bytesLen,buf)=>{
            if(e){
                console.log("读取失败",e);
                return
            }
            console.log("bytesLen 真实读取的字节
数:",bytesLen);
```

```
            console.log("buf 真实读取的字节:",buf.toString());

console.log("buffer:",buffer.slice(offset,b).toString());
            console.log("buffer:",buffer.toString());
            fs.close(fd);
        })
    })
}
```

同步写法

```
import{openSync,readSync,closeSync} from "fs";

try {
    const fd = openSync("./package.json","r")
    console.log("打开成功",fd);

    const buffer = Buffer.alloc(245);
    const offset = 0;
    const len = buffer.length;
    const pos = 0;
    const byteslen = readSync(fd,buffer,offset,len,pos);
    console.log("真实读取的字节数",byteslen);
    console.log("读取的内容",buffer.slice(offset,byteslen).toString());
    closeSync(fd);

    /*try{
        const byteslen = readSync(fd,buffer,offset,len,pos);
        console.log("真实读取的字节数",byteslen);
        console.log("读取的内容",buffer.slice(offset,byteslen).toString());
        closeSync(fd);
    }catch (e) {
        console.log("读取错误",e);
    }*/


 // 优化一下出现错误的方式，避免里面再一次套用 try catch
}catch (e) {
    switch (e.code) {
```

```
        case "ERR_OUT_OF_RANGE":
            console.trace("你尝试读取的字节数大于申请缓存的字节")
            break;
        case "ENOENT":
            console.trace("没有找到对应的入口文件或者目录")
            break;
        default:
            console.log("错误",e);
            break;
    }
}
```

7. 回调与事件

- **回调 CallBack**

```
function sum(a,b,callback) {
    const res = callback(a,b)
    console.log(res);
}

sum(1,2,(p1,p2) => {
    console.log(p1,p2);
    return p1+p2
})
```

  - **老版本写法**

```
function sum(a,b,success,fail) {
    if(typeof a !== "number" || typeof b !== "number"){
        fail("类型不对");
        return
    }
    success(a+b);
}

const ok = msg => console.log(msg);
const err = msg => console.log(msg);

sum(1,4,ok,err);

```

```
sum(3,5,function (msg) {
    console.log(msg);
},function (err) {
    console.log(err);
})
```

- **CPS**

> **a : callback 作为最后一个参数传入**
>
> **b: (err,res) => {} callback 参数顺序: 第一个参数是错误信息**
>
> **第二个是结果信息(可选)**

```
function sumCps(a,b,callback) {
    let res;
    setTimeout(()⇒{
      if(typeof a ≠ "number" || typeof b ≠ "number"){
            return callback(new Error("参数类型错误"));
      }
    },500)
    res  = a+b;
    callback(null,res); //(err,data)
}

sumCps(2,3,4,(e,res)⇒{
    if(e){
        console.log("有错误",e);
    }
    console.log(res);
})

////要求:实现多个数相加(必须2个数以上),传入的最后一个参数必须是函数,实现加法功能;

function sumCps() {
    if(arguments.length < 3){
        throw new Error("你输入的参数过少")
    }
    const callback = arguments[arguments.length-1];
    if(typeof callback ≠= "function"){
        throw new  Error("callback1 is not a function");
    }
    const items = Array.from(arguments).slice(0,arguments.length-1);
    for(const item of items){
```

```
        if(typeof item !=="number"){
            return callback(new Error("参数类型错误"));
        }
    }
    let res;
    setTimeout(() =>{
         res = items.reduce((acc,v)=> acc+v,0);
         callback(null,res);
    },500)
}

sumCps(2,3,-4,-5,4,-3,1,(e,res)=>{
    if(e){
        console.log("有错误",e);
        return
    }
        console.log(res);
})
console.log("end");
```

- **并非所有的回调都是CPS**

> **有些函数虽然可以通过参数接受回调，但这并不意味这函数一定是异步函数，也不意味着它必定是采用CPS编写的**

```
// Array对象的map（）方法
const result = [1,5,7].map(item => item -1)
console.log(result) // [0,4,6]
```

- **观察者模式**

> **Observer模式**定义了**一个对象**（这叫做主题，subject），它会在状态改变的时候通知一组
> 观察者（或者说监听者）。 Observer模式与Callback模式之间的**主要区别**在于，它可以通
> 知多个监听器（也就是观察者），而采用CPS（接续传递风格）所实现的普通Callback模
> 式，通常只会把执行结果传给一个监听器，也就是用户在提交执行请求时传入的那个回调。

- **普通范式**

```
function Observer(name) {
    this.name = name;
    this.showMsg = function (data) {
        console.log(`观察者:${this.name} 收到了天软发布返校最新消息: ${data}`);
    }
}

function Subject() {
    this.observers = [];
}

Subject.prototype.add = function (observer) {
 this.observers.push(observer);
}

Subject.prototype.emit = function (data) {
    this.observers.forEach(observer => observer.showMsg(data));
}

const observer1 = new Observer('小明');
const observer2 = new Observer('小红');
const observer3 = new Observer('小胖');

const subject = new Subject()
subject.add(observer1);
subject.add(observer2);
subject.add(observer3);

subject.emit('马上就要返回学校了');
```

- **回调函数普通模式**

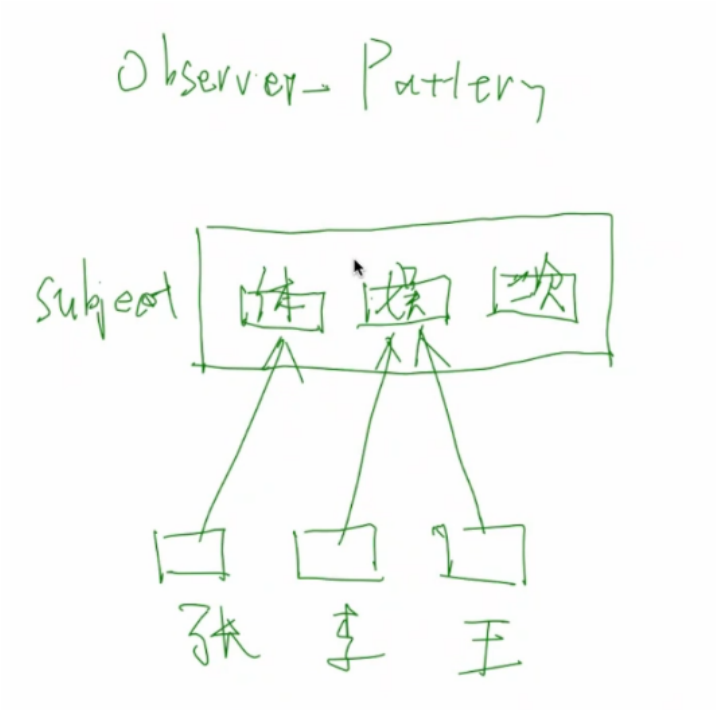
```
function Subject() {
    this.observers = [];
}

Subject.prototype.add = function (observer) {
 this.observers.push(observer);
}

Subject.prototype.emit = function (data) {
    this.observers.forEach(callback => callback(data));
}
```

```
12
13 const subject = new Subject()
14 subject.add(data ⇒ console.log(`小红收到了天软最新发布的返
   校信息 ${data}`))
15 subject.add(data ⇒ console.log(`小明收到了天软最新发布的返
   校信息 ${data}`))
16 subject.add(data ⇒ console.log(`小胖收到了天软最新发布的返
   校信息 ${data}`))
17
18 subject.emit('马上就要返回学校了');
```

- **手写obs**




```
1  function myTv() {
2      const cache = {};
3      const on = (name,callback) ⇒{
4          if(!Array.isArray(cache[name])){
5              cache[name] = [];
6          }
7          cache[name].push(callback);
8      }
9
10     const emit = (name,data) ⇒{
11         if(!Array.isArray(cache[name])) return
12
13         cache[name].forEach(observer ⇒ observer(data));
14     }
15
16     return {
```

```
        on,
        emit
    }
}

const tv = myTv();

tv.on("sports", msg => console.log(`老张：${msg}`));
tv.on("sports", msg => console.log(`老王：${msg}`));
tv.on("star_news", msg => console.log(`老李：${msg}`));

tv.emit("sports", "足球");
tv.emit("star_news", "迪丽热巴");
```

- **EventEmitter**

> **on (event,listener):这个方法可以为某种事件注册一个新的监听器。（事件用字符串表示，监听器用函数表示。）**

> **once (event,listener):这个方法也能注册监听器，但是触发完一次事件后，这个监听器就会遭到移除。**

> **emit (event,args...):这个方法用来触发新事件，并且能够传一些参数给监听器。**

> **removeListener** (event, **listener**):这个方法用来移除某种事件的监听器。
>
> **listener 当你注册事件是匿名的函数，它是不可以移除的，必须赋给一个变量，再传入，即可移除**

- **创建并使用EventEmitter**

```
const {EventEmitter} = require('events');
const fs = require('fs');

function findRegex(filename,regex) {
    const emitter = new EventEmitter();
    fs.readFile(filename,"utf-8",(err,data)=>{
        if(err){
            return emitter.emit('error',err);
        }
```

```
        emitter.emit('startMatch',filename);
    const matchRes = data.match(regex);
        if(matchRes){
            matchRes.forEach(item =>
emitter.emit("match",filename,item));
        }
    });
    return emitter;
}

const emitterInstance = findRegex('../2022-4-
19/callback.js',/c\w+/g);

emitterInstance.on("startMatch",file =>console.log(`Start
match ${file}`));

emitterInstance.on("match",(file,content)
=>console.log(`Match ${content} at ${file}`));

emitterInstance.on("error",err =>console.log("发生错
误",err));
```

8 HTTP 服务器

- **HTTP 服务器的基础知识**

```
import http from 'http';

const server = http.createServer(((req, resp) => {
/*  resp.write('Hello World\n');
    resp.write('Hello Node.js');*/
    resp.end("Hello World\nhello Node.js");
}))

server.listen(3000,()=> console.log('server is running
at port 3000'));
server.on("error",(err)=> console.log("捕获错误",err));
```

```
// Esm 写法
import {createServer} from 'http';
createServer((req,resp) =>{
    const body = `<h1>hello http server</h1>`
```

```
    resp.statusCode = 200; // 状态码
    //setHeader statusCode 在write和end之前调用
    resp.setHeader("Content-Type","text/html;charset=utf-8");
    resp.write("hello nodejs");
    resp.end(body);
}).listen(3000,()⇒console.log("你的服务已经启动,正在监听3000端口号"))
.on('error',err ⇒ console.log("你的服务启动失败，请检查端口号是否被占用",err));


//statusCode
 const body = `<h1>hello http server!</h1>`
     resp.statusCode = 302;
     resp.setHeader("Location","http://www.baidu.com");
     resp.setHeader("Content-Type","text/html;charset=utf-8");
     resp.end(body);

//length
// resp.setHeader("transfer-encoding","chunked"); // 默认 http 1.1

// resp.setHeader("content-length", body.length); // 字符长度 7
 resp.setHeader("content-length",8) //  字符长度 2
// resp.setHeader("content-length", Buffer.byteLength(body)); //  字节长度  11(一个中文3个字节)
```

- **post**

```
// 命令: curl -d "xxx" http://localhost:3000/
req.on("data",(chunk) ⇒{
        count++;
        console.log(`chunk: ${count}`,chunk);
    });

    req.on("end",() ⇒{
        console.log("读取请求内容结束");
        resp.end();
    });
})
```

- **URL**

```
import {URL} from "url";

const u = new URL("http://localhost:3000/2?Devil=暖心");
console.log(u); // URL {protocol: 'http:', slashes: true, ...}
console.log("pathname:",u.pathname); // /2
console.log("pathname 需要的内容:",u.pathname.slice(1)); // 2 ⇒ 字符串
console.log("pathname 需要的内容转型:",parseInt(u.pathname.slice(1))); // 2 ⇒ 数字

console.log("Devil :",u.searchParams.get("Devil")); // searchParams 属于 map 类型 需要get获取
```

- **构建 RESTful Web 服务**

```
//ESM 写法
import http from "http";
import {URL} from "url";

const blogs = [];

http.createServer((req,resp)⇒{
    resp.setHeader("Content-Type","text/plain;charset=utf-8");
    switch(req.method){
       case:"GET":
          blogs.forEach(item ⇒ resp.write(item+"\n"));
            resp.end("查询完毕");
            break;
       case "POST":
            let content ="";
            // req.setEncoding("utf-8");
            req.setEncoding("utf-8");
            req.on("data",(chunk)⇒{
                console.log("chunk",chunk);
                content += chunk;
            })
            req.on("end",()⇒{
                blogs.push(content);
                resp.end("创建成功");
            })
            break;
```

```
27
28              // curl 删除命令 curl -X DELETE
    http://localhost:3000/blogs/1
29          case "DELETE":
30              const u = new
    URL(req.url,"http://localhost");
31              const pathname = u.pathname;
32              const id = pathname.slice(1);
33              const index = parseInt(id,10)-1;
34              if(isNaN(index)){
35                  resp.statusCode = 400;
36                  resp.end("非法的参数");
37              }else if(blogs.length ⩽ index){
38                  resp.statusCode = 404;
39                  resp.end("删除的博文找不到")
40              }else{
41                  blogs.splice(index,1);
42                  resp.end("删除成功")
43              }
44              break;
45          default:
46              resp.end("其他的请求");
47              break;
48      }
49  }).listen(3000,()⇒cosnole.log("blogs server is
    running"))
50  .on("error",err⇒console.log("启动服务错误”, err);
```

- **提供静态文件服务**

```
CommonJS 语法
    __dirname 目录名称

    -在commonJs中,__dirname是一个全局变量,它指向当前执行脚本所在的目录
    -可以得到当前执行脚本的绝对路径

    path.resolve(_,_,_, ... );

    -resolve方法可以将相对路径转换成绝对路径
    -如果某一个是绝对路径,那么这个参数前面的目录名称就会被忽略

    - join.resolve(_,_,_, ... );
    - join方法会把参数拼接成一个路径
```

```
console.log("dirname:", __dirname);

const fs = require("fs");
const {resolve,join} = require("path");

// const newPath = resolve(__dirname,"./1.txt"); // 成功解析路径
// const newPath = resolve(__dirname,"1.txt");   // 成功解析路径
// const newPath = resolve(__dirname,"/1","2","3","1.txt"); // !注意是绝对路径

// const newPath = join(__dirname,"./1.txt");
// const newPath = join(__dirname,"/1.txt");
const newPath = join(__dirname,"1.txt");

fs.readFile(newPath,"utf-8",(err,data)⇒{
        if(err){
                console.log("读取文件失败");
                return;
        }
        console.log("读取文件成功");
        console.log(data);
});
```

- **详细对比 `resolve` 和 `join` 区别**

```
const {resolve, join} = require('path');

const resolve_path = resolve("./test-path",'1.txt');
console.log("resolve_path:",resolve_path);
//resolve_path: D:\webstormCode\node.js\2022-4-29\test-path\1.txt
const join_path = join("./test-path",'1.txt');
console.log("join_path:",join_path); //join_path: test-path\1.txt

/*const resolve_path = resolve("./a","b",'1.txt');
console.log("resolve_path:",resolve_path); //resolve_path: D:\webstormCode\node.js\2022-4-29\a\b\1.txt
console.log("join_path:",join_path);// join_path: a\b\1.txt
*/
```

- **ESM写法**

```
import fs from 'fs';
import {join, dirname} from 'path';
import {fileURLToPath} from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

const newPath = join(__dirname,'/package.json');
fs.readFile(newPath,"utf8",(err,data)=>{
    if(err) return console.log(err);
    console.log(data);
})
```

- **简单静态服务事例**

```
import http from 'http';
import fs from 'fs';
import {URL,fileURLToPath} from "url";
import {dirname,join} from "path";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

http.createServer((req,resp)=> {
    const u = new URL(req.url, "http://localhost:3000");
```

```
11      resp.setHeader("Content-Type",
    "text/html;charset=utf-8");
12      if (u.pathname === "/" || u.pathname ===
    "/favicon.ico"){
13          return resp.end();
14      }
15      //方法一
16      /*try{
17          const data =
    fs.readFileSync(join(__dirname,"public",u.pathname));
18          resp.end(data);
19      }catch (e) {
20          resp.statusCode = 500;
21          resp.end("失败");
22      }*/
23      //方法二
24      /*const stream = fs.createReadStream(join(__dirname,
    "public", u.pathname));
25      stream.on("data", (chunk) => {
26          try {
27              // throw new Error("抛出错误");
28              resp.write(chunk);
29          } catch (error) {
30              console.log("捕获到错误");
31              resp.end("页面发生错误");
32              return;
33          }
34      }).on("end", () => {
35          resp.end();
36      }).on("error", (err) => {
37          resp.end("发生错误");
38      });*/
39
40      //方法三
41      const stream = fs.createReadStream(join(__dirname,
    "public", u.pathname));
42      stream.pipe(resp);
43  }).listen(3000,()=> console.log("static server is
    running"))
44  .on("error",(e)=> console.log(e));
```

- **从表单中接受用户输入**

- **用formidable处理上传的文件**
- **Express**