Node.js

1、用 node.js 构建最简单的服务器端

通过 *node* 打开以上 *http.js* 文件，即可在 *cmd* 或者 *vscode* 的终端中使用命令行启动。

2、nodemon

当服务端文件更新保存后该工具会自动重启服务端，而不需要手动重启。

（*nodejs* 会将硬盘中的程序文件加载到内存中，而改变了硬盘上的程序文件不会改变内存中已有的之前的程序文件，因此需要重启服务才会重新加载硬盘中的新的程序文件。）

3、模块化

若多个 *js* 文件中不暴露/导出变量或其他，则不会造成变量污染，即无法在 *b.js* 中不声明而直接使用 *a.js* 中的变量 *a*。若要在 *b.js* 中使用 *a.js* 中的变量 *a*，则需在 *a.js* 中 *exports a*，如下图所示。

```
module.exports = {
    a
}
```

前端模块化：*AMD => sea.js*　　*CMD => require.js*

node.js 自带模块化，符合 *CommonJS* 规范。

http://javascript.ruanyifeng.com/nodejs/module.html

4、module.exports 和 exports

```
// module.exports = {
//     a,
//     Person
// }
exports.a = a;
exports.Person = Person;
// exports 是 module.exports 的引用;
// module.exports = exports;
```

但 exports = {};这种形式不会改变 module.exports，因为此时相当于新赋值了一个对象。

批注 [M帐1]:

5、引用文件夹

引用文件或 node_modules 外的文件夹，则需：./文件名.js 或 ./文件夹名。

引用 node_modules 里的文件夹，则只需：文件夹名 (加上./会报错) 。


```
let Ma = require("./Ma");

let {a,b} = require("mytest");
```

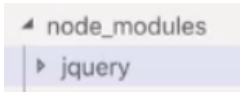
6、npm 包管理器

可通过 npm 下载第三方模块（他人编写的）。

① npm install 包名 / npm i 包名

```
npm install jquery
```

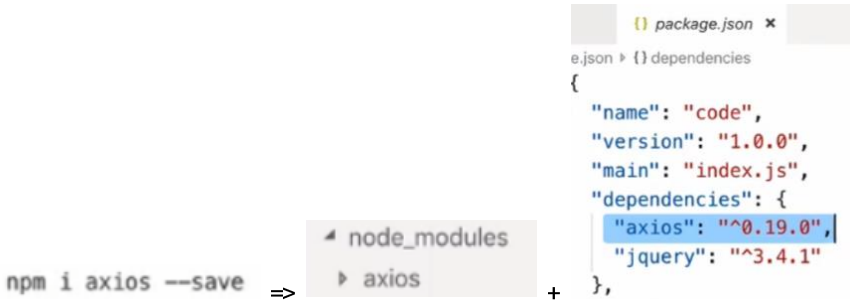
=>



② *npm init*

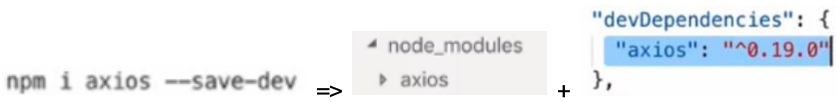
可创建 *package.json*。

③ *npm i* 包名 *—save / npm i* 包名 *-S*



安装引入 *axios*（在 *node_modules* 文件夹下添加 *axios* 文件夹及其内容）的同时，在 *package.json* 的 *dependencies*（运行依赖：开发过程及编译打包后项目上线正式运行中均需要）中添加 *axios* 及其版本号。

④ *npm i* 包名 *—save-dev / npm i* 包名 *-D*



安装引入 *axios*（在 *node_modules* 文件夹下添加 *axios* 文件夹及其内容）的同时，在 *package.json* 的 *devDependencies*（开发依赖：开发过程需要，但编译打包后项目上线正式运行中不需要）中添加 *axios* 及其版本号。

※ 现阶段不加*—save* 也会在 *package.json* 的 *dependencies*（运行依赖）中添加包名及其版本号。*uninstall* 该包时 *package.json* 的 *dependencies*（运行依赖）中也会相应删除包及其版本号。

⑤ *npm install / npm i*

根据 *package.json* 中的各种依赖安装所需的包。默认为局部安装。

若需全局安装则须在以上命令行的最后加上 *-g*，此时会将包安装于↓⑥中的路径。

⑥ *npm i* 包名@版本号

指定安装某个版本的包。

⑦ *npm root -g*

终端打印出根目录下的 *node_modules* 文件夹路径。

※ *require* 引入模块时会自动向上寻找 *node_modules* 文件夹中是否有该模块，即该 *require* 所在文件的所在文件夹下是否有 *node_modules* 及相应模块 => 该 *require* 所在文件的所在文件夹的上一级文件夹下是否有 *node_modules* 及相应模块 => ..... => 直到根目录下的 *node_modules* 文件夹中是否有该模块。

⑧ *npm update* 包名

7、*npm* 内置模块

```
nodejs内置模块有：Buffer，C/C++Addons，Child Processes，Cluster，Console，Cr
ypto，Debugger，DNS，Domain，Errors，Events，File System，
Globals，HTTP，HTTPS，Modules，Net，OS，Path，Process，P unycode，Query Strings，Readline，
REPL，Stream，String De coder，Timers，TLS/SSL，TTY，UDP/Datagram，URL，Utilities，V8，VM，
ZLIB；内置模块不需要安装，外置模块需要安装；
```

8、*fs* - 文件操作

文件操作：增删改查

① *fs.writeFile* 写入文件

```
// a:追加写入; w 写入; r: 读取;
fs.writeFile("1.txt","我是追加的文字",{flag:"a"},function(err){
    if(err){
        return console.log(err);
    }
    console.log("写入成功");
})
```

第三个参数 *flag* 为配置项，默认为 *w*，即写入（如原文件 *1.txt* 中的原文字是“*123*”，写入的文字是“*456*”，则最后写入后 *1.txt* 中的文字会变为“*456*”；也会在不存在 *1.txt* 文件的情况下创建 *1.txt* 并写入“*456*”）。

若为 *a* 则表明追加写入（如原文件 *1.txt* 中的原文字是“*123*”，追加写入的文字是“*456*”，则最后追加写入后 *1.txt* 中的文字会变为“*123456*”）。

② *fs.readFile* 读取文件

```
// 文件读取
// fs.readFile("1.txt","utf8",(err,data)=>{
//     if(err){
//         return console.log(err);
//     }
//     console.log(data);
// })
```

第二个参数表示按何种格式（上图为 *utf8*）读取，若不设置则会默认以二进制的格式读取并以 *buffer*（两位的十六进制）显示，此时若想以正常文本格式显示需将数据转换成 *string*，即 *data.toString()*。

```
fs.readFile("1.txt",(err,data)=>{
        if(err){
            return console.log(err);
        }
        console.log(data.toString());
    })
```

③ *fs.writeFileSync* / *fs.readFileSync* 为同步，没有加 *Sync*（前述①、②）则是异步。

④ *fs.rename* 重命名文件

```
fs.rename("1.txt","2.txt",err=>{
    if(err){
        return console.log(err);
    }
    console.log("修改成功");
});
```

⑤ *fs.unlink* 删除文件

```
fs.unlink("2.txt",(err)=>{
    if(err){
        return console.log(err);
    }
    console.log("删除成功");
})
```

⑥ *fs.copyFile* 复制文件

```
fs.copyFile("index.html","myindex.html",err=>{
    if(err){
        return console.log(err);
    }
    console.log("复制成功! ")
})
```

本质上复制为读取原文件内容并将该内容写入新文件的过程。

```
// 复制
function mycopy(src,dest){
   fs.writeFileSync(dest,fs.readFileSync(src));
}
mycopy("index.html","test.html");
```

9、*fs* - 目录操作

① *fs.mkdir* 创建目录

```
// 创建目录
fs.mkdir("11",err=>{
    if(err){
        return console.log(err);
    }
    console.log("创建成功");
})
```

② *fs.rename* 修改/重命名目录

```
// 修改目录名称
fs.rename("11", "22", err => {
    if (err) {
        return console.log(err);
    }
    console.log("修改成功");
})
```

③ *fs.readdir* 读取目录

```
// 读取目录;
fs.readdir("22",(err,data)=>{
    if(err){
        return console.log(err);
    }
    console.log(data);
})
```

第一个参数为路径（相对路径或绝对路径）。

读取到的结果存放于一个数组中，可以读取文件（有后缀），也可以读取目录。

```
yuweihaideMacBook-Pro:code yuweihai$ node filesystem.js
[ '1.txt', '2.html' ]
yuweihaideMacBook-Pro:code yuweihai$ node filesystem.js
[ '1.txt', '2.html', '33' ]
```

④ *fs.rmdir* 删除空目录（目录下不为空则会报错，无法删除）

```
// 删除目录(空文件夹/目录)
fs.rmdir("22",err=>{
    if(err){
        return console.log(err);
    }
    console.log("删除成功");
})
```

⑤ *fs.exists* 判断文件或目录是否存在，返回值为 *true*（存在）或 *false*（不存在）

```
// 判断文件或者目录是否存在
fs.exists("22",exists=>{
    console.log(exists);
})
```

⑥ *fs.stat* 获取文件或目录的详细信息

```
// 获取文件或者目录的详细信息;
fs.stat("index.html",(err,stat)=>{
    if(err){
        return console.log(err);
    }
    console.log(stat);
})
```

stat.isFile() 可用于判断是否是文件：

```
// 判断文件是否是文件
let res = stat.isFile();
console.log(res);
```

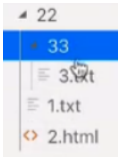
返回值为 *true*（是）或 *false*（不是）

stat.isDirectory() 可用于判断是否是目录：

```
// 是否是一个文件夹;
let res = stat.isDirectory();
console.log(res);
```

返回值为 *true*（是）或 *false*（不是）

⑦ 删除非空目录：遍历删除目录里的文件再删除空目录。




```
// 删除非空文件夹;
// 先把目录里的文件删除-->删除空目录;
// 22
function removeDir(path){
    let data = fs.readdirSync(path);
     // ["33","1.txt","2.html"];
    for(let i=0;i<data.length;i++){
        // 是文件或者是目录; --->?文件 直接删除？目录继续查找;
        let url = path + "/" + data[i];
        let stat =  fs.statSync(url);
        if(stat.isDirectory()){
            //目录 继续查找;
            removeDir(url);
        }else{
            // 文件 删除
            fs.unlinkSync(url);
        }
    }
     //  删除空目录
    fs.rmdirSync(path);
}
removeDir("22");
```

10、buffer

① 创建

创建 Buffer 类

Buffer 提供了以下 API 来创建 Buffer 类：

- **Buffer.alloc(size[, fill[, encoding]])：** 返回一个指定大小的 Buffer 实例，如果没有设置 fill，则默认填满 0
- **Buffer.allocUnsafe(size)：** 返回一个指定大小的 Buffer 实例，但是它不会被初始化，所以它可能包含敏感的数据
- **Buffer.allocUnsafeSlow(size)**
- **Buffer.from(array)：** 返回一个被 array 的值初始化的新的 Buffer 实例（传入的 array 的元素只能是数字，不然就会自动被 0 覆盖）
- **Buffer.from(arrayBuffer[, byteOffset[, length]])：** 返回一个新建的与给定的 ArrayBuffer 共享同一内存的 Buffer。
- **Buffer.from(buffer)：** 复制传入的 Buffer 实例的数据，并返回一个新的 Buffer 实例
- **Buffer.from(string[, encoding])：** 返回一个被 string 的值初始化的新的 Buffer 实例

```
// 创建一个长度为 10、且用 0 填充的 Buffer。
const buf1 = Buffer.alloc(10);

// 创建一个长度为 10、且用 0x1 填充的 Buffer。
const buf2 = Buffer.alloc(10, 1);

// 创建一个长度为 10、且未初始化的 Buffer。
// 这个方法比调用 Buffer.alloc() 更快，
// 但返回的 Buffer 实例可能包含旧数据，
// 因此需要使用 fill() 或 write() 重写。
const buf3 = Buffer.allocUnsafe(10);

// 创建一个包含 [0x1, 0x2, 0x3] 的 Buffer。
const buf4 = Buffer.from([1, 2, 3]);

// 创建一个包含 UTF-8 字节 [0x74, 0xc3, 0xa9, 0x73, 0x74] 的 Buffer。
const buf5 = Buffer.from('tést');

// 创建一个包含 Latin-1 字节 [0x74, 0xe9, 0x73, 0x74] 的 Buffer。
const buf6 = Buffer.from('tést', 'latin1');
```

② 连接

```
// let buffer = Buffer.from("大家好");
// console.log(buffer);
// let buffer = Buffer.from([0xe5,0xa4,0xa7,0xe5,0xae,0xb6,0xe5,0xa5,0xbd]);
// console.log(buffer.toString());

let buffer1 = Buffer.from([0xe5,0xa4,0xa7,0xe5]);
let buffer2 = Buffer.from([0xae,0xb6,0xe5,0xa5,0xbd]);
// console.log(buffer1.toString());
let newbuffer = Buffer.concat([buffer1,buffer2]);
console.log(newbuffer.toString());
```

```
yuweihaideMacBook-Pro:code yuweihai$ node buffer.js
<Buffer e5 a4 a7 e5 ae b6 e5 a5 bd>
yuweihaideMacBook-Pro:code yuweihai$ node buffer.js
大家好
yuweihaideMacBook-Pro:code yuweihai$ node buffer.js
<Buffer e5 a4 a7 e5>
yuweihaideMacBook-Pro:code yuweihai$ node buffer.js
大
yuweihaideMacBook-Pro:code yuweihai$ node buffer.js
大家好
```

StringDecoder 也能实现上述效果，且性能更好。

http://nodejs.cn/api/string_decoder.html

```
let { StringDecoder } = require("string_decoder");
let decoder =  new StringDecoder();
let res1 = decoder.write(buffer1);
let res2 = decoder.write(buffer2);
console.log(res1+res2);
```

```
yuweihaideMacBook-Pro:code yuweihai$ node buffer.js
大家好
```

11、stream 流

① *stream*

```
let rs = fs.createReadStream("1.txt");
rs.on("data",chunk=>{
    console.log(chunk.toString());
})
```

```
流会把数据分成64kb的小文件传输;
```

```
// 流完成了;
rs.on("end",()=>{
    console.log(str);
})
```

② *pipe*

相当于复制，下图代码将读取的 *1.txt* 的内容写入新创建的 *2.txt* 中。

```
let rs = fs.createReadStream("1.txt");
let ws = fs.createWriteStream("2.txt");
rs.pipe(ws);
```

12、node.js 与 javascript 的关系

javascript = js 标准 + *WebAPI*

nodejs = js 标准 + 系统 *API*

13、node.js 的工作

可用来 *CLI* 编程，即可以写基于命令行的工具，即没有界面，如 *Vue-cli*、*creat-react-app*

GUI 编程，即图形化编程、桌面端编程，即有界面，如 *Electron*、*nw.js*

14、node.js 的 http 内置模块

```
server.on('request', (req, res) => {
    // 与请求的客户端信息有关的数据和方法通过 req 对象提供
    // console.log('req', req);
    // 与服务端信息有关的数据和方法通过 req 对象提供
    // console.log('res', res);
```

参见 *nodejs-01* 的课件。