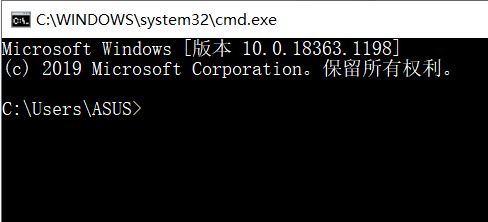
# Node

1. 终端基本使用

这里的终端指的是PC的cmd界面，win+r输入cmd



在cmd中输入这些命令

1、打开应用

notepad 打开记事本

mspaint 打开画图

calc 打开计算机

write 写字板

sysdm.cpl 打开环境变量设置窗口

2、常用命令

在windowv不要乱测试，怕出问题，可以在linux虚拟机上随便搞

md 创建目录

rmdir(rd) 删除目录，目录内没有文档。

echo on a.txt 创建空文件

del 删除文件

rm 文件名 删除文件

cat 文件名 查看文件内容

cat > 文件名 向文件中写上内容

二、Node.js开发环境准备

1、普通安装方式官方网站

在官网下载，LTS那个，它是推荐，比较稳定；current是最新的，可能有bug

、

注：下载.msi直接安装就行，如果是zip还得自己去配置环境变量



2、多版本安装方式

（1）卸载已有的Node.js

（2）下载nvm（node version manager）node版本管理

https://github.com/coreybutler/nvm-windows，进入页面后选择release

（3）在C盘创建目录dev

在dev目中中创建两个子目录nvm和nodejs

把nvm包解压进去nvm目录中

在install.cmd文件上面右键选择【以管理员身份运行】

打开的cmd窗口直接回车会生成一个settings.txt文件，把生成的setting文件另存在nvm目录下。把root改成当前目录，用绝对路径，path改成装nodejs的路径

（4）配置nvm和Node.js系统环境变量

NVM\_HOME:C:\dev\nvm

NVM\_SYMLINK:C:\dev\nodejs（这里到时候其实是一个文件夹的快捷方式）

（5）把配置好的两个环境变量名字追加到“系统环境变量Path”中

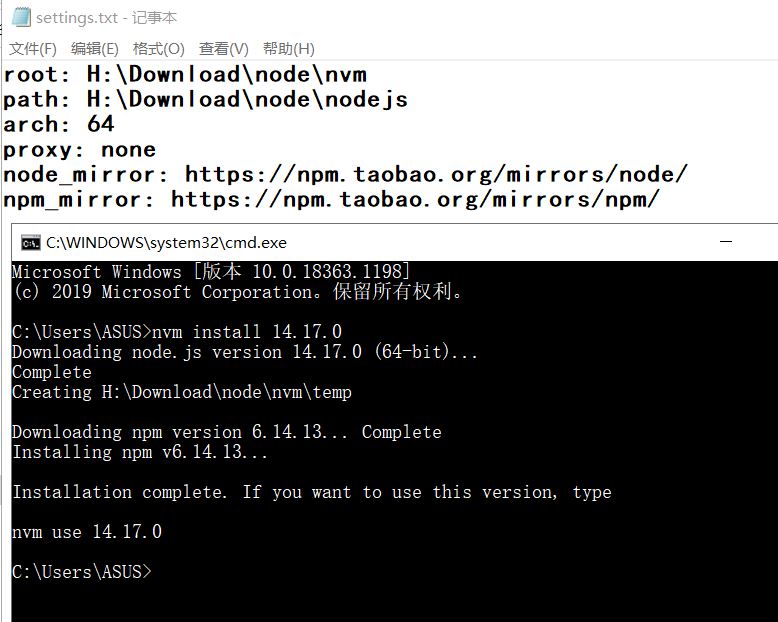
（6）nvm用的好像是国外的资源，重新配置一下镜像

nvm默认使用的是 http://nodejs.org/dist 的镜像，在国内使用是很慢的（可能会导致一些东西没有下载）。

在安装的目录下找到settings.txt文件打开后加上

node\_mirror: https://npm.taobao.org/mirrors/node/

npm\_mirror: https://npm.taobao.org/mirrors/npm/ 使用淘宝镜像，可以提高下载速度



（7）nvm常用的命令

nvm list 查看当前安装的Node.js所有版本

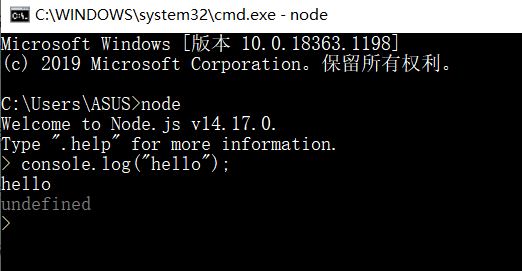
nvm install 版本号 安装指定版本的Node.js（npm会一起装上）

nvm uninstall 版本号 卸载指定版本的Node.js

nvm use 版本号 选择指定版本的Node.js

3、Node.js之HelloWorld

（1）命令行方式REPL



这个环境就叫做 REPL

read-eval-print-loop

read读取屏幕中的代码

eval执行，计算的意思

loop循环这个过程

在REPL环境中，\_表示最后一次之执行的结果，1+1=2，\_代表就是2

.exit退出REPL环境。

4、初识node模块化

node里边不存在window这个对象，有一个类似的概念，叫global（是可以省略的），node环境提供很多的模块，看文档

js文件的命名不能是node.js会出错

（1）全局成员--global的属性

**\_\_filename**：包含文件名称的全路径

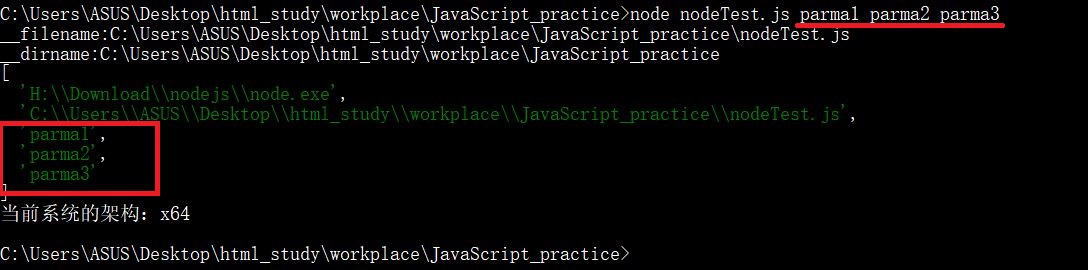
**\_\_dirname**：文件的路径，不包括文件名称

在文件操作中会用到

1

**process.argv**是一个数组，默认情况下，前两项数据分别是Node.js环境的路径；当前执行的js文件的全路径。第三个参数表示cmd命令行输入参数也能获取到。

**arch**表示当前操作系统的架构（64位还是32位），有x64、x32



（2）运行文件方式

可以直接找到js文件，并执行 node xxxx.js

5、模块成员导出（类似于window.a，暴露给windows）

它的代码都是构建在模块化的基础之上

（1）传统非模块开发有如下的缺点：

命名冲突。

文件依赖，需要先引入某个文件，再引入某个文件，才不会有错误。

前端标准的模块化规范

1、AMD对应 requirejs 国外的

2、CMD对应 seajs 阿里的

服务器的模块化规范

1、commonJS 对应 Node.js

（2）模块化相关的规则

如何定义模块：一个js文件就是一个模块，模块内部的成员（变量、函数）都是相互独立的，就是外部无法访问的意思，如果要访问，需要把它导出去。

（3）\*模块的导出（**export**）和引入(**require(js文件路劲)**)

export.sum = sum;

module.exports = sum;

global的方式不常用。

这两种导出的方式，在引入端的使用都是不一样的。

|  |
| --- |
| var add = function(a, b) {  return parseInt(a)+parseInt(b);  }  **exports.add = add;** |
| var module = require('./01.js');  var result = **module.add(10, 10);**  console.log(result); |

|  |
| --- |
| var add = function(a, b) {  return parseInt(a)+parseInt(b);  }  **module.exports = add;** |
| var module = require('./01.js');  var result = **module(10, 100);** |

|  |
| --- |
| var flag = true;  **global.flag = flag;** |
| require('./01.js');  console.log(**global.flag**, typeof global.flag);  输出true boolean |

（4）多次引入模块

已经加载的模块会缓存，为了提高性能，再次require的时候，如果缓存中有相同的内容，他就不去缓存了，它就是根据文件的路径知道是否已经加载，require的参数是一个路径

（5）模块文件的后缀情况：

.js .json .node(这种情况比较少，它是c或者c++开发的)

如果不写文件的后缀，js优先，json次之，node最后

|  |
| --- |
| var data = require('./data.json');  console.log(data) |
| 1 |

不同版本的编译的node还无法运行。

6、模块分类

（1）自定义模块

（2）系统核心模块

fs 文件操作

http 网络操作

path 路径操作

querystring 查询参数解析

url url解析

三、Node.js核心模块API

路径中的斜杆记得要进行转义。

1、文件操作案例

|  |
| --- |
| let root = 'C:**\\**Users\\ASUS\\Desktop\\html\_study\\workplace\\node';  let fs = require('fs');  let path = require('path');  let content = `  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <h3>  hello world  </h3>  </body>  </html>  `;  let initData = {  projectName: 'demo',  data: [{  name: 'css',  type: 'dir',  },{  name: 'js',  type: 'dir',  },{  name: 'images',  type: 'dir',  },{  name: 'index.html',  type: 'file',  }],  }  fs.mkdir(path.join(root,initData.projectName), (err) => {  if (err) {  return;  }  initData.data.forEach((item) => {  if (item.type === 'dir') {  // 同步创建  fs.mkdirSync(path.join(root,initData.projectName, item.name));  } else if (item.type === 'file') {  fs.writeFileSync(path.join(root,initData.projectName, item.name), content);  }  });  }); |

四、Buffer基本操作

1、定义

Buffer对象是Node处理二进制数据的一个接口。它是Node原生提供的全局对象，可以直接使用，不需要require(‘buffer’)。

Buffer本质上是字节数组，存储的数据都是二进制，8个二进制就是一个字节

它的API基本上可以分为三类

1. 构造函数（在es6中称之为“类”比较合理）
2. 静态方法（就是直接通过对象调用，例如内置对象）
3. 实例方法（需要去用构造函数创建出来的对象，去调用它的方法）

2、实例化buffer对象

（1）Buffer.alloc(size[, fill[, encoding]])

size <integer> 新 Buffer 的期望长度。

fill <string> | <Buffer> | <Uint8Array> | <integer> 用于预填充新 Buffer 的值。默认值: 0。

encoding <string> 如果 fill 是一个字符串，则这是它的字符编码。默认值: 'utf8'。

分配一个大小为 size 字节的新 Buffer。 如果 fill 为 undefined，则用**零**填充 Buffer。

|  |
| --- |
| let buff = Buffer.alloc(5);  console.log(buff);  1 |

（2）Buffer.from(string[, encoding])

string <string> 要编码的字符串。

encoding <string> string 的字符编码。默认值: 'utf8'。

创建一个包含 string 的新 Buffer。 encoding 参数指定用于将 string 转换为字节的字符编码。

|  |
| --- |
| let buff = Buffer.from('hello');  console.log(buff);  1 |

（3）Buffer.from(array)

array <integer[]>

使用 0 – 255 范围内的字节数组 array 来分配一个新的 Buffer。 超出该范围的数组条目会被截断以适合它。

|  |
| --- |
| let arr = [0x62, 0x75, 0x66, 0x66, 0x65, 0x72];  let buff = Buffer.from(arr);  console.log(buff);  console.log(buff.toString());  1 |

（4）编码（在页面搜索 hex ）

'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。 这是**默认的字符编码**。 当将 Buffer 解码为不专门包含有效 UTF-8 数据的字符串时，则会使用 Unicode 替换字符 U+FFFD � 来表示这些错误。

1. 静态方法

（1）Buffer.isEncoding() 判断是否支持该编码

（2）Buffer.isBuffer() 判断是否为Buffer

（3）Buffer.byteLength() 返回指定编码的字节长度，默认utf8

（4）Buffer.concat() 将一组Buffer对象合并为一个Buffer对象

|  |
| --- |
| console.log(Buffer.isEncoding('utf8'));  let buff = Buffer.alloc(5);  console.log(Buffer.isBuffer(buff));  console.log(Buffer.byteLength(buff));  let buff1 = Buffer.from('hello');  let buff2 = Buffer.from('hi');  let result = Buffer.concat([buff1, buff2]);  console.log(result.toString());  console.log(Buffer.byteLength(result));  1 |

4、实例方法

（1） write() 向buffer对象中写入内容

buf.write(string[, offset[, length]][, encoding])

|  |
| --- |
| let buff = Buffer.alloc(5);  buff.write('hello');  console.log(buff.toString()); |

（2）slice() 截取新的buffer对象，就是返回一个新的

buf.slice([start[, end]])

start <integer> 新 Buffer 开始的位置。默认值: 0。

end <integer> 新 Buffer 结束的位置（不包含）。默认值: buf.length。

返回: <Buffer>

返回一个新的 Buffer，它引用与原始的 Buffer 相同的内存，但是由 start 和 end 索引进行偏移和裁剪。

|  |
| --- |
| let buff = Buffer.alloc(5);  buff.write('he**ll**o');  console.log(buff.slice(2,4).toString()); |

（3）toString() 把buf对象转成字符串

（4）toJson() 把buf对象转成json形式的字符串

在node环境下，toJson方法不需要显示调用，当JSON.stringify方法调用的时候，会自动调用toJSON方法。

|  |
| --- |
| let buff = Buffer.alloc(5);  buff.write('hello');  console.log(JSON.stringify(buff));  1 |

1. 路径操作

在不同的操作系统之间，斜杆是不一样的，**window使用“\”(书写的时候要写“\\”)**，**linux系统使用“/”，**node它提供一个专门的模块来解决这个问题

1、path.basename(path[, ext])，获取路径中最后的一部分

path <string>

ext <string> 可选的文件扩展名

返回: <string>

path.basename() 方法返回 path 的最后一部分

|  |
| --- |
| let path = require('path');  let str = path.basename(\_\_filename);  console.log(str);  **输出 01.js** |

2、path.dirname(path)，获取路径名称

3、path.extname(path)，获取文件的扩展名

即 path 的最后一部分中从最后一次出现的 .（句点）字符到字符串的结尾。

|  |
| --- |
| let path = require('path');  let str = path.dirname(\_\_filename);  console.log(str);  console.log(path.extname(\_\_filename)); |
| path.extname('index.html');  // 返回: '.html'  path.extname('index.coffee.md');  // 返回: '.md'  **path.extname('index.');**  **// 返回: '.'**  path.extname('index');  // 返回: ''  **path.extname('.index');**  **// 返回: ''**  path.extname('.index.md');  // 返回: '.md' |

4、path.format(pathObject)，路径的格式化处理

任何具有以下属性的 JavaScript 对象：

dir <string>：文件的全路径

root <string>：文件的根路径

base <string>：文件的名称

name <string>：文件名称

ext <string>：扩展名

返回: <string>

path.format() 方法从对象返回路径字符串。 这与 path.parse() 相反。

当向 pathObject 提供属性时，请记住**存在一个属性优先于另一个属性**的组合：

如果提供 pathObject.dir，则忽略 pathObject.root

如果 pathObject.base 存在，则忽略 pathObject.ext 和 pathObject.name

5、path.parse(path)，路径的格式化处理

path.parse() 方法返回一个对象，其属性表示 path 的重要元素。

返回的对象将具有以下属性：

dir <string>

root <string>

base <string>

name <string>

ext <string>

|  |
| --- |
| let path = require('path');  let obj = path.**parse**(\_\_filename);  console.log(obj);  obj = {  root: 'C:\\',  dir: 'C:\\path\\hello',  base: 'temp.txt',  ext: '.txt',  name: 'temp'  };  console.log(path.**format**(obj));  1 |

6、path.isAbsolute(path)，方法确定 path 是否为绝对路径。

返回一个布尔类型的值，如果给定的 path 是**零**长度字符串，则将返回 false。

在 Windows 上：这两个比较特殊，用的也是比较少

path.isAbsolute('//server'); // true

path.isAbsolute('\\\\server'); // true

7、path.join([...paths])，拼接路径，在拼接过程的时候会格式化路径。

两个点表示上一个目录，一个点表示当前目录

|  |
| --- |
| path.join('/foo', 'bar', 'baz/asdf', 'quux', **'..'**);  输出：\foo\bar\baz\asdf |

1. path.normalize(path)，规范化路径。

解析 '..' 和 '.' 片段。

|  |
| --- |
| path.normalize('/foo/bar//baz/asdf/quux/..');  在 Windows 上：  path.normalize('C:\\temp\\\\foo\\bar\\..\\');  由于 Windows 识别多个路径分隔符，两个分隔符都将被 Windows 首选分隔符 (\) 的实例替换：  path.win32.normalize('C:////temp\\\\/\\/\\/foo/bar'); |

1. path.relative(from, to)，计算相对路径，从前面的路径怎么走到后面的路径

|  |
| --- |
| 在 POSIX 上：  path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');  // 返回: **'../../impl/bbb'，它要返回两个目录，然后才能进去impl。**  在 Windows 上：  path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb');  // 返回: '..\\..\\impl\\bbb' |

1. 两个特殊的属性

（1）path.sep

提供特定于平台的路径片段分隔符：

Windows 上是 **“\”**

POSIX 上是 **“/”**

1. path.delimiter

环境变量分隔符：

**;** 用于 Windows

**:** 用于 POSIX

六、异步处理，在node中叫异步I/O（input / output）

1、在浏览器中也存在着异步操作，有三种可以进入事件队列

（1）定时任务

（2）事件处理

（3）Ajax回调处理

1. 在node环境中，可以进入事件队列的有：
2. 文件读写操作
3. 网络的请求响应处理。

当主线程空闲的时候，它会去事件队列中查询，查询到有执行的，就主线程去处理。

注意：js的运行是单线程的，但是浏览器和node环境它们是多线程的

基本是以**异步的回调函数**的编码风格。

七、文件操作，注意回调参数，一般第一个是err。不能去对它进行处理

基本上所有的方法都是一对对的，带**Sync**的都是**同步**的方法，不带的就是**异步**的方法，它带有一个**回调函数**。

一般回调函数的第一个参数是错误对象，如果err为 null ，表示没有错误，否则表示报错了。

1. fs.stat(path[, options], callback)，查看文件的信息

fs.stat(path[, options], callback)

path <string> | <Buffer> | <URL>

options <Object>

bigint <boolean> 返回的 <fs.Stats> 对象中的数值是否应为 bigint。 默认值: false.

callback <Function>

err <Error>

stats <fs.Stats>

（1）需关注的返回值属性

atime 访问文件的时间

ctime 文件的状态信息发生变化的时间（比如文件的权限，在linux下，chrom命令）

mtime 文件数据发生变化的时间

birthtime 文件创建的时间

|  |
| --- |
| 异步  let fs = require('fs');  console.log(1);  fs.stat('./02.js', (err, stats) => {  // console.log(err);  if (err) {  return;  }  if (stats.isFile()) {  console.log('文件');  } else if (stats.isDirectory()) {  console.log('目录');  }  // console.log(stats);  console.log(2);  });  console.log(3);  输出 1、3、2，文件的回调函数丢到事件队列去了，等主线程是 1、3执行完毕后，空闲了再去处理，输出 2。  1 |
| 同步  let fs = require('fs');  console.log(1);  var ret = fs.**statSync**('./02.js');  console.log(ret);  console.log(2);  1  同步就是按照顺序输出了。 |

2、fs.readFile(path[, options], callback)

如果有第二个参数并且是utf8编码，那么回调函数获取到的数据就是字符串

如果没有第二个参数，那么得到的就是Buffer实例对象

|  |
| --- |
| let fs = require('fs');  let path = require('path');  let strPath = path.join(\_\_dirname, '02.js');  fs.**readFile**(strPath, (err, data) => {  if (err) {  return;  }  console.log(typeof data);  console.log(data);  console.log(data.toString());  });  1 |
| fs.readFile(strPath, **'utf8'**, (err, data) => {  if (err) {  return;  }    console.log(data);  }); |

3、fs.readFileSync(path[, options])

|  |
| --- |
| let ret = fs.**readFileSync**(strPath, 'utf8');  console.log(ret); |

3、fs.writeFile(file, data[, options], callback)

直接替换掉该文件

|  |
| --- |
| let fs = require('fs');  let path = require('path');  let str = path.join(\_\_dirname, 'data.json');  /\*fs.**writeFile**(str, 'hello tom', 'utf8', (err) => {  if ( !err) {  console.log('文件写入成功');  } else {  console.log('文件写入失败');  }  });\*/  fs.**writeFileSync**(str, 'hello Jerry'); |

以上的两个API，在文件操作的时候，会把文件的**内容全部先加载到内存中**，小文件没问题，但是大文件可能就不行了，要使用另外的方式。

4、大文件操作（流式操作）

为了处理大文件的方式，node提供了另外的方式：文件的流式操作，数据流就是一部分一部分的进行处理。

基于事件的处理方式：有数据读取的事件，网络请求的事件。

1. fs.createReadStream(path[, options])
2. fs.createWriteStream(path[, options])
3. **data**数据读取事件（读取一块就触发一次），**end**数据结束事件（全部读完，触发一次）

（4）readable.**pipe**(destination[, options])，把输入流（磁盘加载到内存）和输出流（内存写到磁盘上）连接到一块

destination <stream.Writable> 数据写入的目标。

options <Object> 管道选项。

end <boolean> 当读取器结束时终止写入器。默认值: true。

返回: <stream.Writable> 目标可写流，如果是 Duplex 流或 Transform 流则可以形成管道链。

readable.pipe() 方法绑定可写流到可读流，将可读流自动切换到流动模式，并将可读流的所有数据推送到绑定的可写流。 数据流会被自动管理，所以即使可读流更快，目标可写流也不会超负荷。

|  |
| --- |
| let fs = require('fs');  let path = require('path');  let srcvideo = 'C:\\Users\\ASUS\\Desktop\\html\_study\\workplace\\node';  let srcPath = path.join(srcvideo, '8Node.mp4');  let dest = 'C:\\Users\\ASUS\\Desktop';  let destPath = path.join(dest, '8Node.mp4');  let readStream = fs.**createReadStream**(srcPath);  let writeStream = fs.**createWriteStream**(destPath);  /\*let num = 0;  readStream.on(**'data'**, (chunk) => {  writeStream.write(chunk);  num++;  });  readStream.on(**'end'**, () => {  console.log('文件传输完毕', num);  });  \*/  readStream.**pipe**(writeStream); |

1. 目录操作
2. fs.mkdir(path[, options], callback)，异步创建目录
3. fs.readdir(path[, options], callback)，读取目录的内容

读取目录的内容。回调有两个参数 (err, files)，其中 files 是目录中文件名的数组，不包括 '.' 和 '..'。

1. fs.rmdir(path[, options], callback)，删除目录

**callback** <Function>

err <Error>

|  |
| --- |
| let fs = require('fs');  let path = require('path');  let newDir = path.join(\_\_dirname, 'hello');  /\*fs.**mkdir**(newDir, (err) => {  if (err) {  return;  }  });\*/  /\*fs.**readdir**(\_\_dirname, (err, files) => {  if (err) {  return;  }  console.log(files);  files.forEach((item, index) => {  fs.stat(path.join(\_\_dirname, item), (err, stats) => {  if (stats.isFile()) {  console.log(item + ' 文件');  } else if (stats.isDirectory()) {  console.log(item + ' 目录');  }  })  });  });\*/  fs.**rmdir**(newDir, (err) => {  console.log(err);  });  1 |

九、包，电脑要联网

一个包完成一个特定的功能，多个模块可以形成包，不过要满足特定的规则才能形式规范的包。

1. NPM（node.js package management）

全球最大的模块生态系统，里面所有的模块都是开源免费的，也是node.js的包管理工具。

<https://www.npmjs.com/>

1. npm的安装包方式
2. 本地安装

用于实际的开发工作。本地安装的包在**当前目录**下的node\_modules（npm自动帮我们创建的）里面。

1. 全局安装

全局安装的包一般用于命令行工具，直接在cmd中使用这个包。全局安装的包位于Node.js环境中的node\_modules目录下。C:\Users\ASUS\AppData\Roaming\npm\node\_modules\es-checker\bin\check.js（如果装了nvm，会出现在你设置的路径下。）

1. npm常用的命令
2. 安装包（如果没有指定版本号，那么安装最新的版本）

npm install -g 包名称 （全局安装）

npm install 包名称 （本地安装）

安装包的时候可以指定版本

npm install -g **包名称@版本号**

（2）卸载包

npm uninstall -g 包名称

npm uninstall 包名称

（3）更新包（更新到最新的版本）

npm update -g 包名称

npm update 包名称

|  |
| --- |
| 安装包测试  npm install -g es-checker  这个包用来查看当前的node环境对es6的支持  被装在这个目录下C:\Users\ASUS\AppData\Roaming\npm\node\_modules\es-checker\bin\check.js  1  当前环境支持92%的es6 |

1. 本地写的代码做成一个规范的包

首先需要单独的生成一个package.json的文件，这个文件就描述了包的一些规则。

（1）npm init

直接一直回车，使用默认的值，最后生成一个文件。

（2）npm init -y

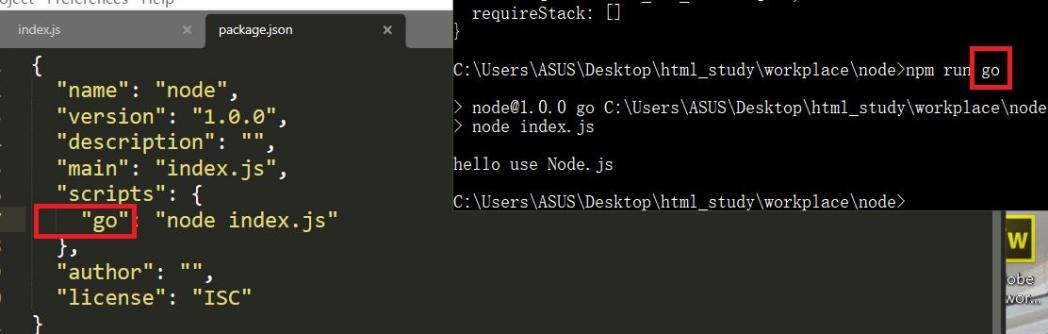
直接就生成这个文件，这样所有的属性值就是默认的

注：name和version是比较重要的

5、包执行的方式

（1）通过node . 的方式去执行这个包，点实际指向package.json里边的main属性

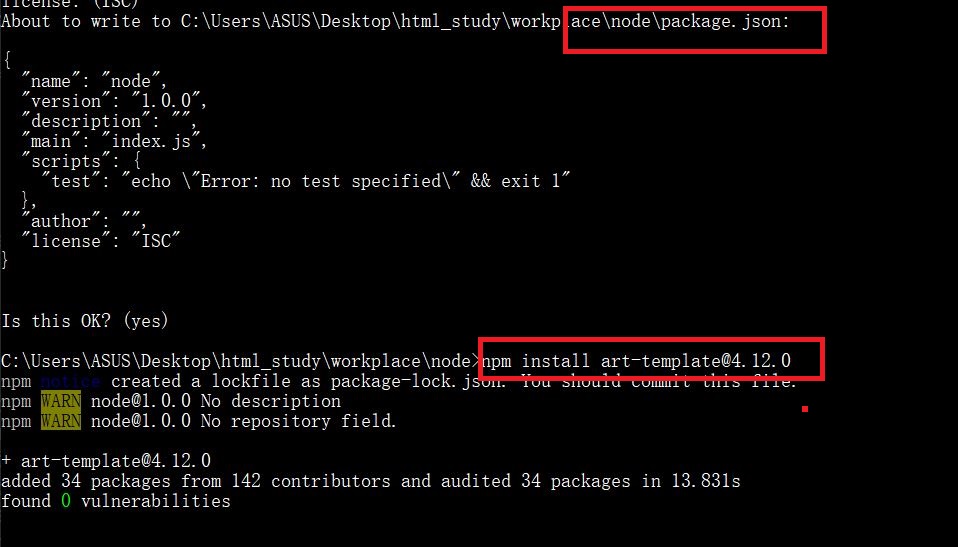
（2）npm run xxx，xxx实际指向package.json里边的script属性的键值对。



6、本地安装art-template模板引擎包

（1）npm init先生成一个package.json文件，否则会出错的。

（2）继续本地安装，它会跟刚才init创建的package.json建立关联，建立依赖

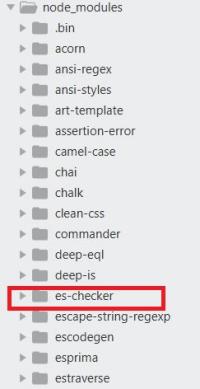
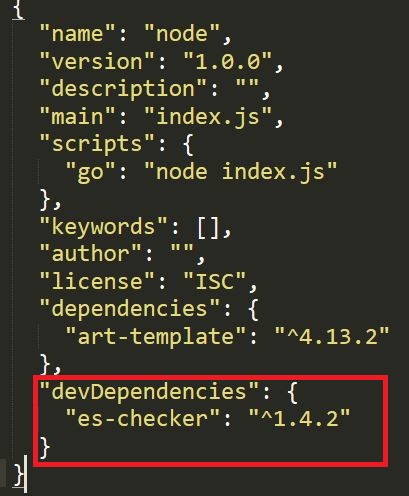




7、npm install 包名 --save / --save-dev

--save：在本地的项目（init之后）包入口添加依赖(如上边截图)。（生产环境）

--save-dev：在开发环境环境使用的依赖。（简单理解就是提高开发效率的一般是按照开发环境）

（1）开发环境（平时开发使用的环境），devDependencies向开发环境添加依赖。

（2）生产环境（项目部署上线之后的服务器环境），dependencies向生产环境添加依赖。

文件是放在devDependencies中的，这些东西产品正式发布后可以删除到的，因为不在需要了，所以这俩命令的区别就是：

你项目上线后，仍然需要用的东西用 --save；

不需要的一些包，只是在开发时借用环境什么的，可以直接用 --save-dev。

devDependencies 节点下的模块是我们在开发时需要用的，比如项目中使用的 gulp ，压缩css、js的模块。这些模块在我们的项目部署后是不需要的，所以我们可以使用 --save-dev 的形式安装。

像 express 这些模块是项目运行必备的，应该安装在 dependencies 节点下，所以我们应该使用 --save 的形式安装。

这里暂时还不是很了解。开发环境，不应该和生产环境一样吗？（后边自己搭建环境后很容易就知道的）

答：开发环境下的代码是没有进行压缩的，有各种打印，如果直接一样，那么生产可能会卡顿，加载速度慢的问题

在发布的时候，node\_modules是不会一起发布的，它的包含量比较大，传输比较麻烦，只会提交写的源代码，它所需要的依赖就添加到package.json文件中，那这个包在真正运行的时候，在随时的去安装它需要的包。

这个依赖为什么要分开装，生产环境的包，是程序运行所需要的包。

（3）npm install --production

拿到这个包之后，可以直接用过 npm install --production，它只会安装dependencies依赖的包。

8、node 开发环境与生产环境（简单理解，开发环境就是提高我们开发效率，生产环境是最终优化后的代码）

（1）什么是开发环境与生产环境

环境，就是指项目运行的地方。

当项目处于开发阶段，项目运行在开发人员的电脑上，项目所处的环境就是开发环境。

当项目开发完成以后，要将项目放到真实的网站服务器电脑中运行，项目所处的环境就是生产环境。

（2）为什么要区分开发环境与生产环境

因为在不同的环境中，项目的配置是不一样的，需要在项目代码中判断当前项目运行的环境，根据不同的环境应用不同的项目配置。

9、解决npm安装包被墙的问题（这个问题，已经通过nvm来解决了）

npm的源在国外，所以国内用户使用起来各种不方便。下面整理出了一部分国内优秀的npm镜像资源，国内用户可以选择使用。

（1）registry

npm config set registry=https//registry.npm.taobao.org

（2）cnpm

淘宝NPM镜像,与官方NPM的同步频率目前为10分钟一次

官网: http://npm.taobao.org/

npm install -g cnpm –registry=https//registry.npm.taobao.org

使用cnpm安装包: cnpm install 包名

（3）nrm

作用：修改镜像源

项目地址：https://www.npmjs.com/package/nrm

安装：npm install -g nrm

nrm的安装使用

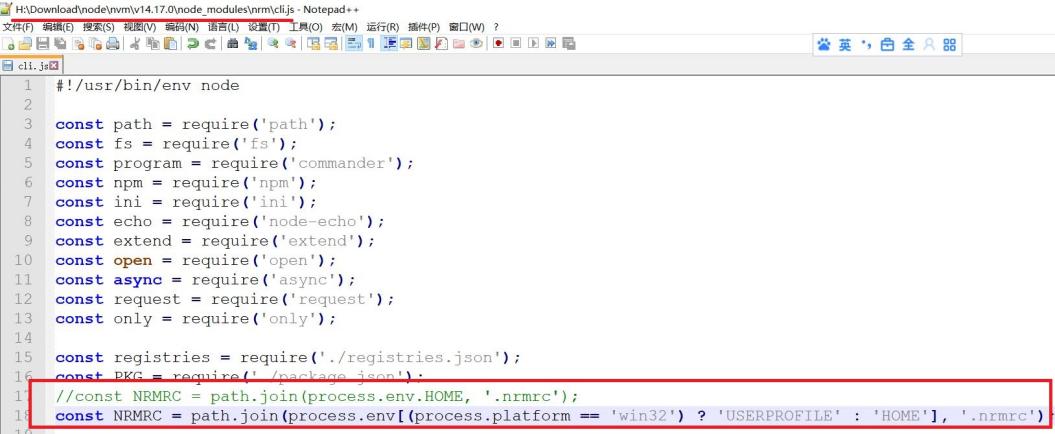
作用：提供了一些最常用的NPM包镜像地址，能够让我们快速的切换安装包时候的服务器地址； 什么是镜像：原来包刚一开始是只存在于国外的NPM服务器，但是由于网络原因，经常访问不到，这时候，我们可以在国内，创建一个和官网完全一样的NPM服务器，只不过，数据都是从人家那里拿过来的，除此之外，使用方式完全一样；

运行npm install nrm -g全局安装nrm包；

使用nrm ls查看当前所有可用的镜像源地址以及当前所使用的镜像源地址；

使用nrm use npm或nrm use taobao切换不同的镜像源地址；

**安装 nrm 后，执行 nrm ls 命令出现错误，解决如下**

****

**const NRMRC = path.join(process.env[(process.platform == 'win32') ? 'USERPROFILE' : 'HOME'], '.nrmrc');**

（1）使用cnpm工具下载和安装包

大家要区分 cnpm 这个工具 和 nrm 中 cnpm 的区别；

（1）在 nrm 中，cnpm 只是一个镜像的地址而已；

（2）在cnpm 这个工具中，它是一个具体的工具，能够像npm一样，去下载和安装包！

运行npm i cnpm -g 全局按照 cnpm 这个下载包工具可以像使用npm装包一样，去使用cnpm装包；安装jQuery为例子

npm install jquery -S

cnpm install jquery -S

十、yarn工具基本使用

Facebook开发的

1. 安装yarn工具

npm install -g yarn

1. 常用命令

（1）初始化包

npm init

yarn init

1. 安装包

npm install xxx --save

yarn add xxx

（3）移除包

npm uninstall xxx

yarn remove xxx

（4）更新包

npm update xxx

yarn upgrade xxx

（5）安装开发环境依赖的

npm install xxx --save-dev

yarn add xxx --dev

（6）全局安装

npm install -g xxx

yarn global add xxx

（7）设置下载镜像的地址（暂时无法理解）

npm config set registry url地址

yarn config set registry url地址

（8）安装所有的依赖

npm install

yarn install

1. 执行包

npm run json文件里那个script的命令

yarn run json文件里那个script的命令

十一、自定义包

1、包的规范

package.json必须在包的顶层目录下

二进制文件应该在bin目录下

JavaScript代码应该在lib目录下

文档应该在doc目录下

单元测试应该在test目录下

2、package.json字段分析

name：包的名称，必须是唯一的，由小写英文字母、数字和下划线组成，不能包含空格

version：符合语义化版本识别规范的版本字符串

description：包的简要说明，描述这个包是干嘛，怎么用

main：包入口文件

keywords：关键字数组，别人在搜索这个包的时候，能搜索到的相关字

maintainers：维护者数组，每个元素要包含name、email（可选）、web（可选）字段

contributors：贡献者数组，格式与maintainers相同。包的作者应该是贡献者数组的第一- 个元素

bugs：提交bug的地址，可以是网站或者电子邮件地址

licenses：许可证数组，每个元素要包含type（许可证名称）和url（链接到许可证文本的- 地址）字段

- repositories：仓库托管地址数组，每个元素要包含type（仓库类型，如git）、url（仓- 库的地址）和path（相对于仓库的路径，可选）字段

- dependencies：生产环境包的依赖，一个关联数组，由包的名称和版本号组成

- devDependencies：开发环境包的依赖，一个关联数组，由包的名称和版本号组成

十二、node实现静态服务器功能

1、http提供了创建服务器的功能，初步实现服务器功能

'request' 事件

request <http.IncomingMessage>

response <http.ServerResponse>响应回复的信息

每当有请求时触发。 每个连接可能有多个请求（对于 HTTP Keep-Alive 连接而言）。

|  |
| --- |
| const http = require('http');  // 创建服务器实例对象  /\*const server = http.createServer();  // 绑定请求事件  server.on('request', (request, response) => {  response.end('hello i am node server');  });  // 监听端口  server.listen(3000,'192.168.43.153',() => {  console.log('start server');  });\*/  http.createServer((require, response) => {  response.end('hello i am server');  }).listen(3000, '192.168.43.153', () =>{  console.log('start server');  }); |
| 在网页端，输入192.163.43.153:3000，就能看到hello i am node server  如果没有监听具体的ip，那么可以通过localhost:3000进行访问。  这样子无论你在路径请求什么，它返回都是一样的。 |

2、请求路径的分发

require：http.IncomingMessage 类的实例对象

response：http.ServerResponse 类的实例对象

|  |
| --- |
| /\*  处理请求路径的分发  \*/  http.createServer((require, response) => {  // require.url可以获取URL中的路径（端口之后部分）  if (require.url.startsWith('/index')) {  // write 向客户端响应内容，可以写多次  response.write('first ');  response.write('second ');  response.write('third ');  response.end('index.html');  } else if (require.url.startsWith('/about')) {  // end方法用来完成响应，只能执行一次  response.end('about.html');  } else {  response.end('no content');  }  }).listen(3000, '192.168.43.153', () =>{  console.log('start server');  }); |

1. 响应完整的页面

（1）response.writeHead(statusCode[, statusMessage][, headers])

statusCode <number>

statusMessage <string>

headers <Object>

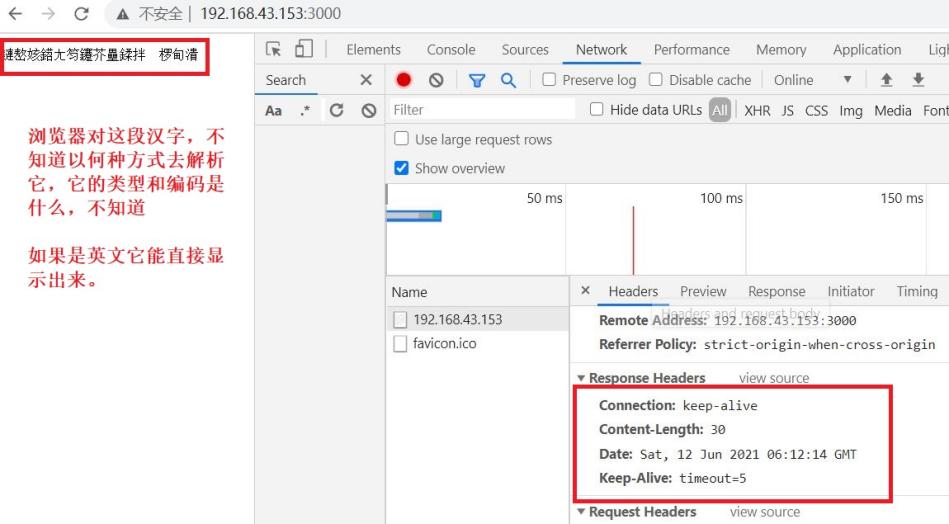
返回: <http.ServerResponse>

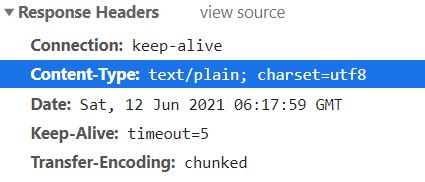
向请求发送响应头。

|  |
| --- |
| /\*  请求完整的页面  \*/  let readFile = (url, response) => {  fs.readFile(path.join(\_\_dirname, 'www', url), 'utf8', (err, fileContent) => {  if (err) {  response.end('server error');  } else {  response.end(fileContent);  }  });  }  http.createServer((require, response) => {  // require.url可以获取URL中的路径（端口之后部分）  if (require.url.startsWith('/index')) {  readFile('index.html', response);  } else if (require.url.startsWith('/about')) {  readFile('about.html', response);  } else if (require.url.startsWith('/list')) {  readFile('list.html', response);  } else {  response.writeHead(200, {  'Content-Type': 'text/plain; charset=utf8',  });  response.end('服务器不能找到该页面');  }  }).listen(3000, '192.168.43.153', () =>{  console.log('start server');  }); |

（2）响应头信息，plain表示纯文本，charset如果不指定的话，那么浏览器会根据操作系统的编码去显示的。（汉字的显示就需要）

在谷歌装个charset插件，看一下默认的编码是什么哪种





格式决定了浏览器怎么显示响应的数据

4、对完整页面的优化

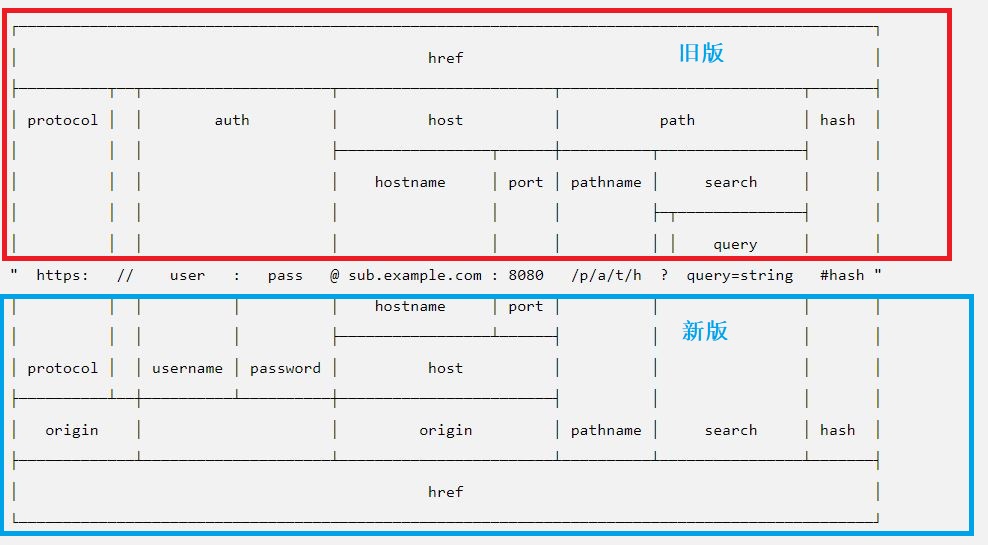
|  |
| --- |
| const http = require('http');  const path = require('path');  const fs = require('fs');  const mime = require('./mime.json');  /\*  请求完整的页面  \*/  http.createServer((require, response) => {  // require.url可以获取URL中的路径（端口之后部分）  fs.readFile(path.join(\_\_dirname, 'www', require.url), (err, fileContent) => {  if (err) {  response.writeHead(404, {  'Content-Type': 'text/plain; charset=utf8',  });  } else {  // 根据文件的后缀，设置不同的响应头  let dtype = 'text/html';  let type = path.extname(require.url);  if (mime[type]) {  dtype = mime[type];  }  // 如果是以text开头的类型，就设置一下编码  if (dtype.startsWith('text')) {  dtype += '; charset=utf8';  }  response.writeHead(200, {  'Content-Type': dtype,  });  response.end(fileContent);  }  });  }).listen(3000, '192.168.43.153', () =>{  console.log('start server');  });  1 |

5、响应完整页面封装成包

|  |
| --- |
| const path = require('path');  const fs = require('fs');  const mime = require('./mime.json');  exports.staticServer = (require, response, staticSource) => {  // require.url可以获取URL中的路径（端口之后部分）  fs.readFile(path.join(staticSource, require.url), (err, fileContent) => {  ...............  });  } |
| const http = require('http');  const ss = require('./server.js');  const path = require('path');  http.createServer((require, response)=>{  let staticSource = path.join(\_\_dirname, 'www');  ss.staticServer(require, response, 'C:\\Users\\ASUS\\Desktop\\html\_study');  }).listen(3000, '192.168.43.153', (err) => {  if (err) {  return;  }  console.log('running');  });  这样的话，静态资源的目录，也能够比较方便设计 |

5、get参数的传递与获取

一个完整的 URL 'https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'



（1）旧版API，get是通过url这个模块

host包括端口

hostname不包括端口

hash哈希，锚点

search表示问号后边的那部分，参数

query就是去掉问号的那部分

pathname问号前到域名后之间的路径

|  |
| --- |
| const url = require('url');  http.createServer((require, response)=>{  var ret = url.parse(require.url, true);  console.log(ret);  response.writeHead(200, {  'Content-Type': 'text/plain utf8',  });  response.end('okoko'+ ret.query.username+' '+ret.query.password);  }).listen(3000, '192.168.43.153', (err) => {  if (err) {  return;  }  console.log('running');  });  1 |

（2）新版API

|  |
| --- |
| const url = new URL('http://wwwbaidu.com?flag=1&a=4');  console.log(url.searchParams.get('flag'));  目前使用的是192开头url，不符合新版的要求，暂时先这样  错误的示例  new URL(require.url, `http://${request.headers.host}`).searchParmas.get('username');  http那里是使用的是反向引号。  正确  new URL(require.url, **`http://${require.headers.host}`**).searchParams.get('username');  1 |

6、post参数的传递与处理

（1）querystring.parse方法的作用就是把参数字符串转成对象

（2）querystring.stringify的作用就是把对象转成参数字符串的格式

|  |
| --- |
| const querystring = require('querystring');  let ret = querystring.parse('flag=1&color=red');  console.log(ret);  let obj = {  color: 'red',  name: 'Tom',  };  console.log(querystring.stringify(obj));  1 |

（3）post方式要基于事件的方式来处理

（4）require绑定data事件，绑定end事件

|  |
| --- |
| 通过html的表单来提交post请求，可以采用postman来模拟post请求  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <form action="http://192.168.43.153:3000/login" method="post">  用户名：<input type="text" name="usrname" /><br />  密码：<input type="text" name="password" />  <input type="submit" name="">  </form>  </body>  </html> |
| const http = require('http');  const ss = require('./server.js');  const path = require('path');  const querystring = require('querystring');  http.createServer((require, response)=>{  console.log(require.url);  if (require.url.startsWith('/login')) {  let content = '';  require.on('data', (chunk) => {  content += chunk;  });  require.on('end', () => {  let result = querystring.parse(content);  response.writeHead(200, {  'Content-Type': 'text/plain; charset=utf8',  });  response.end('接收的用户名：'+result.usrname+' 密码：'+result.password);  });  }  }).listen(3000, '192.168.43.153', (err) => {  if (err) { return; }  console.log('running');  }); |

十三、动态网站

它是与参数的传递是密切相关的，就是根据不同的请求路径，和参数的传递得到不同的内容，浏览器端获取到服务器端的内容，它是动态生成的，并不是向静态资源那样提前写好的。

1、路由（请求路径 + 请求方式），将请求分发到不同处理逻辑中去

2、所遇到问题：

（1）在查询页面无法跳转到成绩页面

在form标签中的action写漏了端口，在http后面写多了一个冒号

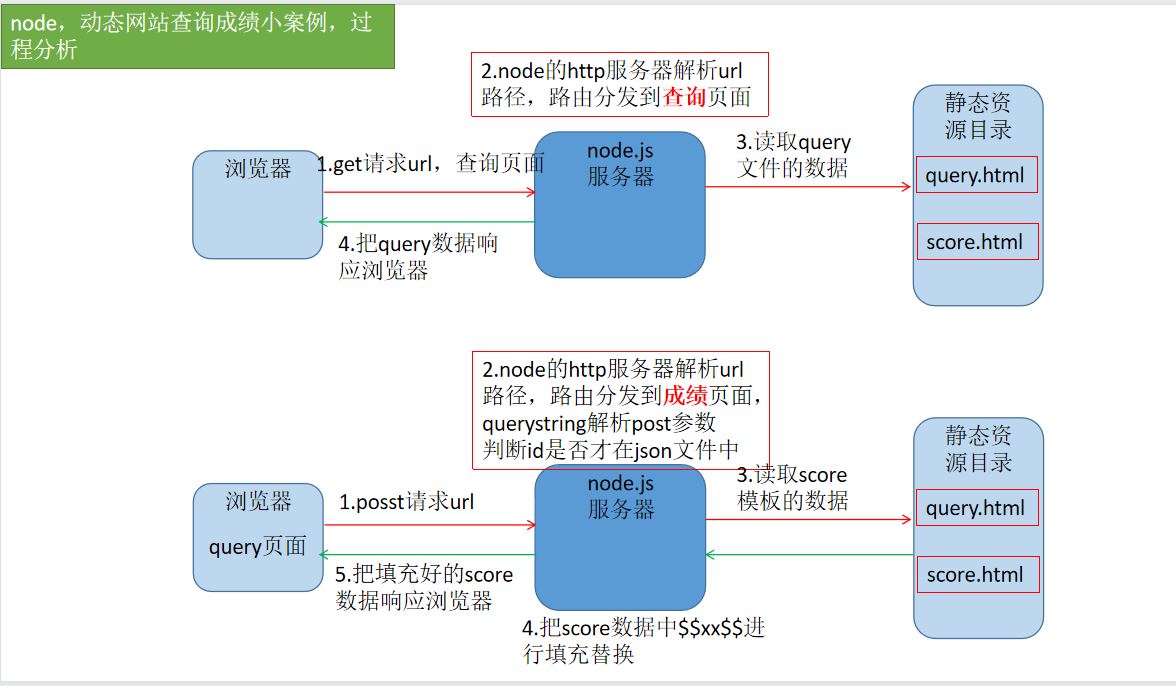
（2）无论输入什么id，一直显示找不到信息

对post、传递过来的数据没有解析（id=xxx）就直接去判断，这样肯定是找不到的，先要进行解析，在处理。

解决思路，把相关的信息量都打印出来，想的跟实际有时候是有区别的，有时候想错了

（3）readFile()中path.join忘记写路径了，导致读取文件错误

（4）replace，的用法忘记了它是返回一个新的字符串，直接用的原来的，发现没有更新成功，看文档后解决。耗时45分钟左右吧。



|  |
| --- |
| Index.html  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>查询页面</title>  </head>  <body>  <!-- 不要忘了写端口，http后面多写了一个冒号-->  <form action="http://192.168.43.153:3000/score.html" method="post">  输入考生号<input type="text" name="id" placeholder="如：no104" />  <input type="submit" name="">  </form>  </body>  </html> |
| score.html  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>成绩</title>  </head>  <body>  <ul>  <li>姓名：$$name$$</li>  <li>语文：$$chinese$$</li>  <li>数学：$$math$$</li>  <li>英语：$$english$$</li>  <li>综合：$$summary$$</li>  </ul>  </body>  </html> |
| info.json  {  "no110": {  "name": "Tom",  "chinese": 100,  "math": 150,  "english": 130,  "summary": 180  },  } |
| const http = require('http');  const path = require('path');  const querystring = require('querystring');  const info = require('./www/info.json');  const fs = require('fs');  http.createServer((require, response)=>{  *// 路由分发，get要大写*  if (require.url.startsWith('/index') && require.method === 'GET') {  *// 把查询网页的内容响应给浏览器*  fs.readFile(path.join(\_\_dirname, 'www', require.url), (err, fileContent) => {  if (err) {  response.writeHead(500, {  'Content-Type': 'text/plain; charset=utf8',  });  response.end('服务器内部错误');  return;  }  response.end(fileContent);  });  } else if (require.url.startsWith('/score') && require.method === 'POST') {  *// 查询成绩*  let stuId = '';  require.on('data', (chunk) => {  stuId += chunk;  });  require.on('end', () => {  *// 还要对post传过来的数据进行解析，不能直接用 id=no110*  let resultId = querystring.parse(stuId).id;  *// 先判断有没有这个id*  if (info[resultId]) {  *// 如果有的话，就把显示的模板数据，读取出来，然后填充数据，再返回给浏览器*  fs.readFile(path.join(\_\_dirname, 'www', '/score.html'), 'utf8', (err, fileContent) => {  if (err) {  response.writeHead(500, {  'Content-Type': 'text/plain; charset=utf8',  });  response.end('服务器内部错误');  return;  }  *// 填充数据,replace会返回一个新的结果*  fileContent = fileContent.replace('$$name$$', info[resultId].name);  fileContent = fileContent.replace('$$chinese$$', info[resultId].chinese);  fileContent = fileContent.replace('$$math$$', info[resultId].math);  fileContent = fileContent.replace('$$english$$', info[resultId].english);  fileContent = fileContent.replace('$$summary$$', info[resultId].summary);  response.end(fileContent);  });  } else {  response.writeHead(500, {  'Content-Type': 'text/plain; charset=utf8',  });  response.end('未能查找到该学生信息');  return;  }  });  }  }).listen(3000, '192.168.43.153', (err) => {  if (err) {  return;  }  console.log('server running');  }); |

3、登录验证功能

1. 问题：  
   username在html中敲错了 usrname，以及‘/www’的斜线容易不见了，都会导致问题

|  |
| --- |
| <!DOCTYPE html>  <**html** lang="en">  <**head**>  <**meta** charset="UTF-8">  <**meta** name="viewport" content="width=device-width, initial-scale=1.0">  <**title**>登录界面</**title**>  </**head**>  <**body**>  <**form** action="http://192.168.43.153:3000/check" method='post'>  用户名：<**input** type="text" name="username" /><**br** />  密码：<**input** type="password" name="password" /><**br**/>  <**input** type="submit">  </**form**>  </**body**>  </**html**> |
| **const** http = require('http');  **const** path = require('path');  **const** querystring = require('querystring');  **const** fs = require('fs');  **const** ss = require('./server.js');  **const** url = require('url');  http.createServer((require, response)=>{  *// 静态资源服务*  **if** (require.url.startsWith('/www')) {  ss.staticServer(require, response, \_\_dirname);  } **else** **if** (require.url.startsWith('/check')) {  *// 动态资源*  **if**(require.method === 'GET') {  *// 新版本的参数暂时不知道怎么去获取和处理，错误示例*  *// let username = new URL(require.url).searchParmas.get('username');*  *// let password = new URL(require.url).searchParmas.get('password');*  **let** username = url.parse(require.url,**true**).query.username;  **let** password = url.parse(require.url,**true**).query.password;   |  | | --- | | *// 新版本的参数获取和处理*  **let** username = **new** URL(require.url, `http:*//${require.headers.host}`).searchParams.get('username');*  **let** password = **new** URL(require.url, `http:*//${require.headers.host}`).searchParams.get('password');* |   response.writeHead(200, {  'Content-Type': 'text/plain; charset=utf8',  });  **if** (username === 'admin' && password === '123') {  response.end('get密码正确');  } **else** {  response.end('get密码错误');  }  }  **if** (require.method === 'POST') {  **let** info = '';  require.on('data', (chunk) => {  info += chunk;  });  require.on(**'end'**, () => {  **let** username = querystring.parse(info).username;  **let** password = querystring.parse(info).password;  response.writeHead(200, {  'Content-Type': 'text/plain; charset=utf8',  });  **if** (username === 'admin' && password === '123') {  response.end('post密码正确');  } **else** {  response.end('post密码错误');  }  });  }  }  }).listen(3000, '192.168.43.153', (err) =>{  **if** (err) {  **return**;  }  console.log('server running');  }); |

1. art-template模板引擎node.js环境下

1、跟在浏览器端还是有区别的。

1. 用法举例

|  |
| --- |
| scoreAll.tpl  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>成绩</title>  </head>  <body>  {{each all}}  <ul>  <li>姓名：{{$value.name}}</li>  <li>语文：{{$value.chinese}}</li>  <li>数学：{{$value.math}}</li>  <li>英语：{{$value.english}}</li>  <li>综合：{{$value.summary}}</li>  </ul>  {{/each}}  </body>  </html> |
| let arr = [];  for (var key in info) {  arr.push(info[key]);  }  let html = template(\_\_dirname + '/scoreAll.tpl', {  **all**: arr,  });  参数要以对象的形式传递进去，属性值要做成一个数组的形式。 |