# NPM管理

###### 介绍:

全称: Node Package Manager , Node的包管理器,是一个应用程序

###### 包是什么:

Node.js的包遵循CommonJS规范,将一组相关模块组合在一起,形成完整的工具

人话:包就是封装好的代码

###### 作用:

通过NPM可以对Node的工具包进行管理(下载安装删除上传).借助别人写好的包,可以让我们开发更加方便

###### 安装:

安装完nodejs后会自动安装npm

## 常用命令:

npm对大写英文和中文支持并不好,统一用小写字母即可;

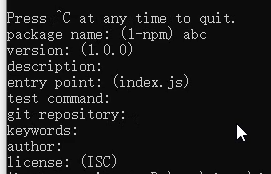
##### 查看版本 npm -v

##### 初始化 npm init / 快速初始化(都是默认值) npm init --yes

//npm init 创建一个JSON格式的配置文件,并且需要配置(文件名,权限等);

//若使用npm init --yes 则文件夹名不能包含中文和大写英文

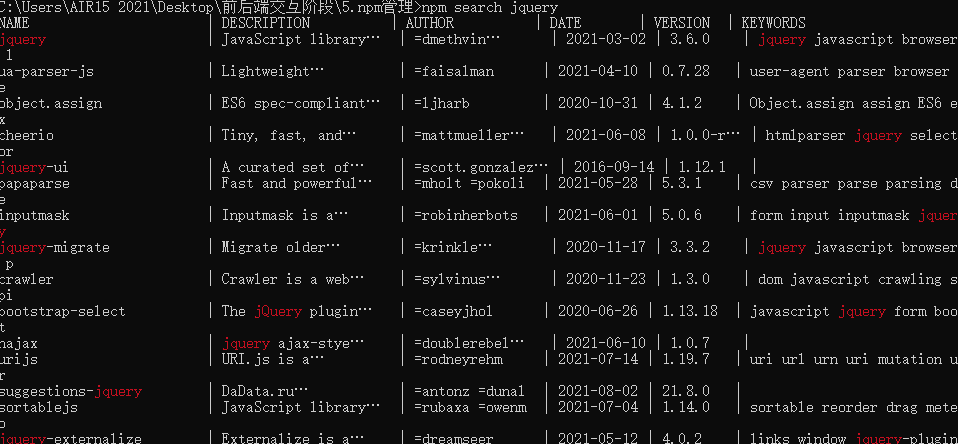
//package.json是npm包的配置文件





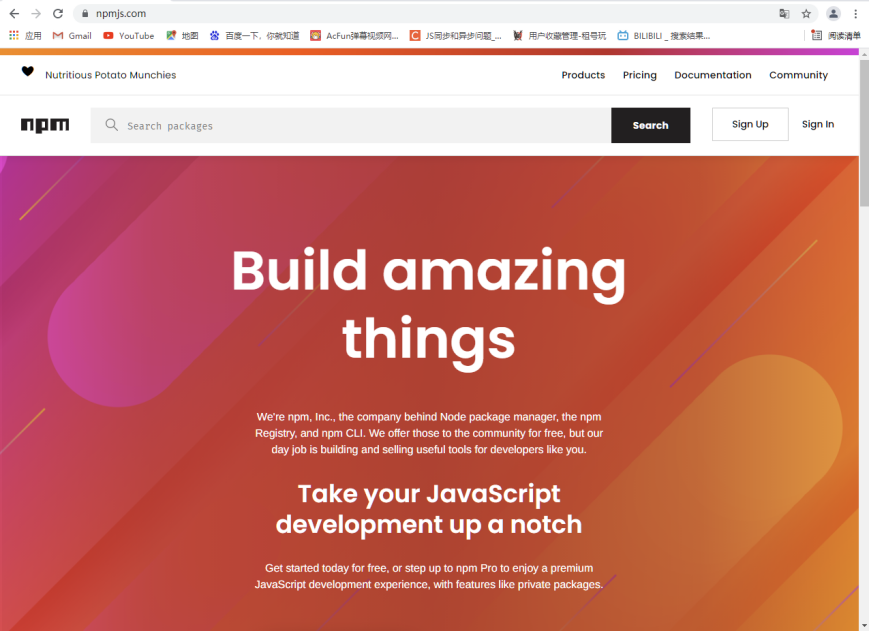
##### 搜索命令 npm search {关键字} / npm s {关键字}

//搜索有关jquery的包 npm search jquery



//比较少用,一般去https://npmjs.org 等网站搜索

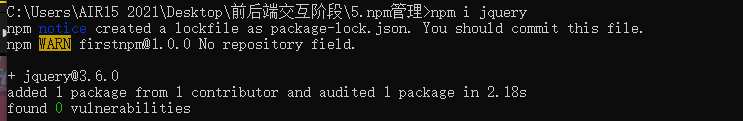
//https://npmjs.org就是npm的中心仓库,所有的包都在此存放,,类似github



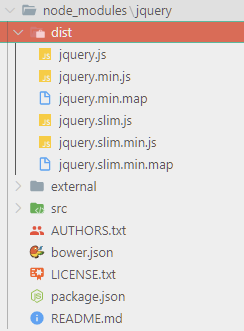
npm用处：使用jquery库,需要从juqery网站下载,然后在index.html文件script标签中引入.比较麻烦,但是许多网站下载非常麻烦(慢),利用npm 下载比较快捷

npm演示:引入jquery

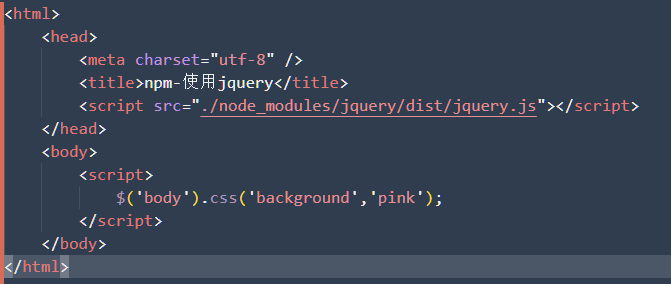
// npm i jquery 下载jquery包



//下载生成jquery的文件夹



//使用juqery



使用步骤小结:

1 初始化 npm init 并配置 //每个文件夹执行一次即可

2 npm search {工具包名} //寻找工具包(网站也可)

3 npm i {工具包名} //安装工具包

4 node\_modules //安装工具包的存放位置

5 引用 工具包

##注意npm命令运行的位置

##package.json所在文件夹才能执行npm命令,否则不起作用

##而package.json一般在项目的根目录

##### npm包在服务端的使用:

// 像上述html直接引用外链式引入jquery是在html文件中的方式

// 而服务端的引入方式使用require方法进行

// 引入工具包还有另一种方式,一般在服务端(nodeJS环境下)进行引入

//步骤:

//使用npm i 或者npm install命令下载工具包

//使用const {变量名} = require(“工具包名”);

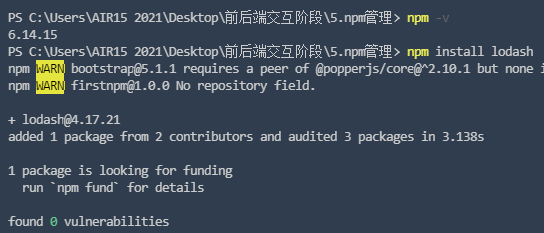
\*\*和nodejs内置模块的使用比较类似

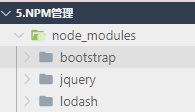


示例: 在服务端nodejs中引入npm包

//在npm初始化过的文件夹(特征是拥有package.json)内运行npm命令

//1.下载lodash工具包 npm install lodash / npm i lodash





// 2. 在node文件中引入lodash模块

// 3. 使用lodash模块



##### 开发依赖和生产依赖的介绍:

//安装命令

//npm i & npm install

安装命令的两组参数(--save -S)(--save--dev -D)

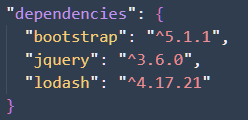
npm install {工具包名} --save

npm install {工具包名} -S

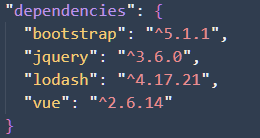
//-S --save参数会将npm安装包的版本纪律在package.JSON的dependencies属性中;

//而6版本之后,npm安装包都会自动添加-S参数,自动保存包的版本信息

//dependencies用来保存当前项目的依赖;

dependencies中文意思: 依赖

npm install vue --save 后



//应用场景(记录依赖信息 ):

记录的执行本项目所依赖的工具包;当电脑不足以运行时,可以查看当前的依赖关系,进行补齐;

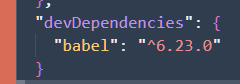
npm install {工具包名} --save-dev

npm install {工具包名} -D

//--save-dev -D会将工具包的版本信息放入package.json的devdependencies属性;

// devdependencies : 开发依赖;

npm i bable --save-dev:



生产依赖dependencies和开发依赖devdependcies的区别:

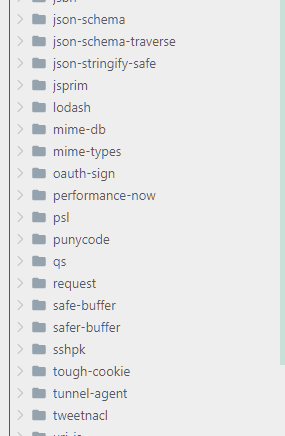
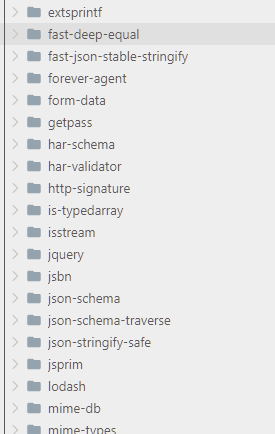
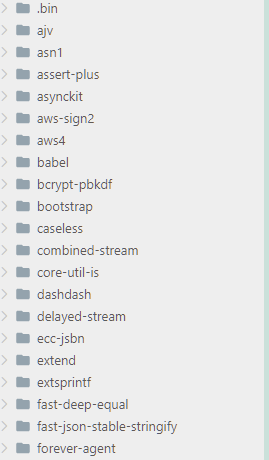
生产依赖:项目运行过程中所依赖的工具包(运行环境),如jquery,vue,bootstrap等

开发依赖:开发项目过程所依赖的工具包: 如webpack等;

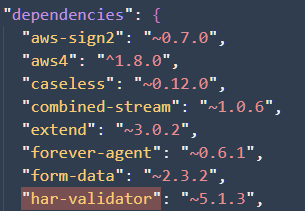
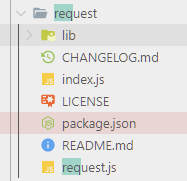
##### 关于依赖的说明:

在安装request工具包后,会发现多出其他文件夹,这是以为request工具包依赖于其他工具包;

就是递归的依赖链条,就像linux系统中接触的那样



每个工具包自身的依赖关系存在工具包目录下的package.json文件下:



##### 全局安装： NPM INSTALL {} -g

//全局安装 -g

npm install less -g

//查看全局安装的路径

npm root -g

//全局安装一般用于安装全局工具,如cnpm,yarn,webpack,gulp等;

//全局安装命令 安装工具包的位置 C:\Users\你的用户名\AppData\Roaming\npm

//1全局安装不会影响当前项目的内容(安装路径不同)

//2在任意的命令行位置都可以执行\*全局安装\* npm i {} -g

//3 使用方式都是在命令行中

//3的注释:普通安装的工具包都是在 外链式引用或node中require引用;

但是全局安装的工具包引用方式在命令行中

例如less工具包lessc指令可把less转换成css代码;

Lessc index.less index.css

例如nodemon工具包可以在ndoejs文件改动的时候自动重启node 服务

##### 环境变量path:

//环境变量本人接触很多:作用就是可执行程序能在任意文件位置执行;

//命令行执行命令,系统会率先寻找当前目录是否有同名的可执行文件,若无

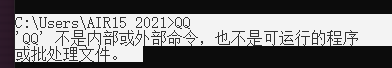
再去环境变量PATH中寻找,若还是没有报错;

//未配置环境变量

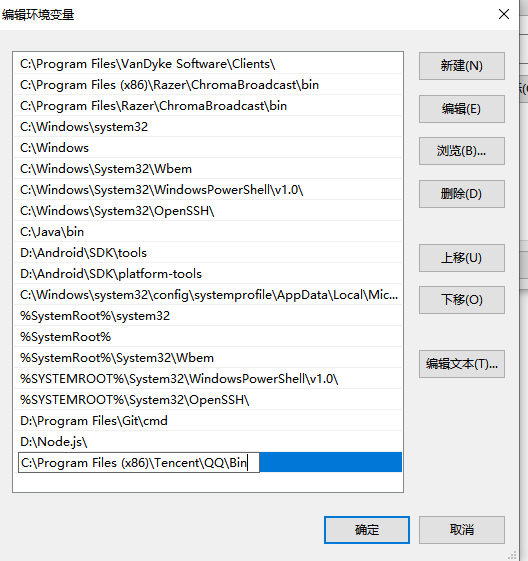
C:\Program Files (x86)\Tencent\QQ\Bin路径下可以执行QQ

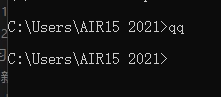


C:\Users\AIR15 2021 该目录下没有该执行程序,不可以执行QQ命令



只要将QQ可执行文件的目录 存入 环境变量path中,就可以任意位置的命令行调用QQ命令;





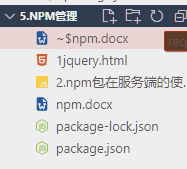
### 纯命令npm i 和 npm install

npm i 和 npm install 不添加工具包名直接调用;

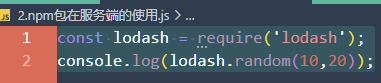
系统会安装 package.JSON中的生产依赖和开发依赖中所记录的工具包;

演示:

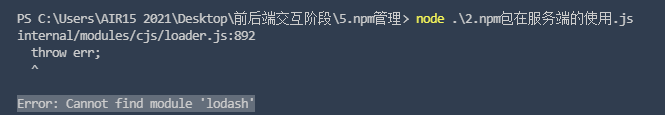
直接把node\_modules文件夹删除



此时执行2.npm包在服务端的使用.js



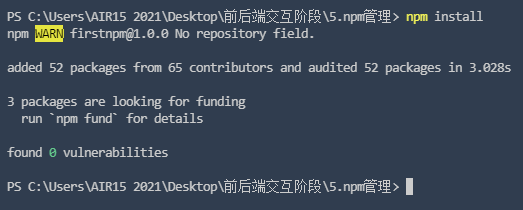
报错:(引入第三方库常见错误)



找不到模组”lodash’, 当然是因为我把node\_modules文件夹删了;

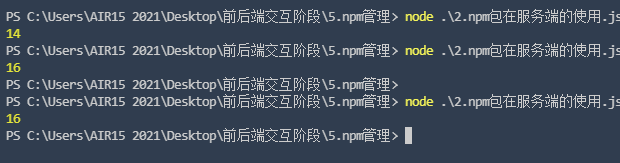
但是package.json下记录了执行依赖和开发依赖;

利用npm i 可以下载安装 package.json下所依赖的工具包;



使用npm i的前提是有package.json文件记录了依赖;

安装完依赖后可以执行2.JS



得出结论:

npm i自动补全package.JSON文件依赖;

Require()引入第三方模组时,nodejs会在node\_modules目录下寻找工具包;

当前目录没有node\_modules会到上一级目录寻找该目录,直到顶层目录,若还是没有,报错,cannot find modules

npm i --production // 只安装 dependencies 中的依赖

### npm 在多人协作中的注意点:

!!!!!##node\_modules 目录是不建议进入Git仓库的;

原因:

//node\_modules目录的内容都是第三方的代码;不是开发人员的代码;

//node\_modules目录一般体积特别庞大,不便于git管理(例如git add命令变慢);

//node\_modules目录内的第三方库一般在开发过程中不会变动;

!!!!!以后上传到仓库的代码要记得附带package.json,里面包含了项目所需依赖;

实际操作:

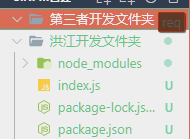
//将node\_modules放入.gitignore文件中;git不再管理这个文件

//由于不上传node\_modules,git中心仓库只有核心代码和package.json文件

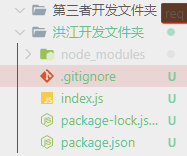
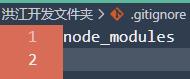
//开发者从仓库克隆的文件只有核心代码和package.json文件,需要npm i 来补全依赖库;

演示:

文件夹结构如下: (第一方开发完毕,第三方开发者准备接手)



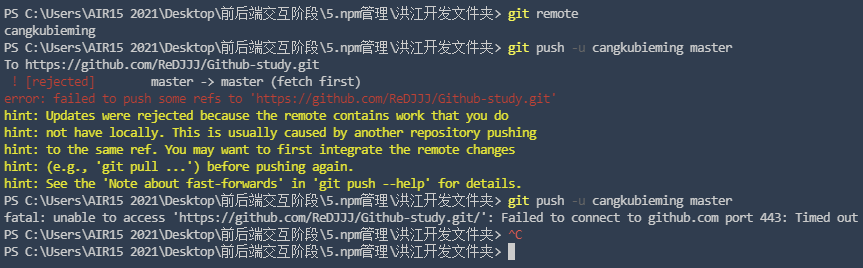
第一方初始化git仓库并设立git忽略文件.gitignore填入node\_modules



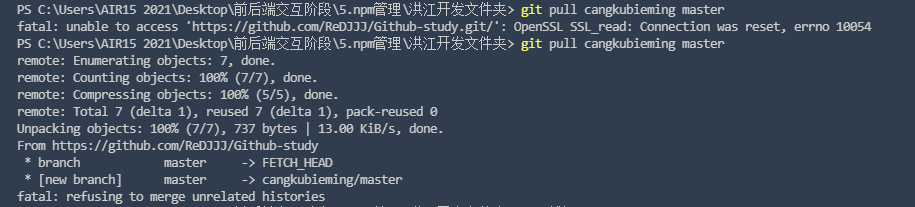
第一方上传

//上传出错,是因为远程库有本地库没有的文件

//git pull进行同步



//同步后就可以上传



//网络质量是在太差,后续就不演示了;

//总之:就是node\_modules忽略git,从中心代码下载的代码npm i 来根据package.josn文件的依赖来进行安装npm包,上传代码的时候也要注意package.json是否成功上传;

### npm和git命令结合使用:

//npm init创建了package.JSON,后续还会创造node\_modules

//git init 创建了.git ,后续还会创造.gitignore

### 自定义封装一个npm包上传到npmjs网站托管

//就像自己为自己封装函数那样,也可以自己封装npm包

封装步骤:

//前提:拥有一个npmjs的账号

\* 编写代码块并使用 module.exports暴露

\* 将npm 指向 官方地址(特别是指向了国内npm镜像)

执行npm config set registry https://registry.npmjs.org/

\* 初始化npm init

npm init 配合--yes将以文件夹名为包名,包名不允许大写 中文

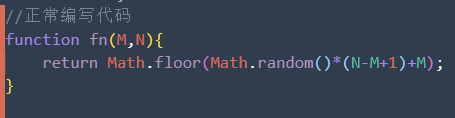
\* 命令行中登录npmjs

一定要在需要打包且已经暴露的文件中登录

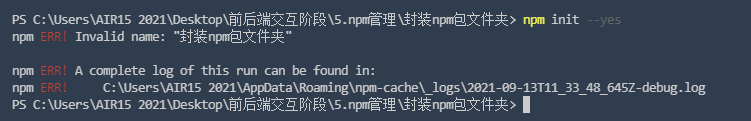
npm login 登录npmjs账号

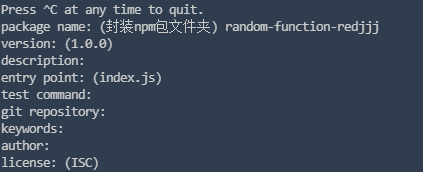
\* 提交包

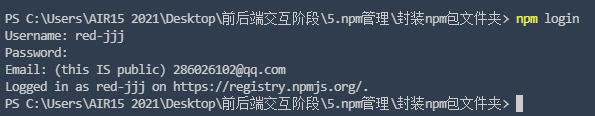
npm publish 将会将此文件夹打包成npm包并上传到npmjs







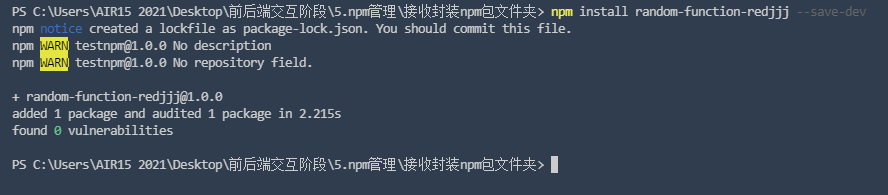






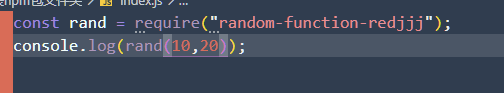


引用包阶段:



//在暴露阶段module.export就是简单的暴露了一个随机数函数

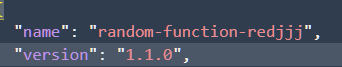
//所以require的返回值就是函数; 引入后直接调入即可;



### 包的升级和包的删除

包的升级:

1. 进入package.json修改版本号
2. npm login
3. npm publish





删除:

删除npmjs 中心库的包 (npmjs账户发布的包)

npm --unpublish 包名

删除本地安装的包

npm remove 包名

# Cnpm(了解即可)

Cnpm: 淘宝对国外npm服务的一个完整镜像版本,也就是淘宝的npm镜像

网站网址http://npm.taobao.org/

Cnpm 版本比较老,需要添加-S 才会保存package.JSON 执行依赖dependencyies中

由于淘宝等国内镜像服务器在国内,因此稳定性和下载速度明显更快

可以将npm命令的地址指向改变

//淘宝镜像

npm config set registry <http://registry>.npm.taobao.org

//官方镜像

npm config set registry <http://registry>.npmjs.org/

//注意,除了官方镜像,其余镜像都是只读的,并且与官方有一定的延迟

//所以配置了国内镜像,在publish工具包的时候要注意转回官方的镜像才可以发布;

### Yarn的介绍与安装

Yarn 是Facebook开源的包管理器,用来代替npm

特点(对于npm):

\* 本地缓存: 安装过的包本地缓存,不会进行远程安装

\* 并行下载: 一次下载多个包,npm是并行下载

\* 精准的版本控制:保证每次安装都跟上次安装是一样的版本

Yarn的安装:

1 npm install yarn -g

2 去官网yarn官网下载

https://classic.yarnpkg.com/en/docs/install#windows-stable

//建议第二种,第一种安装方式的yarn 的全局安装方式默认不可用,需要手动配置环境变量path才可用;

### Yarn常用命令:

查看版本： yarn --version / npm -v

初始化: yarn init / npm init

局部安装: yarn add {包名} / npm install i {包名} --save-dev -save -D -S

局部删除: yarn remove {包名} / npm remove {包名}

全局安装: yarn global add {包名} / npm -g install i {包名}

全局删除: yarn global remove{包名} / npm remove {包名} -g

安装依赖(根据package.json): yarn / npm install i

清除缓存: npm cache clean

// npm5也加入了本地缓存,npm6加入了自动保存执行依赖 -S

查看已经安装的包名 : yarn list

查看工具包的信息： yarn info {包名}

# Npm后半部分:

### npm配置命令别名并执行:

//编写一个简单的http服务;



以往我们执行都是通过node 文件.js 此命令来执行文件

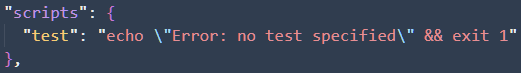


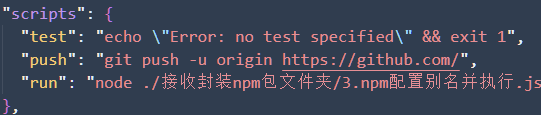
还有另外一种方式可以执行文件(使用npm别名命令)

Package.JSON文件下 “scripts” 属性:

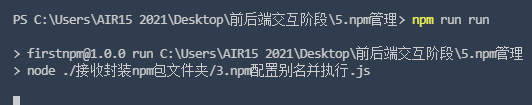
Script是一个对象,里面的键值对就是命令别名和命令本身;

可以添加键值对,给一些复杂的命令起一个简单的别名



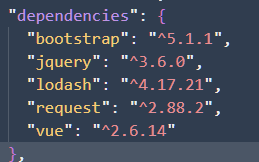


执行方法: npm run {命令别名}



### 版本号(了解):

Package.json中版本号:



格式:

^3.6.0 : 锁定主版本,以后安装包,包版本为3.x.x , x默认取最新

~3.2.0: 锁定小版本,以后安装包,包版本为3.2.x , x默认取最新

3.2.3:锁定完整版本,以后安装包,版本必须是3.1.1

安装指定版本

yarn add [jquery@1.11.2](mailto:jquery@1.11.2)

清除本地缓存

npm cache clean

安装工具包出现疯狂报错可以清除缓存;

## NodeJS 模块化

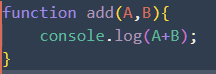
模块化就是大的功能拆分成一个个小的模块;通过不同模块的组合实现一个大功能

\* 在node中一个js文件就是一个模块

\*模块js是一个黑箱就意味着,在不暴露的情况下,一个模块(js文件)无法读取另一个模块(js)文件的内容;

自定义模块的暴露

步骤一: 在目标模块中正常编写代码



步骤二: 在模块中暴露

语法: module.exports = 内容

步骤三: 引入暴露的模块 require

语法: const 变量名 = require(模块路径);



注意:

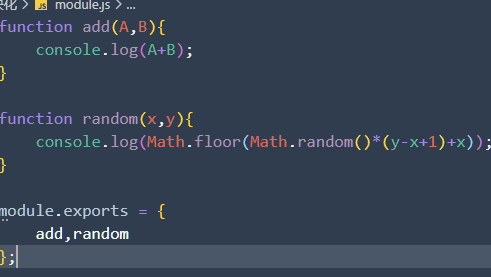
1模块路径./不能省略,省略js会认为是引入工具包和内置模块

2 module.export暴露的内容就是require的返回值;

底层函数内容: return module.exports ;

module.exports可以暴露任何内容(对象,变量等)

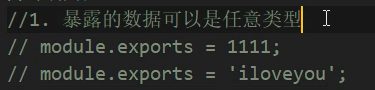
所以可以将需要暴露的内容封装成一个对象



暴露的注意点:

module.exports 可以暴露任意类型数据;一般暴露 引用类型(数组 对象 函数)

module.exports 多次赋值会覆盖(和变量一致)



第二种暴露语法: exports

// 语法 exports.属性名 = 暴露内容

// export 会将所以暴露内容打包成一个对象 返回给 导入模块;

// 所以export.a = b \*n   ~~~ modules.export();

底层原理:

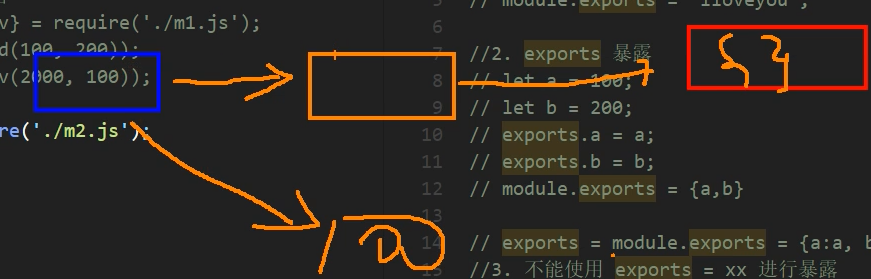
export = module.exports = {};

export就是模组暴露形成的对象;

注意事项：

语法层面上,不能使用exports = 内容 这种方式暴露

(把exports 的指向改变了,不再指向module.export这个暴露方法了)



###### 引入require;

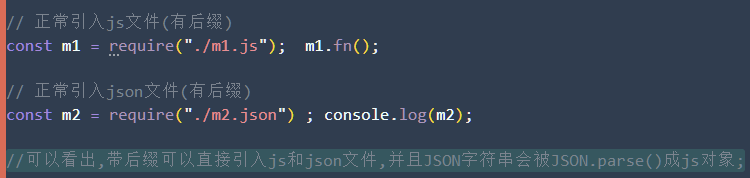
模块不仅仅只js文件,还可以是json文件和.node文件或其他文件

JSON文件: 格式为JSON的纯字符串文件

.node文件: C/C++编写的扩展文件,通过dlopen方法编译;(了解)

###### //正常引入

Json文件不需要暴露export,直接require引入即可;





//正常引入只支持js json node后缀,其他后缀统统以js文件解析;



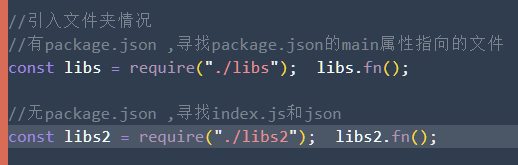


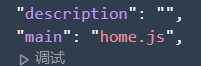
//引入的路径为文件夹的情况

\*会寻找文件中package.json种main属性指向的文件

\*若package.json种main属性指向的文件 或package.json文件不存在

会自动寻找index.js或者index.json

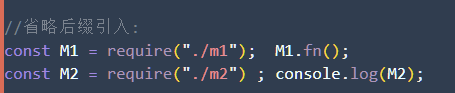


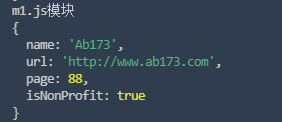




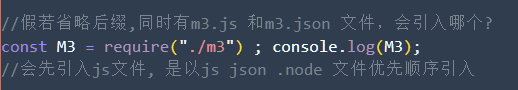
//省略后缀引入:

不写后缀引入js和json文件:





//若省略后缀有多个重名文件,则以JS JSON NODE文件的顺序引入

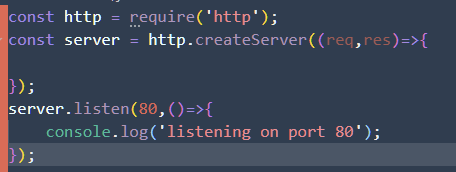
 

### http 服务的拆分:

http服务启动三部曲:可以利用暴露和引入模块化

主要是模块化的思想;

三部曲启动http服务:



可以将createServer的回调函数和listen的回调函数模块化,卸载另一个js文件中,引入即可,这样子就非常直观了,而且修改不用修改index文件,只需要修改模块文件皆可

### Require导入内置模块

//直接require(模块名)

Require(“fs”);

Require(“http”);

Require(“url”);

Require(“querystring”);

### Require导入第三方npm包

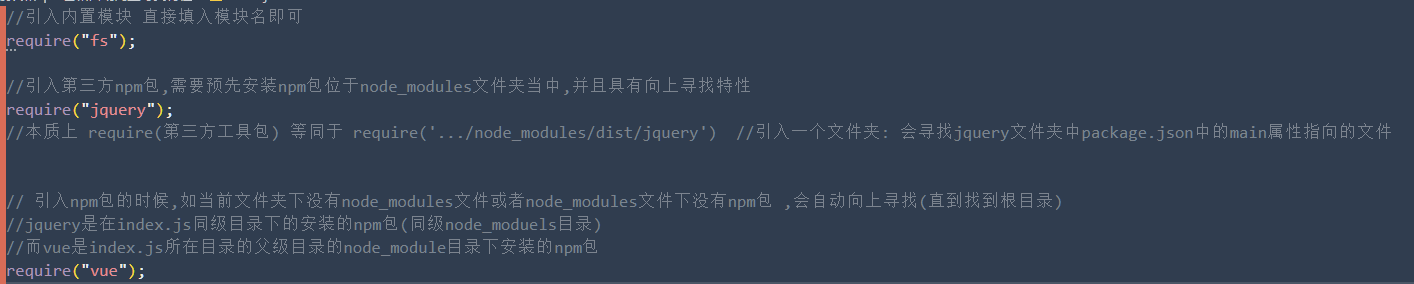
//需要npm包已经安装在node\_ modules文件夹当中

Require(“模块名”)

require(“jquery”)

//引入第三方包实际上就是引入node\_module/目录下的文件夹

//并且node\_module支持向上寻找



#### Require方法的执行过程:

1 读取目标文件的内容

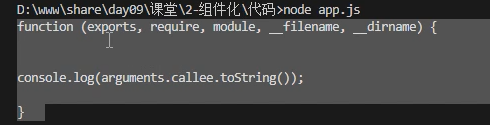
2 执行目标文件内的代码

3 返回module.export 的值

具体执行过程:

Require(“A.js”),读取A.js的内容,并将内容封装成一个匿名函数直接自调用,然后返回module.exports 的值,而该匿名函数有一个参数module用来专门接收module.exports的值,然后将参数module直接当作require方法的返回值;

//在模组js中 加入命令 argument.callee.toString() 来查看模组的调用者是谁



Npm-require 小总结:

