**终端的基本使用**

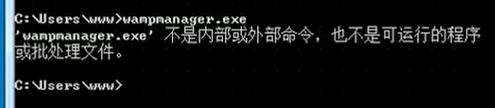
**打开应用**

* notepad 打开记事本 note译笔记 pad设备 notepad译笔记本
* mspaint 打开画图 mspaint译画图板
* calc 打开计算器 cala译计算
* write 写字板 write译写字 写入
* sysdm.cpl 打开环境变量设置窗口 最后这个字母不是i是小写的l(打开后 点高级—环境变量—系统变量—path—就可以加一些环境变量了 除此之外选计算机右键—属性--高级系统设置—环境变量 效果一样的不过命令方更快捷式

环境变量的作用?

在cmd窗口可以启动操作系统当中的应用软件

比如之前安装了wamp 输入命令显示如下

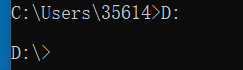


**出现此提示的原因:**

找不到这个文件,为什么会找不到?因为它是在当前目录找的 此目录是没有这个文件的,所以要进入安装目录去运行就能找到 如果手动点文件夹进到对应目录 那就和操作终端没关系了

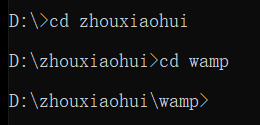
**如何通过终端命令行操作?**

a).首先进入对应盘 切换盘符 D: (D加上冒号再回车 字母大小写均可)这样就进到了D盘



b).如果又想切回去 就C:又回去了

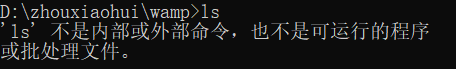
c).进入到D盘后通过cd命令进入到目录



cd命令除了./ 或者.. 还可以直接写其他毫无关联的路径 直接切换到任意路径 不需要一层层走



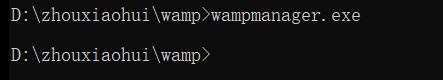
d).进来后直接ls查看当前目录清单(看当前目录文件和文件夹)



**会发现报错,原因是:**

实际上ls是Linux的命令，Windows本身是没有这条命令的（对应Windows上的命令是dir），所以执行会报错。 可以直接输入命令dir等效,也可自定义ls命令实际执行dir,具体自定义操作看csdn博文

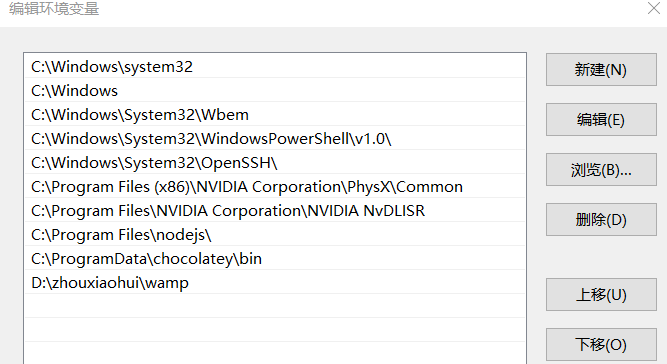
e).进入到文件夹后再输命令就可以启动应用了



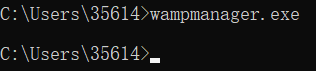
**但是这样做依然比较麻烦 还要切换目录 不方便 那么该怎么做?**

a).把此目录加到环境变量中 这样windows操作系统就知道去这里找此目录

b).进入到文件中把文件路径D:\zhouxiaohui\wamp复制下来,通过前面说的找到环境变量—path,将此目录加到环境变量中 新建—确定 然后重新启动终端窗口



然后可以正常使用(不重启窗口就不行)



综上:环境变量的作用是帮助我们通过命令行的方式去找到要执行的指令

### 常用命令

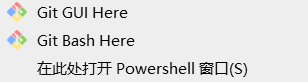
* md 创建目录
* rmdir(rd) 删除目录，目录内没有文档。
* echo >a.txt 创建空文件
* del 文件名 删除文件
* rm 文件名 删除文件
* cat 文件名 查看文件内容
* cat > 文件名 向文件中写上内容。

常用命令一般是目录和文件的基本操作

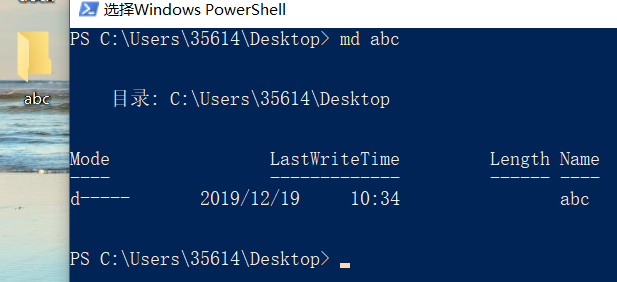
**1. md 创建目录**

Make Directory的缩写 make制造 directory目录

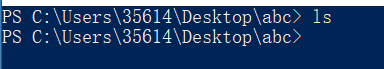
**在桌面按住shift点右键 选择在此处打开窗口 这样当前目录就是桌面**

****

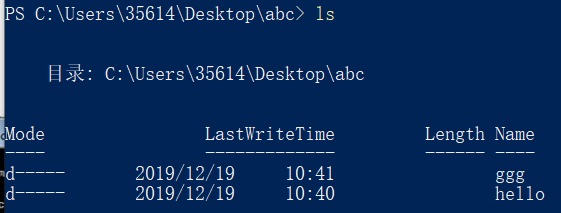
执行md 文件名 如图创建了abc目录



初次创建后 cd abc进到目录 再ls命令会发现里面什么都没有



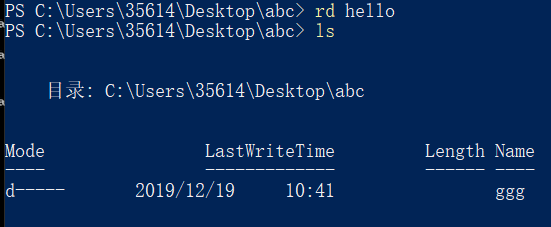
那就再md hello和md ggg后ls可以看到里面有hello目录 图右下角



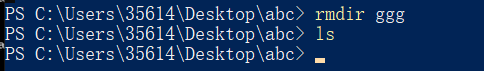
1. **rmdir(rd) 删除目录，目录内没有文档。**

2.1 有两个命令rmdir和rd 两个都是remove directory的缩写

例rd



例rmdir

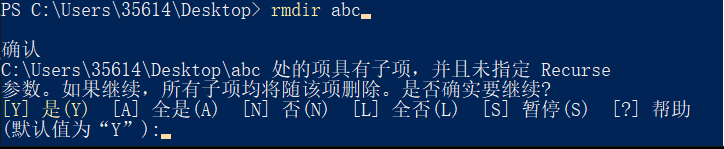


综上rmdir和rd等效

2.2 之前是cd ../进到上层目录 实际cd ..也可以 两个点进到上层目录

2.3 当abc文件内有子文档或子目录 那cd ..后退到abc上级目录可以直接删abc么?

方法一:

****

选择y后 abc就删除了

方法二:(通过桌面shift 右键打开命令运行此指令会报错如下图 如果window+R的命令工具操作 就不会报错)

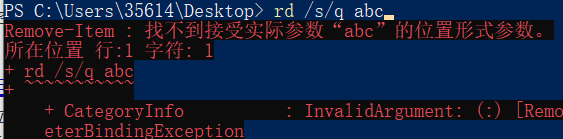
rd /s/q 文件名称 例rd /s/q abc

/s表删除子目录 /q表安静的删除不给任何提示直接删掉

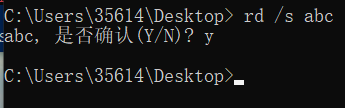
s是subdirectory的简写 sub副 子 分支 subdirectory表示子目录

q是quiet缩写 译安静

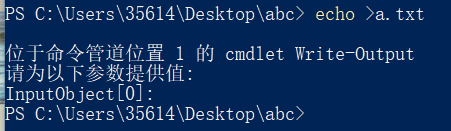
例:桌面打开的命令工具



例:cmd打开的 若不加/q会提示确认 加了就不会

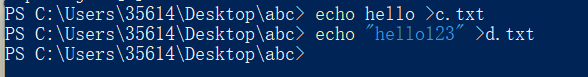
****

1. **echo >a.txt 创建空文件**

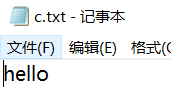


echo >文件名.后缀 创建空文件 提示提供参数值直接ctrl+c即可

a).echo创建有内容的文件



a.1) echo hello >c.txt 这里的hello是文件内的内容



a.2) .内容也可以加引号 等效

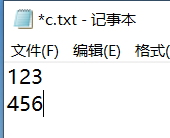
b).追加内容(一个>覆盖 两个>>追加)

echo 内容 >a.txt如果遇到同名的文件 会直接新内容覆盖旧内容

那如果想追加呢? 写两个>>大于号



换行追加



c). cat

作用一:查看文件内容 指令: cat 文件名

cat是concatenate的简写译连接, cat命令的用途是连接文件或标准输入并打印。这个命令常用来显示文件内容



作用二:向文件写内容(报错 正确命令自行百度)

cat >c.txt

前面我们提到echo 123 >c.txt这种可以向文件写内容 其实cat也可以向文件写内容

cat c.txt表查看文件内容

cat >c.txt表向文件写内容 输入此指令后 有竖线一直闪烁提示我们直接写内容 回车表换行继续写 如果想退出直接ctrl+c

此命令会覆盖原内容 将>改成>>表示追加内容 不覆盖

d).删除文件(注意文件和文件夹(目录)是两个概念) 指令二选一

指令: rm 文件名

指令: del 文件名





**node.js环境安装配置**

## Node.js开发环境准备

1. **普通安装方式**[**官方网站**](https://nodejs.org/zh-cn/)**(推荐第二种多版本安装)**

进入网站后会看到两个版本



左边的相对比较稳定,变动小,一般有变动也是修复一些bug或者说对个别特性做细微调整

右边的变动稍微有点大,有可能会有一些bug,新特性的变更一般也会优先在此版本进行更新,左侧的是不会有大变动

**综上**真正在工作项目中一般用左边的版本 相对比较稳定 点击其他下载会看到有针对不同平台的

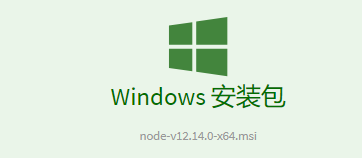


1.1看图知道有32位和64位的 windows下有

* 1. a).安装包(.msi)这个比较简单 下载后安装一直点下一步就行

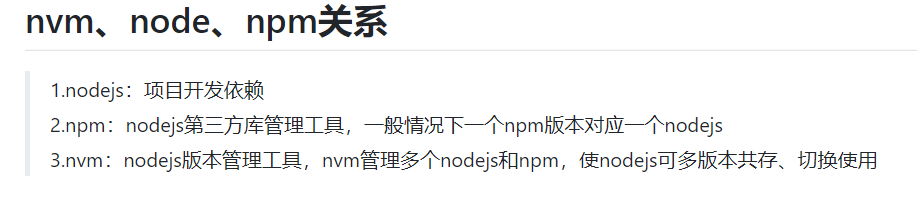
b)二进制文件(.zip)这是压缩包 下载后要解压 还要配置环境变量 比较麻烦 所以一般选a)中的.msi

1.3不过更方便的是直接点



这是根据操作系统和版本直接推荐的下载这个 并安装即可 因为简单所以操作流程就不写了

1. **多版本安装方式(推荐这个)**



**为什么要维护node.js的多个版本呢?**

**因为:**

一般我们在做开发的时候用的版本相对稳定些,如果有新的特性变更我们想测试一下的话,有可能需要变更node.js的版本环境,

有人会说把原来的版本卸载,重新装个新版本,但是这样我们原来的环境就丢失了,会不太方便也很麻烦

为了解决这个问题就有第三方包,可以帮我们管理多个版本的node.js所以我们就用这个方式,这样方便后期做不同版本的切换 同时测试一些新特性

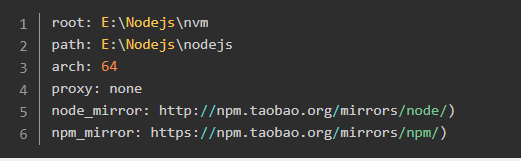
**安装步骤:**

**参考教程**

<https://msd.misuland.com/pd/2884250103256715670>完全照此操作即可

<https://www.jianshu.com/p/9226ae1a343a> 这个也可以

* + 卸载已有的Node.js(一定要完全卸载)
  + 下载[nvm](https://github.com/coreybutler/nvm-windows)(node.js版本管理工具)
  + 其他步骤看教程链接



root是nvm的根路径

path是nodejs的根路径

arch表示计算机多少位 这里64位

proxy没什么用可不管

node\_mirror和npm\_mirror修改为对应的国内淘宝镜像目录。

## 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 nvm I latest表安装最新版变动大的版本
* 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

使用哪个版本的nodejs 那么命令node -v显示的就是使用的nodejs版本

下载了nvm使用很简单 一般我们都只使用use这一个命令 顶多再加上list命令查看当前所用版本

## Node.js 运行的两种方式

* 命令行方式REPL
* 运行文件方式
* 全局对象概览

**一:命令行方式REPL**

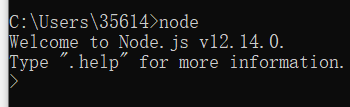
现在开始基于nodejs的环境来学习nodejs,一般我们学习任何一个技术点都是从它最简单的hello world小程序开始

要想写程序 必须先要知道程序怎么执行 对于nodejs来说 它的执行方式和之前我们在浏览器当中写代码是类似的

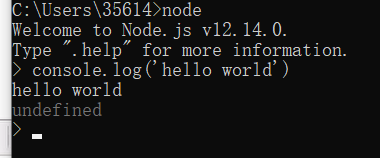
回顾我们之前写js代码 最简单的是在编译器console.log(‘hello world’)打印一个‘hello world’ 这段代码直接在浏览器打开—控制台—看到输出的内容 这是最简单的一行代码

这行代码除了在编译器写 还可直接在控制台写 这里的执行环境就是浏览器自己提供的(只要是符合js语法规范的都可以执行) 写到编译器中又是另一种执行方式 这是我们之前的

在node中也有一个类似环境 进入cmd窗口 输入命令node 回车 显示欢迎来到node



此时这个环境和浏览器当中console.log()环境是类似的



和浏览器控制台显示的一样 undefined表示没有返回值 此时node后就进入到nodejs当中 代码的执行环境 这个环境有一个特定称谓叫REPL

REPL是四个单词缩写 read-eval-print-loop 读取代码-执行-打印结果-循环过程

read读 读取屏幕当中书写的代码

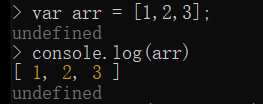
eval执行 即把书写的代码解析并执行

print打印 执行后把结果打印出来

loop循环 循环此过程

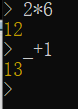
此黑屏执行环境就是这么一种说法

类似声明变量这些也都能操作



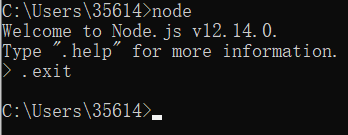
除此之外它还有个小特性 执行1+1会直接计算输出2 这个和浏览器也是一样的

特别的地方是英文下划线+1输出13 这里下划线表示上次输入的结果(最后一次的结果) 此特性浏览器没有



那么怎么退出node环境 直接.exit 回车 注意必须要点. 别漏了

exit译出口

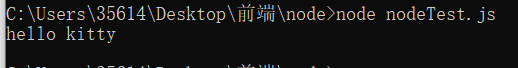


**二:运行文件方式**

不用先node 直接node 文件名就行

除了上面的方式 我们也可以将代码写到文件中 然后执行此文件 从而执行里面的代码

例单独创建一js文件 里面写着console.log()代码 在命令行进到此目录—输入命令 node 文件名 最后可看到执行成功



这是nodejs代码的另外一种执行方式

对于命令行直接通过node的这种方式比较简单 一般是测试一些小特性 但是一般情况做项目或者做案例的时候都是通过文件方式来做的

以上就是nodejs的代码两种执行方式,一种命令行 一种文件

## node模块化 服务器端模块化

**一:初识模块化**

1. 接下来具体看nodejs当中相关的内容点

对于浏览器来说 全局当中提供了很多的api 比如内置的对象 全局的函数都是可以直接使用的

其实nodejs当中 全局范围内也提供了很多类似的一些成员 但是nodejs当中不存在window这个对象 这是和浏览器很大的区别 但是window当中有一个类似的概念 名字不叫window 叫global译全局

在nodejs官网文档 里面提供了nodejs开发环境所提供的很多相关的模块里面包含大量api 后续会对核心模块的常用api进行学习

2.node中的全局对象 global 上面有大量的属性和方法

**全局成员概述**(先初步认识 后续会详细说明)即global上的属性和方法 这些都属于global global全局对象书写的时候可以省略

**2.1**

\_\_dirname 文件路径(不包含文件名) 书写是两个下划线

\_\_filename 包含文件名的全路径

后续做文件操作基本会用到这个文件名 因为这个路径要加上全路径之后才方便处理相关的文件内容 才能方便查找文件从而进行读写操作

**2.2**

* [clearImmediate(immediateObject)](http://nodejs.cn/api/globals.html#globals_clearimmediate_immediateobject)
* [clearInterval(intervalObject)](http://nodejs.cn/api/globals.html#globals_clearinterval_intervalobject)
* [clearTimeout(timeoutObject)](http://nodejs.cn/api/globals.html#globals_cleartimeout_timeoutobject)
* [setImmediate(callback[, ...args])](http://nodejs.cn/api/globals.html#globals_setimmediate_callback_args)
* [setInterval(callback, delay[, ...args])](http://nodejs.cn/api/globals.html#globals_setinterval_callback_delay_args)
* [setTimeout(callback, delay[, ...args])](http://nodejs.cn/api/globals.html#globals_settimeout_callback_delay_args)

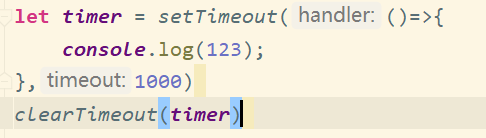
这个其实都是一对一对的 像setTimeout clearTimeout 一个开启定时 一个清除

和浏览器类似 在全局当中都有定时的两个方法 但是除此之外nodejs还提供了setImmediate

immediate 译即时 立刻 /ɪˈmiːdiət/

它与定时方法作用类似 但是有一些细节上的差异 要想理解细节上的差异后面会分析nodejs中的事件模型 其实这个模型和浏览器中的事件模型有相似性 等我们理解了底层原理后两者的区别就会自然而然清晰了

这里先看定时的函数 和之前用法差不多



**2.3**

[Buffer 类](http://nodejs.cn/api/globals.html#globals_class_buffer)这是一个构造函数

# buffer译缓冲

后面文件操作的时候再详细分析 这里先记住它是一个全局的成员

2.4

2.4.1 console打印

2.4.2 还有exports module require()这三个都属于node的核心 是模块化开发中模块化相关规范提供的对象 他们的关系后面详细说明 这三个都和模块化相关是全局上的

2.4.3 process 译过程进程 是一个对象 对象中有很多相关的属性

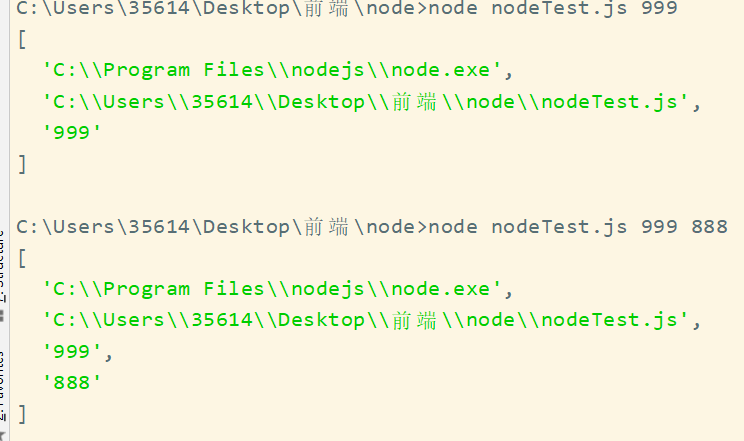
这里先用一个常用的process.argv 实际上是arguments参数

总结:process.argv中argv是一个数组 默认情况下前两项第一个是node执行环境的路径(nodejs安装环境的全路径 这个环境指的是node.exe 因为是用node.exe来执行node相关代码),第二个是当前执行的js文件全路径(即含文件名) 如果命令行有传参 那从第三个参数开始表示命令行参数



process.argv中argv作用主要是接收命令行的参数 从第三个参数开始表示命令行参数

例 执行文件时 在对应命令后 空格然后加参数 会发现打印argv时 参数会放在数组中一起打印出来 若多个参数则每个参数空格隔开 先输入的参数先放到数组中 位置靠前 依次类推 最终结果是数组列表



除此之外 process里面还包括进程 也就是nodejs在执行的过程中涉及到的一些环境信息

比如process.arch表示当前操作系统的架构(64或32位)是多少位的也就是操作系统版本位数

64位是x64 如果是32位就是x86





arch缩写 [**architecture**](javascript:;) 译架构

注意ture别写成true了

**二:模块成员导出详解**

对全局成员有了初步认识后,接下来进入的这个知识点很关键,那就是模块化开发,其实nodejs所有的代码都是构建在模块化基础上的(在模块化的基础上写nodejs代码),所以这部分是核心 是非常关键的内容 对于模块化来说全局中有三个重要的属性与此相关就是module exports require()

**1. 之前前端浏览器中也涉及到模块化 这里进行简单梳理**

为什么要进行模块化开发?

传统非模块化开发有如下缺点:

a).命名冲突

因为在大型项目团队合作中多人可能用同样的名字给变量命名 这样就会造成命名冲突 从而导致某同名代码 被另一个覆盖掉

b).文件依赖

对于前端来说经常可以看到加载很多js文件且一个文件依赖另一个文件 且引入有先后顺序 不能乱否则报错 这个文件依赖的列表如果要程序员自己维护就特别麻烦 尤其是项目不断增大 有人员变更的时候 新来的成员对之前代码的依赖关系不清楚 某天他接到需求是之前的项目已经快完成了 只需要引入js文件 但是他引入后发现缺少某些变量或某些函数未定义 这种情况代表它引入的文件还依赖于别的文件 然后它引入了另外两个文件 可还是报错 因为另外两个文件还依赖别的文件 这种问题就非常麻烦 后来为了解决这个问题很多大牛经过尝试就诞生了模块化开发 这个模块化开发产生的过程中很多人做了尝试在公司中有自己的解决方案 但是不同项目中他们的解决方案虽然能满足自己的需求 却彼此间很难进行代码重用 后来为了统一规则就产生了前端标准的模块化规范

npm安装的包在node\_modules里 可以看到包里面有很多文件夹 是因为此包还依赖很多其他包 将依赖包也安装了 这样就不需要程序员自己维护依赖关系

前端标准的模块化规范

之前我们接触的是前端的 早期接触的有两种

AMD 对应的实现是 requirejs

CMD 对应实现是 seajs 这是阿里巴巴的产品

有了规范后如果一个项目基于模块化框架来做开发 写代码的时候就会非常统一 模块化规范就定义了写代码的规则 只要遵循这个标准 写出来的代码一般风格都是一致的 并且模块化方案解决了上述存在的两个问题

2.服务端模块化

服务端(后端 服务器端)模块化方案和前端模块化是有差异的

commonJS 对应的实现 node.js

他们的不同之处是,和前端模块化规范相比本质上来说差别不大 主要在加载文件方面 对于前端模块化来说它加载文件 需要从服务器经过网络传输到浏览器端

但是服务端模块的文件不需要这样 因为它文件本身就在服务端 只需要将磁盘的文件加载到内存中即可 所以服务端文件的加载是同步的 而前端文件的加载是异步的 毕竟要保证页面渲染的效率

但是在模块的定义 导入 导出 它们之间非常相似

1. **nodejs中模块化具体规则**

**a).如何定义模块:**

对于单文件来说,一个js文件就是一个模块,模块内的成员(声明的变量 函数)是相互独立的 相互独立指此成员和其他模块的同名成员没关系 除非此模块将成员暴露出 否则其他模块不能访问此模块的成员 由此也可看出解决了全局污染

**b).模块成员的导出和引入**

假如a文件导出模块导出的是一个方法 方法内打印变量c(在a文件已定义变量c为1) b文件引入模块后执行方法(在b文件定义c为2) 最后得出的结果是1

虽然方法导出了 我们也引入了 但是执行相当于导出前的文件里执行的 拿的变量都是拿的那文件里的 不会拿现在的文件里已有的东西 模块执行还是相当于在原来的文件里面执行 拿的是原来里面的东西 除非通过方法传入东西进去 否则模块拿不到外面的东西 **模块是相互独立的** 有自己的空间

正常情况函数在哪执行 拿的就是所处环境的东西 但是模块不一样 还是在原来的文件拿(给方法传参例外)

**综上:模块化从代码层面主要是这两个方面**

**c).具体说明**

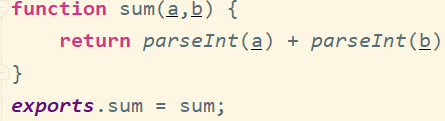
**定义模块?**

写一个js文件 这个js文件就是一个模块

**方式一:导出模块成员 和引入 (配套)**

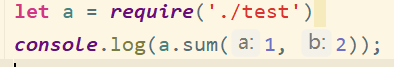
导出:

exports.a = a 表示将成员a导出



引入:

let module = require(‘从当前文件目录出发 开始写路径’)

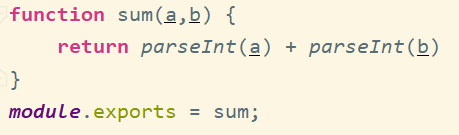


如果不手动导出 直接引入文件 打印接收的变量 显示空对象 这是因为默认导出的exports 而此exports是一个空对象 空对象上加属性 就是导出一个有属性的对象

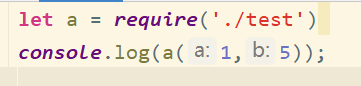
相当于直接导出一个对象 对象里有sum方法 所以使用需要点对象

**方式二:导出和引入**

导出 这样直接导出的是方法本身



导入



**三:模块化细节补充**

前面基本了解了模块的定义和导入导出,这里再补充下模块化的细节

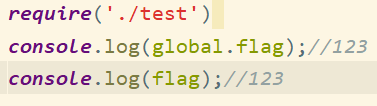
1. **模块的导出除了exports和module.exports还有第三种方式** 不过第三种仅做了解不推荐

就是将成员挂载到global全局对象上 另一个模块中引入文件后直接可以在全局中访问

导出:



导入:



1. **需要记 模块内部实现机制**

多次导入同一个模块只会加载一次,这是nodejs内部实现的机制

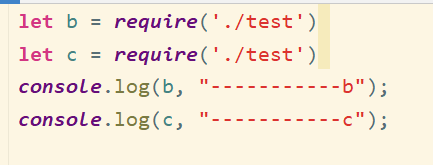
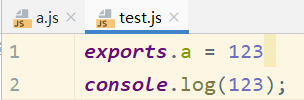
已经加载的模块会被缓存,当加载相同模块时,若内存中已经有缓存了就不会再加载 而是直接用第一次的模块 节省了加载的时间 从而达到提高性能的作用

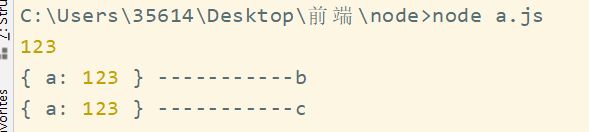
**这里怎么知道内存中已经存在此模块呢?**

实际上是根据模块路径判断的 若路径相同则模块相同

当多次require 时并未再次加载 而是直接用的第一次的内容 这样节省了加载的时间

例





1. **模块文件的后缀通常有3种情况**

**.js** 普通js文件

**.json** saon文件主要提供数据的 不用导出 直接引入即可 引入的数据本身可直接使用

**.node** 这个比较少见 是用C语言和c++写的模块 编译好的二进制文件 这是一个简单的二进制模块

引入的时候变量是个对象 东西都在对象上 需要点它 和exports导入后的使用类似

1. **不写后缀 写同名时 文件加载优先级**

require时导入的文件路径 后缀例.js .json .node这些后缀可以省略 但是当这三种文件都有 且不写后缀又同名的情况下 默认按.js .json .node顺序 从左到右优先加载 左边最优

1. **模块分类**

5,1自定义模块

指的我们自己定义的模块

5.2系统核心模块

除了自定义模块 nodejs平台本身也给我们提供了很多系统核心模块 或者叫 核心模块

1. fs文件操作
2. http网络操作
3. path路径操作
4. querystring1查询参数解析
5. url解析

例require(“fs”)这里的fs不是路径 而是特征的写法 使用都是require一个特征的东西 如果是path模块 就是require(“path”) 不同模块 字符串不一样这是固定格式

后续再详细说 这里只需了解除了自定义模块还有大量系统核心模块供我们使用

**node基本操作**

**Buffer**

Buffer译缓冲 Buffer是一个类class

Buffer对象是Node处理二进制数据的一个接口。它属于全局对象global 是全局成员，可以直接使用，不需要require(‘buffer’)。

* **对于Buffer来说api可以分成三类**
  + 1类: 构造方法(就是构造函数 不过在es6中更确切叫类)
  + 2类: 静态方法
  + 3类: 实例方法

**一: Buffer实例化**

Buffer不是核心模块 而是全局对象global上的. 就是一个全局成员

Buffer是处理二进制数据的 而我们之前对js的相关操作主要是处理字符串 js处理二进制不太方便 所以node提供了处理二进制更有效的办法 提供了专门的Buffer

Buffer本质上就是字节数组 里面存储的一个一个的字节 我们知道计算机存储的数据都是二进制的 那么Buffer二进制位就形成一个字节 Buffer中都是由一系列字节组成的

首先看Buffer怎么用?

要想使用Buffer需要先实例化 产生一个Buffer的实例 才能往里存储数据

一:new的方式已废弃 请使用等效方式 这里可以了解下废弃前的操作

实例化方式按照之前理解是通过构造函数或类来实例化 早期的nodejs可以通过new的方式来实例化

比如有几种new的方式 不同new的方式传参不同

**(**new Buffer(size) 传入一个数值 产生几个字节的buffer实例对象 数值代表数值个字节

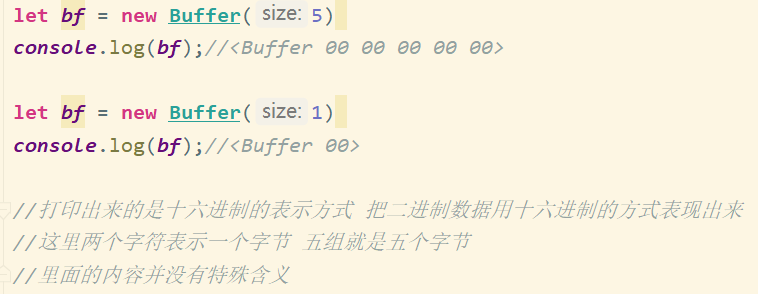
new Buffer(arr) 传入数组

new Buffer(buffer) 传入另外一个buffer

new Buffer(string[,encoding]) 传入字符串

[new Buffer(arrayBuffer[, byteOffset[, length]])](http://nodejs.cn/api/buffer.html#buffer_new_buffer_arraybuffer_byteoffset_length) 传入数组形式的Buffer

例:

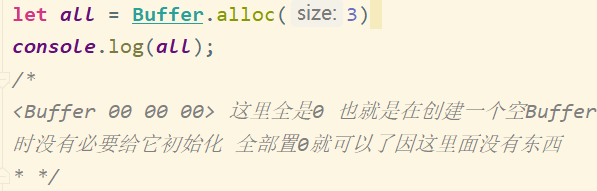
**)**

**二: 静态方法 关键使用这个来实例化**

* 实例化 常用的三种方式
  + Buffer.from(array)
  + Buffer.from(string)
  + Buffer.alloc(size)

**1.Buffer.alloc(size)** 返回一个指定字节长度的 Buffer就是Buffer实例字节长度byteLength 例传5 表分配5字节 后续说Buffer实例直接说为Buffer

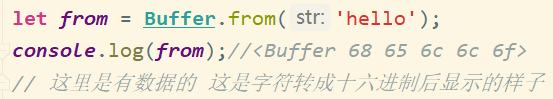
推荐的做法是Buffer.alloc(size) 这是静态方法 alloc译分配 分配内存



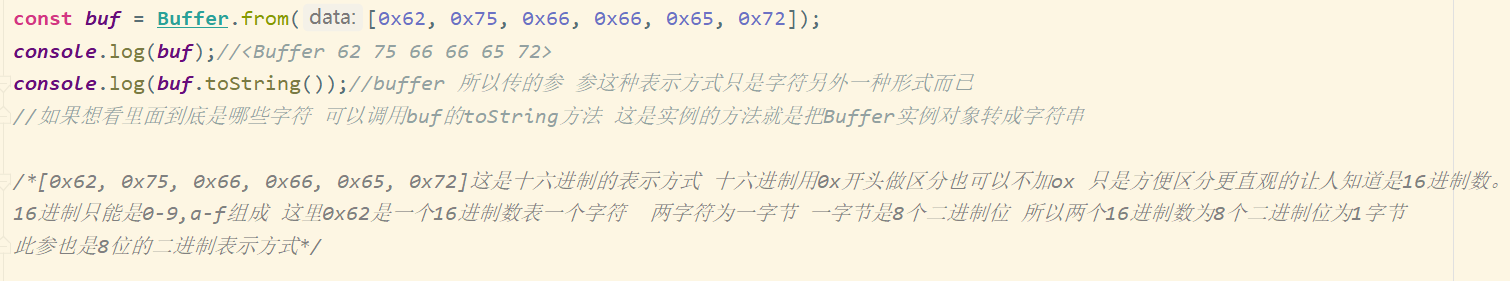
new Buffer()实例化的方式被启用了 所以严格来说Buffer.alloc()这才是第一种Buffer实例化的方式

**2.Buffer.from(string)/Buffer.from(arr)**

例1.传入字符串



例2:传入数组



在16进制中一个字符是一个字节 两字符也是一字节 例A 和AA 字节相同

UTF-8编码：一个英文字符等于一个字节(反正我没看懂 看看就行了别深究)

如果想看传入的是哪些字符 可以调用实例的toString方法 将实例转成字符串

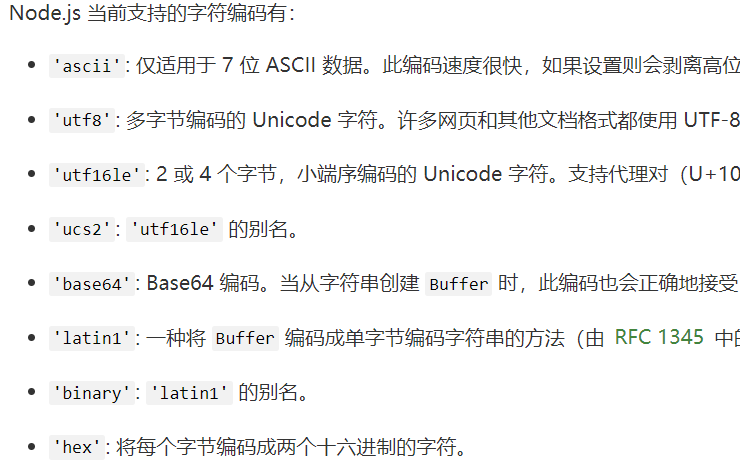
**二:Buffer静态方法**

熟悉了Buffer实例化后 再了解下静态方法中可以实现特定功能的api

* 功能方法
  + Buffer.isEncoding() 判断是否支持该编码
  + Buffer.isBuffer() 判断是否为Buffer实例
  + Buffer.byteLength() 返回指定编码的字节长度，默认utf8
  + Buffer.concat() 将一组Buffer对象合并为一个Buffer对象

判断的都是is

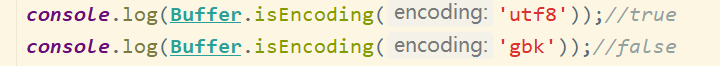
1. **Buffer.isEncoding() 判断是否支持该编码 常用的是utf8**



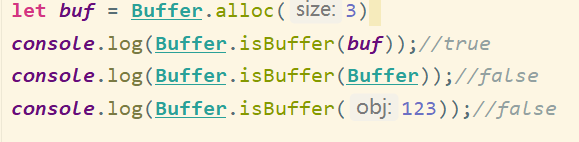
编码在实例化的时候也可以指定(不指定时默认utf8编码,不同编码输出结果不同)



使用



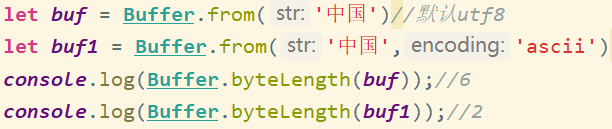
**2. Buffer.isBuffer() 判断是否为Buffer实例**



**3.Buffer.byteLength() 返回内部对应编码的字节长度，**

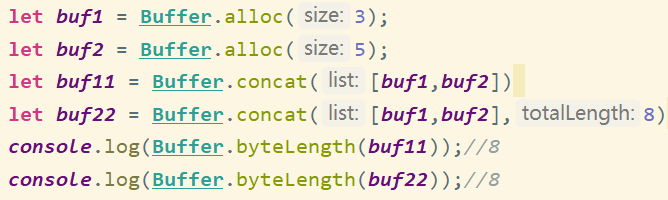
默认utf8 因Buffer实例化不指定编码默认就是utf8

Buffer里面存的东西和编码有关系 不同编码字节长度不同 返回对应内部编码字节长度

****

1. **Buffer.concat() 拼接Buffer 将一组Buffer对象合并为一个Buffer对象**

**实际作用主要是拼接字符串**



参数一是数组 数组里是多个Buffer ,参数二可选 是合并后的数组总长度 若不传会导致执行额外的循环用于计算 总长度 如果已知长度可传 如果不确定则不传 因为如果合并后的长度大于指定的长度 则结果会截止到指定的长度是指定的长度

**let *buf1*** = **Buffer**.from('he');  
**let *buf2*** = **Buffer**.from('llo');  
**let *buf11*** = **Buffer**.concat([***buf1***,***buf2***])  
***console***.log(***buf11***.toString());*//hello*

**三:Buffer实例方法**

* 实例方法
  + write() 向buffer对象中写入内容
  + slice() 截取新的buffer对象
  + toString() 把buf对象转成字符串
  + toJson() 把buf对象转成json形式的字符串

to:译到某状态

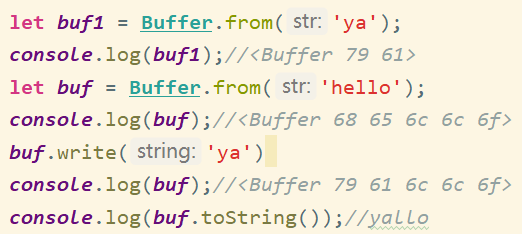
1. **write() 向buffer对象中写入内容**

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

* string要写入的字符串内容 需为字符串
* offset从哪个字节位置开始写 默认0 下标0 1 2
* length 表示写入的字节数量(也叫长度) 默认可写入空间(个数数量 长度)的范围是buf.length-offset 自定义范围是0到buf.length-offset
* encoding 指定的编码 默认utf8 一般不指定

返回值是被写入字节的长度

新写入的会把对应位置的初始值覆盖掉 若实例化传的’hello’则长度5 hello也是初始内容 写入的内容超过5会被截掉



1. **slice() 截取新的buffer对象 和数组的slice类似不过之前是截取数组**

buf.slice([start[, end]])是位置不是长度

返回的是新的buffer 本质不同 和原来的没关系

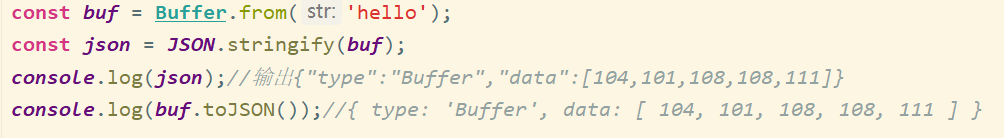
若什么都不传 默认截取整个buffer 那也是新Buffer和原来的不是同一个 是两个不同的Buffer对象

* start [<integer>](http://nodejs.cn/s/SXbo1v) 新 Buffer 开始的位置。默认值: 0。
* end [<integer>](http://nodejs.cn/s/SXbo1v) 新 Buffer 结束的位置（不包含）。默认值: [buf.length](http://nodejs.cn/s/hn6FjL)。
* 返回: [<Buffer>](http://nodejs.cn/s/6x1hD3)

**3. toString() 把buf对象转成字符串**

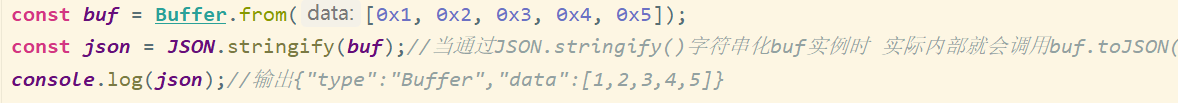
**4.toJson() 把buf对象转成json形式的字符串**

注意:这个方法我们不需要显示调用 当JSON.stringify(buf)内部会自动调用buf.toJSON() 也不要显示调用 因为



虽然JSON.stringify(buf)内部自动调用buf.toJSON()但是可以看到上面结果还是有区别 属性名没引号

所以后续最好用JSON.stringify(buf) 需要记住内部调用的buf.toJSON()即可



type是类型 data是对应的数据 是编码即字符串的另一种表现方式

在编程中JSON四个字母都是大写 Json这种不存在的

若为字符串 则是转成对应十进制的数据

**核心模块API**

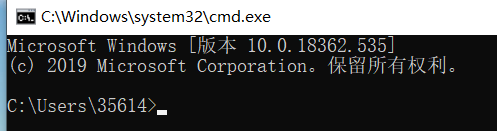
不同的核心模块 require的参数都是固定的

主要是路径和文件相关操作

### 路径操作

### path 模块提供用于处理文件路径和目录路径的实用工具 不是浏览器url路径

* **本质来说就是字符串处理**
* 虽然字符串相关API也可以处理路径 但是路径在**不同操作系统中有差异**如果要手动判断会很麻烦所以提供了专门的模块来解决这问题 比如:
  + - linux下的路径分割符是斜杠 /
    - windows下的路径分隔符是反斜杠 \



路径基本操作API使用:

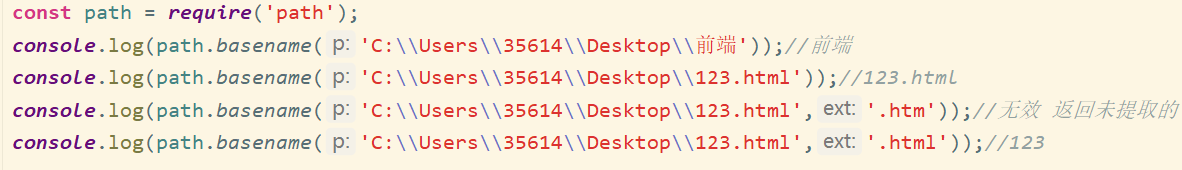
## path.basename(path[, ext]) 返回 path 的最后一部分

base译基底 基础

* path [<string>](http://nodejs.cn/s/9Tw2bK)必须是字符串否则报错
* ext [<string>](http://nodejs.cn/s/9Tw2bK) 提取路径末尾且符合指定string的 若不是末尾就不去掉 作用去掉指定的string

例:去文件后缀 只留文件名

* 返回: [<string>](http://nodejs.cn/s/9Tw2bK)



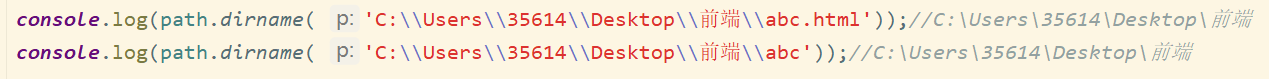
**2.** **path.dirname(path) 方法 返回路径即返回 path 的目录名(全路径去掉文件名)**

* path [<string>](http://nodejs.cn/s/9Tw2bK)
* 返回: [<string>](http://nodejs.cn/s/9Tw2bK)

之前接触的\_\_dirname是获取当前文件路径(不含文件名)



**如果想获取指定路径的目录名(全路径去掉文件名)** 则可以用path.dirname



**注意:如果末尾是目录 也会当做文件不写后缀这种情况 来去掉**

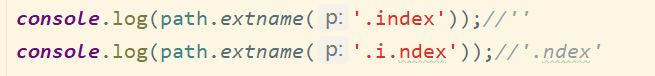
3. **path.extname(path) 获取文件扩展名后缀名(含.点) ext是extension缩写扩展后缀**

path [<string>](http://nodejs.cn/s/9Tw2bK)

返回: [<string>](http://nodejs.cn/s/9Tw2bK)

path.extname() 方法返回 path 的**扩展名**，从最后一次出现 .（句点）字符到 path 最后一部分的字符串结束。 如果在 path 的最后一部分中没有 . ，或者如果 path 的基本名称（参阅 path.basename()）除了第一个字符以外没有 .，则返回空字符串。

如果最后只有.点得到就是点 没有扩展名得到就是空 如果以点开始且后续没点也是空



4.[path.format(pathObject)](http://nodejs.cn/api/path.html#path_path_format_pathobject) 将obj->string 对象转路径

[path.parse(path)](http://nodejs.cn/api/path.html#path_path_parse_path) string->obj 路径转对象

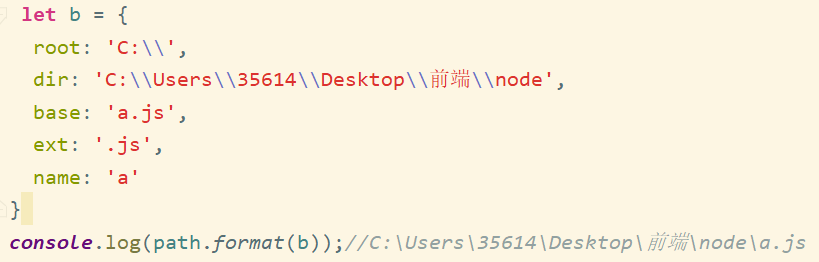
这两个方法是一对 作用相反 这两个实用

**4.1 path.format() 方法 对象转路径** format译格式 格式即最初路径格式是字符串

* 返回: [<string>](http://nodejs.cn/s/9Tw2bK)

当为 pathObject 提供属性时，注意以下组合，其中一些属性优先于另一些属性：

* 如果提供了 pathObject.dir，则忽略 pathObject.root。
* 如果 pathObject.base 存在，则忽略 pathObject.ext 和 pathObject.name。



**4.2 path.parse() 方法 字符串转对象**

* 返回: [<Object>](http://nodejs.cn/s/jzn6Ao)

返回的对象将具有以下属性：



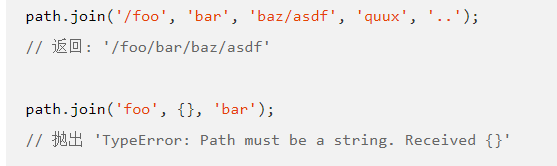
5. path.isAbsolute(path) 方法检测 path 是否为绝对路径。

path [<string>](http://nodejs.cn/s/9Tw2bK) 返回: [<boolean>](http://nodejs.cn/s/jFbvuT)



绝对路径就是c:// 或者/开始(盘符开始的绝对路径见得多一些) 相对路径就是./或者不写开始

6. path.join() 拼接连接路径 可传多个参数 不写斜杠会自动加上



/foo表示根

’ baz/asdf’表baz下的asdf手动加了/不影响

‘..’这个表示上层目录 当前处于quux 指’quux’的上层目录’ asdf’所以刚好走到这里

‘.’一个点表当前路径

‘../../’上走两层 同理

**在拼接的时候会格式化路径即会解析../这类的 所以写的时候也可写../这类**

7. path.normalize(path)会将一个不太合理的路径规范成标准路径格式 normalize译正常化 规范化



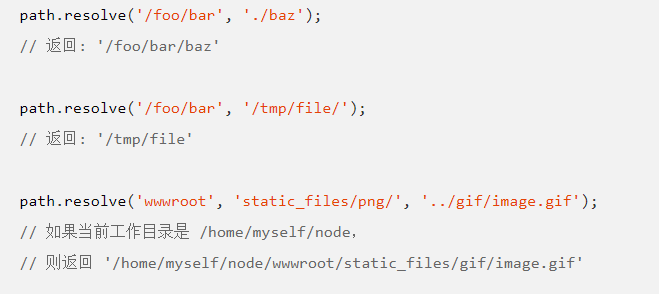
## 8. path.relative(from, to) 方法根据当前工作目录from 返回 from 到 to 的相对路径。因为要../之类的 所以是相对路径 绝对路径直接就写死了 压根不需要from to关系

**是计算两个路径之间的相对路径**

以from为参考点 走到to去 如图可知例1要往上走两层走到orandea再往下走impl/bbb 这就是to结果



9. path.resolve()将路径或路径片段的序列解析为**绝对路径** 这个解析路径方式相当于命令行里面的cd 即先进入到前面这路径 再进 最终形成一个完整的路径



例1是进到此路径 再进到此路径下的baz路径

例2:先进到此路径 但是因为没有./是直接/进到另一个路径 所以和前面路径没关系

cd命令除了./ 或者.. 还可以直接写其他毫无关联的路径 直接切换到任意路径 不需要一层层走



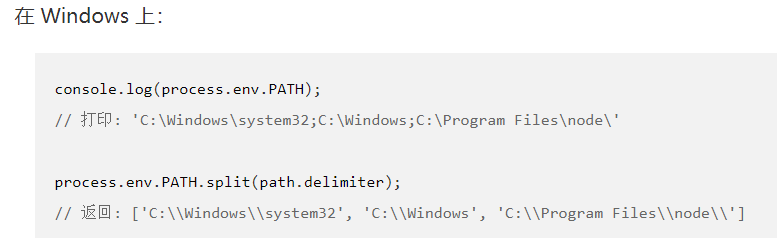
例3:注意第一个是相对路径 所以要先进入到当前路径的这个路径 然后再一一来 最后得出绝对路径

**10.还有path的两个特殊属性**

## 10.1 path.delimiter 环境变量分隔符 delimiter译分隔符 限定器

Windows上是分号;

POSIX上是冒号:



**10.2 path.sep** 路径分隔符是 [separator](javascript:;)缩写 译分隔符

Windows 上是 \

POSIX 上是 /



**异步I/O 和 异步队列事件机制 和 nodejs底层设计思想**

文件操作涉及nodejs设计思想中非常重要的异步**I/O**

文件操作属于**I/O**的一种 平常我们所说的**I/O**就是输入和输出 对于输入和输出来说一个是文件操作一个是网络操作 这也是nodejs开发中经常遇到的两个场景 所以在文件操作之前 需要先把异步**I/O**的思想深入分析

*nodejs*中最常听到的词了**I/O**即指输入和输出，输入*input*和输出*output*，一个地址相对应的就会有一个输入和输出接口，且不同地址的**I/O**也不同。

异步I/O之前在浏览器中也有类似处理 在浏览器中很多场景也是用异步方式 具体方式主要有三种

异步处理 在 node中叫异步I/O input/output缩写

异步处理异步操作 就是异步I/O 只不过读法不同

在node中最常用的操作:

1. 文件操作
2. 网络操作 网络请求和响应过程就是一个I/O过程 即异步过程

异步I/O本质来说操作是异步的 相应在浏览器中也存在异步操作,主要有三种:

1. 定时任务
2. 事件处理函数
3. ajax回调处理

遇到这三种情况在浏览器中会涉及到事件队列(任务队列)这个概念

**总结:**浏览器和node事件模型运行机制大体相同 不过浏览器是遇到以上三种任务扔事件队列 但node是遇到I/O操作扔事件队列 这里的I/O操作指文件读写操作 网络请求和响应操作 又叫文件I/O和网络I/O 我们只需要记住这点就够了 还有些细节的差异不深究

js运行是单线程的,在浏览器中是这样,在nodejs中也这样,都是单线程,但是单线程优缺点就是一旦遇到耗时任务,后续代码都需要等待,所以在单线程基础上又引入了事件队列机制.

之前分析了浏览器的事件模型,在nodejs中也是类似处理方式

**事件模型 事件机制:**

nodejs代码要执行的话,首先需将代码加入到主线程(单线程中)然后才能执行 🡺

从上到下执行 🡺

在执行过程中遇到耗时操作就会扔事件队列里 🡺

事件队列要排队是先进先出 (事件队列主要存放任务 当任务比较耗时就会把任务扔进事件队列中 队列里有一个一个任务 任务来自主线程 什么样的情况会进到任务中?答案是I/O操作(文件读写 网络请求和响应处理 这是进到队列中的任务 实际上总结就是网络I/O和文件I/O 遇到这两种情况就会直接扔队列) 在node中是I/O操作 在浏览器中是前面三种情况 这是node和浏览器有区别的地方) 🡺

主线程遇到网络I/O和文件I/O后会直接扔队列然后接着往下走 这样主线程不堵塞 形成了高并发处理能力 这是先天性的 nodejs本来就有这个能力 因为它的设计思想一开始就是采用的异步I/O这种处理机制 🡺

当主线程所有代码都执行完了 主线程空闲了 才会去事件队列里查询 查询到任务后再返到主线程去执行此任务 主线程执行完就又查又执行 一直循环(单独有一个事件循环线程 循环此操作) 🡺

这点和浏览器的事件模型大体类似 不过里面的细节还是有差异 对于我们来说理解这点就够了 知道代码如何是异步执行的 所以队列中的任务相对于主线程代码肯定是最后执行的 因为要等到主线程空了后再执行

注意js运行时单线程的 但是浏览器环境 nodejs环境它们本身是多线程的 所以不是一个概念

文件的API通过文件的读写操作来演示 所以在nodejs中API基本都提供了两种 同步和异步

nodejs中代码开发风格主要是异步 主要是基于回调函数的代码风格 这是nodejs的特征 因为他的设计就是遵循异步编程模式

node设计思想就是异步 那么当有同步和异步api时尽量用异步

### 文件操作

* 文件信息获取
* 读文件操作
* 写文件操作
* 流式操作(大文件操作)
* 目录操作

nodejs设计是按异步方式来做的 但是也支持同步操作 所以api一般都有异步和同步两种 默认是异步 (都有callback)带sync的是同步(没有callback) 异步的callback是必须写

**fs 是File System缩写 译文件系统**

const fs = require(‘fs’)

**文件信息获取**

**1. fs.stat(path[, options], callback) 异步操作返回值undefined即没有返回值**

**1.1参一path(绝对.相对路径均可)**

**1.2参二callback 回调有两个参数 (err, stats)， stat 译状态 统计**

1.2.1几乎所有回调第一个参数都是**err错误对象** 如果err为空null表没有错误 如果是错误对象表示报错了 报错后回调里面的代码也会执行 但是报错再执行后面的也没意义 所以在开始应判断err 若报错则终止程序 不执行后续代码 err是错误对象 可根据报错信息找错误原因 可帮我们调试 从程序严谨角度来说 每次都应判断return下

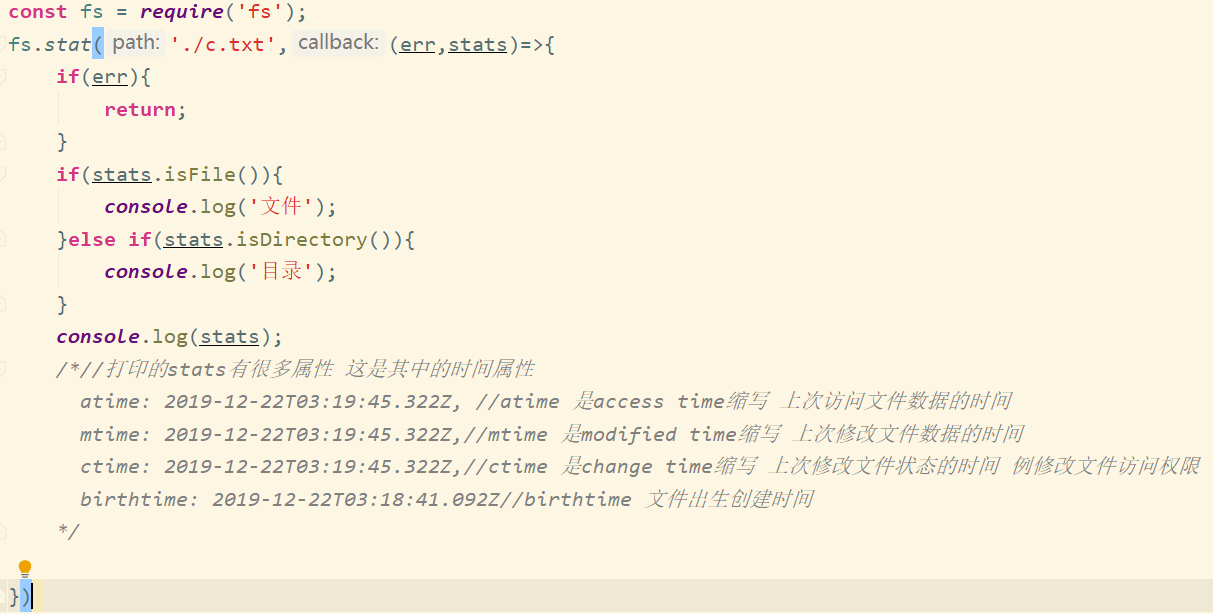
**1.2.2第二个参数是对象 fs.Stats 对象提供了关于文件的信息**

**有两方法**

stats.isDirectory() 判断是目录返回true

stats.isFile() 判断是文件返回true

**Stats打印出是对象 需要对上面的时间属性有了解**



**同步操作fs.statSync(path)**

只有path没有 callback 而stats是通过执行后的返回值得到的 同步操作的问题就是若耗时则后续代码只有等待



### 读文件操作

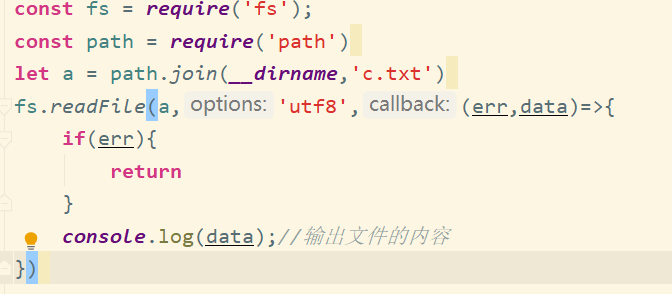
## fs.readFile(path[, options], callback) 默认data是Buffer 最好指定编码

一参:path 二参是可选项指定编码 三参callback

回调有两参:

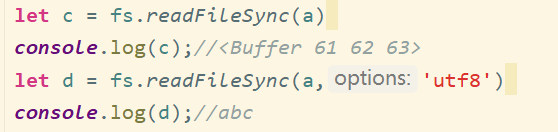
回调一参err

回调二参data data是读文件后得到的文件内容 若中间的可选项不写即不指定编码 默认data是Buffer实例 想查看内容就需要toString转换 若写了指定编码例utf8则不需再手动toString直接打印data就行 等效toString后的结果



**同步操作fs.readFile(path[, options]) 返回值data**

同步操作没有callback返回值data 两参 一参path 二参可选是指定编码 同理若不指定data就是Buffer实例 若指定了就是utf8编码后的内容不需再toString



之前提到readFile读取的内容是Buffer实例,那时是设置编码utf8.即把所有内容都转成了utf8编码,如果是字符串或者html这种文本类型的用此编码可以正常显示,但是若是图片反而会导致无法正常显示内容.

综上以后读取的时候不设编码 在响应时 响应头里设置对应类型 和编码(只有text开头的才设utf8编码)

### 写文件操作

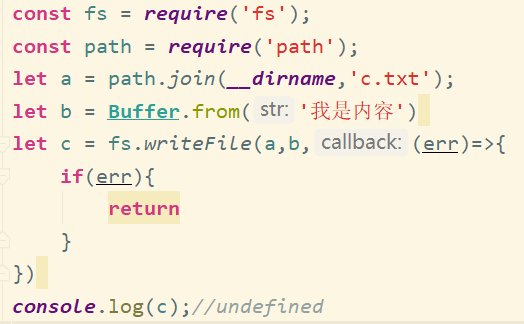
## 1.fs.writeFile(file, data[, options], callback) 没有返回值

一参 path全路径含文件名**(若已有此文件则覆盖文件及内容 若没有则创建文件并写入)**

二参 data写入的内容(若是Buffer则三参会被忽略 若是字符串则需指定三参)

三参 可选指定编码例utf8

四参 回调只有一个err参数





**2.同步操作fs.writeFileSync(file, data[, options])** 没有返回值 没callback

其他操作和前面一样 若data是Buffer则不指定编码 若是字符串则指定编码

大文件操作(流式操作)

前面的读写操作即readFile和writeFile会将文件的内容存到内存中 若文件内容量大 就造成内存消耗量大 影响处理性能所以通常用它处理小文件 针对大文件nodejs提供了另外一种操作 叫文件流式操作

所谓的流式操作和现实中场景相似例如河流的水是一部分一部分流动

所以相对于在数据处理中也有形象说法叫数据流 数据一部分一部分进行处理

nodejs提供了api来一部分一部分处理数据

基于事件的方式读写文件,node不操作dom所以浏览器中的click hover这类事件都没有 只有读文件写文件事件和网络操作事件 基于事件也是node核心之一

**1.fs.createReadStream(path[, options])**

读文件的数据流

## 2.fs.createWriteStream(path[, options])

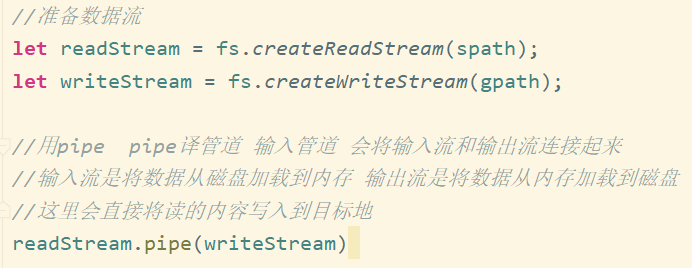
写文件的数据流

案例直接读取再写入到桌面:

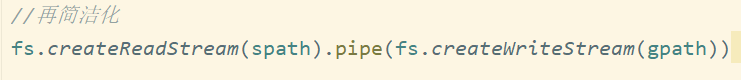
方法一:data事件 end事件



方法二:readStream.pipe()



再简洁 省去变量 一步到位

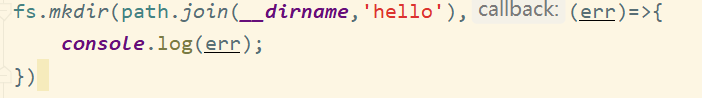


目录操作

**1. 创建目录**

## 1.1 fs.mkdir(path[, options], callback) 异步 (创建后的目录路径 会根据此路径创建)

## 1.2 fs.mkdirSync(path[, options])同步

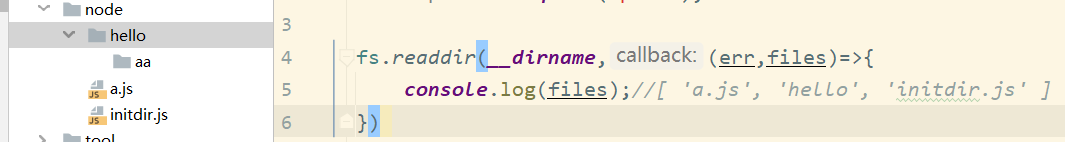




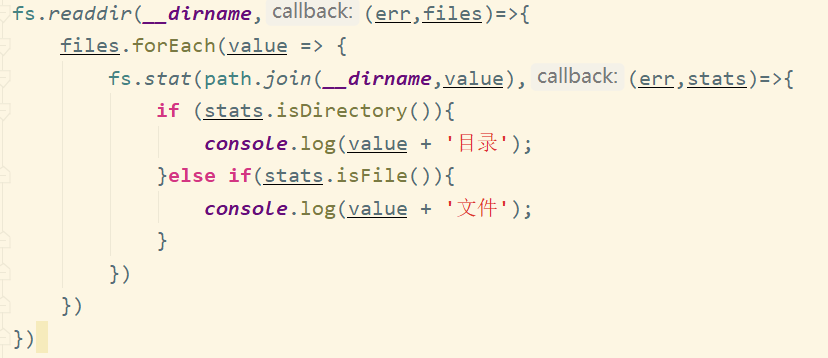
1. **读取目录(得到目录下所有的一级文件名和一级目录名)**

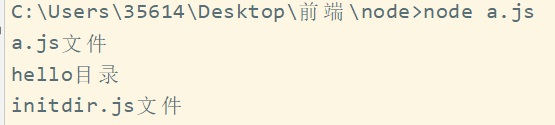
**2.1 fs.readdir(path[, options], callback)**

回调有两个参数 (err, files)，其中 files 是目录中的文件名和目录名的数组（不包括 '.' 和 '..'）即不包括当前目录名字和上层目录名,注意:不读取子目录下的目录 只读取一层



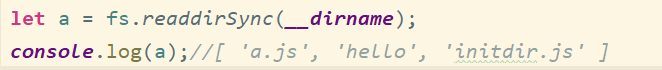
files是数组 若想对数组里的进行判断 判断是文件还是目录 进行如下操作:

****

****

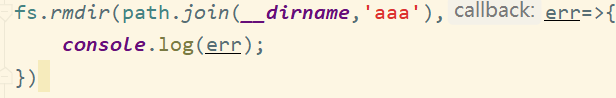
**2.2fs.readdirSync(path[, options])**

同步的区别就是没回调 files通过返回值获得



1. **删除目录** 若目录不为空则报错 只能删空目录

## fs.rmdir(path[, options], callback)



## 3.2 fs.rmdirSync(path[, options])



**流程化创建****基本项目目录 案例**

**总结:**

1. **const fs = require(‘fs’);**
2. **const path = require(‘path);**
3. **path对路径操作 path.join(路径,目录或文件名)** join译拼接 会将路径和后面的文件名拼接上 若拼多个则,逗号隔开继续写名字 会自动加/
4. fs是对目录和文件进行操作的 可创建目录 创建文件 写入文件内容
5. **fs.mkdir(path.join(路径,目录或文件名),callback)**通常和path结合使用
6. **fs.mkdir()** 创建目录 异步操作 参一路径 参二回调 异步操作的callback必须写
7. **fs. mkdirSysnc()**同步操作 没callback
8. **fs.writeFile(路径,写入的内容,call)**
9. **fs.writeFileSysnc(路径,写入的内容)** 相比异步 同步性能打折扣但写代码方便

**案例:**

可以通过node核心模块即api 对工作中一些需求进行流程化处理 做流程化工作

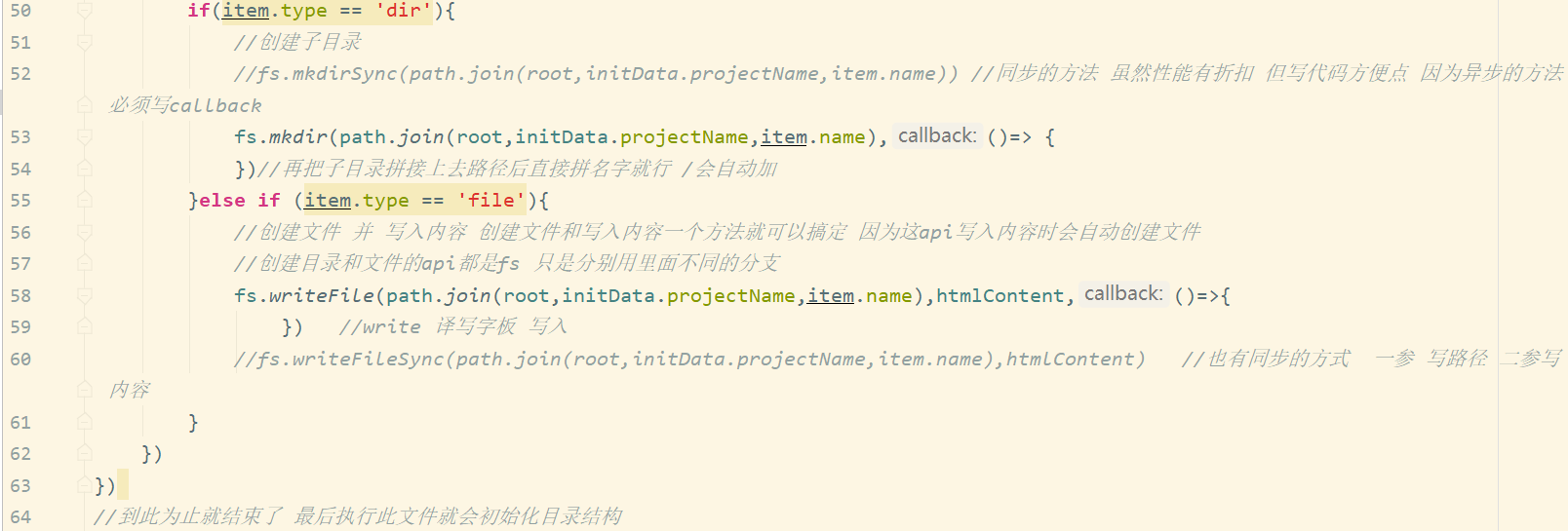
通常我们写一个项目 需要手动创建项目目录及文件(js目录 css目录 img目录 index.html)但是手动比较麻烦 nodejs提供了核心模块 可用nodejs的api实现自动化脚本(脚本是js 自动化脚本就是根据js自动创建) 一个执行此js文件(自动化脚本)的执行命令就自动创建目录

以后可以根据脚本自动完成一些事的 我就叫自动化脚本

例:







**包管理工具**

**npm**

**1. npm对包的管理: 就是 安装 卸载 更新操作**

**2. 包的安装方式:全局安装和本地安装**

**2.1全局安装**

全局安装的包,在任何文件夹下都可以使用此包,用户可以在命令行直接运行该包支持的命令,全局安装的包一般用于命令行工具。全局安装的包在nodejs环境的node\_modules目录下

**2.2 本地安装**

本地安装的包只安装在当前目录下node\_modules目录下,其他目录不可使用,一般用于实际开发工作

**3. npm包常用命令**

**3.1安装包(不指定版本号默认安装最新版本)**

npm install 包名 -g (全局安装 -g表全局安)

npm install 包名 (本地安装 不写-g表本地安装)

install可以直接简写为i 例npm i 包名

* 1. **安装包指定版本**

npm i 包名@版本号 -g (包名@接版本名表指定版本)

* 1. **卸载包**

npm uninstall 包名 -g (若是全局安装的 卸载要加-g若不是就不加)

**3.3 更新包(更新到最新版)**

npm update 包名 -g

更新到指定版本npm update 包名@版本号 -g

若更新最新版也可npm update 包名@latest -g

update这个命令有时候不好使 会不起作用 这时候就可以把之前卸载 重新安指定版本

**总结:包的安装 卸载 更新 其中的-g只要安装写了 那卸载和更新都应该指明-g 不可省略**

### 解决npm安装包被墙的问题

* --registry
  + npm config set registry=https//registry.npm.taobao.org
* 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
* nrm
  + 作用：修改镜像源
  + 项目地址：<https://www.npmjs.com/package/nrm>
  + 安装：npm install -g nrm

**把自己写的项目规范成包**

对于本地安装的包一般是用于开发某种具体功能,所以我们本地写的代码也应该做成一个规范的包.这样方便代码重用.

在学习vue的时候学了webpack 并不是说vue才能用webpack 只是刚好学了而已 这两个是独立的 所以以后写项目都用脚手架 且都用npm下载包 模块引入 减少请求次数 压缩代码

规范的包需要先npm init -y生成packge.json文件 后续再npm安装对应包即可

npm安装的包在node\_modules里 可以看到包里面有很多文件夹 是因为此包还依赖很多其他包 将依赖包也安装了 这样就不需要程序员自己维护依赖关系

npm安装的包都在node\_modules下 直接require(名)即可 包名和包文件里的packge.json中的name属性是同一个名字(这是规则) require的名字实际就是这name属性对应名

运行可以通过脚本运行 或者 node 文件名 或者node . 最后这个命令node .点是运行packge.json里main属性指向的文件

**这里学习下art-template包 这是前后端通用的js模板引擎 art译艺术即艺术模板**

安装

npm install art-template --save

渲染模板

// 基于模板名渲染模板  
template(filename, data);

var template = require('art-template');  
var html = template(\_\_dirname + '/tpl-user.art', {  
 user: {  
 name: 'aui'  
 }  
});

拿过来后这里需要一个文件 filename这就是模板引擎的模板,文件后缀是.art

这文件里即模板里的内容是html和模板语法结合,要用模板的数据就要用模板语法,要求按模板语法写 模板语法有标准语法和原始语法 推荐标准语法

{{if user}}  
 <h2>{{user.name}}</h2>  
{{/if}}

文件写入标准语法的内容后 最后的html是渲染成功的内容 可以直接打印看结果 输出的是<h2>aui</h2> 模板里面的数据已经被填充了 填充的就是user.name

**-S生产环境和-D开发环境**

-S 向生产环境添加依赖 就是在packge.json记录

-D 向开发环境添加依赖

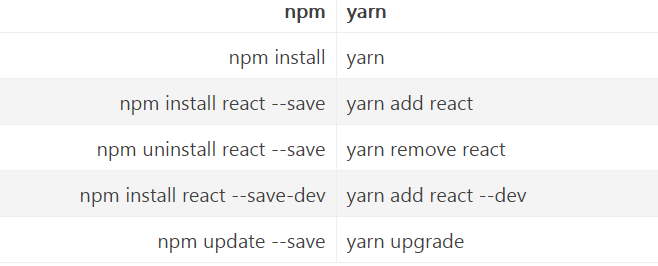
加依赖的作用:包在发布后 这个node\_modules是不会一起发布的 因为里面代码量大 传递不方便 所以提交的时候只会提交源代码 它的依赖只会记录在packge.json中 这样包在运行时会随时根据packge.json文件安装 所以依赖要分情况安装 生产环境的包是程序运行必须的包 开发环境的包一般只在开发阶段使用 真正上线时这些包就没用了 所以要分开处理 分开后可以选择性安装只安装生产环境的 拿到包后怎样安装依赖? 可以npm i -S 这样就只会安装生产环境的依赖 若npm i 则表示安装生产和开发环境全部的包

**yarn基本使用**

作用和npm类似 主要是管理包

区别:性能更优越 提示更友好

npm一方面性能不好 一方面特定场景命令无效例update 所以为了弥补这些问题出现了yarn



安装:npm i yarn -g

初始化包 yarn init

使用命令时不写-S 默认就是-S 开发就是—dev

yarn 全局安装的命令时global



**markdown的工具**

可以生成网页的方式 有目录树

安装npm install -g i5ting\_toc

在对应目录运行命令i5ting\_toc -f 文件名-o

**自定义包**

### 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

### 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
* description：包的简要说明 给下载者看的 描述包功能
* version：符合语义化版本识别规范的版本字符串 两个点隔开的三个数字
* main:入口文件
* keywords：关键字数组，通常用于搜索,别人搜索包可以根据这些关键字匹配
* maintainers：维护者数组,在托管平台中比如github可以让别人帮忙维护，每个元素要包含name、email（可选）、web（可选）字段
* contributors：贡献者数组，这个项目都有哪些人参与了开发工作,格式与maintainers相同。包的作者应该是贡献者数组的第一- 个元素 维护者和开发者可以是同一个也可以是不同的
* bugs：提交bug的地址，可以是网站或者电子邮件地址 如果有bug可以提交到此地址
* licenses：项目遵循的开源协议,协议有一个地址指向这个文档描述具体开源规则,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：代码托管的仓库地址,一般用的github.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

### 自定义包案例

模拟i5ting-toc把markdown文件转成页面 实际上转化的具体工作也不需要我们重新开发 在npm已经有人开发了一个通用功能 包叫markdown-it

我们需要做的是:

1.利用markdown-it将markdown转换为html 转换后的html是一个代码片段

2.准备一个模板 并且将模板内容和代码片段结合 将最终结果放到目标页面

3.要求不改变原模板内容



## Node.js实现静态网站功能

**实现简单的静态服务**

实现静态服务器功能 所谓静态服务器就是提供静态网页服务

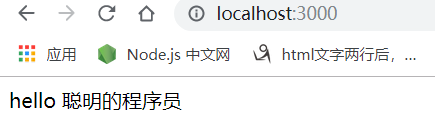
这里需要用到核心模块http

## 1.http.createServer([options][, requestListener])

参数可选 创建服务 返回一个新建的http.Server实例 实例上有一系列事件和相关API

**方法一**

res.end()是完成响应 结束响应 参数可传可不传(若传会显示到页面 只能传一个参数 多传没用) 



**方法二 简写**



**路径分发**

**场景:**前面的案例会出现不管访问localhost:3000/index.html 或者 localhost:3000/about.html返回的都是一样的内容.这是因为我们没有根据不同请求返回不同内容 接来下实现根据路由不同返回不同内容 也就是所谓路径分发

**一:基本路径分发**

request事件里面的req和res分别是不同的实例对象,可以在文档根据对应的实例对象查看对应的属性和方法

req实例**:http.IncomingMessage类**  Incoming译传入 进来

res实例:**http.ServerResponse类**

**req.url属性获取传入的地址**(端口号之后的全部地址(不含#hash) 注意判断的时候要加/ 是/开头)



**这里用到es6的判断字符串以什么开头 以什么结尾 记一下**

判断字符串以什么开头：此方法区分大小写  
\* startsWith()； 返回值是Boolean 类型  
\* endsWith() ； 以什么结尾

接受两个参数

第一个参数，要在此字符串开头搜索的字符，第二个参数是指定从字符串开始的位置，默认从零开始

此方法区分大小写；endsWith()方法和startsWith()方法的语法都是一样的，不过endsWith()方法是从字符串的末尾开始查找。

**二:res.end() 和 res.write()**

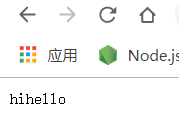
**res.end()**

**完成响应 结束响应** 只能执行一次 参数可传可不传 若传了页面显示对应内容 只能传一个参数 多传没用

**res.write()**

**向客户端响应内容 可写多次** 客户端页面显示对应内容





**三:根据请求响应完整页面**

前面都是响应字符串 这里响应文件html页面

如图:

1.因读文件操作用到多次所以直接封装成方法,调用时传入路径表响应哪个页面

2.按照以前作用域认知 正常不传res就行 但此次不传是undefined 所以必须传 具体原因待定

3.直接返回中文字符串会乱码 所以设置:

res.writeHead(200,{

‘Content-Type’:’text/plain;charset:utf8’ //注意是用分号隔开的 编码是=不是:

})

3.1 res.writeHead()向请求发送响应头

3.2 里面200是状态码 表示成功 根据不同情况设置状态码,这里设置主要是给浏览器看的 如果是未找到文件但依然需发响应提示那么状态码应该404 这里的状态码主要是我们(服务端)根据情况设置 浏览器那边看状态码 才知道是成功还是失败

3.3 headers [<Object>](http://nodejs.cn/s/jzn6Ao)响应头是对象 里面设置响应头类型和编码

‘Content-Type’设置响应头的类型 和编码 这里的text/plain是文本类型 也有text/html



**三.一 根据请求 设置不同响应头类型**

一般浏览器请求的文件和服务器响应的文件是同名的,所以大多时候我们都是直接根据请求来查找同名的文件

**1**.当err报错时 返回响应内容 提示 并且响应头设置为404 路径错误

**2.**mime.json是一个json文件 里面属性是不同请求url的后缀即请求文件类型 值为不同的响应类型

2.1根据url请求的后缀 来找json文件中对应后缀的对应值 此值为响应类型

2.2不同的请求必须设置对应的响应类型 否则会导致页面无法正常显示等错误

2.3当响应类型为text开头时,需要设置编码为utf8;

**3**.之前提到readFile读取的内容是Buffer实例,那时是设置编码utf8.即把所有内容都转成了utf8编码,如果是字符串或者html这种文本类型的用此编码可以正常显示,但是若是图片反而会导致无法正常显示内容.

综上以后读取的时候不设编码 在响应时 响应头里设置对应类型 和编码(只有text开头的才设utf8编码)



**url(获取get请求参数)**

**1. url API 这里只介绍两个**

* [url.format(urlObject)](http://nodejs.cn/api/url.html#url_url_format_urlobject) 将url对象转成字符串 就不详细讲解了 了解了url.parse()就知道
* [url.parse(urlString[, parseQueryString[, slashesDenoteHost]])](http://nodejs.cn/api/url.html#url_url_parse_urlstring_parsequerystring_slashesdenotehost)将url字符串转为对象

**2.这里对url.parse(参数)进行详细讲解:**

**一参** 传入url字符串 返回值 url对象

**二参** 可选参 布尔值true或者false 默认为false 当传true的时候会将query变成对象 也就是查询键为属性 值为值 当为false的时候就是字符串 通常我们都会传ture

**三参** 可选参 一般都不传 这里不做了解 很少用

**综上:** 正常使用url的时候我们都会将url字符串和true一起传递

**2.1完整的url详解图:**



* 1. **案例**

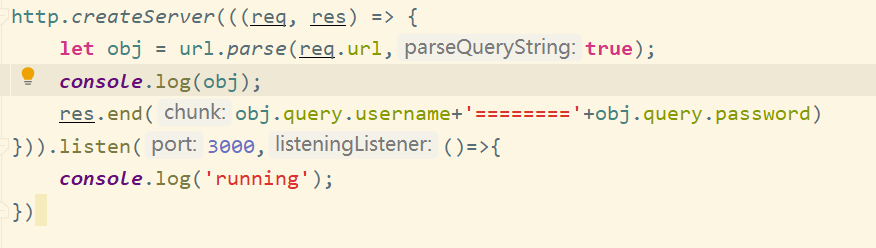
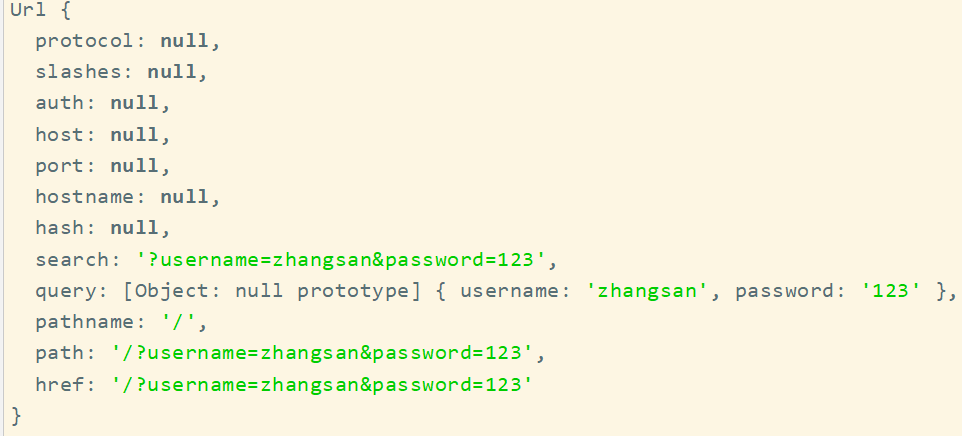
**场景2.2.1:当只传url字符串时的返回结果 及 对结果的详解**



**场景:2.2.2 模拟获取get请求的参数 (这里会传true将query转为对象)**

通过req.url获取的url只能获取端口之后 和 #hash之前(不含#hash)所以对象属性很多为null

res.end()传参顶多只能传一个 多传无效

****

****

**querystring(获取post请求参数)**

前面的url可以获取get参数 但是post请求参数则不能通过url获取 所以这里用到另外一个也就是querystring

**1.这里介绍其中两个api**

let querystring = require(‘querystring’);

**querystring.stringify(obj)**  将对象转成字符串

**querystring.parse(str)**  将字符串转成对象 常用

**2.**有了**处理参数**的 但是要获取到请求传过来的数据才能进一步处理 前面用的req.url这里很明显不能用 所以用到了事件 get请求可以直接url路径里传过来 但是post请求 前端一般都是自己获取到请求的参数 然后拼接成路径 发送ajax请求 这样的方式传过来的 而后端通过获取请求数据的事件来获取

**2.1 获取传递过来的请求数据的事件** 数据不是一次性获取不是只调用一次 而是chunk一部分一部分获取的 获取一部分调用一次即调用多次

**req.on(‘data’,(chunk)=>{**

**})**

**2.2 end事件是数据接收完成的时候调用**

**req.on(‘end’,()=>{**

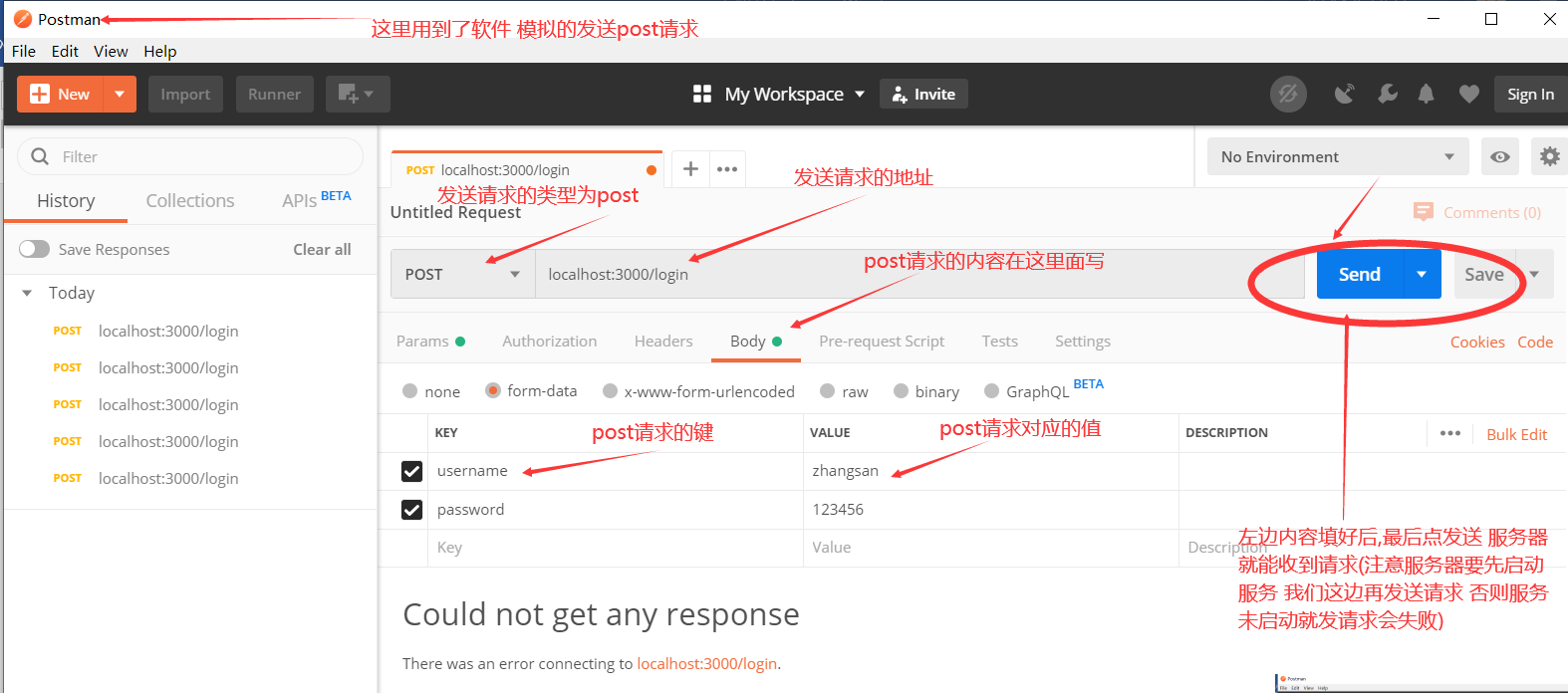
**})**

**案例:获取post请求参数**

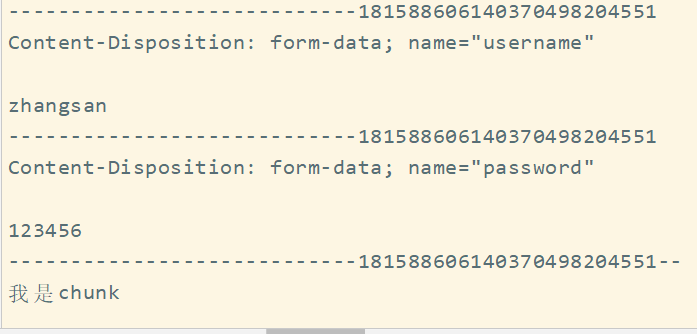
**第一步 服务器做相应配置并启动服务**

****

**第二步:启动服务器后发送请求 这里用到了postman软件 模拟用户发送请求**

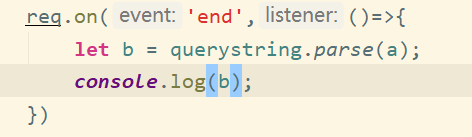


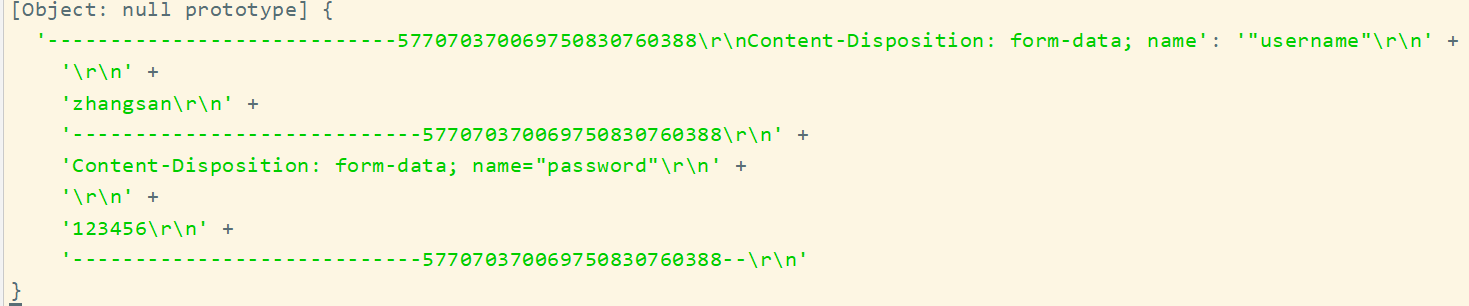
**第三步 当发送请求后 服务器获取到请求 打印的结果**

****

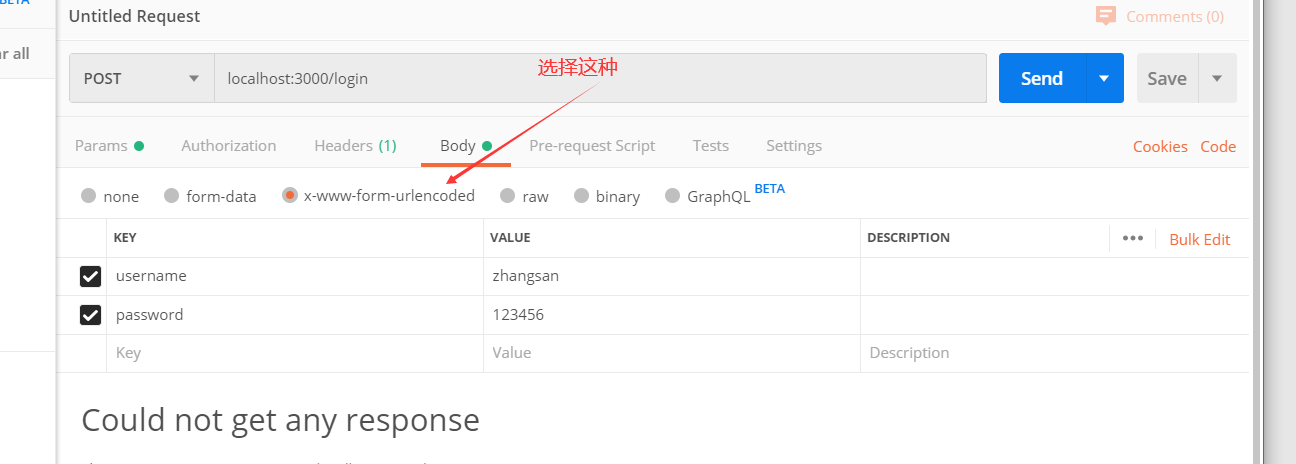
**这里只拿到请求的数据是一大堆字符串 但是我们需要将它转成对象 然后拿里面的特定值**

**可以看到下图转换成对象是这种,很明显我们不能通过正常obj点属性的方式拿 这里和postname的软件有关系 因为我们数据类型选错了**

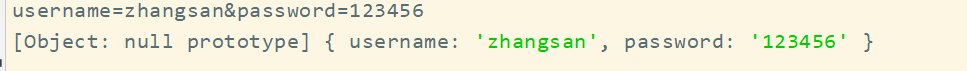
****

****

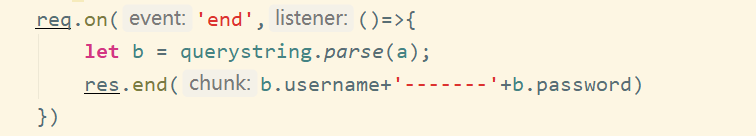
**正确的应该选择如图这种 其他不变 这种格式就是我们之前提交post请求时填的第三个选项的格式 虽然第三选项已经不用填了 不过这里要写正确的格式**

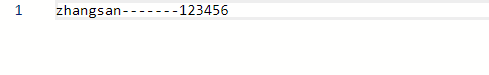


**如图调整格式后拿到的数据就是标准post请求格式的字符串了 用标准格式的字符串转化的也是标准的对象**

****

**接下来获取参数的值 并将其响应到页面**

****

****

**到此获取post请求的参数和处理就完成了**