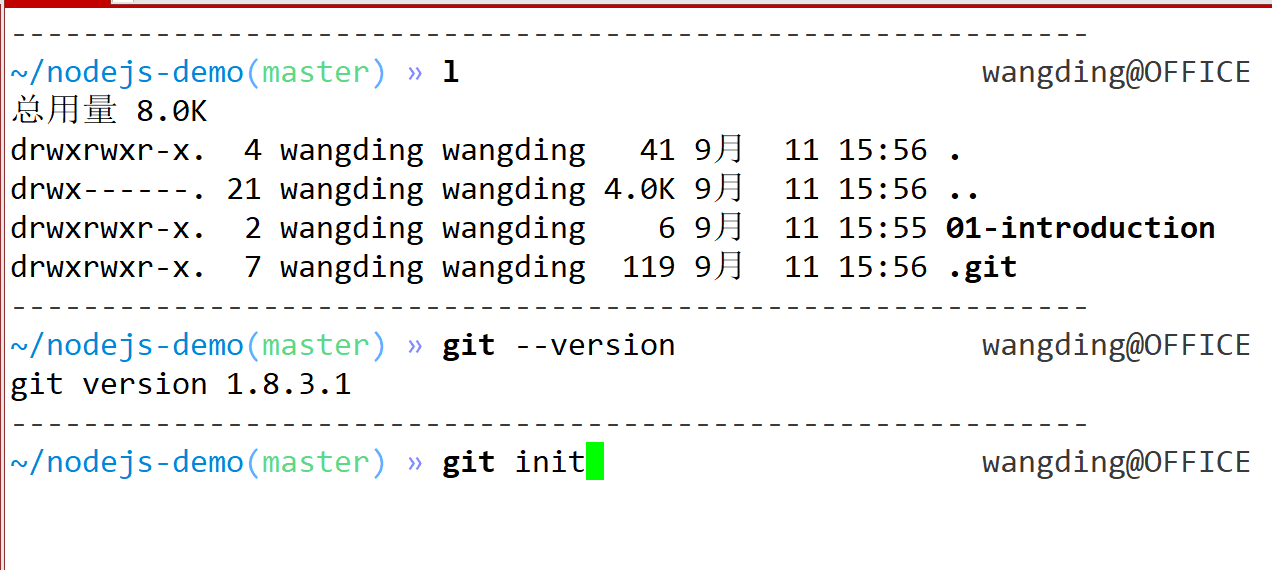
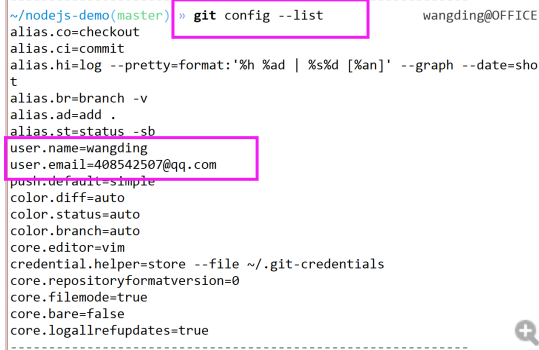
**01 Nodejs简介**

1. 运行环境：它是一个解释器，是个虚拟机

它是接口【相当于积木的接口】

1. 用nodejs可以方便制作分布式的程序【node为节点，js是JavaScript的缩写】
2. Node.js是一个Javascript运行环境【**Node.js是一个服务器端的、非阻断式I/O的、事件驱动的JavaScript运行环境，能够使得javascript脱离浏览器运行。**】
3. Github：





1. 代码补全：

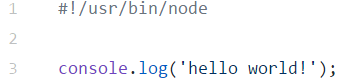


1. 编写hello world：

cd 01-introduction【打开一个文件夹（若想另外创建个文件夹需使用mkdir…）】

touch 01-hello-world.js 【在其创建一个js文件（使用touch）】

vim 01-hello-world.js 【打开刚才创建的文件】



【编写完之后 esc+：wq为保存退出 esc+：q退出不保存】 保存：空格+w

node 01-hello-world.js 【查看刚才的程序，显示helloworld】

1. 改权限后查看更加方便：

sudo chmod u+x【改权限】

./01-hello-world.js【使用此命令查看hello world】

1. 编写hello world web程序

代码：

（1）



（2）

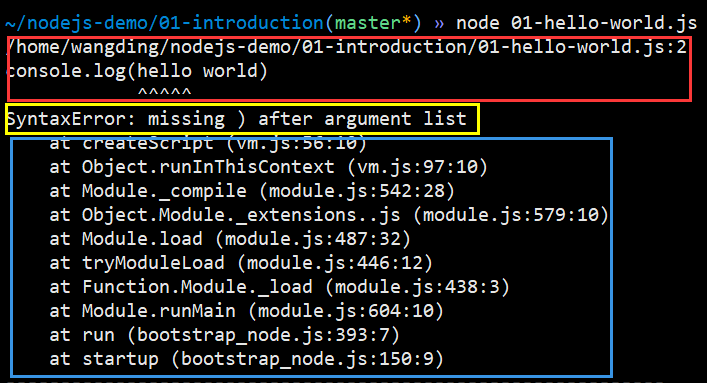


Curl <http://localhost:8080>

1. Node调试环境【跟浏览器中console控制台是一样的】
2. 上传代码到github：
3. gaa
4. gsb
5. gcmsg “ ”【双引号里是上传的注释】
6. gp

【gp -u origin master（第一次上传）】

1. 报错：

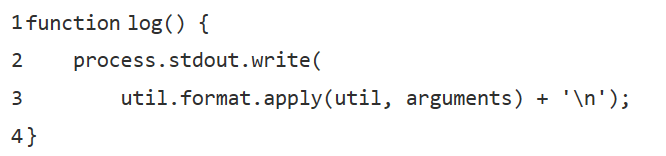


第一：哪里出错，第几行，那一行的代码，关键哪出错

第二：什么错误

第三：调用栈

1. console.log([data][, ...args]) 【中括号表示参数可有可无】
2. NodeJS程序的标准输入流(stdin)、一个标准输出流(stdout)、一个标准错误流(stderr)分别对应process.stdin、process.stdout和process.stderr。第一个是只读数据流，后边两个是只写数据流，对它们的操作按照对数据流的操作方式即可。例如，console.log可以按照以下方式实现。



**02 全局（上）**

**【注意：（）代表函数，[]代表数组，{}代表对象】**

1. filename脚本文件的绝对路径【全局变量，存储的是文件所在的文件目录】

dirname脚本文件对于目录的绝对路径【全局变量，存储的是文件名】

【process.cwd()是项目启动的根目录，require里的相对路径是相对于所在的文件】

1. cp 拷贝
2. 最好是写相对路径【**?**相对路径得到绝对路径】
3. 代码：

#!/usr/bin/node

console.log('file name:', \_\_filename);

console.log('dir name:', \_\_dirname);

var fileName = \_\_dirname + '/views/view.html'; // 方案 1：不好，没有考虑跨平台

**【/斜杠没考虑到跨平台，windows（为\\）和其他相反】**

// 方案 2：不好，代码复杂，平台考虑不全面

switch(process.platform) {

case 'linux':

fileName = \_\_dirname + '/views/view.html';

break;

case 'win32':

fileName = \_\_dirname + '\\views\\view.html';

break;

default:

fileName = 'something wrong';

}

console.log('fileName:', fileName);

// 方案 3：最佳 【path.join(path1，path2，path3.......)  作用：将路径片段使用特定的分隔符（window：\）连接起来形成路径，并规范化生成的路径。】

const path = require('path');【node中自带的模板】

fileName = path.join(\_\_dirname, 'views', 'login.html');

console.log('fileName:', fileName);

1. 控制台格式化输出信息：
2. 代码

#!/usr/bin/node

const user = {

name: '王顶',

age: 41,

qq: '408542507'

};//【定义了个对象】

//方法1

console.log('name: %s', user.name);

console.log('age: %d', user.age);

console.log('JSON: %j', user);

console.log('qq:', user.qq);

//方法2【template】

console.log(`qq: ${user.qq}`);

//方法3【拼接】

console.log('qq:' + user.qq)

console.error('Error! something wrong!');

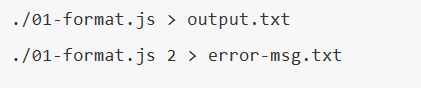
1. 若都是直接输出，则console.log和consol.error两个方法看不出本质的区别

console.log('Error! something wrong!');

console.error('Error! something wrong!');

【都显示Error! something wrong!】

1. 通过命令行重定向，观察 console.log 和 console.error 两个方法所使用流的区别



console.error('Error! something wrong!');【错误信息不会输出，直接保存在error-msg.txt】 =>使用cat 文件名 （查看错误信息）

1. 控制台基准测试

对好使任务运行的时间进行采样【计算机都是动态的，出来的时间可能也不同】

代码：

#!/usr/bin/node

console.time('TEST');

longTask();

console.timeEnd('TEST');

function longTask() {

var n;

for(var i=0; i<10000; i++) {

for(var j=0; j<10000; j++) {

n = i \* j;

}

}

return n;

}

1. 获取平台信息
2. 执行 Linux 命令 ps aux，验证当前进程的 ID 信息和程序中得到的进程 ID 信息相同 console.log('process id:', process.pid);
3. 命令行：

node -v node.js版本

id 登录账户信息

pwd 当前工作目录信息

1. 内存使用情况：
2. 打印环境变量 env

#!/usr/bin/node

const log = console.log;

log('architecture:', process.arch);//多少位例如，x64

log('platform: %s\n', process.platform);// \n是转义字符，换行 platform：linux/Windows

log('process id:', process.pid);

log('exePath: %s\n', process.execPath);

//process.stdin.resume(); 暂停【光标停在这儿】

log('node version:', process.version);//版本

log('user id:', process.getuid());//uid

log('group id:', process.getgid());//gid

log('cwd: %s\n', process.cwd());//路径（path）

log('rss:', process.memoryUsage().rss);//内存使用

log('heapTotal:', process.memoryUsage().heapTotal);//v8引擎

log('heapUsed:', process.memoryUsage().heapUsed);

log('external: %s\n', process.memoryUsage().external);//外部内存大小

//【获取在应用程序所占用内存（单位都是字节）】

log('env:', process.env);//环境变量

log('LOGNAME:', process.env.LOGNAME);//获取登录名

log('host name:', process.env.HOSTNAME);

1. 获取命令行参数

【？文件后面添加命令行参数】

1. 获取命令行参数，命令行参数作为一个数学表达式

for(var i=0;i<process.argv.length;i++){

console.log(`${i} argv:${process.argv[i]}`);

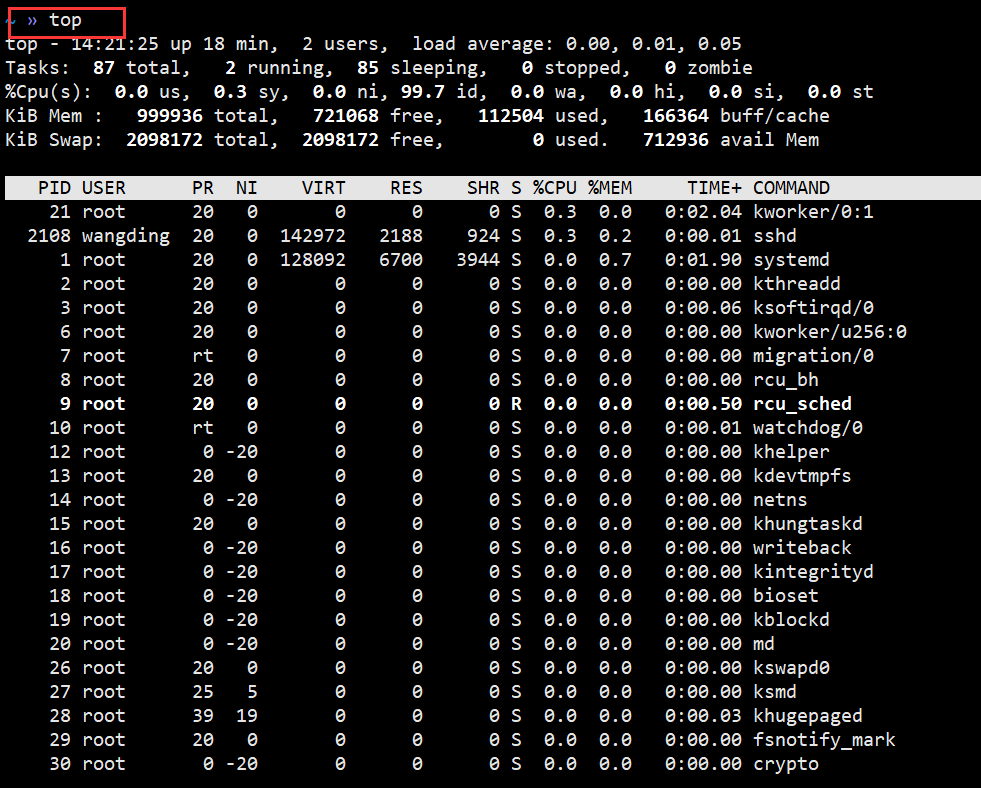
}

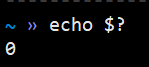
console.log('%s = %d',process.argv[2],eval(process.argv[2]))

1. 处理退出码

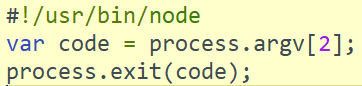
return 0【退出】（是正常退出）

top命令 出现进程表【实时变化的】



正常退出【查看退出码】

若ctrl+c 异常退出 

1. 
2. ./03-exit-code.js 4【运行】
3. Echo $?
4. 输出了退出码 4

但是实际程序是不健壮的 比如说，./03-exit-code.js abc….

//有健壮性的



1. 操作标准输入输出流

stdin/stdout

console.log打印会自动换行

【04-std-io.js脚本】

1. 处理信号量

ps aux【查看进程状态】（sl+ sleep / r run）

kill -2 …【终止进程】例，kill -2 80870

1. 实现 my-kill 程序

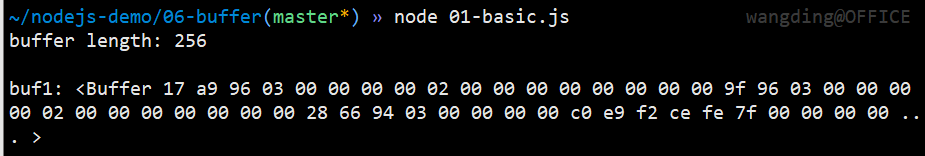
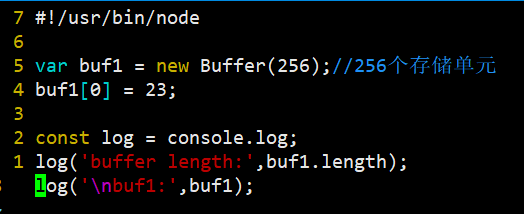
Kill程序前，一定到先查找【ps aux】多少进程 运行时，进程是动态的

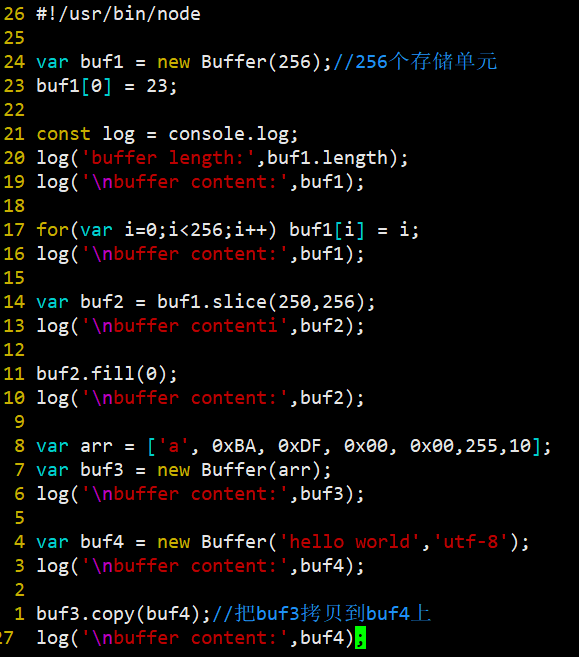
【运行命令：./05-my-kill.js 81316 SIGINT】 后面两个参数一个是进程编号一个是信号量

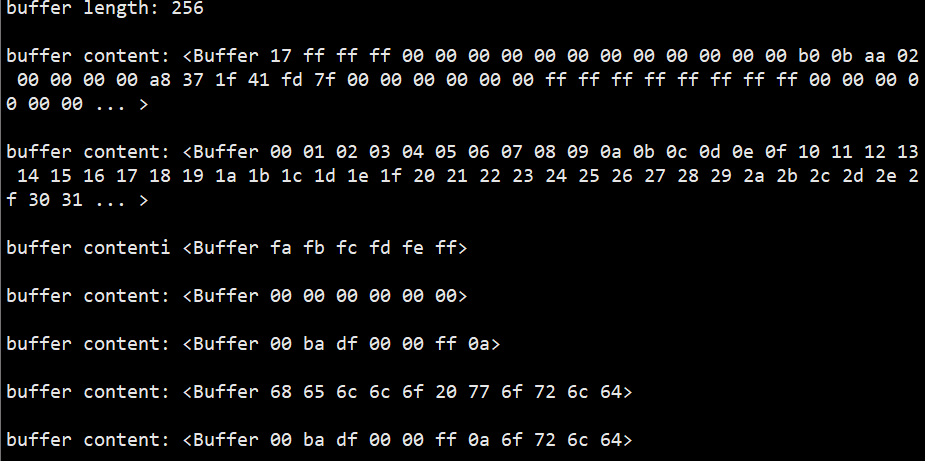
1. 实现延迟执行任务
2. 在xshell中，zR代码展开 zM收缩

**03 全局（下）**

1. Buffer基本操作



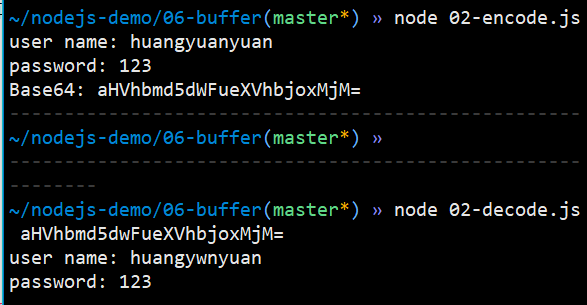




Base64：A-Z a-z 0~9 + / 【一共64位字符】

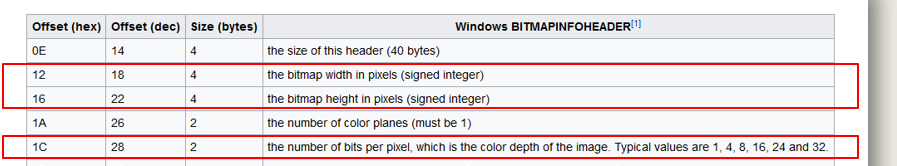
1. Buffer编码：把文字转化为base64

buffer解码：把base64码转化为文字解析出来



1. 二进制文件比文档文件效率更高
2. Buffer读取位图信息：

log('width:', buf.readInt32LE(0x12));



Width/height/color depth

Width：4字节32位readInt32LE（0x12）【0x12是十六进制，18为十进制，LE小数存储（BE大数存储）】

问题：

（1） buffer支持哪几种编码方式：6种

1. ASCII
2. UTF-8
3. UTF-16LE/UCS-2
4. Base64
5. Binary【二进制】
6. Hex【十六进制】

（2）模块化：【早先不支持文件级别模块（现在支持【AMD、nodejsCMD】）】

AMD是异步加载，CMD是磁盘内部加载（比较快，要使用直接从磁盘拿过来了）

（3）全局 require()【引用代码模块】、export【暴露】、module

想要访问其他的必须使其暴露出来，否则是私有的

1. 模块分类：
2. Nodejs API

分类：全局模块【不用require】、普通模块【需要require】

require

1. 第三方模块
2. require
3. npm install【npm网站上得下载下来】 ⬄ 重点：如何自己造模块
4. ccat …【查看文件】 ccat 01-hello.js

tree查看目录树

1. 所有安装的第三方模块都会放在node\_modules文件下

使用第三方模块一定要先下载，否则会报错

1. **Package.json**文件【自己造】：

DevDependencies 开发时候依赖 dependencies 运行时依赖

bin里面放可执行文件

npm install【这个命令会去看package.json文件需要哪些依赖，然后下载下来】

npm init 会生成package.json文件 【此时是没有dependencies这项】

npm install -S 会有dependencies（依赖） 【？-d】

1. 工厂函数 【返回的一定是个对象{}】
2. Var pi = require(‘./02-export-function’) //可以直接调用【后缀都可以不用加】
3. 导出变量、导出函数、导出对象



1. 全局安装有时候会没有权限 su 【以管理员身份】
2. 目录结构：

README.md

LICENCE

.gitignore【被忽略的文件】

Package.json

Index.js

bin\appname【可执行文件】

lib(util)

test\....js【测试文件】

程序结构：

主模块bin\appname

Index.js在根目录放着【把a.js b.js等各种模块暴露出来】

主模块下放着a.js … b.js 各模块

【如何暴露出一个相关接口】直接require 项目文件夹 （不用require 项目文件名/index.js）

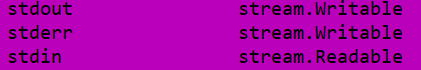
**04 事件**

1. 编写事件响应函数【onclick、onload…】【不管stdin还是什么，想要响应一定是on 例，process.stdin.on（…）】

**【看代码】**

**05 流**

1. wc-l 显示代码行数
2. find . -name “\*.js”【显示当前文件下的所有js文件】
3. 反向管道 find . -name "\*.js" | xargs cat| grep -v ^$ |wc -l
4. 流的操作是读一点儿操作一点儿
5. 三个标准流



1. 可读流可写流

信源【input】——>channel——>信宿【output】

1. http.createServer((req, res) => {}).listen(8080); 【req是发出的请求】【在xshell中curl可以请求 curl http:localhost:8080】

req.headers【请求头】

1. cat xxx.js | ./xxx.js 通过管道显示出来
2. (echo ‘wangding’;sleep 2;echo ‘hello’) | ./01-std-io.js

【若01-std-io.js会将小写转换大写，则会先大写显示WANGDING,隔两秒,HELLO】

1. 流是异步的
2. fs模块的同步&异步：同步方法执行完并返回结果后，才能执行后续的代码。而异步方法采用回调函数接收返回结果，可以立即执行后续代码
3. 流的分类：
4. **可读 Readable**

两种状态：暂停【默认的】 流动

流动的方法：

a. pipe 【】stdin.pipe(process.stdout)【把可读流接到可写】

b. data event

c. read

【从高级到第几的方法 前两个主要】

推送和拉取：=>push

推送push

拉取pull【什么时候读什么时候push】

1. **可写 Writeable**
2. 双工 Duplex
3. 转换 Transform

【可读流和可写流都是单一的（单向，如对讲机）】

【双工，可以读可以写（同时进行）（猫modem也是双工 调制+解调器【即可以发也可以收，是信源也是信宿】）】

【转换 可读流可写流的结合，是中间的转换】

1. 管道不能接反【可读 | 可写】
2. Echo ‘huangyuanyuan’ | head -c5【截取前5个字符】 输出：huang
3. 双工流&转换流

Readable+Writable

转换流的输入和输出是有联系的【数据的压缩，数据的加密（原始数据经过某种运算产生了输出）】，在同一高度 【读入数据->操作数据\_>写出结果】

双工流的输入和输出可以没有关系【不用在同一高度】

**06 文件系统**

1. 软件 = 程序 + 文档

【软件工程】 【数据结构】【文档分为内部文档、外部文档】

程序 = 数据结构 + 算法

内部文档：自己内部的【注释…】

外部文档：客户【用户实验手册…】

【设计时（数据结构）、运行时、编码时】

运行和编码时=>数据【内：变量 外：文件、数据库】

【语法、算法、章法】

1. [注]Buffer编码转换、二进制
2. [fs.lchmod(path, mode, **callback**)](http://nodejs.cn/api/fs.html#fs_fs_lchmod_path_mode_callback) [fs.lchmod**Sync**(path, mode)](http://nodejs.cn/api/fs.html#fs_fs_lchmodsync_path_mode)

Sync表示异步 同步的还有个回调

前缀：f、l

F前缀表示这个api这个参数处理文件id

L前缀表示处理连接

1. 异步的api操作效率高【逻辑复杂，调试不方便】

处理文件小 使用同步

处理文件大 使用异步

1. cat命令是查看文件内容

如，cat 01-information.js cat \*.js

1. 异常处理核心功能：让程序更健壮

代码：核心功能、异常处理、调试功能

异常处理：

1. 数据合法性校验【主动防御】
2. 异常处理 【被动接受】

异常处理分类：

1. 同步代码

Try…catch

1. 异步代码
2. 异步回调：cb【cb的在js中是call-back 也就是回调函数的缩写】

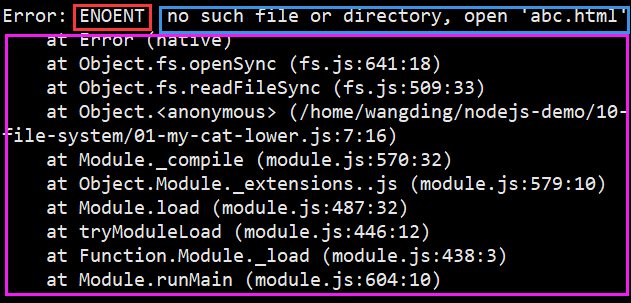
cb的第一个是 error

1. 事件机制：event事件

异常处理Error对象：

1. message 异常信息
2. ID 代码中Exxx[E开头] 错误代码或错误编号
3. trace、stack 错误栈

红色id 蓝色massage 粉色stack



异常处理策略：

1. 错误类型
2. 语法错误
3. 逻辑错误 【应该c=a+b 写成c=a-b】=>测试工程师去检测
4. 运行时错误 【各种异常】=>异常处理

(2) 要有异常处理

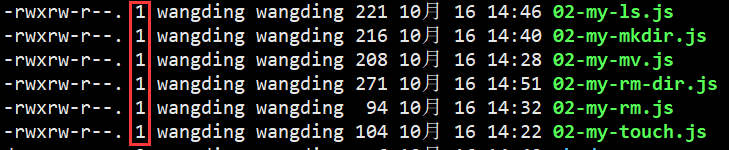
(3） 要有全局错误处理【必须要有】

1. 

第一个 –【文件】 l【连接】 d【文件夹】

1. 链接

（1）分类：两个链接【符号链接（软连接）、硬链接】



红框 链接（硬链接）数

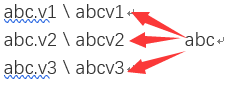
（2）命令

创建硬链接：ln 01-hello.js abc =>01-hello.js 和 abc 的链接数都加1

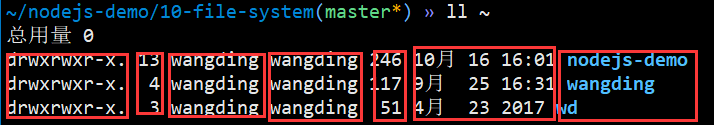
ln xxx.js链接到abc文件 【./abc 01-hello.js】使用xxx中的功能（如，xxx.js的功能是打印文件，则./abc 01-hello.js会打印出1-hello.js的代码）

创建软连接：ln -s xxx.js

(3) 作用



1. ll ~ 【目录的常格式输出】



权限 链接数 用户 用户组 字节数 事件 文件名

1. 属性操作：（三个）权限 操作者 …
2. -rw-rw-r-- sudo chmod u+x【给用户添加了可读权限】
3. Stat xxx 文件信息统计

**07 调试**

1. 很多问题出现在逻辑错误
2. 调试的方法：定位 修改
3. 调试手段：交互式调试、打印调试
4. 交互式调试是控制的方法【例，设断点】
5. 打印调试可以很好的操作打出错误信息
6. 远程调试、本地调试【一般为本地】
7. 动态测试【动态运行】

静态测试【不运行，直接检查源代码】

1. 黑盒测试：【只管输入和输出】

白盒测试：【有关于代码】

1. Walkthroughs 代码走查 【把各个的代码展现出来一起检查】
2. ESLint【静态白盒测试】
3. 可以检测出语法错误=>提高编码效率
4. 可以检测代码规范【团队写代码可以统一】
5. **不能检测逻辑错误**
6. 在xshll安装eslint：npm install -g eslint

eslint -v 检测版本号

1. ll | wc -l【查看行数】
2. eslint xxx 创建测试文件
3. eslint -h【eslint帮助 长参--help】

eslint init 【 Anser n node（用空格）no spaces single unix yes JSON】

eslint . 【查看代码错误】

更改规则【打开规则文件 vim .eslintrc.json】

https://www.jianshu.com/p/8671e439a811 安装nvm链接

1. 换行

\r\n widows

\r linux

\n ios

1. 打印调试中调试代码的处理

层次：

1. 低级：del 然后又加上
2. 中级：注释掉 //单行，半行 /\* \*/块注释
3. 高级：版本，debug，release 【c/c++ =>预编译】#define DEBUG（宏定义） main(){ #ifdef DEBUG\*\*\* #endif }

JavaScript里=>环境：开发，测试，生产

配置

环境变量、配置文件

环境变量的给法：

a. export NODE\_DEBUG=XXX curl <http://192.168.130.144:8080/01.static-sever.js> 【请求服务】=>运行 node..

b.命令行 node… 【例，NODE\_DEBUG=wd node 01-static-web-sever.js】

（wd为要调试的代码）

【打印所有的环境变量 export -p】

【取消环境变量 unset NODE\_DEBUG】

1. 调试命令：

cont, c Resume execution 继续

next, n Continue to next line in current file（单步执行）

s（跳入函数）

o（跳出函数）

sb（设置断点）

list（查看代码）

1. 命令行交互调试 控制代码执行

node inspect 02-dog.js【调试】

查看变量：debug> repl 【进入交互模式】>console.log(name) <taidi

查看变量：watch(this)监视窗口 取消：unwatch

Debug>list(100)查看100行

Debug>sb(12) 在12行设置断点

Debug退出 两次ctrl+c

1. Node inspect xxx.js 命令行交互调试

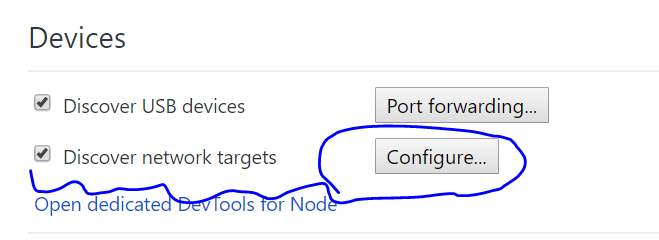
Node –inspect=ip.addr:port xxx.js【x（没有暂停，不太好用）】

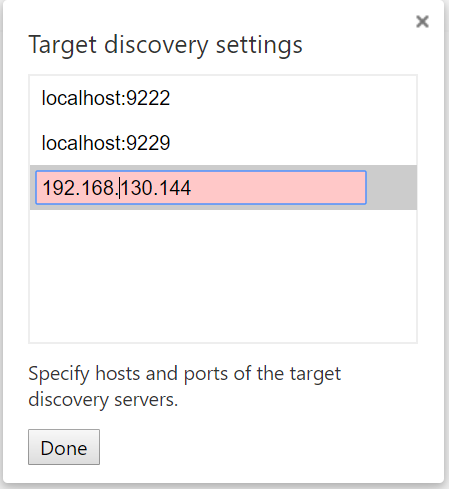
Node inspect-**wx**= ip.addr:port xxx.js【有暂停】

第二个和第三个 进入了图形界面的交互调试 Chrome dev tools

node --inspect-bark=192.168.130.144:8080 03-play-dog.js

1. chrome://inspect





F8 运行 F10暂停

1. API（Application Programming Interface）
2. 代码级

Play-dog.js

CMD M1

Dog.js

App.js -> HTTP API M2

1. 二进制级
2. 模块：【经过编译的】

.lib -> 静态链接库

.dll -> 动态链接库【跟着exe同时发布】

1. 可执行文件（exe）
2. 调用第三方程序的四个方法：spawn execFile exec fork
3. 调用命令行的shell外壳 exec
4. Eslint . 对当前的代码做静态 检查
5. Which git 查看git的路径
6. Echo $path 查看路径
7. Node => >process.env.PATH【环境变量的路径】
8. 请求：curl <http://localhost:8080>
9. cat a.txt | sort | uniq 【sort排序 uniq去重】

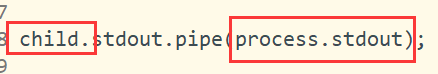
exexFile 不能用管道连接三个命令=>exec能[shell 外壳]

不用外壳的话，spawn能做

1. 正确： xx.exec(‘cat a.txt | sort | uniq ’,(err,out,error) =>{})

错误： xx.execFile(‘cat a.txt | sort | uniq ’,(err,out,error) =>{})【execFile只能识别第一个命令cat】

1. 子进程不分离的话 15s输出10次 若让主进程活6s 只能输出4次 其他的不以输出



住进

1. Process.stdin是一个可读流 child.stdin是可写流
2. 通过第三方程序传到客户端的时候 先运行程序 再请求

先./02-spawn-v3.js 再：curl <http://localhost:8080>

1. Fork 特点：
2. 独立子进程 node.js单线并行异步
3. IPC内置 process.send( ) process.on(‘massage’,…)
4. Spawn特点：使用流 通常来说第三方程序不能流就放到buffer里，【流是流动的，第三方程序产生一点改变就放到主程序（并不是把所有东西都放在缓存中

=>优点（三个）： 响应速度 内存少 管道）】

HTTP 协议

Web应用开发/常规问题

Express框架（DataStorage，MySQL）

=>Web Site

看TCP UDP网络编程