--

[内置模块API 1](#_Toc18069)

[Buffer 1](#_Toc23700)

[Buffer 1](#_Toc4999)

[.write() 1](#_Toc16714)

[.toString() 2](#_Toc18392)

[.toJSON() 2](#_Toc11678)

[Buffer.concat 2](#_Toc21804)

[.equals() 2](#_Toc17488)

[.compare() 2](#_Toc13360)

[.copy() 2](#_Toc8347)

[.slice() 2](#_Toc2920)

[.length 2](#_Toc20039)

[process 2](#_Toc393)

[path url querystring （string） 3](#_Toc8982)

[path 3](#_Toc16292)

[格式化 3](#_Toc3271)

[合并 3](#_Toc2476)

[以应用程序为起点，根据参数字符串解析出一个绝对路径 3](#_Toc16387)

[两路径之间的相对关系 3](#_Toc18457)

[目录 3](#_Toc22518)

[文件名 3](#_Toc29622)

[后缀 4](#_Toc13150)

[url(string) 4](#_Toc7394)

[parse 4](#_Toc847)

[format 4](#_Toc20936)

[resolve 4](#_Toc24846)

[queryString 4](#_Toc854)

[escape 4](#_Toc11170)

[fs 4](#_Toc18177)

[基础 4](#_Toc13071)

[rename重命名 4](#_Toc918)

[exists存在与否 5](#_Toc9820)

[utimes 5](#_Toc18857)

[futimes改变时间戳 5](#_Toc8912)

[chmod改权限 5](#_Toc23650)

[监视 watch 5](#_Toc30217)

[watchFile 5](#_Toc21935)

[unwatchFile 5](#_Toc29146)

[监视文件或目录 5](#_Toc26929)

[文件流stream 5](#_Toc26647)

[说明 5](#_Toc9046)

[创建读取流 6](#_Toc21939)

[创建写入流 6](#_Toc11904)

[writeStream使用string 6](#_Toc16349)

[writeStream使用buffer 6](#_Toc13867)

[writeStream使用readStream 6](#_Toc6914)

[管道pipe实现流读写 6](#_Toc106)

[return 7](#_Toc4828)

[目录 rmdir - mkdir - readdir 7](#_Toc22586)

[rmdir删除目录 7](#_Toc1011)

[mkdir创建目录 7](#_Toc30585)

[readdir读取文件名、文件路径名 7](#_Toc14916)

[文件读写 readFile - writeFile - appendFile 7](#_Toc21889)

[readFile读取文件 7](#_Toc26077)

[writeFile写入文件 7](#_Toc10172)

[appendFile插入到文件 8](#_Toc32186)

[文件 open -[ read - write ]- close 8](#_Toc2981)

[open打开 8](#_Toc18979)

[read 8](#_Toc19993)

[write写 8](#_Toc23384)

[close关闭 8](#_Toc8676)

[fsync 8](#_Toc5720)

[fs demo 9](#_Toc26597)

[获取某个目录下所有的文件路径 9](#_Toc4341)

[复制文件 9](#_Toc32075)

[结合promise 9](#_Toc32678)

[stream 9](#_Toc26802)

[创建可读流 9](#_Toc19111)

[创建写入流 9](#_Toc17705)

[管道流 9](#_Toc19824)

[链式流 9](#_Toc616)

[文件操作相关API 小总结 9](#_Toc6791)

[路径处理 9](#_Toc3011)

[文件状态 10](#_Toc12086)

[文件读取 10](#_Toc13208)

[mini demo of nodeJs 10](#_Toc7604)

[目录中查找某个字符串及所在文件 10](#_Toc28489)

[Node.js复制文件 11](#_Toc10681)

[readline 11](#_Toc4108)

[简要说明 11](#_Toc29339)

[demo：命令交互 12](#_Toc18185)

[demo:文件读取 13](#_Toc24416)

[line-by-line 14](#_Toc30627)

[OS 14](#_Toc25717)

[http 14](#_Toc12515)

[nodejs 模块 14](#_Toc14299)

[http请求 14](#_Toc12124)

[HTTP模块 14](#_Toc28688)

[http-request 15](#_Toc4861)

[event 15](#_Toc11014)

[event模块 15](#_Toc4272)

内置模块API

|  |  |
| --- | --- |
|  | JavaScript 标准参考教程（alpha）[阮一峰](http://www.ruanyifeng.com/)  <http://javascript.ruanyifeng.com/>  node.js中文资料导航  <https://github.com/CGerAJ/node123>  进击Node.js基础（二）  <http://www.imooc.com/learn/637> |
|  |  |
|  | Node.js 工具模块  在 Node.js 模块库中有很多好用的模块。接下来我们为大家介绍几种常用模块的使用：  1 OS 提供基本的系统操作函数。  2 Path 提供了处理和转换文件路的工具。  3 Net 用于底层的网络通信。提供了服务端和客户端的的操作。  4 DNS 用于解析域名。  5 Domain 简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的。 |

Buffer

|  |  |
| --- | --- |
| Buffer | JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。  但在处理像TCP流或文件流时，必须使用到二进制数据。因此在 Node.js中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。  在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理I/O操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。 |
|  | 使用new操作符创建一个Buffer实例，有四种方法创建一个Buffer实例：  new Buffer(size)，创建一个指buffer定大小的buffer  new Buffer(array)，根据一个字节数组来创建一个buffer  new Buffer(str[,encoding])，根据一个字符串和编码格式创建buffer，不指定编码时默认使用utf8  new Buffer(buffer)，根据buffer实例创建一个新的buffer  比如下面的代码可以创建Buffer的实例：  var buf1 = new Buffer(256);  var buf2 = new Buffer("Hello Buffer");  var buf3 = new Buffer([0x65,0x66,0x67]);  var buf4 = new Buffer(buf2);  > 有一点需要说明，使用new Buffer(size)分配的缓冲区，是未初始化的哦。那块内存里，可能是脏的，什么玩意儿都有:  var buf1 = new Buffer(256);  buf1.write('abc');  console.log(`lenght:${buf1.length}`);//256  console.log(`content:${buf1.toString()}`);//content:abc�8=@�:=�<=��  > 使用buf.fill()方法填充一下就好了(字符串是 \0 结尾的，调用了buf1.fill(0)之后，就一切安好)：  var buf1 = new Buffer(256);  buf1.fill(0);  buf1.write('abc');  console.log(`lenght:${buf1.length}`);//256  console.log(`content:${buf1.toString()}`);//content:abc  > 单个中文的字节数是3  var buffer = new Buffer('写入文件数据内容');  ll(buffer.length);//24 |
| .write() | 写入 Node 缓冲区的语法如下所示：  buf.write(string[, offset[, length]][, encoding])  参数:  string - 写入缓冲区的字符串  offset - 缓冲区开始写入的索引值，默认为 0  length - 写入的字节数，默认为 buffer.length  encoding - 使用的编码。默认为 'utf8'  返回值:  返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。  例子：  buf = new Buffer(256);  len = buf.write("www.runoob.com");  ll("写入字节数 : "+ len);  ---写入字节数 : 14 |
| .toString() | buf.toString([encoding[, start[, end]]])  参数：  encoding - 使用的编码。默认为 'utf8'  start - 指定开始读取的索引位置，默认为 0  end - 结束位置，默认为缓冲区的末尾。  返回值:  解码缓冲区数据并使用指定的编码返回字符串。  实例:  buf = new Buffer(26);for (var i = 0 ; i < 26 ; i++) { buf[i] = i + 97; }  ll( buf.toString('ascii')); // 输出: abcdefghijklmnopqrstuvwxyz  ll( buf.toString('ascii',0,5)); // 输出: abcde  ll( buf.toString('utf8',0,5)); // 输出: abcde  ll( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde |
| .toJSON() | 将 Buffer 转换为 JSON 对象:buf.toJSON()  返回值:  返回 JSON 对象  实例:  var buf = new Buffer('www.runoob.com');  var json = buf.toJSON(buf);  ll(json);//{ type: 'Buffer', data: [ 104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100 ] } |
| Buffer.concat | 缓冲区合并：Buffer.concat(list[, totalLength])  参数描述如下：  :list - 用于合并的 Buffer 对象数组列表  :totalLength - 指定合并后Buffer对象的总长度  返回值:返回一个多个成员合并的新 Buffer 对象  实例:  var buffer1 = new Buffer('菜鸟教程 ');  var buffer2 = new Buffer('www.runoob.com');  var buffer3 = Buffer.concat([buffer1,buffer2]);  ll(buffer3.toString());//菜鸟教程 www.runoob.com |
| .equals()  .compare() | buf.equals(otherBuffer)判断当前缓冲区是否和另一个相等，相等时返回true。  buf.compare(otherBuffer)比较当前缓冲区和另一个缓冲区的大小，相等返回0，小于返回-1，大于返回1  实例:  var buf1 = new Buffer('0123');  var buf2 = new Buffer('0124');  var buf3 = new Buffer('0122');  var buf4 = new Buffer('0123');  console.log(buf1.compare(buf2));//-1  console.log(buf1.compare(buf3));//1  console.log(buf1.compare(buf4));//0 |
| .copy() | 拷贝缓冲区:buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])  参数:  :targetBuffer - 要拷贝的 Buffer 对象。  :targetStart - 数字, 可选, 默认: 0  :sourceStart - 数字, 可选, 默认: 0  :sourceEnd - 数字, 可选, 默认: buffer.length  返回值:拷贝对象的长度  实例:  var buffer1 = new Buffer('ABCD');// 拷贝一个缓冲区  var buffer2 = new Buffer('012345');  var result=buffer1.copy(buffer2);  ll(result);//4  ll("buffer1 : " + buffer1.toString());//buffer1 content: ABCD  ll("buffer2 : " + buffer2.toString());//buffer2 content: ABCD45  // buffer1拷贝到buffer2，buffer2将改变 |
| .slice() | 缓冲区裁剪:buf.slice([start[, end]])  参数：  start - 数字, 可选, 默认: 0  end - 数字, 可选, 默认: buffer.length  返回值:返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。  实例:  var buffer1 = new Buffer('runoob');// 剪切缓冲区  var buffer2 = buffer1.slice(0,2);  ll(buffer2.toString());//ru |
| .length | 缓冲区长度:buf.length;  返回: Buffer 对象所占据的内存长度。  实例:  var buffer = new Buffer('www.runoob.com');// 缓冲区长度  ll("buffer length: " + buffer.length);//buffer length: 14 |

process

|  |  |
| --- | --- |
|  | process 是一个全局变量，即 global 对象的属性。  它用于描述当前Node.js 进程状态的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要 和它打交道。  process模块用来与当前进程互动，可以通过全局变量process访问，不必使用require命令加载。它是一个EventEmitter对象的实例。 |
|  | process 对象的一些最常用的成员方法 |
| 事件 | 序号 事件 & 描述  1 exit  当进程准备退出时触发。  2 beforeExit  当 node 清空事件循环，并且没有其他安排时触发这个事件。通常来说，当没有进程安排时 node 退出，但是 'beforeExit' 的监听器可以异步调用，这样 node 就会继续执行。  3 uncaughtException  当一个异常冒泡回到事件循环，触发这个事件。如果给异常添加了监视器，默认的操作（打印堆栈跟踪信息并退出）就不会发生。  4 Signal 事件  当进程接收到信号时就触发。信号列表详见标准的 POSIX 信号名，如 SIGINT、SIGUSR1 等。 |
| （1）exit事件  当前进程退出时，会触发exit事件，可以对该事件指定回调函数。这一个用来定时检查模块的状态的好钩子(hook)(例如单元测试),当主事件循环在执行完’exit’的回调函数后将不再执行,所以在exit事件中定义的定时器可能不会被加入事件列表.  process.on('exit', function () {  fs.writeFileSync('/tmp/myfile', 'This MUST be saved on exit.');  });  （2）uncaughtException事件  当前进程抛出一个没有被捕捉的意外时，会触发uncaughtException事件。  process.on('uncaughtException', function (err) {  console.error('An uncaught error occurred!');  console.error(err.stack);  }); |
|  | 序号. 属性 & 描述  1 stdout  标准输出流。  2 stderr  标准错误流。  3 stdin  标准输入流。  4 argv  argv 属性返回一个数组，由命令行执行脚本时的各个参数组成。它的第一个成员总是node，第二个成员是脚本文件名，其余成员是脚本文件的参数。  5 execPath  返回执行当前脚本的 Node 二进制文件的绝对路径。  6 execArgv  返回一个数组，成员是命令行下执行脚本时，在Node可执行文件与脚本文件之间的命令行参数。  7 env  返回一个对象，成员为当前 shell 的环境变量  8 exitCode  进程退出时的代码，如果进程优通过 process.exit() 退出，不需要指定退出码。  9 version  Node 的版本，比如v0.10.18。  10 versions  一个属性，包含了 node 的版本和依赖.  11 config  一个包含用来编译当前 node 执行文件的 javascript 配置选项的对象。它与运行 ./configure 脚本生成的 "config.gypi" 文件相同。  12 pid  当前进程的进程号。  13 title  进程名，默认值为"node"，可以自定义该值。  14 arch  当前 CPU 的架构：'arm'、'ia32' 或者 'x64'。  15 platform  运行程序所在的平台系统 'darwin', 'freebsd', 'linux', 'sunos' 或 'win32'  16 mainModule  require.main 的备选方法。不同点，如果主模块在运行时改变，require.main可能会继续返回老的模块。可以认为，这两者引用了同一个模块。 |
| (1)stdout  process.stdout用来控制标准输出，也就是在命令行窗口向用户显示内容。它的write方法等同于console.log。  exports.log = function() {  process.stdout.write(format.apply(this, arguments) + '\n');  };  （2）argv  process.argv返回命令行脚本的各个参数组成的数组。  先新建一个脚本文件argv.js。  // argv.js  console.log("argv: ",process.argv);  console.log("argc: ",process.argc);  在命令行下调用这个脚本，会得到以下结果。  node argv.js a b c  # [ 'node', '/path/to/argv.js', 'a', 'b', 'c' ]  上面代码表示，argv返回数组的成员依次是命令行的各个部分。要得到真正的参数部分，可以把argv.js改写成下面这样。  // argv.js  var myArgs = process.argv.slice(2);  console.log(myArgs); |
|  | 序号 方法 & 描述  1 abort()  这将导致 node 触发 abort 事件。会让 node 退出并生成一个核心文件。  2 chdir(directory)  改变当前工作进程的目录，如果操作失败抛出异常。  3 cwd()  返回当前进程的工作目录  4 exit([code])  使用指定的 code 结束进程。如果忽略，将会使用 code 0。  5 getgid()  获取进程的群组标识（参见 getgid(2)）。获取到得时群组的数字 id，而不是名字。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  6 setgid(id)  设置进程的群组标识（参见 setgid(2)）。可以接收数字 ID 或者群组名。如果指定了群组名，会阻塞等待解析为数字 ID 。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  7 getuid()  获取进程的用户标识(参见 getuid(2))。这是数字的用户 id，不是用户名。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  8 setuid(id)  设置进程的用户标识（参见setuid(2)）。接收数字 ID或字符串名字。果指定了群组名，会阻塞等待解析为数字 ID 。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  9 getgroups()  返回进程的群组 iD 数组。POSIX 系统没有保证一定有，但是 node.js 保证有。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  10 setgroups(groups)  设置进程的群组 ID。这是授权操作，所有你需要有 root 权限，或者有 CAP\_SETGID 能力。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  11 initgroups(user, extra\_group)  读取 /etc/group ，并初始化群组访问列表，使用成员所在的所有群组。这是授权操作，所有你需要有 root 权限，或者有 CAP\_SETGID 能力。  注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。  12 kill(pid[, signal])  发送信号给进程. pid 是进程id，并且 signal 是发送的信号的字符串描述。信号名是字符串，比如 'SIGINT' 或 'SIGHUP'。如果忽略，信号会是 'SIGTERM'。  13 memoryUsage()  返回一个对象，描述了 Node 进程所用的内存状况，单位为字节。  14 nextTick(callback)  一旦当前事件循环结束，调用回到函数。  15 umask([mask])  设置或读取进程文件的掩码。子进程从父进程继承掩码。如果mask 参数有效，返回旧的掩码。否则，返回当前掩码。  16 uptime()  返回 Node 已经运行的秒数。  17 hrtime()  返回当前进程的高分辨时间，形式为 [seconds, nanoseconds]数组。它是相对于过去的任意事件。该值与日期无关，因此不受时钟漂移的影响。主要用途是可以通过精确的时间间隔，来衡量程序的性能。  你可以将之前的结果传递给当前的 process.hrtime() ，会返回两者间的时间差，用来基准和测量时间间隔。 |
|  | # '/home/aaa'  process.chdir('/home/bbb')  process.cwd()  # '/home/bbb' |

path url querystring （string）

path

|  |  |
| --- | --- |
| www | 1 path.normalize(p)  规范化路径，注意'..' 和 '.'。  2 path.join([path1][, path2][, ...])  用于连接路径。该方法的主要用途在于，会正确使用当前系统的路径分隔符，Unix系统是"/"，Windows系统是"\"。  3 path.resolve([from ...], to)  将 to 参数解析为绝对路径。  4 path.isAbsolute(path)  判断参数 path 是否是绝对路径。  5 path.relative(from, to)  用于将相对路径转为绝对路径。  6 path.dirname(p)  返回路径中代表文件夹的部分，同 Unix 的dirname 命令类似。  7 path.basename(p[, ext])  返回路径中的最后一部分。同 Unix 命令 bashname 类似。  8 path.extname(p)  返回路径中文件的后缀名，即路径中最后一个'.'之后的部分。如果一个路径中并不包含'.'或该路径只包含一个'.' 且这个'.'为路径的第一个字符，则此命令返回空字符串。  9 path.parse(pathString)  返回路径字符串的对象。  10 path.format(pathObject)  从对象中返回路径字符串，和 path.parse 相反。 |
|  | process.cwd()取当前工作目录（Current Work Directory），这个不同于“脚本所在目录”。如果你是通过命令行调用的，那么就应该是入口脚本的目录；如果是通过其他进程调用的会继承下来。  \_\_dirname ll(\_\_dirname);//执行目录：\_\_dirname  ll(\_\_filename);//执行的js文件路径  process.cwd();//F:\HTML\test\testnode\test |
| 格式化 | /\* \* 路径解析，得到规范化的路径格式 \* path.normalize(p); \* 对window系统，目录分隔为'\' \* 对于UNIX系统，分隔符为'/' \* 针对'..'返回上一级 \* \\都被统一转换 \* \*/ ll(path.normalize(\_\_dirname + '/test/a//b//../c/util\\you.mp3')); //F:\HTML\test\testnode\test\test\a\c\util\you.mp3 |
| 合并 | /\*  \* 路径结合、合并，路径最后不会带目录分隔符  \* path.join([path1],[path2]..[pathn]);  \* ---[path1] 路径或表示目录的字符，  \* \*/ var path1 = 'path1',  path2 = 'path2//pp\\',  path3 = '../path3'; ll(path.*join*(path1, path2, path3)); //path1\path2\path3 |
| 以应用程序为起点，根据参数字符串解析出一个绝对路径 | /\*  \* 获取绝对路径  \* path.resolve(path1, [path2]..[pathn])  \* 以应用程序为起点，根据参数字符串解析出一个绝对路径  \* ---path 必须至少一个路径字符串值  \* ---[pathn] 可选路径字符串  \* \*/ ll(**path**.resolve('path1', 'path2', 'a/b**\\**c/')); //F:\HTML\test\testnode\test\path1\path2\a\b\c  ll(**path**.normalize(\_\_dirname + '/path1'+'/path2'+'/a/b**\\**c/')); ll(**path**.resolve('path1', 'path2', 'a/b**\\**c/')); //F:\HTML\test\testnode\test\path1\path2\a\b\c\ //F:\HTML\test\testnode\test\path1\path2\a\b\c |
| 两路径之间的相对关系 | /\*  \* 获取相对路径  \* path.relative(from, to);  \* 获取两路径之间的相对关系  \* from 当前路径，并且方法返回值是基于from指定到to的相对路径  \* to 到哪路径，  \* \*/ var from = 'c:\\from\\a\\',  to = 'c:/test/b'; ll(path.relative(from, to)); //..\..\test\b; 表示从from到to的相对路径 |
| 目录  文件名  后缀 | /\*\*  \* 获取路径中目录名  \*/  ll(path.**dirname**(\_\_dirname + '/test/util you.mp3'));//F:\HTML\test\testnode\test/test  /\*  \* path.**basename**(path, [ext]) 获取路径中 文件名  \* 后缀是可选的，如果加，请使用'.ext'方式来匹配，则返回值中不包括后缀名；  \* \*/  ll(path.basename(\_\_dirname + '/test/util-you.mp3')); //util-you.mp3  ll(path.basename(\_\_dirname + '/test/util-you.mp3', '.mp3')); //util-you  ll(path.basename(\_\_dirname + '/test/util-you.mp3', ".m3")); //util-you.mp3  /\*\*  \* path.**extname**(path):获取路径中的扩展名，如果没有'.'，则返回空  \*/  ll(path.extname("a.text")); //.text |
|  | ll(path.sep)//\ ll(path.delimiter)//; //path.sep:操作系统中文件分隔符； window是'\\', Unix是'/' //path.delimiter:操作系统中目录分隔符，如window是';', Unix中是':' |

url(string)

|  |  |
| --- | --- |
| parse | **/\* \* string->object \* url.parse(urlStr[, parseQueryString][, slashesDenoteHost]) \* \*/ /\*  \* parse1: urlStr  \*  \* parse2：false | true  \* 会使用querystring模块来解析URL中德查询字符串部分：**  **\* false|.:query: 'query=string',   \* true: query: {query:"string"},  \*  \* parse2 false | true  \* 会把诸如 //foo/bar 这样的URL解析为:  \* { host: 'foo', pathname: '/bar' } 而不是 { pathname: '//foo/bar' }。  \* \*/  ll**(**url**.*parse*('http://user:pass@host.com:8080/p/a/t/h?query=string&abc=123#hash')) **/\***  **{  protocol: 'http:', //请求协议，小写  slashes: true, //协议的“：”号后是否有“/”  auth: 'user:pass', //URL中的认证信息  hash: '#hash', //锚点部分（即：“#”及其后的部分）   host: 'host.com:8080', //URL主机名，包括端口信息，小写  hostname: 'host.com', //主机名，小写  port: '8080', //主机的端口号    path: '/p/a/t/h?query=string', //pathname 和 search的合集  pathname: '/p/a/t/h', //URL中路径  search: '?query=string', //查询对象，即：queryString，包括之前的问号“?”  query: 'query=string', //查询字符串中的参数部分（问号后面部分字符串），或者使用 querystring.parse() 解析后返回的对象:'query=string' or {'query':'string'}   href: 'http://user:pass@host.com:8080/p/a/t/h?query=string#hash' //解析前的完整原始 URL，协议名和主机名已转为小写 }**  **\*/** |
| format | **/\*  输入一个 URL 对象，返回格式化后的 URL 字符串。  传入的URL对象会做以下处理：    href 属性会被忽略  protocol无论是否有末尾的 : (冒号)，会同样的处理  这些协议包括 http, https, ftp, gopher, file 后缀是 :// (冒号-斜杠-斜杠).  所有其他的协议如 mailto, xmpp, aim, sftp, foo, 等 会加上后缀 : (冒号)  auth 如果有将会出现.  hostname 如果 host 属性没被定义，则会使用此属性.  port 如果 host 属性没被定义，则会使用此属性.  host 优先使用，将会替代 hostname 和port  pathname 将会同样处理无论结尾是否有/ (斜杠)  search 将会替代 query属性  query (object类型; 详细请看 querystring) 如果没有 search,将会使用此属性.  search 无论前面是否有 ? (问号)，都会同样的处理  hash无论前面是否有# (井号, 锚点)，都会同样处理 \*/ var urlObj** = {  protocol: 'http:',  slashes: **true**,  hostname: 'itbilu.com',  port: 80,  hash: '#hash',  search: '?query=string',  path: '/nodejs?query=string' } console.log(**url**.format(**urlObj**)); //http://itbilu.com:80?query=string#hash |
| resolve | **/\*url.resolve()方法用于处理URL路径，也可以用于处理锚点。\*/ url**.resolve('/one/two/three', 'four') // '/one/two/four' **url**.resolve('http://example.com/', '/one') // 'http://example.com/one' **url**.resolve('http://example.com/one', '/two') // 'http://example.com/two' |
|  | href  -----------------------------------------------------------------  host path  --------------- ----------------------------  http: // user:pass @ host.com : 8080 /p/a/t/h ?query=string #hash  ----- --------- -------- ---- -------- ------------- -----  protocol auth hostname port pathname search hash  ------------  query |

queryString

|  |  |
| --- | --- |
|  | **/\* \* querystring.stringify(obj, [sep], [eq]) \* 将JSON对象格式化为查询字符串格式的字符串，默认的分隔符为：&\= \* \*/ querystring**.*stringify*({ foo: 'bar', baz: ['qux', 'quux'], corge: '' }) //'foo=bar&baz=qux&baz=quux&corge=' **querystring**.*stringify*({foo: 'bar', baz: 'qux'}, ';', ':') //'foo:bar;baz:qux'  **/\* \* querystring.parse(str, [sep], [eq], [options]) \* 根据“&”和“=”将字符串进行分割，反序列化为JSON对象，而options包含的maxKeys默认设置为1000，如果将其设置为0则表示没这个限制 \* \*/ querystring**.*parse*('foo=bar&baz=qux&baz=quux&corge') //{ foo: 'bar', baz: ['qux', 'quux'], corge: '' } |
| escape | **/\*分别格式化和解码URL字符串\*/ ll**(**querystring**.*escape*("a <")) //a%20%3C **ll**(**querystring**.*unescape*("a%20%3C")) //a < |
|  |  |
|  |  |

## fs

基础

|  |  |  |
| --- | --- | --- |
|  | NodeJS通过fs内置模块提供对文件的操作。fs模块提供的API基本上可以分为以下三类：  > 文件属性读写: 其中常用的有fs.stat、fs.chmod、fs.chown等等。  > 文件内容读写: 其中常用的有fs.readFile、fs.readdir、fs.writeFile、fs.mkdir等等。  > 底层文件操作: 其中常用的有fs.open、fs.read、fs.write、fs.close等等。 | |
| rmdir删除目录 | //删除空目录  //只能删除内部为空的目录，不能删除文件，或内部存在文件的目录 **fs**.rmdir("/大文件/testfs", **function**(e){  console.log(e); }) | |
| mkdir创建目录 | /\*\*  \* 使用fs.mkdir创建目录：fs.mkdir(path, [mode], callback);  \* ---path, 被创建目录的完整路径及目录名；  \* ---[mode], 目录权限，默认0777  \* ---[callback(err)], 创建完目录回调函数,err错误对象  \*/  **/\*绝对路径，同级\*/ fs**.mkdir(\_\_dirname + '/fsDir', **function** (err) {  **if**(err) **throw** err;  console.log('创建目录成功') });  **/\*绝对路径 /HTML/test/testnode/haha1\*/** **fs**.mkdir("/HTML/test/testnode/haha1", **function**(e){  console.log(e); })  **/\*相对路径 hah2\*/** **fs**.mkdir("hah2", "0777",**function**(e){  //文件 mode 默认为 0777。  console.log(e); }) | 相对路径  绝对路径 |
| readdir读取文件名、文件路径名 | /\*\*  \* 使用fs.readdir读取目录，重点其回调函数中files对象；  \* 类似于ll指令，获取下一级的目录及文件名  \* fs.readdir(path, callback);  \* ---path, 要读取目录的完整路径及目录名；  \* ---callback(err, files), 读完目录回调函数：  \* err错误对象  \* files数组：存放读取到的目录中的所有文件名  \*/  /\*\*  \* fs.stat(path, callback);  \* fs.lstat(path, callback); [查看符号链接文件]  \* path, 要查看目录/文件的完整路径及名；  \* [callback(err, stats)], 操作完成回调函数；err错误对象，stat fs.Stat一个对象实例，提供如:isFile, isDirectory,isBlockDevice等方法及size,ctime,mtime等属性  \*/  fs.readdir("/HTML/test/testnode", **function**(err, files){  console.log(err)  console.log(files) }) **/\*  null  [ 'format.js', 'haha1', 'school', 'server.js', 'test' ]  \*/**  **var myPath** = "/HTML/test"; **fs**.readdir(**myPath**, **function** (err, files) {  **if** (err) {  console.error(err);  **return**;  } **else** {  files.*forEach*(**function** (file) {  **var** filePath = **path**.normalize(**myPath** + "/" + file);  **fs**.stat(filePath, **function** (err, stat) {  **if** (stat.isFile()) {  console.log(filePath + ' is: ' + 'file');  }  **if** (stat.isDirectory()) {  console.log(filePath + ' is: ' + 'dir');  }  });  });  **for** (**var** i = 0; i < files.length; i++) {  //使用闭包无法保证读取文件的顺序与数组中保存的致  (**function** () {  **var** filePath = **path**.normalize(**myPath** + "/" + files[i]);  **fs**.stat(filePath, **function** (err, stat) {  **if** (stat.isFile()) {  console.log(filePath + ' is: ' + 'file');  }  **if** (stat.isDirectory()) {  console.log(filePath + ' is: ' + 'dir');  }  });  })();  }  } }); //\HTML\test\.idea is: dir //\HTML\test\testglup is: dir //\HTML\test\testnode is: dir //\HTML\test\Sass is: dir  //\HTML\test\.idea is: dir //\HTML\test\testspm is: dir //\HTML\test\Sass is: dir //\HTML\test\testglup is: dir //\HTML\test\testnode is: dir //\HTML\test\testspm is: dir | |
| rename重命名 | /\*\*  \* 移动/重命名文件或目录  \* fs.rename(oldPath, newPath, callback)  \* ---oldPath, 原目录/文件的完整路径及名  \* ---newPath, 新目录/文件的完整路径及名；如果新路径与原路径相同，而只文件名不同，则是重命名  \* ---[callback(err)], 操作完成回调函数；err操作失败对象  \*/  //绝对路径 **fs**.rename("/HTML/test/testnode/test/testfs5.js", "testfs3.js", **function**(err){  **if**(err) { console.error(err); **return**; }  console.log('重命名成功') }) //相对路径 **fs**.rename("testfs2.js", "testfs5.js", **function**(){  **if**(err) { console.error(err); **return**; }  console.log('重命名成功') }) //目录 **fs**.rename("haha2", "haha", **function**(err){  **if**(err) { console.error(err); **return**; }  console.log('重命名成功') }) | |
| exists存在与否 | //fs.exists(path, callback(exists)) //测试某个路径下的**文件**、**目录**是否存在。回调函数包含一个参数exists，true则文件存在，否则是false。 **fs**.exists('testfs5.js', **function** (exists) {  console.log("文件：")  console.log(exists) }); **fs**.exists('/HTML/test/testnode', **function** (exists) {  console.log("绝对目录：")  console.log(exists) }); **fs**.exists('hah2', **function** (exists) {  console.log("相对目录：")  console.log(exists) });**fs**.exists('../test', **function** (exists) {  console.log("相对目录：")  console.log(exists) }); | |
| utimes | /\*\*  \* 修改文件访问时间与修改时间  \* fs.utimes(path, atime, mtime, callback);  \* ---path, 要查看目录/文件的完整路径及名；  \* ---atime, 新的访问时间  \* ---ctime, 新的修改时间  \* ---[callback(err)], 操作完成回调函数；err操作失败对象  \*/ **fs**.utimes(\_\_dirname + '/test.txt', **new Date**(), **new Date**(), **function** (err) {  **if**(err) {  console.error(err);  **return**;  }  **fs**.stat(\_\_dirname + '/test.txt', **function** (err, stat) {  console.log('访问时间: ' + stat.atime.*toString*() + '; **\n**修改时间：' + stat.mtime);  console.log(stat.mode);  }) }); | |
| futimes改变时间戳 | **/\* 更改一个文件所提供的文件描述符引用的文件的时间戳。 fs.futimes(path, atime, mtime, callback)  path 路径  atime  mtime  callback 回调 \*/** | |
| chmod改权限 | **/\* 该方法以异步的方式来改写文件的读写权限。操作完成后的回调只接收一个参数，可能会出现异常信息。  fs.chmod(path, mode, callback)  -| path 文件路径  -| mode 读写权限[如：777][如：0666 8进制，权限：所有用户可读、写]  -| callback 回调 \*/**  **var oldFilename** = "./processId.txt",  **newFilename** = "./processIdOld.txt"; **fs**.chmod(**oldFilename**, 777, **function** (err) {  **fs**.rename(**oldFilename**, **newFilename**, **function** (err) {  **fs**.lstat(**newFilename**, **function** (err, stats) {  **var** isSymLink = stats.isSymbolicLink();  });  }); }); | |

监视 watch

|  |  |
| --- | --- |
| watchFile | **/\*  \* 对文件进行监视，并且在监视到文件被修改时执行处理  \* fs.watchFile(filename, [options], listener);  \* -| filename, 完整路径及文件名；  \* -| [options], persistent true表示持续监视，不退出程序；interval 单位毫秒，表示每隔多少毫秒监视一次文件  \* -| listener, 文件发生变化时回调  \* curr：一个fs.Stat对象，被修改后文件，  \* prev：一个fs.Stat对象，表示修改前对象  \* \*/**  **fs**.watchFile(\_\_dirname + '/test.txt', {interval: 20}, **function** (curr, prev) {  **if**(**Date**.*parse*(prev.ctime) == 0) {  console.log('文件被创建!');  } **else if**(**Date**.*parse*(curr.ctime) == 0) {  console.log('文件被删除!')  } **else if**(**Date**.*parse*(curr.mtime) != **Date**.*parse*(prev.mtime)) {  console.log('文件有修改');  } }); **fs**.watchFile(\_\_dirname + '/test.txt', **function** (curr, prev) {  console.log('这是第二个watch,监视到文件有修改'); }); |
| unwatchFile | /\*\*  \* 取消对文件进行监视  \* fs.unwatchFile(filename, [listener]);  \* ---filename, 完整路径及文件名；  \* ---[listener], 要取消的监听器事件，如果不指定，则取消所有监听处理事件  \*/ **var listener** = **function** (curr, prev) {  console.log('我是监视函数') } **fs**.unwatchFile(\_\_dirname + '/test.txt', **listener**); |
| 监视文件或目录 | // 对文件或目录进行监视，并且在监视到修改时执行处理； // fs.watch返回一个fs.FSWatcher对象，拥有一个close方法，用于停止watch操作； // 当fs.watch有文件变化时，会触发fs.FSWatcher对象的change(err, filename)事件，err错误对象，filename发生变化的文件名 // fs.watch(filename, [options], [listener]);  /\*\*  \* filename, 完整路径及文件名或目录名；  \* [listener(event, filename], 监听器事件，有两个参数：event 为rename表示指定的文件或目录中有重命名、删除或移动操作或change表示有修改，filename表示发生变化的文件路径  \*/  **var fsWatcher** = **fs**.watch(\_\_dirname + '/test', **function** (event, filename) {  //ll(event) });  //ll(fsWatcher instanceof FSWatcher);  **fsWatcher**.on('change', **function** (event, filename) {  console.log(filename + ' 发生变化') });  //30秒后关闭监视 setTimeout(**function** () {  console.log('关闭')  **fsWatcher**.close(**function** (err) {  **if**(err) {  console.error(err)  }  console.log('关闭watch')  }); }, 30000); |

createReadStream createWriteStream

|  |  |  |  |
| --- | --- | --- | --- |
| 说明 | | **/\*  \* 流，在应用程序中表示一组有序的、有起点有终点的字节数据的传输手段；  \* Node.js中实现了stream.Readable/stream.Writeable接口的对象进行流数据读写；以上接口都继承自EventEmitter类，因此在读/写流不同状态时，触发不同事件；  \* 关于流读取：Node.js不断将文件一小块内容读入缓冲区，再从缓冲区中读取内容；  \* 关于流写入：Node.js不断将流数据写入内在缓冲区，待缓冲区满后再将缓冲区写入到文件中；重复上面操作直到要写入内容写写完；  \* readFile、read、writeFile、write都是将整个文件放入内存而再操作，而流则是文件一部分数据一部分数据操作；  \*  \* -----------------------流读取-------------------------------------  \* 读取数据对象：  \* fs.ReadStream 读取文件  \* http.IncomingMessage 客户端请求或服务器端响应  \* net.Socket Socket端口对象  \* child.stdout 子进程标准输出  \* child.stdin 子进程标准入  \* process.stdin 用于创建进程标准输入流  \* Gzip、Deflate、DeflateRaw 数据压缩  \*  \* 触发事件：  \* readable 数据可读时  \* data 数据读取后  \* end 数据读取完成时  \* error 数据读取错误时  \* close 关闭流对象时  \*  \* 读取数据的对象操作方法：  \* read 读取数据方法  \* setEncoding 设置读取数据的编  \* pause 通知对象众目停止触发data事件  \* resume 通知对象恢复触发data事件  \* pipe 设置数据通道，将读入流数据接入写入流；  \* unpipe 取消通道  \* unshift 当流数据绑定一个解析器时，此方法取消解析器  \*  \* ------------------------流写入-------------------------------------  \* 写数据对象：  \* fs.WriteStream 写入文件对象  \* http.clientRequest 写入HTTP客户端请求数据  \* http.ServerResponse 写入HTTP服务器端响应数据  \* net.Socket 读写TCP流或UNIX流，需要connection事件传递给用户  \* child.stdout 子进程标准输出  \* child.stdin 子进程标准入  \* Gzip、Deflate、DeflateRaw 数据压缩  \*  \* 写入数据触发事件：  \* drain 当write方法返回false时，表示缓存区中已经输出到目标对象中，可以继续写入数据到缓存区  \* finish 当end方法调用，全部数据写入完成  \* pipe 当用于读取数据的对象的pipe方法被调用时  \* unpipe 当unpipe方法被调用  \* error 当发生错误  \*  \* 写入数据方法：  \* write 用于写入数据  \* end 结束写入，之后再写入会报错；  \*/** | |
| 创建读取流 | | /\*\*  \* 创建一个readStream（文件读取流，输入流）对象。（可读流）  \* fs.createReadStream(path, [options])  \* path 文件路径  \* [options] (object) 数组对象包含以下属性 { flags: 'r', encoding: null, fd: null, mode: 0666, autoClose: true }  \*  \* flags:指定文件操作，默认'r',读操作；  \* encoding,指定读取流编码；可以是 'utf8', 'ascii', 或 'base64'三种格式。  \* autoClose, 是否读取完成后自动关闭，默认true；如果autoClose为false时，文件描述符将不会被关闭，即使他们报错了。最好把它关闭掉 并确保不会出现文件描述符泄漏。如果autoClose为true时(默认的行为)，对错误或结束的文件描述符将自动关闭。  \* start指定文件开始读取位置；  \* end指定文件开始读结束位置；可以通过start 和 end 设置 文件 可读取的字节数范围，而不是读取整个文件。如果start 和 end都被包含的情况下 ，将从0开始。  \*/  > 基础事件  var rs = fs.createReadStream(\_\_dirname + '/test.txt', { start: 0, end: 2 });  rs.on('open', function (fd) { console.log('开始读取文件'); });//open是ReadStream对象中表示文件打开时事件  rs.on('data', function (data) { console.log(data.toString()); });  rs.on('end', function () { console.log('读取文件结束') });  rs.on('close', function () { console.log('文件关闭'); });  rs.on('error', function (err) { console.error(err); });  > 暂停和恢复文件读取  var rs = fs.createReadStream(\_\_dirname + '/test.txt');  rs.on('open', function () { console.log('开始读取文件'); });  rs.on('data', function (data) { console.log(data.toString()); });  ll("stop read;")  rs.pause();//暂停  setTimeout(function () {  console.log("reopen read;")  rs.resume();//恢复  }, 2000);  > 截取  fs.createReadStream('sample.txt', { start: 90, end: 99 });  //该例子将读取一个100k的文件中的最后10十字节内容。 | |
| 创建写入流 | | /\*\*  \* fs.createWriteStream(path, [options])  \* path 文件路径  \* [options] flags:指定文件操作，默认'w'  \* encoding:指定读取流编码  \* start:指定写入文件的位置  \*/  **/\* ws.write(chunk, [encoding], [callback]);  \* chunk, 可以为Buffer对象或一个字符串，要写入的数据  \* [encoding], 编码  \* [callback], 写入后回调  \*/  /\* ws.end([chunk], [encoding], [callback]);  \* [chunk], 要写入的数据  \* [encoding], 编码  \* [callback], 写入后回调  \*/**  **//fs.createWriteStream(path, [options])**  **// 1. path 文件路径**  **// 2. option (object) //{ flags: 'w', encoding: null, mode: 0666 }**  **// ##option包含一个启动选项用来允许在某些文件的开头位置写数据。**  **// ##flags 默认值为w，如果你想修改一个文件，而不是取代它需要把flags改成R +。** | |
|  | | 1. writeStream使用string   var ws = fs.createWriteStream(\_\_dirname + '/test.txt', { start: 0 });  ws.write("JavaScript代码风格检查工具 - JSCS 介绍", 'utf8');  ws.end('!');   1. writeStream使用buffer   var ws = fs.createWriteStream(\_\_dirname + '/test.txt', { start: 0 });  var buffer = new Buffer('我喜欢你');  ws.write(buffer, 'utf8', function (err) {  console.log('写入完成，回调函数没有参数');  });  ws.end('!');   1. writeStream使用readStream   //使用流完成复制文件操作  var readSm = fs.createReadStream(\_\_dirname + '/test.txt')  var writeSm = fs.createWriteStream(\_\_dirname + '/test2.txt');  readSm.on('data', function (data) {  console.log("event : readsm data");  console.log("输入流读取数据：" + data);  writeSm.write(data);  });  readSm.on('end', function () {  console.log("event : readsm end")  console.log('输入流读取完毕');  writeSm.end('完成', function () {  console.log('文件全部写入完成')  });  });  writeSm.on('open', function (fd) {  console.log("event : writesm open");  console.log('要写入的数据文件已经打开，文件描述符是： ' + fd);  });  // 上面的写法有一些问题，如果写入的速度跟不上读取的速度，有可能导致数据丢失。  // 正常的情况应该是，写完一段，再读取下一段，如果没有写完的话，就让读取流先暂停，等写完再继续，于是代码可以修改为：  var fs = require('fs');  var readStream = fs.createReadStream('/path/to/source');  var writeStream = fs.createWriteStream('/path/to/dest');  readStream.on('data', function(chunk) { // 当有数据流出时，写入数据  if (writeStream.write(chunk) === false) { // 如果没有写完，暂停读取流  readStream.pause();  }  });  writeStream.on('drain', function() { // 写完后，继续读取  readStream.resume();  });  readStream.on('end', function() { // 当没有数据时，关闭数据流  writeStream.end();  });  // 或者使用更直接的pipe  // pipe自动调用了data,end等事件  fs.createReadStream('/path/to/source').pipe(fs.createWriteStream('/path/to/dest')); | |
|  | Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对http 服务器发起请求的request 对象就是一个 Stream，还有stdout（标准输出）。  Node.js，Stream 有四种流类型：  Readable - 可读操作。  Writable - 可写操作。  Duplex - 可读可写操作.  Transform - 操作被写入数据，然后读出结果。  所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：  data - 当有数据可读时触发。  end - 没有更多的数据可读时触发。  error - 在接收和写入过程中发生错误时触发。  finish - 所有数据已被写入到底层系统时触发。 | |
| pipe | | |
| 管道流 | 管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。  我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。  以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。  var readerStream = fs.createReadStream('input.txt'); // 创建一个可读流  var writerStream = fs.createWriteStream('output.txt'); // 创建一个可写流  readerStream.pipe(writerStream); // 管道读写操作 | |
| 链式流 | 链式是通过连接输出流到另外一个流并创建多个对个流操作链的机制。链式流一般用于管道操作。  接下来我们就是用管道和链式来压缩和解压文件。  创建 compress.js 文件, 代码如下：  var fs = require("fs");  var zlib = require('zlib');  // 压缩 input.txt 文件为 input.txt.gz  fs.createReadStream('input.txt')  .pipe(zlib.createGzip())  .pipe(fs.createWriteStream('input.txt.gz'));    console.log("文件压缩完成。"); | |
| 执行完以上操作后，我们可以看到当前目录下生成了 input.txt 的压缩文件 input.txt.gz。  接下来，让我们来解压该文件，创建 decompress.js 文件，代码如下：  var fs = require("fs");  var zlib = require('zlib');  // 解压 input.txt.gz 文件为 input.txt  fs.createReadStream('input.txt.gz')  .pipe(zlib.createGunzip())  .pipe(fs.createWriteStream('input.txt'));    console.log("文件解压完成。"); | |
|  | var http = require('http');  var fs = require('fs');    http.createServer(function(req, res) {    // The filename is simple the local directory and tacks on the requested url    var filename = \_\_dirname+req.url;      // This line opens the file as a readable stream    var readStream = fs.createReadStream(filename);      // This will wait until we know the readable stream is actually valid before piping    readStream.on('open', function () {      // This just pipes the read stream to the response object (which goes to the client)      readStream.pipe(res);    });      // This catches any errors that happen while creating the readable stream (usually invalid names)    readStream.on('error', function(err) {      res.end(err);    });  }).listen(8080); | |
| 管道pipe实现流读写 | | /\*\*  \* rs.pipe(destination, [options]);  \* ---destination 必须一个可写入流数据对象  \* ---[opations] end 默认为true，表示读取完成立即关闭文件；  \*/  **var rs** = **fs**.createReadStream(\_\_dirname + '/test/Until You.mp3'); **var ws** = **fs**.createWriteStream(\_\_dirname + '/test/untiyou.mp3'); **rs**.pipe(**ws**); **rs**.on('data', **function** (data) {  console.log('数据可读') }); **rs**.on('end', **function** () {  console.log('文件读取完成');  //ws.end('再见') }); | |
|  | |  | |
| return | | //关于WriteStream对象的write方法返回一个布尔类型，当缓存区中数据全部写满时，返回false;  //表示缓存区写满，并将立即输出到目标对象中  //第一个例子  var ws = fs.createWriteStream(\_\_dirname + '/test/test.txt');  for (var i = 0; i < 10000; i++) {  var w\_flag = ws.write(i.toString());  //当缓存区写满时，输出false  console.log(w\_flag);  }  //第二个例子  var ws = fs.createWriteStream(\_\_dirname + '/test/untiyou.mp3');  var rs = fs.createReadStream(\_\_dirname + '/test/Until You.mp3');  rs.on('data', function (data) {  var flag = ws.write(data);  console.log(flag);  });  //系统缓存区数据已经全部输出触发drain事件  ws.on('drain', function () {  console.log('系统缓存区数据已经全部输出。')  }); | |
|  | |  | |

readFile writeFile appendFile

|  |  |
| --- | --- |
| readFile读取文件 | //读取文件readFile函数 //readFile(filename,[options],callback); /\*\*  \* filename, 必选参数，文件名  \* [options],可选参数，可指定flag（文件操作选项，如r+ 读写；w+ 读写，文件不存在则创建）及encoding属性  \* callback 读取文件后的回调函数，参数默认第一个err,第二个data 数据 (Buffer)  \*/  **fs**.readFile(\_\_dirname + '/message.txt', {flag: 'r+', encoding: 'utf8'}, **function** (err, data) {  **if**(err) {  console.error(err);  **return**;  }  **fs**.appendFile('message2.txt', data, **function** (err) {  **if** (err) **throw** err;  console.log('The "data to append" was appended to file!');  });  console.log(data); }); |
| writeFile写入文件 | // fs.writeFile(filename,data,[options],callback); /\*\*  \* filename: 必选参数，文件名  \* data, 写入的数据，可以字符或一个Buffer对象  \* [options],flag,mode(权限),encoding  \* callback 读取文件后的回调函数，参数默认第一个err,第二个data 数据  \*/ **var w\_data** = **new** Buffer('这是一段通过fs.writeFile函数写入的内容；**\r\n**'); **fs**.writeFile(\_\_dirname + '/message2.txt', **w\_data**, {flag: 'a'}, **function** (err) {  **if**(err) {  console.error(err);  } **else** {  console.log('写入成功');  } }); |
| appendFile插入到文件 | //该方法以异步的方式将 data 插入到文件里，如果文件不存在会自动创建。data可以是任意字符串或者缓存。 //fs.appendFile(filename, data, [options], callback) // 1. filename {String} // 2. data {String | Buffer} // 3. options {Object} // encoding {String | Null} default = 'utf8' // mode {Number} default = 438 (aka 0666 in Octal) // flag {String} default = ‘a' // 4. callback {Function} **fs**.appendFile('message.txt', 'data to append', **function** (err) {  **if** (err) **throw** err;  console.log('The "data to append" was appended to file!'); }); |
| 读写文件 | var source = fs.readFileSync('/path/to/source', {encoding: 'utf8'});  fs.writeFileSync('/path/to/dest', source); |
|  | 单字节编码  有时候，我们无法预知需要读取的文件采用哪种编码，因此也就无法指定正确的编码。比如我们要处理的某些CSS文件中，有的用GBK编码，有的用UTF8编码。虽然可以一定程度可以根据文件的字节内容猜测出文本编码，但这里要介绍的是有些局限，但是要简单得多的一种技术。  首先我们知道，如果一个文本文件只包含英文字符，比如Hello World，那无论用GBK编码或是UTF8编码读取这个文件都是没问题的。这是因为在这些编码下，ASCII0~128范围内字符都使用相同的单字节编码。  反过来讲，即使一个文本文件中有中文等字符，如果我们需要处理的字符仅在ASCII0~128范围内，比如除了注释和字符串以外的JS代码，我们就可以统一使用单字节编码来读取文件，不用关心文件的实际编码是GBK还是UTF8。以下示例说明了这种方法。  1. GBK编码源文件内容：  var foo = '中文';  2. 对应字节：  76 61 72 20 66 6F 6F 20 3D 20 27 D6 D0 CE C4 27 3B  3. 使用单字节编码读取后得到的内容：  var foo = '{乱码}{乱码}{乱码}{乱码}';  4. 替换内容：  var bar = '{乱码}{乱码}{乱码}{乱码}';  5. 使用单字节编码保存后对应字节：  76 61 72 20 62 61 72 20 3D 20 27 D6 D0 CE C4 27 3B  6. 使用GBK编码读取后得到内容：  var bar = '中文';  这里的诀窍在于，不管大于0xEF的单个字节在单字节编码下被解析成什么乱码字符，使用同样的单字节编码保存这些乱码字符时，背后对应的字节保持不变。  NodeJS中自带了一种binary编码可以用来实现这个方法，因此在下例中，我们使用这种编码来演示上例对应的代码该怎么写。  function replace(pathname) {  var str = fs.readFileSync(pathname, 'binary');  str = str.replace('foo', 'bar');  fs.writeFileSync(pathname, str, 'binary');  } |

open [read write] close

|  |  |
| --- | --- |
| open打开 | //fs.open(filename, flags, [mode], [callback(err,fd)]) //以异步的方式打开文件。 // 1.filename:被认为是默认存在的（即使该路径下的文件是不存在的） // 2.flags:标识 可能 或 可能不在网络文件系统下运行。 // 'r' - 以读取模式打开文件。 // 'r+' - 以读写模式打开文件。 // 'rs' - 使用同步模式打开并读取文件。指示操作系统忽略本地文件系统缓存。 // 'rs+' - 以同步的方式打开，读取 并 写入文件。 // 注意：这不是让fs.open变成同步模式的阻塞操作。如果想要同步模式请使用fs.openSync()。 // 'w' - 以读取模式打开文件，如果文件不存在则创建 // 'wx' - 和 ' w ' 模式一样，如果文件存在则返回失败 // 'w+' - 以读写模式打开文件，如果文件不存在则创建 // 'wx+' - 和 ' w+ ' 模式一样，如果文件存在则返回失败 // 'a' - 以追加模式打开文件，如果文件不存在则创建 // 'ax' - 和 ' a ' 模式一样，如果文件存在则返回失败 // 'a+' - 以读取追加模式打开文件，如果文件不存在则创建 // 'ax+' - 和 ' a+ ' 模式一样，如果文件存在则返回失败 // 3.mode:用于创建文件时给文件制定权限，默认0666 // 4.callback:打开文件后回调函数，参数默认第一个err,第二个fd为一个整数，表示打开文件返回的文件描述符，window中又称文件句柄  **fs**.open("testfs5.js", "w", **function**(e,fd){  console.log(e);  console.log(fd); }) |
| read | /\*\*  \* fs.read(fd, buffer, offset, length, position, callback)  \* fd, 使用fs.open打开成功后返回的文件描述符  \* buffer, 一个Buffer对象，v8引擎分配的一段内存  \* offset, 整数，向缓存区中写入时的初始位置，以字节为单位  \* length, 整数，读取文件的长度  \* position, 整数，读取文件初始位置；文件大小以字节为单位  \* callback(err, bytesRead, buffer), 读取执行完成后回调函数，  \* bytesRead实际读取字节数，被读取的缓存区对象  \*/ **/\*  \* open(--read(--)--) to buffer 通过open打开的文件，通过read将被打开文件的内容读取 到 缓冲区  \* demo: [message2.txt]  中华0123456789  人民qwertyuiop  共和asdfghjkl  国zxcvbnm  \*/ fs**.open(\_\_dirname + '/message2.txt', 'r', **function** (err, fd) {  **if** (err) {  console.error(err);  **return**;  } **else** {  **var** buffer = **new** Buffer(255);  //每一个汉字utf8编码是3个字节，英文是1个字节  **fs**.read(fd, buffer, 0, 9, 3, **function** (err, bytesRead, buffer) {  **if** (err) {  **throw** err;  } **else** {  console.log(bytesRead);//9  console.log(buffer.*slice*(0, bytesRead).*toString*());//华012345  //读取完后，再使用fd读取时，基点是基于上次读取位置计算；  **fs**.read(fd, buffer, 0, 9, **null**, **function** (err, bytesRead, buffer) {  console.log(bytesRead);//9  console.log(buffer.*slice*(0, bytesRead).*toString*());  //6789  //人  });  }  });  } }); |
| write写 | /\*\*  \* fs.write(fd, buffer, offset, length, position, callback);  \* fd, 使用fs.open打开成功后返回的文件描述符  \* buffer, 一个Buffer对象，v8引擎分配的一段内存  \* offset, 整数，从缓存区中读取时的初始位置，以字节为单位  \* length, 整数，从缓存区中读取数据的字节数  \* position, 整数，写入文件初始位置；  \* callback(err, written, buffer), 写入操作执行完成后回调函数  \* err错误信息  \* written实际写入字节数，  \* buffer被读取的缓存区对象  \*/  fs.write(fd, buffer, offset, length[, position], [callback(err, bytesWritten, buffer)])  fs.write(fd, data[, position[, encoding]], [callback(err, written, string)])  **fs**.open(\_\_dirname + '/message2.txt', 'a', **function** (err, fd) {  **if**(err) {  console.error(err);  **return**;  } **else** {  **var** buffer = **new** Buffer('写入文件数据内容');  //写入'入文件'三个字  **fs**.write(fd, buffer, 3, 9, 12, **function** (err, written, buffer) {  **if**(err) {  console.log('写入文件失败');  console.error(err);  **return**;  } **else** {  console.log(buffer.*toString*());  //写入'数据内'三个字  **fs**.write(fd, buffer, 12, 9, **null**, **function** (err, written, buffer) {  console.log(buffer.*toString*());  })  }  });  } }); |
| close关闭 | //fs.close(path, [callback(err)]) // fd: 路径。 // callback: 回调 |
| fsync | // 使用fs.write写入文件时，操作系统是将数据读到内存，再把数据写入到文件中，当数据读完时并不代表数据已经写完，因为有一部分还可能在内在缓冲区内。 // 因此可以使用fs.fsync方法将内存中数据写入文件；--刷新内存缓冲区；  //fs.fsync(fd, [callback]) /\*\*  \* fd, 使用fs.open打开成功后返回的文件描述符  \* [callback(err, written, buffer)], 写入操作执行完成后回调函数，written实际写入字节数，buffer被读取的缓存区对象  \*/  **fs**.open(\_\_dirname + '/test.txt', 'a', **function** (err, fd) {  **if**(err)  **throw** err;  **var** buffer = **new** Buffer('我爱nodejs编程');  **fs**.write(fd, buffer, 0, 9, 0, **function** (err, written, buffer) {  console.log(written.*toString*());  **fs**.write(fd, buffer, 9, buffer.length - 9, **null**, **function** (err, written) {  console.log(written.*toString*());  **fs**.fsync(fd);  **fs**.close(fd);  })  }); }); |

fs demo

|  |  |
| --- | --- |
| 获取某个目录下所有的文件路径 | **fs**.readdir(process.cwd(), **function**(err, files){  **if**(err){  **return** console.log(err);  }  **for**(**var** i = 0; i < files.length; i += 1){  console.log(files[i]);  } }); |
| 复制文件 | var fs=require("fs");  function copyFile(){  var fs = require('fs');  fs.readFile('test.txt', 'utf-8', function(err, data) {  if (err) {  console.log("读取失败");  } else {  writeFile(data)  return data;  }  });  }  function writeFile(data){  fs.writeFile("test2.txt",data,'utf8',function(error){  if(error){  throw error;  }else{  console.log("文件已保存");  }  });  }  copyFile(); |
| 结合promise | var path = require('path');  var Promise = require('bluebird');  var fs = Promise.promisifyAll(require('fs'));  ll(1)  fs.readdirAsync(\_\_dirname).then(function (data) {  console.log(data);  return data.filter(n => path.extname(n) == ".js");  }).then(function (data) {  console.log(data)  });  ll(2); |

文件操作相关API 小总结

|  |  |
| --- | --- |
| 路径处理 | var path = require('path');  1.首先，我们需要注意的文件路径的规范化，nodejs给我们提供了Path模块，normolize方法能帮我们规范化路径：  path.normalize('/foo/bar/nor/faz/..'); -> /foo/bar/nor  2.当然还有join合并路径:  path.join('/foo', 'bar', 'baz/asdf', 'quux', '..'); ->/foo/bar/baz/asdf  3.解析路径  path.resolve('/foo/bar', './baz'); ->/foo/bar/baz  path.resolve('/foo/bar', '/tmp/file/'); ->/tmp/file  4.在两个相对路径间查找相对路径  path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb'); ->../../impl/bbb  5.抽离路径 [目录 文件名(有无后缀) 后缀]  path.dirname('/foo/bar/baz/asdf/quux.txt'); ->/foo/bar/baz/asdf  path.basename('/foo/bar/baz/asdf/quux.html') ->quux.html  甚至你还还可以将后缀名去掉，只需要在basename中传入第二个参数，参数为后缀名，例如：  path.basename('/foo/bar/baz/asdf/quux.html', '.html'); ->quux  当然文件路径中可能会存在各种不同的文件，我们不可能硬编码后缀来得到我们想要的结果，  所以有一个方法能帮我们得到后缀名：  path.extname('/a/b/index.html'); // => '.html'  path.extname('/a/b.c/index'); // => ''  path.extname('/a/b.c/.'); // => ''  path.extname('/a/b.c/d.'); // => '.' |
| 文件状态 | var fs = require('fs');  fs.exists(path, function(exists) {});//判断文件是否存在(异步回调) fs.existsSync(path);//同步操作 fs.stat(path, function(err, stats) {  if (err) { throw err;}  console.log(stats); });//读取文件状态信息  //控制台输出states的内容大致： { dev: 234881026, ino: 95028917, mode: 33188, nlink: 1, uid: 0, gid: 0, rdev: 0, size: 5086, blksize: 4096, blocks: 0, atime: Fri, 18 Nov 2011 22:44:47 GMT, mtime: Thu, 08 Sep 2011 23:50:04 GMT, ctime: Thu, 08 Sep 2011 23:50:04 GMT }  //同时，stats还具有一些方法，比如： stats.isFile(); stats.isDirectory(); stats.isBlockDevice(); stats.isCharacterDevice(); stats.isSymbolicLink(); stats.isFifo(); stats.isSocket(); |
| 文件读取 | fs.open('/path/to/file', 'r', function(err, fd) { /\* todo\*/ });  第二个参数为操作类型：  r : 只读  r+ : 读写  w : 重写文件  w+ : 重写文件，如果文件不存在则创建  a : 读写文件，在文件末尾追加，如果文件不存在则创建  a+ : 读写文件，在文件末尾追加，如果文件不存在则创建 |
| 下面为一个读取文件的小例子：  var fs = require('fs');  fs.open('test.txt', 'r', function (err, fd) {  if (err) { throw err }  var readBuffer = new Buffer(1024);  fs.read(fd, readBuffer, 0, readBuffer.length, 0, function (err, readBytes, buf) {  if (err) { throw err; }  console.log('just read ' + readBytes + ' bytes');  console.log(buf.toString());  if (readBytes > 0) {  console.log(readBuffer.slice(0, readBytes));  }  });  }); |
| /\*  test.txt：  写入文件数据内入文件  数  readBytes：35  buf.toString()显示的值与readBuffer.slice(0,readBytes)相同；  \*/ |
| 下面为一个写文件的小例子：  var fs = require('fs');  fs.open('./my\_file.txt', 'a', function (err, fd) {  if (err) { throw err; }  var writeBuffer = new Buffer('hello, world!'),  bufferPosition = 0,  bufferLength = writeBuffer.length,  filePosition = null;  fs.write(fd, writeBuffer, bufferPosition, bufferLength, filePosition, function (err, written) {  if (err) { throw err; }  console.log('wrote ' + written + ' bytes');  });  }); |
| var fs = require('fs');  fs.open('./my\_file.txt', 'a', function (err, fd) {  if (err) { throw err; }  fs.write(fd, "中国", function (err, written) {  if (err) { throw err; }  console.log('wrote ' + written + ' bytes');  });  }); |
| fs.openAsync('./my\_file.txt', 'a').then(function (fd) {  return fs.writeAsync(fd, "中国");  }).then(function (leng) {  console.log('wrote ' + leng + ' bytes');  });  fs.openAsync('./my\_file11.txt', 'r').then(function (fd) {  return fs.writeAsync(fd, "中国");  }).then(function (leng) {  console.log('wrote ' + leng + ' bytes');  }).catch(function (e){  console.log(e);  }); |
| 对于文件的读写操作，我们不应该忘记在这些操作都完成之后执行关闭操作，即close(); 下面是一个封装的方法，其中就包括了文件的后期关闭操作，使用起来方便：  var fs = require('fs');  function openAndWriteToSystemLog(writeBuffer, callback) {  fs.open('./my\_file', 'a', function(err, fd) {  if (err) { return callback(err); }  function notifyError(err) {  fs.close(fd, function() {  callback(err);  });  }  var bufferOffset = 0,  bufferLength = writeBuffer.length,  filePosition = null;  fs.write( fd, writeBuffer, bufferOffset, bufferLength, filePosition,function(err, written) {  if (err) { return notifyError(err); }  fs.close(fd, function() {  callback(err);  });  });  });  }  openAndWriteToSystemLog(new Buffer('writing this string'),function(err) {  if (err) {  console.log("error while opening and writing:", err.message);  return;  }  console.log('All done with no errors');  }); |

mini demo of nodeJs

|  |  |
| --- | --- |
| 小文件拷贝 | 我们使用NodeJS内置的fs模块简单实现这个程序如下。  var fs = require('fs');  function copy(src, dst) {  fs.writeFileSync(dst, fs.readFileSync(src));  }  function main(argv) {  copy(argv[0], argv[1]);  }  main(process.argv.slice(2));  以上程序使用fs.readFileSync从源路径读取文件内容，并使用fs.writeFileSync将文件内容写入目标路径。  豆知识： process是一个全局变量，可通过process.argv获得命令行参数。由于argv[0]固定等于NodeJS执行程序的绝对路径，argv[1]固定等于主模块的绝对路径，因此第一个命令行参数从argv[2]这个位置开始。 |
| 大文件拷贝 | 上边的程序拷贝一些小文件没啥问题，但这种一次性把所有文件内容都读取到内存中后再一次性写入磁盘的方式不适合拷贝大文件，内存会爆仓。对于大文件，我们只能读一点写一点，直到完成拷贝。因此上边的程序需要改造如下。  var fs = require('fs');  function copy(src, dst) {  fs.createReadStream(src).pipe(fs.createWriteStream(dst));  }  function main(argv) {  copy(argv[0], argv[1]);  }  main(process.argv.slice(2));  以上程序使用fs.createReadStream创建了一个源文件的只读数据流，并使用fs.createWriteStream创建了一个目标文件的只写数据流，并且用pipe方法把两个数据流连接了起来。连接起来后发生的事情，说得抽象点的话，水顺着水管从一个桶流到了另一个桶。 |
|  | 当内存中无法一次装下需要处理的数据时，或者一边读取一边处理更加高效时，我们就需要用到数据流。  NodeJS中通过各种Stream来提供对数据流的操作。  以上边的大文件拷贝程序为例，我们可以为数据来源创建一个只读数据流，示例如下：  var rs = fs.createReadStream(pathname);  rs.on('data', function (chunk) {  doSomething(chunk);  });  rs.on('end', function () {  cleanUp();  });  //Stream基于事件机制工作，所有Stream的实例都继承于NodeJS提供的[EventEmitter](http://nodejs.org/api/events.html)。  上边的代码中data事件会源源不断地被触发，不管doSomething函数是否处理得过来。代码可以继续做如下改造，以解决这个问题。  var rs = fs.createReadStream(src);  rs.on('data', function (chunk) {  rs.pause();  doSomething(chunk, function () {  rs.resume();  });  });  rs.on('end', function () {  cleanUp();  });  以上代码给doSomething函数加上了回调，因此我们可以在处理数据前暂停数据读取，并在处理数据后继续读取数据。  此外，我们也可以为数据目标创建一个只写数据流，示例如下：  var rs = fs.createReadStream(src);  var ws = fs.createWriteStream(dst);  rs.on('data', function (chunk) {  ws.write(chunk);  });  rs.on('end', function () {  ws.end();  });  我们把doSomething换成了往只写数据流里写入数据后，以上代码看起来就像是一个文件拷贝程序了。但是以上代码存在上边提到的问题，如果写入速度跟不上读取速度的话，只写数据流内部的缓存会爆仓。我们可以根据.write方法的返回值来判断传入的数据是写入目标了，还是临时放在了缓存了，并根据drain事件来判断什么时候只写数据流已经将缓存中的数据写入目标，可以传入下一个待写数据了。因此代码可以改造如下：  var rs = fs.createReadStream(src);  var ws = fs.createWriteStream(dst);  rs.on('data', function (chunk) {  if (ws.write(chunk) === false) {  rs.pause();  }  });  rs.on('end', function () {  ws.end();  });  ws.on('drain', function () {  rs.resume();  });  以上代码实现了数据从只读数据流到只写数据流的搬运，并包括了防爆仓控制。  因为这种使用场景很多，例如上边的大文件拷贝程序，NodeJS直接提供了.pipe方法来做这件事情，其内部实现方式与上边的代码类似。 |
| 目录遍历 | /\*同步遍历\*/  function travel(dir, callback) {  fs.readdirSync(dir).forEach(function (file) {  var pathname = path.join(dir, file);  if (fs.statSync(pathname).isDirectory()) {  travel(pathname, callback);  } else {  callback(pathname);  }  });  }  /\*  可以看到，该函数以某个目录作为遍历的起点。遇到一个子目录时，就先接着遍历子目录。遇到一个文件时，就把文件的绝对路径传给回调函数。回调函数拿到文件路径后，就可以做各种判断和处理。因此假设有以下目录：  - /home/user/  - foo/  x.js  - bar/  y.js  z.css  使用以下代码遍历该目录时，得到的输入如下。  travel('/home/user', function (pathname) {  console.log(pathname);  });  ------------------------  /home/user/foo/x.js  /home/user/bar/y.js  /home/user/z.css  \*/ |
|  | // 异步遍历  // 如果读取目录或读取文件状态时使用的是异步API，目录遍历函数实现起来会有些复杂，但原理完全相同。travel函数的异步版本如下。  function travel(dir, callback, finish) {  fs.readdir(dir, function (err, files) {  (function next(i) {  if (i < files.length) {  var pathname = path.join(dir, files[i]);  fs.stat(pathname, function (err, stats) {  if (stats.isDirectory()) {  travel(pathname, callback, function () {  next(i + 1);  });  } else {  callback(pathname, function () {  next(i + 1);  });  }  });  } else {  finish && finish();  }  }(0));  });  }  // 这里不详细介绍异步遍历函数的编写技巧，在后续章节中会详细介绍这个。总之我们可以看到异步编程还是蛮复杂的。 |
| 目录中查找某个字符串及所在文件 | Node.js实现在目录中查找某个字符串及所在文件  整个目录下大概有40几M，文件无数，由于时间久了， 记不清那个字符串具体在哪个文件，于是。强大，亮瞎双眼的Node.js闪亮登场。  两个参数：第一个参数为“文件夹名称” 第二个参数为“你要查找的字符串”  如图：    打印出文件路径，完事，收工。速度实在是彪悍，亮瞎双眼。。。如果采用java全文搜索，你惨了... |
|  | **var path** = **require**("path"); **var fs** = **require**("fs");  **var filePath** = **process**.argv[2];//文件名 **var lookingForString** = **process**.argv[3];//被搜索的字符串 **recursiveReadFile**(**filePath**);  **function recursiveReadFile**(fileName){  **if**(!**fs**.existsSync(fileName)) **return**;  **if**(**isFile**(fileName)){  **check**(fileName);  }  **if**(**isDirectory**(fileName)){  **var** files = **fs**.readdirSync(fileName);  files.**forEach**(**function**(val,key){  **var** temp = **path**.join(fileName,val);  **if**(**isDirectory**(temp)) **recursiveReadFile**(temp);  **if** (**isFile**(temp)) **check**(temp);  })  } } **function check**(fileName){  **var** data = **readFile**(fileName);  **var** exc = **new** RegExp(**lookingForString**);  **if**(exc.test(data))  console.log(fileName);  } **function isDirectory**(fileName){  **if**(**fs**.existsSync(fileName)) **return fs**.statSync(fileName).isDirectory(); } **function isFile**(fileName){  **if**(**fs**.existsSync(fileName)) **return fs**.statSync(fileName).isFile(); } **function readFile**(fileName){  **if**(**fs**.existsSync(fileName)) **return fs**.readFileSync(fileName,"utf-8"); } |
| Node.js复制文件 | Node.js 本身并没有提供直接复制文件的 API，如果想用 Node.js 复制文件或目录，需要借助其他的 API 来实现。复制单个的文件可以直接用 readFile、writeFile，这样比较简便。如果是复制一个目录下的所有文件，目录下可能还包含了子目录，那么此时就需要用到更高级点的 API 了。  流  流是 Node.js 移动数据的方式，Node.js 中的流是可读/可写的，HTTP 和文件系统模块都有用到流。在文件系统中，使用流来读取文件的时候，对于一个大文件可能并不会一次性读取完，而是会分好几次读取完，读取的时候会响应数据事件，在文件没读取完的时候就可以对读取的数据进行操作。同理，在写入流的时候也和读取时一样，大文件并不会一次性写入。这种移动数据的方式是非常高效的，尤其是对于大文件而言，使用流比等待将大文件全部读取完再来操作文件要快得多。  管道  如果想在读取流和写入流的时候做完全的控制，可以使用数据事件。但对于单纯的文件复制来说读取流和写入流可以通过管道来传输数据。 |
| **var fs** = **require**('fs'), **stat** = **fs**.stat; **/\* 复制目录中的所有文件包括子目录 @param{ String } 需要复制的目录 @param{ String } 复制到指定的目录 \*/ var copy** = **function** (src, dst) { **/\* 读取目录中的所有文件/目录\*/  fs**.readdir(src, **function** (err, paths) {  **if** (err) {  **throw** err;  }  paths.forEach(**function** (path) {  **var** \_src = src + '/' + path, \_dst = dst + '/' + path, readable, writable;  **stat**(\_src, **function** (err, st) {  **if** (err) {  **throw** err;  }  **/\* 判断是否为文件\*/  if** (st.isFile()) { **/\* 创建读取流\*/** readable = **fs**.createReadStream(\_src); **/\* 创建写入流\*/** writable = **fs**.createWriteStream(\_dst); **/\* 通过管道来传输流\*/** readable.pipe(writable);  } **else if** (st.isDirectory()) {**/\* 如果是目录则递归调用自身\*/  exists**(\_src, \_dst, **copy**);  }  });  });  }); }; **/\* 在复制目录前需要判断该目录是否存在，不存在需要先创建目录\*/ var exists** = **function** (src, dst, callback) {  **fs**.exists(dst, **function** (exists) { **/\* 已存在\*/  if** (exists) {  callback(src, dst);  } **/\* 不存在\*/ else** {  **fs**.mkdir(dst, **function** () {  callback(src, dst);  });  }  }); }; **/\* 复制目录\*/ exists**('./src', './build', **copy**); |

readline

简要说明

|  |  |
| --- | --- |
|  | readline模块提供了一个接口用来从一个可读的流中读取数据（比如process.stdin），每次读取一行。  在nodejs的命令行模式下默认引入了readline模块，但是如果要执行Node.js脚本文件的话还是需要显示引入readline模块。  注意：readline适合动态交互式流处理，类似于C++命令行中的cin和cout。不是我们平常意义上说的按行读取文件。当流被创建时，readline只会监听新增的行而忽略已有的行。  注意：readline适合动态交互式流处理，类似于C++命令行中的cin和cout。不是我们平常意义上说的按行读取文件。当流被创建时，readline只会监听新增的行而忽略已有的行。 |
| 参数 | readline.createInterface(options)  input <Readable> 要监听的可读流。该选项是必需的。  output <Writable> 要写入逐行读取数据的可写流。  completer <Function> 一个可选的函数，用于 Tab 自动补全。  terminal <boolean> 如果 input 和 output 应被当作一个 TTY，且要写入 ANSI/VT100 转换的代码，则设为 true。 默认为实例化时在 output 流上检查 isTTY。  historySize <number> 保留的历史行数的最大数量。 设为 0 可禁用历史记录。 默认为 30。 该选项只有当 terminal 被用户或内部 output 设为 true 时才有意义，否则历史缓存机制不会被初始化。  prompt - 要使用的提示字符串。默认为 '> '。  crlfDelay <number> 如果 \r 与 \n 之间的延迟超过 crlfDelay 毫秒，则 \r 和 \n 都会被当作换行分隔符。 默认为 100 毫秒。 crlfDelay 的范围为 [100, 2000]。  removeHistoryDuplicates <boolean> If true, when a new input line added to the history list duplicates an older one, this removes the older line from the list. Defaults to false. |
| 方法 | rl.close()  关闭接口实例 (Interface instance), 放弃控制输入输出流。”close” 事件会被触发  rl.pause()  暂停 readline 的输入流 (input stream), 如果有需要稍后还可以恢复。  rl.prompt([preserveCursor])  为用户输入准备好readline，将现有的setPrompt选项放到新的一行，让用户有一个新的地方开始输入。将preserveCursor设为true来防止光标位置被重新设定成0。  rl.question(query, callback)  预先提示指定的query，然后用户应答后触发指定的callback。 显示指定的query给用户后，当用户的应答被输入后，就触发了指定的callback  rl.resume()  恢复 readline 的输入流 (input stream).  rl.setPrompt(prompt)  设置提示符，例如当你在命令行运行 node 时，你会看到(prompt)  rl.write(data[, key]) |
| 事件 | Class: Interface  Instances of the readline.Interface class are constructed using the readline.createInterface() method. Every instance is associated with a single input Readable stream and a single output Writable stream. The output stream is used to print prompts for user input that arrives on, and is read from, the input stream.  readline.Interface实例是通过readline.createInterface()函数来构造。每一个实例都关联了一个唯一的可读输入流和一个唯一的可写输出流。输出流是为用户即将进行的输入打印提示信息，并且从输入流中读取数据。  Event: ‘close’  The ‘close’ event is emitted when one of the following occur:  The rl.close() method is called and the readline.Interface instance has relinquished control over the input and output streams;  The input stream receives its ‘end’ event;  The input stream receives <ctrl>-D to signal end-of-transmission (EOT);  The input stream receives <ctrl>-C to signal SIGINT and there is no SIGINT event listener registered on the readline.Interface instance.  The listener function is called without passing any arguments.  The readline.Interface instance should be considered to be “finished” once the ‘close’ event is emitted.  当以下的一种情况发生时，’close’事件会被触发：  rl.close() 方法被调用后，readline.Interaface实例便会放弃输入输出流的控制权；  输入流收到它自己的’end’事件；  输入流收到<ctrl>-D，发送传输结束信号（EOF，end-of-transmission）；  输入流收到<ctrl>-C，发送SIGINT信号，并且在readline.Interfacce实例上并没有注册SIGINT事件的监听器；  侦听函数被调用的时候，不传递任何参数。  一旦’close’事件被触发，readline.Interface实例就会被认为是已经终止了。  Event: ‘line’  The ‘line’ event is emitted whenever the input stream receives an end-of-line input (\n, \r, or \r\n). This usually occurs when the user presses the <Enter>, or <Return> keys.  The listener function is called with a string containing the single line of received input.  无论何时，当输入流收到行结束符（\n, \r, 或者 \r\n）的时候，’line’事件都好被触发。它通常发生在用户按下<Enter>或<Return>按键时。  当输入单独的一行的字符串的时候，监听器函数会被调用。  这是一个监听用户输入的利器。  rl.on('line', (cmd) => {  console.log(`You just typed: ${cmd}`);  });  Event: ‘pause’  The ‘pause’ event is emitted when one of the following occur:  The input stream is paused.  The input stream is not paused and receives the SIGCONT event. (See events SIGTSTP and SIGCONT)  The listener function is called without passing any arguments.  ‘pause’事件会在下面任意一种情况发生时被触发：  输入流停止。  输入流没有停止，但是且接收到SIGCONT事件（参阅SIGTSTP 和 SIGCONT 事件）  该监听函数被调用时不会传递任何参数  Event: ‘resume’  The ‘resume’ event is emitted whenever the input stream is resumed.  The listener function is called without passing any arguments.  当输入流被唤醒时，’resume’事件会被触发。  该监听函数被调用时不会传递任何参数。  Event: ‘SIGCONT’  The ‘SIGCONT’ event is emitted when a Node.js process previously moved into the background using -Z (i.e. SIGTSTP) is then brought back to the foreground using fg(1).  If the input stream was paused before the SIGTSTP request, this event will not be emitted.  当使用<ctrl>-Z将Node.js进程移动到后台时，SIGCONT事件将会被触发。然后可以使用 fg(1) 命令将它移动到前台。  如果输入流在收到SIGTSIP之前就已经被暂停了，那么该事件就不会触发。  Event: ‘SIGINT’  The ‘SIGINT’ event is emitted whenever the input stream receives a -C input, known typically as SIGINT. If there are no ‘SIGINT’ event listeners registered when the input stream receives a SIGINT, the ‘pause’ event will be emitted.  The listener function is invoked without passing any arguments.  当输入流收到<ctrl>-C命令时，我们熟知的SIGINT事件就会被触发。如果当时并没有注册任何SIGINT事件监听器，那么当输入流接收到SIGINT信号时，’pause’事件就会被触发。  该监听函数被调用时不会传递任何参数。  rl.on('SIGINT', () => {  rl.question('Are you sure you want to exit?', (answer) => {  if(answer.match(/^y(es)?$/i)) {  rl.pause();  }  });  });  Event: ‘SIGTSTP’  The ‘SIGTSTP’ event is emitted when the input stream receives a -Z input, typically known as SIGTSTP. If there are no SIGTSTP event listeners registered when the input stream receives a SIGTSTP, the Node.js process will be sent to the background.  When the program is resumed using fg(1), the ‘pause’ and SIGCONT events will be emitted. These can be used to resume the input stream.  The ‘pause’ and ‘SIGCONT’ events will not be emitted if the input was paused before the process was sent to the background.  The listener function is invoked without passing any arguments.  当输入流收到<ctrl>-Z命令时，我们熟知的SIGTSTP事件就会被触发。如果没有注册SIGTSTP时间监听器，那么当输入流收到SIGTSTP事件时，Node.js进程就将会被移动到后台。  如果输入在进程被发送到后台之前就已经被暂停了，那么’pause’和’SIGCONT’事件就将不会被触发。  该监听函数被调用时不会传递任何参数。  SIGTSTP和SIGSTOP的区别  SIGTSTP与SIGSTOP都是使进程暂停（都使用SIGCONT让进程重新激活）。唯一的区别是SIGSTOP不可以捕获。  捕捉SIGTSTP后一般处理如下：  1）处理完额外的事  2）恢复默认处理  3）发送SIGTSTP信号给自己。（使进程进入suspend状态。） |

demo：命令交互

|  |  |
| --- | --- |
| 基本用法 | var readline = require('readline');  var rl = readline.createInterface({  input: process.stdin,  output: process.stdout  });  rl.question('What do you think of Node.js? ', (answer) => {  console.log('Thank you for you valuable feedback: ' + answer);  rl.close();  });  rl.on("close", function () {  console.log("after line,do process.exit(0)");  process.exit(0); // 结束程序，执行与否貌似无作用  }); |
|  |
|  | > 过程：  createInterface创建了一个接口实例  question方法来询问姓名  监听readline的close事件，  > Note:  Once this code is invoked, the node.js application will not terminate until the readline.Interface is closed because the interface waits for data to be received on the input stream.  > 注意：  在createInterface里，我们需要传入标准输入输出作为数据的输入输出流  一旦该代码被调用，Node.js程序将不会终止知道readline接口被关闭，因为接口会在输入流中不停的等待数据。因此需要在question方法的回调函数里，我们可以获取到用户的输入并进行处理，同时我们进行了close操作来结束程序，否则程序不会结束  在close事件的监听里，我们执行了process.exit(0)来使程序退出的操作，因为readline模块只要一开始获取用户输入就不会结束，必须使用这种直接的方式来结束程序 |
| 小型命令行 | const readline = require('readline');  const rl = readline.createInterface({  input: process.stdin,  output: process.stdout,  prompt: 'TinyCLI>'  });  rl.prompt();  rl.on('line', (line) => {  switch(line.trim()) {  case 'hello':  console.log('world');  break;  default:  console.log(`Say what? I might have heard '${line.trim()}'`);  break;  }  rl.prompt();  }).on('close', () => {  console.log('Have a great day, Bye!');  //没有exit则会继续监听输入，线程并不会退出  //process.exit(0);  }).on('SIGINT', () => {  console.log('Ctrl+C => SIGINT');  });  ctrl-c 发送 SIGINT 信号给前台进程组中的所有进程。常用于终止正在运行的程序。  ctrl-z 发送 SIGTSTP 信号给前台进程组中的所有进程，常用于挂起一个进程。  ctrl-d 不是发送信号，而是表示一个特殊的二进制值，表示 EOF。  ctrl-\ 发送 SIGQUIT 信号给前台进程组中的所有进程，终止前台进程并生成 core 文件。  kill -SIGCONT PID 发送 SIGCONT信号，让一个停止(stopped)的进程继续执行.  更多信号表请参看 http://blog.csdn.net/chy555chy/article/details/52626515 |
| 输入与输出 | var readline = require('readline');  var rl = readline.createInterface({  input: process.stdin,  output: process.stdout  });  rl.on('line', function(line){  switch(line.trim()) {  case 'copy':  console.log("复制");  break;  case 'hello':  rl.write("Write");  console.log('world!');  break;  case 'close':  rl.close();  break;  default:  console.log('没有找到命令！');  break;  }  });  rl.on('close', function() {  console.log('bye bye');//先触发pause事件  process.exit(0);  });  rl.on('pause', function() {  console.log('Readline 输入暂停.');  });  ‘line’事件，这个事件就是在用户输完一行，按下回车后就会触发的事件，它会将用户输入的数据通过回调函数传回来，可在此方法里处理用户输入的数据 |
| 输入输出 | var readline = require('readline');  var rl = readline.createInterface(process.stdin, process.stdout);  rl.setPrompt('Test> ');  rl.prompt();  rl.on('line', function(line) {  switch(line.trim()) {  case 'copy':  console.log("复制");  break;  case 'hello':  console.log('world!');  break;  case 'close':  rl.close();  break;  default:  console.log('没有找到命令！');  break;  }  rl.prompt();  });  rl.on('close', function() {  console.log('bye bye!');  process.exit(0);  });  运行截图如下：  IMG_256  这个实例里出现了两个新方法  方法setPrompt(promat)，就是给每一行设置一个提示符，就好比window命令行的> ，我们这里设置的是Test>  prompt()可以算是最重要的方法了，因为它才体现了Readline的核心作用，以行为单位读取数据，prompt方法就是在等待用户输入数据  这里又监听了’line’ 事件，因为promat方法调用一次就只会读取一次数据，所以，在这个方法又调用了一次promat方法，这样就可以继续读取用户输入，从而达到一种命令行的效果 |
| 复杂输入处理 | Linux技法中，awk应该是必知必会，但实在是记不住这么怪异的参数格式……  ###正文  nodejs自带的readline可以实现逐行处理，每行一个json，处理方式非常灵活  命令行里这样用：  cat logfile.txt | node main.js  main.js中这样写：  var readline = require('readline');  var rl = readline.createInterface({  input: process.stdin,  output: process.stdout,  terminal: false //这个参数很重要  });  rl.on('line', function(line){  var obj = JSON.parse(line);  //按需处理  console.log(...);  });  通过“管道”把文本内容灌入nodejs的标准输入，简洁清晰，处理过程可以任意复杂!  注意 terminal: false 这个参数很重要，如果不指定false，node会认为输入来自TTY终端（命令行），这种情况下Tab(\t)符号是自动补全的意思，结果就是： 接收到的line值，所有的Tab号都消失了！ |

demo:文件读取

|  |  |
| --- | --- |
| Read File Stream Line-by-Line | const readline = require('readline');  const fs = require('fs');  var rl = readline.createInterface({  input: fs.createReadStream('test.txt'),  terminal: true  });  rl.on('line', (line) => {  console.log('line = ', line);  }).on('close', () => {  console.log('readline is closed');  });  当流创建的时候文件里的所有数据便会被取出来，存储在Interface对象的以数组的方式存储在history属性中。然后当输入流中新增了一行的时候，才会触发 ‘line’ 事件。 |
| 读写1 | var readline = require('readline');  var fs = require('fs');  var os = require('os');  var fRead = fs.createReadStream("test.txt");  var fWrite = fs.createWriteStream("test.out.txt");  var objReadline = readline.createInterface({  input: fRead,  // 这是另一种复制方式，这样on('line')里就不必再调用fWrite.write(line)，当只是纯粹复制文件时推荐使用  // 但文件末尾会多算一次index计数 sodino.com  // output: fWrite,  // terminal: true  });  var index = 1;  objReadline.on('line', (line) => {  var tmp = 'line' + index.toString() + ':' + line;  fWrite.write(tmp + os.EOL); // 下一行  console.log(index, line);  index++;  });  objReadline.on('close', () => {  console.log('readline close...');  }); |
| 读写2 | var readline = require('readline');  var fs = require('fs');  var os = require('os');  var fRead = fs.createReadStream("test.txt");  var fWrite = fs.createWriteStream("test.out.txt");  var rl = readline.createInterface({  input: fRead,  output: fWrite,  terminal: true  });  var enableWriteIndex = true;  fRead.on('end', () => {  console.log('end');  enableWriteIndex = false;  });  var index = 1;  fWrite.write('line' + index.toString() + ':');  rl.on('line', (line) => {  console.log(index, line);  if (enableWriteIndex) {  // 由于readline::output是先写入后调用的on('line')事件，  // 所以已经读取文件完毕时就不需要再写行号了... sodino.com  index++;  var tmp = 'line' + index.toString() + ':';  fWrite.write(tmp);  }  });  rl.on('close', () => {  console.log('readline close...');  }); |

line-by-line

|  |  |
| --- | --- |
|  | is a NodeJS module that helps you reading large text files, line by line, without buffering the files into memory. |
|  | Class: LineReader(path [, options])  > path specifies the file to read  > options is an object with the following defaults:  { encoding: 'utf8',  skipEmptyLines: false }  encoding can be 'utf8', 'ascii', or 'base64'.  If skipEmptyLines set to true, empty lines don't trigger a 'line' event.  You can also pass start and end position to read from file region:  { encoding: 'utf8',  skipEmptyLines: true,  start: 1000 } |
|  | var LineByLineReader = require('line-by-line'),  lr = new LineByLineReader('big\_file.txt');  lr.on('error', function (err) {  // 'err' contains error object  });  lr.on('line', function (line) {  // 'line' contains the current line without the trailing newline character.  });  lr.on('end', function () {  // All lines are read, file is closed now.  }); |

OS

|  |  |
| --- | --- |
|  | Node.js os 模块提供了一些基本的系统操作函数。我们可以通过以下方式引入该模块：  var os = require("os") |
|  | 序号 方法 & 描述  1 os.tmpdir()  返回操作系统的默认临时文件夹。  2 os.endianness()  返回 CPU 的字节序，可能的是 "BE" 或 "LE"。  3 os.hostname()  返回操作系统的主机名。  4 os.type()  返回操作系统名  5 os.platform()  返回操作系统名  6 os.arch()  返回操作系统 CPU 架构，可能的值有 "x64"、"arm" 和 "ia32"。  7 os.release()  返回操作系统的发行版本。  8 os.uptime()  返回操作系统运行的时间，以秒为单位。  9 os.loadavg()  返回一个包含 1、5、15 分钟平均负载的数组。  10 os.totalmem()  返回系统内存总量，单位为字节。  11 os.freemem()  返回操作系统空闲内存量，单位是字节。  12 os.cpus()  返回一个对象数组，包含所安装的每个 CPU/内核的信息：型号、速度（单位 MHz）、时间（一个包含 user、nice、sys、idle 和 irq 所使用 CPU/内核毫秒数的对象）。  13 os.networkInterfaces()  获得网络接口列表。  序号 属性 & 描述  1 os.EOL  定义了操作系统的行尾符的常量。 |
|  |  |

Node.js 异步异常的处理与domain模块解析

|  |  |
| --- | --- |
|  | <https://cnodejs.org/topic/516b64596d38277306407936> |
|  |  |
|  |  |
| 异步异常的特点 | 由于node的回调异步特性，无法通过try catch来捕捉所有的异常：  try {  process.nextTick(function () {  foo.bar();  });  } catch (err) {  //can not catch it  }  而对于web服务而言，其实是非常希望这样的：  //express风格的路由  app.get('/index', function (req, res) {  try {  //业务逻辑  } catch (err) {  logger.error(err);  res.statusCode = 500;  return res.json({ success: false, message: '服务器异常' });  }  });  如果try catch能够捕获所有的异常，这样我们可以在代码出现一些非预期的错误时，能够记录下错误的同时，友好的给调用者返回一个500错误。可惜，try catch无法捕获异步中的异常。所以我们能做的只能是：  app.get('/index', function (req, res) {  // 业务逻辑 }  );  process.on('uncaughtException', function (err) {  logger.error(err);  });  这个时候，虽然我们可以记录下这个错误的日志，且进程也不会异常退出，但是我们是没有办法对发现错误的请求友好返回的，只能够让它超时返回。 |
|  | NodeJS通过process对象提供了捕获全局异常的方法，示例代码如下  process.on('uncaughtException', function (err) {  console.log('Error: %s', err.message);  });  setTimeout(function (fn) {  fn();  });  -- Console ------------------------------  Error: undefined is not a function  虽然全局异常有个地方可以捕获了，但是对于大多数异常，我们希望尽早捕获，并根据结果决定代码的执行路径。我们用以下HTTP服务器代码作为例子：  function async(request, callback) {  // Do something.  asyncA(request, function (err, data) {  if (err) {  callback(err);  } else {  // Do something  asyncB(request, function (err, data) {  if (err) {  callback(err);  } else {  // Do something  asyncC(request, function (err, data) {  if (err) {  callback(err);  } else {  // Do something  callback(null, data);  }  });  }  });  }  });  }  http.createServer(function (request, response) {  async(request, function (err, data) {  if (err) {  response.writeHead(500);  response.end();  } else {  response.writeHead(200);  response.end(data);  }  });  });  以上代码将请求对象交给异步函数处理后，再根据处理结果返回响应。这里采用了使用回调函数传递异常的方案，因此async函数内部如果再多几个异步函数调用的话，代码就变成上边这副鬼样子了。为了让代码好看点，我们可以在每处理一个请求时，使用domain模块创建一个子域（JS子运行环境）。在子域内运行的代码可以随意抛出异常，而这些异常可以通过子域对象的error事件统一捕获。于是以上代码可以做如下改造：  function async(request, callback) {  // Do something.  asyncA(request, function (data) {  // Do something  asyncB(request, function (data) {  // Do something  asyncC(request, function (data) {  // Do something  callback(data);  });  });  });  }  http.createServer(function (request, response) {  var d = domain.create();  d.on('error', function () {  response.writeHead(500);  response.end();  });  d.run(function () {  async(request, function (data) {  response.writeHead(200);  response.end(data);  });  });  });  可以看到，我们使用.create方法创建了一个子域对象，并通过.run方法进入需要在子域中运行的代码的入口点。而位于子域中的异步函数回调函数由于不再需要捕获异常，代码一下子瘦身很多。  陷阱  无论是通过process对象的uncaughtException事件捕获到全局异常，还是通过子域对象的error事件捕获到了子域异常，在NodeJS官方文档里都强烈建议处理完异常后立即重启程序，而不是让程序继续运行。按照官方文档的说法，发生异常后的程序处于一个不确定的运行状态，如果不立即退出的话，程序可能会发生严重内存泄漏，也可能表现得很奇怪。  但这里需要澄清一些事实。JS本身的throw..try..catch异常处理机制并不会导致内存泄漏，也不会让程序的执行结果出乎意料，但NodeJS并不是存粹的JS。NodeJS里大量的API内部是用C/C++实现的，因此NodeJS程序的运行过程中，代码执行路径穿梭于JS引擎内部和外部，而JS的异常抛出机制可能会打断正常的代码执行流程，导致C/C++部分的代码表现异常，进而导致内存泄漏等问题。  因此，使用uncaughtException或domain捕获异常，代码执行路径里涉及到了C/C++部分的代码时，如果不能确定是否会导致内存泄漏等问题，最好在处理完异常后重启程序比较妥当。而使用try语句捕获异常时一般捕获到的都是JS本身的异常，不用担心上诉问题。  小结  本章介绍了JS异步编程相关的知识，总结起来有以下几点：  不掌握异步编程就不算学会NodeJS。  异步编程依托于回调来实现，而使用回调不一定就是异步编程。  异步编程下的函数间数据传递、数组遍历和异常处理与同步编程有很大差别。  使用domain模块简化异步代码的异常处理，并小心陷阱。 |
| domain | 在node v0.8+ 版本的时候，发布了一个模块domain。  这个模块做的就是try catch所无法做到的：捕捉异步回调中出现的异常。  于是乎，我们上面那个无奈的例子好像有了解决的方案：  通过中间件的形式，引入domain来处理异步中的异常。当然，domain虽然捕捉到了异常，但是还是由于异常而导致的堆栈丢失会导致内存泄漏，所以出现这种情况的时候还是需要重启这个进程的，有兴趣的同学可以去看看domain - middleware这个domain中间件。  var domain = require('domain');  //引入一个domain的中间件，将每一个请求都包裹在一个独立的domain中  //domain来处理异常  app.use(function (req, res, next) {  var d = domain.create();  //监听domain的错误事件  d.on('error', function (err) {  logger.error(err);  res.statusCode = 500;  res.json({ sucess: false, messag: '服务器异常' });  d.dispose();  });  d.add(req);  d.add(res);  d.run(next);  });  app.get('/index', function (req, res) {  //处理业务  }); |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

http

|  |  |
| --- | --- |
| nodejs 模块 |  |
| http请求 | 浏览器运行过程：   1. chrome搜索自身的DNS缓存[查看自身缓存：chrome://net-internals/#dns] 2. 搜索操作系统自身的DNS缓存（浏览器没有找到缓存，或者缓存已经失效） 3. 读取本地的HOST文件 4. 浏览器发起一个DNS的系统调用   宽带运营商服务器查看自身缓存  运营商服务器发起一个迭代DNS解析的请求  运营商服务器把结果返回，操作系统内核同时缓存起来  操作系统内核把结果返回浏览器  浏览器取到了www.baidu.com对应的IP地址   1. 浏览器获取域名对应的IP地址后，发起HTTP“三次握手” 2. 发起TCP请求-网卡-TCP/IP协议栈-防火墙-web服务端 3. TCP/IP连接建立起来后，浏览器就可以向服务器发送HTTP请求了，例如：用HTTP的GET方式请求一个根域里的一个域名，协议可以采用HTTP 1.0的一个协议 |
| HTTP模块 | 回调、同步/异步、I/O、单线程/多线程、堵塞/非堵塞、事件、事件驱动、基于事件驱动的回调、事件循环   1. 回调   回调是异步编程最基本的方法，对于nodejs来说，需要按顺序执行异步逻辑的时候，一般采用后续传递的方式，也就是将后续逻辑封装在回调函数中，作为起始函数的参数逐层去嵌套，通过这种方式来让程序按照我们所期待的方式走完整个流程。   1. 异步/同步   同步就是执行一个任务，后一个任务等待前一个任务结束后再执行。程序的执行顺序与任务的排列顺序是一致的。  异步中：程序的执行顺序与任务的排列顺序是不一致的 |
| http-request | //http.request(options[,callback])  //---string|object //---function **/\*  \* host:域名，IP地址  \* hostname:别名  \* port:远端服务器的端口80  \* localAddress:  \* socketPath:绑定的本地接口  \* method:指定http请求的方法的字符串get  \* path:请求的路径，默认是根路径  \* headers:包含请求头的一个对象  \* auth:认证  \* agent:代理  \* keepAlive:套接字false  \* keepAliveMsecs:  \* \*/ var http**=require("http") **var querystring**=require("querystring"); //待发动的数据 **var postData**=**querystring**.*stringify*({  'content':'我爱你',  'cid':333 }); //request的属性 **var options**={  hostname:"www.imooc.com",  port:"80",  path:"/course/document",  method:"POST",  headers:{  "Accept":"",  "Accept-Encoding":"",  "Accept-Language":"",  "Cache-Control":"",  "Connection":"",  "Content-Length":**postData**.length,//提交内容字符的长度  "Content-Type":"",  "Cookie":"",  "Host":"",  "...":""  } } //定义req，添加回调函数 **var req**=**http**.request(**options**,**function**(res){  console.log("status:::"+res.statusCode);  console.log("headers:::"+**JSON**.*stringify*(res.headers));  res.on("data",**function**(chunk){  console.log(Buffer.*isBuffer*(chunk));  console.log(**typeof** chunk);   });  res.on("end",**function**(){  console.log("OK ! ");  }) }); **req**.on("error",**function**(e){  console.log("Error:::"+ e.message); }); //执行 **req**.write(**postData**); **req**.end(); |

event

|  |  |
| --- | --- |
| event模块 | var **EventEmitter** = require("events").EventEmitter; var **life** = new **EventEmitter**(); **life**.on("eventName", function (value) {  console.log("name:::" + value); })  **life**.on("eventName2", function (value) {  console.log("name2:::" + value); }) **life**.emit("eventName", "hello ");  ---对于一个事件不要设置超过10个监听器，可能会导致内存泄露，也通过life.setMaxListeners(11)来扩大值、  ---移除：removeListener不针对匿名函数，只能针对聚名函数实现事件取出  ---批量移除所有：life.removeAllListeners("eventName")  ---移除所有：life.removeAllListeners()  ---获取某个eventName上绑定的具体回调：life.listeners("eventName")  ---获取绑定事件个数：EventEmitter.listenerCount(life,"eventName") |
|  | node.js 如何继承 events 自定义事件及触发函数  <http://yijiebuyi.com/blog/1a8cb4bf8510f8e9267c2873b4cdda33.html>  javascript 中构造函数的私有变量在原型链定义函数可以共用  <http://yijiebuyi.com/blog/8555bd59dfdfb6fae246d0d682411940.html>  node.js 下使用 util.inherits 来实现继承  <http://yijiebuyi.com/blog/ea4b2a30f73596a08ce85211626b68e5.html>  ++  一介布衣  http://yijiebuyi.com/category/nodejs/4.html |