### node全局变量

\_\_filename：当前正在执行的脚本文件的绝对路径

\_\_dirname : 表示当前执行脚本所在的目录。

延时器：

setTimeout(cb, ms)：全局函数在指定的毫秒(ms)数后执行指定函数(cb)。：setTimeout() 只执行一次指定函数。

clearTimeout( t ) 全局函数用于停止一个之前通过 setTimeout() 创建的定时器。 参数 t 是通过 setTimeout() 函数创建的定时器。

setInterval(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。

定时器：

setInterval(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)，会不停的调用函数，直到clearInterval() 被调用或窗口被关闭。

clearInterval(t) 函数来清除定时器。

console.log(“%d”,111) //111 类似于PHP的sprintf

console.time(标签名)

…//执行一些代码

console.timeEnd(标签名)

//标签名：0.17ms

### 常用工具

* 1. util

util.inherits(constructor, superConstructor)是一个实现对象间原型继承的函数；

只能继承原型上的属性和方法

util.inspect(obj) 将任意对象转为字符串

util.isArray(object) 判断参数是否是数组

util.isRegExp(object) 判断参数是否是正则表达式

### 工具模块

* 1. OS模块

var os = require("os")

方法：

|  |  |
| --- | --- |
| **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.hostname()** 返回操作系统的主机名。 |
| 3 | **os.type()** 返回操作系统名 |
| 4 | **os.platform()** 返回操作系统名 |
| 5 | **os.totalmem()** 返回系统内存总量，单位为字节。 |
| 6 | **os.freemem()** 返回操作系统空闲内存量，单位是字节。 |

属性：

os.EOL ：定义了操作系统的行尾符的常量。

* 1. Path模块

var path = require("path")

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 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.resolve(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。 |

属性：

|  |  |
| --- | --- |
| **序号** | **属性 & 描述** |
| 1 | **path.sep** 平台的文件路径分隔符，'\\' 或 '/'。 |
| 2 | **path.delimiter** 平台的分隔符, ; or ':'. |

### 模块

* 1. exports

exports对象是当前模块的导出对象，用于导出模块的公有方法和属性，别的模块通过require函数使用当前模块时得到的就是当前模块的exports对象。

|  |
| --- |
| exports.hello = function () {  console.log('Hello World!');  }; |

* 1. module

module可以用来替换当前模块的导出对象，模块导出对象默认是一个普通对象，如果想改成一个函数的话，可以使用以下方式：

|  |
| --- |
| module.exports = function () {  console.log('Hello World!');  }; |

此时，调用就变成这样了：

|  |
| --- |
| var hello = require(‘./demo.js’);  hello(); //require返回的exports，变成了一个方法 |

### 文件操作

* 1. 小文件复制

|  |
| --- |
| 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)); |

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

* 1. 大文件复制

小文件复制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。

|  |
| --- |
| 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));ream(src).pipe(fs.createWriteStr |

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。

reader.pipe( writer , {end:false} )

默认情况下，当源可读流（the source Readable stream）触发 'end' 事件时，目标流也会调用 stream.end() 方法从而结束写入。要禁用这一默认行为， end 选项应该指定为 false， 这将使目标流保持打开；

* 1. 数据流

数据流是从源数据读取或写入数据到目标对象；

有四种流:

* Readable 数据流，其是用于读操作
* Writable - 数据流，用在写操作
* Duplex - 数据流，其可以用于读取和写入操作
* Transform - 双相类型流，输出基于输入进行计算

每种类型的流是EventEmitter,可以触发以下事件：

* data - 当有数据可读取时触发此事件
* end - 当没有更多的数据读取时触发此事件(读结束)
* error - 当有错误或接收数据写入时触发此事件
* finish - 当所有数据已刷新到底层系统时触发此事件（写结束）

例如：

从数据流读取：

|  |
| --- |
| var readerStream = fs.createReadStream('test.txt');  var data = ‘’;  readerStream.setEncoding('UTF8');  readerStream.on(‘data’,function(chunk){  data += chunk;  })  readerStream.on('end',function(){  console.log(data);  });  readerStream.on('error', function(err){  console.log(err.stack);  }); |

写入数据流：

|  |
| --- |
| var writerStream = fs.createWriteStream('test1.txt');  var data = ‘test’;  writerStream.write(data,'UTF8');  writerStream.end(); //结束写入流，触发finish事件  writerStream.on('finish', function() {  console.log("Write completed.");  }); |

* 1. BOM头移除

在不同的Unicode编码下，BOM字符对应的二进制字节如下：

Bytes Encoding

----------------------------

FE FF UTF16BE

FF FE UTF16LE

EF BB BF UTF8

|  |
| --- |
| function readText(pathname) {  var bin = fs.readFileSync(pathname);  if (bin[0] === 0xEF && bin[1] === 0xBB && bin[2] === 0xBF) {  bin = bin.slice(3);  }  return bin.toString('utf-8');  } |

* 1. node读取GBK文本文件

|  |
| --- |
| var iconv = require('iconv-lite');  function readGBKText(pathname) {  var bin = fs.readFileSync(pathname);  return iconv.decode(bin, 'gbk');  } |

* 1. 单字节编码

|  |
| --- |
| function replace(pathname) {  var str = fs.readFileSync(pathname, 'binary');  str = str.replace('foo', 'bar');  fs.writeFileSync(pathname, str, 'binary');  } |

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。

假如一个文件里有中文字符，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。

示例：

|  |
| --- |
| 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'; |

* 1. 同步遍历目录

|  |
| --- |
| var fs = require('fs');  var path = require('path');  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);  }  });  }  travel('/Users/yangwenju/Project/mynode',function(pathname){  console.log(pathname);  }); |

### 网络操作

* 1. http

http模块有两种操作：

* 作为服务端使用时，创建一个HTTP服务器，监听HTTP客户端请求并返回响应。
* 作为客户端使用时，发起一个HTTP客户端请求，获取服务端响应。

HTTP请求本质上是一个数据流，由请求头（headers）和请求体（body）组成：

|  |
| --- |
| POST / HTTP/1.1  User-Agent: curl/7.26.0  Host: localhost  Accept: \*/\*  Content-Length: 11  Content-Type: application/x-www-form-urlencoded  Hello World |

空行之上是请求头，之下是请求体。HTTP请求在发送给服务器时，可以认为是按照从头到尾的顺序一个字节一个字节地以数据流方式发送的。http模块创建的HTTP服务器在接收到完整的请求头后，就会调用回调函数。

例子：

|  |
| --- |
| http.createServer(function(request,response){  var postData = '';  request.on('data',function(chunk){  postData +=chunk;  })  request.on('end',function(){  response.writeHead(200,{"content-type":"text/html"});  response.end(postData.toString());  })  }).listen(8800); |

客户端：

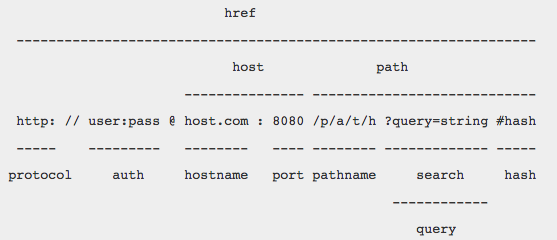
|  |
| --- |
| var options = {  hostname: 'www.example.com',  port: 80,  path: '/upload',  method: 'POST',  headers: {  'Content-Type': 'application/x-www-form-urlencoded'  }  };  var request = http.request(options, function (response) {});  request.write('Hello World');  request.end(); |

.request方法创建了一个客户端，并指定请求目标和请求头数据。之后，就可以把request对象当作一个只写数据流来写入请求体数据和结束请求。

对于GET请求，不需要请求体，有下面编辑Api：

http.get('http://www.example.com/', function (response) {});

* 1. url模块



url.parse():返回一个对象，有下面的属性：

|  |
| --- |
| url.parse('http://user:pass@host.com:8080/p/a/t/h?query=string#hash');  /\* =>  { protocol: 'http:',  auth: 'user:pass',  host: 'host.com:8080',  port: '8080',  hostname: 'host.com',  hash: '#hash',  search: '?query=string',  query: 'query=string',  pathname: '/p/a/t/h',  path: '/p/a/t/h?query=string',  href: 'http://user:pass@host.com:8080/p/a/t/h?query=string#hash' }  \*/ |

url.parse(tempUrl,true,true):

第二个参数：true，query字段不再是字符串，而是对象{ query: 'string'}

第三个参数：true, 可以正确解析不带协议头的url请求。

url.resolve(part1,part2): 拼接url

url.resolve('http://www.example.com/foo/bar', '../baz');

// http://www.example.com/baz

* 1. querystring

querystring模块用于实现URL参数字符串与参数对象的互相转换

|  |
| --- |
| querystring.parse('foo=bar&baz=qux&baz=quux&corge');  /\* =>  { foo: 'bar', baz: ['qux', 'quux'], corge: '' }  \*/  querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });  /\* =>  'foo=bar&baz=qux&baz=quux&corge='  \*/ |

### 进程管理

* 1. 获取命令行参数

process.argv 获取命令行参数，但node执行程序路径和主模块文件路径固定占据了argv[0]和argv[1]两个位置，所以第一个命令行参数从argv[2]开始。

所以，真正的参数是：process.argv.slice(2)

* 1. 退出程序

process.exit(1) //退出并把退出状态码设置为1；

### Buffer

* 1. 创建缓冲区

创建10个字节的缓冲区的语法：

var buf = new Buffer(10);

从给定数组创建一个缓冲区：

var buf = new Buffer([10, 20, 30, 40, 50]);

从给定的字符串和可选的编码类型创建缓冲区：

var buf = new Buffer("Simply Easy Learning", "utf-8");

* 1. 写到缓冲区

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

参数：

string:要写入缓冲区的字符串

offset：从缓冲区的此位置开始写入

length：要写入的字节数

encoding : 编码 ，utf8是默认编码

返回：写入的字节数；

* 1. 从缓冲区读取

buf.toString([encoding][, start][, end])

参数：

encoding：编码，默认utf8

start：开始读取的索引，默认0

end：结束读取的索引，默认为buf.length

* 1. 连接缓冲区

### 事件发射器

var events = require('events');

var eventEmitter = new events.EventEmitter();

当EventEmitter实例出现任何错误，会触发 error 事件；

当新的监听器添加，newListener事件被触发；

当一个监听器被删除，removeListener事件被触发

* 1. EventEmitter类

实例方法：

on(event,listener) :监听event事件，若有触发，执行listenter的回调

emit（event）：触发event事件

removeListener(event,listener) : 移除事件

类方法：

require('events').EventEmitter.listenerCount(emitter, event)

参数：类实例，事件

### moment.js

* 1. add

|  |
| --- |
| let startTime = moment();  let a = startTime.add(1,'days');  let b = startTime.add(2,'days');  let result = [a,b];  输出：  [  "2018-07-01T10:11:47.888Z",  "2018-07-01T10:11:47.888Z"  ]  //这里最后输出的a,b是一样的，add返回的还是同一个moment对象，  因为startTime会累加，会一直变化，所以最后输出的a,b其实都是同一个moment对象。  若要记录某个时间的时间点，可以用valueOf()获取当时的时间戳即可。  let startTime = moment();  let a = startTime.add(1,'days').valueOf();  let b = startTime.add(2,'days').valueOf();  let result = [a,b];  输出：  [  1530267013633,  1530439813633  ] |

注意：

moment().valueOf() 返回时间戳毫秒数

moment().toString() 返回: "Wed Jul 11 2018 17:43:26 GMT+0800"

moment().toISOString() 返回："2018-07-11T09:40:48.641Z"

### Lodash

* 1. \_.isEmpty()：

isEmpty: Number类型和Date类型都返回true，其他都正常

* 1. \_.isEqual(value, other)

执行深比较来确定两者的值是否相等。

var object = { 'a': 1 };

var other = { 'a': 1 };

\_.isEqual(object, other); //true

object === other //false

* 1. \_.filter（）

将数组的所有元素都遍历一遍，返回所有符合条件的元素的数组

|  |
| --- |
| var users = [  { 'user': 'barney', 'age': 36, 'active': true },  { 'user': 'fred', 'age': 40, 'active': false }  ];  //return true则返回  \_.filter(users, function(o) { return !o.active; });  //若匹配到对应的属性和值，则返回  \_.filter(users, { 'user': 'barney', 'active': true });  //第二个参数是数组，分别为属性名和属性值  \_.filter(users, ['active', false]);  //第二个参数是字符串，代表属性名，能筛选出含有该属性的，且属性值不为null,undefined,false等空值的  \_.filter(users, 'age'); |

### js常用功能

* 1. 多级排序

|  |
| --- |
| let test = [  {a:7,b:4,name:'yang'},  {a:2,b:3,name:'wen'},  {a:2,b:5,name:'wen'},  {a:5,b:9,name:'ju'},  ]  //先按a降序排序，若a相同时，按b升序排序  test.sort((o1,o2) => {  //一级排序  if(o1.a !== o2.a){  return o2.a - o1.a; 返回值大于0为降序，小于0为升序  }else{  //二级排序  return -1; 返回-1表示升序，1表示降序  //return o1.b - o2.b;  }  }) |

延伸：js按汉语拼音排序：o1.localeCompare(o2);

* 1. js中几种遍历方法的比较
     1. map 有序遍历

let new\_array = arr.map(function callback(currentValue, index, array) {

// Return element for new\_array

}[, thisArg])

callback只会在有值的索引上被调用，那些未赋值的或delete删除的索引 不会被遍历到。

注意：

遍历数组的范围在callback第一次调用时就已经确定了，在map方法执行过程中，原数组中新增加的元素不会被callback访问到；

若已存在的元素被修改，他们传递到callback中的值是map方法遍历到它时候的值；

未赋过值的索引或被删除的索引不会被访问。

应用：重新格式化数组中的对象

* + 1. filter

var new\_array = arr.filter(callback(element[, index[, array]])[, thisArg])

filter为每个元素调用一次callback，并将callback结果为true或等价于true的元素创建一个新数组。

filter同map类似，遍历范围在第一次执行callback时已经确定

未赋过值或被删除的索引不会被callback调用

* + 1. forEach 有序遍历

array.forEach(callback(currentValue, index, array){

//do something

}, this)

forEach中想要跳过某次循环，直接return即可

forEach没有返回值！！

未赋过值或被删除的索引不会被callback调用

* + 1. for-in 无序

for…in以任意序迭代一个对象的属性

它会遍历对象本身，以及继承的所有可枚举属性

|  |
| --- |
| 注意：通常，在迭代过程中最好不要在对象上进行添加、修改或者删除属性的操作，除非是对当前正在被访问的属性。这里并不保证是否一个被添加的属性在迭代过程中会被访问到，不保证一个修改后的属性（除非是正在被访问的）会在修改前或者修改后被访问，不保证一个被删除的属性将会在它被删除之前被访问。因为for…in是无序遍历的 |

所以，对于Array类型，不适合用for…in遍历，因为数组一般需要有序遍历。

* + 1. for-of 有序

遍历任何可迭代对象，见ES6学习.doc (按顺序)

顺序遍历，未赋过值或被delete的索引也会被访问到。

* + 1. for
    2. reduce

arr.reduce(callback[,initialValue])

参数：

callback(accumulator, currentValue, currentIndex, array)

有四个参数：

accumulator：上一次调用回调时返回的累积值（或初始值）

currentValue：数组正在处理的元素

currentIndex：正在处理元素的索引

array ：调用reduce的数组

initialValue： 用作首次调用 callback时的第一个参数的值，若不提供初 初始值，则使用数组的第一个元素。

返回：累计处理的结果

描述： 若没有提供initialValue，reduce会从索引1的地方开始执行callback， 跳过第一个索引；如果提供initialValue，从索引0开始。

若无初始值： 

若有初始值：

|  |
| --- |
| [0, 1, 2, 3, 4].reduce((accumulator, currentValue, currentIndex, array) => { return accumulator + currentValue; }, 10 ); |



应用：在遍历数组的过程中，把每个元素的内容统一处理到一个返回值中；

* + 1. some

arr.some(callback[, thisArg])

参数：

callback(valule,index,array)接受三个参数：当前值，当前索引，some被调用的数组

thisArg:执行callback时使用的this值；

描述：

为数组中每个元素执行一次callback，直到找到一个使callback返回一个“真值”（可以转换为布尔值true的值），若找到了一个这样的值，some会立即返回true。否则，返回false。

some只会为赋值的索引调用callback！

应用：常用于过滤

find()和some()比较：

find返回的是元素的值，some返回的是true或false。

* + 1. find 有序

arr.find(callback(value, index, arr)[, thisArg])

返回值：第一个使callback返回true的元素的值，否则返回undefined

注意：find会按索引从0到length-1的顺序调用callback，也会调用没有值的索引。

* + 1. findIndex

arr.findIndex(callback(value, index, arr)[, thisArg])

返回值：第一个使callback返回true的元素的索引，否则返回-1

* + 1. every

arr.every(callback[, thisArg])

测试数组的所有元素是否都通过了指定函数的测试

描述：every为每个元素调用一次callback，直到找到一个让callback返回false的元素，如果找到立即返回false。否则，返回true。

callback只会调用已经被赋值的索引，被删除或未赋值的会忽略；

some和every的区别：

some是只要有一个返回true就立即返回true

every是必须所有的都返回true才会返回true。

* 1. \_.filter（）

将数组的所有元素都遍历一遍，返回所有符合条件的元素的数组

|  |
| --- |
| var users = [  { 'user': 'barney', 'age': 36, 'active': true },  { 'user': 'fred', 'age': 40, 'active': false }  ];  //return true则返回  \_.filter(users, function(o) { return !o.active; });  //若匹配到对应的属性和值，则返回  \_.filter(users, { 'user': 'barney', 'active': true });  //第二个参数是数组，分别为属性名和属性值  \_.filter(users, ['active', false]);  //第二个参数是字符串，代表属性名，能筛选出含有该属性的，且属性值不为null,undefined,false等空值的  \_.filter(users, 'age'); |