### 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

exports和module.exports的区别：

<https://cnodejs.org/topic/54e67b70629934692ea99f91>

|  |
| --- |
| 相同之处  exports 和 module.exports 并不是全局变量，而只是对各自的 module 可见。  它们指向同一个对象，其缺省初始值为空 {}。  如果 exports 和 module.exports 没有被重新赋值，这个对象就是将要输出的对象。  不同之处  module 是对当前模块的一个引用。真正输出的是 module.exports。  所以对 exports 直接赋值没有作用。而对 module.exports 直接赋值后，module.exports 就指向新的对象了，这个新的对象成为将要被输出的对象。  详细说明  exports 和 module.exports 指向的是同一个对象，所以给他们任何一个添加属性或方法，另外一个都会接收到变化，因为他们指向的是同一个对象。例如：  exports.afunc = function(){};  module.exports.name = "Wang";  console.log(exports);  console.log(module.exports);  后面两条语句的输出都是： <code> { afunc: [Function], name: ‘Wang’ } </code> 。  但是如果是对其中任何一个直接赋值，就会切断对最初对象的引用。  exports  例如给 exports 直接赋值，就切断了 exports 对之前其和 module.exports 一同指向的对象的引用，但是由于真正输出的是 module.exports 指向的对象，所以对 exports 赋值无效，比如在 ep.js 输入：  exports.name = "Wang";  var afunc = function(){};  exports = afunc; //注意对 require 无效  console.log(exports); //内部有效，输出 [Function]  在另外一个 me.js 输入：  var ep = require('./ep');  console.log(ep);//输出 {name: 'Wang'}，不是 [Function]  要让 exports 输出，可以在 ep.js 加上  module.exports = exports;  两者的 reference 关系再次建立。这样输出都是 [Function] 。  module.exports  如果给 module.exports 直接赋值，也切断了 exports 的引用，同样道理，因为真正输出的是 module.exports 指向的对象，所以新赋给 module.exports 的对象将被输出。例如把 ep.js 改为：  exports.name = "Li";  console.log(exports); //{name: 'Li'}  console.log(module.exports);//一样 {name: 'Li'}  module.exports = "Zhou";  console.log(exports); //还是 {name: 'Li'}  console.log(module.exports);//变为 Zhou  再次运行 me.js，输出 <code> Zhou </code>, 覆盖了之前的对象.  总结  真正输出总是 module.exports。如果两者同时出现或被修改，只有 module.exports 返回，exports 被忽略。 |

* 1. require.main === module

当 Node.js 直接运行一个文件时，require.main 会被设为该文件模块的 module变量。

|  |
| --- |
| if(require.main === module){  // 应用程序直接运行;启动应用服务器  startServer();  } else {  // 应用程序作为一个模块通过 "require" 引入 : 导出函数 // 创建服务器  module.exports = startServer;  } |

### 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);  }); |

* 1. fs.writeSync(2, ` \x1b[91m failed with unique restrict\x1b[39m `)

将内容输出到标准错误上，即终端上显示。

首个参数是fd: 描述控制符

\x1b[91m: 终端控制符，表示颜色

### 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：

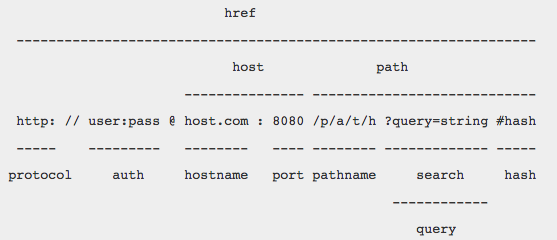
|  |
| --- |
| var options = {  hostname: 'www.example.com',  port: 80,  path: '/upload',  method: 'POST',  headers: {  'Content-Type': 'application/x-www-form-urlencoded'  }  };  var request = http.request(options, function (response) {});  request.write('Hello World');  request.end(); |

.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；

* 1. process.hrtime([time])

返回的时间，都是相对于过去某一时刻(任意时间)的值，与一天中的时钟时间没有关系，因此不受制于时钟偏差.

主要用途是用于测量时间间隔之间的性能

time是可选参数，传入的值是上一次调用precess.hrtime返回的结果，用于和当次调用做差值计算。

### Buffer

* 1. 创建缓冲区

**当存储的数据已确定时：使用Buffer.from:**

Buffer.from(array)

array必须是integer的数组；

Buffer.from(buffer)

拷贝 buffer 的数据到新建的 Buffer。

Buffer.from(string [,encoding])

string:要转化的字符串

注意：Buffer.from不支持传入**数字。**

**当存储数据不确定时：**

Buffer.alloc(size [, fill [, encoding]])

size: 新buffer的长度，数值类型；

fill: 预填充的值，默认为0

encoding：如果 fill 是一个字符串，则这是它的字符编码。默认值: 'utf8'

* 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. buf.toJSON() buffer转json
  2. buf.includes()

buffer是否包含特定值

|  |
| --- |
| const buf = Buffer.from('this is a buffer');  console.log(buf.includes('this')); // true  console.log(buf.indexOf('this')); // 0 |

* 1. buf[index]

索引操作符 [index] 可用于获取或设置 buf 中指定位置的字节.

该值指向单个字节，所以有效的值的范围是 0x00 至 0xFF（**十六进制**），或 0 至 255（**十进制**）。

* 1. buf.fill(value [,offset [,end [,encoding]]])

用value填充buffer, 如果没有指定 offset 与 end，则填充整个 buf.

value是string || integer

清空buffer最快的方法：buf.fill(0)

### 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'); |

* 1. \_.pick \_.omit

\_.pick: 创建一个从 object 中选中的属性的对象

|  |
| --- |
| var object = { 'a': 1, 'b': '2', 'c': 3 };    \_.pick(object, ['a', 'c']);  // => { 'a': 1, 'c': 3 } |

\_.omit:从object中剔除某属性，\_.pick的反向

|  |
| --- |
| var object = { 'a': 1, 'b': '2', 'c': 3 };    \_.omit(object, ['a', 'c']);  // => { 'b': '2' } |

* 1. \_.sample 返回随机数元素（随机抽取）

|  |
| --- |
| \_.sample([1, 2, 3, 4]);  // => 2 |

* 1. \_.random(lower,upper,floating)

lower:下限

upper:上限

floating:是否返回浮点数

* 1. \_.isNaN的坑

\_.isNaN仅仅在\_.isNaN(NaN)时是true,其他情况都是false；

例如：

\_.isNaN('hfdsifuahi') 是false

这一点跟js的Number.isNaN()方法是一样的；

js全局的isNaN()则不同，它先将value转化一下，若能转化成number类型，则返回false,若不能，则返回true;

所以，isNaN('dsffhius') 返回的是true

isNaN("123") 返回的是false

* 1. \_.groupBy

根据两个字段分组，将两个字段拼起来作为key

\_.groupBy(arr, function(o){return o.name + ',' +o.age});

* 1. 创建链对象

\_(value)建立了一个隐式链对象

\_.chain(value)建立了一个显式链对象

也可通过\_(value).chain()从隐式转成显式。

显式链调用的话，需要通过commit() 手动结束链式反应，或者 value() 手动结束链式反应并提取值。

隐式链调用的话，碰到能返回唯一值 (single value) 或原生数据类型（primitive value），才会自动结束链式反应并自动提取值。否则需要你像上面一样手动操作。

例如 sum 可以触发隐式链调用的自动结束，但是 filter 不行。

什么时候用显式/隐式？

* 显式对 commit 和 value 更可控，灵活度更高。
* 隐式可以简洁代码。

|  |
| --- |
| // 隐式链 - 需手动结束  \_([1, 2, 3])  .reverse()  .value();  var users = [  { 'user': 'barney', 'age': 36 },  { 'user': 'fred', 'age': 40 }  ];  // 启用显式链  \_.chain(users)  .head()  .pick("user")  .value();  // => { 'user': 'barney' } |

处理中间结果：

tap 直接修改值

thru 需返回值

|  |
| --- |
| 1、tap - 直接修改值  \_([1, 2, 3])  .tap(function(array) {  // 改变传入的数组  array.pop();  })  .reverse()  .value();  // => [ 2, 1 ]  2、thru - 需返回值  \_(' abc ')  .chain()  .trim()  .thru(function(value) {  return [value];  })  .value();  // => ['abc'] |

### js常用功能

* 1. 多级排序

arr.sort((a,b) =>return a-b)

结论：返回值>0,交换a,b;否则，保持顺序

|  |
| --- |
| 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;  }else{  //二级排序  return o1.b - o2.b;  }  }) |

|  |
| --- |
| arr.sort(function(a,b){  return a-b;//升序  return b-a;//降序  })  若a-b <0 , 返回一个小于零的值，升序  若a-b>0,返回一个大于零的值，降序 |

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'); |

### NPM

#### 安装包

npm install 包名 --save-dev 参数能将该包名保存在package.json的dev依赖里

#### npm scripts

打开package.json,修改scripts字段：

"scripts": {

"start": "node index.js",

}

之后可以使用 npm start 来运行index.js 文件

一些不在npm scripts支持范围内的名字，要用npm run XX.js

npm 脚本有pre和post两个钩子，举例来说，build脚本命令的钩子就是prebuild和postbuild。

用户执行npm run build的时候，会自动按照下面的顺序执行：

npm run prebuild && npm run build && npm run postbuild

|  |
| --- |
| {  "scripts": {  "prebuild ": "echo \"--Pre \"",  "build": "echo \"Hello World\"",  "postbuild": "echo \"--post\""  }  } |

NPM 中所有依赖的 node\_modules bin 都可以在脚本中直接访问，就像在路径中被引用的一样。

|  |
| --- |
| {  "scripts": {  "lint": "./node\_modules/.bin/eslint .",  }  }  // 此写法与上面效果相同  {  "scripts": {  "lint": "eslint ."  }  } |

依次执行多个脚本，用&& npm run lint && npm test

并行执行多个脚本，用& npm run lint＆npm test

访问环境变量：

npm\_config\_<val> 或者 npm\_package\_<val>

|  |
| --- |
| {  "scripts": {  "config:loglevel": "echo \"Loglevel: $npm\_config\_loglevel\"",  }  } |

可以使用的环境变量可以通过npm config ls -l获取

传递参数：

npm run <script> --<argument>="value"

|  |
| --- |
| {  "scripts": {  "ttt": "echo \"ttt $npm\_config\_firstname\""  }  }  npm run ttt --firstname=234  ttt 234 |

npm提供生命周期脚本，比如prepare

会在执行npm install 且不带任何参数前执行

|  |
| --- |
| {  "scripts": {  "prepare": "husky install""  }  } |

这样可以在npm install时提前完成husky的安装

#### npm依赖包的写法

* + 1. 语义化版本号 semver

semver 约定一个包的版本号必须包含3个数字，格式必须为 MAJOR.MINOR.PATCH, 意为 主版本号.小版本号.修订版本号

|  |
| --- |
| "hexo": "^3.3.7" |

* + - * version Must match version exactly
      * >version Must be greater than version
      * >=version etc
      * <version
      * <=version
      * ~version “Approximately equivalent to version” See semver
      * ^version 指定的 MAJOR 版本号下, 所有更新的版本
      * 1.2.x 1.2.0, 1.2.1, etc., but not 1.3.0
      * \* Matches any version
      * "" (just an empty string) Same as \*
      * version1 - version2 Same as >=version1 <=version2.
      * range1 || range2 Passes if either range1 or range2 are satisfied



任意两条规则 ，用空格连接，表示“且”：

如 >=2.3.1 <=2.8.0 可以解读为: >=2.3.1 且 <=2.8.0

* 可以匹配 2.3.1, 2.4.5, 2.8.0
* 但不匹配 1.0.0, 2.3.0, 2.8.1, 3.0.0

任意两条规则，通过 || 连接起来，表示“或”逻辑，即两条规则的并集:

如 ^2 >=2.3.1 || ^3 >3.2

* 可以匹配 2.3.1, 2,8.1, 3.3.1
* 但不匹配 1.0.0, 2.2.0, 3.1.0, 4.0.0

还有下面几种写法：

* 或 x 匹配所有主版本
* 1 或 1.x 匹配 主版本号为 1 的所有版本
* 或 1.2.x 匹配 版本号为 1.2 开头的所有版本
  + 1. git URL

Git urls为以下形式：

|  |
| --- |
| <protocol>://[<user>[:<password>]@]<hostname>[:<port>][:][/]<path>[#<commit-ish> | #semver:<semver>] |

protocol有几种形式：git git+ssh git+http git+https git+file

在path后可以使用 # 指定特定的 git branch/commit/tag

举例：

|  |
| --- |
| git+ssh://git@github.com:npm/npm.git#v1.0.27  git+ssh://git@github.com:npm/npm#semver:^5.0  git+https://isaacs@github.com/npm/npm.git  git://github.com/npm/npm.git#v1.0.27 |

* + 1. gitHub URL

直接指定github目录

|  |
| --- |
| "express": "expressjs/express" |

* + 1. Local Path

指定本地地址，只要指定的目录是有个package

|  |
| --- |
| "bar": "file:../foo/bar" |

* + 1. 区分不同registry

在用户根目录下建立.npmrc文件：

|  |
| --- |
| registry=https://registry.npm.taobao.org/  @xxx:registry=http://registry.npm.xxx.com |

在package.json中以"@xxx/xxx": ""方式书写的package都会使用不同的仓库。

### NPM包

#### node-schedule

|  |
| --- |
| const schedule = require('node-schedule');  const scheduleCronstyle = ()=>{  //每分钟的第30秒定时执行一次:  schedule.scheduleJob('30 \* \* \* \* \*',()=>{  console.log('scheduleCronstyle:' + new Date());  });  }  scheduleCronstyle() |

schedule.scheduleJob的回调函数中写入要执行的任务代码；

6个占位符从左到右分别代表：秒、分、时、日、月、周几；规则类似crontab

取消定时器：

调用 定时器对象的cancl()方法即可:

|  |
| --- |
| const j = schedule.scheduleJob('\* \* \* \* \* \*', function(){  console.log('定时器触发次数：' + counter);  });  j.cancel(); |

#### axios

axios retry的一种实现：

使用axios拦截器：

|  |
| --- |
| const axios = require('axios');  axios.interceptors.response.use(undefined, function axiosRetryInterceptor(err) {  var config = err.config;  // If config does not exist or the retry option is not set, reject  if(!config || !config.headers.retry) return Promise.reject(err);  // Check if we've maxed out the total number of retries  if(config.headers.retry === 0) {  // Reject with the error  return Promise.reject(err);  }    // decrease the retry count  config.headers.retry--;  return axios(config)  });  //把retry放到header里，因为新版本的axios不支持自定义的config  axios.get('www.google.com', {headers: {  retry: 3  }}).then(res => {  console.log(res);  }).catch(err => {  console.log(err);  }) |

#### git-diff

输出两段文字不同的地方，并用颜色区分

const diff = gitDiff(oldStr, newStr, {color: true})

console.log(diff)

#### commander

<https://github.com/tj/commander.js/blob/HEAD/Readme_zh-CN.md>

#### strip-json-comments

去掉注释，返回无注释的json

#### nodemailer

https://nodemailer.com/usage/

|  |
| --- |
| const transporter = nodemailer.createTransport({  host: "smtp.example.com",  port: 587,  secure: false,  auth: {  user: "username",  pass: "password"  }  });  const data = {  from: "sender@server.com",  to: "receiver@sender.com",  subject: "Message title",  text: "Plaintext version of the message",  html: "<p>HTML version of the message</p>"  };  transporter.sendMail(data[, callback]) |

### Mocha

* 1. skip

存在skip时，表示跳过该测试，其他普通的describe或it会执行。

* 1. only

存在only时，其他普通的it或describe就不执行了，只执行only

### node时间轮询

process.nextTick() setImmediate区别

process.nextTick方法可以在当前"执行栈"的尾部----下一次Event Loop（主线程读取"任务队列"）之前----触发回调函数。也就是说，它指定的任务总是发生在所有异步任务之前。

setImmediate方法则是在当前"任务队列"的尾部添加事件，也就是说，它指定的任务总是在下一次Event Loop时执行，这与setTimeout(fn, 0)很像。

当前执行栈-->process.nextTick--> 任务队列（本次loop）--> setImmediate(下次loop)

|  |
| --- |
| process.nextTick(function A() {  console.log(1);  process.nextTick(function B(){console.log(2);});  });  setTimeout(function timeout() {  console.log('TIMEOUT FIRED');  }, 0)  // 1  // 2  // TIMEOUT FIRED |

由于process.nextTick指定的回调函数，总是在当前“执行栈”的尾部触发，所以不仅函数A比setTimeout指定的回调函数timeout先执行，而且函数B也比timeout先执行。说明如果多个process.nextTick语句（无论是否嵌套），将全部在当前“执行栈”执行。

多个process.nextTick语句总是在当前"执行栈"一次执行完，多个setImmediate可能则需要多次loop才能执行完。因为setImmediate总是将时间注册到下一轮event loop。

### crypto

#### crypto.createCipheriv(algorithm, key, iv[, options])

创建并返回Cipher对象。

#### Cipher 类

* + 1. cipher.setAutoPadding([autoPadding])

autoPadding：默认true

当使用块加密算法时， Cipher类会自动添加padding到输入数据中，来适配相应块大小。

当autoPadding是false时，整个输入数据的长度必须是cipher块大小的倍数，否则cipher.final()将抛出一个错误。

cipher.setAutoPadding()必须在cipher.final()之前被调用。

* + 1. cipher.update(data[, inputEncoding][, outputencoding])

inputEncoding: 参数data的字符编码，必须是'utf8' 'ascii' 或'latin1'三种；

data:需要加密的字符串，必须与inputEncoding对应；

outputencoding: 加密数据的输出格式 ,可以是'latin1' 'base64' 'hex'三种；

cipher.update可以多次调用，知道cipher.final()被调用；

* + 1. cipher.final([outputEncoding])

outputEncoding： 'latin1', 'base64' 或者 'hex'；

返回加密内容，若outputEncoding是以上三种类型，返回字符串，否则返回Buffer

* + 1. 使用过程

|  |
| --- |
| const crypto = require('crypto');  const cipher = crypto.createCipheriv('aes-256-cbc','BYkCpvTfyZ%hrJYSrOUWxPhrJAOZcVZo', 'cxZhhYhet2X4OOMq');  let encrypted = cipher.update('some clear text data', 'utf8', 'hex');  encrypted += cipher.final('hex');  console.log(encrypted); |

#### Decipher类

* + 1. crypto.createDecipheriv(algorithm, key, iv[, options])

创建并返回一个Decipher对象；

* + 1. decipher.setAutoPadding([autoPadding])

同cipher；

* + 1. decipher.update(data[, inputEncoding][, outputencoding])

inputEncoding：'latin1', 'base64' 或者 'hex'；

outputEncoding：'latin1', 'ascii'或'utf8'

* + 1. decipher.final([outputEncoding])

返回剩余的解密内容

* + 1. 使用过程

|  |
| --- |
| const crypto = require('crypto');  const decipher = crypto.createDecipheriv('aes-256-cbc', 'BYkCpvTfyZ%hrJYSrOUWxPhrJAOZcVZo', 'cxZhhYhet2X4OOMq');  const encrypted =  'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504';  let decrypted = decipher.update(encrypted, 'base64', 'utf8');  decrypted += decipher.final('utf8');  console.log(decrypted); |

### NODE\_ENV

NODE\_ENV=production app.js

即设置了process.env.NODE\_ENV = production

在express中使用了config模块，config.get()即读取的是production.json里的配置。

使用cross-env解决跨平台设置NODE\_ENV的问题：

问题：windows不支持NODE\_ENV=development的设置方式

使用：

1. 安装cross-env:npm install cross-env --save-dev
2. 在NODE\_ENV=xxxxxxx前面添加cross-env就可以了。

### child\_process

如何让子进程也能够debug:

|  |
| --- |
| // main.js 主进程  const fork = require('child\_process').fork;  // 子进程监听一个新的端口，否则会跟main.js的debug端口号冲突！！  const child = fork('./fork.js',{'execArgv': [`--inspect-brk=9001`]});  child.send('hello son');  // fork.js 子进程  process.on('message', async (data) => {  console.log(data + ' Thanks');  }); |

launch.json

|  |
| --- |
| "configurations": [  {  "type": "node",  "request": "launch",  "name": "main",  "program": "${workspaceFolder}/main.js",  "**autoAttachChildProcesses**": true,  } |

child\_process.fork(modulePath[, args]) 返回一个子进程child

child.send(data)可以向子进程发送消息，父进程通过

process.on('message', (data) => {

console.log(data);

})

来接受消息；

### events事件

EventEmitter 类由 events 模块定义和开放的，所有能触发事件的对象都是 EventEmitter 类的实例

events模块的EventEmitter属性指向该模块本身

|  |
| --- |
| var events = require('events');  console.log(events.EventEmitter === events);//true |

EventEmitter是一个构造函数，可以用来生成事件发生器的实例emitter

|  |
| --- |
| var EventEmitter = require('events');  var emitter = new EventEmitter(); |

#### emitter.on(eventName, listener)

添加listener函数到名为eventName的事件的监听器数组的末尾；

不检查listener是否已被添加，多次on会导致被触发多次；

可链式调用；

emitter.on('test',listener).on(‘test2’,listener2)

别名：emitter.**addListener**(eventName, listener)

#### emitter.emit(eventName[, ...args])

按监听器的注册顺序，同步地调用每个注册到名为eventName事件的监听器，并传入提供的参数；如果事件有监听器返回true，否则返回false；

#### emitter.prependListener()

将事件监听器添加到监听器数组的**开头**

#### emitter.once(eventName, listener)

添加一个单次 listener 函数到名为 eventName 的事件。 触发 eventName 事件时，监听器会被移除

emitter.**prependOnceListener**(eventName, listener)

添加到监听器数组的开头，且只能触发一次

#### emitter.removeListener(eventName, listener)

别名：emitter.**off**(eventName, listener)

从名为 eventName 的事件的监听器数组中移除指定的 listener

listener必须提前定义好的变量, 直接写function不行。

#### emitter.eventNames()

返回一个列出触发器已注册监听器的事件的数组。

数组元素为字符串或符号

#### emitter.listenerCount(eventName)

返回正在监听名为 eventName 的事件的监听器的数量

#### emitter.listeners(eventName)

返回名为 eventName 的事件的监听器数组的副本

|  |
| --- |
| var EventEmitter = require('events');  var emitter = new EventEmitter();  emitter.addListener('test',function(){console.log(1);});  emitter.addListener('test',function(){console.log(2);});  console.log(emitter.listeners('test'));//[ **[Function], [Function] ]**  emitter.listeners('test')[0]();//1 |

#### emitter.getMaxListeners()

返回 EventEmitter 当前的最大监听器限制值

#### emitter.setMaxListeners(n)

设置监听器的上限

#### 'newListener' 事件

其后有新的事件被监听时触发

|  |
| --- |
| emitter.on('test',function(){  console.log(1);  })  emitter.on('newListener',function(){  console.log(2);  })  // 监听'newListener'事件之后，没有其他监听事件了，所以没有触发  emitter.emit('test');// 1 |

|  |
| --- |
| emitter.on('test',function(){  console.log(1);  })  emitter.on('newListener',function(){  console.log(2);  })  //监听'newListener'事件之后, 又监听了test3事件，在监听test3事件时，触发了'newListener'监听器，输出了2  emitter.on('test3',function(){  console.log(3);  })  emitter.emit('test');// 2 3 |

#### removeListener 事件

其后有 listener 被移除时触发

|  |
| --- |
| function show(){  console.log(1);  }  emitter.on('test',show).removeListener('test',show);  // 监听'removeListener' 在removeListener动作之后，故不会触发  emitter.on('removeListener',function(){  console.log(2); //''  }) |

|  |
| --- |
| const show = function(){  console.log(1);  };  emitter.on('test',show);  emitter.on('removeListener', function(){  console.log('removing');  });  emitter.removeListener('test',show); |

### http.ServerResponse对象事件

监听response两个事件可以对接口请求状况做监控

#### close事件

在http.ServerResponse对象的end方法被调用之前,如果连接被中断,将触发http.ServerResponse对象的close事件

res.on(‘close’, handler)

#### finish事件

当response的header和body被发送完之后会触发，此时客户端并不一定收到信息

res.on(‘finish’, handler)