目　录

致谢

当前文档 《解读 node.js api 文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-24。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/node-api-docs

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

前言

> 目前初翻阶段。会在全部翻译完做一遍校对。
> 如有看到错误的同学，请多多帮手提出来。或者等1周左右后再看，理论上应该会翻译完。如果我太懒没更新，可以在 *twitter* 、新浪微博 催更。

前言

本篇有些部分直译自 node 的 api 文档，版本为 0.8.8。与其说是翻译，不如说是个人的一篇笔记。为保证全面，没有做接口的增删，并从中加入自己的注解。希望以学习+传达的方式达到更好的记忆，并把知识传播出去。对这些文字，你可以用于任何非商业的传播而无须征求我的意见；拒绝一切商业传播，包括我自己，希望这篇文档永远都是免费的。

来源(书栈小编注)

> *https://github.com/sofish/node-api-docs*

一、全局对象

一、全局对象

1. global {Object} 全局命名空间

对于浏览器来说，最上层的作用域是全局作用域，这意味着在浏览器的全局作用域中 `var something` 将会定义一个全局变量，但 Node 并不是这样，其最高作用域并非全局，在一个模块中 `var something` 将只是定义了模块的一个本地变量。

2. console {Object}

用以打印于 stdout 和 stderr 的对象。最有用的用是 `console.log()` 来打印内容。支持运行表达式；支持多个参数，用逗号分隔。其次是 `console.dir()` 用以打印对象下属性/方法。

3. require {Function} 引入模块

在 node 中，模块使用 `require('module_name')` 来引用。

4. require.resolve() {Function} 查看模块路径

使用 `require('module_name')` 的方式会加载一个模块，而 `require.resolve('module_name')` 是查

找 `module_name` 的路径，返回一个绝对路径：

```
require.resolve('./src/class')
// /Library/WebServer/Documents/github/sofish/src/class.js
```

5. require.cache {Object} 模块缓存

每个模块被 `require` 一次之后，都会缓存起来，信息存于 `require.cahce` 这个对象中。删除 key 的值可重新加载一个模块，也即进行多一次 `require`，而非从缓存中拿。

6. require.extensions {Array}

告诉 `require` 如何处理一个特定文件扩展名。譬如把 `.sjs` 扩展名当作 `.js` 文件来渲染可以这样做：

```
require.extensions['.sjs'] = require.extensions['.js'];
```

写一个扩展名处理器：

```
require.extensions['.sjs'] = function(module, filename) {
  var content = fs.readFileSync(filename, 'utf8');
  // ...
  // 处理文件内容，并通过 module.exports 导出
  module.exports = content;
}
```

7. __filename {String} 文件名

被执行文件的文件名。在命令行中执行则会返回错误。严格来说它并不是一个全局的对象，而是每个模块都拥有的本地对象。

值得注意的是：访问一个模块的时候，如果模块是文件夹，则返回模块下 `index.js`；而如果模块是单独文件，则返回其本身。返回的值是绝对路径。

8. __dirname {String} 路径名

与 `__dirname` 同理，返回的是一个绝对路径。

9. module {Object} 模块

当前模块的一个引用。比较特殊的是，`module.exports` 等价于 `exports` 对象。更确切来说

`module` 也是一个只存在于每个模块中的本地对象，而非严格意义上的全局对象。

10. exports {Object}

它可共用于当前模块的所有实例，使得模块可以通过 `require()` 来加载。如 #9 所述，它们是等价的。

11. setTimeout(callback, milliseconds) {Function}

延迟至少 milliseconds 执行一个 callback。实际的延迟时间由外部因素，如操作系统的计时间隔和系统负载决定。1秒 < milliseconds < 2,147,483,648秒，如果超出这个范围，则会以 1 秒来处理，也就是说一个 Timer 最多能跨越 24.8 天。

返回一个非对外的对象：

```
{ _idleTimeout: 123,
  _idlePrev:
   { _idleNext: [Circular],
     _idlePrev: [Circular],
     ontimeout: [Function] },
  _idleNext:
   { _idleNext: [Circular],
     _idlePrev: [Circular],
     ontimeout: [Function] },
  _onTimeout: [Function],
  _idleStart: Mon Sep 03 2012 22:26:55 GMT+0800 (CST) }
```

12. clearTimeout() {Function}

停止执行一个彼时操作，`setTimeout(callback, milliseconds)` 的 callback 将不会执行。

13. setInterval(callback, milliseconds) {Function}

在 milliseconds 这个间隔内循环执行 callback。实际的时间同 `setTimeout`，由外部因素决定。间隔的范围也是：1秒 < milliseconds < 2,147,483,648秒，如果超出这个范围，则会以 1 秒来处理。

14. clearInterval() {Function}

停止一个循环的执行。

15. process {Object}

全局对象还包含进行一个进程对象，在后面会详细讲解。

二、console {Object}

二、console {Object}

如大多数浏览器中的 console 对象一样，其输出会发送到 stdout 或 stderr 中。

1. console.log([data], […])

可以接受多个参数，如有字符替换，则默认后续以逗号分隔的参数会替换成相应的字符，如：

```
console.log('打印出数字: %d', '1234', '\n没有替换', '\n%s: %s', '第一个\%s', '第二个\%s');
// 打印出数字: 1234
// 没有替换
// %s: %s 第一个%s 第二个%s
```

默认支持 3 种类型的字符替换操作，%s, %d, %j。分别代表字符串、数字和JSON。间个 % 并不会替换。

与 `console.log()` 还有几个兄弟：

- console.info
- console.error 但输出到 stderr
- console.warn 同上
- console.dir 在全局对象中已经介绍过，输出对象下的属性/方法
- console.trace(label) 打印一个名为 label 的堆栈信息到 stderr

2. console.time(label) / console.timeEnd(lable)

计算程序运行时间的利器，计算 time 和 timeEnd 之间代码的运算时间，以 label 名来确定一个区间的内容，假设我们创建一个 label 名为 *sofish* 的计时器，如：

```
// 这有点类似于 html 的 <div id="sofish"> ... </div>
// 会计算名字为 _sofish_ 的这个区间内 for 循环的运行时间
 console.time('sofish');
 for(var i = 0; i < 1000; i++){
```

```
    console.log('计算这 for 循环的运算时间');
};
console.timeEnd('sofish');
// 在我的机器返回: sofish: 26ms
```

3. console.assert(expression, [message])

与 `assert.ok()` 一样，当 `expression` 这个传入的表达式返回 `false` 的话，抛出一个 AssertionError，`message` 为自定义信息。

```
console.assert(1===0, '出错了')
// AssertionError: 出错了
//    at Object.exports.assert (console.js:75:23)
//    ...

// 如果不指定 message，则返回表达式计算结果
console.assert(1==0)
// AssertionError: false == true
//    at Object.exports.assert (console.js:75:23)
//    ...
```

可能和你常用的测试用例的写法一样。当然，你完全可以用它来做测试用例，或者代替 `throw new Error('hello')` 。

三、计时器

三、计时器

1. setTimeout(callback, delay, [arg], […])

如上面提到的，`callback` 将会在 `delay` 后执行，执行结果返回一个 `timeoutId` 以便可以使用 `clearTimeout(timeoutId)` 来阻止执行这个计时器。而从第三个参数起，传入参数将作为 `callback` 的参数。

2. setInterval(callback, delay, [arg], […])

参数同 `setTimeout`，功能上 `setTimeout` 只执行一次，而 `setInterval` 则会循环执行。返回一个 `intervalId`，可以使用 `clearInterval(intervalId)` 来停下循环。

四、模块

四、模块

Node 自带一个简单的模块加载系统，文件即模板，它可以通过 `require()` 来加载，比如在 foo.js 中加载一个同目录下的 circle.js，我们可以这样做：

foo.js 中的内容：

```
1. var circle = require('./circle.js');
2. console.log( 'The area of a circle of radius 4 is ' + circle.area(4));
```

circle.js 中的内容：

```
1. var PI = Math.PI;
2.
3. exports.area = function (r) {
4.   return PI * r * r;
5. };
6.
```

```
exports.circumference = function (r) {
  return 2 * PI * r;
};
```

由于在 Node 中模块中定义的变量仅为模块私有而非全局变量，因此当需要在引用了当前模块的模块内使用，需要用 `exports` 来导出。如上述 circle.js 中导出了 `area()` 和 `circumference()` 两个方法，将可以在 foo.js 中使用。

1. 循环引用

当使用 `require()` 循环引用时，一个模块返回的时候可能仍未被执行。考虑一下正面的情况：

a.js:

```
console.log('a starting');
exports.done = false;
var b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

b.js:

```
console.log('b starting');
exports.done = false;
var a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

main.js

```
console.log('main starting');
var a = require('./a.js');
var b = require('./b.js');
console.log('in main, a.done=%j, b.done=%j', a.done, b.done);
`
```

当 main.js 加载 a.js，a.js 尝试加载 b.js。在这个时间，b.js 也尝试加载 a.js。为了防止死循环，一个不完全的 a.js 模块副本将会作为在 b.js 这个模块中所调用 a.js 的返回值，b.js 由于得到返回而完成这个加载的过程，而它在 a.js 中被调用，而将本身 `exports` 返回给 a.js。

这时，main.js 也完成了对两个模块的加载，它最终将会返回正面的数据：

```
$ node main.js
main starting
a starting
b starting
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done=true, b.done=true
```

也就是说，你可能需要注意一下不要循环调用以获得更高的性能，一旦你不小心这样搞了，Node 也会帮你阻止这个死循环。

2. 内置模块

Node 自身编译了一批内置模块。这些模块也将在这个文档中被详细介绍。它们定义于 Node 源文件的 `lib/` 目录下。

值得注意的是，内置模块总是优先被加载，就拿 `require('http')` 来说，将会返回 HTTP 模块，即使当前目录存在一个 `http.js` 或者 `http/index.js`。

3. 归存模块

当一个特定的文件名找不到的时候，Node 将会尝试按顺序加载带有 `.js` 、 `.json` 或 `.node` 扩展名的文件。 `.js` 、 `.json` 分别被解析为 JavaScript 和 JSON 文件，而 `.node` 则被解析为由 `dlopen` 来加载的编译插件。

模块路径以 `/` 前缀开始，表示加载以绝对路径出现的模块，比如， `require('/home/marco/foo.js')` 将会加载这个文件 `/home/marco/foo.js`；而以 `./` 则表示所加载的模块路径为相对路径，相对于引用 `require()` 的当前模块；当模块路径不以 `/` 或者 `./` 前缀出现，这个模块非内置模块，即加载自 `node_modules/` 目录。

当给定的模块路径不存在，则 `require()` 会抛出一个包含有属性名 `code` ，值为 `MODULE_NOT_FOUND` 的错误。

4. 从 node_modules/ 目录加载

当传到 `require()` 的模块标识符并不是内置模块，并且不以 `../` 、 `./` 、 `/` 前缀开始，Node 将会从该模块父目录下的 `node_modules/` 目录开始寻找这个模块，若模块未找到，则不断向上查找直到根目录。

打个比方，如果 `/home/ry/projects/foo.js` 调用了 `require('bar.js')` ，那么 Node 可能会从正面这些位置按顺序查找：

- /home/ry/projects/node_modules/bar.js
- /home/ry/node_modules/bar.js
- /home/node_modules/bar.js
- /node_modules/bar.js

这对于为个别项目做依赖的特殊化处理有着非常好的优势，且避免了冲突。

5. 将目录当做模块

将程序或库所有内容放置于同一个目录下，并导出一个统一的对外接口，这将会方便其自身的管理。将文件传入到 `require()` 中作为一个模块，有正面两种方式：

1. 在当前模块根目录放置一个 package.json 文件，在其中的 `main` 属性中指定一个模块，文件内容大致如下：

```
// 如果这个文件位于 ./some-library，那么 require('./some-library')
// 将会加载 ./some-library/lib/some-library.js
{
"name": "some-library",
"main": "./lib/some-library.js"
}
```

2. 当模块根目录下没有 package.json 文件，这时 Node 会尝试加载这个目录下的 `index.js` 或 `index.node` 文件。如 `require('./some-libray')` 将会尝试加载：

    - ./some-library/index.js
    - ./some-library/index.node

6. 缓存

模块在第一次 `require()` 的时候将会被缓存起来，也即当多次调用 `require('foo.js')` 的时候，模块将不会被再一次执行，这将从很大程度上提升了 Node 模块加载的性能。

如果你想多次执行代码，那么你可以将模块导出于一个函数内，然后调用这个模块；当然，还有另一种方式，就是改变 `require.cache` 对象中相应属性的值。

```
// mod.js
var Fish = function(name){
  this.name = name;
  ...
}

module.exports = function(name){
  return new Fish(name);
```

```
}
```

7. Module 缓存贴示

由于模块的查找总是基于文件，有些模块可能由于请求该模块文件位置的关系（从 `node_modules` 目录加载），将不能保证每次 `require('foo')` 放置在不同的位置请求到的都是同一个文件。这是需要注意的。

8. module 对象

在每一个模块中，module 变量总是为当前模块本身的一个引用。再一次提醒，像上面提到的 `module.exports` 与 `exports` 是等价的。

8.1 module.exports {Object}

这在上面我们已经谈过，有两段代码实例可以看一下：

```
// a.js
var EventEmitter = require('events').EventEmitter;

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(function() {
  module.exports.emit('ready');
}, 1000);
```

引用了 a.js 的文件：

```
var a = require('./a');
a.on('ready',
function() {
  console.log('module a is ready');
});
```

这里值得注意的是，`module.exports` 的调用必须是即时的，在回调中执行导出将不会有任何效果，考虑一下正面的代码：

x.js

```
setTimeout(function() {
  module.exports = { a: "hello" };
```

```
}, 0);
```

y.js

```
var x = require('./x');
console.log(x.a);
```

x.js 中的模块将会不会在 y.js 中生效。

8.2 module.require(id)

- id: {String}
- return: `require(id)` 一样模块导出对象

实际上 `require(id) === module.require(id)`，想吐槽一下这 api 存在的意义。不过虽然实现上是这样，但 Node 并不这样认为，可以参考一下这个 issue 上的描述：module.require === require

8.3 module.id {String}

模块标识，通常来说是一个完整的绝对路径。

8.4 module.filename {String}

返回一个绝对路径。

8.4 module.loaded {Boolean}

模块是否加载完成。

8.5 module.parent {Object}

加载当前模块的父模块。

8.6 module.children {Array}

当前模块所加载的模块。

9. 关于模块的（几乎）所有内容放在一起

如需得到 `require()` 所加载模块的文件名，可以使用 `require.resolve()` 方法。正面是关于这个方法如何运行的算法伪代码（这段不好写，直接看）：

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
3. LOAD_NODE_MODULES(X, dirname(Y))
4. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.node is a file, load X.node as binary addon.  STOP

LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
2. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
3. If X/index.node is a file, load X/index.node as binary addon.  STOP

LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let ROOT = index of first instance of "node_modules" in PARTS, or 0
3. let I = count of PARTS - 1
4. let DIRS = []
5. while I > ROOT,
   a. if PARTS[I] = "node_modules" CONTINUE
   c. DIR = path join(PARTS[0 .. I] + "node_modules")
   b. DIRS = DIRS + DIR
   c. let I = I - 1
6. return DIRS
```

10. 从全局路径加载

当给 `NODE_PATH` 环境变量赋以一个以冒号（在 windows 下是分号）分割的绝对路径列表，Node 将会从这个列表中去加载模块。另外，Node 还将查找以下的几个位置：

- $HOME/.node_modules
- $HOME/.node_libraries
- $PREFIX/lib/node

在这里 $HOME 是用户的主目录，而 $PREFIX 则是 Node configured 的 `node_prefix` 值。

这些路径大多具备其历史意义，但推荐你把模块放置于项目本地的 `node_modules/` 目录下，因为这会让你的模块加载更快，并且更可靠。

11. 访问本模块

当一个文件直接以 `$ node filename.js` 的模式运行， `require.main` 被设置为 `module` 对象。这意味着你可以知道当前模块是被直接执行，还是因为被访问而执行。比如我们运行 `$ node foo.js` 这时：

```
1. require.main === module; // true
```

如果我们使用 `require('./foo.js')` 则：

```
1. require.main === module; // false
```

另外，由于 `module` 提供了一个 `filename` 属性（一般情况下它等价于 `__filename` ），那么当前运行的程序路径可以通过 `require.main.filename` 来访问。

12. 附录：模块管理贴示

Node 所提供的 `require()` 从很大程度上可以满足应用的设计而不需要修改就能加载到相应的模块。但当模块之前的引用层级不定和程序引用的位置不同，还是可能出现总是。Node 提供一种模块管理的建议：

- /usr/lib/node/foo/1.2.3/ - foo 包，版本 1.2.3.
- /usr/lib/node/bar/4.3.2/ - foo 所依赖的 bar，版本 4.3.2
- /usr/lib/node/foo/1.2.3/node_modules/bar - 链接到 /usr/lib/node/bar/4.3.2/
- /usr/lib/node/bar/4.3.2/node_modules/* - 链接到其他依赖的模块

这样即使遇到循环引用，抑或引用冲突，这些模块也可以去使用某些可用的版本。不过这看起来还是有点乱。或许我们可以考虑下面的方式：

- /foo/
- /foo/node_modules/bar
- /foo/node_modules/bar/node_modules

把 foo 模块的所有依赖模块都放置于目录中，在 foo/ 放置 node_modules 目录来存放依赖模块。考虑一下 derby.js 的这个模块管理设计：

```
node_modules
├── derby
├── express
│   └── node_modules
├── gzippo
│   └── node_modules
└── racer-db-mongo
    └── node_modules
```

其他的 Node 说了一大堆，感觉用处不大。详见：Package Manager Tips