目錄

Nodejs API 中文文档

前言

这份文档的翻译工作始于 2016年 4月初，由于翻译量较大，加之 Node.js 官方版本更新较快，因而目前尚无法跟上官网的（版本）更新节奏，因此不再进一步更新。

推荐大家阅读 Node.js 中文网 维护的同步文档。

简介

这是一份 Node.js API 的中文文档，使用 GitBook 进行构建，适用于无力阅读英文原版的朋友参考阅读。

文档阅读

Gitbook 版(国内镜像)

Gitbook 版

GitHub 版

许可协议

备注

此文档的创建初衷是为了满足个人的学习需求，因此我目前已根据自己的对 Node.js 学习路线的思考对文档的章节顺序进行了适当的调整，此调整不影响正常的查阅。

关于本文档

无论从参考还是从概念的角度来看，本文档的目的都是全面解释 **Node.js API**。文档的每个部分都会介绍一个内置的模块或更高层次的概念。

在某些情况下，属性类型，方法参数和提供给事件处理程序的参数在主题标题下的列表中有详细说明。

每个 `.html` 文档都有一个相应的 `.json` 文档，以结构化的方式呈现相同的信息。这个特性是实验性的，希望能够为一些需要对文档进行程序化操作的 IDE 或者其他工具提供帮助。

每个 `.html` 和 `.json` 文件都是基于源代码 `doc/api/` 目录下的 `.md` 文件生成的。本文档使用 `tools/doc/generate.js` 这个程序生成。`HTML` 模板位于 `doc/template.html`。

目前本文档是基于 *Gitbook* 生成的，如需查阅 `json` 格式的文档请转至官方文档查阅，译者注

如果你在阅读过程中发现文档中的错误，请提交问题(issue)或查阅文档贡献指南中关于如何提交问题的操作说明。

如果你在阅读过程发现翻译上的错误，请提交问题*(issue)*。

稳定度

在整个文档中，你可能会看到某个部分的稳定性提示。Node.js 的 API 仍然会有一些小的变化，随着它的日益成熟，某些部分会比其他部分更可靠。有一部分接受过严格验证，被大量依赖的 API 几乎是不会改变的。也有一些是新增和实验性的，或已知具有危险性并在重新设计过程中的。

稳定性的指标如下：

> 稳定度：0 - 已废弃

这是一个存在问题的特性，目前正计划修改。请不要使用该特性，使用此功能可能会导致警告，不要指望该特性能够向后兼容。

> 稳定度：1 - 试验性

此功能可能会更改，我们会在命令行中进行提示。它可能会在将来版本中更改或删除。

> 稳定度：2 - 稳定

该 API 已被证明是令人满意的。在 npm 中有着很高的使用率，我们在没有绝对的必要下不会对其修改。

> 稳定度：3 - 已锁定

我们只接受对其进行的安全性、性能或修复 BUG 的建议。请不要在修改 API 方面给出提案，因为我们会拒绝这类建议。

JSON 格式输出

> 稳定度：1 - 试验性

每个通过 markdown 生成的 HTML 文件都对应于一个具有相同数据结构的 JSON 文件。

该特性是在 Node.js v0.6.12 中引入的，目前仍是实验性功能。

目前本文档是基于 *Gitbook* 生成的，如需查阅 `json` 格式的文档请转至官方文档查阅，译者注

系统调用和手册页

系统调用定义了用户程序和底层操作系统之间的接口，例如 open(2) 和 read(2) 。Node.js 的函数只是简单的包装了系统调用，就像文档中的 `fs.open()` 。该文档链接到相应的手册页（以下简称手册页），其中描述了该系统调用的工作方式。

警告：一些系统调用，例如 lchown(2) ，是特定于 BSD 系统。这就意味着 `fs.chown()` 只适用于 Mac OS X 和其他的 BSD 派生系统，在 Linux 上是不可用的。

Windows 环境下的大多数系统回调和 Unix 环境下的等效，但有一些可能与 Linux 和 MAC OS X 不同。以一种微妙的关系为例，Windows 环境下有时不可能找到某些 Unix 系统回调的替代方案，详见 Node issue 4760 。

用法和示例

`node [options] [v8 options] [script.js | -e "script"] [arguments]`

有关使用 Node.js 运行脚本的各种选项和方法的相关信息，请参阅文档中的命令行选项章节。

示例：

一个使用 Node.js 编写的输出 “Hello World” 的 Web 服务器 示例：

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

如果要运行这个服务器，需要先将代码保存名为 `example.js` 的文件，并使用 Node.js 来执行：

```
$ node example.js
Server running at http://127.0.0.1:3000/
```

文档中的所有示例都可以使用相同的方式运行。

全局对象(Globals)

- 方法和属性

这些对象在所有模块中都是可用的。有些对象实际上并非在全局作用域内而是在模块作用域内——这种情况会在文档中特别指出。

此处列出的对象特定于 Node.js。有许多内置对象是 JavaScript 语言本身的一部分，它们也可以全局访问。

方法和属性

Buffer 类

添加：v0.1.103

- {Function}

用于处理二进制数据。详见 Buffer 章节。

__dirname

添加：v0.1.27

- {String}

当前执行脚本所在的目录名称。

示例：在 `/Users/mjr` 中运行 `node example.js`：

```
console.log(__dirname);
// 打印: /Users/mjr
```

`__dirname` 实际上不是全局的，而是每个模块内部的。

例如，给出两个模块：`a` 和 `b`，其中 `b` 是 `a` 的依赖文件。目录结构如下：

- `/Users/mjr/app/a.js`
- `/Users/mjr/app/node_modules/b/b.js`

在 `b.js` 中引用 `__dirname` 会返回 `/Users/mjr/app/node_modules/b`，而在 `a.js` 中引用 `__dirname` 则会返回 `/Users/mjr/app`。

上文的 *`/Users/mjr`* 指代用户目录，在实际应用中请使用自己的目录路径进行操作，译者注。

__filename

添加：v0.0.1

- {String}

当前所执行脚本的文件名。这是该脚本文件经过解析后生成的绝对路径。在模块中此变量值是该模块的文件路径。对于主程序而言，这与命令行中使用的文件名未必相同。

示例：在 `/Users/mjr` 中运行 `node example.js`：

```
console.log(__filename);
// 打印: /Users/mjr/example.js
```

`__filename` 实际上不是全局的，而是每个模块内部的。

上文的 *`/Users/mjr`* 指代用户目录，在实际应用中请使用自己的目录路径进行操作，译者注。

global

添加：v0.1.27

- {Object} 全局命名空间对象。

在浏览器中，顶级作用域就是全局作用域。这也意味着在浏览器中，如果在全局作用域内使用 `var something` 将会声明一个全局变量。在 Node.js 中则不同。顶级作用域并非全局作用域；在 Node.js 模块中使用 `var something` 会生成该模块的一个本地变量。

process

添加：v0.1.7

- {Object}

进程对象。详见进程章节。

module

添加：v0.1.16

- {Object}

当前模块的引用。尤其是 `module.exports` 用于定义模块的导出并确保该模块能够通过 `require()` 引入。

`module` 实际上不是全局的，而是每个模块内部的。

有关的详细信息，请参阅模块系统文档。

exports

添加：v0.1.12

`module.exports` 的快捷引用方式。何时使用 `exports` 以及何时使用 `module.exports` 的详细内容，请参阅模块系统文档。

`exports` 实际上不是全局的，而是每个模块内部的。

有关详细信息，请参阅模块系统文档。

require()

添加：v0.1.13

- {Function}

引入模块。详见模块章节。

`require` 实际上不是全局的，而是每个模块内部的。

require.cache

添加：v0.3.0

- {Object}

模块在引入时会缓存到该对象中。如果从该对象中删除键值，下次调用 `require` 时将重新加载相应模块。请注意，这不适用于原生插件，因为重新加载这类插件将导致错误。

require.extensions

添加：v0.3.0 废弃：v0.10.6

> 稳定度：0 - 已废弃

- {Object}

用于指导 `require` 方法如何处理特定的文件扩展名。

将带有扩展名 `.sjs` 文件作为 `.js` 文件处理：

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

在 `废弃` 之前，该列表用于将通过按需编译的非 JavaScript 模块加载到 Node.js 中。然而，实践中有更好的方式实现这一点，比如通过其他的 Node.js 程序来加载模块，或预先将它们编译成 JavaScript 代码。

由于模块系统被锁定，该特性可能永远不会消失。改动它可能会产生细微的错误和复杂性，所以最好保持不变。

请注意，模块系统必须执行与已注册的扩展数成线性比例的文件系统操作次数，才能将一个 `require(...)` 语句解析为文件名。

换句话说，添加扩展会降低模块加载器的执行效率，不鼓励使用该方法。

require.resolve()

添加：v0.3.0

该方法使用 `require()` 的内部机制来查找模块的位置，但不会加载该模块，只返回解析后的文件名。

console

添加：v0.1.100

- {Object}

用于打印 `stdout` 和 `stderr` 。详见控制台章节。

setTimeout(callback, delay[, ...args])

添加：v0.0.1

有关 setTimeout 的描述在定时器章节。

clearTimeout(timeoutObject)

添加：v0.0.1

有关 clearTimeout 的描述在定时器章节。

setInterval(callback, delay[, ...args])

添加：v0.0.1

有关 setInterval 的描述在定时器章节。

clearInterval(intervalObject)

添加：v0.0.1

有关 clearInterval 的描述在定时器章节。

setImmediate(callback[, ...args])

添加：v0.9.1

有关 setImmediate 的描述在定时器章节。

clearImmediate(immediateObject)

添加：v0.9.1

有关 clearImmediate 的描述在定时器章节。

控制台(Console)

> 稳定度：2 - 稳定

`console` 模块提供了一个简单的调试控制台，与 Web 浏览器提供的 JavaScript 控制台机制类似。

该模块导出两个特定组件：

- 一个 `Console` 类，包含类似于 `console.log()` 、 `console.error()` 和 `console.warn()` 这些方法，可以用于写入任何的 Node.js 流。
- 一个全局的 `console` 实例，用于写入 `stdout` 和 `stderr` 。由于该对象是一个全局变量，它可以在没有调用 `require('console')` 的情况下使用。

使用全局 `console` 的示例：

```
console.log('hello world');
// 在 stdout 中打印: hello world
console.log('hello %s', 'world');
// 在 stdout 中打印: hello world
console.error(new Error('Whoops, something bad happened'));
// 在 stderr 中打印: [Error: Whoops, something bad happened]

const name = 'Will Robinson';
console.warn(`Danger ${name}! Danger!`);
// 在 stderr 中打印: Danger Will Robinson! Danger!
```

使用 `Console` 类的示例：

```
const out = getStreamSomehow();
const err = getStreamSomehow();
const myConsole = new console.Console(out, err);

myConsole.log('hello world');
// 在 stdout 中打印: hello world
myConsole.log('hello %s', 'world');
// 在 stdout 中打印: hello world
myConsole.error(new Error('Whoops, something bad happened'));
// 在 stderr 中打印: [Error: Whoops, something bad happened]

const name = 'Will Robinson';
myConsole.warn(`Danger ${name}! Danger!`);
// 在 stderr 中打印: Danger Will Robinson! Danger!
```

虽然 `Console` 类的 API 是根据浏览器的 `console` 对象设计的，但 Node.js 中的 `Console` 并没有完全复制浏览器中的功能。

Console 类

`Console` 类可用于创建具有可配置的输出流的简单记录器，也可以使用 `require('console').Console` 或 `console.Console`：

```
const Console = require('console').Console;
const Console = console.Console;
```

new Console(stdout[, stderr])

通过一个或两个可写流实例创建一个新的 `Console` 对象。`stdout` 是一个用于打印日志或信息输出的可写流。`stderr` 用于输出警告或错误信息。如果 `stderr` 没有正常输出，警告或错误将被发送到 `stdout`。

```
const output = fs.createWriteStream('./stdout.log');
const errorOutput = fs.createWriteStream('./stderr.log');
// 自定义的简单记录器
const logger = new Console(output, errorOutput);
// 像 console 那样使用
var count = 5;
logger.log('count: %d', count);
// 在 stdout.log 中打印: count 5
```

全局的 `console` 是一个特殊的 `Console` 实例，它的输出会发送到 process.stdout 和 process.stderr。它相当于调用：

```
new Console(process.stdout, process.stderr);
```

console.log([data][, ...args])

添加：v0.1.100

使用换行符将信息打印到 `stdout`。可以传多个参数，第一个作为主要信息，其余的参数作为类似于 `printf(3)` 中的替换值（这些参数都会传给 util.format() 进行处理）。

```
var count = 5;
console.log('count: %d', count);
// 在 stdout 中打印: count: 5
console.log('count:', count);
// 在 stdout 中打印: count: 5
```

如果在第一个字符串中没有找到格式化元素（如，`%d`），那么 util.inspect() 会在每个参数上调用并将结果字符串值拼在一起。详见 util.format()。

console.info([data][, ...args])

添加：v0.1.100

`console.info()` 函数是 console.log() 的别名。

console.error([data][, ...args])

添加：v0.1.100

使用换行符将信息打印到 `stderr`。可以传多个参数，第一个作为主要信息，其余的参数作为类似于 `printf(3)` 中的替换值（这些参数都会传给 util.format() 进行处理）。

```
const code = 5;
console.error('error #%d', code);
// 在 stderr 中打印: error #5
console.error('error', code);
// 在 stderr 中打印: error 5
```

如果在第一个字符串中没有找到格式化元素（如，`%d`），那么 util.inspect() 会在每个参数上调用并将结果字符串值拼在一起。详见 util.format()。

console.warn([data][, ...args])

添加：v0.1.100

`console.warn()` 函数是 console.error() 的别名。

console.dir(obj[, options])

添加：v0.1.101

该函数在 `obj` 上使用 util.inspect() 并将结果字符串打印到 `stdout` 中。此函数会绕过任何定义在 `obj` 上的 `inspect()` 方法。可选 `options` 对象可以传一些用于格式化字符串的内容：

- `showHidden` - 如果为 `true`，那么该对象中的不可枚举属性和 `Symbol` 属性也会显示，默认为 `false`。
- `depth` - 告诉 `util.inspect()` 函数在格式化对象时递归多少次。这对于检查大型复杂对象很有用。默认为 `2`。可以通过设置为 `null` 来实现无限递归。
- `colors` - 如果为 `true`，那么输出结果将使用 ANSI 颜色代码。默认为 `false`。这里的颜色是可定制的，详见定制 util.inspect() 颜色。

console.trace(message[, ...args])

添加：v0.1.104

通过 `stderr` 打印字符串 `'Trace :'`，以及后面通过 util.format() 格式化的消息和堆栈跟踪在代码中的当前位置。

```
console.trace('Show me');
// Prints: (stack trace will vary based on where trace is called)
//  Trace: Show me
//    at repl:2:9
//    at REPLServer.defaultEval (repl.js:248:27)
//    at bound (domain.js:287:14)
//    at REPLServer.runBound [as eval] (domain.js:300:12)
//    at REPLServer.<anonymous> (repl.js:412:12)
//    at emitOne (events.js:82:20)
//    at REPLServer.emit (events.js:169:7)
//    at REPLServer.Interface._onLine (readline.js:210:10)
//    at REPLServer.Interface._line (readline.js:549:8)
//    at REPLServer.Interface._ttyWrite (readline.js:826:14)
```

console.assert(value[, message][, ...args])

添加：v0.1.101

一个简单的断言测试，验证 `value` 是否为真。如果不是，则抛出一个 `AssertionError` 。如果提供错误 `message` 参数，则使用 util.format() 格式化并作为错误信息输出。

```
console.assert(true, 'does nothing');
// OK
console.assert(false, 'Whoops %s', 'didn\'t work');
// AssertionError: Whoops didn't work
```

注意：*`console.assert()`* 方法在 *Node.js* 中的实现方式和用在浏览器中的 *`console.assert()`* 方法是不一样的。

特别是在浏览器中调用假的断言后，`console.assert()` 会导致 `message` 被打印到控制台，但不会中断后续代码的执行。而在 Node.js 中，一个假的断言将导致抛出一个 `AssertionError` 错误。

可以通过扩展 Node.js 的 `console` 并覆盖 `console.assert()` 方法来实现浏览器中实现的类似功能。

在以下示例中，会创建一个简单的模块，该模块扩展并覆盖了 Node.js 中 `console` 的默认行为。

```
'use strict';

// 用一种新的没有采用猴子补丁实现的 assert 来创建一个简单的 console 扩展。
const myConsole = Object.setPrototypeOf({
    assert(assertion, message, ...args) {
        try {
            console.assert(assertion, message, ...args);
        } catch (err) {
            console.error(err.stack);
        }
    }
}, console);

module.exports = myConsole;
```

然后可以用于直接替换内置的控制台：

```
const console = require('./myConsole');
console.assert(false, 'this message will print, but no error thrown');
console.log('this will also print');
```

console.time(label)

添加：v0.1.104

启动一个定时器用以计算一个操作的持续时间。定时器由唯一的 `label` 标识。当调用 console.timeEnd() 时，可以使用相同的 `label` 来停止定时器并以毫秒为单位将持续时间输出到 `stdout` 。定时器持续时间精确到亚毫秒。

console.timeEnd(label)

添加：v0.1.104

停止之前通过 console.time() 启动的定时器并将结果打印到 `stdout` ：

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
    ;
}
console.timeEnd('100-elements');
// 打印 100-elements: 225.438ms
```

注意：从 *Node.js v6.0.0* 开始， *`console.timeEnd()`* 删除了计时器以避免泄漏。在旧版本上，依然保留着计时器，这会允许对同一标签调用 *`console.timeEnd()`* 多次。此功能是非预期的，不再受到支持。

异步与同步的控制台

`console` 函数通常是异步的，除非目标对象是一个文件。带有高速磁盘的操作系统通常采用回写缓存；写入阻塞应该是一个非常罕见的情况，但它确实有可能发生。

此外，作为对 MAC OS X 中极小的（1kb）的缓存大小限制的一种解决方案，在 MAC OS X 上输出到 TTY（终端）时，控制台功能会遭到阻塞。这是为了防止 `stdout` 和 `stderr` 交叉在一起。

定时器(Timers)

> 稳定度：3 - 已锁定

`timer` 模块暴露了一个全局的 API 用于调度在某个未来时间段内调用的函数。因为定时器函数是全局的，所以你没有必要调用 `require('timers')` 来使用该 API。

Node.js 中的计时器函数实现了与 Web 浏览器提供的定时器类似的 API，但它使用了基于 Node.js 事件循环构建的不同内部实现。

预定定时器

Node.js 中的计时器是一种会在一段时间后调用给定的函数的内部构造。定时器函数会在何时被调用，取决于用来创建定时器的方法以及 Node.js 事件循环是否正在做其他工作。

setTimeout(callback, delay[, ...args])

添加：v0.0.1

- `callback` {Function} 当定时器到点时回调的函数。
- `delay` {Number} 在调用 `callback` 之前等待的毫秒数。
- `...args` {Any} 在调用 `callback` 时要传递的可选参数。

在 `delay` 毫秒之后预定执行一次性的 `callback`。返回一个用于 clearTimeout() 的 `Timeout`。

`callback` 可能不会精确地在 `delay` 毫秒被调用。Node.js 不能保证回调被触发的确切时间，也不能保证它们的顺序。回调会在尽可能接近所指定的时间上调用。

注意：当 `delay` 大于 `2147483647` 或小于 `1` 时，`delay` 会被设置为 `1`。

如果 `callback` 不是一个函数，将会抛出一个 TypeError。

setInterval(callback, delay[, ...args])

添加：v0.0.1

- `callback` {Function} 当定时器到点时回调的函数。
- `delay` {Number} 在调用 `callback` 之前等待的毫秒数。
- `...args` {Any} 在调用 `callback` 时要传递的可选参数。

预定每隔 `delay` 毫秒重复执行 `callback`。返回一个用于 clearInterval() 的 `Timeout`。

注意：当 `delay` 大于 `2147483647` 或小于 `1` 时，`delay` 会被设置为 `1`。

如果 `callback` 不是一个函数，将会抛出一个 TypeError。

setImmediate(callback[, ...args])

添加：v0.9.1

- `callback` {Function} 在当前的 Node.js 事件循环回合结束时调用该函数。
- `...args` {Any} 在调用 `callback` 时要传递的可选参数。

预定“立即”执行 `callback`，它是在 I/O 事件的回调之后并在使用 setTimeout() 和 setInterval() 创建的计时器之前被触发。返回一个用于 clearImmediate() 的 `Immediate`。

当多次调用 `setImmediate()` 时，回调函数会按照它们的创建顺序依次执行。每个事件循环迭代都会处理整个回调队列。如果立即定时器正在执行回调中排队，那么该定时器直到下一个事件循环迭代之前将不会被触发。

如果 `callback` 不是一个函数，将会抛出一个 TypeError。

取消定时器

- clearTimeout(timeout)
- clearInterval(timeout)
- clearImmediate(immediate)

每个 setTimeout()、setInterval() 和 setImmediate() 都会返回表示预定计时器的对象。这些可用于取消定时器并防止其触发。

clearTimeout(timeout)

添加：v0.0.1

- `timeout` {Timeout} 由 setTimeout() 返回的 `Timeout` 对象。

取消由 setTimeout() 创建的 `Timeout` 对象。

clearInterval(timeout)

添加：v0.0.1

- `timeout` {Timeout} 由 setInterval() 返回的 `Timeout` 对象。

取消由 setInterval() 创建的 `Timeout` 对象。

clearImmediate(immediate)

添加：v0.9.1

- `immediate` {Immediate} 由 setImmediate() 返回的 `Immediate` 对象。

取消由 setImmediate() 创建的 `Immediate` 对象。

Timeout 类

此对象在内部创建，并从 setTimeout() 和 setInterval() 返回。它可以分别传递给 clearTimeout() 或 clearInterval() 以便取消计划的动作。

默认情况下，当使用 setTimeout() 或 setInterval() 调度定时器时，只要定时器仍处于活动状态，Node.js 事件循环将继续运行。每个由这些函数返回的 `Timeout` 对象都可以导出可用于控制此默认行为的 `timeout.ref()` 和 `timeout.unref()` 函数。

timeout.unref()

添加：v0.9.1

当被调用时，活动的 `Timeout` 对象将不需要 Node.js 事件循环保持活动。如果没有其他活动保持事件循环运行，那么该进程可以在 `Timeout` 对象的回调被调用之前退出。调用 `timeout.unref()` 多次将没有效果。

注意：调用 `timeout.unref()` 创建一个内部定时器，将唤醒 Node.js 的事件循环。创建太多这类定时器可能会对 Node.js 应用程序的性能产生负面影响。

返回对 `Timeout` 的引用。

timeout.ref()

添加：v0.9.1

调用时，只要 `Timeout` 处于活动状态就请求 Node.js 不要退出事件循环。调用 `timeout.ref()` 多次将没有效果。

注意：默认情况下，所有 `Timeout` 对象都处于 `'ref'd'` 状态，通常不需要调用 `timeout.ref()`，除非之前调用了 `timeout.unref()`。

返回对 `Timeout` 的引用。

Immediate 类

此对象在内部创建，并从 setImmediate() 返回。它可以传递给 clearImmediate() 以便取消计划的操作。

模块(Modules)

> 稳定度：3 - 已锁定

Node.js 有一个简单的模块加载系统。在 Node.js 中，文件和模块是一一对应的（每个文件被视为一个单独的模块）。举个例子，`foo.js` 加载同一目录下的 `circle.js` 模块。

`foo.js` 的内容：

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

`circle.js` 的内容:

```
const PI = Math.PI;

exports.area = (r) => PI * r * r;

exports.circumference = (r) => 2 * PI * r;
```

`circle.js` 模块导出了 `area()` 和 `circumference()` 两个函数。为了将函数和对象添加进你的模块根，你可以将它们添加到特殊的 `exports` 对象下。

模块内的本地变量是私有的，因为模块被 Node.js 包装在一个函数中（详见模块包装器）。在这个例子中，变量 `PI` 就是 `circle.js` 私有的。

如果你希望将你的模块根导出为一个函数（比如构造函数）或一次导出一个完整的对象而不是每一次都创建一个属性，请赋值给 `module.exports` 而不是 `exports` 。

下面，我将在 `bar.js` 中使用 `square` 模块导出的构造函数。

```
const square = require('./square.js');
var mySquare = square(2);
console.log(`The area of my square is ${mySquare.area()}`);
```

`square` 模块定义在 `square.js` 中：

```
// 赋值给 exports 将不会修改模块，必须使用 module.exports
module.exports = (width) => {
    return {
        area: () => width * width
    };
}
```

模块系统在 `require("module")` 中实现。

方法和属性

---

module 对象

添加：v0.1.16

- {Object}

在每一个模块中，自由变量 `module` 是对表示当前模块的对象的引用。为了方便起见，`module.exports` 也可以通过模块全局的 `exports` 对象访问到。`module` 实际上不是全局的，而是每个模块内部的。

module.id

添加：v0.1.16

- {String}

模块的标识符。通常是完全解析后的文件名。

module.filename

添加：v0.1.16

- {String}

模块完全解析后的文件名。

module.loaded

添加：v0.1.16

- {Boolean}

显示模块是否已经加载完成，或正在加载中。

module.parent

添加：v0.1.16

- {Object} Module 对象

在模块中最先引入的模块。

module.children

添加：v0.1.16

- {Array}

需要引入该模块的模块对象。

module.exports

添加：v0.1.16

- {Object}

`module.exports` 对象由模块系统创建。有时候这是难以接受的，许多人都希望他们的模块成为某个类的实例。为了实现这一点，你需要将要导出的对象赋值给 `module.exports`。需要注意的是如果你将需要导出的对象赋值给 `exports` 只会简单的绑定到本地变量 `exports` 上，这很可能不是你想要的结果。

例如，假设我们创建了一个名为 `a.js` 模块：

```
const EventEmitter = require('events');

module.exports = new EventEmitter();

// 处理一些工作，并在一段时间后从模块自身内部发出 'ready' 事件。
setTimeout(() => {
    module.exports.emit('ready');
}, 1000);
```

然后，在另一个文件中我们可以这么做：

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

注意，必须立即完成对 `module.expors` 的赋值，而不能在任何回调中完成。这样是不起作用的：

x.js：

```
setTimeout(() => {
    module.exports = { a: 'hello' };
}, 0);
```

y.js：

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

exports 的别名

添加：v0.1.16

变量 `exports` 是模块内部在最开始生成的指向 `module.exports` 的引用。对于任何变量而言，如果你为其赋一个新值，它将不再绑定到以前的值。

为了解释这个情况，我们想象以下对 `require()` 的假设：

```
function require(...) {
    // ...
    ((module, exports) => {
        // 在这里写你的模块代码
        // 重新分配 export，export 不在是 module.exports 的快捷方式，并且不再导出任何内容。
        exports = some_func;
        // 使你的模块导出内容
        module.exports = some_func;
    })(module, module.exports);

    return module;
}
```

原则上，如果你理解不了 `exports` 和 `module.exports` 之间的关系，请忽略 `exports`，只使用 `module.exports`。

module.require(id)

添加：v0.5.1

- `id` {String}
- 返回：{Object} 已解析的模块的 `module.exports`

`module.require` 方法提供了一种像 `require()` 那样从原始模块加载一个模块的方式。

注意，为了做到这一点，你必须获取一个 `module` 对象的引用。因为 `require()` 会返回 `module.exports`，并且 `module` 是一个典型的只能在特定模块作用域内有效的变量，如果想要使用它，就必须明确的导出。

核心模块

Node.js 中有些模块是编译成二进制的。这些模块在本文档的其他地方有更详细的描述。

核心模块定义在 Node.js 源代码的 `lib/` 目录下。

`require()` 总是会优先加载核心模块。例如，`require('http')` 始终返回内置的 HTTP 模块，即使有同名文件。

文件模块

如果按确切的文件名没有查找到该模块，那么 Node.js 会尝试添加 `.js` 和 `.json` 拓展名进行查找，如果还未找到，最后会尝试添加 `.node` 的拓展名进行查找。

`.js` 文件会被解析为 JavaScript 文本文件，`.json` 文件会被解析为 JSON 文本文件。`.node` 文件会被解析为通过 `dlopen` 加载的编译后的插件模块。

请求的模块以 `'/'` 为前缀，则表示绝对路径。例如，`require('/home/marco/foo.js')` 将会加载的是 `/home/marco/foo.js` 文件。

请求的模块以 `'./'` 为前缀，则表示相对于调用 `require()` 的文件的路径。也就是说，`circle.js` 必须和 `foo.js` 在同一目录下以便于 `require('./circle')` 找到它。

当没有以 `'/'`、`'./'` 或 `'../'` 开头来表示文件时，这个模块必须是“核心模块”或加载自 `node_modules` 目录。

如果给定的路径不存在，`require()` 会抛出一个 `code` 属性为 `'MODULE_NOT_FOUND'` 的错误。

模块包装器

在执行模块代码之前，Node.js 会使用一个如下所示的函数包装器将其包装：

```
(function (exports, require, module, __filename, __dirname) {
// 你的模块代码实际上应该在这里
});
```

通过这样做，Node.js 实现了以下几点：

- 它保持顶级变量（用 `var`、`const` 或 `let` 定义）作用域是在模块内部而不是在全局对象上。
- 它有助于提供一些实际上特定于模块的全局变量，例如：
  - 实现者可以用 `module` 和 `exports` 对象从模块中导出值。
  - 快捷变量 `__filename` 和 `__dirname` 包含了模块的文件名和目录的绝对路径。

将文件夹作为模块

可以把程序和库放到一个单独的文件夹里，并提供单一入口来指向它。有三种方式可以将文件夹传递给 `require()` 作为参数。

第一种方式是在文件夹的根目录下创建一个 `package.json` 的文件，它指定一个 `main` 模块。以下是一个 `package.json` 的文件示例：

```
{
    "name": "some-library",
    "main": "./lib/some-library.js"
}
```

如果这是在 `./some-library` 文件夹中，那么 `require('./some-library')` 将尝试加载 `./some-library/lib/some-library.js` 。

这就是 Node.js 处理 `package.json` 文件的方式。

注意：如果 `package.json` 中 `"main"` 条目指定的文件丢失，Node.js 将无法解析该模块，并抛出一个找不到该模块的默认错误：

```
Error: Cannot find module 'some-library'
```

如果目录里没有 `package.json` 这个文件，那么 Node.js 就会尝试去加载这个目录下的 `index.js` 或 `index.node` 文件。例如，如果上面例子中没有 `package.json` ，那么 `require('./some-library')` 就将尝试加载以下文件：

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

从 `node_modules` 文件夹加载模块

如果传递给 `require()` 的模块标识符不是核心模块，也没有以 `'/'` 、 `'../'` 或 `'./'` 开头，那么 Node.js 会从当前模块的父目录开始，尝试在它的 `/node_modules` 文件夹里加载相应模块。Node.js 不会添加 `node_modules` 到已经以 `node_modules` 结尾的路径上。

如果没有找到，那么就再向上一级目录移动，直到文件系统的根目录为止。

例如，假设在 `'/home/ry/projects/foo.js'` 文件里调用了 `require('bar.js')` ，那么 Node.js 查找其位置的顺序依次为：

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

这允许程序本地化它们的依赖，避免它们产生冲突。

通过在模块名称后包含路径后缀，你可以请求特定的文件或分布式的子模块。例如， `require('example-module/path/to/file')` 将被解析为相对于 `example-module` 所在位置的 `path/to/file` 。后缀路径同样遵循模块路径的解析规则。

从全局文件夹加载模块

如果 `NODE_PATH` 环境变量设置为一个以冒号分割的绝对路径列表。如果在其他位置找不到模块时，那么 Node.js 将会从这些路径中搜索。（注意：在 Windows 操作系统中，`NODE_PATH` 是以分号间隔的。）

`NODE_PATH` 最初创建用以支持从不同路径加载模块，它不会在当前模块解析算法运行前被使用。

虽然目前仍然支持 `NODE_PATH` ，但在 Node.js 生态系统约定依赖模块的存放路径后已经很少用到了。有时，部署依赖 `NODE_PATH` 的模块，会在别人不知道必须设置 `NODE_PATH` 的情况下出现异常行为。有时，模块的依赖关系会发生变化，导致在搜索 `NODE_PATH` 时加载了不同的版本（甚至不同的模块）。

此外，Node.js 还会搜索以下路径：

1. `$HOME/.node_modules`
2. `$HOME/.node_libraries`
3. `$PREFIX/lib/node`

其中 `$HOME` 是用户的主目录，`$PREFIX` 是 Node.js 里配置的 `node_prefix` 。

这些大多是由于历史原因产生的。强烈建议你们将所有的依赖模块放在 **`node_modules`** 文件夹中。它们将会更快、更可靠地被加载。

循环

当循环调用 `require()` 时，模块可能在未完成执行时被返回。

考虑这样一种情况:

`a.js` :

```
console.log('a starting');
exports.done = false;
const 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;
const 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');
const a = require('./a.js');
const 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` 时，`a.js` 反向加载 `b.js`。那时，`b.js` 会尝试去加载 `a.js`。为了防止无限的循环，`a.js` 会返回一个 `exports` 对象的 未完成副本 给 `b.js` 模块。之后 `b.js` 完成加载，并将 `exports` 对象返回给 `a.js` 模块。

当 `main.js` 加载这两个模块时，它们都已经完成加载了。因此，该程序的输出将会是：

```
$ 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
```

如果你的程序里有循环的依赖模块，请确保它们按计划执行。

缓存

模块在第一次加载后会被缓存。这也意味着（类似其他缓存机制）如果每次调用 `require('foo')` 都解析到同一个文件，那么它将返回相同的对象。

多次调用 `require(foo)` 未必会导致模块中的代码执行多次。这是一个重要的特性。借助它，可以返回“部分完成”的对象，从而允许传递依赖性加载，即使它们可能导致循环。

如果你希望一个模块能够执行多次，那么，可以导出一个函数，然后多次调用该函数。

模块缓存的注意事项

模块是基于其解析的文件名进行缓存的。由于调用位置的不同，同一模块可能被解析成不同的文件名（比如从 `node_modules` 文件夹加载），如果它被解析成不同的文件时，就不能保证 `require('foo')` 总能返回完全相同的对象。

此外，在不区分大小写的文件系统或操作系统中，被解析成不同的文件名可以指向同一个文件，但缓存仍然会将它们视为不同的模块，并将重新加载该文件多次。例如：`require('./foo')` 和 `require('./FOO')` 返回两个不同的对象，而不会管 `./foo` 和 `./FOO` 是否是相同的文件。

访问主模块

当 Node.js 直接运行一个文件时，`require.main` 就被设置为它的 `module`。这意味着你可以直接在测试中确定文件是否已运行。

```
require.main === module
```

对于 `foo.js` 文件而言，通过 `node foo.js` 运行则为 `true`；通过 `require('./foo')` 运行则为 `false`。

因为 `module` 提供了一个 `filename` 属性（通常等于 `__filename`），所以可以通过 `require.main.filename` 来获取当前应用程序的入口点。

总的来说...

想要获取调用 `require()` 时加载的确切的文件名，请使用 `require.resolve()` 函数。

综上所述，以下用伪代码的形式来表述 require.resolve 中的高级算法是如何工作的：

```
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.json is a file, parse X.json to a JavaScript Object.  STOP
4. 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.json is a file, parse X/index.json to a JavaScript object. STOP
4. 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 I = count of PARTS - 1
3. let DIRS = []
4. while I >= 0,
   a. if PARTS[I] = "node_modules" CONTINUE
   b. DIR = path join(PARTS[0 .. I] + "node_modules")
   c. DIRS = DIRS + DIR
   d. let I = I - 1
5. return DIRS
```

附录：包管理器的技巧

Node.js 的 `require()` 函数的语义被设计的足够通用化，可以支持支持许多合理的目录结构。包管理器程序（如 `dpkg`、`rpm` 和 `npm`）将有可能不用修改就能够从 Node.js 模块中构建本地包。

接下来我们将给你一个可行的目录结构建议：

假设我们想要在 `/usr/lib/node/<some-package>/<some-version>` 目录中保存特定版本的包的内容。

包可以依赖于其他包。为了安装包 `foo`，可能需要安装指定版本的 `bar` 包。`bar` 包也可能具有依赖关系，并且在某些情况下依赖关系可能发生冲突或形成循环。

因为 Node.js 会查找它所加载的模块的真实路径（也就是说会解析符号链接），然后在 `node_modules` 目录中查询依赖关系，如下所述，这种情况使用以下体系结构很容易解决：

- `/usr/lib/node/foo/1.2.3/` - `foo` 1.2.3 版本的包内容
- `/usr/lib/node/bar/4.3.2/` - `foo` 包所依赖的 `bar` 的包内容
- `/usr/lib/node/foo/1.2.3/node_modules/bar` - 指向 `/usr/lib/node/bar/4.3.2/` 的符号链接
- `/usr/lib/node/bar/4.3.2/node_modules/*` - 指向 `bar` 包所依赖的包的符号链接

因此，即便存在循环依赖或依赖冲突，每个模块还是可以获得它所依赖的包的一个可用版本。

当 `foo` 包中的代码调用 `require('bar')`，将获得符号链接 `/usr/lib/node/foo/1.2.3/node_modules/bar` 指向的版本。然后，当 `bar` 包中的代码调用 `require('queue')` ，将会获得符号链接 `/usr/lib/node/bar/4.3.2/node_modules/quux` 指向的版本。

此外，为了进一步优化模块搜索过程，不要将包直接放在 `/usr/lib/node` 目录中，而是将它们放在 `/usr/lib/node_modules/<name>/<version>` 目录中。这样在找不到依赖包的情况下，Node.js 就不会在 `/usr/node_modules` 或 `/node_modules` 目录中查找了。

为了使模块在 Node.js 的 REPL 中可用，你可能需要将 `/usr/lib/node_modules` 目录加入到 `$NODE_PATH` 环境变量中。由于在 `node_modules` 目录中搜索模块使用的是相对路径，使得调用 `require()` 获得的是基于真实路径的文件，因此包本身可以放在任何位置。

事件(Events)

> 稳定度：2 - 稳定

大多数 Node.js 核心 API 都是采用惯用的异步事件驱动架构，其中某些类型的对象（称为“触发器”）周期性地发出命名事件来调用函数对象（“监听器”）。

例如：net.Server 对象会在每次有新连接时发出事件；fs.readStream 对象会在文件被打开时发出事件；stream 对象在每当在数据可读时发出事件。

所有能发出事件的对象都是 `EventEmitter` 类的实例。这些对象公开了一个 `eventEmitter.on()` 函数，它允许将一个或多个函数附加到由该对象发出的命名事件上。通常情况下，事件名称是小写驼峰式 (camel-cased) 字符串，但也可以使用任何有效的 JavaScript 属性名。

每当 `EventEmitter` 发出事件，所有附加到特定事件上的函数都被同步调用。所有由监听器回调返回的值都会被忽略并被丢弃。

以下例子展示了一个只有单个监听器的简单的 `EventEmitter` 实例。`eventEmitter.on()` 用于注册监听器，`eventEmitter.emit()` 用于触发事件。

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
    console.log('发生了一个事件!');
});
myEmitter.emit('event');
```

EventEmitter 类

添加：v0.1.26

`EventEmitter` 类由 `events` 模块定义和公开：

```
const EventEmitter = require('events');
```

所有的事件触发器都会在新的监听器被添加时触发 `'newListener'` 事件；在一个监听器被移除时触发 `'removeListener'` 事件。

'newListener' 事件

添加：v0.1.26

- `eventName` {String} | {Symbol} 要监听的事件名称
- `listener` {Function} 回调函数

`EventEmitter` 实例在将监听器添加到其内部监听器数组之前会触发自身的 `'newListener'` 事件。

注册了 `'newListener'` 事件的监听器将被传递事件名称并添加一个监听器的引用。

事实上，在事件触发前添加监听器会有一个微妙但重要的副作用：任何额外被注册成相同名称的监听器在被添加新的监听器前都会触发内部的 `'newListener'` 回调。

```
const myEmitter = new MyEmitter();
// 只进行一次，所以不用担心无限循环
myEmitter.once('newListener', (event, listener) => {
    if (event === 'event') {
        // 在前面插入新的监听器
        myEmitter.on('event', () => {
            console.log('B');
        });
    }
});
myEmitter.on('event', () => {
    console.log('A');
});
myEmitter.emit('event');
// 打印：
//   B
//   A
```

'removeListener' 事件

添加：v0.9.3

- `eventName` {String} | {Symbol} 事件名
- `listener` {Function} 回调函数

`'removeListener'` 事件在监听器被移除后发出。

EventEmitter.defaultMaxListeners

添加：v0.11.2

任何单一事件默认都可以注册最多 10 个监听器。可以使用 emitter.setMaxListeners(n) 方法来单个 `EventEmitter` 实例更改此限制。可以使用 `EventEmitter.defaultMaxListeners` 属性更改所有 `EventEmitter` 实例的默认设置。

请谨慎设置 `EventEmitter.defaultMaxListeners` 属性，因为这个改变会影响到所有的 `EventEmitter` 实例，包括那些之前创建的实例。因而，调用 emitter.setMaxListeners(n) 仍然优于设置 `EventEmitter.defaultMaxListeners` 。

请注意，这不是一个硬性限制。`EventEmitter` 实例允许添加更多的监听器，但会向 `stderr` 输出跟踪警告：表明已经检测到一个 `possible EventEmitter memory leak` 错误。对于任何 `EventEmitter` 实例单独使用 `emitter.getMaxListeners()` 和 `emitter.setMaxListeners()` 方法可以暂时避免此警告：

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', () => {
    // 处理一些事情
    emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1, 0));
});
```

--trace-warnings 命令行标志可用于显示此类警告的堆栈跟踪。

可以使用 process.on('warning') 检查发出的警告，并具有额外的 `emitter`、`type` 和 `count` 属性，分别代表事件发生器的引用，事件的名称和附加的监听器的数量。它的 `name` 属性设置为 `'MaxListenersExceededWarning'`。

EventEmitter.listenerCount(emitter, eventName)

添加：v0.9.12 废弃：v4.0.0

> 稳定度：0 - 已废弃：请使用 emitter.listenerCount() 替代。

返回给定 `emitter` 上注册的给定 `eventName` 的监听器数量的类方法。

```
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {});
myEmitter.on('event', () => {});
console.log(EventEmitter.listenerCount(myEmitter, 'event'));
// 打印：2
```

emitter.on(eventName, listener)

添加：v0.1.101

- `eventName` {String} | {Symbol} 事件名
- `listener` {Function} 回调函数

将侦听器函数添加到名为 `eventName` 的事件的 `listener` 数组的末尾。不会检测 `listener` 是否已被添加。多次调用传递了相同的 `eventName` 和 `listener` 的组合会导致 `listener` 被添加和调用多次。

```
server.on('connection', (stream) => {
    console.log('有人连接！');
});
```

返回一个当前 `EventEmitter` 的引用以便链式调用。

默认情况下，事件监听器按照添加的顺序依次调用。`emitter.prependListener()` 方法可以用作将事件监听器添加到 `listeners` 数组开头的可选方法。

```
const myEE = new EventEmitter();
myEE.on('foo', () => console.log('a'));
myEE.prependListener('foo', () => console.log('b'));
myEE.emit('foo');
// 打印：
//   b
//   a
```

emitter.once(eventName, listener)

添加：v0.3.0

- `eventName` {String} | {Symbol} 事件名
- `listener` {Function} 回调函数

给名为 `eventName` 的事件添加一个一次性的 `listener` 函数。下一次发出 `eventName` 事件时，此监听器将被移除，并在随后调用。

```
server.once('connection', (stream) => {
    console.log('哈，我们有第一个用户！');
});
```

返回一个当前 `EventEmitter` 的引用以便链式调用。

默认情况下，事件监听器按照添加的顺序依次调用。`emitter.prependOnceListener()` 方法可以用作将事件监听器添加到 `listeners` 数组开头的可选方法。

```
const myEE = new EventEmitter();
myEE.on('foo', () => console.log('a'));
myEE.prependOnceListener('foo', () => console.log('b'));
myEE.emit('foo');
// 打印：
//   b
//   a
```

emitter.addListener(eventName, listener)

添加：v0.1.26

emitter.on(eventName, listener) 的别名。

emitter.prependListener(eventName, listener)

添加：v6.0.0

- `eventName` {String} | {Symbol} 事件名
- `listener` {Function} 回调函数

将监听器函数添加到名为 `eventName` 事件的 `listeners` 数组的开头。不会检测 `listener` 是否已被添加。多次调用传递了相同的 `eventName` 和 `listener` 的组合会导致 `listener` 被添加和调用多次。

```
server.prependListener('connection', (stream) => {
    console.log('有人连接！');
});
```

返回一个当前 `EventEmitter` 的引用以便链式调用。

emitter.prependOnceListener(eventName, listener)

添加：v6.0.0

- `eventName` {String} | {Symbol} 事件名
- `listener` {Function} 回调函数

将名为 `eventName` 事件的一次性的 `listener` 函数添加到 `listeners` 数组的开头。下一次发出 `eventName` 事件时，此监听器将被移除，并在随后调用。

```
server.prependOnceListener('connection', (stream) => {
    console.log('哈，我们有第一个用户！');
});
```

返回一个当前 `EventEmitter` 的引用以便链式调用。

emitter.removeListener(eventName, listener)

添加：v0.1.26

从名为 `eventName` 的事件的监听器数组中移除特定的 `listener`。

```
const callback = (stream) => {
    console.log('有人连接！');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener` 最多只会从当前的监听器数组里移除一个监听器实例。如果任何单一的监听器多次添加特定的 `eventName` 到监听器数组中，必须多次调用 `removeListener` 才能移除每个实例。

请注意，一旦发出事件，所有关联到它的监听器将被按顺序依次触发。这也意味着，任何的 `removeListener()` 或 `removeAllListeners()` 在调用触发后和最后一个监听器执行完毕前不会从 `emit()` 过程中移除它们。随后的事件会像预期的那样发生。

```
const myEmitter = new MyEmitter();

const callbackA = () => {
    console.log('A');
    myEmitter.removeListener('event', callbackB);
};

const callbackB = () => {
    console.log('B');
};

myEmitter.on('event', callbackA);

myEmitter.on('event', callbackB);

// 在 callbackA 中移除监听器 callbackB，但它仍然会被调用
// 内部监听器数组此时触发 [callbackA, callbackB]
myEmitter.emit('event');
// 打印：
//   A
//   B

// callbackB 现在被移除了
// 内部监听器数组为 [callbackA]
myEmitter.emit('event');
// 打印：
//   A
```

因为监听器是使用内部数组进行管理的，所以调用它将改变在监听器被移除后注册的任何监听器的位置索引。虽然这不会影响监听器的调用顺序，但也意味着由 emitter.listeners() 方法返回的任何监听器副本都将需要被重新创建。

返回一个当前 `EventEmitter` 的引用以便链式调用。

emitter.removeAllListeners([eventName])

添加：v0.1.26

移除全部或某些特定 `eventName` 的监听器。

请注意，在代码中移除其他地方添加的监听器是一个不好的做法，尤其是由其他组件或模块（如，套接字或文件流）创建的 `EventEmitter` 实例。

返回一个当前 `EventEmitter` 的引用以便链式调用。

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

添加：v0.1.26

按照监听器的注册顺序同步调用每个以 `eventName` 注册的监听器，并将额外的参数传递给它们。

如果事件有监听器存在就返回 `true` ，否则返回 `false` 。

emitter.listeners(eventName)

添加：v0.1.26

返回名为 `eventName` 的事件的监听器数组的副本。

```
server.on('connection', (stream) => {
    console.log('有人连接！');
});
console.log(util.inspect(server.listeners('connection')));
// 打印：[ [Function] ]
```

emitter.listenerCount(eventName)

添加：v3.2.0

- `eventName` {String} | {Symbol} 事件名

返回正在监听名为 `eventName` 的事件的监听器数量。

emitter.setMaxListeners(n)

添加：v0.3.5

默认情况下，如果为特定事件添加了超过 `10` 个监听器，`EventEmitter` 将打印警告。此限制在寻找内存泄露时非常有用。显然，并不是所有的事件都要被仅限为 `10` 个。`emitter.setMaxListeners()` 方法允许修改特定的 `EventEmitter` 实例的限制数量。如果不想限制监听器的数量，可以将该值设置为 `Infinity`（或 `0`）。

返回一个当前 `EventEmitter` 的引用以便链式调用。

emitter.getMaxListeners()

添加：v1.0.0

返回当前 `EventEmitter` 实例的最大监听器数量，该值可以通过 emitter.setMaxListeners(n) 或 EventEmitter.defaultMaxListeners 的默认值设置。

错误事件

当 `EventEmitter` 实例中发生错误时，典型的行为就是触发一个 `'error'` 事件。这些在 Node.js 中被视为特殊情况。

如果 `EventEmitter` 实例没有注册过至少一个监听器，当一个 `'error'` 事件触发时，将抛出这个错误，打印堆栈跟踪，并退出 Node.js 进程。

```
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));
// Node.js 抛出错误，随后崩溃
```

为了防止 Node.js 进程崩溃，可以在进程对象 uncaughtException 事件上注册监听器或使用域（domain）模块（请注意，`domain` 模块已被弃用）。

```
const myEmitter = new MyEmitter();

process.on('uncaughtException', (err) => {
    console.log('哇哦！这儿有个错误');
});

myEmitter.emit('error', new Error('whoops!'));
// 打印：哇哦！这儿有个错误
```

作为最佳实践，应该始终为 `'error'` 事件注册监听器：

```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
    console.log('哇哦！这儿有个错误');
});
myEmitter.emit('error', new Error('whoops!'));
// 打印：哇哦！这儿有个错误
```

绑定一次性事件

当使用 `eventEmitter.on()` 方法注册监听器时，这个监听器会在每次发出该命名事件时被调用。

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.on('event', () => {
    console.log(++m);
});
myEmitter.emit('event');
// 打印：1
myEmitter.emit('event');
// 打印：2
```

当使用 `eventEmitter.once()` 方法时，可以注册对于特定事件最多调用一次的监听器。一旦触发了该事件，监听器就会被注销，随后调用该事件。

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.once('event', () => {
    console.log(++m);
});
myEmitter.emit('event');
// 打印：1
myEmitter.emit('event');
// 忽略
```

给监听器传参

`eventEmitter.emit()` 方法允许将任意参数传递给监听器函数。需要牢记的是，一个普通的监听器函数被 `EventEmitter` 调用时，标准的 `this` 关键词会被刻意得设置成指向附加到监听器上的这个 `EventEmitter` 实例的引用。

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function (a, b) {
    console.log(a, b, this);
    // 打印：
    //   a b MyEmitter {
    //     domain: null,
    //     _events: { event: [Function] },
    //     _eventsCount: 1,
    //     _maxListeners: undefined }
});
myEmitter.emit('event', 'a', 'b');
```

也可以使用 ES6 的箭头函数作为监听器。然而，当你这么做时，`this` 关键词将不再引用 `EventEmitter` 实例。

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
    console.log(a, b, this);
    // 打印：a b {}
});
myEmitter.emit('event', 'a', 'b');
```

异步和同步

`EventListener` 会按照监听器的注册顺序同步地调用所有监听器。这对于确保事件的正确排序很重要以避免竞争条件或逻辑错误。在适当的时候，监听器函数也可以通过使用 `setImmediate()` 或 `process.nextTick()` 方法切换到异步操作模式：

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
    setImmediate(() => {
        console.log('这是异步发生的');
    });
});
myEmitter.emit('event', 'a', 'b');
```

错误(Errors)

在 Node.js 中运行的应用一般会遇到以下四类错误：

- 标准的 JavaScript 错误，例如：
  - EvalError：当调用 `eval()` 失败时抛出。
  - SyntaxError：当响应错误的 JavaScript 语法时抛出。
  - RangeError：当一个值不在预期范围内时抛出。
  - ReferenceError：当使用未定义的变量时抛出。
  - TypeError：当传递错误类型的参数时抛出。
  - URIError：当全局 URI 处理函数被误用时抛出。
- 由于底层操作系统的限制引发的系统错误。例如，试图打开不存在的文件，试图向一个已关闭的套接字发送数据等；
- 以及由应用程序代码触发的用户指定（User-specified）的错误。
- 断言错误是一种特殊的错误类型，只要 Node.js 检测到不应该发生的异常逻辑违例，就可以触发错误。这些通常由 `assert` 模块引发。

由 Node.js 提出的所有 JavaScript 和系统错误都继承自或是标准的 JavaScript 错误类的实例，并保证至少提供该类可用的属性。

Error类

一个普通的 JavaScript `Error` 对象不会表述错误发生的具体原因。`Error` 对象会捕捉一个详细说明被实例化的 `Error` 对象在代码中的位置的“堆栈跟踪”，并可能提供对此错误的文字描述。

所有由 Node.js 产生的系统和 JavaScript 错误都继承实例化或继承自 `Error` 类。

new Error(message)

新建一个 `Error` 实例并设置它的 `error.message` 属性以提供文本信息。如果 `message` 传的是一个对象，则会通过 `message.toString()` 生成文本信息。`error.stack` 属性会表明 `new Error()` 在代码中的位置。堆栈跟踪是根据 V8 的堆栈跟踪 API 生成的。堆栈跟踪只会取（a）异步代码开始执行前或（b）`Error.stackTraceLimit` 属性给出的栈帧中的最小项。

Error.captureStackTrace(targetObject[, constructorOpt])

当访问调用 `Error.captureStackTrace()` 方法所返回的表示代码中的位置的字符串时，会在 `targetObject` 上创建一个 `.stack` 属性。

```
const myObject = {};
Error.captureStackTrace(myObject);
myObject.stack  // similar to `new Error().stack`
```

堆栈跟踪的第一行会是前缀被替换成 `ErrorType: message` 的 `targetObject.toString()` 的调用结果。

可选的 `constructorOpt` 接受一个函数作为其参数。如果提供了该参数，则 `constructorOpt` 以前包括自身在内的全部栈帧都会被其生成的堆栈跟踪省略。

`constructorOpt` 用在向最终用户隐藏错误生成的具体细节时非常有用。例如：

```
function MyError() {
    Error.captureStackTrace(this, MyError);
}

// Without passing MyError to captureStackTrace, the MyError
// frame would should up in the .stack property. by passing
// the constructor, we omit that frame and all frames above it.
new MyError().stack
```

Error.stackTraceLimit

`Error.stackTraceLimit` 属性用于指定堆栈跟踪所收集的栈帧数量（无论是由 `new Error().stack` 还是由 `Error.captureStackTrace(obj)` 产生的）。

默认值为 `10` ，但可以设置成任何有效 JavaScript 数值。这个修改会影响到值被改变后捕捉到的所有堆栈跟踪。

如果设置成一个非数值或负数，堆栈跟踪将不会捕捉任何栈帧。

error.message

返回由 `new Error(message)` 设置的用来描述错误的字符串。这个传递给构造函数的 `message` 也将出现在 `Error` 的堆栈跟踪的第一行。然而，在 `Error` 对象创建后改变这个属性可能不会改变堆栈跟踪的第一行。

```
const err = new Error('The message');
console.log(err.message);
// Prints: The message
```

error.stack

返回一个描述 `Error` 实例在代码中的位置的字符串。

例如：

```
Error: Things keep happening!
   at /home/gbusey/file.js:525:2
   at Frobnicator.refrobulate (/home/gbusey/business-logic.js:424:21)
   at Actor.<anonymous> (/home/gbusey/actors.js:400:8)
   at increaseSynergy (/home/gbusey/actors.js:701:6)
```

第一行会被格式化为 `<error class name>: <error message>` ，并且随后跟着一系列栈帧（每一行都会以 "at " 开头）。每一帧都描述了一个代码中导致错误生成的调用点。V8 引擎会尝试显示每个函数的名称（变量名、函数名或对象的方法名），但偶尔也可能找不到一个合适的名称。如果 V8 引擎没法确定一个函数的名称，在该栈帧中只会显示仅有的位置信息。否则，在位置信息的附近将会显示已确定的函数名。

注意，这对仅由 JavaScript 函数产生的栈帧非常有用。例如，在自身是一个 JavaScript 函数的情况下，通过同步执行一个名为 `cheetahify` 的 C++ 插件时，代表 `cheetahify` 回调的栈帧将不会出现在当前的堆栈跟踪里：

```
const cheetahify = require('./native-binding.node');

function makeFaster() {
    // cheetahify *synchronously* calls speedy.
    cheetahify(function speedy() {
        throw new Error('oh no!');
    });
}

makeFaster(); // will throw:
// /home/gbusey/file.js:6
//     throw new Error('oh no!');
//           ^
// Error: oh no!
//     at speedy (/home/gbusey/file.js:6:11)
//     at makeFaster (/home/gbusey/file.js:5:3)
//     at Object.<anonymous> (/home/gbusey/file.js:10:1)
//     at Module._compile (module.js:456:26)
//     at Object.Module._extensions..js (module.js:474:10)
//     at Module.load (module.js:356:32)
//     at Function.Module._load (module.js:312:12)
//     at Function.Module.runMain (module.js:497:10)
//     at startup (node.js:119:16)
//     at node.js:906:3
```

其中的位置信息会以下形式出现：

- `native` ，如果栈帧产生自 V8 引擎内部（比如， `[].forEach` ）。
- `plain-filename.js:line:column` ，如果栈帧产生自 Node.js 内部。
- `/absolute/path/to/file.js:line:column` ，如果栈帧产生自用户程序或其依赖。

代表堆栈跟踪的字符串是在访问 `error.stack` 属性时才被生成的。

堆栈跟踪捕获的栈帧的数量是由 `Error.stackTraceLimit` 或当前事件循环可用的栈帧数量中的最小值界定。

系统级的错误由已传参的 `Error` 实例产生，详见此处。

SyntaxError类

是 `Error` 的一个子类用于表示当前程序不是有效的 JavaScript 代码。这些错误只会产生和传播代码的评测结果。代码评测可能产生自 `eval` 、 `Function` 、 `require` 或 vm。这些错误几乎都表示这是一个坏掉的程序。

```
try {
    require('vm').runInThisContext('binary ! isNotOk');
} catch (err) {
    // err will be a SyntaxError
}
```

`SyntaxError` 实例在创建它们的上下文中是不可恢复的 - 它们只可能被其他上下文捕获。

ReferenceError类

是 `Error` 的一个子类用以表示企图访问一个未定义的变量。这些错误通常表示代码中的错别字或一个坏掉的程序。虽然客户端代码可能会产生和传播这些错误，但在实践中，只有 V8 引擎会这么做。

```
doesNotExist;
// throws ReferenceError, doesNotExist is not a variable in this program.
```

`ReferenceError` 实例会有一个 `error.arguments` 属性，其值为一个只有单个元素（一个代表变量未定义的字符串）的数组。

```
const assert = require('assert');
try {
    doesNotExist;
} catch (err) {
    assert(err.arguments[0], 'doesNotExist');
}
```

除非一个应用程序是动态生成并运行的代码，否则 `ReferenceError` 实例会始终被视为在代码或其依赖中的错误（bug）。

RangeError类

是 `Error` 的一个子类用以表示一个给定参数没有被内部设置或函数值不在可接受范围内；无论这是一个数字范围还是给定函数参数的选项之外的设置（原文，whether that is a numeric range, or outside the set of options for a given function parameter。理解了半天也没搞明白这句话到底想说明什么...译者注）。

例如：

```
require('net').connect(-1);
// throws RangeError, port should be > 0 && < 65536
```

Node.js 会生成并以一种参数验证的形式立即抛出 `RangeError` 实例。

TypeError类

是 `Error` 的一个子类用以表明所提供的参数不是一个被允许的类型。例如，将一个函数传递给一个需要字符串的参数时会产生一个 `TypeError` 。

```
require('url').parse(function () {});
// throws TypeError, since it expected a string
```

Node.js 会生成并以一种参数验证的形式立即抛出 `TypeError` 实例。

错误的冒泡和捕捉

- Node.js 风格的回调

Node.js 支持几种在程序运行时发生的错误冒泡和错误处理机制。如何报告和处理这些错误完全取决于 `Error` 的类型和被调用的 API 的风格。

所有的 JavaScript 错误都是作为异常处理的，立即产生并通过标准的 JavaScript `throw` 机制抛出错误。这些都是利用 JavaScript 语言提供的 try / catch construct 处理的。

```
// Throws with a ReferenceError because z is undefined
try {
    const m = 1;
    const n = m + z;
} catch (err) {
    // Handle the error here.
}
```

JavaScript 的 `throw` 机制在任何时候使用都会引发异常，必须通过 `try / catch` 处理，否则 Node.js 会立即退出进程。

但也有少数例外，同步的 API（任何不接受 `callback` 函数的阻塞方法，例如，fs.readFileSync）会使用 `throw` 报告错误。

异步的 API 中发生的错误可能会以多种方式进行报告：

- 大多数的异步方法都接受一个 `callback` 函数，该函数接受给其第一个参数传递一个 `Error` 对象。如果第一个参数不是 `null` 或是一个 `Error` 实例，就应该对发生的错误进行处理。

```
const fs = require('fs');
fs.readFile('a file that does not exist', (err, data) => {
    if (err) {
        console.error('There was an error reading the file!', err);
        return;
    }
    // Otherwise handle the data
});
```

- 当 `EventEmitter` 对象调用一个异步方法时，错误会被分发到该对象的 `'error'` 事件上。

```
const net = require('net');
const connection = net.connect('localhost');

// Adding an 'error' event handler to a stream:
connection.on('error', (err) => {
    // If the connection is reset by the server, or if it can't
    // connect at all, or on any sort of error encountered by
    // the connection, the error will be sent here.
    console.error(err);
});

connection.pipe(process.stdout);
```

- 在 Node.js API 中有一小部分普通的异步方法仍可能使用 `throw` 机制引发错误，必须使用 `try / catch` 处理。这些方法并没有一个完整的列表。请参阅各类方法的文档以确定所需的合适的错误处理机制。

大多数的 stream-based 和 event emitter-based API 都使用相同的 `'error'` 事件机制，它们本身就代表了一系列随着时间推移的异步操作（相对于单一操作，可能有效也可能无效）。

对于所有的 `EventEmitter` 对象而言，如果不能提供一个 `'error'` 事件处理程序，那么错误将被抛出，从而导致 Node.js 进程报告一个未处理的异常并随即崩溃，除非适当的使用 域 模块或已经注册了 process.on('uncaughtException') 事件。

```
const EventEmitter = require('events');
const ee = new EventEmitter();

setImmediate(() => {
    // This will crash the process because no 'error' event
    // handler has been added.
    ee.emit('error', new Error('This will crash'));
});
```

在调用的代码退出后抛出的错误，不能通过 `try / catch` 截获。

开发者必须查阅各类方法的文档以明确在错误发生时这些方法是如何冒泡的。

Node.js 风格的回调

大多数由 Node.js 核心 API 暴露出来的异步方法都遵循称之为“Node.js 风格的回调”的惯用模式。通过这种模式，可以用作为参数的方法传递函数。当操作完成或引发错误时，`Error` 对象（无论如何）都会作为第一个参数被回调函数所调用。如果没有引发错误，第一个参数会作为 `null` 传递。

```
const fs = require('fs');

function nodeStyleCallback(err, data) {
    if (err) {
        console.error('There was an error', err);
        return;
    }
    console.log(data);
}

fs.readFile('/some/file/that/does-not-exist', nodeStyleCallback);
fs.readFile('/some/file/that/does-exist', nodeStyleCallback)
```

JavaScript 的 `try / catch` 机制不应该用于截取由异步 API 引起的错误。尝试使用 `throw` 替代一个 Node.js 风格的回调是一个初学者常犯的错误：

```
// THIS WILL NOT WORK:
const fs = require('fs');

try {
    fs.readFile('/some/file/that/does-not-exist', (err, data) => {
        // mistaken assumption: throwing here...
        if (err) {
            throw err;
        }
    });
} catch (err) {
    // This will not catch the throw!
    console.log(err);
}
```

这并没有什么用，因为 `fs.readFile()` 是一个异步调用的回调函数。当回调函数被调用时，这些代码（包括 `try { } catch(err) { }` 区域）就已经退出。在大对数案例中，抛出回调函数中的错误会引起 **Node.js** 进程崩溃。如果域被启用，或已注册了 process.on('uncaughtException') 事件，那么这样的错误是可以被拦截的。

异常与错误

JavaScript 异常通常是抛出一个无效操作的结果值或作为 `throw` 所表述的目标。虽然它不要求这些值是 `Error` 或继承自 `Error` 的类的实例，但会通过 Node.js 抛出所有异常或将成为 JavaScript 运行时的 `Error` 实例。

这些异常在 JavaScript 层是无法恢复的。这些异常总会引起 Node.js 进程的崩溃。这些例子包括 `assert()` 检测或在 C++ 层调用的 `abort()` 。

系统错误

当程序在运行时环境中发生异常时会产生系统错误。通常，这些是当应用违反操作系统约束（例如尝试读取不存在的文件或当用户没有足够的权限）时发生的操作错误。

系统错误通常是在系统调用（syscall）级别产生：通过在大多数 Unix 上运行 `man 2 intro` 或 `man 3 errno` 或在线查找，可以获取错误代码及其含义的详尽列表。

在 Node.js 中，系统错误表现为添加额外属性的增强型 `Error` 对象。

系统错误类

error.errno

返回表示否定错误码相应的数字，可能引用自 `man 2 intro`。例如，`ENOENT` 错误的 `errno` 的值为 `-2`，因为 `ENOENT` 的错误码为 `2`。

error.code

返回一个表示错误码的字符串，它总是 `E` 后面跟着一串大写字母，并可能引用自 `man 2 intro`。

error.syscall

返回描述失败的系统调用（syscall）的字符串。

通用的系统错误

这个列表并不详尽，但列举了开发 Node.js 程序时可能遇到的许多常见的系统错误。在这里可以找到一个详尽的列表。

- `EACCES`（没有权限）：试图在一个文件权限不允许的文件夹中访问一个文件。
- `EADDRINUSE`（地址已被使用）：试图将一个服务器（net、http 或 https）绑定本地地址失败，原因是本地系统上的另一个服务已占用了该地址。
- `ECONNREFUSED`（连接被拒绝）：目标机器积极拒绝导致的无法连接。这通常是试图连接到国外主机上不活动的服务后的结果。
- `ECONNRESET`（连接被对方重置）：一个连接被对方强行关闭。这通常是因超时或自动重启导致的远程套接字（socket）丢失的结果。在 http 和 net 模块中经常会碰到。
- `EEXIST`（文件已存在）：一个要求目标不存在的文件操作的目标是一个已存在的文件。
- `EISDIR`（是一个目录）：一个对文件的操作，但给定的路径是一个目录。
- `EMFILE`（在系统中打开了过多的文件）：已达到系统中文件描述符允许的最大数量，并且另外一个描述符的请求在至少关闭其中一个之前不能被满足。这在一次并行打开多个文件时会遇到，尤其是在那些在进程中限制了一个较低的文件描述符数量的操作系统上（在 particular 和 OS X 中）。为了破除这个限制，请在与运行 Node.js 进程的同一 `shell` 中运行 `ulimit -n 2048` 。
- `ENOENT`（无此文件或目录）：通常是由文件操作引起的，这表明指定的路径组合不存在——在给定的路径上无法找到任何实体（文件或目录）。
- `ENOTDIR`（不是一个目录）：给定的路径组合存在，但不是所期望的目录。通常是由 fs.readdir 引起的。
- `ENOTEMPTY`（目录非空）：一个需要空目录操作的目标目录是一个实体。通常是由 fs.unlink 引起的。
- `EPERM`（操作不被允许）：试图执行需要提升权限的操作。
- `EPIPE`（管道损坏）：没有进程读取数据时写入 `pipe` 、 `socket` 或 `FIFO` 。在 net 和 http 层经常遇到，表明在远端的流（stream）准备写入前被关闭。
- `ETIMEDOUT`（操作超时）：由于连接方在一段时间后并没有做出合适的响应导致的连接或发送的请求失败。在 http 或 net 中经常遇到——往往标志着 `socket.end()` 没有被合适的调用。

调试器(Debugger)

> 稳定度：2 - 稳定

Node.js 包含一个可以有效地通过 TCP 协议 访问的完整的进程外的全功能调试工具并内置调试客户端。在启动 Node.js 后，通过 `debug` 参数加上需要调试的脚本文件路径的方式使用，在调试器成功启动后会有明显的提示：

```
$ node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(() => {
  3   debugger;
debug>
```

Node.js 的调试器客户端虽然目前还没法支持全部命令，但可以进行一些简单的（调试）步骤和检测（命令）。

在脚本的源代码中插入一个 `debugger;` 声明就可以在当前位置的代码中设置一个断点。

例如，假设 `myscript.js` 是这么写的：

```
// myscript.js
x = 5;
setTimeout(() => {
    debugger;
    console.log('world');
}, 1000);
console.log('hello');
```

一旦运行调试器，将在第4行发生断点：

```
// myscript.js
x = 5;
setTimeout(() => {
  debugger;
  console.log('world');
}, 1000);
console.log('hello');
Once the debugger is run, a breakpoint will occur at line 4:

$ node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(() => {
  3   debugger;
debug> cont
< hello
break in /home/indutny/Code/git/indutny/myscript.js:3
  1 x = 5;
  2 setTimeout(() => {
  3   debugger;
  4   console.log('world');
  5 }, 1000);
debug> next
break in /home/indutny/Code/git/indutny/myscript.js:4
  2 setTimeout(() => {
  3   debugger;
  4   console.log('world');
  5 }, 1000);
  6 console.log('hello');
debug> repl
Press Ctrl + C to leave debug repl
> x
5
> 2+2
4
debug> next
< world
break in /home/indutny/Code/git/indutny/myscript.js:5
  3   debugger;
  4   console.log('world');
  5 }, 1000);
  6 console.log('hello');
  7
debug> quit
```

`repl` 命令允许代码被远程评估。`next` 命令用于跳转到下一行。键入 `help` 可以查看其他的有效命令。

监视器

可以在调试时查看表达式和变量值。每个断点都会在当前上下文中评估观察列表中的每个表达式，并在列举断点的源代码前立即被显示。

通过键入 `watch('my_expression')` 来监视一个表达式， `watchers` 命令会将其打印在激活的监视器中。通过键入 `unwatch('my_expression')` 来移除一个监视器。

命令参考

步进

- `cont`，`c` - 继续执行
- `next`，`n` - 下一步
- `step`，`s` - 介入
- `out`，`o` - 退出介入
- `pause` - 暂停执行代码（类似开发者工具中的暂停按钮）

断点

- `setBreakpoint()`，`sb()` - 在当前行设置断点
- `setBreakpoint(line)`，`sb(line)` - 在指定行设置断点
- `setBreakpoint('fn()')`，`sb(...)` - 在函数体的第一条语句设置断点
- `setBreakpoint('script.js', 1)`，`sb(...)` - 在 script.js 的第一行设置断点
- `clearBreakpoint('script.js', 1)`，`cb(...)` - 清除 script.js 第一行的断点

也可以在一个尚未被加载的文件（模块）中设置断点：

```
$ ./node debug test/fixtures/break-in-module/main.js
< debugger listening on port 5858
connecting to port 5858... ok
break in test/fixtures/break-in-module/main.js:1
  1 var mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();
debug> setBreakpoint('mod.js', 23)
Warning: script 'mod.js' was not loaded yet.
  1 var mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();
debug> c
break in test/fixtures/break-in-module/mod.js:23
 21
 22 exports.hello = () => {
 23   return 'hello from module';
 24 };
 25
debug>
```

信息

- `backtrace`，`bt` - 显示当前执行框架的回溯
- `list(5)` - 显示脚本源代码的 5 行上下文（之前 5 行和之后 5 行）
- `watch(expr)` - 向监视列表添加表达式
- `unwatch(expr)` - 从监视列表移除表达式
- `watchers` - 列出所有监视器和它们的值（每个断点会自动列出）
- `repl` - 在所调试的脚本的上下文中打开调试器的REPL进行评估
- `exec expr` - 在所调试的脚本的上下文中执行一个表达式

执行控制

- `run` - 运行脚本（调试器开始时自动运行）
- `restart` - 重新启动脚本
- `kill` - 终止脚本

杂项

- `scripts` - 列出所有已加载的脚本
- `version` - 显示 V8 引擎的版本号

高级用法

启用和访问调试器的另一种方式是在启动 Node.js 时添加 `--debug` 命令行标志，或向已存在的 Node.js 进程发送 `SIGUSR1` 信号。

一个进程一旦以这种方式进入了调试模式，它就可以被 Node.js 调试器连接使用，通过连接已运行的进程的 `pid` 或访问这个正在监听的调试器的 URI：

- `node debug -p <pid>` - 通过 `pid` 连接进程
- `node debug <URI>` - 通过类似 localhost:5858 的 URI 连接进程

Buffer(Buffer)

> 稳定度：2 - 稳定

随着 ECMAScript 2015（ES6）推出了 `TypedArray` ，JavaScript 语言已经没有机制用于读取或操纵二进制数据流了。 `Buffer` 类被采纳为 Node.js API 的一部分使得在 TCP 流和文件系统操作等的上下文中与八位字节流进行交互成为可能。

目前 `TypedArray` 已经被添加进 ES6 中， `Buffer` 类以一种更加优化和适用于 Node.js 用例的方式实现了 `Uint8Array` API。

`Buffer` 类的实例类似于整数数组，但具有固定大小、分配 V8 外堆的原始内存的特点。 `Buffer` 的大小在创建时被确定，且不能调整大小。

`Buffer` 类在 Node.js 中是一个全局变量，因此不太可能像其他的模块那样永远需要使用 `require('buffer')` 。

```
const buf1 = Buffer.alloc(10);
// Creates a zero-filled Buffer of length 10.

const buf2 = Buffer.alloc(10, 1);
// Creates a Buffer of length 10, filled with 0x01.

const buf3 = Buffer.allocUnsafe(10);
// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using either fill() or write().

const buf4 = Buffer.from([1, 2, 3]);
// Creates a Buffer containing [01, 02, 03].

const buf5 = Buffer.from('test');
// Creates a Buffer containing ASCII bytes [74, 65, 73, 74].

const buf6 = Buffer.from('tést', 'utf8');
// Creates a Buffer containing UTF8 bytes [74, c3, a9, 73, 74].
```

方法和属性

---

buffer.INSPECT_MAX_BYTES

- {Number} 默认：50

在调用 `buffer.inspect()` 时返回最大字节数。这个数值可以被用户模块重写。查阅 util.inspect()可以获得更多关于 `buffer.inspect()` 的行为细节。

注意，这个属性只存在于使用 `require('buffer')` 返回的 `buffer` 模块中，在全局的 Buffer 和 Buffer 实例中是不存在的。

Buffer类

类相关的方法和属性

- new Buffer(size)
- new Buffer(array)
- new Buffer(buffer)
- new Buffer(arrayBuffer[, byteOffset[, length]])
- new Buffer(str[, encoding])
- Buffer.byteLength(string[, encoding])
- Buffer.isBuffer(obj)
- Buffer.isEncoding(encoding)
- Buffer.from(array)
- Buffer.from(buffer)
- Buffer.from(arrayBuffer[, byteOffset[, length]])
- Buffer.from(str[, encoding])
- Buffer.alloc(size[, fill[, encoding]])
- Buffer.allocUnsafe(size)
- Buffer.concat(list[, totalLength])
- Buffer.compare(buf1, buf2)

实例相关的方法和属性

- buf.length
- buf[index]
- buf.toString([encoding[, start[, end]]])
- buf.toJSON()
- buf.slice([start[, end]])
- [buf.fill(value[, offset[, end]], encoding])
- buf.indexOf(value[, byteOffset][, encoding])
- buf.includes(value[, byteOffset][, encoding])
- buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])
- buf.compare(otherBuffer)
- buf.equals(otherBuffer)
- buf.entries()
- buf.keys()
- buf.values()
- buf.swap16()

- buf.swap32()
- buf.readIntBE(offset, byteLength[, noAssert])
- buf.readIntLE(offset, byteLength[, noAssert])
- buf.readFloatBE(offset[, noAssert])
- buf.readFloatLE(offset[, noAssert])
- buf.readDoubleBE(offset[, noAssert])
- buf.readDoubleLE(offset[, noAssert])
- buf.readInt8(offset[, noAssert])
- buf.readInt16BE(offset[, noAssert])
- buf.readInt16LE(offset[, noAssert])
- buf.readInt32BE(offset[, noAssert])
- buf.readInt32LE(offset[, noAssert])
- buf.readUIntBE(offset, byteLength[, noAssert])
- buf.readUIntLE(offset, byteLength[, noAssert])
- buf.readUInt8(offset[, noAssert])
- buf.readUInt16BE(offset[, noAssert])
- buf.readUInt16LE(offset[, noAssert])
- buf.readUInt32BE(offset[, noAssert])
- buf.readUInt32LE(offset[, noAssert])
- [buf.write(string[, offset[, length]], encoding])
- buf.writeIntBE(value, offset, byteLength[, noAssert])
- buf.writeIntLE(value, offset, byteLength[, noAssert])
- buf.writeFloatBE(value, offset[, noAssert])
- buf.writeFloatLE(value, offset[, noAssert])
- buf.writeDoubleBE(value, offset[, noAssert])
- buf.writeDoubleLE(value, offset[, noAssert])
- buf.writeInt8(value, offset[, noAssert])
- buf.writeInt16BE(value, offset[, noAssert])
- buf.writeInt16LE(value, offset[, noAssert])
- buf.writeInt32BE(value, offset[, noAssert])
- buf.writeInt32LE(value, offset[, noAssert])
- buf.writeUIntBE(value, offset, byteLength[, noAssert])
- buf.writeUIntLE(value, offset, byteLength[, noAssert])
- buf.writeUInt8(value, offset[, noAssert])
- buf.writeUInt16BE(value, offset[, noAssert])
- buf.writeUInt16LE(value, offset[, noAssert])
- buf.writeUInt32BE(value, offset[, noAssert])
- buf.writeUInt32LE(value, offset[, noAssert])

Buffer 类是一个全局变量类型，用来直接处理2进制数据的。 它能够使用多种方式构建。

new Buffer(size)

- `size` {Number}

分配一个 `size` 字节大小的新 `Buffer` 。`size` 必须小于等于 `require('buffer').kMaxLength`（在64位架构上 `kMaxLength` 的大小是 `(2^31)-1`）的值，否则将抛出一个 RangeError 的错误。如果 `size` 小于 0 将创建一个特定的 0 长度（zero-length）的 Buffer。

不像 `ArrayBuffers` ，以这种方式创建的 Buffer 实例的底层内存是没被初始化过的。新创建的 Buffer 的内容是未知的，并可能包含敏感数据。通过使用 `buf.fill(0)` 将一个 Buffer 初始化为零。

```
const buf = new Buffer(5);
console.log(buf);
// <Buffer 78 e0 82 02 01>
// (octets will be different, every time)
buf.fill(0);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

new Buffer(array)

- `array` {Array}

分配 8 位字节大小的 `array` 的一个新的 `Buffer` 。

```
const buf = new Buffer([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);
// creates a new Buffer containing ASCII bytes
// ['b','u','f','f','e','r']
```

new Buffer(buffer)

- `buffer` {Buffer}

将所传的 `buffer` 数据拷贝到新建的 `Buffer` 实例中。

```
const buf1 = new Buffer('buffer');
const buf2 = new Buffer(buf1);

buf1[0] = 0x61;
console.log(buf1.toString());
// 'auffer'
console.log(buf2.toString());
// 'buffer' (copy is not changed)
```

new Buffer(arrayBuffer[, byteOffset[, length]])

- `arrayBuffer` - 一个 `arrayBuffer` 或 `new ArrayBuffer()` 的 `.buffer` 属性
- `byteOffset` {Number} 默认：0
- `length` {Number} 默认： `arrayBuffer.length - byteOffset`

当传递的是 `TypedArray` 实例的 `.buffer` 引用时，这个新建的 Buffer 将像 `TypedArray` 那样共享相同的内存分配。

选填的 `byteOffset` 和 `length` 参数指定一个将由 Buffer 共享的 `arrayBuffer` 中的内存范围。

```
const arr = new Uint16Array(2);
arr[0] = 5000;
arr[1] = 4000;

const buf = new Buffer(arr.buffer); // shares the memory with arr;

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// changing the TypdArray changes the Buffer also
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

new Buffer(str[, encoding])

- `str` {String} 需要编码的字符串
- `encoding` {String} 默认： `'utf8'`

创建一个新的 Buffer 包含给定的 JavaScript 字符串 `str`。如果提供 `encoding` 参数，将标识字符串的字符编码。

```
const buf1 = new Buffer('this is a tést');
console.log(buf1.toString());
// prints: this is a tést
console.log(buf1.toString('ascii'));
// prints: this is a tC)st

const buf2 = new Buffer('7468697320697320612074c3a97374', 'hex');
console.log(buf2.toString());
// prints: this is a tést
```

Buffer.byteLength(string[, encoding])

- `string` {String} | {Buffer} | {TypedArray} | {DataView} | {ArrayBuffer}
- `encoding` {String} 默认：`'utf8'`
- 返回：{Number}

返回一个字符串的实际字节长度。这与 String.prototype.length 不同，因为它是返回字符串中的字符数目。

例如：

```
const str = '\u00bd + \u00bc = \u00be';

console.log(`${str}: ${str.length} characters, ` +
            `${Buffer.byteLength(str, 'utf8')} bytes`);

// ½ + ¼ = ¾: 9 characters, 12 bytes
```

当 `string` 是一个 `Buffer` / DataView / TypedArray / `ArrayBuffer` 时，返回实际的字节长度。

除此之外，将转换为 `String` 并返回字符串的字节长度。

Buffer.isBuffer(obj)

- `obj` {Object}
- 返回：{Boolean}

如果 `obj` 是一个 Buffer 则返回 `true`。

Buffer.isEncoding(encoding)

- `encoding` {String} 需要测试的编码字符串
- 返回：{Boolean}

如果 `encoding` 是一个有效的编码参数则返回 `true`，否则返回 `false`。

Buffer.from(array)

- `array` {Array}

使用一个8位字节的数组分配一个新的 Buffer。

```
const buf = Buffer.from([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);
// creates a new Buffer containing ASCII bytes
// ['b','u','f','f','e','r']
```

如果 `array` 不是一个有效的 `Array` 则抛出一个 TypeError 错误。

Buffer.from(buffer)

- `buffer` {Buffer}

将所传的 `buffer` 数据拷贝到这个新的 `Buffer` 实例中。

```
const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;
console.log(buf1.toString());
// 'auffer'
console.log(buf2.toString());
// 'buffer' (copy is not changed)
```

如果 `buffer` 不是一个有效的 `Buffer` 则抛出一个 TypeError 错误。

Buffer.from(arrayBuffer[, byteOffset[, length]])

- `arrayBuffer` - 一个 `TypedArray` 或 `new ArrayBuffer()` 的 `.buffer` 属性
- `byteOffset` {Number} 默认：0

- `length` {Number} 默认：`arrayBuffer.length - byteOffset`

当传递的是 `TypedArray` 实例的 `.buffer` 引用时，这个新建的 Buffer 将像 `TypedArray` 那样共享相同的内存分配。

```
const arr = new Uint16Array(2);
arr[0] = 5000;
arr[1] = 4000;

const buf = Buffer.from(arr.buffer); // shares the memory with arr;

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// changing the TypedArray changes the Buffer also
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

选填的 `byteOffset` 和 `length` 参数指定一个将由 `Buffer` 共享的 `arrayBuffer` 中的内存范围。

```
const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);
console.log(buf.length);
// Prints: 2
```

如果 `arrayBuffer` 不是一个有效的 `ArrayBuffer` 则抛出一个 TypeError 错误。

Buffer.from(str[, encoding])

- `str` {String} 需要编码的字符串
- `encoding` {String} 编码时用到，默认：`'utf8'`

创建一个新的 Buffer 包含给定的 JavaScript 字符串 `str`。如果提供 `encoding` 参数，将标识字符串的字符编码。如果没有提供 `encoding` 参数，默认为 `'utf8'`。

```
const buf1 = Buffer.from('this is a tést');
console.log(buf1.toString());
// prints: this is a tést
console.log(buf1.toString('ascii'));
// prints: this is a tC)st

const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');
console.log(buf2.toString());
// prints: this is a tést
```

如果 `str` 不是一个有效的 `String` 则抛出一个 TypeError 错误。

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

- `size` {Number}
- `fill` {Value} 默认： `undefined`
- `encoding` {String} 默认： `utf8`

分配一个 `size` 字节大小的新 `Buffer` 。如果 `fill` 是 `undefined` ，该 `Buffer` 将被零填充（zero-filled）。

```
const buf = Buffer.alloc(5);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

`size` 必须小于等于 `require('buffer').kMaxLength` （在64位架构上 `kMaxLength` 的大小是 `(2^31)-1` ）的值，否则将抛出一个 RangeError 的错误。如果 `size` 小于 0 将创建一个特定的 0 长度（zero-length ）的 Buffer。

如果指定了 `fill` 参数，将通过调用 buf.fill(fill) 初始化当前 `Buffer` 的分配。

```
const buf = Buffer.alloc(5, 'a');
console.log(buf);
// <Buffer 61 61 61 61 61>
```

如果同时指定了 `fill` 和 `encoding` 参数，将通过调用 buf.fill(fill, encoding) 初始化当前 `Buffer` 的分配。例如：

```
const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');
console.log(buf);
// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

调用 `Buffer.alloc(size)` 方法显然要比替代的 `Buffer.allocUnsafe(size)` 要慢，但可以确保新建的 Buffer 实例的内容不会包含敏感数据。

如果 `size` 不是一个数字则抛出一个 TypeError 错误。

Buffer.allocUnsafe(size)

- `size` {Number}

分配一个 `size` 字节大小的新的非零填充（*non-zero-filled*）的 `Buffer` 。`size` 必须小于等于 `require('buffer').kMaxLength` （在64位架构上 `kMaxLength` 的大小是 `(2^31)-1` ）的值，否则将抛出一个 RangeError 的错误。如果 `size` 小于 0 将创建一个特定的 0 长度（zero-length ）的 Buffer。

以这种方式创建的 Buffer 实例的底层内存是没被初始化过的。新创建的 Buffer 的内容是未知的，并可能包含敏感数据。通过使用 `buf.fill(0)` 将这个 `Buffer` 初始化为零。

```
const buf = Buffer.allocUnsafe(5);
console.log(buf);
// <Buffer 78 e0 82 02 01>
// (octets will be different, every time)
buf.fill(0);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

如果 `size` 不是一个数字则抛出一个 TypeError 错误。

请注意，`Buffer` 模块预分配一个大小为 `Buffer.poolSize` 的内部 `Buffer` 实例作为一个快速分配池，仅当 `size` 小于等于 `Buffer.poolSize >> 1` （浮点类型的 `Buffer.poolSize` 应该除以2）时，用于分配通过 `Buffer.allocUnsafe(size)` 创建的新的 `Buffer` 实例（和 `new Buffer(size)` 构造函数）。默认的 `Buffer.poolSize` 值为 `8192` ，但可以被修改。

使用这个预分配的内部内存池是调用 `Buffer.alloc(size, fill)` 和 `Buffer.allocUnsafe(size).fill(fill)` 的关键不同之处。特别是，如果 `size` 小于或等于 `Buffer.poolSize` 的一半时，`Buffer.allocUnsafe(size).fill(fill)` 将使用内部的 Buffer 池，而 `Buffer.alloc(size, fill)` 将绝不会使用这个内部的 Buffer 池。在一个应用程序需要 `Buffer.allocUnsafe(size)` 提供额外的性能时，（了解）这个细微的不同之处是非常重要的。

Buffer.concat(list[, totalLength])

- `list` {Array} 需要连接的 Buffer 对象数组

- `totalLength` {Number} 上述需要被连接的 Buffer 的总大小。
- 返回：{Buffer}

返回一个连接了 `list` 中所有 Buffer 的新 Buffer 。

如果 `list` 中没有项目，或者当 `totalLength` 为 0 时，将返回一个 0 长度（zero-length）的 Buffer 。

如果没有提供 `totalLength` ，它将计算 `list` 中的 Buffer（以获得该值）。然而，这增加了额外的函数循环，提供精准的长度将加速计算。

例如：将一个包含三个 Buffer 的数组构建为一个单一的 Buffer ：

```
const buf1 = Buffer.alloc(10, 0);
const buf2 = Buffer.alloc(14, 0);
const buf3 = Buffer.alloc(18, 0);
const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength);
const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);
console.log(bufA);
console.log(bufA.length);

// 42
// <Buffer 00 00 00 00 ... >
// 42
```

Buffer.compare(buf1, buf2)

- `buf1` {Buffer}
- `buf2` {Buffer}
- 返回：{Number}

比较 `buf1` 和 `buf2` 通常用于 Buffer 数组的排序目的。这相当于是调用 buf1.compare(buf2) 。

```
const arr = [Buffer.from('1234'), Buffer.from('0123')];
arr.sort(Buffer.compare);
```

实例的属性和方法

buf.length

- {Number}

返回该 Buffer 在字节数上分配的内存量。注意，这并不一定反映 Buffer 内可用的数据量。例如在下面的例子中，一个 Buffer 分配了 1234 字节，但只写入了 11 ASCII 字节。

```
const buf = Buffer.allocUnsafe(1234);

console.log(buf.length);
// Prints: 1234

buf.write('some string', 0, 'ascii');
console.log(buf.length);
// Prints: 1234
```

而 `length` 属性并非一成不变，改变 `length` 值会导致不确定和不一致的行为。应用程序如果希望修改一个 Buffer 的长度，因而需要把 `length` 设为只读并使用 buf.slice() 创建一个新的 Buffer 。

```
var buf = Buffer.allocUnsafe(10);
buf.write('abcdefghj', 0, 'ascii');
console.log(buf.length);
// Prints: 10
buf = buf.slice(0, 5);
console.log(buf.length);
// Prints: 5
```

buf[index]

索引操作符 `[index]` 可用于获取或设置 Buffer 中指定 `index` 位置的8位字节。这个值指的是单个字节，所以这个值合法范围是16进制的 0x00 到 0xFF，或 10进制的 0 到 255。

例如，将一个 ASCII 字符串拷贝到一个 Buffer 中，一次一个字节：

```
const str = "Node.js";
const buf = Buffer.allocUnsafe(str.length);

for (var i = 0; i < str.length; i++) {
    buf[i] = str.charCodeAt(i);
}

console.log(buf.toString('ascii'));
// Prints: Node.js
```

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

- `encoding` {String} 默认：`'utf8'`
- `start` {Number} 默认：0
- `end` {Number} 默认：`buffer.length`
- 返回：{String}

返回使用指定的字符集编码解码 Buffer 数据的字符串。

```
const buf = Buffer.allocUnsafe(26);
for (var i = 0; i < 26; i++) {
    buf[i] = i + 97; // 97 is ASCII a
}
buf.toString('ascii');
// Returns: 'abcdefghijklmnopqrstuvwxyz'
buf.toString('ascii', 0, 5);
// Returns: 'abcde'
buf.toString('utf8', 0, 5);
// Returns: 'abcde'
buf.toString(undefined, 0, 5);
// Returns: 'abcde', encoding defaults to 'utf8'
```

buf.toJSON()

- 返回：{Object}

返回该 Buffer 实例的 JSON 表达式。当字符串化一个 Buffer 实例时会隐式调用 JSON.stringify() 这个函数。

例子：

```
const buf = Buffer.from('test');
const json = JSON.stringify(buf);

console.log(json);
// Prints: '{"type":"Buffer","data":[116,101,115,116]}'

const copy = JSON.parse(json, (key, value) => {
    return value && value.type === 'Buffer' ? Buffer.from(value.data) : value;
});

console.log(copy.toString());
// Prints: 'test'
```

buf.slice([start[, end]])

- `start` {Number} 默认：0
- `end` {Number} 默认：`buffer.length`
- 返回：{Buffer}

返回一个指向相同原始内存的新 Buffer ，但会有偏移并通过 `start` 和 `end` 索引值进行裁剪。

请注意，修改这个新的 **Buffer** 切片，将会修改原始 **Buffer** 的内存，因为这两个对象共享所分配的内存。

例子：创建一个 ASCII 字母的 Buffer，进行切片，然后修改原始 Buffer 上的一个字节。

```
const buf1 = Buffer.allocUnsafe(26);

for (var i = 0; i < 26; i++) {
    buf1[i] = i + 97; // 97 is ASCII a
}

const buf2 = buf1.slice(0, 3);
buf2.toString('ascii', 0, buf2.length);
// Returns: 'abc'
buf1[0] = 33;
buf2.toString('ascii', 0, buf2.length);
// Returns : '!bc'
```

指定负索引会导致产生相对于这个 Buffer 的末尾而不是开头的切片（slice）。

```
const buf = Buffer.from('buffer');

buf.slice(-6, -1).toString();
// Returns 'buffe', equivalent to buf.slice(0, 5)
buf.slice(-6, -2).toString();
// Returns 'buff', equivalent to buf.slice(0, 4)
buf.slice(-5, -2).toString();
// Returns 'uff', equivalent to buf.slice(1, 4)
```

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

- `value` {String} | {Buffer} | {Number}
- `offset` {Number} 默认：0
- `end` {Number} 默认：`buf.length`
- `encoding` {String} 默认：`'utf8'`

- 返回：{Buffer}

使用指定的值填充当前 Buffer 。如果 `offset` (默认是 `0`) 和 `end` (默认是 `buffer.length`) 没有明确给出，将会填充整个 buffer 。该方法返回一个当前 Buffer 的引用，以便于链式调用。这也意味着可以通过这种小而简的方式创建一个 Buffer 。允许在单行内创建和填充 Buffer ：

```
const b = Buffer.alloc(50, 'h');
console.log(b.toString());
// Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
```

`encoding` 只在 `value` 是一个字符串时应用，除此之外，都会被忽略。如果 `value` 不是一个 `String` 或 `Number` ，则会被强制转换到 `uint32` 类型。

`fill()` 操作默默地向 Buffer 里写入字节。即便最终写入落在多字节字符之间，它也会将这些字节塞到被写入的 buffer 里。

```
Buffer.alloc(3, '\u0222');
// Prints: <Buffer c8 a2 c8>
```

buf.indexOf(value[, byteOffset][, encoding])

- `value` {String} | {Buffer} | {Number}
- `byteOffset` {Number} 默认：0
- `encoding` {String} 默认：`'utf8'`
- 返回：{Number}

该操作类似于 Array#indexOf() ，它返回 `value` 在 Buffer 中的最开始的索引位置，如果当前 Buffer 不包含这个 `value` 则返回 `-1` 。这个 `value` 的值可以是 `String` 、`Buffer` 或 `Number` 。字符串会默认用 UTF8 解释执行。Buffer 将会使用整个 Buffer（比较部分 Buffer 请使用 buf.slice() 方法）。数字在 0 到 255 的范围内。

```
const buf = Buffer.from('this is a buffer');

buf.indexOf('this');
// returns 0
buf.indexOf('is');
// returns 2
buf.indexOf(Buffer.from('a buffer'));
// returns 8
buf.indexOf(97); // ascii for 'a'
// returns 8
buf.indexOf(Buffer.from('a buffer example'));
// returns -1
buf.indexOf(Buffer.from('a buffer example').slice(0, 8));
// returns 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'ucs2');

utf16Buffer.indexOf('\u03a3', 0, 'ucs2');
// returns 4
utf16Buffer.indexOf('\u03a3', -4, 'ucs2');
// returns 6
```

buf.includes(value[, byteOffset][, encoding])

- `value` {String} | {Buffer} | {Number}
- `byteOffset` {Number} 默认：0
- `encoding` {String} 默认：`'utf8'`
- 返回：{Boolean}

该操作类似于 Array#includes()。这个 `value` 的值可以是 `String` 、`Buffer` 或 `Number` 。字符串会被作为 UTF8 解释执行，除非你覆盖了 `encoding` 参数。Buffer 将会使用整个 Buffer（比较部分 Buffer 请使用 buf.slice() 方法）。数字在 0 到 255 的范围内。

`byteOffset` 表示在搜索 `buf` 时的初始索引值。

```
const buf = Buffer.from('this is a buffer');

buf.includes('this');
// returns true
buf.includes('is');
// returns true
buf.includes(Buffer.from('a buffer'));
// returns true
buf.includes(97); // ascii for 'a'
// returns true
buf.includes(Buffer.from('a buffer example'));
// returns false
buf.includes(Buffer.from('a buffer example').slice(0, 8));
// returns true
buf.includes('this', 4);
// returns false
```

buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])

- `targetBuffer` {Buffer} 需要拷贝的 Buffer
- `targetStart` {Number} 默认：0
- `sourceStart` {Number} 默认：0
- `sourceEnd` {Number} 默认： `buffer.length`
- 返回：{Number} 被拷贝的字节数

将这个 Buffer 的一个区域的数据拷贝到目标 Buffer 的一个区域，即便与目标是共享内存区域资源的。

例子：创建两个 Buffer ，然后把将 `buf1` 第 16 到 19 位的字节拷贝到 `buf2` 中，并从 `buf2` 的第 8 位开始（覆盖）。

```
const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (var i = 0; i < 26; i++) {
    buf1[i] = i + 97; // 97 is ASCII a
}

buf1.copy(buf2, 8, 16, 20);
console.log(buf2.toString('ascii', 0, 25));
// Prints: !!!!!!!!qrst!!!!!!!!!!!!!
```

例子：创建一个单一的 Buffer ，然后将一块区域的数据拷贝到同一个 Buffer 中另一块交叉的区域。

```
const buf = Buffer.allocUnsafe(26);

for (var i = 0; i < 26; i++) {
    buf[i] = i + 97; // 97 is ASCII a
}

buf.copy(buf, 0, 4, 10);
console.log(buf.toString());

// efghijghijklmnopqrstuvwxyz
```

buf.compare(otherBuffer)

- `otherBuffer` {Buffer}
- 返回：{Number}

比较两个 Buffer 实例，无论 `buf` 在排序上靠前、靠后甚至与 `otherBuffer` 相同都会返回一个数字标识。比较是基于在每个 Buffer 的实际字节序列。

- 如果 `otherBuffer` 和 `buf` 相同，则返回 `0`
- 如果 `otherBuffer` 排在 `buf` 前面，则返回 `1`
- 如果 `otherBuffer` 排在 `buf` 后面，则返回 `-1`

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('BCD');
const buf3 = Buffer.from('ABCD');

console.log(buf1.compare(buf1));
// Prints: 0
console.log(buf1.compare(buf2));
// Prints: -1
console.log(buf1.compare(buf3));
// Prints: 1
console.log(buf2.compare(buf1));
// Prints: 1
console.log(buf2.compare(buf3));
// Prints: 1

[buf1, buf2, buf3].sort(Buffer.compare);
// produces sort order [buf1, buf3, buf2]
```

buf.equals(otherBuffer)

- `otherBuffer` {Buffer}
- 返回：{Boolean}

返回一个 boolean 标识，无论 `this` 和 `otherBuffer` 是否具有完全相同的字节。

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));
// Prints: true
console.log(buf1.equals(buf3));
// Prints: false
```

buf.entries()

- 返回：{Iterator}

从当前 Buffer 的内容中，创建并返回一个 `[index, byte]` 形式的迭代器。

```
const buf = Buffer.from('buffer');
for (var pair of buf.entries()) {
    console.log(pair);
}
// prints:
//   [0, 98]
//   [1, 117]
//   [2, 102]
//   [3, 102]
//   [4, 101]
//   [5, 114]
```

buf.keys()

- 返回：{Iterator}

创建并返回一个包含 Buffer 键名（索引）的迭代器。

```
const buf = Buffer.from('buffer');
for (var key of buf.keys()) {
    console.log(key);
}
// prints:
//   0
//   1
//   2
//   3
//   4
//   5
```

buf.values()

- 返回：{Iterator}

创建并返回一个包含 Buffer 值（字节）的迭代器。当 Buffer 使用 `for..of` 声明时将自动调用该函数。

```
const buf = Buffer.from('buffer');
for (var value of buf.values()) {
    console.log(value);
}
// prints:
//   98
//   117
//   102
//   102
//   101
//   114

for (var value of buf) {
    console.log(value);
}
// prints:
//   98
//   117
//   102
//   102
//   101
//   114
```

buf.swap16()

- 返回：{Buffer}

将 Buffer 解释执行为一个16位的无符号整数数组并以字节顺序交换到位。如果 Buffer 的长度不是16位的倍数，则抛出一个 RangeError 错误。该方法返回一个当前 Buffer 的引用，以便于链式调用。

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);
console.log(buf);
// Prints Buffer(0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8)
buf.swap16();
console.log(buf);
// Prints Buffer(0x2, 0x1, 0x4, 0x3, 0x6, 0x5, 0x8, 0x7)
```

buf.swap32()

- 返回：{Buffer}

将 Buffer 解释执行为一个32位的无符号整数数组并以字节顺序交换到位。如果 Buffer 的长度不是32位的倍数，则抛出一个 RangeError 错误。该方法返回一个当前 Buffer 的引用，以便于链式调用。

```
const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);
console.log(buf);
// Prints Buffer(0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8)
buf.swap32();
console.log(buf);
// Prints Buffer(0x4, 0x3, 0x2, 0x1, 0x8, 0x7, 0x6, 0x5)
```

buf.readIntBE(offset, byteLength[, noAssert])

buf.readIntLE(offset, byteLength[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - byteLength`
- `byteLength` {Number} `0 < byteLength <= 6`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的 `offset` 位置开始读取 `byteLength` 字节数，并将结果解释执行为一个有符号的2的补码值。支持多达48位精度的值。例如：

```
const buf = Buffer.allocUnsafe(6);
buf.writeUInt16LE(0x90ab, 0);
buf.writeUInt32LE(0x12345678, 2);
buf.readIntLE(0, 6).toString(16); // Specify 6 bytes (48 bits)
// Returns: '1234567890ab'

buf.readIntBE(0, 6).toString(16);
// Returns: -546f87a9cbee
```

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

buf.readFloatBE(offset[, noAssert])

buf.readFloatLE(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 4`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的带有特定尾数格式（ `readFloatBE()` 返回一个较大的尾数， `readFloatLE()` 返回一个较小的尾数）的 `offset` 位置开始读取一个32位浮点值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

```
const buf = Buffer.from([1, 2, 3, 4]);

buf.readFloatBE();
// Returns: 2.387939260590663e-38
buf.readFloatLE();
// Returns: 1.539989614439558e-36
buf.readFloatLE(1);
// throws RangeError: Index out of range

buf.readFloatLE(1, true); // Warning: reads passed end of buffer!
// Segmentation fault! don't do this!
```

buf.readDoubleBE(offset[, noAssert])

buf.readDoubleLE(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 8`

- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的带有特定尾数格式（ `readDoubleBE()` 返回一个较大的尾数， `readDoubleLE()` 返回一个较小的尾数）的 `offset` 位置开始读取一个64位双精度值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

```
const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

buf.readDoubleBE();
// Returns: 8.20788039913184e-304
buf.readDoubleLE();
// Returns: 5.447603722011605e-270
buf.readDoubleLE(1);
// throws RangeError: Index out of range

buf.readDoubleLE(1, true); // Warning: reads passed end of buffer!
// Segmentation fault! don't do this!
```

buf.readInt8(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 1`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的 `offset` 位置开始读取一个有符号的8位整数值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

从 Buffer 里读取的整数数值会被解释执行为有符号的2的补码值。

```
const buf = Buffer.from([1, -2, 3, 4]);

buf.readInt8(0);
// returns 1
buf.readInt8(1);
// returns -2
```

buf.readInt16BE(offset[, noAssert])

buf.readInt16LE(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 2`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的带有特定尾数格式（`readInt16BE()` 返回一个较大的尾数，`readInt16LE()` 返回一个较小的尾数）的 `offset` 位置开始读取一个16位整数值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

从 Buffer 里读取的整数数值会被解释执行为有符号的2的补码值。

```
const buf = Buffer.from([1, -2, 3, 4]);

buf.readInt16BE();
// returns 510
buf.readInt16LE(1);
// returns 1022
```

buf.readInt32BE(offset[, noAssert])

buf.readInt32LE(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 4`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的带有特定尾数格式（`readInt32BE()` 返回一个较大的尾数，`readInt32LE()` 返回一个较小的尾数）的 `offset` 位置开始读取一个有符号的32位整数值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

从 Buffer 里读取的整数数值会被解释执行为有符号的2的补码值。

```
const buf = Buffer.from([1, -2, 3, 4]);

buf.readInt32BE();
// returns 33424132
buf.readInt32LE();
// returns 67370497
buf.readInt32LE(1);
// throws RangeError: Index out of range
```

buf.readUIntBE(offset, byteLength[, noAssert])

buf.readUIntLE(offset, byteLength[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - byteLength`
- `byteLength` {Number} `0 < byteLength <= 6`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的 `offset` 位置开始读取 `byteLength` 字节数，并将结果解释执行为一个无符号的整数值。支持多达48位精度的值。例如：

```
const buf = Buffer.allocUnsafe(6);
buf.writeUInt16LE(0x90ab, 0);
buf.writeUInt32LE(0x12345678, 2);
buf.readUIntLE(0, 6).toString(16); // Specify 6 bytes (48 bits)
// Returns: '1234567890ab'

buf.readUIntBE(0, 6).toString(16);
// Returns: ab9078563412
```

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

buf.readUInt8(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 1`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的 `offset` 位置开始读取一个无符号的8位整数值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

```
const buf = Buffer.from([1, -2, 3, 4]);

buf.readUInt8(0);
// returns 1
buf.readUInt8(1);
// returns 254
```

buf.readUInt16BE(offset[, noAssert])

buf.readUInt16LE(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 2`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的带有特定尾数格式（ `readUInt16BE()` 返回一个较大的尾数， `readUInt16LE()` 返回一个较小的尾数）的 `offset` 位置开始读取一个无符号的16位整数值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

例子：

```
const buf = Buffer.from([0x3, 0x4, 0x23, 0x42]);

buf.readUInt16BE(0);
// Returns: 0x0304
buf.readUInt16LE(0);
// Returns: 0x0403
buf.readUInt16BE(1);
// Returns: 0x0423
buf.readUInt16LE(1);
// Returns: 0x2304
buf.readUInt16BE(2);
// Returns: 0x2342
buf.readUInt16LE(2);
// Returns: 0x4223
```

buf.readUInt32BE(offset[, noAssert])

buf.readUInt32LE(offset[, noAssert])

- `offset` {Number} `0 <= offset <= buf.length - 4`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number}

从该 Buffer 指定的带有特定尾数格式（`readUInt32BE()` 返回一个较大的尾数，`readUInt32LE()` 返回一个较小的尾数）的 `offset` 位置开始读取一个无符号的32位整数值。

设置 `noAssert` 为 `true` ，将跳过对 `offset` 的验证。这将允许 `offset` 超出缓冲区的末尾。

例子：

```
const buf = Buffer.from([0x3, 0x4, 0x23, 0x42]);

buf.readUInt32BE(0);
// Returns: 0x03042342
console.log(buf.readUInt32LE(0));
// Returns: 0x42230403
```

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

- `string` {String} 需要被写入到 Buffer 的字节
- `offset` {Number} 默认：0
- `length` {Number} 默认： `buffer.length - offset`
- `encoding` {String} 默认： `'utf8'`
- 返回：{Number} 被写入的字节数

在 Buffer 的 `offset` 位置使用给定的 `encoding` 写入 `string` 。`length` 参数是写入的字节数。如果 Buffer 没有足够的空间以适应整个字符串，只会写入字符串的一部分，然而，它不会只写入已编码的字符部分。

```
const buf = Buffer.allocUnsafe(256);
const len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(`${len} bytes: ${buf.toString('utf8', 0, len)}`);
// Prints: 12 bytes: ½ + ¼ = ¾
```

buf.writeIntBE(value, offset, byteLength[, noAssert])

buf.writeIntLE(value, offset, byteLength[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - byteLength`
- `byteLength` {Number} 默认： `0 < byteLength <= 6`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

通过指定的 `offset` 和 `byteLength` 将 `value` 写入到当前 Buffer 中。支持多达 48 位的精度。例如：

```
const buf1 = Buffer.allocUnsafe(6);
buf1.writeUIntBE(0x1234567890ab, 0, 6);
console.log(buf1);
// Prints: <Buffer 12 34 56 78 90 ab>

const buf2 = Buffer.allocUnsafe(6);
buf2.writeUIntLE(0x1234567890ab, 0, 6);
console.log(buf2);
// Prints: <Buffer ab 90 78 56 34 12>
```

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

当值不是一个整数时，它的行为是不确定的。

buf.writeFloatBE(value, offset[, noAssert])

buf.writeFloatLE(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 4`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

从该 Buffer 指定的带有特定尾数格式（ `writeFloatBE()` 写入一个较大的尾数， `writeFloatLE()` 写入一个较小的尾数）的 `offset` 位置开始写入 `value` 。当值不是一个32位浮点值时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

例子：

```
const buf = Buffer.allocUnsafe(4);
buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer bb fe 4a 4f>
```

buf.writeDoubleBE(value, offset[, noAssert])

buf.writeDoubleLE(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 8`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

从该 Buffer 指定的带有特定尾数格式（`writeDoubleBE()` 写入一个较大的尾数，`writeDoubleLE()` 写入一个较小的尾数）的 `offset` 位置开始写入 `value` 。 `value` 参数应当是一个有效的64位双精度值。当值不是一个64位双精度值时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

例子：

```
const buf = Buffer.allocUnsafe(8);
buf.writeDoubleBE(0xdeadbeefcafebabe, 0);

console.log(buf);
// Prints: <Buffer 43 eb d5 b7 dd f9 5f d7>

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);
// Prints: <Buffer d7 5f f9 dd b7 d5 eb 43>
```

buf.writeInt8(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 1`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

通过指定的 `offset` 将 `value` 写入到当前 Buffer 中。这个 `value` 应当是一个有效的有符号的8位整数。当值不是一个有符号的8位整数时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

这个 `value` 作为一个2的补码的有符号的整数被解释执行和写入。

```
const buf = Buffer.allocUnsafe(2);
buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);
console.log(buf);
// Prints: <Buffer 02 fe>
```

buf.writeInt16BE(value, offset[, noAssert])

buf.writeInt16LE(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 2`
- `noAssert` {Boolean} 默认： `false`

- 返回：{Number} 偏移加上被写入的字节数

从该 Buffer 指定的带有特定尾数格式（`writeInt16BE()` 写入一个较大的尾数，`writeInt16LE()` 写入一个较小的尾数）的 `offset` 位置开始写入 `value` 。 `value` 参数应当是一个有效的有符号的16位整数。当值不是一个有符号的16位整数时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

这个 `value` 作为一个2的补码的有符号的整数被解释执行和写入。

```
const buf = Buffer.allocUnsafe(4);
buf.writeInt16BE(0x0102, 0);
buf.writeInt16LE(0x0304, 2);
console.log(buf);
// Prints: <Buffer 01 02 04 03>
```

buf.writeInt32BE(value, offset[, noAssert])

buf.writeInt32LE(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 4`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

从该 Buffer 指定的带有特定尾数格式（`writeInt32BE()` 写入一个较大的尾数，`writeInt32LE()` 写入一个较小的尾数）的 `offset` 位置开始写入 `value` 。 `value` 参数应当是一个有效的有符号的32位整数。当值不是一个有符号的32位整数时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

这个 `value` 作为一个2的补码的有符号的整数被解释执行和写入。

```
const buf = Buffer.allocUnsafe(8);
buf.writeInt32BE(0x01020304, 0);
buf.writeInt32LE(0x05060708, 4);
console.log(buf);
// Prints: <Buffer 01 02 03 04 08 07 06 05>
```

buf.writeUIntBE(value, offset, byteLength[, noAssert])

buf.writeUIntLE(value, offset, byteLength[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - byteLength`
- `byteLength` {Number} 默认： `0 < byteLength <= 6`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

通过指定的 `offset` 和 `byteLength` 将 `value` 写入到当前 Buffer 中。支持多达 48 位的精度。例如：

```
const buf = Buffer.allocUnsafe(6);
buf.writeUIntBE(0x1234567890ab, 0, 6);
console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

当值不是一个无符号的整数时，它的行为是不确定的。

buf.writeUInt8(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 1`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

通过指定的 `offset` 将 `value` 写入到当前 Buffer 中。这个 `value` 应当是一个有效的无符号的8位整数。当值不是一个无符号的8位整数时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

```
const buf = Buffer.allocUnsafe(4);
buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);
// Prints: <Buffer 03 04 23 42>
```

buf.writeUInt16BE(value, offset[, noAssert])

buf.writeUInt16LE(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 2`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

从该 Buffer 指定的带有特定尾数格式（`writeUInt16BE()` 写入一个较大的尾数，`writeUInt16LE()` 写入一个较小的尾数）的 `offset` 位置开始写入 `value` 。`value` 参数应当是一个有效的无符号的16位整数。当值不是一个无符号的16位整数时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

```
const buf = Buffer.allocUnsafe(4);
buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer ad de ef be>
```

buf.writeUInt32BE(value, offset[, noAssert])

buf.writeUInt32LE(value, offset[, noAssert])

- `value` {Number} 需要被写入到 Buffer 的字节
- `offset` {Number} `0 <= offset <= buf.length - 4`
- `noAssert` {Boolean} 默认： `false`
- 返回：{Number} 偏移加上被写入的字节数

从该 Buffer 指定的带有特定尾数格式（`writeUInt32BE()` 写入一个较大的尾数，`writeUInt32LE()` 写入一个较小的尾数）的 `offset` 位置开始写入 `value` 。`value` 参数应当是一个有效的无符号的32位整数。当值不是一个无符号的32位整数时，它的行为是不确定的。

将 `noAssert` 设为 `true` 将跳过对 `value` 和 `offset` 的验证。这意味着 `value` 可能对于这个特定的函数来说过大，并且 `offset` 可能超出该 Buffer 的末端，导致该值被直接丢弃。除非确定你的内容的正确性否则不应该被使用。

```
const buf = Buffer.allocUnsafe(4);
buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer fe ed fa ce>

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>
```

SlowBuffer类

- new SlowBuffer(size)

返回一个不被池管理的 Buffer 。

为了避免创建大量独立分配的 Buffer 带来的垃圾回收开销，默认情况下小于 4KB 的空间都是切割自一个较大的独立对象。这种策略既提高了性能也改善了内存使用，因为 V8 不需要跟踪和清理很多 `Persistent` 对象。

当开发者需要将池中一小块数据保留不确定的一段时间，较为妥当的办法是用 `SlowBuffer` 创建一个不被池管理的 Buffer 实例并将相应数据拷贝出来。

```
// need to keep around a few small chunks of memory
const store = [];

socket.on('readable', () => {
    var data = socket.read();
    // allocate for retained data
    var sb = SlowBuffer(10);
    // copy the data into the new allocation
    data.copy(sb, 0, 0, 10);
    store.push(sb);
});
```

`SlowBuffer` 应该仅作为开发者已经观察到他们的应用保留了过度的内存时的最后手段。

new SlowBuffer(size)

- `size` {Number}

分配一个 `size` 字节大小的新 `SlowBuffer` 。`size` 必须小于等于 `require('buffer').kMaxLength` （在64位架构上 `kMaxLength` 的大小是 `(2^31)-1` ）的值，否则将抛出一个 RangeError 的错误。如果 `size` 小于 0 将创建一个特定的 0 长度（zero-length ）的 SlowBuffer。

`SlowBuffer` 实例的底层内存是没被初始化过的。新创建的 `SlowBuffer` 的内容是未知的，并可能包含敏感数据。通过使用 `buf.fill(0)` 将一个 `SlowBuffer` 初始化为零。

```
const SlowBuffer = require('buffer').SlowBuffer;
const buf = new SlowBuffer(5);
console.log(buf);
// <Buffer 78 e0 82 02 01>
// (octets will be different, every time)
buf.fill(0);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe()

- 是什么使得 Buffer.allocUnsafe(size) “不安全”？

由于历史原因，`Buffer` 实例通过 `Buffer` 构造函数创建，它基于提供的不同参数分配返回不同的 `Buffer` ：

- 给 `Buffer()` 的第一个参数传参（如，`new Buffer(10)` ），通过指定的大小分配一个新的 Buffer 。给这样的 Buffer 分配的内存是没有初始化过的，并会包含敏感数据。这样的 `Buffer` 对象必须手动通过 buf.fill(0) 初始化或写满这个 `Buffer` 。虽然这种行为是为了提高性能而故意为之的，开发经验已经证明对于是创造一个更慢但很安全的 `Buffer` 还是创建一个快速但未初始化的 `Buffer` 之间需要更加明确的区分。
- 通过给第一个参数传字符串、数组或 Buffer ，可以将所传对象的数据拷贝当前 Buffer 中。
- 传一个 `ArrayBuffer` 返回一个与给定的 `ArrayBuffer` 共享分配的内存的 `Buffer` 。

因为 `new Buffer()` 行为会根据第一个参数所传值的类型不同而显著改变，所以应用程序如果没有适当地验证给 `new Buffer()` 传的输入参数，或未能适当地初始化新分配的 `Buffer` 的内容，会给他们的代码带来安全性和可靠性方面的问题。

为了使创建的 `Buffer` 对象更可靠，更不容易出错，新的 `Buffer.from()` 、 `Buffer.alloc()` 和 `Buffer.allocUnsafe()` 方法作为创建 `Buffer` 实例的替代手段而相继出台。

开发者应当把所有正在使用的 *`new Buffer()`* 构造函数迁移到这些新的*API*之一：

- Buffer.from(array) 返回一个包含所提供的 8位字节的副本的新 `Buffer` 。
- Buffer.from(arrayBuffer[, byteOffset[, length]]) 返回一个与给定的 `ArrayBuffer` 共享分配的内存的 `Buffer` 。
- Buffer.from(buffer) 返回一个包含所提供的 `Buffer` 的副本的新 `Buffer` 。
- Buffer.from(str[, encoding]) 返回一个包含所提供的字符串的副本的新 `Buffer` 。
- Buffer.alloc(size[, fill[, encoding]]) 返回一个指定大小的被填满的 `Buffer` 实例。这种方法会比 Buffer.allocUnsafe(size) 显著地慢，但可确保新创建的 `Buffer` 绝不会包含旧的和潜在的敏感数据。

- Buffer.allocUnsafe(size) 返回一个指定 `size` 的 `Buffer` ，但它的内容必须被 buf.fill(0) 初始化或完全写满。

被 `Buffer.allocUnsafe(size)` 返回的 `Buffer` 实例，如果它的 `size` 小于或等于 `Buffer.poolSize` 的一半，可能被分配进一个共享的内部内存池。

是什么使得 Buffer.allocUnsafe(size) “不安全”？

当调用 `Buffer.allocUnsafe()` 时，被分配的内存段是没被初始化（它不是被零填充的）过的。虽然这样的设计使得内存的分配相当快，但已分配的存储段可能包含潜在的敏感的旧数据。使用通过 `Buffer.allocUnsafe(size)` 创建没有被完全覆写内存的 `Buffer` ，在 `Buffer` 内存是可读的情况下，可能泄露它的旧数据。

虽然在使用 `Buffer.allocUnsafe()` 时有明显的性能优势，但必须额外小心，以避免给应用程序引入安全漏洞。

--zero-fill-buffers命令行选项

Node.js 可以在一开始就使用 `--zero-fill-buffers` 命令行选项强制所有新配置的 `Buffer` 和 `SlowBuffer` 实例，无论是通过 `new Buffer(size)` 还是 `new SlowBuffer(size)` ，都在创建时自动用零填充。使用这个标志改变这些方法的默认行为会对性能有显著的影响。建议只在有绝对必要新分配的 Buffer 实例不能包含潜在的敏感数据时才去执行 `--zero-fill-buffers` 选项。

```
$ node --zero-fill-buffers
> Buffer(5);
<Buffer 00 00 00 00 00>
```

Buffers和字符编码

Buffers 通常用于代表编码的字符序列，比如 UTF8 、 UCS2 、 Base64 甚至 Hex-encoded 的数据。有可能通过使用一个明确的编码方法在 Buffers 和普通的 JavaScript 字符串对象之间进行相互转换。

```
const buf = Buffer.from('hello world', 'ascii');
console.log(buf.toString('hex'));
// prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// prints: aGVsbG8gd29ybGQ=
```

Node.js 目前支持的字符编码包括：

- `'ascii'` - 仅支持 7位 ASCII 数据。如果设置去掉高位的话，这种编码方法是非常快的。
- `'utf8'` - 多字节编码的Unicode字符。许多网页和其他文档格式使用 UTF-8 。
- `'utf16le'` - 2或4个字节，小端编码的Unicode字符。支持代理对（U+10000 to U+10FFFF）。
- `'ucs2'` - `'utf16le'` 的别名。
- `'base64'` - Base64 字符串编码。当从一个字符串创建一个 buffer 时，按照 RFC 4648, Section 5 里的规定，这种编码也将接受正确的“URL和文件名安全字母”。
- `'binary'` - 一种把 buffer 编码成一字节（latin-1）编码字符串的方式。目前不支持 `'latin-1'` 字符串。通过 `'binary'` 来代替 `'latin-1'` 使用 `'latin-1'` 编码。
- `'hex'` - 将每个字节编码为两个十六进制字符。

Buffers和类型数组

Buffers 同样是 `Uint8Array` 类型数组（TypedArray）的实例。但是，和 ECMAScript 2015 中的 `TypedArray` 规范还是有些微妙的不同之处。例如，当 `ArrayBuffer#slice()` 创建了一个切片的副本时，Buffer#slice() 的实现是在现有的 Buffer 上不经过拷贝直接进行创建，这也使得 `Buffer#slice()` 更为有效。

在以下的注意事项下，从一个 Buffer 里创建一个新的类型数组（TypedArray）也是有可能的：

1. 将 Buffer 对象的内存以不共享的方式拷贝到这个类型数组（TypedArray）中。
2. 将 Buffer 对象的内存解释执行为一个不同元素的数组，并且不是目标类型的字节数组。这就像，`new Uint32Array(Buffer.from([1,2,3,4]))` 创建了一个4元素的 `Uint32Array` 的包含元素 `[1,2,3,4]` ，而不是 `Uint32Array` 包含一个单一的元素 `[0x1020304]` 或 `[0x4030201]` 。

也可以通过类型数组对象的 `.buffer` 属性创建一个新的 Buffer ，作为共享同一分配的内存的类型数组实例：

```
const arr = new Uint16Array(2);
arr[0] = 5000;
arr[1] = 4000;

const buf1 = Buffer.from(arr); // copies the buffer
const buf2 = Buffer.from(arr.buffer); // shares the memory with arr;

console.log(buf1);
// Prints: <Buffer 88 a0>, copied buffer has only two elements
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;
console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>
```

请注意，当通过类型数组的 `.buffer` 属性创建一个 Buffer 时，有可能只能通过传入 `byteOffset` 和 `length` 参数使用底层类型数组的一部分。

```
const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);
console.log(buf.length);
// Prints: 16
```

`Buffer.from()` 和 TypedArray.from() （如，`Uint8Array.from()`）有着不同的签名和实现。具体而言，类型数组的变种接受第二参数，在类型数组的每个元素上调用一个映射函数。

但 `Buffer.from()` 不支持使用一个映射函数：

- Buffer.from(array)
- Buffer.from(buffer)
- Buffer.from(arrayBuffer[, byteOffset[, length]])
- Buffer.from(str[, encoding])

Buffers和ES6迭代器

Buffers 可以通过 ECMAScript 2015 (ES6) 的 `for..of` 语法进行遍历。

```
const buf = Buffer(.from[1, 2, 3]);

for (var b of buf)
    console.log(b)

// Prints:
//   1
//   2
//   3
```

另外，`buf.values()` 、`buf.keys()` 和 `buf.entries()` 方法可用于创建迭代器。

流(Stream)

> 稳定度：2 - 稳定

流是一个被 Node.js 中很多对象所实现的抽象接口。比如对一个 HTTP 服务器的请求是一个流，process.stdout 也是一个流。流是可读、可写或兼具两者的。所有流都是 EventEmitter 的实例。

您可以通过 `require('stream')` 加载 `Stream` 基类，这些基类提供了可读（Readable）流、可写（Writable）流、双工（Duplex）流和转换（Transform）流。

本文档分为三个章节：

1. 第一章节介绍了，你在你的程序中使用流时，需要了解的那部分 API 。
2. 第二章节介绍了，当你自己实现一个流时，需要用到的那部分 API ，这些 API 是为了方便你这么做而设计的。
3. 第三章节深入讲解了流的工作方式，包括一些内部机制和函数，除非你明确知道你在做什么，否则尽量不要改动它们。

面向流消费者的API

流可以是可读（Readable）、可写（Writable），或者兼具两者（Duplex，双工）的。

所有流都是 `EventEmitter` 的实例，但它们也有其它自定义的方法和属性，这取决于它们是可读（Readable）、可写（Writable） 或 双工（Duplex）。

如果一个流既可读（Readable）也可写（Writable），那么它就实现了（流的）所有方法和事件。因此，双工（Duplex）或转换（Transform）流充分诠释了这些 API，虽然它们的实现可能有所不同。

没有必要在你的程序里的消费者流中实现流接口。如果你正在自己的程序中实现流接口，请同时参阅面向实现者的API

几乎所有 Node.js 程序，无论多么简单，都以某种方式使用了流。这里有一个使用流的 Node.js 程序的例子：

```
const http = require('http');

var server = http.createServer((req, res) => {
    // req is an http.IncomingMessage, which is a Readable Stream
    // res is an http.ServerResponse, which is a Writable Stream

    var body = '';
    // we want to get the data as utf8 strings
    // If you don't set an encoding, then you'll get Buffer objects
    req.setEncoding('utf8');

    // Readable streams emit 'data' events once a listener is added
    req.on('data', (chunk) => {
        body += chunk;
    });

    // the end event tells you that you have entire body
    req.on('end', () => {
        try {
            var data = JSON.parse(body);
        } catch (er) {
            // uh oh!  bad json!
            res.statusCode = 400;
            return res.end(`error: ${er.message}`);
        }

        // write back something interesting to the user:
        res.write(typeof data);
        res.end();
    });
});

server.listen(1337);

// $ curl localhost:1337 -d '{}'
// object
// $ curl localhost:1337 -d '"foo"'
// string
// $ curl localhost:1337 -d 'not json'
// error: Unexpected token o
```

stream.Readable类

可读（Readable）流接口是你正在读取的一个数据来源的抽象。换言之，数据出自一个可读（Readable）流。

一个可读（Readable）流在你表明你准备接收前将不会开始发出数据。

可读（Readable）流有两种“模式”：流动模式和暂停模式。当处于流动模式时，数据由底层系统读出，并尽可能快地提供给你的程序；当处于暂停模式时，你必须明确地调用 stream.read() 来取出数据块。流开始处于暂停模式。

注意：如果没有绑定数据事件处理器，并且没有 stream.pipe() 目标，同时流被切换到流动模式，那么数据将会流失。

你可以通过下面几种做法切换到流动模式：

- 添加一个 'data' 事件处理器来监听数据。
- 调用 stream.resume() 方法来明确开启数据流。
- 调用 stream.pipe() 方法将数据发送到一个可写（Writable）流。

你可以通过下面的一种做法切换回暂停模式：

- 如果没有导流目标，可以调用 stream.pause() 方法。
- 如果有导流目标，移除所有 'data' 事件 处理器，并通过调用 stream.unpipe() 方法移除所有导流目标。

请注意，为了向后兼容考虑，移除 'data' 事件 处理器并不会自动暂停流。同样的，当有导流目标时，调用 stream.pause() 并不能保证流在那些目标排空并请求更多数据时维持暂停状态。

可读（Readable）流的例子包括：

- 客户端上的 HTTP 响应
- 服务器上的 HTTP 请求
- fs 读取流
- zlib 流
- crypto 流
- TCP 嵌套字
- 子进程的 stdout 和 stderr
- process.stdin

readable事件

当可以从流中读取数据块时，它就会触发一个 `'readable'` 事件。

在某些情况下，假如未准备好，监听一个 `'readable'` 事件会使一些数据从底层系统被读出到内部缓冲区中。

```
var readable = getReadableStreamSomehow();
readable.on('readable', () => {
    // there is some data to read now
});
```

当内部缓冲区被排空后，一旦有更多数据可用时，会再次触发一个 `'readable'` 事件。

`'readable'` 事件不会在“流动”模式中触发的最后唯一一个例外是在流的末尾。

`'readable'` 事件表明流有新的信息：无论新的数据是可用的，还是已经到达流的末尾。在前者的情况下，stream.read() 将会返回那些数据。在后者的情况下，stream.read() 将会返回 `null` 。例如，在下面的例子中， `foo.txt` 是一个空文件：

```
const fs = require('fs');
var rr = fs.createReadStream('foo.txt');
rr.on('readable', () => {
    console.log('readable:', rr.read());
});
rr.on('end', () => {
    console.log('end');
});
```

运行此脚本的输出：

```
$ node test.js
readable: null
end
```

data事件

- `chunk` {Buffer} | {String} 数据块

绑定一个 `'data'` 事件监听器到一个未被明确暂停的流上，流会被切换到流动模式。数据只要可用就会被传递。

如果你想尽可能快地从流中取出所有的数据，这是最佳的方式。

```
var readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
    console.log('got %d bytes of data', chunk.length);
});
```

end事件

这个事件会在没有更多的数据可读时触发。

请注意，`'end'` 事件在数据被完全消费之前不会被触发。它可以在切换到流动模式后或直到你到达末端前通过不停地调用 stream.read() 时被完成。

```
var readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
    console.log('got %d bytes of data', chunk.length);
});
readable.on('end', () => {
    console.log('there will be no more data.');
});
```

close事件

当流和底层数据源（比如，文件描述符）被关闭时触发。并不是所有流都会触发这个事件。这个事件表明没有更多的事件将被触发，并且不会做进一步的计算。

并不是所有的流都会触发 `'close'` 事件。

error事件

- {Error}

在接收数据出错时触发。

readable.read([size])

- `size` {Number} 可选参数，指定要读取多少数据。
- 返回：{String} | {Buffer} | {Null}

`read()` 方法从内部缓冲区中拉取并返回一些数据。当没有数据可用，它会返回 `null`。

如果你传了一个 `size` 参数，那么它就会返回多少字节的数据。如果 `size` 字节不可用，那么它将返回 `null`，除非已经到了数据末端，在这种情况下，它将返回保留在缓冲区中的数据。

如果你没有指定 `size` 参数，那么它就会返回内部缓冲区中的所有数据。

该方法应该仅在暂停模式下被调用。在流动模式下，该方法会被自动调用直到内部缓冲区排空。

```
var readable = getReadableStreamSomehow();
readable.on('readable', () => {
    var chunk;
    while (null !== (chunk = readable.read())) {
        console.log('got %d bytes of data', chunk.length);
    }
});
```

如果该方法返回了一个数据块，那么它也会触发 'data' 事件。

请注意，在 'end' 事件触发后调用 stream.read([size]) 将会返回 `null`，并且不会产生错误警告。

readable.setEncoding(encoding)

- `encoding` {String} 要使用的编码。
- 返回：`this`

调用此函数会使得流返回指定编码的字符串而不是 Buffer 对象。例如，如果你使用 `readable.setEncoding('utf8')`，那么输出数据会被作为 UTF-8 数据解析，并作为字符串返回。如果你 `readable.setEncoding('hex')`，那么数据会被编码成十六进制字符串格式。

该方法能妥善处理多字节字符，如果你直接取出 Buffer 并对它们调用 buf.toString(encoding)，很可能会导致字节错位。如果你想要以字符串形式读取数据，请始终使用该方法。

你还可以使用 `readable.setEncoding(null)` 完全禁用任何编码。如果你在处理二进制数据或将大型的多字节字符串分成多块时，这种做法将非常有用。

```
var readable = getReadableStreamSomehow();
readable.setEncoding('utf8');
readable.on('data', (chunk) => {
    assert.equal(typeof chunk, 'string');
    console.log('got %d characters of string data', chunk.length);
});
```

readable.pipe(destination[, options])

- `destination` {stream.Writable} 写入数据的目标

- `options` {Object} Pipe 选项
  - `end` {Boolean} 当读取结束时终止写入，默认为 `true`。

该方法从可读流中拉取所有数据，并写入到所提供的目标。该方法能自动控制流量以避免目标被快速读取的可读流所淹没。

可以安全地导流到多个目标。

```
var readable = getReadableStreamSomehow();
var writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt'
readable.pipe(writable);
```

该函数返回目标的流，因此你可以建立像这样的导流链：

```
var r = fs.createReadStream('file.txt');
var z = zlib.createGzip();
var w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);
```

例如，模拟 Unix 的 `cat` 命令：

```
process.stdin.pipe(process.stdout);
```

默认情况下，当源数据流触发 'end' 事件时，目标的 stream.end() 会被调用，因此 `destination` 不再可写。传入 `{end: false}` 作为 `options` 可以保持目标流的开启状态。

这让 `writer` 保持开启，因此最后可以写入 "Goodbye"。

```
reader.pipe(writer, {
    end: false
});
reader.on('end', () => {
    writer.end('Goodbye\n');
});
```

请注意 process.stderr 和 process.stdout 在进程结束前都不会被关闭，无论是否指定选项。

readable.unpipe([destination])

- `destination` {stream.Writable} 可选，指定解除导流的流

该方法会移除之前调用 stream.pipe() 所设的钩子。

如果没有指定目标，那么将移除所有的管道。

如果指定了目标，但并没有与之建立导流，那么什么事都不会发生。

```
var readable = getReadableStreamSomehow();
var writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt',
// but only for the first second
readable.pipe(writable);
setTimeout(() => {
    console.log('stop writing to file.txt');
    readable.unpipe(writable);
    console.log('manually close the file stream');
    writable.end();
}, 1000);
```

readable.unshift(chunk)

- `chunk` {Buffer} | {String} 回读队列开头的数据块

该方法在某些情况下很有用，比如一个流正在被一个解析器消费，解析器需要“逆消费”某些刚从源中拉取出来的数据，以便流可以传递给其它消费者。

请注意，`stream.unshift(chunk)` 不能在 'end' 事件 触发后调用，否则将产生一个运行时错误。

如果你发现你必须在你的程序中频繁调用 `stream.unshift(chunk)` ，请考虑实现一个转换（Transform）流作为替代。（详见面向流实现者的 API）

```
// Pull off a header delimited by \n\n
// use unshift() if we get too much
// Call the callback with (error, header, stream)
const StringDecoder = require('string_decoder').StringDecoder;

function parseHeader(stream, callback) {
    stream.on('error', callback);
    stream.on('readable', onReadable);
    var decoder = new StringDecoder('utf8');
    var header = '';

    function onReadable() {
        var chunk;
        while (null !== (chunk = stream.read())) {
            var str = decoder.write(chunk);
            if (str.match(/\n\n/)) {
                // found the header boundary
                var split = str.split(/\n\n/);
                header += split.shift();
                var remaining = split.join('\n\n');
                var buf = new Buffer(remaining, 'utf8');
                if (buf.length)
                    stream.unshift(buf);
                stream.removeListener('error', callback);
                stream.removeListener('readable', onReadable);
                // now the body of the message can be read from the stream.
                callback(null, header, stream);
            } else {
                // still reading the header.
                header += str;
            }
        }
    }
}
```

请注意，不像 stream.push(chunk) 那样，`stream.unshift(chunk)` 不会通过重置流的内部读取状态结束读取过程。如果在读取过程（比如，在一个 stream._read() 内部实现一个自定义流）中调用 `unshift()` 将导致意想不到的结果。在调用 `unshift()` 后立即调用 stream.push('') 会适当地重置读取状态，然而最好简单地避免在执行一个读出过程中调用 `unshift()` 。

readable.pause()

- 返回：`this`

该方法会使一个处于流动模式的流停止触发 'data' 事件，切换到非流动模式，并让后续可用数据留在内部缓冲区中。

```
var readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
    console.log('got %d bytes of data', chunk.length);
    readable.pause();
    console.log('there will be no more data for 1 second');
    setTimeout(() => {
        console.log('now data will start flowing again');
        readable.resume();
    }, 1000);
});
```

readable.isPaused()

- 返回：{Boolean}

这个方法返回是否 `readable` 已通过客户端代码被明确暂停（在不存在相应的 stream.resume() 的情况下使用 stream.pause()）

```
var readable = new stream.Readable

readable.isPaused() // === false
readable.pause()
readable.isPaused() // === true
readable.resume()
readable.isPaused() // === false
```

readable.resume()

- 返回： `this`

该方法让可读流恢复触发 'data' 事件。

该方法会将流切换到流动模式。如果你不想从流中消费数据，但你想得到它的 'end' 事件，你可以调用 stream.resume() 来开启数据流。

```
var readable = getReadableStreamSomehow();
readable.resume();
readable.on('end', () => {
    console.log('got to the end, but did not read anything');
});
```

readable.wrap(stream)

- `stream` {Stream} 一个“旧式”可读流

Node.js v0.10 版本之前的流并未实现现今所有流 API。（更多信息详见"兼容性"章节。）

如果你正在使用一个早期版本的 Node.js 库，它会触发 'data' 事件并且有一个仅作查询用途的 stream.pause() 方法，那么你可以使用 `wrap()` 方法来创建一个使用旧式流作为数据源的可读（Readable）流。

你可能很少需要用到这个函数，但它会作为与旧 Node.js 程序和库交互的简便方法存在。

例如：

```
const OldReader = require('./old-api-module.js').OldReader;
const Readable = require('stream').Readable;
const oreader = new OldReader;
const myReader = new Readable().wrap(oreader);

myReader.on('readable', () => {
    myReader.read(); // etc.
});
```

stream.Writable类

可写（Writable）流接口是一个你正在写入数据的目标的抽象。

可写流的例子包括：

- 客户端上的 HTTP 请求
- 服务器上的 HTTP 响应
- fs 可写流
- zlib 流
- crypto 流
- TCP 嵌套字
- 子进程的 stdin
- process.stdout 和 process.stderr

pipe事件

- `src` {stream.Readable} 被导流到该可写流的来源流

每当 stream.pipe() 方法在一个可读流上被调用并添加当前可写流到所设定的目标时触发。

```
var writer = getWritableStreamSomehow();
var reader = getReadableStreamSomehow();
writer.on('pipe', (src) => {
    console.error('something is piping into the writer');
    assert.equal(src, reader);
});
reader.pipe(writer);
```

unpipe事件

- `src` {stream.Readable} 被解除导流到该可写流的来源流

每当 stream.unpipe() 方法在一个可读流上被调用并移除当前可写流到所设定的目标时触发。

```
var writer = getWritableStreamSomehow();
var reader = getReadableStreamSomehow();
writer.on('unpipe', (src) => {
    console.error('something has stopped piping into the writer');
    assert.equal(src, reader);
});
reader.pipe(writer);
reader.unpipe(writer);
```

drain事件

如果一个 stream.write(chunk) 调用返回 `false` 时，那么 `'drain'` 事件将表明什么时候适合开始向流中写入更多的数据。

```
// Write the data to the supplied writable stream one million times.
// Be attentive to back-pressure.
function writeOneMillionTimes(writer, data, encoding, callback) {
    var i = 1000000;
    write();

    function write() {
        var ok = true;
        do {
            i -= 1;
            if (i === 0) {
                // last time!
                writer.write(data, encoding, callback);
            } else {
                // see if we should continue, or wait
                // don't pass the callback, because we're not done yet.
                ok = writer.write(data, encoding);
            }
        } while (i > 0 && ok);
        if (i > 0) {
            // had to stop early!
            // write some more once it drains
            writer.once('drain', write);
        }
    }
}
```

finish事件

当 stream.end() 方法被调用，并且所有数据已强制写入到底层系统时，此事件会被触发。

```
var writer = getWritableStreamSomehow();
for (var i = 0; i < 100; i++) {
    writer.write('hello, #${i}!\n');
}
writer.end('this is the end\n');
writer.on('finish', () => {
    console.error('all writes are now complete.');
});
```

error事件

- {Error}

当写入或导流数据出现错误时触发。

writable.write(chunk[, encoding][, callback])

- `chunk` {String} | {Buffer} 要写入的数据
- `encoding` {String} 当前编码，如果 `chunk` 是一个字符串
- `callback` {Function} 当数据块被强制写入时回调
- 返回：{Boolean} 如果数据被完全处理时返回 `true` 。

该方法将一些数据写入到底层系统中，并且一旦数据已完全处理，就会调用所提供的回调。如果发生错误，则回调可能会也可能不会将错误作为第一个参数调用。为了检测写入错误，请监听 'error' 事件。

返回值表示你是否应该立即继续写入。如果数据已被滞留在内部，那么它会返回 `false` 。否则，它会返回 `true` 。

这个返回值仅供参考。你可以继续写入，即使它返回 `false` 。然而，写入的数据会被滞留在内存中，所以最好不要过分地这么做。最好的做法是等待 'drain' 事件发生后再继续写入更多数据。

writable.setDefaultEncoding(encoding)

- `encoding` {String} 新的默认编码

设置一个可写流的默认编码。

writable.cork()

强制滞留所有的写入。

滞留的数据会在调用 stream.uncork() 或 stream.end() 时被强制写入。

writable.uncork()

强制写入所有调用 stream.cork() 后滞留的数据。

writable.end([chunk][, encoding][, callback])

- `chunk` {String} | {Buffer} 可选，要写入的数据
- `encoding` {String} 当前编码，如果 `chunk` 是一个字符串
- `callback` {Function} 可选，当流完成时回调

当没有更多数据会被写入到流时调用此方法。如果提供，该回调会作为 'finish' 事件的一个附加的监听器。

在调用 stream.end() 后调用 stream.write() 会引发错误。

```
// write 'hello, ' and then end with 'world!'
var file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// writing more now is not allowed!
```

stream.Duplex类

双工（Duplex）流是同时实现了可读（Readable）和可写（Writable）接口的流。

双工（Duplex）流的例子包括：

- TCP 嵌套字
- zlib 流
- crypto 流

stream.Transform类

转换（Transform）流是一种输出由输入计算所得的双工（Duplex）流。他们同时实现了可读（Readable）和可写（Writable）接口。

转换（Transform）流的例子包括：

- zlib 流
- crypto 流

面向流实现者的API

无论实现任何形式的流，模式都是一样的：

1. 在你的子类中扩展合适的父类。（util.inherits() 方法对此很有帮助。）
2. 在你的构造函数中调用合适的父类的构造函数，以确保内部机制设置正确。
3. 实现一个或多个特定的方法，如下详述。

类扩充和实现方法取决于你要编写哪种流类：

使用情景	类	要实现的方法
只读	Readable	`_read`
只写	Writable	`_write` , `_writev`
读写	Duplex	`_read` , `_write` , `_writev`
操作写入的数据，然后读取结果	Transform	`_transform` , `_flush`

在你的实现代码中，需要强调的是绝对不要调用面向流消费者的 API 中描述的方法。否则，可能会导致在消费你的流接口的过程中产生不良副作用。

stream.Readable类

`stream.Readable` 是一个被设计为拓展底层实现 stream._read(size) 方法的抽象类。

如何在你的程序中消费流，请参阅面向流消费者的 API 。下文解释了如何在你的程序中实现可读流。

new stream.Readable([options])

- `options` {Object}
  - `highWaterMark` {Number} 停止从底层资源读取前能够存储在内部缓冲区的最大字节数。默认：`16384`（16kb）或 `objectMode` 流中的 `16`
  - `encoding` {String} 如果指定，则缓冲区将使用指定的编码字符串解码。默认：`null`
  - `objectMode` {Boolean} 此流是否应该表现为对象流。意味着 stream.read(n) 返回单个值用于替代一个 n 大小的 Buffer 。默认：`false`
  - `read` {Function} 实现 stream._read() 的方法

在拓展自可读（Readable）类的类中，请确保调用 Readable 构造函数，以便缓冲设置可以被正确地初始化。

readable._read(size)

- `size` {Number} 异步读取的字节数

注意：请实现这个方法，但不要直接调用它。

这是一个带有下划线前缀的方法，因为它是在类内部定义的，应该仅被 Readable 类内部的方法的调用。所有的 Readable 流实现都必须提供一个 _read 方法用于从底层资源获取数据。

当调用 `_read()` 时，如果从资源获取的数据可用，`_read()` 的实现应该通过调用 this.push(dataChunk) 开始将数据推入到读取队列中。`_read()` 应该继续从资源处读取并推送数据知道推送返回 `false`，此时应停止从资源处读取。仅当 `_read()` 在停止后被再次调用时，它应该开始从资源处读取更多的数据，并将数据推送到队列中。

注意：一旦调用 `_read()` 方法，直到 stream.push() 方法被调用前将不能被再次调用。

`size` 参数仅做参考。实现中 "read" 是一个单一的回调，返回的数据可以用这个来知道有多少数据获取。实现中不相关的，如 TCP 或 TLS ，可能会忽略此参数，并且简单的提供可用的数据。没有必要在调用 stream.push(chunk) 前，比如 "wait" 直到 `size` 字节的数据可用。

readable.push(chunk[, encoding])

- `chunk` {Buffer} | {Null} | {String} 推入读队列的数据块
- `encoding` {String} 字符串块的编码。必须是一个有效的 Buffer 编码，比如 `'utf8'` 或 `'ascii'` 。
- 返回 {Boolean} 是否应该执行更多的推入

注意：该方法应该在 **Readable** 流的实现者中调用，而不是 **Readable** 流的消费者。

如果传了一个非 `null` 值，`push()` 方法会将数据块放入到流处理器随后消费的队列里。如果传了 `null` ，它标志着已到达流的末尾（EOF），之后没有更多的数据可以被写入。

当触发 'readable' 时间时，用 `push()` 添加的数据可以通过调用 stream.read() 方法拉取。

这个 API 被设计成尽可能地灵活。例如，你可以包裹有某种暂停/恢复机制的低级资源和一个数据回调。在这种情况下，你可以通过这样做来包裹低级资源对象：

```
// source is an object with readStop() and readStart() methods,
// and an `ondata` member that gets called when it has data, and
// an `onend` member that gets called when the data is over.

util.inherits(SourceWrapper, Readable);

function SourceWrapper(options) {
    Readable.call(this, options);

    this._source = getLowlevelSourceObject();

    // Every time there's data, we push it into the internal buffer.
    this._source.ondata = (chunk) => {
        // if push() returns false, then we need to stop reading from source
        if (!this.push(chunk))
            this._source.readStop();
    };

    // When the source ends, we push the EOF-signaling `null` chunk
    this._source.onend = () => {
        this.push(null);
    };
}

// _read will be called when the stream wants to pull more data in
// the advisory size argument is ignored in this case.
SourceWrapper.prototype._read = function (size) {
    this._source.readStart();
};
```

例子：一种计数流

这是一个可读（Readable）流的基本示例。它触发从 1 到 1,000,000 的升序操作，然后结束。

```
const Readable = require('stream').Readable;
const util = require('util');
util.inherits(Counter, Readable);

function Counter(opt) {
    Readable.call(this, opt);
    this._max = 1000000;
    this._index = 1;
}

Counter.prototype._read = function () {
    var i = this._index++;
    if (i > this._max)
        this.push(null);
    else {
        var str = '' + i;
        var buf = new Buffer(str, 'ascii');
        this.push(buf);
    }
};
```

例子：简单的协议 **V1**（次优）

这类似于此处所描述的 `parseHeader` 函数，但作为一个自定义的流来实现。另请注意，这种实现不会将输入的数据转换为字符串。

而然，最好通过转换（Transform）流来实现。参阅简单的协议分析器 V2，一个更好的实现。

```
// A parser for a simple data protocol.
// The "header" is a JSON object, followed by 2 \n characters, and
// then a message body.
//
// NOTE: This can be done more simply as a Transform stream!
// Using Readable directly for this is sub-optimal. See the
// alternative example below under the Transform section.

const Readable = require('stream').Readable;
const util = require('util');

util.inherits(SimpleProtocol, Readable);

function SimpleProtocol(source, options) {
    if (!(this instanceof SimpleProtocol))
        return new SimpleProtocol(source, options);

    Readable.call(this, options);
    this._inBody = false;
    this._sawFirstCr = false;
```

```
    // source is a readable stream, such as a socket or file
    this._source = source;

    source.on('end', () => {
        this.push(null);
    });

    // give it a kick whenever the source is readable
    // read(0) will not consume any bytes
    source.on('readable', () => {
        this.read(0);
    });

    this._rawHeader = [];
    this.header = null;
}

SimpleProtocol.prototype._read = function (n) {
    if (!this._inBody) {
        var chunk = this._source.read();

        // if the source doesn't have data, we don't have data yet.
        if (chunk === null)
            return this.push('');

        // check if the chunk has a \n\n
        var split = -1;
        for (var i = 0; i < chunk.length; i++) {
            if (chunk[i] === 10) { // '\n'
                if (this._sawFirstCr) {
                    split = i;
                    break;
                } else {
                    this._sawFirstCr = true;
                }
            } else {
                this._sawFirstCr = false;
            }
        }

        if (split === -1) {
            // still waiting for the \n\n
            // stash the chunk, and try again.
            this._rawHeader.push(chunk);
            this.push('');
        } else {
            this._inBody = true;
            var h = chunk.slice(0, split);
            this._rawHeader.push(h);
            var header = Buffer.concat(this._rawHeader).toString();
            try {
                this.header = JSON.parse(header);
```

```
            } catch (er) {
                this.emit('error', new Error('invalid simple protocol data'));
                return;
            }
            // now, because we got some extra data, unshift the rest
            // back into the read queue so that our consumer will see it.
            var b = chunk.slice(split);
            this.unshift(b);
            // calling unshift by itself does not reset the reading state
            // of the stream; since we're inside _read, doing an additional
            // push('') will reset the state appropriately.
            this.push('');

            // and let them know that we are done parsing the header.
            this.emit('header', this.header);
        }
    } else {
        // from there on, just provide the data to our consumer.
        // careful not to push(null), since that would indicate EOF.
        var chunk = this._source.read();
        if (chunk) this.push(chunk);
    }
};

// Usage:
// var parser = new SimpleProtocol(source);
// Now parser is a readable stream that will emit 'header'
// with the parsed header data.
```

stream.Writable类

`stream.Writable` 是一个被设计为拓展底层实现 stream._write(chunk, encoding, callback) 方法的抽象类。

如何在你的程序中消费可写流，请参阅面向流消费者的 API 。下文解释了如何在你的程序中实现可写流。

new stream.Writable([options])

- `options` {Object}
  - `highWaterMark` {Number} 当 stream.write() 开始返回 `false` 时的 Buffer 等级。默认： `16384` （16kb）或 `objectMode` 流中的 `16`
  - `decodeStrings` {Boolean} 在传递给 stream._write() 前是否解码字符串到缓冲区。默认： `true`

  - `objectMode` {Boolean} stream.write(anyObj) 是否是一个有效的操作。如果设置，你可以写入任意的数据，而不只是 `Buffer` / `String` 数据。默认： `false`
  - `write` {Function} stream._write() 方法的实现
  - `writev` {Function} stream._writev() 方法的实现

在拓展自可写（Writable）类的类中，请确保调用 Writable 构造函数，以便缓冲设置可以被正确地初始化。

writable._write(chunk, encoding, callback)

- `chunk` {Buffer} | {String} 被写入的（数据）块。总会是一个 buffer ，除非 `decodeStrings` 参数被设置成 `false` 。
- `encoding` {String} 如果该块是一个字符串，那么这是编码类型。如果该块是一个 buffer ，那么这是一个指定的值 - 'buffer'，在这种情况下请忽略它。
- `callback` {Function} 当你处理完成所提供的块时调用此函数（有一个可选的 error 参数）。

所有的 Writable 流实现都必须提供一个 _write 方法用于向底层资源发送数据。

注意：此函数禁止被直接调用。它应该由子类来实现，并且仅被可写（Writable）类的内部方法调用。

该回调函数采用标准的 `callback(error)` 模式来表明写入成功完成还是遇到错误。

如果构造函数选项中设定了 `decodeStrings` 标志，则 `chunk` 可能会是字符串而不是 Buffer，并且 `encoding` 表明了字符串的格式。这种设计是为了支持对某些字符串数据编码提供优化处理的实现。如果您没有明确地将 `decodeStrings` 选项设定为 `false` ，那么您可以安全地忽略 `encoding` 参数，并假定 `chunk` 总是一个 Buffer。

这是一个带有下划线前缀的方法，因为它是在类内部定义的，并且不应该由用户程序直接调用。但是，我们希望你在自己的扩展类中重写此方法。

writable._writev(chunks, callback)

- `chunks` {Array} 被写入的块。每个块都是以下格式： `{ chunk: ..., encoding: ... }` 。
- `callback` {Function} 当你处理完成所提供的块时调用此函数（有一个可选的 error 参数）。

注意：此函数禁止被直接调用。它可能是由子类实现，并且仅被可写（Writable）类的内部方法调用。

此函数是完全可选的实现。在大多数情况下，它是不必要的。如果实现，它将被所有滞留在写入队列中的数据块调用。

stream.Duplex类

“双工”（duplex）流兼具可读和可写特性，比如一个 TCP 嵌套字连接。

值得注意的是，`stream.Duplex` 是一个被设计为拓展底层实现 stream._read(size) 和 stream._write(chunk, encoding, callback) 方法的抽象类，就像你实现可读（Readable）或可写（Writable）类所做的那样。

由于 JavaScript 并不具备多原型继承能力，这个类实际上继承自 Readable，并寄生自 Writable。从而让用户在双工（Duplex）类的拓展中能同时实现低级的 stream._read(n) 和 stream._write(chunk, encoding, callback) 方法。

new stream.Duplex(options)

- `options` {Object} 同时传入可读（Readable）和可写（Writable）构造函数。同时有以下字段：
  - `allowHalfOpen` {Boolean} 默认：`true`。如果设置为 `false`，那么当写入端结束后流将自动结束读取端，反之亦然。
  - `readableObjectMode` {Boolean} 默认：`false`。设置流读取端的 `objectMode`。如果 `objectMode` 为 `true` 也没有影响。
  - `writableObjectMode` {Boolean} 默认：`false`。设置流读取端的 `objectMode`。如果 `objectMode` 为 `true` 也没有影响。

在拓展自双工（Duplex）类的类中，请确保调用其构造函数，以便缓冲设置可以被正确地初始化。

stream.Transform类

“转换”（transform）流实际上是一个输出与输入存在因果关系的双工（duplex）流，比如 zlib 流或 crypto 流。

它并不要求输入和输出需要相同大小、相同块数或同时到达。例如，一个 Hash 流只会在输入结束时产生一个数据块的输出；一个 zlib 流会产生比输入小得多或大得多的输出。

转换（Transform）类必须实现 stream._transform() 方法，而不是 stream._read() 和 stream._write() 方法，同时也可以选择性地实现 stream._flush() 方法。（详见下文。）

new stream.Transform([options])

- `options` {Object}
  - `transform` {Function} 实现 stream._transform() 方法
  - `flush` {Function} 实现 stream._flush() 方法

在拓展自转换（Transform）类的类中，请确保调用其构造函数，以便缓冲设置可以被正确地初始化。

'finish'和'end'事件

'finish' 和 'end' 事件分别来自可写（Writable）父类和可读（Readable）父类。`'finish'` 事件在调用 stream.end() 和所有的块都已被 stream._transform() 处理后触发。`'end'` 在调用回调函数 stream._flush() 输出所有数据后触发。

transform._transform(chunk, encoding, callback)

- `chunk` {Buffer} | {String} 需要被转换的数据块。总会是一个 buffer，除非 `decodeStrings` 选项被设置成 `false`。
- `encoding` {String} 如果该块是一个字符串，那么这是编码类型。如果该块是一个 buffer，那么这是一个指定的值 - 'buffer'，在这种情况下请忽略它。
- `callback` {Function} 当你处理完成所提供的块时调用此函数（有一个可选的 error 参数）

注意：此函数禁止被直接调用。它可能是由子类实现，并且仅被转换（Transform）类的内部方法调用。

所有的转换（Transform）流的实现都必须提供一个 `_transform()` 方法用于接受输入并产生输出。

`_transform()` 应当承担特定的 Transform 类中所有处理被写入的字节、并将它们丢给接口的可读部分的职责，进行异步 I/O，处理其它事情等。

调用 `transform.push(outputChunk)` 零次或多次来从输入块生成输出，这取决于你有多少数据块要作为结果输出。

仅在当前块被完全消费后才调用回调函数。需要注意的是，输出可能会也可能不会作为任何特定的输入块的结果。如果提供了回调函数的第二个参数，它会被传递给 push 方法。换言之，以下是等效的：

```
transform.prototype._transform = function (data, encoding, callback) {
    this.push(data);
    callback();
};

transform.prototype._transform = function (data, encoding, callback) {
    callback(null, data);
};
```

这是一个带有下划线前缀的方法，因为它是在类内部定义的，并且不应该由用户程序直接调用。但是，我们希望你在自己的扩展类中重写此方法。

transform._flush(callback)

- `callback` {Function} 当你强制刷新任何剩余数据时调用此函数（有一个可选的 error 参数）

注意：此函数禁止被直接调用。它可能是由子类实现，如果是这样的话，它仅被转换（Transform）类的内部方法调用。

在一些情景中，你的转换操作可能需要在流的末尾多发一些数据。例如，一个 `Zlib` 压缩流会储存一些内部状态以便更好地压缩输出，但在最后它需要尽可能好地处理剩下的东西以使数据完整。

在这些情况下，你可以实现一个 `_flush()` 方法，它会在所有写入数据被消费后，并且在标志着可读端到达末尾的 'end' 事件触发前的最后一刻被调用。和 stream._transform() 一样，只需在 flush 操作完成时适当地调用 `transform.push(chunk)` 零或多次。

这是一个带有下划线前缀的方法，因为它是在类内部定义的，并且不应该由用户程序直接调用。但是，我们希望你在自己的扩展类中重写此方法。

例子：简单的协议分析器 V2

这里的简易协议解析器例子能够很简单地使用高级的 Transform 流类实现，类似于 `parseHeader` 和 `SimpleProtocol v1` 示例。

在这个示例中，输入会被导流到解析器中，而不是作为一个输入的参数提供。这种做法更符合 Node.js 流的惯例。

```
const util = require('util');
const Transform = require('stream').Transform;
util.inherits(SimpleProtocol, Transform);

function SimpleProtocol(options) {
    if (!(this instanceof SimpleProtocol))
```

```
        return new SimpleProtocol(options);

    Transform.call(this, options);
    this._inBody = false;
    this._sawFirstCr = false;
    this._rawHeader = [];
    this.header = null;
}

SimpleProtocol.prototype._transform = function (chunk, encoding, done) {
    if (!this._inBody) {
        // check if the chunk has a \n\n
        var split = -1;
        for (var i = 0; i < chunk.length; i++) {
            if (chunk[i] === 10) { // '\n'
                if (this._sawFirstCr) {
                    split = i;
                    break;
                } else {
                    this._sawFirstCr = true;
                }
            } else {
                this._sawFirstCr = false;
            }
        }

        if (split === -1) {
            // still waiting for the \n\n
            // stash the chunk, and try again.
            this._rawHeader.push(chunk);
        } else {
            this._inBody = true;
            var h = chunk.slice(0, split);
            this._rawHeader.push(h);
            var header = Buffer.concat(this._rawHeader).toString();
            try {
                this.header = JSON.parse(header);
            } catch (er) {
                this.emit('error', new Error('invalid simple protocol data'));
                return;
            }
            // and let them know that we are done parsing the header.
            this.emit('header', this.header);

            // now, because we got some extra data, emit this first.
            this.push(chunk.slice(split));
        }
    } else {
        // from there on, just provide the data to our consumer as-is.
        this.push(chunk);
    }
    done();
};
```

```
// Usage:
// var parser = new SimpleProtocol();
// source.pipe(parser)
// Now parser is a readable stream that will emit 'header'
// with the parsed header data.
```

stream.PassThrough类

这是转换（Transform）流的一个简单实现，将输入的字节简单地传递给输出。它的主要用途是演示和测试，但偶尔也能在构建某种特殊流时派上用场。

简化的构造函数API

- Readable
- Writable
- Duplex
- Transform

在简单的情况下，不通过继承来构建流，现在会有一些额外的好处。

这可以通过传递适当的方法作为构造选项来实现：

例子：

Readable

```
var readable = new stream.Readable({
    read: function (n) {
        // sets this._read under the hood

        // push data onto the read queue, passing null
        // will signal the end of the stream (EOF)
        this.push(chunk);
    }
});
```

Writable

```
var writable = new stream.Writable({
    write: function (chunk, encoding, next) {
        // sets this._write under the hood

        // An optional error can be passed as the first argument
        next()
    }
});

// or

var writable = new stream.Writable({
    writev: function (chunks, next) {
        // sets this._writev under the hood

        // An optional error can be passed as the first argument
        next()
    }
});
```

Duplex

```
var duplex = new stream.Duplex({
    read: function (n) {
        // sets this._read under the hood

        // push data onto the read queue, passing null
        // will signal the end of the stream (EOF)
        this.push(chunk);
    },
    write: function (chunk, encoding, next) {
        // sets this._write under the hood

        // An optional error can be passed as the first argument
        next()
    }
});

// or

var duplex = new stream.Duplex({
    read: function (n) {
        // sets this._read under the hood

        // push data onto the read queue, passing null
        // will signal the end of the stream (EOF)
        this.push(chunk);
    },
    writev: function (chunks, next) {
        // sets this._writev under the hood

        // An optional error can be passed as the first argument
        next()
    }
});
```

Transform

```
var transform = new stream.Transform({
    transform: function (chunk, encoding, next) {
        // sets this._transform under the hood

        // generate output as many times as needed
        // this.push(chunk);

        // call when the current chunk is consumed
        next();
    },
    flush: function (done) {
        // sets this._flush under the hood

        // generate output as many times as needed
        // this.push(chunk);

        done();
    }
});
```

流的内部细节

缓冲

无论是可写（Writable）流还是可读（Readable）流，它们都会在内部对象中缓冲数据，这两个内部对象分别可以从 `_writableState.getBuffer()` 和 `_readableState.buffer` 中获取。

被缓冲的数据量取决于传递给构造函数的 `highWaterMark` 选项。

可读（Readable）流的滞留发生于实现调用 stream.push(chunk) 时。如果流的消费者没有调用 stream.read()，那么数据将待在内部队列中，直到被消费。

可写（Writable）流的滞留发生于用户反复调用 stream.write(chunk) 时，即便此时返回 `false`。

流（特别是其中的 stream.pipe() 方法）的初衷是将数据的滞留量限制在一个可接受的水平，以便不同速度的资源和目标不会淹没可用内存。

与 Node.js 早期版本的兼容性

在 v0.10 之前版本的 Node.js 中，可读（Readable）流的接口较为简单，同时功能和实用性也较弱。

- 'data' 事件会在开始时立即发生，而不会等你调用 stream.read() 方法。如果你需要进行某些 I/O 来决定如何处理数据，那么你只能将数据块储存到某种缓冲区中以防它们流失。
- stream.pause() 方法仅供参考，不保证生效。意味着即便当流处于暂停状态时，你仍需要准备接收 'data' 事件。

在 Node.js v0.10 中新增了 Readable 类。为了向后兼容考虑，可读（Readable）流会在添加了 'data' 事件监听器或调用 stream.resume() 方法时切换至“流动模式”。其效果是，即便你不使用新的 stream.read() 方法和 'readable' 事件，你也不必担心丢失 'data' 事件产生的数据块。

大多数程序会维持正常功能。然而，会在下列条件下引入一种边界情况：

- 没有添加 'data' 事件处理器。
- stream.resume() 方法从未被调用。
- 流未被导流到任何可写目标。

例如，请留意以下代码：

```
// WARNING!  BROKEN!
net.createServer((socket) => {

    // we add an 'end' method, but never consume the data
    socket.on('end', () => {
        // It will never get here.
        socket.end('I got your message (but didnt read it)\n');
    });

}).listen(1337);
```

在 Node.js v0.10 之前的版本中，传入消息数据会被简单地丢弃。然而在 Node.js v0.10 及之后的版本中，socket 会一直保持暂停。

在这种情况下的解决方法是调用 stream.resume() 方法来开启数据流：

```
// Workaround
net.createServer((socket) => {

    socket.on('end', () => {
        socket.end('I got your message (but didnt read it)\n');
    });

    // start the flow of data, discarding it.
    socket.resume();

}).listen(1337);
```

除了流动模式下新建可读（Readable）流以外，v0.10 之前风格的流可以通过 stream.wrap() 方法包装成 Readable 类。

对象模式

通常情况下，流只操作字符串和 Buffer。

处于对象模式的流除了 Buffer 和字符串外还能读取普通的 JavaScript 值。

一个处于对象模式的可读（Readable）流调用 stream.read(size) 时总会返回单个项目，无论传入什么 `size` 参数。

一个处于对象模式的可写（Writable）流总是会忽略传给 stream.write(data, encoding) 的 `encoding` 参数。

特殊值 `null` 在对象模式流中依旧保持它的特殊性。也就说，对于对象模式的可读流，stream.read() 返回 `null` 意味着没有更多数据，同时 stream.push(null) 标志着流数据已到达末端（`EOF`）。

Node.js 核心不存在对象模式的流。这种设计只被某些用户态流式库所使用。这种模式只用于用户级流库。

你应该在你的流子类构造函数的选项对象中设置 `objectMode`。在流的过程中设置 `objectMode` 是不安全的。

对于双工（Duplex）流而言，`objectMode` 只可以在可读端和可写端分别通过 `readableObjectMode` 和 `writableObjectMode` 来设置。这些选项可以被用于在转换（Transform）流中实现解析器和串行器。

```
const util = require('util');
const StringDecoder = require('string_decoder').StringDecoder;
const Transform = require('stream').Transform;
util.inherits(JSONParseStream, Transform);

// Gets \n-delimited JSON string data, and emits the parsed objects
function JSONParseStream() {
    if (!(this instanceof JSONParseStream))
        return new JSONParseStream();

    Transform.call(this, {
        readableObjectMode: true
    });

    this._buffer = '';
    this._decoder = new StringDecoder('utf8');
}

JSONParseStream.prototype._transform = function (chunk, encoding, cb) {
    this._buffer += this._decoder.write(chunk);
    // split on newlines
    var lines = this._buffer.split(/\r?\n/);
    // keep the last partial line buffered
    this._buffer = lines.pop();
    for (var l = 0; l < lines.length; l++) {
        var line = lines[l];
        try {
            var obj = JSON.parse(line);
        } catch (er) {
            this.emit('error', er);
```

```
            return;
        }
        // push the parsed object out to the readable consumer
        this.push(obj);
    }
    cb();
};

JSONParseStream.prototype._flush = function (cb) {
    // Just handle any leftover
    var rem = this._buffer.trim();
    if (rem) {
        try {
            var obj = JSON.parse(rem);
        } catch (er) {
            this.emit('error', er);
            return;
        }
        // push the parsed object out to the readable consumer
        this.push(obj);
    }
    cb();
};
```

stream.read(0)

在某些情况下，你可能需要在不真正消费任何数据的情况下，触发底层可读流机制的刷新。在这种情况下，你可以调用 `stream.read(0)`，它总会返回 null。

如果内部读取缓冲低于 highWaterMark 水位线，并且流当前不在读取状态，那么调用 `stream.read(0)` 会触发一个低级 stream._read() 调用。

虽然几乎没有必要这么做，但你可以在 Node.js 内部的某些地方看到它确实这么做了，尤其是在可读（Readable）流类的内部。

stream.push('')

推入一个零字节字符串或 Buffer（当不在对象模式时）有一个有趣的副作用。因为它是一个对 stream.push() 的调用，它会结束 `reading` 过程。然而，它没有添加任何数据到可读缓冲中，所以没有东西可以被用户消费。

在极少数情况下，你当时没有提供任何数据，但你的流的消费者（或你的代码的其它部分）会通过调用 stream.read(0) 得知何时需要再次检查。在这中情况下，你可以调用 `stream.push('')`。

到目前为止，这个功能唯一一个使用场景是在 tls.CryptoStream 类中，但在 Node.js/io.js v1.0 中已被废弃。如果你发现你不得不使用 `stream.push('')`，请考虑另一种方式，因为几乎可以明确表明这是某种可怕的错误。

查询字符串(Query Strings)

> 稳定度：2 - 稳定

该模块提供了一些处理查询字符串的实用程序。它提供以下方法：

方法和属性

---

querystring.stringify(obj[, sep][, eq][, options])

序列化一个对象到一个查询字符串。可以选择是否覆盖默认的分割符（`'&'`）和分配符（`'='`）。

Options 对象可能包含 `encodeURIComponent` 属性（默认为 `querystring.escape`）。在有必要时，它可以用 `non-utf8` 码来编码字符串。

例子：

```
querystring.stringify({
        foo: 'bar',
        baz: ['qux', 'quux'],
        corge: ''
    })
    // returns 'foo=bar&baz=qux&baz=quux&corge='

querystring.stringify({
        foo: 'bar',
        baz: 'qux'
    }, ';', ':')
    // returns 'foo:bar;baz:qux'

// Suppose gbkEncodeURIComponent function already exists,
// it can encode string with `gbk` encoding
querystring.stringify({
        w: '中文',
        foo: 'bar'
    }, null, null, {
        encodeURIComponent: gbkEncodeURIComponent
    })
    // returns 'w=%D6%D0%CE%C4&foo=bar'
```

querystring.parse(str[, sep][, eq][, options])

将一个查询字符串反序列化为一个对象。可以选择是否覆盖默认的分割符（`'&'`）和分配符（`'='`）。

Options 对象可能包含 `maxKeys` 属性（默认为 1000），它会被用来限制已处理键（key）的数量。设为 0 可以去除键（key）的数量限制。

Options 对象可能包含 `decodeURIComponent` 属性（默认为 `querystring.unescape`）。在有必要时，它可以将 `non-utf8` 码解码为字符串。

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge')
// returns { foo: 'bar', baz: ['qux', 'quux'], corge: '' }

// Suppose gbkDecodeURIComponent function already exists,
// it can decode `gbk` encoding string
querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null,
  { decodeURIComponent: gbkDecodeURIComponent })
// returns { w: '中文', foo: 'bar' }
```

querystring.escape

供 `querystring.stringify` 使用的转意函数，在必要的时候可被重写。

querystring.unescape

供 `querystring.parse` 使用的反转意函数，在必要的时候可被重写。

它会首先尝试使用 `encodeURIComponent`，但如果失败，它会回滚到一个安全的等效结果上，在处理畸形的 URL 时也不会抛出错误。

字符串解码(String Decoder)

> 稳定度：2 - 稳定

通过 `require('string_decoder')` 使用该模块。这个模块将一个 buffer 解码成一个字符串。它是 `buffer.toString()` 的一个简单接口，但提供对 utf8 的支持。

```
const StringDecoder = require('string_decoder').StringDecoder;
const decoder = new StringDecoder('utf8');

const cent = new Buffer([0xC2, 0xA2]);
console.log(decoder.write(cent));

const euro = new Buffer([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

StringDecoder类

接受唯一一个参数 `encoding`，默认为 `'utf8'`。

decoder.write(buffer)

返回一个解码后的字符串。

decoder.end()

返回被留在缓冲区中的任何末尾字节。

系统(OS)

> 稳定度：2 - 稳定

提供了一些基本的与操作系统相关的实用参数。

通过 `require('os')` 使用该模块。

方法和属性

---

os.EOL

一个定义了操作系统相应的行尾（End-of-line）标识的常量。

os.type()

返回操作系统名称。比如 Linux 中返回 `'Linux'`，OS X 中返回 `'Darwin'`，Windows 中返回 `'Windows_NT'`。

os.release()

返回操作系统版本。

os.platform()

返回操作系统平台。可能的值有 `'darwin'`，`'freebsd'`，`'linux'`，`'sunos'` 和 `'win32'`。返回 process.platform 的值。

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）。

os.cpus 的示例：

```
[{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 252020,
        nice: 0,
        sys: 30340,
        idle: 1070356870,
        irq: 0
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 306960,
        nice: 0,
        sys: 26980,
        idle: 1071569080,
        irq: 0
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 248450,
        nice: 0,
        sys: 21750,
        idle: 1070919370,
        irq: 0
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 256880,
        nice: 0,
```

```
        sys: 19430,
        idle: 1070905480,
        irq: 20
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 511580,
        nice: 20,
        sys: 40900,
        idle: 1070842510,
        irq: 0
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 291660,
        nice: 0,
        sys: 34360,
        idle: 1070888000,
        irq: 10
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 308260,
        nice: 0,
        sys: 55410,
        idle: 1071129970,
        irq: 880
    }
},
{
    model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times: {
        user: 266450,
        nice: 1480,
        sys: 34920,
        idle: 1072572010,
        irq: 30
    }
}]
```

os.arch()

返回操作系统的 CPU 架构。可能的值有 `'x64'`，`'arm'` 和 `'ia32'`。返回 process.arch 的值。

os.endianness()

返回 CPU 的字节序。可能值有大端的 `BE` 和小端的 `LE`。

os.loadavg()

返回一个包含 1、5、15 分钟平均负载的数组。

平均负载是一个系统活跃度指标，它由操作系统计算并表示为一个小数。根据经验，平均负载最好应小于系统逻辑 CPU 的数量。

平均负载是一个非常 UNIX-y 的概念；在 Windows 平台上没有真正的等价物。这就是为什么这个函数在 Windows 上始终返回 `[0,0,0]` 的原因。

os.totalmem()

返回系统内存总量，以字节为单位。

os.freemem()

返回可用的系统内存量，以字节为单位。

os.uptime()

返回操作系统的运行时间，以秒为单位。

os.networkInterfaces()

获取网络接口列表：

```
{
    lo: [{
            address: '127.0.0.1',
            netmask: '255.0.0.0',
            family: 'IPv4',
            mac: '00:00:00:00:00:00',
            internal: true
        },
        {
            address: '::1',
            netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
            family: 'IPv6',
            mac: '00:00:00:00:00:00',
            internal: true
        }],
    eth0: [{
            address: '192.168.1.108',
            netmask: '255.255.255.0',
            family: 'IPv4',
            mac: '01:02:03:0a:0b:0c',
            internal: false
        },
        {
            address: 'fe80::a00:27ff:fe4e:66a1',
            netmask: 'ffff:ffff:ffff:ffff::',
            family: 'IPv6',
            mac: '01:02:03:0a:0b:0c',
            internal: false
        }]
}
```

需要注意的是，由于底层的实现，这里只会返回已分配地址的网络接口。

os.hostname()

返回操作系统的主机名。

os.homedir()

返回当前用户的主目录。

os.tmpdir()

返回操作系统默认的临时文件目录。

进程(Process)

`process` 对象是一个全局对象，可以在任何地方访问。它是一个 EventEmitter 实例。

方法和属性

事件

属性

方法

- process.exit([code])
- process.abort()
- process.kill(pid[, signal])
- process.cwd()
- process.chdir(directory)
- process.memoryUsage()
- process.umask([mask])
- process.uptime()
- process.hrtime()
- process.setuid(id)
- process.getuid()
- process.setgid(id)
- process.getgid()
- process.seteuid(id)
- process.geteuid()
- process.setegid(id)
- process.getegid()
- process.setgroups(groups)
- process.getgroups()
- process.initgroups(user, extra_group)

'message' 事件

- `message` {Object} 一个已解析的 JSON 对象或原始值
- `sendHandle` {Handle 对象} 一个 net.Socket 或 net.Server 或 `undefined`。

通过 ChildProcess.send() 发送的信息，需要在子进程对象中使用 `'message'` 事件获取。

'exit' 事件

当进程即将退出时触发。在这一点上，没有办法防止事件循环的退出，一旦所有的 `'exit'` 监听器已经完成监听，运行的进程将会退出。因此，你只能在处理程序时执行同步操作。这有利于对模块的状态进行检查（如单元测试）。进程代码退出时的回调函数有一个参数。

该事件只有当 Node.js 明确的通关过 `process.exit()` 退出或事件循环隐式外泄时才被触发。

监听 `'exit'` 的示例：

```
process.on('exit', (code) => {
    // do *NOT* do this
    setTimeout(() => {
        console.log('This will not run');
    }, 0);
    console.log('About to exit with code:', code);
});
```

'beforeExit' 事件

该事件在 Node.js 清空其事件循环并且没有安排其他事情的情况下触发。通常情况下，Node.js 在没有计划工作时退出，但 `'beforeExit'` 监听器会进行异步调用，从而导致 Node.js 继续运行。

`'beforeExit'` 在显式终止的情况下不会触发，比如 process.exit() 或未捕获的异常，并且不应该将其替代 `'exit'` 事件，除非你是为了安排更多的工作。

'rejectionHandled' 事件

每当 Promise 被拒绝时触发，并且在一个事件循环后会给它附加一个错误程序（比如 `.catch()`）。此事件触发时带有以下参数：

- `p` 代表在 `'unhandledRejection'` 事件触发前的那个 promise，但目前获得了一个拒绝程序。

对于 promise 链而言，目前没有一个总可以用于处理拒绝的顶级概念。作为一个在本质上是异步，一个 promise 拒绝将在之后得到处理 —— 可能远远超过了事件循环以后触发的 `'unhandledRejection'` 事件的花费。

说明这一点的另一种方式是，不像在同步代码中有一个不断增长的未处理的异常列表，promise 有着不断增长和收缩的未处理的拒绝列表。在同步代码中，`'uncaughtException'` 事件告诉你何时未处理的异常列表在不断增长。在异步代码中，`'unhandledRejection'` 事件告诉你何时未处理的拒绝列表在增长，`'rejectionHandled'` 事件告诉你何时未处理的拒绝列表在收缩。

例如使用拒绝检查钩子以便于在给给定时间内保持一份所有被拒绝的 promise 的理由的映射：

```
const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, p) => {
    unhandledRejections.set(p, reason);
});
process.on('rejectionHandled', (p) => {
    unhandledRejections.delete(p);
});
```

这份映射表会随时间的推移增长和收缩，反映了拒绝在何时未处理，何时被处理。你可以在一些错误日志中记录这些错误，无论是定期（尤其是针对长期运行的程序，在一个错误可能无限增长的的程序下，允许你清理该映射）还是在进程退出时（对脚本而言更方便）。

'unhandledRejection' 事件

每当 Promise 被拒绝时触发，没有错误处理程序附加到事件循环内的 promise 上。当程序设计中的 promise 异常被封装成拒绝 promise 时，这样的 promise 可以使用 promise.catch(...) 捕捉和处理，并且拒绝可以通过 promise 链传播。该事件有利于检测和保持对还没被处理的那些被拒绝的 promise 的跟踪。该事件触发时带有以下参数：

- `reason` 包含被拒绝的 promise 的对象（通常是一个错误实例）。
- `p` 被拒绝的 promise。

这是一个将每一个未处理的拒绝记录到控制台的例子：

```
process.on('unhandledRejection', (reason, p) => {
    console.log("Unhandled Rejection at: Promise ", p, " reason: ", reason);
    // application specific logging, throwing an error, or other logic here
});
```

例如，这是一个将会触发 `'unhandledRejection'` 事件的拒绝：

```
somePromise.then((res) => {
    return reportToUser(JSON.pasre(res)); // note the typo (`pasre`)
}); // no `.catch` or `.then`
```

这是一个也会触发 `'unhandledRejection'` 事件的编码模式的例子：

```
function SomeResource() {
    // Initially set the loaded status to a rejected promise
    this.loaded = Promise.reject(new Error('Resource not yet loaded!'));
}

var resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

在这种情况下，你可能不希望将跟踪拒绝作为一个开发者的错误，就像你对其他 `'unhandledRejection'` 事件那样。为了解决这个问题，你可以在 `resource.loaded` 上附加一个假的 `.catch(() => { })` 处理器，防止触发 `'unhandledRejection'` 事件，或者你也可以使用 'rejectionHandled' 事件。

'uncaughtException' 事件

当异常一路冒泡回事件循环时会触发 `'uncaughtException'` 事件。默认，Node.js 通过在 stderr 中打印堆栈跟踪并退出来处理这种情况。可以通过给 `'uncaughtException'` 事件添加处理器来覆盖这种默认行为。

例如：

```
process.on('uncaughtException', (err) => {
    console.log(`Caught exception: ${err}`);
});

setTimeout(() => {
    console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

警告：请正确使用 'uncaughtException' 事件

请注意，`'uncaughtException'` 是一种异常处理的粗机制，希望只作为最后的手段。该事件不应该被用做 `On Error Resume Next` 的替代品。未处理的异常本质上意味着应用程序处于不确定状态。试图恢复应用程序代码而不是从异常恢复正常可能会导致额外的不可预见的和不可预知的问题。

从事件处理程序中抛出的异常不会被捕获。相反，会以非零退出代码退出进程，并且打印堆栈跟踪。这是为了避免无限递归。

尝试从一个未捕获到异常后恢复正常类似于在升级电脑时拔出电源线——十次里的前九次时间是没有任何反应的——但在第十次的时候系统损坏了。

`'uncaughtException'` 的正确用法是在进程关闭前执行分配资源的同步清理（如，文件描述符、处理程序等）。在 `'uncaughtException'` 事件后恢复正常操作是不安全的。

process.stdin

一个指向标准输入（`stdin` on fd `0`）的 `可读（Readable）流`。

打开标准输入并监听两个事件的示例：

```
process.stdin.setEncoding('utf8');

process.stdin.on('readable', () => {
    var chunk = process.stdin.read();
    if (chunk !== null) {
        process.stdout.write(`data: ${chunk}`);
    }
});

process.stdin.on('end', () => {
    process.stdout.write('end');
});
```

作为流（Stream），`process.stdin` 也可以使用兼容 Node.js v0.10 之前版本编写的脚本的“老”模式。更多信息详见流的兼容性。

在“老”的流模式中，标准输入流默认是暂停状态，所以读取时必须调用 `process.stdin.resume()`。请注意，调用 `process.stdin.resume()` 时也会将流自身转换至“老”模式。

如果你开始一个新项目，你应该更喜欢最“新”的流模式而非“老”模式。

process.stdout

一个指向标准输出（`stdout` on fd `1`）的 `可写（Writable）流`。

例如，`console.log` 的等效写法如下：

```
console.log = (msg) => {
    process.stdout.write(`${msg}\n`);
};
```

`process.stderr` 和 `process.stdout` 不像 Node.js 中其他的流，它们无法被关闭（`end()` 将会抛出），它们从不触发 `'finish'` 事件，并且当输出重定向到一个文件时可以阻止其写入（虽然磁盘很快并且操作系统通常采用回写式缓存，但它应该确实是一个非常罕见的情况。）。

要检查 Node.js 是否运行在一个 TTY 上下文中，可以使用 `process.stderr`、`process.stdout` 或 `process.stdin` 中 `isTTY` 属性：

```
$ node -p "Boolean(process.stdin.isTTY)"
true
$ echo "foo" | node -p "Boolean(process.stdin.isTTY)"
false

$ node -p "Boolean(process.stdout.isTTY)"
true
$ node -p "Boolean(process.stdout.isTTY)" | cat
false
```

更多信息详见tty文档。

process.stderr

一个指向标准错误（`stderr` on fd `2`）的 `可写（Writable）流`。

`process.stderr` 和 `process.stdout` 不像 Node.js 中其他的流，它们无法被关闭（`end()` 将会抛出），它们从不触发 `'finish'` 事件，并且当输出重定向到一个文件时可以阻止其写入（虽然磁盘很快并且操作系统通常采用回写式缓存，但它应该确实是一个非常罕见的情况。）。

process.pid

进程的 PID：

```
console.log(`This process is pid ${process.pid}`);
```

process.config

返回包含了用来编译当前 Node.js 可执行程序的配置选项的 JSON 对象。内容与运行 `./configure` 脚本生成的 `config.gypi` 文件相同。

可能的输出示例如下：

```
{
    target_defaults: {
        cflags: [],
        default_configuration: 'Release',
        defines: [],
        include_dirs: [],
        libraries: []
    },
    variables: {
        host_arch: 'x64',
        node_install_npm: 'true',
        node_prefix: '',
        node_shared_cares: 'false',
        node_shared_http_parser: 'false',
        node_shared_libuv: 'false',
        node_shared_zlib: 'false',
        node_use_dtrace: 'false',
        node_use_openssl: 'true',
        node_shared_openssl: 'false',
        strict_aliasing: 'true',
        target_arch: 'x64',
        v8_use_snapshot: 'true'
    }
}
```

process.env

返回包含用户环境中的对象。详见environ(7)。

此对象的示例如下：

```
{
    TERM: 'xterm-256color',
    SHELL: '/usr/local/bin/bash',
    USER: 'maciej',
    PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
    PWD: '/Users/maciej',
    EDITOR: 'vim',
    SHLVL: '1',
    HOME: '/Users/maciej',
    LOGNAME: 'maciej',
    _: '/usr/local/bin/node'
}
```

你可以写入该对象，但修改不会在你的程序之外得到反映。这意味着以下代码不会起作用：

```
$ node -e 'process.env.foo = "bar"' && echo $foo
```

但这个是可以的：

```
process.env.foo = 'bar';
console.log(process.env.foo);
```

在 `process.env` 上分配的属性会隐式的将值转换为字符串。

示例：

```
process.env.test = null;
console.log(process.env.test);
// => 'null'
process.env.test = undefined;
console.log(process.env.test);
// => 'undefined'
```

可以使用 `delete` 删除 `process.env` 上的属性。

示例：

```
process.env.TEST = 1;
delete process.env.TEST;
console.log(process.env.TEST);
// => undefined
```

process.platform

你正运行在什么平台上：`'darwin'`, `'freebsd'`, `'linux'`, `'sunos'` 或 `'win32'`。

```
console.log(`This platform is ${process.platform}`);
```

process.arch

你正运行在什么处理器架构上：`'arm'`, `'ia32'` 或 `'x64'`。

```
console.log(`This processor architecture is ${process.arch}`);
```

process.release

一个包含当前版本元数据的对象，包括源码包网址和压缩包头。

`process.release` 包含以下属性：

- `name`：带值的字符串。Node.js 中永远是 `'node'`；io.js 中永远是 `'io.js'`。
- `sourceUrl`：一个指向当前版本的 *.tar.gz* 文件的完整 URL。
- `headersUrl`：一个指向当前版本的 *.tar.gz* 头文件的完整 URL。该文件比完整源文件明显要小，并且可用于编译针对 Node.js 的插件。
- `libUrl`：一个指向匹配当前版本的结构和版本号的 *node.lib* 文件。该文件可用于编译针对 Node.js 的插件。此属性仅出现在 *Windows* 版的 *Node.js* 中，并且不会出现在所有其它平台上。

例如：

```
{
    name: 'node',
    sourceUrl: 'https://nodejs.org/download/release/v4.0.0/node-v4.0.0.tar.gz',
    headersUrl: 'https://nodejs.org/download/release/v4.0.0/node-v4.0.0-headers.tar.gz
',
    libUrl: 'https://nodejs.org/download/release/v4.0.0/win-x64/node.lib'
}
```

在从源代码树自定义构建的非发布版本中，只可能存在 `name` 属性，附加属性不应该存在。

process.title

获取/设置 (Getter/setter) `ps` 中显示的进程名。

当设置该属性时，所能设置的字符串最大长度视具体平台而定，如果超过的话会自动截断。

在 Linux 和 OS X 上，它受限于名称的字节长度加上命令行参数的长度，因为它覆写了参数内存（argv memory）。

v0.8 版本允许更长的进程标题字符串，也支持覆盖环境内存，但在一些案例中这是潜在的不安全/混淆（很难说清楚）。

process.connected

- {Boolean} 在调用 process.disconnect() 后设为 false。

如果 `process.connected` 为 false，它不再能够发送信息。

exitCode

当进程正常退出或通过不带指定代码的 process.exit() 退出时，那个代表进程退出代码的数字。

指定 `process.exit(code)` 的代码，将会覆盖任何先前设定的 `process.exitCode` 。

process.mainModule

检索 require.main 的另一种方法。不同的是，如果主模块在运行时改变，`require.main` 仍然会指向发生变化之前请求的模块的原始主模块。通常可以安全地假设两个都是指向相同的模块。

针对 `require.main` 而言，如果没有入口脚本，它可能会变成 `undefined` 。

process.argv

一个包含命令行参数的数组。第一个元素会是 'node'，第二个元素会是该 JavaScript 文件的名称。接下来的元素会是任何额外的命令行参数。

```
// print process.argv
process.argv.forEach((val, index, array) => {
    console.log(`${index}: ${val}`);
});
```

这将产生：

```
$ node process-2.js one two=three four
0: node
1: /Users/mjr/work/node/process-2.js
2: one
3: two=three
4: four
```

process.execPath

这是启动进程可执行文件的绝对路径。

示例：

```
/usr/local/bin/node
```

process.execArgv

这是启动该进程的指定的 Node.js 可执行文件的命令行参数的设置。这些选项不在 `process.argv` 中显示，并且不包括 Node.js 的可执行文件，脚本名称或任何跟在脚本名称后的参数。这些参数在为了使衍生子进程和父进程有相同的执行环境时非常有用。

示例：

```
$ node --harmony script.js --version
```

`process.execArgv` 的结果是：

```
['--harmony']
```

`process.argv` 的结果是：

```
['/usr/local/bin/node', 'script.js', '--version']
```

process.version

一个暴露编译时的 `NODE_VERSION` 的属性。

```
console.log(`Version: ${process.version}`);
```

process.versions

一个暴露 Node.js 及其依赖的版本字符串的属性。

```
console.log(process.versions);
```

将打印类似于：

```
{
    http_parser: '2.3.0',
    node: '1.1.1',
    v8: '4.1.0.14',
    uv: '1.3.0',
    zlib: '1.2.8',
    ares: '1.10.0-DEV',
    modules: '43',
    icu: '55.1',
    openssl: '1.0.1k'
}
```

process.send(message[, sendHandle[, options]][, callback])

- `message` {Object}
- `sendHandle` {Handle object}
- `options` {Object}
- `callback` {Function}
- 返回：{Boolean}

当 Node.js 衍生出一个附加的 IPC 通道时，它可以使用 `process.send()` 向父进程发送信息。每个父进程上的 `ChildProcess` 对象都会收到 'message' 事件。

注意：该函数使用 *JSON.stringify()* 内部序列化 *`message`*。

如果 Node.js 没有衍生出一个 IPC 通道，那么 `process.send()` 会是未定义的。

process.nextTick(callback[, arg][, ...])

- `callback` {Function}

一旦当前的事件循环完成了一圈的运行，就会调用该回调函数。

这不是 setTimeout(fn, 0) 的一个简单的别名，因为它的效率高多了。在事件循环的后续时间刻度（subsequent ticks）中，它运行在任何附加的 I/O 事件（包括定时器）触发之前。

```
console.log('start');
process.nextTick(() => {
    console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

如果你想要在对象创建后，但在任何 I/O 操作发生前执行某些操作，那么这个函数对你而言就十分重要了。

```
function MyThing(options) {
    this.setupOptions(options);

    process.nextTick(() => {
        this.startDoingStuff();
    });
}

var thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.
```

在使用该函数时，请保证你的函数一定是同步或者一定是异步执行的。请思考这个例子：

```
// WARNING!  DO NOT USE!  BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
    if (arg) {
        cb();
        return;
    }

    fs.stat('file', cb);
}
```

这样执行是很危险。如果你还不清楚上述行为的危害请看下面的例子：

```
maybeSync(true, () => {
    foo();
});
bar();
```

那么，使用刚才那个不知道是同步还是异步的操作，在执行时你就会发现，你无法确定到底是 `foo()` 先执行，还是 `bar()` 先执行。

以下是更好的解决方案：

```
function definitelyAsync(arg, cb) {
    if (arg) {
        process.nextTick(cb);
        return;
    }

    fs.stat('file', cb);
}
```

注意：nextTick 的队列会在完全执行完毕之后才会去完成其他的 I/O 操作。因此，递归设置 nextTick 的回调就像一个 `while(true);` 循环一样，将会阻止任何 I/O 操作的发生。

process.disconnect()

关闭父进程的 IPC 通道，一旦没有进程与其保持连接状态，就会允许子进程正常退出。

等同于父进程的 ChildProcess.disconnect()。

如果 Node.js 没有衍生出一个 IPC 通道，那么 `process.disconnect()` 会是未定义的。

process.exit([code])

通过指定的 `code` 结束进程。如果省略，“成功”退出时将使用代码 `0`。

以失败状态（'failure' code）退出：

```
process.exit(1);
```

在执行 Node.js 的 shell 中就可以看到退出代码为 1。

process.abort()

这将导致 Node.js 触发一个 'abort' 事件，这会导致 Node.js 退出并且生成一个核心文件。

process.kill(pid[, signal])

向进程发送一个信号。`pid` 是进程的 id 而 `signal` 则是描述信号的字符串名称。信号名称都类似于 `SIGINT` 或 `SIGHUP`。如果省略 signal 参数，则默认为 `SIGTERM`。详见信号事件和 kill(2)。

如果目标不存在将抛出错误。作为一个特殊情况，`0` 信号可以用于测试信号是否存在。如果在 Windows 平台上的 `pid` 被用于杀死一个进程组，那么它将抛出一个错误。

注意，尽管函数名叫 `process.kill`，但它实际上只是一个信号发送器，就像 `kill` 系统调用。信号的发送可能会做除了杀死目标进程以外的一些其他事情。

将信号发送到自己的例子：

```
process.on('SIGHUP', () => {
    console.log('Got SIGHUP signal.');
});

setTimeout(() => {
    console.log('Exiting.');
    process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

注意：当 SIGUSR1 是由 Node.js 接收时，它将开启调试器，见信号事件。

process.cwd()

返回进程的当前工作目录。

```
console.log(`Current directory: ${process.cwd()}`);
```

process.chdir(directory)

改变进程的当前工作目录，如果操作失败，则抛出一个异常。

```
console.log(`Starting directory: ${process.cwd()}`);
try {
    process.chdir('/tmp');
    console.log(`New directory: ${process.cwd()}`);
} catch (err) {
    console.log(`chdir: ${err}`);
}
```

process.uptime()

返回 Node.js 运行的秒数。

process.hrtime()

以 `[seconds, nanoseconds]` 元组数组的形式返回当前的高精度真实时间。它相对于过去的任意时间。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。

你可以通过之前调用的 `process.hrtime()` 的结果和当前的 `process.hrtime()` 来获取一个差异读数，这对于基准和衡量时间间隔非常有用：

```
var time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
    var diff = process.hrtime(time);
    // [ 1, 552 ]

    console.log('benchmark took %d nanoseconds', diff[0] * 1e9 + diff[1]);
    // benchmark took 1000000527 nanoseconds
}, 1000);
```

process.memoryUsage()

返回一个描述 Node.js 进程的内存使用量的对象，以字节（bytes）为单位。

```
const util = require('util');

console.log(util.inspect(process.memoryUsage()));
```

这会产生：

```
{
    rss: 4935680,
    heapTotal: 1826816,
    heapUsed: 650472
}
```

`heapTotal` 和 `heapUsed` 引用 V8 引擎的内存使用量。

process.umask([mask])

设置或读取进程的文件模式的创建掩码。子进程从父进程中继承掩码。如果设定了 `mask` 参数，则返回旧的掩码，否则返回当前的掩码。

```
const newmask = 0o022;
const oldmask = process.umask(newmask);
console.log(
    `Changed umask from ${oldmask.toString(8)} to ${newmask.toString(8)}`
);
```

process.setuid(id)

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

设置进程用户的标识（见 setuid(2)）。它可以接收数字 ID 或一个用户名字符串。如果指定了用户名，该方法会阻塞运行直到将它解析为一个数字 ID 为止。

```
if (process.getuid && process.setuid) {
    console.log(`Current uid: ${process.getuid()}`);
    try {
        process.setuid(501);
        console.log(`New uid: ${process.getuid()}`);
    } catch (err) {
        console.log(`Failed to set uid: ${err}`);
    }
}
```

process.getuid()

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

获取进程用户标识（见 getuid(2)）。这是数字的用户 id，而非用户名。

process.setgid(id)

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

设置进程组标识（见 setgid(2)）。它可以接收数字 ID 或一个组名称的字符串。如果指定了组名称，该方法会阻塞运行直到将它解析为一个数字 ID 为止。

```
if (process.getgid && process.setgid) {
    console.log(`Current gid: ${process.getgid()}`);
    try {
        process.setgid(501);
        console.log(`New gid: ${process.getgid()}`);
    } catch (err) {
        console.log(`Failed to set gid: ${err}`);
    }
}
```

process.getgid()

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

获取进程组标识（见 getgid(2)）。这是数字的组 id，而非组名称。

```
if (process.getgid) {
    console.log(`Current gid: ${process.getgid()}`);
}
```

process.seteuid(id)

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

设置有效的进程用户的标识（见 seteuid(2)）。它可以接收数字 ID 或一个用户名字符串。如果指定了用户名，该方法会阻塞运行直到将它解析为一个数字 ID 为止。

```
if (process.geteuid && process.seteuid) {
    console.log(`Current uid: ${process.geteuid()}`);
    try {
        process.seteuid(501);
        console.log(`New uid: ${process.geteuid()}`);
    } catch (err) {
        console.log(`Failed to set uid: ${err}`);
    }
}
```

process.geteuid()

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

获取有效的进程用户标识（见 geteuid(2)）。这是数字的用户 id，而非用户名。

```
if (process.geteuid) {
    console.log(`Current uid: ${process.geteuid()}`);
}
```

process.setegid(id)

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

设置有效的进程组标识（见 setegid(2)）。它可以接收数字 ID 或一个组名称的字符串。如果指定了组名称，该方法会阻塞运行直到将它解析为一个数字 ID 为止。

```
if (process.getegid && process.setegid) {
    console.log(`Current gid: ${process.getegid()}`);
    try {
        process.setegid(501);
        console.log(`New gid: ${process.getegid()}`);
    } catch (err) {
        console.log(`Failed to set gid: ${err}`);
    }
}
```

process.getegid()

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

获取有效的进程组标识（见 getegid(2)）。这是数字的组 id，而非组名称。

```
if (process.getegid) {
    console.log(`Current gid: ${process.getegid()}`);
}
```

process.setgroups(groups)

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

设置补充群组的 ID。这是一个特权操作，意味着你需要使用 root 账户或拥有 `CAP_SETGID` 能力。

该列表参数可以包含组 ID、组名称或者混在一起。

process.getgroups()

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

返回补充群组的 ID 数组。如果有效的组 ID 在 POSIX 上未指定，但在 Node.js 中会确保它永远都是包含在内的。（POSIX leaves it unspecified if the effective group ID is included but Node.js ensures it always is. 翻译君表示已阵亡...）

process.initgroups(user, extra_group)

注意：该函数仅在 POSIX 平台（如，非 Windows 和 Android）上有效。

读取 /etc/group 并通过该用户所在的所有分组初始化组（group）访问列表。这是一个特权操作，意味着你需要使用 root 账户或拥有 `CAP_SETGID` 能力。

`user` 是一个用户名或用户 ID。`extra_group` 是一个群组名称或群组 ID。

当你在注销权限时有时需要注意。例如：

```
console.log(process.getgroups());         // [ 0 ]
process.initgroups('bnoordhuis', 1000);   // switch user
console.log(process.getgroups());         // [ 27, 30, 46, 1000, 0 ]
process.setgid(1000);                     // drop root gid
console.log(process.getgroups());         // [ 27, 30, 46, 1000 ]
```

信号事件

当进程接收到信号时触发。标准的 POSIX 信号名称（如，`SIGINT`、`SIGHUP` 等）列表详见（sigaction(2)）。

监听 `SIGINT` 的例子：

```
// Start reading from stdin so we don't exit.
process.stdin.resume();

process.on('SIGINT', () => {
    console.log('Got SIGINT.  Press Control-D to exit.');
});
```

发送 `SIGINT` 信号最简单的方式是在大多数终端程序中使用 `Control-C`。

注意：

`SIGUSR1` 由 Node.js 保留，用以开启调试器。它也可以设置一个监听器但不会中断调试器的启动。

`SIGTERM` 和 `SIGINT` 是非 Windows 平台上在以代码 `128 + signal number` 退出前重置终端模式的默认处理信号。如果监听到其中一个信号被设置，它默认的行为都会被移除（Node.js 将不复存在）。

`SIGPIPE` 默认被忽略。它可以设置监听器。

`SIGHUP` 在 Windows 上，是在控制台窗口关闭时产生，在其他平台上也是在各种类似的条件下产生（详见，signal(7)）。它可以设置监听器，然而 Node.js 约10秒后会被 Windows 无条件终止。在非 Windows 平台上，`SIGHUP` 的默认行为是终止 Node.js ，但一旦设置了监听器，它的默认行为会被移除。

`SIGTERM` 在 Windows 是不被支持，它可以被监听的。

`SIGINT` 在所有平台的终端上都支持，并且通常是由 `CTRL+C`（虽然这也许配置的）产生的。当启用了终端原始模式时，它不再产生。

`SIGBREAK` 在 Windows 平台上，当按下 `CTRL+BREAK` 时发出，在非 Windows 平台上，它是可以被设置的，但没有办法发送或生成它。

`SIGWINCH` 在控制台已经调整后发出。在 Windows 中，当光标被移动时，或者当一个可读的 tty 在原始模式下使用时，这仅会在写入到控制台时发生。

`SIGKILL` 不能设置监听器，它在所有平台上的 Node.js 中会无条件终止。

`SIGSTOP` 不能设置监听器。

请注意，Windows 不支持发送信号，但 Node.js 提供了一些仿真方法 `process.kill()` 和 `child_process.kill()` 。发送信号 `0` 可用于测试一个进程是否存在。发送 `SIGINT` 、`SIGTERM` 和 `SIGKILL` 会导致目标进程无条件终止。

退出码

当所有的异步操作都已经完成时，Node.js 会以 `0` 状态码正常退出。以下状态码在其他情况下使用：

- `1` 未捕获的致命异常 - 这是一个未捕获的异常，并且它不是由域或一个 `'uncaughtException'` 事件处理器处理的。
- `2` 未使用（由 Bash 保留用于固有的误用）
- `3` 内置的 **JavaScript** 解析错误 - 内置的 JavaScript 源代码在 Node.js 的引导过程中导致了解析错误。这是极为罕见的，一般只会在 Node.js 的自身的发展过程中才会发生。
- `4` 内置的 **JavaScript** 评估失败 - 内置的 JavaScript 源代码在 Node.js 的引导过程的评估失败时，返回一个函数值。这是极为罕见的，一般只会在 Node.js 的自身的发展过程中才会发生。
- `5` 致命错误 - 在 V8 中有一个致命的不可恢复的错误。通常会在 stderr 中打印一条前缀为 `FATAL ERROR` 的错误。
- `6` 非函数内部的异常处理程序 - 有一个未捕获的异常，但内部致命异常处理函数在某种程度上被设置为一个非函数，并且不能被调用。
- `7` 内部异常处理程序运行时失败 - 有一个未捕获的异常，但在试图处理它内部的致命异常时，处理函数自身抛出一个错误。这是可能发生的，例如，如果一个 `process.on('uncaughtException')` 或 `domain.on('error')` 处理程序抛出一个错误。
- `8` 未使用。在 Node.js 的早期版本中，退出码8，有时表示未捕获的异常。
- `9` 无效参数 - 无论是指定了一个未知的选项，还是一个需要值的选项没有提供值。
- `10` 内置的 **JavaScript** 运行时失败 - 当引导函数被调用时，内置的 JavaScript 源代码在 Node.js 的引导过程中抛出一个错误。这是极为罕见的，一般只会在 Node.js 的自身的发展过程中才会发生。
- `12` 无效的调试参数 - 设置了 `--debug` 和/或 `--debug-brk` 选项，但选中了一个无效的端口号。
- `>128` 信号退出 - 如果 Node.js 收到了像 `SIGKILL` 或 `SIGHUP` 这样的致命信号，那么它的退出码会是 `128` 以上的信号代码值。这是一个标准的 Unix 的做法，由于退出码被定义为 7位的整数，并将退出码设置为高阶位，然后包含所述信号代码的值。

子进程(Child Processes)

> 稳定度：2 - 稳定

子进程模块提供了衍生子进程的能力，这个能力和 popen(3) 方式上类似，但不完全相同。这种能力主要由 `child_process.spawn()` 函数提供：

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
    console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
    console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
    console.log(`child process exited with code ${code}`);
});
```

默认情况下，在 Node.js 的父进程和衍生的子进程之间会建立 `stdin` 、`stdout` 和 `stderr` 的管道。这也使得数据流可以以无阻塞的方式通过这些管道。但是请注意，有些程序内部使用行缓冲（*line-buffered*）*I/O*。虽然这并不影响 *Node.js*，它可能意味着发送到子过程数据可能无法立即消费。

`child_process.spawn()` 方法异步衍生子进程，不会阻塞 Node.js 的事件循环。`child_process.spawnSync()` 函数以同步的方式提供了同样的功能，它会阻塞事件循环，直到衍生的子进程退出或终止。

为了方便起见，`child_process` 模块提供了少有的同步和异步的替代品 child_process.spawn() 和 child_process.spawnSync() 。请注意，这些替代品是在 *`child_process.spawn()`* 或 *`child_process.spawnSync()`* 的基础上实现的。

- `child_process.exec()` ：衍生一个 shell 并在 shell 内部运行一个命令，当完成时，会向回调函数传递 `stdout` 和 `stderr` 。
- `child_process.execFile()` ：和 `child_process.exec()` 类似，除了它直接衍生命令，而不需要先衍生一个 shell。
- `child_process.fork()` ：衍生一个新的 Node.js 进程，并且通过建立一个允许父进程和子进程之间相互发送信息的 IPC 通讯通道来调用指定的模块。

- `child_process.execSync()`：`child_process.exec()` 的一个同步版本，这会阻塞 Node.js 的事件循环。
- `child_process.execFileSync()`：`child_process.execFile()` 的一个同步版本，这会阻塞 Node.js 的事件循环。

对于某些使用情况，如自动化 shell 脚本，同步版本可能更方便。在多数情况下，同步的方法会显著地影响性能，因为它拖延了事件循环直到衍生进程完成。

ChildProcess类

`ChildProcess` 类是代表衍生子进程的 EventEmitters 的实例。

不打算直接创建 `ChildProcess` 的实例。相反，要使用 child_process.spawn()、child_process.exec()、 child_process.execFile() 或 child_process.fork() 方法创建 `ChildProcess` 实例。

'message' 事件

- `message` {Object} 一个已解析的 JSON 对象或原始值。
- `sendHandle` {Handle} 一个 net.Socket 或 net.Server 对象，或是 `undefined`。

当一个子进程使用 `process.send()` 发送信息时会触发 `'message'` 事件。

'disconnect' 事件

在父进程或子进程中调用 `ChildProcess.disconnect()` 后触发 `'disconnect'` 事件。在断开后它将不能再发送或接收信息，并且 `ChildProcess.connected` 属性会被设为 `false`。

'close' 事件

- `code` {Number} 如果子进程退出自身，该值会是退出码。
- `signal` {String} 子进程被终止时的信号。

当子进程的 stdio 流被关闭时触发 `'close'` 事件。这与 `'exit'` 事件不同，因为多个进程可能共享相同的 stdio 流。

'exit' 事件

- `code` {Number} 如果子进程退出自身，该值会是退出码。
- `signal` {String} 子进程被终止时的信号。

当子进程结束时触发 `'exit'` 事件。如果进程退出，`code` 会是进程的最终退出码，否则会是 `null`。如果进程是由于收到的信号而终止的，`signal` 会是信号的字符串名称，否则会是 `null`。这两个将总有一个是非空的。

注意，当 `'exit'` 事件触发时，子进程的 stdio 流仍可能打开着。

另外，还要注意，Node.js 建立了 `SIGINT` 和 `SIGTERM` 的信号处理程序，并且 Node.js 进程因为收到这些信号，所以不会立即终止。相反，Node.js 将执行一个清理序列的操作然后重新引发处理信号。

详见 `waitpid(2)`。

'error' 事件

- `err` {Error} 该错误对象。

每当出现以下情况时触发 `'error'` 事件：

1. 该进程无法被衍生时；
2. 该进程无法被杀死时；
3. 向子进程发送信息失败时。

请注意，在一个错误发生后，`'exit'` 事件可能会也可能不会触发。如果你同时监听了 `'exit'` 和 `'error'` 事件，谨防处理函数被多次调用。

请同时参阅 ChildProcess#kill() 和 ChildProcess#send()。

child.pid

- {Number} 整数

返回子进程的进程 id（PID）。

```
const spawn = require('child_process').spawn;
const grep = spawn('grep', ['ssh']);

console.log(`Spawned child pid: ${grep.pid}`);
grep.stdin.end();
```

child.connected

- {Boolean} 在调用 `.disconnect` 后被设为 `false`。

`child.connected` 属性指示是否仍可以从一个子进程发送和接收消息。当 `child.connected` 是 `false` 时，它将不再能够发送或接收的消息。

child.stdio

- {Array}

一个到子进程的管道的稀疏数组，对应着传给 child_process.spawn() 的选项中值被设为 `'pipe'` 的 stdio。请注意，与 `child.stdin`、`child.stdout` 和 `child.stderr` 分别对应的 `child.stdio[0]`、`child.stdio[1]` 和 `child.stdio[2]` 同样有效。

在下面的例子中，只有子进程的 fd `1`（stdout）被配置为 pipe，所以只有该父进程的 `child.stdio[1]` 是一个流，且在数组中的其他值都是 `null`。

```
const assert = require('assert');
const fs = require('fs');
const child_process = require('child_process');

const child = child_process.spawn('ls', {
    stdio: [
      0, // Use parents stdin for child
      'pipe', // Pipe child's stdout to parent
      fs.openSync('err.out', 'w') // Direct child's stderr to a file
    ]
});

assert.equal(child.stdio[0], null);
assert.equal(child.stdio[0], child.stdin);

assert(child.stdout);
assert.equal(child.stdio[1], child.stdout);

assert.equal(child.stdio[2], null);
assert.equal(child.stdio[2], child.stderr);
```

child.stdin

- {Stream}

一个代表子进程的 `stdin` 的 `Writable Stream`。

注意，如果一个子进程等待读取所有的输入，子进程直到该流在通过 *`end()`* 关闭前不会继续。

如果衍生的子进程的 `stdio[0]` 设置任何不是 `'pipe'` 的值，那么这会是 `undefined`。

`child.stdin` 是 `child.stdio[0]` 的一个别名。这两个属性都会指向相同的值。

child.stdout

- {Stream}

一个代表子进程的 `stdout` 的 `Readable Stream`。

如果衍生的子进程的 `stdio[1]` 设置任何不是 `'pipe'` 的值，那么这会是 `undefined`。

`child.stdout` 是 `child.stdio[1]` 的一个别名。这两个属性都会指向相同的值。

child.stderr

- {Stream}

一个代表子进程的 `stderr` 的 `Readable Stream`。

如果衍生的子进程的 `stdio[2]` 设置任何不是 `'pipe'` 的值，那么这会是 `undefined`。

`child.stderr` 是 `child.stdio[2]` 的一个别名。这两个属性都会指向相同的值。

child.send(message[, sendHandle[, options]][, callback])

- `message` {Object}
- `sendHandle` {Handle}
- `options` {Object}
- `callback` {Function}
- 返回：{Boolean}

当在父进程和子进程之间建立了一个 IPC 通道时（例如，当使用 child_process.fork() 时），该 `child.send()` 方法可以用于将消息发送到子进程。当该进程是一个 Node.js 实例时，该信息可以通过 `process.on('message')` 事件接收到。

例如，在父脚本：

```
const cp = require('child_process');
const n = cp.fork(`${__dirname}/sub.js`);

n.on('message', (m) => {
    console.log('PARENT got message:', m);
});

n.send({
    hello: 'world'
});
```

然后是子进程脚本，`'sub.js'` 可能看上去像这样：

```
process.on('message', (m) => {
    console.log('CHILD got message:', m);
});

process.send({
    foo: 'bar'
});
```

Node.js 中的子进程有自己的一个 `process.send()` 方法，该方法允许子进程将信息发送回父进程。

当发送的是 `{cmd: 'NODE_foo'}` 消息时，则是一个特例。所有在 `cmd` 属性里包含前缀为 `NODE_` 的属性被认为是预留给 Node.js 核心代码内部使用的，并且不会触发子进程的 `process.on('message')` 事件。相反，这种消息被用于触发 `process.on('internalMessage')` 事件，并且被 Node.js 内部消费。应用程序应避免使用这样的消息或监听 `'internalMessage'` 事件作为不通知该变化的依据。

可选的 `sendHandle` 参数可能被传给 `child.send()`，它是用于将一个 TCP 服务器或 socket 对象传给子进程。子进程会将收到的这个对象作为第二个参数传给注册在 `process.on('message')` 事件上的回调函数。

该 `options` 参数，如果存在的话，是用于发送某些类型的处理程序的参数对象。`options` 支持以下属性：

- `keepOpen` - 当传 `net.Socket` 时的一个可使用的 Boolean 值。当它为 `true` 时，socket 在发送进程中保持开启状态。默认为 `false`。

可选的 `callback` 是一个函数，它在信息发送后，但在子进程可能收到信息之前被调用。该函数被调用后只有一个参数：成功时是 `null`，或在失败时是一个 Error 对象。

如果没有提供 `callback` 函数，并且信息没被发送，一个 `'error'` 事件将被 `ChildProcess` 对象触发。这是有可能发生的，例如，当子进程已经退出时。

如果该通道已关闭或当未发送的信息积压超过阈值，使得它无法发送更多时，`child.send()` 将会返回 `false`。除此以外，该方法返回 `true`。该 `callback` 函数可用于实现流量控制。

例子：发送服务器对象

例如，`sendHandle` 参数可以用于将一个 TCP 服务器的处理程序传递给子进程，如下所示：

```
const child = require('child_process').fork('child.js');

// Open up the server object and send the handle.
const server = require('net').createServer();
server.on('connection', (socket) => {
    socket.end('handled by parent');
});
server.listen(1337, () => {
    child.send('server', server);
});
```

那么子进程将会得到该服务器对象：

```
process.on('message', (m, server) => {
    if (m === 'server') {
        server.on('connection', (socket) => {
            socket.end('handled by child');
        });
    }
});
```

一旦服务器现在是在父进程和子进程之间共享，那么一些连接可以由父进程处理，一些可以由子进程来处理。

虽然上面的例子使用了 `net` 模块创建了一个服务器，使用 `dgram` 模块创建的服务器使用完全相同的工作流程，不同的是它监听一个 `'message'` 事件而不是 `'connection'` 事件并使用 `server.bind` 替代 `server.listen`。但是，目前仅 UNIX 平台支持这一点。

例子：发送 **socket** 对象

同样，`sendHandle` 参数可以用于将一个 socket 处理程序传递给子进程。以下的例子衍生了两个子进程分别用于处理 "normal" 连接或优先处理 "special" 连接：

```
const normal = require('child_process').fork('child.js', ['normal']);
const special = require('child_process').fork('child.js', ['special']);

// Open up the server and send sockets to child
const server = require('net').createServer();
server.on('connection', (socket) => {

    // If this is special priority
    if (socket.remoteAddress === '74.125.127.100') {
        special.send('socket', socket);
        return;
    }
    // This is normal priority
    normal.send('socket', socket);
});
server.listen(1337);
```

该 `child.js` 会收到一个 socket 处理器作为第二个参数传递给事件回调函数：

```
process.on('message', (m, socket) => {
    if (m === 'socket') {
        socket.end(`Request handled with ${process.argv[2]} priority`);
    }
});
```

一旦一个 socket 被传递给了子进程，父进程不再能够跟踪套接字何时被销毁。为了说明这一点，`.connections` 属性会变成 `null`。当发生这种情况时，建议不要使用 `.maxConnections`。

注意，该函数内部使用 *JSON.stringify()* 序列化 `message`。

child.disconnect()

关闭父进程与子进程之间的 IPC 通道，一旦没有其他的连接使保持它活着，将允许子进程正常退出。在调用该方法后，分别在父进程和子进程上的 `child.connected` 和 `process.connected` 属性都会被设置为 `false`，并且它将不再能够在进程之间传递消息。

当正在接收的进程中没有消息时，将会触发 `'disconnect'` 事件。这经常在调用 `child.disconnect()` 后立即被触发。

请注意，当子进程是一个 Node.js 实例（例如，使用 child_process.fork() 衍生）时，最好在子进程内部调用 `process.disconnect()` 方法来关闭 IPC 通道。

child.kill([signal])

- `signal` {String}

`child.kill()` 方法向子进程发送一个信号。如果没有给定参数，该进程将会发送 `'SIGTERM'` 信号。可以在 `signal(7)` 中查阅可用的信号列表。

```
const spawn = require('child_process').spawn;
const grep = spawn('grep', ['ssh']);

grep.on('close', (code, signal) => {
    console.log(`child process terminated due to receipt of signal ${signal}`);
});

// Send SIGHUP to process
grep.kill('SIGHUP');
```

如果信号没有被送达，`ChildProcess` 对象可能会触发一个 `'error'` 事件。向一个已经退出的进程发送一个信号不是一个错误，但可能有无法预知的后果。特别是如果该进程的 id 已经被其他程序注册时，信号会被发送到该进程，反而它可能会有意想不到的结果。

请注意，当函数被调用 `kill` 时，发送到子进程处理的信号实际上可能没有终止该进程。

见 `kill(2)`，以供参考。

还要注意：当试图杀死他们的父进程时，子进程的子进程不会被终止。这可能发生在当在一个 shell 中运行一个新进程时或使用 `ChildProcess` 中的 `shell` 选项时，如本示例所示：

```
'use strict';
const spawn = require('child_process').spawn;

let child = spawn('sh', ['-c',
  `node -e "setInterval(() => {
      console.log(process.pid + 'is alive')
    }, 500);"`
  ], {
    stdio: ['inherit', 'inherit', 'inherit']
});

setTimeout(() => {
    child.kill(); // does not terminate the node process in the shell
}, 2000);
```

创建异步进程

---

child_process.exec(command[, options][, callback])

- `command` {String} 要运行的命令，用空格分隔参数
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `env` {Object} 环境变量键值对
  - `encoding` {String} （默认：'utf8'）
  - `shell` {String} 用于执行命令的 shell（默认：在 UNIX 上为 '/bin/sh'，在 Windows 上为 'cmd.exe'。该 shell 应该能够在 UNIX 上以 `-c` 启动或在 Windows 上以 `/s /c` 启动。在 Windows 中，命令行的解析应与 `cmd.exe` 兼容。）
  - `timeout` {Number} （默认：0）
  - `maxBuffer` {Number} 在 stdout 或 stderr 中所允许的最大数据量（以字节为单位）- 如果超过了该限制，子进程将被终止。（默认：`200*1024`）
  - `killSignal` {String} （默认：'SIGTERM'）
  - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）
  - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）
- `callback` {Function} 在进程终止时与输出一起调用
  - `error` {Error}

    - `stdout` {String} | {Buffer}
    - `stderr` {String} | {Buffer}
- 返回：{ChildProcess}

衍生一个 shell，然后在该 shell 中执行该 `command`，缓冲任何产生的输出。

```
const exec = require('child_process').exec;
const child = exec('cat *.js bad_file | wc -l', (error, stdout, stderr) => {
    console.log(`stdout: ${stdout}`);
    console.log(`stderr: ${stderr}`);
    if (error !== null) {
        console.log(`exec error: ${error}`);
    }
});
```

如果提供了一个 `callback` 函数，它以 `(error, stdout, stderr)` 的参数形式被调用。成功的话，`error` 会是 `null`。失败的话，`error` 会是一个 Error 实例。`error.code` 属性会是进程的退出码，而 `error.signal` 会被设置为终止进程的信号。除 `0` 以外的任何退出码都被认为是一个错误。

传给回调函数的 `stdout` 和 `stderr` 参数会包含子进程的 stdout 和 stderr 的输出结果。默认情况下，Node.js 会将输出解码为 UTF-8 并将结果字符串传递给回调函数。`encoding` 选项可用于指定用于解码 stdout 和 stderr 的字符编码。如果 `encoding` 是 `'buffer'`，`Buffer` 对象将代替传递给回调函数。

`options` 参数可以以第二个参数的形式传递，用于自定义如何衍生进程。默认的选项是：

```
{
    encoding: 'utf8',
    timeout: 0,
    maxBuffer: 200 * 1024,
    killSignal: 'SIGTERM',
    cwd: null,
    env: null
}
```

如果 `timeout` 大于 `0` 并且子进程运行超过 `timeout` 毫秒，父进程将会发送由 `killSignal` 属性标识的信号（默认为 `'SIGTERM'`）。

注意：不像 *POSIX* 系统调用中的 `exec()`，`child_process.exec()` 不会替换现有的进程和使用一个 *shell* 来执行命令。

child_process.execFile(file[, args][, options][, callback])

- `file` {String} 需要运行的可执行文件的名称或路径
- `args` {Array} 字符串参数列表
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `env` {Object} 环境变量键值对
  - `encoding` {String} （默认：'utf8'）
  - `timeout` {Number} （默认：0）
  - `maxBuffer` {Number} 在 stdout 或 stderr 中所允许的最大数据量（以字节为单位）- 如果超过了该限制，子进程将被终止。（默认：`200*1024`）
  - `killSignal` {String} （默认：'SIGTERM'）
  - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）
  - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）
- `callback` {Function} 在进程终止时与输出一起调用
  - `error` {Error}
  - `stdout` {String} | {Buffer}
  - `stderr` {String} | {Buffer}
- 返回：{ChildProcess}

`child_process.execFile()` 函数类似于 child_process.exec()，它们的不同之处在于，前者不需要衍生一个 shell。而被指定的可执行文件直接衍生为一个新进程，也使得它比 child_process.exec() 更有效一些。

它支持和 `child_process.exec()` 一样的选项。由于没有衍生 shell，因此不支持像 I/O 重定向和文件查找这样的行为。

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (error, stdout, stderr) => {
    if (error) {
        throw error;
    }
    console.log(stdout);
});
```

传给回调函数的 `stdout` 和 `stderr` 参数会包含子进程的 stdout 和 stderr 的输出结果。默认情况下，Node.js 会将输出解码为 UTF-8 并将结果字符串传递给回调函数。`encoding` 选项可用于指定用于解码 stdout 和 stderr 的字符编码。如果 `encoding` 是 `'buffer'`，`Buffer` 对象将代替传递给回调函数。

child_process.spawn(command[, args][, options])

- `command` {String} 需要运行的命令
- `args` {Array} 字符串参数列表
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `env` {Object} 环境变量键值对
  - `stdio` {Array} | {String} 子进程的工作配置。（详见 options.stdio）
  - `detached` {Boolean} 准备将子进程独立于父进程运行。它的具体行为取决于平台，详见（options.detached）
  - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）
  - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）
  - `shell` {Boolean} | {String} 如果为 `true`，在一个 shell 中运行 `command`。在 UNIX 上运行 '/bin/sh'，在 Windows 上运行 'cmd.exe'。一个不同的 shell 可以被指定为字符串。该 shell 应该能够在 UNIX 上以 `-c` 启动或在 Windows 上以 `/s /c` 启动。默认为 `false`（没有 shell）。
- 返回：{ChildProcess}

`child_process.spawn()` 方法使用给定的 `command` 带着 `args` 中的命令行参数来衍生一个新进程。如果省略，`args` 默认为一个空数组。

第三个参数可以用来指定其他选项，这些的默认值如下：

```
{
    cwd: undefined,
    env: process.env
}
```

使用 `cwd` 来指定衍生进程的工作目录。如果没有给出，它默认是继承当前的工作目录。

使用 `env` 来指定环境变量，这将在新进程中可见，它默认为 `process.env` 。

运行 `ls -lh /usr` 的例子，捕捉它的 `stdout` 、 `stderr` 以及退出码：

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
    console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
    console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
    console.log(`child process exited with code ${code}`);
});
```

例子：一种执行 `'ps ax | grep ssh'` 的复杂方式

```
const spawn = require('child_process').spawn;
const ps = spawn('ps', ['ax']);
const grep = spawn('grep', ['ssh']);

ps.stdout.on('data', (data) => {
    grep.stdin.write(data);
});

ps.stderr.on('data', (data) => {
    console.log(`ps stderr: ${data}`);
});

ps.on('close', (code) => {
    if (code !== 0) {
        console.log(`ps process exited with code ${code}`);
    }
    grep.stdin.end();
});

grep.stdout.on('data', (data) => {
    console.log(`${data}`);
});

grep.stderr.on('data', (data) => {
    console.log(`grep stderr: ${data}`);
});

grep.on('close', (code) => {
    if (code !== 0) {
        console.log(`grep process exited with code ${code}`);
    }
});
```

检测执行失败的例子：

```
const spawn = require('child_process').spawn;
const child = spawn('bad_command');

child.on('error', (err) => {
    console.log('Failed to start child process.');
});
```

options.stdio

`options.stdio` 选项用于配置子进程与父进程之间建立的管道。默认情况下，子进程的 stdin、 stdout 和 stderr 重定向到 `ChildProcess` 对象上相应的 `child.stdin`、 `child.stdout` 和 `child.stderr` 流。这等同于将 `options.stdio` 设置为 `['pipe', 'pipe', 'pipe']` 。

为了方便起见，`options.stdio` 可以是下列字符串之一：

- `'pipe'` - 等同于 `['pipe', 'pipe', 'pipe']`（默认）
- `'ignore'` - 等同于 `['ignore', 'ignore', 'ignore']`
- `'inherit'` - 等同于 `[process.stdin, process.stdout, process.stderr]` 或 `[0,1,2]`

否则，`option.stdio` 的值是一个每个索引都对应子进程中的一个 fd 的数组。fd 中的 0 、 1 和 2 分别对应 stdin、 stdout 和 stderr。额外的 fd 可以被指定创建父进程和子进程之间的附加管道。该值是下列之一：

1. `'pipe'` - 创建一个子进程和父进程之间的管道。在管道的父端以 `ChildProcess.stdio[fd]` 的形式将父进程作为 `child_process` 对象上的一个属性暴露给外界。为 fd 创建的管道 0-2 分别替换为 `ChildProcess.stdin` 、 `ChildProcess.stdout` 和 `ChildProcess.stderr` 也同样有效。
2. `'ipc'` - 创建一个父进程和子进程之间的 IPC 信道用以传递消息和文件描述符。一个 Child_Process 可能最多只能有一个 IPC 标准输入输出文件描述符。设置该选项可启用 `ChildProcess.send()` 方法。如果子进程写了 JSON 信息到此文件描述符， `ChildProcess.on('message')` 事件处理器会被父进程触发。如果子进程是一个 Node.js 进程，一个已存在的 IPC 信道将可以在子进程中使用 `process.send()` 、 `process.disconnect()` 、 `process.on('disconnect')` 和 `process.on('message')` 。
3. `'ignore'` - 指示 Node.js 在子进程中忽略 fd 。由于 Node.js 总是会为它衍生的进程打开 fds 0-2 ，将 fd 设置为 `'ignore'` 会引起 Node.js 打开 `/dev/null` 并将它附加到子进程的 fd 上。
4. `Stream` 对象 - 共享一个指向子进程的 tty、文件、socket 或管道的可读或可写流。流的底层文件描述符在子进程中重复着到对应该 `stdio` 数组的索引的 fd 。需要注意的是，该流必须要有一个底层描述符（文件流在发生 `'open'` 事件前不需要）。
5. 正整数 - 整数值被解释为正在父进程中打开的文件描述符。它和子进程共享，类似于 `Stream` 是如何被共享的。
6. `null` 、 `undefined` - 使用默认值。对于 stdio fds 0、1 和 2（换句话说，stdin、stdout 和 stderr）而言是创建了一个管道。对于 fd 3 及以上而言，它的默认值为 `'ignore'` 。

例子：

```
const spawn = require('child_process').spawn;

// Child will use parent's stdios
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs presenting a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

值得注意的是，当在父进程和子进程之间建立了一个 *IPC* 信道，并且子进程是一个 *Node.js* 进程，子进程的启动未引用（使用 `unref()`）该 *IPC* 信道，直到子进程为 `process.on('disconnected')` 事件注册了一个事件处理器。这运行子进程在没有进程的情况下正常退出通过开放的 *IPC* 信道保持开放状态。

也可以看看：child_process.exec() 和 child_process.fork() 。

options.detached

在 Windows 上，将 `options.detached` 设置为 `true`，使得子进程在父进程退出后继续运行成为可能。子进程将拥有自己的控制台窗口。一旦启用一个子进程，它将不能被禁用。

在非 Windows 平台上，如果将 `options.detached` 设置为 `true`，那么子进程将成为新的进程组和会话的领导者。请注意，子进程在父进程退出后可以继续运行，不管它们是否被分离。更多信息详见 `setsid(2)`。

默认情况下，父进程会等子进程被分离后才退出。为了防止父进程等待给定的 `child`，请使用 `child.unref()` 方法。这样做会导致父进程的事件循环不包括子进程的引用计数，这将允许父进程独立子进程退出，除非子进程和父进程之间建立了一个 IPC 信道。

当使用 `detached` 选项来启动一个长期运行的进程时，该进程不会在父进程退出后保持后台运行状态，除非它提供了一个不连接到父进程的 `stdio` 配置。如果该父进程的 `stdio` 被继承时，子进程会保持连接到控制终端。

一个长期运行的进程的例子，为了无视父母的终止，通过分离并且同时忽略其父进程的 `stdio` 文件描述符来实现：

```
const spawn = require('child_process').spawn;

const child = spawn(process.argv[0], ['child_program.js'], {
    detached: true,
    stdio: ['ignore']
});

child.unref();
```

另外，你可以将子进程的输出重定向到文件：

```
const fs = require('fs');
const spawn = require('child_process').spawn;
const out = fs.openSync('./out.log', 'a');
const err = fs.openSync('./out.log', 'a');

const child = spawn('prg', [], {
    detached: true,
    stdio: ['ignore', out, err]
});

child.unref();
```

child_process.fork(modulePath[, args][, options])

- `modulePath` {String} 在子进程中运行的模块
- `args` {Array} 字符串参数列表
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `env` {Object} 环境变量键值对
  - `execPath` {String} 用来创建子进程的可执行文件
  - `execArgv` {Array} 传递给可执行文件的字符串参数列表（默认：`process.execArgv`）
  - `silent` {Boolean} 如果为 `true`，子进程中的 stdin、stdout 和 stderr 会被导流到父进程中，否则它们会继承自父进程，详见 child_process.spawn() 的 stdio 中的 `'pipe'` 和 `'inherit'` 选项了解更多信息（默认是 `false`）
  - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）

    - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）

- 返回：{ChildProcess}

`child_process.fork()` 方法是 child_process.spawn() 的一种特殊情况，它被专门用于衍生新的 Node.js 进程。像 `child_process.spawn()` 一样，返回一个 `ChildProcess` 对象。返回的 `ChildProcess` 会有一个额外的内置的通信通道，它允许消息在父进程和子进程之间来回传递。查看 ChildProcess#send() 了解跟多细节。

牢记衍生的 Node.js 子进程与两者之间建立的 IPC 通信信道的异常是独立于父进程的，这一点非常重要。每个进程都有它自己的进程，使用自己的 V8 实例。由于需要额外的资源分配，因此不推荐衍生一大批 Node.js 子进程。

默认情况下，`child_process.fork()` 会使用父进程中的 `process.execPath` 衍生新的 Node.js 实例。在 `options` 对象中的 `execPath` 属性允许替代要使用的执行路径。

Node.js 进程使用自定义的 `execPath` 启动会使用子进程的环境变量 `NODE_CHANNEL_FD` 中确定的文件描述符（fd）与父进程通信。fd 上的输入和输出预计被分割成一行一行的 JSON 对象。

注意，不像 POSIX 系统回调中的 `fork()`，child_process.fork() 不会克隆当前进程。

在 Windows 上衍生 .bat 和 .cmd 文件

`child_process.exec()` 和 `child_process.execFile()` 之间最重要的区别是可以基于平台。在类 Unix 的操作平台（Unix、Linux、OSX）上，`child_process.execFile()` 更有效，因为它不需要衍生一个 shell。在 Windows 上，尽管 `.bat` 和 `.cmd` 文件在没有终端的情况下，它们自己不可自执行，因此不能使用 `child_process.execFile()` 启动。当在 Windows 下运行时，通过使用设置的 shell 选项的 `child_process.spawn()`，`child_process.exec()` 或通过将 `.bat` 和 `.cmd` 文件作为一个参数传给 `cmd.exe` 进行衍生（`shell` 选项和 `child_process.exec()` 所做的工作），这些操作可以调用 `.bat` 和 `.cmd` 文件。

```
// On Windows Only ...
const spawn = require('child_process').spawn;
const bat = spawn('cmd.exe', ['/c', 'my.bat']);

bat.stdout.on('data', (data) => {
    console.log(data);
});

bat.stderr.on('data', (data) => {
    console.log(data);
});

bat.on('exit', (code) => {
    console.log(`Child exited with code ${code}`);
});

// OR...
const exec = require('child_process').exec;
exec('my.bat', (err, stdout, stderr) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(stdout);
});
```

创建同步进程

`child_process.spawnSync()`、`child_process.execSync()` 和 `child_process.execFileSync()` 方法是同步的并且会阻塞 Node.js 的事件循环，暂停任何额外代码的执行直到衍生的进程退出为止。

像这样的阻塞调用有利于简化通用脚本任务，并在启动时有利于简化应用配置的加载/处理。

child_process.execSync(command[, options])

- `command` {String}
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `input` {String} | {Buffer} 该值会被作为传递到衍生进程中的标准输入
    - 提供的值将会覆盖 `stdio[0]`
  - `stdio` {Array} 子进程中的 stdio 配置（默认：`'pipe'`）。
    - `stderr` 除非指定了 `stdio`，否则默认会输出到父进程中的 stderr。
  - `env` {Object} 环境变量键值对
  - `shell` {String} 用于执行命令的 shell（默认：在 UNIX 上为 '/bin/sh'，在 Windows 上为 'cmd.exe'。该 shell 应该能够在 UNIX 上以 `-c` 启动或在 Windows 上以 `/s /c` 启动。在 Windows 中，命令行的解析应与 `cmd.exe` 兼容。）
  - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）
  - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）
  - `timeout` {Number} 进程被运行运行的最大时间量，以毫秒为单位（默认：undefined）
  - `killSignal` {String} 当衍生进程将被杀死时要使用的信号值。（默认：'SIGTERM'）

    - `maxBuffer` {Number} 在 stdout 或 stderr 中所允许的最大数据量（以字节为单位）- 如果超过了该限制，子进程将被终止。
    - `encoding` {String} 用于所有 stdio 输入和输出的编码

- 返回：{Buffer} | {String} 该命令的标准输出

`child_process.execSync()` 基本等同于 `child_process.exec()`，除了该方法直到子进程完全关闭前不会返回结果。当遇到超时并发送了 `killSignal` 信号时，该方法直到进程完全退出前不会返回结果。请注意，如果拦截了子进程并且处理了 `SIGTERM` 信号但没有退出，父进程会一直等待直到子进程退出为止。

如果进程超时，或有一个非零退出码，该方法会抛出错误。该 Error 对象包含从 child_process.spawnSync() 获取的整个结果（对象的 Error 属性）。

child_process.execFileSync(file[, args][, options])

- `file` {String} 要运行的可执行文件的名称或路径
- `args` {Array} 字符串参数列表
- `options` {Object}
    - `cwd` {String} 子进程的当前工作目录
    - `input` {String} | {Buffer} 该值会被作为传递到衍生进程中的标准输入
        - 提供的值将会覆盖 `stdio[0]`
    - `stdio` {Array} 子进程中的 stdio 配置（默认：`'pipe'`）。
        - `stderr` 除非指定了 `stdio`，否则默认会输出到父进程中的 stderr。
    - `env` {Object} 环境变量键值对
    - `shell` {String} 用于执行命令的 shell（默认：在 UNIX 上为 '/bin/sh'，在 Windows 上为 'cmd.exe'。该 shell 应该能够在 UNIX 上以 `-c` 启动或在 Windows 上以 `/s /c` 启动。在 Windows 中，命令行的解析应与 `cmd.exe` 兼容。）
    - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）
    - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）
    - `timeout` {Number} 进程被运行运行的最大时间量，以毫秒为单位（默认：undefined）

    - `killSignal` {String} 当衍生进程将被杀死时要使用的信号值。（默认：'SIGTERM'）
    - `maxBuffer` {Number} 在 stdout 或 stderr 中所允许的最大数据量（以字节为单位）- 如果超过了该限制，子进程将被终止。
    - `encoding` {String} 用于所有 stdio 输入和输出的编码
- 返回：{Buffer} | {String} 该命令的标准输出

`child_process.execFileSync()` 基本等同于 `child_process.execFile()`，除了该方法直到子进程完全关闭前不会返回结果。当遇到超时并发送了 `killSignal` 信号时，该方法直到进程完全退出前不会返回结果。请注意，如果拦截了子进程并且处理了 `SIGTERM` 信号但没有退出，父进程会一直等待直到子进程退出为止。

如果进程超时，或有一个非零退出码，该方法会抛出错误。该 Error 对象包含从 child_process.spawnSync() 获取的整个结果（对象的 Error 属性）。

child_process.spawnSync(command[, args][, options])

- `command` {String} 要运行的命令
- `args` {Array} 字符串参数列表
- `options` {Object}
    - `cwd` {String} 子进程的当前工作目录
    - `input` {String} | {Buffer} 该值会被作为传递到衍生进程中的标准输入
        - 提供的值将会覆盖 `stdio[0]`
    - `stdio` {Array} 子进程中的 stdio 配置。
    - `env` {Object} 环境变量键值对
    - `uid` {Number} 设置该进程的用户标识。（详见 setuid(2)）
    - `gid` {Number} 设置该进程的组标识。（详见 setgid(2)）
    - `timeout` {Number} 进程被运行运行的最大时间量，以毫秒为单位（默认：undefined）
    - `killSignal` {String} 当衍生进程将被杀死时要使用的信号值。（默认：'SIGTERM'）

  - `maxBuffer` {Number} 在 stdout 或 stderr 中所允许的最大数据量（以字节为单位）- 如果超过了该限制，子进程将被终止。
  - `encoding` {String} 用于所有 stdio 输入和输出的编码。（默认：'buffer'）
  - `shell` {Boolean} | {String} 如果为 `true`，在一个 shell 中运行 `command`。在 UNIX 上运行 '/bin/sh'，在 Windows 上运行 'cmd.exe'。一个不同的 shell 可以被指定为字符串。该 shell 应该能够在 UNIX 上以 `-c` 启动或在 Windows 上以 `/s /c` 启动。默认为 `false`（没有 shell）。
- 返回：{Object}
  - `pid` {Number} 子进程的 pid
  - `output` {Array} 从 stdio 输出的结果数组
  - `stdout` {Buffer} | {String} `output[1]` 的内容
  - `stderr` {Buffer} | {String} `output[2]` 的内容
  - `status` {Number} 子进程的退出码
  - `signal` {String} 用于杀死子进程的信号
  - `error` {Error} 如果子进程失败或超时产生的错误对象。

`child_process.spawnSync()` 基本等同于 `child_process.spawn()`，除了该方法直到子进程完全关闭前不会返回结果。当遇到超时并发送了 `killSignal` 信号时，该方法直到进程完全退出前不会返回结果。请注意，如果拦截了子进程并且处理了 *`SIGTERM`* 信号但没有退出，父进程会一直等待直到子进程退出为止。

maxBuffer 和 Unicode

`maxBuffer` 选项是在 `stdout` 或 `stderr` 上用于指定允许的最大八位字节数，记住这一点非常重要 —— 如果超过这个值，子进程将会被终止。这尤其影响包含多字节字符编码的输出，如 UTF-8 或 UTF-16。例如，以下将输出 13 个 UTF-8 编码的八位字节到 stdout，尽管只有 4 个字符：

```
console.log('中文测试');
```

路径(Path)

> 稳定度：2 - 稳定

此模块包含用于处理和转换文件路径的工具。其中大多数方法只执行字符串转换。文件系统不会考虑去检查路径是否有效。

通过 `require('path')` 来使用该模块。

方法和属性

属性

- path.delimiter
- path.sep
- path.posix
- path.win32

方法

- path.resolve([from ...], to)
- path.relative(from, to)
- path.format(pathObject)
- path.parse(pathString)
- path.join([path1][, path2][, ...])
- path.normalize(p)
- path.dirname(p)
- path.basename(p[, ext])
- path.extname(p)
- path.isAbsolute(path)

---

path.delimiter

平台特定的路径分隔符。`';'` 或 `':'` 。

在 *nix 上的例子：

```
console.log(process.env.PATH)
// '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env.PATH.split(path.delimiter)
// returns ['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

在 Windows 上的例子：

```
console.log(process.env.PATH)
// 'C:\Windows\system32;C:\Windows;C:\Program Files\node\'

process.env.PATH.split(path.delimiter)
// returns ['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\node\\']
```

path.sep

平台特定的文件分隔符。`'\\'` 或 `'/'`。

在 *nix 上的例子：

```
'foo/bar/baz'.split(path.sep)
// returns ['foo', 'bar', 'baz']
```

在 Windows 上的例子：

```
'foo\\bar\\baz'.split(path.sep)
// returns ['foo', 'bar', 'baz']
```

path.posix

提供访问前面提到的 `path` 方法，但始终以兼容 posix 的方式进行交互。

path.win32

提供访问前面提到的 `path` 方法，但始终以兼容 win32 的方式进行交互。

path.resolve([from ...], to)

将 `to` 解析为绝对路径。

从右到左的顺序，如果 `to` 不是绝对于 `from` 参数，则前置 `from` 参数，直到找到一个绝对路径为止。如果在使用所有的 `from` 路径后仍然没有找到绝对路径，将会使用当前工作目录。结果路径已经规范化，除非路径被解析到了根目录，否则会删除结尾的斜杠。`from` 中的非字符串参数会被忽略。

另一种思路，是把它当成 shell 中的一系列 `cd` 命令。

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

这类似于：

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

不同的是，不同的路径不需要存在，也可以是文件。

示例：

```
path.resolve('/foo/bar', './baz')
// returns '/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/')
// returns '/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
// if currently in /home/myself/node, it returns
// '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

注意：如果 `resolve` 参数有零长度字符串，那么当前的工作目录将被用来替代它们。

path.relative(from, to)

从 `from` 解析相对路径到 `to`。

有时，我们有两个绝对路径，我们需要导出从一个到另一个的相对路径。这实际上是反向转换 `path.resolve`，这意味着我们可以看到：

```
path.resolve(from, path.relative(from, to)) == path.resolve(to)
```

示例：

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb')
// returns '..\\..\\impl\\bbb'

path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb')
// returns '../../impl/bbb'
```

注意：如果 `relative` 参数有零长度字符串，那么当前的工作目录会被用于代替零长度字符串。如果两个路径相同，那么将会返回一个零长度字符串。

path.format(pathObject)

从一个对象中返回一个路径字符串。这与 path.parse 相反。

如果 `pathObject` 拥有 `dir` 和 `base` 属性，返回的字符串会是 `dir` 属性的级联加上依赖于平台的路径分隔符以及 `base` 属性。

如果没有提供 `dir` 属性，那么 `root` 属性将被用于当成 `dir` 属性。然而，它会假定 `root` 属性已经以依赖于平台的路径分隔符结束。在这种情况下，返回的字符串将会是 `root` 属性的级联加上 `base` 属性。

如果都没有提供 `dir` 和 `root` 属性，那么返回的字符串将是 `base` 属性的内容。

如果没有提供 `base` 属性，`name` 属性的级联加上 `ext` 属性会被用于当成 `base` 属性。

示例：

一些 Posix 系统上例子：

```
// If `dir` and `base` are provided, `dir` + platform separator + `base`
// will be returned.
path.format({
    dir: '/home/user/dir',
    base: 'file.txt'
});
// returns '/home/user/dir/file.txt'

// `root` will be used if `dir` is not specified.
// `name` + `ext` will be used if `base` is not specified.
// If only `root` is provided or `dir` is equal to `root` then the
// platform separator will not be included.
path.format({
    root: '/',
    base: 'file.txt'
});
// returns '/file.txt'

path.format({
    dir: '/',
    root: '/',
    name: 'file',
    ext: '.txt'
});
// returns '/file.txt'

// `base` will be returned if `dir` or `root` are not provided.
path.format({
    base: 'file.txt'
});
// returns 'file.txt'
```

一个 Windows 上的例子：

```
path.format({
    root : "C:\\",
    dir : "C:\\path\\dir",
    base : "file.txt",
    ext : ".txt",
    name : "file"
})
// returns 'C:\\path\\dir\\file.txt'
```

path.parse(pathString)

从路径字符串返回一个对象。

一个在 *nix 上的例子：

```
path.parse('/home/user/dir/file.txt')
// returns
// {
//    root : "/",
//    dir : "/home/user/dir",
//    base : "file.txt",
//    ext : ".txt",
//    name : "file"
// }
```

一个在 Windows 上的例子：

```
path.parse('C:\\path\\dir\\index.html')
// returns
// {
//    root : "C:\\",
//    dir : "C:\\path\\dir",
//    base : "index.html",
//    ext : ".html",
//    name : "index"
// }
```

path.join([path1][, path2][, ...])

把所有的参数加到一起，并规范生成的路径。

参数必须是字符串。在 v0.8 版本中，非字符串参数会被默认忽略。在 v0.10 及以上版本中，会抛出一个错误。

示例：

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
// returns '/foo/bar/baz/asdf'

path.join('foo', {}, 'bar')
// throws exception
TypeError: Arguments to path.join must be strings
```

注意：与其他路径模块函数不同，如果 `join` 参数有零长度字符串，它们会被忽略。如果已合并的路径字符串是零长度字符串，那么将会返回 `'.'`，它代表了当前的工作目录。

path.normalize(p)

规范化一个字符串路径，会考虑 `'..'` 和 `'.'` 部分。

当发现多个斜杠时，它们会由单一的斜杠替代；当路径包含尾斜杠时，它会被保存。在 Windows 上使用反斜杠。

示例：

```
path.normalize('/foo/bar//baz/asdf/quux/..')
// returns '/foo/bar/baz/asdf'
```

注意：如果作为参数传递的路径字符串是零长度字符串，那么将会返回 `'.'`，它代表了当前的工作目录。

path.dirname(p)

返回路径的目录名。类似于 Unix 中的 `dirname` 命令。

示例：

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns '/foo/bar/baz/asdf'
```

path.basename(p[, ext])

返回的路径的最后部分。类似于 Unix 中的 `basename` 命令。

示例：

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns 'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns 'quux'
```

path.extname(p)

返回路径的拓展名，在路径的最后部分中的从最后一个 `'.'` 到结尾的字符串。如果路径的最后部分没有 `'.'` 或它的第一个字符是 `'.'`，那么它会返回一个空字符串。

示例：

```
path.extname('index.html')
// returns '.html'

path.extname('index.coffee.md')
// returns '.md'

path.extname('index.')
// returns '.'

path.extname('index')
// returns ''

path.extname('.index')
// returns ''
```

path.isAbsolute(path)

判定 `path` 是否是一个绝对路径。无论工作目录在哪，绝对路径将始终解析到相同的位置。

Posix 上的例子：

```
path.isAbsolute('/foo/bar') // true
path.isAbsolute('/baz/..')  // true
path.isAbsolute('qux/')     // false
path.isAbsolute('.')        // false
```

Windows 上的例子：

```
path.isAbsolute('//server')  // true
path.isAbsolute('C:/foo/..') // true
path.isAbsolute('bar\\baz')  // false
path.isAbsolute('.')         // false
```

注意：如果作为参数传递的路径字符串是零长度字符串，不像其他路径模块函数，它会保持原样，并返回一个 `false` 。

文件系统(File System)

> 稳定度：2 - 稳定

文件 I/O 是由简单封装的标准 POSIX 函数提供。使用该模块使用 `require('fs')`。所有的方法都异步和同步形式。

异步形式始终以完成时的回调作为最后一个参数。传递给完成时的回调的参数取决于方法，但第一个参数总是留给异常。如果操作成功完成，则第一个参数将是 `null` 或 `undefined`。

当使用异步形式时，任何的异常都会被立即抛出。你可以使用 try/catch 来处理异常，或让它们冒泡。

这里是异步版本的例子：

```
const fs = require('fs');

fs.unlink('/tmp/hello', (err) => {
    if (err) throw err;
    console.log('successfully deleted /tmp/hello');
});
```

这里是同步版本的例子：

```
const fs = require('fs');

fs.unlinkSync('/tmp/hello');
console.log('successfully deleted /tmp/hello');
```

异步方法没法保证执行顺序。所以下面例子容易出错：

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
    if (err) throw err;
    console.log('renamed complete');
});
fs.stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
});
```

`fs.stat` 也可以在 `fs.rename` 之前执行。要做到这一点，正确的方法是采用回调链。

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
    if (err) throw err;
    fs.stat('/tmp/world', (err, stats) => {
        if (err) throw err;
        console.log(`stats: ${JSON.stringify(stats)}`);
    });
});
```

在忙碌的进程中，强烈推荐开发者使用这些函数的异步版本。同步版本将会阻止整个进程，直到他们完成——停止所有连接。

可使用文件名的相对路径。但请记住，该路径将相对 `process.cwd()` 。

大多数 fs 函数，让你忽略回调参数。如果你这么做，一个默认的回调将用于抛出错误。为了得到一个到原来的调用点的跟踪，设置 `NODE_DEBUG` 环境变量：

```
$ cat script.js
function bad() {
  require('fs').readFile('/');
}
bad();

$ env NODE_DEBUG=fs node script.js
fs.js:66
        throw err;
              ^
Error: EISDIR, read
    at rethrow (fs.js:61:21)
    at maybeCallback (fs.js:79:42)
    at Object.fs.readFile (fs.js:153:18)
    at bad (/path/to/script.js:2:17)
    at Object.<anonymous> (/path/to/script.js:5:1)
    <etc.>
```

方法和属性

- fs.readdir(path, callback)
- fs.readdirSync(path)
- fs.mkdir(path[, mode], callback)
- fs.mkdirSync(path[, mode])
- fs.rmdir(path, callback)
- fs.rmdirSync(path)
- fs.realpath(path[, cache], callback)
- fs.realpathSync(path[, cache])
- fs.link(srcpath, dstpath, callback)
- fs.linkSync(srcpath, dstpath)
- fs.symlink(target, path[, type], callback)
- fs.symlinkSync(target, path[, type])
- fs.readlink(path, callback)
- fs.readlinkSync(path)
- fs.unlink(path, callback)
- fs.unlinkSync(path)
- fs.lchmod(path, mode, callback)
- fs.lchmodSync(path, mode)
- fs.lchown(path, uid, gid, callback)
- fs.lchownSync(path, uid, gid)
- fs.lstat(path, callback)
- fs.lstatSync(path)
- fs.createReadStream(path[, options])
- fs.read(fd, buffer, offset, length, position, callback)
- fs.readSync(fd, buffer, offset, length, position)
- fs.createWriteStream(path[, options])
- fs.write(fd, data[, position[, encoding]], callback)
- fs.writeSync(fd, data[, position[, encoding]])
- fs.write(fd, buffer, offset, length[, position], callback)
- fs.writeSync(fd, buffer, offset, length[, position])
- fs.truncate(path, len, callback)
- fs.truncateSync(path, len)
- fs.stat(path, callback)
- fs.statSync(path)
- fs.chmod(path, mode, callback)
- fs.chmodSync(path, mode)

- fs.chown(path, uid, gid, callback)
- fs.chownSync(path, uid, gid)
- fs.utimes(path, atime, mtime, callback)
- fs.utimesSync(path, atime, mtime)
- fs.exists(path, callback)
- fs.existsSync(path)
- fs.open(path, flags[, mode], callback)
- fs.openSync(path, flags[, mode])
- fs.close(fd, callback)
- fs.closeSync(fd)
- fs.access(path[, mode], callback)
- fs.accessSync(path[, mode])
- fs.rename(oldPath, newPath, callback)
- fs.renameSync(oldPath, newPath)
- fs.watch(filename[, options][, listener])
  - 注意事项
    - 可用性
    - 索引节点
    - 文件名参数
- fs.watchFile(filename[, options], listener)
- fs.unwatchFile(filename[, listener])
- fs.readFile(file[, options], callback)
- fs.readFileSync(file[, options])
- fs.writeFile(file, data[, options], callback)
- fs.writeFileSync(file, data[, options])
- fs.appendFile(file, data[, options], callback)
- fs.appendFileSync(file, data[, options])
- fs.ftruncate(fd, len, callback)
- fs.ftruncateSync(fd, len)
- fs.fstat(fd, callback)
- fs.fstatSync(fd)
- fs.fchmod(fd, mode, callback)
- fs.fchmodSync(fd, mode)
- fs.fchown(fd, uid, gid, callback)
- fs.fchownSync(fd, uid, gid)
- fs.futimes(fd, atime, mtime, callback)
- fs.futimesSync(fd, atime, mtime)
- fs.fsync(fd, callback)
- fs.fsyncSync(fd)
- fs.fdatasync(fd, callback)

- fs.fdatasyncSync(fd)

---

fs.readdir(path, callback)

异步的 readdir(3)。读取目录的内容。回调带有两个参数 `(err, files)`，`files` 是该目录中不包括 `'.'` 和 `'..'` 的文件名的数组。

fs.readdirSync(path)

同步的 readdir(3)。返回一个不包括 `'.'` 和 `'..'` 的文件名的数组。

fs.mkdir(path[, mode], callback)

异步的 mkdir(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。`mode` 默认为 `0o777`。

fs.mkdirSync(path[, mode])

同步的 mkdir(2)。返回 `undefined`。

fs.rmdir(path, callback)

异步的 rmdir(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.rmdirSync(path)

同步的 rmdir(2)。返回 `undefined`。

fs.realpath(path[, cache], callback)

异步的 realpath(2)。`callback` 有两个参数 `(err, resolvedPath)`。可以使用 `process.cwd` 解析相对路径。`cache` 是一个对象字面映射路径可用于强制一个特定路径的解析或避免 `fs.stat` 对已知的真实路径的额外调用。

例子：

```
var cache = {
    '/etc': '/private/etc'
};
fs.realpath('/etc/passwd', cache, (err, resolvedPath) => {
    if (err) throw err;
    console.log(resolvedPath);
});
```

fs.realpathSync(path[, cache])

同步的 realpath(2)。返回解析的路径。`cache` 是一个对象字面映射路径可用于强制一个特定路径的解析或避免 `fs.stat` 对已知的真实路径的额外调用。

fs.link(srcpath, dstpath, callback)

异步的 link(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.linkSync(srcpath, dstpath)

同步的 link(2)。返回 `undefined`。

fs.symlink(target, path[, type], callback)

异步的 symlink(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。`type` 参数可以设置为 `'dir'`、`'file'` 或 `'junction'`（默认为 `'file'`）并且仅在 Windows 上可用（在其他平台上忽略）。请注意，Windows 交接点需要的目标路径是绝对的。当使用 `'junction'` 时，`target` 参数会被自动归到绝对路径。

这里有以下例子：

```
fs.symlink('./foo', './new-port');
```

它创建了一个名为“new-port”，指向“foo”的一个符号链接。

fs.symlinkSync(target, path[, type])

同步的 symlink(2)。返回 `undefined`。

fs.readlink(path, callback)

异步的 readlink(2)。回调参数有两个参数 `(err, linkString)`。

fs.readlinkSync(path)

同步的 readlink(2)。返回符号链接字符串值。

fs.unlink(path, callback)

异步的 unlink(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.unlinkSync(path)

同步的 unlink(2)。返回 `undefined`。

fs.lchmod(path, mode, callback)

异步的 lchmod(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

仅在 Mac OS X 上有效。

fs.lchmodSync(path, mode)

同步的 lchmod(2)。返回 `undefined`。

fs.lchown(path, uid, gid, callback)

异步的 lchown(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.lchownSync(path, uid, gid)

同步的 lchown(2)。返回 `undefined`。

fs.lstat(path, callback)

异步的 lstat(2)。该回调函数带有两个参数 `(err, stats)`，其中 `stats` 是一个 `fs.Stats` 对象。`lstat()` 等同于 `stat()`，除非 `path` 是一个 symbolic 链接，那么自身就是该链接，它指向的并不是文件。

fs.lstatSync(path)

同步的 lstat(2)。返回一个 `fs.Stats` 实例。

fs.createReadStream(path[, options])

返回一个新的 ReadStream 对象（详见 可读流章节）。

要知道，不同于在一个可读流上设置的 `highWaterMark` 默认值（16 kb），此方法在相同参数下返回的流具有 64 kb 的默认值。

`options` 是一个对象或字符串带有以下默认值：

```
{
    flags: 'r',
    encoding: null,
    fd: null,
    mode: 0o666,
    autoClose: true
}
```

`options` 可以包括 `start` 和 `end` 值，从文件而不是整个文件读取字节范围。`start` 和 `end` 都需要包括在内，并且起始值都是 0。`encoding` 可以是任何可以被 Buffer 接受的值。

如果指定了 `fd`，`ReadStream` 会忽略 `path` 参数并且将使用指定的文件描述符。这也意味着不会触发 `'open'` 事件。请注意，`fd` 应该是阻塞的；非阻塞的 `fd` 们应该传给 net.Socket。

如果 `autoClose` 是 `false`，那么文件描述符将不会被关闭，即使有错误。将其关闭是你的职责，并且需要确保没有文件描述符泄漏。如果 `autoClose` 被设置为 true（默认行为），在 `error` 或 `end` 时，文件描述符将被自动关闭。

`mode` 用于设置文件模式（权限和粘滞位），但仅在创建该文件时有效。

一个例子来读取 100 字节长的文件的最后 10 个字节：

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

如果 `options` 是一个字符串，那么它指定了编码。

fs.read(fd, buffer, offset, length, position, callback)

从 `fd` 指定的文件中读取数据。

`buffer` 是该数据将被写入到的 buffer。

`offset` 是在 buffer 中开始写入的偏移量。

`length` 是一个整数，指定要读取的字节的数目。

`position` 是一个整数，指定从该文件中开始读取的位置。如果 `position` 是 `null`，数据将从当前文件位置读取。

回调给出的三个参数 `(err, bytesRead, buffer)`。

fs.readSync(fd, buffer, offset, length, position)

同步版的 fs.read()。返回 `bytesRead` 的数。

fs.createWriteStream(path[, options])

返回一个新的 WriteStream 对象（详见 可写流章节）。

`options` 是一个对象或字符串带有以下默认值：

```
{
    flags: 'w',
    defaultEncoding: 'utf8',
    fd: null,
    mode: 0o666,
    autoClose: true
}
```

`options` 也可以包括一个 `start` 选项以允许在以前的文件开头部分的位置写入数据。如果是修改文件而不是替换它的话可能需要 `r+` 的 `flags` 模式而不是默认的 `w` 模式。`defaultEncoding` 可以是任何可以被 Buffer 接受的值。

如果 `autoClose` 被设置为 true（默认行为），在文件描述符发生 `error` 或 `end` 事件时，会被自动关闭。如果 `autoClose` 是 false，那么文件描述符将不会被关闭，即使有错误。将其关闭是你的职责，并且需要确保没有文件描述符泄漏。

类似 ReadStream，如果指定了 `fd`，`WriteStream` 会忽略 `path` 参数，并且将使用指定的文件描述符。。这也意味着不会触发 `'open'` 事件。请注意，`fd` 应该是阻塞的；非阻塞的 `fd` 们应该传给 net.Socket。

如果 `options` 是一个字符串，那么它指定了编码。

fs.write(fd, data[, position[, encoding]], callback)

写入 `data` 到由 `fd` 指定的文件。如果 `data` 不是一个 Buffer 实例，那么该值将被强制转换为字符串。

`position` 指向该数据应从哪里写入的文件的开头的偏移量。如果 `typeof position !== 'number'` 该数据将在当前位置被写入。详见 pwrite(2)。

`encoding` 是预期的字符串编码。

该回调将接收 `(err, written, string)` 参数，`written` 用于指定传递过来的字符串需要被写入多少字节。请注意，写入字节与字符串中的字符不同。详见 Buffer.byteLength。

与写入 `buffer` 不同，整个字符串必须写入。无子字符串可以指定。这是因为字节所得到的数据的偏移量可能与字符串的偏移量不同。

需要注意的是使用 `fs.write` 在不等待回调的情况下对同一文件进行多次写入，这是不安全的操作。对于这种情况，我们强烈推荐使用 `fs.createWriteStream` 方法。

在 Linux 上，当文件以追加模式打开时，位置写入不起作用。内核会忽略位置参数，并总是将数据附加到文件的末尾。

fs.writeSync(fd, data[, position[, encoding]])

同步版的 fs.write()。返回写入的字节数。

fs.write(fd, buffer, offset, length[, position], callback)

写入 `buffer` 到由 `fd` 指定的文件。

`offset` 和 `length` 确定该 buffer 被写入的部分。

`position` 指向该数据应从哪里写入的文件的开头的偏移量。如果 `typeof position !== 'number'` 该数据将在当前位置被写入。详见 pwrite(2)。

该回调会给出三个参数 `(err, written, buffer)`，`written` 用于指定从 `buffer` 中写入多少字节。

需要注意的是使用 `fs.write` 在不等待回调的情况下对同一文件进行多次写入，这是不安全的操作。对于这种情况，我们强烈推荐使用 `fs.createWriteStream` 方法。

在 Linux 上，当文件以追加模式打开时，位置写入不起作用。内核会忽略位置参数，并总是将数据附加到文件的末尾。

fs.writeSync(fd, buffer, offset, length[, position])

同步版的 fs.write()。返回写入的字节数。

fs.truncate(path, len, callback)

异步的 truncate(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。文件描述符也可以作为第一个参数传递。在这种情况下，`fs.ftruncate()` 被调用。

fs.truncateSync(path, len)

同步的 truncate(2)。返回 `undefined`。

fs.stat(path, callback)

异步的 stat(2)。回调有两个参数 `(err, stats)`，其中 `stats` 是一个 fs.Stats 对象。详见 fs.Stats 章节了解更多信息。

fs.statSync(path)

同步的 stat(2)。返回一个 fs.Stats 实例。

fs.chmod(path, mode, callback)

异步的 chmod(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.chmodSync(path, mode)

同步的 chmod(2)。返回 `undefined`。

fs.chown(path, uid, gid, callback)

异步的 chown(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.chownSync(path, uid, gid)

同步的 chown(2)。返回 `undefined`。

fs.utimes(path, atime, mtime, callback)

改变由 `path` 提供的路径所引用的文件的文件时间戳。

注意：`atime` 和 `mtime` 参数跟随的相关函数是否遵循以下规则：

- 如果值是一个像 `'123456789'` 那样的可以数值化的字符串，该值会得到转换为对应的数值。
- 如果值是 `NaN` 或 `Infinity`，该值将被转换为 `Date.now()`。

fs.utimesSync(path, atime, mtime)

同步版的 fs.utimes()。返回 `undefined`。

fs.exists(path, callback)

> 稳定度：0 - 已废弃：使用 fs.stat() 或 fs.access() 代替。

通过使用文件系统检查是否存在给定的路径的测试。然后回调给 `callback` 的参数 `true` 或 `false`。例如：

```
fs.exists('/etc/passwd', (exists) => {
    console.log(exists ? 'it\'s there' : 'no passwd!');
});
```

在调用 `fs.open()` 前，`fs.exists()` 不应该被用于检测文件是否存在。这样做会形成一种竞争状态，因为其他进程可能会在两个调用之间改变文件的状态。反而，用户代码应直接调用 `fs.open()` 并处理如果该文件不存在引发的错误。

fs.existsSync(path)

> 稳定度：0 - 已废弃：使用 fs.statSync() 或 fs.accessSync() 代替。

同步版的 fs.exists()。如果文件存在则返回 `true`，否则返回 `false`。

fs.open(path, flags[, mode], callback)

异步打开文件。详见 open(2)。

`flags` 可以是：

- `'r'` - 以读取模式打开文件。如果该文件不存在会发生异常。
- `'r+'` - 以读写模式打开文件。如果该文件不存在会发生异常。
- `'rs'` - 以同步读取模式打开文件。指示操作系统绕过本地文件系统的高速缓存。

  这对 NFS 挂载模式下打开文件很有用，因为它可以让你跳过潜在的陈旧的本地缓存。它会对 I/O 的性能产生明显地影响，所以除非你需要，否则请不要使用此标志。

  请注意，这不会转换 `fs.open()` 进入同步阻塞调用。如果这是你想要的，那么你应该使用 `fs.openSync()`。
- `'rs+'` - 以读写模式打开文件，告知操作系统同步打开。详见 `'rs'` 中的有关的注意点。
- `'w'` - 以写入模式打开文件。将被创建（如果文件不存在）或截断（如果文件存在）。
- `'wx'` - 类似于 `'w'`，但如果 `path` 存在，将会导致失败。
- `'w+'` - 以读写模式打开文件。将被创建（如果文件不存在）或截断（如果文件存在）。
- `'wx+'` - 类似于 `'w+'`，但如果 `path` 存在，将会导致失败。
- `'a'` - 以追加模式打开文件。如果不存在，该文件会被创建。

- `'ax'` - 类似于 `'a'`，但如果 `path` 存在，将会导致失败。
- `'a+'` - 以读取和追加模式打开文件。如果不存在，该文件会被创建。
- `'ax+'` - 类似于 `'a+'`，但如果 `path` 存在，将会导致失败。

`mode` 设置文件模式（权限和粘滞位），但只有当文件被创建时有效。它默认为 `0666`，可读写。

该回调有两个参数 `(err, fd)`。

独家标志 `'x'`（在 open(2) 中的 `O_EXCL` 标志）确保 `path` 是新创建的。在 POSIX 操作系统中，哪怕是一个符号链接到一个不存在的文件，`path` 也会被视为存在。独家标志有可能可以在网络文件系统中工作。

`flags` 也可以是一个记录在 open(2) 中的数字；常用的辅音可从 `require('constants')` 获取。在 Windows 中，标志被转换为它们等同的替代，可适用如 `O_WRONLY` 到 `FILE_GENERIC_WRITE`，或 `O_EXCL|O_CREAT` 到 `CREATE_NEW` 通过 CreateFileW 接受。

在 Linux 中，当文件以追加模式打开时，位置写入不起作用。内核忽略位置参数，并总是将数据附加到文件的末尾。

fs.openSync(path, flags[, mode])

同步版的 fs.open()。返回表示文件描述符的整数。

fs.close(fd, callback)

异步的 close(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.closeSync(fd)

同步的 close(2)。返回 `undefined`。

fs.access(path[, mode], callback)

测试由 `path` 指定的文件的用户权限。`mode` 是一个可选的整数，指定要执行的辅助检查。以下常量定义 `mode` 的可能值。有可能产生由位或和两个或更多个值组成的掩码。

- `fs.F_OK` - 文件对调用进程可见。这在确定文件是否存在时很有用，但只字未提 rwx 权限。如果 `mode` 没被指定，默认为该值。

- `fs.R_OK` - 文件可以通过调用进程读取。
- `fs.W_OK` - 文件可以通过调用进程写入。
- `fs.X_OK` - 文件可以通过调用进程执行。这对 Windows 没有影响（行为类似 `fs.F_OK`）。

最后一个参数 `callback` 是一个回调函数，调用时可能带有一个 error 参数。如果有任何辅助检查失败，错误参数将被填充。下面的例子将检查 `/etc/passwd` 文件是否可以被读取并被当前进程写入。

```
fs.access('/etc/passwd', fs.R_OK | fs.W_OK, (err) => {
    console.log(err ? 'no access!' : 'can read/write');
});
```

fs.accessSync(path[, mode])

同步版的 fs.access()。如果有任何辅助检查失败将抛出错误，否则什么也不做。

fs.rename(oldPath, newPath, callback)

异步的 rename(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.renameSync(oldPath, newPath)

同步的 rename(2)。返回 `undefined`。

fs.watch(filename[, options][, listener])

监视 `filename` 的变化，`filename` 既可以是一个文件也可以是一个目录。返回的对象是一个 fs.FSWatcher 实例。

第二个参数是可选的。如果提供的话，`options` 应该是一个对象。支持的布尔成员是 `persistent` 和 `recursive`。`persistent` 表示当文件被监视时，该进程是否应该持续运行。`recursive` 表示所有子目录是否应该关注，或仅当前目录。只有在支持的平台（参见注意事项）上可以指定目录适用。

默认为 `{ persistent: true, recursive: false }`。

该监听回调获得两个参数 `(event, filename)`。`event` 既可以是 `'rename'` 也可以是 `'change'`，并且 `filename` 是其中触发事件的文件的名称。

注意事项

该 `fs.watch` API 不是 100% 的跨平台一致，并且在某些情况下不可用。

递归选项只支持 OS X 和 Windows。

可用性

此功能依赖于底层操作系统提供的一种方法来通知文件系统的变化。

- 在 Linux 系统中，使用 `inotify`。
- 在 BSD 系统中，使用 `kqueue`。
- 在 OS X 系统中，对文件使用 `kqueue`，对目录使用 `'FSEvents'`。
- 在 SunOS 系统（包括 Solaris 和 SmartOS）中，使用 `event ports`。
- 在 Windows 系统中，此功能取决于 `ReadDirectoryChangesW`。

如果底层功能出于某种原因不可用，那么 `fs.watch` 也将无法正常工作。例如，在网络文件系统（ZFS，SMB 等）中监视文件或目录往往并不可靠或都不可靠。

你可以仍然使用 `fs.watchFile`，它使用状态查询，但它较慢和不可靠。

索引节点

在 Linux 或 OS X 系统中，`fs.watch()` 解析路径到一个索引节点，并监视该索引节点。如果监视的路径被删除或重建，它会被赋予一个新的索引节点。监视器会发出一个删除事件，但会继续监视该原始索引节点。新建索引节点的事件不会被触发。这是正常现象。

文件名参数

在回调中提供 `filename` 参数，仅在 Linux 和 Windows 系统上支持。即使在支持的平台中，`filename` 也不能保证提供。因此，不要以为 `filename` 参数总是在回调中提供，并在它是 null 的情况下，提供一定的后备逻辑。

```
fs.watch('somedir', (event, filename) => {
    console.log(`event is: ${event}`);
    if (filename) {
        console.log(`filename provided: ${filename}`);
    } else {
        console.log('filename not provided');
    }
});
```

fs.watchFile(filename[, options], listener)

监视 `filename` 的变化。`listener` 回调在每次访问文件时会被调用。

`options` 参数可被省略。如果提供的话，它应该是一个对象。`options` 对象可能包含一个名为 `persistent` 的布尔值，表示当文件被监视时，该进程是否应该持续运行。`options` 对象可以指定一个 `interval` 属性，表示目标多久应以毫秒为单位进行查询。该默认值为 `{ persistent: true, interval: 5007 }`。

`listener` 有两个参数，当前的状态对象和以前的状态对象：

```
fs.watchFile('message.text', (curr, prev) => {
    console.log(`the current mtime is: ${curr.mtime}`);
    console.log(`the previous mtime was: ${prev.mtime}`);
});
```

这里的状态对象是 `fs.Stat` 的实例。

如果你想在文件被修改时得到通知，不只是访问，你也需要比较 `curr.mtime` 和 `prev.mtime`。

注意：当 *`fs.watchFile`* 的运行结果属于一个 *`ENOENT`* 错误时，它将以所有字段为零（或日期、*Unix* 纪元）调用监听器一次。在 *Windows* 中，*`blksize`* 和 *`blocks`* 字段会是 *`undefined`*，而不是零。如果文件是在那之后创建的，监听器会被再次以最新的状态对象进行调用。这是在 *v0.10* 版之后在功能上的变化。

注意：*fs.watch()* 比 *`fs.watchFile`* 和 *`fs.watchFile`* 更高效。可能的话，*`fs.watch`* 应当被用来代替 *`fs.watchFile`* 和 *`fs.unwatchFile`*。

fs.unwatchFile(filename[, listener])

停止监视 `filename` 的变化。如果指定了 `listener`，只会移除特定的监视器。否则，所有的监视器都会被移除，并且你已经有效地停止监视 `filename`。

带有一个没有被监视的文件名来调用 `fs.unwatchFile()` 是一个空操作，而不是一个错误。

注意：*fs.watch()* 比 `fs.watchFile` 和 `fs.watchFile` 更高效。可能的话，`fs.watch` 应当被用来代替 `fs.watchFile` 和 `fs.unwatchFile` 。

fs.readFile(file[, options], callback)

- `file` {String} | {Integer} 文件名或文件描述符
- `options` {Object} | {String}
  - `encoding` {String} | {Null} 默认 = `null`
  - `flag` {String} 默认 = `'r'`
- `callback` {Function}

异步读取一个文件的全部内容。例如：

```
fs.readFile('/etc/passwd', (err, data) => {
    if (err) throw err;
    console.log(data);
});
```

该回调传入两个参数 `(err, data)` ， `data` 是文件的内容。

如果没有指定编码，则返回原始 buffer。

如果 `options` 是一个字符串，那么它指定了编码。例如：

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

任何指定的文件描述符必须支持读取。

注意：指定的文件描述符不会被自动关闭。

fs.readFileSync(file[, options])

同步版的 fs.readFile。返回 `file` 的内容。

如果 `encoding` 参数被指定，则此函数返回一个字符串。否则，它返回一个 buffer。

fs.writeFile(file, data[, options], callback)

- `file` {String} | {Integer} 文件名或文件描述符
- `data` {String} | {Buffer}
- `options` {Object} | {String}
  - `encoding` {String} | {Null} 默认 = `'utf8'`
  - `mode` {Number} 默认 = `0o666`
  - `flag` {String} 默认 = `'w'`
- `callback` {Function}

异步将数据写入到文件，如果它已经存在，则替换该文件。`data` 可以是一个字符串或 buffer。

如果 `data` 是一个 buffer，则忽略 `encoding` 选项。它默认为 `'utf8'`。

例子：

```
fs.writeFile('message.txt', 'Hello Node.js', (err) => {
    if (err) throw err;
    console.log('It\'s saved!');
});
```

如果 `options` 是一个字符串，那么它指定了编码。例如：

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

任何指定的文件描述符必须支持写入。

需要注意的是使用 `fs.writeFile` 在不等待回调的情况下对同一文件进行多次写入，这是不安全的操作。对于这种情况，我们强烈推荐使用 `fs.createWriteStream` 方法。

注意：指定的文件描述符不会被自动关闭。

fs.writeFileSync(file, data[, options])

同步版的 fs.writeFile()。返回 `undefined`。

fs.appendFile(file, data[, options], callback)

- `file` {String} | {Number} 文件名或文件描述符

- `data` {String} | {Buffer}
- `options` {Object} | {String}
  - `encoding` {String} | {Null} 默认 = `'utf8'`
  - `mode` {Number} 默认 = `0o666`
  - `flag` {String} 默认 = `'a'`
- `callback` {Function}

异步追加数据到文件，如果它已经存在，则替换该文件。`data` 可以是一个字符串或 buffer。

例子：

```
fs.appendFile('message.txt', 'data to append', (err) => {
    if (err) throw err;
    console.log('The "data to append" was appended to file!');
});
```

如果 `options` 是一个字符串，那么它指定了编码。例如：

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback);
```

任何指定的文件描述符必须为了追加而被打开。

注意：指定的文件描述符不会被自动关闭。

fs.appendFileSync(file, data[, options])

同步版的 fs.appendFile()。返回 `undefined`。

fs.ftruncate(fd, len, callback)

异步的 ftruncate(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.ftruncateSync(fd, len)

同步的 ftruncate(2)。返回 `undefined`。

fs.fstat(fd, callback)

异步的 fstat(2)。该回调获得两个参数 `(err, stats)`，`stats` 是一个 `fs.Stats` 对象。`fstat()` 与 fs.stat() 类似，除了文件是通过指定的文件描述符 `fd` 来说明。

fs.fstatSync(fd)

同步的 fstat(2)。返回一个 `fs.Stats` 实例。

fs.fchmod(fd, mode, callback)

异步的 fchmod(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.fchmodSync(fd, mode)

同步的 fchmod(2)。返回 `undefined`。

fs.fchown(fd, uid, gid, callback)

异步的 fchown(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.fchownSync(fd, uid, gid)

同步的 fchown(2)。返回 `undefined`。

fs.futimes(fd, atime, mtime, callback)

改变由所提供的文件描述符所引用的文件的文件时间戳。

fs.futimesSync(fd, atime, mtime)

同步版的 fs.futimes()。返回 `undefined`。

fs.fsync(fd, callback)

异步的 fsync(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.fsyncSync(fd)

同步的 fsync(2)。返回 `undefined`。

fs.fdatasync(fd, callback)

异步的 fdatasync(2)。除了一个可能的异常参数外没有其他参数会给到完成时的回调。

fs.fdatasyncSync(fd)

同步的 fdatasync(2)。返回 `undefined`。

Buffer API

`fs` 函数支持传递和接收字符串和 Buffer 式的路径。后者的目的是使得可以在允许非 UTF-8 文件名的文件系统中工作。对于大多数典型用途，能在 Buffer 式的路径中工作是多余的，因为字符串 API 自动转换为 UTF-8 形式。

请注意，在某些文件系统（如 NTFS 和 HFS+），文件名总是被编码为 UTF-8。在这些文件系统中，分配非 UTF-8 编码的 Buffers 到 `fs` 函数将无法像预期那样正常工作。

fs.ReadStream类

`ReadStream` 是一个 可读流。

'open' 事件

- `fd` {Number} 给 ReadStream 使用整数文件描述符。

在打开 ReadStream 文件时触发。

readStream.path

流读取的文件路径。

fs.WriteStream类

`WriteStream` 是一个 可写流。

'open' 事件

- `fd` {Number} 给 WriteStream 使用整数文件描述符。

在打开 WriteStream 文件时触发。

writeStream.path

流写入的文件路径。

writeStream.bytesWritten

迄今为止写入的字节数。不包括仍在排队等待写入的数据。

fs.Stats类

从 fs.stat()、fs.lstat() 和 fs.fstat() 及其同步版本返回的对象都属于这种类型。

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()`（只在 fs.lstat() 中有效）
- `stats.isFIFO()`
- `stats.isSocket()`

对于一个普通文件，util.inspect(stats) 将返回非常类似这样的字符串：

```
{
    dev: 2114,
    ino: 48064969,
    mode: 33188,
    nlink: 1,
    uid: 85,
    gid: 100,
    rdev: 0,
    size: 527,
    blksize: 4096,
    blocks: 8,
    atime: Mon, 10 Oct 2011 23:24:11 GMT,
    mtime: Mon, 10 Oct 2011 23:24:11 GMT,
    ctime: Mon, 10 Oct 2011 23:24:11 GMT,
    birthtime: Mon, 10 Oct 2011 23:24:11 GMT
}
```

请注意 `atime`、`mtime`、`birthtime` 和 `ctime` 是 Date 对象的实例，比较这些对象的值，你应该使用合适的方法。对于大多数一般用途 getTime() 会返回自 *1 January 1970 00:00:00 UTC* 已过的毫秒数并且该整数足以拿来做任何比较，然而也有可用于显示模糊信息的其他方法。更多的细节可以在 MDN JavaScript的参考页 中找到。

状态时间值

在状态对象中的时间有以下语义：

- `atime` “访问时间” - 文件数据最后被访问的时间。会被 `mknod(2)` 、 `utimes(2)` 和 `read(2)` 系统调用更改。
- `mtime` “修改时间” - 文件数据最后被修改的时间。会被 `mknod(2)` 、 `utimes(2)` 和 `write(2)` 系统调用更改。
- `ctime` “更改时间” - 文件状态上次更改的时间（索引节点数据修改）。会被 `chmod(2)` 、 `chown(2)` 、 `link(2)` 、 `mknod(2)` 、 `rename(2)` 、 `unlink(2)` 、 `utimes(2)` 、 `read(2)` 和 `write(2)` 系统调用更改。
- `birthtime` “出生时间” - 文件创建的时间。在创建文件时设定一次。在文件系统中出生日期不可用，这个字段可能代替保存 `ctime` 或 `1970-01-01T00:00Z` （如，Unix 的纪元时间戳 `0` ）两者任一。注意，此值也许比在此情况下的 `atime` 或 `mtime` 更加有用。在 Darwin 和其它的 FreeBSD 衍生系统中，如果时间被明确设置到了一个比目前出生时间较早的值，也会设置使用 `utimes(2)` 系统调用。

在 Node.js v0.12 之前的版本中，在 Windows 系统中， `ctime` 保存 `birthtime` 。请注意在 v0.12 中， `ctime` 不是“创建时间”，并且在 Unix 系统中，它从来都不是。

fs.FSWatcher类

从 `fs.watch()` 返回的对象是该类型。

'change' 事件

- `event` {String} fs 的类型变化。
- `filename` {String} 更改的文件名（如果相关/可用）

在监视的目录或文件更改了一些东西时触发。在 fs.watch() 中了解更多内容。

'error' 事件

- `error` {Error}

当发生错误时触发。

watcher.close()

对给定的 `fs.FSWatcher` 停止监视变化。

加密(Crypto)

> 稳定度：2 - 稳定

该 `crypto` 模块提供了加密功能，包括一组用于包装 OpenSSL 的哈希，HMAC，加密，解密，签名和验证函数。

使用 `require('crypto')` 来访问这个模块。

```
const crypto = require('crypto');

const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
                   .update('I love cupcakes')
                   .digest('hex');
console.log(hash);
// Prints:
//   c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea6ea959d6658e
```

方法和属性

---

crypto.DEFAULT_ENCODING

用于可以接受字符串或 buffers 的函数的默认编码。默认值为 `'buffer'`，这也使得这些方法的默认值为 Buffer 对象。

`crypto.DEFAULT_ENCODING` 机制用于处理期望值为 `'binary'` 的旧程序的向后兼容性。

新的应用程序应该期望默认值为 `'buffer'`。此属性在将来的 Node.js 版本中可能会被弃用。

crypto.createCipher(algorithm, password)

使用给定的 `algorithm` 和 `password` 创建并返回一个 `Cipher` 对象。

`algorithm` 依赖于 OpenSSL，例如 `'aes192'` 等。在最新的 OpenSSL 版本中，`openssl list-cipher-algorithms` 会显示可用的加密算法。

`password` 用于导出密钥和初始化向量（IV）。该值必须是 `'binary'` 编码的字符串或 Buffer。

`crypto.createCipher()` 的实现是使用 OpenSSL 的函数 EVP_BytesToKey 导出密钥，摘要算法设置为 MD5，一次迭代，没有盐。缺少盐将允许字典攻击，因为相同的密码总是创建相同的密钥。低迭代计数和非加密安全散列算法使得密码测试非常快。

OpenSSL 一致建议使用 pbkdf2 代替 EVP_BytesToKey，建议开发人员自己使用 crypto.pbkdf2() 导出密钥和 IV，并使用 crypto.createCipheriv() 创建 `Cipher` 对象。

crypto.createCipheriv(algorithm, key, iv)

使用给定的 `algorithm`、`password` 和初始化向量（`iv`）创建并返回一个 `Cipher` 对象。

`algorithm` 依赖于 OpenSSL，例如 `'aes192'` 等。在最新的 OpenSSL 版本中，`openssl list-cipher-algorithms` 会显示可用的加密算法。

`key` 是 `algorithm` 使用的原始密钥，同时 `iv` 是一个 初始化向量。所有参数都必须是 `'binary'` 编码的字符串或 buffers。

crypto.createCredentials(details)

> 稳定度：0 - 已废弃：使用 tls.createSecureContext() 代替。

`crypto.createCredentials()` 方法是用于创建的已弃用别名并返回一个 `tls.SecureContext` 对象。不应该去使用 `crypto.createCredentials()` 方法。

可选的 `details` 参数是一个带有键值的哈希对象：

- `pfx`：{String} | {Buffer} - PFX 或 PKCS12 编码的私钥、证书和 CA 证书。
- `key`：{String} - PEM 编码的私钥。
- `passphrase`：{String} - 私钥或 PFX 的密码。
- `cert`：{String} - PEM 编码的证书。
- `ca`：{String} | {Array} - PEM 编码的可信任 CA 证书的字符串或字符串数组。

- `crl` ：{String} | {Array} - PEM 编码的 CRL（证书吊销列表）的字符串或字符串数组。
- `ciphers` ：{String} - 使用 OpenSSL 加密列表格式来描述要使用或排除的加密算法。

如果没有给出 `'ca'` 的详细信息，Node.js 会使用 Mozilla 的默认公共信任 CA 列表。

crypto.createDecipher(algorithm, password)

使用给定的 `algorithm` 和 `password` （密钥）创建并返回一个 `Decipher` 对象。

`crypto.createDecipher()` 的实现是使用 OpenSSL 的函数 EVP_BytesToKey 导出密钥，摘要算法设置为 MD5，一次迭代，没有盐。缺少盐将允许字典攻击，因为相同的密码总是创建相同的密钥。低迭代计数和非加密安全散列算法使得密码测试非常快。

OpenSSL 一致建议使用 pbkdf2 代替 EVP_BytesToKey，建议开发人员自己使用 crypto.pbkdf2() 导出密钥和 IV，并使用 crypto.createCipheriv() 创建 `Decipher` 对象。

crypto.createDecipheriv(algorithm, key, iv)

使用给定的 `algorithm` 、 `password` 和初始化向量（ `iv` ）创建并返回一个 `Decipher` 对象。

`algorithm` 依赖于 OpenSSL，例如 `'aes192'` 等。在最新的 OpenSSL 版本中， `openssl list-cipher-algorithms` 会显示可用的加密算法。

`key` 是 `algorithm` 使用的原始密钥，同时 `iv` 是一个 初始化向量。所有参数都必须是 `'binary'` 编码的字符串或 buffers。

crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding])

使用提供的 `prime` 和可选的特定 `generator` 创建一个 `DiffieHellman` 密钥交换对象。

`generator` 参数应该是一个数字、字符串或 Buffer。如果没有指定 `generator` ，将使用值 `2` 。

`prime_encoding` 和 `generator_encoding` 参数可以是 `'binary'` 、 `'hex'` 或 `'base64'` 。

如果指定了 `prime_encoding` ， `prime` 期望是一个字符串或 Buffer。

如果指定了 `generator_encoding` ， `generator` 期望是一个字符串、数字或 Buffer。

crypto.createDiffieHellman(prime_length[, generator])

创建并使用可选的特定数字 `generator` 生成一个主要的 `prime_length` 位 `DiffieHellman` 密钥交换对象。如果没有指定 `generator`，将使用值 `2`。

crypto.createECDH(curve_name)

使用由 `curve_name` 字符串指定的预定义曲线创建一个 Elliptic Curve Diffie-Hellman（`ECDH`）密钥交换对象。使用 crypto.getCurves() 来获取可用的曲线名称列表。在最新的 OpenSSL 版本中，`openssl ecparam -list_curves` 也会显示每个可用的椭圆曲线名称和描述。

crypto.createHash(algorithm)

使用给定的 `algorithm` 创建并返回一个可以用于生成散列摘要的 `Hash` 对象。

`algorithm` 取决于平台上 OpenSSL 版本支持的可用算法。例如 `'sha256'`、`'sha512'` 等。在最新的 OpenSSL 版本中，`openssl list-message-digest-algorithms` 会显示可用的摘要算法。

示例：生成 sha256 的文件摘要：

```
const filename = process.argv[2];
const crypto = require('crypto');
const fs = require('fs');

const hash = crypto.createHash('sha256');

const input = fs.createReadStream(filename);
input.on('readable', () => {
    var data = input.read();
    if (data)
        hash.update(data);
    else {
        console.log(`${hash.digest('hex')} ${filename}`);
    }
});
```

crypto.createHmac(algorithm, key)

使用给定的 `algorithm` 和 `key` 创建并返回一个 `Hmac` 对象。

`algorithm` 取决于平台上 OpenSSL 版本支持的可用算法。例如 `'sha256'`、`'sha512'` 等。在最新的 OpenSSL 版本中，`openssl list-message-digest-algorithms` 会显示可用的摘要算法。

该 `key` 是用于生成加密 HMAC 哈希的 HMAC 密钥。

示例：生成 sha256 的文件 HMAC：

```
const filename = process.argv[2];
const crypto = require('crypto');
const fs = require('fs');

const hmac = crypto.createHmac('sha256', 'a secret');

const input = fs.createReadStream(filename);
input.on('readable', () => {
    var data = input.read();
    if (data)
        hmac.update(data);
    else {
        console.log(`${hmac.digest('hex')} ${filename}`);
    }
});
```

crypto.createSign(algorithm)

使用给定的 `algorithm` 创建并返回一个 `Sign` 对象。在最新的 OpenSSL 版本中，`openssl list-public-key-algorithms` 会显示可用的签名算法。例如 `'RSA-SHA256'`。

crypto.createVerify(algorithm)

使用给定的 `algorithm` 创建并返回一个 `Verify` 对象。在最新的 OpenSSL 版本中，`openssl list-public-key-algorithms` 会显示可用的签名算法。例如 `'RSA-SHA256'`。

crypto.getCiphers()

返回包含受支持的加密算法名称的数组。

示例：

```
const ciphers = crypto.getCiphers();
console.log(ciphers); // ['aes-128-cbc', 'aes-128-ccm', ...]
```

crypto.getCurves()

返回包含受支持的椭圆曲线名称的数组。

示例：

```
const curves = crypto.getCurves();
console.log(curves); // ['secp256k1', 'secp384r1', ...]
```

crypto.getDiffieHellman(group_name)

创建一个预定义的 `DiffieHellman` 密钥交换对象。支持的组是：`'modp1'`、`'modp2'`、`'modp5'`（在 RFC 2412 中定义，但请参阅警告）和 `'modp14'`、`'modp15'`、`'modp16'`、`'modp17'`、`'modp18'`（在 RFC 3526 中定义）。返回的对象模仿由 crypto.createDiffieHellman() 创建的对象接口，但不允许更改密钥（例如用 diffieHellman.setPublicKey()）。使用这种方法的优点是，各方不必预先生成或交换组模量，节省了处理器和通信的时间。

示例（获取共享密钥）：

```
const crypto = require('crypto');
const alice = crypto.getDiffieHellman('modp14');
const bob = crypto.getDiffieHellman('modp14');

alice.generateKeys();
bob.generateKeys();

const alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

crypto.getHashes()

返回包含受支持的散列算法名称的数组。

示例：

```
const hashes = crypto.getHashes();
console.log(hashes); // ['sha', 'sha1', 'sha1WithRSAEncryption', ...]
```

crypto.pbkdf2(password, salt, iterations, keylen[, digest], callback)

提供了一个异步的 Password-Based Key Derivation Function 2（PBKDF2）的实现。通过 `digest` 指定所选的 HMAC 摘要算法从 `password` 、 `salt` 和 `iterations` 应用于导出所请求的字节长度（ `keylen` ）的密钥。如果没有指定 `digest` 算法，将使用默认的 `'sha1'` 。

提供的 `callback` 函数在调用时带有两个参数： `err` 和 `derivedKey` 。如果发生错误， `err` 将被设置；否则， `err` 会是 `null` 。成功生成的 `derivedKey` 将作为 Buffer 传递。

`iterations` 参数必须设置为尽可能大的数字。迭代次数越高，导出密钥将更安全，但需要更长的时间来完成。

`salt` 应该尽可能地唯一。建议盐是随机的，它们的长度应该大于 16 字节。详见 NIST SP 800-132。

示例：

```
const crypto = require('crypto');
crypto.pbkdf2('secret', 'salt', 100000, 512, 'sha512', (err, key) => {
    if (err) throw err;
    console.log(key.toString('hex')); // 'c5e478d...1469e50'
});
```

可以使用 crypto.getHashes() 检索支持的摘要函数数组。

crypto.pbkdf2Sync(password, salt, iterations, keylen[, digest])

提供了一个同步的 Password-Based Key Derivation Function 2（PBKDF2）的实现。通过 `digest` 指定所选的 HMAC 摘要算法从 `password` 、 `salt` 和 `iterations` 应用于导出所请求的字节长度（ `keylen` ）的密钥。如果没有指定 `digest` 算法，将使用默认的 `'sha1'` 。

提供的 `callback` 函数在调用时带有两个参数： `err` 和 `derivedKey` 。如果发生错误， `err` 将被设置；否则， `err` 会是 `null` 。成功生成的 `derivedKey` 将作为 Buffer 传递。

`iterations` 参数必须设置为尽可能大的数字。迭代次数越高，导出密钥将更安全，但需要更长的时间来完成。

`salt` 应该尽可能地唯一。建议盐是随机的，它们的长度应该大于 16 字节。详见 NIST SP 800-132。

示例：

```
const crypto = require('crypto');
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 512, 'sha512');
console.log(key.toString('hex'));  // 'c5e478d...1469e50'
```

可以使用 crypto.getHashes() 检索支持的摘要函数数组。

crypto.privateEncrypt(private_key, buffer)

使用 `private_key` 加密 `buffer`。

`private_key` 可以是一个对象或字符串。如果 `private_key` 是一个字符串，它被视为没有密码短语密钥并使用 `RSA_PKCS1_OAEP_PADDING`。如果 `private_key` 是一个对象，它被解释为带有键值的哈希对象：

- `key`：{String} - PEM 编码的私钥。
- `passphrase`：{String} - 私钥的可选密码。
- `padding`：可选的填充值，是以下值之一：
  - `constants.RSA_NO_PADDING`
  - `constants.RSA_PKCS1_PADDING`
  - `constants.RSA_PKCS1_OAEP_PADDING`

所有的填充值都定义在 `constants` 模块中。

crypto.privateDecrypt(private_key, buffer)

使用 `private_key` 解密 `buffer`。

`private_key` 可以是一个对象或字符串。如果 `private_key` 是一个字符串，它被视为没有密码短语密钥并使用 `RSA_PKCS1_OAEP_PADDING`。如果 `private_key` 是一个对象，它被解释为带有键值的哈希对象：

- `key`：{String} - PEM 编码的私钥。
- `passphrase`：{String} - 私钥的可选密码。

- `padding` ：可选的填充值，是以下值之一：
  - `constants.RSA_NO_PADDING`
  - `constants.RSA_PKCS1_PADDING`
  - `constants.RSA_PKCS1_OAEP_PADDING`

所有的填充值都定义在 `constants` 模块中。

crypto.publicEncrypt(public_key, buffer)

使用 `public_key` 加密 `buffer` 。

`public_key` 可以是一个对象或字符串。如果 `public_key` 是一个字符串，它被视为没有密码短语密钥并使用 `RSA_PKCS1_OAEP_PADDING` 。如果 `public_key` 是一个对象，它被解释为带有键值的哈希对象：

- `key` ：{String} - PEM 编码的公钥。
- `passphrase` ：{String} - 私钥的可选密码。
- `padding` ：可选的填充值，是以下值之一：
  - `constants.RSA_NO_PADDING`
  - `constants.RSA_PKCS1_PADDING`
  - `constants.RSA_PKCS1_OAEP_PADDING`

因为 RSA 公钥可以从私钥导出，所以可以传递私钥而不是公钥。

所有的填充值都定义在 `constants` 模块中。

crypto.publicDecrypt(public_key, buffer)

使用 `public_key` 解密 `buffer` 。

`public_key` 可以是一个对象或字符串。如果 `public_key` 是一个字符串，它被视为没有密码短语密钥并使用 `RSA_PKCS1_OAEP_PADDING` 。如果 `public_key` 是一个对象，它被解释为带有键值的哈希对象：

- `key` ：{String} - PEM 编码的公钥。
- `passphrase` ：{String} - 私钥的可选密码。
- `padding` ：可选的填充值，是以下值之一：

- `constants.RSA_NO_PADDING`
- `constants.RSA_PKCS1_PADDING`
- `constants.RSA_PKCS1_OAEP_PADDING`

因为 RSA 公钥可以从私钥导出，所以可以传递私钥而不是公钥。

所有的填充值都定义在 `constants` 模块中。

crypto.randomBytes(size[, callback])

以加密方式生成强的伪随机数据。`size` 参数是指示要生成的字节数的数字。

如果提供了 `callback` 函数，将异步生成字节并且 `callback` 函数的调用会带有两个参数：`err` 和 `buf`。如果发生错误，`err` 会是一个 Error 对象；否则会是 `null`。`buf` 参数会是一个包含生成的字节的 Buffer。

```
// 异步
const crypto = require('crypto');
crypto.randomBytes(256, (err, buf) => {
    if (err) throw err;
    console.log(`${buf.length} bytes of random data: ${buf.toString('hex')}`);
});
```

如果没有提供 `callback` 函数，将同步产生随机字节并作为 Buffer 返回。如果生成字节时出现问题，将抛出错误。

```
// 同步
const buf = crypto.randomBytes(256);
console.log(`${buf.length} bytes of random data: ${buf.toString('hex')}`);
```

`crypto.randomBytes()` 方法会造成阻塞直到有足够的熵。这通常不会花费多于几毫秒的时间。在启动后，当整个系统仍然处于低熵时，可想而知，在产生随机字节时，它将会阻塞更长的时间。

crypto.setEngine(engine[, flags])

加载和设置某些或所有的 OpenSSL 函数的 `engine`（取决于 `flags`）。

`engine` 可以是引擎共享库的 id 或路径。

可选的 `flags` 参数默认使用 `ENGINE_METHOD_ALL`。`flags` 是一个位字段，采用以下标志之一或其混合（定义在 `constants` 模块中）：

- `ENGINE_METHOD_RSA`
- `ENGINE_METHOD_DSA`
- `ENGINE_METHOD_DH`
- `ENGINE_METHOD_RAND`
- `ENGINE_METHOD_ECDH`
- `ENGINE_METHOD_ECDSA`
- `ENGINE_METHOD_CIPHERS`
- `ENGINE_METHOD_DIGESTS`
- `ENGINE_METHOD_STORE`
- `ENGINE_METHOD_PKEY_METHS`
- `ENGINE_METHOD_PKEY_ASN1_METHS`
- `ENGINE_METHOD_ALL`
- `ENGINE_METHOD_NONE`

Cipher类

`Cipher` 类的实例用于加密数据。该类可以以两种方式进行使用：

- 作为可读和可写的流，写入简单的未加密数据并在可读端上产生加密数据；
- 使用 cipher.update() 和 cipher.final() 方法来产生加密数据。

crypto.createCipher() 和 crypto.createCipheriv() 方法用于创建 `Cipher` 实例。`Cipher` 对象无法直接使用 `new` 关键词创建。

示例：将 `Cipher` 对象用作流：

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

var encrypted = '';
cipher.on('readable', () => {
    var data = cipher.read();
    if (data)
        encrypted += data.toString('hex');
});
cipher.on('end', () => {
    console.log(encrypted);
    // Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504
});

cipher.write('some clear text data');
cipher.end();
```

示例：使用 `Cipher` 并导流：

```
const crypto = require('crypto');
const fs = require('fs');
const cipher = crypto.createCipher('aes192', 'a password');

const input = fs.createReadStream('test.js');
const output = fs.createWriteStream('test.enc');

input.pipe(cipher).pipe(output);
```

示例：使用 cipher.update() 和 cipher.final() 方法：

```
const crypto = require('crypto');
const cipher = crypto.createCipher('aes192', 'a password');

var encrypted = cipher.update('some clear text data', 'utf8', 'hex');
encrypted += cipher.final('hex');
console.log(encrypted);
// Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504
```

cipher.setAAD(buffer)

使用验证加密模式（目前仅支持 `GCM` ）时，`cipher.setAAD()` 方法设置的值用于附加认证数据（AAD）输入参数。

cipher.setAutoPadding(auto_padding=true)

当使用块加密算法时，`Cipher` 类会自动添加适当块大小的填充到输入数据中。调用 `cipher.setAutoPadding(false)` 禁用默认填充。

当 `auto_padding` 为 `false` 时，整个输入数据的长度必须是加密块大小的倍数，否则，cipher.final() 会抛出一个错误。禁用自动填充对于非标准填充有用，例如使用 `0x0` 而不是 PKCS 填充。

`cipher.setAutoPadding()` 方法必须在 cipher.final() 之前调用。

cipher.getAuthTag()

使用验证加密模式（目前仅支持 `GCM` ）时，`cipher.getAuthTag()` 方法返回一个包含已经从给定数据计算得出的认证标签的 Buffer。

`cipher.getAuthTag()` 方法只应在使用 cipher.final() 方法完成加密后才能调用。

cipher.update(data[, input_encoding][, output_encoding])

用 `data` 更新加密内容。如果给定了 `input_encoding` 参数，它的值必须是 `'utf8'`、`'ascii'` 或 `'binary'` 其中之一并且 `data` 参数是使用指定编码的字符串。如果没有给定 `input_encoding` 参数，`data` 必须是一个 Buffer。如果 `data` 是一个 Buffer，那么 `input_encoding` 参数会被忽略。

`output_encoding` 指定加密数据的输出格式，可以是 `'binary'`、`'base64'` 或 `'hex'`。如果指定了 `output_encoding`，将返回使用指定编码的字符串。如果没有提供 `output_encoding`，将返回一个 Buffer。

`cipher.update()` 方法可以在新数据上多次调用，直到调用 cipher.final()。在 cipher.final() 之后调用 `cipher.update()` 将导致抛出错误。

cipher.final([output_encoding])

返回任何剩余的加密内容。如果 `output_encoding` 参数是 `'binary'`、`'base64'` 或 `'hex'` 中的一个，将返回一个字符串。如果没有提供 `output_encoding`，将返回一个 Buffer。

一旦调用了 `cipher.final()` 方法，`Cipher` 对象不能再用于加密数据。尝试多次调用 `cipher.final()` 将导致抛出错误。

Decipher类

`Decipher` 类的实例用于解密数据。该类可以以两种方式进行使用：

- 作为可读和可写的流，写入简单的加密数据并在可读端上产生未加密数据；
- 使用 decipher.update() 和 decipher.final() 方法来产生加密数据。

crypto.createDecipher() 和 crypto.createDecipheriv() 方法用于创建 `Decipher` 实例。`Decipher` 对象无法直接使用 `new` 关键词创建。

示例：将 `Decipher` 对象用作流：

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

var decrypted = '';
decipher.on('readable', () => {
    var data = decipher.read();
    if (data)
        decrypted += data.toString('utf8');
});
decipher.on('end', () => {
    console.log(decrypted);
    // Prints: some clear text data
});

var encrypted = 'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504';
decipher.write(encrypted, 'hex');
decipher.end();
```

示例：使用 `Decipher` 并导流：

```
const crypto = require('crypto');
const fs = require('fs');
const decipher = crypto.createDecipher('aes192', 'a password');

const input = fs.createReadStream('test.enc');
const output = fs.createWriteStream('test.js');

input.pipe(decipher).pipe(output);
```

示例：使用 decipher.update() 和 decipher.final() 方法：

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

var encrypted = 'ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616ed84d815504';
var decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
// Prints: some clear text data
```

decipher.setAAD(buffer)

使用验证加密模式（目前仅支持 `GCM` ）时，`decipher.setAAD()` 方法设置的值用于附加认证数据（AAD）输入参数。

decipher.setAutoPadding(auto_padding=true)

当数据已经被加密而没有标准块填充时，调用 `decipher.setAutoPadding(false)` 将禁用自动填充防止 decipher.final() 检查和删除填充。

关闭自动填充功能仅在输入数据的长度是加密块大小的倍数时有效。

`decipher.setAutoPadding()` 方法必须在 decipher.update() 之前调用。

decipher.setAuthTag(buffer)

使用验证加密模式（目前仅支持 `GCM` ）时，`decipher.setAuthTag()` 方法用于传入接收的认证标签。如果没有提供标签，或者如果密文已经被篡改，decipher.final() 会抛出由于认证失败而应丢弃密文的指示。

cipher.update(data[, input_encoding][, output_encoding])

用 `data` 更新解密内容。如果给定了 `input_encoding` 参数，它的值必须是 `'utf8'`、`'ascii'` 或 `'binary'` 其中之一并且 `data` 参数是使用指定编码的字符串。如果没有给定 `input_encoding` 参数，`data` 必须是一个 Buffer。如果 `data` 是一个 Buffer，那么 `input_encoding` 参数会被忽略。

`output_encoding` 指定加密数据的输出格式，可以是 `'binary'`、`'base64'` 或 `'hex'`。如果指定了 `output_encoding`，将返回使用指定编码的字符串。如果没有提供 `output_encoding`，将返回一个 Buffer。

`decipher.update()` 方法可以在新数据上多次调用，直到调用 decipher.final()。在 decipher.final() 之后调用 `decipher.update()` 将导致抛出错误。

cipher.final([output_encoding])

返回任何剩余的解密内容。如果 `output_encoding` 参数是 `'binary'`、`'base64'` 或 `'hex'` 中的一个，将返回一个字符串。如果没有提供 `output_encoding`，将返回一个 Buffer。

一旦调用了 `decipher.final()` 方法，`Decipher` 对象不能再用于解密数据。尝试多次调用 `decipher.final()` 将导致抛出错误。

Certificate类

SPKAC 是 Netscape 最初实现的证书签名请求机制并且现在正式指定为 HTML5 的 keygen 元素的一部分。

`crypto` 模块提供了 `Certificate` 类用于处理 SPKAC 数据。最常见的用法是处理 HTML5 `<keygen>` 元素生成的输出。Node.js 内部使用 OpenSSL 的 SPKAC 来实现。

new crypto.Certificate()

`Certificate` 类的实例可以通过使用 `new` 关键词或调用 `crypto.Certificate()` 函数来创建：

```
const crypto = require('crypto');

const cert1 = new crypto.Certificate();
const cert2 = crypto.Certificate();
```

certificate.exportPublicKey(spkac)

`spkac` 数据结构包括公钥和咨询。`certificate.exportPublicKey()` 以 Node.js 的 Buffer 形式返回公钥组件。`spkac` 参数既可以是字符串也可以是 Buffer。

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const publicKey = cert.exportPublicKey(spkac);
console.log(publicKey);
// Prints the public key as <Buffer ...>
```

certificate.exportChallenge(spkac)

`spkac` 数据结构包括公钥和咨询。`certificate.exportChallenge()` 以 Node.js 的 Buffer 形式返回公钥组件。`spkac` 参数既可以是字符串也可以是 Buffer。

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
const challenge = cert.exportChallenge(spkac);
console.log(challenge.toString('utf8'));
// Prints the challenge as a UTF8 string
```

certificate.verifySpkac(spkac)

如果给定的 `spkac` 数据结构有效，则返回 `true`，否则，返回 `false`。`spkac` 参数必须是 Node.js 的 Buffer 形式。

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(new Buffer(spkac)));
// Prints true or false
```

DiffieHellman类

`DiffieHellman` 类是一个用于创建 Diffie-Hellman 密钥交换的工具类。

`DiffieHellman` 类的实例可以使用 crypto.createDiffieHellman() 函数来创建。

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createDiffieHellman(2048);
const alice_key = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createDiffieHellman(alice.getPrime(), alice.getGenerator());
const bob_key = bob.generateKeys();

// Exchange and generate the secret...
const alice_secret = alice.computeSecret(bob_key);
const bob_secret = bob.computeSecret(alice_key);

// OK
assert.equal(alice_secret.toString('hex'), bob_secret.toString('hex'));
```

diffieHellman.verifyError

一个包含检查在执行 `DiffieHellman` 对象初始化过程中所产生的任何警告和/或错误的位字段。

以下值对此属性有效（定义在 `constants` 模块中）：

- `DH_CHECK_P_NOT_SAFE_PRIME`
- `DH_CHECK_P_NOT_PRIME`
- `DH_UNABLE_TO_CHECK_GENERATOR`
- `DH_NOT_SUITABLE_GENERATOR`

diffieHellman.generateKeys([encoding])

生成私有和公开 Diffie-Hellman 键值，并返回指定 `encoding` 的公钥。这个密钥应该转移到另一方。编码可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding` 则返回一个字符串，否则返回一个 Buffer。

diffieHellman.getGenerator([encoding])

返回指定 `encoding` 的 Diffie-Hellman 生成器。`encoding` 可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding` 则返回一个字符串，否则返回一个 Buffer。

diffieHellman.setPublicKey(public_key[, encoding])

设置一个 Diffie-Hellman 公钥。如果提供了 `encoding` 参数，且是 `'binary'`、`'hex'` 或 `'base64'` 中的一种，则 `public_key` 期望是一个字符串。如果没有提供 `encoding`，则 `public_key` 期望是一个 Buffer。

diffieHellman.getPublicKey([encoding])

返回指定 `encoding` 的 Diffie-Hellman 公钥。`encoding` 可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding` 则返回一个字符串，否则返回一个 Buffer。

diffieHellman.setPrivateKey(private_key[, encoding])

设置一个 Diffie-Hellman 私钥。如果提供了 `encoding` 参数，且是 `'binary'`、`'hex'` 或 `'base64'` 中的一种，则 `private_key` 期望是一个字符串。如果没有提供 `encoding`，则 `private_key` 期望是一个 Buffer。

diffieHellman.getPrivateKey([encoding])

返回指定 `encoding` 的 Diffie-Hellman 私钥。`encoding` 可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding` 则返回一个字符串，否则返回一个 Buffer。

diffieHellman.getPrime([encoding])

返回指定 `encoding` 的 Diffie-Hellman 素数。`encoding` 可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding` 则返回一个字符串，否则返回一个 Buffer。

diffieHellman.computeSecret(other_public_key[, input_encoding][, output_encoding])

使用 `other_public_key` 作为对方的公钥计算共享密钥并返回计算后的共享密钥。使用指定的 `input_encoding` 解释提供的密钥，并返回使用指定 `output_encoding` 编码的密钥。编码可以是 `'binary'`、`'hex'` 或 `'base64'`。如果没有提供 `input_encoding`，`other_public_key` 期望是一个 Buffer。

如果给出了 `output_encoding`，将返回一个字符串；否则，返回一个 Buffer。

ECDH类

`ECDH` 类是用来创建 Elliptic Curve Diffie-Hellman（ECDH）密钥交换的工具类。

`ECDH` 类的实例可以使用 crypto.createECDH() 函数来创建。

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createECDH('secp521r1');
const alice_key = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createECDH('secp521r1');
const bob_key = bob.generateKeys();

// Exchange and generate the secret...
const alice_secret = alice.computeSecret(bob_key);
const bob_secret = bob.computeSecret(alice_key);

assert(alice_secret, bob_secret);
// OK
```

ecdh.generateKeys([encoding[, format]])

生成私有的和公共的 EC Diffie-Hellman 键值，并返回指定 `format` 和 `encoding` 的公钥。这个密钥应该转移到另一方。

`format` 参数指定点编码，可以是 `'compressed'` 、`'uncompressed'` 或 `'hybrid'` 。如果没有指定 `format` ，这个点会以 `'uncompressed'` 格式返回。

`encoding` 参数可以是 `'binary'` 、`'hex'` 或 `'base64'` 。如果提供了 `encoding` ，会返回一个字符串；否则返回一个 Buffer。

ecdh.setPrivateKey(private_key[, encoding])

设置一个 EC Diffie-Hellman 私钥。`encoding` 可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding`，`private_key` 则期望是个字符串；否则，`private_key` 期望是一个 Buffer。当 `ECDH` 对象被创建时，如果 `private_key` 指定的曲线无效，将抛出错误。在设置私钥时，还在 ECDH 对象中生成和设置了相关联的公共点（密钥）。

ecdh.getPrivateKey([encoding])

返回指定了 `encoding` 的 EC Diffie-Hellman 私钥，它可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `encoding`，会返回一个字符串；否则返回一个 Buffer。

ecdh.setPublicKey(public_key[, encoding])

> 稳定度：0 - 已废弃

设置一个 EC Diffie-Hellman 公钥。密钥的编码可以是 `'binary'`、`'hex'` 或 `'base64'`。如果提供了 `public_key` 则期望是个字符串；否则，期望是一个 Buffer。

请注意，通常没有理由调用此方法，因为 `ECDH` 只需要私钥和对方的公钥来计算共享密钥。通常会调用 ecdh.generateKeys() 或 ecdh.setPrivateKey()。ecdh.setPrivateKey() 方法尝试生成与正被设置的私钥相关联的公共点/密钥。

示例（获取共享密钥）：

```
const crypto = require('crypto');
const alice = crypto.createECDH('secp256k1');
const bob = crypto.createECDH('secp256k1');

// Note: This is a shortcut way to specify one of Alice's previous private
// keys. It would be unwise to use such a predictable private key in a real
// application.
alice.setPrivateKey(
    crypto.createHash('sha256').update('alice', 'utf8').digest()
);

// Bob uses a newly generated cryptographically strong
// pseudorandom key pair bob.generateKeys();

const alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
const bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

// alice_secret and bob_secret should be the same shared secret value
console.log(alice_secret === bob_secret);
```

ecdh.getPublicKey([encoding[, format]])

返回指定了 `encoding` 和 `format` 的 EC Diffie-Hellman 公钥。

`format` 参数指定点编码，可以是 `'compressed'` 、 `'uncompressed'` 或 `'hybrid'` 。如果没有指定 `format` ，这个点会以 `'uncompressed'` 格式返回。

`encoding` 参数可以是 `'binary'` 、 `'hex'` 或 `'base64'` 。如果提供了 `encoding` ，会返回一个字符串；否则返回一个 Buffer。

ecdh.computeSecret(other_public_key[, input_encoding][, output_encoding])

使用作为对方的公钥的 `other_public_key` 计算共享密钥，并返回计算后的共享密钥。使用指定的 `input_encoding` 解释提供的密钥，并返回使用指定 `output_encoding` 编码的密钥。编码可以是 `'binary'` 、 `'hex'` 或 `'base64'` 。如果没有提供 `input_encoding` ， `other_public_key` 期望是一个 Buffer。

如果给出了 `output_encoding` ，将返回一个字符串；否则，返回一个 Buffer。

Hash类

`Hash` 类是用于创建数据散列摘要的工具类。它可以以两种方式之一使用：

- 作为可读和可写的流，写入数据并在可读端产生计算后的散列摘要。
- 使用 hash.update() 和 hash.digest() 产生计算后的散列。

crypto.createHash() 方法用于创建 `Hash` 实例。`Hash` 对象无法直接使用 `new` 关键词创建。

示例：将 `Hash` 对象用作流：

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.on('readable', () => {
    var data = hash.read();
    if (data)
        console.log(data.toString('hex'));
    // Prints:
    //   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
});

hash.write('some data to hash');
hash.end();
```

示例：使用 `Hash` 并导流：

```
const crypto = require('crypto');
const fs = require('fs');
const hash = crypto.createHash('sha256');

const input = fs.createReadStream('test.js');
input.pipe(hash).pipe(process.stdout);
```

示例：使用 hash.update() 和 hash.digest() 方法：

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.update('some data to hash');
console.log(hash.digest('hex'));
// Prints:
//   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f4e3993e50
```

hash.update(data[, input_encoding])

使用给定的 `data` 更新哈希内容，给出的 `input_encoding` 编码，可以是 `'utf8'`、`'ascii'` 或 `'binary'`。如果没有提供 `encoding`，同时 `data` 是一个字符串，将强制使用 `'binary'` 编码。如果 `data` 是一个 Buffer，那么 `input_encoding` 参数会被忽略。

当它作为流时，可以在新数据上多次调用。

hash.digest([encoding])

计算所有传入的数据的散列摘要（使用 hash.update() 方法）。`encoding` 可以是 `'hex'`、`'binary'` 或 `'base64'`。如果提供了 `encoding`，会返回一个字符串；否则返回一个 Buffer。

在调用 `hash.digest()` 方法之后，`Hash` 对象将不能再次使用。多次调用将导致抛出错误。

Hmac类

`Hmac` 类是用于创建加密的 HMAC 摘要的工具类。它可以以两种方式之一使用：

- 作为可读和可写的流，写入数据并在可读端产生计算后的 HMAC 摘要。
- 使用 hmac.update() 和 hmac.digest() 产生计算后的 HMAC 摘要。

crypto.createHmac() 方法用于创建 `Hmac` 实例。`Hmac` 对象无法直接使用 `new` 关键词创建。

示例：将 `Hmac` 对象用作流：

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.on('readable', () => {
    var data = hmac.read();
    if (data)
        console.log(data.toString('hex'));
    // Prints:
    //   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
});

hmac.write('some data to hash');
hmac.end();
```

示例：使用 `Hmac` 并导流：

```
const crypto = require('crypto');
const fs = require('fs');
const hmac = crypto.createHmac('sha256', 'a secret');

const input = fs.createReadStream('test.js');
input.pipe(hmac).pipe(process.stdout);
```

示例：使用 hmac.update() 和 hmac.digest() 方法：

```
const crypto = require('crypto');
const hmac = crypto.createHmac('sha256', 'a secret');

hmac.update('some data to hash');
console.log(hmac.digest('hex'));
// Prints:
//   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f7f77e
```

hmac.update(data[, input_encoding])

使用给定的 `data` 更新 HMAC 内容，给出的 `input_encoding` 编码，可以是 `'utf8'`、`'ascii'` 或 `'binary'`。如果没有提供 `encoding`，同时 `data` 是一个字符串，将强制使用 `'binary'` 编码。如果 `data` 是一个 Buffer，那么 `input_encoding` 参数会被忽略。

当它作为流时，可以在新数据上多次调用。

hmac.digest([encoding])

计算所有传入的的数据的 HMAC 摘要（使用 hmac.update() 方法）。`encoding` 可以是 `'hex'`、`'binary'` 或 `'base64'`。如果提供了 `encoding`，会返回一个字符串；否则返回一个 Buffer。

在调用 `hmac.digest()` 方法之后，`Hmac` 对象将不能再次使用。多次调用将导致抛出错误。

Sign类

`Sign` 类是用于生成签名的工具类。它可以以两种方式之一使用：

- 作为一个可写流，写入要签名的数据，sign.sign() 用于生成和返回签名。
- 使用 sign.update() 和 sign.sign() 方法来产生签名。

crypto.createSign() 方法用于创建 `Sign` 实例。`Sign` 对象无法直接使用 `new` 关键词创建。

示例：将 `Sign` 对象用作流：

```
const crypto = require('crypto');
const sign = crypto.createSign('RSA-SHA256');

sign.write('some data to sign');
sign.end();

const private_key = getPrivateKeySomehow();
console.log(sign.sign(private_key, 'hex'));
// Prints the calculated signature
```

示例：使用 sign.update() 和 sign.sign() 方法：

```
const crypto = require('crypto');
const sign = crypto.createSign('RSA-SHA256');

sign.update('some data to sign');

const private_key = getPrivateKeySomehow();
console.log(sign.sign(private_key, 'hex'));
// Prints the calculated signature
```

`sign` 实例也可以通过传入摘要算法名称来创建，在这种情况下 OpenSSL 将从 PEM 格式的私钥类型推断完全的签名算法，包括没有直接暴露名称常量的算法，如，`'ecdsa-with-SHA256'`。

示例：使用 SHA256 的 ECDSA 签名：

```
const crypto = require('crypto');
const sign = crypto.createSign('sha256');

sign.update('some data to sign');

const private_key = '-----BEGIN EC PRIVATE KEY-----\n' +
        'MHcCAQEEIF+jnWY1D5kbVYDNvxxo/Y+ku2uJPDwS0r/VuPZQrjjVoAoGCCqGSM49\n' +
        'AwEHoUQDQgAEurOxfSxmqIRYzJVagdZfMMSjRNNhB8i3mXyIMq704m2m52FdfKZ2\n' +
        'pQhByd5eyj3lgZ7m7jbchtdgyOF8Io/1ng==\n' +
        '-----END EC PRIVATE KEY-----\n';

console.log(sign.sign(private_key).toString('hex'));
```

sign.update(data[, input_encoding])

用给定的 `data` 更新 `Sign` 内容，给出的 `input_encoding` 编码，可以是 `'utf8'` 、 `'ascii'` 或 `'binary'` 。如果没有提供 `encoding` ，同时 `data` 是一个字符串，将强制使用 `'utf8'` 编码。如果 `data` 是一个 Buffer，那么 `input_encoding` 参数会被忽略。

当它作为流时，可以在新数据上多次调用。

sign.sign(private_key[, output_format])

使用 sign.update() 或 sign.write() 计算所有传递的数据的签名。

`private_key` 参数可以是一个对象或字符串。如果 `private_key` 是一个字符串，它被视为没有密码的原始密钥。如果 `private_key` 是一个对象，它被解释为包含两个属性的哈希：

- `key` ：{String} - PEM 编码的私钥
- `passphrase` ：{String} - 私钥的密码

`output_format` 可以指定为 `'binary'` 、 `'hex'` 或 `'base64'` 中的一种。如果提供了 `output_format` 则返回一个字符串；否则返回一个 Buffer。

在调用 `sign.sign()` 方法之后， `Sign` 对象将不能再次使用。多次调用将导致抛出错误。

Verify类

`Verify` 类是用于验证签名的工具类。它可以以两种方式之一使用：

- 作为一个可写流，写入数据用于根据所提供的签名进行验证。
- 使用 verify.update() 和 verify.verify() 方法来验证签名。

crypto.createVerify() 方法用于创建 `Verify` 实例。`Verify` 对象无法直接使用 `new` 关键词创建。

示例：将 `Verify` 对象用作流：

```
const crypto = require('crypto');
const verify = crypto.createVerify('RSA-SHA256');

verify.write('some data to sign');
verify.end();

const public_key = getPublicKeySomehow();
const signature = getSignatureToVerify();
console.log(sign.verify(public_key, signature));
// Prints true or false
```

示例：使用 verify.update() 和 verify.verify() 方法：

```
const crypto = require('crypto');
const verify = crypto.createVerify('RSA-SHA256');

verify.update('some data to sign');

const public_key = getPublicKeySomehow();
const signature = getSignatureToVerify();
console.log(verify.verify(public_key, signature));
// Prints true or false
```

verifier.update(data[, input_encoding])

用给定的 `data` 更新 `Verify` 内容，给出的 `input_encoding` 编码，可以是 `'utf8'`、`'ascii'` 或 `'binary'`。如果没有提供 `encoding`，同时 `data` 是一个字符串，将强制使用 `'utf8'` 编码。如果 `data` 是一个 Buffer，那么 `input_encoding` 参数会被忽略。

当它作为流时，可以在新数据上多次调用。

verifier.verify(object, signature[, signature_format])

使用给定的 `object` 和 `signature` 验证提供的数据。`object` 参数是一个包含 PEM 编码对象的字符串，它可以是 RSA 公钥、DSA 公钥或一个 X.509 证书。`signature` 参数是先前计算的数据的签名，`signature_format` 可以是 `'binary'`、`'hex'` 或 `'base64'`。如果指定了 `signature_format`，`signature` 期望是一个字符串，否则期望是一个 Buffer。

返回 `true` 或 `false` 取决于数据和公钥的签名的有效性。

在调用 `verify.verify()` 方法之后，`Verify` 对象将不能再次使用。多次调用 `verify.verify()` 将导致抛出错误。

注意

旧的流 API（Node.js v0.10 之前的版本）

在存在统一的 Stream API 的概念之前，Crypto 模块已添加到 Node.js 中，并在之前用 Buffer 对象处理二进制数据。因此，许多 `crypto` 定义的类有通常在其他由 streams API 实现的 Node.js 类（如，`update()` 、`final()` 或 `digest()` ）中找不到的方法。此外，许多方法接受和返回 `'binary'` 编码的字符串，而不是 Buffer。这个默认值在 Node.js v0.8 之后改为默认使用 Buffer 对象。

近期 ECDH 的变化

非动态生成的密钥对的 ECDH 的用法已被简化。如今，ecdh.setPrivateKey() 可以使用预选的私钥来调用，并且相关联的公共点（密钥）将被计算并存储在对象中。这允许代码仅存储和提供 EC 密钥对的私有部分。ecdh.setPrivateKey() 现在还可以验证私钥对所选曲线的有效性。

ecdh.setPublicKey() 方法现在已被废弃，因为把它包含在 API 中的并没有什么用处。应该设置先前存储的私钥，或者调用 ecdh.generateKeys()。使用ecdh.setPublicKey() 的主要缺点是它可能会使得 ECDH 密钥对处于不一致的状态。

对弱的或泄密的算法的支持

`crypto` 模块仍然支持一些已经泄密和目前不推荐使用的算法。API 还允许使用具有较小密钥大小的密码和散列，这些密钥和散列被认为太弱而无法安全使用。

用户应根据其安全要求，对选择的加密算法和密钥大小负全责。

基于 NIST SP 800-131A 的建议：

- MD5 和 SHA-1 在需要抗冲突性的场合（例如，数字签名）时将不可接受。

- 用于 RSA、DSA 和 DH 算法的密钥建议至少 2048 位，并且 ECDSA 和 ECDH 的曲线至少为 224 位，这样的话，在几年内可以安全使用。
- `modp1` 、 `modp2` 和 `modp5` 的 DH 组的密钥大小小于 2048 位，不推荐使用。

有关其他建议和详细信息，请详见参考。

压缩解压(ZLIB)

稳定度：2 - 稳定

你可以访问此模块：

```
const zlib = require('zlib');
```

这打包提供了 Gzip/Gunzip、 Deflate/Inflate 和 DeflateRaw/InflateRaw 类。每个类都采用相同的选项，并且是一个可读/可写流。

方法和属性

- zlib.createGzip([options])
- zlib.createGunzip([options])
- zlib.createUnzip([options])
- zlib.createDeflate([options])
- zlib.createInflate([options])
- zlib.createDeflateRaw([options])
- zlib.createInflateRaw([options])

便捷方法

- zlib.gzip(buf[, options], callback)
- zlib.gzipSync(buf[, options])
- zlib.gunzip(buf[, options], callback)
- zlib.gunzipSync(buf[, options])
- zlib.unzip(buf[, options], callback)
- zlib.unzipSync(buf[, options])
- zlib.deflate(buf[, options], callback)
- zlib.deflateSync(buf[, options])
- zlib.inflate(buf[, options], callback)
- zlib.inflateSync(buf[, options])
- zlib.deflateRaw(buf[, options], callback)
- zlib.deflateRawSync(buf[, options])
- zlib.inflateRaw(buf[, options], callback)
- zlib.inflateRawSync(buf[, options])

---

zlib.createGzip([options])

返回一个用 options 生成的新的 Gzip 对象。

zlib.createGunzip([options])

返回一个用 options 生成的新的 Gunzip 对象。

zlib.createUnzip([options])

返回一个用 options 生成的新的 Unzip 对象。

zlib.createDeflate([options])

返回一个用 options 生成的新的 Deflate 对象。

zlib.createInflate([options])

返回一个用 options 生成的新的 Inflate 对象。

zlib.createDeflateRaw([options])

返回一个用 options 生成的新的 DeflateRaw 对象。

zlib.createInflateRaw([options])

返回一个用 options 生成的新的 InflateRaw 对象。

zlib.gzip(buf[, options], callback)

zlib.gzipSync(buf[, options])

通过 Gzip 来压缩一个 Buffer 或字符串。

zlib.gunzip(buf[, options], callback)

zlib.gunzipSync(buf[, options])

通过 Gunzip 来解压缩一个 Buffer 或字符串。

zlib.unzip(buf[, options], callback)

zlib.unzipSync(buf[, options])

通过 Unzip 来解压缩一个 Buffer 或字符串。

zlib.deflate(buf[, options], callback)

zlib.deflateSync(buf[, options])

通过 Deflate 来压缩一个 Buffer 或字符串。

zlib.inflate(buf[, options], callback)

zlib.inflateSync(buf[, options])

通过 Inflate 来解压缩一个 Buffer 或字符串。

zlib.deflateRaw(buf[, options], callback)

zlib.deflateRawSync(buf[, options])

通过 DeflateRaw 来压缩一个 Buffer 或字符串。

zlib.inflateRaw(buf[, options], callback)

zlib.inflateRawSync(buf[, options])

通过 InflateRaw 来解压缩一个 Buffer 或字符串。

压缩解压类

---

zlib.Zlib 类

这个类未被 `zlib` 模块导出，编入此文档是因为它是其它压缩器/解压缩器的基类。

zlib.flush([kind], callback)

`kind` 默认为 `zlib.Z_FULL_FLUSH`。

Flush 待处理的数据。请勿轻易调用此方法，过早的 flush 会影响压缩算法的有效性。

调用该方法只能 flush 从 zlib 获取的内部状态数据，并且不会执行任何流级别类型的 flushing。它的行为反而像一个正常的 `.write()` 回调，例如，当从流读出数据时，它被排着其他未处理的写入后面并且只会产生输出。

zlib.params(level, strategy, callback)

动态更新压缩级别和压缩策略。仅适用于 deflate 算法。

zlib.reset()

将压缩器/解压缩器重置为默认出厂值。仅对 inflate 和 deflate 算法有效。

zlib.Gzip 类

使用 Gzip 压缩数据。

zlib.Gunzip 类

解压缩一个 Gunzip 流。

zlib.Unzip 类

通过自动检测报头来解压缩一个以 Gzip 或 Deflate 压缩的流。

zlib.Deflate 类

使用 Deflate 压缩数据。

zlib.Inflate 类

解压缩一个 Inflate 流。

zlib.DeflateRaw 类

使用 DeflateRaw 压缩数据，并且不追加 Zlib 头。

zlib.InflateRaw 类

解压缩一个 InflateRaw 流。

类参数

每个类需要一个 options 对象。所有选项都是可选的。

请注意有些选项仅对压缩有效，并会被解压缩类所忽略。

- flush（默认：`zlib.Z_NO_FLUSH`）
- finishFlush（默认：`zlib.Z_FINISH`）
- chunkSize（默认：16*1024）
- windowBits
- level（仅用于压缩）
- memLevel（仅用于压缩）
- strategy（仅用于压缩）
- dictionary（仅用于 deflate/inflate，缺省为空目录）

`deflateInit2` 和 `inflateInit2` 的描述可以在 http://zlib.net/manual.html#Advanced 上查阅到更多内容。

常量

所有定义在 zlib.h 上的常量也同样定义在 `require('zlib')`。在正常的操作过程中，你几乎不会用到这些。编入文档只是为了让你对它们的存在不会感到意外。该章节几乎完全来自 zlib 的文档。详见 http://zlib.net/manual.html#Constants。

允许的 flush 值。

- `zlib.Z_NO_FLUSH`
- `zlib.Z_PARTIAL_FLUSH`
- `zlib.Z_SYNC_FLUSH`
- `zlib.Z_FULL_FLUSH`
- `zlib.Z_FINISH`
- `zlib.Z_BLOCK`
- `zlib.Z_TREES`

压缩/解压缩函数的返回值。负数代表错误，正数代表特殊但正常的事件。

- `zlib.Z_OK`
- `zlib.Z_STREAM_END`
- `zlib.Z_NEED_DICT`
- `zlib.Z_ERRNO`
- `zlib.Z_STREAM_ERROR`
- `zlib.Z_DATA_ERROR`
- `zlib.Z_MEM_ERROR`
- `zlib.Z_BUF_ERROR`
- `zlib.Z_VERSION_ERROR`

压缩级别。

- `zlib.Z_NO_COMPRESSION`
- `zlib.Z_BEST_SPEED`
- `zlib.Z_BEST_COMPRESSION`

- `zlib.Z_DEFAULT_COMPRESSION`

压缩策略。

- `zlib.Z_FILTERED`
- `zlib.Z_HUFFMAN_ONLY`
- `zlib.Z_RLE`
- `zlib.Z_FIXED`
- `zlib.Z_DEFAULT_STRATEGY`

data_type 字段的可能值。

- `zlib.Z_BINARY`
- `zlib.Z_TEXT`
- `zlib.Z_ASCII`
- `zlib.Z_UNKNOWN`

deflate 压缩方法（该版本仅支持一种）。

- `zlib.Z_DEFLATED`

初始化 zalloc / zfree / opaque。

- `zlib.Z_NULL`

优化内存占用

Node.js 中的用法修改自 `zlib/zconf.h`：

deflate 的内存需求（按字节）：

```
(1 << (windowBits+2)) +  (1 << (memLevel+9))
```

表示：128K 的 windowBits = 15 + 128K 的 memLevel = 8（默认值）加上几 KB 的小对象。

例如，如果你需要将默认内存需求从 256K 减少到 128K，设置选项：

```
{ windowBits: 14, memLevel: 7 }
```

当然，这通常会降低压缩等级（天下没有免费午餐）。

inflate 的内存需求（按字节）：

```
1 << windowBits
```

表示：32K 的 windowBits = 15（默认值）加上几 KB 的小对象。

这是除了内部输出的单个缓冲外的 `chunkSize` 大小，默认为 16K。

zlib 的压缩速度主要受压缩级别 `level` 的影响。更高的压缩级别会有更好的压缩率，但也要花费更长时间。更低的压缩级别会有较低压缩率，但速度更快。

通常，使用更多内存的选项意味着 Node.js 能减少对 zlib 的调用，因为单次 `write` 操作能处理更多数据。因此，这是另一个影响速度和内存占用的因素。

Flushing

在一个压缩流中调用 .flush() 会使得 zlib 尽可能多地返回当前的可能值。这可能会降低压缩质量的成本，但这在数据需要尽快使用时非常有用。

在以下的例子中，`flush()` 被用于在客户端写入一个部分压缩的 HTTP 响应：

```
const zlib = require('zlib');
const http = require('http');

http.createServer((request, response) => {
    // For the sake of simplicity, the Accept-Encoding checks are omitted.
    response.writeHead(200, {
        'content-encoding': 'gzip'
    });
    const output = zlib.createGzip();
    output.pipe(response);

    setInterval(() => {
        output.write(`The current time is ${Date()}\n`, () => {
            // The data has been passed to zlib, but the compression algorithm may
            // have decided to buffer the data for more efficient compression.
            // Calling .flush() will make the data available as soon as the client
            // is ready to receive it.
            output.flush();
        });
    }, 1000);
}).listen(1337);
```

示例

压缩或解压缩一个文件可以通过导流一个 fs.ReadStream 到一个 zlib 流，然后到一个 fs.WriteStream 来完成。

```
const gzip = zlib.createGzip();
const fs = require('fs');
const inp = fs.createReadStream('input.txt');
const out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

一步压缩或解压缩数据可以通过快捷方法来完成。

```
const input = '.................................';
zlib.deflate(input, (err, buffer) => {
    if (!err) {
        console.log(buffer.toString('base64'));
    } else {
        // handle error
    }
});

const buffer = new Buffer('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, (err, buffer) => {
    if (!err) {
        console.log(buffer.toString());
    } else {
        // handle error
    }
});
```

要在 HTTP 客户端或服务器中使用此模块，请在请求中使用 accept-encoding 和在响应中使用 content-encoding 头。

注意：这些例子只是极其简单地展示了基础的概念。Zlib 编码消耗非常大，其结果应当被缓存。详见 优化内存占用 中更多的关于 Zlib 用法中对速度 / 内存 / 压缩的权衡取舍。

```
// client request example
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
const request = http.get({
    host: 'izs.me',
    path: '/',
```

```
    port: 80,
    headers: {
        'accept-encoding': 'gzip,deflate'
    }
});
request.on('response', (response) => {
    var output = fs.createWriteStream('izs.me_index.html');

    switch (response.headers['content-encoding']) {
        // or, just use zlib.createUnzip() to handle both cases
    case 'gzip':
        response.pipe(zlib.createGunzip()).pipe(output);
        break;
    case 'deflate':
        response.pipe(zlib.createInflate()).pipe(output);
        break;
    default:
        response.pipe(output);
        break;
    }
});

// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
const zlib = require('zlib');
const http = require('http');
const fs = require('fs');
http.createServer((request, response) => {
    var raw = fs.createReadStream('index.html');
    var acceptEncoding = request.headers['accept-encoding'];
    if (!acceptEncoding) {
        acceptEncoding = '';
    }

    // Note: this is not a conformant accept-encoding parser.
    // See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3
    if (acceptEncoding.match(/\bdeflate\b/)) {
        response.writeHead(200, {
            'content-encoding': 'deflate'
        });
        raw.pipe(zlib.createDeflate()).pipe(response);
    } else if (acceptEncoding.match(/\bgzip\b/)) {
        response.writeHead(200, {
            'content-encoding': 'gzip'
        });
        raw.pipe(zlib.createGzip()).pipe(response);
    } else {
        response.writeHead(200, {});
        raw.pipe(response);
    }
}).listen(1337);
```

默认情况下，Zlib 方法在截断解压缩数据时，会抛出一个错误。然而，如果已知该数据是不完整的，或仅仅是希望检查一个压缩文件的开始部分，通过改变用于压缩输入数据最后一个数据块的 flushing 方法，有可能可以抑制默认的错误处理程序：

```
// This is a truncated version of the buffer from the above examples
const buffer = new Buffer('eJzT0yMA', 'base64');

zlib.unzip(buffer, {
    finishFlush: zlib.Z_SYNC_FLUSH
}, (err, buffer) => {
    if (!err) {
        console.log(buffer.toString());
    } else {
        // handle error
    }
});
```

这不会改变其他情况下抛出错误的行为，例如，当输入无效格式的数据。使用该方法，这将不可能确定输入是否提前结束或是否缺乏完整性检查，因此有必要手动检查该解压缩结果的有效性。

网络(Net)

> 稳定度：2 - 稳定

该 `net` 模块提供了一个异步网络包装。它包含了用于创建服务器和客户端（被称为流）的功能。你可以通过 `require('net');` 包含此模块。

方法和属性

- net.createServer([options][, connectionListener])
- net.createConnection(options[, connectListener])
- net.createConnection(path[, connectListener])
- net.createConnection(port[, host][, connectListener])
- net.connect(options[, connectListener])
- net.connect(path[, connectListener])
- net.connect(port[, host][, connectListener])
- net.isIP(input)
- net.isIPv4(input)
- net.isIPv6(input)

---

net.createServer([options][, connectionListener])

创建一个新的服务器。`connectionListener` 参数自动设置为 'connection' 事件 的一个监听器。

`options` 是一个对象包含以下默认值：

```
{
    allowHalfOpen: false,
    pauseOnConnect: false
}
```

如果 `allowHalfOpen` 是 `true`，那么当其他 socket 端发送了一个 FIN 包时，socket 不会自动发送 FIN 包。该 socket 变成不可读但可写的状态。你应该明确地调用 end() 方法。详见 'end' 事件了解更多详情。

如果 `pauseOnConnect` 是 `true`，那么与每个传入连接关联的 socket 都会被暂停，并且从它的句柄中无法读取数据。这使得连接在不读取原始进程的数据的情况下，在进程之间传递。调用 resume() 从暂停 socket 中开始读取数据。

这是一个监听端口 8124 的连接的回声服务器例子：

```
const net = require('net');
const server = net.createServer((c) => {
    // 'connection' listener
    console.log('client connected');
    c.on('end', () => {
        console.log('client disconnected');
    });
    c.write('hello\r\n');
    c.pipe(c);
});
server.on('error', (err) => {
    throw err;
});
server.listen(8124, () => {
    console.log('server bound');
});
```

使用 `telnet` 进行测试：

```
telnet localhost 8124
```

要监听 socket `/tmp/echo.sock`，只要将倒数的第三行改为

```
server.listen('/tmp/echo.sock', () => {
    console.log('server bound');
});
```

使用 `nc` 来连接一个 `UNIX` 域 socket 服务器：

```
nc -U /tmp/echo.sock
```

net.createConnection(options[, connectListener])

一个工厂函数，返回一个新的 net.Socket 实例，并使用提供的 `options` 自动连接。

`options` 同时被传到 net.Socket 构造函数和 socket.connect 方法。

`connectListener` 参数会被添加为 'connect' 事件 的监听器一次。

这是先前描述的回声服务器的客户端的一个例子：

```
const net = require('net');
const client = net.createConnection({
    port: 8124
}, () => {
    //'connect' listener
    console.log('connected to server!');
    client.write('world!\r\n');
});
client.on('data', (data) => {
    console.log(data.toString());
    client.end();
});
client.on('end', () => {
    console.log('disconnected from server');
});
```

要监听 socket `/tmp/echo.sock`，只要将第二行改为

```
const client = net.createConnection({path: '/tmp/echo.sock'});
```

net.createConnection(path[, connectListener])

一个工厂函数，返回一个新的 unix net.Socket 实例，并使用提供的 `path` 自动连接。

`connectListener` 参数会被添加为 'connect' 事件 的监听器一次。

net.createConnection(port[, host][, connectListener])

一个工厂函数，返回一个新的 net.Socket 实例，并使用提供的 `path` 和 `host` 自动连接。

如果省略了 `host`，`'localhost'` 将会承担其职责。

`connectListener` 参数会被添加为 'connect' 事件 的监听器一次。

net.connect(options[, connectListener])

一个工厂函数，返回一个新的 net.Socket 实例，并使用提供的 `options` 自动连接。

`options` 同时被传到 net.Socket 构造函数和 socket.connect 方法。

`connectListener` 参数会被添加为 'connect' 事件 的监听器一次。

这是先前描述的回声服务器的客户端的一个例子：

```
const net = require('net');
const client = net.connect({
    port: 8124
}, () => {
    // 'connect' listener
    console.log('connected to server!');
    client.write('world!\r\n');
});
client.on('data', (data) => {
    console.log(data.toString());
    client.end();
});
client.on('end', () => {
    console.log('disconnected from server');
});
```

要监听 socket `/tmp/echo.sock`，只要将第二行改为

```
const client = net.connect({path: '/tmp/echo.sock'});
```

net.connect(path[, connectListener])

一个工厂函数，返回一个新的 unix net.Socket 实例，并使用提供的 `path` 自动连接。

`connectListener` 参数会被添加为 'connect' 事件 的监听器一次。

net.connect(port[, host][, connectListener])

一个工厂函数，返回一个新的 net.Socket 实例，并使用提供的 `path` 和 `host` 自动连接。

如果省略了 `host`，`'localhost'` 将会承担其职责。

`connectListener` 参数会被添加为 'connect' 事件 的监听器一次。

net.isIP(input)

检测是否输入是一个 IP 地址。对于无效的字符串，返回 0，对于 IPv4 的地址返回 4，对于 IPv6 地址返回 6。

net.isIPv4(input)

如果输入的是版本 4 的 IP 地址，则返回 true，否则返回 false。

net.isIPv6(input)

如果输入的是版本 6 的 IP 地址，则返回 true，否则返回 false。

net.Server类

此类用于创建 TCP 或本地服务器。

`net.Server` 是一个带有以下事件的 EventEmitter。

'connection' 事件

- {net.Socket} 连接对象

当一个新的连接建立时发出。`socket` 是一个 `net.Socket` 实例。

'listening' 事件

当服务器调用 `server.listen` 后被绑定时发出。

'close' 事件

当服务器关闭时发出。请注意，如果存在连接，不会发出此事件，直到所有的连接都结束了才发出。

'error' 事件

- {Error}

当发生错误时发出。'close' 事件将这个事件后直接调用，详见在 `server.listen` 讨论中的例子。

server.maxConnections

当服务器的连接数变高时，设置此属性可以拒绝更多的连接。

一旦一个 socket 通过 child_process.fork() 被发送到子进程时，不推荐使用此选项。

server.connections

> 稳定度：0 - 已废弃：使用 server.getConnections() 替代。

服务器上的并行连接数。

当通过 child_process.fork() 发送一个 socket 到子进程时，它会变成 `null`。要轮询派生进程并获得当前活动的连接号，请使用异步的 `server.getConnections` 代替。

server.listen(port[, hostname][, backlog][, callback])

在指定的 `port` 和 `hostname` 上，开始接受连接。如果省略 `hostname`，当 `IPv6` 可用时，该服务器将接收任何 IPv6 地址（`::`）的连接，否则，接收任何 IPv4 地址（`0.0.0.0`）的连接。当端口值为零时，将分配一个随机端口。

`backlog` 正在连接队列的最大长度。实际长度会根据你的操作系统通过 sysctl 设置来定，例如在 linux 上的 `tcp_max_syn_backlog` 和 `somaxconn`。该参数的默认值是 `511`（不是 `512`）。

该函数是异步的。当服务器已经绑定，将发出 'listening' 事件。最后的参数 `callback` 将会作为 'listening' 事件的一个监听器。

有一个问题，一些用户在运行后得到了 `EADDRINUSE` 错误。这意味着，另一个服务器已经运行在了请求的端口上。处理的方法之一是等待一秒钟，然后重试。这可以这么做：

```
server.on('error', (e) => {
    if (e.code == 'EADDRINUSE') {
        console.log('Address in use, retrying...');
        setTimeout(() => {
            server.close();
            server.listen(PORT, HOST);
        }, 1000);
    }
});
```

（注意：Node.js 的所有 socket 都已经设置了 `SO_REUSEADDR` 。）

server.listen(path[, backlog][, callback])

- `path` {String}
- `backlog` {Number}
- `callback` {Function}

启动本地 socket 服务器监听指定 `path` 上的连接。

该函数是异步的。当服务器已经绑定，将发出 'listening' 事件。最后的参数 `callback` 将会作为 'listening' 事件的一个监听器。

在 UNIX 中，本地域通常称之为 UNIX 域。路径是一个文件系统路径名。它受到相同的命名约定和在文件创建时完成的权限检查，将在文件系统中可见，并将持续到解除链接。

在 Windows 中，本地域使用命名 pipe 来实现。该路径必须参考 `\\?\pipe\` 或 `\\.\pipe\` 条目。允许任何字符，但后者可能会做 pipe 名称的一些处理，如解析 `..` 序列。尽管出现，该 pipe 命名空间是平的。Pipe 不会持续下去，当它们的最后参考被关闭时，它们将被移除。不要忘记 JavaScript 字符串转义 requirejs 路径用双反斜线来指定。例如：

```
net.createServer().listen(path.join('\\\\?\\pipe', process.cwd(), 'myctl'))
```

`backlog` 参数的行为类似于 server.listen(port[, hostname][, backlog][, callback])。

server.listen(handle[, backlog][, callback])

- `handle` {Object}

- `backlog` {Number}
- `callback` {Function}

`handle` 对象既可以被设置为一个服务器，也可以被设置为一个 socket（任何具有的基础 `_handle` 任何成员）或一个 `{fd: <n>}` 对象。

这会使得服务器接受指定的句柄连接，并假设它是一个文件描述符或已经绑定到端口的句柄或域 socket。

在 Windows 上不支持监听文件描述符。

该函数是异步的。当服务器已经绑定，将发出 'listening' 事件。最后的参数 `callback` 将会作为 'listening' 事件的一个监听器。

`backlog` 参数的行为类似于 server.listen(port[, hostname][, backlog][, callback])。

```
server.listen({
    host: 'localhost',
    port: 80,
    exclusive: true
});
```

server.listen(options[, callback])

- `options` {Object} - 需要。支持以下属性：
  - `port` {Number} - 可选。
  - `host` {String} - 可选。
  - `backlog` {Number} - 可选。
  - `path` {String} - 可选。
  - `exclusive` {Boolean} - 可选。
- `callback` - 可选。

`options` 的 `port`、`host` 和 `backlog` 属性以及可选的回调函数，它们的行为和在 server.listen(port[, hostname][, backlog][, callback]) 中的调用相同。另外，`path` 选项可用于指定 UNIX socket。

如果 `exclusive` 是 `false`（默认），那么集群的工作进程将使用相同的基础句柄，允许连接共享处理业务。当 `exclusive` 是 `true` 时，不共享句柄，并试图端口共享结果会导致错误。监听专用端口上的例子如下所示。

server.address()

返回绑定的地址，该地址名称族和服务器的端口由操作系统报告。当得到一个系统分配的地址时，用于查找哪个端口已经被分配。返回具有三个属性的对象，如 `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }` 。

示例：

```
var server = net.createServer((socket) => {
    socket.end('goodbye\n');
}).on('error', (err) => {
    // handle errors here
    throw err;
});

// grab a random port.
server.listen(() => {
    address = server.address();
    console.log('opened server on %j', address);
});
```

直到发生 'listening' 事件前，不要调用 `server.address()` 。

server.close([callback])

停止服务器接受新的连接，并保持现有的连接。该函数是异步的，当所有的连接都结束并且服务器发生 'close' 事件时，服务器最终关闭。可选的 `callback` 在 `'close'` 事件发生时调用一次。不同于事件，当它被关闭时，如果服务器未打开，它将用 Error 作为其唯一的参数调用。

server.unref()

如果这是在事件系统中唯一的活动服务器，在服务器上调用 `unref` 将允许程序退出。如果服务器已经 `unref` d，再次调用 `unref` 将没有效果。

返回 `server` 。

server.ref()

与 `unref` 相反，在一个之前 `unref` d 的服务器上调用 `ref` ，不会让程序退出，如果它是唯一剩下的服务器（默认行为）。如果服务器已经 `ref` d，再次调用 `ref` 将没有效果。

返回 `server`。

net.Socket类

该对象是 TCP 或本地 socket 的抽象。`net.Socket` 实例实现了一个双工（duplex）流接口。它们可以由用户创建并当成一个客户端（通过 connect()），或由 Node.js 创建并通过服务器的 `'connection'` 事件传递给用户。

'lookup' 事件

在解析主机名之后，但在连接之前发出事件。不适用于 Unix socket 。

- `err` {Error} | {Null} 错误对象。详见 dns.lookup()。
- `address` {String} IP 地址。
- `family` {String | Null} 地址类型。详见 dns.lookup()。
- `host` {String} 主机名。

'connect' 事件

当成功建立了 socket 连接时发出。详见 connect()。

'data' 事件

- {Buffer}

当接收到数据时发出。`data` 参数会是一个 `Buffer` 或 `String` 。数据的编码通过 `socket.setEncoding()` 设置（请参阅只读流章节获取更多信息）。

请注意，当一个 `Socket` 发出 `'data'` 事件时，如果没有设置监听器，数据将会丢失。

'drain' 事件

在写入 buffer 变为空时发出。可用于限制上传。

也可以看看： `socket.write()` 的返回值。

'end' 事件

当 socket 的另一端发送了一个 FIN 包时发生。

默认情况下（allowHalfOpen == false），一旦 socket 写完了待写队列中的数据，它就会销毁它的文件描述符。然而，通过设置 `allowHalfOpen == true`，socket 不会在它这一端自动调用 `end()`，允许用户写入任意数量的数据，用户在他们这一端要求 `end()` 会得到警告。

'close' 事件

- `had_error` {Boolean} `true`，如果该 socket 发生了一个传输错误。

一旦 socket 完全关闭时发出。`had_error` 参数是一个布尔值，表示 socket 是否是由于传输错误而关闭的。

'timeout' 事件

在 socket 闲置超时的时候发出。这只是通知 socket 已经闲置。用户必须手动关闭连接。

也可以看看：socket.setTimeout()。

'error' 事件

- {Error}

在发生错误时发出。`'close'` 事件会直接在该事件之后调用。

new net.Socket([options])

构造一个新的 Socket 对象。

`options` 是一个带有以下默认值的对象：

```
{
    fd: null,
    allowHalfOpen: false,
    readable: false,
    writable: false
}
```

`fd` 允许你指定现有的 socket 文件描述符。设置 `readable` 或 `writable` 为 `true` 将允许在该 socket 上读取或写入（注意：仅当传入 `fd` 时有效）。关于 `allowHalfOpen`，请参考 `createServer()` 和 `'end'` 事件。

`net.Socket` 实例是 EventEmitter。

socket.bufferSize

`net.Socket` 有该属性以保证 `socket.write()` 总是有效。这是为了帮助用户快速启动和运行。计算机不能总是赶上写入到一个 socket 中的数据量——网络连接只可能太慢。Node.js 在内部以队列形式写入到 socket 中，并在合适时机通过报文发送。（在内部，Node.js 轮询 socket 的文件描述符以保证可写）

内部缓存可能会导致内存占用增加。该属性表示要被写入到当前缓冲的字符数。（字符数大约等于被写入的字节数，但 buffer 可能包含字符串并且该字符串是惰性编码的，所以无法知晓字节的确切数目。）

遇到大的或增长的 `bufferSize` 的用户，应该在他们的程序中尝试通过 pause() 和 resume() 进行数据“节流”。

socket.bytesRead

已接收的字节数。

socket.bytesWritten

发送的字节数。

socket.localPort

该数字表示本地端口。例如，`80` 或 `21`。

socket.localAddress

该字符串表示用于远程客户端连接的本地 IP 地址。例如，如果你监听 `'0.0.0.0'` 并且客户端连接到了 `'192.168.1.1'`，该值将会是 `'192.168.1.1'`。

socket.remotePort

该数字表示远程端口。例如，`80` 或 `21`。

socket.remoteFamily

该数字表示远程 IP 族。`'IPv4'` 或 `'IPv6'`。

socket.remoteAddress

该数字表示远程 IP 地址。例如，`'74.125.127.100'` 或 `'2001:4860:a005::68'`。如果 socket 被销毁，那么值也可能是 `undefined`（例如，假设客户端断开连接）。

socket.setEncoding([encoding])

设置 socket 的编码作为一个只读流。详见 stream.setEncoding() 了解更多信息。

socket.setTimeout(timeout[, callback])

设置 socket 在闲置了 `timeout` 毫秒后超时。默认情况下，`net.Socket` 不会超时。当空闲超时被触发后，socket 会收到一个 'timeout' 事件，但连接不会被切断。用户必须手动 end() 或 destroy() socket。

如果 `timeout` 是 0，那么现有的空闲超时会被禁用。

可选的 `callback` 参数将被添加为 'timeout' 事件的一次性监听器。

返回 `socket`。

socket.setNoDelay([noDelay])

禁用 Nagle 算法。默认情况下 TCP 连接使用 Nagle 算法，它们在发送关闭之前缓存数据。将 `noDelay` 设置为 `true`，会在每次调用 `socket.write()` 时立即发出数据。`noDelay` 默认为 `true`。

返回 `socket`。

socket.setKeepAlive([enable][, initialDelay])

启用/禁用保持活跃功能，并且可以在空闲的 socket 发送第一个存活探测之前选择性地设置初始延迟。`enable` 默认为 `false`。

通过 `initialDelay`（以毫秒为单位）设置接收到的最后一个数据包和第一个存活探测之间的延迟。将 `initialDelay` 设置为 0，会改变默认（或之前）设置的值。默认为 `0`。

返回 `socket`。

socket.connect(path[, connectListener])

socket.connect(port[, host][, connectListener])

类似 socket.connect(options[, connectListener])，其选项为 `{port: port, host: host}` 或 `{path: path}` 。

socket.connect(options[, connectListener])

打开一个给定的 socket 连接。

对于 TCP sockets 而言，`options` 参数应该是一个特定的对象：

- `port` ：应该连接到客户端端口（必需）。
- `host` ：应该连接到客户端主机。默认为 `'localhost'` 。
- `localAddress` ：绑定到用于网络连接的本地接口。
- `localPort` ：绑定到用于网络连接的本地端口。
- `family` ：IP 协议栈的版本。默认为 `4` 。
- `hints` ：dns.lookup() 提示。默认为 `0` 。
- `lookup` ：自定义查找功能。默认为 `dns.lookup` 。

对于本地域 sockets，`options` 参数应该是一个特定的对象：

- `path` ：应该连接到客户端路径（必需）。

通常不需要这种方法，作为 `net.createConnection` 打开 socket。只有当你要实现自定义的 Socket 时使用此功能。

此函数是异步的。当 'connect' 事件发出时，会建立 socket。如果连接有问题，不会发出 `'connect'` 事件，将带着异常发出 'error' 事件。

`connectListener` 参数将被添加为 'connect' 事件的监听器。

socket.write(data[, encoding][, callback])

发送 socket 上的数据。第二个参数在 `data` 为字符串的情况下用于指定编码——默认为 UTF8 编码。

如果整个数据被成功刷新到内核缓冲区，则返回 `true`。如果全部或部分的数据还在用户内存队列中，则返回 `false`。当 buffer 再次清空时会发出 'drain' 事件。

当数据最终被写入时，可选的 `callback` 参数将被执行——这可能不会立即执行。

socket.pause()

暂停读取数据。即不会发出 'data' 事件。对于上传节流非常有用。

socket.resume()

在调用 pause() 后恢复读取。

socket.end([data][, encoding])

半关闭 socket。如，它发送一个 FIN 包。服务器仍然可能会发送一些数据。

如果指定了 `data`，这等同于在 `socket.end()` 后调用 `socket.write(data, encoding)`。

socket.destroy()

确保没有更多的 I/O 活动发生此 socket 上。只有在错误（解析错误）的情况下有必要。

socket.address()

返回绑定的地址，socket 的地址族名称和端口由操作系统所决定。返回一个具有三个属性的对象，如 `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`。

socket.unref()

在 socket 中调用 `unref`，如果这是在事件系统中唯一活跃的 socket，将允许程序退出。如果 socket 已经 `unref` d，再次调用 `unref` 将没有效果。

返回 `socket`。

socket.ref()

与 `unref` 相反，在一个先前 `unref`d socket 上调用 `ref`，如果它是唯一剩下的 socket（默认行为），将不会让程序退出。如果 socket 已经 `ref`d，再次调用 `ref` 将没有效果。

返回 `socket`。

域名服务(DNS)

> 稳定度：2 - 稳定

`dns` 模块包含属于两个不同类别的函数：

1）函数使用底层的操作系统设备执行名称解析，并且不一定执行任何类型的网络通信。这个类别只包含一个函数： dns.lookup()。开发者希望和同一操作系统中的其他应用程序的行为相同的方式执行名称解析应该使用 **dns.lookup()**。

例如，监视 `nodejs.org` 。

```
const dns = require('dns');

dns.lookup('nodejs.org', (err, addresses, family) => {
    console.log('addresses:', addresses);
});
```

2）函数连接到实际的DNS服务器执行名称解析，并且总是使用网络来执行 DNS 查询。此类别包含 `dns` 模块里除了 dns.lookup() 以外的所有函数。这些函数不使用与 dns.lookup() 相同的配置文件（如， `/etc/hosts` ）。这些函数应该由不希望使用底层的操作系统设备执行名称解析的开发者使用，并且希望始终执行 DNS 查询。

下面是解析 `'nodejs.org'` ，然后反向解析返回的 IP 地址的例子。

```
const dns = require('dns');

dns.resolve4('nodejs.org', (err, addresses) => {
    if (err) throw err;

    console.log(`addresses: ${JSON.stringify(addresses)}`);

    addresses.forEach((a) => {
        dns.reverse(a, (err, hostnames) => {
            if (err) {
                throw err;
            }
            console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
        });
    });
});
```

一个在另一个之上进行选择会有微妙的后果，详情参阅实施注意事项部分获取更多信息。

方法和属性

---

dns.setServers(servers)

设置解析时要使用的服务器的 IP 地址。`servers` 参数是一组 IPv4 或 IPv6 地址。

如果在一个地址上指定了端口，则该端口会被忽略。

如果提供了无效的地址将会抛出一个错误。

`dns.setServers()` 方法一定不能在 DNS 查询过程中调用。

dns.getServers()

返回一组正被用于名称解析的 IP 地址字符串。

dns.resolve(hostname[, rrtype], callback)

使用 DNS 协议解析主机名（如，`'nodejs.org'`）到由 `rrtype` 指定类型的一组记录。

有效的 `rrtype` 值包括：

- `'A'` - IPV4 地址，默认
- `'AAAA'` - IPV6 地址
- `'MX'` - 邮件交换记录
- `'TXT'` - 文本记录
- `'SRV'` - SRV 记录
- `'PTR'` - 用于逆向 IP 查找
- `'NS'` - 名称服务器记录
- `'CNAME'` - 规范名称记录
- `'SOA'` - 规范记录的开头

`callback` 函数有参数 `(err, addresses)`。如果成功，`addresses` 会是一个数组。在 `addresses` 中的每个项目的类型是由记录类型确定，并且描述在相应查找方法的文档中。

当发生错误时，`err` 是一个 Error 对象，`err.code` 会是列举在这里的错误码的一种。

dns.resolve4(hostname, callback)

使用 DNS 协议将 `hostname` 解析为 IPv4 地址（`A` 记录）。传递给 `callback` 函数的 `addresses` 参数会包含一组 IPv4 地址（如，`['74.125.79.104', '74.125.79.105', '74.125.79.106']`）。

dns.resolve6(hostname, callback)

使用 DNS 协议将 `hostname` 解析为 IPv6 地址（`AAAA` 记录）。传递给 `callback` 函数的 `addresses` 参数会包含一组 IPv6 地址。

dns.resolveMx(hostname, callback)

使用 DNS 协议将 `hostname` 解析为邮件交换记录（`MX` 记录）。传递给 `callback` 函数的 `addresses` 参数会包含一组带有 `priority` 和 `exchange` 属性的对象（如，`[{priority: 10, exchange: 'mx.example.com'}, ...]`）。

dns.resolveTxt(hostname, callback)

使用 DNS 协议将 `hostname` 解析为文本查询（`TXT` 记录）。传递给 `callback` 函数的 `addresses` 参数是一个适用于 `hostname` 的二维阵列文本记录（如，`[['v=spf1 ip4:0.0.0.0', '~all']]`）。每个子阵列包含一个记录的文本块（TXT chunks）。根据使用案例，这些可能被拼接在一起或分别处理。

dns.resolveSrv(hostname, callback)

使用 DNS 协议将 `hostname` 解析为服务记录（`SRV` 记录）。传递给 `callback` 函数的 `addresses` 参数会是一组由以下属性构成的对象：

- `priority`
- `weight`
- `port`
- `name`

```
{
    priority: 10,
    weight: 5,
    port: 21223,
    name: 'service.example.com'
}
```

dns.resolveNs(hostname, callback)

使用 DNS 协议将 `hostname` 解析为名称服务器记录（`NS` 记录）。传递给 `callback` 函数的 `addresses` 参数会包含一组适用于 `hostname` 的名称服务器记录（如，`['ns1.example.com', 'ns2.example.com']`）。

dns.resolveCname(hostname, callback)

使用 DNS 协议将 `hostname` 解析为 `CNAME` 记录。传递给 `callback` 函数的 `addresses` 参数会包含一组适用于 `hostname` 的规范名称记录（如，`['bar.example.com']`）。

dns.resolveSoa(hostname, callback)

使用 DNS 协议将 `hostname` 解析为一个规范记录的开头（`SOA` 记录）。传递给 `callback` 函数的 `addresses` 参数会是一组由以下属性构成的对象：

- `nsname`
- `hostmaster`
- `serial`
- `refresh`
- `retry`
- `expire`
- `minttl`

```
{
    nsname: 'ns.example.com',
    hostmaster: 'root.example.com',
    serial: 2013101809,
    refresh: 10000,
    retry: 2400,
    expire: 604800,
    minttl: 3600
}
```

dns.reverse(ip, callback)

执行反向 DNS 查询将 IPv4 或 IPv6 地址解析为一组主机名。

`callback` 函数有参数 `(err, hostnames)`。`hostnames` 是一组用给定的 `ip` 解析后的主机名。

当发生错误时，`err` 是一个 Error 对象，`err.code` 是 DNS 错误码 中的一个。

dns.lookup(hostname[, options], callback)

解析主机名（如，`'nodejs.org'`）到第一个找到的 A（IPv4）或 AAAA（IPv6）记录。`options` 可以是一个对象或整数。如果没有提供 `options`，那么 IPv4 和 IPv6 地址都是有效的。如果 `options` 是一个整数，那么它必须是 `4` 或 `6`。

另外，`options` 可以是包含这些属性的对象：

- `family`：{Number} - 记录族。如果存在的话，必须是整数 `4` 或 `6`。如果没有提供，IPv4 和 IPv6 的地址都可以接受。

- `hints` ：{Number} - 如果存在的话，它应该是一种或多种被支持的 `getaddrinfo` 标志。如果没有提供 `hints` ，那么没有标志会传递给 `getaddrinfo` 。给 `hints` 的多个标志通过逻辑或（OR）运算的值传递。查阅支持的 getaddrinfo 标志 获取更多有关受支持的标志信息。

- `all` ：{Boolean} 当为 `true` 时，回调函数会在一个数组里返回所有解析后的地址，否则返回一个单一地址。默认为 `false` 。

所有属性都是可选的。选项的使用示例如下所示。

```
{
    family: 4,
    hints: dns.ADDRCONFIG | dns.V4MAPPED,
    all: false
}
```

`callback` 函数有参数 `(err, address, family)` 。 `address` 是一个 IPv4 或 IPv6 地址的字符串表示。 `family` 要么是整数 `4` 或 `6` 并代表着 `address` 族（不一定是最初传给 `lookup` 的值）。

若将选项 `all` 设为 `true` ，参数变为 `(err, addresses)` ， `addresses` 变成一组由 `address` 和 `family` 属性组成的对象。

当发生错误时， `err` 是一个 Error 对象，其中 `err.code` 是错误码。请记住 `err.code` 被设定为 `'ENOENT'` 的情况不局限于域名不存在，也可能在以其他方式查找失败，比如没有可用文件描述符时。

`dns.lookup()` 不一定要在 DNS 协议上做些什么。使用一个操作系统工具的实现，可以与地址名称相关联，反之亦然。此实现可能对任何 Node.js 程序的行为产生微妙但重要的后果。请在使用 `dns.lookup()` 前花一些时间来查阅实现中的注意事项章节。

支持的 getaddrinfo 标志

以下标志可以传递给 dns.lookup() 作为提示：

- `dns.ADDRCONFIG` ：返回的地址类型是由当前系统所支持的地址类型决定的。例如，IPv4 地址仅在当前系统至少配置了一个 IPv4 地址时返回。环回地址不考虑。

- `dns.V4MAPPED` ：如果指定了 IPv6 族，但没有找到 IPv6 地址。那么返回映射到 IPv4 的 IPv6 地址。需要注意的是，它不支持某些操作系统（如，FreeBSD 的 10.1 版本）。

dns.lookupService(address, port, callback)

使用 `getnameinfo` 的操作系统的底层实现来解析给定的 `address` 和 `port` 为主机名和服务器。

`callback` 函数有参数 `(err, hostname, service)`。`hostname` 和 `service` 参数是字符串（例如，分别为 `'localhost'` 和 `'http'`）。

当发生错误时，`err` 是一个 Error 对象，其中 `err.code` 是错误码。

```
const dns = require('dns');
dns.lookupService('127.0.0.1', 22, (err, hostname, service) => {
    console.log(hostname, service);
    // Prints: localhost ssh
});
```

错误代码

每个 DNS 查询都可以返回下列之一的错误代码：

- `dns.NODATA` ：DNS 服务器返回没有数据的应答。
- `dns.FORMERR` ：DNS 服务器查询所要求的格式不正确。
- `dns.SERVFAIL` ：DNS 服务器返回一般故障。
- `dns.NOTFOUND` ：未找到域名。
- `dns.NOTIMP` ：DNS 服务器不执行请求的操作。
- `dns.REFUSED` ：DNS 服务器拒绝查询。
- `dns.BADQUERY` ：格式错误的 DNS 查询。
- `dns.BADNAME` ：格式错误的主机名。
- `dns.BADFAMILY` ：不支持的地址族。
- `dns.BADRESP` ：格式错误的 DNS 应答。
- `dns.CONNREFUSED` ：无法连接到 DNS 服务器。
- `dns.TIMEOUT` ：联系 DNS 服务器超时。
- `dns.EOF` ：文件结尾。
- `dns.FILE` ：读取文件错误。
- `dns.NOMEM` ：内存不足。
- `dns.DESTRUCTION` ：通道被销毁。
- `dns.BADSTR` ：格式错误的字符串。
- `dns.BADFLAGS` ：指定的非法标识。
- `dns.NONAME` ：主机名不是数字。
- `dns.BADHINTS` ：指定的非法提示标识。
- `dns.NOTINITIALIZED` ：还未初始化 c-ares 库。
- `dns.LOADIPHLPAPI` ：加载 iphlpapi.dll 失败。
- `dns.ADDRGETNETWORKPARAMS` ：无法找到获取网络参数的函数。

- `dns.CANCELLED`：取消 DNS 查询。

实现中的注意事项

尽管 dns.lookup() 和每个 `dns.resolve*()/dns.reverse()` 函数都具有网络名称与网络地址相关联的同一个目标（或反之亦然），但是他们的行为却是完全不同的。这些差异可能对 Node.js 程序的行为产生微妙而显著的后果。

dns.lookup()

dns.lookup() 的原理与其他使用相同的操作系统设备的程序一样。例如，dns.lookup() 几乎总是像 `ping` 命令一样的方式解析给定的名称。在大多数类 POSIX 的操作系统中，dns.lookup() 函数的行为可以通过改变 `nsswitch.conf(5)` 和/或 `resolv.conf(5)` 的设置进行修改，但请注意，更改这些文件将改变同一个操作系统上运行的所有其他程序的行为。

尽管从 JavaScript 的角度来看，调用 `dns.lookup()` 会是异步，它通过同步调用运行在 libuv 线程池中的 `getaddrinfo(3)` 实现。由于 libuv 线程池的大小是固定的，这也意味着，如果出于某种原因 `getaddrinfo(3)` 的调用花费了很长时间，其他可能在 libuv 线程池（诸如文件系统操作）中运行的操作将经历性能下降。为了缓解这一问题，一个潜在的解决方案是通过将 'UV_THREADPOOL_SIZE' 环境变量设置为一个大于 4（当前的默认值）的值来增加 libuv 线程池的大小。有关 libuv 线程池的更多信息，请参阅 libuv 的官方文档。

dns.resolve(), dns.resolve*() and dns.reverse()

这些函数与 dns.lookup() 有着完全不同的实现。它们不使用 `getaddrinfo(3)` 并且它们总是执行网络上的 DNS 查询。该网络通信始终异步进行，并且不使用 libuv 线程池。

其结果是，这些函数不能与发生在（可以有 dns.lookup() 的） libuv 线程池中的其他处理具有相同的负面影响。

它们使用与 dns.lookup() 不同的配置文件。例如，他们不使用来自 `/etc/hosts` 的配置。

TLS/SSL(TLS/SSL)

> 稳定度：2 - 稳定

使用 `require('tls')` 访问这个模块。

`tls` 模块使用 OpenSSL 提供传输层安全性和/或安全套接字层：加密流传输。

TLS/SSL 是一个公开/私有密钥的底层结构。每个客户端和每个服务器必须有一个私钥。创建的私钥是这样的：

```
openssl genrsa -out ryans-key.pem 2048
```

所有服务器和一些客户端需要有一个证书。证书是由证书颁发机构签署或自签名的公钥。第一步，获得证书是创建一个“证书签名请求”（CSR）文件。通过这样做：

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

要创建一个 CSR 自签名证书，可以这样做：

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

另外，你可以发送 CSR 到证书颁发机构进行签名。

对于完全正向加密，需要生成 Diffie-Hellman 参数：

```
openssl pkcs12 -export -in agent5-cert.pem -inkey agent5-key.pem -certfile ca-cert.pem
 -out agent5.pfx
```

- `in`：证书
- `inkey`：私钥
- `certfile`：所有的 CA 证书串连在一个文件中，如 `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

方法和属性

---

tls.createServer(options[, secureConnectionListener])

创建一个新的 tls.Server。`connectionListener` 参数自动设置为 'secureConnection' 事件的监听器。`options` 对象可能包含以下字段：

- `pfx` ：一个字符串或 `Buffer` ，包含服务器端 PFX 或 PKCS12 格式的私钥、证书和 CA 证书（ `key` 、 `cert` 和 `ca` 选项互斥）。
- `key` ：一个字符串或 `Buffer` ，包含服务器端 PEM 格式的私钥。为了支持多个私钥使用不同算法。它可以是一组普通的私匙或一组 `{pem: key, passphrase: passphrase}` 格式的对象。（必需）
- `passphrase` ：一个包含私钥 或 pfx 口令的字符串。
- `cert` ：一个字符串、 `Buffer` 、一组字符串或一组 `Buffer` ，包含 PEM 格式的服务器证书密钥。（必需）
- `ca` ：一个字符串、 `Buffer` 、一组字符串或一组 `Buffer` ，PEM 格式的受信任的证书。如果省略，则会使用几个著名的“根”的 CA（例如 VeriSign）。这些用于授权的连接。
- `crl` ：一个字符串或一组 PEM 编码的 CRLs（证书吊销列表）的字符串。
- `ciphers` ：描述要使用或排除的加密字符串，使用 `:` 分隔。默认的加密套件是：

```
ECDHE-RSA-AES128-GCM-SHA256:
ECDHE-ECDSA-AES128-GCM-SHA256:
ECDHE-RSA-AES256-GCM-SHA384:
ECDHE-ECDSA-AES256-GCM-SHA384:
DHE-RSA-AES128-GCM-SHA256:
ECDHE-RSA-AES128-SHA256:
DHE-RSA-AES128-SHA256:
ECDHE-RSA-AES256-SHA384:
DHE-RSA-AES256-SHA384:
ECDHE-RSA-AES256-SHA256:
DHE-RSA-AES256-SHA256:
HIGH:
!aNULL:
!eNULL:
!EXPORT:
!DES:
!RC4:
!MD5:
!PSK:
!SRP:
!CAMELLIA
```

默认加密套件偏好 GCM 加密，Chrome 的“现代密码学”设置，同时也偏好 ECDHE 和 DHE 加密，完全正向加密，同时提供一些向后兼容性。

128 位 AES 优于 192 和 256 位 AES，特定攻击对更大的 AES 密钥大小影响更小。

老的客户端依靠不安全、不推荐使用的 RC4 或基于 DES 的加密算法（如 Internet Explorer 6）是无法完成默认配置的握手过程。如果客户端必须支持这些，TLS 建议可以提供一个兼容的加密套件。有关格式方面的更多细节，详见 OpenSSL 加密列表格式文档。

- `ecdhCurve`：描述了一个名为曲线的字符串用于 ECDH 密钥协议或使用 `false` 禁用 ECDH。

默认为 `prime256v1`（NIST P-256）。使用 crypto.getCurves() 来获得可用的曲线名称列表。在最近的版本中，`openssl ecparam -list_curves` 也将显示名称和每个可用椭圆曲线的描述。

- `dhparam`：一个字符串或 `Buffer`，包含 Diffie Hellman 参数，需要完全正向加密。使用 `openssl dhparam` 来创建它。它的密钥长度应大于或等于 1024 位，否则它将抛出一个错误。强烈建议使用 2048 位或以上来加强安全性。如果省略或无效，它被静默丢弃并且 DHE 加密将无法使用。

- `handshakeTimeout`：如果 SSL/TLS 握手无法在指定毫秒数内完成将会中止连接。默认值是 120 秒。

每当一个握手超时，就会在 `tls.Server` 对象上发出一个 `'clientError'` 事件。

- `honorCipherOrder`：当选择一个加密时，使用服务器端偏好代替客户端偏好。默认为 `true`。
- `requestCert`：如果为 `true`，服务器将请求所连接的客户端的证书，并尝试验证证书。默认为 `false`。
- `rejectUnauthorized`：如果为 `true`，服务器将拒绝未在提供的授权 CA 列表中的任何连接。此选项仅当 `requestCert` 为 `true` 时有效。默认为 `false`。
- `NPNProtocols`：一个可能的 NPN 协议数组或 `Buffer` s。（协议应该由它们的优先级进行排序。）
- `ALPNProtocols`：一个可能的 ALPNP 协议数组或 `Buffer` s。（协议应该由它们的优先级进行排序。）当服务器从客户端同时接收 NPN 和 ALPN 扩展时，ALPN 优先于 NPN 并且服务器不会发送一个 NPN 扩展到客户端。
- `SNICallback(servername, cb)`：如果客户端支持 SNI TLS 扩展，函数将被调用。两个参数将被传递给它：`servername` 和 `cb`。`SNICallback` 应该调用 `cb(null, ctx)`，其中，ctx 一个是 `SecureContext` 实例。（`tls.createSecureContext(...)` 可以用来得到适当的 `SecureContext`）。如果未提供 `SNICallback`，将使用带有高级 API 的默认回调（见下文）。
- `sessionTimeout`：一个整数，指定在服务器创建 TLS 会话标识符之后的秒数并且服务器创建 TLS 会话凭证将超时。详见 SSL_CTX_set_timeout 了解更多细节。
- `ticketKeys`：一个 48 字节的 `Buffer` 实例由一个 16 字节的前缀，16 字节的 HMAC 密钥，和一个 16 字节的 AES 密钥组成。这可以用来接受多个 TLS 服务器上的 TLS 会话凭证。

注意：`cluster` 模块的工作进程之间自动共享。

- `sessionIdContext`：一个字符串包含一个用于会话恢复的不透明的标识符。如果 `requestCert` 为 `true`。默认值是来自命令行产生一个 MD5 哈希值。（在 FIPS 模式中用一个截断的 SHA1 哈希代替。）否则，不提供默认值。
- `secureProtocol`：要使用的 SSL 方法，如，`SSLv3_method` 强制 SSL 版本 3。可能的值取决于你安装的 OpenSSL 并定义不变的 SSL_METHODS。

下面是一个简单的响应服务器例子：

```
const tls = require('tls');
const fs = require('fs');

const options = {
    key: fs.readFileSync('server-key.pem'),
    cert: fs.readFileSync('server-cert.pem'),

    // This is necessary only if using the client certificate authentication.
    requestCert: true,

    // This is necessary only if the client uses the self-signed certificate.
    ca: [fs.readFileSync('client-cert.pem')]
};

var server = tls.createServer(options, (socket) => {
    console.log('server connected',
        socket.authorized ? 'authorized' : 'unauthorized');
    socket.write('welcome!\n');
    socket.setEncoding('utf8');
    socket.pipe(socket);
});
server.listen(8000, () => {
    console.log('server bound');
});
```

或

```
const tls = require('tls');
const fs = require('fs');

const options = {
    pfx: fs.readFileSync('server.pfx'),

    // This is necessary only if using the client certificate authentication.
    requestCert: true,

};

var server = tls.createServer(options, (socket) => {
    console.log('server connected',
        socket.authorized ? 'authorized' : 'unauthorized');
    socket.write('welcome!\n');
    socket.setEncoding('utf8');
    socket.pipe(socket);
});
server.listen(8000, () => {
    console.log('server bound');
});
```

你可以通过使用 `openssl s_client` 连接到这台服务器来做测试：

```
openssl s_client -connect 127.0.0.1:8000
```

tls.connect(options[, callback])

tls.connect(port[, host][, options][, callback])

使用给定的 `port` 和 `host`（老 API）或 `options.port` 和 `options.host` 创建一个新的客户端连接。（如果省略 `host`，它默认为 `localhost`。）`options` 应该是一个可以指定以下属性的对象：

- `host`：客户端应该连接到主机。
- `port`：客户端应该连接到端口。
- `socket`：在一个给定的套接字上建立安全连接，而不是创建一个新的套接字。如果指定了此选项，会忽略 `host` 和 `port`。
- `path`：创建 Unix 套接字连接路径。如果指定了此选项，会忽略 `host` 和 `port`。
- `pfx`：一个字符串或 `Buffer`，包含服务器端 PFX 或 PKCS12 格式的私钥、证书和 CA 证书。
- `key`：一个字符串、`Buffer`、一组字符串或一组 `Buffer`，包含 PEM 格式的客户端私钥。
- `passphrase`：一个包含私钥或 pfx 口令的字符串。
- `cert`：一个字符串、`Buffer`、一组字符串或一组 `Buffer`，包含 PEM 格式的客户端证书密钥。
- `ca`：一个字符串、`Buffer`、一组字符串或一组 `Buffer`，PEM 格式的受信任的证书。如果省略，则会使用几个著名的“根”的 CA（例如 VeriSign）。这些用于授权的连接。
- `ciphers`：描述要使用或排除的加密字符串，使用 `:` 分隔。默认的加密套件和 tls.createServer() 相同。
- `rejectUnauthorized`：如果为 `true`，服务器将拒绝未在提供的授权 CA 列表中的任何连接。如果验证失败，会发出一个 `'error'` 事件；`err.code` 包含 OpenSSL 错误码。默认为 `true`。
- `NPNProtocols`：一个可能的 NPN 协议字符串数组或 `Buffer`s。`Buffer`s 应该有以下格式：`0x05hello0x05world`，其中第一个字节是下一个协议的名称的长度。（传递一个数组通常会简单得多：`['hello', 'world']`）

- `ALPNProtocols`：一个可能的 ALPNP 协议字符串数组或 `Buffer`s。`Buffer`s 应该有以下格式：`0x05hello0x05world`，其中第一个字节是下一个协议的名称的长度。（传递一个数组通常会简单得多：`['hello', 'world']`）
- `servername`：用于 SNI（服务器名称指示）TLS 扩展的服务器名称。
- `checkServerIdentity(servername, cert)`：对服务器名称针对的证书提供覆盖检查。如果验证失败应该返回一个错误。如果通过，返回 `undefined`。
- `secureProtocol`：要使用的 SSL 方法，如，`SSLv3_method` 强制 SSL 版本 3。可能的值取决于你安装的 OpenSSL 并定义不变的 SSL_METHODS。
- `secureContext`：从 `tls.createSecureContext( ... )` 获取的一个可选的 TLS 上下文对象。它可以被用于缓存客户端证书、密钥和 CA 证书。
- `session`：一个 `Buffer` 实例，包含 TLS 会话。
- `minDHSize`：可接受的 TLS 连接的 DH 参数的最小值（以“位”为单位）。当一台服务器提供了一个尺寸小于该值的 DH 参数，该 TLS 连接会被销毁，并抛出错误。默认：`1024`。

`callback` 参数会被添加作为 'secureConnect' 事件的一个监听器。

`tls.connect()` 返回一个 tls.TLSSocket 对象。

这里有一个之前描述的响应服务器的客户端例子：

```
const tls = require('tls');
const fs = require('fs');

const options = {
    // These are necessary only if using the client certificate authentication
    key: fs.readFileSync('client-key.pem'),
    cert: fs.readFileSync('client-cert.pem'),

    // This is necessary only if the server uses the self-signed certificate
    ca: [fs.readFileSync('server-cert.pem')]
};

var socket = tls.connect(8000, options, () => {
    console.log('client connected',
        socket.authorized ? 'authorized' : 'unauthorized');
    process.stdin.pipe(socket);
    process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', (data) => {
    console.log(data);
});
socket.on('end', () => {
    server.close();
});
```

或

```
const tls = require('tls');
const fs = require('fs');

const options = {
    pfx: fs.readFileSync('client.pfx')
};

var socket = tls.connect(8000, options, () => {
    console.log('client connected',
        socket.authorized ? 'authorized' : 'unauthorized');
    process.stdin.pipe(socket);
    process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', (data) => {
    console.log(data);
});
socket.on('end', () => {
    server.close();
});
```

tls.createSecureContext(options)

创建一个凭据对象；该 `options` 对象可能包含以下字段：

- `pfx` ：一个字符串或 `Buffer` ，包含服务器端 PFX 或 PKCS12 格式的私钥、证书和 CA 证书。
- `key` ：一个字符串或 `Buffer` ，包含服务器端 PEM 格式的私钥。为了支持多个私钥使用不同算法。它可以是一组普通的私匙或一组 `{pem: key, passphrase: passphrase}` 格式的对象。（必需）
- `passphrase` ：一个包含私钥 或 pfx 口令的字符串。
- `cert` ：一个包含 PEM 编码的证书。
- `ca` ：一个字符串、 `Buffer` 、一组字符串或一组 `Buffer` ，PEM 格式的受信任的证书。如果省略，则会使用几个著名的“根”的 CA（例如 VeriSign）。这些用于授权的连接。
- `crl` ：一个字符串或一组 PEM 编码的 CRLs（证书吊销列表）的字符串。
- `ciphers` ：一个描述使用或排除的加密方式的字符串。在 https://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT 上参考格式细节。
- `honorCipherOrder` ：当选择一种加密时，请使用服务器的偏好，而不是客户端的偏好。详见 `tls` 模块文档。

如果没有给出 'CA' 细节，那么 Node.js 就会使用 CA 在 http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt 上给出的默认公众信任列表。

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])

创建一个新的具有两个流的安全对对象，其中一个读写经过加密数据和另一个读写明文数据。通常，已加密流通过管道输送到加密数据流，或从加密数据流输入并且明文被用作最初加密流的替代品。

- `credentials` ：从 `tls.createSecureContext( ... )` 获取的一个安全的上下文对象。
- `isServer` ：布尔值，指示此 TLS 连接是否应该被打开作为服务器或客户端。
- `requestCert` ：布尔值，指示服务器是否应该从客户端连接请求证书。只适用于服务器的连接。

- `rejectUnauthorized`：布尔值，指示服务器是否应自动拒绝客户提供无效证书。只适用于服务器 `requestCert` 可用时。
- `options`：普通的 SSL 选项对象。详见 tls.TLSSocket。

`tls.createSecurePair()` 返回一个安全对对象，带有 `cleartext` 和 `encrypted` 流属性。

注意：`cleartext` 和 tls.TLSSocket 有着相同的 API。

tls.getCiphers()

返回一个支持的 SSL 加密的名称数组。

示例：

```
var ciphers = tls.getCiphers();
console.log(ciphers); // ['AES128-SHA', 'AES256-SHA', ...]
```

tls.TLSSocket类

这是一个包装过的 net.Socket 版本，对写入的数据透明加密并且都需要 TLS 协商。

这个实例实现了全双工流接口。它具有普通流所有的方法和事件。

方法返回 TLS 连接元（如，当连接打开时，tls.TLSSocket.getPeerCertificate() 只会返回数据。）。

new tls.TLSSocket(socket[, options])

从现有的 TCP 套接字构造一个新的 TLSSocket 对象。

`socket` 是一个 net.Socket 实例。

`options` 是可能包含以下属性的一个可选对象：

- `secureContext`：一个源自 tls.createSecureContext() 可选的 TLS 上下文对象。

- `isServer`：如果为 `true`，TLS 套接字将在服务器模式下被实例化。默认：`false`。
- `server`：一个可选的 net.Server 实例。
- `requestCert`：可选，详见 tls.createSecurePair()。
- `rejectUnauthorized`：可选，详见 tls.createSecurePair()。
- `NPNProtocols`：可选，详见 tls.createServer()。
- `ALPNProtocols`：可选，详见 tls.createServer()。
- `SNICallback`：可选，详见 tls.createServer()。
- `session`：可选，一个 `Buffer` 实例，包含一个 TLS 会话。
- `requestOCSP`：可选，如果为 `true`，OCSP 状态请求扩展将被添加到客户端问候并在建立安全通信之前的套接字上发出一个 `'OCSPResponse'` 事件。

'secureConnect' 事件

这个事件会在新连接的握手过程已成功完成后发出。监听时将忽略服务器证书是否已被授权调用。测试 `tlsSocket.authorized` 是否是服务器证书是由指定的 CA 之一签署的，是用户的责任。如果 `tlsSocket.authorized === false`，那么错误可以在 `tlsSocket.authorizationError` 中找到。同样，如果使用了 `ALPN` 或 `NPN`，可以用 `tlsSocket.alpnProtocol` 或 `tlsSocket.npnProtocol` 检查协商的协议。

'OCSPResponse' 事件

```
function (response) { }
```

如果设置了 `requestOCSP` 选项将触发此事件。`response` 是一个包含服务器 OCSP 响应的 `Buffer`。

传统上，`response` 是一个来自服务器的 CA 的签名对象，包含有关服务器证书吊销状态的信息。

tlsSocket.localAddress

本地 IP 地址的字符串表示。

tlsSocket.localPort

本地端口的数字表示。

tlsSocket.remoteFamily

远程 IP 族的字符串表示。`'IPv4'` 或 `'IPv6'`。

tlsSocket.remoteAddress

远程 IP 地址的字符串表示。例如，`'74.125.127.100'` 或 `'2001:4860:a005::68'`。

tlsSocket.remotePort

远程端口的数字表示。例如，`443`。

tlsSocket.authorized

布尔值，如果对方的证书是由指定的 CA 之一签署的，则为 `true`，否则为 `false`。

tlsSocket.authorizationError

为什么对方的证书未能通过验证的原因。这个属性只在 `tlsSocket.authorized === false` 时可用。

tlsSocket.encrypted

静态布尔值，总是 `true`，可用来区分那些常规的 TLS 套接字。

tlsSocket.address()

返回由操作系统所报告的绑定地址、地址族名称和服务器端口号。返回一个包含三个属性的对象，即，`{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`。

tlsSocket.getTLSTicket()

注意：仅适用于客户端的 TLS 套接字。有用仅用于调试，用于会话重用提供给 tls.connect() 的 `session` 选项。

返回 TLS 会话凭证，如果没有进行协商，则返回 `undefined` 。

tlsSocket.getPeerCertificate([detailed])

返回表示对等体的证书对象。返回的对象有一些相对于证书字段的属性。如果 `detailed` 是 `true` ，将会返回完整的带有 `issuer` 属性的链，如果为 `false` ，只会返回不带 `issuer` 属性的顶级证书。

示例：

```
{
    subject: {
        C: 'UK',
        ST: 'Acknack Ltd',
        L: 'Rhys Jones',
        O: 'node.js',
        OU: 'Test TLS Certificate',
        CN: 'localhost'
    },
    issuerInfo: {
        C: 'UK',
        ST: 'Acknack Ltd',
        L: 'Rhys Jones',
        O: 'node.js',
        OU: 'Test TLS Certificate',
        CN: 'localhost'
    },
    issuer: {...another certificate...
    },
    raw: < RAW DER buffer > ,
    valid_from: 'Nov 11 09:52:22 2009 GMT',
    valid_to: 'Nov  6 09:52:22 2029 GMT',
    fingerprint: '2A:7A:C2:DD:E5:F9:CC:53:72:35:99:7A:02:5A:71:38:52:EC:8A:DF',
    serialNumber: 'B9B0D332A1AA5635'
}
```

如果对等体不提供证书，它将返回 `null` 或一个空对象。

tlsSocket.getCipher()

返回表示加密名称和 SSL/TLS 协议版本中定义的第一种加密方式的对象。

示例： `{ name: 'AES256-SHA', version: 'TLSv1/SSLv3' }` 。

详见 SSL_CIPHER_get_name() 和 SSL_CIPHER_get_version() 了解更多信息。

tlsSocket.getEphemeralKeyInfo()

返回一个表示在完全正向加密的一个客户端连接上的临时密钥交换的参数的类型、名称和大小的对象。当密钥交换不是临时的，它将返回一个空对象。由于这只支持客户端的套接字，如果在服务器端的套接字上调用，则返回 `null`。支持的类型有 `'DH'` 和 `'ECDH'`。`name` 属性只在 `'ECDH'` 中有效。

示例：

```
{ type: 'ECDH', name: 'prime256v1', size: 256 }
```

tlsSocket.getProtocol()

返回包含当前连接协商的 SSL/TLS 协议版本的字符串。对于连接尚未完成握手过程的套接字会返回 `'unknown'`。对于服务器套接字或断开的客户端套接字会返回 `null`。

示例：

```
'SSLv3'
'TLSv1'
'TLSv1.1'
'TLSv1.2'
'unknown'
```

详见 https://www.openssl.org/docs/manmaster/ssl/SSL_get_version.html 了解更多详情。

tlsSocket.getSession()

返回 ASN.1 编码的会话，如果没有进行协商，则返回 `undefined`。当重新连接到服务器时，可以用于加速握手建立。

tlsSocket.renegotiate(options, callback)

启动 TLS 重新协商过程。`options` 对象可能包含以下字段：`rejectUnauthorized`、`requestCert`。（详见 tls.createServer() 了解更多细节。）。一旦重新成功完成协商，`callback(err)` 会用 `null` 代替 `err` 执行。

注意：可用于在安全连接建立后请求对等体的证书。

另外需要注意：当作为服务器运行时，在 `handshakeTimeout` 超时后，套接字会带着错误被销毁。

tlsSocket.setMaxSendFragment(size)

设置最大的 TLS 片段大小（默认和最大值是：`16384`，最小值是：`512`）。当成功时，返回 `true`，否则返回 `false`。

较小的片段大小可以降低客户端上的缓冲延迟：较大的片段通过 TLS 层缓冲，直到接收到整个片段并验证其完整性；大片段可以跨越多个往返，并且由于数据包丢失或重新排序，其处理可能被延迟。然而，更小的片段会增加额外的 TLS 帧字节和 CPU 开销，这可能会降低服务器的整体吞吐量。

tls.Server类

这个类是 net.Server 的子类，并且有着相同的方法。而不是只接受原始的TCP连接，它接收使用 TLS 或 SSL 加密的连接。

'secureConnection' 事件

`function (tlsSocket) {}`

在一个新的连接握手过程已成功完成后发出此事件。该参数是一个 tls.TLSSocket 实例并且拥有所有的公共流方法和事件。

`socket.authorized` 是一个布尔值，这表明客户端是否已被提供的证书颁发机构的服务器验证。如果 `socket.authorized` 是 `false`，那么 `socket.authorizationError` 被设定为描述授权如何失败。隐晦但值得一提：这取决于 TLS 服务器的设置，未经授权的连接可能被接受。

`socket.npnProtocol` 是包含所选 NPN 协议的字符串，同时 `socket.alpnProtocol` 是包含所选 ALPN 协议的字符串。当同时收到这两个 NPN 和 ALPN 扩展时，ALPN 优先于 NPN 并且 ALPN 将选择下一个协议。当 ALPN 没有已选择的协议时，这会返回 `false`。

`socket.servername` 是含有 SNI 请求的服务器名称的字符串。

'newSession' 事件

`function (sessionId, sessionData, callback) { }`

在建立一个 TLS 会话时发出。可以用来在外部存储中存储会话。最终必须调用 `callback`，否则没有数据将被发送或从安全连接中接收。

注意：添加这个事件监听器只对这之后建立的连接有影响。

'resumeSession' 事件

`function (sessionId, callback) { }`

当客户想要恢复以前的 TLS 会话时发出。该事件监听器可以使用给定的 `sessionId` 在外部存储中执行查找。并在完成时调用一次 `callback(null, sessionData)`。如果无法恢复会话（即，不存在于存储中），可能会调用 `callback(null, null)`。调用 `callback(err)` 将终止传入的连接并销毁套接字。

注意：添加这个事件监听器只对这之后建立的连接有影响。

下面是使用 TLS 会话恢复的例子：

```
var tlsSessionStore = {};
server.on('newSession', (id, data, cb) => {
    tlsSessionStore[id.toString('hex')] = data;
    cb();
});
server.on('resumeSession', (id, cb) => {
    cb(null, tlsSessionStore[id.toString('hex')] || null);
});
```

'OCSPRequest' 事件

`function (certificate, issuer, callback) { }`

当客户端发送一个证书状态请求时发出。可以解析服务器的当前证书来获得 OCSP URL 和证书编号；在获得一个 OCSP 响应之后，之后调用 `callback(null, resp)`，其中 `resp` 是一个 `Buffer` 实例。`certificate` 和 `issuer` 都是 `Buffer` 形式的主要和发行人证书的 DER 表示。它们可被用于获得 OCSP 证书的 ID 和 OCSP 端点 URL。

另外，可以调用 `callback(null, null)`，这意味着没有 OCSP 响应。

调用 `callback(err)`，会导致调用 `socket.destroy(err)`。

典型流程：

1. 客户端连接到服务器并向它发送一个 `'OCSPRequest'`（通过 ClientHello 上的状态信息扩展）。

2. 服务器收到请求并调用 `'OCSPRequest'` 事件监听器，如果存在的话。
3. 服务器从 `certificate` 或 `issuer` 提取 OCSP URL 并向 CA 执行 OCSP 请求。
4. 服务器从 CA 接收 `OCSPResponse` 并通过 `callback` 参数将其发送回客户端。
5. 客户端验证响应，要么销毁套接字，要么执行握手。

注意：如果证书是自签名或发行人不在根证书列表中时，`issuer` 不能是 `null`。（发行人可以通过 `ca` 选项提供。）

注意：添加这个事件监听器只对这之后建立的连接有影响。

注意：像 asn1.js 这样的 npm 模块可用于解析证书。

'clientError' 事件

`function (exception, tlsSocket) { }`

当一个客户端连接在建立安全连接之前发出一个 `'error'` 事件时，它将在这里转发。

`tlsSocket` 是源自 tls.TLSSocket 的错误。

server.connections

在服务器上的并行连接数。

server.maxConnections

将此属性设置为，当服务器的连接数超过指定的阈值时拒绝连接。

server.listen(port[, hostname][, callback])

在指定的 `port` 和 `hostname` 上开始接受连接。如果省略 `hostname`，当 IPv6 可用时，服务器将接受任何 IPv6 地址（`::`），否则将接收任何 IPv4 地址（`0.0.0.0`）。零端口值将被分配一个随机端口。

这个函数是异步的。最后的参数 `callback` 在服务器被绑定时调用。

详见 `net.Server` 了解更多详情。

server.addContext(hostname, context)

如果客户端请求的主机名 SNI 所提供的 `hostname` 匹配（可以使用通配符），添加将被使用的安全上下文。`context` 可以包含 `key`、`cert`、`ca` 或来自 tls.createSecureContext() 中 `options` 参数的任何其他属性。

server.setTicketKeys(keys)

升级用于 TLS 会话凭证 加密/解密的密钥。

注意：buffer 应该是 48 个字节长度。详见在 tls.createServer 中的 `ticketKeys` 选项了解更多如何使用的信息。

注意：该变化仅对之后的服务器连接有效。现有的或目前正在等待的服务器的连接将使用之前的密钥。

server.getTicketKeys()

返回一个保存有目前用于 TLS 会话凭证 加密/解密的密钥的 `Buffer` 实例。

server.close([callback])

服务器停止接受新的连接。该函数是异步的，当服务器发出一个 `'close'` 事件时，该服务器最终关闭。你可以传递一个可选的回调函数到 `'close'` 事件的监听器中。

server.address()

返回由操作系统所报告的绑定地址、地址族名称和服务器端口号。详见 net.Server.address() 了解更多信息。

CryptoStream类

> 稳定度：0 - 已废弃：使用 tls.TLSSocket 替代。

这是一个加密流。

cryptoStream.bytesWritten

一个底层套接字的 bytesWritten 访问器代理，这会返回写入到套接字的总字节数，包括 TLS 开销。

SecurePair类

通过 tls.createSecurePair 返回。

'secure' 事件

一旦该安全对已经成功建立安全的连接，这个事件会从 SecurePair 发出。

作为检测服务器端的 secureConnection 事件的手段，`pair.cleartext.authorized` 应检查并确认使用的证书是否被正确授权。

ALPN、NPN和SNI

ALPN（应用层协议协商拓展）、NPN（下一个协议协议协商）和 SNI（服务器名称表示）是 TLS 握手扩展：

- ALPN/NPN - 允许多种协议使用一个 TLS 服务器（HTTP、SPDY、HTTP/2）。
- SNI - 允许具有不同 SSL 证书的多个主机名使用一个 TLS 服务器。

PFS(完全正向加密)

术语“正向加密”或“完全正向加密”描述密钥协议（即，密钥交换）方法的一个特点。实际上这意味着即使一台服务器的私有密钥被泄露，如果他们设法获得专门为每个会话生成的密钥对，通信只能由窃听者进行解密。

这是通过为每个握手密钥协定随机生成密钥对（与使用所有会话相同的密钥）来实现的。实现该技术的方法被叫做“ephemeral”，从而提供完全正向加密。

目前这两种方法常用于实现完全正向加密（注意字符“E”追加到传统的缩写）：

- DHE：一个 ephemeral 版本的 Diffie Hellman 密钥协商协议。
- ECDHE：一个 ephemeral 版本的 Elliptic Curve Diffie Hellman 密钥协商协议。

Ephemeral 方法可能有一些性能缺陷，因为密钥生成是昂贵的。

修改TLS的默认加密方式

Node.js 建立了一套默认的启用和禁用 TLS 加密套件。当前，默认的加密套件是：

```
ECDHE-RSA-AES128-GCM-SHA256:
ECDHE-ECDSA-AES128-GCM-SHA256:
ECDHE-RSA-AES256-GCM-SHA384:
ECDHE-ECDSA-AES256-GCM-SHA384:
DHE-RSA-AES128-GCM-SHA256:
ECDHE-RSA-AES128-SHA256:
DHE-RSA-AES128-SHA256:
ECDHE-RSA-AES256-SHA384:
DHE-RSA-AES256-SHA384:
ECDHE-RSA-AES256-SHA256:
DHE-RSA-AES256-SHA256:
HIGH:
!aNULL:
!eNULL:
!EXPORT:
!DES:
!RC4:
!MD5:
!PSK:
!SRP:
!CAMELLIA
```

默认值完全可以通过 `--tls-cipher-list` 命令行指令进行覆盖。例如，以下操作使得 `ECDHE-RSA-AES128-GCM-SHA256:!RC4` 成为默认 TLS 加密套件：

```
node --tls-cipher-list="ECDHE-RSA-AES128-GCM-SHA256:!RC4"
```

请注意，包含在 Node.js 内的默认加密套件是经过精心挑选的，反映了当前的安全最佳实践并降低了风险。改变默认的加密套件会对一个应用程序的安全性产生显著影响。 `--tls-cipher-list` 指令应该只有在有绝对必要的情况下使用。

缓解由客户端发起的重新协商攻击

TLS 协议允许客户端重新协商 TLS 会话的某些方面。不幸的是，会话重新协商需要非等比的服务器端的资源，这使得它成为拒绝服务攻击的潜在媒介。

为了减轻这个的影响，重新协商被限制为每 10 分钟三次。当超过该阈值时，tls.TLSSocket 实例会发出一个错误。这些限制是可配置的：

- `tls.CLIENT_RENEG_LIMIT`：重新协商次数限制，默认为 3。
- `tls.CLIENT_RENEG_WINDOW`：重新协商窗口存活时间（以秒为单位），默认为 10 分钟。

不要在没有充分理解的情况下去修改这些默认参数。

测试服务器，使用 `openssl s_client -connect address:port` 连接并键入 `R<CR>`（即，小写的 `R` 后跟着回车键）几次。

HTTP(HTTP)

> 稳定度：2 - 稳定

使用 HTTP 服务器和客户端必须 `require('http')` 。

在 Node.js 中的 HTTP 接口被设计为支持使用该协议中与传统不同的许多功能。尤其是，大型的可能是块编码的消息。该接口谨慎或从不缓存整个请求或响应——用户能够以流形式传输数据。

HTTP消息报头由一个像这样的对象表示：

```
{
    'content-length': '123',
    'content-type': 'text/plain',
    'connection': 'keep-alive',
    'host': 'mysite.com',
    'accept': '*/*'
}
```

键是小写，值没去修改。

为了支持全部可能的 HTTP 应用，Node.js 的 HTTP API 非常低级。它只涉及流处理和消息解析。它解析消息为头部和身体，但它并不解析实际的头部或身体。

请参阅 message.headers 了解如何处理重复的头部。

收到的原始消息头被保留在 `rawHeaders` 属性中，它们是一个类似 `[key, value, key2, value2, ...]` 的数组。

例如，以前的消息头对象可能有类似以下的 `rawHeaders` 列表：

```
[
    'ConTent-Length', '123456',
    'content-LENGTH', '123',
    'content-type', 'text/plain',
    'CONNECTION', 'keep-alive',
    'Host', 'mysite.com',
    'accepT', '*/*'
]
```

方法和属性

---

http.METHODS

- {Array}

由该分析器支持的 HTTP 方法列表。

http.STATUS_CODES

- {Object}

所有标准的 HTTP 响应状态码的集合，以及各自的简短描述。例如，`http.STATUS_CODES[404] === 'Not Found'`。

http.globalAgent

全局代理的实例，默认作为所有 HTTP 客户端的请求。

http.createServer([requestListener])

返回一个新的 http.Server 实例。

`requestListener` 是一个自动添加到 `'request'` 事件中的函数。

http.createClient([port][, host])

> 稳定性：0 - 已废弃：使用 http.request() 代替。

构造一个新的 HTTP 客户端。`port` 和 `host` 指向被连接服务器。

http.request(options[, callback])

Node.js 维护每台服务器的多个连接来实现 HTTP 请求。该函数允许透明地发出请求。

`options` 可以是一个对象或字符串。如果 `options` 是一个字符串，它会自动使用 url.parse() 解析。

选项：

- `protocol` ：使用的协议。默认为 `'http:'` 。
- `host` ：发出请求的服务器域名或 IP 地址。默认为 `'localhost'` 。
- `hostname` ：`host` 的别名。以支持 url.parse() 解析 `hostname` 会优于 `host` 。
- `family` ：在解析 `hostname` 和 `host` 时所使用的 IP 地址族。有效值是 `4` 或 `6` 。当它是 `unspecified` 时，将同时使用 IPv4 和 IPv6。
- `port` ：远程服务器的端口。默认为 `80` 。
- `localAddress` ：绑定到网络连接的本地接口。
- `socketPath` ：Unix Domain Socket（使用 host:port 和 socketPath 的其中之一）
- `method` ：一个指定 HTTP 请求方法的字符串。默认为 `'GET'` 。
- `path` ：请求路径。默认为 `'/'` 。应包括查询字符串（如有的话）。如 `'/index.html?page=12'` 。当请求的路径中包含非法字符时，会引发异常。目前，只有空字符会被拒绝，但在将来可能会发生变化。
- `headers` ：一个包含请求头的对象。
- `auth` ：基本身份验证，如，`'user:password'` 来计算 Authorization 头。
- `agent` ：控制代理行为。如果使用代理请求，默认为 `Connection: keep-alive` 。可能的值：
  - `undefined` （默认）：对主机和端口使用 http.globalAgent。
  - `Agent` 对象：明确地使用代理。
  - `false` ：选择跳出连接池的代理，默认请求 `Connection: close` 。

- `createConnection`：当不使用 `agent` 信息选项时，产生一个用于请求的 socket/stream 的函数。这可以用于避免创建一个自定义的 Agent 类只是为了覆盖默认的 `createConnection` 函数。详见 agent.createConnection() 了解更多信息。

可选的 `callback` 参数会被添加为 `'response'` 事件的一次性监听器。

`http.request()` 返回一个 http.ClientRequest 类的实例。`ClientRequest` 实例是一个可写流。如果一个人想要通过 POST 请求上传一个文件，然后写入到 `ClientRequest` 对象。

示例：

```
var postData = querystring.stringify({
    'msg': 'Hello World!'
});

var options = {
    hostname: 'www.google.com',
    port: 80,
    path: '/upload',
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
        'Content-Length': postData.length
    }
};

var req = http.request(options, (res) => {
    console.log(`STATUS: ${res.statusCode}`);
    console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
    res.setEncoding('utf8');
    res.on('data', (chunk) => {
        console.log(`BODY: ${chunk}`);
    });
    res.on('end', () => {
        console.log('No more data in response.')
    })
});

req.on('error', (e) => {
    console.log(`problem with request: ${e.message}`);
});

// write data to request body
req.write(postData);
req.end();
```

请注意在示例中 `req.end()` 被调用。通过 `http.request()` 时，必须经常调用 `req.end()` 来表示你的请求已经结束 - 即使没有数据被写入请求主体。

如果请求过程中遇到任何错误（DNS 解析错误，TCP 级的错误，或实际的 HTTP 解析错误），在返回请求时，会发出 `'error'` 事件。对于所有的 `'error'` 事件而言，如果没有注册监听器，那么错误将被抛出。

这里有几个应该注意的特殊的头。

- 发送一个 `'Connection: keep-alive'` 将通知 Node.js 到服务器的连接，应一直持续到下一个请求。
- 发送一个 `'Content-length'` 头将禁用默认块编码。
- 发送一个 `'Expect'` 头会立即发送请求头。通常情况下，当发送 `'Expect: 100-continue'`，你应该设置超时并监听 `'continue'` 事件。见 RFC2616 第 8.2.3 节以获取更多信息。
- 发送 Authorization 头将覆盖使用 `auth` 选项计算基本身份验证。

http.get(options[, callback])

由于大多数 GET 请求都没有内容，Node.js 提供了这个便捷方法。该方法与 http.request() 的唯一区别是它设置该方法为 GET 并自动调用 `req.end()`。

示例：

```
http.get('http://www.google.com/index.html', (res) => {
    console.log(`Got response: ${res.statusCode}`);
    // consume response body
    res.resume();
}).on('error', (e) => {
    console.log(`Got error: ${e.message}`);
});
```

http.Agent类

HTTP 代理用于汇总在 HTTP 客户端请求中使用的 sockets。

HTTP 代理也是客户端请求使用 `Connection:keep-alive` 时的默认方式。如果没有待处理的 HTTP 请求，正在等待的一个 socket，会成为自由关闭的 socket。这也意味着，当在负载状态下，但仍不要求开发者使用 KeepAlive 手动关闭 HTTP 客户端时，Node.js 池有着 keep-alive 的好处。

如果你选择使用 HTTP 的 KeepAlive，你可以创建一个将标识设置为 `true` 的 Agent 对象（详见构造器选项）。那么，该 Agent 将保持没用过的 sockets 都在之后使用的池中。它们将被明确标记，以便不保持 Node.js 进程运行。然而，当它们不在被使用时，明确地使用 destroy() KeepAlive 代理仍不失为一个好方法，以便套接字被关闭。

当 socket 发出一个 `'close'` 事件或一个特殊的 `'agentRemove'` 事件时，sockets 将从代理池中移除。这意味着，如果你打算保留一个打开了很长一段时间的 HTTP 请求并且不想让它留在池中，你可以采用链式调用：

```
http.get(options, (res) => {
    // Do stuff
}).on('socket', (socket) => {
    socket.emit('agentRemove');
});
```

或者，你可以只选择完全使用 `agent:false` 退出池：

```
http.get({
    hostname: 'localhost',
    port: 80,
    path: '/',
    agent: false // create a new agent just for this one request
}, (res) => {
    // Do stuff with response
})
```

new Agent([options])

- `options` {Object} 在代理中设置的配置选项。可以有以下字段：
  - `keepAlive` {Boolean} 保持 sockets 在池的周围以便其他的请求可以在之后使用。默认为 `false`。
  - `keepAliveMsecs` {Integer} 当使用 HTTP 的 KeepAlive 时，多久发送 TCP KeepAlive 报文使得 sockets 保持活跃状态。默认为 `1000`。只在 keepAlive 设置为 `true` 应用。
  - `maxSockets` {Number} 每个主机允许的最大 socket 数。默认为 `Infinity`。
  - `maxFreeSockets` {Number} 准许在自由状态下打开的最大 socket 数。只在 keepAlive 设置为 `true` 应用。默认为 `256`。

默认的 http.globalAgent 被用于 http.request() 将所有这些值设置为各自的默认值。

要配置其中任何一个，你必须创建你自己的 http.Agent 对象。

```
const http = require('http');
var keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.sockets

其中包含当前在代理中使用的 socket 队列的对象。不要修改。

agent.requests

其中含有尚未被分配到 sockets 的请求队列中的对象。不要修改。

agent.freeSockets

当使用 HTTP 的 KeepAlive 时，其中包含正在等待被 Agent 使用的 socket 队列的对象。不要修改。

agent.maxSockets

默认设置为无穷大。决定可以为每个来源打开多少个并发的 sockets 代理。来源是一个 `'host:port'` 或 `'host:port:localAddress'` 组合。

agent.maxFreeSockets

默认设置为 256。对于代理支持 HTTP 的 KeepAlive，这是设置的在自由状态下打开的最大 socket 数。

agent.createConnection(options[, callback])

产生一个用于 HTTP 请求的 socket/stream。

默认情况下，该函数类似于 net.createConnection()。然而，自定义代理可以重写此方法的情况下，期望具有更大的灵活性。

socket/stream 可以由以下两种方法提供：从该函数返回 socket/stream，或通过 `callback` 中的 socket/stream。

`callback` 有 `(err, stream)` 参数。

agent.getName(options)

获得一个设置请求选项的唯一名称，以确定连接是否可以再利用。在 HTTP 代理中，这会返回 `host:port:localAddress`。在 HTTPS 代理中，该名称包括CA，证书，暗号和其他 HTTPS 或特定 TLS 选项来确定 socket 的可重用性。

选项：

- `host`：发出请求的服务器的域名或 IP 地址。
- `port`：远程服务器的端口。
- `localAddress`：在本地接口发出请求时绑定的网络连接。

agent.destroy()

注销当前代理正在使用的任何 sockets。

通常没有必要做这一点。然而，如果你使用的是启用 KeepAlive 的代理，那么当你知道它不再被使用时，最好明确关闭代理。否则，在服务器终止之前 sockets 可能会被挂起开放相当长的时间。

http.ClientRequest类

该对象在内部创建并由 http.request()。它表示着一个正在处理的请求，其头部已经进入请求队列。该头部仍可以很容易地通过 `setHeader(name, value)`、`getHeader(name)` 和 `removeHeader(name)` API 进行修改。实际头部将与第一数据块或关闭连接时一起被发送。

为了获得响应对象，将一个 `'response'` 监听器添加到请求对象上。当收到响应头时，请求对象会发出一个 `'response'` 事件。`'response'` 事件只有一个执行参数，该参数是一个 http.IncomingMessage 实例。

在 `'response'` 事件期间，可以为响应对象添加监听器；尤其是监听 `'data'` 事件。

如果没有添加 `'response'` 处理函数，那么响应会被完全忽略。然而，如果你添加了 `'response'` 处理函数，那么你必须消耗从响应对象中获得的数据，每当发生 `'readable'` 事件时调用 `response.read()`，或添加一个 `'data'` 处理函数，或通过调用 `.resume()` 方法。直到数据被消耗完之前，不会发生 `'end'` 事件。同样，如果数据未被读取，它将会消耗内存，最终产生 'process out of memory' 错误。

注意：Node.js 不会检查 Content-Length 和已发送的 body 的长度是否相等。

该请求实现了 可写流 接口。这是一个包含下列事件的 EventEmitter：

'connect' 事件

`function (response, socket, head) { }`

每当服务器响应一个带有 `CONNECT` 方法的请求时都会发生。如果没有监听该事件，客户端接收到 `CONNECT` 方法后会将有自己的连接关闭。

一对客户端和服务端组合会向你展示如何监听 `'connect'` 事件。

```
const http = require('http');
const net = require('net');
const url = require('url');

// Create an HTTP tunneling proxy
var proxy = http.createServer((req, res) => {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('okay');
});
proxy.on('connect', (req, cltSocket, head) => {
    // connect to an origin server
    var srvUrl = url.parse(`http://${req.url}`);
    var srvSocket = net.connect(srvUrl.port, srvUrl.hostname, () => {
        cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
            'Proxy-agent: Node.js-Proxy\r\n' +
            '\r\n');
        srvSocket.write(head);
        srvSocket.pipe(cltSocket);
        cltSocket.pipe(srvSocket);
    });
});

// now that proxy is running
proxy.listen(1337, '127.0.0.1', () => {

    // make a request to a tunneling proxy
    var options = {
        port: 1337,
        hostname: '127.0.0.1',
        method: 'CONNECT',
        path: 'www.google.com:80'
    };

    var req = http.request(options);
    req.end();

    req.on('connect', (res, socket, head) => {
        console.log('got connected!');

        // make a request over an HTTP tunnel
        socket.write('GET / HTTP/1.1\r\n' +
            'Host: www.google.com:80\r\n' +
            'Connection: close\r\n' +
```

```
            '\r\n');
        socket.on('data', (chunk) => {
            console.log(chunk.toString());
        });
        socket.on('end', () => {
            proxy.close();
        });
    });
});
```

'response' 事件

`function (response) { }`

当收到该请求的响应时发生。该事件只发生一次。`response` 的参数会是一个 http.IncomingMessage 的实例。

'socket' 事件

`function (socket) { }`

在该请求分配到一个 socket 后发生。

'continue' 事件

`function () { }`

当服务器发送了一个 '100 Continue' 的 HTTP 响应时发生，通常因为该请求包含 'Expect: 100-continue'。这是客户端应发送请求主体的指令。

'upgrade' 事件

`function (response, socket, head) { }`

每当服务器响应一个升级请求时发生。如果没有监听该事件，客户端接收到一个升级头部后会将有自己的连接关闭。

一对客户端和服务端组合会向你展示如何监听 `'upgrade'` 事件。

```
const http = require('http');

// Create an HTTP server
var srv = http.createServer((req, res) => {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('okay');
});
srv.on('upgrade', (req, socket, head) => {
    socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
        'Upgrade: WebSocket\r\n' +
        'Connection: Upgrade\r\n' +
        '\r\n');

    socket.pipe(socket); // echo back
});

// now that server is running
srv.listen(1337, '127.0.0.1', () => {

    // make a request
    var options = {
        port: 1337,
        hostname: '127.0.0.1',
        headers: {
            'Connection': 'Upgrade',
            'Upgrade': 'websocket'
        }
    };

    var req = http.request(options);
    req.end();

    req.on('upgrade', (res, socket, upgradeHead) => {
        console.log('got upgraded!');
        socket.end();
        process.exit(0);
    });
});
```

'abort' 事件

`function () { }`

当请求已经被客户端中止时发出。该事件只发生在第一次调用 `abort()` 。

'checkExpectation' 事件

```
function (request, response) { }
```

每当收到一个带有 http Expect ，但值不是 '100-continue' 的请求头部时发生。如果没有监听该事件，服务器会自动响应一个适当的 `417 Expectation Failed` 。

注意，当事件发生和处理后，`request` 事件将不会发生。

request.setTimeout(timeout[, callback])

一旦 socket 被分配给该请求，关联的 socket.setTimeout() 会被调用。

- `timeout` {Number} 请求被认为是超时的毫秒数。
- `callback` {Function} 可选的，当发生超时时调用的函数。同样绑定到 `timeout` 事件。

request.setNoDelay([noDelay])

一旦 socket 被分配给该请求，关联的 socket.setNoDelay() 会被调用。

request.setSocketKeepAlive([enable][, initialDelay])

一旦 socket 被分配给该请求，关联的 socket.setKeepAlive() 会被调用。

request.flushHeaders()

强制刷新请求头。

出于效率的考虑，Node.js 通常缓存请求头直到你调用 `request.end()` 或写入请求数据的第一个数据块。然后，它试图将请求头和数据封装成单一的 TCP 数据包。

这通常是你想要的（它节省了 TCP 往返），除了当第一个数据块直到很久之后才会被发送的情况。`request.flushHeaders()` 让你绕过优化并提前开始请求。

request.write(chunk[, encoding][, callback])

发送一个主体数据块。通过多次调用该方法，用户可以以流的形式将请求主体发送到服务器——在这种情况下，当创建请求时，建议使用 `['Transfer-Encoding', 'chunked']` 头部行。

该 `chunk` 参数应该是一个 Buffer 或字符串。

该 `encoding` 参数是可选的，并仅适用于 `chunk` 是字符串的情况。默认为 `'utf8'`。

该 `callback` 参数是可选的，并在数据块刷新时调用。

返回 `request`。

request.end([data][, encoding][, callback])

完成发送请求。如果主体中的任何部分未被发送，它会将它们刷新流中。如果请求被分块，它将发送终止 `'0\r\n\r\n'`。

如果指定了 `data`，这等同于在 `request.end(callback)` 之后调用 response.write(data, encoding)。

如果指定了 `callback`，当请求流结束时，它会被调用。

request.abort()

标志着请求终止。调用该方法将导致剩余的响应数据被丢弃，并销毁 socket。

http.Server类

该类继承 net.Server，并且具有以下附加的事件：

'connection' 事件

```
function (socket) { }
```

当建立一个新的 TCP 流时发出。`socket` 是 net.Socket 对象的一个类型。通常用户无需处理此事件。特别注意，采用协议解析器绑定套接字的方式会使其不发出 `'readable'` 事件。也可以通过 `request.connection` 访问 `socket`。

'connect' 事件

```
function (request, socket, head) { }
```

每当客户端请求一个 http 的 `CONNECT` 方法时发出。如果未监听该事件，那么这些请求 `CONNECT` 方法的客户端连接将会被关闭。

- `request` 是一个 http 请求的参数，与 request 事件中的相同。
- `socket` 是服务端与客户端之间的网络套接字。

- `head` 是一个 Buffer 的实例，隧道流的第一个包，该参数可能为空。

当发出该事件后，请求的套接字将不会有 `'data'` 事件监听器，也就是说你将需要绑定一个监听器到 `'data'` 事件，来处理在套接字上被发送到服务器的数据。

'checkContinue' 事件

```
function (request, response) { }
```

每当收到带有 Expect: 100-continue 的 http 请求时发出。如果未监听该事件，服务器会适当地自动发送 100 Continue 响应。

处理该事件时，如果客户端应该继续发送请求主体则调 response.writeContinue()，否则生成适当的 HTTP 响应（如，400 错误的请求）。

请注意，当这个事件发出并处理后，将不再发出 `'request'` 事件。

'request' 事件

```
function (request, response) { }
```

每次收到一个请求时发出。注意，可能存在每个连接有多个请求（在 `keep-alive` 的情况下）。`request` 是 http.IncomingMessage 的一个实例，`response` 是 http.ServerResponse 的一个实例。

'upgrade' 事件

```
function (request, socket, head) { }
```

每当客户端请求一个 http 升级时发出。如果未监听该事件，那么这些请求升级的客户端连接将会被关闭。

- `request` 是一个 http 请求的参数，与 request 事件中的相同。
- `socket` 是服务端与客户端之间的网络套接字。
- `head` 是一个 Buffer 的实例，隧道流的第一个包，该参数可能为空。

当发出该事件后，请求的套接字将不会有 `'data'` 事件监听器，也就是说你将需要绑定一个监听器到 `'data'` 事件，来处理在套接字上被发送到服务器的数据。

'close' 事件

```
function () { }
```

当服务器关闭时发出。

'clientError' 事件

```
function (exception, socket) { }
```

如果客户端发出了一个 `'error'` 事件，那么它将在这里转发。

`socket` 是发生错误的 net.Socket 对象。

server.listening

一个表示服务器是否监听连接的布尔值。

server.maxHeadersCount

最大请求头数目限制, 默认为 1000个。如果设置为 0, 则表示不做任何限制。

server.timeout

- {Number} 默认为 120000（2分钟）

一个套接字被判断为超时之前的闲置毫秒数。

注意套接字的超时逻辑在连接时设定，所以更改这个值只会影响新创建的连接，而不会影响到现有连接。

设置为 0 将阻止之后建立的连接的一切自动超时行为。

server.listen(handle[, callback])

- `handle` {Object}
- `callback` {Function}

`handle` 对象可以被设置为 server 或 socket (任何以下划线开头的 _handle 成员)，或一个 `{fd: <n>}` 对象。

这将导致服务器使用指定的句柄接受连接，但它假设文件描述符或者句柄已经被绑定到了特定的端口或者域名套接字。

在 Windows 平台上不支持监听文件描述符。

该函数是异步的。最后的参数 `callback` 会被添加到 `'listening'` 事件的监听器中。参见 net.Server.listen()。

返回 `server`。

server.listen(path[, callback])

启动一个 UNIX 套接字服务器监听给定 `path` 上的连接。

该函数是异步的。最后的参数 `callback` 会被添加到 `'listening'` 事件的监听器中。参见 net.Server.listen(path)。

server.listen(port[, hostname][, backlog][, callback])

开始在指定的 `port` 和 `hostname` 上接收连接。如果省略了 `hostname`，当 IPv6 可用时，该服务器会接收任何在 IPv6 地址（`::`）上的连接，否则便接收任何在 IPv4 地址（`0.0.0.0`）上的连接。

监听一个 UNIX 套接字，需要提供文件名而不是端口和主机名。

`backlog` 是等待连接队列的最大长度。实际长度由你的操作系统通过 `sysctl` 设置决定，比如 Linux 上的 `tcp_max_syn_backlog` 和 `somaxconn`。该参数的默认值是 511（不是 512）。

该函数是异步的。最后的参数 `callback` 会被添加到 `'listening'` 事件的监听器中。参见 net.Server.listen(port)。

server.close([callback])

禁止服务端接收新的连接。详见 net.Server.close()。

server.setTimeout(msecs, callback)

- `msecs` {Number}
- `callback` {Function}

为套接字设置超时值。如果一个超时发生，那么 Server 对象上会发出一个 `'timeout'` 事件，同时将该套接字作为参数传递。

如果在 Server 对象上有 `'timeout'` 事件监听器，那么它将被调用，而超时的套接字会作为参数传递给这个监听器。

默认情况下，服务器的超时时间是 2分钟，超时后套接字会被自动销毁。然而，如果你为该服务器的 `'timeout'` 事件分配了回调函数，那么你需要负责处理套接字的超时。

返回 `server`。

http.ServerResponse类

'finish' 事件

```
function () { }
```

当响应已发送时发出。更具体地说，当响应报头和主体的最后一段已经被切换到操作系统用于网络传输时，发出该事件。这并不意味着该客户端已经收到任何东西。

在该事件发生后，响应对象上不再发出其他任何事件。

'close' 事件

```
function () { }
```

表示在调用 response.end() 或能够刷新之前，底层连接已终止。

response.statusCode

当使用隐性头（没有显性地调用 response.writeHead()）时，在消息头获得刷新时，此属性控制将要发送到客户端的状态码。

示例：

```
response.statusCode = 404;
```

响应头被发送到客户端后，此属性表示这是发出的状态码。

response.statusMessage

当使用隐性头（没有显性地调用 response.writeHead()）时，在消息头获得刷新时，此属性控制将要发送到客户端的状态消息。如果它的左边是 `undefined`，那么它将用于该状态码的标准报文。

示例：

```
response.statusMessage = 'Not found';
```

响应头被发送到客户端后，此属性表示这是发出的状态消息。

response.headersSent

布尔值（只读）。如果发送了消息头则为 `true`，否则为 `false`。

response.sendDate

如果为 `true`，如果日期头不存在于现有报头中，则它会自动生成并在响应中发送。默认为 `true`。

这应该只在测试中被禁用; HTTP 需要响应日期头。

response.finished

布尔值，表示响应是否已完成。开始时为 `false`，在执行 response.end() 后，该值会变为 `true`。

response.setHeader(name, value)

为隐性头设置一条单独的头内容。如果这个头内容已经存在将要发送的报头中，这个值会被覆盖。如果你想用相同的名称发送多个头内容，请使用一组字符串数组。

示例：

```
response.setHeader('Content-Type', 'text/html');
```

或

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

试图设置包含无效字符的头字段名称或值会导致抛出一个 TypeError。

当消息头已经通过 response.setHeader() 设置，它们将与任何其他的消息头合并传给 response.writeHead()（优先考虑）。

```
// returns content-type = text/plain
const server = http.createServer((req, res) => {
    res.setHeader('Content-Type', 'text/html');
    res.setHeader('X-Foo', 'bar');
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('ok');
});
```

response.getHeader(name)

读出已经排队但尚未发送到客户端的消息头。请注意，名称不区分大小写。这只能在消息头被隐性刷新前调用。

```
var contentType = response.getHeader('content-type');
```

response.removeHeader(name)

从隐性发送队列中移除一个头内容。

示例：

```
response.removeHeader('Content-Encoding');
```

response.addTrailers(headers)

这个方法用于给响应添加尾部头内容（一种在消息末尾的头内容）。

如果分块编码被用于响应，尾部仅用于发送；如果不是（如，请求是 HTTP/1.0），它们将被丢弃。

请注意，如果你打算发送尾部，HTTP 要求发送 `Trailer` 消息头，带有其值的消息头字段的列表。如，

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667'});
response.end();
```

试图设置包含无效字符的头字段名称或值会导致抛出一个 TypeError。

response.writeHead(statusCode[, statusMessage][, headers])

给请求发送一个响应头。状态码是一个 3位数的 HTTP 状态代码，如 `404` 。最后的参数 `headers` ，是响应头。可选的人类可读的 `statusMessage` 是第二个参数。

示例：

```
var body = 'hello world';
response.writeHead(200, {
    'Content-Length': body.length,
    'Content-Type': 'text/plain'
});
```

该方法只能在消息中调用一次并且它必须在 response.end() 之前调用。

如果你在这之前调用 response.write() 或 response.end()，隐式/可变消息头将被计算并为你调用该函数。

当消息头已经通过 response.setHeader() 设置，它们将与任何其他的消息头合并传给 response.writeHead()（优先考虑）。

```
// returns content-type = text/plain
const server = http.createServer((req, res) => {
    res.setHeader('Content-Type', 'text/html');
    res.setHeader('X-Foo', 'bar');
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('ok');
});
```

请注意，`Content-Length` 是以字节为单位给出而不是字符。以上的例子可以工作时因为字符串 `'hello world'` 仅包含单字节字符。如果主体包含较高的编码的字符，那么 `Buffer.byteLength()` 应被用来确定给定内容的编码字节数。并且 Node.js 不会检查 `Content-Length` 和已发送的主体长度是否相同。

试图设置包含无效字符的头字段名称或值会导致抛出一个 TypeError。

response.write(chunk[, encoding][, callback])

如果调用了此方法并且没有调用 response.writeHead()，它会切换到隐性消息头模式并刷新隐性消息头。

它发送了一块响应主体的数据块。这种方法可以被调用多次，以便连续提供主体部分。

`chunk` 可以是字符串或一个 buffer。如果 `chunk` 是字符串，第二个参数指定如何将它编码成一个字节流。默认 `encoding` 为 `'utf8'`。当这个数据块被刷新时，最后的参数 `callback` 会被调用。

注意：这是原始的 HTTP 主体并且无法使用更高级别的多部分主体编码。

第一次调用 response.write()，它将发送缓冲的标题信息和第一主体到客户端。第二次调用 response.write()，Node.js 假定你要用流数据，并将它们分别发送。那便是，缓存的响应到主体的第一个数据块。

如果整个数据被成功刷新到内核缓冲区，则返回 `true`。如果数据的全部或部分在用户存储器中排队，则返回 `false`。当缓冲区再次空闲时会发出 `'drain'` 事件。

response.writeContinue()

发送一个 HTTP/1.1 100 Continue 信息到客户端。表明该请求主体应该发送。详见 `Server` 中 'checkContinue' 事件。

response.end([data][, encoding][, callback])

该方法标识着服务器端的所有响应头和主体都已经发送；该服务器应该考虑该消息完整性。对于每个响应都必须调用 `response.end()` 方法。

如果指定了 `data`，它等同于在 `response.end(callback)` 之后调用 response.write(data, encoding)。

如果指定了 `callback`，它会在响应流结束时调用。

response.setTimeout(msecs, callback)

- `msecs` {Number}
- `callback` {Function}

设置套接字的超时时间为 `msecs`。如果提供了回调函数，它将添加作为响应对象中的 `'timeout'` 事件的监听器。

如果没有 `'timeout'` 监听器添加到请求、响应或服务器，那么套接字将在它们超时后被销毁。如果你在请求、响应或服务器的 `'timeout'` 事件上分配了处理器，那么你需要负责处理套接字的超时。

返回 `response`。

http.IncomingMessage类

`IncomingMessage` 对象由 http.Server 或 http.ClientRequest 创建，并作为第一参数分别递给 `'request'` 和 `'response'` 事件。它可以用来访问响应状态，报头和数据。

它实现了可读流接口，以及下面的其他事件、方法和属性。

'close' 事件

```
function () { }
```

表明基础连接已关闭。跟 `'end'` 一样，这个事件对于每个响应只会触发一次。

message.httpVersion

在服务器请求的情况下，HTTP 版本号由客户端发送。在客户端响应的情况下，HTTP 版本由所连接到服务器决定。也许是 `'1.1'` 或 `'1.0'`。

同样，`message.httpVersionMajor` 是第一个整数，`message.httpVersionMinor` 是第二个整数。

message.url

仅适用于从 **http.Server** 获得的请求。

请求的 URL 字符串。这仅包含实际存在的 HTTP 请求的 URL。如果请求是：

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

那么 `request.url` 会是：

```
'/status?name=ryan'
```

如果你想将 URL 解析成其组成部分，你可以使用 `require('url').parse(request.url)`。例如：

```
$ node
> require('url').parse('/status?name=ryan')
{
    href: '/status?name=ryan',
    search: '?name=ryan',
    query: 'name=ryan',
    pathname: '/status'
}
```

如果你想从查询字符串中提取参数，你可以使用 `require('querystring').parse` 函数，或传递 `true` 作为 `require('url').parse` 的第二个参数。例如：

```
$ node
> require('url').parse('/status?name=ryan', true)
{
    href: '/status?name=ryan',
    search: '?name=ryan',
    query: {
        name: 'ryan'
    },
    pathname: '/status'
}
```

message.socket

与此连接相关联的 net.Socket 对象。

通过 HTTPS 的支持，使用 request.socket.getPeerCertificate() 获取客户端的认证信息。

message.method

仅适用于从 **http.Server** 获得的请求。

请求方法为字符串。只读。例如：`'GET'`，`'DELETE'`。

message.header

请求/响应头的对象。

键值对的报头名称和值。报头名称为小写。例如：

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

原报头按以下方式根据不同的头名进行重复处理：

- 重复 `age`，`authorization`，`content-length`，`content-type`，`etag`，`expires`，`from`，`host`，`if-modified-since`，`if-unmodified-since`，`last-modified`，`location`，`max-forwards`，`proxy-authorization`，`referer`，`retry-after` 或已丢弃的 `user-agent`。
- `set-cookie` 始终是一个数组。重复被添加到队列。
- 对于所有其他头，其值使用“,”相连。

message.trailers

请求/响应报尾对象。只在 `'end'` 事件时填入。

message.rawHeaders

接收到的原始请求/响应头字段列表。

注意，该键和值是在同一列表中。它不是一个元组列表。偶数偏移量为键，奇数偏移量为对应的值。

报头名称没有转换为小写，也没有合并重复的头。

```
// Prints something like:
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '*/*' ]
console.log(request.rawHeaders);
```

message.rawTrailers

接收到的原始的请求/响应报尾的键和值。只在 `'end'` 事件时填入。

message.statusCode

仅适用于从 **http.ClientRequest** 获得响应。

3位 HTTP 响应状态码。如，`404`。

message.statusMessage

仅适用于从 **http.ClientRequest** 获得响应。

HTTP 响应状态消息（简短的原因）。如，`OK` 或 `Internal Server Error`。

message.setTimeout(msecs, callback)

- `msecs` {Number}
- `callback` {Function}

调用 `message.connection.setTimeout(msecs, callback)`。

返回 `message`。

HTTPS(HTTPS)

> 稳定度：2 - 稳定

HTTPS 是使用 TLS/SSL 的 HTTP 协议。在 Node.js 中，它作为一个单独的模块来实现的。

方法和属性

---

https.globalAgent

https.Agent 的全局实例，应用于所有的 HTTPS 客户端请求。

https.createServer(options[, requestListener])

返回一个新的 HTTPS Web 服务器对象。`options` 类似于 tls.createServer()。`requestListener` 是一个自动添加到 `'request'` 事件上的函数。

示例：

```
// curl -k https://localhost:8000/
const https = require('https');
const fs = require('fs');

const options = {
    key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
    cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, (req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
}).listen(8000);
```

或

```
const https = require('https');
const fs = require('fs');

const options = {
    pfx: fs.readFileSync('server.pfx')
};

https.createServer(options, (req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
}).listen(8000);
```

server.listen(handle[, callback])

server.listen(path[, callback])

server.listen(port[, host][, backlog][, callback])

详见 http.listen()。

server.close([callback])

详见 http.close()。

https.request(options, callback)

建立一个安全的 Web 服务器的请求。

`options` 可以是一个对象或字符串。如果 `options` 是一个字符串，它会自动使用 url.parse() 解析。

所有的 http.request() 选项都是有效的。

示例：

```
const https = require('https');

var options = {
    hostname: 'encrypted.google.com',
    port: 443,
    path: '/',
    method: 'GET'
};

var req = https.request(options, (res) => {
    console.log('statusCode: ', res.statusCode);
    console.log('headers: ', res.headers);

    res.on('data', (d) => {
        process.stdout.write(d);
    });
});
req.end();

req.on('error', (e) => {
    console.error(e);
});
```

选项参数有以下选项：

- `host`：发出请求的服务器域名或 IP 地址。默认为 `'localhost'`。
- `hostname`：`host` 的别名。以支持 url.parse() 解析 `hostname` 会优于 `host`。
- `family`：在解析 `hostname` 和 `host` 时所使用的 IP 地址族。有效值是 `4` 或 `6`。当它是 `unspecified` 时，将同时使用 IPv4 和 IPv6。
- `port`：远程服务器的端口。默认为 `443`。
- `localAddress`：绑定到网络连接的本地接口。
- `socketPath`：Unix Domain Socket（使用 host:port 和 socketPath 的其中之一）
- `method`：一个指定 HTTP 请求方法的字符串。默认为 `'GET'`。
- `path`：请求路径。默认为 `'/'`。应包括查询字符串（如有的话）。如 `'/index.html?page=12'`。当请求的路径中包含非法字符时，会引发异常。目前，只有空字符会被拒绝，但在将来可能会发生变化。
- `headers`：一个包含请求头的对象。
- `auth`：基本身份验证，如，`'user:password'` 来计算 Authorization 头。
- `agent`：控制代理行为。如果使用代理请求，默认为 `Connection: keep-alive`。可能的值：

    - `undefined`（默认）：对主机和端口使用 http.globalAgent。
    - `Agent` 对象：明确地使用代理。
    - `false`：选择跳出连接池的代理，默认请求 `Connection: close`。

来自 tls.connect() 的以下选项也可以指定。然而，这里静默忽略 globalAgent。

- `pfx`：证书，用于 SSL 的私钥和 CA 证书。默认为 `null`。
- `key`：用于 SSL 的私钥。默认为 `null`。
- `passphrase`：用于私钥或 pfx 的密码字符串。默认为 `null`。
- `cert`：使用的公共 x509 证书。默认为 `null`。
- `ca`：一个字符串、Buffer 或字符串数组或受信任证书的 PEM 格式的 Buffers。如果省略，则使用几个著名的“根”的 CA，例如 VeriSign。这些用于授权的连接。
- `ciphers`：一个描述使用或排除的加密方式的字符串。在格式上仔细考虑 https://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT。
- `rejectUnauthorized`：如果 `true`，对服务器证书验证是否存在于提供的 CA 列表中。如果验证失败会发出一个 `'error'` 事件。验证发生在连接层，在发送 HTTP 请求之前。默认为 `true`。
- `secureProtocol`：使用的 SSL 方法。如，`SSLv3_method` 强制 SSL 版本 3。可能的值取决于你安装的 OpenSSL 并定义不变的 SSL_METHODS。
- `servername`：用于 SNI（服务器名称指示）TLS 扩展的服务器名。

为了指定这些选项，使用自定义代理。

示例：

```
var options = {
    hostname: 'encrypted.google.com',
    port: 443,
    path: '/',
    method: 'GET',
    key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
    cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};
options.agent = new https.Agent(options);

var req = https.request(options, (res) => {
    ...
}
```

另外，可以通过不使用代理来退出连接池。

示例：

```
var options = {
    hostname: 'encrypted.google.com',
    port: 443,
    path: '/',
    method: 'GET',
    key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
    cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
    agent: false
};

var req = https.request(options, (res) => {
    ...
}
```

https.get(options, callback)

类似 http.get()，但基于 HTTPS。

`options` 可以是对象或字符串。如果 `options` 是字符串，它会自动使用 url.parse() 解析。

示例：

```
const https = require('https');

https.get('https://encrypted.google.com/', (res) => {
    console.log('statusCode: ', res.statusCode);
    console.log('headers: ', res.headers);

    res.on('data', (d) => {
        process.stdout.write(d);
    });

}).on('error', (e) => {
    console.error(e);
});
```

https.Agent类

HTTPS 代理对象类似于 http.Agent。详见 https.request() 了解更多信息。

https.Server类

这个类是 `tls.Server` 的子类，并且触发的事件和 http.Server 相同。详见 http.Server 了解更多信息。

server.timeout

详见 http.Server#timeout。

server.setTimeout(msecs, callback)

详见 http.Server#setTimeout()。

URL(URL)

> 稳定度：2 - 稳定

该函数包含 URL 分解和解析工具。通过调用 `require('url')` 使用。

方法和属性

URL 模块提供了以下方法：

url.parse(urlStr[, parseQueryString][, slashesDenoteHost])

取一个 URL 字符串，并返回一个对象。

给第二个参数传 `true`，会使用 `querystring` 模块解析查询字符串。如果为 `true`，那么 `query` 属性将总是被指定为一个对象，并且 `search` 属性总是一个（可能为空）字符串。如果 `false`，那么 `query` 属性将不会被解析或解码。默认为 `false`。

给第三个参数传 `true`，将会把 `//foo/bar` 作为 `{ host: 'foo', pathname: '/bar' }` 对待，而不是 `{ pathname: '//foo/bar' }`。默认为 `false`。

url.format(urlObj)

取一个解析的 URL 对象，并返回一个格式化的 URL 字符串。

这里展示的是格式化过程是如何工作的：

- `href` 会被忽略。
- `path` 会被忽略。
- `protocol` 有无尾 `:`（冒号）都被同等对待。
    - 只要 `host` / `hostname` 存在，`http`、`https`、`ftp`、`gopher`、`file` 协议会被补全后缀 `://`（冒号-斜线-斜线）。
    - 其他的协议 `mailto`、`xmpp`、`aim`、`sftp`、`foo` 等，会被补全后缀 `:`（冒号）
- `slashes` 如果协议要求 `://`（冒号-斜线-斜线），设置为 `true`。
    - 只需要对此前未被列为需要斜杠的协议进行设置，如 `mongodb://localhost:8000/`，

    或假设 `host` / `hostname` 不存在。
- `auth` 如果存在的话，会被使用。
- `hostname` 如果 `host` 不存在的话才会被使用。
- `port` 如果 `host` 不存在的话才会被使用。
- `host` 会被用来代替 `hostname` 和 `port` 。
- `pathname` 对有无前导的 `/` （斜线）都一视同仁。
- `query` （对象，详见 `querystring` ）如果 `search` 不存在的话，会被使用。
- `search` 会被用来代替 `query` 。
    - 对有无前导的 `?` （问号）都一视同仁。
- `hash` 对有无前导的 `#` （井号）都一视同仁。

url.resolve(from, to)

取一个基础的 URL，和一个链接 URL，并解析它们作为一个浏览器可以理解的一个锚标记。例子：

```
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'
```

URL解析

- 转义字符

解析的 URL 对象有下列部分或全部的字段，取决于它们是否存在于 URL 字符串中。不在 URL 字符串中的其他部分将不会出现在所解析的对象中。例子所展示的 URL：

```
`'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`。
```

- `href`：原始解析的完整 URL。无论是协议还是主机都是小写。

例子：`'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- `protocol`：请求的协议，小写。

例子：`'http:'`

- `slashes`：该协议要求冒号后斜线。

例子：`true` 或 `false`

- `host`：全部小写的 URL 主机部分，包括端口信息。

例子：`'host.com:8080'`

- `auth`：一个 URL 的认证信息部分。

例子：`'user:pass'`

- `hostname`：只是主机中小写的主机名部分。

例子：`'host.com'`

- `port`：主机的端口号部分。

例子：`'8080'`

- `pathname`：URL 的路径部分，它处于主机之后，查询之前，包括初始的斜线，如果存在的话。不执行解码。

例子：`'/p/a/t/h'`

- `search`：URL 的“查询字符串”部分，包括前导的 `?`。

例子：`'?query=string'`

- `path` ： `pathname` 和 `search` 级联。不执行解码。

例子： `'/p/a/t/h?query=string'`

- `query` :'参数'查询字符串的一部分，或一个已解析的查询字符串对象。

例子： `'query=string'` 或 `{'query':'string'}`

- `hash` ：URL 的“片段”部分，包括前导的 `#` 。

例子： `'#hash'`

转义字符

空格（ `' '` ）和以下字符会在 URL 对象的属性中被自动解析：

```
< > " ` \r \n \t { } | \ ^ '
```

数据报处理(UDP/Datagram)

稳定度：2 - 稳定

`dgram` 模块提供了对 UDP 数据报套接字的实现。

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
    console.log(`server error:\n${err.stack}`);
    server.close();
});

server.on('message', (msg, rinfo) => {
    console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
    var address = server.address();
    console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

方法和属性

dgram.createSocket(options[, callback])

- `options` {Object}
- `callback` {Function} 作为 `'message'` 事件的一个附属监听器。
- 返回：dgram.Socket>

创建一个 `dgram.Socket` 对象。`options` 参数是一个应该包含一个 `udp4` 或 `udp6` 的 `type` 字段和一个可选的 `reuseAddr` 布尔型字段的对象。

当 `reuseAddr` 为 `true` 时，socket.bind() 将重用该地址，即使另一个进程已经绑定了一个套接字。`reuseAddr` 默认为 `false`。一个可选的 `callback` 函数可以通过指定添加为 `'message'` 事件的一个监听器。

一旦创建了套接字，调用 socket.bind() 会指示套接字开始监听数据报消息。当 `address` 和 `port` 没有传给 socket.bind() 时，该方法将会在一个随机端口上绑定套接字到“所有接口”地址（它对 `udp4` 和 `udp6` 套接字都是正确的）。绑定的地址和端口可以使用 socket.address().address 和 socket.address().port 检索。

dgram.createSocket(type[, callback])

- `type` {String} 既可以是 `'udp4'` 也可以是 `'udp6'`
- `callback` {Function} 作为 `'message'` 事件的一个附属监听器。可选
- 返回：dgram.Socket>

通过指定的 `type` 创建一个 `dgram.Socket` 对象。`type` 参数既可以是 `udp4`，也可以是 `udp6`。一个可选的 `callback` 函数可以通过指定添加为 `'message'` 事件的一个监听器。

一旦创建了套接字，调用 socket.bind() 会指示套接字开始监听数据报消息。当 `address` 和 `port` 没有传给 socket.bind() 时，该方法将会在一个随机端口上绑定套接字到“所有接口”地址（它对 `udp4` 和 `udp6` 套接字都是正确的）。绑定的地址和端口可以使用 socket.address().address 和 socket.address().port 检索。

dgram.Socket类

`dgram.Socket` 对象是一个封装了数据报功能的 EventEmitter。

使用 dgram.createSocket() 创建新的 `dgram.Socket` 实例。`new` 关键字不是用来创建 `dgram.Socket` 实例的。

'listening' 事件

每当一个套接字开始监听数据报消息时发出 `'listening'` 事件。这在创建 UDP 套接字时立即发生。

'message' 事件

- `msg` {Buffer} - 消息
- `rinfo` {Object} - 远程地址信息

当新的数据报在套接字上可用时，发出 `'message'` 事件。该事件处理函数传递两个参数：`msg` 和 `rinfo`。`msg` 参数是一个 Buffer，并且 `rinfo` 是一个具有发送方地址信息的对象，该对象提供 `address`、`family` 和 `port` 属性：

```
socket.on('message', (msg, rinfo) => {
    console.log('Received %d bytes from %s:%d\n',
        msg.length, rinfo.address, rinfo.port);
});
```

'close' 事件

在套接字通过 close() 关闭时发出 `'close'` 事件。一旦触发，在这个套接字上将不再发出任何新的 `'message'` 事件。

'error' 事件

- `exception` {Error}

每当发生任何错误时，发出 `'error'` 事件。该事件处理函数传递一个单一的 Error 对象。

socket.address()

返回一个包含套接字地址信息的对象。对 UDP 套接字而言，这个对象会包含 `address`、`family` 和 `port` 属性。

socket.bind(options[, callback])

- `options` {Object} - 必需。支持以下属性：
  - `port` {Number} - 必需。
  - `address` {String} - 可选。
  - `exclusive` {Boolean} - 可选。
- `callback` {Function} - 可选。

对于 UDP 套接字，在一个名为 `port` 和可选的 `address` 作为第一个参数传入的 `options` 对象的属性传递，使得 `dgram.Socket` 可以监听数据报消息。如果没有指定 `port`，操作系统将尝试绑定到随机端口。如果没有指定 `address`，操作系统将尝试监听所有地址。一旦完成绑定，将发出一个 `'listening'` 事件并调用可选的 `callback` 函数。

当在 cluster 模块中使用 `dgram.Socket` 时，`options` 对象可能包含一个额外的 `exclusive` 属性。当 `exclusive` 被设置为 `false`（默认）时，集群工作进程将使用相同的底层套接字句柄以允许共享连接来处理职责。当 `exclusive` 为 `true` 时，然而，该句柄不共享并且尝试端口共享会导致错误。

下面展示的是套接字监听专用端口的例子：

```
socket.bind({
    address: 'localhost',
    port: 8000,
    exclusive: true
});
```

socket.bind([port][, address][, callback])

- `port` {Number} - 整数，可选。
- `address` {String} - 可选。
- `callback` {Function} - 无参数，可选。绑定完成时调用。

对于 UDP 套接字，在一个名为 `port` 和可选的 `address` 作为第一个参数传入的 `options` 对象的属性传递，使得 `dgram.Socket` 可以监听数据报消息。如果没有指定 `port`，操作系统将尝试绑定到随机端口。如果没有指定 `address`，操作系统将尝试监听所有地址。一旦完成绑定，将发出一个 `'listening'` 事件并调用可选的 `callback` 函数。

请注意，同时指定了 `'listening'` 事件监听器和传递了一个 `callback` 到 `socket.bind()` 方法中是没有害处的，但不是很有用。

绑定的数据报套接字使得 Node.js 进程持续运行以接收数据报消息。

如果绑定失败，会生成一个 `'error'` 事件。在罕见的情况下（例如，试图绑定一个已关闭的套接字），将抛出一个 Error。

一个 UDP 服务器在 41234 端口上监听的例子：

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

server.on('error', (err) => {
    console.log(`server error:\n${err.stack}`);
    server.close();
});

server.on('message', (msg, rinfo) => {
    console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.on('listening', () => {
    var address = server.address();
    console.log(`server listening ${address.address}:${address.port}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

socket.send(msg, [offset, length,] port, address[, callback])

- `msg` {Buffer} | {String} | {Array} 要发送的消息。
- `offset` {Number} 整数。可选。在消息开始时的 buffer 中的偏移量。
- `length` {Number} 整数。可选。消息字节数。
- `port` {Number} 整数。目标端口。
- `address` {String} 目标主机名或 IP 地址。
- `callback` {Function} 当消息已发送时调用。可选。

在套接字上广播数据报。必须指定目标 `port` 和 `address`。

`msg` 参数包含要发送的消息。根据其类型，可以应用不同的行为。如果 `msg` 是一个 `Buffer`，`offset` 和 `length` 分别指定在 `Buffer` 中消息开始的偏移量和消息的字节数。如果 `msg` 是一个 `String`，那么它会自动转换为一个 `'utf8'` 编码的 `Buffer`。对于包含多字节字符的消息，`offset` 和 `length` 会计算相对字节长度而不是字符位置。如果 `msg` 是一个数组，不要指定 `offset` 和 `length`。

`address` 参数是一个字符串。如果 `address` 的值是一个主机名，DNS 将用于解析主机的地址。如果没有指定 `address` 或是一个空字符串，将使用 `'127.0.0.1'` 或 `'::1'`。

如果套接字之前没有给 `bind` 绑定一个调用，该套接字会被分配随机端口号，并绑定到“所有接口”地址（ `'0.0.0.0'` 用于 `udp4` 套接字， `'::0'` 用于 `udp6` 套接字。）上。

可选的 `callback` 函数可以被指定为用于报告 DNS 错误或用于确定何时重用该 `buf` 对象是安全的。请注意，DNS 查找会延迟发送至少一个 Node.js 事件循环的时间。

确定数据报已发送的唯一方法是使用一个 `callback` 。如果发生错误并且给出了一个 `callback` 。该错误将作为第一个参数传递给 `callback` 。如果没有给出 `callback` ，该错误将作为 `socket` 对象上的一个 `'error'` 事件发出。

偏移量和长度是可选的，但如果你指定了其中一个，那么你必须指定另一个。此外，它们支持jin当第一个参数是一个 `Buffer` 的情况。

在 `localhost` 上，将 UDP 包发送到一个随机端口的例子：

```
const dgram = require('dgram');
const message = new Buffer('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
    client.close();
});
```

在 `localhost` 上，将由多个 buffer 组成的 UDP 包发送到一个随机端口的例子：

```
const dgram = require('dgram');
const buf1 = new Buffer('Some ');
const buf2 = new Buffer('bytes');
const client = dgram.createSocket('udp4');
client.send([buf1, buf2], 41234, 'localhost', (err) => {
    client.close();
});
```

发送多个 buffer 可能更快或更慢，具体取决于你的应用程序和操作系统：基准测试。通常它会更快。

socket.setTTL(ttl)

- `ttl` {Number} 整数

设置 `IP_TTL` 套接字选项。虽然 TTL 通常代表“存活时间”，在此上下文中，它指定包被允许传输通过的 IP 跳的数目。转发数据包的每个路由器或网关会递减 TTL。如果 TTL 由路由器递减到 0，它将不会被转发。更改 TTL 值通常用于网络探测或组播。

`socket.setTTL()` 的参数是介于 1 和 255 之间的跳数。大多数系统的默认值是 64，但可以变化。

socket.setMulticastTTL(ttl)

- `ttl` {Number} 整数

设置 `IP_MULTICAST_TTL` 套接字选项。虽然 TTL 通常代表“存活时间”，在此上下文中，它指定包被允许传输通过的 IP 跳的数目。转发数据包的每个路由器或网关会递减 TTL。如果 TTL 由路由器递减到 0，它将不会被转发。

传递给 `socket.setMulticastTTL()` 的参数是介于 0 和 255 之间的跳数。大多数系统的默认值是 `1`，但可以变化。

socket.setMulticastLoopback(flag)

- `flag` {Boolean}

设置或清除 `IP_MULTICAST_LOOP` 套接字选项。当设置为 `true` 时，在本地接口上也将接收组播数据包。

socket.setBroadcast(flag)

- `flag` {Boolean}

设置或清除 `SO_BROADCAST` 套接字选项。当设置为 `true` 时，UDP 包可以被发送到本地接口的广播地址。

socket.close([callback])

关闭底层套接字并停止监听它上面的数据。如果提供了回调，它会被添加为 'close' 事件的一个监听器。

socket.addMembership(multicastAddress[, multicastInterface])

- `multicastAddress` {String}
- `multicastInterface` {String} 可选。

告诉内核加入组播组，在给定的 `multicastAddress` 上使用 `IP_ADD_MEMBERSHIP` 套接字选项。如果没有提供 `multicastInterface` 参数，操作系统将尝试给所有有效的网络接口添加成员资格。

socket.dropMembership(multicastAddress[, multicastInterface])

- `multicastAddress` {String}
- `multicastInterface` {String} 可选。

指示内核离开组播组，在 `multicastAddress` 上使用 `IP_DROP_MEMBERSHIP` 套接字选项。当套接字关闭或进程终止时，内核会自动调用此方法，因此大多数应用程序永远不会有理由去调用它。

如果没有指定 `multicastInterface` ，操作系统将尝试删除所有有效接口上的成员资格。

socket.unref()

默认情况下，绑定套接字将导致它阻止 Node.js 进程退出，只要套接字打开着。`socket.unref()` 方法可以用于从引用计数中排除套接字，从而使 Node.js 进程保持活动状态以允许进程退出，即使套接字仍在监听。

多次调用 `socket.unref()` 没有额外的效果。

`socket.unref()` 方法返回套接字的引用，因此可以链式调用。

socket.ref()

默认情况下，绑定套接字将导致它阻止 Node.js 进程退出，只要套接字打开着。`socket.unref()` 方法可以用于从引用计数中排除套接字，从而使 Node.js 进程保持活动状态。`socket.ref()` 方法将套接字添加回引用计数并恢复默认行为。

多次调用 `socket.ref()` 没有额外的效果。

`socket.ref()` 方法返回套接字的引用，因此可以链式调用。

socket.bind() 行为变为异步

截至 Node.js v0.10，dgram.Socket#bind() 更改为异步执行模式。遗留代码假定为同步行为，如以下示例所示：

```
const s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

必须改成传递一个回调函数到 dgram.Socket#bind() 函数：

```
const s = dgram.createSocket('udp4');
s.bind(1234, () => {
    s.addMembership('224.0.0.114');
});
```

有关 UDP 数据报大小的注意事项

一个 `IPv4/v6` 数据报的最大尺寸取决于 `MTU`（最大传输单位）和在 `Payload Length` 上的字段大小。

- `Payload Length` 字段 `16 bits` 宽，这意味着正常的有效载荷超过 64K 八位字节，包括互联网报头和数据（65,507 字节 = 65,535 − 8 字节 UDP 头 − 20 字节 IP 头）；这通常适用于环回接口，但是这样长的数据报消息对于大多数主机和网络是不切实际的。
- `MTU` 是一个给定的链路层技术可以支持的最大数据报消息尺寸。对于任何链路，`IPv4` 规定最小的 `MTU` 限制是 `68` 个八位字节，同时推荐的 IPv4 `MTU` 是 `576`（通常推荐为拨号类型应用程序的 MTU），无论它们是整体还是碎片。

  对于 IPv6，最小的 `MTU` 是 `1280` 个八位字节，然而，强制最小片段重组 buffer 大小为 `1500` 个八位字节。`68` 个八位字节的值非常小，由于大多数当前链路层技术，例如以太网，具有的最小 `MTU` 是 `1500`。

不可能提前知道包可能传输通过的每个链路的 `MTU`。发送大于接收方 `MTU` 的数据报将不起作用，因为包将被静默丢弃，而不会通知源，该数据未到达其预期接收者。

终端(TTY)

> 稳定度：2 - 稳定

`tty` 提供了 `tty.ReadStream` 和 `tty.WriteStream` 类。在大多数情况下，你不需要直接使用此模块。

当 Node.js 检测到它正在 TTY 上下文中运行时，那么 `process.stdin` 会是一个 `tty.ReadStream` 实例并且 `process.stdout` 会是一个 `tty.WriteStream` 实例。检查 Node.js 是否正在 TTY 上下文中运行的首选方法是去检测 `process.stdout.isTTY`：

```
$ node -p -e "Boolean(process.stdout.isTTY)"
true
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

方法和属性

---

tty.isatty(fd)

返回 `true` 或 `false` 取决于 `fd` 是否与终端相关联。

tty.setRawMode(mode)

> 稳定度：0 - 已废弃：使用 tty.ReadStream#setRawMode（即 process.stdin.setRawMode）代替。

ReadStream类

一个 `net.Socket` 子类，它表示 tty 的可读部分。在正常情况下，`process.stdin` 将是任何 Node.js 程序中唯一的 `tty.ReadStream` 实例（仅当 `isatty(0)` 为 true 时）。

rs.isRaw

一个 `Boolean`，初始化为 `false`。它表示 `tty.ReadStream` 实例的当前“原始”状态。

rs.setRawMode(mode)

`mode` 应该是 `true` 或 `false`。这将设置 `tty.ReadStream` 的属性作为原始设备或默认值。`isRaw` 将被设置为结果模式。

WriteStream类

一个 `net.Socket` 子类，它表示 tty 的可写部分。在正常情况下，`process.stdout` 将是任何 Node.js 程序中唯一的 `tty.WriteStream` 实例（仅当 `isatty(1)` 为 true 时）。

'resize' 事件

`function () {}`

当 `columns` 或 `rows` 属性中的任何一个已更改时，由 `refreshSize()` 发出。

```
process.stdout.on('resize', () => {
    console.log('screen size has changed!');
    console.log(`${process.stdout.columns}x${process.stdout.rows}`);
});
```

ws.columns

一个 `Number`，给出 TTY 当前具有的列数。此属性将更新 `'resize'` 事件。

ws.rows

一个 `Number`，给出 TTY 当前具有的行数。此属性将更新 `'resize'` 事件。

逐行读取(Readline)

> 稳定度：2 - 稳定

通过 `require('readline')` 使用该模块。Readline 允许在逐行的基础上读取流（例如 process.stdin）。

请注意，一旦你调用此模块，直到你关闭界面前，你的 Node.js 程序将不会终止。以下代码是如何允许你的程序正常退出：

```
const readline = require('readline');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('What do you think of Node.js? ', (answer) => {
    // TODO: Log the answer in a database
    console.log('Thank you for your valuable feedback:', answer);

    rl.close();
});
```

方法和属性

事件

方法

'SIGINT' 事件

`function () {}`

每当 `input` 流接收到 `^C`（即，`SIGINT`）时发出。当 `input` 流接收到 `SIGINT` 时，如果没有 `SIGINT` 事件监听器存在，将会触发 `pause`。

监听 `SIGINT` 的例子：

```
rl.on('SIGINT', () => {
    rl.question('Are you sure you want to exit?', (answer) => {
        if (answer.match(/^y(es)?$/i)) rl.pause();
    });
});
```

'SIGCONT' 事件

`function () {}`

这在 **Windows** 上不起作用。

每当 `input` 流发送一个 `^Z`（即，`SIGTSTP`）到后台时发出，然后继续 `fg(1)`。此事件仅在将程序发送到后台之前流未被暂停时发出。

监听 `SIGCONT` 的例子：

```
rl.on('SIGCONT', () => {
    // `prompt` will automatically resume the stream
    rl.prompt();
});
```

'SIGTSTP' 事件

`function () {}`

这在 **Windows** 上不起作用。

每当 `input` 流接收到 `^Z`（即，`SIGTSTP`）时发出。当 `input` 流接收到 `SIGTSTP` 时，如果没有 `SIGTSTP` 事件监听器存在，该程序将被发送到后台。

当程序通过 `fg` 恢复时，会发出 `'pause'` 和 `SIGCONT` 事件。你也可以用来恢复流。

如果流在程序发送到后台之前暂停，将不会发出 `'pause'` 和 `SIGCONT` 事件。

监听 `SIGTSTP` 的例子：

```
rl.on('SIGTSTP', () => {
    // This will override SIGTSTP and prevent the program from going to the
    // background.
    console.log('Caught SIGTSTP.');
});
```

'line' 事件

`function (line) {}`

每当 `input` 流接收到接收行结束符（`\n`、`\r` 或 `\r\n`）时发出，通常在用户点击进入或返回时接收。这是一个很好的 hook，可以用来监听用户输入。

监听 `line` 的例子：

```
rl.on('line', (cmd) => {
    console.log(`You just typed: ${cmd}`);
});
```

'pause' 事件

`function () {}`

每当 `input` 流暂停时发出。

同时也在每当 `input` 流没有暂停并接收到 `SIGCONT` 事件时发出。（详见 `SIGTSTP` 和 `SIGCONT` 事件）

监听 `pause` 的例子：

```
rl.on('pause', () => {
    console.log('Readline paused.');
});
```

'resume' 事件

`function () {}`

每当 `input` 流恢复时发出。

监听 `resume` 的例子：

```
rl.on('resume', () => {
    console.log('Readline resumed.');
});
```

'close' 事件

`function () {}`

当调用 `close()` 时发出。

同时也在 `input` 流接收到它的 `'end'` 事件时发出。一旦发出，`Interface` 实例应该被认为已经“完成”。举个例子，当 `input` 流接收到 `^D`（即，`EOT`）时。

当 `input` 流接收到 `^C`（即，`SIGINT`），如果不存在 `SIGINT` 事件监听器时也会被调用。

readline.createInterface(options)

创建一个 readline 的 `Interface` 实例。接收一个有以下值的 `options` 对象：

- `input` - 监听的可读流。（必需）

- `output` - 写入 readline 数据的可写流。（可选）
- `completer` - 一个用于 Tab 自动补全的可选函数。请参见下面的使用示例。
- `terminal` - 如果 `input` 和 `output` 流应该像 TTY 一样对待时传入 `true`，并会写入 ANSI/VT100 转义码。默认在 `output` 流实例化时检测 `isTTY`。
- `historySize` - 保留的最大历史记录行数。默认为 `30`。

`completer` 函数给定用户输入的当前行，并应该返回具有 2 个条目的数组：

1. 用于补全的匹配条目数组。
2. 用于匹配的子字符串。

最后看起来像：`[[substr1, substr2, ...], originalsubstring]`。

例子：

```
function completer(line) {
    var completions = '.help .error .exit .quit .q'.split(' ')
    var hits = completions.filter((c) => {
            return c.indexOf(line) == 0
        })
        // show all completions if none found
    return [hits.length ? hits : completions, line]
}
```

如果它接受两个参数，`completer` 也可以在异步模式下运行：

```
function completer(linePartial, callback) {
    callback(null, [['123'], linePartial]);
}
```

`createInterface` 通常与 process.stdin 和 process.stdout 一起使用以便接受用户输入：

```
const readline = require('readline');
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
```

一旦你有一个 readline 实例，你最常见的是去监听 `'line'` 事件。

如果在这个实例中 `terminal` 为 `true`，那么 `output` 流将获得最佳的兼容性。如果它定义了一个 `output.columns` 属性，如果/当列发生变化时，会在 `output` 上触发一个 `'resize'` 事件。（当 process.stdout 是 TTY 时自动执行此操作）

readline.cursorTo(stream, x, y)

将光标移动到给定 TTY 流中的指定位置。

readline.moveCursor(stream, dx, dy)

将光标相对于其在给定 TTY 流中的当前位置进行移动。

readline.clearLine(stream, dir)

清除给定 TTY 流在指定方向上的当前行。`dir` 应该具有以下值之一：

- `-1` - 从光标向左移动
- `1` - 从光标向右移动
- `0` - 整行

readline.clearScreenDown(stream)

从光标的当前位置开始清空屏幕。

Interface 类

表示具有输入和输出流的 readline 的接口类。

rl.write(data[, key])

向 `output` 流写入 `data`，除非在调用 `createInterface` 时，`output` 被设置为 `null` 或 `undefined`。`key` 是一个对象字面量表示的键序列；如果终端是 TTY，则可用。

这也会恢复 `input` 流，如果它已被暂停。

例子：

```
rl.write('Delete me!');
// Simulate ctrl+u to delete the line written previously
rl.write(null, {ctrl: true, name: 'u'});
```

rl.setPrompt(prompt)

设置提示，例如，当你在命令行中运行 `node` 时，你会看到 `>`，这就是 Node.js 的提示。

rl.prompt([preserveCursor])

为用户的输入准备 readline，将当前的 `setPrompt` 选项放在一个新行上，给用户一个新的写点。设置 `preserveCursor` 为 `true`，防止将光标位置重置为 `0`。

这也会恢复用于 `createInterface` 的 `input` 流，如果它已被暂停。

当调用 `createInterface` 时，如果 `output` 被设置为 `null` 或 `undefined`，该提示不会写入。

rl.question(query, callback)

在提示符前面加上 `query` 并带着用户响应调用 `callback`。向用户显示查询，然后在用户输入后，带着用户响应调用 `callback`。

这也会恢复用于 `createInterface` 的 `input` 流，如果它已被暂停。

当调用 `createInterface` 时，如果 `output` 被设置为 `null` 或 `undefined`，将不会显示。

用法示例：

```
rl.question('What is your favorite food?', (answer) => {
    console.log(`Oh, so your favorite food is ${answer}`);
});
```

rl.pause()

暂停 readline 的 `input` 流，如果需要，可以在之后恢复。

请注意，这不会立即暂停事件流。在调用 `pause` 之后，可能会发出几个事件，包括 `line`。

rl.resume()

恢复 readline 的 `input` 流。

rl.close()

关闭 `Interface` 实例，放弃控制 `input` 和 `output` 流。也同样会发出 `'close'` 事件。

示例：Tiny CLI

这里有一个如何使用所有这些接口一起来创建一个小型命令行界面的示例：

```
const readline = require('readline');
const rl = readline.createInterface(process.stdin, process.stdout);

rl.setPrompt('OHAI> ');
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!');
    process.exit(0);
});
```

示例：逐行读取文件流

一个常见的 `readline` 的 `input` 选项情况是将文件系统的可读流传递给它。这是一个如何处理文件逐行解析的例子：

```
const readline = require('readline');
const fs = require('fs');

const rl = readline.createInterface({
    input: fs.createReadStream('sample.txt')
});

rl.on('line', (line) => {
    console.log('Line from file:', line);
});
```

命令行交互(REPL)

> 稳定度：2 - 稳定

Read-Eval-Print-Loop（REPL）既可作为独立程序使用，也可轻松包含在其他程序中。REPL 提供了一种交互式运行 JavaScript 并查看结果的方法。它可以用于调试、测试或只是尝试新东西。

通过从命令行执行没有任何参数的 `node`，你将被放入 REPL。它有简单的 emacs 行编辑。

```
$ node
Type '.help' for options.
> a = [ 1, 2, 3];
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
...   });
1
2
3
```

对于高级线性编辑器，使用环境变量 `NODE_NO_READLINE=1` 启动 Node.js。在规范终端设置中启动主要的和调试的 REPL，这将允许你使用 `rlwrap`。

例如，你可以将其添加到你的 `bashrc` 文件中：

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

方法和属性

repl.start([options])

返回并启动一个 `REPLServer` 实例，继承自 Readline Interface。接受具有以下值的“options”对象：

- `prompt` - 用于所有 I/O `stream` 的提示。默认为 `>` 。
- `input` - 要监听的可读流。默认为 `process.stdin` 。
- `output` - 写入逐行读取数据的写入流。默认为 `process.stdout` 。
- `terminal` - 如果 `stream` 应该像一个 `TTY` 对待时，传入 `true` ，并写入 ANSI/VT100 转义码。默认在 `output` 流实例化时检测 `isTTY` 。
- `eval` - 将用于评估每个给定的行的函数。默认 `eval()` 为异步封装器。参见下面的自定义示例 `eval` 。
- `useColors` - 指定 `writer` 函数是否应该输出颜色的布尔值。如果设置了一个不同的 `writer` 函数，那么这将没有效果。默认为该 repl 的 `terminal` 值。
- `useGlobal` - 如果设置为 `true` ，那么该 repl 会使用 `global` 对象，代替在单独上下文中运行脚本。默认为 `false` 。
- `ignoreUndefined` - 如果设置为 `true` ，那么 repl 将不会输出返回值为 `undefined` 的命令。默认为 `false` 。
- `writer` - 为每个命令调用的函数，它返回格式化后（包括着色）的显示。默认为 `util.inspect` 。
- `replMode` - 控制 repl 是否以严格模式，默认模式或混合模式（“魔术”模式）运行所有命令。可接受的值为：
  - `repl.REPL_MODE_SLOPPY` - 以粗略模式运行命令。
  - `repl.REPL_MODE_STRICT` - 以严格模式运行命令。这相当于在每个 repl 语句前面带上 `'use strict'` 。

- `repl.REPL_MODE_MAGIC` - 尝试在默认模式下运行命令。如果他们无法解析，请在严格模式下重试。

如果有以下签名的情况下，你可以使用你自己的 `eval` 函数：

```
function eval(cmd, context, filename, callback) {
    callback(null, result);
}
```

选项卡完成时，`eval` 将用 `.scope` 作为输入字符串调用。它期望返回用于自动补全的作用域名称的数组。

多数的 REPLs 可以针对相同的 Node.js 运行实例启动。每个将共享相同的全局对象，但会具有唯一的 I/O。

这里有一个在 stdin、Unix 套接字和 TCP 套接字上运行 REPL 的例子：

```
const net = require('net');
const repl = require('repl');
var connections = 0;

repl.start({
    prompt: 'Node.js via stdin> ',
    input: process.stdin,
    output: process.stdout
});

net.createServer((socket) => {
    connections += 1;
    repl.start({
        prompt: 'Node.js via Unix socket> ',
        input: socket,
        output: socket
    }).on('exit', () => {
        socket.end();
    })
}).listen('/tmp/node-repl-sock');

net.createServer((socket) => {
    connections += 1;
    repl.start({
        prompt: 'Node.js via TCP socket> ',
        input: socket,
        output: socket
    }).on('exit', () => {
        socket.end();
    });
}).listen(5001);
```

在一个命令行中运行这个程序会在 stdin 中启动一个 REPL。其他的 REPL 客户端，可以通过 Unix 套接字或 TCP 套接字进行连接。`telnet` 对于连接到 TCP 套接字很有用，`socat` 可以用于连接到 Unix 和 TCP 套接字。

通过从基于 Unix 套接字的服务器而不是从 stdin 启动 REPL，你可以连接到长期运行的 Node.js 进程，而无需重新启动它。

在一个 `net.Server` 和 `net.Socket` 的实例上运行“全功能”（`terminal`）REPL 的例子，详见：https://gist.github.com/2209310。

在 `curl(1)` 上运行 REPL 实例的例子，详见：https://gist.github.com/2053342

REPLServer类

它继承自 Readline Interface 并带有以下事件：

'reset' 事件

`function (context) {}`

当 REPL 的上下文被重置时发出。这在你键入 `.clear` 时发生。如果你启动时带有 `{ useGlobal: true }`，那么这个事件永远不会被触发。

监听 `'reset'` 事件的例子：

```
// Extend the initial repl context.
var replServer = repl.start({
    options...
});
someExtension.extend(r.context);

// When a new context is created extend it as well.
replServer.on('reset', (context) => {
    console.log('repl has a new context');
    someExtension.extend(context);
});
```

'exit' 事件

`function () {}`

当用户以任何定义的方式退出 REPL 时发出。也就是说，在 repl 中键入 `.exit`，按下 `Ctrl+C` 两次来示意 `SIGINT`，或在 `input` 流中，按下 `Ctrl+D` 来示意 `'end'`。

监听 `'exit'` 事件的例子：

```
replServer.on('exit', () => {
    console.log('Got "exit" event from repl!');
    process.exit();
});
```

replServer.defineCommand(keyword, cmd)

- `keyword` {String}
- `cmd` {Object} | {Function}

使一个命令在 REPL 中可用。该命令通过键入 `.` 后面跟着关键字来调用。该 `cmd` 是一个具有以下值的对象：

- `help` - 当输入 `.help` 时显示的帮助文本信息（可选）。
- `action` - 一个要执行的函数，可能可以接受字符串参数，当调用命令时，绑定到 REPL 服务器实例上（必需）。

如果提供了一个函数而不是一个 `cmd` 对象，它被视为 `action`。

定义命令的例子：

```
// repl_test.js
const repl = require('repl');

var replServer = repl.start();
replServer.defineCommand('sayhello', {
    help: 'Say hello',
    action: function (name) {
        this.write(`Hello, ${name}!\n`);
        this.displayPrompt();
    }
});
```

从 REPL 中调用该命令的例子：

```
> .sayhello Node.js User
Hello, Node.js User!
```

replServer.displayPrompt([preserveCursor])

- `preserveCursor` {Boolean}

像 readline.prompt 那样，当在块内时，添加缩进和省略号。`preserveCursor` 参数是传给 readline.prompt 的。这主要用于 `defineCommand`。它也用于内部渲染每个提示行。

环境变量

内置 repl（通过运行 `node` 或 `node -i` 调用）可以通过以下环境变量进行控制：

- `NODE_REPL_HISTORY` - 当给出有效路径时，持久性的 REPL 历史将被保存到指定的文件而不是在用户的主目录中的 `.node_repl_history`。设置这个变量为 `""` 将禁用持久性的 REPL 历史记录。它会从值中删除空格。
- `NODE_REPL_HISTORY_SIZE` - 默认为 `1000`。如果历史可用，它用于控制保留多少行历史记录。必须为正数。
- `NODE_REPL_MODE` - 可能是 `sloppy`、`strict` 或 `magic` 中的任何值。默认为 `magic`，它将在严格模式下自动运行“strict mode only”语句。

永久历史

默认情况下，REPL 会保持 `node` REPL 会话之间的历史，通过保存到在用户的主目录中的 `.node_repl_history`。这可以通过设置环境变量 `NODE_REPL_HISTORY=""` 来禁用。

NODE_REPL_HISTORY_FILE

> 稳定度：0 - 已废弃：请改用 NODE_REPL_HISTORY。

在 Node.js/io.js v2.x 之前的版本中，REPL 历史是通过使用 `NODE_REPL_HISTORY_FILE` 环境变量来控制的，并且历史记录以 JSON 格式保存。此变量现已被废弃，并且你的 REPL 历史记录将自动转换为使用纯文本。新文件将保存到你的主目录，或由 `NODE_REPL_HISTORY` 变量定义的目录，如此处所述。

REPL新特性

在 REPL 内，Control+D 将退出。可以输入多行表达式。全局变量和局部变量都支持使用 Tab 补全。核心模块将按需加载到环境中。例如，访问 `fs` 将 `require()` 该 `fs` 模块作为 `global.fs` 。

特殊变量 `_`（下划线）包含最后一个表达式的结果。

```
> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
4
```

REPL 提供了对全局作用域中任何变量的访问权限。你可以通过将其分配给与每个 `REPLServer` 关联的 `context` 对象来显式地将一个变量公开给 REPL。例如：

```
// repl_test.js
const repl = require('repl');
var msg = 'message';

repl.start('> ').context.m = msg;
```

`context` 对象中的变量在 REPL 中显示为本地变量：

```
$ node repl_test.js
> m
'message'
```

有一些特殊的 REPL 命令：

- `.break` - 在输入多行表达式时，有时你会迷失或只是不在乎完成它。`.break` 会重来。
- `.clear` - 重置 `context` 对象为一个空对象，并清除任何的多行表达式。
- `.exit` - 关闭 `I/O` 流，这将导致 REPL 退出。
- `.help` - 显示这些特殊的命令列表。
- `.save` - 将当前的 REPL 会话保存到文件中。 `.save ./file/to/save.js`
- `.load` - 将文件加载到当前的 REPL 会话中。 `.load ./file/to/load.js`

REPL 中的以下组合键具有以下特殊效果：

- `<ctrl>C` - 类似于 `.break` 关键字。终止当前命令。在空白行上按两次可强制退出。
- `<ctrl>D` - 类似于 `.exit` 关键字。
- `<tab>` - 显示全局和局部（作用域）的变量。

显示在 REPL 中的自定义对象

当打印值时，REPL 模块在内部使用 util.inspect()。然而，`util.inspect` 将调用委托给对象的 `inspect()` 函数，如果它有的话。你可以在这里阅读有关此委托的更多信息。

例如，如果你已经在一个对象上像这样定义了一个 `inspect()` 函数：

```
> var obj = {foo: 'this will not show up in the inspect() output'};
undefined
> obj.inspect = () => {
...   return {bar: 'baz'};
... };
[Function]
```

并试着在 REPL 中打印 `obj`，它将会调用自定义的 `inspect()` 函数：

```
> obj
{bar: 'baz'}
```

命令行选项(Command Line Options)

Node.js 附带了各种 CLI 选项。这些选项显示了内置调试，多种执行脚本的方式以及其他有用的运行时的选项。

要在终端中将此文档视为手册页，运行 `man node` 。

概述

`node [options] [v8 options] [script.js | -e "script"] [arguments]`

`node debug [script.js | -e "script" | <host>:<port>] …`

`node --v8-options`

执行无参数启动 REPL。

关于 *`node debug`* 的更多信息，请参阅调试器文档。

选项(Options)

-v, --version

打印 Node.js 的版本号。

-h, --help

打印 Node.js 的命令行选项。此选项的输出不如本文档详细。

-e, --eval "script"

将以下参数作为 JavaScript 进行评估。在 REPL 中预定义的模块也可以在 `script` 中使用。

-p, --print "script"

与 `-e` 相同，但会打印结果。

-c, --check

在不执行的情况下，对脚本进行语法检查。

-i, --interactive

打开 REPL，即便 stdin 看起来不像终端。

-r, --require module

在启动时预加载指定的模块。

遵循 `require()` 的模块解析规则。`module` 可以是文件的路径，也可以是 Node.js 的模块名称。

--no-deprecation

静默废弃的警告。

--trace-deprecation

打印废弃的堆栈跟踪。

--throw-deprecation

抛出废弃的错误。

--trace-sync-io

每当在事件循环的第一帧之后检测到同步 I/O 时，打印堆栈跟踪。

--zero-fill-buffers

自动填充所有新分配的 Buffer 和 SlowBuffer 实例。

--track-heap-objects

为堆快照分配的堆栈对象。

--prof-process

使用 v8 选项 `--prof` 处理 v8 分析器生成的输出。

--v8-options

打印 v8 命令行选项。

--tls-cipher-list=list

指定备用的默认 TLS 加密列表。（需要使用支持的加密构建 Node.js 。（默认））

--enable-fips

启动时启用符合 FIPS 标准的加密。（需要使用 `./configure --openssl-fips` 构建 Node.js。）

--icu-data-dir=file

指定 ICU 数据的加载路径。（覆盖 NODE_ICU_DATA）

环境变量

NODE_DEBUG=module[,…]

`','` - 分隔的应该打印调试信息的核心模块列表。

NODE_PATH=path[:…]

`':'` - 分隔的模块搜索路径的前缀列表。

请注意，在 *Windows* 中，是用 `';'` - 分隔的列表。

NODE_DISABLE_COLORS=1

当设置为 `1` 时，将不会在 REPL 中使用颜色。

NODE_ICU_DATA=file

ICU（Intl 对象）数据的数据路径。将在编译时使用 `small-icu` 支持的扩展链接的数据。

NODE_REPL_HISTORY=file

用于存储持久性的 REPL 历史记录的文件的路径。它的默认路径是 `~/.node_repl_history`，该变量覆盖该值。将值设置为空字符串（`""` 或 `" "`）会禁用持久性的 REPL 历史记录。

V8(V8)

> 稳定度：2 - 稳定

本模块暴露 Node.js 中内置的 V8 版本的具体事件和接口。这些接口都受到上游程序的影响，因此稳定性指标不在范围内。

方法和属性

---

setFlagsFromString(string)

设置额外的 V8 命令行标志。请谨慎使用；在虚拟机已经开始后更改设置，可能会导致不可预知的行为，包括崩溃和数据丢失。或者，它可能什么也不做。

一个 Node.js 版本可用的 v8 选项可以通过 `node --v8-options` 确定。一个非官方的社区维护可选列表及其影响可以在 这里 找到。

用法：

```
// Print GC events to stdout for one minute.
const v8 = require('v8');
v8.setFlagsFromString('--trace_gc');
setTimeout(function() { v8.setFlagsFromString('--notrace_gc'); }, 60e3);
```

getHeapStatistics()

返回一个具有以下属性的对象：

```
{
    total_heap_size: 7326976,
    total_heap_size_executable: 4194304,
    total_physical_size: 7326976,
    total_available_size: 1152656,
    used_heap_size: 3476208,
    heap_size_limit: 1535115264
}
```

getHeapSpaceStatistics()

返回关于 V8 堆空间的统计，如，构成 v8 堆的片段。堆空间的秩序而不是堆空间的可用性，可以通过 V8 `GetHeapSpaceStatistics` 函数提供的统计得到保证。

结果举例：

```
[
    {
        "space_name": "new_space",
        "space_size": 2063872,
        "space_used_size": 951112,
        "space_available_size": 80824,
        "physical_space_size": 2063872
    },
    {
        "space_name": "old_space",
        "space_size": 3090560,
        "space_used_size": 2493792,
        "space_available_size": 0,
        "physical_space_size": 3090560
    },
    {
        "space_name": "code_space",
        "space_size": 1260160,
        "space_used_size": 644256,
        "space_available_size": 960,
        "physical_space_size": 1260160
    },
    {
        "space_name": "map_space",
        "space_size": 1094160,
        "space_used_size": 201608,
        "space_available_size": 0,
        "physical_space_size": 1094160
    },
    {
        "space_name": "large_object_space",
        "space_size": 0,
        "space_used_size": 0,
        "space_available_size": 1490980608,
        "physical_space_size": 0
    }
]
```

虚拟机(VM)

> 稳定度：2 - 稳定

你可以访问此模块：

```
const vm = require('vm');
```

JavaScript 代码可以被编译并立即运行或编译、保存并在以后运行。

方法和属性

---

vm.createContext([sandbox])

如果给一个 `sandbox` 的对象，则将沙箱 (sandbox) 对象 “上下文化 (contextify)” 供 vm.runInContext 或 script.runInContext 调用。 以此方式运行的脚本将以 `sandbox` 作为全局对象，该对象将在保留其所有的属性的基础上拥有标准全局对象所拥有的内置对象和函数。 在由 vm 模块运行的脚本之外的地方 `sandbox` 将不会被改变。

如果没有提供沙箱对象，则返回一个新的、空的、没被上下文化的可用沙箱。

此函数可用于创建可执行多个脚本的沙箱，如，在模拟浏览器的时候可以使用该函数创建一个用于表示 `window` 全局对象的沙箱， 并将所有 `<script>` 标签放入沙箱执行。

vm.isContext(sandbox)

返回一个沙箱对象是否通过调用 vm.createContext() 使其上下文化。

vm.runInThisContext(code[, options])

`vm.runInThisContext()` 编译 `code`，运行它并返回结果。运行的代码不能访问本地作用域，但可以访问当前的 `global` 对象。

使用 `vm.runInThisContext()` 和 eval() 运行相同代码的例子：

```
const vm = require('vm');
var localVar = 'initial value';

const vmResult = vm.runInThisContext('localVar = "vm";');
console.log('vmResult: ', vmResult);
console.log('localVar: ', localVar);

const evalResult = eval('localVar = "eval";');
console.log('evalResult: ', evalResult);
console.log('localVar: ', localVar);

// vmResult: 'vm', localVar: 'initial value'
// evalResult: 'eval', localVar: 'eval'
```

`vm.runInThisContext()` 无法访问本地作用域，所以 `localVar` 不变。eval() 可以访问本地作用域，所以 `localVar` 变了。

在这种方式下， `vm.runInThisContext()` 更像一个间接的 eval() 调用，如， `(0,eval)('code')` 。然而，它也有以下附加选项：

- `filename` ：允许你控制产生在任何堆栈跟踪中显示的文件名。
- `lineOffset` ：允许你添加一个显示在堆栈跟踪中行号的偏移量。
- `columnOffset` ：允许你添加一个显示在堆栈跟踪中列数的偏移量。
- `displayErrors` ：是否在抛出异常前输出带高亮错误代码行的错误信息到 stderr。将捕捉从编译 `code` 时获得的语法错误和通过执行编译代码抛出的运行时错误。默认为 `true` 。
- `timeout` ：在终止执行之前，执行 `code` 的毫秒数。如果执行被终止，错误将被抛出。

vm.runInContext(code, contextifiedSandbox[, options])

`vm.runInContext()` 编译 `code` ，然后在 `contextifiedSandbox` 中运行它并返回结果。运行的代码不能访问本地作用域。 `contextifiedSandbox` 对象必须预先通过 vm.createContext() 上下文化。它将被用作 `code` 的全局对象。

`vm.runInContext()` 与 vm.runInThisContext() 使用相同的选项。

例子：编译并在一个现有的上下文中执行不同的脚本。

```
const util = require('util');
const vm = require('vm');

const sandbox = { globalVar: 1 };
vm.createContext(sandbox);

for (var i = 0; i < 10; ++i) {
    vm.runInContext('globalVar *= 2;', sandbox);
}
console.log(util.inspect(sandbox));

// { globalVar: 1024 }
```

vm.runInNewContext(code[, sandbox][, options])

`vm.runInNewContext()` 编译 `code`，如果传入 `sandbox`，则将其上下文化，如果省略参数，则创建一个新的沙箱并将其上下文化，然后将沙箱作为全局对象运行代码并返回结果。

`vm.runInNewContext()` 与 vm.runInThisContext() 使用相同的选项。

例子：编译并执行代码，递增一个全局变量，并设置一个新的。这些全局变量包含在沙箱中。

```
const util = require('util');
const vm = require('vm');

const sandbox = {
    animal: 'cat',
    count: 2
};

vm.runInNewContext('count += 1; name = "kitty"', sandbox);
console.log(util.inspect(sandbox));

// { animal: 'cat', count: 3, name: 'kitty' }
```

请注意，运行不受信任的代码是一个棘手的事情，需要非常小心。`vm.runInNewContext()` 非常有用，但安全运行不受信任的代码需要一个单独的进程。

vm.runInDebugContext(code)

`vm.runInDebugContext()` 在 V8 debug 的上下文中编译和执行 `code`。主要用途是获得访问 V8 调试对象的权限：

```
const Debug = vm.runInDebugContext('Debug');
Debug.scripts().forEach((script) => { console.log(script.name); });
```

请注意，debug 的上下文和对象是 V8 调试器的内部实现，并且可能会改变（或者甚至可能被移除），恕不另行通知。

debug 对象同样可以通过 `--expose_debug_as=` 开关暴露。

Script类

一个保存预编译脚本的类，并在特定的沙箱中运行它们。

new vm.Script(code, options)

创建一个新的 `Script` 编译 `code`，但不运行它。相反，被创建的 `vm.Script` 对象用于表示此编译代码。此脚本可以在之后使用以下方法运行多次。返回的脚本不绑定到任何全局对象。它在每次运行前定向绑定。

创建脚本时的选项有：

- `filename`：允许你控制从该脚本中产生的任何堆栈跟踪中显示的文件名
- `lineOffset`：允许你添加一个显示在堆栈跟踪中行号的偏移量。
- `columnOffset`：允许你添加一个显示在堆栈跟踪中列数的偏移量。
- `displayErrors`：是否在抛出异常前输出带高亮错误代码行的错误信息到 stderr。仅适用于编译代码时的语法错误;而运行代码时的错误，通过脚本的方法选项来控制。
- `timeout`：在终止执行之前，执行 `code` 的毫秒数。如果执行被终止，错误将被抛出。
- `cachedData`：一个可选的 `Buffer` 和提供的源代码的 V8 代码缓存数据。当提供 `cachedDataRejected` 值时会根据由 V8 接受的数据设置 `true` 或 `false`。
- `produceCachedData`：如果为 `true` 并且不存在 `cachedData` - V8 试图生成代码缓存数据的 `code`。一旦成功，将产生和存储一个 `Buffer` 和 V8 的代码缓存数据到返回的 `vm.Script` 实例的缓存数据属性中。`cachedDataProduced` 值会被设置为 `true` 或 `false` 取决于代码缓存数据是否被成功产生。

script.runInThisContext([options])

类似于 vm.runInThisContext() 但是一个预编译 `Script` 对象的方法。`script.runInThisContext()` 运行 `script` 的编译代码并返回结果。运行的代码无权限访问本地作用域，但是有权限访问当前的 `global` 对象。

使用 `script.runInThisContext()` 编译代码一次，运行多次：

```
const vm = require('vm');

global.globalVar = 0;

const script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' });

for (var i = 0; i < 1000; ++i) {
    script.runInThisContext();
}

console.log(globalVar);

// 1000
```

运行脚本的选项有：

- `filename`：允许你控制产生在任何堆栈跟踪中显示的文件名。
- `lineOffset`：允许你添加一个显示在堆栈跟踪中行号的偏移量。
- `columnOffset`：允许你添加一个显示在堆栈跟踪中列数的偏移量。
- `displayErrors`：是否在抛出异常前输出带高亮错误代码行的错误信息到 stderr。将捕捉从编译 `code` 时获得的语法错误和通过执行编译代码抛出的运行时错误。仅适用于执行代码的运行时错误；它不可能创建一个语法错误的 `Script` 实例作为构造函数抛出。
- `timeout`：在终止执行之前，执行 `code` 的毫秒数。如果执行被终止，错误将被抛出。

script.runInContext(contextifiedSandbox[, options])

类似于 vm.runInContext() 但是一个预编译 `Script` 对象的方法。`script.runInThisContext()` 在 `contextifiedSandbox` 中运行 `script` 的编译代码并返回结果。运行的代码无权限访问本地作用域。

`script.runInContext()` 与 script.runInThisContext() 使用相同的选项。

例子：编译递增一个全局变量，并设置一个变量，然后执行该代码多次。这些全局变量包含在沙箱中。

```
const util = require('util');
const vm = require('vm');

var sandbox = {
    animal: 'cat',
    count: 2
};

var context = new vm.createContext(sandbox);
var script = new vm.Script('count += 1; name = "kitty"');

for (var i = 0; i < 10; ++i) {
    script.runInContext(context);
}

console.log(util.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

请注意，运行不受信任的代码是一个棘手的事情，需要非常小心。`script.runInContext()` 非常有用，但安全运行不受信任的代码需要一个单独的进程。

script.runInNewContext([sandbox][, options])

类似于 vm.runInNewContext() 但是一个预编译 `Script` 对象的方法。如果传入 `sandbox`，则使用 `script.runInNewContext()` 将其上下文化，然后运行 `script` 的编译代码将沙箱作为全局对象并返回结果。运行的代码无权限访问本地作用域。

`script.runInNewContext()` 与 script.runInThisContext() 使用相同的选项。

例子：编译代码，设置一个全局变量，然后在不同的上下文中多次执行代码。这些全局变量在沙箱内部被设置。

```
const util = require('util');
const vm = require('vm');

const sandboxes = [{}, {}, {}];

const script = new vm.Script('globalVar = "set"');

sandboxes.forEach((sandbox) => {
    script.runInNewContext(sandbox);
});

console.log(util.inspect(sandboxes));

// [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'set' }]
```

请注意，运行不受信任的代码是一个棘手的事情，需要非常小心。`script.runInNewContext()` 非常有用，但安全运行不受信任的代码需要一个单独的进程。

集群(Cluster)

> 稳定度：2 - 稳定

Node.js 的单个实例在单线程中运行。为了充分利用多核系统的用户有时想要启动 Node.js 进程的集群来处理负载。

集群模块可以使你轻松创建共享所有服务器端口的子进程。

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // Fork workers.
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log(`worker ${worker.process.pid} died`);
    });
} else {
    // Workers can share any TCP connection
    // In this case it is an HTTP server
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('hello world\n');
    }).listen(8000);
}
```

运行 Node.js，现在所有的工作进程会共享 8000 端口：

```
$ NODE_DEBUG=cluster node server.js
23521,Master Worker 23524 online
23521,Master Worker 23526 online
23521,Master Worker 23523 online
23521,Master Worker 23528 online
```

请注意，在 Windows 平台上，目前还不可以在一个工作进程中设置一个已命名的 pipe 服务器。

工作原理

工作进程是使用 child_process.fork() 方法进行衍生的，以便它们可以通过 IPC 与父进程进行通信并通过服务器来处理来回。

集群模块支持两种分发传入连接的方法。

第一种（除 Windows 外，所有平台上的默认方法）为轮询式：主进程监听一个端口，接受新连接，并以轮询的方式分发给工作进程，并用一些内建机制来避免单个工作进程超载。

第二种方式是，主进程建立监听 socket，并将它发送给感兴趣的工作进程，由工作进程直接接受传入的连接。

第二种方式理论上性能最佳。然而在实践中，由于操作系统的调度变幻莫测，分发往往十分不平衡。负载曾被观测到超过 70% 的连接只结束了八个进程中的两个。

因为 `server.listen()` 将大部分工作交给了主进程，所以一个普通的 Node.js 进程和一个集群工作进程会在三种情况下有所区别：

1. `server.listen({fd: 7})` 由于消息被传递到主进程，父进程中的文件描述符 7 会被监听，并且句柄会被传递给工作进程，而不是监听工作进程中文件描述符 7 所引用的东西。

2. `server.listen(handle)` 明确地监听一个句柄会使得工作进程使用所给句柄，而不是与主进程通讯。如果工作进程已经拥有了该句柄，则假定你知道你在做什么。

3. `server.listen(0)` 通常，这会让服务器监听一个随机端口。然而，在集群中，各个工作进程每次 `listen(0)` 都会得到一样的“随机”端口。实际上，端口在第一次时是随机的，但在那之后却是可预知的。如果你想要监听一个唯一的端口，则请根据集群工作进程 ID 来生成端口号。

由于在 Node.js 或你的程序中并没有路由逻辑，工作进程之间也没有共享的状态，因此在你的程序中，诸如会话和登录等功能应当被设计成不能太过依赖于内存中的数据对象。

由于工作进程都是独立的进程，因此它们会根据你的程序的需要被终止或重新衍生，并且不会影响到其它工作进程。只要还有工作进程存在，服务器就会继续接受连接。但是，Node.js 不会自动为你管理工作进程的数量，根据你的程序所需管理工作进程池是你的责任。

方法和属性

'setup' 事件

- `settings` {Object}

每次调用 `.setupMaster()` 时都会触发。

在调用 `.setupMaster()` 时，`cluster.settings` 对象就是当时的 `settings` 对象并且仅供参考，因为可以在单个时钟周期内多次调用 `.setupMaster()`。

如果非常注重精度，请使用 `cluster.settings`。

'fork' 事件

- `worker` {cluster.Worker}

当派生一个新的工作进程时，集群模块会触发一个 `'fork'` 事件。这可以用来记录工作进程的活动，并创建自己的超时时间。

```
var timeouts = [];

function errorMsg() {
    console.error('Something must be wrong with the connection ...');
}

cluster.on('fork', (worker) => {
    timeouts[worker.id] = setTimeout(errorMsg, 2000);
});

cluster.on('listening', (worker, address) => {
    clearTimeout(timeouts[worker.id]);
});

cluster.on('exit', (worker, code, signal) => {
    clearTimeout(timeouts[worker.id]);
    errorMsg();
});
```

'online' 事件

- `worker` {cluster.Worker}

在派生一个新的工作进程后，该工作进程应该用一个 online 信息作为回应。当主机收到了 online 信息，它会触发此事件。`'fork'` 和 `'online'` 的区别是：当主机派生了一个工作进程时，触发 `'fork'` ；当工作进程运行时，触发 `'online'` 。

```
cluster.on('online', (worker) => {
    console.log('Yay, the worker responded after it was forked');
});
```

'listening' 事件

- `worker` {cluster.Worker}
- `address` {Object}

从一个工作进程中调用 `listen()` 后，当在服务器端触发 `'listening'` 事件时，在 `cluster` 主机上也会触发一个 `'listening'` 事件。

该事件处理器在执行时带有两个参数，`worker` 包含工作进程对象和 `address` 对象包含以下连接属性： `address` 、 `port` 和 `addressType` 。如果该工作进程监听了多个地址，这时会非常有用。

```
cluster.on('listening', (worker, address) => {
    console.log(`A worker is now connected to ${address.address}:${address.port}`);
});
```

`addressType` 是以下之一：

- `4`（TCPv4）
- `6`（TCPv6）
- `-1`（unix domain socket）
- `"udp4"` 或 `"udp6"`（UDP v4 或 v6）

'message' 事件

- `worker` {cluster.Worker}
- `message` {Object}

当任何工作进程收到消息时触发。

详见子进程 'message' 事件。

'disconnect' 事件

- `worker` {cluster.Worker}

在工作进程 IPC 通道断开后触发。它可以在工作进程正常退出、杀死或手动断开（如使用 `worker.disconnect()`）时发生。

在 `'disconnect'` 和 `'exit'` 事件之间可能有延迟。这些事件可以被用于检测进程是否卡在清理或处于长连接状态。

```
cluster.on('disconnect', (worker) => {
    console.log(`The worker #${worker.id} has disconnected`);
});
```

'exit' 事件

- `worker` {cluster.Worker}
- `code` {Number} 它正常退出时的退出码。

- `signal` {String} 导致进程被杀死的信号名称（如 `'SIGHUP'` ）。

当任何的工作进程死亡时，集群模块都会触发 `'exit'` 事件。

可以通过再次调用 `.fork()` 来重启工作进程。

```
cluster.on('exit', (worker, code, signal) => {
    console.log('worker %d died (%s). restarting...',
        worker.process.pid, signal || code);
    cluster.fork();
});
```

详见子进程的 'exit' 事件。

cluster.settings

- {Object}
    - `execArgv` {Array} 传递给 Node.js 的可执行字符串参数列表。（默认 = `process.execArgv` ）
    - `exec` {String} 工作进程文件的路径。（默认 = `process.argv[1]` ）
    - `args` {Array} 传递给工作进程的字符串参数。（默认 = `process.argv.slice(2)` ）
    - `silent` {Boolean} 是否将输出发送到父进程的 stdio。（默认 = `false` ）
    - `uid` {Number} 设置进程的用户 id。（详见 setuid(2)）
    - `gid` {Number} 设置进程的组 id。（详见 setgid(2)）

在调用 `.setupMaster()` （或 `.fork()` ）后，设置对象会包含该设置，包括默认值。

它被设置后便不可更改，因为 `.setupMaster()` 只能被调用一次。

cluster.isMaster

- {Boolean}

如果该进程是主进程，则返回 `true` 。这由 `process.env.NODE_UNIQUE_ID` 决定。如果 `process.env.NODE_UNIQUE_ID` 是 `undefined` ，那么 `isMaster` 就是 `true` 。

cluster.isWorker

- {Boolean}

如果该进程不是主进程，则返回 `true` 。（它与 `cluster.isMaster` 相反）

cluster.worker

- {Object}

指向当前的工作进程对象。在主进程中不可用。

```
const cluster = require('cluster');

if (cluster.isMaster) {
    console.log('I am master');
    cluster.fork();
    cluster.fork();
} else if (cluster.isWorker) {
    console.log(`I am worker #${cluster.worker.id}`);
}
```

cluster.workers

- {Object}

存储着活跃的工作进程对象的散列，键为 `id` 字段。它使得遍历工作进程变得容易。只在主进程中可用。

工作进程在已断开连接并退出后，从 `cluster.workers` 中移除。这两个事件之间的顺序不能预先确定。然而，它可以保证在 `'disconnect'` 或 `'exit'` 事件触发前从 `cluster.workers` 列表中除去。

```
// Go through all workers
function eachWorker(callback) {
    for (var id in cluster.workers) {
        callback(cluster.workers[id]);
    }
}
eachWorker((worker) => {
    worker.send('big announcement to all workers');
});
```

如果你希望通过通信信道来引用一个工作进程，使用工作进程的唯一 id 是找到该工作进程的最简单方式。

```
socket.on('data', (id) => {
    var worker = cluster.workers[id];
});
```

cluster.schedulingPolicy

调度策略，`cluster.SCHED_RR` 表示轮流制，`cluster.SCHED_NONE` 表示交由操作系统处理。这是一个全局设置，并且一旦你派生了第一个工作进程或调用了 `cluster.setupMaster()` 后便不可更改。

`SCHED_RR` 是除 Windows 外所有操作系统的默认方式。只要 `libuv` 能够有效地分配 IOCP 句柄并且不产生巨大的性能损失，Windows 也将更改为 `SCHED_RR` 方式。

`cluster.schedulingPolicy` 同样也可以通过 `NODE_CLUSTER_SCHED_POLICY` 环境变量进行设置。有效值为 `"rr"` 和 `"none"`。

cluster.setupMaster([settings])

- `settings` {Object}
  - `exec` {String} 工作进程文件的路径。（默认 = `process.argv[1]`）
  - `args` {Array} 传递给工作进程的字符串参数。（默认 = `process.argv.slice(2)`）
  - `silent` {Boolean} 是否将输出发送到父进程的 stdio。（默认 = `false`）

`setupMaster` 被用于改变 `'fork'` 的默认行为。一旦调用，该设置会成为当前的 `cluster.settings` 中相关的值。

注意：

- 任何的设置变更只影响之后调用的 `.fork()` 并且不影响已经运行的工作进程。
- 工作进程的唯一属性不能通过 `.setupMaster()` 传给 `.fork()` 的 `env` 进行设置。
- 以上的默认值只适用于第一次调用，最后一次的调用默认值是当时调用 `cluster.setupMaster()` 时的当前值。

例子：

```
const cluster = require('cluster');
cluster.setupMaster({
    exec: 'worker.js',
    args: ['--use', 'https'],
    silent: true
});
cluster.fork(); // https worker
cluster.setupMaster({
    exec: 'worker.js',
    args: ['--use', 'http']
});
cluster.fork(); // http worker
```

cluster.fork([env])

- `env` {Object} 添加到工作进程环境中的键值对。
- 返回 {cluster.Worker}

衍生一个新的工作进程。

这只能在主进程中调用。

cluster.disconnect([callback])

- `callback` {Function} 当所有的工作进程断开并关闭句柄时调用。

会在 `cluster.workers` 中的每个工作进程中调用 `.disconnect()`。

当他们断开所有内部句柄时会被关闭，如果不需要等待其他事件时，可以让主进程优雅地退出。

该方法接受一个可选的回调参数，在结束后调用。

这只能在主进程中调用。

Worker类

'online' 事件

类似于 `cluster.on('online')` 事件，但指定为该工作进程。

```
cluster.fork().on('online', () => {
    // Worker is online
});
```

它不是在工作进程上触发的。

'listening' 事件

- `address` {Object}

类似于 `cluster.on('listening')` 事件，但指定为该工作进程。

```
cluster.fork().on('listening', () => {
    // Worker is listening
});
```

它不是在工作进程上触发的。

'message' 事件

- `message` {Object}

类似于 `cluster.on('message')` 事件，但指定为该工作进程。

此事件与 child_process.fork() 上提供的同名事件相同。

在工作进程中，你也可以使用 `process.on('message')`。

作为示例，有一个集群在主进程中使用消息系统保持对请求数量的计数：

```
const cluster = require('cluster');
const http = require('http');

if (cluster.isMaster) {

    // Keep track of http requests
    var numReqs = 0;
    setInterval(() => {
        console.log('numReqs =', numReqs);
    }, 1000);

    // Count requests
    function messageHandler(msg) {
        if (msg.cmd && msg.cmd == 'notifyRequest') {
            numReqs += 1;
        }
    }

    // Start workers and listen for messages containing notifyRequest
    const numCPUs = require('os').cpus().length;
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    Object.keys(cluster.workers).forEach((id) => {
        cluster.workers[id].on('message', messageHandler);
    });

} else {

    // Worker processes have a http server.
    http.Server((req, res) => {
        res.writeHead(200);
        res.end('hello world\n');

        // notify master about the request
        process.send({
            cmd: 'notifyRequest'
        });
    }).listen(8000);
}
```

'disconnect' 事件

类似于 `cluster.on('disconnect')` 事件，但指定为该工作进程。

```
cluster.fork().on('disconnect', () => {
    // Worker has disconnected
});
```

'exit' 事件

- `code` {Number} 在正常退出时的退出码。
- `signal` {String} 导致进程被杀死的信号名称（如，`'SIGHUP'`）。

类似于 `cluster.on('exit')` 事件，但指定为该工作进程。

```
const worker = cluster.fork();
worker.on('exit', (code, signal) => {
    if (signal) {
        console.log(`worker was killed by signal: ${signal}`);
    } else if (code !== 0) {
        console.log(`worker exited with error code: ${code}`);
    } else {
        console.log('worker success!');
    }
});
```

'error' 事件

此事件与 child_process.fork() 上提供的同名事件相同。

在工作进程中，你也可以使用 `process.on('error')`。

worker.id

- {Number}

每个新的工作进程都被赋予它自己唯一的 ID，这个 ID 存储在 `id` 上。

工作进程还活着时，这个就是它在 `cluster.workers` 中的索引键值。

worker.process

- {ChildProcess}

所有的工作进程都是通过 child_process.fork() 创建的，从该函数返回的对象被存储为 `.process`。在工作进程中，全局的 `process` 被存储。

详见：子进程模块。

请注意，如果 `'disconnect'` 事件发生在 `process` 上，并且 `.suicide` 不是 `true`，则工作进程会调用 `process.exit(0)`。这可以防止意外断开。

worker.suicide

- {Boolean}

通过调用 `.kill()` 或 `.disconnect()` 设置，直到那时变成 `undefined`。

`worker.suicide` 的布尔值可以让你区分自行退出和意外退出，主进程可以选择不重新衍生基于该值的工作进程。

```
cluster.on('exit', (worker, code, signal) => {
    if (worker.suicide === true) {
        console.log('Oh, it was just suicide\' – no need to worry').
    }
});

// kill worker
worker.kill();
```

worker.send(message[, sendHandle][, callback])

- `message` {Object}
- `sendHandle` {Handle}
- `callback` {Function}
- 返回：`Boolean`

发送一个消息到一个工作进程或主进程，处理句柄是可选的。

在主进程中，它发送一个消息到一个指定工作进程上。它与 ChildProcess.send() 相同。

在工作进程中，它发送消息到主进程。它与 `process.send()` 相同。

这个例子会回复所有来自主进程的消息：

```
if (cluster.isMaster) {
    var worker = cluster.fork();
    worker.send('hi there');

} else if (cluster.isWorker) {
    process.on('message', (msg) => {
        process.send(msg);
    });
}
```

worker.disconnect()

在工作进程中，该函数会关闭所有的服务器，在那些服务器上等待 `'close'` 事件，然后断开 IPC 通道。

在主进程中，会发送一个内部消息到工作进程，使其自身调用 `.disconnect()`。

使得 `.suicide` 被设置。

注意，一个服务器被关闭后，它将不再接受新的连接，但连接可能会被其他任何正在监听的工作进程所接收。现有的连接将被允许正常关闭。当不存在连接时，详见 server.close()，到工作进程的 IPC 通道会被关闭，且允许其优雅地死去。

以上仅适用于服务器的连接，客户端连接不会通过工作进程自动关闭，并且断开不会等到他们退出之前才关闭。

请注意，在一个工作进程中，存在 `process.disconnect`，但它不是这个函数，它是 disconnect。

因为长期活着的服务器连接可能会阻止工作进程断开，可能对它发送一个消息会很有用，所以应用指定动作可用于关闭它们。它同样对实现超时有用，如果在一段时间后没有触发 `'disconnect'` 事件，杀死该工作进程。

```
if (cluster.isMaster) {
    var worker = cluster.fork();
    var timeout;

    worker.on('listening', (address) => {
        worker.send('shutdown');
        worker.disconnect();
        timeout = setTimeout(() => {
            worker.kill();
        }, 2000);
    });

    worker.on('disconnect', () => {
        clearTimeout(timeout);
    });

} else if (cluster.isWorker) {
    const net = require('net');
    var server = net.createServer((socket) => {
        // connections never end
    });

    server.listen(8000);

    process.on('message', (msg) => {
        if (msg === 'shutdown') {
            // initiate graceful close of any connections to server
        }
    });
}
```

worker.kill([signal='SIGTERM'])

- `signal` {String} 发送给工作进程的 kill 信号名称。

该函数会杀死工作进程。在主进程中，它通过断开 `worker.process` 做到这一点，并且一旦断开，使用 `signal` 杀死进程。在工作进程中，它通过断开信道做到这一点，并在那时使用代码 `0` 退出。

使得 `.suicide` 被设置。

该方法为了向下兼容作为 `worker.destroy()` 的别名。

请注意，在工作进程中，存在 `process.kill()`，但它不是这个函数，它是 kill。

worker.isConnected()

如果工作进程通过它的 IPC 通道连接到了它的主进程，那么这个函数返回 `true`，否则 `false`。一个工作进程在被创建后连接到它的主进程。它在触发 `'disconnect'` 事件后断开。

worker.isDead()

如果工作进程已终止（无论是正常退出还是被信号关闭），这个函数返回 `true`，否则它返回 `false`。

域(Domain)

> 稳定度：0 - 已废弃

该模块正准备废弃。一旦替代的 API 已经敲定，该模块将被完全废弃。大多数最终用户不应该有理由使用这个模块。对于那些绝对必须要用到一个域（domain）所提供的功能的用户可以暂时依靠它，但应该在未来有不得不迁移到不同的解决方案的打算。

域（domain）提供了一种方法来处理多个不同的 IO 操作将其作为一个单独的组。如果有任何注册域（domain）的事件触发器或回调发出一个 `'error'` 事件，或抛出一个错误，那么域（domain）对象将会收到通知，而不是在 `process.on('uncaughtException')` 中处理丢失错误的情况，或导致程序伴随错误码立即退出。

方法和属性

domain.create()

- 返回：{Domain}

返回一个新的 Domain 对象。

Domain类

---

domain.members

- {Array}

已明确添加到域的定时器和事件触发器列表。

domain.bind(callback)

- `callback`：{Function} 回调函数
- 返回：{Function} 绑定函数

返回将提供的回调函数包装后的函数。当返回的函数被调用时，被抛出的任何错误都将被路由到该域的 `'error'` 的事件。

示例

```
const d = domain.create();

function readSomeFile(filename, cb) {
    fs.readFile(filename, 'utf8', d.intercept((data) => {
        // note, the first argument is never passed to the
        // callback since it is assumed to be the 'Error' argument
        // and thus intercepted by the domain.

        // if this throws, it will also be passed to the domain
        // so the error-handling logic can be moved to the 'error'
        // event on the domain instead of being repeated throughout
        // the program.
        return cb(null, JSON.parse(data));
    }));
}

d.on('error', (er) => {
    // an error occurred somewhere.
    // if we throw it now, it will crash the program
    // with the normal line number and stack message.
});
```

domain.intercept(callback)

- `callback` ：{Function} 回调函数
- 返回：{Function} 拦截函数

此方法与 domain.bind(callback) 几乎是相同的。然而，除了捕获抛出的错误，它同样会拦截 Error 对象作为第一个函数的参数发送。

通过这种方式，通用的 `if (err) return callback(err);` 模式可以被替换为在相同地方的一个错误处理程序。

示例

```
const d = domain.create();

function readSomeFile(filename, cb) {
    fs.readFile(filename, 'utf8', d.bind((er, data) => {
        // if this throws, it will also be passed to the domain
        return cb(er, data ? JSON.parse(data) : null);
    }));
}

d.on('error', (er) => {
    // an error occurred somewhere.
    // if we throw it now, it will crash the program
    // with the normal line number and stack message.
});
```

domain.run(fn[, arg][, ...])

- `fn` : {Function}

在域环境中运行所提供的函数，隐式绑定所有的事件触发器、计时器和在这种情况下创建的低频率的请求。可选的，参数可以被传递给该函数。

这是使用一个域的最基本的方法。

例子：

```
const domain = require('domain');
const fs = require('fs');
const d = domain.create();

d.on('error', (er) => {
    console.error('Caught error!', er);
});

d.run(() => {
    process.nextTick(() => {
        setTimeout(() => { // simulating some various async stuff
            fs.open('non-existent file', 'r', (er, fd) => {
                if (er) throw er;
                // proceed...
            });
        }, 100);
    });
});
```

domain.add(emitter)

- `emitter`：{EventEmitter} | {Timer} 被添加到域的事件触发器或计时器。

显式添加一个触发器到域。如果任何由触发器调用的事件处理程序抛出一个错误，或触发器触发了一个 `'error'` 事件，它会被路由到域的 `'error'` 事件，如同隐式绑定。

这也适用于从 setInterval() 和 setTimeout() 返回的定时器。如果它们的回调函数抛出了错误，那么它们会被域的 `'error'` 处理器捕捉到。

如果 Timer 或 EventEmitter 已经绑定到域，它会移那个，并绑定该处理器来代替它。

domain.remove(emitter)

- `emitter`：{EventEmitter} | {Timer} 从域中移除的事件触发器或计时器。

这是 domain.add(emitter) 的反操作。从指定的触发器中移除域处理器。

domain.enter()

`enter` 方法对 `run`、`bind` 和 `intercept` 方法而言就像其管道，用于设置活动域。它在域中设置 `domain.active` 和 `process.domain`，并通过域模块隐式推送域到域栈堆中（详见 domain.exit() 了解更多关于域栈堆的内容）。`enter` 调用隔断了异步调用链的开端和 I/O 操作绑定到域。

调用 `enter` 仅更改活动域，而不会改变域本身。`enter` 和 `exit` 可以在单一域中调用任意次。

如果域的 `enter` 调用已经被释放，`enter` 会被返回而无需设置域。

domain.exit()

`exit` 方法退出当前域，弹出并关闭域栈堆。任何时候执行流程切换到一个不同的异步回调链的上下文时，需要确保当前域已经退出。`exit` 调用隔断了异步调用链的末端或中断异步调用链和 I/O 操作绑定到域。

如果有绑定到当前执行上下文多个嵌套域，`exit` 会退出嵌套在这个域内的任何域。

调用 `exit` 仅更改活动域，而不会改变域本身。`enter` 和 `exit` 可以在单一域中调用任意次。

如果域的 `exit` 调用已经被释放，`exit` 会被返回而无需设置域。

domain.dispose()

> 稳定度：0 - 已废弃。请通过在域中明确地设置错误事件处理程序从失败的 IO 动作恢复。

一旦调用 `dispose`，该域将不可以再使用回调函数通过 `run`、`bind` 或 `intercept` 绑定到该域，并且会触发一个 `'dispose'` 事件。

特殊错误属性

任何时候一个 `Error` 对象通过了域（domain）路由，一些额外的字段会被添加到它上面。

- `error.domain` 首先处理错误的域（domain）。
- `error.domainEmitter` 触发了错误对象的 `'error'` 事件的事件触发器。
- `error.domainBound` 绑定在域（domain）上的回调函数，并把错误作为第一个参数传过去。
- `error.domainThrown` 一个布尔值，指示错误是否被抛出、触发或传递给绑定的回调函数。

隐式绑定

如果域（domain）正在使用中，那么所有新建的 `EventEmitter` 对象（包括 Stream 对象、请求、响应等）将在其创建时被隐式绑定到活动域。

此外，传递给低级事件循环的请求回调（如 fs.open 或其它 callback-taking 方法）会自动绑定到活动域。如果它们抛出，那么域会捕获该错误。

为了防止过多的内存使用情况，域对象本身不会隐性加入活动域中作为其子对象。如果是这样的话，那就很容易防止正常地垃圾回收请求和响应对象。

如果你想嵌套域对象作为父域的子域，那么你必须明确地添加它们。

隐性绑定后，路由会抛出错误和 `'error'` 事件到域（Domain）的 `'error'` 事件上，但不会在域（Domain）上注册 `EventEmitter` ，因此 domain.dispose() 不会关闭 `EventEmitter` 。隐式绑定只关心抛出错误和 `'error'` 事件。

显式绑定

有时，正在使用的域（domain）不是应该被用于一个特定的事件触发器的其中之一。或者，该事件触发器可能已在一个域的上下文中创建，但应该代替绑定到一些其他的域上。

例如，可能有一个正在被 HTTP 服务器使用的域，但也许我们想为每个请求使用一个单独的域。

这可以通过显式绑定实现。

例如：

```
// create a top-level domain for the server
const domain = require('domain');
const http = require('http');
const serverDomain = domain.create();

serverDomain.run(() => {
    // server is created in the scope of serverDomain
    http.createServer((req, res) => {
        // req and res are also created in the scope of serverDomain
        // however, we'd prefer to have a separate domain for each request.
        // create it first thing, and add req and res to it.
        var reqd = domain.create();
        reqd.add(req);
        reqd.add(res);
        reqd.on('error', (er) => {
            console.error('Error', er, req.url);
            try {
                res.writeHead(500);
                res.end('Error occurred, sorry.');
            } catch (er) {
                console.error('Error sending 500', er, req.url);
            }
        });
    }).listen(1337);
});
```

警告: 不要忽视错误!

当发生错误时，域（domain）的错误处理程序不是用于代替停歇你的进程。

根据 JavaScript 中 throw 的工作机制，几乎从来没有任何方式可以安全地“从你离开的地方重新拾起（pick up where you left off）”，无泄漏的引用，或创造一些其他种类的不确定脆弱状态。

对抛出的错误响应最安全的方式就是关闭进程。当然，在正常的 Web 服务器中，你可能有许多的连接打开着，因为错误是由别人引起的，就突然关闭进程，这是不合理的。

更好的方法是发送一个错误响应以触发错误的请求，而让其他人在正常的时间内完成，并停止监听该工作进程的新请求。

通过这种方式，`domain` 的用法是与集群模块携手，因为在工作进程遇到一个错误时，主进程可以产生一个新的工作进程。对于扩展到多台机器的 Node.js 程序而言，在终止代理或服务注册可以记录失败，并作出相应的响应。

例如，这不是一个好主意：

```
// XXX WARNING!  BAD IDEA!

var d = require('domain').create();
d.on('error', (er) => {
    // The error won't crash the process, but what it does is worse!
    // Though we've prevented abrupt process restarting, we are leaking
    // resources like crazy if this ever happens.
    // This is no better than process.on('uncaughtException')!
    console.log('error, but oh well', er.message);
});
d.run(() => {
    require('http').createServer((req, res) => {
        handleRequest(req, res);
    }).listen(PORT);
});
```

通过使用域（domain）的上下文和弹性分离我们的程序到多个工作进程中，我们可以做到更合理地响应，并且更安全地处理错误。

```
// Much better!

const cluster = require('cluster');
const PORT = +process.env.PORT || 1337;

if (cluster.isMaster) {
```

```
    // In real life, you'd probably use more than just 2 workers,
    // and perhaps not put the master and worker in the same file.
    //
    // You can also of course get a bit fancier about logging, and
    // implement whatever custom logic you need to prevent DoS
    // attacks and other bad behavior.
    //
    // See the options in the cluster documentation.
    //
    // The important thing is that the master does very little,
    // increasing our resilience to unexpected errors.

    cluster.fork();
    cluster.fork();

    cluster.on('disconnect', (worker) => {
        console.error('disconnect!');
        cluster.fork();
    });

} else {
    // the worker
    //
    // This is where we put our bugs!

    const domain = require('domain');

    // See the cluster documentation for more details about using
    // worker processes to serve requests.  How it works, caveats, etc.

    const server = require('http').createServer((req, res) => {
        var d = domain.create();
        d.on('error', (er) => {
            console.error('error', er.stack);

            // Note: we're in dangerous territory!
            // By definition, something unexpected occurred,
            // which we probably didn't want.
            // Anything can happen now!  Be very careful!

            try {
                // make sure we close down within 30 seconds
                var killtimer = setTimeout(() => {
                    process.exit(1);
                }, 30000);
                // But don't keep the process open just for that!
                killtimer.unref();

                // stop taking new requests.
                server.close();

                // Let the master know we're dead.  This will trigger a
                // 'disconnect' in the cluster master, and then it will fork
```

```
                // a new worker.
                cluster.worker.disconnect();

                // try to send an error to the request that triggered the problem
                res.statusCode = 500;
                res.setHeader('content-type', 'text/plain');
                res.end('Oops, there was a problem!\n');
            } catch (er2) {
                // oh well, not much we can do at this point.
                console.error('Error sending 500!', er2.stack);
            }
        });

        // Because req and res were created before this domain existed,
        // we need to explicitly add them.
        // See the explanation of implicit vs explicit binding below.
        d.add(req);
        d.add(res);

        // Now run the handler function in the domain.
        d.run(() => {
            handleRequest(req, res);
        });
    });
    server.listen(PORT);
}

// This part isn't important.  Just an example routing thing.
// You'd put your fancy application logic here.
function handleRequest(req, res) {
    switch (req.url) {
    case '/error':
        // We do some async stuff, and then...
        setTimeout(() => {
            // Whoops!
            flerb.bark();
        });
        break;
    default:
        res.end('ok');
    }
}
```

断言测试(Assertion Testing)

> 稳定度：3 - 已锁定

`assert` 模块提供一组简单的，可用于测试不变量断言测试。该模块是供 Node.js 内部使用的，但也可以通过 `require('assert')` 在应用代码中使用。然而，`assert` 不是一个测试框架，并且没有意愿成为通用的断言库。

`assert` 模块的 API 已经锁定。这意味着将不会增加或更改任何由模块实现和公开的方法。

方法和属性

---

assert(value[, message])

assert.ok() 的别名。

```
const assert = require('assert');

assert(true);  // OK
assert(1);     // OK
assert(false);
  // throws "AssertionError: false == true"
assert(0);
  // throws "AssertionError: 0 == true"
assert(false, 'it\'s false');
  // throws "AssertionError: it's false"
```

assert.equal(actual, expected[, message])

浅测试，使用等于比较运算符（ `==` ）来比较 `actual` 和 `expected` 参数。

```
const assert = require('assert');

assert.equal(1, 1);
  // OK, 1 == 1
assert.equal(1, '1');
  // OK, 1 == '1'

assert.equal(1, 2);
  // AssertionError: 1 == 2
assert.equal({a: {b: 1}}, {a: {b: 1}});
  //AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

如果这两个值不相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.strictEqual(actual, expected[, message])

严格相等测试，由全等运算符确定（ `===` ）。

```
const assert = require('assert');

assert.strictEqual(1, 2);
  // AssertionError: 1 === 2

assert.strictEqual(1, 1);
  // OK

assert.strictEqual(1, '1');
  // AssertionError: 1 === '1'
```

如果这两个值不是严格相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.notEqual(actual, expected[, message])

浅测试，使用不等于比较运算符（ `!=` ）比较。

```
const assert = require('assert');

assert.notEqual(1, 2);
  // OK

assert.notEqual(1, 1);
  // AssertionError: 1 != 1

assert.notEqual(1, '1');
  // AssertionError: 1 != '1'
```

如果这两个值相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.notStrictEqual(actual, expected[, message])

严格不相等测试，由不全等运算符确定（ `===` ）。

```
const assert = require('assert');

assert.notStrictEqual(1, 2);
  // OK

assert.notStrictEqual(1, 1);
  // AssertionError: 1 != 1

assert.notStrictEqual(1, '1');
  // OK
```

如果这两个值严格相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.deepEqual(actual, expected[, message])

深度比较 `actual` 和 `expected` 参数，使用比较运算符（ `==` ）比较原始值。

只考虑可枚举的“自身”属性。 `deepEqual()` 的实现不测试对象的原型，连接符号，或不可枚举的属性。这会导致一些潜在的出人意料的结果。例如，下面的例子不会抛出 `AssertionError` ，因为 Error 对象的属性是不可枚举：

```
// WARNING: This does not throw an AssertionError!
assert.deepEqual(Error('a'), Error('b'));
```

深度比较意味着子对象的可枚举“自身”的属性也会进行评估：

```
const assert = require('assert');

const obj1 = {
    a: {
        b: 1
    }
};
const obj2 = {
    a: {
        b: 2
    }
};
const obj3 = {
    a: {
        b: 1
    }
}
const obj4 = Object.create(obj1);

assert.deepEqual(obj1, obj1);
// OK, object is equal to itself

assert.deepEqual(obj1, obj2);
// AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }
// values of b are different

assert.deepEqual(obj1, obj3);
// OK, objects are equal

assert.deepEqual(obj1, obj4);
// AssertionError: { a: { b: 1 } } deepEqual {}
// Prototypes are ignored
```

如果这两个值不相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.deepStrictEqual(actual, expected[, message])

一般情况下等同于 `assert.deepEqual()` ，但有两个例外。首先，原始值是使用全等运算符（ `===` ）进行比较。其次，比较的对象包括严格比较他们的原型。

```
const assert = require('assert');

assert.deepEqual({a:1}, {a:'1'});
  // OK, because 1 == '1'

assert.deepStrictEqual({a:1}, {a:'1'});
  // AssertionError: { a: 1 } deepStrictEqual { a: '1' }
  // because 1 !== '1' using strict equality
```

如果这两个值不相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.notDeepEqual(actual, expected[, message])

深度地不相等比较测试，与 assert.deepEqual() 相反。

```
const assert = require('assert');

const obj1 = {
    a: {
        b: 1
    }
};
const obj2 = {
    a: {
        b: 2
    }
};
const obj3 = {
    a: {
        b: 1
    }
}
const obj4 = Object.create(obj1);

assert.notDeepEqual(obj1, obj1);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj2);
// OK, obj1 and obj2 are not deeply equal

assert.notDeepEqual(obj1, obj3);
// AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj4);
// OK, obj1 and obj2 are not deeply equal
```

如果这两个值深度相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.notDeepStrictEqual(actual, expected[, message])

深度地严格不相等比较测试，与 assert.deepStrictEqual() 相反。

```
const assert = require('assert');

assert.notDeepEqual({a:1}, {a:'1'});
  // AssertionError: { a: 1 } notDeepEqual { a: '1' }

assert.notDeepStrictEqual({a:1}, {a:'1'});
  // OK
```

如果这两个值深度严格相等，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

assert.ok(value[, message])

测试 `value` 是否为真值。它等同于 `assert.equal(!!value, true, message)` 。

如果 `value` 不是真值，将会抛出一个带有 `message` 属性（等于该 `message` 参数的值）的 `AssertionError` 。如果 `message` 参数为 `undefined` ，将会分配一个默认的错误消息。

```
const assert = require('assert');

assert.ok(true);  // OK
assert.ok(1);     // OK
assert.ok(false);
  // throws "AssertionError: false == true"
assert.ok(0);
  // throws "AssertionError: 0 == true"
assert.ok(false, 'it\'s false');
  // throws "AssertionError: it's false"
```

assert.fail(actual, expected, message, operator)

抛出一个 `AssertionError`。如果 `message` 是假值，错误信息会被设置为被 `operator` 分隔在两边 `actual` 和 `expected` 的值。否则，该错误信息会是 `message` 的值。

```
const assert = require('assert');

assert.fail(1, 2, undefined, '>');
  // AssertionError: 1 > 2

assert.fail(1, 2, 'whoops', '>');
  // AssertionError: whoops
```

assert.ifError(value)

如果 `value` 为真值时，抛出 `value`。当测试在回调函数里的参数 `error` 时非常有用。

```
const assert = require('assert');

assert.ifError(0); // OK
assert.ifError(1); // Throws 1
assert.ifError('error') // Throws 'error'
assert.ifError(new Error()); // Throws Error
```

assert.throws(block[, error][, message])

期望 `block` 函数抛出一个错误。

如果指定 `error`，它可以是一个构造函数、正则表达式或验证函数。

如果指定 `message`，如果 block 因为失败而抛出错误，`message` 会是由 `AssertionError` 提供的值。

验证使用构造函数实例：

```
assert.throws(
    () => {
        throw new Error('Wrong value');
    },
    Error
);
```

使用 RegExp 验证错误信息：

```
assert.throws(
    () => {
        throw new Error('Wrong value');
    },
    /value/
);
```

自定义错误验证：

```
assert.throws(
    () => {
        throw new Error('Wrong value');
    },
    function (err) {
        if ((err instanceof Error) && /value/.test(err)) {
            return true;
        }
    },
    'unexpected error'
);
```

请注意，`Error` 不能是字符串。如果字符串是作为第二个参数，那么 `error` 会被假定省略，字符串将会使用 `message` 替代。这很容易导致丢失错误：

```
// THIS IS A MISTAKE! DO NOT DO THIS!
assert.throws(myFunction, 'missing foo', 'did not throw with expected message');

// Do this instead.
assert.throws(myFunction, /missing foo/, 'did not throw with expected message');
```

assert.doesNotThrow(block[, error][, message])

断言 `block` 函数不会抛出错误。查阅 assert.throws() 了解更多详情。

当调用 `assert.doesNotThrow()` 时，它会立即调用 `block` 函数。

如果抛出错误，并且与 `error` 参数指定的类型相同，那么将会抛出一个 `AssertionError`。如果是不同类型的错误，或 `error` 参数是 `undefined`，那么错误会回传给调用者。

以下例子将会引发 TypeError，因为在断言中没有匹配的错误类型：

```
assert.doesNotThrow(
    () => {
        throw new TypeError('Wrong value');
    },
    SyntaxError
);
```

然而，以下将会导致一个带有 'Got unwanted exception (TypeError)..' 信息的 `AssertionError` 。

```
assert.doesNotThrow(
    () => {
        throw new TypeError('Wrong value');
    },
    TypeError
);
```

如果抛出了一个 `AssertionError` ，并且一个值被作为 `message` 参数，`message` 的值会被追加到 `AssertionError` 的消息中：

```
assert.doesNotThrow(
    () => {
        throw new TypeError('Wrong value');
    },
    TypeError,
    'Whoops'
);
// Throws: AssertionError: Got unwanted exception (TypeError). Whoops
```

实用工具(Utilities)

> 稳定度：2 - 稳定

这些函数都在 `'util'` 模块中。使用 `require('util')` 访问它们。

`util` 模块主要用于支持 Node.js 内部 API 的需求。这其中的大部分工具对你自己的程序会非常有用。如果你发现在这些函数中缺少你需要的函数，不过，我们鼓励你编写自己的工具函数。我们对在 `util` 模块中增加不必要的 Node.js 的内部功能的任何需求都不感兴趣。

方法和属性

util.isBoolean(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `Boolean`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isBoolean(1)
  // false
util.isBoolean(0)
  // false
util.isBoolean(false)
  // true
```

util.isNumber(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `Number`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isNumber(false)
  // false
util.isNumber(Infinity)
  // true
util.isNumber(0)
  // true
util.isNumber(NaN)
  // true
```

util.isString(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `String`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isString('')
  // true
util.isString('foo')
  // true
util.isString(String('foo'))
  // true
util.isString(5)
  // false
```

util.isObject(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是一个严格的 `Object` 并且不是一个 `Function`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isObject(5)
  // false
util.isObject(null)
  // false
util.isObject({})
  // true
util.isObject(function(){})
  // false
```

util.isArray(object)

> 稳定度：0 - 已废弃

Array.isArray 的内部别名。

如果给定的 'object' 是 `Array`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isFunction(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `Function`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

function Foo() {}
var Bar = function() {};

util.isFunction({})
  // false
util.isFunction(Foo)
  // true
util.isFunction(Bar)
  // true
```

util.isNull(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是严格的 `null`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isNull(0)
  // false
util.isNull(undefined)
  // false
util.isNull(null)
  // true
```

util.isUndefined(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `undefined`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

var foo;
util.isUndefined(5)
  // false
util.isUndefined(foo)
  // true
util.isUndefined(null)
  // false
```

util.isNullOrUndefined(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `null` 或 `undefined`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isNullOrUndefined(0)
  // false
util.isNullOrUndefined(undefined)
  // true
util.isNullOrUndefined(null)
  // true
```

util.isSymbol(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `Symbol`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isSymbol(5)
  // false
util.isSymbol('foo')
  // false
util.isSymbol(Symbol('foo'))
  // true
```

util.isError(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是一个 Error，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isError(new Error())
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

注意，这个方法依赖于 `Object.prototype.toString()` 的行为。当 `object` 参数操作 `@@toStringTag` 时，它有可能会获得一个不正确的结果。

```
// This example requires the `--harmony-tostring` flag
const util = require('util');
const obj = { name: 'Error', message: 'an error occurred' };

util.isError(obj);
  // false
obj[Symbol.toStringTag] = 'Error';
util.isError(obj);
  // true
```

util.isRegExp(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `RegExp`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是 `Date`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

util.isBuffer(object)

> 稳定度：0 - 已废弃：请使用 Buffer.isBuffer() 代替。

如果给定的 'object' 是 `Buffer`，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isBuffer({ length: 0 })
  // false
util.isBuffer([])
  // false
util.isBuffer(new Buffer('hello world'))
  // true
```

util.isPrimitive(object)

> 稳定度：0 - 已废弃

如果给定的 'object' 是一个基本类型，返回 `true`。否则，返回 `false`。

```
const util = require('util');

util.isPrimitive(5)
  // true
util.isPrimitive('foo')
  // true
util.isPrimitive(false)
  // true
util.isPrimitive(null)
  // true
util.isPrimitive(undefined)
  // true
util.isPrimitive({})
  // false
util.isPrimitive(function() {})
  // false
util.isPrimitive(/^$/)
  // false
util.isPrimitive(new Date())
  // false
```

util.inspect(object[, options])

返回 `object` 的字符串表示，这对调试有用。

可通过可选选项对象改变格式化字符串的某些方面：

- `showHidden` - 如果 `true`，那么对象的不可枚举和 symbol 属性也将被显示。默认为 `false`。
- `depth` - 告诉 `inspect` 在格式化对象时需要递归多少次。这对遍历大型且复杂的对象很有用。默认为 `2`。为了使其无限递归需要传 `null`。
- `colors` - 如果 `true`，那么输出将使用 ANSI 颜色代码样式。默认为 `false`。颜色可以定做，请参阅定制 util.inspect 颜色。
- `customInspect` - 如果 `false`，那么自定义在对象上的 `inspect(depth, opts)` 函数被检查时不会被调用。默认为 `true`。

检查 `util` 对象所有属性的例子：

```
const util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

当他们收到当前所谓的深度递归检查时，和传给 `util.inspect()` 一样的选项对象值会提供给他们自己定制的 `inspect(depth, opts)` 函数。

定制 util.inspect 颜色

通过 `util.inspect.styles` 和 `util.inspect.colors` 对象来全局定制 `util.inspect` 的颜色输出（如果启用）。

`util.inspect.styles` 是一个分配每种风格一个从 `util.inspect.colors` 获得的颜色的映射。高亮显示样式和它们的默认值是：`number`（黄色）`boolean`（黄色）`string`（绿色）`date`（品红）`regexp`（红色）`null`（粗体）`undefined`（灰色）`special` - 此时唯一函数（青色）* `name`（故意没有样式）

预定义的颜色代码是：`white`、`grey`、`black`、`blue`、`cyan`、`green`、`magenta`、`red` 和 `yellow`。同样有 `bold`、`italic`、`underline` 和 `inverse` 代码。

在对象上定制 inspect() 函数

对象也可以定义自己的 `inspect(depth)` 函数，`util.inspect()` 将调用和使用检查对象时的结果：

```
const util = require('util');

var obj = {
    name: 'nate'
};
obj.inspect = function (depth) {
    return `{${this.name}}`;
};

util.inspect(obj);
// "{nate}"
```

你也完全可以返回另一个对象，并且返回的字符串将根据返回对象进行格式化。这类似于 `JSON.stringify()` 的工作机制：

```
var obj = {
    foo: 'this will not show up in the inspect() output'
};
obj.inspect = function (depth) {
    return {
        bar: 'baz'
    };
};

util.inspect(obj);
// "{ bar: 'baz' }"
```

util.format(format[, ...])

返回使用第一个参数作为一个类 `printf` 格式的格式化的字符串。

第一个参数是包含零个或多个占位符的字符串。每个占位符替换其对应参数的转换值。支持的占位符：

- `%s` - 字符串。
- `%d` - 数值（包括整数和浮点数）。
- `%j` - JSON。如果参数包含循环引用，则将其替换为字符串 `'[Circular]'`。
- `%%` - 百分号（ `'%'` ）。这不消耗参数。

如果占位符没有相应的参数，占位符不被替换。

```
util.format('%s:%s', 'foo'); // 'foo:%s'
```

如果有比占位符更多的参数，额外的参数会被强制为字符串（对于对象和 symbol，使用 `util.inspect()`），然后拼接，用空格分隔。

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

如果第一个参数是不是一个格式化的字符串，那么 `util.format()` 返回一个所有参数用空格分隔并连在一起的字符串。每个参数 `util.inspect()` 转换为字符串。

```
util.format(1, 2, 3); // '1 2 3'
```

util.inherits(constructor, superConstructor)

从一个构造函数中继承一个原型方法到另一个。`constructor` 的属性会被设置到从 `superConstructor` 创建的新对象上。

另外的好处是，`superConstructor` 将可以通过 `constructor.super_` 属性访问。

```
const util = require('util');
const EventEmitter = require('events');

function MyStream() {
    EventEmitter.call(this);
}

util.inherits(MyStream, EventEmitter);

MyStream.prototype.write = function (data) {
    this.emit('data', data);
}

var stream = new MyStream();

console.log(stream instanceof EventEmitter); // true
console.log(MyStream.super_ === EventEmitter); // true

stream.on('data', (data) => {
    console.log(`Received data: "${data}"`);
})
stream.write('It works!'); // Received data: "It works!"
```

util.log(string)

输出带时间戳的 `stdout`。

```
require('util').log('Timestamped message.');
```

util.error([...])

> 稳定度：0 - 已废弃：使用 console.error() 代替。

`console.error` 的过时前身。

util.debug(string)

> 稳定度：0 - 已废弃：使用 console.error() 代替。

`console.error` 的过时前身。

util.debuglog(section)

- `section` {String} 进行调试的程序部分
- 返回：{Function} 日志函数

这是用来创建有条件地写入到一个基于 `NODE_DEBUG` 环境变量存在的 `stderr` 的函数。如果 `section` 名称出现在那个环境变量中，那么返回的函数将类似于 `console.error()`。如果没有，那么返回一个无操作（空）的函数。

例如：

```
var debuglog = util.debuglog('foo');

var bar = 123;
debuglog('hello from foo [%d]', bar);
```

如果程序在环境中运行时带有 `NODE_DEBUG=foo`，那么会有这样的输出：

```
FOO 3245: hello from foo [123]
```

`3245` 是进程 id，如果不是带着环境变量设置一起运行，那么它不会打印出任何东西。

你可以用逗号分隔的多个 `NODE_DEBUG` 环境变量。例如，`NODE_DEBUG=fs,net,tls`。

util.print([...])

> 稳定度：0 - 已废弃：使用 console.log() 代替。

`console.log` 的过时前身。

util.puts([...])

> 稳定度：0 - 已废弃：使用 console.log() 代替。

`console.log` 的过时前身。

util.pump(readableStream, writableStream[, callback])

> 稳定度：0 - 已废弃：使用 readableStream.pipe(writableStream) 代替。

`stream.pipe()` 的过时前身。

util.deprecate(function, string)

标志着一个方法不应该再使用。

```
const util = require('util');

exports.puts = util.deprecate(() => {
    for (var i = 0, len = arguments.length; i < len; ++i) {
        process.stdout.write(arguments[i] + '\n');
    }
}, 'util.puts: Use console.log instead');
```

它返回一个修正函数，在默认情况下警告一次。

如果设置了 `--no-deprecation` ，那么这是一个无操作（空）的函数。在运行时通过 `process.noDeprecation` 的 boolean 进行配置（仅在加载模块之前设定时有效）。

如果设置了 `--trace-deprecation` ，在第一次使用已过时的 API 时会将警告和堆栈跟踪记录到控制台中。在运行时通过 `process.traceDeprecation` 的 boolean 进行配置。

如果设置了 `--throw-deprecation` ，那么当使用已过时的 API 时，应用程序会抛出一个异常。在运行时通过 `process.throwDeprecation` 的 boolean 进行配置。

`process.throwDeprecation` 优先于 `process.traceDeprecation` 。

Punycode码(Punycode)

> 稳定度：2 - 稳定

Punycode.js 自 Node.js v0.6.2+ 开始被内置，通过 `require('punycode')` 引入。（要在其它 Node.js 版本中使用它，请先使用 npm 安装 `punycode` 模块。）

方法和属性

属性

方法

---

punycode.version

一个代表当前 Punycode.js 版本号的字符串。

punycode.ucs2

punycode.ucs2.encode(codePoints)

创建一个包含字符串中每个 Unicode 符号的数字编码点的数组。由于 JavaScript 在内部使用 UCS-2，该函数会匹配 UTF-16 将一对替代半部（UCS-2 暴露的单独的字符）转换为单个编码点。

```
punycode.ucs2.decode('abc'); // [0x61, 0x62, 0x63]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

punycode.ucs2.decode(string)

根据一组数字编码点值创建一个字符串。

```
punycode.ucs2.encode([0x61, 0x62, 0x63]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.encode(string)

将一个 Unicode 符号的字符串转换为纯 ASCII 符号的 Punycode 字符串。

```
// encode domain name parts
punycode.encode('mañana'); // 'maana-pta'
punycode.encode('☃-⌘'); // '--dqo34k'
```

punycode.decode(string)

将一个纯 ASCII 符号的 Punycode 字符串转换为 Unicode 符号的字符串。

```
// decode domain name parts
punycode.decode('maana-pta'); // 'mañana'
punycode.decode('--dqo34k'); // '☃-⌘'
```

punycode.toUnicode(domain)

将一个表示域名的 Punycode 字符串转换为 Unicode。只有域名中的 Punycode 部分会转换，也就是说你在一个已经转换为 Unicode 的字符串上调用它也是没问题的。

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn----dqo34k.com'); // '☃-⌘.com'
```

punycode.toASCII(domain)

将一个表示域名的 Unicode 字符串转换为 Punycode。只有域名的非 ASCII 部分会被转换，也就是说你在一个已经是 ASCII 的域名上调用它也是没问题的。

```
// encode domain names
punycode.toASCII('mañana.com'); // 'xn--maana-pta.com'
punycode.toASCII('☃-⌘.com'); // 'xn----dqo34k.com'
```

C/C++插件(C/C++ Addons)

Node.js 插件是动态链接共享对象，用 C 或 C++ 编写，它可以使用 require() 函数加载到 Node.js 中，并且可以把它们当成普通的 Node.js 模块那样使用。它们主要用于提供 Node.js 和 C/C++ 库之间运行 JavaScript 的接口。

目前，用于实现插件的方法相当复杂，涉及多个组件和 API 的知识：

- V8：Node.js 中目前用于提供 JavaScript 实现的 C++ 库。V8 提供了用于创建对象，调用函数等机制。V8 的 API 大部分记录在 `v8.h` 头文件（在 Node.js 的源代码树中的 `deps/v8/include/v8.h`）中，也可以在网上找到。
- libuv：实现了 Node.js 事件循环、工作线程和和所有的平台的异步行为的 C 库。它也可作为一个跨平台的抽象库，使得在所有的主流操作系统中可以像 POSIX 那样访问许多常见的系统任务，如与文件系统、sockets、计时器和系统事件的交互变得容易。libuv 还提供了一个可被用于为更复杂的异步插件供能的类 pthreads 的抽象线程，这需要高于标准的事件循环。鼓励插件作者去思考如何通过 libuv 的无阻塞系统操作，工作线程或自定义使用 libuv 线程在 I/O 或其他时间密集型任务中降低工作负载来避免阻塞事件循环。
- 内置的 Node.js 库。Node.js 自己导出我们可以使用的一些 C/C++ 的 API —— 其中最重要的是 `node::ObjectWrap` 类。
- Node.js 包括一些其他的静态链接库，包括 OpenSSL。这些其他的库位于 Node.js 源代码树中的 `deps/` 目录。只有 V8 和 OpenSSL 标志（symbol）有目的地通过 Node.js 再导出，并且可以通过插件进行不同程度的使用。查阅链接到 Node.js 自己的依赖了解附加信息。

本章节中的所有实例都可以下载，并可以作为你自己的插件的起始点。

Hello world

这个“Hello World”示例是一个简单的插件，用 C++ 编写，这等同于如下的 JavaScript代码：

```
module.exports.hello = () => 'world';
```

首先创建 `hello.cc` 文件：

```
// hello.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(addon, init)

}  // namespace demo
```

请注意，所有的 Node.js 插件必须导出如下格式的初始化函数：

```
void Initialize(Local<Object> exports);
NODE_MODULE(module_name, Initialize)
```

这里在 `NODE_MODULE` 后面没有分号，以表示它不是函数（详见 `node.h` ）。

`module_name` 必须匹配最终二进制的文件名（不包括 .node 后缀）。

在 `hello.cc` 例子中，那么， `init` 是初始化函数，并且 `addon` 是插件模块的名称。

构建

一旦源代码已被写入，它必须被编译成二进制 `addon.node` 文件。要做到这一点，在项目的顶层使用类 JSON 的格式描述你的模块构建配置，用于创建一个叫做 `binding.gyp` 的文件。该文件通过 node-gyp 使用 —— 专门写了一个用于编译 Node.js 插件的工具。

```
{
    "targets": [
        {
            "target_name": "addon",
            "sources": ["hello.cc"]
        }
    ]
}
```

注意：一个实用的 *`node-gyp`* 版本作为 *`npm`* 的一部分，通过 *Node.js* 捆绑和发行。此版本没有做出可以直接供开发者使用的版本，目的只是为了支持使用 *`npm install`* 命令编译并安装插件的能力。希望直接使用 *`node-gyp`* 的开发者可以使用 *`npm install -g node-gyp`* 命令进行安装。查阅 *`node-gyp`* 安装说明了解更多详情，包括平台特定的要求。

一旦 `binding.gyp` 文件被创建，使用 `node-gyp configure` 为当前平台生成相应项目的构建文件。这会在 `build/` 目录下生成一个 `Makefile` (在 Unix 平台上)或 `vcxproj` 文件（在 Windows 上）。

下一步，调用 `node-gyp build` 命令生成 `addon.node` 的编译文件。这会被放到 `build/Release/` 目录下。

当使用 `npm install` 安装一个 Node.js 插件时，npm 使用自身捆绑的 `node-gyp` 版本来执行同样的一套动作，为用户的需要的平台产生一个插件的编译版本。

一旦构建完成，二进制插件就可以通过 `require()` 指向内置的 `addon.node` 模块在 Node.js 内部使用：

```
// hello.js
const addon = require('./build/Release/addon');

console.log(addon.hello()); // 'world'
```

请参阅下面例子获取详细信息或在 https://github.com/arturadib/node-qt 了解关于生产环境中的例子。

因为编译后的二进制插件的确切路径可能取决于它是如何编译的（如，有时它可能在 `./build/Debug/`），插件可以使用绑定包加载已编译的模块。

请注意，虽然 `bindings` 包在如何定位插件模块的实现上更精细，但它本质上是使用类似于一个 `try-catch` 的模式：

```
try {
    return require('./build/Release/addon.node');
} catch (err) {
    return require('./build/Debug/addon.node');
}
```

链接到 Node.js 自己的依赖

Node.js 使用了一些静态链接库，比如 V8 引擎、libuv 和 OpenSSL。所有的插件都需要链接到 V8，并且也可能链接到任何其他的依赖。通常情况下，只要简单的包含相应的 `#include <...>` 声明（如，`#include <v8.h>`），并且 `node-gyp` 会找到适当的头。不过，也有一些注意事项需要注意：

- 当 `node-gyp` 运行时，它会检测 Node.js 的具体发行版本，并下载完整源码包，或只是头。如果下载了全部的源码，插件将会有完整的权限去访问 Node.js 的全套依赖。然而，如果只下载了 Node.js 的头，则仅由 Node.js 导出的标记可用。
- `node-gyp` 可以使用指向本地 Node.js 源代码信息的 `--nodedir` 标识来运行。使用此选项，插件将有机会访问全套依赖。

使用 require() 加载插件

编译后的二进制插件的文件扩展名是 `.node`（而不是 `.dll` 或 `.so`）。require() 函数是被用于查找具有 `.node` 文件扩展名的文件，并初始化这些作为动态链接库。

当调用 require() 时，`.node` 拓展名通常是可以省略的，Node.js 仍可以发现并初始化该插件。警告，然而，Node.js 会首先尝试查找并加载模块或碰巧共享相同的基本名称的 JavaScript 文件。例如，如果有一个 `addon.js` 文件与二进制 `addon.node` 在同一目录下，那么 require('addon') 将优先考虑 `addon.js` 文件并加载它来代替 `addon.node`。

插件实例

以下是旨在帮助开发人员入门的一些插件示例。这些例子利用了 V8 的 API。请参阅在线 V8 参考有助于了解各种 V8 回调和解释 V8 的嵌入者指南中使用的几个概念，如处理器、作用域和函数模板等。

每个示例都使用以下的 `binding.gyp` 文件：

```
{
    "targets": [
        {
            "target_name": "addon",
            "sources": ["addon.cc"]
        }
    ]
}
```

在有一个以上的 `.cc` 文件的情况下，只要将额外的文件名添加到源数组。例如：

```
"sources": ["addon.cc", "myexample.cc"]
```

一旦 `binding.gyp` 文件准备就绪，示例中的插件就可以使用 `node-gyp` 进行配置和构建：

```
$ node-gyp configure build
```

函数的参数

插件通常会暴露可以从运行在 Node.js 中的 JavaScript 访问到的对象和函数。当函数在 JavaScript 中调用时，输入参数和返回值必须与 C/C++ 代码相互映射。

下面的例子说明了如何读取从 JavaScript 传递过来的函数参数和如何返回结果：

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Exception;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// This is the implementation of the "add" method
// Input arguments are passed using the
// const FunctionCallbackInfo<Value>& args struct
void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  // Check the number of arguments passed.
  if (args.Length() < 2) {
    // Throw an Error that is passed back to JavaScript
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong number of arguments")));
    return;
  }

  // Check the argument types
  if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong arguments")));
    return;
  }

  // Perform the operation
  double value = args[0]->NumberValue() + args[1]->NumberValue();
  Local<Number> num = Number::New(isolate, value);

  // Set the return value (using the passed in
  // FunctionCallbackInfo<Value>&)
  args.GetReturnValue().Set(num);
}

void Init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, Init)

}  // namespace demo
```

一旦编译，示例插件就可以在 Node.js 中加载和使用：

```
// test.js
const addon = require('./build/Release/addon');

console.log('This should be eight:', addon.add(3, 5));
```

回调

通过传递 JavaScript 函数到一个 C++ 函数并在那里执行它们，这在插件里是常见的做法。下面的例子说明了如何调用这些回调：

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Null;
using v8::Object;
using v8::String;
using v8::Value;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Function> cb = Local<Function>::Cast(args[0]);
  const unsigned argc = 1;
  Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world") };
  cb->Call(Null(isolate), argc, argv);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", RunCallback);
}

NODE_MODULE(addon, Init)

}  // namespace demo
```

注意，这个例子的 `Init()` 使用两个参数，它接收完整的 `module` 对象作为第二个参数。这使得插件完全覆写 `exports` 以一个单一的函数代替添加函数作为 `exports` 的属性。

为了验证这一情况，请运行下面的 JavaScript：

```
// test.js
const addon = require('./build/Release/addon');

addon((msg) => {
    console.log(msg); // 'hello world'
});
```

需要注意的是，在这个例子中，回调函数是同步调用的。

对象工厂

插件从如下示例中所示的 C++ 函数中创建和返回新对象。一个带有 `msg` 属性的对象被创建和返回，该属性是传递给 `createObject()` 的相呼应的字符串：

```
// addon.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  Local<Object> obj = Object::New(isolate);
  obj->Set(String::NewFromUtf8(isolate, "msg"), args[0]->ToString());

  args.GetReturnValue().Set(obj);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(addon, Init)

}  // namespace demo
```

在 JavaScript 中测试它：

```
// test.js
const addon = require('./build/Release/addon');

var obj1 = addon('hello');
var obj2 = addon('world');
console.log(obj1.msg + ' ' + obj2.msg); // 'hello world'
```

函数工厂

另一种常见情况是创建 JavaScript 函数来包装 C++ 函数，并返回那些使其回到 JavaScript：

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "hello world"));
}

void CreateFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
  Local<Function> fn = tpl->GetFunction();

  // omit this to make it anonymous
  fn->SetName(String::NewFromUtf8(isolate, "theFunction"));

  args.GetReturnValue().Set(fn);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateFunction);
}

NODE_MODULE(addon, Init)

}  // namespace demo
```

去测试：

```
// test.js
const addon = require('./build/Release/addon');

var fn = addon();
console.log(fn()); // 'hello world'
```

包装 C++ 对象

另外，也可以包装 C++ 对象/类允许以使用 JavaScript 的 `new` 操作的方式来创建新的实例：

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Local;
using v8::Object;

void InitAll(Local<Object> exports) {
  MyObject::Init(exports);
}

NODE_MODULE(addon, InitAll)

}  // namespace demo
```

那么，在 `myobject.h` 中，包装类继承自 `node::ObjectWrap`：

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Local<v8::Object> exports);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

在 `myobject.cc` 中，实现要被暴露的各种方法。下面，`plusOne()` 方法是通过将其添加到构造函数的原型来暴露的：

```
// myobject.cc
#include "myobject.h"

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Local<Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  constructor.Reset(isolate, tpl->GetFunction());
  exports->Set(String::NewFromUtf8(isolate, "MyObject"),
               tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
```

```
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

}  // namespace demo
```

要构建这个例子，`myobject.cc` 文件必须被添加到 `binding.gyp`：

```
{
    "targets": [
        {
            "target_name": "addon",
            "sources": [
                "addon.cc",
                "myobject.cc"
            ]
        }
   ]
}
```

测试：

```
// test.js
const addon = require('./build/Release/addon');

var obj = new addon.MyObject(10);
console.log(obj.plusOne()); // 11
console.log(obj.plusOne()); // 12
console.log(obj.plusOne()); // 13
```

包装对象的工厂

另外，它可以使用一个工厂模式，以避免显式地使用 JavaScript 的 `new` 操作来创建对象的实例：

```
var obj = addon.createObject();
// instead of:
// var obj = new addon.Object();
```

首先，在 `addon.cc` 中实现 `createObject()` 方法：

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  MyObject::NewInstance(args);
}

void InitAll(Local<Object> exports, Local<Object> module) {
  MyObject::Init(exports->GetIsolate());

  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(addon, InitAll)

}  // namespace demo
```

在 `myobject.h` 中，添加静态方法 `NewInstance()` 来处理实例化对象。这种方法需要在 JavaScript 中使用 `new`：

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Isolate* isolate);
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

在 `myobject.cc` 中的实现类似与之前的例子：

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}
```

```
MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, constructor);
  Local<Object> instance = cons->NewInstance(argc, argv);

  args.GetReturnValue().Set(instance);
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}
```

```
}  // namespace demo
```

再来一次，建立这个例子，`myobject.cc` 文件必须被添加到 `binding.gyp`：

```
{
    "targets": [
        {
            "target_name": "addon",
            "sources": [
                "addon.cc",
                "myobject.cc"
            ]
        }
   ]
}
```

测试它：

```
// test.js
const createObject = require('./build/Release/addon');

var obj = createObject(10);
console.log(obj.plusOne()); // 11
console.log(obj.plusOne()); // 12
console.log(obj.plusOne()); // 13

var obj2 = createObject(20);
console.log(obj2.plusOne()); // 21
console.log(obj2.plusOne()); // 22
console.log(obj2.plusOne()); // 23
```

传递包装的对象

除了包装和返回的 C++ 对象，也有可能是通过 Node.js 的辅助函数 `node::ObjectWrap::Unwrap` 展开传过来的包装对象。下面的例子显示了一个 `add()` 函数可以采用两个 `MyObject` 对象作为输入参数：

```
// addon.cc
#include <node.h>
#include <node_object_wrap.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  MyObject::NewInstance(args);
}

void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj1 = node::ObjectWrap::Unwrap<MyObject>(
      args[0]->ToObject());
  MyObject* obj2 = node::ObjectWrap::Unwrap<MyObject>(
      args[1]->ToObject());

  double sum = obj1->value() + obj2->value();
  args.GetReturnValue().Set(Number::New(isolate, sum));
}

void InitAll(Local<Object> exports) {
  MyObject::Init(exports->GetIsolate());

  NODE_SET_METHOD(exports, "createObject", CreateObject);
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, InitAll)

}  // namespace demo
```

在 `myobject.h` 中，添加了一个公共方法以允许在展开对象后访问私有值。

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Isolate* isolate);
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);
  inline double value() const { return value_; }

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

在 `myobject.cc` 中的实现类似与之前的例子：

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::Persistent;
using v8::String;
using v8::Value;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}
```

```
MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, constructor);
  Local<Object> instance = cons->NewInstance(argc, argv);

  args.GetReturnValue().Set(instance);
}

}  // namespace demo
```

测试：

```
// test.js
const addon = require('./build/Release/addon');

var obj1 = addon.createObject(10);
var obj2 = addon.createObject(20);
var result = addon.add(obj1, obj2);

console.log(result); // 30
```

AtExit 挂钩

“AtExit” 挂钩是一个函数，它是在 Node.js 事件循环结束后，但在 JavaScript 虚拟机被终止或 Node.js 关闭前调用。“AtExit” 挂钩使用 `node::AtExit` API 注册。

void AtExit(callback, args)

- `callback` ： `void (*)(void*)` - 一个退出时调用的函数的指针。
- `args` ： `void*` - 一个退出时传递给回调的指针。

注册在事件循环结束后并在虚拟机被终止前运行的退出挂钩。

AtExit 有两个参数：指向一个在退出运行的回调函数，和要传递给该回调的无类型的上下文数据的指针。

回调按照后进先出的顺序执行。

下面的 `addon.cc` 实现了 AtExit：

```
// addon.cc
#undef NDEBUG
#include <assert.h>
#include <stdlib.h>
#include <node.h>

namespace demo {

using node::AtExit;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;

static char cookie[] = "yum yum";
static int at_exit_cb1_called = 0;
static int at_exit_cb2_called = 0;

static void at_exit_cb1(void* arg) {
  Isolate* isolate = static_cast<Isolate*>(arg);
  HandleScope scope(isolate);
  Local<Object> obj = Object::New(isolate);
  assert(!obj.IsEmpty()); // assert VM is still alive
  assert(obj->IsObject());
  at_exit_cb1_called++;
}

static void at_exit_cb2(void* arg) {
  assert(arg == static_cast<void*>(cookie));
  at_exit_cb2_called++;
}

static void sanity_check(void*) {
  assert(at_exit_cb1_called == 1);
  assert(at_exit_cb2_called == 2);
}

void init(Local<Object> exports) {
  AtExit(sanity_check);
  AtExit(at_exit_cb2, cookie);
  AtExit(at_exit_cb2, cookie);
  AtExit(at_exit_cb1, exports->GetIsolate());
}

NODE_MODULE(addon, init);

}  // namespace demo
```

在 JavaScript 中运行测试：

```
// test.js
const addon = require('./build/Release/addon');
```

Node.js的原生抽象

文档中所示的每个例子都直接使用 Node.js 和 V8 API 实现插件。理解这一点很重要，V8 API 可以，并且已经在下一个（主要的） V8 发行版本中发生了巨大的变化。伴随着每一个变化，插件为了能够继续工作，可能需要进行更新和重新编译。Node.js 的发布计划是为了尽量减少这种变化的频率和影响，但有一点 Node.js 目前可以做到的是确保 V8 API 的稳定性。

Node.js的原生抽象（或 `nan`）提供了一组推荐插件开发者使用的用以保持过去和未来的 V8 和 Node.js 版本之间的兼容性的工具。详见 `nan` 例子 了解它是如何使用的说明例证。

附录

附录包含着一些在原文档中未曾提供的篇章，这些篇章不属于原文档，但可能对部分有需求的用户会有很大的帮助。

函数速查表

A

assert(value[, message])

assert.equal(actual, expected[, message])

assert.strictEqual(actual, expected[, message])

assert.notEqual(actual, expected[, message])

assert.notStrictEqual(actual, expected[, message])

assert.deepEqual(actual, expected[, message])

assert.deepStrictEqual(actual, expected[, message])

assert.notDeepEqual(actual, expected[, message])

assert.notDeepStrictEqual(actual, expected[, message])

assert.ok(value[, message])

assert.fail(actual, expected, message, operator)

assert.ifError(value)

assert.throws(block[, error][, message])

assert.doesNotThrow(block[, error][, message])

agent.createConnection(options[, callback])

agent.getName(options)

agent.destroy()

B

Buffer类

buffer.INSPECT_MAX_BYTES

Buffer.byteLength(string[, encoding])

Buffer.isBuffer(obj)

Buffer.isEncoding(encoding)

Buffer.from(array)

Buffer.from(buffer)

Buffer.from(arrayBuffer[, byteOffset[, length]])

Buffer.from(str[, encoding])

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

Buffer.allocUnsafe(size)

Buffer.concat(list[, totalLength])

Buffer.compare(buf1, buf2)

buf.length

buf[index]

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

buf.toJSON()

buf.slice([start[, end]])

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

buf.indexOf(value[, byteOffset][, encoding])

buf.includes(value[, byteOffset][, encoding])

buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])

buf.compare(otherBuffer)

buf.equals(otherBuffer)

buf.entries()

buf.keys()

buf.values()

buf.swap16()

buf.swap32()

buf.readIntBE(offset, byteLength[, noAssert])

buf.readIntLE(offset, byteLength[, noAssert])

buf.readFloatBE(offset[, noAssert])

buf.readFloatLE(offset[, noAssert])

buf.readDoubleBE(offset[, noAssert])

buf.readDoubleLE(offset[, noAssert])

buf.readInt8(offset[, noAssert])

buf.readInt16BE(offset[, noAssert])

buf.readInt16LE(offset[, noAssert])

buf.readInt32BE(offset[, noAssert])

buf.readInt32LE(offset[, noAssert])

buf.readUIntBE(offset, byteLength[, noAssert])

buf.readUIntLE(offset, byteLength[, noAssert])

buf.readUInt8(offset[, noAssert])

buf.readUInt16BE(offset[, noAssert])

buf.readUInt16LE(offset[, noAssert])

buf.readUInt32BE(offset[, noAssert])

buf.readUInt32LE(offset[, noAssert])

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

buf.writeIntBE(value, offset, byteLength[, noAssert])

buf.writeIntLE(value, offset, byteLength[, noAssert])

buf.writeFloatBE(value, offset[, noAssert])

buf.writeFloatLE(value, offset[, noAssert])

buf.writeDoubleBE(value, offset[, noAssert])

buf.writeDoubleLE(value, offset[, noAssert])

buf.writeInt8(value, offset[, noAssert])

buf.writeInt16BE(value, offset[, noAssert])

buf.writeInt16LE(value, offset[, noAssert])

buf.writeInt32BE(value, offset[, noAssert])

buf.writeInt32LE(value, offset[, noAssert])

buf.writeUIntBE(value, offset, byteLength[, noAssert])

buf.writeUIntLE(value, offset, byteLength[, noAssert])

buf.writeUInt8(value, offset[, noAssert])

buf.writeUInt16BE(value, offset[, noAssert])

buf.writeUInt16LE(value, offset[, noAssert])

buf.writeUInt32BE(value, offset[, noAssert])

buf.writeUInt32LE(value, offset[, noAssert])

C

Console类

console

console.log([data][, ...args])

console.dir(obj[, options])

console.info([data][, ...args])

console.warn([data][, ...args])

console.error([data][, ...args])

console.trace(message[, ...args])

console.assert(value[, message][, ...args])

console.time(label)

console.timeEnd(label)

clearTimeout(timeout)

clearInterval(timeout)

clearImmediate(immediate)

child_process.exec(command[, options][, callback])

child_process.execFile(file[, args][, options][, callback])

child_process.spawn(command[, args][, options])

child_process.fork(modulePath[, args][, options])

child_process.execSync(command[, options])

child_process.execFileSync(file[, args][, options])

child_process.spawnSync(command[, args][, options])

ChildProcess类

child.pid

child.connected

child.stdio

child.stdin

child.stdout

child.stderr

[child.send(message[, sendHandle[, options]], callback])

child.disconnect()

child.kill([signal])

cluster.setupMaster([settings])

cluster.fork([env])

cluster.disconnect([callback])

cryptoStream.bytesWritten

crypto.DEFAULT_ENCODING

crypto.createCipher(algorithm, password)

crypto.createCipheriv(algorithm, key, iv)

crypto.createCredentials(details)

crypto.createDecipher(algorithm, password)

crypto.createDecipheriv(algorithm, key, iv)

crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding])

crypto.createDiffieHellman(prime_length[, generator])

crypto.createECDH(curve_name)

crypto.createHash(algorithm)

crypto.createHmac(algorithm, key)

crypto.createSign(algorithm)

crypto.createVerify(algorithm)

crypto.getCiphers()

crypto.getCurves()

crypto.getDiffieHellman(group_name)

crypto.getHashes()

crypto.pbkdf2(password, salt, iterations, keylen[, digest], callback)

crypto.pbkdf2Sync(password, salt, iterations, keylen[, digest])

crypto.privateEncrypt(private_key, buffer)

crypto.privateDecrypt(private_key, buffer)

crypto.publicEncrypt(public_key, buffer)

crypto.publicDecrypt(public_key, buffer)

crypto.randomBytes(size[, callback])

crypto.setEngine(engine[, flags])

cipher.setAAD(buffer)

cipher.setAutoPadding(auto_padding=true)

cipher.getAuthTag()

cipher.update(data[, input_encoding][, output_encoding])

cipher.final([output_encoding])

certificate.exportPublicKey(spkac)

certificate.exportChallenge(spkac)

certificate.verifySpkac(spkac)

D

decoder.write(buffer)

decoder.end()

dns.setServers(servers)

dns.getServers()

dns.resolve(hostname[, rrtype], callback)

dns.resolve4(hostname, callback)

dns.resolve6(hostname, callback)

dns.resolveMx(hostname, callback)

dns.resolveTxt(hostname, callback)

dns.resolveSrv(hostname, callback)

dns.resolveNs(hostname, callback)

dns.resolveCname(hostname, callback)

dns.resolveSoa(hostname, callback)

dns.reverse(ip, callback)

dns.lookup(hostname[, options], callback)

dns.lookupService(address, port, callback)

domain.create()

domain.members

domain.bind(callback)

domain.intercept(callback)

domain.run(fn[, arg][, ...])

domain.add(emitter)

domain.remove(emitter)

domain.enter()

domain.exit()

domain.dispose()

dgram.createSocket(options[, callback])

dgram.createSocket(type[, callback])

decipher.setAAD(buffer)

decipher.setAutoPadding(auto_padding=true)

decipher.setAuthTag(buffer)

decipher.update(data[, input_encoding][, output_encoding])

decipher.final([output_encoding])

diffieHellman.generateKeys([encoding])

diffieHellman.getGenerator([encoding])

diffieHellman.setPublicKey(public_key[, encoding])

diffieHellman.getPublicKey([encoding])

diffieHellman.setPrivateKey(private_key[, encoding])

diffieHellman.getPrivateKey([encoding])

diffieHellman.getPrime([encoding])

diffieHellman.computeSecret(other_public_key[, input_encoding][, output_encoding])

E

exports

EventEmitter.defaultMaxListeners

emitter.on(eventName, listener)

emitter.once(eventName, listener)

emitter.addListener(eventName, listener)

emitter.prependListener(eventName, listener)

emitter.prependOnceListener(eventName, listener)

emitter.removeListener(eventName, listener)

emitter.removeAllListeners([eventName])

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

emitter.listeners(eventName)

emitter.listenerCount(eventName)

emitter.setMaxListeners(n)

emitter.getMaxListeners()

Error.captureStackTrace(targetObject[, constructorOpt])

Error.stackTraceLimit

error.message

error.stack

error.errno

error.code

error.syscall

ecdh.generateKeys([encoding[, format]])

ecdh.setPrivateKey(private_key[, encoding])

ecdh.getPrivateKey([encoding])

ecdh.setPublicKey(public_key[, encoding])

ecdh.getPublicKey([encoding[, format]])

ecdh.computeSecret(other_public_key[, input_encoding][, output_encoding])

F

fs.readdir(path, callback)

fs.readdirSync(path)

fs.mkdir(path[, mode], callback)

fs.mkdirSync(path[, mode])

fs.rmdir(path, callback)

fs.rmdirSync(path)

fs.realpath(path[, cache], callback)

fs.realpathSync(path[, cache])

fs.link(srcpath, dstpath, callback)

fs.linkSync(srcpath, dstpath)

fs.symlink(target, path[, type], callback)

fs.symlinkSync(target, path[, type])

fs.readlink(path, callback)

fs.readlinkSync(path)

fs.unlink(path, callback)

fs.unlinkSync(path)

fs.lchmod(path, mode, callback)

fs.lchmodSync(path, mode)

fs.lchown(path, uid, gid, callback)

fs.lchownSync(path, uid, gid)

fs.lstat(path, callback)

fs.lstatSync(path)

fs.createReadStream(path[, options])

fs.read(fd, buffer, offset, length, position, callback)

fs.readSync(fd, buffer, offset, length, position)

fs.createWriteStream(path[, options])

fs.write(fd, data[, position[, encoding]], callback)

fs.writeSync(fd, data[, position[, encoding]])

fs.write(fd, buffer, offset, length[, position], callback)

fs.writeSync(fd, buffer, offset, length[, position])

fs.truncate(path, len, callback)

fs.truncateSync(path, len)

fs.stat(path, callback)

fs.statSync(path)

fs.chmod(path, mode, callback)

fs.chmodSync(path, mode)

fs.chown(path, uid, gid, callback)

fs.chownSync(path, uid, gid)

fs.utimes(path, atime, mtime, callback)

fs.utimesSync(path, atime, mtime)

fs.exists(path, callback)

fs.existsSync(path)

fs.open(path, flags[, mode], callback)

fs.openSync(path, flags[, mode])

fs.close(fd, callback)

fs.closeSync(fd)

fs.access(path[, mode], callback)

fs.accessSync(path[, mode])

fs.rename(oldPath, newPath, callback)

fs.renameSync(oldPath, newPath)

fs.watch(filename[, options][, listener])

fs.watchFile(filename[, options], listener)

fs.unwatchFile(filename[, listener])

fs.readFile(file[, options], callback)

fs.readFileSync(file[, options])

fs.writeFile(file, data[, options], callback)

fs.writeFileSync(file, data[, options])

fs.appendFile(file, data[, options], callback)

fs.appendFileSync(file, data[, options])

fs.ftruncate(fd, len, callback)

fs.ftruncateSync(fd, len)

fs.fstat(fd, callback)

fs.fstatSync(fd)

fs.fchmod(fd, mode, callback)

fs.fchmodSync(fd, mode)

fs.fchown(fd, uid, gid, callback)

fs.fchownSync(fd, uid, gid)

fs.futimes(fd, atime, mtime, callback)

fs.futimesSync(fd, atime, mtime)

fs.fsync(fd, callback)

fs.fsyncSync(fd)

fs.fdatasync(fd, callback)

fs.fdatasyncSync(fd)

fs.ReadStream 类

fs.WriteStream 类

fs.Stats 类

fs.FSWatcher 类

G

global

H

http.createServer([requestListener])

http.createClient([port][, host])

http.request(options[, callback])

http.get(options[, callback])

https.createServer(options[, requestListener])

https.request(options, callback)

https.get(options, callback)

hash.update(data[, input_encoding])

hash.digest([encoding])

hmac.update(data[, input_encoding])

hmac.digest([encoding])

I

J

K

L

M

module.id

module.filename

module.loaded

module.parent

module.children

module.exports

module.require(id)

message.setTimeout(msecs, callback)

N

new Console(stdout[, stderr])

new Error(message)

new Buffer(size)

new Buffer(array)

new Buffer(buffer)

new Buffer(arrayBuffer[, byteOffset[, length]])

new Buffer(str[, encoding])

new SlowBuffer(size)

new stream.Readable([options])

new stream.Writable([options])

new stream.Duplex(options)

new stream.Transform([options])

net.createServer([options][, connectionListener])

net.createConnection(options[, connectListener])

net.createConnection(path[, connectListener])

net.createConnection(port[, host][, connectListener])

net.connect(options[, connectListener])

net.connect(path[, connectListener])

net.connect(port[, host][, connectListener])

net.isIP(input)

net.isIPv4(input)

net.isIPv6(input)

net.Server 类

new net.Socket([options])

new crypto.Certificate()

O

os.EOL

os.type()

os.release()

os.platform()

os.cpus()

os.arch()

os.endianness()

os.loadavg()

os.totalmem()

os.freemem()

os.uptime()

os.networkInterfaces()

os.hostname()

os.homedir()

os.tmpdir()

P

process

punycode.version

punycode.ucs2

punycode.ucs2.encode(codePoints)

punycode.ucs2.decode(string)

punycode.encode(string)

punycode.decode(string)

punycode.toUnicode(domain)

punycode.toASCII(domain)

process.stdin

process.stdout

process.stderr

process.pid

process.config

process.env

process.platform

process.arch

process.release

process.title

process.connected

process.exitCode

process.mainModule

process.argv

process.execPath

process.execArgv

process.version

process.versions

[process.send(message[, sendHandle[, options]], callback])

process.nextTick(callback[, arg][, ...])

process.disconnect()

process.exit([code])

process.abort()

process.kill(pid[, signal])

process.cwd()

process.chdir(directory)

process.memoryUsage()

process.umask([mask])

process.uptime()

process.hrtime()

process.setuid(id)

process.getuid()

process.setgid(id)

process.getgid()

process.seteuid(id)

process.geteuid()

process.setegid(id)

process.getegid()

process.setgroups(groups)

process.getgroups()

process.initgroups(user, extra_group)

path.resolve([from ...], to)

path.relative(from, to)

path.format(pathObject)

path.parse(pathString)

path.join([path1][, path2][, ...])

path.normalize(p)

path.dirname(p)

path.basename(p[, ext])

path.extname(p)

path.isAbsolute(path)

Q

R

require()

require.cache

require.extensions

require.resolve()

readable.read([size])

readable.setEncoding(encoding)

readable.pipe(destination[, options])

readable.unpipe([destination])

readable.unshift(chunk)

readable.pause()

readable.isPaused()

readable.resume()

readable.wrap(stream)

readable._read(size)

readable.push(chunk[, encoding])

request.setNoDelay([noDelay])

request.setSocketKeepAlive([enable][, initialDelay])

request.flushHeaders()

request.write(chunk[, encoding][, callback])

request.end([data][, encoding][, callback])

request.abort()

response.setHeader(name, value)

response.getHeader(name)

response.removeHeader(name)

response.addTrailers(headers)

response.writeHead(statusCode[, statusMessage][, headers])

response.write(chunk[, encoding][, callback])

response.writeContinue()

response.end([data][, encoding][, callback])

response.setTimeout(msecs, callback)

rs.setRawMode(mode)

repl.start([options])

replServer.defineCommand(keyword, cmd)

replServer.displayPrompt([preserveCursor])

readline.createInterface(options)

readline.cursorTo(stream, x, y)

readline.moveCursor(stream, dx, dy)

readline.clearLine(stream, dir)

readline.clearScreenDown(stream)

rl.write(data[, key])

rl.setPrompt(prompt)

rl.prompt([preserveCursor])

rl.question(query, callback)

rl.pause()

rl.resume()

rl.close()

S

setTimeout(callback, delay[, ...args])

setInterval(callback, delay[, ...args])

setImmediate(callback[, ...args])

System Error类

SlowBuffer类

stream.Readable类（流消费者）

stream.Writable类（流消费者）

stream.Duplex类（流消费者）

stream.Transform类（流消费者）

stream.Readable类（流实现者）

stream.Writable类（流实现者）

stream.Duplex类（流实现者）

stream.Transform类（流实现者）

stream.PassThrough类（流实现者）

StringDecoder类

Script类

script.runInThisContext([options])

script.runInContext(contextifiedSandbox[, options])

script.runInNewContext([sandbox][, options])

server.maxConnections

server.connections

server.listening

server.getConnections(callback)

server.listen(port[, hostname][, backlog][, callback])

server.listen(path[, backlog][, callback])

server.listen(handle[, backlog][, callback])

server.listen(options[, callback])

server.address()

server.close([callback])

server.unref()

server.ref()

socket.setEncoding([encoding])

socket.setTimeout(timeout[, callback])

socket.setNoDelay([noDelay])

socket.setKeepAlive([enable][, initialDelay])

socket.connect(path[, connectListener])

socket.connect(port[, host][, connectListener])

socket.connect(options[, connectListener])

socket.write(data[, encoding][, callback])

socket.pause()

socket.resume()

socket.end([data][, encoding])

socket.destroy()

socket.address()

socket.unref()

socket.ref()

server.listen(handle[, callback])

server.listen(path[, callback])

server.listen(port[, hostname][, backlog][, callback])

server.close([callback])

server.setTimeout(msecs, callback)

server.listen(port[, hostname][, callback])

server.addContext(hostname, context)

server.setTicketKeys(keys)

server.getTicketKeys()

server.close([callback])

server.address()

socket.address()

socket.bind(options[, callback])

socket.bind([port][, address][, callback])

socket.send(msg, [offset, length,] port, address[, callback])

socket.setTTL(ttl)

socket.setMulticastTTL(ttl)

socket.setMulticastLoopback(flag)

socket.setBroadcast(flag)

socket.close([callback])

socket.addMembership(multicastAddress[, multicastInterface])

socket.dropMembership(multicastAddress[, multicastInterface])

socket.unref()

socket.ref()

sign.update(data[, input_encoding])

sign.sign(private_key[, output_format])

T

timeout.unref()

timeout.ref()

transform._transform(chunk, encoding, callback)

transform._flush(callback)

tls.createServer(options[, secureConnectionListener])

tls.connect(options[, callback])

tls.connect(port[, host][, options][, callback])

tls.createSecureContext(options)

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])

tls.getCiphers()

tlsSocket.address()

tlsSocket.getTLSTicket()

tlsSocket.getPeerCertificate([detailed])

tlsSocket.getCipher()

tlsSocket.getEphemeralKeyInfo()

tlsSocket.getProtocol()

tlsSocket.getSession()

tlsSocket.renegotiate(options, callback)

tlsSocket.setMaxSendFragment(size)

tty.isatty(fd)

tty.setRawMode(mode)

U

util.isBoolean(object)

util.isNumber(object)

util.isString(object)

util.isObject(object)

util.isArray(object)

util.isFunction(object)

util.isNull(object)

util.isUndefined(object)

util.isNullOrUndefined(object)

util.isSymbol(object)

util.isError(object)

util.isRegExp(object)

util.isDate(object)

util.isBuffer(object)

util.isPrimitive(object)

util.inspect(object[, options])

util.format(format[, ...])

util.inherits(constructor, superConstructor)

util.log(string)

util.error([...])

util.debug(string)

util.debuglog(section)

util.print([...])

util.puts([...])

util.pump(readableStream, writableStream[, callback])

util.deprecate(function, string)

url.parse(urlStr[, parseQueryString][, slashesDenoteHost])

url.format(urlObj)

url.resolve(from, to)

V

v8.setFlagsFromString(string)

v8.getHeapStatistics()

v8.getHeapSpaceStatistics()

vm.createContext([sandbox])

vm.isContext(sandbox)

vm.runInThisContext(code[, options])

vm.runInContext(code, contextifiedSandbox[, options])

vm.runInNewContext(code[, sandbox][, options])

vm.runInDebugContext(code)

verifier.update(data[, input_encoding])

verifier.verify(object, signature[, signature_format])

W

writable.write(chunk[, encoding][, callback])

writable.setDefaultEncoding(encoding)

writable.cork()

writable.uncork()

writable.end([chunk][, encoding][, callback])

writable._write(chunk, encoding, callback)

writable._writev(chunks, callback)

worker.send(message[, sendHandle][, callback])

worker.disconnect()

worker.kill([signal='SIGTERM'])

worker.isConnected()

worker.isDead()

X

Y

Z

zlib.Zlib 类

zlib.Gzip 类

zlib.Gunzip 类

zlib.Unzip 类

zlib.Deflate 类

zlib.Inflate 类

zlib.DeflateRaw 类

zlib.InflateRaw 类

zlib.flush([kind], callback)

zlib.params(level, strategy, callback)

zlib.reset()

zlib.createGzip([options])

zlib.createGunzip([options])

zlib.createUnzip([options])

zlib.createDeflate([options])

zlib.createInflate([options])

zlib.createDeflateRaw([options])

zlib.createInflateRaw([options])

zlib.gzip(buf[, options], callback)

zlib.gzipSync(buf[, options])

zlib.gunzip(buf[, options], callback)

zlib.gunzipSync(buf[, options])

zlib.unzip(buf[, options], callback)

zlib.unzipSync(buf[, options])

zlib.deflate(buf[, options], callback)

zlib.deflateSync(buf[, options])

zlib.inflate(buf[, options], callback)

zlib.inflateSync(buf[, options])

zlib.deflateRaw(buf[, options], callback)

zlib.deflateRawSync(buf[, options])

zlib.inflateRaw(buf[, options], callback)

zlib.inflateRawSync(buf[, options])

其他

__dirname

__filename