目錄

当前分支文档翻译自 Node.js 6.x，持续更新中……

开发列表

章节	进度	认领人	审校人
Assertion Testing	已完成	@pinggod	未审校
Buffer	翻译中...	未认领	未审校
C/C++ Addons	翻译中...	未认领	未审校
Child Processes	翻译中...	未认领	未审校
Cluster	翻译中...	@Crazydogs	未审校
Command Line Options	翻译中...	未认领	未审校
Console	翻译中...	未认领	未审校
Crypto	翻译中...	未认领	未审校
Debugger	翻译中...	未认领	未审校
DNS	翻译中...	未认领	未审校
Domain	已完成	@pinggod	@pinggod
Errors	翻译中...	未认领	未审校
Events	翻译中...	未认领	未审校
File System	翻译中...	未认领	未审校
Globals	翻译中...	未认领	未审校
HTTP	翻译中...	未认领	未审校
HTTPS	翻译中...	未认领	未审校
Modules	翻译中...	未认领	未审校
Net	翻译中...	未认领	未审校
OS	翻译中...	未认领	未审校
Path	翻译中...	未认领	未审校
Process	翻译中...	未认领	未审校
Punycode	已完成	@pinggod	@pinggod
Query Strings	翻译中...	未认领	未审校
Readline	翻译中...	未认领	未审校

REPL	翻译中...	未认领	未审校
Stream	已完成	@Crazydogs	未审校
String Decoder	翻译中...	未认领	未审校
Timers	翻译中...	未认领	未审校
TLS/SSL	翻译中...	未认领	未审校
TTY	翻译中...	未认领	未审校
UDP/Datagram	翻译中...	未认领	未审校
URL	翻译中...	未认领	未审校
Utilities	翻译中...	未认领	未审校
V8	翻译中...	未认领	未审校
VM	翻译中...	未认领	未审校
ZLIB	翻译中...	未认领	未审校

翻译说明

- 只翻译 LTS 版本，也就是 6.x / 8.x ...
- 已废弃（Stability: 0）的模块、函数不翻译，只标注该模块、函数已废弃

贡献者

- @pinggod
- @Crazydogs

当前文档翻译自 Node.js 6.x 官方文档，致力于从概念和实践两方面介绍 Node.js API。文档整体分为多个章节，每一章节针对一个模块或一个高阶概念展开。

稳定性

在阅读文档的过程中，你会经常看到如下四种稳定性标识，用于标识当前 API 的稳定程度：

接口稳定性: 0 - 已过时

当前模块存在已知问题，不建议继续使用该模块；继续使用该模块时，Node.js 系统会抛出警告信息，无法有效保障兼容性。

接口稳定性: 1 - 实验中

当前模块正在开发中，需要使用命令行参数启动，未来可能会被修改或移除。

接口稳定性: 2 - 稳定

当前模块整体表现稳定，除非绝对需要，否则不会修改；以 NPM 开发环境的兼容性为优先开发原则。

接口稳定性: 3 - 已锁定

当前模块的功能已锁定，不接受新的 API 建议，后续只会进行安全、性能或 Bug 方面的维护工作。

> 本文档不会翻译稳定性为 0 的模块，也不建议开发者使用此类模块。

系统调用和 man 页面

类似 open(2) 和 read(2) 的系统调用命令直接存在于用户程序与底层操作系统之间。对于 Node.js 简单封装的系统调用命令都会在文档中显式标注，比如 `fs.open()`。对于系统调用命令，文档会直接链接到相应的 man 页面。

警告：部分系统调用命令是某些系统特有的，比如 lchown(2) 就是 BSD Unix 特有的。也就是说，`fs.lchown()` 只能用于 Mac OS X 和其他 BSD 衍生操作系统，不能用于 Linux 和 Windows 系统。

大多数的 Unix 系统调用命令在 Windows 上都有类似的系统调用命令，但两者的行为可能不同，在某些时候两者微妙的差异导致了绝对的不可替代性，详细信息请参考 Node issue 4760。

使用方式

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

有关 options 的详细信息请参考 Command Line Options 一章。

示例

使用 Node.js 创建 web 服务器：

```
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}/`)
;
});
```

将上述代码保存为 `example.js` ，然后在终端中调用 `node` 命令解释执行：

```
$ 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上面的方法进行测试。

Assert

> 接口稳定性: 3 - 已锁定

`assert` 模块提供了一系列的断言测试函数，设计这些辅助函数的初衷是服务于 Node.js 自身的开发。当然，开发者可以通过 `require('assert')` 将其应用到第三方模块的开发中。不过，`assert` 模块并不是一个测试框架，不建议将其用作通用的断言库。

`assert` 模块当前的 API 已处于锁定状态，这意味着该模块将不再轻易新增或修改 API。

assert(value[, message])

等同于 `assert.ok()`：

```
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.deepEqual(actual, expected[, message])

比较参数 `actual` 和 `expected` 是否深度相等。对于参数中的原始值（primitive value）使用 `==` 进行比较。

`deepEqual()` 只会遍历比较对象自身可枚举的属性，不会比较对象的原型、symbol 和不可枚举的属性，所以在某些情况下结果可能会出人意料。比如在下面的示例中就不会抛出 `AssertionError` ，这是因为 Error 对象的属性是不可枚举的：

```
// WARNING: This does not throw an AssertionError!
assert.deepEqual(Error('a'), Error('b'));
```

方法名中的 `Deep` 一词意指会对传入的两个对象进行深度比较，包括对对象子级可枚举属性的比较：

```
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 {}
// 原型继承而来的属性不计入比较范围
```

如果 `actual` 和 `expected` 不相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.deepStrictEqual(actual, expected[, message])

`assert.deepStrictEqual()` 与 `assert.deepEqual()` 存在两点不同，一是使用 `===` 判断原始值是否相等，二是会比较 `actual` 和 `expected` 的原型对象是否严格相等（即指向同一内存地址的原型对象）：

```
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 `actual` 和 `expected` 不相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.doesNotThrow(block[, error][, message])

`assert.doesNotThrow()` 期望传入的 `block` 函数不会抛出错误，更多信息请查看 `assert.thorws()` 。

执行 `assert.doesNotThrow()` 的同时会立即执行 `block` 函数。

如果 `block` 函数抛出了错误，且错误类型与 `error` 参数指定的类型相符，就会抛出 `AssertionError` 错误；如果 `block` 函数抛出的错误类型与 `error` 参数指定的类型不符，或者未传入可选参数 `error` ，则错误会被抛出给调用者（caller）。

在下面的示例中，由于传入的 `error` 参数为 `SyntaxError` 与 `block` 函数的错误不匹配，所以会抛出 `block` 函数产生的 `TypeError` ：

```
assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  SyntaxError
);
```

在下面的示例中，`error` 参数指定的错误与 `block` 函数抛出的错误匹配，所以抛出 `AssertionError` ：

```
assert.doesNotThrow(
  () => {
    throw new TypeError('Wrong value');
  },
  TypeError
);
```

如果提供了可选参数 `message` ，那么该字符串会被添加到 `AssertionError` 信息之后：

```
assert.doesNotThrow(
    () => {
        throw new TypeError('Wrong value');
    },
    TypeError,
    'Whoops'
);
// Throws: AssertionError: Got unwanted exception (TypeError). W
hoops
```

assert.equal(actual, expected[, message])

使用 `==` 比较 `actual` 参数和 `expected` 参数是否相等：

```
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 `actual` 和 `expected` 不相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.fail(actual, expected, message, operator)

抛出 `AssertionError` 错误和 `message` 错误信息，如果 `message` 为假值（falsy），则系统自动生成格式为 `actual #{operator} expected` 的错误信息：

```
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（truthy），则抛出 `value`，这对于测试回调函数中的 `error` 参数很有用：

```
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.notDeepEqual(actual, expected[, message])

与 `assert.deepEqual()` 的功能相反：

```
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 `actual` 和 `expected` 深度相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.notDeepStrictEqual(actual, expected[, message])

与 `assert.notDeepStrictEqual()` 的功能相反：

```
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 `actual` 和 `expected` 深度严格相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.notEqual(actual, expected[, message])

使用 `!=` 比较 `actual` 参数和 `expected` 参数是否不相等：

```
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 `actual` 和 `expected` 相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.notStrictEqual(actual, expected[, message])

使用 `!==` 比较 `actual` 参数和 `expected` 参数是否不严格相等：

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

assert.notStrictEqual(1, 2);
// OK

assert.notStrictEqual(1, 1);
// AssertionError: 1 != 1

assert.notStrictEqual(1, '1');
// OK
```

如果 `actual` 和 `expected` 严格相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.ok(value[, message])

检验参数 `value` 是否为真值（truthy），等同于 `assert.equal(!!value, true, message)`：

```
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"
```

如果 `value` 不是真值，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.strictEqual(actual, expected[, message])

使用 `===` 比较 `actual` 参数和 `expected` 参数是否严格相等：

```
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 `actual` 和 `expected` 不严格相等，则抛出 `AssertionError` 错误和 `message` 错误信息。如果未定义 `message` 参数，Node.js 会提供默认的错误信息。

assert.throws(block[, error][, message])

`assert.throws()` 期望传入的 `block` 函数会抛出错误（译者注：如果 `block` 函数抛出错误，`assert.throws()` 无返回值，表示正常；如果未抛出错误，则 `assert.throws()` 抛出 AssertionError 错误），可选参数 `error` 可以是构造函数、正则表达式和自定义的检验函数。

使用构造函数校验错误实例：

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  Error
);
```

使用正则表达式校验错误信息：

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /value/
);
```

使用自定义函数校验错误实例和错误信息：

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  function(err) {
    if ( (err instanceof Error) && /value/.test(err) ) {
      return true;
    }
  },
  'unexpected error'
);
```

注意，`error` 参数不能是字符串。如果该参数是字符串，则会被识别为 `message` 参数继而导致错误的结果，这是非常容易被忽略的错误用法：

```
// THIS IS A MISTAKE! DO NOT DO THIS!
assert.throws(myFunction, 'missing foo', 'did not throw with exp
ected message');

// Do this instead.
assert.throws(myFunction, /missing foo/, 'did not throw with exp
ected message');
```

Buffer

接口稳定性: 2 - 稳定

在 ECMAScript 2015（ES6）引入 `TypedArray` 之前，JavaScript 语言中并没有读取和操作二进制数据流的机制。Node.js API 中的 `Buffer` 类提供了处理八进制数据流的能力，常用于处理 TCP 数据流和文件操作。

在 ES6 增加了 `TypedArray` 类型之后，`Buffer` 类基于 `Uint8Array` 接口做了定向优化，从而使其更适用于 Node.js 的实际需求。

`Buffer` 类的实例非常类似整数数组，但其大小是固定不变的，在 V8 堆栈外分配原始内存空间。`Buffer` 类的实例创建之后，其所占用的内存大小就不能再进行调整。

`Buffer` 类是 Node.js 的全局对象，所以不需要使用 `require()` 而直接使用：

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

// Creates a Buffer of length 10, filled with 0x1.
const buf2 = Buffer.alloc(10, 1);

// 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 buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing [0x1, 0x2, 0x3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing ASCII bytes [0x74, 0x65, 0x73, 0x
74].
const buf5 = Buffer.from('test');

// Creates a Buffer containing UTF-8 bytes [0x74, 0xc3, 0xa9, 0x
73, 0x74].
const buf6 = Buffer.from('tést', 'utf8');
```

Buffer.from() / Buffer.alloc() / Buffer.allocUnsafe()

在 Node.js v6.x 之前，通常使用 `new Buffer()` 创建 Buffer 实例，其缺点在于系统会根据参数类型的不同而返回不同类型的 Buffer 实例：

- 如果传入的第一个参数是数值（比如 `new Buffer(10)`），则生成一个特定长度的 Buffer 实例，但是此类 Buffer 实例分配到的内存是未经初始化和包含敏感数据的内存。此类 Buffer 实例必须通过 `buf.fill(0)` 或一次完整的数据重写手动完成内存的初始化操作。虽然这种方法的设计初衷是改善性能，但是过往的开发经验表明，对于创建快速未初始化（fast-but-uninitialized）和慢速已初始化（slower-but-safer）的 Buffer 实例，我们需要职责更加清晰的构造函数
- 如果传入的第一个参数是字符串、数组或 Buffer 实例，则系统会将它们的数据

拷贝到新的 Buffer 实例中
- 如果传入的参数是 ArrayBuffer 的实例，则生成的 Buffer 实例与该 ArrayBuffer 实例共享同一段内存

由于 `new Buffer()` 会根据第一个参数的类型触发不同的处理逻辑，所以当应用程序没有正确校验 `new Buffer()` 的参数或没有正确初始化 Buffer 实例的内容时，就会在不经意间向代码中引入安全性和可靠性问题。

为了避免 Buffer 实例的此类问题，建议使用 `Buffer.from()` 、 `Buffer.alloc()` 和 `Buffer.allocUnsafe()` 方法替代 `new Buffer()` 。

开发者应该使用以下函数替换 `new Buffer()` 的操作：

- `Buffer.from(array)` ，创建并返回一个包含 `array` 数据的 Buffer 实例
- `Buffer.from(arrayBuffer[, byteOffset [, length]])` ，创建并返回一个与 `ArrayBuffer` 共享内存片段的 Buffer 实例
- `Buffer.from(buffer)` ，创建并返回一个包含 `buffer` 数据的 Buffer 实例
- `Buffer.from(str[, encoding])` ，创建并返回一个包含 `str` 数据的 Buffer 实例
- `Buffer.alloc(size[, fill[, encoding]])` ，创建并返回一个已初始化的 Buffer 实例，长度为 `size` ，该方法在性能上虽然明显比 `Buffer.allocUnsafe(size)` 慢很多，但可以保证新建的 Buffer 实例绝对不会包含敏感或遗留数据
- `Buffer.allocUnsafe(size)` 和 `Buffer.allocUnsafeSlow(size)` 都会返回 `size` 大小的 Buffer 实例，但不会自动初始化内存片段，必须使用 `buf.fill(0)` 或重写内存片段的方式进行初始化。

如果 `Buffer.allocUnsafe(size)` 的 `size` 小于或等于 `Buffer.poolSize` 的一半，那么 Buffer 实例就有可能由内部共享的内存池分配内存片段，而使用 `Buffer.allocUnsafeSlow(size)` 生成的 Buffer 实例则不会使用内部共享的内存池。

`--zero-fill-buffers`

如果在命令行启动 Node.js 时附加 `--zero-fill-buffers` 参数，则强制新建 Buffer 实例时自动将内存片段初始化为 0。使用该参数会改变系统的默认行为，并对系统性能造成影响。建议只在处理非敏感数据时使用该参数：

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

为什么 `Buffer.allocUnsafe(size)` 和 `Buffer.allocUnsafeSlow(size)` 不安全?

通过 `Buffer.allocUnsafe(size)` 和 `Buffer.allocUnsafeSlow(size)` 生成的 Buffer 实例，其内存片段都是未经初始化的。虽然这种操作执行速度快，但这些内存片段可能包含敏感的历史数据，那么当系统读取 Buffer 实例时，就有可能发生内存泄露。

虽然 `Buffer.allocUnsafe()` 的执行性能很好，但使用时必须十分小心，避免因此造成安全问题。

Buffer 和字符编码

Buffer 实例常用于处理编码后的字符序列，比如 UTF8 / UCS2 / Base64 甚至是十六进制编码后的数据。通过指定字符编码，数据可以在 Buffer 实例和原始的 JavaScript 字符串之间来回转换：

```
const buf = Buffer.from('hello world', 'ascii');

// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('hex'));

// Prints: aGVsbG8gd29ybGQ=
console.log(buf.toString('base64'));
```

当前 Node.js 支持以下字符编码格式：

- `ascii`，仅支持 7 位（7-bit）的 ASCII 字符，该方法解析速度快，还可以自动去除高位字节
- `utf8`，使用多字节对 unicode 字符进行编码。诸多 web 页面和文档都在采用 UTF-8 编码规范
- `utf16le`，使用二到四个字节、小端字节序对 unicode 字符进行编码，支持

解码代理对（surrogate pair, U+10000 ~ U+10FFFF）
- `ucs2`，`utf16le` 的别名。
- `base64`，Base64 编码。对于使用字符串创建的 Buffer 实例，`base64` 编码格式允许该字符串是 RFC4648, Section 5 中指定的 URL 和文件名安全字符集（URL and Filename Safe Alphabet）。
- `latin1`，将 Buffer 实例转换为单字节（latin-1）字符串的编码格式（as defined by the IANA in RFC1345, page 63, to be the Latin-1 supplement block and C0/C1 control codes）
- `binary`，`latin1` 的别名
- `hex`，将每个字节转换为两个十六进制的字符

> 当今浏览器一般都遵循了 WHATWG 规范，该规范将 `latin1` 和 `ISO-8859` 都视为是 `win-1252` 的别名。这意味着，如果你通过 `http.get()` 获取到后端数据，且数据编码格式是上述 WHATWG 规范中规定的一种，那么你很有可能会接收到编码格式为 `win-1252` 的数据，此时使用 `latin1` 格式对数据解码就有可能会出错。

Buffer 和 TypedArray

Buffer 实例实际上也是 `TypedArray` 中 `Uint8Array` 的实例。不过，这里的 `TypedArray` 与 ECMAScript 2015 规范所规定的 `TypedArray` 稍有不同。举例来说，ECMAScript 2015 规范规定 `ArrayBuffer#slice()` 方法创建一个内存拷贝，而 Node.js 中 `Buffer#slice()` 则会根据既有的 Buffer 实例新建一个视图（View，译者注：ES2015 中有两种视图，分别是 TypedArray 和 DataView），而不是拷贝数据，从而提高执行效率。

基于一个 `Buffer` 实例创建一个 `TypedArray` 实例需要注意以下几点事项：

1. `Buffer` 实例的内存数据会被拷贝到 `TypedArray` 实例中，它们之间不是内存共享的关系。
2. `Buffer` 实例的内存数据会被解释为一个直观的整数数组，而不是一个特定类型的单字节数组。举例来说，`new Uint32Array(new Buffer([1,2,3,4]))` 会创建一个 `Uint32Array` 视图，它包含 `[1,2,3,4]` 四个元素，而不是创建一个 `Uint32Array` 类型的单元素数组 `[0x1020304]` 或 `[0x4030201]`。

如果要创建一个和 `TypedArray` 实例共享内存的 `Buffer` 实例，可以使用 TypedArray 实例的 `.buffer` 属性：

```
const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`
const buf1 = Buffer.from(arr);

// Shares memory with `arr`
const buf2 = Buffer.from(arr.buffer);

// Prints: <Buffer 88 a0>
console.log(buf1);

// Prints: <Buffer 88 13 a0 0f>
console.log(buf2);

arr[1] = 6000;

// Prints: <Buffer 88 a0>
console.log(buf1);

// Prints: <Buffer 88 13 70 17>
console.log(buf2);
```

使用 `TypedArray` 实例的 `.buffer` 属性创建 Buffer 实例时，可以使用 `byteOffset` 和 `length` 参数截取 `ArrayBuffer` 实例的内存数据：

```
const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);

// Prints: 16
console.log(buf.length);
```

`Buffer.from()` 和 `TypedArray.from()`（比如 `Uint8Array.from()`）拥有不同的参数和实现，举例来说，TypedArray 接收一个映射函数（mapping function）作为第二个参数，该函数会遍历处理 TypedArray 实例的每一个元素：

- `TypedArray.from(source[, mapFn[, thisArg]])`

但是 `Buffer.from` 函数不具有这种用法：

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

Buffers 和 ES6 遍历器

使用 ECMAScript 2015 提供的 `for...of` 语法可以遍历 Buffer 实例：

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

// Prints:
//   1
//   2
//   3
for (var b of buf) {
  console.log(b);
}
```

此外，使用 `buf.values()`、`buf.keys()` 和 `buf.entries()` 函数可以创建遍历器实例。

Class: Buffer

Buffer 类是一个用于处理二进制数据的全局对象。通过该对象，Node.js 提供了多种方法来创建 Buffer 实例。

new Buffer()

从 Node.js 6+ 开始，已废弃该构造函数，使用以下函数替换：

- 使用 `Buffer.from(array)` 替换 `new Buffer(array)`
- 使用 `Buffer.from(buffer)` 替换 `new Buffer(buffer)`
- 使用 `Buffer.from(arrayBuffer[, byteOffset[, length]])` 替换 `new Buffer(arrayBuffer[, byteOffset[, length]])`
- 使用 `Buffer.alloc(size)` 或 `Buffer.allocUnsafe(size)` 替换 `new Buffer(size)`
- 使用 `Buffer.from(string[, encoding])` 替换 `new Buffer(string[, encoding])`

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

成员方法：Buffer.byteLength(string[, encoding])

- `string`，字符串
- `encoding`，字符串格式的可选参数，默认值为 `utf8`
- 返回值类型：Number

返回字符串的实际字节长度。与 `String.prototype.length` 不同的是，该方法返回数值表示字符串占多少字节，而 `String.prototype.length` 返回的数值表示字符串有多少个字符。

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

console.log(`${str}: ${str.length} characters, ` +
            `${Buffer.byteLength(str, 'utf8')} bytes`);
// ½ + ¼ = ¾: 9 个字符, 12 个字节
```

成员方法：Buffer.compare(buf1, buf2)

- `buf1`，Buffer 实例
- `buf2`，Buffer 实例
- 返回值类型：Number

比较 `buf1` 和 `buf2` 常常是为了对 Buffer 实例的数组进行排序，该方法等同于 `buf1.compare(buf2)`：

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

成员方法：Buffer.concat(list[, totalLength])

- `list`，用于合并的 Buffer 实例数组
- `totalLength，数值类型的可选参数，用于说明数组中所有 Buffer 实例的字节之和
- 返回值类型：Buffer

返回一个 Buffer 实例，该实例是传入的 Buffer 实例数组中所有元素合并后生成的。如果实例数组为空，或者 `totalLength` 为 0，则返回一个长度为 0 的 Buffer 实例。

如果没有传入 `totalLength` 参数，则系统根据 `list` 参数中所有实例的大小自动求取。不过，这需要系统通过循环计算出来字节总数，所以通过提供该参数可以提高系统的执行效率。

```
const buf1 = new Buffer(10).fill(0);
const buf2 = new Buffer(14).fill(0);
const buf3 = new Buffer(18).fill(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
```

成员方法：Buffer.isBuffer(obj)

- `obj`，对象类型的参数
- 返回值类型：Boolean

如果 `obj` 为 Buffer 实例，则返回 `true`，反之异然。

成员方法：Buffer.isEncoding(encoding)

- `encoding`，字符串形式的字符串编码说明
- 返回值类型：Boolean

如果 `encoding` 参数为合法的字符串编码，则返回 `true`，反之异然。

buf[index]

`[index]` 索引操作可以用于从 Buffer 实例的指定位置存取（get and set）二进制数据。`index` 的值为单字节数据，所以在这里索引的合法范围是 0x00 ~ 0xFF（十六进制）或者 0 ~ 255（十进制）。

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

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

console.log(buf);
// 输出结果: Node.js
```

buf.compare(otherBuffer)

- `otherBuffer`，Buffer 实例
- 返回值类型：Number

比较两个 Buffer 实例，返回一个数值，用于标识 `buf` 和 `otherBuffer` 的先后顺序。比较过程是基于每个 Buffer 实例的元素顺序进行的。

- `0` 表示 `buf` 和 `otherBuffer` 相同
- `1` 表示排序时 `otherBuffer` 应该排在 `buf` 之前
- `-1` 表示排序时 `otherBuffer` 应该排在 `buf` 之后

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

console.log(buf1.compare(buf1));
// 输出结果: 0
console.log(buf1.compare(buf2));
// 输出结果: -1
console.log(buf1.compare(buf3));
// 输出结果: -1
console.log(buf2.compare(buf1));
// 输出结果: 1
console.log(buf2.compare(buf3));
// 输出结果: 1

[buf1, buf2, buf3].sort(Buffer.compare);
// 输出结果: [buf1, buf3, buf2]
```

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

- `targetBuffer`，Buffer 实例，拷贝目标（将 buf 的数据拷贝到 targetBuffer)
- `targetStart`，默认值为 `0`
- `sourceStart`，默认值为 `0`
- `sourceEnd`，默认值为 `buffer.length`
- 返回值类型：Number，说明拷贝的字节总数

该方法用于将数据从 `buf` 拷贝到 `targetBuffer`，即使对自身进行拷贝也没有问题。

```
const buf1 = new Buffer(26);
const buf2 = new Buffer(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!!!!!!!!!!!!!
```

下面代码演示了对自身进行的拷贝：

```
const buf = new Buffer(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.entries()

- 返回值类型：Iterator

根据 Buffer 实例的数据创建并返回一个格式为 `[index, byte]` 的遍历器：

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

buf.equals(otherBuffer)

- `otherBuffer`，Buffer 实例
- 返回值类型：Boolean

该方法用于判断两个 Buffer 实例是否具有相同的值，如果相等则返回 `true`，反之异然。

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

console.log(buf1.equals(buf2));
// 输出结果: true
console.log(buf1.equals(buf3));
// 输出结果: false
```

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

- `value`，字符串或者数值
- `offset`，数值，默认值为 0
- `end`，数值，默认值为 `buffer.length`
- 返回值类型：Buffer

将 Buffer 实例的每个值置为 `value` 参数所表示的值。如果未传入了 `offset` 和 `end` 参数，默认重置 Buffer 实例的所有元素。因为该实例方法返回 Buffer 实例自身，所以可以执行链式调用。

```
const b = new Buffer(50).fill('h');
console.log(b.toString());
// 输出结果: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
```

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

- `value`，字符串，Buffer 实例或者数值
- `byteOffset`，默认值为 0
- `encoding`，默认值为 'utf8'
- 返回值类型：Number

该实例方法类似于 `Array#indexOf()` 方法，如果查找成功，则返回 `value` 的起始位置；如果查找失败，则返回 `-1`。`value` 参数可以是字符串、Buffer 实例或者数值。如果传入的 `vlaue` 参数是字符串类型，默认被解释为 `UTF8` 编码格式；如果传入的 `value` 参数是 Buffer 实例，默认查找与 Buffer 实例整体内容相匹配的值（使用 `buf.slice()` 可以查找 Buffer 实例的部分值）；如果传入的 `value` 参数是数值类型，则只能使用 0 ~ 255 之间数值（如果值为负，则表示从后往前查找）。

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

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

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

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

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

- `value`，字符串，Buffer 实例或者数值
- `byteOffset`，默认值为 0
- `encoding`，默认值为 'utf8'
- 返回值类型：Number

该实例方法类似于 `Array#includes()`，其中 `value` 参数可以是字符串、Buffer 实例或者数值。。如果传入的 `vlaue` 参数是字符串类型，默认被解释为 `UTF8` 编码格式；如果传入的 `value` 参数是 Buffer 实例，默认查找与 Buffer 实例整体内容相匹配的值（使用 `buf.slice()` 可以查找 Buffer 实例的部分值）；如果传入的 `value` 参数是数值类型，则只能使用 0 ~ 255 之间数值（如果值为负，则表示从后往前查找）。

可选参数 `byteOffset` 表示在 `buf` 中检索的起点位置：

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

buf.includes('this');
// 返回结果： true
buf.includes('is');
// 返回结果： true
buf.includes(new Buffer('a buffer'));
// 返回结果： true
buf.includes(97);
// ascii for 'a'
// 返回结果： true
buf.includes(new Buffer('a buffer example'));
// 返回结果： false
buf.includes(new Buffer('a buffer example').slice(0,8));
// 返回结果： true
buf.includes('this', 4);
// 返回结果： false
```

buf.keys()

- 返回值类型：Iterator

根据 Buffer 实例的键创建和返回一个 Iterator:

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

buf.length

- 返回值类型：Number

该属性返回 Buffer 实例在内存中所占有的字节大小。值得注意的是，该属性并不表示 Buffer 实例的使用量，比如下面的这个例子，虽然 Buffer 实例占有 1234 个字节，但是只有 11 个字节用于存储 ASCII 字符，其他空间处于闲置状态：

```
const buf = new Buffer(1234);

console.log(buf.length);
// 输出结果: 1234

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

该属性是不可变属性（immutable），修改该属性将会返回 undefined，且会影响 Buffer 实例的正常使用。如果你想修改 `length` 属性，可以变通地使用 `buf.slice()` 方法创建一个新 Buffer 实例：

```
const buf = new Buffer(10);
buf.write('abcdefghj', 0, 'ascii');
console.log(buf.length);
// 输出结果: 10
buf = buf.slice(0,5);
console.log(buf.length);
// 输出结果: 5
```

buf.rendDoubleBE(offset[, noAssert])

buf.rendDoubleLE(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 8`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 64 位的双精度浮点数（double 类型），如果使用的是 `readDoubleBE()`，则使用大端字节序读取；如果使用的是 `readDoubleLE()`，则使用小端字节序读取。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 8` 。

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

buf.readDoubleBE();
// 返回结果: 8.20788039913184e-304
buf.readDoubleLE();
// 返回结果: 5.447603722011605e-270
buf.readDoubleLE(1);
// throws RangeError: 索引越界

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

buf.rendFloatBE(offset[, noAssert])

buf.rendFloatLE(offset[, noAssert])

- `offset` ，数值，取值范围为 `0 <= offset <= buf.length - 4`
- `noAssert` ，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 32 位的单精度浮点数（float 类型），如果使用的是 `readFloatBE()` ，则使用大端字节序读取；如果使用的是 `readFloatLE()` ，则使用小端字节序读取。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 4` 。

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

buf.readFloatBE();
// 返回结果: 2.387939260590663e-38
buf.readFloatLE();
// 返回结果: 1.539989614439558e-36
buf.readFloatLE(1);
// throws RangeError: 索引越界

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

buf.reanInt8(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 1`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个单字节整数（int 类型）。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 1`。

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

buf.readInt8(0);
// 返回结果: 1
buf.readInt8(1);
// 返回结果: -2
```

buf.readInt16BE(offset[, noAssert])

buf.readInt16LE(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 2`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 16 位的双字节整数（int 类型），如果使用的是 `readInt16BE()`，则使用大端字节序读取；如果使用的是 `readInt16LE()`，则使用小端字节序读取。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 2`。

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

buf.readInt16BE();
// 返回结果: 510
buf.readInt16LE();
// 返回结果: -511
```

buf.readInt32BE(offset[, noAssert])

buf.readInt32LE(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 4`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 32 位的双字节整数（int 类型），如果使用的是 `readInt32BE()`，则使用大端字节序读取；如果使用的是 `readInt32LE()`，则使用小端字节序读取。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 4`。

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

buf.readInt32BE();
// 返回结果: 33424132
buf.readInt32LE();
// 返回结果: 67370497
```

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

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

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - byteLength`
- `byteLength`，数值，取值范围为 `0 < byteLength <= 6`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个长度为 `byteLength` 的字节，最高精度为 48 位。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - byteLength`。

```
const buf = new Buffer(6);
buf.writeUInt16LE(0x90ab, 0);
buf.writeUInt32LE(0x12345678, 2);
buf.readIntLE(0, 6).toString(16);
// 返回结果: '1234567890ab'

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

buf.readUInt8(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 1`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 8 位的无符号整数。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 1`。

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

buf.readUInt8(0);
// 返回结果: 1
buf.readUInt8(1);
// 返回结果: 254
```

buf.readUInt16BE(offset[, noAssert])

buf.readUInt16LE(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 2`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 16 位的无符号整数（int 类型），如果使用的是 `readUInt16BE()`，则使用大端字节序读取；如果使用的是 `readUInt16LE()`，则使用小端字节序读取。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 2`。

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

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

buf.readUInt32BE(offset[, noAssert])

buf.readUInt32LE(offset[, noAssert])

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - 4`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个 32 位的无符号整数（int 类型），如果使用的是 `readUInt32BE()`，则使用大端字节序读取；如果使用的是 `readUInt32LE()`，则使用小端字节序读取。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 4`。

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

buf.readUInt32BE(0);
// 返回结果: 0x03042342
console.log(buf.readUInt32LE(0));
// 返回结果: 0x42230403
```

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

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

- `offset`，数值，取值范围为 `0 <= offset <= buf.length - byteLength`
- `byteLength`，数值，取值范围为 `0 < byteLength <= 6`
- `noAssert`，布尔值，默认值为 false
- 返回值类型：Number

从 Buffer 实例中 `offset` 位置开始读取一个长度为 `byteLength` 的字节，最高精度为 48 位。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - byteLength`。

```
const buf = new Buffer(6);
buf.writeUInt16LE(0x90ab, 0);
buf.writeUInt32LE(0x12345678, 2);
buf.readUIntLE(0, 6).toString(16);
// 返回结果: '1234567890ab'

buf.readUIntBE(0, 6).toString(16);
// 返回结果: ab9078563412
```

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

- `start`，数值，默认值为 0
- `end`，数值，默认值为 `buffer.length`
- 返回值类型：Buffer

根据可选参数 `start` 和 `end` 指定的位置从 `buf` 中切片出新的 Buffer 实例，且它们共享同一段内存。

因为两个 Buffer 实例共享内存，所以其中一个实例修改数据后，另一个实例得到的就是修改后的数据。

```
const buf1 = new Buffer(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);
// 返回结果: 'abc'
buf1[0] = 33;
buf2.toString('ascii', 0, buf2.length);
// 返回结果 : '!bc'
```

使用负值索引可以从后往前执行切片：

```
const buf = new Buffer('buffer');

buf.slice(-6, -1).toString();
// 返回结果: 'buffe', 等同于 buf.slice(0, 5)
buf.slice(-6, -2).toString();
// 返回结果: 'buff', 等同于 buf.slice(0, 4)
buf.slice(-5, -2).toString();
// 返回结果: 'uff', 等同于 buf.slice(1, 4)
```

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

- `encoding`，字符串，默认值为 `utf8`
- `start`，数值，默认值为 0
- `end`，数值，默认值为 `buffer.length`
- 返回值类型：String

该方法根据 `encoding` 指定的编码格式，从 Buffer 实例存储的数据中编码和返回一个字符串：

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

buf.toJSON()

- 返回值类型：Object

该方法返回一个 JSON 对象，用于描述调用它的 Buffer 实例。如果 `JSON.stringify()` 收到的参数是一个 Buffer 实例，那么它会隐式调用 `buf.toJSON()` 进行解析：

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

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

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

console.log(copy.toString());
// 输出结果: 'test'
```

buf.values()

- 返回值类型：Iterator

根据 Buffer 实例的值创建和返回一个 Iterator 对象。当使用 `for...of` 遍历 Buffer 实例时，系统会隐式调用该方法解析 Buffer 实例：

```
const buf = new Buffer('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.write(string[, offset[, length[, encoding]]])

- `string`，字符串，用于写入到 Buffer 实例中数据
- `offset`，数值，默认值为 0
- `length`，数值，默认值为 `buffer.length - offset`
- `encoding`，字符串，默认值为 `utf8`
- 返回值类型：Number，表示成功写入的字节数量

该方法按照 `encoding` 参数指定的编码格式从 `offset` 位置开始向 Buffer 实例中写入 `string` 参数所引用的数据。`length` 参数显式声明要写入到 Buffer 实例的字节数量。如果 Buffer 实例无法容纳所有写入的数据，那么就只会解析和写入部分数据：

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - 8`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，如果使用的是 `writeDoubleBE()`，则使用大端字节序写入；如果使用的是 `writeDoubleLE()`，则使用小端字节序写入。`value` 参数引用的数据必须是有效的 64 位双精度浮点数（double 类型）。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 8`，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - 4`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，如果使用的是 `writeFloatBE()`，则使用大端字节序写入；如果使用的是 `writeFloatLE()`，则使用小端字节序写入。`value` 参数引用的数据必须是有效的 32 位单精度浮点数（float 类型）。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 4`，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

buf.writeFloatLE(0xcafebabe, 0);

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - 1`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，该数据必须是有效地单字节整数（int 类型）。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 1` ，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

```
const buf = new Buffer(2);
buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);
console.log(buf);
// 输出结果: <Buffer 02 fe>
```

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

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

- `value` ，数值，用于写入到 Buffer 实例的数据
- `offset` ，数值，取值范围 `0 <= offset <= buf.length - 2`
- `noAssert` ，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，如果使用的是 `writeInt16BE()` ，则使用大端字节序写入；如果使用的是 `writeInt16LE()` ，则使用小端字节序写入。 `value` 参数引用的数据必须是有效的 16 位双字节整数。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 2` ，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - 4`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，如果使用的是 `writeInt32BE()`，则使用大端字节序写入；如果使用的是 `writeInt32LE()`，则使用小端字节序写入。`value` 参数引用的数据必须是有效的 32 位四字节整数。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 4`，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - byteLength`
- `byteLength`，数值，取值范围 `0 < byteLength <= 6`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，长度为 `byteLength`，最高精度为 48 位。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - byteLength` ，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

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

- `value` ，数值，用于写入到 Buffer 实例的数据
- `offset` ，数值，取值范围 `0 <= offset <= buf.length - 1`
- `noAssert` ，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，该数据必须是无符号的单字节整数（int 类型）。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 1` ，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - 2`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，如果使用的是 `writeUInt16BE()`，则使用大端字节序写入；如果使用的是 `writeUInt16LE()`，则使用小端字节序写入。`value` 参数引用的数据必须是无符号的双字节整数。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 2`，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

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

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - 4`

- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，如果使用的是 `writeUInt32BE()`，则使用大端字节序写入；如果使用的是 `writeUInt32LE()`，则使用小端字节序写入。`value` 参数引用的数据必须是无符号的四字节整数。

如果传入可选参数 `noAssert` 且值为 `true`，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - 4`，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

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

buf.writeUInt32LE(0xfeedface, 0);

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

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

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

- `value`，数值，用于写入到 Buffer 实例的数据
- `offset`，数值，取值范围 `0 <= offset <= buf.length - byteLength`
- `byteLength`，数值，取值范围 `0 < byteLength <= 6`
- `noAssert`，布尔值，默认值为 `false`
- 返回值类型：Number，表示成功写入的字节数量

从 Buffer 实例中 `offset` 位置开始写入 `value` 参数所引用的数据，长度为 `byteLength`，最高精度为 48 位。

如果传入可选参数 `noAssert` 且值为 `true` ，则执行该方法时忽略参数 `offset` 是否符合取值范围 `0 <= offset <= buf.length - byteLength` ，同时忽略 `value` 参数所引用的数据是否过长。除非确信数据的准确性，否则不建议忽略对 `value` 和 `offset` 参数的检查。

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

buffer.INSPECT_MAX_BYTES

- 数值，默认值为 50

该属性用于设置 `buffer.inspect()` 方法返回的最大字节数，可以被开发者自定义的模块修改，更多有关 `buffer.inspect()` 方法的信息请参考 `util.inspect()` 方法。

值得注意的是，该属性并不挂载在全局对象 Buffer 或者 Buffer 实例上，而是存在于 `require('buffer')` 的返回值中。

Class: SlowBuffer

`SlowBuffer` 类用于创建 un-pooled（不放入内存池？）的 Buffer 实例。

为了降低创建 Buffer 实例的开销，避免 Buffer 实例冗杂凌乱的现象，默认情况下对于小于 4KB 的 Buffer 实例使用单个大内存对象存储和管理。这种方式可以有效提高性能和内存利用率，避免 V8 频繁追踪和清理过多的 `Persistent` 对象。

有时候，开发者希望对内存中的一小块空间保留一个不确定的时间，针对这种情况，就可以使用 `SlowBuffer` 类创建 un-pooled（不放入内存池？）的 Buffer 实例，然后通过内存拷贝获取目标数据：

```
// 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 = new SlowBuffer(10);
  // copy the data into the new allocation
  data.copy(sb, 0, 0, 10);
  store.push(sb);
});
```

通过 `SlowBuffer` 类创建 Buffer 实例的方式应该被视为一种最终手段，除非开发者观察到应用中有过多不必要的内存，否则不要使用这种方式。

插件

Node.js 的插件是基于 C/C++ 编写的动态链接共享对象（dynamically-linked shared objects），可以像普通的 Node.js 模块一样通过 `require()` 函数加载到 Node.js 的开发环境中。Node.js 插件主要用于支持 Node.js 中 JavaScript 调用 C/C++ 库的接口。

目前，创建 Node.js 插件的方法比较复杂，需要了解以下几个组件和 API：

- V8：基于 C++ 函数库，当前 Node.js 的 JavaScript 引擎。V8 实现了 JavaScript 创建对象、调用函数等诸多底层机制。V8 的 API 主要集中在 v8.h 头文件（详见 Node.js 项目中的 `deps/v8/include/v8.h`）中，更多信息请参考线上地址 https://v8docs.nodesource.com/。
- libuv：基于 C 的函数库，实现了 Node.js 的事件循环机制、worker 线程机制以及 Node.js 上所有与异步相关的机制。同时，它是一个跨平台的抽象库，提供了简洁、类 POSIX 的接口，方便开发者调用主流操作系统的常规系统任务，比如与文件系统、网络套接字、定时器以及系统事件的交互。此外，libuv 实现了一个类 pthreads 的线程机制，便于开发者在标准事件循环机制之上构建更强大的异步插件。libuv 鼓励开发者思考如何避免 I/O、任务加载失败等因素对系统操作、worker 线程以及自定义 libuv 线程的阻塞行为。
- Node.js 内建库：Node.js 本身内建了一系列 C/C++ API 供插件使用，其中最重要的就是 `node::ObjectWrap` 类。
- Node.js 内置了一系列静态链接库，比如 OpenSSL，其中大部分位于项目的 `deps/` 目录内，只有 V8 和 OpenSSL 刻意被 Node.js 进行了重定向输出，便于诸多插件的调用，更多信息请参考Linking to Node.js's own dependencies。

以下所有示例都可以在 https://github.com/nodejs/node-addon-examples 下载到。如果你想要开发 Node.js 的插件，也可以使用这些示例作为初始模板。

Hello World

该示例是使用 C++ 开发插件的简单示例，等同于下面这段 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
```

值得注意的是，开发 Node.js 插件的开发时，必须输出一个初始化函数，模式如下：

```
void Initialize(Local<Object> exports);
NODE_MODULE(module_name, Initialize)
```

在 `NODE_MODULE` 的行末没有分号，因为它并不是一次函数调用（详见 `node.h` ）。`module_name` 必须二进制文件名相匹配（不包含文件名的 `.node` 后缀）。通过上述代码，我们在 `hello.cc` 文件中声明了初始化函数是 `init` ，插件名称是 `addon` 。

构建

在上面的源代码编写完成后，需要将其编译为二进制文件 `addon.node` 。首先，在项目的根目录创建一个 `binding.gyp` 文件，使用类 JSON 格式配置构建信息。该文件创建完成后会被 `node-gyp` 用来编译模块， `node-gyp` 是一个专门用于编译 Node.js 插件的工具。

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

> 注意：Node.js 中捆绑了一个特殊版本的 `node-gyp` 工具集，它是 `npm` 的一部分，用于支持 `npm install` 命令编译和安装插件，但是不能直接被开发者使用。如果开发者想直接使用 `node-gyp` ，需要使用 `npm install -g node-gyp` 命令安装 `node-gyp` ，更多信息请参考 node-gyp 的安装指南，其中包括了对特定平台的需求信息。

创建完 `bindings.gyp` 文件之后，即可使用 `node-gyp configure` 命令生成适用于当前平台的构建文件，同时会在 `build` 目录下生成一个适用于 UNIX 平台的 `Makefile` 或者适用于 Windows 平台的 `vcxproj` 文件。

接下来，调用 `node-gyp build` 命令在 `build/Release/` 目录下编译生成 `addon.node` 文件。

当使用 `npm install` 命令安装 Node.js 的插件时，npm 会使用自身捆绑的 `node-gyp` 再次编译和生成适用于用户平台的相关文件。

构建完成后，就可以在 Node.js 的开发中使用 `require()` 加载 `addon.node` 模块了：

```
// hello.js
const addon = require('./build/Release/addon');

console.log(addon.hello()); // 'world'
```

更多信息请参考下面的几个示例，或者查看用于生产环境的示例 https://github.com/arturadib/node-qt。

因为插件编译后二进制文件路径（有时候可能是 `./build/Debug/`）由编译方式确定，所以建议使用 bindings 包加载编译后的模块。

值得注意的是，`bindings` 包定位插件模块的方式非常类似于 `try-catch`：

```
try {
  return require('./build/Release/addon.node');
} catch (err) {
  return require('./build/Debug/addon.node');
}
```

链接到 Node.js 自身的依赖

Node.js 自身用到了许多静态链接库，比如 V8、libuv 以及 OpenSSL。所有的插件都可以链接到这些静态链接库，链接方式非常简单，只需要声明 `#include <...>`（比如：`#include <v8.h>`）语句，`node-gyp` 就会自动定位到合适的头文件。不过，开发时需要注意以下几点：

- 运行 `node-gyp` 时，它会检测 Node.js 的特定版本并下载源码或者头文件信息。如果下载的是源码，那么插件就可以完整的使用 Node.js 的依赖；如果下载的只是头文件信息，那么只能使用 Node.js 输出的特定依赖。
- 使用 `node-gyp` 命令时，可以添加 `--nodedir` 标志，用于指定本地的 Node.js 源数据。开启这个可选参数之后，插件可以使用 Node.js 的全部依赖。

使用 require() 加载插件

插件编译后的二进制文件使用 `.node` 作为后缀名，通过 `require()` 函数可以查找以 `.node` 为扩展名的文件并将它们初始化为动态链接库。

调用 `require()` 时，`.node` 扩展名虽然可以被 Node.js 查找到并初始化为动态链接库，但是有一个问题值得注意，那就是如果插件名字和其他文件重名了，Node.js 会优先加载 Node.js 模块和 JavaScript 文件。举例来说，如果在同一目录下有 `addon.js` 和 `addon.node` 两个文件，那么 `require('addon')` 函数会优先加载 `addon.js` 文件。

Node.js 的本地抽象

本文档中的每个示例都直接使用了 Node.js 和 V8 的 API 开发插件，所以有一点非常值得注意，那就是 V8 API 随着版本升级仍在快速迭代之中。对于 V8 和 Node.js 的版本升级，插件需要重新修改和编译以适应新的 API。当前 Node.js 在发布计划中尽量减少 API 的变动对插件所带来的影响，但是这无法确保 V8 API 的稳定性。

Native Abstrations for Node.js（简称 NAN）提供了一系列工具帮助插件开发者在新旧版本的 V8 和 Node.js 之间保持插件的一致性。更多有多 NAN 的使用信息请参考 NAN 开发实例。

插件实例

以下内容是用于帮助开发者快读上手的插件实例，它们都在开发过程中调用了 V8 API。通过在线的 V8 指南 可以了解更多有关 V8 调用的信息，也可以参考 Embedder 的使用指南 了解 V8 的核心概念，比如句柄、作用域、函数模板等等。

每一个示例都使用了如下的 `binding.gyp` 文件：

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "addon.cc" ]
    }
  ]
}
```

如果插件涉及多个 `.cc` 文件，可以使用将其依次添加入 `source` 字段所引用的数组中：

```
"sources": ["addon.cc", "myexample.cc"]
```

`binding.gyp` 创建完成后，最后使用 `node-gyp` 生成配置文件和编译成二进制文件：

```
$ node-gyp configure build
```

函数参数

插件通常都会暴漏某些对象和函数给 Node.js 中的 JavaScript 调用。当 JavaScript 调用函数时，也必须将将传入的参数映射给 C/C++ 代码，在函数调用完成后，还要映射 C/C++ 传回的返回值。

下面代码演示了如何读取 JavaScript 传递来的函数以及如何传输返回值：

```
// 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
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// test.js
const addon = require('./build/Release/addon');

console.log('This should be eight:', addon.add(3,5));
```

回调函数

在 Node.js 开发环境中使用 JavaScript 通过插件向 C++ 函数传递并执行一个回调函数是非常常见的操作，下面代码演示了如何在 C++ 中执行回调函数：

```
// 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, "hell
o 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
```

值得注意的是，在上面的示例中 `Init` 函数使用了两个参数，其中第二个参数用于接收完整的 `module` 对象。这种做法便于插件通过简单的函数重写 `exports`，而无需将函数挂载在 `exports` 之下。

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// test.js
const addon = require('./build/Release/addon');

var obj1 = addon('hello');
var obj2 = addon('world');
console.log(obj1.msg+' '+obj2.msg); // 'hello world'
```

```
// test.js
const addon = require('./build/Release/addon');

addon((msg) => {
  console.log(msg); // 'hello world'
});
```

> 在上面的示例中，回调函数是以同步方式调用的。

对象工厂

在下面的代码中演示了如何使用 C++ 函数创建和返回一个新对象，新对象有一个 `msg` 属性：

```
// 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]->ToStrin
g());

  args.GetReturnValue().Set(obj);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(addon, Init)

}  // namespace demo
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// 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

另一个常见操作就是通过 C++ 函数创建 Javascript 函数并将其返回到 JavaScript 开发环境中：

```
// 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, M
yFunction);
  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
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// test.js
const addon = require('./build/Release/addon');

var fn = addon();
console.log(fn()); // 'hello world'
```

包装 C++ 对象

在某些情况下，可以通过包装 C++ 对象或类让 JavaScript 使用 `new` 操作符创建新的实例：

```
// 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
```

然后，在 `myobject.h` 头文件中，包装类继承 `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>& arg
s);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>&
 args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

在 `myobject.cc` 文件中，具体实现需要暴漏出来的代码逻辑。在下面的代码中，通过将 `plusOne()` 方法挂载在构造器的原型上，将其暴漏给了外部：

```
// 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, N
ew);
  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]->NumberV
```

```
alue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into cons
truct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constru
ctor);
    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
```

构建该示例之前，需要将 `myobject.cc` 文件名添加到 `binding.gyp` 文件中：

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// 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
```

包装对象工厂方法

此外，可以使用工厂模式在 JavaScript 中隐式创建对象实例：

```
var obj = addon.createObject();
// instead of:
// var obj = new addon.Object();
```

首先，需要在 `addon.cc` 文件中实现 `createObject()` 方法的逻辑：

```
// 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` 头文件中，添加静态方法 `NewInstance()` 用于实例化对象，该方法用于替换 JavaScript 中的 `new` 操作：

```
// 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::Val
ue>& args);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& arg
s);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>&
 args);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

`myobject.cc` 文件的具体实现与前例代码非常类似：

```
// 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, N
ew);
  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]->NumberV
alue();
    MyObject* obj = new MyObject(value);
```

```
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into cons
truct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constru
ctor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& ar
gs) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, construct
or);
  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
```

构建示例之前，先将 `myobject.cc` 文件名添加到 `binding.gyp` 配置文件中：

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// 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
```

传递包装对象

在包装盒返回 C++ 对象之外，还可以使用 Node.js 中的辅助函数 `node::ObjectWrap::Unwrap` 解包包装对象。在下面的示例中演示了函数 `add()` 如何解析传入的两个对象参数：

```
// 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` 头文件中，添加一个新的公共方法，用于调用对象解包后的私有属性：

```
// 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::Val
ue>& args);
  inline double value() const { return value_; }

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& arg
s);
  static v8::Persistent<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

`myobject.cc` 的具体实现和前面的示例类似：

```
// 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, N
ew);
  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]->NumberV
alue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into cons
truct call.
```

```
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constru
ctor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& ar
gs) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, construct
or);
  Local<Object> instance = cons->NewInstance(argc, argv);

  args.GetReturnValue().Set(instance);
}

}  // namespace demo
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// 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 钩子函数

"AtExit" 钩子函数在 Node.js 事件循环结束之后、JavaScript VM 终止以及 Node.js 退出之前调用。需要使用 "node::AtExit" 接口注册 "AtExit" 钩子函数。

void AtExit(callback, args)

- `callback`，`void (*)(vodi*)`，一个指向函数的指针，函数结束时调用
- `args`，`void*`，函数结束是传递给 `callback` 的指针

注册 AtExit 钩子函数需要在事件循环之后和 VM 退出之前。`callback` 按照后入先出的顺序执行。下面 `addon.cc` 文件中的代码实现了 AtExit 钩子函数：

```
// 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
```

编译成功之后，在 Node.js 环境中使用 `require()` 函数加载插件：

```
// test.js
const addon = require('./build/Release/addon');
```

子进程

接口稳定性: 2 - 稳定

`child_process` 模块实现了创建子进程的机制，在某些方面类似于 popen(3)，但并不完全相同。这种机制主要由 `child_process.spawn()` 函数实现：

```
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，系统会在 Node.js 主进程和子进程之间为 `stdin`、`stdout` 和 `stderr` 创建数据通道，这些数据通道可以以非阻塞的方式进行数据传输。值得注意的是，某些程序内部使用的是线性缓冲 I/O，虽然这不会影响 Node.js，但这意味着从主进程发给子进程的数据不能被立即使用啊。

`child_process.spawn()` 方法用于异步创建子进程，不会阻塞 Node.js 的事件循环机制。`child_process.spawnSync()` 方法与上一个方法提供了相同的功能，不同之处在于它是同步执行的，执行时会阻塞事件循环机制，知道子进程结束任务并退出。

为了调用方便，`child_process` 模块提供了一系列基于 `child_process.spawn()` 和 `child_process.spawnSync()` 重新包装的方法：

- `child_process.exec()`：生成一个 shell 并执行指定的命令，执行完成后调用传入的回调函数，该回调函数包含两个参数 `stdout` 和 `stderr`
- `child_process.execFile()`：功能与 `child_process.exec()` 相似，不同之处在于无需生成 shell，直接执行指定的命令
- `child_process.fork()`：生成一个新的 Node.js 进程并调用特定模块创建 IPC 信道，用于在父子进程之间进行通信
- `child_process.execSync()`，`child_process.exec()` 方法的同步执行版本，执行时阻塞 Node.js 的事件循环机制
- `child_process.execFileSync()`，`child_process.execFile()` 方法的同步执行版本，执行时阻塞 Node.js 的事件循环机制

在某些情况下，同步执行的方法可能更加合适，比如创建自动化 shell 脚本。不过在大多数情况下，由于必须执行完同步方法才能继续事件循环机制，所以此类方法通常会严重影响 Node.js 的性能。

创建异步进程

`child_process` 模块中的 `spawn()` / `fork()` / `exec()` 和 `execFile()` 方法是 Node.js 中典型的异步编程范式。

这些方法执行完成后会返回一个 `ChildProcess` 实例，这些实例都实现了 Node.js EventEmitter 的 API，便于父进程注册监听函数监听子进程生命周期中发生的事件。

此外，`child_process.exec()` 和 `child_process.execFile()` 方法还允许传递一个可选的回调函数，该回调函数会在子进程结束时被调用。

在 Windows 平台执行 `.bat` 和 `.cmd` 文件

`child_process.exec()` 和 `child_process.execFile()` 两个方法之间的差异取决于当前 Node.js 的运行平台。在 *nix（Unix，Linux，OSX）系统上，`child_process.execFile()` 方法更加高效，这是因为它在执行时无需生成一个 shell。但是在 Windows 平台上，如果脱离了终端，`.bat` 和 `.cmd` 文件是无法执行的，因此也无法使用 `child_process.execFile()` 方法。要想在 Windows 环境下调用 `.bat` 和 `.cmd` 文件，有两种方式，一种是使用 `child_process.exec()`，另一种是使用 `child_process.spawn()` 生成一个 `cmd.exe`，然后输入 `.bat` 或者 `.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);
});
```

child_process.exec(command[, options][, callback])

- `command`，字符串，待执行的命令，多个命令之间以空格分隔
- `options`，对象
  - `cwd`，字符串，子进程当前的工作目录（current working directory）
  - `env`，对象，环境变量
  - `encoding`，字符串，默认值为 `utf8`
  - `shell`，字符串，用于执行命令的 shell，在 UNIX 上默认值为 "/bin/sh"，在 Windows 默认值为 "cmd.exe"，该 shell 必须能够解析 UNIX 中的 `-c` 和 Windows 中的 `/s /c` 开关。在 Windows 环境中，命令行解析机制需要兼容 `cmd.exe`。
  - `timeout`，数值，默认值为 0

    - `maxBuffer`，数值，指定 stdout 和 stderr 的最大数据量，如果超过了最大值，则会关闭子进程，默认值为 200 * 1024
    - `killSignal`，字符串，默认值为 "SIGTERM"
    - `uid`，数值，设置进程的用户标识
    - `gid`，数值，设置进程的组标识
- `callback`，回调函数，进程结束时调用，携带进程的输出数据
    - `error`，Error
    - `stdout`，Buffer 实例
    - `stderr`，Buffer 实例
- 返回值类型：ChildProcess 实例

在 shell 中执行 `cammand` 命令，缓存输出信息：

```
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}`);
    }
});
```

如果传入可选的回调函数 `callback`，则该回调函数接收三个来自系统的参数 `(error, stdout, stderr)`。如果执行成功，则 `error` 为 `null`，否则，`error` 为 Error 的实例。`error.code` 属性被设置为子进程的退出码，`error.signal` 属性被设置为进程结束的信号名。所有 `error.code` 不为 `0` 的执行结果都被视为程序执行过程中存在错误。

可选参数 `options` 是 `child_process.exec()` 的第二个参数，用于自定义子进程的生成方式，下面代码是该参数的默认值：

```
{
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
}
```

如果 `timeout` 的值大于 0 且子线程运行时间超过 `timeout`，则父进程会通过 `killSignal` 属性向子进程发送信号。

`maxBuffer` 参数指定了 stdout 和 stderr 的最大数据量（以字节为单位），如果超过最大值，则会关闭子进程。

> 值得注意的是，`child_process.exec()` 与 POSIX 系统调用的 `exec()` 是不同的，它不会替换现有的进程，而是会使用 shell 去执行命令。

child_process.execFile(file[, args][, options][, callback])

- `file`，字符串，可执行文件的文件名或者路径名
- `args`，字符串形式的参数数组
- `options`，对象
  - `cwd`，字符串，子进程当前的工作目录（current working directory）
  - `env`，对象，环境变量
  - `encoding`，字符串，默认值为 `utf8`
  - `shell`，字符串，用于执行命令的 shell，在 UNIX 上默认值为 "/bin/sh"，在 Windows 默认值为 "cmd.exe"，该 shell 必须能够解析 UNIX 中的 `-c` 和 Windows 中的 `/s /c` 开关。在 Windows 环境中，命令行解析机制需要兼容 `cmd.exe`。
  - `timeout`，数值，默认值为 0
  - `maxBuffer`，数值，指定 stdout 和 stderr 的最大数据量，如果超过了最大值，则会关闭子进程，默认值为 200 * 1024
  - `killSignal`，字符串，默认值为 "SIGTERM"
  - `uid`，数值，设置进程的用户标识
  - `gid`，数值，设置进程的组标识
- `callback`，回调函数，进程结束时调用，携带进程的输出数据

    - `error`，Error
    - `stdout`，Buffer 实例
    - `stderr`，Buffer 实例
- 返回值类型：ChildProcess 实例

`child_process.execFile()` 与 `child_process.exec()` 功能相似，唯一的不同在于不会生成新的 shell。更准确地来说，系统会使用新进程解析可执行文件，相比 `child_process.exec()` 更加高效。

`child_process.execFile()` 的可选参数与 `child_process.exec()` 相同。此外，由于没有生成新的 shell，所以不支持类似 I/O 重定向和文件名匹配的功能：

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (error, stdout, st
derr) => {
  if (error) {
    throw error;
  }
  console.log(stdout);
});
```

child_process.fork(modulePath[, args][, options])

- `modulePath`，字符串，在子进程中执行的模块名
- `args`，字符串形式的参数数组
- `options`，对象
    - `cwd`，字符串，子进程当前的工作目录（current working directory）
    - `env`，对象，环境变量
    - `execPath`，字符串，可执行文件的路径，用于创建子进程
    - `execArgv`，字符串形式的参数数组，传递给可执行文件，默认值为 `process.execArgv`
    - `silent`，布尔值，如果为 `true`，则将 `stdin` / `stdout` 和 `stderr` 的数据输送到父进程，否则从父进程接收数据，更多信息请参考 `child_process.spawn()` 方法中 `stdio` 的 `pipe` 和 `inherit` 参数
    - `uid`，数值，设置进程的用户标识
    - `gid`，数值，设置进程的组标识

- 返回值类型：ChildProcess 实例

`child_process.fork()` 方法是 `child_procees.spawn()` 方法的特殊用例，该方法同样返回一个 `ChildProcess` 实例，但是该实例中内建一个通信信道，用于在父子进程之间传递数据，更多信息请参考 `ChildProcess#send()` 方法。

有一点需要牢记，那就是除 IPC 信道之外，父子进程之间是相互独立的，IPC 信道则存在在父子两者之中。每一个进程都有自己的内存空间，都是独立的 V8 实例。因为创建新的进程需要分配新的空间，所以不建议创建大量的子进程。

默认情况下，`child_process.fork()` 使用父进程的 `process.execPath` 属性创建新的 Node.js 实例。可选参数对象 `options` 中的 `execPath` 属性允许使用自定义的可执行文件路径。

使用自定义 `execPath` 加载的 Node.js 进程会使用在子进程中使用环境变量 `NODE_CHANNEL_FD` 这一文件描述符与父进程进行通信。该文件描述符的输入输出要求是以行界定、冒号分隔的 JSON 对象。

> 与 POSIX 系统调用的 `fork()` 方法有所不同，`child_process.fork()` 方法不会拷贝当前进程。

child_process.spawn(command[, args][, options])

- `command`，字符串，待执行的命令，多个命令之间以空格分隔
- `args`，字符串形式的参数数组
- `options`，对象
  - `cwd`，字符串，子进程当前的工作目录（current working directory）
  - `env`，对象，环境变量
  - `stdio`，数组或字符串，用于配置子进程的 stdio（参考 options.stdio）
  - `detached`，布尔值，是否为子进程独立于父进程之外做准备，实际表现依赖于系统开发平台（参考 options.detached)
  - `uid`，数值，设置进程的用户标识
  - `gid`，数值，设置进程的组标识
  - `shell`，字符串或布尔值，默认值为 false。如果值为 `true`，则在 shell 之中执行 `cammand`。该 shell 在 UNIX 上默认值为 "/bin/sh"，在 Windows 默认值为 "cmd.exe"。自定义的 shell 需要传入字符串形式的路径，且自定义的 shell 必须能够解析 UNIX 中的 `-c` 和 Windows 中的 `/s /c` 开关。
- 返回值类型：ChildProcess 实例

`child_process.spawn()` 方法根据传入的 `cammand` 和 `args` 参数生成新的进程。如果没有传入可选参数 `args`，则默认将 `args` 赋值为空数组 `[]`。

第三个参数可以指定其他可选参数，缺失则使用默认值：

```
{
  cwd: undefined,
  env: process.env
}
```

`cwd` 指定新进程的活动路径，如果为空，则使用使用当前目录。`env` 指定环境变量，可被生成的新进程调用，默认值为 `process.env`。

下面代码演示了使用 `child_process.spawn()` 方法执行 `ls -lh /usr` 命令，并捕获 `stdout` / `stderr` 以及退出码的操作：

```
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}`);
});
```

下面代码更加细致地演示了如何执行 `ps ax | grep ssh`：

```
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}`);
  }
});
```

下面代码演示了如何监听错误事件：

```
const spawn = require('child_process').spawn;
const child = spawn('bad_command');

child.on('error', (err) => {
  console.log('Failed to start child process.');
});
```

options.detached

在 Windows 开发环境下，如果将 `options.detached` 设为 `true`，则即使父进程结束之后，子进程仍可运行并拥有自己的控制台。一旦为子进程设置为 `true`，则无法取消。

在非 Windows 开发环境下，如果将 `options.detached` 设为 `true`，则该子进程将会成为新建子进程组以及会话的主管进程（leader）。注意在这种情况下，无论父子进程是否分离，父进程结束之后，子进程都会继续执行，更多信息请参考 `setsid(2)`。

默认情况下，父进程会等待已分离的子进程结束后再结束进程。如果要取消父进程等待子进程的机制，可以使用 `child.unref()` 方法，使用该方法将会让父进程的事件循环机制忽略该子进程的存在，不计入父进程的引用计数之中，实现独立结束父进程，无需检测子进程，但有一种情况例外，那就是父子进程之间存在 IPC 信道。

下面代码演示了如何分离长时间运行的进程以及如何重定向输出：

```
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();
```

使用 `detached` 选项配置长时间运行的进程时，需要注意的一点，除非提供一个不与父进程进行连接的 `stdio`，否则子进程不会再后台持续运行。如果子进程继承了父进程的 `stdio`，则子进程可以继续与控制终端进行交互。

options.stdio

`options.stdio` 参数常用来配置父子进程之间的通信管道。默认情况下，子进程的 stdin / stdout / stderr 都会被重定向到相应的 `ChildProcess` 对象的 `child.stdin` / `child.stdout` / `child.stderr` 数据流中，等同于设置 `options.stdio` 的值为 `['pipe', 'pipe', 'pipe']`。

为了便于理解，下面列出 `options.stdio` 所有的可选值：

- `'pipe'`，等同于 `['pipe', 'pipe', 'pipe']`，该值为 `options.stdio` 的默认值
- `'ignore'`，等同于 `['ignore', 'ignore', 'ignore']`
- `'inherit'`，等同于 `[process.stdin, process.stdout, process.stderr] or [0,1,2]`

`options.stdio` 的值为一个数组，数组的索引与子进程的文件描述符相对应，0 / 1 / 2 分别对应 stdin / stdout / stderr。此外，可以指定文件描述，在父子进程之间创建额外的管道，可选值为以下几种：

1. `pipe`，在父子进程之间创建一个管道。父进程的管道末端可以通过 `child_process` 对象的一个属性得到，比如 `ChildProcess.stdio[fd]`。为文件描述符指定的 0~2 管道也可以通过 ChildProcess.stdin / ChildProcess.stdout 和 ChildProcess.stderr 访问。
2. `ipc`，在父子进程之间创建 IPC 信道，用于传输消息和文件描述符。ChildProcess 实例最多只允许有一个 IPC stdio 文件描述符。使用该值将允许 ChildProcess 实例使用 `send()` 方法。如果子进程向文件描述符写入 JSON 数据，则会在父进程触发 `ChildProcess.on('message')` 事件。如果子进程是一个 Node.js 进程，则在子进程中允许使用 `process.send()` / `process.disconnect()` / `process.on('disconnect')` 和 `process.on('message')` 方法。
3. `ignore`，声明在子进程中忽略文件描述符。Node.js 默认为新建进程开启 0~2 文件描述符，如果将文件描述符设置为 `ignore`，则 Node.js 会将 `/dev/null` 附加给文件描述符。
4. `Stream` 对象，与子进程共享一个可读写的 tty / file / socket / pipe 数据流。

数据流的底层文件描述符会被子进程复制给与 `stdio` 数组相对应的文件描述符。注意，数据流必须拥有一个底层描述符。

5. 正整数，该值表示在父进程中打开的文件描述符。父子进程共享该值，类似于 `Stream` 对象的共享机制。
6. `null` / `undefined`，使用默认值。为 stdio 的文件描述符 0~2 创建管道，为文件描述符 3 及以上设为 `ignore`

```
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，如果父子进程之间创建了 IPC 信道，且子进程是一个 Node.js 进程，则只有子进程为 `process.on('disconnected')` 注册了事件处理器之后，才会启动开启 IPC 信道（该信道通过 `unref()` 方法，不被父进程引用）的子进程。这一机制允许子进程正常退出，而无需进程通过 IPC 信道保持开启状态。

创建同步进程

`child_process.spawnSync()`、`child_process.execSync()` 和 `child_process.execFileSync()` 都是同步方法，它们在执行过程中会阻塞 Node.js 的事件循环机制、暂停其他代码的执行，直到进程结束。

这些同步执行的方法在某些方面大有用处，比如简化常规的脚本任务、简化应用程序配置的加载和执行过程。

child_process.execFileSync(file[, args][, options])

- `file`，字符串，可执行文件的文件名或者路径名
- `args`，字符串形式的参数数组

- `options`，对象
  - `cwd`，字符串，子进程当前的工作目录（current working directory）
  - `input`，字符串和 Buffer 实例，该值将会作为 stdin 传入新建进程。传入该值会覆盖 `stdio[0]` 的值
  - `stdio`，数组，用于配置子进程的 stdio，默认值为 "pipe"。除非指定 `stdio` 否则默认会将 `stderr` 传输给父进程的 `stderr`
  - `env`，对象，环境变量
  - `uid`，数值，设置进程的用户标识
  - `gid`，数值，设置进程的组标识
  - `timeout`，数值，允许子进程执行的最长时间，单位为毫秒，默认值为 undefined
  - `maxBuffer`，数值，指定 stdout 和 stderr 的最大数据量，如果超过了最大值，则会关闭子进程，默认值为 200 * 1024
  - `killSignal`，字符串，新建子进程结束时使用到的信号值，默认值为 "SIGTERM"
  - `encoding`，字符串，指定所有 stdio 输入输出的编码格式，默认值为 `buffer`
- 返回值类型：Buffer 实例或字符串

`child_process.execFileSync()` 方法与 `child_process.execFile()` 方法非常相似，最大的差别在于只有子进程完全结束，该方法才会然会值。如果子进程执行时间超过了 `tiemout` 或者收到了 `killSignal`，则直到进程完全退出前，该方法都不会返回值。注意，如果子进程拦截处理了 `SIGTERM` 信号，并且进程没有结束，那么父进程就会一直等待子进程结束。

如果进程执行时间超市，或者退出码不为 0，那么该方法就睡抛出错误，抛出的 Error 实例包含来自 `child_process.spawnSync()` 方法的完整的错误信息。

child_process.execSync(command[, options])

- `command`，字符串，待执行的命令，多个命令之间以空格分隔
- `options`，对象
  - `cwd`，字符串，子进程当前的工作目录（current working directory）
  - `input`，字符串和 Buffer 实例，该值将会作为 stdin 传入新建进程。传入该值会覆盖 `stdio[0]` 的值
  - `stdio`，数组，用于配置子进程的 stdio，默认值为 "pipe"。除非指定 `stdio` 否则默认会将 `stderr` 传输给父进程的 `stderr`

    - `env`，对象，环境变量
    - `shell`，字符串，用于执行命令的 shell，在 UNIX 上默认值为 "/bin/sh"，在 Windows 默认值为 "cmd.exe"，该 shell 必须能够解析 UNIX 中的 `-c` 和 Windows 中的 `/s /c` 开关。在 Windows 环境中，命令行解析机制需要兼容 `cmd.exe`。
    - `uid`，数值，设置进程的用户标识
    - `gid`，数值，设置进程的组标识
    - `timeout`，数值，允许子进程执行的最长时间，单位为毫秒，默认值为 undefined
    - `killSignal`，字符串，新建子进程结束时使用到的信号值，默认值为 "SIGTERM"
    - `maxBuffer`，数值，指定 stdout 和 stderr 的最大数据量，如果超过了最大值，则会关闭子进程，默认值为 200 * 1024
    - `encoding`，字符串，指定所有 stdio 输入输出的编码格式，默认值为 `buffer`
- 返回值类型：Buffer 实例或字符串

`child_process.execSync()` 方法与 `child_process.exec()` 方法非常相似，最大的差别在于只有子进程完全结束，该方法才会然会值。如果子进程执行时间超过了 `tiemout` 或者收到了 `killSignal`，则直到进程完全退出前，该方法都不会返回值。注意，如果子进程拦截处理了 `SIGTERM` 信号，并且进程没有结束，那么父进程就会一直等待子进程结束。

如果进程执行时间超市，或者退出码不为 0，那么该方法就睡抛出错误，抛出的 Error 实例包含来自 `child_process.spawnSync()` 方法的完整的错误信息。

child_process.spawnSync(command[, args][, options])

- `command`，字符串，待执行的命令，多个命令之间以空格分隔
- `args`，字符串形式的参数数组
- `options`，对象
    - `cwd`，字符串，子进程当前的工作目录（current working directory）
    - `input`，字符串和 Buffer 实例，该值将会作为 stdin 传入新建进程。传入该值会覆盖 `stdio[0]` 的值
    - `stdio`，数组，用于配置子进程的 stdio，默认值为 "pipe"。除非指定 `stdio` 否则默认会将 `stderr` 传输给父进程的 `stderr`
    - `env`，对象，环境变量

    - `uid`，数值，设置进程的用户标识
    - `gid`，数值，设置进程的组标识
    - `timeout`，数值，允许子进程执行的最长时间，单位为毫秒，默认值为 undefined
    - `killSignal`，字符串，新建子进程结束时使用到的信号值，默认值为 "SIGTERM"
    - `maxBuffer`，数值，指定 stdout 和 stderr 的最大数据量，如果超过了最大值，则会关闭子进程，默认值为 200 * 1024
    - `encoding`，字符串，指定所有 stdio 输入输出的编码格式，默认值为 `buffer`
    - `shell`，字符串或布尔值，默认值为 false。如果值为 `true`，则在 shell 之中执行 `cammand`。该 shell 在 UNIX 上默认值为 "/bin/sh"，在 Windows 默认值为 "cmd.exe"。自定义的 shell 需要传入字符串形式的路径，且自定义的 shell 必须能够解析 UNIX 中的 `-c` 和 Windows 中的 `/s /c` 开关。
- 返回值类型：对象
    - `pid`，数值，子进程的 PID（进程标识符）
    - `output`，数组，来自 stdio 的输出
    - `stdout`，Buffer 实例或字符串，`output[1]` 的内容
    - `stderr`，Buffer 实例或字符串，`output[2]` 的内容
    - `status`，数值， 子进程的退出码
    - `signal`，字符串， 用于杀死子进程的信号
    - `error`，Error 实例，如果子进程失败或超市返回此对象

`child_process.execSync()` 方法与 `child_process.exec()` 方法非常相似，最大的差别在于只有子进程完全结束，该方法才会然会值。如果子进程执行时间超过了 `tiemout` 或者收到了 `killSignal`，则直到进程完全退出前，该方法都不会返回值。注意，如果子进程拦截处理了 `SIGTERM` 信号，并且进程没有结束，那么父进程就会一直等待子进程结束。

CLass: ChildProcess

`ChildProcess` 类的实例是 EventEmitter，相当于创建了新的子进程。`ChildProcess` 实例需要使用 `child_process.spawn()` / `child_process.exec()` / `child_process.execFile()` 或者 `child_process.fork()` 方法创建。

事件：'close'

- `code`，数值，子进程正常结束时的退出码
- `signal`，字符串，子进程中断时的信号

`close` 事件发生在子进程 stdio stream 关闭时，这个 `exit` 事件有所不同，因为同一个 stdio stream 可能被多个进程共享。

事件：'disconnect'

`disconnect` 事件发生在父进程或子进程调用 `ChildProcess.discount()` 方法之后。断开连接之后，将不再收发消息，`ChildProcess.connected` 属性被赋值为 false。

事件：'error'

- `err`，Error 实例

`error` 事件发生的原因有以下几种：

1. 进程创建失败
2. 进程无法杀死
3. 无法向子进程发送消息

注意，`error` 事件发生后，`exit` 事件有可能发生也有可能不发生。如果你同时监听了 `exit` 和 `error` 事件，一定要预防事件处理函数被多次调用的情况。

事件：'exit'

- `code`，数值，子进程正常结束时的退出码
- `signal`，字符串，子进程中断时的信号

`exit` 事件发生在子进程结束之后。进程结束之后，`code` 表示进程正常结束的退出码，否则为 `null`。如果进程因为接收到信号而中断，那么就会使用 `signal` 表示该信号的名称。两者之一必不为 `null`。

注意，触发 `exit` 事件时，子进程的 stdio stream 仍可能处于开放状态。此外，Node.js 回味 `SIGINT` 和 `SIGTERM` 创建信号处理器，所以为了接收这些信号，Node.js 进程并不会立即结束。更深的理解是，Node.js 会执行一系列的清理行为，然后再唤起信号。

事件：'message'

- `message`，对象或原始值，一个解析后的 JSON 对象或原始值
- `sendHandle`，Handle 实例，一个 `net.Socket` 或 `net.Server` 对象，甚至是 undefined

子进程使用 `process.send()` 发送消息时触发 `message` 事件。

child.connected

- 布尔值，`disconnect()` 方法调用后，该值为 false

`child.connected` 属性用于声明是否继续与子进程接发消息，如果值为 `false`，则停止接发消息。

child.disconnect()

该方法用于关闭父子进程之间的 IPC 信道，如果父子进程之间不再通信了，就可以优雅的结束子进程。调用该方法后，父子进程中的 `child.connected` 和 `process.connected` 属性会被设为 false，并且不再进行通信。

如果进程不再收到消息，则触发 `disconnect` 事件。当调用 `child.disconnect()` 方法后，会直接触发 `disconnect` 事件。

注意，如果子进程是 Node.js 实例（比如使用 `child_process.fork()` 创建的进程），那么子进程中的 `process.discounnect()` 方法也会被调用，继而关闭 IPC 信道。

child.kill([signal])

- `signal`，字符串

`child.kill()` 方法用于给子进程发送信号。如果参数为空，则进程发送 `SIGTERM` 信号，完整的信号列表请参考 `signal(7)`。

```
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');
```

如果信号无法传递，则 `ChildProcess` 对象触发 `error` 事件。向已经结束的子进程发送信号并不会触发 `error` 事件，但是可能会发生意想不到的结果。特别值得强调的是，如果 PID 已经被其他进程注册了，那么信号就会被传递给注册了该 PID 的进程，继而导致无法预测的结果。

注意，虽然这个方法的函数名是 `kill`，但传递给子进程的信号有可能无法终端该进程。

child.pid

- 整数

返回子进程的 PID（进程标示符）。

```
const spawn = require('child_process').spawn;
const grep = spawn('grep', ['ssh']);

console.log(`Spawned child pid: ${grep.pid}`);
grep.stdin.end();
```

child.send(message[, sendHandle][, callback])

- `message`，对象
- `sendHandle`，Handle 实例
- `callback`，回调函数
- `Return`，布尔值

父子进程之间创建 IPC 信道之后，就可以使用 `child.send()` 方法向子进程发送消息了。如果子进程是一个 Node.js 实例，那么可以使用 `process.on('message')` 事件来接收消息。下面代码实例是父进程部分的脚本：

```
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' });
```

下面代码实例是子进程部分的脚本：

```
process.on('message', (m) => {
  console.log('CHILD got message:', m);
});

process.send({ foo: 'bar' });
```

子进程同样可以使用自己的 `process.send()` 方法想父进程发送消息。

类似 `{cmd: 'NODE_foo'}` 的消息属于 Node.js 消息中的特例。所有在 `cmd` 字段中以 `NODE_` 为前缀的消息，都会被 Node.js core 储存起来，不会触发子进程中的 `process.on('message')` 事件。更进一步，如果想要捕获此类消息需要使用 `process.on('internalMessage')` 事件。应用程序应该避免传输此类消息，或者使用 `internalMessage` 事件来监听。

`child.sned()` 方法中的可选参数 `sendHandle` 用于向子进程传递 TCP 服务器或者 socket 对象。子进程会将收到的对象作为第二个参数传递给 `process.on('message')` 事件中注册的回调函数。

可选参数 `callback` 是一个回调函数，会在消息发送之后、子进程接收消息之前被调用。该回调函数只有一个参数，当 `child.send()` 执行成功时，该参数的值为 null，反之，则为 Error 实例。

如果没有传入回调函数且消息发送失败，`ChildProcess` 对象就会触发一个 `error` 事件，比如发送消息时子进程已经结束。

如果信道关闭或者积压的消息超过了阈值，则 `child.send()` 就会返回 `false`，反之，则返回 `true`。其中，`callback` 函数可以用来实现流程控制。

如何发送一个 **server** 对象

在下面的代码中，使用 `sendHandle` 参数想子进程传递了一个 TSCP 服务器的处理器：

```
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);
});
```

子进程接收 server 对象：

```
process.on('message', (m, server) => {
  if (m === 'server') {
    server.on('connection', (socket) => {
      socket.end('handled by child');
    });
  }
});
```

现在父子进程之间共享 server，那么两者都可以调用 server 处理任务。

虽然上面使用的 server 使用 `net` 模块创建的，实际上 `dgram` 模块的 server 使用方式也大同小异，差异主要包括：使用 `message` 事件而不是 `connection` 事件进行监听，使用 `server.bind` 而不是 `server.listen` 方法进行绑定。目前，`dgram` 模块的 server 只支持 UNIX 平台。

如何发送一个 **socket** 对象

同样，`sendHandler` 参数也可以用来向子进程发送 socket 对象。下面的代码新建了两个子进程，其中一个的 `priority` 参数为 `normal`，一个为 `special`：

```
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);
```

在子进程中接收 socket 对象，并将其作为第二个参数传递给回调函数：

```
process.on('message', (m, socket) => {
  if (m === 'socket') {
    socket.end(`Request handled with ${process.argv[2]} priority`);
  }
});
```

父进程将 socket 对象传递给子进程之后，如果 socket 对象销毁了，则父进程将不再追踪它，且 `.connections` 属性变为 `null`。当出现这种情况时，不建议使用 `.maxConnections` 属性。

child.stderr

- Stream 实例

子进程 stderr 的可读 Stream 实例。如果生成子进程的 `stdio[2]` 不是 `pipe`，则该值为 undefined。

`child.stderr` 是 `child.stdio[2]` 的别名，它们指向同一个值。

child.stdin

- Stream 实例

子进程 stdin 的可写 Stream 实例。注意，如果一个子进程在等待输入，它会暂停执行，直到 stream 通过 `end()` 被关闭。如果生成子进程的 `stdio[0]` 不是 `pipe`，则该值为 undefined。

`child.stdio` 是 `child.stdio[0]` 的别名，它们指向同一个值。

child.stdio

- 数组

传给子进程的管道数组，与 `child_process.spawn()` 方法中可选参数 `stdio` 的顺序相一致，默认值为 'pipe'。注意，`child.stdio[0]` / `child.stdio[1]` / `child.stdio[2]` 与 `child.stdin` / `child.stdout` / `child.stderr` 对应相等。

在下面的代码中，只有子进程的文件描述符 1 (`stdout`) 的值为 `pipe`，所有只有父进程的 `child.stdio[1]` 是 Stream 实例，其他都为 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.stdout

子进程 stdout 的可读 Stream 实例。如果生成子进程的 `stdio[1]` 不是 `pipe`，则该值为 undefined。

`child.stdout` 是 `child.stdio[1]` 的别名，它们指向同一个值。

集群

接口稳定性: 2 - 稳定

众所周知，Node.js 的实例默认是在单线程中执行的。为了充分利用多核系统的性能，开发者可能会希望创建一个 Node.js 的进程集群处理各类负载。

开发者可以利用 `cluster` 模块快速创建共享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);
}
```

在 Node.js 中运行上面的代码，创建共享 8000 端口的 workers:

```
$ 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 系统中，尚无法在 worker 中创建一个已命名的管道服务器。

工作方式

worker 进程是由 `child_process.fork()` 方法创建的，所以可以通过 IPC 在主进程和子进程之间传递服务器句柄。

`cluster` 模块提供了两种分发连接的方式。

第一种方式，也是默认方式，该方式不适用于 windows 平台，该方式通过轮询（Round-Robin）的方式分发连接，具体过程是：主进程监听端口，接收到新连接之后，通过轮询方式分发给可用的 worker，并通过内建的算法避免某个 worker 进程负载过重。

第二种方式是由主进程创建监听 socket 并将其分发给对合适的 workder，然后当连接进来时，由相应的 worker 直接处理。

理论上来说，第二种方式更加高效，但实际上，由于操作系统调度策略的复杂性，所以往往导致连接分发不平衡的现象，比如 70% 的连接终止于八个进程中的两个。

因为这里用到了 `server.listen()` 方法将大部分的工作移交给主线程，所以常规的 Node.js 进程和集群 worker 之间存在三个区别：

1. `server.listen({fd: 7})`，该消息发往主线程，主线程会监听文件描述 7 并将句柄传递给 worker，而不是监听 worker 对文件描述 7 的处理。
2. `server.listen(handle)`，监听特定的句柄，指定 worker 使用的句柄，而不是通知主线程使用哪一个句柄。如果 worker 中已存在句柄，那么它会认为你对正在进行的处理了如指掌。
3. `server.listen(0)`，通常来说，这会让服务器监听一个随机端口，不过在集群中，当 worker 调用 `listen(0)` 时，所有的 worker 都会收到相同的随机端口。本质上来说，这个端口只有第一次是随机的，随后的端口都是可预测

的。如果你想创建一个独一无二的端口，可以根据集群的 worker ID 创建一个端口号。

在 Node.js 中并没有路由组件，workers 之间也米有共享的状态，因此，开发时应注意类似 session 和登录这样的任务不要过度依赖存储于内存的数据对象。

因为所有的 workers 都是独立的进程，所以你可以根据开发需要安全地杀死或重建 worker，某个进程的修改并不会影响其他进程。只要有 worker 还处于活跃状态，那么服务器就会持续接收到连接。当所有 workers 都结束后，既有连接将会被抛弃，新的连接请求也将会被拒绝。Node.js 不会为开发者自动管理 worker 的数量，所以开发者应该根据应用程序的需要自主管理 worker 池。

Class: Worker

一个 Worker 对象包含了有关 worker 的所有公有信息和方法。在主进程中，可以通过 `cluster.workers` 获取有关 worker 的信息，在 worker 进程中可以通过 `cluster.worker`。

事件：'disconnect'

该事件与 `cluster.on('disconnect')` 事件类似，但针对于特定的 worker:

```
cluster.fork().on('disconnect', () => {
    // Worker has disconnected
});
```

事件：'error'

该事件与 `child_process.fork()` 所提供的同名事件相同。

在 worker 中开发者也可以使用 `process.on('error')`。

事件：'exit'

- `code`，数值，进程正常结束的退出码
- `signal`，字符串，进程中断时的信号（比如 `'SIGHUP'`）

该事件类似于 `cluster.on('exit')` 事件，但针对于特定的 worker:

```
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!');
  }
});
```

事件：'listening'

- `address`，对象

该事件类似于 `cluster.on('listening')` 事件，但针对特定的 worker:

```
cluster.fork().on('listening', (address) => {
  // Worker is listening
});
```

worker 进程无法触发该事件。

事件：'message'

- `message`，对象

该事件类似于 `cluster.on('message')` 事件，但针对特定的 worker。

该事件与 `child_process.fork()` 提供的同名事件相同。

在 worker 中开发者也可以使用 `process.on('worker')`。

在下面的代码中，演示了如何使用消息系统在集群的主进程中记录请求量：

```
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 notifyRequ
est
  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);
}
```

事件：'online'

该事件类似于 `cluster.on('online')` 事件，但针对于特定的 worker:

```
cluster.fork().on('online', () => {
  // Worker is online
});
```

worker 进程无法触发该事件。

worker.disconnect()

在一个 worker 中，调用该函数会结束所有的服务器，然后等待这些服务器的 `close` 事件，最后切断所有的 IPC 信道。

在主进程中，会发送给调用该方法的 worker 进程一条内部消息，用于调用自身的 `disconnect()` 方法。

调用该方法，会设置 `.suicide` 。

注意，服务器被关闭之后，将不会再接收新的连接，但可以使用其他监听 worker 接收这些连接，现有连接可以通过正常的方式关闭。如果没有连接，参考 `server.close()` ，相应的 IPC 信道会被关闭。

以上所述只适用于服务器连接，客户端连接不能被 workers 关闭，而且切断连接时不需要等待客户端结束进程。

注意在一个 worker 中存在 `process.disconnect` ，它不是一个函数，更多信息请参考 `child.disconnect()` 。

服务器的长连接有可能阻塞 workers 进程，一种解决方式是给相关的应用发送消息，这样应用可以主动关闭这些连接。另一种有效的方式是创建一个 timeout 延时方法，如果 worker 在指定时间内没有触发 `disconnect` 事件，则主动关闭它：

```
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.id

- 数值

每一个新建的 worker 都会获得一个独一无二的 id，这个 id 就存储在 `worker.id` 属性之中。

当 worker 正常运行时，该属性也是 `cluster.workers` 的键名之一。

worker.isConnected()

如果 worker 和主进程通过 IPC 正常连接，则该方法返回 true，否则返回 false。worker 创建后会自动与主线程通过 IPC 信道连接，当 `disconnect` 事件触发后，则与主线程断开连接。

worker.isDead()

如果 worker 进程结束了，则该方法返回 true，否则返回 false。

worker.kill([signal='SIGTERM'])

- `signal`，字符串，发送给 worker 进程的扼杀信号

该方法用于结束 worker 进程。在主线程中，它通过关闭 `worker.process` 杀死 worker 进程，进程关闭之后，发送扼杀信号。在 worker 进程中，它通过关闭信道杀死 worker，然后通过退出码 0 退出。

调用该方法，会设置 `.suicide`。

为了向后保持兼容性，该方法是 `worker.destroy()` 的同名函数。

注意，在 worker 进程中存在 `process.kill()`，它不是一个函数，跟多信息请参考 `process.kill(pid[, signal])`。

worker.process

- ChildProcess 实例

所有 worker 进程都是由 `child_process.fork()` 方法创建的，该方法返回的对象被存储为 `.process`。In a worker, the global `process` is stored.

注意，如果 `process` 触发了 `disconnect` 事件且 `.suicide` 的值不为 `true`，则所有的 worker 进程都会调用 `process.exit(0)`，这有助于避免连接的意外性失连。

worker.send(message[, sendHandle][, callback])

- `message`，对象
- `sendHandle`，Handle 实例
- `callback`，函数

- 返回值类型：Boolean

该方法用于向主进程或 worker 进程发送消息，发送时可携带一个句柄。

在主进程中使用该方法向指定 worker 发送消息的作用与 `ChildProcess.send()` 方法相同。

在 worker 进程中使用该方法向主进程发送消息的作用与 `process.send()` 方法相同。

下面的示例会从 worker 进程向主进程回传消息：

```
if (cluster.isMaster) {
  var worker = cluster.fork();
  worker.send('hi there');

} else if (cluster.isWorker) {
  process.on('message', (msg) => {
    process.send(msg);
  });
}
```

worker.suicide

- 布尔值

初始值为 undefined，调用 `.kill()` 和 `.disconnect()` 时会将该值重置为布尔类型。

布尔值 `worker.suicide` 用于帮助开发者辨析进程是主动结束还是意外结束，便于在主线程中根据该值决定是否重建 worker 线程。

```
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();
```

事件：'disconnect'

- `worker`，cluster.Worker 实例

当 worker 的 IPC 信道关闭之后会触发该事件。worker 进程正常退出、被杀死或被手动中断的情况下都会触发该事件。

在 `disconnect` 事件和 `exit` 事件之间可能存在时间延迟。这些事件可以用于检测一个进程是否处于清理状态或者进程是否存在长连接。

```
cluster.on('disconnect', (worker) => {
  console.log(`The worker #${worker.id} has disconnected`);
});
```

事件：'exit'

- `worker`，cluster.Worker 实例
- `code`，数值，进程正常结束时的退出码
- `signal`，字符串，进程中断时的信号（比如 `'SIGHUP'`）

任意 worker 进程结束时，cluster 模块就会触发 `exit` 事件。

该事件可用于重启 worker 进程：

```
cluster.on('exit', (worker, code, signal) => {
  console.log('worker %d died (%s). restarting...',
    worker.process.pid, signal || code);
  cluster.fork();
});
```

更多信息请参考 `child_process event: 'exit'`。

事件：'fork'

- `worker`，cluster.Worker 实例

当一个新的 worker 进程被分立出来时，cluster 模块就会触发 `fork` 事件。该事件可以用记录 worker 的活动日志，创建自定义的 timeout 延时处理函数：

```
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();
});
```

事件：'listening'

- `worker`，cluster.Worker 实例
- `address`，对象

worker 进程调用 `listen()` 方法后，`listening` 事件会同时在服务器和主进程的 `cluster` 触发。

该事件处理器接收两个参数，`worker` 参数是一个 worker 对象，`address` 对象包含多个连接属性：`address`、`port` 和 `addressType`。当 worker 监听多个地址时，该事件非常有用：

```
cluster.on('listening', (worker, address) => {
  console.log(
    `A worker is now connected to ${address.address}:${address.p
ort}`);
});
```

`addressType` 的值为以下之一：

- `4` (TCPv4)
- `6` (TCPv6)
- `-1` (unix domain socket)
- `"udp4"` 或 `"udp6"` (UDP v4 or v6)

事件：'message'

- `worker`，cluster.Worker 实例
- `message`，对象

当 worker 进程接收到消息触发该事件。

更多信息请参考 `child_process event:'message'`。

事件：'online'

- `worker`，cluster.Worker 实例

分立新 worker 进程之后，该 worker 进程应该回应一个 online 消息。当主线程接收到 online 消息之后，就会触发该事件。`fork` 事件和 `online` 事件的差异在于，主分支分立 worker 进程时触发 fork 事件，worker 进程首次运行时触发 `online` 事件。

```
cluster.on('online', (worker) => {
  console.log('Yay, the worker responded after it was forked');
});
```

事件：'setup'

- `settings`，对象

每次调用 `.setupMaster()` 方法时都会触发该事件。

当 `.setupMaster()` 方法被调用时，这里的 `settings` 对象就是 `cluster.settings` 对象，因为对 `.setupMaster()` 的调用都发生在同一个周期内。

如果对精确度要求较高，请使用 `cluster.settings`。

cluster.disconnect([callback])

- `callback`，函数，当所有 workers 都结束和句柄关闭后，调用该回调函数

`cluster.workers` 中的每个 worker 进程都可以调用 `.disconnect()` 方法。

当 worker 继承被中断时，所有的内部句柄都会关闭，如果所有事件都处理完了，那么主进程就会安全退出。

该方法结束后，最终会调用可选的回调函数 `callback`。

该方法只能从主进程调用。

cluster.fork([env])

- `env`，对象，用于添加到 worker 进程环境的键值对
- 返回值类型：cluster.Worker 实例

该方法用于分立一个新的 worker 进程。

该方法只能从主进程调用。

cluster.isMaster

- 布尔值

如果当前进程是主进程，则该值为 true。该值由 `process.env.NODE_UNIQUE_ID` 决定。如果 `process.env.NODE_UNIQUE_ID` 的值为 undefined，则 `isMaster` 的值为 true。

cluster.isWorker

- 布尔值

如果当前进程不是主进程，则值为 true，反之异然。

cluster.schedulingPolicy

调度策略有两种，其中 `cluster.SCHED_RR` 表示的轮询策略，`cluster.SCHED_NONE` 表示由操作系统处理。这是一个全局配置项，一旦创建第一个分支或调用 `cluster.setupMaster()` 之后，该配置项就会生效且被冻结无法修改。

`SCHED_RR` 是除 Windows 以外其他操作系统的默认调度策略。只要 libuv 能够高效地分配 IOCP 句柄且不造成严重的性能损耗，则 Windows 平台也使用该调度策略。

也可以通过环境变量 `NODE_CLUSTER_SCHED_POLICY` 来设置 `cluster.sckedulingPolicy`，有效值为 `rr` 或 `none`。

cluster.settings

- 对象
  - `execArgv`，数组，传递给 Node.js 的可执行字符串数组，默认值为 `process.execArgv`
  - `exec`，字符串，file path to work file，默认值为 `process.argv[1]`
  - `args`，数组，传递给 worker 进程的字符串参数，默认值为 `process.argv.slice(2)`
  - `silent`，布尔值，是否向父进程的 stdio 发送输出信息，默认值为 `false`
  - `uid`，数值，设置进程的用户 ID
  - `gid`，数值，设置进程的组 ID

当调用 `.setupMaster()` 或 `.fork()` 之后，settings 对象就会包含上述配置信息。

因为 `.setupMaster()` 方法只能被调用一次，所以 settings 对象初始化之后就会被冻结。

cluster.setupMaster([settings])

- settings, 对象
  - `exec`，字符串，file path to worker file，默认值为 `process.argv[1]`
  - `args`，数组，传递给 worker 进程的字符串参数，默认值为 `process.argv.slice(2)`
  - `silent`，布尔值，是否向父进程的 stdio 发送输出信息，默认值为 `false`

`setupMaster()` 方法常用于修改猫人的 `fork` 行为。一旦调用该方法，配置信息就会传递给 `cluster.settings`。

注意事项：

```
- 任何配置项的修改都不会影响运行中的 worker 进程，只会影响未来通过 `.fork()` 新建的 worker 进程
- 唯一无法由 `.setupMaster()` 配置的 worker 属性是传递给 `.fork()` 的 `env` 属性
- 默认值只在第一次调用时有效，以后每次调用都使用上一次传递给 `cluster.setupMaster()` 的参数和配置信息
```

```
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
```

该方法只能从主进程调用。

cluster.worker

- 对象

对当前 worker 对象的引用，无法再主进程中使用该属性：

```
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

- 对象

用于存储当前活跃进程的哈希表，键为 `id`。使用该属性便于遍历所有的 worker 进程。该属性只能在主进程中使用。

当 worker 进程退出或与主进程失去连接后，就会从 `cluster.workers` 中删除，这里面的 disconnect 和 exit 是两个独立的事件，无法预测两者的先后顺序。不过，可以明确的是，删除行为肯定发生在 disconnect 和 exit 都发生之后。

```
// 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');
});
```

有时候开发者希望通过信道获取对某个 worker 进程的引用，此时最简单的方法就是使用 worker 进程独一无二的 ID：

```
socket.on('data', (id) => {
  var worker = cluster.workers[id];
});
```

控制台

接口稳定性: 2 - 稳定

`console` 模块提供了一些简单的调试输出函数，与浏览器提供的 consle 方法类似。

该模块主要对外开放了两个组件：

- `Console` 类，该类提供了 `console.log()` / `console.error` 和 `console.warn()` 方法，常用于向 Node.js stream 输出信息。
- 全局的 `console` 实例，用于向 stdout 和 stderr 输出信息。因为该对象是全局的，所以无需使用 `require('console')` 调用。

```
console.log('hello world');
// 输出结果: hello world, to stdout
console.log('hello %s', 'world');
// 输出结果: hello world, to stdout
console.error(new Error('Whoops, something bad happened'));
// 输出结果: [Error: Whoops, something bad happened], to stderr

const name = 'Will Robinson';
console.warn(`Danger ${name}! Danger!`);
// 输出结果: Danger Will Robinson! Danger!, to stderr
```

下面是使用 `Console` 类的示例：

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

myConsole.log('hello world');
// 输出结果: hello world, to out
myConsole.log('hello %s', 'world');
// 输出结果: hello world, to out
myConsole.error(new Error('Whoops, something bad happened'));
// 输出结果: [Error: Whoops, something bad happened], to err

const name = 'Will Robinson';
myConsole.warn(`Danger ${name}! Danger!`);
// 输出结果: Danger Will Robinson! Danger!, to err
```

虽然 `Console` 类的 API 与浏览器中的 `console` 相似，但 Node.js 开发 `Console` 类的目的并不是复制浏览器的 `console` 功能。

异步 VS 同步 console

`console` 函数默认情况下都是异步执行的，除非要将信息输出到文件中。硬盘已经很快了，操作系统通常使用高速缓存进行读写，但仍然不可避免发生进程阻塞的情况。

Class: Console

通过 `require('console').Console` 或 `console.Console` 的方法可以访问 `Console` 类，它常用于创建简单的可配置输出流的日志记录器：

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

new Console(stdout[, stderr])

该方法通过传入一个或两个可写的 stream 实例来创建一个新的 `Console` 实例。`stdout` 是一个可写的 stream 实例，用于打印日志或输出信息。`stderr` 常用于输出提示和错误。如果没有传入 `stderr`，将使用 `stdout` 输出提示和错误：

```
const output = fs.createWriteStream('./stdout.log');
const errorOutput = fs.createWriteStream('./stderr.log');
// custom simple logger
const logger = new Console(output, errorOutput);
// use it like console
var count = 5;
logger.log('count: %d', count);
// in stdout.log: count 5
```

全局的 `console` 是一个特殊的 `Console` 实例，它输出的信息会发送给 `process.stdout` 和 `process.stderr`，等同于：

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

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

该方法是一个简单的断言测试，常用于测试 `value` 是否为真值。如果不是真值，则抛出 `AssertionError`。如果传入了可选参数 `message`，则使用 `util.format()` 格式优化该参数，输出格式化后的字符串：

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

console.dir(obj[, options])

该方法使用 `util.inspect()` 处理 `obj` 参数，并将字符串结果输送给 `stdout`。该方法忽略 `obj` 参数中自定义的 `inspect()` 方法。可选参数对象 `options` 可以用来修改字符串某些部分：

- `showHidden`，如果为 true，则显示对象的不可枚举和 symbol 属性，默认值

为 false

- `depth`，指定 `inspect()` 方法格式化 `object` 时的深度。这对于格式化复杂对象很有用。默认值为 2，如果不限制深度，可以设置为 `null
- `colors`，如果值为 true，则使用 ANSI 颜色码对输出信息加以美化。默认值为 false

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

该方法用于将消息传送给 `stderr`，接受多个参数，第一个参数可以包含占位符，其他参数可以用来替换占位符，实际上所有的参数都传递给了 `util.format()` 函数：

```
const code = 5;
console.error('error #%d', code);
// 输出结果: error #5, to stderr
console.error('error', code);
// 输出结果: error 5, to stderr
```

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

该方法等同于 `console.log()`。

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

该方法用于将信息传送给 `stdout`，接受多个参数，第一个参数可以包含占位符，其他参数可以用来替换占位符，实际上所有的参数都传递给了 `util.format()` 函数：

```
var count = 5;
console.log('count: %d', count);
  // Prints: count: 5, to stdout
console.log('count: ', count);
  // Prints: count: 5, to stdout
```

console.time(label)

该方法用于启动一个计时器，计算某个操作的执行时间，每个计时器都有特定的 `label` 标签。向 `console.timeEnd()` 中传入同一个 `label` 可以停止计时器，并输出有关执行时间的信息到 stdout 中。计时器的精度是亚毫秒级的。

console.timeEnd(label)

该方法用于停止计时器，并输出有关执行时间的信息到 stdout：

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

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

传送有关堆栈跟踪的信息到 stderr 中，首先是一个字符串 `Trace:` ，接下来是使用 `util.format()` 格式化的堆栈跟踪信息：

```
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.warn([data][, ...])

该方法等同于 `console.error()`。

加密

接口稳定性: 2 - 稳定

`crypto` 模块封装了诸多的加密功能，包括 OpenSSL 的哈希、HMAC、加密、解密、签名和验证函数。

需要使用 `require('crypto')` 导入该模块：

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

const secret = 'abcdefg';
const hash = crypto.createHmac('sha256', secret)
                   .update('I love cupcakes')
                   .digest('hex');
console.log(hash);
// 输出结果: c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea
6ea959d6658e
```

Class: Certificate

SPKAC 是由 Netscape 提出的一个证书签发请求机制（Certificate Signing Request mechanism），现在已经正式成为 HTML5 keygen 元素的一部分。

`crypto` 模块提供的 `Certificate` 类就是用于处理 SPKAC 数据，最常见的用处就是处理 HTML5 `<keygen>` 元素生成的数据。Node.js 在内部使用 OpenSSL 所实现的 SPKAC。

new crypto.Certificate()

通过 `new` 操作符或直接调用 `crypto.Certificate()` 方法都可以创建 `Certificate` 类的实例：

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

const cert1 = new crypto.Certificate();
const cert2 = crypto.Certificate();
```

certificate.exportChallenge(spkac)

在 `spkac` 的数据结构中包含一个公钥和一个口令。`certificate.exportChallenge()` 方法会以 Node.js Buffer 的形式返回口令模块。这里的 `spkac` 参数可以字符串，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.exportPublicKey(spkac)

在 `spkac` 的数据结构中包含一个公钥和一个口令。`certificate.exportPublicKey()` 方法会以 Node.js Buffer 的形式返回公钥模块。这里的 `spkac` 参数可以字符串，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.verifySpkac(spkac)

如果 `spkac` 的数据结构是有效地，则该方法返回 `true`，否则返回 `false`。这里的 `spkac` 参数必须是一个 Node.js 的 Buffer 实例：

```
const cert = require('crypto').Certificate();
const spkac = getSpkacSomehow();
console.log(cert.verifySpkac(new Buffer(spkac)));
// Prints true or false
```

Class: Cipher

`Cipher` 类的实例常用于对数据进行加密。使用该类主要有两种方式：

- 包装成可读写的 stream 实例，写入纯数据，读出加密数据
- 使用 `cipher.update()` 和 `cipher.final()` 方法生成加密数据

`crypto.createCipher()` 或 `crypto.createCipheriv()` 两个方法常用于创建 `Cipher` 类的实例。`Cipher` 类的实例不能直接由 `new` 关键字创建。

下面代码演示了如果将一个 `Cipher` 类的实例封装成 stream 实例：

```
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: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b106
16ed84d815504
});

cipher.write('some clear text data');
cipher.end();
```

下面代码演示了如何将 `Cipher` 类的实例封装成管道流（piped stream）：

```
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);
```

下面代码演示了如何使用 `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', 'h
ex');
encrypted += cipher.final('hex');
console.log(encrypted);
// Prints: ca981be48e90867604588e75d04feabb63cc007a8f8ad89b10616
ed84d815504
```

cipher.final([output_encoding])

返回剩余的加密内容。如果 `output_encoding` 参数是 `binary` / `base64` 或 `hex` 中的一个，则返回字符串，否则返回 Buffer 实例。

一旦调用 `cipher.final()` 方法之后，`Cipher` 类的实例就不能再用于加密数据。多次调用 `cipher.final()` 方法将会抛出错误。

cipher.setAAD(buffer)

当使用加密认证模式（目前只支持 GCM）时，`cipher.setAAD()` 方法用于设置附加认证数据（AAD）的参数。

cipher.getAuthTag()

当使用加密认证模式（目前只支持 GCM）时，`cipher.getAuthTag()` 方法会返回一个 Buffer 实例，该实例包含由原始数据计算得来的验证标记（authentication tag）。

`cipher.getAuthTag()` 方法必须在 `cipher.final()` 方法完成加密操作之后再调用。

cipher.setAutoPadding(auto_padding=true)

当使用块加密算法时，`Cipher` 类将会自动为原始数据填充额外的内容，便于切割成指定大小的块。如果开发者希望禁用默认的数据填充行为，可以调用 `cipher.setAutoPadding(false)`。

如果 `auto_padding` 的值为 `false`，那么原始数据的大小必须是加密块的整数倍，否则 `cipher.final()` 方法将会抛出错误。禁用自动填充数据有助于实现非标准的填充，比如使用 `0x0` 而不是 PKCS 进行填充。

`cipher.setAutoPadding()` 方法必须在 `cipher.final()` 方法之后调用。

cipher.update(data[, input_encoding][, output_encoding])

该方法用于通过 `data` 更新加密数据。如果传入了 `input_encoding` 参数，则其值必须为 `utf8`、`ascii` 和 `binary` 中的一个，且 `data` 参数必须为指定编码格式的字符串数据。如果未指定 `input_encoding` 参数，则 `data` 参数必须是 Buffer 实例。如果已知 `data` 参数是 Buffer 实例，则 `input_encoding` 无论是何值都会被忽略。

`out_encoding` 参数制定了加密数据输出的编码格式，值为 `utf8`、`ascii` 和 `binary` 之一。如果指定了 `output_encoding` 参数，则返回与该编码格式相符的字符串；如果没有传入该参数，则返回一个 Buffer 实例。

在 `cipher.final()` 方法调用之前，可以多次调用 `cipher.update()` 方法。如果在 `cipher.final()` 之后调用 `cipher.update()`，则会抛出一个错误。

Class: Decipher

`Decipher` 类的实例常用于对数据进行解密。使用该类主要有两种方式：

- 包装成可读写的 stream 实例，写入纯数据，读出加密数据

- 使用 `decipher.update()` 和 `decipher.final()` 方法生成解密数据

`crypto.createDecipher()` 或 `crypto.createDecipheriv()` 两个方法常用于创建 `Decipher` 类的实例。`Decipher` 类的实例不能直接由 `new` 关键字创建。

下面代码演示了如果将一个 `Decipher` 类的实例封装成 stream 实例：

```
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 = 'ca981be48e90867604588e75d04feabb63cc007a8f8ad89
b10616ed84d815504';
decipher.write(encrypted, 'hex');
decipher.end();
```

下面代码演示了如何将 `Decipher` 类的实例封装成管道流（piped stream)：

```
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);
```

下面代码演示了如何使用 `decipher.update()` 和 `decipher.final()` 方法：

```
const crypto = require('crypto');
const decipher = crypto.createDecipher('aes192', 'a password');

var encrypted = 'ca981be48e90867604588e75d04feabb63cc007a8f8ad89
b10616ed84d815504';
var decrypted = decipher.update(encrypted, 'hex', 'utf8');
decrypted += decipher.final('utf8');
console.log(decrypted);
// Prints: some clear text data
```

decipher.final([output_encoding])

返回剩余的解密内容。如果 `output_encoding` 参数是 `binary` / `base64` 或 `hex` 中的一个，则返回字符串，否则返回 Buffer 实例。

一旦调用 `decipher.final()` 方法之后，`Decipher` 类的实例就不能再用于加密数据。多次调用 `decipher.final()` 方法将会抛出错误。

decipher.setAAD(buffer)

当使用加密认证模式（目前只支持 GCM）时，`decipher.setAAD()` 方法用于设置附加认证数据（AAD）的参数。

decipher.setAuthTag()

当使用加密认证模式（目前只支持 GCM）时，`decipher.setAuthTag()` 方法常用于传递接收到的验证标记（authentication tag）。如果没有提供任何标记，或者加密文本已被篡改，`decipher.final()` 方法将会抛出错误，用于提示开发者当前加密文本未通过验证必须被抛弃。

decipher.setAutoPadding(auto_padding=true)

如果原始数据加密时没有使用标准的块填充，可以调用 `decipher.setAuthPadding(false)` 方法禁用自动填充，从而避免 `decipher.final()` 校验和移除填充数据。

只有原始数据的长度是加密数据块的整数倍，才可以禁用数据的自动填充行为。

`decipher.setAutoPadding()` 方法必须在 `decipher.update()` 方法之前调用。

decipher.update(data[, input_encoding][, output_encoding])

该方法用于通过 `data` 更新解密数据。如果传入了 `input_encoding` 参数，则其值必须为 `utf8` 、 `ascii` 和 `binary` 中的一个，且 `data` 参数必须为指定编码格式的字符串数据。如果未指定 `input_encoding` 参数，则 `data` 参数必须是 Buffer 实例。如果已知 `data` 参数是 Buffer 实例，则 `input_encoding` 无论是何值都会被忽略。

`out_encoding` 参数制定了加密数据输出的编码格式，值为 `utf8` 、 `ascii` 和 `binary` 之一。如果指定了 `output_encoding` 参数，则返回与该编码格式相符的字符串；如果没有传入该参数，则返回一个 Buffer 实例。

在 `decipher.final()` 方法调用之前，可以多次调用 `decipher.update()` 方法。如果在 `decipher.final()` 之后调用 `decipher.update()` ，则会抛出一个错误。

Class: DiffieHellman

`DiffieHellman` 类封装了一系列函数，用于创建 Diffie-Hellman 秘钥交换协议。

通过调用 `crypto.createDiffieHellman()` 函数可以创建 `DiffieHellman` 类的实例：

```
const crypto = require('crypto');
const assert = require('assert');

// Generate Alice's keys...
const alice = crypto.createDiffieHellman(11);
const alice_key = alice.generateKeys();

// Generate Bob's keys...
const bob = crypto.createDiffieHellman(11);
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
```

diffieHellman.computeSecret(other_public_key[, input_encoding][, output_encoding])

该方法使用 `other_public_key` 作为第三方的弓腰计算共享密钥，最后返回计算后得到的共享密钥。 `input_encoding` 参数指定公钥的编码格式， `output_encoding` 参数指定共享密钥的编码格式。编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果未指定 `input_encoding` 参数，则传入的 `other_public_key` 应为 Buffer 实例。

如果指定了 `output_encoding` 参数，则返回字符串，否则返回 Buffer 实例。

diffieHellman.generateKeys([encoding])

该方法用于生成 Diffie-Hellman 公钥和私钥的值，并根据指定的 `encoding` 格式返回公钥。该值必须传给第三方。编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

diffieHellman.getGenerator([encoding])

根据 `encoding` 参数指定的编码格式返回一个 Diffie-Hellman 的生成器，编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

diffieHellman.getPrime([encoding])

根据 `encoding` 参数指定的编码格式返回一个 Diffie-Hellman 的质数，编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

diffieHellman.getPrivateKey([encoding])

根据 `encoding` 参数指定的编码格式返回一个 Diffie-Hellman 的私钥，编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

diffieHellman.getPublicKey([encoding])

根据 `encoding` 参数指定的编码格式返回一个 Diffie-Hellman 的公钥，编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

diffieHellman.setPrivateKey(private_key[, encoding])

该方法用于设置 Diffie-Hellman 的私钥。如果指定了 `encoding` 的值为 `bianry` 、 `hex` 或 `base64` 中的一个，则 `private_key` 需要是字符串数据，否则 `private_key` 需要时 Buffer 实例。

diffieHellman.setPublicKey(public_key[, encoding])

该方法用于设置 Diffie-Hellman 的公钥。如果指定了 `encoding` 的值为 `bianry` 、 `hex` 或 `base64` 中的一个，则 `public_key` 需要是字符串数据，否则 `public_key` 需要时 Buffer 实例。

diffieHellman.verifyError

该属性包含了 `DiffieHellman` 对象初始化时所产生的所有警告和错误。

该属性的有效值（在 `constants` 模块中定义）有以下几种：

- DH_CHECK_P_NOT_SAFE_PRIME
- DH_CHECK_P_NOT_PRIME
- DH_UNABLE_TO_CHECK_GENERATOR
- DH_NOT_SUITABLE_GENERATOR

Class: ECDH

`ECDH` 类封装了一系列函数，用于创建 Elliptic Curve Diffie-Hellman（ECDN）密钥交换协议。

通过调用 `crypto.createECDH()` 方法可以创建 `ECDH` 类的实例：

```
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.computeSecret(other_public_key[, input_encoding][, output_encoding])

该方法使用 `other_public_key` 作为第三方的弓腰计算共享密钥，最后返回计算后得到的共享密钥。 `input_encoding` 参数指定公钥的编码格式， `output_encoding` 参数指定共享密钥的编码格式。编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果未指定 `input_encoding` 参数，则传入的 `other_public_key` 应为 Buffer 实例。

如果指定了 `output_encoding` 参数，则返回字符串，否则返回 Buffer 实例。

ecdh.generateKeys([encoding[, format]])

该方法用于生成 Diffie-Hellman 公钥和私钥的值，并根据指定的 `encoding` 和 `format` 返回公钥。该值必须传给第三方。

`format` 参数指定了编码格式，值为 `compressed` 、 `uncompressed` 或 `hybrid` 中的一个。如果没有指定 `format` 参数，则默认使用 `compressed` 格式。

`encoding` 参数的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

ecdh.getPrivateKey([encoding])

根据 `encoding` 参数指定的编码格式返回一个 EC Diffie-Hellman 的私钥，编码格式的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

ecdh.getPublicKey([encoding[, format]])

该方法根据 `encoding` 和 `format` 参数返回一个 EC Diffie-Hellman 的公钥。

`format` 参数指定了编码格式，值为 `compressed` 、 `uncompressed` 或 `hybrid` 中的一个。如果没有指定 `format` 参数，则默认使用 `compressed` 格式。

`encoding` 参数的值为 `bianry` 、 `hex` 或 `base64` 中的一个。如果指定了 `encoding` 参数，则返回一个字符串，否则返回一个 Buffer 实例。

ecdh.setPrivateKey(private_key[, encoding])

该方法用于设置 Diffie-Hellman 的私钥。如果指定了 `encoding` 的值为 `bianry` 、 `hex` 或 `base64` 中的一个，则 `private_key` 需要是字符串数据，否则 `private_key` 需要时 Buffer 实例。如果 `ECDH` 创建时指定的 carve 对 `private_key` 无效，将会抛出错误。在配置私钥的同时，ECDH 对象中相应的公钥也会被创建和配置。

echd.setPublicKey(public_key[, encoding])

接口稳定性: 0 - 已过时

Class: Hash

`Hash` 类的实例常用于对数据进行哈希化。使用该类主要有两种方式：

- 包装成可读写的 stream 实例，写入纯数据，读出 Hash 数据
- 使用 `hash.update()` 和 `hash.digest()` 方法生成 Hash 数据

`crypto.createHash()` 方法常用于创建 `Hash` 类的实例。`Hash` 类的实例不能直接由 `new` 关键字创建。

下面代码演示了如果将一个 `Hash` 类的实例封装成 stream 实例：

```
const crypto = require('crypto');
const hash = crypto.createHash('sha256');

hash.on('readable', () => {
  var data = hash.read();
  if (data)
    console.log(data.toString('hex'));
    // Prints:
    //   6a2da20943931e9834fc12cfe5bb47bbd9ae43489a30726962b576f
4e3993e50
});

hash.write('some data to hash');
hash.end();
```

下面代码演示了如何将 `Hash` 类的实例封装成管道流（piped stream)：

```
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);
```

下面代码演示了如何使用 `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.digest([encoding])

该方法用于计算原始数据的哈希摘要。`encoding` 参数的值必须为 `hex`、`binary` 或 `base64` 其中的一个。如果指定了有效的 `encoding` 参数，则该方法返回一个字符串，否则返回一个 Buffer 实例。

在 `hash.digest()` 方法执行之后，不能再重复调用 `Hash` 对象，多次调用该方法将会抛出错误。

hash.update(data[, input_encoding])

该方法根据 `data` 参数更新哈希后的数据，可选参数 `input_encoding` 指定的编码格式必须为 `utf8`、`ascii` 或 `binary` 之一。如果没有指定 `encoding` 参数，且 `data` 为字符串数据，则强制使用 `binary` 编码格式。如果 `data` 是一个 Buffer 实例，则会自动忽略 `input_encoding` 参数的值。

当数据被包装成 stream 实例之后，该方法可以调用多次。

Class: Hmac

`Hmac` 类封装了一系列方法，用于创建加密的 HMAC 摘要。使用该类主要有两种方式：

- 包装成可读写的 stream 实例，写入纯数据，读出 HMAC 摘要
- 使用 `hmac.update()` 和 `hmac.digest()` 方法生成 HMAC 摘要

`crypto.createHmac()` 方法常用于创建 `Hmac` 类的实例。`Hmac` 类的实例不能直接由 `new` 关键字创建。

下面代码演示了如果将一个 `Hmac` 类的实例封装成 stream 实例：

```
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:
    //   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45
971f7f77e
});

hmac.write('some data to hash');
hmac.end();
```

下面代码演示了如何将 `Hmac` 类的实例封装成管道流（piped stream)：

```
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);
```

下面代码演示了如何使用 `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:
//   7fd04df92f636fd450bc841c9418e5825c17f33ad9c87c518115a45971f
7f77e
```

hmac.digest([encoding])

该方法用于计算原始数据的 HMAC 摘要。`encoding` 参数的值必须为 `hex`、`binary` 或 `base64` 其中的一个。如果指定了有效的 `encoding` 参数，则该方法返回一个字符串，否则返回一个 Buffer 实例。

在 `hmac.digest()` 方法执行之后，不能再重复调用 `Hmac` 对象，多次调用该方法将会抛出错误。

hmac.update(data)

该方法根据 `data` 参数更新加密的 HMAC 数据。当数据被包装成 stream 实例之后，该方法可以调用多次。

Class: Sign

`Sign` 类封装了一系列方法，用于生成签名。使用该类主要有两种方式：

- 包装成可写的 stream 实例并对数据进行签名，然后使用 `sign.sign()` 方法生成签名
- 使用 `sign.update()` 和 `sign.sign()` 方法生成签名

`crypto.createHmac()` 方法常用于创建 `Sign` 类的实例。`Sign` 类的实例不能直接由 `new` 关键字创建。

下面代码演示了如果将一个 `Sign` 类的实例封装成 stream 实例：

```
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
```

下面代码演示了如何使用 `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.sign(private_key[, output_format])

计算通过 `sign.update()` 或 `sign.write()` 方法传入的所有数据的签名信息。

`private_key` 参数可以是一个对象或是一个字符串。如果 `private_key` 是一个字符串，将会被当成没有 passphrase 的 key；如果 `private_key` 是一个对象，那么它将会被解释为一个哈希数据，该数据包含两个属性：

- `key`，字符串，使用 PEM 格式加密的私钥
- `passphrase`，字符串，私钥的密码

`output_format` 参数的值必须为 `hex`、`binary` 或 `base64` 其中的一个。如果指定了有效的 `output_format` 参数，则该方法返回一个字符串，否则返回一个 Buffer 实例。

在 `sign.sign()` 方法执行之后，不能再重复调用 `Sign` 对象，多次调用该方法将会抛出错误。

sign.update(data)

根据传入的 `data` 更新签名对象。当数据被包装成 stream 实例之后，该方法可以调用多次。

Class: Verify

`Verify` 类封装了一系列的函数，用于验证签名。使用该类主要有两种方式：

- 包装成可写的 stream 实例并对签名进行验证
- 使用 `verify.update()` 和 `verify.verify()` 方法验证签名

`crypto.createVerify()` 方法常用于创建 `Verify` 类的实例。`Verify` 类的实例不能直接由 `new` 关键字创建。

下面代码演示了如果将一个 `Verify` 类的实例封装成 stream 实例：

```
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
```

下面代码演示了如何使用 `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)

该方法根据传入的 `data` 更新验证对象。当数据被包装成 stream 实例之后，该方法可以调用多次。

verifier.verify(object, signature[, signature_format])

该方法根据传入的 `object` 和 `signature` 参数验证数据。`object` 参数是一个字符串，该字符串包含一个使用 PEM 格式加密的对象，该对象可以是一个 RSA 公钥，可以是一个 DSA 公钥，或者是一个 X.509 证书。`signature` 参数是此前根据数据计算出来的签名。`signature_format` 参数的值可以是 `binary` 、 `hex` 或 `base64` 中的一个。如果指定了 `signature_format` 参数，那么 `signature` 必须是一个字符串，否则， `signature` 必须是一个 Buffer 实例。

该方法根据对数据签名和公钥的验证结果返回 true 或 false。

在 `verify.verify()` 方法执行之后，不能再重复调用 `verifier` 对象，多次调用该方法将会抛出错误。

crypto 模块的成员方法和成员属性

crypto.DEFAULT_ENCODING

该属性用于声明函数的默认编码格式，可以是一个字符串或 Buffer 实例，默认值为 `buffer` ，这表示函数默认接收的 Buffer 对象。

`crypto.DEFAULT_ENCODING` 存在的价值就是为了向后兼容老程序，在老程序中默认的编码格式是 `binary` 。

新的应用程序应该使用默认值 `buffer` 。该属性在未来的 Node.js 版本中可能会被抛弃。

crypto.createCipher(algorithm, password)

该方法根据传入的 `algorithm` 和 `password` 参数创建和返回一个 `Cipher` 对象。

`algorithm` 参数取决于 OpenSSL，比如 `aes192` 等。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的加密算法。

`password` 参数用于派生密钥和初始化向量（IV，initialization vector）。该参数的值必须是一个 `binary` 类型的加密字符串或者 Buffer 数组。

`crypto.createCipher()` 方法使用了 OpenSSL 的 `EVP_BytesToKey` 函数来派生密钥，该函数是 MD5 摘要算法，特点是一次迭代，无需加盐。因为没有加盐，所以容易被字典攻击暴力破解（使用相同的密码产生相同的密钥）。低迭代两和非加密的哈希算法则会让密码被快速的试错。

OpenSSL 建议使用 pbkdf2 替代 `EVP_BytesToKey` ，也建议开发者使用 `crypto.pbkdf2()` 派生密钥和 IV，使用 `crypto.createCipheriv()` 创建 `Cipher` 对象。

crypto.createCipheriv(algorithm, key, iv)

该方法根据传入的 `algorithm` 、 `key` 和 `iv` 创建和返回一个 `Cipher` 对象。

`algorithm` 参数取决于 OpenSSL，比如 `aes192` 等。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的加密算法。

这里的 `key` 是用于 `algorithm` 的原始密钥， `iv` 是一个初始化向量。这两个参数都必须是 `binary` 格式的加密字符串或者 buffer 实例。

crypto.createCredentials(details)

接口稳定性: 0 - 已过时

crtpto.createDecipher(algorithm, password)

该方法根据传入的 `algorithm` 和 `password` 参数创建和返回一个 `Decipher` 对象。

`crypto.createDecipher()` 方法使用了 OpenSSL 的 `EVP_BytesToKey` 函数来派生密钥，该函数是 MD5 摘要算法，特点是一次迭代，无需加盐。因为没有加盐，所以容易被字典攻击暴力破解（使用相同的密码产生相同的密钥）。低迭代两和非加密的哈希算法则会让密码被快速的试错。

OpenSSL 建议使用 pbkdf2 替代 `EVP_BytesToKey`，也建议开发者使用 `crypto.pbkdf2()` 派生密钥和 IV，使用 `crypto.createCipheriv()` 创建 `Decipher` 对象。

crypto.createDecipheriv(algorithm, key, iv)

该方法根据传入的 `algorithm`、`key` 和 `iv` 创建和返回一个 `Decipher` 对象。

`algorithm` 参数取决于 OpenSSL，比如 `aes192` 等。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的加密算法。

这里的 `key` 是用于 `algorithm` 的原始密钥，`iv` 是一个初始化向量。这两个参数都必须是 `binary` 格式的加密字符串或者 buffer 实例。

crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding])

该方法根据传入的 `prime` 参数和可选的 `generator` 参数创建一个 `DiffieHellman` 密钥交换对象。

`generator` 参数可以是数值、字符串和 Buffer 实例。如果未指定 `generator` 函数，则为其添加默认值 `2`。

`prime_encoding` 和 `generator_encoding` 参数可以是 `binary` 、 `hex` 或 `base64` 。

如果指定了 `prime_encoding` ，那么 `prime` 需要是一个字符串，否则， `prime` 需要是一个 Buffer 实例。

如果指定了 `generator_encoding` ，那么 `generator` 需要是一个字符串，否则， `generator` 需要是一个数值或 Buffer 实例。

crypto.createDiffieHellman(prime_length[, generator])

该方法用于创建 `DiffieHellman` 密钥交换对象，并根据 `prime_length` 的大小使用可选的数值类型的 `generator` 生成指定大小的质数。如果未指定 `generator` ，则使用默认值 `2` 。

crypto.createECDH(curve_name)

该方法根据字符串参数 `curve_name` 创建一个 Elliptic Curve Diffie-Hellman 密钥交换对象。使用 `crypto.getCurves()` 可以得到一组可用的 curve 名字。在最新的 OpenSSL 版本中，使用 `openssl ecparam -list_curves` 命令可以每一个可用的椭圆曲线（elliptic curve）的名字和描述。

crypto.createHash(algorithm)

该方法用于创建和返回一个 Hash 对象，该对象可以使用指定 `algorithm` 生成哈希摘要。

`algorithm` 参数的值受限于当前平台 OpenSLL 所支持的算法，比如 `sha256` 、 `sha512` 等。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的摘要算法。

下面代码演示了如何生成一个文件的 sha236 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)

该方法根据传入的 `algorithm` 和 `key` 创建和返回一个 Hmac 对象。

`algorithm` 参数的值受限于当前平台 OpenSLL 所支持的算法，比如 `sha256` 、`sha512` 等。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的摘要算法。

这里的参数 `key` 是 HMAC 用于生成加密 HMAC 哈希的密钥。

下面代码演示了如何生成一个文件的 sha256 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)

该方法根据传入的 `algorithm` 创建和返回一个 `Sign` 对象。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的签名算法，比如 `RSA-SHA256` 。

crypto.createVerify(algorithm)

该方法根据传入的 `algorithm` 创建和返回一个 `Verify` 对象。在最新的 OpneSSL 版本中，使用 `openssl list-cipher-algorithms` 命令可以显示所有可用的签名算法，比如 `RSA-SHA256` 。

crypto.getCiphers()

该方法返回一个数组，该数组描述了当前所支持的所有加密算法：

```
const ciphers = crypto.getCiphers();
console.log(ciphers);
// ['aes-128-cbc', 'aes-128-ccm', ...]
```

crypto.getCurves()

该方法返回一个数组，该数组描述了当前所支持的所有椭圆曲线（elliptic curves):

```
const curves = crypto.getCurves();
console.log(curves); // ['secp256k1', 'secp384r1', ...]
```

crypto.getDiffieHellman(group_name)

该方法用于创建一个约定义的 `DiffieHellman` 密钥交换对象。支持的组包括：`modp1` 、`modp2` 、`modp5` （以上定义在 RFC 2412 中）和 `modp14` 、`modp15` 、`modp16` 、`modp17` 、`modp18` （以上定义在 RFC 3526 中）。返回的对象类似 `crypto.createDiffieHellman()` 方法所返回的对象，但不允许交换密钥。使用该方法的优势在于，无需提前生成或交换组余数，节省了计算和通信时间。

```
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()

该方法返回一个数组，数组内容是当前所支持的 Hash 算法的名字：

```
const hashes = crypto.getHashes();
console.log(hashes); // ['sha', 'sha1', 'sha1WithRSAEncryption',
 ...]
```

crypto.pbkdf2(password, salt, iterations, kenlen[, digest], callback)

该方法是 PBKDF2(Password-Based Key Derivation Function 2) 的一个异步版本。参数 `digest` 指定了 HMAC 摘要算法，该算法根据 `password` 、 `salt` 和 `iterations` 参数创建一个 `keylen` 长度的密钥。如果未指定 `digest` 算法，则使用默认值 `sha1` 。

`callback` 回调函数接收两个参数： `err` 和 `derivedKey` 。如果存在错误，则 `err` 会被赋值，否则 err 的值为 null。成功生成的 `derivedKey` 将会被被传递为一个 Buffer 实例。

`iterations` 参数必须是一个尽可能大的数值，数值越大，密钥的安全性越高，则计算时间也会越长。

`salt` 参数应该尽可能独一无二。建议 `salt` 为随机值且长度大于 16 个字节。

```
const crypto = require('crypto');
crypto.pbkdf2('secret', 'salt', 100000, 512, 'sha512', (err, key
) => {
  if (err) throw err;
  console.log(key.toString('hex'));  // 'c5e478d...1469e50'
});
```

调用 `crypto.getHashes()` 方法可以获取当前支持的摘要函数。

crypto.pbkdf2Sync(password, salt, iterations, keylen[, digest])

该方法是 PBKDF2(Password-Based Key Derivation Function 2) 的一个同步版本。参数 `digest` 指定了 HMAC 摘要算法，该算法根据 `password` 、 `salt` 和 `iterations` 参数创建一个 `keylen` 长度的密钥。如果未指定 `digest` 算法，则使用默认值 `sha1` 。

如果存在错误，则 `err` 会被赋值，否则 err 的值为 null。成功生成的 `derivedKey` 将会被被传递为一个 Buffer 实例。

`iterations` 参数必须是一个尽可能大的数值，数值越大，密钥的安全性越高，则计算时间也会越长。

`salt` 参数应该尽可能独一无二。建议 `salt` 为随机值且长度大于 16 个字节。

```
const crypto = require('crypto');
const key = crypto.pbkdf2Sync('secret', 'salt', 100000, 512, 'sh
a512');
console.log(key.toString('hex'));  // 'c5e478d...1469e50'
```

调用 `crypto.getHashes()` 方法可以获取当前支持的摘要函数。

crypto.privateDecrypt(private_key, buffer)

该方法跟根据传入的 `private_key` 解密 `buffer` 。

`private_key` 参数可以是一个对象或是一个字符串。如果 `private_key` 是一个字符串，将会被当成没有 passphrase 的 key，且会使用 `RSA_PKCS1_OAEP_PADDING` ；如果 `private_key` 是一个对象，那么它将会被解释为一个哈希对象，该对象包含以下属性：

- `key` ，字符串，使用 PEM 格式加密的私钥
- `passphrase` ，字符串，私钥的密码
- `padding` ：一个可选的填充值，有效值包括：
  - constants.RSA_NO_PADDING
  - constants.RSA_PKCS1_PADDING
  - constants.RSA_PKCS1_OAEP_PADDING

所有的填充行为都被定义在了 `constants` 模块中。

crypto.privateEncrypt(private_key, buffer)

该方法跟根据传入的 `private_key` 加密 `buffer` 。

`private_key` 参数可以是一个对象或是一个字符串。如果 `private_key` 是一个字符串，将会被当成没有 passphrase 的 key，且会使用 `RSA_PKCS1_PADDING` ；如果 `private_key` 是一个对象，那么它将会被解释为一个哈希对象，该对象包含以下属性：

- `key` ，字符串，使用 PEM 格式加密的私钥
- `passphrase` ，字符串，私钥的密码
- `padding` ：一个可选的填充值，有效值包括：

    - constants.RSA_NO_PADDING
    - constants.RSA_PKCS1_PADDING
    - constants.RSA_PKCS1_OAEP_PADDING

所有的填充行为都被定义在了 `constants` 模块中。

crypto.publicDecrypt(public_key, buffer)

该方法跟根据传入的 `public_key` 解密 `buffer` 。

`public_key` 参数可以是一个对象或是一个字符串。如果 `public_key` 是一个字符串，将会被当成没有 passphrase 的 key，且会使用 `RSA_PKCS1_PADDING` ；如果 `public_key` 是一个对象，那么它将会被解释为一个哈希对象，该对象包含以下属性：

- `key` ，字符串，使用 PEM 格式加密的私钥
- `passphrase` ，字符串，私钥的密码
- `padding` ：一个可选的填充值，有效值包括：
    - constants.RSA_NO_PADDING
    - constants.RSA_PKCS1_PADDING
    - constants.RSA_PKCS1_OAEP_PADDING

因为 RSA 公钥可以由私钥产生，所以可以通过传递私钥代替传递公钥。

所有的填充行为都被定义在了 `constants` 模块中。

crypto.publicEncrypt(public_key, buffer)

该方法跟根据传入的 `public_key` 加密 `buffer` 。

`public_key` 参数可以是一个对象或是一个字符串。如果 `public_key` 是一个字符串，将会被当成没有 passphrase 的 key，且会使用 `RSA_PKCS1_OAEP_PADDING` ；如果 `public_key` 是一个对象，那么它将会被解释为一个哈希对象，该对象包含以下属性：

- `key` ，字符串，使用 PEM 格式加密的私钥
- `passphrase` ，字符串，私钥的密码
- `padding` ：一个可选的填充值，有效值包括：
    - constants.RSA_NO_PADDING
    - constants.RSA_PKCS1_PADDING

- constants.RSA_PKCS1_OAEP_PADDING

因为 RSA 公钥可以由私钥产生，所以可以通过传递私钥代替传递公钥。

所有的填充行为都被定义在了 `constants` 模块中。

crypto.randomBytes(size[, callback])

该方法用于生成一个高安全等级的伪随机数据，这里的 `size` 参数指定了生成的数据大小，单位是字节。

如果指定了 `callback` ，则使用异步方式生成数据，且 `callback` 函数接收两个参数: `err` 和 `buf` 。如果执行过程中抛出错误， `err` 会被赋值为一个 Error 实例，否则为 null。 `buf` 参数是一个 Buffer 实例，该实例引用了生成后的数据：

```
// Asynchronous
const crypto = require('crypto');
crypto.randomBytes(256, (err, buf) => {
  if (err) throw err;
  console.log(`${buf.length} bytes of random data: ${buf.toStrin
g('hex')}`);
});
```

如果未指定 `callback` 函数，则使用同步的方式生成随机数据并返回一个 Buffer 实例。如果生成过程中有任何错误，则会抛出错误：

```
// Synchronous
const buf = crypto.randomBytes(256);
console.log(`${buf.length}` bytes of random data: ${buf.toString(
'hex')});
```

如果没有足够的熵，则 `crypto.randomBytes()` 方法会停止。通常来说，这只会占用几毫秒的时间。唯一可能会阻塞较长时间的可能是在系统启动后生成随机数据，因为此时整个系统的熵比较低。

crypto.setEngine(engine[, flags])

该方法对某些或全部的 OpenSSL 函数加载和设置 `engine` 。

`engine` 可以是 engine 共享库的 id 或路径。

可选参数 `flags` 的默认值为 `ENGINE_METHOD_ALL`，该值可以是以下列出的一个或多个值：

- `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_METH`
- `ENGINE_METHOD_PKEY_ASN1_METH`
- `ENGINE_METHOD_ALL`
- `ENGINE_METHOD_NONE`

后记

Legacy Streams API (pre Node.js v0.10)

The Crypto module was added to Node.js before there was the concept of a unified Stream API, and before there were Buffer objects for handling binary data. As such, the many of the crypto defined classes have methods not typically found on other Node.js classes that implement the streams API (e.g. update(), final(), or digest()). Also, many methods accepted and returned 'binary' encoded strings by default rather than Buffers. This default was changed after Node.js v0.8 to use Buffer objects by default instead.

Recent ECDH Changes

Usage of ECDH with non-dynamically generated key pairs has been simplified. Now, ecdh.setPrivateKey() can be called with a preselected private key and the associated public point (key) will be computed and stored in the object. This

allows code to only store and provide the private part of the EC key pair. ecdh.setPrivateKey() now also validates that the private key is valid for the selected curve.

The ecdh.setPublicKey() method is now deprecated as its inclusion in the API is not useful. Either a previously stored private key should be set, which automatically generates the associated public key, or ecdh.generateKeys() should be called. The main drawback of using ecdh.setPublicKey() is that it can be used to put the ECDH key pair into an inconsistent state.

Support for weak or compromised algorithms

The crypto module still supports some algorithms which are already compromised and are not currently recommended for use. The API also allows the use of ciphers and hashes with a small key size that are considered to be too weak for safe use.

Users should take full responsibility for selecting the crypto algorithm and key size according to their security requirements.

Based on the recommendations of NIST SP 800-131A:

- MD5 and SHA-1 are no longer acceptable where collision resistance is required such as digital signatures. The key used with RSA, DSA and DH algorithms is recommended to have at least 2048 - bits and that of the curve of ECDSA and ECDH at least 224 bits, to be safe to use for several years.
- The DH groups of modp1, modp2 and modp5 have a key size smaller than 2048 bits and are not recommended.

See the reference for other recommendations and details.

调试器

接口稳定性: 2 - 稳定

Node.js 内置了功能强大的调试工具，该工具可通过 TCP 协议或者内建的调试客户端进行调用。使用时，需要以 `debug` 参数启动 Node.js，后跟要调试的脚本文件路径，调试器启动后会显示一个提示符 `debug>` ：

```
$ 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 的调试器客户端现在还不能支持完整的调试命令，仅支持简单的步进和检查。

将 `debugger;` 语句插入到脚本之中，相当于在特定代码行设置了断点。假设我们有如下所示的一段代码：

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

调试器启动欧，在代码的第四行设置了一个断点，所以程序执行会出现暂停：

```
$ 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
```

通过 REPL 命令可以远程执行调试代码，其中 `next` 命令是单步调试命令，输入 `help` 可以查看完整的调试命令。

监视器

在调试过程中，可以同时监视表达式和变量的值。在每一处断点，监视器都会计算监视目标（表达式和变量）的值。输入 `watch('my_expression')` 即可监视表达式，输入 `watchers` 输出当前所有的监视器，输入 `unwatch('my_expression')` 对表达式取消监视。

命令

步进

- `cont` 或 `c`，继续执行
- `next` 或 `n`，单步执行
- `step` 或 `s`，step in
- `out` 或 `o`，step out
- `pause`，暂停

断点

- `setBreakpoint()` 或 `sb() -`，当前行设置断点
- `setBreakpoint(line)` 或 `sb(line)`，在 `line` 行设置断点
- `setBreakpoint('fn()')` 或 `sb(...)`，在函数里的第一行设置断点
- `setBreakpoint('script.js', 1)` 或 `sb(...)`，在脚本文件的第一行设置断点
- `clearBreakpoint('script.js', 1)` 或 `cb(...)`，清除脚本文件第一行的断点

下面代码演示了如何在未加载的文件中设置断点：

```
$ ./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>
```

信息显示

- `backtrace` 或 `bt`，显示当前执行帧的回溯信息
- `list(5)`，显示脚本代码的五行上下文，即 `debugger;` 之前的五行和之后的五行
- `watch(expr)`，添加对表达式 `expr` 的监视
- `unwatch(expr)`，移除对表达式 `expr` 的监视
- `watchers`，显示所有的监视器和相应的值
- `repl`，启动调试器的 REPL 环境，并使用调试脚本的上下文信息处理代码逻辑
- `exec expr`，在脚本文件的上下文中调试表达式

执行控制

- `run`，执行脚本（脚本在调试器启动时会自执行）
- `restart`，重新执行脚本
- `kill`，杀死正在执行的脚本

其他

- `scripts`，显示已加载脚本的列表
- `version`，显示 V8 的版本

高级用法

此外还有两种方式可以打开调试器：一种方法是使用 `--debug` 参数开启调试器，另一种方法是像 Node.js 进程传递 `SIGUSR1` 信号。

Node.js 进程使用上述方式进入调试模式后，可以使用 Node.js 的调试器通过 PID 或 URI 调试该进程：

- `node debug -p <pid>`，通过 PID 连接到进程
- `node debug <URI>`，- 通过类似 localhost:5858 的 URI 连接到进程

DNS

接口稳定性: 2 - 稳定

`dns` 模块封装的函数主要分为两类：

1. 第一类函数使用操作系统底层提供的功能解析 DNS，无需执行任何网络通信。这一类主要包含一个函数：`dns.lookup()`。开发者如果想像同一操作系统下的其他应用程序一样解析域名，那么就应该使用 `dns.lookup()`。

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

dns.lookup('nodejs.org', (err, addresses, family) => {
  console.log('addresses:', addresses);
});
```

1. 第二类函数需要连接到 DNS 服务器解析域名，而且需要一直使用网络解析 DNS 请求。除 `dns.lookup()` 函数之外，`dns` 模块中的其他函数都属于这一类。这些函数所使用的配置文件与 `dns.lookup()` 的配置文件（比如 `/etc/hosts`）不同。不愿意使用操作系统底层提供的域名解析机制的开发者可以使用此类函数。

下面代码演示了如何解析 `nodejs.org` 并反向解析返回的 IP 地址：

```
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)}
`);
    });
  });
});
```

dns.getServers()

该方法返回一个字符串数组，数组内容是用于域名解析的 IP 地址。

dns.lookup(hostname[, options], callback)

该方法用于将主机名（比如 `nodejs.org`）解析为第一个匹配到的 A 记录（IPv4）或 AAAA 记录（IPv6）。可选参数 `options` 可以是对象或整数。如果未指定 `options` 参数，那么 IPv4 和 IPv6 地址都有效。如果 `options` 是一个整数，那么它必须是 `4` 或 `6`。

如果 `options` 是一个对象，那么它需要包含以下属性：

- `family`，数值，协议族，如果存在，值必须为整数 4 或 6；如果不存在，则表示同时接收 IPv4 和 IPv6
- `hints`，数值，如果存在，则该值为一个或多个的 `getaddrinfo` 标记；如果不存在，表示不向 `getaddrinfo` 传递标记。多个标记使用 `OR` 运算符分隔。

- `all`，布尔值，如果值为 true，则回调函数返回一个数组，该数组包含所有解析后的地址；否则返回一个单一地址。默认值为 false

所有的属性都是可选的，下面是一个简单示例：

```
{
  family: 4,
  hints: dns.ADDRCONFIG | dns.V4MAPPED,
  all: false
}
```

`callback` 回调函数接收三个参数 `(err, address, family)`，其中 `address` 是一个字符串形式的 IPv4 或 IPv6 地址，`family` 参数的值为 4 或 6，用于表示 `address`（无需初始化时传递给 `lookup()` 函数）的协议族。

当 `all` 的值为 true 时，该回调函数的参数将会变为 `(err, addresses)`，其中，`address` 参数是一个对象数组，数组中的每个对象都拥有 `address` 和 `family` 两个属性。

当出现错误时，`err` 参数的值为一个 Error 实例，其中 `err.code` 为错误码。需要牢记的是，不仅仅主机名不存在的时候，而且在 lookup() 函数执行错误时（没有可用的文件描述符），错误码都有可能会被赋值为 `ENOENT`。

`dns.lookup()` 并没有任何处理 DNS 协议的逻辑，它使用了操作系统所提供的功能来解析地址。对于不同的 Node.js 程序来说，该方法的执行结果可能会有略微但重要的差异。

受支持的 getaddrinfo 标记

下面标记都可作为 `hints` 字段的值传递给 `dns.lookup()`：

- `dns.ADDRCONFIG`：返回当前系统支持的地址类型，比如，如果当前系统至少配置了一个 IPv4 地址，那么就会返回一个 IPv4 地址。局部回送位址（loopback address）不计在内。
- `dns.V4MAPPED`：如果指定了 IPv6 协议族，但没有找到 IPv6，那么就返回由 IPv4 映射的 IPv6 地址。注意，这一标记在某些操作系统上无效，比如 FreeBSD 10.1。

dns.lookupServce(address, port, callback)

该方法使用操作系统的底层 `getnameinfo` 将传入的 `address` 和 `port` 解析为主机名和服务名。

`callback` 回调函数接收三个参数 `(err, hostname, service)`。这里的 `hostname` 和 `service` 参数都是字符串类型，比如 `localhost` 和 `http`。

当出现错误时，`err` 的值是一个 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
});
```

dns.resolve(hostname[, rrtype], callback)

根据 `rrtype` 指定的记录类型使用 DNS 协议将主机名解析为一个数组。

`rrtype` 的有效值包括：

- `'A'`，IPv4 地址，rrtype 的默认值
- `'AAAA'`，IPV6 地址
- `'MX'`，邮件交换记录
- `'TXT'`，文本记录
- `'SRV'`，SRV 记录
- `'PTR'`，用来反向查找 IP
- `'NS'`，域名服务器记录
- `'CNAME'`，别名记录
- `'SOA'`，授权记录的初始值

`callback` 回调函数接收两个参数 `(err, address)`。执行成功时，`address` 参数为一个数组，数组成员的类型由记录类型决定。

当出现错误时，`err` 的值为一个 Error 对象，其中 `err.code` 是一个错误码。

dns.resolve4(hostname, callback)

该方法使用 DNS 协议根据主机名解析 IPv4 地址（A 记录）。传入 `callback` 回调函数的 `addresses` 参数为一个 IPv4 地址的数组，比如 `['74.125.79.104', '74.125.79.105', '74.125.79.106']`。

dns.resolve6(hostname, callback)

该方法使用 DNS 协议根据主机名解析 IPv6 地址（AAAA 记录）。传入 `callback` 回调函数的 `addresses` 参数为一个 IPv6 地址的数组。

dns.resolveCname(hostname, callback)

该方法使用 DNS 协议根据主机名解析 `CNAME`。传入 `callback` 回调函数的 `addresses` 参数为主机名的别名，比如 `['bar.example.com']`。

dns.resolveMx(hostname, callback)

该方法使用 DNS 协议根据主机名解析邮件交换协议（MX 记录）。传入 `callback` 回调函数的 `addresses` 参数为一个对象数组，数组中的对象都包含 `priority` 和 `exchange` 两个属性，比如 `[{priority: 10, exchange: 'mx.example.com'}, ...]`。

dns.resolveNs(hostname, callback)

该方法使用 DNS 协议根据主机名解析域名服务器记录（NS 记录）。传入 `callback` 回调函数的 `addresses` 参数为一个主机名可用的域名服务器记录数组，比如 `['ns1.example.com', 'ns2.example.com']`。

dns.resolveSoa(hostname, callback)

该方法使用 DNS 协议根据主机名解析授权记录的初始值（SOA 记录）。传入 `callback` 回调函数的 `addresses` 参数为一个包含以下属性的对象：

- `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.resolveSrv(hostname, callback)

该方法使用 DNS 协议根据主机名解析 SRV 记录。传入 `callback` 回调函数的 `addresses` 参数为一个对象数组，数组中的对象都包含以下属性：

- `priority`
- `weight`
- `port`
- `name`

```
{
  priority: 10,
  weight: 5,
  port: 21223,
  name: 'service.example.com'
}
```

dns.resolveTxt(hostanme, callback)

该方法使用 DNS 协议根据主机名解析文本查询（TXT 记录）。传入 `callback` 回调函数的 `addresses` 参数为一个二维数组，比如 `[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]` 。每一个二级数组都包含一条记录的 TXT 块。根据实际需求，可以合并和分开独立使用。

dns.reverse(ip, callback)

该方法用于执行反向 DNS 查询，将 IPv4 或 IPv6 地址解析为一个主机名的地址。

`callback` 回调函数接收两个参数 `(err, hostnames)` ，其中， `hostnames` 是根据传入的 `ip` 解析得来的主机名数组。

当出现错误时， `err` 的值是一个 Error 对象，其中 `err.code` 是错误码。

dns.setServers(servers)

该方法用于指定一组 IP 地址作为解析服务器。 `servers` 参数是一个 IPv4 或 IPv6 地址的数组。

如果地址中包含端口信息，将会被自动移除。

如果传入的地址无效，则将会抛出错误。

不能在 DNS 查询时执行 `dns.setServers()` 方法。

错误码

每一次 DNS 查询都会返回下列错误码中的一个：

- `dns.NODATA` ：DNS 服务器返回无数据应答
- `dns.FORMERR` ：DNS 服务器认为查询各式错误
- `dns.SERVFAIL` ：DNS 服务器返回常规失败
- `dns.NOTFOUND` ：未找到域名
- `dns.NOTIMP` ：DNS 服务器为实现当前请求的操作
- `dns.REFUSED` ：DNS 服务器拒绝查询
- `dns.BADQUERY` ：DNS 查询格式错误
- `dns.BADNAME` ：主机名格式错误
- `dns.BADFAMILY` ：不支持的地址族

- `dns.BADRESP`：DNS 回复格式错误
- `dns.CONNREFUSED`：无法连接到 DNS 服务器
- `dns.TIMEOUT`：连接 DNS 服务器超时
- `dns.EOF`：文件结尾
- `dns.FILE`：读取文件发生错误
- `dns.NOMEM`：内存溢出
- `dns.DESTRUCTION`：信道已被销毁
- `dns.BADSTR`：字符串格式错误
- `dns.BADFLAGS`：非法标识
- `dns.NONAME`：给定的主机名不是数值类型
- `dns.BADHINTS`：非法 Hints 标识
- `dns.NOTINITIALIZED`：c-ares 库尚未初始化
- `dns.LOADIPHLPAPI`：iphlpapi.dll 加载失败
- `dns.ADDRGETNETWORKPARAMS`：无法找到 GetNetworkParams 函数
- `dns.CANCELLED`：DNS 查询已取消

开发提示

虽然 `dns.lookup()` 和 `dns.resolve*()/dns.reverse()` 函数都是用于关联网络名和网络地址，但它们的内部机制还是存在细微不同。差异虽小但在不同的 Node.js 程序中可能会有严重的影响。

dns.lookup()

本质上说，`dns.lookup()` 像大多数程序一样都是使用了操作系统的底层函数。举例来说，`dns.lookup()` 会像 `ping` 命令一样解析域名。在大多数的类 POSIX 操作系统上，可以通过修改 `nsswitch.conf(5)` 或 `resolv.conf(5)` 中的配置修改 `dns.lookup()` 函数的行为，但需要注意的是，上述修改会影响所有运行在当前操作系统上的程序。

虽然从 JavaScript 的角度看 `dns.lookup()` 是异步调用，但实际上在 libuv 的线程池中是通过同步调用 `getaddrinfo(3)` 实现。因为 libuv 线程池是大小固定的，也就是说，如果因为某些因素导致 `getaddrinfo(3)` 函数长时间运行，那么线程池中其他操作的性能就会降低。为了降低这一风险，一个有效的方法是增大环境变量 `'UV_THREADPOOL_SIZE'` 的值（默认值为 4），从而增加线程池的大小。更多信息请参考 libuv 的官方文档。

dns.resolve(), dns.resolve*() 和 dns.reverse()

这些方法和 `dns.lookup()` 的机制完全不一样。它们不使用 `getaddrinfo(3)` ，而是通过网络进行 DNS 查询。通常来说，这里的网络通信都是异步执行的，并不会使用到 libuv 的线程池。

因此，这些函数的操作不会对 libuv 线程池中的其他进程产生类似 `dns.lookup()` 函数所产生的负面影响。

这些函数也不会用到 `dns.lookup()` 函数所用到的配置文件，比如，它们不会使用 `/etc/hosts` 的配置文件。

Domain

接口稳定性: 0 - 已过时

该模块即将被抛弃，在替代该模块的 API 完成之后，将彻底抛弃该模块。

Errors

基于 Node.js 开发的应用通常会遇到以下四类错误：

- 标准的 JavaScript 错误：
  - `[EvalError][]`：调用 `eval()` 出现错误时抛出该错误
  - `SyntaxError`：代码不符合 JavaScript 语法规范时抛出该错误
  - `[RangeError][]`：数组越界时抛出该错误
  - `ReferenceError`：引用未定义的变量时抛出该错误
  - `TypeError`：参数类型错误时抛出该错误
  - `[URIError][]`：误用全局的 URI 处理函数时抛出该错误
- 由操作系统底层触发的系统错误，比如尝试打开不存在的文件、尝试通过已关闭的 socket 发送数据等
- 用户自定义的错误，通常在应用程序运行过程中触发
- 断言错误，当 Node.js 检测到逻辑错误时会触发该错误。通常来说，此类错误由 `assert` 模块抛出

Node.js 抛出的所有 JavaScript 和系统错误都是 `Error` 类的实例，每个实例都至少有一个引用该类的属性。

错误的传播和拦截

Node.js 提供了多种传播和处理错误的机制，具体的而传播和处理机制需要根据错误类型和调用的 API 类型来确定。

所有的 JavaScript 错误都会被视为异常，使用 `tre / catch` 命令可以立即将其抛出：

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

必须配合 `try / catch` 使用 JavaScript 的 `throw` 机制，否则将会立即中断 Node.js 的进程。

除少数情况之外，Node.js 使用同步版本的 API 处理异常。

异步版本的 API 有多种方式记录错误：

- 大多数的异步方法都接收回调函数，回调函数的第一个参数就是抛出的 `Error`。如果第一个参数的值为不是 `null` 且是 Error 的实例，则表示抛出了错误，需要开发者进行处理：

```
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
});
```

- 调用异步方法的 `EventEmitter` 对象，可以通过监听 `error` 时间捕获错误：

```
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);
```

- Node.js 中少数的异步方法仍然采用了 `throw` 机制抛出错误，对此需要使用 `try / catch` 捕获错误。对于此类异步方法，目前尚没有完整的统计列表，请参考本文档中的对 API 的详细说明，采用合适的机制处理错误。

基于 stream 和基于 event emitter 的 API 通常都是通过监听 `error` 事件处理异常的，这些 API 大都封装了一系列的异步操作。

对于所有的 `EventEmitter` 对象，如果未监听 `error` 事件，除非恰当地使用了 `domain` 模块或者为 `process.on('uncaughtException')` 事件设置了处理函数，否则就会上抛错误，导致 Node.js 进程崩溃并报告该错误：

```
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'));
});
```

不能使用 `try / catch` 来捕获此类错误，这是因为捕获行为发生在触发该错误的代码之后。

开发者有必要仔细阅读本文档，了解每一个方法的错误传播方式。

Node.js 风格的回调函数

Node.js 中大多数的异步方法都遵循一个被称为 `Node.js 式回调函数` 的通用模式。在这一模式中，回调函数将传给异步方法作为一个参数。无论异步方法是否成功执行，最终都会执行回调函数，且回调函数的第一个参数就是一个 Error 对象。如果没有错误，则回调函数的第一个参数的值为 `null` ：

```
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)
```

不能使用 `try / catch` 来捕获异步方法的错误。对于初学者常见的错误就是在回调函数中使用 `throw` 的方式：

```
// 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);
}
```

上面的错误捕获方式并不会生效，这是因为 `fs.readFile()` 的回调函数是异步调用的。当回调函数被调用时，`try / catch` 代码块已经退出了执行期。回调函数中抛出的错误往往会让 Node.js 进程崩溃。如果恰当地使用了 `domain` 模块或者为 `process.on('uncaughtException')` 事件设置了处理函数，则可以避免此类崩溃。

Class: Error

JavaScript 的 `Error` 对象并不会显示错误发生的具体环境，但它会显示 `Error` 对象实例化时的堆栈信息，并提供有关的描述信息

在 Node.js 的程序中，无论是系统错误还是 JavaScript 错误，统统是 `Error` 类的实例。

new Error(message)

该方法用于创建 `Error` 对象，并根据 `message` 参数配置 `error.message` 属性。如果 `message` 参数是一个对象，则调用 `message.toString()` 转换为字符串。`error.stack` 属性用于表示 `new Error()` 在代码中被调用的信息。对堆栈的跟踪依赖于 V8 的堆栈跟踪 API。堆栈跟踪只会提供开始执行同步代码的信息或者 `Error.stackTraceLimit` 属性指定范围内的调用帧信息。

Error.captureStackTrace(targetObject[, constructorOpt])

该方法用于在 `targetObject` 上创建一个 `.stack` 属性，该属性的值是一个字符串，表示 `Error.captureStackTrace()` 方法被调用的位置信息。

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

`ErrorType: message` 后的第一行堆栈跟踪信息是 `targetObject.toString()` 的返回值。

可选参数 `constructorOpt` 是一个函数。如果指定了该函数，则从堆栈跟踪开始，`constructorOpt` 之上或之内的调用帧都会被忽略。

`constructorOpt` 参数常用于对用户隐藏错误发生的细节，举例如下:

```
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` 属性指定了堆栈跟踪器收集堆栈信息的最大容量。

虽然该属性的默认值为 10，但是开发者可以修改为任何有效的 JavaScript 数值。该属性值修改后立即生效。

如果将该属性的值为非数值或负值，则堆栈跟踪器不捕获任何调用帧的信息。

error.message

该属性返回一个字符串，即 `new Error(message)` 中的 `message` 参数。传递给构造函数的 `message` 参数将会出现在 `Error` 的堆栈信息的首行。`Error` 对象创建之后再修改该属性也许并不能修改堆栈信息：

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

error.stack

该属性返回一个字符串，表示 Error 对象初始化方面的信息：

```
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)
```

第一行是固定的 `<error class name>: <error message>`，其后是一系列的调用栈信息（每一行都以 `at` 开头）。每一帧都描述了代码中的一个错误抛出点。V8 会尝试显示每一个函数的函数名，但有时候并无法找到合适的名字。如果 V8 无法确定函数名，那么就会只显示位置信息；如果可以确定函数名，则会同时显示函数名和位置信息，此时位置信息置于尾部的括号中。

有一点非常值得注意，那就是只有 JavaScript 函数会产生调用帧。举例来说，如果将可执行的异步方法传给一个名为 `cheetahify` 的 C++ 插件，则发生错误时，并不会显示该插件有关的堆栈跟踪信息：

```
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
```

位置信息为以下类型之一：

- `native`，V8 内部的调用帧
- `plain-filename.js:line:column`，Node.js 内部的调用帧
- `/absolute/path/to/file.js:line:column`，基于 Node.js 的程序或依赖所产生的调用帧

当 `error.stack` 可访问时，该字符串的生成速度较慢。

堆栈跟踪的调用帧数量在小于 `Error.stackTraceLimit`。

Class: RangeError

该类是 `Error` 类的子类，用于表示函数接收到了指定范围之外的参数，该范围可能是数值范围也可能是一个可选列表：

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

参数检验完成后，Node.js 会立即生成和抛出 `RangeError` 实例。

Class: ReferenceError

该类是 `Error` 类的一个子类，用于表示访问的变量不存在。此类错误往往是由代码中的拼写错误或其他编写问题引起的。

虽然客户端代码可以产生或抛出此类错误，但实际上，此类代码是由 V8 产生和抛出的：

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

`ReferenceError` 实例都拥有一个 `error.arguments` 属性，该属性的值为一个数组，且数组只有一个字符串，该字符串用于表示当前变量未定义：

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

除非应用程序动态生成和执行代码，否则代码中或依赖环境中的 `ReferenceError` 实例都应该被视为一个 Bug.

Class: SyntaxError

该类是 `Error` 类的子类，用于表示当前程序代码不符合 JavaScript 语法规范。此类错误通常只会出现在代码评估阶段。代码评估通常指 `eval`、`Function`、`require` 或 `vm` 的执行结果。此类错误通常会中断程序的执行：

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

`SyntaxError` 实例无法在触发该错误的上下文中捕获，只能在其他上下文中捕获。

Class: TypeError

该类是 `Error` 类的子类，用于表示传入的参数不符合函数要求。举例来说，当某个方法只接受字符串参数时，如果传入的参数是一个函数，那么就会抛出该错误：

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

参数检验完成后，Node.js 会立即生成和抛出 `TypeError` 实例。

异常 VS 错误

JavaScript 中的异常都是一个值，表示无效的操作或 `throw` 的对象。这些值要么是 `Error` 的实例，要么就是继承自 `Error`。所有由 Node.js 或 JavaScript 运行环境抛出的异常都是 Error 的实例。

有一些异常无法在 JavaScript 层面上捕获。此类异常通常会让 Node.js 进程崩溃，比如在 C++ 层调用 `abort()` 或 `assert()` 都会产生这种异常。

系统错误

在程序的运行环境中抛出异常时，就会产生系统错误。通常来说，当应用程序违反操作系统的约束条件时就会出触发此类错误，比如试图读取一个不存在的文件或者当前操作的权限过低等等。

系统错误通常在系统调用时产生。在 Unix 系统下，通过执行 `man 2 intro` 或 `man 3 errno` 可以获取完整的错误码列表及其简介。

在 Node.js 中，系统错误丰富了 `Error` 对象，该对象中有专门属性用于描述系统错误。

Class: System Error

error.code && error.errno

该属性返回一个字符串，表示错误码，通常以大写字母 `E` 开头，详见 `man 2 intro` 命令的解释。

`error.code` 和 `error.errno` 的功能相同，返回的值也相同。

error.syscall

该方法返回一个字符串，用于表示失败的系统调用（syscall）。

常见的系统错误

下面的列表并不完整，只是一些开发 Node.js 程序时常见的系统错误：

- `EACCES` (Permission denied): An attempt was made to access a file in a way forbidden by its file access permissions.
- `EADDRINUSE` (Address already in use): An attempt to bind a server (net, http, or https) to a local address failed due to another server on the local system already occupying that address.
- `ECONNREFUSED` (Connection refused): No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
- `ECONNRESET` (Connection reset by peer): A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or reboot. Commonly encountered via the http and

net modules.

- `EEXIST` (File exists): An existing file was the target of an operation that required that the target not exist.
- `EISDIR` (Is a directory): An operation expected a file, but the given pathname was a directory.
- `EMFILE` (Too many open files in system): Maximum number of file descriptors allowable on the system has been reached, and requests for another descriptor cannot be fulfilled until at least one has been closed. This is encountered when opening many files at once in parallel, especially on systems (in particular, OS X) where there is a low file descriptor limit for processes. To remedy a low limit, run ulimit -n 2048 in the same shell that will run the Node.js process.
- `ENOENT` (No such file or directory): Commonly raised by fs operations to indicate that a component of the specified pathname does not exist -- no entity (file or directory) could be found by the given path.
- `ENOTDIR` (Not a directory): A component of the given pathname existed, but was not a directory as expected. Commonly raised by fs.readdir.
- `ENOTEMPTY` (Directory not empty): A directory with entries was the target of an operation that requires an empty directory -- usually fs.unlink.
- `EPERM` (Operation not permitted): An attempt was made to perform an operation that requires elevated privileges.
- `EPIPE` (Broken pipe): A write on a pipe, socket, or FIFO for which there is no process to read the data. Commonly encountered at the net and http layers, indicative that the remote side of the stream being written to has been closed.
- `ETIMEDOUT` (Operation timed out): A connect or send request failed because the connected party did not properly respond after a period of time. Usually encountered by http or net -- often a sign that a socket.end() was not properly called.

事件

接口稳定性: 2 - 稳定

Node.js 的核心 API 大都是基于异步事件驱动架构（asynchronous event-driven architecture）构建的，在这一架构中，触发器（emitter）定期触发某些事件执行监听对象（listener）。

举例来说，每次有连接时，`net.Server` 就会触发一个事件；当文件打开时，`fs.ReadStream` 就会触发一个事件；当数据可访问时，`stream` 就会触发一个事件。

所有可以触发事件的对象都是 `EventEmitter` 类的实例。这些对象通过 `eventEmitter.on()` 函数监听特定的事件。通常来说，事件名称统一使用驼峰字符串表示，但实际上任何有效的 JavaScript 键名都可以做事件名。

当 `EventEmitter` 的实例对象触发事件之后，绑定在该事件之下的所有函数会被以同步执行的方式调用。任何监听器返回的值都会被忽略和抛弃。

下面代码演示了一个简单的 `EventEmitter` 实例及其监听的事件。`eventEmitter.on()` 方法用于注册监听器，`eventEmitter.emit()` 方法用于触发事件。

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

function MyEmitter() {
  EventEmitter.call(this);
}
util.inherits(MyEmitter, EventEmitter);

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

通过继承，任何对象都可以成为 `EventEmitter` 类的实例。上面代码通过 `util.inherits()` 方法使用传统的原型继承实现了继承机制。此外,也可以使用 ES6 的类来实现：

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

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

向监听器传递参数和上下文

`eventEmitter.emit()` 方法允许向监听器函数传递一组参数。值得注意的是， `EventEmitter` 调用监听函数时， `this` 仍然指向监听器：

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

虽然可以使用 ES6 的箭头函数创建监听器，但是这样的话， `this` 将会不再指向 `EventEmitter` 的实例对象：

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

异步 VS 同步

`EventListener` 会根据监听顺序以同步的方式调用所有的监听函数。这有助于确保事件触发的合理顺序，避免竞争条件和逻辑错误。在某些情况下，我们可以通过使用 `setImmediate()` 或 `process.nextTick()` 方法以异步的方式调用监听函数：

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously');
  });
});
myEmitter.emit('event', 'a', 'b');
```

一次性事件

使用 `eventEmitter.on()` 方法注册的监听器对事件的每一次触发做出响应：

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

使用 `eventEmitter.once()` 方法注册的监听器在事件第一次被触发后就会注销该事件：

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

错误事件

当 `EventEmitter` 的实例出现错误时，就会触发 `error` 事件，这在 Node.js 中被视为一个特殊事件。

如果 `EventEmitter` 的实例没有监听 `error` 事件，就会将该错误向上传播，直到抛出该错误，打印出堆栈跟踪信息，最终退出当前的 Node.js 进程：

```
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));
// Throws and crashes Node.js
```

为了避免 Node.js 的进程崩溃，开发者可以使用 `domain` 模块（不过该模块已被抛弃）或者注册一个 `process.on('uncaughtException')` 来解决:

```
const myEmitter = new MyEmitter();

process.on('uncaughtException', (err) => {
  console.log('whoops! there was an error');
});

myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an error
```

最好的预防措施就是，开发者在使用监听器时始终都应该监听 `error` 事件：

```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
  console.log('whoops! there was an error');
});
myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an erro
```

Class: EventEmitter

`EventEmitter` 类定义在 `events` 模块中：

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

所有的事件触发器在注册监听器的时候都会触发 `newListener` 事件，在注销监听器的时候都会触发 `removeListener` 事件。

事件：'newListener'

- `event`，字符串或 Symbol，事件名称
- `listener`，函数，事件处理函数

`EventEmitter` 的实例在监听器被添加到监听列表之前就会首先触发 `newListener` 事件。

`newListener` 时间的监听器接收两个参数，一个是事件名称，另一个是已添加的监听器。

实际上，在添加监听器之前触发该事件有一点负面影响：在 `newListener` 事件中注册的同名事件，将会比外部注册的事件晚触发：

```
const myEmitter = new MyEmitter();
// Only do this once so we don't loop forever
myEmitter.once('newListener', (event, listener) => {
  if (event === 'event') {
    // Insert a new listener in front
    myEmitter.on('event', () => {
      console.log('B');
    });
  }
});
myEmitter.on('event', () => {
  console.log('A');
});
myEmitter.emit('event');
// Prints:
//   B
//   A
```

事件：'removeListener'

- `event`，字符串或 Symbol，事件名称
- `listener`，函数，事件处理函数

注销监听器之后触发该事件。

EventEmitter.listenerCount(emitter, event)

接口稳定性: 0 - 已过时

请使用 emitter.listenerCount() 代替该方法。

EventEmitter.defaultMaxListeners

默认情况下，任何事件可以注册 10 个监听器。`EventEmitter` 的实例可以通过 `emitter.setMaxListeners(n)` 方法修改这一限制值。如果要为所有的 `EventEmitter` 实例修改该属性，需要使用 `EventEmitter.defaultMaxListeners` 属性。

需要注意的是，使用 `EventEmitter.defaultMaxListeners` 修改该限制值会影响所有的 `EventEmitter` 实例，包括之前已经注册的 `EventEmitter` 实例。

此外需要留意的是，这并不是一个硬性限制指标。`EventEmitter` 的实例允许添加多个监听器，但是会在控制条输出一个提示 `possible EventEmitter memory leak` 。可以使用 `emitter.getMaxListeners()` 和 `emitter.setMaxListeners()` 方法临时避免这种警告：

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', () => {
  // do stuff
  emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1
, 0));
});
```

emitter.addListener(event, listener)

emitter.on(event, listener) 的同名函数。

emitter.emit(event[, arg1][, arg2][, ...])

该方法以同步的方式调用指定 `event` 的每一个监听器，并将参数传递给它们。

如果 `event` 拥有监听器，则返回 true，否则返回 false。

emitter.getMaxListeners()

该方法返回 `EventEmitter` 可以设置的最大监听器数量，该值可以由 `emitter.setMaxListeners(n)` 方法修改，其默认值由 `EventEmitter.defaultMaxListeners` 决定。

emitter.listenerCount(event)

- `event` ，事件类型

该方返回当前 `event` 类型所监听的监听器数量。

emitter.listeners(event)

该方法先拷贝 `event` 事件的监听器列表，然后将其以数组的方式返回：

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection')));
// Prints: [ [Function] ]
```

emitter.on(event, listener)

该方法用于给指定的 `event` 添加监听器，并将该监听器置于监听器列表的末位。该方法不会检查是否已经添加过该监听器。重复添加相同的 `event` 和 `listener` 会导致该事件和监听器被重复触发：

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
```

该方法返回一个对 `EventEmitter` 实例的引用，所以可以使用链式调用。

emitter.once(event, listener)

该方法为 `event` 事件添加一个一次性的监听器，该事件第一次触发之后就会被注销：

```
server.once('connection', (stream) => {
  console.log('Ah, we have our first user!');
});
```

该方法返回一个对 `EventEmitter` 实例的引用，所以可以使用链式调用。

emitter.removeAllListener([event])

该方法用于注销所有的事件，或者注销指定的 `event`。

注意，不建议使用该方法注销其他组件或模块监听的事件，尤其是 `EventEmitter` 由其他模块或组件创建的时候。

该方法返回一个对 `EventEmitter` 实例的引用，所以可以使用链式调用。

emitter.removeListener(event, listener)

该方法用于移除 `event` 事件的监听器列表中的 `listener` ：

```
var callback = (stream) => {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener` 对于某个事件的 `listener` 每次只会移除一个，所以如果某个 `listener` 被添加了多次，就需要执行多次 `removeListener` 才能完全注销该监听器。

因为监听器由内部数组进行管理，所以调用该方法会修改监听器在监听列表中的顺序。这并不会影响已调用的监听器，但会影响通过 `emitter.listeners()` 创建的监听器副本，所以可能需要重建该副本。

该方法返回一个对 `EventEmitter` 实例的引用，所以可以使用链式调用。

emitter.setMaxListeners(n)

默认情况下，当某个事件设置的监听器超过十个时，EventEmitter 实例就会抛出警告，这有助于开发者找出程序中的内存泄露。显而易见的是，并不是每一个事件的监听器都会少于十个，所以可以使用 `emitter.setMaxListeners()` 方法修改特定 `EventEmitter` 实例的监听器数量。如果限制值为 `Infinity` 或 `0` ，则表示不限制监听器的数量。

该方法返回一个对 `EventEmitter` 实例的引用，所以可以使用链式调用。

文件系统

接口稳定性: 2 - 稳定

文件系统模块是一个封装了标准 POSIX 文件 I/O 操作的集合。通过 `require('fs')` 可以加载该模块。该模块中的所有方法都有异步执行和同步执行两个版本。

异步执行的方法都接收一个回调函数作为最后一个参数。传递给回调函数的参数取决于具体的异步方法，但第一个参数通常用于接收异步方法的错误信息。如果异步执行没有问题，则第一个表示错误信息的参数通常为 `null` 或 `undefined`。

当执行同步方法时，发生的任何错误都会直接抛出。开发者可以使用 `try/catch` 捕获异常，或者直接放纵异常继续向上传播。

下面代码演示了如何使用异步执行的方法：

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

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

下面代码演示了如何使用同步执行的方法：

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

fs.unlinkSync('/tmp/hello');
console.log('successfully deleted /tmp/hello');
```

由于异步方法没有固定的执行顺序，所以下面的这段代码很有可能会抛出错误：

```
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)}`);
});
```

这是因为 `fs.stat` 方法有可能会在 `fs.rename` 之前执行，正确的方式是嵌套执行这段代码：

```
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)}`);
  });
});
```

在负载较重的进程中，建议使用异步执行的方法，这是因为同步方法会阻塞进程的执行，在同步方法结束之前，所有的连接和任务都会处于挂起状态。

在该模块中可以使用文件的相对路径，但是需要牢记，相对路径的相对目标是 `process.cwd()`。

该模块中的大多数函数都允许开发者忽略回调函数。如果忽略了回调函数，系统就会使用一个默认的回调函数，用于抛出执行过程中出现的错误。为了获取函数调用点的原始堆栈追踪信息，需要开启环境变量 `NODE_DEBUG`：

```
$ 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.>
```

Class: fs.FSWatcher

`fs.watch()` 返回的对象都属于该类型。

事件：'change'

- `event`，字符串，改变的 fs 类型
- `filename`，字符串，修改后的文件名

当监视的文件或目录发生变化时触发该事件。

事件：'error'

- `error`，Error 实例

发生错误时触发该事件。

watcher.close()

该方法用于停止监控 `fs.FSWatcher` 指定的文件或目录。

Class: fs.ReadStream

`ReadStream` 是一个可读的 Stream 实例。

事件：'open'

- `fd`，数值，传递给 ReadStream 的文件描述符的值

当 ReadStream 中的文件打开时，触发该事件。

readStream.path

该属性表示表示 stream 读取的文件路径。

Class: fs.Stats

`fs.stat()`、`fs.lstat()`、`fs.fstat()` 及相应的同步版本都会返回该类型的对象：

- stats.isFile()
- stats.isDirectory()
- stats.isBlockDevice()
- stats.isCharacterDevice()
- stats.isSymbolicLink() (only valid with fs.lstat())
- stats.isFIFO()
- stats.isSocket()

对于普通文件来说，`util.inspect(stats)` 返回类似如下所示的内容：

```
{
  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
}
```

值得注意的是，这里的 `atime` 、 `mtime` 、 `birthtime` 和 `ctime` 都是 Date 对象的实例，需要使用合适的方法来比较相互之间的大小。 `getTime()` 方法返回自 1970 年 1 月 1 日至今为止的毫秒数，这个方法基本满足大多数的比较条件。此外还有一些方法可以显示额外的信息，更多方法请参考 MDN JavaScript Reference。

Stat 的事件值

stat 对象中的时间值有如下含义：

- `atime` ，访问时间，表示文件的最后一次访问时间，调用系统函数 mknod(2)、utimes(2) 和 read(2) 会修改该值。
- `mtime` ，内容修改时间，表示文件内容的最后一次修改时间，调用系统函数 mknod(2)、utimes(2) 和 write(2) 会修改该值。
- `ctime` ，修改时间，表示文件的最后一次修改时间，包括但不限于内容的修改，调用系统函数 chmod(2)、chown(2)、link(2)、mknod(2)、rename(2)、unlink(2)、utimes(2)、read(2) 和 write(2) 会修改该值。
- `birthtime` ，创建时间，表示文件创建的时间，只在文件创建时初始化该值。如果文件的创建时间不可用，那么该值可能是 `ctime` 或 `1970-01-01T00:00Z` 。在 Darwin 或其他 FreeBSD 系统中，会通过 `utimes(2)` 将

`atime` 设置的比 `birthtime` 更早。

在 Node.js v0.12 版本之前，Windows 系统的 ctime 函数 birthtime。注意在 Node.js v0.12 中，ctime 并不是文件创建时间，在 Unix 系统中，它一直都不是文件创建时间。

Class: fs.WriteStream

`WriteStream` 是一个可写的 Stream 实例。

事件：'open'

- `fd`，数值，传递给 WriteStream 的文件描述符的值

当 WriteStream 中的文件打开时，触发该事件。

writeStream.bytesWritten

当前写入的字节数，不包含等待写入的数据。

writeStream.path

该属性表示表示 stream 写入的文件路径。

fs.access(path[, mode], callback)

该方法用于测试用户是否对 `path` 指定的文件有足够的权限。可选参数 `mode` 是一个数值，用于指定需要检查的权限。以下变量是 `mode` 的可选值，通过 OR 可以指定多个值：

- `fs.F_OK`，可见权限，常用于检查文件是否存在，但如果文件的权限是 `rwx` 则不会返回任何值，改制为 `mode` 的默认值
- `fs.R_OK`，可读权限
- `fs.W_OK`，可写权限
- `fs.X_OK`，可执行权限，该权限对 Windows 无效

该函数的最后一个参数是一个回调函数 `callback`，其接收一个引用错误信息的参数。如果任一访问权限不足，就会抛出该错误。下面代码检查了当前进程是否可以对文件 `/etc/passwd` 进行读写：

```
fs.access('/etc/passwd', fs.R_OK | fs.W_OK, (err) => {
  console.log(err ? 'no access!' : 'can read/write');
});
```

fs.accessSync(path[, mode])

该方法是 `fs.access()` 的同步执行版本，当访问权限不足时会直接抛出错误。

fs.appendFile(file, data[, options], callback)

- `file`，字符串或数值，文件名或文件描述符
- `data`，字符串或 Buffer 实例
- `options`，对象或字符串
  - `encoding`，字符串或 null，默认值为 `utf-8`
  - `mode`，数值，默认值为 `0o666`
  - `flag`，字符串，默认值为 `a`
- `callback`，函数

该方法以同步的方式向指定文件添加数据，如果文件不存在则先创建该文件再添加内容。`data` 可以是一个字符串或 Buffer 实例：

```
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，则用来指定编码格式：

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback)
;
```

任何指定的文件描述符必须保持打开用于添加数据。注意，指定的文件描述符不能自动关闭。

fs.appendFileSync(file, data[, options])

该方法是 `fs.appendFile()` 的同步执行版本，返回 `undefined` 。

fs.chmod(path, mode, callback)

该方法是 `chmod(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.chmodSync(path, mode)

该方法是 `chmod(2)` 的同步执行版本，返回 `undefined` 。

fs.chown(path, uid, gid, callback)

该方法是 `chown(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.chownSync(path, uid, gid)

该方法是 `chown(2)` 的同步执行版本，返回 `undefined` 。

fs.close(fd, callback)

该方法是 `close(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.closeSync(fd)

该方法是 `close(2)` 的同步执行版本，返回 `undefined` 。

fs.createReadStream(path[, options])

该方法返回一个新建的 ReadStream 对象。

注意，虽然默认情况下一个可读的 Stream 实例的大小由 `highWaterMark`（16 kb）决定，但该方法返回的 Stream 对象的默认大小为 64 kb。

`options` 参数是一个对象或字符串，默认值为：

```
{
  flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

`options` 可以包含 `start` 和 `end` 两个属性，用于从文件中读取指定范围内的内容而不是读取文件的内容。`encoding` 参数可以是任何 Buffer 接收的编码格式。

如果指定了 `fd`，`ReadStream` 将会忽略 `path` 参数并使用该文件描述符，这意味着将不会触发 `open` 事件。注意，这里的 `fd` 应该可以阻塞进程，非阻塞的 `fd` 应该使用 `net.Socket` 处理。

如果 `autoClose` 的值为 false，那么文件描述符就不会被自动关闭，即使执行过程中出现错误也不会关闭。开发者有责任去管理文件描述的关闭和开启，并确保没有遗漏任何文件描述符。`autoClose` 的默认值为 true，当出现 `error` 或读到 `end` 时都会自动关闭文件描述符。

`mode` 用于设置文件模式（权限和粘滞位），但只对已创建的文件有效。

下面代码演示了读取 100 bytes 文件的最后 10 bytes：

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

如果 `options` 参数是一个字符串，则用于指定编码格式。

fs.createWriteStream(path[, options])

该方法返回一个 `WriteStream` 对象。

`options` 是一个对象或字符串，默认值为：

```
{
  flags: 'w',
  defaultEncoding: 'utf8',
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

`options` 中可以包含一个 `start` 属性，用于指定数据写入的起始位置。当修改文件的内容时，需要设置 `flag` 为 `r+` ，而不是使用默认的模式 `w` 。`defaultEncoding` 可以是任何 Buffer 接收的编码格式。

如果指定了 `fd` ，`WriteStream` 将会忽略 `path` 参数并使用该文件描述符，这意味着将不会触发 `open` 事件。注意，这里的 `fd` 应该可以阻塞进程，非阻塞的 `fd` 应该使用 `net.Socket` 处理。

如果 `autoClose` 的值为 false，那么文件描述符就不会被自动关闭，即使执行过程中出现错误也不会关闭。开发者有责任去管理文件描述的关闭和开启，并确保没有遗漏任何文件描述符。`autoClose` 的默认值为 true，当出现 `error` 或读到 `end` 时都会自动关闭文件描述符。

如果 `options` 参数是一个字符串，则用于指定编码格式。

fs.exists(path, callback)

接口稳定性: 0 - 已过时

使用 fs.stat() 后 fs.access() 代替。

fs.existsSync(path)

接口稳定性: 0 - 已过时

使用 fs.statSync() 或 fs.accessSync() 代替。

fs.fchmod(fd, mode, callback)

该方法是 `fchmod(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.fchmodSync(fd, mode)

该方法是 `fchmod(2)` 的同步执行版本，返回 `undefined` 。

fs.fchown(fd, uid, gid, callback)

该方法是 `fchown(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.fchownSync(fd, uid, gid)

该方法是 `fchown(2)` 的同步执行版本，返回 `undefined` 。

fs.fdatasync(fd, callback)

该方法是 `fdatasync(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.fdatasyncSync(fd)

该方法是 `fdatasync(2)` 的同步执行版本，返回 `undefined` 。

fs.fstat(fd, mode, callback)

该方法是 `fstat(2)` 的异步执行版本，其回调函数接收两个参数 `(err, stats)` ，其中 `stat` 是一个 `fs.Stats` 对象。 `fstat()` 等同于 stat()，唯一不同的这里的文件可以由文件描述符 `fd` 指定。

fs.fstatSync(fd)

该方法是 `fstat(2)` 的同步执行版本，返回 `fs.Stats` 的实例。

fs.fsync(fd, callback)

该方法是 `fsync(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.fsyncSync(fd)

该方法是 `fsync(2)` 的同步执行版本，返回 `undefined` 。

fs.ftruncate(fd, len, callback)

该方法是 `ftruncate(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.ftruncateSync(fd, len)

该方法是 `ftruncate(2)` 的同步执行版本，返回 `undefined` 。

fs.futimes(fd, callback)

该方法根据传入的文件描述符 `fd` 修改文件的时间戳。

fs.futimesSync(fd)

该方法是 `fs.futimesSync` 的同步执行版本，返回 `undefined` 。

fs.lchmod(path, mode, callback)

该方法是 `lchmod(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

该方法只在 Mac OS X 下可用。

fs.lchmodSync(path, mode)

该方法是 `lchmod(2)` 的同步执行版本，返回 `undefined` 。

fs.link(scrpath, dstpath, callback)

该方法是 `link(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。

fs.linkSync(scrpath, dstpath)

该方法是 `link(2)` 的同步执行版本，返回 `undefined` 。

fs.lstat(path, callback)

该方法是 `lstat(2)` 的异步执行版本，其回调函数接收两个参数 `(err, stats)` ，其中 `stat` 是一个 `fs.Stats` 对象。 `lstat()` 等同于 stat()，唯一不同的是，当 `path` 是一个软链接时，显示的是软链接的状态而不是对应文件的状态。

fs.lstatSync(path)

该方法是 `lstat(2)` 的同步执行版本，返回 `fs.Stats` 的实例。

fs.mkdir(path[, mode], callback)

该方法是 `mkdir(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数，其中 `mode` 的默认值为 `0o777` 。

fs.mkdirSync(path[, mode])

该方法是 `mkdir(2)` 的同步执行版本，返回 `undefined` 。

fs.open(path, flags[, mode], callback)

该方法是 `open(2)` 的异步执行版本，其中 `flags` 包含以下可选值：

- `'r'` ，以只读模式打开，如果文件不存在则抛出错误
- `'r+'` ，以读写模式代开，如果文件不存在则抛出错误
- `'rs'` ，以同步只读模式打开，通知操作系统忽略本地文件系统的缓存。这一功能常用于打开 NFS 挂载的文件，从而忽略可能已失效的文件缓存。同时，这一功能会严重影响 I/O 的性能，所以应该慎用该标志。注意，这里的 `fs.open()` 不是同步阻塞方法，如果你想使用同步执行的方法，请使用 `fs.openSync()` 。
- `'rs+'` ，以同步读写模式打开文件
- `'w'` ，以只写模式打开文件，如果文件不存在则创建该文件，如果文件存在则覆盖该文件
- `'wx'` ，类似 'w'，但如果路径不存在则抛出错误
- `'w+'` ，以读写模式打开文件，如果文件不存在则创建该文件，如果文件存在则覆盖该文件
- `'wx+'` ，类似 'w+'，但如果路径不存在则抛出错误
- `'a'` ，以追加数据的模式打开文件，如果文件不存在则创建该文件
- `'ax'` ，类似 'a'，但如果路径不存在则抛出错误
- `'a+'` ，以读写模式打开文件，如果文件不存在则创建该文件
- `'ax+'` ，类似 'a+'，但如果路径不存在则抛出错误

`mode` 参数用于配置文件的权限和粘滞位，且只在文件存在时有效。该参数的默认值为 `0666` ，表示可读写。

回调函数接收两个参数 `(err, fd)` 。

排异标志 `x`（对应 open(2) 中的 `O_EXCL`）用于确保 `path` 是新建的。在 POSIX 系统中，即使 `path` 是一个软链接且指向不存在的文件，也会被视为存在。该排异标志尚不能确定在网络文件系统是否有效。

`flags` 可以是 `open(2)` 指定的数值。常用的变量可以通过 `require('constants')` 获得。在 Windows 系统中，标志会被转换为 CreateFileW 接受的等效标志，比如，`O_WRONLY` 到 `FILE_GENERIC_WRITE`, or `O_EXCL|O_CREAT` 到 `CREATE_NEW`。

在 Linux 系统中，当文件以添加模式打开时，不能指定写入的位置。kernal 会忽略位置参数，并总是将数据添加到文件的尾部。

fs.openSync(path, flags[, mode])

该方法是 `fs.open()` 的同步执行版本，返回一个表示文件描述符的数值。

fs.read(fd, buffer, offset, length, position, callback)

该方法根据 `fd` 从文件中读取数据。

`buffer` 是一个 Buffer 实例，用于存储从文件中读取到的数据。

`offset` 表示向 Buffer 实例写入数据的起始位置。

`length` 是一个数值，用于表示读取的字节量。

`position` 是一个数值，用于表示文件读取的起始位置，如果该值为 `null`，则从文件的当前位置开始读取。

`callback` 是一个回调函数，接收三个参数 `(err, bytesRead, buffer)`。

fs.readdir(path, callback)

该方法是 `readdir(3)` 的异步执行版本，用于读取一个目录的内容。`callback` 接收两个参数 `(err, files)`，其中 `files` 是一个数组，数组成员为当前目录下的文件名，不包含 `.` 和 `..`。

fs.readdirSync(path)

该方法是 `readdir(3)` 的同步执行版本，返回一个不包含 `.` 和 `..` 的文件名数组。

fs.readFile(file[, options], callback)

- `file`，字符串或数值，文件名或文件描述符
- `options`，对象或字符串
  - `encoding`，字符串或 null，默认值为 null
  - `flag`，字符串，默认值为 `r`
- `callback`，函数

该方法以异步方式读取文件的整个内容：

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

回调函数接收两个参数 `(err, data)`，其中 `data` 是文件的内容。

如果未指定 `encoding`，则返回原始的 Buffer 数据。

如果 `options` 是一个字符串，则该字符串表示编码格式：

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

任何指定的文件描述符必须支持可读模式。注意，指定的文件描述符不会自动关闭。

fs.readFileSync(file[, options])

该方法是 `fs.readFile` 的同步执行版本，返回 `file` 的内容。

如果指定了可选参数 `encoding`，则该方法返回一个字符串，否则返回 Buffer 数据。

fs.readlink(path, callback)

该方法是 `readlink(2)` 的异步执行版本，其中回调函数接收两个参数 `(err, linkString)` 。

fs.readlinkSync(path)

该方法是 `readlink(2)` 的同步执行版本，以字符串形式返回软链接的值。

fs.realpath(path[, cache], callback)

该方法是 `realpath(2)` 的异步执行版本，其中 `callback` 接收两个参数 `(err, resolvedPath)` 。可以使用 `process.cwd` 解析相对路径。 `cache` 是一个映射路径的字面量，常用于强制解析路径或避免对已知真实路径调用额外的 `fs.stat` ：

```
var cache = {'/etc':'/private/etc'};
fs.realpath('/etc/passwd', cache, (err, resolvedPath) => {
  if (err) throw err;
  console.log(resolvedPath);
});
```

fs.readSync(fd, buffer, offset, length, position)

该方法是 `fs.read()` 的同步执行版本，返回 `bytesRead` 的数值。

fs.realpathSync(path[, cache])

该方法是 `realpath(2)` 的同步执行版本，返回解析后的路径。 `cache` 是一个映射路径的字面量，常用于强制解析路径或避免对已知真实路径调用额外的 `fs.stat` 。

fs.rename(oldPath, newPath, callback)

该方法是 `rename(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数

fs.rename(oldPath, newPath)

该方法是 `rename(2)` 的同步执行版本，返回 `undefined` 。

fs.rmdir(path, callback)

该方法是 `rmdir(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数

fs.rmdir(path)

该方法是 `rmdir(2)` 的同步执行版本，返回 `undefined` 。

fs.stat(path, callback)

该方法是 `stat(2)` 的异步执行版本，其回调函数接收两个参数 `(err, stats)` ，其中 `stats` 是一个 `fs.Stats` 对象。更多信息请参考 fs.Stats。

fs.statSync(path)

该方法是 `stat(2)` 的同步执行版本，返回一个 `fs.Stats` 的实例。

fs.symlink(target, path[, type], callback)

该方法是 `symlink(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。 `type` 的只可以是 'dir'、'file' 或 'junction'，默认值为 'file'，且该参数只适用于 Windows 系统，在其他平台会被忽略。注意，Windows 连接点必须是绝对路径。当使用 `junction` 时， `target` 参数会被自动解析为绝对路径：

```
fs.symlink('./foo', './new-port');
```

上面代码创建了一个名为 `new-port` 指向 `foo` 的软链接。

fs.symlinkSync(target, path[, type])

该方法是 `symlink(2)` 的同步执行版本，返回 `undefined` 。

fs.truncate(path, len, callback)

该方法是 `truncate(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数。当第一个参数是文件描述符时，系统调用 `fs.ftruncate()` 。

fs.truncateSync(path, len)

该方法是 `truncate(2)` 的同步执行版本，返回 `undefined` 。

fs.unlink(path, callback)

该方法是 `unlink(2)` 的异步执行版本，其回调函数只接收一个引用错误信息的参数

fs.unlink(path)

该方法是 `unlink(2)` 的同步执行版本，返回 `undefined` 。

fs.unwatchFile(filename[, listener])

该方法用于停止监控 `filename` 的修改。如果指定了 `listener` ，则值移除该 `listener` ，否则则移除所有的 `listener` ，完全地停止监控 `filename` 的修改。

调用 `fs.unwatchFile()` 停止监视未被监视的文件不会触发错误，本质上只是一个无效操作。

注意， `fs.watch()` 比 `fs.watchFile()` 和 `fs.unwatchFile()` 更加高效。应尽可能使用 `fs.watch()` 替代 `fs.watchFile()` 和 `fs.unwatchFile()` 。

fs.utimes(path, atime, mtime, callback)

该方法根据指定的 `path` 修改文件的时间戳。

注意，参数 `atime` 和 `mtime` 遵循以下规则：

- 如果它们的值是字符串类型的数值，比如 '123456789'，那么会被自动转换为数值类型
- 如果它们的值是 `NaN` 或 `Infinity`，那么它们就会被 `Date.now()` 的值覆盖。

fs.utimesSync(path, atime, mtime)

该方法是 `fs.utimes()` 的同步执行版本，返回 `undefined`。

fs.watch(filename[, options][, listener])

该方法用于监视 `filename` 的变动，其中 `filename` 可以是一个文件也可以是一个目录，返回值为 `fs.FSWatcher` 对象。

该方法的第二个参数 `options` 为对象类型的可选参数，该对象包含两个属性 `persistent` 和 `recursive`，均为布尔值。`persistent` 用于指定是否让进程持续运行。`recursive` 用于指定是监控所有的子目录还是之间空当前目录。该参数只有当指定的目标是目录时才会生效，且只支持部分平台（详见下文的 Caveats）。

`options` 的默认值是是 `{ persistent: true, recursive: false }`。

用作监听器的回调函数接收两个参数 `(event, filename)`，其中 `event` 要么是 `rename` 要么就是 `change`，`filename` 则是触发事件的文件名。

Caveats

`fs.watch` 方法在不同的平台上的表现并不一致，甚至在某些情况下是不可用的。

其中递归操作只适用于 OS X 和 Windows 系统。

Availability

这些特性依赖于操作系统底层提供的文件变动的通知接口：

- 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 系统中（包括 Solaris 和 SmartOS），使用 `event ports`
- 在 Windows 系统中，该特性取决于 `ReadDirectoryChangesW`

如果因为某些原因导致底层接口不可用，那么 `fs.watch` 也将不可用。举例来说，在网络文件系统中（NFS/SMB 等），对文件变动的监视往往收效甚微。

开发者仍可使用 `fs.watchFile`，该方法使用了状态轮询，所以较慢和不可靠。

文件名参数

回调函数接收的 `filename` 参数只在 Linux 和 Windows 系统中有效。即使在支持该参数的平台上，也无法完全保证可以获得 `filename`。因此，永远不要假想一定可以获得 `filename` 属性，并且为 `filename` 为 null 的情况设置降级措施：

```
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)

该方法用于监视 `filename` 的变动，每当 `filename` 被访问时都会调用一次回调函数 `listener`。

可选参数 `options` 是一个对象，可以忽略。`options` 可以包含一个布尔值属性 `persistent`，用于指定是否让进程持续运行。`options` 还可以包含一个 `interval` 属性，用于指定轮询周期。`options` 的默认值是 `{ persistent: true, interval: 5007 }`。

回调函数 `listener` 接收两个参数，分别是当前状态和先前状态：

```
fs.watchFile('message.text', (curr, prev) => {
  console.log(`the current mtime is: ${curr.mtime}`);
  console.log(`the previous mtime was: ${prev.mtime}`);
});
```

这里的状态对象都属于 `fs.Stat` 的实例。

如果你想同时监听文件访问和文件修改事件，可以通过比较 `curr.mtime` 和 `prev.mtime` 来实现。

注意，当 `fs.watchFile` 抛出 `ENOENT` 错误时，那么就会只调用一次监听器，且所有字段被重置为 0。在 Windows 系统中，`blksize` 和 `blocks` 会被赋值为 `undefined`，而不是 0。如果随后创建了文件，那么就会再次调用监听器设置状态对象。Node.js v0.10 之后支持该功能。

此外，值得注意的是，`fs.watch()` 比 `fs.watchFile()` 和 `fs.unwatchFile()` 更加高效。应尽可能使用 `fs.watch()` 替代 `fs.watchFile()` 和 `fs.unwatchFile()`。

fs.write(fd, buffer, offset, length[, position], callback)

该方法用于向指定的 `fd` 写入 Buffer 数据。

`offset` 和 `length` 用于提取 `buffer` 中指定位置的数据。

`position` 指定在文件中开始写入的位置。如果 `typeof position !== 'number'`，那么就会从当前位置开始写入，更多信息请参考 pwrite(2)。

回调函数 `callback` 接收三个参数 `(err, written, buffer)`，其中 `written` 用于指定写入的字节量。

注意，如果未执行完回调函数就多次执行 `fs.write` 是不安全的，对于这种需求，建议使用 `fs.createWriteStream` 。

在 Linux 系统中，当文件以追加模式打开时，不能指定写入的位置。kernel 会忽略位置参数，并总是将数据追加到文件的尾部。

fs.write(fd, data[, position[, encoding]], callback)

该方法用于向指定的 `fd` 写入 `data` 。如果 `data` 是一个 Buffer 实例，则内容会被强制转换为字符串格式。

`position` 指定在文件中开始写入的位置。如果 `typeof position !== 'number'` ，那么就会从当前位置开始写入，更多信息请参考 pwrite(2)。

`encoding` 用于指定字符串的编码格式。

回调函数 `callback` 接收三个参数 `(err, written, buffer)` ，其中 `written` 用于指定写入的字节量。

与写入 `buffer` 数据不同的是，如果数据是字符串，则必须一次性全部写入，这是因为字节的偏移量和字符的偏移量是不同的。

注意，如果未执行完回调函数就多次执行 `fs.write` 是不安全的，对于这种需求，建议使用 `fs.createWriteStream` 。

在 Linux 系统中，当文件以追加模式打开时，不能指定写入的位置。kernel 会忽略位置参数，并总是将数据追加到文件的尾部。

fs.writeFile(file, data[, options], callback)

- `file` ，字符串或数值，传入的文件名或文件描述符
- `data` ，字符串或 Buffer 实例
- `options` ，对象或字符串
  - `encoding` ，字符串或 null，默认值为 `utf-8`
  - `mode` ，数值，默认值为 `0o66`
  - `flag` ，字符串，默认值为 `w`
- `callback` ，函数

该方法以异步执行的方式向文件中写入数据，如果文件已存在，则替换该文件。`data` 可以是字符串或 Buffer 实例。

如果 `data` 是 Buffer 实例，则 `encoding` 参数，该参数的默认值为 `utf8`。

```
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，则用于指定编码格式；

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

传入的文件描述符必须允许写入。

注意，如果未执行完回调函数就多次执行 `fs.write` 是不安全的，对于这种需求，建议使用 `fs.createWriteStream`。此外，还需注意指定的文件描述符并不会自动关闭。

fs.writeFileSync(file, data[, options])

该方法是 `fs.writeFile()` 的同步执行版本，返回 `undefined`。

fs.writeSync(fd, buffer, offset, length[, position])

fs.writeSync(fd, data[, position[, encoding]])

该方法是 `fs.write()` 的同步执行版本，返回一个数值，该数值表示成功写入的字节量。

全局对象

下面这些方法可以直接在模块中使用，值得注意的一点是，其中有一些对象是挂载在全局作用域中的，有一些则是挂载在具体的模块作用域中。

Class: Buffer

- 函数

用于处理二进制数据。

__dirname

- 字符串

当前脚本执行时所在的目录：

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

`__dirname` 实际上并不属于全局，而是存在于每一个模块之中。

__filename

- 字符串

代码所在文件的文件名，返回一个绝对路径。对主程序来说，这个值和命令行中使用的文件名未必一致。在模块内，该值为模块文件的路径。

下面代码是在 `/Users/mjr` 目录下使用 `node example.js` 启动的：

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

`__filename` 实际上并不属于全局，而是存在于每一个模块之中。

clearInterval(t)

取消使用 `setInterval()` 创建的定时器。

clearTimeout(t)

取消使用 `setTimeout()` 创建的定时器。

console

- 对象

用于输出数据到 stdout 和 stderr。

exports

该对象是对 `module.exports` 的引用，更多有关 `exports` 和 `module.exports` 的用法请参考本文档的 `module` 模块。

`exports` 实际上并不属于全局，而是存在于每一个模块之中。

global

- 对象，全局命名对象

在浏览器中，顶层作用域为全局作用域，这意味着在全局作用域定义一个变量的话，这个变量就自动变成了全局变量。在 Node.js 中有所不同的是，顶层作用于并不是全局作用域，在 Node.js 模块内声明的变量属于模块内部的局部变量。

module

- 对象

该对象是对当前模块的引用。`module.exports` 主要用来定义模块暴漏给外部的变量，便于其他模块通过 `require()` 获取这些变量。

`module` 实际上并不属于全局，而是存在于每一个模块之中。

process

- 对象

进程对象。

require()

- 函数

该方法用于加载模块。

require.cache

- 对象

该对象缓存请求到的模块，通过删除该对象的键值对，可以在下次 `require()` 模块时重新加载模块。

require.extensions

接口稳定性: 0 - 已过时

require.resolve()

使用内部的 `require()` 机制查找模块的位置，并不加载模块，只会返回模块的文件名。

setInterval(cb, ms)

参见 timer 模块的 setInterval()。

setTimeout(cb, ms)

参考 timer 模块的 setTimeout()。

HTTP

接口稳定性: 2 - 稳定

通过 `require('htt')` 加载 HTTP 模块可以创建和使用 HTTP 服务器和客户端。

Node.js 中的 HTTP 接口对协议有良好的支持，包括那些传统上比较难处理的特性，比如大规模、块编码信息。这些接口不会缓存完整的请求或响应，从而允许开发者使用数据流。

HTTP 的头信息类似如下所示：

```
{
    'content-length': '123',
    'content-type': 'text/plain',
    'connection': 'keep-alive',
    'host': 'mysite.com',
    'accept': '*/*'
}
```

头信息中的键都是小写的，值是不能修改的。

为了支持尽可能多的 HTTP 应用程序，Node.js 的 HTTP API 通常都是比较偏底层的。通常只能处理流和信息。它可以将信息解析成报文头或报文体，但不能直接报文头和报文体。

查看 message.headers 可以了解更多如何处理重复报文头的信息。

接收到的原始报文头会被保存在 `rawHeaders` 属性中，该属性通常是一个类似 `[key, value, key2, value2, ...]` 的数组，比如之前提到的头信息对象有可能包含如下所示的 `rasHeaders` ：

```
[
    'ConTent-Length', '123456',
    'content-LENGTH', '123',
    'content-type', 'text/plain',
    'CONNECTION', 'keep-alive',
    'Host', 'mysite.com',
    'accepT', '*/*'
]
```

Class: http.Agent

HTTP Agent 常用于 HTTP 客户端请求的 socket 池中。

HTTP Agent 默认为客户端的请求设置 `Connection:keep-alive` 。如果 socket 处于空闲状态，则 socket 将会自动被关闭，这也就是说，使用 `keep-alive` 简化了开发者对 Node.js 请求池的管理。

如果开发者决定使用 HTTP `keep-alive` ，可以通过

开发者可以通过创建 HTTP Agent 对象时修改初始化属性使用 HTTP `keep-alive` ，该对象将会复用空闲的 socket。此类对象都会被明确标记，防止它们常驻 Node.js 的进程。不过，通过 `destory()` 方法显式注销 `keep-alive` 的对象是一个更好的做法。

当 socket 触发一个 `close` 或者 `agentRemove` 事件时，agent 池就会移除该 socket。这也就是说，如果开发者想让 socket 长时间运行，且在运行后自动关闭，那么就可以通过触发这两类事件实现：

```
http.get(options, (res) => {
  // Do stuff
}).on('socket', (socket) => {
  socket.emit('agentRemove');
});
```

此外，也可以通过设置 `agent: false` 不适用资源池：

```
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`，对象，用于配置 agent 对象的可选参数，该对象拥有以下属性
  - `keepAlive`，布尔值，决定允许其他请求复用 socket 池中空闲的 socket，默认值为 `false`
  - `keepAliveMsecs`，数值，当 `keepAlive` 为 true 时，该属性用于决定通过在线的 socket 发送 TCP KeepAlive 的频率，默认值为 1000
  - `macSockets`，数值，决定每个主机允许的最大 socket 量，默认值为 `Infinity`
  - `maxFreeSockets`，数值，当 `keepAlive` 为 true 时，该属性决定空闲状态下开启 socket 的最大量，默认值为 `256`

用于 `http.request()` 的 `http.globalAgent()` 在默认情况下会对每一个 agent 对象使用默认值。如果要设置其中某个对象的属性，必须创建自定义的 `http.Agent` 对象：

```
const http = require('http');
var keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.createConnection(optionsp[, callback])

该方法用于创建一个用于 HTTP 请求的 socket 或 stream 实例。

默认情况下，该方法和 `[net.createConnection()][]` 相同。但是，该方法可以通过修改 Agent 实例的默认配置，进而获得更适合的 Agent 对象。

有两种方式可以获取到 socket 或 stream 实例：一是通过该方法的返回值获取；一是从 `callback` 的参数获取。

`callabck` 接收两个参数 `(err, stream)`。

agent.destory()

该方法用与注销 Agent 实例所使用的全部 socket。

通产来说开发者无需显式注销 socket。不过，如果是 `keep-alive` 的 Agent 实例，那么最好由开发者显式注销 socket。因为对于此类 socket，除非服务器终端这些它们，否则它们就会一直存在。

agent.freeSockets

该属性是一个对象，包含了 `keep-alive` 的 Agent 实例所拥有的空闲 socket。不建议修改该属性。

agent.getName(options)

该方法为一组请求设置一个独一无二的标识符，用于标志一个链接是否可以复用。对于 http agent，该方法返回 `host:port:localAddress`；对于 https agent，该方法的返回值会包含 CA、cert、密钥和其他 HTTPS/TLS 所指定的 socket 复用配置项。

agent.maxFreeSockets

该属性的默认值为 256。对于 `keep-alive` 的 HTTP Agent 实例，该属性用于指定空闲状态下开启 socket 的最大量。

agent.maxSockets

该属性的默认值为 `Infinity`，用于指定同一源地址上 Agent 实例所能打开的 socket 的最大并发量。原地址可以是 `host:port` 或 `host:port:localAddress`。

agent.requrests

该属性是一个对象，包含尚未分配给 socket 的请求队列。不建议修改该属性。

agent.sockets

该属性是一个对象，包含 Agent 实例正在使用 socket。不建议修改该属性。

Class: http.ClientRequest

该对象由 `http.request()` 创建并作为返回值返回到外部。该对象用于表示一个头信息已加入处理队列的请求。虽然头信息已加入请求队列，但仍然可以通过 `setHeader(name, value)`、`getHeader(name)` 和 `removeHeader(name)` 来修改。当第一次发送数据或关闭连接时，头信息会随数据块一起发送出去。

通过监听 `response` 事件可以接收相应信息。当请求对象接收到响应头信息之后，就会触发 `response` 事件。`response` 事件接收一个参数，该参数是 `http.IncomingMessage` 的实例。

在 `response` 事件发生期间，可以给响应对象添加监听器，特别是用于监听 `data` 事件的监听器。

如果未对 `response` 事件设置处理函数，则响应信息就会被忽略。不过，如果添加了 `response` 事件的处理函数，则无论是使用 `response.read()`，或者添加 `data` 事件处理器，还是调用 `.resume()` 方法，都必须对响应对象的数据进行处理。只有数据经过处理之后，才会触发 `end` 事件。此外，除非读取数据，否则将会占用不必要的内容，直至触发 `process out of memory` 错误。

注意，Node.js 不会的对内容和消息体的长度进行比较。

该请求对象实现自 `Writable Stream` 接口，它是拥有可以触发如下事件 EventEmitter 实例：

事件：'abort'

- `function () {}`

当客户端取消请求时触发该事件，且该方法只在第一次调用 `abort()` 时触发。

事件：'checkExpectation'

- `function (request, response) {}`

每当接请求对象收到预期的 http 头消息（不包含 `Expect:100-continue` ）时就会触发该事件。如果没有监听该事件，服务器就会自动发送一个包含 417 预期失败的响应对象。

注意，当该事件被触发并处理之后，`request` 事件将不会再被触发。

事件：'connect'

- `function (response, socket, head) { }`

每次服务器通过 `CONNECT` 方法响应请求对象时，都会触发该事件。如果该事件未被监听，接收到 `CONNECT` 方法的客户端就会关闭连接。

下面代码演示了如何在客户端和服务器端监听 `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();
    });
  });
});
```

事件：'continue'

- `function () {}`

当请求对象包含 `Expect: 100-continue` 时，服务器通常会发送一个包含 `100 Continue` 的 HTTP 响应信息，此时就会触发该事件。该事件的作用是告诉客户端可以发送请求体了。

事件：'response'

- `function (response) {}`

当请求对象接收到响应信息时就会触发该事件，且只触发一次，其中 `response` 参数是 `http.IncomingMessage` 的实例。

可选配置项：

- `host`，请求的服务器域名或 IP 地址
- `port`，远程服务器的端口号
- `socketPath`，Unix 域名 Socket（`host:port` 或 socketPath）

事件：'socket'

- `function (socket) {}`

当一个 socket 被分配给请求兑现过之后就会触发该事件。

事件：'upgrade'

- `function (response, socket, head) {}`

每次服务器相应一个 upgrade 请求时就会触发该事件。如果为监听该事件，收到 upgrade 头信息的客户端就会关闭连接。

下面代码演示了如何在客户端和服务器端监听 `connect` 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);
  });
});
```

request.abort()

该方法用于终止请求。调用该方法将会使剩余的相应信息被丢弃且销毁 socket。

request.end([data][, encoding][, callback])

该方法用于结束发送请求，如果调用该方法时消息体的某些部分尚未发送，则这些部分将会被丢弃给 Stream 实例；如果请求是分块发送的，则调用该方法会直接发送终止块 `'0\r\n\r\n'`。

如果指定了 `data` 参数，相当于先调用了 `response.write(data, encoding)`，然后调用了 `request.end(callback)`。

如果指定了 `callback`，则当 request stream 结束时调用该回调函数。

request.flushHeaders()

该方法用于刷新请求头信息。

处于性能方面的原因，Node.js 通常会缓存请求头信息，除非开发者显式调用 `request.end()` 或写入第一个请求数据块。然后，系统就会将请求头信息和数据打包进一个 TCP 包。

虽然通常这已经满足了开发者的需求（可以节省一次 TCP 握手），但是发送第一块数据的时间可能会有点久。`request.flushHeaders()` 便于开发者忽略这些优化而发起请求。

request.setNoDelay([noDelay])

当请求对象分配到 socket 并连接之后就会调用 `socket.setNoDelay()` 方法。

request.setSocketKeepAlive([enable][, initialDelay])

当请求对象分配到 socket 并连接之后就会调用 `socket.setKeepAlive()` 方法。

request.setTimeout(timeout[, callback])

当请求对象分配到 socket 并连接之后就会调用 `socket.setTimeout()` 方法。

- `timeout`，数值，决定请求超时的时间
- `callback`，函数，请求超时后调用的回调函数

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

该方法用于发送请求体的数据块。多次调用该方法，可以从客户端向服务端发送请求体。对于发送请求体这种任务，建议创建请求对象时配置头信息 `['Transfer-Encoding', 'chunked']`。

`chunk` 参数可以是一个 Buffer 实例或字符串。

可选参数 `encoding` 只有当 `chunk` 为字符串时才可用，默认值为 `utf8`。

可选参数 `callback` 只有当数据被刷新掉时才会触发。

该函数的返回类型为 request 对象。

Class: http.Server

该类继承自 `net.Server`，可以触发以下事件：

事件：'checkContinue'

- `function (request, response) {}`

每当接请求对象收到 `Expect:100-continue` http 头消息时就会触发该事件。如果没有监听该事件，服务器就会自动发送一个包含 100 Continue 的响应对象。

如果客户端需要继续发送请求体或者生成适当的 HTTP 响应（比如 400，无效请求），则可以通过调用 `response.writeContinue()` 处理该事件。

注意，当该事件被触发并处理之后，`request` 事件将不会再被触发。

事件：'clientError'

- `function (exception, socket) {}`

如果客户端在连接过程中触发一个 `error` 事件，就会被转义到该事件上。

触发错误的 `socket` 对象是 `net.Socket` 的实例。

事件：'close'

- `function () {}`

当服务器关闭时触发该事件。

事件：'connect'

- `function (request, socket, head) {}`

每次客户端通过 `CONNECT` 方法发起请求时，都会触发该事件。如果该事件未被监听，接收到 `CONNECT` 方法的客户端就会关闭连接。

- `request` 参数是 request 事件中的请求对象
- `socket` 是服务器和客户端之间的网络套接字
- `head` 是一个 Buffer 实例，隧道 stream 的第一个数据块，值可能为空

触发该事件之后，请求的 socket 将不会给 `data` 事件绑定监听器，这也就是说开发者需要显式声明 `data` 事件的监听器，用于处理经 socket 发送给服务器的数据。

事件：'connection'

- `function (socket) {}`

新建 TCP 流时会触发该事件。`socket` 是一个 `net.Socket` 的实例。通常开发者并不会使用该事件，但有时候协议解析器绑定 socket 的方式，并不会让 socket 触发 `readable` 事件。`socket` 也可以通过 `request.connection` 方法获得。

事件：'request'

- `function (request, response) {}`

每次接收到请求对象时都会触发该事件。注意，每个链接可能有多个请求对象。`request` 是 `http.IncomingMessage` 的实例，`response` 是 `http.ServerResponse` 的实例。

事件：'upgrade'

- `function (request, socket, head) {}`

每次客户端请求 http upgrade 时就会触发该事件。如果为监听该事件，收到 upgrade 头信息的客户端就会关闭连接。

- `request` 参数是 request 事件中的请求对象
- `socket` 是服务器和客户端之间的网络套接字
- `head` 是一个 Buffer 实例，隧道 stream 的第一个数据块，值可能为空

触发该事件之后，请求的 socket 将不会给 `data` 事件绑定监听器，这也就是说开发者需要显式声明 `data` 事件的监听器，用于处理经 socket 发送给服务器的数据。

server.close([callback])

该方法用于终止服务器接收新的连接请求。

server.listen(handle[, callback])

- `handle` ，对象
- `callback` ，函数

`handle` 对象可以是一个 server、socket（任意以下划线开头的 `_handle` 成员）或 `{fd: <n>}` 对象。

使用该方法会锁定无法根据指定的 `handle` 接收连接，并默认为文件描述符或者 `handle` 已经绑定到了端口和域名 socket 上。

Windows 不支持对文件描述符的监听。

该方法是异步执行的方法，最后一个参数 `callback` 用于为 `listening` 时间添加监听器。

该方法的返回值是一个 `server` 对象。

server.listen(path[, callback])

给方法启动一个 UNIX socket 服务器，根据指定的 `path` 监听连接请求。

该方法是异步执行的方法，最后一个参数 `callback` 用于为 `listening` 时间添加监听器。

server.listen(port[, hostname][, backlog][, callback])

该方法根据指定的 `port` 和 `hostname` 接收连接。如果未指定 `hostname`，则当 IPv6 可用时，服务器会接受任意的 IPv6 地址，否则就接收任意的 IPv4 地址。如果端口值为 0，则随机分配一个端口号。

如果要监听 UNIX socket，则需要使用文件名，而不是端口或域名。

`backlog` 参数指定了等待连接队列的最大长度。该参数的实际值由操作系统通过 sysctl 配置项决定，比如 Linux 中的 `tcp_max_syn_backlog` 和 `somaxconn`。该参数的默认值为 `511`，而不是 522。

该方法是异步执行的方法，最后一个参数 `callback` 用于为 `listening` 时间添加监听器。

server.listening

该属性是一个布尔值，用于标识服务器是否在监听连接请求。

server.maxHeadersCount

该属性限制请求头的最大数量，默认值为 1000。如果值为 0，则表示不限制。

server.setTimeout(msecs, callback)

- `msecs`，数值
- `callback`，函数

该方法为 socket 设置超时时间，并触发 `timeout` 事件。如果触发了超时，则参数为 socket。

如果 server 对象监听了 `timeout` 事件并设置了监听器，那么超时时就会调用该监听器，且参数为超时的 socket。

默认你情况下，服务器的超时事件是两分钟，如果 socket 超时就会别自动销毁。不过，如果开发者为服务器的 `timeout` 事件设定了自定义的监听器，则由开发者负责销毁 socket。

server.timeout

- 数值，默认值为 120000（两分钟）

该属性指定 socket 处于空闲状态的超时时间。

注意，socket 超时的逻辑在连接时初始化，所以之后该属性的修改只对新创建的连接有效，不会影响现有的 socket。

如果该属性的值为 0，则对连接禁用任何类型的超时处理。

Class: http.ServerResponse

该对象由 HTTP 服务器创建，而不是由开发者创建。它常作为 `request` 事件的第二个参数。

该响应对象实现自 `Writable Stream` 接口，它是拥有可以触发如下事件 EventEmitter 实例：

事件：'close'

- `function () {}`

有两种行为会触发该事件：一是在调用 `response.end()` 之前中断底层连接；一是刷新数据。

事件：'finish'

- `function () {}`

响应信息发送之后会触发该事件。更准确地说，当响应信息的最后一部分被操作系统传输给网络时触发该事件。这并不表示客户端是否收到了数据。

该事件触发之后，响应对象不会再触发其他事件。

response.addTrailers(headers)

该方法用于给 response 对象添加 HTTP trailing header（信息尾部的头信息）。

只有响应对象使用了块编码时，才会触发 Trailer。如果没有使用块编码（比如使用 HTTP/1.0 的 request），则会被毫无声息地抛弃。

注意，如果要触发 Trailer，则 HTTP 需要发送 `Trailer` 头信息，该头信息包含以下属性和值：

```
response.writeHead(200, { 'Content-Type': 'text/plain','Trailer'
: 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': '7895bf4b8828b55ceaf47747b4
bca667'});
response.end();
```

如果修改头信息时使用了无效字符作为键值，则会导致系统抛出 TypeError。

response.end([data][, encoding][,callback])

该方法用于通知服务器所有的额响应信息已经发送完成。每一个响应对象都必须调用该方法。

如果指定了 `data`，则相当于先调用了 `resposne.write(data, encoding)`，然后调用了 `response.end(callback)`。

如果指定了 `callback`，那么在响应信息流结束后就会调用该回调函数。

response.finished

该属性是一个布尔值，用于标识响应过程是否结束，默认值为 `false`。执行 `response.end()` 之后，该值变为 `true`。

response.getHeader(name)

该方法用于读取已进入队列尚未发送给客户端的头信息。注意，这里的 `name` 需要注意大小写。该方法只能在头信息刷新之后调用。

```
var contentType = response.getHeader('content-type');
```

response.headerSent

该属性是一个只读的布尔值。如果头信息已被发送，则值为 true，否则为 false。

response.removeHeader(name)

该方法从头待发送队列中删除指定的头信息。

```
response.removeHeader('Content-Encoding');
```

response.sendDate

当该属性为 true 时，如果头信息没有日期，则会自动在响应信息中生成日期并发送出去，默认值为 true。

建议只在测试时禁用该属性，因为 HTTP 需要知道响应信息的

response.setHeader(name, value)

该方法用于配置头信息。如果该头信息即将被发送，则相应的值会被替换。如果你需要发送多个名字相同的头信息，可以使用字符串数组。

```
response.setHeader('Content-Type', 'text/html');

// or
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

如果修改头信息时使用了无效字符作为键值，则会导致系统抛出 TypeError。

使用 `response.setHeader()` 配置头信息之后，这些头信息会被其他头信息合并并传递给 `response.writeHead()`，但是传递给 `response.writeHead()` 的头信息优先级较高。

```
// 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.setTimeout(msecs,callback)

- `msecs`，数值

- `callback`，函数

该方法为 socket 设置超时时间。如果传入了回调函数，则该回调函数会被视为 `timeout` 事件的监听器。

如果请求对象、响应对象或服务器监听了 `timeout` 事件，那么 socket 超时时就会被销毁。如果开发者为服务器、请求对象或响应对象的 `timeout` 事件设定了自定义的监听器，则由开发者负责销毁 socket。

该方法的返回值是 `resposne` 对象。

response.statusCode

当使用隐式头信息（没有显式调用 `response.writeHead()` 的头信息）时，该属性决定了头信息刷新后发送给客户端的状态码。

```
response.statusCode = 404;
```

响应头信息发送给客户端之后，该属性表示已发送的状态码。

response.statusMessage

当使用隐式头信息（没有显式调用 `response.writeHead()` 的头信息）时，该属性决定了头信息刷新后发送给客户端的状态信息。如果该属性的值以 `undefined` 发送，则会使用标准的状态码的状态信息。

```
response.statusMessage = 'Not found';
```

响应头信息发送给客户端之后，该属性表示已发送的状态信息。

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

如果调用该方法之前没有调用 `response.writeHead()`，那么该方法将会切换到隐式头信息模式并刷新隐式头信息。

该方法用于发送一块响应体。该方法可以被多次调用，以保证响应体成功发送出去了。

`chunk` 可以是一个字符串或 Buffer 实例。如果 `chunk` 是一个字符串，那么第二个参数则指定了该字符串转化为字节码 stream 的编码格式。`encoding` 的默认值为 `utf8`。数据块刷新时会调用回调函数 `callback`。

注意，这是原始的 HTTP body，和 higher-level multi-part body 编码无关。

第一次调用 `response.write()` 时，将会发送缓存中的头信息和第一块 body 信息给客户端。第二次调用 `response.write()` 时，Node.js 将会默认独立发送流数据，也就是说，响应信息已经缓存在第一块 body 中了。

如果所有的数据已经成功刷新进了 kenel buffer，那么该方法返回 `true`；如果尚有数据存在于内存中，则该方法返回 `false`。当 buffer 为空时将会触发 `drain` 事件。

response.writeContinue()

该方法向客户端发送一个 HTTP1.1 100 Continue 的消息，用于指示应该发送请求体了。

response.writeHead(statusCode[, statusMessage][, headers])

该方法向请求对象发送一个响应头。状态码是一个三位的 HTTP 状态码，比如 404。最后一个参数 `headers` 是响应头信息。可选参数 `statusMessage` 是易于辨识状态信息。

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain' });
```

每次消息只能调用该方法一次，且必须在 `response.end()` 之前调用。

如果你在调用 `response.write()` 和 `response.end()` 之后调用该方法，那么就会产生隐式或可变的头信息。

使用 `response.setHeader()` 配置头信息之后，这些头信息会被其他头信息合并并传递给 `response.writeHead()`，但是传递给 `response.writeHead()` 的头信息优先级较高。

```
// 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');
});
```

注意，`content-length` 是字节数而不是字符数。上述代码之所以正常运行，是因为这里的 `hello world` 只是字节字符。如果 body 包含高级编码字符，就必须使用 `Buffer.byteLength()` 根据指定的编码格式计算字节数。此外，Node.js 并不会 content 和 body 的长度是否一致。

如果修改头信息时使用了无效字符作为键值，则会导致系统抛出 TypeError。

Class: http.IncomingMessage

`IncomingMessage` 对象可以由 `http.Server` 或 `http.ClientRequest` 创建，并且可以作为 `request` 和 `response` 事件监听器的第一个参数。它可以用于访问响应的状态、头信息和数据。

该对象实现了 `Readable Stream` 接口，具有以下事件、方法和属性：

事件：'close'

- `function () {}`

当底层链接关闭时触发该事件。和 `end` 事件一样，该事件对于每个响应对象只发生一次。

message.headers

该属性是包含请求或响应头信息的对象。

该属性内部以键值对的形式存储头信息，头信息的键名必须是小写：

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

根据头信息中的键名，对于重复的原始头信息分为以下多种处理方式：

- 重复的 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, or user-agent 会被丢弃
- `set-cookie` 是一个数组，重复的键值会被添加到该数组中
- 对于头信息其他键名的键值，使用 `,` 拼接在一起

message.httpVersion

该属性表示客户端支持的 HTTP 版本，通常为 '1.1' 或 '1.0'。

此外，也可以通过 `message.httpVersionMajor` 获取大版本号，通过 `message.httpVersionMinor` 获取小版本号。

message.method

该属性只对来自 `http.Server` 的请求有效。

请求方法是一个只读字符串，比如 'GET'、'DELETE'。

message.rawHeaders

该属性是一个接收到的原始请求或响应头信息的列表。

注意，键名和键值同处于该列表中，偶数索引为键名，奇数索引为键值。

头信息的键名不限制大小写，即使存在重复的键名也不会被合并。

```
// 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

该属性包含了接收到的原始请求或响应 trailer 的键值对，且只存在与 `end` 事件中。

message.setTimeout(msecs, callback)

- `msecs`，数值
- `callback`，函数

相当于调用 `message.connection.setTimeout(msecs, callback)`。

该方法的返回值是 `message` 对象。

message.statusCode

该属性只对来自 `http.ClientRequest` 的响应有效。

该属性的值是 3 位 HTTP 响应状态码，比如 `404`。

message.statusMessage

该属性只对来自 `http.ClientRequest` 的响应有效。

该属性的值用于表示 HTTP 响应状态信息，比如 `OK` 或 `Internal Server Error`。

message.socket

该属性表示与当前链接有关的 `net.Socket` 对象。

在 HTTPS 中，使用 `request.socket.getPeerCertificate()` 方法可以获取客户端的验证资料。

message.trailers

该属性包含了请求或响应的 trailer 对象，且只存在与 `end` 事件中。

message.url

该属性只对来自 `http.Server` 的请求有效。

该属性表示发起请求的 URL，值为字符串形式。该属性只包含白哦是实际 HTTP 请求的 URL，比如有如下所示的请求：

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

则 `request.url` 的值为：

```
'/status?name=ryan'
```

如果你想讲 URL 拆解开来，可以使用 `require(url).parse(request.url)` 来处理：

```
$ node
> require('url').parse('/status?name=ryan')
{
  href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status'
}
```

如果你想去除查询字符串中的参数，可以使用 `require('querystring').parse` 或给 `require('url').parse` 方法传递 `true` 参数：

```
$ node
> require('url').parse('/status?name=ryan', true)
{
  href: '/status?name=ryan',
  search: '?name=ryan',
  query: {name: 'ryan'},
  pathname: '/status'
}
```

http.METHODS

- 数组

该属性是一个数组，包含解析器所支持的 HTTP 方法。

http.STATUS_CODES

- 对象

该对象包含所有标准的 HTTP 响应状态码，以及相应的描述信息，比如 `http.STATUS_CODES[404] === 'Not Found'` 。

http.createClient([port][, host])

接口稳定性: 0 - 已过时

使用 http.request() 替代。

http.createServer([requestListener])

该方法返回一个 `http.Server` 新建的实例。

`requestListener` 是一个函数，将会被系统自动添加给 `request` 事件。

http.get(options[, callback])

由于大部分的请求都是没有 body 的 `GET` 请求，所以 Node.js 提供了这一方法。该方法与 `http.request()` 唯一的不同在于，它的默认 HTTP 方法就是 `GET`，并会自动调用 `req.end()`。

```
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.globalAgent

该属性是 Agent 的全局性实例，也是所有 HTTP 客户端请求的默认目标。

http.request(options[, callback])

Node.js 和每个服务器之间又存在链接，用于发送 HTTP 请求。开发者也可以使用该方法发出请求。

`options` 可以是对象或字符串。如果是字符串，则系统调用 `url.parse()` 解析该参数。`options` 包含的属性包括：

- `protocal`，使用的协议，默认值为 `http:`
- `hsot`，发出请求的服务器的域名或 IP 地址
- `hostname`，`host` 的别名，就 `url.parse()` 的支持度而言，`hostname` 要优于 `host`
- `family`，解析 host 或 hostname 时使用的 IP 地址簇。有效值为 4 和 6。如果未指定该参数，则 IPv4 和 IPv6 都可以使用。
- `port`，远程服务器的端口号，默认值为 80
- `localAddress`，绑定网络连接的本地接口
- `socketPath`，Unix Domain Socket（host:port 或 socketPath）
- `method`，字符串形式的 HTTP 请求方法，默认值为 `GET`
- `path`，请求路径，默认值为 `/`。如果存在查询字符串，则应该包含查询字符串，比如 `'/index.html?page=12'`。如果请求路径包含非法字符，则系统会抛出异常。目前，只有空格是不允许的，未来可能对此进行修改。
- `headers`，包含请求头信息的对象。

- `auth`，基本的验证信息，比如用于计算验证头信息的 `user:password`
- `agent`，用于控制 Agent 的行为。当使用 Agent 时，请求默认使用 `Connection: keep-alive`。可选值包括：
  - `undefined`，默认值，在这个主机和端口上使用 `http.globalAgent`
  - `Agent`，对象，在 `Agent` 中显式使用传入的参数
  - `false`，退出 Agent 连接池，默认请求 `Connection: close`
- `createConnection`，函数，当未使用 `agent` 参数时，该方法用于生成 request 需要的 socket 或 stream。该方法可以避免创建一个只为了重写 `createConnection` 函数的自定义的 Agent。

可选参数 `callback` 会被设为 `response` 事件的一次性监听器。

`http.request()` 方法返回一个 `http.ClientRequest` 的实例。`ClientRequest` 的实例是一个可写 stream。如果开发者需要使用 `POST` 方法上传文件，那么就可以将其写入到 `ClientRequest` 对象中。

```
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();
```

注意，在上面代码的最后调用了 `req.end()`。对于 `http.request` 方法，必须调用 `req.end()` 方法显式结束请求，即使未向请求体写入任何数据。

如果请求过程中出现了任何错误（DNS 解析、TCP 级别或 HTTP 解析），就会在返回的请求对象中触发一个 `error` 事件。对于所有的 `error` 事件，如果没有注册监听器，则该错误会抛出到全局。

下面是几个需要特别注意的头信息：

- 发送一个包含 `Connection: keep-alive` 的头信息会让 Node.js 保持服务器的链接，知道下次请求的到来
- 发送一个包含 `Content-length` 的头信息会替换默认的块编码格式
- 发送一个包含 `Expect` 的头信息会立即发送请求头。通常来说，如果发送的是 `Expect: 100-continue'` ，则开发者需要设置超时时间并监听 `continue` 事件。
- 发送一个包含验证信息的头信息会替代使用 `auth` 参数计算出来的验证信息

HTTPS

> 接口稳定性: 2 - 稳定

HTTPS 是一个基于 TLS/SSL 的 HTTP。在 Node.js 中，封装了一个单独的 HTTPS 模块。

Class: https.Agent

该类与 `http.Agent` 类似。

Class: https.Server

该类是 `tls.Server` 的子类，其可以触发的事件和 `http.Server` 相似。

server.setTimeout(msecs, callback)

该方法与 `http.Server#setTimeout()` 相似。

server.timeout

该方法与 `http.Server#timeout()` 相似。

https.createServer(options[, requestListener])

该方法返回一个新的 HTTPS web 服务器对象。`options` 参数与 `tls.createServer()` 中的 `options` 参数类似。`requestListener` 是一个参数，会被系统自动设为 `request` 事件的监听器。

```
// 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);

// or
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.close()

该方法与 `http.close()` 方法类似。

server.listen(handle[, callback])

server.listen(path[, callback])

server.listen(port[, host][, backlog][, callback])

上述方法与 `http.listen()` 方法类似。

https.get(options, callback)

该方法与 `http.get()` 方法类似，但只用于 HTTPS。

`options` 可以是对象或字符串。如果 `options` 是字符串，则系统自动调用 `url.parse()` 方法解析该参数。

```
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.globalAgent

该属性是全局性的 `https.Agent` 实例，是所有 HTTPS 客户端的请求目标。

https.request(options, callback)

该方法用于向安全的 web 服务器发送请求。

`options` 可以是对象或字符串。如果 `options` 是字符串，则系统自动调用 `url.parse()` 方法解析该参数。

所有能被 `http.request()` 接受的 `options` 对 `https.request()` 都是有效的。

```
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);
});
```

`options` 参数包含以下可选项：

- `hsot`，发出请求的服务器的域名或 IP 地址
- `hostname`，`host` 的别名，就 `url.parse()` 的支持度而言，`hostname` 要优于 `host`
- `family`，解析 host 或 hostname 时使用的 IP 地址簇。有效值为 4 和 6。如果未指定该参数，则 IPv4 和 IPv6 都可以使用。
- `port`，远程服务器的端口号，默认值为 80
- `localAddress`，绑定网络连接的本地接口
- `socketPath`，Unix Domain Socket（host:port 或 socketPath）
- `method`，字符串形式的 HTTP 请求方法，默认值为 `GET`
- `path`，请求路径，默认值为 `/`。如果存在查询字符串，则应该包含查询字符串，比如 `'/index.html?page=12'`。如果请求路径包含非法字符，则系统会抛出异常。目前，只有空格是不允许的，未来可能对此进行修改。
- `headers`，包含请求头信息的对象。

- `auth`，基本的验证信息，比如用于计算验证头信息的 `user:password`
- `agent`，用于控制 Agent 的行为。当使用 Agent 时，请求默认使用 `Connection: keep-alive`。可选值包括：
  - `undefined`，默认值，在这个主机和端口上使用 `http.globalAgent`
  - `Agent`，对象，在 `Agent` 中显式使用传入的参数
  - `false`，退出 Agent 连接池，默认请求 `Connection: close`

也可以指定下述来自 `tls.connect()` 的可选参数。不过，`globalAgent` 会忽略这些选项。

- `pfx`，SSL 使用的证书、私钥和 CA 证书，默认值为 `null`
- `key`，SSL 使用的私钥，默认值为 `null`
- `passphrase`，字符串，私钥或 pfx 的口令，默认值为 `null`
- `cert`，使用的公有 x509 证书，默认值为 `null`
- `ca`，字符串、Buffer 实例、字符串数组或具有 PEM 格式可信证书的 Buffer 实例。如果并不是这几种类型，则使用 "root" CA，比如 VeriSign。它们常用于授权链接。
- `ciphers`，使用或拒用的字符串密码，具体格式请参考 https://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT
- `rejectUnauthorized`，如果该值为 true，则使用提供的 CA 列表验证服务器证书，当验证失败时触发 `error` 事件。验证发生在 HTTP 请求发送之前的链接阶段。该属性的默认值为 true。
- `secureProtocal`，指定使用的 SSL 方法，比如 `SSLv3_method` 强制使用 SSL v3。具体的值取决于安装的 OpenSSL 和 `SSL_METHODS` 变量
- `servername`，SNI（Server Name INdication） TLS 扩展的服务器名字。

为了使用上述可选项，可以创建一个自定义的 `Agent`。

```
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) => {
  ...
}
```

或者不适用 `Agent` 退出连接池：

```
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) => {
  ...
}
```

Modules

接口稳定性: 3 - 已锁定

Node.js 内置了模块加载系统，文件和模块具有一一对应的关系。举例来说，`foo.js` 文件加载了同一目录下的 `circle.js` 模块，则在 `foo.js` 中可以向如下代码所示加载外部模块：

```
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` 模块向外暴漏了两个方法：area() 和 circuference()。如果要将对象或函数置于模块的顶层作用域，则可以将它们挂载在 `exports` 对象下。

模块的本地变量和内部封装的方法都是私有的，在上面的代码中变量 `PI` 就是 `circle.js` 私有的变量。

如果你只想从模块输出一个包含一切的函数或对象，那么可以使用 `module.exports` 而不是 `exports`。

在下面的代码中，`bar.js` 引用了 `square` 模块，该模块输出了一个构造函数：

```
const square = require('./square.js');
var mySquare = <a href="http://man7.org/linux/man-pages/man2/square.2.html">square(2)</a>;
console.log(`The area of my square is ${mySquare.area()}`);
```

`square` 模块定义在 `square.js` 中：

```
// assigning to exports will not modify module, must use module.exports
module.exports = (width) => {
  return {
    area: () => width * width
  };
}
```

`require('module')` 定义了 Node.js 的模块系统。

访问主模块

由 Node.js 直接访问的入口文件也被叫做主模块，此时 `require.main` 就等于该模块。这也即是说，你可以通过以下代码测试当前文件是否是主文件：

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

假设有一个 `foo.js` 文件，如果运行 `node foo.js`，那么上述代码就会返回 true；如果运行 `require('./foo')`，那么上述代码就会返回 false。

因为 `module` 提供了 `filename` 属性（通常等于 `__filename`），所以当前项目的入口文件可以通过 `require.main.filename` 获得。

包管理技巧

Node.js 内置的 `require()` 函数设计之初就是为了支持各种常规的目录结构。类似 dpkg/rpm/npm 的包管理器有助于开发者无需修改 Node.js 的模块即可构建本地的包。

下面我们将给出一个建设性的目录结构：

假设我们有一个文件夹 `/usr/lib/node/<some-package>/<some-version>` ，用于包含指定版本的包。

包之间可以相互依赖。为了安装包 `foo` ，开发者有可能需要安装指定版本的 `bar` 。 `bar` 又有可能有其他的依赖，这些依赖甚至会存在冲突或相互引用。

Node.js 首先会查找模块的 realpath（即遇到软链接会解析为真是链接），然后查找存储依赖的 `node_modules` 目录，具体的查找过程如下所以：

1. `/usr/lib/node/foo/1.2.3/` ，foo 包，版本为 1.2.3.
2. `/usr/lib/node/bar/4.3.2/` ，foo 的依赖包 bar
3. `/usr/lib/node/foo/1.2.3/node_modules/bar` ，软链接 `/usr/lib/node/bar/4.3.2/` 解析后获得真实地址
4. `/usr/lib/node/bar/4.3.2/node_modules/*` ，bar 所以来的包的软链接

因此，即使遇到循环引用，或者依赖冲突，每一个模块都能得到可用的特定版本的依赖。

当开发者在 `foo` 中使用 `require('bar')` 请求加载 `bar` 后，系统会解析 `bar` 的软链接，获取真实的地址 `/usr/lib/node/foo/1.2.3/node_modules/bar` 。然后如果在 `bar` 解析到了 `require('quux')` ，那么系统会继续解析软链接拿到真实路径 `/usr/lib/node/bar/4.3.2/node_modules/quux` 。

此外，为了优化模块查找效率，我们最好将模块置于 `/usr/lib/node_modules/<name>/<version>` 而不是直接置于 `/usr/lib/node` 。

为了在 Node.js 的 REPL 中使用模块，最好将 `/usr/lib/node_modules` 添加给换进变量 `$NODE_PATH` 。因为模块查找的 `node_modules` 文件夹使用的都是相对路径，且调用基于文件的真实路径执行 `require()` ，所以包实际上可以置于任意位置。

通过 `require()` 加载的模块可以通过 `require.resolve()` 函数获取模块的真实路径。

下面是用伪代码演示的 `require.resolve()` 解析过程:

```
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.  ST
OP
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
   c. DIR = path join(PARTS[0 .. I] + "node_modules")
   b. DIRS = DIRS + DIR
   c. let I = I - 1
5. return DIRS
```

缓存

系统第一次加载模块时会缓存这些模块，这也就是说，每次调用 `require('foo')` 都会得到相同的返回对象。

缓存有一个很重要的特性的，那就是多次调用 `require('foo')` 并不会让该模块多次运行。根据该特性，当模块返回结束对象之后，即使其他依赖存在循环引用也无所谓。

如果你想多次调用模块的某块代码，最好的方法是从该模块向外输出一个函数，在外部多次调用该函数。

模块缓存警告

缓存是基于模块的文件名进行解析的，所以如果调用位置不同，加载的模块就有可能不同，也就是说在不同的文件中，无法保证 `require('foo')` 每次都返回相同的对象。

此外，在对大小写敏感的操作系统中，不同的文件名有可能指向相同的文件，但缓存仍将其视为不同的模块，继而会多次重载该模块。比如，`require('./foo')` 和 `require('./FOO')` 会返回两个不同的对象，系统并不会检查 `./foo` 和 `./FOO` 是否会指向同一个文件。

核心模块

Node.js 内置了一些已经编译成二进制文件的模块，有关这些模块的详细介绍请参考本文的相应章节。

核心模块由 Node.js 源代码定义和实现，保存在 `lib/` 文件夹中。

`require()` 函数总是优先加载核心模块，举例来说，即使存在 `http` 文件，`require('http')` 也总会返回一个 Node.js 内建的 HTTP 模块。

循环引用

当 `require()` 出现循环引用时，引用的模块内部可能尚未执行完就返回了值。

我们假设有三个文件，其中 `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);
```

在上面的代码中，`main.js` 引用了 `a.js` ，`a.js` 引用了 `b.js` ，当系统继续解析 `b.js` 时，发现 `b.js` 有引用了 `a.js` 。为了避免无限循环引用，系统就会将一个 `a.js` 的不完全拷贝返回给 `b.js` 模块，然后 `b.js` 完成相应的解析，最后将 `exports` 对象提供给 `a.js` 模块。

`main.js` 加载这两个模块后完成相应的操作，输出结果如下：

```
$ 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
```

如果你的项目中存在模块的循环引用，建议据此解决。

文件模块

如果根据文件名没有找到模块，那么 Node.js 就会尝试使用不同的扩展名去加载模块，比如 `.js` 、 `.json` ，最后是 `.node` 。

`.js` 文件会被解析为 JavaScript 文本文件， `.json` 会被解析为 JSON 文本文件， `.node` 文件会被解析为被 `dlopen` 解析过的插件模块。

以 `/` 开头的路径为文件的绝对路径，举例来说，对于 `require('/home/marco/foo.js')` ，系统会查找 `/home/marco/foo.js` 。

以 `./` 开头的路径为文件的相对路径，表示相对于当前文件所在的目录，举例来说，对于 `foo.js` 文件中的 `require('./circle')` ，系统会在 `foo.js` 所在目录下查找 `cicle.js` 。

对于不以 `/` 、 `./` 或 `../` 开头的路径，系统会从核心模块或 `node_modules` 目录查找模块。

如果指定的路径不存在， `require()` 会抛出一个 Error 实例，该实例具有一个值为 `MODULE_NOT_FOUND` 的 `code` 属性。

文件夹即模块

将程序和依赖打包到同一个文件夹内并提供一个入口文件，是一种非常便捷的打包方式。这里有三种方式使用 `require()` 加载此类模块。

第一种方式是在根目录创建一个 `package.json` 文件，用于指定 `main` 模块：

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

如果该模块位于 `./some-library`，那么 `require('./some-libaray')` 就会尝试加载 `./some-library/lib/some-library.js`。

Node.js 可以正确解析 package.json 配置文件。

如果模块的根目录下没有 package.json，Node.js 就会尝试加载根目录下的 `index.js` 或 `index.node` 文件。举例来说，在某个模块的根目录下没有 package.json，那么 `require('./some-library')` 就会尝试加载：

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

从 node_module 加载模块

如果传给 `require()` 的不是一个原生模块，且不以 `/`、`./` 或 `../` 开头，那么 Node.js 就会从当前模块的父级目录查找 `node_modules`，并尝试从 `node_modules` 加载模块。

如果还是找不到，继续查找上一级目录，如此递归，直到找到或到达系统的根目录。

比如，如果在文件 `'/home/ry/projects/foo.js'` 中调用了 `require('bar.js')`，那么 Node.js 就会一次查找以下文件：

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

这种方式有助于限制依赖的作用范围，避免冲突。

开发者还可以通过添加后缀加载指定的文件或子模块，举例来说，`require('example-module/path/to/file')` 会加载 `example-module` 模块下的 `path/to/file` 文件。模块后缀的路径解析方式和上述模块路径的惊喜方式一致。

从全局文件夹加载

Node.js 如果在上述所有地方都找不到模块的话，就会检索环境变量 `NODE_PATH` 下是否存在，在大多数的系统中，`NODE_PATH` 中的路径以冒号分隔，而在 Windows 中，`NODE_PATH` 以逗号分隔。

创建环境变量 `NODE_PATH` 的本意是在上述的模块检索算法失效后检索更多的 Node.js 路径。

虽然现在 Node.js 还支持环境变量 `NODE_PATH`，但该变量已经越来越不重要了，这是因为 Node.js 圈约定俗成的使用本地存放依赖模块。如果模块和 `NODE_PATH` 绑定的话，会让那些不知道 `NODE_PATH` 的用户茫然不知所措。此外，当模块的依赖关系发生变化时，那么检索 `NODE_PATH` 就会加载不同版本的模块，甚至是与预期完全不同的模块。

除了 `NODE_PATH`，Node.js 还会检索以下位置：

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

这里的 `$HOME` 是用户的主页目录，`$PREFIX` 是 Node.js 配置的 `node_prefix`。

上述处理方式大都是由于历史原因形成的。强烈建议开发者将依赖存储于 `node_modules` 文件夹，这将有助于提高系统的解析速度，增强模块的稳定性。

module 对象

- 对象

在每一个模块中，变量 `module` 就是一个引用当前模块的对象。为了简便起见，可以使用 `exports` 替代 `module.exports`。`module` 实际上并不是一个全局对象，而是存在于每一个模块的本地变量。

module.children

- 数组

该属性的值包含了当前模块所加载的 module 对象。

module.exports

- 对象

`module.exports` 对象由模块系统所创建。很多开发者想要让模块是类的实例，那么就可以将期望的对象赋值给 `module.exports`。注意，如果赋值给了 `exports`，实际上只是简单地重新绑定到了本地的 `exports` 变量，这有可能会发生意料之外的结果。

举例来说，假设我们有一个模块 `a.js`：

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

module.exports = new EventEmitter();

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

在其他文件中加载该模块：

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

注意，`module.exports` 的赋值语句必须立即执行，而不能将其置于任何的回调函数之中。在下面的情况下，`module.exports` 的赋值操作是无效的：

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

加载 `x.js`：

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

exports 和 module.exports

模块中的变量 `exports` 最开始的时候是对 `module.exports` 的引用。如果开发者给 `exports` 替换成了对其他变量的引用，那么两者之间就没有任何关系了：

```
function require(...) {
  // ...
  ((module, exports) => {
    // Your module code here
    exports = some_func;        // re-assigns exports, exports i
s no longer
                                // a shortcut, and nothing is ex
ported.
    module.exports = some_func; // makes your module export 0
  })(module, module.exports);
  return module;
}
```

如果你无法理解 `exports` 和 `module.exports` 之间的关系，建议你忽略 `exports`，一切都使用 `module.exports`。

module.filename

- 字符串

该属性表示模块解析后的文件名。

module.id

- 字符串

该属性表示模块的标识符，通产来说就是模块解析后的文件名。

module.loaded

- 布尔值

该属性表示模块是已经加载完成还是处于加载过程中。

module.parent

- 对象，模块对象

该属性表示第一个加载该模块的文件。

module.require(id)

- `id`，字符串
- 返回值类型：对象，模块解析后返回的 `module.exports`

`module.require()` 方法提供了一种类似原始模块调用 `require()` 的模块加载方式。

注意，开发者必须获得 `module` 对象的引用才可以这么做。因为 `require()` 需要返回 `module.exports`，而 `module` 只代表特定的模块，所以必须显式导出 `module` 才能使用它。

net

> 接口稳定性: 2 - 稳定

`net` 模块封装了一系列的异步网络操作，常用于创建服务器和客户端（也被称为 stream）。通过 `require('net')` 可以引用该模块。

Class: net.Server

该类常用于创建 TCP 或本地服务器。

`net.Server` 是一个拥有以下事件的 `EventEmitter` 实例：

事件：'close'

当服务器关闭时触发该事件。注意，如果存在连接，那么直到连接关闭才会触发该事件。

事件：'connection'

- `net.Socket` 实例，连接对象

当创建新连接时触发该事件，其中 `socket` 是 `net.Socket` 的实例。

事件：'error'

- `Error` 实例

当出现错误时触发该事件。发生该事件之后会立即触发 `close` 事件。

事件：'listening'

当服务器通过 `server.listening` 绑定之后就会触发该事件。

server.address()

该方法返回一个对象，包含操作系统提供的绑定地址、地址族和服务器端口等信息。常用于查找那个端口已被系统绑定。该对象的常见格式：`{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`。

```
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);
});
```

在 `listening` 事件触发之前不要调用 `server.address()`。

server.close([callback])

该方法用于停止服务器接收新的连接请求并保持现有的连接。该方法是异步执行的，当服务器所有的连接都关闭之后就会触发 `close` 事件。可选参数 `callback` 在 `close` 事件触发后立即被调用。当服务器执行 `close` 事件时没有处于运行状态，那么就会生成一个错误传递给 `close` 事件的回调函数的第一个参数。

server.connections

接口稳定性: 0 - 已过时

使用 server.getConnections() 替代。

server.getConnections(callback)

该方法以异步执行的方法获取当前服务器的并发连接数，且只在 socket 发送给子进程之后才有效。

回调函数接受 `err` 和 `count` 两个参数。

server.listen(handle[, backlog][, callback])

- `handle`，对象
- `backlog`，数值
- `callback`，函数

`handle` 对象可以是一个 server、socket（拥有 `_handle` 属性）或 `{fd: <n>}` 对象。

该方法让服务器根据指定的 `handle` 接收连接，但是系统会假定文件描述符或 handle 已经绑定给了端口或域 socket。

在 Windows 平台上不支持对文件描述符的监听。

该方法是以异步方式执行的。当服务器绑定以后，就会触发 `listening` 事件。参数 `callback` 会被系统绑定为 `Listening` 事件的处理器。

参数 `backlog` 的行为与 `server.listen(port[, hostname][, backlog][, callback])` 中的 `backlog` 一致。

server.listen(options[, callback])

- `options`，对象，包含以下属性
  - `port`，数值
  - `host`，字符串
  - `backlog`，数值
  - `path`，字符串
  - `exclusive`，布尔值
- `callback`，函数

`options` 对象中的 `port`、`host` 和 `backlog` 属性，以及可选参数 `callback`，它们的行为和 `server.listen(port[, hostname][, backlog][, callback])` 中相应的属性一致。此外，`path` 可用于指定一个 UXIN socket。

如果 `exclusive` 的值为 false（默认值），cluster worker 将会使用相同的底层 socket 句柄，并允许连接处理共享的任务；如果 `exclusive` 的值为 true，则不允许共享句柄，即使共享端口也会抛出错误。

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

server.listen(path[, backlog][, callback])

- `path`，字符串
- `backlog`，数值
- `callback`，函数

该方法根据指定的 `paht` 创建一个本地的 socket 服务器用于监听连接请求。

该方法是以异步方式执行的。当服务器绑定以后，就会触发 `listening` 事件。参数 `callback` 会被系统绑定为 `Listening` 事件的处理器。

在 UNIX 系统上，本地域名通常是 UNIX 域名，路径是以文件的路径名。当使用该方法创建文件时需要遵循相关的命名约定和权限检查，并且在文件系统中可见，直到文件被取消关联。

在 Windows 系统上，本地域名使用命名管道实现，路径必须以 `\\?\pipe\` 或 `\\.\pipe\` 开头，支持任意字符，但是稍后系统会对管道名做一些处理，比如解析 `..` 序列。除去这些表象之外，管道命名空间实际上是扁平的。管道并不会持久存在，当对它们的最后一个引用关闭后，管道就会被移除。一定不要忘记 JavaScript 字符串转义需要使用两个反斜线指定路径：

```
net.createServer().listen(path.join('\\\\?\\pipe', process.cwd()
, 'myctl'))
```

参数 `backlog` 的行为和 `server.listen(port[, hostname][, backlog][, callback])` 方法中相应的参数一致。

server.listen(port[, hostname][, backlog][, callback])

该方法根据指定的 `port` 和 `hostname` 接收连接。当没有传入 `hostname` 时，如果 IPv6 可用，那么服务器就会接收任何来自 IPv6 地址（`::`）的连接，否则接收 IPv4 地址（`0.0.0.0`）。如果 `port` 的值为 0，则系统分配一个随机端

口。

`backlog` 参数指定连接等待队列的最大长度。该参数的实际值有操作系统的 sysctl 配置绝对，比如 Linux 上的 `tcp_max_syn_backlog` 和 `somaxconn` 。该参数的默认是 511，而不是 502。

该方法是以异步方式执行的。当服务器绑定以后，就会触发 `listening` 事件。参数 `callback` 会被系统绑定为 `Listening` 事件的处理器。

某些用户在运行时可能会遇到 `EADDRINUSE` 错误，这通常是由于其他运行中的服务器占用了请求的端口。一种处理方法就是等会再次尝试运行：

```
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 中的所有 socket 都已经被设置了 `SO_REUSEADDR` 。

server.listenning

该属性是一个布尔值，用于表示服务器是否正在监听连接请求。

server.maxConnections

设置该参数可以限制服务器的最大连接数。

对于 socket 向 `child_process.fork()` 创建的子进程发送消息这一情况，不建议设置该属性。

server.ref()

该方法是 `unref()` 方法的对立方法，在默认情况下，如果服务器已经调用了 `unref()` ，那么在此服务器上调用 `ref()` 并不会结束程序。此外，反复调用 `ref()` 并没有额外的效果。

该方法返回一个 server 实例。

server.unref()

如果某个服务器是事件系统中唯一存在的服务器，那么在该服务器上调用 `unref` 将不会允许程序退出。此外，反复调用 `unref()` 并没有额外的效果。

该方法返回一个 server 实例。

Class: net.Socket

该对象是对 TCP 或本地 socket 的抽象。`net.Socket` 的实例实现双工 Stream 接口。用户可以通过 `connect()` 方法创建该对象，将其视为一个客户端，或者由 Node.js 创建，由服务器的 `connection` 方法传递给用户。

new net.Socket([options])

该构造器用于创建一个初始化一个新的 socket 对象。

`options` 是一个包含以下默认值的对象：

```
{
    fd: null,
    allowHalfOpen: false,
    readable: false,
    writable: false
}
```

`fd` 用于指定现有的 socket 文件描述符。将 `readable` 和 `writeable` 设置为 `true`，将允许对 socket 的读写，注意，只有当传入 `fd` 时才有效。`allowHalfOpen` 参数和 `createServer()` 和 `end` 事件有关。

`net.Socket` 是拥有以下事件的 `EventEmitter` 实例：

事件：'close'

- `had_error`，布尔值，如果 socket 发生了传输错误，则该值为 `true`

当 socket 完全关闭时触发该事件，且只触发一次。参数 `had_error` 是一个布尔值，用于表示 socket 关闭时是否发生了传输错误。

事件：'connect'

当 socket 连接建立成功时触发该事件。

事件：'data'

- `Buffer` 实例

当接收到数据时触发该事件。参数 `data` 可以是 Buffer 实例或字符串，并通过 `socket.setEncoding()` 编码数据。

注意，如果 `Socket` 触发了 `data` 事件但没有给该事件设置监听器的时候，将会丢失数据。

事件：'drain'

当写入的 buffer 为空时触发该事件。该事件可用于控制上传。

其他信息请参考 `socket.write()` 的返回值。

事件：'end'

当 socket 的另一端发送 FIN 包时触发该事件。

默认情况下（ `allowHalfOpen == false` ），当 socket 将队列中的数据写完时就会销毁它的文件描述符。不过，如果设置 `allowHalfOpen == true` 了，则 socket 将不会自动调用 `end()` ，而是允许用户随意写入数据，但是最终要开发者自己调用 `end()` 事件。

事件：'error'

- `Error` 实例

当出现错误时就会触发该事件。该事件发生之后会立即触发 `close` 事件。

事件：'lookup'

在主机名解析之后、连接建立之前触发该事件，且不适用于 UNIX socket。

- `err`，Error 实例或 Null，错误对象
- `address`，字符串，IP 地址
- `family` 吗，字符串或 Null，地址类型

事件：'timeout'

如果 socket 的空闲时间超时，则触发该事件，该事件只用于提醒 socket 处于空闲状态。开发者必须手动关闭连接。

socket.address()

该方法返回一个对象，包含操作系统提供的绑定地址、地址族和服务器端口等信息。常用于查找那个端口已被系统绑定。该对象的常见格式：`{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`。

socket.bufferSize

该变量是 `net.Socket` 的一个属性，常用于 `socket.write()`，帮助用户获取更快的运行速度。由于网络连接有时候太慢，所以计算机不能一直高速向 socket 写入数据。Node.js 将排队数据写入 socket 并在网络可用时发送这些数据（轮询 socket 文件描述符直到变为可写）。

这种内部机制将会让内存占用增大，所以可以通过该属性控制缓存的字符数量（这里的字符串数量约等于准备写入的字节量，但在缓存中还有一些字符串，且这些字符串使用了惰性编码，所以实际的字节量是不可知的）。

如果开发者遇到了增长较快且较大的 `bufferSize`，那么应该尝试使用 `pause()` 和 `resume()` 方法对数据流进行节流操作。

socket.bytesRead

该属性表示接收到的字节量。

socket.bytesWritten

该属性表示发送出去的字节量。

socket.connect(options[, connectListener])

该方法为指定的 socket 打开连接。

对于 TCP socket，`options` 参数是一个对象，包含以下参数：

- `port`，客户端需要连接的端口
- `host`，客户端需要连接的主机，默认值为 `localhost`
- `localAddress`，为网络连接绑定的本地接口
- `localPort`，为网络连接绑定的本地端口
- `family`，IP 协议栈的版本，默认值为 `4`
- `lookup`，自定义的查找方法，默认值为 `dns.lookup`

对于本地域名的 socket，`options` 参数是一个对象，包含以下参数：

- `path`，客户端需要连接的路径。

通常来说用不到该方法，因为 `net.createConnection` 已经打开了 socket。只有开发者实现自定义的 Socket 时才需要使用该方法。

该方法是异步执行的。当 socket 建立之后就会触发 `connect` 事件。如果连接出现问题，就不会触发 `connect` 事件，而是触发 `error` 事件并抛出异常。

参数 `connectListener` 会被绑定为 `connect` 事件的监听器。

socket.connect(path[, connectListener])

socket.connect(port[, host][, connectListener])

该方法和 `socket.connect(options\[, connectListener\])` 类似，其中 `options` 可以是 `{port: port, host: host}` 或 `{path: path}`。

socket.destory()

该方法用于确保当前 socket 没有 I/O 活动，且只有在发生错误时才需要该方法。

socket.send([data][, encoding])

该方法用于半关闭 socket，比如发送 FIN 包。执行该方法后，socket 仍有可能发送数据。

如果指定了 `data`，那么该方法相当于先调用了 `socket.write(data, encoding)`，又调用了 `socket.end()`。

socket.localAddress

该属性是一个字符串，用于表示远程服务器连接到的本地 IP 地址。比如，如果你正在监听 `0.0.0.0`，而客户端链接到了 `192.168.1.1`，那么这个值就会变成 `192.168.1.1`。

socket.localPort

该属性是一个数值，表示本地端口，比如 `80` 和 `21`。

socket.pause()

该方法用于暂停读取数据，执行之后会触发 `data` 事件，常用于控制上传。

socket.ref()

该方法是 `unref()` 方法的对立方法，在默认情况下，如果服务器已经调用了 `unref()`，那么在此服务器上调用 `ref()` 并不会结束程序。此外，反复调用 `ref()` 并没有额外的效果。

该方法返回一个 socket 实例。

socket.remoteAddress

该属性是一个字符串，表示远程 IP 地址。比如 `'74.125.127.100'` 或 `'2001:4860:a005::68'`。如果 socket 已经被销毁，那么该属性的值可能是 undefined。

socket.remoteFamily

该属性是一个字符串，表示远程 IP 族，要么是 'IPv4'，要么就是 'IPv6'。

socket.resume()

该方法用户恢复调用了 `pause()` 方法的 socket 继续读取数据。

socket.setEncoding([encoding])

该方法用于设置作为可读 Stream 的 socket 的编码格式。

socket.setKeepAlive([enable][, initialDelay])

该方法用于开启或关闭 `keep-alive` 功能，其参数用于在存活探测分节（keepalive probe）第一次发送给空闲 socket 时，设置初始化的延迟时间。`enable` 的默认值为 `false`。

通过设置 `initialDelay` 可以控制接收到最后一个数据报和第一个存活探测分节之间的延迟时间。如果该参数的值wie 0，则系统会使用默认值或之前的值，默认值为 0。

该方法返回一个 socket 实例。

socket.setNoDelay([noDelay])

该方法用于禁用延迟传送算法。默认情况下，TCP 连接会使用延迟传送算法，会在数据发送之前缓存它们。如果 `noDelay` 的值为 `true`，则每次调用 `socket.write()` 时就会理解销毁数据。`noDelay` 的默认值为 `true`。

该方法返回一个 socket 实例。

socket.setTimeout(timeout[, callback])

该方法用于设置 socket 的空闲时间。默认情况下，`net.Socket` 没有超时限制。

当发生超时时，socket 会接收到一个 `timeout` 事件，但连接不会中断。开发者必须手动使用 `end()` 或 `destory()` 结束 socket。

如果 `timeout` 的值为 0，那么就会禁用闲置超时。

可选参数 `callback` 将会被绑定为 `timeout` 事件的监听器。

该方法返回一个 socket 实例。

socket.unref()

如果某个服务器是事件系统中唯一存在的服务器，那么在该服务器上调用 `unref` 将不会允许程序退出。此外，反复调用 `unref()` 并没有额外的效果。

该方法返回一个 socket 实例。

socket.write(data[, encoding][, callback])

该方法用于通过 socket 发送数据。第二个参数指定了字符串的编码格式，默认编码格式为 UTF8。

如果所有数据都成功刷新到了内核缓存中，该函数返回 `true`，如果所有的货部分数据加入到了内存中，则会返回 `false`。当缓存为空时，就会触发 `drain` 事件。

当所有数据都写完后，就会执行可选参数 `callback`，不过可能不是立即调用。

net.connect(options[, connectListener])

该方法是一个工厂方法，会返回一个新的 `new.Socket` 实例并自动根据 `options` 参数进行连接。

`options` 会被传送 `net.Socket` 的构造器和 `socket.connect` 方法。

`connectListener` 参数会被绑定为 `connect` 事件的监听器。

下面的代码演示了之前描述的 echo 服务器的客户端：

```
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');
});
```

如果要连接到 `/tmp/echo.sock`，只需要修改第二行：

```
const client = net.connect({path: '/tmp/echo.sock'});
```

net.connect(path[, connectListener])

该方法是一个工厂方法，会返回一个新的 `new.Socket` 实例并自动根据 `path` 参数进行连接。

`connectListener` 参数会被绑定为 `connect` 事件的监听器。

net.connect(port[, host][, connectListener])

该方法是一个工厂方法，会返回一个新的 `new.Socket` 实例并自动根据 `port` 和 `host` 参数进行连接。

如果未指定 `host` 参数，则会使用 `localhost`。

`connectListener` 参数会被绑定为 `connect` 事件的监听器。

net.createConnection(options[, connectListener])

该方法是一个工厂方法，会返回一个新的 `new.Socket` 实例并自动根据 `options` 参数进行连接。

`options` 会被传送 `net.Socket` 的构造器和 `socket.connect` 方法。

`connectListener` 参数会被绑定为 `connect` 事件的监听器。

下面的代码演示了之前描述的 echo 服务器的客户端：

```
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');
});
```

如果要连接到 `/tmp/echo.sock` ，只需要修改第二行：

```
const client = net.connect({path: '/tmp/echo.sock'});
```

net.createConnection(path[, connectListener])

该方法是一个工厂方法，会返回一个新的 `new.Socket` 实例并自动根据 `path` 参数进行连接。

`connectListener` 参数会被绑定为 `connect` 事件的监听器。

net.createConnection(port[, host][, connectListener])

该方法是一个工厂方法，会返回一个新的 `new.Socket` 实例并自动根据 `port` 和 `host` 参数进行连接。

如果未指定 `host` 参数，则会使用 `localhost` 。

`connectListener` 参数会被绑定为 `connect` 事件的监听器。

net.createServer([options][, connectionListener])

该方法用于创建新的服务器，其中，`connectListener` 参数会被绑定为 `connect` 事件的监听器。

`options` 参数是一个包含以下默认值的对象：

```
{
    allowHalfOpen: false,
    pauseOnConnect: false
}
```

如果 `allowHalfOpen === true` ，当 socket 的另一端发送了一个 FIN 包时，socket 不会自动发送 FIN 包。socket 会变成不可读但可写的状态。开发者需要显式调用 `end()` 。

如果 `allowHalfOpen === true` ，与 socket 有关的所有连接都会被暂停，它的 handle 也不会读取任何数据。它允许在原始进程不读取数据的情况下，让连接在进程之间传递。从暂停的 socket 读取数据之前需要先调用 `resume()` 。

下面代码演示了一个监听 8124 端口的 echo 服务器：

```
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');
});
```

通过 `talnet` 命令测试：

```
telnet localhost 8124
```

如果要监听 `/tmp/echo.sock` socket，只需修改最后三行代码：

```
server.listen('/tmp/echo.sock', () => {
  console.log('server bound');
});
```

使用 `nc` 连接到 UNIX 域名 socket 服务器：

```
nc -U /tmp/echo.sock
```

net.isIP(input)

该方法用于测试 `input` 是否是一个 IP 地址。如果 `input` 是无效的的字符串，则返回 0，如果是 IPv4，则返回 4，如果是 IPv6，则返回 6。

net.isIPv4(input)

如果 `input` 是 IPv4 的 IP 地址，则返回 true，否则返回 false。

net.isIPv6(input)

如果 `input` 是 IPv6 的 IP 地址，则返回 true，否则返回 false。

操作系统

接口稳定性: 2 - 稳定

该模块提供了一些和操作系统有关的辅助函数，使用 `require('os')` 可以加载该模块。

os.EOL

常量，返回当前操作系统的换行符。

os.arch()

返回当前操作系统的 CPU 架构，常见值包括： `x64` / `arm` 和 `ia32` ，实际上该值引用的是 `process.arch` 。

os.cpus()

返回一个对象数组，每一个对象描述一个 CPU 的数据：

下面是一个 `os.cpus()` 的返回信息：型号，速度和使用时间。

```
[ { 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 } } ]
```

注意，其中 `nice` 的值只对 UNIX 有效，在 Windows 中该值为 0。

os.endianness()

返回 CPU 的字节序，如果是大端字节序，返回 `BE`，如果是小端字节序，则返回 `LE`。

os.freemem()

返回系统空闲内存的大小，单位是字节。

os.homedir()

返回当前用户的根目录。

os.hostname()

返回当前操作系统的主机名。

os.loadavg()

返回一个数组，包含 1，5 和 15 分钟内的负载均衡信息。负载均衡在一定程度上反映了操作系统的活动情况，它的值有操作系统计算得出，通常数值较小。一般来说，负载均衡都会比操作系统中的 CPU 数量小。

负载均衡的概念仅存在于 UNIX 系统中，在 Windows 平台中，该值为 `[0, 0, 0]` 。

os.networkInterfaces()

返回网络接口的列表：

```
{ 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 } ] }
```

注意，由于底层实现的原因，该方法只会返回已分配地址的网络接口。

os.platform()

放回操作系统平台，常见值包括：`darwin` / `freebsd` / `linux` / `sunos` 和 `win32`，实际上该值引用的是 `process.platform()`。

os.release()

返回操作系统的分发版本号。

os.tmpdir()

返回操作系统默认的临时文件夹。

os.totalmem()

返回操作系统内存空间的大小，单位是字节。

os.type()

返回操作系统的名称，比如 Linux 的名称是 `Linux`，OS X 的名称是 `Darwin`，Windows 的名称是 `Windows_NT`。

os.uptime()

返回操作系统的运行时间，单位是秒。

Path

接口稳定性: 2 - 稳定

该模块封装了一些处理和转换文件路径的辅助函数，其中大多数主要用于对路径字符串的转换。文件系统不会去校验路径是否有效。

通过 `require('path')` 可以加载该模块。

path.basname(p[, ext])

该方法返回路径的最后一部分，类似于 Unix 中的 `basename` 命令：

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns 'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns 'quux'
```

path.delimiter

该方法根据当前平台特性返回路径定界符，比如 `;` 或 `:`。

下面代码演示了在 *nix 下的执行结果：

```
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/loca
l/bin']
```

下面代码演示了在 Windows 下的执行结果：

```
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.dirname(p)

该方法返回一个路径的目录名，类似于 Unix 的 `dirnaem` 命令：

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns '/foo/bar/baz/asdf'
```

path.extname(p)

该方法返回路径的扩展名，该扩展名为路径的最后一个 `.` 到结尾部分的字符串。如果路径最后一部分中没有 `.` ，或者第一个字符就是 `.` ，则返回一个空字符串：

```
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.format(pathObject)

该方法根据传入的 `pathObject` 返回一个字符串形式的路径，与之相对的方法是 `path.parse` ：

```
path.format({
    root : "/",
    dir : "/home/user/dir",
    base : "file.txt",
    ext : ".txt",
    name : "file"
})
// returns '/home/user/dir/file.txt'
```

path.isAbsolute(path)

该方法用于判断一个路径是否是绝对路径。绝对路径不需要依赖当前共走路径即可定位位置。

在 POSIX 系统下：

```
path.isAbsolute('/foo/bar') // true
path.isAbsolute('/baz/..')  // true
path.isAbsolute('qux/')     // false
path.isAbsolute('.')        // false
```

在 Windows 系统下：

```
path.isAbsolute('//server')  // true
path.isAbsolute('C:/foo/..') // true
path.isAbsolute('bar\\baz')  // false
path.isAbsolute('.')         // false
```

注意，如果传入的路径是一个空字符串，与其他 path 模块的方法所不同的是，该方法会返回 `false` 。

path.join([path1][, path2][, ...])

该方法拼接所有的参数，然后标准化出一个路径。

该方法中的参数必须是字符串。在 v0.8 中，非字符串参数会被忽略，但在 v0.10+ 中，会抛出错误。

```
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 st
rings
```

注意，如果传入的路径是一个空字符串，与其他 path 模块的方法所不同的是，该方法会返回 `.`（代表当前路径）。

path.normalize(p)

该方法用于标准化路径，注意 `..` 和 `.` 部分。

如果路径中存在多个斜线，则会被替换为单个斜线；如果路劲尾部存在斜线，则会被保留。在 Windows 中路径使用反斜线分隔。

```
path.normalize('/foo/bar//baz/asdf/quux/..')
// returns '/foo/bar/baz/asdf'
```

注意，如果传入的路径是一个空字符串，该方法会返回 `.`（代表当前路径）。

path.parse(pathString)

该方法根据字符串路径返回一个对象。

在 *nix 系统下：

```
path.parse('/home/user/dir/file.txt')
// returns
{
    root : "/",
    dir : "/home/user/dir",
    base : "file.txt",
    ext : ".txt",
    name : "file"
}
```

在 Windows 系统下：

```
path.parse('C:\\path\\dir\\index.html')
// returns
{
    root : "C:\\",
    dir : "C:\\path\\dir",
    base : "index.html",
    ext : ".html",
    name : "index"
}
```

path.posix

该属性决定是否以 POSX 兼容模式执行上述路径处理方法。

path.relative(from, to)

该方法根据 `from` 和 `to` 解析相对路径。

该方法根据传入的两个绝对路径，计算它们的相对路径。该方法实际上是 `path.resolve` 的逆方法：`path.resolve(from, path.relative(from, to)) == path.resolve(to)` 。

```
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，如果传入的路径存在空字符串，该空字符串会被 `.`（代表当前路径）代替。如果两个路径相同，则返回一个空字符串。

path.resolve([from...], to)

该方法将 `to` 解析为绝对路径。

如果 `to` 不是相对 `from` 的绝对路径，那么 `to` 就会被添加到 `from` 的右侧，直到找到一个而绝对路径。如果用尽所有的 `from` 都没有找到绝对路径，就会使用当前工作目录。最终的路径是标准化之后，已经去除了斜线，除非结果是根目录。非字符串的 `from` 参数会被忽略。

下面是一个该犯法的用例：

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

类似于：

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

不同之处在于，不同的路径可以不存在或者是文件。

```
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，如果传入的路径存在空字符串，该空字符串会被 `.`（代表当前路径）代替。

path.sep

该属性表示当前平台的文件分隔符是 '\' 还是 '/'。

在 *nix 系统下：

```
'foo/bar/baz'.split(path.sep)
// returns ['foo', 'bar', 'baz']
```

在 Windows 系统下：

```
'foo\\bar\\baz'.split(path.sep)
// returns ['foo', 'bar', 'baz']
```

path.win32

该属性决定是否以 win32 兼容模式执行上述路径处理方法。

process

`process` 是一个全局对象，也是 `EventEmitter` 的实例。

事件：'beforeExit'

当 Node.js 清空事件循环机制且没有其他调度任务时就会触发该事件。当没有调度任务时，Node.js 就会退出，但是 `beforeExit` 的事件监听器可以以异步的形式执行，从而保持 Node.js 持续运行。

`beforeExit` 事件并不是进程终端的条件，process.exit() 或者是未捕获的异常才是。`beforeExit` 事件不应该被当做 `exit` 事件，除非开发者需要大量的调度工作。

事件：'exit'

当进程将要退出时触发该事件。此时将无法停止事件循环的执行，一旦所有的 `exit` 事件监听器都执行完毕，那么进程就会退出，因此，开发者只应该使用同步执行的监听器。这是一个检查模块状态（比如单元测试）的好时机。回调函数接收一个参数，即进程的退出码。

该事件只有通过 `process.exit()` 显式结束 Node.js 或事件循环的内存耗尽时才会触发。

```
process.on('exit', (code) => {
  // do *NOT* do this
  setTimeout(() => {
    console.log('This will not run');
  }, 0);
  console.log('About to exit with code:', code);
});
```

事件：'message'

- `message`，对象，解析后的 JSON 对象或原始值

- `sendHandle`，Handle 对象，net.Socket 或 net.Server 对象，或 undefined

子进程对象可以通过监听 `message` 事件获取 `ChildProcess.send()` 发送的消息。

事件：'rejectionHandled'

当 Promise 被驳回且在事件循环开始后绑定了错误处理函数时，就会触发该事件。该事件包含以下参数：

- `p`，经 `unhandleRejection` 事件处理后绑定驳回函数的 Promise。

Promise 调用链中无法绝对肯定驳回会被处理。由于 Promise 天生就是异步的，所以驳回可以在未来的某个时间被处理，这个时间可能远大于事件循环机制将其移交给 `unhandledRejection` 事件的时间。

另一种处理该问题的方式是这样的，在同步执行的代码中总有一个持续增长的未处理异常列表，但在 Promise 中则有一个可伸缩的未处理异常列表。对于同步执行的代码，`uncaughtException` 事件可以通知开发者未处理异常列表在什么时间增大了；对于异步执行的代码，`unhandledRejection` 事件可以通知开发者未处理异常列表在什么时间增大了，而 `rejectionHandled` 事件则可以通知开发者未处理异常列表在什么时间缩小了。

下面代码演示了使用驳回检测方式记录指定时间内所有被驳回的 Promise 原因：

```
const unhandledRejections = new Map();
process.on('unhandledRejection', (reason, p) => {
  unhandledRejections.set(p, reason);
});
process.on('rejectionHandled', (p) => {
  unhandledRejections.delete(p);
});
```

该记录的内容会随着时间收缩不定，反映出何时发生驳回，何时驳回被处理。开发者可以使用错误日志记录周期性的（适合耗时应用）或在进程退出前（适合脚本）记录错误。

事件：'uncaughtException'

当异常传播到事件循环机制时就会触发该事件。默认情况下，Node.js 对于此类异常会直接将其堆栈跟踪信息输出给 stderr 并结束进程，而为 `uncaughtException` 事件添加一个事件监听器则可以覆盖该默认行为。

```
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.');
```

合理使用 'uncaughtException'

注意，`uncaughtException` 是一种处理异常的非常规手段，除非万不得已，不要使用它。该事件不能被认为等同于 `On Error Resume Next`。存在未处理的异常就意味着应用程序处于一种未知的状态。如果不能合理地根据异常信息会发应用程序的状态，那么很有可能会触发其他不可预见的问题。

事件处理器内部抛出的异常将不会被捕获，这些异常将会中断 Node.js 的进程并返回一个非零的退出码，最后输出相应的堆栈信息，这一机制有助于避免无限递归。

在系统抛出异常之后尝试恢复的过程很类似于在电脑升级系统的时候拔掉电源，虽然这么做十次中有九次都没有问题，但只要有一次有问题，系统就会挂掉。

正确使用 `uncaughtException` 的方式是在结束进程前使用同步执行的方法清理分配到的资源（文件描述符、句柄等）。在 `uncaughtException` 事件之后执行常规的恢复操作并不安全。

事件：'unhandledRejection'

当 `Promise` 被驳回且没有被绑定错误处理函数时，就会触发该事件。当 Promise 遇到异常时都要将其封装为异常 Promise，可以通过 `promise.catch(...)` 处理此类 Promise，驳回信息也会通过 Promise 调用链逐步向上传播。该事件对检测和

跟踪未被处理的驳回 Promise 的堆栈信息很有用。该事件接收以下参数：

- `reason`，携带 Promise 被驳回信息的对象，通常为 Error 实例
- `p` 被驳回的 Promise

下面代码演示了使用控制台记录所有未处理的驳回：

```
process.on('unhandledRejection', (reason, p) => {
    console.log("Unhandled Rejection at: Promise ", p, " reason:
 ", reason);
    // application specific logging, throwing an error, or other
 logic here
});
```

下面代码演示了一个会触发 `unhandledRejection` 事件的驳回逻辑：

```
somePromise.then((res) => {
  return reportToUser(JSON.pasre(res)); // note the typo (`pasre
`)
});
// no `.catch` or `.then`
```

下面代码是一个触发 `unhandledRejection` 事件的代码模式：

```
function SomeResource() {
  // Initially set the loaded status to a rejected promise
  this.loaded = Promise.reject(new Error('Resource not yet loade
d!'));
}

var resource = new SomeResource();
// no .catch or .then on resource.loaded for at least a turn
```

对于此类事件，如果你不想像开发错误一样追踪驳回信息，那么你就可以使用 `unhandledRejection` 事件。为了实现这一方案，开发者既可以给 `resource.loaded` 绑定 `.catch(() => {})`，避免触发 `unhandledRejection` 事件，也可以使用 `rejectionHandled` 事件。

退出码

如果没有异步操作正在执行，那么 Node.js 正常退出的退出码就是 0，否则则为以下情况：

- `1`，存在未被捕获的重大异常。该退出码表示存在未被捕获的异常，且没有被域名或 `uncaughtException` 事件监听器处理
- `2`，该退出码尚处于保留状态，Bash 使用该状态码表示内建的误用信息
- `3`，JavaScript 语法解析错误。该退出码表示 JavaScript 源代码在 Node.js 启动阶段发生了解析错误。这种错误很少见，且只在 Node.js 的开发阶段会被触发。
- `4`，JavaScript 执行错误。该退出码表示 JavaScript 在 Node.js 启动进程时的失败，失败时系统会返回一个函数值。这种错误很少见，且只在 Node.js 的开发阶段会被触发。
- `5`，重大错误。该退出码表示 V8 里出现的不可修复的重大错误。系统通常会向使用 stderr 发送以 `FATAL ERROR` 开头的消息。
- `6`，非函数的异常处理器。系统中存在未捕获的异常，但是内部的异常捕获器却不是函数类型，无法被调用。
- `7`，异常处理器运行失败。系统中存在未捕获的异常，但是内部的异常捕获器运行时抛出了异常。举例来说，`process.on('uncaughtException')` 或 `domain.on('error')` 处理器就会抛出此类错误。
- `8`，该退出码处于保留状态。在早先的 Node.js 中，退出码 8 表示一个未被捕获的异常。
- `9`，无效参数。该退出码表示传入了未知的参数或指定了无效的值。
- `10`，JavaScript 运行失败。该退出码表示 Node.js 的引导进程部分的 JavaScript 代码存在问题。这种错误很少见，且只在 Node.js 的开发阶段会被触发。
- `12`，无效的调试参数。设置了 `--debug` 或 `--debug-brk` 参数，但选择了错误的端口。
- `>128`，信号推出。如果 Node.js 接收到了 `SIGKILL` 或 `SIGHUP` 信号，那么它就会结束进程且退出码为 128 + 信号码。这是一个标准的 Unix 做法，因为退出码为低位 7 字节的整数，信号码高位 7 字节整数，所以它们的值大于 128。

信号事件

当继承接收到信号时就会触发此类事件。查看 `sigaction(2)` 可以获得标准的 POSIX 信号名，比如 `SIGINT` 、 `SIGHUP` 等。

下面代码演示了如何监听 `SIGINT` ：

```
// Start reading from stdin so we don't exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Got SIGINT.  Press Control-D to exit.');
});
```

在大多数的终端环境中，使用 `Control-C` 组合键是一个发送 `SIGINT` 的简单方式。

- `SIGUSER1` ，Node.js 使用该信号开启调试模式。在调试模式时可以设置监听器，且不会中断调试模式。
- `SIGTERM` 和 `SIGINT` ，在非 Windows 平台上有默认的处理函数，用于在使用退出码 `128 + signal number` 退出前重置终端模式。如果这些信号中的某一个设置了监听器，那么就会移除上述的默认行为（Node.js 将不再会退出）。
- `SIGPIPE` ，该信号默认会被系统忽略，可以设置一个监听器。
- `SIGHUB` ，当 Windows 的控制台窗口退出时会生成该信号，其他平台出现类似情况时，也会生成该信号，详见 `signal(7)` 。该信号可以设置一个监听器，不过在 Windows 将会无条件地在 10 秒后终端 Node.js，在非 Windows 的平台上， `SIGHUB` 的默认行为是关闭 Node.js，但如果为该信号设置了监听器，则会移除默认欣慰。
- `SIGTERM` ，该信号可以设置监听事件，但不支持 Windows 平台。
- `SIGINT` ，该信号支持所有平台，通常使用 `Control+C` 发出。当终端使用原始模式时不会产生该信号。
- `SIGBREAK` ，在 Windows 平台上按下 `CTRL + BREAK` 组合键时发出该信号，在非 Windows 平台上可以监听该事件但不能产生该信号。
- `SIGWINCH` ，当控制台大小发生变化时发出该信号。在 Windows 平台上，只有当控制台位置发生变化或者使用原始模式运行可读 TTY 时才会向控制台发出该信号。
- `SIGKILL` ，在所有平台上都不能对该信号设置监听器，它会无条件地终端 Node.js 进程。

- `SIGSTOP`，不允许对该信号设置监听器

注意，Windows 平台不支持发送信号，但 Node.js 通过 `process.kill()` 和 `child_process.kill()` 等方法提供了一些模拟信号发送的机制。发送信号 `0` 可以测试某个进程是否存在。发送 `SIGINT`、`SIGTERM` 和 `SIGKILL` 将会无条件地终端目标进程。

process.abort()

该方法可以让系统触发 `abort` 事件，并让 Node.js 退出且生成一个核心文件。

process.arch

该属性显示当前运行平台的处理器架构，常见值：arm / ia32 / x64。

```
console.log('This processor architecture is ' + process.arch);
```

process.argv

该属性是一个包含命令行参数的数组，其数组的第一个元素一定是 `node`，第二个元素一定是 JavaScript 的文件名，剩下元素为其他的命令行参数。

```
// print process.argv
process.argv.forEach((val, index, array) => {
  console.log(`${index}: ${val}`);
});
```

执行效果：

```
$ 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.chdir(directory)

该方法用于修改进程的当前工作目录，如果执行失败则抛出异常：

```
console.log(`Starting directory: ${process.cwd()}`);
try {
  process.chdir('/tmp');
  console.log(`New directory: ${process.cwd()}`);
}
catch (err) {
  console.log(`chdir: ${err}`);
}
```

process.config

该属性是一个对象，包含了编译 Node.js 执行文件的配置参数。该文件类似于运行 `./configure` 脚本时生成的 `config.gupi` 。

```
{
  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.connected

- 布尔值，调用 `process.disconnect()` 之后系统会将该属性设为 false

如果 `process.connected === false` ，则进程不再发送消息。

process.cwd()

该方法返回进程的当前工作目录：

```
console.log(`Current directory: ${process.cwd()}`);
```

process.disconnect()

该方法用于关闭与父进程的 IPC 信道，并允许子进程在没有其他连接时平缓结束活动。

该方法和父进程的 `ChildProcess.disconnect()` 完全相同。

如果 Node.js 没有使用 IPC 信道分立子进程，则 `process.disconnect()` 的值为 undefined。

process.env

该属性是一个对象，包含了开发环境的信息，详见 `environ(7)` 。

```
{
    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'
}
```

开发者可以修改该属性，但是修改后的配置信息并不会影响进程，这也就是说如下操作不会有什么效果：

```
$ node -e 'process.env.foo = "bar"' && echo $foo
```

但下面的处理方式会有效果：

```
process.env.foo = 'bar';
console.log(process.env.foo);
```

赋予 `process.env` 内部属性的值都会被隐式转换为字符串：

```
process.env.test = null;
console.log(process.env.test);
// => 'null'
process.env.test = undefined;
console.log(process.env.test);
// => 'undefined'
```

使用 `delete` 可以删除 `process.env` 内部的属性：

```
process.env.TEST = 1;
delete process.env.TEST;
console.log(process.env.TEST);
// => undefined
```

process.execArgv

该属性包含了一组 Node.js 进程运行时可用的命令行选项。这些选项并不会出现在 `process.argv` 中，且不包含可执行文件、脚本文件名以及文件名之后的所有选项。这些选项有助于使用和父进程相同的参数来分立子进程：

```
$ node --harmony script.js --version
```

`process.execArgv` 的值：

```
['--harmony']
```

`process.argv` 的值：

```
['/usr/local/bin/node', 'script.js', '--version']
```

process.execPath

该属性表示启动进程的脚本文件的绝对路径。

```
/usr/local/bin/node
```

process.exit([code])

该方法根据指定的 `code` 参数终端进程。如果缺失参数，则系统默认 `code` 为成功退出码 `0` 。

下面代码演示了一种失败的退出：

```
process.exit(1);
```

执行上述 Node.js 代码的 shell 可以看到退出码 `1` 。

process.exitCode

该属性是一个表示退出码的数值，无论是进程是平稳退出还是使用 `process.exit()` 强制退出，都会返回相应的退出码。

如果使用 `process.exit(code)` 退出进程，则 `code` 将会覆盖 `process.exitCode` 之前的值。

process.getegid()

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于获取进程的有效组身份，详见 `getegid(2)` ，且返回值是数值类型的组 id，而不是组名：

```
if (process.getegid) {
  console.log(`Current gid: ${process.getegid()}`);
}
```

process.geteuid()

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于获取进程的有效用户身份，详见 `geteuid(2)` ，且返回值是数值类型的用户 id，而不是用户名：

```
if (process.geteuid) {
  console.log(`Current uid: ${process.geteuid()}`);
}
```

process.getgid()

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于获取进程的组身份，详见 `getgid(2)` ，且返回值是数值类型的组 id，而不是组名：

```
if (process.getgid) {
  console.log(`Current gid: ${process.getgid()}`);
}
```

process.getgroups()

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法返回一个包含组 ID 信息的数组。如果存在有效组 ID，则在 POSIX 系统下不确定是否存在该方法返回的数组，而 Node.js 系统会认为一定有该数组。

process.getuid()

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于获取进程的用户身份，详见 `getuid(2)` ，且返回值是数值类型的用户 id，而不是用户名：

```
if (process.getuid) {
  console.log(`Current uid: ${process.getuid()}`);
}
```

process.hrtime()

该方法以 `[seconds, nanoseconds]` 形式的数组返回高解析度的实时时间。因为它是相对于过去的任意时间而，所以不存在时间漂移。该方法常用于测量代码间歇性执行的性能。

下面代码演示了如何使用 `process.hrtime()` 获取间歇执行的差异时间，这对基准测试非常有用：

```
var time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(() => {
  var diff = process.hrtime(time);
  // [ 1, 552 ]

  console.log('benchmark took %d nanoseconds', diff[0] * 1e9 + d
iff[1]);
  // benchmark took 1000000527 nanoseconds
}, 1000);
```

process.initgroups(user, extra_group)

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于读取 `/etc/group` 并初始化群组访问列表，从中选择 `user` 所在的所有群组。这是一个需要超级权限的操作，也就是说开发者需要有 root 权限，或者有 `CAP_SETGID` 的能力。

`user` 是一个用户名或用户 ID，`extra_group` 是一个组名或组 ID。

需要特别留意失去超级权限的情况：

```
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 ]
```

process.kill(pid[，signal])

该方法用于向进程发送信号，`pid` 是进程 id，`signal` 是字符串形式的信号，比如 `SIGINT` 、`SIGHUP` 。如果没有传入 `signal` 参数，那么系统会使用默认值 `SIGTERM` ，更多信息请参考信号事件和 `kill(2)` 。

如果目标进程不存在，则系统会抛出错误，这里有一个特殊情况，那就是可以使用信号 `0` 测试进程是否存在。如果使用 `pid` 杀死一个进程组，那么 Windows 平台会抛出错误。

注意，虽然该方法的名字是 `process.kill` ，但是它只是一个信号发送器，类似系统调用的 `kill` 命令。使用该方法发送信号做得事情实际上比单纯的杀死进程要多很多。

```
process.on('SIGHUP', () => {
  console.log('Got SIGHUP signal.');
});

setTimeout(() => {
  console.log('Exiting.');
  process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

注意，当 Node.js 接收到 `SIGUSER1` 信号时，系统会启动一个调试器，详见信号事件。

process.mainModule

该方法是获取 `require.main` 的替代方法。两者的差异在于如果主模块在运行时发生了改变，`require.main` 仍可能引用之前设置的主模块。通常来说，将 `process.mainModule` 和 `require.main` 视为同一个引用是比较安全的。

如果没有入门脚本，那么 `require.main` 的值为 `undefined` 。

process.memoryUsage()

该方法返回一个对象，用于描述 Node.js 进程的内存使用情况，单位为字节：

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

console.log(util.inspect(process.memoryUsage()));
```

输出结果：

```
{
    rss: 4935680,
    heapTotal: 1826816,
    heapUsed: 650472
}
```

其中，`heapTotal` 和 `heapUsed` 表示 V8 的内存使用情况。

process.nextTick(callback[, arg][, ...])

- `callback` ，函数

当前事件循环完成后，会立即调用回调函数 `callback` 。

该方法并不是 `setTimout(fn, 0)` 的同名函数，而是更加高效的方法。该方法在任何 I/O 事件（包括 timers）触发之前执行。

```
console.log('start');
process.nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

这一方法对开发 API 非常有用，有助于开发者在对象创建之后 I/O 出现之前分配事件处理器。

```
function MyThing(options) {
  this.setupOptions(options);

  process.nextTick(() => {
    this.startDoingStuff();
  });
}

var thing = new MyThing();
thing.getReadyForStuff();
// thing.startDoingStuff() gets called now, not before.
```

对 API 来说 100% 的同步执行或 100% 的异步执行都非常重要。请思考一下以下的代码：

```
// WARNING!  DO NOT USE!  BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
  if (arg) {
    cb();
    return;
  }

  fs.stat('file', cb);
}
```

这一 API 是有风险的，比如：

```
maybeSync(true, () => {
  foo();
});
bar();
```

这里无法看清是先执行 `foo()` 还是先执行 `bar()`。

下面的代码实现更加健壮：

```
function definitelyAsync(arg, cb) {
  if (arg) {
    process.nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

注意，nextTick 队列中的事件完全执行完毕后才会执行 I/O 操作。因此，递归调用 nextTick 回调函数就会阻塞 I/O 操作，类似于 `while(true)` 循环。

process.pid

该属性拜师进程的 PID：

```
console.log(`This process is pid ${process.pid}`);
```

process.platform

该属性表示当前 Node.js 的运行平台，常见值包括：darwin / freebsd / linux / sunos / win32。

```
console.log(`This platform is ${process.platform}`);
```

process.release

该属性是一个对象，包含了当前分发版本的元信息，比如源码的 URL 和 headers-only。

`process.release` 包含以下属性：

- `name`，对于 Node.js，该属性的值永远都是字符串形式的 `node`；对于 io.js，该属性的值是 `io.js`
- `sourceUrl`，该属性的值是指向当前版本源代码的 `.tar.gz` 文件的网址
- `headersUrl`，该属性的值是指向包含当前版本头信息的 `.tar.gz` 文件的网址。该文件远小于源代码的文件，可用于编译 Node.js 的插件
- `libUrl`，该属性的值是指向匹配当前版本架构的 `node.lib` 文件的网址，可用于编译 Node.js 的插件。该属性只会出现在 Windows 平台的 Node.js 中，不会出现在其他平台。

```
{
    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'
}
```

开发者根据非分发版本自主构建的 Node.js 版本也许只会包含 `name` 属性，其他属性不确定是否会存在。

process.send(message[, sendHandle[, options]][, callback])

- `message`，对象
- `sendHandle`，Handle 对象
- `options`，对象
- `callback`，函数
- 返回值类型：布尔值

如果 Node.js 分立的子进程绑定了 IPC 信道，则该子进程可以通过 `process.send()` 方法向父进程发送消息，父进程的 `ChildProcess` 对象可以通过 `message` 事件接收消息。

注意，该方法在内部使用 `JSON.stringify()` 序列化 `message`。

如果 Node.js 新建的子进程没有 IPC 信道，则 `process.send()` 的返回值为 undefined。

process.setegid(id)

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于设置进程的有效组 id，详见 `setegid(2)`。该方法接收一个数值 id 或字符串形式的组名作为参数。如果指定了组名，则该方法会阻塞进程，直到组名被解析为组 ID。

```
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.seteuid(id)

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于设置进程的有效用户 ID，详见 `seteuid(2)` 。该方法接收一个数值 ID 或字符串类型的用户名作为参数。如果指定了用户名，则该方法会阻塞进程，直到用户名被解析为用户 ID。

```
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.setgid(id)

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于设置进程的组 id，详见 `setgid(2)` 。该方法接收一个数值 id 或字符串形式的组名作为参数。如果指定了组名，则该方法会阻塞进程，直到组名被解析为组 ID。

```
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.setgroups(groups)

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于设置组 ID。这是一个需要超级权限的操作，也就是说开发者需要 root 权限或拥有 `CAP_SETGID` 能力。

`groups` 列表可以包含组 ID、组名或者两者的混合。

process.setuid(id)

注意，该函数只在 POSIX 平台可用，不适用于 Windows、Android 平台。

该方法用于设置进程的用户 ID，详见 `setuid(2)` 。该方法接收一个数值 ID 或字符串类型的用户名作为参数。如果指定了用户名，则该方法会阻塞进程，直到用户名被解析为用户 ID。

```
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.stderr

该属性是一个指向 stderr 的可写 stream。

`proces.stderr` 和 `process.stdout` 与 Node.js 中的其他 stream 不同，它们不能被关闭，也不会触发 `finish` 事件，此外，写入数据时会阻塞进程输出数据时会重定向到一个文件。因为磁盘速度通常比较快，而且操作系统会使用回写式缓存，所以阻塞进程的情况很少见。

process.stdin

该属性是一个 stdin 的可读 stream。

下面代码演示了如何打开表述输入并监听事件的操作：

```
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 Stream 的实例，`process.stdin` 也可以很好的兼容 Node.js v0.10 之前的代码。

在老版的 Stream 模式中，stdin stream 默认是暂停的，所以必须通过调用 `process.stdin.resume()` 去写入数据。注意，也可以通过调用 `process.stdin.resume()` 将 stream 切换为老版模式。

如果你正在创建一个新项目，那么最好使用新的 stream 模式。

process.stdout

该属性是一个指向 stdout 的可写 stream。

在下面的代码中模拟了 console.log：

```
console.log = (msg) => {
  process.stdout.write(`${msg}\n`);
};
```

`proces.stderr` 和 `process.stdout` 与 Node.js 中的其他 stream 不同，它们不能被关闭，也不会触发 `finish` 事件，此外，写入数据时会阻塞进程输出数据时会重定向到一个文件。因为磁盘速度通常比较快，而且操作系统会使用回写式缓存，所以阻塞进程的情况很少见。

通过 `process.stderr` 、 `process.stdout` 或 `process.stdin` 方法读取 `isTTY` 属性，可以检查当前 Node.js 是否运行在 TTY 上下文环境：

```
$ 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
```

process.title

该属性用于设置或读取 `ps` 中显示的信息。

当使用该方法设置信息时，信息的最大长度由操作系统指定，实际上可能比较短。

在 Linux 和 OS X，该属性的长度受限于二进制名称的长度加上命令行参数的长度，因为它会覆盖参数内存。

虽然 Node.js v0.8 允许较长的进程名称字符串，也支持覆盖环境内存，但是在某些情况下存在不安全或冲突问题。

process.umask([mask])

该方法用于设置或读取进程文件的掩码。子进程工父进程继承掩码。如果指定了 `mask` 参数，则返回一个旧的掩码，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.toStri
ng(8)}`
);
```

process.uptime()

该方法返回 Node.js 运行的时间，单位为秒。

process.version

该属性是一个编译属性，包含 `NODE_VERSION` ：

```
console.log(`Version: ${process.version}`);
```

process.versions

该属性包含了 Node.js 以及相关依赖的版本信息：

```
console.log(process.versions);
```

输出结果如下所示：

```
{
    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'
}
```

Punycode

接口稳定性: 0 - 已过时

punycode 模块已废弃，如果需要使用该模块，可以安装开源社区提供的 Punycode.js。

查询字符串

接口稳定性: 2 - 稳定

该模块提供了处理查询字符串的辅助函数。

querystring.escape

该转义函数供 `querystring.stringify` 使用，必要时可以重写。

querystring.parse(str[, sep][, eq][, options])

该方法用于将字符串反序列化为对象，第二和第三个参数分别表示分隔符（默认为 '&'）和分配符（默认为 '='）。

`options` 对象可以包含 `maxKeys` 属性（默认值为 1000），该属性用于限制处理过的进程数量，如果该属性的值为 0，则表示不限制。

`options` 对象可以包含 `decodeURIComponent` 属性（默认值为 querystring.unescape），如有需要可用于解码非 UTF-8 编码的字符串。

```
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.stringify(obj[, sep][, eq][, options])

该方法用于将对象序列化为字符串，第二和第三个参数分别表示分隔符（默认为 '&'）和分配符（默认为 '='）。

`options` 对象可以包含 `encodeURIComponent` 属性（默认值为 querystring.escape），如有需要可用于转义非 UTF-8 编码的字符串。

```
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.unescape

该编码函数供 `querystring.parse` 使用，必要时可以重写。该函数首先会尝试使用 `decodeURIComponent` 进行解码，如果解码失败会回退到一个安全状态，不会抛出错误的 URL。

Readline

> 接口稳定性: 2 - 稳定

通过 `require('readline')` 可以加载该模块。Readline 模块提供了逐行读取 stream （比如 `process.stdin` ）的能力。

注意，一旦调用该模块，Node.js 程序将不会自动关闭，需要开发者显式关闭接口。下面的代码演示了如何退出程序：

```
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();
});
```

Class: Interface

该类是一个包含输入输出 stream 的 readline 接口。

rl.close()

该方法用于关闭 `Interface` 实例，取消对 `input` 和 `output` stream 的控制。调用该方法会触发 `close` 事件。

rl.pause()

该方法用于暂停执行 readline 中的 `input` stream，该 stream 稍后可以回复状态继续执行。

注意，该方法不会立即暂停 stream 的操作。调用该方法后将会触发多个事件，包括 `line` 。

rl.prompt([preserveCursor])

该方法为用户配置好输入环境，并将当前的 `setPrompt` 置于新行，指定用户输入的位置。如果 `preserveCursor` 的值为 `true` ，则不允许光标位置被重置为 `0` 。

如果 `input` stream 已经暂停，也可以使用 `createInterface` 重置。

如果调用 `createInterface` 时， `output` 的值为 `null` 或 `undefined` ，那么就不会写入提示符。

rl.question(query, callback)

该方法向用户显示 `query` ，并在用户响应后执行 `callback` 。

如果 `input` stream 已经暂停，也可以使用 `createInterface` 重置。

如果调用 `createInterface` 时， `output` 的值为 `null` 或 `undefined` ，那么就不会写入提示符。

```
rl.question('What is your favorite food?', (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});
```

rl.resume()

该方法用于恢复 `input` stream 的状态。

rl.setPrompt(prompt)

该方法用于设置提示符，举例来说，当你在命令行中运行 Node.js 时，你会发现一个 `>` 符号，这就是 Node.js 的提示符。

rl.write(data[, key])

该方法用于将 `data` 写入 `output` stream，除非调用 `createInterface` 时，`output` 的值为 `null` 或 `undefined`。`key` 是一个对象，包含了一系列的键名。只有终端是 TTY 时，该方法才可用。

如果 `input` stream 已经暂停，也可以该方法重置。

```
rl.write('Delete me!');
// Simulate ctrl+u to delete the line written previously
rl.write(null, {ctrl: true, name: 'u'});
```

事件：'close'

- `function () {}`

当调用 `close()` 时会触发该事件。

当 `input` stream 接收到 `end` 事件时也会触发该事件。该事件触发之后，即可将 `Interface` 的实例视为已结束，比如 `input` stream 收到 `^D` 时，就会被认为是 `EOT`。

当 `input` sream 接收到 `^C`（SIGINT）时，如果没有 `SIGINT` 事件监听器，那么也会触发该事件。

事件：'line'

- `function (line) {}`

当 `input` stream 接收到行结束符（`\n`、`\r`、或 `\r\n`）时，就会触发该事件。通常来说，当用户敲下回车键或者 return 键时，就会触发该事件。这对监听用户的输入来说是一个非常有用的钩子。

```
rl.on('line', (cmd) => {
  console.log(`You just typed: ${cmd}`);
});
```

事件：'pause'

- `function () {}`

当 `input` stream 暂停时触发该事件。此外，当 `input` stream 接收到 `SIGCONT` 事件时也会触发该事件。

```
rl.on('pause', () => {
  console.log('Readline paused.');
});
```

事件：'resume'

- `function () {}`

当 `input` stream 从暂停中回复过来时触发该事件。

```
rl.on('resume', () => {
  console.log('Readline resumed.');
});
```

事件：'SIGCONT'

- `function () {}`

该事件在 Windows 上不会触发。

当 `input` stream 向后台传送 `^Z` （ `SIGTSTP` ）时就会触发该事件，然后继续执行 `fs(1)` 。该事件只有程序切换到后台前 stream 没有暂停才会触发。

```
rl.on('SIGCONT', () => {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

事件：'SIGINT'

- `function () {}`

当 `input` stream 接收到 `^C` （ `SIGINT` ）时就会触发该事件。如果接收到 `SIGINT` 却没有 `SIGINT` 事件监听器，那么就会触发 `pause` 事件。

```
rl.on('SIGINT', () => {
  rl.question('Are you sure you want to exit?', (answer) => {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

事件：'SIGTSTP'

- `function () {}`

该事件在 Windows 上不会触发。

当 `input` stream 向后台传送 `^Z`（`SIGTSTP`）时就会触发该事件。如果接收到 `SIGINT` 却没有 `SIGINT` 事件监听器，那么就会将程序切换到后台。

当程序通过 `fg` 回复时，会触发 `pause` 和 `SIGCONT` 事件。开发者可以使用任意一个事件恢复 stream 的状态。

如果程序切换到后台前 stream 暂停了，那么就不会触发 `pause` 和 `SIGCONT` 事件。

```
rl.on('SIGTSTP', () => {
  // This will override SIGTSTP and prevent the program from goi
ng to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

实例：Tiny CLI

下面代码演示了如何使用上述方法创建一个命令行接口：

```
const readline = require('readline');
const rl = readline.createInterface(process.stdin, process.stdou
t);

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);
});
```

实例：逐行读取文件 stream

一个常见的操作就是使用 readline 的 `input` 选项读取文件 stream。下面带那么演示了如何逐行解析文件：

```
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);
});
```

readline.clearLine(stream, dir)

该方法从指定的方法清除 TTY stream 中的当前行，其中 `dir` 有如下可选值：

- `-1`，从光标向左
- `1`，从光标向右
- `0`，正行

readline.clearScreenDown(stream)

该方法用于清空屏幕上从光标位置开始的内容。

readline.createInterface(options)

该方法创建一个 readline `Interface` 实例。可选参数 `options` 包含以下属性：

- `input`，监听的可读 stream，必选属性
- `output`，输入 realine 数据的可写 stream，可选属性
- `completer`，可选函数，用于自动补全
- `terminal`，如果需要 `input` 和 `output` stream 的行为类似于 TTY 并使用 ANSI/VT1000 编码，则设置该属性为 true。默认在 `output` stream 实例化时执行 `isTTY` 检查
- `historySize`，指定保留多少行的历史记录，默认值为 30

`completer` 函数接收用户的输入并返回一个数组，该数组包含两条记录：

1. 复核补全的字符串数组
2. 用于匹配的子串

最终结果类似于：`[[substr1, substr2, ...], originalsubstring]`。

```
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]
}
```

如果给 `completer` 传入两个参数，那么也可以以异步的方式执行：

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```

`createInterface` 常常和 `process.stdin` 和 `process.stdout` 配合使用，用来接收用户的输入：

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

一旦创建了 readline 的实例之后，开发者通常会监听 `line` 事件。

如果 readline 实例的 `terminal` 的值为 `true` 并且定义了 `output.columns` 属性，那么 `output` stream 就会保持最佳兼容性。当行大小发生变化时（如果是 TTY，则 process.stdout 会自动处理此类变化），会在 `output` stream 上触发 `resize` 事件。

readline.cursorTo(stream, x, y)

在给定的 TTY stream 中，该方法用于将光标移动到特定位置。

readline.moveCursor(stream, dx, dy)

在给定的 TTY 中，该方法相对于光标的当前位置移动光标。

REPL

接口稳定性: 2 - 稳定

Read-Eval-Print-Loop（REPL）既可用作独立的程序，也可以集成到其他程序中。REPL 提供了交互式运行 JavaScript 的方式，可以用来调试或测试代码。

从命令行执行 `node` 命令就可以进入 REPL，该 REPL 提供了一个简化版的 emacs 行编辑模式：

```
$ node
Type '.help' for options.
> a = [ 1, 2, 3];
[ 1, 2, 3 ]
> a.forEach((v) => {
...   console.log(v);
...   });
1
2
3
```

如果要使用高级行编辑器，可以通过环境变量 `NODE_NO_READLINE=1` 启动 Node.js，在该编辑器中可以使用 `rlwrap` 。

此外，你也可以将将其添加到 `bashrc` 文件中：

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

环境变量

以下环境变量可以用于调整内建的 REPL（通过 `node` 或 `node -i` 启动）：

- `NODE_REPL_HISTORY` ，该变量用于指定 REPL 历史记录的存放位置，默认为用户主页目录的 `.node_repl_history` 。

- `NODE_REPL_HISTORY_SIZE`，默认值为 `1000`，用于限制历史记录的数量，必须为正数
- `NODE_REPL_MODE`，可选值为 sloppy/strict/magin，默认值为 magic。默认在严格模式下运行只适用于严格模式的语句。

永久化历史记录

默认情况下，REPL 为将 REPL 中的回话激励保存在 `.node_repl_history` 文件中。如果想禁用该特性，可以通过修改环境变量 `NODE_REPL_HISTORY=""` 实现。

NODE_REPL_HISTORY_FILE

接口稳定性: 0 - 已过时

REPL 特性

在 REPL 中，通过 `Control + D` 可推出 REPL 环境。REPL 支持输入多行语句以及本地或全局变量的 tab 补全。

核心模块会被按需加载到 REPL 中，比如使用 `fs` 模块时，系统会自动通过 `require()` 将 fs 模块加载为 `global.fs`。

特殊变量 `_` 表示上一次执行的结果：

```
> [ 'a', 'b', 'c' ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
4
```

REPL 支持对任意全局变量的访问。开发者可以通过将变量绑定给 `context` 对象显式暴漏给 REPL：

```
// repl_test.js
const repl = require('repl');
var msg = 'message';

repl.start('> ').context.m = msg;
```

在 `context` 对象中的属性，可以在 REPL 中以本地变量的形式使用：

```
$ node repl_test.js
> m
'message'
```

下面是一些特殊的 REPL 命令：

- `.break`，当输入多行语句时，可能会后悔，此时使用 `.break` 可以重新开始
- `.clear`，该命令用于将 `context` 对象重置为空对象，并清空多行表达式
- `.exit`，该命令用于关闭 I/O stream
- `.help`，该命令用于显示 REPL 中的特殊命令
- `.save`，该命令用于将当前 REPL 会话保存为文件
- `.load`，该命令用于将会话文件加载到当前 REPL 中

下面是一些特殊的 REPL 快捷键：

- `Control-C`，类似 .break，用于关闭当前 REPL 会话
- `Control-D`，类似 .exit 命令
- `tab`，显示全局和本地变量

自定义对象

REPL 模块内部使用了 `util.inspect()` 打印值。不过，如果对象下挂载 `inspect()` 方法，那么 `util.inspect()` 就会调用对象的 `inspect()` 方法。

在下面的代码中，假设你在对象上定义了一个 `inspect()` 方法：

```
> var obj = {foo: 'this will not show up in the inspect() output'
};
undefined
> obj.inspect = () => {
...   return {bar: 'baz'};
... };
[Function]
```

然后在 REPL 中查看 `obj`，就会发现 REPL 自动调用了 `inspect()` 方法：

```
> obj
{bar: 'baz'}
```

Class: REPLServer

该类继承自 `Readline Interface`，并拥有一下事件：

事件：'exit'

- `function () {}`

当用户退出 REPL 时就会触发该事件，常见的退出方式包括输入 `.exit` 命令退出 REPL，按两次 Control + C 发送 `SIGINT` 信号，或者按 Control + D 向 `input` stream 发送 `end` 信号。

```
replServer.on('exit', () => {
  console.log('Got "exit" event from repl!');
  process.exit();
});
```

事件：'reset'

- `function (context) {}`

当 REPL 的 context 被重置时触发该事件。在 REPL 中输入 `.clear` 命令就会触发该事件。如果开发者使用 `{ useGlobal: true }` 启动 REPL，那么就不会触发该事件。

```
// 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);
});
```

replServer.defineComman(keyword, cmd)

- `keyword`，字符串
- `cmd`，对象或函数

该方法用于定义 REPL 可用的命令，此类命令以 `.` 开头。`cmd` 对象拥有以下属性：

- `help`，在 REPL 中输入 `.help` 时显示的辅助信息，可选属性
- `action`，函数，接收一个字符串参数。在 REPL 调用自定义命令时，该函数绑定到 REPLServer 的实例上，必选属性

如果传给 `cmd` 对象的是一个函数而不是对象，那么该函数会被视为是 `action`。

```
// 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();
  }
});
```

在 REPL 中执行的效果：

```
> .sayhello Node.js User
Hello, Node.js User!
```

replServer.displayPrompt([preserveCursor])

- `preserveCursor`，布尔值

该方法与 `readline.prompt` 类似，不同之处在于使用省略号添加了缩进。`preserveCursor` 参数将会被传递给 `readline.prompt`。该方法类似于 `defineCommand` 命令。在 REPL 内部使用该方法渲染每一行的提示符。

repl.start([options])

该方法创建和返回一个 `REPLServer` 实例，该实例继承自 `Readline Interface`，其中 `options` 参数是一个对象，包含以下属性：

- `prompt`，所有 I/O 的提示符和 stream，默认值为 `>`
- `input`，监听的可读 stream，默认值为 `process.stdin`
- `output`，写入 readline 数据的可写 stream，默认值为 `process.stdout`
- `terminal`，如果需要 `input` 和 `output` stream 的行为类似于 TTY 并使用 ANSI/VT1000 编码，则设置该属性为 true。默认在 `output` stream 实例化时执行 `isTTY` 检查
- `eval`，函数，用于计算每一行的语句。默认是 `eval()` 的异步封装。

- `useColors`，布尔值，指定是否允许 `writer` 函数输出颜色。如果设置了其他的 `writer` 函数，则该属性无效。默认值为 REPL 的 `terminal`。
- `useGlobal`，如果值为 `true`，REPL 会使用 `global` 对象而不是独立的 `context` 对象执行脚本，默认值为 false
- `ignoreUndefined`，如果只为 `true`，则 REPL 不会输出 `undefined` 值，默认值为 `false`
- `writer`，用于格式化输出结果的函数，默认值为 `util.inspect()`
- `replMode`，该属性决定以何种模式运行所有的命令，包括严格模式、默认模式和混合模式（`magic`），可选值包括：
  - `repl.REPL_MODE_SLOPPY`，sloppy 模式
  - `repl.REPL_MODE_STRICT`，严格模式
  - `repl.REPL_MODE_MAGIC`，首先会尝试默认模式，如果失败则使用严格模式

开发者也可以使用自定义的 `eval` 函数，该函数需要包含一下特性：

```
function eval(cmd, context, filename, callback) {
  callback(null, result);
}
```

对于 tab 补全，系统会使用 `.scope` 命令调用 `eval`，然后返回一个包含作用域名的数组用于自动补全。

同一个 Node.js 实例可以运行多个 REPL，所有的 REPL 共享全局对象，但拥有独立的 I/O。

下面代码演示了在 stdin、Unix socket 和 TCP socket 上启动一个 REPL：

```
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);
```

从命令行运行上段代码将会在 stdin 启动一个 REPL。其他 REPL 可以通过 Unix socket 或 TCP socket 连接到此 REPL。`telnet` 常用于连接 TCP scoket，`socat` 既可以连接 Unix 也可以连接 TCP socket。

从一个基于 socket 的 Unix 服务器启动 REPL 的话，那么可以直接连接到长时间运行的 Node.js 进程，而无需重启该进程。

有关通过 `net.Server` 和 `net.Socket` 实例运行全功能 REPL 的实例，请参考 https://gist.github.com/2209310。

有关通过 `curl(1)` 运行 REPL 的实例，请参考 https://gist.github.com/2053342。

Stream

stream 是一个在 Node.js 上使用流数据的抽象接口，提供了一些基础的 API， 方便我们基于它实现流式的接口。

Node.js 提供了很多原生的流对象，如 HTTP 服务的 request，process 模块的 stdout 等。

流可以是可读的，可写的或者是同时可读写的。所有的流都是 EventEmitter 实例。

可以通过下面这种方式加载 `stream` 模块

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

尽管所有的 Node.js 用户都必须弄明白 stream 是如何工作的，但 stream 模块本身只对正在创建新类型流的开发人员最有用。 其他开发者很少须要直接使用 stream 模块。

本文档的结构

本文档分为两个主要部分，和一个附加注释部分。

- 第一部分介绍了开发者需要在开发中使用 steam 所涉及的 API。
- 第二部分介绍了开发者创建自定义 stream 所需要的 API。

Stream 的类型

在 Node.js 中 stream 一共有 4 种基本类型

- Readable 可以读取数据的流(如 fs.createReadStream())
- Writable 可以写入数据的流(如 fs.createWriteStream())
- Duplex 同时可读又可写的流(如 net.Socket())
- Transform 在写入和读取过程中对数据进行修改变换的 Duplex 流(如 zlib.createDeflate())

对象模式

通过 Node API 创建的流，只能够对字符串或者 buffer 对象进行操作。但其实流的实现是可以基于其他的 Javascript 类型(除了 null, 它在流中有特殊的含义)的。这样的流就处在 "对象模式" 中。

在创建流对象的时候，可以通过提供 objectMode 参数来生成对象模式的流。 试图将现有的流转换为对象模式是不安全的。

缓冲区

Readable 和 Writable 流都会将数据储存在内部的缓冲区中。 缓冲区可以分别通过 `writable._writableState.getBuffer()` 和 `readable._readableState.buffer` 来访问。

缓冲区中能容纳的数据数量由 stream 构造函数的 `highWaterMark` 选项决定。对于普通的流来说， `highWaterMark` 选项表示总共可容纳的比特数。对于对象模式的流，该参数表示可以容纳的对象个数。

当一个可读实例调用 stream.push() 方法的时候， 数据将会被推入缓冲区。如果没有数据的消费者出现，调用 stream.read() 方法的话，数据就会一直留在缓冲队列中。

如果可读实例内部的缓冲区大小达到了创建时由 `highWaterMark` 指定的阈值，可读流就会暂时停止从底层资源汲取数据，直到当前缓冲的数据成功被消耗掉 (也就是说，流停止调用内部用来填充缓冲区的 readable._read() 方法)。

在一个在可写实例上调用 writable.write(chunk) 方法的时候，数据会写入可写流的缓冲区。如果缓冲区的数据量低于 `highWaterMark` 设定的值， 调用 `writable.write()` 方法会返回 `true` ，否则 write 方法会返回 `false` 。

stream 模块的 API，特别是 stream.pipe()， 最主要的目的就是将数据的流动缓冲到一个可接受的水平，不让不同速度的数据源之间的差异导致内存被占满。

Duplex 流和 Transform 流都是同时可读写的， 所以他们会在内部维持两个缓冲区，分别用于读取和写入，这样就可以允许两边同时独立操作， 维持高效的数据流。比如说 `net.Socket` 是一个 Duplex 流，Readable 端允许从 socket 获取、消耗数据，Writable 端允许向 socket 写入数据。数据写入的速度很有可能与消耗的速度有差距， 所以两端可以独立操作和缓冲是很重要的。

使用流涉及的 API

几乎所有的 Node.js 应用，无论多简单，多多少少都会以某种方式用到流。下面是一个在 实现 Http 服务的 Node 应用中对流的使用。

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

    const server = http.createServer( (req, res) => {
      // req is an http.IncomingMessage, which is a Readable Str
eam
      // res is an http.ServerResponse, which is a Writable Stre
am

      let body = '';
      // Get the data as utf8 strings.
      // If an encoding is not set, Buffer objects will be recei
ved.
      req.setEncoding('utf8');

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

      // the end event indicates that the entire body has been r
eceived
      req.on('end', () => {
        try {
          const data = JSON.parse(body);
          // write back something interesting to the user:
          res.write(typeof data);
          res.end();
        } catch (er) {
          // uh oh!  bad json!
          res.statusCode = 400;
          return res.end(`error: ${er.message}`);
        }
      });
```

```
});

server.listen(1337);

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

Writable 流(如上例中的 res)暴露了 `write()` 和 `end()` 这样的接口，用于向流中写入数据。

Readable 流使用 EventEmitter 的 API 来通知应用，流中有可读取的数据了。有多种方式可以获取这些数据。

Readable 流和 Writable 流都以各种方法使用 EventEmitter API 来传达流的当前状态。

Duplex 流和 Transform 流都同时是 Readable 流与 Writable 流。

向流中写入数据或者消耗数据的应用并不需要直接实现流的接口，而且通常并不需要调用 `require(stream)`

想要创造新的 stream 类型的开发者请参考本文档的[自定义 stream 所需要的 API](#创建自定义-stream-所需要的-api)

Writable Streams

Writable Streams，可写流，是对数据写入的目标的一种抽象。

常见的可写流包括：

- 在客户端的 HTTP request
- 在服务端的 HTTP responses
- 可写文件流(fs.createWriteStream)
- zlib 流
- crypto 流
- TCP sockets

- child process stdin
- process.stdout, process.stderr

注意：上面列举的部分列子实际上是双工 stream，即同时实现了可读流的接口

所有的可写流都实现了 `stream.Writable` 类定义的接口。

虽然各种可写流在不同的方面有所区别，但所有的可写流都遵循下面这样的基本使用方式。

```
const myStream = getWritableStreamSomehow();
myStream.write('some data');
myStream.write('some more data');
myStream.end('done writing data');
```

stream.Writable 类

添加于 v0.9.4

close 事件

添加于 v0.9.4

当流或者流的基础资源(比如文件描述符)被关闭，就会触发 `close` 事件。该事件表示， 没有其他事件会触发，也不会再进行更多的计算。

并不是所有的可写流都会触发 `close` 事件

drain 事件

添加于 v0.9.4

如果调用 stream.write(chunk) 的时候返回了 `false`，那么当可以重新开始向流中写入数据的时候，就会触发 `drain` 事件。

```
    // Write the data to the supplied writable stream one millio
n times.
    // Be attentive to back-pressure.
    function writeOneMillionTimes(writer, data, encoding, callba
ck) {
      let i = 1000000;
      write();
      function write() {
        var ok = true;
        do {
          i--;
          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 y
et.
            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);
        }
      }
    }
```

error 事件

添加于 v0.9.4

- 参数 \

`error` 事件会在写入/传输数据发生错误的时候触发。事件的回调函数在调用的时候， 会接受一个 `Error` 参数。

finish 事件

添加于 v0.9.4

在执行 stream.end() 之后，会触发 `finish` 事件，此时所有数据都应该已经写入底层。

```
const 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.');
});
```

pipe 事件

添加于 v0.9.4

- 参数 `src` \<stream.Readable> 流向此可写流的源

`pipe` 事件会在一个可读流调用 stream.pipe()， 将一个可写流添加到他的目标集合的时候触发。

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

unpipe 事件

添加于 v0.9.4

- 参数 `src` <Readable Stream> 停止流向此可写流的源

`pipe` 事件会在一个可读流调用 stream.unpipe()， 将一个可写流从他的目标集合移除的时候触发。

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

writable.cork()

添加于 v0.11.2 `writable.cork()` 方法强制让已接受到的数据留在内存中，直到调用 `writable.uncork()` 或者 `writable.end()` 时才开始将数据写入目标。

`writable.cork()` 的主要功能是避免很多小数据块依次写入流中的情况，就不用在内部缓冲区生成备份， 以至于影响性能。在这种情况下，实现了 `writable._writev()` 方法的实例可以用更好的方式执行写入操作。

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

添加于 v0.9.4

- 参数 `chunk` \|\|\ 可选的将要写入的数据， 对于处于正常模式的流来说，chunk 必须是字符串或者 Buffer 对象，而对于处于对象模式的流， chunk 可以是除了 null 之外的任意 JavaScript 值
- 参数 `encoding` \ 如果 chunk 是字符串，此参数为其编码
- 参数 `callback` \ 可选的回调函数，当流完成时调用。

调用 `writable.end()` 方法表示流将不会再有新的数据写入。可选的 chunk 和 encoding 参数允许在流关闭之前写入最后一块数据。如果提供了可选的 callback 参数，这个函数会成为 finish 事件的一个监听器。

在调用 writable.end() 之后调用 writable.write() 会抛出一个错误。

```
// write 'hello, ' and then end with 'world!'
const file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// writing more now is not allowed!
```

writable.setDefaultEncoding(encoding)

添加于 v0.11.15

- 参数 `encoding` \ 新的默认编码
- 返回值 `this`

`writable.setDefaultEncoding()` 方法用于设置可写流的默认编码

writable.uncork()

添加于 v0.11.2

`writable.uncork()` 方法用于将被 `writable.cork()` 锁定在缓冲区中的数据释放。

当使用 `writable.cork()` 和 `writable.uncork()` 方法来管理可写流的缓冲区时，建议使用 `process.nextTick()` 方法来调用 `writable.uncork()` 。这样做允许在一个 Node.js 循环批量处理所有 `writable.write()` 调用。

```
stream.cork();
stream.write('some ');
stream.write('data ');
process.nextTick(() => stream.uncork());
```

如果 `writable.cork()` 方法在同一个流上连续调用了多次，那么必须调用同样次数的 `writable.uncork()` 方法才能将缓冲区的数据刷新到底层。

```
    stream.cork();
    stream.write('some ');
    stream.cork();
    stream.write('data ');
    process.nextTick(() => {
      stream.uncork();
      // The data will not be flushed until uncork() is called a
second time.
      stream.uncork();
    });
```

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

- 参数 chunk \| \ 待写入的数据
- 参数 encoding \ 如果 chunk 是字符串，则此参数为其编码
- 参数 callback \ 当数据块被刷新到底层的时候触发
- 返回值 \ 如果流想要等 `drain` 事件触发后，再写入新的数据，则返回 `false`，否则返回 `true`

`writable.write()` 方法将数据写入流，并在数据完成处理之后调用提供的回调函数。如果发生了错误，回调函数可能可以接收到一个 error 作为其第一个参数。如果想要确保监听到错误，请为 `error` 事件添加监听器。

返回值表示写入的块是否已经在内部缓冲区，并且缓冲区已经达到了创建流时设置的 `highWaterMark`。如果返回了 `false`，应该停止继续尝试向流写入数据，直到流发射了 `drain` 事件。

处于对象模式的可写流会忽略 `encoding` 参数。

Readable Streams

可读流是对数据来源的一种抽象。

常见的可读流包括：

- 在客户端的 HTTP 返回
- 在服务端的 HTTP 请求
- 可读文件流(fs.createReadStream)

- zlib 流
- crypto 流
- TCP sockets
- child process stdout 和 stderr
- process.stdin

所有的可读流都实现了 `stream.Readable` 定义的接口。

两种模式

可读流可以工作在两种模式，流模式和暂停模式。

在流模式下，流会从底层系统尽快地读取数据，然后通过 `EventEmitter` 的事件接口提供给应用。

在暂停模式下，必须显示的调用 stream.read 从流中读取数据。

处在暂停模式的流可以通过下面的方式切换到流模式

- 给流的 data 事件添加监听器
- 调用流的 stream.resume() 方法
- 调用 stream.pipe() 方法将数据送入一个可写流

还可以通过下面的方式切回暂停模式

- 流没有 pipe 的目标时调用 stream.pause() 方法
- 流如果有 pipe 的目标，须要移除所有 data 事件的监听器，并通过 stream.unpipe() 方法移除所有的 pipe 目标。

一个重要的概念是，如果没有消耗或忽略数据的机制的话，可读流将不会生成数据。 如果数据的消费者失效了或者被移除，可读流将会尝试停止生成数据。

注意：由于向后兼容的原因，移除 data 事件监听器并不会将流暂停。另外，如果还有 pipe 的目标的话，调用 stream.pause() 方法并不能保证当 pipe 目标索取数据时， 流还能保持在暂停状态。

注意：如果可读流切换到流模式，但却没有数据的消费者接收数据，则数据会流失。 如在没有监听 `data` 事件就调用 `readable.resume()` 方法，或者从一个可读流移除 `data` 事件监听器，就会发生这种情况。

三种状态

可读流的两种操作模式，是对其内部更复杂的状态管理的简化抽象。

具体来说就是，在任意时间点，可读流都处于下面三种状态中的一种：

- `readable._readableState.flowing = null`
- `readable._readableState.flowing = false`
- `readable._readableState.flowing = true`

当 `readable._readableState.flowing` 为 `null` 时，没有提供消耗可读流数据的机制， 因此流并不会生成数据。

为 `data` 事件添加监听器，或者调用 `readable.pipe()` 方法，又或者调用 `readable.resume()` 方法，将会把流的 `readable._readableState.flowing` 切换为 `true` ，此时，可读流开始生成数据， 并在数据生成时主动发射事件。

调用 `readable.pause()` ， `readable.unpipe()` 或者接收 `back pressure` 事件会将流的 `readable._readableState.flowing` 设置为 `false` ，暂时停止事件的流动， 但并不会停止数据生成。

当 `readable._readableState.flowing` 为 `false` 时，数据可能会在流的内部缓冲区累积。

选择一种方法

可读流API在多个 Node.js 版本中不断改进，并提供了多种消耗流数据的方法。一般来说， 开发者应该选择使用其中一种，而不应该使用多个方法来从单个流中消耗数据。

建议对大多数用户使用 `readable.pipe()` 方法，它提供了最简单的消耗流数据的方法。 需要对数据传输和生成进行更细粒度控制的开发者可以使用 `EventEmitter` 和 `readable.pause() / readable.resume()` API。

stream.Readable 类

添加于 v0.9.4

close 事件

添加于 v0.9.4

当流或者流依赖的底层资源(如文件描述符)关闭时，会发射 `close` 事件。该事件表示， 不会再有新的事件触发，也不会有进一步的计算。

并不是所有的可读流都会发射 `close` 事件。

data 事件

添加于 v0.9.4

- 参数 chunk \| \| \ 数据块。对于处在对象模式的流，chunk 可以是除了 `null` 之外任意的 JavaScript 值，否则 chunk 只能是字符串或者 `Buffer` 对象。

`data` 事件会在流放弃对一个数据块的所有权时触发。可读流通过调用 `readable.pipe()` 或 `readable.resume()` 或添加 `data` 监听器切换到流模式来放弃数据的所有权。调用 `readable.read()` ，释放出一个数据块的时候，也会触发 `data` 事件。

给一个没有显式暂停的流添加 `data` 事件监听将会把流切换到流模式。 一旦数据可用了就会开始传递。

如果使用 `readable.setEncoding()` 方法为流指定了默认编码，回调函数会被传入字符串作为参数， 否则就会传入 `Buffer` 对象。

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
```

end 事件

添加于 v0.9.4

`end` 事件会在可读流已经没有更多数据提供的时候触发。

注意：除非数据被完全消耗，否则 `end` 事件是不会触发的。可以通过切换到流模式， 或者不停调用 stream.read() 方法耗尽数据使其触发。

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
});
readable.on('end', () => {
  console.log('There will be no more data.');
});
```

error 事件

添加于 v0.9.4

- 参数 \

`error` 事件可以在任意时间触发。通常是由于底层内部错误或者流尝试推送无效数据块导致的。

监听器回调函数将会传入一个 `Error` 对象作为参数。

readable 事件

添加于 v0.9.4

当流有可用的数据可以被获取时，会触发 `readable` 事件。某些情况下，给 `readable` 事件添加监听器会导致一些数据被读入内部的缓冲区。

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

在到达流数据结尾但还没发出 `end` 事件之前，会先触发 `readable` 事件。

`readable` 事件表示了流有新的信息，可能是新的数据可用，或者到达流数据的末尾。如果是前者，调用 stream.read() 将会返回可用的数据，如果是后者，stream.read() 会返回 `null`。在下面的示例中，`foo.txt` 是一个空文件：

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

运行代码会输出：

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

通常来说，使用 `readable.pipe()` 或者 `data` 事件优于使用 `readable` 事件。

readable.isPaused()

- 返回值 \

`readable.isPaused()` 方法返回当前可读流所处的状态。此方法主要是被 `readable.pipe()` 方法的底层机制调用，一般来说是没有理由直接使用此方法的。

```
const readable = new stream.Readable

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

readable.pause()

添加于 v0.9.4

- 返回值 `this`

`readable.pause()` 方法会让处于流模式的可读流停止发射 data 事件，并退出流模式。 新的可用数据将会留在内部的缓冲区中

```
    const readable = getReadableStreamSomehow();
    readable.on('data', (chunk) => {
      console.log(`Received ${chunk.length} bytes of data.`);
      readable.pause();
      console.log('There will be no additional data for 1 second
.');
      setTimeout(() => {
        console.log('Now data will start flowing again.');
        readable.resume();
      }, 1000);
    });
```

readable.pipe(destination[, options])

添加于 v0.9.4

- 参数 `destination` \<stream.Writable> 写入数据的目标
- 参数 `options` \pip 操作的参数
  - `end` \是否在可读流结束的时候关闭可写流，默认为 `true`

`readable.pipe()` 方法将一个可写流附到可读流上，同时将可写流切换到流模式， 并把所有数据推给可写流。数据流会被自动管理，所以不用担心可写流被快速的可读流打满溢出。

下面这个例子中，可读流讲所有数据写到 `file.txt` 文件中。

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

可以将多个可写流附加到单个可读流。

`readable.pipe()` 方法返回值是对 pipe 目标的引用，以便使用链式调用：

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

默认情况下，当可读源发射 `end` 事件的时候，目标可写流会自动调用 stream.end() 方法，导致可写流不能再写入。 如果想阻止此默认行为，须要将 `end` 选项设置为 false，让目标可写流保持打开状态， 像下面的例子：

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

一个重要的警告是，如果可读流抛出错误，目标可写流并不会自动关闭。如果发生错误， 则必须手动关闭每一个流，以防止内存泄露。

注意：prcess.stderr 和 process.stdout 可写流在 Node.js 进程退出之前从不关闭，不管传入什么选项都会被忽视。

readable.read([size])

添加于 v0.9.4

- 参数 size \ 可选参数，指定要读取的数据量。
- 返回值 \|\|\

`readable.read()` 方法从内部缓冲区抓取并返回数据。如果没有可用数据，则返回 `null` 。 数据默认以 `Buffer` 对象返回，除非使用 `readable.setEncoding()` 方法设定了编码， 或者流在对象模式下运行。

`size` 参数指定了要读取的字节数。如果没有那么多字节的数据可用，除非已经到了数据的末尾， 否则将会返回 `null` 。如果到达了流末尾，将返回保留在内部缓冲区的所有数据 (即使这些数据已经超过了指定字节数)。

如果未指定 `size` 参数，此方法将返回包含在内部缓冲区中的所有数据。

只有在暂停模式下的留才可以调用 `readable.read()` 方法。在流模式下， `readable.read()` 会自动被调用，直到内部缓冲区耗尽。

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

一般来说，建议开发者避免使用 `readable` 和 `readable.read()` 方法来支持 `readable.pipe()` 或 `data` 事件的使用。

对象模式下的流调用 `read.read(size)` 总是返回单个对象，无视 `size` 参数的值。

注意：如果 `readable.read()` 方法返回了一个数据块，还会触发一个 `data` 事件。

注意：在 `end` 事件触发后调用 stream.read([size]) 方法将返回 `null`，不会产生错误。

readable.resume()

添加于 v0.9.4

- 返回值 `this`

`readable.resume()` 方法让一个显式暂停的流重新开始发射 data 事件，切换到流模式。

`readable.resume()` 方法可以用于完全消耗掉数据流，而实际上并不对数据做任何处理， 可参考下面的例子：

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

readable.setEncoding(encoding)

添加于 v0.9.4

- 参数 `encoding` \ 要使用的编码
- 返回值 `this`

`readable.setEncoding()` 方法可以设置从可读流读取的数据的默认字符编码。

设置编码会让流数据以字符串的形式传递而不是默认的 `Buffer` 对象。例如，调用 `readable.setEncoding('utf8')` 会让输出的数据按 UTF-8 解析，并以字符串传递。调用 `readable.setEncoding('hex')` 会让数据按照十六进制格式编码。

可读流可以正确的处理流中的多字节字符，如果只是简单的拉取 `Buffer` 对象，会导致多字节字符不适当地解码。

可以使用 `readable.setEncoding(null)` 来禁用编码。这在处理二进制数据，或分布在多个块上的大型多字节字符时很有效。

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

readable.unpipe([destination])

添加于 v0.9.4

- 参数 `destination` \<stream.Writable> 可选参数，指定要解绑的流。

`readable.unpipe()` 方法会移除之前使用 stream.pipe() 附加的可写流。

如果没有指定 `destination` 参数，则会移除附加的所有可写流。

如果指定了 `destination` 参数，但指定的目标并没有附加在可读流上，则此方法不做任何操作。

```
const readable = getReadableStreamSomehow();
const 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)

添加于 v0.9.11

- 参数 `chunk` \|\ 要退回可读流的数据

`readable.unshift()` 方法将一个数据块推回内部缓冲区。此方法主要用于数据被不应该消耗数据的代码消耗了， 须要撤销这次数据消耗的情况，退回缓冲区让数据可以传到其他正确的地方去。

注意：`stream.unshift(chunk)` 方法无法在 `end` 事件触发或者抛出运行错误之后调用。

使用 `stream.unshift()` 的开发者通常应该考虑使用 Transform 流代替。更多信息，可以参考本文档第二部分 开发者创建自定义 stream 所需要的 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').StringDecode
r;
    function parseHeader(stream, callback) {
      stream.on('error', callback);
      stream.on('readable', onReadable);
      const 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();
            const remaining = split.join('\n\n');
            const buf = Buffer.from(remaining, 'utf8');
            stream.removeListener('error', callback);
            // set the readable listener before unshifting
            stream.removeListener('readable', onReadable);
            if (buf.length)
              stream.unshift(buf);
            // now the body of the message can be read from the
stream.
            callback(null, header, stream);
          } else {
            // still reading the header.
            header += str;
          }
        }
      }
    }
```

注意：`stream.unshift(chunk)` 与 stream.push(chunk) 方法不同，并不会改变流的内部状态来结束读取过程。在读取数据期间(比如在一个自定义流的 stream._read() 中)使用 `stream.unshift()` 方法可能会导致意外的结果。 在调

用 `readable.unshift()` 后立即调用 stream.push(chunk) 方法会正确的设置流的内部状态。但最好还是不要在读取过程中调用此方法。

readable.wrap(stream)

添加于 v0.9.4

- 参数 `stream` \<Stream> 旧的可读流

在 0.10 版本之前的 Node.js 中，并没有实现当前定义的流模块的 API。(详见 兼容性部分)

在使用旧版的 Node.js 库，使用只有 stream.pause() 方法和 data 事件发射的流时，可以用 `readable.wrap()` 方法将其包装成一个新的可读流。

很少会有使用 `readable.wrap()` 方法的时候，它主要是方便与较早的 Node.js 应用和库交互的。

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.
});
```

Duplex and Transform Streams

stream.Duplex 类

添加于 v0.9.4

双工流(Duplex stream) 是同时实现了可读和可写接口的流。

常见的双工流包括：

- TCP sockets

- zlib streams
- crypto streams

stream.Transform 类

添加于 v0.9.4

Transform 流是输出以某种方式依赖于输入的双工流。作为双工流，他也实现了可读与可写的接口。

常见的 Transform 流包括：

- zlib streams
- crypto streams

创建自定义 stream 所需要的 API

stream 模块的 API 设计使其可以用 JavaScript 的原型继承简单地实现自定义的流。

首先，开发者须要声明一个新的 JavaScript 类，扩展自四个基本类之一（`stream.Writable`，`stream.Readable`，`stream.Duplex`，`stream.Transform`），并确保调用了父类的构造函数：

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    super(options);
  }
}
```

这个新的类须要实现一个或多个特定的方法，具体取决于要创建流的类型，详见下表：

用途	类	实现的方法
只读	Readable	_read
只写	Writable	_write, _writev
读写	Duplex	_read, _write, _writev
写数据读结果	Transform	_transform, _flush

注意：实现中请不要调用流模块的"公用"方法(在 使用流涉及的 API 部分中讲到的)。 这样做可能会导致消耗流数据是产生不良的副作用。

构造简单的流

对于很多简单的情况，可以不通过继承，直接通过传递适当的选项调用构造函数来创建 `stream.Writable` , `stream.Readable` , `stream.Duplex` , `stream.Transform` 实例。

例如：

```
const Writable = require('stream').Writable;

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  }
});
```

实现一个可写流

可以通过继承扩展 `stream.Writable` 类来实现一个可写流。

自定义流须必须调用 `new stream.Writable([options])` 构造函数并实现 `writable._write()` 方法，也可以实现可选的 `writable._writev()` 方法。

构造函数 new stream.Writable([options])

- 参数 `options` \
  - `highWaterMark` \指定缓存数量达到什么水平的时候 stream.write() 开

始返回 `false` 。默认是 `16384` (16kb) 或对象模式的流是 `16` (个对象)
- `decodeStrings` \ 指定是否在将字符串数据传给 _write 方法前解码，默认为 `true`
- `objectMode` \ 决定调用 stream.write(anyObj) 是否合法。如果设置了此属性，则可以向流中写入除了字符串和 `Buffer` 对象之外的其他 JavaScript 值。默认为 `false`
- `write` \ 对 stream._write() 的实现
- `writev` \ 对 stream._writev() 的实现

例如：

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    // Calls the stream.Writable() constructor
    super(options);
  }
}
```

或者使用 ES6 之前的风格构造：

```
const Writable = require('stream').Writable;
const util = require('util');

function MyWritable(options) {
  if (!(this instanceof MyWritable))
    return new MyWritable(options);
  Writable.call(this, options);
}
util.inherits(MyWritable, Writable);
```

或者使用简化构造方法

```
const Writable = require('stream').Writable;

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    // ...
  },
  writev(chunks, callback) {
    // ...
  }
});
```

writable._write(chunk, encoding, callback)

- 参数 `chunk` \|\ 将要被写入的数据。除非 `decodeStrings` 被设为 `false`，否则都会是 `Buffer` 对象
- 参数 `encoding` \ 如果 chunk 为字符串，那此参数为字符串的编码，如果 chunk 是 `Buffer` 对象或者流处于对象模式，`encoding` 参数被忽略
- 参数 `callback` \ 在完成数据块的处理后须调用此函数(带可选的 error 参数)

所有的可写流都必须提供 stream._write() 的实现，用于将数据写入底层。

注意：Transform 流会提供自己的 stream._write() 函数实现。

注意：此函数不应该被直接调用。子类只是提供它的实现，它只应被可写流的内部方法调用。

`callback` 函数必须被调用，以通知对数据块的处理完成或是出现错误。如果发生错误，则传入一个错误对象作为他的第一个参数，否则传入 `null` 即可。

须要注意的是，在调用 `writable._write()` 到 `callback` 函数被调用期间，调用 `writable.write()` 写入的数据将会被放入缓冲区。调用 `callback` 函数将会发射一个 `drain` 事件。如果流具备一次处理多个数据块的能力，则应该实现 `wratable._writev()` 方法。

如果设置了 `decodeStrings` 参数，那么 chunk 可能是一个字符串而不是一个 `Buffer` 对象，`encoding` 参数将表示其字符编码。这是为了支持一些针对特定编码进行优化的实现。如果将 `decodeStrings` 显式设置为 `true` (此处存疑，原文为 false)，则可以安全地忽略 `encoding` 参数，chunk 将会是 `Buffer` 对象。

`writable._write()` 方法带有下划线前缀，说明它是一个内部方法，只提供给定义它的类内部使用， 开发者不应直接调用它。

writable._writev(chunks, callback)

- 参数 `chunks` \将要被写入的数据块，每一块数据的格式会是这样： `{ chunk: ..., encoding: ...}` 。
- 参数 `callback` \接受一个可选的错误参数的回调函数，须要在处理完数据块后调用

注意：此函数不应该被直接调用。子类只是提供它的实现，它只应被可写流的内部方法调用。

`writable._writev()` 函数是 `writeable.write()` 的补充，用于同时接收多个数据块， 并进行处理。如果实现了此方法, 将会用当前缓冲区内的所有数据作为参数来调用它。

`writable._writev()` 方法带有下划线前缀，说明它是一个内部方法，只提供给定义它的类内部使用， 开发者不应直接调用它。

可写流中的错误处理

建议通过向 `callback` 函数传入错误对象作为第一个参数，来处理 `writable._write()` 和 `writable._writev()` 过程中发生的错误。这样可写流就会抛出 `error` 事件。 在 `writable._write()` 中直接抛出错误可能会导致与预期不一致的行为，具体取决于如何使用流。 使用回调可确保一致和可预测的错误处理。

```
const Writable = require('stream').Writable;

const myWritable = new Writable({
  write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'));
    } else {
      callback();
    }
  }
});
```

一个可写流示例

下面展示一个非常简单(可以说没有意义)的自定义可写流实现。 虽然这个可写流实例没有任何实际的用处，但它展示了自定义可写流实例的每个必需元素：

```
const Writable = require('stream').Writable;

class MyWritable extends Writable {
  constructor(options) {
    super(options);
  }

  _write(chunk, encoding, callback) {
    if (chunk.toString().indexOf('a') >= 0) {
      callback(new Error('chunk is invalid'));
    } else {
      callback();
    }
  }
}
```

实现一个可读流

可以通过继承扩展 `stream.Readable` 类来实现一个可写流。

自定义流须必须调用 `new stream.Readable([options])` 构造函数并实现 `writable._write()` 方法。

构造函数 new stream.Readable([options])

- 参数 `options` \
  - `highWaterMark` \缓冲区最大容量，默认为 16384(16kb)，对于对象模式的流， 默认为 16(个对象)。达到最大容量后停止从底层系统汲取数据。
  - `encoding` \如果指定了此参数，数据将会按照此编码解码为字符串，默认值为 `null`
  - `objectMode` \流是否工作在流模式下。即调用 stream.read(n) 返回一个值， 而不是返回指定长度的字符串或 `Buffer` 对象。
  - `read` \对 stream._read() 的实现

例如：

```
const Readable = require('stream').Readable;

class MyReadable extends Readable {
  constructor(options) {
    // Calls the stream.Readable(options) constructor
    super(options);
  }
}
```

或者使用 ES6 之前的风格构造：

```
const Readable = require('stream').Readable;
const util = require('util');

function MyReadable(options) {
  if (!(this instanceof MyReadable))
    return new MyReadable(options);
  Readable.call(this, options);
}
util.inherits(MyReadable, Readable);
```

或者使用简化构造方法

```
const Readable = require('stream').Readable;

const myReadable = new Readable({
  read(size) {
    // ...
  }
});
```

readable._read(size)

- 参数 `Size` \要异步读取的数据量。

注意：此函数不应该被直接调用。子类只是提供它的实现，它只应被可读流的内部方法调用。

所有的可读流都必须提供自己的 `radable._read()` 方法实现，用于从底层系统汲取数据。

当 `readable._read()` 被调用时，如果底层数据可用，则应该通过 this.push(dataChunk) 方法将数据推入可读流的缓冲队列中。`_read()` 将持续的将数据汲取到缓冲队列中， 直到 `readable.push()` 返回 `false` 。再次调用 `_read()` 将使其重新开始重复以上过程。

注意： `readable._read()` 被调用后，直到调用 readable.push 之前，都不会再次被调用。

`size` 参数并不是强制性的。对于读取操作是简单地返回数据的情况，可以用 `size` 参数来决定要提取多少数据。其他情况下可以直接忽略此参数，当有数据可用就直接返回。 没有必要等到可用数据达到 `size` 指定大小再去调用 stream.push(chunk)

`readable._read()` 方法以下划线为前缀，表明它属于定义它的类的内部， 并且不应该被用户程序直接调用。

readable.push(chunk[, encoding])

- 参数 `chunk` \|\|\ 将要推入读取队列的数据。

- 参数 `encoding` \ chunk 的编码。必须是一个合法 Buffer 编码，如 `utf8` 或 `ascii` 等
- 返回值 \ 如果还需要继续推入更多数据则返回 `ture` ，否则返回 `false` 。

当 chunk 是 `Buffer` 对象或字符串时，数据块将被添加到内部缓冲区以供流的消费者使用。 如果 chunk 是 `null` 则表示流的末尾(EOF)，之后不能写入更多的数据。

如果流处在暂停模式下，那么使用 `readable.push()` 添加的数据在 readable 事件触发后， 可以通过调用 readable.read() 方法获取。

如果流处在流模式下， `readable.push()` 添加的数据将通过发射一个 `data` 事件交付出去。

`readable.push()` 方法的设计非常灵活。例如，我们要使用一个底层数据源，可以使用一个自定义可读流来包装它，以提供暂停/恢复功能和数据的回调机制。如下所示

```
    // source is an object with readStop() and readStart() metho
ds,
    // and an `ondata` member that gets called when it has data,
 and
    // an `onend` member that gets called when the data is over.

    class SourceWrapper extends Readable {
      constructor(options) {
        super(options);

        this._source = getLowlevelSourceObject();

        // Every time there's data, push it into the internal bu
ffer.
        this._source.ondata = (chunk) => {
          // if push() returns false, then stop reading from sou
rce
          if (!this.push(chunk))
            this._source.readStop();
        };

        // When the source ends, push the EOF-signaling `null` c
hunk
        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.
      _read(size) {
        this._source.readStart();
      }
    }
```

注意 `readable.push()` 方法只能在可读流实例中调用，并且只能在 `readable._read()` 中调用。

可读流中的错误处理

建议在 `readable._read()` 过程中发生的错误通过 `error` 事件抛出，而不是直接抛出。 从 readable._read() 内部直接抛出错误可能导致预期和不一致的行为，具体取决于流是以流模式还是以暂停模式运行。 使用 `error` 事件可确保一致且可预测的错误处理。

```
const Readable = require('stream').Readable;

const myReadable = new Readable({
  read(size) {
    if (checkSomeErrorCondition()) {
      process.nextTick(() => this.emit('error', err));
      return;
    }
    // do some work
  }
});
```

一个可读流示例

以下是可读流的基本示例，其以升序从 1 到 1,000,000 输出数字，然后结束。

```
const Readable = require('stream').Readable;

class Counter extends Readable {
  constructor(opt) {
    super(opt);
    this._max = 1000000;
    this._index = 1;
  }

  _read() {
    var i = this._index++;
    if (i > this._max)
      this.push(null);
    else {
      var str = '' + i;
      var buf = Buffer.from(str, 'ascii');
      this.push(buf);
    }
  }
}
```

实现一个双工流

双工流是同时可读写的流，如 TCP socket 连接。

因为 JavaScript 不支持多重继承，所以要继承 `stream.Duplex` 类来实现双工流(而不是同时继承 `steam.Writable` 类和 `stream.Readable` 类)。

注意：`stream.Duplex` 类原型继承自 `stream.Readable`，并寄生自 `stream.Writable`。不过 `instanceof` 操作符对两个类都有效，因为在 `stream.Writable` 的 `Symbol.hasInstance` 进行了重写。

自定义双工流必须调用构造函数 `new stream.Duplex([options])`，并实现 `readable._read()` 和 `writable._write()` 方法。

new stream.Duplex(options)

- 参数 `options` \传给可读与可写构造器的选项，拥有以下字段

- `allowHalfOpen` \默认为 `true` 。如果设置为 `false` ，流将会在可写端关闭的时候自动关闭可读端，反之亦然。
- `readableObjectMode` \默认为 `false` 。设置可读端是否以对象模式运行，如果 `objectMode` 为 `true` 则没有效果
- `writableObjectMode` \默认为 `false` 。设置可写端是否以对象模式运行，如果 `objectMode` 为 `true` 则没有效果

例如：

```
const Duplex = require('stream').Duplex;

class MyDuplex extends Duplex {
  constructor(options) {
    super(options);
  }
}
```

或者使用 ES6 之前的风格构造：

```
const Duplex = require('stream').Duplex;
const util = require('util');

function MyDuplex(options) {
  if (!(this instanceof MyDuplex))
    return new MyDuplex(options);
  Duplex.call(this, options);
}
util.inherits(MyDuplex, Duplex);
```

或者使用简化构造方法

```
const Duplex = require('stream').Duplex;

const myDuplex = new Duplex({
  read(size) {
    // ...
  },
  write(chunk, encoding, callback) {
    // ...
  }
});
```

一个双工流示例

下面是一个简单的例子。假设我们有一个底层的数据源，可读可写，但与 Node.js 的流 API 不兼容。我们可以用一个双工流对它进行包装，用可写流的接口缓冲写入的数据， 并用可读流的接口获取数据。

```
const Duplex = require('stream').Duplex;
const kSource = Symbol('source');

class MyDuplex extends Duplex {
  constructor(source, options) {
    super(options);
    this[kSource] = source;
  }

  _write(chunk, encoding, callback) {
    // The underlying source only deals with strings
    if (Buffer.isBuffer(chunk))
      chunk = chunk.toString();
    this[kSource].writeSomeData(chunk);
    callback();
  }

  _read(size) {
    this[kSource].fetchSomeData(size, (data, encoding) => {
      this.push(Buffer.from(data, encoding));
    });
  }
}
```

双工流很重要的一点是，尽管在同一个实例中，它的可读侧和可写侧是相互独立的。

双工流的对象模式

对于双工流，objectMode 可以使用 readableObjectMode 和 writableObjectMode 选项为可读端和可写端专门设置。

在下面这个例子中，创建了一个 Transform 流(一种双工流)，其可写端是对象模式，接受一个数字，并将其按照十六进制转换为字符串从可读端输出。

```
const Transform = require('stream').Transform;

// All Transform streams are also Duplex Streams
const myTransform = new Transform({
  writableObjectMode: true,

  transform(chunk, encoding, callback) {
    // Coerce the chunk to a number if necessary
    chunk |= 0;

    // Transform the chunk into something else.
    const data = chunk.toString(16);

    // Push the data onto the readable queue.
    callback(null, '0'.repeat(data.length % 2) + data);
  }
});

myTransform.setEncoding('ascii');
myTransform.on('data', (chunk) => console.log(chunk));

myTransform.write(1);
  // Prints: 01
myTransform.write(10);
  // Prints: 0a
myTransform.write(100);
  // Prints: 64
```

实现一个 Transform 流

Transform 流是一种双工流，其输出是根据输入计算得到的。比如用于压缩数据的 zlib 流，加密解密数据的 crypto 流等。

注意：Transform 流并不要求输入与输出的大小相同，数据块数量相同，输入输出时间相同。 比如一个 hash 流只会在输入结束的时候输出一个数据块，zlib 流的输出会比其输入大得多或者小得多。

继承 `stream.Transform` 类以实现一个 Transform 流。

`stream.Transform` 原型继承于 `stream.Duplex`，并实现了自己的 `writable._write()` 方法和 `readable._read()` 方法。自定义的 Transform 流实例必须实现 transform._transform() 方法，可以实现可选的 transform.flush() 方法。

注意：使用 Transform 流的时候要小心，如果可读端数据没有被消耗，向可写端写入数据，有可能导致可写端进入暂停状态。

new stream.Transform([options])

- 参数 `options` \ 传给可读与可写构造器的选项，拥有以下字段
  - transform \ 对 `stream._transform` 的实现
  - flush \ 对 `stream._flush` 的实现

例如：

```
const Transform = require('stream').Transform;

class MyTransform extends Transform {
  constructor(options) {
    super(options);
  }
}
```

或者使用 ES6 之前的风格构造：

```
const Transform = require('stream').Transform;
const util = require('util');

function MyTransform(options) {
  if (!(this instanceof MyTransform))
    return new MyTransform(options);
  Transform.call(this, options);
}
util.inherits(MyTransform, Transform);
```

或者使用简化构造方法

```
const Transform = require('stream').Transform;

const myTransform = new Transform({
  transform(chunk, encoding, callback) {
    // ...
  }
});
```

`finish` 和 `end` 事件

finish 事件和 end 事件分别来自 stream.Writable 和 stream.Readable 类。`fiinsh` 事件会在调用 stream.end() 之后触发。`end` 事件会在所有数据输出，并在 transform._flush() 调用之后触发。

transform._flush(callback)

- 参数 `callback` \ 在剩余的数据都被输出后调用的回调函数(可传递一个错误对象作为参数)

注意：此函数不应该被直接调用。子类只是提供它的实现，它只应被可读流的内部方法调用。

某些情况下，Transform 流可能须要在流的末端添加额外的数据。例如 zlib 流会储存一些用于优化压缩输出的内部状态。当流结束的时候，须要将这些状态也进行输出， 这样压缩数据才是完整的。

自定义 Transform 流实现可以实现 `transform._flush()` 方法。 当没有更多写入数据以供消耗时， 在发射 end 事件通知可读流结束之前，将调用该方法。

在 `transform._flush()` 实现中， `readable.push()` 方法可能不被调用或调用多次。 当数据操作完成时，必须调用 `callbak` 函数。

`transform._flush()` 方法以下划线为前缀，表明它属于定义它的类的内部， 并且不应该被用户程序直接调用。

transform._transform(chunk, encoding, callback)

- 参数 `chunk` \|\ 要被转换的数据。除非 `decodeStrings` 选项设置为

`false`，否则将为 Buffer 对象。
- 参数 `encoding` \ 如果 `chunk` 是字符串，此参数为其编码。如果是 Buffer 对象，此参数为一个特殊值 'buffer'，此情况下可以忽略它。
- 参数 `callback` \ 一个在数据块处理完成后要调用的回调函数 (可以接受一个错误对象作为参数)

注意：此函数不应该被直接调用。子类只是提供它的实现，它只应被可读流的内部方法调用。

所有Transform流实现必须提供一个 `_transform()` 方法来接受输入并产生输出。 `transform._transform()` 的实现须要接收写入的数据，计算输出，然后使用 `readable.push()` 方法将输出传递到可读部分。

对于单个数据块输入， `transform.push()` 方法可以被调用 0 次或多次，这取决于想要输出多少数据块。

可以接受输入但并不产生任何输出。

必须在当前数据块被完全处理后调用 `callback` 函数。如果处理过程中发生了错误， `callback` 函数的第一个参数必须是一个 `Error` 对象。如果给 `callback` 传入了第二个参数， 它将会被转发到 `readable.push()` 方法，也就是说下面这两段是等价的：

```
    transform.prototype._transform = function (data, encoding, c
allback) {
      this.push(data);
      callback();
    };

    transform.prototype._transform = function (data, encoding, c
allback) {
      callback(null, data);
    };
```

`transform._transform()` 方法以下划线为前缀，表明它属于定义它的类的内部， 并且不应该被用户程序直接调用。

stream.PassThrough 类

`stream.PassThrough` 是一个简单的 Transform 流，简单地将输入字节传递到输出。 它的目的主要是用于示例和测试。但某些情况下，`stream.PassThrough` 可以作为一种构建块的新型流。

附加注释

与旧版 Node.js 间的兼容性

在 Node.js 0.10 之前，可读流接口比较简单，但是功能还不完善。

- 之前的可读流不会等待调用 stream.read() 方法，data 事件会立即开始发射。须要做一些额外的工作，接收数据并保存到缓冲区，防止数据流失。
- stream.pause() 方法并不能保证使流暂停。这意味着即使是暂停的流，也有必要做好接收数据的准备。

在 Node.js v0.10 中添加了 Readable 类。为了和旧的 Node.js 程序兼容， 可读流在添加 data 事件监听器或者调用 stream.resume() 方法时会切换到流模式。现在即使不添加额外的 readable 事件监听或者调用 stream.read 方法，也不用担心数据丢失了。

虽然大部分的应用可以正常工作，但是在同时满足这些条件时会引入边界情况：

- 没有添加任何 data 事件的监听。
- stream.resume() 方法没有被调用。
- 流没有被 pipe 到任何可写流。

例如，考虑以下代码：

```
    // 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('The message was received but was not process
ed.\n');
      });

    }).listen(1337);
```

在 0.10 之前的版本，传入的数据会被直接丢弃，而在之后的版本中，socket 会保持在暂停模式。

解决这个问题的方法是在流开始的时候调用一次 `stream.resume()` 。

除了这样，还可以使用 `readable.wrap()` 方法包装 0.10 之前的可读流，来确保切换到流模式。

```
    // Workaround
    net.createServer((socket) => {

      socket.on('end', () => {
        socket.end('The message was received but was not process
ed.\n');
      });

      // start the flow of data, discarding it.
      socket.resume();

    }).listen(1337);
```

readable.read(0)

在某些情况下，有必要触发底层可读流机制的刷新，而不实际消耗任何数据。此时， 可以调用 `readable.read(0)` 方法，它总是返回null。

如果内部缓冲区大小低于 `highWaterMark`，并且流当前没有在读取数据，调用 `stream.read(0)` 将触发底层的 `stream._read()` 调用。

大多数应用都不需要这么做，但在Node.js中，特别是在可读流类内部，有这样做的情况，

readable.push('')

不推荐使用 `readable.push('')`。

将零字节字符串或缓冲区推送到非对象模式的流有一个有趣的副作用。因为它是对 readable.push() 的调用，调用将结束读取过程。 但是，因为参数是一个空字符串，所以并没有数据被添加到可读缓冲区，所以没有什么数据可供消费。

字符串解码器

接口稳定性: 2 - 稳定

使用该模块前，需要通过 `require('string_decoder')` 加载该模块。字符串解码器常用于将 Buffer 数据解码为 String。它是 `buffer.toString` 方法的一个简单实现，但额外提供了对 `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));
```

Class: StringDecoder

接受一个 `encoding` 参数，默认值为 `utf8`。

decoder.end()

返回所有留在缓冲区中的尾字节。

decoder.write()

返回解码后的字符串。

Timers

> 接口稳定性: 3 - 已锁定

`timer` 模块定义了一系列全局函数，无需使用 `require()` 即可使用。

Node.js 提供的 `timer` API 类似于浏览器提供的 `timer` API，但它们的内部实现是不同的——Node.js 的 `timer` 模块是基于 Node.js Event Loop 创建的。

Class: Immediate

clearImmediate(immediateObject)

取消一个 immediate 定时器。

clearInterval(intervalObject)

取消一个 interval 定时器。

clearTimeout(timeoutObject)

取消一个 timeout 定时器。

ref()

如果之前 `unref()` 了定时器，可以使用 `ref()` 请求定时器保持打开状态。如果之前已经使用 `ref()` 掉用过定时器，则再次调用无效。

该方法返回一个定时器。

setImmediate(callback[, arg][, arg][, ...])

该方法用于设定在 I/O 事件回调之后、在 setTimeout 和 setInterval 之前触发的定时器。返回值 `immediateObject` 常用于 `clearImmediate()` 方法取消 immediate 定时器。此外，该方法接收不定量的可选参数，这些参数会被传递给回调函数，作为回调函数的参数。

回调函数根据它们创建的顺序依次加入到回调队列之中。回调队列在事件循环中遍历执行，如果向正在执行的回调中加入一个 immediate 定时器，那么该定时器不会立即执行，而是等到下一轮事件循环时执行。

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

该方法用于设定每隔 `delay` 毫秒执行一次的 `callback` 回调函数。返回值 `intervalObject` 常用于 `clearInterval()` 方法取消 interval 定时器。此外，该方法接收不定量的可选参数，这些参数会被传递给回调函数，作为回调函数的参数。

根据浏览器的表现来看，如果 `delay` 大于 2147483647 毫秒或小于 1，该值将会被重置为 1。

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

该方法用于设定 `delay` 毫秒后执行 `callback` 回调函数。返回值 `timeoutObject` 常用于 `clearTimeout()` 方法取消 timeout 定时器。此外，该方法接收不定量的可选参数，这些参数会被传递给回调函数，作为回调函数的参数。

该回调函数延迟触发的时间可能并不等于 `delay`。Node.js 无法保证回调函数准确触发的时间以及触发的顺序，只能保证回调函数触发的时间尽可能接近 `delay` 时间。

根据浏览器的表现来看，如果 `delay` 大于 2147483647 毫秒或小于 1，则该定时器会被立即执行，且该值将会被重置为 1。

unref()

`setTimeout()` 和 `setInterval()` 方法返回的值都拥有一个实例方法 `timer.unref()`，该方法允许开发者创建一个定时器，但如果事件循环队列中只有这一个定时器，其将不会执行。如果定时器已经调用过 `unref()`，则再次调用时没有效果。

使用 `unref()` 创建的 `setTimeout` 定时器是一个独立的定时器，它会唤醒事件循环机制，创建过多的此类定时器会影响事件循环机制的性能。

该方法返回一个定时器。

TLS(SSL)

接口稳定性: 2 - 稳定

通过 `require('tls')` 可以加载该模块。

`tls` 模块使用 OpenSSL 为安全传输层和安全套接层提供了加密的流通讯。

TLS/SSL 是一个公钥/私钥基础架构。每一个客户端和服务器都必须拥有一个私钥。创建私钥的操作如下所示：

```
openssl genrsa -out ryans-key.pem 2048
```

所有的服务器和部分客户端需要拥有整数。证书是由认证中心或以自认证的方法签发的公钥。获取证书的第一步是创建一个 `证书签发请求（Certificate Signing Requrest, CSR）` 文件：

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

使用 CSR 以自认证的方式创建的证书：

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out
ryans-cert.pem
```

此外，开发者也可以将 CSR 文件发送给认证中心获取证书。

对于完全前向保密（Perfect forward secrecy），需要生成 Diffie-Hellman 参数：

```
openssl dhparam -outform PEM -out dhparam.pem 2048
```

创建 .pfx 或者 .p12：

```
openssl pkcs12 -export -in agent5-cert.pem -inkey agent5-key.pem
 \
      -certfile ca-cert.pem -out agent5.pfx
```

- `in`，整数
- `inkey`，私钥
- `certfile`，将所有的 CA 证书合并为一个文件，比如 `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

ALPN / NPN / SNI

ALPN (Application-Layer Protocol Negotiation Extension), NPN (Next Protocol Negotiation) 和 SNI (Server Name Indication) 都是 TLS 握手的扩展:

- `ALPN/NPN`，允许同一个 TLS 服务器使用多种协议，比如 HTTP，SPDY，HTTP/2
- `SNI`，允许同一个 TLS 服务器使用多个 SSL 整数不同的主机名

降低客户端发起的 renegotiation 攻击

TLS 协议允许客户端协商 TLS 会话的某些部分。不幸的是，会话协商需要大量的服务端资源，而这有可能让其成为服务器拒绝（denial-of-service）攻击的媒介。

为了降低这种潜在的危险，会话协商没十分钟只能发起三次。当超出限制时，`tls.TLSSocket` 的实例就会抛出一个错误。这项限制是可配置的：

- `tls.CLIENT_RENEG_LIMIT`，会话协商限制，默认值为 3
- `tls.CLIENT_RENEG_WINDOW`，会话协商窗口的限制时间，单位为秒，默认为 10 分钟

除非开发者了解正在做的事情，否则不要轻易修改默认配置。

为了测试服务器，可以通过 `openssl s_client -connect address:port` 连接到服务器，然后敲几次 `R<CR>`（字母 R + 回车键）看看。

修改默认的 TLS 加密套件

Node.js 内建了一套开启和禁用 TLS 加密的套件。在最新的 Node.js 中默认的加密套件是：

```
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
```

通过 `--tls-cipher-list` 参数在命令行中可以修改这些默认参数。举例来说，下面的命令将 `ECDHE-RSA-AES128-GCM-SHA256:!RC4` 设为了默认的 TLS 加密套件：

```
node --tls-cipher-list="ECDHE-RSA-AES128-GCM-SHA256:!RC4"
```

注意，Node.js 中默认的加密套件是精心筛选过的，通常具有最佳的安全性和最轻的风险。修改默认的加密套件有可能严重影响应用程序的安全性。除非绝对需要，否则不要使用 `--tls-cipher-list` 参数。

完全前向保密

`前向保密（Forward Secrecy）` 和 `完全前向保密（Perfect Fowward Secrecy）` 描述了密钥协商（比如密钥交换）方法的特点。事实上，即使服务器的私钥被窃取了，那么窃取者必须努力后去每次会话的密钥对才能完整破解会话。

完全正向保密本质上是通过每次握手时为密钥协商随机生成密钥对实现的。实现这一技术的方法被称为 `ephemeral`。

目前有两种方法可以实现完全前向保密：

- `DHE`，Diffie Hellman 密钥协商协议的 ephemeral 版本
- `ECDHE`，Elliptic Curve Diffie Hellman 密钥协商协议的 ephemeral 版本

ephermeral 的缺点主要集中在性能上，这是因为生成密钥的过程非常耗费资源。

Class: CryptoStream

接口稳定性: 0 - 已过时

使用 tls.TLSSocket 替代。

Class: SecurePair

该类可由 `tls.createSecurePair` 返回。

事件：'secure'

当 SecurePair 成功创阿金了安全连接之后就会触发该事件。

与检查服务器的 `secureConnection` 事件相似，`pair.cleartext.authorized` 常用于确认证书是否经过授权。

Class: tls.Server

该类是 `net.Server` 的子类，且两者拥有相同的方法。该类接收使用 TLS 或 SSL 加密过的连接而不是原始的 TCP 连接。

事件：'clientError'

- `function (exception, tlsSocket) {}`

如果客户端连接在安全建立之前触发了 `error` 事件，则客户端连接将会被转发到该方法。

`tlsSocket` 是错误的发起源 `tls.TLSSocket`。

事件：'newSession'

- `function (sessionId, sessionData, callback) {]`

该事件在创建 TLS 会话之后触发。可用于在外部储存中存储会话数据。最终必须调用 `callback`，否则安全连接将不会发送或接收任何数据。

注意，此类事件监听器只有在连接建立之后添加才会生效。

事件：'OCSPRequest'

- `function (certificate, issuer, callback) {}`

当客户端发送证书状态请求时就会触发该事件。开发者可以通过解析的服务器的整数获取 OCSP url 和整数 id，然后通过调用 `callback(null, resp)` 方法获取 OCSP 响应，其中 `resp` 是 Buffer 实例。`certificate` 和 `issuer` 都是最主要的的 DER-representations Buffer 实例和发行者的整数。它们可用于获取 OCSP 整数 id 和 OCSP 终结点 url。

此外，还可以调用 `callback(null, null)`，表示没有 OCSP 响应。

典型流程：

1. 客户端连接服务器并发送 `OCSPRequest`（通过 ClientHello 中的状态信息扩展）
2. 服务器接收请求并调用 `OCSPRequest` 事件监听器
3. 服务器根据 certificate 或 issuer 抓取 OCSP url，并对 CA 执行 OCSP 请求
4. 服务器从 CA 接收 `OCSPResponse` 并将通过 `callback` 发送给客户端
5. 客户端验证响应，要么销毁 socket 要么执行一次握手

注意，如果证书是自认证（self-signed）或发证者不在根证书列表中，那么 `issuer` 的值可能是 `null`。

注意，此类事件监听器只有在连接建立之后添加才会生效。

注意，开发者可以通过 npm 模块解析证书，比如 asn1.js。

事件：'resumeSession'

- `function (sessionId, callback) {}`

当客户端要恢复先前的 TLS 会话时就会触发该事件。事件监听器也许会根据指定的 `sessionId` 到外部储存中查找，一旦查找完成就会调用 `callback(null, sessionData)` 。如果会话不能恢复（比如不存在这个会话），那么系统可能会调用 `callback(null, null)` 。调用 `callback(err)` 将会中断连接并销毁 socket。

注意，此类事件监听器只有在连接建立之后添加才会生效。

下面代码演示了如何恢复 TLS 会话：

```
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);
});
```

事件：'secureConnection'

- `function (tlsSocket) {}`

当新连接成功握手后会触发该事件，参数 `tslSocket` 是 `tls.TLSSocket` 的实例，该参数拥有 stream 的常见方法和事件。

`socket.authorized` 是一个布尔值，用于表示客户端是否已经被服务器的证书中心校验通过。如果 `socket.authorized` 的值为 false，那么就会抛出 `socket.authorizationError` ，用以说明授权失败。值得提醒的是：开发者的链接是否可以被服务器接收，很大因素取决于 TLS 服务器的配置，也就是说，开发者未被授权的链接有可能会被接收。

`socket.npnProtocol` 是一个字符串，包含了已选择的 NPN 协议，`socket.alpnProtocol` 也是一个字符串，包含了已选择的 ALPN 协议。当同时接收到 NPN 和 ALPN 扩展时，ALPN 的优先级高于 NPN，且下一个协议由 ALPN 选择。当 ALPN 没有可选择的协议时，返回 false。

`socket.servername` 是一个字符串，包含了 SNI 请求的服务器名。

server.addContext(hostname, context)

当客户端请求的 SNI 主机名匹配 `hostanme` 时，该方法添加的 context 就会生效。`context` 包含 key / cert / ca 和其他来自 `tsl.createSecureContext()` 中 `options` 参数所包含的属性。

server.address()

该方法返回操作系统记录的绑定地址、地址族和服务器接口。

server.close([callback])

该方法中断服务器接收新的链接。该方法是异步执行的，当服务器触发 `close` 事件后，服务器将中断和退出执行。此外，开发者可以向 `close` 事件传递一个 `callback` 作为监听器。

server.connections

该属性是一个数值，表示服务器的并发连接数。

server.getTicketKeys()

该方法返回一个 Buffer 实例，示例中包含 TLS Session Tickets 加密和解密所用到的密钥。

server.listen(port[, hostname][, callback])

该方法根据指定的 `port` 和 `hostname` 接收连接。当没有传入 `hostname` 时，如果 IPv6 可用，那么服务器就会接收任何来自 IPv6 地址（ `::` ）的连接，否则接收 IPv4 地址（ `0.0.0.0` ）。如果 `port` 的值为 0，则系统分配一个随机端口。

该方法是异步执行的，当服务器被绑定后系统会调用回调函数 `callback` 。

server.setTicketKeys(keys)

该方法用于更新 TLS Session Tickets 加密和解密所用到的密钥。

注意，buffer 的大小必须为 48 字节，更多使用信息请查看 `ticketKeys` 的可选配置。

注意，修改后的密钥只对新的服务器连接有效，现有的或当前正在执行的服务器连接不受影响。

server.maxConnections

该属性可以限制服务器的连接数。

Class: tls.TLSSocket

该类是 `net.Socket` 的包装类，用于处理写入数据的透明传输和所有的 TLS 会话协商。

`tls.TLSSocket` 的实例实现双工 Stream 接口，该实例拥有 stream 实例常见的方法和事件。

当连接打开时，返回 TLS 连接原数据的该类成员方法只会返回数据。

new tls.TLSSocket(socket[, options])

该构造器根据传入的 TCP socket 创建一个新的 TLSSocket。

参数 `socket` 是 net.Socket 的实例。

可选参数 `options` 是一个对象，包含以下属性：

- `secureContext` ，来自 tls.createSecureContext() 的 TLS 上下文对象
- `isServer` ，如果值为 true，则 TLS socket 将会在服务器模式下初始化，默认值是 false
- `server` ，可选的额 net.Server 实例
- `requestCert` ，可选，参见 tls.createSecurePair()
- `rejectUnauthorized` ，可选，参见 tls.createSecurePair()

- `NPNProtocols`，可选，参见 tls.createServer()
- `ALPNProtocols`，可选，参见 tls.createServer()
- `SNICallback`，可选，参见 tls.createServer()
- `session`，可选，一个包含 TLS 会话的 Buffer 实例
- `requestOCSP`，可选，如果值为 `true`，则 OCSP 状态请求扩展将会被添加到 client hello 中，并在安全通讯创建之前在 socket 上触发 `OCSPResponse` 事件

事件：'OCSPRespnse'

- `function (response) {}`

如果设置了 `requestOCSP` 属性就会触发该事件。`response` 是一个 Buffer 对象，包含了服务器的 OCSP 响应。

传统上来说，`response` 是一个服务器 CA 签发的签发对象，包含了服务器证书吊销状态等信息。

事件：'secureConnect'

当连接成功握手后会触发该事件。无论服务器的证书是否授权，都会调用该事件监听器。如果是否测试 `tlsSocket.authorized` 来查看服务器证书的合法性完全取决于开发者。如果 `tlsSocket.authorized === false`，那么就可以从 `tlsSocket.authorizationError` 查看错误信息。此外，如果使用了 ALPN 或 NPN，那么可以通过 `tlsSocket.alpnProtocal` 或 `tlsSocket.npnProtocol` 来查看会话协商协议。

tlsSocket.address()

该方法返回一个对象，包含操作系统提供的绑定地址、地址族和服务器端口等信息。常用于查找那个端口已被系统绑定。该对象的常见格式：`{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`。

tlsSocket.authorized

该属性是一个布尔值，如果对等实体证书（peer certificate）由指定的 CA 签发，那么就会返回 true，否则返回 false。

tlsSocket.authorizationError

该属性表示对等实体证书验证失败的原因。只有当 `tlsSocket.authorized === false` 时，该属性才会可用。

tlsSocket.encrypted

该属性是一个静态的布尔值，总是等于 true。常用于区分 TLS socket 和常规对象。

tlsSocket.getCipher()

该方法返回一个对象，用于白哦是加密名和第一个定义该加密名的 SSL/TLS 的协议版本，比如 `{ name: 'AES256-SHA', version: 'TLSv1/SSLv3' }`.

更多信息请参考 https://www.openssl.org/docs/manmaster/ssl/SSL_CIPHER_get_name.html 中的 `SSL_CIPHER_get_name()` 和 `SSL_CIPHER_get_version()`。

tlsSocket.getEphemeralKeyInfo()

该方法返回一个对象，表示客户端完全前向保密连接的 ephemeral 密钥交换参数中的类型、名称和大小。如果不是 ephemeral 密钥交换则返回一个空对象。由于只支持客户端 socket，所以如果在服务器 socket 上调用该方法则会返回 null。该方法支持的类型包括 `DH` 和 `ECDH`。`name` 属性只在 `ECDH` 中可用：

```
{ type: 'ECDH', name: 'prime256v1', size: 256 }
```

tlsSocket.getPeerCertificate([detailed])

该方法返回一个包含对等实体证书的对象，该对象包含的属性与证书的内容相对应。如果 `detailed === true`，则返回包含 `issuer` 属性的完整链路，如果返回的只是没有 issuer 属性的顶级证书，则返回 false。

```
{ 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'
}
```

如果对等实体没有提供证书，则返回 null 或空对象。

tlsSocket.getProtocol()

该方法返回一个字符串，表示当前连接会话协商后的 SSL/TLS 协议版本。如果已连接 socket 没有完整的握手进程，则返回 `unknown`；如果是服务器连接或未连接的客户端 socket，则返回 null：

```
'SSLv3'
'TLSv1'
'TLSv1.1'
'TLSv1.2'
'unknown'
```

更多信息请查看
https://www.openssl.org/docs/manmaster/ssl/SSL_get_version.html。

tlsSocket.getSession()

该方法返回使用 ASN.1 加密后的 TLS 会话，如果没有经过会话协商，则返回 `undefined`。当重连到服务器时，使用 Cloud 可以加快握手过程。

tlsSocket.getTLSTicket()

注意，该方法只对客户端 TLS socket 有效，常用于调试程序，通过向 `tls.connect()` 传递 `session` 参数可以复用会话。

该方法返回使用 TLS session ticket，如果没有经过会话协商，则返回 `undefined`。

tlsSocket.localAddress

该属性是一个字符串，表示本地 IP 地址。

tlsSocket.localPort

该属性是一个数值，表示本地端口。

tlsSocket.remoteAddress

该属性是一个字符串，表示远程 IP 地址，比如 `74.125.127.100` 或 `2001:4860:a005::68`。

tlsSocket.remoteFamily

该属性是一个字符串，表示远程 IP 族，比如 `IPv4` 或 `IPv6`。

tlsSocket.remotePort

该属性是一个数值，表示远程端口，比如 `443`。

tlsSocket.renegotiate(options, callback)

该方法用于初始化 TLS 会话协商进程，其中 `options` 参数包含了 rejectUnauthorized、requestCert 字段。如果会话协商成功，则 `callback(err)` 中的 `err` 的值为 `null`。

注意，在安全连接建立之后，该方法可用于请求对等实体证书。

注意，当在服务端运行时，如果超过了 `handshakeTimeout` 的限制时间，则 socket 会被销毁并抛出错误。

tlsSocket.setMaxSendFragment(size)

该方法用于设置 TLS 片段的最大值，默认值和最大值都是 16384，最小值是 512，如果设置成功则返回 true，否则返回 false。

片段体积越小，则客户端缓存的等待时间越短：大体积的片段会被 TLS 层缓存，直到所有的片段都被接收且通过了完整性测试；大体积的片段需要多次往返传输，且处理进程会由于保丢失或重排序等原因被延迟。不过，小体积的片段增加了 TLS 帧字节和 CPU 负载，这也会降低服务器的吞吐量。

tls.connect(options[, callback])

tls.connect(port[, host][, options][, callback])

该方法根据 `port` 和 `host` 或 `options.port` 和 `options.host`（如果没有传入 host 信息，则使用 localhost）创建一个新的客户端连接。`options` 是一个包含以下属性的对象：

- `host`，客户端需要连接的主机
- `port`，客户端需要连接的端口
- `socket`，根据指定的 socket 创建安全连接，而不是创建新的 socket。如果指定了该参数，则系统会忽略 host 和 port 参数
- `path`，创建连接到该路径的 Unix socket，如果指定了该参数，则系统会忽略 host 和 port 参数
- `pfx`，该参数是一个字符串或 Buffer 实例，包含了 PFX 或 PKCS12 格式的客户端私钥、证书和 CA 证书
- `key`，该参数是一个字符串或 Buffer 实例，包含了 PEM 格式的客户端私钥
- `passphrase`，该参数是一个字符串格式的私钥或 pfx 密码

- `cert`，该参数是一个字符串或 Buffer 实例，包含了 PEM 格式的客户端私钥
- `ca`，该参数是一个 PEM 格式的字符串、Buffer 实例、字符串数组，用于表示可信证书。如果没有设置该参数，则系统使用众多周知的几个根 CA，比如 VeriSign。它们常用于授权连接。
- `ciphers`，该参数是一个字符串，指定了使用或不使用的密码，使用 `:` 分隔。使用的默认密码和 `tls.createServer()` 一样
- `rejectUnauthorized`，如果值为 true，则根据指定的 CA 列表校验服务器整数。如果校验失败，则触发 `error` 事件，其中 `err.code` 表示 OpenSSL 的错误代码，默认值为 true
- `NPNProtocols`，该参数是一个字符串数组或 Buffer 实例，表示支持的 NPN 协议。Buffer 实例需要遵循类似 `0x05hello0x05world` 的格式，即第一个字节是下一个协议名的长度（传入的数组则简单多了：`['hello', 'world']`）。
- `ALPNProtocols`，该参数是一个字符串数组或 Buffer 实例，表示支持的 ALPN 协议。Buffer 实例需要遵循类似 `0x05hello0x05world` 的格式，即第一个字节是下一个协议名的长度（传入的数组则简单多了：`['hello', 'world']`）。
- `servername`，用于 SNI TLS 扩展的服务器名字
- `checkServerIndentity(servername, cert)`，该方法覆盖了系统的相关功能，用来根据证书检查服务器的主机名。如果校验失败，则返回一个错误，否则返回 udnefined
- `secureProtocol`，使用的 SSL 方法，比如 `SSLv3_method` 强制使用 SSL 第三版。该属性的值取决于安装的 OpenSSL 和 SSL_METHODS 常量
- `secureContext`，一个来自 `tls.createSecureContext(...)` 的可选 TLS 上下文对象。可用于缓存客户端证书、密钥和 CA 证书
- `session`，一个包含 TLS 会话的 Buffer 实例
- `minDHSize`，允许接收的 TLS 连接的 DH 参数的最小字节量。如果服务器提供的 DH 参数小于该值，则 TLS 连接被销毁并抛出错误，默认值为 1024

回调函数 `callback` 会被绑定为 `secureConnect` 事件的事件处理器。

`tls.connect()` 返回一个 `tls.TLSSocket` 对象。

下面代码演示了一个 echo 服务器：

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

const options = {
  // These are necessary only if using the client certificate au
thentication
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // This is necessary only if the server uses the self-signed c
ertificate
  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();
});
```

或者：

```
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(details)

该方法用于创建一个凭证对象，其中 `details` 包含以下属性:

- `pfx` ，该属性是一个字符串或 Buffer 实例，包含了 PFX 或 PKCS12 格式的客户端私钥、证书和 CA 证书
- `key` ，该属性是一个字符串或 Buffer 实例，包含了 PEM 格式的客户端私钥。如果值为数组，则支持多个密钥使用不同的算法。该值也可以是纯密钥数组，或对象数组，比如 `{pem: key, passphrase: passphrase}`
- `passphrase` ，该属性是一个字符串格式的私钥或 pfx 密码
- `cert` ，该属性是一个字符串或 Buffer 实例，包含了 PEM 格式的客户端私钥
- `ca` ，该属性是一个 PEM 格式的字符串、Buffer 实例、字符串数组，用于表示可信证书。如果没有设置该属性，则系统使用众多周知的几个根 CA，比如 VeriSign。它们常用于授权连接。
- `crl` ，该属性要么是一个字符串，要么是一个 PEM 格式的 CRL 列表

- `ciphers`，该属性是一个字符串，指定了使用或不使用的密码，更多有关格式的信息请参考 https://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT
- `honorCipherOrder`，当选择加密算法时，使用服务器配置替代客户端配置。

如果没有指定 `ca`，那么 Node.js 就会使用默认的、来自 http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt CA 公信列表。

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])

该方法新建一个包含两个 stream 的安全对对象，其中一个用于读写加密数据，一个用于读写明文数据。通常来说，加密 stream 用于传递或接受加密数据流，明文 stream 用于替代初始化加密 stream。

- `credentials`，一个来自 `tls.createSecureContext()` 的安全上下文对象
- `isServer`，该参数是一个布尔值，用于表示是否以服务器或客户端代开当前的 tls 连接
- `requestCert`，该参数是一个布尔值，用于表示服务器是否应该从连接的客户端请求证书，只对服务器连接有效。
- `rejectUnauthorized`，该参数是一个布尔值，用于表示服务器是否应该自动拒绝携带无效证书的客户端，只对开启 `requestCert` 选项的服务器有效
- `options`，一个包含常见 SSL 选项的对象

`tls.createSecurePair()` 返回一个 SecurePair 对象，该对象包含了 `cleartext` 和 `encrypted` stream 属性。

注意，`cleartext` 和 `tls.TLSSocket` 具有相同的 API。

tls.createServer(options[, secureConnectionListener])

该方法用于创建一个新的 `tls.Server`。`secureConnectionListener` 会被绑定为 `secureConnection` 事件的监听器。`options` 对象包含以下属性：

- `pfx`，该属性是一个字符串或 Buffer 实例，包含了 PFX 或 PKCS12 格式的客户端私钥、证书和 CA 证书
- `key`，该属性是一个字符串或 Buffer 实例，包含了 PEM 格式的客户端私钥。如果值为数组，则支持多个密钥使用不同的算法。该值也可以是纯密钥数组，或对象数组，比如 `{pem: key, passphrase: passphrase}`
- `passphrase`，该属性是一个字符串格式的私钥或 pfx 密码
- `cert`，该属性是一个字符串或 Buffer 实例，包含了 PEM 格式的客户端私钥
- `ca`，该属性是一个 PEM 格式的字符串、Buffer 实例、字符串数组，用于表示可信证书。如果没有设置该属性，则系统使用众多周知的几个根 CA，比如 VeriSign。它们常用于授权连接。
- `crl`，该属性要么是一个字符串，要么是一个 PEM 格式的 CRL 列表
- `ciphers`，该参数是一个字符串，指定了使用或不使用的密码，使用 `:` 分隔，默认的加密套件包括：

```
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
```

默认的加密套件使用 GCM 支持 Chrome's modern cryptography setting，使用 ECDHE 和 DHE 加密完全前向保密，并提供了一些向后兼容性。

考虑到 specific attacks affecting larger AES key sizes，128 位的 AES 加密算法要优于 192 位或 256 位的 AES。

对于那些使用了不安全或已抛弃的加密算法（RC4、基于 DES）并不能使用默认的配置完成握手。如果开发者一定要支持这些浏览器，可以参考 TLS 建议的兼容性加密套件，更多信息请参考 OpenSSL cipher list format documentation。

- `ecdhCurve`，该参数是一个字符串，表示一个用于 ECDH 密钥协商的 named curve，或者是一个 false，表示禁用 ECDH。默认值是 `prime256v1`（NIST P-256）。使用 `crypto.getCurves()` 可以后去可用的 curve name 列表。在最新的版本中，使用 `openssl ecparam -list_curves` 命令也可以显示可用 elliptic curve 的名字和描述信息。
- `dhparam`，该参数是一个包含 Diffie Hellman 参数的字符串或 Buffer 实例，用于完全前向保密。使用 `openssl dhparam` 可以创建该参数。它的密钥长度要大于等于 1024 位，否则系统会抛出异常。强烈建议使用 2048 位或更长的长度增强安全性。如果忽略了该属性或者为无效属性，则会被系统忽略且不能使用 DHE 加密算法。
- `handshakeTimeout`，如果 SSL/TLS 握手时间超时就会终止连接，默认值为 120 秒。当握手超时时，`tls.Server` 对象会触发一个 `clientError` 的事件
- `honorCipherOrder`，如果值为 true，则在选在加密算法时，使用服务器的配置替代客户端的配置
- `requestCert`，如果值为 true，则服务器将会请求客户端发送证书并校验证书，默认值为 false
- `rejectUnauthorized`，如果值为 true，则根据指定的 CA 列表校验服务器证书。只有当 `requestCert` 的值为 true 时，该参数才会生效，默认值为 false
- `NPNProtocols`，该参数是一个字符串数组或 Buffer 实例，表示支持的 NPN 协议。
- `ALPNProtocols`，该参数是一个字符串数组或 Buffer 实例，表示支持的 ALPN 协议。当服务器同时从客户端接收到 NPN 和 ALPN 扩展时，ALPN 的优先级高于 NPN，且服务器不会向客户端发送 NPN 扩展
- `SNICallback(serveranme, cb)`，如果客户端支持 SNI TLS 扩展就会调用该方法。该方法接收两个参数：`servername` 和 `cb`。`SNICallback` 应该调用 `cb(null, ctx)`，其中 `ctx` 是一个 SecureContext 实例。如果没有指定 `SNICallback`，系统就会使用默认的高阶 API 回调函数。
- `sessionTimeout`，该参数是一个整数，用于指定服务器创建 TLS 会话标识

符和 TLS session tickets 的超时时间。

- `ticketKeys`，该参数是一个 48 字节的 Buffer 实例，由 16 字节的前缀、16 字节的 hmac 密钥和 16 字节的 AES 密钥组成。开发者可以使用它从多个 tls 服务器接收 session tickets。注意，该属性会自动在集群模块的 worker 之间共享。
- `sessionIdContext`，该参数是一个字符串，包含了一个不透明的会话恢复标识符。如果 `requestCert === true`，则默认值为命令行生成的 MD5 哈希值（在 FIPS 模式下，使用缩减的 SHA1 哈希值）。否则，不提供默认值。
- `secureProtocol`，使用的 SSL 方法，比如 `SSLv3_method` 强制使用 SSL 第三版。该属性的值取决于安装的 OpenSSL 和 SSL_METHODS 常量

下面代码演示了一个 echo 服务器：

```
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 auth
entication.
  requestCert: true,

  // This is necessary only if the client uses the self-signed c
ertificate.
  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');
});
```

或者：

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

const options = {
  pfx: fs.readFileSync('server.pfx'),

  // This is necessary only if using the client certificate auth
entication.
  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');
});
```

使用 `openssl s_client` 可以连接到服务器进行测试：

```
openssl s_client -connect 127.0.0.1:8000
```

tls.getCiphers()

该方法返回一个数组，数组内包含系统支持的 SSL 加密算法的名称：

```
var ciphers = tls.getCiphers();
console.log(ciphers); // ['AES128-SHA', 'AES256-SHA', ...]
```

TTY

> 接口稳定性: 2 - 稳定

TTY 模块包含 `tty.ReadStream` 和 `tty.WriteStream` 两个类。在大多数情况下，开发者都需要直接调用该模块。

当 Node.js 检测到当前环境处于 TTY 上下文时，则 `process.stdin` 会指向一个 `tty.ReadStream` 实例，`process.stdout` 会指向一个 `tty.WriteStream` 实例。检测 Node.js 是否处于 TTY 上下文有一个更方便的方式，那就是检查 `process.stdout.isTTY` ：

```
$ node -p -e "Boolean(process.stdout.isTTY)"
true
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

Class: ReadStream

该类是 `net.socket` 的子类，表示 TTY 中的可读部分。在大多数情况下，任何 Node.js 程序（ `isatty(0)` 为 true）中的 `process.stdin` 都是唯一的 `tty.ReadStream` 实例。

rs.isRaw

布尔值，默认值为 false，标示当前 `tty.ReadStream` 实例是否为 `raw` 状态。

rs.setRawMode(mode)

`mode` 的值必须为 true 或者 false，该值决定了 `tty.ReadStream` 为原始设备或默认设置，并且它也决定了 `isRaw` 的值。

Class: WriteStram

该类是 `net.socket` 的子类，表示 TTY 中的可写部分。在大多数情况下，任何 Node.js 程序（ `isatty(0)` 为 true）中的 `process.stdout` 都是唯一的 `tty.WriteStream` 实例。

事件：'resize'

`columns` 或 `rows` 属性变化时， `refreshSize()` 方法就会触发该事件：

```
process.stdout.on('resize', () => {
  console.log('screen size has changed!');
  console.log(`${process.stdout.columns}x${process.stdout.rows}`
);
});
```

ws.columns

数值，标示 TTY 当前的列数，该属性会被 `resize` 事件更新。

ws.rows

数值，标示 TTY 当前的行数，该属性会被 `resize` 事件更新。

tty.isatty(fd)

如果 `fd` 参数和终端有关，那么该方法返回 `true` ，否则返回 `false` 。

tty.setRawMode(mode)

接口稳定性: 0 - 已过时

该方法已被废除，请使用 `tty.ReadStream#setRawMode()` 替代，比如 `process.stdin.setRawMode`。

UDP / Datagram Sockets

接口稳定性: 2 - 稳定

`dgram` 模块封装了对 UDP Datagram sockets 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.por
t}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

Class: dgram.Socket

`dgram.Socket` 是一个封装了和数据报相关的函数的 `EventEmitter` 实例对象。

使用 `dgram.createSocket()` 可以创建新的 `dgram.Socket` 实例，且不需要使用 `new` 关键字。

事件：'close'

当 socket 使用 `close()` 关闭之后就会触发 `close` 事件。一旦触发该事件，socket 就不再触发 `message` 事件。

事件：'error'

- `exception`，Error 实例

当出现任意错误时就会触发 `error` 事件，该事件的监听器函数只接收一个错误对象。

事件：'listening'

当 socket 开始监听数据报信息时就会触发 `listening` 事件，且在 UDP socket 创建时就可以触发该事件。

事件：'message'

- `msg`，Buffer 实例，消息
- `rinfo`，对象，远程地址信息

当 socket 接收到新的数据报时就会触发 `message` 事件。该事件的处理函数接收两个参数：`msg` 和 `rifno`，其中 `msg` 参数是一个 Buffer 实例，`rinfo` 参数是一个对象，该对象包含了发报者的地址属性 address/family/port：

```
socket.on('message', (msg, rinfo) => {
  console.log('Received %d bytes from %s:%d\n', msg.length, rinf
o.address, rinfo.port);
});
```

socket.addMembership(multicastAddress[, multicastInterface])

- `multicastAddress`，字符串
- `multicastInterface`，字符串

该方法根据给定的 `multicastAddress`（使用 `IP_ADD_MEMBERSHIP` socket 选项）通知 kernel 加入广播组。如果未指定 `multicastInterface` 参数，操作系统会尝试为所有的有效网络接口添加广播关系。

socket.address()

该方法返回一个包含 socket 地址信息的对象。对于 UDP socket，该对象包含 address、family 和 port 属性。

socket.bind([port][, address][, callback])

- `port`，整数
- `address`，字符串
- `callback`，函数，不接收任何参数，完成绑定时调用

对于 UDP socket，该方法用于让 `dgram.Socket` 监听 `port` 和 `address` 指定接口的数据报信息。如果未指定 `port`，则操作系统绑定一个随机接口。如果未指定 `address`，则操作系统会尝试监听所有的地址。绑定完成之后会立即出发 `listening` 事件并调用 `callback` 回调函数。

注意，同时为 `socket.bind()` 设置 `listening` 事件监听器和 `callback` 回调函数虽然没有坏处，但略显多余。

socket 绑定数据报之后就会让 Node.js 进程持续运行以接收数据报信息。

如果绑定失败，则触发 `error` 事件。极少数情况下，会抛出一个 Error 实例，比如尝试绑定一个已关闭的 socket。

下面代码演示了监听 41234 端口的 UDP 服务器：

```
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.por
t}`);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

socket.bind(options[, callback])

- `options`，对象，包含以下属性
  - `port`，数值
  - `address`，字符串
  - `exclusive`，布尔值
- `callback`，函数

对于 UDP socket，该方法用于让 `dgram.Socket` 监听 `options` 参数中 `port` 和 `address` 所指定的接口的数据报信息。如果未指定 `port`，则操作系统绑定一个随机接口。如果未指定 `address`，则操作系统会尝试监听所有的地址。绑定完成之后会立即出发 `listening` 事件并调用 `callback` 回调函数。

只有 `dgram.Socket` 配合 `cluster` 模块使用时，`options` 对象中的可选属性 `exclusive` 才有效。如果 `exclusive` 的值为 false（默认值），cluster worker 将会使用相同的底层 socket 句柄，并允许连接处理共享的任务；如果

`exclusive` 的值为 true，则不允许共享句柄，即使共享端口也会抛出错误。

下面代码演示了 socket 监听独享端口的示例：

```
socket.bind({
  address: 'localhost',
  port: 8000,
  exclusive: true
});
```

socket.close([callback])

该方法用于关闭底层 socket 并停止监听数据。如果指定了 `callback`，则该回调函数会被系统自动添加为 `close` 事件的监听器。

socket.dropMembership(multicastAddress[, multicastInterface])

- `multicastAddress`，字符串
- `multicastInterface`，字符串

该方法根据给定的 `multicastAddress`（使用 `IP_ADD_MEMBERSHIP` socket 选项）通知 kernel 离开广播组。当 socket 关闭或者进程终端时，kernel 会自动调用该方法，所以对于大多数的应用来说，基本不需要主动调用该方法。

如果未指定 `multicastInterface` 参数，操作系统会尝试为所有的有效网络接口移除广播关系。

socket.send(msg[, offset, length], port, address[, callback])

- `msg`，Buffer 实例或字符串或数组，用于发送的消息
- `offset`，整数，指定在 Buffer 实例中数据发送的起始位置
- `length`，整数，消息的字节量
- `port`，整数，分发端口
- `address`，字符串，分发主机名或 IP 地址
- `callback`，函数，消息发送之后调用的回调函数

该方法通过 socket 广播数据报，使用时必须指定 `port` 和 `address`。

`msg` 参数包含了发送的消息，且根据 `msg` 的类型不同，系统会做不同的处理。如果 `msg` 是一个 Buffer 实例，那么 `offset` 和 `length` 属性就用来指定数据发送的起始位置及长度。如果 `msg` 是一个字符串，则会被系统自动转换为 `utf8` 格式的 Buffer 实例。当消息中含有多字节字符时，`offset` 和 `length` 参数指定的就不再试字符位置了，而是字节长度。如果 `msg` 是一个数组，那么一定不要指定 `offset` 和 `length

`address` 参数是一个字符串。如果 `address` 是一个主机名，那么 DNS 就会解析该主机的地址；如果未指定 `address` 或者该参数是一个字符串，那么系统就会自动为其赋值为 `127.0.0.1` 或 `::1`。

如果 socket 之前没有通过 `bind()` 方法进行绑定，那么 socket 就会被分配一个随机接口并绑定到所有的接口地址，比如在 udp4 socket 中是 `0.0.0.0`，在 udp6 socket 中是 `::0`。

可选参数 `callback` 可用于记录 DNS 错误或指定何时复用 `buf` 对象。注意，DNS 查询会推迟消息发送的是时间，这段延迟多于 Node.js 事件循环一次的时间。

确定数据报被发送的唯一方式就是使用 `callback`。如果发送过程中出现了错误且指定了 `callback`，那么 `callback` 的第一个参数就包含了该错误。如果未指定 `callback`，那么在 `socket` 对象上就会触发 `error` 事件。

`offset` 和 `length` 是可选参数，但如果指定了其中之一，那么就必须指定另一个。此外，只有 `msg` 是 Buffer 实例或字符串时，才可以传递该参数。

下面代码演示了如何使用 `localhost` 上的随机端口发送 UDP 包：

```
const dgram = require('dgram');
const message = new Buffer('Some bytes');
const client = dgram.createSocket('udp4');
client.send(message, 41234, 'localhost', (err) => {
  client.close();
});
```

下面代码演示了如何使用 `localhost` 上的随机端口发送包含多个 Buffer 实例的 UDP 包：

```
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 实例的快慢取决于开发者的应用程序和操作系统，所以开发者需要测试一下相关的性能。通常来说，这种方式还是比较快的。

关于 UDP 数据报的长短

一个 `IPv4/6` 数据报的最大长度取决于 `MTU`（Maximum Transmission Unit）和 `Payload Length` 的值：

- `Payload Length` 的长度为 16 个字节，这意味着一个包含网络头信息和数据（65,507 bytes = 65,535 − 8 bytes UDP header − 20 bytes IP header）的有效负荷不能超过 64K。这对于环回接口（loopback interfaces）没有什么问题，但是对于大多数的主机和网络就不那么够用了。
- `MTU` 是链路层对数据报消息支持的最大长度。对于 IPv4 连接，无论是完整接收还是碎片接收，允许的 MTU 最小值是 68 个八位字节，IPv4 的建议值为 576 个八位字节。对于 IPv6，最小值是 1280 个八位字节，最小片段重组缓存值是 1500 个八位字节。68 个八位字节太小了，所以对于大多数的链路层最小值都是 1500 个八位字节。

开发者无法预先知道一个包所经过的所有链接的 MTU。如果发送的数据报大于接收者的 MTU 上限，那么包就会被静默丢弃，既没有任何反馈也不会到大目的地。

socket.setBroadcast(flag)

- `flag`，布尔值

该方法用于设置请清除 socket 的 `SO_BROADCAST` 选项。如果值为 true，则 UDP 包可能会被发往本地的广播地址。

socket.setMulticastLoopback(flag)

- `flag`，布尔值

该方法用于设置请清除 socket 的 `SO_BROADCAST` 选项。如果值为 true，则组播包也会被本地接口接收。

socket.setMulticastTTL(ttl)

- `ttl`，整数

该方法用于设置 `IP_MULTICAST_TTL`。虽然通常来说 TTL 表示生存时间 `Time to Live`，但在这里它表示一个包允许经过的 IP 跳跃点数量。每一个路由或网关都会减少包的 TTL 值，如果 TTL 的值为 0，那么该包将不会被转发。通常使用网络探测器或多播时会修改 TTL 的值。

`socket.setTTL()` 的参数是 1~255 之间的数值，用于表示跳跃点的数量，在大多数系统上默认值为 64 且可以被人为的改动。

socket.ref()

默认情况下，绑定 socket 会从 socket 创建到打开的过程阻塞 Node.js 进行的执行。`socket.unref()` 方法可用于从引用计数中去除 socket，以保持 Node.js 进程的活跃。`socket.ref()` 方法用于将 socket 添加到引用计数中并回复为默认行为。

反复调用 `socket.ref()` 并不会有额外的效果。

由于 `socket.ref()` 返回一个 socket 的引用，所以该方法之后可以进行链式调用。

socket.unref()

默认情况下，绑定 socket 会从 socket 创建到打开的过程阻塞 Node.js 进行的执行。`socket.unref()` 方法可用于从引用计数中去除 socket，以保持 Node.js 进程的活跃，并且即使 socket 仍在监听，也可以退出进程。

反复调用 `socket.unref()` 并不会有额外的效果。

由于 `socket.unref()` 返回一个 socket 的引用，所以该方法之后可以进行链式调用。

异步 socket.bind()

从 Node.js v0.10 开始，`dgram.Socket#bind()` 方法被修改成了异步执行的方法。类似如下的同步执行代码：

```
const s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

必须修改成接收回调函数的方法：

```
const s = dgram.createSocket('udp4');
s.bind(1234, () => {
  s.addMembership('224.0.0.114');
});
```

dgram 模块方法

dgram.createSocket(options[, callback])

- `options`，对象
- `callback`，函数，和 `message` 事件所绑定
- 返回值类型：dgram.Socket 实例

该方法用于创建一个 `dgram.Socket` 对象。`options` 参数是一个包含 `type` 必选属性，`udp4` 或 `udp6` 两者之一以及可选布尔属性 `reuseAddr` 的对象。

如果 `reuseAddr` 的值为 true，即使其他进程已经绑定了 socket，`socket.bind()` 也可以复用地址。`reuseAddr` 的默认值为 false。可选参数 `callback` 会和 `message` 事件绑定。

一旦创建了 socket，调用 `socket.bind()` 就会让 socket 开始监听数据报消息。当 `socket.bind()` 没有接收到 `address` 和 `port` 参数时，系统就会将 socket 绑定到所有的接口地址的随机端口上。使用 `socket.address().address` 和 `socket.address().port` 可以获取相应的绑定地址和端口信息。

dgram.createSocket(type[, callback])

- `type`，字符串，udp4 和 udp6 中的一个
- `callback`，函数，和 `message` 事件绑定
- 返回值类型：dgram.Socket 实例

该方法根据指定的 `type` 创建一个 `dgram.Socket` 对象。`type` 参数可以是 udp4 和 udp6 中的一个。可选参数 `callback` 会和 `message` 事件绑定。

一旦创建了 socket，调用 `socket.bind()` 就会让 socket 开始监听数据报消息。当 `socket.bind()` 没有接收到 `address` 和 `port` 参数时，系统就会将 socket 绑定到所有的接口地址的随机端口上。使用 `socket.address().address` 和 `socket.address().port` 可以获取相应的绑定地址和端口信息。

URL

该模块提供了解析 URL 的辅助函数，使用 `require('url')` 调用。

解析 URL

解析后的 URL 对象通常具有以下的部分或全部字段。如果 URL 字符串中没有，则解析后的 URL 对象也不会存在，下面是一个 URL 的示例：

```
'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'
```

- `href`，待解析的、完整的 URL，其中协议和主机都是小写形式，举例：'http://user:pass@host.com:8080/p/a /t/h?query=string#hash'
- `protocol`，请求协议，举例：'http:'
- `slashes`，协议冒号之后是否需要反斜线，举例：true or false
- `host`，URL 的主机部分，包含端口信息，举例：'host.com:8080'
- `auth`，URL 的授权信息部分，举例：'user:pass'
- `hostname`，主机的主机名部分，举例：'host.com'
- `port`，主机的端口号，举例：'8080'
- `pathname`，URL 的路径部分，位于主机之后、查询字符串之前，如果该值存在则第一个字符为反斜线，不对字符串解码，举例：'/p/a/t/h'
- `search`，URL 的查询字符串部分, 以问号开头，举例：'?query=string'
- `path`，URL 的路径名和查询字符串，不对字符串解码，举例：'/p/a/t/h?query=string'
- `query`，查询字符串中的参数部分，或者是使用 querystring 解析后的对象，举例：'query=string' or {'query':'string'}
- `hash`，以 # 开头的 URL 片段，举例：'#hash'

转义字符

默认情况下，如果有空格和以下字符出现在 URL 中，则会被自动转义：

```
< > " ` \r \n \t { } | \ ^ '
```

url.format(urlObj)

接收一个解析后的 URL 对象，返回一个格式化的 URL 字符串。

下面是格式化流程：

- `href` 会被忽略
- `path` 会被忽略
- `protocol`，允许缺少冒号，只要存在 host 或者 hostname 字段，`http` / `https` / `ftp` / `gopher` / `file` 协议就会被添加 `://` 后缀，其他协议会被添加 `:` 后缀，比如 `mailto` / `xmpp` / `aim` / `sftp` / `foo` 协议等
- `slashes`，如果缺少 host 或者 hostname 字段、协议不属于默认添加 `://` 后缀的协议且需要添加 `://` 后缀，则需要将该值赋值为 true 如果协议需要需要 `://`
- `auth`，存在即有效
- `hostname`，host 缺失时有效
- `port`，host 缺失时有效
- `host`，用来替代 hostname 和 port
- `pathname`，无论是否以反斜线开头，都做相同的梳理
- `query`，search 缺失时有效
- `search`，用于替代 query，无论是否以问号开头，都会做通常的处理
- `hash`，无论是否以 # 开头，都会做通常的处理

url.parse(urlStr[, parseQueryString][, slashesDenoteHost])

该方法用于接收一个 URL 字符串，返回一个 URL 对象。

如果第二个参数的值为 true，则使用 `queryString` 模块解析查询字符串，`query` 属性指向一个对象，`search` 属性指向一个字符串。如果值为 false，`query` 属性不会被解析或编码。默认值为 false。

如果第三个参数的值为 true，则将 `//foo/bar` 解析为 `{ host: 'foo', pathname: '/bar' }`，而不是 `{ pathname: '//foo/bar' }`。默认值为 false。

url.resolve(from, to)

接收一个 base URL 和一个 href URL，返回一个跳转后的链接地址：

```
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'
```

工具集

接口稳定性: 2 - 稳定

下面这些函数位于 `util` 模块，需要使用 `require()` 方法加载和访问。

`util` 模块设计之初的目的是为 Node.js 的内部 API 服务。该模块中的大部分辅助函数对开发者开发程序也很有帮助。如果你发现这些函数并不能满足你的需求，那么建议你开发自己的辅助工具集。我们无意创建一个功能齐全的 `util` 模块，只会在该模块中编写对 Node.js 内部 API 开发有用的函数。

util.debuglog(setion)

- `section`，字符串，待调试的程序代码
- 返回值类型：Function，记录函数

该方法常用于创建一个限制性函数，该函数根据是否有 `NODE_DEBUG` 环境变量向 stderr 输出数据。如果 `section` 的名称出现在环境变量中，那么就会返回一个类似 `console.error()` 的函数，否则返回一个空操作。

```
var debuglog = util.debuglog('foo');

var bar = 123;
debuglog('hello from foo [%d]', bar);
```

如果程序在 `NODE_DEBUG=foo` 的环境中运行，那么它就会输出类似下面这样的信息：

```
FOO 3245: hello from foo [123]
```

这里的 `3245` 是进程 ID。如果程序运行的环境中没有这样的变量，则不会输出任何信息。此外，你可以使用逗号分隔多个 `NODE_DEBUG` 环境变量，比如 `NODE_DEBU=fs,net,tls`。

util.deprecate(function, string)

该方法用来标明某个函数不能再被使用：

```
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');
```

最终返回一个修改后的函数，调用该函数时默认会发出一次警示。

如果开启了 `--no-deprecation` 选项，则该函数为空操作。通过布尔类型的 `process.noDeprecation` ，可以在程序运行中配置该参数，需要注意的是，必须在模块加载前执行该配置。

如果开启了 `--trace-deprecation` 选项，则该函数第一次被调用时，会在控制台输出提示和堆栈跟踪信息。通过布尔类型的 `process.traceDeprecation` ，可以在程序运行中配置该参数。

如果开启了 `--throw-deprecation` 选项，则该函数被调用时抛出错误。通过布尔类型的 `process.throwDeprecation` ，可以在程序运行中配置该参数。 `process.throwDeprecation` 优先级高于 `process.traceDeprecation` 。

util.format(format[, ...])

该方法类似 `printf` 函数，返回一个格式化后的字符串。

第一个参数是一个字符串，包含一个或多个占位符。支持的占位符包括以下几种：

- `%s` ，字符串
- `%d` ，数值，整数或浮点数
- `%j` ，JSON，如果参数中包含循环引用，则使用 `[Circular]` 表示
- `%%` ，百分号

如果占位符缺少对应的参数，则直接输出该占位符：

```
util.format('%s:%s', 'foo');
// 'foo:%s'
```

如果参数多余占位符，则该参数被转换为字符串（如果参数是对象或者 symbol，则使用 `util.inspect()` 进行转换），然后使用空格连接成一个字符串：

```
util.format('%s:%s', 'foo', 'bar', 'baz');
// 'foo:bar baz'
```

如果第一个参数不包含占位符，则 `util.format()` 返回一个字符串，该字符串由所有参数连接而成，参数之间以空格分隔。参数转换时，使用 `util.inspect()` 方法进行转换：

```
util.format(1, 2, 3);
// '1 2 3'
```

util.inherits(constructor, superConstructor)

该方法使 `constructor` 构造函数继承 `superConstructor` 构造函数的原型方法。这里的 `superConstructor` 也可以通过 `constructor.super_` 属性得到。

```
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.inspect(object[, options])

该方法返回 `object` 参数的字符串形式，常用于代码调试。

可选的参数对象 `options` 可以用来改变字符串格式化后的样式：

- `showHidden`，如果为 true，则显示对象的不可枚举和 symbol 属性，默认值为 false
- `depth`，指定 `inspect()` 方法格式化 `object` 时的深度。这对于格式化复杂对象很有用。默认值为 2，如果不限制深度，可以设置为 `null
- `colors`，如果值为 true，则使用 ANSI 颜色码对输出信息加以美化。默认值为 false

- `customInspect`，如果值为 false，则 `object` 对象上自定义的 `inspect(depth, opts)` 函数不会被执行，默认值为 true。

下面代码演示了如何审查 `util` 对象的所有属性：

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

console.log(util.inspect(util, { showHidden: true, depth: null }
));
```

该方法被调用时，`object` 对象可以使用自定义的 `inspect(depth, opts)` 方法，该方法接收两个参数，一个是当前的递归深度，另一个是传递给 `util.inspect()` 的可选对象。

自定义 util.inspect 颜色

`util.inspect()` 的输出颜色可由 `util.inspect.styles` 和 `util.inspect.colors` 进行配置。

`util.inspect.styles` 与 `util.inpect.colors` 中的颜色一一对应。常见数据类型的默认高亮样式：

- Number，黄色
- Boolean，黄色
- String，绿色
- Date，洋红
- Regexp，红色
- Null，黑体，
- Undefined，灰色，
- Special，青色，目前 special 只包括 Function 类型
- Name，默认无样式

系统预定义的颜色包括：`white` / `grey` / `black` / `blue` / `cyan` / `green` / `magenta` / `red` 和 `yellow`。此外，还有 `bold` / `italic` / `underline` 和 `inverse` 样式。

自定义对象的 inspect() 函数

对象也可以自定义 `inspect(depth)` 函数，当 `util.inspect()` 方法审查该对象时就会调用这一自定义的方法：

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

var obj = { name: 'nate' };
obj.inspect = function(depth) {
  return `{${this.name}}`;
};

util.inspect(obj);
// "{nate}"
```

也可以从自定义的方法返回另一个对象，`util.inspect()` 方法会返回根据该对象返回格式化的字符串，非常类似 `JSON.stringify()` 的工作方式：

```
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.log(string)

在 `stdout` 输出的同时带上时间戳。

```
require('util').log('Timestamped message.');
```

以下函数已被抛弃

接口稳定性: 0 - 已过时

- util.debug(string)
- util.error([...])
- util.isArray(object)
- util.isBoolean(object)
- util.isBuffer(object)
- util.isDate(object)
- util.isError(object)
- util.isFunction(object)
- util.isNull(object)
- util.isNullOrUndefined(object)
- util.isNumber(object)
- util.isObject(object)
- util.isPrimitive(object)
- util.isRegExp(object)
- util.isString(object)
- util.isSymbol(object)
- util.isUndefined(object)
- util.print([...])
- util.pump(readableStream, writableStream[, callback])
- util.puts([...])

V8

> 接口稳定性: 2 - 稳定

该模块用于开放 Node.js 内建的 V8 引擎的事件和接口。这些接口由 V8 底层决定，所以无法保证绝对的稳定性。

getHeapStatistics()

返回一个带有下列属性的对象：

```
{
  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()

返回和 V8 堆空间有关的统计数据，比如构成 V8 堆空间的段信息。由于堆空间的统计信息是由 V8 的 `GetHeapSpaceStatistics` 函数提供的，所以无法保证堆空间或者可用堆空间的顺序。

```
[
  {
    "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
  }
]
```

setFlagsFromString(string)

该方法用于添加额外的 V8 命令行标志。使用时需要注意，在 VM 启动后修改配置可能会发生不可预测的行为、崩溃和数据丢失，或者什么反应都没有。

通过 `node --v8-options` 命令可以查询当前 Node.js 环境中有哪些可用的 V8 options。此外，还可以参考非官方维护的一个 V8 options 列表，链接地址在这里 https://github.com/thlorenz/v8-flags/blob/master/flags-0.11.md。

```
// Print GC events to stdout for one minute.
const v8 = require('v8');
v8.setFlagsFromString('--trace_gc');
setTimeout(function() { v8.setFlagsFromString('--notrace_gc'); }
, 60e3);
```

VM

> 接口稳定性: 2 - 稳定

通过 `require('vm')` 可以加载该模块。利用该模块可以立即编译执行或先编译保存后执行 JavaScript 代码。

Class: Script

该类封装了一些预编译脚本，并在沙盒中运行这些脚本。

new vm.Script(code, options)

该方法是一个构造函数，创建一个 `Script` 实例并编译 `code`，但不运行编译后的代码，最终返回的 `vm.Script` 对象存储了编译后的代码。使用下面的函数可以反复运行 `vm.Script` 中的代码。`vm.Script` 中的代码并没有绑定到任何的全局对象上，它们会在运行前绑定，并运行后解绑。

参数 `options` 包含以下属性：

- `filename`，该属性允许开发者更改堆栈跟踪信息中的文件名
- `lineOffset`，该属性用于在堆栈跟踪信息中为行号设置偏移位置
- `columnOffset`，该属性用于在堆栈跟踪信息中为列号设置偏移位置
- `displayErrors`，该属性决定在抛出异常之前，是否向 stderr 输出错误并高亮问题代码。只有编译期的语法问题会被抛出，执行期的问题则有 Script 的函数处理
- `timeout`，用于指定 `code` 执行的超时时间，超时则中断执行，并抛出错误
- `cacheData`，该属性是一个包含 V8 代码缓存数据的 Buffer。`cachedDataRejected` 是布尔类型，其值取决于是否接受 V8 的数据。
- `produceCachedDate`，如果值为 true 且没有 `cachedData` 参数，V8 会尝试为 `code` 缓存代码，如果缓存成功，就会生成一个 Buffer 实例存储代码的缓存数据，该 Buffer 实例也会成为返回的 `vm.Script` 的一个实

例。`cachedDataProduced` 是一个布尔类型，其值取决于系统是否成功生成了代码缓存数据。

script.renInContext(contextifiedSandbox[, options])

该方法与 `vm.runInContext()` 类似，但后者是挂载在预编译 `Script` 对象下的一个方法。`scirpt.runInContext()` 方法在 `contextifiedSandbox` 中运行 `Script` 编译后的代码并返回相应的结果。运行代码时并不会访问本地作用域。

`script.runInContext` 接收的 `options` 和 `script.runInThisContext()` 一致。

下面代码演示了使用 VM 模块编译后的代码修改全局变量并反复执行的效果：

```
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' }
```

注意运行不可信代码时需要万分小心。`script.runInContext()` 虽然很有用，但是运行不可信代码时最好使用一个独立的进程。

script.runInNewContext([sandbox][, options])

该方法与 `vm.runInNewContext()` 类似，但后者是挂载在预编译 `Script` 对象下的一个方法。`scirpt.runInNewContext()` 如果没有收到 `sandbox` 就会自建一个沙盒，然后在沙盒中以全局对象的形式运行 `Script` 编译后的代码。运行代码时并不会访问本地作用域。

`script.runInNewContext` 接收的 `options` 和 `script.runInThisContext()` 一致。

下面代码演示了使用 VM 模块编译后的代码修改全局变量并反复执行的效果：

```
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: 'se
t' }]
```

注意运行不可信代码时需要万分小心。`script.runInNewContext()` 虽然很有用，但是运行不可信代码时最好使用一个独立的进程。

script.runInThisContext([options])

该方法与 `vm.runInThisContext()` 类似，但后者是挂载在预编译 `Script` 对象下的一个方法。`scirpt.runInThisContext()` 方法运行 `Script` 编译后的代码并返回相应的结果。运行代码时没有访问本地作用域的权限，但有访问 `global` 对象的权限。

下面代码演示了使用 `script.runInThisContext()` 编译代码并多次运行的效果：

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

global.globalVar = 0;

const script = new vm.Script('globalVar += 1', { filename: 'myfi
le.vm' });

for (var i = 0; i < 1000; ++i) {
  script.runInThisContext();
}

console.log(globalVar);
// 1000
```

`options` 参数接收一下参数：

- `filename`，该属性允许开发者更改堆栈跟踪信息中的文件名
- `lineOffset`，该属性用于在堆栈跟踪信息中为行号设置偏移位置
- `columnOffset`，该属性用于在堆栈跟踪信息中为列号设置偏移位置
- `displayErrors`，该属性决定在抛出异常之前，是否向 stderr 输出错误并高亮问题代码。只有编译期的语法问题会被抛出，执行期的问题则有 Script 的函数处理
- `timeout`，用于指定 `code` 执行的超时时间，超时则中断执行，并抛出错误

vm.createContext([sandbox])

如果指定了 `sandbox` 对象，那么该使用该对象调用 `vm.runInContext()` 或 `script.runInContext()` 方法。在脚本内部，`sandbox` 会被视为全局对象，包含所有的已有属性以及标准全局对象所拥有的内建对象和方法。在 VM 模块所执行的脚本外部，`sandbox` 不会被修改。

如果指定沙盒对象，那么该方法返回一个新的空沙盒对象。

该方法常用于创建一个可以运行多个脚本的沙盒，比如为一个模拟浏览器创建煞和对象，该沙盒对象包含了全局对象 window 的属性和方法，最后将所有的 `script` 标签置于沙盒中运行。

vm.isContext(sandbox)

该方法返回一个布尔值，用于判定沙盒对象是否经由 `vm.createContext()` 初始化过。

vm.runInContext(code, contextifiedSandbox[, options])

`vm.runInContext()` 方法先编译 `code`，然后使用指定的 `contextifiedSandbox` 运行代码并返回结果。运行中的代码没有访问本地作用域的权限。`contextifiedSandbox` 对象必须使用 `vm.createContext()` 创建或初始化，它会被视为 `code` 的全局对象。

`vm.runInContext` 接收的 `options` 和 `vm.runInThisContext()` 一致。

下面代码演示了使用同一个上下文环境编译和执行不同脚本的效果：

```
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 }
```

注意运行不可信代码时需要万分小心。`vm.runInContext()` 虽然很有用，但是运行不可信代码时最好使用一个独立的进程。

vm.runInDebugContext(code)

`vm.runInDebugContext()` 在 V8 的调试上下文环境中编译和执行 `code`。该方法最主要的场景是获取 V8 的调试对象：

```
const Debug = vm.runInDebugContext('Debug');
Debug.scripts().forEach(function(script) { console.log(script.na
me); });
```

注意，这里的调试上下文和对象都是和 V8 的调试器绑定在一起的，所以当 V8 引擎有所改动时可能并不会有警告。

通过 `--expose_debug_as= switch` 也可以暴漏调试对象。

vm.runInNewContext(code[, sandbox][, options])

`vm.runInNewContext()` 如果没有收到 `sandbox` 就会自建一个沙盒，然后在沙盒中编译 `code`，最后使用沙盒作为全局对象执行编译后的代码并返回结果。

`vm.runInNewContext` 接收的 `options` 和 `vm.runInThisContext()` 一致。

下面代码演示了使用 VM 模块编译后的代码修改全局变量并反复执行的效果：

```
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' }
```

注意运行不可信代码时需要万分小心。`vm.runInNewContext()` 虽然很有用，但是运行不可信代码时最好使用一个独立的进程。

vm.runInThisContext(code[, options])

`vm.runInThisContext()` 方法用于编译执行 `code` 并返回结果。运行代码时没有访问本地作用域的权限，但有访问 `global` 对象的权限。

下面代码演示了使用 `vm.runInThisContext()` 和 `eval()` 运行相同代码的差别：

```
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()` 并没有访问本地作用域的权限，所以 `localVar` 不会被修改。`eval` 具有访问本地作用域的权限，所以可以修改 `loalVar` 。

虽然 `vm.runInThisContext()` 的这种用法很类似直接调用 `eval()` ，比如 `(0,eval)('code')` ，但是 `vm.runInThisContext()` 的优势在于还可以接收以下参数：

- `filename` ，该属性允许开发者更改堆栈跟踪信息中的文件名
- `lineOffset` ，该属性用于在堆栈跟踪信息中为行号设置偏移位置
- `columnOffset` ，该属性用于在堆栈跟踪信息中为列号设置偏移位置
- `displayErrors` ，该属性决定在抛出异常之前，是否向 stderr 输出错误并高亮问题代码。只有编译期的语法问题会被抛出，执行期的问题则有 Script 的函数处理
- `timeout` ，用于指定 `code` 执行的超时时间，超时则中断执行，并抛出错误

Zlib

接口稳定性: 2 - 稳定

通过 `require('zilb')` 的方式可以访问 `zlib` 模块，该模块提供了对 Gzip/Gunzip，Deflate/Inflate 和 DeflateRaw/InflateRaw 类的绑定，这些类接收相同的参数，都属于可读写的 Stream 实例。

实例

下面代码演示了通过 `fs.ReadStream` 管道传输到 zlib stream 在传输到 `fs.WriteStream` 所实现的文件解压缩流程：

```
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);
```

下面代码演示了使用这些方法一步实现数据解压缩的过程：

```
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
  }
});
```

在 HTTP 客户端或服务端使用该模块时，请使用 `accept-encoding` 发送请求，使用 `context-encoding` 头信息发送响应。

注意：这些示例的目的只是用于演示该模块的基础概念。Zlib 的编码过程是非常耗时的，所以开发者应该缓存编码结果。更多有关 zlib 模块中对 speed/memory/compression 的权衡和考虑，请参考 Memory Usage Tuning。

```
// 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,def
late' } });
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 buffe
r.
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);
```

内存调优

ZLIB 模块的配置文件存放于 `zili/zconf.h`，通过修改该文件可以调整 Node.js 系统的性能。

deflate 压缩算法需要的内存计算方式：

```
(1 << (windowBits+2)) +  (1 << (memLevel+9))
```

默认情况下，总共需要 128K（windowBits = 15）+ 128K（memLevel = 8），再加上一些小对象占用的字节。

举例来说，如果开发者想将默认的内存大小从 256K 减少为 128K，那么就可以传入参数：

```
{ windowBits: 14, memLevel: 7 }
```

当然，这么做会降低压缩等级。

inflate 解压算法需要的内存计算方式：

```
1 << windowBits
```

默认情况下，总共需要 32K（windowBits = 15），再加上一些小对象占用的字节。

上面的计算都排除了内部输出缓存 `chunkSize`，其默认值为 16K。

zlib 的压缩速度主要受 `level` 的影响，压缩级别越高，压缩效果就会越好，但压缩时间就会越长，反之异然。

通常来说，可用内存越多，意味着对 zlib 模块在内存的交换越少，通过一次 `write` 操作所能处理的数据也就越多，所以内存的可用量也是一个影响压缩速度的重要因素。

常量

所有定义在 `zlib.h` 中的变量也可以通过 `require('zlib')` 获取。对于常规操作，开发者通常无需配置这些变量。本节内容主要选取自 zlib 的文档，更多相关信息请参考 http://zlib.net/manual.html#Constants。

允许刷新的值：

- 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

以下是压缩、解压函数的返回值，负值表示错误，正值表示特殊但正常的事件：

- 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` 的可选值：

- zlib.Z_BINARY
- zlib.Z_TEXT
- zlib.Z_ASCII
- zlib.Z_UNKNOWN

deflate 压缩算法：

- zlib.Z_DEFLATED

初始化 zalloc/zfree/opaque：

- zlib.Z_NULL

Class Options

每一个类都接收一个可选对象作为配置参数，且所有的配置参数都是可选的。

注意，有些可选属性只对压缩过程有效，对解压过程无效：

- flush (默认值 zlib.Z_NO_FLUSH)
- chunkSize (默认值 16*1024)
- windowBits
- level (只对压缩有效)
- memLevel (只对压缩有效)
- strategy (只对压缩有效)
- dictionary (只对 deflate/inflate 算法有效, 默认为空目录)

更多有关 `deflateInit2` 和 `inflateInit2` 的信息请参考 http://zlib.net/manual.html#Advanced。

Class: zlib.Deflate

使用 deflate 算法压缩数据。

Class: zlib.DeflateRaw

使用 deflate 算法压缩数据，且不添加 zlib 头信息。

Class: zlib.Gunzip

使用 gzip 算法解压数据。

Class: zlib.Gzip

使用 gzip 算法压缩数据。

Class: zlib.Inflate

使用 inflate 算法解压数据。

Class: zlib.InflateRaw

使用 deflate 算法解压数据。

Class: zlib.Unzip

通过自动检测头信息解压 gzip 或 deflate 数据。

Class: zlib.Zlib

`zilb` 模块并不会输出该类，之所以在这里列出，是因为它是前天解压缩类的基类。

zlib.flush([kind], callback)

`kind` 的默认值为 `zlib.Z_FULL_FLUSH`。

该方法用于输入缓冲数据，请谨慎使用该方法，因为过早的刷新会严重影响压缩算法的效率。

zlib.params(level, strategy, callback)

该方法用于动态更新压缩级别和压缩策略，且只对 deflate 算法有效。

zlib.reset()

该方法用于重置解压缩算法，且只对 inflate/defalte 算法有效。

zlib.createDeflate(options)

该方法根据指定的 `options` 返回一个新的 Defalte 对象。

zlib.createDeflateRaw(options)

该方法根据指定的 `options` 返回一个新的 DeflateRaw 对象。

zlib.createGunzip(options)

该方法根据指定的 `options` 返回一个新的 Gunzip 对象。

zlib.createGzip(options)

该方法根据指定的 `options` 返回一个新的 Gzip 对象。

zlib.createInflate(options)

该方法根据指定的 `options` 返回一个新的 Inflate 对象。

zlib.createInflateRaw(options)

该方法根据指定的 `options` 返回一个新的 InfalteRaw 对象。

zlib.createUnzip(options)

该方法根据指定的 `options` 返回一个新的 Unzip 对象。

简便方法

以下所有方法的第一个参数都是 Buffer 实例或字符串，第二个参数是一个可选的 `options`，第三个参数是一个回调函数，其形式为 `callback(error, result)`。

每一个方法都有一个 `*Sync` 版本，接收相同的参数，但不接收回调函数。

zlib.deflate(buf[, options], callback)

zlib.deflateSync(buf[, options])

该方法使用 Defalte 压缩 Buffer 实例或字符串。

zlib.deflateRaw(buf[, options], callback)

zlib.deflateRawSync(buf[, options])

该方法使用 DefalteRaw 压缩 Buffer 实例或字符串。

zlib.gunzip(buf[, options], callback)

zlib.gunzipSync(buf[, options])

该方法使用 Gunzip 解压 Buffer 实例或字符串。

zlib.gzip(buf[, options], callback)

zlib.gzipSync(buf[, options])

该方法使用 Gzip 解压 Buffer 实例或字符串。

zlib.inflate(buf[, options], callback)

zlib.inflateSync(buf[, options])

该方法使用 Infalte 解压 Buffer 实例或字符串。

zlib.inflateRaw(buf[, options], callback)

zlib.inflateRawSync(buf[, options])

该方法使用 InflateRaw 解压 Buffer 实例或字符串。

zlib.unzip(buf[, options], callback)

zlib.unzipSync(buf[, options])

该方法使用 Unzip 解压 Buffer 实例或字符串。