node.js 官方文档中文版

★ 基于 Google V8 引擎的服务器端 JavaScript 解释器

极客学院出版

前言

> 本文档翻译自 Node.js (https://nodejs.org/api/) 官方文档，适用于 V0.12.2。

本文档从引用参考和概念两个方面全面的解释了 Node.js API。每个章节描述了一个模块或高级概念。

一般情况下，属性、方法参数，及事件都会列在主标题下的列表中。

每个 `.html` 文档都有对应的 `.json` ，它们包含相同的结构化内容。这些东西目前还是实验性的，主要为各种集成开发环境（ IDE ）和开发工具提供便利。

每个 `.html` 和 `.json` 文件都和 `doc/api/` 目录下的 `.markdown` 文件相对应。这些文档使用 `tools/doc/generate.js` 程序生成。 HTML 模板位于 `doc/template.html` 。

稳定性标志

在文档中，你会看到每个章节的稳定性标志。Node.js API 还在改进中，成熟的部分会比其他章节值得信赖。经过大量验证和依赖的 API 是一般是不会变的。其他新增的，试验性的，或被证明具有危险性的部分正重新设计。

稳定性标志包括以下内容:

稳定性（ Stability ）: 0 – 抛弃
这部分内容有问题，并已计划改变。不要使用这些内容，可能会引起警告。不要想什么向后兼容性了。

稳定性（ Stability ）: 1 – 试验
这部分内容最近刚引进，将来的版本可能会改变也可能会被移除。你可以试试并提供反馈。如果你用到的部分对你来说非常重要，可以

稳定性（ Stability ）: 2 – 不稳定
这部分 API 正在调整中，还没在实际工作测试中达到满意的程度。如果合理的话会保证向后兼容性。

稳定性（ Stability ）: 3 – 稳定
这部分 API 验证过基本能令人满意，但是清理底层代码时可能会引起小的改变。保证向后的兼容性。

稳定性（ Stability ）: 4 – API 冻结
这部分的 API 已经在产品中广泛试验，不太可能被改变。

稳定性（ Stability ）: 5 – 锁定
除非发现了严重 bug，否则这部分代码永远不会改变。请不要对这部分内容提出更改建议，否则会被拒绝。

JSON 输出

稳定性 (Stability) : 1 – 试验

每个通过 markdown 生成的 HTML 文件都有相应的 JSON 文件。

这个特性从 v0.6.12 开始,是试验性功能。

更新日期	更新内容
2015-04-21	第一版发布，翻译自 Node.js V0.12.2 官方文档

目录

1

概述

第一个 服务器 () 的例子就从 “Hello World” 开始：

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

把代码拷贝到 example.js 文件里，用 node 程序执行

```
> node example.js
Server running at http://127.0.0.1:8124/
```

文档中所有的例子都可以这么执行。

2

断言测试

稳定性: 5 – 锁定

这个模块可用于应用的单元测试，通过 `require('assert')` 可以使用这个模块。

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

使用参数 operator 测试参数 `actual` (实际值) 和 `expected` （期望值）是否相等。

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

测试参数 `value` 是否为 `true` ,此函数和 `assert.equal(true, !!value, message);` 等价。

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

判断实际值 `actual` 和期望值 `expected` 是否相等。

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

判断实际值 `actual` 和期望值 `expected` 是否不等。

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

执行深度比较，判断实际值 `actual` 和期望值 `expected` 是否相等。

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

深度比较两个参数是否不相等。

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

深度比较两个参数是否相等。

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

此函数使用操作符 ‘!==’ 严格比较是否两参数不相等。

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

声明一个 `block` 用来抛出错误（ `error` ）， `error` 可以是构造函数，正则表达式或其他验证器。

使用构造函数验证实例：

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  Error
);
```

使用正则表达式验证错误信息：

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /value/
);
```

用户自定义的错误验证器：

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

assert.doesNotThrow(block[, message])

声明 `block` 不抛出错误，详细信息参见 `assert.throws` 。

assert.ifError(value)

判断参数 value 是否为 false ， 如果是 true 抛出异常。通常用来测试回调中第一个参数 error。

3

Buffer

稳定性: 3 – 稳定

纯 Javascript 语言对 Unicode 友好，但是难以处理二进制数据。在处理 TCP 流和文件系统时经常需要操作字节流。Node 提供了一些机制，用于操作、创建、以及消耗字节流。

在 Buffer 类实例化中存储了原始数据。 Buffer 类似于一个整数数组，在 V8 堆（the V8 heap）原始存储空间给它分配了内存。一旦创建了 Buffer 实例，则无法改变大小。

`Buffer` 类是全局对象，所以访问它不必使用 `require('buffer')` 。

Buffers 和 Javascript 字符串对象之间转换时需要一个明确的编码方法。下面是字符串的不同编码。

- `'ascii'` – 7位的 ASCII 数据。这种编码方式非常快，它会移除最高位内容。
- `'utf8'` – 多字节编码 Unicode 字符。大部分网页和文档使用这类编码方式。
- `'utf16le'` – 2个或4个字节, Little Endian (LE) 编码 Unicode 字符。编码范围 (U+10000 到 U+10FFFF) 。
- `'ucs2'` – `'utf16le'` 的子集。
- `'base64'` – Base64 字符编码。
- `'binary'` – 仅使用每个字符的头8位将原始的二进制信息进行编码。在需使用 Buffer 的情况下，应该尽量避免使用这个已经过时的编码方式。这个编码方式将会在未来某个版本中弃用。
- `'hex'` – 每个字节都采用 2 进制编码。

在 `Buffer` 中创建一个数组，需要注意以下规则：

1. Buffer 是内存拷贝，而不是内存共享。
2. Buffer 占用内存被解释为一个数组，而不是字节数组。比如， `new Uint32Array(new Buffer([1,2,3,4]))` 创建了4个 `Uint32Array` ，它的成员为 `[1,2,3,4]` ,而不是 `[0x1020304]` 或 `[0x4030201]` 。

注意：Node.js v0.8 只是简单的引用了 `array.buffer` 里的 buffer ,而不是克隆(cloning)。

介绍一个高效的方法， `ArrayBuffer#slice()` 拷贝了一份切片，而 `Buffer#slice()` 新建了一份。

类: Buffer

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

new Buffer(size)

- `size` Number 类型

分配一个新的 `size` 大小单位为8位字节的 buffer。 注意, `size` 必须小于 kMaxLength (页 0)，否则，将会抛出异常 `RangeError` 。

new Buffer(array)

- `array` Array

使用一个8位字节 `array` 数组分配一个新的 buffer。

new Buffer(buffer)

- `buffer` {Buffer}

拷贝参数 `buffer` 的数据到 `Buffer` 实例。

new Buffer(str[, encoding])

- `str` String 类型 – 需要编码的字符串。
- `encoding` String 类型 – 编码方式, 可选。

分配一个新的 buffer ， 其中包含着传入的 `str` 字符串。 `encoding` 编码方式默认为 `'utf8'` 。

类方法: Buffer.isEncoding(encoding)

- `encoding` {String} 用来测试给定的编码字符串

如果参数编码 `encoding` 是有效的，返回 true，否则返回 false。

类方法: Buffer.isBuffer(obj)

- `obj` 对象
- 返回： Boolean

`obj` 如果是 `Buffer` 返回 true, 否则返回 false。

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

- `string` String 类型
- `encoding` String 类型, 可选, 默认: 'utf8'
- 返回： Number 类型

将会返回这个字符串真实字节长度。 encoding 编码默认是： `utf8` 。这和 `String.prototype.length` 不一样，因为那个方法返回这个字符串中字符的数量。

例如:

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

console.log(str + ": " + str.length + " characters, " +
  Buffer.byteLength(str, 'utf8') + " bytes");

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

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

- `list` {Array} 用来连接的数组
- `totalLength` {Number 类型} 数组里所有对象的总buffer大小

返回一个buffer对象，它将参数 buffer 数组中所有 buffer 对象拼接在一起。

如果传入的数组没有内容，或者 totalLength 是 0，那将返回一个长度为 0 的buffer。

如果数组长度为 1，返回数组第一个成员。

如果数组长度大于 0，将会创建一个新的 Buffer 实例。

如果没有提供 totalLength 参数，会根据 buffer 数组计算，这样会增加一个额外的循环计算，所以提供一个准确的 totalLength 参数速度更快。

类方法: Buffer.compare(buf1, buf2)

- `buf1` {Buffer}

- `buf2` {Buffer}

和 `buf1.compare(buf2)` (页 0)一样。 用来对数组排序:

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

buf.length

- Number 类型

返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容的大小。`length` 是 buffer 对象所分配的内存数，它不会随着这个 buffer 对象内容的改变而改变。

```
buf = new Buffer(1234);

console.log(buf.length);
buf.write("some string", 0, "ascii");
console.log(buf.length);

// 1234
// 1234
```

`length` 不能改变，如果改变 `length` 将会导致不可以预期的结果。如果想要改变 buffer 的长度，需要使用 `buf.slice` 来创建新的 buffer。

```
buf = new Buffer(10);
buf.write("abcdefghj", 0, "ascii");
console.log(buf.length); // 10
buf = buf.slice(0,5);
console.log(buf.length); // 5
```

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

- `string` String 类型 – 写到 buffer 里
- `offset` Number 类型, 可选参数， 默认值: 0
- `length` Number 类型, 可选参数， 默认值: `buffer.length – offset`
- `encoding` String 类型， 可选参数， 默认值: 'utf8'

根据参数 offset 偏移量和指定的 encoding 编码方式，将参数 string 数据写入buffer。 offset 偏移量默认值是 0, encoding 编码方式默认是 `utf8` 。 length 长度是将要写入的字符串的 bytes 大小。 返回 number 类型，表

示写入了多少 8 位字节流。如果 buffer 没有足够的空间来放整个 string，它将只会只写入部分字符串。 length 默认是 buffer.length – offset。 这个方法不会出现写入部分字符。

```
buf = new Buffer(256);
len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(len + " bytes: " + buf.toString('utf8', 0, len));
```

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

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

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

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

- `value` {Number 类型} 准备写到 buffer 字节数
- `offset` {Number 类型} `0 <= offset <= buf.length`
- `byteLength` {Number 类型} `0 < byteLength <= 6`
- `noAssert` {Boolean} 默认值: false
- 返回： {Number 类型}

将 `value` 写入到 buffer 里， 它由 `offset` 和 `byteLength` 决定，支持 48 位计算，例如:

```
var b = new Buffer(6);
b.writeUIntBE(0x1234567890ab, 0, 6);
// <Buffer 12 34 56 78 90 ab>
```

`noAssert` 值为 `true` 时，不再验证 `value` 和 `offset` 的有效性。 默认是 `false` 。

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

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

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

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

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

- `byteLength` {Number 类型} `0 < byteLength <= 6`
- `noAssert` {Boolean} 默认值: false
- 返回: {Number 类型}

支持 48 位以下的数字读取。 例如:

```
var b = new Buffer(6);
b.writeUint16LE(0x90ab, 0);
b.writeUInt32LE(0x12345678, 2);
b.readUIntLE(0, 6).toString(16);  // 指定为 6 bytes (48 bits)
// 输出: '1234567890ab'
```

`noAssert` 值为 true 时， `offset` 不再验证是否超过 buffer 的长度，默认为 `false` 。

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

- `encoding` String 类型， 可选参数， 默认值: 'utf8'
- `start` Number 类型， 可选参数， 默认值: 0
- `end` Number 类型， 可选参数， 默认值: `buffer.length`

根据 encoding 参数（默认是 'utf8'）返回一个解码过的 string 类型。还会根据传入的参数 start (默认是 0) 和 end (默认是 buffer.length)作为取值范围。

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

查看上面 `buffer.write()` 例子。

buf.toJSON()

返回一个 JSON 表示的 Buffer 实例。 `JSON.stringify` 将会默认调用字符串序列化这个 Buffer 实例。

例如:

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

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

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

console.log(copy);
// <Buffer 74 65 73 74>
```

buf[index]

获取或设置指定 index 位置的 8 位字节。这个值是指单个字节，所以必须在合法的范围取值，16 进制的 0x00 到 0xFF，或者 0 到 255。

例如: 拷贝一个 ASCII 编码的 string 字符串到一个 buffer, 一次一个 byte 进行拷贝:

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

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

console.log(buf);

// node.js
```

buf.equals(otherBuffer)

- `otherBuffer` {Buffer}

如果 `this` 和 `otherBuffer` 拥有相同的内容，返回 true。

buf.compare(otherBuffer)

- `otherBuffer` {Buffer}

返回一个数字，表示 `this` 在 `otherBuffer` 之前，之后或相同。

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

- `targetBuffer` Buffer 对象 – Buffer to copy into
- `targetStart` Number 类型， 可选参数， 默认值: 0
- `sourceStart` Number 类型， 可选参数， 默认值: 0
- `sourceEnd` Number 类型， 可选参数， 默认值: `buffer.length`

buffer 拷贝，源和目标可以相同。 `targetStart` 目标开始偏移和 `sourceStart` 源开始偏移默认都是 0。 `sourceEnd` 源结束位置偏移默认是源的长度 `buffer.length` 。

例如:创建 2 个Buffer，然后把 buf1 的 16 到 19 位内容拷贝到 buf2 第 8 位之后。

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

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
  buf2[i] = 33; // ASCII !
}

buf1.copy(buf2, 8, 16, 20);
console.log(buf2.toString('ascii', 0, 25));

// !!!!!!!!qrst!!!!!!!!!!!!!
```

例如: 在同一个buffer中，从一个区域拷贝到另一个区域

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

- `start` Number 类型， 可选参数， 默认值: 0
- `end` Number 类型， 可选参数， 默认值: `buffer.length`

返回一个新的buffer，这个buffer将会和老的buffer引用相同的内存地址，根据 `start` (默认是 `0`) 和 `end` (默认是 `buffer.length`) 偏移和裁剪了索引。 负的索引是从 buffer 尾部开始计算的。

修改这个新的 buffer 实例 slice 切片，也会改变原来的 buffer!

例如: 创建一个 ASCII 字母的 Buffer，进行 slice 切片，然后修改源 Buffer 上的一个 byte。

```
var buf1 = new Buffer(26);

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

var buf2 = buf1.slice(0, 3);
console.log(buf2.toString('ascii', 0, buf2.length));
buf1[0] = 33;
console.log(buf2.toString('ascii', 0, buf2.length));

// abc
// !bc
```

buf.readUInt8(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，读取一个有符号 8 位整数 整形。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 如果这样 `offset` 可能会超出buffer 的末尾。默认是 `false` 。

例如:

```
var buf = new Buffer(4);
```

```
buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

for (ii = 0; ii < buf.length; ii++) {
  console.log(buf.readUInt8(ii));
}

// 0x3
// 0x4
// 0x23
// 0x42
```

buf.readUInt16LE(offset[, noAssert])

buf.readUInt16BE(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从 buffer 对象里，根据指定的偏移量，使用特殊的 endian 字节序格式读取一个有符号 16 位整数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 `offset` 可能会超出 buffer 的末尾。默认是 `false` 。

例如:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

console.log(buf.readUInt16BE(0));
console.log(buf.readUInt16LE(0));
console.log(buf.readUInt16BE(1));
console.log(buf.readUInt16LE(1));
console.log(buf.readUInt16BE(2));
console.log(buf.readUInt16LE(2));
```

```
// 0x0304
// 0x0403
// 0x0423
// 0x2304
// 0x2342
// 0x4223
```

buf.readUInt32LE(offset[, noAssert])

buf.readUInt32BE(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 `offset` 可能会超出buffer 的末尾。默认是 `false` 。

例如:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

console.log(buf.readUInt32BE(0));
console.log(buf.readUInt32LE(0));

// 0x03042342
// 0x42230403
```

buf.readInt8(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，读取一个 signed 8 位整数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 `false` 。

返回和 `buffer.readUInt8` 一样，除非 buffer 中包含了有作为2 的补码的有符号值。

buf.readInt16LE(offset[, noAssert])

buf.readInt16BE(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，使用特殊的 endian 格式读取一个 signed 16 位整数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 `false` 。

返回和 `buffer.readUInt16` 一样，除非 buffer 中包含了有作为 2 的补码的有符号值。

buf.readInt32LE(offset[, noAssert])

buf.readInt32BE(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可 位选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，使用指定的 endian 字节序格式读取一个 signed 32 位整数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 `offset` 可能会超出buffer 的末尾。默认是 `false` 。

和 `buffer.readUInt32` 一样返回，除非 buffer 中包含了有作为 2 的补码的有符号值。

buf.readFloatLE(offset[, noAssert])

buf.readFloatBE(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位浮点数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 `offset` 可能会超出buffer的末尾。默认是 `false` 。

例如:

```
var buf = new Buffer(4);

buf[0] = 0x00;
buf[1] = 0x00;
buf[2] = 0x80;
buf[3] = 0x3f;

console.log(buf.readFloatLE(0));

// 0x01
```

buf.readDoubleLE(offset[, noAssert])

buf.readDoubleBE(offset[, noAssert])

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false
- 返回： Number 类型

从这个 buffer 对象里，根据指定的偏移量，使用指定的 endian字节序格式读取一个 64 位double。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 `offset` 可能会超出buffer 的末尾。默认是 `false` 。

例如:

```
var buf = new Buffer(8);

buf[0] = 0x55;
buf[1] = 0x55;
buf[2] = 0x55;
buf[3] = 0x55;
buf[4] = 0x55;
buf[5] = 0x55;
buf[6] = 0xd5;
buf[7] = 0x3f;

console.log(buf.readDoubleLE(0));

// 0。3333333333333333
```

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

- `value` Number 类型
- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false

根据传入的 `offset` 偏移量将 `value` 写入 buffer。注意： `value` 必须是一个合法的有符号 8 位整数。

若参数 `noAssert` 为 true 将不会验证 `offset` 偏移量参数。 这意味着 `value` 可能过大，或者 `offset` 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则不要使用。默认是 `false` 。

例如:

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

console.log(buf);

// <Buffer 03 04 23 42>
```

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

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

- `value` Number 类型
- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false

根据传入的 `offset` 偏移量和指定的 endian 格式将 `value` 写入 buffer。注意： `value` 必须是一个合法的有符号 16 位整数。

若参数 `noAssert` 为 true 将不会验证 `value` 和 `offset` 偏移量参数。 这意味着 `value` 可能过大，或者 `offset` 可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 `false` 。

例如:

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

console.log(buf);

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

console.log(buf);

// <Buffer de ad be ef>
// <Buffer ad de ef be>
```

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

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

- `value` Number 类型
- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false

根据传入的 offset 偏移量和指定的 endian 格式将 value 写入buffer。注意： value 必须是一个合法的有符号 32 位整数。

若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。

例如:

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

console.log(buf);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);

// <Buffer fe ed fa ce>
// <Buffer ce fa ed fe>
```

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

- value Number 类型
- offset Number 类型
- noAssert Boolean, 可选参数， 默认值: false

根据传入的 offset 偏移量将 value 写入 buffer 。注意： value 必须是一个合法的 signed 8 位整数。

若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。

和 buffer.writeUInt8 一样工作，除非是把有2的补码的 有符号整数 有符号整形写入buffer。

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

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

- value Number 类型

- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false

根据传入的 `offset` 偏移量和指定的 endian 格式将 `value` 写入 buffer。注意： `value` 必须是一个合法的 signed 16 位整数。

若参数 `noAssert` 为 true 将不会验证 `value` 和 `offset` 偏移量参数。 这意味着 `value` 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 `value` 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 `false` 。

和 `buffer.writeUInt16*` 一样工作，除非是把有2的补码的有符号整数 有符号整形写入 `buffer` 。

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

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

- `value` Number 类型
- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false

根据传入的 `offset` 偏移量和指定的 endian 格式将 `value` 写入 buffer。注意： `value` 必须是一个合法的 signed 32 位整数。

若参数 `noAssert` 为 true 将不会验证 `value` 和 `offset` 偏移量参数。 这意味着 `value` 可能过大，或者 `offset` 可能会超出 buffer 的末尾从而造成 `value` 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 `false` 。

和 `buffer.writeUInt32*` 一样工作，除非是把有2的补码的有符号整数 有符号整形写入buffer。

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

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

- `value` Number 类型
- `offset` Number 类型
- `noAssert` Boolean, 可选参数， 默认值: false

根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer 。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。

若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。

例如:

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

console.log(buf);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);

// <Buffer 4f 4a fe bb>
// <Buffer bb fe 4a 4f>
```

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

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

- value Number 类型
- offset Number 类型
- noAssert Boolean, 可选参数， 默认值: false

根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意： value 必须是一个有效的 64 位double 类型的值。

若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。

例如:

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

```
console.log(buf);

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);

// <Buffer 43 eb d5 b7 dd f9 5f d7>
// <Buffer d7 5f f9 dd b7 d5 eb 43>
```

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

- `value`
- `offset` Number 类型， Optional
- `end` Number 类型， Optional

使用指定的 `value` 来填充这个 buffer。如果没有指定 `offset` (默认是 0) 并且 `end` (默认是 `buffer.length`)，将会填充整个buffer。

```
var b = new Buffer(50);
b.fill("h");
```

buffer.iNSPECT_MAX_BYTES

- Number 类型， 默认值: 50

设置当调用 `buffer.inspect()` 方法后，将会返回多少 bytes 。用户模块重写这个值可以。

注意这个属性是 `require('buffer')` 模块返回的。这个属性不是在全局变量 Buffer 中，也不在 buffer 的实例里。

类: SlowBuffer

返回一个不被池管理的 `Buffer` 。

大量独立分配的 Buffer 容易带来垃圾，为了避免这个情况，小于 4KB 的空间都是切割自一个较大的独立对象。这种策略既提高了性能也改善了内存使用率。V8 不需要跟踪和清理过多的 `Persistent` 对象。

当开发者需要将池中一小块数据保留一段时间，比较好的办法是用 SlowBuffer 创建一个不被池管理的 Buffer 实例，并将相应数据拷贝出来。

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

socket。on('readable', function() {
  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);
});
```

请谨慎使用，仅作为经常发现他们的应用中过度的内存保留时的最后手段。

C/C++ 插件

插件 Addons 是动态链接的共享对象。他提供了 C/C++ 类库能力。这些API比较复杂，他包以下几个类库:

- V8 JavaScript, C++ 类库。用来和 JavaScript 交互，比如创建对象，调用函数等等。 在 `v8.h` 头文件中(目录地址`deps/v8/include/v8.h`)，线上地址 online (http://izs.me/v8-docs/main.html) 。
- libuv (https://github.com/joyent/libuv) ，C 事件循环库。 等待文件描述符变为可读，等待定时器，等待信号时，会和 libuv 打交道。或者说，如果你需要和 I/O 打交道，就会用到 libuv。
- 内部 Node 类库。 其中最重要的类 `node::ObjectWrap` ，你会经常派生自它。
- 其他的参见 `deps/` 。

Node 已经将所有的依赖编译成可以执行文件，所以你不必当心这些类库的链接问题。

以下所有例子可以在 download (https://github.com/rvagg/node-addon-examples) 下载，也许你可以从中找一个作为你的扩展插件。

Hello world

现在我们来写一个 C++ 插件的小例子，它的效果和以下 JS 代码一致:

```
module.exports.hello = function() { return 'world'; };
```

创建 `hello.cc` 文件:

```
// hello.cc
#include <node.h>

using namespace v8;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}

void init(Handle<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(addon, init)
```

注意：所有的 Node 插件必须输出一个初始化函数:

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

`NODE_MODULE` 之后的代码没有分号，因为它不是一个函数 (参见 `node.h`)。

`module_name` 必须和二进制文件名字一致 (后缀是 .node)。

源文件会编译成 `addon.node` 二进制插件。 为此我们创建了一个很像 JSON 的 `binding.gyp` 文件， 它包含配置信息，这个文件用node-gyp (https://github.com/TooTallNate/node-gyp) 编译。

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

下一步创建一个 `node-gyp configure` 工程，在平台上生成这些文件。

创建后，在 `build/` 文件夹里拥有一个 `Makefile` (Unix 系统) 文件或者 `vcxproj` 文件 (Windows 系统)。 接着调用 `node-gyp build` 命令编译，生成 `.node` 文件。 这些文件在 `build/Release/` 目录里。

现在，你能在 Node 工程中使用这些 2 进制扩展插件，在 `hello.js` 中声明 `require` 之前编译的 `hello.node` :

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

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

更多的信息请参考https://github.com/arturadib/node-qt 。

插件模式

下面是一些 addon 插件的模式，帮助你开始编码。v8 reference (http://izs.me/v8-docs/main.html) 文档里包含 v8 的各种接口，Embedder's Guide (http://code.google.com/apis/v8/embed.html) 这个文档包含各种说明，比如 handles, scopes, function templates, 等等。

在使用这些例子前，你需要先用 `node-gyp` 编译。 创建 `binding.gyp` 文件:

```
{
  "targets": [
    {
```

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

将文件名加入到 `sources` 数组里就可以使用多个 `.cc` 文件，例如：

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

准备好 `binding.gyp` 文件后，你就能配置并编译插件：

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

函数参数

从以下模式中解释了如何从 JavaScript 函数中读取参数，并返回结果。仅需要一个 `addon.cc` 文件：

```
// addon.cc
#include <node.h>

using namespace v8;

void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  if (args.Length() < 2) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong number of arguments")));
    return;
  }

  if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong arguments")));
    return;
  }

  double value = args[0]->NumberValue() + args[1]->NumberValue();
  Local<Number> num = Number::New(isolate, value);

  args.GetReturnValue().Set(num);
}
```

```
void Init(Handle<Object> exports) {
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, Init)
```

可以用以下的 JavaScript 代码片段测试：

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

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

回调Callbacks

你也能传 JavaScript 函数给 C++ 函数，并执行它。在 `addon.cc` 中:

```
// addon.cc
#include <node.h>

using namespace v8;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  Local<Function> cb = Local<Function>::Cast(args[0]);
  const unsigned argc = 1;
  Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world") };
  cb->Call(isolate->GetCurrentContext()->Global(), argc, argv);
}

void Init(Handle<Object> exports, Handle<Object> module) {
  NODE_SET_METHOD(module, "exports", RunCallback);
}

NODE_MODULE(addon, Init)
```

注意，这个例子中使用了 `Init()` 里的 2 个参数，`module` 对象是第二个参数。它允许 addon 使用一个函数完全重写 `exports` 。

可以用以下的代码来测试：

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

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

对象工厂

在 addon.cc 模式里，你能用 C++ 函数创建并返回一个新的对象，这个对象所包含的 msg 属性是由 createObject() 函数传入:

```
// addon.cc
#include <node.h>

using namespace v8;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  Local<Object> obj = Object::New(isolate);
  obj->Set(String::NewFromUtf8(isolate, "msg"), args[0]->ToString());

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

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

NODE_MODULE(addon, Init)
```

使用 JavaScript 测试:

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

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

工厂模式

这个模式里展示了如何创建并返回一个 JavaScript 函数，它是由 C++ 函数包装的 :

```
// addon.cc
#include <node.h>

using namespace v8;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "hello world"));
}

void CreateFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
  Local<Function> fn = tpl->GetFunction();

  // omit this to make it anonymous
  fn->SetName(String::NewFromUtf8(isolate, "theFunction"));

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

void Init(Handle<Object> exports, Handle<Object> module) {
  NODE_SET_METHOD(module, "exports", CreateFunction);
}

NODE_MODULE(addon, Init)
```

测试:

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

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

包装 C++ 对象

以下会创建一个 C++对象的包装 `MyObject` ，这样 他就能再 JavaScript 中用 `new` 实例化。首先在 `addon.cc` 中准备主要模块:

```
// addon.cc
#include <node.h>
#include "myobject.h"

using namespace v8;

void InitAll(Handle<Object> exports) {
  MyObject::Init(exports);
}

NODE_MODULE(addon, InitAll)
```

接着在 myobject.h 创建包装，它继承自 node::ObjectWrap :

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Handle<v8::Object> exports);

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

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

#endif
```

在 myobject.cc 中实现各种暴露的方法，通过给构造函数添加 prototype 属性来暴露 plusOne 方法:

```
// myobject.cc
#include "myobject.h"

using namespace v8;

Persistent<Function> MyObject::constructor;
```

```
MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Handle<Object> exports) {
  Isolate* isolate = Isolate::GetCurrent();

  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  constructor.Reset(isolate, tpl->GetFunction());
  exports->Set(String::NewFromUtf8(isolate, "MyObject"),
               tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
```

```
  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}
```

测试:

```
// test.js
var 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

当你想创建本地对象，又不想在 JavaScript 中严格的使用 `new` 初始化的时候，以下方法非常实用。

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

在 `addon.cc` 中注册 `createObject` 方法:

```
// addon.cc
#include <node.h>
#include "myobject.h"

using namespace v8;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
  MyObject::NewInstance(args);
}

void InitAll(Handle<Object> exports, Handle<Object> module) {
  MyObject::Init();

  NODE_SET_METHOD(module, "exports", CreateObject);
}

NODE_MODULE(addon, InitAll)
```

在 `myobject.h` 中有静态方法 `NewInstance`，他能实例化对象（它就像 JavaScript 的 `new`）:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init();
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);

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

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

#endif
```

这个实现方法和 `myobject.cc` 类似:

```
// myobject.cc
#include <node.h>
#include "myobject.h"

using namespace v8;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init() {
  Isolate* isolate = Isolate::GetCurrent();
  // Prepare constructor template
```

```
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

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

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

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
```

```
  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}
```

测试:

```
// test.js
var 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

除了包装并返回 C++ 对象，你可以使用 Node 的 `node::ObjectWrap::Unwrap` 帮助函数来解包。在下面的 `addon.cc` 中，我们介绍了一个 `add()` 函数，它能获取2个 `MyObject` 对象：

```
// addon.cc
#include <node.h>
#include <node_object_wrap.h>
#include "myobject.h"

using namespace v8;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
  MyObject::NewInstance(args);
}

void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  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(Handle<Object> exports) {
  MyObject::Init();

  NODE_SET_METHOD(exports, "createObject", CreateObject);
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(addon, InitAll)
```

介绍 `myobject.h` 里的一个公开方法，它能在解包后使用私有变量:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

class MyObject : public node::ObjectWrap {
 public:
  static void Init();
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);
  inline double value() const { return value_; }

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

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

#endif
```

`myobject.cc` 的实现方法和之前的类似:

```
// myobject.cc
#include <node.h>
#include "myobject.h"

using namespace v8;

Persistent<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init() {
  Isolate* isolate = Isolate::GetCurrent();

  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  constructor.Reset(isolate, tpl->GetFunction());
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(argc, argv));
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
```

```
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

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

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

测试:

```
// test.js
var 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
```

5

子进程

稳定性: 3 – 稳定

Node 通过 child_process 模块提供了 popen(3) 数据流。

它能在非阻塞的方式中，通过 stdin , stdout , 和 stderr 传递数据。(请注意，某些程序使用内部线性缓冲 I/O， 它并不妨碍 node.js，只是你发送给子进程的数据不会被立即消。)

使用 require('child_process').spawn() 或 require('child_process').fork() 创建一个子进程。这两种方法有区别，底下会解释 below (页 0).

开发过程中查看 synchronous counterparts (页 0) 效率会更高。

类: ChildProcess

ChildProcess 是一个 EventEmitter (events.html#events_class_events_eventemitter) 。

子进程有三个相关的流 child.stdin , child.stdout , 和 child.stderr 。他们可能和会父进程的 stdio streams 共享, 也可作为独立的对象。

不能直接调用 ChildProcess 类，使用 spawn() , exec() , execFile() , 或 fork() 方法来创建子进程的实例。

事件: 'error'

- err {Error Object} 错误。

发生于:

1. 无法创建进程。
2. 无法杀死进程。
3. 无论什么原因导致给子进程发送消息失败。

注意， exit 事件有可能在错误发生后调用，也可能不调用，所以如果你监听这两个事件来触发函数，记得预防函数会被调用2次。

参考 ChildProcess#kill() (页 0) 和 ChildProcess#send() (页 0)。

事件: 'exit'

- code {Number} 退出代码, 正常退出时才有效。
- signal {String} 如果是被父进程杀死，则它为传递给子进程的信号

子进程结束的时候触发这个事件。如果子进程正常终止，则 `code` 为最终的退出代码，否则为 `null` 。如果是由 `signal` 引起的终止，则 `signal` 为字符串，否则为 `null` 。

注意，子进程的 stdio 流可能仍为开启模式。

注意，node 为 `'SIGINT'` 和 `'SIGTERM'` 建立句柄，所以当信号来临的时候，他们不会终止而是退出。

参见 `waitpid(2)` 。

事件: 'close'

- `code` {Number} 退出代码, 正常退出时才有效。
- `signal` {String} 如果是被父进程杀死，则它为传递给子进程的信号。

子进程里所有 stdio 流都关闭时触发这个事件。要和 'exit' 区分开, 因为多进程可以共享一个 stdio 流。

Event: 'disconnect'

父进程或子进程中调用 `.disconnect()` 方法后触发这个事件。 断开后不会在互发消息，并且 `.connected` 属性值为 false。

Event: 'message'

- `message` {Object} 一个解析过的 JSON 对象，或者一个原始值
- `sendHandle` {Handle object} 一个 Socket 或 Server 对象

通过 `.send(message, [sendHandle])` 传递消息。

child.stdin

- {Stream object}

子进程的 `stdin` 是 `Writable Stream`（可写流）。如果子进程在等待输入，它就会暂停直到通过调用 `end()` 来关闭。

`child.stdin` 是 `child.stdio[0]` 的缩写。这两个都指向同一个对象，或者 null。

child.stdout

- {Stream object}

子进程的 `stdout` 是 `Readable Stream`（可读流）。

`child.stdout` 是 `child.stdio[1]` 的缩写。这两个都指向同一个对象，或者 null。

child.stderr

- {Stream object}

子进程的 `stderr` 是 `Readable Stream`（可写流）。

`child.stderr` 是 `child.stdio[2]` 缩写。这两个都指向同一个对象，或者 null。

child.stdio

- {Array}

子进程的管道数组和 spawn (页 0) 的 stdio (页 0) 里设置为 `'pipe'` 的内容次序相对应。

注意，流[0-2]也能分别用 ChildProcess.stdin,ChildProcess.stdout, 和 ChildProcess.stderrNote 来表示。

在下面的例子里，只有子进程的 fd `1` 设置为 pipe管道，所以父进程的 `child.stdio[1]` 是流（stream）, 数组里其他值为 `null`。

```
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.pid

- {Integer}

子进程的 PID。

例子:

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

console.log('Spawned child pid: ' + grep.pid);
grep.stdin.end();
```

child.connected

- {Boolean} 调用 `.disconnect' 后设置为 false

如果 `.connected` 为 false, 消息不再可用。

child.kill([signal])

- `signal` {String}

发送信号给子进程。如果没有参数，会发送 `'SIGTERM'`，参见 `signal(7)` 里的可用的信号列表。

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

grep.on('close', function (code, signal) {
  console.log('child process terminated due to receipt of signal '+signal);
});

// send SIGHUP to process
grep.kill('SIGHUP');
```

当信号无法传递的时候会触发 `'error'` 事件。给已经终止的进程发送信号不会触发 `'error'` 事件，但是可以能引起不可预知的后果: 因为有可能 PID (进程 ID) 已经重新分配给其他进程, 信号就会被发送到新的进程里，无法想象这样会引发什么样的事情。

注意，当函数调用 kill 信号的时候，它实际并并不会杀死进程，只是发送信号给进程。

参见 kill(2)

child.send(message[, sendHandle])

- message {Object}
- sendHandle {Handle object}

使用 child_process.fork() 的时候，你能用 child.send(message, [sendHandle]) 给子进程写数据，子进程通过 'message' 接收消息。

例如:

```
var cp = require('child_process');

var n = cp.fork(__dirname + '/sub.js');

n.on('message', function(m) {
  console.log('PARENT got message:', m);
});

n.send({ hello: 'world' });
```

子进程的代码 'sub.js' :

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

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

子进程代码里的 process 对象拥有 send() 方法，当它通过信道接收到信息时会触发，并返回对象。

注意，父进程和子进程 send() 是同步的，不要用来发送大块的数据（可以用管道来代替，参见 child_process.spawn (页 0))。

不过发送 {cmd: 'NODE_foo'} 消息是特殊情况。所有包含 NODE_ 前缀的消息都不会被触发，因为它们是 node 的内部的核心消息，它们会在 internalMessage 事件里触发，尽量避免使用这个特性。

child.send() 里的 sendHandle 属性用来发送 TCP 服务或 socket 对象给其他的进程，子进程会用接收到的对象作为 message 事件的第二个参数。

如果不能发出消息会触发 'error' 事件，比如子进程已经退出。

例子: 发送 server 对象

以下是例子:

```
var child = require('child_process').fork('child.js');

// Open up the server object and send the handle.
var server = require('net').createServer();
server.on('connection', function (socket) {
  socket.end('handled by parent');
});
server.listen(1337, function() {
  child.send('server', server);
});
```

子进程将会收到这个 server 对象:

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

注意，现在父子进程共享了server，某些连接会被父进程处理，某些会被子进程处理。

dgram 服务器，工作流程是一样的，监听的是 message 事件，而不是 connection，使用 server.bind 而不是 server.listen。(目前仅支持 UNIX 平台)

例子: 发送 socket 对象

以下是发送 socket 对象的例子。 他将会创建 2 个子线程，并且同时处理连接，一个将远程地址 74.125.127.100 当做 VIP 发送到一个特殊的子进程，另外一个发送到正常进程。 var normal = require('child_process').fork('child.js', ['normal']); var special = require('child_process').fork('child.js', ['special']);

```
// Open up the server and send sockets to child
var server = require('net').createServer();
server.on('connection', function (socket) {

  // if this is a VIP
  if (socket.remoteAddress === '74.125.127.100') {
    special.send('socket', socket);
    return;
  }
```

```
  // just the usual dudes
  normal.send('socket', socket);
});
server.listen(1337);
```

`child.js` 代码如下:

```
process.on('message', function(m, socket) {
  if (m === 'socket') {
    socket.end('You were handled as a ' + process.argv[2] + ' person');
  }
});
```

注意，当 socket 发送给子进程后，如果这个 socket 被销毁，父进程不再跟踪它，相应的 `.connections` 属性会变为 `null` 。这种情况下，不建议使用 `.maxConnections` 。

child.disconnect()

关闭父子进程间的所有 IPC 通道，能让子进程优雅的退出。调用这个方法后，父子进程里的 `.connected` 标志会变为 `false` ，之后不能再发送消息。

当进程里没有消息需要处理的时候，会触发 'disconnect' 事件。

注意，在子进程还有 IPC 通道的情况下（如 `fork()` ），也可以调用 `process.disconnect()` 来关闭它。

创建异步处理

这些方法遵从常用的异步处理模式（比如回调，或者返回一个事件处理）。

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

- `command` {String} 要运行的命令
- `args` {Array} 字符串参数表
- `options` {Object}
  - `cwd` {String} 子进程的工作目录
  - `env` {Object} 环境
  - `stdio` {Array|String} 子进程的 stdio 配置。(见 below (页 0))
  - `customFds` {Array} **Deprecated** 作为子进程 stdio 使用的 文件标示符。(见 below (页 0))

- `detached` {Boolean} 子进程将会变成一个进程组的领导者。(参见 below (页 0))
- `uid` {Number} 设置用户进程的ID。(参见 setuid(2))
- `gid` {Number} 设置进程组的ID。(参见 setgid(2))

- 返回: {ChildProcess object}

用指定的 `command` 发布一个子进程, `args` 是命令行参数。如果忽略, `args` 是空数组。

第三个参数用来指定附加设置，默认值:

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

创建的子进程里使用 `cwd` 指定工作目录，如果没有指定，默认继承自当前的工作目录。

使用 `env` 来指定新进程可见的环境变量。默认是 `process.env` 。

例如，运行 `ls -lh /usr` , 获取 `stdout` , `stderr` , 和退出代码:

```
var spawn = require('child_process').spawn,
    ls    = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

ls.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

ls.on('close', function (code) {
  console.log('child process exited with code ' + code);
});
```

例如: 一个非常精巧的方法执行 'ps ax | grep ssh'

```
var spawn = require('child_process').spawn,
    ps    = spawn('ps', ['ax']),
    grep  = spawn('grep', ['ssh']);

ps.stdout.on('data', function (data) {
  grep.stdin.write(data);
});
```

```
ps.stderr.on('data', function (data) {
  console.log('ps stderr: ' + data);
});

ps.on('close', function (code) {
  if (code !== 0) {
    console.log('ps process exited with code ' + code);
  }
  grep.stdin.end();
});

grep.stdout.on('data', function (data) {
  console.log('' + data);
});

grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

grep.on('close', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

options.stdio

`stdio` 可能是以下几个参数之一:

- `'pipe'` - `['pipe', 'pipe', 'pipe']`, 默认值
- `'ignore'` - `['ignore', 'ignore', 'ignore']`
- `'inherit'` - `[process.stdin, process.stdout, process.stderr]` 或 `[0,1,2]`

`child_process.spawn()` 里的 'stdio' 参数是一个数组，它和子进程的 fd 相对应，它的值如下:

1. `'pipe'` - 创建在父进程和子进程间的 pipe。管道的父进程端以 `child_process` 的属性形式暴露给父进程，例如 `ChildProcess.stdio[fd]` 。为 fds 0 - 2 创建的管道也可以通过 ChildProcess.stdin, ChildProcess.stdout 和 ChildProcess.stderr 来独立的访问。
2. `'ipc'` - 在父进程和子进程间创建一个 IPC 通道来传递消息/文件描述符。 一个子进程最多有1个 IPC stdio 文件标识。设置这个选项会激活 ChildProcess.send() 方法。如果子进程向此文件标识写入 JSON 消

息，则会触发 ChildProcess.on('message') 。如果子进程是 Node.js 程序，那么 IPC 通道会激活 process.send() 和 process.on('message')。

3. 'ignore' – 在子进程里不要设置这个文件标识，注意，Node 总会为其 spawn 的进程打开 fd 0-2。如果任何一个被 ignored，node 将会打开 /dev/null 并赋给子进程的 fd。

4. Stream 对象 – 共享一个tty, file, socket, 或刷（ pipe ）可读或可写流给子进程。该流底层（ underlying ）的文件标识在子进程中被复制给stdio数组索引对应的文件标识（ fd ）。

5. 正数 – 这个整数被理解为一个在父进程中打开的文件标识，它和子进程共享，就和共享 Stream 对象类似。

6. null , undefined – 使用默认值。 对于 stdio fds 0, 1 and 2 (换句话说, stdin, stdout, and stderr)，pipe管道被建立。 对于 fd 3 及之后, 默认是 'ignore' 。

例如:

```
var 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 present a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

options.detached

如果设置了 detached 选项, 子进程将会被作为新进程组的leader，这使得子进程可以在父进程退出后继续运行。

缺省情况下父进程会等 detached 的子进程退出。要阻止父进程等待一个这样的子进程，调用 child.unref() 方法，则父进程的事件循环引用计数中将不会包含这个子进程。

detaching 一个长期运行的进程，并重新将输出指向文件：

```
var fs = require('fs'),
    spawn = require('child_process').spawn,
    out = fs.openSync('./out.log', 'a'),
    err = fs.openSync('./out.log', 'a');
```

```
var child = spawn('prg', [], {
  detached: true,
  stdio: [ 'ignore', out, err ]
});

child.unref();
```

使用 `detached` 选项来启动一个长时间运行的进程时，进程不会在后台保持运行，除非他提供了一个不连接到父进程的 `stdio` 。如果继承了父进程的 `stdio` ，则子进程会继续控制终端。

options.customFds

已废弃， `customFds` 允许指定特定文件描述符作为子进程的 `stdio` 。该 API 无法移植到所有平台，因此被废弃。使用 `customFds` 可以将新进程的 [`stdin` , `stdout` , `stderr`] 钩到已有流上； -1 表示创建新流。自己承担使用风险。

参见: `child_process.exec()` and `child_process.fork()`

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

- `command` {String} 要执行的命令，空格分割
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `env` {Object} 环境变量
  - `encoding` {String} (默认: 'utf8')
  - `shell` {String} 运行命令的 shell (默认: '/bin/sh' UNIX, 'cmd.exe' Windows, 该 shell 必须接收 UNIX 上的 `-c` 开关 ，或者 Windows上的 `/s /c` 开关 。Windows 上，命令解析必须兼容 `cmd.exe` 。)
  - `timeout` {Number} (默认: 0)
  - `maxBuffer` {Number} (默认: `200*1024`)
  - `killSignal` {String} (默认: 'SIGTERM')
  - `uid` {Number} 设置进程里的用户标识。(见 setuid(2)。)
  - `gid` {Number} 设置进程里的群组标识。(见 setgid(2)。)
- `callback` {Function} 进程终止的时候调用

    - `error` {Error}
    - `stdout` {Buffer}
    - `stderr` {Buffer}
- 返回: ChildProcess 对象

在 shell 里执行命令，并缓冲输出。

```
var exec = require('child_process').exec,
    child;

child = exec('cat *.js bad_file | wc -l',
  function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);
    if (error !== null) {
      console.log('exec error: ' + error);
    }
});
```

回调参数是 `(error, stdout, stderr)`。如果成功，`error` 值为 `null`。如果失败，`error` 变为 `Error` 的实例，`error.code` 等于子进程退出码，并且 `error.signal` 会被设置为结束进程的信号名。

第二个参数可以设置一些选项。缺省是：

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

如果 `timeout` 大于 0，子进程运行时间超过 `timeout` 时会被终止。`killSignal`（默认：`'SIGTERM'`）能杀死子进程。`maxBuffer` 设定了 stdout 或 stderr 的最大数据量，如果子进程的数量量超过了，将会被杀死。

(file[, args][, options][, callback])

- `file` {String} 要运行的程序的文件名
- `args` {Array} 参数列表
- `options` {Object}

    - `cwd` {String} 子进程的工作目录
    - `env` {Object} 环境
    - `encoding` {String} (默认: 'utf8')
    - `timeout` {Number} (默认: 0)
    - `maxBuffer` {Number} (默认: 200*1024)
    - `killSignal` {String} (默认: 'SIGTERM')
    - `uid` {Number} 设置进程里的用户标识。(见 setuid(2)。)
    - `gid` {Number} 设置进程里的群组标识。(见 setgid(2)。)
- `callback` {Function} 进程终止的时候调用
    - `error` {Error}
    - `stdout` {Buffer}
    - `stderr` {Buffer}
- 返回: ChildProcess 对象

和 `child_process.exec()` 类似，不同之处在于这是执行一个指定的文件，因此它比 `child_process.exec` 精简些，参数相同。

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

- `modulePath` {String} 子进程里运行的模块
- `args` {Array} 参数列表
- `options` {Object}
    - `cwd` {String} 子进程的工作目录
    - `env` {Object} 环境
    - `execPath` {String} 执行文件路径
    - `execArgv` {Array} 执行参数 (默认: `process.execArgv`)
    - `silent` {Boolean} 如果是 true ，子进程将会用父进程的 stdin, stdout, and stderr ，否则，将会继承自父进程, 更多细节，参见 `spawn()` 的 `stdio` 参数里的 "pipe" 和 "inherit" 选项(默认 false)
    - `uid` {Number} 设置进程里的用户标识。(见 setuid(2)。)

    - `gid` {Number} 设置进程里的群组标识。(见 setgid(2)。)
- 返回: ChildProcess 对象

这是 `spawn()` 的特殊例子，用于派生 Node 进程。除了拥有子进程的所有方法，它的返回对象还拥有内置通讯通道。参见 `child.send(message, [sendHandle])` 。

这些 Nodes 是全新的 V8 实例化，假设每个 Node 最少需要 30ms 的启动时间，10mb 的存储空间，可想而知，创建几千个 Node 是不太现实的。

`options` 对象中的 `execPath` 属性可以用于执行文件（非当前 `node` ）创建子进程。这需要小心使用，缺省情况下 fd 表示子进程的 `NODE_CHANNEL_FD` 环境变量。该 fa 的输入和输出是以行分割的 JSON 对象。

创建同步进程

以下这些方法是**同步**的，意味着他会**阻塞**事件循环，并暂停执行代码，直到 spawned 的进程退出。

同步方法简化了任务进程，比如大为简化在应用初始化加载/处理过程。

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

- `command` {String} 要执行的命令
- `args` {Array} 参数列表
- `options` {Object}
    - `cwd` {String} 子进程的当前工作目录
    - `input` {String|Buffer} 传递给spawned 进程的值，这个值将会重写 `stdio[0]`
    - `stdio` {Array} 子进程的 stdio 配置。
    - `env` {Object} 环境变量
    - `uid` {Number} 设置用户进程的ID。(参见 setuid(2)。)
    - `gid` {Number} 设置进程组的ID。(参见 setgid(2)。)
    - `timeout` {Number} 子进程运行最大毫秒数。(默认: undefined)
    - `killSignal` {String} 用来终止子进程的信号。(默认: 'SIGTERM')
    - `maxBuffer` {Number}
    - `encoding` {String} stdio 输入和输出的编码方式。(默认: 'buffer')

- 返回: {Object}
  - `pid` {Number} 子进程的 pid
  - `output` {Array} stdio 输出的结果数组
  - `stdout` {Buffer|String} `output[1]` 的内容
  - `stderr` {Buffer|String} `output[2]` 的内容
  - `status` {Number} 子进程的退出代码
  - `signal` {String} 用来杀死子进程的信号
  - `error` {Error} 子进程错误或超时的错误代码

`spawnSync` 直到子进程关闭才会返回。超时或者收到 `killSignal` 信号，也不会返回，直到进程完全退出。进程处理完 `SIGTERM` 信号后并不会结束，直到子进程完全退出。

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

- `command` {String} 要执行的命令
- `args` {Array} 参数列表
- `options` {Object}
  - `cwd` {String} 子进程的当前工作目录
  - `input` {String|Buffer}传递给spawned 进程的值，这个值将会重写 `stdio[0]`
  - `stdio` {Array}子进程的 stdio 配置。(默认: 'pipe')
  - `stderr` 默认情况下会输出给父进程的' stderr 除非指定了 `stdio`
  - `env` {Object} 环境变量
  - `uid` {Number} 设置用户进程的ID。(参见 setuid(2)。)
  - `gid` {Number} 设置进程组的ID。(参见 setgid(2)。)
  - `timeout` {Number} 进程运行最大毫秒数。(默认: undefined)
  - `killSignal` {String} 用来终止子进程的信号。(默认: 'SIGTERM')
  - `maxBuffer` {Number}
  - `encoding` {String} stdio 输入和输出的编码方式。(默认: 'buffer')
- 返回: {Buffer|String} 来自命令的 stdout

直到子进程完全退出， execFileSync 才会返回。超时或者收到 killSignal 信号，也不会返回，直到进程完全退出。进程处理完 SIGTERM 信号后并不会结束，直到子进程完全退出。

如果进程超时，或者非正常退出，这个方法将会抛出异常。 Error 会包含整个 child_process.spawnSync (页0)结果。

child_process.execSync(command[, options])

- command {String} 要执行的命令
- options {Object}
  - cwd {String} 子进程的当前工作目录
  - input {String|Buffer} 传递给spawned 进程的值，这个值将会重写 stdio[0]
  - stdio {Array} 子进程的 stdio 配置。(默认: 'pipe')
  - stderr 默认情况下会输出给父进程的' stderr 除非指定了 stdio
  - env {Object} 环境变量
  - uid {Number} 设置用户进程的ID。(参见 setuid(2)。)
  - gid {Number} 设置进程组的ID。(参见 setgid(2)。)
  - timeout {Number} 进程运行最大毫秒数。(默认: undefined)
  - killSignal {String} 用来终止子进程的信号。(默认: 'SIGTERM')
  - maxBuffer {Number}
  - encoding {String} stdio 输入和输出的编码方式。(默认: 'buffer')
- 返回: {Buffer|String} 来自命令的 stdout

直到子进程完全退出， execSync 才会返回。超时或者收到 killSignal 信号，也不会返回，直到进程完全退出。进程处理完 SIGTERM 信号后并不会结束，直到子进程完全退出。

如果进程超时，或者非正常退出，这个方法将会抛出异常。 Error 会包含整个 child_process.spawnSync (页0)结果。

6

集群

```
稳定性: 2 - 不稳定
```

单个 Node 实例运行在一个线程中。为了更好的利用多核系统的能力，可以启动 Node 集群来处理负载。

在集群模块里很容易就能创建一个共享所有服务器接口的进程。

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', function(worker, code, signal) {
    console.log('worker ' + worker.process.pid + ' died');
  });
} else {
  // Workers can share any TCP connection
  // In this case its a HTTP server
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");
  }).listen(8000);
}
```

运行 Node 后，将会在所有工作进程里共享 8000 端口。

```
% NODE_DEBUG=cluster node server.js
23521,Master Worker 23524 online
23521,Master Worker 23526 online
23521,Master Worker 23523 online
23521,Master Worker 23528 online
```

这个特性是最近才引入的，大家可以试试并提供反馈。

还要注意，在 Windows 系统里还不能在工作进程中创建一个被命名的管道服务器。

如何工作

`child_process.fork` 方法派生工作进程，所以它能通过 IPC 和父进程通讯，并相互传递句柄。

集群模块通过2种分发模式来处理连接。

第一种（默认方法，除了 Windows 平台）为循环式。主进程监听一个端口，接收新的连接，再轮流的分发给工作进程。

第二种，主进程监听 socket，并发送给感兴趣的工作进程，工作进程直接接收连接。

第二种方法理论上性能最高。实际上，由于操作系统各式各样，分配往往分配不均。列如，70%的连接终止于2个进程，实际上共有8个进程。

因为 `server.listen()` 将大部分工作交给了主进程，所以一个普通的 Node.js 进程和一个集群工作进程会在三种情况下有所区别：

1. `server.listen({fd: 7})` 由于消息被传回主进程，所以将会监听*主进程*里的文件描述符，而不是其他工作进程里的文件描述符 7。
2. `server.listen(handle)` 监听一个明确地句柄，会使得工作进程使用指定句柄，而不是与主进程通讯。如果工作进程已经拥有了该句柄，前提是您知道在做什么。
3. `server.listen(0)` 通常它会让服务器随机监听端口。然而在集群里每个工作进程 `listen(0)` 时会收到相同的端口。实际上仅第一次是随机的，之后是可预测的。如果你想监听一个特定的端口，可以根据集群的工作进程的ID生产一个端口ID 。

在 Node.js 或你的程序里没有路由逻辑，工作进程见也没有共享状态。因此，像登录和会话这样的工作，不要设计成过度依赖内存里的对象。

因为工作线程都是独立的，你可以根据需求来杀死或者派生而不会影响其他进程。只要仍然有工作进程，服务器还会接收连接。Node 不会自动管理工作进程的数量，这是你的责任，你可以根据自己需求来管理。

cluster.schedulingPolicy

调度策略 `cluster.SCHED_RR` 表示轮流制， `cluster.SCHED_NONE` 表示操作系统处理。这是全局性的设定，一旦你通过 `cluster.setupMaster()` 派生了第一个工作进程，它就不可更改了。

`SCHED_RR` 是除 Windows 外所有系统的默认设置。只要 libuv 能够有效地分配 IOCP 句柄并且不产生巨大的性能损失，Windows 也会改为 SCHED_RR 方式。

`cluster.schedulingPolicy` 也可通过环境变量 `NODE_CLUSTER_SCHED_POLICY` 来更改。有效值为 `"rr"` 和 `"none"` 。

cluster.settings

- {Object}
  - `execArgv` {Array} 传给可执行的 Node 的参数列表(默认= `process.execArgv`)
  - `exec` {String} 执行文件的路径。(默认= `process.argv[1]`)
  - `args` {Array} 传给工作进程的参数列表(默认= `process.argv.slice(2)`)
  - `silent` {Boolean}是否将输出发送给父进程的 stdio。(默认= `false`)
  - `uid` {Number} 设置用户进程的ID。(See setuid(2)。)
  - `gid` {Number} 设置进程组的ID。(See setgid(2)。)

调用 `.setupMaster()` (或 `.fork()`) 方法后，这个 settings 对象会包含设置内容，包括默认值。

设置后会立即冻结，因为 `.setupMaster()` 只能调用一次。

这个对象不应该被手动改变或设置。

cluster.isMaster

- {Boolean}

如果是主进程，返回 true。如果 `process.env.NODE_UNIQUE_ID` 未定义， `isMaster` 为 `true` 。

cluster.isWorker

- {Boolean}

如果不是主进程返回 true (和 `cluster.isMaster` 相反)。

事件: 'fork'

- `worker` {Worker object}

当一个新的工作进程被分支出来，集群模块会产生 'fork' 事件。它可用于记录工作进程，并创建自己的超时管理。

```
var timeouts = [];
function errorMsg() {
  console.error("Something must be wrong with the connection ...");
}

cluster.on('fork', function(worker) {
  timeouts[worker.id] = setTimeout(errorMsg, 2000);
});
cluster.on('listening', function(worker, address) {
  clearTimeout(timeouts[worker.id]);
});
cluster.on('exit', function(worker, code, signal) {
  clearTimeout(timeouts[worker.id]);
  errorMsg();
});
```

事件: 'online'

- `worker` {Worker object}

分支出一个新的工作进程后，它会响应在线消息。当主线程接收到在线消息后，它会触发这个事件。'fork' 和 'online' 之间的区别在于，主进程分支一个工作进程后会调用 fork，而工作进程运行后会调用 emitted。

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

事件: 'listening'

- `worker` {Worker object}
- `address` {Object}

工作进程调用 `listen()` 时，服务器会触发'listening'事件，同时也会在主进程的集群里触发。

事件处理函数有两个参数， `worker` 包含工作进程对象， `address` 包含以下属性： `address` ， `port` 和 `addressType` 。如果工作进程监听多个地址的时候，这些东西非常有用。

```
cluster.on('listening', function(worker, address) {
  console.log("A worker is now connected to " + address.address + ":" + address.port);
});
```

`addressType` 是以下内容:

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

事件: 'disconnect'

- `worker` {Worker object}

当一个工作进程的 IPC 通道关闭时会触发这个事件。当工作进程正常退出，被杀死，或者手工关闭（例如worker.disconnect()）时会调用。

`disconnect` 和 `exit` 事件间可能存在延迟。 这些事件可以用来检测进程是否卡在清理过程中，或者存在长连接。

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

事件: 'exit'

- `worker` {Worker object}
- `code` {Number} 如果正常退出，则为退出代码.
- `signal` {String} 使得进程被杀死的信号名 (比如. `'SIGHUP'`)

当任意一个工作进程终止的时候，集群模块会触发 'exit' 事件。

可以调用 `.fork()` 重新启动工作进程。

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

参见 child_process event: 'exit' (https://nodejs.org/api/child_process.html#child_process_event_exit) .

事件: 'setup'

- `settings` {Object}

调用 `.setupMaster()` 后会被触发。

`settings` 对象就是 `cluster.settings` 对象。

详细内容参见 `cluster.settings` 。

cluster.setupMaster([settings])

- `settings` {Object}
  - `exec` {String} 执行文件的路径。(默认= `process.argv[1]`)
  - `args` {Array}传给工作进程的参数列表(默认= `process.argv.slice(2)`)
  - `silent` {Boolean} 是否将输出发送给父进程的 stdio.

`setupMaster` 用来改变默认的 'fork' 。一旦调用，settings值将会出现在 `cluster.settings` 里。

注意:

- 改变任何设置，仅会对未来的工作进程产生影响，不会影响对目前已经运行的进程
- 工作进程里，仅能改变传递给 `.fork()` 的 `env` 属性。
- 以上的默认值，仅在第一次调用的时候有效，之后的默认值是调用 `cluster.setupMaster()` 后的值。

例如:

```
var cluster = require('cluster');
cluster.setupMaster({
  exec: 'worker.js',
  args: ['--use', 'https'],
  silent: true
});
cluster.fork(); // https worker
cluster.setupMaster({
  args: ['--use', 'http']
});
cluster.fork(); // http worker
```

仅能在主进程里调用。

cluster.fork([env])

- `env` {Object} 添加到子进程环境变量中的键值。
- return {Worker object}

派生一个新的工作进程。

仅能在主进程里调用。

cluster.disconnect([callback])

- `callback` {Function} 当所有工作进程都断开连接，并且关闭句柄后被调用。

`cluster.workers` 里的每个工作进程可调用 `.disconnect()` 关闭。

关闭所有的内部句柄连接，并且没有任何等待处理的事件时，允许主进程优雅的退出。

这个方法有一个可选参数，会在完成时被调用。

仅能在主进程里调用。

cluster.worker

- {Object}

对当前工作进程对象的引用。主进程中不可用。

```
var cluster = require('cluster');

if (cluster.isMaster) {
  console.log('I am master');
  cluster.fork();
  cluster.fork();
} else if (cluster.isWorker) {
  console.log('I am worker #' + cluster.worker.id);
}
```

cluster.workers

- {Object}

存储活跃工作对象的哈希表，主键是 `id`，能方便的遍历所有工作进程，仅在主进程可用。

当工作进程关闭连接并退出后，将会从 cluster.workers 里移除。这两个事件的次序无法确定，仅能保证从cluster.workers 移除会发生在 `'disconnect'` 或 `'exit'` 之后。

```
// Go through all workers
function eachWorker(callback) {
  for (var id in cluster.workers) {
    callback(cluster.workers[id]);
  }
}
eachWorker(function(worker) {
  worker.send('big announcement to all workers');
});
```

如果希望通过通讯通道引用工作进程，那么使用工作进程的 id 来查询最简单。

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

Class: Worker

一个 Worker 对象包含工作进程所有公开的信息和方法。在主进程里可用通过 `cluster.workers` 来获取，在工作进程里可以通过 `cluster.worker` 来获取。

worker.id

- {String}

每一个新的工作进程都有独立的唯一标示，它就是 `id`。

当工作进程可用时，`id` 就是 cluster.workers 里的主键。

worker.process

- {ChildProcess object}

所有工作进程都是通用 child_process.fork() 创建的，该函数返回的对象被储存在 process 中。

参见: Child Process module ()

注意，当 `process` 和 `.suicide` 不是 `true` 的时候，会触发 `'disconnect'` 事件，并使得工作进程调用 `process.exit(0)` 。它会保护意外的连接关闭。

worker.suicide

- {Boolean}

调用 `.kill()` 或 `.disconnect()` 后设置，在这之前是 `undefined` 。

`worker.suicide` 能让你区分出是自愿的还是意外退出，主进程可以根据这个值，来决定是否是重新派生成工作进程。

```
cluster.on('exit', function(worker, code, signal) {
  if (worker.suicide === true) {
    console.log('Oh, it was just suicide\' – no need to worry').
  }
});

// kill worker
worker.kill();
```

worker.send(message[, sendHandle])

- `message` {Object}
- `sendHandle` {Handle object}

这个函数和 child_process.fork() 提供的 send 方法相同。主进程里你必须使用这个函数给指定工作进程发消息。

在工作进程里，你也可以用 `process.send(message)` 。

这个例子会回应所有来自主进程的消息:

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

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

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

- signal {String}发送给工作进程的杀死信号的名称

这个函数会杀死工作进程。在主进程里，它会关闭 worker.process ，一旦关闭会发送杀死信号。在工作进程里，关闭通道，退出，返回代码 0 。

会导致 .suicide 被设置。

为了保持兼容性，这个方法的别名是 worker.destroy() 。

注意，在工作进程里有 process.kill() ，于此不同。

worker.disconnect()

在工作进程里，这个函数会关闭所有服务器，等待 'close' 事件，关闭 IPC 通道。

在主进程里，发给工作进程一个内部消息，用来调用 .disconnect()

会导致 .suicide 被设置。

注意，服务器关闭后，不再接受新的连接，但可以接受新的监听。已经存在的连接允许正常退出。当连接为空得时候，工作进程的 IPC 通道运行优雅的退出。

以上仅能适用于服务器的连接，客户端的连接由工作进程关闭。

注意，在工作进程里，存在 process.disconnect ，但并不是这个函数，它是 disconnect。

由于长连接可能会阻塞进程关闭连接，有一个较好的办法是发消息给应用，这样应用会想办法关闭它们。超时管理也是不错，如果超过一定时间后还没有触发 disconnect 事件，将会杀死进程。

```
if (cluster.isMaster) {
  var worker = cluster.fork();
```

```
  var timeout;

  worker.on('listening', function(address) {
    worker.send('shutdown');
    worker.disconnect();
    timeout = setTimeout(function() {
      worker.kill();
    }, 2000);
  });

  worker.on('disconnect', function() {
    clearTimeout(timeout);
  });

} else if (cluster.isWorker) {
  var net = require('net');
  var server = net.createServer(function(socket) {
    // connections never end
  });

  server.listen(8000);

  process.on('message', function(msg) {
    if(msg === 'shutdown') {
      // initiate graceful close of any connections to server
    }
  });
}
```

worker.isDead()

工作进程结束，返回 `true`， 否则返回 `false`。

worker.isConnected()

当工作进程通过 IPC 通道连接主进程时，返回 `true` ，否则 `false` 。工作进程创建后会连接到主进程。当 `disconnect` 事件触发后会关闭连接。

事件: 'message'

- `message` {Object}

该事件和 `child_process.fork()` 所提供的一样。在主进程中您应当使用该事件，而在工作进程中您也可以使用 `process.on('message')` 。

例如，有一个集群使用消息系统在主进程中统计请求的数量:

```
var cluster = require('cluster');
var http = require('http');

if (cluster.isMaster) {

  // Keep track of http requests
  var numReqs = 0;
  setInterval(function() {
    console.log("numReqs =", numReqs);
  }, 1000);

  // Count requestes
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd == 'notifyRequest') {
      numReqs += 1;
    }
  }

  // Start workers and listen for messages containing notifyRequest
  var numCPUs = require('os').cpus().length;
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  Object.keys(cluster.workers).forEach(function(id) {
    cluster.workers[id].on('message', messageHandler);
  });

} else {

  // Worker processes have a http server.
  http.Server(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");

    // notify master about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```

事件: 'online'

和 `cluster.on('online')` 事件类似, 仅能在特定工作进程里触发。

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

不会在工作进程里触发。

事件: 'listening'

- `address` {Object}

和 `cluster.on('listening')` 事件类似, 仅能在特定工作进程里触发。

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

不会在工作进程里触发。

事件: 'disconnect'

和 `cluster.on('disconnect')` 事件类似, 仅能在特定工作进程里触发。

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

事件: 'exit'

- `code` {Number} 正常退出时的退出代码.
- `signal` {String} 使得进程被终止的信号的名称（比如 `SIGHUP` ）。

和 `cluster.on('exit')` 事件类似, 仅能在特定工作进程里触发。

```
var worker = cluster.fork();
worker.on('exit', function(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: 'error'

和 child_process.fork() 事件类似。

工作进程里，你也可以用 process.on('error') 。

7

控制台

稳定性: 4 – 冻结

- {Object}

用于打印输出字符到 stdout 和 stderr。和多数浏览器提供的 console 对象函数一样，Node 也是输出到 stdout 和 stderr。

当输出目标是终端或文件的时候，console 函数是同步的（为了防止意外退出数据丢失），输出是管道的时候是异步的（防止阻塞时间太长）。

下面的例子里，stdout 是非阻塞的，而 stderr 是阻塞的。

```
$ node script.js 2> error.log | tee info.log
```

平常使用过程中，不用考虑阻塞或非阻塞问题，除非有大批量的数据。

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

输出到 stdout 并新起一行。和 `printf()` 类似，stdout 可以传入多个参数。例如：

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

如果第一个字符里没有找到格式化的元素， `util.inspect` 将会应用到各个参数，参见util.format() (https://nodejs.org/api/util.html#util_util_format_format)

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

参见 `console.log` 。

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

参见 `console.log` ，不同的是打印到 stderr。

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

参见 `console.error` 。

console.dir(obj[, options])

在 `obj` 使用 `util.inspect`，并打印结果到 stdout，而这个函数绕过 `inspect()` 。`options` 参数可能传入以下几种：

- `showHidden` – 如果是 `true`，将会展示对象的非枚举属性，默认是 `false` 。
- `depth` – `inspect` 对象递归的次数，对于复杂对象的扫描非常有用。默认是 `2` 。想要严格递归，传入 `null` 。
- `colors` – 如果是 `true`, 输出会格式化为 ANSI 颜色代码。默认是 `false` 。颜色可以定制，下面会介绍。

console.time(label)

标记一个时间点。

console.timeEnd(label)

计时器结束的时候，记录输出，例如：

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
  ;
}
console.timeEnd('100-elements');
// prints 100-elements: 262ms
```

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

输出当前位置的栈跟踪到 stderr `'Trace :'` 。

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

和 assert.ok() (https://nodejs.org/api/assert.html#assert_assert_value_message_assert_ok_value_message) 类似, 但是错误的输出格式为： `util.format(message...)` 。

8

加密

稳定性: 2 – 不稳定; 正在讨论未来版本的 API 改进，会尽量减少重大变化。详见后文。

使用 `require('crypto')` 来访问这个模块。

加密模块提供了 HTTP 或 HTTPS 连接过程中封装安全凭证的方法。

它也提供了 OpenSSL 的哈希，hmac, 加密（cipher）, 解密（decipher）, 签名（sign）和 验证（verify）方法的封装。

crypto.setEngine(engine[, flags])

为某些/所有 OpenSSL 函数加载并设置引擎（根据参数 flags 来设置）。

`engine` 可能是 id，或者是指向引擎共享库的路径。

`flags` 是可选参数，默认值是 `ENGINE_METHOD_ALL` ，它可以是以下一个或多个参数的组合（在 `constants` 里定义）:

- `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`

crypto.getCiphers()

返回支持的加密算法名数组。

例如：

```
var ciphers = crypto.getCiphers();
console.log(ciphers); // ['AES-128-CBC', 'AES-128-CBC-HMAC-SHA1', ...]
```

crypto.getHashes()

返回支持的哈希算法名数组。

例如：

```
var hashes = crypto.getHashes();
console.log(hashes); // ['sha', 'sha1', 'sha1WithRSAEncryption', ...]
```

crypto.createCredentials(details)

稳定性: 0 - 抛弃. 用 [tls.createSecureContext][] 替换.

根据参数 details，创建一个加密凭证对象。参数为字典，key 包括:

- `pfx` : 字符串或者buffer对象，表示经PFX或PKCS12编码产生的私钥、证书以及CA证书
- `key` : 进过 PEM 编码的私钥
- `passphrase` : 私钥或 pfx 的密码
- `cert` : PEM 编码的证书
- `ca` : 字符串或字符串数组，PEM 编码的可信任的 CA 证书。
- `crl` : 字符串或字符串数组，PEM 编码的 CRLs（证书吊销列表Certificate Revocation List）。
- `ciphers` : 字符串，使用或者排除的加密算法。参见http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT。

如果没有指定 'ca'，Node.js将会使用下面列表中的CAhttp://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt。

crypto.createHash(algorithm)

创建并返回一个哈希对象，使用指定的算法来生成哈希摘要。

参数 `algorithm` 取决于平台上 OpenSSL 版本所支持的算法。例如，`'sha1'`，`'md5'`，`'sha256'`，`'sha512'` 等等。在最近的版本中，`openssllist-message-digest-algorithms` 会显示所有算法。

例如： 这个程序会计算文件的 sha1 的和。

```
var filename = process.argv[2];
var crypto = require('crypto');
var fs = require('fs');

var shasum = crypto.createHash('sha1');

var s = fs.ReadStream(filename);
s.on('data', function(d) {
  shasum.update(d);
});

s.on('end', function() {
  var d = shasum.digest('hex');
  console.log(d + '  ' + filename);
});
```

类： Hash

用来生成数据的哈希值。

它是可读写的流 stream () 。写入的数据来用计算哈希值。当写入流结束后，使用 `read()` 方法来获取计算后的哈希值。也支持老的 `update` 和 `digest` 方法。

通过 `crypto.createHash` 返回。

hash.update(data[, input_encoding])

根据 `data` 来更新哈希内容，编码方式根据 `input_encoding` 来定，有 `'utf8'`，`'ascii'` 或 `'binary'`。如果没有传入值，默认编码方式是 `'binary'`。如果 `data` 是 `Buffer`， `input_encoding` 将会被忽略。

因为它是流式数据，所以可以使用不同的数据调用很多次。

hash.digest([encoding])

计算传入的数据的哈希摘要。`encoding` 可以是 `'hex'`，`'binary'` 或 `'base64'`，如果没有指定 `encoding`，将返回 buffer。
注意：调用 `digest()` 后不能再用 `hash` 对象。

crypto.createHmac(algorithm, key)

创建并返回一个 hmac 对象，用指定的算法和秘钥生成 hmac 图谱。

它是可读写的流 stream () 。写入的数据来用计算 hmac。当写入流结束后，使用 `read()` 方法来获取计算后的值。也支持老的 `update` 和 `digest` 方法。

参数 `algorithm` 取决于平台上 OpenSSL 版本所支持的算法，参见前面的 createHash。`key` 是 hmac 算法中用的 key。

类： Hmac

用来创建 hmac 加密图谱。

通过 `crypto.createHmac` 返回。

hmac.update(data)

根据 `data` 更新 hmac 对象。因为它是流式数据，所以可以使用新数据调用多次。

hmac.digest([encoding])

计算传入数据的 hmac 值。`encoding` 可以是 `'hex'`，`'binary'` 或 `'base64'`，如果没有指定 `encoding`，将返回 buffer。

注意：调用 `digest()` 后不能再用 `hmac` 对象。

crypto.createCipher(algorithm, password)

使用传入的算法和秘钥来生成并返回加密对象。

algorithm 取决于 OpenSSL，例如 'aes192' 等。最近发布的版本中， openssl list-cipher-algorithms 将会展示可用的加密算法。 password 用来派生 key 和 IV，它必须是一个 'binary' 编码的字符串或者一个buffer () 。

它是可读写的流 stream () 。写入的数据来用计算 hmac。当写入流结束后，使用 read() 方法来获取计算后的值。也支持老的 update 和 digest 方法。

注意，OpenSSL 函数EVP_BytesToKey (https://www.openssl.org/docs/crypto/EVP_BytesToKey.html) 摘要算法如果是一次迭代（one iteration），无需盐值（no salt）的 MD5 时， createCipher 为它派生秘钥。缺少盐值使得字典攻击，相同的密码总是生成相同的key，低迭代次数和非加密的哈希算法，使得密码测试非常迅速。

OpenSSL推荐使用 pbkdf2 来替换 EVP_BytesToKey，推荐使用 crypto.pbkdf2 (页 0) 来派生 key 和 iv ，推荐使用 createCipheriv() (页 0) 来创建加密流。

crypto.createCipheriv(algorithm, key, iv)

创建并返回一个加密对象，用指定的算法，key 和 iv。

algorithm 参数和 createCipher() 一致。 key 在算法中用到. iv 是一个initialization vector (http://en.wikipedia.org/wiki/Initialization_vector) .

key 和 iv 必须是 'binary' 的编码字符串或buffers () .

类： Cipher

加密数据的类。.

通过 crypto.createCipher 和 crypto.createCipheriv 返回。

它是可读写的流 stream () 。写入的数据来用计算 hmac。当写入流结束后，使用 read() 方法来获取计算后的值。也支持老的 update 和 digest 方法。

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

根据 data 来更新哈希内容，编码方式根据 input_encoding 来定，有 'utf8' , 'ascii' or 'binary' 。如果没有传入值，默认编码方式是 'binary' 。如果 data 是 Buffer ， input_encoding 将会被忽略。

`output_encoding` 指定了输出的加密数据的编码格式，它可用是 `'binary'`，`'base64'` 或 `'hex'`。如果没有提供编码，将返回 buffer 。

返回加密后的内容，因为它是流式数据，所以可以使用不同的数据调用很多次。

cipher.final([output_encoding])

返回加密后的内容，编码方式是由 `output_encoding` 指定，可以是 `'binary'`，`'base64'` 或 `'hex'`。如果没有传入值，将返回 buffer。

注意： `cipher` 对象不能在 `final()` 方法之后调用。

cipher.setAutoPadding(auto_padding=true)

你可以禁用输入数据自动填充到块大小的功能。如果 `auto_padding` 是false， 那么输入数据的长度必须是加密器块大小的整倍数，否则 `final` 会失败。这对非标准的填充很有用，例如使用0x0而不是PKCS的填充。这个函数必须在 `cipher.final` 之前调用。

cipher.getAuthTag()

加密认证模式（目前支持：GCM），这个方法返回经过计算的*认证标志* `Buffer`。必须使用 `final` 方法完全加密后调用。

cipher.setAAD(buffer)

加密认证模式（目前支持：GCM），这个方法设置附加认证数据（ AAD ）。

crypto.createDecipher(algorithm, password)

根据传入的算法和密钥，创建并返回一个解密对象。这是 createCipher() (页 0) 的镜像。

crypto.createDecipheriv(algorithm, key, iv)

根据传入的算法，密钥和 iv，创建并返回一个解密对象。这是 createCipheriv() (页 0) 的镜像。

类： Decipher

解密数据类。

通过 `crypto.createDecipher` 和 `crypto.createDecipheriv` 返回。

解密对象是可读写的流 streams () 。用写入的加密数据生成可读的纯文本数据。也支持老的 `update` 和 `digest` 方法。

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

使用参数 `data` 更新需要解密的内容，其编码方式是 `'binary'` , `'base64'` 或 `'hex'` 。如果没有指定编码方式，则把 `data` 当成 `buffer` 对象。

如果 `data` 是 `Buffer` ，则忽略 `input_encoding` 参数。

参数 `output_decoding` 指定返回文本的格式，是 `'binary'` , `'ascii'` 或 `'utf8'` 之一。如果没有提供编码格式，则返回 buffer。

decipher.final([output_encoding])

返回剩余的解密过的内容，参数 `output_encoding` 是 `'binary'` , `'ascii'` 或 `'utf8'` ，如果没有指定编码方式，返回 buffer。

注意， `decipher` 对象不能在 `final()` 方法之后使用。

decipher.setAutoPadding(auto_padding=true)

如果加密的数据是非标准块，可以禁止其自动填充，防止 `decipher.final` 检查并移除。仅在输入数据长度是加密块长度的整数倍的时才有效。你必须在 `decipher.update` 前调用。

decipher.setAuthTag(buffer)

对于加密认证模式（目前支持：GCM），必须用这个方法来传递接收到的*认证标志*。如果没有提供标志，或者密文被篡改，将会抛出 `final` 标志，认证失败，密文会被抛弃，

decipher.setAAD(buffer)

对于加密认证模式（目前支持：GCM），用这个方法设置附加认证数据（ AAD ）。

crypto.createSign(algorithm)

根据传入的算法创建并返回一个签名数据。 OpenSSL 的最近版本里， `openssl list-public-key-algorithms` 会列出所有算法，比如 `'RSA-SHA256'` 。

类： Sign

生成数字签名的类。

通过 `crypto.createSign` 返回。

签名对象是可读写的流 streams () 。可写数据用来生成签名。当所有的数据写完， `sign` 签名方法会返回签名。也支持老的 `update` 和 `digest` 方法。

sign.update(data)

用参数 `data` 来更新签名对象。因为是流式数据，它可以被多次调用。

sign.sign(private_key[, output_format])

根据传送给sign的数据来计算电子签名。

`private_key` 可以是一个对象或者字符串。如果是字符串，将会被当做没有密码的key。

`private_key` :

- `key` : 包含 PEM 编码的私钥
- `passphrase` : 私钥的密码

返回值 `output_format` 包含数字签名， 格式是 `'binary'` , `'hex'` 或 `'base64'` 之一。如果没有指定 `encoding` ，将返回 buffer。

注意： `sign` 对象不能在 `sign()` 方法之后调用。

crypto.createVerify(algorithm)

根据传入的算法，创建并返回验证对象。是签名对象（signing object）的镜像。

类： Verify

用来验证签名的类。

通过 `crypto.createVerify` 返回。

是可写流 streams () 。可写数据用来验证签名。一旦所有数据写完后，如签名正确 `verify` 方法会返回 `true` 。

也支持老的 `update` 方法。

verifier.update(data)

用参数 `data` 来更新验证对象。因为是流式数据，它可以被多次调用。

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

使用 `object` 和 `signature` 验证签名数据。参数 `object` 是包含了 PEM 编码对象的字符串，它可以是 RSA 公钥, DSA 公钥, 或 X.509 证书。`signature` 是之前计算出来的数字签名。`signature_format` 可以是 `'binary'`，`'hex'` 或 `'base64'` 之一，如果没有指定编码方式 ，则默认是buffer 对象。

根据数据和公钥验证签名有效性，来返回 true 或 false。

注意： `verifier` 对象不能在 `verify()` 方法之后调用。

crypto.createDiffieHellman(prime_length[, generator])

创建一个 Diffie-Hellman 密钥交换(Diffie-Hellman key exchange)对象，并根据给定的位长度生成一个质数。如果没有指定参数 `generator` ，默认为 `2` 。

crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding])

使用传入的 `prime` 和 `generator` 创建 Diffie-Hellman 秘钥交互对象。

`generator` 可以是数字，字符串或Buffer。

如果没有指定 `generator` ，使用 `2` .

`prime_encoding` 和 `generator_encoding` 可以是 `'binary'` , `'hex'` , 或 `'base64'` 。

如果没有指定 `prime_encoding` ， 则 Buffer 为 `prime` 。

如果没有指定 `generator_encoding` ，则 Buffer 为 `generator` 。

类： DiffieHellman

创建 Diffie-Hellman 秘钥交换的类。

通过 `crypto.createDiffieHellman` 返回。

diffieHellman.verifyError

在初始化的时候，如果有警告或错误，将会反应到这。它是以下值（定义在 `constants` 模块）：

- `DH_CHECK_P_NOT_SAFE_PRIME`
- `DH_CHECK_P_NOT_PRIME`
- `DH_UNABLE_TO_CHECK_GENERATOR`
- `DH_NOT_SUITABLE_GENERATOR`

diffieHellman.generateKeys([encoding])

生成秘钥和公钥，并返回指定格式的公钥。这个值必须传给其他部分。编码方式： `'binary'` , `'hex'` , 或 `'base64'` 。如果没有指定编码方式，将返回 buffer。

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

使用 other_public_key 作为第三方公钥来计算并返回共享秘密（shared secret）。秘钥用 input_encoding 编码。编码方式为： 'binary' , 'hex' , 或 'base64' 。如果没有指定编码方式，默认为 buffer。

如果没有指定返回编码方式，将返回 buffer。

diffieHellman.getPrime([encoding])

用参数 encoding 指明的编码方式返回 Diffie-Hellman 质数，编码方式为: 'binary' , 'hex' , 或 'base64' 。如果没有指定编码方式，将返回 buffer。

diffieHellman.getGenerator([encoding])

用参数 encoding 指明的编码方式返回 Diffie-Hellman 生成器，编码方式为: 'binary' , 'hex' , 或 'base64' . 如果没有指定编码方式，将返回 buffer。

diffieHellman.getPublicKey([encoding])

用参数 encoding 指明的编码方式返回 Diffie-Hellman 公钥，编码方式为: 'binary' , 'hex' , 或 'base64' . 如果没有指定编码方式，将返回 buffer。

diffieHellman.getPrivateKey([encoding])

用参数 encoding 指明的编码方式返回 Diffie-Hellman 私钥，编码方式为: 'binary' , 'hex' , 或 'base64' . 如果没有指定编码方式，将返回 buffer。

diffieHellman.setPublicKey(public_key[, encoding])

设置 Diffie-Hellman 的公钥，编码方式为: 'binary' , 'hex' , 或 'base64' ，如果没有指定编码方式，默认为 buffer。

diffieHellman.setPrivateKey(private_key[, encoding])

设置 Diffie-Hellman 的私钥，编码方式为: 'binary' , 'hex' , 或 'base64' ，如果没有指定编码方式 ，默认为 buffer。

crypto.getDiffieHellman(group_name)

创建一个预定义的 Diffie-Hellman 秘钥交换对象。支持的组： 'modp1' , 'modp2' , 'modp5' (定义于RFC 2412 (http://www.rfc-editor.org/rfc/rfc2412.txt)) and 'modp14' , 'modp15' , 'modp16' , 'modp17' , 'modp18' (定义于RFC 3526 (http://www.rfc-editor.org/rfc/rfc3526.txt)). 返回对象模仿了上述创建的crypto.createDiffieHellman() (页 0)对象，但是不允许修改秘钥交换 (例如，diffieHellman.setPublicKey() (页 0)) 。使用这套流程的好处是，双方不需要生成或交换组组余数，节省了计算和通讯时间。

例如 (获取一个共享秘密):

```
var crypto = require('crypto');
var alice = crypto.getDiffieHellman('modp5');
var bob = crypto.getDiffieHellman('modp5');

alice.generateKeys();
bob.generateKeys();

var alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
var bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

crypto.createECDH(curve_name)

使用传入的参数 curve_name ,创建一个 Elliptic Curve (EC) Diffie-Hellman 秘钥交换对象。

类： ECDH

这个类用来创建 EC Diffie-Hellman 秘钥交换。

通过 crypto.createECDH 返回。

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

生成 EC Diffie-Hellman 的秘钥和公钥，并返回指定格式和编码的公钥，它会传递给第三方。

参数 `format` 是 `'compressed'`, `'uncompressed'`, 或 `'hybrid'`. 如果没有指定，将返回 `'uncompressed'` 格式.

参数 `encoding` 是 `'binary'`, `'hex'`, 或 `'base64'`. 如果没有指定编码方式，将返回 buffer。

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

以 `other_public_key` 作为第三方公钥计算共享秘密，并返回。秘钥会以 `input_encoding` 来解读。编码是： `'binary'`, `'hex'`, 或 `'base64'` 。如果没有指定编码方式，默认为 buffer。

如果没有指定编码方式，将返回 buffer。

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

用参数 encoding 指明的编码方式返回 EC Diffie-Hellman 公钥，编码方式为: `'compressed'`, `'uncompressed'`, 或 `'hybrid'`. 如果没有指定编码方式，将返回 `'uncompressed'` 。

编码是： `'binary'`, `'hex'`, 或 `'base64'` 。如果没有指定编码方式，默认为 buffer。

ECDH.getPrivateKey([encoding])

用参数 encoding 指明的编码方式返回 EC Diffie-Hellman 私钥，编码是： `'binary'`, `'hex'`, 或 `'base64'` 。如果没有指定编码方式，默认为 buffer。

ECDH.setPublicKey(public_key[, encoding])

设置 EC Diffie-Hellman 的公钥，编码方式为: `'binary'`, `'hex'`, 或 `'base64'`，如果没有指定编码方式，默认为 buffer。

ECDH.setPrivateKey(private_key[, encoding])

设置 EC Diffie-Hellman 的私钥，编码方式为: `'binary'`, `'hex'`, 或 `'base64'`，如果没有指定编码方式，默认为 buffer。

例如 (包含一个共享秘密):

```
var crypto = require('crypto');
var alice = crypto.createECDH('secp256k1');
var bob = crypto.createECDH('secp256k1');

alice.generateKeys();
bob.generateKeys();

var alice_secret = alice.computeSecret(bob.getPublicKey(), null, 'hex');
var bob_secret = bob.computeSecret(alice.getPublicKey(), null, 'hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

crypto.pbkdf2(password, salt, iterations, keylen[, digest], callback)

异步 PBKDF2 提供了一个伪随机函数 HMAC-SHA1，根据给定密码的长度，salt 和 iterations 来得出一个密钥。回调函数得到两个参数 (err, derivedKey)。

例如:

```
crypto.pbkdf2('secret', 'salt', 4096, 512, 'sha256', function(err, key) {
  if (err)
    throw err;
  console.log(key.toString('hex'));  // 'c5e478d...1469e50'
});
```

在 crypto.getHashes() (页 0) 里有支持的摘要函数列表。

crypto.pbkdf2Sync(password, salt, iterations, keylen[, digest])

异步 PBKDF2 函数. 返回 derivedKey 或抛出错误。

crypto.randomBytes(size[, callback])

生成一个密码强度随机的数据:

```
// async
crypto.randomBytes(256, function(ex, buf) {
```

```
  if (ex) throw ex;
  console.log('Have %d bytes of random data: %s', buf.length, buf);
});

// sync
try {
  var buf = crypto.randomBytes(256);
  console.log('Have %d bytes of random data: %s', buf.length, buf);
} catch (ex) {
  // handle error
  // most likely, entropy sources are drained
}
```

注意：如果没有足够积累的熵来生成随机强度的密码，将会抛出错误，或调用回调函数返回错误。换句话说，没有回调函数的 `crypto.randomBytes` 不会阻塞，即使耗尽所有的熵。

crypto.pseudoRandomBytes(size[, callback])

生成非密码学强度的伪随机数据。如果数据足够长会返回一个唯一数据，但是这个数可能是可以预期的。因此，当不可预期很重要的时候，不要用这个函数。例如，在生成加密的秘钥时。

用法和 `crypto.randomBytes` 相同。

类： Certificate

这个类和签过名的公钥打交道。最重要的场景是处理 `<keygen>` 元素，http://www.openssl.org/docs/apps/spkac.html。

通过 `crypto.Certificate` 返回.

Certificate.verifySpkac(spkac)

根据 SPKAC 返回 true 或 false。

Certificate.exportChallenge(spkac)

根据提供的SPKAC，返回加密的公钥。

Certificate.exportPublicKey(spkac)

输出和 SPKAC 关联的编码 challenge。

crypto.publicEncrypt(public_key, buffer)

使用 `public_key` 加密 `buffer` 。目前仅支持 RSA。

`public_key` 可以是对象或字符串。如果 `public_key` 是一个字符串，将会当做没有密码的key，并会用`RSA_PKCS1_OAEP_PADDING` 。

`public_key` :

- `key` : 包含有 PEM 编码的私钥。
- `padding` : 填充值，如下
  - `constants.RSA_NO_PADDING`
  - `constants.RSA_PKCS1_PADDING`
  - `constants.RSA_PKCS1_OAEP_PADDING`

注意: 所有的填充值 定义于 `constants` 模块.

crypto.privateDecrypt(private_key, buffer)

使用 `private_key` 来解密 `buffer` .

`private_key` :

- `key` : 包含有 PEM 编码的私钥
- `passphrase` : 私钥的密码
- `padding` : 填充值，如下:
  - `constants.RSA_NO_PADDING`
  - `constants.RSA_PKCS1_PADDING`
  - `constants.RSA_PKCS1_OAEP_PADDING`

注意: 所有的填充值 定义于 `constants` 模块.

crypto.DEFAULT_ENCODING

函数所用的编码方式可以是字符串或 buffer ，默认值是 'buffer'。这是为了加密模块兼容默认 'binary' 为编码方式的遗留程序。

注意，新程序希望用 buffer 对象，所以这是暂时手段。

Recent API Changes

在统一的流 API 概念出现前，在引入 Buffer 对象来处理二进制数据之前，Crypto 模块就已经添加到 Node。

因此，流相关的类里没有其他的 Node 类里的典型方法，并且很多方法接收并返回二级制编码的字符串，而不是 Buffers。在最近的版本中，这些函数改成默认使用 Buffers。

对于一些场景来说这是重大变化。

例如，如果你使用默认参数给签名类，将结果返回给认证类，中间没有验证数据，程序会正常工作。之前你会得到二进制编码的字符串，并传递给验证类，现在则是 Buffer。

如果你之前使用的字符串数据在 Buffers 对象不能正常工作（比如，连接数据，并存储在数据库里 ）。或者你传递了二进制字符串给加密函数，但是没有指定编码方式，现在就需要提供编码参数。如果想切换回原来的风格，将 `crypto.DEFAULT_ENCODING` 设置为 'binary'。注意，新的程序希望是 buffers,所以之前的方法只能作为临时的办法。

9

调试器

```
稳定性: 3 – 稳定
```

V8 提供了强大的调试工具，可以通过 TCP protocol (http://code.google.com/p/v8/wiki/DebuggerProtocol) 从外部访问。Node 内置这个调试工具客户端。要使用这个调试器，以 `debug` 参数启动 Node，出现提示：

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

Node 的调试器不支持所有的命令，但是简单的步进和检查还是可以的。在代码里嵌入 `debugger;` ，可以设置断点。

例如，`myscript.js` 代码如下：

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

如果启动 debugger，它会断在第四行：

```
% 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(function () {
  3   debugger;
debug> cont
< hello
break in /home/indutny/Code/git/indutny/myscript.js:3
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
  4   console.log("world");
  5 }, 1000);
debug> next
break in /home/indutny/Code/git/indutny/myscript.js:4
```

```
 2 setTimeout(function () {
 3   debugger;
 4   console.log("world");
 5 }, 1000);
 6 console.log("hello");
debug> repl
Press Ctrl + C to leave debug repl
> x
5
> 2+2
4
debug> next
< world
break in /home/indutny/Code/git/indutny/myscript.js:5
 3   debugger;
 4   console.log("world");
 5 }, 1000);
 6 console.log("hello");
 7
debug> quit
%
```

`repl` 命令能执行远程代码； `next` 能步进到下一行。此外可以输入 `help` 查看哪些命令可用。

监视器-Watchers

调试的时候可以查看表达式和变量。每个断点处，监视器都会显示上下文。

输入 `watch("my_expression")` 开始监视表达式， `watchers` 显示活跃的监视器。输入 `unwatch("my_expression")` 可以移除监视器。

命令参考-Commands reference

步进-Stepping

- `cont` , `c` - 继续执行
- `next` , `n` - Step next
- `step` , `s` - Step in
- `out` , `o` - Step out
- `pause` - 暂停 (类似开发工具的暂停按钮)

断点Breakpoints

- `setBreakpoint()`, `sb()` - 当前行设置断点
- `setBreakpoint(line)`, `sb(line)` - 在指定行设置断点
- `setBreakpoint('fn()')`, `sb(...)` - 在函数里的第一行设置断点
- `setBreakpoint('script.js', 1)`, `sb(...)` - 在 script.js 第一行设置断点。
- `clearBreakpoint`, `cb(...)` - 清除断点

也可以在尚未加载的文件里设置断点。

```
% ./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 = function() {
 23   return 'hello from module';
 24 };
 25
debug>
```

信息Info

- `backtrace`, `bt` - 打印当前执行框架的backtrace
- `list(5)` - 显示脚本代码的 5 行上下文（之前 5 行和之后 5 行）
- `watch(expr)` - 监视列表里添加表达式
- `unwatch(expr)` - 从监视列表里删除表达式
- `watchers` - 显示所有的监视器和它们的值（每个断点都会自动列出）

- `repl` – 在所调试的脚本的上下文中，打开调试器的 repl

执行控制Execution control

- `run` – 运行脚本 (开始调试的时候自动运行)
- `restart` – 重新运行脚本
- `kill` – 杀死脚本

杂项Various

- `scripts` – 列出所有已经加载的脚本
- `version` – 显示 v8 版本

高级应用Advanced Usage

V8 调试器可以用两种方法启用和访问，`--debug`命令启动调试，或向已经启动 Node 发送 `SIGUSR1` 。

一旦一个进程进入调试模式，它可以被 node 调试器连接。调试器可以通过`pid` 或 URI 来连接。

- `node debug -p <pid>` – 通过 `pid` 连接进程
- `node debug <URI>` – 通过 URI（比如localhost:5858）连接进程w

10

DNS

稳定性: 3 – 稳定

调用 `require('dns')` 可以访问这个模块。

这个模块包含的函数属于2个不同的分类：

1）使用系统底层的特性，完成名字解析，这个过程不需要网络通讯，这个分类仅有一个函数： `dns.lookup` 。**开发者在同一个系统里名字解析都是用 `dns.lookup` .**

下面的例子，解析 `www.google.com` .

```
var dns = require('dns');

dns.lookup('www.google.com', function onLookup(err, addresses, family) {
  console.log('addresses:', addresses);
});
```

2）连接到 DNS 服务器进行名字解析，**始终**使用网络来进行域名查询。这个分类包含除了 `dns.lookup` 外的所有函数。这些函数不会和 `dns.lookup` 使用同一套配置文件。如果你不想使用系统底层的特性来进行名字解析，而想进行 DNS 查询的话，可以用这个分类的函数。

下面的例子，解析了 `'www.google.com'` ，并反向解析返回的 IP 地址。

```
var dns = require('dns');

dns.resolve4('www.google.com', function (err, addresses) {
  if (err) throw err;

  console.log('addresses: ' + JSON.stringify(addresses));

  addresses.forEach(function (a) {
    dns.reverse(a, function (err, hostnames) {
      if (err) {
        throw err;
      }

      console.log('reverse for ' + a + ': ' + JSON.stringify(hostnames));
    });
  });
});
```

更多细节参考Implementation considerations section (页 0)。

dns.lookup(hostname[, options], callback)

将域名（比如 'google.com' ）解析为第一条找到的记录 A （IPV4）或 AAAA(IPV6)。参数 options 可以是一个对象或整数。如果没有提供 options，IP v4 和 v6 地址都可以。如果 options 是整数，则必须是 4 或 6 。

options 参数可能是包含 family 和 hints 两个属性的对象。这两个属性都是可选的。如果提供了 family，则必须是 4 或 6，否则，IP v4 和 v6 地址都可以。如果提供了 hints，可以是一个或者多个 getaddrinfo 标志，若不提供，没有标志会传给 getaddrinfo 。多个标志位可以通过或运算来整合。以下的例子展示如何使用 options 。

```
{
  family: 4,
  hints: dns.ADDRCONFIG | dns.V4MAPPED
}
```

参见 supported getaddrinfo flags (页 0) 查看更多的标志位。

回调函数包含参数 (err, address, family) 。 address 参数表示 IP v4 或 v6 地址。family 参数是4 或 6，表示 address 家族（不一定是之前传入 lookup 的值）。

出错时，参数 err 是 Error 对象， err.code 是错误代码。请记住， err.code 等于 'ENOENT' ，不仅可能是因为域名不存在，还有可能是是其他原因，比如没有可用文件描述符。

dns.lookup 不必和 DNS 协议有关系。它使用了操作系统的特性，能将名字和地址关联。

实现这些东西也许很简单，但是对于 Node.js 程序来说都重要，所以在使用前请花点时间阅读Implementation considerations section (页 0)。

第 10 章 dns.lookupService(address, port, callback)

使用 `getnameinfo` 解析传入的地址和端口为域名和服务。

这个回调函数的参数是 `(err, hostname, service)`。 `hostname` 和 `service` 都是字符串 (比如 `'localhost'` 和 `'http'`)。

出错时，参数 `err` 是 `Error` 对象， `err.code` 是错误代码。

dns.resolve(hostname[, rrtype], callback)

将一个域名 (如 'google.com') 解析为一个 rrtype 指定记录类型的数组。

有效的 rrtypes 值为:

- `'A'` (IPV4 地址, 默认)
- `'AAAA'` (IPV6 地址)
- `'MX'` (邮件交换记录)
- `'TXT'` (text 记录)
- `'SRV'` (SRV 记录)
- `'PTR'` (用来反向 IP 查找)
- `'NS'` (域名服务器 记录)
- `'CNAME'` (别名 记录)
- `'SOA'` (授权记录的初始值)

回调参数为 `(err, addresses)`. 其中 `addresses` 中每一项的类型都取决于记录类型, 详见下文对应的查找方法。

出错时，参数 `err` 是 `Error` 对象， `err.code` 是错误代码。

dns.resolve4(hostname, callback)

和 `dns.resolve()` 类似, 仅能查询 IPv4 (`A` 记录)。 `addresses` IPv4 地址数组 (比如， `['74.125.79.104', '74.125.79.105', '74.125.79.106']`)。

dns.resolve6(hostname, callback)

和 dns.resolve4() 类似， 仅能查询 IPv4(AAAA 查询 ）。

dns.resolveMx(hostname, callback)

和 dns.resolve() 类似, 仅能查询邮件交换(MX 记录)。

addresses 是 MX 记录数组, 每一个包含优先级和交换属性(比如， [{'priority': 10, 'exchange': 'mx.example.com'},...] ）。

dns.resolveTxt(hostname, callback)

和 dns.resolve() 类似, 仅能进行文本查询 (TXT 记录 ）。 addresses 是 2-d 文本记录数组。(比如， [['v=spf1 ip4:0.0.0.0 ', '~all']] ）。 每个子数组包含一条记录的 TXT 块。根据使用情况可以连接在一起，也可单独使用。

dns.resolveSrv(hostname, callback)

和 dns.resolve() 类似, 仅能进行服务记录查询 (SRV 记录 ）。 addresses 是 hostname 可用的 SRV 记录数组。 SRV 记录属性有优先级 (priority ），权重 (weight), 端口 (port), 和名字 (name ）(比如， [{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...] ）。

dns.resolveSoa(hostname, callback)

和 dns.resolve() 类似, 仅能查询权威记录(SOA 记录 ）。

addresses 是包含以下结构的对象:

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
  serial: 2013101809,
  refresh: 10000,
  retry: 2400,
  expire: 604800,
```

```
  minttl: 3600
}
```

dns.resolveNs(hostname, callback)

和 `dns.resolve()` 类似, 仅能进行域名服务器记录查询(`NS` 记录 ）。 `addresses` 是域名服务器记录数组（ `hostname` 可以使用 ）（比如, `['ns1.example.com', 'ns2.example.com']` ）。

dns.resolveCname(hostname, callback)

和 `dns.resolve()` 类似, 仅能进行别名记录查询 (`CNAME` 记录)。 `addresses` 是对 `hostname` 可用的别名记录数组 (比如，, `['bar.example.com']` ）。

dns.reverse(ip, callback)

反向解析 IP 地址，返回指向该 IP 地址的域名数组。

回调函数参数 `（err, hostnames）`。

出错时，参数 `err` 是 `Error` 对象， `err.code` 是错误代码。

dns.getServers()

返回一个用于当前解析的 IP 地址的数组的字符串。

dns.setServers(servers)

指定一组 IP 地址作为解析服务器。

如果你给地址指定了端口，端口会被忽略，因为底层库不支持。

传入无效参数，会抛出以下错误:

Error codes

每个 DNS 查询都可能返回以下错误:

- `dns.NODATA`：DNS 服务器返回无数据应答。
- `dns.FORMERR`：DNS 服务器声称查询格式错误。
- `dns.SERVFAIL`：DNS 服务器返回一般失败。
- `dns.NOTFOUND`：没有找到域名。
- `dns.NOTIMP`：DNS 服务器未实现请求的操作。
- `dns.REFUSED`：DNS 服务器拒绝查询。
- `dns.BADQUERY`：DNS 查询格式错误。
- `dns.BADNAME`：域名格式错误。
- `dns.BADFAMILY`：地址协议不支持。
- `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`：文件末端。
- `dns.FILE`：读文件错误。
- `dns.NOMEM`：内存溢出。
- `dns.DESTRUCTION`：通道被摧毁。
- `dns.BADSTR`：字符串格式错误。
- `dns.BADFLAGS`：非法标识符。
- `dns.NONAME`：所给主机不是数字。
- `dns.BADHINTS`：非法HINTS标识符。
- `dns.NOTINITIALIZED`：c c-ares 库尚未初始化。
- `dns.LOADIPHLPAPI`：加载 iphlpapi.dll 出错。
- `dns.ADDRGETNETWORKPARAMS`：无法找到 GetNetworkParams 函数。
- `dns.CANCELLED`：取消 DNS 查询。

支持的 getaddrinfo 标志

以下内容可作为 hints 标志传给 `dns.lookup`

- `dns.ADDRCONFIG`：返回当前系统支持的地址类型。例如，如果当前系统至少配置了一个 IPv4 地址，则返回 IPv4地址。
- `dns.V4MAPPED`：如果指定了 IPv6 家族， 但是没有找到 IPv6 地址，将返回 IPv4 映射的 IPv6地址。

Implementation considerations

虽然 `dns.lookup` 和 `dns.resolve*/dns.reverse` 函数都能实现网络名和网络地址的关联，但是他们的行为不太一样。这些不同点虽然很巧妙，但是会对 Node.js 程序产生显著的影响。

dns.lookup

`dns.lookup` 和绝大多数程序一样使用了相同的系统特性。例如， `dns.lookup` 和 `ping` 命令用相同的方法解析了一个指定的名字。多数类似 POSIX 的系统， `dns.lookup` 函数可以通过改变 `nsswitch.conf(5)` 和/或 `resolv.conf(5)` 的设置调整。如果改变这些文件将会影响系统里的其他应用。

虽然，JavaScript 调用是异步的，它的实现是同步的调用 libuv 线程池里的 `getaddrinfo(3)` 。因为 libuv 线程池固定大小，所以如果调用 `getaddrinfo(3)` 的时间太长，会使的池里的其他操作（比如文件操作）性能降低。为了降低这个风险，可以通过增加 'UV_THREADPOOL_SIZE' 的值，让它超过4，来调整libuv线程池大小，更多信息参见[the official libuv documentation](http://docs.libuv.org/en/latest/threadpool.html）。

dns.resolve, functions starting with dns.resolve and dns.reverse

这些函数的实现和 `dns.lookup` 不大相同。他们不会用到 `getaddrinfo(3)` ，而是始终进行网络查询。这些操作都是异步的，和libuv线程池无关。

因此，这些操作对于其他线程不会产生负面影响，这和 `dns.lookup` 不同。

它们不会用到 `dns.lookup` 的配置文件（例如 `/etc/hosts` _）。

11

域

稳定性: 2 – 不稳定

域提供了一种方法，它能把多个不同的 IO 操作看成一个单独组。如果任何一个注册到域的事件或者回调触发 error 事件，或者抛出一个异常，域就会接收到通知，而不是在 process.on('uncaughtException') 处理函数一样丢失错误的上下文，也不会使程序立即退出。

警告：不要忽视错误!

域错误处理程序并不是一个错误发生时关闭你的进程的替代品。

基于 JavaScript 中抛出异常的工作原理，基本上不可能在不泄露引用，或者不造成一些其他未定义的状态下，完全重现现场。

响应抛出错误最安全的方法就是关闭进程。一个正常的服务器会可能有很多活跃的连接，因为某个错误就关闭所有连接显然是不合理的。

比较好的方法是给触发错误的请求发送回应，让其他连接正常工作时，停止监听触发错误的人的新请求。

按这种方法， 域 和集群（cluster）模块可以协同工作，当某个进程遇到错误时，主进程可以复制一个新的进程。对于 Node 程序，终端代理或者注册的服务，可以留意错误并做出反应。

举例来说，下面的代码就不是好办法:

```
javascript
// XXX WARNING!  BAD IDEA!

var d = require('domain').create();
d.on('error', function(er) {
  // The error won't crash the process, but what it does is worse!
  // Though we've prevented abrupt process restarting, we are leaking
  // resources like crazy if this ever happens.
  // This is no better than process.on('uncaughtException')!
  console.log('error, but oh well', er.message);
});
d.run(function() {
  require('http').createServer(function(req, res) {
    handleRequest(req, res);
  }).listen(PORT);
});
```

通过使用域的上下文，并将程序切为多个工作进程，我们能够更合理的响应，处理错误更安全。

```
javascript
// 好一些的做法!

var cluster = require('cluster');
var PORT = +process.env.PORT || 1337;

if (cluster.isMaster) {
  // In real life, you'd probably use more than just 2 workers,
  // and perhaps not put the master and worker in the same file.
  //
  // You can also of course get a bit fancier about logging, and
  // implement whatever custom logic you need to prevent DoS
  // attacks and other bad behavior.
  //
  // See the options in the cluster documentation.
  //
  // The important thing is that the master does very little,
  // increasing our resilience to unexpected errors.

  cluster.fork();
  cluster.fork();

  cluster.on('disconnect', function(worker) {
    console.error('disconnect!');
    cluster.fork();
  });

} else {
  // the worker
  //
  // This is where we put our bugs!

  var domain = require('domain');

  // See the cluster documentation for more details about using
  // worker processes to serve requests.  How it works, caveats, etc.

  var server = require('http').createServer(function(req, res) {
    var d = domain.create();
    d.on('error', function(er) {
      console.error('error', er.stack);

      // Note: we're in dangerous territory!
      // By definition, something unexpected occurred,
      // which we probably didn't want.
```

```
    // Anything can happen now!  Be very careful!

    try {
      // make sure we close down within 30 seconds
      var killtimer = setTimeout(function() {
        process.exit(1);
      }, 30000);
      // But don't keep the process open just for that!
      killtimer.unref();

      // stop taking new requests.
      server.close();

      // Let the master know we're dead.  This will trigger a
      // 'disconnect' in the cluster master, and then it will fork
      // a new worker.
      cluster.worker.disconnect();

      // try to send an error to the request that triggered the problem
      res.statusCode = 500;
      res.setHeader('content-type', 'text/plain');
      res.end('Oops, there was a problem!\n');
    } catch (er2) {
      // oh well, not much we can do at this point.
      console.error('Error sending 500!', er2.stack);
    }
  });

  // Because req and res were created before this domain existed,
  // we need to explicitly add them.
  // See the explanation of implicit vs explicit binding below.
  d.add(req);
  d.add(res);

  // Now run the handler function in the domain.
  d.run(function() {
    handleRequest(req, res);
  });
});
server.listen(PORT);
}

// This part isn't important.  Just an example routing thing.
// You'd put your fancy application logic here.
function handleRequest(req, res) {
```

```
 switch(req.url) {
   case '/error':
     // We do some async stuff, and then...
     setTimeout(function() {
       // Whoops!
       flerb.bark();
     });
     break;
   default:
     res.end('ok');
 }
}
```

错误对象的附加内容

任何时候一个错误被路由传到一个域的时，会添加几个字段。

- `error.domain` 第一个处理错误的域
- `error.domainEmitter` 用这个错误对象触发 'error' 事件的事件分发器
- `error.domainBound` 绑定到 domain 的回调函数，第一个参数是 error。
- `error.domainThrown` boolean 值，表明是抛出错误，分发，或者传递给绑定的回到函数。

隐式绑定

如果域正在使用中，所有新分发的对象（包括 流对象，请求，响应等）将会隐式的绑定到这个域。

另外，传递给底层事件循环（比如 fs.open 或其他接收回调的方法）的回调函数将会自动的绑定到这个域。如果他们抛出异常，域会捕捉到错误信息。

为了避免过度使用内存，域对象不会象隐式的添加为有效域的子对象。如果这样做的话，很容易影响到请求和响应对象的垃圾回收。

如果你想将域对象作为子对象嵌入到父域里，就必须显式的添加它们。

隐式绑定路由抛出的错误和 `'error'` 事件，但是不会注册事件分发器到域，所以 `domain.dispose()` 不会关闭事件分发器。隐式绑定仅需注意抛出的错误和 `'error'` 事件。

显式绑定

有时候正在使用的域并不是某个事件分发器的域。或者说，事件分发器可能在某个域里创建，但是被绑定到另外一个域里。

例如，HTTP 服务器使用正一个域对象，但我们希望可以每一个请求使用一个不同的域。

这可以通过显式绑定来实现。

例如:

```
// create a top-level domain for the server
var serverDomain = domain.create();

serverDomain.run(function() {
  // server is created in the scope of serverDomain
  http.createServer(function(req, res) {
    // req and res are also created in the scope of serverDomain
    // however, we'd prefer to have a separate domain for each request.
    // create it first thing, and add req and res to it.
    var reqd = domain.create();
    reqd.add(req);
    reqd.add(res);
    reqd.on('error', function(er) {
      console.error('Error', er, req.url);
      try {
        res.writeHead(500);
        res.end('Error occurred, sorry.');
      } catch (er) {
        console.error('Error sending 500', er, req.url);
      }
    });
  }).listen(1337);
});
```

domain.create()

- return: {Domain}

返回一个新的域对象。

Class: Domain

这个类封装了将错误和没有捕捉到的异常到有效对象功能。

域是 EventEmitter (events.html#events_class_events_eventemitter) 的子类. 监听它的 `error` 事件来处理捕捉到的错误。

domain.run(fn)

- `fn` {Function}

在域的上下文运行提供的函数，隐式的绑定了所有的事件分发器，计时器和底层请求。

这是使用域的基本方法。

例如:

```
var d = domain.create();
d.on('error', function(er) {
  console.error('Caught error!', er);
});
d.run(function() {
  process.nextTick(function() {
    setTimeout(function() { // simulating some various async stuff
      fs.open('non-existent file', 'r', function(er, fd) {
        if (er) throw er;
        // proceed...
      });
    }, 100);
  });
});
```

这个例子里程序不会崩溃，而会触发 `d.on('error')` 。

domain.members

- {Array}

显式添加到域里的计时器和事件分发器数组。

domain.add(emitter)

- `emitter` {EventEmitter | Timer} 添加到域里的计时器和事件分发器

显式将一个分发器添加到域。如果分发器调用的事件处理函数抛出错误，或者分发器遇到 `error` 事件，将会导向域的 `error` 事件，和隐式绑定一样。

对于 `setInterval` 和 `setTimeout` 返回的计时器同样适用。如果这些回调函数抛出错误，将会被域的 'error' 处理器捕捉到。

如果计时器或分发器已经绑定到域，那它将会从上一个域移除，绑定到当前域。

domain.remove(emitter)

- `emitter` {EventEmitter | Timer} 要移除的分发器或计时器

与 domain.add(emitter) 函数恰恰相反，这个函数将分发器移除出域。

domain.bind(callback)

- `callback` {Function} 回调函数
- return: {Function}被绑定的函数

返回的函数是一个对于所提供的回调函数的包装函数。当调用这个返回的函数被时，所有被抛出的错误都会被导向到这个域的 `error` 事件。

Example

```
var d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.bind(function(er, data) {
    // if this throws, it will also be passed to the domain
    return cb(er, data ? JSON.parse(data) : null);
  }));
}

d.on('error', function(er) {
  // an error occurred somewhere.
  // if we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

domain.intercept(callback)

- `callback` {Function} 回调函数
- return: {Function} 被拦截的函数

和 `domain.bind(callback)` 类似。除了捕捉被抛出的错误外，它还会拦截 Error 对象作为参数传递到这个函数。

这种方式下，常见的 `if (er) return callback(er);` 模式，能被一个地方一个错误处理替换。

Example

```
var d = domain.create();

function readSomeFile(filename, cb) {
  fs.readFile(filename, 'utf8', d.intercept(function(data) {
    // note, the first argument is never passed to the
    // callback since it is assumed to be the 'Error' argument
    // and thus intercepted by the domain.

    // if this throws, it will also be passed to the domain
    // so the error-handling logic can be moved to the 'error'
    // event on the domain instead of being repeated throughout
    // the program.
    return cb(null, JSON.parse(data));
  }));
}

d.on('error', function(er) {
  // an error occurred somewhere.
  // if we throw it now, it will crash the program
  // with the normal line number and stack message.
});
```

domain.enter()

这个函数就像 `run` , `bind` , 和 `intercept` 的管道系统，它设置有效域。它设定了域的 `domain.active` and `process.domain` ，还隐式的将域推到域模块管理的域栈（关于域栈的细节详见 `domain.exit()` ）。enter函数的调用，分隔了异步调用链以及绑定到一个域的I/O操作的结束或中断。

调用 `enter` 仅改变活动的域，而不改变域本身。在一个单独的域里可以调用任意多次 `Enter` 和 `exit` 。

domain.exit()

`exit` 函数退出当前域，并从域的栈里移除。每当程序的执行流程要切换到不同的异步调用链的时候，要保证退出当前域。调用 exit 函数，分隔了异步调用链，和绑定到一个域的I/O操作的结束或中断。

如果有多个嵌套的域绑定到当前的上下文，`exit` 函数将会退出所有嵌套。

调用 `exit` 仅改变活跃域，不会改变自身域。在一个单独的域里可以调用任意多次 `Enter` 和 `exit` 。

如果在这个域名下 exit 已经被设置，exit 将不退出域返回。

domain.dispose()

稳定性: 0 – 抛弃。通过域里设置的错误事件来显示的消除失败的 IO 操作。

调用 `dispos` 后，通过 run，bind 或 intercept 绑定到域的回调函数不再使用这个域，并且分发 `dispose` 事件。

12

事件

文档: 4 - API 冻结

Node 里很多对象会分发事件： 每次有连接的时候 `net.Server` 会分发事件，当文件打开的时候 `fs.readStream` 会分发事件。所有能分发事件的对象都是 `events.EventEmitter` 的实例。通过 `require("events");` 能访问这个模块。

一般来说，事件名都遵照驼峰规则，但这不是强制规定，任何形式的字符串都可以做为事件名。

为了处理事件，通常将函数关联到对象上。这些函数也叫监听者(listeners)。在这个函数里， `this` 指向 监听者所关联的 `EventEmitter` 。

()

类: events.EventEmitter

通过 `require('events').EventEmitter` 获取 EventEmitter 类。

`EventEmitter` 实例遇到错误后，通常会触发一个错误事件。错误事件在 node 里是特殊例子。如果没有监听者，默认的操作是打印一个堆栈信息并退出程序。

当添加新的监听者时， EventEmitters 会触发 `'newListener'` 事件，当移除时会触发 `'removeListener'` 。

emitter.addListener(event, listener)

emitter.on(event, listener)

添加一个监听者到特定 `event` 的监听数组的尾部，触发器不会检查是否已经添加过这个监听者。 多次调用相同的 `event` 和 `listener` 将会导致 `listener` 添加多次。

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
```

返回 emitter。

emitter.once(event, listener)

给事件添加一个一次性的 listener，这个 listener 只会被触发一次，之后就会被移除。

```
server.once('connection', function (stream) {
  console.log('Ah, we have our first user!');
});
```

返回emitter。

emitter.removeListener(event, listener)

从一个某个事件的 listener 数组中移除一个 listener。**注意**，这个操作会改变 listener 数组内容的次序。

```
var callback = function(stream) {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener` 最多会移除数组里的一个 listener。如果多次添加同一个 listener 到数组，那就需要多次调用 `removeListener` 来移除每一个实例。

返回emitter。

emitter.removeAllListeners([event])

移除所有的 listener，或者某个事件的 listener。最好不要移除全部 listener，尤其是那些不是你传入的（比如 socket 或 文件流）。

返回emitter。

emitter.setMaxListeners(n)

默认情况下，给单个事件添加超过 10 个 listener ，事件分发器会打印警告。这样有利于检查内存泄露。不过不是所有的分发器都应该限制在 10 个，这个函数允许改变 listener 数量，无论是 0 还是更多。

返回emitter。

EventEmitter.defaultMaxListeners

`emitter.setMaxListeners(n)` 设置一个分发器的最大 listener 数，而这个函数会立即设置**所有** `EventEmitter` 的当前值和默认值。要小心使用。

请注意， `emitter.setMaxListeners(n)` 的优先级高于 `EventEmitter.defaultMaxListeners`.

emitter.listeners(event)

返回事件的 listener 数组。

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection'))); // [ [Function] ]
```

emitter.emit(event[, arg1][, arg2][, ...])

使用指定的参数顺序的执行每一个 listener.

如果事件有 listener,返回 `true`， 否则 `false`

类方法: EventEmitter.listenerCount(emitter, event)

返回指定事件的 listener 数量。

Event: 'newListener'

- `event` {String} 事件名
- `listener` {Function} 事件处理函数

添加 listener 的时候会触发这个事件。当这个事件触发的时候，listener 可能还没添加到 listener 数组。

Event: 'removeListener'

- `event` {String} 事件名
- `listener` {Function} 事件处理函数

删除 listener 的时候会触发这个事件。当这个事件触发的时候，listener 可能还还没从 listener 数组移除。

13

文件系统

```
稳定性: 3 – 稳定
```

文件系统模块是一个封装了标准的 POSIX 文件 I/O 操作的集合。通过 `require('fs')` 使用这个模块。所有的方法都有同步和异步两种模式。

异步方法最后一个参数都是回调函数，这个回调的参数取决于方法，不过第一个参数一般都是异常。如果操作成功，那么第一个参数就是 `null` 或 `undefined` 。

当使用一个同步操作的时候，任意的异常都立即抛出，可以用 try/catch 来处理异常，使得程序正常运行。

这是异步操作的例子:

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

fs.unlink('/tmp/hello', function (err) {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

这是同步操作的例子:

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

fs.unlinkSync('/tmp/hello');
console.log('successfully deleted /tmp/hello');
```

异步方法不能保证操作顺序，因此下面的例子很容易出错：

```
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', function (err, stats) {
  if (err) throw err;
  console.log('stats: ' + JSON.stringify(stats));
});
```

可能先执行了 `fs.stat` 方法。正确的方法：

```
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  fs.stat('/tmp/world', function (err, stats) {
    if (err) throw err;
    console.log('stats: ' + JSON.stringify(stats));
  });
});
```

在繁忙的进程里，强烈建议使用异步方法。同步方法会阻塞整个进程，直到方法完成。

可能会用到相对路径，路径是相对 `process.cwd()` 来说的。

大部分 fs 函数会忽略回调参数，如果忽略，将会用默认函数抛出异常。如果想得到原调用点的堆栈信息，需要设置环境变量 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.>
```

fs.rename(oldPath, newPath, callback)

异步函数 rename(2)。回调函数只有一个参数：可能出现的异常。

fs.renameSync(oldPath, newPath)

同步函数 rename(2)。返回 `undefined` 。

fs.ftruncate(fd, len, callback)

异步函数 ftruncate(2)。回调函数只有一个参数：可能出现的异常。

fs.ftruncateSync(fd, len)

同步函数 ftruncate(2)。返回 `undefined`。

fs.truncate(path, len, callback)

异步函数 truncate(2)。回调函数只有一个参数：可能出现的异常。文件描述符也可以作为第一个参数，如果这种情况，调用 `fs.ftruncate()`。

fs.truncateSync(path, len)

同步函数 truncate(2)。返回 `undefined`。

fs.chown(path, uid, gid, callback)

异步函数 chown(2)。回调函数只有一个参数：可能出现的异常。

fs.chownSync(path, uid, gid)

同步函数 chown(2)。返回 `undefined`。

fs.fchown(fd, uid, gid, callback)

异步函数 fchown(2)。回调函数只有一个参数：可能出现的异常。

fs.fchownSync(fd, uid, gid)

同步函数 fchown(2)。返回 `undefined`。

fs.lchown(path, uid, gid, callback)

异步函数 lchown(2)。回调函数只有一个参数：可能出现的异常。

fs.lchownSync(path, uid, gid)

同步函数 lchown(2)。返回 `undefined` 。

fs.chmod(path, mode, callback)

异步函数 chmod(2)。回调函数只有一个参数：可能出现的异常。

fs.chmodSync(path, mode)

同步函数 chmod(2)。返回 `undefined` 。

fs.fchmod(fd, mode, callback)

异步函数 fchmod(2)。回调函数只有一个参数：可能出现的异常。

fs.fchmodSync(fd, mode)

同步函数 fchmod(2)。返回 `undefined` 。

fs.lchmod(path, mode, callback)

异步函数 lchmod(2)。回调函数只有一个参数：可能出现的异常。

仅在 Mac OS X 可用。

fs.lchmodSync(path, mode)

同步函数 lchmod(2)。返回 `undefined` 。

fs.stat(path, callback)

异步函数 stat(2)。回调函数有两个参数：(err, stats)，其中 `stats` 是一个 `fs.Stats` 对象。详情请参考 fs.Stats。

fs.lstat(path, callback)

异步函数 lstat(2)。回调函数有两个参数：(err, stats)，其中 `stats` 是一个 `fs.Stats` 对象。`lstat()` 与 `stat()` 基本相同, 区别在于，如果 `path` 是链接，读取的是链接本身，而不是它所链接到的文件。

fs.fstat(fd, callback)

异步函数 fstat(2)。回调函数有两个参数：(err, stats)，其中 `stats` 是一个 `fs.Stats` 对象。

fs.statSync(path)

同步函数 stat(2)。返回 `fs.Stats` 实例。

fs.lstatSync(path)

同步函数 lstat(2)。返回 `fs.Stats` 实例。

fs.fstatSync(fd)

同步函数 fstat(2)。返回 `fs.Stats` 实例。

fs.link(srcpath, dstpath, callback)

异步函数 link(2)。回调函数只有一个参数：可能出现的异常。

fs.linkSync(srcpath, dstpath)

同步函数 link(2)。返回 `undefined`。

fs.symlink(srcpath, dstpath[, type], callback)

异步函数 symlink(2)。回调函数只有一个参数：可能出现的异常。

`type` 可能是 `'dir'`，`'file'`, 或 `'junction'` (默认 `'file'`)，仅在 Windows（不考虑其他系统）有效。注意， Windows junction 要求目的地址需要绝对的。当使用 `'junction'` 的时候， `destination` 参数将会自动转换为绝对路径。

fs.symlinkSync(srcpath, dstpath[, type])

同步函数 symlink(2)。返回 `undefined`。

fs.readlink(path, callback)

异步函数 readlink(2)。回调函数有2个参数 `(err, linkString)`.

fs.readlinkSync(path)

同步函数 readlink(2)。返回符号链接的字符串值。

fs.realpath(path[, cache], callback)

异步函数 realpath(2)。回调函数有2个参数 `(err,resolvedPath)`。可以使用 `process.cwd` 来解决相对路径问题。

例如:

```
var cache = {'/etc':'/private/etc'};
fs.realpath('/etc/passwd', cache, function (err, resolvedPath) {
  if (err) throw err;
```

```
  console.log(resolvedPath);
});
```

fs.realpathSync(path[, cache])

同步函数 realpath(2)。返回解析出的路径。

fs.unlink(path, callback)

异步函数 unlink(2)。回调函数只有一个参数：可能出现的异常.

fs.unlinkSync(path)

同步函数 unlink(2)。返回 `undefined` 。

fs.rmdir(path, callback)

异步函数 rmdir(2)。回调函数只有一个参数：可能出现的异常.

fs.rmdirSync(path)

同步函数 rmdir(2)。返回 `undefined` 。

fs.mkdir(path[, mode], callback)

异步函数 mkdir(2)。回调函数只有一个参数：可能出现的异常. `mode` 默认s to `0777` .

fs.mkdirSync(path[, mode])

同步函数 mkdir(2)。返回 `undefined` 。

fs.readdir(path, callback)

异步函数 readdir(3)。 读取文件夹的内容。回调有2个参数 `(err, files)` files 是文件夹里除了名字为， `'.'` 和 `'..'`之外的所有文件名。

fs.readdirSync(path)

同步函数 readdir(3)。 返回除了文件名为 `'.'` 和 `'..'` 之外的所有文件.

fs.close(fd, callback)

异步函数 close(2)。 回调函数只有一个参数：可能出现的异常.

fs.closeSync(fd)

同步函数 close(2)。 返回 `undefined` 。

fs.open(path, flags[, mode], callback)

异步函数 file open. 参见 open(2)。 `flags` 是:

- `'r'` – 以只读模式打开.如果文件不存在，抛出异常。
- `'r+'` –以读写模式打开.如果文件不存在，抛出异常。
- `'rs'` – 同步的，以只读模式打开. 指令绕过操作系统直接使用本地文件系统缓存。 这个功能主要用来打开 NFS 挂载的文件，因为它能让你跳过可能过时的本地缓存。如果对 I/O 性能很在乎，就不要使用这个标志位。

  这里不是调用 `fs.open()` 变成同步阻塞请求，如果你想要这样，可以调用 `fs.openSync()` 。
- `'rs+'` – 同步模式下以读写方式打开文件。注意事项参见 `'rs'` .
- `'w'` – 以只写模式打开。文件会被创建 (如果文件不存在) 或者覆盖 (如果存在)。
- `'wx'` – 和 `'w'` 类似，如果文件存储操作失败
- `'w+'` – 以可读写方式打开。文件会被创建 (如果文件不存在) 或者覆盖 (如果存在)

- `'wx+'` – 和 `'w+'` 类似，如果文件存储操作失败。
- `'a'` – 以附加的形式打开。如果文件不存在则创建一个。
- `'ax'` – 和 `'a'` 类似，如果文件存储操作失败。
- `'a+'` – 以只读和附加的形式打开文件.若文件不存在，则会建立该文件
- `'ax+'` – 和 `'a+'` 类似，如果文件存储操作失败.

如果文件存在，参数 `mode` 设置文件模式 (permission 和 sticky bits)。 默认是 `0666`, 可读写.

回调有2个参数 `(err, fd)`.

排除标记 `'x'` (对应 open(2)的 `O_EXCL` 标记) 保证 `path` 是新创建的。在 POSIX 系统里，即使文件不存在，也会被认定为文件存在。 排除标记不能确定在网络文件系统中是否有效。

Linux系统里，无法对以追加模式打开的文件进行指定位置写。系统核心忽略了位置参数，每次把数据写到文件的最后。

fs.openSync(path, flags[, mode])

`fs.open()` 的同步版本. 返回整数形式的文件描述符。.

fs.utimes(path, atime, mtime, callback)

改变指定路径文件的时间戳。

fs.utimesSync(path, atime, mtime)

`fs.utimes()` 的同步版本. 返回 `undefined` 。

fs.futimes(fd, atime, mtime, callback)

改变传入的文件描述符指向文件的时间戳。

fs.futimesSync(fd, atime, mtime)

`fs.futimes()` 的同步版本. 返回 `undefined` 。

fs.fsync(fd, callback)

异步函数 fsync(2)。 回调函数只有一个参数：可能出现的异常.

fs.fsyncSync(fd)

同步 fsync(2)。 返回 `undefined` 。

fs.write(fd, buffer, offset, length[, position], callback)

将 `buffer` 写到 `fd` 指定的文件里。

参数 `offset` 和 `length` 确定写哪个部分的缓存。

参数 `position` 是要写入的文件位置。如果 `typeof position !== 'number'` ，将会在当前位置写入。参见 pwrite(2)。

回调函数有三个参数 `(err, written, buffer)` ， `written` 指定 `buffer` 的多少字节用来写。

注意，如果 `fs.write` 的回调还没执行，就多次调用 `fs.write` ，这样很不安全。因此，推荐使用 `fs.createWriteStream` 。

Linux系统里，无法对以追加模式打开的文件进行指定位置写。系统核心忽略了位置参数，每次把数据写到文件的最后。

fs.write(fd, data[, position[, encoding]], callback)

将 `buffer` 写到 `fd` 指定的文件里。如果 `data` 不是 buffer,那么它就会被强制转换为字符串。

参数 `position` 是要写入的文件位置。如果 `typeof position !== 'number'` ，将会在当前位置写入。参见 pwrite(2)。

参数 `encoding` ： 字符串的编码方式.

回调函数有三个参数 `(err, written, buffer)` ， `written` 指定 `buffer` 的多少字节用来写。注意写入的字节（bytes）和字符（string characters）不同。参见Buffer.byteLength (页 0)。

和写入 `buffer` 不同，必须写入整个字符串，不能截取字符串。这是因为返回的字节的位移跟字符串的位移是不一样的。

注意，如果 `fs.write` 的回调还没执行，就多次调用 `fs.write` ，这样很不安全。因此，推荐使用 `fs.createWriteStream`

Linux系统里，无法对以追加模式打开的文件进行指定位置写。系统核心忽略了位置参数，每次把数据写到文件的最后。

fs.writeSync(fd, buffer, offset, length[, position])

fs.writeSync(fd, data[, position[, encoding]])

`fs.write()` 的同步版本. 返回要写的bytes数.

fs.read(fd, buffer, offset, length, position, callback)

读取 `fd` 指定文件的数据。

`buffer` 是缓冲区，数据将会写入到这里.

`offset` 写入的偏移量

`length` 需要读的文件长度

`position` 读取的文件起始位置，如果是 `position` 是 `null` ， 将会从当前位置读。

回调函数有3个参数，`(err, bytesRead, buffer)`.

fs.readSync(fd, buffer, offset, length, position)

`fs.read` 的同步版本. 返回 `bytesRead` 的数量.

fs.readFile(filename[, options], callback)

- `filename` {String}
- `options` {Object}
  - `encoding` {String | Null} 默认 = `null`
  - `flag` {String} 默认 = `'r'`
- `callback` {Function}

异步读取整个文件的内容。例如：

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

回调函数有2个参数 `(err, data)`, 参数 `data` 是文件的内容。 如果没有指定参数 `encoding`, 返回原生 buffer

fs.readFileSync(filename[, options])

`fs.readFile` 的同步版本. 返回整个文件的内容.

如果没有指定参数 `encoding`, 返回buffer。

fs.writeFile(filename, data[, options], callback)

- `filename` {String}
- `data` {String | Buffer}
- `options` {Object}
  - `encoding` {String | Null} 默认 = `'utf8'`
  - `mode` {Number} 默认 = `438` (aka `0666` in Octal)
  - `flag` {String} 默认 = `'w'`
- `callback` {Function}

异步写文件，如果文件已经存在则替换。 `data` 可以是缓存或者字符串。

如果参数 `data` 是 buffer，会忽略参数 `encoding` 。默认值是 `'utf8'` 。

列如:

```
fs.writeFile('message.txt', 'Hello Node', function (err) {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

fs.writeFileSync(filename, data[, options])

`fs.writeFile` 的同步版本. 返回 `undefined` 。

fs.appendFile(filename, data[, options], callback)

- `filename` {String}
- `data` {String | Buffer}
- `options` {Object}
  - `encoding` {String | Null} 默认 = `'utf8'`
  - `mode` {Number} 默认 = `438` (aka `0666` in Octal)
  - `flag` {String} 默认 = `'a'`
- `callback` {Function}

异步的给文件添加数据，如果文件不存在，就创建一个。 `data` 可以是缓存或者字符串。

例如:

```
fs.appendFile('message.txt', 'data to append', function (err) {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

fs.appendFileSync(filename, data[, options])

`fs.appendFile` 的同步版本. 返回 `undefined` 。

fs.watchFile(filename[, options], listener)

稳定性: 2 – 不稳定。 尽可能的用 fs.watch 来替换。

监视 filename 文件的变化。每当文件被访问的时候都会调用 listener 。

第二个参数可选。如果有，它必须包含两个 boolean 参数（ persistent 和 interval ）的对象。 persistent 指定文件被监视时进程是否继续运行。 interval 指定了查询文件的间隔，以毫秒为单位。缺省值为{ persistent: true, interval: 5007 }。

listener 有两个参数，第一个为文件现在的状态，第二个为文件的前一个状态：

```
fs.watchFile('message.text', function (curr, prev) {
  console.log('the current mtime is: ' + curr.mtime);
  console.log('the previous mtime was: ' + prev.mtime);
});
```

listener中的文件状态对象类型为 fs.Stat。

如果想修改文件时被通知，而不是访问的时候就通知，可以比较 curr.mtime 和 prev.mtime 。

fs.unwatchFile(filename[, listener])

稳定性: 2 – 不稳定. 尽可能的用 fs.watch 来替换。

停止监视 filename 文件的变化。如果指定了 listener ，那只会移除这个 listener 。否则，移除所有的 listener，并会停止监视 filename 。

调用 fs.unwatchFile() 停止监视一个没被监视的文件，不会触发错误，而会发生一个no-op。

fs.watch(filename[, options][, listener])

稳定性: 2 – 不稳定.

观察 filename 指定的文件或文件夹的改变。返回对象是 fs.FSWatcher (页 0)。

第二个参数可选。如果有，它必须是包含两个 boolean 参数（ persistent 和 recursive ）的对象。 persistent 指定文件被监视时进程是否继续运行。 recursive 表明是监视所有的子文件夹还是当前文件夹，这个参数只有监视对象是文件夹时才有效，而且仅在支持的系统里有效（参见下面注意事项）。

默认值 { persistent: true, recursive: false } .

回调函数有2个参数 (event, filename) 。 event 是 rename 或 change 。 filename 是触发事件的文件名。

注意事项

fs.watch API 不是 100% 的跨平台兼容，可能在某些情况下不可用。

recursive 参数仅在 OS X 上可用。仅 FSEvents 支持这个类型文件的监视，所以未来也不太可能有新的平台加入。

可用性

这些特性依赖于底层系统提供文件系统变动的通知。

- Linux 系统,使用 inotify .
- BSD 系统,使用 kqueue .
- OS X,文件使用 kqueue ，文件夹使用 FSEvents .
- SunOS 系统(包括 Solaris 和 SmartOS),使用 event ports .
- Windows 系统, 依赖与 ReadDirectoryChangesW .

如果底层系统函数不可用，那么 fs.watch 就无法工作。例如，监视网络文件系统(NFS, SMB, 等)经常不能用。你仍然可以用 fs.watchFile 查询，但是会比较慢，且不可靠。

文件名参数

回调函数中提供文件名参数，不是每个平台都能用（ Linux 和 Windows 就不行 ）。即使在可用的平台，也不能保证都能提供。所以不要假设回调函数中 filename 参数有效，要在代码里添加一些为空的逻辑判断。

```
fs.watch('somedir', function (event, filename) {
  console.log('event is: ' + event);
  if (filename) {
    console.log('filename provided: ' + filename);
  } else {
    console.log('filename not provided');
  }
});
```

fs.exists(path, callback)

判断文件是否存在，回调函数参数是 bool 值。例如：

```
fs.exists('/etc/passwd', function (exists) {
  util.debug(exists ? "it's there" : "no passwd!");
});
```

`fs.exists()` 是老版本的函数，因此在代码里不要用。

另外，打开文件前判断是否存在有漏洞，在 `fs.exists()` 和 `fs.open()` 调用中间，另外一个进程有可能已经移除了文件。最好用 `fs.open()` 来打开文件，根据回调函数来判断是否有错误。

`fs.exists()` 未来会被移除。

fs.existsSync(path)

`fs.exists()` 的同步版本. 如果文件存在返回 `true`, 否则返回 `false` 。

`fs.existsSync()` 未来会被移除。

fs.access(path[, mode], callback)

测试由参数 `path` 指向的文件的用户权限。可选参数 `mode` 为整数，它表示需要检查的权限。下面列出了所有值。`mode` 可以是单个值，或者可以通过或运算，掩码运算实现多个权限检查。

- `fs.F_OK` - 文件是对于进程可见，可以用来检查文件是否存在。参数 `mode` 的默认值。
- `fs.R_OK` - 文件对于进程是否可读。
- `fs.W_OK` - 文件对于进程是否可写。
- `fs.X_OK` - 文件对于进程是否可执行。（Windows系统不可用，执行效果等同 `fs.F_OK` ）

第三个参数是回调函数。如果检查失败，回调函数的参数就是响应的错误。下面的例子检查文件 `/etc/passwd` 是否能被当前的进程读写。

```
fs.access('/etc/passwd', fs.R_OK | fs.W_OK, function(err) {
  util.debug(err ? 'no access!' : 'can read/write');
});
```

fs.accessSync(path[, mode])

`fs.access` 的同步版本. 如果发生错误抛出异常，否则不做任何事情。

类: fs.Stats

`fs.stat()`，`fs.lstat()` 和 `fs.fstat()` 以及同步版本的返回对象。

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()` (only valid with `fs.lstat()`)
- `stats.isFIFO()`
- `stats.isSocket()`

对普通文件使用 `util.inspect(stats)`，返回的字符串和下面类似：

```
{ 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 }
```

`atime`，`mtime`，`birthtime`，和 `ctime` 都是 Date (https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Date) 的实例，需要使用合适的方法来比较这些值。通常使用 getTime() (https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Date/getTime) 来获取时间戳（毫秒，从 *1 January 1970 00:00:00 UTC* 开始算），这个整数基本能满足任何比较条件。也有一些其他方法来显示额外信息。更多参见MDN JavaScript Reference (https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Date)

Stat Time Values

状态对象（stat object）有以下语义：

- `atime` 访问时间 – 文件的最后访问时间. `mknod(2)`, `utimes(2)`, 和 `read(2)` 等系统调用可以改变.
- `mtime` 修改时间 – 文件的最后修改时间. `mknod(2)`, `utimes(2)`, 和 `write(2)` 等系统调用可以改变.
- `ctime` 改变时间 – 文件状态(inode)的最后修改时间. `chmod(2)`, `chown(2)`, `link(2)`, `mknod(2)`, `rename(2)`, `unlink(2)`, `utimes(2)`, `read(2)`, 和 `write(2)` 等系统调用可以改变.
- `birthtime` "Birth Time" – 文件创建时间，文件创建时生成。 在一些不提供文件 birthtime 的文件系统中, 这个字段会使用 ctime 或 1970-01-01T00:00Z (ie, unix epoch timestamp 0)来填充。 在 Darwin 和其他 FreeBSD 系统变体中, 也将 atime 显式地设置成比它现在的 birthtime 更早的一个时间值，这个过程使用了 utimes(2) 系统调用。

在 Node v0.12 版本之前, Windows 系统里 ctime 有 birthtime 值. 注意在v.0.12版本中, ctime 不再是"creation time", 而且在Unix系统中，他一直都不是。

fs.createReadStream(path[, options])

返回可读流对象 (见 `Readable Stream`)。

`options` 默认值如下:

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0666,
  autoClose: true
}
```

参数 `options` 提供 `start` 和 `end` 位置来读取文件的特定范围内容，而不是整个文件。`start` 和 `end` 都在文件范围里，并从 0 开始， `encoding` 是 `'utf8'`, `'ascii'`, 或 `'base64'`。

如果给了 `fd` 值， `ReadStream` 将会忽略 `path` 参数，而使用文件描述，这样不会触发任何 `open` 事件。

如果 `autoClose` 为 false，即使发生错误文件也不会关闭，需要你来负责关闭，避免文件描述符泄露。如果 `autoClose` 是 true（默认值），遇到 `error` 或 `end` ，文件描述符将会自动关闭。

例如，从100个字节的文件里，读取最少10个字节：

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

Class: fs.ReadStream

`ReadStream` 是 Readable Stream (页 0)。

Event: 'open'

- `fd` {Integer} ReadStream 所使用的文件描述符。

当创建文件的ReadStream时触发。

fs.createWriteStream(path[, options])

返回一个新的写对象 (参见 `Writable Stream`)。

`options` 是一个对象，默认值:

```
{ flags: 'w',
  encoding: null,
  fd: null,
  mode: 0666 }
```

options 也可以包含一个 start 选项，在指定文件中写入数据开始位置。 修改而不替换文件需要 flags 的模式指定为 r+ 而不是默值的 w.

和之前的 `ReadStream` 类似，如果 `fd` 不为空， `WriteStream` 将会忽略 `path` 参数，转而使用文件描述，这样不会触发任何 `open` 事件。

类: fs.WriteStream

`WriteStream` 是 Writable Stream (页 0)。

Event: 'open'

- `fd` {Integer} WriteStream 所用的文件描述符

打开 WriteStream file 时触发。

file.bytesWritten

目前写入的字节数，不含等待写入的数据。

Class: fs.FSWatcher

`fs.watch()` 返回的对象就是这个类.

watcher.close()

停止观察 `fs.FSWatcher` 对象中的更改。

Event: 'change'

- `event` {String} fs 改变的类型
- `filename` {String} 改变的文件名 (if relevant/available)

当监听的文件或文件夹改变的时候触发，参见fs.watch (页 0)。

Event: 'error'

- `error` {Error object}

错误发生时触发。

14

全局对象

这些对象在所有模块里都可用。有些对象不是在全局作用域而是在模块作用域里，这些情况下面文档都会标注出来。

global

- {Object} 全局命名空间对象。

浏览器里，全局作用域就是顶级域。如果在全局域内定义变量 `var something` 将会是全局变量。 Node 里不同，顶级域并不是全局域；在模块里定义变量 `var something` 只是模块内可用。

process

- {Object}

进程对象。 参见 process object (process.html#process_process) 章节.

console

- {Object}

用来打印 stdout 和 stderr. 参见console (console.html) 章节.

Class: Buffer

- {Function}

用来处理二进制数据。 参见buffer 章节 (buffer.html) 。

require()

- {Function}

引入模块。 参见Modules (modules.html#modules_modules) 章节. `require` 实际上并非全局的，而是各个本地模块有效。

require.resolve()

使用内部 `require()` 机制来查找 module 位置，但是不加载模块，只是返回解析过的文件名。

require.cache

- {Object}

引入模块时会缓存到这个对象。通过删除该对象键值，下次调用 `require` 将会重载该模块。

require.extensions

稳定性: 0 – 抛弃

- {Object}

指导 `require` 如何处理特定的文件扩展名。

将 `.sjs` 文件当 `.js` 文件处理:

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

抛弃 以前这个列表用来加载按需编译的非 JavaScript 模块到 node。实际上，有更好的办法来解决这个问题，比如通过其他 node 程序来加载模块，或者提前编译成 JavaScript。

由于模块系统已经锁定，该功能可能永远不会去掉。改动它可能会产生 bug，所以最好不要动它。

__filename

- {String}

被执行的代码的文件名是相对路径。对于主程序来说，这和命令行里未必用同一个文件名。模块里的值是模块文件的路径。

列如，运行 `/Users/mjr` 里的 `node example.js`：

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

`__filename` 不是全局的，而是模块本地的。

__dirname

- {String}

执行的 script 代码所在的文件夹的名字。

列如，运行 `/Users/mjr` 里的 `node example.js`：

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

`__dirname` 不是全局的，而是模块本地的。

module

- {Object}

当前模块的引用。通过 `require()`， `module.exports` 定义了哪个模块输出可用。

`module` 不是全局的，而是模块本地的。

更多信息参见module system documentation (modules.html) 。

exports

`module.exports` 的引用。何时用 `exports` 和 `module.exports` 可参加module system documentation (modules.html) 。

`module` 不是全局的，而是模块本地的。

更多信息参见 module system documentation (modules.html) 。

更多信息参见module 章节 (modules.html) 。

setTimeout(cb, ms)

最少 `ms` 毫秒后调回调函数。实际的延迟依赖于外部因素，比如操作系统的粒度和负载。

timeout 值有效范围 1–2,147,483,647。如果超过范围，将会变为 1 毫秒。通常，定时器不应该超过 24.8 天。

返回一个代表定时器的句柄值。

clearTimeout(t)

停止一个之前通过 `setTimeout()` 创建的定时器。不会再被执行回调。

setInterval(cb, ms)

每隔 `ms` 毫秒调用回调函数 `cb` 。实际的间隔依赖于外部因素，比如操作系统的粒度和系统负载。通常会大于 `ms` 。

间隔值有效范围 1-2,147,483,647。如果超过范围，将会变为 1 毫秒。通常，定时器不应该超过 24.8 天。

返回一个代表该定时器的句柄值。

clearInterval(t)

停止一个之前通过 `setInterval()` 创建的定时器。不会再被执行回调。

timer 函数是全局变量。 参见timers (timers.html) 章节。

HTTP

稳定性: 3 – 稳定

使用 HTTP 服务器或客户端功能必须调用 `require('http')` 。

Node 里的 HTTP 接口支持协议里原本比较难用的特性。特别是很大的或块编码的消息。这些接口不会完全缓存整个请求或响应，这样用户可以在请求或响应中使用数据流。

HTTP消息头对象和下面的例子类似：

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
  'accept': '*/*' }
```

Keys 都是小写，值不能修改。

为了能支持尽可能多的 HTTP 应用程序，Node 提供的 HTTP API 接口都是底层的。仅能处理流和消息。它把消息解析成报文头和报文体，但是不解析实际的报文头和报文体内容。

定义报文头的时候，多个值间可用 `,` 分隔。除了 `set-cookie` 和 `cookie` 头，因为它们表示值的数组。诸如 `content-length` 的只有一个值的报文头，直接解析，并且只有单值可以表示成已经解析好的对象。

接收到的原始头信息以数组（ `[key, value, key2, value2, ...]` ）的形式保存在 `rawHeaders` 里。例如，之前提到的消息对象会有如下的 `rawHeaders` ：

```
[ 'ConTent-Length', '123456',
  'content-LENGTH', '123',
  'content-type', 'text/plain',
  'CONNECTION', 'keep-alive',
  'Host', 'mysite.com',
  'accepT', '*/*' ]
```

http.METHODS

- {Array}

解析器支持的 HTTP 方法列表。

http.STATUS_CODES

- {Object}

全部标准 HTTP 响应状态码的和描述的集合。例如， `http.STATUS_CODES[404] === 'Not Found'` 。

http.createServer([requestListener])

返回 http.Server (页 0) 的新实例。

`requestListener` 函数自动加到 `'request'` 事件里。

http.createClient([port][, host])

这个函数已经被抛弃，用 http.request() (页 164) 替换。创建一个新的 HTTP 客户端，连接到服务器的 `port` 和 `host` 。

类： http.Server

这是事件分发器 EventEmitter (events.html#events_class_events_eventemitter) ，有以下事件：

事件： 'request'

`function (request, response) { }`

每当有请求的时候触发。注意：每个连接可以有多个请求（在 keep-alive 连接中）。 `request` 是 http.IncomingMessage (页 0) 实例， `response` 是 http.ServerResponse (页 0) 的实例。

事件： 'connection'

`function (socket) { }`

当建立新的 TCP 流的时候。 `socket` 是一个 `net.Socket` 对象。通常用户不会访问这个事件。协议解析器绑定套接字时采用的方式使套接字不会出发 readable 事件。也能通过 `request.connection` 访问 `socket` 。

事件： 'close'

`function () { }`

服务器关闭的时候触发。

事件： 'checkContinue'

`function (request, response) { }`

当 http 收到 100-continue 的 http 请求时会触发。如果没有监听这个事件，服务器将会自动发送 100 Continue 的响应。

如果客户端需要继续发送请求主题，或者生成合适的 HTTP 响应（如，400 请求无效），可以通过调用 response.writeContinue() (页 0) 来处理。

注意：触发并处理这个事件的时候，不会再触发 `request` 事件。

事件： 'connect'

`function (request, socket, head) { }`

当客户端请求 http 连接时触发。如果没有监听这个事件，客户端请求连接的时候会被关闭。

- `request` 是 http 请求的参数，与 request 事件参数相同。
- `socket` 是服务器和客户端间的 socket。
- `head` 是 buffer 的实例。网络隧道的第一个包，可能为空。

这个事件触发后，请求的 socket 不会有 `data` 事件监听器，也就是说你需要绑定一个监听器到 `data` 上，来处理在发送到服务器上的 socket 数据。

事件： 'upgrade'

`function (request, socket, head) { }`

当客户端请求 http upgrage 时候会触发。如果没有监听这个事件，客户端请求一个连接的时候会被关闭。

- `request` 是 http 请求的参数，与 request 事件参数相同。
- `socket` 是服务器和客户端间的 socket。
- `head` 是 buffer 的实例。网络隧道的第一个包，可能为空。

这个事件触发后，请求的 socket 不会有 `data` 事件监听器，也就是说你需要绑定一个监听器到 `data` 上，来处理在发送到服务器上的 socket 数据。

事件： 'clientError'

`function (exception, socket) { }`

如果一个客户端连接触发了一个 'error' 事件, 它就会转发到这里.

`socket` 是导致错误的 `net.Socket` 对象。

()

server.listen(port[, hostname][, backlog][, callback])

在指定的的端口和主机名上开始接收连接。 如果忽略主机名，服务器将会接收指向任意 IPv4 的地址(`INADDR_ANY`)。

监听一个 unix socket，需要提供一个文件名而不是主机名和端口。

积压量 backlog 为等待连接队列的最大长度。实际的长度由你的操作系统的 sysctl 设置决定 (比如 linux 上的 `tcp_max_syn_backlog` and `somaxconn`) 。默认参数值为 511 (不是 512)

这是异步函数。最后一个参数 `callback` 会作为事件监听器添加到 `listening` 事件。参见net.Server.listen(port) (net.html#net_server_listen_port_host_backlog_callback) 。

server.listen(path[, callback])

启动一个 UNIX socket 服务器所给路径 `path`

这是异步函数。最后一个参数 `callback` 会作为事件监听器添加到 `listening` 事件。参见net.Server.listen(port) (net.html#net_server_listen_port_host_backlog_callback) 。

server.listen(handle[, callback])

- `handle` {Object}
- `callback` {Function}

  `handle` 对象可以是 server 或 socket (任意以下划线 `_handle` 开头的成员) ，或者 `{fd: <n>}` 对象。

这会导致 server 用参数 `handle` 接收连接，前提条件是文件描述符或句柄已经连接到端口或域 socket。

Windows 不能监听文件句柄。

这是异步函数。最后一个参数 `callback` 会作为事件监听器添加到 `listening` 事件。参见net.Server.listen(port) (net.html#net_server_listen_port_host_backlog_callback) 。

()

server.close([callback])

禁止 server 接收连接。参见 net.Server.close() (net.html#net_server_close_callback) .

server.maxHeadersCount

最大请求头的数量限制，默认 1000。如果设置为 0，则不做任何限制。

()

server.setTimeout(msecs, callback)

- `msecs` {Number}
- `callback` {Function}

为 socket 设置超时时间，单位为毫秒，如果发生超时，在 server 对象上触发 `'timeout'` 事件，参数为 socket 。

如果在 Server 对象上有一个 `'timeout'` 事件监听器，超时的时候，将会调用它，参数为 socket 。

默认情况下，Server 的超时为 2 分钟，如果超时将会销毁 socket。如果你给 Server 的超时事件设置了回调函数，那你就得负责处理 socket 超时。

()

server.timeout

- {Number} Default = 120000 (2 minutes)

超时的时长，单位为毫秒。

注意，socket 的超时逻辑在连接时设定，所以有新的连接时才能改变这个值。

设为 0 时，建立连接的自动超时将失效。

类： http.ServerResponse

这是一个由 HTTP 服务器（而不是用户）内部创建的对象。作为第二个参数传递给 `'request'` 事件。

该响应实现了 Writable Stream (stream.html#stream_class_stream_writable) 接口。这是一个包含下列事件的 EventEmitter (events.html#events_class_events_eventemitter)：

事件： 'close'

`function () { }`

在调用 response.end() (页 0)，或准备 flush 前，底层连接结束。

事件： 'finish'

`function () { }`

发送完响应触发。响应头和响应体最后一段数据被剥离给操作系统后，通过网络来传输时被触发。这并不代表客户端已经收到数据。

这个事件之后，响应对象不会再触发任何事件。

response.writeContinue()

发送 HTTP/1.1 100 Continue 消息给客户端，表示请求体可以发送。可以在服务器上查看'checkContinue' (页 0) 事件。

response.writeHead(statusCode[, statusMessage][, headers])

发送一个响应头给请求。状态码是 3 位数字，如 `404` 。最后一个参数 `headers` 是响应头。建议第二个参数设置为可以看的懂的消息。

例如:

```
var body = 'hello world';
response.writeHead(200, {
```

```
'Content-Length': body.length,
'Content-Type': 'text/plain' });
```

这个方法仅能在消息中调用一次，而且必须在 response.end() (页 0) 前调用。

如果你在这之前调用 response.write() (页 0) 或 response.end() (页 0),将会计算出不稳定的头。

Content-Length 是字节数，而不是字符数。上面的例子 'hello world' 仅包含一个字节字符。如果 body 包含高级编码的字符， Buffer.byteLength() 就必须确定指定编码的字符数。Node 不会检查Content-Length 和 body 的长度是否相同。

response.setTimeout(msecs, callback)

- msecs {Number}
- callback {Function}

设置 socket 超时时间，单位为毫秒。如果提供了回调函数，将会在 response 对象的 'timeout' 事件上添加监听器。

如果没有给请求、响应、服务器添加 'timeout' 监视器，超时的时候将会销毁 socket。如果你给请求、响应、服务器加了处理函数，就需要负责处理 socket 超时。

response.statusCode

使用默认的 headers 时（没有显式的调用 response.writeHead() (页 0) ），这个属性表示将要发送给客户端状态码。

例如:

```
response.statusCode = 404;
```

响应头发送给客户端的后，这个属性表示状态码已经发送。

response.statusMessage

使用默认 headers 时（没有显式的调用 response.writeHead() (页 0) ）, 这个属性表示将要发送给客户端状态信息。 如果这个没有定义，将会使用状态码的标准消息。

例如:

```
response.statusMessage = 'Not found';
```

当响应头发送给客户端的时候，这个属性表示状态消息已经发送。

response.setHeader(name, value)

设置默认头某个字段内容。如果这个头即将被发送，内容会被替换。如果你想设置更多的头， 就使用一个相同名字的字符串数组。

例如:

```
response.setHeader("Content-Type", "text/html");
```

或

```
response.setHeader("Set-Cookie", ["type=ninja", "language=javascript"]);
```

response.headersSent

Boolean (只读)。如果headers发送完毕,则为 true,反之为 false。

response.sendDate

默认值为 true。若为 true,当 headers 里没有 Date 值时，自动生成 Date 并发送。

只有在测试环境才能禁用; 因为 HTTP 要求响应包含 Date 头.

response.getHeader(name)

读取一个在队列中但是还没有被发送至客户端的header。名字是大小写敏感。仅能再头被flushed前调用。

例如:

```
var contentType = response.getHeader('content-type');
```

response.removeHeader(name)

从即将发送的队列里移除头。

例如:

```
response.removeHeader("Content-Encoding");
```

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

如果调用了这个方法，且还没有调用 response.writeHead() (页 0)，将会切换到默认的 header，并更新这个header。

这个方法将发送响应体数据块。可能会多次调用这个方法，以提供 body 成功的部分内容。

`chunk` 可以是字符串或 buffer。如果 `chunk` 是字符串，第二个参数表明如何将它编码成字节 流。`encoding` 的默认值是 `'utf8'` 。最后一个参数在刷新这个数据块时调用。

注意：这个是原始的 HTTP body，和高级的multi-part body 编码无关。

第一次调用 `response.write()` 的时候，将会发送缓存的头信息和第一个 body 给客户端。第二次，将会调用 `response.write()` 。Node 认为你将会独立发送流数据。这意味着，响应缓存在第一个数据块中。

如果成功的刷新全部数据到内核缓冲区，返回 `true` 。如果部分或全部数据在用户内存中还处于排队状况，返回 `false` 。当缓存再次释放的时候，将会触发 `'drain'` 。

response.addTrailers(headers)

这个方法给响应添加 HTTP 的尾部 header（消息末尾的 header）。

只有数据块编码用于响应体时，才会触发 Trailers；如果不是（例如，请求是HTTP/1.0），它们将会被自动丢弃。

如果你想触发 trailers， HTTP 会要求发送 `Trailer` 头，它包含一些信息，比如：

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': "7895bf4b8828b55ceaf47747b4bca667"});
response.end();
```

response.end([data][, encoding][, callback])

这个方法告诉服务器，所有的响应头和响应体已经发送；服务器可以认为消息结束。`response.end()` 方法必须在每个响应中调用。

如果指定了参数 `data` ，将会在响应流结束的时候调用。

()

http.request(options[, callback])

Node 维护每个服务器的连接来生成 HTTP 请求。这个函数让你可以发布请求。

参数 options 是对象或字符串。如果 options 是字符串，会通过 url.parse() (url.html#url_url_parse_urlstr_parsequerystring_slashesdenotehost) 自动解析。

options 值:

- host : 请求的服务器域名或 IP 地址，默认： 'localhost'
- hostname : 用于支持 url.parse() 。 hostname 优于 host
- port : 远程服务器端口。 默认： 80.
- localAddress : 用于绑定网络连接的本地接口
- socketPath : Unix域 socket (使用host:port或socketPath
- method : 指定 HTTP 请求方法。 默认： 'GET' .
- path : 请求路径。 默认： '/' 。如果有查询字符串，则需要包含。例如'/index.html?page=12'。请求路径包含非法字符时抛出异常。目前，只有空格不行，不过在未来可能改变。
- headers : 包含请求头的对象
- auth : 用于计算认证头的基本认证，即 user:password
- agent : 控制Agent的行为。当使用了一个Agent的时候，请求将默认为 Connection: keep-alive 。可能的值为:
    - undefined (default): 在这个主机和端口上使用 global Agent (页 0)
    - Agent object: 在 Agent 中显式使用 passed .
    - false : 选择性停用连接池,默认请求为： Connection: close .
    - keepAlive : {Boolean} 持资源池周围的socket，用于未来其它请求。默认值为 false 。
    - keepAliveMsecs : {Integer} 使用HTTP KeepAlive 的时候，通过正在保持活动的sockets发送TCP KeepAlive包的频繁程度。默认值为1000。仅当keepAlive为true时才相关。

可选参数 callback 将会作为一次性的监视器，添加给 'response' (页 0) 事件。

http.request() 返回一个 http.ClientRequest类的实例。ClientRequest实例是一个可写流对象。如果需要用 POST 请求上传一个文件的话，就将其写入到ClientRequest对象。

例如：

```
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, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// write data to request body
req.write(postData);
req.end();
```

注意，例子里调用了 `req.end()` 。`http.request()` 必须调用 `req.end()` 来表明请求已经完成，即使没有数据写入到请求 body 里。

如果在请求的时候遇到错误（DNS 解析、TCP 级别的错误或实际 HTTP 解析错误），在返回的请求对象时会触发一个 'error' 事件。

有一些特殊的头需要注意：

- 发送 `Connection: keep-alive` 告诉服务器保持连接，直到下一个请求到来。
- 发送 `Content-length` 头将会禁用chunked编码。

- 发送一个 `Expect` 头，会立即发送请求头，一般来说，发送 `Expect: 100-continue` ,你必须设置超时，并监听 `continue` 事件。更多细节参见 RFC2616 Section 8.2.3 。
- 发送一个授权头，将会使用 `auth` 参数重写，来计算基本的授权。

http.get(options[, callback])

因为多数请求是没有报文体的 GET 请求， Node 提供了这个简便的方法。和 `http.request()` 唯一不同点在于，这个方法自动设置 GET，并自动调用 `req.end()` 。

例如：

```
http.get("http://www.google.com/index.html", function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

()

类： http.Agent

HTTP Agent 用于 socket 池，用于 HTTP 客户端请求。

HTTP Agent 也把客户端请求默认为使用 Connection:keep-alive 。如果没有 HTTP 请求正在等着成为空闲 socket 的话，那么 socket 将关闭。这意味着，Node 的资源池在负载的情况下对 keep-alive 有利，但是仍然不需要开发人员使用 KeepAlive 来手动关闭 HTTP 客户端。

如果你选择使用 HTTP KeepAlive， 可以创建一个 Agent 对象，将 flag 设置为 `true` . (参见下面的 constructor options (页 0)) ，这样 Agent 会把没用到的 socket 放到池里，以便将来使用。他们会被显式的标志，让 Node 不运行。但是，当不再使用它的时候，需要显式的调用 `destroy()` (页 0)，这样 socket 将会被关闭。

当 socket 事件触发 `close` 事件或特殊的 `agentRemove` 事件时，socket 将会从 agent 池里移除。如果你要保持 HTTP 请求保持长时间打开，并且不希望他们在池里，可以参考以下代码：

```
http.get(options, function(res) {
  // Do stuff
}).on("socket", function (socket) {
  socket.emit("agentRemove");
});
```

另外，你可以使用 `agent:false` 让资源池停用：

```
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false  // create a new agent just for this one request
}, function (res) {
  // Do stuff with response
})
```

new Agent([options])

- `options` {Object} agent 上的设置选项集合，有以下字段内容:
  - `keepAlive` {Boolean} 持资源池周围的 socket，用于未来其它请求。默认值为 `false` 。
  - `keepAliveMsecs` {Integer} 使用 HTTP KeepAlive 的时候，通过正在保持活动的 sockets 发送 TCP KeepAlive 包的频繁程度。默认值为 1000。仅当 keepAlive 为 true 时才相关。.】
  - `maxSockets` {Number} 在空闲状态下,还依然开启的 socket 的最大值。仅当 `keepAlive` 设置为 true 的时候有效。默认值为 256。

被 `http.request` 使用的默认的 `http.globalAgent` ,会设置全部的值为默认。

必须在创建你自己的 `Agent` 对象后，才能配置这些值。

```
var http = require('http');
var keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.maxSockets

默认值为 Infinity。决定了每台主机上的 agent 可以拥有的并发 socket 的打开数量，主机可以是 `host:port` 或 `host:port:localAddress` 。

agent.maxFreeSockets

默认值 256. 对于支持 HTTP KeepAlive 的 Agent 而言，这个方法设置了空闲状态下仍然打开的套接字数的最大值。

agent.sockets

这个对象包含了当前 Agent 使用中的 socket 数组。不要修改它。

agent.freeSockets

使用 HTTP KeepAlive 的时候，这个对象包含等待当前 Agent 使用的 socket 数组。不要修改它。

agent.requests

这个对象包含了还没分配给 socket 的请求数组。不要修改它。

agent.destroy()

销毁任意一个被 agent 使用 的socket。

通常情况下不要这么做。如果你正在使用一个允许 KeepAlive 的 agent，当你知道不在使用它的时候，最好关闭 agent。否则，socket 会一直保存打开状态，直到服务器关闭。

agent.getName(options)

获取一组请求选项的唯一名，来确定某个连接是否可重用。在 http agent 里，它会返回 `host:port:localAddress` 。在 http agent 里， name 包括 CA，cert, ciphers, 和其他 HTTPS/TLS 特殊选项来决定 socket 是否可以重用。

http.globalAgent

Agent 的全局实例，是 http 客户端的默认请求。

类： http.ClientRequest

该对象在内部创建并从 `http.request()` 返回。他是 *正在* 处理的请求，其头部已经在队列中。使用 `setHeader(name, value)` ，`getHeader(name)` ，`removeHeader(name)` API 可以改变header。当关闭连接的时候，header将会和第一个数据块一起发送。

为了获取响应，可以给请求对象的 'response' 添加监听器。当接收到响应头的时候将会从请求对象里触发 'response' 。'response' 事件执行时有一个参数，该参数为 http.IncomingMessage (页 0) 的实例。

在 'response' 事件期间，可以给响应对象添加监视器，监听 'data' 事件。

如果没有添加 'response' 处理函数，响应将被完全忽略。如果你添加了 'response' 事件处理函数，那你 **必须** 消费掉从响应对象获取的数据，可以在 'readable' 事件里调用 response.read() ，或者添加一个 'data' 处理函数，或者调用 .resume() 方法。如果未读取数据，它将会消耗内存，最终产生 process out of memory 错误。

Node 不会检查 Content-Length 和 body 的长度是否相同。

该请求实现了 Writable Stream (stream.html#stream_class_stream_writable) 接口。这是一个包含下列事件的 EventEmitter (events.html#events_class_events_eventemitter) 。

事件： 'response'

`function (response) { }`

当接收到请求的时候会触发，仅会触发一次。 response 的参数是 http.IncomingMessage (页 0) 的实例。

Options:

- host : 要请求的服务器域名或 IP 地址
- port : 远程服务器的端口
- socketPath : Unix 域 Socket (使用 host:port 或 socketPath 之一)

事件： 'socket'

`function (socket) { }`

Socket 附加到这个请求的时候触发。

事件： 'connect'

`function (response, socket, head) { }`

每次服务器使用 CONNECT 方法响应一个请求时触发。如果这个这个事件未被监听，接收 CONNECT 方法的客户端将关闭他们的连接。

下面的例子展示了一对匹配的客户端/服务器如何监听 `connect` 事件。 var http = require('http'); var net = require('net'); var url = require('url');

```
// Create an HTTP tunneling proxy
var proxy = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
proxy.on('connect', function(req, cltSocket, head) {
  // connect to an origin server
  var srvUrl = url.parse('http://' + req.url);
  var srvSocket = net.connect(srvUrl.port, srvUrl.hostname, function() {
    cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
                    'Proxy-agent: Node-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', function() {

  // 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', function(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', function(chunk) {
      console.log(chunk.toString());
```

```
  });
  socket.on('end', function() {
    proxy.close();
  });
 });
});
```

事件： 'upgrade'

`function (response, socket, head) { }`

每当服务器响应 upgrade 请求时触发。如果没有监听这个事件,客户端会收到 upgrade 头后关闭连接。

下面的例子展示了一对匹配的客户端/服务器如何监听 `upgrade` 事件。

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

// Create an HTTP server
var srv = http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('okay');
});
srv.on('upgrade', function(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', function() {

 // 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', function(res, socket, upgradeHead) {
    console.log('got upgraded!');
    socket.end();
    process.exit(0);
  });
});
```

事件： 'continue'

`function () { }`

当服务器发送 '100 Continue' HTTP 响应的时候触发，通常因为请求包含 'Expect: 100-continue'。该指令表示客户端应发送请求体。

request.flushHeaders()

刷新请求的头。

考虑效率因素，Node.js 通常会缓存请求的头直到你调用 `request.end()` ，或写入请求的第一个数据块。然后，包装请求的头和数据到一个独立的 TCP 包里。

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

发送一个请求体的数据块。通过多次调用这个函数，用户能流式的发送请求给服务器，这种情况下，建议使用 `['Transfer-Encoding', 'chunked']` 头。

`chunk` 参数必须是 Buffer (buffer.html#buffer_buffer) 或字符串。

回调参数可选，当这个数据块被刷新的时候会被调用。

request.end([data][, encoding][, callback])

发送请求完毕。如果 body 的数据没被发送，将会将他们刷新到流里。如果请求是分块的，该方法会发送终结符 0\r\n\r\n 。

如果指定了 `data` ，等同于先调用 `request.write(data, encoding)` ，再调用 `request.end(callback)` 。

如果有 `callback` ，将会在请求流结束的时候调用。

request.abort()

终止一个请求. (v0.3.8 开始新加入)。

request.setTimeout(timeout[, callback])

如果 socket 被分配给这个请求，并完成连接，将会调用 socket.setTimeout() (net.html#net_socket_settimeout_timeout_callback) 。

request.setNoDelay([noDelay])

如果 socket 被分配给这个请求，并完成连接，将会调用 socket.setNoDelay() (net.html#net_socket_setnodelay_nodelay) 。

request.setSocketKeepAlive([enable][, initialDelay])

如果 socket 被分配给这个请求，并完成连接，将会调用 socket.setKeepAlive() (net.html#net_socket_setkeepalive_enable_initialdelay) 。

http.IncomingMessage

http.Server (页 0) 或 http.ClientRequest (页 0) 创建了 `IncomingMessage` 对象，作为第一个参数传递给 `response'` 。它可以用来访问应答的状态，头文件和数据。

它实现了 Readable Stream (stream.html#stream_class_stream_readable) 接口，以及以下额外的事件，方法和属性。

事件： 'close'

`function () { }`

表示底层连接已经关闭。 和 `'end'` 类似，这个事件每个应答只会发送一次。

message.httpVersion

客户端向服务器发送请求时，客户端发送的 HTTP 版本；或者服务器想客户端返回应答时，服务器的 HTTP 版本。通常是 `'1.1'` 或 `'1.0'` 。

另外， `response.httpVersionMajor` 是第一个整数， `response.httpVersionMinor` 是第二个整数。

message.headers

请求/响应头对象。

只读的头名称和值的映射。头的名字是小写，比如：

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

message.rawHeaders

接收到的请求/响应头字段列表。

注意，键和值在同一个列表中。它并非一个元组列表。所以，偶数偏移量为键，奇数偏移量为对应的值。

头名字不是小写敏感，也没用合并重复的头。 // Prints something like: // // ['user-agent', // 'this is invalid because there can be only one', // 'User-Agent', // 'curl/7.22.0', // 'Host', // '127.0.0.1:8000', // 'ACCEPT', // '/'] console.log(request.rawHeaders);

message.trailers

请求/响应 的尾部对象。只在 'end' 事件中存在。

message.rawTrailers

接收到的原始的请求/响应尾部键和值。仅在 'end' 事件中存在。

message.setTimeout(msecs, callback)

- `msecs` {Number}
- `callback` {Function}

调用 `message.connection.setTimeout(msecs, callback)`.

message.method

仅对从 http.Server (页 0) 获得的请求有效。

请求方法如果一个只读的字符串。 例如：`'GET'`，`'DELETE'`.

message.url

仅对从 http.Server (页 0) 获得的请求有效。

请求的 URL 字符串。它仅包含实际的 HTTP 请求中所提供的 URL，比如请求如下：

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

`request.url` 就是:

```
'/status?name=ryan'
```

如果你想将 URL 分解，可以用 `require('url').parse(request.url)`，例如：

```
node> require('url').parse('/status?name=ryan')
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status' }
```

如果想从查询字符串中解析出参数，可以用 `require('querystring').parse` 函数，或者将 `true` 作为第二个参数传递给 `require('url').parse`。 例如：

```
node> require('url').parse('/status?name=ryan', true)
{ href: '/status?name=ryan',
  search: '?name=ryan',
```

```
query: { name: 'ryan' },
pathname: '/status' }
```

message.statusCode

仅对从 `http.ClientRequest` 获取的响应有效。

3位数的 HTTP 响应状态码 `404` 。

message.statusMessage

仅对从 `http.ClientRequest` 获取的响应有效。

HTTP 的响应消息。比如， `OK` 或 `Internal Server Error` .

message.socket

和连接相关联的 `net.Socket` 对象。

通过 HTTPS 支持，使用 `request.connection.verifyPeer()` 和 `request.connection.getPeerCertificate()` 获取客户端的身份信息。

HTTPS

稳定性: 3 – 稳定

HTTPS 是基于 TLS/SSL 的 HTTP 协议。在 Node 里作为单独的模块来实现。

类: https.Server

这是 `tls.Server` 的子类，并且和 `http.Server` 一样触发事件。更多信息参见 `http.Server` 。

server.setTimeout(msecs, callback)

详情参见 http.Server#setTimeout() (http.md#http_server_settimeout_msecs_callback) .

server.timeout

详情参见 http.Server#timeout (http.md#http_server_timeout) .

https.createServer(options[, requestListener])

返回一个新的 HTTPS 服务器对象。其中 `options` 类似于 [tls.createServer()][tls.md#tls_tls_createserver_options_secureconnectionlistener]。 `requestListener` 函数自动加到 `'request'` 事件里。

例如:

```
// curl -k https://localhost:8000/
var https = require('https');
var fs = require('fs');

var options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

或

```
var https = require('https');
var fs = require('fs');

var options = {
  pfx: fs.readFileSync('server.pfx')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

server.listen(port[, host][, backlog][, callback])

server.listen(path[, callback])

server.listen(handle[, callback])

详情参见 http.listen() (http.md#http_server_listen_port_hostname_backlog_callback) 。

server.close([callback])

详情参见 http.close() (http.md#http_server_close_callback) 。

()

https.request(options, callback)

发送请求到安全 web 服务器。

`options` 可以是一个对象或字符串。如果 `options` 是字符串。会被 url.parse() (url.md#url.parse) 解析。

所有来自 http.request() (http.md#http_http_request_options_callback) 选项都是经过验证的。

例如:

```
var https = require('https');

var options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
```

```
  method: 'GET'
};

var req = https.request(options, function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

option 参数有以下的值：

- `host`：请求的服务器域名或 IP 地址，默认：`'localhost'`
- `hostname`：用于支持 `url.parse()`。`hostname` 优于 `host`
- `port`：远程服务器端口。默认：`443`.
- `method`：指定 HTTP 请求方法。默认：`'GET'`.
- `path`：请求路径。默认：`'/'`。如果有查询字符串，则需要包含。比如 `'/index.html?page=12'`
- `headers`：包含请求头的对象
- `auth`：用于计算认证头的基本认证，即 `user:password`
- `agent`：控制Agent的行为。当使用了一个Agent的时候，请求将默认为 `Connection: keep-alive`。可能的值为：
  - `undefined` (default): 在这个主机和端口上使用 [global Agent][]
  - `Agent` object: 在 `Agent` 中显式使用 passed.
  - `false`：选择性停用连接池,默认请求为：`Connection: close`

tls.connect() (tls.md#tls_tls_connect_options_callback) 的参数也能指定。但是，globalAgent (https.md#https_https_globalagent) 会忽略他们。

- `pfx`：SSL 使用的证书，私钥，和证书Certificate, 默认 `null`.
- `key`：SSL 使用的私钥. 默认 `null`.
- `passphrase`：私钥或 pfx 的口令字符串. 默认 `null`.

- `cert`: 所用公有 x509 证书. 默认 `null`.
- `ca`: 用于检查远程主机的证书颁发机构或包含一系列证书颁发机构的数组。
- `ciphers`: 描述要使用或排除的密码的字符串，格式请参阅 http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT
- `rejectUnauthorized`: 如为 `true` 则服务器证书会使用所给 CA 列表验证。如果验证失败则会触发 `error` 事件。验证过程发生于连接层，在 `HTTP` 请求发送之前。缺省为 `true`.
- `secureProtocol`: 所用的 SSL 方法，比如 `TLSv1_method` 强制使用 TLS version 1。可取值取决于您安装的 OpenSSL, 和定义于 SSL_METHODS (http://www.openssl.org/docs/ssl/ssl.html#DEALING_WITH_PROTOCOL_METHODS) 的常量。

要指定这些选项，使用一个自定义 `Agent`.

例如:

```
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, function(res) {
  ...
}
```

或者不使用 `Agent`.

例如:

```
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, function(res) {
  ...
}
```

https.get(options, callback)

和 `http.get()` 类似，不过是 HTTPS 版本的.

`options` 可以是字符串对象. 如果 `options` 是字符串, 会自动使用 url.parse() (页 324) 解析。

例如:

```
var https = require('https');

https.get('https://encrypted.google.com/', function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });

}).on('error', function(e) {
  console.error(e);
});
```

()

类: https.Agent

HTTPS 的 Agent 对象，和 http.Agent (http.md#http_class_http_agent) 类似。 详情参见 https.request() (https.md#https_https_request_options_callback) 。

()

https.globalAgent

所有 HTTPS 客户端请求的 https.Agent (https.md#https_class_https_agent) 全局实例。

17

模块

稳定性: 5 – 锁定

Node 有简单的模块加载系统。在 Node 里，文件和模块是一一对应的。下面例子里， foo.js 加载同一个文件夹里的 circle.js 模块。

foo.js 内容:

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

circle.js 内容:

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

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

circle.js 模块输出了 area() 和 circumference() 函数。想要给根模块添加函数和对象，你可以将他们添加到特定的 exports 对象。

加载到模块里的变量是私有的，仿佛模块是包含在一个函数里。在这个例子里， PI 是 circle.js 的私有变量。

如果你想模块里的根像一个函数一样的输出（比如 构造函数），或者你想输出一个完整对象,那就分派给 module.exports ，而不是 exports 。

bar.js 使用 square 模块, 它输出了构造函数:

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

square 定义在 square.js 文件里:

```
// assigning to exports will not modify module, must use module.exports
module.exports = function(width) {
  return {
    area: function() {
      return width * width;
    }
```

```
 };
}
```

模块系统在 `require("module")` 模块里实现。

Cycles

环形调用 `require()` ，当返回时模块可能都没执行结束。

考虑以下场景:

`a.js` :

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

`b.js` :

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

`main.js` :

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

当 `main.js` 加载 `a.js` ， `a.js` 加载 `b.js` 。此时， `b.js` 试着加载 `a.js` 。为了阻止循环调用， `a.js` 输出对象的不完全拷贝返回给 `b.js` 模块。 `b.js` 会结束加载，并且它的 `exports` 对象提供给 `a.js` 模块。

`main.js` 加载完两个模块时，它们都会结束。这个程序的输出如下:

```
$ 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
```

如果你的程序有环形模块依赖，需要保证是线性的。

核心模块

Node 有很多模块编译成二进制。这些模块在本文档的其他地方有更详细的描述。

核心模块定义在 Node 的源代码 `lib/` 目录里。

`require()` 总是会优先加载核心模块。例如， `require('http')` 总是返回编译好的 HTTP 模块，而不管这个文件的名字。

文件模块

如果按照文件名没有找到模块，那么 Node 会试着加载添加了后缀 `.js` ，`.json` 的文件，如果还没好到，再试着加载添加了后缀 `.node` 的文件。

`.js` 会解析为 JavaScript 的文本文件， `.json` 会解析为 JSON 文本文件， `.node` 会解析为编译过的插件模块，由 `dlopen` 负责加载。

模块的前缀 `'/'` 表示绝对路径。例如 `require('/home/marco/foo.js')` 将会加载 `/home/marco/foo.js` 文件。

模块的前缀 `'./'` 表示相对于调用 `require()` 的路径。就是说， `circle.js` 必须和 `foo.js` 在 同一个目录里， `require('./circle')` 才能找到。

文件前没有 `/` 或 `./` 前缀，表示模块可能是 `core module` ，或者已经从 `node_modules` 文件夹里加载过了。

如果指定的路径不存在， `require()` 将会抛出一个 `code` 属性为 `'MODULE_NOT_FOUND'` 的异常。

从 `node_modules` 目录里加载

如传递给 `require()` 的模块不是一个本地模块，并且不以 `'/'` , `'../'` , 或 `'./'` 开头，那么 Node 会从当前模块的父目录开始，尝试在它的 `node_modules` 文件夹里加载模块。

如果没有找到，那么会到父目录，直到到文件系统的根目录里找。

例如，如果 '/home/ry/projects/foo.js' 里的文件加载 require('bar.js') ，那么 Node 将会按照下面的顺序查找:

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

这样允许程序独立，不会产生冲突。

可以请求指定的文件或分布子目录里的模块，在模块名后添加路径后缀。例如， require('example-module/path/to/file') 会解决 path/to/file 相对于 example-module 的加载位置。路径后缀使用相同语法。

文件夹作为模块

可以把程序和库放到独立的文件夹里，并提供单一的入口指向他们。有三种方法可以将文件夹作为参数传给 require() 。

第一个方法是，在文件夹的根创建一个 package.json 文件，它指定了 main 模块。 package.json 的例子如下:

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

如果这是在 ./some-library 里的文件夹， require('./some-library') 将会试着加载 ./some-library/lib/some-library.js 。

如果文件夹里没有 package.json 文件，Node 会试着加载 index.js 或 index.node 文件。例如，如果上面的例子里没有 package.json 文件。那么 require('./some-library') 将会试着加载:

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

缓存

模块第一次加载后会被被缓存。这就是说，每次调用 require('foo') 都会返回同一个对象，当然，必须每次都要解析到同一个文件。

多次调用 require('foo') 也许不会导致模块代码多次执行。这是很重要的特性，这样就可以返回 "partially done" 对象，允许加载过渡性的依赖关系，即使可能会引起环形调用。

如果你希望多次调用一个模块，那么就输出一个函数，然后调用这个函数。

模块换成预警

模块的缓存依赖于解析后的文件名。因此随着调用位置的不同，模块可能解析到不同的文件（例如，从 node_modules 文件夹加载）。如果解析为不同的文件，require('foo') 可能会返回不同的对象。

module 对象

- {Object}

在每个模块中，变量 module 是一个代表当前模块的对象的引用。为了方便，module.exports 可以通过 exports 全局模块访问。module 不是事实上的全局对象，而是每个模块内部的。

module.exports

- {Object}

模块系统创建 module.exports 对象。很多人希望自己的模块是某个类的实例。因此，把将要导出的对象赋值给 module.exports 。注意，将想要的对象赋值给 exports ，只是简单的将它绑定到本地 exports 变量，这可能并不是你想要的。

例如，假设我们有一个模块叫 a.js 。

```
var EventEmitter = require('events').EventEmitter;

module.exports = new EventEmitter();

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

另一个文件可以这么写：

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

注意，赋给 `module.exports` 必须马上执行，并且不能在回调中执行。

x.js:

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

y.js:

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

exports alias

`exports` 变量在引用到 `module.exports` 的模块里可用。和其他变量一样，如果你给他赋一个新的值，它不再指向老的值。

为了展示这个特性，假设实现： `require()`：

```
function require(...) {
  // ...
  function (module, exports) {
    // Your module code here
    exports = some_func;        // re-assigns exports, exports is no longer
                                // a shortcut, and nothing is exported.
    module.exports = some_func; // makes your module export 0
  } (module, module.exports);
  return module;
}
```

如果你对 `exports` 和 `module.exports` 间的关系感到迷糊，那就只用 `module.exports` 就好。

module.require(id)

- `id` {String}
- 返回: {Object} 已经解析模块的 `module.exports`

`module.require` 方法提供了一种像 `require()` 一样从最初的模块加载一个模块的方法。

为了能这样做，你必须获得 module 对象的引用。require() 返回 module.exports，并且 module 是一个典型的只能在特定模块作用域内有效的变量，如果要使用它，就必须明确的导出。

module.id

- {String}

模块的标识符。通常是完全解析的文件名。

module.filename

- {String}

模块完全解析的文件名。

module.loaded

- {Boolean}

模块是已经加载完毕，还是在加载中。

module.parent

- {Module Object}

引入这个模块的模块。

module.children

- {Array}

由这个模块引入的模块。

其他...

为了获取即将用 require() 加载的准确文件名，可以使用 require.resolve() 函数。

综上所述，下面用伪代码的高级算法形式演示了 require.resolve 的工作流程：

```
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.  STOP
4. If X.node is a file, load X.node as binary addon.  STOP

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

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

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

从全局文件夹加载

如果环境变量 NODE_PATH 设置为冒号分割的绝对路径列表，并且在模块在其他地方没有找到，Node 将会搜索这些路径。（注意，Windows 里， NODE_PATH 用分号分割 ）。

另外， Node 将会搜索这些路径。

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

$HOME 是用户的 home 文件夹， $PREFIX 是 Node 里配置的 node_prefix 。

这大多是历史原因照成的。强烈建议将所以来的模块放到 node_modules 文件夹里。这样加载会更快。

访问主模块

当 Node 运行一个文件时， require.main 就会设置为它的 module 。也就是说你可以通过测试判断文件是否被直接运行。

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

对于 foo.js 文件。 如果直接运行 node foo.js ，返回 true ， 如果通过 require('./foo') 是间接运行。

因为 module 提供了 filename 属性（通常等于 __filename ），程序的入口点可以通过检查 require.main.filename 来获得。

附录: 包管理技巧

Node 的 require() 函数语义定义的足够通用，它能支持各种常规目录结构。诸如 dpkg , rpm , 和 npm 包管理程序，不用修改就可以从 Node 模块构建本地包。

下面我们介绍一个可行的目录结构：

假设我们有一个文件夹 /usr/lib/node/<some-package>/<some-version> ，包含指定版本的包内容。

一个包可以依赖于其他包。为了安装包 foo，可能需要安装特定版本的 bar 包。 bar 包可能有自己的包依赖，某些条件下，依赖关系可能会发生冲突或形成循环。

因为 Node 会查找他所加载的模块的 realpath（也就是说会解析符号链接），然后按照上文描述的方式在 node_modules 目录中寻找依赖关系，这种情形跟以下体系结构非常相像：

- /usr/lib/node/foo/1.2.3/ – foo 包, version 1.2.3.
- /usr/lib/node/bar/4.3.2/ – foo 依赖的 bar 包内容
- /usr/lib/node/foo/1.2.3/node_modules/bar – 指向 /usr/lib/node/bar/4.3.2/ 的符号链接
- /usr/lib/node/bar/4.3.2/node_modules/* – 指向 bar 包所依赖的包的符号链接

因此，即使存在循环依赖或依赖冲突，每个模块还可以获得他所依赖的包得可用版本。

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

另外，为了让模块搜索更快些，不要将包直接放在 /usr/lib/node 目录中，而是将它们放在 /usr/lib/node_modules/<name>/<version> 目录中。 这样在依赖的包找不到的情况下，就不会一直寻找 / usr/node_modules 目录或 /node_modules 目录了。基于调用 require() 的文件所在真实路径，因此包本身可以放在任何位置。

为了让 Node 模块对 Node REPL 可用，可能需要将 /usr/lib/node_modules 文件夹路径添加到环境变量 $NODE_PATH 。由于模块查找 $NODE_PATH 文件夹都是相对路径，因此包可以放到任何位置。

18

网络

稳定性: 3 – 稳定

net 模块提供了异步网络封装，它包含了创建服务器/客户端的方法（调用 streams ）。可以通过调用 require('net') 包含这个模块。

net.createServer([options][, connectionListener])

创建一个 TCP 服务器。参数 connectionListener 自动给 'connection' (页 202) 事件创建监听器。

options 包含有以下默认值:

```
{
  allowHalfOpen: false,
  pauseOnConnect: false
}
```

如果 allowHalfOpen = true， 当另一端 socket 发送 FIN 包时 socket 不会自动发送 FIN 包。socket 变为不可读，但仍可写。你需要显式的调用 end() 方法。更多信息参见 'end' (页 204) 事件。

如果 pauseOnConnect = true ,当连接到来的时候相关联的 socket 将会暂停。它允许在初始进程不读取数据情况下，让连接在进程间传递。调用 resume() 从暂停的 socket 里读取数据。

下面是一个监听 8124 端口连接的应答服务器的例子:

```
var net = require('net');
var server = net.createServer(function(c) { //'connection' listener
  console.log('client connected');
  c.on('end', function() {
    console.log('client disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.listen(8124, function() { //'listening' listener
  console.log('server bound');
});
```

使用 telnet 来测试:

```
telnet localhost 8124
```

要监听 socket t /tmp/echo.sock，仅需要改倒数第三行代码:

```
server.listen('/tmp/echo.sock', function() { //'listening' listener
```

使用 nc 连接到一个 UNIX domain socket 服务器:

```
nc -U /tmp/echo.sock必须
```

net.connect(options[, connectionListener])

net.createConnection(options[, connectionListener])

工厂方法，返回一个新的 'net.Socket' (页 0)，并连接到指定的地址和端口。

当 socket 建立的时候，将会触发 'connect' (页 0) 事件。

和'net.Socket' (页 0)有相同的方法。

对于 TCP sockets，参数 options 因为下列参数的对象:

- port : 客户端连接到 Port 的端口（必须）。
- host : 客户端要连接到得主机。默认 'localhost' .
- localAddress : 网络连接绑定的本地接口。
- localPort : 网络连接绑定的本地端口。
- family : IP 栈版本。默认 4 .

对于本地域socket，参数 options 因为下列参数的对象:

- path : 客户端连接到得路径(必须).

通用选项:

- 如果 allowHalfOpen = true ， 当另一端 socket 发送 FIN 包时 socket 不会自动发送 FIN 包。socket 变为不可读，但仍可写。你需要显式的调用 end() 方法。更多信息参见 'end' (页 204) 事件。

  connectListener 参数将会作为监听器添加到'connect' (页 0)事件上。

下面是一个用上述方法应答服务器的客户端例子:

```
var net = require('net');
var client = net.connect({port: 8124},
    function() { //'connect' listener
  console.log('connected to server!');
  client.write('world!\r\n');
```

```
});
client.on('data', function(data) {
  console.log(data.toString());
  client.end();
});
client.on('end', function() {
  console.log('disconnected from server');
});
```

要连接到 socket `/tmp/echo.sock`，仅需将第二行代码改为：

```
var client = net.connect({path: '/tmp/echo.sock'});
```

net.connect(port[, host][, connectListener])

net.createConnection(port[, host][, connectListener])

创建一个到端口 `port` 和 主机 `host` 的 TCP 连接。如果忽略主机 `host`，则假定为 `'localhost'`。参数 `connectListener` 将会作为监听器添加到 'connect' (页 0) 事件。

这是工厂方法，返回一个新的 'net.Socket' (页 0)。

net.connect(path[, connectListener])

net.createConnection(path[, connectListener])

创建到 `path` 的 unix socket 连接。 参数 `connectListener` 将会作为监听器添加到 'connect' (页 0) 事件上。

这是工厂方法，返回一个新的 'net.Socket' (页 0)。

Class: net.Server

这个类用来创建一个 TCP 或本地服务器。

server.listen(port[, host][, backlog][, callback])

开始接受指定端口 `port` 和 主机 `host` 的连接。如果忽略主机 `host`， 服务器将会接受任何 IPv4 地址(`INADDR_ANY`)的直接连接。端口为 0，则会分配一个随机端口。

积压量（Backlog）为连接等待队列的最大长度。实际长度由您的操作系统通过 sysctl 设定，比如 linux 上的 tcp_max_syn_backlog 和 somaxconn 。这个参数默认值是 511 (不是 512)。

这是异步函数。当服务器被绑定时会触发 'listening' (页 0) 事件。最后一个参数 callback 将会作为'listening' (页 0) 事件的监听器。

有些用户会遇到 EADDRINUSE 错误，它表示另外一个服务器已经运行在所请求的端口上。处理这个情况的办法是等一段事件再重试：

```
server.on('error', function (e) {
  if (e.code == 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(function () {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

(注意: Node 中的所有 socket 已设置了 SO_REUSEADDR)

server.listen(path[, callback])

- path {String}
- callback {Function}

启动一个本地 socket 服务器，监听指定 path 的连接。

这是异步函数。绑定服务器后，会触发 'listening' (页 0) 事件。最后一个参数 callback 将会作为'listening' (页 0) 事件的监听器。

UNIX 上，本地域通常默认为 UNIX 域。参数 path 是文件系统路径，就和创建文件时一样，它也遵从命名规则和权限检查，并且在文件系统里可见，并持续到关闭关联。

Windows上，本地域通过命名管道实现。路径必须是以 \\?\pipe\ 或 \\.\pipe\ 入口。任意字符串都可以，不过之后进行相同的管道命名处理，比如解决 .. 序列。管道命名空间是平的。管道不会一直持久，当最后一个引用关闭的时候，管道将会移除。不要忘记 javascript 字符字符串转义要求路径使用双反斜杠，比如：

```
net.createServer().listen(
  path.join('\\\\?\\pipe', process.cwd(), 'myctl'))
```

server.listen(handle[, callback])

- `handle` {Object}
- `callback` {Function}

  `handle` 对象可以设置成 server 或 socket（任意以下划线 `_handle` 开头的成员），或者是 `{fd: <n>}` 对象。

这将是服务器用指定的句柄接收连接，前提是文件描述符或句柄已经绑定到端口或域 socket。

Windows 不支持监听文件句柄。

这是异步函数。当服务器已经被绑定，将会触发'listening' (页 0) 事件。最后一个参数 `callback` 将会作为 'listening' (页 0) 事件的监听器。

server.listen(options[, callback])

- `options` {Object} – 必须. 支持以下属性:
  - `port` {Number} – 可选.
  - `host` {String} – 可选.
  - `backlog` {Number} – 可选.
  - `path` {String} – 可选.
  - `exclusive` {Boolean} – 可选.
- `callback` {Function} – 可选.

`options` 的属性：端口 `port`，主机 `host`，和 `backlog`，以及可选参数 callback 函数，他们在一起调用server.listen(port, [host], [backlog], [callback]) (页 0)。还有，参数 `path` 可以用来指定 UNIX socket。

如果参数 `exclusive` 是 `false`（默认值），集群进程将会使用同一个句柄，允许连接共享。当参数 `exclusive` 是 `true` 时，句柄不会共享，如果共享端口会返回错误。监听独家端口例子如下：

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

server.close([callback])

服务器停止接收新的连接，保持现有连接。这是异步函数，当所有连接结束的时候服务器会关闭，并会触发 'close' 事件。你可以传一个回调函数来监听 'close' 事件。如果存在，将会调用回调函数，错误（如果有）作为唯一参数。

server.address()

操作系统返回绑定的地址，协议族名和服务器端口。查找哪个端口已经被系统绑定时，非常有用。返回的对象有3个属性，比如： { port: 12346, family: 'IPv4', address: '127.0.0.1' }

例如:

```
var server = net.createServer(function (socket) {
  socket.end("goodbye\n");
});

// grab a random port.
server.listen(function() {
  address = server.address();
  console.log("opened server on %j", address);
});
```

在 'listening' 事件触发前，不要调用 server.address() 。

server.unref()

如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。如果服务器已被 unref，则再次调用 unref 并不会产生影响。

server.ref()

与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref ，则再次调用 ref 并不会产生影响。

server.maxConnections

设置这个选项后，当服务器连接数超过数量时拒绝新连接。

一旦已经用 child_process.fork() 方法将 socket 发送给子进程， 就不推荐使用这个选项。

server.connections

已经抛弃这个函数。请用 server.getConnections() (页 0) 代替。服务器上当前连接的数量。

当调用 child_process.fork() 发送一个 socket 给子进程时，它将变为 null 。 要轮询子进程来获取当前活动连接的数量，请用 server.getConnections 代替.

server.getConnections(callback)

异步获取服务器当前活跃连接的数量。当 socket 发送给子进程后才有效；

回调函数有 2 个参数 err 和 count 。

net.Server 是事件分发器 EventEmitter (events.html#events_class_events_eventemitter) ， 有以下事件:

事件： 'listening'

当服务器调用 server.listen 绑定后会触发。

()

事件：'connection'

- {Socket object} 连接对象

当新连接创建后会被触发。 socket 是 net.Socket 实例。

事件： 'close'

服务器关闭时会触发。注意，如果存在连接，这个事件不会被触发直到所有的连接关闭。

事件： 'error'

- {Error Object}

发生错误时触发。'close' 事件将被下列事件直接调用。请查看 server.listen 例子。

Class: net.Socket

这个对象是 TCP 或 UNIX Socket 的抽象。net.Socket 实例实现了一个双工流接口。他们可以在用户创建客户端(使用 connect())时使用, 或者由 Node 创建它们，并通过 connection 服务器事件传递给用户。

new net.Socket([options])

构造一个新的 socket 对象。

options 对象有以下默认值:

```
{ fd: null
  allowHalfOpen: false,
  readable: false,
  writable: false
}
```

参数 fd 允许你指定一个存在的文件描述符。将 readable 和（或） writable 设为 true ，允许在这个 socket 上读和（或）写（注意，仅在参数 fd 有效时）。关于 allowHalfOpen ，参见 createServer() 和 'end' 事件。

socket.connect(port[, host][, connectListener])

socket.connect(path[, connectListener])

使用传入的 socket 打开一个连接。如果指定了端口 port 和 主机 host ，TCP socket 将打开 socket 。如果忽略参数 host ，则默认为 localhost 。如果指定了 path ，socket 将会被指定路径的 unix socket 打开。

通常情况不需要使用这个函数，比如使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。

这是异步函数。当 'connect' (页 0) 事件被触发时， socket 已经建立。如果这是问题连接， 'connect' 事件不会被触发， 将会抛出 'error' 事件。

参数 connectListener 将会作为监听器添加到 'connect' (页 0) 事件。

socket.bufferSize

是 `net.Socket` 的一个属性，用于 `socket.write()` 。帮助用户获取更快的运行速度。计算机不能一直处于写入大量数据状态--网络连接可能太慢。Node 在内部会将排队数据写入到socket，并在网络可用时发送。（ 内部实现：轮询 socket 的文件描述符直到变为可写 ）。

这种内部缓冲的缺点是会增加内存使用量。这个属性表示当前准备写的缓冲字符数。（ 字符的数量等于准备写入的字节的数量，但是缓冲区可能包含字符串，这些字符串是惰性编码的，所以准确的字节数还无法知道 ）。

遇到很大增长很快的 `bufferSize` 时，用户可用尝试用 `pause()` 和 `resume()` 来控制字符流。

socket.setEncoding([encoding])

设置 socket 的编码为可读流。更多信息参见stream.setEncoding() (stream.html#stream_stream_setencoding_encoding)

socket.write(data[, encoding][, callback])

在 socket 上发送数据。第二个参数指定了字符串的编码，默认是 UTF8 编码。

如果所有数据成功刷新到内核缓冲区，返回 `true` 。如果数据全部或部分在用户内存里，返回 `false` 。当缓冲区为空的时候会触发 `'drain'` 。

当数据最终被完整写入的的时候，可选的 `callback` 参数会被执行，但不一定会马上执行。

()

socket.end([data][, encoding])

半关闭 socket。例如，它发送一个 FIN 包。可能服务器仍在发送数据。

如果参数 `data` 不为空，等同于调用 `socket.write(data, encoding)` 后再调用 `socket.end()` 。

socket.destroy()

确保没有 I/O 活动在这个套接字上。只有在错误发生情况下才需要。（ 处理错误等等 ）。

socket.pause()

暂停读取数据。就是说，不会再触发 `data` 事件。对于控制上传非常有用。

socket.resume()

调用 `pause()` 后想恢复读取数据。

socket.setTimeout(timeout[, callback])

socket 闲置时间超过 `timeout` 毫秒后 ，将 socket 设置为超时。

触发空闲超时事件时，socket 将会收到 `'timeout'` 事件，但是连接不会被断开。用户必须手动调用 `end()` 或 `destroy()` 这个socket。

如果 `timeout` = 0, 那么现有的闲置超时会被禁用

可选的 callback 参数将会被添加成为 'timeout' 事件的一次性监听器。

socket.setNoDelay([noDelay])

禁用纳格（Nagle）算法。默认情况下 TCP 连接使用纳格算法，在发送前他们会缓冲数据。将 `noDelay` 设置为 `true` 将会在调用 `socket.write()` 时立即发送数据。 `noDelay` 默认值为 `true` 。

socket.setKeepAlive([enable][, initialDelay])

禁用/启用长连接功能，并在发送第一个在闲置 socket 上的长连接 probe 之前，可选地设定初始延时。默认为 false。

设定 `initialDelay` （毫秒），来设定收到的最后一个数据包和第一个长连接probe之间的延时。将 initialDelay 设为0，将会保留默认（或者之前）的值。默认值为0.

socket.address()

操作系统返回绑定的地址，协议族名和服务器端口。返回的对象有 3 个属性，比如 `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }` 。

socket.unref()

如果这是事件系统中唯一一个活动的服务器，调用 `unref` 将允许程序退出。如果服务器已被 unref，则再次调用 unref 并不会产生影响。

socket.ref()

与 `unref` 相反，如果这是唯一的服务器，在之前被 `unref` 了的服务器上调用 `ref` 将不会让程序退出（默认行为）。如果服务器已经被 `ref`，则再次调用 `ref` 并不会产生影响。

socket.remoteAddress

远程的 IP 地址字符串，例如： `'74.125.127.100'` or `'2001:4860:a005::68'`.

socket.remoteFamily

远程IP协议族字符串，比如 `'IPv4'` or `'IPv6'`.

socket.remotePort

远程端口，数字表示，例如： `80` or `21`.

socket.localAddress

网络连接绑定的本地接口 远程客户端正在连接的本地 IP 地址，字符串表示。例如，如果你在监听'0.0.0.0'而客户端连接在'192.168.1.1'，这个值就会是 '192.168.1.1'。

socket.localPort

本地端口地址，数字表示。例如： `80` or `21`.

socket.bytesRead

接收到得字节数。

socket.bytesWritten

发送的字节数。

net.Socket 是事件分发器 EventEmitter (events.html#events_class_events_eventemitter) 的实例， 有以下事件:

事件： 'lookup'

在解析域名后，但在连接前，触发这个事件。对 UNIX sokcet 不适用。

- err {Error | Null} 错误对象。 参见 dns.lookup() (dns.html#dns_dns_lookup_domain_family_callback) .
- address {String} IP 地址。
- family {String | Null} 地址类型。 参见 dns.lookup() (dns.html#dns_dns_lookup_domain_family_callback) .

事件： 'connect'

当成功建立 socket 连接时触发。 参见 connect() .

事件： 'data'

- {Buffer object}

当接收到数据时触发。参数 data 可以是 Buffer 或 String 。使用 socket.setEncoding() 设定数据编码。(更多信息参见 Readable Stream (stream.html#stream_class_stream_readable))。

当 Socket 触发一个 'data' 事件时，如果没有监听器，数据将会丢失。

事件： 'end'

当 socket 另一端发送 FIN 包时，触发该事件。

默认情况下(allowHalfOpen == false)，一旦 socket 将队列里的数据写完毕，socket 将会销毁它的文件描述符。如果 allowHalfOpen == true ,socket 不会从它这边自动调用 end()，使的用户可以随意写入数据，而让用户端自己调用 end()。

事件： 'timeout'

当 socket 空闲超时时触发，仅是表明 socket 已经空闲。用户必须手动关闭连接。

参见： `socket.setTimeout()`

事件： 'drain'

当写缓存为空得时候触发。可用来控制上传。

参见： `socket.write()` 的返回值。

事件： 'error'

- {Error object}

错误发生时触发。以下事件将会直接触发 `'close'` 事件。

事件： 'close'

- `had_error` {Boolean} 如果 socket 传输错误，为 `true`

当 socket 完全关闭时触发。参数 `had_error` 是 boolean，它表示是否因为传输错误导致 socket 关闭。

net.isIP(input)

测试是否输入的为 IP 地址。字符串无效时返回 0。 IPV4 情况下返回 4， IPV6情况下返回 6.

net.isIPv4(input)

如果输入的地址为 IPV4， 返回 true，否则返回 false。

net.isIPv6(input)

如果输入的地址为 IPV6， 返回 true，否则返回 false。

19

系统

稳定性: 4 – API 冻结

提供一些基本的操作系统相关函数。

使用 `require('os')` 访问这个模块。

os.tmpdir()

返回操作系统的默认临时文件夹

os.endianness()

返回 CPU 的字节序，可能的是 "BE" 或 "LE"。

os.hostname()

返回操作系统的主机名。

os.type()

返回操作系统名。

os.platform()

返回操作系统名

os.arch()

返回操作系统 CPU 架构，可能的值有 "x64"、"arm" 和 "ia32"。

os.release()

返回操作系统的发行版本

os.uptime()

返回操作系统运行的时间，以秒为单位。

os.loadavg()

显示原文其他翻译纠错 返回一个包含 1、5、15 分钟平均负载的数组。

平均负载是系统的一个指标，操作系统计算，用一个很小的数字表示。理论上来说，平均负载最好比系统里的 CPU 低。

平均负载是一个非常 UNIX-y 的概念，windows 系统没有相同的概念。所以 windows 总是返回 `[0, 0, 0]` 。

os.totalmem()

返回系统内存总量，单位为字节。

os.freemem()

返回操作系统空闲内存量，单位是字节。

os.cpus()

返回一个对象数组，包含所安装的每个 CPU/内核的信息：型号、速度（单位 MHz）、时间（一个包含 user、nice、sys、idle 和 irq 所使用 CPU/内核毫秒数的对象）。

os.cpus 的例子:

```
[ { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 252020,
       nice: 0,
       sys: 30340,
       idle: 1070356870,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
```

```
  speed: 2926,
  times:
   { user: 306960,
     nice: 0,
     sys: 26980,
     idle: 1071569080,
     irq: 0 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 248450,
     nice: 0,
     sys: 21750,
     idle: 1070919370,
     irq: 0 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 256880,
     nice: 0,
     sys: 19430,
     idle: 1070905480,
     irq: 20 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 511580,
     nice: 20,
     sys: 40900,
     idle: 1070842510,
     irq: 0 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 291660,
     nice: 0,
     sys: 34360,
     idle: 1070888000,
     irq: 10 } },
{ model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
  speed: 2926,
  times:
   { user: 308260,
     nice: 0,
     sys: 55410,
```

```
      idle: 1071129970,
      irq: 880 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 266450,
       nice: 1480,
       sys: 34920,
       idle: 1072572010,
       irq: 30 } } ]
```

os.networkInterfaces()

获得网络接口列表：

```
{ lo:
   [ { address: '127.0.0.1',
       netmask: '255.0.0.0',
       family: 'IPv4',
       mac: '00:00:00:00:00:00',
       internal: true },
     { address: '::1',
       netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
       family: 'IPv6',
       mac: '00:00:00:00:00:00',
       internal: true } ],
  eth0:
   [ { address: '192.168.1.108',
       netmask: '255.255.255.0',
       family: 'IPv4',
       mac: '01:02:03:0a:0b:0c',
       internal: false },
     { address: 'fe80::a00:27ff:fe4e:66a1',
       netmask: 'ffff:ffff:ffff:ffff::',
       family: 'IPv6',
       mac: '01:02:03:0a:0b:0c',
       internal: false } ] }
```

os.EOL

定义了操作系统的 End-of-line 的常量。

20

路径

稳定性: 3 – 稳定

本模块包含一系列处理和转换文件路径的工具集。基本所有的反复都仅对字符串转换。文件系统不会检查路径是否有效。

通过 `require('path')` 来访问这个模块。提供了以下方法：

path.normalize(p)

规范化路径，注意 `'..'` 和 `'.'`。

发现多个斜杠时，会替换成一个斜杠。当路径末尾包含一个斜杠时，保留。Windows 系统使用反斜杠。

例如:

```
path.normalize('/foo/bar//baz/asdf/quux/..')
// returns
'/foo/bar/baz/asdf'
```

path.join([path1][, path2][, ...])

连接所有的参数，并规范化输出路径。

参数必须是字符串。在 v0.8 版本，非字符参数会被忽略。v0.10之后的版本后抛出异常。

例如:

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
// returns
'/foo/bar/baz/asdf'

path.join('foo', {}, 'bar')
// throws exception
TypeError: Arguments to path.join must be strings
```

path.resolve([from ...], to)

将 `to` 参数解析为绝对路径。

如果参数 `to` 不是一个相对于参数 `from` 的绝对路径， `to` 会添加到 `from` 右侧，直到找到一个绝对路径为止。如果使用所有 `from` 参数后，还是没有找到绝对路径，将会使用当前工作目录。返回的路径已经规范化过，并且去掉了尾部的斜杠（除非是根目录）。非字符串的参数会被忽略。

另一种思路就是在shell里执行一系列的 `cd` 命令。

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

类似于:

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

不同点是，不同的路径不需要存在的，也可能是文件。

例如:

```
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'
```

path.isAbsolute(path)

判断参数 `path` 是否是绝对路径。一个绝对路径解析后都会指向相同的位置，无论当前的工作目录在哪里。

Posix 例子:

```
path.isAbsolute('/foo/bar') // true
path.isAbsolute('/baz/..')  // true
path.isAbsolute('qux/')     // false
path.isAbsolute('.')        // false
```

Windows 例子:

```
path.isAbsolute('//server')  // true
path.isAbsolute('C:/foo/..') // true
path.isAbsolute('bar\\baz')   // false
path.isAbsolute('.')         // false
```

path.relative(from, to)

解决从 `from` 到 `to` 的相对路径。

有时我们会有2个绝对路径，需要从中找到相对目录。这是 `path.resolve` 的逆实现：

```
path.resolve(from, path.relative(from, to)) == path.resolve(to)
```

例如:

```
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'
```

path.dirname(p)

返回路径 `p` 所在的目录。和 Unix `dirname` 命令类似。

例如:

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns
'/foo/bar/baz/asdf'
```

path.basename(p[, ext])

返回路径的最后一个部分。和 Unix `basename` 命令类似。

例如:

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns
'quux.html'
```

```
path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns
'quux'
```

path.extname(p)

返回路径 `p` 的扩展名，从最后一个 '.' 到字符串的末尾。如果最后一个部分没有 '.' ，或者路径是以 '.' 开头，则返回空字符串。例如。

```
path.extname('index.html')
// returns
'.html'

path.extname('index.coffee.md')
// returns
'.md'

path.extname('index.')
// returns
'.'

path.extname('index')
// returns
''
```

path.sep

特定平台的文件分隔符， `'\\'` 或 `'/'` 。

*nix 上的例子:

```
'foo/bar/baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

Windows 的例子:

```
'foo\\bar\\baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

path.delimiter

特定平台的分隔符，`;` or `':'`.

*nix 上的例子:

```
console.log(process.env.PATH)
// '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env.PATH.split(path.delimiter)
// returns
['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

Windows 例子:

```
console.log(process.env.PATH)
// 'C:\Windows\system32;C:\Windows;C:\Program Files\nodejs\'

process.env.PATH.split(path.delimiter)
// returns
['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\nodejs\\']
```

path.parse(pathString)

返回路径字符串的对象。

*nix 上的例子:

```
path.parse('/home/user/dir/file.txt')
// returns
{
   root : "/",
   dir : "/home/user/dir",
   base : "file.txt",
   ext : ".txt",
   name : "file"
}
```

Windows 例子:

```
path.parse('C:\\path\\dir\\index.html')
// returns
{
```

```
    root : "C:\\",
    dir : "C:\\path\\dir",
    base : "index.html",
    ext : ".html",
    name : "index"
}
```

path.format(pathObject)

从对象中返回路径字符串，和 path.parse 相反。

```
path.format({
    root : "/",
    dir : "/home/user/dir",
    base : "file.txt",
    ext : ".txt",
    name : "file"
})
// returns
'/home/user/dir/file.txt'
```

path.posix

提供上述 path 路径访问，不过总是以 posix 兼容的方式交互。

path.win32

提供上述 path 路径访问，不过总是以 win32 兼容的方式交互。

21

进程

`process` 是全局对象，能够在任意位置访问，是 EventEmitter (events.html#events_class_events_eventemitter) 的实例。

退出状态码

当没有新的异步的操作等待处理时，Node 正常情况下退出时会返回状态码 `0` 。下面的状态码表示其他状态：

- `1` **未捕获的致命异常-Uncaught Fatal Exception** - 有未捕获异常，并且没有被域或 `uncaughtException` 处理函数处理。
- `2` - Unused (保留)
- `3` **JavaScript解析错误-Internal JavaScript Parse Error** - JavaScript的源码启动 Node 进程时引起解析错误。非常罕见，仅会在开发 Node 时才会有。
- `4` **JavaScript评估失败-Internal JavaScript Evaluation Failure** - JavaScript的源码启动 Node 进程，评估时返回函数失败。非常罕见，仅会在开发 Node 时才会有。
- `5` **致命错误-Fatal Error** - V8 里致命的不可恢复的错误。通常会打印到 stderr ，内容为： `FATAL ERROR`
- `6` **Non-function 异常处理-Non-function Internal Exception Handler** - 未捕获异常，内部异常处理函数不知为何设置为on-function，并且不能被调用。
- `7` **异常处理函数运行时失败-Internal Exception Handler Run-Time Failure** - 未捕获的异常， 并且异常处理函数处理时自己抛出了异常。例如，如果 `process.on('uncaughtException')` 或 `domain.on('error')` 抛出了异常。
- `8` - Unused保留. 之前版本的 Node， 8 有时表示未捕获异常。
- `9` - **参数非法-Invalid Argument** - 可能是给了未知的参数，或者给的参数没有值。
- `10` **运行时失败-Internal JavaScript Run-Time Failure** - JavaScript的源码启动 Node 进程时抛出错误，非常罕见，仅会在开发 Node 时才会有。
- `12` **无效的 Debug 参数-Invalid Debug Argument** - 设置了参数 `--debug` 和/或 `--debug-brk` ，但是选择了错误端口。
- `>128` **信号退出-Signal Exits** - 如果 Node 接收到致命信号，比如 `SIGKILL` 或 `SIGHUP` ，那么退出代码就是 `128` 加信号代码。这是标准的 Unix 做法，退出信号代码放在高位。

事件: 'exit'

当进程准备退出时触发。此时已经没有办法阻止从事件循环中推出。因此，你必须在处理函数中执行同步操作。这是一个在固定事件检查模块状态（比如单元测试）的好时机。回调函数有一个参数，它是进程的退出代码。

监听 `exit` 事件的例子:

```
process.on('exit', function(code) {
  // do *NOT* do this
  setTimeout(function() {
    console.log('This will not run');
  }, 0);
  console.log('About to exit with code:', code);
});
```

事件: 'beforeExit'

当 node 清空事件循环，并且没有其他安排时触发这个事件。通常来说，当没有进程安排时 node 退出，但是 'beforeExit' 的监听器可以异步调用，这样 node 就会继续执行。

'beforeExit' 并不是明确退出的条件， `process.exit()` 或异常捕获才是，所以不要把它当做'exit' 事件。除非你想安排更多的工作。

事件: 'uncaughtException'

当一个异常冒泡回到事件循环，触发这个事件。如果给异常添加了监视器，默认的操作（打印堆栈跟踪信息并退出）就不会发生。

监听 `uncaughtException` 的例子:

```
process.on('uncaughtException', function(err) {
  console.log('Caught exception: ' + err);
});

setTimeout(function() {
  console.log('This will still run.');
}, 500);
```

```
// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

注意， `uncaughtException` 是非常简略的异常处理机制。

尽量不要使用它，而应该用 domains () 。如果你用了，每次未处理异常后，重启你的程序。

不要使用 node.js 里诸如 `On Error Resume Next` 这样操作。每个未处理的异常意味着你的程序，和你的 node.js 扩展程序，一个未知状态。盲目的恢复意味着 *任何事情* 都可能发生

你在升级的系统时拉掉了电源线，然后恢复了。可能10次里有9次没有问题，但是第10次，你的系统可能就会挂掉。

Signal 事件

当进程接收到信号时就触发。信号列表详见标准的 POSIX 信号名，如 SIGINT、SIGUSR1 等

监听 `SIGINT` 的例子:

```
// Start reading from stdin so we don't exit.
process.stdin.resume();

process.on('SIGINT', function() {
  console.log('Got SIGINT.  Press Control-D to exit.');
});
```

在大多数终端程序里，发送 `SIGINT` 信号的简单方法是按下 `信号Control-C` 。

注意:

- `SIGUSR1` node.js 接收这个信号开启调试模式。可以安装一个监听器，但开始时不会中断调试。
- `SIGTERM` 和 `SIGINT` 在非 Windows 系统里，有默认的处理函数，退出（伴随退出代码 `128 + 信号码` ）前，重置退出模式。如果这些信号有监视器，默认的行为将会被移除。
- `SIGPIPE` 默认情况下忽略，可以加监听器。
- `SIGHUP` 当 Windowns 控制台关闭的时候生成，其他平台的类似条件，参见signal(7)。可以添加监听者，Windows 平台上 10 秒后会无条件退出。在非 Windows 平台上， `SIGHUP` 的默认操作是终止 node，但是一旦添加了监听器，默认动作将会被移除。 `SIGHUP` is to terminate node, but once a listener has been installed its
- `SIGTERM` Windows 不支持, 可以被监听。

- `SIGINT` 所有的终端都支持，通常由 `CTRL+C` 生成（可能需要配置）。当终端原始模式启用后不会再生成。
- `SIGBREAK` Windows 里，按下 `CTRL+BREAK` 会发送。非 Windows 平台，可以被监听，但是不能发送或生成。
- `SIGWINCH` - 当控制台被重设大小时发送。Windows 系统里，仅会在控制台上输入内容时，光标移动，或者可读的 tty在原始模式上使用。
- `SIGKILL` 不能有监视器，在所有平台上无条件关闭 node。
- `SIGSTOP` 不能有监视器。

Windows 不支持发送信号，但是 node 提供了很多 `process.kill()` 和 `child_process.kill()` 的模拟： - 发送 Sending 信号 `0` 可以查找运行中得进程 - 发送 `SIGINT`, `SIGTERM`, 和 `SIGKILL` 会引起目标进程无条件退出。

process.stdout

一个 `Writable Stream` 执向 `stdout` (on fd `1`).

例如: `console.log` 的定义:

```
console.log = function(d) {
  process.stdout.write(d + '\n');
};
```

`process.stderr` 和 `process.stdout` 和 node 里的其他流不同，他们不会被关闭（ `end()` 将会被抛出），它们不会触发 `finish` 事件，并且写是阻塞的。

- 引用指向常规文件或 TTY 文件描述符时，是阻塞的。
- 引用指向 pipe 管道时:
    - 在 Linux/Unix 里阻塞.
    - 在 Windows 像其他流一样，不被阻塞

检查 Node 是否运行在 TTY 上下文中，从 `process.stderr`, `process.stdout`, 或 `process.stdin` 里读取 `isTTY` 属性。

```
$ node -p "Boolean(process.stdin.isTTY)"
true
$ echo "foo" | node -p "Boolean(process.stdin.isTTY)"
```

```
false

$ node -p "Boolean(process.stdout.isTTY)"
true
$ node -p "Boolean(process.stdout.isTTY)" | cat
false
```

更多信息参见 the tty docs (页 0)。

process.stderr

一个指向 stderr (on fd `2`)的可写流。

`process.stderr` 和 `process.stdout` 和 node 里的其他流不同，他们不会被关闭（ `end()` 将会被抛出），它们不会触发 `finish` 事件，并且写是阻塞的。

- 引用指向常规文件或 TTY 文件描述符时，是阻塞的。
- 引用指向 pipe 管道时:
    - 在 Linux/Unix 里阻塞.
    - 在 Windows 像其他流一样，不被阻塞

process.stdin

一个指向 stdin (on fd `0`)的可读流。

以下例子：打开标准输入流，并监听两个事件:

```
process.stdin.setEncoding('utf8');

process.stdin.on('readable', function() {
  var chunk = process.stdin.read();
  if (chunk !== null) {
    process.stdout.write('data: ' + chunk);
  }
});

process.stdin.on('end', function() {
  process.stdout.write('end');
});
```

`process.stdin` 可以工作在老模式里，和 v0.10 之前版本的 node 代码兼容。

更多信息参见Stream compatibility (页 0).

在老的流模式里，stdin流默认暂停，必须调用 process.stdin.resume() 读取。可以调用 process.stdin.resume() 切换到老的模式。

如果开始一个新的工程，最好选择新的流，而不是用老的流。

process.argv

包含命令行参数的数组。第一个元素是'node'，第二个参数是 JavaScript 文件的名字，第三个参数是任意的命令行参数。

```
// print process.argv
process.argv.forEach(function(val, index, array) {
  console.log(index + ': ' + val);
});
```

将会生成:

```
$ node process-2.js one two=three four
0: node
1: /Users/mjr/work/node/process-2.js
2: one
3: two=three
4: four
```

process.execPath

开启当前进程的执行文件的绝对路径。

例子:

```
/usr/local/bin/node
```

process.execArgv

启动进程所需的 node 命令行参数。这些参数不会在 process.argv 里出现，并且不包含 node 执行文件的名字，或者任何在名字之后的参数。这些用来生成子进程，使之拥有和父进程有相同的参数。

例子:

```
$ node --harmony script.js --version
```

process.execArgv 的参数:

```
['--harmony']
```

process.argv 的参数:

```
['/usr/local/bin/node', 'script.js', '--version']
```

process.abort()

这将导致 node 触发 abort 事件。会让 node 退出并生成一个核心文件。

process.chdir(directory)

改变当前工作进程的目录，如果操作失败抛出异常。

```
console.log('Starting directory: ' + process.cwd());
try {
  process.chdir('/tmp');
  console.log('New directory: ' + process.cwd());
}
catch (err) {
  console.log('chdir: ' + err);
}
```

process.cwd()

返回当前进程的工作目录

```
console.log('Current directory: ' + process.cwd());
```

process.env

包含用户环境的对象，参见 environ(7).

这个对象的例子:

```
{ TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node' }
```

你可以写入这个对象，但是不会改变当前运行的进程。以下的命令不会成功：

```
node -e 'process.env.foo = "bar"' && echo $foo
```

这个会成功:

```
process.env.foo = 'bar';
console.log(process.env.foo);
```

process.exit([code])

使用指定的 code 结束进程。如果忽略，将会使用 code `0`

使用失败的代码退出：

```
process.exit(1);
```

Shell 将会看到退出代码为1.

process.exitCode

进程退出时的代码，如果进程优雅的退出，或者通过 `process.exit()` 退出，不需要指定退出码。

设定 `process.exit(code)` 将会重写之前设置的 `process.exitCode` 。

process.getgid()

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

获取进程的群组标识（参见 getgid(2)）。获取到得时群组的数字 id，而不是名字。

```
if (process.getgid) {
  console.log('Current gid: ' + process.getgid());
}
```

process.setgid(id)

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

设置进程的群组标识（参见 setgid(2)）。可以接收数字 ID 或者群组名。如果指定了群组名，会阻塞等待解析为数字 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.getuid()

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

获取进程的用户标识(参见 getuid(2))。这是数字的用户 id，不是用户名

```
if (process.getuid) {
  console.log('Current uid: ' + process.getuid());
}
```

process.setuid(id)

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

设置进程的用户标识（参见setuid(2)）。接收数字 ID或字符串名字。果指定了群组名，会阻塞等待解析为数字 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.getgroups()

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

返回进程的群组 iD 数组。POSIX 系统没有保证一定有，但是 node.js 保证有。

process.setgroups(groups)

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

设置进程的群组 ID。这是授权操作，所有你需要有 root 权限，或者有 CAP_SETGID 能力。

列表可以包含群组 IDs，群组名，或者两者都有。

process.initgroups(user, extra_group)

注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。

读取 /etc/group ，并初始化群组访问列表，使用成员所在的所有群组。这是授权操作，所有你需要有 root 权限，或者有 CAP_SETGID 能力。

`user` 是用户名或者用户 ID， `extra_group` 是群组名或群组 ID。

当你在注销权限 (dropping privileges) 的时候需要注意. 例子:

```
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.version

一个编译属性，包含 `NODE_VERSION` .

```
console.log('Version: ' + process.version);
```

process.versions

一个属性，包含了 node 的版本和依赖.

```
console.log(process.versions);
```

打印出来:

```
{ http_parser: '1.0',
  node: '0.10.4',
  v8: '3.14.5.8',
  ares: '1.9.0-DEV',
  uv: '0.10.3',
  zlib: '1.2.3',
  modules: '11',
  openssl: '1.0.1e' }
```

process.config

一个包含用来编译当前 node 执行文件的 javascript 配置选项的对象。它与运行 ./configure 脚本生成的 "config.gypi" 文件相同。

一个可能的输出:

```
{ 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_v8: '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.kill(pid[, signal])

发送信号给进程. `pid` 是进程id，并且 `signal` 是发送的信号的字符串描述。信号名是字符串，比如 'SIGINT' 或 'SIGHUP'。如果忽略，信号会是 'SIGTERM'.更多信息参见 Signal 事件 (页 0) 和 kill(2) .

如果进程没有退出，会抛出错误。信号 `0` 可以用来测试进程是否存在。

注意，虽然这个这个函数名叫 `process.kill` ，它真的仅是信号发射器，就像 `kill` 系统调用。信号发射可以做其他事情，不仅是杀死目标进程。

例子， 给自己发信号:

```
process.on('SIGHUP', function() {
  console.log('Got SIGHUP signal.');
});

setTimeout(function() {
  console.log('Exiting.');
  process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

注意: 当 Node.js 接收到 SIGUSR1 信号，它会开启 debugger 调试模式, 参见 Signal Events (页 0).

process.pid

当前进程的 PID

```
console.log('This process is pid ' + process.pid);
```

process.title

获取/设置(Getter/setter) 'ps' 中显示的进程名。

使用 setter 时，字符串的长度由系统指定，可能会很短。

在 Linux 和 OS X 上，它受限于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(argv memory)。

v0.8 版本允许更长的进程标题字符串，也支持覆盖环境内存，但是存在潜在的不安全和混乱（很难说清楚）。

process.arch

当前 CPU 的架构：'arm'、'ia32' 或者 'x64'.

```
console.log('This processor architecture is ' + process.arch);
```

process.platform

运行程序所在的平台系统 `'darwin'`, `'freebsd'`, `'linux'`, `'sunos'` or `'win32'`

```
console.log('This platform is ' + process.platform);
```

process.memoryUsage()

返回一个对象，描述了 Node 进程所用的内存状况，单位为字节。

```
var util = require('util');

console.log(util.inspect(process.memoryUsage()));
```

将会生成:

```
{ rss: 4935680,
  heapTotal: 1826816,
  heapUsed: 650472 }
```

`heapTotal` and `heapUsed` refer to V8's memory usage.

process.nextTick(callback)

- `callback` {Function}

一旦当前事件循环结束，调用回到函数。

这不是 `setTimeout(fn, 0)` 的简单别名，这个效率更高。在任何附加 I/O 事件在子队列事件循环中触发前，它就会运行。

```
console.log('start');
process.nextTick(function() {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

在对象构造后，在 I/O 事件发生前，你又想改变附加事件处理函数时，这个非常有用。

```
function MyThing(options) {
  this.setupOptions(options);

  process.nextTick(function() {
    this.startDoingStuff();
  }.bind(this));
}

var thing = new MyThing();
thing.getReadyForStuff();

// thing.startDoingStuff() gets called now, not before.
```

要保证你的函数一定是 100% 同步执行，或者 100% 异步执行。例子:

```
// WARNING!  DO NOT USE!  BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
  if (arg) {
    cb();
    return;
  }
```

```
  fs.stat('file', cb);
}
```

这个 API 非常危险. 如果你这么做:

```
maybeSync(true, function() {
  foo();
});
bar();
```

不清楚 `foo()` 或 `bar()` 哪个先执行。

更好的方法:

```
function definitelyAsync(arg, cb) {
  if (arg) {
    process.nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

注意：nextTick 队列会在完全执行完毕之后才调用 I/O 操作。因此，递归设置 nextTick 的回调就像一个 `while(true);` 循环一样，将会阻止任何 I/O 操作的发生。

process.umask([mask])

设置或读取进程文件的掩码。子进程从父进程继承掩码。如果 `mask` 参数有效，返回旧的掩码。否则，返回当前掩码。

```
var oldmask, newmask = 0022;

oldmask = process.umask(newmask);
console.log('Changed umask from: ' + oldmask.toString(8) +
            ' to ' + newmask.toString(8));
```

process.uptime()

返回 Node 已经运行的秒数。

process.hrtime()

返回当前进程的高分辨时间，形式为 [seconds, nanoseconds] 数组。它是相对于过去的任意事件。该值与日期无关，因此不受时钟漂移的影响。主要用途是可以通过精确的时间间隔，来衡量程序的性能。

你可以将之前的结果传递给当前的 process.hrtime() ，会返回两者间的时间差，用来基准和测量时间间隔。

```
var time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(function() {
  var diff = process.hrtime(time);
  // [ 1, 552 ]

  console.log('benchmark took %d nanoseconds', diff[0] * 1e9 + diff[1]);
  // benchmark took 1000000527 nanoseconds
}, 1000);
```

process.mainModule

require.main (页 0) 的备选方法。不同点，如果主模块在运行时改变， require.main 可能会继续返回老的模块。可以认为，这两者引用了同一个模块。

Alternate way to retrieve require.main (页 0). The difference is that if the main module changes at run time, require.main might still refer to the original main module in modules that were required before the change occurred. Generally it's safe to assume that the two refer to the same module.

和 require.main 一样, 如果没有入口脚本，将会返回 undefined 。

22

Punycode

稳定性: 2 - 不稳定

Punycode.js (http://mths.be/punycode) 从 Node.js v0.6.2+ 开始内置. 使用 `require('punycode')` 来访问。(要在其他 Node.js 版本中访问,先用 npm 来 `punycode` 安装)。

punycode.decode(string)

将一个纯 ASCII 的 Punycode 字符串转换为 Unicode 字符串。

```
// decode domain name parts
punycode.decode('maana-pta'); // 'mañana'
punycode.decode('--dqo34k'); // '☃-⌘'
```

punycode.encode(string)

将一个纯 Unicode Punycode 字符串转换为 纯 ASCII 字符串。

```
// encode domain name parts
punycode.encode('mañana'); // 'maana-pta'
punycode.encode('☃-⌘'); // '--dqo34k'
```

punycode.toUnicode(domain)

将一个表示域名的 Punycode 字符串转换为 Unicode。只有域名中的 Punycode 部分会被转换，也就是说你也可以在一个已经转换为 Unicode 的字符串上调用它。

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn----dqo34k.com'); // '☃-⌘.com'
```

punycode.toASCII(domain)

将一个表示域名的 Unicode 字符串转换为 Punycode。只有域名中的非 ASCII 部分会被转换，也就是说你也可以在一个已经转换为 ASCII 的字符串上调用它。

```
// encode domain names
punycode.toASCII('mañana.com'); // 'xn--maana-pta.com'
punycode.toASCII('☃-⌘.com'); // 'xn----dqo34k.com'
```

punycode.ucs2

punycode.ucs2.decode(string)

创建一个包含字符串中每个 Unicode 符号的数字编码点的数组，。由于 JavaScript uses UCS-2 internally (http://mathiasbynens.be/notes/javascript-encoding) 在内部使用 UCS-2， 该函数会按照 UTF-16 将一对代半数（UCS-2 暴露的单独的字符）转换为单独一个编码点。

```
punycode.ucs2.decode('abc'); // [0x61, 0x62, 0x63]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

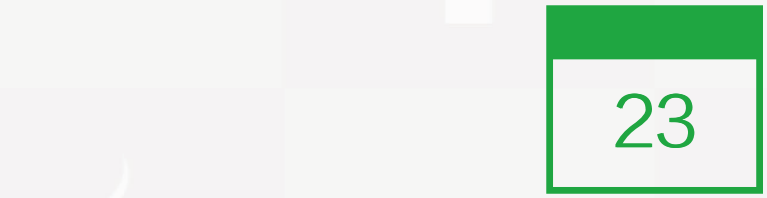
punycode.ucs2.encode(codePoints)

创建以一组数字编码点为基础一个字符串。

```
punycode.ucs2.encode([0x61, 0x62, 0x63]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.version

表示当前 Punycode.js 版本数字的字符串。



23

Query String

稳定性: 3 – 稳定

这个模块提供了一些处理 query strings 的工具，包括以下方法：

querystring.stringify(obj[, sep][, eq][, options])

将一个对象序列化化为一个 query string 。

可以选择重写默认的分隔符('&') 和分配符 ('=')。

Options 对象可能包含 encodeURIComponent 属性 (默认： querystring.escape),如果需要，它可以用 non-utf8 编码字符串。

例子:

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })
// returns
'foo=bar&baz=qux&baz=quux&corge='

querystring.stringify({foo: 'bar', baz: 'qux'}, ';', ':')
// returns
'foo:bar;baz:qux'

// Suppose gbkEncodeURIComponent function already exists,
// it can encode string with `gbk` encoding
querystring.stringify({ w: '中文', foo: 'bar' }, null, null,
  { encodeURIComponent: gbkEncodeURIComponent })
// returns
'w=%D6%D0%CE%C4&foo=bar'
```

querystring.parse(str[, sep][, eq][, options])

将 query string 反序列化为对象。

可以选择重写默认的分隔符('&') 和分配符 ('=')。

Options 对象可能包含 maxKeys 属性（默认：1000），用来限制处理过的健值（keys）。设置为 0 的话，可以去掉键值的数量限制。

Options 对象可能包含 decodeURIComponent 属性（默认： querystring.unescape ），如果需要，可以用来解码 non-utf8 编码的字符串。

例子:

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge')
// returns
{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }

// Suppose gbkDecodeURIComponent function already exists,
// it can decode `gbk` encoding string
querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null,
  { decodeURIComponent: gbkDecodeURIComponent })
// returns
{ w: '中文', foo: 'bar' }
```

querystring.escape

escape 函数供 `querystring.stringify` 使用，必要时，可以重写。

querystring.unescape

unescape函数供 `querystring.parse` 使用。必要时，可以重写。

首先会尝试用 `decodeURIComponent` ，如果失败，会回退，不会抛出格式不正确的 URLs。

24

逐行读取

```
稳定性: 2 - 不稳定
```

使用 `require('readline')`，可以使用这个模块。逐行读取（Readline）可以逐行读取流（比如 `process.stdin` ）

一旦你开启了这个模块，node 程序将不会终止，直到你关闭接口。以下的代码展示了如何优雅的退出程序：

```
var readline = require('readline');

var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question("What do you think of node.js? ", function(answer) {
  // TODO: Log the answer in a database
  console.log("Thank you for your valuable feedback:", answer);

  rl.close();
});
```

readline.createInterface(options)

创建一个逐行读取（Readline） `Interface` 实例. 参数 "options" 对象有以下值:

- `input` - 监听的可读流 (必填).
- `output` - 逐行读取（Readline）数据要写入的可写流(可选).
- `completer` - 用于 Tab 自动补全的可选函数。参见下面的例子。
- `terminal` - 如果希望和 TTY 一样，对待 `input` 和 `output` 流，设置为 true。 并且由 ANSI/VT100 转码。默认情况下，检查 `isTTY` 是否在 `output` 流上实例化。

`completer` 给出当前行的入口，应该返回包含2条记录的数组

1. 一个匹配当前输入补全的字符串数组
2. 用来匹配的子字符串

最终像这样: `[[substr1, substr2, ...], originalsubstring]` .

例子：

```
function completer(line) {
  var completions = '.help .error .exit .quit .q'.split(' ')
```

```
  var hits = completions.filter(function(c) { return c.indexOf(line) == 0 })
  // show all completions if none found
  return [hits.length ? hits : completions, line]
}
```

同时， `completer` 可以异步运行，此时接收到2个参数:

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```

为了接受用户输入， `createInterface` 通常和 `process.stdin` ， `process.stdout` 一起使用:

```
var readline = require('readline');
var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

如果你有逐行读取（Readline）实例, 通常会监听 `"line"` 事件.

如果这个实例参数 `terminal` = `true` ，而且定义了 `output.columns` 属性，那么 `output` 流将会最佳兼容性，并且，当 columns 变化时（当它是 TTY 时， `process.stdout` 会自动这么做），会在 `output` 流上触发 `"resize"` 事件。

Class: Interface

代表一个包含输入/输出流的逐行读取（Readline）接口的类，

rl.setPrompt(prompt)

设置提示符，比如当你再命令行里运行 `node` 时，可以看到 node 的提示符 `>` 。

rl.prompt([preserveCursor])

为用户输入准备好逐行读取（Readline），将当前 `setPrompt` 选项方法哦新的行中，让用户有新的地方输入。设置 `preserveCursor` 为 `true` ，防止当前的游标重置为 `0` 。

如果暂停，使用 `createInterface` 也可以重置 `input` 输入流。

调用 `createInterface` 时，如果 `output` 设置为 `null` 或 `undefined` ，不会重新写提示符。

rl.question(query, callback)

预先提示 `query`，用户应答后触发 `callback`。给用户显示 query 后，用户应答被输入后，调用 `callback`。

如果暂停，使用 `createInterface` 也可以重置 `input` 输入流。

调用 `createInterface` 时，如果 `output` 设置为 `null` 或 `undefined`，不会重新写提示符。

例子:

```
interface.question('What is your favorite food?', function(answer) {
  console.log('Oh, so your favorite food is ' + answer);
});
```

rl.pause()

暂停逐行读取（Readline）的 `input` 输入流, 如果需要可以重新启动。

注意，这不会立即暂停流。调用 `pause` 后还会有很多事件触发，包含 `line`。

rl.resume()

恢复 逐行读取（Readline） `input` 输入流.

rl.close()

关闭 `Interface` 实例, 放弃控制输入输出流。会触发"close" 事件。

rl.write(data[, key])

调用 `createInterface` 后，将数据 `data` 写到 `output` 输出流，除非 `output` 为 `null`，或未定义 `undefined`。`key` 是一个代表键序列的对象；当终端是一个 TTY 时可用。

暂停 `input` 输入流后，这个方法可以恢复。

例子:

```
rl.write('Delete me!');
// Simulate ctrl+u to delete the line written previously
rl.write(null, {ctrl: true, name: 'u'});
```

Events

事件: 'line'

`function (line) {}`

`input` 输入流收到 `\n` 后触发，通常因为用户敲回车或返回键。这是监听用户输入的好办法。

监听 `line` 的例子:

```
rl.on('line', function (cmd) {
  console.log('You just typed: '+cmd);
});
```

事件: 'pause'

`function () {}`

暂停 `input` 输入流后，会触发这个方法。

当输入流未被暂停，但收到 `SIGCONT` 也会触发。(详见 `SIGTSTP` 和 `SIGCONT` 事件)

监听 `pause` 的例子:

```
rl.on('pause', function() {
  console.log('Readline paused.');
});
```

事件: 'resume'

`function () {}`

恢复 `input` 输入流后，会触发这个方法。

监听 `resume` 的例子:

```
rl.on('resume', function() {
  console.log('Readline resumed.');
});
```

事件: 'close'

`function () {}`

调用 `close()` 方法时会触发。

当 `input` 输入流收到 "end" 事件时会触发。一旦触发，可以认为 `Interface` 实例结束。例如当 `input` 输入流收到 `^D`，被当做 `EOT`。

如果没有 `SIGINT` 事件监听器，当 `input` 输入流接收到 `^C`（被当做 `SIGINT`），也会触发这个事件。

事件: 'SIGINT'

`function () {}`

当 `input` 输入流收到 `^C` 时会触发， 被当做 `SIGINT`。如果没有 `SIGINT` 事件监听器，当 `input` 输入流接收到 `SIGINT`（被当做 `SIGINT`），会触发 `pause` 事件。

监听 `SIGINT` 的例子:

```
rl.on('SIGINT', function() {
  rl.question('Are you sure you want to exit?', function(answer) {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

事件: 'SIGTSTP'

`function () {}`

Windows 里不可用

当 `input` 输入流收到 `^Z` 时会触发，被当做 `SIGTSTP`。如果没有 `SIGINT` 事件监听器，当 `input` 输入流接收到 `SIGTSTP`，程序将会切换到后台。

当程序通过 `fg` 恢复，将会触发 `pause` 和 `SIGCONT` 事件。你可以使用两者中任一事件来恢复流。

程切换到后台前，如果暂停了流， `pause` 和 `SIGCONT` 事件不会被触发。

监听 `SIGTSTP` 的例子:

```
rl.on('SIGTSTP', function() {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

事件: 'SIGCONT'

`function () {}`

Windows 里不可用

一旦 input 流中含有 ^Z并被切换到后台就会触发。被当做 `SIGTSTP` ，然后继续执行 `fg(1)` 。程切换到后台前，如果流没被暂停，这个事件可以被触发。

监听 `SIGCONT` 的例子:

```
rl.on('SIGCONT', function() {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

例子： Tiny CLI

以下的例子，展示了如何所有这些方法的命令行接口:

```
var readline = require('readline'),
    rl = readline.createInterface(process.stdin, process.stdout);

rl.setPrompt('OHAI> ');
rl.prompt();

rl.on('line', function(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', function() {
```

```
  console.log('Have a great day!');
  process.exit(0);
});
```

readline.cursorTo(stream, x, y)

在TTY 流里，移动光标到指定位置。

readline.moveCursor(stream, dx, dy)

在TTY 流里，移动光标到当前位置的相对位置。

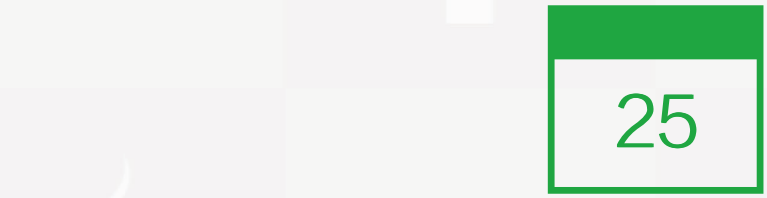
readline.clearLine(stream, dir)

清空 TTY 流里指定方向的行。`dir` 是以下值：

- `-1` - 从光标到左边
- `1` - 从光标到右边
- `0` - 整行

readline.clearScreenDown(stream)

清空屏幕上从当前光标位置起的内容。



25

REPL

```
稳定性: 3 – 稳定
```

Read-Eval-Print-Loop (REPL 读取-执行-输出循环)即可作为独立程序，也可以集成到其他程序中。REPL 提供了一种交互的执行 JavaScript 并查看输出结果的方法。可以用来调试，测试，或仅是用来试试。

在命令行中不带任何参数的执行 `node` ，就是 REPL 模式。它提供了简单的 emacs 行编辑。

```
mjr:~$ node
Type '.help' for options.
> a = [ 1, 2, 3];
[ 1, 2, 3 ]
> a.forEach(function (v) {
...   console.log(v);
...   });
1
2
3
```

若想使用高级的编辑模式，使用环境变量 `NODE_NO_READLINE=1` 打开 node。这样会开启 REPL 模式，允许你使用 `rlwrap` 。

例如，你可以添加以下代码到你的 bashrc 文件里。

```
alias node="env NODE_NO_READLINE=1 rlwrap node"
```

repl.start(options)

启动并返回一个 `REPLServer` 实例。它继承自[Readline Interface][]。接收的参数 "options" 有以下值:

- `prompt` – 所有输入输出的提示符和流。默认是 `>` .
- `input` – 需要监听的可读流，默认 `process.stdin` .
- `output` – 用来输出数据的可写流，默认为 `process.stdout` .
- `terminal` – 如果 `stream` 被当成 TTY，并且有 ANSI/VT100 转义， 传 `true` 。默认在实例的输出流上检查 `isTTY` 。
- `eval` – 用来对每一行进行求值的函数。默认为 `eval()` 的异步封装。参见后面的自定义 `eval` 例子。
- `useColors` – 写函数输出是否有颜色。如果设定了不同的 `writer` 函数则无效。默认为 repl 的 `terminal` 值。

- `useGlobal` - 如果为 `true` ，则 repl 将会使用全局对象，而不是在独立的上下文中运行scripts。默认为 `false` 。
- `ignoreUndefined` - 如果为 `true` ，repl 不会输出未定义命令的返回值。默认为 `false` 。
- `writer` - 每个命令行被求值时都会调用这个函数，它会返回格式化显示内容（包括颜色）。默认是 `util.inspect` 。

如果有以下特性，可以使用自己的 `eval` 函数：

```
function eval(cmd, context, filename, callback) {
  callback(null, result);
}
```

在同一个 node 的运行实例上，可以打开多个 REPLs。每个都会共享一个全局对象，但会有独立的 I/O。

以下的例子，在stdin, Unix socket, 和 TCP socket 上开启 REPL：

```
var net = require("net"),
    repl = require("repl");

connections = 0;

repl.start({
  prompt: "node via stdin> ",
  input: process.stdin,
  output: process.stdout
});

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via Unix socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  })
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via TCP socket> ",
    input: socket,
```

```
    output: socket
  }).on('exit', function() {
    socket.end();
  });
}).listen(5001);
```

从命令行运行这个程序，将会在 stdin 上启动 REPL。其他的 REPL 客户端可能通过 Unix socket 或 TCP socket 连接。`telnet` 常用于连接 TCP socket， `socat` 用于连接Unix 和 TCP sockets

从Unix socket-based 服务器启动 REPL（而非stdin），你可以建立长连接，不用重启它们。

通过 `net.Server` 和 `net.Socket` 实例运行"full-featured" (`terminal`) REPL的例子，参见: https://gist.github.com/2209310

通过 `curl(1)` 实例运行 REPL的例子，参见: https://gist.github.com/2053342

Event: 'exit'

`function () {}`

当用户通过预定义的方式退出 REPL 将会触发这个事件。预定义的方式包括，在 repl 里输入 `.exit`，按 Ctrl+C 两次来发送 SIGINT 信号，或者在 `input` 流上按 Ctrl+D 来发送"end"。

监听 `exit` 的例子:

```
r.on('exit', function () {
  console.log('Got "exit" event from repl!');
  process.exit();
});
```

Event: 'reset'

`function (context) {}`

重置 REPL 的上下文的时候触发。当你输入 `.clear` 会重置。如果你用 `{ useGlobal: true }` 启动 repl，那这个事件永远不会被触发。

监听 `reset` 的例子:

```
// Extend the initial repl context.
r = repl.start({ options ... });
someExtension.extend(r.context);
```

```
// When a new context is created extend it as well.
r.on('reset', function (context) {
  console.log('repl has a new context');
  someExtension.extend(context);
});
```

REPL 特性

在 REPL 里， Control+D 会退出。可以输入多行表达式。支持全局变量和局部变量的 TAB 自动补全。

特殊变量 _ (下划线)包含上一个表达式的结果。

```
> [ "a", "b", "c" ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
4
```

REPL支持在全局域里访问任何变量。将变量赋值个和 REPLServer 关联的上下文对象，你可以显示的讲变量暴露给 REPL，例如：

```
// repl_test.js
var repl = require("repl"),
    msg = "message";

repl.start("> ").context.m = msg;
```

context 对象里的东西，会以局部变量的形式出现：

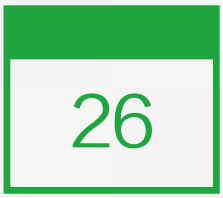
```
mjr:~$ node repl_test.js
> m
'message'
```

有一些特殊的REPL命令：

- .break – 当你输入多行表达式时，也许你走神了或者不想完成了， .break 可以重新开始。
- .clear – 重置 context 对象为空对象，并且清空多行表达式。
- .exit – 关闭输入/输出流，会让 REPL 退出。
- .help – 打印这些特殊命令。
- .save – 保存当前 REPL 会话到文件。 >.save ./file/to/save.js
- .load – 加载一个文件到当前REPL 会话 >.load ./file/to/load.js

下面的组合键在 REPL 中有以下效果：

- `<ctrl>C` – 和 `.break` 键类似. 在一个空行连按两次会强制退出。
- `<ctrl>D` – 和 `.exit` 键类似。



26

Smalloc

稳定性: 1 – 试验

类: smalloc

由简单内存分配器(处理扩展原始内存的分配)支持的缓存。Smalloc 有以下函数:

smalloc.alloc(length[, receiver][, type])

- `length` {Number} `<= smalloc.kMaxLength`
- `receiver` {Object} 默认: `new Object`
- `type` {Enum} 默认: `Uint8`

返回 `receiver` 对象，包含分配的外部数组数据。如果没有传入 `receiver` ，将会创建并返回一个新的对象。

这可用于创建你自己的类似 buffer 的类。不会设置其他属性，因此使用者需要跟踪其他所需信息（比如分配的长度）。

```
function SimpleData(n) {
  this.length = n;
  smalloc.alloc(this.length, this);
}

SimpleData.prototype = { /* ... */ };
```

仅检查 `receiver` 是否是非数组的对象。因此，可以分配扩展数据数据，不仅是普通对象。

```
function allocMe() { }
smalloc.alloc(3, allocMe);

// { [Function allocMe] '0': 0, '1': 0, '2': 0 }
```

v8 不支持给数组分配扩展数组对象，如果这么做，将会抛出。

你可以指定外部数组数据的类型。所有可用类型在 `smalloc.Types` 列出，例如:

```
var doubleArr = smalloc.alloc(3, smalloc.Types.Double);

for (var i = 0; i < 3; i++)
  doubleArr = i / 10;

// { '0': 0, '1': 0.1, '2': 0.2 }
```

使用 `Object.freeze`，`Object.seal` 和 `Object.preventExtensions` 不能冻结，封锁，阻止对象的使用扩展数据扩展。

smalloc.copyOnto(source, sourceStart, dest, destStart, copyLength);

- `source` {Object} 分配了外部数组的对象
- `sourceStart` {Number} 负责的起始位置
- `dest` {Object} 分配了外部数组的对象
- `destStart` {Number} 拷贝到目标的起始位置
- `copyLength` {Number} 需要拷贝的长度

从一个外部数组拷贝内存到另外一个，所有的参数都必填，否则会抛出异常。

```
var a = smalloc.alloc(4);
var b = smalloc.alloc(4);

for (var i = 0; i < 4; i++) {
  a[i] = i;
  b[i] = i * 2;
}

// { '0': 0, '1': 1, '2': 2, '3': 3 }
// { '0': 0, '1': 2, '2': 4, '3': 6 }

smalloc.copyOnto(b, 2, a, 0, 2);

// { '0': 4, '1': 6, '2': 2, '3': 3 }
```

`copyOnto` 会在内部自动检测分配的长度，因此不必设置任何附加参数。

smalloc.dispose(obj)

- `obj` Object

释放通过 `smalloc.alloc` 给对象分配的内存。

```
var a = {};
smalloc.alloc(3, a);

// { '0': 0, '1': 0, '2': 0 }
```

```
smalloc.dispose(a);

// {}
```

有利于减轻垃圾回收器的负担，但是开发时候还是要小心。程序里可能会出现难以跟踪的错误。

```
var a = smalloc.alloc(4);
var b = smalloc.alloc(4);

// perform this somewhere along the line
smalloc.dispose(b);

// now trying to copy some data out
smalloc.copyOnto(b, 2, a, 0, 2);

// now results in:
// RangeError: copy_length > source_length
```

调用 `dispose()` ，对象依旧拥有外部数据，例如 `smalloc.hasExternalData()` 会返回 `true` 。`dispose()` 不支持缓存，如果传入将会抛出。

smalloc.hasExternalData(obj)

- `obj` {Object}

如果 `obj` 拥有外部分配的内存，返回 `true` 。

smalloc.kMaxLength

可分配的最大数量。同样适用于缓存创建。

smalloc.Types

外部数组的类型，包含：

- `Int8`
- `Uint8`
- `Int16`
- `Uint16`

- Int32
- Uint32
- Float
- Double
- Uint8Clamped

27

流

稳定性: 2 – 不稳定

流是一个抽象接口，在 Node 里被不同的对象实现。例如request to an HTTP server (页 0) 是流，stdout (process.html#process_process_stdout) 是流。流是可读，可写，或者可读写。所有的流是 EventEmitter (events.html#events_class_events_eventemitter) 的实例。

你可以通过 `require('stream')` 加载 Stream 基类。其中包括了 `Readable` 流、`Writable` 流、`Duplex` 流和 `Transform` 流的基类。

这个文档分为 3 个章节。第一个章节解释了在你的程序中使用流时候需要了解的部分。如果你不用实现流式 API，可以只看这个章节。

如果你想实现你自己的流，第二个章节解释了这部分 API。这些 API 让你的实现更加简单。

第三个部分深入的解释了流是如何工作的，包括一些内部机制和函数，这些内容不要改动，除非你明确知道你要做什么。

面向流消费者的 API

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

所有的流都是事件分发器（EventEmitters），但是也有自己的方法和属性，这取决于他它们是可读（Readable），可写（Writable），或者兼具两者（Duplex，双工）的。

如果流式可读写的，则它实现了下面的所有方法和事件。因此，这个章节 API 完全阐述了Duplex (页 0) 或 Transform (页 0) 流，即便他们的实现有所不同。

没有必要为了消费流而在你的程序里实现流的接口。如果你正在你的程序里实现流接口，请同时参考下面的API for Stream Implementors (页 0)。

基本所有的 Node 程序，无论多简单，都会使用到流。这有一个使用流的例子。

```
javascript
var http = require('http');

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

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

```
// 可读流（Readable stream） emit 'data' 事件 once a 监听器（listener） is added
req.on('data', function (chunk) {
  body += chunk;
});

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

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

server.listen(1337);

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

类： stream.Readable

可读流（Readable stream）接口是对你正在读取的数据的来源的抽象。换句话说，数据来来自

可读流（Readable stream）不会分发数据，直到你表明准备就绪。

可读流（Readable stream） 有2种模式: **流动模式（flowing mode）** 和 **暂停模式（paused mode）**. 流动模式（flowing mode）时,尽快的从底层系统读取数据并提供给你的程序。 暂停模式（paused mode）时, 你必须明确的调用 `stream.read()` 来读取数据。 暂停模式（paused mode） 是默认模式。

注意: 如果没有绑定数据处理函数，并且没有 [`pipe()`][] 目标，流会切换到流动模式（flowing mode），并且数据会丢失。

可以通过下面几个方法，将流切换到流动模式（flowing mode）。

- 添加一个 ['data' 事件][] 事件处理器来监听数据.
- 调用 [resume()][] 方法来明确的开启数据流。
- 调用 [pipe()][] 方法来发送数据给Writable (页 0).

可以通过以下方法来切换到暂停模式（paused mode）:

- 如果没有 导流（pipe） 目标，调用 [pause()][]方法.
- 如果有 导流（pipe） 目标, 移除所有的 ['data' 事件][]处理函数, 调用 [unpipe()][] 方法移除所有的 导流（pipe） 目标。

注意, 为了向后兼容考虑， 移除 'data' 事件监听器并不会自动暂停流。同样的，当有导流目标时，调用 pause() 并不能保证流在那些目标排空后，请求更多数据时保持暂停状态。

可读流（Readable stream）例子包括:

- http responses, on the client (https://nodejs.org/api/http.html#http_http_incomingmessage)
- http requests, on the server (https://nodejs.org/api/http.html#http_http_incomingmessage)
- fs read streams (https://nodejs.org/api/fs.html#fs_class_fs_readstream)
- zlib streams (zlib.html)
- crypto streams (crypto.html)
- tcp sockets (net.html#net_class_net_socket)
- child process stdout and stderr (child_process.html#child_process_child_stdout)
- process.stdin (process.html#process_process_stdin)

事件: 'readable'
当一个数据块可以从流中读出，将会触发 'readable' 事件.`

某些情况下, 如果没有准备好，监听一个 'readable' 事件将会导致一些数据从底层系统读取到内部缓存。

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

一旦内部缓存排空，一旦有更多数据将会再次触发 readable 事件。

事件: 'data'

- `chunk` {Buffer | String} 数据块

绑定一个 `data` 事件的监听器（listener）到一个未明确暂停的流，会将流切换到流动模式。数据会尽额能的传递。

如果你像尽快的从流中获取数据，这是最快的方法。

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

事件: 'end'

如果没有更多的可读数据，将会触发这个事件。

注意，除非数据已经被完全消费， the `end` 事件**才会触发**。 可以通过切换到流动模式（flowing mode）来实现，或者通过调用重复调用 `read()` 获取数据，直到结束。

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

事件: 'close'

当底层资源（例如源头的文件描述符）关闭时触发。并不是所有流都会触发这个事件。

事件: 'error'

- {Error Object}

当接收数据时发生错误触发。

readable.read([size])

- `size` {Number} 可选参数， 需要读入的数据量
- 返回 {String | Buffer | null}

`read()` 方法从内部缓存中拉取数据。如果没有可用数据，将会返回 `null`

如果传了 `size` 参数，将会返回相当字节的数据。如果 `size` 不可用，将会返回 `null`

如果你没有指定 size 参数。将会返回内部缓存的所有数据。

这个方法仅能再暂停模式（paused mode）里调用. 流动模式（flowing mode）下这个方法会被自动调用直到内存缓存排空。

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

如果这个方法返回一个数据块, 它同时也会触发['data' 事件][].

readable.setEncoding(encoding)

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

调用此函数会使得流返回指定编码的字符串，而不是 Buffer 对象。例如，如果你调用 readable.setEncoding('utf8') ，输出数据将会是UTF-8 编码，并且返回字符串。如果你调用 readable.setEncoding('hex') ，将会返回2进制编码的数据。

该方法能正确处理多字节字符。如果不想这么做，仅简单的直接拉取缓存并调 buf.toString(encoding) ，可能会导致字节错位。因此，如果你想以字符串读取数据，请使用这个方法。

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

readable.resume()

- 返回: this

这个方法让可读流（Readable stream）继续触发 data 事件.

这个方法会将流切换到流动模式（flowing mode）. 如果你不想从流中消费数据，而想得到 end 事件，可以调用 [readable.resume()][] 来打开数据流。

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

readable.pause()

- 返回: this

这个方法会使得流动模式（flowing mode）的流停止触发 data 事件, 切换到流动模式（flowing mode）. 并让后续可用数据留在内部缓冲区中。

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

readable.isPaused()

- 返回: Boolean

这个方法返回 readable 是否被客户端代码 **明确**的暂停（调用 readable.pause() ）。

``javascript var readable = new stream.Readable

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

```
 #### readable.pipe(destination[, options])

* `destination` {[Writable][] Stream} 写入数据的目标
* `options` {Object} 导流（pipe） 选项
  * `end` {Boolean} 读取到结束符时，结束写入者。默认 = `true`

这个方法从可读流（Readable stream）拉取所有数据, 并将数据写入到提供的目标中。自动管理流量，这样目标不会快速的可读流（

可以导流到多个目标。
```

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

这个函数返回目标流, 因此你可以建立导流链:

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

例如, 模拟 Unix 的 `cat` 命令:

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

默认情况下，当源数据流触发 `end`的时候调用[`end()`][]，所以 `destination` 不可再写。传 `{ end:false }`作为`options`，可以保持

这会让 `writer`保持打开状态，可以在最后写入"Goodbye" 。

javascript reader.pipe(writer, { end: false }); reader.on('end', function() { writer.end('Goodbye\n'); });

注意 `process.stderr` 和 `process.stdout` 直到进程结束才会关闭，无论是否指定

readable.unpipe([destination])

* `destination` {[Writable][] Stream} 可选，指定解除导流的流

这个方法会解除之前调用 `pipe()` 设置的钩子（ `pipe()` ）。

如果没有指定 `destination`,所有的 导流（ pipe ） 都会被移除。

如果指定了 `destination`，但是没有建立如果没有指定 `destination`，则什么事情都不会发生。

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

readable.unshift(chunk)

* `chunk` {Buffer | String} 数据块插入到读队列中

这个方法很有用，当一个流正被一个解析器消费，解析器可能需要将某些刚拉取出的数据“逆消费”，返回到原来的源，以便流能将它

如果你在程序中必须经常调用 `stream.unshift(chunk)` ，那你可以考虑实现[Transform][]来替换（参见下文API for Stream Implem

javascript // Pull off a header delimited by \n\n // use unshift() if we get too much // Call the callback wit h (error, header, stream) var StringDecoder = require('string_decoder').StringDecoder; function pars eHeader(stream, callback) { stream.on('error', callback); stream.on('readable', onReadable); var deco der = new StringDecoder('utf8'); var header = ''; function onReadable() { var chunk; while (null !== (ch unk = stream.read())) { var str = decoder.write(chunk); if (str.match(/\n\n/)) { // found the header boun dary var split = str.split(/\n\n/); header += split.shift(); var remaining = split.join('\n\n'); var buf = new Bu ffer(remaining, 'utf8'); if (buf.length) stream.unshift(buf); stream.removeListener('error', callback); stre am.removeListener('readable', onReadable); // now the body of the message can be read from the str eam. callback(null, header, stream); } else { // still reading the header. header += str; } } } }

readable.wrap(stream)

* `stream` {Stream} 一个旧式的可读流（Readable stream）

v0.10 版本之前的 Node 流并未实现现在所有流的API（更多信息详见下文“兼容性”章节）。

如果你使用的是旧的 Node 库，它触发 `"data"` 事件，并拥有仅做查询用的[`pause()`][] 方法，那么你能使用`wrap()` 方法来创建一个

你应该很少需要用到这个函数，但它会留下方便和旧版本的 Node 程序和库交互。

例如:

javascript var OldReader = require('./old-api-module.js').OldReader; var oreader = new OldReader; v ar Readable = require('stream').Readable; var myReader = new Readable().wrap(oreader);

myReader.on('readable', function() { myReader.read(); // etc. });

类: stream.Writable

<!--type=class-->

可写流（Writable stream ）接口是你正把数据写到一个目标的抽象。

可写流（Writable stream ）的例子包括:

* [http requests, on the client](#http_class_http_clientrequest)
* [http responses, on the server](#http_class_http_serverresponse)

```
* [fs write streams](#fs_class_fs_writestream)
* [zlib streams][]
* [crypto streams][]
* [tcp sockets][]
* [child process stdin](#child_process_child_stdin)
* [process.stdout][], [process.stderr][]

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

* `chunk` {String | Buffer} 准备写的数据
* `encoding` {String} 编码方式（如果`chunk` 是字符串）
* `callback` {Function} 数据块写入后的回调
* 返回: {Boolean} 如果数据已被全部处理返回true

这个方法向底层系统写入数据，并在数据处理完毕后调用所给的回调。

返回值表示你是否应该继续立即写入。如果数据要缓存在内部，将会返回`false`。否则返回 `true`。

返回值仅供参考。即使返回 `false`，你也可能继续写。但是写会缓存在内存里，所以不要做的太过分。最好的办法是等待`drain` 事件后

 #### 事件: 'drain'

如果调用 [`writable.write(chunk)`][] 返回 false, `drain` 事件会告诉你什么时候将更多的数据写入到流中。
```

javascript // Write the data to the supplied 可写流（Writable stream ）1MM times. // Be attentive to back-pressure. function writeOneMillionTimes(writer, data, encoding, callback) { var i = 1000000; write(); function write() { var ok = true; do { i -= 1; if (i === 0) { // last time! writer.write(data, encoding, callback); } else { // see if we should continue, or wait // don't pass the callback, because we're not done yet. ok = writer.write(data, encoding); } } while (i > 0 && ok); if (i > 0) { // had to stop early! // write some more once it drains writer.once('drain', write); } } }

```
 #### writable.cork()

强制缓存所有写入。

调用 `.uncork()` 或 `.end()`后，会把缓存数据写入。

 #### writable.uncork()

写入所有 .cork() 调用之后缓存的数据。

 #### writable.setDefaultEncoding(encoding)
```

* `encoding` {String} 新的默认编码
* 返回: `Boolean`

给写数据流设置默认编码方式，如编码有效，返回 `true` ，否则返回 `false`。

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

* `chunk` {String | Buffer} 可选，要写入的数据
* `encoding` {String} 编码方式（如果 `chunk` 是字符串）
* `callback` {Function} 可选， stream 结束时的回调函数

当没有更多的数据写入的时候调用这个方法。如果给出，回调会被用作 finish 事件的监听器。

调用 [`end()`][] 后调用 [`write()`][] 会产生错误。

javascript // write 'hello, ' and then end with 'world!' var file = fs.createWriteStream('example.txt'); file.write('hello, '); file.end('world!'); // writing more now is not allowed!

事件: 'finish'

调用[`end()`][] 方法后，并且所有的数据已经写入到底层系统，将会触发这个事件。

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

事件: 'pipe'

* `src` {[Readable][] Stream} 是导流（pipe）到可写流的源流

无论何时在可写流（Writable stream ）上调用`pipe()` 方法，都会触发 'pipe' 事件，添加这个流到目标。

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

事件: 'unpipe'

* `src` {[Readable][] Stream} The source stream that [unpiped][] this writable

无论何时在可写流（Writable stream ）上调用`unpipe()` 方法,都会触发 'unpipe' 事件,将这个流从目标上移除。

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

事件: 'error'

* {Error object}

写或导流（pipe）数据时，如果有错误会触发。

类： stream.Duplex

双工流（Duplex streams）是同时实现了 [Readable][] and [Writable][] 接口。用法详见下文。

双工流（Duplex streams）的例子包括:

* [tcp sockets][]
* [zlib streams][]
* [crypto streams][]

类： stream.Transform

转换流（Transform streams）是双工 [Duplex][] 流，它的输出是从输入计算得来。 它实现了[Readable][] 和 [Writable][] 接口. 用

转换流（Transform streams）的例子包括:

* [zlib streams][]
* [crypto streams][]

API for Stream Implementors

<!--type=misc-->

无论实现什么形式的流，模式都是一样的:

1. 在你的子类中扩展适合的父类. ([`util.inherits`][] 方法很有帮助)
2. 在你的构造函数中调用父类的构造函数，以确保内部的机制初始化正确。
2. 实现一个或多个方法，如下所列

所扩展的类和要实现的方法取决于你要编写的流类。

```
<table>
 <thead>
  <tr>
   <th>
    <p>Use-case</p>
   </th>
   <th>
    <p>Class</p>
   </th>
   <th>
    <p>方法(s) to implement</p>
   </th>
  </tr>
 </thead>
 <tr>
  <td>
   <p>Reading only</p>
  </td>
  <td>
   <p>[Readable](#stream_class_stream_readable_1)</p>
  </td>
  <td>
   <p><code>[_read][]</code></p>
  </td>
 </tr>
 <tr>
  <td>
   <p>Writing only</p>
  </td>
  <td>
   <p>[Writable](#stream_class_stream_writable_1)</p>
  </td>
  <td>
   <p><code>[_write][]</code></p>
  </td>
 </tr>
 <tr>
  <td>
   <p>Reading and writing</p>
  </td>
  <td>
   <p>[Duplex](#stream_class_stream_duplex_1)</p>
  </td>
  <td>
```

```
    <p><code>[_read][]</code>, <code>[_write][]</code></p>
   </td>
  </tr>
  <tr>
   <td>
    <p>Operate on written data, then read the result</p>
   </td>
   <td>
    <p>[Transform](#stream_class_stream_transform_1)</p>
   </td>
   <td>
    <p><code>_transform</code>, <code>_flush</code></p>
   </td>
  </tr>
</table>

在你的代码里，千万不要调用 [API for Stream Consumers][] 里的方法。否则可能会引起消费流的程序副作用。

 ### 类： stream.Readable

<!--type=class-->

`stream.Readable` 是一个可被扩充的、实现了底层 `_read(size)` 方法的抽象类。

参照之前的[API for Stream Consumers][]查看如何在你的程序里消费流。底下内容解释了在你的程序里如何实现可读流（Readable

 #### Example: 计数流

<!--type=example-->

这是可读流（Readable stream）的基础例子. 它将从 1 至 1,000,000 递增地触发数字，然后结束。
```

javascript var Readable = require('stream').Readable; var util = require('util'); util.inherits(Counter, Readable);

function Counter(opt) { Readable.call(this, opt); this._max = 1000000; this._index = 1; }

Counter.prototype._read = function() { var i = this._index++; if (i > this._max) this.push(null); else { var str = '' + i; var buf = new Buffer(str, 'ascii'); this.push(buf); } };

```
 #### Example: 简单协议 v1 (初始版)

和之前描述的 `parseHeader` 函数类似, 但它被实现为自定义流。注意这个实现不会将输入数据转换为字符串。
```

实际上，更好的办法是将他实现为 [Transform][] 流。下面的实现方法更好。

javascript // A parser for a simple data protocol. // "header" is a JSON object, followed by 2 \n characte rs, and // then a message body. // // 注意: This can be done more simply as a Transform stream! // Usi ng Readable directly for this is sub-optimal. See the // alternative example below under Transform se ction.

var Readable = require('stream').Readable; var util = require('util');

util.inherits(SimpleProtocol, Readable);

function SimpleProtocol(source, options) { if (!(this instanceof SimpleProtocol)) return new SimpleProt ocol(source, options);

Readable.call(this, options;

this._inBody = false; this._sawFirstCr = false;

// source is 可读流（Readable stream）, such as a socket or file this._source = source;

var self = this; source.on('end', function() { self.push(null); });

// give it a kick whenever the source is readable // read(0) will not consume any bytes source.on('read able', function() { self.read(0); });

this._rawHeader = []; this.header = null; }

SimpleProtocol.prototype._read = function(n) { if (!this._inBody) { var chunk = this._source.read();

```
// if the source doesn't have data, we don't have data yet.
if (chunk === null)
  return this.push('');

// check if the chunk has a \n\n
var split = -1;
for (var i = 0; i < chunk.length; i++) {
  if (chunk[i] === 10) { // '\n'
    if (this._sawFirstCr) {
      split = i;
      break;
    } else {
      this._sawFirstCr = true;
    }
  } else {
```

```
    this._sawFirstCr = false;
  }
}

if (split === -1) {
  // still waiting for the \n\n
  // stash the chunk, and try again.
  this._rawHeader.push(chunk);
  this.push('');
} else {
  this._inBody = true;
  var h = chunk.slice(0, split);
  this._rawHeader.push(h);
  var header = Buffer.concat(this._rawHeader).toString();
  try {
    this.header = JSON.parse(header);
  } catch (er) {
    this.emit('error', new Error('invalid simple protocol data'));
    return;
  }
  // now, because we got some extra data, unshift the rest
  // back into the 读取队列 so that our consumer will see it.
  var b = chunk.slice(split);
  this.unshift(b);

  // and let them know that we are done parsing the header.
  this.emit('header', this.header);
}
```

} else { // from there on, just provide the data to our consumer. // careful not to push(null), since that w ould indicate EOF. var chunk = this._source.read(); if (chunk) this.push(chunk); } };

// Usage: // var parser = new SimpleProtocol(source); // Now parser is 可读流（Readable stream） tha t will emit 'header' // with the parsed header data.

```
 #### new stream.Readable([options])

* `options` {Object}
  * `highWaterMark` {Number} 停止从底层资源读取数据前，存储在内部缓存的最大字节数。默认=16kb，`objectMode` 流是16.
  * `encoding` {String} 若指定，则 Buffer 会被解码成所给编码的字符串。缺省为 null
  * `objectMode` {Boolean} 该流是否为对象的流。意思是说 stream.read(n) 返回一个单独的值，而不是大小为 n 的 Buffer。

Readable 的扩展类中，确保调用了 Readable 的构造函数，这样才能正确初始化。
```

```
#### readable.\_read(size)

* `size` {Number} 异步读取的字节数

注意: **实现这个函数, 但不要直接调用.**

这个函数不要直接调用. 在子类里实现,仅能被内部的 Readable 类调用。

所有可读流（Readable stream）的实现必须停供一个 `_read` 方法，从底层资源里获取数据。

这个方法以下划线开头，是因为对于定义它的类是内部的，不会被用户程序直接调用。 你可以在自己的扩展类中实现。

当数据可用时，通过调用`readable.push(chunk)` 将之放到读取队列中。再次调用 `_read` ，需要继续推出更多数据。

`size` 参数仅供参考. 调用 “read” 可以知道知道应当抓取多少数据；其余与之无关的实现，比如 TCP 或 TLS，则可忽略这个参数，

#### readable.push(chunk[, encoding])

* `chunk` {Buffer | null | String} 推入到读取队列的数据块
* `encoding` {String} 字符串块的编码。必须是有效的 Buffer 编码，比如 utf8 或 ascii。
* 返回 {Boolean} 是否应该继续推入

注意: **这个函数必须被 Readable 实现者调用, 而不是可读流（Readable stream）的消费者.**

`_read()` 函数直到调用`push(chunk)` 后才能被再次调用。

`Readable` 类将数据放到读取队列，当 `'readable'` 事件触发后，被 `read()` 方法取出。`push()` 方法会插入数据到读取队列中。如

这个 API 被设计成尽可能地灵活。比如说，你可以包装一个低级别的，具备某种暂停/恢复机制，和数据回调的数据源。这种情况下，你
```

javascript // source is an object with readStop() and readStart() 方法s, // and an `ondata` member that gets called when it has data, and // an `onend` member that gets called when the data is over.

util.inherits(SourceWrapper, Readable);

function SourceWrapper(options) { Readable.call(this, options);

this._source = getLowlevelSourceObject(); var self = this;

// Every time there's data, we push it into the internal buffer. this._source.ondata = function(chunk) { // if push() 返回 false, then we need to stop reading from source if (!self.push(chunk)) self._source.read Stop(); };

// When the source ends, we push the EOF-signaling `null` chunk this._source.onend = function() { s elf.push(null); }; }

// _read will be called when the stream wants to pull more data in // the advisory size 参数 is ignored in this case. SourceWrapper.prototype._read = function(size) { this._source.readStart(); };

```
 ### 类： stream.Writable

<!--type=class-->

`stream.Writable` 是个抽象类，它扩展了一个底层的实现[`_write(chunk, encoding, callback)`][] 方法.

参考上面的[API for Stream Consumers][]，来了解在你的程序里如何消费可写流。下面内容介绍了如何在你的程序里实现可写流。

 #### new stream.Writable([options])

* `options` {Object}
  * `highWaterMark` {Number} 当 [`write()`][] 返回 false 时的缓存级别. 默认=16kb,`objectMode` 流是 16.
  * `decodeStrings` {Boolean} 传给 [`_write()`][] 前是否解码为字符串。 默认=true
  * `objectMode` {Boolean} `write(anyObj)` 是否是有效操作.如果为 true，可以写任意数据，而不仅仅是`Buffer` / `String`. 默认=f

请确保 Writable 类的扩展类中，调用构造函数以便缓冲设定能被正确初始化。

 #### writable.\_write(chunk, encoding, callback)

* `chunk` {Buffer | String} 要写入的数据块。总是 buffer， 除非 `decodeStrings` 选项为 `false`。
* `encoding` {String} 如果数据块是字符串，这个参数就是编码方式。如果是缓存，则忽略。注意，除非`decodeStrings` 被设置为 `f
* `callback` {函数} 当你处理完数据后调用这个函数 (错误参数为可选参数)。

所以可写流（ Writable stream ） 实现必须提供一个 [`_write()`][]方法，来发送数据给底层资源。

注意: **这个函数不能直接调用** ,由子类实现， 仅内部可写方法可以调用。

使用标准的 `callback(error)` 方法调用回调函数，来表明写入完成或遇到错误。

如果构造函数选项中设定了 `decodeStrings` 标识，则 `chunk` 可能会是字符串而不是 Buffer， `encoding` 表明了字符串的格式。这

该方法以下划线开头，是因为对于定义它的类来说，这个方法是内部的，并且不应该被用户程序直接调用。你应当在你的扩充类中重写这

 ### writable.\_writev(chunks, callback)

* `chunks` {Array} 准备写入的数据块，每个块格式如下: `{ chunk: ..., encoding: ... }`.
```

* `callback` {函数} 当你处理完数据后调用这个函数 (错误参数为可选参数)。

注意: **这个函数不能直接调用。** 由子类实现，仅内部可写方法可以调用.

这个函数的实现是可选的。多数情况下，没有必要实现。如果实现，将会在所有数据块缓存到写队列后调用。

类： stream.Duplex

<!--type=class-->

双工流 (duplex stream) 同时兼具可读和可写特性，比如一个 TCP socket 连接。

注意 `stream.Duplex` 可以像 Readable 或 Writable 一样被扩充，实现了底层 _read(sise) 和 _write(chunk, encoding, callback)

由于 JavaScript 并没有多重继承能力，因此这个类继承自 Readable，寄生自 Writable.从而让用户在双工扩展类中同时实现低级别的

new stream.Duplex(options)

* `options` {Object} 传递 Writable and Readable 构造函数，有以下的内容:
  * `allowHalfOpen` {Boolean} 默认=true. 如果设置为 `false`, 当写端结束的时候，流会自动的结束读端，反之亦然。
  * `readableObjectMode` {Boolean} 默认=false. 将 `objectMode` 设为读端的流，如果为 `true`，将没有效果。
  * `writableObjectMode` {Boolean} 默认=false. 将 `objectMode`设为写端的流，如果为 `true`，将没有效果。

扩展自 Duplex 的类，确保调用了父亲的构造函数，保证缓存设置能正确初始化。

类： stream.Transform

转换流 (transform class) 是双工流 (duplex stream)，输入输出端有因果关系，比如[zlib][] 流或 [crypto][] 流。

输入输出没有要求大小相同，块数量相同，到达时间相同。例如，一个 Hash 流只会在输入结束时产生一个数据块的输出；一个 zlib 流

转换流 (transform class) 必须实现`_transform()` 方法，而不是[`_read()`][] 和 [`_write()`][] 方法，也可以实现`_flush()` 方法 (参

new stream.Transform([options])

* `options` {Object} 传递给 Writable and Readable 构造函数。

扩展自 转换流 (transform class) 的类，确保调用了父亲的构造函数，保证缓存设置能正确初始化。

transform._transform(chunk, encoding, callback)

* `chunk` {Buffer | String} 准备转换的数据块。是buffer，除非 `decodeStrings` 选项设置为 `false`。
* `encoding` {String} 如果数据块是字符串, 这个参数就是编码方式，否则就忽略这个参数
* `callback` {函数} 当你处理完数据后调用这个函数 (错误参数为可选参数)。

注意: **这个函数不能直接调用。** 由子类实现，仅内部可写方法可以调用.

所有的转换流（transform class）实现必须提供 `_transform`方法来接收输入，并生产输出。

`_transform` 可以做转换流（transform class)里的任何事，处理写入的字节，传给接口的写端，异步 I/O,处理事情等等。

调用 `transform.push(outputChunk)`0或多次，从这个输入块里产生输出，依赖于你想要多少数据作为输出。

仅在当前数据块完全消费后调用这个回调。注意，输入块可能有，也可能没有对应的输出块。如果你提供了第二个参数，将会传给push

javascript transform.prototype._transform = function (data, encoding, callback) { this.push(data); callback(); }

transform.prototype._transform = function (data, encoding, callback) { callback(null, data); }

该方法以下划线开头，是因为对于定义它的类来说，这个方法是内部的，并且不应该被用户程序直接调用。你应当在你的扩充类中重写

transform._flush(callback)

* `callback` {函数} 当你处理完数据后调用这个函数 (错误参数为可选参数)

注意: **这个函数不能直接调用。** 由子类实现，仅内部可写方法可以调用.

某些情况下，转换操作可能需要分发一点流最后的数据。例如， `Zlib`流会存储一些内部状态，以便优化压缩输出。

有些时候，你可以实现 `_flush` 方法，它可以在最后面调用，当所有的写入数据被消费后，分发`end`告诉读端。和 `_transform` 一样

该方法以下划线开头，是因为对于定义它的类来说，这个方法是内部的，并且不应该被用户程序直接调用。你应当在你的扩充类中重写

事件: 'finish' and 'end'

[`finish`][] 和 [`end`][] 事件 分别来自 Writable 和 Readable 类。`.end()`事件结束后调用 `finish` 事件，所有的数据已经被`_transfo

Example: `SimpleProtocol` parser v2

上面的简单协议分析例子列子可以通过使用高级别的[Transform][] 流来实现，和 `parseHeader` ， `SimpleProtocol v1`列子类似。

在这个示例中，输入会被导流到解析器中，而不是作为参数提供。这种做法更符合 Node 流的惯例。

javascript var util = require('util'); var Transform = require('stream').Transform; util.inherits(SimpleProtocol, Transform);

function SimpleProtocol(options) { if (!(this instanceof SimpleProtocol)) return new SimpleProtocol(opt ions);

Transform.call(this, options); this._inBody = false; this._sawFirstCr = false; this._rawHeader = []; thi s.header = null; }

SimpleProtocol.prototype._transform = function(chunk, encoding, done) { if (!this._inBody) { // check if the chunk has a \n\n var split = -1; for (var i = 0; i < chunk.length; i++) { if (chunk[i] === 10) { // '\n' if (thi s._sawFirstCr) { split = i; break; } else { this._sawFirstCr = true; } } else { this._sawFirstCr = false; } }

```
if (split === -1) {
  // still waiting for the \n\n
  // stash the chunk, and try again.
  this._rawHeader.push(chunk);
} else {
  this._inBody = true;
  var h = chunk.slice(0, split);
  this._rawHeader.push(h);
  var header = Buffer.concat(this._rawHeader).toString();
  try {
    this.header = JSON.parse(header);
  } catch (er) {
    this.emit('error', new Error('invalid simple protocol data'));
    return;
  }
  // and let them know that we are done parsing the header.
  this.emit('header', this.header);

  // now, because we got some extra data, emit this first.
  this.push(chunk.slice(split));
}
```

} else { // from there on, just provide the data to our consumer as-is. this.push(chunk); } done(); };

// Usage: // var parser = new SimpleProtocol(); // source.pipe(parser) // Now parser is 可读流（Readab le stream） that will emit 'header' // with the parsed header data.

类： stream.PassThrough

这是[Transform][] 流的简单实现，将输入的字节简单的传递给输出。它的主要用途是测试和演示。偶尔要构建某种特殊流时也会用到。

流: 内部细节

<!--type=misc-->

缓冲

<!--type=misc-->

可写流（Writable streams ）和 可读流（Readable stream ）都会缓存数据到内部对象上，叫做 `_writableState.buffer` 或 `_rea

缓存的数据量，取决于构造函数是传入的 `highWaterMark` 参数。

调用 [`stream.push(chunk)`][] 时，缓存数据到可读流（Readable stream ）。在数据消费者调用 `stream.read()` 前，数据会一直

调用 [`stream.write(chunk)`][] 时，缓存数据到可写流（Writable stream ）。即使 `write()` 返回 `false`。

流（尤其是`pipe()` 方法 ）得目的是限制数据的缓存量到一个可接受的水平，使得不同速度的源和目的不会淹没可用内存。

`stream.read(0)`

某些时候，你可能想不消费数据的情况下，触发底层可读流（Readable stream ）机制的刷新。这种情况下可以调用 stream.read(0)

如果内部读取缓冲低于 `highWaterMark`，并且流当前不在读取状态，那么调用 `read(0)` 会触发一个低级 `_read` 调用。

虽然基本上没有必要这么做。但你在 Node 内部的某些地方看到它确实这么做了，尤其是在 Readable 流类的内部。

`stream.push('')`

推一个0字节的字符串或缓存 (不在[Object mode][]时)会发送有趣的副作用. 因为它是一个对
[`stream.push()`][] 的调用, 它将会结束 `reading` 进程. 然而，它没有添加任何数据到可读缓冲区中，所以没有东西可供用户消费。

少数情况下，你当时没有提供数据，但你的流的消费者（或你的代码的其它部分 ）会通过调用 `stream.read(0)` 得知何时再次检查。在

到目前为止，这个功能唯一一个使用情景是在 [tls.CryptoStream][] 类中，但它将在 Node v0.12 中被废弃。如果你发现你不得不使用

和老版本的兼容性

<!--type=misc-->

v0.10 版本前，可读流（Readable stream ）接口比较简单，因此功能和用处也小。

* `'data'`事件会立即开始触发，而不会等待你调用 `read()` 方法。如果你需要进行某些 I/O 来决定如何处理数据，那么你只能将数据块
* [`pause()`][] 方法仅供参考，而不保证生效。这意味着，即便流处于暂停状态时，你仍然需要准备接收 'data' 事件。

在 Node v0.10中, 加入了下文所述的 Readable 类。为了考虑向后兼容，添加了 'data' 事件监听器或 resume() 方法被调用时，可读流

大多数程序会维持正常功能。然而，下列条件下也会引入边界情况：

* 没有添加 [`'data'` 事件][] 处理器
* 从来没有调用 [`resume()`][] 方法
* 流从来没有被倒流（ pipe ）到任何可写目标上、

例如：

javascript // WARNING! BROKEN! net.createServer(function(socket) {

// we add an 'end' 方法, but never consume the data socket.on('end', function() { // It will never get her e. socket.end('I got your message (but didnt read it)\n'); });

}).listen(1337);

v0.10 版本前的 Node, 流入的消息数据会被简单的抛弃。之后的版本，socket 会一直保持暂停。

这种情形下，调用`resume()` 方法来开始工作：

javascript // Workaround net.createServer(function(socket) {

socket.on('end', function() { socket.end('I got your message (but didnt read it)\n'); });

// start the flow of data, discarding it. socket.resume();

}).listen(1337);

可读流（ Readable stream ）切换到流动模式（ flowing mode ）,v0.10 版本前，可以使用`wrap()` 方法将风格流包含在一个可读类里

Object Mode

<!--type=misc-->

通常情况下，流仅操作字符串和缓存。

处于 **object mode** 的流，除了 缓存和字符串，还可以可以读出普通 JavaScript值。

在对象模式里，可读流（ Readable stream ） 调用 `stream.read(size)`总会返回单个项目，无论是什么参数。

在对象模式里， 可写流（ Writable stream ） 总会忽略传给`stream.write(data, encoding)`的 `encoding`参数。

特殊值 `null` 在对象模式里，依旧保持它的特殊性。也就说，对于对象模式的可读流（ Readable stream ），`stream.read()` 返回 n

Node 核心不存在对象模式的流，这种设计只被某些用户态流式库所使用。

应该在你的子类构造函数里，设置`objectMode` 。在过程中设置不安全。

对于双工流（ Duplex streams ），`objectMode`可以用`readableObjectMode` 和 `writableObjectMode` 分别为读写端分别设置。

javascript var util = require('util'); var StringDecoder = require('string_decoder').StringDecoder; var Tr ansform = require('stream').Transform; util.inherits(JSONParseStream, Transform);

// Gets \n-delimited JSON string data, and emits the parsed objects function JSONParseStream() { if (!(this instanceof JSONParseStream)) return new JSONParseStream();

Transform.call(this, { readableObjectMode : true });

this._buffer = ''; this._decoder = new StringDecoder('utf8'); }

JSONParseStream.prototype._transform = function(chunk, encoding, cb) { this._buffer += this._deco der.write(chunk); // split on newlines var lines = this._buffer.split(/\r?\n/); // keep the last partial line buff ered this._buffer = lines.pop(); for (var l = 0; l < lines.length; l++) { var line = lines[l]; try { var obj = JSO N.parse(line); } catch (er) { this.emit('error', er); return; } // push the parsed object out to the readable consumer this.push(obj); } cb(); };

JSONParseStream.prototype._flush = function(cb) { // Just handle any leftover var rem = this._buffe r.trim(); if (rem) { try { var obj = JSON.parse(rem); } catch (er) { this.emit('error', er); return; } // push th e parsed object out to the readable consumer this.push(obj); } cb(); }; ```

28

字符串解码器

稳定性: 3 – 稳定

通过 `require('string_decoder')` ，可以使用这个模块。字符串解码器（StringDecoder）将缓存（buffer）解码为字符串。这是 `buffer.toString()` 的简单接口，提供了 utf8 支持。

```
var StringDecoder = require('string_decoder').StringDecoder;
var decoder = new StringDecoder('utf8');

var cent = new Buffer([0xC2, 0xA2]);
console.log(decoder.write(cent));

var euro = new Buffer([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

Class: StringDecoder

接受一个参数 `encoding`，默认值 `utf8`。

decoder.write(buffer)

返回解码后的字符串。

decoder.end()

返回 buffer 里剩下的末尾字节。

29

定时器

稳定性: 5 – 锁定

所有的定时器函数都是全局的。不需要通过 `require()` 就可以访问。

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

`delay` 毫秒之后执行 `callback` 。返回 `timeoutObject` 对象，可能会用来 `clearTimeout()` 。你也可以给回调函数传参数。

需要注意，你的回调函数可能不会非常准确的在 `delay` 毫秒后执行，Node.js 不保证回调函数的精确时间和执行顺序。回调函数会尽量的靠近指定的时间。

clearTimeout(timeoutObject)

阻止一个 timeout 被触发。

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

每隔 `delay` 毫秒就重复执行 `callback` 。返回 `timeoutObject` 对象，可能会用来 `clearTimeout()` 。你也可以给回调函数传参数。

clearInterval(intervalObject)

阻止一个 interval 被触发。

unref()

`setTimeout` 和 `setInterval` 所返回的值，拥有 `timer.unref()` 方法，它能让你创建一个活动的定时器，但是它所在的事件循环中如果仅剩它一个计时器，将不会保持程序运行。如果计时器已经调用了 `unref` ，再次调用将无效。

在 `setTimeout` 场景中，当你使用 `unref` 并创建了一个独立定时器它将会唤醒事件循环。创建太多的这样的东西会影响事件循环性能，所以谨慎使用。

ref()

如果你之前已经使用 `unref()` 一个定时器，就可以使用 `ref()` 来明确的请求定时器保持程序打开状态。果计时器已经调用了 `ref（）`，再次调用将无效。

setImmediate(callback[, arg][, ...])

在 `setTimeout` 和 `setInterval` 事件前，在输入/输出事件后，安排一个 `callback` "immediate" 立即执行。

immediates 的回调以它们创建的顺序加入队列。整个回调队列会在事件循环迭代中执行。如果你将immediates加入到一个正在执行回调中，那么将不会触发immediate，直到下次事件循环迭代。

clearImmediate(immediateObject)

停止一个 immediate 的触发。

30

TLS/SSL

```
Stability: 3 - Stable
```

可以使用 `require('tls')` 来访问这个模块。

`tls` 模块 使用 OpenSSL 来提供传输层（Transport Layer）安全性和（或）安全套接层（Secure Socket Layer）：加密过的流通讯。

TLS/SSL 是一种公钥/私钥基础架构。每个客户端和服务端都需要一个私钥。私钥可以用以下方法创建的：

```
openssl genrsa -out ryans-key.pem 2048
```

所有服务器和某些客户端需要证书。证书由认证中心（Certificate Authority）签名，或者自签名。获得证书第一步是创建一个证书签名请求 "Certificate Signing Request" (CSR)文件。证书可以用以下方法创建的：

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

使用CSR创建一个自签名的证书：

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

或者你可以发送 CSR 给认证中心（Certificate Authority）来签名。

(TODO: 创建 CA 的文档, 感兴趣的读者可以在 Node 源码 `test/fixtures/keys/Makefile` 里查看)

创建 .pfx 或 .p12，可以这么做：

```
openssl pkcs12 -export -in agent5-cert.pem -inkey agent5-key.pem \
  -certfile ca-cert.pem -out agent5.pfx
```

- `in` : certificate
- `inkey` : private key
- `certfile` : all CA certs concatenated in one file like `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

协议支持

Node.js 默认遵循 SSLv2 和 SSLv3 协议，不过这些协议被禁用。因为他们不太可靠，很容易受到威胁，参见 CVE-2014-3566 (https://access.redhat.com/articles/1232123) 。某些情况下，旧版本客户端/服务器（比如 IE6）可能会产生问题。如果你想启用 SSLv2 或 SSLv3 ，使用参数 `--enable-ssl2` 或 `--enable-ssl3` 运行 Node。Node.js 的未来版本中不会再默认编译 SSLv2 和 SSLv3。

有一个办法可以强制 node 进入仅使用 SSLv3 或 SSLv2 模式，分别指定 `secureProtocol` 为 `'SSLv3_method'` 或 `'SSLv2_method'` 。

Node.js 使用的默认协议方法准确名字是 `AutoNegotiate_method`， 这个方法会尝试并协商客户端支持的从高到底协议。为了提供默认的安全级别，Node.js(v0.10.33 版本之后)通过将 `secureOptions` 设为 `SSL_OP_NO_SSLv3|SSL_OP_NO_SSLv2` ，明确的禁用了 SSLv3 和 SSLv2（除非你给 `secureProtocol` 传值 `--enable-ssl3`, 或 `--enable-ssl2`, 或 `SSLv3_method` ）。

如果你设置了 `secureOptions`，我们不会重新这个参数。

改变这个行为的后果：

- 如果你的应用被当做为安全服务器， `SSLv3` 客户端不能协商建立连接，会被拒绝。这种情况下，你的服务器会触发 `clientError` 事件。错误消息会包含错误版本数字(`'wrong version number'`).
- 如果你的应用被当做安全客户端，和一个不支持比 SSLv3 更高安全性的方法的服务器通讯，你的连接不会协商成功。这种情况下，你的客户端会触发 `clientError` 事件。错误消息会包含错误版本数字(`'wrong version number'`).

Client-initiated renegotiation attack mitigation

TLS 协议让客户端协商 TLS 会话的某些方法内容。但是，会话协商需要服务器端响应的资源，这回让它成为阻断服务攻击（denial-of-service attacks）的潜在媒介。

为了降低这种情况的发生，重新协商被限制为每10分钟3次。当超出这个界限时，在 tls.TLSSocket (页 0) 实例上会触发错误。这个限制可设置：

- `tls.CLIENT_RENEG_LIMIT`: 重新协商 limit, 默认是 3.
- `tls.CLIENT_RENEG_WINDOW`: 重新协商窗口的时间，单位秒, 默认是 10 分钟.

除非你明确知道自己在干什么，否则不要改变默认值。

要测试你的服务器的话，使用 `openssl s_client -connect address:port` 连接服务器，并敲 `R<CR>`（字母 R 键加回车）几次。

NPN and SNI

NPN (Next Protocol Negotiation 下次协议协商) 和 SNI (Server Name Indication 域名指示) 都是 TLS 握手扩展，运行你：

- NPN - 同一个TLS服务器使用多种协议 (HTTP, SPDY)

- SNI – 同一个TLS服务器使用多个主机名（不同的 SSL 证书）。 certificates.

完全正向保密

"Forward Secrecy (http://en.wikipedia.org/wiki/Perfect_forward_secrecy) " 或 "Perfect Forward Secrecy–完全正向保密" 协议描述了秘钥协商（比如秘钥交换）方法的特点。实际上这意味着及时你的服务器的秘钥有危险，通讯仅有可能被一类人窃听，他们必须设法获的每次会话都会生成的秘钥对。

完全正向保密是通过每次握手时为秘钥协商随机生成密钥对来完成（和所有会话一个 key 相反）。实现这个技术（提供完全正向保密–Perfect Forward Secrecy）的方法被称为 "ephemeral"。

通常目前有2个方法用于完成完全正向保密（Perfect Forward Secrecy）：

- DHE (https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange) – 一个迪菲–赫尔曼密钥交换密钥协议（Diffie Hellman key–agreement protocol）短暂（ephemeral）版本。
- ECDHE (https://en.wikipedia.org/wiki/Elliptic_curve_Diffie%E2%80%93Hellman) – 一个椭圆曲线密钥交换密钥协议（ Elliptic Curve Diffie Hellman key–agreement protocol）短暂（ephemeral）版本。

短暂（ephemeral）方法有性能缺点，因为生成 key 非常耗费资源。

tls.getCiphers()

返回支持的 SSL 密码名数组。

例子：

```
var ciphers = tls.getCiphers();
console.log(ciphers); // ['AES128–SHA', 'AES256–SHA', ...]
```

()

tls.createServer(options[, secureConnectionListener])

创建一个新的 tls.Server (页 0)。参数 `connectionListener` 会自动设置为 secureConnection (页 0) 事件的监听器。参数 `options` 对象有以下可能性：

- `pfx` : 包含私钥，证书和服务器的 CA 证书（PFX 或 PKCS12 格式）字符串或缓存 `Buffer` 。（ `key` , `cert` 和 `ca` 互斥）。

- `key`: 包含服务器私钥（PEM 格式）字符串或缓存 `Buffer` 。（可以是keys的数组）（必传）。
- `passphrase`: 私钥或 pfx 的密码字符串
- `cert`: 包含服务器证书key（PEM 格式）字符串或缓存 `Buffer` 。（可以是certs的数组）（必传）。
- `ca`: 信任的证书（PEM 格式）的字符串/缓存数组。如果忽略这个参数，将会使用"root" CAs，比如 VeriSign。用来授权连接。
- `crl`: 不是 PEM 编码 CRLs（证书撤销列表 Certificate Revocation List）的字符串就是字符串列表.
- `ciphers`: 要使用或排除的密码（cipher）字符串

  为了减轻BEAST attacks (http://blog.ivanristic.com/2011/10/mitigating-the-beast-attack-on-tls.html)，推荐使用这个参数和之后会提到的 `honorCipherOrder` 参数来优化non-CBC 密码（cipher）

  默认： `ECDHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA256:AES128-GCM-SHA256:RC4:HIGH:!MD5:!aNULL` . 格式上更多细节参见 OpenSSL cipher list format documentation (http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT)

  `ECDHE-RSA-AES128-SHA256`，`DHE-RSA-AES128-SHA256` 和 `AES128-GCM-SHA256` 都是 TLS v1.2 密码（cipher），当 node.js 连接 OpenSSL 1.0.1 或更早版本（比如）时使用。注意， `honorCipherOrder` 设置为 enabled 后，现在任然可以和 TLS v1.2 客户端协商弱密码（cipher），

  `RC4` 可作为客户端和老版本 TLS 协议通讯的备用方法。`RC4` 这些年受到怀疑，任何对信任敏感的对象都会考虑其威胁性。国家级别（state-level）的参与者拥有中断它的能力。

  注意: 早些版本的修订建议， `AES256-SHA` 作为可以接受的密码（cipher）.Unfortunately, `AES256-SHA` 是一个 CBC 密码（cipher），容易受到 BEAST attacks (http://blog.ivanristic.com/2011/10/mitigating-the-beast-attack-on-tls.html) 攻击。 *不要* 使用它。
- `ecdhCurve`: 包含用来 ECDH 秘钥交换弧形（curve）名字符串，或者 false 禁用 ECDH。

  默认 `prime256v1` . 更多细节参考 RFC 4492 (http://www.rfc-editor.org/rfc/rfc4492.txt) 。
- `dhparam`: DH 参数文件，用于 DHE 秘钥协商。使用 `openssl dhparam` 命令来创建。如果加载文件失败，会悄悄的抛弃它。
- `handshakeTimeout`: 如果 SSL/TLS 握手事件超过这个参数，会放弃里连接。默认是 120 秒.

  握手超时后， `tls.Server` 对象会触发 `'clientError'` 事件。
- `honorCipherOrder`: 当选择一个密码（cipher）时, 使用服务器配置，而不是客户端的。

虽然这个参数默认不可用，还是推荐你用这个参数，和 ciphers 参数连接使用，减轻 BEAST 攻击。

注意，如果使用了 SSLv2，服务器会发送自己的配置列表给客户端，客户端会挑选密码（cipher）。默认不支持 SSLv2，除非 node.js 配置了 ./configure --with-sslv2 。

- requestCert：如果设为 true，服务器会要求连接的客户端发送证书，并尝试验证证书。默认： false 。
- rejectUnauthorized：如果为 true ，服务器将会拒绝任何不被 CAs 列表授权的连接。仅 requestCert 参数为 true 时这个参数才有效。默认： false 。
- checkServerIdentity(servername, cert)：提供一个重写的方法来检查证书对应的主机名。如果验证失败，返回 error。如果验证通过，返回 undefined 。
- NPNProtocols：NPN 协议的 Buffer 数组（协议需按优先级排序）。
- SNICallback(servername, cb)：如果客户端支持 SNI TLS 扩展会调用这个函数。会传入2个参数： servername 和 cb 。 SNICallback 必须调用 cb(null, ctx) ，其中 ctx 是 SecureContext 实例。（你可以用 tls.createSecureContext(...) 来获取相应的 SecureContext 上下文）。如果 SNICallback 没有提供，将会使用高级的 API（参见下文）.
- sessionTimeout：整数，设定了服务器创建TLS 会话标示符（TLS session identifiers）和 TLS 会话票据（TLS session tickets）后的超时时间（单位：秒）。更多细节参见：SSL_CTX_set_timeout (http://www.openssl.org/docs/ssl/SSL_CTX_set_timeout.html) 。
- ticketKeys：一个 48 字节的 Buffer 实例，由 16 字节的前缀，16 字节的 hmac key，16 字节的 AES key 组成。可用用它来接受 tls 服务器实例上的 tls会话票据（tls session tickets）。

  注意: 自动在集群模块（ cluster module）工作进程间共享。
- sessionIdContext：会话恢复（session resumption）的标示符字符串。如果 requestCert 为 true 。默认值为命令行生成的 MD5 哈希值。否则不提供默认值。
- secureProtocol：SSL 使用的方法，例如， SSLv3_method 强制 SSL 版本为3。可能的值定义于你所安装的 OpenSSL 中的常量SSL_METHODS (http://www.openssl.org/docs/ssl/ssl.html#DEALING_WITH_PROTOCOL_METHODS) 。
- secureOptions：设置服务器配置。例如设置 SSL_OP_NO_SSLv3 可用禁用 SSLv3 协议。所有可用的参数见SSL_CTX_set_options (https://www.openssl.org/docs/ssl/SSL_CTX_set_options.html)

响应服务器的简单例子：

```
var tls = require('tls');
var fs = require('fs');

var options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),

  // This is necessary only if using the client certificate authentication.
  requestCert: true,

  // This is necessary only if the client uses the self-signed certificate.
  ca: [ fs.readFileSync('client-cert.pem') ]
};

var server = tls.createServer(options, function(socket) {
  console.log('server connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  socket.write("welcome!\n");
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, function() {
  console.log('server bound');
});
```

或

```
var tls = require('tls');
var fs = require('fs');

var options = {
  pfx: fs.readFileSync('server.pfx'),

  // This is necessary only if using the client certificate authentication.
  requestCert: true,

};

var server = tls.createServer(options, function(socket) {
  console.log('server connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  socket.write("welcome!\n");
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, function() {
```

```
  console.log('server bound');
});
```

你可以通过 openssl s_client 连接服务器来测试：

```
openssl s_client -connect 127.0.0.1:8000
```

()

tls.connect(options[, callback])

tls.connect(port[, host][, options][, callback])

创建一个新的客户端连接到指定的端口和主机（ port and host ）（老版本 API），或者 options.port 和 options.host （如果忽略 host，默认为 localhost ）。 options 是一个包含以下值得对象：

- host : 客户端需要连接到的主机
- port : 客户端需要连接到的端口
- socket : 在指定的 socket（而非新建）上建立安全连接。如果这个参数有值，将忽略 host 和 port 参数。
- path : 创建 到参数 path 的 unix socket 连接。如果这个参数有值，将忽略 host 和 port 参数。
- pfx : 包含私钥，证书和客户端（ PFX 或 PKCS12 格式）的 CA 证书的字符串或 Buffer 缓存。
- key : 包含客户端（ PEM 格式）的 私钥的字符串或 Buffer 缓存。可以是 keys 数组。
- passphrase : 私钥或 pfx 的密码字符串。
- cert : 包含客户端证书key（PEM 格式）字符串或缓存 Buffer 。（可以是certs的数组）。
- ca : 信任的证书（PEM 格式）的字符串/缓存数组。如果忽略这个参数，将会使用"root" CAs，比如 VeriSign。用来授权连接。
- rejectUnauthorized : 如果为 true ，服务器证书根据 CAs 列表授权列表验证。如果验证失败，触发 'error' 事件； err.code 包含 OpenSSL 错误代码。默认： true 。
- NPNProtocols : NPN 协议的字符串或 Buffer 数组。 Buffer 必须有以下格式 0x05hello0x05world ，第一个字节是下一个协议名字的长度。（传的数组通常非常简单，比如： ['hello', 'world'] ）。
- servername : SNI（域名指示 Server Name Indication） TLS 扩展的服务器名。

- `secureProtocol`: SSL 使用的方法，例如， `SSLv3_method` 强制 SSL 版本为3。可能的值定义于你所安装的 OpenSSL 中的常量SSL_METHODS (http://www.openssl.org/docs/ssl/ssl.html#DEALING_WITH_PROTOCOL_METHODS) 。
- `session`: 一个 `Buffer` 实例, 包含 TLS 会话.

参数 `callback` 添加到 'secureConnect' (页 0) 事件上，其效果如同监听器。

`tls.connect()` 返回一个 tls.TLSSocket (页 0) 对象。

这是一个简单的客户端应答服务器例子:

```
var tls = require('tls');
var fs = require('fs');

var options = {
  // These are necessary only if using the client certificate authentication
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // This is necessary only if the server uses the self-signed certificate
  ca: [ fs.readFileSync('server-cert.pem') ]
};

var socket = tls.connect(8000, options, function() {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', function(data) {
  console.log(data);
});
socket.on('end', function() {
  server.close();
});
```

或

```
var tls = require('tls');
var fs = require('fs');

var options = {
  pfx: fs.readFileSync('client.pfx')
```

```
};

var socket = tls.connect(8000, options, function() {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', function(data) {
  console.log(data);
});
socket.on('end', function() {
  server.close();
});
```

类: tls.TLSSocket

net.Socket (net.html#net_class_net_socket) 实例的封装，取代内部 socket 读写程序，执行透明的输入/输出数据的加密/解密。

new tls.TLSSocket(socket, options)

从现有的 TCP socket 里构造一个新的 TLSSocket 对象。

`socket` 一个 net.Socket (net.html#net_class_net_socket) 的实例

`options` 一个包含以下属性的对象:

- `secureContext` : 来自 `tls.createSecureContext( ... )` 的可选 TLS 上下文对象。
- `isServer` : 如果为 true， TLS socket 将会在服务器模式(server-mode)初始化。
- `server` : 一个可选的 net.Server (net.html#net_class_net_server) 实例
- `requestCert` : 可选, 参见 tls.createSecurePair (页 0)
- `rejectUnauthorized` : 可选, 参见 tls.createSecurePair (页 0)
- `NPNProtocols` : 可选, 参见 tls.createServer (页 295)
- `SNICallback` : 可选, 参见 tls.createServer (页 295)

- `session`: 可选, 一个 `Buffer` 实例, 包含 TLS 会话
- `requestOCSP`: 可选, 如果为 `true` – OCSP 状态请求扩展将会被添加到客户端 hello，并且 `OCSPResponse` 事件将会在 socket 上建立安全通讯前触发。

tls.createSecureContext(details)

创建一个凭证（credentials）对象，包含字典有以下的key：

- `pfx`：包含 PFX 或 PKCS12 加密的私钥，证书和服务器的 CA 证书（PFX 或 PKCS12 格式）字符串或缓存 `Buffer`。（`key`，`cert` 和 `ca` 互斥）。
- `key`：包含 PEM 加密过的私钥的字符串。
- `passphrase`：私钥或 pfx 的密码字符串。
- `cert`：包含 PEM 加密过的证书的字符串。
- `ca`：信任的 PEM 加密过的可信任的证书（PEM 格式）字符串/缓存数组。
- `crl`:PEM 加密过的 CRLs（证书撤销列表）
- `ciphers`：要使用或排除的密码（cipher）字符串。更多格式上的细节参见 http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT
- `honorCipherOrder`：当选择一个密码（cipher）时, 使用服务器配置，而不是客户端的。更多细节参见 `tls` 模块文档。

如果没给 'ca' 细节，node.js 将会使用默认的公开信任的 CAs 列表（参见http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt）。

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized])

创建一个新的安全对（secure pair）对象，包含2个流，其中一个读/写加密过的数据，另外一个读/写明文数据。通常加密端数据来自是从输入的加密数据流，另一端被当做初始加密流。

- `credentials`：来自 tls.createSecureContext(...) 的安全上下文对象。
- `isServer`：是否以服务器/客户端模式打开这个 tls 连接。
- `requestCert`：是否服务器需要连接的客户端发送证书。仅适用于服务端连接。

- rejectUnauthorized :非法证书时，是否服务器需要自动拒绝客户端。启用 requestCert 后，才适用于服务器。

tls.createSecurePair() 返回一个安全对（SecurePair）对象，包含明文 cleartext 和 密文 encrypted 流 。

?注意: cleartext 和 tls.TLSSocket (页 0) 拥有相同的 API。

类: SecurePair

通过 tls.createSecurePair 返回。

事件: 'secure'

一旦安全对（SecurePair）成功建立一个安全连接，安全对（SecurePair）将会触发这个事件。

和检查服务器 'secureConnection' 事件一样，pair.cleartext.authorized 必须检查确认是否适用的证书是授权过的。

类: tls.Server

这是 net.Server 的子类，拥有相同的方法。这个类接受适用 TLS 或 SSL 的加密连接，而不是接受原始 TCP 连接。

事件: 'secureConnection'

function (tlsSocket) {}

新的连接握手成功后回触发这个事件。参数是 tls.TLSSocket (页 0) 实例。它拥有常用的流方法和事件。

socket.authorized 是否客户端被证书（服务器提供）授权。如果 socket.authorized 为 false， socket.authorizationError 是如何授权失败。值得一提的是，依赖于 TLS 服务器的设置，你的未授权连接可能也会被接受。 socket.authorizationError 如何授权失败。值得一提的是，依赖于 TLS 服务器的设置，你的未授权连接可能也会被接受。 socket.npnProtocol 包含选择的 NPN 协议的字符串. socket.servername 包含 SNI 请求的服务器名的字符串。

事件: 'clientError'

`function (exception, tlsSocket) { }`

在安全连接建立前，客户端连接触发 'error' 事件会转发到这里来。

`tlsSocket` 是 tls.TLSSocket (页 0)，错误是从这里触发的。

事件: 'newSession'

`function (sessionId, sessionData, callback) { }`

创建 TLS 会话的时候会触发。可能用来在外部存储器里存储会话。`callback` 必须最后调用，否则没法从安全连接发送/接收数据。

注意： 添加这个事件监听器仅会在连接连接时有效果。

事件: 'resumeSession'

`function (sessionId, callback) { }`

当客户端想恢复之前的 TLS 会话时会触发。事件监听器可能会使用 `sessionId` 到外部存储器里查找，一旦结束会触发 `callback(null, sessionData)`。如果会话不能恢复（比如不存在这个会话），可能会调用 `callback(null, null)`。调用 `callback(err)` 将会终止连接，并销毁 socket。

注意： 添加这个事件监听器仅会在连接连接时有效果。

事件: 'OCSPRequest'

`function (certificate, issuer, callback) { }`

当客户端发送证书状态请求时会触发。你可以解析服务器当前的证书，来获取 OCSP 网址和证书 id，获取 OCSP 响应调用 `callback(null, resp)`，其中 `resp` 是 `Buffer` 实例。 证书（`certificate`）和发行者（`issuer`）都是初级表达式缓存（`Buffer` DER-representations of the primary）和证书的发行者。它可以用来获取 OCSP 证书和 OCSP 终点网址。

可以调用 `callback(null, null)`，表示没有 OCSP 响应。

调用 `callback(err)` 可能会导致调用 `socket.destroy(err)`。

典型流程:

1. 客户端连接服务器，并发送 `OCSPRequest` (通过 ClientHello 里的状态信息扩展)。
2. 服务器收到请求，调用 `OCSPRequest` 事件监听器。
3. 服务器从 `certificate` 或 `issuer` 获取 OCSP 网址，并执行 OCSP request (http://en.wikipedia.org/wiki/OCSP_stapling) 到 CA
4. 服务器 CA 收到 `OCSPResponse` ， 并通过 `callback` 参数送回到客户端
5. 客户端验证响应，并销毁 socket 或 执行握手

注意: 如果证书是自签名的，或者如果发行者不再根证书列表里（你可以通过参数提供一个发行者）。 `issuer` 就可能为 null。

注意： 添加这个事件监听器仅会在连接连接时有效果。

注意： 你可以能想要使用 npm 模块（比如 asn1.js (http://npmjs.org/package/asn1.js) ）来解析证书。

server.listen(port[, host][, callback])

在指定的端口和主机上开始接收连接。如果 `host` 参数没传，服务接受通过 IPv4 地址(`INADDR_ANY`)的直连。

这是异步函数。当服务器已经绑定后回调用最后一个参数 `callback` 。

更多信息参见 `net.Server` 。

server.close()

停止服务器，不再接收新连接。这是异步函数，当服务器触发 `'close'` 事件后回最终关闭。

server.address()

返回绑定的地址，地址家族名和服务器端口。更多信息参见 net.Server.address() (net.html#net_server_address)

server.addContext(hostname, context)

如果客户端请求 SNI 主机名和传入的 `hostname` 相匹配，将会用到安全上下文（secure context）。 `context` 可以包含 `key`，`cert`，`ca` 和/或 `tls.createSecureContext` `options` 参数的其他任何属性。

server.maxConnections

设置这个属性可以在服务器的连接数达到最大值时拒绝连接。

server.connections

当前服务器连接数。

类: CryptoStream

稳定性: 0 – 抛弃. 使用 tls.TLSSocket 替代.

这是一个加密的流

cryptoStream.bytesWritten

底层 socket 写字节访问器（bytesWritten accessor）的代理，将会返回写到 socket 的全部字节数。*包括 TLS 的开销*。

类: tls.TLSSocket

net.Socket (net.html#net_class_net_socket) 实例的封装，透明的加密写数据和所有必须的 TLS 协商。

这个接口实现了一个双工流接口。它包含所有常用的流方法和事件。

事件: 'secureConnect'

新的连接成功握手后回触发这个事件。无论服务器证书是否授权，都会调用监听器。用于用户测试 `tlsSocket.authorized` 看看如果服务器证书已经被指定的 CAs 签名。如果 `tlsSocket.authorized === false` ，可以在 `tlsSoc`

ket.authorizationError 里找到错误。如果使用了 NPN，你可以检 tlsSocket.npnProtocol 获取协商协议（negotiated protocol）。

事件: 'OCSPResponse'

function (response) { }

如果启用 requestOCSP 参赛会触发这个事件。 response 是缓存对象，包含服务器的 OCSP 响应。

一般来说， response 是服务器 CA 签名的对象，它包含服务器撤销证书状态的信息。

tlsSocket.encrypted

静态 boolean 变量，一直是 true 。可以用来区别 TLS socket 和 常规对象。

tlsSocket.authorized

boolean 变量，如果 对等实体证书（ peer's certificate ）被指定的某个 CAs 签名，返回 true ，否则 false 。

tlsSocket.authorizationError

对等实体证书（ peer's certificate ）没有验证通过的原因。当 tlsSocket.authorized === false 时，这个属性才可用。

tlsSocket.getPeerCertificate([detailed])

返回一个代表对等实体证书（ peer's certificate ）的对象。这个返回对象有一些属性和证书内容相对应。如果参数 detailed 是 true ，将会返回包含发行者 issuer 完整链。如果 false ，仅有顶级证书没有发行者 issuer 属性。

例子：

```
{ 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 peer 没有提供证书，返回 `null` 或空对象。

tlsSocket.getCipher()

返回一个对象，它代表了密码名和当前连接的 SSL/TLS 协议的版本。

例子： { name: 'AES256-SHA', version: 'TLSv1/SSLv3' }

更多信息参见http://www.openssl.org/docs/ssl/ssl.html#DEALING_WITH_CIPHERS 里的 SSL_CIPHER_get_name() 和 SSL_CIPHER_get_version() 。

tlsSocket.renegotiate(options, callback)

初始化 TLS 重新协商进程。参数 `options` 可能包含以下内容： `rejectUnauthorized` ，`requestCert` (细节参见 tls.createServer (页 295))。一旦重新协商成（ renegotiation ）功完成，将会执行 `callback(err)` ，其中 `err` 为 `null` 。

注意： 当安全连接建立后，可以用这来请求对等实体证书（ peer's certificate ）。

注意： 作为服务器运行时， `handshakeTimeout` 超时后，socket 将会被销毁。

tlsSocket.setMaxSendFragment(size)

设置最大的 TLS 碎片大小（默认最大值为： `16384` ，最小值为： `512` ）。成功的话，返回 `true` ，否则返回 `false` 。

小的碎片包会减少客户端的缓存延迟：大的碎片直到接收完毕后才能被 TLS 层完全缓存，并且验证过完整性；大的碎片可能会有多次往返，并且可能会因为丢包或重新排序导致延迟。而小的碎片会增加额外的 TLS 帧字节和 CPU 负载，这会减少 CPU 的吞吐量。

tlsSocket.getSession()

返回 ASN.1 编码的 TLS 会话，如果没有协商，会返回。连接到服务器时，可以用来加速握手的建立。

tlsSocket.getTLSTicket()

注意： 仅和客户端 TLS socket 打交道。仅在调试时有用，会话重用是，提供 `session` 参数给 `tls.connect` 。

返回 TLS 会话票据（ticket），或如果没有协商（negotiated），返回 `undefined` 。

tlsSocket.address()

返回绑定的地址，地址家族名和服务器端口。更多信息参见 net.Server.address() (net.html#net_server_address) 。返回三个属性, 比如： `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

tlsSocket.remoteAddress

表示远程 IP 地址（字符串表示），例如： `'74.125.127.100'` 或 `'2001:4860:a005::68'` .

tlsSocket.remoteFamily

表示远程 IP 家族， `'IPv4'` 或 `'IPv6'` .

tlsSocket.remotePort

远程端口（数字表示），列如, `443` .

tlsSocket.localAddress

本地 IP 地址（字符串表示）。

tlsSocket.localPort

本地端口号（数字表示）。

TTY

稳定性: 2 – 不稳定

tty 模块包含 tty.ReadStream 和 tty.WriteStream 类。多数情况下，你不必直接使用这个模块。

当 node 检测到自己正运行于 TTY 上下文时， process.stdin 将会是一个 tty.ReadStream 实例，并且 process.stdout 将会是 tty.WriteStream 实例。检测 node 是否运行在 TTY 上下文的好方法是检测 process.stdout.isTTY：

```
$ node -p -e "Boolean(process.stdout.isTTY)"
true
$ node -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

tty.isatty(fd)

如果 fd 和终端相关联返回 true ，否则返回 false 。

tty.setRawMode(mode)

已经抛弃。使用 tty.ReadStream#setRawMode() （比如 process.stdin.setRawMode() ）替换。

Class: ReadStream

net.Socket 的子类，表示 tty 的可读部分。通常情况，在任何 node 程序里（仅当 isatty(0) 为 true 时）， process.stdin 是 tty.ReadStream 的唯一实例。

rs.isRaw

Boolean 值，默认为 false 。它代表当前 tty.ReadStream 实例的 "raw" 状态。

rs.setRawMode(mode)

mode 需是 true 或 false 。它设定 tty.ReadStream 属性为原始设备或默认。 isRaw 将会设置为结果模式。

Class: WriteStream

`net.Socket` 的子类，代表 tty 的可写部分。通常情况下， `process.stdout` 是 `tty.WriteStream` 唯一实例（仅当 `isatty(1)` 为 true 时）。

ws.columns

TTY 当前 拥有的列数。触发 "resize" 事件时会更新这个值。

ws.rows

TTY 当前 拥有的行数。触发 "resize" 事件时会更新这个值。

Event: 'resize'

`function () {}`

行或列变化时会触发 `refreshSize()` 事件。

```
process.stdout.on('resize', function() {
  console.log('screen size has changed!');
  console.log(process.stdout.columns + 'x' + process.stdout.rows);
});
```

32

UDP/Datagram Sockets

稳定性: 3 – 稳定

调用 `require('dgram')` ，可以使用数据报文 sockets(Datagram sockets)。

重要提醒： `dgram.Socket#bind()` 的行为在 v0.10 做了改动 ，它总是异步的。如果你的代码像下面的一样：

```
var s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

现在需要改为：

```
var s = dgram.createSocket('udp4');
s.bind(1234, function() {
  s.addMembership('224.0.0.114');
});
```

dgram.createSocket(type[, callback])

- `type` 字符串. 'udp4' 或 'udp6'
- `callback` 函数. 附加到 `message` 事件的监听器。可选参数。
- 返回: Socket 对象

创建指定类型的数据报文(datagram) Socket。有效类型是 `udp4` 和 `udp6`

接受一个可选的回调，会被添加为 `message` 的监听事件。

如果你想接收数据报文(datagram)可以调用 `socket.bind()` 。`socket.bind()` 将会绑定到所有接口（"all interfaces"）的随机端口上（ `udp4` 和 `udp6` sockets 都适用）。你可以通过 `socket.address().address` 和 `socket.address().port` 获取地址和端口。

dgram.createSocket(options[, callback])

- `options` 对象
- `callback` 函数. 给 `message` 事件添加事件监听器.
- 返回: Socket 对象

参数 `options` 必须包含 `type` 值(`udp4` 或 `udp6`),或可选的 boolean 值 `reuseAddr` 。

当 `reuseAddr` 为 true 时， `socket.bind()` 将会重用地址，即使另一个进程已经绑定 socket。 `reuseAddr` 默认为 `false` 。

回调函数为可选参数，作为 `message` 事件的监听器。

如果你想接受数据报文(datagram)，可以调用 `socket.bind()` 。`socket.bind()` 将会绑定到所有接口（ "all interfaces" ）地址的随机端口上（ `udp4` 和 `udp6` sockets 都适用 ）。你可以通过 `socket.address().address` 和 `socket.address().port` 获取地址和端口。

Class: dgram.Socket

报文数据 Socket 类封装了数据报文(datagram) 函数。必须通过 `dgram.createSocket(...)` 函数创建。

Event: 'message'

- `msg` 缓存对象. 消息。
- `rinfo` 对象. 远程地址信息。

当 socket 上新的数据报文(datagram)可用的时候，会触发这个事件。`msg` 是一个缓存， `rinfo` 是一个包含发送者地址信息的对象

```
socket.on('message', function(msg, rinfo) {
  console.log('Received %d bytes from %s:%d\n',
              msg.length, rinfo.address, rinfo.port);
});
```

Event: 'listening'

当 socket 开始监听数据报文(datagram)时触发。在 UDP socket 创建时触发。

Event: 'close'

当 socket 使用 `close()` 关闭时触发。在这个 socket 上不会触发新的消息事件。

Event: 'error'

- `exception` Error 对象

当发生错误时触发。

socket.send(buf, offset, length, port, address[, callback])

- `buf` 缓存对象 或 字符串. 要发送的消息。
- `offset` 整数. 消息在缓存中得偏移量。
- `length` 整数. 消息的比特数。
- `port` 整数. 端口的描述。
- `address` 字符串. 目标的主机名或 IP 地址。
- `callback` 函数. 当消息发送完毕的时候调用。可选。

对于 UDP socket，必须指定目标端口和地址。 `address` 参数可能是字符串，它会被 DNS 解析。

如果忽略地址或者地址是空字符串，将使用 `'0.0.0.0'` 或 `'::0'` 替代。依赖于网络配置，这些默认值有可能行也可能不行。

如果 socket 之前没被调用 `bind` 绑定，则它会被分配一个随机端口并绑定到所有接口（ "all interfaces" ）地址(`udp4` sockets 的 `'0.0.0.0'` ， `udp6` sockets 的 `'::0'`)

回调函数可能用来检测 DNS 错误，或用来确定什么时候重用 `buf` 对象。注意，DNS 查询会导致发送tick延迟。通过回调函数能确认数据报文(datagram)是否已经发送的

考虑到多字节字符串情况，偏移量和长度是字节长度byte length (页 0)，而不是字符串长度。

下面的例子是在 `localhost` 上发送一个 UDP 包给随机端口：

```
var dgram = require('dgram');
var message = new Buffer("Some bytes");
var client = dgram.createSocket("udp4");
client.send(message, 0, message.length, 41234, "localhost", function(err) {
  client.close();
});
```

关于 UDP 数据报文(datagram) 尺寸

`IPv4/v6` 数据报文(datagram)的最大长度依赖于 `MTU` (*Maximum Transmission Unit*)和 `Payload Length` 的长度。

- `Payload Length` 内容为 16 位宽，它意味着 Payload 的最大字节说不超过 64k，其中包括了头信息和数据（65,507 字节 = 65,535 − 8 字节 UDP 头 − 20 字节 IP 头）;对于环回接口（loopback interfaces）这是真的，但对于多数主机和网络来说不太现实。
- `MTU` 能支持数据报文(datagram)的最大值（以目前链路层技术来说）。对于任何连接，`IPv4` 允许的最小值为 `68` 的 `MTU`，推荐值为 `576` （通常推荐作拨号应用的 `MTU`）,无论他们是完整接收还是碎片接收。

  对于 `IPv6`， `MTU` 的最小值为 `1280` 字节，最小碎片缓存大小为 `1500` 字节。16 字节实在是太小，所以目前链路层一般最小 `MTU` 大小为 `1500` 。

我们不可能知道一个包可能进过的每个连接的MTU。通常发送一个超过接收端 MTU 大小的数据报文(datagram)会失效。（数据包会被悄悄的抛弃，不会通知发送端数据包没有到达接收端）。

socket.bind(port[, address][, callback])

- `port` 整数
- `address` 字符串, 可选
- `callback` 没有参数的函数, 可选。绑定时会调用回调。

对于 UDP socket，在一个端口和可选地址上监听数据报文(datagram)。如果没有指定地点，系统将会参数监听所有的地址。绑定完毕后，会触发 "listening" 事件，并会调用传入的回调函数。指定监听事件和回调函数非常有用。

一个绑定了的数据报文 socket 会保持 node 进程运行来接收数据。

如果绑定失败，会产生错误事件。极少数情况（比如绑定一个关闭的 socket）。这个方法会抛出一个错误。

以下是 UDP 服务器监听端口 41234 的例子：

```
var dgram = require("dgram");

var server = dgram.createSocket("udp4");

server.on("error", function (err) {
  console.log("server error:\n" + err.stack);
  server.close();
});

server.on("message", function (msg, rinfo) {
  console.log("server got: " + msg + " from " +
```

```
    rinfo.address + ":" + rinfo.port);
});

server.on("listening", function () {
  var address = server.address();
  console.log("server listening " +
      address.address + ":" + address.port);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

socket.bind(options[, callback])

- `options` {对象} – 必需. 有以下的属性:
  - `port` {Number} – 必需.
  - `address` {字符串} – 可选.
  - `exclusive` {Boolean} – 可选.
- `callback` {函数} – 可选.

`options` 的可选参数 `port` 和 `address`，以及可选参数 `callback`，好像在调用 socket.bind(port, [address], [callback]) (页 0)。

如果 `exclusive` 是 `false` （默认），集群进程将会使用相同的底层句柄，允许连接处理共享的任务。当 `exclusive` 为 `true` 时，句柄不会共享，尝试共享端口也会失败。监听 `exclusive` 端口的例子如下：

```
socket.bind({
  address: 'localhost',
  port: 8000,
  exclusive: true
});
```

socket.close()

关闭底层 socket 并且停止监听数据。

socket.address()

返回一个包含套接字地址信息的对象。对于 UDP socket，这个对象会包含 `address` ，`family` 和 `port` 。

socket.setBroadcast(flag)

- `flag` Boolean

设置或清除 `SO_BROADCAST` socket 选项。设置这个选项后，UDP 包可能会发送给一个本地的接口广播地址。

socket.setTTL(ttl)

- `ttl` 整数

设置 `IP_TTL` socket 选项。 TTL 表示生存时间（Time to Live），但是在这个上下文中它指的是报文允许通过的 IP 跃点数。各个转发报文的路由器或者网关都会递减 TTL。如果 TTL 被路由器递减为0，则它将不会被转发。改变 TTL 的值通常用于网络探测器或多播。

`setTTL()` 的参数为 1 到 255 的跃点数。多数系统默认值为 64.

socket.setMulticastTTL(ttl)

- `ttl` 整数

设置 `IP_MULTICAST_TTL` socket 选项. TTL 表示生存时间（Time to Live），但是在这个上下文中它指的是报文允许通过的 IP 跃点数。 各个转发报文的路由器或者网关都会递减 TTL。如果 TTL 被路由器递减为0，则它将不会被转发。改变 TTL 的值通常用于网络探测器或多播。

`setMulticastTTL()` 的参数为 1 到 255 的跃点数。多数系统默认值为 1.

socket.setMulticastLoopback(flag)

- `flag` Boolean

设置或清空 `IP_MULTICAST_LOOP` socket 选项。设置完这个选项后，当该选项被设置时，组播报文也会被本地接口收到。

socket.addMembership(multicastAddress[, multicastInterface])

- `multicastAddress` 字符串

- `multicastInterface` 字符串, 可选

告诉内核加入广播组，选项为 `IP_ADD_MEMBERSHIP` socket

如果没有指定 `multicastInterface`，操作系统会给所有可用的接口添加关系。

socket.dropMembership(multicastAddress[, multicastInterface])

- `multicastAddress` 字符串
- `multicastInterface` 字符串, 可选

和 `addMembership` 相反 – 用 `IP_DROP_MEMBERSHIP` 选项告诉内核离开广播组 。 如果没有指定 `multicastInterface`，操作系统会移除所有可用的接口关系。

socket.unref()

在 socket 上调用 `unref` 允许程序退出，如果这是在事件系统中唯一的活动 socket。如果 socket 已经 `unref`，再次调用 `unref` 将会无效。

socket.ref()

和 `unref` 相反，如果这是唯一的 socket，在一个之前被 unref 了的 socket 上调用 ref 将不会让程序退出（缺省行为）。如果一个 socket 已经被 ref，则再次调用 ref 将会无效。

33

URL

稳定性: 3 – 稳定

这个模块包含分析和解析 URL 的工具。调用 `require('url')` 来访问模块。

解析 URL 对象有以下内容，依赖于他们是否在 URL 字符串里存在。任何不在 URL 字符串里的部分，都不会出现在解析对象里。例子如下：

`'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- `href`: 准备解析的完整的 URL，包含协议和主机（小写）。

  例子： `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- `protocol`: 请求协议, 小写.

  例子： `'http:'`

- `slashes`: 协议要求的斜杠（冒号后）

  例子： true 或 false

- `host`: 完整的 URL 小写 主机部分，包含端口信息。

  例子： `'host.com:8080'`

- `auth`: url 中的验证信息。

  例子： `'user:pass'`

- `hostname`: 域名中的小写主机名

  例子： `'host.com'`

- `port`: 主机的端口号

  例子： `'8080'`

- `pathname`: URL 中的路径部分，在主机名后，查询字符前，包含第一个斜杠。

  例子： `'/p/a/t/h'`

- `search`: URL 中得查询字符串，包含开头的问号

  例子： `'?query=string'`

- `path`： `pathname` 和 `search` 连在一起

例子： '/p/a/t/h?query=string'

- query : 查询字符串中得参数部分，或者使用 querystring.parse() 解析后返回的对象。

  例子： 'query=string' or {'query':'string'}

- hash : URL 的 “#” 后面部分（包括 # 符号）

  例子： '#hash'

URL 模块提供了以下方法：

()

url.parse(urlStr[, parseQueryString][, slashesDenoteHost])

输入 URL 字符串，返回一个对象。

第二个参数为 true 时，使用 querystring 来解析查询字符串。如果为 true ， query 属性将会一直赋值为对象，并且 search 属性将会一直是字符串(可能为空)。默认为 false 。

第三个参数为 true ，把 //foo/bar 当做 { host: 'foo', pathname: '/bar' } ，而不是 { pathname: '//foo/bar' } 。默认为 false 。

url.format(urlObj)

输入一个解析过的 URL 对象，返回格式化过的字符串。

格式化的工作流程：

- href 会被忽略
- protocol 无论是否有末尾的 : (冒号)，会同样的处理
  - http , https , ftp , gopher , file 协议会被添加后缀 ://
  - mailto , xmpp , aim , sftp , foo , 等协议添加后缀 :
- slashes 如果协议需要 :// ，设置为 true
  - 仅需对之前列出的没有斜杠的协议，比如议 mongodb://localhost:8000/
- auth 如果出现将会使用.

- `hostname` 仅在缺少 `host` 时使用
- `port` 仅在缺少 `host` 时使用
- `host` 用来替换 `hostname` 和 `port`
- `pathname` 无论结尾是否有 / 将会同样处理
- `search` 将会替代 query属性
    - 无论前面是否有 / 将会同样处理
- `query` (对象; 参见 `querystring`) 如果没有 search,将会使用
- `hash` 无论前面是否有#，都会同样处理

url.resolve(from, to)

给一个基础 URL， href URL，如同浏览器一样的解析它们可以带上锚点，例如：

```
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'
```

34

实用工具

稳定性: 4 – 锁定

这些函数都在 'util' 模块里。使用 require('util') 来访问他们。

util 模块原先设计的初衷是用来支持 node 的内部 API 的。这里的很多的函数对你的程序来说都非常有用。如果你觉得这些函数不能满足你的要求，那你可以写自己的工具函数。我们不希望 'util' 模块里添加对于 node 内部函数无用的扩展。

util.debuglog(section)

- section {字符串} 被调试的程序节点部分
- Returns: {Function} 日志函数

用来创建一个有条件的写到 stderr 的函数（基于 NODE_DEBUG 环境变量）。如果 section 出现在环境变量里，返回函数将会和 console.error() 类似。否则，返回一个空函数。

例如:

```
javascript
var debuglog = util.debuglog('foo');

var bar = 123;
debuglog('hello from foo [%d]', bar);
```

如果这个程序以 NODE_DEBUG=foo 的环境运行，将会输出:

```
FOO 3245: hello from foo [123]
```

3245 是进程 ID。如果没有运行在这个环境变量里，将不会打印任何东西。

可以用逗号切割多个 NODE_DEBUG 环境变量。例如: NODE_DEBUG=fs,net,tls 。

util.format(format[, ...])

使用第一个参数返回一个格式化的字符串，类似 printf 。

第一个参数是字符串，它包含 0 或更多的占位符。每个占位符被替换成想要参数转换的值。支持的占位符包括:

- %s – 字符串.
- %d – 数字 (整数和浮点数).
- %j – JSON. 如果参数包含循环引用，将会用字符串替换R

- `%%` – 单独一个百分号（`'%'`）。不会消耗一个参数。

如果占位符没有包含一个相应的参数，占位符不会被替换。

```
util.format('%s:%s', 'foo'); // 'foo:%s'
```

如果参数超过占位符，多余的参数将会用 `util.inspect()` 转换成字符串，并拼接在一起，用空格隔开。

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

如果第一个参数不是格式化字符串，那么 `util.format()` 会返回所有参数拼接成的字符串（空格分割）。每个参数都会用 `util.inspect()` 转换成字符串。

```
util.format(1, 2, 3); // '1 2 3'
```

util.log(string)

在 `stdout` 输出并带有时间戳.

```
require('util').log('Timestamped message.');
```

util.inspect(object[, options])

返回一个对象的字符串表现形式，在代码调试的时候非常有用。

通过加入一些可选选项，来改变对象的格式化输出形式：

- `showHidden` – 如果为 `true`，将会显示对象的不可枚举属性。默认为 `false`。
- `depth` – 告诉 `inspect` 格式化对象时递归多少次。这在格式化大且复杂对象时非常有用。默认为 `2`。如果想无穷递归的话，传 `null`。
- `colors` – 如果为 `true`, 输出内容将会格式化为有颜色的代码。默认为 `false`， 颜色可以自定义，参见下文。
- `customInspect` – 如果为 `false`, 那么定义在被检查对象上的inspect(depth, opts) 方法将不会被调用。默认为true。

检查 `util` 对象上所有属性的例子：

```
var util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

当被调用的时候，参数值可以提供自己的自定义inspect(depth, opts)方法。该方法会接收当前的递归检查深度，以及传入util.inspect()的其他参数。

自定义 util.inspect 颜色

util.inspect 通过 util.inspect.styles 和 util.inspect.colors 对象，自定义全局的输出颜色，

util.inspect.styles 和 util.inspect.colors 组成风格颜色的一对映射。

高亮风格和他们的默认值： * 数字 (黄色) * boolean (黄色) * 字符串 (绿色) * date (洋红) * regexp (红色) * null (粗体) * undefined (斜体) * special – (青绿色) * name (内部用，不是风格)

预定义的颜色为: white , 斜体 , black , blue , cyan , 绿色 , 洋红 , 红色 和 黄色 . 以及 粗体 , 斜体 , 下划线 和 反选 风格.

对象上德自定义 inspect() 函数

对象也能自定义 inspect(depth) 函数， 当使用util.inspect()检查该对象的时候，将会执行对象自定义的检查方法:

```
var util = require('util');

var obj = { name: 'nate' };
obj.inspect = function(depth) {
  return '{' + this.name + '}';
};

util.inspect(obj);
  // "{nate}"
```

你可以返回另外一个对象，返回的字符串会根据返回的对象格式化。这和 JSON.stringify() 的工作流程类似。 You may also return another Object entirely, and the returned 字符串 will be formatted according to the returned Object. This is similar to how JSON.stringify() works:

```
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.isArray(object)

Array.isArray 的内部别名。

如果参数 "object" 是数组，返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isRegExp(object)

如果参数 "object" 是 `RegExp` 返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

如果参数 "object" 是 `Date` 返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isDate(new Date())
  // true
util.isDate(Date())
```

```
 // false (without 'new' returns a String)
util.isDate({})
 // false
```

util.isError(object)

如果参数 "object" 是 `Error` 返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isError(new Error())
 // true
util.isError(new TypeError())
 // true
util.isError({ name: 'Error', message: 'an error occurred' })
 // false
```

util.inherits(constructor, superConstructor)

从一个构造函数constructor (https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/constructor) 继承原型方法到另一个。构造函数的原型将被设置为一个新的从超类（ `superConstructor` ）创建的对象。

通过 `constructor.super_` 属性可以访问 `superConstructor` 。

```
var util = require("util");
var events = require("events");

function MyStream() {
    events.EventEmitter.call(this);
}

util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
    this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); // true
console.log(MyStream.super_ === events.EventEmitter); // true
```

```
stream.on("data", function(data) {
    console.log('Received data: "' + data + '"');
})
stream.write("It works!"); // Received data: "It works!"
```

util.deprecate(function, string)

标明该方法不要再使用。

```
exports.puts = exports.deprecate(function() {
  for (var i = 0, len = arguments.length; i < len; ++i) {
    process.stdout.write(arguments[i] + '\n');
  }
}, 'util.puts: Use console.log instead')
```

返回一个修改过的函数，默认情况下仅警告一次。如果设置了 `--no-deprecation` 该函数不做任何事。如果设置了 `--throw-deprecation` ，如果使用了该 API 应用将会抛出异常

util.debug(string)

```
稳定性: 0 - 抛弃: 使用 console.error() 替换。
```

`console.error` 的前身。

util.error([...])

```
稳定性: 0 - 抛弃: 使用 console.error() 替换。
```

`console.error` 的前身。

util.puts([...])

```
稳定性: 0 - 抛弃:使用 console.log() 替换。
```

`console.log` 的前身。

util.print([...])

稳定性: 0 – 抛弃: 使用 console.log() 替换。

`console.log` 的前身。

util.pump(readableStream, writableStream[, callback])

稳定性: 0 – 抛弃: Use readableStream.pipe(writableStream)

`stream.pipe` 的前身。

35

虚拟机

```
稳定性: 3 – 稳定
```

可以通过以下方法访问该模块:

```
var vm = require('vm');
```

JavaScript 可以立即编译立即执行，也可以编译，保存，之后再运行。

vm.runInThisContext(code[, options])

`vm.runInThisContext()` 对 参数 `code` 编译，运行并返回结果。 运行的代码没有权限访问本地作用域（local scope），但是可以访问全局对象。

使用 `vm.runInThisContext` 和 `eval` 方法运行同样代码的例子：

```
var localVar = 'initial value';

var vmResult = vm.runInThisContext('localVar = "vm";');
console.log('vmResult: ', vmResult);
console.log('localVar: ', localVar);

var evalResult = eval('localVar = "eval";');
console.log('evalResult: ', evalResult);
console.log('localVar: ', localVar);

// vmResult: 'vm', localVar: 'initial value'
// evalResult: 'eval', localVar: 'eval'
```

`vm.runInThisContext` 没有访问本地作用域，所以没有改变 `localVar`。 `eval` 范围了本地作用域，所以改变了 `localVar`。

`vm.runInThisContext` 用起来很像间接调用 `eval`，比如 `(0,eval)('code')`。但是， `vm.runInThisContext` 也包含以下选项:

- `filename`: 允许更改显示在站追踪（stack traces）的文件名。
- `displayErrors`: 是否在 stderr 上打印错误，抛出异常的代码行高亮显示。会捕获编译时的语法错误，和执行时抛出的错误。默认为 `true`。
- `timeout`: 中断前代码执行的毫秒数。如果执行终止，将会抛出错误。

vm.createContext([sandbox])

如果参数 `sandbox` 不为空，调用 `vm.runInContext` 或 `script.runInContext` 时可以调用沙箱的上下文。以此方式运行的脚本， `sandbox` 是全局对象，它保留自己的属性同时拥有标准全局对象（ global object (http://es5.github.io/#x15.1) ）拥有的内置对象和函数。

如果参数 sandbox 对象为空，返回一个可用的新且空的上下文相关的沙盒对象。

这个函数对于创建一个可运行多脚本的沙盒非常有用。比如，在模拟浏览器的时候可以使用该函数创建一个用于表示 window 全局对象的沙箱，并将所有 `<script>` 标签放入沙箱执行。

vm.isContext(sandbox)

沙箱对象是否已经通过调用 `vm.createContext` 上下文化。

vm.runInContext(code, contextifiedSandbox[, options])

`vm.runInContext` 编译代码，运行在 `contextifiedSandbox` 并返回结果。运行代码不能访问本地域。 `contextifiedSandbox` 对象必须通过 `vm.createContext` 上下文化； `code` 会通过全局变量使用它。

`vm.runInContext` 和 `vm.runInThisContext` 参数相同。

在同一个上下文中编译并执行不同的脚本，例子：

```
var util = require('util');
var vm = require('vm');

var 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，执行不被信任的代码是需要技巧且要非常的小心。 `vm.runInContext` 非常有用，不过想要安全些，最好还是在独立的进程里运行不被信任的代码。

vm.runInNewContext(code[, sandbox][, options])

`vm.runInNewContext` 编译代码, 如果提供了 sandbox ，则将 sandbox 上下文化，否则创建一个新的上下文化过的沙盒，将沙盒作为全局变量运行代码并返回结果。

`vm.runInNewContext` 和 `vm.runInThisContext` 参数相同。

编译并执行代码，增加全局变量值，并设置一个新的。这些全局变量包含在一个新的沙盒里。

```
var util = require('util');
var vm = require('vm'),

var 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，执行不被信任的代码是需要技巧且要非常的小心。`vm.runInNewContext` 非常有用，不过想要安全些，最好还是在独立的进程里运行不被信任的代码。

vm.runInDebugContext(code)

`vm.runInDebugContext` 在 V8 的调试上下文中编译并执行。最主要的应用场景是获得 V8 调试对象访问权限。

```
var Debug = vm.runInDebugContext('Debug');
Debug.scripts().forEach(function(script) { console.log(script.name); });
```

注意，调试上下文和对象内部绑定到 V8 的调试实现里，并可能在没有警告时改变（或移除）。

可以通过 `--expose_debug_as=` 开关暴露调试对象。

Class: Script

包含预编译脚本的类，并在指定的沙盒里执行。

new vm.Script(code, options)

创建一个新的脚本编译代码，但是不运行。使用被创建的 `vm.Script` 来表示编译完的代码。这个代码可以使用以下的方法调用多次。返回的脚本没有绑定到任何全局变量。在运行前绑定，执行后释放。

创建脚本的选项有:

- `filename`: 允许更改显示在站追踪（stack traces）的文件名。
- `displayErrors`: 是否在 stderr 上打印错误，抛出异常的代码行高亮显示。只会捕获编译时的语法错误，执行时抛出的错误由脚本的方法的选项来控制。默认为 `true`。

script.runInThisContext([options])

和 `vm.runInThisContext` 类似，只是作为 `Script` 脚本对象的预编译方法。`script.runInThisContext` 执行编译过的脚本并返回结果。被运行的代码没有本地作用域访问权限，但是拥有权限访问全局对象。

以下例子，使用 `script.runInThisContext` 编译代码一次，并运行多次:

```
var vm = require('vm');

global.globalVar = 0;

var script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' });

for (var i = 0; i < 1000; ++i) {
  script.runInThisContext();
}

console.log(globalVar);

// 1000
```

所运行的代码选项:

- `displayErrors`: 是否在 stderr 上打印错误，抛出异常的代码行高亮显示。仅适用于执行时抛出的错误。不能创建一个语法错误的 `Script` 实例，因为构造函数会抛出。
- `timeout`: 中断前代码执行的毫秒数。如果执行终止，将会抛出错误。

script.runInContext(contextifiedSandbox[, options])

和 `vm.runInContext` 类似，只是作为预编译的 `Script` 对象方法。`script.runInContext` 运行脚本（在 `contextifiedSandbox` 中编译）并返回结果。运行的代码没有权限访问本地域。

`script.runInContext` 的选项和 `script.runInThisContext` 类似。

例子: 编译一段代码，并执行多次，这段代码实现了一个全局变量的自增，并创建一个新的全局变量。这些全局变量保存在沙盒里。

```
var util = require('util');
var vm = require('vm');

var sandbox = {
  animal: 'cat',
  count: 2
};

var script = new vm.Script('count += 1; name = "kitty"');

for (var i = 0; i < 10; ++i) {
  script.runInContext(sandbox);
}

console.log(util.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

注意，执行不被信任的代码是需要技巧且要非常的小心。`script.runInContext` 非常有用，不过想要安全些，最好还是在独立的进程里运行不被信任的代码。

script.runInNewContext([sandbox][, options])

和 `vm.runInNewContext` 类似，只是作为预编译的 `Script` 对象方法。若提供 sandbox 则 script.runInNewContext 将 sandbox 上下文化，若未提供，则创建一个新的上下文化的沙箱。

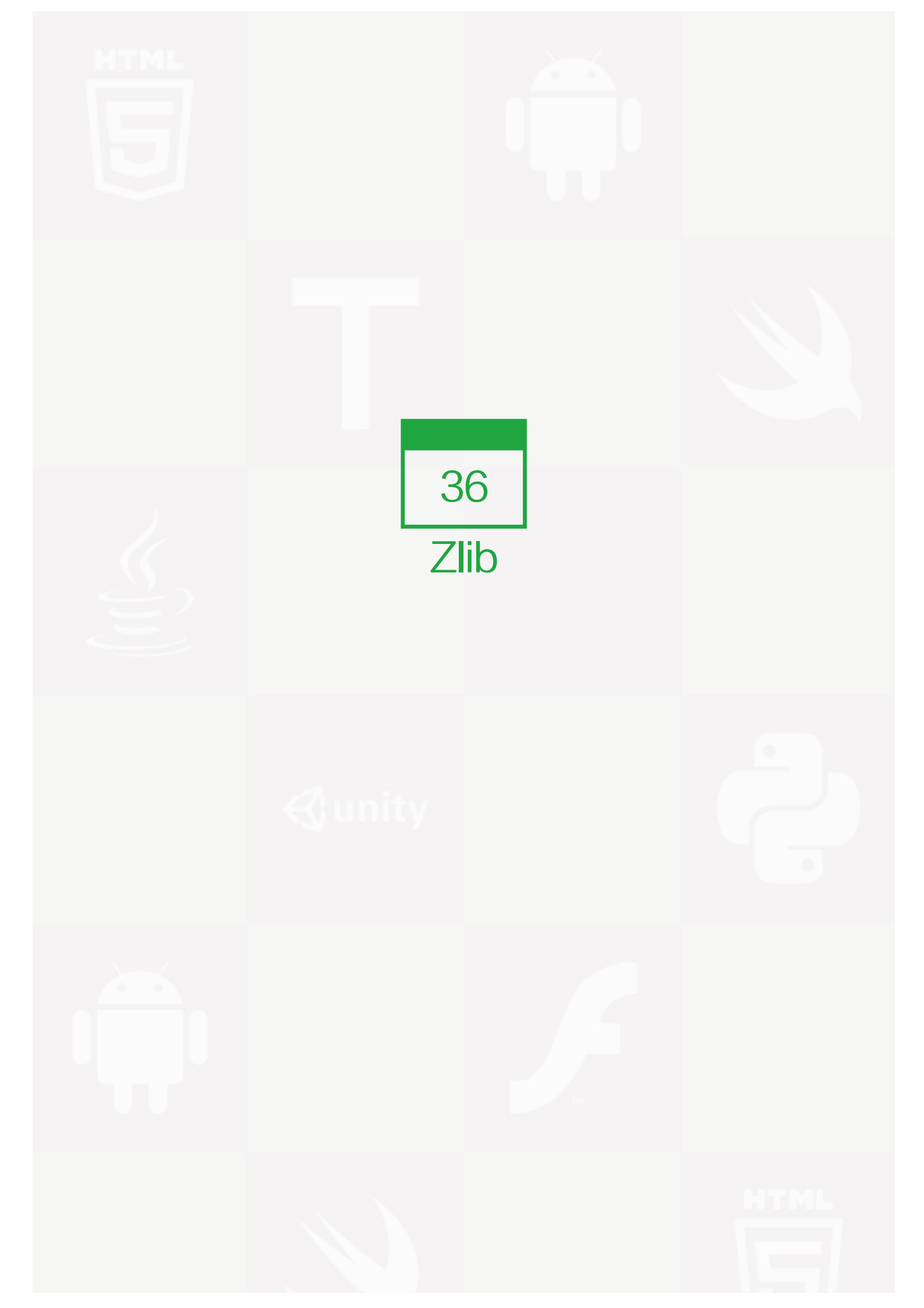
`script.runInNewContext` 和 `script.runInThisContext` 的参数类似。

例子: 编译代码（设置了一个全局变量）并在不同的上下文里执行多次。这些全局变量会被保存在沙箱中。

```
var util = require('util');
var vm = require('vm');
```

```
var sandboxes = [{}, {}, {}];

var script = new vm.Script('globalVar = "set"');

sandboxes.forEach(function (sandbox) {
  script.runInNewContext(sandbox);
});

console.log(util.inspect(sandboxes));

// [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'set' }]
```

注意，执行不被信任的代码是需要技巧且要非常的小心。`script.runInNewContext` 非常有用，不过想要安全些，最好还是在独立的进程里运行不被信任的代码。



36

Zlib

```
稳定性: 3 – 文档
```

可以通过以下方式访问这个模块:

```
var zlib = require('zlib');
```

这个模块提供了对 Gzip/Gunzip, Deflate/Inflate, 和 DeflateRaw/InflateRaw 类的绑定。每个类都有相同的参数和可读/写的流。

例子

压缩/解压缩一个文件，可以通过倒流（piping）一个 fs.ReadStream 到 zlib 流里来，再到一个 fs.fs.WriteStream.

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

一步压缩/解压缩数据可以通过一个简便方法来实现。

```
var input = '.................................';
zlib.deflate(input, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString('base64'));
  }
});

var buffer = new Buffer('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString());
  }
});
```

要在一个 HTTP 客户端或服务器中使用这个模块，可以在请求时使用 accept-encoding (http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3)，响应时使用 content-encoding (http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.11) 头。

注意: 这些例子只是简单展示了基本概念。 Zlib 编码可能消耗非常大，并且结果可能要被缓存。更多使用 zlib 相关的速度/内存/压缩的权衡选择细节参见后面的 Memory Usage Tuning (页 0)。

```
// client request example
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
var request = http.get({ host: 'izs.me',
                         path: '/',
                         port: 80,
                         headers: { 'accept-encoding': 'gzip,deflate' } });
request.on('response', function(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;
    默认:
      response.pipe(output);
      break;
  }
});

// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffer.
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
http.createServer(function(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);
```

zlib.createGzip([options])

根据参数 options (页 0) 返回一个新的 Gzip (页 0) 对象。

zlib.createGunzip([options])

根据参数 options (页 0) 返回一个新的 Gunzip (页 0) 对象。

zlib.createDeflate([options])

根据参数 options (页 0) 返回一个新的 Deflate (页 0) 对象。

zlib.createInflate([options])

根据参数 options (页 0) 返回一个新的 Inflate (页 0) 对象。

zlib.createDeflateRaw([options])

根据参数 options (页 0) 返回一个新的 DeflateRaw (页 0) 对象。

zlib.createInflateRaw([options])

根据参数 options (页 0) 返回一个新的 InflateRaw (页 0) 对象。

zlib.createUnzip([options])

根据参数 options (页 0) 返回一个新的 Unzip (页 0) 对象。

Class: zlib.Zlib

这个类未被 `zlib` 模块导出。之所以写在这，是因为这是压缩/解压缩类的基类。

zlib.flush([kind], callback)

参数 `kind` 默认为 `zlib.Z_FULL_FLUSH` 。

刷入缓冲数据。不要轻易调用这个方法，过早的刷会对压缩算法产生负面影响。

zlib.params(level, strategy, callback)

动态更新压缩基本和压缩策略。仅对 deflate 算法有效。

zlib.reset()

重置压缩/解压缩为默认值。仅适用于 inflate 和 deflate 算法。

Class: zlib.Gzip

使用 gzip 压缩数据。

Class: zlib.Gunzip

使用 gzip 解压缩数据。

Class: zlib.Deflate

使用 deflate 压缩数据。

Class: zlib.Inflate

解压缩 deflate 流。

Class: zlib.DeflateRaw

使用 deflate 压缩数据，不需要拼接 zlib 头。

Class: zlib.InflateRaw

解压缩一个原始 deflate 流。

Class: zlib.Unzip

通过自动检测头解压缩一个 Gzip- 或 Deflate-compressed 流。

简便方法

所有的这些方法第一个参数为字符串或缓存，第二个可选参数可以供 zlib 类使用，回调函数为 `callback(error, result)` 。

每个方法都有一个 `*Sync` 伴随方法，它接收相同参数，不过没有回调。

zlib.deflate(buf[, options], callback)

zlib.deflateSync(buf[, options])

使用 Deflate 压缩一个字符串。

zlib.deflateRaw(buf[, options], callback)

zlib.deflateRawSync(buf[, options])

使用 DeflateRaw 压缩一个字符串。

zlib.gzip(buf[, options], callback)

zlib.gzipSync(buf[, options])

使用 Gzip 压缩一个字符串。

zlib.gunzip(buf[, options], callback)

zlib.gunzipSync(buf[, options])

使用 Gunzip 解压缩一个原始的 Buffer。

zlib.inflate(buf[, options], callback)

zlib.inflateSync(buf[, options])

使用 Inflate 解压缩一个原始的 Buffer。

zlib.inflateRaw(buf[, options], callback)

zlib.inflateRawSync(buf[, options])

使用 InflateRaw 解压缩一个原始的 Buffer。

zlib.unzip(buf[, options], callback)

zlib.unzipSync(buf[, options])

使用 Unzip 解压缩一个原始的 Buffer。

Options

每个类都有一个选项对象。所有选项都是可选的。

注意：某些选项仅在压缩时有用，解压缩时会被忽略。

- flush (默认： `zlib.Z_NO_FLUSH`)
- chunkSize (默认： 16*1024)
- windowBits
- level (仅压缩有效)
- memLevel (仅压缩有效)
- strategy (仅压缩有效)
- dictionary (仅 deflate/inflate 有效, 默认为空字典)

参见 `deflateInit2` 和 `inflateInit2` 的描述，它们位于http://zlib.net/manual.html#Advanced。

使用内存调优

来自 `zlib/zconf.h` ,修改为 node's 的用法:

deflate 的内存需求（单位：字节）：

```
(1 << (windowBits+2)) +  (1 << (memLevel+9))
```

windowBits=15 的 128K 加 memLevel = 8 的 128K（缺省值），加其他对象的若干 KB。

例如，如果你想减少默认的内存需求（从 256K 减为 128k），设置选项：

```
{ windowBits: 14, memLevel: 7 }
```

当然这通常会降低压缩等级。

inflate 的内存需求（单位：字节）：

```
1 << windowBits
```

windowBits=15 (默认值)32K 加其他对象的若干 KB。

这是除了内部输出缓冲外 chunkSize 的大小，缺省为 16K

影响 zlib 的压缩速度最大因素为 `level` 压缩级别。 `level` 越大，压缩率越高，速度越慢， `level` 越小，压缩率越小，速度会更快。

通常来说，使用更多的内存选项，意味着 node 必须减少对 zlib 掉哟过，因为可以在一个 `write` 操作里可以处理更多的数据。所以，这是另一个影响速度和内存使用率的因素，

常量

所有常量定义在 zlib.h ，也定义在 `require('zlib')` 。

通常的操作，基本用不到这些常量。写到文档里是想你不会对他们的存在感到惊讶。这个章节基本都来自 zlib documentation (http://zlib.net/manual.html#Constants) 。更多细节参见 http://zlib.net/manual.html#Constants。

允许 flush 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`

压缩/解压缩函数的返回值。负数代表错误，正数代表特殊但正常的事件：

- `zlib.Z_OK`
- `zlib.Z_STREAM_END`
- `zlib.Z_NEED_DICT`
- `zlib.Z_ERRNO`
- `zlib.Z_STREAM_ERROR`
- `zlib.Z_DATA_ERROR`
- `zlib.Z_MEM_ERROR`
- `zlib.Z_BUF_ERROR`

- zlib.Z_VERSION_ERROR

压缩级别：

- zlib.Z_NO_COMPRESSION
- zlib.Z_BEST_SPEED
- zlib.Z_BEST_COMPRESSION
- zlib.Z_DEFAULT_COMPRESSION

压缩策略：

- zlib.Z_FILTERED
- zlib.Z_HUFFMAN_ONLY
- zlib.Z_RLE
- zlib.Z_FIXED
- zlib.Z_DEFAULT_STRATEGY

data_type 字段的可能值：

- zlib.Z_BINARY
- zlib.Z_TEXT
- zlib.Z_ASCII
- zlib.Z_UNKNOWN

deflate 的压缩方法：

- zlib.Z_DEFLATED

初始化 zalloc, zfree, opaque：

- zlib.Z_NULL

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/nodejs/