NodeJS 中文文档 V0.2.3

感谢老狗、Tytsim、Zarknight、Zbm2001、LemonHall、利奥、元元、Seasontop、Frank、魔力鸟以及更多没有留下姓名的同志们。

Synopsis 总述

使用 node 实现的 web 服务器示例，它返回'Hello World'：

```
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/');
```

将上述代码保存为 example.js，并使用如下命令执行这个服务程序：

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

文档中的其他示例，均可以类似的方式执行。

Standard Modules 标准模块

node 附带了一些模块，这些模板都被编译进了 node 二进制文件中，其中大部分将在下文介绍。 使用这些模块最常见的方法是调用“require('name')”并将返回值赋给一个和模块同名的局部变量。

例如：

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

同样可以使用其他模块扩展 node。 详情参看"模块"章节。

Buffers 缓存对象

纯粹的 Javascript 对 Unicode 很友好，但是操作二进制数据就不怎么在行了。处理 TCP 数据流或者文件时，必须要操作二进制数据流。 node 提供了一些方法来创建、操作和接收二进制数据流。

原始的数据保存在 Buffer 类的实例中。Buffer 类似于一个整数数组，不同之处在于它和在 V8内存堆之外分配的一段内存数据相对应。Buffer 对象的大小不能调整。你可以通过"require('buffer').Buffer"来使用这个类。

Buffer 对象是全局对象。

Buffer 和 Javascirpt 中 string 对象之间的转换需要指定编码方式。如下是 node 支持的各种编码方式：

- 'ascii' - 应用于7位的 ASCII 数据。这种编码方式速度很快，它会删除字节的高位。
- 'utf8' - Unicode 字符。许多网页和其他文档使用 UTF-8。
- 'base64' - Base64编码。
- 'binary' - 一种只使用每个字符前8个字节将原始的二进制数据编码进字符串的方式。这个方式已经废弃，应当尽量使用 buffer 对象。这个编码将会在未来的 node 中删除。
-

new Buffer(size)

创建指定大小的 buffer 对象。

new Buffer(array)

从数组新建 buffer 对象。

new Buffer(str, encoding='utf8')

新建一个保存指定字符串的 buffer 对象。

buffer.write(string, offset=0, encoding='utf8')

使用指定的编码方式将字符串从指定偏移开始写入 buffer，然后返回实际写入的大小。 如果 buffer 空间不足，则只会写入部分字符串。在本例中使用'utf8'编码，这种方式不会写入半个字符。

示例：将一个 utf8字符串写入 buffer，然后打印出来

```
buf = new Buffer(256);
len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(len + " bytes: " + buf.toString('utf8', 0, len));
// 12 bytes: ½ + ¼ = ¾
```

buffer.toString(encoding, start=0, end=buffer.length)

解码 buffer 数据并使用指定的编码返回字符串，转换从 start 参数指定的位置开始，到 end 结束。

参看上面 buffer.write()的例子。

buffer[index]

获取或设置指定的字节。返回值代表一个字节，所以返回值的合法范围是十六进制0x00到0xFF 或者十进制0至255。

例如：将一个 ASCII 字符串复制进 buffer，每次一个字节：

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

for (var i = 0; i < str.length ; i += 1) {
   buf[i] = str.charCodeAt(i);
}
console.log(buf);
// node.js
```

Buffer.byteLength(string, encoding='utf8')

返回字符串的实际字节数。这个函数和 String.prototype.length 不同，后者返回字符串的字符数。

Example:

```
str = '\u00bd + \u00bc = \u00be';
console.log(str + ": " + str.length + " characters, " +
   Buffer.byteLength(str, 'utf8') + " bytes");
// ½ + ¼ = ¾: 9 characters, 12 bytes
```

buffer.length

buffer 的大小（以字节为单位）。请注意，这个不是存放内容的大小，而是分配给 buffer 对象的内存大小。这个大小不随 buffer 中存放内容的多少而改变。

```
buf = new Buffer(1234);
console.log(buf.length);
buf.write("some string", "ascii", 0);
console.log(buf.length);
// 1234
// 1234
```

buffer.copy(targetBuffer, targetStart, sourceStart, sourceEnd=buffer.length)

在两个 buffer 之间执行内存拷贝。

例如：新建两个 buffer 对象，然后将 buf1中16至19字节拷贝到 buf2中第八字节开始的空间中。

```
buf1 = new Buffer(26),
buf2 = new Buffer(26),
i;
for (var i = 0 ; i < 26 ; i += 1) {
   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!!!!!!!!!!!!!
```

buffer.slice(start, end)

返回和老的 buffer 引用同一段内存的新 buffer 对象，但是开始和结束的位置由 start 和 end 参数指定。

修改新的 buffer 对象将会改动原来的 buffer。

例如：使用字母表建立一个 buffer 对象，并剪切出一个新的 buffer，然后修改原始 buffer 的一个字节。

```
buf1 = new Buffer(26), buf2,
i;
for (var i = 0 ; i < 26 ; i += 1) {
   buf1[i] = i + 97; // 97 is ASCII a
}
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
```

EventEmitter 事件触发器

Node 中的很多对象都会触发事件，例如：一个 TCP 服务器在收发每个数据流时都触发事件；子进程在退出时会触发事件。 所有能够触发事件的对象都是 events.EventEmitter 的实例。

事件命名方式使用大小写分隔的风格。例如：'stream', 'data', 'messageBegin'。

可以将函数注册给对象，使其在事件触发时执行， 此类函数被称作‘监听器‘。

通过调用 require('events').EventEmitter， 我们可以使用 EventEmitter（事件触发器）类。

当向 EventEmitters（事件触发器）对象上注册新的时间监听器时，都会触发'newListener'事件。

当事件触发器过程中出现错误时，典型的处理方式是它将触发一个'error'事件。 Error 事件的特殊性在于：如果没有函数处理这个事件，它将会输出调用堆栈，并随之退出应用程序。

Event: 'newListener'

```
function (event, listener) { }
```

该事件在添加新监听器时被触发。

Event: 'error'

```
function (exception) { }
```

如果发生错误，'error'事件将会被触发。 这是一个特殊事件，如果没有相应的监听函数监听这个事件， node 将会结束应用程序的执行并显示异常堆栈。

emitter.on(event, listener)

向指定的事件监听器数组尾部添加一个新监听器。

```
server.on('stream', function (stream) {
        console.log('someone connected!');
});
```

emitter.removeListener(event, listener)

从指定监听器数组中删除一个监听器。 需要注意的是，此操作将会改变处于被删监听器之后的那些监听器的索引。

```
var callback = function(stream) {
   console.log('someone connected!');
};
```

```
server.on('stream', callback);
// ...
server.removeListener('stream', callback);
```

emitter.removeAllListeners(event)

删除指定事件的所有监听器。

emitter.listeners(event)

返回指定事件的监听器数组。 你可以操作数组的内容，比如说删除一个监听器。

```
server.on('stream', function (stream) {
  console.log('someone connected!');
});
console.log(sys.inspect(server.listeners('stream'));
// [ [Function] ]
```

emitter.emit(event, [arg1], [arg2], [...])

使用所提供的参数，依次执行事件监听器数组中的每一个监听函数。

Streams 流

stream 是一个抽象接口，node 中有很对象实现了这个接口。 例如，对 http 服务器发起请求的 request 对象就是一个 stream，还有 stdout（标准输出）。Stream 可以是只读、可写，也可以同时可读可写。 所有的 Stream 对象都是 EventEmitter 的实例。

Readable Stream 只读流

一个只读流有如下方法、成员、和事件。

Event: 'data'

function (data) { }

'data'事件的参数是 Buffer（默认情况下），如果调用过 setEncoding()方法，则参数为一个字符串。

Event: 'end'

function () { }

此事件在流遇到 EOF(在 TCP 中为 FIN)时被触发，表示该流不会再有数据（不会再次触发'data'事件）。如果该流也是可写流，则它还可以继续写入。

Event: 'error'

function (exception) { }

在收取数据出错时被触发。

Event: 'close'

function () { }

内部的文件描述符被关闭时被触发，并不是所有的流都会触发此事件。（例如，一个进入的(incoming)HTTP 请求将不会触发'close'事件）。

Event: 'fd'

function (fd) { }

当数据流接收到文件描述符信息时触发该事件（一个文件数据流包含两部分信息：文件描述符信息和文件的数据信息）。 本事件只支持 Unix 数据流，其他类型的流不会触发该事件。

stream.readable

一个布尔值，默认为 true。当遇到错误或流读到结尾或者调用 destory()函数后，该值被设置为 false。

stream.setEncoding(encoding)

该函数设置 data 事件返回字符串而不是 Buffer 对象。 编码类型可以设置为"utf8"，"ascii"或"base64"。

stream.pause()

暂停触发 data 事件。

stream.resume()

恢复触发'data'事件。

stream.destroy()

关闭内部的文件描述符。 这样该流将不会再触发任何事件。

Writable Stream 可写流

一个可写流具备以下方法、成员、和事件。

Event: 'drain'

function () { }

在一个 wrire() 方法被调用并返回 false 后触发，表明可以安全的再次写入该 stream。

Event: 'error'

function (exception) { }

在异常发生赤错误时被触发。

Event: 'close'

function () { }

当底层的文件描述符已终止时发出。

stream.writeable

一个 *boolean* 值，缺省为 *true* ，但是在一个'error'产生或是 end() / destroy() 被调用后，会变为 *false* 。

stream.write(string, encoding='utf8', [fd])

使用指定的编码将字符串字符串写入到流中。如果字符串已被刷新到内核缓冲区，返回 true。返回 false 则表明内核缓冲区已满，数据将在未来被发送出去。'drain'事件用来通知内核缓冲区何时为空。此方法的默认编码为'utf8'。

如果指定了可选参数 fd，它将被当做一个文件描述符并通过流来发送。它只支持 UNIX 流，否则会被忽略且没有任何提示。当用这种方式发送文件描述符时，在流清空之前关闭文件描述符可能导致发送出非法的描述符。

stream.write(buffer)

同上，除了使用一个原始缓冲区。

stream.end()

通过 *EOF* 或 *FIN* 来终止流。

stream.end(string, encoding)

根据指定的编码发送字符串，并通过 *EOF* 或 *FIN* 来终止流。这对于减少发送数据包的数量是非常有用的。

stream.end(buffer)

同上，但使用一个缓冲区。

stream.destroy()

终止底层的文件描述符，此后流不再发出任何事件。

Global Objects 全局对象

这些对象在全局范围内都可用，并且可以从任何位置访问。

global

全局命名空间对象

process

有关 process 对象请参见'process'章节。

require()

To require modules. See the 'Modules' section.

用来加载模块。参见“Modules 模块”这一节。

require.paths

一个保存了 require 函数搜索路径的数组。你可以修改此数组添加自定义路径。

例子：在搜索路径列表开头添加一个路径。

```
require.paths.unshift('/usr/local/node');
console.log(require.paths);
// /usr/local/node,/Users/mjr/.node_libraries
```

__filename

当前正在执行的脚本的文件名。此为绝对路径，且和命令行参数所指定的文件名不一定相同。

例子：执行/Users/mjr 下的 example.js 文件。

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

__dirname

当前执行脚本的文件夹。

例子：执行/Users/mid 下的 example.js 文件。

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

module

指向当前模块的引用。特别指出，module.exports 就是 exprots 对象。更多信息请参看源代码文件 src/process.js。

process 进程

process 对象是一个全局对象，可以在任何地方访问。它也是一个 EventEmitter 的实例。

Event: 'exit'

function () {}

此事件在进程退出时被触发。这是一个检查模块状态的好地方（例如，做单元测试）。 由于主事件循环在'exit'返回方法之后将不会继续执行，所以计时器(timers)可能不会生效。

示例，监听 exit 事件：

```
process.on('exit', function () {
  process.nextTick(function () {
    console.log('This will not run');
  });
  console.log('About to exit.');
});
```

Event: 'uncaughtException'

function (err) { }

发生未处理的异常时，此事件会被触发。 如果该事件有监听函数，则不执行默认行为（默认行为将打印错误堆栈并结束应用程序的执行）。

示例，监听'uncaughtException'事件：

```
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 事件是一种非常原始的异常处理机制。 你可以在程序中使用 try/cache 来获得对程序流的更多控制权。 特别是对于要长时间执行的服务器端程序，uncaughtException 事件是个很有用的安全机制。

Signal Events

function () {}

当进程接收到信号时被触发。 要查看如 SIGINT 或 SIGUSR1之类的标准 POSIX 信号列表，请参看参看 sigaction(2)。

监听 SIGINT 信号的示例：

```
var stdin = process.openStdin();

process.on('SIGINT', function () {
  console.log('Got SIGINT.  Press Control-D to exit.');
});
```

发送 SIGIINT 信号最简单的方法是使用 Control-C，大多数情况下这会终止应用程序的执行。

process.stdout

一个代表标准输出的流对象。

示例：console.log 的定义

```
console.log = function (d) {
  process.stdout.write(d + '\n');
};
```

process.openStdin()

打开标准输入流，返回一个只读流对象。

打开标准输入并同时监听两个事件的示例：

```
var stdin = process.openStdin();

stdin.setEncoding('utf8');

stdin.on('data', function (chunk) {
  process.stdout.write('data: ' + chunk);
});

stdin.on('end', function () {
  process.stdout.write('end');
});
```

process.argv

保存命令行参数的数组。 第一个参数是"node"，第二个参数是 Javascript 文件的文件名，接下来是附加的命令行参数。

```
// print process.argv
```

```
process.argv.forEach(function (val, index, array) {
   console.log(index + ': ' + val);
});
```

上述代码将产生如下输出：

```
$ 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

此参数为进程可执行文件的绝对路径。

Example:

例如：

```
/usr/local/bin/node
```

process.chdir(directory)

改变进程的当前目录，失败时抛出异常。

```
console.log('Starting directory: ' + process.cwd());
try {
   process.chdir('/tmp');
   console.log('New directory: ' + process.cwd());
}
catch (err) {
   console.log('chdir: ' + err);
}
```

process.compile(code, filename)

同 evel 方法相同，但是你可以指定文件名，这样可以更好的输出错误信息，并且运行的代码(指通过 code 参数传递的代码)无法访问本地作用域。如果编译的代码产生堆栈输出，filename 参数将会被用作这段代码的文件名。

示例，使用 process.compile 和 eval 执行同一段代码：

```
var localVar = 123,
      compiled, evaled;

compiled = process.compile('localVar = 1;', 'myfile.js');
console.log('localVar: ' + localVar + ', compiled: ' + compiled);
evaled = eval('localVar = 1;');
console.log('localVar: ' + localVar + ', evaled: ' + evaled);
```

```
// localVar: 123, compiled: 1
// localVar: 1, evaled: 1
```

process.compile 并没有访问本地作用域，所以 localVar 变量并没有改变。eval 可以访问本地作用域，所以 localVar 被改变了。

当代码中有语法错误时，process.compile 将会是应用程序退出。

参见：脚本（Script）章节

process.cwd()

返回进程的当前工作目录。

```
console.log('Current directory: ' + process.cwd());
```

process.env

一个保存用户环境变量的对象。参看 environ(7)。

process.exit(code=0)

使用进程退出代码（main 函数的返回值）并退出进程。 如果不指定参数，exit 将使用表示成功的代码0。

示例，退出程序，并返回错误状态。

```
process.exit(1);
```

执行 node 的 shell 将会得到返回值1。

process.getgid()

返回进程的用户组标识。（参见 getgid(2).）这个是数字形式的组 ID，并非组名。

```
console.log('Current gid: ' + process.getgid());
```

process.setgid(id)

当前进程的用户组标识。（参见 setgid(2).）这个函数可以接受数字形式的组 ID 或者是字符串形式的组名。如果指定组名，此函数会阻塞进程直至将组名解析成为数字 ID。

```
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()

返回当前进程的用户标识。（参看 getuid(2).）此函数返回数字形式的用户 ID，而不是用户名。

```
console.log('Current uid: ' + process.getuid());
```

process.setuid(id)

指定当前进程的用户标识。（参看 setuid(2).）这个函数可以接受数字形式的用户 ID 或者字符串形式的用户名。如果指定用户名，此方法在将用户名解析成用户 ID 时会阻塞。

```
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.version

编译进可执行文件的属性，代表 NODE_VERSION。

```
console.log('Version: ' + process.version);
```

process.installPrefix

编译进可执行文件的属性，代表 NODE_PREFIX。

```
console.log('Prefix: ' + process.installPrefix);
```

process.kill(pid, signal='SIGINT')

向一个进程发送信号，参数 pid 为进程 ID，signal 是一个描述要发送信号的字符串，如‘SIGINT’或者‘SIGUSR1’。如果不指定，默认发送'SIGINT'信号。 更多信息请参看 kill(2)。

请注意，虽然此函数名为 process.kill，但是它仅仅用于发送信号，就像 kill 系统调用。发送的信号做除了结束目标进程外，还可能做其他的事情。

发送信号的示例：

```
process.on('SIGHUP', function () {
  console.log('Got SIGHUP signal.');
});

setTimeout(function () {
  console.log('Exiting.');
  process.exit(0);
}, 100);

process.kill(process.pid, 'SIGHUP');
```

process.pid

当前进程的 ID

```
console.log('This process is pid ' + process.pid);
```

process.title

设置、获取 ps 命令中显示的名称。

process.platform

表示程序运行的平台，如'linux2','darwin'等。

```
console.log('This platform is ' + process.platform);
```

process.memoryUsage()

返回一个描述 Node 进程内存占用的对象。

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

console.log(sys.inspect(process.memoryUsage()));
```

如上代码将输出：

```
{ rss: 4935680
, vsize: 41893888
, heapTotal: 1826816
, heapUsed: 650472
}
```

heapTotal 和 heapUsed 表示 V8占用的内存。

process.nextTick(callback)

在事件循环的下一轮调用这个回调。 此函数不是 setTimeout(fn, 0)的别名，它更加高效。

```
process.nextTick(function () {
   console.log('nextTick callback');
});
```

process.umask([mask])

设置或读取进程的文件创建模式掩码，子进程会从父进程继承这个掩码。如果使用此函数设置新的掩码，则它返回旧的掩码，否则返回当前掩码。

```
var oldmask, newmask = 0644;
oldmask = process.umask(newmask);
console.log('Changed umask from: ' + oldmask.toString(8) +
            ' to ' + newmask.toString(8));
```

sys

本章介绍的函数包含在 sys 模块中，可以通过 require('sys')访问他们。

sys.print(string)

此函数和 console.log()类似，只是它不输出结尾的换行符。

```
require('sys').print('String with no newline');
```

sys.debug(string)

同步输出函数，此函数将阻塞进程并将字符串打印到标准错误输出（stderr）。

```
require('sys').debug('message on stderr');
```

sys.log(string)

将字符串输出至标准输出（stdout,就是控制台），附加时间戳。

```
require('sys').log('Timestmaped message.');
```

sys.inspect(object, showHidden=false, depth=2)

将对象转化为字符串的形式返回，对调试非常有用。

如果 showHidden 参数设定为 true，则对象的非枚举属性也会被转化。

如果指定 depth 参数，它告诉解析器(inspectter)格式化对象的时候要递归的次数。 这个参数对于解析(inspecting)复杂的对象很有用。

默认只递归两次。要想无限递归，请传递 null。

例子，解析 sys 对象的所有属性：

```
var sys = require('sys');
console.log(sys.inspect(sys, true, null));
```

sys.pump(readableStream, writeableStream, [callback])

Experimental 实验性的

从 readableStream 读取数据并写入 writableStream。如果 writeableStream.write(data)返回 flase，readableStream 将暂停，直到 writableStream 的 drain 事件被触发。当 writableStream 关闭或者错误发生时，回调函数（第三个参数）会被调用，并接受一个表示错误的参数。

Timers 计时器

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

设置延时 delay 毫秒之后执行回调函数(callback)。 该函数返回 timeoutId，可以使用 clearTimeout()清除定时。你也可以传递额外的参数给回调函数。

clearTimeout(timeoutId)

清除指定的定时。

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

设置重复延时调用 callback。 该函数返回 intervalId，可以使用 clearTimeout()清除定时。 你也可以传递可选的参数给回调函数。

clearInterval(intervalId)

清楚指定的重复延时回调。

Child Processes 子进程

node 通过 ChildProcess 类提供全面的 popen(3)功能。

程序可以通过子进程的标准输入(stdin)、标准输出(stdout)、标准错误输出(stderr)以完全非阻塞的形式传递数据。

你可以使用 require('child_process').spawn()创建子进程。

每个子进程总是带有三个流对象：child.stdin, child.stdout 和 child.stderr。

每个 ChildProcess 类也是一个事件触发器。

Event: 'exit'

function (code, signal) {}

此事件在子进程结束后被触发。 如果进程正常结束，code 参数的值就是子进程退出后的返回值，否则此参数为 null。 如果进程因为信号而终止，参数 signal 就是信号的名称，否则此参数为 null。

触发此事件之后，node 将不再触发'output'和'error'事件。

See waitpid(2).

child.stdin

可写流(Writable Stream)，代表子进程的标准输入(stdin)。 使用 end()函数关闭此流(stream)，通常会终止子进程。

child.stdout

只读流(Readable Stream)，代表子进程的标准输出(stdout)。

child.stderr

只读流(Readable Stream)，代表子进程的标准错误输出(stderr)。

child.pid

子进程的 PID

Example:

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

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

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

使用指定的命令行参数创建新线程。 如果不指定参数，args 默认为空数组。

第三个参数用于指定一些额外的选项，默认值如下：

```
{ cwd: undefined
, env: process.env,
, customFds: [-1, -1, -1]
}
```

cwd 可以用于指定新进程的工作目录，env 指定新进程可见的环境变量，而 customFds 则可以将新进程的[stdin,stdout,stderr]绑定到指定的流，-1指创建新的流。

例子：执行命令'ls -lh /usr'并捕获标准输出、标准错误输出和退出代码（程序退出时，main 函数返回的代码）。

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

ls.stdout.on('data', function (data) {
    sys.print('stdout: ' + data);
});

ls.stderr.on('data', function (data) {
    sys.print('stderr: ' + data);
});

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

例子： 实现'ps ax | grep ssh'命令。

```
var sys     = require('sys'),
      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) {
  sys.print('ps stderr: ' + data);
});

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

grep.stdout.on('data', function (data) {
  sys.print(data);
});

grep.stderr.on('data', function (data) {
  sys.print('grep stderr: ' + data);
});

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

例子，检测退出代码：

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

child.stderr.on('data', function (data) {
  if (/^execvp\(\)/.test(data.asciiSlice(0,data.length))) {
    console.log('Failed to start child process.');
  }
});
```

请参见：child_process.exec()

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

使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。

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

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

回调函数得到的参数分别为(error, stdout, stderr)。 成功时 error 参数是 null；失败时 error 参数是 Error 的实例。error.code 是子进程的返回值，error.signal 保存终止进程的信号。

exec 函数的第二个可选参数可以用来指定一些选项。默认值为

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

如果 timeout 为大于0的值，node 将终止运行时间超过 timeout（单位为毫秒）的子进程。子进程将被终止信号(killSignal，默认值为'SIGKILL')终止。 maxBuffer 指定 stdout 和 stderr 最大可缓存数据的大小，如果超过这个值子进程将被终止。

child.kill(signal='SIGTERM')

向子进程发送信号。如果不指定参数，默认发送'SIGTERM'信号。所有可用信号列表请参考 signal(7)。

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

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

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

注意，虽然此函数名为 kill，但发送的信号不一定终止子进程。kill 实际上只会向进程发送信号。

See kill(2)

更多信息请参考 kill(2)

Script 脚本

Script 类可以编译执行 JavaScript 代码。你可以用以下方式访问 Script 类：

```
var Script = process.binding('evals').Script;
```

JavaScript 代码可以被编译、立刻执行或者编译、保存、延时执行。

Script.runInThisContext(code, [filename])

同 process.compile 函数类似，Script.runInThisContext 函数编译执行 code 参数包含的代码并返回结果，就如同这些代码是从 filename 参数指定文件中加载的一样。这些代码不能访问本地作用域。filename 参数是可选的。（译注：filename 参数的作用是为了更好的输出错误信息）

示例：演示使用 Script.runInThisContext 函数和 eval 函数执行同一段代码：

```
var localVar = 123,
    usingscript, evaled,
    Script = process.binding('evals').Script;

usingscript = Script.runInThisContext('localVar = 1;',
  'myfile.js');
console.log('localVar: ' + localVar + ', usingscript: ' +
  usingscript);
evaled = eval('localVar = 1;');
console.log('localVar: ' + localVar + ', evaled: ' +
  evaled);

// localVar: 123, usingscript: 1
// localVar: 1, evaled: 1
```

Script.runInThisContext 函数执行的代码并不访问本地作用域，所以 localVar 变量的值并没有改变。 eval 函数执行的代码可以访问本地作用域，所以 localVal 的值被改变了。

如果代码有语法错误，Script.runInThisContext 会输出错误信息到控制台（stderr）并抛出异常。

Script.runInNewContext(code, [sandbox], [filename])

Script.runInNewContext 将代码编译并在 sandbox 参数指定的作用域内执行代码并返回结果，就如同代码是从文件中加载的一样。执行的代码并不访问本地作用域，sandbox 参数指定的对象将作为代码执行的全局对象。sandbox 和 filename 参数都是可选的。

例子：编译并执行一段代码，这段代码递增并新建一个全局变量。这些全局变量都保存在 sandbox 中。

```
var sys = require('sys'),
    Script = process.binding('evals').Script,
    sandbox = {
      animal: 'cat',
      count: 2
    };
```

```
Script.runInNewContext(
   'count += 1; name = "kitty"', sandbox, 'myfile.js');
console.log(sys.inspect(sandbox));

// { animal: 'cat', count: 3, name: 'kitty' }
```

请注意，执行不信任的代码(untrusted code)是一项需要技巧的工作。Script.runInNewContext 函数非常有用，它可以在一个独立的线程中执行不信任的代码防止全局变量被意外修改。

如果代码有语法错误，Script.runInThisContext 会输出错误信息到控制台（stderr）并抛出异常。

new Script(code, [filename])

新建 Script 对象会编译 code 参数指定的代码，就如同代码是从 filename 参数指定的文件中加载的一样。和其他函数不同的是，它将返回一个代表经过编译的代码的 Script 对象，这个对象可以使用下面介绍的函数执行内部编译好的代码。这个 script 对象并不绑定到任何全局对象，但是可以在运行时绑定到指定对象，每次绑定仅在本次运行时生效。filename 参数是可选的。

如果代码有语法错误，new Script emits 会输出错误信息到控制台（stderr）并抛出异常。

script.runInThisContext()

这个函数和 Script.runInThisContext 函数类似（对象名首字母'S'的大小写不同），不同的是此函数是 Script 对象的方法。script.runInThisContext 函数执行对象中的代码并返回结果。 执行的代码并不会访问本地作用域，但是可以访问全局作用域（v8: in actual context）。

例子：使用 script.runInThisContext 函数实现代码的一次编译多次执行。

```
var Script = process.binding('evals').Script,
      scriptObj, i;

globalVar = 0;

scriptObj = new Script('globalVar += 1', 'myfile.js');

for (i = 0; i < 1000 ; i += 1) {
   scriptObj.runInThisContext();
}

console.log(globalVar);

// 1000
```

script.runInNewContext([sandbox])

此函数和 Script.runInNewContext 函数类似（对象名首字母'S'的大小写不同），不同的是此函数是 Script 对象的方法。script.runInNewContext 函数将 sandbox 参数指定的对象作为全局对象执行代码，并返回结果。执行的代码并不访问本地作用域。sandbox 参数是可选的。

例子：编译并执行一段代码，这段代码递增并新建一个全局变量，这些全局变量都保存在 sandbox 中。然后多

次执行这段代码，这些全局变量都保存在沙盒(sandbox)中。

```
var sys = require('sys'),
    Script = process.binding('evals').Script,
    scriptObj, i,
    sandbox = {
      animal: 'cat',
      count: 2
    };

scriptObj = new Script(
    'count += 1; name = "kitty"', 'myfile.js');

for (i = 0; i < 10 ; i += 1) {
  scriptObj.runInNewContext(sandbox);
}

console.log(sys.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

请注意，执行不信任的代码(untrusted code)是一项需要技巧的工作。script.runInNewContext 函数非常有用，它可以在一个单独的线程中执行不信任的代码防止全局变量被意外修改。

File System 文件系统

(注：同步与异步方式是指同步或异步于程序执行，并非函数彼此之间的同步关系。)

文件的 I/O 是由标准 POSIX 函数封装而成。需要使用"require('fs')"操作这个类。所有的方法设有异步方式和同步方式。

异步形式下的方法其最后一个参数，总是一个完整的回调函数（callback）。这个回调函数有那些参数，就取决于异步方法怎么送入参数，但通常来说，第一个送入的参数是异常对象。如果是没有任何问题的操作，那么这个异常对象就变为 null 或者 undefined，表示操作正常。

异步方式的例子：

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

fs.unlink('/tmp/hello', function (err) {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

同步方式的例子：

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

fs.unlinkSync('/tmp/hello')
```

```
console.log('successfully deleted /tmp/hello');
```

异步函数没有一定的顺序，所以以下例子容易发生错误：

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

这有可能 fs.state 于 fs.rename 前执行。正确的做法是嵌套回传函数。

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

当执行动作繁杂时，强烈建议使用异步方式调用此类。同步方式在其完成之前将会阻挡一切随后的动作，这代表搁置所有连接。

fs.rename(path1, path2, [callback])

异步命名(rename(2))。只传递异常给回调函数。

fs.renameSync(path1, path2)

同步命名(rename(2))。

fs.truncate(fd, len, [callback])

异步截断(ftruncate(2))。只传递异常给回调函数。

fs.truncateSync(fd, len)

同步截断(ftruncate(2))。

fs.chmod(path, mode, [callback])

异步更改文件权限(chmod(2))。只传递异常给回调函数。

fs.chmodSync(path, mode)

同步更改文件权限(chmod(2))。

fs.stat(path, [callback])

利用路径异步读取属性(stat(2))。回调函数的第二个参数是 fs.Stats 对象(err, stats)，例如：

```
{ dev: 2049
, ino: 305352
, mode: 16877
, nlink: 12
, uid: 1000
, gid: 1000
, rdev: 0
, size: 4096
, blksize: 4096
, blocks: 8
, atime: '2009-06-29T11:11:55Z'
, mtime: '2009-06-29T11:11:40Z'
, ctime: '2009-06-29T11:11:40Z'
}
```

参考 fs.Stats 部份以取得详细资料。

fs.lstat(path, [callback])

利用路径异步读取属性(lstat(2))。如果这个文件参数是一个符号连接，则返回该符号连接的属性。回调函数的第二个参数是 fs.Stats 对象。(err, stats)

fs.fstat(fd, [callback])

利用存在的指标异步读取属性(fstat(2))。回调函数的第二个参数是 fs.Stats 对象。(err, stats)

fs.statSync(path)

同步读取属性(stat(2))。返回 fs.Stats。

fs.lstatSync(path)

利用路径同步读取属性(lstat(2))。返回 fs.Stats。

fs.fstatSync(fd)

利用存在的指标同步读取属性(fstat(2))。返回 fs.Stats。

fs.link(srcpath, dstpath, [callback])

异步建立连接(link(2))。只传递异常给回调函数。

fs.linkSync(dstpath, srcpath)

同步建立连接(link(2))。

fs.symlink(linkdata, path, [callback])

异步建立符号连接(symlink(2))。只传递异常给回调函数。

fs.symlinkSync(linkdata, path)

同步建立符号连接(symlink(2))。

fs.readlink(path, [callback])

异步读取连接(readlink(2))。回调函数的第二个参数是已解析的文件路径。(err, resolvedPath)

fs.readlinkSync(path)

同步读取连接(readlink(2))。返回已解析的文件路径。

fs.realpath(path, [callback])

异步读取絶对的路径名称(realpath(2))。回调函数的第二个参数是已解析的文件路径。(err, resolvedPath)

fs.realpathSync(path)

同步读取絶对的路径名称(realpath(2))。返回已解析的文件路径。

fs.unlink(path, [callback])

异步删除连接(~=删除文件)(unlink(2))。只传递异常给回调函数。

fs.unlinkSync(path)

同步删除连接(~=删除文件)(unlink(2))。

fs.rmdir(path, [callback])

异步删除目录(rmdir(2))。只传递异常给回调函数。

fs.rmdirSync(path)

同步删除目录(rmdir(2))。

fs.mkdir(path, mode, [callback])

异步建立目录(mkdir(2))。只传递异常给回调函数。

fs.mkdirSync(path, mode)

同步建立目录(mkdir(2))。

fs.readdir(path, [callback])

异步读取目录中的内容(readdir(3))。回调函数的第二个参数是以阵列构成的目录内对象的名称('.'与'..'除外)。(err, files)

fs.readdirSync(path)

同步读取目录中的内容(readdir(3))。返回以阵列构成的目录内对象名称('.'与'..'除外)。

fs.close(fd, [callback])

异步结束(close(2))。只传递异常给回调函数。

fs.closeSync(fd)

同步结束(close(2))。

fs.open(path, flags, mode=0666, [callback])

异步开启文件，详阅 open(2)。标签可为'r', 'r+', 'w', 'w+', 'a', 或 'a+'。回调函数的第二个参数是指标。(err, fd)

fs.openSync(path, flags, mode=0666)

同步开启文件。

fs.write(fd, buffer, offset, length, position, [callback])

透过指标(fd)写入缓冲区至文件。

offset 偏移 和 length 长度 决定哪一部份的缓冲区被写入。

position 写入位置 若 position 为空，则写入至现存位置。详阅 pwrite(2)。

回调函数的第二个参数是写入动作的数据大小(bytes)。(err, written)

fs.writeSync(fd, buffer, offset, length, position)

fs.write(缓冲区)的同步方式。返回写入动作的数据大小。

fs.writeSync(fd, str, position, encoding='utf8')

fs.write(字串)的同步方式。返回写入动作的数据大小。

fs.read(fd, buffer, offset, length, position, [callback])

透过指标(fd)读取数据。

buffer 是读取的数据的存放位置。

offset 是标注哪里开始写入缓冲区。

length 是以整数型态(INT)标注读取的数据大小。

position 是以整数型态(INT)标注文件的读取位置。若 position 为空，则由现存位置读取。

回调函数的第二个参数是读取动作的数据大小(bytes)。(err, bytesRead)

fs.read(fd, length, position, encoding, [callback])

透过指标(fd)读取数据。

length 是以整数型态(INT)标注读取的数据大小。

position 是以整数型态(INT)标注文件的读取位置。若 position 为空，则由现存位置读取。

encoding 是读取数据的预期编码。

回调函数的第二个参数是读数的数据而第三个参数是读取动作的数据大小(bytes)。(err, str, bytesRead)

fs.readSync(fd, buffer, offset, length, position)

fs.read(缓冲区)的同步方式。返回读取动作的数据大小。

fs.readSync(fd, length, position, encoding)

fs.read(字串)的同步方式。返回读取动作的数据大小。

fs.readFile(filename, [encoding], [callback])

透过文件路径异步读取内容，例子：

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

回调函数的第二个参数是文件内容。(err, datad)

若编码(encoding)没有被指定，则返回原始缓冲区。

fs.readFileSync(filename, [encoding])

fs.readFile()的同步方式。返回文件内容。

若编码(encoding)被指定，则返回字串，反之则返回原始缓冲区。

fs.writeFile(filename, data, encoding='utf8', [callback])

异步写入数据至文件，data(数据)可以为字串或缓冲区。

例子：

```
fs.writeFile('message.txt', 'Hello Node', function (err) {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

fs.writeFileSync(filename, data, encoding='utf8')

fs.writeFile()的同步方式。

fs.watchFile(filename, [options], listener)

观察文件异变。文件异动时会触发监听函数。

第二个参数是可选的。选项(options)参数应为对象(object)，当中包含一布林值(BOOL)持续检查(persistent)与检测相隔时间(interval)(单位为毫秒)。

传递给监听函数的参数分别是当前状态对象(curr)以及前次状态对象(prev)。

```
fs.watchFile(f, function (curr, prev) {
  console.log('the current mtime is: ' + curr.mtime);
  console.log('the previous mtime was: ' + prev.mtime);
});
```

stat 对象是 fs.Stat 的实例。

fs.unwatchFile(filename)

停止观察文件异变。

fs.Stats 获取文件信息

fs.stat()和 fs.lstat()函数返回如下类型的对象：

- stats.isFile()
- stats.isDirectory()
- stats.isBlockDevice()
- stats.isCharacterDevice()
- stats.isSymbolicLink() (only valid with fs.lstat())
- stats.isFIFO()
- stats.isSocket()

fs.ReadStream 读取文件

ReadStream 是一个只读流。

fs.createReadStream(path, [options])

fs.createReadStream 函数新建只读流对象。（请参考“只读流”章节）

options 是一个默认值如下所示的对象：

```
{
'flags': 'r'
, 'encoding': null
, 'mode': 0666
, 'bufferSize': 4 * 1024
}
```

options 对象可以包含'start'和'end'参数用于从文件中读取一个范围内的数据，而不是整个文件。开始和结束都包含并且从偏移0的位置开始，使用时必须同时指定这两个参数。

An example to read the last 10 bytes of a file which is 100 bytes long:

例子，读取一个100字节文件的最后十字节。

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

fs.WriteStream 写入文件

WriteStream 是一个可写流。

Event: 'open' function (fd) { }

fd 是可写流使用的文件描述符。

fs.createWriteStream(path, [options])

此函数新建一个 WriteStream 对象（参见"可写流"章节）。

options is an object with the following defaults:

options 是一个具有如下默认值的对象：

```
{
'flags': 'w'
, 'encoding': 'null'
, 'mode': 0666
}
```

HTTP

如果要使用 HTTP 的 server 以及 client 模块则必须使用 require('http')加载 http 模块

NODE 中的 HTTP 接口被设计成为支持 HTTP 协议的很多特性，这些特性通常那难以掌控，特别是 large,possible

chunk-encoded(块编码),messages。 这个接口特意不缓冲整个请求(request)或者响应(responses)使用户可以使用流的形式操作数据。

以下是用对象的形式表示的 HTTP 信息头

{ 'content-length': '123' , 'content-type': 'text/plain' , 'stream': 'keep-alive' , 'accept': '*/*' }

所有 key 都是小写，数值不能被修改

为了支持尽可能多的 HTTP 应用，NODE 的 HTTP API 非常底层。 其只处理到流(stream)相关的操作以及信息解析。API 将信息解析成为信息头和信息体，但并不解析实际的信息头和信息体的具体内容。

如果在基础平台上的 OpenSSL 是可用的则 HTTPS 也能够被支持

http.Server

此模块会触发以下事件

Event: 'request'

function (request, response) { }

request 是 http.ServerRequest 的一个实例，而 response 则是 http.ServerResponse 的一个实例

Event: 'connection'

function (stream) { }

当一个新的 TCP stream 建立后发出此消息。stream 是一个 net.Stream 的对象，通常用户不会访问/使用这个事件。参数 stream 也可以在 request.connection 中访问到.

Event: 'close'

function (errno) { }

当服务器关闭的时候触发此事件。

Event: 'request'

function (request, response) {}

每个请求发生的时候均会被触发。请记住，每个连接可能会有多个请求(在 keep-alive 连接情况下)

Event: 'upgrade'

function (request, socket, head)

每当一个客户端请求一个 http upgrade 时候发出此消息。如果这个事件没有监听，那么请求 upgrade 的客户端对应的连接将被关闭。

- 参数“request”代表一个 http 请求，和'request'事件的参数意义相同。
- socket 是在服务器与客户端之间连接用的网络 socket
- head 是 Buffer 的一个实例,是 upgraded stream(升级版 stream....应当就是 http upgrade)所发出的第一个包,

这个参数可以为空。

当此事件被触发后，该请求所使用的 socket 并不会有一个数据事件的监听者,这意味着你如果需要处理通过这个 SOCKET 发送到服务器端的数据的话则需要自己绑定数据事件监听器

Event: 'clientError'

function (exception) {}

如果一个客户端连接的'error'事件被触发，此函数将被执行。

http.createServer(requestListener)

返回一个新的 web server 对象。

requestListener 是一个会去自动监听'request'事件的函数。

server.listen(port, [hostname], [callback])

在指定端口和主机名上接受连接。如果 hostname 没有写,这个服务器将直接在此机器的所有 IPV4地址上接受连接(INADDR_ANY).

如果要在 UNIX SOCKET 上监听的话，则需要提供一个文件名来替换端口和主机名.

这个方法是一个异步的方法，作为最后一个参数的回调方法将在服务器已经在此端口上绑定好后被调用.

server.listen(path, [callback])

建立一个 UNIX SOCKET 服务器并在指定路径监听。

这个方法是一个异步的方法，作为最后一个参数的回调方法将在服务器已经在此端口上绑定好后被调用。

server.setSecure(credentials)

允许此服务器支持 HTTS，配合 crypto 模块 credentials 指定私钥以及此服务器的证书，并且也可选择数字中心认证的证书作为客户端的认证（方式）。

如果 authentication 中有一或多个数字认证中心证书，则服务器将请求客户端发出一个客户端证书作为 HTTPS 握手的一部分。想要验证、访问证书合法性及内容则可以通过服务器的 request.connection 的 verifyPeer() 以及 getPeerCertificate() 来实现。

server.close()

使此服务器停止接受任何新连接。

server.maxConnections

设置此属性使服务器的连接数高于此数值时拒绝连接。

server.connections

代表当前服务器的连接数。

http.ServerRequest

这个对象通常由 HTTP SERVER 建立而非用户手动建立，并且会作为传递给 'request'事件监听器第一个参数

此对象的可以触发以下事件：

Event: 'data'

function (chunk) { }

当接收到信息体中的一部分时候会发出 data 事件。

例如:代表消息体的数据块将作为唯一的参数传递给回调函数。这个时候数据已经按照传输编码进行了解码（不是字符集编码）。消息体本身是一个字符串，可以使用 request.setBodyEncoding()方法设定消息体的编码。

Event: 'end'

function () { }

每次完全接收完信息后都会触发一次。没有参数，当这个事件发出后，将不会再触发其他事件。

request.method

request.method 是一个只读字符串。例如'GET','DELETE'

request.url

代表所请求的 URL 字符串.他仅包括实际的 HTTP 请求中的 URL 地址。如果这个请求是

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

则 request.url 应当是

```
'/status?name=ryan'
```

如果你想要解析这个 URL 中的各个部分，你应当使用 require('url').parse(request.url).

Example:

```
node> require('url').parse('/status?name=ryan')
{ href: '/status?name=ryan'
, search: '?name=ryan'
, query: 'name=ryan'
```

```
, pathname: '/status'
}
```

如果你想从查询字符串中提出这些参数，你可以使用 require('querystring').parse 方法,或者传一个 true 作为第二个参数给 require('url').parse 方法。

Example:

```
node> require('url').parse('/status?name=ryan', true)
{ href: '/status?name=ryan'
, search: '?name=ryan'
, query: { name: 'ryan' }
, pathname: '/status'
}
```

request.headers

只读

request.httpVersion

这是 HTTP 协议版本（字符串形式），只读。例如'1.1','1.0'。request.httpVersionMajor 是第一个数字，request.httpVersionMinor 是第二个数字。

request.setEncoding(encoding='null')

设置此请求的包体的字集编码,'utf8'或者'binary'。缺省值是 null，这表示'data'事件的参数将会是一个 Buffer 对象。

request.pause()

暂停此 request 触发事件.对于控制上传非常有用。

request.resume()

恢复一个暂停的 request。

request.connection

request.connection 是一个代表当前连接的 net.Stream 对象。

对于 HTTPS，使用 request.connection.verifyPeer() 和 request.connection.getPeerCertificate()来获得客户端（浏览器）的认证详情。

http.ServerResponse

这个对象一般由 HTTP 服务器建立而非用户自己手动建立。它作为'request'事件的第二个参数，这是一个可写流。

response.writeHead(statusCode, [reasonPhrase], [headers])

这个方法的是用来发送一个响应报文头给本次的请求方，第一个参数状态码是由一个3位数字所构成的 HTTP 状态，比如404之类的。最后一个参数 headers 是响应头具体内容.也可以使用一个方便人们直观了解的 reasonPhrase 作为第二个参数。

例如：

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain'
});
```

在一次完整信息交互中此方法只能调用一次，并且必须在调用 response.end()之前调用。

response.write(chunk, encoding='utf8')

此方法必须在 writeHead 方法调用后才可以被调用，他负责发送响应报文中的部分数据。如果要发送一个报文体的多个部分，则可以多次调用此方法。

参数 chunk 可以是一个字符串或者一个 buffer。如果 chunk 是一个字符串，则第二个参数指定如何将这个字符串编码成字节流，缺省情况下，编码为'utf8'。

注意:这是一个原始格式 http 报文体，和高层协议中的多段消息体编码格式({'Transfer-Encoding':'chunked'})无关。

第一次调用 response.write()时，此方法会将已经缓冲的消息头和第一块消息体发送给客户。 当第二次调用 response.write()的时候，node 将假定你想要以流的形式发送数据（分别发送每一个数据块并不做缓存）。 这样，其实 response 对象只是缓存消息体的第一个数据块。

response.end([data], [encoding])

这个方法会告诉服务器此响应的所有报文头及报文体已经发出；服务器在此调用后认为这条信息已经发送完毕;这个方法必须对每个响应调用一次。

如果指定 data 参数，他就相当于调用了 response.write(data, encoding)然后跟着调用了 response.end()。

http.Client

使用服务器地址作为参数来构造一个 HTTP client，其返回的句柄可用来发出一个或者多个请求。根据连接的服务器不同，这个客户端可以使用管道处理机制来处理请求或者每个请求重新构建 stream。当前的实现方式并没有用管道处理机制处理请求.

Example of connecting to google.com:

```
var http = require('http');
var google = http.createClient(80, 'www.google.com');
var request = google.request('GET', '/',
```

```
  {'host': 'www.google.com'});
request.end();
request.on('response', function (response) {
  console.log('STATUS: ' + response.statusCode);
  console.log('HEADERS: ' + JSON.stringify(response.headers));
  response.setEncoding('utf8');
  response.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});
```

如下消息头应当注意：

Node 并不会添加'Host'，但是这个属性对于一个网站来说通常是必须的。

发送'Connection: keep-alive'将告知 Node 和服务器之间的连接应当是持久连接，直到下一次请求才断开。

发送'Content-length'标记将禁用默认的消息体编码。

Event: 'upgrade'

function (request, socket, head)

当服务器响应 upgrade 请求时触发此事件，如果这个消息没有被监听，客户端接收到一个 upgrade 头的话会导致这个连接被关闭。

可以查看 http.Server 关于 upgrade 事件的解释来了解更多内容。

http.createClient(port, host='localhost', secure=false, [credentials])

构造一个新的 HTTP 客户端.port 和 host 指明了将要连接的目标。在发出请求之前不会建立流(establishe a stream)。

secure 是一个可选的布尔值，用来表示是否启用 HTTPS。credentials 是一个来自于 crypto 模块的可选参数，credentials 中可以包含 client 的私钥，证书以及一个可信任的数字认证中心的证书列表.

如果连接使用了 secure 但是没有把数字认证中心证书传给 credentials，那么 NODEJS 将缺省使用公开的可信任数字认证中心整数列表,就比如 http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt

client.request(method='GET', path, [request_headers])

发出一个请求，在必须时建立一个流。该函数返回一个 http.ClientRequest 对象。method 是可选项，缺省会以 GET 方式发出请求

request_headers 是可选项。请求头的额外部分一般由 NODE 内部实现。该函数返回一个 ClientRequest 对象

如果你就想要发送一个信息体，记得要在头信息里包含 Content-Length 项。如果你想要将 BODY 通过流的方式传输发送，或许需要设置 Transfer-Encoding: chunked.

译注:大多数的站点相应用户请求时发送的 HTTP Headers 中包含 Content-Length 头.此头信息定义在 HTTP1.0协议 RFC 1945 10.4章节中.该信息是用来告知用户代理,通常意义上就是浏览器,服务端发送的文档内容长度.浏览器接受到此信息后,接收完 Content-Length 中定义的长度字节后开始解析页面.如果服务端有部分数据延迟发送,

那么浏览器就会白屏.这样导致比较糟糕的用户体验. 解决方法在 HTTP1.1协议.RFC2616中14.41章节中定义的 Transfer-Encoding:chunked 的头信息.chunked 编码定义在3.6.1中.根据此定义浏览器不需要等到内容字节全部下载完成,只要接收到一个 chunked 块就可解析页面.并且可以下载 html 中定义的页面内容,包括 js,css,image 等.

注意：这个请求并不完全。这个方法仅仅发送了头和请求。需要发送一个 request.end()来真正的完成当前这个请求并且接收回应。（这听起来有点绕,但是这正好就提供了用户通过使用 request.write()方法来通过流方式发送 body 到服务器端的机会）

client.verifyPeer()

返回 true/false 并在上下文附带服务器定义的或者缺省数字认证中心的有效证书列表。

client.getPeerCertificate()

返回用 JSON 结构详尽表述的服务器方证书，其中包含证书的‘主题’，‘发布者’，'有效来源','有效目标'（'subject', 'issuer', 'valid_from' and 'valid_to'）。

http.ClientRequest

http.Client 的 request()方法建立并返回 http.ClientRequest 对象。该对象代表一个进行中的请求（request），该请求的消息头已经发送出去。

要获得回应，可以为 request 对象增加一个'response'事件的监听器。‘response’事件将在 request 对象接收到响应头的时候被触发，'response'事件的处理函数接收一个参数，该参数是 http.ClientResponse 的实例。

在'response'事件中，可以为 response 对象增加监听器，监听'data'事件尤为有用。要记住，'response'事件是在接收到回应信息体之前被触发，所以这里不需要担心信息体的第一部分不能被捕获。只要在处理'response'事件过程中增加'data'事件监听器，信息体是肯定可以被捕获的。

```
// Good
request.on('response', function (response) {
  response.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

// Bad - misses all or part of the body
request.on('response', function (response) {
  setTimeout(function () {
    response.on('data', function (chunk) {
      console.log('BODY: ' + chunk);
    });
  }, 10);
});
```

这是一个可写流

如下是此对象可以触发的事件。

Event 'response'

function (response) { }

在响应被接收后触发。这个事件仅会被发出一次，参数 response 是 http.ClientResponse 的实例。

request.write(chunk, encoding='utf8')

发送 body 中的一块。用户可以通过多次调用这个方法将请求数据包通过流的方式发送到服务器。在这个时候我们建议使用在建立请求的时候把['Transfer-Encoding', 'chunked']放在请求头里。

参数'chunk'应当是一个数字索引的数组或字符串。

参数'encoding'是可选的，仅在 chunk 为字符串的时使用。

request.end([data], [encoding])

完成本次请求的发送。如果消息体中的任何一个部分没有来得及发送，request.end 将把他们全部刷新到流中。如果本次请求是分块的，这个函数将发出结束字符'0\r\n\r\n'。

如果使用参数 data，就等于在调用 request.write(data, encoding)之后紧接着调用 request.end()。

http.ClientResponse

这个对象在使用 http.Client 发起请求时被创建，它会以参数的形式传递给 request 对象'response'事件的响应函数。

'response'实现了可读流的接口。

Event: 'data'

function (chunk) {}

当接收到消息体一部分的时候触发。

例如：此方法有一个参数，是消息体的一个块，这个块的内容已经被解码成一个字符串。信息体的编码通过 `response.setBodyEncoding()`来设置。

Event: 'end'

function () {}

该事件对于每个受到的消息会触发一次。它没有参数，在触发过这个事件后将不会再触发其他事件将了。

response.statusCode

一个三位数字所表示的 HTTP 回应状态，比如404。

response.httpVersion

所连接到的服务器所使用 HTTP 协议版本。大致上是'1.1'或者'1.0'。response.httpVersionMajor 则表示第一个数字 response.httpVersionMinor 表示第二个。

response.headers

http 信息头对象。

response.setEncoding(encoding='null')

设置回应信息体的编码，'utf8'、'ascii'或者'binary'。默认值为空，着表示'data'事件将会使用一个 Buffer 对象作为参数。

response.pause()

暂停 response 触发事件。通常在控制一个下载动作时候使用。

response.resume()

恢复一个暂停的请求。

response.client

保存 response 所属的 http.Client 的引用。

net.Server TCP 服务器模块

这个类(net.Server)是用来建立 TCP 或者 UNIX 服务器的。

下面有一个在8124端口等待连结的 echo server 的例子：

```
var net = require('net');
var server = net.createServer(function (stream) {
   stream.setEncoding('utf8');
   stream.on('connect', function () {
      stream.write('hello\r\n');
   });
   stream.on('data', function (data) {
      stream.write(data);
   });
   stream.on('end', function () {
      stream.write('goodbye\r\n');
      stream.end();
   });
});
server.listen(8124, 'localhost');
```

如果要使用 UNIX SOCKET '/tmp/echo.sock'，最后一行需要改成。

```
server.listen('/tmp/echo.sock');
```

如下是该对象可以触发的事件：

Event: 'connection'

function (stream) {}

当一个新连接建立后触发（发出）这个事件，stream 是 net.Stream 类的一个实例。

Event: 'close'

function () {}

当一个 SERVER 关闭的时候触发（发出）这个事件。

net.createServer(connectionListener)

建立一个新的 TCP SERVER。connectionListener 参数会自动设置为'connection'事件的监听函数。

server.listen(port, [host], [callback])

在指定端口和主机上接受一个连接请求。如果 HOST 这个参数忘记写了，该 SERVER 将在机器的所有 IPV4地址(INADDR_ANY)上接受连接请求。

这是一个异步函数，最后一个参数‘callback’将在服务器被绑定（应当是指当 listen 正常执行完并且进入正常监听流程后）后被调用。

server.listen(path, [callback])

建立一个 UNIX SOCKET SERVER 并监听在指定路径上的连接。

这个函数是一个异步方法，最后一个参数‘callback’将在服务器被绑定（应当是指当 listen 正常执行完并且进入正常监听流程后）后被调用。

server.listenFD(fd)

建立一个 SERVER 并监听在给定的文件描述符上。

这个文件描述符必须是已经在其上调用过 bind(2)、listen(2)系统调用的。

server.close()

停止服务器，此函数是异步的。服务器在触发'close'事件后才会最终关闭。

net.Stream TCP 流模块

这个对象是对 TCP 或者 UNIX SOCKET 的抽象，它实现了全双工的流接口。net.Stream 可以由用户手动建立，并且作为一个客户端来使用(和 connect())，也可以被 node 建立并通过服务器的'connection'事件传递给用户。(译注：如 http.Server 的 connection 事件，会将 net.Stream 的实例当作参数传递给响应函数)

net.Stream 实例会发出下列事件：

Event: 'connect'

function () { }

当成功建立连接后触发此事件事件。详见 connect()。

Event: 'secure'

function () { }

当一个 stream 与其对等端安全建立一个 SSL 握手后触发。

Event: 'data'

function (data) { }

当接收到数据时触发该事件，数据会是 Buffer 或者 String，数据的编码通过 stream.setEncoding()来设计(查看可读流那部分文章来获得更多信息)。

Event: 'end'

function () { }

当 stream 发出一个 FIN 包后触发此事件。这个事件发出后准备状态会变为‘只写’(writeOnly)。当这个事件被发出后，唯一能做的事情或许只是 call stream.end()了。

Event: 'timeout'

function () { }

当流因为不活动而超时时触发这个事件。这是唯一一个因为 stream 空闲而通知的事件，这个时候用户必须手动关闭这个连接。

参见：stream.setTimeout()

Event: 'drain'

function () { }

当写缓冲区变空的时候触发这个事件，这个事件可以用来控制/调节上传。

Event: 'error'

function (exception) { }

当发生一个错误时候触发。‘close’事件将跟随这个事件被发出。

Event: 'close'

function (had_error) { }

当 stream 被完全关闭时发出这个事件。参数 had_error 是一个用来标示 stream 关闭是否是因为传输错误所导致的标志。

net.createConnection(port, host='127.0.0.1')

构造一个新的 stream 对象并且打开一个 stream 到指定的端口和主机，如果第二个参数没有写，则假设主机为 localhost

建立连接后触发 connect 事件。

stream.connect(port, host='127.0.0.1')

在指定端口和主机打开一个 stream。createConnection()也可以建立连接，所以通常我们并不需要使用这个方法。只有在一个 stream 被关闭，并且你希望重新使用这个对象连接到其他主机的时候我们才用这个方法。

这个函数是异步的。在连接建立之后会触发‘connect’事件。如果在连接过程中产生问题，将产生'error'事件，而不会被触发'connect'。

stream.remoteAddress

这个字符串代表远程计算机的 IP 地址，例如'74.125.127.100' 或者 '2001:4860:a005::68'。

这个成员变量只存在于服务器端连接。

stream.readyState

stream.readyState 可以是'closed', 'open', 'opening', 'readOnly' 'writeOnly' 中的一个。

stream.setEncoding(encoding='null')

为接受到的数据设置编码格式(只能'ascii', 'utf8', 'base64'中的一个)。

stream.setSecure([credentials])

配合 crypto 模块提供的私钥和证书（在 peer authentication 中是 CA 证书），此方法可以为 stream 提供 https 支持。

如果 credentials 包含一个或多个数字认证中心证书（CA certificates），则 stream 将请求对等待方提交一个客户端证书作为 HTTPS 连接握手的一部分，证书的合法性及内容可以通过 verifyPeer()及 getPeerCertificate 来查看。

stream.verifyPeer()

根据被信任的证书列表或对方连接证书上下文的验证结果返回 true 或 false。

stream.getPeerCertificate()

返回用 JSON 结构详尽表述的对等方证书，其中包含一个证书的‘主题’，‘发布者’，'有效来源','有效目标'。

stream.write(data, encoding='ascii')

通过 stream 发送数据,第二个参数指定这个字符串实例的编码，缺省为 ASCII，因为实用 UTF8编码比较慢。

如果全部数据安全写入内核缓冲区则返回 true，如果全部或者部分数据仍然在用户空间排队则返回 false, 当缓冲区再次空闲的时候'drain'事件会被触发。

stream.end([data], [encoding])

半关闭 stream，也就是说，steram 发出一个 FIN 包。这个时候可能服务器还会发出一些数据。当 call 这个函数后,readyState 会变成'readOnly'。

如果使用了 data 这个参数,这就相当于 call stream.write(data, encoding), 其次后面跟随着 stream.end().stream.destroy()

请确保在这个 stream 上没有任何 I/O 活动。只有在错误发生的时候（为了调试错误）才需要调用此函数。

stream.pause()

暂停数据的读取.更确切的说，'data'事件将不会被发出,这个方法在控制上传的时候非常有用。

stream.resume()

当恢复读取数据。

stream.setTimeout(timeout)

设置 timeout 这么多毫秒作为 stream 的超时时间，默认情况下 net.Stream 没有超时时间。

当超时事件被触发，stream 将受到一个‘timeout’事件，但是连接将不会被断掉，用户必须执行 end()或者 destroy 来结束这个 stream。

如果 timeout 为0，会禁用超时。

stream.setNoDelay(noDelay=true)

禁止 Nagle algorithm(Nagle 运算模式被设计用来减少 LAN 和其它网络拥塞)，缺省情况下 TCP 连接使用 Nagle algorithm，此模式会在真正将数据发出前将其缓冲起来。设置 noDelay 将在每次 call stream.write()时立刻将数据连续发出。

stream.setKeepAlive(enable=false, [initialDelay])

允许/禁止 keep-alive 功能，在一个空闲 stream 首次收到 keepalive 之前可以设置 killalive 的延时。设置 initialDelay(毫秒)变量将设置接收到最后的数据包和首次 keepalive 探测之间的时差。设置为0则保持缺省（或者上一次的）设置的数值不变。

Crypto 加密模块

使用 require('crypto')来访问这个模块。

crypto 模块需要 node 所运行的运行支持 OpenSSL，该模块为使用安全证书实现 HTTPS 安全网络以及 HTTP 连接提供了支持。

模块同样为 OpenSSL 的 hash、hmac、cipher、decipher、sign 以及 verify 方法提供一层包装（以方便在 Node 中使用）。

crypto.createCredentials(details)

建立一个证书对象，参数 detail 是由键值对组成的字典。

key : 一个字符串，包含 PEM 编码的私钥

cert: 一个字符串，包含 PEM 编码的证书

ca : 一个包含 PEM 编码的、可信任的数字中心认证证书的字符串或者字符串列表

如果参数 details 中没有 'ca'，那么 node.js 将缺省使用 http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt 中给出的可信任的公钥。

crypto.createHash(algorithm)

通过参数 algorithm 指定算法建立并且返回一个哈希对象，可以用来产生哈希摘要。

algorithm 参数依赖于 node 运行平台上 OpenSSL 所支持的有效算法。例如 sha1,md5,sha256,sha512等，在最近发布的版本中, openssl list-message-digest-algorithms 将显示有效的算法摘要。

hash.update(data)

使用 data 更新哈希表。当以流的方式接受新数据时（数据可能被为分多次接收），可多次调用此方法。

hash.digest(encoding='binary')

计算所有传递来数据的哈希摘要。编码可以是'hex','binary'或者'base64'。

crypto.createHmac(algorithm, key)

通过指定算法(algorithm)和密钥(key)建立并返回一个加密的 hmac 对象。

和 createhash 函数一样，参数 algorithm 的选择依赖于 node 运行平台上 OpenSSL 所支持的有效算法，key 是要使用的 HMAC 私钥。

hmac.update(data)

更新指定数据(参数 data)的 hmac 的内容，当以流的方式接收新数据(参数 data)时，可多次调用此方法。

hmac.digest(encoding='binary')

计算所有传递来数据的 hmac 摘要。编码可以是'hex','binary'或者'base64' 。

crypto.createCipher(algorithm, key)

通过指定算法(algorithm)和密钥(key)建立并返回一个 cipher 对象

算法参数的内容依赖于 OPENSSL 所支持的有效算法，例如 aes192等等。OpenSSL 的 list-cipher-algorithms 将显示有效的 cipher 算法。

cipher.update(data, input_encoding='binary', output_encoding='binary')

更新参数 data 所代表的 cipher， input_encodin 是初始数据的编码，编码可以是'utf8','ascii'或者'binary'。output_encoding 参数指定了加密数据的输出编码，编码可以是'binary','base64'或者'hex'。

返回加密后的内容，当以流的方式接收新数据时，可多次调用此方法。

cipher.final(output_encoding='binary')

返回剩余的已加密内容，output_eocoding 可以是'binary','ascii','utf8'中的一个。

crypto.createDecipher(algorithm, key)

通过参数 algorithm 和 key 建立并返回一个 decipher 对象。这是前面 cipher 对象的一个镜像.

decipher.update(data, input_encoding='binary', output_encoding='binary')

更新参数 data 所代表的 decipher ， input_encoding 是初始数据的编码，编码可以是'binary','base64'或者'hex'。output_encoding 参数指定了已解密的铭文的输出编码，编码可以是'binary','base64'或者'hext'

decipher.final(output_encoding='binary')

返回其余解密后的文本。参数 output_encoding 是 'binary', 'ascii' or 'utf8'中的一个.

crypto.createSign(algorithm)

通过参数 algorithm 建立并返回一个 signing 对象。根据当前 openSSL 版本,penssl 的 list-public-key-algorithms 将显示 singning 的有效算法.例如 'RSA-SHA256'.

signer.update(data)

用参数 data 更新 signer 对象，当以流的方式接收新数据时，可多次调用此方法。

signer.sign(private_key, output_format='binary')

计算所有 signer 里已经更新的数据的签名。private_key 是一个字符串，包含用于签名的 PEM 编码的私钥。

返回用 output_format 指定编码的签名，编码可以是'binary', 'hex' or 'base64'

crypto.createVerify(algorithm)

通过指定 algorithm 建立并返回一个 verification 对象.这是上面 signing 对象的一个镜像。

verifier.update(data)

用新数据(参数 data)更新 verifyer 对象，当以流的方式接收新数据时，可多次调用此方法。

verifier.verify(public_key, signature, signature_format='binary')

用包含有 PEM 编码的公钥，即参数 public_key 来验证电子签名数据。参数 signature 是先前用此数据计算的签名，参数 signature_format 可以是'binary', 'hex' or 'base64'.

根据签名和公钥对数据的验证结果返回 true 或 false。

DNS 域名解析

（译著，Windows 版本的 NodeJS 暂时没有实现 DNS 功能）

使用 require('dns')来访问这个模块。

下面是一个先解析'www.google.com'，然后将解析出来的 IP 地址再做反向解析。

```
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, domains) {
      if (err) {
        console.log('reverse for ' + a + ' failed: ' +
          err.message);
      } else {
        console.log('reverse for ' + a + ': ' +
          JSON.stringify(domains));
      }
```

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

dns.lookup(domain, family=null, callback)

将一个域名(例如. 'google.com')解析成为找到的第一个 A(IPv4)或者 AAAA(IPv6)记录。

回调函数的有(err, address, family)这三个参数。address 参数是一个代表 IPv4或 IPv6的地址的字符串。family 是一个表示地址版本的整形数字4或6(并不一定是解析域名时传递的数字)。

dns.resolve(domain, rrtype='A', callback)

将参数 domain(比如'google.com')按照参数 rrtype 所指定数据类型解析到一个数组中。合法的类型为 A（IPV4地址），AAAA（IPV6地址），MX（mail exchange records）,TXT(text records)，SRV（SRV records），和 PTR（used for reveres IP lookups）。

回调函数（callback）接受两个参数：err 和 address。参数 address 中的每一项根据记录类型(record type)分割，在下面 lookup 方法的文档里有详细的解释。

当有错误发生时，参数 err 的内容是一个 Error 对象的实例，err 的 errno 属性是下面错误代码列表中的一个，err 的 message 属性是一个用英语表述的错误解释。

dns.resolve4(domain, callback)

与 dns.resolve()类似,但是仅对 IPV4地址进行查询(A records)。addresses 是一个 IPV4地址数组(例如['74.125.79.104', '74.125.79.105', '74.125.79.106'])

dns.resolve6(domain, callback)

除了这个函数是对 IPV6地址的查询外与 dns.resolve4()很类似(一个 AAAA 查询)。

dns.resolveMx(domain, callback)

与 dns.resolve()很类似.但是仅做 mail exchange 查询(MX 类型记录)。

回调函数的参数 address 是一个 MX 类型记录的数组,每个记录有一个优先级属性和一个交换属性(类似[{'priority': 10, 'exchange': 'mx.example.com'},...])

dns.resolveTxt(domain, callback)

与 dns.resolve()很相似,但是仅可以进行 text 查询(TXT 记录).地址是一个对于域来说有效的 text 记录数组（类似['v=spf1 ip4:0.0.0.0 ~all']）

dns.resolveSrv(domain, callback)

与 dns.resolve()很类似,但仅是只查询 service 记录(srv records)。地址是一个对于域来说有效的 SRV 记录的数组，SRV 记录的属性有优先级、权重、端口，名字(例如 [{'priority': 10, {'weight': 5, 'port': 21223, 'name':

'service.example.com'}, ...])

dns.reverse(ip, callback)

反向解析一个 IP 地址到一个域名数组。

callback 参数有两个参数(err,domains)。

如果发生了错误,err 为 Error 对象的实例。

每个 DNS 查询可以返回如下错误代码：

- 超时、返回 SERVFAIL 或者类似的错误
- 返回内容里有乱码
- 域名不存在
- 域名存在但是没有所请求的查询类型的数据
- 处理过程中内存溢出
- 查询语句异常

dgram 数据报

要使用数据包 SOCKET 需要调用 require('dgram'),数据报一般用来处理 IP/UDP 信息，但是数据报也可用在 UNIX DOMAIN SOCKETS 上

Event: 'message'

function (msg, rinfo) { }

Emitted when a new datagram is available on a socket. msg is a Buffer and rinfo is an object with the sender's address information and the number of bytes in the datagram.

当一个 SOCKET 接收到一个新的数据包的时候触发此事件， msg 是缓冲区变量,rinfo 是一个包含了发送者地址信息以及数据报字节长度的对象.

Event: 'listening'

function () { }

当一个 SOCKET 开始监听数据报的时候触发,当 UDP SOCKET 建立后就会触发这个事件。而 UNIX DOMAIN SOCKET 直到在 SOCKET 上调用了 bind()方法才会触发这个消息.

Event: 'close'

function () { }

当一个 SOCKET 使用 close()方法关闭时触发此事件.在此事件之后此 SOCKET 不会有任何消息事件被触发.

dgram.createSocket(type, [callback])

建立一个指定类型的数据报 SOCKET,有效类型有:udp4,udp6,unix_dgram

callback 作为一个可选项，可作为 message 事件的监听器被加入。

dgram.send(buf, offset, length, path, [callback])

对于 unix domain datagram xockets 来说,他的目标地址是一个使用文件系统表示的路径名,callback 作为一个可选项会在系统调用 sendto 完毕后被触发。除非 callback 被触发，否则重复使用 buf 是很不安全的。要注意除非这个 socket 已经使用 bind()方法绑定到一个路径名上，否则这个 SOCKET 无法接收到任何信息。

下面是一个通过 unix domain socket /var/run/syslog 发送消息到 syslogd 的例子：

```
var dgram = require('dgram'),
message = new Buffer("A message to log.");
client = dgram.createSocket("unix_dgram");

client.send(message, 0, message.length, "/var/run/syslog",
   function (err, bytes) {
      if (err) {
         throw err;
      }
      console.log("Wrote " + bytes + " bytes to socket.");
});
```

从 MESSAGE 中偏移为0的地方开始，长度为 MESSAGE.LENGTH 的这些内容通过/var/run/syslog 发送 系统调用发送后，将调用 CALLBACK，如果有错误则抛出异常，否则 console.log 实际发送了多少个字节。

dgram.send(buf, offset, length, port, address, [callback])

对于 UDPSOCKETS 来说，目标端口和 IP 地址是必须要指定的，可以用字符串来指定地址参数，并且这个参数是可以通过 DNS 解析的，CALLBACK 作为可选项可以检测到任何 DNS 错误和是否 BUF 重复使用了.请记住 DNS 搜索将会使 SEND 动作最少延迟到下一个执行时间片发生， ，唯一能确定已经 SEND 得方法是使用 CALLBACK

下面是一个发送 UDP 数据包到本机一个随机端口的例子

```
var dgram = require('dgram'),
message = new Buffer("Some bytes");
client = dgram.createSocket("udp4");
client.send(message, 0, message.length, 41234, "localhost");
client.close();
```

dgram.bind(path)

只有在 Unxi DOMAIN DATAGRAM SOCKET 中使用,开始在一个指定路径上监听一个 SOCKET 过来的的数据报。要记得，客户端可以不是用 BIND()方法而直接调用 SEND()方法，但是不使用 BIND()方法是无法接收到任何信息的。

下面是一个使用 UNIX DOMAIN 数据包服务器来做接受信息回显的例子：

```
var dgram = require("dgram");
var serverPath = "/tmp/dgram_server_sock";
var server = dgram.createSocket("unix_dgram");

server.on("message", function (msg, rinfo) {
  console.log("got: " + msg + " from " + rinfo.address);
  server.send(msg, 0, msg.length, rinfo.address);
});

server.on("listening", function () {
  console.log("server listening " + server.address().address);
})

server.bind(serverPath);
```

下面是一个 UNIX DOMAIN DATAGRAM 客户端与服务器交互的例子

```
var dgram = require("dgram");
var serverPath = "/tmp/dgram_server_sock";
var clientPath = "/tmp/dgram_client_sock";

var message = new Buffer("A message at " + (new Date()));

var client = dgram.createSocket("unix_dgram");

client.on("message", function (msg, rinfo) {
  console.log("got: " + msg + " from " + rinfo.address);
});

client.on("listening", function () {
  console.log("client listening " + client.address().address);
  client.send(message, 0, message.length, serverPath);
});

client.bind(clientPath);
```

dgram.bind(port, [address])

对于 UDP SOCKETS，这个方法会在指定端口和可选地址上监听，如果地址没有指定，则系统会尝试监听所有有效地址。

下面是一个监听在41234端口的 UDP 服务器的例子

```
var dgram = require("dgram");
```

```
var server = dgram.createSocket("udp4");
var messageToSend = new Buffer("A message to send");

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
```

dgram.close()

这个方法关闭非延迟的 SOCKET 并且停止在其上监听数据。即使没有调用 BIND()方法 UDP SOCKET 也会自动监听消息。

dgram.address()

返回包含 SOCKET 地址信息的一个对象，对于 UDP SOCKETS 来说，这个对象将包含地址和端口，对于 UNIX DOMAIN SOCKETS 来说，这个对象仅包含地址。

dgram.setBroadcast(flag)

设置或者清除 SO_BROADCAST 选项，当这个选项设置后，UDP 包可以发送到本地接口的广播地址。

dgram.setTTL(ttl)

设置 IP_TTL 这个选项，TTL 表示“存活时间”，但是在这个上下文环境中，他也可以指定 IP 的 HOPS（每个节点在转发数据包时的消耗。如果 Hop limit 消耗到0，则取消数据包）来确定一个数据包大致允许经过多少节点。每经过个路由器或者网关都会减少 TTL 数值，如果 TTL 被一个路由器减少到0，这个数据报将不会继续转发，修改 TTL 数值经常用来当网络探针或者作为数据多播使用

ttl 用来设置 HOPS 的数值从1到255，大多数系统缺省会设置为64

Assert 断言

此模块是用来编写单元测试的，你可以使用 require('assert')访问该模块。

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

此函数使用参数 operator 测试 actual 和 expected 是否相等。

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])

此函数为简便测试函数，使用操作符‘==’比较 actual 和 expected 是否相等。

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

此函数为简便测试函数，使用操作符’！=‘比较 actual 和 expected 是否相等。

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

执行深度比较是否相等。

译注：三种比较的不同之处（感谢 tytsim）：

```
a = {'a':'1','b':'2'};
b = {'b':'2','a':'1'};
equal(a, b)  返回 false
deepEqual(a, b)  返回 true
strictEqual(a, b)  返回 false
```

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

深度比较是否不相等。

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

此函数使用操作符‘===’严格比较是否两参数相等。

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])

测试代码块，期待其抛出异常。

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

测试代码块，期待其不抛出异常。

assert.ifError(value)

判断参数 value 是否为 false，如果为 true 则抛出异常。通常用在回调函数中判断是否发生了错误。

译著，实现代码：　assert.ifError = function (err) { if (err) {throw err; }};

Path 模块

此模块包含很多用于处理文件路径的小工具。你可以通过 require('path')使用该模块。 它提供了如下函数：

path.join([path1], [path2], [...])

将所有参数连接在一起并解析生成新的路径。

示例：

```
node> require('path').join(
...     '/foo', 'bar', 'baz/asdf', 'quux', '..')
'/foo/bar/baz/asdf'
```

path.normalizeArray(arr)

转化路径的各部分，将'..'和'.'替换为实际的路径。

示例：

```
path.normalizeArray([,
   'foo', 'bar', 'baz', 'asdf', 'quux', '..'])
// returns
[ , 'foo', 'bar', 'baz', 'asdf ]
```

path.normalize(p)

转化路径字符串，将'..'和'.'替换为实际的路径。

示例：

```
path.normalize('/foo/bar/baz/asdf/quux/..')
// returns
'/foo/bar/baz/asdf'
```

path.dirname(p)

返回路径中代表文件夹的部分，同 Unix 的 dirname 命令类似。

示例：

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns
'/foo/bar/baz/asdf'
```

path.basename(p, [ext])

返回路径中的最后一部分。同 Unix 命令 bashname 类似。

示例“

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns
'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns
'quux'
```

path.extname(p)

返回路径中文件的后缀名，即路径中最后一个'.'之后的部分。如果一个路径中并不包含'.'或该路径只包含一个'.'且这个'.'为路径的第一个字符，则此命令返回空字符串。如下所示：

```
path.extname('index.html')
// returns
'.html'

path.extname('index')
// returns
```

path.exists(p, [callback])

检测给定的文件路径是否存在。然后传递结果(true 或 false)给回调函数。

```
path.exists('/etc/passwd', function (exists) {
  sys.debug(exists ? "it's there" : "no passwd!");
});
```

URL 模块

此模块用于解析 URL，你可以通过 require('url')来使用它。

由于各 URL 不尽相同，经过解析的 URL 对象有如下部分或者全部的域。任何 URL 中不包含的域将不会出现在解析后的 URL 对象中。如下所示：

'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'

- href

原始的 URL。例如：'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'

- protocol

请求的协议。例如：'http'

- host

URL 中主机地址部分，包括端口和验证信息。 例如：'user:pass@host.com:8080'

- auth

URL 中的验证信息。例如：'user:pass'

- hostname

仅仅包括主机地址。例如：'host.com'

- port

主机的端口号。例如：'8080'

- pathname

URL 中的路径(path)部分，即紧跟在主机地址之后查询参数之前的部分，包括路径开头的斜线。例如：'/p/a/t/h'

- search

URL 中的参数部分，包括开头的问号。例如：*?query=string*

- query

查询字符串中的参数部分或者是解析过的查询字串对象（译注：根据解析 URL 时设置的参数不同，此属性的内容也不同）。例如：'query=string' 或者 {'query':'string'}

- hash

URL 中的锚记部分，包括开头的井号。例如：'#hash'

URL 模块还提供如下方法：

url.parse(urlStr, parseQueryString=false)

此函数接受一个 URL 字符串并返回一个对象。如果第二个参数传递 true，node 会使用 querystring 模块解析查询字符串。

url.format(urlObj)

此函数接受一个 URL 对象，并返回一个格式化后的 URL 字符串。

url.resolve(from, to)

此函数接受一个 base URL 和一个 href URL，并像浏览器解析锚记一样解析它们。

Query String 查询字串

此模块能处理查询字串(query strings)，提供以下方法：

querystring.stringify(obj, sep='&', eq='=', munge=true)

序列化对象至查询字串。选择性地覆写默认分割符和增补字符(assignment characters)。

例子：

```
querystring.stringify({foo: 'bar'})
```

```
// returns
'foo=bar'

querystring.stringify({foo: 'bar', baz: 'bob'}, ';', ':')
// returns
'foo:bar;baz:bob'
```

此方法默认由阵列和对象(obj)排列成 PHP/Rails 风格的查询字串，例子：

```
querystring.stringify({foo: ['bar', 'baz', 'boz']})
// returns
'foo%5B%5D=bar&foo%5B%5D=baz&foo%5B%5D=boz'

querystring.stringify({foo: {bar: 'baz'}})
// returns
'foo%5Bbar%5D=baz'
```

若希望停用字元解析 (例如当生成参数予 Java servlet 时)，可以设置 munge(字元解析) 参数成 false(假值)，例子：

```
querystring.stringify({foo: ['bar', 'baz', 'boz']}, '&', '=', false)
// returns
'foo=bar&foo=baz&foo=boz'
```

注意当 munge(字元解析) 为 false 时，参数名称仍会被解析。

querystring.parse(str, sep='&', eq='=')

反序列化查询字串至对象。选择性地覆写默认分割符和增补字符(assignment characters)。

```
querystring.parse('a=b&b=c')
// returns
{ 'a': 'b'
, 'b': 'c'
}
```

这方法可以解析已解析和未解析的查询字串。

querystring.escape

querystring.stringify 中所使用的 escape 方法。您可以覆写它。

querystring.unescape

querystring.parse 中所使用的 unescape 方法。您可以覆写它。

REPL 交互执行

node 的“读入、运行、输出循环模式(REPL)”既可以单独执行也很容易嵌入其它程序中。 REPL 提供了一种交互式执行 Javascript 并查看结果的模式。这种模式可以用来调试、测试或者仅仅用来某些新特性。

如果直接执行 node 而不跟任何参数就会进入 REPL 模式。它类似于简化的 emacs 行编辑模式。

```
mjr:~$ node
Type '.help' for options.
node> a = [ 1, 2, 3];
[ 1, 2, 3 ]
node> a.forEach(function (v) {
...     console.log(v);
...     });
1
2
3
```

要使用高级行编辑功能，设置环境变量 NODE_NO_READLINE=1并执行 node。这样 REPL 就会使用标准终端设置，如此一来你就可以使用 rlwarp 来执行高级行编辑。

示例，你可以在 bashrc 文件中添加如下指令：

```
alias node="env NODE_NO_READLINE=1 rlwrap node"

repl.start(prompt='node> ', stream=process.openStdin())
```

REPL 执行时将使用 prompt 参数的值作为输入提示符，并使用 stream 参数执行所有 I/O 操作。 prompt 为可选参数，默认值为'node>', stream 为可选参数，默认值为 process.openStdin()；

同一个 node 进程可以启动多个 REPL，每个 REPL 将会共享部分全局对象，但是它们都有自己唯一的 I/O。

示例，分别使用标准输出（控制台）、Unix Socket 和 TCP Socket 启动 REPL：

```
var net = require("net"),
      repl = require("repl");

connections = 0;

repl.start("node via stdin> ");

net.createServer(function (socket) {
   connections += 1;
   repl.start("node via Unix socket> ", socket);
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
   connections += 1;
```

```
  repl.start("node via TCP socket> ", socket);
}).listen(5001);
```

在控制台执行上述程序将使用标准输入（当前控制台）启动 REPL，同时其他 REPL 客户端可以通过 Unix socket 或者 TCP socket 连接。 你可以使用 telnet 连接到 TCP socket，用 socat 连接到 Unix 或 TCP sockets。

不使用标准输入（控制台）而是用 Unix socket 服务启动 REPL，可以让你轻易连接到一个长时间运行的 node 进程而不用重新启动该进程。

REPL Features REPL 支持的特性

在 REPL 执行时，可以输入 Control+D 退出。你也可以输入跨越多行的表达式。

特殊标量'_'（下划线）保存了上一个表达式执行后的值。

```
node> [ "a", "b", "c" ]
[ 'a', 'b', 'c' ]
node> _.length
3
node> _ += 1
4
```

REPL 提供了访问全局作用域内任何变量的能力，你也可以通过将变量赋值给 REPL 的 context 对象来向 REPL 暴露该变量。 例如：

```
// repl_test.js
var repl = require("repl"),
      msg = "message";

repl.start().context.m = msg;
```

对于 REPL 来说，context 对象中的值就犹如在本地作用域内：

```
mjr:~$ node repl_test.js
node> m
'message'
```

如下是一些 REPL 命令：

- .break - 当想要放弃当前输入的多行命令时，可以使用.break 命令重新开始输入。
- .clear - 将 context 重置为空对象并清空（当前正在输入的）多行表达式。
- .exit - 管理 I/O 流，此操作将关闭 REPL。
- .help - 显示特殊命令的帮助。

Modules 模块

Node 使用 CommonJS 的模块系统。

Node 同时也拥有一个简单的模块加载系统。在 Node 的世界里，文件和模块是一一对应的，比如，以下的程序 foo.js 会向大家演示加载同一目录下的 circle.js 模块。

The contents of foo.js:

foo.js 的代码：

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

The contents of circle.js:

cirle.js 的代码：

```
var PI = 3.14;

exports.area = function (r) {
  return PI * r * r;
};

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

模块 circle.js 有两个方法 area()和 circumference()。为了使其对外可见，将其导出到一个特殊的对象 exprots（还可以用 this 来替代 exports)。此模块中的本地变量是私有的。在这个例子中，PI 便是 circle 模块的似有变量，puts()方法则是引用自系统自带模块'sys'。没有'./'前缀的模块都是写内建的。我们将会详细讲解这个特性。

使用'./'前缀加载的模块必须和加载模块的文件位于同一文件夹下，所以将 circle.js 和 foo.js 放在同一目录下。

如果不用'./'前缀，比如 require('assert')，那么则会在 require.paths 数组指定的路径下寻找，require.paths 在我的机器上输出如下：

['/home/ryan/.node_libraries'] ['/home/ryan/.node_libraries']

所以当呼叫 require('assert')是，系统会沿着如下路径寻找该模块：

```
    * 1: /home/ryan/.node_libraries/assert.js
    * 2: /home/ryan/.node_libraries/assert.node
    * 3: /home/ryan/.node_libraries/assert/index.js
    * 4: /home/ryan/.node_libraries/assert/index.node
```

有'.node'后缀的模块是 Node 系统的二进制模块:可以参考'Addons'来得到更多信息。'index.js'允许我们把一个模块作为一个目录打包。

require.paths 可以在运行时修改或者在类 UNIX 系统下通过修改 NODE_PATH 环境变量来达到同样的目的。

Addons 扩展

扩展是动态链接的共享对象，可以与 C 和 C++ 库链合。目前 API 是相当复杂，涉及数个库的知识：

- V8 JavaScript, C++ 库。能在 C++中与 JavaScript 链合：创建对象，调用函数等。文档大部份存放於 v8.h 的标头文件 (deps/v8/include/v8.h)。
- libev, C 语言 事件循环库。(提供一个能当文档描述符有特定事件发生，或等待时间超过时，执行回调函数的机制。) 当 I/O 执行时，需要使用 libev。Node 利用 EV_DEFAULT 事件循环。文档存放於 http://cvs.schmorp.de/libev/ev.html。
- libeio，C 语言 执行绪集区库。能使 POSIX 系统异步执行。由於通常已封装於 src/file.cc ，所以毋必要使用。若需使用，查阅标头文件 deps/libeio/eio.h。
- 内部 Node 库，最主要的是 node::ObjectWrap 类，经常用作参考。
- 其他，查阅 deps/。

Node 静态编译所有组件成可执行文件。当您编译您的模块时，您不必考虑以上库的连结。

制作一个小型扩展能达到以下效用(C++ 除外)：

```
exports.hello = 'world';
```

创建文件 hello.cc：

```
#include <v8.h>

using namespace v8;

extern "C" void
init (Handle<Object> target)
{
  HandleScope scope;
  target->Set(String::New("hello"), String::New("World"));
}
```

此源文件需要编译成 hello.node（二进制扩展）。需要创建一个 python 文件 wscript：

```
srcdir = '.'
blddir = 'build'
VERSION = '0.0.1'

def set_options(opt):
  opt.tool_options('compiler_cxx')

def configure(conf):
  conf.check_tool('compiler_cxx')
  conf.check_tool('node_addon')
```

```
def build(bld):
  obj = bld.new_task_gen('cxx', 'shlib', 'node_addon')
  obj.target = 'hello'
  obj.source = 'hello.cc'
```

执行 node-waf configure build 将会创建您的扩展文件至 build/default/hello.node。

node-waf 是 http://code.google.com/p/waf/[WAF]，基於 python 的编译系统。node-waf 为使用者提供轻易。

所有 Node 扩展必须输出一函数 init ，并包含此声明：

```
extern 'C' void init (Handle<Object> target)
```

至现时为止，此乃完整的扩展说明文件。请阅 http://github.com/ry/node_postgres 以取得真实范例。

Appendix - Third Party Modules 附录：第三方模块

目前为止，已经有很多基于 Node 的第三方模块，模块的代码主仓库地址为 http://github.com/ry/node/wiki/modules[Wiki 页面]

本目录旨在作为一个入门性的向导，帮助刚接触 Node 的用户快速查找他们想要的高质量的模块。但是这并不是一个完整的模块列表，你可能会发现你在其他地方找到的列表会比这儿这更全。

- 模块安装器: npm
- HTTP 中间件: Connect
- Web 框架: Express
- Web Sockets: Socket.IO
- HTML 解析: HTML5
- mDNS/Zeroconf/Bonjour
- RabbitMQ, AMQP
- mysql
- 序列化: msgpack
- 解析器: Apricot
- 调试器: ndb 是一个命令行界面的调试器；inspector 则是一个基于 web 的调试工具。
- pcap binding
- ncurses
- Testing/TDD/BDD: vows, expresso, mjsunit.runner

欢迎各位补全该列表。