Node.js Manual & Documentation

nodejsBBS.com

Table Of Contents

    - process.stdout
    - process.stderr
    - process.stdin
    - process.argv
    - process.execPath
    - process.chdir(directory)
    - process.cwd()
    - process.env
    - process.exit(code=0)
    - process.getgid()
    - process.setgid(id)
    - process.getuid()
    - process.setuid(id)
    - process.version
    - process.installPrefix
    - process.kill(pid, signal='SIGTERM')
    - process.pid
    - process.title
    - process.platform
    - process.memoryUsage()
    - process.nextTick(callback)
    - process.umask([mask])
- util 工具模块
    - util.debug(string)
    - util.log(string)
    - util.inspect(object, showHidden=false, depth=2)
    - util.pump(readableStream, writableStream, [callback])
    - util.inherits(constructor, superConstructor)
- Events 事件模块
    - events.EventEmitter
        - emitter.addListener(event, listener)
        - emitter.on(event, listener)
        - emitter.once(event, listener)
        - emitter.removeListener(event, listener)
        - emitter.removeAllListeners(event)
        - emitter.setMaxListeners(n)
        - emitter.listeners(event)
        - emitter.emit(event, [arg1], [arg2], [...])
        - Event: 'newListener' 事件: 'newListener'
- Buffers 缓冲器
    - new Buffer(size)
    - new Buffer(array)
    - new Buffer(str, encoding='utf8')
    - buffer.write(string, offset=0, encoding='utf8')

    - cipher.update(data, input_encoding='binary', output_encoding='binary')
    - cipher.final(output_encoding='binary')
    - crypto.createDecipher(algorithm, key)
    - decipher.update(data, input_encoding='binary', output_encoding='binary')
    - decipher.final(output_encoding='binary')
    - crypto.createSign(algorithm)
    - signer.update(data)
    - signer.sign(private_key, output_format='binary')
    - crypto.createVerify(algorithm)
    - verifier.update(data)
    - verifier.verify(cert, signature, signature_format='binary')
- TLS (SSL) TLS (SSL)模块
    - s = tls.connect(port, [host], [options], callback)
    - tls.Server
        - tls.createServer(options, secureConnectionListener)
        - Event: 'secureConnection' 事件: 'secureConnection'
        - server.listen(port, [host], [callback])
        - server.close()
        - server.maxConnections
        - server.connections
- File System 文件系统模块
    - fs.rename(path1, path2, [callback])
    - fs.renameSync(path1, path2)
    - fs.truncate(fd, len, [callback])
    - fs.truncateSync(fd, len)
    - fs.chmod(path, mode, [callback])
    - fs.chmodSync(path, mode)
    - fs.stat(path, [callback])
    - fs.lstat(path, [callback])
    - fs.fstat(fd, [callback])
    - fs.statSync(path)
    - fs.lstatSync(path)
    - fs.fstatSync(fd)
    - fs.link(srcpath, dstpath, [callback])
    - fs.linkSync(srcpath, dstpath)
    - fs.symlink(linkdata, path, [callback])
    - fs.symlinkSync(linkdata, path)
    - fs.readlink(path, [callback])
    - fs.readlinkSync(path)
    - fs.realpath(path, [callback])
    - fs.realpathSync(path)
    - fs.unlink(path, [callback])

    - dns.resolve6(domain, callback)
    - dns.resolveMx(domain, callback)
    - dns.resolveTxt(domain, callback)
    - dns.resolveSrv(domain, callback)
    - dns.reverse(ip, callback)
- UDP / Datagram Sockets 数据报套接字模块
    - Event: 'message' 事件：'message'
    - Event: 'listening' 事件：'listening'
    - Event: 'close' 事件：'close'
    - dgram.createSocket(type, [callback])
    - dgram.send(buf, offset, length, path, [callback])
    - dgram.send(buf, offset, length, port, address, [callback])
    - dgram.bind(path)
    - dgram.bind(port, [address])
    - dgram.close()
    - dgram.address()
    - dgram.setBroadcast(flag)
    - dgram.setTTL(ttl)
    - dgram.setMulticastTTL(ttl)
    - dgram.setMulticastLoopback(flag)
    - dgram.addMembership(multicastAddress, [multicastInterface])
    - dgram.dropMembership(multicastAddress, [multicastInterface])
- HTTP HTTP 模块
- http.Server
    - Event: 'request' 事件：'request'
    - Event: 'connection' 事件：'connection'
    - Event: 'close' 事件：'close'
    - Event: 'request' 事件：'request'
    - Event: 'checkContinue' 事件：'checkContinue'
    - Event: 'upgrade' 事件：'upgrade'
    - Event: 'clientError' 事件：'clientError'
    - http.createServer(requestListener)
    - server.listen(port, [hostname], [callback])
    - server.listen(path, [callback])
    - server.close()
- http.ServerRequest
    - Event: 'data' 事件：'data'
    - Event: 'end' 事件：'end'
    - request.method
    - request.url
    - request.headers
    - request.trailers
    - request.httpVersion
    - request.setEncoding(encoding=null)

Synopsis 概要

An example of a web server written with Node which responds with 'Hello World':

下边是一个用 Node 编写的对所有请求简单返回'Hello World‘的 web 服务器例子：

```
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/');
```

To run the server, put the code into a file called `example.js` and execute it with the node program
要运行这个服务器程序，只要将上述代码保存为文件 `example.js` 并用 node 程序执行此文件：

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

All of the examples in the documentation can be run similarly.

此文档中所有例子均可用同样的方法运行。 ## Global Objects 全局对象

These object are available in the global scope and can be accessed from anywhere.

这些对象在全局范围内均可用，你可以在任何位置访问这些对象。

global

The global namespace object.

全局命名空间对象

In browsers, the top-level scope is the global scope. That means that in browsers if you're in the global scope `var something` will define a global variable. In Node this is different. The top-level scope is not the global scope; `var something` inside a Node module will be local to that module.
在浏览器中，顶级作用域为全局作用域，在全局作用域下通过 `var something` 即定义了一个全局变量。但是在 Node 中并不如此，顶级作用域并非是全局作用域，在 Node 模块中通过 `var something` 定义的变量仅作用于该模块。

process

The process object. See the 'process object' section.

进程对象，参见'process object'章节。

require()

To require modules. See the 'Modules' section.

加载模块，参见'Modules'章节。

require.resolve()

Use the internal `require()` machinery to look up the location of a module, but rather than loading the module, just return the resolved filename.
使用内部函数 `require()` 的机制查找一个模块的位置，而不用加载模块，只是返回解析后的文件名。

require.paths

An array of search paths for `require()`. This array can be modified to add custom paths.

`require()`的搜索路径数组，你可以修改该数组添加自定义的搜索路径。
Example: add a new path to the beginning of the search list

例如：将一个新的搜索路径插入到搜索列表的头部。

```
require.paths.unshift('/usr/local/node');
```

__filename

The filename of the script being executed. This is the absolute path, and not necessarily the same filename passed in as a command line argument.

当前正在执行的脚本的文件名。这是一个绝对路径，可能会和命令行参数中传入的文件名不同。

Example: running `node example.js` from `/Users/mjr`
例如：在目录`/Users/mjr`下运行`node example.js`

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

__dirname

The dirname of the script being executed.

当前正在执行脚本所在的目录名。

Example: running `node example.js` from `/Users/mjr`
例如：在目录`/Users/mjr`下运行`node example.js`

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

module

A reference to the current module. In particular `module.exports` is the same as the `exports` object. See `src/node.js` for more information.
指向当前模块的引用。特别的，当你通过 `module.exports` 和 `exports` 两种方式访问的将是同一个对象，参见 `src/node.js`。

Timers 定时器

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

To schedule execution of `callback` after `delay` milliseconds. Returns a `timeoutId` for possible use with `clearTimeout()`. Optionally, you can also pass arguments to the callback.
设定一个 `delay` 毫秒后执行 `callback` 回调函数的计划。返回值 `timeoutId` 可被用于 `clearTimeout()`。可以设定要传递给回调函数的参数。

clearTimeout(timeoutId)

Prevents a timeout from triggering.

清除定时器，阻止指定的 timeout（超时）定时器被触发。

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

To schedule the repeated execution of `callback` every `delay` milliseconds. Returns a `intervalId` for possible use with `clearInterval()`. Optionally, you can also pass arguments to the callback.
设定一个每 `delay` 毫秒重复执行 `callback` 回调函数的计划。返回值 `intervalId` 可被用于 `clearInterval()`。可以设定要传递给回调函数的参数。

clearInterval(intervalId)

Stops a interval from triggering.

清除定时器，阻止指定的 interval（间隔）定时器被触发。 ## Modules 模块

Node uses the CommonJS module system. Node has a simple module loading system. In Node, files and modules are in one-to-one correspondence. As an example, `foo.js` loads the module `circle.js` in the same directory.
Node 使用 CommonJS 模块系统。Node 有一个简单的模块装载系统，在 Node 中，文件和模块是一一对应的。下面的例子展示了 `foo.js` 文件如何在相同的目录中加载 `circle.js` 模块。

The contents of `foo.js`:
`foo.js` 的内容为：

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

The contents of `circle.js`:
`circle.js` 的内容为：

```
var PI = Math.PI;

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

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

The module `circle.js` has exported the functions `area()` and `circumference()`. To export an object, add to the special `exports` object.
`circle.js` 模块输出了 `area()`和 `circumference()`两个函数，为了以对象的形式输出，需将要输出的函数加入到一个特殊的 `exports` 对像中。
Variables local to the module will be private. In this example the variable `PI` is private to `circle.js`.
模块的本地变量是私有的。在上面的例子中，变量 `PI` 就是 `circle.js` 私有的。

Core Modules 核心模块

Node has several modules compiled into the binary. These modules are described in greater detail elsewhere in this documentation.

Node 有一些编译成二进制的模块。这些模块在这篇文档的其他地方有详细描述。

The core modules are defined in node's source in the `lib/` folder.
核心模块在 node 源代码中的 lib 文件夹下。

Core modules are always preferentially loaded if their identifier is passed to `require()`. For instance, `require('http')` will always return the built in HTTP module, even if there is a file by that name.
核心模块总是被优先加载，如果它们的标识符被`require()`调用。例如，`require('http')`将总是返回内建的 HTTP 模块，即便又一个同名文件存在。

File Modules 文件模块

If the exact filename is not found, then node will attempt to load the required filename with the added extension of `.js`, and then `.node`. `.js` files are interpreted as JavaScript text files, and `.node` files are interpreted as compiled addon modules loaded with `dlopen`.
如果没有找到确切的文件名，node 将尝试以追加扩展名`.js`后的文件名读取文件，如果还是没有找到则尝试追加扩展名`.node`。`.js`文件被解释为 JavaScript 格式的纯文本文件，`.node`文件被解释为编译后的 addon（插件）模块，并使用`dlopen`来加载。
A module prefixed with `'/'` is an absolute path to the file. For example, `require('/home/marco/foo.js')` will load the file at `/home/marco/foo.js`.
以`'/'`为前缀的模块是一个指向文件的绝对路径，例如`require('/home/marco/foo.js')`将加载文件`/home/marco/foo.js`。
A module prefixed with `'./'` is relative to the file calling `require()`. That is, `circle.js` must be in the same directory as `foo.js` for `require('./circle')` to find it.
以`'./'`为前缀的模块是指向文件的相对路径，相对于调用`require()`的文件。也就是说为了使`require('./circle')`能找到正确的文件，`circle.js`必须位于与`foo.js`相同的路径之下。
Without a leading '/' or './' to indicate a file, the module is either a "core module" or is loaded from a `node_modules` folder.
如果标明一个文件时没有 '/' 或 './'前缀，该模块或是"核心模块"，或者位于`node_modules`目录中。

Loading from `node_modules` Folders 从`node_modules`目录中加载

If the module identifier passed to `require()` is not a native module, and does not begin with `'/'`, `'../'`, or `'./'`, then node starts at the parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location.
如果传递到`require()`的模块标识符不是一个核心模块，并且不是以`'/'`，`'../'`或`'./'`开头，node 将从当前模块的父目录开始，在其`/node_modules`子目录中加载该模块。

If it is not found there, then it moves to the parent directory, and so on, until either the module is found, or the root of the tree is reached.

如果在那里没有找到，就转移到上一级目录，依此类推，直到找到该模块或到达目录树的根结点。

For example, if the file at `'/home/ry/projects/foo.js'` called `require('bar.js')`, then node would look in the following locations, in this order:
例如，如果在文件 `'/home/ry/projects/foo.js'`中调用 `require('bar.js')`，node 将会依次查找以下位置：

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

This allows programs to localize their dependencies, so that they do not clash.

这允许程序本地化他们的依赖关系，避免发生冲突。

Optimizations to the `node_modules` Lookup Process 优化 `node_modules` 的查找过程

When there are many levels of nested dependencies, it is possible for these file trees to get fairly long. The following optimizations are thus made to the process.

如果有很多级的嵌套信赖，文件树会变得相当的长，下面是对这一过程的一些优化。

First, `/node_modules` is never appended to a folder already ending in `/node_modules`.
首先， `/node_modules` 不要添加到以 `/node_modules` 结尾的目录上。
Second, if the file calling `require()` is already inside a `node_modules` hierarchy, then the top-most `node_modules` folder is treated as the root of the search tree.
其次，如果调用 `require()`的文件已经位于一个 `node_modules` 层次中，最上级的 `node_modules` 目录将被作为搜索的根。
For example, if the file at `'/home/ry/projects/foo/node_modules/bar/node_modules/baz/quux.js'`

called `require('asdf.js')`, then node would search the following locations:
例如，如果文件 `'/home/ry/projects/foo/node_modules/bar/node_modules/baz/quux.js'` 调用 `require('asdf.js')`，node 会在下面的位置进行搜索：

- `/home/ry/projects/foo/node_modules/bar/node_modules/baz/node_modules/asdf.js`
- `/home/ry/projects/foo/node_modules/bar/node_modules/asdf.js`
- `/home/ry/projects/foo/node_modules/asdf.js`

Folders as Modules 目录作为模块

It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to that library. There are three ways in which a folder may be passed to `require()` as an argument.
很方便将程序或库组织成自包含的目录，并提供一个单独的入口指向那个库。有三种方式可以将一个子目录作为参数传递给 `require()` 。
The first is to create a `package.json` file in the root of the folder, which specifies a `main` module. An example package.json file might look like this:
第一种方法是在目录的根下创建一个名为 `package.json` 的文件，它指定了一个 `main` 模块。一个 package.jso 文件的例子如下面所示：

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

If this was in a folder at `./some-library`, then `require('./some-library')` would attempt to load `./some-library/lib/some-library.js`.
如果此文件位于`./some-library` 目录中，`require('./some-library')`将试图加载文件`./some-library/lib/some-library.js`。
This is the extent of Node's awareness of package.json files.

这是 Node 感知 package.json 文件的范围。

If there is no package.json file present in the directory, then node will attempt to load an `index.js` or `index.node` file out of that directory. For example, if there was no package.json file in the above example, then `require('./some-library')` would attempt to load:
如果在目录中没有 package.json 文件，node 将试图在该目录中加载 `index.js` 或 `index.node` 文件。例如，在上面的例子中没有 package.json 文件，`require('./some-library')`将试图加载：

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

Caching 缓存

Modules are cached after the first time they are loaded. This means (among other things) that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.
模块在第一次加载后将被缓存。这意味着（类似其他缓存）每次调用`require('foo')`如果解析到相同的文件，那么将返回同一个对象。

All Together... 总结一下...

To get the exact filename that will be loaded when `require()` is called, use the `require.resolve()` function.
可使用`require.resolve()`函数，获得调用`require()`时将加载的准确的文件名。
Putting together all of the above, here is the high-level algorithm in pseudocode of what require.resolve does:

综上所述，这里以伪代码的形式给出 require.resolve 的算法逻辑：

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

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

LOAD_AS_DIRECTORY(X)
```

```
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
2. LOAD_AS_FILE(X/index)

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

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

Loading from the `require.paths` Folders 从`require.paths`目录中加载

In node, `require.paths` is an array of strings that represent paths to be searched for modules when they are not prefixed with `'/'`, `'./'`, or `'../'`. For example, if require.paths were set to:
在 node 中，`require.paths`是一个保存模块搜索路径的字符串数组。当模块不以`'/'`，`'./'`或`'../'`为前缀时，将从此数组中的路径里进行搜索。例如，如果 require.paths 如下设置：

```
[ '/home/micheil/.node_modules',
```

```
  '/usr/local/lib/node_modules' ]
```

Then calling `require('bar/baz.js')` would search the following locations:
当调用`require('bar/baz.js')`时将搜索下列位置：

- 1: '/home/micheil/.node_modules/bar/baz.js'
- 2: '/usr/local/lib/node_modules/bar/baz.js'

The `require.paths` array can be mutated at run time to alter this behavior.
可以在运行时改变`require.paths`数组的内容，以改变路径搜索行为。
It is set initially from the `NODE_PATH` environment variable, which is a colon-delimited list of absolute paths. In the previous example, the `NODE_PATH` environment variable might have been set to:
此数组使用`NODE_PATH`环境变量进行初始化，此环境变量是冒号分割的路径列表。在之前的例子中，`NODE_PATH`环境变量被设置为如下内容：

```
/home/micheil/.node_modules:/usr/local/lib/node_modules
```

Loading from the `require.paths` locations is only performed if the module could not be found using the `node_modules` algorithm above. Global modules are lower priority than bundled dependencies.
只有当使用上面介绍的`node_modules`算法无法找到模块时，才会从`require.paths`地址里进行加载。全局模块比绑定抵赖的模块优先级低。

Note: Please Avoid Modifying `require.paths` ** 注意：** 请不要修改`requires.paths`

For compatibility reasons, `require.paths` is still given first priority in the module lookup process. However, it may disappear in a future release.
由于兼容性的原因，`require.paths`仍然在模块查询过程中处于第一优先级。然而，在未来发布的版本中这个问题将被解决。
While it seemed like a good idea at the time, and enabled a lot of useful experimentation, in practice a mutable `require.paths` list is often a troublesome source of confusion and headaches.
虽然在当时看起来这是个好注意，可以支持很多有用的实验手段。但在实践中发现，修改`require.paths`列表往往是造成混乱和麻烦的源头。

Setting `require.paths` to some other value does nothing. 将`require.paths`设为其他值不会产生任何作用

This does not do what one might expect:

下述做法不会其他你期望的任何效果：

```
require.paths = [ '/usr/lib/node' ];
```

All that does is lose the reference to the *actual* node module lookup paths, and create a new reference to some other thing that isn't used for anything.

这么做将会丢失对*真正的*模块搜索路径列表对象的引用，同时指向了一个新创建的对象，而这个对象将不会其任何作用。

Putting relative paths in `require.paths` is... weird. 不建议在`require.paths`中发入相对路径

If you do this:

如果你这样做：

```
require.paths.push('./lib');
```

then it does *not* add the full resolved path to where `./lib` is on the filesystem. Instead, it literally adds `'./lib'`, meaning that if you do `require('y.js')` in `/a/b/x.js`, then it'll look in `/a/b/lib/y.js`. If you then did `require('y.js')` in `/l/m/n/o/p.js`, then it'd look in `/l/m/n/o/lib/y.js`.
这样只会添加`'./lib'`字符串到搜索路径列表，而*不会*解析`./lib`在文件系统中的绝对路径。这意味着如果你在`/a/b/x.js`中调用`require('y.js')`，将找到`/a/b/lib/y.js`。而如果你在`/l/m/n/o/p.js`中调用`require('y.js')`，将找到`/l/m/n/o/lib/y.js`。
In practice, people have used this as an ad hoc way to bundle dependencies, but this technique is brittle.

在实践中，有用户使用这种特别的方式来实现绑定依赖，但这种方式是很脆弱的。

There is (by regrettable design), only one `require.paths` array used by all modules.
由于设计的失误，所有模块都共享同一个 `require.paths` 数组。
As a result, if one node program comes to rely on this behavior, it may permanently and subtly alter the behavior of all other node programs in the same process. As the application stack grows, we tend to assemble functionality, and it is a problem with those parts interact in ways that are difficult to predict.

造成的结果是，如果一个 node 程序依赖于这种行为，它将永久的并且隐蔽的改变处在同个进程内的所有其他 node 程序的行为。一旦应用程序变大，我们往往进行功能集成，各部分功能以不可预料的方式互相影响将成为问题。

Addenda: Package Manager Tips 附录：包管理技巧

The semantics of Node's `require()` function were designed to be general enough to support a number of sane directory structures. Package manager programs such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from Node modules without modification.
Node 的 `require()`函数的语义被设计的足够通用化，以支持各种常规目录结构。包管理程序如 `dpkg`，`rpm` 和 `npm` 将不用修改就能够从 Node 模块构建本地包。
Below we give a suggested directory structure that could work:

接下来我们将给你一个可行的目录结构建议：

Let's say that we wanted to have the folder at `/usr/lib/node/<some-package>/<some-version>` hold the contents of a specific version of a package.
假设我们希望将一个包的指定版本放在`/usr/lib/node/<some-package>/<some-version>`目录中。
Packages can depend on one another. In order to install package `foo`, you may have to install a specific version of package `bar`. The `bar` package may itself have dependencies, and in some cases, these dependencies may even collide or form cycles.
包可以依赖于其他包。为了安装包 `foo`，可能需要安装包 `bar` 的一个指定版本。包 `bar` 也可能有依赖关系，在一些情况下依赖关系可能发生冲突或形成循环。
Since Node looks up the `realpath` of any modules it loads (that is, resolves symlinks), and then looks for their dependencies in the

`node_modules` folders as described above, this situation is very simple to resolve with the following architecture:
因为 Node 会查找它所加载的模块的真实路径（也就是说会解析符号链接），然后按照上文描述的方式在 `node_modules` 目录中寻找依赖关系，所以可以使用如下的目录结构解决这个问题：

- `/usr/lib/node/foo/1.2.3/` - Contents of the `foo` package, version 1.2.3. `/usr/lib/node/foo/1.2.3/` - 包 `foo` 的 1.2.3 版本内容。
- `/usr/lib/node/bar/4.3.2/` - Contents of the `bar` package that `foo` depends on. `/usr/lib/node/bar/4.3.2/` - 包 `foo` 依赖的包 `bar` 的内容。
- `/usr/lib/node/foo/1.2.3/node_modules/bar` - Symbolic link to `/usr/lib/node/bar/4.3.2/`.
  `/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/*` - Symbolic links to the packages that `bar` depends on.
  `/usr/lib/node/bar/4.3.2/node_modules/*` - 指向包 `bar` 所依赖的包的符号链接。

Thus, even if a cycle is encountered, or if there are dependency conflicts, every module will be able to get a version of its dependency that it can use.

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

When the code in the `foo` package does `require('bar')`, it will get the version that is symlinked into `/usr/lib/node/foo/1.2.3/node_modules/bar`. Then, when the code in the `bar` package calls `require('quux')`, it'll get the version that is symlinked into `/usr/lib/node/bar/4.3.2/node_modules/quux`.
当包 `foo` 中的代码调用 `require('bar')`，将获得符号链接 `/usr/lib/node/foo/1.2.3/node_modules/bar` 指向的版本。同样，当包 `bar` 中的代码调用 `require('queue')`，降火的符号链接 `/usr/lib/node/bar/4.3.2/node_modules/quux` 指向的版本。
Furthermore, to make the module lookup process even more optimal, rather than putting packages directly in `/usr/lib/node`, we could put them in `/usr/lib/node_modules/<name>/<version>`. Then node will not bother looking for missing dependencies in `/usr/node_modules` or `/node_modules`.
为了进一步优化模块搜索过程，不要将包直接放在`/usr/lib/node` 目录中，而是将它们放在`/usr/lib/node_modules/<name>/<version>`目录中。这样在依赖的包找不到的情况下，就不会一直寻找到`/usr/node_modules` 目录或 `/node_modules` 目录中了。

In order to make modules available to the node REPL, it might be useful to also add the `/usr/lib/node_modules` folder to the `$NODE_PATH` environment variable. Since the module lookups using `node_modules` folders are all relative, and based on the real path of the files making the calls to `require()`, the packages themselves can be anywhere.
为了使模块在 node REPL 中可用，你可能需要将`/usr/lib/node_modules`目录加入到`$NODE_PATH`环境变量中。由于在`node_modules`目录中搜索模块使用的是相对路径，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。 ## Addons 扩展插件
Addons are dynamically linked shared objects. They can provide glue to C and C++ libraries. The API (at the moment) is rather complex, involving knowledge of several libraries:

扩展插件（Addons）是动态链接的共享对象，这些对象提供了使用 C/C++类库的能力。由于涉及了多个类库导致了这类 API 目前比较繁杂，主要包括下述几个主要类库：

- V8 JavaScript, a C++ library. Used for interfacing with JavaScript: creating objects, calling functions, etc. Documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node source tree).
  V8 JavaScript，C++类库，作为 JavaScript 的接口类，主要用于创建对象、调用方法等功能。大部分功能在头文件`v8.h`（在 node 文件夹下的路径为`deps/v8/include/v8.h`）中有详细文档。
- libev, C event loop library. Anytime one needs to wait for a file descriptor to become readable, wait for a timer, or wait for a signal to received one will need to interface with libev. That is, if you perform any I/O, libev will need to be used. Node uses the `EV_DEFAULT` event loop. Documentation can be found here.
  libev，基于 C 的事件循环库。当需要等待文件描述（file descriptor）为可读时，等待定时器时，或者等待接受信号时，会需要调用 libev 库。也可以说，任何 IO 操作都需要调用 libev 库。Node 使用`EV_DEFAULT`事件循环机制。在这里可以查阅相关文档。
- libeio, C thread pool library. Used to execute blocking POSIX system calls asynchronously. Mostly wrappers already exist for such calls, in `src/file.cc` so you will probably not need to use it. If you do need it, look at the header file `deps/libeio/eio.h`.
  libeio，基于 C 的线程池库，用于以异步方式执行阻塞式 POSIX 系统调用。因为大部分这类调用都在`src/file.cc`中被封装了，你一般不需要直接使用 libeio。如果必须使用该类库时，可查看其头文件`deps/libeio/eio.h`。
- Internal Node libraries. Most importantly is the `node::ObjectWrap` class which you will likely want to derive from.

内部 Node 库。在该库中，最重要的类是我们可能用于进行派生的 `node::ObjectWrap` 基类。

- Others. Look in `deps/` for what else is available.
  其他的一些类库同样可以在 `deps/` 中找到。

Node statically compiles all its dependencies into the executable. When compiling your module, you don't need to worry about linking to any of these libraries.

Node 已将所有依赖关系静态地编译成可执行文件，因此我们在编译自己的组件时不需要担心和这些类库的链接问题。

To get started let's make a small Addon which does the following except in C++:

让我们着手编写一个 Addon 的小例子，来达到如下模块同样的效果：

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

To get started we create a file `hello.cc`:
首先我们需要创建一个 `hello.cc` 文件：

```
#include <v8.h>

using namespace v8;

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

This source code needs to be built into `hello.node`, the binary Addon. To do this we create a file called `wscript` which is python code and looks like this:
这些源码会编译成一个二进制的 Addon 文件 `hello.node`。为此我们用 python 编写如下的名为 `wscript` 的文件：

```
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'
```

Running `node-waf configure build` will create a file `build/default/hello.node` which is our Addon.
运行 `node-waf configure build`，我们就创建了一个 Addon 实例 `build/default/hello.node`。
`node-waf` is just WAF, the python-based build system. `node-waf` is provided for the ease of users.
`node-waf` 就是 WAF,，一种基于 python 的编译系统，而 `node-waf` 更加易于使用。
All Node addons must export a function called `init` with this signature:
另外，在 Node 中任何的 Addon 必须使用输出一个如下声明的 `init` 函数：

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

For the moment, that is all the documentation on addons. Please see http://github.com/ry/node_postgres for a real example.

目前关于 addon 的所有文档就是这些。另外，在 http://github.com/ry/node_postgres 中还提供了一个 Addon 的实例。 ## process 进程

The `process` object is a global object and can be accessed from anywhere.
`process` 对象是一个全局对象，可以在任何地方访问它。
It is an instance of `EventEmitter`.
它是 `EventEmitter` 事件触发器类型的一个实例。

Event: 'exit' 事件：'exit'

`function () {}`
Emitted when the process is about to exit. This is a good hook to perform constant time checks of the module's state (like for unit tests). The main event loop will no longer be run after the 'exit' callback finishes, so timers may not be scheduled.

当进程对象要退出时会触发此方法，这是检查模块状态（比如单元测试）的好时机。当'exit'被调用完成后主事件循环将终止，所以计时器将不会按计划执行。

Example of listening for `exit`:
监听 `exit` 行为的示例：

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

Event: 'uncaughtException' 事件：'uncaughtException'

`function (err) { }`
Emitted when an exception bubbles all the way back to the event loop. If a listener is added for this exception, the default action (which is to print a stack trace and exit) will not occur.

当一个异常信息一路冒出到事件循环时，该方法被触发。如果该异常有一个监听器，那么默认的行为（即打印一个堆栈轨迹并退出）将不会发生。

Example of listening for `uncaughtException`:

监听 `uncaughtException` 事件的示例：

```
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.');
```

Note that `uncaughtException` is a very crude mechanism for exception handling. Using try / catch in your program will give you more control over your program's flow. Especially for server programs that are designed to stay running forever, `uncaughtException` can be a useful safety mechanism.
注意：就异常处理来说，`uncaughtException` 是一个很粗糙的机制。在程序中使用 try/catch 可以更好好控制程序流程。而在服务器编程中，因为要持续运行，`uncaughtException` 还是一个很有用的安全机制。

Signal Events 信号事件

`function () {}`
Emitted when the processes receives a signal. See sigaction(2) for a list of standard POSIX signal names such as SIGINT, SIGUSR1, etc.

该事件会在进程接收到一个信号时被触发。可参见 sigaction(2)中的标准 POSIX 信号名称列表，比如 SIGINT，SIGUSR1 等等。

Example of listening for `SIGINT`:
监听 `SIGINT` 的示例：

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

An easy way to send the `SIGINT` signal is with `Control-C` in most terminal programs.
在大多数终端程序中，一个简易发送 `SIGINT` 信号的方法是在使用 `Control-C` 命令操作。

process.stdout

A `Writable Stream` to `stdout`.
一个指向标准输出 `stdout` 的 `Writable Stream` 可写流。
Example: the definition of `console.log`
示例：`console.log` 的定义。

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

process.stderr

A writable stream to stderr. Writes on this stream are blocking.

一个指向错误的可写流，在这个流上的写操作是阻塞式的。

process.stdin

A `Readable Stream` for stdin. The stdin stream is paused by default, so one must call `process.stdin.resume()` to read from it.
一个到标准输入的可读流 `Readable Stream`。默认情况下标准输入流是暂停的，要从中读取内容需要调用方法 `process.stdin.resume()`。
Example of opening standard input and listening for both events:

示例：打开标准输入与监听两个事件：

```
process.stdin.resume();
process.stdin.setEncoding('utf8');
```

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

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

process.argv

An array containing the command line arguments. The first element will be 'node', the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments.

一个包含命令行参数的数组。第一个元素是'node'，第二个元素是 JavaScript 文件的文件名。接下来的元素则是附加的命令行参数。

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

This will generate:

这产生如下的信息：

```
$ 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

This is the absolute pathname of the executable that started the process.

这是一个启动该进程的可执行程序的绝对路径名。

Example:

例如：

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

process.chdir(directory)

Changes the current working directory of the process or throws an exception if that fails.

改变进程的当前工作目录，如果操作失败则抛出异常。

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

process.cwd()

Returns the current working directory of the process.

返回进程的当前工作目录。

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

process.env

An object containing the user environment. See environ(7).

一个包括用户环境的对象。可参见 environ(7)。

process.exit(code=0)

Ends the process with the specified `code`. If omitted, exit uses the 'success' code `0`.
用指定的 `code` 代码结束进程。如果不指定，退出时将使用'success'（成功）代码 `0`。
To exit with a 'failure' code:

以'failure'（失败）代码退出的示例：

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

The shell that executed node should see the exit code as 1.

执行 node 的 shell 会把退出代码视为 1。

process.getgid()

Gets the group identity of the process. (See getgid(2).) This is the numerical group id, not the group name.

获取进程的群组标识（详见 getgid(2)）。这是一个数字的群组 ID，不是群组名称。

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

process.setgid(id)

Sets the group identity of the process. (See setgid(2).) This accepts either a numerical ID or a groupname string. If a groupname is specified, this method blocks while resolving it to a numerical ID.

设置进程的群组标识（详见 getgid(2)）。参数可以是一个数字 ID 或者群组名字符串。如果指定了一个群组名，这个方法会阻塞等待将群组名解析为数字 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()

Gets the user identity of the process. (See getuid(2).) This is the numerical userid, not the username.

获取进程的用户 ID（详见 getgid(2)）。这是一个数字用户 ID，不是用户名。

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

process.setuid(id)

Sets the user identity of the process. (See setuid(2).) This accepts either a numerical ID or a username string. If a username is specified, this method blocks while resolving it to a numerical ID.

设置进程的用户 ID（详见 getgid(2)）。参数可以使一个数字 ID 或者用户名字符串。如果指定了一个用户名，那么该方法会阻塞等待将用户名解析为数字 ID。

```
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

A compiled-in property that exposes `NODE_VERSION`.
一个编译内置的属性，用于显示 `NODE_VERSION`（Node 版本）。

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

process.installPrefix

A compiled-in property that exposes `NODE_PREFIX`.
一个编译内置的属性，用于显示 `NODE_PREFIX`（Node 安装路径前缀）。

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

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

Send a signal to a process. `pid` is the process id and `signal` is the string describing the signal to send. Signal names are strings like

'SIGINT' or 'SIGUSR1'. If omitted, the signal will be 'SIGTERM'. See kill(2) for more information.
发送一个信号到进程。`pid` 是进程的 ID，参数 `signal` 是欲发送信号的字符串描述。信号名称是像'SIGINT'或者'SIGUSR1'这样的字符串。如果参数 `signal` 忽略，则信号为'SIGTERM'。详见 kill(2)。
Note that just because the name of this function is `process.kill`, it is really just a signal sender, like the `kill` system call. The signal sent may do something other than kill the target process.
注意该函数名为 `process.kill`，实际上也就像 `kill` 系统调用一样仅仅是一个信号发送器。发送的信号可能是要终止目标进程，也可能是实现其他不同的目的。
Example of sending a signal to yourself:

一个给自己发送信号的示例：

```
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

The PID of the process.

进程的 PID。

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

process.title

Getter/setter to set what is displayed in 'ps'.

获取或设置在'ps'命令中显示的进程的标题。

process.platform

What platform you're running on. `'linux2'`, `'darwin'`, etc.
运行 Node 的平台信息，如'linux2'，'darwin'`等等。

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

process.memoryUsage()

Returns an object describing the memory usage of the Node process.

返回一个描述 Node 进程内存使用情况的对象。

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

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

This will generate:

这会生成如下信息：

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

`heapTotal` and `heapUsed` refer to V8's memory usage.
`heapTotal` 与 `heapUsed` 指 V8 的内存使用情况。

process.nextTick(callback)

On the next loop around the event loop call this callback. This is *not* a simple alias to `setTimeout(fn, 0)`, it's much more efficient.
在事件循环的下一次循环中调用 callback 回调函数。这*不是* `setTimeout(fn, 0)`的一个别名，因为它有效率多了。

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

process.umask([mask])

Sets or reads the process's file mode creation mask. Child processes inherit the mask from the parent process. Returns the old mask if `mask` argument is given, otherwise returns the current mask.
设置或者读取进程的文件模式创建掩码。子进程从父进程中继承这个掩码。如果设定了参数 `mask` 那么返回旧的掩码，否则返回当前的掩码。

```
var oldmask, newmask = 0644;

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

util 工具模块

These functions are in the module `'util'`. Use `require('util')` to access them.
下列函数属于`'util'`（工具）模块，可使用 `require('util')`访问它们。

util.debug(string)

A synchronous output function. Will block the process and output `string` immediately to `stderr`.
这是一个同步输出函数，将 `string` 参数的内容实时输出到 `stderr` 标准错误。调用此函数时将阻塞当前进程直到输出完成。

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

util.log(string)

Output with timestamp on `stdout`.
将 `string` 参数的内容加上当前时间戳，输出到 `stdout` 标准输出。

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

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

Return a string representation of `object`, which is useful for debugging.
以字符串形式返回 `object` 对象的结构信息，这对程序调试非常有帮助。
If `showHidden` is `true`, then the object's non-enumerable properties will be shown too.
如果 `showHidden` 参数设置为 `true`，则此对象的不可枚举属性也会被显示。
If `depth` is provided, it tells `inspect` how many times to recurse while formatting the object. This is useful for inspecting large complicated objects.
可使用 `depth` 参数指定 `inspect` 函数在格式化对象信息时的递归次数。这对分析复杂对象的内部结构非常有帮助。
The default is to only recurse twice. To make it recurse indefinitely, pass in `null` for `depth`.
默认情况下递归两次，如果想要无限递归可将 `depth` 参数设为 `null`。
Example of inspecting all properties of the `util` object:
显示 `util` 对象所有属性的例子如下：

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

console.log(util.inspect(util, true, null));
```

util.pump(readableStream, writableStream, [callback])

Experimental

实验性的

Read the data from `readableStream` and send it to the `writableStream`. When `writableStream.write(data)` returns `false` `readableStream` will be paused until the `drain` event occurs on the `writableStream`. `callback` gets an error as its only argument and is called when `writableStream` is closed or when an error occurs.
从 `readableStream` 参数所指定的可读流中读取数据，并将其写入到 `writableStream` 参数所指定的可写流中。当 `writeableStream.write(data)` 函数调用返回为 `false` 时，`readableStream` 流将被暂停，直到在 `writableStream` 流上发生 `drain` 事件。当 `writableStream` 流被关闭或发生一个错误时，`callback` 回调函数被调用。此回调函数只接受一个参数用以指明所发生的错误。

util.inherits(constructor, superConstructor)

Inherit the prototype methods from one constructor into another. The prototype of `constructor` will be set to a new object created from `superConstructor`.
将一个构造函数的原型方法继承到另一个构造函数中。`constructor` 构造函数的原型将被设置为使用 `superConstructor` 构造函数所创建的一个新对象。
As an additional convenience, `superConstructor` will be accessible through the `constructor.super_` property.
此方法带来的额外的好处是，可以通过 `constructor.super_` 属性来访问 `superConstructor` 构造函数。

```
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!"
```

Events 事件模块

Many objects in Node emit events: a `net.Server` emits an event each time a peer connects to it, a `fs.readStream` emits an event when the file is opened. All objects which emit events are instances of `events.EventEmitter`. You can access this module by doing: `require("events");`
Node 引擎中很多对象都会触发事件：例如 `net.Server` 会在每一次有客户端连接到它时触发事件，又如 `fs.readStream` 会在文件打开时触发事件。所有能够触发事件的对象都是 `events.EventEmitter` 的实例。你可以通过 `require("events");`访问这个模块。
Typically, event names are represented by a camel-cased string, however, there aren't any strict restrictions on that, as any string will be accepted.

通常情况下，事件名称采用驼峰式写法，不过目前并没有对事件名称作任何的限制，也就是说任何的字符串都可以被接受。

Functions can then be attached to objects, to be executed when an event is emitted. These functions are called *listeners*.

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*。

events.EventEmitter

To access the EventEmitter class, `require('events').EventEmitter`.
通过调用 `require('events').EventEmitter`，我们可以使用事件触发器类。
When an `EventEmitter` instance experiences an error, the typical action is to emit an `'error'` event. Error events are treated as a special case in node. If there is no listener for it, then the default action is to print a stack trace and exit the program.

当 `EventEmitter` 事件触发器遇到错误时，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。
All EventEmitters emit the event `'newListener'` when new listeners are added.
当新的事件监听器被添加时，所有的事件触发器都将触发名为`'newListener'`的事件。

emitter.addListener(event, listener)

emitter.on(event, listener)

Adds a listener to the end of the listeners array for the specified event.

将一个监听器添加到指定事件的监听器数组的末尾。

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

emitter.once(event, listener)

Adds a **one time** listener for the event. The listener is invoked only the first time the event is fired, after which it is removed.

为事件添加**一次性**的监听器。该监听器在事件第一次触发时执行，过后将被移除。

```
server.once('connection', function (stream) {
  console.log('Ah, we have our first user!');
});
```

emitter.removeListener(event, listener)

Remove a listener from the listener array for the specified event. **Caution**: changes array indices in the listener array behind the listener.

将监听器从指定事件的监听器数组中移除出去。 **小心**：此操作将改变监听器数组的下标。

```
var callback = function(stream) {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

emitter.removeAllListeners(event)

Removes all listeners from the listener array for the specified event.

将指定事件的所有监听器从监听器数组中移除。

emitter.setMaxListeners(n)

By default EventEmitters will print a warning if more than 10 listeners are added to it. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.

默认情况下当事件触发器注册了超过 10 个以上的监听器时系统会打印警告信息，这个默认配置将有助于你查找内存泄露问题。很显然并不是所有的事件触发器都需要进行 10 个监听器的限制，此函数允许你手动设置该数量值，如果值为 0 意味值没有限制。

emitter.listeners(event)

Returns an array of listeners for the specified event. This array can be manipulated, e.g. to remove listeners.

返回指定事件的监听器数组对象，你可以对该数组进行操作，比如说删除监听器等。

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

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

Execute each of the listeners in order with the supplied arguments.

以提供的参数作为监听器函数的参数，顺序执行监听器列表中的每个监听器函数。

Event: 'newListener' 事件：'newListener'

`function (event, listener) { }`
This event is emitted any time someone adds a new listener.

任何时候只要新的监听器被添加时该事件就会触发。

Buffers 缓冲器

Pure Javascript is Unicode friendly but not nice to binary data. When dealing with TCP streams or the file system, it's necessary to handle octet streams. Node has several strategies for manipulating, creating, and consuming octet streams.

纯 Javascript 语言是 Unicode 友好性的，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 提供了一些列机制，用于操作、创建、以及消耗（consuming）字节流。

Raw data is stored in instances of the `Buffer` class. A `Buffer` is similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. A `Buffer` cannot be resized.

在实例化的 `Buffer` 类中存储了原始数据。`Buffer` 类似于一个整数数组，但 `Buffer` 对应了在 V8 堆（the V8 heap）外的原始存储空间分配。一旦创建了 `Buffer` 实例，则无法改变其大小。
The `Buffer` object is global.
另外，`Buffer` 是一个全局对象。
Converting between Buffers and JavaScript string objects requires an explicit encoding method. Here are the different string encodings;

在缓冲器（Buffers）和 JavaScript 间进行字符串的转换需要调用特定的编码方法。如下列举了不同的编码方法：

- `'ascii'` - for 7 bit ASCII data only. This encoding method is very fast, and will strip the high bit if set.
  `'ascii'` - 仅对应 7 位的 ASCII 数据。虽然这种编码方式非常迅速，并且如果设置了最高位，则会将其移去。
- `'utf8'` - Multi byte encoded Unicode characters. Many web pages and other document formats use UTF-8.
  `'utf8'` - 对应多字节编码 Unicode 字符。大量网页和其他文件格式使用这类编码方式。
- `'ucs2'` - 2-bytes, little endian encoded Unicode characters. It can encode only BMP(Basic Multilingual Plane, U+0000 - U+FFFF).
  `'ucs2'` - 2 字节的，低字节序编码 Unicode 字符。只能编码 BMP（第零平面，U+0000 - U+FFFF）字符。
- `'base64'` - Base64 string encoding.
  `'base64'` - Base64 字符串编码.
- `'binary'` - A way of encoding raw binary data into strings by using only the first 8 bits of each character. This encoding method is depreciated and should be avoided in favor of `Buffer` objects where possible. This encoding will be removed in future versions of Node.
  `'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。而且，这个编码方式不会出现在未来版本的 Node 中。
- `'hex'` - Encode each byte as two hexidecimal characters.
  `'hex'` - 将一个字节编码为两个 16 进制字符。

new Buffer(size)

Allocates a new buffer of `size` octets.
分配给一个新创建的 buffer 实例一个大小为 `size` 字节的空间。

new Buffer(array)

Allocates a new buffer using an `array` of octets.

使用 `array` 的空间创建一个 buffer 实例。

new Buffer(str, encoding='utf8')

Allocates a new buffer containing the given `str`.
创建一个包含给定 `str` 的 buffer 实例。

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

Writes `string` to the buffer at `offset` using the given encoding. Returns number of octets written. If `buffer` did not contain enough space to fit the entire string, it will write a partial amount of the string. In the case of `'utf8'` encoding, the method will not write partial characters.
通过给定的编码方式把 `string` 写入到 buffer 的 `offset`（偏移地址）中，并且返回写入的字节数。如果当前的 `buffer` 没有足够存储空间，字符串会部分地保存在 `buffer` 中，而不是整串字符。需要注意的是，如果使用`'utf8'`进行编码，该方法不会对零散的字符进行编写。
Example: write a utf8 string into a buffer, then print it

例如：将一串 utf8 格式的字符串写入 Buffer，然后输出：

```
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)

Decodes and returns a string from buffer data encoded with `encoding` beginning at `start` and ending at `end`.
对缓冲器中的以 `encoding` 方式编码的，以 `start` 标识符开始，以 `end` 标识符结尾的缓冲数据进行解码，并输出字符串。
See `buffer.write()` example, above.
参见上文的 `buffer.write()`例子。

buffer[index]

Get and set the octet at `index`. The values refer to individual bytes, so the legal range is between `0x00` and `0xFF` hex or `0` and `255`.
获取或者设置位于 `index` 字节的值。由于返回值为单个的字节，因此其范围应该在 `0x00` 到 `0xFF`（16 进制）或者 `0` and `255`（10 进制）之间
Example: copy an ASCII string into a buffer, one byte at a time:

例如：通过每次仅输入一个字符的方式将整串 ASCII 字符录入 Buffer 中：

```
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
```

Buffer.isBuffer(obj)

Tests if `obj` is a `Buffer`.
验证 `obj` 的类别是否为 `Buffer` 类。

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

Gives the actual byte length of a string. This is not the same as `String.prototype.length` since that returns the number of *characters* in a string.
返回字符串长度的实际值。与 `String.prototype.length` 的区别之处在于该方法返回的是字符串中 *characters* 的个数。
Example:

例如：

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

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

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

buffer.length

The size of the buffer in bytes. Note that this is not necessarily the size of the contents. `length` refers to the amount of memory allocated for the buffer object. It does not change when the contents of the buffer are changed.
返回 Buffer 占用的字节数。需要注意的是，`length` 并非其内容占的大小，而是指分配给 Buffer 实例的存储空间的大小，因此该值不会随 Buffer 内容的变化而变化。

```
buf = new Buffer(1234);

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

// 1234
// 1234
```

buffer.copy(targetBuffer, targetStart=0, sourceStart=0, sourceEnd=buffer.length)

Does a memcpy() between buffers.

在两个 Buffer 之间进行 memcpy() 操作。

Example: build two Buffers, then copy `buf1` from byte 16 through byte 19 into `buf2`, starting at the 8th byte in `buf2`.

例如：创建 2 个 Buffer 实例，然后将 `buf1` 中第 16 字节到第 19 字节间的信息复制到 `buf2` 中，并使在 `buf2` 中新的字符串首字符位于第 8 字节：

```
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!!!!!!!!!!!!!
```

buffer.slice(start, end=buffer.length)

Returns a new buffer which references the same memory as the old, but offset and cropped by the `start` and `end` indexes.
返回一个和原 Buffer 引用相同存储空间的新 Buffer，但是新 Buffer 中的偏移地址截取了原 Buffer 偏移地址中自 `start` 到 `end` 的部分。
Modifying the new buffer slice will modify memory in the original buffer!

特别注意：通过修改新的 Buffer 切片（slice）中的内容同样会修改存储在原 Buffer 中的信息！

Example: build a Buffer with the ASCII alphabet, take a slice, then modify one byte from the original Buffer.

例如：建立一个 ASCII 码型的字母表，再建立一个切片，并在原 Buffer 中修改一个字节：

```
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
```

Streams 流

A stream is an abstract interface implemented by various objects in Node. For example a request to an HTTP server is a stream, as is stdout. Streams are readable, writable, or both. All streams are instances of `EventEmitter`.
在 Node 中，Stream（流）是一个由不同对象实现的抽象接口。例如请求 HTTP 服务器的 request 是一个流，类似于 stdout（标准输出）。流可以是可读的，可写的，或者既可读又可写。所有流都是 `EventEmitter` 的实例。

Readable Stream 可读流

A `Readable Stream` has the following methods, members, and events.
一个`可读流`具有下述的方法、成员、及事件。

Event: 'data' 事件：'data'

`function (data) { }`
The `'data'` event emits either a `Buffer` (by default) or a string if `setEncoding()` was used.
`'data'`事件的回调函数参数默认情况下是一个 `Buffer` 对象。如果使用了 `setEncoding()` 则参数为一个字符串。

Event: 'end' 事件：'end'

`function () { }`
Emitted when the stream has received an EOF (FIN in TCP terminology). Indicates that no more `'data'` events will happen. If the stream is also writable, it may be possible to continue writing.

当流中接收到 EOF（TCP 中为 FIN）时此事件被触发，表示流的读取已经结束，不会再发生任何`'data'`事件。如果流同时也是可写的，那它还可以继续写入。

Event: 'error' 事件：'error'

`function (exception) { }`
Emitted if there was an error receiving data.

接收数据的过程中发生任何错误时，此事件被触发。

Event: 'close' 事件：'close'

`function () { }`
Emitted when the underlying file descriptor has been closed. Not all streams will emit this. (For example, an incoming HTTP request will not emit `'close'`.)
当底层的文件描述符被关闭时触发此事件，并不是所有流都会触发这个事件。（例如，一个连接进入的 HTTP request 流就不会触发`'close'`事件。）

Event: 'fd' 事件：'fd'

`function (fd) { }`
Emitted when a file descriptor is received on the stream. Only UNIX streams support this functionality; all others will simply never emit this event.

当在流中接收到一个文件描述符时触发此事件。只有 UNIX 流支持这个功能，其他类型的流均不会触发此事件。

stream.readable

A boolean that is `true` by default, but turns `false` after an `'error'` occurred, the stream came to an `'end'`, or `destroy()` was called.
这是一个布尔值，默认值为`true`。当`'error'`事件或`'end'`事件发生后，或者`destroy()`被调用后，这个属性将变为`false`。

stream.setEncoding(encoding)

Makes the data event emit a string instead of a `Buffer`. `encoding` can be `'utf8'`, `'ascii'`, or `'base64'`.
调用此方法会影响`'data'`事件的回调函数参数形式，默认为`Buffer`对象，调用此方法后为字符串。`encoding`参数可以是`'utf8'`、`'ascii'`、或`'base64'`。

stream.pause()

Pauses the incoming `'data'` events.
暂停`'data'`事件的触发。

stream.resume()

Resumes the incoming `'data'` events after a `pause()`.
恢复被`pause()`调用暂停的`'data'`事件触发。

stream.destroy()

Closes the underlying file descriptor. Stream will not emit any more events.

关闭底层的文件描述符。流上将不会再触发任何事件。

stream.destroySoon()

After the write queue is drained, close the file descriptor.

在写队列清空后（所有写操作完成后），关闭文件描述符。

stream.pipe(destination, [options])

This is a `Stream.prototype` method available on all `Stream`s.
这是`Stream.prototype`（Stream 原型对象）的一个方法，对所有`Stream`对象有效。
Connects this read stream to `destination` WriteStream. Incoming data on this stream gets written to `destination`. The destination and source streams are kept in sync by pausing and resuming as necessary.
用于将这个可读流和`destination`目标可写流连接起来，传入这个流中的数据将会写入到`destination`流中。通过在必要时暂停和恢复流，来源流和目的流得以保持同步。
Emulating the Unix `cat` command:
模拟 Unix 系统的`cat`命令：

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

By default `end()` is called on the destination when the source stream emits `end`, so that `destination` is no longer writable. Pass `{ end: false }` as `options` to keep the destination stream open.

默认情况下，当来源流的 `end` 事件触发时目的流的 `end()` 方法会被调用，此时 `destination` 目的流将不再可写入。要在这种情况下为了保持目的流仍然可写入，可将 `options` 参数设为 `{ end: false }`。
This keeps `process.stdout` open so that "Goodbye" can be written at the end.
这使 `process.stdout` 保持打开状态，因此"Goodbye"可以在 end 事件发生后被写入。

```
process.stdin.resume();

process.stdin.pipe(process.stdout, { end: false });

process.stdin.on("end", function() {
  process.stdout.write("Goodbye\n");
});
```

NOTE: If the source stream does not support `pause()` and `resume()`, this function adds simple definitions which simply emit `'pause'` and `'resume'` events on the source stream.
注意：如果来源流不支持 `pause()` 和 `resume()` 方法，此函数将在来源流对象上增加这两个方法的简单定义，内容为触发 `'pause'` 和 `'resume'` 事件。

Writable Stream 可写流

A `Writable Stream` has the following methods, members, and events.
一个可写流具有下列方法、成员、和事件。

Event: 'drain' 事件：'drain'

`function () { }`
Emitted after a `write()` method was called that returned `false` to indicate that it is safe to write again.
发生在 `write()` 方法被调用并返回 `false` 之后。此事件被触发说明内核缓冲区已空，再次写入是安全的。

Event: 'error' 事件：'error'

`function (exception) { }`
Emitted on error with the exception `exception`.
发生错误时被触发，回调函数接收一个异常参数 `exception`。

Event: 'close' 事件：'close'

`function () { }`
Emitted when the underlying file descriptor has been closed.

底层文件描述符被关闭时被触发。

Event: 'pipe' 事件：'pipe'

`function (src) { }`
Emitted when the stream is passed to a readable stream's pipe method.

当此可写流作为参数传给一个可读流的 pipe 方法时被触发。

stream.writable

A boolean that is `true` by default, but turns `false` after an `'error'` occurred or `end()` / `destroy()` was called.
一个布尔值，默认值为 `true`。在`'error'`事件被触发之后，或 `end()` / `destroy()`方法被调用后此属性被设为 `false`。

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

Writes `string` with the given `encoding` to the stream. Returns `true` if the string has been flushed to the kernel buffer. Returns `false` to indicate that the kernel buffer is full, and the data will be sent out in the future. The `'drain'` event will indicate when the kernel buffer is empty again. The `encoding` defaults to `'utf8'`.
使用指定编码 `encoding` 将字符串 `string` 写入到流中。如果字符串被成功写入内核缓冲区，此方法返回 `true`。如果内核缓冲区已满，此方法返回 `false`，数据将在以后被送出。当内核缓冲区再次被清空后'drain'事件将被触发。 encoding 参数默认为'utf8'`。
If the optional `fd` parameter is specified, it is interpreted as an integral file descriptor to be sent over the stream. This is only supported for UNIX streams, and is silently ignored otherwise. When writing a file descriptor in this manner, closing the descriptor before the stream drains risks sending an invalid (closed) FD.
如果指定了可选参数 `fd`，它将被作为一个文件描述符通过流传送。此功能仅被 Unix 流所支持，对于其他流此操作将被忽略而没有任何提示。当使用此方法传送一个文件描述符时，如果在流没有清空前关闭此文件描述符，将造成传送一个无效（已关闭）FD 的风险。

stream.write(buffer)

Same as the above except with a raw buffer.

除了用一个 Buffer 对象替代字符串之外，其他同上。

stream.end()

Terminates the stream with EOF or FIN.

使用 EOF 或 FIN 结束一个流的输出。

stream.end(string, encoding)

Sends `string` with the given `encoding` and terminates the stream with EOF or FIN. This is useful to reduce the number of packets sent.
以指定的字符编码 `encoding` 传送一个字符串 `string`，然后使用 EOF 或 FIN 结束流的输出。这对降低数据包传输量有所帮助。

stream.end(buffer)

Same as above but with a `buffer`.
除了用一个 `buffer` 对象替代字符串之外，其他同上。

stream.destroy()

Closes the underlying file descriptor. Stream will not emit any more events.

关闭底层文件描述符。在此流上将不会再触发任何事件。

Crypto 加密模块

Use `require('crypto')` to access this module.
使用 `require('crypto')`调用加密模块。
The crypto module requires OpenSSL to be available on the underlying platform. It offers a way of encapsulating secure credentials to be used as part of a secure HTTPS net or http connection.

加密模块需要底层系统提供 OpenSSL 的支持。它提供了一种安全凭证的封装方式，可以用于 HTTPS 安全网络以及普通 HTTP 连接。

It also offers a set of wrappers for OpenSSL's hash, hmac, cipher, decipher, sign and verify methods.

该模块还提供了一套针对 OpenSSL 的 hash（哈希），hmac（密钥哈希），cipher（编码），decipher（解码），sign（签名）以及 verify（验证）等方法的封装。

crypto.createCredentials(details)

Creates a credentials object, with the optional details being a dictionary with keys:

创建一个凭证对象，可选参数 details 为一个带键值的字典：

- `key` : a string holding the PEM encoded private key
  `key`：为字符串型，PEM 编码的私钥。
- `cert` : a string holding the PEM encoded certificate
  `cert`：为字符串型，PEM 编码的认证证书。
- `ca` : either a string or list of strings of PEM encoded CA certificates to trust.
  `ca`：字符串形式的 PEM 编码可信 CA 证书，或证书列表。

If no 'ca' details are given, then node.js will use the default publicly trusted list of CAs as given in http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt.

如果没有给出'ca'的详细内容，那么 node.js 将会使用默认的公开受信任列表，该表位于 http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt。

crypto.createHash(algorithm)

Creates and returns a hash object, a cryptographic hash with the given algorithm which can be used to generate hash digests.

创建并返回一个 hash 对象，它是一个指定算法的加密 hash，用于生成 hash 摘要。

`algorithm` is dependent on the available algorithms supported by the version of OpenSSL on the platform. Examples are `'sha1'`, `'md5'`, `'sha256'`, `'sha512'`, etc. On recent releases, `openssl list-message-digest-algorithms` will display the available digest algorithms.
参数 `algorithm` 可选择系统上安装的 OpenSSL 版本所支持的算法。例如：`'sha1'`, `'md5'`, `'sha256'`, `'sha512'`等。在近期发行的版本中，`openssl list-message-digest-algorithms` 会显示这些可用的摘要算法。

hash.update(data)

Updates the hash content with the given `data`. This can be called many times with new data as it is streamed.

更新 hash 的内容为指定的 `data`。当使用流数据时可能会多次调用该方法。

hash.digest(encoding='binary')

Calculates the digest of all of the passed data to be hashed. The `encoding` can be `'hex'`, `'binary'` or `'base64'`.
计算所有传入数据的 hash 摘要。参数 `encoding`（编码方式）可以为`'hex'`, `'binary'` 或者`'base64'`。

crypto.createHmac(algorithm, key)

Creates and returns a hmac object, a cryptographic hmac with the given algorithm and key.

创建并返回一个 hmac 对象，它是一个指定算法和密钥的加密 hmac。

`algorithm` is dependent on the available algorithms supported by OpenSSL - see createHash above. `key` is the hmac key to be used.
参数 `algorithm` 可选择 OpenSSL 支持的算法 - 参见上文的 createHash。参数 `key` 为 hmac 所使用的密钥。

hmac.update(data)

Update the hmac content with the given `data`. This can be called many times with new data as it is streamed.
更新 hmac 的内容为指定的 `data`。当使用流数据时可能会多次调用该方法。

hmac.digest(encoding='binary')

Calculates the digest of all of the passed data to the hmac. The `encoding` can be `'hex'`, `'binary'` or `'base64'`.
计算所有传入数据的 hmac 摘要。参数 `encoding`（编码方式）可以为`'hex'`, `'binary'` 或者`'base64'`。

crypto.createCipher(algorithm, key)

Creates and returns a cipher object, with the given algorithm and key.

使用指定的算法和密钥创建并返回一个 cipher 对象。

`algorithm` is dependent on OpenSSL, examples are `'aes192'`, etc. On recent releases, `openssl list-cipher-algorithms` will display the available cipher algorithms.

参数 `algorithm` 可选择 OpenSSL 支持的算法，例如`'aes192'`等。在最近的发行版中，`openssl list-cipher-algorithms`会显示可用的加密的算法。

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

Updates the cipher with `data`, the encoding of which is given in `input_encoding` and can be `'utf8'`, `'ascii'` or `'binary'`. The `output_encoding` specifies the output format of the enciphered data, and can be `'binary'`, `'base64'` or `'hex'`.
使用参数 `data` 更新要加密的内容，其编码方式由参数 `input_encoding` 指定，可以为 `'utf8'`, `'ascii'`或者`'binary'`。参数`output_encoding`指定了已加密内容的输出编码方式，可以为 `'binary'`, `'base64'`或`'hex'`。
Returns the enciphered contents, and can be called many times with new data as it is streamed.

返回已加密的内容，当使用流数据时可能会多次调用该方法。

cipher.final(output_encoding='binary')

Returns any remaining enciphered contents, with `output_encoding` being one of: `'binary'`, `'ascii'` or `'utf8'`.
返回所有剩余的加密内容，`output_encoding`输出编码为`'binary'`, `'ascii'`或`'utf8'`其中之一。

crypto.createDecipher(algorithm, key)

Creates and returns a decipher object, with the given algorithm and key. This is the mirror of the cipher object above.

使用给定的算法和密钥创建并返回一个解密对象。该对象为上述加密对象的反向运算。

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

Updates the decipher with `data`, which is encoded in `'binary'`, `'base64'` or `'hex'`. The `output_decoding` specifies in what format to return the deciphered plaintext: `'binary'`, `'ascii'` or `'utf8'`.

使用参数 `data` 更新要解密的内容，其编码方式为`'binary'`，`'base64'`或`'hex'`。参数 `output_encoding` 指定了已解密的明文内容的输出编码方式，可以为 `'binary'`，`'ascii'`或`'utf8'`。

decipher.final(output_encoding='binary')

Returns any remaining plaintext which is deciphered, with `output_encoding' being one of:` ’binary’`,` ’ascii’ `or` ’utf8’`.`
返回全部剩余的已解密的明文，其 `output_encoding' 为`’binary’`,` ’ascii’ 或’utf8’`其中之一。

crypto.createSign(algorithm)

Creates and returns a signing object, with the given algorithm. On recent OpenSSL releases, `openssl list-public-key-algorithms` will display the available signing algorithms. Examples are `'RSA-SHA256'`.
使用给定的算法创建并返回一个签名器对象。在现有的 OpenSSL 发行版中，`openssl list-public-key-algorithms` 会显示可用的签名算法，例如：`'RSA-SHA256'`。

signer.update(data)

Updates the signer object with data. This can be called many times with new data as it is streamed.

使用 data 参数更新签名器对象。当使用流数据时可能会多次调用该方法。

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

Calculates the signature on all the updated data passed through the signer. `private_key` is a string containing the PEM encoded private key for signing.
对所有传入签名器的数据计算其签名。`private_key` 为字符串，它包含了 PEM 编码的用于签名的私钥。
Returns the signature in `output_format` which can be `'binary'`, `'hex'` or `'base64'`.
返回签名，其 `output_format` 输出可以为`'binary'`, `'hex'` 或者`'base64'`。

crypto.createVerify(algorithm)

Creates and returns a verification object, with the given algorithm. This is the mirror of the signing object above.

使用给定算法创建并返回一个验证器对象。它是上述签名器对象的反向运算。

verifier.update(data)

Updates the verifier object with data. This can be called many times with new data as it is streamed.

使用 data 参数更新验证器对象。当使用流数据时可能会多次调用该方法。

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

Verifies the signed data by using the `cert` which is a string containing the PEM encoded public key, and `signature`, which is the previously calculates signature for the data, in the `signature_format` which can be `'binary'`, `'hex'` or `'base64'`.
使用参数 `cert` 和 `signature` 验证已签名的数据，`cert` 为经过 PEM 编码的公钥字符串，`signature` 为之前已计算的数据的签名，`signature_format` 可以为`'binary'`，`'hex'` 或者`'base64'`。
Returns true or false depending on the validity of the signature for the data and public key.

根据对数据和公钥进行签名有效性验证的结果，返回 true 或者 false。 ## TLS (SSL) TLS (SSL)模块

Use `require('tls')` to access this module.
使用 `require('tls')`访问此模块。
The `tls` module uses OpenSSL to provide Transport Layer Security and/or Secure Socket Layer: encrypted stream communication.
`tls` 模块使用 OpenSSL 提供 Transport Layer Security（传输层安全协议）和 / 或 Secure Socket Layer（安全套接层协议）：加密的通信流。
TLS/SSL is a public/private key infrastructure. Each client and each server must have a private key. A private key is created like this

TLS/SSL 基于公钥/私钥的非对称加密体系，每一个客户端与服务器都需要拥有一个私有密钥。私有密钥可用如下方式生成：

```
openssl genrsa -out ryans-key.pem 1024
```

All severs and some clients need to have a certificate. Certificates are public keys signed by a Certificate Authority or self-signed. The

first step to getting a certificate is to create a "Certificate Signing Request" (CSR) file. This is done with:

所有服务器和一部分客户端需要拥有一份数字证书。数字证书是由某个 CA（数字证书认证机构）使用其公钥签名授予的，或者也可以用户自签名。要获得一份数字证书，首先需要生成一个 CSR（证书签名请求）文件。方法如下：

```
openssl req -new -key ryans-key.pem -out ryans-csr.pem
```

To create a self-signed certificate with the CSR, do this:

要使用 CSR 文件生成一个自签名的数字证书，方法如下：

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

Alternatively you can send the CSR to a Certificate Authority for signing.

你也可以将 CSR 文件发给一家 CA 以获得签名。

(TODO: docs on creating a CA, for now interested users should just look at `test/fixtures/keys/Makefile` in the Node source code)
（关于如何创建 CA 的文档有待补充。感兴趣的用户可以直接浏览 Node 源代码中的 `test/fixtures/keys/Makefile` 文件）

s = tls.connect(port, [host], [options], callback)

Creates a new client connection to the given `port` and `host`. (If `host` defaults to `localhost`.) `options` should be an object which specifies
建立一个到指定端口 `port` 和主机 `host` 的新的客户端连接。（`host` 参数的默认值为 `localhost`。）`options` 是一个包含以下内容的对象：

- `key`: A string or `Buffer` containing the private key of the server in PEM format. (Required)
  `key`：包含服务器私钥的字符串或 `Buffer` 对象。密钥的格式为 PEM。（必选）

- `cert`: A string or `Buffer` containing the certificate key of the server in PEM format.
  `cert`：包含服务器数字证书密钥的字符串或 `Buffer` 对象。密钥的格式为 PEM。
- `ca`: An array of strings or `Buffer`s of trusted certificates. If this is omitted several well known "root" CAs will be used, like VeriSign. These are used to authorize connections.
  `ca`：包含可信任数字证书字符串或 `Buffer` 对象的数组。如果忽略此属性，则会使用几个常见的"根"CA 的数字证书，如 VeriSign。这些数字证书将被用来对连接进行验证。

`tls.connect()` returns a cleartext `CryptoStream` object.
`tls.connect()`返回一个明文的 `CryptoStream` 对象。
After the TLS/SSL handshake the `callback` is called. The `callback` will be called no matter if the server's certificate was authorized or not. It is up to the user to test `s.authorized` to see if the server certificate was signed by one of the specified CAs. If `s.authorized === false` then the error can be found in `s.authorizationError`.
TLS/SSL 连接握手之后 `callback` 回调函数会被调用。无论服务器的数字证书是否通过验证，`callback` 函数都会被调用。用户应该检查 `s.authorized` 以确定服务器数字证书是否通过了验证（被某个可信任的 CA 签名）。当 `s.authorized === false` 时可以从 `s.authorizationError` 中获得具体的错误。

tls.Server

This class is a subclass of `net.Server` and has the same methods on it. Instead of accepting just raw TCP connections, this accepts encrypted connections using TLS or SSL.
这是 `net.Server` 的子类，拥有和 `net.Server` 完全一样的方法。区别在于这个类使用 TLS 或 SSL 建立加密的连接，而非仅仅接受原始的 TCP 连接。
Here is a simple example echo server:

下面是一个简单的回声服务器的例子：

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

var options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem')
```

```
};

tls.createServer(options, function (s) {
  s.write("welcome!\n");
  s.pipe(s);
}).listen(8000);
```

You can test this server by connecting to it with `openssl s_client`:
你可以使用 `openssl s_client` 连接到这个服务器进行测试：

```
openssl s_client -connect 127.0.0.1:8000
```

tls.createServer(options, secureConnectionListener)

This is a constructor for the `tls.Server` class. The options object has these possibilities:
这是 `tls.Server` 类的构造函数。参数 options 对象可以包含下列内容：

- `key`: A string or `Buffer` containing the private key of the server in PEM format. (Required)
  `key`：包含服务器私钥的字符串或 `Buffer` 对象。密钥的格式为 PEM。（必选）
- `cert`: A string or `Buffer` containing the certificate key of the server in PEM format. (Required)
  `cert`：包含服务器数字证书密钥的字符串或 `Buffer` 对象。密钥的格式为 PEM。（必选）
- `ca`: An array of strings or `Buffer`s of trusted certificates. If this is omitted several well known "root" CAs will be used, like VeriSign. These are used to authorize connections.
  `ca`：包含可信任数字证书字符串或 `Buffer` 对象的数组。如果忽略此属性，则会使用几个常见的"根"CA 的数字证书，如 VeriSign。这些证书将被用来对连接进行验证。
- `requestCert`: If `true` the server will request a certificate from clients that connect and attempt to verify that certificate. Default: `false`.
  `requestCert`：如果设为 `true` 则服务器会向建立连接的客户端要求一个数字证书，并且试图去验证这份数字证书。默认为 `false`。
- `rejectUnauthorized`: If `true` the server will reject any connection which is not authorized with the list of supplied CAs.

This option only has an effect if `requestCert` is `true`. Default: `false`.
`rejectUnauthorized`：如果设为 `true` 则服务器将拒绝任何没有通过 CA 验证的连接。此选项仅在 `requestCert` 设为 `true` 时有效。默认为 `false`。

Event: 'secureConnection' 事件：'secureConnection'

`function (cleartextStream) {}`
This event is emitted after a new connection has been successfully handshaked. The argument is a duplex instance of `stream.Stream`. It has all the common stream methods and events.
当一个新的连接成功完成握手过程后此事件被触发。参数是一个可读可写的 `stream.Stream` 实例对象，此对象具有 Stream（流）对象所有公共的方法和事件。
`cleartextStream.authorized` is a boolean value which indicates if the client has verified by one of the supplied certificate authorities for the server. If `cleartextStream.authorized` is false, then `cleartextStream.authorizationError` is set to describe how authorization failed. Implied but worth mentioning: depending on the settings of the TLS server, you unauthorized connections may be accepted.
`cleartextStream.authorized` 是一个布尔值，用以表明客户端是否通过了服务器所指定的可信任 CA 的验证。如果 `cleartextStream.authorized` 值为 false，则可以从 `cleartextStream.authorizationError` 中获得验证失败的原因。这意味着：未经验证的连接是有可能被接受的，这依赖于 TLS 服务器的具体设置。

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

Begin accepting connections on the specified `port` and `host`. If the `host` is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`).
开始在指定的端口 `port` 和主机名 `host` 上接受连接。如果没有设置 `host` 参数，服务器将接受到达本机所有 IPv4 地址（`INADDR_ANY`）的连接。
This function is asynchronous. The last parameter `callback` will be called when the server has been bound.
此函数是异步的。最后一个参数 `callback` 所指定的回调函数会在服务器绑定完成后被调用。
See `net.Server` for more information.
更多信息参见 `net.Server`。

server.close()

Stops the server from accepting new connections. This function is asynchronous, the server is finally closed when the server emits a `'close'` event.
关闭服务器，停止接受新的连接请求。此函数是异步的，当服务器触发一个`'close'`事件时才真正被关闭。

server.maxConnections

Set this property to reject connections when the server's connection count gets high.

服务器最大连接数量。服务器会拒绝超过此数量限制的连接，以防止同时建立的连接数过多。

server.connections

The number of concurrent connections on the server.

服务器并发连接数量。

File System 文件系统模块

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.
文件的 I/O 是由标准 POSIX 函数封装而成。需要使用 `require('fs')`访问这个模块。所有的方法都提供了异步和同步两种方式。
The asynchronous form always take a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.
异步形式下，方法的最后一个参数需要传入一个执行完成时的回调函数。传给回调函数的参数取决于具体的异步方法，但第一个参数总是保留给异常对象。如果操作成功，那么该异常对象就变为 `null`或者 `undefined`。
Here is an example of the asynchronous version:

这里是一个异步调用的例子：

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

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

Here is the synchronous version:

这里是进行相同操作的同步调用的例子：

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

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

With the asynchronous methods there is no guaranteed ordering. So the following is prone to error:

由于异步方法调用无法保证执行的顺序，所以下面的代码容易导致出现错误。

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

It could be that `fs.stat` is executed before `fs.rename`. The correct way to do this is to chain the callbacks.
这样做有可能导致 `fs.stat` 在 `fs.rename` 之前执行，正确的做法是链式调用回调函数。

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

In busy processes, the programmer is *strongly encouraged* to use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete--halting all connections.

当需要频繁操作时，*强烈建议*使用异步方法。同步方式在其完成之前将会阻塞当前的整个进程，即搁置所有连接。

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

Asynchronous rename(2). No arguments other than a possible exception are given to the completion callback.

异步调用 rename(2)，重命名某个文件，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.renameSync(path1, path2)

Synchronous rename(2).

同步调用重命名 rename(2)，重命名某个文件。

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

Asynchronous ftruncate(2). No arguments other than a possible exception are given to the completion callback.

异步调用 ftruncate(2)，截断某个文件，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.truncateSync(fd, len)

Synchronous ftruncate(2).

同步调用重命名 ftruncate(2)，截断某个文件 s。

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

Asynchronous chmod(2). No arguments other than a possible exception are given to the completion callback.

异步调用 chmod(2)，修改文件权限，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.chmodSync(path, mode)

Synchronous chmod(2).

同步调用 chmod(2)，修改文件权限。

fs.stat(path, [callback])

Asynchronous stat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. It looks like this:
异步调用 stat(2)，读取文件元信息，回调函数将返回两个参数`(err, stats)`，其中 `stats` 是 `fs.Stats` 的一个对象，如下所示：

```
{ 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' }
```

See the `fs.Stats` section below for more information.
有关详细信息，请参阅下面的 `fs.Stats` 部分

fs.lstat(path, [callback])

Asynchronous lstat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object. lstat() is identical to stat(), except that if path is a symbolic link, then the link itself is stat-ed, not the file that it refers to.
异步形式调用 lstat(2)，回调函数返回两个参数`(err, stats)`，其中 `stats` 是 `fs.Stats` 的一个对象，lstat()和 stat()类似，区别在于当 path 是一个符号链接时，它指向该链接的属性，而不是所指向文件的属性.

fs.fstat(fd, [callback])

Asynchronous fstat(2). The callback gets two arguments `(err, stats)` where `stats` is a `fs.Stats` object.
异步形式调用 fstat(2)，回调函数返回两个参数`(err, stats)`，其中 `stats` 是 `fs.Stats` 的一个对象。

fs.statSync(path)

Synchronous stat(2). Returns an instance of `fs.Stats`.
同步形式调用 stat(2)，返回 `fs.Stats` 的一个实例。

fs.lstatSync(path)

Synchronous lstat(2). Returns an instance of `fs.Stats`.
同步形式调用 lstat(2)，返回 `fs.Stats` 的一个实例。

fs.fstatSync(fd)

Synchronous fstat(2). Returns an instance of `fs.Stats`.
同步形式调用 fstatSync(2)，返回 `fs.Stats` 的一个实例。

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

Asynchronous link(2). No arguments other than a possible exception are given to the completion callback.

异步调用 link(2)，创建符号连接，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.linkSync(srcpath, dstpath)

Synchronous link(2).

同步调用 link(2)。

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

Asynchronous symlink(2). No arguments other than a possible exception are given to the completion callback.

异步调用 symlink(2)，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.symlinkSync(linkdata, path)

Synchronous symlink(2).

同步调用 symlink(2)。

fs.readlink(path, [callback])

Asynchronous readlink(2). The callback gets two arguments `(err, resolvedPath)`.
异步调用 readlink，回调函数返回两个参数`(err,resolvedPath)`，`resolvedPath` 为解析后的文件路径。

fs.readlinkSync(path)

Synchronous readlink(2). Returns the resolved path.

同步调用 readlink(2)，返回解析后的文件路径。

fs.realpath(path, [callback])

Asynchronous realpath(2). The callback gets two arguments `(err, resolvedPath)`.
异步调用 realpath(2)，回调函数返回两个参数`(err,resolvedPath)`，resolvedPath 为解析后的文件路径。

fs.realpathSync(path)

Synchronous realpath(2). Returns the resolved path.

同步调用 realpath(2)，返回解析后的文件路径。

fs.unlink(path, [callback])

Asynchronous unlink(2). No arguments other than a possible exception are given to the completion callback.

异步调用 unlink(2)，删除链接或者文件，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.unlinkSync(path)

Synchronous unlink(2).

同步调用 unlink(2)。

fs.rmdir(path, [callback])

Asynchronous rmdir(2). No arguments other than a possible exception are given to the completion callback.

异步调用 rmdir(2)，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.rmdirSync(path)

Synchronous rmdir(2).

同步调用 rmdir(2)。

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

Asynchronous mkdir(2). No arguments other than a possible exception are given to the completion callback.

异步调用 mkdir(2)，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.mkdirSync(path, mode)

Synchronous mkdir(2).

同步调用 mkdir(2)。

fs.readdir(path, [callback])

Asynchronous readdir(3). Reads the contents of a directory. The callback gets two arguments `(err, files)` where `files` is an array of the names of the files in the directory excluding `'.'` and `'..'`.
异步调用 readdir(3)，读取目录中的内容。回调函数接受两个参数`(err, files)`，其中 `files` 参数是保存了目录中所有文件名的数组（`'.'`和`'..'`除外）。

fs.readdirSync(path)

Synchronous readdir(3). Returns an array of filenames excluding `'.'` and `'..'`.
同步调用 readdir(3)。返回目录中文件名数组(`'.'`与`'..'`除外)。

fs.close(fd, [callback])

Asynchronous close(2). No arguments other than a possible exception are given to the completion callback.

异步同步调用 close(2)，关闭文件，除非回调函数执行过程出现了异常，否则不会传递任何参数。

fs.closeSync(fd)

Synchronous close(2).

同步调用 close(2)。

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

Asynchronous file open. See open(2). Flags can be 'r', 'r+', 'w', 'w+', 'a', or 'a+'. The callback gets two arguments `(err, fd)`.
异步开启文件，详阅 open(2)。标签可为'r', 'r+', 'w', 'w+', 'a', 或 'a+'。回调函数接受两个参数`(err, fd)`。

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

Synchronous open(2).

同步调用 open(2)。

fs.utimes(path, atime, mtime, callback)

fs.utimesSync(path, atime, mtime)

Change file timestamps.

更改文件时间戳。

fs.futimes(path, atime, mtime, callback)

fs.futimesSync(path, atime, mtime)

Change file timestamps with the difference that if filename refers to a symbolic link, then the link is not dereferenced.

另一种更改文件时间戳的方式。区别在于如果文件名指向一个符号链接，则改变此符号链接的时间戳，而不改变所引用文件的时间戳。

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

Write `buffer` to the file specified by `fd`.
将 `buffer` 缓冲器内容写入 `fd` 文件描述符。
`offset` and `length` determine the part of the buffer to be written.
`offset` 和 `length` 决定了将缓冲器中的哪部分写入文件。
`position` refers to the offset from the beginning of the file where this data should be written. If `position` is `null`, the data will be written at the current position. See pwrite(2).
`position` 指明将数据写入文件从头部算起的偏移位置，若 `position` 为 `null`，数据将从当前位置开始写入，详阅 pwrite(2)。
The callback will be given two arguments `(err, written)` where `written` specifies how many *bytes* were written.
回调函数接受两个参数`(err, written)`，其中 `written` 标识有多少*字节*的数据已经写入。

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

Synchronous version of buffer-based `fs.write()`. Returns the number of bytes written.
基于缓冲器的 `fs.write()`的同步版本，返回写入数据的字节数。

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

Synchronous version of string-based `fs.write()`. Returns the number of bytes written.
基于字符串的 `fs.write()`的同步版本，返回写入数据的字节数。

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

Read data from the file specified by `fd`.
从 `fd` 文件描述符中读取数据。

`buffer` is the buffer that the data will be written to.
`buffer` 为写入数据的缓冲器。
`offset` is offset within the buffer where writing will start.
`offset` 为写入到缓冲器的偏移地址。
`length` is an integer specifying the number of bytes to read.
`length` 指明了欲读取的数据字节数。
`position` is an integer specifying where to begin reading from in the file. If `position` is `null`, data will be read from the current file position.
`position` 为一个整形变量，标识从哪个位置开始读取文件，如果 `position` 参数为 `null`，数据将从文件当前位置开始读取。
The callback is given the two arguments, `(err, bytesRead)`.
回调函数接受两个参数，`(err, bytesRead)`。

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

Synchronous version of buffer-based `fs.read`. Returns the number of `bytesRead`.
基于缓冲器的 `fs.read` 的同步版本，返回读取到的 `bytesRead` 字节数。

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

Synchronous version of string-based `fs.read`. Returns the number of `bytesRead`.
基于字符串的 `fs.read` 的同步版本，返回已经读入的数据的字节数。

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

Asynchronously reads the entire contents of a file. Example:

异步读取一个文件的所有内容，例子如下：

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

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.
回调函数将传入两个参数`(err, data)`，其中 `data` 为文件内容。

If no encoding is specified, then the raw buffer is returned.

如果没有设置编码，那么将返回原始内容格式的缓冲器。

fs.readFileSync(filename, [encoding])

Synchronous version of `fs.readFile`. Returns the contents of the `filename`.
同步调用 `fs.readFile` 的版本，返回指定文件 `filename` 的文件内容。
If `encoding` is specified then this function returns a string. Otherwise it returns a buffer.
如果设置了 `encoding` 参数，将返回一个字符串。否则返回一个缓冲器。

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

Asynchronously writes data to a file. `data` can be a string or a buffer.
异步写入数据到某个文件中，`data` 可以是字符串或者缓冲器。
Example:

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')

The synchronous version of `fs.writeFile`.
同步调用 `fs.writeFile` 的方式。

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

Watch for changes on `filename`. The callback `listener` will be called each time the file is accessed.
监听指定文件 `filename` 的变化，回调函数 `listener` 将在每次该文件被访问时被调用。

The second argument is optional. The `options` if provided should be an object containing two members a boolean, `persistent`, and `interval`, a polling value in milliseconds. The default is `{ persistent: true, interval: 0 }`.
第二个参数是可选项，如果指定了 `options` 参数，它应该是一个包含如下内容的对象：名为 `persistent` 的布尔值，和名为 `interval` 单位为毫秒的轮询时间间隔，默认值为`{ persistent: true, interval: 0 }`。
The `listener` gets two arguments the current stat object and the previous stat object:
`listener` 监听器将获得两个参数，分别标识当前的状态对象和改变前的状态对象。

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

These stat objects are instances of `fs.Stat`.
这些状态对象为 `fs.Stat` 的实例。
If you want to be notified when the file was modified, not just accessed you need to compare `curr.mtime` and `prev.mtime.
如果你想在文件被修改而不是被访问时得到通知，你还需要比较 `curr.mtime` 和 `prev.mtime` 的值。

fs.unwatchFile(filename)

Stop watching for changes on `filename`.
停止监听文件 `filename` 的变化。

fs.Stats

Objects returned from `fs.stat()` and `fs.lstat()` are of this type.
`fs.stat()`和 `fs.lstat()`方法返回的对象为此类型。

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

fs.ReadStream

`ReadStream` is a `Readable Stream`.
`ReadStream` 是一个 `Readable Stream` 可读流。

fs.createReadStream(path, [options])

Returns a new ReadStream object (See `Readable Stream`).
返回一个新的可读流对象（参见 `Readable Stream`）。
`options` is an object with the following defaults:
`options` 是包含如下默认值的对象：

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0666,
  bufferSize: 64 * 1024
}
```

`options` can include `start` and `end` values to read a range of bytes from the file instead of the entire file. Both `start` and `end` are inclusive and start at 0. When used, both the limits must be specified always.
如果不想读取文件的全部内容，可以在 `options` 参数中设置 `start` 和 `end` 属性值以读取文件中指定范围的内容。`start` 和 `end` 包含在范围中（闭集合），取值从 0 开始。这两个参数需要同时设置。
An example to read the last 10 bytes of a file which is 100 bytes long:

一个例子演示了从一个长度为 100 字节的文件中读取最后 10 个字节：

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

fs.WriteStream

`WriteStream` is a `Writable Stream`.
`WriteStream` 为可写流。

Event: 'open' 事件：'open'

`function (fd) { }`

`fd` is the file descriptor used by the WriteStream.
`fd`是可写流所使用的文件描述符。

fs.createWriteStream(path, [options])

Returns a new WriteStream object (See `Writable Stream`).
返回一个新的可写流对象（参见`Writable Stream`）。
`options` is an object with the following defaults:
`options`参数是包含如下默认值的对象：

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

Path 路径模块

This module contains utilities for dealing with file paths. Use `require('path')` to use it. It provides the following methods:
该模块包括了一些处理文件路径的功能，可以通过`require('path')`方法来使用它。该模块提供了如下的方法：

path.normalize(p)

Normalize a string path, taking care of `'..'` and `'.'` parts.
该方法用于标准化一个字符型的路径，请注意`'..'` 与 `'.'` 的使用。
When multiple slashes are found, they're replaces by a single one; when the path contains a trailing slash, it is preserved. On windows backslashes are used.

当发现有多个斜杠（/）时，系统会将他们替换为一个斜杠；如果路径末尾中包含有一个斜杠，那么系统会保留这个斜杠。在 Windows 中，上述路径中的斜杠（/）要换成反斜杠（\）。

Example:

示例：

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

```
'/foo/bar/baz/asdf'
```

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

Join all arguments together and normalize the resulting path.

该方法用于合并方法中的各参数并得到一个标准化合并的路径字符串。

Example:

示例：

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

path.resolve([from ...], to)

Resolves `to` to an absolute path.
将 `to` 参数解析为绝对路径。
If `to` isn't already absolute `from` arguments are prepended in right to left order, until an absolute path is found. If after using all `from` paths still no absolute path is found, the current working directory is used as well. The resulting path is normalized, and trailing slashes are removed unless the path gets resolved to the root directory.
如果参数 `to` 当前不是绝对的，系统会将 `from` 参数按从右到左的顺序依次前缀到 `to` 上，直到在 `from` 中找到一个绝对路径时停止。如果遍历所有 `from` 中的路径后，系统依然没有找到一个绝对路径，那么当前工作目录也会作为参数使用。最终得到的路径是标准化的字符串，并且标准化时系统会自动删除路径末尾的斜杠，但是如果获取的路径是解析到根目录的，那么系统将保留路径末尾的斜杠。
Another way to think of it is as a sequence of `cd` commands in a shell.
你也可以将这个方法理解为 Shell 中的一组 `cd` 命令。

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

Is similar to:

就类似于：

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

The difference is that the different paths don't need to exist and may also be files.

该方法与 `cd` 命令的区别在于该方法中不同的路径不一定存在，而且这些路径也可能是文件。
Examples:

示例：

```
path.resolve('/foo/bar', './baz')
// returns
'/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/')
// returns
'/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
// if currently in /home/myself/node, it returns
'/home/myself/node/wwwroot/static_files/gif/image.gif'
```

path.dirname(p)

Return the directory name of a path. Similar to the Unix `dirname` command.
该方法返回一个路径的目录名，类似于 Unix 中的 `dirname` 命令。
Example:

示例：

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

path.basename(p, [ext])

Return the last portion of a path. Similar to the Unix `basename` command.
该方法返回一个路径中最低一级目录名，类似于 Unix 中的 `basename` 命令。
Example:

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)

Return the extension of the path. Everything after the last '.' in the last portion of the path. If there is no '.' in the last portion of the path or the only '.' is the first character, then it returns an empty string.

该方法返回路径中的文件扩展名，即路径最低一级的目录中'.'字符后的任何字符串。如果路径最低一级的目录中'没有'.' 或者只有'.'，那么该方法返回一个空字符串。

Examples:

示例：

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

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

path.exists(p, [callback])

Test whether or not the given path exists. Then, call the `callback` argument with either true or false. Example:
该方法用于测试参数 `p` 中的路径是否存在。然后以 true 或者 false 作为参数调用 `callback` 回调函数。示例：

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

path.existsSync(p)

Synchronous version of `path.exists`.
`path.exists` 的同步版本。 ## net 网络模块
The `net` module provides you with an asynchronous network wrapper. It contains methods for creating both servers and clients (called streams). You can include this module with `require("net");`
`net` 模块为你提供了一种异步网络包装器，它包含创建服务器和客户端（称为 streams）所需的方法，您可以通过调用 `require("net")` 来使用此模块。

net.createServer([options], [connectionListener])

Creates a new TCP server. The `connectionListener` argument is automatically set as a listener for the `'connection'` event.
创建一个新的 TCP 服务器，参数 `connectionListener` 被自动设置为 connection 事件的监听器。
`options` is an object with the following defaults:
`options` 参数为一个对象，默认值如下： { allowHalfOpen: false }
If `allowHalfOpen` is `true`, then the socket won't automatically send FIN packet when the other end of the socket sends a FIN packet. The socket becomes non-readable, but still writable. You should call the end() method explicitly. See `'end'` event for more information.
如果 `allowHalfOpen` 参数为 `true`，则当客户端 socket 发送 FIN 包时，服务器端 socket 不会自动发送 FIN 包。此情况下服务器端 socket 将变为不可读状态，但仍然可写。你需要明确的调用 end()方法来关闭连接。更多内容请参照`'end'` 事件。

net.createConnection(arguments...)

Construct a new socket object and opens a socket to the given location. When the socket is established the `'connect'` event will be emitted.
创建一个新的 socket 对象，并建立到指定地址的 socket 连接。当 socket 建立后，`'connect'`事件将被触发。
The arguments for this method change the type of connection:

不同的参数决定了连接的类型：

- `net.createConnection(port, [host])`
  Creates a TCP connection to `port` on `host`. If `host` is omitted, `localhost` will be assumed.
  创建一个到主机 `host` 的 `port` 端口的 TCP 连接，如果略了 `host` 参数，默认连接到 `localhost`。
- `net.createConnection(path)`
  Creates unix socket connection to `path`
  创建连接到 `path` 路径的 unix socket。

net.Server

This class is used to create a TCP or UNIX server.

这个类用于创建一个 TCP 或 UNIX 服务器。

Here is an example of a echo server which listens for connections on port 8124:

下面的例子创建了一个在 8124 端口监听的 `echo` 服务器。

```
var net = require('net');
var server = net.createServer(function (c) {
  c.write('hello\r\n');
  c.pipe(c);
});
server.listen(8124, 'localhost');
```

Test this by using `telnet`:
使用 `telnet` 测试该服务器。

```
telnet localhost 8124
```

To listen on the socket `/tmp/echo.sock` the last line would just be changed to
如要监听 socket `/tmp/echo.sock`，最后一行代码需要修改成：

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

Use `nc` to connect to a UNIX domain socket server:
使用 `nc` 命令连接到一个 UNIX 域 socket 服务器：

```
nc -U /tmp/echo.sock
```

`net.Server` is an `EventEmitter` with the following events:
`net.Server` 是下列事件的 `EventEmitter`（事件触发器）：

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

Begin accepting connections on the specified `port` and `host`. If the `host` is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`).
开始接收特定主机 `host` 的 `port` 端口的连接，如果省略了 `host` 参数，服务器将接收任何指向 IPV4 地址的连接。
This function is asynchronous. The last parameter `callback` will be called when the server has been bound.
此函数是异步的，在服务器被绑定时，最后一个参数 `callback` 回调函数将被调用。
One issue some users run into is getting `EADDRINUSE` errors. Meaning another server is already running on the requested port. One way of handling this would be to wait a second and the try again. This can be done with
一些用户可能会遇到 `EADDRINUSE` 错误，该错误消息的意思是已经有另一个服务运行在请求的端口上，一个解决方法就是等一会再试一下，就像下面的代码这样：

```
server.on('error', function (e) {
  if (e.code == 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(function () {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

(Note: All sockets in Node are set SO_REUSEADDR already)

（注意：Node 中所有的 socket 都已经设置成 SO_REUSEADDR 端口重用模式）

server.listen(path, [callback])

Start a UNIX socket server listening for connections on the given `path`.
启动一个 UNIX socket 服务，监听指定的 `path` 路径上的连接。
This function is asynchronous. The last parameter `callback` will be called when the server has been bound.

此函数是异步的，在服务器被绑定时，最后一个参数 `callback` 回调函数将被调用。

server.listenFD(fd)

Start a server listening for connections on the given file descriptor.

启动一个服务，监听指定的文件描述符上的连接。

This file descriptor must have already had the `bind(2)` and `listen(2)` system calls invoked on it.
此文件描述符上必须已经执行了 `bind(2)` 和 `listen(2)` 系统调用。

server.close()

Stops the server from accepting new connections. This function is asynchronous, the server is finally closed when the server emits a `'close'` event.
关闭服务，停止接收新的连接。该函数是异步的，当服务发出`'close'`事件时该服务器被最终关闭。

server.address()

Returns the bound address of the server as seen by the operating system. Useful to find which port was assigned when giving getting an OS-assigned address

返回绑定到操作系统的服务器地址。如果绑定地址是由操作系统自动分配的，可用此方法查看具体的端口号。

Example:

```
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);
```

```
});
```

server.maxConnections

Set this property to reject connections when the server's connection count gets high.

设置该属性的值，以便当服务器达到最大连接数时不再接受新的连接。

server.connections

The number of concurrent connections on the server.

服务器的并发连接数。

Event: 'connection' 事件：'connection'

`function (socket) {}`
Emitted when a new connection is made. `socket` is an instance of `net.Socket`.
当一个新的连接建立时触发。`socket` 是 `net.Socket` 的一个实例。

Event: 'close'

`function () {}`
Emitted when the server closes.

当服务器关闭时触发。

---

net.Socket

This object is an abstraction of of a TCP or UNIX socket. `net.Socket` instances implement a duplex Stream interface. They can be created by the user and used as a client (with `connect()`) or they can be

created by Node and passed to the user through the `'connection'` event of a server.
这是 TCP 或 UNIX socket 的抽象对象。`net.Socket` 实例实现了一个全双工的流接口。此实例可以是由用户建立用作客户端（使用 `connect()` 方法），也可能由 Node 建立并通过服务器的 `'connection'` 事件传给用户。
`net.Socket` instances are EventEmitters with the following events:
`net.Socket` 的实例是下列事件的事件触发器:

new net.Socket([options])

Construct a new socket object.

构造一个新的 socket 对象。

`options` is an object with the following defaults:
`options` 参数是一个对象，默认值如下：

```
{ fd: null
  type: null
  allowHalfOpen: false
}
```

`fd` allows you to specify the existing file descriptor of socket. `type` specified underlying protocol. It can be `'tcp4'`, `'tcp6'`, or `'unix'`. About `allowHalfOpen`, refer to `createServer()` and `'end'` event.
`fd` 参数允许你指定一个已经存在的 socket 的文件描述符。`type` 参数用于指定底层协议，可选值包括 `'tcp4'`，`'tcp6'` 或 `'unix'`。关于 `allowHalfOpen`，可参考 `createServer()` 和 `'end'` 事件。

socket.connect(port, [host], [callback])

socket.connect(path, [callback])

Opens the connection for a given socket. If `port` and `host` are given, then the socket will be opened as a TCP socket, if `host` is omitted, `localhost` will be assumed. If a `path` is given, the socket will be opened as a unix socket to that path.
打开一下指定 socket 的连接。如果给出了 `port` 和 `host`，将作为一个 TCP socket 打开，如果省略了 `host`，将默认连接到 `localhost`。如果指定了 `path`，该 socket 将作为一个 UNIX socket 打开，并连接到 `path` 路径。

Normally this method is not needed, as `net.createConnection` opens the socket. Use this only if you are implementing a custom Socket or if a Socket is closed and you want to reuse it to connect to another server.
通常情况下该方法并不需要，使用 `net.createConnection` 就可以打开 socket。只有在你实现一个自定义的 socket，或者你想重用一个已经关闭的 socket 连接到另一个服务器。
This function is asynchronous. When the `'connect'` event is emitted the socket is established. If there is a problem connecting, the `'connect'` event will not be emitted, the `'error'` event will be emitted with the exception.
这个函数是异步函数。当发生 `'connect'`事件时 socket 被建立，如果连接遇到问题， `'connect'`事件不会被触发，而携带异常信息的`'error'` 事件将被触发。
The `callback` parameter will be added as an listener for the 'connect' event.
参数 `callback` 将作为 `connect` 事件的监听器被增加进来。

socket.bufferSize

`net.Socket` has the property that `socket.write()` always works. This is to help users get up an running quickly. The computer cannot necessarily keep up with the amount of data that is written to a socket - the network connection simply might be too slow. Node will internally queue up the data written to a socket and send it out over the wire when it is possible. (Internally it is polling on the socket's file descriptor for being writable).
`net.Socket` 有一个特性，那就是 `socket.write()`随时可用，这是为了使 程序更快的运行。计算机发送数据的速度可能无法跟上程序向 socket 写入数据的速度——考虑网络速度很慢的情况。Node 在内部维护了一个队列用于保存 写入 socket 的数据，并在网络允许的时候将队列中的数据发送出去。（内部实现是探测 socket 的文件描述符是否可写。）
The consequence of this internal buffering is that memory may grow. This property shows the number of characters currently buffered to be written. (Number of characters is approximately equal to the number of bytes to be written, but the buffer may contain strings, and the strings are lazily encoded, so the exact number of bytes is not known.)

这种内部缓冲区的机制会增加内存消耗。此属性显示当前被缓冲的待写入字符数量。（字符的数量约等于字节数，但缓冲区中可能包含字符串，而字符串是延迟编码的，因此精确的字节数不可知。）

Users who experience large or growing `bufferSize` should attempt to "throttle" the data flows in their program with `pause()` and `resume()`.
用户可以在程序中使用`pause()`和`resume()`来"截流"数据流，以控制大量或不断增长的`bufferSize`。

socket.setEncoding(encoding=null)

Sets the encoding (either `'ascii'`, `'utf8'`, or `'base64'`) for data that is received.
设置接收到的数据的编码(可以是`'ascii'`，`'utf8'`或`'base64'`) 。

socket.setSecure()

This function has been removed in v0.3. It used to upgrade the connection to SSL/TLS. See the TLS for the new API.

该函数用来将连接升级到 SSL/TLS，在 v0.3 版本已经被废弃。参考 TLS 中新的 API 说明。

socket.write(data, [encoding], [callback])

Sends data on the socket. The second parameter specifies the encoding in the case of a string--it defaults to UTF8 encoding.

向 socket 发送数据，第二个参数指定在发送字符串数据时的编码方式，默认的是 UTF8 编码。

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is again free.
在所有数据被成功的写入系统内核缓冲区时返回`true`，如果全部或部分数据进入了用户内存的队列则返回`false`。 当缓冲区再次变空时，`'drain'` 事件将被触发。
The optional `callback` parameter will be executed when the data is finally written out - this may not be immediately.
可选参数`callback` 将在数据最终被写出时执行——可能不是立即执行。

socket.write(data, [encoding], [fileDescriptor], [callback])

For UNIX sockets, it is possible to send a file descriptor through the socket. Simply add the `fileDescriptor` argument and listen for the `'fd'` event on the other end.
对于 UNIX socket，可以通过 socket 发送一个文件描述符，简单的增加参数 `fileDescriptor`，并在另一端监听`'fd'`事件。

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

Half-closes the socket. I.E., it sends a FIN packet. It is possible the server will still send some data.

发送一个 FIN 数据包，关闭 socket 半连接。服务器仍可能发送一些数据。

If `data` is specified, it is equivalent to calling `socket.write(data, encoding)` followed by `socket.end()`.
如果指定了 `data` ，等同于依次调用 `socket.write(data, encoding)`和 `socket.end()`。

socket.destroy()

Ensures that no more I/O activity happens on this socket. Only necessary in case of errors (parse error or so).

确保该 socket 上不再有活动的 I/O 操作，仅在发生错误的情况下需要（如解析错误等）。

socket.pause()

Pauses the reading of data. That is, `'data'` events will not be emitted. Useful to throttle back an upload.
暂停读取数据，`'data'`将不会被触发，便于控制上传速度。

socket.resume()

Resumes reading after a call to `pause()`.
用于在调用 `pause()`后，恢复读取数据。

socket.setTimeout(timeout, [callback])

Sets the socket to timeout after `timeout` milliseconds of inactivity on the socket. By default `net.Socket` do not have a timeout.
设置 socket 不活动时间超过 `timeout` 毫秒后进入超时状态。默认情况下 `net.Socket` 不会超时。

When an idle timeout is triggered the socket will receive a `'timeout'` event but the connection will not be severed. The user must manually `end()` or `destroy()` the socket.
当闲置超时发生时，socket 会接收到一个`'timeout'`事件，但是连接不会被断开，用户必须手动的`end()`或`destroy()`该 socket。
If `timeout` is 0, then the existing idle timeout is disabled.
如果 `timeout` 设置成 0，已经存在的闲置超时将被禁用。
The optional `callback` parameter will be added as a one time listener for the `'timeout'` event.
可选参数 `callback` 将作为一次性监听器添加到 `'timeout'` 事件。

socket.setNoDelay(noDelay=true)

Disables the Nagle algorithm. By default TCP connections use the Nagle algorithm, they buffer data before sending it off. Setting `noDelay` will immediately fire off data each time `socket.write()` is called.
禁用 Nagle 算法。默认情况下 TCP 连接使用 Nagle 算法，在发送数据之前缓存它们。设置 `noDelay` 将使每次 `socket.write()`调用都实时发送数据。

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

Enable/disable keep-alive functionality, and optionally set the initial delay before the first keepalive probe is sent on an idle socket. Set `initialDelay` (in milliseconds) to set the delay between the last data packet received and the first keepalive probe. Setting 0 for initialDelay will leave the value unchanged from the default (or previous) setting.
开启或禁用 keep-alive（连接保持）功能。可选择设置在一个闲置 socket 上第一次发送存活探测之前的延迟时间。设置 `initialDelay`（单位毫秒）以设置最后从一个数据包接收到第一个存活探测发送之间的延时。将 initialDelay 设为 0 将不改变此参数默认（或之前）的设置。

socket.remoteAddress

The string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.
以字符串形式表示的远端设备 IP 地址。例如：`'74.125.127.100'`或`'2001:4860:a005::68'`。
This member is only present in server-side connections.

此成员只存在于服务器端连接中。

Event: 'connect' 事件：'connect'

`function () { }`
Emitted when a socket connection successfully is established. See `connect()`.
当一个 socket 连接成功建立时触发，参考 `connect()`。

Event: 'data' 事件：'data'

`function (data) { }`
Emitted when data is received. The argument `data` will be a `Buffer` or `String`. Encoding of data is set by `socket.setEncoding()`. (See the section on `Readable Socket` for more information.)
当收到数据时触发，参数 `data` 将是一个缓冲区（`Buffer`）或者字符串（`String`）。数据的编码方式通过 `socket.setEncoding()`设置。（ 更多信息请参考章节 `Readable Socket`）

Event: 'end' 事件：'end'

`function () { }`
Emitted when the other end of the socket sends a FIN packet.

当 socket 的远端发送了一个 FIN 数据包时触发。

By default (`allowHalfOpen == false`) the socket will destroy its file descriptor once it has written out its pending write queue. However, by setting `allowHalfOpen == true` the socket will not automatically `end()` its side allowing the user to write arbitrary amounts of data, with the caveat that the user is required to `end()` their side now.
默认情况下（`allowHalfOpen == false`）一旦待写出队列中的内容全部被写出，socket 将自动销毁它的文件描述符。然而如果设置 `allowHalfOpen == true`，则 socket 不会自动调用 `end()`，而是允许用户继续写入任意数量的数据，这种情况下需要用户主动调用 `end()`关闭半连接。

Event: 'timeout' 事件：'timeout'

`function () { }`
Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.

当 socket 闲置超时情况下触发，它只是用来通知那个 socket 已经空闲，用户必须手动的关闭该连接。

See also: `socket.setTimeout()`
参考: `socket.setTimeout()`

Event: 'drain' 事件：'drain'

`function () { }`
Emitted when the write buffer becomes empty. Can be used to throttle uploads.

当缓冲区变空时触发，可以被用来调节上传速度。

Event: 'error' 事件：'error'

`function (exception) { }`
Emitted when an error occurs. The `'close'` event will be called directly following this event.
当有错误发生时触发， `'close'` 事件紧跟其后被调用。

Event: 'close' 事件：'close'

`function (had_error) { }`
Emitted once the socket is fully closed. The argument `had_error` is a boolean which says if the socket was closed due to a transmission error.
当连接字完全被关闭时触发，参数 `had_error` 是一个布尔型变量， 用来说明连接字是否由于一个传输错误而关闭。

---

net.isIP

net.isIP(input)

Tests if input is an IP address. Returns 0 for invalid strings, returns 4 for IP version 4 addresses, and returns 6 for IP version 6 addresses.

测试输入参数是否是一个 IP 地址，如果是一个无效的字符串，返回 0；如果是 IPv4 地址，返回 4；如果是 IPv6 地址，返回 6。

net.isIPv4(input)

Returns true if input is a version 4 IP address, otherwise returns false.

如果 input 是一个 IPv4 地址则返回 true，否则返回 false。

net.isIPv6(input)

Returns true if input is a version 6 IP address, otherwise returns false.

如果 input 是一个 IPv6 地址则返回 true，否则返回 false。

DNS DNS 模块

Use `require('dns')` to access this module.
使用 `require('dns')`来访问这个模块。
Here is an example which resolves `'www.google.com'` then reverse resolves the IP addresses which are returned.
下面这个例子首先将`'www.google.com'`解析为 IP 地址，再对返回的 IP 地址做反向解析。

```
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)

Resolves a domain (e.g. `'google.com'`) into the first found A (IPv4) or AAAA (IPv6) record.
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。
The callback has arguments `(err, address, family)`. The `address` argument is a string representation of a IP v4 or v6 address. The `family` argument is either the integer 4 or 6 and denotes the family of `address` (not necessarily the value initially passed to `lookup`).
回调函数有`(err, address, family)`这三个参数。`address` 参数是一个代表 IPv4 或 IPv6 地址的字符串。`family` 是一个表示地址版本的整数 4 或 6(并不一定和调用 `lookup` 时传入的 `family` 参数值相同)。

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

Resolves a domain (e.g. `'google.com'`) into an array of the record types specified by rrtype. Valid rrtypes are `A` (IPV4 addresses), `AAAA` (IPV6 addresses), `MX` (mail exchange records), `TXT` (text records), `SRV` (SRV records), and `PTR` (used for reverse IP lookups).
将域名(比如`'google.com'`)按照参数 rrtype 所指定类型的解析结果放到一个数组中。合法的类型为 `A`（IPV4 地址），`AAAA`（IPV6 地址），`MX`（邮件交换记录），`TXT`（文本记录），SRV（SRV 记录），和 PTR（用于反向 IP 解析）。
The callback has arguments `(err, addresses)`. The type of each item in `addresses` is determined by the record type, and described in the documentation for the corresponding lookup methods below.
回调函数接受两个参数：`(err, addresses)`。参数 `address` 中的每一项的类型根据所要求的记录类型进行判断，在下面相应的解析方法的文档里有详细的解释。
On error, `err` would be an instanceof `Error` object, where `err.errno` is one of the error codes listed below and `err.message` is a string describing the error in English.

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

dns.resolve4(domain, callback)

The same as `dns.resolve()`, but only for IPv4 queries (`A` records). `addresses` is an array of IPv4 addresses (e.g. `['74.125.79.104', '74.125.79.105', '74.125.79.106']`).
与 `dns.resolve()`类似，但是仅对 IPV4 地址进行查询（`A` 记录）。`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)

The same as `dns.resolve4()` except for IPv6 queries (an `AAAA` query).
除了这个函数是对 IPV6 地址的查询（一个 `AAAA` 查询）外与 `dns.resolve4()` 很类似。

dns.resolveMx(domain, callback)

The same as `dns.resolve()`, but only for mail exchange queries (`MX` records).
与 `dns.resolve()`很类似，但是仅做邮件交换地址查询（`MX` 类型记录）。
`addresses` is an array of MX records, each with a priority and an exchange attribute (e.g. `[{'priority': 10, 'exchange': 'mx.example.com'},...]`).
回调函数的参数 `addresses` 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)

The same as `dns.resolve()`, but only for text queries (`TXT` records). `addresses` is an array of the text records available for `domain` (e.g., `['v=spf1 ip4:0.0.0.0 ~all']`).
与 `dns.resolve()`很相似，但是仅可以进行文本查询（`TXT` 记录）。
`addressed` 是一个对于 `domain` 域有效的文本记录数组（类似`['v=spf1 ip4:0.0.0.0 ~all']`）。

dns.resolveSrv(domain, callback)

The same as `dns.resolve()`, but only for service records (`SRV` records). `addresses` is an array of the SRV records available for `domain`. Properties of SRV records are priority, weight, port, and

name (e.g., `[{'priority': 10, {'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]`).
与 `dns.resolve()`很类似，但仅是只查询服务记录（`SRV` 类型记录）。
`addresses` 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)

Reverse resolves an ip address to an array of domain names.

反向解析一个 IP 地址到一个域名数组。

The callback has arguments `(err, domains)`.
回调函数的参数为 `(err, domains)`。
If there an an error, `err` will be non-null and an instanceof the Error object.
如果发生了错误，`err` 将被置为一个非空的 Error 对象实例。
Each DNS query can return an error code.

每个 DNS 查询可以返回如下错误代码：

- `dns.TEMPFAIL`: timeout, SERVFAIL or similar.
  `dns.TEMPFAIL`: 超时，SERVFAIL 或者类似的错误。
- `dns.PROTOCOL`: got garbled reply.
  `dns.PROTOCOL`: 返回内容混乱。
- `dns.NXDOMAIN`: domain does not exists.
  `dns.NXDOMAIN`: 域名不存在。
- `dns.NODATA`: domain exists but no data of reqd type.
  `dns.NODATA`: 域名存在但是没有所请求的查询类型的数据。
- `dns.NOMEM`: out of memory while processing.
  `dns.NOMEM`: 处理过程中内存溢出。
- `dns.BADQUERY`: the query is malformed.
  `dns.BADQUERY`: 查询语句语法错误。 ## UDP / Datagram Sockets 数据报套接字模块

Datagram sockets are available through `require('dgram')`. Datagrams are most commonly handled as IP/UDP messages but they can also be used over Unix domain sockets.
要使用数据报套接字模块需要调用 `require('dgram')`。数据报通常作为 IP/UDP 消息来处理，但它也可用于 Unix 域套接字。

Event: 'message' 事件：'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.
当套接字接收到一个新的数据报的时候触发此事件。`msg`是一个`Buffer`缓冲器，`rinfo`是一个包含了发送方地址信息以及数据报字节长度的对象。

Event: 'listening' 事件：'listening'

`function () { }`
Emitted when a socket starts listening for datagrams. This happens as soon as UDP sockets are created. Unix domain sockets do not start listening until calling `bind()` on them.
当一个套接字开始监听数据报的时候触发。一旦 UDP 套接字建立后就会触发这个事件。而 Unix 域套接字直到调用了`bind()`方法才会开始监听。

Event: 'close' 事件：'close'

`function () { }`
Emitted when a socket is closed with `close()`. No new `message` events will be emitted on this socket.
当调用`close()`方法关闭一个套接字时触发此事件。此后不会再有新的`message`事件在此套接字上发生。

dgram.createSocket(type, [callback])

Creates a datagram socket of the specified types. Valid types are: `udp4`, `udp6`, and `unix_dgram`.
建立一个指定类型的数据报套接字，有效类型有：`udp4`，`udp6`，以及`unix_dgram`。
Takes an optional callback which is added as a listener for `message` events.
可选参数 callback 指定了`message`事件的监听器回调函数。

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

For Unix domain datagram sockets, the destination address is a pathname in the filesystem. An optional callback may be supplied that is invoked after the `sendto` call is completed by the OS. It is not safe to re-use `buf` until the callback is invoked. Note that unless the socket is bound to a pathname with `bind()` there is no way to receive messages on this socket.
对于 Unix 域数据报套接字而言，目标地址是文件系统中的一个路径。可选参数 callback 指定了系统调用`sendto`完成后的回调函数。在回调函数 callback 执

行前重复使用 buf 是很不安全的。要注意除非这个套接字已经使用 `bind()` 方法绑定到一个路径上，否则这个套接字无法接收到任何信息。
Example of sending a message to syslogd on OSX via Unix domain socket `/var/run/syslog`:
一个在 OSX 系统上，通过 Unix 域套接字 `/var/run/syslog` 向 syslogd 发送消息的例子：

```
var dgram = require('dgram');
var message = new Buffer("A message to log.");
var 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.");
});
```

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

For UDP sockets, the destination port and IP address must be specified. A string may be supplied for the `address` parameter, and it will be resolved with DNS. An optional callback may be specified to detect any DNS errors and when `buf` may be re-used. Note that DNS lookups will delay the time that a send takes place, at least until the next tick. The only way to know for sure that a send has taken place is to use the callback.
对于 UDP 套接字来说，目标端口和 IP 地址是必须要指定的。可以用字符串来指定 `address` 参数，这个参数通过 DNS 进行解析。可选参数 callback 指定一个回调函数，用于检测 DNS 解析错误以及什么时候 `buf` 可被重用。注意 DNS 查询将会使发送动作最少延迟到下一个时间片执行，如果你想确定发送动作是否发生，使用回调将是唯一的办法。
Example of sending a UDP packet to a random port on `localhost`;
下面是一个发送 UDP 数据包到 `localhost`（本机）一个随机端口的例子：

```
var dgram = require('dgram');
var message = new Buffer("Some bytes");
var client = dgram.createSocket("udp4");
```

```
client.send(message, 0, message.length, 41234, "localhost");
client.close();
```

dgram.bind(path)

For Unix domain datagram sockets, start listening for incoming datagrams on a socket specified by `path`. Note that clients may `send()` without `bind()`, but no datagrams will be received without a `bind()`.
对 Unix 域数据报套接字来说，通过指定一个 `path`（路径）开始在套接字上监听数据报。注意客户端可以无需调用 `bind()`直接使用 `send()`方法发送数据报，但是不使用 `bind()`方法将无法接收到任何数据报。
Example of a Unix domain datagram server that echoes back all messages it receives:

下面是一个 Unix 域数据报服务器的例子，此服务器将接收到的所有消息原样返回：

```
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);
```

Example of a Unix domain datagram client that talks to this server:

下面是一个与上述服务器通信的 Unix 域数据报客户端的例子：

```
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])

For UDP sockets, listen for datagrams on a named `port` and optional `address`. If `address` is not specified, the OS will try to listen on all addresses.
对于 UDP 套接字而言，该方法会在 `port` 指定的端口和可选地址 `address` 上监听数据报。如果 `address` 没有指定，则操作系统会监听所有有效地址。
Example of a UDP server listening on port 41234:

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()

Close the underlying socket and stop listening for data on it. UDP sockets automatically listen for messages, even if they did not call `bind()`.
该函数关闭底层的套接字并停止在其上监听数据。UDP 套接字在没有调用 `bing()` 方法的情况下也自动监听消息。

dgram.address()

Returns an object containing the address information for a socket. For UDP sockets, this object will contain `address` and `port`. For Unix domain sockets, it will contain only `address`.
返回包含套接字地址信息的对象。对于 UDP 套接字来说，这个对象将包含 `address`（地址）和 `port`（端口）。对于 UNIX 域套接字来说，这个对象仅包含 `address`（地址）。

dgram.setBroadcast(flag)

Sets or clears the `SO_BROADCAST` socket option. When this option is set, UDP packets may be sent to a local interface's broadcast address.
设置或者清除套接字的 `SO_BROADCAST`（广播）选项。当该设置生效时，UDP 数据包将被发送至本地网络接口的广播地址。

dgram.setTTL(ttl)

Sets the `IP_TTL` socket option. TTL stands for "Time to Live," but in this context it specifies the number of IP hops that a packet is allowed to go through. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded. Changing TTL values is typically done for network probes or when multicasting.
设置套接字的 `IP_TTL` 选项，TTL 表示“存活时间”，但是在这个上下文环境中，特指一个数据包允许经过的 IP 跳数。每经过一个路由器或者网关 TTL 的值都会减一，如果 TTL 被一个路由器减少到 0，这个数据包将不会继续转发。在网络探针或组播应用中会需要修改 TTL 数值。
The argument to `setTTL()` is a number of hops between 1 and 255. The default on most systems is 64.
在 `setTTL()`调用中设置的跳数介于 1 到 255 之间，大多数系统缺省会设置为 64。

dgram.setMulticastTTL(ttl)

Sets the `IP_MULTICAST_TTL` socket option. TTL stands for "Time to Live," but in this context it specifies the number of IP hops that a packet is allowed to go through, specifically for multicast traffic. Each router or gateway that forwards a packet decrements the TTL. If the TTL is decremented to 0 by a router, it will not be forwarded.
设置套接字的 `IP_MULTICAST_TTL` 选项。TTL 全称"Time to Live"，原指存活时间，但在这里特指在组播通信中数据包允许经过的 IP 跳数。每经过一个路由器或者网关 TTL 的值都会减一。如果 TTL 被一个路由器减少到 0，那么该数据包将不会继续传播。
The argument to `setMulticastTTL()` is a number of hops between 0 and 255. The default on most systems is 64.
`setMulticastTTL()`的参数为 1 至 255 之间的跳数，大部分操作系统的默认值为 64。

dgram.setMulticastLoopback(flag)

Sets or clears the `IP_MULTICAST_LOOP` socket option. When this option is set, multicast packets will also be received on the local interface.
设置或清除套接字的 `IP_MULTICAST_LOOP` 选项。当该选项生效时，多路传播的数据包也会被本地网络接口接收到。

dgram.addMembership(multicastAddress, [multicastInterface])

Tells the kernel to join a multicast group with `IP_ADD_MEMBERSHIP` socket option.

该方法通知系统内核使用 `IP_ADD_MEMBERSHIP` 套接字选项将套接字加入一个组播组。
If `multicastAddress` is not specified, the OS will try to add membership to all valid interfaces.
如果没有指定 `multicastInterface` 参数，操作系统将尝试把所有有效的网络接口加入组播组。

dgram.dropMembership(multicastAddress, [multicastInterface])

Opposite of `addMembership` - tells the kernel to leave a multicast group with `IP_DROP_MEMBERSHIP` socket option. This is automatically called by the kernel when the socket is closed or process terminates, so most apps will never need to call this.
与 `addMembership` 相反，该方法通知系统内核使用 `IP_DROP_MEMBERSHIP` 选项使套接字脱离组播组。由于当套接字关闭或进程终止时该操作会自动执行，所以大部分的应用不需要手动调用该方法。
If `multicastAddress` is not specified, the OS will try to drop membership to all valid interfaces.
如果 `multicastInterface` 没有指定，操作系统会尝试将所有可用网络接口从组播组里脱离。

HTTP HTTP 模块

To use the HTTP server and client one must `require('http')`.
如果要使用 HTTP 的服务器以及客户端，需使用 `require('http')`加载 HTTP 模块。
The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses--the user is able to stream data.

Node 中的 HTTP 接口在设计时就考虑到了要支持 HTTP 协议的很多特性，并且使用简单。特别是可以处理那些内容庞大，有可能是块编码的消息。该接口被设计为从不缓冲整个请求或相应，这样用户就可以以流的方式处理数据。

HTTP message headers are represented by an object like this:

HTTP 头信息以如下对象形式表示：

```
{ 'content-length': '123',
```

```
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

Keys are lowercased. Values are not modified.

所有键名被转为小写，而值不会被修改。

In order to support the full spectrum of possible HTTP applications, Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.

为了支持尽可能多的 HTTP 应用，Node 提供非常底层的 HTTP API。它只处理流相关的操作以及进行信息解析。API 将信息解析为头部和正文，但并不解析实际的头部和正文内的具体内容。

http.Server

This is an `EventEmitter` with the following events:
这是一个带有如下事件的 `EventEmitter` 事件触发器：

Event: 'request' 事件：'request'

`function (request, response) { }`
`request` is an instance of `http.ServerRequest` and `response` is an instance of `http.ServerResponse`
`request` 是 `http.ServerRequest` 的一个实例，而 `response` 是 `http.ServerResponse` 的一个实例。

Event: 'connection' 事件：'connection'

`function (stream) { }`
When a new TCP stream is established. `stream` is an object of type `net.Stream`. Usually users will not want to access this event. The `stream` can also be accessed at `request.connection`.
当一个新的 TCP 流建立后触发此事件。`stream` 是一个 `net.Stream` 类型的对象，通常用户不会使用这个事件。参数 `stream` 也可以在 `request.connection` 中获得。

Event: 'close' 事件：'close'

`function (errno) { }`
Emitted when the server closes.

当服务器关闭的时候触发此事件。

Event: 'request' 事件：'request'

`function (request, response) {}`
Emitted each time there is request. Note that there may be multiple requests per connection (in the case of keep-alive connections).

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

Event: 'checkContinue' 事件：'checkContinue'

`function (request, response) {}`
Emitted each time a request with an http Expect: 100-continue is received. If this event isn't listened for, the server will automatically respond with a 100 Continue as appropriate.

每当带有 Exception: 100-continue 头的请求被接收到时触发此事件。如果该事件未被监听，服务器会视情况自动的使用 100 Continue 应答。

Handling this event involves calling `response.writeContinue` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g., 400 Bad Request) if the client should not continue to send the request body.
该事件的处理涉及两种情况，如果客户端应当继续发送请求正文，那么需要调用 `response.writeContinue`，而如果客户端不应该继续发送请求正文，那么应该产生一个适当的 HTTP 回应（如 400 错误请求）。
Note that when this event is emitted and handled, the `request` event will not be emitted.
注意如果该事件被触发并处理的话，那么将不再触发 `request` 事件。

Event: 'upgrade' 事件：'upgrade'

`function (request, socket, head)`
Emitted each time a client requests a http upgrade. If this event isn't listened for, then clients requesting an upgrade will have their connections closed.

每当一个客户端请求 http upgrade 时触发此消息。如果这个事件没有监听，那么请求 upgrade 的客户端的连接将被关闭。

- `request` is the arguments for the http request, as it is in the request event. `request` 代表一个 http 请求的相关参数，和它在 request 事件中的意思相同。
- `socket` is the network socket between the server and client. `socket` 是在服务器与客户端之间连接使用的网络套接字。
- `head` is an instance of Buffer, the first packet of the upgraded stream, this may be empty. `head` 是一个缓冲器实例，是 upgraded 流的第一个包，这个缓冲器可以是空的。

After this event is emitted, the request's socket will not have a `data` event listener, meaning you will need to bind to it in order to handle data sent to the server on that socket.
当此事件被触发后，该请求所使用的套接字将不会有一个 `data` 事件监听器。这意味着你如果需要处理通过这个套接字发送到服务器端的数据则需要自己绑定 `data` 事件监听器。

Event: 'clientError' 事件：'clientError'

`function (exception) {}`
If a client connection emits an 'error' event - it will forwarded here.

当客户端连接出现错误时会触发'error'事件。

http.createServer(requestListener)

Returns a new web server object.

返回一个新的 web server 对象。

The `requestListener` is a function which is automatically added to the `'request'` event.
requestListener 监听器会自动添加到`'request'`事件中。

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

Begin accepting connections on the specified port and hostname. If the hostname is omitted, the server will accept connections directed to any IPv4 address (`INADDR_ANY`).
在指定端口和主机名上接受连接。如果 hostname 没有指定，服务器将直接在此机器的所有 IPV4 地址上接受连接(`INADDR_ANY`)。

To listen to a unix socket, supply a filename instead of port and hostname.

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

This function is asynchronous. The last parameter `callback` will be called when the server has been bound to the port.
这个方法是一个异步的方法，当服务器已经在此端口上完成绑定后讲调用 `callback` 回调函数。

server.listen(path, [callback])

Start a UNIX socket server listening for connections on the given `path`.
建立一个 UNIX 套接字服务器并在指定 `path` 路径上监听。
This function is asynchronous. The last parameter `callback` will be called when the server has been bound.
这个方法是一个异步的方法，当服务器完成绑定后将调用 `callback` 回调函数。

server.close()

Stops the server from accepting new connections.

使此服务器停止接受任何新连接。

http.ServerRequest

This object is created internally by a HTTP server -- not by the user -- and passed as the first argument to a `'request'` listener.
这个对象通常由 HTTP 服务器（而非用户）自动建立，并作为第一个参数传给 `'request'` 监听器。
This is an `EventEmitter` with the following events:
这是一个带有如下事件的 `EventEmitter` 事件触发器：

Event: 'data' 事件：'data'

`function (chunk) { }`
Emitted when a piece of the message body is received.

当接收到信息正文中的一部分时候会触发此事件。

Example: A chunk of the body is given as the single argument. The transfer-encoding has been decoded. The body chunk is a string. The body encoding is set with `request.setBodyEncoding()`.

例如：正文的数据块将作为唯一的参数传递给回调函数。此时传输编码已被解码。正文数据块是一个字符串，正文的编码由 `request.setBodyEncoding()`方法设定。

Event: 'end' 事件：'end'

`function () { }`
Emitted exactly once for each message. No arguments. After emitted no other events will be emitted on the request.

每次完全接收完信息后都会触发一次，不接受任何参数。当这个事件被触发后，将不会再触发其他事件。

request.method

The request method as a string. Read only. Example: `'GET'`, `'DELETE'`.
表示请求方式的只读字符串。例如`'GET'`，`'DELETE'`。

request.url

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

代表所请求 URL 的字符串。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
```

Then `request.url` will be:
则 `request.url` 应当是：

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

If you would like to parse the URL into its parts, you can use `require('url').parse(request.url)`. Example:
如果你想要解析这个 URL 中的各个部分，可以使用 `require('url').parse(request.url)`。例如：

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

If you would like to extract the params from the query string, you can use the `require('querystring').parse` function, or pass `true` as the second argument to `require('url').parse`. Example:
如果你想从查询字符串中提取所有参数，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。例如：

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

request.headers

Read only.

只读。

request.trailers

Read only; HTTP trailers (if present). Only populated after the 'end' event.

只读，HTTP 尾部（如果存在的话），只有在'end'事件被触发后该值才会被填充。

request.httpVersion

The HTTP protocol version as a string. Read only. Examples: `'1.1'`, `'1.0'`. Also `request.httpVersionMajor` is the first integer and `request.httpVersionMinor` is the second.

只读的，以字符串形式表示 HTTP 协议版本。例如`'1.1'`，`'1.0'`。
`request.httpVersionMajor` 对应版本号的第一个数字，
`request.httpVersionMinor` 则对应第二个数字。

request.setEncoding(encoding=null)

Set the encoding for the request body. Either `'utf8'` or `'binary'`. Defaults to `null`, which means that the `'data'` event will emit a `Buffer` object..
设置此请求正文的字符编码，`'utf8'`或者`'binary'`。缺省值是`null`，这表示`'data'`事件的参数将会是一个缓冲器对象。

request.pause()

Pauses request from emitting events. Useful to throttle back an upload.

暂停此请求的事件触发。对于控制上传非常有用。

request.resume()

Resumes a paused request.

恢复一个暂停的请求。

request.connection

The `net.Stream` object associated with the connection.
与当前连接相关联的 `net.Stream` 对象。
With HTTPS support, use request.connection.verifyPeer() and request.connection.getPeerCertificate() to obtain the client's authentication details.

对于使用 HTTPS 的连接，可使用 request.connection.verifyPeer()和 request.connection.getPeerCertificate()来获得客户端的认证详情。

http.ServerResponse

This object is created internally by a HTTP server--not by the user. It is passed as the second parameter to the `'request'` event. It is a `Writable Stream`.
这个对象由 HTTP 服务器（而非用户）自动建立。它作为`'request'`事件的第二个参数，这是一个 `Writable Stream` 可写流。

response.writeContinue()

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the the `checkContinue` event on `Server`.
发送 HTTP/1.1 100 Continue 消息给客户端，通知客户端可以发送请求的正文。参见服务器 `Server` 中的 `checkContinue` 事件。

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

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, are the response headers. Optionally one can give a human-readable `reasonPhrase` as the second argument.
这个方法用来发送一个响应头，statusCode 是一个由 3 位数字所构成的 HTTP 状态码，比如 `404` 之类。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理解的 `reasonPhrase` 作为第二个参数。
Example:

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

This method must only be called once on a message and it must be called before `response.end()` is called.
在一次请求响应中此方法只能调用一次，并且必须在调用 `response.end()`之前调用。
If you call `response.write()` or `response.end()` before calling this, the implicit/mutable headers will be calculated and call this function for you.
如果你在 `response.write()`或者 `response.end()`之后调用此方法，响应头的内容将是不确定而且不可知的。

response.statusCode

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be send to the client when the headers get flushed.

当隐式的发送响应头信息（没有明确调用 `response.writeHead()`）时，使用此属性将设置返回给客户端的状态码。状态吗将在响应头信息发送时一起被发送。
Example:

例如：

```
response.statusCode = 404;
```

response.setHeader(name, value)

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, it's value will be replaced. Use an array of strings here if you need to send multiple headers with the same name.

在隐式的响应头基础上设置单个头信息。如果存在同名的待发送头信息，那么该头信息的值将被替换。如果你想发送相同名字的多个头部信息，可以使用字符串数组的形式设置。

Example:

例如：

```
response.setHeader("Content-Type", "text/html");
```

或者

```
response.setHeader("Set-Cookie", ["type=ninja",
"language=javascript"]);
```

response.getHeader(name)

Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive. This can only be called before headers get implicitly flushed.

读取已经排列好但尚未发送给客户端的头部信息，注意参数名不区分大小写。此方法必须在响应头信息隐式发送之前调用。

Example:

例如：

```
var contentType = response.getHeader('content-type');
```

response.removeHeader(name)

Removes a header that's queued for implicit sending.

移除等待隐式发送的头部信息。

Example:

例如：

```
response.removeHeader("Content-Encoding");
```

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

If this method is called and `response.writeHead()` has not been called, it will switch to implicit header mode and flush the implicit headers.
如果在`response.writeHead()`调用之前调用该函数，将会切换到隐式发送响应头信息的模式并发送隐式的头部信息。
This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

它负责发送响应正文中的一部分数据，可以多次调用此方法以发送正文中多个连续的部分。

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`.
`chunk`可以是一个字符串或者一个缓冲器。如果`chunk`是一个字符串，则第二个参数指定用何种编码方式将字符串编码为字节流。缺省情况下，`encoding`为`'utf8'`。
Note: This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

注意：这是一个原始格式 HTTP 正文，和高层协议中的多段正文编码格式无关。

The first time `response.write()` is called, it will send the buffered header information and the first body to the client. The second time `response.write()` is called, Node assumes you're going to be streaming data, and sends that separately. That is, the response is buffered up to the first chunk of body.
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正文发送给客户。 当第二次调用`response.write()`的时候，Node 将假定你想要逐次发送流数据。换句话说，响应被缓冲直到正文的第一块被发送。

response.addTrailers(headers)

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

该方法在响应中添加 HTTP 尾部头信息（在消息尾部的头信息）。

Trailers will **only** be emitted if chunked encoding is used for the response; if it is not (e.g., if the request was HTTP/1.0), they will be silently discarded.

仅当响应报文使用 chunked 编码时，尾部信息才会发送；否则（例如请求的协议版本为 HTTP/1.0）它们会被抛弃而没有提示。

Note that HTTP requires the `Trailer` header to be sent if you intend to emit trailers, with a list of the header fields in its value. E.g.,
注意如果你想发送尾部信息，则需要在 HTTP 头中添加`Trailer`。

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'TraceInfo' });
response.write(fileData);
```

```
response.addTrailers({'Content-MD5':
"7895bf4b8828b55ceaf47747b4bca667"});
response.end();
```

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

This method signals to the server that all of the response headers and body has been sent; that server should consider this message complete. The method, `response.end()`, MUST be called on each response.
这个方法通知服务器所有的响应头和响应正文都已经发出；服务器在此调用后认为这条信息已经发送完毕。在每个响应上都必须调用 `response.end()`方法。
If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `response.end()`.
如果指定了`data`参数，就相当先调用`response.write(data, encoding)`再调用`response.end()`。

http.request(options, callback)

Node maintains several connections per server to make HTTP requests. This function allows one to transparently issue requests.

Node 为一个目标服务器维护多个连接用于 HTTP 请求。通过这个方法可以向服务器发送请求。

Options:

选项：

- `host`: A domain name or IP address of the server to issue the request to. `host`: 请求的服务器域名或者 IP 地址。
- `port`: Port of remote server. `port`: 远端服务器的端口。
- `method`: A string specifying the HTTP request method. Possible values: `'GET'` (default), `'POST'`, `'PUT'`, and `'DELETE'`. `method`: 指定 HTTP 请求的方法类型，可选的值有：`'GET'`（默认），`'POST'`，`'PUT'`，以及`'DELETE'`。
- `path`: Request path. Should include query string and fragments if any. E.G. `'/index.html?page=12'` `path`: 请求地址，可包含查询字符串以及可能存在的锚点。例如`'/index.html?page=12'`。
- `headers`: An object containing request headers. `headers`: 一个包含请求头的对象。

`http.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.
`http.request()`函数返回`http.ClientRequest`类的一个实例。`ClientRequest`对象是一个可写流，如果你需要用 POST 方法上传一个文件，可将其写入到`ClientRequest`对象中。
Example:

例子：

```
var options = {
  host: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST'
};

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);
  });
});

// write data to request body
req.write('data\n');
req.write('data\n');
req.end();
```

Note that in the example `req.end()` was called. With `http.request()` one must always call `req.end()` to signify that you're done with the request - even if there is no data being written to the request body.
注意这个例子中`req.end()`被调用了。无论请求正文是否包含数据，每一次调用`http.request()`最后都需要调用一次`req.end()`表示已经完成了请求。

If any error is encountered during the request (be that with DNS resolution, TCP level errors, or actual HTTP parse errors) an `'error'` event is emitted on the returned request object.
如果在请求过程中出现了错误（可能是 DNS 解析、TCP 的错误、或者 HTTP 解析错误），返回的请求对象上的`'error'`的事件将被触发。
There are a few special headers that should be noted.

如下特别的消息头应当注意：

- Sending a 'Connection: keep-alive' will notify Node that the connection to the server should be persisted until the next request.

  发送'Connection: keep-alive'头部将通知 Node 此连接将保持到下一此请求。

- Sending a 'Content-length' header will disable the default chunked encoding.

  发送'Content-length'头将使默认的分块编码无效。

- Sending an 'Expect' header will immediately send the request headers. Usually, when sending 'Expect: 100-continue', you should both set a timeout and listen for the `continue` event. See RFC2616 Section 8.2.3 for more information.
  发送'Expect'头部将引起请求头部立即被发送。通常情况，当发送'Expect: 100-continue'时，你需要监听 `continue` 事件的同时设置超时。参见 RFC2616 8.2.3 章节以获得更多的信息。

http.get(options, callback)

Since most requests are GET requests without bodies, Node provides this convenience method. The only difference between this method and `http.request()` is that it sets the method to GET and calls `req.end()` automatically.
由于大部分请求是不包含正文的 GET 请求，Node 提供了这个方便的方法。与`http.request()`唯一的区别是此方法将请求方式设置为 GET，并且自动调用`req.end()`。
Example:

例子：

```
var options = {
  host: 'www.google.com',
  port: 80,
  path: '/index.html'
};

http.get(options, function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

http.Agent

http.getAgent(host, port)

`http.request()` uses a special `Agent` for managing multiple connections to an HTTP server. Normally `Agent` instances should not be exposed to user code, however in certain situations it's useful to check the status of the agent. The `http.getAgent()` function allows you to access the agents.
`http.request()`使用一个特别的 `Agent` 代理来管理到一个服务器的多个连接，通常 `Agent` 对象不应该暴露给用户。但在某些特定的情况下，检测代理的状态是非常有用的。`http.getAgent()`函数允许你访问代理对象。

Event: 'upgrade' 事件：'upgrade'

`function (request, socket, head)`
Emitted each time a server responds to a request with an upgrade. If this event isn't being listened for, clients receiving an upgrade header will have their connections closed.

当服务器响应 upgrade 请求时触发此事件。如果这个事件没有被监听，客户端接收到 upgrade 头会导致连接被关闭。

See the description of the `upgrade` event for `http.Server` for further details.
可以查看 http.Server 关于 upgrade 事件的解释来了解更多内容。

Event: 'continue' 事件：'continue'

`function ()`
Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

当服务器发送'100 Continue'答复时触发此事件，这通常是因为请求头信息中包含'Expect: 100-continue'。此事件指示客户端可是开始发送请求正文了。

agent.maxSockets

By default set to 5. Determines how many concurrent sockets the agent can have open.

默认值为 5，指定代理能同时并发打开的套接字数量。

agent.sockets

An array of sockets currently in use by the Agent. Do not modify.

当前代理使用的套接字数组，不能更改。

agent.queue

A queue of requests waiting to be sent to sockets.

待发送到套接字的请求队列。

http.ClientRequest

This object is created internally and returned from `http.request()`. It represents an *in-progress* request whose header has already been queued. The header is still mutable using the `setHeader(name, value)`, `getHeader(name)`, `removeHeader(name)` API. The actual header will be sent along with the first data chunk or when closing the connection.
这个对象是在调用 `http.request()`时产生并返回的。它表示一个*正在进行*中且头部信息已经排列好了的请求。这时候通过 `setHeader(name, value)`，`getHeader(name)`，`removeHeader(name)`这些 API 还可以改变头部信息，实际的头部信息将随着第一块数据发送，或者在关闭连接时发送出去。
To get the response, add a listener for `'response'` to the request object. `'response'` will be emitted from the request object when the response headers have been received. The `'response'` event is

executed with one argument which is an instance of `http.ClientResponse`.
为了获得响应，为请求对象增加一个对响应的监听器。

During the `'response'` event, one can add listeners to the response object; particularly to listen for the `'data'` event. Note that the `'response'` event is called before any part of the response body is received, so there is no need to worry about racing to catch the first part of the body. As long as a listener for `'data'` is added during the `'response'` event, the entire body will be caught.
在`'response'`事件中，可以给响应对象添加监听器，特别是监听`'data'`事件，注意`'response'`事件在正文接收之前就已经被调用，所以不需要担心捕获不到正文的第一部分，一旦在`'response'`事件中添加了对`'data'`的监听器，那么整个正文将被捕获。

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

This is a `Writable Stream`.
这是一个`Writable Stream`可写流。
This is an `EventEmitter` with the following events:
这是一个包含下述事件的`EventEmitter`事件触发器：

Event 'response' 事件：'response'

`function (response) { }`

Emitted when a response is received to this request. This event is emitted only once. The `response` argument will be an instance of `http.ClientResponse`.
当请求的响应到达时触发，该事件仅触发一次。`response`参数是`http.ClientResponse`的一个实例。

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

Sends a chunk of the body. By calling this method many times, the user can stream a request body to a server--in that case it is suggested to use the `['Transfer-Encoding', 'chunked']` header line when creating the request.
发送正文中的一块。用户可以通过多次调用这个方法将请求正文以流的方式发送到服务器。此种情况建议在建立请求时使用`['Transfer-Encoding', 'chunked']`请求头。
The `chunk` argument should be an array of integers or a string.
参数`chunk`应当是一个整数数组或字符串。
The `encoding` argument is optional and only applies when `chunk` is a string.
参数`encoding`是可选的，仅在`chunk`为字符串时可用。

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

Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream. If the request is chunked, this will send the terminating `'0\r\n\r\n'`.
完成本次请求的发送。如果正文中的任何一个部分没有来得及发送，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'`。
If `data` is specified, it is equivalent to calling `request.write(data, encoding)` followed by `request.end()`.
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()。

request.abort()

Aborts a request. (New since v0.3.8.)

阻止一个请求。（v0.3.8 中新增的方法。）

http.ClientResponse

This object is created when making a request with `http.request()`. It is passed to the `'response'` event of the request object.

这个对象在使用 `http.request()`发起请求时被创建，它会以参数的形式传递给 request 对象的`'response'`事件。
The response implements the `Readable Stream` interface.
'response'实现了可读流的接口。

Event: 'data' 事件：'data'

`function (chunk) {}`
Emitted when a piece of the message body is received.

当接收到消息正文一部分的时候触发。

Event: 'end' 事件：'end'

`function () {}`
Emitted exactly once for each message. No arguments. After emitted no other events will be emitted on the response.

对每次消息请求只触发一次，该事件被触发后将不会再有任何事件在响应中被触发。

response.statusCode

The 3-digit HTTP response status code. E.G. `404`.
3 个数字组成的 HTTP 响应状态吗。例如 `404`。

response.httpVersion

The HTTP version of the connected-to server. Probably either `'1.1'` or `'1.0'`. Also `response.httpVersionMajor` is the first integer and `response.httpVersionMinor` is the second.
连接至服务器端的 HTTP 版本，可能的值为`'1.1'` or `'1.0'`，你也可以使用`response.httpVersionMajor`获得版本号第一位，使用`response.httpVersionMinor`获得版本号第二位。

response.headers

The response headers object.

响应头部对象。

response.trailers

The response trailers object. Only populated after the 'end' event.

响应尾部对象，在'end'事件发生后填充该对象。

response.setEncoding(encoding=null)

Set the encoding for the response body. Either `'utf8'`, `'ascii'`, or `'base64'`. Defaults to `null`, which means that the `'data'` event will emit a `Buffer` object..
设置响应正文的编码，可以是`'utf8'`，`'ascii'`，或者`'base64'`。默认值为`null`，此种情况下`'data'`事件将发送缓冲器对象。

response.pause()

Pauses response from emitting events. Useful to throttle back a download.

暂停响应的事件激发，对控制下载流量非常有用。

response.resume()

Resumes a paused response.

恢复一个已经暂停的响应。

HTTPS HTTPS 模块

HTTPS is the HTTP protocol over TLS/SSL. In Node this is implemented as a separate module.

HTTPS 是基于 TLS（Transport Layer Security 传输层安全）/SSL（Secure Sockets Layer 安全套接层）的 HTTP 协议，在 Node 中，它作为一个独立的模块被实现

https.Server
https.createServer

Example:

例子：

```
// 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);
```

https.request(options, callback)

Makes a request to a secure web server. Similar options to `http.request()`.
向安全的 web 服务器发送请求，可选参数和 `http.request()`类似。
Example:

例子：

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

var options = {
  host: '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);
});
```

The options argument has the following options

options 参数可包含以下内容：

- host: IP or domain of host to make request to. Defaults to `'localhost'`. host: 要访问的主机的 IP 地址或域名。默认为 `'localhost'`。
- port: port of host to request to. Defaults to 443. port: 要访问的主机端口。默认为 433。
- path: Path to request. Default `'/'`. path: 要访问的路径。默认为 `'/'`。
- method: HTTP request method. Default `'GET'`. method: HTTP 请求方式。默认为`'GET'`。
- key: Private key to use for SSL. Default `null`. key: SSL 所使用的私钥。默认为 `null`。
- cert: Public x509 certificate to use. Default `null`. cert: 所使用的 x509 公钥证书。默认为 `null`。
- ca: An authority certificate or array of authority certificates to check the remote host against. ca: 用于验证远程主机身份的一个认证中心证书（或多个认证中心证书数组）。

https.get(options, callback)

Like `http.get()` but for HTTPS.
类似 `http.get()`但它基于 HTTPS 协议。
Example:

例子：

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

https.get({ host: 'encrypted.google.com', path: '/' }, 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);
});
```

URL URL 模块

This module has utilities for URL resolution and parsing. Call `require('url')` to use it.
此模块包含用于解析和分析 URL 的工具。可通过 `require('url')`访问他们。
Parsed URL objects have some or all of the following fields, depending on whether or not they exist in the URL string. Any parts that are not in the URL string will not be in the parsed object. Examples are shown for the URL

解析后的 URL 对象包含下述部分或全部字段。具体包含哪些字段取决于解析前的 URL 字符串中是否存在这些字段。在原始的 URL 字符串中不存在的字段在解析后的对象中也不会包含。以下面这个 URL 为例：

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

- `href`: The full URL that was originally parsed.
  `href`：完整的原始 URL 字符串。
  Example:
  `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`
  例如：
  `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`
- `protocol`: The request protocol.
  `protocol`：请求所使用的协议。
  Example: `'http:'`

例如：`'http:'`

- `host`: The full host portion of the URL, including port and authentication information.
  `host`：URL 中关于主机的完整信息，包括端口以及用户身份验证信息。
  Example: `'user:pass@host.com:8080'`
  例如：`'user:pass@host.com:8080'`
- `auth`: The authentication information portion of a URL.
  `auth`：URL 中的用户身份验证信息。
  Example: `'user:pass'`
  例如：`'user:pass'`
- `hostname`: Just the hostname portion of the host.
  `hostname`：主机信息中的主机名部分。
  Example: `'host.com'`
  例如：`'host.com'`
- `port`: The port number portion of the host.
  `port`：主机信息中的端口部分。
  Example: `'8080'`
  例如：`'8080'`
- `pathname`: The path section of the URL, that comes after the host and before the query, including the initial slash if present.
  `pathname`：URL 中的路径部分，这部分信息位于主机信息之后查询字符串之前。如果存在最上层根目录符号`'/'`，也将包含在此信息中。
  Example: `'/p/a/t/h'`
  例如：`'/p/a/t/h'`
- `search`: The 'query string' portion of the URL, including the leading question mark.
  `search`：URL 中的'query string'（查询字符串）部分，包括前导的`'?'`。
  Example: `'?query=string'`
  例如：`'?query=string'`
- `query`: Either the 'params' portion of the query string, or a querystring-parsed object.
  `query`：查询字符串中的参数部分，或者是由查询字符串解析出的对象。
  Example: `'query=string'` or `{'query':'string'}`
  例如：`'query=string'` or `{'query':'string'}`
- `hash`: The 'fragment' portion of the URL including the pound-sign.
  `hash`：URL 中的锚点部分，包含前导的`'#'`。
  Example: `'#hash'`
  例如：`'#hash'`

The following methods are provided by the URL module:

URL 模块提供了如下方法：

url.parse(urlStr, parseQueryString=false)

Take a URL string, and return an object. Pass `true` as the second argument to also parse the query string using the `querystring` module.
以一个 URL 字符串为参数，返回一个解析后的对象。如设置第二个参数为 `true`，则会使用 `querystring` 模块解析 URL 中的查询字符串。

url.format(urlObj)

Take a parsed URL object, and return a formatted URL string.

以一个解析后的 URL 对象为参数，返回格式化的 URL 字符串。

url.resolve(from, to)

Take a base URL, and a href URL, and resolve them as a browser would for an anchor tag.

指定一个默认 URL 地址，和一个链接的目标 URL 地址，返回链接的绝对 URL 地址。处理方式与浏览器处理锚点标签的方法一致。 ## Query String 查询字符串模块

This module provides utilities for dealing with query strings. It provides the following methods:

该模块为处理查询字符串提供了一些实用的功能。包括如下的方法：

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

Serialize an object to a query string. Optionally override the default separator and assignment characters.

将一个对象序列化为一个查询字符串，可选择是否覆盖默认的分隔符和赋值符。

Example:

例如：

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

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

```
// returns
'foo:bar;baz:bob'
```

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

Deserialize a query string to an object. Optionally override the default separator and assignment characters.

将一个查询字符串反序列化为一个对象，可选择是否覆盖默认的分隔符和赋值符。

Example:

例如：

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

querystring.escape

The escape function used by `querystring.stringify`, provided so that it could be overridden if necessary.
由 `querystring.stringify` 使用的转义函数，需要时可重置其内容。

querystring.unescape

The unescape function used by `querystring.parse`, provided so that it could be overridden if necessary.
由 `querystring.parse` 使用的反转义函数，需要时可重置其内容。

REPL 交互式解释器

A Read-Eval-Print-Loop (REPL) is available both as a standalone program and easily includable in other programs. REPL provides a way to interactively run JavaScript and see the results. It can be used for debugging, testing, or just trying things out.

交互式解释器（REPL）既可以作为一个独立的程序运行，也可以很容易地包含在其他程序中作为整体程序的一部分使用。REPL 为运行 JavaScript 脚本与查看运行结果提供了一种交互方式，通常 REPL 交互方式可以用于调试、测试以及试验某种想法。

By executing `node` without any arguments from the command-line you will be dropped into the REPL. It has simplistic emacs line-editing.
在命令行中不带任何参数地运行 `node` 命令，就可以进入 REPL 环境，在该环境下你可以进行一些类似 Emacs 的行编辑操作。

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

For advanced line-editors, start node with the environmental variable `NODE_NO_READLINE=1`. This will start the REPL in canonical terminal settings which will allow you to use with `rlwrap`.
为了进行高级的行编辑操作，可以设置环境变量 `NODE_NO_READLINE=1` 并启动 node。这种情况 REPL 会进入标准终端设置模式，这此模式下你可以使用 `rlwrap`。
For example, you could add this to your bashrc file:

比如，你可以把下列设置添加到你的 bashrc 文件中：

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

repl.start(prompt='> ', stream=process.stdin)

Starts a REPL with `prompt` as the prompt and `stream` for all I/O. `prompt` is optional and defaults to `> `. `stream` is optional and defaults to `process.stdin`.
启动一个 REPL，使用 `prompt` 作为输入提示符，在 `stream` 上进行所有 I/O 操作。`prompt` 是可选参数，默认值为`> `，`stream` 也是可选参数，默认值为 `process.stdin`。
Multiple REPLs may be started against the same running instance of node. Each will share the same global object but will have unique I/O.

一个 node 实例中可以启动多个 REPL，它们共享相同的全局对象，但拥有各自独立的 I/O。

Here is an example that starts a REPL on stdin, a Unix socket, and a TCP socket:

下面是的例子展示分别在标准输入，Unix 套接字及 TCP 套接字上启动的 REPL 示例：

```
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);
```

Running this program from the command line will start a REPL on stdin. Other REPL clients may connect through the Unix socket or TCP socket. `telnet` is useful for connecting to TCP sockets, and `socat` can be used to connect to both Unix and TCP sockets.

在命令行中运行这段程序将首先使用标准输入启动 REPL。其他的 REPL 终端可以通过 Unix 套接字或者 TCP 套接字进行连接。可使用 `telnet` 程序连接到 TCP 套接字，而 `socat` 程序既可以连接到 Unix 套接字也可以连接连接到 TCP 套接字。
By starting a REPL from a Unix socket-based server instead of stdin, you can connect to a long-running node process without restarting it.

若要连接到一个长时间运行的 node 进程而无需重启进程，你应该将 REPL 启动在 Unix 套接字上，非不要将其启动在标准输出上。

REPL Features REPL 特性

Inside the REPL, Control+D will exit. Multi-line expressions can be input.

在 REPL 中，操作组合键 Control+D 可以退出。可以输入多行表达式。

The special variable `_` (underscore) contains the result of the last expression.
特殊变量`_`（下划线）包含了上一表达式的结果。

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

The REPL provides access to any variables in the global scope. You can expose a variable to the REPL explicitly by assigning it to the `context` object associated with each `REPLServer`. For example:
在 REPL 可以访问全局作用域中的任何变量。为了在 REPL 中访问一个变量，你只要将此变量显性地分配给相应 `REPLServer` 的 `context` 对象。示例如下：

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

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

Things in the `context` object appear as local within the REPL:
`context` 对象中的变量在 REPL 中看起来就像是本地变量。

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

There are a few special REPL commands:

以下是另外一些特殊的 REPL 命令：

- `.break` - While inputting a multi-line expression, sometimes you get lost or just don't care about completing it. `.break` will start over. `.break` - 当你输入多行表达式，如果想放弃当前的输入，可以用`.break` 跳出。
- `.clear` - Resets the `context` object to an empty object and clears any multi-line expression. `.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` - Close the I/O stream, which will cause the REPL to exit. `.exit` - 该命令用于关闭 I/O 流，并退出 REPL。
- `.help` - Show this list of special commands. `.help` - 输出特殊命令的列表。

Child Processes 子进程

Node provides a tri-directional `popen(3)` facility through the `ChildProcess` class.
在 Node 里，`ChildProcess` 类提供了一个 3 向的 `popen(3)`机制。
It is possible to stream data through the child's `stdin`, `stdout`, and `stderr` in a fully non-blocking way.
子进程类中的 `stdin`， `stdout`，和 `stderr` 可以使数据流以完全非阻塞式的方式（non-blocking way）流动（stream）
To create a child process use `require('child_process').spawn()`.
调用 `require('child_process').spawn()`可以创建一个子进程（child process）
Child processes always have three streams associated with them.
`child.stdin`, `child.stdout`, and `child.stderr`.
`child.stdin`，`child.stdout`，和 `child.stderr` 等 3 个流总是伴随着子进程。

`ChildProcess` is an `EventEmitter`.
`ChildProcess` 是一种 `EventEmitter`（事件触发器）。

Event: 'exit' 事件：'exit'

`function (code, signal) {}`
This event is emitted after the child process ends. If the process terminated normally, `code` is the final exit code of the process, otherwise `null`. If the process terminated due to receipt of a signal, `signal` is the string name of the signal, otherwise `null`.
当子进程结束时，（Node）触发该事件。如果进程正常终结，那么进程的最终退出代码（ final exit code）为 `code`，否则为 `null`。如果进程的结束是因为接收到一个信号，那么 `signal` 为 string 型的信号名称，否则为 `null`。
See `waitpid(2)`.
参见 `waitpid(2)`。

child.stdin

A `Writable Stream` that represents the child process's `stdin`.
Closing this stream via `end()` often causes the child process to terminate.
一个 `Writable Stream`（可写流），表示子进程的 `stdin`。调用 `end()`来关闭这个流通常会终结整个子进程。

child.stdout

A `Readable Stream` that represents the child process's `stdout`.
一个 `Readable Stream`（可读流），表示子进程的 `stdout`（标准输出）。

child.stderr

A `Readable Stream` that represents the child process's `stderr`.
一个 `Readable Stream`（可读流），表示子进程的 `stderr`（标准错误）。

child.pid

The PID of the child process.

子进程的 PID（进程编号）。

Example:

例如：

```
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])

Launches a new process with the given `command`, with command line arguments in `args`. If omitted, `args` defaults to an empty Array.
使用指定的 `command` 创建一个新进程，命令行参数为 `args`。缺省下，`args` 默认为一个空数组。
The third argument is used to specify additional options, which defaults to:

第三个参数用于指定附加的选项，默认如下：

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

`cwd` allows you to specify the working directory from which the process is spawned. Use `env` to specify environment variables that will be visible to the new process. With `customFds` it is possible to hook up the new process' [stdin, stout, stderr] to existing streams; `-1` means that a new stream should be created. `setsid`, if set true, will cause the subprocess to be run in a new session.
参数 `cwd` 允许你指定要创建的子进程的工作目录(working directory)。参数 `env` 可以指定哪些环境变量在新进程中是可见的。参数 `customFds` 可以使新进程中的[stdin, stout, stderr]和已存在的流进行挂接（hook up）。参数`-1` 可以建立一个新的流。如果设置参数 `setsid` 为 true，该子进程将转入到一个新会话（session）中运行。
Example of running `ls -lh /usr`, capturing `stdout`, `stderr`, and the exit code:

例：运行 `ls -lh /usr` 命令，捕获 `stdout`， `stderr` 和退出代码：

```
var util   = require('util'),
    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('exit', function (code) {
  console.log('child process exited with code ' + code);
});
```

Example: A very elaborate way to run 'ps ax | grep ssh'

例：运行'ps ax | grep ssh'命令的完整方法：

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

Example of checking for failed exec:

检测 exec 执行是否失败的例子：

```
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.');
  }
});
```

See also: `child_process.exec()`
可参见：`child_process.exec()`

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

High-level way to execute a command as a child process, buffer the output, and return it all in a callback.

以子进程方式执行一个命令的高级方法。所有输出经过缓冲后在同一个回调函数中返回。

```
var util   = require('util'),
    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);
    }
});
```

The callback gets the arguments `(error, stdout, stderr)`. On success, `error` will be `null`. On error, `error` will be an instance of `Error` and `err.code` will be the exit code of the child process, and `err.signal` will be set to the signal that terminated the process.
回调函数获得`(error, stdout, stderr)`3 个参数。成功时，`error` 为 `null`。错误时 `error` 为一个 `Error` 实例，`err.code` 为该子进程的退出代码，`err.signal` 为使该进程结束的信号。
There is a second optional argument to specify several options. The default options are

可选的第二个参数用于指定一些选项。默认选项如下：

```
{ encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
```

```
  killSignal: 'SIGTERM',
  cwd: null,
  env: null }
```

If `timeout` is greater than 0, then it will kill the child process if it runs longer than `timeout` milliseconds. The child process is killed with `killSignal` (default: `'SIGTERM'`). `maxBuffer` specifies the largest amount of data allowed on stdout or stderr - if this value is exceeded then the child process is killed.
如果参数 `timeout` 的值超过 0，那么当运行超过 `timeout` 毫秒后子进程将终止。`killSignal` 为终止子进程的信号(默认为： `'SIGTERM'`)。 参数 `maxBuffer` 指定了 stdout 或 stderr 流最大数据量，一旦超过该值，子进程将会终止。

child.kill(signal='SIGTERM')

Send a signal to the child process. If no argument is given, the process will be sent `'SIGTERM'`. See `signal(7)` for a list of available signals.
给子进程发送信号。如果没有指定参数，（Node）将会发送`'SIGTERM'`信号。在 `signal(7)` 中可查阅到可用的信号列表。

```
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');
```

Note that while the function is called `kill`, the signal delivered to the child process may not actually kill it. `kill` really just sends a signal to a process.
注意：虽然函数名为 `kill`（杀死），发送的信号并不会真正杀死子进程。`kill` 仅仅是向该进程发送一个信号。
See `kill(2)`
参见 `kill(2)`

Assert 断言模块

This module is used for writing unit tests for your applications, you can access it with `require('assert')`.
断言（Assert）模块用于为应用编写单元测试，可以通过 `require('assert')` 对该模块进行调用。

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

Tests if `actual` is equal to `expected` using the operator provided.
使用指定操作符测试 `actual`（真实值）是否和 `expected`（期望值）一致。

assert.ok(value, [message])

Tests if value is a `true` value, it is equivalent to `assert.equal(true, value, message);`
测试实际值是否为 `true`，和 `assert.equal(true, value, message);` 作用一致

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

Tests shallow, coercive equality with the equal comparison operator (`==`).
使用等值比较操作符(`==`)测试真实值是否浅层地（shallow），强制性地（coercive）和预期值相等。

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

Tests shallow, coercive non-equality with the not equal comparison operator (`!=`).
使用不等比较操作符(`!=`)测试真实值是否浅层地（shallow），强制性地（coercive）和预期值不相等。

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

Tests for deep equality.

测试真实值是否深层次地和预期值相等。

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

Tests for any deep inequality.

测试真实值是否深层次地和预期值不相等。

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

Tests strict equality, as determined by the strict equality operator (`===`)
使用严格相等操作符 (`===`)测试真实值是否严格地（strict）和预期值相等。

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

Tests strict non-equality, as determined by the strict not equal operator (`!==`)
使用严格不相等操作符 (`!==`)测试真实值是否严格地（strict）和预期值不相等。

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

Expects `block` to throw an error. `error` can be constructor, regexp or validation function.
预期 `block` 时抛出一个错误（error）， `error` 可以为构造函数，正则表达式或者其他验证器。
Validate instanceof using constructor:

使用构造函数验证实例：

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

Validate error message using RegExp:

使用正则表达式验证错误信息：

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

Custom error validation:

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, [error], [message])

Expects `block` not to throw an error, see assert.throws for details.
预期 `block` 时不抛出错误，详细信息请见 assert.throws。

assert.ifError(value)

Tests if value is not a false value, throws if it is a true value. Useful when testing the first argument, `error` in callbacks.
测试值是否不为 false，当为 true 时抛出。常用于回调中第一个参数 `error` 的测试。 ## TTY 终端模块

Use `require('tty')` to access this module.
可使用 `require('tty')`访问此模块。
Example:

示例：

```
var tty = require('tty');
tty.setRawMode(true);
process.stdin.resume();
process.stdin.on('keypress', function(char, key) {
  if (key && key.ctrl && key.name == 'c') {
    console.log('graceful exit');
    process.exit()
  }
});
```

tty.open(path, args=[])

Spawns a new process with the executable pointed to by `path` as the session leader to a new pseudo terminal.
用 `path` 路径所指向的可执行文件启动一个新的进程，并将其作为一个新的伪终端的控制进程。
Returns an array `[slaveFD, childProcess]`. `slaveFD` is the file descriptor of the slave end of the pseudo terminal. `childProcess` is a child process object.
返回一个数组 `[slaveFD, childProcess]`。`slaveFD` 是这个伪终端的从设备文件描述符，`childProcess` 是子进程的对象。

tty.isatty(fd)

Returns `true` or `false` depending on if the `fd` is associated with a terminal.
当 `fd` 所表示的文件描述符与一个终端相关联时返回 `true`，否则返回 `false`。

tty.setRawMode(mode)

`mode` should be `true` or `false`. This sets the properties of the current process's stdin fd to act either as a raw device or default.
`mode` 参数可以设为 `true` 或 `false`。此方法设置当前进程的 stdin（标准输入）为原始设备方式，或默认方式。

tty.setWindowSize(fd, row, col)

`ioctl`s the window size settings to the file descriptor.
使用 `ioctl` 设置文件描述符对应的终端窗口大小（行数与列数）。

tty.getWindowSize(fd)

Returns `[row, col]` for the TTY associated with the file descriptor.
返回文件描述符所对应的终端的窗口大小`[row, col]`（行数与列数）。

os Module 操作系统模块

Use `require('os')` to access this module.
可以通过 `require('os')`访问这个 os 模块。

os.hostname()

Returns the hostname of the operating system.

该方法返回当前操作系统的主机名。

os.type()

Returns the operating system name.

该方法返回当前操作系统名称。

os.release()

Returns the operating system release.

返回当前操作系统的发型版本。

os.uptime()

Returns the system uptime in seconds.

该方法返回当前系统的正常运行时间，时间以秒为单位。

os.loadavg()

Returns an array containing the 1, 5, and 15 minute load averages.

该方法返回一个数组，该数组存储着系统 1 分钟，5 分钟，以及 15 分钟的负载均值。

os.totalmem()

Returns the total amount of system memory in bytes.

返回系统存储空间总值，该值以字节（byte）为单位。

os.freemem()

Returns the amount of free system memory in bytes.

返回系统存储的剩余空间，该值以字节（byte）为单位。

os.cpus()

Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of CPU ticks spent in: user, nice, sys, idle, and irq).

该方法返回一个对象数组，该数组包含了关于系统每个 CPU/内核的信息：型号，速度（以 MHz 为单位），以及 CPU 时间使用情况（包含 CPU 时间片在用户态、改变过优先级的用户进程、内核态、空闲、以及 IRQ 各方面的消耗）。

Example inspection of os.cpus:

os.cpus 以一个示例如下：

```
[ { 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 } } ]
```

Debugger 调试器

V8 comes with an extensive debugger which is accessible out-of-process via a simple TCP protocol. Node has a built-in client for this debugger. To use this, start Node with the `debug` argument; a prompt will appear:

V8 引擎自身配备了全面的调试器，该调试器通过简单的 TCP 协议在进程外访问。Node 中内置了该调试器的客户端，要使用它可以在启动 Node 时附加 `debug` 参数，此模式下将显示 debug 提示符：

```
% node debug myscript.js
debug>
```

At this point `myscript.js` is not yet running. To start the script, enter the command `run`. If everything works okay, the output should look like this:

此时 `myscript.js` 还没有开始执行，若要执行这段脚本，还需要输入 `run` 命令。如果一切运行正常的话输出信息应该如下所示：

```
% node debug myscript.js
```

```
debug> run
debugger listening on port 5858
connecting...ok
```

Node's debugger client doesn't support the full range of commands, but simple step and inspection is possible. By putting the statement `debugger;` into the source code of your script, you will enable a breakpoint.

Node 的调试器客户端虽然没有支持所有的命令，但是实现简单的单步调试还是可以的。你可以在脚本代码中声明 `debugger;`语句从而启用一个断点。

For example, suppose `myscript.js` looked like this:

例如，假设 `myscript.js` 代码如下：

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

Then once the debugger is run, it will break on line 4.

一旦调试器开始运行，那么它将在执行到第 4 行代码处时停止。

```
% ./node debug myscript.js
debug> run
debugger listening on port 5858
connecting...ok
hello
break in #<an Object>._onTimeout(), myscript.js:4
  debugger;
  ^
debug> next
break in #<an Object>._onTimeout(), myscript.js:5
```

```
  console.log("world");
  ^
debug> print x
5
debug> print 2+2
4
debug> next
world
break in #<an Object>._onTimeout() returning undefined,
myscript.js:6
}, 1000);
^
debug> quit
A debugging session is active. Quit anyway? (y or n) y
%
```

The `print` command allows you to evaluate variables. The `next` command steps over to the next line. There are a few other commands available and more to come type `help` to see others.
`print` 命令允许将变量输出到控制台进行查看。`next` 命令单步调试到下一行代码。还有其他一些可用的命令你可以通过输入 `help` 进行查看。

Advanced Usage 高级用法

The V8 debugger can be enabled and accessed either by starting Node with the `--debug` command-line flag or by signaling an existing Node process with `SIGUSR1`.
要启用 V8 引擎调试器你可以在启动 Node 时增加命令行参数`--debug` 或者给一个已经存在的 Node 进程发送值为 `SIGUSR1` 的信号量。

Appendixes 附录

Appendix 1 - Third Party Modules 附录 1 - 第三方模块

There are many third party modules for Node. At the time of writing, August 2010, the master repository of modules is the wiki page.

Node 中包含许多第三方模块。截至撰写时（2010 年 8 月），此 wiki 页是存放 Node 模块的主仓库。

This appendix is intended as a SMALL guide to new-comers to help them quickly find what are considered to be quality modules. It is not intended to be a complete list. There may be better more complete modules found elsewhere.

本附录的编写目的在于简要指导 Node 的新用户，帮助他们可以迅速找到公认的优秀 Node 模块。这里并非一份完整的列表，在其他地方也许可以找到其他更好更完善的模块。

- Module Installer: npm

  模块安装：npm

- HTTP Middleware: Connect

  HTTP 中间件（Middleware）：Connect

- Web Framework: Express

  Web 框架：Express

- Web Sockets: Socket.IO

- HTML Parsing: HTML5

  HTML 解析器：HTML5

- mDNS/Zeroconf/Bonjour

- RabbitMQ, AMQP

- mysql

- Serialization: msgpack

  序列化工具：msgpack

- Scraping: Apricot

抓取器：Apricot

- Debugger: ndb is a CLI debugger inspector is a web based tool.

  调试工具：ndb 是一个 CLI 调试工具 inspector 是一个基于 Web 的调试工具

- pcap binding

- ncurses

- Testing/TDD/BDD: vows, expresso, mjsunit.runner

  测试工具/TDD/BDD： vows, expresso, mjsunit.runner

Patches to this list are welcome.

欢迎大家对本列表进行补充。