'Hello World' と返答する Node で書かれたWebサーバの例:

```
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/');
```

このサーバを実行するには、コードを **example.js** というファイルに保存し、 node コマンドで実行してください。

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

このドキュメントの全てのサンプルは同じように実行することができます。

標準モジュール

Node にはプロセス内でコンパイルされるいくつかのモジュールが含まれていて、 そのほとんどは以下に文書化されています。これらのモジュールを使用するもっとも一般的な方法は、 **require('name')** の戻り値を、モジュールと同じ名前のローカル変数に割り当てることです。

例:

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

その他のモジュールで node を拡張することも可能です。「モジュール」を参照してください。

バッファ

純粋な JavaScript は Unicode と相性がいいものの、バイナリデータの扱いはうまくありません。 TCP ストリームやファイルシステムを扱う場合は、オクテットストリームを処理する必要があります。 Node にはオクテットストリームを操作、作成、消費するためにいくつかの戦略があります。

生のデータは **Buffer** クラスのインスタンスに保存されます。 **Buffer** は整数の配列と似ていますが、 V8 ヒープの外部に割り当てられた生のメモリに対応します。 **Buffer** のサイズを変更することはできません。

`Buffer` オブジェクトはグローバルです。

バッファを JavaScript 文字列オブジェクトとの間で変換するにはエンコーディング方式を明示する必要があります。 いくつかのエンコーディング方式があります。

- `'ascii'` - 7bit の ASCII データ専用です。 このエンコーディング方式はとても高速で、もし上位ビットがセットされていれば取り除かれます。
- `'utf8'` - Unicode文字。 多くのWebページやその他のドキュメントは UTF-8 を使っています。
- `'base64'` - Base64 文字列エンコーディング.
- `'binary'` - 生のバイナリデータを各文字の最初の 8bit として使用するエンコーディング方式。 このエンコーディング方式はもはや価値がなく、`Buffer` オブジェクトでは可能な限り使用すべきではありません。 このエンコーディングは、Node の将来のバージョンで削除される予定です。

new Buffer(size)

`size` オクテットの新しいバッファを割り当てます。

new Buffer(array)

オクテットの `array` を使用する新しいバッファを割り当てます。

new Buffer(str, encoding='utf8')

与えられた `str` を内容とする新しいバッファを割り当てます。

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

与えられたエンコーディングを使用して、 `string` をバッファの `offset` から書き込みます。 書き込まれたオクテット数を返します。 もし `buffer` が文字列全体を挿入するのに十分なスペースを含んでいなければ、文字列の一部だけを書き込みます。 `'utf8'` エンコーディングの場合、このメソッドは文字の一部だけを書き込むことはありません。

例: utf8 の文字列をバッファに書き込み、それをプリントします

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

// 12 bytes: ? + ? = ?
```

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

`encoding` でエンコードされたバッファデータの `start` から `end` までをデコードした文字列を返します。

上の **buffer.write()** の例を参照してください。

buffer[index]

index の位置のオクテットを取得および設定します。 その値は個々のバイトを参照するので、妥当な範囲は 16 進の **0x00** から **0xFF** または **0** から**255**までの間です。

例: ASCII 文字列を 1 バイトずつバッファにコピーします

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

文字列の実際のバイト数を返します。これは文字列の 文字数を返す **String.prototype.length** と同じではありません。

例:

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

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

// ? + ? = ?: 9 characters, 12 bytes
```

buffer.length

バイト数によるバッファのサイズ。これは実際の内容のサイズではないことに注意してください。 **length** はバッファオブジェクトに割り当てられたメモリ全体を参照します。

```
buf = new Buffer(1234);

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

// 1234
// 1234
```

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

バッファ間で memcpy() をします。

例: バッファを2個作成し、**buf1** の 16 バイト目から 19 バイト目を、 **buf2** の 8 バイト目から始まる位置へコピーします。

```
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)

元のバッファと同じメモリを参照しますが、**start** と **end** で示されるオフセットで始まり短くされた新しいバッファを返します。

新しいバッファスライスの変更は、オリジナルバッファのメモリを変更することになります！

例: ASCII のアルファベットでバッファを構築してスライスし、元のバッファで 1 バイトを変更します。

```
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
```

イベント生成

Node のオブジェクトの多くはイベントを生成します: TCP サーバはストリームがあるとそのたびにイベントを生成します。 子プロセスは終了する際にイベントを生成します。 イベントを生成する全てのオブジェクトは **events.EventEmitter** のインスタンスです。

イベントはキャメル記法による文字列で表現されます。 いくつかの例:

`'stream'`、`'data'`、`'messageBegin'`

関数をオブジェクトにアタッチすることができ、それはイベントが生成された時に実行されます。 これらの関数はリスナーと呼ばれます。

`require('events').EventEmitter` で `EventEmitter` クラスにアクセスします。

全ての EventEmitter は、新しいリスナーが加えられるとイベント `'newListener'` を生成します。

`EventEmitter` がエラーに遭遇した時、典型的な動作は `'error'` イベントを生成することです。 エラーイベントは特別です ー もしそのハンドラがなければ、スタックトレースが出力されてプログラムは終了します。

イベント: 'newListener'

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

このイベントは誰かが新しいリスナーを追加するといつでも生成されます。

イベント: 'error'

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

エラーに遭遇すると、このイベントが生成されます。 このイベントは特別です ー エラーを受け取るリスナーが存在しない場合、 Node は実行を終了して例外のスタックトレースを表示します。

emitter.on(event, listener)

指定されたイベントに対するリスナー配列の最後にリスナーを追加します。

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

emitter.removeListener(event, listener)

指定されたイベントに対するリスナー配列からリスナーを削除します。

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

emitter.removeAllListeners(event)

指定されたイベントに対するリスナー配列から全てのリスナーを削除します。

emitter.listeners(event)

指定されたイベントに対するリスナー配列を返します。 この配列は変更することができます、例えばリスナーを削除するなど。

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

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

それぞれのリスナーを引数で渡された順に実行します。

ストリーム

ストリームは Node の様々なオブジェクトで実装される抽象的なインタフェースです。 例えば HTTP サーバへのリクエストは標準出力と同様にストリームです。 ストリームは読み込み可能、書き込み可能、またはその両方です。 全てのストリームは `EventEmitter` のインスタンスです。

読み込み可能なストリーム

`Readable Stream` には以下のメソッド、メンバー、そしてイベントがあります。

イベント: 'data'

`function (data) { }`

`'data'` イベントは `Buffer` (デフォルト) または、 `setEncoding()` された場合は文字列のどちらかを生成します

イベント: 'end'

`function () { }`

ストリームが EOF (TCP 用語では FIN) を受信した時に生成されます。 `'data'` イベントがもう発生しないことを示します。 ストリームがもし書き込み可能でもあるなら、書き込みを続けることは可能かもしれません。

イベント: 'error'

`function (exception) { }`

データ受信でエラーがあると生成されます。

イベント: 'close'

`function () { }`

下層でファイル記述子がクローズされた時に生成されます。 全てのストリームがこのイベントを発生するわけではありません。 (例えば、インカミングの HTTP リクエス

トは `'close'` イベントを生成しません。)

イベント: 'fd'

```
function (fd) { }
```

ストリームに関するファイル記述子を受け取った時に生成されます。 UNIX ストリームだけがこの機能をサポートしています; その他の全てのストリームはこのイベントを生成しません。

stream.readable

デフォルトでは `true` ですが、`'error'` が発生した後、 ストリームが `'end'` に達した後、または `destroy()` が呼ばれた後で、 `false` に設定される boolean です。

stream.setEncoding(encoding)

data イベントが `Buffer` ではなく文字列を生成するようにします。 `encoding` には `'utf8'`、`'ascii'`、 または `'base64'` を指定することができます。

stream.pause()

`'data'` イベントの到着を中断します。

stream.resume()

`pause()` の後で `'data'` イベントの到着を再開します。

stream.destroy()

下層のファイル記述子をクローズします。ストリームはそれ以上イベントを生成しなくなります。

書き込み可能なストリーム

`Writable Stream` には以下のメソッド、メンバー、そしてイベントがあります。

イベント: 'drain'

```
function () { }
```

呼び出された `write()` メソッドが `false` で戻った後に生成され、 再び安全に書き込むことができるようになったことを示します。

イベント: 'error'

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

`exception`例外によるエラーについて生成されます。

イベント: 'close'

```
function () { }
```

下層でファイル記述子がクローズされた時に生成されます。

stream.writeable

デフォルトでは `true` ですが、`'error'` が発生した後、 `end()` / `destroy()` が呼ばれた後で `false` に設定される boolean です。

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

与えられた `encoding` で `string` を書き込みます。 文字列がカーネルバッファにフラッシュされた場合は `true` が返ります。 カーネルバッファがいっぱいの場合は、データが将来カーネルバッファに送られることを示すために、 `false` が返ります。 `'drain'` イベントがカーネルバッファが再び空いたことを示します。 `encoding` のデフォルトは `'utf8'` です。

オプションの `fd` 引数が指定されると、 ストリームに送信するための基礎となるファイル記述子として解釈されます。 これは UNIX ストリームでのみサポートされており、その他では黙って無視されます。 このようにファイル記述子に書き込む場合、ストリームが流れきる前にファイル記述子をクローズすると、 データが不正な (クローズされた) ファイル記述子に送られるリスクがあります。

stream.write(buffer)

生のバッファを使うこと以外は上記と同じです。

stream.end()

ストリームを EOF または FIN で終了します。

stream.end(string, encoding)

与えられた `encoding` で `string` を送信してからEOFまたはFINでストリームを終了します。 これは送信するパケットの数を減らすために便利です。

stream.end(buffer)

`buffer` であること以外は上記と同じです。

stream.destroy()

下層のファイル記述子をクローズします。ストリームはそれ以上イベントを生成しなくなります。

グローバルオブジェクト

これらのオブジェクトはグローバルなスコープで有効であり、どこからでもアクセスすることができます。

global

グローバルなネームスペースのオブジェクトです。

process

プロセスオブジェクトです。 `'process object'` の節を参照してください。

require()

要求されたモジュールを指します。 'Modules' の節を参照してください。

require.paths

require() のためのサーチパスの配列です。 この配列はカスタムパスを追加するために変更することができます。

例: サーチリストの先頭に新しいパスを追加する

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

__filename

実行されているスクリプトのファイル名です。これは絶対パスであり、 必ずしもコマンドライン引数で渡されたファイル名と同じというわけではありません。

例: node example.js を /Users/mjr で実行する

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

__dirname

スクリプトが実行されているディレクトリ名です。

例: node example.js を /Users/mjr で実行する

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

module

現在の (process.Module 型である) モジュールへの参照です。 特に module.exports は exports オブジェクトと同じです。 より詳しくは src/process.js を参照してください。

process

process はグローバルオブジェクトで、どこからでもアクセスすることができます。 それは EventEmitter のインスタンスです。

イベント: 'exit'

function () {}

プロセスが終了しようとしている時に生成されます。 これは (ユニットテストのように) モジュールの状態を一定の時間でチェックするのに適したフックとなります。 メインのイベントループは 'exit' コールバックが終了するともはや動作しないので、 タイマーはスケジュールされないかもしれません。

exit を監視する例:

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

イベント: 'uncaughtException'

function (err) { }

発生した例外がイベントループまでたどり着いた場合に生成されます。 もしこの例外に対するリスナーが加えられていれば、 デフォルトの動作 (それはスタックトレースをプリントして終了します) は起こりません。

uncaughtException を監視する例:

```
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.');
```

uncaughtException は例外を扱うとても荒削りなメカニズムであることに注意してください。 プログラムの中で try / catch を使えばもっとプログラムの流れをうまく制御できるでしょう。 特にサーバプログラムはいつまでも実行し続けるように設計されるので、 **uncaughtException** は有益で安全なメカニズムになり得ます。

シグナルイベント

function () {}

プロセスがシグナルを受信た場合に生成されます。 SIGINT、SIGUSR1、その他の POSIX 標準シグナル名の一覧について はsigaction(2) を参照してください。

SIGINTを監視する例:

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

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

多くの端末プログラムで簡単に **SIGINT** を送る方法は **Control-C** を押すことです。

process.stdout

stdout に対する **Writable Stream** です。

例: **console.log** の定義

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

process.openStdin()

標準入力ストリームをオープンし、 **Readable Stream** を返します。

標準入力をオープンして二つのイベントを監視する例:

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

stdin.setEncoding('utf8');

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

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

process.argv

コマンドライン引数を含む配列です。 最初の要素は 'node'、2 番目の要素は JavaScript ファイルの名前になります。 その後の要素はコマンドラインの追加の引数になります。

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

このように出力されます:

```
$ 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

プロセスによって開始された実行可能ファイルの絶対パスです。

例:

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

process.chdir(directory)

プロセスのカレントワーキングディレクトリを変更します。 もし失敗した場合は例外をスローします。

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

process.compile(code, filename)

eval と似ていますが、よりよいエラー報告のために **filename** を指定することができ、 **code** はローカルスコープから不可視です。 **filename** の値は、もしコンパイルされたコードによってスタックトレースが生成されると、 ファイル名として使われます。

process.compile と **eval** で同じコードを使った例:

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

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

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

process.compile はローカルスコープにアクセスしないので、 **localVar** は変更されません。 **eval** はローカルスコープにアクセスするので、 **localVar** は変更されます。

code が文法エラーとなるケースでは、**process.compile** は node を終了します。

関連項目: **Script**

process.cwd()

プロセスのカレントワーキングディレクトリを返します。

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

process.env

ユーザの環境を含むオブジェクトです。environ(7) を参照してください。

process.exit(code=0)

指定の `code` でプロセスを終了します。 もし省略されると、「成功」を示すコード `0` を使って終了します。

「失敗」を示すコードで終了する例:

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

node を実行したシェルで終了コードが 1 であることを見ることができるでしょう。

process.getgid()

プロセスのグループ識別子を取得します (getgid(2) 参照)。 これは数値によるグループ ID で、グループ名ではありません。

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

process.setgid(id)

プロセスのグループ識別子を設定します (setgid(2) 参照)。 これは数値による ID もグループ名の文字列のどちらも受け入れます。 もしグループ名が指定されると、数値による ID が解決できるまでこのメソッドはブロックします。

```
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()

プロセスのユーザ識別子を取得します (getuid(2) 参照)。 これは数値によるユーザ ID で、ユーザ名ではありません。

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

process.setuid(id)

プロセスのユーザ識別子を設定します (setuid(2) 参照)。 これは数値による ID もユーザ名の文字列のどちらも受け入れます。 もしユーザ名が指定されると、数値による ID

が解決できるまでこのメソッドはブロックします。

```
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

NODE_VERSION を提示するコンパイル済みプロパティです。

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

process.installPrefix

NODE_PREFIX を提示するコンパイル済みプロパティです。

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

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

プロセスにシグナルを送ります。 **pid** はプロセス ID で **signal** は送信されるシグナルを文字列で記述したものです。 シグナルの名前は 'SIGINT' や 'SIGUSR1' のような文字列です。 省略すると、シグナルは 'SIGINT' となります。 詳細は kill(2) を参照してください。

この関数の名前が **process.kill** であるとおり、 これは **kill** システムコールのように本当にシグナルを送信することに注意してください。 対象のプロセスを殺すだけでなく、他のシグナルも送信されます。

自身にシグナルを送信する例:

```
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

プロセスの PID です。

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

process.title

'ps' でどのよう表示されるかを設定するための getter/setter です。

process.platform

どのプラットフォームで動いているかです。 **'linux2'**、**'darwin'**、など。

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

process.memoryUsage()

Node プロセスのメモリ使用状況を記述したオブジェクトを返します。

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

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

このように生成されます:

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

heapTotal と **heapUsed** は V8 のメモリ使用状況を参照します。

process.nextTick(callback)

イベントループの次以降のループでコールバックを呼び出します。 これは **setTimeout(fn, 0)** の単純なエイリアスではなく、 はるかに効率的です。

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

process.umask([mask])

プロセスのファイルモード作成マスクを設定または読み込みます。 子プロセスは親プロセスからマスクを継承します。 **mask** 引数が与えられると元のマスクが返され、そうでなければ現在のマスクが返されます。

```
var oldmask, newmask = 0644;

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

sys

これらの関数はモジュール `'sys'` 内にあります。 `require('sys')` を使うことでこれらにアクセスします。

sys.print(string)

`console.log()` と似ていますが、後に改行を続けません。

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

sys.debug(string)

同期的な出力関数です。プロセスをブロックして即座に `string` を `stderr` に出力します。

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

sys.log(string)

タイムスタンプとともに `stdout` へ出力します。

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

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

デバッグで有用な、`object` の文字列表現を返します。

`showHidden` が `true` の場合、 オブジェクトの Enumerable でないプロパティも表示されます。

`depth` が与えられた場合、 オブジェクトをフォーマットするために何回再帰するかを `inspect` に伝えます。 これは巨大で複雑なオブジェクトを調査する場合に便利です。

デフォルトでは 2 回だけ再帰します。 無限に再帰するには、`depth` に `null` を渡します。

`sys` オブジェクトの全てのプロパティを調査する例:

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

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

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

実験的

`readableStream` からデータを読み、それ を`writableStream` に送ります。 `writeableStream.write(data)` が `false` を返す場合、 `writableStream` が `drain` イベントを生成するまで、 `readableStream` は中断します。 `writableStream` がクローズされるかエラーが発生すると、`callback` は error を唯一の引数として呼び出されます。

Timers

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

`delay` ミリ秒が経過した後で `callback` が実行されるようにスケジュールします。 `clearTimeout()` で使うことができる `timeoutId` を返します。 オプションとして、コールバックへの引数を渡すこともできます。

clearTimeout(timeoutId)

タイムアウトがトリガーされるのを止めます。

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

`delay` ミリ秒が経過するごとに繰り返し `callback` が実行されるようにスケジュールします。 `clearInterval()` で使うことができる `intervalId` を返します。 オプションとして、コールバックへの引数を渡すこともできます。

clearInterval(intervalId)

インターバルがトリガーされるのを止めます。

子プロセス

Nodeは `ChildProcess` クラスを通じて 3 方向の `popen(3)` 機能を提供します。

それは完全にノンブロッキングな方法で子プロセスの `stdin`、 `stdout`、 そして `stderr` を通じてデータを流すことを可能にします。

子プロセスの生成は `require('child_process').spawn()` を使います。

子プロセスは常に 3 本のストリームと関連づけられています。 `child.stdin`、 `child.stdout`、そして `child.stderr` です。

`ChildProcess` は `EventEmitter` です。

イベント: 'exit'

```
function (code, signal) {}
```

このイベントは子プロセスが終了した後で生成されます。 プロセスが普通に終了した場合、 `code` はプロセスの終了コードです。 それ以外の場合は `null` です。 プロセスがシグナルを受け取って終了した場合、 `signal` は文字列によるシグナルの名前です。 それ以外の場合は `null` です。

このイベントが生成された後は、 `'output'` および `'error'` コールバックはもう呼ばれなくなります。

`waitpid(2)` を参照してください。

child.stdin

子プロセスの **stdin** を表現する **Writable Stream** です。 多くの場合、**end()** を通じてこのストリームを閉じると子プロセスが終了する原因となります。

child.stdout

子プロセスの **stdout** を表現する **Readable Stream** です。

child.stderr

子プロセスの **stderr** を表現する **Readable Stream** です。

child.pid

子プロセスの PID です。

例:

```
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])

args をコマンドライン引数として、与えられた **command** で新しいプロセスを起動します。 **args** が省略された場合、空の配列がデフォルトです。

第 3 引数は追加のオプションを指定するために使われ、そのデフォルトは:

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

cwd で起動されたプロセスのワーキングディレクトリを指定することができます。 **env** は新しいプロセスに見える環境変数を指定するために使います。 **customFds** は新しいプロセスの [stdin, stout, stderr] を既存のストリームに接続することを可能にします; **-1** は新しいストリームが作られなければならないことを意味します。

ls -lh /usr を実行して **stdout**、**stderr**、および終了コードを取得する例:

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

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

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

```
});

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

とても手の込んだ方法で実行する 'ps ax | grep ssh' の例:

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

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

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

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

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

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

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

exec の失敗をチェックする例:

```
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.');
  }
```

```
});
```

関連項目: **child_process.exec()**

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

コマンドを子プロセスとして実行し、その出力を蓄えて、その全てをコールバックに渡す高水準の方法です。

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

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

コールバックは引数 **(error, stdout, stderr)** を得ます。 成功すると、**error** は **null** になります。 エラーだと、**error** は **Error** のインスタンスとなり、 **err.code** は子プロセスの終了コード、 **err.signal** はプロセスを終了させたシグナルとなります。

任意の第 2 引数でいくつかのオプションを指定することができます。 オプションのデフォルトは

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

もし **timeout** が 0 より大きいと、 子プロセスは実行時間が **timeout** ミリ秒よりも長くなると kill されます。 子プロセスは **killSignal** で kill されます (デフォルト: **'SIGKILL'**)。 **maxBuffer** は標準出力と標準エラーの最大のデータ量を指定します ー この値を超えると子プロセスは kill されます。

child.kill(signal='SIGTERM')

子プロセスにシグナルを送ります。 引数が与えられない場合、子プロセスには **'SIGTERM'** が送られます。 利用可能なシグナルの一覧は **signal(7)** を参照してください。

```
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');
```

この関数は **kill** と呼ばれるものの、 子プロセスに届けられるシグナルが実際には子プロセスを殺さないかもしれないことに注意してください。 **kill** はただプロセスにシグナルを送るだけです。

kill(2) を参照してください。

Script

Script は JavaScript コードをコンパイルおよび実行するクラスです。 このクラスには次のようにアクセスできます:

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

新しい JavaScript コードは、コンパイルされてすぐに実行されるか、コンパイルおよび保存されて後から実行されます。

Script.runInThisContext(code, [filename])

process.compile と同様です。 **Script.runInThisContext** は **code** を **filename** からロードされたかのようにコンパイルし、 それを実行して結果を返します。実行されるコードはローカルスコープにアクセスしません。 **filename** はオプションです。

Script.runInThisContext と **eval** で同じコードを実行する例:

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

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

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

Script.runInThisContext はローカルスコープにアクセスしないので、 **localVar** は変更されません。 **eval** はローカルスコープにアクセスするので、 **localVar** は変更され

ます。

code が文法エラーとなるケースでは、 **Script.runInThisContext** は標準エラーに文法エラーを出力し、例外をスローします。

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

Script.runInNewContext は **code** を **filename** からロードされたかのようにコンパイルし、 それを **sandbox** の中で実行して結果を返します。 実行されるコードはローカルスコープにアクセスせず、 **sandbox** が **code** にとってのグローバルオブジェクトとして使われます。 **sandbox** および **filename** はオプションです。

例: グローバル変数をインクリメントして新しい値をセットするコードをコンパイルおよび実行します。

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

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

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

慎重を要するビジネスでは、信頼できないコードの実行は細心の注意が求められることに注意してください。 偶然グローバル変数が漏れてしまうことを防ぐために、 **Script.runInNewContext** はとても役立ちますが、 信頼できないコードを安全に実行するために別のプロセスを要求します。

code が文法エラーとなるケースでは、 **Script.runInNewContext** は標準エラーに文法エラーを出力し、例外をスローします。

new Script(code, [filename])

new Script は **code** を **filename** からロードされたかのようにコンパイルしますが、実行はしません。 代わりに、コンパイルされたコードを表現する **Script** オブジェクトを返します。 このスクリプトは後述のメソッドを使って後から何度でも実行することができます。 返されるスクリプトはどのグローバルオブジェクトとも結びつけられていません。 それぞれの実行前に結びつけることで、そのとおりに実行されます。 **filename** はオプションです。

code が文法エラーとなるケースでは、 **new Script** は標準エラーに文法エラーを出力し、例外をスローします。

script.runInThisContext()

Script.runInThisContext (大文字の'S'に注意) と同様ですが、 こちらはコンパイル済みのスクリプトオブジェクトのメソッドです。 **script.runInThisContext** は **script** のコードを実行してその結果を返します。 実行されるコードはローカルスコープにアクセスしませんが、 **global** オブジェクト (v8: 実際のコンテキスト) にはアクセスします。

script.runInThisContext を使ってコードを一度だけコンパイルし、複数回実行する例:

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

globalVar = 0;

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

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

console.log(globalVar);

// 1000
```

script.runInNewContext([sandbox])

Script.runInNewContext (大文字の'S'に注意) と同様ですが、 こちらはコンパイル済みのスクリプトオブジェクトのメソッドです。 **script.runInNewContext** は **sandbox** がグローバルオブジェクトであるかのように **script** のコードを実行してその結果を返します。 実行されるコードはローカルスコープにアクセスしません。 **sandbox** はオプションです。

例: グローバル変数をインクリメントしてセットするコードをコンパイルして、このコードを複数回実行します。 これらのグローバル変数はサンドボックスに含まれます。

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

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

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

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

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

慎重を要するビジネスでは、信頼できないコードの実行は細心の注意が求められることに注意してください。 偶然グローバル変数が漏れてしまうことを防ぐために、**script.runInNewContext** はとても役立ちますが、 信頼できないコードを安全に実行するために別のプロセスを要求します。

ファイルシステム

File I/O は POSIX 標準の関数に対する単純なラッパーとして提供されます。 このモジュールを使用するには **require('fs')** してください。 全てのメソッドは非同期と同期の形式があります。

非同期の形式は常に最後の引数として完了コールバックを受け取ります。 引数として渡される完了コールバックはメソッドに依存しますが、 最初の引数は常に例外のために予約されています。 操作が成功で完了すると最初の引数は **null** または **undefined** となります

非同期バージョンの例です:

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

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

同期バージョンです:

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

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

非同期メソッドでは順序の保証はありません。 以下のような傾向のエラーがあります。

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

`fs.stat` は `fs.rename` より先に実行される可能性がありrます。 正しい方法はコールバックをチェーンすることです。

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

忙しいプロセスでは、プログラマはこれらの非同期バージョンを使うことが強く推奨されます。 同期バージョンはそれが完了するまでプロセス全体をブロックします ー全ての接続を停止します。

fs.rename(path1, path2, [callback])
非同期の rename(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.renameSync(path1, path2)
同期の rename(2)。

fs.truncate(fd, len, [callback])
非同期の ftruncate(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.truncateSync(fd, len)
同期の ftruncate(2)。

fs.chmod(path, mode, [callback])
非同期の chmod(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.chmodSync(path, mode)
同期の chmod(2)。

fs.stat(path, [callback])
非同期の stat(2)。コールバックは 2 つの引数を受け取る `(err, stats)` で、 `stats` は `fs.Stats` オブジェクトです。

```
{ 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'
}
```

より詳しくは後述の `fs.Stats` の節を参照してください。

fs.lstat(path, [callback])
非同期の lstat(2)。コールバックは 2 つの引数を受け取る `(err, stats)` で、 `stats` は `fs.Stats` オブジェクトです。

fs.fstat(fd, [callback])
非同期の fstat(2)。コールバックは 2 つの引数を受け取る `(err, stats)` で、 `stats` は `fs.Stats` オブジェクトです。

fs.statSync(path)
同期の stat(2)。`fs.Stats` のインスタンスを返します。

fs.lstatSync(path)
同期の lstat(2)。`fs.Stats` のインスタンスを返します。

fs.fstatSync(fd)
同期の fstat(2)。`fs.Stats` のインスタンスを返します。

fs.link(srcpath, dstpath, [callback])
非同期の link(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.linkSync(dstpath, srcpath)
同期の link(2)。

fs.symlink(linkdata, path, [callback])
非同期の symlink(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.symlinkSync(linkdata, path)
同期の symlink(2)。

fs.readlink(path, [callback])
非同期の readlink(2)。コールバックは 2 つの引数を受け取る `(err, resolvedPath)` です。

fs.readlinkSync(path)

同期の readlink(2)。解決されたパスを返します。

fs.realpath(path, [callback])
非同期の realpath(2)。コールバックは 2 つの引数を受け取る `(err, resolvedPath)` です。

fs.realpathSync(path)
同期の realpath(2)。解決されたパスを返します。

fs.unlink(path, [callback])
非同期の unlink(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.unlinkSync(path)
同期の unlink(2)。

fs.rmdir(path, [callback])
非同期の rmdir(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.rmdirSync(path)
同期の rmdir(2)。

fs.mkdir(path, mode, [callback])
非同期の mkdir(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.mkdirSync(path, mode)
同期の mkdir(2)。

fs.readdir(path, [callback])
非同期の readdir(3)。ディレクトリの内容を読み込みます。 コールバックは 2 つの引数を受け取る `(err, files)` で、 `files` は `'.'` と `'..'` を除くディレクトリ内のファイル名の配列です。

fs.readdirSync(path)
同期の readdir(3)。`'.'` と `'..'` を除くディレクトリ内のファイル名の配列を返します。

fs.close(fd, [callback])
非同期の close(2)。完了コールバックには発生し得る例外以外に引数が渡されることはありません。

fs.closeSync(fd)
同期の close(2)。

fs.open(path, flags, mode=0666, [callback])
非同期のファイルオープン。open(2) を参照してください。 フラグは 'r'、'r+'、'w'、'w+'、'a'、あるいは 'a+' です。 コールバックは 2 つの引数を受け取る `(err, fd)` です。

fs.openSync(path, flags, mode=0666)
同期の open(2)。

fs.write(fd, buffer, offset, length, position, [callback])
`fd` で指定されたファイルに `buffer` を書き込みます。

`offset` と `length` でバッファのどの部分が書き込まれるかが決まります。

`position` はデータが書き込まれる位置をファイルの先頭からのオフセットで示します。 `position` が `null` の場合、データは現在の位置から書き込まれます。 pwrite(2) を参照してください。

コールバックは 2 つの引数が与えられる `(err, written)` で、 `written` は書き込まれたバイト数を示します。

fs.writeSync(fd, buffer, offset, length, position)
同期版のバッファベース `fs.write()`。書き込まれたバイト数を返します。

fs.writeSync(fd, str, position, encoding='utf8')
同期版の文字列ベース `fs.write()`。書き込まれたバイト数を返します。

fs.read(fd, buffer, offset, length, position, [callback])
`fd` で指定されたファイルからデータを読み込みます。

`buffer` はデータが書き込まれるバッファです。

`offset` は書き込みを開始するバッファ内のオフセットです。

`length` は読み込むバイト数を指定する整数です。

`position` はファイルの読み込みを開始する位置を指定する整数です。 `position` が `null` の場合、データは現在の位置から読み込まれます。

コールバックは2つの引数が与えられる `(err, bytesRead)` です。

fs.readSync(fd, buffer, offset, length, position)
同期版のバッファベース `fs.read`。`bytesRead` の数を返します。

fs.readSync(fd, length, position, encoding)
同期版の文字列ベース `fs.read`。`bytesRead` の数を返します。

fs.readFile(filename, [encoding], [callback])
ファイル全体の内容を非同期に読み込みます。例:

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

コールバックは 2 つの引数が渡される `(err, data)` で、`data` はファイルの内容です。

エンコーディングが指定されなければ、生のバッファが渡されます。

fs.readFileSync(filename, [encoding])

同期版の `fs.readFile`。`filename` の内容を返します。

`encoding` が指定されるとこの関数は文字列を返します。 そうでなければバッファを返します。

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

非同期にデータをファイルに書き込みます。`data` は文字列またはバッファです。

例:

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

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

同期版の `fs.writeFile`。

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

`filename` の変更を監視します。コールバックの `listener` はファイルが変更される度に呼び出されます。

2 番目の引数はオプションです。`options` が与えられるなら、 オブジェクトは boolean の 2 つのメンバ、`persistent` と `interval` (ポーリング間隔のミリ秒) を持つべきです。 デフォルトは `{persistent: true, interval: 0}` です。

`listener` は現在の状態オブジェクトと前の状態オブジェクトの 2 つの引数を受け取ります:

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

これらの状態オブジェクトは `fs.Stat` のインスタンスです。

fs.unwatchFile(filename)

`filename` の変更に対する監視を終了します。

fs.Stats

`fs.stat()` と `fs.lstat()` から返されるオブジェクトはこの型です。

- `stats.isFile()`
- `stats.isDirectory()`
- `stats.isBlockDevice()`
- `stats.isCharacterDevice()`
- `stats.isSymbolicLink()` (`fs.lstat()`でのみ有効)
- `stats.isFIFO()`
- `stats.isSocket()`

fs.ReadStream

`ReadStream` は `Readable Stream` です。

fs.createReadStream(path, [options])

新しい ReadStream オブジェクトを返します (`Readable Stream` を参照してください)。

`options` は以下のデフォルト値を持つオブジェクトです:

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

ファイル全体を読み込む代わりに一部の範囲を読み込むため、 `options` に `start` および `end` を含めることができます。 `start` と `end` はどちらも包含的で0から始まります。 使う際にはいつでも、両方を同時に指定しなければなりません。

100 バイトの長さを持つファイルの最後の 10 バイトを読み込む例:

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

fs.WriteStream

`WriteStream` は `Writable Stream` です。

イベント: 'open'

`function (fd) { }`

`fd` は WriteStream に使われているファイル記述子です。

fs.createWriteStream(path, [options])

新しい WriteStream オブジェクトを返します (`Writable Stream` を参照してください)。

`options` は以下のデフォルト値を持つオブジェクトです:

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

HTTP

HTTP サーバおよびクライアントを使用するにはいずれも `require('http')` が必要です。

Node の HTTP インタフェースは、 伝統的に扱いが難しかったプロトコルの多くの機能をサポートするように設計されています。 とりわけ大きくて、場合によってはチャンク化されたメッセージです。 インタフェースは決してリクエストまたはレスポンス全体をバッファリングしないように気をつけています ー 利用者はストリームデータを使うことができます。

HTTP メッセージヘッダはこのようなオブジェクトとして表現されます:

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

キーは小文字化されます。値は変更されません。

考えられる HTTP アプリケーションを完全にサポートするために、 Node の HTTP API はとても低水準です。それはストリームのハンドリングとメッセージの解析だけに対処します。 解析はメッセージをヘッダとボディに分けますが、実際のヘッダとボディは解析しません。

プラットフォームで OpenSSL が利用可能であれば HTTPS がサポートされます。

http.Server

これは以下のイベントを持つ `EventEmitter` です:

イベント: 'request'

`function (request, response) { }`

`request` は `http.ServerRequest` のインスタンス、 `response` は `http.ServerResponse`

のインスタンスです。

イベント: 'connection'

```
function (stream) { }
```

新しい TCP ストリームが確立した時。 `stream` は `net.Stream` 型のオブジェクトです。通常の利用者がこのイベントにアクセスしたくなることはないでしょう。 `stream` は `request.connection` からアクセスすることもできます。

イベント: 'close'

```
function (errno) { }
```

サーバがクローズした時に生成されます。

イベント: 'request'

```
function (request, response) {}
```

リクエストの度に生成されます。コネクションごとに複数のリクエストがあることに注意してください (Keep Alive なコネクションの場合)。

イベント: 'upgrade'

```
function (request, socket, head)
```

クライアントが HTTP のアップグレードを要求する度に生成されます。 このイベントが監視されない場合、アップグレードを要求したクライアントのコネクションはクローズされます。

- `request` はリクエストイベントと同様に HTTP リクエストへの引数です。
- `socket` はサーバとクライアントの間のネットワークソケットです。
- `head` はアップグレードストリームの最初のパケットを持つ Buffer のインスタンスです。 空の場合もあります。

このイベントが生成された後、リクエスト元のソケットはもう `data` イベントリスナーを持ちません。 このソケットでサーバへ送られたデータを扱うためにそれをバインドしなければならないことを意味します。

イベント: 'clientError'

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

クライアントコネクションが 'error' イベントを発した場合 ー ここに転送されます。

http.createServer(requestListener)

新しい Web サーバオブジェクトを返します。

`requestListener` は自動的に `'request'` イベントに加えられる関数です。

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

指定されたポートとホスト名でコネクションの受け入れを開始します。 ホスト名が省略されると、サーバはどんな IPv4 アドレスへの接続も受け入れます (`INADDR_ANY`)。

UNIX ドメインソケットを待ち受ける場合、ポートとホスト名ではなくファイル名を提供します。

この関数は非同期です。最後の引数の `callback` はサーバがポートをバインドすると呼び出されます。

server.listen(path, [callback])

`path` で与えられたコネクションを待ち受ける UNIX ドメインソケットのサーバを開始します。

この関数は非同期です。最後の引数の `callback` はサーバがバインドすると呼び出されます。

server.setSecure(credentials)

秘密鍵とサーバ証明書を指定した暗号モジュールの認証情報で、サーバに対して HTTPS サポートを有効にします。 オプションで認証局で証明されたクライアント認証を使うこともできます。

認証情報が一つ以上の認証局の証明書を持っている場合、 サーバは HTTPS コネクションにおけるハンドシェークの一部としてクライアント証明書を送るようクライアントに要求します。 その有効性と内容は、サーバの `resuest.connection` から `verifyPeer()` と `getPeerCertificate()` を通じてアクセスできます。

server.close()

サーバが新しいコネクションを受け付けるのを終了します。

http.ServerRequest

このオブジェクトは HTTP サーバ内部 − ユーザではなく − で作成され、 `'request'` リスナーの第1引数として渡されます。

これは以下のイベントを持つ `EventEmitter` です:

イベント: 'data'

```
function (chunk) { }
```

メッセージボディの断片を受信した場合に生成されます。

例: 一つの引数としてボディのチャンクが与えられます。 転送エンコーディングでデコードされます。 ボディのチャンクは文字列です。 ボディのエンコーディングは `request.setBodyEncoding()` で設定されます。

イベント: 'end'

```
function () { }
```

メッセージごとに厳密に一回生成されます。引数はありません。 このイベントが生成された後、このリクエストで生成されるイベントはありません。

request.method

リクエストメソッドを表す文字列です。参照のみ可能です。 例: **'GET'**、**'DELETE'**

request.url

リクエスト URL を表す文字列です。 これは実際の HTTP リクエストに存在する URL だけを含みます。

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

この場合の **request.url** はこうなります:

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

URL の要素を解析したい場合は、 **require('url').parse(request.url)** を参照してください。例:

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

問い合わせ文字列からパラメータを取り出したい場合は、 **require('querystring').parse** 関数を参照するか、 **require('url').parse**の第 2 引数に **true** を渡してください。例:

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

request.headers

参照のみ可能です。

request.httpVersion

HTTP プロトコルのバージョンを表す文字列です。参照のみ可能です。例: **'1.1'**、**'1.0'**。 同様に **request.httpVersionMajor** は最初の整数、**request.httpVersionMinor** は 2 番目の整数です。

request.setEncoding(encoding=null)

リクエストボディのエンコーディングを設定します。 `'utf8'` または `'binary'` のいずれかです。 デフォルトは `null` で、 `'data'` イベントが `Buffer` を生成することを意味します。

request.pause()

リクエストによるイベントの生成を中断します。アップロード速度を落とすのに便利です。

request.resume()

中断されたリクエストを再開します。

request.connection

コネクションに関連づけられた `net.Stream` オブジェクトです。

HTTPS では `request.connection.verifyPeer()` と `request.connection.getPeerCertificate()` で クライアントの認証の詳細を取得できます。

http.ServerResponse

このオブジェクトは HTTP サーバ内部 − ユーザではなく − で作成されます。 `'request'` リスナーの第 2 引数として渡されます。 これは `Writable Stream` です。

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

レスポンスヘッダを送信します。 ステータスコードは `404` のような 3 桁の数字による HTTP ステータスコードです。 最後の引数 `headers` は、レスポンスヘッダです。 オプションとして人に読める形式の `reasonPhrase` を第 2 引数で与えることができます。

例:

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

このメソッドはメッセージごとに 1 回だけ呼び出されなくてはならず、 `response.end()` の前に呼び出されなければなりません。

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

このメソッドは `writeHead` の後に呼び出されなければなりません。 これはレスポンスボディのチャンクを送信します。 このメソッドはボディの連続した部分を提供するために複数回呼び出されるかもしれません。

`chunk` は文字列またはバッファにすることができます。 `chunk` が文字列の場合、どのエンコードでバイトストリームにするかを第 2 引数で指定します。 デフォルトの `encoding` は `'utf8'` です。

注意: これは生の HTTP ボディで、 高水準のマルチパートボディエンコーディングで使われるものとは無関係です。

初めて `response.write()` が呼び出されると、 バッファリングされていたヘッダ情報と最初のボディがクライアントに送信されます。 2 回目に `response.write()` が呼ばれると、 Node はストリーミングデータを分割して送信しようとしていると仮定します。 すなわち、レスポンスはボディの最初のチャンクまでバッファリングされます。

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

このメソッドはレスポンスの全てのヘッダとボディを送信したことをサーバに伝えます; サーバはメッセージが終了したと考えるべきです。 この `response.end()` メソッドは各レスポンスごとに呼び出さなければなりません。

`data` が指定された場合、 `response.write(data, encoding)` に続けて `response.end()` を呼び出すのと等価です。

http.Client

HTTP クライアントは引数として渡されるサーバアドレスによって構築され、 戻り値のハンドルは一つまたはそれ以上のリクエストを発行するのに使われます。 接続されたサーバに応じて、クライアントはパイプライン化されたリクエストまたは、 それぞれのストリームの後でストリームを再確立するかもしれません。 現在の実装はリクエストをパイプライン化しません。

`google.com` に接続する例:

```
var http = require('http');
var google = http.createClient(80, 'www.google.com');
var request = google.request('GET', '/',
  {'host': 'www.google.com'});
request.end();
request.on('response', function (response) {
  console.log('STATUS: ' + response.statusCode);
  console.log('HEADERS: ' + JSON.stringify(response.headers));
  response.setEncoding('utf8');
  response.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});
```

少数の特別なヘッダがあることに注意してください。

- 'Host' ヘッダは Node によって加えられませんが、通常の Web サイトに必要とされます。

- 'Connection: keep-alive' を送信することで、サーバへの接続を次のリクエストまで維持することを Node に通知します。

- 'Content-length' ヘッダを送信することで、デフォルトのチャンクエンコーディングが無効になります。

イベント: 'upgrade'

```
function (request, socket, head)
```

サーバがアップグレード要求に応答する度に生成されます。 このイベントが監視されていない場合、クライアントがアップグレードヘッダを受信するとそのコネクションはクローズされます。

より詳しくは `http.Server` の `upgrade` イベントの説明を参照してください。

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

新しい HTTP クライアントを構築します。 `port` と `host` は接続先となるサーバを参照します。 リクエストが発行されるまでストリームは確立されません。

オプションの`secure` は boolean のフラグで HTTPS サポートを有効にし、 オプションの`credentials` は暗号モジュールの認証情報オブジェクトで、 クライアントの秘密鍵、証明書、そして信頼できる認証局の証明書のリストを含むことができます。

コネクションがセキュアな場合、証明情報で認証局の証明書が明示的に渡されないと、 node.js はデフォルトの信頼できる認証局のリストとして http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt を与えます。

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

リクエストを発行します; 必要であればストリームを確立します。 `http.ClientRequest` のインスタンスを返します。

`method` はオプションで、省略された場合のデフォルトは 'GET'です。

`request_headers` はオプションです。 Node 内部で付加的なリクエストヘッダが加えられることがあります。 `ClientRequest` オブジェクトを返します。

ボディを送信しようとしている場合は、 `Content-Length` ヘッダを含めることを忘れないでください。 ボディをストリーム化する場合は、おそらく `Transfer-Encoding: chunked` をセットしてください。

注意: リクエストは完了していません。このメソッドはリクエストのヘッダを送信するだけです。 リクエストを完了してレスポンスを読み出すには `request.end()` を呼ぶ必

要があります。(複雑に感じるかもしれませんが、これは **request.write()** でボディをストリーム化するチャンスをユーザに提供します))。

client.verifyPeer()
指定された、あるいはデフォルトの信頼された認証局の証明書において、サーバの証明書の妥当性に応じて true または false を返します。

client.getPeerCertificate()
サーバ証明書の詳細を、'subject'、'issuer'、'valid_from' そして 'valid_to' をキーとする証明書の辞書を含む JSON 形式で返します。

http.ClientRequest

このオブジェクトは HTTP サーバ内部で作成され、**http.Client** の **request()** メソッドから返されます。それはヘッダが送信された進行中のリクエストを表現します。

レスポンスを取得するには、**'response'** 用のリスナーをリクエストオブジェクトに加えます。**'response'** イベントはレスポンスヘッダを受信するとリクエストオブジェクトによって生成されます。**'response'** イベントは **http.ClientResponse** のインスタンスを唯一の引数として実行されます。

'response' イベントの間、レスポンスオブジェクトにリスナーを加えることができます; とりわけ **'data'** イベントのリスナーです。**'response'** イベントはレスポンスボディのどの部分を受信するよりも前に呼び出されることに注意してください。そのため、ボディの最初の部分の受信と競合することを心配する必要はありません。**'response'** イベントの間に **'data'** イベントのリスナーが加えられる限り、ボディ全体を受信することができます。

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

これは **Writable Stream** です。

これは以下のイベントを持つ **EventEmitter** です。

イベント 'response'

```
function (response) { }
```

このリクエストに対するレスポンスを受信した時に生成されます。 このイベントは一回だけ生成されます。 `response` 引数は `http.ClientResponse` のインスタンスです。

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

ボディのチャンクを送信します。 このメソッドを何回も呼び出すと、サーバへのリクエストボディをストリーム化できます ー このケースは `['Transfer-Encoding', 'chunked']` ヘッダでリクエストを生成したことを意味します。

`chunk` 引数は整数の配列か文字列になります。

`encoding` 引数はオプションで、 `chunk` が文字列の場合だけ適用されます。

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

リクエストの送信を終了します。 ボディのいくつかの部分がまだ送信されていない場合、それはストリームにフラッシュされます。 リクエストがチャンク化されている場合、これは終端の `'0\r\n\r\n'` を送信します。

`data` が指定された場合は、 `request.write(data, encoding)` に続けて `request.end()` を呼び出すのと等価です。

http.ClientResponse

このオブジェクトは `http.Client` によってリクエストと一緒に作成されます。 これはリクエストオブジェクトの `'response'` イベントに渡されます。

レスポンスは `Readable Stream` を実装します。

イベント: 'data'

```
function (chunk) {}
```

メッセージボディの断片を受信した場合に生成されます。

例: ボディのチャンクは一つの引数として与えられます。 転送エンコーディングでデコードされます。 ボディのチャンクは文字列です。 ボディエンコーディングは `response.setBodyEncoding()` によって設定されます。

イベント: 'end'

```
function () {}
```

メッセージごとに厳密に一回だけ生成されます。 このイベントが生成された後、このレスポンスはどんなイベントも生成しません。

response.statusCode

3 桁の数字によるレスポンスのステータスコードです。例えば `404`。

response.httpVersion

接続しているサーバとの HTTP のバージョンです。 おそらく `'1.1'` または `'1.0'` のどちらかです。 同様に `response.httpVersionMajor` は最初の整数、 `response.httpVersionMinor` は 2 番目の整数です。

response.headers

レスポンスヘッダオブジェクトです。

response.setEncoding(encoding=null)

レスポンスボディのエンコーディングを設定します。 `'utf8'`、`'ascii'`、あるいは `'base64'` のいずれかです。 デフォルトは `null` で、 `'data'` イベントが `Buffer` を生成することを意味します。

response.pause()

イベントの生成によるレスポンスを中断します。ダウンロード速度を落とすのに便利です。

response.resume()

中断されていたレスポンスを再開します。

response.client

このレスポンスを所有する `http.Client` への参照です。

net.Server

このクラスは TCP または UNIX ドメインのサーバを作成するために使われます。

8124 番のポートを待ち受けるエコーサーバの例:

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

`'/tmp/echo.sock'` へのソケットを待ち受けるには、最後の行をこのように変更します

。

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

これは以下のイベントを持つ **EventEmitter** です:

イベント: 'connection'

function (stream) {}

新しいコネクションが作成されると生成されます。 **stream** は **net.Stream** のインスタンスです。

イベント: 'close'

function () {}

サーバがクローズした時に生成されます。

net.createServer(connectionListener)

新しい TCP サーバを作成します。 **connectionListener** 引数は **'connection'** イベントに対するリスナーとして自動的に加えられます。

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

指定された **port** と **host** でコネクションの受け入れを開始します。 **host** が省略されると、サーバはどんな IPv4 アドレスへの接続も受け入れます (**INADDR_ANY**)。

この関数は非同期です。最後の引数の **callback** はサーバがバインドすると呼び出されます。

server.listen(path, [callback])

与えられた **path** へのコネクションを待ち受けるする UNIX ドメインソケットのサーバを開始します。

この関数は非同期です。 最後の引数の **callback** はサーバがバインドすると呼び出されます。

server.listenFD(fd)

与えられたファイル記述子上のコネクションを待ち受けるサーバを開始します。

このファイル記述子は既に **bind(2)** および **listen(2)** システムコールが呼び出されていなければなりません。

server.close()

サーバが新しいコネクションを受け付けるのを終了します。 この関数は非同期で、サーバは最終的に **'close'** イベントを生成した時にクローズされます。

server.maxConnections

サーバの接続数が大きくなった時に接続を拒否するためにこのプロパティを設定し

ます。

server.connections
このサーバ上の並行コネクションの数です。

net.Stream

このオブジェクトは TCP または UNIX ドメインのソケットを抽象化したものです。 `net.Stream` のインスタンスは双方向のストリームインタフェースを実装します。 それらはユーザによって作成されて (`connect()` によって) クライアントとして使われるか、 Node によって作成されてサーバの `'connection'` イベントを通じてユーザに渡されます。

`net.Stream` のインスタンスは以下のイベントを持つ EventEmitter です:

イベント: 'connect'
`function () { }`

ストリームコネクションの確立が成功した場合に生成されます。 `connect()` を参照してください。

イベント: 'secure'
`function () { }`

ストリームコネクションにおいて、接続相手との SSL ハンドシェークの確立が成功した場合に生成されます。

イベント: 'data'
`function (data) { }`

データを受信した場合に生成されます。 `data` 引数は `Buffer` または `String` です。 データのエンコーディングは `stream.setEncoding()` で設定されます。 (より詳しい情報は `Readable Stream` を参照してください)。

イベント: 'end'
`function () { }`

ストリームの相手側が FIN パケットを送信した場合に生成されます。 このイベントが生成された後、`readyState` は `'writeOnly'` になります。 このイベントが生成されると、おそらく `stream.end()` を呼ばなければならないでしょう。

イベント: 'timeout'
`function () { }`

ストリームがタイムアウトして非アクティブになった場合に生成されます。 これはストリームがアイドルになったことを通知するだけです。 利用者は手動でコネクションをクローズする必要があります。

関連項目: `stream.setTimeout()`

イベント: 'drain'

```
function () { }
```

書き込みバッファが空になった場合に生成されます。アップロード速度を落とすために使うことができます。

イベント: 'error'

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

エラーが発生した場合に生成されます。`'close'` イベントはこのイベントの後に直接呼び出されます。

イベント: 'close'

```
function (had_error) { }
```

ストリームが完全にクローズした場合に生成されます。 引数 `had_error` は boolean で、ストリームが転送エラーでクローズされたのかどうかを示します。

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

新しいストリームオブジェクトを構築し、 指定の `port` と `host` へのストリームをオープンします。 第 2 引数が省略されると、ローカルホストが仮定されます。

ストリームが確立されると、`'connect'` イベントが生成されます。

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

指定の `port` と `host` でストリームをオープンします。 `createConnection()` もまたストリームをオープンします: 通常このメソッドは必要とされません。 これを使うのは、ストリームがクローズされた後にオブジェクトを再利用して別のサーバに接続する場合だけです。

この関数は非同期です。ストリームが確立されると `'connect'` イベントが生成されます。 接続で問題があった場合は `'connect'` イベントは生成されず、 例外とともに `'error'` イベントが生成されます。

stream.remoteAddress

リモートの IP アドレスを表現する文字列です。 例えば、`'74.125.127.100'` あるいは `'2001:4860:a005::68'`。

このメンバはサーバサイドのコネクションでのみ与えられます。

stream.readyState

`'closed'`、`'open'`、`'opening'`、`'readOnly'`、 あるいは `'writeOnly'` のいずれかです。

stream.setEncoding(encoding=null)
受信したデータのエンコーディングを設定します (`'ascii'`、`'utf8'`、 あるいは `'base64'` のいずれかです)。

stream.setSecure([credentials])
秘密鍵とサーバ証明書を指定した暗号モジュールの認証情報で、 ストリームに対して SSL サポートを有効にします。 オプションで認証局で証明された相手側の認証を使うこともできます。

認証情報が一つ以上の認証局の証明書を持っている場合、 ストリームは SSL コネクションにおけるハンドシェークの一部としてクライアント証明書を送るよう相手に要求します。 その有効性と内容は、`verifyPeer()` と `getPeerCertificate()` を通じてアクセスできます。

stream.verifyPeer()
指定された、あるいはデフォルトの信頼された認証局の証明書において、 相手の証明書の妥当性に応じて true または false を返します。

stream.getPeerCertificate()
相手の証明書の詳細を、'subject'、'issuer'、'valid_from' そして 'valid_to' をキーとする証明書の辞書を含む JSON 形式で返します。

stream.write(data, encoding='ascii')
ストリームにデータを送信します。 文字列の場合、第 2 引数はエンコーディングを指定します ー UTF8 はより遅いため、デフォルトは ASCII です。

データ全体のカーネルバッファへのフラッシュが成功すると `true` を返します。 データ全体または一部がユーザメモリ内のキューに入れられた場合は `false` を返します。 再びバッファが空いた場合は `'drain'` イベントが生成されます。

stream.end([data], [encoding])
ストリームをハーフクローズします。例えば FIN パケットを送信します。 サーバがデータを送り続けてくることがあり得ます。 このメソッドを呼び出した後の `readyState` は `'readOnly'` になります。

`data` が指定された場合は、 `stream.write(data, encoding)` に続けて `stream.end()` を呼び出すのと等価です。

stream.destroy()
このストリーム上でどんな I/O も起こらないことを保証します。 (パースエラーなどの) エラーの場合にだけ必要です。

stream.pause()
データの読み込みを中断します。つまり、`'data'` イベントは生成されません。 アップロード速度を落とすために便利です。

stream.resume()

`pause()` を呼び出した後で読み込みを再開します。

stream.setTimeout(timeout)

非アクティブなストリームが `timeout` ミリ秒後にタイムアウトするようにストリームを設定します。 デフォルトでは `net.Stream` はタイムアウトしません。

アイドルタイムアウトが引き起こされると、ストリームは `'timeout'` イベントを受信しますが、 コネクションは切断されません。 ユーザは手動で `end()` または `destroy()` を呼び出す必要があります。

`timeout` が 0 の場合、アイドルタイムアウトは無効にされます。

stream.setNoDelay(noDelay=true)

Nagle アルゴリズムを無効にします。 デフォルトでは TCP コネクションは Nagle アルゴリズムを使用し、データを送信する前にバッファリングします。 `noDelay` に設定すると、データは `stream.write()` を呼び出す度に即座に送信されます。

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

キープアライブ機能を有効/無効にします。 オプションでキープアライブの最初の調査がアイドルストリームに送信されるまでの初期遅延を設定します。 `initialDelay` (ミリ秒) が設定されると、 最後にデータパケットを受信してから最初のキープアライブ調査までの遅延が設定されます。 初期遅延に 0 が設定されると、デフォルト設定から値を変更されないようにします。

暗号化

このモジュールにアクセスするには `require('crypto')` を使用します。

暗号化モジュールは下層のプラットフォームで OpenSSL が有効であることを必要とします。 それは安全な HTTPS ネットワークや http コネクションの一部として使われる、 安全な認証情報をカプセル化する方法を提供します。

同時に OpenSSL のハッシュ、HMAC、暗号、復号、署名、そして検証へのラッパーを一式提供します。

crypto.createCredentials(details)

認証情報オブジェクトを作成します。オプションの `details` は以下のキーを持つ辞書です:

`key` : PEM でエンコードされた秘密鍵を保持する文字列

`cert` : PEM でエンコードされた証明書を保持する文字列

`ca` : 信頼できる認証局の証明書が PEM でエンコードされた文字列または文字列の配列

'ca' の詳細が与えられなかった場合、node.js はデフォルトとして http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt で与えられる、信頼できる認証局の公開されたリストを使用します。

crypto.createHash(algorithm)
ハッシュオブジェクトを生成して返します。 与えられたアルゴリズムによる暗号ハッシュ関数はダイジェストの生成に使われます。

`algorithm` は、 プラットフォーム上の OpenSSL のバージョンでサポートされている利用可能なアルゴリズムに依存します。 例えば sha1、md5、sha256、sha512、などです。 最近のリリースでは、`openssl list-message-digest-algorithms` で利用可能なダイジェストアルゴリズムが表示されます。

hash.update(data)
与えられた `data` でハッシュの内容を更新します。 これは新しいデータがストリームに流される際に何度も呼び出されます。

hash.digest(encoding='binary')
渡された全てのデータがハッシュ化されたダイジェストを計算します。 `encoding` は 'hex'、'binary'、または 'base64' のいずれかです。

crypto.createHmac(algorithm, key)
与えられたアルゴリズムとキーで HMAC を計算する、HMAC オブジェクトを作成して返します。

`algorithm` は OpenSSL でサポートされているアルゴリズムに依存します ー 前述の `createHash` を参照してください。

hmac.update(data)
与えられた `data` で HMAC の内容を更新します。 これは新しいデータがストリームに流される際に何度も呼び出されます。

hmac.digest(encoding='binary')
渡された全てのデータが HMAC 化されたダイジェストを計算します。 `encoding` は 'hex'、'binary'、または 'base64' のいずれかです。

crypto.createCipher(algorithm, key)
与えられたアルゴリズムとキーを使用する暗号オブジェクトを作成して返します。

`algorithm` は、OpenSSL に依存します。例えば aes192 などです。 最近のリリースでは、`openssl list-cipher-algorithms` で利用可能な暗号アルゴリズムが表示されます。

cipher.update(data, input_encoding='binary', output_encoding='binary')
`data` で暗号を更新します。 `input_encoding` で与えられるエンコーディングは 'utf8'、

'ascii'、'binary' のいずれかです。 `output_encoding` は暗号化されたデータの出力フォーマットを指定するもので、 'binary'、'base64' または 'hex' のいずれかです。

暗号化されたコンテンツが返されます。これは新しいデータがストリームに流される際に何度も呼び出されます。

cipher.final(output_encoding='binary')
暗号化されたコンテンツの残りを返します。 `output_encoding` は次のいずれかです: 'binary'、'base64' または 'hex'

crypto.createDecipher(algorithm, key)
与えられたアルゴリズムとキーを使用する復号オブジェクトを作成して返します。 これは前述の暗号オブジェクトの鏡写しです。

decipher.update(data, input_encoding='binary', output_encoding='binary')
'binary'、'base64' または 'hex' のいずれかでエンコードされた復号を `data` で更新します。 `output_decoding` は復号化されたプレーンテキストのフォーマットを指定するもので、 'binary'、'ascii' あるいは 'utf8' のいずれかです。

decipher.final(output_encoding='binary')
復号化されたプレーンテキストの残りを返します。 `output_decoding` は 'binary'、'ascii' あるいは 'utf8' のいずれかです。

crypto.createSign(algorithm)
与えられたアルゴリズムで署名オブジェクトを作成して返します。 最近のOpenSSLのリリースでは、`openssl list-public-key-algorithms` で利用可能な署名アルゴリズムの一覧が表示されます。例えば 'RSA-SHA256'。

signer.update(data)
署名オブジェクトをデータで更新します。 これは新しいデータがストリームに流される際に何度も呼び出されます。

signer.sign(private_key, output_format='binary')
署名オブジェクトに渡された全ての更新データで署名を計算します。 `private_key` は PEM でエンコードされた秘密鍵を内容とする文字列です。

'binary'、'hex'、あるいは 'base64' のいずれかを指定した `output_format` による署名を返します。

crypto.createVerify(algorithm)
与えられたアルゴリズムで検証オブジェクトを作成して返します。これは前述の署名オブジェクトと鏡写しです。

verifier.update(data)
検証オブジェクトをデータで更新します。 これは新しいデータがストリームに流される際に何度も呼び出されます。

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

署名されたデータを `public_key` と `signature` で検証します。 `public_key` は PEM でエンコードされた公開鍵を含む文字列です。 `signature` は先に計算したデータの署名で、 その `signature_format` は 'binary'、'hex'、または 'base64' のいずれかです。

署名されたデータと公開鍵による検証の結果によって true または false を返します。

DNS

このモジュールにアクセスするには `require('dns')` を使用します。

これは `'www.google.com'` を解決して、返された IP アドレスを逆引きで解決する例です。

```
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)

ドメイン (例 `'google.com'`) を解決して最初に見つかった A (IPv4) または AAAA (IPv6) レコードにします。

コールバックは引数 `(err, address, family)` を持ちます。 `address` 引数は IP v4 または v6 アドレスを表現する文字列です。 `family` 引数は 4 または 6 の整数で、`address` のファミリーを意味します (この値は必ずしも最初に `lookup` に渡す必要はありません)。

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

ドメイン (例 `'google.com'`) を解決して `rrtype` で指定されたレコードタイプの配列にします。 妥当な `rrtype` は `A` (IPV4アドレス)、`AAAA` (IPV6アドレス)、 `MX` (mail

exchangeレコード), `TXT` (テキストレコード)、 `SRV` (SRVレコード)、 `PTR` (IP を逆引きでルックアップするために使われる) です。

コールバックは引数 `(err, addresses)` を持ちます。 `addresses` の各要素の種類はレコードの種類によって決まり、 対応する後述のルックアップメソッドで記述されます。

エラー発生時、`err` は `Error` オブジェクトのインスタンスであり、 `err.errno` は後述するエラーコードのいずれか、 `err.message` はエラーを英語で説明する文字列となります。

dns.resolve4(domain, callback)

`dns.resolve()` と同じですが、IPv4 アドレス (`A` レコード) だけを問い合わせます。 `addresses` は IPv4 アドレスの配列です (例 `['74.125.79.104', '74.125.79.105', '74.125.79.106']`)

dns.resolve6(domain, callback)

IPv6 (`AAAA` レコード) を問い合わせることを除いて `dns.resolve4()` と同じです。

dns.resolveMx(domain, callback)

`dns.resolve()` と同じですが、mail exchange (`MX` レコード) だけを問い合わせます。

`addresses`は MX レコードの配列で、それぞれは priority と exchange の属性を持ちます (例 `[{'priority': 10, 'exchange': 'mx.example.com'},...]`)。

dns.resolveTxt(domain, callback)

`dns.resolve()` と同じですが、テキスト (`TXT` レコード) だけを問い合わせます。 `addresses` は利用可能な `domain` のテキストレコードの配列です。 (例、 `['v=spf1 ip4:0.0.0.0 ~all']`)

dns.resolveSrv(domain, callback)

`dns.resolve()` と同じですが、サービスレコード (`SRV` レコード) だけを問い合わせます。 `addresses` は利用可能な `domain` の SRV レコードの配列です。 SRV レコードのプロパティは priority、weight、port、そして name です (例 `[{'priority': 10, {'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]`)。

dns.reverse(ip, callback)

IP アドレスからドメイン名の配列へ逆引きで解決します。

コールバックは引数 `(err, domains)` を持ちます。

エラーがあった場合、`err` は非 null で Error オブジェクトのインスタンスとなります。

どの DNS 問い合わせもエラーコードを返せます。

- `dns.TEMPFAIL`: タイムアウト、SERVFAIL あるいは同様のもの。
- `dns.PROTOCOL`: 応答が不正。

- `dns.NXDOMAIN`: ドメインが存在しない。
- `dns.NODATA`: ドメインは存在するが、要求された種類のデータがない。
- `dns.NOMEM`: 処理中にメモリが不足。
- `dns.BADQUERY`: 問い合わせが不正な形式。

データグラム

データグラムソケットは `require('dgram')` で利用可能になります。 データグラムはほとんどの場合 IP/UDP メッセージで扱われますが、UNIX ドメインソケットでも使用することができます。

イベント: 'message'

```
function (msg, rinfo) { }
```

ソケット上で新しいデータグラムが到着した時に生成されます。 `msg` は `Buffer` で、`rinfo`は送信者のアドレス情報とデータグラムのバイト数を持ったオブジェクトです。

イベント: 'listening'

```
function () { }
```

ソケットでデータグラムの待ち受けを開始すると生成されます。 これは UDP ソケットが作成されるとすぐに発生します。 UNIX ドメインソケットでは `bind()` を呼び出すまで待ち受けを開始しません。

イベント: 'close'

```
function () { }
```

`close()` によってソケットがクローズすると生成されます。 このソケットでは新しい `message` イベントは生成されなくなります。

dgram.createSocket(type, [callback])

指定された種類のデータグラムソケットを作成します。 妥当な種類は: `udp4`、`udp6`、そして`unix_dgram` です。

オプションのコールバックは `message` イベントのリスナーとして加えられます。

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

UNIX ドメインのデータグラムソケット用です。相手先のアドレスはファイルシステムのパス名です。 オプションのコールバックはOSによって`sendto`の呼び出しが完了した後に起動されるために提供されるかもしれません。 コールバックが呼び出されるまで `buf` の再利用は安全ではありません。 `bind()` によってソケットがパスネームにバインドされていない限り、 このソケットでメッセージを受信することはないことに注意してください。

UNIXドメインソケット `/var/run/syslog` を通じて OSX 上の syslog にメッセージを送信する例:

```
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])

UDP ソケット用です。相手先のポートと IP アドレスは必ず指定しなければなりません。 **address** パラメータに文字列を提供すると、それは DNS によって解決されます。 DNS エラーと **buf** が再利用可能になった時のためにオプションのコールバックを指定することができます。 DNS ルックアップは送信を少なくとも次の機会まで遅らせることに注意してください。 送信が行われたことを確実に知る唯一の手段はコールバックを使うことです。

localhost の適当なポートに UDP パケットを送信する例;

```
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)

UNIX ドメインのデータグラムソケット用です。 **path** で指定されたソケット上でデータグラムの着信待ち受けを開始します。 クライアントは **bind()** しなくても **send()** することができますが、 **bind()** しないでデータグラムを受信することはありません。

受信した全てのメッセージをエコーバックする UNIX ドメインのデータグラムソケットサーバの例:

```
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);
```

このサーバと対話する UNIX ドメインのデータグラムクライアントの例:

```
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])

UDP ソケット用です。**port** とオプションの **address** でデータグラムを待ち受けます。**address** が指定されなければ、OS は全てのアドレスからの待ち受けを試みます。

41234 番ポートを待ち受ける UDP サーバの例:

```
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()

下層のソケットをクローズし、データの待ち受けを終了します。 `bind()` が呼び出されていない、自動的にメッセージを待ち受けていた UDP ソケットでも同じです。

dgram.address()
オブジェクトが持っているソケットのアドレス情報を返します。 UDP ソケットでは、このオブジェクトは `address` と `port` を持っています。 UNIX ドメインソケットでは、`address` だけを持っています。

dgram.setBroadcast(flag)
ソケットのオプション `SO_BROADCAST` を設定またはクリアします。 このオプションが設定されると、UDP パケットはローカルインタフェースのブロードキャスト用アドレスに送信されます。

dgram.setTTL(ttl)
ソケットオプションの `IP_TTL` を設定します。 TTL は「生存期間」を表しますが、このコンテキストではパケットが通過を許可される IP のホップ数を指定します。 各ルーターまたはゲートウェイはパケットを送出する際 TTL をデクリメントします。 ルータによって TTL がデクリメントされて 0 になるとそれは送出されません。 TTL 値の変更は通常、ネットワークの調査やマルチキャストで使われます。

`setTTL()` の引数は 1 から 255 のホップ数でです。ほとんどのシステムでデフォルトは 64 です。

表明

このモジュールはアプリケーションの単体テストを記述するために使用され、`require('assert')` でアクセスできます。

assert.fail(actual, expected, message, operator)
`actual` が `expected` と等しいか、提供された演算子を使ってテストします。

assert.ok(value, [message])
`value` が `true` かテストします、 これは `assert.equal(true, value, message);` と等価です。

assert.equal(actual, expected, [message])
`==` 演算子を強制して浅い同値性をテストします。

assert.notEqual(actual, expected, [message])
`==` 演算子を強制して浅い非同値性をテストします。

assert.deepEqual(actual, expected, [message])
深い同値性をテストします。

assert.notDeepEqual(actual, expected, [message])
深い非同値性をテストします。

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

`===` 演算子で厳密な同値性をテストします。

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

`!==` 演算子で厳密な非同値性をテストします。

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

`block` がエラーをスローすることを期待します。

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

`block` がエラーをスローしないことを期待します。

assert.ifError(value)

`value` が false でないことをテストし、true だったらそれをスローします。 コールバックの第 1 引数である `error` をテストするのに便利です。

パス

このモジュールはファイルパスを扱うユーティリティを含みます。 利用するには`require('path')`を呼び出してください。このモジュールは以下のメソッドを提供します。

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

全ての引数を一つに結合し、結果として得られるパスを決定します。

例:

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

path.normalizeArray(arr)

パスの要素の配列を正規化します。`'..'` と `'.'` の要素には注意してください。

例:

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

path.normalize(p)

文字列によるパスを正規化します。`'..'` と `'.'` の要素には注意してください。

例:

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

path.dirname(p)

パスに含まれるディレクトリ名を返します。Unixの **dirname** コマンドと同様です。

例:

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

path.basename(p, [ext])

パスの最後の要素を返します。Unixの **basename** コマンドと同様です。

例:

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

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

path.extname(p)

パスの拡張子を返します。パスの最後の要素について、最後の'.'から後にある文字列が対象になります。 最後の要素に'.'が含まれていなかった場合、もしくは'.'が最初の文字だった場合は、空の文字列を返します。例:

```
path.extname('index.html')
// 戻り値
'.html'

path.extname('index')
// 戻り値
''
```

path.exists(p, [callback])

与えられたパスが存在するかどうか検査します。そして引数の **callback** を真か偽か検査の結果とともに呼び出します。例:

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

URL

このモジュールはURLの解決や解析の為のユーティリティを持ちます。 利用するには `require('url')` を呼び出してください。

解析されたURLオブジェクトは、URL文字列の中に存在するかどうかに応じて 次に示すフィールドをいくつかもしくは全てを持ちます。 URL文字列に含まれないフィールドは解析結果のオブジェクトに含まれません。 次のURLで例を示します。

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

- `href`

  解析する前の完全URL。 例: `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- `protocol`

  リクエストプロトコル。 例: `'http:'`

- `host`

  URLの完全なホスト情報。認証情報を含みます。 例: `'user:pass@host.com:8080'`

- `auth`

  URLの認証情報。 例: `'user:pass'`

- `hostname`

  ホスト情報の中のホスト名。 例: `'host.com'`

- `port`

  ホスト情報の中のポート番号。 例: `'8080'`

- `pathname`

  URLのパス部分。ホスト情報からクエリまでの間に位置し、最初にスラッシュが存在する場合はそれも含みます。例: `'/p/a/t/h'`

- `search`

  URLのクエリ文字列。先頭の?マークも含みます。 例: `'?query=string'`

- `query`

  クエリの変数部分の文字列、もしくはクエリ文字列を解析したオブジェクト。 例:

`'query=string'` or `{'query':'string'}`

- `hash`

  URLの#マークを含む部分。 例: `'#hash'`

以下のメソッドはURLモジュールにより提供されます:

url.parse(urlStr, parseQueryString=false)

URL文字列を引数に取り、解析結果のオブジェクトを返します。 `querystring`モジュールを使ってクエリ文字列も解析したい場合は、 第2引数に `true` を渡してください。

url.format(urlObj)

URLオブジェクトを引数に取り、フォーマットしたURL文字列を返します。

url.resolve(from, to)

ベースとなるURLと相対URLを引数に取り、ブラウザがアンカータグに対して行うのと同様にURLを解決します。

クエリ文字列

このモジュールはクエリ文字列を処理するユーティリティを提供します。 以下のメソッドから成ります:

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

クエリオブジェクトを文字列へ直列化します。オプションとしてデフォルトの区切り文字と代入文字を上書き指定できます。 例:

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

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

この関数はデフォルトでPHP/Railsスタイルのようにパラメータに複雑な処理を行います。 配列やオブジェクトに対して `obj` に含まれる値を用いて処理を行います。 例:

```
querystring.stringify({foo: ['bar', 'baz', 'boz']})
// 戻り値
'foo%5B%5D=barfoo%5B%5D=bazfoo%5B%5D=boz'

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

もし配列への複雑な処理を無効にしたい場合は (Javaサーブレット用にパラメータを

生成する時など)、 引数の **munge** に対して **false** を設定することができます。例:

```
querystring.stringify({foo: ['bar', 'baz', 'boz']}, '', '=', false)
// 戻り値
'foo=barfoo=bazfoo=boz'
```

munge に **false** が設定されている時でも、 値にオブジェクトが設定されている場合は複雑に処理されたままであることに注意してください。

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

クエリ文字列をオブジェクトに復元します。オプションとしてデフォルトの区切り文字と代入文字を上書き指定できます。

```
querystring.parse('a=bb=c')
// 戻り値
{ 'a': 'b'
, 'b': 'c'
}
```

この関数は複雑化したクエリ文字列/していないクエリ文字列どちらに対しても解析することができます。 (詳細は **stringify** を参照)。

querystring.escape

escape関数は **querystring.stringify** で使用されていて、必要な場合にオーバーライドできるよう提供されています。

querystring.unescape

unescape関数は **querystring.parse** で使用されていて、必要な場合にオーバーライドできるよう提供されています。

REPL

Read-Eval-Print-Loop (REPL) は単独のプログラムとしても他のプログラムに手軽に取り込む形でも利用することができます。 REPLは対話的にJavaScriptを実行して結果を確認する手段を提供します。 デバッグやテストやその他の様々なことを試す用途で利用されます。

コマンドラインから **node** を引数無しで実行することで、REPLプログラムに入ります。 REPLはEmacs風の簡易な行編集機能を備えています。

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

```
1
2
3
```

より進んだ行編集を行うには、環境変数に **`NODE_NO_READLINE=1`** を設定してnodeを起動してください。 これによって正規の端末設定でREPLを起動し、**`rlwrap`** を有効にした状態でREPLを利用することができます。

例として、bashrcファイルに以下のように設定を追加します:

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

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

`prompt` でプロンプト記号を、 **`stream`** でI/Oを引数に取ってREPLを起動します。 **`prompt`** は省略可能で、 デフォルトは **`node`** です。 **`stream`** は省略可能で、 デフォルトは **`process.openStdin()`** です。

複数のREPLを起動した場合、同一のnodeインスタンスが実行されないことがあります。 それぞれのREPLはグローバルオブジェクトを共有しますが、I/Oは固有のものを持ちます。

REPLを標準入力、Unixドメインソケット、TCPソケットのもとで起動する例を示します:

```
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);
```

このプログラムをコマンドラインから実行すると、標準入力のもとでREPLが起動します。 他のREPLクライアントはUnixドメインソケットかTCPソケットを介して接続することができます。 **`telnet`** がTCPソケットへの接続に便利です。 **`socat`** はUnixドメイン/TCP両方のソケットへの接続に利用できます。

標準入力の代わりにUnixドメインソケットをベースとしたサーバからREPLを起動する

ことによって、 再起動することなくnodeの常駐プロセスへ接続することができます。

REPLの特長

REPLの中で Control+D を実行すると終了します。複数行に渡る式を入力とすることができます。

特別な変数である _ (アンダースコア) は一番最後の式の結果を保持します。

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

REPLはグローバルスコープに存在する全ての変数にアクセス可能です。 それぞれの**`REPLServer`** に紐づく**`context`** オブジェクトに変数を付与することで、 明示的に変数を公開させることが可能です。 例:

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

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

`context` オブジェクトに設定された変数は、REPLの中ではローカルな変数として現れます:

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

REPLには多くの特別なコマンドがあります:

- **`.break`** - 複数行に渡って式を入力している間に、途中で分からなくなったり完了させなくても良くなることがあります。**`.break`** で最初からやり直します。
- **`.clear`** - **`context`** オブジェクトを空の状態にリセットし、複数行に入力している式をクリアします。
- **`.exit`** - I/Oストリームを閉じ、REPLを終了させます。
- **`.help`** - このコマンドの一覧を表示します。

モジュール

Node は CommonJS のモジュールシステムを使います。

Node はシンプルなモジュールローディングシステムを持ちます。 Node では、ファイルとモジュールは 1 対 1 で対応しています。 例として、 `foo.js` は、同じディレクトリにある `circle.js` をロードしています。

`foo.js` の内容:

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

`circle.js` の内容:

```
var PI = 3.14;

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

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

`circle.js` モジュールは `area()` と `circumference()` をエクスポートしています。 エクスポートされたオブジェクトは、 `exports` という特別なオブジェクトに加えられます (`exports` の代わりに `this` を使うことも出来ます)。 モジュールのローカル変数はプライベートです。 この例の場合、変数 `PI` は `circle.js` のプライベート変数です。 関数 `puts()` は ビルトインモジュールである `'sys'` の物です。 プレフィックス `'./'` のないモジュールはビルトインモジュールです。詳細は以降で説明します。

プレフィックス `'./'` が付けられたモジュールは `require()` を呼び出したモジュールからの相対パスとなります。 つまり `circle.js` は `require('./circle')` が見つけられるように `foo.js` と同じディレクトリにある必要が有ります。

先頭の `'./'` 無しで、例えば `require('assert')` の様にモジュールを指定した場合、モジュールは `require.paths` の配列内の場所を起点に検索されます。 私のシステムでは、 `require.paths` はこの様になっています:

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

これによって、`require('assert')` が呼ばれた場合、Node は以下の順でモジュールを検索します

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

ファイルが見つかると、その時点で検索は終了します。 ファイル名が `'.node'` で終わるファイルは、バイナリ形式のアドオンモジュールです。 詳細は「アドオン」を参照してください。 `'index.js'` は、ディレクトリをモジュールとして一つにまとめることを可能にします。

`require.paths` は、配列に新しいパスを加えるか、`NODE_PATH` 環境変数と共に起動することで変更することが出来ます (この場合は、コロンで区切られたパスのリストを渡す必要があります)。

アドオン

アドオンは動的に共有オブジェクトをリンクします。 それは、C や C++ のライブラリに接合点を提供します。 API はいくつかのライブラリの知識を含んでおり、(現時点では) かなり複雑です。

- V8 JavaScript は C++ のライブラリです。 JavaScript のオブジェクト作成や関数呼び出し等のインタフェースに使用されます。 ドキュメントは主に、 `v8.h` のヘッダファイルに記されています (Node のソースツリーの中の `deps/v8/include/v8.h`)。

- libev は C の event loop ライブラリです。 ファイル記述子が読み取り専用になるのを待つとき、タイマーを待つとき、シグナルを受信するのを待つとき等に、libv のインタフェースが必要になります。 つまり、何らかの I/O 処理をすると必ず libev を使う必要があるということです。 Node は `EV_DEFAULT` というイベントループを使います。 ドキュメントは、http:/cvs.schmorp.de/libev/ev.html[here] にあります。

- libeio は C のスレッドプールライブラリです。 ブロックする POSIX システムコールを非同期に実行するために使用します。 こういった呼び出しのための大抵のラッパーは、既に `src/file.cc` に用意されているので、おそらくそれを使うことになるでしょう。 必要になったら、 `deps/libeio/eio.h` のヘッダファイルを参照して下さい。

- Node の内部ライブラリにおいて、もっとも重要なのは `node::ObjectWrap` クラスです。 このクラスから派生させることが多くなるでしょう。

- 他にどのような物が有るかは、 `deps/` 以下をご覧下さい。

Node は実行時に依存するソースを静的にコンパイルします。 モジュールのコンパイル時には、それらのリンクについて一切に気にする必要は有りません。

では、 C++ で以下の様に動作する小さなアドオンを作成してみましょう。

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

まず **hello.cc** というファイルを作成します:

```
#include v8.h

using namespace v8;

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

このソースコードは、 **hello.node** にバイナリアドオンとしてビルドされる必要が有ります。 以上を実行するために **wscript** という以下のようなコードを Python で書きました。

```
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'
```

node-waf configure build を実行すると、**build/default/hello.node** が作成されます。これが作成したアドオンです。

node-waf は http://code.google.com/p/waf/[WAF] にあります。 Python ベースのビルドシステムである **node-waf** は、ユーザの負担を減らすために提供されています。

Node のアドオンは全て、**init** というシグネチャで呼び出せる様に、エクスポートされる必要が有ります。:

```
extern 'C' void init (HandleObject target)
```

現時点では、アドオンのドキュメントはこれで全てです。実際の例は、http://github.com/ry/node_postgres をご覧下さい。

付録 - サードパーティのモジュール

Node 向けにサードパーティ製のモジュールが数多くあります。 執筆時点 (2010 年 8 月) では、モジュールのマスタリポジトリは http://github.com/ry/node/wiki/modules[the wiki page] です。

この付録は、良質だと考えられているモジュールを初心者が素早く見つけることを手助けする「小さな」ガイドを意図しています。 これは完全なリストは意図していません。 どこかでより完全なモジュールが見つかるかもしれません。

- Module Installer: npm
- HTTP Middleware: Connect
- Web Framework: Express
- Web Sockets: Socket.IO
- HTML Parsing: HTML5
- mDNS/Zeroconf/Bonjour
- RabbitMQ, AMQP
- mysql
- Serialization: msgpack
- Scraping: Apricot
- Debugger: ndb is a CLI debugger inspector is a web based tool.
- pcap binding
- ncurses
- Testing/TDD/BDD: vows, expresso, mjsunit.runner

このリストへのパッチを歓迎します。