NodeJS v6.1 API 中文文档

Lyndon

Published with GitBook

Table of Contents

Node.js v6.1 API 中文文档

前言

本书是对NodeJS v6.1版本API的英文文档的翻译，撰写此书的目的主要是为了广大的国人提供一个比较靠谱的NodeJS官方API文档的中文解释稿，方便大家日后的开发工作，同时我本人也能在翻译的过程中深入地学习NodeJS的各个模块和它们的API，加深我自己对NodeJS本身的了解。由于本人能力有限，时间有限，因此对源文档中少数不重要的部分做了删减，同时书中纰漏之处也在所难免，希望大家在看到错误的时候能够及时指正，谢谢。

授权声明

有关该文档

我们撰写该文档的目的是为了综合性地阐释Node.js的API设计，既给出参考，同时也解释它们的概念。每一个部分都描述了一个内建的模块或者一个高层概念。

属性的类型、方法的参数以及事件处理函数的参数都陈列在主题栏下方合适的位置。

如果你找到了文档中的错误，请提交这一问题或者参阅这份指南来获取如何提交一个补丁的方法。

稳定性标示

贯穿整篇文档，在某些部分你会看到稳定性标示部分。因为Node.js的API仍然处在变化中，在它日趋成熟的过程中，某些部分相比于其它部分会变得更加稳定可靠。某些部分甚至已经非常稳定了，以至于它们不太可能变化。而其它部分仍然是全新的或者是实验性质的，亦或者是有害的并且正处在重构的过程中。

稳定性标示如下所示：

- 稳定性：0-弃用。这一特性已知是有问题的，并且已经有了更新的计划。不要依赖这些功能。使用这些功能也许会引发警告。后向兼容将不考虑这些功能。
- 稳定性：1-实验。这些特性会被改动，并且由命令行旗标来控制。在未来的版本中它们也许会被更改或者删除。
- 稳定性：2-稳定。这些API已经被证明是令人满意的。它们与NPM生态系统的兼容性被优先考虑，并且在没有非常必要的情况下它们不会被更改。
- 稳定性：3-锁定。只有关于安全、性能或者bug的修正会被接受。请不要提议任何更改这些API的建议，它们不会被接受。

系统命令和man帮助页

有些系统命令比如open(2)以及read(2)定义了用户程序以及底层操作系统的接口。只是简单包装了系统命令的Node函数会在文档中列出，比如 `fs.open()` 。这些文档提供了相应man帮助页的链接，这些帮助信息会解释相应的系统命令是如何工作的。

警告：某些系统命令，比如lchown(2)，是BSD系统特有的。这意味着类似 `fs.lchown()` 这样的命令只会在Mac OS X以及其它BSD衍生系统中工作，而在Linux上则无法工作。

大多数Unix系统命令都有它们的Windows对应版本，但是他们的行为可能会有所不同。Node issue 4760给出了其中一个微妙的例子。

用例

内容

使用

`node [options] [v8 options] [script.js | -e "script"] [arguments]` 请参阅命令行参数选项获取不同命令行参数的信息以及使用Node.js运行脚本的方法（英文版）。

样例

下面是一个使用Node.js写的web服务器的样例，它会返回 `'Hello World'` 信息。

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

把上述代码写入一个名叫 `example.js` 的文件然后用Node.js来执行。

```
$ node example.js
Server running at http://127.0.0.1:3000/
```

文档中所有的示例代码都可以用相似的方法来运行。

Assertion Testing

内容

Assert

> 稳定性：3-锁定 `assert` 模块提供了测试不变量的简单工具集合。该模块计划供Node.js内部使用，但是也可以通过 `require('assert')` 的方式在个人应用中使用。尽管如此，`assert` 并不是一个测试框架，而且也并不打算成为一个普适性的断言库。

`assert` 模块提供的API是锁定的。这意味着该模块中实现的任何方法都不会有任何添加或者变化。

assert

assert.ok()的别名。

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

assert(true); // OK
assert(1);  // OK
assert(false);
  // 抛出 "AssertionError: false == true"
assert(0);
  // 抛出 "AssertionError: 0 == true"
assert(false, 'it\'s false');
  // 抛出 "AssertionError: it's false"
```

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

测试 `actual` 和 `expected` 之间是否深度相等。在原生类型的值之间使用 `==` 进行比较。

只有“自有”的可枚举属性才会被比较。 `deepEqual()` 的实现并不测试对象的原型、附属标记以及不可枚举的属性。这也许会产生一些令人惊讶的结果。比如，下面的代码并不会抛出 `AssertionError` ，因为 `Error` 对象的属性是不可枚举的。

```
// 警告：下面的代码不会抛出AssertionError！
assert.deepEqual(Error('a'), Error('b'));
```

“深度”相等意味着一个对象的子对象的“自有”可枚举属性也会被检测：

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

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);

assert.deepEqual(obj1, obj1);
  // OK, 对象相等

assert.deepEqual(obj1, obj2);
  // AssertionError: { a: { b: 1 } } deepEqual { a: { b: 2 } }
  // b的值并不相等

assert.deepEqual(obj1, obj3);
  // OK, 对象相等

assert.deepEqual(obj1, obj4);
  // AssertionError: { a: { b: 1 } } deepEqual {}
  // 原型被忽略
```

如果比较的值深度不相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined`，默认的错误信息会被赋值给该属性。

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

基本上和 `assert.deepEqual()` 相同，但有两点不同。首先，原生类型的值使用 `===` 进行比较。其次，对象间的比较包括了原型的严格相等性检查。

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

assert.deepEqual({a: 1}, {a: '1'});
  // OK, 因为 1 == '1'

assert.deepStrictEqual({a: 1}, {a: '1'});
  // AssertionError: { a: 1 } deepStrictEqual { a: '1' }
  // 因为在使用严格相等符进行比较时1 !== '1'
```

如果比较的值深度严格不相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined` ，默认的错误信息会被赋值给该属性。

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

断言 `block` 函数不会抛出一个错误。assert.throws()提供了更多细节。

`assert.doesNotThrow()` 被调用时，它会立即执行 `block` 函数。

如果一个和 `error` 类型一致的错误被抛出， `AssertionError` 会被抛出。如果抛出的错误与 `error` 属于不同的类型或者 `error` 参数未被设置，那么该错误会被传播给调用者。

下面的代码会抛出 `TypeError` ，因为在断言中没有声明相符合的错误类型。

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

但是下面的代码会抛出 `AssertionError` 并附带有'Got unwanted exception (TypeError)..'的信息。

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

如果一个 `AssertionError` 被抛出，并且 `message` 属性已经通过 `message` 参数进行了赋值， `message` 参数的信息会被添加到 `AssertionError` 的末尾。

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

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

使用 `==` 运算符测试 `actual` 和 `expected` 是否粗浅相等。

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

assert.equal(1, 1);
  // OK, 1 == 1

assert.equal(1, '1');
  // OK, 1 == '1'

assert.equal(1, 2);
  // AssertionError: 1 == 2
assert.equal({a: {b : 1}}, {a: {b: 1}});
  // AssertionError: { a: { b: 1 } } == { a: { b: 1 } }
```

如果比较的值不相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined` ，默认的错误信息会被赋值给该属性。

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

期望抛出一个 `AssertionError` 。如果 `message` 未被定义， `actual` 和 `expected` 会被 `operator` 分隔并以此作为错误信息。反之， `message` 会被作为错误信息输出。

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

assert.fail(1, 2, undefined, '>');
  // AssertionError: 1 > 2

assert.fail(1, 2, 'whoops', '>');
  // AssertionError: whoops
```

assert.ifError(value)

如果 `value` 为真则抛出该 `value` 。在测试回调函数中的 `error` 参数时非常有用。

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

assert.ifError(0); // OK
assert.ifError(1); // 抛出1
assert.ifError('error'); // 抛出 'error'
assert.ifError(new Error()); // 抛出Error
```

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

测试深度不等性，与assert.deepEqual()相反。

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

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    b: 2
  }
};
const obj3 = {
  a: {
    b: 1
  }
};
const obj4 = Object.create(obj1);

assert.notDeepEqual(obj1, obj1);
  // AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj2);
  // OK, obj1和obj2深度不等

assert.notDeepEqual(obj1, obj3);
  // AssertionError: { a: { b: 1 } } notDeepEqual { a: { b: 1 } }

assert.notDeepEqual(obj1, obj4);
  // OK, obj1和obj4深度不等
```

如果比较的值深度相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined` ，默认的错误信息会被赋值给该属性。

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

测试深度严格不等性，与assert.deepStrictEqual()相反。

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

assert.notDeepEqual({a: 1}, {a: '1'});
  // AssertionError: { a: 1 } notDeepEqual { a: '1'}

assert.notDeepStrictEqual({a: 1}, {a: '1'});
  // OK
```

如果比较的值深度严格相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined` ，默认的错误信息会被赋值给该属性。

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

使用 `!=` 测试粗浅不等性。

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

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

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

assert.notEqual(1, '1');
  // AssertionError: 1 != '1'
```

如果比较的值相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined` ，默认的错误信息会被赋值给该属性。

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

使用 `!==` 测试严格不等性。

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

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

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

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

如果比较的值严格相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined`，默认的错误信息会被赋值给该属性。

assert.ok(value[,message])

测试 `value` 是否为真。这与 `assert.equal(!!value, true, message)` 效果相同。

如果 `value` 不为真，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined`，默认的错误信息会被赋值给该属性。

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

assert.ok(true); // OK
assert.ok(1);  //OK
assert.ok(false);
  // 抛出 "AssertionError: false == true"
assert.ok(0);
  // 抛出 "AssertionError: 0 == true"
assert.ok(false, 'it\'s false');
  // 抛出 "AssertionError: it's false"
```

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

使用 `===` 测试严格相等性。

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

assert.strictEqual(1, 2);
  // AssertionError: 1 === 2

assert.strictEqual(1, 1);
  // OK

assert.strictEqual(1, '1');
  // AssertionError: 1 === '1'
```

如果比较的值严格相等，`AssertionError` 会被抛出并带有 `message` 参数设定的 `message` 属性。如果 `message` 参数为 `undefined`，默认的错误信息会被赋值给该属性。

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

期望 `block` 函数抛出一个错误。

`error` 参数如果被设置，它可以是一个构造器、正则表达式或者检验函数。

`message` 参数如果被设置，它将会被添加在 `AssertionError` 输出的末尾。

使用构造器验证错误类型的例子：

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

使用正则表达式验证错误信息的例子：

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

自定义的错误验证：

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

注意 `error` 不能是字符串。如果该函数的第二个参数是字符串，那么 `error` 参数会被省略，该值会被赋予 `message` 参数。这可能会不经意地造成错误：

```
// 这是一个错误！不要这样做！
assert.throws(myFunction, 'missing foo', 'did not throw with expected message');

// 应该这样做。
assert.throws(myFunction, /missing foo/, 'did not throw with expected message');
```

Buffer

内容

- buf.includes(value[,byteOffset][,encoding])
- buf.length
- buf.readDoubleBE(offset[,noAssert])
- buf.readDoubleLE(offset[,noAssert])
- buf.readFloatBE(offset[,noAssert])
- buf.readFloatLE(offset[,noAssert])
- buf.readInt8(offset[,noAssert])
- buf.readInt16BE(offset[,noAssert])
- buf.readInt16LE(offset[,noAssert])
- buf.readInt32BE(offset[,noAssert])
- buf.readInt32LE(offset[,noAssert])
- buf.readIntBE(offset,byteLength[,noAssert])
- buf.readIntLE(offset,byteLength[,noAssert])
- buf.readUInt8(offset[,noAssert])
- buf.readUInt16BE(offset[,noAssert])
- buf.readUInt16LE(offset[,noAssert])
- buf.readUInt32BE(offset[,noAssert])
- buf.readUInt32LE(offset[,noAssert])
- buf.readUIntBE(offset,byteLength[,noAssert])
- buf.readUIntLE(offset,byteLength[,noAssert])
- buf.slice([start[,end]])
- buf.swap16()
- buf.swap32()
- buf.toString([encoding[,start[,end]]])
- buf.toJSON()
- buf.values()
- buf.write(string[,offset[,length]],encoding])
- buf.writeDoubleBE(value,offset[,noAssert])
- buf.writeDoubleLE(value,offset[,noAssert])
- buf.writeFloatBE(value,offset[,noAssert])
- buf.writeFloatLE(value,offset[,noAssert])
- buf.writeInt8(value,offset[,noAssert])
- buf.writeInt16BE(value,offset[,noAssert])
- buf.writeInt16LE(value,offset[,noAssert])
- buf.writeInt32BE(value,offset[,noAssert])
- buf.writeInt32LE(value,offset[,noAssert])
- buf.writeIntBE(value,offset,byteLength[,noAssert])
- buf.writeIntLE(value,offset,byteLength[,noAssert])
- buf.writeUInt8(value,offset[,noAssert])
- buf.writeUInt16BE(value,offset[,noAssert])

缓冲

> 稳定性：2-稳定。

在ECMAScript 2015(ES6)引入 `TypedArray` 之前，JavaScript并没有读取或者操纵二进制流数据的机制。 `Buffer` 类被Node.js引入作为其API的一部分，使得Node.js可以与八进制流数据进行交互，比如处理TCP流或者文件系统操作等。

虽然 `TypedArray` 已经被加入到了ES6中，但是 `Buffer` 类实现了优化的、更适合Node.js的 `Uint8Array` 的API。

`Buffer` 类的实例与整型数组类似，但是它们被分配在固定大小的、独立于V8堆之外的内存空间里。 `Buffer` 的大小在它们被创建时就已经固定了，不能被更改。

`Buffer` 类是Node.js的一个全局模块，使用者可以无需声明 `require('buffer')` 语句就直接使用。

```
const buf1 = Buffer.alloc(10);
  // 建立一个由0填充的、长度为10的缓冲实例。

const buf2 = Buffer.alloc(10, 1);
  // 建立一个由0x01填充的、长度为10的缓冲实例。

const buf3 = Buffer.allocUnsafe(10);
  // 建立一个没有经过初始化的、长度为10的缓冲实例。
  // 该方法的执行速度比Buffer.alloc()更快，但是返回的实例也许含有旧数据，需要使用fill()或者write()函

const buf4 = Buffer.from([1,2,3]);
  // 建立一个含有[01, 02, 03]数据的缓冲。

const buf5 = Buffer.from('test');
  // 建立一个含有ASCII字节数据[74, 65, 73, 74]的缓冲。

const buf6 = Buffer.from('tést', 'utf8');
  // 建立一个含有UTF8字节数据[74, c3, a9, 73, 74]的缓冲。
```

Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe()

Class: SlowBuffer

> 稳定性：0-弃用。使用Buffer.allocUnsafeSlow(size)来代替。

new SlowBuffer(size)

> 稳定性：0-弃用。使用Buffer.allocUnsafeSlow(size)来代替。

Command Line Options

内容

命令行选项

Node.js拥有很多命令行选项。这些选项包括了内建调试功能、多种执行Node.js脚本的方式以及有助宜的运行时选项等。

在终端中执行 `man node` 来查看 `node` 命令行选项的帮助页。

概要

`node [options] [v8 options] [script.js | -e "script"] [arguments]` `node debug [script.js | -e "script" | <host>:<port>] ...` `node --v8-options` 不使用任何参数执行 `node` 命令将会开启一个 `REPL` 会话。

Options

-v, --version

打印node的版本号信息。

-h, --help

打印node的命令行选项信息。该信息相比于本文档而言比较简略。

-e, --eval "script"

把后续的参数当做JavaScript来执行。在REPL中预定义的模块也可以在 `script` 中使用。

-p, --print "script"

与 `-e` 选项相同，但是该命令会打印出执行结果。

-c, --check

检查脚本的语法错误但不执行脚本。

-i, --interactive

打开REPL会话，无论终端是否被设定为标准输入。

-r, --require module

在启动时预加载指定的模块。

遵循 `require()` 的模块解析规则。 `module` 可以是文件路径或者node模块名。

--no-deprecation

关闭弃用警告。

--trace-deprecation

打印弃用模块的堆栈跟踪信息。

--throw-deprecation

执行弃用模块时抛出错误。

--no-warnings

关闭所有警告（包括弃用警告）。

--trace-warnings

打印所有进程的警告信息（包括弃用模块）。

--trace-sync-io

当同步I/O在事件循环的第一轮执行中被检测到时打印堆栈跟踪信息。

--zero-fill-buffers

自动使用0填充所有新分配的**Buffer**以及**SlowBuffer**实例。

--track-heap-objects

为堆快照跟踪堆对象的分配情况。

--prof-process

使用v8选项 `--prof` 处理v8 profiler产生的输出。

--v8-options

打印v8命令行选项。

--tls-cipher-list=list

指明替代的默认TLS加密器列表。（默认要求Node.js在编译时有crypto支持）

--enable-fips

在启动时开启FIPS-compliant crypto。（要求使用 `./configure --openssl-fips` 编译Node.js）

--force-fips

在启动时强制实施FIPS-compliant。（不能在执行脚本中取消）（与 `--enable-fips` 要求相同）

--icu-data-dir=file

指定ICU数据加载路径。（覆盖 `NODE_ICU_DATA` 变量的值）

环境变量

NODE_DEBUG=module[,...]

输入以 `,` 分隔的核心模块列表，在该列表中的模块被执行时将会打印出调试信息。

NODE_PATH=path[:...]

输入以 `:` 分隔的目录路径列表，Node.js会在该列表提供的路径中搜索需要加载的模块。

注意：在Windows中该列表以 `;` 分隔。

NODE_DISABLE_COLORS=1

当被设定为 `1` 时，REPL将不会使用彩色模式。

NODE_ICU_DATA=file

存放ICU（Intl对象）数据的路径。在使用small-icu支持的情况下编译时将会扩展关联的数据。

NODE_REPL_HISTORY=file

存放REPL运行历史的文件所在路径。默认路径是 `~/.node_repl_history` ，可以使用该变量来改写。可以通过将该变量设为 `""` 或者 `" "` 的方式来关闭REPL运行历史的持久化。

Modules

内容

模块

> 稳定性：3-锁定。

Node.js有一个简单地模块加载系统。在Node.js中文件和模块是一对一对应的。举个例子，`foo.js`在相同的文件夹中读取`circle.js`模块。`foo.js`的代码如下所示：

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

`circle.js` 的代码如下所示：

```
const PI = Math.PI;
exports.area = (r) => PI * r * r;
exports.circumference = (r) => 2 * PI * r;
```

`circle.js` 模块导出了 `area()` 以及 `circumference()` 函数。你可以将函数和对象加入到一个特别的叫做 `exports` 的对象中使它们加入到你的根模块中。

模块的本地变量是模块私有的，因为模块本身被Node.js包装在一个函数里。在这个例子中，变量 `PI` 是 `circle.js` 的私有变量。

如果你想要让你的模块导出为一个函数（比如一个构造器）或者如果你想要导出一个完整的对象来赋值给一个变量而不是一次构建一个属性，你可以把要导出的内容赋值给 `module.exports` 而不是 `exports` 。

在下面的例子中， `bar.js` 使用了 `square` 模块，这个模块是一个构造器：

```
const square = require('./square.js');
var mySquare = square(2);
console.log(`The area of my square is ${mySquare.area()}`);
```

`square` 模块定义在 `square.js` 中：

```
// 赋值给exports不会修改这个模块，必须要使用module.exports
module.exports = (width) => {
  return {
    area: () => width * width;
  };
}
```

模块系统在 `require('module')` 中实现。

获取主模块

当一个文件直接通过Node.js来运行时 `require.main` 被设置为 `module` 。这意味着你可以通过测试一下语句来获知是否一个文件是否正在被Node.js直接运行：

```
require.main === module
```

比如一个名叫 `foo.js` 的文件，如果通过 `node foo.js` 来运行那么上面代码的输出会是 `true` ，但是如果通过 `require('./foo')` 的方式来运行，那么输出将会是 `false` 。

因为 `module` 提供了 `filename` 属性（一般而言和 `__filename` 等同），现在正在运行的应用的入口就可以通过检查 `require.main.filename` 来获知。

附录：包管理的建议

获取主模块

缓存

模块缓存的注意事项

核心模块

周期

文件模块

把文件夹作为模块

从**node_modules**文件夹下读取

从全局文件夹下读取

模块包装器

模块对象

module.children

module.exports

module.alias

module.filename

module.id

module.loaded

module.parent

module.require(id)

String Decoder

内容

StringDecoder

> 稳定性：2-稳定。

通过 `require('string_decoder')` 来使用该模块。该模块解码一个数据缓冲为字符串。这是 `buffer.toString()` 的简化版接口，但是提供了针对utf8的额外支持。

```
const StringDecoder = require('string_decoder').StringDecoder;
const decoder = new StringDecoder('utf8');

const cent = Buffer.from([0xC2, 0xA2]);
console.log(decoder.write(cent));

const euro = Buffer.from([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

Class: StringDecoder

接受一个单一参数 `encoding` ，该参数默认为 `'utf8'`

decoder.end()

返回缓冲中剩余的字节。

decoder.write(buffer)

返回一个解码的字符串。