

Node.js 前端开发指南

[原文链接](#)

[关于作者](#)

2018年6月21日出版

本指南面向了解JavaScript但尚未十分熟悉Node.js的前端开发人员。我这里不专注于语言本身 -- Node.js 使用 V8 引擎，所以和Google Chrome的解释器是一样的，这点您或许已经了解（但是，它也可以在不同的VM上运行，请参阅 [node-chakracore](#)）

Node.js 前端开发指南

[Node 版本](#)

[不需要Babel](#)

[回调风格](#)

[事件循环](#)

[Node.js中的事件循环](#)

[事件发射器](#)

[流](#)

[模块系统](#)

[环境变量](#)

[综合运用](#)

[总结](#)

[相关文章](#)

我们经常跟Node.js打交道，即使你是一名前端开发人员 -- [npm脚本](#)，webpack配置，gulp任务，[程序打包](#) 或 [运行测试](#)等。即使你真的不需要深入理解这些任务，但有时候你会感到困惑，会因为缺少Node.js的一些核心概念而以非常奇怪的方式来编码。熟悉Node.js之后，您还可以让某些原本需要手动操作的东西自动执行，让您可以更自信地查看服务器端代码，并编写更复杂的脚本。

Node 版本

Node.js与客户端代码最大的区别在于您可以根据运行环境来决定，并且可以完全清楚它支持哪些特性 -- 您可以根据具体的需求和可用的服务器来选择使用哪个版本。

Node.js有一个公开发布时间表，告诉我们奇数版本没有被长期支持。当前的LTS（long-term support）版本将被积极开发到2019年4月，然后2019年12月31日之前，通过更新关键代码进行维护。Node.js新版本正在积极开发，它们带来了许多新功能，以及安全性和性能方面的提升。这也许是使用当前活跃版本的一个好理由。然而，没有人真正强迫你，如果你不想这样做，使用旧版本也可以，等到您觉得时机合适再更新就行。

Node.js被广泛应用于现代前端工具链 - 我们很难想象一个现代项目没有使用Node工具进行任何处理。因此，您可能已经熟悉nvm（node版本管理器），它允许你同时安装几个Node版本，为每个项目选择正确的版本。使用这种工具的原因在于，不同项目经常使用不同的Node版本，并且你不想永远保持它们同步，您只想保留编写和测试它们的环境。其它语言也有很多这样的工具，例如用于Python的virtualenv，用于Ruby的rbenv等等。

不需要Babel

由于您可以自由选择任何Node.js版本，所以您很有可能使用LTS版本。该版本在本文撰写时为8.11.3，几乎支持所有ECMAScript 2015的规范，除了尾递归。

这意味着我们不需要Babel，除非您遇到一个非常旧的Node.js版本，需要转换JSX，或者需要其它前沿的转换器。在实践中，Babel并不是那么重要，所以您运行的代码可以和编写的代码相同，不需要任何编译器 -- 这个我们已经遗忘的客户端天才。

我们也不需要webpack或browserify，那么我们就没有工具来重新加载我们的代码 -- 如果您在开发类似Web服务器的东西，您可以使用nodemon，在文件更改后来重新加载您的应用程序。

而且因为我们不在任何地方传送代码，所以不需要缩小它 -- 省了一步：您只需原封不动地使用代码，真的很神奇！

回调风格

以前，Node.js中的异步函数接受带有签名 `(err, data)` 的回调，其中第一个参数代表错误信息 - 如果它为null，则全部正确，否则您必须处理错误。这些处理程序会在操作完成，我们得到响应后调用。例如，让我们读取一个文件：

```
const fs = require('fs');
fs.readFile('myFile.js', (err, file) => {
  if (err) {
    console.error('There was an error reading file :(');
    // process is a global object in Node
    // https://nodejs.org/api/process.html#process_process_exit_code
    process.exit(1);
  }

  // do something with file content
});
```

我们很快就发现，这种风格很难编写可读和可维护的代码，甚至造成回调地狱。后来，一种新的原生的异步处理方式 `Promise` 被引入了。它在ECMAScript 2015上标准化（是浏览器和Node.js运行时的全局对象）。近来，`async / await` 在ECMAScript 2017中标准化了，Node.js 7.6+ 都支持这个规范，所以您可以在LTS版本中使用它。

有了 `Promise`，我们避免了“回调地狱”。但是，现在我们遇到的问题是旧代码和许多内置模块仍然使用回调的方式。将它们转换为 `Promise` 并不是很难 -- 为了阐释清楚，我们将fs.readFile转成 `Promise`：

```
const fs = require('fs');
function readFile(...arguments) {
  return new Promise((resolve, reject) => {
    fs.readFile(...arguments, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}
```

这种模式可以很容易地扩展到任何函数，并且内置的utils模块中有一个特殊的函数 - `utils.promisify`。官方文档中的示例：

```
const util = require('util');
const fs = require('fs');
const stat = util.promisify(fs.stat);

stat('.').then((stats) => {
  // Do something with stats
}).catch((error) => {
  // Handle the error.
});
```

Node.js核心团队明白我们需要从旧风格中迁移出来，他们尝试引入一个内置模块的 `promisified` 版本 - 已经有 `promisified` 文件系统模块了，虽然写这篇文章时它还在处于试验阶段。

你仍然会遇到很多旧式的、带回调的Node.js代码，为了保持一致性，建议使用 `utils.promisify` 把它们包装一下。

事件循环

事件循环几乎与在浏览器环境下一样，只是有一些扩展。然而，由于这个主题比较高深，我将全面讲解下，不仅仅是差异（我会重点强调这部分，让您知道哪些是Node.js特有的）。



Node.js中的事件循环

JavaScript在构建时考虑了异步行为，因此我们通常不会马上执行所有操作。以下列举的方法，事件不会直接按顺序执行：

- microtasks

例如，立即处理Promises，如Promise.resolve。它意味着这段代码会在同一个的事件循环中被执行，但得等到所有同步代码执行完后。

process.nextTick

这是Node.js特有的方法，它不存在于任何浏览器（以及进程对象）中。它的行为类似于微任务(microtask)，但具有优先级。这意味着它将在所有同步代码之后立即执行，即使之前引入了其他微任务 - 这是很危险的，可能导致无限循环。从命名上讲是不对的，因为它是在同一个事件循环中执行的，而不是在它的 `next tick` 中执行。但是由于兼容性原因，它可能保持不变。

setImmediate

虽然它确实存在于某些浏览器中，但并未在所有浏览器中达到一致的行为，因此在浏览器中使用时，您需要非常小心。它类似于 `setTimeout(0)` 代码，但有时会优先于它。这里的命名也不是最好的 - 我们在谈论下一个事件循环迭代，它并不是真正的 `immediate`。

setTimeout/setInterval

定时器在Node和浏览器中的表现形式是相同的。关于定时器的一个重要的事情是，我们提供的延迟不代表在这个时间之后回调就会被执行。它的真正含义是，一旦主线程完成所有操作（包括微任务）并且没有其它具有更高优先级的定时器，Node.js将在此时间之后执行回调。

让我们看看这个例子：

往下看我会给出脚本执行后正确的输出，但是如果你愿意，请尝试自己完成它（当一回“JavaScript解释器”）：

```
const fs = require('fs');
console.log('beginning of the program');
const promise = new Promise(resolve => {
  // function, passed to the Promise constructor
  // is executed synchronously!
  console.log('I am in the promise function!');
  resolve('resolved message');
});
promise.then(() => {
  console.log('I am in the first resolved promise');
}).then(() => {
  console.log('I am in the second resolved promise');
});
process.nextTick(() => {
  console.log('I am in the process next tick now');
});
fs.readFile('index.html', () => {
  console.log('=====');
  setTimeout(() => {
    console.log('I am in the callback from setTimeout with 0ms delay');
  }, 0);
  setImmediate(() => {
    console.log('I am from setImmediate callback');
  });
});
setTimeout(() => {
  console.log('I am in the callback from setTimeout with 0ms delay');
}, 0);
```

```
setImmediate(() => {
  console.log('I am from setImmediate callback');
});
```

正确的执行顺序如下：

```
node event-loop.js
beginning of the program
I am in the promise function!
I am in the process next tick now
I am in the first resolved promise
I am in the second resolved promise
I am in the callback from setTimeout with 0ms delay
I am from setImmediate callback
=====
I am from setImmediate callback
I am in the callback from setTimeout with 0ms delay
```

您可以在Node.js官方文档中获取更多有关事件循环和process.nextTick的信息。

事件发射器

Node.js中的许多核心模块派发或接收不同的事件。它有一个EventEmitter的实现，是一个发布 - 订阅模式。这与浏览器DOM事件非常相似，语法略有不同，理解它最好的方式就是亲自来实现一下：

```
class EventEmitter {
  constructor() {
    this.events = {};
  }
  checkExistence(event) {
    if (!this.events[event]) {
      this.events[event] = [];
    }
  }
  once(event, cb) {
    this.checkExistence(event);
    const cbwithRemove = (...args) => {
      cb(...args);
      this.off(event, cbwithRemove);
    };
    this.events[event].push(cbwithRemove);
  }
  on(event, cb) {
    this.checkExistence(event);
    this.events[event].push(cb);
  }
  off(event, cb) {
    this.checkExistence(event);
    this.events[event] = this.events[event].filter(
      registeredCallback => registeredCallback !== cb
    );
  }
}
```

```
emit(event, ...args) {
  this.checkExistence(event);
  this.events[event].forEach(cb => cb(...args));
}
}
```

以上代码只显示模式本身，并没有针对确切的功能 - 请不要在您的代码中使用它！

这是我们需要的所有基础代码！它允许您订阅事件，稍后取消订阅，并派发不同的事件。例如，响应体，请求体，流 - 它们实际上都扩展或实现了EventEmitter！

正因为它是一个如此简单的概念，所以被用于许多的NPM包。所以，如果你想在浏览器中使用相同的事件发射器，可以随时使用它们。

流

“Streams是Node.js最好用、最容易被误解的概念。”

多米尼克塔尔(Dominic Tarr)

Streams允许您以块的形式来处理数据，而不仅仅是完整操作（如读取文件）。为了理解它们的作用，让我们来看个简单的例子：假设我们想要向用户返回任意大小的请求文件。我们的代码可能如下所示：

```
function (req, res) {
  const filename = req.url.slice(1);
  fs.readFile(filename, (err, data) => {
    if (err) {
      res.statusCode = 500;
      res.end('Something went wrong');
    } else {
      res.end(data);
    }
  });
}
```

这段代码可以使用，特别是在本地开发的机器上，但它可也会失败 - 您看出问题了吗？如果文件太大，我们读取文件时就会遇到问题，我们将所有内容放入内存中，如果没有足够的内存空间，这将无法正常工作。如果我们有很多并发请求，这段代码也不会生效 - 我们必须将数据对象保留在内存中，直到我们发送了所有内容。

然而，我们根本不需要这个文件 - 我们只需要从文件系统返回它，我们自己不会查看内容，所以我们可以读取它的一部分，立即返回给客户端来释放我们的内存，重复这样一个过程，直到我们完成了整个文件的发送。这是对 **Streams** 的简短介绍 - 我们有一种以块的形式来接收数据的机制，并且 *我们* 决定如何处理这些数据。例如，我们同样可以这样处理：

```
function (req, res) {
  const filename = req.url.slice(1);
  const filestream = fs.createReadStream(filename, { encoding: 'utf-8' });
  let result = '';
  filestream.on('data', chunk => {
    result += chunk;
  });
  filestream.on('end', () => {
    res.end(result);
  });
}
```

```
});
// if file does not exist, error callback will be called
filestream.on('error', () => {
  res.statusCode = 500;
  res.end('Something went wrong');
});
}
```

这里我们创建一个流来读取文件 - 这个流执行EventEmitter这个类，在 data 事件上我们接收下一个块，在 end 事件中，我们得到一个信号，表示流已结束，然后读取完整文件。这样的实现跟前面的一样 - 我们等待整个文件被读取，然后在响应中返回它。此外，它也有同样的问题：我们将整个文件保留在内存中，然后再发送回来。如果我们知道响应对象本身实现了可写流，我们可以解决这个问题，我们可以将信息写入该流而不将其保留在内存中：

```
function (req, res) {
  const filename = req.url.slice(1);
  const filestream = fs.createReadStream(filename, { encoding: 'utf-8' });
  filestream.on('data', chunk => {
    res.write(chunk);
  });
  filestream.on('end', () => {
    res.end();
  });
  // if file does not exist, error callback will be called
  filestream.on('error', () => {
    res.statusCode = 500;
    res.end('Something went wrong');
  });
}
```

响应体实现可写流，fs.createReadStream 创建可读流，还有双向和转换流。它们之间的区别以及工作原理，不在本教程的范围内，但是了解它们的存在还是大有裨益的。

这样我们不再需要结果变量了，只需要把已读的块立即写入响应体，不将它保留在内存中！这意味着我们甚至可以读取大文件，而不必担心高并发请求 - 因为文件没有被保存在内存中，所以不会超出内存所能承载的数量。但是，存在一个问题。在我们的解决方案中，我们从一个流（文件系统读取文件）中读取文件，并将其写入另一个（网络请求），这两个事物具有不同的延迟。这里强调是真的不同，经过一段时间后，我们的响应流将不堪重负，因为它要慢得多。这个问题是对背压的描述，Node有一个解决方案：每个可读流都有一个管道方法，它将所有数据重定向到与其负载相关的给定流中：如果它正忙，它将暂停原始流并恢复它。使用此方法，我们可以将代码简化为：

```
function (req, res) {
  const filename = req.url.slice(1);
  const filestream = fs.createReadStream(filename, { encoding: 'utf-8' });
  filestream.pipe(res);
  // if file does not exist, error callback will be called
  filestream.on('error', () => {
    res.statusCode = 500;
    res.end('Something went wrong');
  });
}
```


在Node的历史进程中，Streams改变了几次，所以在阅读旧手册时要格外小心，并经常查看官方文档！

模块系统

Node.js使用commonjs模块。您或许使用过 - 每次使用require来获取webpack配置中的某个模块时，您实际上就使用了commonjs模块；每次声明 `module.exports` 时也在使用它。然而，您可能还会看到像 `exports.some = {}` 这样的写法，没有 `module`，在这一节中我们将看下它究竟是如何工作的。

首先，我们来讨论commonjs模块，它们通常都有 `.js` 的扩展，而不是 `.esm / .mjs` 文件（ECMAScript模块），它们允许您使用 `import/export` 的语法。另外，重要的是要明白，webpack和browserify（以及其它打包工具）使用自己的 `require` 函数，所以请不要混淆 - 这里不讲解它们，只要明白它们是不同的东西就行（即使它们表现得非常相似）。

那么，我们实际上是在哪里获得这些“全局”对象，如 `module`，`require` 和 `exports`？实际上，是Node.js在运行时添加的 - 它不是仅执行给定的javascript文件，实际上是将它包含在具有所有这些变量的函数中：

```
function (exports, require, module, __filename, __dirname) {  
  // your module  
}
```

您可以在命令行中执行以下代码段来查看这个包：

```
1node -e "console.log(require('module').wrapper)"
```

这些是注入到模块中的变量，可以作为“全局”变量使用，即使它们不是真正的全局变量。我强烈建议你研究它们，尤其是模块变量。你可以在javascript文件中调用 `console.log (module)`，对比从 `main` 文件打印和从 `required` 的文件打印出来的结果。

接下来，让我们看一下 `exports` 对象 - 这里有一个小例子，显示一些与之相关的警告：

```
exports.name = 'our name';  
// this works  
  
exports = { name: 'our name' };  
// this doesn't work  
  
module.exports = { name: 'our name' };  
// this works!
```

上面的例子可能会让你感到困惑 为什么会这样？答案是 `exports` 对象的本质 - 它只是一个传递给函数的参数，所以在我们给它指定一个新对象的情况时，我们只是重写这个变量，旧的引用就不存在了。尽管它没有完全消失 - `module.exports` 是同一个对象 - 所以它们实际上是对单个对象的相同引用：

```
module.exports === exports;  
// true
```

最后一部分是 `require` - 它是一个获取模块名称并返回该模块的 `exports`对象的函数。它究竟是如何解析模块的？有一个非常简单的规则：

- 根据名称检索核心模块
- 如果路径以 `./` 或 `../` 开头，则尝试解析文件

- 如果找不到文件，尝试在其中找到包含 `index.js` 文件的目录
- 如果 `path` 不以 `./` 或 `../` 开头，请转到 `node_modules /` 并检查文件夹/文件：
 - 在我们运行脚本的文件夹中
 - 上面一级，直到我们到达 `/ node_modules`

还有其它一些位置，主要是为了兼容性，您还可以通过指定变量 `NODE_PATH` 来提供查找路径，这也许很有用。如果您要查看解析 `node_modules` 的确切顺序，只需在脚本中打印模块对象并查找 `paths` 属性。我操作后，打印了如下内容：

```
→ tmp node test.js

Module {
  id: '.',
  exports: {},
  parent: null,
  filename: '/Users/seva.zaikov/tmp/test.js',
  loaded: false,
  children: [],
  paths:
    [ '/Users/seva.zaikov/tmp/node_modules',
      '/Users/seva.zaikov/node_modules',
      '/Users/node_modules',
      '/node_modules' ] }
```

关于 `require` 的另一个有趣的事情是，在第一个 `require` 调用模块被缓存后，将不会再次执行，我们将只返回缓存的 `export` 对象 - 这意味着你可以做一些逻辑并确保它会在第一次 `require` 调用之后只执行一次（这不完全正确 - 如果再次需要，你可以从 `require.cache` 中删除模块 `id`，然后重新加载模块）

环境变量

正如在 十二因素应用程序 所述，将配置存储在环境变量中是一种很好的做法。您可以为 `shell` 会话设置变量：

```
export MY_VARIABLE="some variable value"
```

Node 是一个跨平台引擎，理想情况下，您的应用程序应该可以在任何平台上运行（例如，开发环境。您选择生产环境来运行您的代码，通常它是一些 Linux 分发版）。我的示例仅涵盖 MacOS / Linux，不适用于 Windows。Windows 中环境变量的语法跟这里的不同，你可以使用像 `cross-env` 这样的东西，但在其它情况下，你也应该记住这点。

您可以把下面这行代码添加到 `bash / zsh` 配置文件中，以便在任何新的终端会话中进行设置。然而，您通常只在运行应用程序时，为这些实例提供特有的变量：

```
APP_DB_URI="...." SECRET_KEY="secret key value" node server.js
```

您可以使用 `process.env` 对象来访问 Node.js 应用程序中的这些变量：

```
const CONFIG = {
  db: process.env.APP_DB_URI,
  secret: process.env.SECRET_KEY
}
```

综合运用

在下面的例子中，我们将创建一个简单的http服务，它将返回一个文件，以url / 后面的字符串来命名。如果文件不存在，我们将返回 `404 Not Found` 的错误信息，如果用户试图投机取巧，使用相对路径或嵌套路径，我们则返回403错误。我们之前使用过其中的一些函数，但没有真正记录它们 - 这次它将包含大量的信息：

```
// we require only built-in modules, so Node.js
// does not traverse our node_modules folders
// https://nodejs.org/api/http.html#http_http_createserver_options_requestlistener

const { createServer } = require("http");
const fs = require("fs");
const url = require("url");
const path = require("path");

// we pass the folder name with files as an environment variable
// so we can use a different folder locally

const FOLDER_NAME = process.env.FOLDER_NAME;
const PORT = process.env.PORT || 8080;
const server = createServer((req, res) => {
  // req.url contains full url, with querystring
  // we ignored it before, but here we want to ensure
  // that we only get pathname, without querystring
  // https://nodejs.org/api/http.html#http_message_url

  const parsedURL = url.parse(req.url);

  // we don't need the first / symbol
  const pathname = parsedURL.pathname.slice(1);

  // in order to return a response, we have to call res.end()
  // https://nodejs.org/api/http.html#http_response_end_data_encoding_callback
  //
  // > The method, response.end(), MUST be called on each response.
  // if we don't call it, the connection won't close and a requester
  // will wait for it until the timeout
  //
  // by default, we return a response with [code 200]
  // (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)
  // in case something went wrong, we are supposed to return
  // a correct status code, using the res.statusCode = ... property:
  // https://nodejs.org/api/http.html#http_response_statuscode

  if (pathname.startsWith(".")) {
    res.statusCode = 403;
    res.end("Relative paths are not allowed");
  } else if (pathname.includes("/")) {
    res.statusCode = 403;
    res.end("Nested paths are not allowed");
  } else {
    // https://nodejs.org/en/docs/guides/working-with-different-file-systems/
```

```
// in order to stay cross-platform, we can't just create a path on our own
// we have to use the platform-specific separator as a delimiter
// path.join() does exactly that for us:
// https://nodejs.org/api/path.html#path_path_join_paths
const filePath = path.join(__dirname, FOLDER_NAME, pathname);
const fileStream = fs.createReadStream(filePath);
fileStream.pipe(res);
fileStream.on("error", e => {
  // we handle only non-existent files, but there are plenty
  // of possible error codes. you can get all common codes from the docs:
  // https://nodejs.org/api/errors.html#errors_common_system_errors

  if (e.code === "ENOENT") {
    res.statusCode = 404;
    res.end("This file does not exist.");
  } else {
    res.statusCode = 500;
    res.end("Internal server error");
  }
});
});
server.listen(PORT, () => {
  console.log(application is listening at the port ${PORT});
});
```

总结

在本指南中，我们介绍了许多基本的Node.js原则。我们没有深入研究特定的API，我们确实错过了一些东西。但是，本指南应该是一个很好的起点，让您在阅读API，编辑现有的代码，或者创建新脚本时有信心。您现在能够理解错误，清楚内置模块使用的接口，以及从典型的Node.js对象和接口中能获取到哪些东西。

下一次，我们将深入介绍使用Node.js的Web服务，Node.js REPL，如何编写CLI应用程序，以及如何使用Node.js编写小脚本。您可以订阅以获取有关这些新文章的通知。

相关文章

2017年7月9日» Node.js REPL深度

2018年6月5日» 不要使用缩略词

2018年6月3日» 单元测试