

Node.js

从入门到“精通”

目录

content

- NPM
- CommonJS
- 核心模块
- process
- Http框架
- 工具库介绍

Hello World



```
console.log('hello node.js');  
process.stdout.write('hello node.js\n');
```

NPM

常用命令 / npx / nrm / nvm / 第三方工具

常用命令



创建 package.json 文件

```
$ npm init -y
```

安装一个 node 包

```
$ npm install <name>
```

安装一个 node 包到 dependencies

```
$ npm install <name> --save
```

安装一个 node 包到 devDependencies

```
$ npm install <name> --save-dev
```

全局安装一个包

```
$ npm install -g <name>
```

卸载一个node包

```
$ npm uninstall <name>
```

NPX

包管理器，运行使用 **Node.js** 构建并通过 **NPM** 仓库发布的代码



```
$ npx cnpm install vue --save
```

NRM

镜像源管理工具，可以快速地在 **NPM** 源间切换



```
# 全局安装nrm
$ npm install -g nrm

# 查看可选的源
$ nrm ls

# 使用 taobao 源
$ nrm use taobao

# 使用官方源
$ nrm use npm
```

NVM

版本管理工具，在同一台机器安装多个版本的 Node.js



```
# 安装最新版本
$ nvm install node

# 安装指定版本
$ nvm install 0.12.1

# 使用已安装的最新版本
$ nvm use node

# 进入指定版本的REPL环境
$ nvm run 0.12
```


第三方工具

- **Yarn** 由 Facebook 提供，相比 **NPM** 速度更快
- **cnpm** 国内淘宝镜像
- **pnpm** 运行超快，对比 **Yarn** 坑比较深

CommonJS

概念 / 例子

概念

- Node.js 采用模块化结构，按照 CommonJS 规范定义和使用模块。模块与文件是一一对应关系，即加载一个模块，实际上就是加载对应的一个模块文件。
- require 命令用于指定加载模块，加载时可以省略脚本文件的后缀名。



```
// circle.js
module.exports = function(x) {
  console.log(x);
};

// index.js
var circle = require('./circle');
circle('这是自定义模块');
```

module.exports 和 export 有什么区别？

前者导出了它指向的对象。 后者导出了它指向的对象的属性。 — 鲁迅


核心模块

fs / path / events / Buffer / stream

fs

- 读取文件
- 写文件
- **fs.unlinkSync** 删除文件
- **fs.existsSync** 判断文件是否存在，返回 **boolean**
- **fs.createReadStream** 创建一个可读文件流
- **fs.createWriteStream** 创建一个可写文件流

读取文件



```
/* 方法一 */
// 打开文件
const fd = fs.openSync('./test.txt', 'r');
// 申请buffer空间
const buff = Buffer.alloc(1024);
// 读取文件句柄
fs.readSync(fd, buff, 0, 1024, null);
// 输出文件
process.stdout.write(buff);
// 关闭文件
fs.closeSync(fd);

/* 方法二 */
const buff2 = fs.readFileSync('./test.txt');
process.stdout.write(buff2);
```


写文件




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

/* 方法一 */
// 打开文件
const fd = fs.openSync('./test.txt', 'w');
// 将 "hello node.js" 写入缓冲区
const buff = Buffer.from('hello node.js');
// 再将缓冲写入文件
fs.writeFileSync(fd, buff);
// 关闭文件
fs.closeSync(fd);

/* 方法二 */
fs.writeFileSync('./test.txt', 'hello node.js');
```

path

- **__dirname** 返回当前模块（文件）所在文件夹的绝对路径（**常量**）
- **__filename** 返回当前模块（文件）所在文件的绝对路径（**常量**）
- **path.resolve** 通过相对路径计算出绝对路径
- **path.join** 连接路径的两个或多个部分
- **path.parse** 解析对象的路径为组成其的片段



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

console.log(__dirname);
// /Users/denglei/Workspace/node_learn
console.log(__filename);
// /Users/denglei/Workspace/node_learn/path-demo.js

const nowDirPath = path.resolve(__dirname);
console.log(nowDirPath);
// /Users/denglei/Workspace/node_learn

const nowFilePath = path.resolve(__filename);
console.log(nowFilePath);
// /Users/denglei/Workspace/node_learn/path-demo.js

const joinRes = path.join(__dirname, __filename);
console.log(joinRes);
///Users/denglei/Workspace/node_learn/Users/denglei/Workspace/node_learn/path-demo.js

const parser = path.parse(__filename);
console.log(parser);
// {
//   root: '/', 根路径
//   dir: '/Users/denglei/Workspace/node_learn', 从根路径开始的文件夹路径
//   base: 'path-demo.js', 文件名 + 扩展名
//   ext: '.js', 文件扩展名
//   name: 'path-demo' 文件名
// }
```

events

概念

- `Events` 是 `Node.js` 中一个 **非常重要** 的 core 模块，在 node 中有许多重要的 core API 都是依赖其建立的。
- 比如 `Stream` 是基于 `Events` 实现的，而 `fs`，`net`，`http` 等模块都依赖 `Stream`，所以 `Events` 模块的重要性可见一斑。

events

函数介绍

- **Events.on()** 监听一个事件
- **Events.once()** 监听一个事件，只触发一次
- **Events.off()** 卸载一个监听的事件
- **Events.addListener()** **on** 的别名
- **Events.removeListener()** **off** 的别名
- **Events.removeAllListeners()** 移除所有监听特定事件的监听器
- **Events.emit()** 触发事件



```
const { EventEmitter } = require('events');
```

```
const ee = new EventEmitter();
```

```
// 监听一个my-event事件
```

```
ee.on('my-event', () => {  
    console.log('my-event');  
});
```

```
// 触发my-event事件
```

```
ee.emit('my-event');
```

Eventemitter 的 emit 是同步还是异步?

Node.js 中 Eventemitter 的 emit 是同步的。 — 鲁迅

Buffer

概念

- Buffer 是内存区域 Node.js 中用于处理二进制数据的类，其中与 IO 相关的操作 (网络/文件等) 均基于 Buffer。
- 可以将 Buffer 视为整数数组，每个整数代表一个数据字节。
- 它是 Node.js 原生提供的全局对象，可以直接使用，不需要 require()。

函数介绍

- **Buffer.from** 根据已有数据生成一个 Buffer 对象
- **Buffer.alloc** 创建一个初始化后的 Buffer 对象
- **Buffer.allocUnsafe** 创建一个未初始化的 Buffer 对象



```
// 初始化一个 “hello node.js” buffer
const buff = Buffer.from('hello node.js');

// 创建一个 1KB 的 buffer
const buff2 = Buffer.alloc(1024);
// 向 buffer 内写入 “hello node.js”
buff2.write('hello node.js');

// 创建一个 1KB 的 不安全buffer
const buff3 = Buffer.allocUnsafe(1024);
buff3.write('hello node.js');

process.stdout.write(buff); // hello node.js
process.stdout.write(buff2); // hello node.js
process.stdout.write(buff3); // hello node.js (后面还可能带有一下未知的数据)
```

怎么读写 **100GB** 文件呢？

我们紧张的往下看

stream

概念

- 流是为 Node.js 应用程序提供动力的基本概念之一。
- 它们是一种以高效的方式处理读/写文件、网络通信、或任何类型的端到端的信息交换。
- 它们是几十年前在 Unix 操作系统中引入的，程序可以通过管道运算符（|）对流进行相互交互。
- 例如，在传统的方式中，当告诉程序读取文件时，这会将文件从头到尾读入内存，然后进行处理。使用流，则可以逐个片段地读取并处理（而无需全部保存在内存中）。
- 所有的流都是 EventEmitter 的实例

stream

类型说明

- Readable | 只读 | _read
- Writable | 只写 | _write
- Duplex | 读写 | _read, _write
- Transform | 操作被写入数据, 然后读出结果 | _transform, _flush

stream

函数介绍

- **Writable.write** 写入数据到流
- **Writable.end** 结束写入流
- **Writable.setDefaultEncoding** 为可写流设置默认的 encoding
- **Readable.read** 从内部缓冲拉取并返回数据
- **Readable.pipe** 绑定 可写流 到 可读流

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

function readStream(request, response) {
  const fStream = fs.createReadStream('./test.txt');
  fStream.pipe(response);

  fStream.on('end', () => {
    console.log('read end');
  });
}

function writeStream(request, response) {
  const fStream = fs.createWriteStream('./write.txt', {
    flags: 'a+'
  });
  request.pipe(fStream);

  request.on('end', () => {
    console.log('write end');
    response.write('ok');
    response.end();
  });
}

const server = http.createServer((request, response) => {
  if (request.url === '/write') {
    writeStream(request, response);
  } else {
    readStream(request, response);
  }
});

server.listen(8080, 'localhost');
```

process

概念 / 函数介绍 / 例子

process

概念

- `process` 对象是一个全局变量
- 提供了有关当前 `Node.js` 进程的信息并对其进行控制。
- 它始终可供 `Node.js` 应用程序使用，无需使用 `require()`

process

函数介绍

- **process.env** 返回包含用户环境的对象
- **process.argv** 返回一个数组，包含启动时传入的命令行参数
- **process.exit** 以退出状态 **code** 指示 Node.js 同步地终止进程
- **process.stdin** 返回连接到 **stdin** (fd 0) 的流
- **process.stdout** 返回连接到 **stdout** (fd 1) 的流
- **process.stderr** 返回连接到 **stderr** (fd 2) 的流
- **process.nextTick** 将 **callback** 添加到下一个时间点的队列
- **process.on('uncaughtException')** 当未捕获的 JavaScript 异常时触发
- **process.on('unhandledRejection')** **Promise** 被拒绝，并且没有绑定错误处理器时触发



```
const env = process.env;
console.log(env);

const argv = process.argv;
console.log(argv);
// [
//     '/usr/local/Cellar/node/13.11.0/bin/node',
//     '/Users/denglei/Workspace/node_learn/process-demo.js'
// ]

process.stdout.write('process-demo.js \n');
process.stderr.write('process-demo1.js \n');

process.nextTick(() => {
    console.log('nextTick');
});

process.on('uncaughtException', e => {
    console.error(e);
    process.exit(0);
});

process.on('unhandledRejection', e => {
    console.error(e);
    process.exit(0);
});
```

HTTP框架

[http模块](#) / [express](#) / [更多](#)

http模块

- **http.createServer** 创建 http 服务器，并返回一个 **net.Server** 实例
- **http.get** 发送一个 get 请求
- **http.request** 发送一个 http 请求



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

// 创建一个http服务器
const server = http.createServer((request, response) => {
  response.statusCode = 200;
  response.setHeader('content-type', 'text/plain');
  response.end('hello node.js');
});

// 监听 127.0.0.1:8080 端口
server.listen(8080, '127.0.0.1');

http.get('https://www.baidu.com', res => {
  res.pipe(process.stdout);
});

http.request({
  host: 'https://www.baidu.com/',
  method: 'POST'
}, res => {
  res.pipe(process.stdout);
});
```

Express

基于 Node.js 平台，快速、开放、极简的 Web 开发框架



```
const express = require('express');
const app = express();

// 监听一个 / get请求
app.get('/', (request, response) => {
  response.status(200).send('hello express, get method');
});

// 监听一个 / post请求
app.post('/', (request, response) => {
  response.status(200).send('hello express, post method')
});

// 监听本地 8080 端口
app.listen(8080);
```

更多

- **Koa.js** 由 Express 团队开发，轻量级高效 Web 开发框架
- **Egg.js** 企业级开发框架
- **Nest.js** 渐进式 Web 开发框架 (TypeScript)

工具库介绍

工具库介绍

- **pm2** node进程管理工具
- **fs-extra** 文件操作拓展工具
- **cross-env** 跨平台设置环境变量工具
- **commander.js** 命令行解决方案
- **shell.js** unix shell命令工具
- **inquirer.js** 交互式命令行美化工具
- **mongoose.js** 连接 **mongodb** 数据库

THINKS

邓六石 / 2020.11.16