Node.js 从入门到 “精通” 🐶

NPM

NPM 是 Node.js 的包管理工具，用来安装各种 Node.js 的扩展

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>
```

package.json 文件

- **name** 设置了应用程序/软件包的名称。
- **version** 表明了当前的版本。
- **description** 是应用程序/软件包的简短描述。
- **private** 如果设置为 true，则可以防止应用程序/软件包被意外地发布到 npm。
- **scripts** 定义了一组可以运行的 node 脚本。
- **dependencies** 设置了作为依赖安装的 npm 软件包的列表。
- **devDependencies** 设置了作为开发依赖安装的 npm 软件包的列表。

dependencies

> 设置作为依赖安装的 npm 软件包的列表。

devDependencies

> 设置作为开发依赖安装的 npm 软件包的列表。 它们不同于 dependencies，因为它们只需安装在开发机器上，而无需在生产环境中运行代码。

npx 包运行器

`npx` 可以运行使用 Node.js 构建并通过 npm 仓库发布的代码。

例子：

```
// package.json 文件
{
    "cnpm": "^6.1.1"
}
```

运行 `node_modules` 内的 `cnpm` 命令安装 `vue`

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

nvm 版本管理工具

在同一台机器，同时安装多个版本的node.js，就需要用到版本管理工具nvm。

```
# 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

# 使用指定版本的node
$ nvm use 0.12

# 进入指定版本的REPL环境
$ nvm run 0.12
```

nrm 源管理工具

nrm(npm registry manager)是npm的镜像源管理工具，有时候国外资源太慢，使用这个就可以快速地在 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
```

yarn / cnpm /pnpm 第三方包管理工具

yarn / cnpm / pnpm 区别，文档链接

commonJS 模块化结构

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');
```

运行 `index.js` 脚本

```
$ node index
```

`module.exports` 和 `export` 之间有什么区别?

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。

核心模块

fs

`fs` 模块可用于与文件系统进行交互

- **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.writeSync(fd, buff);
// 关闭文件
fs.closeSync(fd);

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

- **更多操作**

方法	用途
`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

path

方法	用途
`__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）所在文件的绝地路径 （ `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

例子：

```
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 事件模块

`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。

方法	用途
`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

例子：

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

const ee = new EventEmitter();

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

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

Eventemitter 的 emit 是同步还是异步?

`Node.js` 中 `Eventemitter` 的 `emit` 是同步的。在官方文档中有说明：

> The `EventEmitter` calls all listeners synchronously in the order in which they were registered. This ensures the proper sequencing of events and helps avoid race conditions and logic errors.

Buffer

什么是Buffer?

`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('buffer')` 。

创建Buffer

方法	用途
`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

例子：

```
// 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）
```

stream（流）

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

流的类型

类	使用场景	重写方法
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

常用函数

方法	用途
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`

如何读取 / 写入 `100GB` 文件到客户端（服务端）？

例子：

```
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 进程

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。 作为全局变量，它始终可供 `Node.js` 应用程序使用，无需使用 `require()` 。

方法	用途
`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，并且此 `Promise` 没有绑定错误处理器时触发

例子：

```
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);
});
```

事件循环

消息队列

作业队列

setTimeout

process.nextTick

setImmediate

Http框架

http / https 原生模块

方法	用途
`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

```
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`
- `egg.js`
- `nest.js`

Node.js 工具

pm2 进程守护工具

fs-extra 文件拓展工具

cross-env 环境变量注入工具

commander.js

shell.js

inquirer.js

上手开发

RESTful

连接数据库

进阶

chrome debug

vs code debug

内存分析

C++ 原生插件开发

学习资源

Node.js 文档阅读 “奇技淫巧”