

# 目 录

致谢

介绍

1. 搭建Node.js开发环境
  2. 了解并使用Http模块
  3. Node模块与npm
  4. 搭建静态文件服务器
  5. Node中的stream (流)
  6. Node的readline (逐行读取)
  6. node命令行工具开发
  7. Node中的网络编程
  8. Node操作数据库
  8. Node操作MySQL
  8. Node操作MongoDB数据库
  9. Koa快速入门教程
  10. Node.js使用Nodemailer发送邮件
  11. node爬虫：送你一大波美腿图
- Node相关入门资料

## 致谢

当前文档《Node.js入门教程》由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建，生成于 2006-01-02。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/node-abc>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

## 介绍

- [Node.js入门教程](#)
  - [介绍](#)
  - [目录](#)
  - [more](#)
  - [联系作者](#)
  - [微信公众号：JavaScript之禅](#)
  - [License](#)
  - [来源\(书栈小编注\)](#)

## Node.js入门教程

学习Node.js有段时间了，入门时很迷茫。于是想以自己的入门水平来写这教程给未入门的朋友。希望能够帮助到更多刚入门朋友少一点儿迷茫。

由于水平有限，编写的过程难免会有诸多错误，也希望大家在看的过程中发现了问题及时联系

抛砖引玉

## 介绍

这个课程不是文档式的，也不是纯案例。而是在学习每个知识点后，提供个Node.js 实战例子来稳固这些知识点，希望可以通过每一节精心安排的课程，让 Node.js 的初学者们可以循序渐进地，有目的地开展 Node.js 的学习，少一点儿迷茫。

## 目录

- Lesson1 - [搭建Node.js开发环境](#)

- Lesson2 - [了解并使用Http模块](#)
- Lesson3 - [Node模块与npm](#)
- Lesson4 - [搭建静态文件服务器](#)
- Lesson5 - [Node中的stream（流）](#)
- Lesson6 - [Node的readline（逐行读取）](#)
- Lesson7 - [Node中的网络编程](#)
- Lesson8 - [Node操作数据库](#)
- Lesson9 - [Koa快速入门教程](#)
- Lesson10 - [Node.js使用Nodemailer发送邮件](#)
- Lesson10 - [node爬虫：送你一大波美腿图](#)

more

---

- [Node相关入门资料](#)

## 联系作者

---

- 邮箱: [ogilhinn@gmail.com](mailto:ogilhinn@gmail.com)
- 博客: [刘星的博客](#)

微信公众号: JavaScript之禅

---



## License

---

本作品采用[知识共享署名-非商业性使用 4.0 国际许可协议](#)进行许可。



## 来源(书栈小编注)

---

<https://github.com/ogilhinn/node-abc>

## 1. 搭建Node.js开发环境

- 【Node入门教程】搭建Node.js开发环境
  - 安装配置
    - Windows和mac平台
    - Linux平台(Ubuntu)
      - Node.js官网提供的安装方式
      - 从源代码安装Node.js
      - apt-get安装
      - 使用 nvm
        - 安装nvm
        - nvm使用
  - hello world
  - 一些有用的工具
  - 总结

## 【Node入门教程】搭建Node.js开发环境

---

### HELLO WORLD

本章节我们将向大家介绍在各个平台上(win, mac与ubuntu)安装Node.js的方法。本安装教程以Latest LTS Version: **v6.10.2** (includes npm 3.10.10)版本为例

## 安装配置

---

要学习一门语言，我们首先应该去它的官网逛逛，Node.js的官网地址为<https://nodejs.org>（现在腾讯团队翻译的Node.js中文网貌似也进行的差不多了）

进入到Node.js的官网，我们点击Download可以看到各个平台Node.js的安装包，现在我们就来看看如何在各个平台安装Node（当


然官网也提供了详细的[安装指引](#) )


## Downloads


Latest LTS Version: v6.10.2 (includes npm 3.10.10)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

**LTS**  
Recommended For Most Users  
  
**Current**  
Latest Features

  
**Windows Installer**  
node-v6.10.2-x86.msi

  
**Macintosh Installer**  
node-v6.10.2.pkg

  
**Source Code**  
node-v6.10.2.tar.gz

[Windows Installer \(.msi\)](#)  
[Windows Binary \(.zip\)](#)  
[macOS Installer \(.pkg\)](#)  
[macOS Binaries \(.tar.gz\)](#)  
[Linux Binaries \(x86/x64\)](#)  
[Linux Binaries \(ARM\)](#)  
[Source Code](#)

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	
32-bit	64-bit
ARMv6	ARMv7
	ARMv8
node-v6.10.2.tar.gz	

## Windows和mac平台

其实windows和mac上安装Node.js没有什么好说的，和安装其他软件一样，同意协议然后一直点击下一步就好了。Duang的一下安装完后，我们就可以打开命令行查看是否安装成功

1. `$ node -v`
2. `v6.10.2` #如果出现如下结果，那么恭喜你安装成功了

## Linux平台(Ubuntu)

相比于Windows与mac，Linux平台的安装还是有些许繁琐，但无非也就是使用命令行

- Node.js官网提供的安装方式

1. `$ curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -`

```
2. $ sudo apt-get install -y nodejs
```

当然了可能会不成功，不要怕还可以试试我们其他方式

## • 从源代码安装Node.js

首先我们更新下系统，并下载编译需要的包

```
1. $ apt-get update
2. $ apt-get install python gcc make g++
```

然后使用wget下载源码包

```
1. $ wget https://nodejs.org/dist/v6.10.2/node-v6.10.2-linux-x64.tar.gz
2. # 移动目录
3. $ cp node-v6.10.2-linux-x64.tar.gz /usr/local/src/
```

现在我们解压源代码，并进入目录下开始编译

```
1. $ tar zxvf node-v0.12.4.tar.gz
2. $ cd node-v0.12.4/
3. $ ./configure
4. $ make install
```

最后我们 `node -v` 现在我们应该可以看见Node.js的版本号

上述每一步操作注意权限问题\*

## • apt-get安装

我们也有简单的选择 `apt-get`

```
1. $ sudo apt-get install nodejs
```



但是 `apt-get` 安装有一个问题就是版本有点老

- 使用 [nvm](#)

这放在最后说就说明这玩意儿不简单，毕竟重量级选手都最后出场。

nvm是Node版本管理器：nvm。简单的bash脚本来管理多个活跃的node.js版本，与nvm类似的还有n模块

安装nvm

我们可以使用curl或者wget安装

```
1. curl -o-  
    https://raw.githubusercontent.com/creationix/nvm/v0.33.1/install  
    | bash
```

```
1. wget -qO-  
    https://raw.githubusercontent.com/creationix/nvm/v0.33.1/install  
    | bash
```

nvm使用

使用nvm可以方便的下载安装删除各个版本的Node.js

```
1. nvm install stable #安装最新稳定版 node, 现在是 5.0.0  
2. nvm install 4.2.2 #安装 4.2.2 版本  
3. nvm install 0.12.7 #安装 0.12.7 版本  
4.  
5. # 特别说明：以下模块安装仅供演示说明，并非必须安装模块  
6. nvm use 0 #切换至 0.12.7 版本  
7. nvm use 4 #切换至 4.2.2 版本
```

具体使用请参考[nvm官网](#)

# hello world

---

到此想必各位看官已经在自己的电脑上安装好了Node.js开发环境（我想大家没看这个就已经安好了，安装指引只是例行公事 ~.~）

下面我们来一个hello world开启本次学习之路

这是官网的一个例子，一个使用 Node.js 编写的 web 服务器，响应返回 'Hello World'

```
1. const http = require('http');
2.
3. const hostname = '127.0.0.1';
4. const port = 3000;
5.
6. const server = http.createServer((req, res) => {
7.   res.statusCode = 200;
8.   res.setHeader('Content-Type', 'text/plain');
9.   res.end('Hello World\n');
10. });
11.
12. server.listen(port, hostname, () => {
13.   console.log(`服务器运行在 http://${hostname}:${port}/`);
14. });
```

## 一些有用的工具

---

- [nrm](#) 快速切换 NPM 源
- [cnpm](#) 淘宝 NPM 镜像
- [supervisor](#) Supervisor实现监测文件修改并自动重启应用

## 总结

---

### 1. 在本机成功安装Node.js

2. 用http模块起一个http服务器，打开Node的大门
3. 了解一些有用的模块（详细使用请自行了解）

既然这节用http模块说了hello world，下一节，我们就来一起看看http模块

## 2. 了解并使用Http模块

- Node入门教程 - 了解并使用Http模块
  - 1.Http服务器
    - 获取客户端请求信息
      - request对象
    - response对象
    - URL解析
      - `querystring` 模块 用于 URL 处理与解析
      - `url` 模块提供了一些实用函数，用于 URL 处理与解析
    - 实例：HTTP JSON API 服务器
  - 2.Http客户端
  - 3.一些好用的包
  - 4.总结

## Node入门教程 - 了解并使用Http模块

传送门: [GitHub地址](#)

上一节，我们用`Http`模块搭建了一个`hello world`服务器。现在我们就来了解了解`Http`模块，学习它的常用API，并在最后实现两个小案例。

`http`模块主要用于搭建HTTP服务。使用Node搭建HTTP服务器非常简单。

按理说，我们应该先讲讲NPM，`package.json`。但我觉得先讲讲`Http`模块实现两个小案例，可以让各位看官更有兴趣，能够愉悦的学习使用`Node.js`

我想你可能已经学会了看官网的文档，但脑子在想这是什么鬼，的确，`Node.js`的文档对初学者不太友好。很多用不上，例子太少。大体来说`Http`模块，主要的应用是两部分，一部分是`http.createServer` 担

当web服务器，另一部分是`http.createServer`，担当客户端，实现爬虫之类的工作。下文将从这两方面着手介绍HTTP api。

## 1.Http服务器

看上一节例子

```
1. const http = require('http')
2.
3. const hostname = '127.0.0.1'
4. const port = 3000
5.
6. const server = http.createServer((req, res) => {
7.   res.statusCode = 200
8.   res.setHeader('Content-Type', 'text/plain') //writeHead, 200表示页面正常, text/plain表示是文字。
9.   res.end('Hello World\n') // end 完成写入
10. })
11.
12. server.listen(port, hostname, () => {
13.   console.log(`服务器运行在 http://${hostname}:${port}`)
14. })
```

1. 首先 使用 HTTP 服务器和客户端必须 `require('http')`。
2. 然后使用 `http.createServer([requestListener])` 来创建一个web服务器，其中传入一个可选的回调函数，这回调函数有两个参数分别代表客户端请求与服务器端的响应对象
3. 使用 `server.listen([port][, hostname][, backlog][, callback])` 开始在指定的 port 和 hostname 上接受连接

简单的3步一个http服务器就建好，有打开就有关闭

```
1. server.close([callback]) // 停止服务端接收新的连接
```

Node.js Http模块也提供了 `server.timeout` 用于查看或设置超时

```
1. server.timeout = 1000 //设置超时为1秒
2. console.log(server.timeout)
```

## 获取客户端请求信息

### request对象

- `request.url` 客户端请求的url地址
- `request.headers` 客户端请求的http header
- `request.method` 获取请求的方式，一般有几个选项，POST, GET和DELETE等，服务器可以根据客户端的不同请求方法进行不同的处理。
- `request.httpVersion` http的版本
- `request.trailers` 存放附加的一些http头信息
- `request.socket` 用于监听客户端请求的socket对象

我们可以写一小段js讲客户端的请求信息保存在log.txt中

```
1. const http = require('http') //引入http模块
2. const fs = require('fs') //引入fs模块 文件 I/O
3. const hostname = '127.0.0.1'
4. const port = 3000
5.
6. const server = http.createServer((req, res) => { //创建一个http服务器
7.   res.statusCode = 200
8.   res.setHeader('Content-Type', 'text/plain')
9.   if(req.url !== '/favicon.ico'){
10.     let out = fs.createWriteStream('./log.txt') // 创建写入流
11.     out.write(`请求方法:${req.method} \n`)
12.     out.write(`请求url:${req.url} \n`)
13.     out.write(`请求头对象:${JSON.stringify(req.headers, null, 4)}
    \n`)
```

```

14.         out.write(`请求http版本: ${req.httpVersion} \n`)
15.     }
16.     res.end('Hello World\n')
17. })
18.
19. server.listen(port, hostname, () => {
20.     console.log(`服务器运行在 http://${hostname}:${port}`)
21. })
22.
23. log.txt
24. =====
25. 请求方法: GET
26. 请求url: /
27. 请求头对象: {
28.     "host": "127.0.0.1:3000",
29.     "connection": "keep-alive",
30.     "cache-control": "max-age=0",
31.     "upgrade-insecure-requests": "1",
32.     "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133
    Safari/537.36",
33.     "accept":
    "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp, */*
34.     "accept-encoding": "gzip, deflate, sdch, br",
35.     "accept-language": "zh-CN,zh;q=0.8,en;q=0.6"
36. }
37. 请求http版本: 1.1

```

## response对象

- `response.writeHead(statusCode, [reasonPhrase], [headers])`
- `response.statusCode` html页面状态值
- `response.header` 返回的http header, 可以是字符串, 也可以是对象
- `response.setTimeout(msecs, callback)` 设置http超时返回的时间,

一旦超过了设定时间，连接就会被丢弃

- `response.statusCode` 设置返回的网页状态码
- `response.setHeader(name, value)` 设置http协议头
- `response.headersSent` 判断是否设置了http的头
- `response.write(chunk, [encoding])` 返回的网页数据，  
[encoding] 默认是 utf-8
- `response.end([data], [encoding])`

## URL解析

在Node.js中，提供了一个 `url` 模块与 `querystring` (查询字符串) 模块

`querystring` 模块 用于 URL 处理与解析

1. `querystring.parse(str, [sep], [eq], [options])` // 将字符串转成对象

欲转换的字符串	str
设置分隔符，默认为 '&'	sep
设置赋值符，默认为 '='	eq
可接受字符串的最大长度，默认为1000	[options] maxKeys

```
1. // 例子:
2. querystring.parse('foo=bar&baz=qux&baz=quux&corge')
3. // returns
4. { foo: 'bar', baz: ['qux', 'quux'], corge: '' }
```

`querystring.stringify(obj, separator, eq, options)` 这个方法是将一个对象序列化成一个字符串，与 `querystring.parse` 相对。

```
1. querystring.stringify({name: 'whitemu', sex: [ 'man', 'women' ] });
2.
3. // returns
4. 'name=whitemu&sex=man&sex=women'
```



`url` 模块提供了一些实用函数，用于 URL 处理与解析

一个 URL 字符串是一个结构化的字符串，它包含多个有意义的组成部分。当被解析时，会返回一个 URL 对象，它包含每个组成部分作为属性。

以下详情描述了一个解析后的 URL 的每个组成部分



`url.format(urlObject)` 回一个从 `urlObject` 格式化后的 URL 字符串

`url.parse(urlString[, parseQueryString[, slashesDenoteHost]])` 解析一个 URL 字符串并返回一个 URL 对象

我们可以使用 `url.parse()` 解析出的对象来获取URL中的各个值

更多请参考[node中文网](#)

## 实例：HTTP JSON API 服务器

编写一个 HTTP 服务器，每当接收到一个路径为

`‘/api/parsetime’` 的 GET

请求的时候，响应一些 JSON 数据。我们期望请求会包含一个查询参数 (query

string)，key 是 `“iso”`，值是 ISO 格式的时间。

- `/api/parsetime?iso=2017-04-05T12:10:15.474Z`

所响应的 JSON 应该只包含三个属性：`‘hour’`，`‘minute’` 和 `‘second’`。例如：

```
1. {  
2.   "hour":21,  
3.   "minute":45,  
4.   "second":30  
5. }
```

- `/api/unixtime?iso=2017-04-05T12:10:15.474Z`，它的返回会包含一个属性：`‘unixtime’`，相应值是一个 UNIX

时间戳。例如：

```
1.   { "unixtime": 1376136615474 }
```

具体代码如下

```
1. const http = require('http')  
2. const url = require('url')  
3.  
4. const hostname = '127.0.0.1'
```

```
5.  const port = 3000
6.
7.  // 解析时间的函数
8.  function parsetime(time) {
9.      return {
10.          hour: time.getHours(),
11.          minute: time.getMinutes(),
12.          second: time.getSeconds()
13.      }
14.  }
15.
16.  function unixtime(time) {
17.      return { unixtime: time.getTime() }
18.  }
19.
20.  const server = http.createServer((req, res) => {
21.      let parsedUrl = url.parse(req.url, true)
22.      let time = new Date(parsedUrl.query.iso)
23.      let result
24.      // 主页 返回当前时间的json
25.      if(req.url=='/'){
26.          result = parsetime(new Date())
27.      }
28.      // 返回查询时间的json
29.      else if (/^\/api\/parsetime/.test(req.url)) {
30.          result = parsetime(time)
31.      }
32.      // 返回查询时间的unixtime
33.      else if (/^\/api\/unixtime/.test(req.url)) {
34.          result = unixtime(time)
35.      }
36.
37.      if (result) {
38.          res.writeHead(200, { 'Content-Type': 'application/json' })
39.          res.end(JSON.stringify(result))
40.      } else {
41.          res.writeHead(404)
42.          res.end()
```

```

43.     }
44. })
45.
46. server.listen(port, hostname, () => {
47.     console.log(`服务器运行在 http://${hostname}:${port}`)
48. })

```

## 2.Http客户端

在Node.js可以很容易的使用request方法想向其他网站求数据，也可以用 `http.get(options[, callback])`

```
1. http.request(options, callback)
```

request方法的options参数，可以是一个对象，也可以是一个字符串。如果是字符串，就表示这是一个URL，Node内部就会自动调用 `url.parse()`，处理这个参数。

`http.request()` 返回一个 `http.ClientRequest` 类的实例。它是一个可写数据流，如果你想通过POST方法发送一个文件，可以将文件写入这个ClientRequest对象

现在我们拿[www.example.com](http://www.example.com)试试

```

1. const http = require('http')
2. let options = {
3.     hostname: 'www.example.com',
4.     port: 80,
5.     path: '/',
6.     method: 'GET'
7. }
8.
9. const req = http.request(options, (res) => {
10.     console.log(`STATUS: ${res.statusCode}`) //返回状态码
11.     console.log(`HEADERS: ${JSON.stringify(res.headers, null, 4)}`)

```

```
// 返回头部
12.     res.setEncoding('utf8') // 设置编码
13.     res.on('data', (chunk) => { //监听 'data' 事件
14.         console.log(`主体: ${chunk}`)
15.     })
16.
17. })
18.
19. req.end() // end方法结束请求
```

到此我们请求到了网站上的信息，基于这些我们可以开发出更有用的爬虫，提取到有用的信息，后续将介绍

## 3. 一些好用的包

---

- [express](#) Express 是一个简洁而灵活的 node.js Web应用框架，提供了一系列强大特性帮助你创建各种 Web 应用，和丰富的 HTTP 工具。
- [request](#) request模块让http请求变的更加简单

## 4. 总结

---

本节我们学习使用Http模块建立服务端以及发起本地请求的客户端，并展示了两个超级简单的例子。想必各位看官都已经迫不及待的将Node.js用了起来，甚至觉得有点小激动。下一节，我们还是退回去学习Node的npm，package.json，以及模块机制。然后再学习其他的

## 3. Node模块与npm

- Node模块与npm
  - Node模块
  - 小示例
    - 1. 模块引用
    - 2. 模块定义
- npm模块管理器
  - npm包
  - package.json
  - npm的使用
    - npm 命令安装模块
    - 卸载模块
    - 更新模块
    - 创建模块
    - 模块发布
  - 推荐推荐

## Node模块与npm

为了让Node.js的文件可以相互调用，Node.js提供了一个基于CommonJS的模块系统。

模块是Node.js 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个Node.js 文件就是一个模块，这个文件可能是JavaScript 代码、JSON 或者编译过的C/C++ 扩展。

我们都知道JavaScript先天就缺乏一种功能：模块。浏览器环境的js模块划分只能通过 `src` 引入使用。然而，我们是幸运的，我们在这个前端高速发展的时代（当然了，这也是一种压力，一觉醒来又有新东西诞生了）。高速发展下社区总结出了CommonJS这算得上是最为重要的里程碑。CommonJS制定了解决这些问题的一些规范，而

Node.js就是这些规范的一种实现。Node.js自身实现了require方法作为其引入模块的方法，同时NPM也是基于CommonJS定义的包规范

## Node模块

每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。commonJS这套规范的出现使得用户不必再考虑变量污染，命名空间这些问题了。

## 小示例

```
1. // add.js
2. const add = (a, b)=>{
3.     return a+b
4. }
5. =====
6. // index.js
7. const add = require('./add')
8.
9. let result = add(1, 2)
10. console.log(result)
```

## 1. 模块引用

`require()` 这个方法存在接受一个模块标识，以此引入模块

```
1. const fs = require('fs')
```

Node中引入模块要经历一下三步：

1. 路径分析
2. 文件定位
3. 编译执行

Node优先从缓存中加载模块。Node的模块可分为两类：

- Node提供的核心模块
- 用户编写的文件模块

Node核心模块加载速度仅次于缓存中加载，然后路径形式的模块次之，最慢的是自定义模块。

## 2. 模块定义

在模块中，上下文提供了 `exports` 来导出模块的方法或者变量。它是唯一出口

```
1. exports.add = function(){
2.   // TODO
3. }
```

在模块中还存在一个`module`对象，它代表模块自身，`exports` 是它的属性。为了方便，Node为每个模块提供一个`exports`变量，指向`module.exports`。这等同在每个模块头部，有一行这样的命令

```
1. var exports = module.exports
```

不能直接将`exports`变量指向一个值，因为这样等于切断了 `exports` 与 `module.exports` 的联系

### `exports`和`module.exports` 区别

`exports`仅仅是`module.exports`的一个地址引用。nodejs只会导出`module.exports`的指向，如果`exports`指向变了，那就仅仅是`exports`不在指向`module.exports`，于是不会再被导出

- `module.exports`才是真正的接口，`exports`只不过是它的一个



辅助工具。最终返回给调用的是`module.exports`而不是`exports`。

- 所有的`exports`收集到的属性和方法，都赋值给了`module.exports`。当然，这有个前提，就是`module.exports`本身不具备任何属性和方法。如果，`module.exports`已经具备一些属性和方法，那么`exports`收集来的信息将被忽略
- Node开发者建议导出对象用`module.exports`，导出多个方法和变量用`exports`

## npm模块管理器

`npm` 的出现则是为了在CommonJS规范的基础上，实现解决包的安装卸载，依赖管理，版本管理等问题，`npm` 不需要单独安装。在安装Node的时候，会连带一起安装 `npm`

- 允许用户从NPM服务器下载别人编写的第三方包到本地使用。
- 允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

## npm包

一个符合CommonJS规范的包应该是如下这种结构：

- 一个`package.json`文件应该存在于包顶级目录下
- 二进制文件应该包含在`bin`目录下。
- JavaScript代码应该包含在`lib`目录下。
- 文档应该在`doc`目录下。
- 单元测试应该在`test`目录下

## package.json

- **name**: 包名，需要在NPM上是唯一的，小写字母和数字组成可包含 `_` `-` `.` 但不能有空格
- **description**: 包简介。通常会显示在一些列表中
- **version**: 版本号。一个语义化的版本号 (<http://semver.org/>)，通常为x.y.z。该版本号十分重要，常常用于一些版本控制的场合
- **keywords**: 关键字数组。用于NPM中的分类搜索
- **maintainers**: 包维护者的数组。数组元素是一个包含name、email、web三个属性的JSON对象
- **contributors**: 包贡献者的数组。第一个就是包的作者本人。在开源社区，如果提交的patch被merge进master分支的话，就应当加上这个贡献patch的人。格式包含name和email
- **bugs**: 一个可以提交bug的URL地址。可以是邮件地址 (<mailto:mailxx@domain>)，也可以是网页地址
- **licenses**: 包所使用的许可证
- **repositories**: 托管源代码的地址数组
- **dependencies**: 当前包需要的依赖。这个属性十分重要，NPM会通过这个属性，帮你自动加载依赖的包

除了前面提到的几个必选字段外，还有一些额外的字段，如bin、scripts、engines、devDependencies、author

## npm的使用

行下面的命令，查看各种信息

1. # 查看 npm 命令列表
2. \$ npm help
- 3.

```

4. # 查看各个命令的简单用法
5. $ npm -l
6.
7. # 查看 npm 的版本
8. $ npm -v
9.
10. # 查看 npm 的配置
11. $ npm config list -l

```

## npm 命令安装模块

Node模块采用 `npm install` 命令安装。

每个模块可以“全局安装”，也可以“本地安装”。“全局安装”指的是将一个模块安装到系统目录中，各个项目都可以调用。一般来说，全局安装只适用于工具模块。“本地安装”指的是将一个模块下载到当前项目的 `node_modules` 子目录，然后只有在项目目录之中，才能调用这个模块。

```

1. # 本地安装
2. $ npm install <package name>
3.
4. # 全局安装
5. $ sudo npm install -global <package name>
6. $ sudo npm install -g <package name>

```

指定所安装的模块属于哪一种性质的依赖关系

- `-save`：模块名将被添加到dependencies，可以简化为参数 `-S`。
- `-save-dev`：模块名将被添加到devDependencies，可以简化为参数 `-D`。

```

1. $ npm install <package name> --save
2. $ npm install <package name> --save-dev

```

## 卸载模块

我们可以使用以下命令来卸载 Node.js 模块

```
1. $ npm update <package name>
```

## 更新模块

我们可以使用以下命令来卸载 Node.js 模块

```
1. $ npm uninstall <package name>
```

## 创建模块

我们可以使用以下命令来创建 Node.js 模块

```
1. $ npm init
```

`npm init` 创建模块会在交互命令行帮我们生产package.json文件

```
1. $ npm init
2. This utility will walk you through creating a package.json file.
3. It only covers the most common items, and tries to guess sensible
   defaults.
4.
5. See `npm help json` for definitive documentation on these fields
6. and exactly what they do.
7.
8. Use `npm install <pkg> --save` afterwards to install a package and
9. save it as a dependency in the package.json file.
10.
11. Press ^C at any time to quit.
12. name: (node_modules) test # 模块名
13. version: (1.0.0)
14. description: Node.js 测试模块 # 描述
```

```

15. entry point: (index.js)
16. test command: make test
17. git repository: https://github.com/test/test.git # Github 地址
18. keywords:
19. author:
20. license: (ISC)
21. About to write to ...../node_modules/package.json: # 生成地址
22.
23. {
24.   "name": "test",
25.   "version": "1.0.0",
26.   "description": "Node.js 测试模块",
27.   .....
28. }
29.
30.
31. Is this ok? (yes) yes

```

以上的信息，你需要根据你自己的情况输入。默认回车即可。在最后输入“yes”后会生成 package.json 文件。

## 模块发布

发布模块前首先要在npm注册用户

```

1. $ npm adduser
2. Username: liuxing
3. Password:
4. Email: (this IS public) ogilhinn@gmail.com

```

然后

```

1. $ npm publish

```

现在我们的npm包就成功发布了。

更多请查看*npm*帮助信息[npm 文档](#)

## 推荐推荐

- [npm 文档](#)
- [深入Node.js的模块机制](#)

## 4. 搭建静态文件服务器

- [?Node入门教程-搭建静态文件服务器](#)
  - [Node.js 文件系统](#)
    - 写入文件
    - 打开文件
    - 读取目录
- [Path模块](#)
- [Node静态文件服务器](#)
  - [1. 首先起一个http服务器](#)
  - [2. 处理URL请求](#)
  - [3. 读取文件发送给浏览器](#)
- [总结:](#)

## ?Node入门教程-搭建静态文件服务器

传送门: [GitHub地址](#)

通过前面的几篇介绍,我们这个教程算正式打开Node开发的大门,学习了[环境搭建](#)、然后为了提高各位看官的新趣粗略的介绍了[Http模块](#)、之后又了解了[Node的模块](#)。之前说过,我们将通过实例来学习Node,从这一篇开始,我们就将用实例来学习各个模块。

这一节我们将学习[File System \(文件系统\)](#)以及[Path\(路径\)](#)并结合之前学习的知识打造 ? 一个Node静态文件服务器

## Node.js 文件系统

Node.js提供本地文件的读写能力,基本上类似 UNIX (POSIX) 标准的文件操作API。 所有的方法都有异步和同步的形式。例如读取文件内容的函数有异步的`fs.readFile()` 和同步的`fs.readFileSync()`。

异步的方法函数最后一个参数为回调函数,回调函数的第一个参数包含了错误信息(error),则第一个参数会是 `null` 或 `undefined`。

```

1. const fs = require('fs')
2. /*异步读取
3.  *=====*/
4. fs.readFile('README.md', function (err, data) {
5.     if (err) {
6.         return console.error(err)
7.     }
8.     console.log("异步读取: " + data.toString())
9. })
10. console.log("程序执行完毕。")

```

结果: `程序执行完毕。` 会被先打印出来

```

1. /*同步读取
2.  *=====*/
3. const fs = require('fs')
4. const data = fs.readFileSync('README.md')
5. console.log("同步读取: " + data.toString())
6.
7. console.log("程序执行完毕。")

```

结果: `程序执行完毕。` 后打印出来

强烈推荐大家是用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞

接下来我们一起来看看fs模块的常用方法

## 写入文件

```
1. fs.writeFile(file, data[, options], callback)
```

参数说明:

- `file` - 文件名或文件描述符
- `data` - 要写入文件的数据，可以是 `String`(字符串) 或



## Buffer(流) 对象

- `options` - 该参数是一个对象，包含 {encoding, mode, flag}。默认编码为 utf8，模式为 0666，flag 为 'w'，\*如果是一个字符串，则它指定了字符编码
- `callback` - 回调函数

以追加模式往README.me写入字符串Hello Node.js

```
1. fs.writeFile('README.md', 'Hello Node.js', {flag: 'a+'}, (err) => {
2.   if (err) throw err
3.   console.log('It\'s saved!')
4. })
```

这里我们介绍下 `flags`：

Flag	描述
r	以读取模式打开文件。如果文件不存在抛出异常。
r+	以读写模式打开文件。如果文件不存在抛出异常。
rs	以同步的方式读取文件。
rs+	以同步的方式读取和写入文件。
w	以写入模式打开文件，如果文件不存在则创建。
wx	类似 'w'，但是如果文件路径存在，则文件写入失败。
w+	以读写模式打开文件，如果文件不存在则创建。
wx+	类似 'w+'，但是如果文件路径存在，则文件读写失败。
a	以追加模式打开文件，如果文件不存在则创建。
ax	类似 'a'，但是如果文件路径存在，则文件追加失败。
a+	以读取追加模式打开文件，如果文件不存在则创建。
ax+	类似 'a+'，但是如果文件路径存在，则文件读取追加失败。

## 打开文件

- 同步 `fs.open(path, flags[, mode], callback)`
- 异步 `fs.openSync(path, flags[, mode])`

### 参数说明：

- `path` - 文件的路径
- `flags` - 文件打开的行为。具体值详见下文
- `mode` - 设置文件模式(权限)，文件创建默认权限为 0666(可读，可写)
- `callback` - 回调函数，带有两个参数如：callback(err, fd)

```
```javascript
```

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

```
fs.open('README.md', 'r+', function(err, fd) {
  if (err) {
    return console.error(err)
  }
  console.log("文件打开成功！")
})
```

```
1.
2. ##### 读取文件
3.
4. ```javascript
5. fs.read(fd, buffer, offset, length, position, callback)
```

### 参数说明：

- `fd` - 通过 `fs.open()` 方法返回的文件描述符
- `buffer` - 是数据将被写入到的 `buffer`
- `offset` - 是 `buffer` 中开始写入的偏移量
- `length` - 是一个整数，指定要读取的字节数
- `position` - 是一个整数，指定从文件中开始读取的位置。 如果 `position` 为 `null`，则数据从当前文件位置开始读取

- `callback` - 回调函数，有三个参数`err`，`bytesRead`，`buffer`，`err` 为错误信息，`bytesRead` 表示读取的字节数，`buffer` 为缓冲区对象

```

1. const fs = require("fs");
2. let buf = new Buffer(1024)
3. fs.open('README.md', 'r+', function(err, fd) {
4.     if (err) {
5.         return console.error(err)
6.     }
7.     fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
8.         if (err){
9.             console.error(err);
10.        }
11.        console.log(bytes + " 字节被读取")
12.        // 仅输出读取的字节
13.        if(bytes > 0){
14.            console.log(buf.slice(0, bytes).toString())
15.        }
16.    })
17. })

```

现在我们就可以从README.md中读取1kb的数据

## 读取目录

`readdir` 方法用于读取目录，返回一个所包含的文件和子目录的数组。

```
1. fs.readdir(path[, options], callback)
```

同步版本：

```
1. fs.readdirSync(path[, options])
```

我们来写个遍历目录的方法吧！这是个同步的(同步版本简单点)

遍历目录时一般使用递归算法，否则就难以编写出简洁的代码。递归算法与数学归纳法类似，通过不断缩小问题的规模来解决问题，目录是一个树状结构，在遍历时一般使用深度优先+先序遍历算法

```

1. function travel(dir, callback) {
2.     fs.readdirSync(dir).forEach(function (file) {
3.         var pathname = path.join(dir, file)
4.
5.         if (fs.statSync(pathname).isDirectory()) {
6.             travel(pathname, callback)
7.         } else {
8.             callback(pathname)
9.         }
10.    })
11. }
```

该函数以某个目录作为遍历的起点。遇到一个子目录时，就先接着遍历子目录。遇到一个文件时，就把文件的绝对路径传给回调函数。回调函数拿到文件路径后，就可以做各种判断和处理了：

```

1. travel(__dirname, function (pathname) {
2.     console.log(pathname)
3. })
```

接下来，大家可以试着去实现异步遍历，原理都是一样的

关于File System（文件系统）的更多API请自行查看[Node中文网](#)，

## Path模块

`path` 模块提供了一些工具函数，用于处理文件与目录的路径，`path` 模块的默认操作会根据 `Node.js` 应用程序运行的操作系统的不同而变化。比如，当运行在 `Windows` 操作系统上时，`path` 模块会认为使用的是 `Windows` 风格的路径

### 常用方法介绍

`path.join([...paths])` 方法用于连接路径

```
1. path.join('foo', "bar");
2. // 返回: '/foo/bar'
3.
4. path.join('foo', {}, 'bar')
5. // 抛出 TypeError: path.join 的参数必须为字符串
```

`path.resolve()` 方法会把一个路径或路径片段的序列解析为一个绝对路径方法用于将相对路径转为绝对路径

```
1. path.resolve('/foo/bar', './baz')
2. // 返回: '/foo/bar/baz'
3.
4. path.resolve('/foo/bar', '/tmp/file/')
5. // 返回: '/tmp/file'
6.
7. path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
8. // 如果当前工作目录为 /home/myself/node,
9. // 则返回 '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

`path.extname()` 方法返回 `path` 的扩展名，即从 `path` 的最后部分中的最后一个 `.`（句号）字符到字符串结束

```
1. path.extname('index.html')
2. // 返回: '.html'
3.
4. path.extname('index.coffee.md')
5. // 返回: '.md'
```

关于Path（路径）的更多API请自行查看[Node中文网](#)

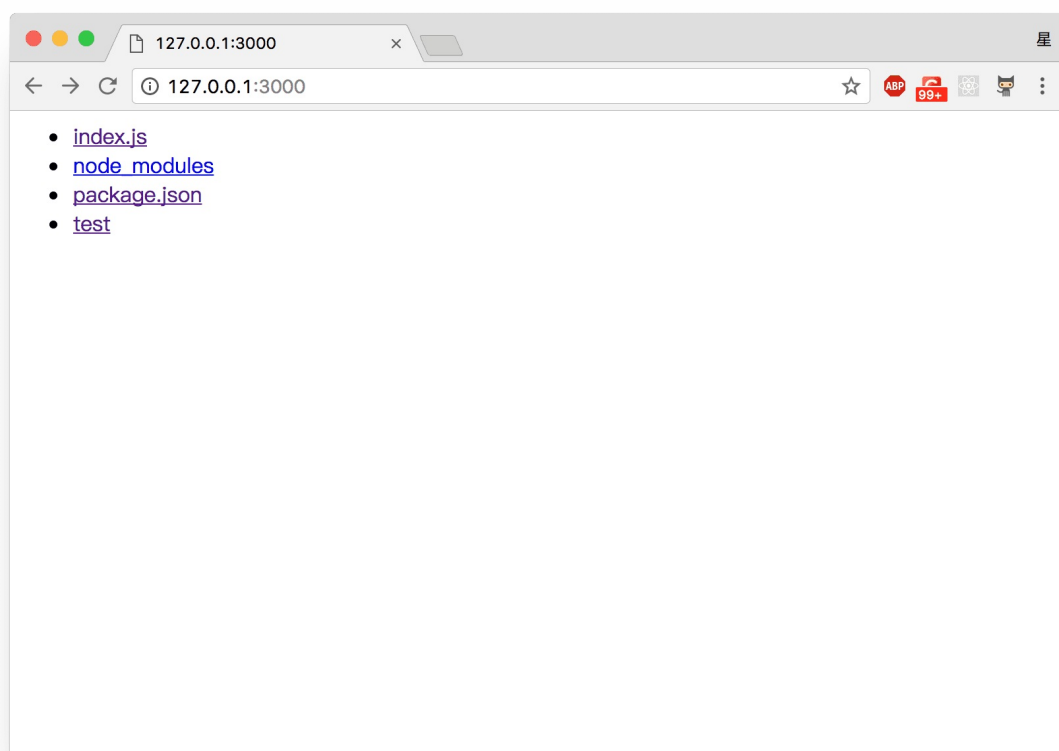
上面介绍了fs与path模块的几个常用API，在使用时我们应该经常查看API文档，上面所学习的方法已经足够打造一静态文件服务器了，下面我们就一起开完成这个小案例吧。

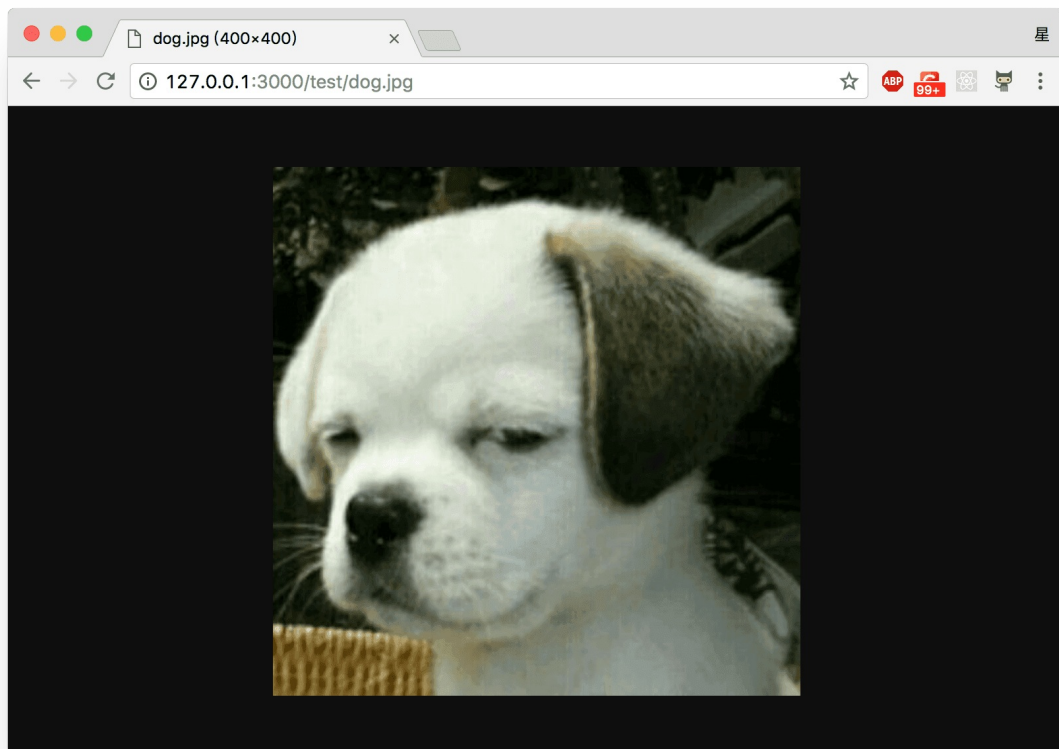
## Node静态文件服务器

这个静态文件服务器大致是这样的：浏览器发送URL，服务端解析URL，对应到硬盘上的文件。如果文件存在，返回200状态码，并发送文件到浏览器端，我们要实现的功能如下：

- 主页显示当前目录下的文件和文件夹
- 点击链接可以打开文件或者文件夹

有图有真相，话不多说看图：





现在我们来一步一步实现这个小项目

## 1. 首先起一个http服务器

我们先来初始化工作：

1. 新建文件夹 `file-server`
2. `npm init` 初识初始化生成 `package.json`
3. 新建 `index.js` 文件

现在我们在 `index.js` 文件中起一个服务器：

```
1. const http = require('http')
2.
3. const hostname = '127.0.0.1'
4. const port = 3000
5.
6. const server = http.createServer((req, res) => {
7.   res.statusCode = 200
8.   res.setHeader('Content-Type', 'text/plain')
9.   res.end('Hello World\n')
10. })
11.
12. server.listen(port, hostname, () => {
13.   console.log(`服务器运行在 http://${hostname}:${port}`)
14. })
```

还记得我们第一节让大家自己如了解的 `supervisor` 工具吗？它可以实现监测文件修改并自动重启应用

```
1. $ supervisor --harmony index.js
2. Running node-supervisor with
3.   program '--harmony index.js'
4.   --watch '.'
5.   --extensions 'node,js'
6.   --exec 'node'
```



- 7.
8. Starting child process with 'node --harmony index.js'
9. Watching directory '/Users/lx/Documents/workspace/node-abc/lesson4/file-server' for changes.
10. Press rs for restarting the process.
11. 服务器运行在 http://127.0.0.1:3000

现在我们这个服务器跑了起来，而且每次更改文件后不需要手动重启服务。

## 2. 处理URL请求

现在我们要url模块与path来识别请求的文件(还记得第二节了解的url模块吗?)

```

1. const http = require('http')
2. const url = require('url') //引入url模块
3.
4. const hostname = '127.0.0.1'
5. const port = 3000
6.
7. const server = http.createServer((req, res) => {
8.   if(req.url == '/favicon.ico') return //不响应favicon请求
9.   // 获取url->pathname 即文件名
10.  let pathname = path.join(__dirname, url.parse(req.url).pathname)
11.  pathname = decodeURIComponent(pathname) // url解码, 防止中文路径出错
12.  console.log(pathname) // .../node-abc/lesson4/file-server/ 请求的
    pathname
13. })
14.
15. server.listen(port, hostname, () => {
16.  console.log(`服务器运行在 http://${hostname}:${port}`)
17. })

```

## 3. 读取文件发送给浏览器

接下来我们就运用本节所讲的文件系统的知识来处理文件，

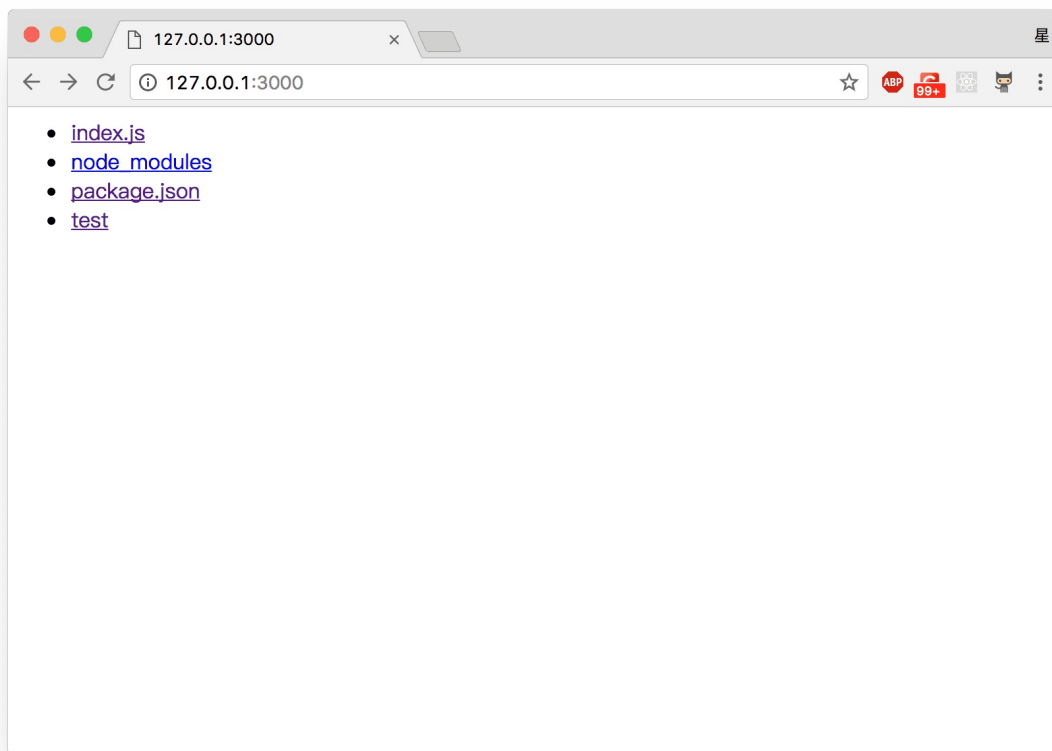
## 1. 先来处理文件夹：

```

1. ....
2.   if(req.url == '/favicon.ico') return //不响应favicon请求
3.   // 获取url->pathname 即文件名
4.   let pathname = path.join(__dirname, url.parse(req.url).pathname)
5.   pathname = decodeURIComponent(pathname) // url解码，防止中文路径出错
6.   console.log(pathname) // .../node-abc/lesson4/file-server/ 请求的
   pathname
7.
8.   /**
9.    * 判断文件是否是文件夹
10.   * 是：返回文件列表
11.   * 否：读取文件内容
12.   */
13.   // stat方法的参数是一个文件或目录，它产生一个对象，该对象包含了该文件或目录
   的具体信息。我们往往通过该方法，判断正在处理的到底是一个文件，还是一个目录，这儿
   使用的是它的同步版本
14.   if(fs.statSync(pathname).isDirectory()){
15.       // 设置响应头
16.       res.writeHead(200, {'Content-Type': 'text/html';
   charset=utf-8'})
17.       fs.readdir(pathname, (err, files)=>{
18.           res.write('<ul>')
19.           files.forEach((item)=>{
20.               // 处理路径
21.               let link = path.join(url.parse(req.url).pathname,
   item)
22.               res.write(`<li><a href="${link}">${item}</a></li>`)
23.           })
24.           res.end('</ul>')
25.       })
26.   }
27.   ...

```

我们先用 `fs.statSync(pathname).isDirectory()` 来判断是否为文件夹，是则给浏览器返回当前文件夹下的文件列表，请求 `/` 直返回：



此处我们新建了个 `test` 文件夹放了一些文件作为测试文件

## 2. 文件处理

因为我们的服务器同时要存放html, css, js, png, gif, jpg等等文件。并非每一种文件的MIME类型都是text/html的，所以这里我们引入了mime模块，来处理mime支持

```
1. const mime = require('mime');
2.
3. mime.lookup('/path/to/file.txt');           // => 'text/plain'
4. mime.lookup('file.txt');                     // => 'text/plain'
5. mime.lookup('.TXT');                         // => 'text/plain'
6. mime.lookup('htm');                         // => 'text/html'
7. //具体使用异步官网
```

这儿用到了前面讲的文件读取：

```
1. else{
2.   // 以binary读取文件
3.   fs.readFile(pathname, 'binary', (err, data)=>{
4.     if(err){
5.       res.writeHead(500, { 'Content-Type': 'text/plain'})
6.       res.end(JSON.stringify(err))
7.       return false
8.     }
9.     res.writeHead(200, {
10.      'Content-Type': `${mime.lookup(pathname)}; charset=UTF-8`
11.    })
12.    res.write(data, 'binary')
13.    res.end()
14.  })
15. }
```

如果路径不是文件夹，就读取具体的文件，这儿我们以二进制(binary)编码读取，你也可以试试UTF-8比较他们的区别。

到此我们这个Node静态文件服务器就算搭建完成???, 当然了还有许多优化的地方如：缓存、Gzip、页面美化等等...在此就不做介绍了，请自行搜索了解。

源码地址[node-abc](#)

## 总结：

这一节我们了解了fs模块，path模块。并运用这些知识搭建一简单的文件服务器

一切学问最重要的是融会贯通, 要把它转变成自身的学问。



## 5. Node中的stream (流)

- Node入门教程-Node中的stream (流)
  - 1.了解Node Stream(流)
    - Node.js 中有四种基本的流类型：
    - 所有的 Stream 对象都是 EventEmitter 的实例
    - 可读流 (Readable streams)
      - 可读流常用事件：
      - 可读流还提供了一些方法，我们可以用它们读取或操作流：
    - 可写流 (Writable streams)
      - 可写流常用事件
      - 可写流常用方法
    - 管道流
    - 链式流
    - 在fs中使用Stream
  - 2.通过Stream实现文件复制功能
    - 1. 简单实现文件的复制
    - 2. 添加显示处理状态的功能
  - 相关连接

## Node入门教程-Node中的stream (流)

传送门: [GitHub地址](#)

流 (*stream*) 在 *Node.js* 中是处理流数据的抽象接口 (*abstract interface*)。

`stream` 模块提供了基础的 *API*。使用这些 *API* 可以很容易地来构建实现流接口的对象。

*Node.js* 提供了多种流对象。例如，*HTTP* 请求 和 `process.stdout` 就都是流的实例。流可以是可读的、可写的，或是可读写的。所有的流都是 `EventEmitter` (*Node*

事件机制将在后续讲解，可以先自行了解)的实例。

尽管所有的 `Node.js` 用户都应该理解流的工作方式，这点很重要，但是 `stream` 模块本身只对于那些需要创建新的流的实例的开发者最有用处。对于主要是消费流的开发者来说，他们很少（如果有的话）需要直接使用 `stream` 模块。

## 1. 了解Node Stream(流)

数据流 (`stream`) 是处理系统缓存的一种方式。操作系统采用数据块 (`chunk`) 的方式读取数据，每收到一次数据，就存入缓存。`Node`应用程序有两种缓存的处理方式，第一种是等到所有数据接收完毕，一次性从缓存读取，这就是传统的读取文件的方式(遇上大文件很容易使内存爆仓)；第二种是采用“数据流”的方式，收到一块数据，就读取一块，即在数据还没有接收完成时，就开始处理它(像流水一样)

`Node.js` 中有四种基本的流类型：

- `Readable` - 可读的流（例如 `fs.createReadStream()`）。
- `Writable` - 可写的流（例如 `fs.createWriteStream()`）。
- `Duplex` - 可读写的流（例如 `net.Socket`）。
- `Transform` - 在读写过程中可以修改和变换数据的 `Duplex` 流（例如 `zlib.createDeflate()`）。

所有的 `Stream` 对象都是 `EventEmitter` 的实例

常用的事件有：

- `data` - 当有数据可读时触发。
- `end` - 没有更多的数据可读时触发。
- `error` - 在接收和写入过程中发生错误时触发。
- `finish` - 所有数据已被写入到底层系统时触发。

## 可读流 (Readable streams)

可读流 (*Readable streams*) 是对提供数据的 源头 (*source*) 的抽象

可读数据流有两种状态：流动状态和暂停状态。处于流动状态时，数据会尽快地从数据源导向用户的程序(就像流水一样)；处于暂停态时，必须显式调用 `stream.read()` 等指令，“可读数据流”才会释放数据，(就像流水的闸门，打开它水才继续流下去)

可读流在创建时都是暂停模式，暂停模式和流动模式可以互相转换。

要从暂停模式切换到流动模式，有下面三种办法：

- 给“data”事件关联了一个处理器
- 显式调用 `resume()`
- 调用 `pipe()` 方法将数据送往一个可写数据流

要从流动模式切换到暂停模式，有两种途径：

- 如果这个可读的流没有桥接可写流组成管道，直接调用 `pause()`
- 如果这个可读的流与若干可写流组成了管道，需要移除与“data”事件关联的所有处理器，并且调用 `unpipe()` 方法断开所有管道

可读流常用事件：

- `readable`：在数据块可以从流中读取的时候发出。它对应的处理器没有参数，可以在处理器里调用 `read([size])` 方法读取数据。
- `data`：有数据可读时发出。它对应的处理器有一个参数，代表数据。如果你只想快速地读取一个流的数据，给data关联一个处理器是最方便的办法。处理器的参数是Buffer对象，如果你调用了Readable的 `setEncoding(encoding)` 方法，处理器的参数就是String对象。
- `end`：当数据被读完时发出。对应的处理器没有参数。



- `close` : 当底层的资源, 如文件, 已关闭时发出。不是所有的 `Readable` 流都会发出这个事件。对应的处理器没有参数。
- `error` : 当在接收数据中出现错误时发出。对应的处理器参数是 `Error` 的实例, 它的 `message` 属性描述了错误原因, `stack` 属性保存了发生错误时的堆栈信息。

可读流还提供了一些方法, 我们可以用它们读取或操作流:

- `read([size])` : 该方法可以接受一个整数作为参数, 表示所要读取数据的数量, 然后会返回该数量的数据。如果读不到足够数量的数据, 返回 `null`。如果不提供这个参数, 默认返回系统缓存之中的所有数据。
- `setEncoding(encoding)` : 给流设置一个编码格式, 用于解码读到的数据。调用此方法后, `read([size])` 方法返回 `String` 对象。
- `pause()` : 暂停可读流, 不再发出 `data` 事件
- `resume()` : 恢复可读流, 继续发出 `data` 事件
- `pipe(destination, [options])` : 绑定一个 `Writable` 到 `readable` 上, 将可写流自动切换到 `flowing` 模式并将所有数据传给绑定的 `Writable`。数据流将被自动管理。这样, 即使是可读流较快, 目标可写流也不会超负荷 (`overwhelmed`)
- `unpipe([destination])` : 该方法移除 `pipe` 方法指定的数据流目的地。如果没有参数, 则移除所有的 `pipe` 方法目的地。如果有参数, 则移除该参数指定的目的地。如果没有匹配参数的目的地, 则不会产生任何效果

## 可写流 (Writable streams)

*Writable streams 是 destination 的一种抽象, 这种 destination 允许数据写入*

`write` 方法用于向“可写数据流”写入数据。它接受两个参数, 一个是写入的内容, 可以是字符串, 也可以是一个 `stream` 对象 (比如可读数

据流) 或 `buffer` 对象 (表示二进制数据), 另一个是写入完成后的回调函数, 它是可选的, `write` 方法返回一个布尔值, 表示本次数据是否处理完成

## 可写流常用事件

- `drain` `writable.write(chunk)` 返回 `false` 以后, 当缓存数据全部写入完成, 可以继续写入时, 会触发 `drain` 事件
- `finish` 调用 `end` 方法时, 所有缓存的数据释放, 触发 `finish` 事件。该事件的回调函数没有参数
- `pipe` 可写数据流调用 `pipe` 方法, 将数据流导向写入目的地时, 触发该事件
- `unpipe` 可读数据流调用 `unpipe` 方法, 将可写数据流移出写入目的地时, 触发该事件
- `error` 如果写入数据或 `pipe` 数据时发生错误, 就会触发该事件

## 可写流常用方法

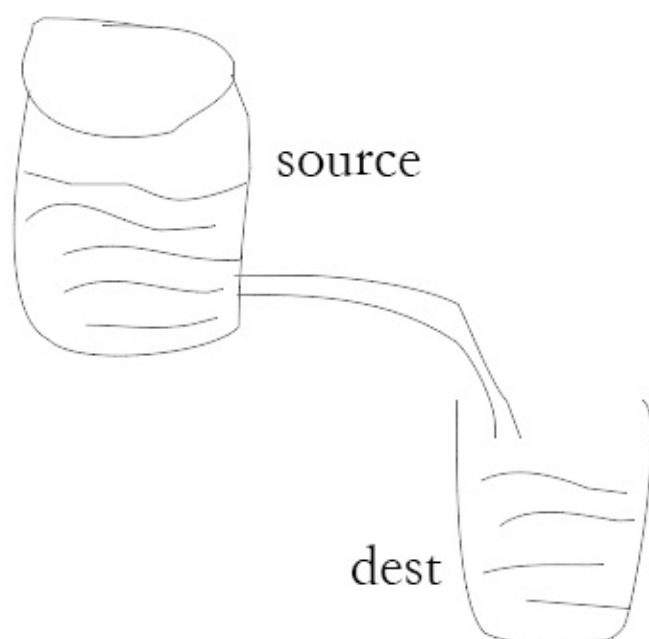
- `write()` 用于向“可写数据流”写入数据。它接受两个参数, 一个是写入的内容, 可以是字符串, 也可以是一个 `stream` 对象 (比如可读数据流) 或 `buffer` 对象 (表示二进制数据), 另一个是写入完成后的回调函数, 它是可选的。
- `cork()`, `uncork()` `cork` 方法可以强制等待写入的数据进入缓存。当调用 `uncork` 方法或 `end` 方法时, 缓存的数据就会吐出。
- `setDefaultEncoding()` 用于将写入的数据编码成新的格式。它返回一个布尔值, 表示编码是否成功, 如果返回 `false` 就表示编码失败。
- `end()` 用于终止“可写数据流”。该方法可以接受三个参数, 全部

都是可选参数。第一个参数是最后所要写入的数据，可以是字符串，也可以是 `stream` 对象或 `buffer` 对象；第二个参数是写入编码；第三个参数是一个回调函数，`finish` 事件发生时，会触发这个回调函数。

## 管道流

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中>(我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(*pipe*)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程)

千言万语抵不过这图：



## 链式流

链式是通过连接输出流到另外一个流并创建多个对个流操作链的机制。

链式流一般用于管道操作。

接下来我们就是用管道和链式来压缩文件  
创建 `compress.js` 文件，代码如下：

```
1. const fs = require("fs");
2. const zlib = require('zlib')
3.
4. // 压缩 README.md 文件为 README.md.gz
5. fs.createReadStream('./README.md')
6.   .pipe(zlib.createGzip())
7.   .pipe(fs.createWriteStream('README.md.gz'))
8.
9. console.log("文件压缩完成")
```

## 在fs中使用Stream

在介绍完stream后我们用流来实现文件读取

```
1. const fs = require("fs")
2.
3. //可读数据流
4. //=====
5. let data = ''
6. //创建可读流
7. let readerStream = fs.createReadStream('./README.md')
8.
9. // 设置编码为 utf8
10. readerStream.setEncoding('UTF8')
11.
12. // 处理流事件 --> data, end, and error
13. readerStream.on('data', function(chunk) {
14.     data += chunk
15. })
16.
17. readerStream.on('end', function(){
18.     console.log(data)
```

```

19. })
20.
21. readerStream.on('error', function(err){
22.     console.log(err.stack)
23. })
24.
25. console.log("程序执行完毕")

```

大致介绍了Node的Stream(流)很少需要直接使用 `stream` 模块，更多信息参考[Node中文网](#)

下面我们就进入今天的实例小项目：实现文件的复制功能

## 2. 通过Stream实现文件复制功能

本例参考 [chshouyu: nodejs中流\(stream\)的理解](#)

Node的 `fs` 模块并没有提供一个 `copy` 的方法，但我们可以很容易的实现一个。

我们先来看看在不知道流之前我们会怎么做：

```

1. const fs = require('fs')
2. const file = fs.readFileSync('./README.md', {encoding: 'utf8'})
3. fs.writeFileSync('./TEST.md', file)

```

这种方式是把文件内容全部读入内存，然后再写入文件，对于小型的文本文件，这没有多大问题。但对大文件来说要花很长时间，才能进入数据处理的步骤。甚至引起内存爆仓

既然我们学了Stream，现在我们就用Stream来实现这个简单的功能：

### 1. 简单实现文件的复制

```

1. const fs = require('fs')

```

```

2.
3.  let pathname = {
4.      src: './copy.js',
5.      dist: './test.js'
6.  }
7.
8.  let ReadStream = fs.createReadStream(pathname.src)
9.  let WriteStream = fs.createWriteStream(pathname.dist)
10.
11.  ReadStream.pipe(WriteStream)
12.
13.  // 复制完成触发的事件
14.  WriteStream.on('finish', ()=>{
15.      console.log('复制完成')
16.  })

```

我们可以通过管道流很方便的实现单个文件的复制。

## 2. 添加显示处理状态的功能

这里很简单就直接贴代码了。

```

1.  const fs = require('fs')
2.  const out = process.stdout
3.
4.  let paths = {
5.      src: '../test/test.mp4',
6.      dist: '../test1.mp4'
7.  }
8.
9.  function copy(paths){
10.      let {src, dist} = paths
11.      let readStream = fs.createReadStream(src)
12.      let writeStream = fs.createWriteStream(dist)
13.
14.      let stat = fs.statSync(src),
15.          totalSize = stat.size,
16.          progress = 0,

```

```

17.         lastSize = 0,
18.         startTime = Date.now()
19.
20.         readStream.on('data', function(chunk) {
21.             progress += chunk.length;
22.         })
23.
24.         // 我们添加了一个递归的setTimeout来做一个旁观者
25.         // 每500ms观察一次完成进度，并把已完成的大小、百分比和复制速度一并写到控制
台上
26.         // 当复制完成时，计算总的耗费时间
27.         setTimeout(function show() {
28.             let percent = Math.ceil((progress / totalSize) * 100)
29.             let size = Math.ceil(progress / 1000000)
30.             let diff = size - lastSize
31.             lastSize = size
32.             out.clearLine()
33.             out.cursorTo(0)
34.             out.write(`已完成${size}MB, ${percent}%, 速度: ${diff *
2}MB/s`)
35.             if (progress < totalSize) {
36.                 setTimeout(show, 500)
37.             } else {
38.                 let endTime = Date.now()
39.                 console.log(`共用时: ${((endTime - startTime) / 1000)}秒。
`)
40.             }
41.         }, 500)
42.     }
43.
44.     copy(paths)

```

到此我们就用流来处理了文件复制。当然我们还可以用它来处理HTTP requests, on the client、HTTP responses, on the server、fs write streams、zlib streams、crypto streams、TCP sockets、child process stdin、

`process.stdout` , 、 `process.stderr` , 大家可以自己试试。

对所有流来说, 通常使用 `pipe` 方法更为简便直接

本文介绍了Stream的一些基础, 并用它实现了一个小的文件复制功能。这是只是入门小文档, 关于Node的Stream更多的知识, 需要大家自己去了解

抛砖引玉

## 相关链接

- [GitHub地址](#)
- [Node中文网](#)
- [阮一峰](#)
- [nodejs中流\(stream\)的理解](#)



## 6. Node的readline (逐行读取)

- Node的readline (逐行读取)
  - readline基本用法
    - 创建Readline实例
    - 方法
    - 事件
    - readline 实例
      - 1.使用readline实现一个可交互的命令行
      - 2.使用http模块发起请求
      - 3. 实现命令行搜索
  - 相关链接

## Node的readline (逐行读取)

传送门: [github地址](#)

`readline` 模块提供了一个接口，用于从可读流（如 `process.stdin`）读取数据，每次读取一行

先来看这个基本示例：

```

1. const readline = require('readline');
2.
3. const rl = readline.createInterface({
4.   input: process.stdin,
5.   output: process.stdout
6. });
7.
8. rl.question('你叫什么名字?', (answer) => {
9.   // 对答案进行处理
10.  console.log(`你好: ${answer}`);
11.
12.  rl.close();

```

```
13. });
```

在命令行中执行 `node hello.js` 将出现如下结果：

```
1. $ node hello.js
2. 你叫什么名字? liu
3. 早上好, liu
```

通过这个示例，我们可以发现 `readline` 模块可以方便的实现命令行的交互功能。实现如[Node命令行工具开发【看段子】](#)治疗的小应用

下面我们先来了解 `readline` 的基本用法，然后再实现一个有趣的小功能。

## readline基本用法

`readline.Interface` 类的实例是使用 `readline.createInterface()` 方法构造的。每个实例都关联一个 `input` 可读流和一个 `output` 可写流。`output` 流用于为到达的用户输入打印提示，且从 `input` 流读取

## 创建Readline实例

```
1. readline.createInterface(options)
```

创建一个 `readline` 的接口实例。接受一个Object类型参数，可传递以下几个值：

- `input` - 要监听的可读流（必需）
- `output` - 要写入 `readline` 的可写流（必须）。
- `completer` - 用于 Tab 自动补全的可选函数。（不常用）
- `terminal` - 如果希望 `input` 和 `output` 流像 TTY 一样对

待，那么传递参数 `true`，并且经由 ANSI/VT100 转码。默认情况下检查 `isTTY` 是否在 `output` 流上实例化。（不常用）

## 方法

- `r1.close()` 关闭接口实例 (Interface instance)，放弃控制输入输出流。“close”事件会被触发
- `r1.pause()` 暂停 readline 的输入流 (input stream)，如果有需要稍后还可以恢复。
- `r1.prompt([preserveCursor])` 为用户输入准备好readline，将现有的setPrompt选项放到新的一行，让用户有一个新的地方开始输入。将preserveCursor设为true来防止光标位置被重新设定成0。
- `r1.question(query, callback)` 预先提示指定的query，然后用户应答后触发指定的callback。显示指定的query给用户后，当用户的应答被输入后，就触发了指定的callback
- `r1.resume()` 恢复 readline 的输入流 (input stream)。
- `r1.setPrompt(prompt)` 用于设置每当 `r1.prompt()` 被调用时要被写入到 `output` 的提示。
- `r1.write(data[, key])` 把 `data` 或一个由 `key` 指定的按键序列写入到 `output`

## 事件

- `line` 事件：在 input 流接受了一个 `\n` 时触发，通常在用户敲击回车或者返回时接收。这是一个监听用户输入的利器。
- `pause` 事件：输入流被暂停就会触发。同样当输入流未被暂停，但收到 SIGCONT 也会触发。（详见 SIGTSTP 和 SIGCONT 事件）

- `resume` 事件：只要输入流重新启用就会触发
- `close` 事件：当 `close()` 被调用时触发。当 `input` 流接收到 `end` 事件时也会被触发。流接收到表示结束传输的 `<ctrl>-D`，收到表示 `SIGINT` 的 `<ctrl>-C`，且 `readline.Interface` 实例上没有注册 `SIGINT` 事件监听器。

更多请参考[node中文网](#)

## readline 实例

现在我们来实现一个命令行可交互的百度搜索

### 1. 使用readline实现一个可交互的命令行

```

1. // 先来实现一个可交互命令行
2. const rl = readline.createInterface({
3.   input: process.stdin,
4.   output: process.stdout,
5.   prompt: 'search>>> '
6. })
7.
8. rl.prompt()
9.
10. rl.on('line', (line) => {
11.   console.log(line)
12.   rl.prompt()
13. }).on('close', () => {
14.   console.log('再见!')
15.   process.exit(0)
16. })

```

`node index.js` 运行这段代码出现可交互的命令行，等待我们输入，输入完成回车后会打印出输入的值

### 2. 使用http模块发起请求

这里为了加深对原生`http`模块的理解我们没用第三方模块，有很多好用的第三方模块如：[`request`](#)、[`superagent`](#)

```

1.  ...
2.  function search(words, callback) { // es6默认参数
3.      let options = {
4.          hostname: 'www.baidu.com',
5.          port: 80,
6.          path: `/s?wd=${encodeURIComponent(words)}`,
7.          method: 'GET'
8.      }
9.
10.     const req = http.request(options, (res) => {
11.         // console.log(`STATUS: ${res.statusCode}`) //返回状态码
12.         // console.log(`HEADERS: ${JSON.stringify(res.headers,
13.         null, 4)}`) // 返回头部
14.         res.setEncoding('utf8') // 设置编码
15.         let body = ''
16.         res.on('data', (chunk) => { //监听 'data' 事件
17.             body+=chunk
18.         })
19.         res.on('end', ()=>{
20.             let $ = cheerio.load(body)
21.             $('.t a').each(function(i, el){
22.                 console.log($(this).text(),
23.                 $(this).attr('href'), '\n')
24.             })
25.             callback()
26.         })
27.     })
28.     req.end() // end方法结束请求
29. }

```

这里我们用`http`模块发起客户端请求，并打印出来搜索结果的标题与链接，如果你忘记了`http`模块的使用可以回头查看[了解并使用Http模](#)

块，http模块是node相当重要的模块，后续我们还将继续学习。

这里我们用到了cheerio具体使用可参考其官网

### 3. 实现命令行搜索

到此，我们只需将命令行的代码与http请求的代码整合起来就可以完成这个小项目了

```

1. ...
2. rl.on('line', (line) => {
3.     search(line.trim(), ()=>{
4.         rl.prompt()
5.     })
6. }).on('close', () => {
7.     console.log('再见!')
8.     process.exit(0)
9. })
10. ...

```

源码地址：[github](#)

现在在快命令行中试试我们一起完成的这个小案例吧！！！！

现在我们已经基本完成了命令行百度搜索这个小案例，我们还可为他加入更多的功能。如，我们可以把它开发为全局命令行工具，只需要一个命令便可调用。如果有兴趣可以参考[Node命令行工具开发【看段子】](#)、[node命令行小工具开发【翻译小工具】](#)，建议大家都去试一试，实现更多有趣的功能

## 相关链接

- [Node中文网](#)
- [Node命令行工具开发【看段子】](#)

- node命令行小工具开发【翻译小工具】

## 6. node命令行工具开发

- node命令行工具开发
  - 一.初探
    - 一个最简单的命令行工具
    - 处理参数
    - Commander.js
    - Commander API
  - 二.开发命令行翻译工具
    - 1.新建并初始化项目
    - 2.coding
  - 三.小结
    - 相关链接

## node命令行工具开发

NodeJs有许多命令行工具。它们全局安装，并提供一个命令供我们使用，完成相应的功能。  
现在我们就用node来开发一个实用的命令行小工具

### 一.初探

#### 一个最简单的命令行工具

- 1.首先我们新建一目录，然后执行 `npm init` 生成package.json文件
- 2.新建一bin目录并在目录下创建一个hi.js

```
1. #! /usr/bin/env node
2. console.log("hi")
```

执行 `node hi.js` 我们可以看到终端输出‘hi’。。当然这并不是我们要



的命令行工具，我们需要直接运行 `hi` 就可出现结果

3. 现在我们告诉npm可执行文件是哪个，在`package.json`里添加如下信息：

```
1.  "bin": {
2.    "hi": "bin/hi.js"
3.  }
```

1. `npm link`

现在我们执行 `npm link` 启用命令行，现在再试试在终端直接输入 `hi` 命令，这次我们可以如愿见到结果

## 处理参数

命令行参数可通过系统变量 `process.argv` 获取。 `process.argv` 返回一个数组 第一个是`node` 第二个是脚本文件 第三个是输入的参数， `process.argv[2]` 开始得到才是真正的参数部分

```
1.  #! /usr/bin/env node
2.
3.  let argv = process.argv.slice(2)
4.  let yourName = argv[0]
5.  console.log(`hi, ${yourName}!`)
6.
7.  // 执行 hi liu
8.  // hi, liu!
```

## Commander.js

对于参数处理，我们一般使用`commander`，`commander`是一个轻巧的nodejs模块，提供了用户命令行输入和参数解析强大功能如：自记录代码、自动生成帮助、合并短参数（“ABC”==“-A-B-C”）、默认选

## 项、强制选项、命令解析、提示符

```
1. $ npm install commander --save
```

```
1.  #!/usr/bin/env node
2.
3.  /**
4.    * Module dependencies.
5.    */
6.
7.  var program = require('commander')
8.
9.  program
10.   .version('0.0.1')
11.   .option('-p, --peppers', 'Add peppers')
12.   .option('-P, --pineapple', 'Add pineapple')
13.   .option('-b, --bbq-sauce', 'Add bbq sauce')
14.   .option('-c, --cheese [type]', 'Add the specified type of cheese
    [marble]', 'marble')
15.   .parse(process.argv)
16.
17.  console.log('you ordered a pizza with:')
18.  if (program.peppers) console.log(' - peppers')
19.  if (program.pineapple) console.log(' - pineapple')
20.  if (program.bbqSauce) console.log(' - bbq')
21.  console.log(' - %s cheese', program.cheese)
```

## Commander API

- `Option()`：初始化自定义参数对象，设置“关键字”和“描述”
- `Command()`：初始化命令行参数对象，直接获得命令行输入
- `Command#command()`：定义一个命令名字
- `Command#action()`：注册一个callback函数
- `Command#option()`：定义参数，需要设置“关键字”和“描述”，关键字包括“简写”和“全写”两部分，以“,”、“|”、“空格”做分隔。

- `Command#parse()` : 解析命令行参数argv
- `Command#description()` : 设置description值
- `Command#usage()` : 设置usage值
- 更多参考 [commander官网](#)

除了commander外, [yargs](#)也是一个优秀的命令行参数处理模块

## 二. 开发命令行翻译工具

### 1. 新建并初始化项目

新建 文件夹translator/进入目录下执行 `npm init` 生成 `package.json`文件

```
1. npm install commander superagent cli-table2 --save
```

- [cli-table2](#)命令行表格输出
- [superagent](#)用于http请求

新建bin/translator.js文件, 并加入package.json文件中

```
1. "bin": {  
2.   "translator": "bin/translator.js"  
3. },
```

然后

```
1. npm link
```

这里我们会用到[有道API](#)

一切准备就绪我们就可以进行编码了

## 2.coding

由于代码量很小，这里就直接贴代码，在代码中以注释讲解

```

1.  #!/usr/bin/env node
2.  // 引入需要的模块
3.  const program = require('commander')
4.  const Table = require('cli-table2') // 表格输出
5.  const superagent = require('superagent') // http请求
6.  // 初始化commander
7.  program
8.    .allowUnknownOption()
9.    .version('0.0.1')
10.   .usage('translator <cmd> [input]')
11.
12.  // 有道api
13.  const API = 'http://fanyi.youdao.com/openapi.do?
    keyfrom=toaijf&key=868480929&type=data&doctype=json&version=1.1'
14.
15.  // 添加自定义命令
16.  program
17.    .command('query')
18.    .description('翻译输入')
19.    .action(function(word) {
20.      // 发起请求
21.      superagent.get(API)
22.      .query({ q: word})
23.      .end(function (err, res) {
24.        if(err){
25.          console.log('excuse me, try again')
26.          return false
27.        }
28.        let data = JSON.parse(res.text)
29.        let result = {}
30.
31.      // 返回的数据处理
32.      if(data.basic){
33.        result[word] = data['basic']['explains']

```

```

34.         }else if(data.translation){
35.             result[word] = data['translation']
36.         }else {
37.             console.error('error')
38.         }
39.
40.         // 输出表格
41.         let table = new Table()
42.         table.push(result)
43.         console.log(table.toString())
44.     })
45. })
46.
47. // 没有参数时显示帮助信息
48. if (!process.argv[2]) {
49.     program.help();
50.     console.log();
51. }
52.
53. program.parse(process.argv)

```

现在在终端中愉快的使用 `translator` 了

```

1. $ translator
2. Usage:  translator <cmd> [input]
3.   Commands:
4.     query    翻译输入
5.   Options:
6.     -h, --help      output usage information
7.     -V, --version   output the version number

```

## 三. 小结

1. 了解nodeJs 可执行脚步
2. 了解命令行参数解析
3. 了解commander, cli-table2, superagent等第三方模块

抛砖引玉，更多请参考各个模块的官方示例及API文档

## 相关链接

- [CommandJs](#)
- [cli-table2](#)
- [superagent](#)
- [Node.js 命令行程序开发教程 - 阮一峰](#)
- [Commander写自己的Nodejs命令 - 粉丝日志](#)

## 7. Node中的网络编程

- Node中的网络编程
  - 网络编程基本概念
  - Node 中的网络编程
    - 简易聊天室
      - 1. 聊天室的服务端
      - 2. 聊天室的客户端
  - 相关阅读

## Node中的网络编程

*Node*的初衷就是建立一个高性能的Web服务器，这儿我们将着重于*Node*的网络编程。

注：前面我们讲解了*Node*的许多模块，如*http*，*fs*，*path*...但也有好多模块现在没有进行讲解（我们将在遇到的时候进行讲解），因为这些模块在实际使用中并不是那么频繁，希望大家自己去官网查看文档，这对理解学习*Node*有相当大的帮助，同时也是必经之路。

## 网络编程基本概念

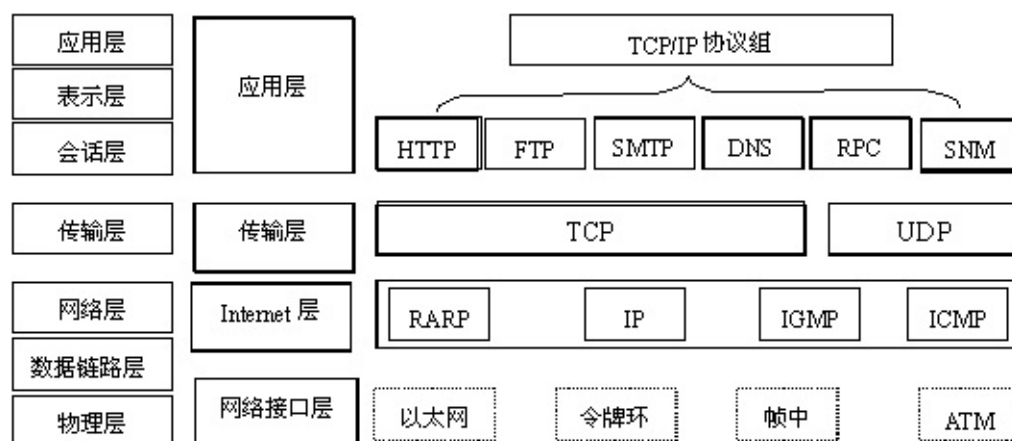
通过使用套接字来达到进程间通信目的的编程就是网络编程

通常情况下，我们要使用网络提供的功能，可以有以下几种方式：

1. 使用应用软件提供的网络通信功能来获取网络服务，如浏览器，它在应用层上使用*http*协议，在传输层基于*TCP*协议；
2. 在命令行方式下使用*shell* 命令获取系统提供的网络服务，如*telnet*，*ftp*等；
3. 使用编程的方式通过系统调用获取操作系统提供给我们的网络服

务。

对于网络编程的基础，大概要从**OSI**的七层协议模型开始了，除了OSI模型还有**TCP/IP** 协议模型



以FTP为例：

物理层到电缆连接，数据链路层到网卡，网络层路由到主机，传输层到端口，会话层维持会话，表示层表达数据格式，应用层就是具体FTP中的各种命令功能了。

在了解了这些之后，我们可以来看socket了。**socket**是在应用层和传输层之间的一个抽象层，它把**TCP/IP**层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。也就是说socket（套接字）就是将操作系统中对于传输层及其以下各层中对于网络操作的处理进行了封装，然后提供一个socket对象，供我们在应用程序中调用这个对象及其方法来达到进程间通信的目的。

上面只是一个粗劣无比的介绍，推荐阅读一下文章：

- [简单理解Socket](#)
- [全栈必备：网络编程基础](#)

## Node 中的网络编程



Node.js也提供了对socket的支持，它提供了一个`net(网络)`模块用来处理和TCP相关的操作，提供了`dgram`模块用来处理UDP(数据报)相关操作

`net` 模块给你提供了一个异步的网络封装。它包含创建服务器和客户端（称为流）的功能

## 1. 创建TCP客户端

`net`模块通过 `net.createServer` 方法创建TCP服务器

```
1. // server.js
2. const net = require('net')
3. // 创建TCP服务器
4. const server = net.createServer((socket) => {
5.   console.log('客户端连接')
6.   // 监听客户端的数据
7.   socket.on('data', (data) => {
8.     console.log('监听客户端的数据: ', data.toString())
9.   });
10.  // 监听客户端断开连接事件
11.  socket.on('end', (data) => {
12.    console.log('客户端断开连接')
13.  });
14.  // 发送数据给客户端
15.  socket.write('哈哈，我是一个测试 \r\n')
16. })
17. // 启动服务
18. server.listen(8080, () => {
19.   console.log('服务创建')
20. })
```

## 2. 创建客户端

通过 `net.connect` 方法创建客户端去连接服务器

```

1. // client.js
2. const net = require('net');
3. // 连接服务器
4. const client = net.connect({port: 8080}, () => {
5.     console.log('连接服务器');
6.     client.write('http://xingxin.me \r\n')
7. })
8. // 接收服务端的数据
9. client.on('data', (data) => {
10.    console.log('接收服务端的数据: ', data.toString())
11.    // 断开连接
12.    client.end()
13. })
14. // 断开连接
15. client.on('end', () => {
16.    console.log('断开连接')
17. })

```

现在我们在命令行中执行

```

1. $ node server.js
2.
3. # 再开一个命令行执行
4. $ node client.js

```

你会看到如下结果

```

1. $ node client.js
2. 连接服务器
3. 接收服务端的数据:  哈哈，我是一个测试
4.
5. 断开连接
6.
7. #####
8.
9. $ node server.js
10. server bound

```

11. 客户端连接
12. 监听客户端的数据: `http://xingxin.me`
- 13.
14. 客户端断开连接

这儿我们介绍了使用Net模块创建Socket服务端与客户端，关于更多使用参见[Node中文网](#)

## 简易聊天室

前面我们介绍了网络编程以及Node中的实现，我们现在就成热打铁来写一个简易的聊天室

传送门[GitHub](#)

### 1. 聊天室的服务端

```
1. const net = require('net')
2. // 创建TCP服务器
3. const server = net.createServer()
4. // 存储所有客户端socket
5. let sockets = []
6. server.on('connection', function(socket) {
7.   console.log('Got a new connection')
8.   sockets.push(socket)
9.   socket.on('data', function(data) {
10.    console.log('Got data: ', data)
11.    sockets.forEach(function(otherSocket) {
12.      if (otherSocket !== socket) {
13.        otherSocket.write(data)
14.      }
15.    })
16.  })
17.  socket.on('close', function() {
18.    console.log('A client connection closed')
19.    let index = sockets.indexOf(socket)
```

```

20.         sockets.splice(index, 1)
21.     })
22. })
23. server.on('error', function(err) {
24.     console.log('Server error: ', err.message)
25. })
26. server.on('close', function() {
27.     console.log('Server closed')
28. })
29. server.listen(8080)

```

## 2. 聊天室的客户端

```

1. const net = require('net')
2. process.stdin.resume()
3. process.stdin.setEncoding('utf8');
4. const client = net.connect({ port: 8080 }, () => {
5.     console.log('Connected to server')
6.     // 获取输入的字符串
7.     console.log('input: ')
8.     process.stdin.on('data', (data) => {
9.         // 发送输入的字符串到服务器
10.         console.log('input: ')
11.         client.write(data)
12.         // 输入 'close' 字符串时关闭连接
13.         if (data === 'close\n') {
14.             client.end()
15.         }
16.     });
17. });
18. // 获取服务端发送过来的数据
19. client.on('data', (data) => {
20.     console.log('Other user\'s input', data.toString())
21. })
22. client.on('end', () => {
23.     console.log('Disconnected from server')
24.     // 退出客户端程序
25.     process.exit()

```

```
26.  })
```

在命令行中 `node server.js` 创建一个服务器，然后用 `node client.js` 起多个客户端，便可以看见这个简陋的聊天小程序

## 相关阅读

---

- [互联网协议入门](#)
- [简单理解Socket](#)
- [全栈必备：网络编程基础](#)

### 抛砖引玉

最近一直在状态，所以这个系列有好久没更新了，后续将加紧进度

后续计划：以构建一个完整的Node web框架来巩固前面的学习

## 8. Node操作数据库

- [Node数据库操作](#)

## Node数据库操作

---

*Web应用离不开数据库的操作，我们将陆续了解Node操作[MongoDB](#)与[MySQL](#)这两个具有代表性的数据库，非关系型数据库(*NoSQL*)及关系型数据库(*SQL*)。*

- [Node操作MongoDB](#)
- [Node操作MySQL](#)

## 8. Node操作MySQL

- [Node操作MySQL](#)
  - [再讲讲 MySQL](#)
    - [MySQL安装](#)
  - [Node 操作 MySQL](#)
  - [相关链接](#)

## Node操作MySQL

上一篇我们主要介绍了Node操作MongoDB，比较了关系型数据库与非关系型数据库，并简单认识了MongoDB与MySQL这两个颇具代表性的数据库。这一节，我们将主要了解Node操作MySQL。

## 再讲讲 MySQL

MySQL是一个关系型数据库管理系统(RDBMS)，由瑞典MySQL AB公司开发，目前属于Oracle公司。MySQL是一种关联数据库管理系统。MySQL是相当优秀的数据库。

所谓关系型数据库，可以理解成“表格”吧，一个关系型数据库由一个或数个表格组成：

id	name	age
0	流口水流	21
1	liu	22
2	ogilhinn	23

看看这个表格，MySQL就是这样的结构：

- 表头(header)：每一列的名称
- 列(row)：具有相同数据类型的数据的集合
- 行(col)：每一行用来描述某个人/物的具体信息

- 值(**value**)：行的具体信息，每个值必须与该列的数据类型相同
- 键(**key**)：即表中的id，表中用来识别某个特定的人\物的方法，键的值在当前列中具有唯一性

## MySQL安装

请直接去官网看看[MySQL](#)

由于这是续篇，所以这里就不在赘述。直接进入重点

## Node 操作 MySQL

这里我们将用到[mysql](#)这个库来实现，node与mysql的交互。

```
1. $ npm install mysql
```

### 1. 连接MySQL

MySQL不会像mongodb那样自动建立一个数据库，你在使用这数据库前，你要先自己新创建个。

在MySQL命令行执行下面的命令，建立个test数据库，后面的所有操作都将在这个中间完成

```
1. create database test # 创建test数据库
2.
3. use test # 使用test
```

```
1. const mysql      = require('mysql')
2. let connection = mysql.createConnection({
3.   host      : 'localhost',
4.   user      : 'root',
5.   password  : '123456',
6.   database  : 'test'
```



```
7.  })
8.
9.  connection.connect()
10.
11. connection.query('SELECT 1 + 1 AS solution', function (error,
    results, fields) {
12.     if (error) throw error
13.     console.log('The solution is: ', results[0].solution)
14. })
15.
16. connection.end()
```

未完待续~~~

未完待续~~~

未完待续~~~

未完待续~~~

未完待续~~~

## 相关链接

---

- [个人博客](#)
- [MySQL菜鸟教程](#)

## 8. Node操作MongoDB数据库

- Node操作MongoDB数据库
  - 非关系型数据库 - NoSQL
    - NoSQL的优势
    - NoSQL的分类
  - 关系型数据库 - SQL
    - SQL的优点
    - RDBMS 数据库程序
- Node操作MongoDB
  - 1. 安装Mongoose
    - 2. 使用Mongoose进行CRUD
    - 查询
    - 创建
    - 删除
    - 修改
  - 图书管理系统
    - 1. 准备工作
    - 2. 功能设计以及路由配置
    - 3. 功能实现, Mongoose操作MongoDB
  - 相关连接

## Node操作MongoDB数据库

Web应用离不开数据库的操作，我们将陆续了解Node操作MongoDB与MySQL这两个具有代表性的数据库，非关系型数据库(NoSQL)及关系型数据库(SQL)。这一节，我们主要了解node中使用MongoDB，并与express结合实现一个简单图书管理小应用

我们来简单看看关系型数据库与非关系型数据库

## 非关系型数据库 - NoSQL

在NoSQL之前，数据库中SQL一支独秀。随着web2.0的快速发展，非关系型、分布式数据存储得到了快速的发展，访问量巨大，传统关系型数据库遇到各种瓶颈(如：高并发读写需求，高扩展性和可用性，复杂SQL，特别是多表关联查询等等)。NoSQL就诞生于此背景下，NoSQL数据库的出现，弥补了关系数据（比如MySQL）在某些方面的不足，在某些方面能极大的节省开发成本和维护成本

### NoSQL的优势

- 易扩展
- 高性能
- 灵活的数据模型
- 高可用

### NoSQL的分类

NoSQL可以大体上分为4个种类：**Key-value**、**Document-Oriented**、**Column-Family Databases**以及 **Graph-Oriented Databases**

类型	代表	特点
键值 (Key-Value)	MemcacheDB	键值数据库就像在传统语言中使用的哈希表。你可以通过key来添加、查询或者删除数据，鉴于使用主键访问，所以会获得不错的性能及扩展性。
面向文档 (Document-Oriented)	MongoDB	文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有机会对某些字段建立索引，实现关系数据库的某些功能。
列存储 (Wide Column Store/Column-Family)	Cassandra	顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，针对某一行或者某几列的查询有非常大的IO优势。
图 (Graph-Oriented)	Neo4J	图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。

## 关系型数据库 - SQL

SQL指结构化查询语言，全称是 Structured Query Language，关系数据库，是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据，简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织

### SQL的优点

- 容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解
- 使用方便：通用的SQL语言使得操作关系型数据库非常方便
- 易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率

前文提到了SQL遇到了瓶颈，并不是说SQL不行(个人认为MySQL是业内相当优秀的数据库)，只是应用场景的不同

### RDBMS 数据库程序

RDBMS 指关系型数据库管理系统(*Relational Database Management System*)。RDBMS 是 SQL 的基础，同样也是所有现代数据库系统的基础，比如 MS SQL Server、IBM DB2、Oracle、MySQL 以及 Microsoft Access。RDBMS 中的数据存储在被称为表的数据对象中。表是相关的数据项的集合，它由列和行组成

接下来我们就一起来使用Node操作MongoDB(Node操作MySQL将在下一篇介绍)，并使用它来写一个建议的图书管理小案例

## Node操作MongoDB

MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。下面这个表就展示出来MongoDB与SQL数据库的一个简单比较

SQL术语/概念	MongoDB术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
index	index	索引
table joins		表连接, MongoDB不支持
primary key	primary key	主键, MongoDB自动将_id字段设置为主键

MongoDB和Node.js特别配，因为MongoDB是基于文档的，文档是按BSON（JSON的轻量化二进制格式）存储的，增删改查等管理数据库的命令和JavaScript语法很像，这里我们选择mongoose来进行增删改查，mongoose构建在MongoDB之上，提供了Schema、Model和Document对象，用起来很方便

## 1. 安装Mongoose

```
1. $ npm install mongoose
```

安装好后 `require('mongoose')` 就可以使用了

## 2. 使用Mongoose进行CRUD

连接数据库

```
1. const mongoose = require("mongoose")
2. // 使用原生promise, mongoose自带promise不再支持了
```

```

3. mongoose.Promise = global.Promise
4.
5. const db=mongoose.connect('mongodb://localhost/test')
6.
7. db.connection.on("error", function (error) {
8.   console.log("数据库连接失败：" + error)
9. })
10.
11. db.connection.on("open", function () {
12.   console.log("数据库连接成功")
13. })

```

我们来看看Mongoose的几个名词

- **Schema** : 一种以文件形式存储的数据库模型骨架，不具备数据库的操作能力
- **Model** : 由 **Schema** 发布生成的模型，具有抽象属性和行为的数据库操作对
- **Entity** : 由 **Model** 创建的实体，他的操作也会影响数据库

**Schema** 生成 **Model** ， **Model** 创造 **Entity** ， **Model** 和 **Entity** 都可对数据库操作造成影响，但 **Model** 比 **Entity** 更具操作性

关于mongoose最重要的就是理解 **Schema** **Model** **Entity** ，它的各种方法直接去查文档使用就好。

## Schema

schema是mongoose里会用到的一种数据模式，可以理解为表结构的定义；每个schema会映射到mongodb中的一个collection，它不具备操作数据库的能力

```

1. const kittySchema = mongoose.Schema({
2.   name: String
3. })

```

## Schema.Type

`Schema.Type` 是由 `Mongoose` 内定的一些数据类型，基本数据类型都在其中，他也内置了一些 `Mongoose` 特有的 `Schema.Type`。当然，你也可以自定义 `Schema.Type`，只有满足 `Schema.Type` 的类型才能定义在 `Schema` 内

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- Objectid
- Array

## Model

定义好了Schema，接下就是生成Model。

model是由schema生成的模型，可以对数据库的操作

```
1. var Kitten = mongoose.model('Kitten', kittySchema)
```

## Entity

用Model创建Entity，`Entity` 可以对数据库操作

```
1. var silence = new Kitten({ name: 'Silence' })
2. console.log(silence.name); // 'Silence'
```

## 查询

```
1. model.find({}, field, callback) // 参数1忽略, 或为空对象则返回所有集合文档
```

```

2.
3. model.find({}, {'name':1, 'age':0}, callback) //过滤查询, 参数2:
   {'name':1, 'age':0} 查询文档的返回结果包含name , 不包含age. (_id默认是1)
4.
5. model.find({}, null, {limit:20}) // 游标操作 limit 限制返回结果数量为20个,
   如不足20个则返回所有
6.
7. model.findOne({}, callback) // 查询找到的第一个文档
8.
9. model.findById('obj._id', callback) // 查询找到的第一个文档, 只接受 _id
   的值查询

```

## 创建

```

1. // 在集合中创建一个文档
2. Model.create(doc(s), [callback])
3.
4. Entity.save(callback)

```

## 删除

```

1. Model.remove([criteria], [callback]) // 根据条件查找到并删除
2.
3. Model.findByIdAndRemove(id, [options], [callback]) // 根据id查找到并
   删除

```

## 修改

```

1. Model.update(conditions, update, [options], [callback]) // 根据参数找
   到并更新
2.
3. Model.findByIdAndUpdate(id, [update], [options], [callback]) // 根
   据id查找到并更新

```

上面简单写了几个常用操作，关于Mongoose的更多使用请移步[官网](#)



，我就不搬了（推荐阅读：[Mongoose学习参考文档—基础篇](#)）

## 图书管理系统

了解了MongoDB以及Mongoose的简单使用，我们一起来实现一个图书管理的小案例，其有最基本的增删改查，同时我们将了解到express的基本使用，同时会认识下模板引擎，但这些只是简略了解，这节的重点是Mongoose操作MongoDB

UI采用了漂亮的UIkit3

图书管理			
书名	简介	价格	操作
node-abc	《Node.js入门教程》By 流口水 github地址： <a href="https://github.com/ogilhin/n/node-abc">https://github.com/ogilhin/n/node-abc</a>	0	<a href="#">修改</a> <a href="#">删除</a>
精通CSS：高级Web标准解决方案（第2版）	Amazon**css畅销书全新改版，令人叫绝的CSS技术汇总，涵盖css和HTML5。	32	<a href="#">修改</a> <a href="#">删除</a>
深入浅出Nodejs	最好的Node书籍，没有之一	49	<a href="#">修改</a> <a href="#">删除</a>
新增			

传送门：[Github](#)

可以去github拉下来，然后 `npm install` 然后 `node index.js` 即可跑起来

### 1. 准备工作

我们先随便新建一个文件夹，然后在这个目录下

```
1. $ npm init
```

初始化项目完成后使用下载express, mongoose, nunjucks(模板引擎), body-parser(bodyParser中间件用来解析http请求体)

```
1. $ npm install express mongoose nunjucks body-parser --save
```

接下来我们新建index.js文件，在里面将express跑起来

```
1. const express = require('express')
2. const nunjucks = require('nunjucks')
3. const path = require('path')
4. const bodyParser = require('body-parser')
5. const app = express()
6.
7. // 静态文件目录
8. app.use(express.static(path.join(__dirname, 'public')))
9.
10. // 配置模板引擎
11. nunjucks.configure(path.join(__dirname, 'views'), {
12.   autoescape: true,
13.   express: app
14. })
15. app.set('view engine', 'html')
16. // 配置bodyParser
17. app.use(bodyParser.json())
18. app.use(bodyParser.urlencoded({extended: false}))
19.
20. // 路由
21. app.get('/', (req, res)=>{
22.   res.send('HELLO mongo')
23. })
24.
25. const server = app.listen(3000, function () {
26.   console.log('app listening at http://localhost:3000')
27. })
```

现在我们执行 `node index.js` 便可以跑起来了，当然更推荐使用以前介绍到的[supervisor](#)

这里我们再聊一聊Node web应用的模板引擎，这儿我们用了

**nunjucks** 这是mozilla维护的一个模板引擎，他是 **jinja2**的 javascript版本，用过python的jinja2一定会感觉很亲切，除此之外，很有很多有些的模板引擎如**ejs**，**jade**。但个人认为jade是反人类的，因此更推荐Nunjucks及ejs。当然了，这取决于大家的喜好，更多模板引擎请自行搜索了解。

我们新建一个views文件夹，放置模板。这儿只需要一个主页显示所有图书index.html，add.html添加图书，edit.html编辑图书，base.html作为基础模板，其他模板文件可以继承它

关于nunjucks，我们以他官网的那一段小代码来简单看一下

```

1.  {% extends "base.html" %} {# 继承base.html 这是注释 #}
2.
3.  {# 区块 #}
4.  {% block header %}
5.  <h1>{{ title }}</h1>
6.  {% endblock %}
7.
8.  {% block content %}
9.  <ul>
10.    {# 循环 #}
11.    {% for name, item in items %}
12.    <li>{{ name }}: {{ item }}</li>
13.    {% endfor %}
14.  </ul>
15.  {% endblock %}

```

更多使用请自行查看[官网](#)，

## 2. 功能设计以及路由配置

这儿我们就来看看，这个小的图书管理系统需要的功能。增删改查 就是新增图书，删除图书，修改图书，显示所有图书

我们就可以根据这几个功能来配置我们的路由了

```
1. const express = require('express')
2. const router = express.Router()
3.
4. // GET 首页显示全部书籍
5. router.get('/', (req, res) => {
6.
7. })
8.
9. // GET 新增书籍
10. router.get('/add', (req, res) => {
11.
12. })
13. // POST 新增书籍
14. router.post('/add', (req, res) => {
15.
16. })
17.
18. // GET 删除
19. router.get('/:bookId/remove', (req, res) => {
20.
21. })
22. // GET 编辑
23. router.get('/:bookId/edit', (req, res) => {
24.
25. })
26.
27. // POST 编辑
28. router.post('/:bookId/edit', (req, res) => {
29.
30. })
```

这儿为了项目结构更清晰，我们不把路由写在index.js中，而是提取到routes目录下，我们新建routes目录，在下面新建book.js，然后将相关路由全部放到其中并导出，

```

1. const express = require('express')
2. const router = express.Router()
3.
4. ...
5. ...
6. ...
7.
8. module.exports = router // 导出

```

然后新建一个index.js文件

```

1. module.exports = function (app) {
2.     app.use('/', require('./book'))
3. }

```

这儿这样划分，在这可能看不出太多优势，但是在大一点的应用中，我们这样配置可以让功能划分很清晰。

最后我们在入口文件index.js中将路由 `require` 进去，就可以使用了，

```

1. ...
2. routes(app)
3. ...

```

到此，前置工作就差不多了，下面我们就可以进入今天的重头戏**Mongoose**

### 3. 功能实现，Mongoose操作MongoDB

新建lib文件夹，新建mongo.js文件，连接数据库，在其中定义Schema 并发布为model

```

1. const mongoose = require('mongoose')
2. const Schema = mongoose.Schema

```

```

3.
4.  mongoose.Promise = global.Promise
5.
6.  // MongoDB会自动建立books数据库
7.  const db = mongoose.connect('mongodb://localhost:27017/books')
8.
9.  db.connection.on("error", function (error) {
10.      console.log("数据库连接失败：" + error)
11.  })
12.
13.  db.connection.on("open", function () {
14.      console.log("数据库连接成功")
15.  })
16.
17.  const BookSchema = Schema({
18.      title: {
19.          unique: true, // 唯一的不可重复
20.          type: 'String', // Schema.Type String类型
21.      },
22.      summary: 'String',
23.      price: 'Number',
24.      meta: {
25.          createdAt: {
26.              type: Date,
27.              default: Date.now()
28.          }
29.      }
30.  })
31.
32.  exports.Book = mongoose.model('Book', BookSchema)

```

新建Models文件夹，在其中新建books.js放置对MongoDB 的一些操作，这里面使用了promise，如果还不会那你就得去补补了

```

1.  const Book = require('../lib/mongo').Book
2.
3.  module.exports = {

```

```

4.     getBooks(){
5.         return Book
6.             .find({})
7.             .sort({_id: -1})
8.             .exec()
9.     },
10.    getBook(id){
11.        return Book
12.            .findById(id)
13.            .exec()
14.    },
15.    editBook(id, data){
16.        return Book
17.            .findByIdAndUpdate(id, data)
18.            .exec()
19.    },
20.    addBook(book){
21.        return Book.create(book)
22.    },
23.    delBook(id){
24.        return Book
25.            .findByIdAndRemove(id)
26.            .exec()
27.    }
28. }

```

这里面的一些方法，我们在前面讲Mongoose的时候都了解过了，想了解更多还是推荐去[官网看看](#)

最后我们就是根据不同的路由进行不同的处理了

```

1. const express = require('express')
2. const router = express.Router()
3. const BookModel = require('../models/books')
4.
5. router.get('/', (req, res) => {
6.     BookModel.getBooks()

```

```
7.     .then((books) => {
8.         res.render('index', {books})
9.     })
10. })
11.
12. router.get('/add', (req, res) => {
13.     res.render('add')
14. })
15.
16. router.post('/add', (req, res) => {
17.     let book = req.body
18.     BookModel.addBook(book)
19.     .then((result) => {
20.         res.redirect('/')
21.     })
22. })
23.
24. router.get('/:bookId/remove', (req, res) => {
25.     BookModel.delBook(req.params.bookId)
26.     .then((book) => {
27.         res.redirect('/')
28.     })
29. })
30.
31.
32. router.get('/:bookId/edit', (req, res) => {
33.     let book = req.body
34.     BookModel.getBook(req.params.bookId)
35.     .then((book) => {
36.         res.render('edit', {
37.             book,
38.             bookid: req.params.bookId
39.         })
40.     })
41. })
42.
43. router.post('/:bookId/edit', (req, res) => {
44.     let book = req.body
```



```
45.     BookModel.editBook(req.params.bookId, book)
46.     .then((result)=>{
47.         res.redirect('/')
48.     })
49. })
50.
51. module.exports = router
```

OK!!!到此，我们这小项目基本就算完成了。代码详见[GitHub](#) 作为一个学习案例，这算完成了，但其中可以优化完善的地方还很多，大家可以自行探索...

抛砖引玉

## 相关链接

- [express](#)
- [mongoose](#)
- [bodyParser](#)
- [JavaScript Promise](#)迷你书中文版



## 9. Koa快速入门教程

- Koa快速入门教程（一）
  - 一、快速开始
    - 1.1 开发环境
    - 1.2 必修的 hello world 应用：
    - 1.3 Context 对象
  - 二、路由(URL处理)
    - 2.1 手动实现简易路由
    - 2.2 使用koa-router中间件
  - 三、中间件
  - 四、模板引擎
  - 五、静态资源服务器
  - 六、请求数据的获取
    - 6.1 GET请求参数的获取
    - 6.2 POST请求数据获取
  - 参考链接

### Koa快速入门教程（一）

---

# koa

next generation web framework for node.js

*Koa* 是由 *Express* 原班人马打造的，致力于成为一个更小、更富有表现力、更健壮的 *Web* 框架，采用了 `async` 和 `await` 的方式执行异步操作。

*Koa*有v1.0与v2.0两个版本，随着node对 `async` 和 `await` 的支持，*Koa2*的正式发布，本文*Koa*均指*Koa2*

如果你还不熟悉 `async` 函数可查阅阮大的[ECMAScript 6 入门](#)

这是一篇从零开始的简易教程，话不多说，先来快速开始：hello world！

## 一、快速开始

### 1.1 开发环境

Koa 依赖 **node v7.6.0** 或 ES2015及更高版本和 `async` 方法支持，你可以使用自己喜欢的版本管理器快速安装支持的 `node` 版本

1. `$ node -v`
2. `v8.9.1`

如果你的版本号小于v7.6.0, 请自行升级。如使用nvm

在确认好环境后, 我们就可以新建一个项目, 在里面自由操练了

```
1. $ mkdir KoaTutorial && cd KoaTutorial
2.
3. $ npm i koa --save
```

## 1.2 必修的 hello world 应用:

```
1. const Koa = require('koa');
2. const app = new Koa();
3.
4. app.use(async ctx => {
5.   ctx.body = 'Hello World';
6. });
7.
8. app.listen(3000);
```

打开浏览器, 访问 <http://localhost:3000/>, 你会看到那可爱的 `Hello World`。就是这么简单的几行代码, 我们就起了一个HTTP服务,

来看看这个hello world程序, 其中前两行和后一行是架设一个HTTP 服务。中间的则是对用户访问的处理。`ctx` 则是Koa所提供的 `Context` 对象(上下文), `ctx.body=` 则是 `ctx.response.body=` 的 alias(别名), 这是响应体设置的API。

## 1.3 Context 对象

Koa Context 将 node 的 `request` 和 `response` 对象封装到单个对象中, 为编写 web 应用程序和 API 提供了许多有用的方法。上例的 `ctx.body = ''` 即是发送给用户内容, 它是 `ctx.response.body` 的

简写(更多请查阅[官网](#))。 `ctx.response` 代表 HTTP Response。 `ctx.request` 代表 HTTP Request。

## 二、路由(URL处理)

### 2.1 手动实现简易路由

koa是个极简的web框架，简单到连路由模块都没有配备，我们先来可以根据 `ctx.request.url` 或者 `ctx.request.path` 获取用户请求的路径，来实现简单的路由。

```
1. const Koa = require('koa');
2. const app = new Koa();
3.
4. app.use(async ctx => {
5.   let _html = '404 NotFound'
6.   switch (ctx.url) {
7.     case '/':
8.       _html = '<h1>Index</h1>';
9.       break;
10.    case '/adout':
11.      _html = '<h1>About</h1>';
12.      break;
13.    case '/hello':
14.      _html = '<h1>world</h1>';
15.      break;
16.    default:
17.      break;
18.   }
19.   ctx.body = _html;
20. });
21.
22. app.listen(3000);
```

运行这段代码，访问<http://localhost:3000/hello>将看见

world, 访问<http://localhost:3000/about>将看见返回about, 访问<http://localhost:3000>将看见Index。是不是很有成就感...但是这也太麻烦了吧。如果依靠`ctx.request.url`去手动处理路由, 将会写很多代码, 这时候就需要对应的路由中间件来对路由进行控制:

koa-router

## 2.2 使用koa-router中间件

下载并引入koa-router

```
1. npm i koa-router --save
```

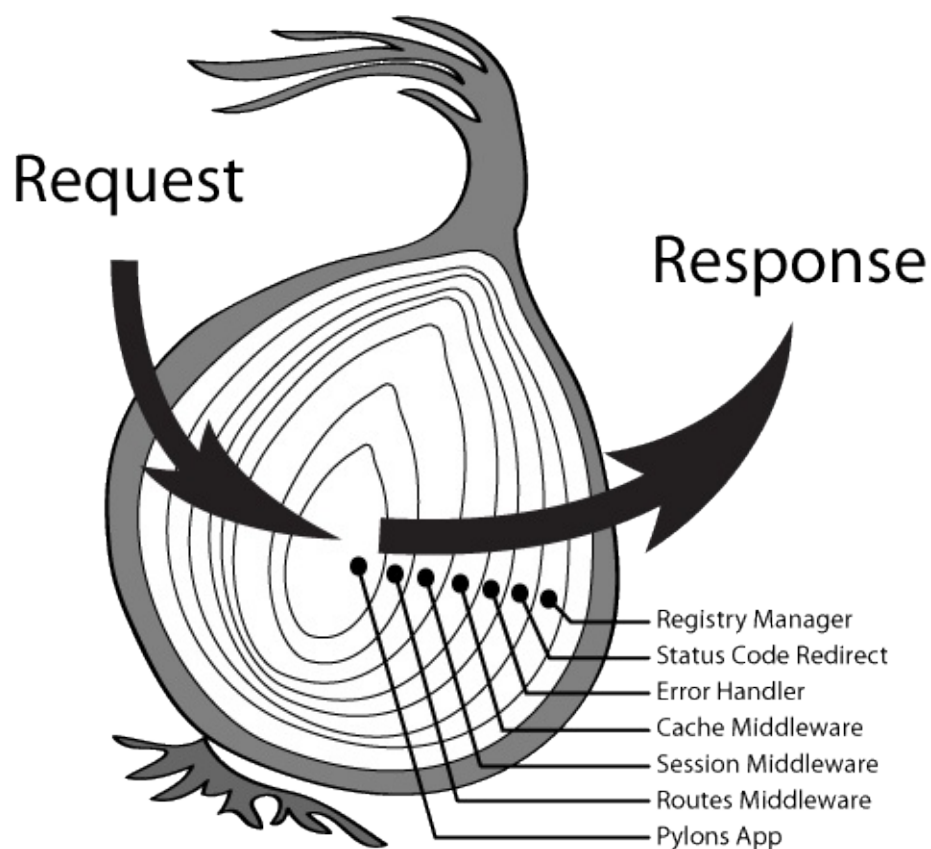
```
1. const Koa = require('koa');
2. const Router = require('koa-router');
3.
4. const app = new Koa();
5. const router = new Router();
6.
7. router.get('/', async (ctx) => {
8.   let html = `
9.     <ul>
10.       <li><a href="/hello">helloworld</a></li>
11.       <li><a href="/about">about</a></li>
12.     </ul>
13.   `
14.   ctx.body = html
15. }).get('/hello', async (ctx) => {
16.   ctx.body = 'helloworld'
17. }).get('/about', async (ctx) => {
18.   ctx.body = 'about'
19. })
20.
21. app.use(router.routes(), router.allowedMethods())
22.
23. app.listen(3000);
```

运行这个 demo，我们将看到与上栗一样的效果。在这儿我们使用到了第三方中间件。

### 三、中间件

Koa 的最大特色，也是最重要的一个设计，就是中间件 (middleware) Koa 应用程序是一个包含一组中间件函数的对象，它是按照类似堆栈的方式组织和执行的。Koa中使用 `app.use()` 用来加载中间件，基本上Koa 所有的功能都是通过中间件实现的。每个中间件默认接受两个参数，第一个参数是 Context 对象，第二个参数是 `next` 函数。只要调用 `next` 函数，就可以把执行权转交给下一个中间件。

下图为经典的Koa洋葱模型



我们来运行Koa官网这个小例子：

```

1. const Koa = require('koa');
2. const app = new Koa();
3.
4. // x-response-time
5.
6. app.use(async (ctx, next) => {
7.   const start = Date.now();
8.   await next();
9.   const ms = Date.now() - start;
10.  ctx.set('X-Response-Time', `${ms}ms`);
11. });
12.
13. // logger
14.
15. app.use(async (ctx, next) => {
16.   const start = Date.now();
17.   await next();
18.   const ms = Date.now() - start;
19.   console.log(`${ctx.method} ${ctx.url} - ${ms}`);
20. });
21.
22. // response
23.
24. app.use(async ctx => {
25.   ctx.body = 'Hello World';
26. });
27.
28. app.listen(3000);

```

上面的执行顺序就是：请求 ==> x-response-time中间件 ==> logger中间件 ==> 响应中间件 ==> logger中间件 ==> response-time中间件 ==> 响应。通过这个顺序我们可以发现这是个栈结构以“先进后出”（first-in-last-out）的顺序执行。Koa已经有了很多好用的中间件 (<https://github.com/koajs/koa/wiki#middleware>) 你需要的常用功能基本上都有人实现了



## 四、模板引擎

在实际开发中，返回给用户的网页往往都写成模板文件。Koa 先读取模板文件，然后将这个模板返回给用户，这事我们就需要使用模板引擎了，关于Koa的模版引擎，我们只需要安装koa模板使用中间件[koa-views](#) 然后在下载你喜欢的模板引擎([支持列表](#))便可以愉快的使用了。如安装使用[ejs](#)

```
1. # 安装koa模板使用中间件
2. $ npm i --save koa-views
3.
4. # 安装ejs模板引擎
5. $ npm i --save ejs
```

```
1. const Koa = require('koa')
2. const views = require('koa-views')
3. const path = require('path')
4. const app = new Koa()
5.
6. // 加载模板引擎
7. app.use(views(path.join(__dirname, './view'), {
8.   extension: 'ejs'
9. }))
10.
11. app.use(async (ctx) => {
12.   let title = 'Koa2'
13.   await ctx.render('index', {
14.     title,
15.   })
16. })
17.
18. app.listen(3000)
```

```
./view/index.ejs
```

模板

```
1. <!DOCTYPE html>
2. <html>
3. <head>
4.     <title><%= title %></title>
5. </head>
6. <body>
7.     <h1><%= title %></h1>
8.     <p>EJS Welcome to <%= title %></p>
9. </body>
10. </html>
```

打开<http://localhost:3000/>，你将看到返回了页面：

## Koa2

EJS Welcome to Koa2

关于ejs语法请访问ejs官网学习：<https://github.com/mde/ejs>

## 五、静态资源服务器

网站一般都提供静态资源（图片、字体、样式表、脚本.....），我们可以自己实现一个静态资源服务器，但这没必要，[koa-static](#)模块封装了这部分功能。

```
1. $ npm i --save koa-static
```

```
1. const Koa = require('koa')
2. const path = require('path')
3. const static = require('koa-static')
4.
5. const app = new Koa()
6.
7. // 静态资源目录对于相对入口文件index.js的路径
8. const staticPath = './static'
9.
10. app.use(static(
11.   path.join(__dirname, staticPath)
12. ))
13.
14.
15. app.use(async (ctx) => {
16.   ctx.body = 'hello world'
17. })
18.
19. app.listen(3000)
```

我们访问<http://localhost:3000/css/app.css> 将返回 `app.css` 的内容，访问<http://localhost:3000/koa2.png>我们将看见返回下图

# koa

next generation web framework for node.js

## 六、请求数据的获取

前文我们主要都在处理数据的响应，这儿我们来了解下Koa获取请求数据，主要为 `GET` 和 `POST` 方式。

### 6.1 GET请求参数的获取

在koa中，获取GET请求数据源头是koa中request对象中的query方法或querystring方法，query返回是格式化好的参数对象，querystring返回的是请求字符串。

- 请求对象 `ctx.query`(或 `ctx.request.query`)，返回如 `{ a:1, b:2 }`
- 请求字符串 `ctx.querystring`(或 `ctx.request.querystring`)，返回如 `a=1&b=2`

```
1. const Koa = require('koa')
2. const app = new Koa()
3.
```

```

4. app.use( async ( ctx ) => {
5.   const url = ctx.url
6.   const query = ctx.query
7.   const querystring = ctx.querystring
8.
9.   ctx.body = {
10.    url,
11.    query,
12.    querystring
13.  }
14. })
15.
16. app.listen(3000)

```

运行程序并访问<http://localhost:3000/?page=2&limit=10>, 我们将得到如下结果

```

1. { "url": "/?page=2&limit=10", "query":
    { "page": "2", "limit": "10"}, "querystring": "page=2&limit=10" }

```

对了，在这儿推荐个插件：[JSONView](#)，用了它你将得到格式化json数据，如下：

```

1. {
2.   url: "/?page=2&limit=10",
3.   query: {
4.     page: "2",
5.     limit: "10"
6.   },
7.   querystring: "page=2&limit=10"
8. }

```

更多Koa Request API 请查看<http://koa.js.com/#request>

## 6.2 POST请求数据获取

对于POST请求的处理，koa2没有封装获取参数的方法，需要通过自己解析上下文context中的原生node.js请求对象req，将POST表单数据解析成querystring（例如：`a=1&b=2&c=3`），再将querystring解析成JSON格式（例如：`{"a": "1", "b": "2", "c": "3"}`），我们来直接使用koa-bodyparser 模块从 POST 请求的数据体里面提取键值对。

```
1. const Koa = require('koa')
2. const app = new Koa()
3. const bodyParser = require('koa-bodyparser')
4.
5. // 使用koa-bodyparser中间件
6. app.use(bodyParser())
7.
8. app.use(async (ctx) => {
9.
10.   if (ctx.url === '/' && ctx.method === 'GET') {
11.     // 当GET请求时候返回表单页面
12.     let html = `
13.       <h1>koa-bodyparser</h1>
14.       <form method="POST" action="/">
15.         Name:<input name="name" /><br/>
16.         Age:<input name="age" /><br/>
17.         Email: <input name="email" /><br/>
18.         <button type="submit">submit</button>
19.       </form>
20.     `
21.     ctx.body = html
22.   } else if (ctx.url === '/' && ctx.method === 'POST') {
23.     // 当POST请求的时候，中间件koa-bodyparser解析POST表单里的数据，并显示出来
24.     ctx.body = ctx.request.body
25.   } else {
26.     // 404
27.     ctx.body = '<h1>404 Not Found</h1>'
28.   }
```

```
29. })  
30.  
31. app.listen(3000)
```

运行程序，填写并提交表单，请求结果为：

```
1. {  
2.   name: "ogilhinn",  
3.   age: "120",  
4.   email: "ogilhinn@gmail.com"  
5. }
```

关于更多的Koa知识快打开搜索引擎搜索([常用的搜索引擎技巧])继续学习吧，后续将继续数据库的操作以及实现一个简单的小案例。

## 参考链接

---

- [Koa官网](#)
- [Koa进阶学习笔记](#)
- [Koa 框架教程](#)
- [Koa wiki](#)

## 10. Node.js使用Nodemailer发送邮件

- [Node.js使用Nodemailer发送邮件](#)
  - [Nodemailer简介](#)
  - [安装使用](#)
  - [发出个真实的邮件](#)
  - [更多配置](#)
  - [好看的HTML邮件](#)
    - [使用模板引擎](#)
  - [HTML email 框架推荐](#)
    - [最后福利干货来了](#)
    - [36个国内外精致电子邮件HTML模版](#)
      - [关注公众号【JavaScript之禅】回复【 666 】，免费获取](#)

## Node.js使用Nodemailer发送邮件

原文链接：[Node.js使用Nodemailer发送邮件](#)

电子邮件是一种用电子手段提供信息交换的通信方式，是互联网应用最广的服务。通过网络的电子邮件系统，用户可以以非常低廉的价格（不管发送到哪里，都只需负担网费）、非常快速的方式（几秒钟之内可以发送到世界上任何指定的目的地），与世界上任何一个角落的网络用户联系。

在很多项目中，我们都会遇到邮件注册，邮件反馈等需求。在node中收发电子邮件也非常简单，因为强大的社区有各种各样的包可以供我们直接使用。[Nodemailer](#)包就可以帮助我们快速实现发送邮件的功能。

Github源码：<https://github.com/ogilhinn/node->



[abc/tree/master/lesson10](#)

## Nodemailer简介

---

Nodemailer是一个简单易用的Node.js邮件发送组件

官网地址: <https://nodemailer.com>

GitHub地址: <https://github.com/nodemailer/nodemailer>

Nodemailer的主要特点包括:

- 支持Unicode编码
- 支持Window系统环境
- 支持HTML内容和普通文本内容
- 支持附件(传送大附件)
- 支持HTML内容中嵌入图片
- 支持SSL/STARTTLS安全的邮件发送
- 支持内置的transport方法和其他插件实现的transport方法
- 支持自定义插件处理消息
- 支持XOAUTH2登录验证

## 安装使用

---

首先,我们肯定是要下载安装 注意: **Node.js v6+**

```
1. npm install nodemailer --save
```

打开官网可以看见一个小例子

```
1. 'use strict';
2. const nodemailer = require('nodemailer');
3.
```

```

4. // Generate test SMTP service account from ethereal.email
5. // Only needed if you don't have a real mail account for testing
6. nodemailer.createTestAccount((err, account) => {
7.
8.     // create reusable transporter object using the default SMTP
    transport
9.     let transporter = nodemailer.createTransport({
10.         host: 'smtp.ethereal.email',
11.         port: 587,
12.         secure: false, // true for 465, false for other ports
13.         auth: {
14.             user: account.user, // generated ethereal user
15.             pass: account.pass // generated ethereal password
16.         }
17.     });
18.
19.     // setup email data with unicode symbols
20.     let mailOptions = {
21.         from: '"Fred Foo ?" <foo@blurdybloop.com>', // sender
    address
22.         to: 'bar@blurdybloop.com, baz@blurdybloop.com', // list of
    receivers
23.         subject: 'Hello 🍌', // Subject line
24.         text: 'Hello world?', // plain text body
25.         html: '<b>Hello world?</b>' // html body
26.     };
27.
28.     // send mail with defined transport object
29.     transporter.sendMail(mailOptions, (error, info) => {
30.         if (error) {
31.             return console.log(error);
32.         }
33.         console.log('Message sent: %s', info.messageId);
34.         // Preview only available when sending through an Ethereal
    account
35.         console.log('Preview URL: %s',
        nodemailer.getTestMessageUrl(info));
36.

```

```

37.         // Message sent: <b658f8ca-6296-ccf4-8306-
           87d57a0b4321@blurdybloop.com>
38.         // Preview URL:
           https://ethereal.email/message/WaQKMgKddxQDoou...
39.     });
40. });

```

这个小例子是生成了Ethereal的账户进行邮件发送演示的。但是这多没意思，我们来使用自己的邮箱来发送邮件

## 发出个真实的邮件

这里我使用了我的qq邮箱给163邮箱发送email。

```

1.  'use strict';
2.
3.  const nodemailer = require('nodemailer');
4.
5.  let transporter = nodemailer.createTransport({
6.      // host: 'smtp.ethereal.email',
7.      service: 'qq', // 使用了内置传输发送邮件 查看支持列表：
           https://nodemailer.com/smtp/well-known/
8.      port: 465, // SMTP 端口
9.      secureConnection: true, // 使用了 SSL
10.     auth: {
11.         user: 'xxxxxx@qq.com',
12.         // 这里密码不是qq密码，是你设置的smtp授权码
13.         pass: 'xxxxxx',
14.     }
15. });
16.
17. let mailOptions = {
18.     from: '"JavaScript之禅" <xxxxx@qq.com>', // sender address
19.     to: 'xxxxxxxx@163.com', // list of receivers
20.     subject: 'Hello', // Subject line
21.     // 发送text或者html格式
22.     // text: 'Hello world?', // plain text body

```

```

23.   html: '<b>Hello world?</b>' // html body
24. };
25.
26. // send mail with defined transport object
27. transporter.sendMail(mailOptions, (error, info) => {
28.   if (error) {
29.     return console.log(error);
30.   }
31.   console.log('Message sent: %s', info.messageId);
32.   // Message sent: <04ec7731-cc68-1ef6-303c-61b0f796b78f@qq.com>
33. });

```

运行程序，成功将返回messageId。这是便可以去收件箱查看这个新邮件啦



这里我们需要注意，auth.pass 不是邮箱账户的密码而是smtp的授权码。

到此我们就掌握了发邮件的基本操作。

## 更多配置

- CC: Carbon Copy(抄送)，用于通知相关的人，收件人可以看到都邮件都抄送给谁了。一般回报工作或跨部门沟通时，都会CC给收件人的领导一份

- BCC:Blind Carbon Copy(暗抄送), 也是用于通知相关的人, 但是收件人是看不到邮件被密送给谁了。
- attachments: 附件

更多配置项: <https://nodemailer.com/message/>

这里我们就不演示CC、BCC了, 请自行尝试。我们来试试发送附件

```
1. ...
2. // 只需添加attachments配置项即可
3. attachments: [
4.   { // utf-8 string as an attachment
5.     filename: 'text.txt',
6.     content: 'hello world!'
7.   },
8.   {
9.     filename: 'ZenQcode.png',
10.    path: path.resolve(__dirname, 'ZenQcode.png'),
11.   }
12. ]
13. ...
```

发送email, 就可以收到一个内容为hello world的text.txt文件, 以及一个我公众号的二维码。

## 好看的HTML邮件

HTML Email 编写指南:

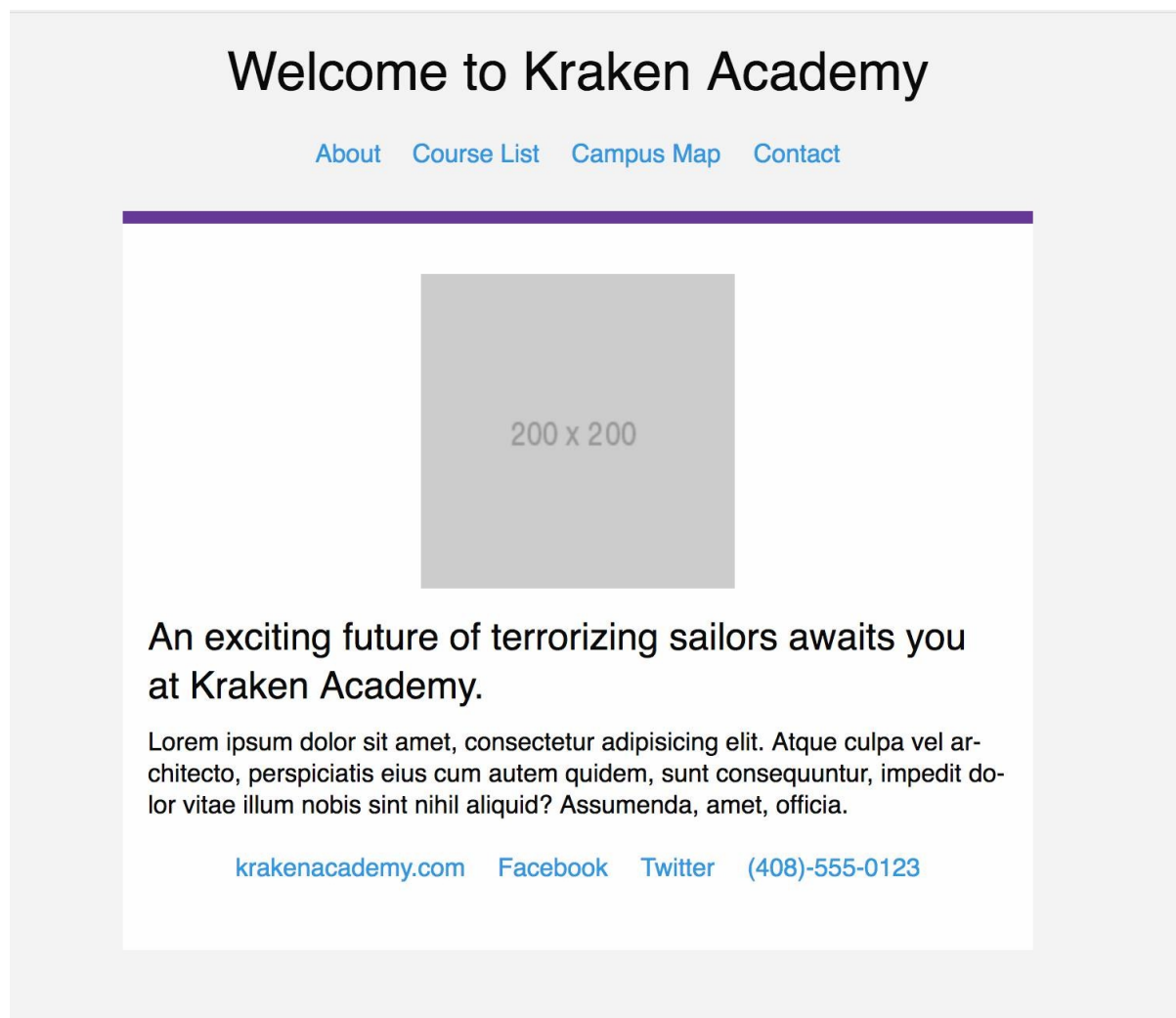
[http://www.ruanyifeng.com/blog/2013/06/html\\_email.html](http://www.ruanyifeng.com/blog/2013/06/html_email.html)

这儿, 我们使用Foundation for Emails:

<https://foundation.zurb.com/emails.html>的模板

```
1.  'use strict';
2.
3.  const nodemailer = require('nodemailer');
4.  const ejs = require('ejs');
5.  const fs = require('fs');
6.  const path = require('path');
7.
8.  let transporter = nodemailer.createTransport({
9.    // host: 'smtp.ethereal.email',
10.    service: 'qq', // 使用内置传输发送邮件 查看支持列表:
        https://nodemailer.com/smtp/well-known/
11.    port: 465, // SMTP 端口
12.    secureConnection: true, // 使用 SSL
13.    auth: {
14.      user: 'xxxxxx@qq.com',
15.      // 这里密码不是qq密码, 是你设置的smtp授权码
16.      pass: 'xxxxxx',
17.    }
18.  });
19.
20.  let mailOptions = {
21.    from: '"JavaScript之禅" <xxxxxx@qq.com>', // sender address
22.    to: 'xxxxxxxx@163.com', // list of receivers
23.    subject: 'Hello', // Subject line
24.    // 发送text或者html格式
25.    // text: 'Hello world?', // plain text body
26.    html: fs.createReadStream(path.resolve(__dirname, 'email.html'))
        // 流
27.  };
28.
29.  // send mail with defined transport object
30.  transporter.sendMail(mailOptions, (error, info) => {
31.    if (error) {
32.      return console.log(error);
33.    }
34.    console.log('Message sent: %s', info.messageId);
35.    // Message sent: <04ec7731-cc68-1ef6-303c-61b0f796b78f@qq.com>
36.  });
```

运行程序，你将如愿以偿手打如下Email。样式可能会有细微偏差



上面email中我们用了外链的图片，我们也可以使用附件的方式，将图片嵌入进去。给附件加个 `cid` 属性即可。

```
1. ...
2. let mailOptions = {
3.   ...
4.   html: '', // html body
5.   attachments: [
6.     {
7.       filename: 'ZenQcode.png',
8.       path: path.resolve(__dirname, 'ZenQcode.png'),
9.       cid: '01',
```

```

10.     }
11.   ]
12. };
13. ...

```

## 使用模板引擎

邮件信息一般都不是固定的，我们可以引入模板引擎对HTML内容进行渲染。

这里我们使用Ejs: <https://github.com/mde/ejs/>来做演示

```
1. $ npm install ejs --save
```

*ejs*具体语法请参看官方文档

先建立一个email.ejs文件

```

1. <h1>hello <%= title %></h1>
2. <p><%= desc %></p>

```

修改我们的js文件

```

1. ...
2. const template =
    ejs.compile(fs.readFileSync(path.resolve(__dirname, 'email.ejs'),
    'utf8'));
3.
4. const html = template({
5.   title: 'Ejs',
6.   desc: '使用Ejs渲染模板',
7. });
8.
9. let mailOptions = {
10.   from: '"JavaScript之禅" <xxxxx@qq.com>', // sender address
11.   to: 'xxxxx@163.com', // list of receivers

```



```

12.   subject: 'Hello', // Subject line
13.   html: html, // html body
14. };
15. ...

```

到此，你的邮箱将收到一个动态渲染的hello Ejs。

本文到此告一段落，在此基础上你可以实现更多有用的功能

## HTML email 框架推荐

- MJML: <https://mjml.io/>
- emailframe <http://emailframe.work/>
- Foundation for Emails 2:  
<https://foundation.zurb.com/emails.html>
- responsive HTML email template:  
<https://github.com/leemunroe/responsive-html-email-template>
- campaignmonitor: <https://www.campaignmonitor.com/a/>

左手代码，右手砖，抛砖引玉

如果你知道更多好用HTML email资源，留言交流让更多人知道。

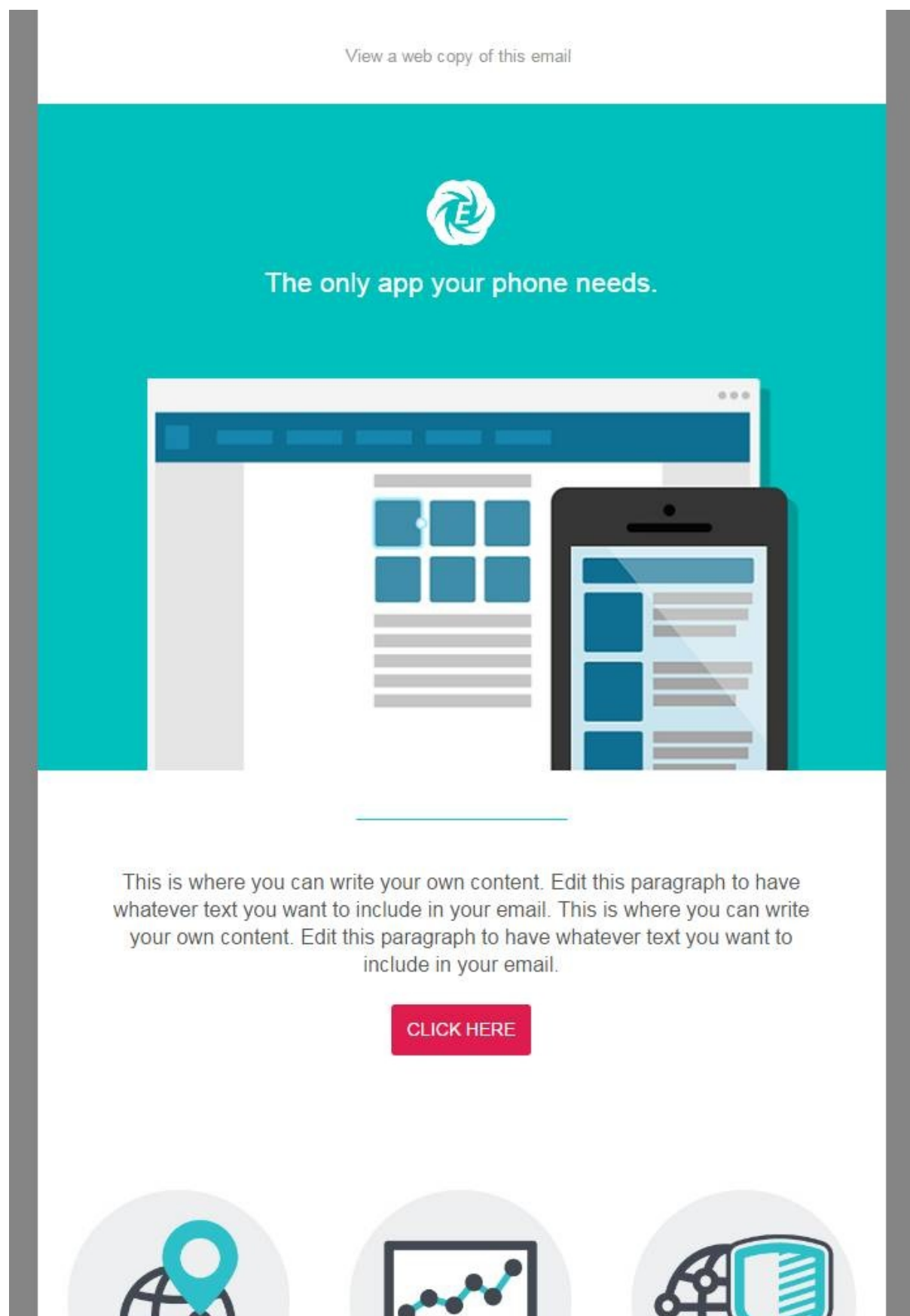
## 最后福利干货来了

...

...

...

## 36个国内外精致电子邮件HTML模版



如邮件无法正常显示，请点击[此处查看](#)



**亲爱的 用户名：**

盐，是海洋的结晶。

对知乎而言，认真、专业、友善的知友们，就是知识海洋析出的智慧之盐。

作为知乎「盐」系品牌最重要的活动，一年一度的「知乎盐 Club」很快又将和大家见面了。

与往届不同，第四届知乎盐 Club「荣誉会员」将首次采用知友投票的方式。这 150 名候选人中，谁是你心目中的知乎盐 Club「荣誉会员」？

快来投票吧！

[立即投票](#)

© 2017 你的公司名称

[退订](#)

如邮件无法显示，请点击[此处查看](#)

## 知乎 每周精选

第 258 期

### 柯洁和 AlphaGo 的第二盘棋值得关注之处有哪些？



高飞龙，雕翎不腐，鼎镬不锈。

2302 赞同

今天柯洁的发挥可以说非常出色，完成了另一种风格的测试。前天我说柯洁虽然看上去最后只输了1/4个子，但全场落后一步找不到拼的机会，AlphaGo发挥出色到可以说为，稳健到让柯洁想玉碎都找不到...

[阅读全文](#)



### 深圳有哪些好吃又不贵的餐厅？



吴泽泳，个人微信号：zeyong\_wu

1166 赞同

这道题我大约在三个多月乃至四个月前开始关注，一直没有回答，因为一家餐厅，要做到便宜又好吃，那是非常非常困难的。我先说说我的筛选标准吧：便宜：1. 人均100以下，食量特别大的不算。...

[阅读全文](#)



### 中国的食品安全状况有多糟糕？



孙亚飞，分子美食家/专栏<http://zhuanlan.zhihu.com/renchouduo> 341 赞同

谢@batmanBatman - 知乎 邀请；简单来说吧，中国的食品安全问题并没有太糟糕，至少和我们的物质水平相匹配，甚至还更优秀一些。不说别的。东南亚国家的发展水平和我国接近（各国贫富差距还是蛮...

[阅读全文](#)

触目惊心的结果，近86%的受访人认为经常接触到变质食品，而认为自己每周中最突出的是认为自己吃到农药...

关注公众号【JavaScript之禅】回复【 666 】，免费获取



长按二维码小白变大神

## JavaScript 之禅

关注前端开发，关注技术成长

汇集最新前端资讯

JavaScript、Node.js、HTML5/CSS3等

一系列教程和经验分享

## 11. node爬虫：送你一大波美腿图

- [node爬虫：送你一大波美腿图](#)

### node爬虫：送你一大波美腿图

---



# Node相关入门资料