

Node.js 教程

简单的说 Node.js 就是**运行在服务端的 JavaScript**。

- Node.js 是一个基于 Chrome JavaScript 运行时建立的一个平台。
- Node.js 是一个事件驱动 I/O 服务端 JavaScript 环境，基于 Google 的 V8 引擎，V8 引擎执行 Javascript 的速度非常快，性能非常好。

1. Node.js 创建第一个应用

在我们创建 Node.js 第一个 "Hello, World!" 应用前，让我们先了解下 Node.js 应用是由哪几部分组成的：

1. **引入 required 模块**：我们可以使用 **require** 指令来载入 Node.js 模块。
2. **创建服务器**：服务器可以监听客户端的请求，类似于 Apache、Nginx 等 HTTP 服务器。
3. **接收请求与响应请求** 服务器很容易创建，客户端可以使用浏览器或终端发送 HTTP 请求，服务器接收请求后返回响应数据。

1.1 步骤一、引入 required 模块

我们使用 **require** 指令来载入 http 模块，并将实例化的 HTTP 赋值给变量 http，实例如下：

```
1 | var http = require("http");
```

1.2 步骤二、创建服务器

接下来我们使用 `http.createServer()` 方法创建服务器，并使用 `listen` 方法绑定 8888 端口。函数通过 `request`, `response` 参数来接收和响应数据。

实例如下，在你项目的根目录下创建一个叫 `server.js` 的文件，并写入以下代码：

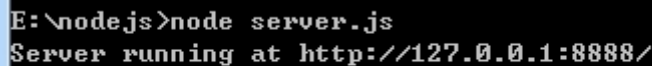
```
1 | var http = require('http');
2 |
3 | http.createServer(function (request, response) {
4 |
5 |     // 发送 HTTP 头部
6 |     // HTTP 状态值：200 : OK
7 |     // 内容类型：text/plain
8 |     response.writeHead(200, {'Content-Type': 'text/plain'});
9 |
10 |    // 发送响应数据 "Hello world"
11 |    response.end('Hello world\n');
12 | }).listen(8888);
13 |
14 | // 终端打印如下信息
15 | console.log('Server running at http://127.0.0.1:8888/');
```

1.3 步骤三、接收请求与响应请求

以上代码我们完成了一个可以工作的 HTTP 服务器。

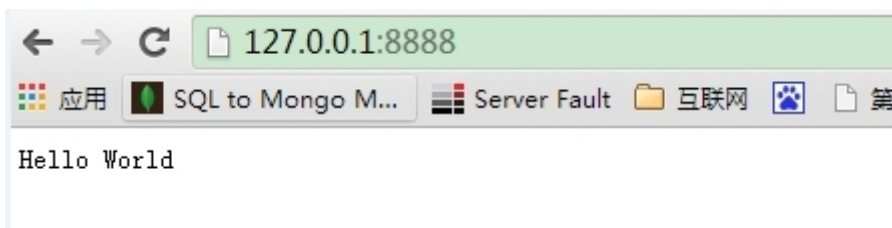
使用 **node** 命令执行以上的代码：

```
1 node server.js
2 Server running at http://127.0.0.1:8888/
```



```
E:\nodejs>node server.js
Server running at http://127.0.0.1:8888/
```

接下来，打开浏览器访问 <http://127.0.0.1:8888/>，你会看到一个写着 "Hello World" 的网页。



1.4 总结

分析Node.js 的 HTTP 服务器：

- 第一行请求 (require) Node.js 自带的 http 模块，并且把它赋值给 http 变量。
- 接下来我们调用 http 模块提供的函数：createServer。这个函数会返回 一个对象，这个对象有一个叫做 listen 的方法，这个方法有一个数值参数，指定这个 HTTP 服务器监听的端口号。

2. NPM使用介绍

NPM是随同NodeJS一起安装的包管理工具，能解决NodeJS代码部署上的很多问题，常见的使用场景有以下几种：

- 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。

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 "**npm -v**" 来测试是否成功安装。

如果你安装的是旧版本的 npm，可以很容易得通过 npm 命令来升级，命令如下：

```
1 $ sudo npm install npm -g
```

如果是 Window 系统使用以下命令即可：

```
1 npm install npm -g
```

使用淘宝镜像的命令：

```
1 npm install -g cnpm --registry=https://registry.npmmirror.com
```

2.1 使用 npm 命令安装模块

npm 安装 Node.js 模块语法格式如下：

```
1 | $ npm install <Module Name>
```

以下实例，我们使用 npm 命令安装常用的 Node.js web 框架模块 **express**：

```
1 | $ npm install express
```

安装好之后，express 包就放在了工程目录下的 node_modules 目录中，因此在代码中只需要通过 **require('express')** 的方式就好，无需指定第三方包路径。

```
1 | var express = require('express');
```

2.2 全局安装与本地安装

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有 -g 而已，比如

```
1 | npm install express          # 本地安装
2 | npm install express -g      # 全局安装
```

如果出现以下错误：

```
1 | npm err! Error: connect ECONNREFUSED 127.0.0.1:8087
```

解决办法为：

```
1 | $ npm config set proxy null
```

2.2.1 本地安装

- \1. 将安装包放在 ./node_modules 下（运行 npm 命令时所在的目录），如果没有 node_modules 目录，会在当前执行 npm 命令的目录下生成 node_modules 目录。
- \2. 可以通过 require() 来引入本地安装的包。

2.2.2 全局安装

- \1. 将安装包放在 /usr/local 下或者你 node 的安装目录。
- \2. 可以直接在命令行里使用。

如果你希望具备两者功能，则需要在两个地方安装它或使用 **npm link**。

接下来我们使用全局方式安装 express

```
1 | $ npm install express -g
```

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

```
1 | express@4.13.3 node_modules/express
2 | └─ escape-html@1.0.2
```

```

3 | └─ range-parser@1.0.2
4 | └─ merge-descriptors@1.0.0
5 | └─ array-flatten@1.1.1
6 | └─ cookie@0.1.3
7 | └─ utils-merge@1.0.0
8 | └─ parseurl@1.3.0
9 | └─ cookie-signature@1.0.6
10 | └─ methods@1.1.1
11 | └─ fresh@0.3.0
12 | └─ vary@1.0.1
13 | └─ path-to-regexp@0.1.7
14 | └─ content-type@1.0.1
15 | └─ etag@1.7.0
16 | └─ serve-static@1.10.0
17 | └─ content-disposition@0.5.0
18 | └─ depd@1.0.1
19 | └─ qs@4.0.0
20 | └─ finalhandler@0.4.0 (unpipe@1.0.0)
21 | └─ on-finished@2.3.0 (ee-first@1.1.1)
22 | └─ proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)
23 | └─ debug@2.2.0 (ms@0.7.1)
24 | └─ type-is@1.6.8 (media-typer@0.3.0, mime-types@2.1.6)
25 | └─ accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.6)
26 | └─ send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-
    errors@1.3.1)

```

2.2.3 查看安装信息

你可以使用以下命令来查看所有全局安装的模块：

```

1 | $ npm list -g
2 |
3 | └─┬─ cnpm@4.3.2
4 |   │ └─ auto-correct@1.0.0
5 |   │ └─ bagpipe@0.3.5
6 |   │ └─ colors@1.1.2
7 |   │ └─ commander@2.9.0
8 |   │   └─ graceful-readlink@1.0.1
9 |   │ └─ cross-spawn@0.2.9
10 |   │   └─ lru-cache@2.7.3
11 |   .....

```

如果要查看某个模块的版本号，可以使用命令如下：

```

1 | $ npm list grunt
2 |
3 | projectName@projectVersion /path/to/project/folder
4 | └─ grunt@0.4.1

```

2.3 使用 package.json

package.json 位于模块的目录下，用于定义包的属性。接下来让我们来看下 express 包的 package.json 文件，位于 node_modules/express/package.json 内容：

```
1  {
2    "name": "express",
3    "description": "Fast, unopinionated, minimalist web framework",
4    "version": "4.13.3",
5    "author": {
6      "name": "TJ Holowaychuk",
7      "email": "tj@vision-media.ca"
8    },
9    "contributors": [
10     {
11       "name": "Aaron Heckmann",
12       "email": "aaron.heckmann+github@gmail.com"
13     },
14     {
15       "name": "Ciaran Jessup",
16       "email": "ciaranj@gmail.com"
17     },
18     {
19       "name": "Douglas Christopher Wilson",
20       "email": "doug@somethingdoug.com"
21     },
22     {
23       "name": "Guillermo Rauch",
24       "email": "rauchg@gmail.com"
25     },
26     {
27       "name": "Jonathan Ong",
28       "email": "me@jongleberry.com"
29     },
30     {
31       "name": "Roman Shtylman",
32       "email": "shtylman+expressjs@gmail.com"
33     },
34     {
35       "name": "Young Jae Sim",
36       "email": "hanul@hanul.me"
37     }
38   ],
39   "license": "MIT",
40   "repository": {
41     "type": "git",
42     "url": "git+https://github.com/strongloop/express.git"
43   },
44   "homepage": "http://expressjs.com/",
45   "keywords": [
46     "express",
47     "framework",
48     "sinatra",
49     "web",
50     "rest",
51     "restful",
52     "router",
```

```
53     "app",
54     "api"
55 ],
56 "dependencies": {
57     "accepts": "~1.2.12",
58     "array-flatten": "1.1.1",
59     "content-disposition": "0.5.0",
60     "content-type": "~1.0.1",
61     "cookie": "0.1.3",
62     "cookie-signature": "1.0.6",
63     "debug": "~2.2.0",
64     "depd": "~1.0.1",
65     "escape-html": "1.0.2",
66     "etag": "~1.7.0",
67     "finalhandler": "0.4.0",
68     "fresh": "0.3.0",
69     "merge-descriptors": "1.0.0",
70     "methods": "~1.1.1",
71     "on-finished": "~2.3.0",
72     "parseurl": "~1.3.0",
73     "path-to-regexp": "0.1.7",
74     "proxy-addr": "~1.0.8",
75     "qs": "4.0.0",
76     "range-parser": "~1.0.2",
77     "send": "0.13.0",
78     "serve-static": "~1.10.0",
79     "type-is": "~1.6.6",
80     "utils-merge": "1.0.0",
81     "vary": "~1.0.1"
82 },
83 "devDependencies": {
84     "after": "0.8.1",
85     "ejs": "2.3.3",
86     "istanbul": "0.3.17",
87     "marked": "0.3.5",
88     "mocha": "2.2.5",
89     "should": "7.0.2",
90     "supertest": "1.0.1",
91     "body-parser": "~1.13.3",
92     "connect-redis": "~2.4.1",
93     "cookie-parser": "~1.3.5",
94     "cookie-session": "~1.2.0",
95     "express-session": "~1.11.3",
96     "jade": "~1.11.0",
97     "method-override": "~2.3.5",
98     "morgan": "~1.6.1",
99     "multiparty": "~4.1.2",
100    "vhost": "~3.0.1"
101 },
102 "engines": {
103     "node": ">= 0.10.0"
104 },
105 "files": [
106     "LICENSE",
107     "History.md",
108     "Readme.md",
109     "index.js",
110     "lib/"
```

```
111 ],
112   "scripts": {
113     "test": "mocha --require test/support/env --reporter spec --bail --
check-leaks test/ test/acceptance/",
114     "test-ci": "istanbul cover node_modules/mocha/bin/_mocha --report
lcovonly -- --require test/support/env --reporter spec --check-leaks test/
test/acceptance/",
115     "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --require
test/support/env --reporter dot --check-leaks test/ test/acceptance/",
116     "test-tap": "mocha --require test/support/env --reporter tap --check-
leaks test/ test/acceptance/"
117   },
118   "gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",
119   "bugs": {
120     "url": "https://github.com/strongloop/express/issues"
121   },
122   "_id": "express@4.13.3",
123   "_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
124   "_from": "express@*",
125   "_npmVersion": "1.4.28",
126   "_npmUser": {
127     "name": "dougwilson",
128     "email": "doug@somethingdoug.com"
129   },
130   "maintainers": [
131     {
132       "name": "tjholowaychuk",
133       "email": "tj@vision-media.ca"
134     },
135     {
136       "name": "jongleberry",
137       "email": "jonathanrichardong@gmail.com"
138     },
139     {
140       "name": "dougwilson",
141       "email": "doug@somethingdoug.com"
142     },
143     {
144       "name": "rfeng",
145       "email": "enjoyjava@gmail.com"
146     },
147     {
148       "name": "aredridel",
149       "email": "aredridel@dinhe.net"
150     },
151     {
152       "name": "strongloop",
153       "email": "callback@strongloop.com"
154     },
155     {
156       "name": "defunctzombie",
157       "email": "shtylman@gmail.com"
158     }
159   ],
160   "dist": {
161     "shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
162     "tarball": "http://registry.npmjs.org/express/-/express-4.13.3.tgz"
163   },
```

```
164   "directories": {},
165   "_resolved": "https://registry.npmjs.org/express/-/express-4.13.3.tgz",
166   "readme": "ERROR: No README data found!"
167 }
```

2.3.1 Package.json 属性说明

- **name** - 包名。
- **version** - 包的版本号。
- **description** - 包的描述。
- **homepage** - 包的官网 url。
- **author** - 包的作者姓名。
- **contributors** - 包的其他贡献者姓名。
- **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node_module 目录下。
- **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
- **main** - main 字段指定了程序的主入口文件，require('moduleName') 就会加载这个文件。这个字段的默认值是模块根目录下面的 index.js。
- **keywords** - 关键字

2.4 卸载模块

我们可以使用以下命令来卸载 Node.js 模块。

```
1 | $ npm uninstall express
```

卸载后，你可以到 /node_modules/ 目录下查看包是否还存在，或者使用以下命令查看：

```
1 | $ npm ls
```

2.5 更新模块

我们可以使用以下命令更新模块：

```
1 | $ npm update express
```

2.6 搜索模块

使用以下来搜索模块：

```
1 | $ npm search express
```

2.7 创建模块

创建模块，package.json 文件是必不可少的。我们可以使用 NPM 生成 package.json 文件，生成的文件包含了基本的结果。

```
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) runoob          # 模块名
13 | version: (1.0.0)
14 | description: Node.js 测试模块(www.runoob.com) # 描述
15 | entry point: (index.js)
16 | test command: make test
17 | git repository: https://github.com/runoob/runoob.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": "runoob",
25 |   "version": "1.0.0",
26 |   "description": "Node.js 测试模块(www.runoob.com)",
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。在最后输入 "yes" 后会生成 package.json 文件。

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

```

1 | $ npm adduser
2 | Username: mcmohd
3 | Password:
4 | Email: (this is public) mcmohd@gmail.com

```

接下来我们就用以下命令来发布模块：

```

1 | $ npm publish

```

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

2.8 版本号

使用 NPM 下载和发布代码时都会接触到版本号。NPM 使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为X.Y.Z三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

- 如果只是修复bug，需要更新Z位。
- 如果是新增了功能，但是向下兼容，需要更新Y位。
- 如果有大变动，向下不兼容，需要更新X位。

版本号有了这个保证后，在申明第三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如"argv": "0.0.x"表示依赖于0.0.x系列的最新版argv。

NPM支持的所有版本号范围指定方式可以查看[官方文档](#)。

2.9 NPM 常用命令

除了本章介绍的部分外，NPM还提供了很多功能，package.json里也有很多其它有用的字段。

除了可以在npmjs.org/doc/查看官方文档外，这里再介绍一些NPM常用命令。

NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。

- NPM提供了很多命令，例如 `install` 和 `publish`，使用 `npm help` 可查看所有命令。
- 使用 `npm help <command>` 可查看某条命令的详细帮助，例如 `npm help install`。
- 在 `package.json` 所在目录下使用 `npm install . -g` 可先在本地安装当前命令行程序，可用于发布前的本地测试。
- 使用 `npm update <package>` 可以把当前目录下 `node_modules` 子目录里边的对应模块更新至最新版本。
- 使用 `npm update <package> -g` 可以把全局安装的对应命令行程序更新至最新版。
- 使用 `npm cache clear` 可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。
- 使用 `npm unpublish <package>@<version>` 可以撤销发布自己发布过的某个版本代码。

2.10 使用淘宝 NPM 镜像

由于国内直接使用 npm 的官方镜像是非常慢的，这里推荐使用淘宝 NPM 镜像。

淘宝 NPM 镜像是一个完整 npmjs.org 镜像，你可以用此代替官方版本(只读)，同步频率目前为 10分钟一次以保证尽量与官方服务同步。

你可以使用淘宝定制的 `cnpm` (gzip 压缩支持) 命令行工具代替默认的 `npm`:

```
1 | $ npm install -g cnpm --registry=https://registry.npmmirror.com
```

这样就可以使用 `cnpm` 命令来安装模块了：

```
1 | $ cnpm install [name]
```

3. Node.js REPL(交互式解释器)

Node.js REPL(Read Eval Print Loop:交互式解释器) 表示一个电脑的环境，类似 Windows 系统的终端或 Unix/Linux shell，我们可以在终端中输入命令，并接收系统的响应。

Node 自带了交互式解释器，可以执行以下任务：

- **读取** - 读取用户输入，解析输入的 Javascript 数据结构并存储在内存中。
- **执行** - 执行输入的数据结构
- **打印** - 输出结果
- **循环** - 循环操作以上步骤直到用户两次按下 `ctrl-c` 按钮退出。

Node 的交互式解释器可以很好的调试 Javascript 代码。

3.1 启动 Node 的终端

我们可以输入以下命令来启动 Node 的终端：

```
1 | $ node
2 | >
```

这时我们就可以在 > 后输入简单的表达式，并按下回车键来计算结果。

3.2 简单的表达式运算

接下来让我们在 Node.js REPL 的命令行窗口中执行简单的数学运算：

```
1 $ node
2 > 1 + 4
3 5
4 > 5 / 2
5 2.5
6 > 3 * 6
7 18
8 > 4 - 1
9 3
10 > 1 + ( 2 * 3 ) - 4
11 3
12 >
```

3.3 使用变量

你可以将数据存储在变量中，并在你需要的时候使用它。

变量声明需要使用 **var** 关键字，如果没有使用 var 关键字变量会直接打印出来。

使用 **var** 关键字的变量可以使用 console.log() 来输出变量。

```
1 $ node
2 > x = 10
3 10
4 > var y = 10
5 undefined
6 > x + y
7 20
8 > console.log("Hello world")
9 Hello world
10 undefined
11 > console.log("www.runoob.com")
12 www.runoob.com
13 undefined
```

3.4 多行表达式

Node REPL 支持输入多行表达式，这就有点类似 JavaScript。接下来让我们来执行一个 do-while 循环：

```
1 $ node
2 > var x = 0
3 undefined
4 > do {
5 ... x++;
6 ... console.log("x: " + x);
7 ... } while ( x < 5 );
8 x: 1
9 x: 2
10 x: 3
```

```
11 x: 4
12 x: 5
13 undefined
14 >
```

... 三个点的符号是系统自动生成的，你回车换行后即可。Node 会自动检测是否为连续的表达式。

3.5 下划线(_)变量

你可以使用下划线(_)获取上一个表达式的运算结果：

```
1 $ node
2 > var x = 10
3 undefined
4 > var y = 20
5 undefined
6 > x + y
7 30
8 > var sum = _
9 undefined
10 > console.log(sum)
11 30
12 undefined
13 >
```

3.6 REPL 命令

- **ctrl + c** - 退出当前终端。
- **ctrl + c 按下两次** - 退出 Node REPL。
- **ctrl + d** - 退出 Node REPL。
- **向上/向下 键** - 查看输入的历史命令
- **tab 键** - 列出当前命令
- **.help** - 列出使用命令
- **.break** - 退出多行表达式
- **.clear** - 退出多行表达式
- **.save *filename*** - 保存当前的 Node REPL 会话到指定文件
- **.load *filename*** - 载入当前 Node REPL 会话的文件内容。

3.7 停止 REPL

前面我们已经提到按下两次 **ctrl + c** 键就能退出 REPL：

```
1 $ node
2 >
3 (^C again to quit)
4 >
```

4. Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程（多线程）依托于回调来实现，但不能说使用了回调后程序就异步化了。

回调函数在完成任务后就会被调用，Node 使用了大量的回调函数，Node 所有 API 都支持回调函数。

例如，我们可以一边读取文件，一边执行其他命令，在文件读取完成后，我们将文件内容作为回调函数的参数返回。这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。

回调函数一般作为函数的最后一个参数出现：

```
1 function foo1(name, age, callback) { }
2 function foo2(value, callback1, callback2) { }
```

4.1 阻塞代码实例

创建一个文件 input.txt，内容如下：

```
1 新迎二十大，奋进新征程！
```

创建 main.js 文件, 代码如下：

```
1 var fs = require("fs");
2
3 var data = fs.readFileSync('input.txt');
4
5 console.log(data.toString());
6 console.log("程序执行结束！");
```

以上代码执行结果如下：

```
1 $ node main.js
2 新迎二十大，奋进新征程！
3
4 程序执行结束！
```

4.2 非阻塞代码实例

创建一个文件 input.txt，内容如下：

```
1 新迎二十大，奋进新征程！
```

创建 main.js 文件, 代码如下：

```
1 var fs = require("fs");
2
3 fs.readFile('input.txt', function (err, data) {
4     if (err) return console.error(err);
5     console.log(data.toString());
6 });
7
8 console.log("程序执行结束！");
```

以上代码执行结果如下：

```
1 $ node main.js
2 程序执行结束！
3 新迎二十大，奋进新征程！
```

以上两个实例我们了解了阻塞与非阻塞调用的不同。第一个实例在文件读取完后才执行程序。第二个实例我们不需要等待文件读取完，这样就可以在读取文件时同时执行接下来的代码，大大提高了程序的性能。

因此，阻塞是按顺序执行的，而非阻塞是不需要按顺序的，所以如果需要处理回调函数的参数，我们就需要写在回调函数内。

5. Node.js 事件循环

Node.js 是单进程单线程应用程序，但是因为 V8 引擎提供的异步执行回调接口，通过这些接口可以处理大量的并发，所以性能非常高。

Node.js 几乎每一个 API 都是支持回调函数的。

Node.js 基本上所有的事件机制都是用设计模式中[观察者模式](#)实现。

Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数。

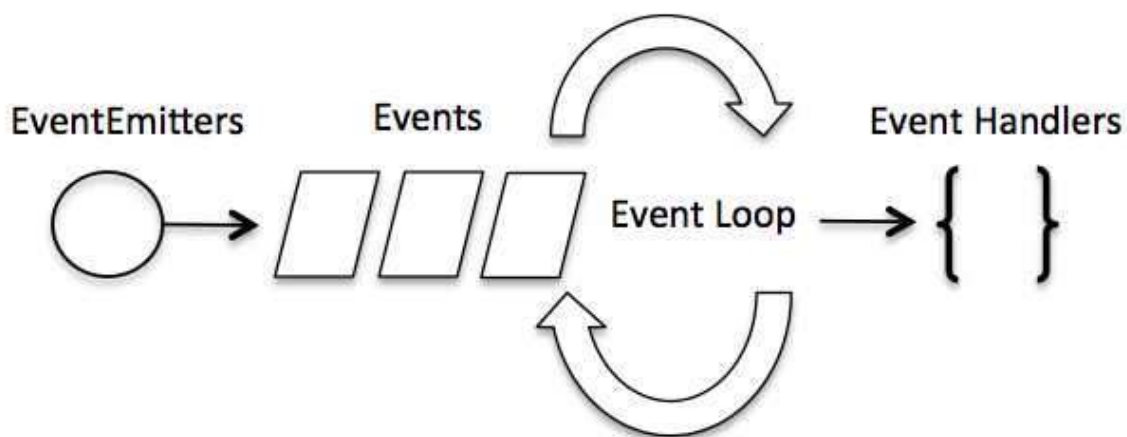
5.1 事件驱动程序

Node.js 使用事件驱动模型，当web server接收到请求，就把它关闭然后进行处理，然后去服务下一个web请求。

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

这个模型非常高效可扩展性非常强，因为 webserver 一直接受请求而不等待任何读写操作。（这也称之为非阻塞式IO或者事件驱动IO）

在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。



整个事件驱动的流程就是这么实现的，非常简洁。有点类似于观察者模式，事件相当于一个主题 (Subject)，而所有注册到这个事件上的处理函数相当于观察者 (Observer)。

Node.js 有多个内置的事件，我们可以通过引入 events 模块，并通过实例化 EventEmitter 类来绑定和监听事件，如下实例：

```
1 // 引入 events 模块
2 var events = require('events');
3 // 创建 EventEmitter 对象
4 var EventEmitter = new events.EventEmitter();
```

以下程序绑定事件处理程序：

```
1 // 绑定事件及事件的处理程序
2 emitter.on('eventName', eventHandler);
```

我们可以通过程序触发事件：

```
1 // 触发事件
2 emitter.emit('eventName');
```

5.1.1 实例

创建 main.js 文件，代码如下所示：

```
1 // 引入 events 模块
2 var events = require('events');
3 // 创建 EventEmitter 对象
4 var EventEmitter = new events.EventEmitter();
5
6 // 创建事件处理程序
7 var connectHandler = function connected() {
8     console.log('连接成功。');
9
10    // 触发 data_received 事件
11    emitter.emit('data_received');
12 }
13
14 // 绑定 connection 事件处理程序
15 emitter.on('connection', connectHandler);
16
17 // 使用匿名函数绑定 data_received 事件
18 emitter.on('data_received', function(){
19     console.log('数据接收成功。');
20 });
21
22 // 触发 connection 事件
23 emitter.emit('connection');
24
25 console.log("程序执行完毕。");
```

接下来让我们执行以上代码：

```
1 $ node main.js
2 连接成功。
3 数据接收成功。
4 程序执行完毕。
```

5.2 Node 应用程序是如何工作的？

在 Node 应用程序中，执行异步操作的函数将回调函数作为最后一个参数，回调函数接收错误对象作为第一个参数。

接下来让我们来重新看下前面的实例，创建一个 input.txt 文件，内容如下：

```
1 | 新迎二十大，奋进新征程！
```

创建 main.js 文件，代码如下：

```
1 | var fs = require("fs");
2 |
3 | fs.readFile('input.txt', function (err, data) {
4 |     if (err){
5 |         console.log(err.stack);
6 |         return;
7 |     }
8 |     console.log(data.toString());
9 | });
10 | console.log("程序执行完毕");
```

以上程序中 fs.readFile() 是异步函数用于读取文件。如果在读取文件过程中发生错误，错误 err 对象就会输出错误信息。

如果没发生错误，readFile 跳过 err 对象的输出，文件内容就通过回调函数输出。

执行以上代码，执行结果如下：

```
1 | 程序执行完毕
2 | 新迎二十大，奋进新征程！
```

接下来我们删除 input.txt 文件，执行结果如下所示：

```
1 | 程序执行完毕
2 | Error: ENOENT, open 'input.txt'
```

因为文件 input.txt 不存在，所以输出了错误信息。

6. Node.js EventEmitter

Node.js 所有的异步 I/O 操作在完成时都会发送一个事件到事件队列。

Node.js 里面的许多对象都会分发事件：一个 net.Server 对象会在每次有新连接时触发一个事件，一个 fs.readStream 对象会在文件被打开的时候触发一个事件。所有这些产生事件的对象都是 events.EventEmitter 的实例。

6.1 EventEmitter 类

events 模块只提供了一个对象：events.EventEmitter。EventEmitter 的核心就是事件触发与事件监听器功能的封装。

你可以通过require("events");来访问该模块。


```

1 // 引入 events 模块
2 var events = require('events');
3 // 创建 EventEmitter 对象
4 var EventEmitter = new events.EventEmitter();

```

EventEmitter 对象如果在实例化时发生错误，会触发 error 事件。当添加新的监听器时，newListener 事件会触发，当监听器被移除时，removeListener 事件被触发。

下面我们用一个简单的例子说明 EventEmitter 的用法：

```

1 //event.js 文件
2 var EventEmitter = require('events').EventEmitter;
3 var event = new EventEmitter();
4 event.on('some_event', function() {
5     console.log('some_event 事件触发');
6 });
7 setTimeout(function() {
8     event.emit('some_event');
9 }, 1000);

```

执行结果如下：

运行这段代码，1 秒后控制台输出了 **'some_event 事件触发'**。其原理是 event 对象注册了事件 some_event 的一个监听器，然后通过 setTimeout 在 1000 毫秒以后向 event 对象发送事件 some_event，此时会调用 some_event 的监听器。

```

1 $ node event.js
2 some_event 事件触发

```

EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持若干个事件监听器。

当事件触发时，注册到这个事件的事件监听器被依次调用，事件参数作为回调函数参数传递。

让我们以下面的例子解释这个过程：

```

1 //event.js 文件
2 var events = require('events');
3 var emitter = new events.EventEmitter();
4 emitter.on('someEvent', function(arg1, arg2) {
5     console.log('listener1', arg1, arg2);
6 });
7 emitter.on('someEvent', function(arg1, arg2) {
8     console.log('listener2', arg1, arg2);
9 });
10 emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');

```

执行以上代码，运行的结果如下：

```

1 $ node event.js
2 listener1 arg1 参数 arg2 参数
3 listener2 arg1 参数 arg2 参数

```

以上例子中，emitter 为事件 someEvent 注册了两个事件监听器，然后触发了 someEvent 事件。

运行结果中可以看到两个事件监听器回调函数被先后调用。 这就是EventEmitter最简单的用法。

EventEmitter 提供了多个属性，如 **on** 和 **emit**。**on** 函数用于绑定事件函数，**emit** 属性用于触发一个事件。接下来我们来具体看下 EventEmitter 的属性介绍。

6.2 方法

序号	方法 & 描述
1	addListener(event, listener) 为指定事件添加一个监听器到监听器数组的尾部。
2	on(event, listener) 为指定事件注册一个监听器，接受一个字符串 event 和一个回调函数。 <code>server.on('connection', function (stream) { console.log('someone connected!'); });</code>
3	once(event, listener) 为指定事件注册一个单次监听器，即 监听器最多只会触发一次，触发后立刻解除该监听器。 <code>server.once('connection', function (stream) { console.log('Ah, we have our first user!'); });</code>
4	removeListener(event, listener) 移除指定事件的某个监听器，监听器必须是该事件已经注册过的监听器。它接受两个参数，第一个是事件名称，第二个是回调函数名称。 <code>var callback = function(stream) { console.log('someone connected!'); }; server.on('connection', callback); // ... server.removeListener('connection', callback);</code>
5	removeAllListeners([event]) 移除所有事件的所有监听器， 如果指定事件，则移除指定事件的所有监听器。
6	setMaxListeners(n) 默认情况下，EventEmitters 如果你添加的监听器超过 10 个就会输出警告信息。setMaxListeners 函数用于改变监听器的默认限制的数量。
7	listeners(event) 返回指定事件的监听器数组。
8	emit(event, [arg1], [arg2], [...]) 按监听器的顺序执行每个监听器，如果事件有注册监听返回 true，否则返回 false。

6.3 类方法

序号	方法 & 描述
1	listenerCount(emitter, event) 返回指定事件的监听器数量。

1	<code>events.EventEmitter.listenerCount(emitter, eventName)</code> //已废弃，不推荐
2	<code>events.emitter.listenerCount(eventName)</code> //推荐

6.4 事件

序号	事件 & 描述
1	newListener event - 字符串, 事件名称 listener - 处理事件函数该事件在添加新监听器时被触发。
2	removeListener event - 字符串, 事件名称 listener - 处理事件函数从指定监听器数组中删除一个监听器。需要注意的是, 此操作将会改变处于被删监听器之后的那些监听器的索引。

6.5 实例

以下实例通过 connection（连接）事件演示了 EventEmitter 类的应用。

创建 main.js 文件, 代码如下:

```
1 var events = require('events');
2 var eventEmitter = new events.EventEmitter();
3
4 // 监听器 #1
5 var listener1 = function listener1() {
6     console.log('监听器 listener1 执行。');
7 }
8
9 // 监听器 #2
10 var listener2 = function listener2() {
11     console.log('监听器 listener2 执行。');
12 }
13
14 // 绑定 connection 事件, 处理函数为 listener1
15 eventEmitter.addListener('connection', listener1);
16
17 // 绑定 connection 事件, 处理函数为 listener2
18 eventEmitter.on('connection', listener2);
19
20 var eventListeners = eventEmitter.listenerCount('connection');
21 console.log(eventListeners + " 个监听器监听连接事件。");
22
23 // 处理 connection 事件
24 eventEmitter.emit('connection');
25
26 // 移除监绑定的 listener1 函数
27 eventEmitter.removeListener('connection', listener1);
28 console.log("listener1 不再受监听。");
29
30 // 触发连接事件
31 eventEmitter.emit('connection');
32
33 eventListeners = eventEmitter.listenerCount('connection');
34 console.log(eventListeners + " 个监听器监听连接事件。");
35
36 console.log("程序执行完毕。");
```

以上代码, 执行结果如下所示:

```
1 $ node main.js
2 2 个监听器监听连接事件。
3 监听器 listener1 执行。
4 监听器 listener2 执行。
5 listener1 不再受监听。
6 监听器 listener2 执行。
7 1 个监听器监听连接事件。
8 程序执行完毕。
```

6.6 error事件

EventEmitter 定义了一个特殊的事件 error，它包含了错误的语义，我们在遇到异常的时候通常会触发 error 事件。

当 error 被触发时，EventEmitter 规定如果没有响应的监听器，Node.js 会把它当作异常，退出程序并输出错误信息。

我们一般要为会触发 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。例如：

```
1 var events = require('events');
2 var emitter = new events.EventEmitter();
3 emitter.emit('error');
```

运行时会显示以下错误：

```
1 node.js:201
2   throw e; // process.nextTick error, or 'error' event on first tick
3   ^
4 Error: Uncaught, unspecified 'error' event.
5   at EventEmitter.emit (events.js:50:15)
6   at Object.<anonymous> (/home/byvoid/error.js:5:9)
7   at Module._compile (module.js:441:26)
8   at Object..js (module.js:459:10)
9   at Module.load (module.js:348:31)
10  at Function._load (module.js:308:12)
11  at Array.0 (module.js:479:10)
12  at EventEmitter._tickCallback (node.js:192:40)
```

6.7 继承 EventEmitter

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。为什么要这样做呢？原因有两点：

- 首先，具有某个实体功能的对象实现事件符合语义，事件的监听和发生应该是一个对象的方法。
- 其次 JavaScript 的对象机制是基于原型的，支持部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

7. Node.js Buffer(缓冲区)

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

但在处理像TCP流或文件流时，必须使用到二进制数据。因此在 Node.js中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。

在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理 I/O 操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

在 v6.0 之前创建 Buffer 对象直接使用 `new Buffer()` 构造函数来创建对象实例，但是 Buffer 对内存的权限操作相比很大，可以直接捕获一些敏感信息，所以在 v6.0 以后，官方文档里面建议使用 `Buffer.from()` 接口去创建 Buffer 对象。

7.1 Buffer 与字符编码

Buffer 实例一般用于表示编码字符的序列，比如 UTF-8、UCS2、Base64、或十六进制编码的数据。通过使用显式的字符编码，就可以在 Buffer 实例与普通的 JavaScript 字符串之间进行相互转换。

```
1 const buf = Buffer.from('runoob', 'ascii');
2
3 // 输出 72756e666662
4 console.log(buf.toString('hex'));
5
6 // 输出 cnVub29i
7 console.log(buf.toString('base64'));
```

Node.js 目前支持的字符编码包括：

- **ascii** - 仅支持 7 位 ASCII 数据。如果设置去掉高位的话，这种编码是非常快的。
- **utf8** - 多字节编码的 Unicode 字符。许多网页和其他文档格式都使用 UTF-8。
- **utf16le** - 2 或 4 个字节，小字节序编码的 Unicode 字符。支持代理对 (U+10000 至 U+10FFFF)。
- **ucs2** - **utf16le** 的别名。
- **base64** - Base64 编码。
- **latin1** - 一种把 Buffer 编码成一字节编码的字符串的方式。
- **binary** - **latin1** 的别名。
- **hex** - 将每个字节编码为两个十六进制字符。

7.2 创建 Buffer 类

Buffer 提供了以下 API 来创建 Buffer 类：

- **Buffer.alloc(size[, fill[, encoding]])**：返回一个指定大小的 Buffer 实例，如果没有设置 fill，则默认填满 0
- **Buffer.allocUnsafe(size)**：返回一个指定大小的 Buffer 实例，但是它不会被初始化，所以它可能包含敏感的数据
- **Buffer.allocUnsafeSlow(size)**
- **Buffer.from(array)**：返回一个被 array 的值初始化的新的 Buffer 实例（传入的 array 的元素只能是数字，不然就会自动被 0 覆盖）
- **Buffer.from(arrayBuffer[, byteOffset[, length]])**：返回一个新建的与给定的 ArrayBuffer 共享同一内存的 Buffer。
- **Buffer.from(buffer)**：复制传入的 Buffer 实例的数据，并返回一个新的 Buffer 实例
- **Buffer.from(string[, encoding])**：返回一个被 string 的值初始化的新的 Buffer 实例

```
1 // 创建一个长度为 10、且用 0 填充的 Buffer。
2 const buf1 = Buffer.alloc(10);
3
4 // 创建一个长度为 10、且用 0x1 填充的 Buffer。
5 const buf2 = Buffer.alloc(10, 1);
```

```

6
7 // 创建一个长度为 10、且未初始化的 Buffer。
8 // 这个方法比调用 Buffer.alloc() 更快，
9 // 但返回的 Buffer 实例可能包含旧数据，
10 // 因此需要使用 fill() 或 write() 重写。
11 const buf3 = Buffer.allocUnsafe(10);
12
13 // 创建一个包含 [0x1, 0x2, 0x3] 的 Buffer。
14 const buf4 = Buffer.from([1, 2, 3]);
15
16 // 创建一个包含 UTF-8 字节 [0x74, 0xc3, 0xa9, 0x73, 0x74] 的 Buffer。
17 const buf5 = Buffer.from('tést');
18
19 // 创建一个包含 Latin-1 字节 [0x74, 0xe9, 0x73, 0x74] 的 Buffer。
20 const buf6 = Buffer.from('tést', 'latin1');

```

7.3 写入缓冲区

7.3.1 语法

写入 Node 缓冲区的语法如下所示：

```
1 buf.write(string[, offset[, length]][, encoding])
```

7.3.2 参数

参数描述如下：

- **string** - 写入缓冲区的字符串。
- **offset** - 缓冲区开始写入的索引值，默认为 0。
- **length** - 写入的字节数，默认为 buffer.length
- **encoding** - 使用的编码。默认为 'utf8'。

根据 encoding 的字符编码写入 string 到 buf 中的 offset 位置。length 参数是写入的字节数。如果 buf 没有足够的空间保存整个字符串，则只会写入 string 的一部分。只部分解码的字符不会被写入。

7.3.3 返回值

返回实际写入的大小。如果 buffer 空间不足，则只会写入部分字符串。

7.3.4 实例

```

1 buf = Buffer.alloc(256);
2 len = buf.write("新迎二十大，奋进新征程！");
3
4 console.log("写入字节数： "+ len);

```

执行以上代码，输出结果为：

```

1 $node main.js
2 写入字节数： 14

```

7.4 从缓冲区读取数据

7.4.1 语法

读取 Node 缓冲区数据的语法如下所示：

```
1 buf.toString([encoding[, start[, end]]])
```

7.4.2 参数

参数描述如下：

- **encoding** - 使用的编码。默认为 'utf8'。
- **start** - 指定开始读取的索引位置，默认为 0。
- **end** - 结束位置，默认为缓冲区的末尾。

7.4.3 返回值

解码缓冲区数据并使用指定的编码返回字符串。

7.4.4 实例

```
1 buf = Buffer.alloc(26);
2 for (var i = 0 ; i < 26 ; i++) {
3   buf[i] = i + 97;
4 }
5
6 console.log( buf.toString('ascii'));           // 输出: abcdefghijklmnopqrstuvwxyz
7 console.log( buf.toString('ascii',0,5));       // 使用 'ascii' 编码，并输出: abcde
8 console.log( buf.toString('utf8',0,5));        // 使用 'utf8' 编码，并输出: abcde
9 console.log( buf.toString(undefined,0,5));     // 使用默认的 'utf8' 编码，并输出:
          abcde
```

执行以上代码，输出结果为：

```
1 $ node main.js
2 abcdefghijklmnopqrstuvwxyz
3 abcde
4 abcde
5 abcde
```

7.5 将 Buffer 转换为 JSON 对象

7.5.1 语法

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

```
1 buf.toJSON()
```

当字符串化一个 Buffer 实例时，[JSON.stringify\(\)](#) 会隐式地调用该 `toJSON()`。

7.5.2 返回值

返回 JSON 对象。

7.5.3 实例

```
1  const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
2  const json = JSON.stringify(buf);
3
4  // 输出: {"type":"Buffer","data":[1,2,3,4,5]}
5  console.log(json);
6
7  const copy = JSON.parse(json, (key, value) => {
8    return value && value.type === 'Buffer' ?
9      Buffer.from(value.data) :
10     value;
11  });
12
13  // 输出: <Buffer 01 02 03 04 05>
14  console.log(copy);
```

执行以上代码，输出结果为：

```
1  {"type":"Buffer","data":[1,2,3,4,5]}
2  <Buffer 01 02 03 04 05>
```

7.6 缓冲区合并

7.6.1 语法

Node 缓冲区合并的语法如下所示：

```
1  Buffer.concat(list[, totalLength])
```

7.6.2 参数

参数描述如下：

- **list** - 用于合并的 Buffer 对象数组列表。
- **totalLength** - 指定合并后 Buffer 对象的总长度。

7.6.3 返回值

返回一个多个成员合并的新 Buffer 对象。

7.6.4 实例

```
1  var buffer1 = Buffer.from('新迎二十大，');
2  var buffer2 = Buffer.from('奋进新征程！');
3  var buffer3 = Buffer.concat([buffer1,buffer2]);
4  console.log("buffer3 内容： " + buffer3.toString());
```

执行以上代码，输出结果为：

```
1  buffer3 内容： 新迎二十大，奋进新征程！
```


7.7 缓冲区比较

7.7.1 语法

Node Buffer 比较的函数语法如下所示, 该方法在 Node.js v0.12.2 版本引入:

```
1 buf.compare(otherBuffer);
```

7.7.2 参数

参数描述如下:

- **otherBuffer** - 与 **buf** 对象比较的另外一个 Buffer 对象。

7.7.3 返回值

返回一个数字, 表示 **buf** 在 **otherBuffer** 之前, 之后或相同。

7.7.4 实例

```
1 var buffer1 = Buffer.from('ABC');
2 var buffer2 = Buffer.from('ABCD');
3 var result = buffer1.compare(buffer2);
4
5 if(result < 0) {
6   console.log(buffer1 + " 在 " + buffer2 + "之前");
7 }else if(result == 0){
8   console.log(buffer1 + " 与 " + buffer2 + "相同");
9 }else {
10   console.log(buffer1 + " 在 " + buffer2 + "之后");
11 }
```

执行以上代码, 输出结果为:

```
1 ABC在ABCD之前
```

7.8 拷贝缓冲区

7.8.1 语法

Node 缓冲区拷贝语法如下所示:

```
1 buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])
```

7.8.2 参数

参数描述如下:

- **targetBuffer** - 要拷贝的 Buffer 对象。
- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length

7.8.3 返回值

没有返回值。

7.8.4 实例

```
1 var buf1 = Buffer.from('abcdefghijkl');
2 var buf2 = Buffer.from('RUNOOB');
3
4 //将 buf2 插入到 buf1 指定位置上
5 buf2.copy(buf1, 2);
6
7 console.log(buf1.toString());
```

执行以上代码，输出结果为：

```
1 | abRUNOOBijkl
```

7.9 缓冲区裁剪

Node 缓冲区裁剪语法如下所示：

```
1 | buf.slice([start[, end]])
```

7.9.1 参数

参数描述如下：

- **start** - 数字, 可选, 默认: 0
- **end** - 数字, 可选, 默认: buffer.length

7.9.2 返回值

返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。

7.9.3 实例

```
1 var buffer1 = Buffer.from('runoob');
2 // 剪切缓冲区
3 var buffer2 = buffer1.slice(0,2);
4 console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
1 | buffer2 content: ru
```

7.9.4 注意点

注意：裁剪功能返回的实际是原始缓存区 buffer 或者一部分，操作的是与原始 buffer 同一块内存区域。

```

1 // 裁剪
2 var buffer_origin = Buffer.from('runoob');
3 var buffer_slice = buffer_origin.slice(0,2);
4 console.log("buffer slice content: "+buffer_slice.toString());
5 console.log("buffer origin content: "+buffer_origin.toString());
6 buffer_slice.write("wirte"); // write buffer slice
7
8 // 裁剪前与原始字符串的改变
9 console.log("buffer slice content: "+buffer_slice.toString());
10 console.log("buffer origin content: "+buffer_origin.toString());

```

输出:

```

1 buffer slice content: ru
2 buffer origin content: runoob
3 buffer slice content: wi
4 buffer origin content: winoob
5 可以看到对裁剪返回的 buffer 进行写操作同时，也对原始 buffer 进行了写操作。

```

7.10 缓冲区长度

7.10.1 语法

Node 缓冲区长度计算语法如下所示:

```

1 buf.length;

```

7.10.2 返回值

返回 Buffer 对象所占据的内存长度。

7.10.3 实例

```

1 var buffer = Buffer.from('www.runoob.com');
2 // 缓冲区长度
3 console.log("buffer length: " + buffer.length);

```

执行以上代码，输出结果为:

```

1 buffer length: 14

```

7.11 方法参考手册

以下列出了 Node.js Buffer 模块常用的方法（注意有些方法在旧版本是没有的）：

序号	方法 & 描述
1	new Buffer(size) 分配一个新的 size 大小单位为8位字节的 buffer。注意, size 必须小于 kMaxLength, 否则, 将会抛出异常 RangeError。废弃的: 使用 Buffer.alloc() 代替 (或 Buffer.allocUnsafe())。
2	new Buffer(buffer) 拷贝参数 buffer 的数据到 Buffer 实例。废弃的: 使用 Buffer.from(buffer) 代替。
3	new Buffer(str[, encoding]) 分配一个新的 buffer, 其中包含着传入的 str 字符串。encoding 编码方式默认为 'utf8'。废弃的: 使用 Buffer.from(string[, encoding]) 代替。
4	buf.length 返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容的大小。length 是 buffer 对象所分配的内存数, 它不会随着这个 buffer 对象内容的改变而改变。
5	buf.write(string[, offset[, length]][, encoding]) 根据参数 offset 偏移量和指定的 encoding 编码方式, 将参数 string 数据写入buffer。offset 偏移量默认值是 0, encoding 编码方式默认是 utf8。length 长度是要写入的字符串的 bytes 大小。返回 number 类型, 表示写入了多少 8 位字节流。如果 buffer 没有足够的空间来放整个 string, 它将只会只写入部分字符串。length 默认是 buffer.length - offset。这个方法不会出现写入部分字符。
6	buf.writeUIntLE(value, offset, byteLength[, noAssert]) 将 value 写入到 buffer 里, 它由 offset 和 byteLength 决定, 最高支持 48 位无符号整数, 小端对齐, 例如: <code>const buf = Buffer.allocUnsafe(6); buf.writeUIntLE(0x1234567890ab, 0, 6); // 输出: <Buffer ab 90 78 56 34 12> console.log(buf);</code> noAssert 值为 true 时, 不再验证 value 和 offset 的有效性。默认是 false。
7	buf.writeUIntBE(value, offset, byteLength[, noAssert]) 将 value 写入到 buffer 里, 它由 offset 和 byteLength 决定, 最高支持 48 位无符号整数, 大端对齐。noAssert 值为 true 时, 不再验证 value 和 offset 的有效性。默认是 false。 <code>const buf = Buffer.allocUnsafe(6); buf.writeUIntBE(0x1234567890ab, 0, 6); // 输出: <Buffer 12 34 56 78 90 ab> console.log(buf);</code>
8	buf.writeIntLE(value, offset, byteLength[, noAssert]) 将value 写入到 buffer 里, 它由offset 和 byteLength 决定, 最高支持48位有符号整数, 小端对齐。noAssert 值为 true 时, 不再验证 value 和 offset 的有效性。默认是 false。
9	buf.writeIntBE(value, offset, byteLength[, noAssert]) 将value 写入到 buffer 里, 它由offset 和 byteLength 决定, 最高支持48位有符号整数, 大端对齐。noAssert 值为 true 时, 不再验证 value 和 offset 的有效性。默认是 false。
10	buf.readUIntLE(offset, byteLength[, noAssert]) 支持读取 48 位以下的无符号数字, 小端对齐。noAssert 值为 true 时, offset 不再验证是否超过 buffer 的长度, 默认为 false。
11	buf.readUIntBE(offset, byteLength[, noAssert]) 支持读取 48 位以下的无符号数字, 大端对齐。noAssert 值为 true 时, offset 不再验证是否超过 buffer 的长度, 默认为 false。
12	buf.readIntLE(offset, byteLength[, noAssert]) 支持读取 48 位以下的有符号数字, 小端对齐。noAssert 值为 true 时, offset 不再验证是否超过 buffer 的长度, 默认为 false。
13	buf.readIntBE(offset, byteLength[, noAssert]) 支持读取 48 位以下的有符号数字, 大端对齐。noAssert 值为 true 时, offset 不再验证是否超过 buffer 的长度, 默认为 false。

序号	方法 & 描述
14	buf.toString([encoding[, start[, end]]]) 根据 encoding 参数（默认是 'utf8'）返回一个解码过的 string 类型。还会根据传入的参数 start（默认是 0）和 end（默认是 buffer.length）作为取值范围。
15	buf.toJSON() 将 Buffer 实例转换为 JSON 对象。
16	buf[index] 获取或设置指定的字节。返回值代表一个字节，所以返回值的合法范围是十六进制 0x00 到 0xFF 或者十进制 0 至 255。
17	buf.equals(otherBuffer) 比较两个缓冲区是否相等，如果是返回 true，否则返回 false。
18	buf.compare(otherBuffer) 比较两个 Buffer 对象，返回一个数字，表示 buf 在 otherBuffer 之前，之后或相同。
19	buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]]) buffer 拷贝，源和目标可以相同。targetStart 目标开始偏移和 sourceStart 源开始偏移默认都是 0。sourceEnd 源结束位置偏移默认是源的长度 buffer.length。
20	buf.slice([start[, end]]]) 剪切 Buffer 对象，根据 start（默认是 0）和 end（默认是 buffer.length）偏移和裁剪了索引。负的索引是从 buffer 尾部开始计算的。
21	buf.readUInt8(offset[, noAssert]) 根据指定的偏移量，读取一个无符号 8 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。如果这样 offset 可能会超出 buffer 的末尾。默认是 false。
22	buf.readUInt16LE(offset[, noAssert]) 根据指定的偏移量，使用特殊的 endian 字节序格式读取一个无符号 16 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
23	buf.readUInt16BE(offset[, noAssert]) 根据指定的偏移量，使用特殊的 endian 字节序格式读取一个无符号 16 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
24	buf.readUInt32LE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个无符号 32 位整数，小端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
25	buf.readUInt32BE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个无符号 32 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
26	buf.readInt8(offset[, noAssert]) 根据指定的偏移量，读取一个有符号 8 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
27	buf.readInt16LE(offset[, noAssert]) 根据指定的偏移量，使用特殊的 endian 格式读取一个有符号 16 位整数，小端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。
28	buf.readInt16BE(offset[, noAssert]) 根据指定的偏移量，使用特殊的 endian 格式读取一个有符号 16 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出 buffer 的末尾。默认是 false。

序号	方法 & 描述
29	buf.readInt32LE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数，小端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出buffer的末尾。默认是 false。
30	buf.readInt32BE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出buffer的末尾。默认是 false。
31	buf.readFloatLE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位双浮点数，小端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出buffer的末尾。默认是 false。
32	buf.readFloatBE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位双浮点数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出buffer的末尾。默认是 false。
33	buf.readDoubleLE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 64 位双精度数，小端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出buffer的末尾。默认是 false。
34	buf.readDoubleBE(offset[, noAssert]) 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 64 位双精度数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 offset 可能会超出buffer的末尾。默认是 false。
35	buf.writeUInt8(value, offset[, noAssert]) 根据传入的 offset 偏移量将 value 写入 buffer。注意：value 必须是一个合法的无符号 8 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。除非你对这个参数非常有把握，否则不要使用。默认是 false。
36	buf.writeUInt16LE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的无符号 16 位整数，小端对齐。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出buffer的末尾从而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
37	buf.writeUInt16BE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的无符号 16 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出buffer的末尾从而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
38	buf.writeUInt32LE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式(LITTLE-ENDIAN:小字节序)将 value 写入buffer。注意：value 必须是一个合法的无符号 32 位整数，小端对齐。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
39	buf.writeUInt32BE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式(Big-Endian:大字节序)将 value 写入buffer。注意：value 必须是一个合法的有符号 32 位整数。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。

序号	方法 & 描述
40	buf.writeInt8(value, offset[, noAssert])
41	buf.writeInt16LE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 16 位整数。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
42	buf.writeInt16BE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 16 位整数。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
43	buf.writeInt32LE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 32 位整数。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
44	buf.writeInt32BE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 32 位整数。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
45	buf.writeFloatLE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
46	buf.writeFloatBE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
47	buf.writeDoubleLE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个有效的 64 位 double 类型的值。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
48	buf.writeDoubleBE(value, offset[, noAssert]) 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个有效的 64 位 double 类型的值。若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾而造成 value 被丢弃。除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。
49	buf.fill(value[, offset][, end]) 使用指定的 value 来填充这个 buffer。如果没有指定 offset (默认是 0) 并且 end (默认是 buffer.length)，将会填充整个 buffer。

8. Node.js Stream(流)

Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对 http 服务器发起请求的 request 对象就是一个 Stream，还有 stdout（标准输出）。

Node.js，Stream 有四种流类型：

- **Readable** - 可读操作。
- **Writable** - 可写操作。
- **Duplex** - 可读可写操作。
- **Transform** - 操作被写入数据，然后读出结果。

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。

8.1 从流中读取数据

创建 input.txt 文件，内容如下：

```
1 | 菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件，代码如下：

```
1 | var fs = require("fs");
2 | var data = '';
3 |
4 | // 创建可读流
5 | var readerStream = fs.createReadStream('input.txt');
6 |
7 | // 设置编码为 utf8。
8 | readerStream.setEncoding('UTF8');
9 |
10 | // 处理流事件 --> data, end, and error
11 | readerStream.on('data', function(chunk) {
12 |     data += chunk;
13 | });
14 |
15 | readerStream.on('end', function(){
16 |     console.log(data);
17 | });
18 |
19 | readerStream.on('error', function(err){
20 |     console.log(err.stack);
21 | });
22 |
23 | console.log("程序执行完毕");
```

以上代码执行结果如下：

```
1 | 程序执行完毕
2 | 菜鸟教程官网地址：www.runoob.com
```


8.2 写入流

创建 main.js 文件, 代码如下:

```
1  var fs = require("fs");
2  var data = '菜鸟教程官网地址: www.runoob.com';
3
4  // 创建一个可以写入的流, 写入到文件 output.txt 中
5  var writerStream = fs.createWriteStream('output.txt');
6
7  // 使用 utf8 编码写入数据
8  writerStream.write(data, 'UTF8');
9
10 // 标记文件末尾
11 writerStream.end();
12
13 // 处理流事件 --> finish、error
14 writerStream.on('finish', function() {
15     console.log("写入完成。");
16 });
17
18 writerStream.on('error', function(err){
19     console.log(err.stack);
20 });
21
22 console.log("程序执行完毕");
```

以上程序会将 data 变量的数据写入到 output.txt 文件中。代码执行结果如下:

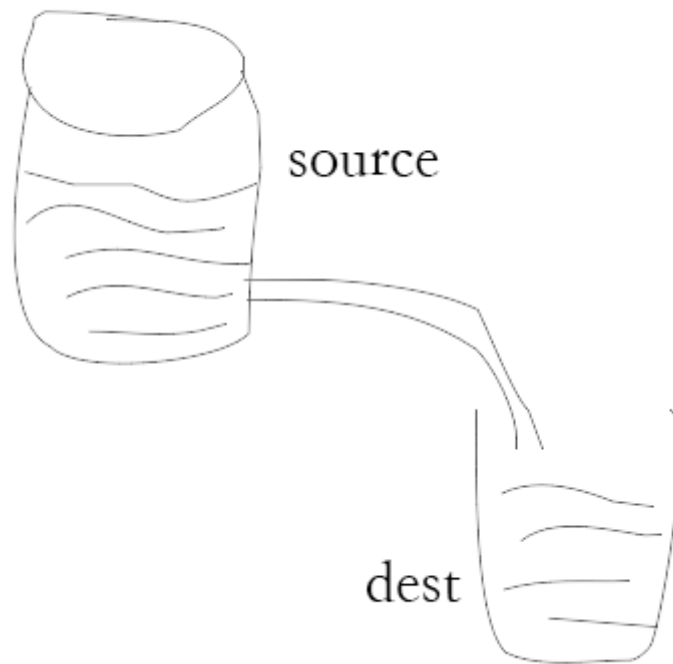
```
1  $ node main.js
2  程序执行完毕
3  写入完成。
```

查看 output.txt 文件的内容:

```
1  $ cat output.txt
2  菜鸟教程官网地址: www.runoob.com
```

8.3 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。



如上面的图片所示，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。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

```
1 菜鸟教程官网地址：www.runoob.com
2 管道流操作实例
```

创建 main.js 文件, 代码如下：

```
1  var fs = require("fs");
2
3  // 创建一个可读流
4  var readerStream = fs.createReadStream('input.txt');
5
6  // 创建一个可写流
7  var writerStream = fs.createWriteStream('output.txt');
8
9  // 管道读写操作
10 // 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
11 readerStream.pipe(writerStream);
12
13 console.log("程序执行完毕");
```

代码执行结果如下：

```
1 $ node main.js
2 程序执行完毕
```

查看 output.txt 文件的内容：

```
1 $ cat output.txt
2 菜鸟教程官网地址: www.runoob.com
3 管道流操作实例
```

8.4 链式流

链式是通过连接输出流到另外一个流并创建多个流操作链的机制。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和解压文件。

创建 compress.js 文件, 代码如下:

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

代码执行结果如下:

```
1 $ node compress.js
2 文件压缩完成。
```

执行完以上操作后, 我们可以看到当前目录下生成了 input.txt 的压缩文件 input.txt.gz。

接下来, 让我们来解压该文件, 创建 decompress.js 文件, 代码如下:

```
1 var fs = require("fs");
2 var zlib = require('zlib');
3
4 // 解压 input.txt.gz 文件为 input.txt
5 fs.createReadStream('input.txt.gz')
6   .pipe(zlib.createGunzip())
7   .pipe(fs.createWriteStream('input.txt'));
8
9 console.log("文件解压完成。");
```

代码执行结果如下:

```
1 $ node decompress.js
2 文件解压完成。
```

9. Node.js模块系统

为了让Node.js的文件可以相互调用, Node.js提供了一个简单的模块系统。

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。

9.1 引入模块

在 Node.js 中，引入一个模块非常简单，如下我们创建一个 **main.js** 文件并引入 **hello** 模块，代码如下：

```
1 var hello = require('./hello');
2 hello.world();
```

以上实例中，代码 `require('./hello')` 引入了当前目录下的 `hello.js` 文件（`./` 为当前目录，`node.js` 默认后缀为 `js`）。

Node.js 提供了 `exports` 和 `require` 两个对象，其中 `exports` 是模块公开的接口，`require` 用于从外部获取一个模块的接口，即所获取模块的 `exports` 对象。

接下来我们就来创建 `hello.js` 文件，代码如下：

```
1 exports.world = function() {
2   console.log('Hello world');
3 }
```

在以上示例中，`hello.js` 通过 `exports` 对象把 `world` 作为模块的访问接口，在 `main.js` 中通过 `require('./hello')` 加载这个模块，然后就可以直接访问 `hello.js` 中 `exports` 对象的成员函数了。

有时候我们只是想把一个对象封装到模块中，格式如下：

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

例如：

```
1 //hello.js
2 function Hello() {
3   var name;
4   this.setName = function(thyName) {
5     name = thyName;
6   };
7   this.sayHello = function() {
8     console.log('Hello ' + name);
9   };
10 };
11 module.exports = Hello;
```

这样就可以直接获得这个对象了：

```
1 //main.js
2 var Hello = require('./hello');
3 hello = new Hello();
4 hello.setName('BYvoid');
5 hello.sayHello();
```

模块接口的唯一变化是使用 `module.exports = Hello` 代替了 `exports.world = function(){}`。在外部引用该模块时，其接口对象就是要输出的 `Hello` 对象本身，而不是原先的 `exports`。

9.2 服务端的模块放在哪里

也许你已经注意到，我们已经在代码中使用了模块了。像这样：

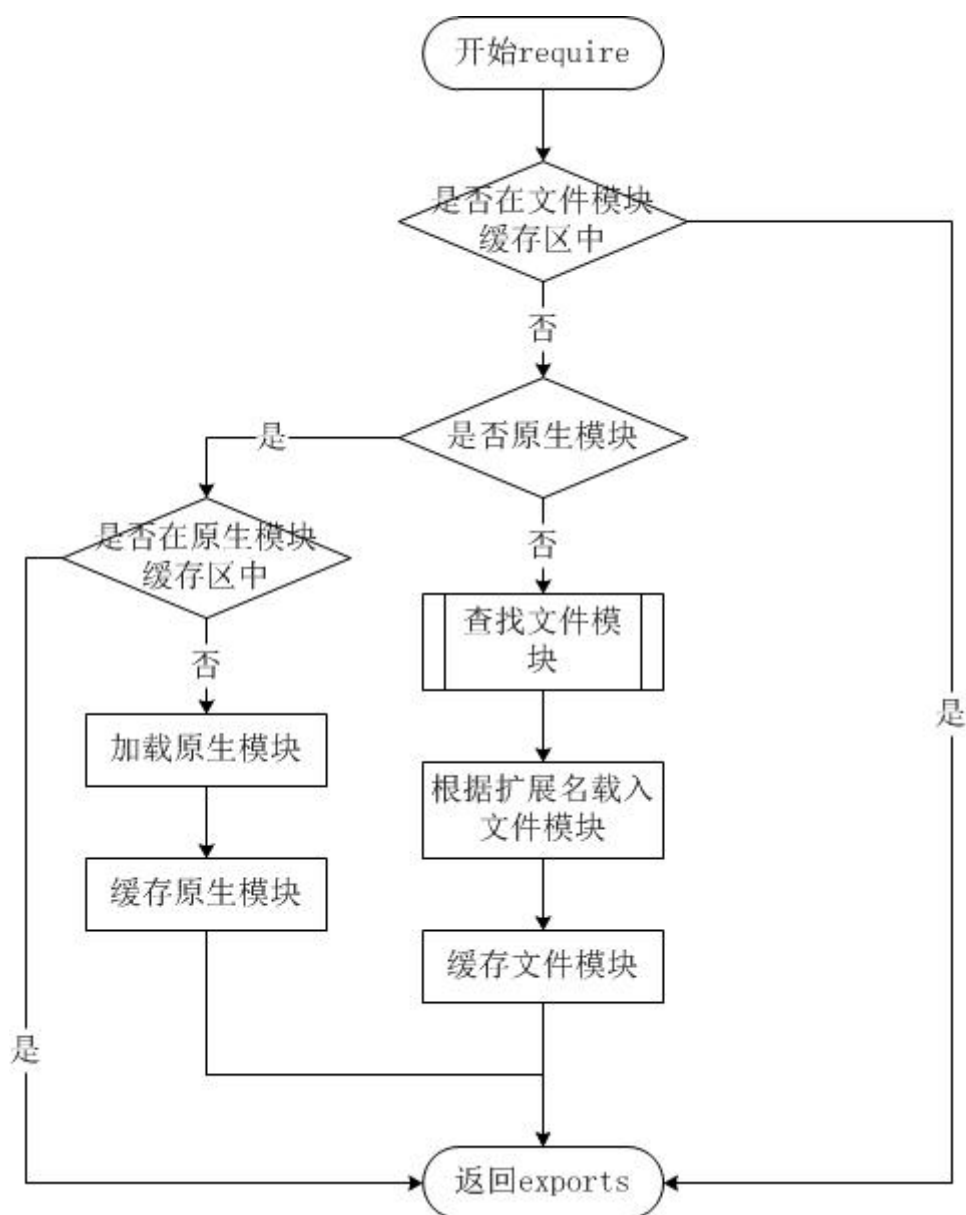
```
1 var http = require("http");
2
3 ...
4
5 http.createServer(...);
```

Node.js 中自带了一个叫做 **http** 的模块，在我们的代码中请求它并把返回值赋给一个本地变量。

这把我们本地变量变成了一个拥有所有 http 模块所提供的公共方法的对象。

Node.js 的 require 方法中的文件查找策略如下：

由于 Node.js 中存在 4 类模块（原生模块和 3 种文件模块），尽管 require 方法极其简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同。如下图所示：



9.2.1 从文件模块缓存中加载

尽管原生模块与文件模块的优先级不同，但是都会优先从文件模块的缓存中加载已经存在的模块。

9.2.2 从原生模块加载

原生模块的优先级仅次于文件模块缓存的优先级。require 方法在解析文件名之后，优先检查模块是否存在于原生模块列表中。以http模块为例，尽管在目录下存在一个 http/http.js/http.node/http.json 文件，require("http") 都不会从这些文件中加载，而是从原生模块中加载。

原生模块也有一个缓存区，同样也是优先从缓存区加载。如果缓存区没有被加载过，则调用原生模块的加载方式进行加载和执行。

9.2.3 从文件加载

当文件模块缓存中不存在，而且不是原生模块的时候，Node.js 会解析 require 方法传入的参数，并从文件系统中加载实际的文件，加载过程中的包装和编译细节在前一节中已经介绍过，这里我们将详细描述查找文件模块的过程，其中，也有一些细节值得知晓。

require方法接受以下几种参数的传递：

- http、fs、path等，原生模块。
- ./mod或../mod，相对路径的文件模块。
- /pathtomodule/mod，绝对路径的文件模块。
- mod，非原生模块的文件模块。

在路径 Y 下执行 require(X) 语句执行顺序：

```
1  1. 如果 X 是内置模块
2    a. 返回内置模块
3    b. 停止执行
4  2. 如果 X 以 '/' 开头
5    a. 设置 Y 为文件根路径
6  3. 如果 X 以 './' 或 '/' or '../' 开头
7    a. LOAD_AS_FILE(Y + X)
8    b. LOAD_AS_DIRECTORY(Y + X)
9  4. LOAD_NODE_MODULES(X, dirname(Y))
10 5. 抛出异常 "not found"
11
12 LOAD_AS_FILE(X)
13 1. 如果 X 是一个文件，将 X 作为 JavaScript 文本载入并停止执行。
14 2. 如果 X.js 是一个文件，将 X.js 作为 JavaScript 文本载入并停止执行。
15 3. 如果 X.json 是一个文件，解析 X.json 为 JavaScript 对象并停止执行。
16 4. 如果 X.node 是一个文件，将 X.node 作为二进制插件载入并停止执行。
17
18 LOAD_INDEX(X)
19 1. 如果 X/index.js 是一个文件，将 X/index.js 作为 JavaScript 文本载入并停止执行。
20 2. 如果 X/index.json 是一个文件，解析 X/index.json 为 JavaScript 对象并停止执行。
21 3. 如果 X/index.node 是一个文件，将 X/index.node 作为二进制插件载入并停止执行。
22
23 LOAD_AS_DIRECTORY(X)
24 1. 如果 X/package.json 是一个文件，
25    a. 解析 X/package.json，并查找 "main" 字段。
26    b. let M = X + (json main 字段)
27    c. LOAD_AS_FILE(M)
28    d. LOAD_INDEX(M)
29 2. LOAD_INDEX(X)
30
31 LOAD_NODE_MODULES(X, START)
```

```

32 1. let DIRS=NODE_MODULES_PATHS(START)
33 2. for each DIR in DIRS:
34   a. LOAD_AS_FILE(DIR/X)
35   b. LOAD_AS_DIRECTORY(DIR/X)
36
37 NODE_MODULES_PATHS(START)
38 1. let PARTS = path split(START)
39 2. let I = count of PARTS - 1
40 3. let DIRS = []
41 4. while I >= 0,
42   a. if PARTS[I] = "node_modules" CONTINUE
43   b. DIR = path join(PARTS[0 .. I] + "node_modules")
44   c. DIRS = DIRS + DIR
45   d. let I = I - 1
46 5. return DIRS

```

exports 和 module.exports 的使用

如果要对外暴露属性或方法，就用 **exports** 就行，要暴露对象(类似class，包含了很多属性和方法)，就用 **module.exports**。

10. Node.js 函数

在 JavaScript 中，一个函数可以作为另一个函数的参数。我们可以先定义一个函数，然后传递，也可以在传递参数的地方直接定义函数。

Node.js 中函数的使用与 JavaScript 类似，举例来说，你可以这样做：

```

1  function say(word) {
2    console.log(word);
3  }
4
5  function execute(someFunction, value) {
6    someFunction(value);
7  }
8
9  execute(say, "Hello");

```

以上代码中，我们把 say 函数作为 execute 函数的第一个变量进行了传递。这里传递的不是 say 的返回值，而是 say 本身！

这样一来，say 就变成了 execute 中的本地变量 someFunction，execute 可以通过调用 someFunction()（带括号的形式）来使用 say 函数。

当然，因为 say 有一个变量，execute 在调用 someFunction 时可以传递这样一个变量。

10.1 匿名函数

我们可以把一个函数作为变量传递。但是我们不一定要绕这个“先定义，再传递”的圈子，我们可以直接在另一个函数的括号中定义和传递这个函数：

```

1  function execute(someFunction, value) {
2    someFunction(value);
3  }
4
5  execute(function(word){ console.log(word) }, "Hello");

```

我们在 `execute` 接受第一个参数的地方直接定义了我们准备传递给 `execute` 的函数。

用这种方式，我们甚至不用给这个函数起名字，这也是为什么它被叫做匿名函数。

10.2 函数传递是如何让HTTP服务器工作的

带着这些知识，我们再来看看我们简约而不简单的HTTP服务器：

```
1 var http = require("http");
2
3 http.createServer(function(request, response) {
4   response.writeHead(200, {"Content-Type": "text/plain"});
5   response.write("Hello world");
6   response.end();
7 }).listen(8888);
```

现在它看上去应该清晰了很多：我们向 `createServer` 函数传递了一个匿名函数。

用这样的代码也可以达到同样的目的：

```
1 var http = require("http");
2
3 function onRequest(request, response) {
4   response.writeHead(200, {"Content-Type": "text/plain"});
5   response.write("Hello world");
6   response.end();
7 }
8
9 http.createServer(onRequest).listen(8888);
```

11. Node.js 路由

我们要为路由提供请求的 URL 和其他需要的 GET 及 POST 参数，随后路由需要根据这些数据来执行相应的代码。

因此，我们需要查看 HTTP 请求，从中提取出请求的 URL 以及 GET/POST 参数。这一功能应当属于路由还是服务器（甚至作为一个模块自身的功能）确实值得探讨，但这里暂定其为我们的HTTP服务器的功能。

我们需要的所有数据都会包含在 `request` 对象中，该对象作为 `onRequest()` 回调函数的第一个参数传递。但是为了解析这些数据，我们需要额外的 Node.js 模块，它们分别是 `url` 和 `querystring` 模块。

```
1      url.parse(string).query
2
3      url.parse(string).pathname
4      |
5      |
6      -----
7 http://localhost:8888/start?foo=bar&hello=world
8      ---      ----
9      |          |
10     |          |
11     querystring.parse(queryString)["foo"]    |
12     |
13     querystring.parse(queryString)["hello"]
```


当然我们也可以用 querystring 模块来解析 POST 请求体中的参数，稍后会有演示。

现在我们来给 onRequest() 函数加上一些逻辑，用来找出浏览器请求的 URL 路径：

```
1 // server.js 文件代码
2 var http = require("http");
3 var url = require("url");
4
5 function start() {
6   function onRequest(request, response) {
7     var pathname = url.parse(request.url).pathname;
8     console.log("Request for " + pathname + " received.");
9     response.writeHead(200, {"Content-Type": "text/plain"});
10    response.write("Hello world");
11    response.end();
12  }
13
14  http.createServer(onRequest).listen(8888);
15  console.log("Server has started.");
16 }
17
18 exports.start = start;
```

好了，我们的应用现在可以通过请求的 URL 路径来区别不同请求了--这使我们得以使用路由（还未完成）来将请求以 URL 路径为基准映射到处理程序上。

在我们所要构建的应用中，这意味着来自 /start 和 /upload 的请求可以使用不同的代码来处理。稍后我们将看到这些内容是如何整合到一起的。

现在我们可以来编写路由了，建立一个名为 **router.js** 的文件，添加以下内容：

```
1 // router.js 文件代码：
2 function route(pathname) {
3   console.log("About to route a request for " + pathname);
4 }
5
6 exports.route = route;
```

如你所见，这段代码什么也没干，不过对于现在来说这是应该的。在添加更多的逻辑以前，我们先来看看如何把路由和服务器整合起来。

我们的服务器应当知道路由的存在并加以有效利用。我们当然可以通过硬编码的方式将这一依赖项绑定到服务器上，但是其它语言的编程经验告诉我们这会是一件非常痛苦的事，因此我们将使用依赖注入的方式较松散地添加路由模块。

首先，我们来扩展一下服务器的 start() 函数，以便将路由函数作为参数传递过去，**server.js** 文件代码如下：

```
1 // server.js 文件代码：
2 var http = require("http");
3 var url = require("url");
4
5 function start(route) {
6   function onRequest(request, response) {
7     var pathname = url.parse(request.url).pathname;
8     console.log("Request for " + pathname + " received.");
9
```

```

10     route(pathname);
11
12     response.writeHead(200, {"Content-Type": "text/plain"});
13     response.write("Hello world");
14     response.end();
15 }
16
17 http.createServer(onRequest).listen(8888);
18 console.log("Server has started.");
19 }
20
21 exports.start = start;

```

同时，我们会相应扩展 index.js，使得路由函数可以被注入到服务器中：

```

1 // index.js 文件代码：
2 var server = require("./server");
3 var router = require("./router");
4
5 server.start(router.route);

```

在这里，我们传递的函数依旧什么也没做。

如果现在启动应用（node index.js，始终记得这个命令行），随后请求一个URL，你将会看到应用输出相应的信息，这表明我们的HTTP服务器已经在使用路由模块了，并将请求的路径传递给路由：

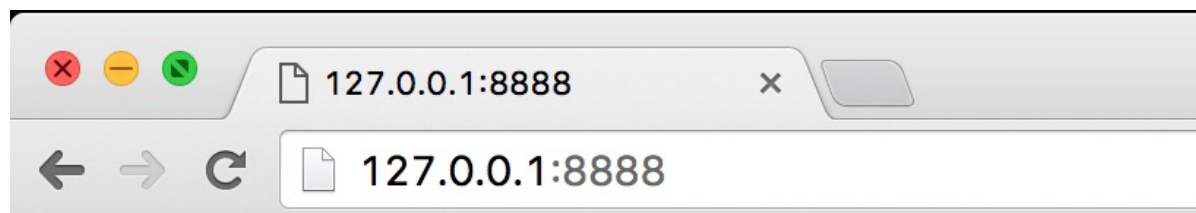
```

1 $ node index.js
2 Server has started.

```

以上输出已经去掉了比较烦人的 /favicon.ico 请求相关的部分。

浏览器访问 <http://127.0.0.1:8888/>，输出结果如下：



Hello World

根据前面的 event 绑定，直接将事件名定义成路径还是很不错的，不过要区分 request.method 是 get 还是 post，可以分两个文件处理

```

1 var http = require("http");
2 var url = require('url');
3 const { exit } = require("process");
4 var events = require('events');
5

```

```

6 // 创建 EventEmitter 对象
7 var EventEmitter = new events.EventEmitter();
8
9 // route 根路径
10 EventEmitter.on('/', function(method, response){
11     response.writeHead(200, {'Content-Type': 'text/plain'});
12     response.end('Hello World\n');
13 });
14
15 // route 404
16 EventEmitter.on('404', function(method, url, response){
17     response.writeHead(404, {'Content-Type': 'text/plain'});
18     response.end('404 Not Found\n');
19 });
20
21 // 启动服务
22 http.createServer(function (request, response) {
23     console.log(request.url);
24
25     // 分发
26     if (EventEmitter.listenerCount(request.url) > 0){
27         EventEmitter.emit(request.url, request.method, response);
28     }
29     else {
30         EventEmitter.emit('404', request.method, request.url, response);
31     }
32
33 }).listen(8888);
34
35 console.log('Server running at http://127.0.0.1:8888/');

```

12. Node.js 全局对象

JavaScript 中有一个特殊的对象，称为全局对象（Global Object），它及其所有属性都可以在程序的任何地方访问，即全局变量。

在浏览器 JavaScript 中，通常 window 是全局对象，而 Node.js 中的全局对象是 global，所有全局变量（除了 global 本身以外）都是 global 对象的属性。

在 Node.js 我们可以直接访问到 global 的属性，而不需要在应用中包含它。

12.1 全局对象与全局变量

global 最根本的作用是作为全局变量的宿主。按照 ECMAScript 的定义，满足以下条件的变量是全局变量：

- 在最外层定义的变量；
- 全局对象的属性；
- 隐式定义的变量（未定义直接赋值的变量）。

当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注意的是，在 Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的，而模块本身不是最外层上下文。

注意：最好不要使用 var 定义变量以避免引入全局变量，因为全局变量会污染命名空间，提高代码的耦合风险。

12.2 __filename

__filename 表示当前正在执行的脚本的文件名。它将输出文件所在位置的绝对路径，且和命令行参数所指定的文件名不一定相同。如果在模块中，返回的值是模块文件的路径。

12.2.1 实例

创建文件 main.js，代码如下所示：

```
1 // 输出全局变量 __filename 的值
2 console.log( __filename );
```

执行 main.js 文件，代码如下所示：

```
1 $ node main.js
2 /web/com/runoob/nodejs/main.js
```

12.3 __dirname

__dirname 表示当前执行脚本所在的目录。

12.3.1 实例

创建文件 main.js，代码如下所示：

```
1 // 输出全局变量 __dirname 的值
2 console.log( __dirname );
```

执行 main.js 文件，代码如下所示：

```
1 $ node main.js
2 /web/com/runoob/nodejs
```

12.4 setTimeout(cb, ms)

setTimeout(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。：setTimeout() 只执行一次指定函数。

返回一个代表定时器的句柄值。

12.4.1 实例

创建文件 main.js，代码如下所示：

```
1 function printHello(){
2     console.log( "Hello, world!");
3 }
4 // 两秒后执行以上函数
5 setTimeout(printHello, 2000);
```

执行 main.js 文件，代码如下所示：

```
1 $ node main.js
2 Hello, world!
```

12.5 clearTimeout(t)

clearTimeout(t) 全局函数用于停止一个之前通过 `setTimeout()` 创建的定时器。参数 **t** 是通过 `setTimeout()` 函数创建的定时器。

12.5.1 实例

创建文件 `main.js`，代码如下所示：

```
1 function printHello(){
2     console.log( "Hello, world!");
3 }
4 // 两秒后执行以上函数
5 var t = setTimeout(printHello, 2000);
6
7 // 清除定时器
8 clearTimeout(t);
```

执行 `main.js` 文件，代码如下所示：

```
1 $ node main.js
```

12.6 setInterval(cb, ms)

setInterval(cb, ms) 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。

返回一个代表定时器的句柄值。可以使用 **clearInterval(t)** 函数来清除定时器。

`setInterval()` 方法会不停地调用函数，直到 `clearInterval()` 被调用或窗口被关闭。

12.6.1 实例

创建文件 `main.js`，代码如下所示：

```
1 function printHello(){
2     console.log( "Hello, world!");
3 }
4 // 两秒后执行以上函数
5 setInterval(printHello, 2000);
```

执行 `main.js` 文件，代码如下所示：

```
1 $ node main.js
```

Hello, World! Hello, World! Hello, World! Hello, World! Hello, World!

以上程序每隔两秒就会输出一次"Hello, World!"，且会永久执行下去，直到你按下 **ctrl + c** 按钮。

12.7 console

`console` 用于提供控制台标准输出，它是由 Internet Explorer 的 JScript 引擎提供的调试工具，后来逐渐成为浏览器的实施标准。

Node.js 沿用了这个标准，提供与习惯行为一致的 `console` 对象，用于向标准输出流 (stdout) 或标准错误流 (stderr) 输出字符。

12.7.1 console 方法

以下为 console 对象的方法:

序号	方法 & 描述
1	console.log([data][, ...]) 向标准输出流打印字符并以换行符结束。该方法接收若干个参数, 如果只有一个参数, 则输出这个参数的字符串形式。如果有多个参数, 则以类似于C语言 printf() 命令的格式输出。
2	console.info([data][, ...]) 该命令的作用是返回信息性消息, 这个命令与console.log差别并不大, 除了在chrome中只会输出文字外, 其余的会显示一个蓝色的惊叹号。
3	console.error([data][, ...]) 输出错误消息的。控制台在出现错误时会显示是红色的叉子。
4	console.warn([data][, ...]) 输出警告消息。控制台出现有黄色的惊叹号。
5	console.dir(obj[, options]) 用来对一个对象进行检查 (inspect) , 并以易于阅读和打印的格式显示。
6	console.time(label) 输出时间, 表示计时开始。
7	console.timeEnd(label) 结束时间, 表示计时结束。
8	console.trace(message[, ...]) 当前执行的代码在堆栈中的调用路径, 这个测试函数运行很有帮助, 只要给想测试的函数里面加入 console.trace 就行了。
9	console.assert(value[, message][, ...]) 用于判断某个表达式或变量是否为真, 接收两个参数, 第一个参数是表达式, 第二个参数是字符串。只有当第一个参数为false, 才会输出第二个参数, 否则不会有任何结果。

console.log(): 向标准输出流打印字符并以换行符结束。

console.log 接收若干个参数, 如果只有一个参数, 则输出这个参数的字符串形式。如果有多个参数, 则以类似于C语言 printf() 命令的格式输出。

第一个参数是一个字符串, 如果没有参数, 只打印一个换行。

```
1 console.log('Hello world');
2 console.log('byvoid%diovyb');
3 console.log('byvoid%diovyb', 1991);
```

运行结果为:

```
1 Hello world
2 byvoid%diovyb
3 byvoid1991iovyb
```

- console.error(): 与console.log() 用法相同, 只是向标准错误流输出。
- console.trace(): 向标准错误流输出当前的调用栈。

```
1 console.trace();
```

运行结果为:

```
1 | Trace:
2 | at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)
3 | at Module._compile (module.js:441:26)
4 | at Object..js (module.js:459:10)
5 | at Module.load (module.js:348:31)
6 | at Function._load (module.js:308:12)
7 | at Array.0 (module.js:479:10)
8 | at EventEmitter._tickCallback (node.js:192:40)
```

12.7.2 实例

创建文件 main.js，代码如下所示：

```
1 | console.info("程序开始执行：");
2 |
3 | var counter = 10;
4 | console.log("计数：%d", counter);
5 |
6 | console.time("获取数据");
7 | //
8 | // 执行一些代码
9 | //
10 | console.timeEnd('获取数据');
11 |
12 | console.info("程序执行完毕。")
```

执行 main.js 文件，代码如下所示：

```
1 | $ node main.js
2 | 程序开始执行：
3 | 计数： 10
4 | 获取数据： 0ms
5 | 程序执行完毕
```

12.8 process

process 是一个全局变量，即 global 对象的属性。

它用于描述当前 Node.js 进程状态的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要和它打交道。下面将会介绍 process 对象的一些最常用的成员方法。

序号	事件 & 描述
1	exit 当进程准备退出时触发。
2	beforeExit 当 node 清空事件循环，并且没有其他安排时触发这个事件。通常来说，当没有进程安排时 node 退出，但是 'beforeExit' 的监听器可以异步调用，这样 node 就会继续执行。
3	uncaughtException 当一个异常冒泡回到事件循环，触发这个事件。如果给异常添加了监视器，默认的操作（打印堆栈跟踪信息并退出）就不会发生。
4	Signal 事件 当进程接收到信号时就触发。信号列表详见标准的 POSIX 信号名，如 SIGINT、SIGUSR1 等。

12.8.1 实例

创建文件 main.js，代码如下所示：

```
1 process.on('exit', function(code) {  
2  
3     // 以下代码永远不会执行  
4     setTimeout(function() {  
5         console.log("该代码不会执行");  
6     }, 0);  
7  
8     console.log('退出码为:', code);  
9 });  
10 console.log("程序执行结束");
```

执行 main.js 文件，代码如下所示：

```
1 $ node main.js  
2 程序执行结束  
3 退出码为：0
```

12.8.2 退出状态码

退出状态码如下所示：

状态码	名称 & 描述
1	Uncaught Fatal Exception 有未捕获异常，并且没有被域或 <code>uncaughtException</code> 处理函数处理。
2	Unused 保留，由 Bash 预留用于内置误用
3	Internal JavaScript Parse Error JavaScript的源码启动 Node 进程时引起解析错误。非常罕见，仅会在开发 Node 时才会有。
4	Internal JavaScript Evaluation Failure JavaScript 的源码启动 Node 进程，评估时返回函数失败。非常罕见，仅会在开发 Node 时才会有。
5	Fatal Error V8 里致命的不可恢复的错误。通常会打印到 <code>stderr</code> ，内容为：FATAL ERROR
6	Non-function Internal Exception Handler 未捕获异常，内部异常处理函数不知为何设置为 <code>on-function</code> ，并且不能被调用。
7	Internal Exception Handler Run-Time Failure 未捕获的异常，并且异常处理函数处理时自己抛出了异常。例如，如果 <code>process.on('uncaughtException')</code> 或 <code>domain.on('error')</code> 抛出了异常。
8	Unused 保留，在以前版本的 Node.js 中，退出码 8 有时表示未捕获的异常。
9	Invalid Argument 可能是给了未知的参数，或者给的参数没有值。
10	Internal JavaScript Run-Time Failure JavaScript的源码启动 Node 进程时抛出错误，非常罕见，仅会在开发 Node 时才会有。
12	Invalid Debug Argument 设置了参数 <code>--debug</code> 和/或 <code>--debug-brk</code> ，但是选择了错误端口。
128	Signal Exits 如果 Node 接收到致命信号，比如 <code>SIGKILL</code> 或 <code>SIGHUP</code> ，那么退出代码就是 128 加信号代码。这是标准的 Unix 做法，退出信号代码放在高位。

12.8.3 Process 属性

Process 提供了很多有用的属性，便于我们更好的控制系统的交互：

序号.	属性 & 描述
1	stdout 标准输出流。
2	stderr 标准错误流。
3	stdin 标准输入流。
4	argv argv 属性返回一个数组，由命令行执行脚本时的各个参数组成。它的第一个成员总是 node，第二个成员是脚本文件名，其余成员是脚本文件的参数。
5	execPath 返回执行当前脚本的 Node 二进制文件的绝对路径。
6	execArgv 返回一个数组，成员是命令行下执行脚本时，在Node可执行文件与脚本文件之间的命令行参数。
7	env 返回一个对象，成员为当前 shell 的环境变量
8	exitCode 进程退出时的代码，如果进程通过 process.exit() 退出，不需要指定退出码。
9	version Node 的版本，比如v0.10.18。
10	versions 一个属性，包含了 node 的版本和依赖。
11	config 一个包含用来编译当前 node 执行文件的 javascript 配置选项的对象。它与运行 ./configure 脚本生成的 "config.gypi" 文件相同。
12	pid 当前进程的进程号。
13	title 进程名，默认值为"node"，可以自定义该值。
14	arch 当前 CPU 的架构：'arm'、'ia32' 或者 'x64'。
15	platform 运行程序所在的平台系统 'darwin', 'freebsd', 'linux', 'sunos' 或 'win32'
16	mainModule require.main 的备选方法。不同点，如果主模块在运行时改变，require.main可能会继续返回老的模块。可以认为，这两者引用了同一个模块。

12.8.3.1 实例

创建文件 main.js，代码如下所示：

```

1 // 输出到终端
2 process.stdout.write("Hello world!" + "\n");
3
4 // 通过参数读取
5 process.argv.forEach(function(val, index, array) {
6     console.log(index + ': ' + val);
7 });
8
9 // 获取执行路径
10 console.log(process.execPath);
11
12
13 // 平台信息
14 console.log(process.platform);

```

执行 main.js 文件，代码如下所示:

```
1 $ node main.js
2 Hello world!
3 0: node
4 1: /web/www/node/main.js
5 /usr/local/node/0.10.36/bin/node
6 darwin
```

12.8.4 方法参考手册

Process 提供了很多有用的方法，便于我们更好的控制系统的交互：

序号	方法 & 描述
1	abort() 这将导致 node 触发 abort 事件。会让 node 退出并生成一个核心文件。
2	chdir(directory) 改变当前工作进程的目录，如果操作失败抛出异常。
3	cwd() 返回当前进程的工作目录
4	exit([code]) 使用指定的 code 结束进程。如果忽略，将会使用 code 0。
5	getgid() 获取进程的群组标识（参见 getgid(2)）。获取到的是群组的数字 id，而不是名字。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
6	setgid(id) 设置进程的群组标识（参见 setgid(2)）。可以接收数字 ID 或者群组名。如果指定了群组名，会阻塞等待解析为数字 ID。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
7	getuid() 获取进程的用户标识(参见 getuid(2))。这是数字的用户 id，不是用户名。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
8	setuid(id) 设置进程的用户标识（参见 setuid(2)）。接收数字 ID 或字符串名字。如果指定了群组名，会阻塞等待解析为数字 ID。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
9	getgroups() 返回进程的群组 ID 数组。POSIX 系统没有保证一定有，但是 node.js 保证有。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
10	setgroups(groups) 设置进程的群组 ID。这是授权操作，所以你需要有 root 权限，或者有 CAP_SETGID 能力。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
11	initgroups(user, extra_group) 读取 /etc/group，并初始化群组访问列表，使用成员所在的所有群组。这是授权操作，所以你需要有 root 权限，或者有 CAP_SETGID 能力。注意：这个函数仅在 POSIX 平台上可用(例如，非Windows 和 Android)。
12	kill(pid[, signal]) 发送信号给进程。pid 是进程id，并且 signal 是发送的信号字符串描述。信号名是字符串，比如 'SIGINT' 或 'SIGHUP'。如果忽略，信号会是 'SIGTERM'。
13	memoryUsage() 返回一个对象，描述了 Node 进程所用的内存状况，单位为字节。
14	nextTick(callback) 一旦当前事件循环结束，调用回调函数。
15	umask([mask]) 设置或读取进程文件的掩码。子进程从父进程继承掩码。如果 mask 参数有效，返回旧的掩码。否则，返回当前掩码。
16	uptime() 返回 Node 已经运行的秒数。
17	hrtime() 返回当前进程的高分辨时间，形式为 [seconds, nanoseconds] 数组。它是相对于过去的任意事件。该值与日期无关，因此不受时钟漂移的影响。主要用途是可以通过精确的时间间隔，来衡量程序的性能。你可以将之前的结果传递给当前的 process.hrtime()，会返回两者间的时间差，用来基准和测量时间间隔。

12.8.4.1 实例

创建文件 main.js，代码如下所示：

```
1 // 输出当前目录
2 console.log('当前目录：' + process.cwd());
3
4 // 输出当前版本
5 console.log('当前版本：' + process.version);
6
7 // 输出内存使用情况
8 console.log(process.memoryUsage());
```

执行 main.js 文件，代码如下所示：

```
1 $ node main.js
2 当前目录： /web/com/runoob/nodejs
3 当前版本： v0.10.36
4 { rss: 12541952, heapTotal: 4083456, heapUsed: 2157056 }
```

13. Node.js 常用工具

util 是一个 Node.js 核心模块，提供常用函数的集合，用于弥补核心 JavaScript 的功能过于精简的不足。

使用方法如下：

```
1 const util = require('util');
```

13.1 util.callbackify

util.callbackify(original) 将 `async` 异步函数（或者一个返回值为 `Promise` 的函数）转换成遵循异常优先的回调风格的函数，例如将 `(err, value) => ...` 回调作为最后一个参数。在回调函数中，第一个参数为拒绝的原因（如果 `Promise` 解决，则为 `null`），第二个参数则是解决的值。

13.1.1 实例

以上代码输出结果为：

```
1 hello world
```

回调函数是异步执行的，并且有异常堆栈错误追踪。如果回调函数抛出一个异常，进程会触发一个 `'uncaughtException'` 异常，如果没有被捕获，进程将会退出。

`null` 在回调函数中作为一个参数有其特殊的意义，如果回调函数的首个参数为 `Promise` 拒绝的原因且带有返回值，且值可以转换成布尔值 `false`，这个值会被封装在 `Error` 对象里，可以通过属性 `reason` 获取。

```

1 function fn() {
2   return Promise.reject(null);
3 }
4 const callbackFunction = util.callbackify(fn);
5
6 callbackFunction((err, ret) => {
7   // 当 Promise 被以 `null` 拒绝时, 它被包装为 Error 并且原始值存储在 `reason` 中。
8   err && err.hasOwnProperty('reason') && err.reason === null; // true
9 });

```

original 为 async 异步函数。该函数返回传统回调函数。

13.2 util.inherits

util.inherits(constructor, superConstructor) 是一个实现对象间原型继承的函数。

JavaScript 的面向对象特性是基于原型的, 与常见的基于类的不同。JavaScript 没有提供对象继承的语言级别特性, 而是通过原型复制来实现的。

在这里我们只介绍 util.inherits 的用法, 示例如下:

```

1 var util = require('util');
2 function Base() {
3   this.name = 'base';
4   this.base = 1991;
5   this.sayHello = function() {
6     console.log('Hello ' + this.name);
7   };
8 }
9 Base.prototype.showName = function() {
10   console.log(this.name);
11 };
12 function Sub() {
13   this.name = 'sub';
14 }
15 util.inherits(Sub, Base);
16 var objBase = new Base();
17 objBase.showName();
18 objBase.sayHello();
19 console.log(objBase);
20 var objSub = new Sub();
21 objSub.showName();
22 //objSub.sayHello();
23 console.log(objSub);

```

我们定义了一个基础对象 Base 和一个继承自 Base 的 Sub, Base 有三个在构造函数内定义的属性和一个原型中定义的函数, 通过 util.inherits 实现继承。运行结果如下:

```

1 base
2 Hello base
3 { name: 'base', base: 1991, sayHello: [Function] }
4 sub
5 { name: 'sub' }

```

注意: Sub 仅仅继承了 Base 在原型中定义的函数, 而构造函数内部创造的 base 属性和 sayHello 函数都没有被 Sub 继承。

同时，在原型中定义的属性不会被 `console.log` 作为对象的属性输出。如果我们去掉 `objSub.sayHello()`；这行的注释，将会看到：

```
1 node.js:201
2 throw e; // process.nextTick error, or 'error' event on first tick
3 ^
4 TypeError: Object #<Sub> has no method 'sayHello'
5 at Object.<anonymous> (/home/byvoid/utilinherits.js:29:8)
6 at Module._compile (module.js:441:26)
7 at Object..js (module.js:459:10)
8 at Module.load (module.js:348:31)
9 at Function._load (module.js:308:12)
10 at Array.0 (module.js:479:10)
11 at EventEmitter._tickCallback (node.js:192:40)
```

13.3 util.inspect

`util.inspect(object,[showHidden],[depth],[colors])` 是一个将任意对象转换为字符串的方法，通常用于调试和错误输出。它至少接受一个参数 `object`，即要转换的对象。

`showHidden` 是一个可选参数，如果值为 `true`，将会输出更多隐藏信息。

`depth` 表示最大递归的层数，如果对象很复杂，你可以指定层数以控制输出信息的多少。如果不指定 `depth`，默认会递归 2 层，指定为 `null` 表示将不限递归层数完整遍历对象。如果 `colors` 值为 `true`，输出格式将会以 ANSI 颜色编码，通常用于在终端显示更漂亮的效果。

特别要指出的是，`util.inspect` 并不会简单地直接把对象转换为字符串，即使该对象定义了 `toString` 方法也不会调用。

```
1 var util = require('util');
2 function Person() {
3     this.name = 'byvoid';
4     this.toString = function() {
5         return this.name;
6     };
7 }
8 var obj = new Person();
9 console.log(util.inspect(obj));
10 console.log(util.inspect(obj, true));
```

运行结果是：

```
1 Person { name: 'byvoid', toString: [Function] }
2 Person {
3   name: 'byvoid',
4   toString:
5     { [Function]
6       [length]: 0,
7       [name]: '',
8       [arguments]: null,
9       [caller]: null,
10      [prototype]: { [constructor]: [Circular] } } }
```

13.4 util.isArray(object)

如果给定的参数 "object" 是一个数组返回 true, 否则返回 false。

```
1 var util = require('util');
2
3 util.isArray([])
4 // true
5 util.isArray(new Array)
6 // true
7 util.isArray({})
8 // false
```

13.5 util.isRegExp(object)

如果给定的参数 "object" 是一个正则表达式返回true, 否则返回false。

```
1 var util = require('util');
2
3 util.isRegExp(/some regexp/)
4 // true
5 util.isRegExp(new RegExp('another regexp'))
6 // true
7 util.isRegExp({})
8 // false
```

13.6 util.isDate(object)

如果给定的参数 "object" 是一个日期返回true, 否则返回false。

```
1 var util = require('util');
2
3 util.isDate(new Date())
4 // true
5 util.isDate(Date())
6 // false (without 'new' returns a String)
7 util.isDate({})
8 // false
```

更多详情可以访问 <http://nodejs.org/api/util.html> 了解详细内容。

14. Node.js 文件系统

Node.js 提供一组类似 UNIX (POSIX) 标准的文件操作API。Node 导入文件系统模块(fs)语法如下所示:

```
1 var fs = require("fs")
```


14.1 异步和同步

Node.js 文件系统（fs 模块）模块中的方法均有异步和同步版本，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)。

建议大家使用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞。

14.1.1 实例

创建 input.txt 文件，内容如下：

```
1 菜鸟教程官网地址：www.runoob.com
2 文件读取实例
```

创建 file.js 文件，代码如下：

```
1  var fs = require("fs");
2
3  // 异步读取
4  fs.readFile('input.txt', function (err, data) {
5      if (err) {
6          return console.error(err);
7      }
8      console.log("异步读取: " + data.toString());
9  });
10
11 // 同步读取
12 var data = fs.readFileSync('input.txt');
13 console.log("同步读取: " + data.toString());
14
15 console.log("程序执行完毕。");
```

以上代码执行结果如下：

```
1 $ node file.js
2 同步读取： 菜鸟教程官网地址：www.runoob.com
3 文件读取实例
4
5 程序执行完毕。
6 异步读取： 菜鸟教程官网地址：www.runoob.com
7 文件读取实例
```

接下来，让我们来具体了解下 Node.js 文件系统的方法。

14.2 打开文件

14.2.1 语法

以下为在异步模式下打开文件的语法格式：

```
1 fs.open(path, flags[, mode], callback)
```

14.2.2 参数

参数使用说明如下：

- **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)。

flags 参数可以是以下值：

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。

14.2.3 实例

接下来我们创建 file.js 文件，并打开 input.txt 文件进行读写，代码如下所示：

```
1  var fs = require("fs");
2
3  // 异步打开文件
4  console.log("准备打开文件！");
5  fs.open('input.txt', 'r+', function(err, fd) {
6      if (err) {
7          return console.error(err);
8      }
9      console.log("文件打开成功！");
10 })
```

以上代码执行结果如下：

```
1  $ node file.js
2  准备打开文件！
3  文件打开成功！
```

14.3 获取文件信息

14.3.1 语法

以下为通过异步模式获取文件信息的语法格式：

```
1 fs.stat(path, callback)
```

14.3.2 参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，带有两个参数如：(err, stats), **stats** 是 fs.Stats 对象。

fs.stat(path)执行后，会将stats类的实例返回给其回调函数。可以通过stats类中的提供方法判断文件的相关属性。例如判断是否为文件：

```
1 var fs = require('fs');
2
3 fs.stat('/Users/liuht/code/itbilu/demo/fs.js', function (err, stats) {
4     console.log(stats.isFile());           //true
5 })
```

stats类中的方法有：

方法	描述
stats.isFile()	如果是文件返回 true，否则返回 false。
stats.isDirectory()	如果是目录返回 true，否则返回 false。
stats.isBlockDevice()	如果是块设备返回 true，否则返回 false。
stats.isCharacterDevice()	如果是字符设备返回 true，否则返回 false。
stats.isSymbolicLink()	如果是软链接返回 true，否则返回 false。
stats.isFIFO()	如果是FIFO，返回true，否则返回 false。FIFO是UNIX中的一种特殊类型的命令管道。
stats.isSocket()	如果是 Socket 返回 true，否则返回 false。

14.3.3 实例

接下来我们创建 file.js 文件，代码如下所示：

```
1 var fs = require("fs");
2
3 console.log("准备打开文件！");
4 fs.stat('input.txt', function (err, stats) {
5     if (err) {
6         return console.error(err);
7     }
8     console.log(stats);
9     console.log("读取文件信息成功！");
10 })
```

```

11 // 检测文件类型
12 console.log("是否为文件(isFile) ? " + stats.isFile());
13 console.log("是否为目录(isDirectory) ? " + stats.isDirectory());
14 });

```

以上代码执行结果如下：

```

1 $ node file.js
2 准备打开文件！
3 { dev: 16777220,
4   mode: 33188,
5   nlink: 1,
6   uid: 501,
7   gid: 20,
8   rdev: 0,
9   blksize: 4096,
10  ino: 40333161,
11  size: 61,
12  blocks: 8,
13  atime: Mon Sep 07 2015 17:43:55 GMT+0800 (CST),
14  mtime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST),
15  ctime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST) }
16 读取文件信息成功！
17 是否为文件(isFile) ? true
18 是否为目录(isDirectory) ? false

```

14.4 写入文件

14.4.1 语法

以下为异步模式下写入文件的语法格式：

```

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

```

writeFile 直接打开文件默认是 **w** 模式，所以如果文件存在，该方法写入的内容会覆盖旧的文件内容。

14.4.2 参数

参数使用说明如下：

- **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(缓冲) 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'。
- **callback** - 回调函数，回调函数只包含错误信息参数(err)，在写入失败时返回。

14.4.3 实例

接下来我们创建 file.js 文件，代码如下所示：

```

1 var fs = require("fs");
2
3 console.log("准备写入文件");
4 fs.writeFile('input.txt', '我是通过fs.writeFile 写入文件的内容', function(err)
5 {
6     if (err) {

```

```

6         return console.error(err);
7     }
8     console.log("数据写入成功! ");
9     console.log("-----我是分割线-----");
10    console.log("读取写入的数据! ");
11    fs.readFile('input.txt', function (err, data) {
12        if (err) {
13            return console.error(err);
14        }
15        console.log("异步读取文件数据: " + data.toString());
16    });
17 });

```

以上代码执行结果如下:

```

1  $ node file.js
2  准备写入文件
3  数据写入成功!
4  -----我是分割线-----
5  读取写入的数据!
6  异步读取文件数据: 我是通过fs.writeFile 写入文件的内容

```

14.5 读取文件

14.5.1 语法

以下为异步模式下读取文件的语法格式:

```

1  fs.read(fd, buffer, offset, length, position, callback)

```

该方法使用了文件描述符来读取文件。

14.5.2 参数

参数使用说明如下:

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **buffer** - 数据写入的缓冲区。
- **offset** - 缓冲区写入的写入偏移量。
- **length** - 要从文件中读取的字节数。
- **position** - 文件读取的起始位置, 如果 position 的值为 null, 则会从当前文件指针的位置读取。
- **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。

14.5.3 实例

input.txt 文件内容为:

```

1  菜鸟教程官网地址: www.runoob.com

```

接下来我们创建 file.js 文件, 代码如下所示:

```

1  var fs = require("fs");
2  var buf = new Buffer.alloc(1024);
3

```

```

4 console.log("准备打开已存在的文件!");
5 fs.open('input.txt', 'r+', function(err, fd) {
6     if (err) {
7         return console.error(err);
8     }
9     console.log("文件打开成功!");
10    console.log("准备读取文件:");
11    fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
12        if (err){
13            console.log(err);
14        }
15        console.log(bytes + " 字节被读取");
16
17        // 仅输出读取的字节
18        if(bytes > 0){
19            console.log(buf.slice(0, bytes).toString());
20        }
21    });
22 });

```

以上代码执行结果如下:

```

1 $ node file.js
2 准备打开已存在的文件!
3 文件打开成功!
4 准备读取文件:
5 42 字节被读取
6 菜鸟教程官网地址: www.runoob.com

```

14.6 关闭文件

14.6.1 语法

以下为异步模式下关闭文件的语法格式:

```
1 fs.close(fd, callback)
```

该方法使用了文件描述符来读取文件。

14.6.2 参数

参数使用说明如下:

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **callback** - 回调函数, 没有参数。

14.6.3 实例

input.txt 文件内容为:

```
1 菜鸟教程官网地址: www.runoob.com
```

接下来我们创建 file.js 文件, 代码如下所示:

```
1 var fs = require("fs");
```

```

2  var buf = new Buffer.alloc(1024);
3
4  console.log("准备打开文件!");
5  fs.open('input.txt', 'r+', function(err, fd) {
6      if (err) {
7          return console.error(err);
8      }
9      console.log("文件打开成功!");
10     console.log("准备读取文件!");
11     fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
12         if (err){
13             console.log(err);
14         }
15
16         // 仅输出读取的字节
17         if(bytes > 0){
18             console.log(buf.slice(0, bytes).toString());
19         }
20
21         // 关闭文件
22         fs.close(fd, function(err){
23             if (err){
24                 console.log(err);
25             }
26             console.log("文件关闭成功");
27         });
28     });
29 });

```

以上代码执行结果如下:

```

1  $ node file.js
2  准备打开文件!
3  文件打开成功!
4  准备读取文件!
5  菜鸟教程官网地址: www.runoob.com
6  文件关闭成功

```

14.7 截取文件

14.7.1 语法

以下为异步模式下截取文件的语法格式:

```

1  fs.ftruncate(fd, len, callback)

```

该方法使用了文件描述符来读取文件。

14.7.2 参数

参数使用说明如下:

- **fd** - 通过 `fs.open()` 方法返回的文件描述符。
- **len** - 文件内容截取的长度。
- **callback** - 回调函数, 没有参数。

14.7.3 实例

input.txt 文件内容为：

```
1 | site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
1  var fs = require("fs");
2  var buf = new Buffer.alloc(1024);
3
4  console.log("准备打开文件！");
5  fs.open('input.txt', 'r+', function(err, fd) {
6      if (err) {
7          return console.error(err);
8      }
9      console.log("文件打开成功！");
10     console.log("截取10字节内的文件内容，超出部分将被去除。");
11
12     // 截取文件
13     fs.ftruncate(fd, 10, function(err){
14         if (err){
15             console.log(err);
16         }
17         console.log("文件截取成功。");
18         console.log("读取相同的文件");
19         fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
20             if (err){
21                 console.log(err);
22             }
23
24             // 仅输出读取的字节
25             if(bytes > 0){
26                 console.log(buf.slice(0, bytes).toString());
27             }
28
29             // 关闭文件
30             fs.close(fd, function(err){
31                 if (err){
32                     console.log(err);
33                 }
34                 console.log("文件关闭成功！");
35             });
36         });
37     });
38 });
```

以上代码执行结果如下：


```
1 | $ node file.js
2 | 准备打开文件！
3 | 文件打开成功！
4 | 截取10字节内的文件内容，超出部分将被去除。
5 | 文件截取成功。
6 | 读取相同的文件
7 | site:www.r
8 | 文件关闭成功
```

14.8 删除文件

14.8.1 语法

以下为删除文件的语法格式：

```
1 | fs.unlink(path, callback)
```

14.8.2 参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

14.8.3 实例

input.txt 文件内容为：

```
1 | site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
1 | var fs = require("fs");
2 |
3 | console.log("准备删除文件！");
4 | fs.unlink('input.txt', function(err) {
5 |     if (err) {
6 |         return console.error(err);
7 |     }
8 |     console.log("文件删除成功！");
9 | });
```

以上代码执行结果如下：

```
1 | $ node file.js
2 | 准备删除文件！
3 | 文件删除成功！
```

再去查看 input.txt 文件，发现已经不存在了。

14.9 创建目录

14.9.1 语法

以下为创建目录的语法格式：

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

14.9.2 参数

参数使用说明如下：

- **path** - 文件路径。
- **options** 参数可以是：
 - **recursive** - 是否以递归的方式创建目录，默认为 false。
 - **mode** - 设置目录权限，默认为 0777。
- **callback** - 回调函数，没有参数。

14.9.3 实例

接下来我们创建 file.js 文件，代码如下所示：

```
1 var fs = require("fs");
2 // tmp 目录必须存在
3 console.log("创建目录 /tmp/test/");
4 fs.mkdir("/tmp/test/", function(err){
5     if (err) {
6         return console.error(err);
7     }
8     console.log("目录创建成功。");
9 });
```

以上代码执行结果如下：

```
1 $ node file.js
2 创建目录 /tmp/test/
3 目录创建成功。
```

可以添加 recursive: true 参数，不管创建的目录 /tmp 和 /tmp/a 是否存在：

```
1 fs.mkdir('/tmp/a/apple', { recursive: true }, (err) => {
2     if (err) throw err;
3 });
```

14.10 读取目录

14.10.1 语法

以下为读取目录的语法格式：

```
1 fs.readdir(path, callback)
```

14.10.2 参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，回调函数带有两个参数err, files, err 为错误信息，files 为 目录下的文件数组列表。

14.10.3 实例

接下来我们创建 file.js 文件，代码如下所示：

```
1 var fs = require("fs");
2
3 console.log("查看 /tmp 目录");
4 fs.readdir("/tmp/",function(err, files){
5     if (err) {
6         return console.error(err);
7     }
8     files.forEach( function (file){
9         console.log( file );
10    });
11 });
```

以上代码执行结果如下：

```
1 $ node file.js
2 查看 /tmp 目录
3 input.out
4 output.out
5 test
6 test.txt
```

14.11 删除目录

14.11.1 语法

以下为删除目录的语法格式：

```
1 fs.rmdir(path, callback)
```

14.11.2 参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

14.11.3 实例

接下来我们创建 file.js 文件，代码如下所示：

```
1 var fs = require("fs");
2 // 执行前创建一个空的 /tmp/test 目录
3 console.log("准备删除目录 /tmp/test");
4 fs.rmdir("/tmp/test",function(err){
```

```
5     if (err) {
6         return console.error(err);
7     }
8     console.log("读取 /tmp 目录");
9     fs.readdir("/tmp/",function(err, files){
10         if (err) {
11             return console.error(err);
12         }
13         files.forEach( function (file){
14             console.log( file );
15         });
16     });
17 });
```

以上代码执行结果如下：

```
1 $ node file.js
2 准备删除目录 /tmp/test
3 读取 /tmp 目录
4 .....
```

14.12 文件模块方法参考手册

以下为 Node.js 文件模块相同的方法列表：

序号	方法 & 描述
1	fs.rename(oldPath, newPath, callback) 异步 rename().回调函数没有参数, 但可能抛出异常。
2	fs.ftruncate(fd, len, callback) 异步 ftruncate().回调函数没有参数, 但可能抛出异常。
3	fs.ftruncateSync(fd, len) 同步 ftruncate()
4	fs.truncate(path, len, callback) 异步 truncate().回调函数没有参数, 但可能抛出异常。
5	fs.truncateSync(path, len) 同步 truncate()
6	fs.chown(path, uid, gid, callback) 异步 chown().回调函数没有参数, 但可能抛出异常。
7	fs.chownSync(path, uid, gid) 同步 chown()
8	fs.fchown(fd, uid, gid, callback) 异步 fchown().回调函数没有参数, 但可能抛出异常。
9	fs.fchownSync(fd, uid, gid) 同步 fchown()
10	fs.lchown(path, uid, gid, callback) 异步 lchown().回调函数没有参数, 但可能抛出异常。
11	fs.lchownSync(path, uid, gid) 同步 lchown()
12	fs.chmod(path, mode, callback) 异步 chmod().回调函数没有参数, 但可能抛出异常。
13	fs.chmodSync(path, mode) 同步 chmod()。
14	fs.fchmod(fd, mode, callback) 异步 fchmod().回调函数没有参数, 但可能抛出异常。
15	fs.fchmodSync(fd, mode) 同步 fchmod()。
16	fs.lchmod(path, mode, callback) 异步 lchmod().回调函数没有参数, 但可能抛出异常。 Only available on Mac OS X.
17	fs.lchmodSync(path, mode) 同步 lchmod()。
18	fs.stat(path, callback) 异步 stat(). 回调函数有两个参数 err, stats, stats 是 fs.Stats 对象。
19	fs.lstat(path, callback) 异步 lstat(). 回调函数有两个参数 err, stats, stats 是 fs.Stats 对象。
20	fs.fstat(fd, callback) 异步 fstat(). 回调函数有两个参数 err, stats, stats 是 fs.Stats 对象。
21	fs.statSync(path) 同步 stat(). 返回 fs.Stats 的实例。
22	fs.lstatSync(path) 同步 lstat(). 返回 fs.Stats 的实例。
23	fs.fstatSync(fd) 同步 fstat(). 返回 fs.Stats 的实例。
24	fs.link(srcpath, dstpath, callback) 异步 link().回调函数没有参数, 但可能抛出异常。
25	fs.linkSync(srcpath, dstpath) 同步 link()。

序号	方法 & 描述
26	fs.symlink(srcpath, dstpath[, type], callback) 异步 symlink().回调函数没有参数, 但可能抛出异常。type 参数可以设置为 'dir', 'file', 或 'junction' (默认为 'file')。
27	fs.symlinkSync(srcpath, dstpath[, type]) 同步 symlink()。
28	fs.readlink(path, callback) 异步 readlink(). 回调函数有两个参数 err, linkString。
29	fs.realpath(path[, cache], callback) 异步 realpath(). 回调函数有两个参数 err, resolvedPath。
30	fs.realpathSync(path[, cache]) 同步 realpath()。返回绝对路径。
31	fs.unlink(path, callback) 异步 unlink().回调函数没有参数, 但可能抛出异常。
32	fs.unlinkSync(path) 同步 unlink()。
33	fs.rmdir(path, callback) 异步 rmdir().回调函数没有参数, 但可能抛出异常。
34	fs.rmdirSync(path) 同步 rmdir()。
35	fs.mkdir(path[, mode], callback) 异步 mkdir(2).回调函数没有参数, 但可能抛出异常。访问权限默认为 0777。
36	fs.mkdirSync(path[, mode]) 同步 mkdir()。
37	fs.readdir(path, callback) 异步 readdir(3). 读取目录的内容。
38	fs.readdirSync(path) 同步 readdir().返回文件数组列表。
39	fs.close(fd, callback) 异步 close().回调函数没有参数, 但可能抛出异常。
40	fs.closeSync(fd) 同步 close()。
41	fs.open(path, flags[, mode], callback) 异步打开文件。
42	fs.openSync(path, flags[, mode]) 同步 version of fs.open()。
43	fs.utimes(path, atime, mtime, callback)
44	fs.utimesSync(path, atime, mtime) 修改文件时间戳, 文件通过指定的文件路径。
45	fs.futimes(fd, atime, mtime, callback)
46	fs.futimesSync(fd, atime, mtime) 修改文件时间戳, 通过文件描述符指定。
47	fs.fsync(fd, callback) 异步 fsync.回调函数没有参数, 但可能抛出异常。
48	fs.fsyncSync(fd) 同步 fsync。
49	fs.write(fd, buffer, offset, length[, position], callback) 将缓冲区内容写入到通过文件描述符指定的文件。
50	fs.write(fd, data[, position[, encoding]], callback) 通过文件描述符 fd 写入文件内容。
51	fs.writeSync(fd, buffer, offset, length[, position]) 同步版的 fs.write()。
52	fs.writeSync(fd, data[, position[, encoding]]) 同步版的 fs.write()。

序号	方法 & 描述
53	fs.read(fd, buffer, offset, length, position, callback) 通过文件描述符 fd 读取文件内容。
54	fs.readSync(fd, buffer, offset, length, position) 同步版的 fs.read.
55	fs.readFile(filename[, options], callback) 异步读取文件内容。
56	fs.readFileSync(filename[, options])
57	fs.writeFile(filename, data[, options], callback) 异步写入文件内容。
58	fs.writeFileSync(filename, data[, options]) 同步版的 fs.writeFile。
59	fs.appendFile(filename, data[, options], callback) 异步追加文件内容。
60	fs.appendFileSync(filename, data[, options]) The 同步 version of fs.appendFile.
61	fs.watchFile(filename[, options], listener) 查看文件的修改。
62	fs.unwatchFile(filename[, listener]) 停止查看 filename 的修改。
63	fs.watch(filename[, options][, listener]) 查看 filename 的修改，filename 可以是文件或目录。返回 fs.FSWatcher 对象。
64	fs.exists(path, callback) 检测给定的路径是否存在。
65	fs.existsSync(path) 同步版的 fs.exists.
66	fs.access(path[, mode], callback) 测试指定路径用户权限。
67	fs.accessSync(path[, mode]) 同步版的 fs.access。
68	fs.createReadStream(path[, options]) 返回ReadStream 对象。
69	fs.createWriteStream(path[, options]) 返回 WriteStream 对象。
70	fs.symlink(srcpath, dstpath[, type], callback) 异步 symlink().回调函数没有参数，但可能抛出异常。

更多内容，请查看官网文件模块描述：[File System](#)。

15. Node.js GET/POST请求

在很多场景中，我们的服务器都需要跟用户的浏览器打交道，如表单提交。

表单提交到服务器一般都使用 GET/POST 请求。

15.1 获取GET请求内容

由于GET请求直接被嵌入在路径中，URL是完整的请求路径，包括了?后面的部分，因此你可以手动解析后面的内容作为GET请求的参数。

node.js 中 url 模块中的 parse 函数提供了这个功能。

15.1.1 实例

```
1 var http = require('http');
2 var url = require('url');
3 var util = require('util');
4
5 http.createServer(function(req, res){
6     res.writeHead(200, {'Content-Type': 'text/plain; charset=utf-8'});
7     res.end(util.inspect(url.parse(req.url, true)));
8 }).listen(3000);
```

在浏览器中访问 <http://localhost:3000/user?name=菜鸟教程&url=www.runoob.com> 然后查看返回结果:



```
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?name=%E8%8F%9C%E9%B8%9F%E6%95%99%E7%A8%8B&url=www.runoob.com',
  query: { name: '菜鸟教程', url: 'www.runoob.com' },
  pathname: '/user',
  path: '/user?name=%E8%8F%9C%E9%B8%9F%E6%95%99%E7%A8%8B&url=www.runoob.com',
  href: '/user?name=%E8%8F%9C%E9%B8%9F%E6%95%99%E7%A8%8B&url=www.runoob.com' }
```

15.1.2 获取 URL 的参数

我们可以使用 `url.parse` 方法来解析 URL 中的参数, 代码如下:

15.1.2.1 实例

```
1 var http = require('http');
2 var url = require('url');
3 var util = require('util');
4
5 http.createServer(function(req, res){
6     res.writeHead(200, {'Content-Type': 'text/plain'});
7
8     // 解析 url 参数
9     var params = url.parse(req.url, true).query;
10    res.write("网站名: " + params.name);
11    res.write("\n");
12    res.write("网站 URL: " + params.url);
13    res.end();
14
15 }).listen(3000);
```

在浏览器中访问 <http://localhost:3000/user?name=菜鸟教程&url=www.runoob.com> 然后查看返回结果:



15.2 获取 POST 请求内容

POST 请求的内容全部的都在请求体中，`http.ServerRequest` 并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作。

比如上传文件，而很多时候我们可能并不需要理会请求体的内容，恶意的POST请求会大大消耗服务器的资源，所以 `node.js` 默认是不会解析请求体的，当你需要的时候，需要手动来做。

15.2.1 基本语法结构说明

```
1 var http = require('http');
2 var querystring = require('querystring');
3 var util = require('util');
4
5 http.createServer(function(req, res){
6     // 定义了一个post变量，用于暂存请求体的信息
7     var post = '';
8
9     // 通过req的data事件监听函数，每当接受到请求体的数据，就累加到post变量中
10    req.on('data', function(chunk){
11        post += chunk;
12    });
13
14    // 在end事件触发后，通过querystring.parse将post解析为真正的POST请求格式，然后向客户端返回。
15    req.on('end', function(){
16        post = querystring.parse(post);
17        res.end(util.inspect(post));
18    });
19 }).listen(3000);
```

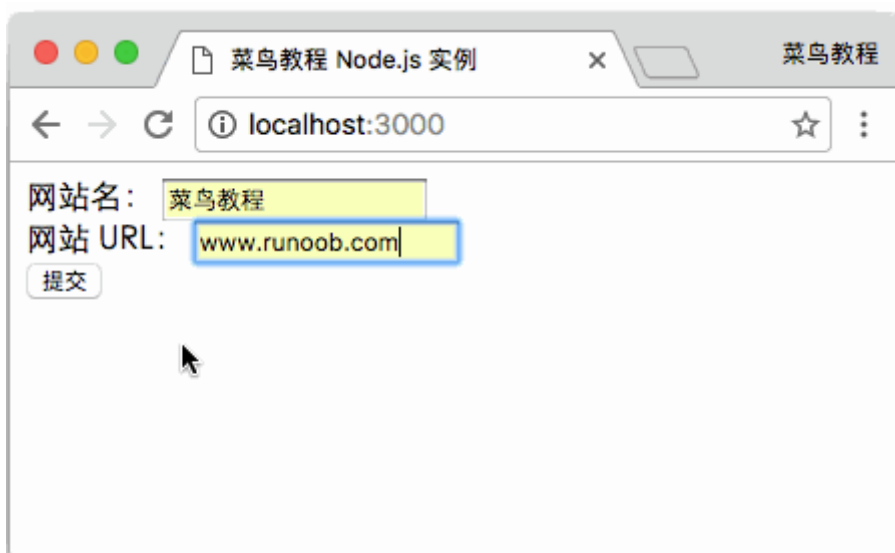
以下实例表单通过 POST 提交并输出数据：

```
1 var http = require('http');
2 var querystring = require('querystring');
3
4 var postHTML =
5     '<html><head><meta charset="utf-8"><title>菜鸟教程 Node.js 实例</title>' +
6     '</head>' +
7     '<body>' +
8     '<form method="post">' +
9     '网站名: <input name="name"><br>' +
10    '网站 URL: <input name="url"><br>' +
11    '<input type="submit">' +
12    '</form>' +
13    '</body></html>';
14
15 http.createServer(function (req, res) {
16     var body = "";
```

```

16 req.on('data', function (chunk) {
17     body += chunk;
18 });
19 req.on('end', function () {
20     // 解析参数
21     body = querystring.parse(body);
22     // 设置响应头部信息及编码
23     res.writeHead(200, {'Content-Type': 'text/html; charset=utf8'});
24
25     if(body.name && body.url) { // 输出提交的数据
26         res.write("网站名: " + body.name);
27         res.write("<br>");
28         res.write("网站 URL: " + body.url);
29     } else { // 输出表单
30         res.write(postHTML);
31     }
32     res.end();
33 });
34 }).listen(3000);

```



16. Node.js 工具模块

在 Node.js 模块库中有很多好用的模块。接下来我们为大家介绍几种常用模块的使用：

序号	模块名 & 描述
1	OS 模块 提供基本的系统操作函数。
2	Path 模块 提供了处理和转换文件路径的工具。
3	Net 模块 用于底层的网络通信。提供了服务端和客户端的操作。
4	DNS 模块 用于解析域名。
5	Domain 模块 简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的。

17. Node.js Web 模块

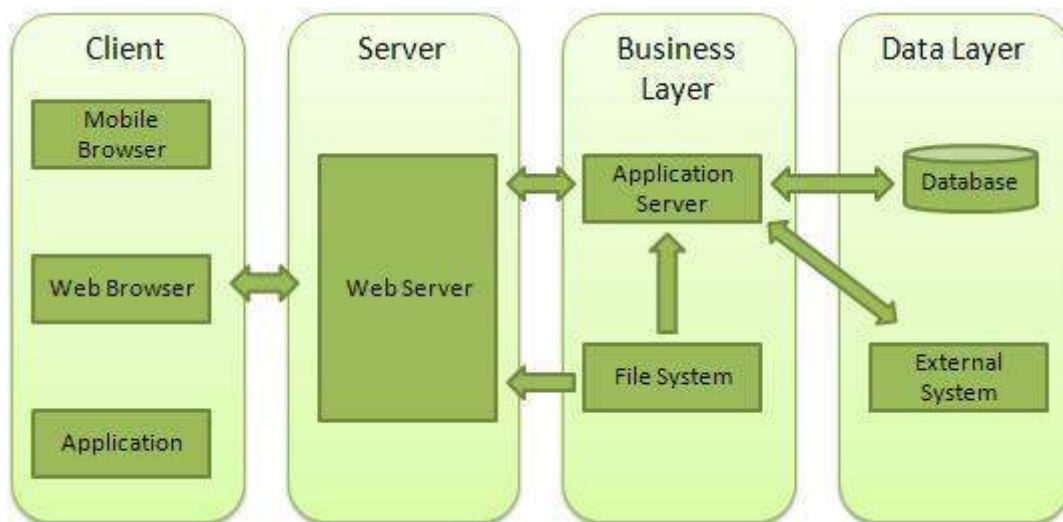
17.1 什么是 Web 服务器?

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，Web服务器的基本功能就是提供Web信息浏览服务。它只需支持HTTP协议、HTML文档格式及URL，与客户端的网络浏览器配合。

大多数 web 服务器都支持服务端的脚本语言（php、python、ruby）等，并通过脚本语言从数据库获取数据，将结果返回给客户端浏览器。

目前最主流的三个Web服务器是Apache、Nginx、IIS。

17.2 Web 应用架构



- **Client** - 客户端，一般指浏览器，浏览器可以通过 HTTP 协议向服务器请求数据。
- **Server** - 服务端，一般指 Web 服务器，可以接收客户端请求，并向客户端发送响应数据。
- **Business** - 业务层，通过 Web 服务器处理应用程序，如与数据库交互，逻辑运算，调用外部程序等。
- **Data** - 数据层，一般由数据库组成。

17.3 使用 Node 创建 Web 服务器

Node.js 提供了 http 模块，http 模块主要用于搭建 HTTP 服务端和客户端，使用 HTTP 服务器或客户端功能必须调用 http 模块，代码如下：

```
1 var http = require('http');
```

以下是演示一个最基本的 HTTP 服务器架构(使用 8080 端口)，创建 server.js 文件，代码如下所示：

```
1 var http = require('http');
2 var fs = require('fs');
3 var url = require('url');
4
5
6 // 创建服务器
7 http.createServer( function (request, response) {
8     // 解析请求，包括文件名
9     var pathname = url.parse(request.url).pathname;
10
11     // 输出请求的文件名
12     console.log("Request for " + pathname + " received.");
13 }
```

```

14 // 从文件系统中读取请求的文件内容
15 fs.readFile(pathname.substr(1), function (err, data) {
16     if (err) {
17         console.log(err);
18         // HTTP 状态码: 404 : NOT FOUND
19         // Content Type: text/html
20         response.writeHead(404, {'Content-Type': 'text/html'});
21     }else{
22         // HTTP 状态码: 200 : OK
23         // Content Type: text/html
24         response.writeHead(200, {'Content-Type': 'text/html'});
25
26         // 响应文件内容
27         response.write(data.toString());
28     }
29     // 发送响应数据
30     response.end();
31 });
32 }).listen(8080);
33
34 // 控制台会输出以下信息
35 console.log('Server running at http://127.0.0.1:8080/');

```

接下来我们在该目录下创建一个 index.html 文件，代码如下：

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>菜鸟教程(runoob.com)</title>
6 </head>
7 <body>
8     <h1>我的第一个标题</h1>
9     <p>我的第一个段落。</p>
10 </body>
11 </html>

```

执行 server.js 文件：

```

1 $ node server.js
2 Server running at http://127.0.0.1:8080/

```



执行 server.js 的控制台输出信息如下：

```
1 | Server running at http://127.0.0.1:8080/
2 | Request for /index.html received.      # 客户端请求信息
```

17.4 使用 Node 创建 Web 客户端

Node 创建 Web 客户端需要引入 http 模块，创建 client.js 文件，代码如下所示：

```
1 | var http = require('http');
2 |
3 | // 用于请求的选项
4 | var options = {
5 |     host: 'localhost',
6 |     port: '8080',
7 |     path: '/index.html'
8 | };
9 |
10 | // 处理响应的回调函数
11 | var callback = function(response){
12 |     // 不断更新数据
13 |     var body = '';
14 |     response.on('data', function(data) {
15 |         body += data;
16 |     });
17 |
18 |     response.on('end', function() {
19 |         // 数据接收完成
20 |         console.log(body);
21 |     });
22 | }
23 | // 向服务端发送请求
24 | var req = http.request(options, callback);
25 | req.end();
```

新开一个终端，执行 client.js 文件，输出结果如下：

```
1 | $ node client.js
2 | <!DOCTYPE html>
3 | <html>
4 | <head>
5 | <meta charset="utf-8">
6 | <title>菜鸟教程(runoob.com)</title>
7 | </head>
8 | <body>
9 |     <h1>我的第一个标题</h1>
10 |     <p>我的第一个段落。</p>
11 | </body>
12 | </html>
```

执行 server.js 的控制台输出信息如下：

```
1 | Server running at http://127.0.0.1:8080/
2 | Request for /index.html received.      # 客户端请求信息
```

18. Node.js Express 框架

18.1 Express 简介

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。

使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架核心特性:

- 可以设置中间件来响应 HTTP 请求。
- 定义了路由表用于执行不同的 HTTP 请求动作。
- 可以通过向模板传递参数来动态渲染 HTML 页面。

18.2 安装 Express

安装 Express 并将其保存到依赖列表中:

```
1 $ cnpm install express --save
```

以上命令会将 Express 框架安装在当前目录的 **node_modules** 目录中, **node_modules** 目录下会自动创建 express 目录。以下几个重要的模块是需要与 express 框架一起安装的:

- **body-parser** - node.js 中间件, 用于处理 JSON, Raw, Text 和 URL 编码的数据。
- **cookie-parser** - 这就是一个解析Cookie的工具。通过req.cookies可以取到传过来的cookie, 并把它们转成对象。
- **multer** - node.js 中间件, 用于处理 enctype="multipart/form-data" (设置表单的MIME编码) 的表单数据。

```
1 $ cnpm install body-parser --save
2 $ cnpm install cookie-parser --save
3 $ cnpm install multer --save
```

安装完后, 我们可以查看下 express 使用的版本号:

```
1 $ cnpm list express
2 /data/www/node
3 └─ express@4.15.2 -> /Users/tianqixin/www/node/node_modules/.4.15.2@express
```

18.3 第一个 Express 框架实例

接下来我们使用 Express 框架来输出 "Hello World".

以下实例中我们引入了 express 模块, 并在客户端发起请求后, 响应 "Hello World" 字符串。

创建 express_demo.js 文件, 代码如下所示:

```
1 // express_demo.js 文件代码:
2 var express = require('express');
3 var app = express();
4
5 app.get('/', function (req, res) {
6   res.send('Hello world');
7 })
8
9 var server = app.listen(8081, function () {
10
```

```

11     var host = server.address().address
12     var port = server.address().port
13
14     console.log("应用实例，访问地址为 http://%s:%s", host, port)
15
16 })

```

执行以上代码：

```

1 $ node express_demo.js
2 应用实例，访问地址为 http://0.0.0.0:8081

```

在浏览器中访问 <http://127.0.0.1:8081>，结果如下图所示：



18.4 请求和响应

Express 应用使用回调函数的参数：**request** 和 **response** 对象来处理请求和响应的数据。

```

1 app.get('/', function (req, res) {
2     // --
3 })

```

request 和 **response** 对象的具体介绍：

Request 对象 - request 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。常见属性有：

1. req.app：当callback为外部文件时，用req.app访问express的实例
2. req.baseUrl：获取路由当前安装的URL路径
3. req.body / req.cookies：获得「请求主体」/ Cookies
4. req.fresh / req.stale：判断请求是否还「新鲜」
5. req.hostname / req.ip：获取主机名和IP地址
6. req.originalUrl：获取原始请求URL
7. req.params：获取路由的parameters
8. req.path：获取请求路径
9. req.protocol：获取协议类型
10. req.query：获取URL的查询参数串

11. req.route: 获取当前匹配的路由
12. req.subdomains: 获取子域名
13. req.accepts(): 检查可接受的请求的文档类型
14. req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages: 返回指定字符集的第一个可接受字符编码
15. req.get(): 获取指定的HTTP请求头
16. req.is(): 判断请求头Content-Type的MIME类型

Response 对象 - response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。常见属性有：

1. res.app: 同req.app一样
2. res.append(): 追加指定HTTP头
3. res.set()在res.append()后将重置之前设置的头
4. res.cookie(name, value [, option]): 设置Cookie
5. option: domain / expires / httpOnly / maxAge / path / secure / signed
6. res.clearCookie(): 清除Cookie
7. res.download(): 传送指定路径的文件
8. res.get(): 返回指定的HTTP头
9. res.json(): 传送JSON响应
10. res.jsonp(): 传送JSONP响应
11. res.location(): 只设置响应的Location HTTP头，不设置状态码或者close response
12. res.redirect(): 设置响应的Location HTTP头，并且设置状态码302
13. res.render(view,[locals],callback): 渲染一个view，同时向callback传递渲染后的字符串，如果在渲染过程中有错误发生next(err)将会被自动调用。callback将会被传入一个可能发生的错误以及渲染后的页面，这样就不会自动输出了。
14. res.send(): 传送HTTP响应
15. res.sendFile(path [, options] [, fn]): 传送指定路径的文件 -会自动根据文件extension设定 Content-Type
16. res.set(): 设置HTTP头，传入object可以一次设置多个头
17. res.status(): 设置HTTP状态码
18. res.type(): 设置Content-Type的MIME类型

18.5 路由

我们已经了解了 HTTP 请求的基本应用，而路由决定了由谁(指定脚本)去响应客户端请求。

在HTTP请求中，我们可以通过路由提取出请求的URL以及GET/POST参数。

接下来我们扩展 Hello World，添加一些功能来处理更多类型的 HTTP 请求。

创建 express_demo2.js 文件，代码如下所示：

```
1 // express_demo2.js 文件代码：
2 var express = require('express');
3 var app = express();
4
5 // 主页输出 "Hello world"
6 app.get('/', function (req, res) {
7   console.log("主页 GET 请求");
8   res.send('Hello GET');
9 })
10
11
12 // POST 请求
13 app.post('/', function (req, res) {
```



```

14     console.log("主页 POST 请求");
15     res.send('Hello POST');
16 })
17
18 // /del_user 页面响应
19 app.get('/del_user', function (req, res) {
20     console.log("/del_user 响应 DELETE 请求");
21     res.send('删除页面');
22 })
23
24 // /list_user 页面 GET 请求
25 app.get('/list_user', function (req, res) {
26     console.log("/list_user GET 请求");
27     res.send('用户列表页面');
28 })
29
30 // 对页面 abcd, abxcd, ab123cd, 等响应 GET 请求
31 app.get('/ab*cd', function(req, res) {
32     console.log("/ab*cd GET 请求");
33     res.send('正则匹配');
34 })
35
36
37 var server = app.listen(8081, function () {
38
39     var host = server.address().address
40     var port = server.address().port
41
42     console.log("应用实例，访问地址为 http://%s:%s", host, port)
43
44 })

```

执行以上代码：

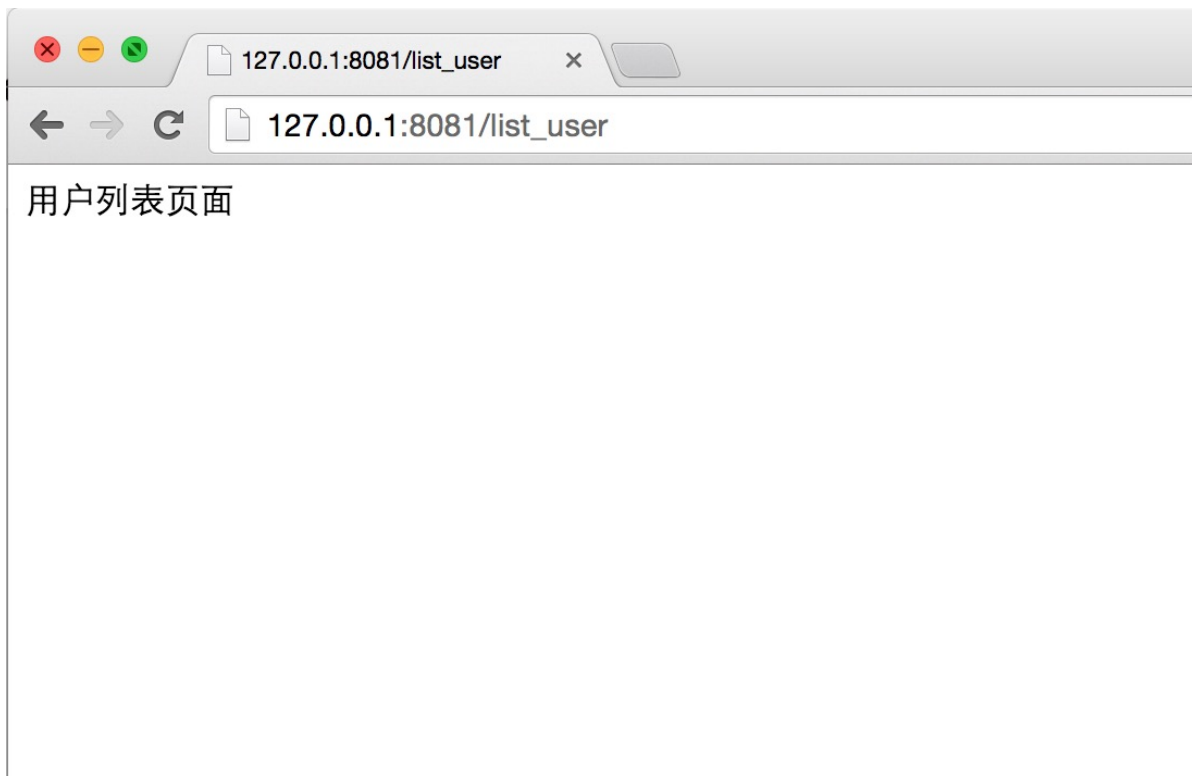
```

1 $ node express_demo2.js
2 应用实例，访问地址为 http://0.0.0.0:8081

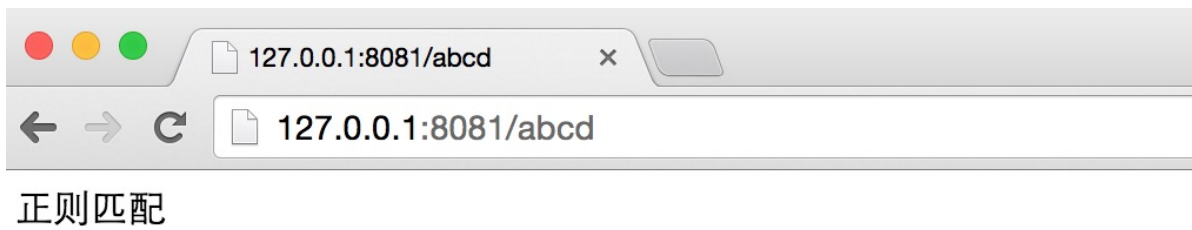
```

接下来你可以尝试访问 <http://127.0.0.1:8081> 不同的地址，查看效果。

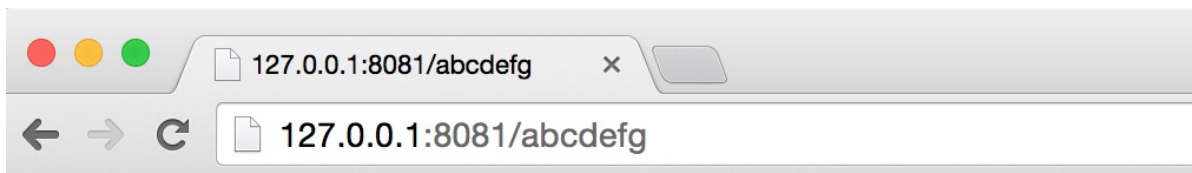
在浏览器中访问 http://127.0.0.1:8081/list_user，结果如下图所示：



在浏览器中访问 <http://127.0.0.1:8081/abcd>，结果如下图所示：



在浏览器中访问 <http://127.0.0.1:8081/abcdefg>，结果如下图所示：



Cannot GET /abcdefg
无法解析该地址

18.6 静态文件

Express 提供了内置的中间件 **express.static** 来设置静态文件如：图片，CSS, JavaScript 等。

你可以使用 **express.static** 中间件来设置静态文件路径。例如，如果你将图片，CSS, JavaScript 文件放在 `public` 目录下，你可以这么写：

```
1 app.use('/public', express.static('public'));
```

我们可以到 `public/images` 目录下放些图片,如下所示：

```
1 node_modules
2 server.js
3 public/
4 public/images
5 public/images/logo.png
```

让我们再修改下 "Hello World" 应用添加处理静态文件的功能。

创建 `express_demo3.js` 文件，代码如下所示：

```
1 // express_demo3.js 文件代码：
2 var express = require('express');
3 var app = express();
4
5 app.use('/public', express.static('public'));
6
7 app.get('/', function (req, res) {
8   res.send('Hello world');
9 })
10
11 var server = app.listen(8081, function () {
12
13   var host = server.address().address
14   var port = server.address().port
```

```

15
16     console.log("应用实例，访问地址为 http://%s:%s", host, port)
17
18 })

```

执行以上代码：

```

1 $ node express_demo3.js
2 应用实例，访问地址为 http://0.0.0.0:8081

```

执行以上代码：

在浏览器中访问 <http://127.0.0.1:8081/public/images/logo.png>（本实例采用了菜鸟教程的 logo），结果如下图所示：



18.7 GET 方法

以下实例演示了在表单中通过 GET 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

```

1 // index.html 文件代码：
2 <html>
3 <body>
4 <form action="http://127.0.0.1:8081/process_get" method="GET">
5 First Name: <input type="text" name="first_name"> <br>
6
7 Last Name: <input type="text" name="last_name">
8 <input type="submit" value="Submit">
9 </form>
10 </body>
11 </html>

```

```

1 // server.js 文件代码：
2 var express = require('express');
3 var app = express();
4
5 app.use('/public', express.static('public'));
6
7 app.get('/index.html', function (req, res) {
8     res.sendFile( __dirname + "/" + "index.html" );
9 })
10
11 app.get('/process_get', function (req, res) {
12
13     // 输出 JSON 格式
14     var response = {
15         "first_name": req.query.first_name,
16         "last_name": req.query.last_name
17     };
18     console.log(response);
19     res.end(JSON.stringify(response));
20 })
21

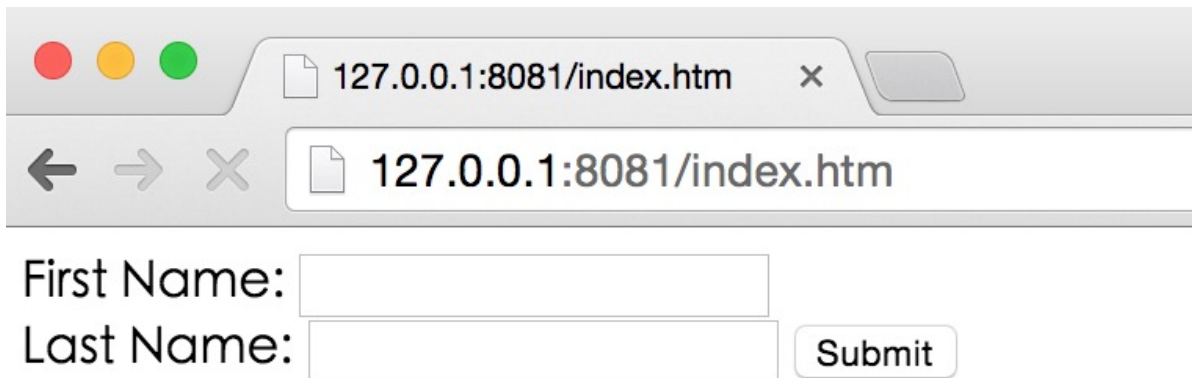
```

```
22 var server = app.listen(8081, function () {
23
24     var host = server.address().address
25     var port = server.address().port
26
27     console.log("应用实例，访问地址为 http://%s:%s", host, port)
28
29 })
```

执行以上代码：

```
1 node server.js
2 应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 <http://127.0.0.1:8081/index.html>，如图所示：



127.0.0.1:8081/index.htm

127.0.0.1:8081/index.htm

First Name:

Last Name:

18.8 POST 方法

以下实例演示了在表单中通过 POST 方法提交两个参数，我们可以使用 server.js 文件内的 `process_post` 路由器来处理输入：

```
1 // index.html 文件代码：
2 <html>
3 <body>
4 <form action="http://127.0.0.1:8081/process_post" method="POST">
5 First Name: <input type="text" name="first_name"> <br>
6
7 Last Name: <input type="text" name="last_name">
8 <input type="submit" value="Submit">
9 </form>
10 </body>
11 </html>
```

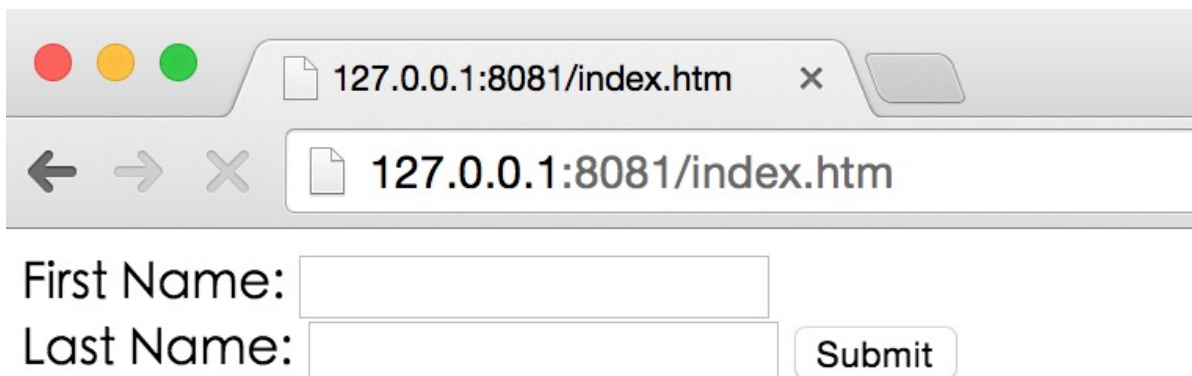
```
1 // server.js 文件代码：
2 var express = require('express');
3 var app = express();
4 var bodyParser = require('body-parser');
5
6 // 创建 application/x-www-form-urlencoded 编码解析
7 var urlencodedParser = bodyParser.urlencoded({ extended: false })
8
```

```
9 app.use('/public', express.static('public'));
10
11 app.get('/index.html', function (req, res) {
12     res.sendFile(__dirname + "/" + "index.html" );
13 })
14
15 app.post('/process_post', urlencodedParser, function (req, res) {
16
17     // 输出 JSON 格式
18     var response = {
19         "first_name":req.body.first_name,
20         "last_name":req.body.last_name
21     };
22     console.log(response);
23     res.end(JSON.stringify(response));
24 })
25
26 var server = app.listen(8081, function () {
27
28     var host = server.address().address
29     var port = server.address().port
30
31     console.log("应用实例, 访问地址为 http://%s:%s", host, port)
32
33 })
```

执行以上代码：

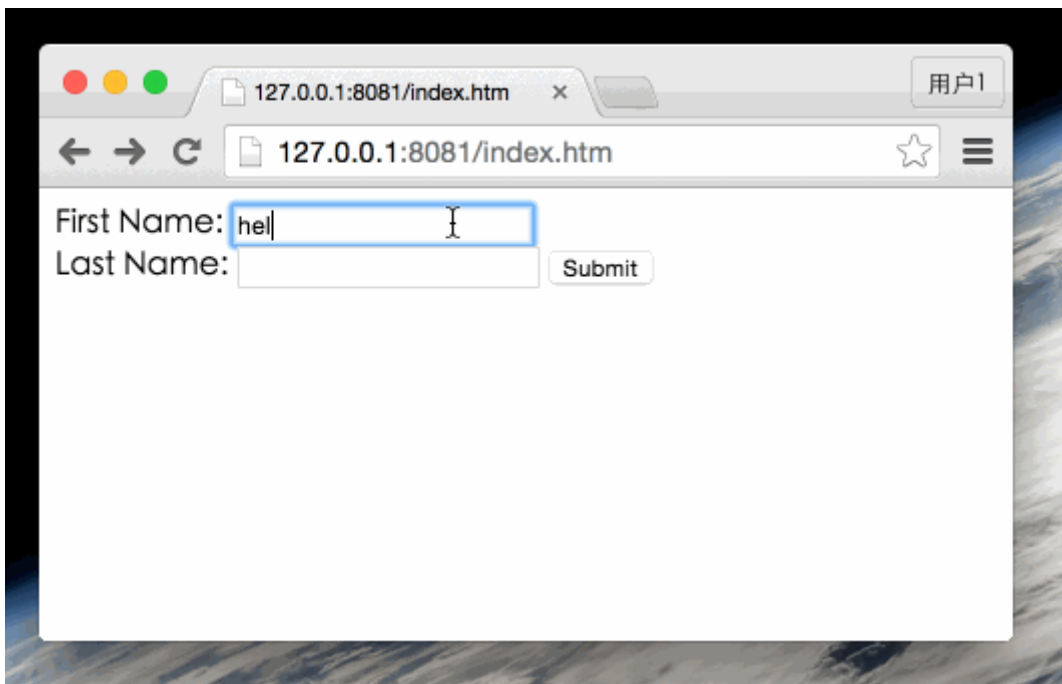
```
1 $ node server.js
2 应用实例, 访问地址为 http://0.0.0.0:8081
```

浏览器访问 <http://127.0.0.1:8081/index.html>，如图所示：



The screenshot shows a web browser window with a single tab titled '127.0.0.1:8081/index.htm'. The address bar also displays '127.0.0.1:8081/index.htm'. Below the address bar, there is a form with two text input fields. The first field is labeled 'First Name:' and the second is labeled 'Last Name:'. To the right of these fields is a button labeled 'Submit'.

现在你可以向表单输入数据，并提交，如下演示：



18.9 文件上传

以下我们创建一个用于上传文件的表单，使用 POST 方法，表单 `enctype` 属性设置为 `multipart/form-data`。

```
1 // index.html 文件代码:
2 <html>
3 <head>
4 <title>文件上传表单</title>
5 </head>
6 <body>
7 <h3>文件上传: </h3>
8 选择一个文件上传: <br />
9 <form action="/file_upload" method="post" enctype="multipart/form-data">
10 <input type="file" name="image" size="50" />
11 <br />
12 <input type="submit" value="上传文件" />
13 </form>
14 </body>
15 </html>
```

```
1 // server.js 文件代码:
2 var express = require('express');
3 var app = express();
4 var fs = require("fs");
5
6 var bodyParser = require('body-parser');
7 var multer = require('multer');
8
9 app.use('/public', express.static('public'));
10 app.use(bodyParser.urlencoded({ extended: false }));
11 app.use(multer({ dest: '/tmp/' }).array('image'));
12
13 app.get('/index.html', function (req, res) {
14   res.sendFile( __dirname + "/" + "index.html" );
15 })
16
```

```

17 app.post('/file_upload', function (req, res) {
18
19     console.log(req.files[0]); // 上传的文件信息
20
21     var des_file = __dirname + "/" + req.files[0].originalname;
22     fs.readFile( req.files[0].path, function (err, data) {
23         fs.writeFile(des_file, data, function (err) {
24             if( err ){
25                 console.log( err );
26             }else{
27                 response = {
28                     message:'File uploaded successfully',
29                     filename:req.files[0].originalname
30                 };
31             }
32             console.log( response );
33             res.end( JSON.stringify( response ) );
34         });
35     });
36 })
37
38 var server = app.listen(8081, function () {
39
40     var host = server.address().address
41     var port = server.address().port
42
43     console.log("应用实例，访问地址为 http://%s:%s", host, port)
44
45 })

```

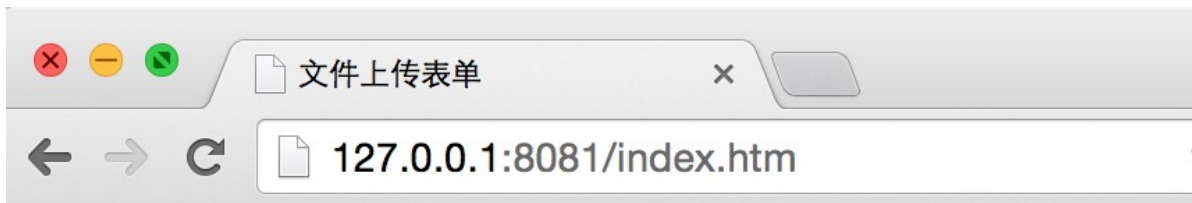
执行以上代码：

```

1 $ node server.js
2 应用实例，访问地址为 http://0.0.0.0:8081

```

浏览器访问 <http://127.0.0.1:8081/index.html>，如图所示：

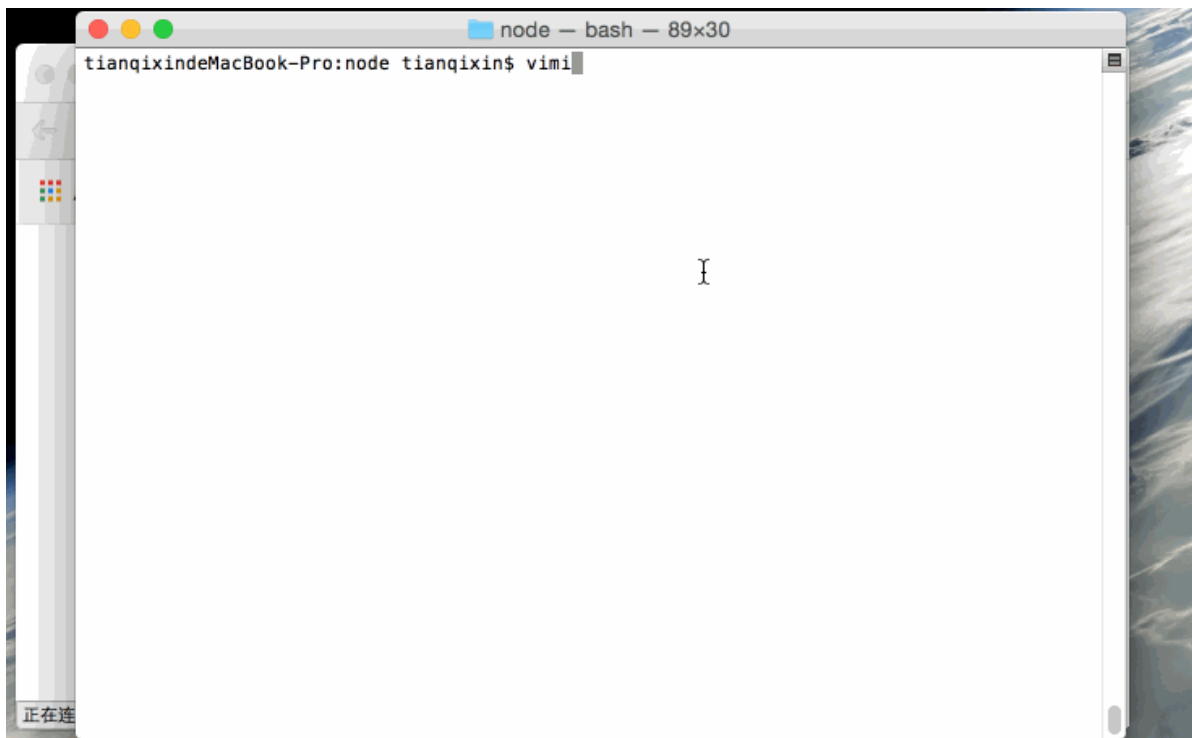


文件上传：

选择一个文件上传：

未选择任何文件

现在你可以向表单输入数据，并提交，如下演示：



18.10 Cookie 管理

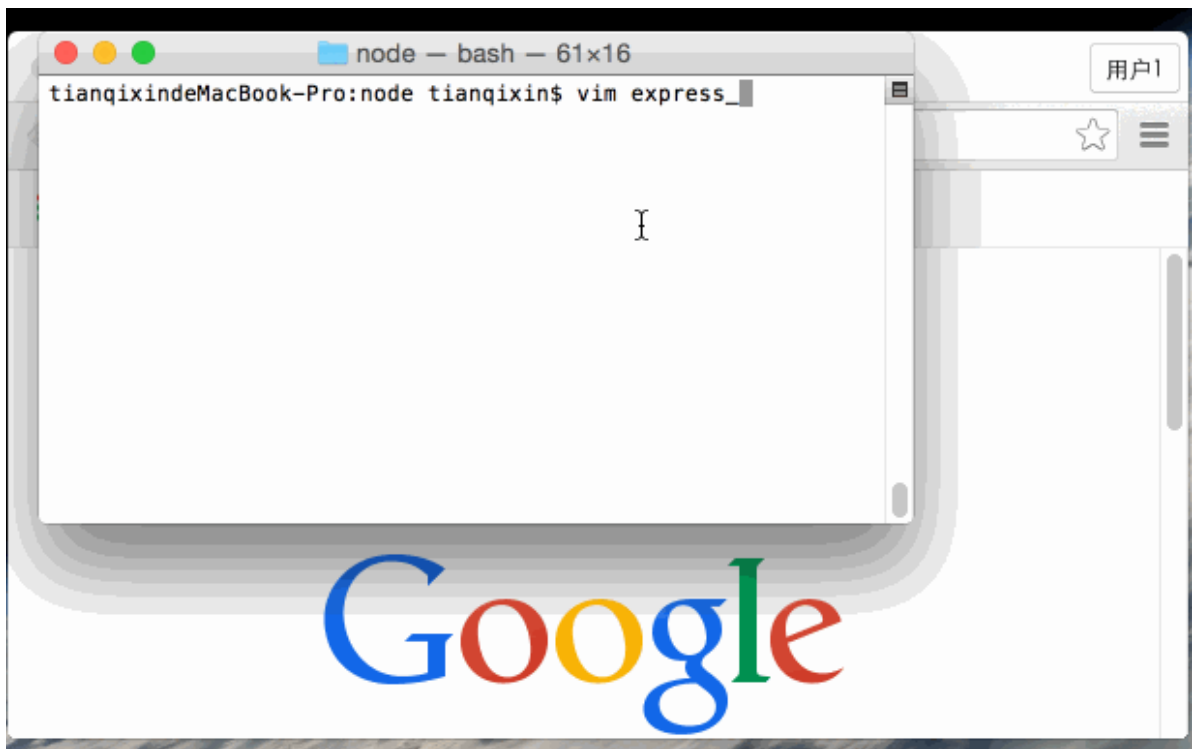
我们可以使用中间件向 Node.js 服务器发送 cookie 信息，以下代码输出了客户端发送的 cookie 信息：

```
1 // express_cookie.js 文件代码：
2 var express = require('express')
3 var cookieParser = require('cookie-parser')
4 var util = require('util');
5
6 var app = express()
7 app.use(cookieParser())
8
9 app.get('/', function(req, res) {
10     console.log("Cookies: " + util.inspect(req.cookies));
11 })
12
13 app.listen(8081)
```

执行以上代码：

```
1 $ node express_cookie.js
```

现在你可以访问 <http://127.0.0.1:8081> 并查看终端信息的输出，如下演示：



18.11 相关资料

- Express官网: <http://expressjs.com/>
- Express4.x API 中文版: [Express4.x API Chinese](#)
- Express4.x API: <http://expressjs.com/zh-cn/4x/api.html>

19. Node.js RESTful API

19.1 什么是 REST?

REST即表述性状态传递（英文：Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。

表述性状态转移是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful。需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML（标准通用标记语言下的一个子集）以及HTML（标准通用标记语言下的一个应用）这些现有的广泛流行的协议和标准。REST 通常使用JSON 数据格式。

19.1.1 HTTP 方法

以下为 REST 基本架构的四个方法：

- **GET** - 用于获取数据。
- **PUT** - 用于更新或添加数据。
- **DELETE** - 用于删除数据。
- **POST** - 用于添加数据。

19.2 RESTful Web Services

Web service是一个平台独立的，低耦合的，自包含的、基于可编程的web的应用程序，可使用开放的XML（标准通用标记语言下的一个子集）标准来描述、发布、发现、协调和配置这些应用程序，用于开发分布式的互操作的应用程序。

基于 REST 架构的 Web Services 即是 RESTful。

由于轻量级以及通过 HTTP 直接传输数据的特性，Web 服务的 RESTful 方法已经成为最常见的替代方法。可以使用各种语言（比如 Java 程序、Perl、Ruby、Python、PHP 和 Javascript[包括 Ajax]）实现客户端。

RESTful Web 服务通常可以通过自动客户端或代表用户的应用程序访问。但是，这种服务的简便性让用户能够与之直接交互，使用它们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

更多介绍，可以查看：[RESTful 架构详解](#)

19.3 创建 RESTful

首先，创建一个 json 数据资源文件 users.json，内容如下：

```
1 {
2   "user1" : {
3     "name" : "mahesh",
4     "password" : "password1",
5     "profession" : "teacher",
6     "id": 1
7   },
8   "user2" : {
9     "name" : "suresh",
10    "password" : "password2",
11    "profession" : "librarian",
12    "id": 2
13  },
14  "user3" : {
15    "name" : "ramesh",
16    "password" : "password3",
17    "profession" : "clerk",
18    "id": 3
19  }
20 }
```

基于以上数据，我们创建以下 RESTful API：

序号	URI	HTTP 方法	发送内容	结果
1	listUsers	GET	空	显示所有用户列表
2	addUser	POST	JSON 字符串	添加新用户
3	deleteUser	DELETE	JSON 字符串	删除用户
4	:id	GET	空	显示用户详细信息

19.3.1 获取用户列表

以下代码，我们创建了 RESTful API **listUsers**，用于读取用户的信息列表，server.js 文件代码如下所示：

```
1 var express = require('express');
2 var app = express();
3 var fs = require("fs");
4
5 app.get('/listUsers', function (req, res) {
```

```

6     fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data)
7     {
8         console.log( data );
9         res.end( data );
10    });
11  })
12  var server = app.listen(8081, function () {
13
14      var host = server.address().address
15      var port = server.address().port
16
17      console.log("应用实例，访问地址为 http://%s:%s", host, port)
18
19  })

```

接下来执行以下命令：

```

1 $ node server.js
2 应用实例，访问地址为 http://0.0.0.0:8081

```

在浏览器中访问 <http://127.0.0.1:8081/listUsers>，结果如下所示：

```

1  {
2      "user1" : {
3          "name" : "mahesh",
4          "password" : "password1",
5          "profession" : "teacher",
6          "id": 1
7      },
8      "user2" : {
9          "name" : "suresh",
10         "password" : "password2",
11         "profession" : "librarian",
12         "id": 2
13     },
14     "user3" : {
15         "name" : "ramesh",
16         "password" : "password3",
17         "profession" : "clerk",
18         "id": 3
19     }
20 }

```

19.3.2 添加用户

以下代码，我们创建了 RESTful API **addUser**，用于添加新的用户数据，server.js 文件代码如下所示：

```

1  var express = require('express');
2  var app = express();
3  var fs = require("fs");
4
5  //添加的新用户数据
6  var user = {
7      "user4" : {
8          "name" : "mohit",

```

```

9      "password" : "password4",
10     "profession" : "teacher",
11     "id": 4
12   }
13 }
14
15 app.get('/addUser', function (req, res) {
16   // 读取已存在的数据
17   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data)
18   {
19     data = JSON.parse( data );
20     data["user4"] = user["user4"];
21     console.log( data );
22     res.end( JSON.stringify(data));
23   });
24 })
25
26 var server = app.listen(8081, function () {
27
28   var host = server.address().address
29   var port = server.address().port
30   console.log("应用实例，访问地址为 http://%s:%s", host, port)
31 })

```

接下来执行以下命令：

```

1 $ node server.js
2 应用实例，访问地址为 http://0.0.0.0:8081

```

在浏览器中访问 <http://127.0.0.1:8081/addUser>，结果如下所示：

```

1 { user1:
2   { name: 'mahesh',
3     password: 'password1',
4     profession: 'teacher',
5     id: 1 },
6   user2:
7     { name: 'suresh',
8       password: 'password2',
9       profession: 'librarian',
10      id: 2 },
11   user3:
12     { name: 'ramesh',
13       password: 'password3',
14       profession: 'clerk',
15       id: 3 },
16   user4:
17     { name: 'mohit',
18       password: 'password4',
19       profession: 'teacher',
20       id: 4 }
21 }

```

19.3.3 显示用户详情

以下代码，我们创建了 RESTful API **:id (用户id)**，用于读取指定用户的详细信息，server.js 文件代码如下所示：

```
1  var express = require('express');
2  var app = express();
3  var fs = require("fs");
4
5  app.get('/:id', function (req, res) {
6    // 首先我们读取已存在的用户
7    fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data)
8    {
9      data = JSON.parse( data );
10     var user = data["user" + req.params.id]
11     console.log( user );
12     res.end( JSON.stringify(user));
13   });
14 })
15
16 var server = app.listen(8081, function () {
17
18   var host = server.address().address
19   var port = server.address().port
20   console.log("应用实例，访问地址为 http://%s:%s", host, port)
21
22 })
```

接下来执行以下命令：

```
1  $ node server.js
2  应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 <http://127.0.0.1:8081/2>，结果如下所示：

```
1  {
2    "name": "suresh",
3    "password": "password2",
4    "profession": "librarian",
5    "id": 2
6  }
```

19.3.4 删除用户

以下代码，我们创建了 RESTful API **deleteUser**，用于删除指定用户的详细信息，以下实例中，用户 id 为 2，server.js 文件代码如下所示：

```
1  var express = require('express');
2  var app = express();
3  var fs = require("fs");
4
5  var id = 2;
6
7  app.get('/deleteUser', function (req, res) {
8
```

```

9      // First read existing users.
10     fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data)
11     {
12         data = JSON.parse( data );
13         delete data["user" + id];
14
15         console.log( data );
16         res.end( JSON.stringify(data));
17     });
18 })
19 var server = app.listen(8081, function () {
20
21     var host = server.address().address
22     var port = server.address().port
23     console.log("应用实例，访问地址为 http://%s:%s", host, port)
24
25 })

```

接下来执行以下命令：

```

1 $ node server.js
2 应用实例，访问地址为 http://0.0.0.0:8081

```

在浏览器中访问 <http://127.0.0.1:8081/deleteUser>，结果如下所示：

```

1 { user1:
2   { name: 'mahesh',
3     password: 'password1',
4     profession: 'teacher',
5     id: 1 },
6   user3:
7     { name: 'ramesh',
8       password: 'password3',
9       profession: 'clerk',
10      id: 3 }
11 }

```

20. Node.js 多进程

我们都知道 Node.js 是以单线程的模式运行的，但它使用的是事件驱动来处理并发，这样有助于我们在多核 cpu 的系统上创建多个子进程，从而提高性能。

每个子进程总是带有三个流对象：child.stdin, child.stdout 和 child.stderr。他们可能会共享父进程的 stdio 流，或者也可以是独立的被导流的流对象。

Node 提供了 child_process 模块来创建子进程，方法有：

- **exec** - child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。
- **spawn** - child_process.spawn 使用指定的命令行参数创建新进程。
- **fork** - child_process.fork 是 spawn() 的特殊形式，用于在子进程中运行的模块，如 fork('./son.js') 相当于 spawn('node', ['./son.js'])。与 spawn 方法不同的是，fork 会在父进程与子进程之间，建立一个通信管道，用于进程之间的通信。

20.1 exec() 方法

child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。

语法如下所示：

```
1 child_process.exec(command[, options], callback)
```

20.1.1 参数

参数说明如下：

command：字符串，将要运行的命令，参数使用空格隔开

options：对象，可以是：

- cwd，字符串，子进程的当前工作目录
- env，对象 环境变量键值对
- encoding，字符串，字符编码（默认：'utf8'）
- shell，字符串，将要执行命令的 Shell（默认：在 UNIX 中为 /bin/sh，在 Windows 中为 cmd.exe，Shell 应当能识别 -c 开关在 UNIX 中，或 /s /c 在 Windows 中。在 Windows 中，命令行解析应当能兼容 cmd.exe）
- timeout，数字，超时时间（默认：0）
- maxBuffer，数字，在 stdout 或 stderr 中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死（默认：200*1024）
- killSignal，字符串，结束信号（默认：'SIGTERM'）
- uid，数字，设置用户进程的 ID
- gid，数字，设置进程组的 ID

callback：回调函数，包含三个参数error, stdout 和 stderr。

exec() 方法返回最大的缓冲区，并等待进程结束，一次性返回缓冲区的内容。

20.1.2 实例

让我们创建两个 js 文件 support.js 和 master.js。

```
1 // support.js 文件代码：
2 console.log("进程 " + process.argv[2] + " 执行。");
```

```
1 // master.js 文件代码：
2 const fs = require('fs');
3 const child_process = require('child_process');
4
5 for(var i=0; i<3; i++) {
6     var workerProcess = child_process.exec('node support.js '+i, function
7     (error, stdout, stderr) {
8         if (error) {
9             console.log(error.stack);
10            console.log('Error code: '+error.code);
11            console.log('signal received: '+error.signal);
12        }
13        console.log('stdout: ' + stdout);
14        console.log('stderr: ' + stderr);
15    });
16 }
```



```

15
16     workerProcess.on('exit', function (code) {
17         console.log('子进程已退出，退出码 '+code);
18     });
19 }

```

执行以上代码，输出结果为：

```

1  $ node master.js
2  子进程已退出，退出码 0
3  stdout: 进程 1 执行。
4
5  stderr:
6  子进程已退出，退出码 0
7  stdout: 进程 0 执行。
8
9  stderr:
10 子进程已退出，退出码 0
11 stdout: 进程 2 执行。
12
13 stderr:

```

20.2 spawn() 方法

child_process.spawn 使用指定的命令行参数创建新进程，语法格式如下：

```

1  child_process.spawn(command[, args][, options])

```

20.2.1 参数

参数说明如下：

command： 将要运行的命令

args： Array 字符串参数数组

options Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- stdio Array|String 子进程的 stdio 配置
- detached Boolean 这个子进程将会变成进程组的领导
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

spawn() 方法返回流 (stdout & stderr)，在进程返回大量数据时使用。进程一旦开始执行时 spawn() 就开始接收响应。

20.2.2 实例

让我们创建两个 js 文件 support.js 和 master.js。

```

1  // support.js 文件代码：
2  console.log("进程 " + process.argv[2] + " 执行。" );

```

```

1  // master.js 文件代码：

```

```

2  const fs = require('fs');
3  const child_process = require('child_process');
4
5  for(var i=0; i<3; i++) {
6      var workerProcess = child_process.spawn('node', ['support.js', i]);
7
8      workerProcess.stdout.on('data', function (data) {
9          console.log('stdout: ' + data);
10     });
11
12     workerProcess.stderr.on('data', function (data) {
13         console.log('stderr: ' + data);
14     });
15
16     workerProcess.on('close', function (code) {
17         console.log('子进程已退出, 退出码 '+code);
18     });
19 }

```

执行以上代码，输出结果为：

```

1  $ node master.js stdout: 进程 0 执行。
2
3  子进程已退出, 退出码 0
4  stdout: 进程 1 执行。
5
6  子进程已退出, 退出码 0
7  stdout: 进程 2 执行。
8
9  子进程已退出, 退出码 0

```

20.3 fork 方法

`child_process.fork` 是 `spawn()` 方法的特殊形式，用于创建进程，语法格式如下：

```

1  child_process.fork(modulePath[, args][, options])

```

20.3.1 参数

参数说明如下：

modulePath：String，将要在子进程中运行的模块

args：Array 字符串参数数组

options：Object

- `cwd` String 子进程的当前工作目录
- `env` Object 环境变量键值对
- `execPath` String 创建子进程的可执行文件
- `execArgv` Array 子进程的可执行文件的字符串参数数组（默认： `process.execArgv`）
- `silent` Boolean 如果为 `true`，子进程的 `stdin`，`stdout` 和 `stderr` 将会被关联至父进程，否则，它们将会从父进程中继承。（默认为： `false`）
- `uid` Number 设置用户进程的 ID
- `gid` Number 设置进程组的 ID

返回的对象除了拥有ChildProcess实例的所有方法，还有一个内建的通信信道。

20.3.2 实例

让我们创建两个 js 文件 support.js 和 master.js。

```
1 // support.js 文件代码:
2 console.log("进程 " + process.argv[2] + " 执行。");
```

```
1 // master.js 文件代码:
2 const fs = require('fs');
3 const child_process = require('child_process');
4
5 for(var i=0; i<3; i++) {
6     var worker_process = child_process.fork("support.js", [i]);
7
8     worker_process.on('close', function (code) {
9         console.log('子进程已退出, 退出码 ' + code);
10    });
11 }
```

执行以上代码，输出结果为：

```
1 $ node master.js
2 进程 0 执行。
3 子进程已退出, 退出码 0
4 进程 1 执行。
5 子进程已退出, 退出码 0
6 进程 2 执行。
7 子进程已退出, 退出码 0
```

21. Node.js JXcore 打包

Node.js 是一个开放源代码、跨平台的、用于服务器端和网络应用的运行环境。

JXcore 是一个支持多线程的 Node.js 发行版本，基本不需要对你现有的代码做任何改动就可以直接线程安全地以多线程运行。

这篇文章主要是要向大家介绍 JXcore 的打包功能。

21.1 JXcore 安装

下载 JXcore 安装包，并解压，在解压的的目录下提供了 jx 二进制文件命令，接下来我们主要使用这个命令。

21.1.1 步骤1、下载

下载 JXcore 安装包 <https://github.com/jxcore/jxcore-release>，你需要根据你自己的系统环境来下载安装包。

1、Window 平台下载： [Download\(Windows x64 \(V8\)\)](#)。

2、Linux/OSX 安装命令：

```
1 $ curl
  https://raw.githubusercontent.com/jxcore/jxcore/master/tools/jx_install.sh |
  bash
```

如果权限不足，可以使用以下命令：

```
1 $ curl
  https://raw.githubusercontent.com/jxcore/jxcore/master/tools/jx_install.sh |
  sudo bash
```

以上步骤如果操作正确，使用以下命令，会输出版本号信息：

```
1 $ jx --version
2 v0.10.32
```

21.2 包代码

例如，我们的 Node.js 项目包含以下几个文件，其中 index.js 是主文件：

```
1 drwxr-xr-x  2 root root  4096 Nov 13 12:42 images
2 -rwxr-xr-x  1 root root 30457 Mar  6 12:19 index.htm
3 -rwxr-xr-x  1 root root 30452 Mar  1 12:54 index.js
4 drwxr-xr-x 23 root root  4096 Jan 15 03:48 node_modules
5 drwxr-xr-x  2 root root  4096 Mar 21 06:10 scripts
6 drwxr-xr-x  2 root root  4096 Feb 15 11:56 style
```

接下来我们使用 **jx** 命令打包以上项目，并指定 index.js 为 Node.js 项目的主文件：

```
1 $ jx package index.js index
```

以上命令执行成功，会生成以下两个文件：

- **index.jxp** 这是一个中间件文件，包含了需要编译的完整项目信息。
- **index.jx** 这是一个完整包信息的二进制文件，可运行在客户端上。

21.3 载入 JX 文件

Node.js 的项目运行：

```
1 $ node index.js command_line_arguments
```

使用 JXcore 编译后，我们可以使用以下命令来执行生成的 jx 二进制文件：

```
1 $ jx index.jx command_line_arguments
```

更多 JXcore 安装参考：<https://github.com/jxcore/jxcore/blob/master/doc/INSTALLATION.md>。

更多 JXcore 功能特性你可以参考官网：<https://github.com/jxcore/jxcore>。

22. Node.js 连接 MySQL

本章节我们将为大家介绍如何使用 Node.js 来连接 MySQL，并对数据库进行操作。

如果你还没有 MySQL 的基本知识，可以参考我们的教程：[MySQL 教程](#)。

本教程使用到的 Websites 表 SQL 文件：[websites.sql](#)。

22.1 安装驱动

本教程使用了[淘宝定制的 cnpm 命令](#)进行安装：

```
1 | $ cnpm install mysql
```

22.2 连接数据库

在以下实例中根据你的实际配置修改数据库用户名、及密码及数据库名：

```
1  // test.js 文件代码：
2  var mysql      = require('mysql');
3  var connection = mysql.createConnection({
4    host        : 'localhost',
5    user        : 'root',
6    password    : '123456',
7    database    : 'test'
8  });
9
10 connection.connect();
11
12 connection.query('SELECT 1 + 1 AS solution', function (error, results,
13   fields) {
14   if (error) throw error;
15   console.log('The solution is: ', results[0].solution);
16 });
```

执行以下命令输出结果为：

```
1 | $ node test.js
2 | The solution is: 2
```

22.3 数据库连接参数说明

参数	描述
host	主机地址（默认：localhost）
user	用户名
password	密码
port	端口号（默认：3306）
database	数据库名
charset	连接字符集（默认：'UTF8_GENERAL_CI'，注意字符集的字母都要大写）
localAddress	此IP用于TCP连接（可选）
socketPath	连接到unix域路径，当使用 host 和 port 时会被忽略
timezone	时区（默认：'local'）
connectTimeout	连接超时（默认：不限制；单位：毫秒）
stringifyObjects	是否序列化对象
typeCast	是否将列值转化为本地JavaScript类型值（默认：true）
queryFormat	自定义query语句格式化方法
supportBigNumbers	数据库支持bigint或decimal类型列时，需要设此option为true（默认：false）
bigNumberStrings	supportBigNumbers和bigNumberStrings启用 强制bigint或decimal列以JavaScript字符串类型返回（默认：false）
dateStrings	强制timestamp,datetime,data类型以字符串类型返回，而不是JavaScript Date类型（默认：false）
debug	开启调试（默认：false）
multipleStatements	是否许一个query中有多个MySQL语句（默认：false）
flags	用于修改连接标志
ssl	使用ssl参数（与crypto.createCredenitals参数格式一至）或一个包含ssl配置文件名称的字符串，目前只捆绑Amazon RDS的配置文件

更多说明可参见：<https://github.com/mysqljs/mysql>

22.4 数据库操作(CURD)

在进行数据库操作前，你需要将本站提供的 Websites 表 SQL 文件[websites.sql](#) 导入到你的 MySQL 数据库中。

本教程测试的 MySQL 用户名为 root，密码为 123456，数据库为 test，你需要根据自己配置情况修改。

22.4.1 查询数据

将上面我们提供的 SQL 文件导入数据库后，执行以下代码即可查询出数据：

```
1 var mysql = require('mysql');
2
3 var connection = mysql.createConnection({
4   host      : 'localhost',
5   user      : 'root',
6   password  : '123456',
7   port      : '3306',
8   database  : 'test'
9 });
10
11 connection.connect();
12
13 var sql = 'SELECT * FROM websites';
14 //查
15 connection.query(sql,function (err, result) {
16   if(err){
17     console.log('[SELECT ERROR] - ',err.message);
18     return;
19   }
20
21   console.log('-----SELECT-----
22   ----');
23   console.log(result);
24   console.log('-----
25   ----\n\n');
26 });
27
28 connection.end();
```

执行以下命令输出就结果为：

```
1 $ node test.js
2 -----SELECT-----
3 [ RowDataPacket {
4   id: 1,
5   name: 'Google',
6   url: 'https://www.google.cm/',
7   alexa: 1,
8   country: 'USA' },
9   RowDataPacket {
10    id: 2,
11    name: '淘宝',
12    url: 'https://www.taobao.com/',
13    alexa: 13,
14    country: 'CN' },
15   RowDataPacket {
16    id: 3,
17    name: '菜鸟教程',
18    url: 'http://www.runoob.com/',
19    alexa: 4689,
20    country: 'CN' },
21   RowDataPacket {
22    id: 4,
```

```

23     name: '微博',
24     url: 'http://weibo.com/',
25     alexa: 20,
26     country: 'CN' },
27   RowDataPacket {
28     id: 5,
29     name: 'Facebook',
30     url: 'https://www.facebook.com/',
31     alexa: 3,
32     country: 'USA' } ]
33   -----

```

22.4.2 插入数据

我们可以向数据表 websites 插入数据：

```

1  var mysql = require('mysql');
2
3  var connection = mysql.createConnection({
4    host      : 'localhost',
5    user      : 'root',
6    password  : '123456',
7    port      : '3306',
8    database  : 'test'
9  });
10
11 connection.connect();
12
13 var addSql = 'INSERT INTO websites(Id,name,url,alexa,country)
14 VALUES(0,?,?,?,?)';
15 var addSqlParams = ['菜鸟工具', 'https://c.runoob.com', '23453', 'CN'];
16 //增
17 connection.query(addSql,addSqlParams,function (err, result) {
18   if(err){
19     console.log('[INSERT ERROR] - ',err.message);
20     return;
21   }
22   console.log('-----INSERT-----
23   ');
24   //console.log('INSERT ID:',result.insertId);
25   console.log('INSERT ID:',result);
26   console.log('-----
27   \n\n');
28 });
29 connection.end();

```

执行以下命令输出就结果为：


```

1 $ node test.js
2 -----INSERT-----
3 INSERT ID: OkPacket {
4   fieldCount: 0,
5   affectedRows: 1,
6   insertId: 6,
7   serverStatus: 2,
8   warningCount: 0,
9   message: '',
10  protocol41: true,
11  changedRows: 0 }
12 -----

```

执行成功后，查看数据表，即可以看到添加的数据：

1	Google	https://www.googl	1	USA
2	淘宝	https://www.taoba	13	CN
3	菜鸟教程	http://www.runoot	4689	CN
4	微博	http://weibo.com/	20	CN
5	Facebook	https://www.faceb	3	USA
6	菜鸟工具	https://c.runoob.c	23453	CN

插入的数据

22.4.3 更新数据

我们也可以对数据库的数据进行修改：

```

1 var mysql = require('mysql');
2
3 var connection = mysql.createConnection({
4   host      : 'localhost',
5   user      : 'root',
6   password  : '123456',
7   port      : '3306',
8   database  : 'test'
9 });
10
11 connection.connect();
12
13 var modSql = 'UPDATE websites SET name = ?,url = ? WHERE Id = ?';
14 var modSqlParams = ['菜鸟移动站', 'https://m.runoob.com',6];
15 //改
16 connection.query(modSql,modSqlParams,function (err, result) {
17   if(err){
18     console.log('[UPDATE ERROR] - ',err.message);
19     return;
20   }
21   console.log('-----UPDATE-----
22 ');
23   console.log('UPDATE affectedRows',result.affectedRows);
24   console.log('-----\n\n');
25 });
26 connection.end();

```

执行以下命令输出就结果为：

```
1 -----UPDATE-----
2 UPDATE affectedRows 1
3 -----
```

执行成功后，查看数据表，即可以看到更新的数据：

	id	name	url	alexa	country
	1	Google	https://www.google.cm/	1	USA
	2	淘宝	https://www.taobao.com/	13	CN
	3	菜鸟教程	http://www.runoob.com/	4689	CN
	4	微博	http://weibo.com/	20	CN
	5	Facebook	https://www.facebook.com/	3	USA
	6	菜鸟移动站	https://m.runoob.com	23453	CN

22.4.4 删除数据

我们可以使用以下代码来删除 id 为 6 的数据：

```
1 var mysql = require('mysql');
2
3 var connection = mysql.createConnection({
4   host      : 'localhost',
5   user      : 'root',
6   password  : '123456',
7   port      : '3306',
8   database  : 'test'
9 });
10
11 connection.connect();
12
13 var delSql = 'DELETE FROM websites where id=6';
14 //删
15 connection.query(delSql,function (err, result) {
16   if(err){
17     console.log('[DELETE ERROR] - ',err.message);
18     return;
19   }
20
21   console.log('-----DELETE-----');
22   console.log('DELETE affectedRows',result.affectedRows);
23   console.log('-----\n\n');
24 });
25
26 connection.end();
```

执行以下命令输出就结果为：

```
1 -----DELETE-----
2 DELETE affectedRows 1
3 -----
```

执行成功后，查看数据表，即可以看到 id=6 的数据已被删除：

1	Google	https://www.google.cm/	1	USA
2	淘宝	https://www.taobao.com/	13	CN
3	菜鸟教程	http://www.runoob.com/	4689	CN
4	微博	http://weibo.com/	20	CN
5	Facebook	https://www.facebook.com/	3	USA
id 为 6 的数据已被删除				