

Node.js

专业人士笔记

Chapter 15: Executing files or commands with Child Processes

Section 15.1: Spawning a new process to execute a command

To spawn a new process in which you need unbuffered output (e.g. long-running processes) we output over a period of time rather than printing and exiting immediately, use `child_process.spawn()`. This method spawns a new process using a given command and an array of arguments. Both are instances of `ChildProcess`, which in turn provides the `stdout` and `stderr` properties. Both are instances of `Stream`.

The following code is equivalent to using running the command `ls -l /usr`:

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-l', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});

Another example command:
lsb -l /var/www/html

```

Maybe be written as:

```
spawn('lsb', ['-l', '/var/www/html']).on('data', (data) => {
  console.log(`stdout: ${data}`);
}).on('error', (err) => {
  console.log(`Error: ${err}`);
});
```

Section 15.2: Spawning a shell to execute a command

To run a command in a shell, in which you required buffered output (i.e. it's child process). For example, if you wanted to run the command `cat file.txt`:

```
const exec = require('child_process').exec;
exec('cat file.txt', (err, stdout, stderr) => {
  if (err) {
    console.error(`error: ${err}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

The function accepts up to three parameters:

Notes for Professionals

Chapter 21: Using Streams

Parameter

Readable Stream: type of stream where data can be read from
Writable Stream: type of stream where data can be written to
Duplex Stream: type of stream that is both readable and writable
Transform Stream: type of duplex stream that can transform data as it is being read and then written

Section 21.1: Read Data from TextFile with Streams

I/O in node is asynchronous, so interacting with the disk and network involves passing callbacks to functions. You might be tempted to write code that serves up a file from disk like this:

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

var server = http.createServer(function (req, res) {
  fs.readFile('data.txt', function (err, data) {
    res.end(data);
  });
});

server.listen(8000);
```

This code works but it's bulky and buffers up the entire `data.txt` file into memory for every request before writing the result back to clients. If data is very large, your program could start eating a lot of memory as it serves lots of users concurrently, particularly for users on slow connections.

The user experience is poor too because users will need to wait for the whole file to be buffered into memory on your server before they can start receiving any contents.

Luckily both the `req` and `res` arguments are streams, which means we can write this in a much better way using `fs.createReadStream()` instead of `fs.readFileSync()`:

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

var server = http.createServer(function (req, res) {
  var stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(res);
});

server.listen(8000);
```

Here `pipe()` takes care of listening for 'data' and 'end' events from the `fs.createReadStream()`. This code is not cleaner, but now the `data.txt` file will be written to clients one chunk at a time immediately as they are requested from the disk.

Section 21.2: Piping streams

Readable streams can be "piped" or connected to writable streams. This makes data flow from the source to the destination stream without much effort.

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

var readable = fs.createReadStream('file.txt');

readable.pipe(process.stdout);
```

Notes for Professionals

Chapter 57: TCP Sockets

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple TCP server

Section 57.1: A simple

目录

关于	1
第1章：Node.js入门	2
第1.1节：Hello World HTTP服务器	4
第1.2节：Hello World命令行	5
第1.3节：使用Express的Hello World	6
第1.4节：安装和运行Node.js	6
第1.5节：调试你的NodeJS应用程序	7
第1.6节：Hello World基础路由	7
第1.7节：REPL中的Hello World	8
第1.8节：在线部署你的应用程序	9
第1.9节：核心模块	9
第1.10节：TLS套接字：服务器和客户端	14
第1.11节：如何搭建一个基本的HTTPS网页服务器！	16
第2章：npm	19
第2.1节：安装软件包	19
第2.2节：卸载软件包	22
第2.3节：设置软件包配置	23
第2.4节：运行脚本	24
第2.5节：基础语义版本控制	24
第2.6节：发布软件包	25
第2.7节：移除多余的软件包	26
第2.8节：列出目前已安装的软件包	26
第2.9节：更新npm和软件包	26
第2.10节：作用域和仓库	27
第2.11节：链接项目以加快调试和开发	27
第2.12节：将模块锁定到特定版本	28
第2.13节：为全局安装的软件包进行设置	28
第3章：使用Express构建Web应用	30
第3.1节：入门	30
第3.2节：基础路由	31
第3.3节：模块化Express应用	32
第3.4节：使用模板引擎	33
第3.5节：使用ExpressJS的JSON API	34
第3.6节：提供静态文件	35
第3.7节：添加中间件	36
第3.8节：错误处理	36
第3.9节：从请求中获取信息	37
第3.10节：Express中的错误处理	38
第3.11节：钩子：如何在任何请求之前和任何响应之后执行代码	38
第3.12节：使用cookie-parser设置Cookie	39
第3.13节：Express中的自定义中间件	39
第3.14节：Django风格的命名路由	39
第3.15节：你好，世界	40
第3.16节：使用中间件和next回调	40
第3.17节：错误处理	42
第3.18节：处理POST请求	43
第4章：文件系统输入输出	45

Contents

About	1
Chapter 1: Getting started with Node.js	2
Section 1.1: Hello World HTTP server	4
Section 1.2: Hello World command line	5
Section 1.3: Hello World with Express	6
Section 1.4: Installing and Running Node.js	6
Section 1.5: Debugging Your NodeJS Application	7
Section 1.6: Hello World basic routing	7
Section 1.7: Hello World in the REPL	8
Section 1.8: Deploying your application online	9
Section 1.9: Core modules	9
Section 1.10: TLS Socket: server and client	14
Section 1.11: How to get a basic HTTPS web server up and running!	16
Chapter 2: npm	19
Section 2.1: Installing packages	19
Section 2.2: Uninstalling packages	22
Section 2.3: Setting up a package configuration	23
Section 2.4: Running scripts	24
Section 2.5: Basic semantic versioning	24
Section 2.6: Publishing a package	25
Section 2.7: Removing extraneous packages	26
Section 2.8: Listing currently installed packages	26
Section 2.9: Updating npm and packages	26
Section 2.10: Scopes and repositories	27
Section 2.11: Linking projects for faster debugging and development	27
Section 2.12: Locking modules to specific versions	28
Section 2.13: Setting up for globally installed packages	28
Chapter 3: Web Apps With Express	30
Section 3.1: Getting Started	30
Section 3.2: Basic routing	31
Section 3.3: Modular express application	32
Section 3.4: Using a Template Engine	33
Section 3.5: JSON API with ExpressJS	34
Section 3.6: Serving static files	35
Section 3.7: Adding Middleware	36
Section 3.8: Error Handling	36
Section 3.9: Getting info from the request	37
Section 3.10: Error handling in Express	38
Section 3.11: Hook: How to execute code before any req and after any res	38
Section 3.12: Setting cookies with cookie-parser	39
Section 3.13: Custom middleware in Express	39
Section 3.14: Named routes in Django-style	39
Section 3.15: Hello World	40
Section 3.16: Using middleware and the next callback	40
Section 3.17: Error handling	42
Section 3.18: Handling POST Requests	43
Chapter 4: Filesystem I/O	45

第4.1节：异步读取文件	45
第4.2节：使用readdir或readdirSync列出目录内容	45
第4.3节：通过管道流复制文件	46
第4.4节：同步读取文件	47
第4.5节：检查文件或目录的权限	47
第4.6节：检查文件或目录是否存在	48
第4.7节：确定文本文件的行数	49
第4.8节：逐行读取文件	49
第4.9节：创建或使用现有目录时避免竞态条件	49
第4.10节：使用流克隆文件	50
第4.11节：使用writeFile或writeFileSync写入文件	51
第4.12节：更改文本文件的内容	51
第4.13节：使用unlink或unlinkSync删除文件	52
第4.14节：使用流将文件读取到缓冲区	52
第5章：导出和使用模块	53
第5.1节：创建hello-world.js模块	53
第5.2节：加载和使用模块	54
第5.3节：文件夹作为模块	55
第5.4节：每个模块仅注入一次	55
第5.5节：从node_modules加载模块	56
第5.6节：构建你自己的模块	56
第5.7节：使模块缓存失效	57
第6章：node.js中的导出与导入模块	58
第6.1节：使用ES6语法导出	58
第6.2节：在node.js中使用简单模块	58
第7章：模块的加载方式	59
第7.1节：全局模式	59
第7.2节：加载模块	59
第8章：集群模块	60
第8.1节：Hello World	60
第8.2节：集群示例	60
第9章：Readline	62
第9.1节：逐行读取文件	62
第9.2节：通过命令行界面提示用户输入	62
第10章：package.json	63
第10.1节：探索package.json	63
第10.2节：脚本	66
第10.3节：基本项目定义	67
第10.4节：依赖项	67
第10.5节：扩展项目定义	68
第11章：事件发射器	69
第11.1节：基础知识	69
第11.2节：获取已订阅事件的名称	69
第11.3节：通过事件发射器进行HTTP分析	70
第11.4节：获取注册监听特定事件的监听器数量	70
第12章：变更时自动重载	72
第12.1节：使用nodemon进行源代码变更自动重载	72
第12.2节：Browsersync	72
第13章：环境	74

Section 4.1: Asynchronously Read from Files	45
Section 4.2: Listing Directory Contents with readdir or readdirSync	45
Section 4.3: Copying files by piping streams	46
Section 4.4: Reading from a file synchronously	47
Section 4.5: Check Permissions of a File or Directory	47
Section 4.6: Checking if a file or a directory exists	48
Section 4.7: Determining the line count of a text file	49
Section 4.8: Reading a file line by line	49
Section 4.9: Avoiding race conditions when creating or using an existing directory	49
Section 4.10: Cloning a file using streams	50
Section 4.11: Writing to a file using writeFile or writeFileSync	51
Section 4.12: Changing contents of a text file	51
Section 4.13: Deleting a file using unlink or unlinkSync	52
Section 4.14: Reading a file into a Buffer using streams	52
Chapter 5: Exporting and Consuming Modules	53
Section 5.1: Creating a hello-world.js module	53
Section 5.2: Loading and using a module	54
Section 5.3: Folder as a module	55
Section 5.4: Every module injected only once	55
Section 5.5: Module loading from node_modules	56
Section 5.6: Building your own modules	56
Section 5.7: Invalidating the module cache	57
Chapter 6: Exporting and Importing Module in node.js	58
Section 6.1: Exporting with ES6 syntax	58
Section 6.2: Using a simple module in node.js	58
Chapter 7: How modules are loaded	59
Section 7.1: Global Mode	59
Section 7.2: Loading modules	59
Chapter 8: Cluster Module	60
Section 8.1: Hello World	60
Section 8.2: Cluster Example	60
Chapter 9: Readline	62
Section 9.1: Line-by-line file reading	62
Section 9.2: Prompting user input via CLI	62
Chapter 10: package.json	63
Section 10.1: Exploring package.json	63
Section 10.2: Scripts	66
Section 10.3: Basic project definition	67
Section 10.4: Dependencies	67
Section 10.5: Extended project definition	68
Chapter 11: Event Emitters	69
Section 11.1: Basics	69
Section 11.2: Get the names of the events that are subscribed to	69
Section 11.3: HTTP Analytics through an Event Emitter	70
Section 11.4: Get the number of listeners registered to listen for a specific event	70
Chapter 12: Autoreload on changes	72
Section 12.1: Autoreload on source code changes using nodemon	72
Section 12.2: Browsersync	72
Chapter 13: Environment	74

第13.1节：访问环境变量	74
第13.2节：process.argv命令行参数	74
第13.3节：从“属性文件”加载环境属性	75
第13.4节：为不同环境（如开发、测试、预发布等）使用不同的属性/配置	75
第14章：从回调到Promise	77
第14.1节：将回调转换为Promise	77
第14.2节：手动将回调转换为Promise	77
第14.3节：setTimeout的Promise化	78
第15章：使用子进程执行文件或命令	79
第15.1节：生成新进程以执行命令	79
第15.2节：生成shell以执行命令	79
第15.3节：生成进程以运行可执行文件	80
第16章：异常处理	82
第16.1节：Node.js中的异常处理	82
第16.2节：未处理异常管理	83
第16.3节：错误与Promise	84
第17章：保持Node应用程序持续运行	86
第17.1节：使用PM2作为进程管理器	86
第17.2节：运行和停止Forever守护进程	87
第17.3节：使用nohup持续运行	88
第18章：卸载Node.js	89
第18.1节：在Mac OSX上完全卸载Node.js	89
第18.2节：在Windows上卸载Node.js	89
第19章：nvm - Node版本管理器	90
第19.1节：安装NVM	90
第19.2节：检查NVM版本	90
第19.3节：安装特定的Node版本	90
第19.4节：使用已安装的Node版本	90
第19.5节：在Mac OSX上安装nvm	91
第19.6节：在子shell中使用所需的Node版本运行任意命令	91
第19.7节：为Node版本设置别名	92
第20章：http	93
第20.1节：http服务器	93
第20.2节：http客户端	94
第21章：使用流	95
第21.1节：使用流从文本文件读取数据	95
第21.2节：流的管道传输	95
第21.3节：创建你自己的可读/可写流	96
第21.4节：为什么使用流？	97
第22章：在生产环境中部署Node.js应用	99
第22.1节：设置NODE_ENV="production"	99
第22.2节：使用进程管理器管理应用	100
第22.3节：使用进程管理器进行部署	100
第22.4节：使用PM2进行部署	101
第22.5节：针对不同环境（如开发、测试、预发布等）使用不同的属性/配置	102
第22.6节：利用集群优势	103
第23章：保护Node.js应用程序安全	104

Section 13.1: Accessing environment variables	74
Section 13.2: process.argv command line arguments	74
Section 13.3: Loading environment properties from a "property file"	75
Section 13.4: Using different Properties/Configuration for different environments like dev, qa, staging etc	75
Chapter 14: Callback to Promise	77
Section 14.1: Promisifying a callback	77
Section 14.2: Manually promisifying a callback	77
Section 14.3: setTimeout promisified	78
Chapter 15: Executing files or commands with Child Processes	79
Section 15.1: Spawning a new process to execute a command	79
Section 15.2: Spawning a shell to execute a command	79
Section 15.3: Spawning a process to run an executable	80
Chapter 16: Exception handling	82
Section 16.1: Handling Exception In Node.Js	82
Section 16.2: Unhandled Exception Management	83
Section 16.3: Errors and Promises	84
Chapter 17: Keep a node application constantly running	86
Section 17.1: Use PM2 as a process manager	86
Section 17.2: Running and stopping a Forever daemon	87
Section 17.3: Continuous running with nohup	88
Chapter 18: Uninstalling Node.js	89
Section 18.1: Completely uninstall Node.js on Mac OSX	89
Section 18.2: Uninstall Node.js on Windows	89
Chapter 19: nvm - Node Version Manager	90
Section 19.1: Install NVM	90
Section 19.2: Check NVM version	90
Section 19.3: Installing an specific Node version	90
Section 19.4: Using an already installed node version	90
Section 19.5: Install nvm on Mac OSX	91
Section 19.6: Run any arbitrary command in a subshell with the desired version of node	91
Section 19.7: Setting alias for node version	92
Chapter 20: http	93
Section 20.1: http server	93
Section 20.2: http client	94
Chapter 21: Using Streams	95
Section 21.1: Read Data from TextFile with Streams	95
Section 21.2: Piping streams	95
Section 21.3: Creating your own readable/writable stream	96
Section 21.4: Why Streams?	97
Chapter 22: Deploying Node.js applications in production	99
Section 22.1: Setting NODE_ENV="production"	99
Section 22.2: Manage app with process manager	100
Section 22.3: Deployment using process manager	100
Section 22.4: Deployment using PM2	101
Section 22.5: Using different Properties/Configuration for different environments like dev, qa, staging etc	102
Section 22.6: Taking advantage of clusters	103
Chapter 23: Securing Node.js applications	104

第23.1节：Node.js中的SSL/TLS	104
第23.2节：防止跨站请求伪造（CSRF）	104
第23.3节：设置HTTPS服务器	105
第23.4节：使用HTTPS	107
第23.5节：保护express.js 3应用程序	107
第24章：Mongoose库	109
第24.1节：使用Mongoose连接MongoDB	109
第24.2节：使用Mongoose、Express.js路由和\$text操作符在MongoDB中查找数据	109
第24.3节：使用Mongoose和Express.js路由保存数据到MongoDB	111
第24.4节：使用Mongoose和Express.js路由查找MongoDB中的数据	113
第24.5节：有用的Mongoose函数	115
第24.6节：模型中的索引	115
第24.7节：使用Promise在MongoDB中查找数据	117
第25章：async.js	120
第25.1节：并行：多任务处理	120
第25.2节：async.each（高效处理数据数组）	121
第25.3节：串行：独立单任务处理	122
第25.4节：瀑布流：依赖的单任务处理	123
第25.5节：async.times（更好地处理for循环）	124
第25.6节：async.series（逐个处理事件）	124
第26章：文件上传	125
第26.1节：使用multer进行单文件上传	125
第26.2节：使用formidable模块	126
第27章：Socket.io通信	128
第27.1节：使用socket消息的“Hello world!”	128
第28章：Mongodb集成	129
第28.1节：简单连接	129
第28.2节：使用Promise的简单连接	129
第28.3节：连接到MongoDB	129
第28.4节：插入文档	130
第28.5节：读取集合	131
第28.6节：更新文档	131
第28.7节：删除文档	132
第28.8节：删除多个文档	132
第29章：在Node.js中处理POST请求	134
第29.1节：仅处理POST请求的示例node.js服务器	134
第30章：基于简单REST的CRUD API	135
第30.1节：Express 3+中的CRUD REST API	135
第31章：模板框架	136
第31.1节：Nunjucks	136
第32章：Node.js架构与内部工作原理	138
第32.1节：Node.js - 内部机制	138
第32.2节：Node.js - 实践应用	138
第33章：调试Node.js应用程序	139
第33.1节：核心Node.js调试器和Node Inspector	139
第34章：无框架的Node服务器	142
第34.1节：无框架的Node服务器	142
第34.2节：解决跨域资源共享（CORS）问题	143

Section 23.1: SSL/TLS in Node.js	104
Section 23.2: Preventing Cross Site Request Forgery (CSRF)	104
Section 23.3: Setting up an HTTPS server	105
Section 23.4: Using HTTPS	107
Section 23.5: Secure express.js 3 Application	107
Chapter 24: Mongoose Library	109
Section 24.1: Connect to MongoDB Using Mongoose	109
Section 24.2: Find Data in MongoDB Using Mongoose, Express.js Routes and \$text Operator	109
Section 24.3: Save Data to MongoDB using Mongoose and Express.js Routes	111
Section 24.4: Find Data in MongoDB Using Mongoose and Express.js Routes	113
Section 24.5: Useful Mongoose functions	115
Section 24.6: Indexes in models	115
Section 24.7: find data in mongodb using promises	117
Chapter 25: async.js	120
Section 25.1: Parallel : multi-tasking	120
Section 25.2: async.each(To handle array of data efficiently)	121
Section 25.3: Series : independent mono-tasking	122
Section 25.4: Waterfall : dependent mono-tasking	123
Section 25.5: async.times(To handle for loop in better way)	124
Section 25.6: async.series(To handle events one by one)	124
Chapter 26: File upload	125
Section 26.1: Single File Upload using multer	125
Section 26.2: Using formidable module	126
Chapter 27: Socket.io communication	128
Section 27.1: "Hello world!" with socket messages	128
Chapter 28: Mongodb integration	129
Section 28.1: Simple connect	129
Section 28.2: Simple connect, using promises	129
Section 28.3: Connect to MongoDB	129
Section 28.4: Insert a document	130
Section 28.5: Read a collection	131
Section 28.6: Update a document	131
Section 28.7: Delete a document	132
Section 28.8: Delete multiple documents	132
Chapter 29: Handling POST request in Node.js	134
Section 29.1: Sample node.js server that just handles POST requests	134
Chapter 30: Simple REST based CRUD API	135
Section 30.1: REST API for CRUD in Express 3+	135
Chapter 31: Template frameworks	136
Section 31.1: Nunjucks	136
Chapter 32: Node.js Architecture & Inner Workings	138
Section 32.1: Node.js - under the hood	138
Section 32.2: Node.js - in motion	138
Chapter 33: Debugging Node.js application	139
Section 33.1: Core node.js debugger and node inspector	139
Chapter 34: Node server without framework	142
Section 34.1: Framework-less node server	142
Section 34.2: Overcoming CORS Issues	143

第35章：Node.JS与ES6	144
第35.1节：Node ES6支持及使用Babel创建项目	144
第35.2节：在你的NodeJS应用中使用JS ES6	145
第36章：与控制台交互	148
第36.1节：日志记录	148
第37章：Cassandra集成	150
第37.1节：你好，世界	150
第38章：使用Node.js创建API	151
第38.1节：使用Express的GET接口	151
第38.2节：使用Express的POST接口	151
第39章：优雅关闭	153
第39.1节：优雅关闭 - SIGTERM	153
第40章：使用IISNode在IIS中托管Node.js Web应用	154
第40.1节：通过 <appSettings> 使用 IIS 虚拟目录或嵌套应用程序	154
第40.2节：入门指南	155
第40.3节：使用 Express 的基本 Hello World 示例	155
第40.4节：在 IISNode 中使用 Socket.io	157
第41章：命令行界面（CLI）	158
第41.1节：命令行选项	158
第42章：NodeJS框架	161
第42.1节：Web服务器框架	161
第42.2节：命令行界面框架	161
第43章：grunt	163
第43.1节：GruntJs简介	163
第43.2节：安装grunt插件	164
第44章：在Node.JS中使用WebSocket	165
第44.1节：安装WebSocket	165
第44.2节：向文件中添加WebSocket	165
第44.3节：使用WebSocket和WebSocket服务器	165
第44.4节：一个简单的WebSocket服务器示例	165
第45章：金属工匠	166
第45.1节：构建一个简单的博客	166
{{ title }}	166
第46章：解析命令行参数	167
第46.1节：传递动作（动词）和值	167
第46.2节：传递布尔开关	167
第47章：客户端-服务器通信	168
第47.1节：使用Express、jQuery和Jade	168
第48章：Node.js设计基础	170
第48.1节：Node.js理念	170
第49章：连接MongoDB	171
第49.1节：从Node.js连接MongoDB的简单示例	171
第49.2节：使用核心Node.js连接MongoDB的简单方法	171
第50章：性能挑战	172
第50.1节：使用Node处理长时间运行的查询	172
第51章：发送网页通知	176
第51.1节：使用GCM（谷歌云消息系统）发送网页通知	176

Chapter 35: Node.JS with ES6	144
Section 35.1: Node ES6 Support and creating a project with Babel	144
Section 35.2: Use JS es6 on your NodeJS app	145
Chapter 36: Interacting with Console	148
Section 36.1: Logging	148
Chapter 37: Cassandra Integration	150
Section 37.1: Hello world	150
Chapter 38: Creating API's with Node.js	151
Section 38.1: GET api using Express	151
Section 38.2: POST api using Express	151
Chapter 39: Graceful Shutdown	153
Section 39.1: Graceful Shutdown - SIGTERM	153
Chapter 40: Using IISNode to host Node.js Web Apps in IIS	154
Section 40.1: Using an IIS Virtual Directory or Nested Application via <appSettings>	154
Section 40.2: Getting Started	155
Section 40.3: Basic Hello World Example using Express	155
Section 40.4: Using Socket.io with IISNode	157
Chapter 41: CLI	158
Section 41.1: Command Line Options	158
Chapter 42: NodeJS Frameworks	161
Section 42.1: Web Server Frameworks	161
Section 42.2: Command Line Interface Frameworks	161
Chapter 43: grunt	163
Section 43.1: Introduction To GruntJs	163
Section 43.2: Installing gruntplugins	164
Chapter 44: Using WebSocket's with Node.JS	165
Section 44.1: Installing WebSocket's	165
Section 44.2: Adding WebSocket's to your file's	165
Section 44.3: Using WebSocket's and WebSocket Server's	165
Section 44.4: A Simple WebSocket Server Example	165
Chapter 45: metalsmith	166
Section 45.1: Build a simple blog	166
{{ title }}	166
Chapter 46: Parsing command line arguments	167
Section 46.1: Passing action (verb) and values	167
Section 46.2: Passing boolean switches	167
Chapter 47: Client-server communication	168
Section 47.1: /w Express, jQuery and Jade	168
Chapter 48: Node.js Design Fundamental	170
Section 48.1: The Node.js philosophy	170
Chapter 49: Connect to Mongodbs	171
Section 49.1: Simple example to Connect mongoDB from Node.JS	171
Section 49.2: Simple way to Connect mongoDB with core Node.JS	171
Chapter 50: Performance challenges	172
Section 50.1: Processing long running queries with Node	172
Chapter 51: Send Web Notification	176
Section 51.1: Send Web notification using GCM (Google Cloud Messaging System)	176

第52章：Node.JS远程调试	178	Chapter 52: Remote Debugging in Node.JS	178
第52.1节：在Linux上通过端口使用代理进行调试	178	Section 52.1: Use the proxy for debugging via port on Linux	178
第52.2节：NodeJS运行配置	178	Section 52.2: NodeJS run configuration	178
第52.3节：IntelliJ/Webstorm配置	178	Section 52.3: IntelliJ/Webstorm Configuration	178
第53章：数据库（使用Mongoose的MongoDB）	180	Chapter 53: Database (MongoDB with Mongoose)	180
第53.1节：Mongoose连接	180	Section 53.1: Mongoose connection	180
第53.2节：模型	180	Section 53.2: Model	180
第53.3节：插入数据	181	Section 53.3: Insert data	181
第53.4节：读取数据	181	Section 53.4: Read data	181
第54章：良好的编码风格	183	Chapter 54: Good coding style	183
第54.1节：注册的基本程序	183	Section 54.1: Basic program for signup	183
第55章：Restful API设计：最佳实践	187	Chapter 55: Restful API Design: Best Practices	187
第55.1节：错误处理：获取所有资源	187	Section 55.1: Error Handling: GET all resources	187
第56章：传送HTML或其他类型的文件	189	Chapter 56: Deliver HTML or any other sort of file	189
第56.1节：在指定路径传送HTML	189	Section 56.1: Deliver HTML at specified path	189
第57章：TCP套接字	190	Chapter 57: TCP Sockets	190
第57.1节：一个简单的TCP服务器	190	Section 57.1: A simple TCP server	190
第57.2节：一个简单的TCP客户端	190	Section 57.2: A simple TCP client	190
第58章：Hack	192	Chapter 58: Hack	192
第58.1节：为require()添加新扩展	192	Section 58.1: Add new extensions to require()	192
第59章：Bluebird承诺	193	Chapter 59: Bluebird Promises	193
第59.1节：将nodeback库转换为Promises	193	Section 59.1: Converting nodeback library to Promises	193
第59.2节：功能性Promise	193	Section 59.2: Functional Promises	193
第59.3节：协程（生成器）	193	Section 59.3: Coroutines (Generators)	193
第59.4节：自动资源释放（Promise.using）	193	Section 59.4: Automatic Resource Disposal (Promise.using)	193
第59.5节：串行执行	194	Section 59.5: Executing in series	194
第60章：Async/Await	195	Chapter 60: Async/Await	195
第60.1节：Promise与Async/Await的比较	195	Section 60.1: Comparison between Promises and Async/Await	195
第60.2节：带有Try-Catch错误处理的异步函数	195	Section 60.2: Async Functions with Try-Catch Error Handling	195
第60.3节：在await处停止执行	196	Section 60.3: Stops execution at await	196
第60.4节：从回调函数的进展	196	Section 60.4: Progression from Callbacks	196
第61章：Koa框架v2	198	Chapter 61: Koa Framework v2	198
第61.1节：Hello World示例	198	Section 61.1: Hello World example	198
第61.2节：使用中间件处理错误	198	Section 61.2: Handling errors using middleware	198
第62章：单元测试框架	199	Chapter 62: Unit testing frameworks	199
第62.1节：Mocha异步（async/await）	199	Section 62.1: Mocha Asynchronous (async/await)	199
第62.2节：Mocha同步	199	Section 62.2: Mocha synchronous	199
第62.3节：Mocha 异步（回调）	199	Section 62.3: Mocha asynchronous (callback)	199
第63章：使用 Node.js 的 ECMAScript 2015 (ES6)	200	Chapter 63: ECMAScript 2015 (ES6) with Node.js	200
第63.1节：const/let 声明	200	Section 63.1: const/let declarations	200
第63.2节：箭头函数	200	Section 63.2: Arrow functions	200
第63.3节：箭头函数示例	200	Section 63.3: Arrow Function Example	200
第63.4节：解构赋值	201	Section 63.4: destructuring	201
第63.5节：流程	201	Section 63.5: flow	201
第63.6节：ES6类	201	Section 63.6: ES6 Class	201
第64章：使用Express.JS路由AJAX请求	203	Chapter 64: Routing AJAX requests with Express.JS	203
第64.1节：AJAX的简单实现	203	Section 64.1: A simple implementation of AJAX	203
第65章：向客户端发送文件流	205	Chapter 65: Sending a file stream to client	205

第65.1节：使用fs和pipe从服务器流式传输静态文件	205
第65.2节：使用fluent-mpeg进行流式传输	206
第66章：NodeJS与Redis	207
第66.1节：入门	207
第66.2节：存储键值对	207
第66.3节：node_redis支持的一些重要操作	209
第67章：使用Browserify解决浏览器中的'required'错误	211
第67.1节：示例 - file.js	211
第68章：Node.JS与MongoDB。	213
第68.1节：连接数据库	213
第68.2节：创建新集合	213
第68.3节：插入文档	214
第68.4节：读取	214
第68.5节：更新	215
第68.6节：删除	216
第69章：护照集成	218
第69.1节：本地认证	218
第69.2节：入门指南	219
第69.3节：Facebook认证	220
第69.4节：简单的用户名-密码认证	221
第69.5节：谷歌护照认证	221
第70章：依赖注入	224
第70.1节：为什么使用依赖注入	224
第71章：NodeJS初学者指南	225
第71.1节：你好，世界！	225
第72章：Node.js的使用案例	226
第72.1节：HTTP服务器	226
第72.2节：带命令提示符的控制台	226
第73章：Sequelize.js	228
第73.1节：定义模型	228
第73.2节：安装	229
第74章：PostgreSQL集成	230
第74.1节：连接到PostgreSQL	230
第74.2节：使用连接对象查询	230
第75章：MySQL集成	231
第75.1节：连接到MySQL	231
第75.2节：使用连接池	231
第75.3节：使用参数查询连接对象	232
第75.4节：不使用参数查询连接对象	233
第75.5节：使用连接池中的单个连接执行多条查询	233
第75.6节：导出连接池	233
第75.7节：发生错误时返回查询	234
第76章：MySQL连接池	235
第76.1节：无数据库时使用连接池	235
第77章：MSSQL集成	236
第77.1节：通过mssql npm模块连接SQL	236
第78章：Node.js与Oracle	238
第78.1节：连接到Oracle数据库	238

Section 65.1: Using fs And pipe To Stream Static Files From The Server	205
Section 65.2: Streaming Using fluent-ffmpeg	206
Chapter 66: NodeJS with Redis	207
Section 66.1: Getting Started	207
Section 66.2: Storing Key-Value Pairs	207
Section 66.3: Some more important operations supported by node_redis	209
Chapter 67: Using Browserify to resolve 'required' error with browsers	211
Section 67.1: Example - file.js	211
Chapter 68: Node.JS and MongoDB.	213
Section 68.1: Connecting To a Database	213
Section 68.2: Creating New Collection	213
Section 68.3: Inserting Documents	214
Section 68.4: Reading	214
Section 68.5: Updating	215
Section 68.6: Deleting	216
Chapter 69: Passport integration	218
Section 69.1: Local authentication	218
Section 69.2: Getting started	219
Section 69.3: Facebook authentication	220
Section 69.4: Simple Username-Password Authentication	221
Section 69.5: Google Passport authentication	221
Chapter 70: Dependency Injection	224
Section 70.1: Why Use Dependency Injection	224
Chapter 71: NodeJS Beginner Guide	225
Section 71.1: Hello World!	225
Chapter 72: Use Cases of Node.js	226
Section 72.1: HTTP server	226
Section 72.2: Console with command prompt	226
Chapter 73: Sequelize.js	228
Section 73.1: Defining Models	228
Section 73.2: Installation	229
Chapter 74: PostgreSQL integration	230
Section 74.1: Connect To PostgreSQL	230
Section 74.2: Query with Connection Object	230
Chapter 75: MySQL integration	231
Section 75.1: Connect to MySQL	231
Section 75.2: Using a connection pool	231
Section 75.3: Query a connection object with parameters	232
Section 75.4: Query a connection object without parameters	233
Section 75.5: Run a number of queries with a single connection from a pool	233
Section 75.6: Export Connection Pool	233
Section 75.7: Return the query when an error occurs	234
Chapter 76: MySQL Connection Pool	235
Section 76.1: Using a connection pool without database	235
Chapter 77: MSSQL Intergration	236
Section 77.1: Connecting with SQL via. mssql npm module	236
Chapter 78: Node.js with Oracle	238
Section 78.1: Connect to Oracle DB	238

第78.2节：使用本地模块以简化查询	238
第78.3节：查询无参数的连接对象	239
第79章：Node.js中的同步与异步编程	241
第79.1节：使用async	241
第80章：Node.js错误管理	242
第80.1节：try...catch块	242
第80.2节：创建错误对象	242
第80.3节：抛出错误	243
第81章：Node.js v6 新特性与改进	244
第81.1节：默认函数参数	244
第81.2节：剩余参数	244
第81.3节：箭头函数	244
第81.4节：箭头函数中的“this”	245
第81.5节：扩展运算符	246
第82章：事件循环	247
第82.1节：事件循环概念的发展	247
第83章：Node.js 历史	249
第83.1节：每年的关键事件	249
第84章：passport.js	252
第84.1节：passport.js中LocalStrategy的示例	252
第85章：异步编程	253
第85.1节：回调函数	253
第85.2节：回调地狱	255
第85.3节：原生Promise	256
第85.4节：代码示例	257
第85.5节：异步错误处理	258
第86章：Node.js中不使用任何库的标准输入输出代码	259
第86.1节：程序	259
第87章：Node.js/Express.js的MongoDB集成	260
第87.1节：安装MongoDB	260
第87.2节：创建Mongoose模型	260
第87.3节：查询您的Mongo数据库	261
第88章：Lodash	262
第88.1节：过滤集合	262
第89章：Node.js中的CSV解析器	263
第89.1节：使用FS读取CSV	263
第90章：Loopback - 基于REST的连接器	264
第90.1节：添加基于网页的连接器	264
第91章：将 node.js 作为服务运行	266
第91.1节：将 Node.js 作为 systemd 守护进程	266
第92章：Node.js 与跨域资源共享（CORS）	267
第92.1节：在 express.js 中启用 CORS	267
第93章：Node 性能分析入门	268
第93.1节：分析一个简单的 Node 应用程序	268
第94章：Node.js性能	270
第94.1节：启用gzip	270
第94.2节：事件循环	270
第94.3节：增加最大套接字数	271

Section 78.2: Using a local module for easier querying	238
Section 78.3: Query a connection object without parameters	239
Chapter 79: Synchronous vs Asynchronous programming in nodejs	241
Section 79.1: Using async	241
Chapter 80: Node.js Error Management	242
Section 80.1: try...catch block	242
Section 80.2: Creating Error object	242
Section 80.3: Throwing Error	243
Chapter 81: Node.js v6 New Features and Improvement	244
Section 81.1: Default Function Parameters	244
Section 81.2: Rest Parameters	244
Section 81.3: Arrow Functions	244
Section 81.4: "this" in Arrow Function	245
Section 81.5: Spread Operator	246
Chapter 82: Eventloop	247
Section 82.1: How the concept of event loop evolved	247
Chapter 83: Nodejs History	249
Section 83.1: Key events in each year	249
Chapter 84: passport.js	252
Section 84.1: Example of LocalStrategy in passport.js	252
Chapter 85: Asynchronous programming	253
Section 85.1: Callback functions	253
Section 85.2: Callback hell	255
Section 85.3: Native Promises	256
Section 85.4: Code example	257
Section 85.5: Async error handling	258
Chapter 86: Node.js code for STDIN and STDOUT without using any library	259
Section 86.1: Program	259
Chapter 87: MongoDB Integration for Node.js/Express.js	260
Section 87.1: Installing MongoDB	260
Section 87.2: Creating a Mongoose Model	260
Section 87.3: Querying your Mongo Database	261
Chapter 88: Lodash	262
Section 88.1: Filter a collection	262
Chapter 89: csv parser in node js	263
Section 89.1: Using FS to read in a CSV	263
Chapter 90: Loopback - REST Based connector	264
Section 90.1: Adding a web based connector	264
Chapter 91: Running node.js as a service	266
Section 91.1: Node.js as a systemd dæmon	266
Chapter 92: Node.js with CORS	267
Section 92.1: Enable CORS in express.js	267
Chapter 93: Getting started with Nodes profiling	268
Section 93.1: Profiling a simple node application	268
Chapter 94: Node.js Performance	270
Section 94.1: Enable gzip	270
Section 94.2: Event Loop	270
Section 94.3: Increase maxSockets	271

第95章：Yarn包管理器	273
第95.1节：创建基础包	273
第95.2节：Yarn安装	273
第95.3节：使用Yarn安装包	275
第96章：OAuth 2.0	276
第96.1节：使用Redis实现OAuth 2 - grant type : password	276
第97章：Node.js本地化	282
第97.1节：使用i18n模块维护Node.js应用的本地化	282
第98章：无停机时间部署Node.js应用	283
第98.1节：使用PM2无停机时间部署	283
第99章：Node.js (express.js) 与angular.js示例代码	285
第99.1节：创建我们的项目	285
第100章：NodeJs 路由	288
第100.1节：Express 网络服务器路由	288
第101章：创建一个同时支持 Promise 和错误优先回调的 Node.js 库	292
第101.1节：使用 Bluebird 的示例模块及对应程序	292
第102章：项目结构	295
第102.1节：一个简单的带有MVC和API的Node.js应用	295
第103章：避免回调地狱	297
第103.1节：异步模块	297
第103.2节：异步模块	297
第104章：Arduino与Node.js通信	299
第104.1节：通过serialport实现Node.js与Arduino通信	299
第105章：N-API	301
第105.1节：N-API简介	301
第106章：多线程	303
第106.1节：集群	303
第106.2节：子进程	303
第107章：node.js下的Windows身份验证	305
第107.1节：使用activedirectory	305
第108章：Require()	306
第108.1节：开始使用带有函数和文件的require()	306
第108.2节：开始使用带有NPM包的require()	307
第109章：ExpressJS的路由-控制器-服务结构	308
第109.1节：模型-路由-控制器-服务目录结构	308
第109.2节：模型-路由-控制器-服务代码结构	308
第110章：推送通知	310
第110.1节：网页通知	310
第110.2节：苹果	311
附录A：安装Node.js	312
第A.1节：使用Node版本管理器 (nvm)	312
第A.2节：使用包管理器在Mac上安装Node.js	313
第A.3节：在Windows上安装Node.js	313
第A.4节：在Ubuntu上安装Node.js	314
第A.5节：使用n安装Node.js	314
第A.6节：使用APT包管理器从源码安装Node.js	315

Chapter 95: Yarn Package Manager	273
Section 95.1: Creating a basic package	273
Section 95.2: Yarn Installation	273
Section 95.3: Install package with Yarn	275
Chapter 96: OAuth 2.0	276
Section 96.1: OAuth 2 with Redis Implementation - grant_type: password	276
Chapter 97: Node JS Localization	282
Section 97.1: using i18n module to maintains localization in node js app	282
Chapter 98: Deploying Node.js application without downtime.	283
Section 98.1: Deployment using PM2 without downtime	283
Chapter 99: Node.js (express.js) with angular.js Sample code	285
Section 99.1: Creating our project	285
Chapter 100: NodeJs Routing	288
Section 100.1: Express Web Server Routing	288
Chapter 101: Creating a Node.js Library that Supports Both Promises and Error-First Callbacks	292
Section 101.1: Example Module and Corresponding Program using Bluebird	292
Chapter 102: Project Structure	295
Section 102.1: A simple nodejs application with MVC and API	295
Chapter 103: Avoid callback hell	297
Section 103.1: Async module	297
Section 103.2: Async Module	297
Chapter 104: Arduino communication with nodeJs	299
Section 104.1: Node Js communication with Arduino via serialport	299
Chapter 105: N-API	301
Section 105.1: Hello to N-API	301
Chapter 106: Multithreading	303
Section 106.1: Cluster	303
Section 106.2: Child Process	303
Chapter 107: Windows authentication under node.js	305
Section 107.1: Using activedirectory	305
Chapter 108: Require()	306
Section 108.1: Beginning require() use with a function and file	306
Section 108.2: Beginning require() use with an NPM package	307
Chapter 109: Route-Controller-Service structure for ExpressJS	308
Section 109.1: Model-Routes-Controllers-Services Directory Structure	308
Section 109.2: Model-Routes-Controllers-Services Code Structure	308
Chapter 110: Push notifications	310
Section 110.1: Web notification	310
Section 110.2: Apple	311
Appendix A: Installing Node.js	312
Section A.1: Using Node Version Manager (nvm)	312
Section A.2: Installing Node.js on Mac using package manager	313
Section A.3: Installing Node.js on Windows	313
Section A.4: Install Node.js on Ubuntu	314
Section A.5: Installing Node.js with n	314
Section A.6: Install Node.js From Source with APT package manager	315

第A.7节：在CentOS、RHEL和Fedora上从源码安装Node.js	315
第A.8节：在Fish Shell下使用Oh My Fish!安装Node版本管理器	316
第A.9节：在树莓派上安装Node.js	316
学分	318
你可能也喜欢	323

Section A.7: Install Node.js from source on Centos, RHEL and Fedora	315
Section A.8: Installing with Node Version Manager under Fish Shell with Oh My Fish!	316
Section A.9: Installing Node.js on Raspberry PI	316
Credits	318
You may also like	323

belindoc.com

请随意免费分享此 PDF,
本书的最新版本可从以下网址下载：

<https://goalkicker.com/NodeJSBook>

这本 *Node.js* 专业人士笔记是从 [Stack Overflow Documentation](#) 汇编而成，内容由 Stack Overflow 的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费书籍，旨在教育用途，且与官方 Node.js 组织或公司及 Stack Overflow 无关。所有商标和注册商标均为其各自公司所有者所有。

本书中提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至 web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
<https://goalkicker.com/NodeJSBook>

This *Node.js Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow.
Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Node.js group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：Node.js入门

版本发布日期

v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02
v7.7.0	2017-02-28
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
v7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.22	2017-04-04
v6.10.12	2017-03-21
v6.10.02	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26
v6.4.0	2016-08-12
v6.3.1	2016-07-21

Chapter 1: Getting started with Node.js

Version Release Date

v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02
v7.7.0	2017-02-28
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
v7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.2	2017-04-04
v6.10.1	2017-03-21
v6.10.0	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26
v6.4.0	2016-08-12
v6.3.1	2016-07-21

v6.3.0 2016-07-06
v6.2.2 2016-06-16
v6.2.1 2016-06-02
v6.2.0 2016-05-17
v6.1.0 2016-05-05
v6.0.0 2016-04-26
v5.12.0 2016-06-23
v5.11.1 2016-05-05
v5.11.0 2016-04-21
v5.10.1 2016-04-05
v5.10 2016-04-01
v5.9 2016-03-16
v5.8 2016-03-09
v5.7 2016-02-23
v5.6 2016-02-09
v5.5 2016-01-21
v5.4 2016-01-06
v5.3 2015-12-15
v5.2 2015-12-09
v5.1 2015-11-17
v5.0 2015-10-29
v4.4 2016-03-08
v4.3 2016-02-09
v4.2 2015-10-12
v4.1 2015-09-17
v4.0 2015-09-08
io.js v3.3 2015-09-02
io.js v3.2 2015-08-25
io.js v3.1 2015-08-19
io.js v3.0 2015-08-04
io.js v2.5 2015-07-28
io.js v2.4 2015-07-17
io.js v2.3 2015-06-13
io.js v2.2 2015-06-01
io.js v2.1 2015-05-24
io.js v2.0 2015-05-04
io.js v1.8 2015-04-21
io.js v1.7 2015-04-17
io.js v1.6 2015-03-20
io.js v1.5 2015-03-06
io.js v1.4 2015-02-27
io.js v1.3 2015-02-20
io.js v1.2 2015-02-11
io.js v1.1 2015-02-03
io.js v1.0 2015-01-14
v0.12 2016-02-09

v6.3.0 2016-07-06
v6.2.2 2016-06-16
v6.2.1 2016-06-02
v6.2.0 2016-05-17
v6.1.0 2016-05-05
v6.0.0 2016-04-26
v5.12.0 2016-06-23
v5.11.1 2016-05-05
v5.11.0 2016-04-21
v5.10.1 2016-04-05
v5.10 2016-04-01
v5.9 2016-03-16
v5.8 2016-03-09
v5.7 2016-02-23
v5.6 2016-02-09
v5.5 2016-01-21
v5.4 2016-01-06
v5.3 2015-12-15
v5.2 2015-12-09
v5.1 2015-11-17
v5.0 2015-10-29
v4.4 2016-03-08
v4.3 2016-02-09
v4.2 2015-10-12
v4.1 2015-09-17
v4.0 2015-09-08
io.js v3.3 2015-09-02
io.js v3.2 2015-08-25
io.js v3.1 2015-08-19
io.js v3.0 2015-08-04
io.js v2.5 2015-07-28
io.js v2.4 2015-07-17
io.js v2.3 2015-06-13
io.js v2.2 2015-06-01
io.js v2.1 2015-05-24
io.js v2.0 2015-05-04
io.js v1.8 2015-04-21
io.js v1.7 2015-04-17
io.js v1.6 2015-03-20
io.js v1.5 2015-03-06
io.js v1.4 2015-02-27
io.js v1.3 2015-02-20
io.js v1.2 2015-02-11
io.js v1.1 2015-02-03
io.js v1.0 2015-01-14
v0.12 2016-02-09

v0.11 2013-03-28
v0.10 2013-03-11
v0.9 2012-07-20
v0.8 2012-06-22
v0.7 2012-01-17
v0.6 2011-11-04
v0.5 2011-08-26
v0.4 2011-08-26
v0.3 2011-08-26
v0.2 2011-08-26
v0.1 2011-08-26

v0.11 2013-03-28
v0.10 2013-03-11
v0.9 2012-07-20
v0.8 2012-06-22
v0.7 2012-01-17
v0.6 2011-11-04
v0.5 2011-08-26
v0.4 2011-08-26
v0.3 2011-08-26
v0.2 2011-08-26
v0.1 2011-08-26

第1.1节：Hello World HTTP服务器

首先，为您的平台安装 Node.js。

在此示例中，我们将创建一个监听 1337 端口的 HTTP 服务器，向浏览器发送Hello, World!。注意，您可以使用任何当前未被其他服务占用的端口号，而不仅限于端口 1337。

http模块是 Node.js 的核心模块（Node.js 源码中包含的模块，无需安装额外资源）。http模块提供了使用 `http.createServer()`方法创建 HTTP 服务器的功能。要创建该应用程序，请创建一个包含以下 JavaScript 代码的文件。

```
const http = require('http'); // 加载 http 模块

http.createServer((request, response) => {

    // 1. 告诉浏览器一切正常（状态码 200），且数据为纯文本格式
    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });

    // 2. 将声明的文本写入页面主体
    response.write('Hello, World!');

    // 3. 告诉服务器所有响应头和主体内容已发送完毕
    response.end();

}).listen(1337); // 4. 告诉服务器监听哪个端口
```

将文件保存为任意文件名。在本例中，如果我们将其命名为hello.js，可以通过进入文件所在目录并使用以下命令来运行该应用程序：

```
node hello.js
```

然后可以通过浏览器访问创建的服务器，网址为<http://localhost:1337>或<http://127.0.0.1:1337>。

一个简单的网页将显示，顶部有“Hello, World!”文本，如下图所示。

Section 1.1: Hello World HTTP server

First, install Node.js for your platform.

In this example we'll create an HTTP server listening on port 1337, which sends Hello, World! to the browser. Note that, instead of using port 1337, you can use any port number of your choice which is currently not in use by any other service.

The `http` module is a Node.js **core module** (a module included in Node.js's source, that does not require installing additional resources). The `http` module provides the functionality to create an HTTP server using the `http.createServer()` method. To create the application, create a file containing the following JavaScript code.

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

    // 1. Tell the browser everything is OK (Status code 200), and the data is in plain text
    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });

    // 2. Write the announced text to the body of the page
    response.write('Hello, World!\n');

    // 3. Tell the server that all of the response headers and body have been sent
    response.end();

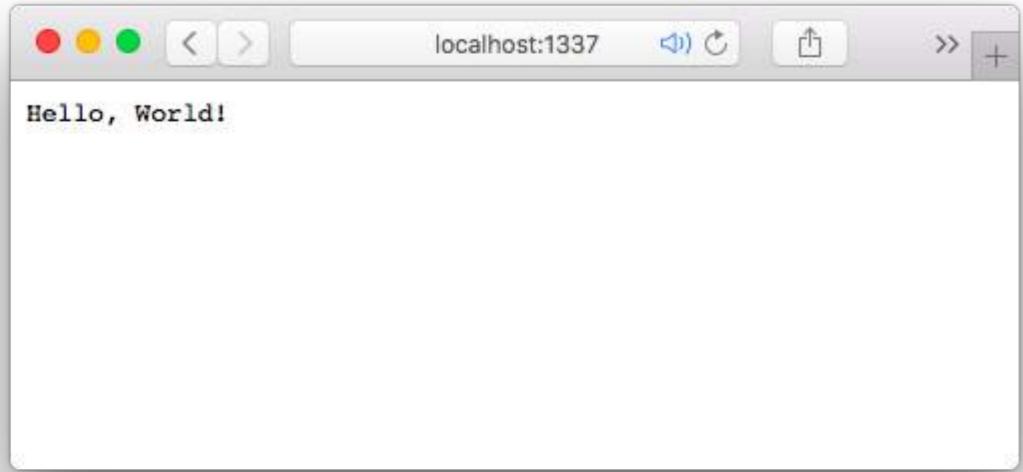
}).listen(1337); // 4. Tells the server what port to be on
```

Save the file with any file name. In this case, if we name it `hello.js` we can run the application by going to the directory the file is in and using the following command:

```
node hello.js
```

The created server can then be accessed with the URL <http://localhost:1337> or <http://127.0.0.1:1337> in the browser.

A simple web page will appear with a “Hello, World!” text at the top, as shown in the screenshot below.



[可在线编辑示例。](#)

第1.2节：Hello World命令行

Node.js也可以用来创建命令行工具。下面的示例读取命令行中的第一个参数并打印一条Hello消息。

在Unix系统上运行此代码：

1. 创建一个新文件并粘贴以下代码。文件名无关紧要。
2. 使该文件可执行，命令为 `chmod 700 FILE_NAME`
3. 使用 `./APP_NAME David` 运行应用程序

在 Windows 上执行第 1 步，然后用 `node APP_NAME David` 运行

```
#!/usr/bin/env node

'use strict';

/*
命令行参数存储在 `process.argv` 数组中,
其结构如下:
[0] 启动 Node.js 进程的可执行文件路径
[1] 此应用程序的路径
[2-n] 命令行参数

示例: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
来源: https://nodejs.org/api/process.html#process\_process\_argv
*/
// 将第一个参数存储为用户名。
var username = process.argv[2];

// 检查是否未提供用户名。
if (!username) {

  // 提取文件名
  var appName = process.argv[1].split(require('path').sep).pop();
```

[Editable online example.](#)

Section 1.2: Hello World command line

Node.js can also be used to create command line utilities. The example below reads the first argument from the command line and prints a Hello message.

To run this code on an Unix System:

1. Create a new file and paste the code below. The filename is irrelevant.
2. Make this file executable with `chmod 700 FILE_NAME`
3. Run the app with `./APP_NAME David`

On Windows you do step 1 and run it with `node APP_NAME David`

```
#!/usr/bin/env node

'use strict';

/*
The command line arguments are stored in the `process.argv` array,
which has the following structure:
[0] The path of the executable that started the Node.js process
[1] The path to this application
[2-n] the command line arguments

Example: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
src: https://nodejs.org/api/process.html#process\_process\_argv
*/
// Store the first argument as username.
var username = process.argv[2];

// Check if the username hasn't been provided.
if (!username) {

  // Extract the filename
  var appName = process.argv[1].split(require('path').sep).pop();
```

```

// 给用户一个如何使用该应用的示例。
console.error('缺少参数！示例：%s YOUR_NAME', appName);

// 退出应用程序 (成功: 0, 错误: 1)。
// 错误会停止执行链。例如：
// ./app.js && ls      -> 不会执行 ls
// ./app.js David && ls -> 会执行 ls
process.exit(1);
}

// 将消息打印到控制台。
console.log('Hello %s!', username);

```

第1.3节：使用Express的Hello World

下面的示例使用Express创建一个监听3000端口的HTTP服务器，响应内容为“Hello, World!”。Express是一个常用的Web框架，适合创建HTTP API。

首先，创建一个新文件夹，例如myApp。进入myApp并新建一个JavaScript文件，内容如下（例如命名为hello.js）。然后通过命令行使用npm install --save express安装express模块。有关如何安装包的更多信息，请参阅此文档。

```

// 导入express的顶层函数
const express = require('express');

// 使用顶层函数创建Express应用
const app = express();

// 定义端口号为3000
const port = 3000;

// 将HTTP GET请求路由到指定路径"/"并使用指定的回调函数
app.get('/', function(request, response) {
  response.send('Hello, World!');
});

// 让应用监听3000端口
app.listen(port, function() {
  console.log('服务器正在监听 http://localhost:' + port);
});

```

从命令行运行以下命令：

```
node hello.js
```

打开浏览器并访问 <http://localhost:3000> 或 <http://127.0.0.1:3000> 查看响应。

有关Express框架的更多信息，您可以查看“使用Express的Web应用”部分

第1.4节：安装和运行Node.js

首先，在您的开发电脑上安装 Node.js。

Windows：访问[下载页面](#)并下载/运行安装程序。

Mac：访问[下载页面](#)并下载/运行安装程序。或者，您也可以通过Homebrew 使用 brew install node 来安装 Node。Homebrew 是 Mac 的命令行包管理器，更多

```

// Give the user an example on how to use the app.
console.error('Missing argument! Example: %s YOUR_NAME', appName);

// Exit the app (success: 0, error: 1).
// An error will stop the execution chain. For example:
// ./app.js && ls      -> won't execute ls
// ./app.js David && ls -> will execute ls
process.exit(1);
}

// Print the message to the console.
console.log('Hello %s!', username);

```

Section 1.3: Hello World with Express

The following example uses Express to create an HTTP server listening on port 3000, which responds with "Hello, World!". Express is a commonly-used web framework that is useful for creating HTTP APIs.

First, create a new folder, e.g. myApp. Go into myApp and make a new JavaScript file containing the following code (let's name it `hello.js` for example). Then install the express module using `npm install --save express` from the command line. Refer to this documentation for more information on how to install packages.

```

// Import the top-level function of express
const express = require('express');

// Creates an Express application using the top-level function
const app = express();

// Define port number as 3000
const port = 3000;

// Routes HTTP GET requests to the specified path "/" with the specified callback function
app.get('/', function(request, response) {
  response.send('Hello, World!');
});

// Make the app listen on port 3000
app.listen(port, function() {
  console.log('Server listening on http://localhost:' + port);
});

```

From the command line, run the following command:

```
node hello.js
```

Open your browser and navigate to <http://localhost:3000> or <http://127.0.0.1:3000> to see the response.

For more information about the Express framework, you can check the Web Apps With Express section

Section 1.4: Installing and Running Node.js

To begin, install Node.js on your development computer.

Windows: Navigate to the [download page](#) and download/run the installer.

Mac: Navigate to the [download page](#) and download/run the installer. Alternatively, you can install Node via Homebrew using `brew install node`. Homebrew is a command-line package manager for Macintosh, and more

信息可以在 Homebrew 网站 上找到。

Linux：按照您发行版的说明，在 命令行安装页面 上操作。

运行 Node 程序

要运行 Node.js 程序，只需运行 `node app.js` 或 `nodejs app.js`，其中 `app.js` 是您的 Node 应用源代码文件名。Node 不需要包含 `.js` 后缀即可找到您想运行的脚本。

或者，在基于 UNIX 的操作系统下，Node 程序可以作为终端脚本执行。为此，脚本需要以指向 Node 解释器的 shebang 开头，例如 `#!/usr/bin/env node`。文件还必须设置为可执行，可以使用 `chmod` 完成。现在脚本可以直接从命令行运行。

第 1.5 节：调试您的 NodeJS 应用程序

您可以使用 `node-inspector`。运行此命令通过 `npm` 安装它：

```
npm install -g node-inspector
```

然后你可以使用以下命令调试你的应用程序

```
node-debug app.js
```

Github 仓库地址：<https://github.com/node-inspector/node-inspector>

本地调试

你也可以通过以下方式本地调试 `node.js`：

```
node debug your-script.js
```

要在你想要的代码行准确设置断点，请使用：

```
debugger;
```

更多信息请见 [here](#)。

在 `node.js 8` 中使用以下命令：

```
node --inspect-brk 你的-脚本.js
```

然后在较新版本的谷歌浏览器中打开 `about://inspect`，选择你的 Node 脚本，以获得 Chrome 开发者工具的调试体验。

第1.6节：Hello World 基础路由

一旦你了解了如何用 `node` 创建 HTTP 服务器，理解如何根据用户访问的路径让服务器“执行”操作就很重要了。这种现象称为“路由”。

最基本的例子是检查 `if (request.url === 'some/path/here')`，然后调用一个返回新文件的函数。

information about it can be found on the [Homebrew website](#).

Linux: Follow the instructions for your distro on the [command line installation page](#).

Running a Node Program

To run a Node.js program, simply run `node app.js` or `nodejs app.js`, where `app.js` is the filename of your node app source code. You do not need to include the `.js` suffix for Node to find the script you'd like to run.

Alternatively under UNIX-based operating systems, a Node program may be executed as a terminal script. To do so, it needs to begin with a shebang pointing to the Node interpreter, such as `#!/usr/bin/env node`. The file also has to be set as executable, which can be done using `chmod`. Now the script can be directly run from the command line.

Section 1.5: Debugging Your NodeJS Application

You can use the `node-inspector`. Run this command to install it via `npm`:

```
npm install -g node-inspector
```

Then you can debug your application using

```
node-debug app.js
```

The Github repository can be found here: <https://github.com/node-inspector/node-inspector>

Debugging natively

You can also debug `node.js` natively by starting it like this:

```
node debug your-script.js
```

To breakpoint your debugger exactly in a code line you want, use this:

```
debugger;
```

For more information see [here](#).

In `node.js 8` use the following command:

```
node --inspect-brk your-script.js
```

Then open `about://inspect` in a recent version of Google Chrome and select your Node script to get the debugging experience of Chrome's DevTools.

Section 1.6: Hello World basic routing

Once you understand how to create an HTTP Server with `node`, it's important to understand how to make it "do" things based on the path that a user has navigated to. This phenomenon is called, "routing".

The most basic example of this would be to check `if (request.url === 'some/path/here')`, and then call a function that responds with a new file.

这里可以看到一个示例：

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

function index (request, response) {
  response.writeHead(200);
  response.end('Hello, World!');
}

http.createServer(function (request, response) {

  if (request.url === '/') {
    return index(request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);
}).listen(1337);
```

不过，如果你继续这样定义“路由”，最终会得到一个庞大的回调函数，我们不想要那样一团糟，所以让我们看看能否整理一下。

首先，让我们把所有路由存储在一个对象中：

```
var routes = {
  '/': function index (request, response) {
    response.writeHead(200);
    response.end('Hello, World!');
  },
  '/foo': function foo (request, response) {
    response.writeHead(200);
    response.end('You are now viewing "foo"');
  }
}
```

现在我们已经把两个路由存储在对象中，可以在主回调中检查它们：

```
http.createServer(function (request, response) {

  if (request.url in routes) {
    return routes[request.url](request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);
}).listen(1337);
```

现在，每次你尝试浏览你的网站时，它都会检查该路径是否存在于你的路由中，并调用相应的函数。如果未找到路由，服务器将返回404（未找到）响应。

这就是使用 HTTP 服务器 API 进行路由的简单方法。

第1.7节：REPL中的Hello World

当不带参数调用时，Node.js 会启动一个 REPL（读取-求值-打印-循环），也称为“Node shell”。

An example of this can be seen here:

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

function index (request, response) {
  response.writeHead(200);
  response.end('Hello, World!');
}

http.createServer(function (request, response) {

  if (request.url === '/') {
    return index(request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);
}).listen(1337);
```

If you continue to define your "routes" like this, though, you'll end up with one massive callback function, and we don't want a giant mess like that, so let's see if we can clean this up.

First, let's store all of our routes in an object:

```
var routes = {
  '/': function index (request, response) {
    response.writeHead(200);
    response.end('Hello, World!');
  },
  '/foo': function foo (request, response) {
    response.writeHead(200);
    response.end('You are now viewing "foo"');
  }
}
```

Now that we've stored 2 routes in an object, we can now check for them in our main callback:

```
http.createServer(function (request, response) {

  if (request.url in routes) {
    return routes[request.url](request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);
}).listen(1337);
```

Now every time you try to navigate your website, it will check for the existence of that path in your routes, and it will call the respective function. If no route is found, the server will respond with a 404 (Not Found).

And there you have it--routing with the HTTP Server API is very simple.

Section 1.7: Hello World in the REPL

When called without arguments, Node.js starts a REPL (Read-Eval-Print-Loop) also known as the “Node shell”.

在命令提示符下输入node。

```
$ node  
>
```

在 Node shell 提示符>下输入 "Hello World!"

```
$ node  
> "Hello World!"  
'Hello World!'
```

第1.8节：在线部署你的应用

当你将应用部署到（Node.js专用的）托管环境时，该环境通常会提供一个PORT-环境变量，你可以使用它来运行服务器。将端口号更改为process.env.PORT即可访问该应用。

例如，

```
http.createServer(function(request, response) {  
    // 你的服务器代码  
}).listen(process.env.PORT);
```

另外，如果你想在调试时离线访问，可以使用这个：

```
http.createServer(function(request, response) {  
    // 你的服务器代码  
}).listen(process.env.PORT || 3000);
```

其中 3000 是离线端口号。

第1.9节：核心模块

Node.js 是一个 Javascript 引擎（谷歌 Chrome 的 V8 引擎，使用 C++ 编写），允许在浏览器外运行 Javascript。虽然有许多库可用于扩展 Node 的功能，但该引擎自带一套实现基本功能的核心模块。

Node 当前包含34个核心模块：

```
[ 'assert',  
  'buffer',  
  'c/c++_addons',  
  'child_process',  
  'cluster',  
  'console',  
  'crypto',  
  'deprecated_apis',  
  'dns',  
  'domain',  
  'Events',  
  'fs',  
  'http',  
  'https',  
  'module',  
  'net',  
  'os',  
  'path',
```

At a command prompt type node.

```
$ node  
>
```

At the Node shell prompt > type "Hello World!"

```
$ node  
> "Hello World!"  
'Hello World!'
```

Section 1.8: Deploying your application online

When you deploy your app to a (Node.js-specific) hosted environment, this environment usually offers a PORT-environment variable that you can use to run your server on. Changing the port number to process.env.PORT allows you to access the application.

For example,

```
http.createServer(function(request, response) {  
    // your server code  
}).listen(process.env.PORT);
```

Also, if you would like to access this offline while debugging, you can use this:

```
http.createServer(function(request, response) {  
    // your server code  
}).listen(process.env.PORT || 3000);
```

where 3000 is the offline port number.

Section 1.9: Core modules

Node.js is a Javascript engine (Google's V8 engine for Chrome, written in C++) that allows to run Javascript outside the browser. While numerous libraries are available for extending Node's functionalities, the engine comes with a set of *core modules* implementing basic functionalities.

There's currently 34 core modules included in Node:

```
[ 'assert',  
  'buffer',  
  'c/c++_addons',  
  'child_process',  
  'cluster',  
  'console',  
  'crypto',  
  'deprecated_apis',  
  'dns',  
  'domain',  
  'Events',  
  'fs',  
  'http',  
  'https',  
  'module',  
  'net',  
  'os',  
  'path',
```

```
'punycode',
'querystring',
'readline',
'repl',
'stream',
'string_decoder',
'timers',
'tls_(ssl)',
'tracing',
'tty',
'dgram',
'url',
'util',
'v8',
'vm',
'zlib' ]
```

此列表来源于 Node 文档 API <https://nodejs.org/api/all.html> (JSON 文件：
<https://nodejs.org/api/all.json>)。

所有核心模块一览

assert

assert 模块提供了一组简单的断言测试，可用于测试不变量。

buffer

在 ECMAScript 2015 (ES6) 引入 `TypedArray` 之前，JavaScript 语言没有读取或操作二进制数据流的机制。Buffer 类作为 Node.js API 的一部分被引入，使得在处理如 TCP 流和文件系统操作等场景中能够与八位字节流进行交互成为可能。

现在 ES6 中已添加了 `TypedArray`，Buffer 类以更优化且适合 Node.js 使用场景的方式实现了

```
Uin  
t8Array
```

APIs。

c/c++_addons

Node.js 插件是用 C 或 C++ 编写的动态链接共享对象，可以通过 `require()` 函数加载到 Node.js 中，并像普通的 Node.js 模块一样使用。它们主要用于提供运行在 Node.js 中的 JavaScript 与 C/C++ 库之间的接口。

child_process

`child_process` 模块提供了以类似但不完全相同于 `popen(3)` 的方式生成子进程的能力。

集群

Node.js 的单个实例运行在单线程中。为了利用多核系统，用户有时会希望启动一个 Node.js 进程集群来处理负载。`cluster` 模块允许你轻松创建共享服务器端口的子进程。

```
'punycode',
'querystring',
'readline',
'repl',
'stream',
'string_decoder',
'timers',
'tls_(ssl)',
'tracing',
'tty',
'dgram',
'url',
'util',
'v8',
'vm',
'zlib' ]
```

This list was obtained from the Node documentation API <https://nodejs.org/api/all.html> (JSON file: <https://nodejs.org/api/all.json>).

All core modules at-a-glance

assert

The assert module provides a simple set of assertion tests that can be used to test invariants.

buffer

Prior to the introduction of `TypedArray` in ECMAScript 2015 (ES6), the JavaScript language had no mechanism for reading or manipulating streams of binary data. The Buffer class was introduced as part of the Node.js API to make it possible to interact with octet streams in the context of things like TCP streams and file system operations.

Now that `TypedArray` has been added in ES6, the Buffer class implements the

```
Uin  
t8Array
```

API in a manner that is more optimized and suitable for Node.js' use cases.

c/c++_addons

Node.js Addons are dynamically-linked shared objects, written in C or C++, that can be loaded into Node.js using the `require()` function, and used just as if they were an ordinary Node.js module. They are used primarily to provide an interface between JavaScript running in Node.js and C/C++ libraries.

child_process

The `child_process` module provides the ability to spawn child processes in a manner that is similar, but not identical, to `popen(3)`.

cluster

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node.js processes to handle the load. The cluster module allows you to easily create child processes that all share server ports.

控制台

console 模块提供了一个简单的调试控制台，类似于网页浏览器提供的 JavaScript 控制台机制。

加密

crypto 模块提供加密功能，包括一组针对 OpenSSL 的哈希、HMAC、加密、解密、签名和验证函数的封装。

已废弃的 API

当出现以下情况时，Node.js 可能会废弃某些 API：(a) 该 API 被认为不安全，(b) 已有更好的替代 API 可用，或(c) 预计在未来的重大版本中该 API 会有破坏性更改。

域名 (DNS)

dns 模块包含属于两类的函数：

1. 使用底层操作系统功能执行名称解析的函数，这些函数不一定执行任何网络通信。该类别仅包含一个函数：`dns.lookup()`。
2. 连接到实际的 DNS 服务器执行名称解析，并且始终使用网络执行 DNS 查询。该类别包含 dns 模块中除 `dns.lookup()` 以外的所有函数。

域

该模块即将被弃用。一旦替代 API 确定，该模块将被完全弃用。大多数终端用户不需要使用该模块。必须使用域功能的用户目前可以依赖它，但应预期将来需要迁移到其他解决方案。

事件

Node.js 核心 API 很大程度上基于惯用的异步事件驱动架构，其中某些类型的对象（称为“发射器”）周期性地发出命名事件，导致函数对象（“监听器”）被调用。

fs

文件 I/O 由对标准 POSIX 函数的简单封装提供。使用此模块请执行 `require('fs')`。所有方法均有异步和同步两种形式。

http

Node.js 中的 HTTP 接口设计用于支持协议中许多传统上难以使用的功能。特别是大型的、可能是分块编码的消息。该接口小心地从不缓冲整个请求或响应——用户能够进行数据流式传输。

https

console

The `console` module provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

crypto

The `crypto` module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions.

deprecated_apis

Node.js may deprecate APIs when either: (a) use of the API is considered to be unsafe, (b) an improved alternative API has been made available, or (c) breaking changes to the API are expected in a future major release.

dns

The dns module contains functions belonging to two different categories:

1. Functions that use the underlying operating system facilities to perform name resolution, and that do not necessarily perform any network communication. This category contains only one function: `dns.lookup()`.
2. Functions that connect to an actual DNS server to perform name resolution, and that *always* use the network to perform DNS queries. This category contains all functions in the dns module except `dns.lookup()`.

domain

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most end users should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

Events

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called.

fs

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

http

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses--the user is able to stream data.

https

HTTPS 是基于 TLS/SSL 的 HTTP 协议。在 Node.js 中，这作为一个独立的模块实现。

模块

Node.js 有一个简单的模块加载系统。在 Node.js 中，文件和模块是一一对应的（每个文件被视为一个独立的模块）。

net

net 模块为你提供了一个异步网络封装。它包含用于创建服务器和客户端（称为流）的函数。你可以通过 `require('net')` 来引入该模块。

os

os 模块提供了许多与操作系统相关的实用方法。

路径

path 模块提供用于处理文件和目录路径的实用工具。

punycode

Node.js 中捆绑的 punycode 模块版本正在被弃用。

querystring

querystring 模块提供用于解析和格式化 URL 查询字符串的实用工具。

readline

readline 模块提供一个接口，用于从可读流（例如 `process.stdin`）一次读取一行数据。

repl

repl 模块提供一个读-求值-打印循环（REPL）实现，可作为独立程序使用，也可包含在其他应用程序中。

流

流是 Node.js 中用于处理流数据的抽象接口。stream 模块提供了一个基础 API，使得构建实现流接口的对象变得简单。

Node.js 提供了许多流对象。例如，HTTP 服务器的请求和 `process.stdout` 都是流实例。

string_decoder

string_decoder 模块提供了一个 API，用于以保留编码的多字节 UTF-8 和 UTF-16 字符的方式，将 Buffer 对象解码为字符串。

timers

HTTPS is the HTTP protocol over TLS/SSL. In Node.js this is implemented as a separate module.

module

Node.js has a simple module loading system. In Node.js, files and modules are in one-to-one correspondence (each file is treated as a separate module).

net

The net module provides you with an asynchronous network wrapper. It contains functions for creating both servers and clients (called streams). You can include this module with `require('net')`.

os

The os module provides a number of operating system-related utility methods.

path

The path module provides utilities for working with file and directory paths.

punycode

The version of the punycode module bundled in Node.js is being deprecated.

querystring

The querystring module provides utilities for parsing and formatting URL query strings.

readline

The readline module provides an interface for reading data from a Readable stream (such as `process.stdin`) one line at a time.

repl

The repl module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includable in other applications.

stream

A stream is an abstract interface for working with streaming data in Node.js. The stream module provides a base API that makes it easy to build objects that implement the stream interface.

There are many stream objects provided by Node.js. For instance, a request to an HTTP server and `process.stdout` are both stream instances.

string_decoder

The string_decoder module provides an API for decoding Buffer objects into strings in a manner that preserves encoded multi-byte UTF-8 and UTF-16 characters.

timers

timer 模块公开了一个全局 API，用于安排函数在未来某个时间段被调用。由于计时器函数是全局的，因此使用该 API 无需调用 `require('timers')`。

Node.js 中的计时器函数实现了与 Web 浏览器提供的计时器 API 类似的接口，但使用了基于 Node.js 事件循环的不同内部实现。

tls_(ssl)

`tls` 模块提供了基于 OpenSSL 构建的传输层安全 (TLS) 和安全套接字层 (SSL) 协议的实现。

追踪

Trace Event 提供了一种机制，用于集中管理由 V8、Node 核心和用户空间代码生成的追踪信息。

可以通过在启动 Node.js 应用时传递 `--trace-events-enabled` 标志来启用追踪功能。

tty

`tty` 模块提供了 `tty.ReadStream` 和 `tty.WriteStream` 类。在大多数情况下，通常不需要或无法直接使用该模块。

dgram

`dgram` 模块提供了 UDP 数据报套接字的实现。

url

`url` 模块提供了用于 URL 解析和解析的实用工具。

util

`util` 模块主要设计用于支持 Node.js 自身内部 API 的需求。然而，许多实用工具对应用程序和模块开发者也非常有用。

v8

`v8` 模块暴露了特定于内置于 Node.js 二进制文件中的 V8 版本的 API。

注意：API 和实现随时可能发生变化。

vm

`vm` 模块提供了在 V8 虚拟机上下文中编译和运行代码的 API。JavaScript 代码可以立即编译并运行，也可以编译后保存，稍后运行。

注意：vm 模块不是安全机制。不要用它来运行不受信任的代码。

zlib

`zlib` 模块提供了使用 Gzip 和 Deflate/Inflate 实现的压缩功能。

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call `require('timers')` to use the API.

The timer functions within Node.js implement a similar API as the timers API provided by Web Browsers but use a different internal implementation that is built around [the Node.js Event Loop](#).

tls_(ssl)

The `tls` module provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL.

tracing

Trace Event provides a mechanism to centralize tracing information generated by V8, Node core, and userspace code.

Tracing can be enabled by passing the `--trace-events-enabled` flag when starting a Node.js application.

tty

The `tty` module provides the `tty.ReadStream` and `tty.WriteStream` classes. In most cases, it will not be necessary or possible to use this module directly.

dgram

The `dgram` module provides an implementation of UDP Datagram sockets.

url

The `url` module provides utilities for URL resolution and parsing.

util

The `util` module is primarily designed to support the needs of Node.js' own internal APIs. However, many of the utilities are useful for application and module developers as well.

v8

The `v8` module exposes APIs that are specific to the version of `V8` built into the Node.js binary.

Note: The APIs and implementation are subject to change at any time.

vm

The `vm` module provides APIs for compiling and running code within V8 Virtual Machine contexts. JavaScript code can be compiled and run immediately or compiled, saved, and run later.

*Note: The vm module is not a security mechanism. **Do not use it to run untrusted code.***

zlib

The `zlib` module provides compression functionality implemented using Gzip and Deflate/Inflate.

第 1.10 节：TLS 套接字：服务器和客户端

这与普通 TCP 连接的唯一主要区别是你需要将私钥和公钥证书设置到一个选项对象中。

如何创建密钥和证书

此安全流程的第一步是创建一个私钥。什么是私钥呢？基本上，它是一组用于加密信息的随机噪声。理论上，你可以创建一个密钥，用它来加密任何你想加密的内容。但最佳做法是为特定用途使用不同的密钥。因为如果有人窃取了你的私钥，就相当于有人偷了你的房门钥匙。想象一下，如果你用同一把钥匙锁车、车库、办公室等。

```
openssl genrsa -out private-key.pem 1024
```

一旦我们有了私钥，就可以创建一个CSR（证书签名请求），这是我们请求权威机构为私钥签名的申请。这就是为什么你需要输入与你公司相关的信息。这些信息将被签名机构查看并用来验证你的身份。在我们的例子中，输入什么内容并不重要，因为下一步我们将自己签署证书。

```
openssl req -new -key private-key.pem -out csr.pem
```

现在我们已经填写好了相关资料，是时候假装自己是一个酷炫的签名机构了。

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-cert.pem
```

现在你拥有了私钥和公钥证书，就可以在两个NodeJS应用之间建立安全连接。如示例代码所示，这个过程非常简单。

重要！

由于我们自己创建了公钥证书，坦白说，我们的证书毫无价值，因为我们不是权威机构。NodeJS服务器默认不会信任这样的证书，这就是为什么我们需要通过以下选项告诉它实际信任我们的证书：rejectUnauthorized: false。非常重要：切勿在生产环境中将此变量设置为true。

TLS 套接字服务器

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem')
};

var server = tls.createServer(options, function(socket) {

  // 发送一条友好的消息
  socket.write("我是服务器，给你发送一条消息。");

  // 打印我们收到的数据
  socket.on('data', function(data) {
```

Section 1.10: TLS Socket: server and client

The only major differences between this and a regular TCP connection are the private Key and the public certificate that you'll have to set into an option object.

How to Create a Key and Certificate

The first step in this security process is the creation of a private Key. And what is this private key? Basically, it's a set of random noise that's used to encrypt information. In theory, you could create one key, and use it to encrypt whatever you want. But it is best practice to have different keys for specific things. Because if someone steals your private key, it's similar to having someone steal your house keys. Imagine if you used the same key to lock your car, garage, office, etc.

```
openssl genrsa -out private-key.pem 1024
```

Once we have our private key, we can create a CSR (certificate signing request), which is our request to have the private key signed by a fancy authority. That is why you have to input information related to your company. This information will be seen by the signing authority, and used to verify you. In our case, it doesn't matter what you type, since in the next step we're going to sign our certificate ourselves.

```
openssl req -new -key private-key.pem -out csr.pem
```

Now that we have our paper work filled out, it's time to pretend that we're a cool signing authority.

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-cert.pem
```

Now that you have the private key and the public cert, you can establish a secure connection between two NodeJS apps. And, as you can see in the example code, it is a very simple process.

Important!

Since we created the public cert ourselves, in all honesty, our certificate is worthless, because we are nobodies. The NodeJS server won't trust such a certificate by default, and that is why we need to tell it to actually trust our cert with the following option rejectUnauthorized: false. **Very important:** never set this variable to true in a production environment.

TLS Socket Server

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem')
};

var server = tls.createServer(options, function(socket) {

  // Send a friendly message
  socket.write("I am the server sending you a message.");

  // Print the data that we received
  socket.on('data', function(data) {
```

```

console.log('接收: %s [长度为 %d 字节]',
           data.toString().replace(/\n/gm, ""),
           data.length);

});

// 当传输结束时通知我们
socket.on('结束', function() {
    console.log('EOT (传输结束)');
});

// 开始监听指定端口和地址
server.listen(PORT, HOST, function() {
    console.log("我在监听 %s, 端口 %s", HOST, PORT);
});

// 发生错误时显示错误信息。
server.on('错误', function(error) {
    console.error(error);

    // 发生错误后关闭连接。
    server.destroy();
});

```

TLS 套接字客户端

```

'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

// 将证书传递给服务器，并让其处理未授权的证书。
var options = {
key: fs.readFileSync('private-key.pem'),
cert: fs.readFileSync('public-cert.pem'),
rejectUnauthorized: false
};

var client = tls.connect(PORT, HOST, options, function() {

// 检查授权是否成功
if (client.authorized) {
    console.log("连接已被证书颁发机构授权。");
} else {
    console.log("连接未授权: " + client.authorizationError)
}

// 发送一条友好的消息
client.write("我是发送消息的客户端。");

});

```

```

console.log('Received: %s [it is %d bytes long]',
           data.toString().replace(/\n/gm, ""),
           data.length);

});

// Let us know when the transmission is over
socket.on('end', function() {
    console.log('EOT (End Of Transmission)');
});

// Start listening on a specific port and address
server.listen(PORT, HOST, function() {
    console.log("I'm listening at %s, on port %s", HOST, PORT);
});

// When an error occurs, show it.
server.on('error', function(error) {
    console.error(error);

    // Close the connection after the error occurred.
    server.destroy();
});

```

TLS Socket Client

```

'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

// Pass the certs to the server and let it know to process even unauthorized certs.
var options = {
key: fs.readFileSync('private-key.pem'),
cert: fs.readFileSync('public-cert.pem'),
rejectUnauthorized: false
};

var client = tls.connect(PORT, HOST, options, function() {

// Check if the authorization worked
if (client.authorized) {
    console.log("Connection authorized by a Certificate Authority.");
} else {
    console.log("Connection not authorized: " + client.authorizationError)
}

// Send a friendly message
client.write("I am the client sending you a message.");

});

```

```

client.on("data", function(data) {
    console.log('接收: %s [长度为 %d 字节]', data.toString().replace(/\n/gm, ""),
    data.length);

    // 接收消息后关闭连接
    client.end();
});

client.on('close', function() {
    console.log("连接已关闭");
});

// 发生错误时显示错误信息。
client.on('error', function(error) {
    console.error(error);

    // 发生错误后关闭连接。
    client.destroy();
});

```

第1.11节：如何搭建并运行一个基本的HTTPS网页服务器！

一旦你在系统上安装了 node.js，就可以按照以下步骤运行一个支持 HTTP 和 HTTPS 的基础网络服务器！

步骤 1：建立证书颁发机构

1. 创建一个用于存放密钥和证书的文件夹：

```
mkdir conf
```

2. 进入该目录：

```
cd conf
```

3. 获取此ca.cnf文件，作为配置快捷方式使用：

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

```

client.on("data", function(data) {
    console.log('Received: %s [it is %d bytes long]',
    data.toString().replace(/\n/gm, ""),
    data.length);

    // Close the connection after receiving the message
    client.end();
});

client.on('close', function() {
    console.log("Connection closed");
});

// When an error occurs, show it.
client.on('error', function(error) {
    console.error(error);

    // Close the connection after the error occurred.
    client.destroy();
});

```

Section 1.11: How to get a basic HTTPS web server up and running!

Once you have node.js installed on your system, you can just follow the procedure below to get a basic web server running with support for both HTTP and HTTPS!

Step 1 : Build a Certificate Authority

1. create the folder where you want to store your key & certificate :

```
mkdir conf
```

2. go to that directory :

```
cd conf
```

3. grab this ca.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. 使用此配置创建一个新的证书颁发机构：

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. 现在我们已经有了证书颁发机构的 ca-key.pem 和 ca-cert.pem，接下来生成服务器的私钥：
服务器：

```
openssl genrsa -out key.pem 4096
```

6. 获取这个 server.cnf 文件作为配置快捷方式：

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. 使用此配置生成证书签名请求：

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. 签署请求：

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

4. create a new certificate authority using this configuration :

```
openssl req -new -x509 -days 999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. now that we have our certificate authority in ca-key.pem and ca-cert.pem, let's generate a private key for the server :

```
openssl genrsa -out key.pem 4096
```

6. grab this server.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. generate the certificate signing request using this configuration :

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. sign the request :

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

步骤2：将您的证书安装为根证书

1. 将您的证书复制到根证书文件夹：

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. 更新CA存储：

```
sudo update-ca-certificates
```

Step 2 : Install your certificate as a root certificate

1. copy your certificate to your root certificates' folder :

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. update CA store :

```
sudo update-ca-certificates
```

步骤3：启动您的节点服务器

Step 3 : Starting your node server

首先，您需要创建一个server.js文件，其中包含您的实际服务器代码。

Node.js中HTTPS服务器的最小配置大致如下：

```
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world");
}

https.createServer(httpsOptions, app).listen(4433);
```

如果你也想支持 http 请求，只需做这个小改动：

```
var http = require('http');
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

1. 进入你的 server.js 所在的目录：

```
cd /path/to
```

2. 运行 server.js：

```
node server.js
```

First, you want to create a `server.js` file that contains your actual server code.

The minimal setup for an HTTPS server in Node.js would be something like this :

```
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

If you also want to support http requests, you need to make just this small modification :

```
var http = require('http');
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

1. go to the directory where your `server.js` is located :

```
cd /path/to
```

2. run `server.js`:

```
node server.js
```

第2章：npm

参数	示例
访问权限	npm publish --access=public
二进制文件	npm bin -g
编辑	npm edit connect
帮助	npm 帮助 初始化
初始化	npm 初始化
安装	npm 安装
链接	npm 链接
修剪	npm 修剪
发布	npm publish ./
重启	npm 重启
启动	npm 启动
停止	npm 启动
更新	npm 更新
版本	npm 版本

节点包管理器（npm）提供以下两个主要功能：用于node.js的在线仓库，包/模块可在search.nodejs.org上搜索。命令行工具用于安装Node.js包，进行版本管理和依赖管理。

第2.1节：安装包

介绍

包是npm用来表示开发者可以用于其项目的工具的术语。这包括从库和框架（如jQuery和AngularJS）到任务运行器（如Gulp.js）的一切。包通常会放在一个名为node_modules的文件夹中，该文件夹还包含一个package.json文件。该文件包含所有包的信息，包括任何依赖项，即使用特定包所需的额外模块。

npm使用命令行来安装和管理包，因此尝试使用npm的用户应熟悉其操作系统上的基本命令，例如：目录遍历以及查看目录内容。

安装NPM

请注意，安装包之前，必须先安装NPM。

推荐的安装NPM方式是使用Node.js下载页面上的安装程序之一。你可以通过运行 `npm -v` 或 `npm version` 命令来检查是否已经安装了node.js。

通过Node.js安装程序安装NPM后，请务必检查更新。因为NPM的更新频率高于Node.js安装程序。要检查更新，请运行以下命令：

```
npm install npm@latest -g
```

如何安装软件包

Chapter 2: npm

Parameter	Example
access	npm publish --access=public
bin	npm bin -g
edit	npm edit connect
help	npm help init
init	npm init
install	npm install
link	npm link
prune	npm prune
publish	npm publish ./
restart	npm restart
start	npm start
stop	npm start
update	npm update
version	npm version

Node Package Manager (npm) provides following two main functionalities: Online repositories for node.js packages/modules which are searchable on search.nodejs.org. Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

Section 2.1: Installing packages

Introduction

Package is a term used by npm to denote tools that developers can use for their projects. This includes everything from libraries and frameworks such as jQuery and AngularJS to task runners such as Gulp.js. The packages will come in a folder typically called node_modules, which will also contain a package.json file. This file contains information regarding all the packages including any dependencies, which are additional modules needed to use a particular package.

Npm uses the command line to both install and manage packages, so users attempting to use npm should be familiar with basic commands on their operating system i.e.: traversing directories as well as being able to see the contents of directories.

Installing NPM

Note that in order to install packages, you must have NPM installed.

The recommended way to install NPM is to use one of the installers from the [Node.js download page](#). You can check to see if you already have node.js installed by running either the `npm -v` or the `npm version` command.

After installing NPM via the Node.js installer, be sure to check for updates. This is because NPM gets updated more frequently than the Node.js installer. To check for updates run the following command:

```
npm install npm@latest -g
```

How to install packages

要安装一个或多个包，请使用以下命令：

```
npm install <package-name>
```

或者

```
npm i <package-name>...
```

```
# 例如，安装 lodash 和 express  
npm install lodash express
```

注意：这将在当前命令行所在的目录中安装包，因此
检查是否选择了合适的目录非常重要

如果当前工作目录已有一个 package.json 文件且其中定义了依赖项，则 npm install 会自动解析并安装文件中列出的所有依赖项。你也可以使用 npm install 命令的简写形式：npm i

如果你想安装某个包的特定版本，请使用：

```
npm install <name>@<version>
```

```
# 例如，安装 lodash 包的 4.11.1 版本  
npm install lodash@4.11.1
```

如果您想安装符合特定版本范围的版本，请使用：

```
npm install <name>@<version range>
```

```
# 例如，安装符合“版本 >= 4.10.1”且“版本 < 4.11.1”的 lodash 包  
npm install lodash@>=4.10.1 <4.11.1"
```

如果您想安装最新版本，请使用：

```
npm install <name>@latest
```

上述命令将在中央 npm 仓库 npmjs.com 中搜索包。如果您不打算从 npm 注册表安装，还支持其他选项，例如：

```
# 以 tarball 形式分发的包  
npm install <tarball file>  
npm install <tarball url>
```

```
# 本地可用的包  
npm install <local path>
```

```
# 以 git 仓库形式可用的包  
npm install <git remote url>
```

```
# GitHub 上可用的包  
npm install <用户名>/<仓库>
```

```
# 作为 gist 可用的软件包（需要 package.json）  
npm install gist:<gist-id>
```

```
# 来自特定仓库的软件包  
npm install --registry=http://myreg.mycompany.com <package name>
```

To install one or more packages use the following:

```
npm install <package-name>
```

or

```
npm i <package-name>...
```

```
# e.g. to install lodash and express  
npm install lodash express
```

Note: This will install the package in the directory that the command line is currently in, thus it is important to check whether the appropriate directory has been chosen

If you already have a package.json file in your current working directory and dependencies are defined in it, then npm install will automatically resolve and install all dependencies listed in the file. You can also use the shorthand version of the npm install command which is: npm i

If you want to install a specific version of a package use:

```
npm install <name>@<version>
```

```
# e.g. to install version 4.11.1 of the package lodash  
npm install lodash@4.11.1
```

If you want to install a version which matches a specific version range use:

```
npm install <name>@<version range>
```

```
# e.g. to install a version which matches "version >= 4.10.1" and "version < 4.11.1"  
# of the package lodash  
npm install lodash@>=4.10.1 <4.11.1"
```

If you want to install the latest version use:

```
npm install <name>@latest
```

The above commands will search for packages in the central npm repository at npmjs.com. If you are not looking to install from the npm registry, other options are supported, such as:

```
# packages distributed as a tarball  
npm install <tarball file>  
npm install <tarball url>
```

```
# packages available locally  
npm install <local path>
```

```
# packages available as a git repository  
npm install <git remote url>
```

```
# packages available on GitHub  
npm install <username>/<repository>
```

```
# packages available as gist (need a package.json)  
npm install gist:<gist-id>
```

```
# packages from a specific repository  
npm install --registry=http://myreg.mycompany.com <package name>
```

```
# 来自相关软件包组的软件包  
# 参见 npm 作用域  
npm install @<scope>/<name>(@<version>)  
  
# 作用域对于通过为特定作用域设置注册表，将托管在私有注册表上的私有软件包与公共软件包分开非常有用  
  
npm config set @mycompany:registry http://myreg.mycompany.com  
npm install @mycompany/<package name>
```

通常，模块会被安装在名为 `node_modules` 的本地文件夹中，该文件夹位于您当前的工作目录中。这个目录是 `require()` 用来加载模块以使其可用的目录。

如果您已经创建了 `package.json` 文件，可以使用 `--save`（简写为 `-S`）选项或其变体之一，自动将安装的软件包添加为 `package.json` 中的依赖项。如果其他人安装您的软件包，`npm` 会自动从 `package.json` 文件中读取依赖项并安装列出的版本。请注意，您仍然可以通过编辑该文件来添加和管理依赖项，因此通常建议跟踪依赖项，例如使用：

```
npm install --save <name> # 安装依赖项  
# 或  
npm install -S <name> # 简写版本 --save  
# 或者  
npm i -S <name>
```

为了安装包并且仅在它们用于开发时保存，而不是用于运行应用程序时保存，请使用以下命令：

```
npm install --save-dev <name> # 安装用于开发的依赖  
# 或者  
npm install -D <name> # 简写版本 --save-dev  
# 或者  
npm i -D <name>
```

安装依赖

有些模块不仅为你提供库供使用，还提供一个或多个二进制文件，旨在通过命令行使用。虽然你仍然可以本地安装这些包，但通常更倾向于全局安装它们，以便启用命令行工具。在这种情况下，`npm` 会自动将二进制文件链接到合适的路径（例如 `/usr/local/bin/<name>`），以便可以从命令行使用。要全局安装包，请使用：

```
npm install --global <name>  
# 或者  
npm install -g <name>  
# 或者  
npm i -g <name>  
  
# 例如，安装 grunt 命令行工具  
npm install -g grunt-cli
```

如果你想查看当前工作区中所有已安装包及其对应版本的列表，请使用：

```
npm list  
npm list <name>
```

添加一个可选的名称参数可以检查特定包的版本。

```
# packages from a related group of packages  
# See npm scope  
npm install @<scope>/<name>(@<version>)  
  
# Scoping is useful for separating private packages hosted on private registry from  
# public ones by setting registry for specific scope  
npm config set @mycompany:registry http://myreg.mycompany.com  
npm install @mycompany/<package name>
```

Usually, modules will be installed locally in a folder named `node_modules`, which can be found in your current working directory. This is the directory `require()` will use to load modules in order to make them available to you.

If you already created a package `.json` file, you can use the `--save` (shorthand `-S`) option or one of its variants to automatically add the installed package to your package `.json` as a dependency. If someone else installs your package, `npm` will automatically read dependencies from the package `.json` file and install the listed versions. Note that you can still add and manage your dependencies by editing the file later, so it's usually a good idea to keep track of dependencies, for example using:

```
npm install --save <name> # Install dependencies  
# or  
npm install -S <name> # shortcut version --save  
# or  
npm i -S <name>
```

In order to install packages and save them only if they are needed for development, not for running them, not if they are needed for the application to run, follow the following command:

```
npm install --save-dev <name> # Install dependencies for development purposes  
# or  
npm install -D <name> # shortcut version --save-dev  
# or  
npm i -D <name>
```

Installing dependencies

Some modules do not only provide a library for you to use, but they also provide one or more binaries which are intended to be used via the command line. Although you can still install those packages locally, it is often preferred to install them globally so the command-line tools can be enabled. In that case, `npm` will automatically link the binaries to appropriate paths (e.g. `/usr/local/bin/<name>`) so they can be used from the command line. To install a package globally, use:

```
npm install --global <name>  
# or  
npm install -g <name>  
# or  
npm i -g <name>  
  
# e.g. to install the grunt command line tool  
npm install -g grunt-cli
```

If you want to see a list of all the installed packages and their associated versions in the current workspace, use:

```
npm list  
npm list <name>
```

Adding an optional name argument can check the version of a specific package.

注意：如果你在尝试全局安装 npm 模块时遇到权限问题，请不要轻易使用 a sudo npm install -g ... 来解决。授予第三方脚本以提升权限在你的系统上运行是危险的。权限问题可能意味着你的 npm 本身的安装方式存在问题。如果你有兴趣在沙箱用户环境中安装 Node，可以尝试使用nvm。

如果你有构建工具或其他仅用于开发的依赖（例如 Grunt），你可能不希望它们被打包到你部署的应用中。如果是这样，你需要将它作为开发依赖，列在 package.json 的 devDependencies 中。要安装仅用于开发的依赖包，请使用 --save-dev (或 -D)。

```
npm install --save-dev <name> // 安装开发依赖，不会包含在生产环境中
```

```
# 或者  
npm install -D <name>
```

你会看到该包随后被添加到你的 package.json 的 devDependencies 中。

要安装已下载/克隆的 node.js 项目的依赖项，您只需使用

```
npm install  
# 或者  
npm i
```

npm 会自动从 package.json 中读取依赖项并安装它们。

代理服务器后的 NPM

如果您的网络访问通过代理服务器，您可能需要修改访问远程仓库的 npm install 命令。npm 使用一个配置文件，可以通过命令行进行更新：

```
npm config set
```

您可以从浏览器的设置面板中找到代理设置。一旦获得代理设置（服务器 URL、端口、用户名和密码）；您需要按如下方式配置 npm 配置。

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>  
$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

username、password、port 字段是可选的。设置完成后，您的 npm install、npm i -g 等命令将能正常工作。

第2.2节：卸载软件包

要卸载一个或多个本地安装的包，请使用：

```
npm uninstall <包名>
```

npm 的卸载命令有五个别名也可以使用：

```
npm remove <包名>  
npm rm <包名>  
npm r <包名>  
  
npm unlink <包名>
```

Note: If you run into permission issues while trying to install an npm module globally, resist the temptation to issue a `sudo npm install -g ...` to overcome the issue. Granting third-party scripts to run on your system with elevated privileges is dangerous. The permission issue might mean that you have an issue with the way npm itself was installed. If you're interested in installing Node in sandboxed user environments, you might want to try using [nvm](#).

If you have build tools, or other development-only dependencies (e.g. Grunt), you might not want to have them bundled with the application you deploy. If that's the case, you'll want to have it as a development dependency, which is listed in the `package.json` under `devDependencies`. To install a package as a development-only dependency, use `--save-dev` (or `-D`).

```
npm install --save-dev <name> // Install development dependencies which is not included in production  
# or  
npm install -D <name>
```

You will see that the package is then added to the `devDependencies` of your `package.json`.

To install dependencies of a downloaded/cloned node.js project, you can simply use

```
npm install  
# or  
npm i
```

npm will automatically read the dependencies from `package.json` and install them.

NPM Behind A Proxy Server

If your internet access is through a proxy server, you might need to modify npm install commands that access remote repositories. npm uses a configuration file which can be updated via command line:

```
npm config set
```

You can locate your proxy settings from your browser's settings panel. Once you have obtained the proxy settings (server URL, port, username and password); you need to configure your npm configurations as follows.

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>  
$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

username, password, port fields are optional. Once you have set these, your npm `install`, `npm i -g` etc. would work properly.

Section 2.2: Uninstalling packages

To uninstall one or more locally installed packages, use:

```
npm uninstall <package name>
```

The `uninstall` command for npm has five aliases that can also be used:

```
npm remove <package name>  
npm rm <package name>  
npm r <package name>  
  
npm unlink <package name>
```

```
npm un <包名>
```

如果您想在卸载时将包从package.json文件中移除，请使用--save标志
(简写：-S)：

```
npm uninstall --save <包名>  
npm uninstall -S <包名>
```

对于开发依赖，请使用--save-dev标志 (简写：-D)：

```
npm uninstall --save-dev <包名>  
npm uninstall -D <包名>
```

对于可选依赖，请使用--save-optional标志 (简写：-O)：

```
npm uninstall --save-optional <包名>  
npm uninstall -O <包名>
```

对于全局安装的包，请使用 --global 标志 (简写：-g)：

```
npm uninstall -g <包名>
```

第2.3节：设置包配置

Node.js 包配置包含在名为 package.json 的文件中，该文件位于每个项目的根目录。你可以通过调用以下命令来设置一个全新的配置文件：

```
npm 初始化
```

该命令会尝试读取当前工作目录中的 Git 仓库信息（如果存在）和环境变量，以尝试为你自动完成一些占位符的值。否则，它会提供一个输入对话框用于基本选项。

如果你想使用默认值创建一个 package.json 文件，可以使用：

```
npm init --yes  
# 或者  
npm init -y
```

如果你正在为一个不会作为 npm 包发布的项目创建 package.json（即仅用于整理依赖项），你可以在你的 package.json 文件中表达这一意图：

1. 可选择将private属性设置为true，以防止意外发布。
2. 可选择将license属性设置为“UNLICENSED”，以拒绝他人使用您的包的权利。

要安装一个包并自动保存到您的package.json中，请使用：

```
npm install --save <package>
```

该包及相关元数据（如包版本）将出现在您的依赖项中。如果您将其保存为开发依赖（使用--save-dev），该包将出现在您的devDependencies中。

使用这个简易的package.json时，安装或升级包时会遇到警告信息，提示您缺少描述和仓库字段。虽然可以安全地忽略这些信息，您

```
npm un <package name>
```

If you would like to remove the package from the package.json file as part of the uninstallation, use the --save flag (shorthand: -S):

```
npm uninstall --save <package name>  
npm uninstall -S <package name>
```

For a development dependency, use the --save-dev flag (shorthand: -D):

```
npm uninstall --save-dev <package name>  
npm uninstall -D <package name>
```

For an optional dependency, use the --save-optional flag (shorthand: -O):

```
npm uninstall --save-optional <package name>  
npm uninstall -O <package name>
```

For packages that are installed globally use the --global flag (shorthand: -g):

```
npm uninstall -g <package name>
```

Section 2.3: Setting up a package configuration

Node.js package configurations are contained in a file called package.json that you can find at the root of each project. You can setup a brand new configuration file by calling:

```
npm init
```

That will try to read the current working directory for Git repository information (if it exists) and environment variables to try and autocomplete some of the placeholder values for you. Otherwise, it will provide an input dialog for the basic options.

If you'd like to create a package.json with default values use:

```
npm init --yes  
# or  
npm init -y
```

If you're creating a package.json for a project that you are not going to be publishing as an npm package (i.e. solely for the purpose of rounding up your dependencies), you can convey this intent in your package.json file:

1. Optionally set the private property to true to prevent accidental publishing.
2. Optionally set the license property to "UNLICENSED" to deny others the right to use your package.

To install a package and automatically save it to your package.json, use:

```
npm install --save <package>
```

The package and associated metadata (such as the package version) will appear in your dependencies. If you save it as a development dependency (using --save-dev), the package will instead appear in your devDependencies.

With this bare-bones package.json, you will encounter warning messages when installing or upgrading packages, telling you that you are missing a description and the repository field. While it is safe to ignore these messages, you

可以通过在任何文本编辑器中打开package.json，并向JSON对象添加以下行来消除它们：

```
[...]
"description": "无描述",
"repository": {
  "private": true
},
[...]
```

第2.4节：运行脚本

您可以在您的package.json中定义脚本，例如：

```
{
  "name": "your-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "echo": "echo hello!"
  }
}
```

要运行 echo 脚本，请在命令行中运行 `npm run echo`。任意脚本，例如上面的 echo，必须通过 `npm run <脚本名>` 来运行。npm 还有一些官方脚本，会在包的生命周期的特定阶段运行（例如 `preinstall`）。完整的 npm 脚本字段处理概览请参见 [here](#)。

npm 脚本最常用于启动服务器、构建项目和运行测试。这里有一个更实际的例子：

```
"scripts": {
  "test": "mocha tests",
  "start": "pm2 start index.js"
}
```

在 scripts 条目中，像 mocha 这样的命令行程序无论是全局安装还是本地安装都能工作。如果命令行入口不存在于系统 PATH 中，npm 也会检查您本地安装的包。

如果您的脚本变得非常长，可以将其拆分成多个部分，像这样：

```
"scripts": {
  "very-complex-command": "npm run chain-1 && npm run chain-2",
  "chain-1": "webpack",
  "chain-2": "node app.js"
}
```

第2.5节：基本语义版本控制

在发布包之前，必须为其设定版本。npm 支持语义版本控制，这意味着有补丁版本、次要版本和主要版本发布。

例如，如果你的包版本是1.2.3，要更改版本需要：

可以通过在任何文本编辑器中打开package.json，并向JSON对象添加以下行来消除它们：

```
[...]
"description": "No description",
"repository": {
  "private": true
},
[...]
```

Section 2.4: Running scripts

You may define scripts in your package.json, for example:

```
{
  "name": "your-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "echo": "echo hello!"
  }
}
```

To run the echo script, run `npm run echo` from the command line. Arbitrary scripts, such as echo above, have to be be run with `npm run <script name>`. npm also has a number of official scripts that it runs at certain stages of the package's life (like `preinstall`). See [here](#) for the entire overview of how npm handles script fields.

Npm scripts are used most often for things like starting a server, building the project, and running tests. Here's a more realistic example:

```
"scripts": {
  "test": "mocha tests",
  "start": "pm2 start index.js"
}
```

In the scripts entries, command-line programs like mocha will work when installed either globally or locally. If the command-line entry does not exist in the system PATH, npm will also check your locally installed packages.

If your scripts become very long, they can be split into parts, like this:

```
"scripts": {
  "very-complex-command": "npm run chain-1 && npm run chain-2",
  "chain-1": "webpack",
  "chain-2": "node app.js"
}
```

Section 2.5: Basic semantic versioning

Before publishing a package you have to version it. npm supports [semantic versioning](#), this means there are **patch**, **minor** and **major** releases.

For example, if your package is at version 1.2.3 to change version you have to:

1. 补丁发布：npm version patch => 1.2.4
2. 次要发布：npm version minor => 1.3.0
3. 主要发布：npm version major => 2.0.0

你也可以直接指定版本：

```
npm 版本 3.1.4 => 3.1.4
```

当你使用上述 npm 命令设置包版本时，npm 会修改 package.json 文件中的 version 字段，提交更改，并且创建一个以“v”为前缀的 Git 标签，就好像你执行了以下命令一样：

```
git tag v3.1.4
```

与 Bower 等其他包管理器不同，npm 注册表不依赖于为每个版本创建 Git 标签。但是，如果你喜欢使用标签，记得在升级包版本后推送新创建的标签：

```
git push origin master (推送 package.json 的更改)
```

```
git push origin v3.1.4 (推送新标签)
```

或者你可以一次性完成：

```
git push origin master --tags
```

第 2.6 节：发布包

首先，确保你已经配置好你的包（如“设置包配置”中所述）。然后，你必须登录到 npmjs。

如果你已经有 npm 用户

npm 登录

如果你还没有用户

```
npm adduser
```

检查你的用户是否已在当前客户端注册

```
npm config ls
```

之后，当你的包准备好发布时，使用

```
npm publish
```

就完成了。

如果你需要发布新版本，确保按照基本语义版本控制更新你的包版本。否则，npm 将不允许你发布该包。

```
{
  name: "package-name",
  version: "1.0.4"
```

1. patch release: npm version **patch** => 1.2.4
2. minor release: npm version **minor** => 1.3.0
3. major release: npm version **major** => 2.0.0

You can also specify a version directly with:

```
npm version 3.1.4 => 3.1.4
```

When you set a package version using one of the npm commands above, npm will modify the version field of the package.json file, commit it, and also create a new Git tag with the version prefixed with a "v", as if you've issued the command:

```
git tag v3.1.4
```

Unlike other package managers like Bower, the npm registry doesn't rely on Git tags being created for every version. But, if you like using tags, you should remember to push the newly created tag after bumping the package version:

```
git push origin master (to push the change to package.json)
```

```
git push origin v3.1.4 (to push the new tag)
```

Or you can do this in one swoop with:

```
git push origin master --tags
```

Section 2.6: Publishing a package

First, make sure that you have configured your package (as said in Setting up a package configuration). Then, you have to be logged in to npmjs.

If you already have a npm user

npm login

If you don't have a user

```
npm adduser
```

To check that your user is registered in the current client

```
npm config ls
```

After that, when your package is ready to be published use

```
npm publish
```

And you are done.

If you need to publish a new version, ensure that you update your package version, as stated in Basic semantic versioning. Otherwise, npm will not let you publish the package.

```
{
  name: "package-name",
  version: "1.0.4"
```

}

第2.7节：移除多余的包

要移除多余的软件包（已安装但不在依赖列表中的软件包），请运行以下命令：

npm 修剪

要移除所有dev软件包，请添加--production标志：

npm prune --production

[更多内容](#)

第2.8节：列出当前已安装的软件包

要生成当前已安装软件包的列表（树状视图），请使用

npm list

ls、**la**和**ll**是**list**命令的别名。**la**和**ll**命令显示扩展信息，如描述和仓库。

选项

可以通过传递选项来更改响应格式。

npm list --json

- **json** - 以 json 格式显示信息
- **long** - 显示扩展信息
- **parseable** - 显示可解析列表而非树形结构
- **global** - 显示全局安装的包
- **depth** - 依赖树的最大显示深度
- **dev/development** - 显示开发依赖 (devDependencies)
- **prod/production** - 显示生产依赖

如果需要，你也可以访问该包的主页。

npm home <包名>

第2.9节：更新 npm 和包

由于 npm 本身是一个 Node.js 模块，因此可以使用它自身进行更新。

如果操作系统是 Windows，必须以管理员身份运行命令提示符

npm install -g npm@latest

如果你想检查更新版本，可以执行：

npm outdated

}

Section 2.7: Removing extraneous packages

To remove extraneous packages (packages that are installed but not in dependency list) run the following command:

npm prune

To remove all dev packages add --production flag:

npm prune --production

[More on it](#)

Section 2.8: Listing currently installed packages

To generate a list (tree view) of currently installed packages, use

npm list

ls, **la** and **ll** are aliases of **list** command. **la** and **ll** commands shows extended information like description and repository.

Options

The response format can be changed by passing options.

npm list --json

- **json** - Shows information in json format
- **long** - Shows extended information
- **parseable** - Shows parseable list instead of tree
- **global** - Shows globally installed packages
- **depth** - Maximum display depth of dependency tree
- **dev/development** - Shows devDependencies
- **prod/production** - Shows dependencies

If you want, you can also go to the package's home page.

npm home <package name>

Section 2.9: Updating npm and packages

Since npm itself is a Node.js module, it can be updated using itself.

If OS is Windows must be running command prompt as Admin

npm install -g npm@latest

If you want to check for updated versions you can do:

npm outdated

为了更新特定的包：

```
npm update <包名>
```

这将根据 package.json 中的限制将包更新到最新版本

如果你还想将更新后的版本锁定在 package.json 中：

```
npm update <包名> --save
```

第2.10节：作用域和仓库

```
# 为作用域 "myscope" 设置仓库  
npm config set @myscope:registry http://registry.corporation.com  
  
# 在仓库登录并将其与作用域 "myscope" 关联  
npm adduser --registry=http://registry.corporation.com --scope=@myscope  
  
# 从作用域 "myscope" 安装包 "mylib"  
npm install @myscope/mylib
```

如果您自己的包名以@myscope开头，且作用域“myscope”关联到不同的仓库，`npm publish`将会把您的包上传到该仓库。

您也可以将这些设置保存在`.npmrc`文件中：

```
@myscope:registry=http://registry.corporation.com  
//registry.corporation.com/_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxx
```

这在例如自动化构建的CI服务器上非常有用。

第2.11节：链接项目以加快调试和开发

构建项目依赖有时是一项繁琐的任务。与其发布一个包版本到NPM并安装依赖来测试更改，不如使用`npm link`。`npm link`会创建一个符号链接，这样最新代码可以在本地环境中测试。这使得测试全局工具和项目依赖更容易，因为可以在发布版本之前运行最新代码。

帮助文本

名称
`npm-link` - 创建包文件夹的符号链接

用法
`npm link` (在 package 目录中)
`npm link [<@scope>/]<pkg>[@<version>]`

别名：`npm ln`

链接项目依赖的步骤

创建依赖链接时，请注意包名是父项目中将被引用的名称。

In order to update a specific package:

```
npm update <package name>
```

This will update the package to the latest version according to the restrictions in package.json

In case you also want to lock the updated version in package.json:

```
npm update <package name> --save
```

Section 2.10: Scopes and repositories

```
# Set the repository for the scope "myscope"  
npm config set @myscope:registry http://registry.corporation.com  
  
# Login at a repository and associate it with the scope "myscope"  
npm adduser --registry=http://registry.corporation.com --scope=@myscope  
  
# Install a package "mylib" from the scope "myscope"  
npm install @myscope/mylib
```

If the name of your own package starts with @myscope and the scope "myscope" is associated with a different repository, `npm publish` will upload your package to that repository instead.

You can also persist these settings in a `.npmrc` file:

```
@myscope:registry=http://registry.corporation.com  
//registry.corporation.com/_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxx
```

This is useful when automating the build on a CI server f.e.

Section 2.11: Linking projects for faster debugging and development

Building project dependencies can sometimes be a tedious task. Instead of publishing a package version to NPM and installing the dependency to test the changes, use `npm link`. `npm link` creates a symlink so the latest code can be tested in a local environment. This makes testing global tools and project dependencies easier by allowing the latest code run before making a published version.

Help text

NAME
`npm-link` - Symlink a package folder

SYNOPSIS
`npm link` (`in` package `dir`)
`npm link [<@scope>/]<pkg>[@<version>]`

alias: `npm ln`

Steps for linking project dependencies

When creating the dependency link, note that the package name is what is going to be referenced in the parent project.

1. 进入依赖目录（例如：cd .. /my-dep）
2. npm 链接
3. 进入将使用该依赖的项目目录
4. npm link my-dep 或者如果有命名空间 npm link @namespace/my-dep

链接全局工具的步骤

1. 进入项目目录（例如：cd eslint-watch）
2. npm 链接
3. 使用该工具
4. esw --quiet

可能出现的问题

如果依赖项或全局工具已经安装，链接项目有时会导致问题。运行 `npm uninstall (-g) <pkg>` 然后正常运行 `npm link` 通常可以解决可能出现的任何问题。

第2.12节：将模块锁定到特定版本

默认情况下，npm 会根据每个依赖项的语义版本安装最新可用的模块版本。如果模块作者不遵守语义版本控制（semver），并在模块更新中引入破坏性更改，这可能会导致问题。

要将每个依赖项的版本（以及它们的依赖项版本等）锁定到本地 `node_modules` 文件夹中安装的特定版本，请使用

```
npm shrinkwrap
```

这将创建一个 `npm-shrinkwrap.json` 文件，位于你的 `package.json` 文件旁边，列出依赖项的具体版本。

第2.13节：为全局安装的软件包进行设置

你可以使用 `npm install -g` 来“全局”安装软件包。通常这样做是为了安装一个可执行文件，方便你将其添加到路径中运行。例如：

```
npm install -g gulp-cli
```

如果你更新了路径，就可以直接调用 `gulp`。

在许多操作系统中，`npm install -g` 会尝试写入用户可能无权限写入的目录，例如 `/usr/bin`。此时你不应使用 `sudo npm install`，因为以 `sudo` 运行任意脚本存在安全风险，且 `root` 用户可能会在你的主目录下创建你无法写入的目录，这会使未来的安装更加困难。

你可以通过配置文件 `~/.npmrc` 告诉 npm 全局模块的安装位置，这称为 `prefix`，你可以用 `npm prefix` 查看。

```
prefix=~/.npm-global-modules
```

每当你运行 `npm install -g` 时都会使用该 `prefix`。你也可以在安装时使用 `npm install --prefix ~/.npm-global-modules` 来设置 `prefix`。如果 `prefix` 与配置文件中相同，则无需使用 `-g`。

1. CD into a dependency directory (ex: `cd .. /my-dep`)
2. npm link
3. CD into the project that is going to use the dependency
4. npm link my-dep or if namespaced npm link @namespace/my-dep

Steps for linking a global tool

1. CD into the project directory (ex: `cd eslint-watch`)
2. npm link
3. Use the tool
4. esw --quiet

Problems that may arise

Linking projects can sometimes cause issues if the dependency or global tool is already installed. `npm uninstall (-g) <pkg>` and then running `npm link` normally resolves any issues that may arise.

Section 2.12: Locking modules to specific versions

By default, npm installs the latest available version of modules according to each dependencies' semantic version. This can be problematic if a module author doesn't adhere to semver and introduces breaking changes in a module update, for example.

To lock down each dependencies' version (and the versions of their dependencies, etc) to the specific version installed locally in the `node_modules` folder, use

```
npm shrinkwrap
```

This will then create a `npm-shrinkwrap.json` alongside your `package.json` which lists the specific versions of dependancies.

Section 2.13: Setting up for globally installed packages

You can use `npm install -g` to install a package "globally." This is typically done to install an executable that you can add to your path to run. For example:

```
npm install -g gulp-cli
```

If you update your path, you can call `gulp` directly.

On many OSes, `npm install -g` will attempt to write to a directory that your user may not be able to write to such as `/usr/bin`. You should **not** use `sudo npm install` in this case since there is a possible security risk of running arbitrary scripts with sudo and the root user may create directories in your home that you cannot write to which makes future installations more difficult.

You can tell npm where to install global modules to via your configuration file, `~/.npmrc`. This is called the `prefix` which you can view with `npm prefix`.

```
prefix=~/.npm-global-modules
```

This will use the `prefix` whenever you run `npm install -g`. You can also use `npm install --prefix ~/.npm-global-modules` to set the `prefix` when you install. If the `prefix` is the same as your configuration, you don't need to use `-g`.

为了使用全局安装的模块，需要将其添加到你的路径中：

```
export PATH=$PATH:~/npm-global-modules/bin
```

现在当你运行 `npm install -g gulp-cli` 时，就可以使用 `gulp` 了。

注意：当你 `npm install` (不带 `-g`) 时，前缀将是包含 `package.json` 的目录，或者如果在层级中未找到，则为当前目录。这也会创建一个目录 `node_modules/.bin`，里面包含可执行文件。如果你想使用特定于某个项目的可执行文件，则不必使用 `npm install -g`。

你可以使用位于 `node_modules/.bin` 中的那个。

belindoc.com

In order to use the globally installed module, it needs to be on your path:

```
export PATH=$PATH:~/npm-global-modules/bin
```

Now when you run `npm install -g gulp-cli` you will be able to use `gulp`.

Note: When you `npm install` (without `-g`) the prefix will be the directory with `package.json` or the current directory if none is found in the hierarchy. This also creates a directory `node_modules/.bin` that has the executables. If you want to use an executable that is specific to a project, it's not necessary to use `npm install -g`. You can use the one in `node_modules/.bin`.

第3章：使用 Express 构建 Web 应用

参数

路径 指定给定回调函数将处理的路径部分或 URL。

一个或多个函数，这些函数将在回调之前被调用。本质上是多个回调函数的链式调用。

中间件 回调函数。对于更具体的处理，例如授权或错误

处理，非常有用。

回调函数 一个用于处理指定路径请求的函数。它将被调用为

`callback(request, response, next)`，其中`request`、`response`和`next`的含义如下。

回调函数中的`request`是一个封装了有关 HTTP 请求详细信息的对象，该回调函数将处理该请求。

响应 用于指定服务器应如何响应请求的对象。

下一个 一个回调函数，将控制权传递给下一个匹配的路由。它接受一个可选的错误对象。

Express 是一个简洁且灵活的 Node.js 网络应用框架，提供了一套强大的功能用于构建网络应用。

Express 的官方网站是 expressjs.com。源代码可以在 GitHub 上找到。

第3.1节：入门

你首先需要创建一个目录，在终端中进入该目录，并通过运行 `npm`

`install express --save` 来安装 Express

创建一个文件并命名为 `app.js`，添加以下代码，该代码创建了一个新的 Express 服务器并添加了一个端点 `(/ping)`，使用 `app.get` 方法：

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

const app = express();

app.get('/ping', (request, response) => {
  response.send('pong');
});

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

要运行您的脚本，请在终端中使用以下命令：

```
> node app.js
```

您的应用程序将接受来自本地主机 (`localhost`) 8080端口的连接。如果省略了传递给`app.listen`的主机名参数，服务器将接受来自机器IP地址和本地主机的连接。如果端口值为0，操作系统将分配一个可用端口。

脚本运行后，您可以在终端中测试，确认从服务器获得预期响应“pong”：

```
> curl http://localhost:8080/ping
pong
```

您也可以打开网页浏览器，访问网址<http://localhost:8080/ping>查看输出内容

Chapter 3: Web Apps With Express

Parameter

path Specifies the path portion or the URL that the given callback will handle.

middleware One or more functions which will be called before the callback. Essentially a chaining of multiple callback functions. Useful for more specific handling for example authorization or error handling.

callback A function that will be used to handle requests to the specified path. It will be called like `callback(request, response, next)`, where `request`, `response`, and `next` are described below.

callback request An object encapsulating details about the HTTP request that the callback is being called to handle.

response An object that is used to specify how the server should respond to the request.

next A callback that passes control on to the next matching route. It accepts an optional error object.

Express is a minimal and flexible Node.js web application framework, providing a robust set of features for building web applications.

The official website of Express is expressjs.com. The source can be found [on GitHub](#).

Section 3.1: Getting Started

You will first need to create a directory, access it in your shell and install Express using npm by running `npm`

`install express --save`

Create a file and name it `app.js` and add the following code which creates a new Express server and adds one endpoint to it `(/ping)` with the `app.get` method:

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

const app = express();

app.get('/ping', (request, response) => {
  response.send('pong');
});

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

To run your script use the following command in your shell:

```
> node app.js
```

Your application will accept connections on localhost port 8080. If the hostname argument to `app.listen` is omitted, then server will accept connections on the machine's IP address as well as localhost. If port value is 0, the operating system will assign an available port.

Once your script is running, you can test it in a shell to confirm that you get the expected response, "pong", from the server:

```
> curl http://localhost:8080/ping
pong
```

You can also open a web browser, navigate to the url <http://localhost:8080/ping> to view the output

第3.2节：基础路由

首先创建一个express应用：

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

然后您可以这样定义路由：

```
app.get('/someUri', function (req, res, next) {})
```

该结构适用于所有HTTP方法，期望第一个参数为路径，第二个参数为该路径的处理函数，该函数接收请求和响应对象。因此，对于基本的HTTP方法，这些是路由

```
// GET www.domain.com/myPath
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})

// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})

// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

你可以在这里查看支持的所有HTTP动词完整列表。如果你想为某个路由定义相同的行为，适用于所有HTTP方法，可以使用：

```
app.all('/myPath', function (req, res, next) {})
```

或者

```
app.use('/myPath', function (req, res, next) {})
```

或者

```
app.use('*', function (req, res, next) {})
```

```
// * 通配符将匹配所有路径
```

你可以为单一路由链式定义路由

```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

你也可以为任何 HTTP 方法添加函数。它们会在最终回调之前运行，并以 (req, res, next) 作为参数。

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

你的最终回调可以存放在外部文件中，以避免在一个文件中写入过多代码：

Section 3.2: Basic routing

First create an express app:

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

Then you can define routes like this:

```
app.get('/someUri', function (req, res, next) {})
```

That structure works for all HTTP methods, and expects a path as the first argument, and a handler for that path, which receives the request and response objects. So, for the basic HTTP methods, these are the routes

```
// GET www.domain.com/myPath
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})

// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})

// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

You can check the complete list of supported verbs [here](#). If you want to define the same behavior for a route and all HTTP methods, you can use:

```
app.all('/myPath', function (req, res, next) {})
```

or

```
app.use('/myPath', function (req, res, next) {})
```

or

```
app.use('*', function (req, res, next) {})
```

```
// * wildcard will route for all paths
```

You can chain your route definitions for a single path

```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

You can also add functions to any HTTP method. They will run before the final callback and take the parameters (req, res, next) as arguments.

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

Your final callbacks can be stored in an external file to avoid putting too much code in one file:

```
// other.js
exports.doSomething = function(req, res, next) {/* 执行一些操作 */};
```

然后在包含路由的文件中：

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

这将使你的代码更加简洁。

第3.3节：模块化的express应用

为了使express网页应用模块化，使用路由工厂：

模块：

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // 路由工厂
  const router = express.Router();

  router.get('/greet', (req, res, next) => {
    res.end(options.greeting);
  });

  return router;
};
```

应用：

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

express()
  .use('/api/v1/', greetMiddleware({ greeting:'Hello world' }))
  .listen(8080);
```

这将使您的应用程序模块化、可定制，并且代码可重用。

访问 `http://<hostname>:8080/api/v1/greet` 时，输出将是 Hello world

更复杂的示例

展示中间件工厂优势的服务示例。

模块：

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // 路由工厂
  const router = express.Router();
  // 获取控制器
  const {service} = options;

  router.get('/greet', (req, res, next) => {
```

```
// other.js
exports.doSomething = function(req, res, next) {/* do some stuff */};
```

And then in the file containing your routes:

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

This will make your code much cleaner.

Section 3.3: Modular express application

To make express web application modular use router factories:

Module:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();

  router.get('/greet', (req, res, next) => {
    res.end(options.greeting);
  });

  return router;
};
```

Application:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

express()
  .use('/api/v1/', greetMiddleware({ greeting:'Hello world' }))
  .listen(8080);
```

This will make your application modular, customisable and your code reusable.

When accessing `http://<hostname>:8080/api/v1/greet` the output will be Hello world

More complicated example

Example with services that shows middleware factory advantages.

Module:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();
  // Get controller
  const {service} = options;

  router.get('/greet', (req, res, next) => {
```

```

res.end(
service.createGreeting(req.query.name || '陌生人')
);
});

return router;
};

```

应用：

```

// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

class GreetingService {
constructor(问候语 = 'Hello') {
    this.问候语 = 问候语;
}

createGreeting(名字) {
    return `${this.问候语}, ${名字}!`;
}

express()
.use('/api/v1/service1', greetMiddleware({
    service: new GreetingService('Hello'),
}))
.use('/api/v1/service2', greetMiddleware({
    service: new GreetingService('Hi'),
}))
.listen(8080);

```

访问 <http://<hostname>:8080/api/v1/service1/greet?name=World> 时，输出将是 Hello, World

访问 <http://<hostname>:8080/api/v1/service2/greet?name=World> 时，输出将是 Hi, World。

第3.4节：使用模板引擎

使用模板引擎

以下代码将设置 Jade 作为模板引擎。 (注意：Jade 自 2015 年 12 月起已更名为 pug。)

```

const express = require('express'); //导入 express 模块
const app = express(); //创建 express 模块的实例

const PORT = 3000; //随机选择的端口

app.set('view engine','jade'); //设置 jade 作为视图引擎 / 模板引擎
app.set('views','src/views'); //设置存放所有视图 (jade 文件) 的目录。

//创建根路由
app.get('/',function(req, res){
res.render('index'); //将 index.jade 文件渲染为 html 并作为响应返回。
render 函数可选地接收传递给视图的数据。
});

//启动 Express 服务器并带回调函数
app.listen(PORT, function(err) {
if (!err) {
console.log('服务器正在端口运行', PORT);
}

```

```

res.end(
    service.createGreeting(req.query.name || 'Stranger')
);
});

return router;
};

```

Application:

```

// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

class GreetingService {
constructor(greeting = 'Hello') {
    this.greeting = greeting;
}

createGreeting(name) {
    return `${this.greeting}, ${name}!`;
}

express()
.use('/api/v1/service1', greetMiddleware({
    service: new GreetingService('Hello'),
}))
.use('/api/v1/service2', greetMiddleware({
    service: new GreetingService('Hi'),
}))
.listen(8080);

```

When accessing <http://<hostname>:8080/api/v1/service1/greet?name=World> the output will be Hello, World and accessing <http://<hostname>:8080/api/v1/service2/greet?name=World> the output will be Hi, World.

Section 3.4: Using a Template Engine

Using a Template Engine

The following code will setup Jade as template engine. (Note: Jade has been renamed to pug as of December 2015.)

```

const express = require('express'); //Imports the express module
const app = express(); //Creates an instance of the express module

const PORT = 3000; //Randomly chosen port

app.set('view engine','jade'); //Sets jade as the View Engine / Template Engine
app.set('views','src/views'); //Sets the directory where all the views (.jade files) are stored.

//Creates a Root Route
app.get('/',function(req, res){
    res.render('index'); //renders the index.jade file into html and returns as a response. The render function optionally takes the data to pass to the view.
});

//Starts the Express server with a callback
app.listen(PORT, function(err) {
    if (!err) {
        console.log('Server is running at port', PORT);
    }
});

```

```

} else {
  console.log(JSON.stringify(err));
}
});

```

同样，也可以使用其他模板引擎，如Handlebars (hbs) 或 ejs。记得也要通过npm 安装模板引擎。对于Handlebars，我们使用hbs包，Jade使用jade包，EJS使用ejs包。

EJS 模板示例

使用 EJS (和其他 Express 模板类似)，你可以运行服务器代码并从 HTML 中访问服务器变量。在 EJS 中，使用“`<%`”作为起始标签，“`%>`”作为结束标签，传入渲染参数的变量可以通过`<%=var_name%>`访问。

例如，如果你的服务器代码中有 `supplies` 数组，你可以使用以下方式循环遍历它

```

<h1><%= title %></h1>
<ul>
<% for(var i=0; i<supplies.length; i++) { %>
  <li>
    <a href='supplies/<%= supplies[i] %>'>
      <%= supplies[i] %>
    </a>
  </li>
<% } %>

```

正如示例中所示，每次在服务器端代码和HTML之间切换时，都需要关闭当前的EJS标签，然后稍后再打开一个新的标签，这里我们想在for命令内部创建li，因此需要在for结束时关闭EJS标签，并专门为大括号创建新的标签。

另一个示例

如果我们想将输入的默认值设置为来自服务器端的变量，我们使用`<%=`。例如：

```

消息:<br>
<input type="text" value="<%= message %>" name="message" required>

```

这里传递自服务器端的message变量将作为输入的默认值，请注意，如果你没有从服务器端传递message变量，EJS将抛出异常。你可以使用`res.render('index', {message: message});` (针对名为index.ejs的ejs文件) 来传递参数。

在EJS标签中，你也可以使用if、while或任何其他你想要的JavaScript命令。

第3.5节：使用ExpressJS的JSON API

```

var express = require('express');
var cors = require('cors'); // 使用cors模块以启用跨源资源共享

var app = express();
app.use(cors()); // 适用于所有路由

var port = process.env.PORT || 8080;

app.get('/', function(req, res) {
  var info = {
    'string_value': 'StackOverflow',
  }
  res.json(info);
});

```

```

} else {
  console.log(JSON.stringify(err));
}
});

```

Similarly, other Template Engines could be used too such as Handlebars(hbs) or ejs. Remember to npm `install` the Template Engine too. For Handlebars we use hbs package, for Jade we have a jade package and for EJS, we have an ejs package.

EJS Template Example

With EJS (like other express templates), you can run server code and access your server variables from your HTML. In EJS it's done using “`<%`” as start tag and “`%>`” as end tag, variables passed as the render params can be accessed using `<%=var_name%>`

For instance, if you have supplies array in your server code you can loop over it using

```

<h1><%= title %></h1>
<ul>
<% for(var i=0; i<supplies.length; i++) { %>
  <li>
    <a href='supplies/<%= supplies[i] %>'>
      <%= supplies[i] %>
    </a>
  </li>
<% } %>

```

As you can see in the example every time you switch between server side code and HTML you need to close the current EJS tag and open a new one later, here we wanted to create li inside the `for` command so we needed to close our EJS tag at the end of the `for` and create new tag just for the curly brackets another example

if we want to put input default version to be a variable from the server side we use `<%=` for example:

```

Message:<br>
<input type="text" value="<%= message %>" name="message" required>

```

Here the message variable passed from your server side will be the default value of your input, please be noticed that if you didn't pass message variable from your server side, EJS will throw an exception. You can pass parameters using `res.render('index', {message: message})`; (for ejs file called index.ejs).

In the EJS tags you can also use `if`, `while` or any other javascript command you want.

Section 3.5: JSON API with ExpressJS

```

var express = require('express');
var cors = require('cors'); // Use cors module for enable Cross-origin resource sharing

var app = express();
app.use(cors()); // for all routes

var port = process.env.PORT || 8080;

app.get('/', function(req, res) {
  var info = {
    'string_value': 'StackOverflow',
  }
  res.json(info);
});

```

```

    'number_value': 8476
}
res.json(info);

// 或者
/* res.send(JSON.stringify({
  string_value: 'StackOverflow',
  number_value: 8476
})) */

// 你可以为 JSON 响应添加状态码
/* res.status(200).json(info) */
}

app.listen(port, function() {
  console.log('Node.js listening on port ' + port)
})

```

在http://localhost:8080/ 输出对象

```
{
  string_value: "StackOverflow",
  number_value: 8476
}
```

第3.6节：提供静态文件

使用Express构建Web服务器时，通常需要同时提供动态内容和静态文件。

例如，你可能有index.html和script.js，这些是保存在文件系统中的静态文件。

通常会使用名为“public”的文件夹来存放静态文件。在这种情况下，文件夹结构可能如下所示：

```
项目根目录
├── server.js
├── package.json
└── public
    ├── index.html
    └── script.js
```

以下是配置Express以提供静态文件的方法：

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

app.use(express.static('public'));
```

注意：一旦文件夹配置完成，index.html、script.js 以及“public”文件夹中的所有文件都将在根路径下可用（URL 中不应指定/public/）。这是因为 express 会相对于配置的静态文件夹查找文件。你可以像下面这样指定虚拟路径前缀：

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

将使资源在/static/前缀下可用。

多个文件夹

```

    'number_value': 8476
}
res.json(info);

// or
/* res.send(JSON.stringify({
  string_value: 'StackOverflow',
  number_value: 8476
})) */

//you can add a status code to the json response
/* res.status(200).json(info) */
}

app.listen(port, function() {
  console.log('Node.js listening on port ' + port)
})

```

On http://localhost:8080/ output object

```
{
  string_value: "StackOverflow",
  number_value: 8476
}
```

Section 3.6: Serving static files

When building a webserver with Express it's often required to serve a combination of dynamic content and static files.

For example, you may have index.html and script.js which are static files kept in the file system.

It is common to use folder named 'public' to have static files. In this case the folder structure may look like:

```
project root
├── server.js
├── package.json
└── public
    ├── index.html
    └── script.js
```

This is how to configure Express to serve static files:

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

app.use(express.static('public'));
```

Note: once the folder is configured, index.html, script.js and all the files in the "public" folder will be available in at the root path (you must not specify /public/ in the url). This is because, express looks up for the files relative to the static folder configured. You can specify *virtual path prefix* as shown below:

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

will make the resources available under the **/static/** prefix.

Multiple folders

可以同时定义多个文件夹：

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static('files'));
```

在提供资源时，Express 会按照定义顺序检查文件夹。如果存在同名文件，将提供第一个匹配文件夹中的文件。

第3.7节：添加中间件

中间件函数是能够访问请求对象（req）、响应对象（res）以及应用中请求-响应周期中下一个中间件函数的函数。

中间件函数可以执行任何代码，对res和req对象进行修改，结束响应周期并调用下一个中间件。

中间件的一个非常常见的例子是cors模块。要添加CORS支持，只需安装它，引用它，并添加这一行：

```
app.use(cors());
```

在任何路由或路由函数之前。

第3.8节：错误处理

基本错误处理

默认情况下，Express 会在/views目录中查找“error”视图进行渲染。只需创建“error”视图并将其放置在views目录中即可处理错误。错误信息包括错误消息、状态和堆栈跟踪，例如：

views/error.pug

```
html
  body
    h1= message
    h2= error.status
    p= error.stack
```

高级错误处理

请将错误处理中间件函数定义在中间件函数栈的最末端。这些函数有四个参数，而不是三个（err, req, res, next），例如：

app.js

```
// 捕获404并转发到错误处理器
app.use(function(req, res, next) {
  var err = new Error('未找到');
  err.status = 404;

  // 将错误传递给下一个匹配的路由。
  next(err);
});
```

It is possible to define multiple folders at the same time:

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static('files'));
```

When serving the resources Express will examine the folder in definition order. In case of files with the same name, the one in the first matching folder will be served.

Section 3.7: Adding Middleware

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.

Middleware functions can execute any code, make changes to res and req objects, end response cycle and call next middleware.

Very common example of middleware is cors module. To add CORS support, simply install it, require it and put this line:

```
app.use(cors());
```

before any routers or routing functions.

Section 3.8: Error Handling

Basic Error Handling

By default, Express will look for an 'error' view in the /views directory to render. Simply create the 'error' view and place it in the views directory to handle errors. Errors are written with the error message, status and stack trace, for example:

views/error.pug

```
html
  body
    h1= message
    h2= error.status
    p= error.stack
```

Advanced Error Handling

Define your error-handling middleware functions at the very end of the middleware function stack. These have four arguments instead of three (err, req, res, next) for example:

app.js

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;

  // pass error to the next matching route.
  next(err);
});
```

```
// 处理错误，打印堆栈信息
app.use(function(err, req, res, next) {
  res.status(err.status || 500);

  res.render('error', {
    message: err.message,
    error: err
  });
});
```

您可以定义多个错误处理中间件函数，就像定义普通中间件函数一样。

第3.9节：从请求中获取信息

要从请求的URL中获取信息（注意 `req` 是路由处理函数中的请求对象）。考虑这个路由定义 `/settings/:user_id` 以及这个具体示例 `/settings/32135?field=name`

```
// 获取完整路径
req.originalUrl // => /settings/32135?field=name

// 获取user_id参数
req.params.user_id // => 32135

// 获取field的查询值
req.query.field // => 'name'
```

你也可以这样获取请求头

```
req.get('Content-Type')
// "text/plain"
```

为了简化获取其他信息，你可以使用中间件。例如，要获取请求体信息，你可以使用 `body-parser` 中间件，它会将原始请求体转换为可用格式。

```
var app = require('express');
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // 用于解析 application/json
app.use(bodyParser.urlencoded({ extended: true })); // 用于解析 application/x-www-form-urlencoded
```

现在假设有如下请求

```
PUT /settings/32135
{
  "name": "彼得"
}
```

您可以这样访问发布的名称

```
req.body.name
// "彼得"
```

以类似的方式，您可以从请求中访问 cookie，您还需要像 `cookie-parser` 这样的中间件

```
req.cookies.name
```

```
// handle error, print stacktrace
app.use(function(err, req, res, next) {
  res.status(err.status || 500);

  res.render('error', {
    message: err.message,
    error: err
  });
});
```

You can define several error-handling middleware functions, just as you would with regular middleware functions.

Section 3.9: Getting info from the request

To get info from the requesting url (notice that `req` is the request object in the handler function of routes). Consider this route definition `/settings/:user_id` and this particular example `/settings/32135?field=name`

```
// get the full path
req.originalUrl // => /settings/32135?field=name

// get the user_id param
req.params.user_id // => 32135

// get the query value of the field
req.query.field // => 'name'
```

You can also get headers of the request, like this

```
req.get('Content-Type')
// "text/plain"
```

To simplify getting other info you can use middlewares. For example, to get the body info of the request, you can use the `body-parser` middleware, which will transform raw request body into usable format.

```
var app = require('express');
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded
```

Now suppose a request like this

```
PUT /settings/32135
{
  "name": "Peter"
}
```

You can access the posted name like this

```
req.body.name
// "Peter"
```

In a similar way, you can access cookies from the request, you also need a middleware like `cookie-parser`

```
req.cookies.name
```

第3.10节：Express中的错误处理

在 Express 中，您可以定义统一的错误处理程序来处理应用程序中发生的错误。请在所有路由和逻辑代码的末尾定义该处理程序。

示例

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

//GET /names/john
app.get('/names/:name', function(req, res, next){
  if (req.params.name == 'john'){
    return res.send('有效的名字');
  } else{
    next(new Error('无效的名字')); //传递给错误处理程序
  }
});

//错误处理程序
app.use(function(err, req, res, next){
  console.log(err.stack); // 例如, 无效的名字
  return res.status(500).send('发生内部服务器错误');
});

app.listen(3000);
```

第3.11节：钩子：如何在任何请求之前和任何响应之后执行代码

app.use() 和中间件可以用于“之前”，close 和 finish 事件的组合可以用于“之后”。

```
app.use(function (req, res, next) {
  function afterResponse() {
    res.removeListener('finish', afterResponse);
    res.removeListener('close', afterResponse);

    // 响应后的操作
  }
  res.on('finish', afterResponse);
  res.on('close', afterResponse);

  // 请求前的操作
  // 最终调用 `next()`
  next();
};

app.use(app.router);
```

一个例子是 logger 中间件，默认会在响应后追加日志。

只需确保这个“中间件”在 app.router 之前使用，因为顺序很重要。

原始帖子在这里 [here](#)

Section 3.10: Error handling in Express

In Express, you can define unified error handler for handling errors occurred in application. Define then handler at the end of all routes and logic code.

Example

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

//GET /names/john
app.get('/names/:name', function(req, res, next){
  if (req.params.name == 'john'){
    return res.send('Valid Name');
  } else{
    next(new Error('Not valid name')); //pass to error handler
  }
});

//error handler
app.use(function(err, req, res, next){
  console.log(err.stack); // e.g., Not valid name
  return res.status(500).send('Internal Server Occurred');
});

app.listen(3000);
```

Section 3.11: Hook: How to execute code before any req and after any res

app.use() and middleware can be used for "before" and a combination of the [close](#) and [finish](#) events can be used for "after".

```
app.use(function (req, res, next) {
  function afterResponse() {
    res.removeListener('finish', afterResponse);
    res.removeListener('close', afterResponse);

    // actions after response
  }
  res.on('finish', afterResponse);
  res.on('close', afterResponse);

  // action before request
  // eventually calling `next()`
  next();
};

app.use(app.router);
```

An example of this is the [logger](#) middleware, which will append to the log after the response by default.

Just make sure this "middleware" is used before app.router as order does matter.

Original post is [here](#)

第3.12节：使用cookie-parser设置Cookie

以下是使用cookie-parser模块设置和读取Cookie的示例：

```
var express = require('express');
var cookieParser = require('cookie-parser'); // 用于解析Cookie的模块
var app = express();
app.use(cookieParser());

app.get('/setcookie', function(req, res){
    // 设置Cookie
    res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });
    return res.send('Cookie已设置');
});

app.get('/getcookie', function(req, res) {
    var username = req.cookies['username'];
    if (username) {
        return res.send(username);
    }

    return res.send('未找到Cookie');
});

app.listen(3000);
```

第3.13节：Express中的自定义中间件

在 Express 中，你可以定义中间件，用于检查请求或设置响应中的某些头信息。

```
app.use(function(req, res, next){}); // 签名
```

示例

以下代码将 user 添加到请求对象中，并将控制权传递给下一个匹配的路由。

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

// 每个请求都会经过这里
app.use(function(req, res, next){
    req.user = 'testuser';
    next(); // 它会将控制权传递给下一个匹配的路由
});

app.get('/', function(req, res){
    var user = req.user;
    console.log(user); // testuser
    return res.send(user);
});

app.listen(3000);
```

第3.14节：Django风格的命名路由

一个大问题是 Express 默认不支持有价值的命名路由。解决方案是安装支持的第三方包，例如 express-reverse：

Section 3.12: Setting cookies with cookie-parser

The following is an example for setting and reading cookies using the [cookie-parser](#) module:

```
var express = require('express');
var cookieParser = require('cookie-parser'); // module for parsing cookies
var app = express();
app.use(cookieParser());

app.get('/setcookie', function(req, res){
    // setting cookies
    res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });
    return res.send('Cookie has been set');
});

app.get('/getcookie', function(req, res) {
    var username = req.cookies['username'];
    if (username) {
        return res.send(username);
    }

    return res.send('No cookie found');
});

app.listen(3000);
```

Section 3.13: Custom middleware in Express

In Express, you can define middlewares that can be used for checking requests or setting some headers in response.

```
app.use(function(req, res, next){}); // signature
```

Example

The following code adds user to the request object and pass the control to the next matching route.

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

// each request will pass through it
app.use(function(req, res, next){
    req.user = 'testuser';
    next(); // it will pass the control to next matching route
});

app.get('/', function(req, res){
    var user = req.user;
    console.log(user); // testuser
    return res.send(user);
});

app.listen(3000);
```

Section 3.14: Named routes in Django-style

One big problem is that valuable named routes is not supported by Express out of the box. Solution is to install supported third-party package, for example [express-reverse](#):

```
npm install express-reverse
```

将其插入您的项目中：

```
var app = require('express')();
require('express-reverse')(app);
```

然后这样使用：

```
app.get('test', '/hello', function(req, res) {
  res.end('hello');
});
```

这种方法的缺点是你不能像高级路由使用中那样使用route Express模块。解决方法是将你的app作为参数传递给路由工厂：

```
require('./middlewares/routing')(app);
```

然后这样使用：

```
module.exports = (app) => {
  app.get('test', '/hello', function(req, res) {
    res.end('hello');
  });
};
```

从现在开始你可以自己摸索如何定义函数，将其合并到指定的自定义命名空间，并指向相应的控制器。

第3.15节：Hello World

这里我们使用 Express 创建了一个基本的 Hello World 服务器。路由：

- '/'
- '/wiki'

其余的将返回“404”，即页面未找到。

```
'use strict';

const port = process.env.PORT || 3000;

var app = require('express')();
app.listen(port);

app.get('/',(req,res)=>res.send('HelloWorld!'));
app.get('/wiki',(req,res)=>res.send('这是维基页面。'));
app.use((req,res)=>res.send('404-页面未找到'));
```

注意：我们将 404 路由放在最后，因为 Express 会按顺序堆叠路由，并对每个请求依次处理。

第 3.16 节：使用中间件和 next 回调

Express 会向每个路由处理器和中间件函数传递一个 next 回调，可用于中断逻辑

```
npm install express-reverse
```

Plug it in your project:

```
var app = require('express')();
require('express-reverse')(app);
```

Then use it like:

```
app.get('test', '/hello', function(req, res) {
  res.end('hello');
});
```

The downside of this approach is that you can't use route Express module as shown in Advanced router usage. The workaround is to pass your app as a parameter to your router factory:

```
require('./middlewares/routing')(app);
```

And use it like:

```
module.exports = (app) => {
  app.get('test', '/hello', function(req, res) {
    res.end('hello');
  });
};
```

You can figure it out from now on, how define functions to merge it with specified custom namespaces and point at appropriate controllers.

Section 3.15: Hello World

Here we create a basic hello world server using Express. Routes:

- '/'
- '/wiki'

And for rest will give "404", i.e. page not found.

```
'use strict';

const port = process.env.PORT || 3000;

var app = require('express')();
app.listen(port);

app.get('/',(req,res)=>res.send('HelloWorld!'));
app.get('/wiki',(req,res)=>res.send('This is wiki page.'));
app.use((req,res)=>res.send('404-PageNotFound'));
```

Note: We have put 404 route as the last route as Express stacks routes in order and processes them for each request sequentially.

Section 3.16: Using middleware and the next callback

Express passes a next callback to every route handler and middleware function that can be used to break logic for

跨多个处理程序的单一路由。调用 `next()` 不带参数时，表示 `express` 继续执行下一个匹配的中间件或路由处理程序。调用 `next(err)` 并传入错误时，会触发任何错误处理的中间件。

调用 `next('route')` 会跳过当前路由上的后续中间件，直接跳转到下一个匹配的路由。这允许将领域逻辑解耦成自包含的可重用组件，更易于测试、维护和修改。

多个匹配路由

对 `/api/foo` 或 `/api/bar` 的请求会先运行初始处理程序以查找成员，然后将控制权传递给每个路由的实际处理程序。

```
app.get('/api', function(req, res, next) {
  // /api/foo 和 /api/bar 都会运行此处理程序
  lookupMember(function(err, member) {
    if (err) return next(err);
    req.member = member;
    next();
  });
});

app.get('/api/foo', function(req, res, next) {
  // 只有 /api/foo 会运行此处理程序
  doSomethingWithMember(req.member);
});

app.get('/api/bar', function(req, res, next) {
  // 只有 /api/bar 会运行此处理程序
  doSomethingDifferentWithMember(req.member);
});
```

错误处理器

错误处理器是具有签名 `function(err, req, res, next)` 的中间件。它们可以针对每个路由设置（例如 `app.get('/foo', function(err, req, res, next))`），但通常，一个渲染错误页面的单一错误处理器就足够了。

```
app.get('/foo', function(req, res, next) {
  doSomethingAsync(function(err, data) {
    if (err) return next(err);
    renderPage(data);
  });

  // 如果 doSomethingAsync 返回错误，这个特殊的
  // 错误处理器中间件将被调用，错误作为
  // 第一个参数传入。
  app.use(function(err, req, res, next) {
    renderErrorPage(err);
  });
});
```

中间件

上述每个函数实际上都是中间件函数，只要请求匹配定义的路由就会运行，但单个路面上可以定义任意数量的中间件函数。这允许中间件在不同文件中定义，并且可以在多个路由间复用公共逻辑。

```
app.get('/bananas', function(req, res, next) {
  getMember(function(err, member) {
```

single routes across multiple handlers. Calling `next()` with no arguments tells `express` to continue to the next matching middleware or route handler. Calling `next(err)` with an error will trigger any error handler middleware. Calling `next('route')` will bypass any subsequent middleware on the current route and jump to the next matching route. This allows domain logic to be decoupled into reusable components that are self-contained, simpler to test, and easier to maintain and change.

Multiple matching routes

Requests to `/api/foo` or to `/api/bar` will run the initial handler to look up the member and then pass control to the actual handler for each route.

```
app.get('/api', function(req, res, next) {
  // Both /api/foo and /api/bar will run this
  lookupMember(function(err, member) {
    if (err) return next(err);
    req.member = member;
    next();
  });
});

app.get('/api/foo', function(req, res, next) {
  // Only /api/foo will run this
  doSomethingWithMember(req.member);
});

app.get('/api/bar', function(req, res, next) {
  // Only /api/bar will run this
  doSomethingDifferentWithMember(req.member);
});
```

Error handler

Error handlers are middleware with the signature `function(err, req, res, next)`. They could be set up per route (e.g. `app.get('/foo', function(err, req, res, next))`) but typically, a single error handler that renders an error page is sufficient.

```
app.get('/foo', function(req, res, next) {
  doSomethingAsync(function(err, data) {
    if (err) return next(err);
    renderPage(data);
  });

  // In the case that doSomethingAsync return an error, this special
  // error handler middleware will be called with the error as the
  // first parameter.
  app.use(function(err, req, res, next) {
    renderErrorPage(err);
  });
});
```

Middleware

Each of the functions above is actually a middleware function that is run whenever a request matches the route defined, but any number of middleware functions can be defined on a single route. This allows middleware to be defined in separate files and common logic to be reused across multiple routes.

```
app.get('/bananas', function(req, res, next) {
  getMember(function(err, member) {
```

```

if (err) return next(err);
// 如果没有成员, 不要尝试查找
// 数据。直接渲染页面即可。
if (!member) return next('route');
// 否则, 调用下一个中间件并获取
// 该成员的数据。
req.member = member;
next();
},
function(req, res, next) {
getMemberData(req.member, function(err, data) {
  if (err) return next(err);
  // 如果该成员没有数据, 则不必
  // 解析它。直接渲染页面即可。
  if (!data) return next('route');
  // 否则, 调用下一个中间件并解析
  // 该成员的数据。然后渲染页面。
  req.member.data = data;
  next();
}),
function(req, res, next) {
req.member.parsedData = parseMemberData(req.member.data);
next();
});

app.get('/bananas', function(req, res, next) {
  renderBananas(req.member);
});
}

```

在此示例中, 每个中间件函数要么位于其自己的文件中, 要么位于文件中其他变量中,以便在其他路由中重用。

第3.17节 : 错误处理

基本文档可以在 [here](#) 找到

```

app.get('/path/:id(\d+)', function (req, res, next) { // 请注意：“next”被传递
  if (req.params.id == 0) // 验证参数
    return next(new Error('Id 是 0')); // 跳转到第一个错误处理器, 见下文

  // 捕获同步操作中的错误
  var data;
  try {
data = JSON.parse('/file.json');
  } catch (err) {
    return next(err);
  }

  // 如果出现严重错误则停止应用程序
  if (!data)
    throw new Error('出现错误');

  // 如果需要向错误处理器发送额外信息
  // 则发送自定义错误 (见附录B)
  if (smth)
    next(new MyError('smth wrong', arg1, arg2))

  // 通过 res.render 或 res.end 完成请求
  res.status(200).end('OK');
});

```

```

if (err) return next(err);
// If there's no member, don't try to look
// up data. Just go render the page now.
if (!member) return next('route');
// Otherwise, call the next middleware and fetch
// the member's data.
req.member = member;
next();
});
},
function(req, res, next) {
getMemberData(req.member, function(err, data) {
  if (err) return next(err);
  // If this member has no data, don't bother
  // parsing it. Just go render the page now.
  if (!data) return next('route');
  // Otherwise, call the next middleware and parse
  // the member's data. THEN render the page.
  req.member.data = data;
  next();
});
},
function(req, res, next) {
req.member.parsedData = parseMemberData(req.member.data);
next();
});

app.get('/bananas', function(req, res, next) {
  renderBananas(req.member);
});
}

```

In this example, each middleware function would be either in its own file or in a variable elsewhere in the file so that it could be reused in other routes.

Section 3.17: Error handling

Basic docs can be found [here](#)

```

app.get('/path/:id(\d+)', function (req, res, next) { // please note: "next" is passed
  if (req.params.id == 0) // validate param
    return next(new Error('Id is 0')); // go to first Error handler, see below

  // Catch error on sync operation
  var data;
  try {
    data = JSON.parse('/file.json');
  } catch (err) {
    return next(err);
  }

  // If some critical error then stop application
  if (!data)
    throw new Error('Smth wrong');

  // If you need send extra info to Error handler
  // then send custom error (see Appendix B)
  if (smth)
    next(new MyError('smth wrong', arg1, arg2))

  // Finish request by res.render or res.end
  res.status(200).end('OK');
});

```

```
// 注意：app.use 的顺序很重要
// 错误处理器
app.use(function(err, req, res, next) {
  if (smth-check, e.g. req.url != 'POST')
    return next(err); // 跳转到错误处理器 2。

console.log(req.url, err.message);

  if (req.xhr) // 如果请求是通过 ajax, 则发送 json, 否则渲染错误页面
    res.json(err);
  else
    res.render('error.html', {error: err.message});
});

// 错误处理器 2
app.use(function(err, req, res, next) {
  // 在这里做一些操作, 例如检查错误是否为 MyError
  if (err instanceof MyError) {
    console.log(err.message, err.arg1, err.arg2);
  }
  ...
  res.end();
});
```

附录 A

```
// "在 Express 中, 404 响应不是错误的结果,
// 因此错误处理中间件不会捕获它们。"
// 你可以更改它。
app.use(function(req, res, next) {
  next(new Error(404));
});
```

附录 B

```
// 如何定义自定义错误
var util = require('util');

function MyError(message, arg1, arg2) {
  this.message = message;
  this.arg1 = arg1;
  this.arg2 = arg2;
  Error.captureStackTrace(this, MyError);
}
util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';
```

第3.18节：处理POST请求

就像你使用app.get方法处理Express中的GET请求一样，你可以使用app.post方法来处理POST请求。

但在你能处理POST请求之前，你需要使用body-parser中间件。它只是解析POST、PUT、DELETE和其他请求的请求体。

Body-Parser中间件解析请求体，并将其转换成一个可通过req.body访问的对象

```
var bodyParser = require('body-parser');
```

```
// Be sure: order of app.use have matter
// Error handler
app.use(function(err, req, res, next) {
  if (smth-check, e.g. req.url != 'POST')
    return next(err); // go-to Error handler 2.

  console.log(req.url, err.message);

  if (req.xhr) // if req via ajax then send json else render error-page
    res.json(err);
  else
    res.render('error.html', {error: err.message});
});

// Error handler 2
app.use(function(err, req, res, next) {
  // do smth here e.g. check that error is MyError
  if (err instanceof MyError) {
    console.log(err.message, err.arg1, err.arg2);
  }
  ...
  res.end();
});
```

Appendix A

```
// "In Express, 404 responses are not the result of an error,
// so the error-handler middleware will not capture them."
// You can change it.
app.use(function(req, res, next) {
  next(new Error(404));
});
```

Appendix B

```
// How to define custom error
var util = require('util');

function MyError(message, arg1, arg2) {
  this.message = message;
  this.arg1 = arg1;
  this.arg2 = arg2;
  Error.captureStackTrace(this, MyError);
}
util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';
```

Section 3.18: Handling POST Requests

Just like you handle get requests in Express with app.get method, you can use app.post method to handle post requests.

But before you can handle POST requests, you will need to use the body-parser middleware. It simply parses the body of POST, PUT, DELETE and other requests.

Body-Parser middleware parses the body of the request and turns it into an object available in req.body

```
var bodyParser = require('body-parser');
```

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

const app = express();

// 解析 POST、PUT、DELETE 等请求的请求体
app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.post('/post-data-here', function(req, res, next){

    console.log(req.body); // req.body 包含请求体的解析内容。

});

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

belindoc.com

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

const app = express();

// Parses the body for POST, PUT, DELETE, etc.
app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.post('/post-data-here', function(req, res, next){

    console.log(req.body); // req.body contains the parsed body of the request.

});

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

第4章：文件系统输入输出

第4.1节：异步读取文件

使用文件系统模块进行所有文件操作：

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

带编码

在此示例中，从目录 `/tmp` 读取 `hello.txt`。此操作将在后台完成，回调函数将在完成或失败时触发：

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) => {
  // 如果发生错误，输出错误信息并返回
  if(err) return console.error(err);

  // 没有发生错误，内容是字符串
  console.log(content);
});
```

无编码

从当前目录异步后台读取二进制文件 `binary.txt`。注意我们没有设置`'encoding'`选项——这防止Node.js将内容解码为字符串：

```
fs.readFile('binary', (err, binaryContent) => {
  // 如果发生错误，输出错误信息并返回
  if(err) return console.error(err);

  // 没有发生错误，内容是Buffer，以下以
  // 十六进制表示输出。
  console.log(content.toString('hex'));
});
```

相对路径

请记住，一般情况下，您的脚本可能在任意当前工作目录下运行。要访问相对于当前脚本的文件，请使用`_dirname`或`_filename`：

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent) => {
  // 函数其余部分
})
```

第4.2节：使用readdir或 readdirSync列出目录内容

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

// 异步读取目录/usr/local/bin的内容。
// 操作完成或失败后，回调函数将被调用。

fs.readdir('/usr/local/bin', (err, files) => {
  // 出错时，显示错误并返回
})
```

Chapter 4: Filesystem I/O

Section 4.1: Asynchronously Read from Files

Use the filesystem module for all file operations:

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

With Encoding

In this example, read `hello.txt` from the directory `/tmp`. This operation will be completed in the background and the callback occurs on completion or failure:

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a string
  console.log(content);
});
```

Without Encoding

Read the binary file `binary.txt` from the current directory, asynchronously in the background. Note that we do not set the `'encoding'` option - this prevents Node.js from decoding the contents into a string:

```
fs.readFile('binary', (err, binaryContent) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a Buffer, output it in
  // hexadecimal representation.
  console.log(content.toString('hex'));
});
```

Relative paths

Keep in mind that, in general case, your script could be run with an arbitrary current working directory. To address a file relative to the current script, use `__dirname` or `__filename`:

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent) => {
  // Rest of Function
})
```

Section 4.2: Listing Directory Contents with readdir or readdirSync

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

// Read the contents of the directory /usr/local/bin asynchronously.
// The callback will be invoked once the operation has either completed
// or failed.
fs.readdir('/usr/local/bin', (err, files) => {
  // On error, show it and return
})
```

```

if(err) return console.error(err);

// files是一个数组，包含目录中所有条目的名称// 不包括'.' (目录本身)
// 和'..' (父目录)。

// 显示目录条目
console.log(files.join(' '));
});

```

有一个同步版本readdirSync，它会阻塞主线程，因此会阻止异步代码的同时执行。大多数开发者为了提升性能，避免使用同步IO函数。

```

let files;

try {
  files = fs.readdirSync('/var/tmp');
} catch(err) {
  // 发生错误
  console.error(err);
}

```

使用生成器

```

const fs = require('fs');

// 遍历通过
// 'yield' 语句获取的所有项目
// 由于'readdir'方法需要，回调函数被传递给生成器函数

function run(gen) {
  var iter = gen((err, data) => {
    if (err) { iter.throw(err); }

    return iter.next(data);
  });

  iter.next();
}

const dirPath = '/usr/local/bin';

// 执行生成器函数
run(function* (resume) {
  // 从生成器中发出目录中的文件列表
  var contents = yield fs.readdir(dirPath, resume);
  console.log(contents);
});

```

第4.3节：通过管道流复制文件

该程序使用可读流和可写流，通过流类提供的pipe()函数复制文件

```

// 引入文件系统模块
var fs = require('fs');

/*

```

```

if(err) return console.error(err);

// files is an array containing the names of all entries
// in the directory, excluding '.' (the directory itself)
// and '..' (the parent directory).

// Display directory entries
console.log(files.join(' '));
});

```

A synchronous variant is available as readdirSync which blocks the main thread and therefore prevents execution of asynchronous code at the same time. Most developers avoid synchronous IO functions in order to improve performance.

```

let files;

try {
  files = fs.readdirSync('/var/tmp');
} catch(err) {
  // An error occurred
  console.error(err);
}

```

Using a generator

```

const fs = require('fs');

// Iterate through all items obtained via
// 'yield' statements
// A callback is passed to the generator function because it is required by
// the 'readdir' method
function run(gen) {
  var iter = gen((err, data) => {
    if (err) { iter.throw(err); }

    return iter.next(data);
  });

  iter.next();
}

const dirPath = '/usr/local/bin';

// Execute the generator function
run(function* (resume) {
  // Emit the list of files in the directory from the generator
  var contents = yield fs.readdir(dirPath, resume);
  console.log(contents);
});

```

Section 4.3: Copying files by piping streams

This program copies a file using readable and a writable stream with the pipe() function provided by the stream class

```

// require the file system module
var fs = require('fs');

/*

```

```

创建当前目录下名为'node.txt'的可读流
使用utf8编码
以16字节为块读取数据
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 * 1024 });

// 创建可写流
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');

// 使用管道将可读流复制到可写流
readable.pipe(writable);

```

第4.4节：同步读取文件

对于任何文件操作，您都需要文件系统模块：

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

读取字符串

`fs.readFileSync` 的行为类似于 `fs.readFile`，但不接受回调，因为它是同步完成的，因此会阻塞主线程。大多数Node.js开发者更喜欢异步版本，因为它几乎不会延迟程序执行。

如果指定了 `encoding` 选项，将返回字符串，否则将返回 `Buffer`。

```

// 同步从另一个文件读取字符串
let content;
try {
  content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
  // 发生错误
  console.error(err);
}

```

第4.5节：检查文件或目录的权限

`fs.access()` 用于确定路径是否存在以及用户对该路径下的文件或目录拥有什么权限。`fs.access`不会返回结果，而是如果不返回错误，则表示路径存在且用户拥有所需的权限。

权限模式作为属性存在于 `fs` 对象上，即 `fs.constants`

- `fs.constants.F_OK` - 拥有读/写/执行权限（如果未提供模式，则默认为此）
- `fs.constants.R_OK` - 拥有读取权限
- `fs.constants.W_OK` - 拥有写入权限
- `fs.constants.X_OK` - 拥有执行权限（在Windows上与 `fs.constants.F_OK` 功能相同）

异步地

```

var fs = require('fs');
var path = '/path/to/check';

// 检查执行权限
fs.access(path, fs.constants.X_OK, (err) => {
  if (err) {
    console.log("%s 不存在", path);
  }
});

```

```

Create readable stream to file in current directory named 'node.txt'
Use utf8 encoding
Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');

// use pipe to copy readable to writable
readable.pipe(writable);

```

Section 4.4: Reading from a file synchronously

For any file operations, you will need the `filesystem` module:

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

Reading a String

`fs.readFileSync` behaves similarly to `fs.readFile`, but does not take a callback as it completes synchronously and therefore blocks the main thread. Most node.js developers prefer the asynchronous variants which will cause virtually no delay in the program execution.

If an `encoding` option is specified, a string will be returned, otherwise a `Buffer` will be returned.

```

// Read a string from another file synchronously
let content;
try {
  content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
  // An error occurred
  console.error(err);
}

```

Section 4.5: Check Permissions of a File or Directory

`fs.access()` determines whether a path exists and what permissions a user has to the file or directory at that path. `fs.access` doesn't return a result rather, if it doesn't return an error, the path exists and the user has the desired permissions.

The permission modes are available as a property on the `fs` object, `fs.constants`

- `fs.constants.F_OK` - Has read/write/execute permissions (If no mode is provided, this is the default)
- `fs.constants.R_OK` - Has read permissions
- `fs.constants.W_OK` - Has write permissions
- `fs.constants.X_OK` - Has execute permissions (Works the same as `fs.constants.F_OK` on Windows)

Asynchronously

```

var fs = require('fs');
var path = '/path/to/check';

// checks execute permission
fs.access(path, fs.constants.X_OK, (err) => {
  if (err) {
    console.log("%s doesn't exist", path);
  }
});

```

```

} else {
  console.log('可以执行 %s', path);
}
});
// 检查是否有读/写权限
// 指定多个权限模式时
// 每个模式用管道符 `|` 分隔
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
  if (err) {
    console.log("%s 不存在", path);
  } else {
    console.log('可以读/写 %s', path);
  }
});

```

同步方式

`fs.access` 也有同步版本 `fs.accessSync`。使用 `fs.accessSync` 时必须将其放在 `try/catch` 块中。

```

// 检查写权限
try {
  fs.accessSync(path, fs.constants.W_OK);
  console.log('可以写入 %s', path);
}
catch (err) {
  console.log("%s 不存在", path);
}

```

第4.6节：检查文件或目录是否存在

异步方式

```

var fs = require('fs');

fs.stat('path/to/file', function(err) {
  if (!err) {
    console.log('文件或目录存在');
  } else if (err.code === 'ENOENT') {
    console.log('文件或目录不存在');
  }
});

```

同步方式

这里，我们必须将函数调用包裹在 `try/catch` 块中以处理错误。

```

var fs = require('fs');

try {
  fs.statSync('path/to/file');
  console.log('文件或目录存在');
}
catch (err) {
  if (err.code === 'ENOENT') {
    console.log('文件或目录不存在');
  }
}

```

```

} else {
  console.log('can execute %s', path);
}
});
// Check if we have read/write permissions
// When specifying multiple permission modes
// each mode is separated by a pipe : '|'
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
  if (err) {
    console.log("%s doesn't exist", path);
  } else {
    console.log('can read/write %s', path);
  }
});

```

Synchronously

`fs.access` 也有同步版本 `fs.accessSync`。当使用 `fs.accessSync` 时，你必须将其放在 `try/catch` 块中。

```

// Check write permission
try {
  fs.accessSync(path, fs.constants.W_OK);
  console.log('can write %s', path);
}
catch (err) {
  console.log("%s doesn't exist", path);
}

```

Section 4.6: Checking if a file or a directory exists

Asynchronously

```

var fs = require('fs');

fs.stat('path/to/file', function(err) {
  if (!err) {
    console.log('file or directory exists');
  } else if (err.code === 'ENOENT') {
    console.log('file or directory does not exist');
  }
});

```

Synchronously

在这里，我们必须将函数调用包裹在 `try/catch` 块中以处理错误。

```

var fs = require('fs');

try {
  fs.statSync('path/to/file');
  console.log('file or directory exists');
}
catch (err) {
  if (err.code === 'ENOENT') {
    console.log('file or directory does not exist');
  }
}

```

第4.7节：确定文本文件的行数

app.js

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

var file = 'path.to.file';
var linesCount = 0;
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});
rl.on('line', function (line) {
  linesCount++; // 每遇到一个换行符，'linesCount'加1
});
rl.on('close', function () {
  console.log(linesCount); // 当触发'close'事件时打印结果
});
```

用法：

```
node app
```

第4.8节：逐行读取文件

app.js

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

var file = 'path.to.file';
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});

rl.on('line', function (line) {
  console.log(line); // 在每个换行处打印该行内容
});
```

用法：

```
node app
```

第4.9节：避免在创建或使用 已存在目录时发生竞态条件

由于Node的异步特性，通过以下方式创建或使用目录：

1. 先使用`fs.stat()`检查其是否存在，接着
2. 根据存在性检查的结果创建或使用该目录，

如果在检查和创建之间目录被创建，可能会导致“竞态条件”。

Section 4.7: Determining the line count of a text file

app.js

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

var file = 'path.to.file';
var linesCount = 0;
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});
rl.on('line', function (line) {
  linesCount++; // on each linebreak, add +1 to 'linesCount'
});
rl.on('close', function () {
  console.log(linesCount); // print the result when the 'close' event is called
});
```

Usage:

```
node app
```

Section 4.8: Reading a file line by line

app.js

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

var file = 'path.to.file';
var rl = readline.createInterface({
  input: fs.createReadStream(file),
  output: process.stdout,
  terminal: false
});

rl.on('line', function (line) {
  console.log(line); // print the content of the line on each linebreak
});
```

Usage:

```
node app
```

Section 4.9: Avoiding race conditions when creating or using an existing directory

Due to Node's asynchronous nature, creating or using a directory by first:

1. checking for its existence with `fs.stat()`, then
2. creating or using it depending of the results of the existence check,

can lead to a [race condition](#) if the folder is created between the time of the check and the time of the creation. The

下面的方法将 `fs.mkdir()` 和 `fs.mkdirSync()` 包装在错误捕获包装器中，如果异常的代码是 `EEXIST`（已存在），则允许异常通过。如果错误是其他类型，比如 `EPERM`（权限被拒绝），则像原生函数一样抛出或传递错误。

使用`fs.mkdir()`的异步版本

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

function mkdir (dirPath, callback) {
  fs.mkdir(dirPath, (err) => {
    callback(err && err.code !== 'EEXIST' ? err : null);
  });
}

mkdir('./existingDir', (err) => {
  if (err)
    return console.error(err.code);

  // 在这里对`./existingDir`进行操作
});

```

使用`fs.mkdirSync()`的同步版本

```
function mkdirSync (dirPath) {
  try {
    fs.mkdirSync(dirPath);
  } catch(e) {
    if (e.code !== 'EEXIST') throw e;
  }
}

mkdirSync('./existing-dir');
// 现在对`./existing-dir`进行操作

```

第4.10节：使用流克隆文件

本程序演示了如何使用文件系统模块提供的可读流和可写流，通过 `createReadStream()` 和 `createWriteStream()` 函数复制文件。

```
// 引入文件系统模块
var fs = require('fs');

/*
创建一个指向当前目录（_dirname）下名为 'node.txt' 文件的可读流
使用utf8编码
以16千字节为单位读取数据
*/
var readable = fs.createReadStream(_dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 * 1024 });

// 创建可写流
var writable = fs.createWriteStream(_dirname + '/nodeCopy.txt');

// 将每个数据块写入可写流
readable.on('data', function(chunk) {
  writable.write(chunk);
});
```

method below wraps `fs.mkdir()` and `fs.mkdirSync()` in error-catching wrappers that let the exception pass if its code is `EEXIST` (already exists). If the error is something else, like `EPERM` (permission denied), throw or pass an error like the native functions do.

Asynchronous version with `fs.mkdir()`

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

function mkdir (dirPath, callback) {
  fs.mkdir(dirPath, (err) => {
    callback(err && err.code !== 'EEXIST' ? err : null);
  });
}

mkdir('./existingDir', (err) => {
  if (err)
    return console.error(err.code);

  // Do something with `./existingDir` here
});

```

Synchronous version with `fs.mkdirSync()`

```
function mkdirSync (dirPath) {
  try {
    fs.mkdirSync(dirPath);
  } catch(e) {
    if (e.code !== 'EEXIST') throw e;
  }
}

mkdirSync('./existing-dir');
// Do something with `./existing-dir` now

```

Section 4.10: Cloning a file using streams

This program illustrates how one can copy a file using readable and writable streams using the `createReadStream()`, and `createWriteStream()` functions provided by the file system module.

```
//Require the file System module
var fs = require('fs');

/*
Create readable stream to file in current directory (_dirname) named 'node.txt'
Use utf8 encoding
Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(_dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(_dirname + '/nodeCopy.txt');

// Write each chunk of data to the writable stream
readable.on('data', function(chunk) {
  writable.write(chunk);
});
```

```
});
```

第4.11节：使用writeFile或writeFileSync写入文件

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

// 使用默认编码 (utf8) 将字符串"Hello world!"保存到目录"/tmp"中的文件"hello.txt"中。
// 该操作将在后台完成，回调函数将在操作完成或失败时被调用。
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {
  // 如果发生错误，显示错误并返回
  if(err) return console.error(err);
  // 成功写入文件！
});

// 将二进制数据保存到当前目录下名为"binary.txt"的文件中。同样，该操作将在后台完成。
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);
fs.writeFile('binary.txt', buffer, function(err) {
  // 如果发生错误，显示错误并返回
  if(err) return console.error(err);
  // 成功将二进制内容写入文件！
});
```

fs.writeFileSync的行为类似于fs.writeFile，但它不接受回调函数，因为它是同步完成的因此会阻塞主线程。大多数Node.js开发者更喜欢异步版本，因为它几乎不会导致程序执行延迟。

注意：阻塞主线程在 `node.js` 中是不好的做法。同步函数应仅在调试时或没有其他选项时使用。

```
// 将字符串写入另一个文件并将文件模式设置为 0755
try {
  fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
  // 发生错误
  console.error(err);
}
```

第 4.12 节：更改文本文件内容

示例。它将使用简单的正则表达式替换文本文件 `index.txt` 中的单词 `email` 为 `name`

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

fs.readFile('index.txt', 'utf-8', function(err, data) {
  if (err) throw err;

  var newValue = data.replace(/email/gim, 'name');

  fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
    if (err) throw err;
    console.log('完成！');
  })
});
```

```
});
```

Section 4.11: Writing to a file using writeFile or writeFileSync

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

// Save the string "Hello world!" in a file called "hello.txt" in
// the directory "/tmp" using the default encoding (utf8).
// This operation will be completed in background and the callback
// will be called when it is either done or failed.
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote to the file!
});

// Save binary data to a file called "binary.txt" in the current
// directory. Again, the operation will be completed in background.
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);
fs.writeFile('binary.txt', buffer, function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote binary contents to the file!
});
```

`fs.writeFileSync` behaves similarly to `fs.writeFile`, but does not take a callback as it completes synchronously and therefore blocks the main thread. Most node.js developers prefer the asynchronous variants which will cause virtually no delay in the program execution.

Note: Blocking the main thread is bad practice in node.js. Synchronous function should only be used when debugging or when no other options are available.

```
// Write a string to another file and set the file mode to 0755
try {
  fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Section 4.12: Changing contents of a text file

Example. It will be replacing the word `email` to a name in a text file `index.txt` with simple RegExp

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

fs.readFile('index.txt', 'utf-8', function(err, data) {
  if (err) throw err;

  var newValue = data.replace(/email/gim, 'name');

  fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
    if (err) throw err;
    console.log('Done!');
  })
});
```

第 4.13 节：使用 unlink 或 unlinkSync 删除文件

异步删除文件：

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

fs.unlink('/path/to/file.txt', function(err) {
  if (err) throw err;

  console.log('文件已删除');
});
```

你也可以同步删除*：

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

fs.unlinkSync('/path/to/file.txt');
console.log('文件已删除');
```

* 避免使用同步方法，因为它们会阻塞整个进程直到执行完成。

第4.14节：使用流将文件读取到缓冲区中

虽然使用fs.readFile()方法读取文件内容已经是异步的，但有时我们希望以流的形式获取数据，而不是简单的回调。这使我们能够将数据传输到其他位置，或者在数据到达时逐步处理，而不是等全部数据到齐后一次性处理。

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

// 将文件数据块存储在此数组中
let chunks = [];
// 我们可以使用此变量存储最终数据
let fileBuffer;

// 将文件读取到 stream.Readable 中
let fileStream = fs.createReadStream('text.txt');

// 流中发生错误
fileStream.once('error', (err) => {
  // 请务必正确处理此情况！
  console.error(err);
});

// 文件读取完成
fileStream.once('end', () => {
  // 从数据块创建最终的 Buffer;
  fileBuffer = Buffer.concat(chunks);

  // 当然，你也可以在这里做任何其他需要的操作，比如触发事件！
});

// 数据以块的形式从 fileStream 中刷新，
// 该回调函数将在每个数据块时执行
fileStream.on('data', (chunk) => {
  chunks.push(chunk); // 将数据块推入数组

  // 我们可以对目前已有的部分数据执行操作！
});
```

Section 4.13: Deleting a file using unlink or unlinkSync

Delete a file asynchronously:

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

fs.unlink('/path/to/file.txt', function(err) {
  if (err) throw err;

  console.log('file deleted');
});
```

You can also delete it synchronously*:

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

fs.unlinkSync('/path/to/file.txt');
console.log('file deleted');
```

* avoid synchronous methods because they block the entire process until the execution finishes.

Section 4.14: Reading a file into a Buffer using streams

While reading content from a file is already asynchronous using the `fs.readFile()` method, sometimes we want to get the data in a Stream versus in a simple callback. This allows us to pipe this data to other locations or to process it as it comes in versus all at once at the end.

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

// Store file data chunks in this array
let chunks = [];
// We can use this variable to store the final data
let fileBuffer;

// Read file into stream.Readable
let fileStream = fs.createReadStream('text.txt');

// An error occurred with the stream
fileStream.once('error', (err) => {
  // Be sure to handle this properly!
  console.error(err);
});

// File is done being read
fileStream.once('end', () => {
  // create the final data Buffer from data chunks;
  fileBuffer = Buffer.concat(chunks);

  // Of course, you can do anything else you need to here, like emit an event!
});

// Data is flushed from fileStream in chunks,
// this callback will be executed for each chunk
fileStream.on('data', (chunk) => {
  chunks.push(chunk); // push data chunk to array

  // We can perform actions on the partial data we have so far!
});
```

第5章：导出和使用模块

第5.1节：创建 hello-world.js 模块

Node 提供了 `module.exports` 接口，用于向其他文件暴露函数和变量。最简单的方式是只导出一个对象（函数或变量），如第一个示例所示。

hello-world.js

```
module.exports = function(subject) {
  console.log('Hello ' + subject);
};
```

如果我们不想让整个导出成为单个对象，可以将函数和变量作为 `exports` 对象的属性导出。以下三个示例都以稍有不同的方式演示了这一点：`exports` 对象。以下三个示例都以稍有不同的方式演示了这一点：

- `hello-venus.js`：函数定义先单独完成，然后作为 `module.exports` 的属性添加
- `hello-jupiter.js`：函数定义直接作为 `module.exports` 属性的值
- `hello-mars.js`：函数定义直接声明为 `exports` 的属性，`exports` 是 `module.exports` 的简写

hello-venus.js

```
function hello(subject) {
  console.log('金星说你好 ' + subject);
}

module.exports = {
  hello: hello
};
```

hello-jupiter.js

```
module.exports = {
  hello: function(subject) {
    console.log('木星说你好 ' + subject);
  },
  bye: function(subject) {
    console.log('木星说再见 ' + subject);
  }
};
```

hello-mars.js

```
exports.hello = function(subject) {
  console.log('火星说你好 ' + subject);
};
```

加载带目录名的模块

我们有一个名为hello的目录，里面包含以下文件：

index.js

Chapter 5: Exporting and Consuming Modules

Section 5.1: Creating a hello-world.js module

Node provides the `module.exports` interface to expose functions and variables to other files. The most simple way to do so is to export only one object (function or variable), as shown in the first example.

hello-world.js

```
module.exports = function(subject) {
  console.log('Hello ' + subject);
};
```

If we don't want the entire export to be a single object, we can export functions and variables as properties of the `exports` object. The three following examples all demonstrate this in slightly different ways :

- `hello-venus.js` : the function definition is done separately then added as a property of `module.exports`
- `hello-jupiter.js` : the functions definitions are directly put as the value of properties of `module.exports`
- `hello-mars.js` : the function definition is directly declared as a property of `exports` which is a short version of `module.exports`

hello-venus.js

```
function hello(subject) {
  console.log('Venus says Hello ' + subject);
}

module.exports = {
  hello: hello
};
```

hello-jupiter.js

```
module.exports = {
  hello: function(subject) {
    console.log('Jupiter says hello ' + subject);
  },
  bye: function(subject) {
    console.log('Jupiter says goodbye ' + subject);
  }
};
```

hello-mars.js

```
exports.hello = function(subject) {
  console.log('Mars says Hello ' + subject);
};
```

Loading module with directory name

We have a directory named `hello` which includes the following files:

index.js

```
// hello/index.js
module.exports = function(){
  console.log('Hej');
};
```

main.js

```
// hello/main.js
// 我们可以使用 `require()` 方法包含我们定义的其他文件
var hw = require('./hello-world.js'),
  hm = require('./hello-mars.js'),
  hv = require('./hello-venus.js'),
  hj = require('./hello-jupiter.js'),
  hu = require('./index.js');

// 因为我们将函数赋值给了整个 `module.exports` 对象,
// 所以可以直接使用它
hw('World!'); // 输出 "Hello World!"

// 在这种情况下, 我们将函数赋值给了 exports 的 `hello` 属性,
// 所以这里也必须使用该属性
hm.hello('Solar System!'); // 输出 "Mars says Hello Solar System!"

// 一次性赋值 module.exports 的结果与 hello-world.js 中相同
hv.hello('Milky Way!'); // 输出 "Venus says Hello Milky Way!"

hj.hello('Universe!'); // 输出 "Jupiter says hello Universe!"
hj.bye('Universe!'); // 输出 "Jupiter says goodbye Universe!"

hu(); // 输出 'hej'
```

第5.2节：加载和使用模块

模块可以通过`require()`函数“导入”或“引用”。例如，要加载Node.js自带的`http`模块，可以使用以下代码：

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

除了运行时自带的模块外，你还可以引用从npm安装的模块，比如`express`。如果你已经通过`npm install express`安装了`express`，可以直接写：

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

你也可以包含自己编写的模块作为应用程序的一部分。在这种情况下，要包含与当前文件同目录下名为`lib.js`的文件：

```
const mylib = require('./lib');
```

注意你可以省略扩展名，默认会假设为`.js`。加载模块后，该变量会被赋值为一个对象，包含从所引用文件导出的属性和方法。完整示例：

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

// `http` 模块有属性 `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // 输出 'Not Found'
```

```
// hello/index.js
module.exports = function(){
  console.log('Hej');
};
```

main.js

```
// hello/main.js
// We can include the other files we've defined by using the `require()` method
var hw = require('./hello-world.js'),
  hm = require('./hello-mars.js'),
  hv = require('./hello-venus.js'),
  hj = require('./hello-jupiter.js'),
  hu = require('./index.js');

// Because we assigned our function to the entire `module.exports` object, we
// can use it directly
hw('World!'); // outputs "Hello World!"

// In this case, we assigned our function to the `hello` property of exports, so we must
// use that here too
hm.hello('Solar System!'); // outputs "Mars says Hello Solar System!"

// The result of assigning module.exports at once is the same as in hello-world.js
hv.hello('Milky Way!'); // outputs "Venus says Hello Milky Way!"

hj.hello('Universe!'); // outputs "Jupiter says hello Universe!"
hj.bye('Universe!'); // outputs "Jupiter says goodbye Universe!"

hu(); // output 'hej'
```

Section 5.2: Loading and using a module

A module can be "imported", or otherwise "required" by the `require()` function. For example, to load the `http` module that ships with Node.js, the following can be used:

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

Aside from modules that are shipped with the runtime, you can also require modules that you have installed from npm, such as `express`. If you had already installed `express` on your system via `npm install express`, you could simply write:

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

You can also include modules that you have written yourself as part of your application. In this case, to include a file named `lib.js` in the same directory as current file:

```
const mylib = require('./lib');
```

Note that you can omit the extension, and `.js` will be assumed. Once you load a module, the variable is populated with an object that contains the methods and properties published from the required file. A full example:

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

// The `http` module has the property `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // outputs 'Not Found'
```

```
// 还包含 `createServer()`
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<html><body>模块测试</body></html>');
  res.end();
}).listen(80);
```

第5.3节：文件夹作为模块

模块可以分布在同一文件夹中的多个.js文件中。以下是在my_module文件夹中的示例：

function_one.js

```
module.exports = function() {
  return 1;
}
```

function_two.js

```
module.exports = function() {
  return 2;
}
```

index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

像这样的模块通过文件夹名称来引用使用：

```
var split_module = require('./my_module');
```

请注意，如果你在 require 函数的参数中省略了 ./ 或任何指向文件夹路径的标识，Node 会尝试从 node_modules 文件夹加载模块。

或者你可以在同一文件夹中创建一个 package.json 文件，内容如下：

```
{
  "name": "my_module",
  "main": "./your_main_entry_point.js"
}
```

这样你就不必将主模块文件命名为 "index"。

第5.4节：每个模块只注入一次

NodeJS 只在第一次 require 模块时执行该模块。之后的任何 require 调用都会重用同一个对象，因此不会再次执行模块中的代码。同时，Node 会缓存模块首次通过 require 加载时的内容。这减少了文件读取次数，有助于加快应用程序的速度。

myModule.js

```
console.log(123);
exports.var1 = 4;
```

```
// Also contains `createServer()`
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<html><body>Module Test</body></html>');
  res.end();
}).listen(80);
```

Section 5.3: Folder as a module

Modules can be split across many .js files in the same folder. An example in a *my_module* folder:

function_one.js

```
module.exports = function() {
  return 1;
}
```

function_two.js

```
module.exports = function() {
  return 2;
}
```

index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

A module like this one is used by referring to it by the folder name:

```
var split_module = require('./my_module');
```

Please note that if you required it by omitting ./ or any indication of a path to a folder from the require function argument, Node will try to load a module from the *node_modules* folder.

Alternatively you can create in the same folder a package.json file with these contents:

```
{
  "name": "my_module",
  "main": "./your_main_entry_point.js"
}
```

This way you are not required to name the main module file "index".

Section 5.4: Every module injected only once

NodeJS executes the module only the first time you require it. Any further require functions will re-use the same Object, thus not executing the code in the module another time. Also Node caches the modules first time they are loaded using require. This reduces the number of file reads and helps to speed up the application.

myModule.js

```
console.log(123);
exports.var1 = 4;
```

index.js

```
var a=require('./myModule') ; // 输出 123
var b=require('./myModule') ; // 无输出
console.log(a.var1) ; // 输出 4
console.log(b.var1) ; // 输出 4
a.var2 = 5 ;
console.log(b.var2) ; // 输出 5
```

第5.5节：从 node_modules 加载模块

模块可以通过放置在一个名为 node_modules 的特殊目录中，require 时无需使用相对路径。

例如，要从文件 index.js 中 require 一个名为 foo 的模块，可以使用以下目录结构：

```
index.js
\-\ node_modules
  \-\ foo
    \-\ foo.js
    \-\ package.json
```

模块应放置在一个目录内，并包含一个 package.json 文件。 package.json 文件中的 main 字段应指向模块的入口文件——当用户执行 require('your-module') 时导入的文件。如果未提供，main 默认为 index.js。或者，你也可以通过在 require 调用中附加相对路径来引用模块内的文件： require('your-module/path/to/file')。

模块也可以从文件系统层级向上的 node_modules 目录中被 require。如果我们有以下目录结构：

```
我的项目
- node_modules
  | - foo  // foo 模块
  |   ...
- baz  // baz 模块
  - node_modules
- bar  // bar 模块
```

我们将能够从 bar 中的任何文件使用 require('foo') 来 require foo 模块。

注意，node 只会匹配文件系统层级中离文件最近的模块，从（文件当前目录/node_modules）开始。Node 会以这种方式匹配目录直到文件系统根目录。

你可以从 npm 注册表或其他 npm 注册表安装新模块，或者自己制作模块。

第 5.6 节：构建你自己的模块

你也可以引用一个对象来公开导出，并持续向该对象追加方法：

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // 使用电子邮件地址进行身份验证
}

auth.facebook = function (data, callback) {
```

index.js

```
var a=require('./myModule') ; // Output 123
var b=require('./myModule') ; // No output
console.log(a.var1) ; // Output 4
console.log(b.var1) ; // Output 4
a.var2 = 5 ;
console.log(b.var2) ; // Output 5
```

Section 5.5: Module loading from node_modules

Modules can be required without using relative paths by putting them in a special directory called node_modules.

For example, to require a module called foo from a file index.js, you can use the following directory structure:

```
index.js
\-\ node_modules
  \-\ foo
    \-\ foo.js
    \-\ package.json
```

Modules should be placed inside a directory, along with a package.json file. The main field of the package.json file should point to the entry point for your module--this is the file that is imported when users do require('your-module').main defaults to index.js if not provided. Alternatively, you can refer to files relative to your module simply by appending the relative path to the require call: require('your-module/path/to/file').

Modules can also be required from node_modules directories up the file system hierarchy. If we have the following directory structure:

```
my-project
\-\ node_modules
  \-\ foo  // the foo module
  \-\ ...
\-\ baz  // the baz module
  \-\ node_modules
    \-\ bar  // the bar module
```

we will be able to require the module foo from any file within bar using require('foo').

Note that node will only match the module that is closest to the file in the filesystem hierarchy, starting from (the file's current directory/node_modules). Node matches directories this way up to the file system root.

You can either install new modules from the npm registry or other npm registries, or make your own.

Section 5.6: Building your own modules

You can also reference an object to publicly export and continuously append methods to that object:

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // Authenticate with an email address
}

auth.facebook = function (data, callback) {
```

```
// 使用 Facebook 账号进行身份验证  
}  
  
auth.twitter = function (data, callback) {  
  // 使用 Twitter 账号进行身份验证  
}  
  
auth.slack = function (data, callback) {  
  // 使用 Slack 账号进行身份验证  
}  
  
auth.stack_overflow = function (data, callback) {  
  // 使用 Stack Overflow 账号进行身份验证  
}
```

要使用这些，只需像平常一样引入该模块：

```
const auth = require('./auth')  
  
module.exports = function (req, res, next) {  
  auth.facebook(req.body, function (err, user) {  
    if (err) return next(err)  
  
    req.user = user  
    next()  
  })  
}
```

第5.7节：使模块缓存失效

在开发过程中，你可能会发现对同一个模块多次使用require()总是返回相同的模块，即使你对该文件进行了修改。这是因为模块在第一次加载时会被缓存，之后的任何模块加载都会从缓存中加载。

为了解决这个问题，你需要删除缓存中的条目。例如，如果你加载了一个模块：

```
var a = require('./a');
```

然后你可以删除缓存条目：

```
var rpath = require.resolve('./a.js');  
delete require.cache[rpath];
```

然后再次require该模块：

```
var a = require('./a');
```

请注意，这在生产环境中不推荐使用，因为delete只会删除对已加载模块的引用，而不会删除已加载的数据本身。模块不会被垃圾回收，因此不当使用此功能可能导致内存泄漏。

```
// Authenticate with a Facebook account  
}  
  
auth.twitter = function (data, callback) {  
  // Authenticate with a Twitter account  
}  
  
auth.slack = function (data, callback) {  
  // Authenticate with a Slack account  
}  
  
auth.stack_overflow = function (data, callback) {  
  // Authenticate with a Stack Overflow account  
}
```

To use any of these, just require the module as you normally would:

```
const auth = require('./auth')  
  
module.exports = function (req, res, next) {  
  auth.facebook(req.body, function (err, user) {  
    if (err) return next(err)  
  
    req.user = user  
    next()  
  })  
}
```

Section 5.7: Invalidating the module cache

In development, you may find that using require() on the same module multiple times always returns the same module, even if you have made changes to that file. This is because modules are cached the first time they are loaded, and any subsequent module loads will load from the cache.

To get around this issue, you will have to **delete** the entry in the cache. For example, if you loaded a module:

```
var a = require('./a');
```

You could then delete the cache entry:

```
var rpath = require.resolve('./a.js');  
delete require.cache[rpath];
```

And then require the module again:

```
var a = require('./a');
```

Do note that this is not recommended in production because the **delete** will only delete the reference to the loaded module, not the loaded data itself. The module is not garbage collected, so improper use of this feature could lead to leaking memory.

第6章：在node.js中导出和导入模块

第6.1节：使用ES6语法导出

这是另一个示例的等效写法，但使用的是 ES6。

```
export function printHelloWorld() {  
    console.log("Hello World!!!");  
}
```

第6.2节：在node.js中使用简单模块

什么是node.js模块 ([文章链接](#)) :

模块将相关代码封装成一个代码单元。创建模块时，可以理解为将所有相关函数移入一个文件中。

现在来看一个例子。假设所有文件都在同一目录下：

文件：printer.js

```
"use strict";  
  
exports.printHelloWorld = function (){  
    console.log("Hello World!!!");  
}
```

使用模块的另一种方式：

文件 animals.js

```
"use strict";  
  
module.exports = {  
    lion: function() {  
        console.log("ROAARR!!!");  
    }  
};
```

文件：app.js

通过进入你的目录并输入以下命令运行此文件：node app.js

```
"use strict";  
  
//require('./path/to/module.js') 指定 node 加载哪个模块  
var printer = require('./printer');  
var animals = require('./animals');  
  
printer.printHelloWorld(); //打印 "Hello World!!!"  
animals.lion(); //打印 "ROAARR!!!"
```

Chapter 6: Exporting and Importing Module in node.js

Section 6.1: Exporting with ES6 syntax

This is the equivalent of the other example but using ES6 instead.

```
export function printHelloWorld() {  
    console.log("Hello World!!!");  
}
```

Section 6.2: Using a simple module in node.js

What is a node.js module ([link to article](#)):

A module encapsulates related code into a single unit of code. When creating a module, this can be interpreted as moving all related functions into a file.

Now lets see an example. Imagine all files are in same directory:

File: printer.js

```
"use strict";  
  
exports.printHelloWorld = function (){  
    console.log("Hello World!!!");  
}
```

Another way of using modules:

File animals.js

```
"use strict";  
  
module.exports = {  
    lion: function() {  
        console.log("ROAARR!!!");  
    }  
};
```

File: app.js

Run this file by going to your directory and typing: node app.js

```
"use strict";  
  
//require('./path/to/module.js') node which module to load  
var printer = require('./printer');  
var animals = require('./animals');  
  
printer.printHelloWorld(); //prints "Hello World!!!"  
animals.lion(); //prints "ROAARR!!!"
```

第7章：模块是如何加载的

第7.1节：全局模式

如果你使用默认目录安装了 Node，在全局模式下，NPM 会将包安装到 /usr/local/lib/node_modules。如果你在 shell 中输入以下命令，NPM 会在目录/usr/local/lib/node_modules/express中搜索、下载并安装名为 sax 的包的最新版本：

```
$ npm install -g express
```

确保您对该文件夹有足够的访问权限。这些模块将对该机器上运行的所有节点进程可用

在本地模式安装中，Npm 会通过创建一个名为 node_modules的新文件夹，在当前工作文件夹中下载并安装模块。例如，如果您位于 /home/user/apps/my_app，则会创建一个名为 node_modules/home/user/apps/my_app/node_modules的新文件夹（如果尚不存在）

第7.2节：加载模块

当我们在代码中引用模块时，Node 会首先在 require 语句中引用的文件夹内查找 node_module文件夹。如果模块名不是相对路径且不是核心模块，Node 会尝试在当前目录的 node_modules文件夹中查找。例如，如果您执行以下操作，Node 会尝试查找文件

```
./node_modules/myModule.js :
```

```
var myModule = require('myModule.js');
```

如果 Node 未能找到该文件，它会在父文件夹中查找..../node_modules/myModule.js。如果仍未找到，它会继续向上查找父文件夹，直到到达根目录或找到所需模块为止。

如果您愿意，也可以省略.js扩展名，这种情况下，Node 会自动添加.js扩展名并搜索该文件。

加载文件夹模块

您可以使用文件夹路径来加载模块，方法如下：

```
var myModule = require('./myModuleDir');
```

如果这样做，Node 将在该文件夹内搜索。Node 会假定该文件夹是一个包，并尝试查找包定义。该包定义应是一个名为 package.json的文件。如果该文件夹不包含名为package.json的包定义文件，包入口点将假定默认值为index.js，Node 在这种情况下会在路径./myModuleDir/index.js下查找文件。

如果模块在任何文件夹中都未找到，最后的选择是全局模块安装文件夹。

Chapter 7: How modules are loaded

Section 7.1: Global Mode

If you installed Node using the default directory, while in the global mode, NPM installs packages into /usr/local/lib/node_modules. If you type the following in the shell, NPM will search for, download, and install the latest version of the package named sax inside the directory /usr/local/lib/node_modules/express:

```
$ npm install -g express
```

Make sure that you have sufficient access rights to the folder. These modules will be available for all node process which will be running in that machine

In local mode installation. Npm will down load and install modules in the current working folders by creating a new folder called node_modules for example if you are in /home/user/apps/my_app a new folder will be created called node_modules /home/user/apps/my_app/node_modules if its not already exist

Section 7.2: Loading modules

When we refer the module in the code, node first looks up the node_module folder inside the referenced folder in required statement If the module name is not relative and is not a core module, Node will try to find it inside the node_modules folder in the current directory. For instance, if you do the following, Node will try to look for the file ./node_modules/myModule.js:

```
var myModule = require('myModule.js');
```

If Node fails to find the file, it will look inside the parent folder called ../node_modules/myModule.js. If it fails again, it will try the parent folder and keep descending until it reaches the root or finds the required module.

You can also omit the .js extension if you like to, in which case node will append the .js extension and will search for the file.

Loading a Folder Module

You can use the path for a folder to load a module like this:

```
var myModule = require('./myModuleDir');
```

If you do so, Node will search inside that folder. Node will presume this folder is a package and will try to look for a package definition. That package definition should be a file named package.json. If that folder does not contain a package definition file named package.json, the package entry point will assume the default value of index.js, and Node will look, in this case, for a file under the path ./myModuleDir/index.js.

The last resort if module is not found in any of the folders is the global module installation folder.

第8章：集群模块

第8.1节：Hello World

这是你的cluster.js文件：

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // 创建工作进程。
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} 退出`);
  });
} else {
  // 工作线程可以共享任何 TCP 连接
  // 在此情况下它是一个 HTTP 服务器
  require('./server.js')();
}
```

这是你的主server.js文件：

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

function startServer() {
  const server = http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello Http');
  });

  server.listen(3000);
}

if(!module.parent) {
  // 如果文件被直接运行则启动服务器
  startServer();
} else {
  // 如果文件通过集群引用，则导出服务器
  module.exports = startServer;
}
```

在此示例中，我们托管了一个基本的网络服务器，但我们使用内置的**cluster**模块启动了工作进程（子进程）。进程的数量取决于可用的CPU核心数。这使得Node.js应用程序能够利用多核CPU，因为单个Node.js实例运行在单线程中。该应用程序现在将在所有进程之间共享端口8000。负载将默认使用轮询（Round-Robin）方法自动分配给各个工作进程。

第8.2节：集群示例

单个Node.js实例运行在单线程中。为了利用多核系统，应用程序可以在Node.js进程集群中启动以处理负载。

Chapter 8: Cluster Module

Section 8.1: Hello World

This is your `cluster.js`:

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  require('./server.js')();
}
```

This is your main `server.js`:

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

function startServer() {
  const server = http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello Http');
  });

  server.listen(3000);
}

if(!module.parent) {
  // Start server if file is run directly
  startServer();
} else {
  // Export server, if file is referenced via cluster
  module.exports = startServer;
}
```

In this example, we host a basic web server, however, we spin up workers (child processes) using the built-in **cluster** module. The number of processes forked depend on the number of CPU cores available. This enables a Node.js application to take advantage of multi-core CPUs, since a single instance of Node.js runs in a single thread. The application will now share the port 8000 across all the processes. Loads will automatically be distributed between workers using the Round-Robin method by default.

Section 8.2: Cluster Example

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems, application can be launched in a cluster of Node.js processes to handle the load.

cluster模块允许你轻松创建共享服务器端口的子进程。

以下示例在主进程中创建工作子进程，以跨多个核心处理负载。

示例

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; // CPU数量

if (cluster.isMaster) {
  // 创建工作进程。
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork(); // 创建子进程
  }

  // 在集群退出时
  cluster.on('exit', (worker, code, signal) => {
    if (signal) {
      console.log(`worker was killed by signal: ${signal}`);
    } else if (code !== 0) {
      console.log(`worker exited with error code: ${code}`);
    } else {
      console.log('worker success!');
    }
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

The `cluster` module allows you to easily create child processes that all share server ports.

Following example create the worker child process in main process that handles the load across multiple cores.

Example

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; //number of CPUS

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork(); //creating child process
  }

  //on exit of cluster
  cluster.on('exit', (worker, code, signal) => {
    if (signal) {
      console.log(`worker was killed by signal: ${signal}`);
    } else if (code !== 0) {
      console.log(`worker exited with error code: ${code}`);
    } else {
      console.log('worker success!');
    }
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

第9章：Readline

第9.1节：逐行读取文件

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

const rl = readline.createInterface({
  input: fs.createReadStream('text.txt')
});

// 每当流接收到 |r|、|n| 或 |r\n| 时，每一新行都会触发一个事件
rl.on('line', (line) => {
  console.log(line);
});

rl.on('close', () => {
  console.log('读取文件完成');
});
```

第9.2节：通过命令行界面提示用户输入

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('你叫什么名字?', (name) => {
  console.log(`你好, ${name}!`);

  rl.close();
});
```

Chapter 9: Readline

Section 9.1: Line-by-line file reading

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

const rl = readline.createInterface({
  input: fs.createReadStream('text.txt')
});

// Each new line emits an event - every time the stream receives |r|, |n|, or |r\n|
rl.on('line', (line) => {
  console.log(line);
});

rl.on('close', () => {
  console.log('Done reading file');
});
```

Section 9.2: Prompting user input via CLI

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

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('What is your name?', (name) => {
  console.log(`Hello ${name}!`);

  rl.close();
});
```

第10章：package.json

第10.1节：探索 package.json

一个package.json文件，通常存在于项目根目录，包含有关您的应用或模块的元数据，以及运行 `npm install`时需要从npm安装的依赖列表。

要初始化一个package.json类型，在命令提示符中使用`npm init`。

要创建一个带有默认值的package.json，请使用：

```
npm init --yes  
# 或者  
npm init -y
```

要安装一个包并保存到package.json中，请使用：

```
npm install {package name} --save
```

你也可以使用简写形式：

```
npm i -S {package name}
```

NPM将-S别名为--save，将-D别名为--save-dev，分别保存到生产依赖或开发依赖中。

该包将出现在你的dependencies中；如果使用--save-dev代替--save，包将出现在你的devDependencies中。

package.json的重要属性：

```
{  
  "name": "module-name",  
  "version": "10.3.1",  
  "description": "一个用于说明 package.json 用法的示例模块",  
  "author": "你的名字 <your.name@example.org>",  
  "contributors": [{  
    "name": "Foo Bar",  
    "email": "foo.bar@example.com"  
  }],  
  "bin": {  
    "module-name": "./bin/module-name"  
  },  
  "scripts": {  
    "test": "vows --spec --isolate",  
    "start": "node index.js",  
    "predeploy": "echo About to deploy",  
    "postdeploy": "echo Deployed",  
    "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"  
  },  
  "main": "lib/foo.js",  
  "repository": {  
    "type": "git",  
    "url": "https://github.com/username/repo"  
  },  
  "bugs": {  
    "url": "https://github.com/username/issues"  
  }
```

Chapter 10: package.json

Section 10.1: Exploring package.json

A package `.json` file, usually present in the project root, contains metadata about your app or module as well as the list of dependencies to install from npm when running `npm install`.

To initialize a package `.json` type `npm init` in your command prompt.

To create a package `.json` with default values use:

```
npm init --yes  
# or  
npm init -y
```

To install a package and save it to package `.json` use:

```
npm install {package name} --save
```

You can also use the shorthand notation:

```
npm i -S {package name}
```

NPM aliases `-S` to `--save` and `-D` to `--save-dev` to save in your production or development dependencies respectively.

The package will appear in your dependencies; if you use `--save-dev` instead of `--save`, the package will appear in your devDependencies.

Important properties of package `.json`:

```
{  
  "name": "module-name",  
  "version": "10.3.1",  
  "description": "An example module to illustrate the usage of a package.json",  
  "author": "Your Name <your.name@example.org>",  
  "contributors": [{  
    "name": "Foo Bar",  
    "email": "foo.bar@example.com"  
  }],  
  "bin": {  
    "module-name": "./bin/module-name"  
  },  
  "scripts": {  
    "test": "vows --spec --isolate",  
    "start": "node index.js",  
    "predeploy": "echo About to deploy",  
    "postdeploy": "echo Deployed",  
    "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"  
  },  
  "main": "lib/foo.js",  
  "repository": {  
    "type": "git",  
    "url": "https://github.com/username/repo"  
  },  
  "bugs": {  
    "url": "https://github.com/username/issues"  
  }
```

```

},
"keywords": [
  "example"
],
"dependencies": {
  "express": "4.2.x"
},
"devDependencies": {
  "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
},
"peerDependencies": {
  "moment": ">2.0.0"
},
"preferGlobal": true,
"private": true,
"publishConfig": {
  "registry": "https://your-private-hosted-npm.registry.domain.com"
},
"subdomain": "foobar",
"analyze": true,
"license": "MIT",
"files": [
  "lib/foo.js"
]
}

```

关于一些重要属性的信息：

名称

您包的唯一名称，应全部小写。此属性是必需的，缺少它您的包将无法安装。

1. 名称必须小于或等于214个字符。
2. 名称不能以点或下划线开头。
3. 新的包名称中不得包含大写字母。

版本

该软件包的版本由语义化版本控制 (semver) 指定。其假设版本号的写法为 MAJOR.MINOR.PATCH，且你需要递增：

1. 当你进行不兼容的 API 更改时递增 MAJOR 版本
2. 当你以向后兼容的方式添加功能时递增 MINOR 版本
3. 当你进行向后兼容的错误修复时递增 PATCH 版本

描述

项目的描述。尽量保持简短和简洁。

作者

该软件包的作者。

二进制文件

一个用于暴露软件包中二进制脚本的对象。该对象假设键是二进制脚本的名称

```

},
"keywords": [
  "example"
],
"dependencies": {
  "express": "4.2.x"
},
"devDependencies": {
  "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
},
"peerDependencies": {
  "moment": ">2.0.0"
},
"preferGlobal": true,
"private": true,
"publishConfig": {
  "registry": "https://your-private-hosted-npm.registry.domain.com"
},
"subdomain": "foobar",
"analyze": true,
"license": "MIT",
"files": [
  "lib/foo.js"
]
}

```

Information about some important properties:

name

The unique name of your package and should be down in lowercase. This property is required and your package will not install without it.

1. The name must be less than or equal to 214 characters.
2. The name can't start with a dot or an underscore.
3. New packages must not have uppercase letters in the name.

version

The version of the package is specified by [Semantic Versioning](#) (semver). Which assumes that a version number is written as MAJOR.MINOR.PATCH and you increment the:

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backwards-compatible manner
3. PATCH version when you make backwards-compatible bug fixes

description

The description of the project. Try to keep it short and concise.

author

The author of this package.

bin

An object which is used to expose binary scripts from your package. The object assumes that the key is the name of

值是脚本的相对路径。

该属性用于包含命令行界面（CLI）的软件包。

脚本

一个暴露额外 npm 命令的对象。该对象假设键是 npm 命令，值是脚本路径。当你运行 `npm run {command name}` 或 `npm run-script {commandname}` 时，这些脚本可以被执行。

包含命令行界面的本地安装包可以无需相对路径调用。因此，你可以直接调用 mocha，而不是调用 `./node-modules/.bin/mocha`。

主入口

包的主入口点。当在 node 中调用 `require('{module name}')` 时，实际被加载的就是这个文件。

强烈建议 require 主文件时不要产生任何副作用。例如，require 主文件时不应启动 HTTP 服务器或连接数据库。相反，你应该创建类似 `exports.init = function () {...}` 这样的函数在你的主脚本中。

关键词

描述你包的关键词数组。这些关键词将帮助人们找到你的包。

开发依赖

这些依赖项仅用于模块的开发和测试。除非设置了 `NODE_ENV=production` 环境变量，否则这些依赖项将自动安装。如果是这种情况，您仍然可以使用 `npm install --dev` 安装这些包。

peerDependencies

如果您正在使用此模块，则 peerDependencies 列出了您必须与此模块一起安装的模块。例如，`moment-timezone` 必须与 `moment` 一起安装，因为它是 `moment` 的一个插件，即使它不直接 `require("moment")`。

preferGlobal

一个属性，表示此页面更倾向于使用 `npm install -g {module-name}` 全局安装。此属性用于包含 CLI（命令行界面）的包。

在所有其他情况下，您不应使用此属性。

publishConfig

`publishConfig` 是一个包含发布模块时使用的配置值的对象。设置的配置值会覆盖您默认的 npm 配置。

`publishConfig` 最常见的用途是将您的包发布到私有 npm 注册表，这样您仍然可以享受 npm 的好处，但用于私有包。只需将私有 npm 的 URL 设置为 `registry` 键的值即可实现。

the binary script and the value a relative path to the script.

This property is used by packages that contain a CLI (command line interface).

script

A object which exposes additional npm commands. The object assumes that the key is the npm command and the value is the script path. These scripts can get executed when you run `npm run {command name}` or `npm run-script {command name}`.

Packages that contain a command line interface and are installed locally can be called without a relative path. So instead of calling `./node-modules/.bin/mocha` you can directly call `mocha`.

main

The main entry point to your package. When calling `require(' {module name}')` in node, this will be actual file that is required.

It's highly advised that requiring the main file does not generate any side affects. For instance, requiring the main file should not start up a HTTP server or connect to a database. Instead, you should create something like `exports.init = function () {...}` in your main script.

keywords

An array of keywords which describe your package. These will help people find your package.

devDependencies

These are the dependencies that are only intended for development and testing of your module. The dependencies will be installed automatically unless the `NODE_ENV=production` environment variable has been set. If this is the case you can still these packages using `npm install --dev`

peerDependencies

If you are using this module, then peerDependencies lists the modules you must install alongside this one. For example, `moment-timezone` must be installed alongside `moment` because it is a plugin for `moment`, even if it doesn't directly `require("moment")`.

preferGlobal

A property that indicates that this page prefers to be installed globally using `npm install -g {module-name}`. This property is used by packages that contain a CLI (command line interface).

In all other situations you should NOT use this property.

publishConfig

The publishConfig is an object with configuration values that will be used for publishing modules. The configuration values that are set override your default npm configuration.

The most common use of the publishConfig is to publish your package to a private npm registry so you still have the benefits of npm but for private packages. This is done by simply setting URL of your private npm as value for the registry key.

files

这是一个包含所有要包含在发布包中的文件的数组。可以使用文件路径或文件夹路径。文件夹路径的所有内容都会被包含。这通过仅包含要分发的正确文件来减少包的总体大小。此字段与 `.npmignore` 规则文件配合使用。

来源

第10.2节：脚本

您可以定义在另一个脚本之前或之后执行或触发的脚本。

```
{  
  "scripts": {  
    "pretest": "scripts/pretest.js",  
    "test": "scripts/test.js",  
    "posttest": "scripts/posttest.js"  
  }  
}
```

在这种情况下，您可以通过运行以下任一命令来执行脚本：

```
$ npm run-script test  
$ npm run test  
$ npm test  
$ npm t
```

预定脚本

脚本名称	描述
prepublish	在软件包发布之前运行。
发布, 发布后	在包发布后运行。
安装前	在包安装前运行。
安装, 安装后	在包安装后运行。
卸载前, 卸载	在包卸载前运行。
卸载后	在包卸载后运行。
预版本, 版本	在提升包版本之前运行。
后版本	在提升包版本之后运行。
预测试, 测试, 后测试	由 <code>npm test</code> 命令运行
预停止, 停止, 后停止	由 <code>npm stop</code> 命令运行
预启动, 启动, 后启动	由 <code>npm start</code> 命令运行
预重启、重启、重启后	由 <code>npm restart</code> 命令运行

用户自定义脚本

你也可以像使用预定脚本一样定义你自己的脚本：

```
{  
  "scripts": {  
    "preci": "scripts/preci.js",  
    "ci": "scripts/ci.js",  
    "postci": "scripts/postci.js"  
  }  
}
```

files

This is an array of all the files to include in the published package. Either a file path or folder path can be used. All the contents of a folder path will be included. This reduces the total size of your package by only including the correct files to be distributed. This field works in conjunction with a `.npmignore` rules file.

Source

Section 10.2: Scripts

You can define scripts that can be executed or are triggered before or after another script.

```
{  
  "scripts": {  
    "pretest": "scripts/pretest.js",  
    "test": "scripts/test.js",  
    "posttest": "scripts/posttest.js"  
  }  
}
```

In this case, you can execute the script by running either of these commands:

```
$ npm run-script test  
$ npm run test  
$ npm test  
$ npm t
```

Pre-defined scripts

Script Name	Description
prepublish	Run before the package is published.
publish, postpublish	Run after the package is published.
preinstall	Run before the package is installed.
install, postinstall	Run after the package is installed.
preuninstall, uninstall	Run before the package is uninstalled.
postuninstall	Run after the package is uninstalled.
preversion, version	Run before bump the package version.
postversion	Run after bump the package version.
pretest, test, posttest	Run by the <code>npm test</code> command
prestop, stop, poststop	Run by the <code>npm stop</code> command
prestart, start, poststart	Run by the <code>npm start</code> command
prerestart, restart, postrestart	Run by the <code>npm restart</code> command

User-defined scripts

You can also define your own scripts the same way you do with the pre-defined scripts:

```
{  
  "scripts": {  
    "preci": "scripts/preci.js",  
    "ci": "scripts/ci.js",  
    "postci": "scripts/postci.js"  
  }  
}
```

在这种情况下，您可以通过运行以下任一命令来执行脚本：

```
$ npm run-script ci  
$ npm run ci
```

用户自定义脚本同样支持pre和post脚本，如上例所示。

第10.3节：基本项目定义

```
{  
  "name": "my-project",  
  "version": "0.0.1",  
  "description": "这是一个项目。",  
  "author": "某人 <someone@example.com>",  
  "contributors": [  
    {"name": "另一个人",  
     "email": "else@example.com"}],  
  "keywords": ["改进", "搜索"]  
}
```

字段 描述

名称	一个必填字段，用于安装包。需要小写，单词且无空格。（允许使用连字符和下划线）
版本	一个必填字段，表示包版本，使用语义化版本控制。
描述项目的简短说明\作者	指定包的作者
贡献者	一个对象数组，每个贡献者一个
关键词	字符串数组，有助于人们找到你的包

第10.4节：依赖项

```
"dependencies": { "module-name": "0.1.0" }
```

- exact: 0.1.0 将安装该模块的特定版本。
- newest minor version: ^0.1.0 将安装最新的小版本，例如0.2.0，但不会安装主版本更高的模块，例如1.0.0
- newest patch: 0.1.x 或 ~0.1.0 将安装最新的补丁版本，例如0.1.4，但不会安装主版本或小版本更高的模块，例如0.2.0或1.0.0。
- wildcard: * 将安装该模块的最新版本。
- **git repository**: 以下命令将从git仓库的master分支安装tar包。也可以提供#sha、#tag或#branch：
 - GitHub: user/project 或 user/project#v1.0.0
 - url: git://gitlab.com/user/project.git 或 git://gitlab.com/user/project.git#develop
- **local path**: file:./lib/project

将它们添加到你的package.json后，在终端的项目目录中使用命令 `npm install`。

开发依赖

```
"devDependencies": {  
  "module-name": "0.1.0"  
}
```

仅在开发时需要的依赖，如测试、样式代理等。这些开发依赖不会被

In this case, you can execute the script by running either of these commands:

```
$ npm run-script ci  
$ npm run ci
```

User-defined scripts also supports *pre* and *post* scripts, as shown in the example above.

Section 10.3: Basic project definition

```
{  
  "name": "my-project",  
  "version": "0.0.1",  
  "description": "This is a project.",  
  "author": "Someone <someone@example.com>",  
  "contributors": [  
    {"name": "Someone Else",  
     "email": "else@example.com"}],  
  "keywords": ["improves", "searching"]  
}
```

Field	Description
name	a required field for a package to install. Needs to be lowercase, single word without spaces. (Dashes and underscores allowed)
version	a required field for the package version using semantic versioning .
description	a short description of the project
author	specifies the author of the package
contributors	an array of objects, one for each contributor
keywords	an array of strings, this will help people finding your package

Section 10.4: Dependencies

```
"dependencies": { "module-name": "0.1.0" }
```

- **exact**: 0.1.0 will install that specific version of the module.
- **newest minor version**: ^0.1.0 will install the newest minor version, for example 0.2.0, but won't install a module with a higher major version e.g. 1.0.0
- **newest patch**: 0.1.x or ~0.1.0 will install the newest patch version available, for example 0.1.4, but won't install a module with higher major or minor version, e.g. 0.2.0 or 1.0.0.
- **wildcard**: * will install the latest version of the module.
- **git repository**: the following will install a tarball from the master branch of a git repo. A #sha, #tag or #branch can also be provided:
 - GitHub: user/project or user/project#v1.0.0
 - url: git://gitlab.com/user/project.git or git://gitlab.com/user/project.git#develop
- **local path**: file:./lib/project

After adding them to your package.json, use the command `npm install` in your project directory in terminal.

devDependencies

```
"devDependencies": {  
  "module-name": "0.1.0"  
}
```

For dependencies required only for development, like testing styling proxies ext. Those dev-dependencies won't be

在生产模式下运行“npm install”时安装。

第10.5节：扩展项目定义

一些额外的属性会被npm网站解析，如repository、bugs或homepage，并显示在该包的信息框中

```
{  
  "main": "server.js",  
  "repository" : {  
    "type": "git",  
    "url": "git+https://github.com/<accountname>/<repositoryname>.git"  
  },  
  "bugs": {  
    "url": "https://github.com/<accountname>/<repositoryname>/issues"  
  },  
  "homepage": "https://github.com/<accountname>/<repositoryname>/readme",  
  "files": [  
    "server.js", // 源文件  
    "README.md", // 附加文件  
    "lib" // 包含所有文件的文件夹  
  ]  
}
```

字段 描述

主要 此包的入口脚本。当用户调用该包时，将返回此脚本。

仓库 公开仓库的位置和类型

bugs 此包的错误跟踪器（例如 github）

主页 此包或整个项目的主页

files 当用户执行 npm install <packagename> 时应下载的文件和文件夹列表

installed when running "npm install" in production mode.

Section 10.5: Extended project definition

Some of the additional attributes are parsed by the npm website like repository, bugs or homepage and shown in the infobox for this packages

```
{  
  "main": "server.js",  
  "repository" : {  
    "type": "git",  
    "url": "git+https://github.com/<accountname>/<repositoryname>.git"  
  },  
  "bugs": {  
    "url": "https://github.com/<accountname>/<repositoryname>/issues"  
  },  
  "homepage": "https://github.com/<accountname>/<repositoryname>/readme",  
  "files": [  
    "server.js", // source files  
    "README.md", // additional files  
    "lib" // folder with all included files  
  ]  
}
```

Field	Description
main	Entry script for this package. This script is returned when a user requires the package.
repository	Location and type of the public repository
bugs	Bugtracker for this package (e.g. github)
homepage	Homepage for this package or the general project
files	List of files and folders which should be downloaded when a user does a npm install <packagename>

第11章：事件触发器

第11.1节：基础知识

事件触发器内置于 Node 中，用于发布-订阅模式，其中 **发布者** 会触发事件，**订阅者** 可以监听并响应这些事件。在 Node 术语中，发布者称为 **事件触发器**，负责触发事件，而订阅者称为 **监听器**，负责响应事件。

```
// 引入 events 模块以开始使用
const EventEmitter = require('events').EventEmitter;
// 狗有事件要发布，或发出
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// 当 myDog 咀嚼时，运行以下函数
myDog.on('chew', (item) => {
  if (item instanceof Food) {
    console.log('好狗');
  } else {
    console.log(`该买另一个 ${item}`);
  }
});

myDog.emit('chew', 'shoe'); // 会导致 console.log('该买另一个 shoe')
const bacon = new Food();
myDog.emit('chew', bacon); // 会导致 console.log('好狗')
```

在上述示例中，狗是发布者/EventEmitter，而检查 item 的函数是订阅者/监听器。你也可以添加更多监听器：

```
myDog.on('bark', () => {
  console.log('谁在门口？');
  // Panic
});
```

一个事件也可以有多个监听器，甚至可以移除监听器：

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// 用下一行代码撤销上一行操作：
myDog.removeListener('chew', calmDown);
```

如果你只想监听一次事件，可以使用：

```
myDog.once('chew', pet);
```

这会自动移除监听器，避免竞态条件。

第11.2节：获取已订阅事件的名称

函数EventEmitter.eventNames()会返回一个数组，包含当前已订阅的事件名称。

Chapter 11: Event Emitters

Section 11.1: Basics

Event Emitters are built into Node, and are for pub-sub, a pattern where a *publisher* will emit events, which *subscribers* can listen and react to. In Node jargon, publishers are called *Event Emitters*, and they emit events, while subscribers are called *listeners*, and they react to the events.

```
// Require events to start using them
const EventEmitter = require('events').EventEmitter;
// Dogs have events to publish, or emit
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// When myDog is chewing, run the following function
myDog.on('chew', (item) => {
  if (item instanceof Food) {
    console.log('Good dog');
  } else {
    console.log(`Time to buy another ${item}`);
  }
});

myDog.emit('chew', 'shoe'); // Will result in console.log('Time to buy another shoe')
const bacon = new Food();
myDog.emit('chew', bacon); // Will result in console.log('Good dog')
```

In the above example, the dog is the publisher/EventEmitter, while the function that checks the item was the subscriber/listener. You can make more listeners too:

```
myDog.on('bark', () => {
  console.log('WHO'S AT THE DOOR?');
  // Panic
});
```

There can also be multiple listeners for a single event, and even remove listeners:

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// Undo the previous line with the next one:
myDog.removeListener('chew', calmDown);
```

If you want to listen to a event only once, you can use:

```
myDog.once('chew', pet);
```

Which will remove the listener automatically without race conditions.

Section 11.2: Get the names of the events that are subscribed to

The function **EventEmitter.eventNames()** will return an array containing the names of the events currently subscribed to.

```

const EventEmitter = require("events");
class MyEmitter extends EventEmitter{};

var emitter = new MyEmitter();

emitter
.on("message", function(){ //监听 message 事件
  console.log("触发了一条消息！");
})
.on("message", function(){ //监听 message 事件
  console.log("这不是正确的消息");
})
.on("data", function(){ //监听 data 事件
  console.log("刚刚发生了一条数据！！");
});

console.log(emitter.eventNames()); //=> ["message","data"]
emitter.removeAllListeners("data"); //=> 移除 data 事件的所有监听器
console.log(emitter.eventNames()); //=> ["message"]

```

[在 RunKit 中运行](#)

第11.3节：通过事件发射器进行HTTP分析

在HTTP服务器代码中（例如server.js）：

```

const EventEmitter = require('events')
const serverEvents = new EventEmitter()

// 设置一个HTTP服务器
const http = require('http')
const httpServer = http.createServer((request, response) => {
  // 处理请求...
  // 然后触发一个关于发生事件的事件
  serverEvents.emit('request', request.method, request.url)
});

// 暴露事件触发器
module.exports = serverEvents

```

在 supervisor 代码中（例如 supervisor.js）：

```

const server = require("./server.js")
// 由于服务器导出了一个事件触发器，我们可以监听它的变化：
server.on('request', (method, url) => {
  console.log(`收到请求: ${method} ${url}`)
})

```

每当服务器收到请求时，它会触发一个名为 `request` 的事件，supervisor 正在监听该事件，然后 supervisor 可以对该事件做出反应。

第11.4节：获取注册监听特定事件的监听器数量

函数 `Emitter.listenerCount(eventName)` 将返回当前正在监听作为参数提供的事件的监听器数量

```
const EventEmitter = require("events");
```

```

const EventEmitter = require("events");
class MyEmitter extends EventEmitter{};

var emitter = new MyEmitter();

emitter
.on("message", function(){ //listen for message event
  console.log("a message was emitted!");
})
.on("message", function(){ //listen for message event
  console.log("this is not the right message");
})
.on("data", function(){ //listen for data event
  console.log("a data just occurred!!");
});

console.log(emitter.eventNames()); //=> ["message", "data"]
emitter.removeAllListeners("data"); //=> removeAllListeners to data event
console.log(emitter.eventNames()); //=> ["message"]

```

[Run in RunKit](#)

Section 11.3: HTTP Analytics through an Event Emitter

In the HTTP server code (e.g. `server.js`):

```

const EventEmitter = require('events')
const serverEvents = new EventEmitter()

// Set up an HTTP server
const http = require('http')
const httpServer = http.createServer((request, response) => {
  // Handler the request...
  // Then emit an event about what happened
  serverEvents.emit('request', request.method, request.url)
});

// Expose the event emitter
module.exports = serverEvents

```

In supervisor code (e.g. `supervisor.js`):

```

const server = require('./server.js')
// Since the server exported an event emitter, we can listen to it for changes:
server.on('request', (method, url) => {
  console.log(`Got a request: ${method} ${url}`)
})

```

Whenever the server gets a request, it will emit an event called `request` which the supervisor is listening for, and then the supervisor can react to the event.

Section 11.4: Get the number of listeners registered to listen for a specific event

The function `Emitter.listenerCount(eventName)` will return the number of listeners that are currently listening for the event provided as argument

```
const EventEmitter = require("events");
```

```

class MyEmitter extends EventEmitter{
  var emitter = new MyEmitter();

  emitter
    .on("data", ()=>{ // 为 data 事件添加监听器
      console.log("data event emitter");
    });

  console.log(emitter.listenerCount("data")) // => 1
  console.log(emitter.listenerCount("message")) // => 0

  emitter.on("message", function mListener(){ //为 message 事件添加监听器
    console.log("message event emitted");
  });
  console.log(emitter.listenerCount("data")) // => 1
  console.log(emitter.listenerCount("message")) // => 1

  emitter.once("data", (stuff)=>{ //为 data 事件添加另一个监听器
    console.log(`告诉我我的 ${stuff}`);
  })

  console.log(emitter.listenerCount("data")) // => 2
  console.log(emitter.listenerCount("message")) // => 1

```

belindoc.com

```

class MyEmitter extends EventEmitter{
  var emitter = new MyEmitter();

  emitter
    .on("data", ()=>{ // add listener for data event
      console.log("data event emitter");
    });

  console.log(emitter.listenerCount("data")) // => 1
  console.log(emitter.listenerCount("message")) // => 0

  emitter.on("message", function mListener(){ //add listener for message event
    console.log("message event emitted");
  });
  console.log(emitter.listenerCount("data")) // => 1
  console.log(emitter.listenerCount("message")) // => 1

  emitter.once("data", (stuff)=>{ //add another listener for data event
    console.log(`Tell me my ${stuff}`);
  })

  console.log(emitter.listenerCount("data")) // => 2
  console.log(emitter.listenerCount("message")) // => 1

```

第12章：变更时自动重载

第12.1节：使用

nodemon 进行源代码变更时的自动重载

nodemon 包使您在修改源代码中的任何文件时能够自动重新加载程序。

全局安装 nodemon

```
npm install -g nodemon (或 npm i -g nodemon)
```

本地安装 nodemon

如果你不想全局安装

```
npm install --save-dev nodemon (或 npm i -D nodemon)
```

使用 nodemon

用 nodemon entry.js (或 nodemon entry) 运行你的程序

这替代了通常使用的 node entry.js (或 node entry)。

你也可以将 nodemon 启动命令添加为 npm 脚本，如果你想传递参数而不想每次都手动输入，这会很有用。

添加到 package.json：

```
"scripts": {  
  "start": "nodemon entry.js -devmode -something 1"  
}
```

这样你就可以直接在控制台使用 npm start。

第12.2节：Browsersync

概述

Browsersync 是一个支持实时文件监控和浏览器自动刷新功能的工具。它作为一个 NPM 包 提供。

安装

要安装 Browsersync，您首先需要安装 Node.js 和 NPM。更多信息请参见 SO 上关于安装和运行 Node.js 的文档。

项目设置完成后，您可以使用以下命令安装 Browsersync：

```
$ npm install browser-sync -D
```

这将把 Browsersync 安装到本地的 node_modules 目录中，并将其保存到您的开发依赖中。

如果您想全局安装，可以用 -g 标志替代 -D 标志。

Windows 用户

Chapter 12: Autoreload on changes

Section 12.1: Autoreload on source code changes using nodemon

The nodemon package makes it possible to automatically reload your program when you modify any file in the source code.

Installing nodemon globally

```
npm install -g nodemon (or npm i -g nodemon)
```

Installing nodemon locally

In case you don't want to install it globally

```
npm install --save-dev nodemon (or npm i -D nodemon)
```

Using nodemon

Run your program with nodemon entry.js (or nodemon entry)

This replaces the usual use of node entry.js (or node entry).

You can also add your nodemon startup as an npm script, which might be useful if you want to supply parameters and not type them out every time.

Add package.json:

```
"scripts": {  
  "start": "nodemon entry.js -devmode -something 1"  
}
```

This way you can just use npm start from your console.

Section 12.2: Browsersync

Overview

Browsersync 是一个允许进行实时文件监控和浏览器自动刷新的工具。它作为一个 NPM 包 提供。

Installation

To install Browsersync you'll first need to have Node.js 和 NPM 安装。更多信息请参见 SO 上关于安装和运行 Node.js 的文档。

Once your project is set up you can install Browsersync with the following command:

```
$ npm install browser-sync -D
```

This will install Browsersync in the local node_modules 目录并将其保存到您的开发依赖中。

If you'd rather install it globally use the -g flag in place of the -D flag.

Windows Users

如果您在 Windows 上安装 Browsersync 遇到问题，您可能需要安装 Visual Studio，以便访问构建工具来安装 Browsersync。然后，您需要像这样指定您使用的 Visual Studio 版本：

```
$ npm install browser-sync --msvs_version=2013 -D
```

此命令指定使用2013版本的Visual Studio。

基本用法

要在项目中每次更改JavaScript文件时自动重新加载您的网站，请使用以下命令：

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

将myproject.dev替换为您用于访问项目的网页地址。Browsersync将输出一个可通过代理访问您网站的备用地址。

高级用法

除了上述命令行界面外，Browsersync还可以与[Grunt.js](#)和[Gulp.js](#)一起使用。

Grunt.js

与Grunt.js一起使用需要一个插件，可以通过以下方式安装：

```
$ npm install grunt-browser-sync -D
```

然后你需要将这一行添加到你的 gruntfile.js 中：

```
grunt.loadNpmTasks('grunt-browser-sync');
```

Gulp.js

Browsersync 作为一个 [CommonJS](#) 模块工作，因此不需要 Gulp.js 插件。只需像下面这样引入模块：

```
var browserSync = require('browser-sync').create();
```

你现在可以使用 [Browsersync API](#) 来根据你的需求进行配置。

API

Browsersync API 可在此处找到：<https://browsersync.io/docs/api>

If you're having trouble installing Browsersync on Windows you may need to install Visual Studio so you can access the build tools to install Browsersync. You'll then need to specify the version of Visual Studio you're using like so:

```
$ npm install browser-sync --msvs_version=2013 -D
```

This command specifies the 2013 version of Visual Studio.

Basic Usage

To automatically reload your site whenever you change a JavaScript file in your project use the following command:

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

Replace myproject.dev with the web address that you are using to access your project. Browsersync will output an alternate address that can be used to access your site through the proxy.

Advanced Usage

Besides the command line interface that was described above Browsersync can also be used with [Grunt.js](#) and [Gulp.js](#).

Grunt.js

Usage with Grunt.js requires a plugin that can be installed like so:

```
$ npm install grunt-browser-sync -D
```

Then you'll add this line to your gruntfile.js:

```
grunt.loadNpmTasks('grunt-browser-sync');
```

Gulp.js

Browsersync works as a [CommonJS](#) module, so there's no need for a Gulp.js plugin. Simply require the module like so:

```
var browserSync = require('browser-sync').create();
```

You can now use the [Browsersync API](#) to configure it to your needs.

API

The Browsersync API can be found here: <https://browsersync.io/docs/api>

第13章：环境

第13.1节：访问环境变量

`process.env` 属性返回一个包含用户环境的对象。

它返回一个类似这样的对象：

```
{  
  TERM: 'xterm-256color',  
  SHELL: '/usr/local/bin/bash',  
  USER: 'maciej',  
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',  
  PWD: '/Users/maciej',  
  EDITOR: 'vim',  
  SHLVL: '1',  
  HOME: '/Users/maciej',  
  LOGNAME: 'maciej',  
  _: '/usr/local/bin/node'  
}  
  
process.env.HOME // '/Users/maciej'
```

如果你将环境变量 `FOO` 设置为 `foobar`, 它将可以通过以下方式访问：

```
process.env.FOO // 'foobar'
```

第13.2节：process.argv 命令行参数

`process.argv` 是一个包含命令行参数的数组。第一个元素是 `node`, 第二个元素是JavaScript文件的名称。接下来的元素是任何额外的命令行参数。

代码示例：

输出所有命令行参数的和

```
index.js
```

```
var sum = 0;  
for (i = 2; i < process.argv.length; i++) {  
  sum += Number(process.argv[i]);  
}  
  
console.log(sum);
```

使用示例：

```
node index.js 2 5 6 7
```

输出将是 20

代码简要说明：

这里在 `for` 循环中 `for (i = 2; i < process.argv.length; i++)` 循环从 2 开始，因为 `process.argv` 数组中的前两个元素始终是 `['path/to/node.exe', 'path/to/js/file', ...]`

Chapter 13: Environment

Section 13.1: Accessing environment variables

The `process.env` property returns an object containing the user environment.

It returns an object like this one :

```
{  
  TERM: 'xterm-256color',  
  SHELL: '/usr/local/bin/bash',  
  USER: 'maciej',  
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',  
  PWD: '/Users/maciej',  
  EDITOR: 'vim',  
  SHLVL: '1',  
  HOME: '/Users/maciej',  
  LOGNAME: 'maciej',  
  _: '/usr/local/bin/node'  
}  
  
process.env.HOME // '/Users/maciej'
```

If you set environment variable `FOO` to `foobar`, it will be accessible with:

```
process.env.FOO // 'foobar'
```

Section 13.2: process.argv command line arguments

`process.argv` is an array containing the command line arguments. The first element will be `node`, the second element will be the name of the JavaScript file. The next elements will be any additional command line arguments.

Code Example:

Output sum of all command line arguments

```
index.js
```

```
var sum = 0;  
for (i = 2; i < process.argv.length; i++) {  
  sum += Number(process.argv[i]);  
}  
  
console.log(sum);
```

Usage Example:

```
node index.js 2 5 6 7
```

Output will be 20

A brief explanation of the code:

Here in for loop `for (i = 2; i < process.argv.length; i++)` loop begins with 2 because first two elements in `process.argv` array **always** is `['path/to/node.exe', 'path/to/js/file', ...]`

转换为数字 Number(process.argv[i]) 是因为 process.argv 数组中的元素始终是字符串

第13.3节：从属性文件加载环境属性

- 安装属性读取器：

```
npm install properties-reader --save
```

- 创建一个目录 env 用于存放你的属性文件：

```
mkdir env
```

- 创建 environments.js：

```
process.argv.forEach(function (val, index, array) {  
  var arg = val.split('=');  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./env/' + arg[1] + '.properties');  
      module.exports = env;  
    }  
  }  
});
```

- 示例development.properties属性文件：

```
# 开发属性  
[main]  
# 运行节点服务器的应用端口  
app.port=8080  
  
[database]  
# 连接到mysql的数据库  
mysql.host=localhost  
mysql.port=2500  
...
```

- 加载属性的示例用法：

```
var enviornment = require('./environments');  
var PropertiesReader = require('properties-reader');  
var properties = new PropertiesReader(enviornment);  
  
var someVal = properties.get('main.app.port');
```

- 启动快速服务器

```
npm start env=development
```

或者

```
npm start env=production
```

第13.4节：为不同环境（如开发、测试、预发布等）使用不同的属性/配置

大型应用程序在不同环境下运行时通常需要不同的属性。我们可以实现

Converting to number Number (process.argv[i]) because elements in process.argv array **always** is string

Section 13.3: Loading environment properties from a "property file"

- Install properties reader:

```
npm install properties-reader --save
```

- Create a **directory env** to store your properties files:

```
mkdir env
```

- Create **environments.js**:

```
process.argv.forEach(function (val, index, array) {  
  var arg = val.split('=');  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./env/' + arg[1] + '.properties');  
      module.exports = env;  
    }  
  }  
});
```

- Sample **development.properties** properties file:

```
# Dev properties  
[main]  
# Application port to run the node server  
app.port=8080
```

```
[database]  
# Database connection to mysql  
mysql.host=localhost  
mysql.port=2500  
...
```

- Sample usage of the loaded properties:

```
var enviornment = require('./environments');  
var PropertiesReader = require('properties-reader');  
var properties = new PropertiesReader(enviornment);  
  
var someVal = properties.get('main.app.port');
```

- Starting the express server

```
npm start env=development
```

or

```
npm start env=production
```

Section 13.4: Using different Properties/Configuration for different environments like dev, qa, staging etc

Large scale applications often need different properties when running on different environments. we can achieve

通过向 Node.js 应用程序传递参数，并在节点进程中使用相同的参数加载特定的环境属性文件。

假设我们有两个用于不同环境的属性文件。

- dev.json

```
{  
  端口 : 3000,  
  数据库 : {  
    主机 : "localhost",  
    用户 : "bob",  
    password : "12345"  
  }  
}
```

- qa.json

```
{  
  PORT : 3001,  
  DB : {  
    host : "where_db_is_hosted",  
    user : "bob",  
    password : "54321"  
  }  
}
```

应用中的以下代码将导出我们想要使用的相应属性文件。

假设代码在 environment.js 中

```
process.argv.forEach(function (val, index, array) {  
  var arg = val.split("=".  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./' + arg[1] + '.json');  
      module.exports = env;  
    }  
  }  
});
```

我们给应用传递的参数如下

```
node app.js env=dev
```

如果我们使用像forever这样的进程管理器，那么就很简单

```
forever start app.js 环境=开发
```

如何使用配置文件

```
var 环境= require("environment.js");
```

this by passing arguments to Node.js application and using same argument in node process to load specific environment property file.

Suppose we have two property files for different environment.

- dev.json

```
{  
  PORT : 3000,  
  DB : {  
    host : "localhost",  
    user : "bob",  
    password : "12345"  
  }  
}
```

- qa.json

```
{  
  PORT : 3001,  
  DB : {  
    host : "where_db_is_hosted",  
    user : "bob",  
    password : "54321"  
  }  
}
```

Following code in application will export respective property file which we want to use.

Suppose the code is in environment.js

```
process.argv.forEach(function (val, index, array) {  
  var arg = val.split("=".  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./' + arg[1] + '.json');  
      module.exports = env;  
    }  
  }  
});
```

We give arguments to the application like following

```
node app.js env=dev
```

if we are using process manager like forever than it as simple as

```
forever start app.js env=dev
```

How to use the configuration file

```
var env= require("environment.js");
```

第14章：回调到Promise

第14.1节：将回调转换为Promise

基于回调：

```
db.通知邮件.查找({主题: 'promisify callback'}, (错误, 结果) => {
  如果 (错误) {
    控制台.日志(错误);
  }
  // 这里是正常代码
});
```

这使用了 bluebird 的 promisifyAll 方法，将传统基于回调的代码如上进行 promisify。bluebird 会为对象中的所有方法创建一个基于 Promise 的版本，这些基于 Promise 的方法名会在后面加上 Async：

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {
  // 这里是正常代码
})
.catch(console.error);
```

如果只需要对特定方法进行 promisify，则只需使用其 promisify：

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {
  // 这里是正常代码
})
.catch(console.error);
```

有些库（例如 MassiveJS）如果方法的直接对象没有作为第二个参数传入，则无法进行 promisify。在这种情况下，只需将需要 promisify 的方法的直接对象作为第二个参数传入，并将其封装在 context 属性中。

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {
  // 这里是正常代码
})
.catch(console.error);
```

第14.2节：手动将回调函数 promisify

有时可能需要手动将回调函数 promisify。这可能是因为回调函数不遵循标准的“错误优先”格式，或者需要额外的逻辑来进行 promisify：

示例使用 `fs.exists(path, callback)`：

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

Chapter 14: Callback to Promise

Section 14.1: Promisifying a callback

Callback-based:

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
  if (error) {
    console.log(error);
  }
  // normal code here
});
```

This uses bluebird's promisifyAll method to promisify what is conventionally callback-based code like above. bluebird will make a promise version of all the methods in the object, those promise-based methods names has Async appended to them:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {
  // normal code here
})
.catch(console.error);
```

If only specific methods need to be promisified, just use its promisify:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {
  // normal code here
})
.catch(console.error);
```

There are some libraries (e.g., MassiveJS) that can't be promisified if the immediate object of the method is not passed on second parameter. In that case, just pass the immediate object of the method that need to be promisified on second parameter and enclosed it in context property.

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {
  // normal code here
})
.catch(console.error);
```

Section 14.2: Manually promisifying a callback

Sometimes it might be necessary to manually promisify a callback function. This could be for a case where the callback does not follow the standard [error-first format](#) or if additional logic is needed to promisify:

Example with `fs.exists(path, callback)`:

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

```

var existsAsync = function(path) {
  return new Promise(function(resolve, reject) {
    fs.exists(path, function(exists) {
      // exists 是一个布尔值
      if (exists) {
        // 成功时调用 resolve
        resolve();
      } else {
        // 出错时调用 reject
        reject(new Error('path does not exist'));
      }
    });
  });

// 现在作为 Promise 使用
existsAsync('/path/to/some/file').then(function() {
  console.log('file exists!');
}).catch(function(err) {
  // 文件不存在
  console.error(err);
});

```

第14.3节：setTimeout的Promise化

```

函数 wait(毫秒) {
  返回新的 Promise(函数 (resolve, reject) {
    setTimeout(resolve, 毫秒)
  })
}

```

```

var existsAsync = function(path) {
  return new Promise(function(resolve, reject) {
    fs.exists(path, function(exists) {
      // exists is a boolean
      if (exists) {
        // Resolve successfully
        resolve();
      } else {
        // Reject with error
        reject(new Error('path does not exist'));
      }
    });
  });

// Use as a promise now
existsAsync('/path/to/some/file').then(function() {
  console.log('file exists!');
}).catch(function(err) {
  // file does not exist
  console.error(err);
});

```

Section 14.3: setTimeout promisified

```

function wait(ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, ms)
  })
}

```

第15章：使用子进程执行文件或命令

第15.1节：生成新进程以执行命令

要生成一个新进程，并且需要无缓冲输出（例如，可能会在一段时间内打印输出的长时间运行进程，而不是立即打印后退出），请使用`child_process.spawn()`。

此方法使用给定的命令和参数数组生成一个新进程。返回值是一个`ChildProcess`实例，该实例提供了`stdout`和`stderr`属性。这两个流都是`stream.Readable`的实例。

下面的代码等同于运行命令`ls -lh /usr`。

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`子进程以代码 ${code} 退出`);
});
```

另一个示例命令：

```
zip -Ovr "archive" ./image.png
```

可能写成：

```
spawn('zip', ['-Ovr', "archive", './image.png']);
```

第15.2节：生成一个shell来执行命令

要在shell中运行命令，并且需要缓冲输出（即非流式输出），请使用`child_process.exec`。例如，如果你想运行命令`cat *.js file | wc -l`，且无任何选项，则写法如下：

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`执行错误: ${err}`);
    return;
  }

  console.log(`标准输出: ${stdout}`);
  console.log(`标准错误: ${stderr}`);
});
```

该函数最多接受三个参数：

Chapter 15: Executing files or commands with Child Processes

Section 15.1: Spawning a new process to execute a command

To spawn a new process in which you need *unbuffered* output (e.g. long-running processes which might print output over a period of time rather than printing and exiting immediately), use `child_process.spawn()`.

This method spawns a new process using a given command and an array of arguments. The return value is an instance of `ChildProcess`, which in turn provides the `stdout` and `stderr` properties. Both of those streams are instances of `stream.Readable`.

The following code is equivalent to using running the command `ls -lh /usr`.

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

Another example command:

```
zip -Ovr "archive" ./image.png
```

Might be written as:

```
spawn('zip', ['-Ovr', "archive", './image.png']);
```

Section 15.2: Spawning a shell to execute a command

To run a command in a shell, in which you required buffered output (i.e. it is not a stream), use `child_process.exec`. For example, if you wanted to run the command `cat *.js file | wc -l`, with no options, that would look like this:

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }

  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

The function accepts up to three parameters:

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

command 参数是字符串，且为必需，options 对象和 callback 都是可选的。如果未指定 options 对象，则 exec 将使用以下默认值：

```
{  
  encoding: 'utf8',  
  timeout: 0,  
  maxBuffer: 200*1024,  
  killSignal: 'SIGTERM',  
  cwd: null,  
  env: null  
}
```

options 对象还支持一个 shell 参数，默认情况下 UNIX 上为 /bin/sh，Windows 上为 cmd.exe，还有用于设置进程用户身份的 uid 选项，以及用于设置组身份的 gid 选项。

当命令执行完成时调用的回调函数，会带有三个参数 (err, stdout, stderr)。如果命令执行成功，err 将为 null，否则它将是一个 Error 的实例，其中 err.code 是进程的退出代码，err.signal 是用于终止进程的信号。

命令的 stdout 和 stderr 参数是命令的输出。它会使用 options 对象中指定的编码进行解码（默认：string），否则可以作为 Buffer 对象返回。

还有一个同步版本的 exec，叫做 execSync。同步版本不接受回调函数，并且会返回 stdout 而不是 ChildProcess 实例。如果同步版本遇到错误，它会抛出异常并停止程序。用法如下：

```
const execSync = require('child_process').execSync;  
const stdout = execSync('cat *.js file | wc -l');  
console.log(`stdout: ${stdout}`);
```

第15.3节：生成进程以运行可执行文件

如果你想运行一个文件，比如可执行文件，使用 child_process.execFile。它不会像 child_process.exec 那样生成一个 shell，而是直接创建一个新进程，这比运行命令稍微高效一些。函数用法如下：

```
const execFile = require('child_process').execFile;  
const child = execFile('node', ['--version'], (err, stdout, stderr) => {  
  if (err) {  
    throw err;  
  }  
  
  console.log(stdout);  
});
```

与 child_process.exec 不同，此函数最多接受四个参数，其中第二个参数是你想传递给可执行文件的参数数组：

```
child_process.execFile(file[, args][, options][, callback]);
```

否则，options 和 callback 的格式与 child_process.exec 完全相同。同步版本的函数也是如此：

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

The command parameter is a string, and is required, while the options object and callback are both optional. If no options object is specified, then exec will use the following as a default:

```
{  
  encoding: 'utf8',  
  timeout: 0,  
  maxBuffer: 200*1024,  
  killSignal: 'SIGTERM',  
  cwd: null,  
  env: null  
}
```

The options object also supports a shell parameter, which is by default /bin/sh on UNIX and cmd.exe on Windows, a uid option for setting the user identity of the process, and a gid option for the group identity.

The callback, which is called when the command is done executing, is called with the three arguments (err, stdout, stderr). If the command executes successfully, err will be null, otherwise it will be an instance of Error, with err.code being the exit code of the process and err.signal being the signal that was sent to terminate it.

The stdout and stderr arguments are the output of the command. It is decoded with the encoding specified in the options object (default: string), but can otherwise be returned as a Buffer object.

There also exists a synchronous version of exec, which is execSync. The synchronous version does not take a callback, and will return stdout instead of an instance of ChildProcess. If the synchronous version encounters an error, it will throw and halt your program. It looks like this:

```
const execSync = require('child_process').execSync;  
const stdout = execSync('cat *.js file | wc -l');  
console.log(`stdout: ${stdout}`);
```

Section 15.3: Spawning a process to run an executable

If you are looking to run a file, such as an executable, use child_process.execFile. Instead of spawning a shell like child_process.exec would, it will directly create a new process, which is slightly more efficient than running a command. The function can be used like so:

```
const execFile = require('child_process').execFile;  
const child = execFile('node', ['--version'], (err, stdout, stderr) => {  
  if (err) {  
    throw err;  
  }  
  
  console.log(stdout);  
});
```

Unlike child_process.exec, this function will accept up to four parameters, where the second parameter is an array of arguments you'd like to supply to the executable:

```
child_process.execFile(file[, args][, options][, callback]);
```

Otherwise, the options and callback format are otherwise identical to child_process.exec. The same goes for the synchronous version of the function:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

belindoc.com

第16章：异常处理

第16.1节：Node.js中的异常处理

Node.js有三种基本的异常/错误处理方式：

1. **try-catch**块
2. **error** 作为回调函数的第一个参数
3. 使用 **eventEmitter** 发出一个错误事件

try-catch 用于捕获同步代码执行中抛出的异常。如果调用者（或调用者的调用者，依此类推）使用了 **try/catch**，那么他们可以捕获该错误。如果所有调用者都没有使用 **try-catch**，那么程序将崩溃。

如果在异步操作上使用 **try-catch**，而异常是从异步方法的回调中抛出的，那么该异常不会被 **try-catch** 捕获。要捕获异步操作回调中的异常，建议使用 **promises**。

示例以便更好理解

```
// ** 示例 - 1 **
function doSomeSynchronousOperation(req, res) {
  if(req.body.username === ''){
    throw new Error('用户名不能为空');
  }
  return true;
}

// 调用上述方法
try {
  // 同步代码
  doSomeSynchronousOperation(req, res)
  catch(e) {
    // 异常在此处处理
    console.log(e.message);
  }
}

// ** 示例 - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
  // 模拟异步操作
  return setTimeout(function(){
    cb(null, []);
  }, 1000);
}

try {
  // 异步代码
  doSomeAsynchronousOperation(req, res, function(err, rs){
    throw new Error("async operation exception");
  })
} catch(e) {
  // 异常不会在这里处理
  console.log(e.message);
}
// 该异常未被处理，因此会导致应用程序崩溃
```

回调主要用于 Node.js，因为回调异步传递事件。用户传递给你一个函数（回调函数），你在异步操作完成后某个时间点调用它。

通常的模式是回调以 **callback(err, result)** 的形式被调用，其中 **err** 和 **result** 只有一个非空，取决于操作是成功还是失败。

Chapter 16: Exception handling

Section 16.1: Handling Exception In Node.Js

Node.js has 3 basic ways to handle exceptions/errors:

1. **try-catch** block
2. **error** as the first argument to a callback
3. emit an **error** event using **eventEmitter**

try-catch is used to catch the exceptions thrown from the synchronous code execution. If the caller (or the caller's caller, ...) used try/catch, then they can catch the error. If none of the callers had try-catch than the program crashes.

If using try-catch on an async operation and exception was thrown from callback of async method than it will not get caught by try-catch. To catch an exception from async operation callback, it is preferred to use **promises**.

Example to understand it better

```
// ** Example - 1 **
function doSomeSynchronousOperation(req, res) {
  if(req.body.username === ''){
    throw new Error('User Name cannot be empty');
  }
  return true;
}

// calling the method above
try {
  // synchronous code
  doSomeSynchronousOperation(req, res)
  catch(e) {
    //exception handled here
    console.log(e.message);
  }
}

// ** Example - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
  // imitating async operation
  return setTimeout(function(){
    cb(null, []);
  }, 1000);
}

try {
  // asynchronous code
  doSomeAsynchronousOperation(req, res, function(err, rs){
    throw new Error("async operation exception");
  })
} catch(e) {
  // Exception will not get handled here
  console.log(e.message);
}
// The exception is unhandled and hence will cause application to break
```

callbacks are mostly used in Node.js as callback delivers an event asynchronously. The user passes you a function (the callback), and you invoke it sometime later when the asynchronous operation completes.

The usual pattern is that the callback is invoked as a **callback(err, result)**, where only one of err and result is non-null, depending on whether the operation succeeded or failed.

```

function 执行某异步操作(请求, 响应, 回调) {
  setTimeout(function(){
    return 回调(new Error('用户名不能为空'));
  }, 1000);
  return true;
}

执行某异步操作(请求, 响应, function(错误, 结果) {
  if (错误) {
    // 异常在此处处理
    console.log(错误.message);
  }

  // 使用有效数据进行一些操作
});

```

emit 对于更复杂的情况，不使用回调函数，函数本身可以返回一个 `EventEmitter` 对象，调用者需要监听该发射器上的错误事件。

```

const EventEmitter = require('events');

function doSomeAsynchronousOperation(req, res) {
  let myEvent = new EventEmitter();

  // 异步运行
  setTimeout(function(){
    myEvent.emit('error', new Error('用户名不能为空'));
  }, 1000);

  return myEvent;
}

// 调用该函数
let event = doSomeAsynchronousOperation(req, res);

event.on('error', function(err) {
  console.log(err);
});

event.on('done', function(result) {
  console.log(result); // true
});

```

第16.2节：未处理异常管理

由于 Node.js 在单个进程上运行，未捕获的异常是开发应用程序时需要注意的问题。

静默处理异常

大多数人让 Node.js 服务器静默地吞掉错误。

- 静默处理异常

```

process.on('uncaughtException', function (err) {
  console.log(err);
});

```

这是不好的，虽然能工作，但：

```

function doSomeAsynchronousOperation(req, res, callback) {
  setTimeout(function(){
    return callback(new Error('User Name cannot be empty'));
  }, 1000);
  return true;
}

doSomeAsynchronousOperation(req, res, function(err, result) {
  if (err) {
    //exception handled here
    console.log(err.message);
  }

  //do some stuff with valid data
});

```

emit For more complicated cases, instead of using a callback, the function itself can return an `EventEmitter` object, and the caller would be expected to listen for error events on the emitter.

```

const EventEmitter = require('events');

function doSomeAsynchronousOperation(req, res) {
  let myEvent = new EventEmitter();

  // runs asynchronously
  setTimeout(function(){
    myEvent.emit('error', new Error('User Name cannot be empty'));
  }, 1000);

  return myEvent;
}

// Invoke the function
let event = doSomeAsynchronousOperation(req, res);

event.on('error', function(err) {
  console.log(err);
});

event.on('done', function(result) {
  console.log(result); // true
});

```

Section 16.2: Unhandled Exception Management

Because Node.js runs on a single process uncaught exceptions are an issue to be aware of when developing applications.

Silently Handling Exceptions

Most of the people let node.js server(s) silently swallow up the errors.

- Silently handling the exception

```

process.on('uncaughtException', function (err) {
  console.log(err);
});

```

This is bad, it will work but:

- 根本原因将保持未知，因此无法帮助解决导致异常（错误）的原因。
- 如果数据库连接（连接池）因某种原因关闭，这将导致错误不断传播，意味着服务器虽然运行，但不会重新连接数据库。

返回初始状态

在发生“uncaughtException”时，最好重启服务器并将其恢复到我们知道能正常工作的初始状态。异常被记录，应用程序终止，但由于它运行在容器中，容器会确保服务器继续运行，从而实现服务器的重启（返回到初始工作状态）。

- 安装 forever（或其他 CLI 工具以确保 node 服务器持续运行）

```
npm install forever -g
```

- 使用 forever 启动服务器

```
forever start app.js
```

启动的原因以及我们使用 forever 的原因是：当服务器被**终止**时，forever 进程会重新启动服务器。

- 重启服务器

```
process.on('uncaughtException', function (err) {
  console.log(err);

  // 一些日志机制
  // ...

  process.exit(1); // 终止进程
});
```

顺便提一下，也有使用Clusters 和 Domains来处理异常的方法。

域已被弃用，更多信息请见 [here](#)。

第16.3节：错误与承诺

承诺（Promise）处理错误的方式不同于同步或回调驱动的代码。

```
const p = new Promise(function (resolve, reject) {
  reject(new Error('哎呀'));
});

// 在承诺内部被 `reject` 的任何内容都可以通过 catch 捕获
// 当承诺被拒绝时，`then` 不会被调用
p
.then(() => {
  console.log("不会被调用");
})
.catch(e => {
  console.log(e.message); // 输出：哎呀
})
```

- Root cause will remain unknown, as such will not contribute to resolution of what caused the Exception (Error).
- In case of database connection (pool) gets closed for some reason this will result in constant propagation of errors, meaning that server will be running but it will not reconnect to db.

Returning to Initial state

In case of an " uncaughtException " it is good to restart the server and return it to its **initial state**, where we know it will work. Exception is logged, application is terminated but since it will be running in a container that will make sure that the server is running we will achieve restarting of the server (returning to the initial working state).

- Installing the forever (or other CLI tool to make sure that node server runs continuously)

```
npm install forever -g
```

- Starting the server in forever

```
forever start app.js
```

Reason why is it started and why we use forever is after the server is **terminated** forever process will start the server again.

- Restarting the server

```
process.on('uncaughtException', function (err) {
  console.log(err);

  // some logging mechanism
  // ...

  process.exit(1); // terminates process
});
```

On a side note there was a way also to handle exceptions with **Clusters and Domains**.

Domains are deprecated more information [here](#).

Section 16.3: Errors and Promises

Promises handle errors differently to synchronous or callback-driven code.

```
const p = new Promise(function (resolve, reject) {
  reject(new Error('Oops'));
});

// anything that is `reject`ed inside a promise will be available through catch
// while a promise is rejected, `then` will not be called
p
.then(() => {
  console.log("won't be called");
})
.catch(e => {
  console.log(e.message); // output: Oops
})
```

```
// 一旦错误被捕获，执行流程将继续
.then(() => {
  console.log('hello!'); // 输出：hello!
});
```

当前，在 Promise 中抛出的未被捕获的错误会被吞掉，这会导致难以追踪错误。这个问题可以通过使用像 eslint 这样的代码检查工具来解决，或者确保你总是有一个 catch 子句。

这种行为在 Node 8 中已被弃用，改为终止 Node 进程。

```
// once the error is caught, execution flow resumes
.then(() => {
  console.log('hello!'); // output: hello!
});
```

currently, errors thrown in a promise that are not caught results in the error being swallowed, which can make it difficult to track down the error. This can be solved using linting tools like [eslint](#) or by ensuring you always have a **catch** clause.

This behaviour is deprecated [in node 8](#) in favour of terminating the node process.

第17章：保持 Node 应用程序持续运行

第17.1节：使用 PM2 作为进程管理器

PM2 让你可以让 Node.js 脚本永久运行。如果你的应用程序崩溃，PM2 也会帮你重启它。

全局安装 PM2 来管理你的 Node.js 实例

```
npm install pm2 -g
```

导航到您的 nodejs 脚本所在的目录，每次想要启动由 pm2 监控的 nodejs 实例时，运行以下命令：

```
pm2 start server.js --name "app1"
```

监控进程的常用命令

1. 列出所有由 pm2 管理的 nodejs 实例

```
pm2 list
```

App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
<code>bashscript.sh</code>	6	fork	8278	online	0	10s	1.379 MB	/home/tknew/.pm2/logs/bashscript.sh-err.log
<code>checker</code>	5	cluster	0	stopped	0	2m	0 B	/home/tknew/.pm2/logs/checker-err.log
<code>interface-api</code>	3	cluster	7526	online	0	3m	15.445 MB	/home/tknew/.pm2/logs/interface-api-err.log
<code>interface-api</code>	2	cluster	7517	online	0	3m	15.453 MB	/home/tknew/.pm2/logs/interface-api-err.log
<code>interface-api</code>	1	cluster	7512	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log
<code>interface-api</code>	0	cluster	7507	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log

2. 停止某个特定的 nodejs 实例

```
pm2 stop <实例名称>
```

3. 删除某个特定的 nodejs 实例

```
pm2 delete <实例名称>
```

4. 重启某个特定的 nodejs 实例

```
pm2 restart <实例名称>
```

5. 监控所有 Node.js 实例

Chapter 17: Keep a node application constantly running

Section 17.1: Use PM2 as a process manager

PM2 lets you run your nodejs scripts forever. In the event that your application crashes, PM2 will also restart it for you.

Install PM2 globally to manage your nodejs instances

```
npm install pm2 -g
```

Navigate to the directory in which your nodejs script resides and run the following command each time you want to start a nodejs instance to be monitored by pm2:

```
pm2 start server.js --name "app1"
```

Useful commands for monitoring the process

1. List all nodejs instances managed by pm2

```
pm2 list
```

```
[tknew:~/Unitech	pm2] master(+84/-121)+* ± pm2 list
○ PM2 Process listing
```

App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
<code>bashscript.sh</code>	6	fork	8278	online	0	10s	1.379 MB	/home/tknew/.pm2/logs/bashscript.sh-err.log
<code>checker</code>	5	cluster	0	stopped	0	2m	0 B	/home/tknew/.pm2/logs/checker-err.log
<code>interface-api</code>	3	cluster	7526	online	0	3m	15.445 MB	/home/tknew/.pm2/logs/interface-api-err.log
<code>interface-api</code>	2	cluster	7517	online	0	3m	15.453 MB	/home/tknew/.pm2/logs/interface-api-err.log
<code>interface-api</code>	1	cluster	7512	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log
<code>interface-api</code>	0	cluster	7507	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log

2. Stop a particular nodejs instance

```
pm2 stop <instance named>
```

3. Delete a particular nodejs instance

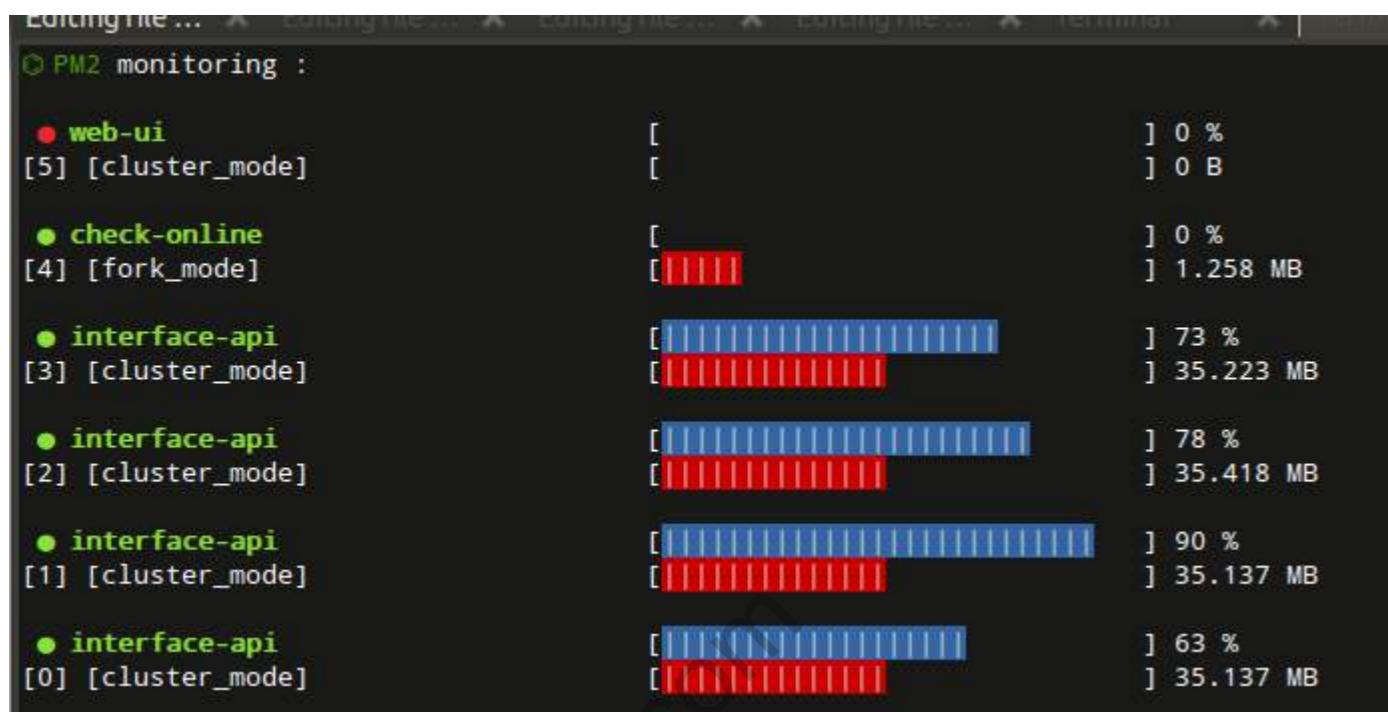
```
pm2 delete <instance name>
```

4. Restart a particular nodejs instance

```
pm2 restart <instance name>
```

5. Monitoring all nodejs instances

pm2 monit



6. 停止 pm2

pm2 kill

7. 与重启（杀死并重启进程）相反，reload 实现了零秒停机的重载

pm2 reload <实例名称>

8. 查看日志

pm2 logs <实例名称>

第17.2节：运行和停止 Forever 守护进程

启动进程：

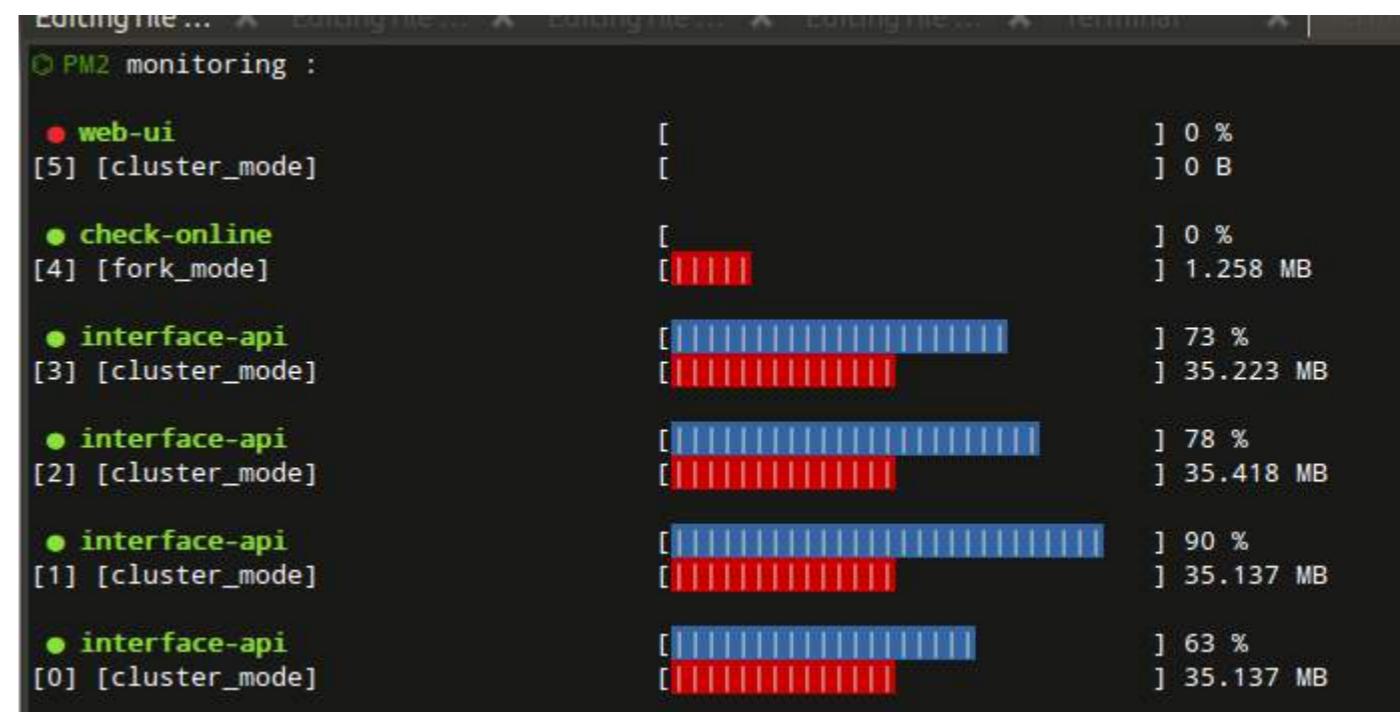
```
$ forever start index.js
warn: --minUptime 未设置. 默认值为: 1000毫秒
warn: --spinSleepTime 未设置. 如果脚本未能保持运行至少1000毫秒, 将会退出
info: Forever 正在处理文件: index.js
```

列出正在运行的 Forever 实例：

```
$ forever list
info: 正在运行的 Forever 进程

| 数据: | 索引 | uid | 命令      | 脚本      | forever 进程ID | 进程ID | 日志文件
| 运行时间 |          |          |           |           |           |           |           |
| ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
```

pm2 monit



6. Stop pm2

pm2 kill

7. As opposed to restart, which kills and restarts the process, reload achieves a 0-second-downtime reload

pm2 reload <instance name>

8. View logs

pm2 logs <instance name>

Section 17.2: Running and stopping a Forever daemon

To start the process:

```
$ forever start index.js
warn: --minUptime not set. Defaulting to: 1000ms
warn: --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
info: Forever processing file: index.js
```

List running Forever instances:

```
$ forever list
info: Forever processes running

| data: | index | uid | command      | script      | forever pid | id | logfile
| uptime |          |          |           |           |           |           |           |
| ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
```

```
|-----|  
|数据: | [0]  |f4Kt | /usr/bin/nodejs  | src/index.js|2131    | 2146|/root/.forever/f4Kt.log |  
0:0:0:11.485 |
```

停止第一个进程：

```
$ forever stop 0  
$ forever stop 2146  
$ forever stop --uid f4Kt  
$ forever stop --pidFile 2131
```

第17.3节：使用nohup持续运行

Linux上forever的替代方案是nohup。

启动nohup实例

1. 切换到app.js或www文件夹所在位置
2. 运行nohup nodejs app.js &

终止进程

1. 运行ps -ef|grep nodejs
2. kill -9 <进程号>

```
|-----|  
|data: | [0]  |f4Kt | /usr/bin/nodejs  | src/index.js|2131    | 2146|/root/.forever/f4Kt.log |  
0:0:0:11.485 |
```

Stop the first process:

```
$ forever stop 0  
$ forever stop 2146  
$ forever stop --uid f4Kt  
$ forever stop --pidFile 2131
```

Section 17.3: Continuous running with nohup

An alternative to forever on Linux is nohup.

To start a nohup instance

1. cd to the location of app.js or wwwfolder
2. run nohup nodejs app.js &

To kill the process

1. run ps -ef|grep nodejs
2. kill -9 <the process number>

第18章：卸载Node.js

第18.1节：在Mac OSX上完全卸载Node.js

在Mac操作系统的终端中，输入以下两个命令：

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do sudo rm /usr/local/${f}; done
```

```
sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

第18.2节：在Windows上卸载Node.js

要在Windows上卸载Node.js，请按如下方式使用“添加或删除程序”：

1. 从开始菜单打开“添加或删除程序”。
2. 搜索Node.js。

Windows 10 :

3. 点击Node.js。
4. 点击卸载。
5. 点击新的卸载按钮。

Windows 7-8.1 :

3. 点击 Node.js 下的卸载按钮。

Chapter 18: Uninstalling Node.js

Section 18.1: Completely uninstall Node.js on Mac OSX

In Terminal on your Mac operating system, enter the following 2 commands:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do sudo rm /usr/local/${f}; done
```

```
sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

Section 18.2: Uninstall Node.js on Windows

To uninstall Node.js on Windows, use Add or Remove Programs like this:

1. Open Add or Remove Programs from the start menu.
2. Search for Node.js.

Windows 10:

3. Click Node.js.
4. Click Uninstall.
5. Click the new Uninstall button.

Windows 7-8.1:

3. Click the Uninstall button under Node.js.

第19章：nvm - Node版本管理器

第19.1节：安装NVM

你可以使用curl：

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

或者你可以使用wget：

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

第19.2节：检查NVM版本

要验证 nvm 是否已安装，请执行：

```
command -v nvm
```

如果安装成功，应该输出“nvm”。

第19.3节：安装特定的Node版本

列出可供安装的远程版本

```
nvm ls-remote
```

安装远程版本

```
nvm install <version>
```

例如

```
nvm install 0.10.13
```

第19.4节：使用已安装的Node版本

列出通过 NVM 可用的本地 Node 版本：

```
nvm ls
```

例如，如果 nvm ls 返回：

```
$ nvm ls
  v4.3.0
  v5.5.0
```

你可以通过以下命令切换到 v5.5.0：

```
nvm use v5.5.0
```

Chapter 19: nvm - Node Version Manager

Section 19.1: Install NVM

You can use curl:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Or you can use wget:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Section 19.2: Check NVM version

To verify that nvm has been installed, do:

```
command -v nvm
```

which should output 'nvm' if the installation was successful.

Section 19.3: Installing an specific Node version

Listing available remote versions for installation

```
nvm ls-remote
```

Installing a remote version

```
nvm install <version>
```

For example

```
nvm install 0.10.13
```

Section 19.4: Using an already installed node version

To list available local versions of node through NVM:

```
nvm ls
```

For example, if nvm ls returns:

```
$ nvm ls
  v4.3.0
  v5.5.0
```

You can switch to v5.5.0 with:

```
nvm use v5.5.0
```

第19.5节：在 Mac OSX 上安装 nvm

安装过程

你可以使用 git、curl 或 wget 安装 Node 版本管理器。你需要在Mac OSX的终端中运行这些命令。

curl 示例：

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

wget 示例：

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

测试 NVM 是否正确安装

要测试 nvm 是否正确安装，请关闭并重新打开终端，然后输入 nvm。如果出现 **nvm: command not found** 消息，说明您的操作系统可能没有必要的 **.bash_profile** 文件。在终端中输入 touch **~/.bash_profile**，然后再次运行上述安装脚本。

如果仍然出现 nvm: command not found，尝试以下操作：

- 在终端中输入 nano **.bashrc**。您应该会看到一个几乎与以下内容相同的导出脚本：

```
export NVM_DIR="/Users/johndoe/.nvm" [ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
```

- 复制导出脚本并从 **.bashrc** 中删除它保存并关闭 **.bas**
- hrc** 文件 (CTRL+O – 回车 – CTRL+X) 接着，输入 nano
- .bash_profile** 打开 Bash 配置文件将复制的导出脚本粘贴
- 到 Bash 配置文件的新行中保存并关闭 Bash 配置文件 (CTRL+O – 回车 – CTRL+X)
- 最后输入 nano **.bashrc** 重新打开 **.bashrc** 文件
- 将以下行粘贴到文件中：

```
source ~/.nvm/nvm.sh
```

- 保存并关闭 (CTRL+O – 回车 – CTRL+X)
- 重启终端并输入 nvm 测试是否生效

第19.6节：在子shell中使用所需版本的 Node 运行任意命令

列出所有已安装的 Node 版本

```
nvm ls  
v4.5.0  
v6.7.0
```

使用任意已安装的 Node 版本运行命令

```
nvm run 4.5.0 --version 或 nvm exec 4.5.0 node --version
```

Section 19.5: Install nvm on Mac OSX

INSTALLATION PROCESS

You can install Node Version Manager using git, curl or wget. You run these commands in **Terminal** on **Mac OSX**.

curl example:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

wget example:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

TEST THAT NVM WAS PROPERLY INSTALLED

To test that nvm was properly installed, close and re-open Terminal and enter nvm. If you get a **nvm: command not found** message, your OS may not have the necessary **.bash_profile** file. In Terminal, enter touch **~/.bash_profile** and run the above install script again.

If you still get **nvm: command not found**, try the following:

- In Terminal, enter nano **.bashrc**. You should see an export script almost identical to the following:

```
export NVM_DIR="/Users/johndoe/.nvm" [ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
```

- Copy the export script and remove it from **.bashrc**
- Save and Close the **.bashrc** file (CTRL+O – Enter – CTRL+X)
- Next, enter nano **.bash_profile** to open the Bash Profile
- Paste the export script you copied into the Bash Profile on a new line
- Save and Close the Bash Profile (CTRL+O – Enter – CTRL+X)
- Finally enter nano **.bashrc** to re-open the **.bashrc** file
- Paste the following line into the file:

```
source ~/.nvm/nvm.sh
```

- Save and Close (CTRL+O – Enter – CTRL+X)
- Restart Terminal and enter nvm to test if it's working

Section 19.6: Run any arbitrary command in a subshell with the desired version of node

List all the node versions installed

```
nvm ls  
v4.5.0  
v6.7.0
```

Run command using any node installed version

```
nvm run 4.5.0 --version 或 nvm exec 4.5.0 node --version
```

正在运行 node v4.5.0 (npm v2.15.9)
v4.5.0

nvm run 6.7.0 --version 或 nvm exec 6.7.0 node --version
正在运行 node v6.7.0 (npm v3.10.3)
v6.7.0

使用别名

nvm run default --version 或 nvm exec default node --version
正在运行 node v6.7.0 (npm v3.10.3)
v6.7.0

安装 Node LTS 版本

nvm install --lts

版本切换

nvm 使用 v4.5.0 或 nvm 使用 stable (别名)

第19.7节：为 node 版本设置别名

如果你想为已安装的 node 版本设置别名，执行：

nvm alias <名称> <版本>

类似于取消别名，执行：

nvm unalias <名称>

一个合适的用例是，如果你想将除 stable 版本以外的其他版本设置为默认别名。**默认**别名版本会在控制台默认加载。

例如：

nvm alias default 5.0.1

那么每次控制台/终端启动时，默认会使用 5.0.1 版本。

注意：

nvm alias # 列出所有在 nvm 上创建的别名

Running node v4.5.0 (npm v2.15.9)
v4.5.0

nvm run 6.7.0 --version 或 nvm exec 6.7.0 node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0

using alias

nvm run default --version 或 nvm exec default node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0

To install node LTS version

nvm install --lts

Version Switching

nvm use v4.5.0 或 nvm use stable (alias)

Section 19.7: Setting alias for node version

If you want to set some alias name to installed node version, do:

nvm alias <name> <version>

Similary to unalias, do:

nvm unalias <name>

A proper usecase would be, if you want to set some other version than stable version as default alias. **default** aliased versions are loaded on console by default.

Like:

nvm alias **default** 5.0.1

Then every time **console/terminal** starts 5.0.1 would be present by default.

Note:

nvm alias # lists all aliases created on nvm

第20章：http

第20.1节：http服务器

HTTP服务器的一个基本示例。

在 `http_server.js` 文件中编写以下代码：

```
var http = require('http');
var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('请求已接收: ' + connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('客户端IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log("启动 HTTP 端口 " + httpPort)
}
```

然后在你的 `http_server.js` 所在位置运行此命令：

```
node http_server.js
```

你应该会看到以下结果：

```
> 启动 HTTP 端口 80
```

现在你需要测试你的服务器，打开你的浏览器并访问以下网址：

```
http://127.0.0.1:80
```

如果你的机器运行的是Linux服务器，你可以这样测试：

```
curl 127.0.0.1:80
```

你应该看到以下结果：

```
ok
```

在运行该应用的控制台中，你会看到以下结果：

```
> 请求已接收: HTTP GET /
> 客户端IP: ::ffff:127.0.0.1
```

Chapter 20: http

Section 20.1: http server

A basic example of HTTP server.

write following code in `http_server.js` file:

```
var http = require('http');
var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: ' + connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('Client IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

then from your `http_server.js` location run this command:

```
node http_server.js
```

you should see this result:

```
> Start HTTP on port 80
```

now you need to test your server, you need to open your internet browser and navigate to this url:

```
http://127.0.0.1:80
```

if your machine running Linux server you can test it like this:

```
curl 127.0.0.1:80
```

you should see following result:

```
ok
```

in your console, that running the app, you will see this results:

```
> Request received: HTTP GET /
> Client IP: ::ffff:127.0.0.1
```

第20.2节：HTTP客户端

HTTP客户端的一个基本示例：

在http_client.js文件中写入以下代码：

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

var options = {
  主机名: '127.0.0.1',
  端口: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(options, function(res) {
  console.log('状态: ' + res.statusCode);
  console.log('头信息: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('数据', function(chunk) {
    console.log('响应: ' + chunk);
  });
  res.on('结束', function(chunk) {
    console.log('响应结束');
  });
});

req.on('错误', function(e) {
  console.log('请求出现问题: ' + e.message);
});

req.end();
```

然后在你的 http_client.js 所在位置运行此命令：

```
node http_client.js
```

你应该会看到以下结果：

```
> 状态: 200
> 头信息: {"content-type": "text/plain", "date": "Thu, 21 Jul 2016 11:27:17
  GMT", "connection": "close", "transfer-encoding": "chunked"}
> 响应: OK
> 响应结束
```

注意：此示例依赖于 http 服务器示例。

Section 20.2: http client

a basic example for http client:

write the following code in http_client.js file:

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

var options = {
  hostname: '127.0.0.1',
  port: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function(chunk) {
    console.log('Response: ' + chunk);
  });
  res.on('end', function(chunk) {
    console.log('Response ENDED');
  });
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

req.end();
```

then from your http_client.js location run this command:

```
node http_client.js
```

you should see this result:

```
> STATUS: 200
> HEADERS: {"content-type": "text/plain", "date": "Thu, 21 Jul 2016 11:27:17
  GMT", "connection": "close", "transfer-encoding": "chunked"}
> Response: OK
> Response ENDED
```

note: this example depend on http server example.

第21章：使用流

参数	定义
可读流	一种可以读取数据的流类型
可写流	一种可以写入数据的流类型
双工流	既可读又可写的流类型
转换流	一种双工流，可以在读取和写入数据时对数据进行转换

第21.1节：使用流从文本文件读取数据

Node 中的 I/O 是异步的，因此与磁盘和网络交互时需要向函数传递回调。你可能会想写出如下代码来从磁盘提供文件：

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

var server = http.createServer(function (req, res) {
  fs.readFile(__dirname + '/data.txt', function (err, data) {
    res.end(data);
  });
server.listen(8000);
```

这段代码可以运行，但它笨重且会在每次请求时将整个 data.txt 文件缓冲到内存中，然后再将结果写回客户端。如果 data.txt 文件非常大，当程序同时为大量用户服务时，尤其是连接较慢的用户，程序可能会开始占用大量内存。

用户体验也很差，因为用户需要等待整个文件被缓冲到服务器内存中，才能开始接收任何内容。

幸运的是，(req, res) 两个参数都是流，这意味着我们可以用更好的方式来写这段代码，使用 fs.createReadStream() 替代 fs.readFile()：

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

var server = http.createServer(function (req, res) {
  var stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(res);
});
server.listen(8000);
```

这里 .pipe() 负责监听 fs.createReadStream() 的 'data' 和 'end' 事件。这段代码不仅更简洁，而且 data.txt 文件会被一块一块地立即写入客户端，数据从磁盘读取后立刻传输。

第21.2节：管道流

可读流可以“管道”连接到可写流。这使得数据从源流流向目标流变得非常简单。

```
var fs = require('fs')

var readable = fs.createReadStream('file1.txt')
```

Chapter 21: Using Streams

Parameter	Definition
Readable Stream	type of stream where data can be read from
Writable Stream	type of stream where data can be written to
Duplex Stream	type of stream that is both readable and writeable
Transform Stream	type of duplex stream that can transform data as it is being read and then written

Section 21.1: Read Data from TextFile with Streams

I/O in node is asynchronous, so interacting with the disk and network involves passing callbacks to functions. You might be tempted to write code that serves up a file from disk like this:

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

var server = http.createServer(function (req, res) {
  fs.readFile(__dirname + '/data.txt', function (err, data) {
    res.end(data);
  });
});
server.listen(8000);
```

This code works but it's bulky and buffers up the entire data.txt file into memory for every request before writing the result back to clients. If data.txt is very large, your program could start eating a lot of memory as it serves lots of users concurrently, particularly for users on slow connections.

The user experience is poor too because users will need to wait for the whole file to be buffered into memory on your server before they can start receiving any contents.

Luckily both of the (req, res) arguments are streams, which means we can write this in a much better way using fs.createReadStream() instead of fs.readFile():

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

var server = http.createServer(function (req, res) {
  var stream = fs.createReadStream(__dirname + '/data.txt');
  stream.pipe(res);
});
server.listen(8000);
```

Here .pipe() takes care of listening for 'data' and 'end' events from the fs.createReadStream(). This code is not only cleaner, but now the data.txt file will be written to clients one chunk at a time immediately as they are received from the disk.

Section 21.2: Piping streams

Readable streams can be "piped," or connected, to writable streams. This makes data flow from the source stream to the destination stream without much effort.

```
var fs = require('fs')

var readable = fs.createReadStream('file1.txt')
```

```
var writable = fs.createWriteStream('file2.txt')
readable.pipe(writable) // 返回 writable
```

当可写流同时也是可读流，即它们是双工流时，你可以继续将其管道传输到其他可写流。

```
var zlib = require('zlib')

fs.createReadStream('style.css')
.pipe(zlib.createGzip()) // 返回的对象 zlib.Gzip 是一个双工流。
.pipe(fs.createWriteStream('style.css.gz'))
```

可读流也可以被管道传输到多个流中。

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.css.gz'))
readable.pipe(fs.createWriteStream('output.css'))
```

请注意，必须在任何数据“流动”之前同步（同时）将数据管道传输到输出流。否则可能导致数据流不完整。

还要注意，流对象可能会触发错误事件；请务必在每个流上根据需要负责地处理这些事件：

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

第21.3节：创建你自己的可读/可写流

我们会看到像 `fs` 这样的模块返回流对象，但如果我想创建自己的流对象怎么办？

要创建流对象，我们需要使用 Node.js 提供的 `stream` 模块

```
var fs = require("fs");
var stream = require("stream").Writable;

/*
 * 在可写流类中实现 write 函数。
 * 当其他流被管道传入此可写流时，将使用此函数。
 */
stream.prototype._write = function(chunk, data){
    console.log(data);
}

var customStream = new stream();

fs.createReadStream("am1.js").pipe(customStream);
```

这将给我们自己的自定义可写流。我们可以在 `_write` 函数中实现任何内容。上述方法适用于 Node.js 4.x.x 版本，但在 Node.js 6.x 版本中，**ES6** 引入了类，因此语法发生了变化。以下是 Node.js 6.x 版本的代码

```
var writable = fs.createWriteStream('file2.txt')
```

```
readable.pipe(writable) // returns writable
```

When writable streams are also readable streams, i.e. when they're *duplex* streams, you can continue piping it to other writable streams.

```
var zlib = require('zlib')

fs.createReadStream('style.css')
.pipe(zlib.createGzip()) // The returned object, zlib.Gzip, is a duplex stream.
.pipe(fs.createWriteStream('style.css.gz'))
```

Readable streams can also be piped into multiple streams.

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.css.gz'))
readable.pipe(fs.createWriteStream('output.css'))
```

Note that you must pipe to the output streams synchronously (at the same time) before any data 'flows'. Failure to do so might lead to incomplete data being streamed.

Also note that stream objects can emit error events; be sure to responsibly handle these events on *every* stream, as needed:

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

Section 21.3: Creating your own readable/writable stream

We will see stream objects being returned by modules like `fs` etc but what if we want to create our own streamable object.

To create Stream object we need to use the `stream` module provided by Node.js

```
var fs = require("fs");
var stream = require("stream").Writable;

/*
 * Implementing the write function in writable stream class.
 * This is the function which will be used when other stream is piped into this
 * writable stream.
 */
stream.prototype._write = function(chunk, data){
    console.log(data);
}

var customStream = new stream();

fs.createReadStream("am1.js").pipe(customStream);
```

This will give us our own custom writable stream. we can implement anything within the `_write` function. Above method works in Node.js 4.x.x version but in Node.js 6.x **ES6** introduced classes therefore syntax have changed. Below is the code for 6.x version of Node.js

```

const Writable = require('stream').Writable;

class MyWritable extends Writable {
constructor(options) {
    super(options);
}

_write(chunk, encoding, callback) {
    console.log(chunk);
}
}

```

第21.4节：为什么使用流？

让我们检查以下两个读取文件内容的示例：

第一个示例，使用异步方法读取文件，并提供一个回调函数，该函数在文件完全读入内存后被调用：

```

fs.readFile(`$__dirname/utils.js`, (err, data) => {
    if (err) {
        handleError(err);
    } else {
        console.log(data.toString());
    }
})

```

第二个示例使用streams来逐块读取文件内容：

```

var fileStream = fs.createReadStream(`$__dirname/file`);
var fileContent = '';
fileStream.on('data', data => {
    fileContent += data.toString();
})

fileStream.on('end', () => {
    console.log(fileContent);
})

fileStream.on('error', err => {
    handleError(err)
})

```

值得一提的是，这两个示例做的是完全相同的事情。那么区别是什么呢？

- 第一个更简短，看起来更优雅
- 第二个允许你在文件**读取过程中**进行一些处理（!）

当你处理的文件很小时，使用streams不会有太大影响，但当文件很大时会发生什么呢？（大到需要10秒钟才能读入内存）

如果不使用streams，你将一直等待，什么也做不了（除非你的进程还做其他事情），直到10秒过去，文件被完全读取，之后你才能开始处理文件。

使用streams，你可以分块获取文件内容，在内容可用时立即获取——这让你可以在文件读取的同时处理它。

```

const Writable = require('stream').Writable;

class MyWritable extends Writable {
constructor(options) {
    super(options);
}

_write(chunk, encoding, callback) {
    console.log(chunk);
}
}

```

Section 21.4: Why Streams?

Lets examine the following two examples for reading a file's contents:

The first one, which uses an `async` method for reading a file, and providing a callback function which is called once the file is fully read into the memory:

```

fs.readFile(`$__dirname/utils.js`, (err, data) => {
    if (err) {
        handleError(err);
    } else {
        console.log(data.toString());
    }
})

```

And the second, which uses `streams` in order to read the file's content, piece by piece:

```

var fileStream = fs.createReadStream(`$__dirname/file`);
var fileContent = '';
fileStream.on('data', data => {
    fileContent += data.toString();
})

fileStream.on('end', () => {
    console.log(fileContent);
})

fileStream.on('error', err => {
    handleError(err)
})

```

It's worth mentioning that both examples do the **exact same thing**. What's the difference then?

- The first one is shorter and looks more elegant
- The second lets you do some processing on the file **while** it is being read (!)

When the files you deal with are small then there is no real effect when using `streams`, but what happens when the file is big? (so big that it takes 10 seconds to read it into memory)

Without `streams` you'll be waiting, doing absolutely nothing (unless your process does other stuff), until the 10 seconds pass and the file is **fully read**, and only then you can start processing the file.

With `streams`, you get the file's contents piece by piece, **right when they're available** - and that lets you process the file **while** it is being read.

上面的例子没有展示如何利用streams来完成回调方式无法完成的工作，所以我们来看另一个例子：

我想下载一个gzip文件，解压它并将内容保存到磁盘。给定文件的url，需要完成以下操作：

- 下载文件
- 解压文件
- 保存到磁盘

这里有一个存储在我S3存储中的[小文件][1]。下面的代码以回调方式完成上述操作。

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data) => {
  // 这里，整个文件已被下载

  zlib.gunzip(data.Body, (err, data) => {
    // 这里，整个文件已被解压

    fs.writeFile(`$__dirname/tweets.json`, data, err => {
      if (err) console.error(err)

      // 这里，整个文件已被写入磁盘
      var endTime = Date.now()
      console.log(`${endTime - startTime} 毫秒`) // 1339 毫秒
    })
  })
}

// 1339 毫秒
```

这就是使用streams的效果：

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream(`$__dirname/tweets.json`));

// 1204 毫秒
```

是的，处理小文件时并不更快——测试的文件大小为80KB。对一个更大的文件进行测试，71MB压缩后（解压后为382MB），显示streams版本快得多

- 下载71MB、解压并写入382MB到磁盘花费了20925毫秒——使用回调函数方式。
- 相比之下，使用streams版本完成相同操作只花了13434毫秒（快了35%，对于一个不算太大的文件）

The above example does not illustrate how streams can be utilized for work that cannot be done when going the callback fashion, so let's look at another example:

I would like to download a gzip file, unzip it and save its content to the disk. Given the file's url this is what's need to be done:

- Download the file
- Unzip the file
- Save it to disk

Here's a [small file][1], which is stored in my S3 storage. The following code does the above in the callback fashion.

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data) => {
  // here, the whole file was downloaded

  zlib.gunzip(data.Body, (err, data) => {
    // here, the whole file was unzipped

    fs.writeFile(`$__dirname/tweets.json`, data, err => {
      if (err) console.error(err)

      // here, the whole file was written to disk
      var endTime = Date.now()
      console.log(`${endTime - startTime} milliseconds`) // 1339 milliseconds
    })
  })
}

// 1339 milliseconds
```

This is how it looks using streams:

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream(`$__dirname/tweets.json`));

// 1204 milliseconds
```

Yep, it's not faster when dealing with small files - the tested file weights 80KB. Testing this on a bigger file, 71MB gzipped (382MB unzipped), shows that the streams version is much faster

- It took 20925 milliseconds to download 71MB, unzip it and then write 382MB to disk - **using the callback fashion**.
- In comparison, it took 13434 milliseconds to do the same when using the streams version (35% faster, for a not-so-big file)

第22章：在生产环境中部署Node.js应用程序

第22.1节：设置NODE_ENV="production"

生产环境的部署方式会有很多不同，但在生产环境部署时的一个标准惯例是定义一个名为NODE_ENV的环境变量，并将其值设置为"production"。

运行时标志

您应用程序中运行的任何代码（包括外部模块）都可以检查NODE_ENV的值：

```
if(process.env.NODE_ENV === 'production') {
  // 我们正在生产模式下运行
} else {
  // 我们正在开发模式下运行
}
```

依赖项

当NODE_ENV环境变量设置为'production'时，package.json文件中的所有devDependencies在运行npm install时将被完全忽略。你也可以通过--production标志来强制执行：

```
npm install --production
```

设置NODE_ENV你可以使用以下任一方法

方法1：为所有Node应用设置NODE_ENV

Windows：

```
set NODE_ENV=production
```

Linux或其他基于Unix的系统：

```
export NODE_ENV=production
```

这会为当前的bash会话设置NODE_ENV，因此在此语句之后启动的任何应用程序的NODE_ENV都会被设置为production。

方法2：为当前应用设置NODE_ENV

```
NODE_ENV=production node app.js
```

这只会为当前应用设置NODE_ENV。当我们想在不同环境中测试应用时，这很有用。

方法3：创建.env文件并使用它

这使用了前面解释的想法。详细说明请参考这篇文章。

基本上，你创建一个.env文件，并运行一些bash脚本来设置环境变量。

为了避免编写bash脚本，可以使用env-cmd包来加载在

Chapter 22: Deploying Node.js applications in production

Section 22.1: Setting NODE_ENV="production"

Production deployments will vary in many ways, but a standard convention when deploying in production is to define an environment variable called NODE_ENV and set its value to "production".

Runtime flags

Any code running in your application (including external modules) can check the value of NODE_ENV:

```
if(process.env.NODE_ENV === 'production') {
  // We are running in production mode
} else {
  // We are running in development mode
}
```

Dependencies

When the NODE_ENV environment variable is set to 'production' all devDependencies in your package.json file will be completely ignored when running npm `install`. You can also enforce this with a --production flag:

```
npm install --production
```

For setting NODE_ENV you can use any of these methods

method 1: set NODE_ENV for all node apps

Windows：

```
set NODE_ENV=production
```

Linux or other unix based system：

```
export NODE_ENV=production
```

This sets NODE_ENV for current bash session thus any apps started after this statement will have NODE_ENV set to production.

method 2: set NODE_ENV for current app

```
NODE_ENV=production node app.js
```

This will set NODE_ENV for the current app only. This helps when we want to test our apps on different environments.

method 3: create .env file and use it

This uses the idea explained [here](#). Refer this post for more detailed explanation.

Basically you create .env file and run some bash script to set them on environment.

To avoid writing a bash script, the [env-cmd](#) package can be used to load the environment variables defined in the

```
env-cmd .env node app.js
```

方法4：使用cross-env包

该package允许在所有平台上以统一方式设置环境变量。

通过npm安装后，你只需将其添加到部署脚本的package.json中，示例如下：

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

第22.2节：使用进程管理器管理应用

使用进程管理器控制NodeJS应用是一种良好实践。进程管理器有助于让应用永远保持运行状态，失败时自动重启，无停机重载，并简化管理。它们中最强大的（如PM2）内置了负载均衡器。PM2还支持应用日志管理、监控和集群功能。

PM2进程管理器

安装PM2：

```
npm install pm2 -g
```

进程可以以集群模式启动，利用集成的负载均衡器在进程间分配负载：

```
pm2 start app.js -i 0 --name "api" (-i用于指定启动进程数。如果为0，则进程数基于CPU核心数)
```

在生产环境中多个用户时，必须为PM2设置一个单一入口点。因此，pm2命令必须以一个位置（用于PM2配置）作为前缀，否则它会为每个用户在各自的主目录中生成一个新的pm2进程，这样会导致不一致。

用法：PM2_HOME=/etc/.pm2 pm2 start app.js

第 22.3 节：使用进程管理器进行部署

进程管理器通常用于生产环境中部署Node.js应用。进程管理器的主要功能包括在服务器崩溃时重启服务器、检查资源消耗、提升运行时性能、监控等。

一些由Node社区开发的流行进程管理器有forever、pm2等。

Forever

forever是一个命令行界面工具，用于确保指定脚本持续运行。forever简单的界面使其非常适合运行较小规模的Node.js应用和脚本部署。

forever会监控你的进程并在其崩溃时重启。

全局安装forever。

```
$ npm install -g forever
```

运行应用程序：

.env file.

```
env-cmd .env node app.js
```

method 4: Use cross-env package

This package allows environment variables to be set in one way for every platform.

After installing it with npm, you can just add it to your deployment script in package.json as follows:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

Section 22.2: Manage app with process manager

It's a good practice to run NodeJS apps controlled by process managers. Process manager helps to keep application alive forever, restart on failure, reload without downtime and simplifies administrating. Most powerful of them (like PM2) have a built-in load balancer. PM2 also enables you to manage application logging, monitoring, and clustering.

PM2 process manager

Installing PM2:

```
npm install pm2 -g
```

Process can be started in cluster mode involving integrated load balancer to spread load between processes:

```
pm2 start app.js -i 0 --name "api" (-i is to specify number of processes to spawn. If it is 0, then process number will be based on CPU cores count)
```

While having multiple users in production, its must to have a single point for PM2. Therefore pm2 command must be prefixed with a location (for PM2 config) else it will spawn a new pm2 process for every user with config in respective home directory. And it will be inconsistent.

Usage: PM2_HOME=/etc/.pm2 pm2 start app.js

Section 22.3: Deployment using process manager

Process manager is generally used in production to deploy a nodejs app. The main functions of a process manager are restarting the server if it crashes, checking resource consumption, improving runtime performance, monitoring etc.

Some of the popular process managers made by the node community are forever, pm2, etc.

Forever

forever is a command-line interface tool for ensuring that a given script runs continuously. forever's simple interface makes it ideal for running smaller deployments of Node.js apps and scripts.

forever monitors your process and restarts it if it crashes.

Install forever globally.

```
$ npm install -g forever
```

Run application :

```
$ forever start server.js
```

这将启动服务器并为进程分配一个ID（从0开始）。

重启应用程序：

```
$ forever restart 0
```

这里的0是服务器的ID。

停止应用程序：

```
$ forever stop 0
```

与重启类似，0是服务器的ID。你也可以用进程ID或脚本名称替代forever给出的ID。

更多命令请见：<https://www.npmjs.com/package/forever>

第22.4节：使用PM2进行部署

PM2 是一个用于Node.js 应用的生产进程管理器，允许你让应用永远保持运行并且在不中断的情况下重新加载它们。PM2 还支持管理应用日志、监控和集群。

全局安装 pm2。

```
npm install -g pm2
```

然后，使用 PM2 运行 node.js 应用。

```
pm2 start server.js --name "my-app"
```

```
$ pm2 start app.js --name my-app  
[PM2] restartProcessId process id 0
```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-app	0	fork	64029	online	1	0s	17.816 MB	disabled

Use the `pm2 show <id|name>` command to get more details about an app.

以下命令在使用 PM2 时非常有用。

列出所有运行中的进程：

```
pm2 list
```

停止一个应用：

```
pm2 stop my-app
```

```
$ forever start server.js
```

This starts the server and gives an id for the process(starts from 0).

Restart application :

```
$ forever restart 0
```

Here 0 is the id of the server.

Stop application :

```
$ forever stop 0
```

Similar to restart, 0 is the id the server. You can also give process id or script name in place of the id given by the forever.

For more commands : <https://www.npmjs.com/package/forever>

Section 22.4: Deployment using PM2

PM2 is a production process manager for Node.js applications, that allows you to keep applications alive forever and reload them without downtime. PM2 also enables you to manage application logging, monitoring, and clustering.

Install pm2 globally.

```
npm install -g pm2
```

Then, run the node.js app using PM2.

```
pm2 start server.js --name "my-app"
```

```
$ pm2 start app.js --name my-app  
[PM2] restartProcessId process id 0
```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-app	0	fork	64029	online	1	0s	17.816 MB	disabled

Use the `pm2 show <id|name>` command to get more details about an app.

Following commands are useful while working with PM2.

List all running processes:

```
pm2 list
```

Stop an app:

```
pm2 stop my-app
```

重启一个应用：

```
pm2 restart my-app
```

查看应用程序的详细信息：

```
pm2 show my-app
```

从 PM2 的注册表中移除应用程序：

```
pm2 delete my-app
```

第22.5节：为不同环境（如开发、测试、预发布等）使用不同的属性/配置

大型应用程序在不同环境下运行时通常需要不同的属性。我们可以通过向 Node.js 应用传递参数，并在节点进程中使用相同参数来加载特定的环境属性文件，从而实现这一点。

假设我们有两个用于不同环境的属性文件。

- dev.json

```
{  
  "PORT": 3000,  
  "DB": {  
    "host": "localhost",  
    "user": "bob",  
    "password": "12345"  
  }  
}
```

- qa.json

```
{  
  "PORT": 3001,  
  "DB": {  
    "host": "where_db_is_hosted",  
    "user": "bob",  
    "password": "54321"  
  }  
}
```

应用中的以下代码将导出我们想要使用的相应属性文件。

```
process.argv.forEach(function (val) {  
  var arg = val.split("=".  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./' + arg[1] + '.json');  
      exports.prop = env;  
    }  
  }  
})
```

Restart an app:

```
pm2 restart my-app
```

To view detailed information about an app:

```
pm2 show my-app
```

To remove an app from PM2's registry:

```
pm2 delete my-app
```

Section 22.5: Using different Properties/Configuration for different environments like dev, qa, staging etc

Large scale applications often need different properties when running on different environments. we can achieve this by passing arguments to Node.js application and using same argument in node process to load specific environment property file.

Suppose we have two property files for different environment.

- dev.json

```
{  
  "PORT": 3000,  
  "DB": {  
    "host": "localhost",  
    "user": "bob",  
    "password": "12345"  
  }  
}
```

- qa.json

```
{  
  "PORT": 3001,  
  "DB": {  
    "host": "where_db_is_hosted",  
    "user": "bob",  
    "password": "54321"  
  }  
}
```

Following code in application will export respective property file which we want to use.

```
process.argv.forEach(function (val) {  
  var arg = val.split("=".  
  if (arg.length > 0) {  
    if (arg[0] === 'env') {  
      var env = require('./' + arg[1] + '.json');  
      exports.prop = env;  
    }  
  }  
})
```

```
});
```

我们给应用传递的参数如下

```
node app.js env=dev
```

如果我们使用像forever这样的进程管理器，那么就很简单

```
forever start app.js env=dev
```

第22.6节：利用集群

Node.js的单个实例运行在单线程中。为了利用多核系统，用户有时会想启动一个Node.js进程集群来处理负载。

```
var cluster = require('cluster');

var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // 在实际应用中，你可能会使用超过2个工作进程，
    // 并且可能不会将主进程和工作进程放在同一个文件中。
    //
    // 你当然也可以对日志记录进行更复杂的处理，
    // 实现你需要的任何自定义逻辑来防止拒绝服务 (DoS)
    // 攻击和其他不良行为。
    //
    // 请参阅集群文档中的选项。
    //
    // 重要的是主进程做的事情非常少，
    // 这提高了我们对意外错误的恢复能力。
    // This increases our resilience to unexpected errors.
    console.log('你的服务器正在使用 ' + numCPUs + ' 个核心');

    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('disconnect', function(worker) {
        console.error('断开连接！');
        //clearTimeout(timeout);
        cluster.fork();
    });

} else {
    require('./app.js');
}
```

```
});
```

We give arguments to the application like following

```
node app.js env=dev
```

if we are using process manager like *forever* than it as simple as

```
forever start app.js env=dev
```

Section 22.6: Taking advantage of clusters

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems the user will sometimes want to launch a cluster of Node.js processes to handle the load.

```
var cluster = require('cluster');

var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // In real life, you'd probably use more than just 2 workers,
    // and perhaps not put the master and worker in the same file.
    //
    // You can also of course get a bit fancier about logging, and
    // implement whatever custom logic you need to prevent DoS
    // attacks and other bad behavior.
    //
    // See the options in the cluster documentation.
    //
    // The important thing is that the master does very little,
    // increasing our resilience to unexpected errors.
    console.log('your server is working on ' + numCPUs + ' cores');

    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('disconnect', function(worker) {
        console.error('disconnect!');
        //clearTimeout(timeout);
        cluster.fork();
    });

} else {
    require('./app.js');
}
```

第23章：保护Node.js应用程序

第23.1节：Node.js中的SSL/TLS

如果您选择在Node.js应用程序中处理SSL/TLS，请考虑您也需要负责此时的SSL/TLS攻击防护。在许多服务器-客户端架构中，SSL/TLS通常终止于反向代理，既可以减少应用程序的复杂性，也可以缩小安全配置的范围。

如果您的Node.js应用程序需要处理SSL/TLS，可以通过加载密钥和证书文件来实现安全。

如果您的证书提供商要求证书颁发机构（CA）链，可以通过ca选项以数组形式添加。包含多个条目的链如果在单个文件中，必须拆分成多个文件，并按顺序放入数组中，因为Node.js当前不支持单个文件中有多个ca条目。下面代码示例中使用了文件1_ca.crt和2_ca.crt。如果需要ca数组但未正确设置，客户端浏览器可能会显示无法验证证书真实性的提示。

示例

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

const options = {
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem'),
  ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world');
}).listen(8000);
```

第23.2节：防止跨站请求伪造 (CSRF)

CSRF是一种攻击，强制终端用户在其当前已认证的网络应用程序上执行不想要的操作。

这可能是因为每个请求都会携带 Cookie——即使这些请求来自不同的网站。

我们可以使用csurf模块来创建 CSRF 令牌并进行验证。

示例

```
var express = require('express')
var cookieParser = require('cookie-parser') // 用于解析 Cookie
var csrf = require('csurf') // CSRF 模块
var bodyParser = require('body-parser') // 用于解析请求体

// 设置路由中间件
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// 创建 express 应用
var app = express()
```

Chapter 23: Securing Node.js applications

Section 23.1: SSL/TLS in Node.js

If you choose to handle SSL/TLS in your Node.js application, consider that you are also responsible for maintaining SSL/TLS attack prevention at this point. In many server-client architectures, SSL/TLS terminates on a reverse proxy, both to reduce application complexity and reduce the scope of security configuration.

If your Node.js application should handle SSL/TLS, it can be secured by loading the key and cert files.

If your certificate provider requires a certificate authority (CA) chain, it can be added in the ca option as an array. A chain with multiple entries in a single file must be split into multiple files and entered in the same order into the array as Node.js does not currently support multiple ca entries in one file. An example is provided in the code below for files `1_ca.crt` and `2_ca.crt`. If the ca array is required and not set properly, client browsers may display messages that they could not verify the authenticity of the certificate.

Example

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

const options = {
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem'),
  ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Section 23.2: Preventing Cross Site Request Forgery (CSRF)

CSRF is an attack which forces end user to execute unwanted actions on a web application in which he/she is currently authenticated.

It can happen because cookies are sent with every request to a website - even when those requests come from a different site.

We can use `csurf` module for creating csrf token and validating it.

Example

```
var express = require('express')
var cookieParser = require('cookie-parser') //for cookie parsing
var csrf = require('csurf') //csrf module
var bodyParser = require('body-parser') //for body parsing

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()
```

```
// 解析 Cookie
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
  // 生成并传递 csrfToken 到视图
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('数据正在处理')
})
```

所以，当我们访问GET/form时，它会将csrf令牌csrfToken传递给视图。

现在，在视图中，将csrfToken值设置为名为 csrf的隐藏输入字段的值。

例如，针对handlebar模板

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}>
  名称: <input type="text" name="name">
  <button type="submit">提交</button>
</form>
```

例如，针对jade模板

```
form(action="/process" method="post")
  input(type="hidden", name="_csrf", value=csrfToken)

  span 名称:
  input(type="text", name="name", required=true)
  br

  input(type="submit")
```

例如，用于ejs模板

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  名称: <input type="text" name="name">
  <button type="submit">提交</button>
</form>
```

第23.3节：设置HTTPS服务器

一旦你在系统上安装了node.js，只需按照以下步骤即可运行一个支持HTTP和HTTPS的基本网络服务器！

步骤 1：建立证书颁发机构

1. 创建一个用于存放密钥和证书的文件夹：

```
mkdir conf
```

```
// parse cookies
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
  // generate and pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('data is being processed')
})
```

So, when we access GET /form, it will pass the csrf token csrfToken to the view.

Now, inside the view, set the csrfToken value as the value of a hidden input field named _csrf.

e.g. for handlebar templates

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}>
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

e.g. for jade templates

```
form(action="/process" method="post")
  input(type="hidden", name="_csrf", value=csrfToken)

  span Name:
  input(type="text", name="name", required=true)
  br

  input(type="submit")
```

e.g. for ejs templates

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

Section 23.3: Setting up an HTTPS server

Once you have node.js installed on your system, just follow the procedure below to get a basic web server running with support for both HTTP and HTTPS!

Step 1 : Build a Certificate Authority

1. create the folder where you want to store your key & certificate :

```
mkdir conf
```

2. 进入该目录：

```
cd conf
```

3. 获取此ca.cnf文件，作为配置快捷方式使用：

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. 使用此配置创建一个新的证书颁发机构：

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. 现在我们已经有了证书颁发机构的 ca-key.pem 和 ca-cert.pem，接下来生成服务器的私钥：服务器：

```
openssl genrsa -out key.pem 4096
```

6. 获取这个 server.cnf 文件作为配置快捷方式：

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. 使用此配置生成证书签名请求：

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. 签署请求：

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

2. go to that directory :

```
cd conf
```

3. grab this ca.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. create a new certificate authority using this configuration :

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. now that we have our certificate authority in ca-key.pem and ca-cert.pem, let's generate a private key for the server :

```
openssl genrsa -out key.pem 4096
```

6. grab this server.cnf file to use as a configuration shortcut :

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. generate the certificate signing request using this configuration :

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. sign the request :

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

步骤2：将您的证书安装为根证书

1. 将您的证书复制到根证书文件夹：

Step 2 : Install your certificate as a root certificate

1. copy your certificate to your root certificates' folder :

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. 更新CA存储：

```
sudo update-ca-certificates
```

第23.4节：使用HTTPS

Node.js中HTTPS服务器的最小配置大致如下：

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

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world");
}

https.createServer(httpsOptions, app).listen(4433);
```

如果你还想支持http请求，只需做这个小修改：

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

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

第23.5节：安全的express.js 3应用程序

使用 express.js（自版本 3 起）建立安全连接的配置：

```
var fs = require('fs');
var http = require('http');
var https = require('https');
var privateKey = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');
```

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. update CA store :

```
sudo update-ca-certificates
```

Section 23.4: Using HTTPS

The minimal setup for an HTTPS server in Node.js would be something like this :

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

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

If you also want to support http requests, you need to make just this small modification:

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

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

Section 23.5: Secure express.js 3 Application

The configuration to make a secure connection using express.js (Since version 3):

```
var fs = require('fs');
var http = require('http');
var https = require('https');
var privateKey = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');
```

```
// 定义您的密钥和证书
```

```
var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();

// 这里是您的 express 配置

var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);

// HTTP 使用端口 8080, HTTPS 使用端口 8443

httpServer.listen(8080);
httpsServer.listen(8443);
```

通过这种方式，您将 express 中间件提供给原生的 http/https 服务器

如果您想让应用运行在 1024 以下的端口，您需要使用 sudo 命令（不推荐）或使用反向代理（例如 nginx、haproxy）。

```
// Define your key and cert
```

```
var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();

// your express configuration here

var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);

// Using port 8080 for http and 8443 for https

httpServer.listen(8080);
httpsServer.listen(8443);
```

In that way you provide express middleware to the native http/https server

If you want your app running on ports below 1024, you will need to use sudo command (not recommended) or use a reverse proxy (e.g. nginx, haproxy).

第24章：Mongoose库

第24.1节：使用Mongoose连接MongoDB

首先，安装Mongoose：

```
npm install mongoose
```

然后，将其作为依赖添加到server.js中：

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

接下来，创建数据库模式和集合名称：

```
var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});
```

创建一个模型并连接到数据库：

```
var Model = mongoose.model('模型', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
```

接下来，启动 MongoDB 并使用 node server.js 运行 server.js 要检查是否成功连接到数据库，我们可以使用来自 mongoose.connection 对象的事件 open 和 error。

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, '连接错误:'));
db.once('open', function() {
  // 我们已连接！
});
```

第24.2节：使用 Mongoose、Express.js 路由和 \$text 操作符在 MongoDB 中查找数据

设置

首先，使用以下命令安装必要的包：

```
npm install express cors mongoose
```

代码

然后，添加对 server.js 的依赖，创建数据库模式和集合名称，创建一个 Express.js 服务器，并连接到 MongoDB：

```
var express = require('express');
var cors = require('cors'); // 我们将使用CORS来启用跨域请求。
var mongoose = require('mongoose');
```

Chapter 24: Mongoose Library

Section 24.1: Connect to MongoDB Using Mongoose

First, install Mongoose with:

```
npm install mongoose
```

Then, add it to server.js as dependencies:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

Next, create the database schema and the name of the collection:

```
var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});
```

Create a model and connect to the database:

```
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
```

Next, start MongoDB and run server.js using node server.js

To check if we have successfully connected to the database, we can use the events open, error from the mongoose.connection object.

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

Section 24.2: Find Data in MongoDB Using Mongoose, Express.js Routes and \$text Operator

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to server.js, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
```

```

var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('模型', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js监听端口 ' + port);
});

```

现在添加我们将用来查询数据的Express.js路由：

```

app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error: '错误'
      }))
    }
  })
}

```

假设以下文档存在于模型中的集合中：

```

{
  "_id": ObjectId("578abe97522ad414b8eeb55a"),
  "request": "JavaScript 很棒",
  "time": 1468710551
}

{
  "_id": ObjectId("578abe9b522ad414b8eeb55b"),
  "request": "JavaScript 很棒",
  "time": 1468710555
}

{
  "_id": ObjectId("578abea0522ad414b8eeb55c"),
  "request": "JavaScript 很棒",
  "time": 1468710560
}

```

目标是查找并显示所有在"request"

为此，首先在集合中为"request"创建一个文本索引。为此，请将以下代码添加到server.js中：

```

var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});

```

Now add Express.js routes that we will use to query the data:

```

app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error: 'Error'
      }))
    }
  })
}

```

Assume that the following documents are in the collection in the model:

```

{
  "_id": ObjectId("578abe97522ad414b8eeb55a"),
  "request": "JavaScript is Awesome",
  "time": 1468710551
}

{
  "_id": ObjectId("578abe9b522ad414b8eeb55b"),
  "request": "JavaScript is Awesome",
  "time": 1468710555
}

{
  "_id": ObjectId("578abea0522ad414b8eeb55c"),
  "request": "JavaScript is Awesome",
  "time": 1468710560
}

```

And that the goal is to find and display all the documents containing only "JavaScript" word under the "request" key.

To do this, first create a *text index* for "request" in the collection. For this, add the following code to server.js:

```
schemaName.index({ request: 'text' });
```

并替换为：

```
Model.find({
  'request': query
}, function(err, result) {
```

替换为：

```
Model.find({
  $text: {
    $search: query
  }
}, function(err, result) {
```

这里，我们使用了\$text和\$search MongoDB操作符来查找集合collectionName 中包含指定查询中至少一个单词的所有文档。

用法

要使用此功能查找数据，请在浏览器中访问以下URL：

```
http://localhost:8080/find/<query>
```

其中 <query> 是搜索查询。

示例：

```
http://localhost:8080/find/JavaScript
```

输出：

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

第24.3节：使用Mongoose和Express.js路由将数据保存到MongoDB

设置

```
schemaName.index({ request: 'text' });
```

And replace:

```
Model.find({
  'request': query
}, function(err, result) {
```

With:

```
Model.find({
  $text: {
    $search: query
  }
}, function(err, result) {
```

Here, we are using \$text and \$search MongoDB operators for find all documents in collection collectionName which contains at least one word from the specified find query.

Usage

To use this to find data, go to the following URL in a browser:

```
http://localhost:8080/find/<query>
```

Where <query> is the search query.

Example:

```
http://localhost:8080/find/JavaScript
```

Output:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Section 24.3: Save Data to MongoDB using Mongoose and Express.js Routes

Setup

首先，使用以下命令安装必要的包：

```
npm install express cors mongoose
```

代码

然后，将依赖添加到你的server.js文件中，创建数据库模式和集合名称，创建一个Express.js服务器，并连接到MongoDB：

```
var express = require('express');
var cors = require('cors'); // 我们将使用CORS来启用跨域请求。
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('模型', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js监听端口 ' + port);
});
```

现在添加我们将用来写入数据的Express.js路由：

```
app.get('/save/:query', cors(), function(req, res) {
  var query = req.params.query;

  var savedata = new Model({
    'request': query,
    'time': Math.floor(Date.now() / 1000) // 保存数据的时间，Unix时间戳格式
  }).save(function(err, result) {
    if (err) throw err;

    if(result) {
      res.json(result)
    }
  })
})
```

这里的query变量将是来自传入HTTP请求的<query>参数，该参数将被保存到MongoDB：

```
var savedata = new Model({
  'request': query,
  //...
```

如果尝试写入MongoDB时发生错误，您将在控制台收到错误信息。如果一切成功，您将在页面上看到以JSON格式保存的数据。

```
//...
```

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to your server.js file, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Now add Express.js routes that we will use to write the data:

```
app.get('/save/:query', cors(), function(req, res) {
  var query = req.params.query;

  var savedata = new Model({
    'request': query,
    'time': Math.floor(Date.now() / 1000) // Time of save the data in unix timestamp format
  }).save(function(err, result) {
    if (err) throw err;

    if(result) {
      res.json(result)
    }
  })
})
```

Here the query variable will be the <query> parameter from the incoming HTTP request, which will be saved to MongoDB:

```
var savedata = new Model({
  'request': query,
  //...
```

If an error occurs while trying to write to MongoDB, you will receive an error message on the console. If all is successful, you will see the saved data in JSON format on the page.

```
//...
```

```
}).save(function(err, result) {
  if (err) throw err;

  if(result) {
    res.json(result)
  }
})
//...
```

现在，您需要启动MongoDB并使用node server.js运行您的server.js文件。

用法

要使用此功能保存数据，请在浏览器中访问以下URL：

```
http://localhost:8080/save/<query>
```

其中<query>是您希望保存的新请求。

示例：

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

JSON 格式输出：

```
{
  _v: 0,
  request: "JavaScript is Awesome",
  time: 1469411348,
  _id: "57957014b93bc8640f2c78c4"
}
```

第 24.4 节：使用 Mongoose 和 Express.js 路由在 MongoDB 中查找数据

Express.js 路由在 MongoDB 中查找数据

设置

首先，使用以下命令安装必要的包：

```
npm install express cors mongoose
```

代码

然后，添加对server.js的依赖，创建数据库模式和集合名称，创建一个Express.js服务器，并连接到MongoDB：

```
var express = require('express');
var cors = require('cors'); // 我们将使用CORS来启用跨域请求。
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
```

```
}).save(function(err, result) {
  if (err) throw err;

  if(result) {
    res.json(result)
  }
})
//...
```

Now, you need to start MongoDB and run your server.js file using node server.js.

Usage

To use this to save data, go to the following URL in your browser:

```
http://localhost:8080/save/<query>
```

Where <query> is the new request you wish to save.

Example:

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

Output in JSON format:

```
{
  __v: 0,
  request: "JavaScript is Awesome",
  time: 1469411348,
  _id: "57957014b93bc8640f2c78c4"
}
```

Section 24.4: Find Data in MongoDB Using Mongoose and Express.js Routes

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to server.js, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
```

```
});
```

```
var Model = mongoose.model('模型', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js 监听端口 ' + port);
});
```

现在添加我们将用来查询数据的Express.js路由：

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error: '错误'
      }))
    }
  })
})
```

假设以下文档存在于模型中的集合中：

```
{
  "_id": ObjectId("578abe97522ad414b8eeb55a"),
  "request": "JavaScript 很棒",
  "time": 1468710551
}

{
  "_id": ObjectId("578abe9b522ad414b8eeb55b"),
  "request": "JavaScript 很棒",
  "time": 1468710555
}

{
  "_id": ObjectId("578abaea0522ad414b8eeb55c"),
  "request": "JavaScript 很棒",
  "time": 1468710560
}
```

目标是查找并显示所有在"request"

为此，启动 MongoDB 并使用 node server.js 运行 server.js：

用法

要使用此功能查找数据，请在浏览器中访问以下 URL：

```
http://localhost:8080/find/<query>
```

其中 <query> 是搜索查询。

```
});
```

```
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Now add Express.js routes that we will use to query the data:

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error: 'Error'
      }))
    }
  })
})
```

Assume that the following documents are in the collection in the model:

```
{
  "_id": ObjectId("578abe97522ad414b8eeb55a"),
  "request": "JavaScript is Awesome",
  "time": 1468710551
}

{
  "_id": ObjectId("578abe9b522ad414b8eeb55b"),
  "request": "JavaScript is Awesome",
  "time": 1468710555
}

{
  "_id": ObjectId("578abaea0522ad414b8eeb55c"),
  "request": "JavaScript is Awesome",
  "time": 1468710560
}
```

And the goal is to find and display all the documents containing "JavaScript is Awesome" under the "request" key.

For this, start MongoDB and run server.js with node server.js:

Usage

To use this to find data, go to the following URL in a browser:

```
http://localhost:8080/find/<query>
```

Where <query> is the search query.

示例：

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

输出：

```
[ {  
  _id: "578abe97522ad414b8eeb55a",  
  request: "JavaScript is Awesome",  
  time: 1468710551,  
  __v: 0  
},  
,  
{  
  _id: "578abe9b522ad414b8eeb55b",  
  request: "JavaScript is Awesome",  
  time: 1468710555,  
  __v: 0  
},  
,  
{  
  _id: "578abea0522ad414b8eeb55c",  
  request: "JavaScript is Awesome",  
  time: 1468710560,  
  __v: 0  
}]
```

第24.5节：有用的Mongoose函数

Mongoose包含一些基于标准find()的内置函数。

```
doc.find({'some.value':5},function(err,docs){  
  //返回数组docs  
});  
  
doc.findOne({'some.value':5},function(err,doc){  
  //返回文档doc  
});  
  
doc.findById(obj._id,function(err,doc){  
  //返回文档doc  
});
```

第24.6节：模型中的索引

MongoDB支持二级索引。在Mongoose中，我们在模式中定义这些索引。当需要创建复合索引时，必须在模式级别定义索引。

Mongoose连接

```
var strConnection = 'mongodb://localhost:27017/dbName';  
var db = mongoose.createConnection(strConnection)
```

创建一个基本的架构

```
var Schema = require('mongoose').Schema;  
var usersSchema = new Schema({  
  username: {  
    type: String,
```

Example:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Output:

```
[ {  
  _id: "578abe97522ad414b8eeb55a",  
  request: "JavaScript is Awesome",  
  time: 1468710551,  
  __v: 0  
},  
,  
{  
  _id: "578abe9b522ad414b8eeb55b",  
  request: "JavaScript is Awesome",  
  time: 1468710555,  
  __v: 0  
},  
,  
{  
  _id: "578abea0522ad414b8eeb55c",  
  request: "JavaScript is Awesome",  
  time: 1468710560,  
  __v: 0  
}]
```

Section 24.5: Useful Mongoose functions

Mongoose contains some built in functions that build on the standard find().

```
doc.find({ 'some.value' :5 }, function(err,docs){  
  //returns array docs  
});  
  
doc.findOne({ 'some.value' :5 }, function(err,doc){  
  //returns document doc  
});  
  
doc.findById(obj._id, function(err,doc){  
  //returns document doc  
});
```

Section 24.6: Indexes in models

MongoDB supports secondary indexes. In Mongoose, we define these indexes within our schema. Defining indexes at schema level is necessary when we need to create compound indexes.

Mongoose Connection

```
var strConnection = 'mongodb://localhost:27017/dbName';  
var db = mongoose.createConnection(strConnection)
```

Creating a basic schema

```
var Schema = require('mongoose').Schema;  
var usersSchema = new Schema({  
  username: {  
    type: String,
```

```

required: true,
unique: true
},
email: {
  type: String,
  required: true
},
password: {
  type: String,
  required: true
},
created: {
  type: Date,
  default: Date.now
}
});

var usersModel = db.model('users', usersSchema);
module.exports = usersModel;

```

默认情况下，mongoose 会在我们的模型中添加两个新字段，即使这些字段未在模型中定义。这些字段是：

_id

如果在 Schema 构造函数中未传入 _id 字段，Mongoose 会默认为每个模式分配一个 _id 字段。分配的类型是 ObjectId，以符合 MongoDB 的默认行为。如果你完全不想在模式中添加 _id，可以使用此选项禁用它。

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  },
  _id: false
});

```

_v 或 versionKey

versionKey 是 Mongoose 在首次创建每个文档时设置的一个属性。该键的值包含文档的内部修订版本。此文档属性的名称是可配置的。

你可以在模型配置中轻松禁用此字段：

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  },
  versionKey: false
});

```

复合索引

我们可以创建除 Mongoose 创建的索引之外的其他索引。

```
usersSchema.index({username: 1});
```

```

required: true,
unique: true
},
email: {
  type: String,
  required: true
},
password: {
  type: String,
  required: true
},
created: {
  type: Date,
  default: Date.now
}
});

var usersModel = db.model('users', usersSchema);
module.exports = usersModel;

```

By default, mongoose adds two new fields into our model, even when those are not defined in the model. Those fields are:

_id

Mongoose assigns each of your schemas an _id field by default if one is not passed into the Schema constructor. The type assigned is an ObjectId to coincide with MongoDB's default behavior. If you don't want an _id added to your schema at all, you may disable it using this option.

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  },
  _id: false
});

```

_v or versionKey

The versionKey is a property set on each document when first created by Mongoose. This key's value contains the internal revision of the document. The name of this document property is configurable.

You can easily disable this field in the model configuration:

```

var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  },
  versionKey: false
});

```

Compound indexes

We can create another indexes besides those Mongoose creates.

```
usersSchema.index({username: 1});
```

```
usersSchema.index({email: 1});
```

在这些情况下，我们的模型有两个额外的索引，一个是针对字段 `username`，另一个是针对字段 `email`。但我们也就可以创建复合索引。

```
usersSchema.index({username: 1, email: 1});
```

索引性能影响

默认情况下，mongoose 总是依次调用 `ensureIndex` 为每个索引创建索引，并在所有 `ensureIndex` 调用成功或出现错误时，在模型上触发一个 'index' 事件。

在 MongoDB 中，`ensureIndex` 自 3.0.0 版本起已被弃用，现在是 `createIndex` 的别名。

建议通过将 schema 的 `autoIndex` 选项设置为 `false`，或通过将连接的 `config.autoIndex` 选项设置为 `false` 来全局禁用该行为。

```
usersSchema.set('autoIndex', false);
```

第24.7节：使用Promise在mongodb中查找数据

设置

首先，使用以下命令安装必要的包：

```
npm install express cors mongoose
```

代码

然后，添加对 `server.js` 的依赖，创建数据库模式和集合名称，创建一个 Express.js 服务器，并连接到 MongoDB：

```
var express = require('express');
var cors = require('cors'); // 我们将使用CORS来启用跨域请求。
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('模型', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js 监听端口 ' + port);
});

app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('发生错误！');
});
```

```
usersSchema.index({email: 1});
```

In these case our model have two more indexes, one for the field `username` and another for `email` field. But we can create compound indexes.

```
usersSchema.index({username: 1, email: 1});
```

Index performance impact

By default, mongoose always call the `ensureIndex` for each index sequentially and emit an 'index' event on the model when all the `ensureIndex` calls succeeded or when there was an error.

In MongoDB `ensureIndex` is deprecated since 3.0.0 version, now is an alias for `createIndex`.

Is recommended disable the behavior by setting the `autoIndex` option of your schema to `false`, or globally on the connection by setting the option `config.autoIndex` to `false`.

```
usersSchema.set('autoIndex', false);
```

Section 24.7: find data in mongodb using promises

Setup

First, install the necessary packages with:

```
npm install express cors mongoose
```

Code

Then, add dependencies to `server.js`, create the database schema and the name of the collection, create an Express.js server, and connect to MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});

app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

```
app.use(function(req, res, next) {
  res.status(404).send('抱歉，找不到该页面！');
});
```

现在添加我们将用来查询数据的Express.js路由：

```
app.get('/find/:query', cors(), function(req, res, next) {
  var query = req.params.query;

  Model.find({
    'request': query
  })
  .exec() //记得加exec, 查询有.then属性但不是Promise
  .then(function(result) {
    if (result) {
      res.json(result)
    } else {
      next() //传递给404处理程序
    }
  })
  .catch(next) //传递给错误处理器
})
```

假设以下文档存在于模型中的集合中：

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript 很棒",
  "time" : 1468710551
}

{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript 很棒",
  "time" : 1468710555
}

{
  "_id" : ObjectId("578abaea0522ad414b8eeb55c"),
  "request" : "JavaScript 很棒",
  "time" : 1468710560
}
```

目标是查找并显示所有在"request"

为此，启动 MongoDB 并使用 node server.js 运行 server.js：

用法

要使用此功能查找数据，请在浏览器中访问以下URL：

<http://localhost:8080/find/<query>>

其中 <query> 是搜索查询。

示例：

<http://localhost:8080/find/JavaScript%20is%20Awesome>

```
app.use(function(req, res, next) {
  res.status(404).send('Sorry can't find that!');
});
```

Now add Express.js routes that we will use to query the data:

```
app.get('/find/:query', cors(), function(req, res, next) {
  var query = req.params.query;

  Model.find({
    'request': query
  })
  .exec() //remember to add exec, queries have a .then attribute but aren't promises
  .then(function(result) {
    if (result) {
      res.json(result)
    } else {
      next() //pass to 404 handler
    }
  })
  .catch(next) //pass to error handler
})
```

Assume that the following documents are in the collection in the model:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}

{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}

{
  "_id" : ObjectId("578abaea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}
```

And the goal is to find and display all the documents containing "JavaScript is Awesome" under the "request" key.

For this, start MongoDB and run `server.js` with `node server.js`:

Usage

To use this to find data, go to the following URL in a browser:

<http://localhost:8080/find/<query>>

Where <query> is the search query.

Example:

<http://localhost:8080/find/JavaScript%20is%20Awesome>

输出：

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

belindoc.com

Output:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

第25章：async.js

第25.1节：并行：多任务处理

`async.parallel(tasks, afterTasksCallback)` 将并行执行一组任务，并等待所有任务结束（通过调用callback函数报告）。

当任务完成后，`async` 会调用主回调函数，传入所有错误和所有任务结果。

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, '长时间函数结果');
  }, 1000);
}

async.parallel([
  shortTimeFunction,
  mediumTimeFunction,
  longTimeFunction
],
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });

```

结果：["短时间函数结果", "中等时间函数结果", "长时间函数结果"]。

使用对象调用`async.parallel()`

你可以用一个对象替换`tasks`数组参数。在这种情况下，`results`也将是一个对象，其键与`tasks`相同。

这对于计算多个任务并轻松找到每个结果非常有用。

```
异步。并行({
  短: shortTimeFunction,
  中: mediumTimeFunction,
  长: longTimeFunction
},
  function(err, results) {
    if (err) {
      return console.error(err);
    }
}
```

Chapter 25: async.js

Section 25.1: Parallel : multi-tasking

`async.parallel(tasks, afterTasksCallback)` will execute a set of tasks in parallel and **wait the end of all tasks** (reported by the call of `callback` function).

When tasks are finished, `async` call the main callback with all errors and all results of tasks.

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}

async.parallel([
  shortTimeFunction,
  mediumTimeFunction,
  longTimeFunction
],
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });

```

Result : ["resultOfShortTime", "resultOfMediumTime", "resultOfLongTime"].

Call `async.parallel()` with an object

You can replace the `tasks` array parameter by an object. In this case, `results` will be also an object **with the same keys than tasks**.

It's very useful to compute some tasks and find easily each result.

```
async.parallel({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
  function(err, results) {
    if (err) {
      return console.error(err);
    }
})
```

```
console.log(results);
});
```

结果 : {短时间: "resultOfShortTime", 中等时间: "resultOfMediumTime", 长时间: "resultOfLongTime"}。

解析多个值

每个并行函数都会传入一个回调函数。该回调函数可以作为第一个参数返回错误，或者随后返回成功值。如果回调函数传入多个成功值，这些结果将作为数组返回。

```
async.parallel({
  短时间: function shortTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
    }, 200);
  },
  中等时间: function mediumTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
    }, 500);
  },
  function(err, results) {
    if (err) {
      return console.error(err);
    }
  }
});

console.log(results);
});
```

结果 :

```
{
  short: [ "resultOfShortTime1", "resultOfShortTime2" ],
  medium: [ "resultOfMediumTime1", "resultOfMediumTime2" ]
}
```

第25.2节 : async.each (高效处理数据数组)

当我们想要处理数据数组时，最好使用async.each。当我们想对所有数据执行某些操作并在所有操作完成后获得最终回调时，这个方法非常有用。它是以并行方式处理的。

```
function createUser(userName, callback)
{
  //在数据库中创建用户
  callback(null)//或根据创建情况返回错误
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {

  // 对每个用户执行操作。
  console.log('正在创建用户 '+eachUserName);
  // 必须返回回调。否则即使我们忘记返回回调，也不会得到最终回调
  callback();
});
```

```
console.log(results);
});
```

Result : {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}。

Resolving multiple values

Each parallel function is passed a callback. This callback can either return an error as the first argument or success values after that. If a callback is passed several success values, these results are returned as an array.

```
async.parallel({
  short: function shortTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
    }, 200);
  },
  medium: function mediumTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
    }, 500);
  },
  function(err, results) {
    if (err) {
      return console.error(err);
    }
  }
});

console.log(results);
});
```

Result :

```
{
  short: [ "resultOfShortTime1", "resultOfShortTime2" ],
  medium: [ "resultOfMediumTime1", "resultOfMediumTime2" ]
}
```

Section 25.2: async.each(To handle array of data efficiently)

When we want to handle array of data, its better to use **async.each**. When we want to perform something with all data & want to get the final callback once everything is done, then this method will be useful. This is handled in parallel way.

```
function createUser(userName, callback)
{
  //create user in db
  callback(null)//or error based on creation
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {

  // Perform operation on each user.
  console.log('Creating user '+eachUserName);
  //Returning callback is must. Else it won't get the final callback, even if we miss to return one
  callback();
});
```

```

createUser(eachUserName, callback);

}, function(err) {
  //如果任何用户创建失败，可能会抛出错误。
  if( err ) {
    //其中一次迭代产生了错误。
    //所有处理现在将停止。
    console.log('无法创建用户');
  } else {
    console.log('所有用户创建成功');
  }
});

```

要一次处理一个，可以使用`async.eachSeries`

第25.3节：系列：独立单任务处理

`async.series(tasks, afterTasksCallback)`将执行一组任务。每个任务依次执行。如果某个任务失败，`async`会立即停止执行并跳转到主回调。

当任务成功完成时，`async`会调用“主”回调，传入所有错误和所有任务结果。

```

function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, '长时间函数结果');
  }, 1000);
}

async.series([
  mediumTimeFunction,
  shortTimeFunction,
  longTimeFunction
],
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});

```

结果 : ["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"]。

调用`async.series()`并传入一个对象

你可以用一个对象替换`tasks`数组参数。在这种情况下，`results`也将是一个对象，其键与`tasks`相同。

```

createUser(eachUserName, callback);

}, function(err) {
  //If any of the user creation failed may throw error.
  if( err ) {
    // One of the iterations produced an error.
    // All processing will now stop.
    console.log('unable to create user');
  } else {
    console.log('All user created successfully');
  }
});

```

To do one at a time can use `async.eachSeries`

Section 25.3: Series : independent mono-tasking

`async.series(tasks, afterTasksCallback)` will execute a set of tasks. Each task are executed **after another**. If a task fails, `async` stops immediately the execution and jump into the main callback.

When tasks are finished successfully, `async` call the "master" callback with all errors and all results of tasks.

```

function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}

async.series([
  mediumTimeFunction,
  shortTimeFunction,
  longTimeFunction
],
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});

```

Result : ["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"].

Call `async.series()` with an object

You can replace the `tasks` array parameter by an object. In this case, `results` will be also an object **with the same keys than tasks**.

这对于计算多个任务并轻松找到每个结果非常有用。

```
async.series({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

结果 : {短时间: "resultOfShortTime", 中等时间: "resultOfMediumTime", 长时间: "resultOfLongTime"}。

第25.4节：瀑布流：依赖的单任务处理

`async.waterfall(tasks, afterTasksCallback)`将执行一组任务。每个任务依次执行，一个任务的结果会传递给下一个任务。与`async.series()`类似，如果某个任务失败，`async`会停止执行并立即调用主回调函数。

当任务成功完成时，`async`会调用“主”回调，传入所有错误和所有任务结果。

```
function getUserRequest(callback) {
  // 我们用超时模拟请求
  setTimeout(function() {
    var userResult = {
      name : 'Aamu'
    };

    callback(null, userResult);
    }, 500);
}

function getUserFriendsRequest(user, callback) {
  // 另一个使用超时模拟的请求
  setTimeout(function() {
    var friendsResult = [];

    if (user.name === "Aamu"){
      friendsResult = [
        {
          name : '爱丽丝'
        },
        {
          name: '鲍勃'
        }
      ];
    }

    callback(null, friendsResult);
    }, 500);
}

async.waterfall([
  getUserRequest,
  getUserFriendsRequest
],
function(err, results) {
  if (err) {
    return console.error(err);
  }
```

It's very useful to compute some tasks and find easily each result.

```
async.series({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

Result : {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}。

Section 25.4: Waterfall : dependent mono-tasking

`async.waterfall(tasks, afterTasksCallback)` will execute a set of tasks. Each task are executed **after another, and the result of a task is passed to the next task**. As `async.series()`, if a task fails, `async` stop the execution and call immediately the main callback.

When tasks are finished successfully, `async` call the "master" callback with all errors and all results of tasks.

```
function getUserRequest(callback) {
  // We simulate the request with a timeout
  setTimeout(function() {
    var userResult = {
      name : 'Aamu'
    };

    callback(null, userResult);
    }, 500);
}

function getUserFriendsRequest(user, callback) {
  // Another request simulate with a timeout
  setTimeout(function() {
    var friendsResult = [];

    if (user.name === "Aamu"){
      friendsResult = [
        {
          name : 'Alice'
        },
        {
          name: 'Bob'
        }
      ];
    }

    callback(null, friendsResult);
    }, 500);
}

async.waterfall([
  getUserRequest,
  getUserFriendsRequest
],
function(err, results) {
  if (err) {
    return console.error(err);
  }
```

```

    }

console.log(JSON.stringify(results));
});

```

结果：results 包含 waterfall 最后一个函数的第二个回调参数，在该情况下是 friendsResult。

第 25.5 节：async.times（更好地处理 for 循环）

在 node.js 中循环执行函数，短循环使用 for 循环是可以的。但循环较长时，使用for 循环会增加处理时间，可能导致 node 进程挂起。在这种情况下，可以使用：async.times

```

function recursiveAction(n, callback)
{
    //重复执行任意操作
    callback(err, result);
}

async.times(5, function(n, next) {
    recursiveAction(n, function(err, result) {
        next(err, result);
    });
}, function(err, results) {
    // 我们现在应该有5个结果
});

```

这称为并行调用。当我们想一次调用一个时，使用：async.timesSeries

第25.6节：async.series（按顺序处理事件）

在async.series中，所有函数按顺序执行，每个函数的合并输出传递给最终的 callback。例如

```

var async = require('async');
async.series([
    function (callback) {
        console.log('第一次执行..');
        callback(null, 'userPersonalData');
    },
    function (callback) {
        console.log('第二次执行.. ');
        callback(null, 'userDependentData');
    }
], 

function (err, result) {
    console.log(result);
});

```

输出：

第一次执行.. 第二次执行.. ['userPersonalData','userDependentData'] //结果

```

    }

    console.log(JSON.stringify(results));
});

```

Result: results contains the second callback parameter of the last function of the waterfall, which is friendsResult in that case.

Section 25.5: async.times(To handle for loop in better way)

To execute a function within a loop in node.js, it's fine to use a **for** loop for short loops. But the loop is long, using **for** loop will increase the time of processing which might cause the node process to hang. In such scenarios, you can use: **async.times**

```

function recursiveAction(n, callback)
{
    //do whatever want to do repeatedly
    callback(err, result);
}

async.times(5, function(n, next) {
    recursiveAction(n, function(err, result) {
        next(err, result);
    });
}, function(err, results) {
    // we should now have 5 result
});

```

This is called in parallel. When we want to call it one at a time, use: **async.timesSeries**

Section 25.6: async.series(To handle events one by one)

In **async.series**, all the functions are executed in series and the consolidated outputs of each function is passed to the final callback. e.g

```

var async = require('async');
async.series([
    function (callback) {
        console.log('First Execute.. ');
        callback(null, 'userPersonalData');
    },
    function (callback) {
        console.log('Second Execute.. ');
        callback(null, 'userDependentData');
    }
], 

function (err, result) {
    console.log(result);
});

```

Output:

First Execute.. Second Execute.. ['userPersonalData','userDependentData'] //result

第26章：文件上传

第26.1节：使用multer进行单文件上传

请记住

- 为上传创建文件夹（示例中为\uploads\u）。
- 安装 multer `npm i -S multer`

server.js :

```
var express = require("express");
var multer = require('multer');
var app = express();
var fs = require('fs');

app.get('/', function(req, res){
    res.sendFile(__dirname + "/index.html");
});

var storage = multer.diskStorage({
  destination: function (req, file, callback) {
    fs.mkdir('./uploads', function(err) {
      if(err) {
        console.log(err.stack)
      } else {
        callback(null, './uploads');
      }
    })
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

app.post('/api/file', function(req, res){
  var upload = multer({ storage : storage}).single('userFile');
  upload(req, res, function(err) {
    if(err) {
      return res.end("Error uploading file.");
    }
    res.end("文件已上传");
  });
});

app.listen(3000, function(){
  console.log("Working on port 3000");
});
```

index.html:

```
<form id="uploadForm" 
      enctype="multipart/form-data"
      action="/api/file"
      method="post">
<input type="file" name="userFile" />
<input type="submit" value="Upload File" name="submit">
```

Chapter 26: File upload

Section 26.1: Single File Upload using multer

Remember to

- create folder for upload (uploads in example).
- install multer `npm i -S multer`

server.js:

```
var express = require("express");
var multer = require('multer');
var app = express();
var fs = require('fs');

app.get('/', function(req, res){
    res.sendFile(__dirname + "/index.html");
});

var storage = multer.diskStorage({
  destination: function (req, file, callback) {
    fs.mkdir('./uploads', function(err) {
      if(err) {
        console.log(err.stack)
      } else {
        callback(null, './uploads');
      }
    })
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

app.post('/api/file', function(req, res){
  var upload = multer({ storage : storage}).single('userFile');
  upload(req, res, function(err) {
    if(err) {
      return res.end("Error uploading file.");
    }
    res.end("File is uploaded");
  });
});

app.listen(3000, function(){
  console.log("Working on port 3000");
});
```

index.html:

```
<form id="uploadForm" 
      enctype="multipart/form-data"
      action="/api/file"
      method="post">
<input type="file" name="userFile" />
<input type="submit" value="Upload File" name="submit">
```

```
</form>
```

注意：

要上传带扩展名的文件，可以使用 Node.js 内置库[path](#)

为此，只需在server.js文件中引入[path](#)：

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

并修改：

```
callback(null, file.fieldname + '-' + Date.now());
```

以下方式添加文件扩展名：

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
```

如何按扩展名过滤上传：

在此示例中，查看如何上传文件以仅允许某些扩展名。

例如，仅允许图片扩展名。只需添加到 var upload = multer({ storage : storage}).single('userFile'); fileFilter 条件

```
var upload = multer({
  storage: storage,
  fileFilter: function (req, file, callback) {
    var ext = path.extname(file.originalname);
    if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
      return callback(new Error('只允许图片上传'))
    }
  }
}).single('userFile');
```

现在您只能上传扩展名为png、jpg、gif或jpeg的图像文件

第26.2节：使用formidable模块

安装模块并阅读文档

```
npm i formidable@latest
```

8080端口服务器示例

```
var formidable = require('formidable'),
  http = require('http'),
  util = require('util');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // 解析文件上传
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      if (err)
        do-smth; // 处理错误
    });
  }
}).listen(8080);
```

```
</form>
```

Note:

To upload file with extension you can use Node.js [path](#) built-in library

For that just require path to server.js file:

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

and change:

```
callback(null, file.fieldname + '-' + Date.now());
```

adding a file extension in the following way:

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
```

How to filter upload by extension:

In this example, view how to upload files to allow only certain extensions.

For example only images extensions. Just add to `var upload = multer({ storage : storage}).single('userFile');` fileFilter condition

```
var upload = multer({
  storage: storage,
  fileFilter: function (req, file, callback) {
    var ext = path.extname(file.originalname);
    if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
      return callback(new Error('Only images are allowed'))
    }
  }
}).single('userFile');
```

Now you can upload only image files with png, jpg, gif or jpeg extensions

Section 26.2: Using formidable module

Install module and read [docs](#)

```
npm i formidable@latest
```

Example of server on 8080 port

```
var formidable = require('formidable'),
  http = require('http'),
  util = require('util');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      if (err)
        do-smth; // process error
    });
  }
}).listen(8080);
```

```

// 从临时位置复制文件
// var fs = require('fs');
// fs.rename(file.path, <targetPath>, function (err) { ... });

// 向客户端发送结果
res.writeHead(200, {'content-type': 'text/plain'});
    res.write('接收到  
上传：');
res.end(util.inspect({fields: fields, files: files}));
};

return;
}

// 显示文件上传表单
res.writeHead(200, {'content-type' : 'text/html'});
res.end(
'<form action="/upload" enctype="multipart/form-data" method="post">'+
'<input type="text" name="title"><br>'+
'<input type="file" name="upload" multiple="multiple"><br>'+
'<input type="submit" value="Upload">'+
'</form>';
);
}).listen(8080);

```

belindoc.com

```

// Copy file from temporary place
// var fs = require('fs');
// fs.rename(file.path, <targetPath>, function (err) { ... });

// Send result on client
res.writeHead(200, {'content-type': 'text/plain'});
res.write('received upload:\n\n');
res.end(util.inspect({fields: fields, files: files}));
);

return;
}

// show a file upload form
res.writeHead(200, {'content-type' : 'text/html'});
res.end(
'<form action="/upload" enctype="multipart/form-data" method="post">'+
'<input type="text" name="title"><br>'+
'<input type="file" name="upload" multiple="multiple"><br>'+
'<input type="submit" value="Upload">'+
'</form>';
));
}).listen(8080);

```

第27章：Socket.io通信

第27.1节：“你好，世界！”通过socket消息

安装节点模块

```
npm install express  
npm install socket.io
```

Node.js服务器

```
const express = require('express');  
const app = express();  
const server = app.listen(3000,console.log("Socket.io Hello World server started!"));  
const io = require('socket.io')(server);  
  
io.on('connection', (socket) => {  
    //console.log("客户端已连接！");  
    socket.on('客户端到服务器的消息', (msg) => {  
        console.log(msg);  
    })  
    socket.emit('服务器到客户端的消息', '你好，世界！');  
});
```

浏览器客户端

```
<!DOCTYPE html>  
<html lang="zh">  
  <head>  
    <meta charset="UTF-8">  
    <title>使用 Socket.io 的你好，世界</title>  
  </head>  
  <body>  
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>  
    <script>  
      var socket = io("http://localhost:3000");  
      socket.on("服务器到客户端的消息", function(msg) {  
        document.getElementById('消息').innerHTML = msg;  
      });  
      socket.emit('客户端到服务器的消息', '你好，世界！');  
    </script>  
    <p>Socket.io Hello World 客户端已启动!</p>  
    <p id="message"></p>  
  </body></html>
```

Chapter 27: Socket.io communication

Section 27.1: "Hello world!" with socket messages

Install node modules

```
npm install express  
npm install socket.io
```

Node.js server

```
const express = require('express');  
const app = express();  
const server = app.listen(3000,console.log("Socket.io Hello World server started!"));  
const io = require('socket.io')(server);  
  
io.on('connection', (socket) => {  
    //console.log("Client connected!");  
    socket.on('message-from-client-to-server', (msg) => {  
        console.log(msg);  
    })  
    socket.emit('message-from-server-to-client', 'Hello World!');  
});
```

Browser client

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <title>Hello World with Socket.io</title>  
  </head>  
  <body>  
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>  
    <script>  
      var socket = io("http://localhost:3000");  
      socket.on("message-from-server-to-client", function(msg) {  
        document.getElementById('message').innerHTML = msg;  
      });  
      socket.emit('message-from-client-to-server', 'Hello World!');  
    </script>  
    <p>Socket.io Hello World client started!</p>  
    <p id="message"></p>  
  </body>  
</html>
```

第28章：Mongodb集成

参数	详细
文件	表示文档的JavaScript对象
文件	一组文件
查询	定义搜索查询的对象
过滤器	定义搜索查询的对象
回调函数	操作完成时调用的函数
选项	(可选) 可选设置 (默认值: null)
w	(可选) 写入关注级别
wtimeout	(可选) 写入关注超时时间。 (默认值: null)
j	(可选) 指定日志写入关注 (默认值: false)
upsert	(可选) 更新操作 (默认 : false)
multi	(可选) 更新一个/所有文档 (默认值: false)
serializeFunctions	(可选) 对任意对象序列化函数 (默认值: false)
forceServerObjectId	(可选) 强制服务器分配 _id 值而非驱动程序 (默认值: false)
bypassDocumentValidation	(可选) 允许驱动程序绕过 MongoDB 3.2 及以上版本的模式验证 (默认值: false)

第28.1节：简单连接

```
MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) { if(error) return  
  console.log(error); const collection = database.collection('collectionName'); collection.insert({key: 'value'},  
  function(error, result) { console.log(error, result); });});
```

第28.2节：使用Promise的简单连接

```
const MongoDB = require('mongodb');  
  
MongoDB.connect('mongodb://localhost:27017/databaseName')  
  .then(function(database) {  
    const collection = database.collection('collectionName');  
    return collection.insert({key: 'value'});  
  })  
  .then(function(result) {  
    console.log(result);  
  })  
  .catch(function(error) {  
    console.error(error);  
  });
```

第28.3节：连接到MongoDB

连接到MongoDB，打印“Connected!”并关闭连接。

```
const MongoClient = require('mongodb').MongoClient;  
  
var url = 'mongodb://localhost:27017/test';  
  
MongoClient.connect(url, function(err, db) { // MongoClient方法 'connect'  
  if (err) throw new Error(err);  
  console.log("Connected!");  
  db.close(); // 完成后别忘了关闭连接  
});
```

Chapter 28: Mongodb integration

Parameter	Details
document	A javascript object representing a document
documents	An array of documents
query	An object defining a search query
filter	An object defining a search query
callback	Function to be called when the operation is done
options	(optional) Optional settings (default: null)
w	(optional) The write concern
wtimeout	(optional) The write concern timeout. (default: null)
j	(optional) Specify a journal write concern (default: false)
upsert	(optional) Update operation (default: false)
multi	(optional) Update one/all documents (default: false)
serializeFunctions	(optional) Serialize functions on any object (default: false)
forceServerObjectId	(optional) Force server to assign _id values instead of driver (default: false)
bypassDocumentValidation	(optional) Allow driver to bypass schema validation in MongoDB 3.2 or higher (default: false)

Section 28.1: Simple connect

```
MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) { if(error) return  
  console.log(error); const collection = database.collection('collectionName'); collection.insert({key: 'value'},  
  function(error, result) { console.log(error, result);});});
```

Section 28.2: Simple connect, using promises

```
const MongoDB = require('mongodb');  
  
MongoDB.connect('mongodb://localhost:27017/databaseName')  
  .then(function(database) {  
    const collection = database.collection('collectionName');  
    return collection.insert({key: 'value'});  
  })  
  .then(function(result) {  
    console.log(result);  
  })  
  .catch(function(error) {  
    console.error(error);  
  });
```

Section 28.3: Connect to MongoDB

Connect to MongoDB, print 'Connected!' and close the connection.

```
const MongoClient = require('mongodb').MongoClient;  
  
var url = 'mongodb://localhost:27017/test';  
  
MongoClient.connect(url, function(err, db) { // MongoClient method 'connect'  
  if (err) throw new Error(err);  
  console.log("Connected!");  
  db.close(); // Don't forget to close the connection when you are done  
});
```

MongoClient 方法 Connect()

```
MongoClient.connect(url, options, callback)
```

参数类型	描述
url	字符串 指定服务器 IP/主机名、端口和数据库的字符串
选项	对象 (可选) 可选设置 (默认值: null)
callback函数	连接尝试完成后调用的函数

该callback函数接受两个参数

- err : 错误 - 如果发生错误, err 参数将被定义
- db : 对象 - MongoDB 实例

第28.4节：插入文档

插入一个名为 'myFirstDocument' 的文档，并设置2个属性，greetings 和 farewell

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // 插入方法 'insertOne'
    "myFirstDocument": {
      "greetings": "Hello",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("已向 myCollection 集合插入文档！");
    db.close(); // 完成后别忘了关闭连接
  });
});
```

采集方法 insertOne()

```
db.collection(collection).insertOne(document, options, callback)
```

参数	类型	描述
collection	字符串	指定集合的字符串
document	对象	要插入集合的文档
选项	对象 (可选) 可选设置 (默认值: null)	
回调函数	函数	插入操作完成后调用的函数

该callback函数接受两个参数

- err : 错误 - 如果发生错误, err 参数将被定义
- result : 对象 - 包含插入操作详细信息的对象

MongoClient method Connect()

```
MongoClient.connect(url, options, callback)
```

Argument	Type	Description
url	string	A string specifying the server ip/hostame, port and database
options	object	(optional) Optional settings (default: null)
callback	Function	Function to be called when the connection attempt is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- db : object - The MongoDB instance

Section 28.4: Insert a document

Insert a document called 'myFirstDocument' and set 2 properties, greetings and farewell

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // Insert method 'insertOne'
    "myFirstDocument": {
      "greetings": "Hello",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("Inserted a document into the myCollection collection!");
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

Collection method insertOne()

```
db.collection(collection).insertOne(document, options, callback)
```

Argument	Type	Description
collection	string	A string specifying the collection
document	object	The document to be inserted into the collection
options	object	(optional) Optional settings (default: null)
callback	Function	Function to be called when the insert operation is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- result : object - An object containing details about the insert operation

第28.5节：读取集合

获取集合 'myCollection' 中的所有文档并打印到控制台。

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // 读取方法 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // 打印所有文档
    } else {
      db.close(); // 完成后别忘了关闭连接
    }
  });
});
```

集合方法 find()

```
db.collection(collection).find()
```

参数类型	描述
collection	字符串，指定集合名称

第28.6节：更新文档

查找属性为 { greetings: 'Hello' } 的文档并将其更改为 { greetings: 'Whut?' }

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('我的集合').updateOne({ // 更新方法 'updateOne'
    greetings: "Hello"
  }, {
    $set: { greetings: "Whut?" }
  }, function (err, result) {
    if (err) throw new Error(err);
  });
  db.close(); // 完成后别忘了关闭连接
});
```

集合方法 updateOne()

```
db.collection(collection).updateOne(filter, update, options. callback)
```

参数类型	描述
过滤器	对象 指定选择条件
更新	对象 指定要应用的修改

Section 28.5: Read a collection

Get all documents in the collection 'myCollection' and print them to the console.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // Read method 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // Print all documents
    } else {
      db.close(); // Don't forget to close the connection when you are done
    }
  });
});
```

Collection method find()

```
db.collection(collection).find()
```

Argument Type	Description
collection	string A string specifying the collection

Section 28.6: Update a document

Find a document with the property { greetings: 'Hello' } and change it to { greetings: 'Whut?' }

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').updateOne({ // Update method 'updateOne'
    greetings: "Hello"
  }, {
    $set: { greetings: "Whut?" }
  }, function (err, result) {
    if (err) throw new Error(err);
  });
  db.close(); // Don't forget to close the connection when you are done
});
```

Collection method updateOne()

```
db.collection(collection).updateOne(filter, update, options. callback)
```

Parameter	Type	Description
filter	object	Specifies the selection criteria
update	object	Specifies the modifications to apply

选项 对象 (可选) 可选设置 (默认值: null)
回调函数 函数 操作完成后调用的函数

该callback函数接受两个参数

- err : 错误 - 如果发生错误, err 参数将被定义
- db : 对象 - MongoDB 实例

第28.7节：删除文档

删除具有属性{ greetings: 'Whut?' }的文档

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteOne("// 删除方法 'deleteOne'
    { greetings: 'Whut?' },
    function (err, result) {
      if (err) throw new Error(err);
    db.close(); // 完成后别忘了关闭连接
  });
});
```

集合方法deleteOne()

```
db.collection(collection).deleteOne(filter, options, callback)
```

参数类型	描述
过滤器	对象 指定选择条件的文档
选项	对象 (可选) 可选设置 (默认值: null)
回调函数	函数 操作完成后调用的函数

该callback函数接受两个参数

- err : 错误 - 如果发生错误, err 参数将被定义
- db : 对象 - MongoDB 实例

第28.8节：删除多个文档

删除所有属性 'farewell' 设置为 'okay' 的文档。

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteMany("// MongoDB 删除方法 'deleteMany'
    { farewell: 'okay' }, // 删除所有属性为 'farewell: okay' 的文档
    function (err, result) {
      if (err) throw new Error(err);
    db.close(); // 完成后别忘了关闭连接
  });
});
```

options object (optional) Optional settings (default: null)
callback Function Function to be called when the operation is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- db : object - The MongoDB instance

Section 28.7: Delete a document

Delete a document with the property { greetings: 'Whut?' }

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteOne("// Delete method 'deleteOne'
    { greetings: 'Whut?' },
    function (err, result) {
      if (err) throw new Error(err);
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

Collection method deleteOne()

```
db.collection(collection).deleteOne(filter, options, callback)
```

Parameter	Type	Description
filter	object	A document specifying the selection criteria
options	object	(optional) Optional settings (default: null)
callback	Function	Function to be called when the operation is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- db : object - The MongoDB instance

Section 28.8: Delete multiple documents

Delete ALL documents with a 'farewell' property set to 'okay'.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteMany("// MongoDB delete method 'deleteMany'
    { farewell: 'okay' }, // Delete ALL documents with the property 'farewell: okay'
    function (err, result) {
      if (err) throw new Error(err);
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

```
});  
});
```

集合方法 deleteMany()

```
db.collection(collection).deleteMany(filter, options, callback)
```

参数	类型	描述
过滤器	document	一个指定选择条件的文档
选项	对象	(可选) 可选设置 (默认值: null)
回调函数	函数	操作完成时调用的函数

该callback函数接受两个参数

- err : 错误 - 如果发生错误, err 参数将被定义
- db : 对象 - MongoDB 实例

```
});  
});
```

Collection method deleteMany()

```
db.collection(collection).deleteMany(filter, options, callback)
```

Parameter	Type	Description
filter	document	A document specifying the selection criteria
options	object	(optional) Optional settings (default: null)
callback	function	Function to be called when the operation is done

The callback function takes two arguments

- err : Error - If an error occurs the err argument will be defined
- db : object - The MongoDB instance

第29章：在

Node.js中处理POST请求

第29.1节：仅处理POST

请求的Node.js示例服务器

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
  request.on('data', chunk => {
    buffer += chunk;
  });
  request.on('end', () => {
    const responseString = `接收到的字符串 ${buffer}`;
    console.log(`响应内容: ${responseString}`);
    response.writeHead(200, "Content-Type: text/plain");
    response.end(responseString);
  });
}).listen(PORT, () => {
  console.log(`监听端口 ${PORT}`);
});
```

Chapter 29: Handling POST request in Node.js

Section 29.1: Sample node.js server that just handles POST requests

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
  request.on('data', chunk => {
    buffer += chunk;
  });
  request.on('end', () => {
    const responseString = `Received string ${buffer}`;
    console.log(`Responding with: ${responseString}`);
    response.writeHead(200, "Content-Type: text/plain");
    response.end(responseString);
  });
}).listen(PORT, () => {
  console.log(`Listening on ${PORT}`);
});
```

第30章：基于简单REST的CRUD API

第30.1节：Express 3+中的REST CRUD API

```
var express = require("express"),
bodyParser = require("body-parser"),
server = express();

//用于解析请求体的body解析器
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

//内存中用于临时存储`item`
var itemStore = [];

//获取所有项目
server.get('/item', function (req, res) {
  res.json(itemStore);
});

//获取指定id的项目
server.get('/item/:id', function (req, res) {
  res.json(itemStore[req.params.id]);
});

//新增项目 (POST)
server.post('/item', function (req, res) {
  itemStore.push(req.body);
  res.json(req.body);
});

//用指定id的项目替换编辑后的项目 (PUT)
server.put('/item/:id', function (req, res) {
  itemStore[req.params.id] = req.body
  res.json(req.body);
});

//删除指定id的项目
server.delete('/item/:id', function (req, res) {
  itemStore.splice(req.params.id, 1)
  res.json(req.body);
});

//启动服务器
server.listen(3000, function () {
  console.log("Server running");
})
```

Chapter 30: Simple REST based CRUD API

Section 30.1: REST API for CRUD in Express 3+

```
var express = require("express"),
bodyParser = require("body-parser"),
server = express();

//body parser for parsing request body
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

//temporary store for `item` in memory
var itemStore = [];

//GET all items
server.get('/item', function (req, res) {
  res.json(itemStore);
});

//GET the item with specified id
server.get('/item/:id', function (req, res) {
  res.json(itemStore[req.params.id]);
});

//POST new item
server.post('/item', function (req, res) {
  itemStore.push(req.body);
  res.json(req.body);
});

//PUT edited item in-place of item with specified id
server.put('/item/:id', function (req, res) {
  itemStore[req.params.id] = req.body
  res.json(req.body);
});

//DELETE item with specified id
server.delete('/item/:id', function (req, res) {
  itemStore.splice(req.params.id, 1)
  res.json(req.body);
});

//START SERVER
server.listen(3000, function () {
  console.log("Server running");
})
```

第31章：模板框架

第31.1节：Nunjucks

服务器端引擎，支持块继承、自动转义、宏、异步控制等功能。深受jinja2启发，与Twig (php) 非常相似。

文档 - <http://mozilla.github.io/nunjucks/>

安装 - npm i nunjucks

下面是与Express的基本用法。

app.js

```
var express = require('express');
var nunjucks = require('nunjucks');

var app = express();
app.use(express.static('/public'));

// 应用nunjucks并添加自定义过滤器和函数（例如）。
var env = nunjucks.configure(['views'], { // 设置模板文件夹
    autoescape: true,
    express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
    console.log('myFilter', obj, arg1, arg2);
    // 对 obj 进行某些操作
    return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
    console.log('myFunc', obj, arg1);
    // 对 obj 进行某些操作
    return obj;
});

app.get('/', function(req, res){
    res.render('index.html', {title: '主页'});
});

app.get('/foo', function(req, res){
    res.locals.smthVar = '这是斯巴达！';
    res.render('foo.html', {title: 'Foo 页面'});
});

app.listen(3000, function() {
    console.log('示例应用正在监听 3000 端口...');
});
```

/views/index.html

```
<html>
<head>
    <title>Nunjucks 示例</title>

<body>
    {% block content %}
    {{title}}
```

Chapter 31: Template frameworks

Section 31.1: Nunjucks

Server-side engine with block inheritance, autoescaping, macros, asynchronous control, and more. Heavily inspired by jinja2, very similar to Twig (php).

Docs - <http://mozilla.github.io/nunjucks/>

Install - npm i nunjucks

Basic usage with [Express](#) below.

app.js

```
var express = require('express');
var nunjucks = require('nunjucks');

var app = express();
app.use(express.static('/public'));

// Apply nunjucks and add custom filter and function (for example).
var env = nunjucks.configure(['views'], { // set folders with templates
    autoescape: true,
    express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
    console.log('myFilter', obj, arg1, arg2);
    // Do smth with obj
    return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
    console.log('myFunc', obj, arg1);
    // Do smth with obj
    return obj;
});

app.get('/', function(req, res){
    res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
    res.locals.smthVar = 'This is Sparta!';
    res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000...');
});
```

/views/index.html

```
<html>
<head>
    <title>Nunjucks example</title>
</head>
<body>
    {% block content %}
    {{title}}
```

```
{% endblock %}  
</body>  
</html>
```

/views/foo.html

```
{% extends "index.html" %}  
  
{# 这是注释 #}  
{% block content %}  
  <h1>{{title}}</h1>  
  {# 应用自定义函数并接着构建-输入和自定义过滤器 #}  
  {{ myFunc(smthVar) | lower | myFilter(5, 'abc') }}  
{% endblock %}
```

```
{% endblock %}  
</body>  
</html>
```

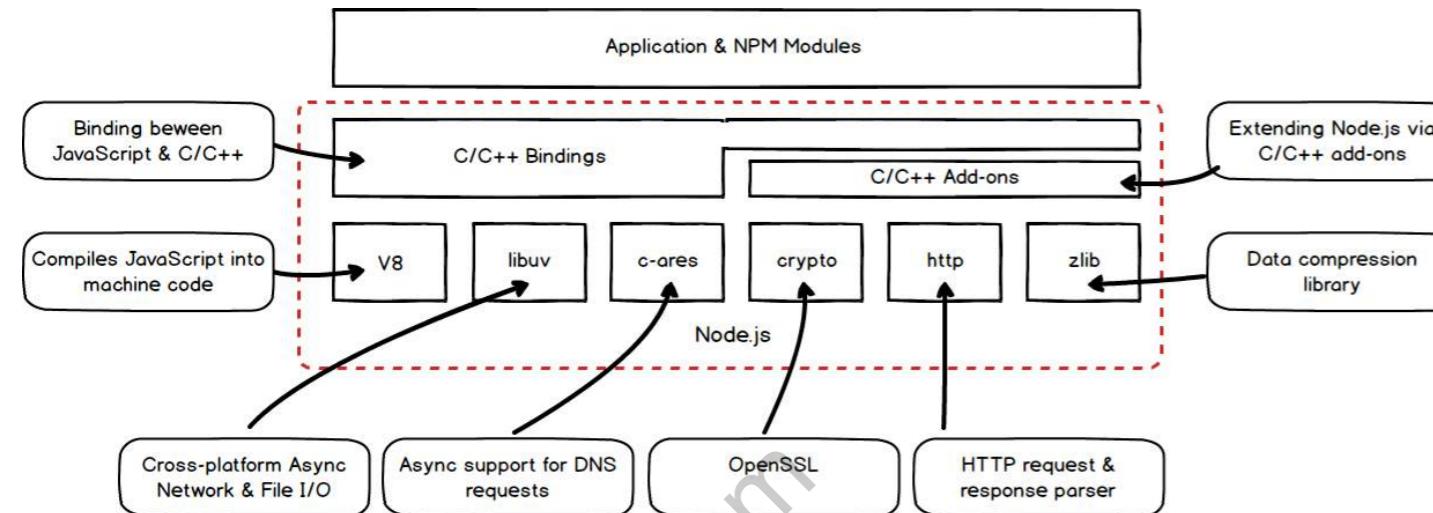
/views/foo.html

```
{% extends "index.html" %}  
  
{# This is comment #}  
{% block content %}  
  <h1>{{title}}</h1>  
  {# apply custom function and next build-in and custom filters #}  
  {{ myFunc(smthVar) | lower | myFilter(5, 'abc') }}  
{% endblock %}
```

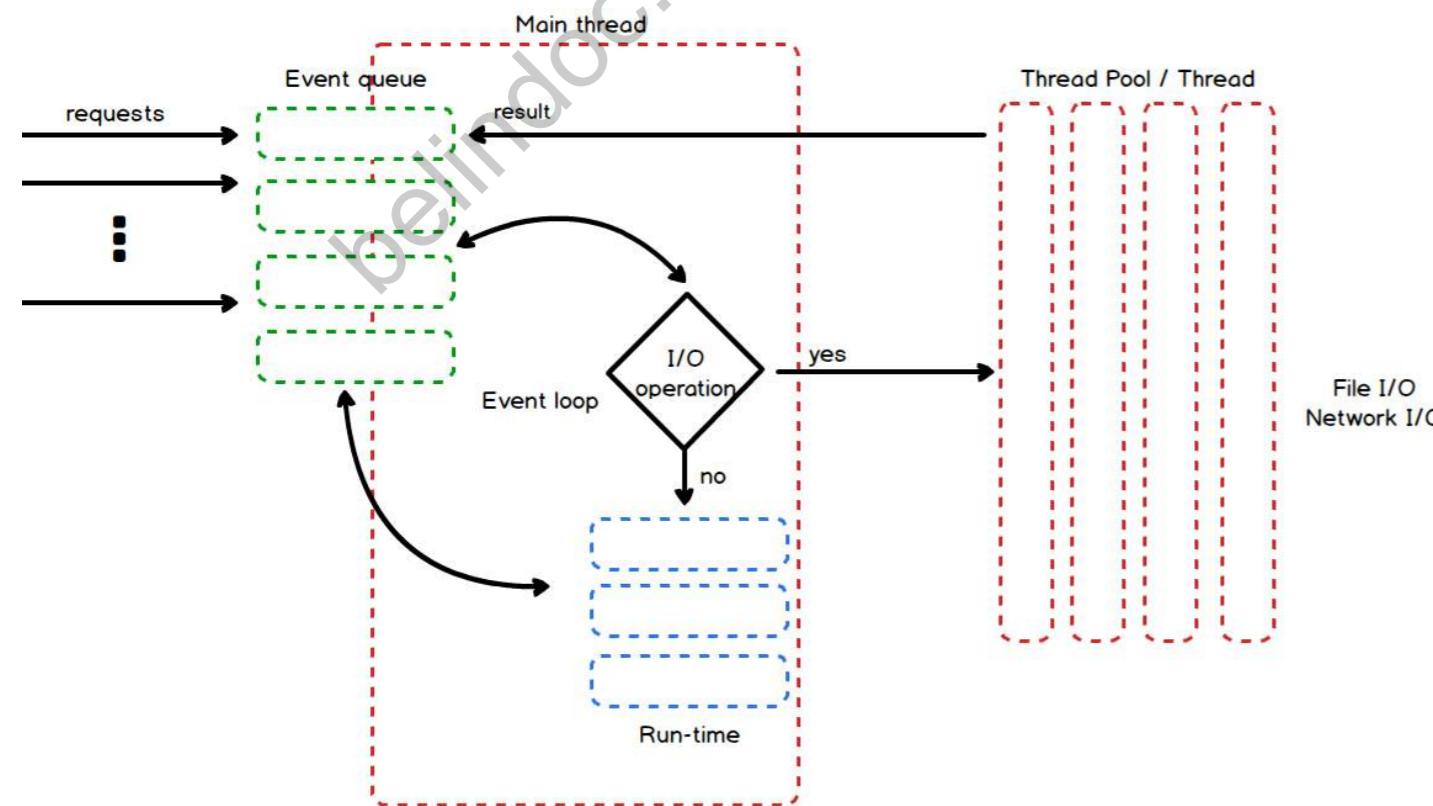
belindoc.com

第32章：Node.js架构与内部工作原理

第32.1节：Node.js - 内部机制

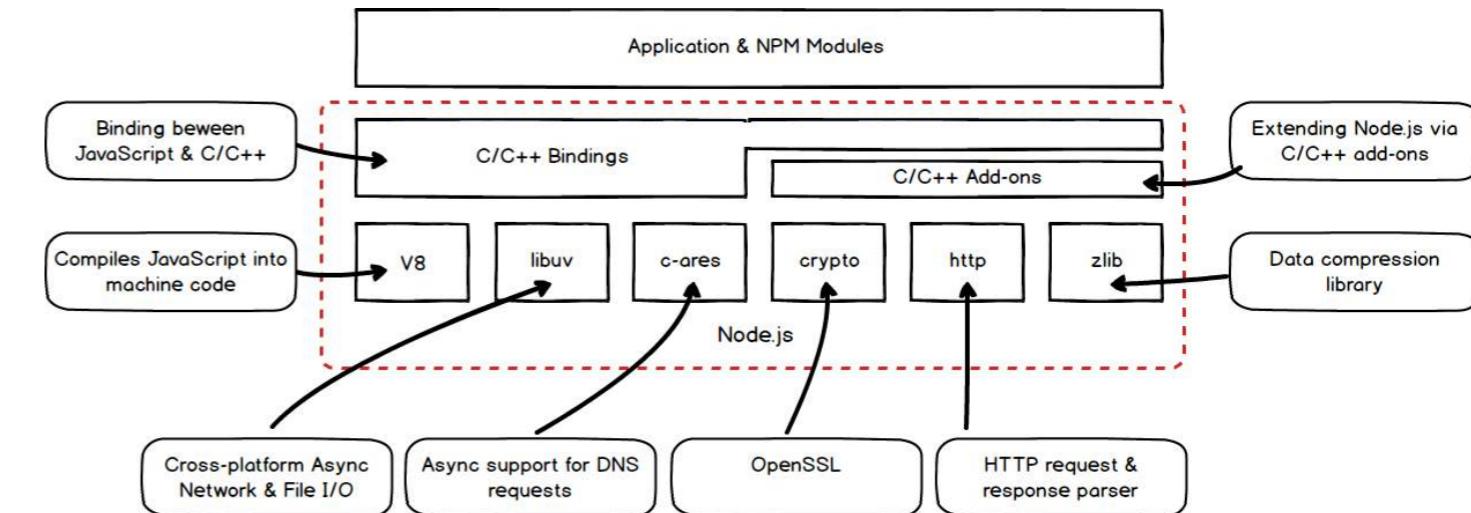


第32.2节：Node.js - 运行中

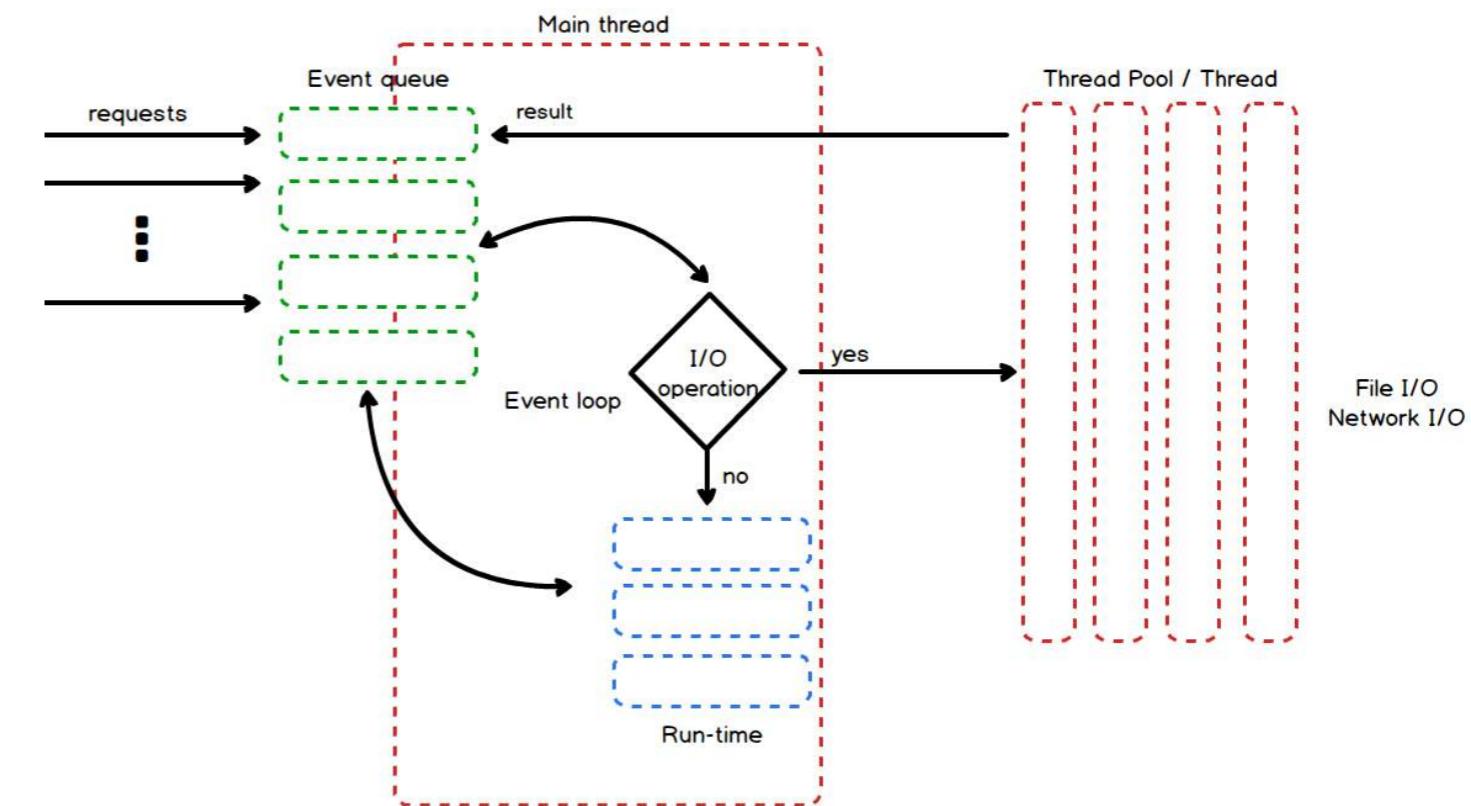


Chapter 32: Node.js Architecture & Inner Workings

Section 32.1: Node.js - under the hood



Section 32.2: Node.js - in motion



第33章：调试Node.js应用程序

第33.1节：核心Node.js调试器和Node Inspector

使用核心调试器

Node.js提供了一个内置的非图形化调试工具。要启动内置调试器，请使用以下命令启动应用程序：

节点调试 filename.js

考虑以下包含在debugDemo.js中的简单Node.js应用程序

```
'use strict';

函数 addTwoNumber(a, b){
// 函数返回两个数字的和
debugger
    返回 a + b;
}

变量 result = addTwoNumber(5, 9);
控制台.log(result);
```

关键字 debugger 会在代码的该点停止调试器。

命令参考

1. 单步执行

```
cont, c - 继续执行
next, n - 执行下一步
step, s - 进入执行
out, o - 跳出执行
```

2. 断点

```
setBreakpoint(), sb() - 在当前行设置断点
setBreakpoint(line), sb(line) - 在指定行设置断点
```

调试上述代码，请运行以下命令

```
node debug debugDemo.js
```

运行上述命令后，您将看到以下输出。要退出调试器界面，请输入
process.exit()

Chapter 33: Debugging Node.js application

Section 33.1: Core node.js debugger and node inspector

Using core debugger

Node.js provides a build in non graphical debugging utility. To start the build in the debugger, start the application with this command:

node debug filename.js

Consider the following simple Node.js application contained in the debugDemo.js

```
'use strict';

function addTwoNumber(a, b){
// function returns the sum of the two numbers
debugger
    return a + b;
}

var result = addTwoNumber(5, 9);
console.log(result);
```

The keyword debugger will stop the debugger at that point in the code.

Command reference

1. Stepping

```
cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
```

2. Breakpoints

```
setBreakpoint(), sb() - Set breakpoint on current line
setBreakpoint(line), sb(line) - Set breakpoint on specific line
```

To Debug the above code run the following command

node debug debugDemo.js

Once the above commands runs you will see the following output. To exit from the debugger interface, type
process.exit()

```

ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
 1 // A Demo Code Showing the basic capabilities of the nodejs debugging module
 2
> 3 'use strict';
 4
 5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
 9 }
10
>11 let result = addTwoNumber(5, 9);
12 console.log(result);
13
debug> c
break in debugDemo.js:7
 5 function addTwoNumber(a, b){
 6 // function returns the sum of the two numbers
> 7 debugger
 8   return a + b;
 9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $ 

```

使用 `watch(expression)` 命令添加您想监视其值的变量或表达式，使用 `restart` 重新启动应用和调试。

使用 `repl` 以交互方式输入代码。`repl` 模式与您正在调试的代码行具有相同的上下文。这允许您检查变量内容并测试代码行。按 `Ctrl+C` 退出调试 `repl`。

使用内置 Node 检查器

版本 \geq v6.3.0

你可以运行 Node 内置的v8 inspector！不再需要node-inspector插件。

只需传入 `inspector` 标志，你就会获得一个指向 `inspector` 的 URL

`node --inspect server.js`

使用 Node inspector

安装 node inspector：

`npm install -g node-inspector`

使用 `node-debug` 命令运行你的应用：

`node-debug filename.js`

之后，在 Chrome 中访问：

`http://localhost:8080/debug?port=5858`

```

ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
 1 // A Demo Code Showing the basic capabilities of the nodejs debugging module
 2
> 3 'use strict';
 4
 5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
 9 }
10
>11 let result = addTwoNumber(5, 9);
12 console.log(result);
13
debug> c
break in debugDemo.js:7
 5 function addTwoNumber(a, b){
 6 // function returns the sum of the two numbers
> 7 debugger
 8   return a + b;
 9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $ 

```

Use `watch(expression)` command to add the variable or expression whose value you want to watch and restart to restart the app and debugging.

Use `repl` to enter code interactively. The `repl` mode has the same context as the line you are debugging. This allows you to examine the contents of variables and test out lines of code. Press `Ctrl+C` to leave the debug `repl`.

Using Built-in Node inspector

Version \geq v6.3.0

You can run node's [built in](#) v8 inspector! The [node-inspector](#) plug-in is not needed anymore.

Simply pass the `inspector` flag and you'll be provided with a URL to the inspector

`node --inspect server.js`

Using Node inspector

Install the node inspector:

`npm install -g node-inspector`

Run your app with the `node-debug` command:

`node-debug filename.js`

After that, hit in Chrome:

`http://localhost:8080/debug?port=5858`

有时端口8080可能在您的计算机上不可用。您可能会遇到以下错误：

无法在0.0.0.0:8080启动服务器。错误：listen EACCES。

在这种情况下，使用以下命令在不同端口启动节点调试器。

```
$node-inspector --web-port=6500
```

你会看到类似如下内容：

```
// A Demo Code Showing the basic capabilities of the nodejs debugging module
'use strict';
function addTwoNumber(a, b){
  // function returns the sum of the two numbers
  return a + b;
}
var result = addTwoNumber(5, 9);
console.log(result);
```

Call Stack

Function	File	Ln	Col
anonymous(exports, require, module, ...)	nodejs/nodejsDebuggin...	10	1
Module._compile(content, filename)	module.js	409	26
Module._extensions.(module, filena...)	module.js	416	10
Module.load(filename)	module.js	343	32
Module._load(request, parent, isMain)	module.js	300	12
Module.runMain()	module.js	441	10
listOnTimeout()	timers.js	92	15

Local Variables

Variable	Value	Type
__dirname	/home/ubuntu/workspace/nodejs/nod...	string
__filename	/home/ubuntu/workspace/nodejs/nod...	string
addTwoNumber	function ()	function
exports	[Object]	object
module	[Object]	object
require	function ()	function

Sometimes port 8080 might not be available on your computer. You may get the following error:

Cannot start the server at 0.0.0.0:8080. Error: listen EACCES.

In this case, start the node inspector on a different port using the following command.

```
$node-inspector --web-port=6500
```

You will see something like this:

```
// A Demo Code Showing the basic capabilities of the nodejs debugging module
'use strict';
function addTwoNumber(a, b){
  // function returns the sum of the two numbers
  return a + b;
}
var result = addTwoNumber(5, 9);
console.log(result);
```

Call Stack

Function	File	Ln	Col
anonymous(exports, require, module, ...)	nodejs/nodejsDebuggin...	10	1
Module._compile(content, filename)	module.js	409	26
Module._extensions.(module, filena...)	module.js	416	10
Module.load(filename)	module.js	343	32
Module._load(request, parent, isMain)	module.js	300	12
Module.runMain()	module.js	441	10
listOnTimeout()	timers.js	92	15

Local Variables

Variable	Value	Type
__dirname	/home/ubuntu/workspace/nodejs/nod...	string
__filename	/home/ubuntu/workspace/nodejs/nod...	string
addTwoNumber	function ()	function
exports	[Object]	object
module	[Object]	object
require	function ()	function

第34章：无框架的Node服务器

第34.1节：无框架的Node服务器

```
var http = require('http');
var fs = require('fs');
var path = require('path');

http.createServer(function (request, response) {
  console.log('request ', request.url);

  var filePath = '.' + request.url;
  if (filePath == './')
    filePath = './index.html';

  var extname = String(path.extname(filePath)).toLowerCase();
  var contentType = 'text/html';
  var mimeTypes = {
    '.html': 'text/html',
    '.js': 'text/javascript',
    '.css': 'text/css',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpg',
    '.gif': 'image/gif',
    '.wav': 'audio/wav',
    '.mp4': 'video/mp4',
    '.woff': 'application/font-woff',
    '.ttf': 'application/font-ttf',
    '.eot': 'application/vnd.ms-fontobject',
    '.otf': 'application/font-otf',
    '.svg': 'application/image/svg+xml'
  };

  contentType = mimeTypes[extname] || 'application/octet-stream';

  fs.readFile(filePath, function(error, content) {
    if (error) {
      if(error.code == 'ENOENT'){
        fs.readFile('./404.html', function(error, content) {
          response.writeHead(200, { 'Content-Type': contentType });
          response.end(content, 'utf-8');
        });
      } else {
        response.writeHead(500);
        response.end('Sorry, check with the site admin for error: '+error.code+' ..\n');
      }
    } else {
      response.writeHead(200, { 'Content-Type': contentType });
      response.end(content, 'utf-8');
    }
  });

}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');
```

Chapter 34: Node server without framework

Section 34.1: Framework-less node server

```
var http = require('http');
var fs = require('fs');
var path = require('path');

http.createServer(function (request, response) {
  console.log('request ', request.url);

  var filePath = '.' + request.url;
  if (filePath == './')
    filePath = './index.html';

  var extname = String(path.extname(filePath)).toLowerCase();
  var contentType = 'text/html';
  var mimeTypes = {
    '.html': 'text/html',
    '.js': 'text/javascript',
    '.css': 'text/css',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpg',
    '.gif': 'image/gif',
    '.wav': 'audio/wav',
    '.mp4': 'video/mp4',
    '.woff': 'application/font-woff',
    '.ttf': 'application/font-ttf',
    '.eot': 'application/vnd.ms-fontobject',
    '.otf': 'application/font-otf',
    '.svg': 'application/image/svg+xml'
  };

  contentType = mimeTypes[extname] || 'application/octet-stream';

  fs.readFile(filePath, function(error, content) {
    if (error) {
      if(error.code == 'ENOENT'){
        fs.readFile('./404.html', function(error, content) {
          response.writeHead(200, { 'Content-Type': contentType });
          response.end(content, 'utf-8');
        });
      } else {
        response.writeHead(500);
        response.end('Sorry, check with the site admin for error: '+error.code+' ..\n');
      }
    } else {
      response.writeHead(200, { 'Content-Type': contentType });
      response.end(content, 'utf-8');
    }
  });

}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');
```

第34.2节：解决跨域资源共享（CORS）问题

```
// 你希望允许连接的网站  
response.setHeader('Access-Control-Allow-Origin', '*');

// 您希望允许的请求方法  
response.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');

// 您希望允许的请求头  
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// 如果您需要网站在发送到API的请求中包含cookie（例如使用会话时），请设置为true  
response.setHeader('Access-Control-Allow-Credentials', true);
```

belindoc.com

Section 34.2: Overcoming CORS Issues

```
// Website you wish to allow to connect to  
response.setHeader('Access-Control-Allow-Origin', '*');

// Request methods you wish to allow  
response.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');

// Request headers you wish to allow  
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// Set to true if you need the website to include cookies in the requests sent  
// to the API (e.g. in case you use sessions)  
response.setHeader('Access-Control-Allow-Credentials', true);
```

第35章：使用ES6的Node.JS

ES6、ECMAScript 6或ES2015是JavaScript的最新规范，为语言引入了一些语法糖。这是对语言的一次重大更新，带来了许多新特性。

有关Node和ES6的更多详细信息，请访问他们的网站 <https://nodejs.org/en/docs/es6/>

第35.1节：Node ES6支持及使用Babel创建项目

整个 ES6 规范尚未完全实现，因此您只能使用部分新特性。

您可以在 <http://node.green/> 查看当前支持的 ES6 特性列表。

自 NodeJS v6 起，支持情况相当不错。因此，如果您使用的是 NodeJS v6 或更高版本，就可以享受使用 ES6 的便利。不过，您可能还想使用一些未发布的特性以及更前沿的功能。为此，您需要使用转译器（transpiler）。

可以在运行时和构建时运行转译器，以使用所有 ES6 特性及更多功能。最流行的 JavaScript 转译器叫做 Babel。

Babel 允许您使用 ES6 规范中的所有特性以及一些额外的非规范特性，如 'stage-0' 阶段的 import 语法，替代 var 变量 = require('模块名') 的写法。

如果我们想创建一个使用 'stage-0' 特性（如 import）的项目，就需要添加 Babel 作为转译器。您会发现使用 React、Vue 以及其他基于 CommonJS 模式的项目经常实现 stage-0 特性。

创建一个新的 Node 项目

```
mkdir my-es6-app  
cd my-es6-app  
npm init
```

安装 Babel、ES6 预设和 stage-0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

创建一个名为server.js的新文件，并添加一个基本的HTTP服务器。

```
import http from 'http'  
  
http.createServer((req, res) => {  
  res.writeHead(200, {'Content-Type': 'text/plain'})  
  res.end('Hello World')  
}).listen(3000, '127.0.0.1')  
  
console.log('服务器运行在 http://127.0.0.1:3000/')
```

注意我们使用了import http from 'http'，这是一个stage-0特性，如果它能工作，说明我们的转译器配置正确。

如果你运行 node server.js，它会失败，因为不知道如何处理import。

在项目根目录创建一个.babelrc文件，并添加以下配置

Chapter 35: Node.JS with ES6

ES6, ECMAScript 6 or ES2015 is the latest [specification](#) for JavaScript which introduces some syntactic sugar to the language. It's a big update to the language and introduces a lot of new [features](#).

More details on Node and ES6 can be found on their site <https://nodejs.org/en/docs/es6/>

Section 35.1: Node ES6 Support and creating a project with Babel

The whole ES6 spec is not yet implemented in its entirety so you will only be able to use some of the new features. You can see a list of the current supported ES6 features at <http://node.green/>

Since NodeJS v6 there has been pretty good support. So if you using NodeJS v6 or above you can enjoy using ES6. However, you may also want to use some of the unreleased features and some from beyond. For this you will need to use a transpiler.

It is possible to run a transpiler at run time and build, to use all of the ES6 features and more. The most popular transpiler for JavaScript is called [Babel](#)

Babel allows you to use all of the features from the ES6 specification and some additional not-in-spec features with 'stage-0' such as `import` thing from '`thing`' instead of `var` `thing = require('thing')`

If we wanted to create a project where we use 'stage-0' features such as import we would need to add Babel as a transpiler. You'll see projects using react and Vue and other commonJS based patterns implement stage-0 quite often.

create a new node project

```
mkdir my-es6-app  
cd my-es6-app  
npm init
```

Install babel the ES6 preset and stage-0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

Create a new file called `server.js` and add a basic HTTP server.

```
import http from 'http'  
  
http.createServer((req, res) => {  
  res.writeHead(200, {'Content-Type': 'text/plain'})  
  res.end('Hello World\n')  
}).listen(3000, '127.0.0.1')  
  
console.log('Server running at http://127.0.0.1:3000/')
```

Note that we use an `import http from 'http'` this is a stage-0 feature and if it works it means we've got the transpiler working correctly.

If you run `node server.js` it will fail not knowing how to handle the import.

Creating a `.babelrc` file in the root of your directory and add the following settings

```
{  
  "presets": ["es2015", "stage-2"],  
  "plugins": []  
}
```

你现在可以用 `node src/index.js --exec babel-node` 来运行服务器。最后说明，在生产环境中运行时使用转译器并不是一个好主意。不过我们可以在 `package.json` 中实现一些脚本，使开发更方便。

```
"scripts": {  
  "start": "node dist/index.js",  
  "dev": "babel-node src/index.js",  
  "build": "babel src -d dist",  
  "postinstall": "npm run build"  
},
```

以上操作将在 `npm install` 时构建转译后的代码到 `dist` 目录，允许 `npm start` 使用转译后的代码来运行我们的生产应用。

`npm run dev` 将启动服务器和 `babel` 运行时，这在本地开发项目时是合适且推荐的。

更进一步，你可以安装 `nodemon` `npm install nodemon --save-dev` 来监视更改，然后重启 `node` 应用。

这确实加快了使用 `babel` 和 `NodeJS` 的开发速度。在你的 `package.json` 中只需更新 `"dev"` 脚本以使用 `nodemon`

```
"dev": "nodemon src/index.js --exec babel-node",
```

第35.2节：在你的 NodeJS 应用中使用 JS es6

JS es6（也称为 es2015）是一组 JS 语言的新特性，旨在使其在使用面向对象编程或面对现代开发任务时更加直观。

前提条件：

1. 请查看新 es6 特性，网址为 <http://es6-features.org> —— 如果你真的打算使用它，这可能会帮你理清思路在你下一个 NodeJS 应用中
2. 请在 <http://node.green> 查看您的节点版本兼容性等级
3. 如果一切正常——那就开始编码吧！

这里有一个非常简短的简单 `hello world` 应用示例，使用 JS es6

```
'use strict'  
  
class Program  
{  
  constructor()  
  {  
    this.message = 'hello es6 :)';  
  }  
  
  print()  
  {  
    setTimeout(() =>
```

```
{  
  "presets": ["es2015", "stage-2"],  
  "plugins": []  
}
```

you can now run the server with `node src/index.js --exec babel-node`

Finishing off it is not a good idea to run a transpiler at runtime on a production app. We can however implement some scripts in our `package.json` to make it easier to work with.

```
"scripts": {  
  "start": "node dist/index.js",  
  "dev": "babel-node src/index.js",  
  "build": "babel src -d dist",  
  "postinstall": "npm run build"  
},
```

The above will on `npm install` build the transpiled code to the `dist` directory allow `npm start` to use the transpiled code for our production app.

`npm run dev` will boot the server and `babel` runtime which is fine and preferred when working on a project locally.

Going one further you could then install `nodemon` `npm install nodemon --save-dev` to watch for changes and then reboot the node app.

This really speeds up working with `babel` and `NodeJS`. In your `package.json` just update the `"dev"` script to use `nodemon`

```
"dev": "nodemon src/index.js --exec babel-node",
```

Section 35.2: Use JS es6 on your NodeJS app

JS es6 (also known as es2015) is a set of new features to JS language aim to make it more intuitive when using OOP or while facing modern development tasks.

Prerequisites:

1. Check out the new es6 features at <http://es6-features.org> - it may clarify to you if you really intend to use it on your next NodeJS app
2. Check the compatibility level of your node version at <http://node.green>
3. If all is ok - let's code on!

Here is a very short sample of a simple `hello world` app with JS es6

```
'use strict'  
  
class Program  
{  
  constructor()  
  {  
    this.message = 'hello es6 :)';  
  }  
  
  print()  
  {  
    setTimeout(() =>
```

```

    {
    console.log(this.message);

    this.print();

    }, Math.random() * 1000);
}

new Program().print();

```

你可以运行这个程序，观察它如何一遍又一遍地打印相同的信息。

现在.. 让我们逐行解析：

```
'use strict'
```

如果你打算使用 js es6，实际上这行是必须的。严格模式 (strict mode) 有意与普通代码的语义不同（请在 MDN 上阅读更多相关内容 -

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

```
class Program
```

难以置信——一个class关键字！简单回顾一下——在 es6 之前，js 中定义类的唯一方式是使用 function关键字！

```
function MyClass() // 类定义
{
```

```
var myClassObject = new MyClass(); // 生成一个类型为 MyClass 的新对象
```

使用面向对象编程时，类是一项非常基础的能力，帮助开发者表示系统的特定部分（当代码变得庞大时，拆分代码非常关键.....例如：编写服务器端代码时）

```
constructor()
{
    this.message = 'hello es6 :)';
}
```

你得承认——这非常直观！这是我类的构造函数——每当从这个特定类创建对象时，这个独特的“函数”都会执行（在我们的程序中——只执行一次）

```
print()
{
setTimeout(() => // 这是一个“箭头”函数
{
    console.log(this.message);

    this.print(); // 这里我们调用类模板本身的“print”方法（在这个特定情况下是递归）

    }, Math.random() * 1000);
}
```

因为 print 定义在类的作用域内——它实际上是一个方法——可以从对象或

```

    {
        console.log(this.message);

        this.print();

    }, Math.random() * 1000);
}

new Program().print();

```

You can run this program and observe how it prints the same message over and over again.

Now.. let's break it down line by line:

```
'use strict'
```

This line is actually required if you intend to use js es6. strict mode, intentionally, has different semantics from normal code (please read more about it on MDN -

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

```
class Program
```

Unbelievable - a class keyword! Just for a quick reference - before es6 the only way to define a class in js was with the... function keyword!

```
function MyClass() // class definition
{
```

```
var myClassObject = new MyClass(); // generating a new object with a type of MyClass
```

When using OOP, a class is a very fundamental ability which assist the developer to represent a specific part of a system (breaking down code is crucial when the code is getting larger.. for instance: when writing server-side code)

```
constructor()
{
    this.message = 'hello es6 :)';
}
```

You got to admit - this is pretty intuitive! This is the c'tor of my class - this unique "function" will occur every time an object is created from this particular class (in our program - only once)

```
print()
{
    setTimeout(() => // this is an 'arrow' function
    {
        console.log(this.message);

        this.print(); // here we call the 'print' method from the class template itself (a recursion
                     // in this particular case)

    }, Math.random() * 1000);
}
```

Because print is defined in the class scope - it is actually a method - which can be invoked from either the object or

来自类外部或类内部！

那么.....到目前为止我们已经定义了我们的类.....是时候使用它了：

```
new Program().print();
```

这实际上等同于：

```
var prog = new Program(); // 定义一个类型为 'Program' 的新对象  
prog.print(); // 使用该程序打印自身
```

总之： JS es6 可以简化你的代码——使其更直观、更易理解（相比于之前版本的 JS）.....你可以尝试重写你现有的代码，亲自感受差异

祝你玩得开心 :)

belindoc.com

the class or from within the class itself!

So.. till now we defined our class.. time to use it:

```
new Program().print();
```

Which is truly equals to:

```
var prog = new Program(); // define a new object of type 'Program'  
prog.print(); // use the program to print itself
```

In conclusion: JS es6 can simplify your code - make it more intuitive and easy to understand (comparing with the previous version of JS).. you may try to re-write an existing code of yours and see the difference for yourself

ENJOY :)

第36章：与控制台交互

第36.1节：日志记录

控制台模块

类似于浏览器环境中的 JavaScript，node.js 提供了一个**console**模块，提供简单的日志记录和调试功能。

console 模块提供的最重要的方法是**console.log**、**console.error**和**console.time**。但还有其他一些方法，比如**console.info**。

console.log

参数将被打印到标准输出（stdout），并换行。

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

console.error

参数将被打印到标准错误（stderr），并换行。

```
console.error('哦，抱歉，发生了错误。');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

console.time, console.timeEnd

console.time 启动一个带有唯一标签的计时器，可用于计算操作的持续时间。当你使用相同标签调用 **console.timeEnd** 时，计时器停止，并将经过的时间（毫秒）打印到stdout。

```
> console.time("label");
undefined
> console.timeEnd("label");
label: 9297.320ms
```

进程模块

可以使用 **process** 模块直接写入控制台的标准输出。因此存在方法 **process.stdout.write**。与 **console.log** 不同，此方法不会在输出前添加换行符。

因此，在以下示例中，该方法被调用了两次，但它们的输出之间没有添加新行。

```
> process.stdout.write("123");process.stdout.write("456");
123456true
```

格式化

可以使用终端（控制）代码来发出特定命令，如切换颜色或定位光标。

```
> console.log("\033[31mThis will be red");
This will be red
```

Chapter 36: Interacting with Console

Section 36.1: Logging

Console Module

Similar to the browser environment of JavaScript node.js provides a **console** module which provides simple logging and debugging possibilities.

The most important methods provided by the console module are **console.log**, **console.error** and **console.time**. But there are several others like **console.info**.

console.log

The parameters will be printed to the standard output (stdout) with a new line.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

console.error

The parameters will be printed to the standard error (stderr) with a new line.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

console.time, console.timeEnd

console.time starts a timer with an unique label that can be used to compute the duration of an operation. When you call **console.timeEnd** with the same label, the timer stops and it prints the elapsed time in milliseconds to stdout.

```
> console.time("label");
undefined
> console.timeEnd("label");
label: 9297.320ms
```

Process Module

It is possible to use the **process** module to write **directly** into the standard output of the console. Therefore it exists the method **process.stdout.write**. Unlike **console.log** this method does not add a new line before your output.

So in the following example the method is called two times, but no new line is added in between their outputs.

```
> process.stdout.write("123");process.stdout.write("456");
123456true
```

Formatting

One can use **terminal (control) codes** to issue specific commands like switching colors or positioning the cursor.

```
> console.log("\033[31mThis will be red");
This will be red
```

一般

效果	代码
重置	\033[0m
高亮色	\033[1m
下划线	\033[4m
反转	\033[7m

字体颜色

效果	代码
黑色	\033[30m
红色	\033[31m
绿色	\033[32m
黄色	\033[33m
蓝色	\033[34m
品红色	\033[35m
青色	\033[36m
白色	\033[37m

背景颜色

效果	代码
黑色	\033[40m
红色	\033[41m
绿色	\033[42m
黄色	\033[43m
蓝色	\033[44m
Magenta	\033[45m
青色	\033[46m
白色	\033[47m

General

Effect	Code
Reset	\033[0m
Hicolor	\033[1m
Underline	\033[4m
Inverse	\033[7m

Font Colors

Effect	Code
Black	\033[30m
Red	\033[31m
Green	\033[32m
Yellow	\033[33m
Blue	\033[34m
Magenta	\033[35m
Cyan	\033[36m
White	\033[37m

Background Colors

Effect	Code
Black	\033[40m
Red	\033[41m
Green	\033[42m
Yellow	\033[43m
Blue	\033[44m
Magenta	\033[45m
Cyan	\033[46m
White	\033[47m

第37章：Cassandra集成

第37.1节：Hello world

要访问Cassandra，可以使用DataStax的[cassandra-driver](#)模块。它支持所有功能，并且可以轻松配置。

```
const cassandra = require("cassandra-driver");
const clientOptions = {
  contactPoints: ["host1", "host2"],
  keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
  if (err) {
    return console.error(err);
  }

  console.log(results.rows);
});
```

Chapter 37: Cassandra Integration

Section 37.1: Hello world

For accessing Cassandra [cassandra-driver](#) module from DataStax can be used. It supports all the features and can be easily configured.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
  contactPoints: ["host1", "host2"],
  keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
  if (err) {
    return console.error(err);
  }

  console.log(results.rows);
});
```

第38章：使用Node.js创建API

第38.1节：使用Express的GET接口

Node.js的接口可以很容易地在Express网络框架中构建。

下面的示例创建了一个用于列出所有用户的简单GET接口。

示例

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

var users =[{
id: 1,
    name: "约翰·多伊",
    age : 23,
email: "john@doe.com"
}];

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users); //返回JSON格式的响应
});

app.listen('3000', function(){
    console.log('服务器正在监听3000端口');
});
```

第38.2节：使用Express的POST接口

以下示例使用Express创建POST接口。该示例与GET示例类似，区别在于使用了body-parser来解析POST数据并将其添加到req.body中。

示例

```
var express = require('express');
var app = express();
// 用于解析POST请求中的body
var bodyParser = require('body-parser');

var users =[{
id: 1,
name: "约翰·多伊",
    age : 23,
email: "john@doe.com"
}];

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users);
});

/* POST /api/users
```

Chapter 38: Creating API's with Node.js

Section 38.1: GET api using Express

Node.js apis can be easily constructed in Express web framework.

Following example creates a simple GET api for listing all users.

Example

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

var users =[{
id: 1,
    name: "John Doe",
    age : 23,
email: "john@doe.com"
}];

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users); //return response as JSON
});

app.listen('3000', function(){
    console.log('Server listening on port 3000');
});
```

Section 38.2: POST api using Express

Following example create POST api using Express. This example is similar to GET example except the use of body-parser that parses the post data and add it to req.body.

Example

```
var express = require('express');
var app = express();
// for parsing the body in POST request
var bodyParser = require('body-parser');

var users =[{
id: 1,
    name: "John Doe",
    age : 23,
email: "john@doe.com"
}];

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users);
});

/* POST /api/users
```

```
{  
    "user": {  
        "id": 3,  
        "name": "测试用户",  
        "age" : 20,  
    "email": "test@test.com"  
    }  
}  
*/  
app.post('/api/users', function (req, res) {  
    var user = req.body.user;  
users.push(user);  
  
    return res.send('用户已成功添加');  
});  
  
app.listen('3000', function(){  
    console.log('服务器正在监听3000端口');  
});
```

```
{  
    "user": {  
        "id": 3,  
        "name": "Test User",  
        "age" : 20,  
    "email": "test@test.com"  
    }  
}  
*/  
app.post('/api/users', function (req, res) {  
    var user = req.body.user;  
users.push(user);  
  
    return res.send('User has been added successfully');  
});  
  
app.listen('3000', function(){  
    console.log('Server listening on port 3000');  
});
```

第39章：优雅关闭

第39.1节：优雅关闭 - SIGTERM

通过使用server.close()和process.exit()，我们可以捕获服务器异常并进行优雅关闭。

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

var server = http.createServer(function (req, res) { setTimeout(function () { //模拟一个长时间请求    res.writeHead(200, {'Content-Type': 'text/plain'});    res.end('你好，世界');}, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
  process.exit(0);
});});
```

Chapter 39: Graceful Shutdown

Section 39.1: Graceful Shutdown - SIGTERM

By using **server.close()** and **process.exit()**, we can catch the server exception and do a graceful shutdown.

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

var server = http.createServer(function (req, res) { setTimeout(function () { //simulate a long request    res.writeHead(200, {'Content-Type': 'text/plain'});    res.end('Hello World\n');}, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
  process.exit(0);
});});
```

第40章：使用 IISNode 在 IIS 中托管 Node.js Web 应用程序

第40.1节：通过 `<appSettings>` 使用 IIS 虚拟目录或嵌套应用程序

在 IIS 中使用虚拟目录或嵌套应用程序是一个常见场景，很可能是你在使用 IISNode 时想要利用的功能。

IISNode 不通过配置直接支持虚拟目录或嵌套应用程序，因此为了实现这一点，我们需要利用 IISNode 的一个不属于配置且鲜为人知的功能。所有 `<appSettings>` 元素的子项在 `Web.config` 中都会作为属性使用 `appSetting` 键添加到 `process.env` 对象中。

让我们在 `<appSettings>` 中创建一个虚拟目录

```
<appSettings>
  <add key="virtualDirPath" value="/foo" />
</appSettings>
```

在我们的 Node.js 应用中，我们可以访问 `virtualDirPath` 设置

```
console.log(process.env.virtualDirPath); // 输出 /foo
```

既然我们可以使用 `<appSettings>` 元素进行配置，就让我们利用它并在我们的服务器代码中使用它。

```
// 访问 virtualDirPath appSettings，并在不存在或未设置时赋予默认值 '/'
var virtualDirPath = process.env.virtualDirPath || '/';

// 我们还要确保 virtualDirPath
// 总是以正斜杠开头
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// 在应用的索引处设置路由
server.get(virtualDirPath, (req, res) => {
  return res.status(200).send('Hello World');
});
```

我们也可以将 `virtualDirPath` 用于我们的静态资源

```
// 公共目录
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower 组件
server.use('/bower_components', express.static(path.join(virtualDirPath, 'bower_components')));
```

让我们把这些放在一起

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

const port = process.env.PORT || 3000;
```

Chapter 40: Using IISNode to host Node.js Web Apps in IIS

Section 40.1: Using an IIS Virtual Directory or Nested Application via `<appSettings>`

Using a Virtual Directory or Nested Application in IIS is a common scenario and most likely one that you'll want to take advantage of when using IISNode.

IISNode doesn't provide direct support for Virtual Directories or Nested Applications via configuration so to achieve this we'll need to take advantage of a feature of IISNode that isn't part of the configuration and is much lesser known. All children of the `<appSettings>` element with the `Web.config` are added to the `process.env` object as properties using the `appSetting` key.

Lets create a Virtual Directory in our `<appSettings>`

```
<appSettings>
  <add key="virtualDirPath" value="/foo" />
</appSettings>
```

Within our Node.js App we can access the `virtualDirPath` setting

```
console.log(process.env.virtualDirPath); // prints /foo
```

Now that we can use the `<appSettings>` element for configuration, lets take advantage of that and use it in our server code.

```
// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
  return res.status(200).send('Hello World');
});
```

We can use the `virtualDirPath` with our static resources as well

```
// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath, 'bower_components')));
```

Lets put all of that together

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

const port = process.env.PORT || 3000;
```

```

// 访问 virtualDirPath appSettings，并在不存在或未设置时赋予默认值 '/'
var virtualDirPath = process.env.virtualDirPath || '/';

// 我们还要确保 virtualDirPath
// 总是以正斜杠开头
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// 公共目录
server.use(express.static(path.join(virtualDirPath, 'public')));

// Bower 组件
server.use('/bower_components', express.static(path.join(virtualDirPath, 'bower_components')));

// 在应用的索引处设置路由
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});

```

第40.2节：入门

IISNode 允许像托管.NET应用程序一样在IIS 7/8上托管Node.js Web应用程序。当然，你可以在Windows上自行托管你的node.exe进程，但既然可以直接在IIS中运行你的应用程序，为什么还要那样做呢。

IISNode 将处理跨多个核心的扩展、node.exe 的进程管理，以及在应用更新时自动回收您的 IIS 应用程序，这仅是其优势中的一部分。

要求

在您可以在 IIS 中托管 Node.js 应用之前，IISNode 有一些要求。

- 必须在 IIS 主机上安装 Node.js，支持 32 位或 64 位。
- IISNode 安装为 x86 或 x64，应与您的 IIS 主机的位数相匹配。
- [在您的 IIS 主机上安装 Microsoft URL-Rewrite Module for IIS](#)。
 - 这是关键，否则对您的 Node.js 应用的请求将无法按预期工作。
- 在您的 Node.js 应用根文件夹中有一个 Web.config 文件。
- 通过 iisnode.yml 文件或 Web.config 中的 <iisnode> 元素进行 IISNode 配置。

第 40.3 节：使用 Express 的基本 Hello World 示例

要使此示例正常工作，您需要在 IIS 主机上创建一个 IIS 7/8 应用程序，并将包含 Node.js Web 应用程序的目录添加为物理目录。确保您的应用程序/应用程序池身份可以访问 Node.js 安装目录。此示例使用的是 Node.js 64 位安装版本。

项目结构

这是 IISNode/Node.js Web 应用程序的基本项目结构。它看起来几乎与任何非 IISNode Web 应用程序相同，唯一的区别是多了一个Web.config文件。

```

- /app_root
  - package.json
  - server.js

```

```

// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));

// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath, 'bower_components')));

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});

```

Section 40.2: Getting Started

IISNode 允许 Node.js Web Apps to be hosted on IIS 7/8 just like a .NET application would. Of course, you can self host your node.exe process on Windows but why do that when you can just run your app in IIS.

IISNode will handle scaling over multiple cores, process management of node.exe, and auto-recycle your IIS Application whenever your app is updated, just to name a few of its [benefits](#).

Requirements

IISNode does have a few requirements before you can host your Node.js app in IIS.

- Node.js must be installed on the IIS host, 32-bit or 64-bit, either are supported.
- IISNode installed [x86](#) or [x64](#), this should match the bitness of your IIS Host.
- The [Microsoft URL-Rewrite Module for IIS](#) installed on your IIS host.
 - This is key, otherwise requests to your Node.js app won't function as expected.
- A [Web.config](#) in the root folder of your Node.js app.
- IISNode configuration via an [iisnode.yml](#) file or an [<iisnode>](#) element within your [Web.config](#).

Section 40.3: Basic Hello World Example using Express

To get this example working, you'll need to create an IIS 7/8 app on your IIS host and add the directory containing the Node.js Web App as the Physical Directory. Ensure that your Application/Application Pool Identity can access the Node.js install. This example uses the Node.js 64-bit installation.

Project Structure

This is the basic project structure of a IISNode/Node.js Web app. It looks almost identical to any non-IISNode Web App except for the addition of the [Web.config](#).

```

- /app_root
  - package.json
  - server.js

```

-Web.config

server.js - Express 应用程序

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

// 我们需要获取 IISNode 传递给我们的端口
// 使用 PORT 环境变量, 如果未设置则使用默认值
const port = process.env.PORT || 3000;

// 在应用程序的索引处设置路由
server.get('/', (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

配置 & Web.config

Web.config 就像其他任何 IIS 的 Web.config 文件一样，除了必须包含以下两项内容，URL `<rewrite><rules>` 和 IISNode `<handler>`。这两个元素都是 `<system.webServer>` 元素的子元素。

配置

您可以通过使用 iisnode.yml 文件或在 Web.config 的 `<system.webServer>` 中添加 `<iisnode>` 元素来配置 IISNode。这两种配置可以结合使用，但在这种情况下，Web.config 需要指定 iisnode.yml 文件，且任何配置冲突将以 iisnode.yml 文件中的配置为准。此配置覆盖不能反过来发生。

IISNode 处理程序

为了让 IIS 知道 server.js 包含我们的 Node.js Web 应用程序，我们需要明确告知它。我们可以通过向 `<handlers>` 元素添加 IISNode `<handler>` 来实现。

```
<handlers>
    <add name="iisnode" path="server.js" verb="*" modules="iisnode"/>
</handlers>
```

URL 重写规则

配置的最后一部分是确保进入我们 IIS 实例的、针对我们的 Node.js 应用的流量被定向到 IISNode。没有 URL 重写规则，我们需要通过访问 `http://<host>/server.js` 来访问我们的应用，更糟的是，当尝试请求由 server.js 提供的资源时，你会得到一个 404 错误。这就是为什么 IISNode Web 应用需要 URL 重写的原因。

```
<rewrite>
    <rules>
        <!-- 首先我们考虑传入的 URL 是否匹配 /public 文件夹中的物理文件
        -->
        <rule name="StaticContent" patternSyntax="Wildcard">
            <action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true"/>
            <conditions>
                <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"/>
            </conditions>
            <match url="*.*"/>
        </rule>
    </rules>
</rewrite>
```

- Web.config

server.js - Express Application

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

// We need to get the port that IISNode passes into us
// using the PORT environment variable, if it isn't set use a default value
const port = process.env.PORT || 3000;

// Setup a route at the index of our app
server.get('/', (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Configuration & Web.config

The Web.config is just like any other IIS Web.config except the following two things must be present, URL `<rewrite><rules>` and an IISNode `<handler>`. Both of these elements are children of the `<system.webServer>` element.

Configuration

You can configure IISNode by using a `iisnode.yml` file or by adding the `<iisnode>` element as a child of `<system.webServer>` in your Web.config. Both of these configuration can be used in conjunction with one another however, in this case, Web.config will need to specify the `iisnode.yml` file **AND** any configuration conflicts will be take from the `iisnode.yml` file instead. This configuration overriding cannot happen the other way around.

IISNode Handler

In order for IIS to know that `server.js` contains our Node.js Web App we need to explicitly tell it that. We can do this by adding the IISNode `<handler>` to the `<handlers>` element.

```
<handlers>
    <add name="iisnode" path="server.js" verb="*" modules="iisnode"/>
</handlers>
```

URL-Rewrite Rules

The final part of the configuration is ensuring that traffic intended for our Node.js app coming into our IIS instance is being directed to IISNode. Without URL rewrite rules, we would need to visit our app by going to `http://<host>/server.js` and even worse, when trying to request a resource supplied by `server.js` you'll get a 404. This is why URL rewriting is necessary for IISNode web apps.

```
<rewrite>
    <rules>
        <!-- First we consider whether the incoming URL matches a physical file in the /public folder
        -->
        <rule name="StaticContent" patternSyntax="Wildcard">
            <action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true"/>
            <conditions>
                <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"/>
            </conditions>
            <match url="*.*"/>
        </rule>
    </rules>
</rewrite>
```

```

<!-- 所有其他 URL 都映射到 Node.js 应用的入口点 -->
<rule name="DynamicContent">
  <conditions>
    <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
  </conditions>
  <action type="Rewrite" url="server.js"/>
</rule>
</rules>
</rewrite>

```

这是一个可用的Web.config文件，适用于本示例，配置为64位Node.js安装环境。

就是这样，现在访问您的IIS站点，查看您的Node.js应用程序是否正常运行。

第40.4节：在IISNode中使用Socket.io

要使Socket.io在IISNode中工作，当不使用虚拟目录/嵌套
应用程序时，唯一需要更改的是Web.config文件。

由于Socket.io发送的请求以/socket.io开头，IISNode需要通知IIS，这些请求也应由IISNode处理，而不仅仅是静态文
件请求或其他流量。这需要一个不同的<handler>，区别于标准的IISNode应用程序。

```

<handlers>
  <add name="iisnode-socketio" path="server.js" verb="*" modules="iisnode" />
</handlers>

```

除了对<handlers>的更改外，我们还需要添加一个额外的URL重写规则。该重写规则
将所有/socket.io流量发送到运行Socket.io服务器的服务器文件。

```

<rule name="SocketIO" patternSyntax="ECMAScript">
  <match url="socket.io.+"/>
  <action type="Rewrite" url="server.js"/>
</rule>

```

如果您使用的是IIS 8，除了添加上述处理程序和重写规则外，还需要在Web.config中禁用webSockets设置。IIS 7中
不需要这样做，因为它不支持WebSocket。

```
<webSocket enabled="false" />
```

```

<!-- All other URLs are mapped to the Node.js application entry point -->
<rule name="DynamicContent">
  <conditions>
    <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
  </conditions>
  <action type="Rewrite" url="server.js"/>
</rule>
</rules>
</rewrite>

```

This is a working Web.config file for this example, setup for a 64-bit Node.js install.

That's it, now visit your IIS Site and see your Node.js application working.

Section 40.4: Using Socket.io with IISNode

To get Socket.io working with IISNode, the only changes necessary when not using a Virtual Directory/Nested Application are within the Web.config.

Since Socket.io sends requests starting with /socket.io, IISNode needs to communicate to IIS that these should also be handled IISNode and aren't just static file requests or other traffic. This requires a different <handler> than standard IISNode apps.

```

<handlers>
  <add name="iisnode-socketio" path="server.js" verb="*" modules="iisnode" />
</handlers>

```

In addition to the changes to the <handlers> we also need to add an additional URL rewrite rule. The rewrite rule sends all /socket.io traffic to our server file where the Socket.io server is running.

```

<rule name="SocketIO" patternSyntax="ECMAScript">
  <match url="socket.io.+"/>
  <action type="Rewrite" url="server.js"/>
</rule>

```

If you are using IIS 8, you'll need to disable your webSockets setting in your Web.config in addition to adding the above handler and rewrite rules. This is unnecessary in IIS 7 since there is no webSocket support.

```
<webSocket enabled="false" />
```

第41章：命令行界面 (CLI)

第41.1节：命令行选项

`-v, --version`

新增于：v0.1.3 打印node的版本。

`-h, --help`

新增于：v0.1.3 打印node命令行选项。此选项的输出不如本文档详细。

`-e, --eval "script"`

新增于：v0.5.2 将以下参数作为JavaScript进行评估。REPL中预定义的模块也可以在脚本中使用。

`-p, --print "script"`

新增于：v0.6.4 与 `-e` 相同，但会打印结果。

`-c, --check`

新增于：v5.0.0 语法检查脚本但不执行。

`-i, --interactive`

新增于：v0.7.7 即使标准输入(stdin)看起来不是终端，也会打开REPL。

`-r, --require module`

新增于：v1.6.0 启动时预加载指定模块。

遵循 `require()` 的模块解析规则。`module` 可以是文件路径，也可以是Node模块名称。

`--no-deprecation`

添加于：v0.8.0 静默弃用警告。

`--trace-deprecation`

添加于：v0.8.0 打印弃用的堆栈跟踪。

`--throw-deprecation`

添加于：v0.11.14 对弃用抛出错误。

`--no-warnings`

添加于：v6.0.0 静默所有进程警告（包括弃用）。

`--trace-warnings`

Chapter 41: CLI

Section 41.1: Command Line Options

`-v, --version`

Added in: v0.1.3 Print node's version.

`-h, --help`

Added in: v0.1.3 Print node command line options. The output of this option is less detailed than this document.

`-e, --eval "script"`

Added in: v0.5.2 Evaluate the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script.

`-p, --print "script"`

Added in: v0.6.4 Identical to `-e` but prints the result.

`-c, --check`

Added in: v5.0.0 Syntax check the script without executing.

`-i, --interactive`

Added in: v0.7.7 Opens the REPL even if stdin does not appear to be a terminal.

`-r, --require module`

Added in: v1.6.0 Preload the specified module at startup.

Follows `require()`'s module resolution rules. `module` may be either a path to a file, or a node module name.

`--no-deprecation`

Added in: v0.8.0 Silence deprecation warnings.

`--trace-deprecation`

Added in: v0.8.0 Print stack traces for deprecations.

`--throw-deprecation`

Added in: v0.11.14 Throw errors for deprecations.

`--no-warnings`

Added in: v6.0.0 Silence all process warnings (including deprecations).

`--trace-warnings`

新增于：v6.0.0 打印进程警告的堆栈跟踪（包括弃用警告）。

--trace-sync-io

新增于：v2.1.0 当在事件循环的第一次执行后检测到同步 I/O 时打印堆栈跟踪。

--zero-fill-buffers

新增于：v6.0.0 自动将所有新分配的 Buffer 和 SlowBuffer 实例填充为零。

--preserve-symlinks

新增于：v6.3.0 指示模块加载器在解析和缓存模块时保留符号链接。

默认情况下，当 Node.js 从符号链接到不同磁盘位置的路径加载模块时，Node.js 会取消引用该链接，并使用模块的实际磁盘“真实路径”作为标识符以及定位其他依赖模块的根路径。在大多数情况下，这种默认行为是可以接受的。然而，当使用符号链接的同级依赖时，如下面的示例所示，如果 moduleA 尝试将 moduleB 作为同级依赖进行 require，默认行为会导致抛出异常：

```
{appDir}
  └── app
    ├── index.js
    └── node_modules
      └── moduleA -> {appDir}/moduleA
        └── moduleB
          ├── index.js
          └── package.json
    └── moduleA
      ├── index.js
      └── package.json
```

--preserve-symlinks 命令行标志指示 Node.js 使用模块的符号链接路径，而不是实际路径，从而允许找到符号链接的同级依赖。

但请注意，使用 --preserve-symlinks 可能会有其他副作用。具体来说，如果符号链接的本地模块从依赖树中的多个位置链接，可能会加载失败（Node.js 会将它们视为两个独立的模块，并尝试多次加载该模块，导致抛出异常）。

--track-heap-objects

新增于：v2.4.0 跟踪堆快照的堆对象分配。

--prof-process

新增于：v6.0.0 处理使用 v8 选项 --prof 生成的 v8 分析器输出。

--v8-options

新增于：v0.1.3 打印 v8 命令行选项。

注意：v8 选项允许单词由连字符（-）或下划线（_）分隔。

Added in: v6.0.0 Print stack traces for process warnings (including deprecations).

--trace-sync-io

Added in: v2.1.0 Prints a stack trace whenever synchronous I/O is detected after the first turn of the event loop.

--zero-fill-buffers

Added in: v6.0.0 Automatically zero-fills all newly allocated Buffer and SlowBuffer instances.

--preserve-symlinks

Added in: v6.3.0 Instructs the module loader to preserve symbolic links when resolving and caching modules.

By default, when Node.js loads a module from a path that is symbolically linked to a different on-disk location, Node.js will dereference the link and use the actual on-disk “real path” of the module as both an identifier and as a root path to locate other dependency modules. In most cases, this default behavior is acceptable. However, when using symbolically linked peer dependencies, as illustrated in the example below, the default behavior causes an exception to be thrown if moduleA attempts to require moduleB as a peer dependency:

```
{appDir}
  └── app
    ├── index.js
    └── node_modules
      └── moduleA -> {appDir}/moduleA
        └── moduleB
          ├── index.js
          └── package.json
    └── moduleA
      ├── index.js
      └── package.json
```

The --preserve-symlinks command line flag instructs Node.js to use the symlink path for modules as opposed to the real path, allowing symbolically linked peer dependencies to be found.

Note, however, that using --preserve-symlinks can have other side effects. Specifically, symbolically linked native modules can fail to load if those are linked from more than one location in the dependency tree (Node.js would see those as two separate modules and would attempt to load the module multiple times, causing an exception to be thrown).

--track-heap-objects

Added in: v2.4.0 Track heap object allocations for heap snapshots.

--prof-process

Added in: v6.0.0 Process v8 profiler output generated using the v8 option --prof.

--v8-options

Added in: v0.1.3 Print v8 command line options.

Note: v8 options allow words to be separated by both dashes (-) or underscores (_).

例如，`--stack-trace-limit` 等同于 `--stack_trace_limit`。

`--tls-cipher-list=list`

新增于：v4.0.0 指定备用的默认 TLS 密码列表。 (需要 Node.js 以支持 `crypto` 构建。)
(默认)

`--enable-fips`

新增于：v6.0.0 启用符合 FIPS 的加密启动。 (需要 Node.js 使用 `./configure --openssl-fips` 构建)

`--force-fips`

新增于：v6.0.0 强制在启动时使用符合 FIPS 的加密。 (无法通过脚本代码禁用) (与 `--enable-fips` 具有相同要求)

`--icu-data-dir=file`

新增于：v0.11.15 指定 ICU 数据加载路径。 (覆盖 `NODE_ICU_DATA`)

环境变量

`NODE_DEBUG=module[...]`

新增于：v0.1.32 用逗号分隔的核心模块列表，这些模块应打印调试信息。

`NODE_PATH=path[...]`

新增于：v0.1.32 用冒号分隔的目录列表，作为模块搜索路径的前缀。

注意：在 Windows 上，此列表使用分号分隔。

`NODE_DISABLE_COLORS=1`

新增于：v0.3.0 设置为 1 时，REPL 中将不使用颜色。

`NODE_ICU_DATA=file`

新增于：v0.11.15 ICU (Intl 对象) 数据的数据路径。编译时启用 `small-icu` 支持时，将扩展链接的数据。

`NODE_REPL_HISTORY=file`

新增于：v5.0.0 用于存储持久化REPL历史记录的文件路径。默认路径是`~/.node_repl_history`，该变量会覆盖默认路径。将该值设置为空字符串 (""" 或 " ") 则禁用持久化REPL历史记录。

For example, `--stack-trace-limit` is equivalent to `--stack_trace_limit`.

`--tls-cipher-list=list`

Added in: v4.0.0 Specify an alternative default TLS cipher list. (Requires Node.js to be built with `crypto` support.)
(Default)

`--enable-fips`

Added in: v6.0.0 Enable FIPS-compliant crypto at startup. (Requires Node.js to be built with `./configure --openssl-fips`)

`--force-fips`

Added in: v6.0.0 Force FIPS-compliant crypto on startup. (Cannot be disabled from script code.) (Same requirements as `--enable-fips`)

`--icu-data-dir=file`

Added in: v0.11.15 Specify ICU data load path. (overrides `NODE_ICU_DATA`)

Environment Variables

`NODE_DEBUG=module[, ...]`

Added in: v0.1.32 '-'-separated list of core modules that should print debug information.

`NODE_PATH=path[...]`

Added in: v0.1.32 ':'-separated list of directories prefixed to the module search path.

Note: on Windows, this is a ';' -separated list instead.

`NODE_DISABLE_COLORS=1`

Added in: v0.3.0 When set to 1 colors will not be used in the REPL.

`NODE_ICU_DATA=file`

Added in: v0.11.15 Data path for ICU (Intl object) data. Will extend linked-in data when compiled with `small-icu` support.

`NODE_REPL_HISTORY=file`

Added in: v5.0.0 Path to the file used to store the persistent REPL history. The default path is `~/.node_repl_history`，which is overridden by this variable. Setting the value to an empty string (""" or " ") disables persistent REPL history.

第42章：NodeJS框架

第42.1节：Web服务器框架

Express

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('示例应用正在监听3000端口！');
});
```

Koa

```
var koa = require('koa');
var app = koa();

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

第42.2节：命令行界面框架

Commander.js

```
var program = require('commander');

program
.version('0.0.1')

program
.command('hi')
.description('初始化项目配置')
.action(function(){
  console.log('Hi my Friend!!!');
});

程序
.command('bye [name]')
.description('初始化项目配置')
.action(function(name){
  console.log('再见 ' + name + '。很高兴见到你！');
});

程序
.command('*')
```

Chapter 42: NodeJS Frameworks

Section 42.1: Web Server Frameworks

Express

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Koa

```
var koa = require('koa');
var app = koa();

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

Section 42.2: Command Line Interface Frameworks

Commander.js

```
var program = require('commander');

program
.version('0.0.1')

program
.command('hi')
.description('initialize project configuration')
.action(function(){
  console.log('Hi my Friend!!!');
});

program
.command('bye [name]')
.description('initialize project configuration')
.action(function(name){
  console.log('Bye ' + name + '. It was good to see you!');
});

program
.command('*')
```

```
.action(function(env){  
  console.log('请输入有效命令');  
  terminate(true);  
});  
  
program.parse(process.argv);
```

Vorpal.js

```
const vorpal = require('vorpal')();  
  
vorpal  
.command('foo', '输出 "bar".')  
.action(function(args, callback) {  
  this.log('bar');  
callback();  
});  
  
vorpal  
.delimiter('myapp$')  
.show();
```

```
.action(function(env){  
  console.log('Enter a Valid command');  
  terminate(true);  
});  
  
program.parse(process.argv);
```

Vorpal.js

```
const vorpal = require('vorpal')();  
  
vorpal  
.command('foo', 'Outputs "bar".')  
.action(function(args, callback) {  
  this.log('bar');  
callback();  
});  
  
vorpal  
.delimiter('myapp$')  
.show();
```

第43章：grunt

第43.1节：GruntJs简介

Grunt是一个JavaScript任务运行器，用于自动化重复性任务，如压缩、编译、单元测试、代码检查等。

为了开始使用，你需要全局安装Grunt的命令行界面（CLI）。

```
npm install -g grunt-cli
```

准备一个新的Grunt项目：一个典型的设置会涉及向你的项目添加两个文件：package.json和Gruntfile。

package.json：此文件由npm用于存储作为npm模块发布的项目的元数据。你需要在此文件中将grunt和项目所需的Grunt插件列为devDependencies。

Gruntfile：此文件名为 Gruntfile.js，用于配置或定义任务并加载 Grunt 插件。

示例 package.json:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
  }
}
```

示例 gruntfile:

```
module.exports = function(grunt) {

  // 项目配置。
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/!*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */',
      }
    },
    build: {
      src: 'src/<%= pkg.name %>.js',
      dest: 'build/<%= pkg.name %>.min.js'
    }
  });

  // 加载提供"uglify"任务的插件。
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // 默认任务。
  grunt.registerTask('default', ['uglify']);
};


```

Chapter 43: grunt

Section 43.1: Introduction To GruntJs

Grunt is a JavaScript Task Runner, used for automation of repetitive tasks like minification, compilation, unit testing, linting, etc.

In order to get started, you'll want to install Grunt's command line interface (CLI) globally.

```
npm install -g grunt-cli
```

Preparing a new Grunt project: A typical setup will involve adding two files to your project: package.json and the Gruntfile.

package.json: This file is used by npm to store metadata for projects published as npm modules. You will list grunt and the Grunt plugins your project needs as devDependencies in this file.

Gruntfile: This file is named Gruntfile.js and is used to configure or define tasks and load Grunt plugins.

Example package.json:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
  }
}
```

Example gruntfile:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/!*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n',
      }
    },
    build: {
      src: 'src/<%= pkg.name %>.js',
      dest: 'build/<%= pkg.name %>.min.js'
    }
  });

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['uglify']);

};


```

第43.2节：安装grunt插件

添加依赖

要使用grunt插件，首先需要将其作为依赖添加到你的项目中。以jshint插件为例。

```
npm install grunt-contrib-jshint --save-dev
```

--save-dev 选项用于将插件添加到 package.json 中，这样插件会在每次 npm install 后自动安装。

加载插件

你可以在 gruntfile 文件中使用 loadNpmTasks 来加载你的插件。

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

配置任务

你可以在 gruntfile 中通过向传递给 grunt.initConfig 的对象添加一个名为 jshint 的属性来配置任务。

```
grunt.initConfig({
  jshint: {
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']
  }
});
```

别忘了你还可以为你使用的其他插件添加其他属性。

运行任务

要仅使用该插件运行任务，可以使用命令行。

```
grunt jshint
```

或者你可以将jshint添加到另一个任务中。

```
grunt.registerTask('default', ['jshint']);
```

默认任务通过终端中不带任何选项的grunt命令运行。

Section 43.2: Installing gruntplugins

Adding dependency

To use a gruntplugin, you first need to add it as a dependency to your project. Let's use the jshint plugin as an example.

```
npm install grunt-contrib-jshint --save-dev
```

The --save-dev option is used to add the plugin in the package.json, this way the plugin is always installed after a npm install.

Loading the plugin

You can load your plugin in the gruntfile file using loadNpmTasks.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Configuring the task

You configure the task in the gruntfile adding a property called jshint to the object passed to grunt.initConfig.

```
grunt.initConfig({
  jshint: {
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']
  }
});
```

Don't forget you can have other properties for other plugins you are using.

Running the task

To just run the task with the plugin you can use the command line.

```
grunt jshint
```

Or you can add jshint to another task.

```
grunt.registerTask('default', ['jshint']);
```

The default task runs with the grunt command in the terminal without any options.

第44章：在Node.js中使用WebSocket

第44.1节：安装WebSocket

有几种方法可以将WebSocket安装到你的项目中。以下是一些示例：

```
npm install --save ws
```

或者在你的package.json中使用：

```
"dependencies": {  
  "ws": "*"  
},
```

第44.2节：向你的文件中添加WebSocket

要向你的文件中添加ws，只需使用：

```
var ws = require('ws');
```

第44.3节：使用WebSocket和WebSocket服务器

要打开一个新的WebSocket，只需添加如下内容：

```
var WebSocket = require("ws");  
var ws = new WebSocket("ws://host:8080/OptionalPathName");  
// 继续你的代码...
```

或者要打开一个服务器，使用：

```
var WebSocketServer = require("ws").Server;  
var ws = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

第44.4节：一个简单的WebSocket服务器示例

```
var WebSocketServer = require('ws').Server  
, wss = new WebSocketServer({ port: 8080 }); // 如果你想添加路径，也可以使用 path:  
"PathName"  
  
wss.on('connection', function connection(ws) {  
  ws.on('message', function incoming(message) {  
    console.log('接收到: %s', message);  
  });  
  
  ws.send('某些内容');  
});
```

Chapter 44: Using WebSocket's with Node.js

Section 44.1: Installing WebSocket's

There are a few way's to install WebSocket's to your project. Here are some example's:

```
npm install --save ws
```

or inside your package.json using:

```
"dependencies": {  
  "ws": "*"  
},
```

Section 44.2: Adding WebSocket's to your file's

To add ws to your file's simply use:

```
var ws = require('ws');
```

Section 44.3: Using WebSocket's and WebSocket Server's

To open a new WebSocket, simply add something like:

```
var WebSocket = require("ws");  
var ws = new WebSocket("ws://host:8080/OptionalPathName");  
// Continue on with your code...
```

Or to open a server, use:

```
var WebSocketServer = require("ws").Server;  
var ws = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

Section 44.4: A Simple WebSocket Server Example

```
var WebSocketServer = require('ws').Server  
, wss = new WebSocketServer({ port: 8080 }); // If you want to add a path as well, use path:  
"PathName"  
  
wss.on('connection', function connection(ws) {  
  ws.on('message', function incoming(message) {  
    console.log('received: %s', message);  
  });  
  
  ws.send('something');  
});
```

第45章：metalsmith（金属工匠）

第45.1节：构建一个简单的博客

假设你已经安装并可使用 node 和 npm，创建一个包含有效package.json的项目文件夹。
安装必要的依赖：

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

在项目文件夹根目录下创建一个名为build.js的文件，内容如下：

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
.use(inPlace('handlebars'))
.build(function(err) {
  if (err) throw err;
  console.log('Build finished!');
});
```

在项目文件夹根目录下创建一个名为 src的文件夹。在 src中创建 index.html，内容如下：

```
---
title: My awesome blog
---

{{ title }}
```

运行 node build.js现在将构建 src中的所有文件。运行此命令后，你将在 build文件夹中得到 index.html，内容如下：

```
<h1>My awesome blog</h1>
```

Chapter 45: metalsmith

Section 45.1: Build a simple blog

Assuming that you have node and npm installed and available, create a project folder with a valid package.json.
Install the necessary dependencies:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Create a file called build.js at the root of your project folder, containing the following:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
.use(inPlace('handlebars'))
.build(function(err) {
  if (err) throw err;
  console.log('Build finished!');
});
```

Create a folder called src at the root of your project folder. Create index.html in src, containing the following:

```
---
title: My awesome blog
---

{{ title }}
```

Running node build.js will now build all files in src. After running this command, you'll have index.html in your build folder, with the following contents:

```
<h1>My awesome blog</h1>
```

第46章：解析命令行参数

第46.1节：传递动作（动词）和值

```
const options = require("commander");

options
  .option("-v, --verbose", "详细模式");

options
  .command("convert")
    .alias("c")
  .description("将输入文件转换为输出文件")
  .option("-i, --in-file <file_name>", "输入文件")
  .option("-o, --out-file <file_name>", "输出文件")
  .action(doConvert);

options.parse(process.argv);

if (!options.args.length) options.help();

function doConvert(options){
  //对 options.inFile 和 options.outFile 进行操作
}
```

第46.2节：传递布尔开关

```
const options = require("commander");

options
  .option("-v, --verbose")
  .parse(process.argv);

if (options.verbose){
  console.log("让我们制造点声音！");
}
```

Chapter 46: Parsing command line arguments

Section 46.1: Passing action (verb) and values

```
const options = require("commander");

options
  .option("-v, --verbose", "Be verbose");

options
  .command("convert")
    .alias("c")
  .description("Converts input file to output file")
  .option("-i, --in-file <file_name>", "Input file")
  .option("-o, --out-file <file_name>", "Output file")
  .action(doConvert);

options.parse(process.argv);

if (!options.args.length) options.help();

function doConvert(options){
  //do something with options.inFile and options.outFile
}
```

Section 46.2: Passing boolean switches

```
const options = require("commander");

options
  .option("-v, --verbose")
  .parse(process.argv);

if (options.verbose){
  console.log("Let's make some noise!");
}
```

第47章：客户端-服务器通信

第47.1节：使用Express、jQuery和Jade

```
//'client.jade'

//放置一个按钮；类似于HTML中的写法
button(type='button', id='send_by_button') 修改数据

#modify Lorem ipsum 发送者

//加载jQuery；也可以从在线资源加载
script(src='./js/jquery-2.2.0.min.js')

//使用jQuery的AJAX请求
脚本
$(function () {
  $('#send_by_button').click(function (e) {
    e.preventDefault();

    //测试：点击该按钮时，括号内的文本应显示
    //window.alert('你点击了我。 - jQuery');

    //代码中初始化的变量和JSON
    var predeclared = "肩盛";
    var data = {
      Title: "Name_SenderTest",
      Nick: predeclared,
      FirstName: "佐尔坦",
      Surname: "施密特"
    };

    //带有给定参数的AJAX请求
    $.ajax({
      类型: 'POST',
      数据: JSON.stringify(data),
      内容类型: 'application/json',
      地址: 'http://localhost:7776/domaintest',
      //成功时，接收到的数据作为'data'函数的输入
      成功: 函数 (data) {
        window.alert('请求已发送；数据已接收。');

        var jsonstr = JSON.stringify(data);
        var jsonobj = JSON.parse(jsonstr);

        //如果 JSON 的 'nick' 成员不等于预先声明的字符串（初始化时的值），则表示后端脚本已执行，意味着通信已建立
        if(data.Nick != predeclared){
          document.getElementById("modify").innerHTML = "JSON 已更改！" +jsonstr;
        }
      }
    });
  });
}

//'domaintest_route.js'
```

Chapter 47: Client-server communication

Section 47.1: /w Express, jQuery and Jade

```
//'client.jade'

//a button is placed down; similar in HTML
button(type='button', id='send_by_button') Modify data

#modify Lorem ipsum Sender

//loading jQuery; it can be done from an online source as well
script(src='./js/jquery-2.2.0.min.js')

//AJAX request using jQuery
script
  $(function () {
    $('#send_by_button').click(function (e) {
      e.preventDefault();

      //test: the text within brackets should appear when clicking on said button
      //window.alert('You clicked on me. - jQuery');

      //a variable and a JSON initialized in the code
      var predeclared = "Katamori";
      var data = {
        Title: "Name_SenderTest",
        Nick: predeclared,
        FirstName: "Zoltan",
        Surname: "Schmidt"
      };

      //an AJAX request with given parameters
      $.ajax({
        type: 'POST',
        data: JSON.stringify(data),
        contentType: 'application/json',
        url: 'http://localhost:7776/domaintest',
        //on success, received data is used as 'data' function input
        success: function (data) {
          window.alert('Request sent; data received.');

          var jsonstr = JSON.stringify(data);
          var jsonobj = JSON.parse(jsonstr);

          //if the 'nick' member of the JSON does not equal to the predeclared string
          //as it was initialized, then the backend script was executed, meaning that communication has been
          //established
          if(data.Nick != predeclared){
            document.getElementById("modify").innerHTML = "JSON changed!\n" +
              jsonstr;
          }
        }
      });
    });
  });
}

//'domaintest_route.js'
```

```

var express = require('express');
var router = express.Router();

// 一个监听GET请求的Express路由器——在本例中，它是空的，意味着当你访问 'localhost/domaintest' 时不会显示任何内容
router.get('/', function(req, res, next) {
});

// POST请求同理——注意，上面AJAX请求被定义为POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    // 这里生成内容
    var some_json = {
        Title: "Test",
        Item: "Crate"
    };

    var result = JSON.stringify(some_json);

    // 内容发送到 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";

    res.send(sent_data);
});

module.exports = router;

```

//基于个人使用的gist: <https://gist.github.com/Katamori/5c9850f02e4baf6e9896>

```

var express = require('express');
var router = express.Router();

//an Express router listening to GET requests - in this case, it's empty, meaning that nothing is displayed when you reach 'localhost/domaintest'
router.get('/', function(req, res, next) {
});

//same for POST requests - notice, how the AJAX request above was defined as POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    //content generated here
    var some_json = {
        Title: "Test",
        Item: "Crate"
    };

    var result = JSON.stringify(some_json);

    //content got 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";

    res.send(sent_data);
});

module.exports = router;

```

//based on a personally used gist: <https://gist.github.com/Katamori/5c9850f02e4baf6e9896>

第48章：Node.js设计基础

第48.1节：Node.js哲学

小核心，小模块：

构建小而单一功能的模块，不仅仅是代码大小方面，还包括服务单一目的的范围

- a - “小即是美”
- b - “让每个程序只做好一件事。”

反应器模式

反应器模式是Node.js异步特性的核心。允许系统作为单线程进程实现，配合一系列事件生成器和事件处理器，通过持续运行的事件循环实现。

Node.js的非阻塞I/O引擎libuv

观察者模式(EventEmitter)维护依赖者/观察者列表并通知它们

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

var ringBell = function ringBell()
{
  console.log('叮铃叮铃叮铃');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

Chapter 48: Node.js Design Fundamental

Section 48.1: The Node.js philosophy

Small Core, Small Module:

Build small and single purpose modules not in term of code size only, but also in term of scope that serves a single purpose

- a - "Small is beautiful"
- b - "Make each program do one thing well."

The Reactor Pattern

The Reactor Pattern is the heart of the node.js asynchronous nature. Allowed the system to be implemented as a single-threaded process with a series of event generators and event handlers, with the help of event loop that runs continuously.

The non-blocking I/O engine of Node.js – libuv -

The Observer Pattern(EventEmitter) maintains a list of dependents/observers and notifies them

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

var ringBell = function ringBell()
{
  console.log('tring tring tring');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

第49章：连接MongoDB

MongoDB是一个免费且开源的跨平台面向文档的数据库程序。作为NoSQL数据库程序，MongoDB使用带有模式的类JSON文档。

更多详情请访问 <https://www.mongodb.com/>

第49.1节：从Node.JS连接MongoDB的简单示例

```
MongoClient.connect('mongodb://localhost:27017/myNewDB', function (err, db) {  
    if(err)  
        console.log("无法连接数据库。错误：" + err)  
    else  
        console.log('已连接到数据库');  
  
    db.close();  
});
```

myNewDB 是数据库名称，如果数据库中不存在该名称，则此调用将自动创建。

第49.2节：将mongoDB与核心Node.JS连接的简单方法

```
var MongoClient = require('mongodb').MongoClient;  
  
//与mongoDB的连接  
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {  
    //检查连接  
    if(err){  
        console.log("连接失败。");  
    }else{  
        console.log("成功连接到mongoDB。");  
    }});
```

Chapter 49: Connect to Mongodb

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas.

For more details go to <https://www.mongodb.com/>

Section 49.1: Simple example to Connect mongoDB from Node.JS

```
MongoClient.connect('mongodb://localhost:27017/myNewDB', function (err, db) {  
    if(err)  
        console.log("Unable to connect DB. Error: " + err)  
    else  
        console.log('Connected to DB');  
  
    db.close();  
});
```

myNewDB is DB name, if it does not exists in database then it will create automatically with this call.

Section 49.2: Simple way to Connect mongoDB with core Node.JS

```
var MongoClient = require('mongodb').MongoClient;  
  
//connection with mongoDB  
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {  
    //check the connection  
    if(err){  
        console.log("connection failed.");  
    }else{  
        console.log("successfully connected to mongoDB.");  
    }});
```

第50章：性能挑战

第50.1节：使用Node处理长时间运行的查询

由于Node是单线程的，如果遇到长时间运行的计算，则需要变通方法。

注意：这是一个“即用型”示例。只是，别忘了获取 jQuery 并安装所需的模块。

此示例的主要逻辑：

1. 客户端向服务器发送请求。
2. 服务器在独立的节点实例中启动例程，并立即返回包含相关任务 ID 的响应。
3. 客户端持续向服务器发送检查请求，以获取指定任务 ID 的状态更新。

项目结构：

```
project
|   package.json
|   index.html
|
|---js
|   main.js
|   jquery-1.12.0.min.js
|
|---srv
|   app.js
|   models
|   task.js
|   tasks
data-processor.js
```

Chapter 50: Performance challenges

Section 50.1: Processing long running queries with Node

Since Node is single-threaded, there is a need of workaround if it comes to a long-running calculations.

Note: this is "ready to run" example. Just, don't forget to get jQuery and install the required modules.

Main logic of this example:

1. Client sends request to the server.
2. Server starts the routine in separate node instance and sends immediate response back with related task ID.
3. Client continuously sends checks to a server for status updates of the given task ID.

Project structure:

```
project
|   package.json
|   index.html
|
|---js
|   main.js
|   jquery-1.12.0.min.js
|
|---srv
|   app.js
|   models
|   task.js
|   tasks
|   data-processor.js
```

app.js:

```
var express      = require('express');
var app         = express();
var http        = require('http').Server(app);
var mongoose    = require('mongoose');
var bodyParser  = require('body-parser');

var childProcess= require('child_process');

var Task        = require('./models/task');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
  response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
  //create new task item for status tracking
  var t = new Task({ status: 'Starting ...' });
  data-processor.js
```

app.js:

```
var express      = require('express');
var app         = express();
var http        = require('http').Server(app);
var mongoose    = require('mongoose');
var bodyParser  = require('body-parser');

var childProcess= require('child_process');

var Task        = require('./models/task');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
  response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
  //create new task item for status tracking
  var t = new Task({ status: 'Starting ...' });
  data-processor.js
```

```

t.save(function(err, task){
    //在另一个线程中运行独立任务创建新的节点实例
    taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

    //处理来自任务处理器的消息
    taskProcessor.on('消息', function(msg){
        任务.状态 = msg.状态;
        任务.保存();
        } .绑定(this));
    });

    //完成后移除之前打开的节点实例
    taskProcessor.on('关闭', function(msg){
        this.终止();
    });

    //向我们的独立任务发送一些参数
    var 参数 = {
        消息: '来自主线程的问候'
    };

    taskProcessor.发送(参数);
    响应.状态(200).json(任务);
});

//路由检查请求是否完成计算
app.post('/is-ready', function(request, response){
    Task
    .findById(request.body.id)
    .exec(function(err, task){
        response.status(200).json(task);
    });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');

```

task.js:

```

var mongoose = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);

module.exports = mongoose.model('Task');

```

data-processor.js:

```

process.on('message', function(msg){
    init = function(){
    processData(msg.message);
    }.bind(this)();

    function processData(message){
        //向主应用发送状态更新
        process.send({ status: '我们已开始处理您的数据。' });
    }
});

```

```

t.save(function(err, task){
    //create new instance of node for running separate task in another thread
    taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

    //process the messages comming from the task processor
    taskProcessor.on('message', function(msg){
        task.status = msg.status;
        task.save();
        }.bind(this));

    //remove previously opened node instance when we finished
    taskProcessor.on('close', function(msg){
        this.kill();
    });

    //send some params to our separate task
    var params = {
        message: 'Hello from main thread'
    };

    taskProcessor.send(params);
    response.status(200).json(task);
});

//route to check is the request is finished the calculations
app.post('/is-ready', function(request, response){
    Task
    .findById(request.body.id)
    .exec(function(err, task){
        response.status(200).json(task);
    });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');

```

task.js:

```

var mongoose = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);

module.exports = mongoose.model('Task');

```

data-processor.js:

```

process.on('message', function(msg){
    init = function(){
    processData(msg.message);
    }.bind(this)();

    function processData(message){
        //send status update to the main app
        process.send({ status: 'We have started processing your data.' });
    }
});

```

```

//长时间计算中 ..
setTimeout(function(){
    process.send({ status: '完成！' });

    //通知节点，我们已完成此任务
process.disconnect();
}, 5000);
});

process.on('uncaughtException',function(err){
    console.log("发生错误: " + err.message + "" + err.stack + ".");
    console.log("优雅地结束例程。");
});

```

index.html:

```

<!DOCTYPE html>
<html>
<head>
<script src=".js/jquery-1.12.0.min.js"></script>
<script src=".js/main.js"></script>
</head>
<body>
<p>处理长时间运行节点请求的示例。</p>
<button id="go" type="button">运行</button>

<br />

<p>日志 :</p>
<textarea id="log" rows="20" cols="50"></textarea>
</body>
</html>

```

main.js:

```

$(document).on('ready', function(){

    $('#go').on('click', function(e){
        //clear log
        $("#log").val('');

        $.post("/long-running-request", {some_params: 'params' })
            .done(function(task){
                $("#log").val( $("#log").val() + " + task.status);

                //用于跟踪任务状态的函数
                function updateStatus(){
                    $.post("/is-ready", {id: task._id })
                        .done(function(response){
                            $("#log").val( $("#log").val() + " + response.status);

                            if(response.status != 'Done!'){
                                checkTaskTimeout = setTimeout(updateStatus, 500);
                            }
                        });
                }

                //开始检查任务
                var checkTaskTimeout = setTimeout(updateStatus, 100);
            });
    });
})

```

```

//long calculations ..
setTimeout(function(){
    process.send({ status: 'Done!' });

    //notify node, that we are done with this task
process.disconnect();
}, 5000);
});

process.on('uncaughtException',function(err){
    console.log("Error happened: " + err.message + "\n" + err.stack + ".\n");
    console.log("Gracefully finish the routine.");
});

```

index.html:

```

<!DOCTYPE html>
<html>
<head>
<script src=".js/jquery-1.12.0.min.js"></script>
<script src=".js/main.js"></script>
</head>
<body>
<p>Example of processing long-running node requests.</p>
<button id="go" type="button">Run</button>

<br />

<p>Log:</p>
<textarea id="log" rows="20" cols="50"></textarea>
</body>
</html>

```

main.js:

```

$(document).on('ready', function(){

    $('#go').on('click', function(e){
        //clear log
        $("#log").val('');

        $.post("/long-running-request", {some_params: 'params' })
            .done(function(task){
                $("#log").val( $("#log").val() + '\n' + task.status);

                //function for tracking the status of the task
                function updateStatus(){
                    $.post("/is-ready", {id: task._id })
                        .done(function(response){
                            $("#log").val( $("#log").val() + '\n' + response.status);

                            if(response.status != 'Done!'){
                                checkTaskTimeout = setTimeout(updateStatus, 500);
                            }
                        });
                }

                //start checking the task
                var checkTaskTimeout = setTimeout(updateStatus, 100);
            });
    });
})

```

```
});  
});
```

package.json :

```
{  
  "name": "nodeProcessor",  
  "dependencies": {  
    "body-parser": "^1.15.2",  
    "express": "^4.14.0",  
    "html": "0.0.10",  
    "mongoose": "^4.5.5"  
  }  
}
```

免责声明：本示例旨在为您提供基本的概念。要在生产环境中使用，需要进行改进。

```
});  
});
```

package.json:

```
{  
  "name": "nodeProcessor",  
  "dependencies": {  
    "body-parser": "^1.15.2",  
    "express": "^4.14.0",  
    "html": "0.0.10",  
    "mongoose": "^4.5.5"  
  }  
}
```

Disclaimer: this example is intended to give you basic idea. To use it in production environment, it needs improvements.

第51章：发送网页通知

第51.1节：使用GCM（谷歌云消息系统）发送网页通知

此类示例在PWA（渐进式网页应用）中广为人知，本例中我们将使用NodeJS和ES6发送一个简单的后端通知

1. 安装Node-GCM模块：npm install node-gcm
2. 安装Socket.io：npm install socket.io
3. 使用Google控制台创建一个支持GCM的应用程序。
4. 获取您的GCM应用程序ID（我们稍后会用到）
5. 获取您的GCM应用程序密钥。
6. 打开你喜欢的代码编辑器并添加以下代码：

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] 配置我们的 GCM 通道。
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
  data: {
    key1: 'msg1'
  }
});

// [*] 配置我们的静态文件。
app.use(express.static('public'));

// [*] 配置路由。
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// [*] 配置我们的 Socket 连接。
app.io.on('connection', socket => {
  console.log('我们有一个新的连接 ...');
  socket.on('new_user', (reg_id) => {
    // [*] 将我们的用户通知令牌添加到通常隐藏在秘密地方的列表中。
    if (regTokens.indexOf(reg_id) === -1) {
      regTokens.push(reg_id);

      // [*] 发送我们的推送消息
      sender.send(message, {
        registrationTokens: regTokens
      }, (err, response) => {
        if (err) console.error('err', err);
        else console.log(response);
      });
    }
  });
});
```

Chapter 51: Send Web Notification

Section 51.1: Send Web notification using GCM (Google Cloud Messaging System)

Such Example is knowing wide spreading among **PWAs** (Progressive Web Applications) and in this example we're going to send a simple Backend like notification using **NodeJS** and **ES6**

1. Install Node-GCM Module : npm **install** node-gcm
2. Install Socket.io : npm **install** socket.io
3. Create a GCM Enabled application using [Google Console](#).
4. Grabe your GCM Application Id (we will need it later on)
5. Grabe your GCM Application Secret code.
6. Open Your favorite code editor and add the following code :

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] Configuring our GCM Channel.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
  data: {
    key1: 'msg1'
  }
});

// [*] Configuring our static files.
app.use(express.static('public'));

// [*] Configuring Routes.
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// [*] Configuring our Socket Connection.
app.io.on('connection', socket => {
  console.log('we have a new connection ...');
  socket.on('new_user', (reg_id) => {
    // [*] Adding our user notification registration token to our list typically hided in a secret place.
    if (regTokens.indexOf(reg_id) === -1) {
      regTokens.push(reg_id);

      // [*] Sending our push messages
      sender.send(message, {
        registrationTokens: regTokens
      }, (err, response) => {
        if (err) console.error('err', err);
        else console.log(response);
      });
    }
  });
});
```

```
    })
  });
}

module.exports = app
```

附注：我这里使用了一个特殊的技巧，使 Socket.io 能够与 Express 一起工作，因为它默认情况下无法直接使用。

现在创建一个 json 文件，命名为：Manifest.json，打开它并粘贴以下内容：

```
{
  "name": "应用名称",
  "gcm_sender_id": "GCM 项目 ID"
}
```

关闭并保存到你的应用程序根目录中。

附注：Manifest.json 文件必须放在根目录，否则无法工作。

在上面的代码中，我做了以下操作：

1. 我设置并发送了一个普通的 index.html 页面，该页面也将使用 socket.io。
2. 我监听了一个 connection 事件，该事件由前端即我的 index.html 页面触发（当有新的客户端成功连接到我们预定义的链接时，该事件会被触发）
3. 我通过 socket.io 的 new_user 事件从 index.html 发送了一个特殊的令牌，称为注册令牌，该令牌将作为用户的唯一通行码，每个通行码通常由支持 Web 通知 API 的浏览器生成（更多内容请阅读[这里](#)）
4. 我只是使用 node-gcm 模块来发送我的通知，稍后会被处理并显示
[使用 Service Workers](#)。

这是从 **NodeJS** 的角度来看。在其他示例中，我将展示如何在我们的

推送消息中发送自定义数据、图标等。

附注：你可以在这里找到完整的工作演示。[_____](#)

```
    }
  });

module.exports = app
```

PS : I'm using here a special hack in order to make Socket.io works with Express because simply it doesn't work outside of the box.

Now Create a **.json** file and name it : **Manifest.json**, open it and past the following :

```
{
  "name": "Application Name",
  "gcm_sender_id": "GCM Project ID"
}
```

Close it and save in your application **ROOT** directory.

PS : the Manifest.json file needs to be in root directory or it won't work.

In the code above I'm doing the following :

1. I seted up and sent a normal index.html page that will use socket.io also.
2. I'm listening on a **connection** event fired from the **front-end** aka my **index.html page** (it will be fired once a new client successfully connected to our pre-defined link)
3. I'm sending a special token know's as the **registration token** from my index.html via socket.io **new_user** event, such token will be our user unique passcode and each code is generated usually from a supporting browser for the **Web notification API** (read more [here](#)).
4. I'm simply using the **node-gcm** module to send my notification which will be handled and shown later on using [Service Workers](#).

This is from **NodeJS** point of view. in other examples I will show how we can send custom data, icons ..etc in our push message.

PS : you can find the full working demo over [here](#).

第52章：Node.JS中的远程调试

第52.1节：在Linux上通过端口使用代理进行调试

如果你在Linux上启动应用程序，使用代理通过端口进行调试，例如：

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

然后使用端口9958进行远程调试。

第52.2节：NodeJS运行配置

要设置 Node 远程调试，只需使用 `--debug` 标志运行 `node` 进程。您可以使用 `--debug=<端口>` 来添加调试器应运行的端口。

当你的节点进程启动时，你应该会看到以下信息

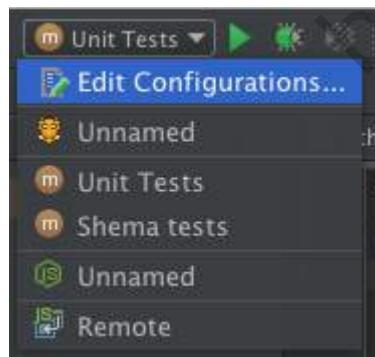
```
Debugger listening on port <port>
```

这将告诉你一切准备就绪。

然后你在特定的集成开发环境（IDE）中设置远程调试目标。

第52.3节：IntelliJ/Webstorm 配置

1. 确保已启用 NodeJS 插件
2. 选择你的运行配置（屏幕）



3. 选择 + > Node.js 远程调试

Chapter 52: Remote Debugging in Node.JS

Section 52.1: Use the proxy for debugging via port on Linux

If you start your application on Linux, use the proxy for debugging via port, for example:

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

Use port 9958 for remote debugging then.

Section 52.2: NodeJS run configuration

To set up Node remote debugging, simply run the node process with the `--debug` flag. You can add a port on which the debugger should run using `--debug=<port>`.

When your node process starts up you should see the message

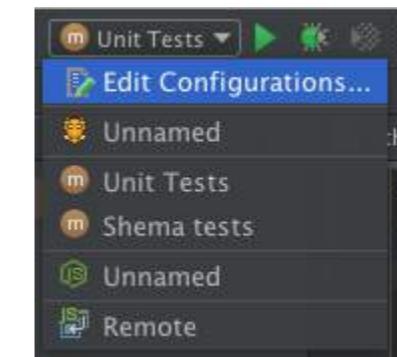
```
Debugger listening on port <port>
```

Which will tell you that everything is good to go.

Then you set up the remote debugging target in your specific IDE.

Section 52.3: IntelliJ/Webstorm Configuration

1. Make sure that the NodeJS plugin is enabled
2. Select your run configurations (screen)



3. Select + > Node.js Remote Debug



4. 确保输入上述选择的端口以及正确的主机地址

Name: Share Single instance only

Host:

Port:

配置完成后，只需像平常一样运行调试目标，程序将在断点处停止。



4. Make sure you enter the port selected above as well as the correct host

Name: Share Single instance only

Host:

Port:

Once those are configured simply run the debug target as you normally would and it will stop on your breakpoints.

第53章：数据库（MongoDB与Mongoose）

第53.1节：Mongoose连接

请确保先启动mongodb ! mongod --dbpath data/

package.json

```
"dependencies": {  
    "mongoose": "^4.5.5",  
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
const db = mongoose.connection;  
db.on('error', console.error.bind(console, '数据库连接错误！'));
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
var db = mongoose.connection;  
db.on('error', console.error.bind(console, '数据库连接错误！'));
```

第53.2节：模型

定义你的模型：

app/models/user.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
const userSchema = new mongoose.Schema({  
    name: String,  
    password: String  
});  
  
const User = mongoose.model('User', userSchema);  
  
export default User;
```

app/model/user.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
var userSchema = new mongoose.Schema({  
    name: String,  
    password: String  
});  
  
var User = mongoose.model('User', userSchema);
```

Chapter 53: Database (MongoDB with Mongoose)

Section 53.1: Mongoose connection

Make sure to have mongodb running first! mongod --dbpath data/
package.json

```
"dependencies": {  
    "mongoose": "^4.5.5",  
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
const db = mongoose.connection;  
db.on('error', console.error.bind(console, 'DB connection error!'));
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
var db = mongoose.connection;  
db.on('error', console.error.bind(console, 'DB connection error!'));
```

Section 53.2: Model

Define your model(s):

app/models/user.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
const userSchema = new mongoose.Schema({  
    name: String,  
    password: String  
});  
  
const User = mongoose.model('User', userSchema);  
  
export default User;
```

app/model/user.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
var userSchema = new mongoose.Schema({  
    name: String,  
    password: String  
});  
  
var User = mongoose.model('User', userSchema);
```

```
module.exports = User
```

第53.3节：插入数据

ECMA 6:

```
const user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save((err) => {
  if (err) throw err;

  console.log('User saved!');
});
```

ECMA5.1:

```
var user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save(function (err) {
  if (err) throw err;

  console.log('User saved!');
});
```

第53.4节：读取数据

ECMA6:

```
User.findOne({
  name: 'stack'
}, (err, user) => {
  if (err) throw err;

  if (!user) {
    console.log('未找到用户');
  } else {
    console.log('找到用户');
  }
});
```

ECMA5.1:

```
User.findOne({
  name: 'stack'
}, function (err, user) {
  if (err) throw err;

  if (!user) {
    console.log('未找到用户');
  } else {
    console.log('找到用户');
  }
});
```

```
module.exports = User
```

Section 53.3: Insert data

ECMA 6:

```
const user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save((err) => {
  if (err) throw err;

  console.log('User saved!');
});
```

ECMA5.1:

```
var user = new User({
  name: 'Stack',
  password: 'Overflow',
});

user.save(function (err) {
  if (err) throw err;

  console.log('User saved!');
});
```

Section 53.4: Read data

ECMA6:

```
User.findOne({
  name: 'stack'
}, (err, user) => {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```

ECMA5.1:

```
User.findOne({
  name: 'stack'
}, function (err, user) {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```

belindoc.com

第54章：良好的编码风格

第54.1节：注册的基本程序

通过这个例子，将说明如何将Node.js代码划分为不同的模块/文件夹以便更好地理解。采用这种技术可以让其他开发者更容易理解代码，因为他们可以直接查看相关文件，而不必浏览整个代码。主要用途是在团队合作中，当有新开发者后期加入时，他能更容易地融入代码本身。

index.js：该文件将管理服务器连接。

```
//导入库
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
request = require('request');

//导入自定义模块
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

//连接到Mongo数据库
mongoose.connect(config.getDBString());

//创建一个新的Express应用并进行配置
var app = express();

//配置路由
app.use(config.API_PATH, userRoutes);

//启动服务器
app.listen(config.PORT);
console.log('服务器启动于 - ' + config.URL + ":" + config.PORT);
```

config.js：该文件将管理所有相关的配置参数，这些参数在整个过程中保持不变。

```
var config = {
  版本: 1,
  构建: 1,
  URL: 'http://127.0.0.1',
  API_PATH : '/api',
  端口 : process.env.PORT || 8080,
  数据库 : {
    //MongoDB 配置
    主机 : 'localhost',
    端口 : '27017',
    数据库 : 'db'
  },
  /*
  * 获取连接 MongoDB 数据库的数据库连接字符串
  */
  getDBString : function(){
    return 'mongodb://' + this.DB.HOST + ':' + this.DB.PORT + '/' + this.DB.DATABASE;
  },
  /*
  * 获取 http URL
  */
```

Chapter 54: Good coding style

Section 54.1: Basic program for signup

Through this example, it will be explained to divide the **node.js** code into different **modules/folders** for better understandability. Following this technique makes it easier for other developers to understand the code, as they can directly refer to the concerned file instead of going through the whole code. The major use is when you are working in a team and a new developer joins at a later stage, it will get easier for him to gel up with the code itself.

index.js: This file will manage server connection.

```
//Import Libraries
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
    request = require('request');

//Import custom modules
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

//Connect to Mongo DB
mongoose.connect(config.getDBString());

//Create a new Express application and Configure it
var app = express();

//Configure Routes
app.use(config.API_PATH, userRoutes);

//Start the server
app.listen(config.PORT);
console.log('Server started at - ' + config.URL + ":" + config.PORT);
```

config.js: This file will manage all the configuration related params which will remain same throughout.

```
var config = {
  VERSION: 1,
  BUILD: 1,
  URL: 'http://127.0.0.1',
  API_PATH : '/api',
  PORT : process.env.PORT || 8080,
  DB : {
    //MongoDB configuration
    HOST : 'localhost',
    PORT : '27017',
    DATABASE : 'db'
  },
  /*
  * Get DB Connection String for connecting to MongoDB database
  */
  getDBString : function(){
    return 'mongodb://' + this.DB.HOST + ':' + this.DB.PORT + '/' + this.DB.DATABASE;
  },
  /*
  * Get the http URL
  */
```

```

/*
getHTTPUrl : function(){
    return 'http://' + this.URL + ":" + this.PORT;
}

module.exports = config;

```

user.js: 定义模式的模型文件

```

var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// 用户的模式
var UserSchema = new Schema({
    name: {
        type: String,
        // required: true
    },
    email: {
        type: '字符串'
    },
    password: {
        type: '字符串',
        //必填: true
    },
    dob: {
        type: '日期',
        //必填: true
    },
    gender: {
        type: '字符串', // 男女
        // 必填: true
    }
});

//定义用户模型
var 用户;
if(mongoose.models.用户)
    用户 = mongoose.model('用户');
else
    用户 = mongoose.model('用户', 用户Schema);

//导出用户模型
module.exports = User;

```

userController: 该文件包含用户注册的函数

```

var User = require('../models/user');
var crypto = require('crypto');

//用户控制器
var UserController = {

    //创建用户
    create: function(req, res){
        var repassword = req.body.repassword;
        var password = req.body.password;
        var userEmail = req.body.email;

        //检查邮箱地址是否已存在
        User.find({ "email": userEmail }, function(err, usr){

```

```

*/
getHTTPUrl : function(){
    return 'http://' + this.URL + ":" + this.PORT;
}

module.exports = config;

```

user.js: Model file where schema is defined

```

var mongoose = require('mongoose');
var Schema = mongoose.Schema;

//Schema for User
var UserSchema = new Schema({
    name: {
        type: String,
        // required: true
    },
    email: {
        type: String
    },
    password: {
        type: String,
        //required: true
    },
    dob: {
        type: Date,
        //required: true
    },
    gender: {
        type: String, // Male/Female
        // required: true
    }
});

```

```

//Define the model for User
var User;
if(mongoose.models.User)
    User = mongoose.model('User');
else
    User = mongoose.model('User', UserSchema);

//Export the User Model
module.exports = User;

```

userController: This file contains the function for user signUp

```

var User = require('../models/user');
var crypto = require('crypto');

//Controller for User
var UserController = {

    //Create a User
    create: function(req, res){
        var repassword = req.body.repassword;
        var password = req.body.password;
        var userEmail = req.body.email;

        //Check if the email address already exists
        User.find({ "email": userEmail }, function(err, usr){

```

```

if(usr.length > 0){
    //邮箱已存在
    res.json('邮箱已存在');
    return;
}
else {
    //新邮箱

    //检查密码是否相同
    if(password != repassword){
        res.json('密码不匹配');
        return;
    }

    //基于sha1生成密码哈希
    var shasum = crypto.createHash('sha1');
    shasum.update(req.body.password);
    var passwordHash = shasum.digest('hex');

    //创建用户
    var user = new User();
    user.name = req.body.name;
    user.email = req.body.email;
    user.password = passwordHash;
    user.dob = Date.parse(req.body.dob) || "";
    user.gender = req.body.gender;

    //验证用户
    user.validate(function(err){
        if(err){
            res.json(err);
            return;
        } else{
            //最终保存用户
            user.save(function(err){
                if(err)
                    {
                        res.json(err);
                        return;
                    }
                //发送用户详情前移除密码
                user.password = undefined;
                res.json(user);
                return;
            });
        }
    });
}

module.exports = UserController;

```

userRoutes.js : 这是UserController的路由

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

```

if(usr.length > 0){
    //Email Exists
    res.json('Email already exists');
    return;
}
else {
    //New Email

    //Check for same passwords
    if(password != repassword){
        res.json('Passwords does not match');
        return;
    }

    //Generate Password hash based on sha1
    var shasum = crypto.createHash('sha1');
    shasum.update(req.body.password);
    var passwordHash = shasum.digest('hex');

    //Create User
    var user = new User();
    user.name = req.body.name;
    user.email = req.body.email;
    user.password = passwordHash;
    user.dob = Date.parse(req.body.dob) || "";
    user.gender = req.body.gender;

    //Validate the User
    user.validate(function(err){
        if(err){
            res.json(err);
            return;
        } else{
            //Finally save the User
            user.save(function(err){
                if(err)
                    {
                        res.json(err);
                        return;
                    }
                //Remove Password before sending User details
                user.password = undefined;
                res.json(user);
                return;
            });
        }
    });
}

module.exports = UserController;

```

userRoutes.js: This the route for UserController

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

```
var UserController = require('../controllers/userController');

//用户路由
var UserRoutes = function(app)
{
    var router = express.Router();

    router.route('/users')
        .post(UserController.create);

    return router;
}

module.exports = UserRoutes;
```

belindoc.com

上面的例子看起来可能太大，但如果是一个对node.js有一点express知识的新手去学习，会发现它既简单又非常有帮助。

```
var UserController = require('../controllers/userController');

//Routes for User
var UserRoutes = function(app)
{
    var router = express.Router();

    router.route('/users')
        .post(UserController.create);

    return router;
}

module.exports = UserRoutes;
```

The above example may appear too big but if a beginner at node.js with a little blend of express knowledge tries to go through this will find it easy and really helpful.

第55章：Restful API设计：最佳实践

第55.1节：错误处理：获取所有资源

你如何处理错误，而不是将它们记录到控制台？

错误的做法：

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("即将保存请求", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: '保存请求时出错' });
      } else {
        res.json(r);
      }
    });
  });
}

更好的方式：
```

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        return next(err)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        return next(err)
      } else {
```

Chapter 55: Restful API Design: Best Practices

Section 55.1: Error Handling: GET all resources

How do you handle errors, rather than log them to the console?

Bad way:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: 'there was an error saving your r' });
      } else {
        res.json(r);
      }
    });
  });
}

Better way:
```

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        return next(err)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        return next(err)
      } else {
```

```
res.json(r);  
}  
});  
});
```

```
res.json(r);  
}  
});  
});
```

belindoc.com

第56章：传送HTML或任何其他类型的文件

第56.1节：在指定路径传送HTML

以下是如何创建一个Express服务器并默认提供index.html（空路径/），以及为/page1路径提供page1.html的方法。

文件夹结构

```
项目根目录
|   server.js
| __ views
|   |   index.html
|   |   page1.html
```

server.js

```
var express = require('express');
var path = require('path');
var app = express();

// 如果没有请求文件，则返回 index.html
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// 如果请求 page1，则返回 page1.html
app.get('/page1', function (request, response) {
  response.sendFile(path.join(__dirname, 'views', 'page1.html'), function(error) {
    if (error) {
      // 出错时执行某些操作
      console.log(err);
      response.end(JSON.stringify({error:"page not found"}));
    }
  });
}

app.listen(8080);
```

请注意，sendFile() 只是将静态文件作为响应流式传输，无法对其进行修改。如果你要提供一个 HTML 文件并希望包含动态数据，则需要使用 模板引擎，如 Pug、Mustache 或 EJS。

Chapter 56: Deliver HTML or any other sort of file

Section 56.1: Deliver HTML at specified path

Here's how to create an Express server and serve index.html by default (empty path /), and page1.html for /page1 path.

Folder structure

```
project root
|   server.js
| __ views
|   |   index.html
|   |   page1.html
```

server.js

```
var express = require('express');
var path = require('path');
var app = express();

// deliver index.html if no file is requested
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// deliver page1.html if page1 is requested
app.get('/page1', function (request, response) {
  response.sendFile(path.join(__dirname, 'views', 'page1.html'), function(error) {
    if (error) {
      // do something in case of error
      console.log(err);
      response.end(JSON.stringify({error:"page not found"}));
    }
  });
}

app.listen(8080);
```

Note that sendFile() just streams a static file as response, offering no opportunity to modify it. If you are serving an HTML file and want to include dynamic data with it, then you will need to use a *template engine* such as Pug, Mustache, or EJS.

第57章：TCP套接字

第57.1节：一个简单的TCP服务器

```
// 引入Nodejs的net模块。
const Net = require('net');
// 服务器监听的端口。
const port = 8080;

// 在代码中使用net.createServer()。这里仅作示例说明。
// 创建一个新的TCP服务器。
const server = new Net.Server();
// 服务器监听一个套接字，等待客户端发起连接请求。
// 可以将套接字看作一个端点。
server.listen(port, () => {
  console.log(`服务器正在监听本地主机端口 ${port} 的连接请求。`);
});

// 当客户端请求与服务器建立连接时，服务器会为该客户端创建一个新的
// 套接字。
server.on('connection', function(socket) {
  console.log('已建立新的连接。');

  // 既然TCP连接已建立，服务器可以通过向其套接字写入数据来发送数据给
  // 客户端。
  socket.write('你好，客户端。');

  // 服务器也可以通过从套接字读取数据来接收客户端发送的数据。
  socket.on('data', function(chunk) {
    console.log(`从客户端接收到数据：${chunk.toString()}。`);
  });
});

// 当客户端请求结束与服务器的TCP连接时，服务器会关闭该连接。

socket.on('end', function() {
  console.log('正在关闭与客户端的连接');
});

// 别忘了捕获错误，为了你自己着想。
socket.on('error', function(err) {
  console.log(`错误：${err}`);
});
});
```

第57.2节：一个简单的TCP客户端

```
// 引入Nodejs的net模块。
const Net = require('net');
// 服务器的端口号和主机名。
const port = 8080;
const host = 'localhost';

// 创建一个新的TCP客户端。
const client = new Net.Socket();
// 向服务器发送连接请求。
client.connect({ port: port, host: host }, () => {
  // 如果没有错误，服务器已接受请求并为我们创建了一个新的
  // 专用socket。
  console.log('与服务器的TCP连接已建立。');
});
```

Chapter 57: TCP Sockets

Section 57.1: A simple TCP server

```
// Include Nodejs' net module.
const Net = require('net');
// The port on which the server is listening.
const port = 8080;

// Use net.createServer() in your code. This is just for illustration purpose.
// Create a new TCP server.
const server = new Net.Server();
// The server listens to a socket for a client to make a connection request.
// Think of a socket as an end point.
server.listen(port, () => {
  console.log(`Server listening for connection requests on socket localhost:${port}`);
});

// When a client requests a connection with the server, the server creates a new
// socket dedicated to that client.
server.on('connection', function(socket) {
  console.log('A new connection has been established.');

  // Now that a TCP connection has been established, the server can send data to
  // the client by writing to its socket.
  socket.write('Hello, client.');

  // The server can also receive data from the client by reading from its socket.
  socket.on('data', function(chunk) {
    console.log(`Data received from client: ${chunk.toString()}`);
  });
});

// When the client requests to end the TCP connection with the server,
// ends the connection.
socket.on('end', () => {
  console.log('Closing connection with the client');
});

// Don't forget to catch error, for your own sake.
socket.on('error', function(err) {
  console.log(`Error: ${err}`);
});
```

Section 57.2: A simple TCP client

```
// Include Nodejs' net module.
const Net = require('net');
// The port number and hostname of the server.
const port = 8080;
const host = 'localhost';

// Create a new TCP client.
const client = new Net.Socket();
// Send a connection request to the server.
client.connect({ port: port, host: host }, () => {
  // If there is no error, the server has accepted the request and created a new
  // socket dedicated to us.
  console.log('TCP connection established with the server.');
});
```

```
// 客户端现在可以通过写入其socket向服务器发送数据。  
client.write('你好，服务器。');  
});  
  
// 客户端也可以通过读取其套接字从服务器接收数据。  
client.on('data', function(chunk) {  
  console.log(`从服务器接收到的数据: ${chunk.toString()}.`);  
  
  // 在接收到数据后请求断开连接。  
  client.end();  
});  
  
client.on('end', function() {  
  console.log('请求断开TCP连接');  
});
```

```
// The client can now send data to the server by writing to its socket.  
client.write('Hello, server.');//  
});  
  
// The client can also receive data from the server by reading from its socket.  
client.on('data', function(chunk) {  
  console.log(`Data received from the server: ${chunk.toString()}.`);  
  
  // Request an end to the connection after the data has been received.  
  client.end();  
});  
  
client.on('end', function() {  
  console.log('Requested an end to the TCP connection');  
});
```

belindoc.com

第58章：黑客

第58.1节：为require()添加新扩展

你可以通过扩展require.extensions来为require()添加新扩展。

以XML为例：

```
// 为require()添加.xml支持
require.extensions['.xml'] = (module, filename) => {
  const fs = require('fs')
  const xml2js = require('xml2js')

  module.exports = (callback) => {
    // 读取所需文件。
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        callback(err)
        return
      }
      // 解析它。
      xml2js.parseString(data, (err, result) => {
        callback(null, result)
      })
    })
  }
}
```

如果 hello.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>baz</bar>
  <qux />
</foo>
```

你可以通过 require() 来读取和解析它：

```
require('./hello')((err, xml) {
  if (err)
    抛出 err;
  console.log(err);
})
```

它打印 { foo: { bar: ['baz'], qux: [''] } }。

Chapter 58: Hack

Section 58.1: Add new extensions to require()

You can add new extensions to require() by extending require.extensions.

For a XML example:

```
// Add .xml for require()
require.extensions['.xml'] = (module, filename) => {
  const fs = require('fs')
  const xml2js = require('xml2js')

  module.exports = (callback) => {
    // Read required file.
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        callback(err)
        return
      }
      // Parse it.
      xml2js.parseString(data, (err, result) => {
        callback(null, result)
      })
    })
  }
}
```

If the content of hello.xml is following:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>baz</bar>
  <qux />
</foo>
```

You can read and parse it through require():

```
require('./hello')((err, xml) {
  if (err)
    throw err;
  console.log(err);
})
```

It prints { foo: { bar: ['baz'], qux: [''] } }.

第59章：Bluebird Promise

第59.1节：将nodeback库转换为Promise

```
const Promise = require('bluebird'),  
fs = require('fs')  
  
Promise.promisifyAll(fs)  
  
// 现在你可以在 'fs' 上使用带 Async 后缀的基于 Promise 的方法  
fs.readFileAsync('file.txt').then(contents => {  
  console.log(contents)  
}).catch(err => {  
  console.error('读取错误', err)  
})
```

第59.2节：函数式 Promise

map 示例：

```
Promise.resolve([ 1, 2, 3 ]).map(el => {  
  return Promise.resolve(el * el) // 在实际应用中返回某个异步操作  
})
```

filter 示例：

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {  
  return Promise.resolve(el % 2 === 0) // 在实际应用中返回某个异步操作  
}).then(console.log)
```

reduce 示例：

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {  
  return Promise.resolve(prev + curr) // 在实际应用中返回某些异步操作  
}).then(console.log)
```

第59.3节：协程（生成器）

```
const promiseReturningFunction = Promise.coroutine(function* (file) {  
  const data = yield fs.readFileAsync(file) // 这将返回一个Promise并解析为文件内容  
  
  return data.toString().toUpperCase()  
})  
  
promiseReturningFunction('file.txt').then(console.log)
```

第59.4节：自动资源释放（Promise.using）

```
function somethingThatReturnsADisposableResource() {  
  return getSomeResourceAsync(...).disposer(resource => {  
    resource.dispose()  
  })
}
```

Chapter 59: Bluebird Promises

Section 59.1: Converting nodeback library to Promises

```
const Promise = require('bluebird'),  
fs = require('fs')  
  
Promise.promisifyAll(fs)  
  
// now you can use promise based methods on 'fs' with the Async suffix  
fs.readFileAsync('file.txt').then(contents => {  
  console.log(contents)  
}).catch(err => {  
  console.error('error reading', err)  
})
```

Section 59.2: Functional Promises

Example of map:

```
Promise.resolve([ 1, 2, 3 ]).map(el => {  
  return Promise.resolve(el * el) // return some async operation in real world  
})
```

Example of filter:

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {  
  return Promise.resolve(el % 2 === 0) // return some async operation in real world  
}).then(console.log)
```

Example of reduce:

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {  
  return Promise.resolve(prev + curr) // return some async operation in real world  
}).then(console.log)
```

Section 59.3: Coroutines (Generators)

```
const promiseReturningFunction = Promise.coroutine(function* (file) {  
  const data = yield fs.readFileAsync(file) // this returns a Promise and resolves to the file  
  contents  
  
  return data.toString().toUpperCase()  
})  
  
promiseReturningFunction('file.txt').then(console.log)
```

Section 59.4: Automatic Resource Disposal (Promise.using)

```
function somethingThatReturnsADisposableResource() {  
  return getSomeResourceAsync(...).disposer(resource => {  
    resource.dispose()  
  })
}
```

```
Promise.using(somethingThatReturnsADisposableResource(), resource => {
  // 在这里使用资源, disposer将在Promise.using退出时自动关闭它
})
```

第59.5节：串行执行

```
Promise.resolve([1, 2, 3])
.mapSeries(el => Promise.resolve(el * el)) // 在实际应用中, 使用返回 Promise 的异步函数
.then(console.log)
```

```
Promise.using(somethingThatReturnsADisposableResource(), resource => {
  // use the resource here, the disposer will automatically close it when Promise.using exits
})
```

Section 59.5: Executing in series

```
Promise.resolve([1, 2, 3])
.mapSeries(el => Promise.resolve(el * el)) // in real world, use Promise returning async function
.then(console.log)
```

belindoc.com

第60章：异步/Await

Async/await 是一组关键字，允许以过程化的方式编写异步代码，而无需依赖回调（callback hell）或 Promise 链式调用（.then().then().then()）。

其原理是使用 await 关键字暂停异步函数的状态，直到 Promise 解决，并使用 async 关键字声明此类异步函数，这些函数返回一个 Promise。

Async/await 默认从 node.js 8 版本开始支持，7 版本需使用标志 --harmony-async-await。

第60.1节：Promise 与 Async/Await 的比较

使用 Promise 的函数：

```
function myAsyncFunction() {
  return aFunctionThatReturnsAPromise()
    // doSomething 是同步函数
    .then(result => doSomething(result))
    .catch(handleError);
}
```

这时 Async/Await 就派上用场了，使我们的函数更简洁：

```
async function myAsyncFunction() {
  let result;

  try {
    result = await aFunctionThatReturnsAPromise();
  } catch (error) {
    handleError(error);
  }

  // doSomething 是一个同步函数
  return doSomething(result);
}
```

所以关键字async类似于写return new Promise((resolve, reject) => {...})。

而await类似于在then回调中获取你的结果。

这里我放了一个非常简短的动图，看完后不会留下任何疑问：

[GIF](#)

第60.2节：带有Try-Catch错误处理的异步函数

async/await语法的最佳特性之一是可以使用标准的try-catch编码风格，就像你在编写同步代码一样。

```
const myFunc = async (req, res) => {
  try {
    const result = await somePromise();
  } catch (err) {
    // 在此处处理错误
  }
};
```

Chapter 60: Async/Await

Async/await is a set of keywords that allows writing of asynchronous code in a procedural manner without having to rely on callbacks (callback hell) or promise-chaining (.then().then().then()).

This works by using the await keyword to suspend the state of an async function, until the resolution of a promise, and using the async keyword to declare such async functions, which return a promise.

Async/await is available from node.js 8 by default or 7 using the flag --harmony-async-await.

Section 60.1: Comparison between Promises and Async/Await

Function using promises:

```
function myAsyncFunction() {
  return aFunctionThatReturnsAPromise()
    // doSomething is a sync function
    .then(result => doSomething(result))
    .catch(handleError);
}
```

So here is when Async/Await enter in action in order to get cleaner our function:

```
async function myAsyncFunction() {
  let result;

  try {
    result = await aFunctionThatReturnsAPromise();
  } catch (error) {
    handleError(error);
  }

  // doSomething is a sync function
  return doSomething(result);
}
```

So the keyword `async` would be similar to write `return new Promise((resolve, reject) => {...})`.

And `await` similar to get your result in `then` callback.

Here I leave a pretty brief gif that will not left any doubt in mind after seeing it:

[GIF](#)

Section 60.2: Async Functions with Try-Catch Error Handling

One of the best features of async/await syntax is that standard try-catch coding style is possible, just like you were writing synchronous code.

```
const myFunc = async (req, res) => {
  try {
    const result = await somePromise();
  } catch (err) {
    // handle errors here
  }
};
```

这是一个使用 Express 和 promise-mysql 的示例：

```
router.get('/flags/:id', async (req, res) => {
  try {
    const connection = await pool.createConnection();

    try {
      const sql = `SELECT f.id, f.width, f.height, f.code, f.filename
      FROM flags f
      WHERE f.id = ?
      LIMIT 1`;
      const flags = await connection.query(sql, req.params.id);
      if (flags.length === 0)
        return res.status(404).send({ message: 'flag not found' });

      return res.send({ flags[0] });
    } finally {
      pool.releaseConnection(connection);
    }
  } catch (err) {
    // 在这里处理错误
  }
});
```

第60.3节：在await处停止执行

如果promise没有返回任何内容，异步任务可以使用await完成。

```
try{
  await User.findByIdAndUpdate(user._id, {
    $push: {
      tokens: token
    }
  }).exec()
} catch(e){
  handleError(e)
}
```

第60.4节：从回调函数的进展

起初有回调函数，回调函数还可以：

```
const getTemperature = (callback) => {
  http.get('www.temperature.com/current', (res) => {
    callback(res.data.temperature)
  })
}

const getAirPollution = (callback) => {
  http.get('www.pollution.com/current', (res) => {
    callback(res.data.pollution)
  });
}

getTemperature(function(temp) {
```

Here's an example with Express and promise-mysql:

```
router.get('/flags/:id', async (req, res) => {
  try {
    const connection = await pool.createConnection();

    try {
      const sql = `SELECT f.id, f.width, f.height, f.code, f.filename
      FROM flags f
      WHERE f.id = ?
      LIMIT 1`;
      const flags = await connection.query(sql, req.params.id);
      if (flags.length === 0)
        return res.status(404).send({ message: 'flag not found' });

      return res.send({ flags[0] });
    } finally {
      pool.releaseConnection(connection);
    }
  } catch (err) {
    // handle errors here
  }
});
```

Section 60.3: Stops execution at await

If the promise doesn't return anything, the async task can be completed using await.

```
try{
  await User.findByIdAndUpdate(user._id, {
    $push: {
      tokens: token
    }
  }).exec()
} catch(e){
  handleError(e)
}
```

Section 60.4: Progression from Callbacks

In the beginning there were callbacks, and callbacks were ok:

```
const getTemperature = (callback) => {
  http.get('www.temperature.com/current', (res) => {
    callback(res.data.temperature)
  })
}

const getAirPollution = (callback) => {
  http.get('www.pollution.com/current', (res) => {
    callback(res.data.pollution)
  });
}

getTemperature(function(temp) {
```

```
getAirPollution(function(pollution) {
  console.log(`温度是 ${temp}，污染指数是 ${pollution}。`)
  // 温度是27，污染指数是0.5
})
})
```

但是回调函数存在一些非常令人沮丧的问题，所以我们都开始使用Promise。

```
const getTemperature = () => {
  return new Promise((resolve, reject) => {
    http.get('www.temperature.com/current', (res) => {
      resolve(res.data.temperature)
    })
  })
}

const getAirPollution = () => {
  return new Promise((resolve, reject) => {
    http.get('www.pollution.com/current', (res) => {
      resolve(res.data.pollution)
    })
  })
}

getTemperature()
.then(temp => console.log(`温度是 ${temp}`))
.then(() => getAirPollution())
.then(pollution => console.log(`污染程度是 ${pollution}`))
// 温度是 32
// 污染程度是 0.5
```

这稍微好一些。最后，我们找到了async/await。它底层仍然使用Promise。

```
const temp = await getTemperature()
const pollution = await getAirPollution()
```

```
getAirPollution(function(pollution) {
  console.log(`the temp is ${temp} and the pollution is ${pollution}.`)
  // The temp is 27 and the pollution is 0.5
})
})
```

But there were a few really frustrating issues with callbacks so we all started using promises.

```
const getTemperature = () => {
  return new Promise((resolve, reject) => {
    http.get('www.temperature.com/current', (res) => {
      resolve(res.data.temperature)
    })
  })
}

const getAirPollution = () => {
  return new Promise((resolve, reject) => {
    http.get('www.pollution.com/current', (res) => {
      resolve(res.data.pollution)
    })
  })
}

getTemperature()
.then(temp => console.log(`the temp is ${temp}`))
.then(() => getAirPollution())
.then(pollution => console.log(`and the pollution is ${pollution}`))
// the temp is 32
// and the pollution is 0.5
```

This was a bit better. Finally, we found async/await. Which still uses promises under the hood.

```
const temp = await getTemperature()
const pollution = await getAirPollution()
```

第61章：Koa 框架 v2

第61.1节：Hello World 示例

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
  ctx.body = 'Hello World'
})

app.listen(8080)
```

第61.2节：使用中间件处理错误

```
app.use(async (ctx, next) => {
  try {
    await next() // 尝试调用下游的下一个中间件
  } catch (err) {
    handleError(err, ctx) // 定义你自己的错误处理函数
  }
})
```

Chapter 61: Koa Framework v2

Section 61.1: Hello World example

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
  ctx.body = 'Hello World'
})

app.listen(8080)
```

Section 61.2: Handling errors using middleware

```
app.use(async (ctx, next) => {
  try {
    await next() // attempt to invoke the next middleware downstream
  } catch (err) {
    handleError(err, ctx) // define your own error handling function
  }
})
```

第62章：单元测试框架

第62.1节：Mocha 异步 (async/await)

```
const { expect } = require('chai')

describe('测试套件名称', function() {
  describe('#方法()', function() {
    it('应该无错误运行', async function() {
      const result = await answerToTheUltimateQuestion()
      expect(result).to.be.equal(42)
    })
  })
})
```

第62.2节：Mocha 同步

```
describe('测试套件名称', function() {
  describe('#方法()', function() {
    it('应该无错误运行', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

第62.3节：Mocha 异步 (回调)

```
var expect = require("chai").expect;
describe('测试套件名称', function() {
  describe('#方法()', function() {
    it('应该无错误运行', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

Chapter 62: Unit testing frameworks

Section 62.1: Mocha Asynchronous (async/await)

```
const { expect } = require('chai')

describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', async function() {
      const result = await answerToTheUltimateQuestion()
      expect(result).to.be.equal(42)
    })
  })
})
```

Section 62.2: Mocha synchronous

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

Section 62.3: Mocha asynchronous (callback)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

第63章：ECMAScript 2015 (ES6) 与 Node.js

第63.1节：const/let 声明

与var不同，const/let绑定的是词法作用域而非函数作用域。

```
{  
  var x = 1 // 会逃逸作用域  
  let y = 2 // 绑定到词法作用域  
  const z = 3 // 绑定到词法作用域, 常量  
}
```

```
console.log(x) // 1  
console.log(y) // ReferenceError: y 未定义  
console.log(z) // ReferenceError: z 未定义
```

[在 RunKit 中运行](#)

第63.2节：箭头函数

箭头函数会自动绑定到外围代码的 'this' 词法作用域。

```
performSomething(result => {  
  this.someVariable = result  
})
```

vs

```
performSomething(function(result) {  
  this.someVariable = result  
}.bind(this))
```

第63.3节：箭头函数示例

让我们来看这个示例，它输出数字3、5和7的平方：

```
let nums = [3, 5, 7]  
let squares = nums.map(function (n) {  
  return n * n  
})  
console.log(squares)
```

[在 RunKit 中运行](#)

传递给.map的函数也可以通过去掉function关键字，改用箭头=>来写成箭头函数：

```
let nums = [3, 5, 7]  
let squares = nums.map((n) => {  
  return n * n  
})  
console.log(squares)
```

Chapter 63: ECMAScript 2015 (ES6) with Node.js

Section 63.1: const/let declarations

Unlike `var`, `const/let` are bound to lexical scope rather than function scope.

```
{  
  var x = 1 // will escape the scope  
  let y = 2 // bound to lexical scope  
  const z = 3 // bound to lexical scope, constant  
}
```

```
console.log(x) // 1  
console.log(y) // ReferenceError: y is not defined  
console.log(z) // ReferenceError: z is not defined
```

[Run in RunKit](#)

Section 63.2: Arrow functions

Arrow functions automatically bind to the 'this' lexical scope of the surrounding code.

```
performSomething(result => {  
  this.someVariable = result  
})
```

vs

```
performSomething(function(result) {  
  this.someVariable = result  
}.bind(this))
```

Section 63.3: Arrow Function Example

Let's consider this example, that outputs the squares of the numbers 3, 5, and 7:

```
let nums = [3, 5, 7]  
let squares = nums.map(function (n) {  
  return n * n  
})  
console.log(squares)
```

[Run in RunKit](#)

The function passed to `.map` can also be written as arrow function by removing the `function` keyword and instead adding the arrow `=>`:

```
let nums = [3, 5, 7]  
let squares = nums.map((n) => {  
  return n * n  
})  
console.log(squares)
```

[在 RunKit 中运行](#)

但是，这可以写得更简洁。如果函数体只包含一条语句，并且该语句计算返回值，则可以去掉包裹函数体的花括号，以及return关键字。

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

[在 RunKit 中运行](#)

第63.4节：解构

```
let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

第63.5节：flow

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
});
```

第63.6节：ES6类

```
class 哺乳动物 {
  constructor(腿数){
    this.腿数 = 腿数;
  }
  吃(){
    console.log('正在吃...');
  }
  static 计数(){
    console.log('静态计数...');
  }
}
```

[Run in RunKit](#)

However, this can be written even more concise. If the function body consists of only one statement and that statement computes the return value, the curly braces of wrapping the function body can be removed, as well as the `return` keyword.

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

[Run in RunKit](#)

Section 63.4: destructuring

```
let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

Section 63.5: flow

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
});
```

Section 63.6: ES6 Class

```
class Mammel {
  constructor(legs){
    this.legs = legs;
  }
  eat(){
    console.log('eating...');
  }
  static count(){
    console.log('static count...');
  }
}
```

```
class Dog extends Mammel{
  constructor(name, legs){
    super(legs);
    this.name = name;
  }
  sleep(){
    super.eat();
  }
  console.log('sleeping');
}
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

belindoc.com

```
class Dog extends Mammel{
  constructor(name, legs){
    super(legs);
    this.name = name;
  }
  sleep(){
    super.eat();
    console.log('sleeping');
  }
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

第64章：使用Express.js路由AJAX请求

第64.1节：AJAX的简单实现

你应该已经有了基本的express-generator模板

在app.js中，添加（你可以在`var app = express.app()`之后的任何位置添加）：

```
app.post(function(req, res, next){  
    next();  
});
```

现在在你的index.js文件（或相应的匹配文件）中，添加：

```
router.get('/ajax', function(req, res){  
    res.render('ajax', {title: '一个Ajax示例', quote: "AJAX很棒！"});  
});  
router.post('/ajax', function(req, res){  
    res.render('ajax', {title: '一个Ajax示例', quote: req.body.quote});  
});
```

在/views目录下创建一个ajax.jade / ajax.pug 或 ajax.ejs文件，添加：

对于Jade/PugJS：

```
extends layout  
script(src="http://code.jquery.com/jquery-3.1.0.min.js")  
script(src="/magic.js")  
h1 引用: #{quote}  
form(method="post" id="changeQuote")  
    input(type='text', placeholder='设置每日名言', name='quote')  
    input(type="submit", value="保存")
```

对于 EJS：

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>  
<script src="/magic.js"></script>  
<h1>引用: <%=quote%> </h1>  
<form method="post" id="changeQuote">  
    <input type="text" placeholder="设置每日名言" name="quote"/>  
    <input type="submit" value="保存">  
</form>
```

现在，在/public目录下创建一个名为magic.js的文件

```
$(document).ready(function(){  
    $("form#changeQuote").on('submit', function(e){  
        e.preventDefault();  
        var data = $('input[name=quote]').val();  
        $.ajax({  
            类型: 'post',  
            网址: '/ajax',  
            数据: data,  
            数据类型: 'text'  
        })  
    })
```

Chapter 64: Routing AJAX requests with Express.js

Section 64.1: A simple implementation of AJAX

You should have the basic express-generator template

In app.js, add(you can add it anywhere after `var app = express.app()`):

```
app.post(function(req, res, next){  
    next();  
});
```

Now in your index.js file (or its respective match), add:

```
router.get('/ajax', function(req, res){  
    res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is great!"});  
});  
router.post('/ajax', function(req, res){  
    res.render('ajax', {title: 'An Ajax Example', quote: req.body.quote});  
});
```

Create an ajax.jade / ajax.pug or ajax.ejs file in /views directory, add:

For Jade/PugJS:

```
extends layout  
script(src="http://code.jquery.com/jquery-3.1.0.min.js")  
script(src="/magic.js")  
h1 Quote: !{quote}  
form(method="post" id="changeQuote")  
    input(type='text', placeholder='Set quote of the day', name='quote')  
    input(type="submit", value="Save")
```

For EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>  
<script src="/magic.js"></script>  
<h1>Quote: <%=quote%> </h1>  
<form method="post" id="changeQuote">  
    <input type="text" placeholder="Set quote of the day" name="quote"/>  
    <input type="submit" value="Save">  
</form>
```

Now, create a file in /public called magic.js

```
$(document).ready(function(){  
    $("form#changeQuote").on('submit', function(e){  
        e.preventDefault();  
        var data = $('input[name=quote]').val();  
        $.ajax({  
            type: 'post',  
            url: '/ajax',  
            data: data,  
            dataType: 'text'  
        })  
    })
```

```
.完成(函数(data){  
    $('#h1').html(data.quote);  
});  
});
```

就是这样！当你点击保存时，引用内容将会改变！

belindoc.com

```
.done(function(data){  
    $('#h1').html(data.quote);  
});  
});
```

And there you have it! When you click Save the quote will change!

第65章：向客户端发送文件流

第65.1节：使用 fs 和 pipe 从服务器流式传输静态文件

一个好的点播（视频点播）服务应该从基础做起。假设你的服务器上有一个目录
不对外公开，但你想通过某种门户或付费墙让用户访问你的
媒体内容。

```
var movie = path.resolve('./public/' + req.params.filename);

fs.stat(movie, function (err, stats) {
  var range = req.headers.range;
  if (!range) {
    return res.sendStatus(416);
  }

  //Chunk 逻辑在此
  var positions = range.replace(/bytes=/, "").split("-");
  var start = parseInt(positions[0], 10);
  var total = stats.size;
  var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
  var chunksize = (end - start) + 1;

res.writeHead(206, {
  'Transfer-Encoding': 'chunked',
  "Content-Range": "bytes " + start + "-" + end + "/" + total,
  "Accept-Ranges": "bytes",
  "Content-Length": chunksize,
  "Content-Type": mime.lookup(req.params.filename)
});

var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true })

.on('end', function () {
  console.log('Stream Done');
})

.on("error", function (err) {
  res.end(err);
})

.pipe(res, { end: true });
});
```

Chapter 65: Sending a file stream to client

Section 65.1: Using fs And pipe To Stream Static Files From The Server

A good VOD (Video On Demand) service should start with the basics. Lets say you have a directory on your server that is not publicly accessible, yet through some sort of portal or paywall you want to allow users to access your media.

```
var movie = path.resolve('./public/' + req.params.filename);

fs.stat(movie, function (err, stats) {
  var range = req.headers.range;
  if (!range) {
    return res.sendStatus(416);
  }

  //Chunk logic here
  var positions = range.replace(/bytes=/, "").split("-");
  var start = parseInt(positions[0], 10);
  var total = stats.size;
  var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
  var chunksize = (end - start) + 1;

res.writeHead(206, {
  'Transfer-Encoding': 'chunked',
  "Content-Range": "bytes " + start + "-" + end + "/" + total,
  "Accept-Ranges": "bytes",
  "Content-Length": chunksize,
  "Content-Type": mime.lookup(req.params.filename)
});

var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true })

.on('end', function () {
  console.log('Stream Done');
})

.on("error", function (err) {
  res.end(err);
})

.pipe(res, { end: true });
});
```

上面的代码片段是如何将视频流传输给客户端的基本示例。分块逻辑取决于多种因素，包括网络流量和延迟。平衡分块大小与数量。

最后，.pipe 调用让 node.js 知道要保持与服务器的连接，并在需要时发送额外的分块。

第65.2节：使用 fluent-ffmpeg 进行流式传输

你也可以使用 fluent-ffmpeg 将 .mp4 文件转换为 .flv 文件或其他类型：

```
res.contentType('flv');

var pathToMovie = './public/' + req.params.filename;

var proc = ffmpeg(pathToMovie)

.preset('flashvideo')

.on('end', function () {

    console.log('流结束');

})

.on('error', function (err) {

    console.log('发生错误: ' + err.message);

    res.send(err.message);

})

.pipe(res, { end: true });
```

The above snippet is a basic outline for how you would like to stream your video to a client. The chunk logic depends on a variety of factors, including network traffic and latency. It is important to balance chuck size vs. quantity.

Finally, the .pipe call lets node.js know to keep a connection open with the server and to send additional chunks as needed.

Section 65.2: Streaming Using fluent-ffmpeg

You can also use fluent-ffmpeg to convert .mp4 files to .flv files, or other types:

```
res.contentType('flv');

var pathToMovie = './public/' + req.params.filename;

var proc = ffmpeg(pathToMovie)

.preset('flashvideo')

.on('end', function () {

    console.log('Stream Done');

})

.on('error', function (err) {

    console.log('an error happened: ' + err.message);

    res.send(err.message);

})

.pipe(res, { end: true });
```

第66章：NodeJS与Redis

第66.1节：入门

node_redis，正如你可能猜到的，是Node.js的Redis客户端。你可以通过npm使用以下命令安装它。

```
npm install redis
```

安装node_redis模块后，你就可以开始了。让我们创建一个简单的文件app.js，看看如何从Node.js连接到Redis。

app.js

```
var redis = require('redis');
client = redis.createClient(); //创建一个新的客户端
```

默认情况下，redis.createClient()会使用127.0.0.1和6379作为主机名和端口。如果你有不同的主机/端口，可以按如下方式提供：

```
var client = redis.createClient(port, host);
```

现在，一旦连接建立，你就可以执行某些操作。基本上，你只需要监听如下所示的connect事件。

```
client.on('connect', function() {
  console.log('connected');
});
```

所以，以下代码片段放入 app.js 中：

```
var redis = require('redis');
var client = redis.createClient();

client.on('connect', function() {
  console.log('connected');
});
```

现在，在终端输入 node app 来运行该应用。运行此代码片段前，请确保你的 Redis 服务器已启动并运行。

第66.2节：存储键值对

既然你已经知道如何从 Node.js 连接到 Redis，接下来让我们看看如何在 Redis 存储中存储键值对。

存储字符串

所有 Redis 命令都作为客户端对象上的不同函数暴露出来。要存储一个简单的字符串，使用以下语法：

Chapter 66: NodeJS with Redis

Section 66.1: Getting Started

node_redis, as you may have guessed, is the [Redis client for Node.js](#). You can install it via npm using the following command.

```
npm install redis
```

Once you have installed node_redis module you are good to go. Let's create a simple file, app.js, and see how to connect with Redis from Node.js.

app.js

```
var redis = require('redis');
client = redis.createClient(); //creates a new client
```

By default, redis.createClient() will use 127.0.0.1 and 6379 as the hostname and port respectively. If you have a different host/port you can supply them as following:

```
var client = redis.createClient(port, host);
```

Now, you can perform some action once a connection has been established. Basically, you just need to listen for connect events as shown below.

```
client.on('connect', function() {
  console.log('connected');
});
```

So, the following snippet goes into app.js:

```
var redis = require('redis');
var client = redis.createClient();

client.on('connect', function() {
  console.log('connected');
});
```

Now, type node app in the terminal to run the app. Make sure your Redis server is up and running before running this snippet.

Section 66.2: Storing Key-Value Pairs

Now that you know how to connect with Redis from Node.js, let's see how to store key-value pairs in Redis storage.

Storing Strings

All the Redis commands are exposed as different functions on the client object. To store a simple string use the following syntax:

```
client.set('framework', 'AngularJS');
```

或者

```
client.set(['framework', 'AngularJS']);
```

以上代码片段将简单字符串 AngularJS 存储在键 framework 下。你应该注意，这两个代码片段做的是同一件事。唯一的区别是第一个传递可变数量的参数，而后者传递一个 args 数组给 client.set() 函数。你也可以传递一个可选的回调函数，以便在操作完成时收到通知：

```
client.set('framework', 'AngularJS', function(err, reply) {
  console.log(reply);
});
```

如果操作因某种原因失败，回调函数中的 err 参数表示错误。要获取该键的值，请执行以下操作：

```
client.get('framework', function(err, reply) {
  console.log(reply);
});
```

client.get() 允许你从 Redis 中检索存储的键。键的值可以通过回调函数的参数 reply 访问。如果键不存在，reply 的值将为空。

存储哈希

很多时候，存储简单值无法解决你的问题。你需要在 Redis 中存储哈希（对象）。为此，你可以使用 hmset() 函数，如下所示：

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css', 'Bootstrap', 'node', 'Express');

client.hgetall('frameworks', function(err, object) {
  console.log(object);
});
```

上面的代码片段在 Redis 中存储了一个哈希，映射每种技术到其框架。第一个参数是 hmset() 的键名。后续参数表示键值对。类似地，hgetall() 用于获取键的值。如果找到该键，回调函数的第二个参数将包含该值，且该值是一个对象。

注意 Redis 不支持嵌套对象。对象中的所有属性值在存储前都会被强制转换为字符串。你也可以使用以下语法在 Redis 中存储对象：

```
client.hmset('frameworks', {
  'javascript': 'AngularJS',
  'css': 'Bootstrap',
  'node': 'Express'
});
```

还可以传入一个可选的回调函数，以便知道操作何时完成。

所有函数（命令）都可以用大小写等效形式调用。例如，client.hmset()

```
client.set('framework', 'AngularJS');
```

Or

```
client.set(['framework', 'AngularJS']);
```

The above snippets store a simple string AngularJS against the key framework. You should note that both the snippets do the same thing. The only difference is that the first one passes a variable number of arguments while the later passes an args array to client.set() function. You can also pass an optional callback to get a notification when the operation is complete:

```
client.set('framework', 'AngularJS', function(err, reply) {
  console.log(reply);
});
```

If the operation failed for some reason, the err argument to the callback represents the error. To retrieve the value of the key do the following:

```
client.get('framework', function(err, reply) {
  console.log(reply);
});
```

client.get() lets you retrieve a key stored in Redis. The value of the key can be accessed via the callback argument reply. If the key doesn't exist, the value of reply will be empty.

Storing Hash

Many times storing simple values won't solve your problem. You will need to store hashes (objects) in Redis. For that you can use hmset() function as following:

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css', 'Bootstrap', 'node', 'Express');

client.hgetall('frameworks', function(err, object) {
  console.log(object);
});
```

The above snippet stores a hash in Redis that maps each technology to its framework. The first argument to hmset() is the name of the key. Subsequent arguments represent key-value pairs. Similarly, hgetall() is used to retrieve the value of the key. If the key is found, the second argument to the callback will contain the value which is an object.

Note that Redis doesn't support nested objects. All the property values in the object will be coerced into strings before getting stored. You can also use the following syntax to store objects in Redis:

```
client.hmset('frameworks', {
  'javascript': 'AngularJS',
  'css': 'Bootstrap',
  'node': 'Express'
});
```

An optional callback can also be passed to know when the operation is completed.

All the functions (commands) can be called with uppercase/lowercase equivalents. For example, client.hmset()

和 client.HMSET() 是相同的。存储列表

如果你想存储一个项目列表，可以使用 Redis 列表。存储列表的语法如下：

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err, reply) {
  console.log(reply); //打印 2
});
```

上述代码片段创建了一个名为 frameworks 的列表，并向其中添加了两个元素。因此，列表的长度现在是两。如你所见，我传递了一个 args 数组给 rpush。数组的第一个元素表示键的名称，剩余的元素表示列表的内容。你也可以使用 lpush() 代替 rpush()，将元素推入列表的左侧。

要检索列表中的元素，可以使用lrange()函数，方法如下：

```
client.lrange('frameworks', 0, -1, function(err, reply) {
  console.log(reply); // ['angularjs', 'backbone']
});
```

只需注意，通过将 -1 作为第三个参数传递给 lrange()，可以获取列表中的所有元素。如果您想要列表的子集，则应在此处传递结束索引。

存储集合

集合类似于列表，但区别在于它们不允许重复。因此，如果你不想让列表中有任何重复元素，可以使用集合。下面是如何将之前的代码片段修改为使用集合而不是列表的方法。

```
client.sadd(['标签', 'angularjs', 'backbonejs', 'emberjs'], function(err, reply) {
  console.log(reply); // 3
});
```

如您所见，sadd() 函数会创建一个包含指定元素的新集合。这里，集合的长度是三。要获取集合的成员，请使用如下的 smembers() 函数：

```
client.成员('tags', 函数(错误, 回复) {
  控制台.日志(回复);
});
```

此代码片段将检索集合中的所有成员。请注意，检索成员时顺序不会被保留。

这是每个使用 Redis 的应用程序中最重要的数据结构列表。除了字符串、列表、集合和哈希之外，你还可以在 Redis 中存储有序集合、HyperLogLogs 等。如果你想要完整的命令和数据结构列表，请访问官方 Redis 文档。请记住，几乎每个 Redis 命令都可以通过 node_redis 模块提供的客户端对象调用。

第66.3节：node_redis支持的一些更重要的操作

检查键的存在性

有时您可能需要检查某个键是否已存在并据此进行操作。为此，您可以使用 exists()。

and client.HMSET() are the same. Storing Lists

If you want to store a list of items, you can use Redis lists. To store a list use the following syntax:

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err, reply) {
  console.log(reply); //prints 2
});
```

The above snippet creates a list called frameworks and pushes two elements to it. So, the length of the list is now two. As you can see I have passed an args array to rpush. The first item of the array represents the name of the key while the rest represent the elements of the list. You can also use lpush() instead of rpush() to push the elements to the left.

To retrieve the elements of the list you can use the lrange() function as following:

```
client.lrange('frameworks', 0, -1, function(err, reply) {
  console.log(reply); // ['angularjs', 'backbone']
});
```

Just note that you get all the elements of the list by passing -1 as the third argument to lrange(). If you want a subset of the list, you should pass the end index here.

Storing Sets

Sets are similar to lists, but the difference is that they don't allow duplicates. So, if you don't want any duplicate elements in your list you can use a set. Here is how we can modify our previous snippet to use a set instead of list.

```
client.sadd(['tags', 'angularjs', 'backbonejs', 'emberjs'], function(err, reply) {
  console.log(reply); // 3
});
```

As you can see, the sadd() function creates a new set with the specified elements. Here, the length of the set is three. To retrieve the members of the set, use the smembers() function as following:

```
client.smembers('tags', function(err, reply) {
  console.log(reply);
});
```

This snippet will retrieve all the members of the set. Just note that the order is not preserved while retrieving the members.

This was a list of the most important data structures found in every Redis powered app. Apart from strings, lists, sets, and hashes, you can store sorted sets, hyperLogLogs, and more in Redis. If you want a complete list of commands and data structures, visit the official Redis documentation. Remember that almost every Redis command is exposed on the client object offered by the node_redis module.

Section 66.3: Some more important operations supported by node_redis

Checking the Existence of Keys

Sometimes you may need to check if a key already exists and proceed accordingly. To do so you can use exists()

函数如下所示：

```
client.exists('key', function(err, reply) {  
  if (reply === 1) {  
    console.log('存在');  
  } else {  
    console.log('不存在');  
  }  
});
```

删除和过期密钥

有时您需要清除某些键并重新初始化它们。要清除键，可以使用如下所示的 del 命令：

```
client.del('frameworks', function(err, reply) {  
  console.log(reply);  
});
```

您也可以为现有的键设置过期时间，方法如下：

```
client.set('key1', 'val1');  
client.expire('key1', 30);
```

上面的代码片段为键 key1 设置了 30 秒的过期时间。

递增和递减

Redis 也支持对键进行递增和递减。要递增一个键，可以使用 incr() 函数，如下所示：

```
client.set('key1', 10, function() {  
  client.incr('key1', function(err, reply) {  
    console.log(reply); // 11  
  });  
});
```

incr() 函数将键的值递增 1。如果需要递增不同的数值，可以使用 incrby() 函数。同样，要递减一个键，可以使用 decr() 和 decrby() 函数。

function as shown below:

```
client.exists('key', function(err, reply) {  
  if (reply === 1) {  
    console.log('exists');  
  } else {  
    console.log('doesn\'t exist');  
  }  
});
```

Deleting and Expiring Keys

At times you will need to clear some keys and reinitialize them. To clear the keys, you can use del command as shown below:

```
client.del('frameworks', function(err, reply) {  
  console.log(reply);  
});
```

You can also give an expiration time to an existing key as following:

```
client.set('key1', 'val1');  
client.expire('key1', 30);
```

The above snippet assigns an expiration time of 30 seconds to the key key1.

Incrementing and Decrementing

Redis also supports incrementing and decrementing keys. To increment a key use incr() function as shown below:

```
client.set('key1', 10, function() {  
  client.incr('key1', function(err, reply) {  
    console.log(reply); // 11  
  });  
});
```

The incr() function increments a key value by 1. If you need to increment by a different amount, you can use incrby() function. Similarly, to decrement a key you can use the functions like decr() and decrby().

第 67 章：使用 Browserify 解决浏览器中的 'required' 错误

第 67.1 节：示例 - file.js

在这个例子中，我们有一个名为file.js的文件。

假设你需要使用JavaScript和NodeJS的querystring模块来解析一个URL。

为此，你只需在文件中插入以下语句：

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

这段代码在做什么？

首先，我们创建了一个querystring模块，它提供了解析和格式化URL查询字符串的工具。它可以通过以下方式访问：

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

然后，我们使用.parse()方法解析URL。它将URL查询字符串 (str) 解析为键值对的集合。

例如，查询字符串'foo=bar&abc=xyz&abc=123'被解析为：

```
{ foo: 'bar', abc: ['xyz', '123'] }
```

不幸的是，浏览器没有定义require方法，但Node.js有。

安装 Browserify

使用 Browserify，你可以像在 Node 中使用 *require* 一样编写代码。那么，你如何解决这个问题呢？很简单。

1.首先安装 Node，它自带 npm。然后执行：

```
npm install -g browserify
```

2. 切换到包含你的 file.js 文件的目录，并使用 npm 安装我们的 querystring 模块：

```
npm install querystring
```

注意：如果你没有切换到指定目录，命令会失败，因为找不到包含该模块的文件。

3.现在使用 browserify 命令，从 file.js 开始递归地将所有所需模块打包成一个名为 bundle.js (或你喜欢的其他名字) 的单个文件：

```
browserify file.js -o bundle.js
```

Chapter 67: Using Browserify to resolve 'required' error with browsers

Section 67.1: Example - file.js

In this example we have a file called **file.js**.

Let's assume that you have to parse an URL using JavaScript and NodeJS querystring module.

To accomplish this all you have to do is to insert the following statement in your file:

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

What is this snippet doing?

Well, first, we create a querystring module which provides utilities for parsing and formatting URL query strings. It can be accessed using:

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

Then, we parse a URL using the .parse() method. It parses a URL query string (str) into a collection of key and value pairs.

For example, the query string '`foo=bar&abc=xyz&abc=123`' is parsed into:

```
{ foo: 'bar', abc: ['xyz', '123'] }
```

Unfortunately, Browsers don't have the *require* method defined, but Node.js does.

Install Browserify

With Browserify you can write code that uses *require* in the same way that you would use it in Node. So, how do you solve this? It's simple.

1. First install node, which ships with npm. Then do:

```
npm install -g browserify
```

2. Change into the directory in which your file.js is and Install our querystring module with npm:

```
npm install querystring
```

Note: If you don't change in the specific directory the command will fail because it can't find the file which contains the module.

3. Now recursively bundle up all the required modules starting at file.js into a single file called bundle.js (or whatever you like to name it) with the **browserify command**:

```
browserify file.js -o bundle.js
```

4. 最后将一个标签放入你的 HTML 中，完成！

```
<script src="bundle.js"></script>
```

发生的情况是，你会得到一个结合了旧的 .js 文件（即 file.js）和新创建的 bundle.js 文件的组合。这两个文件被合并成一个单一的文件。

重要

请记住，如果你想对 file.js 进行任何更改，并且不会影响程序的行为
你的更改只有在编辑新创建的 bundle.js 时才会生效

这是什么意思？

这意味着如果你出于任何原因想编辑 file.js，改动不会产生任何效果。你必须真正编辑 bundle.js，因为它是 bundle.js 和 file.js 的合并文件。

4. FinallyDrop a single tag into your html and you're done!

```
<script src="bundle.js"></script>
```

What happens is that you get a combination of your old .js file (**file.js** that is) and your newly created **bundle.js** file. Those two files are merged into one single file.

Important

Please keep in mind that if you want to make any changes to your file.js and will not affect the behaviour of your program. **Your changes will only take effect if you edit the newly created bundle.js**

What does that mean?

This means that if you want to edit **file.js** for any reasons, the changes will not have any effects. You really have to edit **bundle.js** since it is a merge of **bundle.js** and **file.js**.

第68章：Node.JS 和 MongoDB。

第68.1节：连接到数据库

要从 Node 应用程序连接到 Mongo 数据库，我们需要 mongoose。

安装 Mongoose，进入应用程序根目录并通过以下命令安装 mongoose

```
npm install mongoose
```

接下来我们连接到数据库。

```
var mongoose = require('mongoose');

//连接到运行在本地主机默认 mongod 端口的测试数据库
mongoose.connect('mongodb://localhost/test');
```

```
//使用自定义凭据连接
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');
```

```
//使用连接池大小来定义打开的连接数
//你也可以使用回调函数进行错误处理
mongoose.connect('mongodb://localhost:27017/consumers',
  {server: { poolSize: 50 }},
  function(err) {
    if(err) {
      console.log('此处出错')
      console.log(err);
      // 进行相应的错误处理
    } else {
      console.log('已连接到数据库');
    }
});
```

第68.2节：创建新集合

在Mongoose中，一切都源自于Schema。让我们创建一个Schema。

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var AutoSchema = new Schema({
  name : String,
  countOf: Number,
});
// 定义文档结构

// 默认情况下，数据库中创建的集合名称是我们使用的第一参数（或其复数形式）
module.exports = mongoose.model('Auto', AutoSchema);

// 我们可以通过在第三个参数中指定集合名称来覆盖默认名称。
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');
```

Chapter 68: Node.JS and MongoDB.

Section 68.1: Connecting To a Database

To connect to a mongo database from node application we require mongoose.

Installing Mongoose Go to the root of your application and install mongoose by

```
npm install mongoose
```

Next we connect to the database.

```
var mongoose = require('mongoose');

//connect to the test database running on default mongod port of localhost
mongoose.connect('mongodb://localhost/test');
```

```
//Connecting with custom credentials
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');
```

```
//Using Pool Size to define the number of connections opening
//Also you can use a call back function for error handling
mongoose.connect('mongodb://localhost:27017/consumers',
  {server: { poolSize: 50 }},
  function(err) {
    if(err) {
      console.log('error in this')
      console.log(err);
      // Do whatever to handle the error
    } else {
      console.log('Connected to the database');
    }
});
```

Section 68.2: Creating New Collection

With Mongoose, everything is derived from a Schema. Lets create a schema.

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var AutoSchema = new Schema({
  name : String,
  countOf: Number,
});
// defining the document structure

// by default the collection created in the db would be the first parameter we use (or the plural of it)
module.exports = mongoose.model('Auto', AutoSchema);

// we can over write it and define the collection name by specifying that in the third parameters.
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');
```

```
// 我们也可以在模型中定义方法。
AutoSchema.methods.speak = function () {
  var greeting = this.name
  ? "你好，我是 " + this.name + "，我有 " + this.countOf
  : "我没有名字";
  console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

请记住，方法必须在使用 `mongoose.model()` 编译之前添加到 schema 中，如上所示。

第68.3节：插入文档

要向集合中插入新文档，我们需要创建一个 schema 的对象。

```
var Auto = require('models/auto')
var autoObj = new Auto({
  name: "NewName",
  countOf: 10
});
```

我们像下面这样保存它

```
autoObj.save(function(err, insertedAuto) {
  if (err) return console.error(err);
  insertedAuto.speak();
  // 输出: Hello this is NewName and I have counts of 10
});
```

这将在集合中插入一个新文档

第68.4节：读取

从集合中读取数据非常简单。获取集合中的所有数据。

```
var Auto = require('models/auto')
Auto.find({}, function (err, autos) {
  if (err) return console.error(err);
  // 将返回集合中所有文档的json数组
  console.log(autos);
})
```

带条件读取数据

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
  if (err) return console.error(err);
  // 将返回集合中所有 count 大于等于 5 的文档的 JSON 数组
  5
  console.log(autos);
})
```

你也可以指定第二个参数为你需要的所有字段的对象

```
Auto.find({}, {name:1}, function (err, autos) {
  if (err) return console.error(err);
  // 将返回集合中所有文档的 name 字段的 JSON 数组
  console.log(autos);
```

```
// We can also define methods in the models.
AutoSchema.methods.speak = function () {
  var greeting = this.name
  ? "Hello this is " + this.name + " and I have counts of " + this.countOf
  : "I don't have a name";
  console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

Remember methods must be added to the schema before compiling it with `mongoose.model()` like done above ..

Section 68.3: Inserting Documents

For inserting a new document in the collection, we create a object of the schema.

```
var Auto = require('models/auto')
var autoObj = new Auto({
  name: "NewName",
  countOf: 10
});
```

We save it like the following

```
autoObj.save(function(err, insertedAuto) {
  if (err) return console.error(err);
  insertedAuto.speak();
  // output: Hello this is NewName and I have counts of 10
});
```

This will insert a new document in the collection

Section 68.4: Reading

Reading Data from the collection is very easy. Getting all data of the collection.

```
var Auto = require('models/auto')
Auto.find({}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of all the documents in the collection
  console.log(autos);
})
```

Reading data with a condition

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of all the documents in the collection whose count is greater than
  5
  console.log(autos);
})
```

You can also specify the second parameter as object of what all fields you need

```
Auto.find({}, {name:1}, function (err, autos) {
  if (err) return console.error(err);
  // will return a json array of name field of all the documents in the collection
  console.log(autos);
```

```
)
```

在集合中查找一个文档。

```
Auto.findOne({name:"newName"}, function (err, auto) {
  if (err) return console.error(err);
  // 将返回第一个 name 为 "newName" 的文档对象
  console.log(auto);
})
```

通过id在集合中查找一个文档。

```
Auto.findById(123, function (err, auto) {
  if (err) return console.error(err);
  // 将返回 id 为 123 的文档的第一个 json 对象
  console.log(auto);
})
```

第68.5节：更新

对于更新集合和文档，我们可以使用以下任意方法：

方法

- update()
- updateOne()
- updateMany()
- replaceOne()

Update()

`update()` 方法修改一个或多个文档（更新参数）

```
db.lights.update(
  { room: "卧室" },
  { status: "开" }
)
```

此操作在 'lights' 集合中搜索 `room` 为 卧室（第一个参数）的文档。然后将匹配文档的 `status` 属性更新为 开（第二个参数），并返回一个 `WriteResult` 对象，其内容如下：

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

UpdateOne

`UpdateOne()` 方法修改一个文档（更新参数）

```
db.countries.update(
  { country: "瑞典" },
  { capital: "斯德哥尔摩" }
)
```

此操作在 'countries' 集合中搜索 `country` 为 瑞典（第一个参数）的文档。然后将匹配文档的 `capital` 属性更新为 斯德哥尔摩（第二个参数），并返回一个 `WriteResult` 对象

```
)
```

Finding one document in a collection.

```
Auto.findOne({name:"newName"}, function (err, auto) {
  if (err) return console.error(err);
  // will return the first object of the document whose name is "newName"
  console.log(auto);
})
```

Finding one document in a collection by id .

```
Auto.findById(123, function (err, auto) {
  if (err) return console.error(err);
  // will return the first json object of the document whose id is 123
  console.log(auto);
})
```

Section 68.5: Updating

For updating collections and documents we can use any of these methods:

Methods

- update()
- updateOne()
- updateMany()
- replaceOne()

Update()

The `update()` method modifies one or many documents (update parameters)

```
db.lights.update(
  { room: "Bedroom" },
  { status: "On" }
)
```

This operation searches the 'lights' collection for a document where `room` is **Bedroom** (1st parameter). It then updates the matching documents `status` property to **On** (2nd parameter) and returns a `WriteResult` object that looks like this:

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

UpdateOne

The `UpdateOne()` method modifies ONE document (update parameters)

```
db.countries.update(
  { country: "Sweden" },
  { capital: "Stockholm" }
)
```

This operation searches the 'countries' collection for a document where `country` is **Sweden** (1st parameter). It then updates the matching documents `capital` property to **Stockholm** (2nd parameter) and returns a `WriteResult` object

其内容如下：

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

UpdateMany

UpdateMany() 方法修改多个文档 (更新参数)

```
db.food.updateMany(  
  { sold: { $lt: 10 } },  
  { $set: { sold: 55 } }  
)
```

此操作更新所有文档 (在“food”集合中) , 其中 sold 小于 10 (第一个参数) , 通过将 sold 设置为 55。然后返回一个类似如下的 WriteResult 对象：

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

a = 匹配的文档数量

b = 修改的文档数量

ReplaceOne

替换第一个匹配的文档 (替换文档)

这个名为countries的示例集合包含3个文档：

```
{ "_id" : 1, "country" : "瑞典" }  
{ "_id" : 2, "country" : "挪威" }  
{ "_id" : 3, "country" : "西班牙" }
```

下面的操作将文档{ country: "西班牙" }替换为文档{ country: "芬兰" }

```
db.countries.replaceOne(  
  { country: "西班牙" },  
  { country: "芬兰" }  
)
```

并返回：

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

示例集合 countries 现在包含：

```
{ "_id" : 1, "country" : "瑞典" }  
{ "_id" : 2, "country" : "挪威" }  
{ "_id" : 3, "country" : "芬兰" }
```

第68.6节：删除

在mongoose中删除集合中的文档的方式如下。

```
Auto.remove({_id:123}, function(err, result){  
  if (err) return console.error(err);
```

that looks like this:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

UpdateMany

The UpdateMany() method modifies multible documents (update parameters)

```
db.food.updateMany(  
  { sold: { $lt: 10 } },  
  { $set: { sold: 55 } }  
)
```

This operation updates all documents (in a 'food' collection) where sold is **lesser than 10** *(1st parameter) by setting sold to **55**. It then returns a WriteResult object that looks like this:

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

a = Number of matched documents

b = Number of modified documents

ReplaceOne

Replaces the first matching document (replacement document)

This example collection called **countries** contains 3 documents:

```
{ "_id" : 1, "country" : "Sweden" }  
{ "_id" : 2, "country" : "Norway" }  
{ "_id" : 3, "country" : "Spain" }
```

The following operation replaces the document { country: "Spain" } with document { country: "Finland" }

```
db.countries.replaceOne(  
  { country: "Spain" },  
  { country: "Finland" }  
)
```

And returns:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

The example collection **countries** now contains:

```
{ "_id" : 1, "country" : "Sweden" }  
{ "_id" : 2, "country" : "Norway" }  
{ "_id" : 3, "country" : "Finland" }
```

Section 68.6: Deleting

Deleting documents from a collection in mongoose is done in the following manner.

```
Auto.remove({_id:123}, function(err, result){  
  if (err) return console.error(err);
```

```
console.log(result); // 这将显示mongo默认的删除结果。  
});
```

```
console.log(result); // this will specify the mongo default delete result.  
});
```

belindoc.com

第69章：Passport集成

第69.1节：本地认证

passport-local 模块用于实现本地身份验证。

该模块允许您在 Node.js 应用程序中使用用户名和密码进行身份验证。

注册用户：

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// 使用命名策略，因为使用了两个本地策略：
// 一个用于注册，另一个用于登录
passport.use('localSignup', new LocalStrategy({
    // 覆盖默认的预期参数，
    // 即 'username' 和 'password'
    usernameField: 'email',
    passwordField: 'password',
    passReqToCallback: true // 允许我们将整个请求传回回调函数。
},
function(req, email, password, next) {
    // 检查数据库中用户是否已注册
    findUserByEmail(email, function(user) {
        // 如果邮箱已存在，终止注册流程并
        // 向回调函数传递 'false'
        if (user) return next(null, false);
        // 否则，我们创建用户
        else {
            // 密码必须进行哈希处理！
            let newUser = createUser(email, password);

            newUser.save(function() {
                // 将用户传递给回调函数
                return next(null, newUser);
            });
        });
    });
});
```

用户登录：

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
},
function(email, password, next) {
    // 查找用户
    findUserByEmail(email, function(user) {
        // 如果未找到用户，终止登录过程
        // 可以在验证回调中提供自定义消息
        // 以便向用户提供有关认证失败的更多细节
        if (!user)
```

Chapter 69: Passport integration

Section 69.1: Local authentication

The **passport-local** module is used to implement a local authentication.

This module lets you authenticate using a username and password in your Node.js applications.

Registering the user :

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// A named strategy is used since two local strategy are used :
// one for the registration and the other to sign-in
passport.use('localSignup', new LocalStrategy({
    // Overriding defaults expected parameters,
    // which are 'username' and 'password'
    usernameField: 'email',
    passwordField: 'password',
    passReqToCallback: true // allows us to pass back the entire request to the callback .
},
function(req, email, password, next) {
    // Check in database if user is already registered
    findUserByEmail(email, function(user) {
        // If email already exists, abort registration process and
        // pass 'false' to the callback
        if (user) return next(null, false);
        // Else, we create the user
        else {
            // Password must be hashed !
            let newUser = createUser(email, password);

            newUser.save(function() {
                // Pass the user to the callback
                return next(null, newUser);
            });
        }
    });
});
```

Logging in the user :

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
},
function(email, password, next) {
    // Find the user
    findUserByEmail(email, function(user) {
        // If user is not found, abort signing in process
        // Custom messages can be provided in the verify callback
        // to give the user more details concerning the failed authentication
        if (!user)
```

```

        return next(null, false, {message: '该电子邮件地址未关联任何
账户。'});
    // 否则，我们检查密码是否有效
    else {
        // 如果密码不正确，终止登录过程
        if (!isValidPassword(password)) return next(null, false);
        // 否则，将用户传递给回调函数
        else return next(null, user);
    }
});
```

创建路由：

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// 登录路由
// Passport 策略是中间件
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
});

// 注册路由
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
});

// 调用 req.logout() 以登出
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);
```

第69.2节：入门

Passport 必须使用 `passport.initialize()` 中间件进行初始化。要使用登录会话，`passport.session()` 中间件是必需的。

注意必须定义 `passport.serialize()` 和 `passport.deserializeUser()` 方法。**Passport** 将对用户实例进行序列化和反序列化，以便在会话中存取

```

const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// 需要读取cookie
app.use(cookieParser());

passport.serializeUser(function(user, next) {
    // 在会话中序列化用户
    next(null, user);
});
```

```

        return next(null, false, {message: 'This e-mail address is not associated with any
account.'});
    // Else, we check if password is valid
    else {
        // If password is not correct, abort signing in process
        if (!isValidPassword(password)) return next(null, false);
        // Else, pass the user to callback
        else return next(null, user);
    }
});
```

Creating routes :

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Sign-in route
// Passport strategies are middlewares
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
});

// Sign-up route
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
});

// Call req.logout() to log out
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);
```

Section 69.2: Getting started

Passport must be initialized using `passport.initialize()` middleware. To use login sessions, `passport.session()` middleware is required.

Note that `passport.serialize()` and `passport.deserializeUser()` methods must be defined. **Passport** will serialize and deserialize user instances to and from the session

```

const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Required to read cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
    // Serialize the user in the session
    next(null, user);
});
```

```

passport.deserializeUser(function(user, next) {
    // 使用之前序列化的用户
    next(null, user);
});

// 配置 express-session 中间件
app.use(session({
    secret: '蛋糕是谎言',
    resave: true,
    saveUninitialized: true
}));

// 初始化 passport
app.use(passport.initialize());
app.use(passport.session());

// 在端口3000启动express服务器
app.listen(3000);

```

第69.3节：Facebook认证

使用passport-facebook模块来实现Facebook认证。在此示例中，如果用户在登录时不存在，则会创建该用户。

实现策略：

```

const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// 策略默认命名为 'facebook'
passport.use({
    clientID: 'yourclientid',
    clientSecret: 'yourclientsecret',
    callbackURL: '/auth/facebook/callback'
},
// Facebook将发送令牌和用户资料
function(token, refreshToken, profile, next) {
    // 在数据库中检查用户是否已注册
    findUserByFacebookId(profile.id, function(user) {
        // 如果用户存在，返回其数据到回调
        if (user) return next(null, user);
        // 否则，创建该用户
        else {
            let newUser = createUserFromFacebook(profile, token);

            newUser.save(function() {
                // 将用户传递给回调函数
                return next(null, newUser);
            });
        }
    });
});

```

创建路由：

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// 认证路由

```

```

passport.deserializeUser(function(user, next) {
    // Use the previously serialized user
    next(null, user);
});

// Configuring express-session middleware
app.use(session({
    secret: 'The cake is a lie',
    resave: true,
    saveUninitialized: true
}));

// Initializing passport
app.use(passport.initialize());
app.use(passport.session());

// Starting express server on port 3000
app.listen(3000);

```

Section 69.3: Facebook authentication

The **passport-facebook** module is used to implement a **Facebook** authentication. In this example, if the user does not exist on sign-in, he is created.

Implementing strategy :

```

const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Strategy is named 'facebook' by default
passport.use({
    clientID: 'yourclientid',
    clientSecret: 'yourclientsecret',
    callbackURL: '/auth/facebook/callback'
},
// Facebook will send a token and user's profile
function(token, refreshToken, profile, next) {
    // Check in database if user is already registered
    findUserByFacebookId(profile.id, function(user) {
        // If user exists, returns his data to callback
        if (user) return next(null, user);
        // Else, we create the user
        else {
            let newUser = createUserFromFacebook(profile, token);

            newUser.save(function() {
                // Pass the user to the callback
                return next(null, newUser);
            });
        }
    });
});

```

Creating routes :

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Authentication route

```

```

app.get('/auth/facebook', passport.authenticate('facebook', {
    // 请求 Facebook 更多权限
    scope : 'email'
}));

// Facebook 认证用户后调用
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect : '/me',
        failureRedirect : '/'
    });

//...
app.listen(3000);

```

第69.4节：简单用户名-密码认证

在你的 routes/index.js 中

这里 user 是 userSchema 的模型

```

router.post('/login', function(req, res, next) {
    if (!req.body.username || !req.body.password) {
        return res.status(400).json({
            message: '请填写所有字段'
        });
    }

    passport.authenticate('local', function(err, user, info) {
        if (err) {
            console.log("错误：" + err);
            return next(err);
        }

        if(user) {
            console.log("用户已存在！")
            //可以通过 user.x 访问用户的所有数据
            res.json({ "success" : true});
            return;
        } else {
            res.json({ "success" : false});
            console.log("错误" + errorResponse());
            return;
        }
    })(req, res, next);
});

```

第69.5节：Google Passport认证

我们在npm中有一个简单的模块用于Google认证，名为**passport-google-oauth20**

考虑以下示例，在此示例中创建了一个名为config的文件夹，里面有passport.js和google.js文件，位于根目录。在你的app.js中包含以下内容

```

var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // passport文件所在路径

```

```

app.get('/auth/facebook', passport.authenticate('facebook', {
    // Ask Facebook for more permissions
    scope : 'email'
}));

// Called after Facebook has authenticated the user
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect : '/me',
        failureRedirect : '/'
    });

//...
app.listen(3000);

```

Section 69.4: Simple Username-Password Authentication

In your routes/index.js

Here user is the model for the userSchema

```

router.post('/login', function(req, res, next) {
    if (!req.body.username || !req.body.password) {
        return res.status(400).json({
            message: 'Please fill out all fields'
        });
    }

    passport.authenticate('local', function(err, user, info) {
        if (err) {
            console.log("ERROR：" + err);
            return next(err);
        }

        if(user) {
            console.log("User Exists!");
            //All the data of the user can be accessed by user.x
            res.json({ "success" : true});
            return;
        } else {
            res.json({ "success" : false});
            console.log("Error" + errorResponse());
            return;
        }
    })(req, res, next);
});

```

Section 69.5: Google Passport authentication

We have simple module available in npm for google authentication name **passport-google-oauth20**

Consider the following example In this example have created a folder namely config having the passport.js and google.js file in the root directory. In your app.js include the following

```

var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // path where the passport file placed

```

```
var app = express();
passport(app);
```

// 其他初始化服务器、错误处理的代码

在 config 文件夹中的 passport.js 文件中包含以下代码

```
var passport = require ('passport'),
google = require('./google'),
User = require('../model/user'); // User 是 mongoose 模型

module.exports = function(app){
app.use(passport.initialize());
app.use(passport.session());
passport.serializeUser(function(user, done){
    done(null, user);
});
passport.deserializeUser(function (user, done) {
    done(null, user);
});
google();
};
```

在同一 config 文件夹中的 google.js 文件中包含以下内容

```
var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,
User = require('../model/user');
module.exports = function () {
passport.use(new GoogleStrategy({
    clientID: 'CLIENT ID',
    clientSecret: 'CLIENT SECRET',
    callbackURL: "http://localhost:3000/auth/google/callback"
},
function(accessToken, refreshToken, profile, cb) {
    User.findOne({ googleId : profile.id }, function (err, user) {
        if(err){
            return cb(err, false, {message : err});
        }else {
            if (user != '' && user != null) {
                return cb(null, user, {message : "User "});
            } else {
                var username = profile.displayName.split(' ');
                var userData = new User({
name : profile.displayName,
username : username[0],
password : username[0],
facebookId : '',
googleId : profile.id,
});
                // 向用户发送邮件，以防需要将新创建的
                // 凭证发送给用户，方便将来无需使用谷歌登录即可登录
                userData.save(function (err, newuser) {
                    if (err) {
                        return cb(null, false, {message : err + " !!! 请重试"});
                    }else{
                        return cb(null, newuser);
                    }
                });
            }
        }
    });
});
```

```
var app = express();
passport(app);
```

// other code to initialize the server , error handle

In the passport.js file in the config folder include the following code

```
var passport = require ('passport'),
google = require('./google'),
User = require('../model/user'); // User is the mongoose model

module.exports = function(app){
app.use(passport.initialize());
app.use(passport.session());
passport.serializeUser(function(user, done){
    done(null, user);
});
passport.deserializeUser(function (user, done) {
    done(null, user);
});
google();
};
```

In the google.js file in the same config folder include following

```
var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,
User = require('../model/user');
module.exports = function () {
passport.use(new GoogleStrategy({
    clientID: 'CLIENT ID',
    clientSecret: 'CLIENT SECRET',
    callbackURL: "http://localhost:3000/auth/google/callback"
},
function(accessToken, refreshToken, profile, cb) {
    User.findOne({ googleId : profile.id }, function (err, user) {
        if(err){
            return cb(err, false, {message : err});
        }else {
            if (user != '' && user != null) {
                return cb(null, user, {message : "User "});
            } else {
                var username = profile.displayName.split(' ');
                var userData = new User({
name : profile.displayName,
username : username[0],
password : username[0],
facebookId : '',
googleId : profile.id,
});
                // send email to user just in case required to send the newly created
                // credentials to user for future login without using google login
                userData.save(function (err, newuser) {
                    if (err) {
                        return cb(null, false, {message : err + " !!! Please try again"});
                    }else{
                        return cb(null, newuser);
                    }
                });
            }
        }
    });
});
```

```
    });
}
));
};
```

在此示例中，如果用户不在数据库中，则使用用户模型中的字段名 `googleId` 在数据库中创建一个新用户以供本地引用。

```
    });
}
});
};
```

Here in this example, if user is not in DB then creating a new user in DB for local reference using the field name `googleId` in user model.

belindoc.com

第70章：依赖注入

第70.1节：为什么使用依赖注入

1. 快速开发流程
2. 解耦
3. 单元测试编写

快速开发流程

使用依赖注入时，开发人员可以加快开发进程，因为在依赖注入之后，代码冲突更少，且更容易管理所有模块。

解耦

模块之间的耦合度降低，维护起来更容易。

单元测试编写

硬编码的依赖可以传入模块，从而方便为每个模块编写单元测试。

Chapter 70: Dependency Injection

Section 70.1: Why Use Dependency Injection

1. **Fast Development process**
2. **Decoupling**
3. **Unit test writing**

Fast Development process

When using dependency injection node developer can faster their development process because after DI there is less code conflict and easy to manage all module.

Decoupling

Modules becomes less couple then it is easy to maintain.

Unit test writing

Hardcoded dependencies can pass them into the module then easy to write unit test for each module.

第71章：NodeJS初学者指南

第71.1节：Hello World！

将以下代码放入名为helloworld.js的文件中

```
console.log("Hello World");
```

保存文件，并通过Node.js执行：

```
node helloworld.js
```

belindoc.com

Chapter 71: NodeJS Beginner Guide

Section 71.1: Hello World！

Place the following code into a file name `helloworld.js`

```
console.log("Hello World");
```

Save the file, and execute it through Node.js:

```
node helloworld.js
```

第72章：Node.js的使用案例

第72.1节：HTTP服务器

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

console.log('启动服务器...');

var config = {
  port: 80,
  contentType: 'application/json; charset=utf-8'
};

// 端口80上的JSON-API服务器

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
  if (err.code == 'EADDRINUSE') console.error('端口 ' + config.port + ' 已被占用');
  else console.error(err.message);
});

server.on('request', (request, res) => {
  var remoteAddress = request.headers['x-forwarded-for'] || request.connection.remoteAddress; // 客户端地址
  console.log(remoteAddress + ' ' + request.method + ' ' + request.url);

  var out = {};
  // 这里你可以根据 `request.url` 修改输出
  out.test = request.url;
  res.writeHead(200, {
    'Content-Type': config.contentType
  });
  res.end(JSON.stringify(out));
});

server.on('listening', () => {
  c.info('服务器可用：http://localhost:' + config.port);
});
```

第72.2节：带命令提示符的控制台

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('这里正在发生一些长时间的操作...');

var cliConfig = {
  promptPrefix: ' > '
}

/*
命令识别
开始
*/
var commands = {
  eval: function(arg) { // 尝试在控制台输入：eval 2 * 10 ^ 3 + 2 ^ 4
    arg = arg.join(' ');
    try { console.log(eval(arg)); }
    catch (e) { console.log(e); }
  },
  exit: function(arg) {
```

Chapter 72: Use Cases of Node.js

Section 72.1: HTTP server

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

console.log('Starting server...');

var config = {
  port: 80,
  contentType: 'application/json; charset=utf-8'
};

// JSON-API server on port 80

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
  if (err.code == 'EADDRINUSE') console.error('Port ' + config.port + ' is already in use');
  else console.error(err.message);
});

server.on('request', (request, res) => {
  var remoteAddress = request.headers['x-forwarded-for'] || request.connection.remoteAddress; // Client address
  console.log(remoteAddress + ' ' + request.method + ' ' + request.url);

  var out = {};
  // Here you can change output according to `request.url`
  out.test = request.url;
  res.writeHead(200, {
    'Content-Type': config.contentType
  });
  res.end(JSON.stringify(out));
});

server.on('listening', () => {
  c.info('Server is available: http://localhost:' + config.port);
});
```

Section 72.2: Console with command prompt

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('Something long is happening here...');

var cliConfig = {
  promptPrefix: ' > '
}

/*
Commands recognition
BEGIN
*/
var commands = {
  eval: function(arg) { // Try typing in console: eval 2 * 10 ^ 3 + 2 ^ 4
    arg = arg.join(' ');
    try { console.log(eval(arg)); }
    catch (e) { console.log(e); }
  },
  exit: function(arg) {
```

```

process.exit();
}
};

rl.on('line', (str) => {
  rl.pause();
  var arg = str.trim().match(/([^\"]+)/|"(?:[^"\\"|\\]|\\.)+"/g); // 应用正则表达式去除除双引号内以外的所有空格：

http://stackoverflow.com/a/14540319/2396907
  if (arg) {
    for (let n in arg) {
      arg[n] = arg[n].replace(/^\\"/|\\"$/g, '');
    }
    var commandName = arg[0];
    var command = commands[commandName];
    if (command) {
      arg.shift();
      command(arg);
    }
    else console.log('Command "' + commandName + '" doesn\'t exist');
  }
  rl.prompt();
});
/*
结束
命令识别
*/
rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();

```

belindoc.com

```

process.exit();
}
};

rl.on('line', (str) => {
  rl.pause();
  var arg = str.trim().match(/([^\"]+)/|"(?:[^"\\"|\\]|\\.)+"/g); // Applying regular expression for
removing all spaces except for what between double quotes:
http://stackoverflow.com/a/14540319/2396907
  if (arg) {
    for (let n in arg) {
      arg[n] = arg[n].replace(/^\\"/|\\"$/g, '');
    }
    var commandName = arg[0];
    var command = commands[commandName];
    if (command) {
      arg.shift();
      command(arg);
    }
    else console.log('Command "' + commandName + '" doesn\'t exist');
  }
  rl.prompt();
});
/*
END OF
Commands recognition
*/
rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();

```

第73章：Sequelize.js

第73.1节：定义模型

在sequelize中定义模型有两种方式；使用sequelize.define(...), 或sequelize.import(...). 两个函数都返回一个sequelize模型对象。

1. sequelize.define(modelName, attributes, [options])

如果你想将所有模型定义在一个文件中，或者想对模型定义有更多控制权，这就是推荐的做法。

```
/* 初始化 Sequelize */
const config = {
  username: "数据库用户名",
  password: "数据库密码",
  database: "数据库名称",
  host: "数据库主机地址",
  dialect: "mysql" // 其他选项有 postgres、sqlite、mariadb 和 mssql。
}
var Sequelize = require("sequelize");
var sequelize = new Sequelize(config);

/* 定义模型 */
sequelize.define("我的模型", {
  name: Sequelize.STRING,
  comment: Sequelize.TEXT,
  date: {
    type: Sequelize.DATE,
    allowNull: false
  }
});
```

有关文档和更多示例，请查看doclets文档，或sequelize.com的文档。

2. sequelize.import(path)

如果你的模型定义拆分到每个文件中，那么import是你的好帮手。在初始化Sequelize的文件中，你需要这样调用import：

```
/* 初始化 Sequelize */
// 查看前面的代码片段了解初始化

/* 定义模型 */
sequelize.import("./models/my_model.js"); // 路径可以是相对路径或绝对路径
```

然后在你的模型定义文件中，代码大致如下：

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define("我的模型", {
    name: DataTypes.STRING,
    comment: DataTypes.TEXT,
    date: {
      type: DataTypes.DATE,
      allowNull: false
    }
  });
};
```

Chapter 73: Sequelize.js

Section 73.1: Defining Models

There are two ways to define models in sequelize; with sequelize.`define`(...), or sequelize.`import`(...). Both functions return a sequelize model object.

1. sequelize.define(modelName, attributes, [options])

This is the way to go if you'd like to define all your models in one file, or if you want to have extra control of your model definition.

```
/* Initialize Sequelize */
const config = {
  username: "database username",
  password: "database password",
  database: "database name",
  host: "database's host URL",
  dialect: "mysql" // Other options are postgres, sqlite, mariadb and mssql.
}
var Sequelize = require("sequelize");
var sequelize = new Sequelize(config);

/* Define Models */
sequelize.define("MyModel", {
  name: Sequelize.STRING,
  comment: Sequelize.TEXT,
  date: {
    type: Sequelize.DATE,
    allowNull: false
  }
});
```

For the documentation and more examples, check out the [doclets documentation](#), or [sequelize.com's documentation](#).

2. sequelize.import(path)

If your model definitions are broken into a file for each, then `import` is your friend. In the file where you initialize Sequelize, you need to call import like so:

```
/* Initialize Sequelize */
// Check previous code snippet for initialization

/* Define Models */
sequelize.import("./models/my_model.js"); // The path could be relative or absolute
```

Then in your model definition files, your code will look something like this:

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define("MyModel", {
    name: DataTypes.STRING,
    comment: DataTypes.TEXT,
    date: {
      type: DataTypes.DATE,
      allowNull: false
    }
  });
};
```

```
});  
};
```

有关如何使用import的更多信息，请查看GitHub上sequelize的express示例。

第73.2节：安装

请确保您已先安装Node.js和npm。然后使用npm安装sequelize.js

```
npm install --save sequelize
```

您还需要安装支持的数据库Node.js模块。您只需安装您正在使用的那个

对于MySQL和Mariadb

```
npm install --save mysql
```

对于PostgreSQL

```
npm install --save pg pg-hstore
```

对于SQLite

```
npm install --save sqlite
```

用于MSSQL

```
npm install --save tedious
```

安装完成后，您可以像下面这样引入并创建一个新的 Sequelize 实例。

ES5 语法

```
var Sequelize = require('sequelize');  
var sequelize = new Sequelize('database', 'username', 'password');
```

ES6 stage-0 Babel 语法

```
import Sequelize from 'sequelize';  
const sequelize = new Sequelize('database', 'username', 'password');
```

您现在有一个可用的sequelize实例。如果您愿意，也可以将其命名为其他名称，例如

```
var db = new Sequelize('database', 'username', 'password');
```

或者

```
var database = new Sequelize('database', 'username', 'password');
```

那部分由你决定。一旦安装完成，你就可以根据API文档在你的应用程序中使用它

文档地址 <http://docs.sequelizejs.com/en/v3/api/sequelize/>

安装后的下一步是设置你自己的模型

```
});  
};
```

For more information on how to use import, check out sequelize's express example on GitHub.

Section 73.2: Installation

Make sure that you first have Node.js and npm installed. Then install sequelize.js with npm

```
npm install --save sequelize
```

You will also need to install supported database Node.js modules. You only need to install the one you are using

For MySQL and Mariadb

```
npm install --save mysql
```

For PostgreSQL

```
npm install --save pg pg-hstore
```

For SQLite

```
npm install --save sqlite
```

For MSSQL

```
npm install --save tedious
```

Once you have you set up installed you can include and create a new Sequelize instance like so.

ES5 syntax

```
var Sequelize = require('sequelize');  
var sequelize = new Sequelize('database', 'username', 'password');
```

ES6 stage-0 Babel syntax

```
import Sequelize from 'sequelize';  
const sequelize = new Sequelize('database', 'username', 'password');
```

You now have an instance of sequelize available. You could if you so feel inclined call it a different name such as

```
var db = new Sequelize('database', 'username', 'password');
```

or

```
var database = new Sequelize('database', 'username', 'password');
```

that part is your prerogative. Once you have this installed you can use it inside of your application as per the API documentation <http://docs.sequelizejs.com/en/v3/api/sequelize/>

Your next step after install would be to [set up your own model](#)

第74章：PostgreSQL集成

第74.1节：连接到PostgreSQL

使用PostgreSQLnpm模块。

从npm安装依赖

```
npm install pg --save
```

现在你需要创建一个PostgreSQL连接，之后可以进行查询。

假设您的数据库名为 students，主机为 localhost，数据库用户为 postgres

```
var pg = require("pg");
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

第74.2节：使用连接对象查询

如果您想使用连接对象查询数据库，可以使用此示例代码。

```
var queryString = "SELECT name, age FROM students" ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
result.addRow(row);
});

query.on("end", function (result) {
//逻辑处理
});
```

Chapter 74: PostgreSQL integration

Section 74.1: Connect To PostgreSQL

Using PostgreSQLnpm module.

install dependency from npm

```
npm install pg --save
```

Now you have to create a PostgreSQL connection, which you can later query.

Assume you Database_Name = students, Host = localhost and DB_User= postgres

```
var pg = require("pg");
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

Section 74.2: Query with Connection Object

If you want to use connection object for query database you can use this sample code.

```
var queryString = "SELECT name, age FROM students" ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
result.addRow(row);
});

query.on("end", function (result) {
//LOGIC
});
```

第75章：MySQL集成

在本主题中，您将学习如何使用MySQL数据库管理工具与Node.js集成。您将学习使用Node.js程序和脚本连接并操作存储在MySQL中的数据的各种方法。

第75.1节：连接到MySQL

连接MySQL最简单的方法之一是使用`mysql`模块。该模块处理Node.js应用程序与MySQL服务器之间的连接。你可以像安装其他模块一样安装它：

```
npm install --save mysql
```

现在你需要创建一个mysql连接，之后可以用它来执行查询。

```
const mysql      = require('mysql');
const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'me',
  password   : 'secret',
  database  : 'database_schema'
});

connection.connect();

// 执行一些查询语句
// 例如 SELECT * FROM FOO

connection.end();
```

在下一个示例中，你将学习如何查询connection对象。

第75.2节：使用连接池

a. 同时运行多个查询

MySQL 连接中的所有查询都是一个接一个地执行。这意味着如果你想执行 10 个查询，每个查询需要 2 秒，那么完成整个执行将需要 20 秒。解决方案是创建 10 个连接，并在不同的连接中运行每个查询。这可以通过连接池自动完成。

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass',
  database        : 'schema'
});

for(var i=0;i<10;i++){
  pool.query('SELECT ` as example', function(err, rows, fields) {
    if (err) throw err;
    console.log(rows[0].example); //显示 1
  });
}
```

它将并行运行所有 10 个查询。

Chapter 75: MySQL integration

In this topic you will learn how to integrate with Node.js using MySQL database management tool. You will learn various ways to connect and interact with data residing in mysql using a nodejs program and script.

Section 75.1: Connect to MySQL

One of the easiest ways to connect to MySQL is by using `mysql` module. This module handles the connection between Node.js app and MySQL server. You can install it like any other module:

```
npm install --save mysql
```

Now you have to create a mysql connection, which you can later query.

```
const mysql      = require('mysql');
const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'me',
  password   : 'secret',
  database  : 'database_schema'
});

connection.connect();

// Execute some query statements
// I.e. SELECT * FROM FOO

connection.end();
```

In the next example you will learn how to query the connection object.

Section 75.2: Using a connection pool

a. Running multiple queries at same time

All queries in MySQL connection are done one after another. It means that if you want to do 10 queries and each query takes 2 seconds then it will take 20 seconds to complete whole execution. The solution is to create 10 connection and run each query in a different connection. This can be done automatically using connection pool

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass',
  database        : 'schema'
});

for(var i=0;i<10;i++){
  pool.query('SELECT ` as example', function(err, rows, fields) {
    if (err) throw err;
    console.log(rows[0].example); //Show 1
  });
}
```

It will run all the 10 queries in parallel.

当你使用pool时，不再需要单独的连接。你可以直接对连接池进行查询。MySQL 模块会寻找下一个空闲连接来执行你的查询。

b. 在数据库服务器上实现多租户，不同的数据库托管在其上。

多租户是当今企业应用的常见需求，为数据库服务器中的每个数据库创建连接池并不推荐。因此，我们可以做的是为数据库服务器创建连接池，然后根据需求在数据库服务器上托管的不同数据库之间切换连接。

假设我们的应用程序在数据库服务器上为每个公司托管了不同的数据库。当用户访问应用程序时，我们将连接到相应的公司数据库。以下是如何实现的示例：

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

让我来分解这个示例：

在定义连接池配置时，我没有指定数据库名称，只指定了数据库服务器，即

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

所以当我们想要使用数据库服务器上的特定数据库时，我们通过以下方式请求连接访问数据库：

```
connection.changeUser({database : "firm1"});
```

你可以参考官方文档 [here](#)

第75.3节：使用参数查询连接对象

当你想在SQL中使用用户生成的内容时，需要使用参数。例如，要搜索名字为aminadav的用户，你应该这样做：

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
```

When you use pool you don't need the connection anymore. You can query directly the pool. MySQL module will search for the next free connection to execute your query.

b. Achieving multi-tenancy on database server with different databases hosted on it.

Multitenancy is a common requirement of enterprise application nowadays and creating connection pool for each database in database server is not recommended. so, what we can do instead is create connection pool with database server and then switch them between databases hosted on database server on demand.

Suppose our application has different databases for each firm hosted on database server. We will connect to respective firm database when user hits the application. Here is the example on how to do that:

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Let me break down the example:

When defining pool configuration i did not give the database name but only gave database server i.e

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

so when we want to use the specific database on database server, we ask the connection to hit database by using:

```
connection.changeUser({database : "firm1"});
```

you can refer the official documentation [here](#)

Section 75.3: Query a connection object with parameters

When you want to use user generated content in the SQL, it with done with parameters. For example for searching user with the name aminadav you should do:

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
```

```

rows.forEach(function(row) {
  console.log(row.name, 'email address is', row.email);
});
} else {
  console.log('没有结果。');
}
});

```

第75.4节：查询无参数的连接对象

你以字符串形式发送查询，并在响应回调中接收答案。回调会给你错误、行数组和字段。每一行包含返回表的所有列。以下是对接下来说明的代码片段。

```

connection.query('从 users 表中选择 name,email', function(err, rows, fields) {
  if (err) throw err;

  console.log('共有:', rows.length, '个用户');
  console.log('第一个用户名是:', rows[0].name)
});

```

第75.5节：使用连接池中的单个连接运行多个查询

可能会出现你已经设置了一个 MySQL 连接池，但你有多个查询想要顺序执行的情况：

```
SELECT 1; SELECT 2;
```

你可以像其他地方看到的那样使用`pool.query`直接运行它们，然而如果连接池中只有一个空闲连接，你必须等待连接可用后才能运行第二个查询。

不过，你可以从连接池中保留一个活动连接，并使用`pool.getConnection`通过单个连接运行任意数量的查询：

```

pool.getConnection(function (err, conn) { if (err) return callback(err); conn.query('SELECT 1 AS seq', function (err, rows) { if (err) throw err; conn.query('SELECT 2 AS seq', function (err, rows) { if (err) throw err; conn.release(); callback(); });}); });

```

注意：你必须记得释放连接，否则可用的 MySQL 连接数会减少，影响连接池中其他连接的使用！

有关 MySQL 连接池的更多信息，请查看 [MySQL 文档](#)。

第 75.6 节：导出连接池

```

// db.js

const mysql = require('mysql');

const pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
});

```

```

rows.forEach(function(row) {
  console.log(row.name, 'email address is', row.email);
});
} else {
  console.log('There were no results.');
}
});

```

Section 75.4: Query a connection object without parameters

You send the query as a string and in response callback with the answer is received. The callback gives you `error`, array of `rows` and `fields`. Each row contains all the column of the returned table. Here is a snippet for the following explanation.

```

connection.query('SELECT name,email from users', function(err, rows, fields) {
  if (err) throw err;

  console.log('There are:', rows.length, 'users');
  console.log('First user name is:', rows[0].name)
});

```

Section 75.5: Run a number of queries with a single connection from a pool

There may be situations where you have setup a pool of MySQL connections, but you have a number of queries you would like to run in sequence:

```
SELECT 1; SELECT 2;
```

You could just run them using `pool.query` as seen elsewhere, however if you only have one free connection in the pool you must wait until a connection becomes available before you can run the second query.

You can, however, retain an active connection from the pool and run as many queries as you would like using a single connection using `pool.getConnection`:

```

pool.getConnection(function (err, conn) { if (err) return callback(err); conn.query('SELECT 1 AS seq', function (err, rows) { if (err) throw err; conn.query('SELECT 2 AS seq', function (err, rows) { if (err) throw err; conn.release(); callback(); });}); });

```

Note: You must remember to `release` the connection, otherwise there is one less MySQL connection available to the rest of the pool!

For more information on pooling MySQL connections [check out the MySQL docs](#).

Section 75.6: Export Connection Pool

```

// db.js

const mysql = require('mysql');

const pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
});

```

```

});;

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}

// app.js

const db = require('./db');

db.getConnection((err, conn) => {
  conn.query('从某表中选择某些内容', (error, results, fields) => {
    // 获取结果
    conn.release();
  });
});

```

第75.7节：发生错误时返回查询

当发生错误时，您可以将执行的查询附加到您的 err 对象上：

```

var q = mysql.query('从 `pokedex` 中选择 `name`，条件是 `id` = ?', [ 25 ], function (err, result) {
  if (err) {
    // 表 `test.pokedex` 不存在
    err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
    callback(err);
  } else {
    callback(null, result);
  }
});

```

```

});;

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}

// app.js

const db = require('./db');

db.getConnection((err, conn) => {
  conn.query('SELECT something from sometable', (error, results, fields) => {
    // get the results
    conn.release();
  });
});

```

Section 75.7: Return the query when an error occurs

You can attach the query executed to your err object when an error occurs:

```

var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [ 25 ], function (err, result) {
  if (err) {
    // Table `test.pokedex` doesn't exist
    err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
    callback(err);
  } else {
    callback(null, result);
  }
});

```

第76章：MySQL连接池

第76.1节：无数据库连接池的使用

在数据库服务器上托管多个数据库，实现多租户功能。

多租户是现代企业应用的常见需求，不建议为数据库服务器上的每个数据库创建连接池。因此，我们可以创建一个连接池连接数据库服务器，然后根据需求在数据库服务器上切换托管的数据库。

假设我们的应用程序在数据库服务器上为每个公司托管不同的数据库。当用户访问应用时，我们将连接到相应的公司数据库。以下是如何实现的示例：

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

让我来分解这个示例：

在定义连接池配置时，我没有指定数据库名称，只指定了数据库服务器，即

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

所以当我们想要使用数据库服务器上的特定数据库时，我们通过以下方式请求连接访问数据库：

```
connection.changeUser({database : "firm1"});
```

你可以参考官方文档 [here](#)

Chapter 76: MySQL Connection Pool

Section 76.1: Using a connection pool without database

Achieving multitenancy on database server with multiple databases hosted on it.

Multitenancy is common requirement of enterprise application nowadays and creating connection pool for each database in database server is not recommended. so, what we can do instead is create connection pool with database server and then switch between databases hosted on database server on demand.

Suppose our application has different databases for each firm hosted on database server. We will connect to respective firm database when user hits the application. here is the example on how to do that:

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Let me break down the example:

When defining pool configuration i did not give the database name but only gave database server i.e

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

so when we want to use the specific database on database server, we ask the connection to hit database by using:

```
connection.changeUser({database : "firm1"});
```

you can refer the official documentation [here](#)

第77章：MSSQL集成

要将任何数据库与Node.js集成，您需要一个驱动程序包，或者称为npm模块，它将为您提供连接数据库和执行交互的基本API。MSSQL数据库也是如此，这里我们将MSSQL与Node.js集成，并对SQL表执行一些基本查询。

第77.1节：通过mssql npm模块连接SQL

我们将从创建一个具有基本结构的简单Node应用程序开始，然后连接本地SQL服务器数据库，并对该数据库执行一些查询。

步骤1： 创建一个名为您打算创建的项目的目录/文件夹。使用 `npm init` 命令初始化一个Node应用程序，该命令将在当前目录创建`package.json`文件。

```
mkdir mySqlApp  
//文件夹已创建  
cd mySqlApp  
//切换到新创建的目录  
npm init  
//回答所有问题 ..  
npm 安装  
//由于我们还没有向应用添加任何包，这将很快完成。
```

步骤 2: 现在我们将在此目录下创建一个 `App.js` 文件，并安装一些连接 SQL 数据库所需的包。

```
sudo gedit App.js  
//这将创建 App.js文件，你也可以使用你喜欢的文本编辑器 :)  
npm install --save mssql  
//这将安装 mssql 包到你的应用中
```

步骤 3: 现在我们将为应用添加一个基本的配置变量，`mssql` 模块将使用它来建立连接。

```
console.log("你好，世界，这是一个连接到 SQL 服务器的应用。");  
var config = {  
    "user": "myusername", // 默认是 sa  
    "password": "yourStrong(!)Password",  
    "server": "localhost", // 本地机器  
    "database": "staging", // 数据库名称  
    "options": {  
        "encrypt": true  
    }  
}  
  
sql.connect(config, err => {  
    if(err){  
        throw err;  
    }  
    console.log("连接成功！");  
  
    new sql.Request().query('select 1 as number', (err, result) => {  
        //处理错误  
        console.dir(result)  
        // 此示例使用回调策略获取结果。  
    })
```

Chapter 77: MSSQL Intergration

To integrate any database with nodejs you need a driver package or you can call it a npm module which will provide you with basic API to connect with the database and perform interactions . Same is true with mssql database , here we will integrate mssql with nodejs and perform some basic queries on SQL tables.

Section 77.1: Connecting with SQL via. mssql npm module

We will start with creating a simple node application with a basic structure and then connecting with local sql server database and performing some queries on that database.

Step 1: Create a directory/folder by the name of project which you intent to create. Initialize a node application using `npm init` command which will create a `package.json` in current directory .

```
mkdir mySqlApp  
//folder created  
cd mySqlApp  
//change to newly created directory  
npm init  
//answer all the question ..  
npm install  
//This will complete quickly since we have not added any packages to our app.
```

Step 2: Now we will create a `App.js` file in this directory and install some packages which we are going to need to connect to sql db.

```
sudo gedit App.js  
//This will create App.js file , you can use your fav. text editor :)  
npm install --save mssql  
//This will install the mssql package to you app
```

Step 3: Now we will add a basic configuration variable to our application which will be used by `mssql` module to establish a connection .

```
console.log("Hello world, This is an app to connect to sql server.");  
var config = {  
    "user": "myusername", //default is sa  
    "password": "yourStrong(!)Password",  
    "server": "localhost", // for local machine  
    "database": "staging", // name of database  
    "options": {  
        "encrypt": true  
    }  
}  
  
sql.connect(config, err => {  
    if(err){  
        throw err;  
    }  
    console.log("Connection Successful !");  
  
    new sql.Request().query('select 1 as number', (err, result) => {  
        //handle err  
        console.dir(result)  
        // This example uses callbacks strategy for getting results.  
    })
```

```
});  
  
sql.on('error', err => {  
    // ... 错误处理程序  
    console.log("Sql数据库连接错误 " ,err);  
})
```

步骤4：这是最简单的步骤，我们启动应用程序，应用程序将连接到SQL服务器并打印一些简单的结果。

```
node App.js  
// 输出：  
// 你好，世界，这是一个连接到SQL服务器的应用程序。  
// 连接成功！  
// 1
```

要使用Promise或异步执行查询，请参考mssql包的官方文档：

- [Promise](#)
- [异步/Await](#)

```
});  
  
sql.on('error', err => {  
    // ... error handler  
    console.log("Sql database connection error " ,err);  
})
```

Step 4: This is the easiest step ,where we start the application and the application will connect to the sql server and print out some simple results .

```
node App.js  
// Output :  
// Hello world, This is an app to connect to sql server.  
// Connection Successful !  
// 1
```

To use promises or async for query execution refer the official documents of the mssql package :

- [Promises](#)
- [Async/Await](#)

第78章：Node.js与Oracle

第78.1节：连接到Oracle数据库

连接到ORACLE数据库的一个非常简单的方法是使用`oracledb`模块。该模块处理你的Node.js应用程序与Oracle服务器之间的连接。你可以像安装其他模块一样安装它：

```
npm install oracledb
```

现在你需要创建一个ORACLE连接，之后可以进行查询。

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

oracledb.getConnection(
{
    user        : "oli",
    password    : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
},
connExecute
);
```

`connectString "ORACLE_DEV_DB_TNS_NAME"` 可能存在于同一目录下的 `tnsnames.org` 文件中，或者存在于你的 Oracle Instant Client 安装目录中。

如果你的开发机器上没有安装任何 Oracle Instant Client，可以按照你操作系统的[instant client 安装指南](#)进行安装。

第78.2节：使用本地模块简化查询

为了简化您从ORACLE-DB的查询，您可能想这样调用您的查询：

```
const oracle = require('./oracle.js');

const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
.then(function(result) {
    console.log(result.rows[0]['C2']);
})
.catch(function(err) {
    next(err);
});
```

建立连接并执行包含在此 `oracle.js` 文件中，内容如下：

```
'use strict';
const oracledb = require('oracledb');

const oracleDbRelease = function(conn) {
    conn.release(function (err) {
        if (err)
            console.log(err.message);
    });
}

function queryArray(sql, bindParams, options) {
```

Chapter 78: Node.js with Oracle

Section 78.1: Connect to Oracle DB

A very easy way to connect to an ORACLE database is by using `oracledb` module. This module handles the connection between your Node.js app and Oracle server. You can install it like any other module:

```
npm install oracledb
```

Now you have to create an ORACLE connection, which you can later query.

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

oracledb.getConnection(
{
    user        : "oli",
    password    : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
},
connExecute
);
```

The `connectString "ORACLE_DEV_DB_TNS_NAME"` may live in a `tnsnames.org` file in the same directory or where your oracle instant client is installed.

If you don't have any oracle instant client installed on your development machine you may follow the [instant client installation guide](#) for your operating system.

Section 78.2: Using a local module for easier querying

To simplify your querying from ORACLE-DB, you may want to call your query like this:

```
const oracle = require('./oracle.js');

const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
.then(function(result) {
    console.log(result.rows[0]['C2']);
})
.catch(function(err) {
    next(err);
});
```

Building up the connection and executing is included in this `oracle.js` file with content as follows:

```
'use strict';
const oracledb = require('oracledb');

const oracleDbRelease = function(conn) {
    conn.release(function (err) {
        if (err)
            console.log(err.message);
    });
}

function queryArray(sql, bindParams, options) {
```

```

options.isAutoCommit = false; // 我们只执行 SELECT 操作

return new Promise(function(resolve, reject) {
    oracledb.getConnection(
        {
            user : "oli",
            password : "password",
            connectString : "ORACLE_DEV_DB_TNA_NAME"
        })
    .然后(function(connection){
        //console.log("sql 日志: " + sql + " 参数 " + bindParams);
        connection.execute(sql, bindParams, options)
        .然后(function(results) {
            resolve(results);
            process.nextTick(function() {
                oracleDbRelease(connection);
            });
        })
    .捕获(function(err) {
        reject(err);
    });
});

process.nextTick(function() {
    oracleDbRelease(connection);
});
});

捕获(function(err) {
    reject(err);
});

function queryObject(sql, bindParams, options) {
    options['outFormat'] = oracledb.OBJECT; // 默认是 oracledb.ARRAY
    return queryArray(sql, bindParams, options);
}

module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;

```

请注意，您可以对您的 `oracle` 对象调用 `queryArray` 和 `queryObject` 两种方法。

第 78.3 节：查询无参数的连接对象

现在可以使用 `connExecute` 函数执行查询。您可以选择以对象或数组的形式获取查询结果。结果将打印到 `console.log`。

```

function connExecute(err, connection)
{
    if (err) {
        console.error(err.message);
        return;
    }
    sql = "select 'test' as c1, 'oracle' as c2 from dual";
    connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, // or oracledb.ARRAY
        function(err, result)
        {
            if (err) {
                console.error(err.message);
                connRelease(connection);
            }
        }
    );
}

```

```

options.isAutoCommit = false; // we only do SELECTs

return new Promise(function(resolve, reject) {
    oracledb.getConnection(
        {
            user : "oli",
            password : "password",
            connectString : "ORACLE_DEV_DB_TNA_NAME"
        })
    .then(function(connection){
        //console.log("sql log: " + sql + " params " + bindParams);
        connection.execute(sql, bindParams, options)
        .then(function(results) {
            resolve(results);
            process.nextTick(function() {
                oracleDbRelease(connection);
            });
        })
        .catch(function(err) {
            reject(err);
        });
    });
});

process.nextTick(function() {
    oracleDbRelease(connection);
});
});

.catch(function(err) {
    reject(err);
});

function queryObject(sql, bindParams, options) {
    options['outFormat'] = oracledb.OBJECT; // default is oracledb.ARRAY
    return queryArray(sql, bindParams, options);
}

module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;

```

Note that you have both methods `queryArray` and `queryObject` to call on your `oracle` object.

Section 78.3: Query a connection object without parameters

Use may now use the `connExecute`-Function for executing a query. You have the option to get the query result as an object or array. The result ist printed to `console.log`.

```

function connExecute(err, connection)
{
    if (err) {
        console.error(err.message);
        return;
    }
    sql = "select 'test' as c1, 'oracle' as c2 from dual";
    connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, // or oracledb.ARRAY
        function(err, result)
        {
            if (err) {
                console.error(err.message);
                connRelease(connection);
            }
        }
    );
}

```

```
    return;
}
console.log(result.metaData);
  console.log(result.rows);
  connRelease(connection);
});
}
```

由于我们使用了非连接池连接，因此必须再次释放连接。

```
function connRelease(connection)
{
connection.close(
  function(err) {
    if (err) {
      console.error(err.message);
    }
  });
}
```

对象的输出将是

```
[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]
```

数组的输出将是

```
[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]
```

```
    return;
}
console.log(result.metaData);
  console.log(result.rows);
  connRelease(connection);
});
}
```

Since we used a non-pooling connection, we have to release our connection again.

```
function connRelease(connection)
{
  connection.close(
    function(err) {
      if (err) {
        console.error(err.message);
      }
    });
}
```

The output for an object will be

```
[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]
```

and the output for an array will be

```
[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]
```

第79章：Node.js中的同步与异步编程

第79.1节：使用async

async包提供了用于异步代码的函数。

使用auto函数可以定义两个或多个函数之间的异步关系：

```
var async = require('async');

async.auto({
  get_data: function(callback) {
    console.log('获取数据中');
    // 异步代码获取一些数据
    callback(null, '数据', '转换为数组');
  },
  make_folder: function(callback) {
    console.log('创建文件夹中');
    // 异步代码创建一个目录以存储文件
    // 这与获取数据同时运行
    callback(null, '文件夹');
  },
  write_file: ['get_data', 'make_folder', function(results, callback) {
    console.log('写入文件中', JSON.stringify(results));
    // 一旦有数据且目录存在,
    // 将数据写入目录中的文件
    callback(null, 'filename');
  }],
  email_link: ['write_file', function(results, callback) {
    console.log('in email_link', JSON.stringify(results));
    // 一旦文件写入完成, 我们将发送一个链接到该文件的邮件...
    // results.write_file 包含 write_file 返回的文件名。
    callback(null, {'file':results.write_file, 'email':'user@example.com'});
  }]
}, function(err, results) {
  console.log('err = ', err);
  console.log('results = ', results);
});
```

这段代码本可以同步执行，只需按正确顺序调用get_data、make_folder、write_file和email_link。Async 会帮你跟踪结果，如果发生错误（callback的第一个参数不等于null），它会停止执行其他函数。

Chapter 79: Synchronous vs Asynchronous programming in nodejs

Section 79.1: Using async

The [async package](#) provides functions for asynchronous code.

Using the `auto` function you can define asynchronous relations between two or more functions:

```
var async = require('async');

async.auto({
  get_data: function(callback) {
    console.log('in get_data');
    // async code to get some data
    callback(null, 'data', 'converted to array');
  },
  make_folder: function(callback) {
    console.log('in make_folder');
    // async code to create a directory to store a file in
    // this is run at the same time as getting the data
    callback(null, 'folder');
  },
  write_file: ['get_data', 'make_folder', function(results, callback) {
    console.log('in write_file', JSON.stringify(results));
    // once there is some data and the directory exists,
    // write the data to a file in the directory
    callback(null, 'filename');
  }],
  email_link: ['write_file', function(results, callback) {
    console.log('in email_link', JSON.stringify(results));
    // once the file is written let's email a link to it...
    // results.write_file contains the filename returned by write_file.
    callback(null, {'file':results.write_file, 'email':'user@example.com'});
  }]
}, function(err, results) {
  console.log('err = ', err);
  console.log('results = ', results);
});
```

This code could have been made synchronously, by just calling the `get_data`, `make_folder`, `write_file` and `email_link` in the correct order. Async keeps track of the results for you, and if an error occurred (first parameter of `callback` unequal to `null`) it stops the execution of the other functions.

第80章：Node.js错误管理

我们将学习如何创建错误对象以及如何在Node.js中抛出和处理错误

未来将编辑与错误处理最佳实践相关的内容。

第80.1节：try...catch块

try...catch 块用于处理异常，记住异常指的是抛出的错误，而不是错误本身。

```
try {
    var a = 1;
    b++; //这会导致错误，因为 b 未定义
    console.log(b); //这行代码不会被执行
} catch (error) {
    console.log(error); //这里我们处理 try 块中引发的错误
}
```

在 try 块中，`b++` 会导致错误，该错误会传递到 catch 块，在那里可以处理该错误，或者在 catch 块中重新抛出相同的错误，或者稍作修改后再抛出。让我们看下一个例子。

```
try {
    var a = 1;
    b++;
    console.log(b);
} catch (error) {
    error.message = "b 变量未定义，因此无法对未定义的值进行自增"
    throw error;
}
```

在上述示例中，我们修改了 message 属性的 error 对象，然后抛出了修改后的 error。

你可以在 try 块中抛出任何错误，并在 catch 块中处理它：

```
try {
    var a = 1;
    throw new Error("Some error message");
    console.log(a); //这行代码不会被执行
} catch (error) {
    console.log(error); //将是上面抛出的错误
}
```

第 80.2 节：创建 Error 对象

new Error(message)

创建新的错误对象，其中值 message 被设置为创建对象的 message 属性。通常 message 参数作为字符串传递给 Error 构造函数。然而，如果 message 参数是对象而非字符串，则 Error 构造函数会调用传入对象的 `toString()` 方法，并将该值设置为创建的错误对象的 message 属性。

```
var err = new Error("错误信息");
console.log(err.message); //打印：错误信息
console.log(err);
//输出
//Error: 错误信息
```

Chapter 80: Node.js Error Management

We will learn how to create Error objects and how to throw & handle errors in Node.js

Future edits related to best practices in error handling.

Section 80.1: try...catch block

try...catch block is for handling exceptions, remember exception means the thrown error not the error.

```
try {
    var a = 1;
    b++; //this will cause an error because be is undefined
    console.log(b); //this line will not be executed
} catch (error) {
    console.log(error); //here we handle the error caused in the try block
}
```

In the `try` block `b++` cause an error and that error passed to `catch` block which can be handled there or even can be thrown the same error in catch block or make little bit modification then throw. Let's see next example.

```
try {
    var a = 1;
    b++;
    console.log(b);
} catch (error) {
    error.message = "b variable is undefined, so the undefined can't be incremented"
    throw error;
}
```

In the above example we modified the message property of error object and then throw the modified error.

You can throw any error in your try block and handle it in the catch block:

```
try {
    var a = 1;
    throw new Error("Some error message");
    console.log(a); //this line will not be executed
} catch (error) {
    console.log(error); //will be the above thrown error
}
```

Section 80.2: Creating Error object

new Error(message)

Creates new error object, where the value message is being set to message property of the created object. Usually the message arguments are being passed to Error constructor as a string. However if the message argument is object not a string then Error constructor calls `.toString()` method of the passed object and sets that value to message property of the created error object.

```
var err = new Error("The error message");
console.log(err.message); //prints: The error message
console.log(err);
//output
//Error: The error message
```

```
// at ...
```

每个错误对象都有堆栈跟踪。堆栈跟踪包含错误信息，并显示错误发生的位置（上面的输出显示了错误堆栈）。一旦错误对象被创建，系统会捕获当前行错误的堆栈跟踪。要获取堆栈跟踪，请使用任何已创建错误对象的 `stack` 属性。下面两行是相同的：

```
console.log(err);
console.log(err.stack);
```

第80.3节：抛出错误

抛出错误意味着异常，如果任何异常未被处理，则 Node 服务器将崩溃。

下面这行代码会抛出错误：

```
throw new Error("Some error occurred");
```

或者

```
var err = new Error("Some error occurred");
throw err;
```

或者

```
throw "Some error occurred";
```

最后一个示例（抛出字符串）不是良好做法，不推荐使用（应始终抛出 `Error` 对象的实例）。

请注意，如果你在代码中 `throw` 一个错误，那么系统将在该行崩溃（如果没有异常处理程序），该行之后的任何代码都不被执行。

```
var a = 5;
var err = new Error("Some error message");
throw err; //这将打印错误堆栈并且 Node 服务器将停止
a++; //这行代码永远不会被执行
console.log(a); //这行也不会被执行
```

但是在这个例子中：

```
var a = 5;
var err = new Error("Some error message");
console.log(err); //这将打印错误堆栈
a++;
console.log(a); //这行代码将被执行并打印 6
```

```
// at ...
```

Each error object has stack trace. Stack trace contains the information of error message and shows where the error happened (the above output shows the error stack). Once error object is created the system captures the stack trace of the error on current line. To get the stack trace use `stack` property of any created error object. Below two lines are identical:

```
console.log(err);
console.log(err.stack);
```

Section 80.3: Throwing Error

Throwing error means exception if any exception is not handled then the node server will crash.

The following line throws error:

```
throw new Error("Some error occurred");
```

or

```
var err = new Error("Some error occurred");
throw err;
```

or

```
throw "Some error occurred";
```

The last example (throwing strings) is not good practice and is not recommended (always throw errors which are instances of `Error` object).

Note that if you `throw` an error in your, then the system will crash on that line (if there is no exception handlers), no any code will be executed after that line.

```
var a = 5;
var err = new Error("Some error message");
throw err; //this will print the error stack and node server will stop
a++; //this line will never be executed
console.log(a); //and this one also
```

But in this example:

```
var a = 5;
var err = new Error("Some error message");
console.log(err); //this will print the error stack
a++;
console.log(a); //this line will be executed and will print 6
```

第81章：Node.js v6 新特性和改进

随着 Node 6 成为 Node 的新 LTS 版本，我们可以看到通过引入新的 ES6 标准，语言有了许多改进。我们将介绍一些新特性及其实现示例。

第81.1节：函数默认参数

```
function addTwo(a, b = 2) {  
    return a + b;  
}  
  
addTwo(3) // 返回结果 5
```

随着默认函数参数的加入，你现在可以使参数变为可选，并让它们默认取你选择的值。

第81.2节：剩余参数

```
function argumentLength(...args) {  
    return args.length;  
}  
  
argumentLength(5) // 返回 1  
argumentLength(5, 3) // 返回 2  
argumentLength(5, 3, 6) // 返回 3
```

通过在函数的最后一个参数前加上...，传入函数的所有参数都会被读取为一个数组。在这个例子中，我们传入多个参数，并获取由这些参数创建的数组的长度。

第81.3节：箭头函数

箭头函数是 ECMAScript 6 中定义函数的新方式。

```
// 传统的声明和定义函数的方式  
var sum = function(a,b)  
{  
    return a+b;  
}  
  
// 箭头函数  
let sum = (a, b)=> a+b;  
  
// 使用多行定义函数  
let checkIfEven = (a) => {  
    if( a % 2 == 0 )  
        return true;  
    else  
        return false;  
}
```

Chapter 81: Node.js v6 New Features and Improvement

With node 6 becoming the new LTS version of node. We can see an number of improvements to the language through the new ES6 standards introduced. We'll be walking through some of the new features introduced and examples of how to implement them.

Section 81.1: Default Function Parameters

```
function addTwo(a, b = 2) {  
    return a + b;  
}  
  
addTwo(3) // Returns the result 5
```

With the addition of default function parameters you can now make arguments optional and have them default to a value of your choice.

Section 81.2: Rest Parameters

```
function argumentLength(...args) {  
    return args.length;  
}  
  
argumentLength(5) // returns 1  
argumentLength(5, 3) //returns 2  
argumentLength(5, 3, 6) //returns 3
```

By prefacing the last argument of your function with ... all arguments passed to the function are read as an array. In this example we get pass in multiple arguments and get the length of the array created from those arguments.

Section 81.3: Arrow Functions

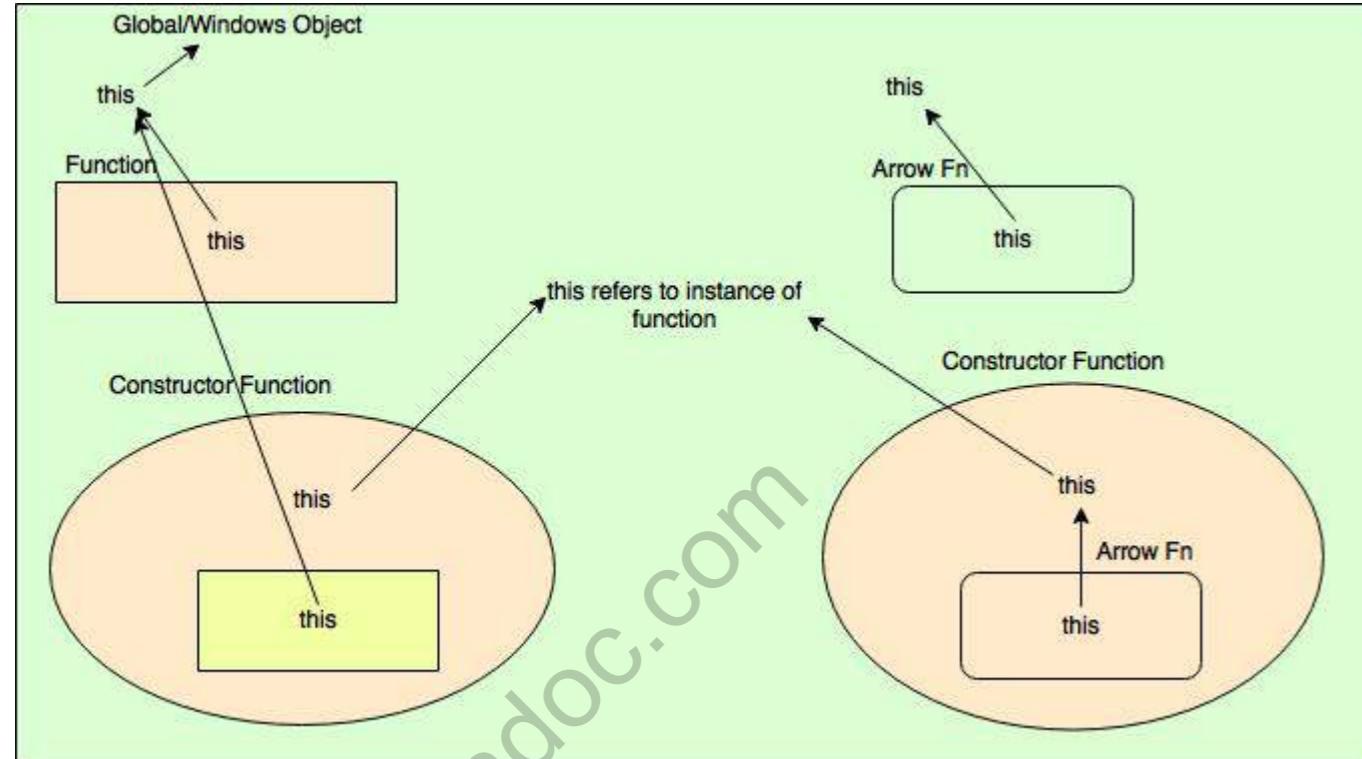
Arrow function is the new way of defining a function in ECMAScript 6.

```
// traditional way of declaring and defining function  
var sum = function(a,b)  
{  
    return a+b;  
}  
  
// Arrow Function  
let sum = (a, b)=> a+b;  
  
//Function defination using multiple lines  
let checkIfEven = (a) => {  
    if( a % 2 == 0 )  
        return true;  
    else  
        return false;  
}
```

第81.4节：“this”在箭头函数中的含义

函数中的>this指的是调用该函数的实例对象，但箭头函数中的>this等同于定义该箭头函数的函数的>this。

让我们通过图示来理解



通过示例来理解。

```
var normalFn = function(){
  console.log(this) // 指向全局/window对象。
}

var arrowFn = () => console.log(this); // 作为函数在全局/窗口对象作用域中定义，指向 window 或全局对象

var service = {

  constructorFn : function(){

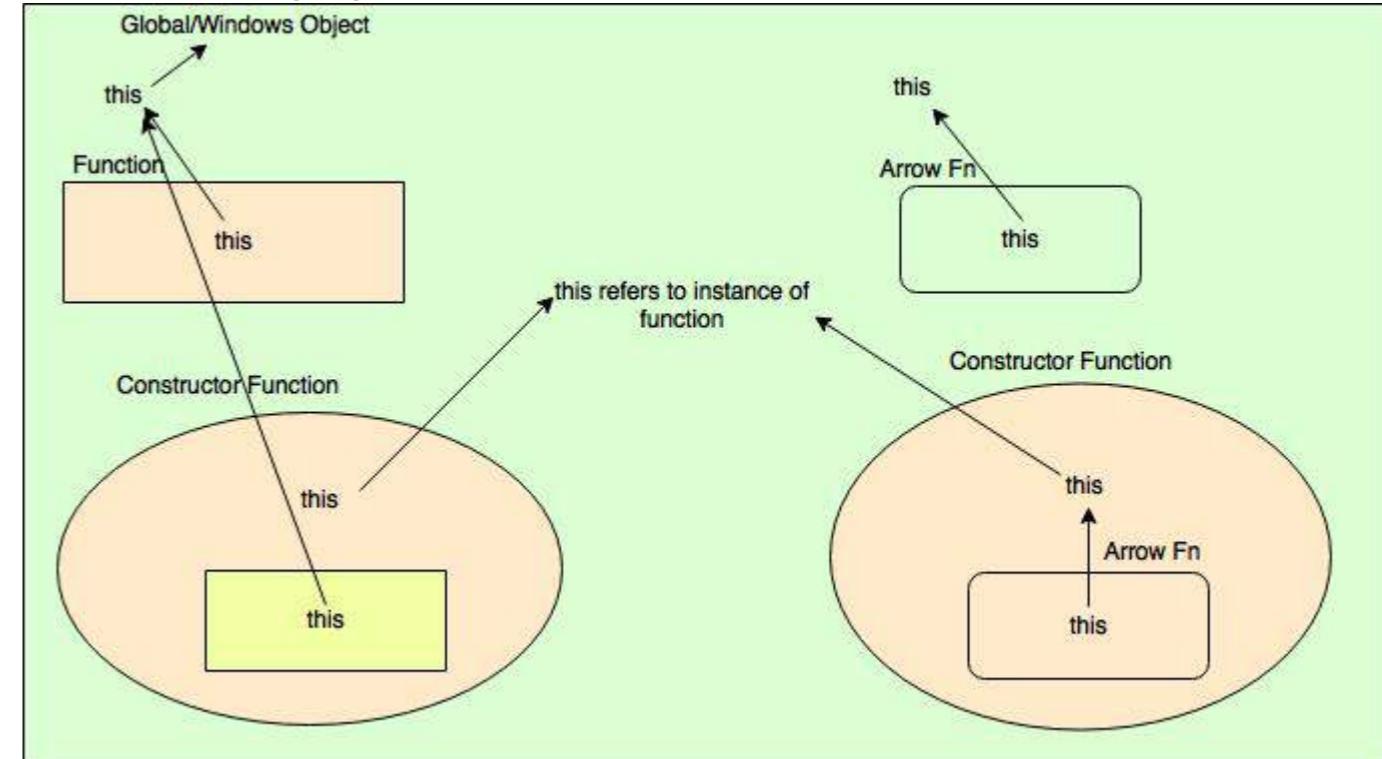
    console.log(this); // 指向 service，因为 service 对象用于调用该方法。

    var nestedFn = function(){
      console.log(this); // 指向 window 或全局对象，因为调用该方法时未使用实例对象。
    }
    nestedFn();
  },
  arrowFn : function(){
    console.log(this); // 指向 service，因为 service 对象用于调用该方法。
    let fn = () => console.log(this); // 指向 service 对象，因为箭头函数定义在通过实例对象调用的函数中。
    fn();
  }
}
```

Section 81.4: "this" in Arrow Function

this in function refers to instance object used to call that function but this in arrow function is equal to this of function in which arrow function is defined.

Let's understand using diagram



Understanding using examples.

```
var normalFn = function(){
  console.log(this) // refers to global/window object.
}

var arrowFn = () => console.log(this); // refers to window or global object as function is defined
in scope of global/window object

var service = {

  constructorFn : function(){

    console.log(this); // refers to service as service object used to call method.

    var nestedFn = function(){
      console.log(this); // refers window or global object because no instance object was used
      to call this method.
    }
    nestedFn();
  },
  arrowFn : function(){
    console.log(this); // refers to service as service object was used to call method.
    let fn = () => console.log(this); // refers to service object as arrow function defined in
    function which is called using instance object.
    fn();
  }
}
```

```
// 调用已定义的函数  
constructorFn();  
arrowFn();  
service.constructorFn();  
service.arrowFn();
```

在箭头函数中，this 是词法作用域，即箭头函数定义所在的函数作用域。

第一个例子是传统的函数定义方式，因此，this 指向global/window对象。

在第二个例子中，this 用在箭头函数内，因此this 指向定义它的作用域（即

windows 或 global 对象）。在第三个例子中，this 是 service 对象，因为调用函数的是 service 对象。

在第四个例子中，箭头函数定义并调用于作用域为service的函数内，因此它打印 service对象。

注意：- 在 Node.js 中打印的是全局对象，在浏览器中打印的是 window 对象。

第81.5节：扩展运算符

```
function myFunction(x, y, z) { }  
var args = [0, 1, 2];  
myFunction(...args);
```

扩展语法允许在需要多个参数（用于函数调用）、多个元素（用于数组字面量）或多个变量的地方展开一个表达式。就像剩余参数一样，只需在你的数组前加上...

```
// calling defined functions  
constructorFn();  
arrowFn();  
service.constructorFn();  
service.arrowFn();
```

In arrow function, this is lexical scope which is the scope of function where arrow function is defined.

The first example is the traditional way of defining functions and hence, this refers to global/window object.

In the second example this is used inside arrow function hence this refers to the scope where it is defined(which is windows or global object). In the third example this is service object as service object is used to call the function.

In fourth example, arrow function is defined and called from the function whose scope is service, hence it prints service object.

Note: - global object is printed in Node.js and windows object in browser.

Section 81.5: Spread Operator

```
function myFunction(x, y, z) { }  
var args = [0, 1, 2];  
myFunction(...args);
```

The spread syntax allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) or multiple variables are expected. Just like the rest parameters simply preface your array with ...

第82章：事件循环

在这篇文章中，我们将讨论事件循环的概念是如何产生的，以及它如何被用于高性能服务器和事件驱动的应用程序，如图形用户界面（GUI）。

第82.1节：事件循环概念的演变

事件循环的伪代码

事件循环是一个等待事件发生然后对这些事件作出反应的循环

```
while true:  
    等待某事发生  
    对发生的事情作出反应
```

无事件循环的单线程HTTP服务器示例

```
while true:  
    socket = 等待下一个 TCP 连接 从 (socket) 读取 HTTP  
    P 请求头  
  
    将 (file_contents) 写入 (socket)  
    关闭(socket)
```

这是一个简单的 HTTP 服务器示例，它是单线程的，但没有事件循环。这里的问题是它会等待每个请求完成后才开始处理下一个请求。如果读取 HTTP

请求头或从磁盘获取文件需要一段时间，我们应该能够在等待

完成的同时开始处理下一个请求。

最常见的解决方案是让程序支持多线程。

无事件循环的多线程 HTTP 服务器示例

```
function 处理连接(socket):  
    从 (socket) 读取 HTTP 请求头  
        file_contents = 从磁盘获取请求的文件  
        将 HTTP 响应头写入 (socket)  
        将 (file_contents) 写入 (socket)  
        关闭(socket)  
    while true:  
        socket = 等待下一个 TCP 连接  
        创建一个 新 线程执行 处理连接(socket)
```

现在我们已经让这个小型 HTTP 服务器支持多线程。这样，我们可以立即开始处理下一个请求，因为当前请求在后台线程中运行。包括 Apache 在内的许多服务器都采用这种方法。

但这并不完美。一个限制是你只能创建有限数量的线程。对于连接数量巨大，但每个连接只需偶尔关注的工作负载，多线程模型的性能不会很好。针对这些情况的解决方案是使用事件循环：

HTTP 服务器事件循环示例

```
while true:  
    event = 等待下一个事件发生  
        if (event.type == 新的_TCP_连接):  
            conn = 新建 连接  
            conn.socket = event.socket
```

Chapter 82: Eventloop

In this post we are going to discuss how the concept of Eventloop emerged and how it can be used for high performance servers and event driven applications like GUIs.

Section 82.1: How the concept of event loop evolved

Eventloop in pseudo code

An event loop is a loop that waits for events and then reacts to those events

```
while true:  
    wait for something to happen  
    react to whatever happened
```

Example of a single-threaded HTTP server with no event loop

```
while true:  
    socket = wait for the next TCP connection  
    read the HTTP request headers from (socket)  
    file_contents = fetch the requested file from disk  
    write the HTTP response headers to (socket)  
    write the (file_contents) to (socket)  
    close(socket)
```

Here's a simple form of a HTTP server which is a single threaded but no event loop. The problem here is that it waits until each request is finished before starting to process the next one. If it takes a while to read the HTTP request headers or to fetch the file from disk, we should be able to start processing the next request while we wait for that to finish.

The most common solution is to make the program multi-threaded.

Example of a multi-threaded HTTP server with no event loop

```
function handle_connection(socket):  
    read the HTTP request headers from (socket)  
    file_contents = fetch the requested file from disk  
    write the HTTP response headers to (socket)  
    write the (file_contents) to (socket)  
    close(socket)  
while true:  
    socket = wait for the next TCP connection  
    spawn a new thread doing handle_connection(socket)
```

Now we have made our little HTTP server multi threaded. This way, we can immediately move on to the next request because the current request is running in a background thread. Many servers, including Apache, use this approach.

But it's not perfect. One limitation is that you can only spawn so many threads. For workloads where you have a huge number of connections, but each connection only requires attention every once in a while, the multi-threaded model won't perform very well. The solution for those cases is to use an event loop:

Example of a HTTP server with event loop

```
while true:  
    event = wait for the next event to happen  
        if (event.type == NEW_TCP_CONNECTION):  
            conn = new Connection  
            conn.socket = event.socket
```

```

开始从 (conn.socket) 读取 HTTP 请求头, 用户数据 = (conn)
else if (event.type == 从套接字读取完成):
conn = event.userdata
开始从磁盘获取请求的文件, 用户数据 = (conn)
else if (event.type == 从磁盘读取完成):
conn = event.userdata
conn.file_contents = 我们从磁盘获取的数据
conn.current_state = "writing headers"
开始向(conn.socket)写入HTTP响应头, 用户数据= (conn)
否则如果(event.type == FINISHED_WRITING_TO_SOCKET):
conn = event.userdata
如果(conn.current_state == "writing headers"):
conn.current_state = "writing file contents"
开始向(conn.socket)写入数据到(conn.socket), 用户数据= (conn)
否则如果(conn.current_state == "writing file contents"):
关闭(conn.socket)

```

希望这个伪代码是可以理解的。事情的流程是这样的：我们等待事件发生。每当有新的连接创建或现有连接需要我们处理时，我们就去处理它，然后继续等待。这样，当有许多连接且每个连接很少需要关注时，我们的性能表现会很好。

在Linux上运行的真实应用程序（非伪代码）中，“等待下一个事件发生”部分会通过调用poll()或epoll()系统调用来实现。“开始读取/写入套接字数据”部分会通过以非阻塞模式调用recv()或send()系统调用来实现。

参考文献：

[1]. “事件循环是如何工作的？” [在线]. 可用链接：<https://www.quora.com/How-does-an-event-loop-work>

```

start reading HTTP request headers from (conn.socket) with userdata = (conn)
else if (event.type == FINISHED_READING_FROM_SOCKET):
conn = event.userdata
start fetching the requested file from disk with userdata = (conn)
else if (event.type == FINISHED_READING_FROM_DISK):
conn = event.userdata
conn.file_contents = the data we fetched from disk
conn.current_state = "writing headers"
start writing the HTTP response headers to (conn.socket) with userdata = (conn)
else if (event.type == FINISHED_WRITING_TO_SOCKET):
conn = event.userdata
if (conn.current_state == "writing headers"):
conn.current_state = "writing file contents"
start writing (conn.file_contents) to (conn.socket) with userdata = (conn)
else if (conn.current_state == "writing file contents"):
close(conn.socket)

```

Hopefully this pseudocode is intelligible. Here's what's going on: We wait for things to happen. Whenever a new connection is created or an existing connection needs our attention, we go deal with it, then go back to waiting. That way, we perform well when there are many connections and each one only rarely requires attention.

In a real application (not pseudocode) running on Linux, the "wait for the next event to happen" part would be implemented by calling the poll() or epoll() system call. The "start reading/writing something to a socket" parts would be implemented by calling the recv() or send() system calls in non-blocking mode.

Reference:

[1]. "How does an event loop work?" [Online]. Available : <https://www.quora.com/How-does-an-event-loop-work>

第83章：Node.js历史

这里我们将讨论Node.js的历史、版本信息及其当前状态。

第83.1节：每年的关键事件

2009

- 3月3日：项目命名为“node”10月1日：[npm \(Node包管理器\) 的第一个非常早期预览版](#)
- 11月8日：[Ryan Dahl \(Node.js创始人\) 在2009年JSConf上的原始Node.js演讲](#)

2010

- Express：一个Node.js网页开发框架
- Socket.io初始版本发布
- 4月28日：[Heroku上实验性Node.js支持](#)
- 7月28日：[Ryan Dahl在Google的Node.js技术演讲](#)
- 8月20日：[Node.js 0.2.0版本发布](#)

2011

- 3月31日：Node.js 指南
- 5月1日：[npm 1.0 发布](#)
- 5月1日：[Ryan Dahl 在 Reddit 上的AMA \(问我任何事\)](#)
- 7月10日：[《Node 初学者指南》，Node.js 入门书籍，完成。](#)
 - 一份面向初学者的全面 Node.js 教程。
- 8月16日：[LinkedIn 使用 Node.js](#)
 - LinkedIn 推出了全新改版的移动应用，带来了新功能和全新的底层架构。
- 10月5日：[Ryan Dahl 讲述 Node.js 的历史及其创作原因](#)
- 12月5日：[Uber 生产环境中使用 Node.js](#)
 - Uber 工程经理 Curtis Chambers 解释了为什么公司完全重新设计了他们的应用，采用 Node.js 以提高效率并改善合作伙伴和客户体验。

2012

- 1月30日：[Node.js 创始人瑞安·达尔 \(Ryan Dahl\) 退出 Node.js 的日常管理](#)
- 6月25日：[Node.js v0.8.0 \[稳定版\] 发布](#)
- 12月20日：[Node.js 框架 Hapi 发布](#)

2013

- 4月30日：[MEAN 技术栈：MongoDB、ExpressJS、AngularJS 和 Node.js](#)
- 5月17日：[我们如何构建 eBay 的第一个 Node.js 应用](#)
- 11月15日：[PayPal 发布 Kraken，一个 Node.js 框架](#)
- 11月22日：[沃尔玛发生 Node.js 内存泄漏事件](#)
 - 沃尔玛实验室的埃兰·哈默 (Eran Hammer) 向 Node.js 核心团队反映了他追踪了数月的内存泄漏问题。
- 12月19日：[Koa - Node.js 的 Web 框架](#)

2014

- 1月15日：[TJ Fontaine 接管 Node 项目](#)

Chapter 83: Nodejs History

Here we are going to discuss about the history of Node.js, version information and it's current status.

Section 83.1: Key events in each year

2009

- 3rd March : [The project was named as "node"](#)
- 1st October : [First very early preview of npm, the Node package manager](#)
- 8th November : [Ryan Dahl's \(Creator of Node.js\) Original Node.js Talk at JSConf 2009](#)

2010

- Express: A Node.js web development framework
- Socket.io initial release
- 28th April : [Experimental Node.js Support on Heroku](#)
- 28th July : [Ryan Dahl's Google Tech Talk on Node.js](#)
- 20th August : [Node.js 0.2.0 released](#)

2011

- 31st March : [Node.js Guide](#)
- 1st May : [npm 1.0: Released](#)
- 1st May : [Ryan Dahl's AMA on Reddit](#)
- 10th July : [The Node Beginner Book, an introduction to Node.js, is completed.](#)
 - A comprehensive Node.js tutorial for beginners.
- 16th August : [LinkedIn uses Node.js](#)
 - LinkedIn launched its completely overhauled mobile app with new features and new parts under the hood.
- 5th October : [Ryan Dahl talks about the history of Node.js and why he created it](#)
- 5th December : [Node.js in production at Uber](#)
 - Uber Engineering Manager Curtis Chambers explains why his company completely re-engineered their application using Node.js to increase efficiency and improve the partner and customer experience.

2012

- 30th January : [Node.js creator Ryan Dahl steps away from Node's day-to-day](#)
- 25th June : [Node.js v0.8.0 \[stable\] is out](#)
- 20th December : [Hapi, a Node.js framework is released](#)

2013

- 30th April : [The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js](#)
- 17th May : [How We Built eBay's First Node.js Application](#)
- 15th November : [PayPal releases Kraken, a Node.js framework](#)
- 22nd November : [Node.js Memory Leak at Walmart](#)
 - Eran Hammer of Wal-Mart labs came to the Node.js core team complaining of a memory leak he had been tracking down for months.
- 19th December : [Koa - Web framework for Node.js](#)

2014

- 15th January : [TJ Fontaine takes over Node project](#)

- 10月23日 : Node.js 咨询委员会

- Joyent 和 Node.js 社区的几位成员宣布了一项关于 Node.js 咨询委员会的提案，作为向 Node.js 开源项目完全开放治理模式迈出的下一步。

- 11月19日 : Node.js 在火焰图中的应用 - Netflix

- 11月28日 : IO.js – V8 Javascript 的事件驱动 I/O

2015年

第一季度

- 1月14日 : IO.js 1.0.0 版本发布

- 2月10日 : Joyent 推动成立 Node.js 基金会 Joyent、IBM、微软

- PayPal、富达、SAP 和 Linux 基金会联手支持 Node.js 社区，推动中立且开放的治理

- 2月27日 : IO.js 与 Node.js 和解提案

Q2

- 4月14日 : npm 私有模块

- 5月28日 : Node 负责人 TJ Fontaine 辞职并离开 Joyent

- 5月13日 : Node.js 和 io.js 在 Node 基金会下合并

Q3

- 8月2日 : Trace - Node.js 性能监控与调试 Trace 是一个可视化的微服

- 监控工具，能为你提供操作微服务时所需的所有指标。

- 8月13日 : 4.0 是新的 1.0

Q4

- 10月12日 : Node v4.2.0, 第一个长期支持版本

- 12月8日 : Apigee、RisingStack 和雅虎加入 Node.js 基金会

- 12月8日 & 9日 : Node Interactive 会议

- Node.js 基金会举办的首届年度 Node.js 会议

2016年

第一季度

- 2月10日 : Express 成为孵化项目

- 3月23日 : leftpad 事件

- 3月29日 : 谷歌云平台加入 Node.js 基金会

Q2

- 4月26日 : npm 用户达到 210,000

Q3

- 7月18日 : CJ Silverio 成为 npm 的首席技术官

- 8月1日 : Node.js 调试解决方案 Trace 正式发布

- 9月15日 : 欧洲首届 Node Interactive 会议

Q4

- 10月11日 : Yarn 包管理器发布

- 10月18日 : Node.js 6 成为长期支持版本

参考文献

- 23rd October : Node.js Advisory Board

- Joyent and several members of the Node.js community announced a proposal for a Node.js Advisory Board as a next step towards a fully open governance model for the Node.js open source project.

- 19th November : Node.js in Flame Graphs - Netflix

- 28th November : IO.js – Evented I/O for V8 Javascript

2015

Q1

- 14th January : IO.js 1.0.0

- 10th February : Joyent Moves to Establish Node.js Foundation

- Joyent, IBM, Microsoft, PayPal, Fidelity, SAP and The Linux Foundation Join Forces to Support Node.js Community With Neutral and Open Governance

- 27th February : IO.js and Node.js reconciliation proposal

Q2

- 14th April : npm Private Modules

- 28th May : Node lead TJ Fontaine is stepping down and leaving Joyent

- 13th May : Node.js and io.js are merging under the Node Foundation

Q3

- 2nd August : Trace - Node.js performance monitoring and debugging

- Trace is a visualized microservice monitoring tool that gets you all the metrics you need when operating microservices.

- 13th August : 4.0 is the new 1.0

Q4

- 12th October : Node v4.2.0, first Long Term Support release

- 8th December : Apigee, RisingStack and Yahoo join the Node.js Foundation

- 8th & 9th December : Node Interactive

- The first annual Node.js conference by the Node.js Foundation

2016

Q1

- 10th February : Express becomes an incubated project

- 23rd March : The leftpad incident

- 29th March : Google Cloud Platform joins the Node.js Foundation

Q2

- 26th April : npm has 210,000 users

Q3

- 18th July : CJ Silverio becomes the CTO of npm

- 1st August : Trace, the Node.js debugging solution becomes generally available

- 15th September : The first Node Interactive in Europe

Q4

- 11th October : The yarn package manager got released

- 18th October : Node.js 6 becomes the LTS version

Reference

belindoc.com

第84章：passport.js

Passport 是一个流行的 Node 授权模块。简单来说，它处理应用中用户的所有授权请求。Passport 支持超过300种策略，因此你可以轻松集成 Facebook / Google 或任何其他社交网络的登录。这里我们讨论的策略是 Local，即使用你自己的注册用户数据库（通过用户名和密码）来验证用户身份。

第84.1节：passport.js 中 LocalStrategy 的示例

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.serializeUser(function(user, done) { //在 serializeUser 中你决定存储什么到 session。这里我只存储用户ID。
done(null, user.id);
});

passport.deserializeUser(function(id, done) { //这里你通过之前在 serializeUser 中存储的用户ID，从 session 存储中检索用户的所有信息。
db.findById(id, function(err, user) {
  done(err, user);
});
}

passport.use(new LocalStrategy(function(username, password, done) {
  db.findOne({username:username},function(err,student){
    if(err) return done(err,{message:message}); //错误的学号或密码;
    var pass_retrieved = student.pass_word;
    bcrypt.compare(password, pass_retrieved, function(err3, correct) {
      if(err3){
        message = [{"msg": "密码错误！"}];
        return done(null, false, {message:message}); // 密码错误
      }
      if(correct){
        return done(null, student);
      }
    });
  });
}));

app.use(session({ secret: 'super secret' })); // 让 passport 在其他页面记住用户
// 另请阅读 session 存储。我使用了 express-sessions。
app.use(passport.initialize());
app.use(passport.session());

app.post('/', passport.authenticate('local'),{successRedirect:'/users' failureRedirect: '/'},
  function(req, res, next){
});
```

Chapter 84: passport.js

Passport is a popular authorisation module for node. In simple words it handles all the authorisation requests on your app by users. Passport supports over 300 strategies so that you can easily integrate login with Facebook / Google or any other social network using it. The strategy that we will discuss here is the Local where you authenticate an user using your own database of registered users(using username and password).

Section 84.1: Example of LocalStrategy in passport.js

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.serializeUser(function(user, done) { //In serialize user you decide what to store in the session. Here I'm storing the user id only.
done(null, user.id);
});

passport.deserializeUser(function(id, done) { //Here you retrieve all the info of the user from the session storage using the user id stored in the session earlier using serialize user.
db.findById(id, function(err, user) {
  done(err, user);
});
}

passport.use(new LocalStrategy(function(username, password, done) {
  db.findOne({username:username},function(err,student){
    if(err) return done(err,{message:message}); //wrong roll_number or password;
    var pass_retrieved = student.pass_word;
    bcrypt.compare(password, pass_retrieved, function(err3, correct) {
      if(err3){
        message = [{"msg": "Incorrect Password!"}];
        return done(null, false, {message:message}); // wrong password
      }
      if(correct){
        return done(null, student);
      }
    });
  });
}));

app.use(session({ secret: 'super secret' })); //to make passport remember the user on other pages too.(Read about session store. I used express-sessions.)
app.use(passport.initialize());
app.use(passport.session());

app.post('/', passport.authenticate('local'),{successRedirect:'/users' failureRedirect: '/'},
  function(req, res, next){
});
```

第85章：异步编程

Node是一种编程语言，所有操作都可以以异步方式运行。下面你可以找到一些示例以及异步工作中的典型内容。

第85.1节：回调函数

JavaScript中的回调函数

回调函数在JavaScript中很常见。回调函数之所以在JavaScript中成为可能，是因为函数是一等公民。

同步回调。

回调函数可以是同步的也可以是异步的。由于异步回调函数可能更复杂，这里给出一个同步回调函数的简单示例。

```
// 一个使用名为 `cb` 的回调函数作为参数的函数
function getSyncMessage(cb) {
  cb("Hello World!");
}

console.log("调用 getSyncMessage 之前");
// 调用一个函数并传入一个回调函数作为参数。
getSyncMessage(function(message) {
  console.log(message);
});
console.log("调用 getSyncMessage 之后");
```

上述代码的输出结果是：

```
> 调用 getSyncMessage 之前
> Hello World!
> 调用 getSyncMessage 之后
```

首先我们将逐步讲解上述代码的执行过程。这部分内容主要针对那些还不理解回调函数概念的人，如果你已经理解，可以跳过这段。首先代码被解析，然后第6行被执行，输出调用 `getSyncMessage` 之前到控制台。接着第8行被执行，调用 `getSyncMessage`，并传入一个匿名函数作为参数，赋值给 `getSyncMessage` 函数中的参数 `cb`。执行进入第3行的 `getSyncMessage` 函数，执行刚传入的函数 `cb`，这次调用传入了字符串参数 "Hello World"，对应匿名函数中的参数 `message`。然后执行跳转到第9行，输出 "Hello World!" 到控制台。接着执行过程经历调用栈的退出（参见相关内容），依次执行第10行、第4行，最后回到第11行。

关于回调的一些基本信息：

- 你传入函数作为回调的函数可能被调用零次、一次或多次。这完全取决于实现方式。
- 回调函数可能被同步调用，也可能被异步调用，甚至可能同时以同步和异步方式调用。
- 就像普通函数一样，你给函数参数起的名字并不重要，但参数的顺序很重要。
例如，在第8行，参数 `message` 可以被命名为 `statement`、`msg`，或者如果你想胡闹的话，甚至可以叫 `jellybean`。所以你应该知道传入回调的参数是什么，

Chapter 85: Asynchronous programming

Node is a programming language where everything could run on an asynchronous way. Below you could find some examples and the typical things of asynchronous working.

Section 85.1: Callback functions

Callback functions in JavaScript

Callback functions are common in JavaScript. Callback functions are possible in JavaScript because [functions are first-class citizens](#).

Synchronous callbacks.

Callback functions can be synchronous or asynchronous. Since Asynchronous callback functions may be more complex here is a simple example of a synchronous callback function.

```
// a function that uses a callback named `cb` as a parameter
function getSyncMessage(cb) {
  cb("Hello World!");
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getSyncMessage(function(message) {
  console.log(message);
});
console.log("After getSyncMessage call");
```

The output for the above code is:

```
> Before getSyncMessage call
> Hello World!
> After getSyncMessage call
```

First we will step through how the above code is executed. This is more for those who do not already understand the concept of callbacks if you do already understand it feel free to skip this paragraph. First the code is parsed and then the first interesting thing to happen is line 6 is executed which outputs `Before getSyncMessage call` to the console. Then line 8 is executed which calls the function `getSyncMessage` sending in an anonymous function as an argument for the parameter named `cb` in the `getSyncMessage` function. Execution is now done inside the `getSyncMessage` function on line 3 which executes the function `cb` which was just passed in, this call sends an argument string "Hello World" for the param named `message` in the passed in anonymous function. Execution then goes to line 9 which logs `Hello World!` to the console. Then the execution goes through the process of exiting the [callstack](#) ([see also](#)) hitting line 10 then line 4 then finally back to line 11.

Some information to know about callbacks in general:

- The function you send in to a function as a callback may be called zero times, once, or multiple times. It all depends on implementation.
- The callback function may be called synchronously or asynchronously and possibly both synchronously and asynchronously.
- Just like normal functions the names you give parameters to your function are not important but the order is. So for example on line 8 the parameter `message` could have been named `statement`, `msg`, or if you're being nonsensical something like `jellybean`. So you should know what parameters are sent into your callback so

这样你才能以正确的顺序和合适的名称获取它们。

异步回调。

关于JavaScript，有一点需要注意的是它默认是同步的，但环境（浏览器、Node.js等）中提供了一些API可以使其变为异步（关于这点有更多内容在[here](#)）。

在接受回调的JavaScript环境中，一些常见的异步操作包括：

- 事件
- setTimeout
- setInterval
- fetch API
- Promises

此外，任何使用上述函数之一的函数都可以用一个接受回调函数的函数来包装，回调函数随后将是一个异步回调（尽管用一个接受回调函数的函数来包装 Promise 可能被视为反模式，因为处理 Promise 有更优先的方法）。

基于这些信息，我们可以构造一个类似于上述同步函数的异步函数。

```
// 一个使用名为 `cb` 的回调函数作为参数的函数
function getAsyncMessage(cb) {
  setTimeout(function () { cb("Hello World!") }, 1000);

  console.log("调用 getSyncMessage 之前");
  // 调用一个函数并传入一个回调函数作为参数。
  getAsyncMessage(function(message) {
    console.log(message);
  });
  console.log("调用 getSyncMessage 之后");
```

这会在控制台打印以下内容：

```
> 在调用 getSyncMessage 之前
> 在调用 getSyncMessage 之后
// 暂停 1000 毫秒, 无输出
> Hello World!
```

代码执行到第 6 行，打印“在调用 getSyncMessage 之前”。然后执行到第 8 行，调用 getAsyncMessage 并传入一个回调函数作为参数cb。接着执行第 3 行，调用 setTimeout，传入一个回调函数作为第一个参数，数字 300 作为第二个参数。setTimeout 会执行它的操作并保存该回调函数，以便在 1000 毫秒后调用，但在设置超时并暂停之前，它会将执行权交还给之前暂停的位置，继续执行第 4 行，然后是第 11 行，随后暂停 1 秒，setTimeout 再调用它的回调函数，执行权回到第 3 行，getAsyncMessages 回调函数以参数 "Hello World" 调用message，随后在第9行的控制台中记录该值。

Node.js中的回调函数

NodeJS具有异步回调，通常会向你的函数传递两个参数，有时习惯上称为错误和数据。以下是读取文件文本的示例。

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

you can get them in the right order with proper names.

Asynchronous callbacks.

One thing to note about JavaScript is it is synchronous by default, but there are APIs given in the environment (browser, Node.js, etc.) that could make it asynchronous (there's more about that [here](#)).

Some common things that are asynchronous in JavaScript environments that accept callbacks:

- Events
- setTimeout
- setInterval
- the fetch API
- Promises

Also any function that uses one of the above functions may be wrapped with a function that takes a callback and the callback would then be an asynchronous callback (although wrapping a promises with a function that takes a callback would likely be considered an anti-pattern as there are more preferred ways to handle promises).

So given that information we can construct an asynchronous function similar to the above synchronous one.

```
// a function that uses a callback named `cb` as a parameter
function getAsyncMessage(cb) {
  setTimeout(function () { cb("Hello World!") }, 1000);

  console.log("Before getSyncMessage call");
  // calling a function and sending in a callback function as an argument.
  getAsyncMessage(function(message) {
    console.log(message);
  });
  console.log("After getSyncMessage call");
```

Which prints the following to the console:

```
> Before getSyncMessage call
> After getSyncMessage call
// pauses for 1000 ms with no output
> Hello World!
```

Line execution goes to line 6 logs "Before getSyncMessage call". Then execution goes to line 8 calling getAsyncMessage with a callback for the param cb. Line 3 is then executed which calls setTimeout with a callback as the first argument and the number 300 as the second argument. setTimeout does whatever it does and holds on to that callback so that it can call it later in 1000 milliseconds, but following setting up the timeout and before it pauses the 1000 milliseconds it hands execution back to where it left off so it goes to line 4, then line 11, and then pauses for 1 second and setTimeout then calls its callback function which takes execution back to line 3 where getAsyncMessages callback is called with value "Hello World" for its parameter message which is then logged to the console on line 9.

Callback functions in Node.js

NodeJS has asynchronous callbacks and commonly supplies two parameters to your functions sometimes conventionally called err and data. An example with reading a file text.

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

```

fs.readFile("./test.txt", "utf8", function(错误, 数据) {
  if(错误) {
    // 处理错误
  } else {
    // 使用data处理文件文本
  }
});

```

这是一个只调用一次的回调示例。

即使你只是记录错误或抛出错误，处理错误也是良好的实践。如果你抛出或返回，else语句不是必须的，可以移除以减少缩进，只要你通过抛出或返回等方式停止当前函数的执行即可。

虽然常见的回调参数是错误和数据，但你的回调函数不一定总是采用这种模式，最好查看文档确认。

另一个回调示例来自express库 (express 4.x) :

```

// 这段代码摘自 http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();

// 这个 app.get 方法接受一个要监听的 URL 路由和一个回调函数
// 当用户请求该路由时调用该回调函数。
app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);

```

此示例展示了一个会被多次调用的回调函数。回调函数接收两个对象作为参数，这里命名为 req 和 res，这两个名称分别对应请求和响应，提供了查看传入请求和设置将发送给用户的响应的方法。

如你所见，回调函数有多种方式可用于执行 JavaScript 中的同步和异步代码，回调函数在 JavaScript 中非常普遍。

第85.2节：回调地狱

回调地狱（也称为厄运金字塔或回旋镖效应）是指在一个回调函数内部嵌套了过多的回调函数。以下是一个读取文件的示例（ES6写法）。

```

const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;

fs.exists(filename, exists => {
  if(exists) {
    fs.stat(filename, (err, stats) => {
      if(err) {
        抛出 err;
      }
      if(stats.isFile()) {
        fs.readFile(filename, null, (err, data) => {
          if(err) {
            抛出 err;
          }
        })
      }
    })
  }
  console.log(data);
}

```

```

fs.readFile("./test.txt", "utf8", function(err, data) {
  if(err) {
    // handle the error
  } else {
    // process the file text given with data
  }
});

```

This is an example of a callback that is called a single time.

It's good practice to handle the error somehow even if you just log it or throw it. The else is not necessary if you throw or return and can be removed to decrease indentation so long as you stop execution of the current function in the if by doing something like throwing or returning.

Though it may be common to see err, data it may not always be the case that your callbacks will use that pattern it's best to look at documentation.

Another example callback comes from the express library (express 4.x):

```

// this code snippet was on http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();

// this app.get method takes a url route to watch for and a callback
// to call whenever that route is requested by a user.
app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);

```

This example shows a callback that is called multiple times. The callback is provided with two objects as params named here as req and res these names correspond to request and response respectively, and they provide ways to view the request coming in and set up the response that will be sent to the user.

As you can see there are various ways a callback can be used to execute sync and async code in JavaScript and callbacks are very ubiquitous throughout JavaScript.

Section 85.2: Callback hell

Callback hell (also a pyramid of doom or boomerang effect) arises when you nest too many callback functions inside a callback function. Here is an example to read a file (in ES6).

```

const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;

fs.exists(filename, exists => {
  if(exists) {
    fs.stat(filename, (err, stats) => {
      if(err) {
        抛出 err;
      }
      if(stats.isFile()) {
        fs.readFile(filename, null, (err, data) => {
          if(err) {
            抛出 err;
          }
          console.log(data);
        })
      }
    })
  }
}

```

```

    });
}
else {
    throw new Error("此位置不包含文件");
}
};

else {
    throw new Error("404 : 文件未找到");
}
);
}
);

```

如何避免“回调地狱”

建议嵌套不超过2个回调函数。这将帮助你保持代码的可读性，并且将来维护起来更容易。如果需要嵌套超过2个回调，尝试使用分布式事件代替。

还有一个名为async的库可用于管理回调及其执行，该库可在npm上获取。它提高了回调代码的可读性，并让你对回调代码流程有更多控制，包括允许你并行或串行运行它们。

第85.3节：原生Promise

版本 ≥ v6.0.0

Promise是异步编程的工具。在JavaScript中，Promise以其then方法而闻名。Promise有两个主要状态“pending”（等待中）和“settled”（已完成）。一旦Promise处于“settled”状态，就不能回到“pending”状态。这意味着Promise主要适用于只发生一次的事件。“settled”状态又分为“resolved”（已解决）和“rejected”（已拒绝）两种状态。你可以使用new关键字并传入一个函数来创建新的Promise。

```
new Promise(function (resolve, reject) {});
```

传入 Promise 构造函数的函数总是接收第一个和第二个参数，通常命名为 resolve 和 reject。这两个参数的命名是约定，但它们会将 Promise 置于“已解决”状态或“已拒绝”状态。当调用其中一个时，Promise 会从“等待中”变为“已完成”。resolve 在期望的操作（通常是异步操作）完成时被调用，reject 则在操作出错时使用。

下面的 timeout 是一个返回 Promise 的函数。

```

function timeout (ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve("它已被解决！");
        }, ms)
    });
}

timeout(1000).then(function (dataFromPromise) {
    // 输出 "它已被解决！"
    console.log(dataFromPromise);
})

console.log("等待中...");

```

控制台输出

```

    });
}
else {
    throw new Error("This location contains not a file");
}
});
else {
    throw new Error("404: file not found");
}
);

```

How to avoid "Callback Hell"

It is recommended to nest no more than 2 callback functions. This will help you maintain code readability and will make much easier to maintain in the future. If you have a need to nest more than 2 callbacks, try to make use of [distributed events](#) instead.

There also exists a library called [async](#) that helps manage callbacks and their execution available on npm. It increases the readability of callback code and gives you more control over your callback code flow, including allowing you to run them in parallel or in series.

Section 85.3: Native Promises

Version ≥ v6.0.0

Promises are a tool for async programming. In JavaScript promises are known for their then methods. Promises have two main states 'pending' and 'settled'. Once a promise is 'settled' it cannot go back to 'pending'. This means that promises are mostly good for events that only occur once. The 'settled' state has two states as well 'resolved' and 'rejected'. You can create a new promise using the **new** keyword and passing a function into the constructor **new Promise(function (resolve, reject) {})**.

The function passed into the Promise constructor always receives a first and second parameter usually named resolve and reject respectively. The naming of these two parameters is convention, but they will put the promise into either the 'resolved' state or the 'rejected' state. When either one of these is called the promise goes from being 'pending' to 'settled'. resolve is called when the desired action, which is often asynchronous, has been performed and reject is used if the action has errored.

In the below timeout is a function that returns a Promise.

```

function timeout (ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve("It was resolved!");
        }, ms)
    });
}

timeout(1000).then(function (dataFromPromise) {
    // logs "It was resolved!"
    console.log(dataFromPromise);
})

console.log("waiting...");

```

console output

等待中...

// <>暂停一秒>>

它已被解决！

当调用timeout时，传递给Promise构造函数的函数会立即执行。然后执行setTimeout方法，其回调被设置为在接下来的毫秒内触发，这里毫秒=1000。由于setTimeout的回调尚未触发，timeout函数会将控制权返回给调用范围。随后then方法链被存储，等待Promise解决后调用。如果这里有catch方法，它们也会被存储，但会在Promise“拒绝”时触发。

脚本随后打印“waiting...”。一秒后，setTimeout调用其回调，该回调调用resolve函数，传入字符串“It was resolved!”。该字符串随后传入then方法的回调，并被记录输出给用户。

同理，你可以将需要回调的异步setTimeout函数封装成Promise，也可以用Promise封装任何单一的异步操作。

更多关于Promise的信息请参阅JavaScript文档中的Promises。

第85.4节：代码示例

问题：下面代码的输出是什么，为什么？

```
setTimeout(function() {
  console.log("A");
}, 1000);

setTimeout(function() {
  console.log("B");
}, 0);

getDataFromDatabase(function(err, data) {
  console.log("C");
  setTimeout(function() {
    console.log("D");
  }, 1000);
});

console.log("E");
```

输出：这是确定的：EBAD。 C 何时被打印是未知的。

解释：编译器不会在setTimeout和getDataFromDatabase方法上停止。所以他首先打印的行是E。回调函数（setTimeout的第一个参数）将在超时后以异步方式运行！

更多细节：

1. E 没有setTimeout
2. 有一个0毫秒的setTimeout
3. 有一个1000毫秒的setTimeout
4. D 必须请求数据库，之后必须等待1000毫秒，所以它在A之后执行。
5. C 是未知的，因为数据库数据请求的时间未知。它可能在A之前或之后。

waiting...

// <> pauses for one second>>

It was resolved!

When timeout is called the function passed to the Promise constructor is executed without delay. Then the setTimeout method is executed and its callback is set to fire in the next ms milliseconds, in this case ms=1000. Since the callback to the setTimeout isn't fired yet the timeout function returns control to the calling scope. The chain of then methods are then stored to be called later when/if the Promise has resolved. If there were catch methods here they would be stored as well, but would be fired when/if the promise 'rejects'.

The script then prints 'waiting...'. One second later the setTimeout calls its callback which calls the resolve function with the string "It was resolved!". That string is then passed into the then method's callback and is then logged to the user.

In the same sense you can wrap the asynchronous setTimeout function which requires a callback you can wrap any singular asynchronous action with a promise.

Read more about promises in the JavaScript documentation Promises.

Section 85.4: Code example

Question: What is the output of code below and why?

```
setTimeout(function() {
  console.log("A");
}, 1000);

setTimeout(function() {
  console.log("B");
}, 0);

getDataFromDatabase(function(err, data) {
  console.log("C");
  setTimeout(function() {
    console.log("D");
  }, 1000);
});

console.log("E");
```

Output: This is known for sure: EBAD. C is unknown when it will be logged.

Explanation: The compiler will not stop on the setTimeout and the getDataFromDatabase methodes. So the first line he will log is E. The callback functions (first argument of setTimeout) will run after the set timeout on a asynchronous way!

More details:

1. E has no setTimeout
2. B has a set timeout of 0 milliseconds
3. A has a set timeout of 1000 milliseconds
4. D must request a database, after it must D wait 1000 milliseconds so it comes after A.
5. C is unknown because it is unknown when the data of the database is requested. It could be before or after A.

第85.5节：异步错误处理

尝试捕获

错误必须始终被处理。如果你使用同步编程，可以使用try catch。但如果你使用异步编程，这种方法不起作用！例如：

```
try {
setTimeout(function() {
    throw new Error("我是一个未捕获的错误，会导致服务器停止！");
}, 100);
}
catch (ex) {
console.error("在异步情况下此错误无法被处理：" + ex);
}
```

异步错误只能在回调函数内部被处理！

可行的解决方案

版本 ≤ v0.8

事件处理器程序

Node.js 的最初版本引入了事件处理器程序。

```
process.on("UncaughtException", function(err, data) {
    if (err) {
        // 错误处理
    }
});
```

版本 ≥ v0.8

域 (Domains)

在域内部，错误通过事件发射器释放。通过使用它，所有错误、定时器、回调方法隐式地只注册在该域内。发生错误时，会发送一个错误事件，但不会导致应用程序崩溃。

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

d1.run(function() {
d2.add(setTimeout(function() {
    throw new Error("error on the timer of domain 2");
}, 0));
});

d1.on("error", function(err) {
    console.log("域1错误：" + err);
});

d2.on("error", function(err) {
    console.log("域2错误：" + err);
});
```

Section 85.5: Async error handling

Try catch

Errors must always be handled. If you are using synchronous programming you could use a **try catch**. But this does not work if you work asynchronous! Example:

```
try {
setTimeout(function() {
    throw new Error("I'm an uncaught error and will stop the server!");
}, 100);
}
catch (ex) {
    console.error("This error will not be work in an asynchronous situation：" + ex);
}
```

Async errors will only be handled inside the callback function!

Working possibilities

Version ≤ v0.8

Event handlers

The first versions of Node.js got an event handler.

```
process.on("UncaughtException", function(err, data) {
    if (err) {
        // error handling
    }
});
```

Version ≥ v0.8

Domains

Inside a domain, the errors are released via the event emitters. By using this are all errors, timers, callback methods implicitly only registered inside the domain. By an error, be an error event sent and didn't crash the application.

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

d1.run(function() {
d2.add(setTimeout(function() {
    throw new Error("error on the timer of domain 2");
}, 0));
});

d1.on("error", function(err) {
    console.log("error at domain 1：" + err);
});

d2.on("error", function(err) {
    console.log("error at domain 2：" + err);
});
```

第86章：Node.js中不使用任何库的STDIN和STDOUT代码

这是一个简单的Node.js程序，接收用户输入并将其打印到控制台。

process对象是一个全局对象，提供关于当前Node.js进程的信息和控制。作为全局对象，它始终可供Node.js应用程序使用，无需require()。

第86.1节：程序

process.stdin属性返回一个可读流，等同于或关联于标准输入(stdin)。

process.stdout属性返回一个可写流，等同于或关联于标准输出(stdout)。

```
process.stdin.resume()  
console.log('请输入要显示的数据');  
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

Chapter 86: Node.js code for STDIN and STDOUT without using any library

This is a simple program in node.js to which takes input from the user and prints it to the console.

The **process** object is a global that provides information about, and control over, the current Node.js process. As a global, it is always available to Node.js applications without using require().

Section 86.1: Program

The **process.stdin** property returns a Readable stream equivalent to or associated with stdin.

The **process.stdout** property returns a Writable stream equivalent to or associated with stdout.

```
process.stdin.resume()  
console.log('Enter the data to be displayed');  
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

第87章：Node.js/Express.js的MongoDB集成

MongoDB 是最受欢迎的 NoSQL 数据库之一，这要归功于 MEAN 技术栈的支持。一旦你理解了有点怪异的查询语法，从 Express 应用连接 Mongo 数据库就变得快速且简单。我们将使用 Mongoose 来帮助我们。

第 87.1 节：安装 MongoDB

```
npm install --save mongodb  
npm install --save mongoose //一个简单的封装，便于开发
```

在你的服务器文件中（通常命名为 index.js 或 server.js）

```
const express = require('express');  
const mongodb = require('mongodb');  
const mongoose = require('mongoose');  
const mongoConnectionString = 'http://localhost/database_name';  
  
mongoose.connect(mongoConnectionString, (err) => {  
  if (err) {  
    console.log('无法连接到数据库');  
  }  
});
```

第 87.2 节：创建 Mongoose 模型

```
const Schema = mongoose.Schema;  
const ObjectId = Schema.Types.ObjectId;  
  
const Article = new Schema({  
  title: {  
    type: String,  
    unique: true,  
    required: [true, '文章必须有标题']  
  },  
  author: {  
    type: ObjectId,  
    ref: 'User'  
  }  
});  
  
module.exports = mongoose.model('Article', Article);
```

让我们来解析一下。MongoDB 和 Mongoose 使用 JSON（实际上是 BSON，但这里无关紧要）作为数据格式。在顶部，我设置了一些变量以减少输入量。

我创建了一个 new Schema 并将其赋值给一个常量。它是简单的 JSON，每个属性都是另一个对象，带有帮助强制执行更一致模式的属性。unique 强制插入数据库的新实例显然是唯一的。这对于防止用户在服务上创建多个账户非常有用。

required 是另一个，以数组形式声明。第一个元素是布尔值，第二个是当插入或更新的值不存在时的错误信息。

ObjectId 用于模型之间的关系。例如“用户有许多评论”。其他

Chapter 87: MongoDB Integration for Node.js/Express.js

MongoDB is one of the most popular NoSQL databases, thanks to the help of the MEAN stack. Interfacing with a Mongo database from an Express app is quick and easy, once you understand the kinda-wonky query syntax. We'll use Mongoose to help us out.

Section 87.1: Installing MongoDB

```
npm install --save mongodb  
npm install --save mongoose //A simple wrapper for ease of development
```

In your server file (normally named index.js or server.js)

```
const express = require('express');  
const mongodb = require('mongodb');  
const mongoose = require('mongoose');  
const mongoConnectionString = 'http://localhost/database_name';  
  
mongoose.connect(mongoConnectionString, (err) => {  
  if (err) {  
    console.log('Could not connect to the database');  
  }  
});
```

Section 87.2: Creating a Mongoose Model

```
const Schema = mongoose.Schema;  
const ObjectId = Schema.Types.ObjectId;  
  
const Article = new Schema({  
  title: {  
    type: String,  
    unique: true,  
    required: [true, 'Article must have title']  
  },  
  author: {  
    type: ObjectId,  
    ref: 'User'  
  }  
});  
  
module.exports = mongoose.model('Article', Article);
```

Let's dissect this. MongoDB and Mongoose use JSON(actually BSON, but that's irrelevant here) as the data format. At the top, I've set a few variables to reduce typing.

I create a new Schema and assign it to a constant. It's simple JSON, and each attribute is another Object with properties that help enforce a more consistent schema. Unique forces new instances being inserted in the database to, obviously, be unique. This is great for preventing a user creating multiple accounts on a service.

Required is another, declared as an array. The first element is the boolean value, and the second the error message should the value being inserted or updated fail to exist.

ObjectIds are used for relationships between Models. Examples might be 'Users have many Comments'. Other

属性也可以用来代替 ObjectId。字符串如用户名就是一个例子。

最后，将模型导出以供您的API路由使用，可以访问您的模式。

第87.3节：查询您的Mongo数据库

一个简单的GET请求。假设上面示例中的模型位于文件./db/models/Article.js中。

```
const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });

  app.use('/api', routes);
};
```

现在我们可以通过向该端点发送HTTP请求来获取数据库中的数据。不过，有几点需要注意：

1. 限制的作用正如其字面意思。我只收到了5份文件。
2. Lean 从原始 BSON 中剔除了一些内容，减少了复杂性和开销。不是必须的，但很有用。
3. 使用find而不是findOne时，确认doc.length大于0。这是因为find 总是返回一个数组，所以空数组不会处理你的错误，除非检查其长度。我個人喜欢以这种格式发送错误信息。根据你的需要进行更改。返回的文档也是同样的处理。
4. 我个人喜欢以这种格式发送错误信息。根据你的需要进行更改。返回的文档也是同样的处理。
5. 本示例中的代码假设你已将其放在另一个文件中，而不是直接放在 express 服务器上。要在服务器中调用它，请在服务器代码中包含以下行：

```
const app = express();
require('./path/to/this/file')(app) //
```

attributes can be used instead of ObjectId. Strings like a username is one example.

Lastly, exporting the model for use with your API routes provides access to your schema.

Section 87.3: Querying your Mongo Database

A simple GET request. Let's assume the Model from the example above is in the file ./db/models/Article.js.

```
const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });

  app.use('/api', routes);
};
```

We can now get the data from our database by sending an HTTP request to this endpoint. A few key things, though:

1. Limit does exactly what it looks like. I'm only getting 5 documents back.
2. Lean strips away some stuff from the raw BSON, reducing complexity and overhead. Not required. But useful.
3. When using find instead of findOne, confirm that the doc.length is greater than 0. This is because find always returns an array, so an empty array will not handle your error unless it is checked for length.
4. I personally like to send the error message in that format. Change it to suit your needs. Same thing for the returned document.
5. The code in this example is written under the assumption that you have placed it in another file and not directly on the express server. To call this in the server, include these lines in your server code:

```
const app = express();
require('./path/to/this/file')(app) //
```

第88章：Lodash

Lodash 是一个方便的 JavaScript 工具库。

第88.1节：过滤集合

下面的代码片段展示了使用 lodash 过滤对象数组的各种方法。

```
let lodash = require('lodash');

var countries = [
  {"key": "DE", "name": "德国", "active": false},
  {"key": "ZA", "name": "南非", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
  return country.key === "DE";
});
// => [{"key": "DE", "name": "德国"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "德国"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "南非"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "南非"}];
```

Chapter 88: Lodash

Lodash is a handy JavaScript utility library.

Section 88.1: Filter a collection

The code snippet below shows the various ways you can filter on an array of objects using lodash.

```
let lodash = require('lodash');

var countries = [
  {"key": "DE", "name": "Deutschland", "active": false},
  {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
  return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, [ "key", "ZA" ]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

第89章：Node.js中的csv解析器

从csv读取数据可以通过多种方式处理。一种解决方案是将csv文件读取到数组中。从那里你可以对数组进行操作。

第89.1节：使用FS读取CSV文件

fs 是 node 中的文件系统 API。我们可以在 fs 变量上使用 readFile 方法，传入一个data.csv文件、格式和一个读取并拆分csv以便进一步处理的函数。

这假设你在同一文件夹中有一个名为data.csv的文件。

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {var dataArray = d
ata.split(/\r?\n/);
console.log(dataArray);
});
```

你现在可以像使用其他数组一样使用该数组进行操作。

Chapter 89: csv parser in node js

Reading data in from a csv can be handled in many ways. One solution is to read the csv file into an array. From there you can do work on the array.

Section 89.1: Using FS to read in a CSV

fs is the [File System API](#) in node. We can use the method readFile on our fs variable, pass it a data.csv file, format and function that reads and splits the csv for further processing.

This assumes you have a file named data.csv in the same folder.

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {
  var dataArray = data.split(/\r?\n/);
  console.log(dataArray);
});
```

You can now use the array like any other to do work on it.

第90章：Loopback - 基于REST的连接器

基于REST的连接器及其处理方法。我们都知道 Loopback 对基于REST的连接不够优雅

第90.1节：添加基于网页的连接器

```
//此示例获取来自 iTunes 的响应
{
  "rest": {
    "name": "rest",
    "connector": "rest",
    "debug": true,
    "options": {
      "useQueryString": true,
      "timeout": 10000,
      "headers": {
        "accepts": "application/json",
        "content-type": "application/json"
      }
    },
    "operations": [
      {
        "template": {
          "method": "GET",
          "url": "https://itunes.apple.com/search",
          "query": {
            "term": "{keyword}",
            "country": "{country=IN}",
            "media": "{itemType=music}",
            "limit": "{limit=10}",
            "explicit": "false"
          }
        },
        "functions": {
          "search": [
            "keyword",
            "country",
            "itemType",
            "limit"
          ]
        }
      },
      {
        "template": {
          "method": "GET",
          "url": "https://itunes.apple.com/lookup",
          "query": {
            "id": "{id}"
          }
        },
        "functions": {
          "findById": [
            "id"
          ]
        }
      }
    ]
  }
}
```

Chapter 90: Loopback - REST Based connector

Rest based connectors and how to deal with them. We all know Loopback does not provide elegance to REST based connections

Section 90.1: Adding a web based connector

```
//This example gets the response from iTunes
{
  "rest": {
    "name": "rest",
    "connector": "rest",
    "debug": true,
    "options": {
      "useQueryString": true,
      "timeout": 10000,
      "headers": {
        "accepts": "application/json",
        "content-type": "application/json"
      }
    },
    "operations": [
      {
        "template": {
          "method": "GET",
          "url": "https://itunes.apple.com/search",
          "query": {
            "term": "{keyword}",
            "country": "{country=IN}",
            "media": "{itemType=music}",
            "limit": "{limit=10}",
            "explicit": "false"
          }
        },
        "functions": {
          "search": [
            "keyword",
            "country",
            "itemType",
            "limit"
          ]
        }
      },
      {
        "template": {
          "method": "GET",
          "url": "https://itunes.apple.com/lookup",
          "query": {
            "id": "{id}"
          }
        },
        "functions": {
          "findById": [
            "id"
          ]
        }
      }
    ]
  }
}
```

```
    ]  
  }  
}
```

```
    ]  
  }  
}
```

belindoc.com

第91章：将node.js作为服务运行

与许多网络服务器不同，Node默认并未作为服务安装。但在生产环境中，最好让它作为守护进程运行，由初始化系统管理。

第91.1节：Node.js作为systemd守护进程

systemd是大多数Linux发行版中的事实上的初始化系统。配置Node以使用systemd运行后，可以使用service命令来管理它。

首先，需要一个配置文件，我们来创建它。对于基于Debian的发行版，文件路径为
/etc/systemd/system/node.service

```
[Unit]
Description=我的超级nodejs应用

[Service]
# 设置工作目录以保持相对路径一致
WorkingDirectory=/var/www/app

# 启动服务器文件（文件相对于WorkingDirectory）
ExecStart=/usr/bin/node serverCluster.js

# 如果进程崩溃，始终尝试重启
Restart=always

# 在崩溃和重启之间等待500毫秒
RestartSec=500ms

# 将日志发送到syslog（不会与应用自身的其他日志配置冲突）
StandardOutput=syslog
StandardError=syslog

# syslog中的nodejs进程名称
SyslogIdentifier=nodejs

# 用户和组启动应用程序
User=www-data
Group=www-data

# 设置环境（开发，生产...）
Environment=NODE_ENV=production

[Install]
# 在多用户系统级别启动节点（= sysVinit 运行级别 3）
WantedBy=multi-user.target
```

现在可以分别使用以下命令启动、停止和重启应用：

```
service node start
service node stop
service node restart
```

要让systemd在启动时自动启动node，只需输入：systemctl enable node。

就是这样，node现在作为守护进程运行。

Chapter 91: Running node.js as a service

Unlike many web servers, Node isn't installed as a service out of the box. But in production, it's better to have it run as a *dæmon*, managed by an init system.

Section 91.1: Node.js as a systemd *dæmon*

systemd is the *de facto* init system in most Linux distributions. After Node has been configured to run with systemd, it's possible to use the `service` command to manage it.

First of all, it needs a config file, let's create it. For Debian based distros, it will be in
`/etc/systemd/system/node.service`

```
[Unit]
Description=My super nodejs app

[Service]
# set the working directory to have consistent relative paths
WorkingDirectory=/var/www/app

# start the server file (file is relative to WorkingDirectory here)
ExecStart=/usr/bin/node serverCluster.js

# if process crashes, always try to restart
Restart=always

# let 500ms between the crash and the restart
RestartSec=500ms

# send log to syslog here (it doesn't compete with other log config in the app itself)
StandardOutput=syslog
StandardError=syslog

# nodejs process name in syslog
SyslogIdentifier=nodejs

# user and group starting the app
User=www-data
Group=www-data

# set the environment (dev, prod...)
Environment=NODE_ENV=production

[Install]
# start node at multi user system level (= sysVinit runlevel 3)
WantedBy=multi-user.target
```

It's now possible to respectively start, stop and restart the app with:

```
service node start
service node stop
service node restart
```

To tell systemd to automatically start node on boot, just type: `systemctl enable node`.

That's all, node now runs as a *dæmon*.

第92章：带有 CORS 的 Node.js

第92.1节：在 express.js 中启用 CORS

由于 node.js 常用于构建 API，如果你希望能够从不同域请求该 API，正确的 CORS 设置可能是救命稻草。

在示例中，我们将为更广泛的配置进行设置（允许来自任何域的所有请求类型）。

在你的 server.js 中初始化 express 之后：

```
// 创建 express 服务器
const app = express();

app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');

    // 预检请求的授权头
    // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
    next();
}

app.options('*', (req, res) => {
    // 允许的 XHR 方法
    res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
    res.send();
});
```

通常，Node 在生产服务器上是运行在代理服务器后面的。因此，反向代理服务器（如 Apache 或 Nginx）将负责 CORS 配置。

为了方便适应这种情况，可以只在开发环境中启用 node.js 的 CORS。

这可以通过检查 NODE_ENV 轻松实现：

```
const app = express();

if (process.env.NODE_ENV === 'development') {
    // CORS 设置
}
```

Chapter 92: Node.js with CORS

Section 92.1: Enable CORS in express.js

As node.js is often used to build API, proper CORS setting can be a life saver if you want to be able to request the API from different domains.

In the example, we'll set it up for the wider configuration (authorize all request types from any domain).

In your server.js after initializing express:

```
// Create express server
const app = express();

app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');

    // authorized headers for preflight requests
    // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
    next();
}

app.options('*', (req, res) => {
    // allowed XHR methods
    res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
    res.send();
});
```

Usually, node is ran behind a proxy on production servers. Therefore the reverse proxy server (such as Apache or Nginx) will be responsible for the CORS config.

To conveniently adapt this scenario, it's possible to only enable node.js CORS when it's in development.

This is easily done by checking NODE_ENV:

```
const app = express();

if (process.env.NODE_ENV === 'development') {
    // CORS settings
}
```

第93章：开始使用 Node 的性能分析

本文的目的是入门 Node.js 应用的性能分析，以及如何理解这些结果以捕捉错误或内存泄漏。运行中的 Node.js 应用本质上是一个 V8 引擎进程，在很多方面类似于浏览器中运行的网站，我们基本上可以捕获与网站进程相关的所有指标，用于 Node 应用。

我偏好的工具是 Chrome 开发者工具或 Chrome 检查器，配合 node-inspector 使用。

第93.1节：对简单的 Node 应用进行性能分析

步骤1：使用 npm 在你的机器上全局安装 node-inspector 包

```
$ npm install -g node-inspector
```

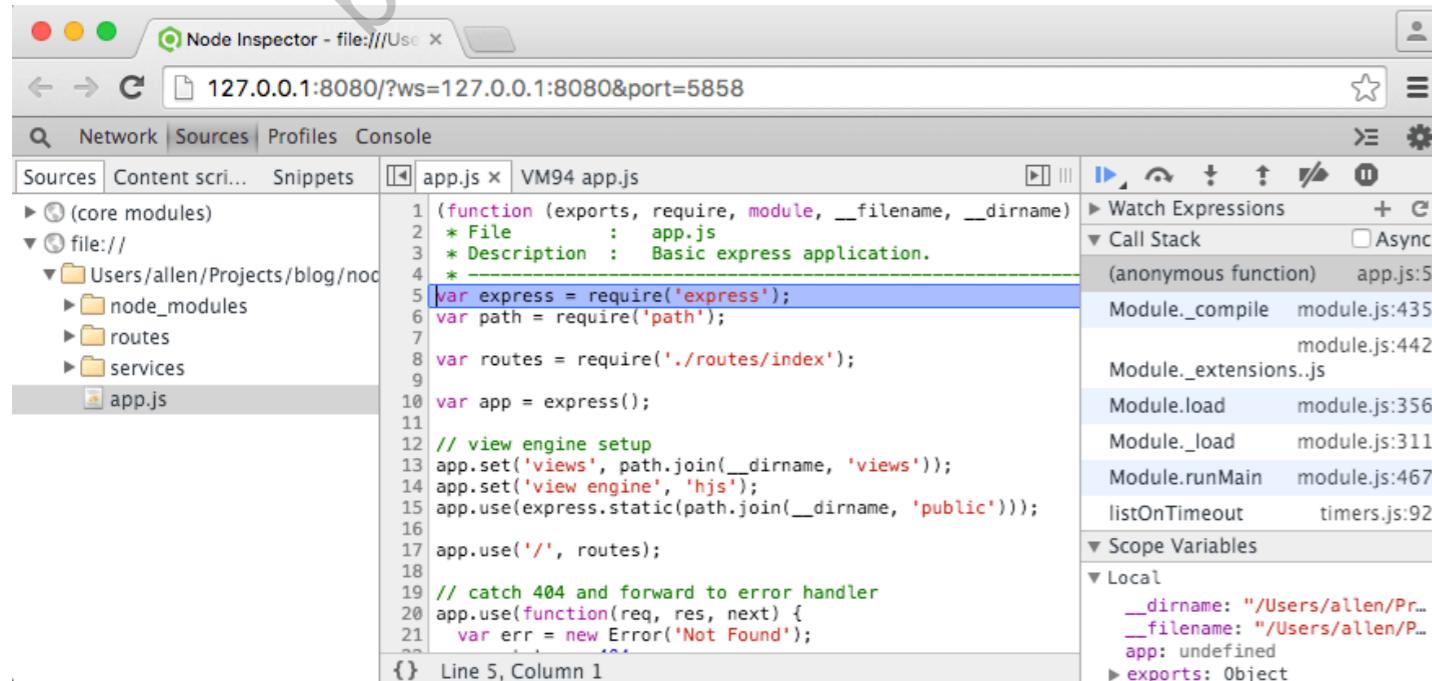
步骤 2：启动 node-inspector 服务器

```
$ node-inspector
```

步骤 3：开始调试你的 node 应用程序

```
$ node --debug-brk your/short/node/script.js
```

步骤 4：在 Chrome 浏览器中打开 <http://127.0.0.1:8080/?port=5858>。你将看到一个带有你的 nodejs 应用程序源代码的 Chrome 开发者工具界面，源代码显示在左侧面板。由于我们在调试应用时使用了调试断点选项，代码执行将在第一行代码处停止。



步骤 5：这是简单的部分，你切换到性能分析 (profiling) 标签页并开始分析应用程序。如果你想获取某个特定方法或流程的性能分析，确保代码执行在该代码片段执行之前被断点暂停。

Chapter 93: Getting started with Nodes profiling

The aim of this post is to get started with profiling nodejs application and how to make sense of this results to capture a bug or a memory leak. A nodejs running application is nothing but a v8 engine processes which is in many terms similar to a website running on a browser and we can basically capture all the metrics which are related to a website process for a node application.

The tool of my preference is chrome devtools or chrome inspector coupled with the node-inspector.

Section 93.1: Profiling a simple node application

Step 1 : Install the node-inspector package using npm globally on you machine

```
$ npm install -g node-inspector
```

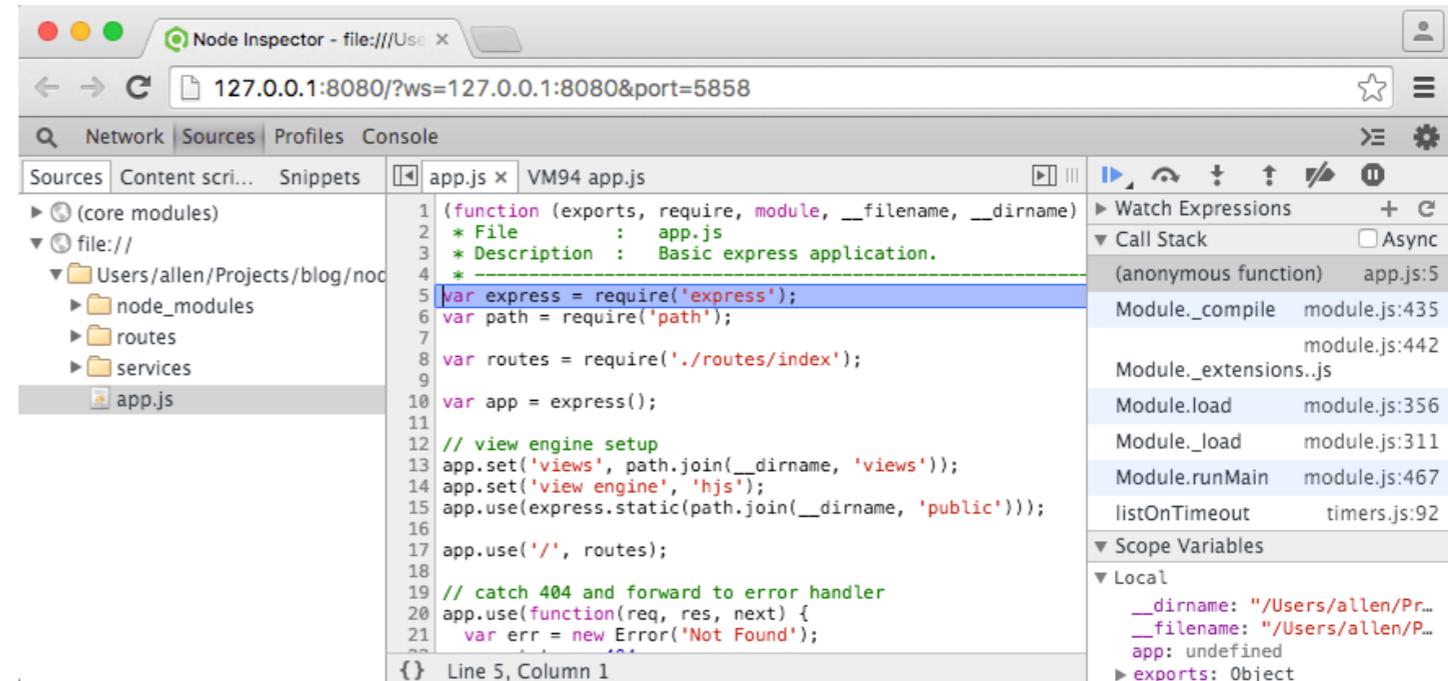
Step 2 : Start the node-inspector server

```
$ node-inspector
```

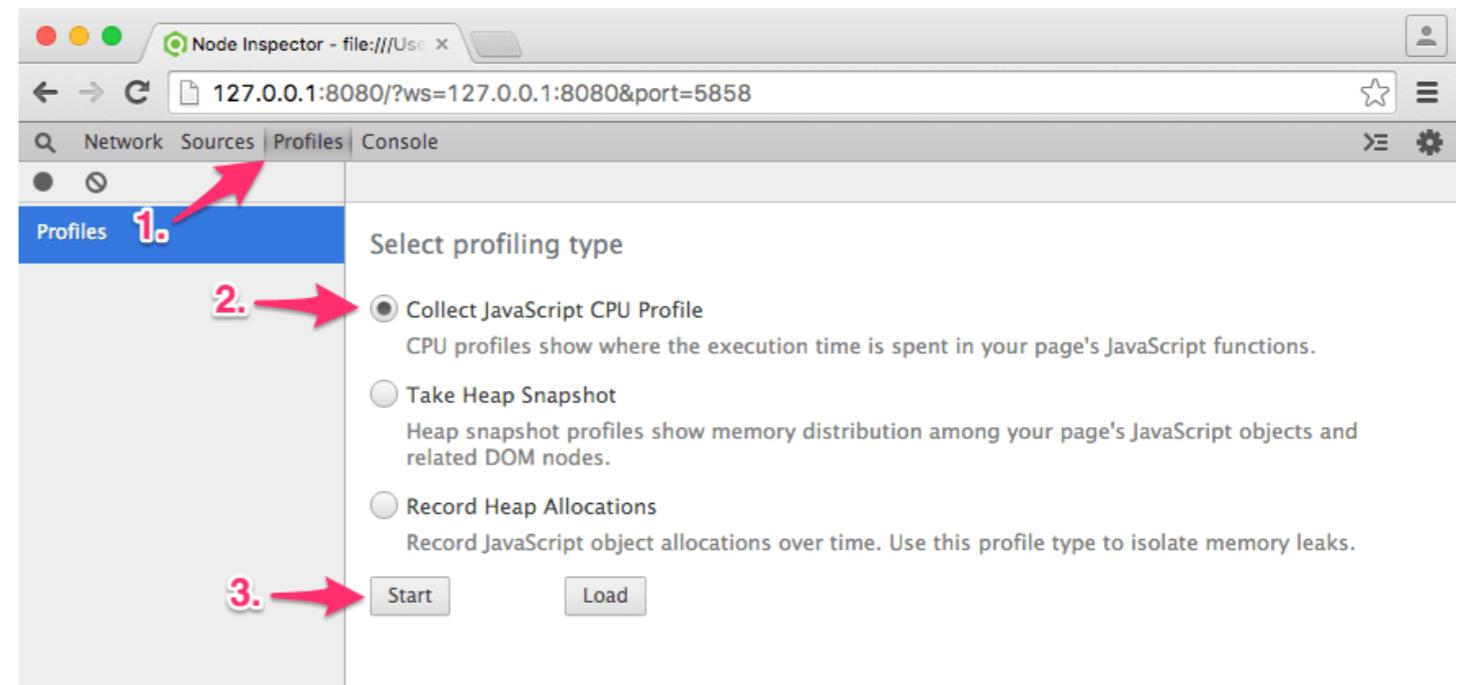
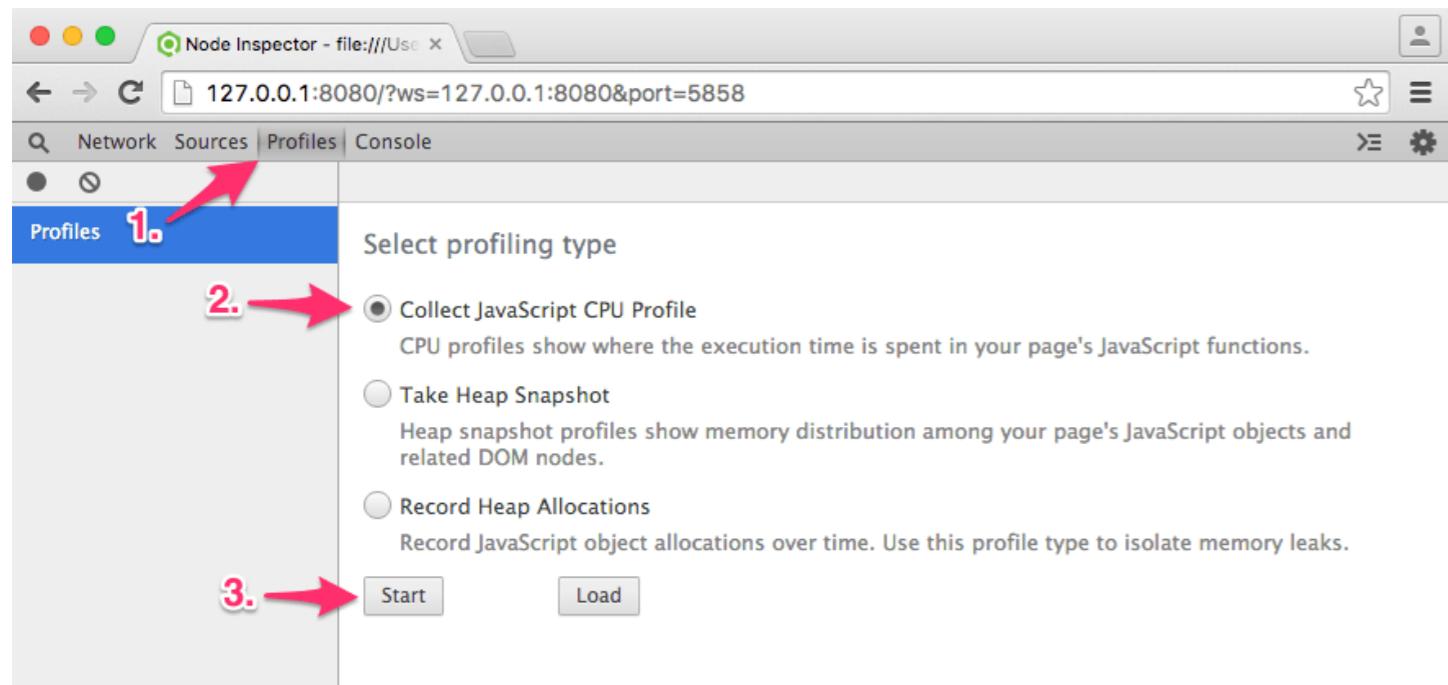
Step 3 : Start debugging your node application

```
$ node --debug-brk your/short/node/script.js
```

Step 4 : Open <http://127.0.0.1:8080/?port=5858> in the Chrome browser. And you will see a chrom-dev tools interface with your nodejs application source code in left panel . And since we have used debug break option while debugging the application the code execution will stop at the first line of code.



Step 5 : This is the easy part where you switch to the profiling tab and start profiling the application . In case you want get the profile for a particular method or flow make sure the code execution is break-pointed just before that piece of code is executed.



步骤 6：一旦你录制了 CPU 性能分析、堆转储/快照或堆分配，你可以在同一窗口查看结果，或者保存到本地磁盘以便后续分析或与其他性能分析进行比较。

你可以使用这篇文章了解如何阅读性能分析：

- [阅读 CPU 性能分析](#)
- [Chrome CPU 分析器和堆分析器](#)

Step 6 : Once you have recorded your CPU profile or heap dump/snapshot or heap allocation you can then view the results in the same window or save them to local drive for later analysis or comparison with other profiles.

You can use this articles to know how to read the profiles :

- [Reading CPU Profiles](#)
- [Chrome CPU profiler and Heap profiler](#)

第94章：Node.js性能

第94.1节：启用gzip

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

http.createServer((request, response) => {
  const stream = fs.createReadStream('index.html')
  const acceptsEncoding = request.headers['accept-encoding']

  let encoder = {
    hasEncoder: false,
    contentEncoding: {},
    createEncoder: () => throw '没有编码器'
  }

  if (!acceptsEncoding) {
    acceptEncoding = ''
  }

  if (acceptsEncoding.match(/\bdeflate\b/)) {
    encoder = {
      hasEncoder: true,
      contentEncoding: { 'content-encoding': 'deflate' },
      createEncoder: zlib.createDeflate
    }
  } else if (acceptsEncoding.match(/\bgzip\b/)) {
    encoder = {
      hasEncoder: true,
      contentEncoding: { 'content-encoding': 'gzip' },
      createEncoder: zlib.createGzip
    }
  }

  response.writeHead(200, encoder.contentEncoding)

  if (encoder.hasEncoder) {
    stream = stream.pipe(encoder.createEncoder())
  }

  stream.pipe(response)
}).listen(1337)
```

第94.2节：事件循环

阻塞操作示例

```
let loop = (i, max) => {
  while (i < max) i++
  return i
}

// 该操作将阻塞 Node.js
// 因为它是 CPU 密集型的
// 你应该对这种代码保持谨慎
loop(0, 1e+12)
```

Chapter 94: Node.js Performance

Section 94.1: Enable gzip

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

http.createServer((request, response) => {
  const stream = fs.createReadStream('index.html')
  const acceptsEncoding = request.headers['accept-encoding']

  let encoder = {
    hasEncoder: false,
    contentEncoding: {},
    createEncoder: () => throw 'There is no encoder'
  }

  if (!acceptsEncoding) {
    acceptEncoding = ''
  }

  if (acceptsEncoding.match(/\bdeflate\b/)) {
    encoder = {
      hasEncoder: true,
      contentEncoding: { 'content-encoding': 'deflate' },
      createEncoder: zlib.createDeflate
    }
  } else if (acceptsEncoding.match(/\bgzip\b/)) {
    encoder = {
      hasEncoder: true,
      contentEncoding: { 'content-encoding': 'gzip' },
      createEncoder: zlib.createGzip
    }
  }

  response.writeHead(200, encoder.contentEncoding)

  if (encoder.hasEncoder) {
    stream = stream.pipe(encoder.createEncoder())
  }

  stream.pipe(response)
}).listen(1337)
```

Section 94.2: Event Loop

Blocking Operation Example

```
let loop = (i, max) => {
  while (i < max) i++
  return i
}

// This operation will block Node.js
// Because, it's CPU-bound
// You should be careful about this kind of code
loop(0, 1e+12)
```

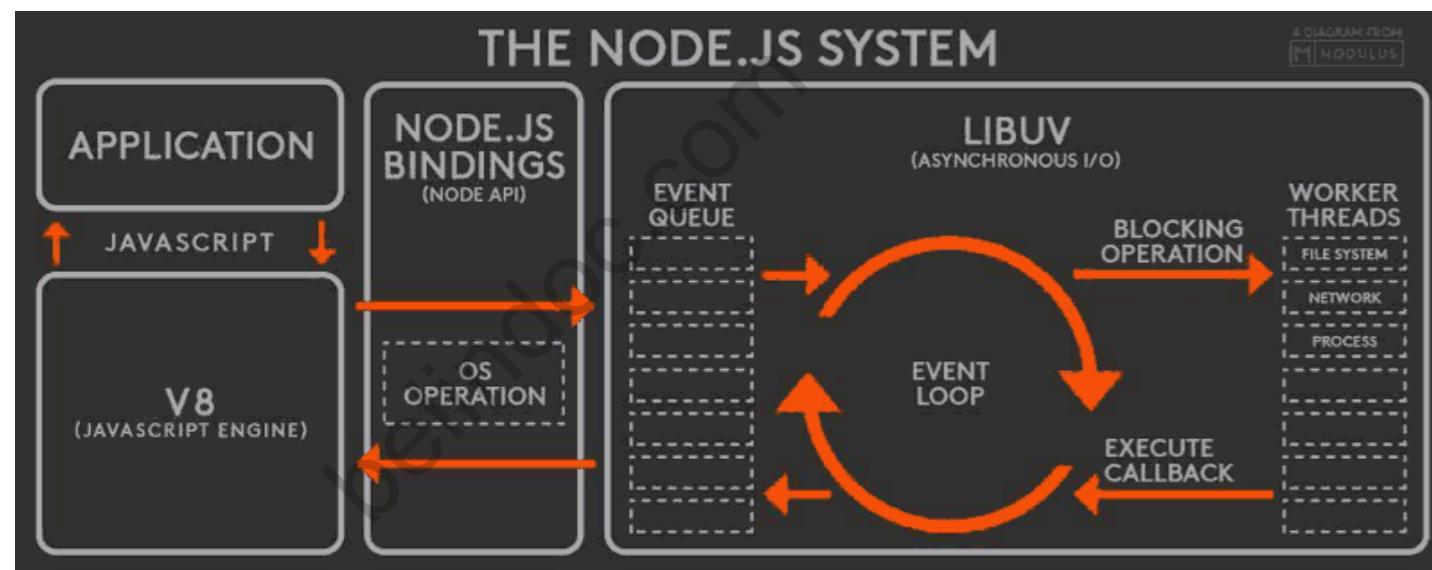
非阻塞IO操作示例

```
let i = 0

const step = max => {
  while (i < max) i++
  console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// 这将把tick运行step的while循环推迟到事件循环周期
// 任何其他IO绑定操作 (如文件系统读取) 都可以并行进行
//
tick(1e+6)
tick(1e+7)
console.log('这将在所有tick操作之前输出。i = %d', i)
console.log('因为tick操作将被推迟')
tick(1e+8)
```



简单来说，事件循环是一种单线程队列机制，它会执行你的CPU密集型代码直到执行结束，并以非阻塞的方式执行IO密集型代码。

然而，Node.js 在底层通过libuv库为其部分操作使用了多线程。

性能考虑

- 非阻塞操作不会阻塞队列，也不会影响循环的性能。
- 但是，CPU密集型操作会阻塞队列，因此你应当小心避免在Node.js代码中执行CPU密集型操作。

Node.js之所以非阻塞IO，是因为它将工作卸载给操作系统内核，当IO操作提供数据（作为一个事件）时，会通过你提供的回调通知你的代码。

第94.3节：增加maxSockets

基础知识

```
require('http').globalAgent.maxSockets = 25

// 你可以通过试验将25改为Infinity或其他值
```

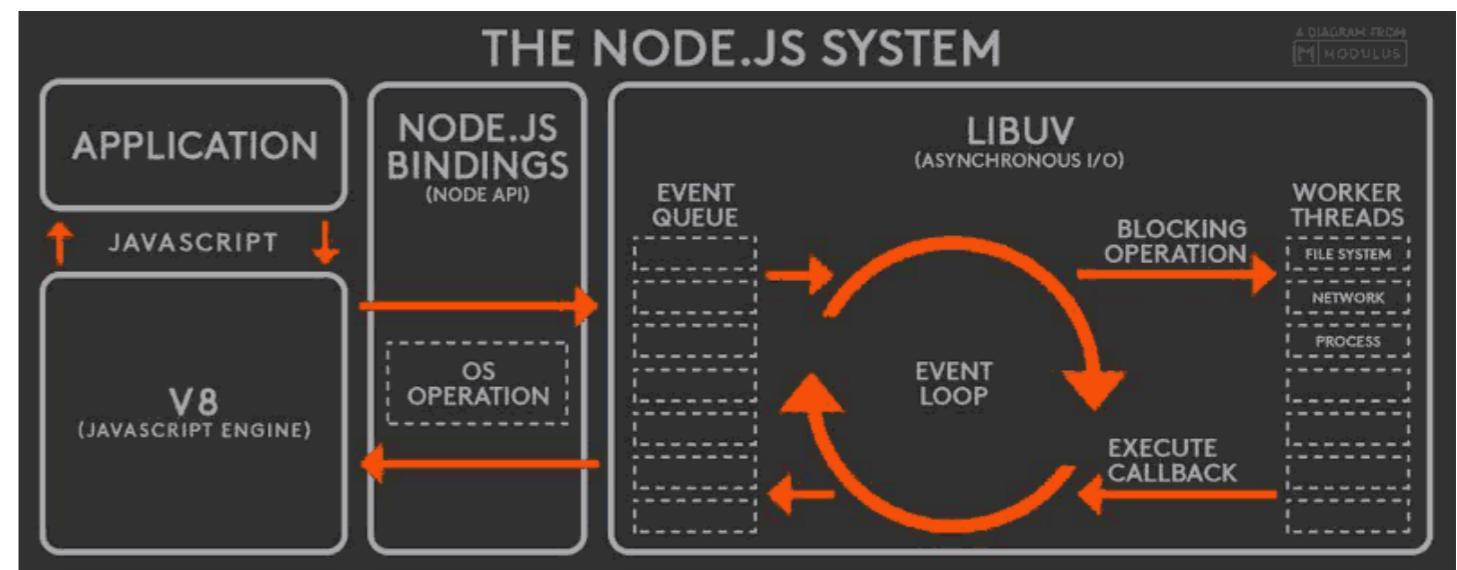
Non-Blocking IO Operation Example

```
let i = 0

const step = max => {
  while (i < max) i++
  console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// this will postpone tick run step's while-loop to event loop cycles
// any other IO-bound operation (like filesystem reading) can take place
// in parallel
tick(1e+6)
tick(1e+7)
console.log('this will output before all of tick operations. i = %d', i)
console.log('because tick operations will be postponed')
tick(1e+8)
```



In simpler terms, Event Loop is a single-threaded queue mechanism which executes your CPU-bound code until end of its execution and IO-bound code in a non-blocking fashion.

However, Node.js under the carpet uses multi-threading for some of its operations through [libuv](#) Library.

Performance Considerations

- Non-blocking operations will not block the queue and will not effect the performance of the loop.
- However, CPU-bound operations will block the queue, so you should be careful not to do CPU-bound operations in your Node.js code.

Node.js non-blocks IO because it offloads the work to the operating system kernel, and when the IO operation supplies data (as an event), it will notify your code with your supplied callbacks.

Section 94.3: Increase maxSockets

Basics

```
require('http').globalAgent.maxSockets = 25

// You can change 25 to Infinity or to a different value by experimenting
```

Node.js 默认同时使用 `maxSockets = Infinity` (自 v0.12.0 起)。在 Node v0.12.0 之前, 默认值是 `maxSockets = 5` (参见 v0.11.0)。因此, 超过 5 个请求后, 它们将被排队。如果你想要并发处理, 请增加这个数值。

设置你自己的代理

http API 使用一个 "全局代理"。你可以提供你自己的代理。如下所示：

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity

http.request({ ..., agent: myGloriousAgent }, ...)
```

完全关闭 Socket 池

```
const http = require('http')
const options = {....}

options.agent = false

const request = http.request(options)
```

注意事项

- 如果你想要相同的效果, 应该对 https API 做同样的设置
- 注意, 例如, AWS 会使用 50 而不是 Infinity。

Node.js by default is using `maxSockets = Infinity` at the same time (since v0.12.0). Until Node v0.12.0, the default was `maxSockets = 5` (see v0.11.0). So, after more than 5 requests they will be queued. If you want concurrency, increase this number.

Setting your own agent

http API is using a "[Global Agent](#)". You can supply your own agent. Like this:

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity

http.request({ ..., agent: myGloriousAgent }, ...)
```

Turning off Socket Pooling entirely

```
const http = require('http')
const options = {....}

options.agent = false
```

```
const request = http.request(options)
```

Pitfalls

- You should do the same thing for https API if you want the same effects
- Beware that, for example, AWS will use 50 instead of Infinity.

第95章：Yarn 包管理器

[Yarn](#)是一个用于Node.js的包管理器，类似于npm。虽然两者有很多共同点，但Yarn和npm之间存在一些关键的区别。

第95.1节：创建一个基本包

yarn init命令将引导你创建一个package.json文件，以配置有关你的包的一些信息。这类似于npm中的npm init命令。

创建并进入一个新目录以存放你的包，然后运行yarn init

```
mkdir my-package && cd my-package  
yarn init
```

在命令行界面中回答接下来的问题

```
问题名称(my-package): my-package  
问题版本(1.0.0):  
问题描述: 一个测试包  
问题入口点(index.js):  
问题仓库网址:  
问题作者: StackOverflow Documentation  
问题许可证(MIT):  
成功保存包.json  
□完成用时27.31秒。
```

这将生成一个类似于以下内容的package.json文件

```
{  
  "name": "my-package",  
  "version": "1.0.0",  
  "description": "一个测试包",  
  "main": "index.js",  
  "author": "StackOverflow Documentation",  
  "license": "MIT"  
}
```

现在让我们尝试添加一个依赖。基本语法是 yarn add [package-name]

运行以下命令安装ExpressJS

```
yarn add express
```

这将在你的package.json中添加一个dependencies部分，并添加ExpressJS

```
"dependencies": {  
  "express": "^4.15.2"  
}
```

第95.2节：Yarn安装

本示例说明了针对您的操作系统安装 Yarn 的不同方法。

macOS

Chapter 95: Yarn Package Manager

[Yarn](#) is a package manager for Node.js, similar to npm. While sharing a lot of common ground, there are some key differences between Yarn and npm.

Section 95.1: Creating a basic package

The `yarn init` command will walk you through the creation of a package.json file to configure some information about your package. This is similar to the `npm init` command in npm.

Create and navigate to a new directory to hold your package, and then run `yarn init`

```
mkdir my-package && cd my-package  
yarn init
```

Answer the questions that follow in the CLI

```
question name (my-package): my-package  
question version (1.0.0):  
question description: A test package  
question entry point (index.js):  
question repository url:  
question author: StackOverflow Documentation  
question license (MIT):  
success Saved package.json  
□ Done in 27.31s.
```

This will generate a package.json file similar to the following

```
{  
  "name": "my-package",  
  "version": "1.0.0",  
  "description": "A test package",  
  "main": "index.js",  
  "author": "StackOverflow Documentation",  
  "license": "MIT"  
}
```

Now lets try adding a dependency. The basic syntax for this is `yarn add [package-name]`

Run the following to install ExpressJS

```
yarn add express
```

This will add a dependencies section to your package.json, and add ExpressJS

```
"dependencies": {  
  "express": "^4.15.2"  
}
```

Section 95.2: Yarn Installation

This example explains the different methods to install Yarn for your OS.

macOS

Homebrew

```
brew update  
brew install yarn
```

MacPorts

```
sudo port install yarn
```

将 Yarn 添加到您的 PATH

将以下内容添加到您首选的 shell 配置文件 (.profile、.bashrc、.zshrc 等) 中

```
export PATH="$PATH:`yarn global bin`"
```

Windows

安装程序

首先，如果尚未安装 Node.js，请先安装。

从Yarn 网站下载 Yarn 安装程序，格式为.msi。

Chocolatey

```
choco install yarn
```

Linux

Debian / Ubuntu

确保为您的发行版安装了 Node.js，或者运行以下命令

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

配置 YarnPkg 仓库

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -  
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list
```

安装 Yarn

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

如果尚未安装 Node.js，请先安装

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

安装 Yarn

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo  
sudo yum install yarn
```

Arch

通过 AUR 安装 Yarn。

Homebrew

```
brew update  
brew install yarn
```

MacPorts

```
sudo port install yarn
```

Adding Yarn to your PATH

Add the following to your preferred shell profile (.profile, .bashrc, .zshrc etc)

```
export PATH="$PATH:`yarn global bin`"
```

Windows

Installer

First, install Node.js if it is not already installed.

Download the Yarn installer as an .msi from the [Yarn website](#).

Chocolatey

```
choco install yarn
```

Linux

Debian / Ubuntu

Ensure Node.js is installed for your distro, or run the following

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

Configure the YarnPkg repository

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -  
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list
```

Install Yarn

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

Install Node.js if not already installed

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

Install Yarn

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo  
sudo yum install yarn
```

Arch

Install Yarn via AUR.

使用 yaourt 的示例：

```
yaourt -S yarn
```

Solus

```
sudo eopkg install yarn
```

所有发行版

将以下内容添加到您首选的 shell 配置文件 (.profile、.bashrc、.zshrc 等) 中

```
export PATH="$PATH:`yarn global bin`"
```

替代安装方法

Shell 脚本

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

或指定版本进行安装

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

压缩包

```
cd /opt  
wget https://yarnpkg.com/latest.tar.gz  
tar zvxf latest.tar.gz
```

Npm

如果你已经安装了 npm，只需运行

```
npm install -g yarn
```

安装后操作

通过运行以下命令检查已安装的 Yarn 版本

```
yarn --version
```

第 95.3 节：使用 Yarn 安装包

Yarn 使用与 npm 相同的注册表。这意味着在 npm 上可用的每个包在 Yarn 上也是相同的。

要安装一个包，运行 `yarn add package`。

如果你需要特定版本的包，可以使用 `yarn add package@version`。

如果你需要安装的版本被打了标签，可以使用 `yarn add package@tag`。

Example using yaourt:

```
yaourt -S yarn
```

Solus

```
sudo eopkg install yarn
```

All Distributions

Add the following to your preferred shell profile (.profile, .bashrc, .zshrc etc)

```
export PATH="$PATH:`yarn global bin`"
```

Alternative Method of Installation

Shell script

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

or specify a version to install

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

Tarball

```
cd /opt  
wget https://yarnpkg.com/latest.tar.gz  
tar zvxf latest.tar.gz
```

Npm

If you already have npm installed, simply run

```
npm install -g yarn
```

Post Install

Check the installed version of Yarn by running

```
yarn --version
```

Section 95.3: Install package with Yarn

Yarn uses the same registry that npm does. That means that every package that is available on npm is the same on Yarn.

To install a package, run `yarn add package`.

If you need a specific version of the package, you can use `yarn add package@version`.

If the version you need to install has been tagged, you can use `yarn add package@tag`.

第96章：OAuth 2.0

第96.1节：使用Redis实现OAuth 2 - grant_type: password

在此示例中，我将使用带有Redis数据库的REST API中的oauth2

重要提示：您需要在您的机器上安装Redis数据库，Linux用户请从 [here](#) 下载，Windows版本请从 [here](#) 安装，我们将使用Redis管理桌面应用程序，请从 [here](#) 安装。

现在我们必须设置我们的Node.js服务器以使用Redis数据库。

• 创建服务器文件：app.js

```
var express = require('express'),
bodyParser = require('body-parser'),
oauthserver = require('oauth2-server'); // 应为：'oauth2-server'

var app = express();

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

app.oauth = oauthserver({
  model: require('./routes/Oauth2/model'),
  grants: ['password', 'refresh_token'],
  debug: true
});

// 处理令牌授权请求
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
  // 需要有效的 access_token
  res.send('秘密区域');
});

app.get('/public', function (req, res) {
  // 不需要 access_token
  res.send('公共区域');
});

// 错误处理
app.use(app.oauth.errorHandler());

app.listen(3000);
```

• 在 routes/Oauth2/model.js 中创建 OAuth2 模型

```
var model = module.exports,
util = require('util'),
```

Chapter 96: OAuth 2.0

Section 96.1: OAuth 2 with Redis Implementation - grant_type: password

In this example I will be using oauth2 in rest api with redis database

Important: You will need to install redis database on your machine, Download it from [here](#) for linux users and from [here](#) to install windows version, and we will be using redis manager desktop app, install it from [here](#).

Now we have to set our node.js server to use redis database.

• Creating Server file: app.js

```
var express = require('express'),
bodyParser = require('body-parser'),
oauthserver = require('oauth2-server'); // Would be: 'oauth2-server'

var app = express();

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

app.oauth = oauthserver({
  model: require('./routes/Oauth2/model'),
  grants: ['password', 'refresh_token'],
  debug: true
});

// Handle token grant requests
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
  // Will require a valid access_token
  res.send('Secret area');
});

app.get('/public', function (req, res) {
  // Does not require an access_token
  res.send('Public area');
});

// Error handling
app.use(app.oauth.errorHandler());

app.listen(3000);
```

• Create OAuth2 model in routes/Oauth2/model.js

```
var model = module.exports,
util = require('util'),
```

```

redis = require('redis');

var db = redis.createClient();

var keys = {
  token: 'tokens:%s',
  client: 'clients:%s',
  refreshToken: 'refresh_tokens:%s',
  grantTypes: 'clients:%s:grant_types',
  user: 'users:%s'
};

model.getAccessToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.token, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      accessToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.getClient = function (clientId, clientSecret, callback) {
  db.hgetall(util.format(keys.client, clientId), function (err, client) {
    if (err) return callback(err);

    if (!client || client.clientSecret !== clientSecret) return callback();

    callback(null, {
      clientId: client.clientId,
      clientSecret: client.clientSecret
    });
  });
};

model.getRefreshToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.refreshToken, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      refreshToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.grantTypeAllowed = function (clientId, grantType, callback) {
  db.sismember(util.format(keys.grantTypes, clientId), grantType, callback);
};

model.saveAccessToken = function (accessToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.token, accessToken), {
    accessToken: accessToken,

```

```

    redis = require('redis');

    var db = redis.createClient();

    var keys = {
      token: 'tokens:%s',
      client: 'clients:%s',
      refreshToken: 'refresh_tokens:%s',
      grantTypes: 'clients:%s:grant_types',
      user: 'users:%s'
    };

    model.getAccessToken = function (bearerToken, callback) {
      db.hgetall(util.format(keys.token, bearerToken), function (err, token) {
        if (err) return callback(err);

        if (!token) return callback();

        callback(null, {
          accessToken: token.accessToken,
          clientId: token.clientId,
          expires: token.expires ? new Date(token.expires) : null,
          userId: token.userId
        });
      });
    };

    model.getClient = function (clientId, clientSecret, callback) {
      db.hgetall(util.format(keys.client, clientId), function (err, client) {
        if (err) return callback(err);

        if (!client || client.clientSecret !== clientSecret) return callback();

        callback(null, {
          clientId: client.clientId,
          clientSecret: client.clientSecret
        });
      });
    };

    model.getRefreshToken = function (bearerToken, callback) {
      db.hgetall(util.format(keys.refreshToken, bearerToken), function (err, token) {
        if (err) return callback(err);

        if (!token) return callback();

        callback(null, {
          refreshToken: token.accessToken,
          clientId: token.clientId,
          expires: token.expires ? new Date(token.expires) : null,
          userId: token.userId
        });
      });
    };

    model.grantTypeAllowed = function (clientId, grantType, callback) {
      db.sismember(util.format(keys.grantTypes, clientId), grantType, callback);
    };

    model.saveAccessToken = function (accessToken, clientId, expires, user, callback) {
      db.hmset(util.format(keys.token, accessToken), {
        accessToken: accessToken,

```

```

clientId: clientId,
expires: expires ? expires.toISOString() : null,
  userId: user.id
}, callback);
};

model.saveRefreshToken = function (refreshToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.refreshToken, refreshToken), {
    refreshToken: refreshToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.getUser = function (username, password, callback) {
  db.hgetall(util.format(keys.user, username), function (err, user) {
    if (err) return callback(err);

    if (!user || password !== user.password) return callback();

    callback(null, {
      id: username
    });
  });
};

```

你只需要在你的机器上安装redis并运行以下node文件

```

#!/usr/bin/env node

var db = require('redis').createClient();

db.multi()
.hmset('users:username', {
  id: 'username',
  username: 'username',
  password: 'password'
})
.hmset('clients:client', {
  clientId: 'client',
  clientSecret: 'secret'
})//clientId + clientSecret to base 64 will generate Y2xpZW50OnNlY3JldA==
.sadd('clients:client:grant_types', [
  'password',
  'refresh_token'
])
.exec(function (errs) {
  if (errs) {
    console.error(errs[0].message);
    return process.exit(1);
  }

  console.log('Client and user added successfully');
  process.exit();
});

console.log('Client and user added successfully');
process.exit();
});

```

注意：此文件将为您的前端设置凭证以请求令牌，因此在调用上述文件后，您从示例 Redis 数据库的请求如下：

```

clientId: clientId,
expires: expires ? expires.toISOString() : null,
  userId: user.id
}, callback);
};

model.saveRefreshToken = function (refreshToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.refreshToken, refreshToken), {
    refreshToken: refreshToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.getUser = function (username, password, callback) {
  db.hgetall(util.format(keys.user, username), function (err, user) {
    if (err) return callback(err);

    if (!user || password !== user.password) return callback();

    callback(null, {
      id: username
    });
  });
};

```

You only need to install redis on your machine and run the following node file

```

#!/usr/bin/env node

var db = require('redis').createClient();

db.multi()
.hmset('users:username', {
  id: 'username',
  username: 'username',
  password: 'password'
})
.hmset('clients:client', {
  clientId: 'client',
  clientSecret: 'secret'
})//clientId + clientSecret to base 64 will generate Y2xpZW50OnNlY3JldA==
.sadd('clients:client:grant_types', [
  'password',
  'refresh_token'
])
.exec(function (errs) {
  if (errs) {
    console.error(errs[0].message);
    return process.exit(1);
  }

  console.log('Client and user added successfully');
  process.exit();
});

```

Note: This file will set credentials for your frontend to request token So your request from Sample redis database after calling the above file:

Redis Desktop Manager v.0.8.8.384

local

- db0 (3/3)
 - clients (2)
 - client (1)
 - clients:client:grant_types
 - users (1)
 - users:username
- db1 (0)
- db2 (0)
- db3 (0)
- db4 (0)
- db5 (0)
- db6 (0)
- db7 (0)
- db8 (0)
- db9 (0)
- db10 (0)
- db11 (0)
- db12 (0)
- db13 (0)
- db14 (0)
- db15 (0)

local::db0::users:username

HASH: users:username

row	key	value
1	id	username
2	username	username
3	password	password

Add row **Delete row** **Reload Value**

Page 1 of 1 **Set Page**

Key: size in bytes: 0 **View as:** Plain Text

Value: size in bytes: 0 **View as:** Plain Text

Save

2017-03-28 18:16:02 : Connection: local > Response received :
 2017-03-28 18:16:02 : Connection: local > [runCommand] HLEN users:username
 2017-03-28 18:16:02 : Connection: local > Response received :
 2017-03-28 18:16:02 : Connection: local > [runCommand] HSCAN users:username 0 COUNT 10000
 2017-03-28 18:16:02 : Connection: local > Response received : Array

Import / Export **Connect to Redis Server** **System log**

请求将如下所示：

调用 API 示例

ARC

Request

HTTP request > http://localhost:3000/oauth/token

Socket

History

Saved

Projects

Raw headers Headers form Headers sets Variables

authorization: Basic Y2xpZW50OnNlY3JldA==

Content-Type: application/x-www-form-urlencoded

ADD HEADER

Raw payload Data form Files

ENCODE PAYLOAD DECODE PAYLOAD

Form data for x-www-form-urlencoded parameters

username	username
password	password
grant_type	password

89 bytes

Selected environment: default

请求头：

1. 授权：Basic 后跟您首次设置 Redis 时设定的密码：

a. clientId + secretId 转为 base64

Redis Desktop Manager v.0.8.8.384

local

- db0 (3/3)
 - clients (2)
 - client (1)
 - clients:client:grant_types
 - users (1)
 - users:username
- db1 (0)
- db2 (0)
- db3 (0)
- db4 (0)
- db5 (0)
- db6 (0)
- db7 (0)
- db8 (0)
- db9 (0)
- db10 (0)
- db11 (0)
- db12 (0)
- db13 (0)
- db14 (0)
- db15 (0)

local::db0::users:username

HASH: users:username

row	key	value
1	id	username
2	username	username
3	password	password

Add row **Delete row** **Reload Value**

Page 1 of 1 **Set Page**

Key: size in bytes: 0 **View as:** Plain Text

Value: size in bytes: 0 **View as:** Plain Text

Save

2017-03-28 18:16:02 : Connection: local > Response received :
 2017-03-28 18:16:02 : Connection: local > [runCommand] HLEN users:username
 2017-03-28 18:16:02 : Connection: local > Response received :
 2017-03-28 18:16:02 : Connection: local > [runCommand] HSCAN users:username 0 COUNT 10000
 2017-03-28 18:16:02 : Connection: local > Response received : Array

Import / Export **Connect to Redis Server** **System log**

Request will be as follows:

Sample Call to api

ARC

Request

HTTP request > http://localhost:3000/oauth/token

Socket

History

Saved

Projects

Raw headers Headers form Headers sets Variables

authorization: Basic Y2xpZW50OnNlY3JldA==

Content-Type: application/x-www-form-urlencoded

ADD HEADER

Raw payload Data form Files

ENCODE PAYLOAD DECODE PAYLOAD

Form data for x-www-form-urlencoded parameters

username	username
password	password
grant_type	password

89 bytes

Selected environment: default

Header:

1. authorization: Basic followed by the password set when you first setup redis:

a. clientId + secretId to base64

2. 数据格式：

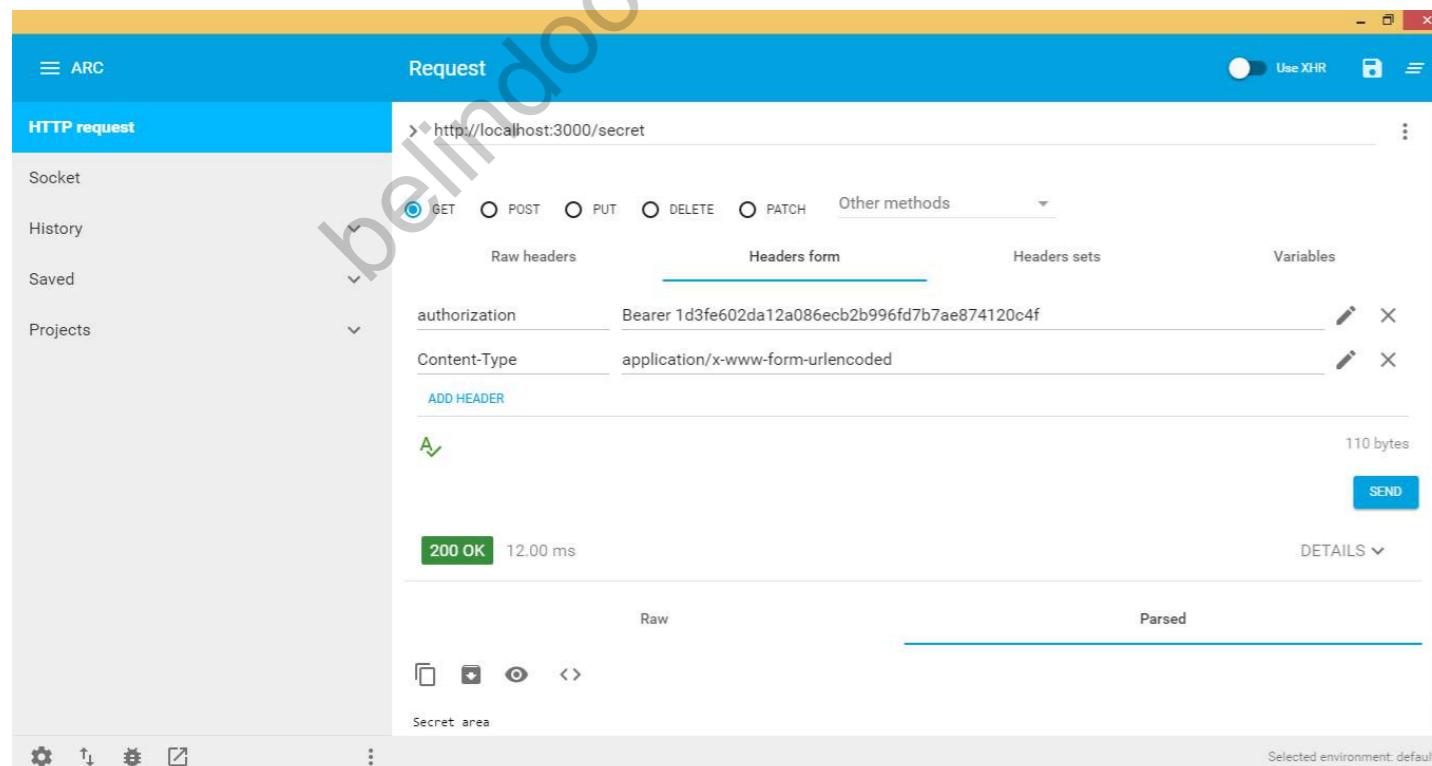
username：请求令牌的用户

password：用户密码

grant_type：取决于你想要的选项，我选择了 password，它只需要用户名和密码来创建 redis 中的数据，redis 中的数据如下：

```
{  
  "access_token": "1d3fe602da12a086ecb2b996fd7b7ae874120c4f",  
  "token_type": "bearer", // 将用于访问 API + access+token, 例如 bearer  
1d3fe602da12a086ecb2b996fd7b7ae874120c4f  
  "expires_in": 3600,  
  "refresh_token": "b6ad56e5c9aba63c85d7e21b1514680bbf711450"  
}
```

所以我们需要调用我们的 API，并使用刚创建的访问令牌获取一些受保护的数据，见下文：



当令牌过期时，API 会抛出令牌过期的错误，您将无法访问任何 API 调用，见下图：

2. Data form:

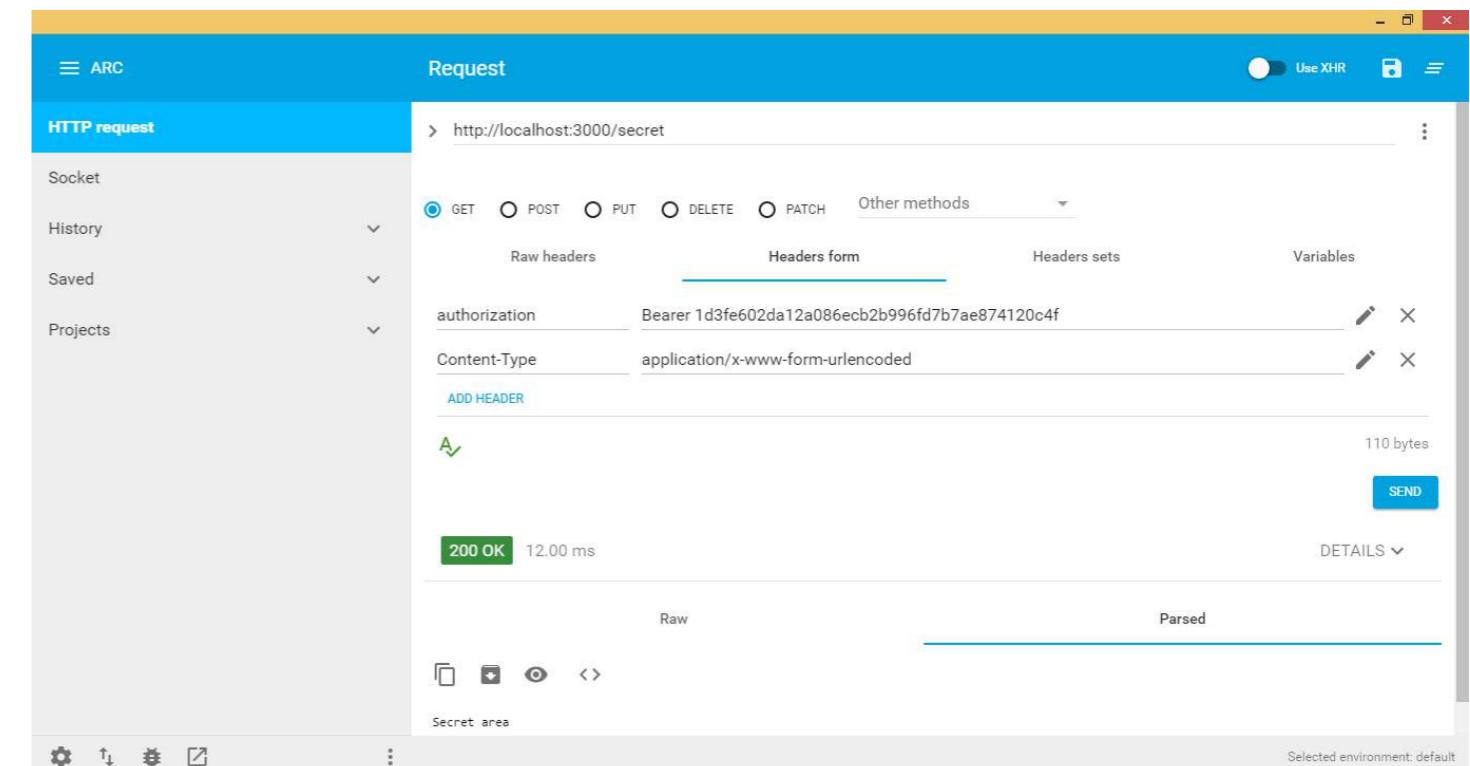
username: user that request token

password: user password

grant_type: depends on what options do you want, I choose passwod which takes only username and password to be created in redis, Data on redis will be as below:

```
{  
  "access_token": "1d3fe602da12a086ecb2b996fd7b7ae874120c4f",  
  "token_type": "bearer", // Will be used to access api + access+token e.g. bearer  
1d3fe602da12a086ecb2b996fd7b7ae874120c4f  
  "expires_in": 3600,  
  "refresh_token": "b6ad56e5c9aba63c85d7e21b1514680bbf711450"  
}
```

So We need to call our api and grab some secured data with our access token we have just created, see below:



when token expires api will throw an error that the token expires and you cannot have access to any of the api calls, see image below :

ARC

HTTP request

GET POST PUT DELETE PATCH Other methods

Raw headers Headers form Headers sets Variables

authorization: Bearer 1d3fe602da12a086ecb2b996fd7b7ae874120c4f
Content-Type: application/x-www-form-urlencoded

110 bytes

401 Unauthorized 9.00 ms DETAILS

Raw JSON

{
 "code": 401,
 "error": "invalid_token",
 "error_description": "The access token provided has expired."
}

Selected environment: default

让我们看看令牌过期时该怎么办，先给您解释一下，如果访问令牌过期，redis 中会存在一个 refresh_token，它引用了已过期的访问令牌。我们需要做的是使用 refresh_token 作为 grant_type 再次调用 oauth/token，并将授权设置为 BasicclientId:clientsecret（进行 Base64 编码！），最后发送 refresh_token，这将生成一个带有新过期时间的新访问令牌。

下图展示了如何获取新的访问令牌：

ARC

Request

HTTP request

POST GET PUT DELETE PATCH Other methods application/x-www-form-urlencoded

Raw headers Headers form Headers sets Variables

authorization: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

ADD HEADER

A Raw payload Data form Files

89 bytes

ENCODE PAYLOAD DECODE PAYLOAD

Form data for x-www-form-urlencoded parameters

refresh_token: b6ad56e5c9aba63c85d7e21b1514680bbf711450
grant_type: refresh_token

ADD ANOTHER PARAMETER

Selected environment: default

希望能帮到您！

ARC

HTTP request

GET POST PUT DELETE PATCH Other methods

Raw headers Headers form Headers sets Variables

authorization: Bearer 1d3fe602da12a086ecb2b996fd7b7ae874120c4f
Content-Type: application/x-www-form-urlencoded

110 bytes

401 Unauthorized 9.00 ms DETAILS

Raw JSON

{
 "code": 401,
 "error": "invalid_token",
 "error_description": "The access token provided has expired."
}

Selected environment: default

Lets see what to do if the token expires, Let me first explain it to you, if access token expires a refresh_token exists in redis that reference the expired access_token So what we need is to call oauth/token again with the refresh_token grant_type and set the authorization to the Basic clientId:clientsecret (to base 64 !) and finally send the refresh_token, this will generate a new access_token with a new expiry data.

The following picture shows how to get a new access token:

ARC

Request

HTTP request

POST GET PUT DELETE PATCH Other methods application/x-www-form-urlencoded

Raw headers Headers form Headers sets Variables

authorization: Basic Y2xpZW50OnNlY3JldA==
Content-Type: application/x-www-form-urlencoded

ADD HEADER

A Raw payload Data form Files

89 bytes

ENCODE PAYLOAD DECODE PAYLOAD

Form data for x-www-form-urlencoded parameters

refresh_token: b6ad56e5c9aba63c85d7e21b1514680bbf711450
grant_type: refresh_token

ADD ANOTHER PARAMETER

Selected environment: default

Hope to Help!

第97章：Node JS 本地化

维护 Node.js Express 本地化非常简单

第97.1节：使用 i18n 模块维护

Node.js 应用的本地化

轻量级简单的翻译模块，支持动态 JSON 存储。支持纯净的 Node.js 应用，并且应该兼容任何暴露 app.use() 方法并传入 res 和 req 对象的框架（如 Express、Restify 及可能更多）。在应用和模板中使用常见的 `__(...)` 语法。将语言文件存储为兼容 webtranslateit JSON 格式的 JSON 文件。首次在应用中使用时会动态添加新字符串，无需额外解析。

express + i18n-node + cookieParser，避免并发问题

```
// 常规要求
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
    // 设置一些语言环境 - 其他语言环境默认为英文
    locales: ['en', 'ru', 'de'],

    // 设置自定义的 cookie 名称以解析语言环境设置
    cookie: 'yourcookiename',

    // 存储 json 文件的位置 - 默认为 './locales'
    directory: __dirname + '/locales'
});

app.configure(function () {
    // 你需要使用 cookieParser 来将 cookie 暴露给 req.cookies
    app.use(express.cookieParser());

    // i18n 初始化解析请求中的语言头、Cookie 等。
    app.use(i18n.init());
});

// 提供主页服务
app.get('/', function (req, res) {
    res.send(res.__('Hello World'));
});

// 启动服务器
if (!module.parent) {
    app.listen(3000);
}
```

Chapter 97: Node JS Localization

Its very easy to maintain localization nodejs express

Section 97.1: using i18n module to maintains localization in node js app

Lightweight simple translation module with dynamic json storage. Supports plain vanilla node.js apps and should work with any framework (like express, restify and probably more) that exposes an app.use() method passing in res and req objects. Uses common `__(...)` syntax in app and templates. Stores language files in json files compatible to webtranslateit json format. Adds new strings on-the-fly when first used in your app. No extra parsing needed.

express + i18n-node + cookieParser and avoid concurrency issues

```
// usual requirements
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
    // setup some locales - other locales default to en silently
    locales: ['en', 'ru', 'de'],

    // sets a custom cookie name to parse locale settings from
    cookie: 'yourcookiename',

    // where to store json files - defaults to './locales'
    directory: __dirname + '/locales'
});

app.configure(function () {
    // you will need to use cookieParser to expose cookies to req.cookies
    app.use(express.cookieParser());

    // i18n init parses req for language headers, cookies, etc.
    app.use(i18n.init());
});

// serving homepage
app.get('/', function (req, res) {
    res.send(res.__('Hello World'));
});

// starting server
if (!module.parent) {
    app.listen(3000);
}
```

第98章：无停机时间部署Node.js应用程序。

第98.1节：使用PM2无停机时间部署

ecosystem.json

```
{  
  "name": "app-name",  
  "script": "server",  
  "exec_mode": "cluster",  
  "instances": 0,  
  "wait_ready": true  
  "listen_timeout": 10000,  
  "kill_timeout": 5000,  
}  
  
wait_ready
```

不再等待监听事件的重新加载，而是等待 `process.send('ready');`

listen_timeout

应用未监听时强制重新加载的时间，单位毫秒。

kill_timeout

发送最终 `SIGKILL` 信号前的等待时间，单位毫秒。

server.js

```
const http = require('http');  
const express = require('express');  
  
const app = express();  
const server = http.Server(app);  
const port = 80;  
  
server.listen(port, function() {  
  process.send('ready');  
});  
  
process.on('SIGINT', function() {  
  server.close(function() {  
    process.exit(0);  
  });  
});
```

您可能需要等待您的应用程序与您的数据库/缓存/工作进程/其他建立连接。PM2 需要等待，才能将您的应用程序视为在线。为此，您需要在进程文件中提供 `wait_ready: true`。这样 PM2 会监听该事件。在您的应用程序中，当您希望应用程序被视为已准备好时，您需要添加 `process.send('ready');`。

当进程被 PM2 停止或重启时，会按一定顺序向您的进程发送一些系统信号。

首先会向您的进程发送一个 `SIGINT` 信号，您可以捕获该信号以知道您的进程即将被停止。

Chapter 98: Deploying Node.js application without downtime.

Section 98.1: Deployment using PM2 without downtime

ecosystem.json

```
{  
  "name": "app-name",  
  "script": "server",  
  "exec_mode": "cluster",  
  "instances": 0,  
  "wait_ready": true  
  "listen_timeout": 10000,  
  "kill_timeout": 5000,  
}  
  
wait_ready
```

Instead of reload waiting for listen event, wait for `process.send('ready');`

listen_timeout

Time in ms before forcing a reload if app not listening.

kill_timeout

Time in ms before sending a final SIGKILL.

server.js

```
const http = require('http');  
const express = require('express');  
  
const app = express();  
const server = http.Server(app);  
const port = 80;  
  
server.listen(port, function() {  
  process.send('ready');  
});  
  
process.on('SIGINT', function() {  
  server.close(function() {  
    process.exit(0);  
  });  
});
```

You might need to wait for your application to have established connections with your DBs/caches/workers/whatever. PM2 needs to wait before considering your application as online. To do this, you need to provide `wait_ready: true` in a process file. This will make PM2 listen for that event. In your application you will need to add `process.send('ready');` when you want your application to be considered as ready.

When a process is stopped/restarted by PM2, some system signals are sent to your process in a given order.

First a `SIGINT` signal is sent to your processes, signal you can catch to know that your process is going to be

如果您的应用程序在 1.6 秒内（可自定义）未自行退出，它将收到一个 SIGKILL 信号以强制进程退出。因此，如果您的应用程序需要清理某些状态或任务，您可以捕获 SIGINT 信号来准备应用程序退出。

stopped. If your application does not exit by itself before 1.6s (customizable) it will receive a SIGKILL signal to force the process exit. So if your application need to clean-up something states or jobs you can catch the SIGINT signal to prepare your application to exit.

belindoc.com

第 99 章：Node.js (express.js) 与 angular.js 示例代码

本示例展示了如何创建一个基本的 express 应用程序，然后提供 AngularJS 服务。

第 99.1 节：创建我们的项目

我们准备好了，所以我们再次从控制台运行：

```
mkdir our_project  
cd our_project
```

现在我们进入代码存放的位置。要创建我们项目的主存档，可以运行

好的，但我们如何创建 express 骨架项目？

很简单：

```
npm install -g express express-generator
```

Linux 发行版和 Mac 应该使用 sudo 来安装，因为它们安装在 nodejs 目录中，该目录只有root用户可以访问。如果一切顺利，我们终于可以创建 express-app 骨架，只需运行

```
express
```

该命令将在我们的文件夹内创建一个 express 示例应用。结构如下：

```
bin/  
public/  
routes/  
views/  
app.js  
package.json
```

现在如果我们运行 npm start 并访问 <http://localhost:3000>，我们将看到 Express 应用程序正在运行，公平地说，我们生成了一个 Express 应用程序，没有太多麻烦，但我们如何将其与 AngularJS 混合使用呢？

Express 是如何工作的，简要介绍？

Express 是建立在 Nodejs 之上的一个框架，你可以在 Express 官网 查看官方文档。但就我们的目的而言，我们需要知道当我们输入例如<http://localhost:3000/home> 时，Express 负责渲染我们应用程序的主页。[从最近创建的应用程序中我们可以检查](#)：

```
文件: routes/index.js  
var express = require('express');  
var router = express.Router();  
  
/* 获取主页(GET home page) */  
router.get('/', function(req, res, next) {  
  res.render('index', { title: 'Express' });  
});  
  
module.exports = router;
```

Chapter 99: Node.js (express.js) with angular.js Sample code

This example shows how to create a basic express app and then serve AngularJS.

Section 99.1: Creating our project

We're good to go so, we run, again from console:

```
mkdir our_project  
cd our_project
```

Now we're in the place where our code will live. To create the main archive of our project you can run

Ok, but how we create the express skeleton project?

It's simple:

```
npm install -g express express-generator
```

Linux distros and Mac should use **sudo** to install this because they're installed in the nodejs directory which is only accessible by the **root** user. If everything went fine we can, finally, create the express-app skeleton, just run

```
express
```

This command will create inside our folder an express example app. The structure is as follow:

```
bin/  
public/  
routes/  
views/  
app.js  
package.json
```

Now if we run **npm start** an go to <http://localhost:3000> we'll see the express app up and running, fair enough we've generated an express app without too much trouble, but how can we mix this with AngularJS?.

How express works, briefly?

Express is a framework built on top of **Nodejs**, you can see the official documentation at the [Express Site](#). But for our purpose we need to know that **Express** is the responsible when we type, for example, <http://localhost:3000/home> of rendering the home page of our application. From the recently created app created we can check:

```
FILE: routes/index.js  
var express = require('express');  
var router = express.Router();  
  
/* GET home page. */  
router.get('/', function(req, res, next) {  
  res.render('index', { title: 'Express' });  
});  
  
module.exports = router;
```

这段代码告诉我们，当用户访问 `http://localhost:3000` 时，必须渲染 `index` 视图，并传递一个带有 `title` 属性和值为 Express 的 JSON。但当我们检查 `views` 目录并打开 `index.jade` 时，我们可以看到：

```
extends layout
block content
h1= title
p 欢迎来到 #{title}
```

这是 Express 的另一个强大功能，模板引擎，它们允许你通过传递变量来渲染页面内容，或者继承另一个模板，使你的页面更简洁且易于他人理解。文件扩展名是 `.jade`，据我所知，Jade 改名为 Pug，基本上是同一个模板引擎，但带有一些更新和核心修改。

安装 Pug 并更新 Express 模板引擎。

好的，要开始使用 Pug 作为我们项目的模板引擎，我们需要运行：

```
npm install --save pug
```

这将安装 Pug 作为我们项目的依赖并保存到 `package.json`。要使用它，我们需要修改文件 `app.js`：

```
var app = express();
// 视图引擎设置
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

并将视图引擎的那一行替换为 `pug`，就完成了。我们可以再次运行项目，使用 `npm start`，你会看到一切运行正常。

AngularJS 在这一切中扮演什么角色？

AngularJS 是一个 Javascript MVW（模型-视图-任意）框架，主要用于创建 SPA（单页应用程序），安装非常简单，你可以访问 AngularJS 网站 下载最新版本，当前是 v1.6.4。

下载 AngularJS 后，我们应将文件复制到项目中的 `public/javascripts` 文件夹，简单说明一下，这个文件夹用于存放网站的静态资源，如图片、CSS、JavaScript 文件等。当然，这可以通过 `app.js` 文件进行配置，但我们保持简单。现在我们在 `javascripts` 公共文件夹中创建一个名为 `ng-app.js` 的文件，这是我们的应用程序所在的文件，就放在 AngularJS 所在的位置。要启用 AngularJS，我们需要按如下方式修改 `views/layout.pug` 的内容：

```
doctype html
html(ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    body(ng-controller='indexController')
      block content

      script(type='text-javascript', src='javascripts/angular.min.js')
      script(type='text-javascript', src='javascripts/ng-app.js')
```

我们在这里做什么？嗯，我们正在引入 AngularJS 核心和我们最近创建的文件 `ng-app.js`，这样当

What this code is telling us is that when the user goes to `http://localhost:3000` it must render the `index` view and pass a **JSON** with a `title` property and value Express. But when we check the `views` directory and open `index.jade` we can see this:

```
extends layout
block content
h1= title
p Welcome to #{title}
```

This is another powerful Express feature, **template engines**, they allow you to render content in the page by passing variables to it or inherit another template so your pages are more compact and better understandable by others. The file extension is `.jade` as far as I know **Jade** changed the name for **Pug**, basically is the same template engine but with some updates and core modifications.

Installing Pug and updating Express template engine.

Ok, to start using Pug as the template engine of our project we need to run:

```
npm install --save pug
```

This will install Pug as a dependency of our project and save it to `package.json`. To use it we need to modify the file `app.js`:

```
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

And replace the line of view engine with `pug` and that's all. We can run again our project with `npm start` and we'll see that everything is working fine.

How AngularJS fits in all of this?

AngularJS is an Javascript **MVV**(Model-View-Whatever) Framework mainly used to create **SPA**(Simple Page Application) installing is fairly simple, you can go to [AngularJS website](#) and download the latest version which is **v1.6.4**.

After we downloaded AngularJS when should copy the file to our **public/javascripts** folder inside our project, a little explanation, this is the folder that serves the static assets of our site, images, css, javascript files and so on. Of course this is configurable through the `app.js` file, but we'll keep it simple. Now we create a file named **ng-app.js**, the file where our application will live, inside our `javascripts` public folder, just where AngularJS lives. To bring AngularJS up we need to modify the content of `views/layout.pug` as follow:

```
doctype html
html(ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    body(ng-controller='indexController')
      block content

      script(type='text-javascript', src='javascripts/angular.min.js')
      script(type='text-javascript', src='javascripts/ng-app.js')
```

What are we doing here?, well, we're including AngularJS core and our recently created file `ng-app.js` so when the

模板被渲染时，它会启动AngularJS，注意ng-app指令的使用，这告诉AngularJS这是我们的应用名称，它应该遵循这个名称。

所以，我们的ng-app.js的内容将是：

```
angular.module('first-app', [])
.controller('indexController', ['$scope', indexController]);

function indexController($scope) {
  $scope.name = 'sigfried';
}
```

我们这里使用的是最基本的 AngularJS 功能，双向数据绑定，这使我们能够即时刷新视图和控制器的内容，这只是一个非常简单的解释，但你可以在 Google 或 StackOverflow 上查找，了解它的具体工作原理。

所以，我们已经有了 AngularJS 应用的基本模块，但还有一件事要做，我们需要更新我们的 index.pug 页面以查看 Angular 应用的更改，开始吧：

```
extends layout
block content
div(ng-controller='indexController')
  h1= title
  p 欢迎 {{name}}
  input(type='text' ng-model='name')
```

这里我们只是将输入框绑定到控制器中 AngularJS 作用域里定义的属性 name：

```
$scope.name = 'sigfried';
```

这样做的目的是每当我们更改输入框中的文本时，上面的段落内容会在 {{name}} 中更新，这称为插值，又是 AngularJS 用于在模板中渲染内容的一个功能。

所以，一切都设置好了，我们现在可以运行 `npm start`，访问 `http://localhost:3000`，看到我们的 Express 应用提供页面，AngularJS 管理应用前端。

template is rendered it will bring AngularJS up, notice the use of the **ng-app** directive, this is telling AngularJS that this is our application name and it should stick to it.

So, the content of our **ng-app.js** will be:

```
angular.module('first-app', [])
  .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
  $scope.name = 'sigfried';
}
```

We're using the most basic AngularJS feature here, **two-way data binding**, this allows us to refresh the content of our view and controller instantly, this is a very simple explanation, but you can make a research in Google or StackOverflow to see how it really works.

So, we have the basic blocks of our AngularJS application, but there is something we got to do, we need to update our index.pug page to see the changes of our angular app, let's do it:

```
extends layout
block content
div(ng-controller='indexController')
  h1= title
  p Welcome {{name}}
  input(type='text' ng-model='name')
```

Here we're just binding the input to our defined property name in the AngularJS scope inside our controller:

```
$scope.name = 'sigfried';
```

The purpose of this is that whenever we change the text in the input the paragraph above will update its content inside the {{name}}, this is called **interpolation**, again another AngularJS feature to render our content in the template.

So, all is setup, we can now run **npm start** go to `http://localhost:3000` and see our express application serving the page and AngularJS managing the application frontend.

第100章：NodeJs 路由

如何在 Node.js 下设置基本的 Express 网络服务器及探索 Express 路由器。

第100.1节：Express 网络服务器路由

创建 Express 网络服务器

Express 服务器非常方便，得到了许多用户和社区的深入使用，正在变得越来越流行。

让我们创建一个 Express 服务器。为了包管理和依赖的灵活性，我们将使用 NPM (Node 包管理器)。

1. 进入项目目录并创建 package.json 文件。package.json

```
{  
  "name": "expressRouter",  
  "version": "0.0.1",  
  "scripts": {  
    "start": "node Server.js"  
  },  
  "dependencies": {  
    "express": "^4.12.3"  
  }  
}
```

2. 保存文件并使用以下命令安装 express 依赖 `npm install`。这样将会创建在你的项目目录中包含 `node_modules` 以及所需的依赖项。

3. 让我们创建 Express Web 服务器。进入项目目录并创建 server.js 文件。server.js

```
var express = require("express");  
var app = express();  
  
// 创建 Router() 对象  
var router = express.Router();  
// 在这里提供所有路由，这是主页的路由。  
router.get("/", function(req, res){  
  res.json({ "message" : "Hello World" });  
});  
app.use("/api", router);  
// 监听此端口  
app.listen(3000, function(){  
  console.log("Live at Port 3000");  
});
```

4. 通过输入以下命令运行服务器。

```
node server.js
```

如果服务器成功运行，你将看到类似如下内容。

Chapter 100: NodeJs Routing

How to set up basic Express web server under the node js and Exploring the Express router.

Section 100.1: Express Web Server Routing

Creating Express Web Server

Express server came handy and it deeps through many user and community. It is getting popular.

Lets create a Express Server. For Package Management and Flexibility for Dependency We will use NPM(Node Package Manager).

1. Go to the Project directory and create package.json file. **package.json**

```
{  
  "name": "expressRouter",  
  "version": "0.0.1",  
  "scripts": {  
    "start": "node Server.js"  
  },  
  "dependencies": {  
    "express": "^4.12.3"  
  }  
}
```

2. Save the file and install the express dependency using following command `npm install`. This will create `node_modules` in you project directory along with required dependency.

3. Let's create Express Web Server. Go to the Project directory and create server.js file. **server.js**

```
var express = require("express");  
var app = express();  
  
//Creating Router() object  
var router = express.Router();  
// Provide all routes here, this is for Home page.  
router.get("/", function(req, res){  
  res.json({ "message" : "Hello World" });  
});  
app.use("/api", router);  
// Listen to this Port  
app.listen(3000, function(){  
  console.log("Live at Port 3000");  
});
```

4. Run the server by typing following command.

```
node server.js
```

If Server runs successfully, you will see something like this.

```
pralad@pralad:~/reactjs/routing-express  
pralad@pralad:~/reactjs/routing-express$ node server.js  
Live at Port 3000
```

5. 现在打开浏览器或 Postman 并发送请求

<http://localhost:3000/api/>

输出将会是



```
localhost:3000/api/ ×  
← → C ⌂ ⓘ localhost:3000/api/  
{"message": "Hello World"}
```

这就是 Express 路由的基础。

现在让我们处理 GET、POST 等请求。

```
pralad@pralad:~/reactjs/routing-express  
pralad@pralad:~/reactjs/routing-express$ node server.js  
Live at Port 3000
```

5. Now go to the browser or postman and made a request

<http://localhost:3000/api/>

The output will be



```
localhost:3000/api/ ×  
← → C ⌂ ⓘ localhost:3000/api/  
{"message": "Hello World"}
```

That is all, the basic of Express routing.

Now let's handle the GET, POST etc.

修改你的 server.js 文件如下

```
var express = require("express");
var app = express();

// 创建 Router() 对象

var router = express.Router();

// 路由中间件, 在定义路由之前提到过。

router.use(function(req, res, next) {
  console.log("/" + req.method);
  next();
});

// 在这里提供所有路由, 这是主页用的。
router.get("/", function(req, res) {
  res.json({ "message" : "Hello World" });
});

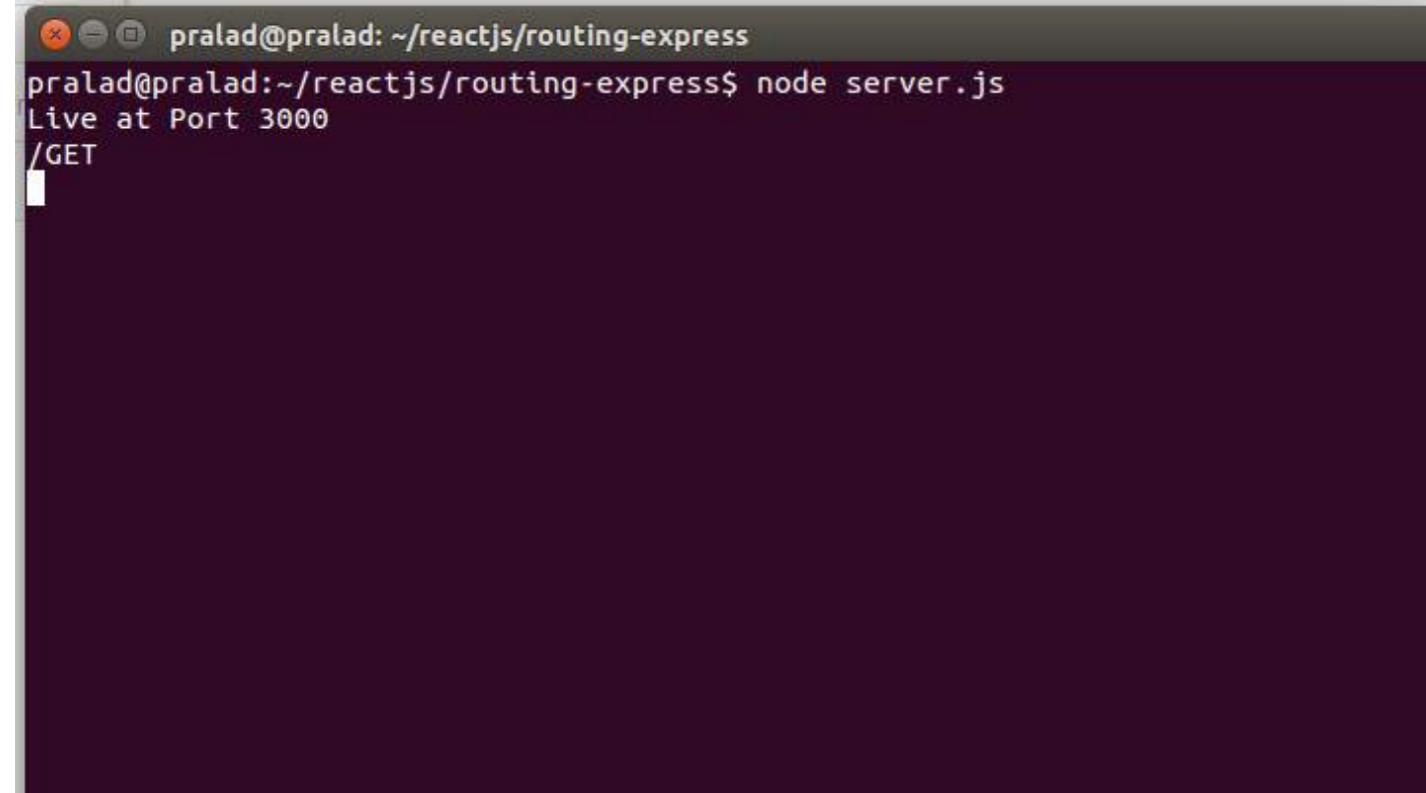
app.use("/api", router);

app.listen(3000, function(){
  console.log("Live at Port 3000");
});
```

现在如果你重启服务器并发出请求到

```
http://localhost:3000/api/
```

你将看到类似的内容



```
pralad@pralad:~/reactjs/routing-express
pralad@pralad:~/reactjs/routing-express$ node server.js
Live at Port 3000
/GET
```

路由中访问参数

Change your server.js file like

```
var express = require("express");
var app = express();

// Creating Router() object

var router = express.Router();

// Router middleware, mentioned it before defining routes.

router.use(function(req, res, next) {
  console.log("/" + req.method);
  next();
});

// Provide all routes here, this is for Home page.
router.get("/", function(req, res){
  res.json({ "message" : "Hello World" });
});

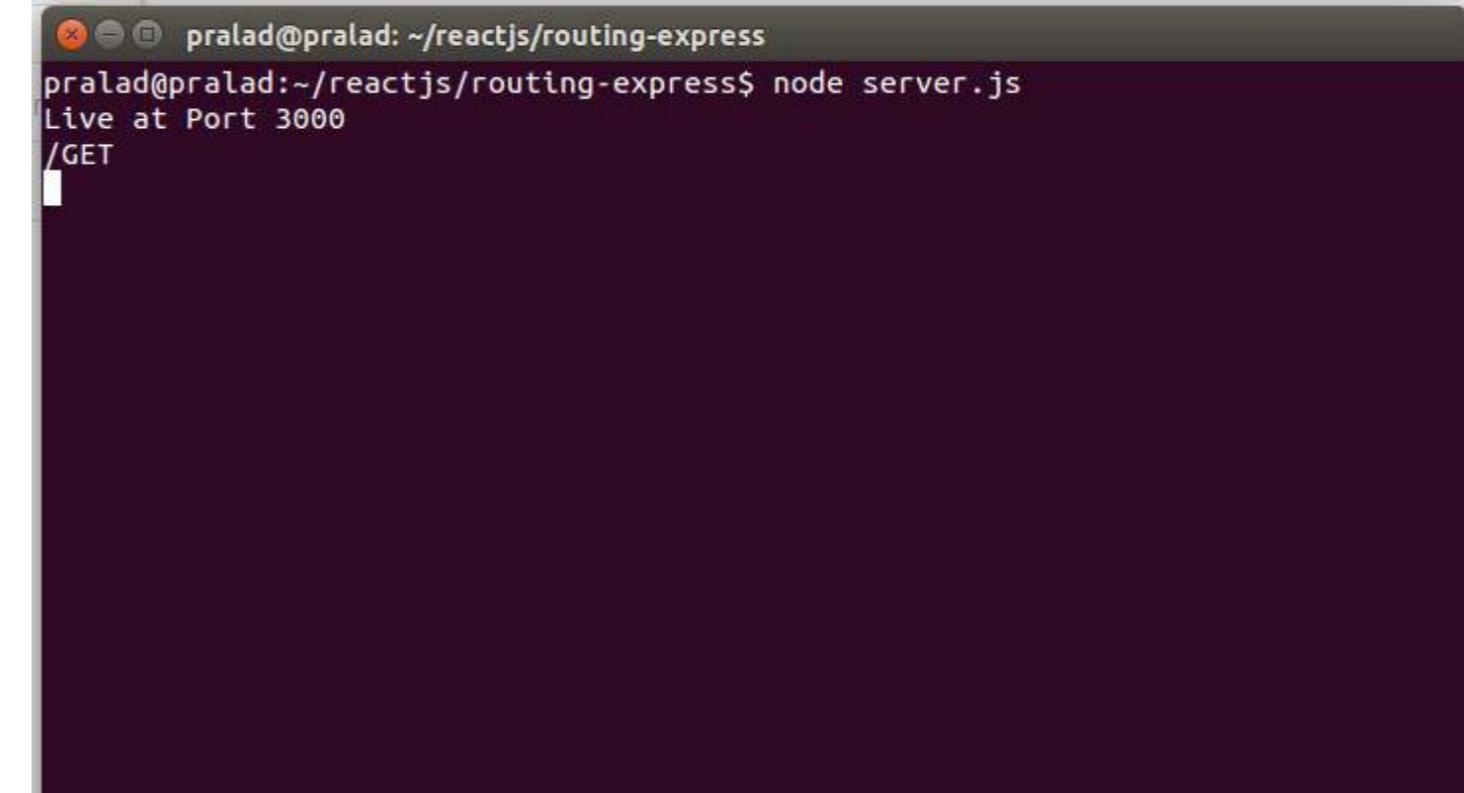
app.use("/api", router);

app.listen(3000, function(){
  console.log("Live at Port 3000");
});
```

Now if you restart the server and made the request to

```
http://localhost:3000/api/
```

You Will see something like



```
pralad@pralad:~/reactjs/routing-express
pralad@pralad:~/reactjs/routing-express$ node server.js
Live at Port 3000
/GET
```

Accessing Parameter in Routing

你也可以从 URL 中访问参数，比如 `http://example.com/api/:name/`。这样就可以访问 name 参数。

将以下代码添加到你的 `server.js` 中

```
router.get("/user/:id", function(req, res){  
  res.json({"message" : "Hello "+req.params.id});  
});
```

现在重启服务器并访问 `[http://localhost:3000/api/user/Adem][4]`，输出将如下所示

A screenshot of a web browser window. The address bar shows `localhost:3000/api/`. Below the address bar, there are navigation buttons (back, forward, refresh, home) and a link labeled `localhost:3000/api/user/Adem`. The main content area of the browser displays the JSON response: `{"message": "Hello Adem"}`.

You can access the parameter from url also, Like `http://example.com/api/:name/`. So name parameter can be access.

Add the following code into your `server.js`

```
router.get("/user/:id", function(req, res){  
  res.json({"message" : "Hello "+req.params.id});  
});
```

Now restart server and go to `[http://localhost:3000/api/user/Adem][4]`, the output will be like

A screenshot of a web browser window. The address bar shows `localhost:3000/api/`. Below the address bar, there are navigation buttons (back, forward, refresh, home) and a link labeled `localhost:3000/api/user/Adem`. The main content area of the browser displays the JSON response: `{"message": "Hello Adem"}`.

第101章：创建一个同时支持Promise和错误优先回调的Node.js库

许多人喜欢使用Promise和/或async/await语法，但在编写模块时，支持经典回调风格的方法对某些程序员来说也很有用。与其创建两个模块，或两套函数，或让程序员为你的模块添加Promise支持，不如让你的模块同时支持这两种编程方式，使用bluebird的asCallback()或Q的nodeify()即可。

第101.1节：使用Bluebird的示例模块及对应程序

math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // 仅回调方法的示例
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error("a 必须是数字"));
    if (typeof b !== 'number')
      return callback(new Error("b 必须是数字"));

    return callback(null, a + b);
  },

  // 仅使用 Promise 的方法示例
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error("a 必须是数字"));
      if (typeof b !== 'number')
        return reject(new Error("b 必须是数字"));
      resolve(a + b);
    });
  },

  // 一个既可以作为 Promise 使用也可以使用回调的方法
  sum: function(a, b, callback) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error("a 必须是数字"));
      if (typeof b !== 'number')
        return reject(new Error("b 必须是数字"));
      resolve(a + b);
    }).asCallback(callback);
  },
};
```

index.js

Chapter 101: Creating a Node.js Library that Supports Both Promises and Error-First Callbacks

Many people like working with promises and/or async/await syntax, but when writing a module it would be useful to some programmers to support classic callback style methods as well. Rather than creating two modules, or two sets of functions, or having the programmer promisify your module, your module can support both programming methods at one using bluebird's asCallback() or Q's nodeify().

Section 101.1: Example Module and Corresponding Program using Bluebird

math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // example of a callback-only method
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error("a must be a number"));
    if (typeof b !== 'number')
      return callback(new Error("b must be a number"));

    return callback(null, a + b);
  },

  // example of a promise-only method
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error("a must be a number"));
      if (typeof b !== 'number')
        return reject(new Error("b must be a number"));
      resolve(a + b);
    });
  },

  // a method that can be used as a promise or with callbacks
  sum: function(a, b, callback) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error("a must be a number"));
      if (typeof b !== 'number')
        return reject(new Error("b must be a number"));
      resolve(a + b);
    }).asCallback(callback);
  },
};
```

index.js

```
'use strict';

const math = require('./math');
```

// 经典回调

```
math.callbackSum(1, 3, function(err, result) {
  if (err)
    console.log('测试 1: ' + err);
  else
    console.log('测试 1: 答案是 ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
  if (err)
    console.log('测试 2: ' + err);
  else
    console.log('测试 2: 答案是 ' + result);
});
```

// Promise

```
math.promiseSum(2, 5)
  .then(function(result) {
    console.log('测试3: 答案是 ' + result);
  })
  .catch(function(err) {
    console.log('测试 3: ' + err);
  });

math.promiseSum(1)
  .then(function(result) {
    console.log('测试 4: 答案是 ' + result);
  })
  .catch(function(err) {
    console.log('测试 4: ' + err);
  });
```

// promise/callback 方法像 promise 一样使用

```
math.sum(8, 2)
  .then(function(result) {
    console.log('测试 5: 答案是 ' + result);
  })
  .catch(function(err) {
    console.log('测试 5: ' + err);
  });

// promise/callback 方法与回调函数一起使用
```

```
math.sum(7, 11, function(err, result) {
  if (err)
    console.log('测试 6: ' + err);
  else
    console.log('测试 6: 答案是 ' + result);
});
```

```
'use strict';
```

```
const math = require('./math');
```

// classic callbacks

```
math.callbackSum(1, 3, function(err, result) {
  if (err)
    console.log('Test 1: ' + err);
  else
    console.log('Test 1: the answer is ' + result);
});
```

```
math.callbackSum(1, 'd', function(err, result) {
  if (err)
    console.log('Test 2: ' + err);
  else
    console.log('Test 2: the answer is ' + result);
});
```

// promises

```
math.promiseSum(2, 5)
  .then(function(result) {
    console.log('Test 3: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 3: ' + err);
  });

math.promiseSum(1)
  .then(function(result) {
    console.log('Test 4: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 4: ' + err);
  });
```

// promise/callback method used like a promise

```
math.sum(8, 2)
  .then(function(result) {
    console.log('Test 5: the answer is ' + result);
  })
  .catch(function(err) {
    console.log('Test 5: ' + err);
  });

// promise/callback method used with callbacks
```

```
math.sum(7, 11, function(err, result) {
  if (err)
    console.log('Test 6: ' + err);
  else
    console.log('Test 6: the answer is ' + result);
});
```

// 使用 promise/callback 方法，结合 async/await 语法像使用 promise 一样

```
(async () => {  
  try {  
    let x = await math.sum(6, 3);  
    console.log('测试 7a: ' + x);  
  
    let y = await math.sum(4, 's');  
    console.log('测试 7b: ' + y);  
  
  } catch(err) {  
    console.log(err.message);  
  }  
})();
```

// promise/callback method used like a promise with async/await syntax

```
(async () => {  
  try {  
    let x = await math.sum(6, 3);  
    console.log('Test 7a: ' + x);  
  
    let y = await math.sum(4, 's');  
    console.log('Test 7b: ' + y);  
  
  } catch(err) {  
    console.log(err.message);  
  }  
})();
```

belindoc.com

第102章：项目结构

Nodejs 项目的结构受个人偏好、项目架构和所使用的模块注入策略影响。同时也受基于事件的架构影响，该架构使用动态模块实例化机制。要实现 MVC 结构，必须将服务器端和客户端的源代码分开，因为客户端代码通常会被压缩后发送到浏览器，且本质上是公开的。服务器端或后端将提供用于执行 CRUD 操作的 API。

第 102.1 节：一个简单的带有 MVC 和 API 的 nodejs 应用

- 第一个主要区别是在用于托管的动态生成目录和源代码目录之间。
- 源代码目录将包含一个配置文件或文件夹，具体取决于你可能拥有的配置量。这包括环境配置和业务逻辑配置，你可以选择将其放在 config 目录中。

```
|-- Config  
  |-- config.json  
  |-- appConfig  
    |-- pets.config  
    |-- payment.config
```

- 现在最重要的目录是区分服务器端/后端和前端模块的目录。两个目录server和webapp分别代表后端和前端，我们可以选择将它们放在名为 src 的源代码目录中。

你可以根据个人喜好为服务器端或 webapp 选择不同的名称，取决于什么对你来说更合理。确保不要让名称过长或过于复杂，因为它最终是内部项目结构的一部分。

- 在 server 目录中，你可以放置控制器、App.js/index.js（这是你的主要 nodejs 文件和入口点）。服务器目录还可以包含 dto 目录，里面存放所有由 API 控制器使用的数据传输对象。

```
|-- 服务器  
  |-- dto  
    |-- pet.js  
    |-- payment.js  
  |-- 控制器  
    |-- PetsController.js  
    |-- PaymentController.js  
  |-- App.js
```

- webapp 目录可以分为两个主要部分 public 和 mvc，这同样取决于你想使用的构建策略。我们使用 browserify 来构建 webapp 的 MVC 部分，并简单地压缩 mvc 目录中的内容。

```
-- webapp -- public -- mvc
```

- 现在 public 目录可以包含所有静态资源、图片、css（你也可以有 saas 文件）和

Chapter 102: Project Structure

The structure of nodejs project is influenced by the personal preferences, project's architecture and module injection strategy being used. Also on event based arc' which uses dynamic module instantiation mechanism. To have a MVC structure it is imperative to separate out the server side and client side source code as the client side code will probably be minimized and sent to browser and is public in its basic nature. And the server side or backend will provide API to perform CRUD operations

Section 102.1: A simple nodejs application with MVC and API

- The first major distinction is between the dynamically generated directories which will be used for hosting and source directories.
- The source directories will have a config file or folder depending on the amount of configuration you may have . This includes the environment configuration and business logic configuration which you may choose to put inside config directory.

```
|-- Config  
  |-- config.json  
  |-- appConfig  
    |-- pets.config  
    |-- payment.config
```

- Now the most vital directories where we distinguish between the server side/backend and the frontend modules . The 2 directories server and webapp represent the backend and frontend respectively which we can choose to put inside a source directory viz. src.

You can go with different names as per personal choice for server or webapp depending on what makes sense for you. Make sure you don't want to make it too long or to complex as it is in the end internal project structure.

- Inside the server directory you can have the controller ,the App.js/index.js which will be your main nodejs file and start point .The server dir. can also have the dto dir which holds all the data transfer objects which will be used by API controllers.

```
|-- server  
  |-- dto  
    |-- pet.js  
    |-- payment.js  
  |-- controller  
    |-- PetsController.js  
    |-- PaymentController.js  
  |-- App.js
```

- The webapp directory can be divided into two major parts public and mvc , this is again influenced by what build strategy you want to use. We are using browserify to build the MVC part of webapp and minimize the contents from mvc directory simply put.

```
-- webapp -- public -- mvc
```

- Now the public directory can contain all the static resources,images,css(you can have saas files as well) and

最重要的是HTML文件。

```
|-- public
  |-- build // 将包含压缩后的脚本 (mvc)
  |-- images
    |-- mouse.jpg
    |-- cat.jpg
  |-- styles
    |-- style.css
  |-- views
    |-- petStore.html
    |-- paymentGateway.html
    |-- header.html
    |-- footer.html
  |-- index.html
```

- mvc目录将包含前端逻辑，包括models、view controllers以及作为UI一部分可能需要的其他utils模块。此外，index.js或shell.js（任选其一）也属于该目录的一部分。

most importantly the HTML files .

```
|-- public
  |-- build // will contain minified scripts(mvc)
  |-- images
    |-- mouse.jpg
    |-- cat.jpg
  |-- styles
    |-- style.css
  |-- views
    |-- petStore.html
    |-- paymentGateway.html
    |-- header.html
    |-- footer.html
  |-- index.html
```

- The mvc directory will contain the front-end logic including the models, the view controllers and any other utils modules you may need as part of UI. Also the index.js or shell.js whichever may suite you is part of this directory as well.

```
|-- mvc
  |-- controllers
    |-- Dashboard.js
    |-- Help.js
    |-- Login.js
  |-- utils
  |-- index.js
```

所以总的来说，整个项目结构如下所示。一个简单的构建任务如gulp browserify将会压缩mvc脚本并发布到public目录。然后我们可以通过 express.use(static('public'))接口将该public目录作为静态资源提供。

```
|-- node_modules
|-- src
|-- 服务器
  |-- 控制器
  |-- App.js // node 应用
|-- webapp
  |-- public
    |-- styles
    |-- images
    |-- index.html
  |-- mvc
    |-- 控制器
    |-- shell.js // mvc shell
|-- 配置
|-- 说明文档.md
|-- .gitignore
|-- package.json
```

```
|-- mvc
  |-- controllers
    |-- Dashboard.js
    |-- Help.js
    |-- Login.js
  |-- utils
  |-- index.js
```

So in conclusion the entire project structure will look like below. And a simple build task like gulp browserify will minify the mvc scripts and publish in public directory. We can then provide this public directory as static resource via **express.use(static('public'))** api.

```
|-- node_modules
|-- src
|-- server
  |-- controller
  |-- App.js // node app
|-- webapp
  |-- public
    |-- styles
    |-- images
    |-- index.html
  |-- mvc
    |-- controller
    |-- shell.js // mvc shell
|-- config
|-- Readme.md
|-- .gitignore
|-- package.json
```

第103章：避免回调地狱

第103.1节：Async模块

源代码可从GitHub下载。或者，你也可以使用npm安装：

```
$ npm install --save async
```

也可以使用Bower：

```
$ bower install async
```

示例：

```
var async = require("async");
async.parallel([
  function(callback) { ... },
  function(callback) { ... }
], function(err, results) {
  // 可选回调
});
});
```

第103.2节：Async模块

幸运的是，像Async.js这样的库存在，试图解决这个问题。Async在你的代码之上添加了一层薄薄的函数封装，但通过避免回调嵌套，可以大大降低复杂性。

Async中存在许多辅助方法，可用于不同的场景，如串行、并行、瀑布流等。每个函数都有特定的使用场景，因此花些时间学习哪个函数适合哪种情况。

尽管 Async 很好，但像任何东西一样，它并不完美。很容易因为组合使用 series、parallel、forever 等而失控，这时你的代码又回到了开始时那种混乱状态。要小心不要过早优化。仅仅因为几个异步任务可以并行运行，并不意味着它们就应该并行。实际上，由于 Node 只是单线程的，使用 Async 并行运行任务几乎没有性能提升。

源码可从 <https://github.com/caolan/async> 下载。或者，你也可以使用

npm 安装：

```
$ npm install --save async
```

也可以使用Bower：

```
$ bower install async
```

Async 的 waterfall 示例：

```
var fs = require('fs');
var async = require('async');

var myFile = '/tmp/test';

async.waterfall([
  function(callback) {
    fs.readFile(myFile, 'utf8', callback);
  },
  function(txt, callback) {
```

Chapter 103: Avoid callback hell

Section 103.1: Async module

The source is available for download from GitHub. Alternatively, you can install using npm:

```
$ npm install --save async
```

As well as using Bower:

```
$ bower install async
```

Example:

```
var async = require("async");
async.parallel([
  function(callback) { ... },
  function(callback) { ... }
], function(err, results) {
  // optional callback
});
```

Section 103.2: Async Module

Thankfully, libraries like Async.js exist to try and curb the problem. Async adds a thin layer of functions on top of your code, but can greatly reduce the complexity by avoiding callback nesting.

Many helper methods exist in Async that can be used in different situations, like series, parallel, waterfall, etc. Each function has a specific use-case, so take some time to learn which one will help in which situations.

As good as Async is, like anything, it's not perfect. It's very easy to get carried away by combining series, parallel, forever, etc, at which point you're right back to where you started with messy code. Be careful not to prematurely optimize. Just because a few async tasks can be run in parallel doesn't always mean they should. In reality, since Node is only single-threaded, running tasks in parallel on using Async has little to no performance gain.

The source is available for download from <https://github.com/caolan/async>. Alternatively, you can install using npm:

```
$ npm install --save async
```

As well as using Bower:

```
$ bower install async
```

Async's waterfall Example:

```
var fs = require('fs');
var async = require('async');

var myFile = '/tmp/test';

async.waterfall([
  function(callback) {
    fs.readFile(myFile, 'utf8', callback);
  },
  function(txt, callback) {
```

```
txt = txt + 'Appended something!';
fs.writeFile
(myFile, txt, callback);

], function (err, result) {
  if(err) return console.log(err);
  console.log('Appended text!');
});
```

```
txt = txt + '\nAppended something!';
fs.writeFile(myFile, txt, callback);
}

], function (err, result) {
  if(err) return console.log(err);
  console.log('Appended text!');
});
```

belindoc.com

第104章：Arduino与nodeJs的通信

展示Node.js如何与Arduino Uno通信的方法。

第104.1节：Node Js通过serialport与Arduino通信

Node js代码

本主题的示例是Node.js服务器通过serialport与Arduino通信。

```
npm install express --save
npm install serialport --save
```

示例app.js：

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
  baudrate: 9600
});

arduinoSerialPort.on('open', function() {
  console.log('串口 ' + arduinoCOMPort + ' 已打开。');
});

app.get('/', function (req, res) {
  return res.send('运行中');
})

app.get('/:action', function (req, res) {
  var action = req.params.action || req.param('action');

  if(action == 'led'){
    arduinoSerialPort.write("w");
    return res.send('LED 灯已打开！');
  }
  if(action == 'off'){
    arduinoSerialPort.write("t");
    return res.send("LED 灯已关闭！");
  }

  return res.send('动作: ' + action);
});

app.listen(port, function () {
  console.log('示例应用正在监听端口 http://0.0.0.0:' + port + '!');
});
```

Chapter 104: Arduino communication with nodeJs

Way to show how Node.js can communicate with Arduino Uno.

Section 104.1: Node Js communication with Arduino via serialport

Node js code

Sample to start this topic is Node.js server communicating with Arduino via serialport.

```
npm install express --save
npm install serialport --save
```

Sample app.js:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
  baudrate: 9600
});

arduinoSerialPort.on('open', function() {
  console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
});

app.get('/', function (req, res) {
  return res.send('Working');
})

app.get('/:action', function (req, res) {
  var action = req.params.action || req.param('action');

  if(action == 'led'){
    arduinoSerialPort.write("w");
    return res.send('Led light is on!');
  }
  if(action == 'off') {
    arduinoSerialPort.write("t");
    return res.send("Led light is off!");
  }

  return res.send('Action: ' + action);
});

app.listen(port, function () {
  console.log('Example app listening on port http://0.0.0.0:' + port + '!');
});
```

```
});
```

启动示例 express 服务器：

```
node app.js
```

Arduino 代码

```
// setup 函数在按下重置或给板子上电时运行一次
void setup() {
    // 将数字引脚 LED_BUILTIN 初始化为输出模式。
    Serial.begin(9600); // Begen 在端口 9600 上监听串口
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
}

// 循环函数会不断重复执行
void loop() {

    if(Serial.available() > 0) // 从串口读取数据
    {
        char ReaderFromNode; // 存储当前字符
        ReaderFromNode = (char) Serial.read();
        convertToState(ReaderFromNode); // 将字符转换为状态
    }
    delay(1000);
}

void convertToState(char chr) {
    if(chr=='o'){
        digitalWrite(LED_BUILTIN, HIGH);
        delay(100);
    }
    if(chr=='f'){
        digitalWrite(LED_BUILTIN, LOW);
        delay(100);
    }
}
```

启动

1. 将Arduino连接到你的设备。
2. 启动服务器

通过Node.js Express服务器控制内置LED。

打开LED：

```
http://0.0.0.0:3000/led
```

关闭LED：

```
http://0.0.0.0:3000/off
```

```
});
```

Starting sample express server:

```
node app.js
```

Arduino code

```
// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin LED_BUILTIN as an output.

    Serial.begin(9600); // Begen listening on port 9600 for serial
    pinMode(LED_BUILTIN, OUTPUT);

    digitalWrite(LED_BUILTIN, LOW);
}

// the loop function runs over and over again forever
void loop() {

    if(Serial.available() > 0) // Read from serial port
    {
        char ReaderFromNode; // Store current character
        ReaderFromNode = (char) Serial.read();
        convertToState(ReaderFromNode); // Convert character to state
    }
    delay(1000);
}

void convertToState(char chr) {
    if(chr=='o'){
        digitalWrite(LED_BUILTIN, HIGH);
        delay(100);
    }
    if(chr=='f'){
        digitalWrite(LED_BUILTIN, LOW);
        delay(100);
    }
}
```

Starting Up

1. Connect the arduino to your machine.
2. Start the server

Control the build in led via node js express server.

To turn on the led:

```
http://0.0.0.0:3000/led
```

To turn off the led:

```
http://0.0.0.0:3000/off
```

第105章：N-API

N-API是为NodeJS创建本地模块的一种新的更好的方式。N-API处于早期阶段，因此文档可能存在不一致的情况。

第105.1节：N-API入门

该模块在 hello 模块上注册了 hello 函数。hello 函数使用printf在控制台打印 Hello world 并从本地函数返回1373给 JavaScript 调用者。

```
#include <node_api.h>
#include <stdio.h>

napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
        * 用于属性键的字符串描述，采用 UTF8 编码。
        */
        .utf8name = "hello",
        /*
        * 设置此项以使属性描述符对象的 value 属性
        * 成为由 method 表示的 JavaScript 函数。
        */
        /*
        * 如果传入此项，则将 value、getter 和 setter 设置为 NULL (因为这些成员不会被使用)。
        */
        /*
        */
        .method = say_hello,
        /*
        * 当对属性执行获取访问时调用的函数。
        */
        /*
        * 如果传入此项，则将 value 和 method 设置为 NULL (因为这些成员不会被使用)。
        */
        /*
        * 当从 JavaScript 代码访问属性时，运行时会隐式调用给定的函数
        * (或者如果使用 N-API 调用对属性执行获取操作)。
        */
        /*
        */
        .getter = NULL,
        /*
        * 当对属性执行设置访问时调用的函数。
        */
        /*
        * 如果传入此项，则将 value 和 method 设置为 NULL (因为这些成员不会被使用)。
        */
        /*
        * 当从 JavaScript 代码设置属性时，运行时会隐式调用给定的函数
        * (或者如果使用 N-API 调用对属性执行设置操作)。
        */
        /*
        */
        .setter = NULL,
        /*
        * 如果属性是数据属性，通过 get 访问该属性时检索到的值。
        */
        /*
        * 如果传入此项，则将 getter、setter、method 和 data 设置为 NULL (因为这些成员不会被使用)。
        */
    };
}
```

Chapter 105: N-API

The N-API is a new and better way for creating native module for NodeJS. N-API is in early stage so it may have inconsistent documentation.

Section 105.1: Hello to N-API

This module register hello function on hello module. hello function prints Hello world on console with printf and return 1373 from native function into javascript caller.

```
#include <node_api.h>
#include <stdio.h>

napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world\n");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
        * String describing the key for the property, encoded as UTF8.
        */
        .utf8name = "hello",
        /*
        * Set this to make the property descriptor object's value property
        * to be a JavaScript function represented by method.
        */
        /*
        * If this is passed in, set value, getter and setter to NULL (since these members won't be
        * used).
        */
        .method = say_hello,
        /*
        * A function to call when a get access of the property is performed.
        */
        /*
        * If this is passed in, set value and method to NULL (since these members won't be used).
        */
        /*
        * The given function is called implicitly by the runtime when the property is set
        * from JavaScript code (or if a set on the property is performed using a N-API call).
        */
        .getter = NULL,
        /*
        * A function to call when a set access of the property is performed.
        */
        /*
        * If this is passed in, set value and method to NULL (since these members won't be used).
        */
        /*
        * The given function is called implicitly by the runtime when the property is set
        * from JavaScript code (or if a set on the property is performed using a N-API call).
        */
        .setter = NULL,
        /*
        * The value that's retrieved by a get access of the property if the property is a data
        * property.
        */
        /*
        * If this is passed in, set getter, setter, method and data to NULL (since these members
        * won't be used).
        */
    };
}
```

```
 */
.value = NULL,
/*
* 与特定属性相关联的属性。参见 napi_property_attributes。
*/
.attributes = napi_default,
/*
* 如果调用此函数，传递给 method、getter 和 setter 的回调数据。
*/
.data = NULL
};

/*
* 此方法允许在给定对象上高效定义多个属性。
*/
status = napi_define_properties(env, exports, 1, &desc);

if (status != napi_ok)
    return;
}
```

NAPI_MODULE(hello, init)

belindoc.com

```
 */
.value = NULL,
/*
* The attributes associated with the particular property. See napi_property_attributes。
*/
.attributes = napi_default,
/*
* The callback data passed into method, getter and setter if this function is invoked。
*/
.data = NULL
};

/*
* This method allows the efficient definition of multiple properties on a given object。
*/
status = napi_define_properties(env, exports, 1, &desc);

if (status != napi_ok)
    return;
}
```

NAPI_MODULE(hello, init)

第106章：多线程

Node.js 被设计为单线程。因此，实际上，使用 Node 启动的应用程序将在单个线程上运行。

然而，Node.js 本身是多线程运行的。I/O 操作等将从线程池中运行。此外，任何 Node 应用程序的实例都会在不同的线程上运行，因此要运行多线程应用程序，需要启动多个实例。

第106.1节：集群

cluster 模块允许多次启动相同的应用程序。

当不同实例具有相同的执行流程且彼此不依赖时，集群是理想的。在这种情况下，有一个主进程可以启动分叉进程（forks）和子进程（children）。子进程独立工作，拥有自己的内存空间和事件循环。

设置集群对网站/API 有益。任何线程都可以服务任何客户，因为它不依赖于其他线程。数据库（如 Redis）用于共享 Cookie，因为变量不能在线程之间共享！

```
// 在每个实例中运行
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('我总是被调用');

if (cluster.isMaster) {
    // 只运行一次 (在主进程中);
    console.log('我是主进程，正在启动工作进程！');
    for(var i = 0; i < numCPUs; i++) cluster.fork();

} else {
    // 在每个分叉中运行
    console.log('我是一个分叉！');
}

// 这里可以启动一个示例，比如一个网络服务器

}

console.log('我也总是被调用');
```

第106.2节：子进程

当需要独立运行具有不同初始化和关注点的进程时，子进程是首选方式。像集群中的分叉一样，child_process在其线程中运行，但与分叉不同的是，它有与父进程通信的方式。

通信是双向的，因此父进程和子进程都可以监听消息并发送消息。

Parent (./parent.js)

```
var child_process = require('child_process');
console.log('[父进程]', '初始化');

var child1 = child_process.fork(__dirname + '/child');
```

Chapter 106: Multithreading

Node.js has been designed to be single threaded. So for all practical purposes, applications that launch with Node will run on a single thread.

However, Node.js itself runs multi-threaded. I/O operations and the like will run from a thread pool. Further will any instance of a node application run on a different thread, therefore to run multi-threaded applications one launches multiple instances.

Section 106.1: Cluster

The cluster module allows one to start the same application multiple times.

Clustering is desirable when the different instances have the same flow of execution and don't depend on one another. In this scenario, you have one master that can start forks and the forks (or children). The children work independently and have their own space of Ram and Event Loop.

Setting up clusters can be beneficial for websites / APIs. Any thread can serve any customer, as it doesn't depend on other threads. A Database (like Redis) would be used to share Cookies, as **variables can't be shared!** between the threads.

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('I am always called');

if (cluster.isMaster) {
    // runs only once (within the master);
    console.log('I am the master, launching workers!');
    for(var i = 0; i < numCPUs; i++) cluster.fork();

} else {
    // runs in each fork
    console.log('I am a fork');

    // here one could start, as an example, a web server
}

console.log('I am always called as well');
```

Section 106.2: Child Process

Child Processes are the way to go when one wants to run processes independently with different initialization and concerns. Like forks in clusters, a child_process runs in its thread, but unlike forks, it has a way to communicate with its parent.

The communication goes both ways, so parent and child can listen for messages and send messages.

Parent (./parent.js)

```
var child_process = require('child_process');
console.log('[Parent]', 'initialize');

var child1 = child_process.fork(__dirname + '/child');
```

```

child1.on('消息', function(msg) {
  console.log('[父进程]', '来自子进程的回答:', msg);
});

// 可以发送任意多条消息
child1.send('你好'); // 你好啊 :)
child1.send('你好'); // 你好啊 :)

// 也可以有多个子进程
var child2 = child_process.fork(__dirname + '/child');

```

子进程 (./child.js)

```

// 这里会初始化孩子进程
// 这段代码只会执行一次
console.log('[子进程]', '初始化');

// 这里监听来自父进程的新任务
process.on('message', function(messageFromParent) {

  // 在这里执行一些密集的工作
  console.log('[子进程]', '子进程正在执行一些密集的工作');

  if(messageFromParent == 'Hello') process.send('你好 :)');
  else process.send('什么?');

})

```

除了 message，还可以监听许多事件，如 'error'、'connected' 或 'disconnect'。

启动子进程会产生一定的开销，因此希望尽可能少地创建子进程。

```

child1.on('message', function(msg) {
  console.log('[Parent]', 'Answer from child: ', msg);
});

// one can send as many messages as one want
child1.send('Hello'); // Hello to you too :)
child1.send('Hello'); // Hello to you too :)

// one can also have multiple children
var child2 = child_process.fork(__dirname + '/child');

```

Child (./child.js)

```

// here would one initialize this child
// this will be executed only once
console.log('[Child]', 'initialize');

// here one listens for new tasks from the parent
process.on('message', function(messageFromParent) {

  //do some intense work here
  console.log('[Child]', 'Child doing some intense work');

  if(messageFromParent == 'Hello') process.send('Hello to you too :)');
  else process.send('what?');

})

```

Next to message one can listen to [many events](#) like 'error', 'connected' or 'disconnect'.

Starting a child process has a certain cost associated with it. One would want to spawn as few of them as possible.

第107章：Node.js 下的 Windows 认证

第107.1节：使用 activedirectory

下面的示例取自完整文档，可在 [here \(GitHub\)](#) 或 [here \(NPM\)](#) 获取。

安装

```
npm install --save activedirectory
```

用法

```
// 初始化
var ActiveDirectory = require('activedirectory');
var config = {
  url: 'ldap://dc.domain.com',
  baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';
// 认证
ad.authenticate(username, password, function(err, auth) {
  if (err) {
    console.log('错误: ' + JSON.stringify(err));
    return;
  }
  if (auth) {
    console.log('认证成功!');
  } else {
    console.log('认证失败!');
  }
});
```

Chapter 107: Windows authentication under node.js

Section 107.1: Using activedirectory

The example below is taken from the full docs, available [here \(GitHub\)](#) or [here \(NPM\)](#).

Installation

```
npm install --save activedirectory
```

Usage

```
// Initialize
var ActiveDirectory = require('activedirectory');
var config = {
  url: 'ldap://dc.domain.com',
  baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';
// Authenticate
ad.authenticate(username, password, function(err, auth) {
  if (err) {
    console.log('ERROR: ' + JSON.stringify(err));
    return;
  }
  if (auth) {
    console.log('Authenticated!');
  } else {
    console.log('Authentication failed!');
  }
});
```

第108章：Require()

本说明文档重点解释了NodeJS语言中包含的require()语句的用途和用法。

Require是用于导入NodeJS模块中某些文件或包的语句。它用于改善代码结构和使用。`require()`用于本地安装的文件，路径是从调用`require`的文件直接指向目标文件的。

第108.1节：使用函数和文件开始使用require()

Require是Node解释为某种意义上的getter函数的语句。例如，假设你有一个名为`analysis.js`的文件，文件内容如下，

```
function analyzeWeather(weather_data) {  
    console.log('天气信息：' + weather_data.time + ':');  
    console.log('降雨量：' + weather_data.precip);  
    console.log('温度：' + weather_data.temp);  
    //更多weather_data的分析/打印...  
}
```

该文件仅包含方法`analyzeWeather(weather_data)`。如果我们想使用此函数，必须在该文件内部使用，或者复制到想要使用该函数的文件中。然而，Node提供了一个非常有用的工具来帮助代码和文件的组织，即modules。

为了使用我们的函数，我们必须首先通过在开头的语句中导出该函数。我们的新文件看起来是这样的，

```
module.exports = {  
    analyzeWeather: analyzeWeather  
}  
  
function analyzeWeather(weather_data) {  
    console.log('天气信息：' + weather_data.time + ':');  
    console.log('降雨量：' + weather_data.precip);  
    console.log('温度：' + weather_data.temp);  
    //更多weather_data的分析/打印...  
}
```

通过这个小的`module.exports`语句，我们的函数现在可以在文件外部使用。剩下要做的就是使用`require()`。

当使用`require`引入函数或文件时，语法非常相似。通常在文件开头进行，并赋值给`var`或`const`以便在整个文件中使用。例如，我们有另一个文件（与`analyze.js`同级）名为`handleWeather.js`，内容如下，

```
const analysis = require('./analysis.js');  
  
weather_data = {  
    time: '01/01/2001',  
    precip: 0.75,  
    temp: 78,  
    //更多天气数据...  
};  
analysis.analyzeWeather(weather_data);
```

在这个文件中，我们使用`require()`来获取`analysis.js`文件。使用时，只需调用赋值给`require`的变量或常量

Chapter 108: Require()

This documentation focuses on explaining the uses and of the `require()` statement that NodeJS includes in their language.

Require is an import of certain files or packages used with NodeJS's modules. It is used to improve code structure and uses. `require()` is used on files that are installed locally, with a direct route from the file that is `require`'ing.

Section 108.1: Beginning require() use with a function and file

Require is a statement that Node interprets as, in some sense, a getter function. For example, say you have a file named `analysis.js`, and the inside of your file looks like this,

```
function analyzeWeather(weather_data) {  
    console.log('Weather information for ' + weather_data.time + ':');  
    console.log('Rainfall: ' + weather_data.precip);  
    console.log('Temperature: ' + weather_data.temp);  
    //More weather_data analysis/printing...  
}
```

This file contains only the method, `analyzeWeather(weather_data)`. If we want to use this function, it must be either used inside of this file, or copied to the file it wants to be used by. However, Node has included a very useful tool to help with code and file organization, which is [modules](#).

In order to utilize our function, we must first `export` the function through a statement at the beginning. Our new file looks like this,

```
module.exports = {  
    analyzeWeather: analyzeWeather  
}  
  
function analyzeWeather(weather_data) {  
    console.log('Weather information for ' + weather_data.time + ':');  
    console.log('Rainfall: ' + weather_data.precip);  
    console.log('Temperature: ' + weather_data.temp);  
    //More weather_data analysis/printing...  
}
```

With this small `module.exports` statement, our function is now ready for use outside of the file. All that is left to do is to use `require()`.

When `require`'ing a function or file, the syntax is very similar. It is usually done at the beginning of the file and set to `var`'s or `const`'s for use throughout the file. For example, we have another file (on the same level as `analyze.js`) named `handleWeather.js` that looks like this,

```
const analysis = require('./analysis.js');  
  
weather_data = {  
    time: '01/01/2001',  
    precip: 0.75,  
    temp: 78,  
    //More weather data...  
};  
analysis.analyzeWeather(weather_data);
```

In this file, we are using `require()` to grab our `analysis.js` file. When used, we just call the variable or constant

并使用其中导出的任何函数。

第108.2节：开始使用NPM包的require()

Node的require在与NPM包配合使用时也非常有用。例如，假设你想在名为getWeather.js的文件中使用NPM包require。通过命令行（git install request）完成NPM安装后，你就可以使用它了。你的getWeather.js文件可能看起来像这样，

```
var https = require('request');

//构建你的url变量...
https.get(url, function(error, response, body) {
  if (error) {
    console.log(error);
  } else {
    console.log('响应 => ' + response);
    console.log('内容 => ' + body);
  }
});
```

当运行此文件时，它首先require（导入）你刚安装的名为request的包。在request文件中，有许多你现在可以访问的函数，其中一个叫做get。在接下来的几行中，使用该函数来发起一个HTTP GET请求。

assigned to this require and use whatever function inside that is exported.

Section 108.2: Beginning require() use with an NPM package

Node's require is also very helpful when used in tandem with an [NPM package](#). Say, for example, you would like to use the NPM package [require](#) in a file named `getWeather.js`. After [NPM installing](#) your package through your command line (`git install request`), you are ready to use it. Your `getWeather.js` file might look this,

```
var https = require('request');

//Construct your url variable...
https.get(url, function(error, response, body) {
  if (error) {
    console.log(error);
  } else {
    console.log('Response => ' + response);
    console.log('Body => ' + body);
  }
});
```

When this file is run, it first require's (imports) the package you just installed called `request`. Inside of the `request` file, there are many functions you now have access to, one of which is called `get`. In the next couple lines, the function is used in order to make an [HTTP GET request](#).

第109章：ExpressJS的路由-控制器-服务结构

第109.1节：模型-路由-控制器-服务目录结构

```
├── 模型
│   └── user.model.js
├── 路由
│   └── user.route.js
├── 服务
│   └── user.service.js
└── 控制器
    └── user.controller.js
```

为了实现模块化的代码结构，逻辑应划分到这些目录和文件中。

Models - 模型的模式定义

Routes - 映射到控制器的API路由

Controllers - 控制器负责处理验证请求参数、查询的所有逻辑，并发送带有正确状态码的响应。

Services - 服务包含数据库查询，返回对象或抛出错误

这样编码者最终会写更多的代码，但代码将更加易于维护且分离明确。

第109.2节：模型-路由-控制器-服务代码结构

user.model.js

```
var mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  name: String
});

const User = mongoose.model('User', UserSchema)

module.exports = User;
```


user.routes.js

```
var express = require('express');
var router = express.Router();

var UserController = require('../controllers/user.controller')
```

Chapter 109: Route-Controller-Service structure for ExpressJS

Section 109.1: Model-Routes-Controllers-Services Directory Structure

```
├── models
│   └── user.model.js
├── routes
│   └── user.route.js
├── services
│   └── user.service.js
└── controllers
    └── user.controller.js
```

For modular code structure the logic should be divided into these directories and files.

Models - The schema definition of the Model

Routes - The API routes maps to the Controllers

Controllers - The controllers handles all the logic behind validating request parameters, query, Sending Responses with correct codes.

Services - The services contains the database queries and returning objects or throwing errors

This coder will end up writing more codes. But at the end the codes will be much more maintainable and separated.

Section 109.2: Model-Routes-Controllers-Services Code Structure

user.model.js

```
var mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  name: String
});
```

```
const User = mongoose.model('User', UserSchema)
```

```
module.exports = User;
```

user.routes.js

```
var express = require('express');
var router = express.Router();

var UserController = require('../controllers/user.controller')
```

```

router.get('/', UserController.getUsers)

module.exports = router;

```

user.controllers.js

```

var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
  // 使用 express-validator 验证请求参数和查询

  var page = req.params.page ? req.params.page : 1;
  var limit = req.params.limit ? req.params.limit : 10;
  try {
    var users = await UserService.getUsers({}, page, limit)
    return res.status(200).json({ status: 200, data: users, message: "成功获取用户" });
  } catch (e) {
    return res.status(400).json({ status: 400, message: e.message });
  }
}

```

user.services.js

```

var User = require('../models/user.model')

exports.getUsers = async function (query, page, limit) {

  try {
    var users = await User.find(query)
    return users;
  } catch (e) {
    // 记录错误日志
    throw Error('分页用户时出错')
  }
}

```

```

router.get('/', UserController.getUsers)

module.exports = router;

```

user.controllers.js

```

var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
  // Validate request parameters, queries using express-validator

  var page = req.params.page ? req.params.page : 1;
  var limit = req.params.limit ? req.params.limit : 10;
  try {
    var users = await UserService.getUsers({}, page, limit)
    return res.status(200).json({ status: 200, data: users, message: "Successfully Users Retrieved" });
  } catch (e) {
    return res.status(400).json({ status: 400, message: e.message });
  }
}

```

user.services.js

```

var User = require('../models/user.model')

exports.getUsers = async function (query, page, limit) {

  try {
    var users = await User.find(query)
    return users;
  } catch (e) {
    // Log Errors
    throw Error('Error while Paginating Users')
  }
}

```

第110章：推送通知

模块/框架

node.js/express Node.js应用的简单后端框架，非常易用且功能强大

Socket.io Socket.IO实现了实时双向基于事件的通信。它适用于所有平台、浏览器或设备，兼顾可靠性和速度。

Push.js 世界上最通用的桌面通知框架

OneSignal 只是苹果设备的另一种推送通知形式

Firebase Firebase 是谷歌的移动平台，帮助你快速开发高质量的应用并发展你的业务。

所以如果你想制作网页应用通知，我建议你使用 Push.js 或 OneSignal 框架来开发网页/移动应用。

Push 是使用 Javascript 通知最快速启动的方法。作为官方规范中相对较新的补充，Notification API 允许现代浏览器如 Chrome、Safari、Firefox 和 IE 9+ 向用户的桌面推送通知。

你需要使用 Socket.io 和一些后端框架，这里我将以 Express 为例。

第 110.1 节：网页通知

首先，你需要安装 Push.js 模块。

```
$ npm install push.js --save
```

或者通过 CDN 将其导入到你的前端应用中

```
<script src="./push.min.js"></script> <!-- CDN 链接 -->
```

完成这些后，你应该就可以开始了。如果你想制作简单的通知，效果应该是这样的：

```
Push.create('Hello World!')
```

我假设你已经知道如何将Socket.io与应用程序设置好。以下是我使用express的后端应用的一些代码示例：

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function (socket) {

  socket.emit('pushNotification', { success: true, msg: 'hello' });

});
```

Chapter 110: Push notifications

module/framework

node.js/express Simple back-end framework for Node.js application, very easy to use and extremely powerful

Socket.io Socket.IO enables real-time bidirectional event-based communication. It works on every platform, browser or device, focusing equally on reliability and speed.

Push.js The world's most versatile desktop notifications framework

OneSignal Just another form off push notifications for Apple devices

Firebase Firebase is Google's mobile platform that helps you quickly develop high-quality apps and grow your business.

So if you wanna make web app notification I suggest you to use Push.js or OneSignal framework for Web/mobile app.

Push is the fastest way to get up and running with Javascript notifications. A fairly new addition to the official specification, the Notification API allows modern browsers such as Chrome, Safari, Firefox, and IE 9+ to push notifications to a user's desktop.

You will have to use Socket.io and some backend framework, I will use Express for this example.

Section 110.1: Web notification

First, you will need to install [Push.js](#) module.

```
$ npm install push.js --save
```

Or import it to your front-end app through [CDN](#)

```
<script src="./push.min.js"></script> <!-- CDN link -->
```

After you are done with that, you should be good to go. This is how it should look like if you wanna make simple notification:

```
Push.create('Hello World!')
```

I will assume that you know how to setup [Socket.io](#) with your app. Here is some code example of my backend app with express:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function (socket) {

  socket.emit('pushNotification', { success: true, msg: 'hello' });

});
```

服务器设置完成后，你应该可以继续进行前端部分。现在我们只需导入Socket.ioCDN，并将以下代码添加到我的index.html文件中：

```
<script src="../socket.io.js"></script> <!-- CDN链接 -->
<script>
var socket = io.connect('http://localhost');
socket.on('pushNotification', function (data) {
  console.log(data);
Push.create("Hello world!", {
  body: data.msg, //这应该打印"hello"
  icon: '/icon.png',
timeout: 4000,
  onClick: function () {
    window.focus();
    this.close();
  }
});
  });
</script>
```

好了，现在你应该能够显示通知了，这也适用于任何安卓设备，如果你想使用Firebase云消息服务，可以配合这个模块使用，这里是由Nick (Push.js的创建者) 编写的示例链接

After your server is all set up, you should be able to move on to front-end stuff. Now all we have to do is import Socket.io [CDN](#) and add this code to my *index.html* file:

```
<script src="../socket.io.js"></script> <!-- CDN link -->
<script>
  var socket = io.connect('http://localhost');
  socket.on('pushNotification', function (data) {
    console.log(data);
    Push.create("Hello world!", {
      body: data.msg, //this should print "hello"
      icon: '/icon.png',
      timeout: 4000,
      onClick: function () {
        window.focus();
        this.close();
      }
    });
  });
</script>
```

There you go, now you should be able to display your notification, this also works on any Android device, and if you wanna use [Firebase](#) cloud messaging, you can use it with this module, [Here](#) is link for that example written by Nick (creator of Push.js)

第110.2节：苹果

请记住，这在苹果设备上不起作用（我没有全部测试过），但如果你想实现推送通知，可以查看OneSignal插件。

Section 110.2: Apple

Keep in mind that this will not work on Apple devices (I didnt test them all), but if you want to make push notifications check [OneSignal](#) plugin.

附录A：安装Node.js

A.1节：使用Node版本管理器 (nvm)

Node版本管理器，简称nvm，是一个bash脚本，用于简化多个Node.js版本的管理。

要安装nvm，请使用提供的安装脚本：

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

对于Windows，有一个带安装程序的 nvm-windows 包。该[GitHub](#)页面包含安装和使用 nvm-windows 包的详细信息。

安装 nvm 后，从命令行运行 "nvm on"。这将启用 nvm 来管理 Node 版本。

注意：您可能需要重启终端，才能识别新安装的 nvm 命令。

然后安装最新的 Node 版本：

```
$ nvm install node
```

您也可以通过传递主版本号、次版本号和/或补丁版本号来安装特定的 Node 版本：

```
$ nvm install 6  
$ nvm install 4.2
```

列出可供安装的版本：

```
$ nvm ls-remote
```

然后，您可以通过传递版本号的方式切换版本，就像安装时一样：

```
$ nvm use 5
```

您可以通过输入以下命令将已安装的特定版本的 Node 设置为默认版本：

```
$ nvm alias default 4.2
```

要显示您机器上已安装的 Node 版本列表，请输入：

```
$ nvm ls
```

要使用项目特定的 Node 版本，您可以将版本保存到 .nvmrc 文件中。这样，在从仓库拉取另一个项目后开始工作时，出错的可能性会更小。

```
$ echo "4.2" > .nvmrc  
$ nvm use  
找到 '/path/to/project/.nvmrc'，版本为 <4.2>  
现在使用 node v4.2 (npm v3.7.3)
```

通过 nvm 安装 Node 时，我们不需要使用 sudo 来安装全局包，因为它们安装在用户主目录中。因此，npm i -g http-server 可以正常运行，不会出现权限错误。

Appendix A: Installing Node.js

Section A.1: Using Node Version Manager (nvm)

[Node Version Manager](#), otherwise known as nvm, is a bash script that simplifies the management of multiple Node.js versions.

To install nvm, use the provided install script:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

For windows there is a nvm-windows package with an installer. This [GitHub](#) page has the details for installing and using the nvm-windows package.

After installing nvm, run "nvm on" from command line. This enables nvm to control the node versions.

Note: You may need to restart your terminal for it to recognize the newly installed nvm command.

Then install the latest Node version:

```
$ nvm install node
```

You can also install a specific Node version, by passing the major, minor, and/or patch versions:

```
$ nvm install 6  
$ nvm install 4.2
```

To list the versions available for install:

```
$ nvm ls-remote
```

You can then switch versions by passing the version the same way you do when installing:

```
$ nvm use 5
```

You can set a specific version of Node that you installed to be the **default version** by entering:

```
$ nvm alias default 4.2
```

To display a list of Node versions that are installed on your machine, enter:

```
$ nvm ls
```

To use project-specific node versions, you can save the version in .nvmrc file. This way, starting to work with another project will be less error-prone after fetching it from its repository.

```
$ echo "4.2" > .nvmrc  
$ nvm use  
Found '/path/to/project/.nvmrc' with version <4.2>  
Now using node v4.2 (npm v3.7.3)
```

When Node is installed via nvm we don't have to use sudo to install global packages since they are installed in home folder. Thus npm i -g http-server works without any permission errors.

附录 A.2：使用包管理器在 Mac 上安装 Node.js

Homebrew

您可以使用 Homebrew 包管理器安装 Node.js。[here](#)

首先更新 brew：

```
brew update
```

您可能需要更改权限或路径。最好在继续之前运行此命令：

```
brew doctor
```

接下来，您可以通过运行以下命令安装 Node.js：

```
brew install node
```

Node.js 安装完成后，您可以通过运行以下命令验证安装的版本：

```
node -v
```

Macports

您也可以通过 Macports 安装 node.js。[here](#)

首先更新它以确保引用的是最新的软件包：

```
sudo port selfupdate
```

然后安装 nodejs 和 npm

```
sudo port install nodejs npm
```

现在你可以通过命令行直接运行 node 来使用 node。同时，你也可以使用以下命令检查当前的 node 版本

```
node -v
```

A.3 节：在 Windows 上安装 Node.js

标准安装

所有 Node.js 的二进制文件、安装程序和源文件都可以从 [here](#) 下载。

你可以只下载 node.exe 运行时，或者使用 Windows 安装程序 (.msi)，它还会安装 npm，这是 Node.js 推荐的包管理器，并配置路径。

通过包管理器安装

你也可以通过包管理器 Chocolatey (软件管理自动化) 进行安装。

```
# choco install nodejs.install
```

关于当前版本的更多信息，你可以在 choco 仓库这里找到。[here](#)

Section A.2: Installing Node.js on Mac using package manager

Homebrew

You can install Node.js using the [Homebrew](#) package manager.

Start by updating brew:

```
brew update
```

You may need to change permissions or paths. It's best to run this before proceeding:

```
brew doctor
```

Next you can install Node.js by running:

```
brew install node
```

Once Node.js is installed, you can validate the version installed by running:

```
node -v
```

Macports

You can also install node.js through [Macports](#).

First update it to make sure the lastest packages are referenced:

```
sudo port selfupdate
```

Then install nodejs and npm

```
sudo port install nodejs npm
```

You can now run node through CLI directly by invoking node. Also, you can check your current node version with

```
node -v
```

Section A.3: Installing Node.js on Windows

Standard installation

All Node.js binaries, installers, and source files can be downloaded [here](#).

You can download just the node.exe runtime or use the Windows installer (.msi), which will also install npm, the recommended package manager for Node.js, and configure paths.

Installation by package manager

You can also install by package manager [Chocolatey](#) (Software Management Automation).

```
# choco install nodejs.install
```

More information about current version, you can find in the choco repository [here](#).

第A.4节：在Ubuntu上安装Node.js

使用apt包管理器

```
sudo apt-get update  
sudo apt-get install nodejs  
sudo apt-get install npm  
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

apt中的node和npm版本较旧。你可以这样更新它们：

```
sudo npm install -g npm  
sudo npm install -g n  
sudo n stable # (或 lts, 或指定版本)
```

直接从 nodesource 使用特定版本的最新版本（例如 LTS 6.x）

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
apt-get install -y nodejs
```

此外，关于正确安装全局 npm 模块的方法，设置它们的个人目录（消除使用 sudo 的需求并避免 EACCES 错误）：

```
mkdir ~/.npm-global  
echo "export PATH=~/npm-global/bin:$PATH" >> ~/.profile  
source ~/.profile  
npm config set prefix '~/.npm-global'
```

A.5 节：使用 n 安装 Node.js

首先，有一个非常好的工具包装器用于在系统上设置 n。只需运行：

```
curl -L https://git.io/n-install | bash
```

安装 n。然后以多种方式安装二进制文件：

最新

n 最新

稳定

n 稳定

长期支持版

任何其他版本

n <版本>

例如 n 4.4.7

如果此版本已安装，此命令将激活该版本。

切换版本

仅输入 n 会显示已安装二进制文件的选择列表。使用上下键找到所需版本，按回车键激活它。

Section A.4: Install Node.js on Ubuntu

Using the apt package manager

```
sudo apt-get update  
sudo apt-get install nodejs  
sudo apt-get install npm  
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

the node & npm versions in apt are outdated. This is how you can update them:

```
sudo npm install -g npm  
sudo npm install -g n  
sudo n stable # (or lts, or a specific version)
```

Using the latest of specific version (e.g. LTS 6.x) directly from nodesource

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
apt-get install -y nodejs
```

Also, for the right way to install global npm modules, set the personal directory for them (eliminates the need for sudo and avoids EACCES errors):

```
mkdir ~/.npm-global  
echo "export PATH=~/npm-global/bin:$PATH" >> ~/.profile  
source ~/.profile  
npm config set prefix '~/.npm-global'
```

Section A.5: Installing Node.js with n

First, there is a really nice wrapper for setting up n on your system. Just run:

```
curl -L https://git.io/n-install | bash
```

to install n. Then install binaries in a variety of ways:

latest

n latest

stable

n stable

lts

n lts

Any other version

n <version>

e.g. n 4.4.7

If this version is already installed, this command will activate that version.

Switching versions

n by itself will produce a selection list of installed binaries. Use up and down to find the one you want and Enter to activate it.

第 A.6 节：使用 APT 包管理器从源码安装 Node.js

先决条件

```
sudo apt-get install build-essential  
sudo apt-get install python
```

[可选]
sudo apt-get install git

获取源码并编译

```
cd ~  
git clone https://github.com/nodejs/node.git
```

或者 获取最新的LTS Node.js版本6.10.2

```
cd ~  
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz  
tar -xzvf node-v6.10.2.tar.gz
```

切换到源码目录，例如 cd ~/node-v6.10.2

```
./configure  
make  
sudo make install
```

A.7节：在Centos、RHEL和

Fedora上从源码安装Node.js

先决条件

- git
- clang 和 clang++ 3.4^ 或 gcc 和 g++ 4.8^
- Python 2.6 或 2.7
- GNU Make 3.81^

获取源代码

Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

构建

```
cd node  
./configure  
make -jX  
su -c make install
```

Section A.6: Install Node.js From Source with APT package manager

Prerequisites

```
sudo apt-get install build-essential  
sudo apt-get install python
```

[optional]
sudo apt-get install git

Get source and build

```
cd ~  
git clone https://github.com/nodejs/node.git
```

OR For the latest LTS Node.js version 6.10.2

```
cd ~  
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz  
tar -xzvf node-v6.10.2.tar.gz
```

Change to the source directory such as in cd ~/node-v6.10.2

```
./configure  
make  
sudo make install
```

Section A.7: Install Node.js from source on Centos, RHEL and Fedora

Prerequisites

- git
- clang and clang++ 3.4^ or gcc and g++ 4.8^
- Python 2.6 or 2.7
- GNU Make 3.81^

Get source

Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

Build

```
cd node  
./configure  
make -jX  
su -c make install
```

X - 处理器核心数，大大加快构建速度

清理 [可选]

```
cd  
rm -rf node
```

第A.8节：在带有Oh My Fish!的Fish Shell下使用Node版本管理器安装

Node版本管理器 (nvm) 极大地简化了Node.js版本的管理和安装，并且在处理包时（例如 `npm install ...`）无需使用 `sudo`。Fish Shell (fish) “是一款适用于OS X、Linux及其他系统的智能且用户友好的命令行shell”，在程序员中作为常见shell如bash的流行替代品。最后，Oh My Fish (omf) 允许在Fish shell中进行自定义和安装包。

本指南假设您已经在使用Fish作为您的shell。

安装nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

安装Oh My Fish

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(注意：此时系统会提示您重启终端。请立即进行重启。)

为 Oh My Fish 安装插件 nvm

我们将通过 Oh My Fish 安装 plugin-nvm，以在 Fish shell 中启用 nvm 功能：

```
omf install nvm
```

使用 Node 版本管理器安装 Node.js

您现在可以使用 nvm 了。您可以安装并使用您喜欢的 Node.js 版本。以下是一些示例：

- 安装最新的 Node 版本：`nvm install node`
- 安装特定版本 6.3.1：`nvm install 6.3.1`
- 列出已安装的版本：`nvm ls`
- 切换到之前安装的 4.3.1 版本：`nvm use 4.3.1`

最后说明

再次提醒，使用此方法管理 Node.js 时不再需要 `sudo`！Node 版本、包等都安装在您的主目录中。

附录 A.9：在树莓派上安装 Node.js

要安装 v6.x，请先更新软件包

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

使用apt包管理器

X - the number of processor cores, greatly speeds up the build

Cleanup [Optional]

```
cd  
rm -rf node
```

Section A.8: Installing with Node Version Manager under Fish Shell with Oh My Fish!

Node Version Manager (nvm) greatly simplifies the management of Node.js versions, their installation, and removes the need for sudo when dealing with packages (e.g. `npm install ...`). Fish Shell (fish) "is a smart and user-friendly command line shell for OS X, Linux, and the rest of the family" that is a popular alternative among programmers to common shells such as bash. Lastly, Oh My Fish (omf) allows for customizing and installing packages within Fish shell.

This guide assumes you are already using Fish as your shell.

Install nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

Install Oh My Fish

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(Note: You will be prompted to restart your terminal at this point. Go ahead and do so now.)

Install plugin-nvm for Oh My Fish

We will install `plugin-nvm` via Oh My Fish to expose nvm capabilities within the Fish shell:

```
omf install nvm
```

Install Node.js with Node Version Manager

You are now ready to use nvm. You may install and use the version of Node.js of your liking. Some examples:

- Install the most recent Node version: `nvm install node`
- Install 6.3.1 specifically: `nvm install 6.3.1`
- List installed versions: `nvm ls`
- Switch to a previously installed 4.3.1: `nvm use 4.3.1`

Final Notes

Remember again, that we no longer need sudo when dealing with Node.js using this method! Node versions, packages, and so on are installed in your home directory.

Section A.9: Installing Node.js on Raspberry Pi

To install v6.x update the packages

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

Using the apt package manager

```
sudo apt-get install -y nodejs
```

belindoc.com

```
sudo apt-get install -y nodejs
```

致谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

阿卜杜勒·阿齐兹·莫克纳切	第一章
艾奥林加门费尔	第一章和第四章
艾哈迈德·梅特瓦利	第20章
艾肯·莫格怀	第3章、第26章和第31章
aisflat439	第89章
阿吉特杰·考希克	第54章
akinjide	第23章
亚历克斯·洛根	第17章和第24章
alex2	第3章
阿利斯特	第33章
阿尔约沙·迈耶	第一章
阿米拉·桑帕斯	第一章
阿米纳达夫	第5章、第75章和第111章
阿姆里什·泰亚吉	第2章
安德烈斯·恩卡纳西翁	第75章
安德烈斯·C·维斯卡	第3章
安德鲁·布鲁克	第95章
安德鲁·莱昂纳迪	第6章
安高	第111章
安基特·戈姆卡莱	第一章
安库尔·阿南德	第1、2、10、33和48章
安藤	第50章
阿夫	第3章
Apidcloud	第22章
弧	第106章
阿萨夫·马纳森	第3和10章
asherbar	第111章
阿泰斯·戈拉尔	第2章
B Thuy	第57章
巴兰斯基斯塔德	第一章
巴茨	第3章和第111章
贝林顿	第17章
本杰明	第8章
贝肖伊·汉纳	第65章
巴拉特	第6章
大块头	第67章
布雷特·杰克逊	第22章
brianmearns	第3章
布祖特	第91、92和111章
ccnokes	第2章
CD..	第2章
Chance Snow	第10章
Chezzwizz	第1章和第111章
克里斯托夫·马罗瓦 (Christophe Marois)	第4章
克里斯托弗·罗宁 (Christopher Ronning)	第一章
CJ Harries	第107章
克莱门特·雅各布 (Clement JACOB)	第14章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Abdelaziz Mokhnache	Chapter 1
Aeolingamenfel	Chapters 1 and 4
Ahmed Metwally	Chapter 20
Aikon Mogwai	Chapters 3, 26 and 31
aisflat439	Chapter 89
Ajitej Kaushik	Chapter 54
akinjide	Chapter 23
Alex Logan	Chapters 17 and 24
alex2	Chapter 3
Alister	Chapter 33
Aljoscha Meyer	Chapter 1
Amila Sampath	Chapter 1
Aminadav	Chapters 5, 75 and 111
Amreesh Tyagi	Chapter 2
Andrés Encarnación	Chapter 75
Andres C. Viesca	Chapter 3
Andrew Brooke	Chapter 95
AndrewLeonardi	Chapter 6
Anh Cao	Chapter 111
Ankit Gomkale	Chapter 1
Ankur Anand	Chapters 1, 2, 10, 33 and 48
Antenka	Chapter 50
Aph	Chapter 3
Apidcloud	Chapter 22
arcs	Chapter 106
Asaf Manassen	Chapters 3 and 10
asherbar	Chapter 111
Ates Goral	Chapter 2
B Thuy	Chapter 57
baranskistad	Chapter 1
Batsu	Chapters 3 and 111
Bearington	Chapter 17
Benjamin	Chapter 8
Beshoy Hanna	Chapter 65
Bharat	Chapter 6
Big Dude	Chapter 67
Brett Jackson	Chapter 22
brianmearns	Chapter 3
Buzut	Chapters 91, 92 and 111
ccnokes	Chapter 2
CD..	Chapter 2
Chance Snow	Chapter 10
Chezzwizz	Chapters 1 and 111
Christophe Marois	Chapter 4
Christopher Ronning	Chapter 1
CJ Harries	Chapter 107
Clement JACOB	Chapter 14

[commonSenseCode](#)
[cyanbeam](#)
[受损有机物](#)
[damitj07](#)
[丹尼尔·维雷姆](#)
[黑马](#)
[戴夫](#)
[大卫·G.](#)
[大卫·加蒂](#)
[大卫·徐](#)
[迪恩·拉瑟](#)
[devnull69](#)
[迪吉厄斯](#)
[德米特里·鲍里索夫](#)
[多姆·维尼亞德](#)
[多米尼克·瓦伦西亚纳](#)
[德拉卡SAN](#)
[dthree](#)
[杜利·金斯基](#)
[duncanhall](#)
[efeder](#)
[埃里克·福尔坦](#)
[埃里克·斯梅肯斯](#)
[evalsocket](#)
[Everettss](#)
[爆炸药丸](#)
[F.考德尔](#)
[法比安库克](#)
[菲克拉](#)
[弗洛里安·哈默勒](#)
[福里文](#)
[弗雷迪·科尔曼](#)
[fresh5447](#)
[gentlejo](#)
[GilZ](#)
[gnerkus](#)
[guleria](#)
[H. Pauwelyn](#)
[长篇大论](#)
[哈桑·A·尤塞夫](#)
[六氟化物](#)
[希玛尼·阿格拉瓦尔](#)
[侯塞姆·亚希奥伊](#)
[HungryCoder](#)
[冰人](#)
[伊南奇·古穆斯](#)
[Ionică Bizău](#)
[iSkore](#)
[伊万·赫里斯托夫](#)
[雅采克·拉布达](#)
[jakerella](#)
[詹姆斯·比林厄姆](#)
[詹姆斯·泰勒](#)

[commonSenseCode](#)
[cyanbeam](#)
[Damaged Organic](#)
[damitj07](#)
[Daniel Verem](#)
[Dark Horse](#)
[Dave](#)
[David G.](#)
[David Gatti](#)
[David Xu](#)
[Dean Rather](#)
[devnull69](#)
[Djizeus](#)
[Dmitriy Borisov](#)
[Dom Vinyard](#)
[DominicValenciana](#)
[DrakaSAN](#)
[dthree](#)
[Duly Kinsky](#)
[duncanhall](#)
[efeder](#)
[Eric Fortin](#)
[Eric Smekens](#)
[evalsocket](#)
[Everettss](#)
[Explosion Pills](#)
[F. Kauder](#)
[FabianCook](#)
[Fikra](#)
[Florian Hämerle](#)
[Forivin](#)
[Freddie Coleman](#)
[fresh5447](#)
[gentlejo](#)
[GilZ](#)
[gnerkus](#)
[guleria](#)
[H. Pauwelyn](#)
[Harangue](#)
[Hasan A Yousef](#)
[hexacyanide](#)
[Himani Agrawal](#)
[Houssem Yahiaoui](#)
[HungryCoder](#)
[Iceman](#)
[Inanc Gumus](#)
[Ionică Bizău](#)
[iSkore](#)
[Ivan Hristov](#)
[Jacek Labuda](#)
[jakerella](#)
[James Billingham](#)
[James Taylor](#)

jamescostian
杰森
贾斯珀
jdrydn
jemiloii
杰里米·班克斯
杰瑞
若昂·安德拉德
joeyfb
约翰
约翰·斯莱格斯
约翰·文森特·贾丁
johni
JohnnyCoder
乔纳斯·S
乔什
约书亚·克莱维特
卡皮尔·瓦茨
卡伦
凯鲁姆·塞纳纳亚克
克伦特·辛格
科内尔
莱奥·马丁
劳里斯
卢菲卢夫
路易斯·巴兰凯罗
路易斯·冈萨雷斯
M. A. 科尔代罗
m02ph3u5
M1kstur
马切伊·罗斯塔ński
马纳斯·贾扬特
曼努埃尔
manuerumx
马里奥·罗齐奇
马修·哈伍德
马修·珊利
马修·勒莫因
蒙蒂市长
我的世界
mezzode
迈克尔·布恩
midnightsyntax
米哈伊尔
MindlessRanger
Mindsers
莫希特·甘格拉德
MSB
穆克什·夏尔马
纳伊姆·谢赫
奈内什·拉瓦尔
本地编码器
ndugger

第11章
第5章
第一章
第5章和第75章
第3章
第一章
第25章
第2章
第2章
第85章
第1章和第23章
第18、19和111章
第21章
第75章
第12章
第2章
第12章
第75章
第80章
第82章和第83章
第13章、第16章、第21章、第22章、第75章、第76章和第81章
第10章
第69章
第10章
第5章
第4章和第13章
第19章
第69章
第2章和第5章
第88章
第111章
第29章
第60章
第24章
第110章
第2章
第33章
第22章
第8章
第一章
第10章
第14章
第4章、第28章和第68章
第2章、第3章、第4章、第13章、第24章和第26章
第11章
第2章
第3章
第5章
第3、8、22、23和38章
第3、24和43章
第49章
第85章
第一章

jamescostian
Jason
Jasper
jdrydn
jemiloii
Jeremy Banks
jerry
João Andrade
joeyfb
John
John Slegers
John Vincent Jardin
johni
JohnnyCoder
Jonas S
Josh
Joshua Kleveter
Kapil Vats
Karlen
Kelum Senanayake
KlwntSingh
Kornel
Léo Martin
lauriys
Loufylouf
Louis Barranqueiro
Luis González
M. A. Cordeiro
m02ph3u5
M1kstur
Maciej Rostański
Manas Jayanth
Manuel
manuerumx
Mario Rozic
Matthew Harwood
Matthew Shanley
MatthieuLemoine
MayorMonty
Meinkraft
mezzode
Michael Buen
midnightsyntax
Mikhail
MindlessRanger
Mindsers
Mohit Gangrade
MSB
Mukesh Sharma
Naeem Shaikh
Nainesh Raval
Native Coder
ndugger

Chapter 11
Chapter 5
Chapter 1
Chapters 5 and 75
Chapter 3
Chapter 1
Chapter 25
Chapter 2
Chapter 2
Chapter 85
Chapters 1 and 23
Chapters 18, 19 and 111
Chapter 21
Chapter 75
Chapter 12
Chapter 2
Chapter 12
Chapter 75
Chapter 80
Chapters 82 and 83
Chapters 13, 16, 21, 22, 75, 76 and 81
Chapter 10
Chapter 69
Chapter 10
Chapter 5
Chapters 4 and 13
Chapter 19
Chapter 69
Chapters 2 and 5
Chapter 88
Chapter 111
Chapter 29
Chapter 60
Chapter 24
Chapter 110
Chapter 2
Chapter 33
Chapter 22
Chapter 8
Chapter 1
Chapter 10
Chapter 14
Chapters 4, 28 and 68
Chapters 2, 3, 4, 13, 24 and 26
Chapter 11
Chapter 2
Chapter 3
Chapter 5
Chapters 3, 8, 22, 23 and 38
Chapters 3, 24 and 43
Chapter 49
Chapter 85
Chapter 1

尼克
尼克斯
[尼罗山·拉纳帕蒂](#)
尼维什
[诺曼宾·侯赛因](#)
新手
[数字8er](#)
新潮者
[oliolioli](#)
奥利弗
[omgimanerd](#)
擎天柱
[奥萨马·巴里](#)
帕尔哈姆·阿尔瓦尼
[帕拉德·诺帕内](#)
[帕沙·鲁姆金](#)
[保利·加西亚](#)
[佩德罗·奥特罗](#)
[peteb](#)
[彼得·G](#)
[菲利普·科内利乌斯·格洛弗](#)
[菲利普·弗伦克尔](#)
[皮特·赫罗伦](#)
[pranspach](#)
吴琼
[QoP](#)
[Quill](#)
[拉法尔·维利ński](#)
[拉古](#)
[rdegges](#)
红色
[refaelos](#)
里克
[里希凯什·钱德拉](#)
[riyadhalnur](#)
[里佐夫斯基](#)
[罗比](#)
[鲁佩什](#)
[罗恩·哈利](#)
[鲁帕利·佩马雷](#)
[萨米尔·斯里瓦斯塔瓦](#)
[桑凯斯·卡塔](#)
[萨蒂什](#)
[萨蒂亚姆·S](#)
[sBanda](#)
[ScientiaEtVeritas](#)
[沙宾·哈希姆](#)
[希卡尔·班萨尔](#)
[施里加内什·科尔赫](#)
[西比什·韦努](#)
[悉达特·斯里瓦斯塔瓦](#)
[西格弗里德](#)
[信号](#)

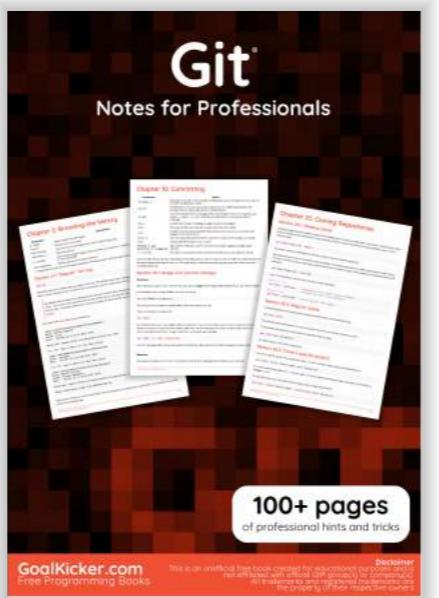
第1和2章
第3章
第70、71、73和74章
第3、12和16章
第60和109章
第3章
第2章
第111章
第78章
第6章
第85章
第17章
第17章
第63章和第97章
第105章
第100章
第3章
第23章
第5章
第4、9和40章
第23章
第108章
第2章
第2章
第19章和第23章
第24章
第2章
第2章
第2章和第3章
第22章
第2章
第84章
第2章
第52章
第一章
第8章和第16章
第2章
第10章
第90章
第44章
第69章
第17章
第14章
第39章
第68章
第13、16和104章
第36章
第1、3、5、10和111章
第69章
第25和49章
第一章
第111章
第99章
第58章

[Nick](#)
[Niklas](#)
[Niroshan Ranapathi](#)
[Nivesh](#)
[nomanbinhussein](#)
[noob](#)
[num8er](#)
[NuSkooler](#)
[oliolioli](#)
[Oliver](#)
[omgimanerd](#)
[optimus](#)
[Osama Bari](#)
[Parham Alvani](#)
[parlad neupane](#)
[Pasha Rumkin](#)
[Pauly Garcia](#)
[Pedro Otero](#)
[peteb](#)
[Peter G](#)
[Philip Cornelius Glover](#)
[Philipp Flenker](#)
[Pieter Herroelen](#)
[pranspach](#)
[Qiong Wu](#)
[QoP](#)
[Quill](#)
[Rafal Wiliński](#)
[raghu](#)
[rdegges](#)
[Red](#)
[refaelos](#)
[Rick](#)
[Rishikesh Chandra](#)
[riyadhalnur](#)
[Rizowski](#)
[Robbie](#)
[Roopesh](#)
[Rowan Harley](#)
[Rupali Pemare](#)
[Sameer Srivastava](#)
[Sanketh Katta](#)
[Sathish](#)
[Satyam S](#)
[sBanda](#)
[ScientiaEtVeritas](#)
[Shabin Hashim](#)
[shikhar bansal](#)
[Shriganesh Kolhe](#)
[Sibeesh Venu](#)
[Siddharth Srivastva](#)
[sigfried](#)
[signal](#)

Chapters 1 and 2
Chapter 3
Chapters 70, 71, 73 and 74
Chapters 3, 12 and 16
Chapters 60 and 109
Chapter 3
Chapter 2
Chapter 111
Chapter 78
Chapter 6
Chapter 85
Chapter 17
Chapters 63 and 97
Chapter 105
Chapter 100
Chapter 3
Chapter 23
Chapter 5
Chapters 4, 9 and 40
Chapter 23
Chapter 108
Chapter 2
Chapter 2
Chapter 2
Chapters 19 and 23
Chapter 24
Chapter 2
Chapter 2
Chapters 2 and 3
Chapter 22
Chapter 2
Chapter 84
Chapter 2
Chapter 52
Chapter 1
Chapters 8 and 16
Chapter 2
Chapter 10
Chapter 90
Chapter 44
Chapter 69
Chapter 17
Chapter 14
Chapter 39
Chapter 68
Chapters 13, 16 and 104
Chapter 36
Chapters 1, 3, 5, 10 and 111
Chapter 69
Chapters 25 and 49
Chapter 1
Chapter 111
Chapter 99
Chapter 58

Simplans	第10章	Simplans	Chapter 10
sjmarshy	第16章和第22章	sjmarshy	Chapters 16 and 22
Skanda	第2章	Skanda	Chapter 2
skiila	第62章和第95章	skiila	Chapters 62 and 95
天空	第25章	Sky	Chapter 25
snuggles08	第18章	snuggles08	Chapter 18
索朗瓦拉·阿巴萨利	第75章	Sorangwala Abbasali	Chapter 75
史蒂夫·莱西	第10章	SteveLacy	Chapter 10
仍在学习	第2章和第4章	still_learning	Chapters 2 and 4
萨布	第2章	subbu	Chapter 2
斯文_31415	第10章	Sven_31415	Chapter 10
斯维拉图姆	第111章	Sveratum	Chapter 111
沙姆·普拉迪普	第86章	Syam Pradeep	Chapter 86
SynapseTech	第64章	SynapseTech	Chapter 64
tandrewnichols	第3章和第111章	tandrewnichols	Chapters 3 and 111
the12	第2章	the12	Chapter 2
thefourtheye	第一章	thefourtheye	Chapter 1
theunexpected1	第一章	theunexpected1	Chapter 1
蒂姆·琼斯	第4章	Tim_Jones	Chapter 4
tlo	第2章	tlo	Chapter 2
托马斯·卡尼巴诺	第10章	Tomás Cañibano	Chapter 10
托尼·维列纳	第60章	Toni Villena	Chapter 60
topheman	第16章	topheman	Chapter 16
图沙尔·古普塔	第19章	Tushar Gupta	Chapter 19
tverdohleb	第3章和第22章	tverdohleb	Chapters 3 and 22
tyehia	第22章、第96章和第103章	tyehia	Chapters 22, 96 and 103
umesh	第7章	umesh	Chapter 7
user2314737	第1章和第56章	user2314737	Chapters 1 and 56
user6939352	第10章	user6939352	Chapter 10
uzaif	第2章和第5章	uzaif	Chapters 2 and 5
V1P3R	第10章和第111章	V1P3R	Chapters 10 and 111
Veger	第79章	Veger	Chapter 79
victorkohl	第10章和第111章	victorkohl	Chapters 10 and 111
vintprojkt	第72章	vintprojkt	Chapter 72
VladNeacsu	第2章	VladNeacsu	Chapter 2
VooVoo	第52章	VooVoo	Chapter 52
弗谢沃洛德·戈洛维兹宁	第37章	Vsevolod Goloviznin	Chapter 37
vsjn3290ckjnaojij2jikndckjb	第45章	vsjn3290ckjnaojij2jikndckjb	Chapter 45
水卷	第43章	Waterscroll	Chapter 43
遗嘱	第24章	Will	Chapter 24
威廉·卡伦	第87章	William Carron	Chapter 87
xam	第1、35和73章	xam	Chapters 1, 35 and 73
xims	第3章	xims	Chapter 3
耶尔科·帕尔马	第2章和第3章	Yerko Palma	Chapters 2 and 3
ymz	第35章	ymz	Chapter 35
yrtimiD	第46章	yrtimiD	Chapter 46
扎农	第4章	Zanon	Chapter 4
泽鲁比乌斯	第21章和第41章	Ze Rubeus	Chapters 21 and 41
佐尔坦·施密特	第47章	Zoltán Schmidt	Chapter 47
zurfyx	第12章、第35章和第53章	zurfyx	Chapters 12, 35 and 53

你可能也喜欢



You may also like

