**Node.js 教程**



简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行Javascript的速度非常快，性能非常好。

**谁适合阅读本教程？**

如果你是一个前端程序员，你不懂得像PHP、Python或Ruby等动态编程语言，然后你想创建自己的服务，那么Node.js是一个非常好的选择。

Node.js 是运行在服务端的 JavaScript，如果你熟悉Javascript，那么你将会很容易的学会Node.js。

当然，如果你是后端程序员，想部署一些高性能的服务，那么学习Node.js也是一个非常好的选择。

**学习本教程前你需要了解**

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Javascript,PHP，Java等编程语言，将有助于你更快的了解Node.js编程。

**使用的版本**

我们可以使用以下命令来查看当前的 Node 版本：

$ node -v

v4.4.3

**注意：**不同版本间可能是有差异的。

**第一个Node.js程序：Hello World！**

**脚本模式**

以下是我们的第一个Node.js程序：

**实例**

console.log("Hello World");

[**运行实例 »**](http://www.runoob.com/try/shownodejs.php?filename=helloworld)

保存该文件，文件名为 helloworld.js， 并通过 node命令来执行：

node helloworld.js

程序执行后，正常的话，就会在终端输出 Hello World。

**交互模式**

打开终端，键入node进入命令交互模式可以输入一条代码语句后立即执行并显示结果，例如：

$ node

> console.log('Hello World!');

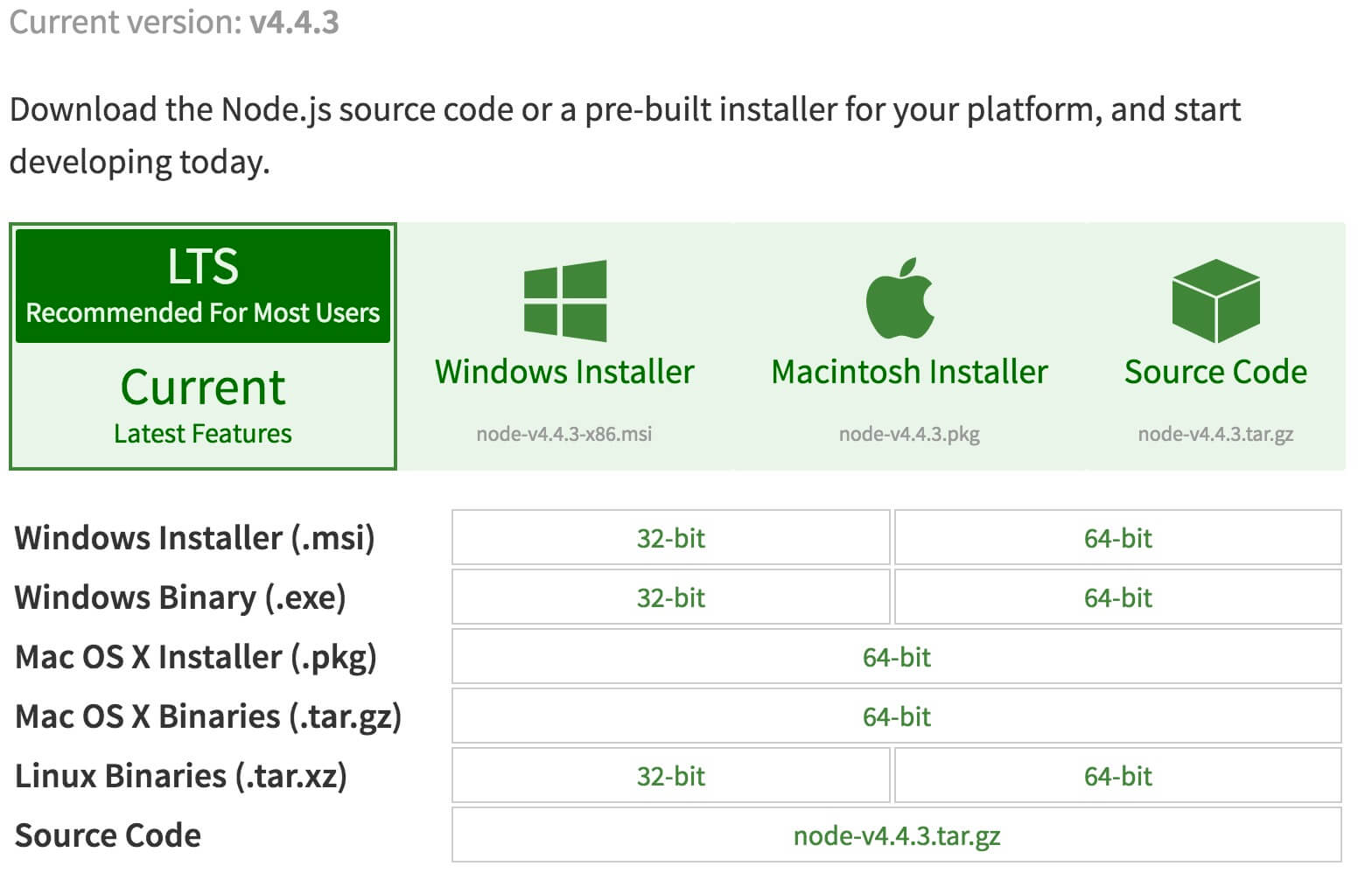
Hello World!

**Node.js 安装配置**

本章节我们将向大家介绍在window和Linux上安装Node.js的方法。

本安装教程以Node.js v4.4.3 LTS(长期支持版本)版本为例。

Node.js安装包及源码下载地址为：<https://nodejs.org/en/download/>。



你可以根据不同平台系统选择你需要的Node.js安装包。

Node.js 历史版本下载地址：<https://nodejs.org/dist/>

**注意：**Linux上安装Node.js需要安装Python 2.6 或 2.7 ，不建议安装Python 3.0以上版本。

**Window 上安装Node.js**

你可以采用以下两种方式来安装。

**1、Windows 安装包(.msi)**

32 位安装包下载地址 : <https://nodejs.org/dist/v4.4.3/node-v4.4.3-x86.msi>

64 位安装包下载地址 : <https://nodejs.org/dist/v4.4.3/node-v4.4.3-x64.msi>

本文实例以 v0.10.26 版本为例，其他版本类似， 安装步骤：

步骤 1 : 双击下载后的安装包 **v0.10.26**，如下所示：



步骤 2 : 点击以上的Run(运行)，将出现如下界面：



步骤 3 : 勾选接受协议选项，点击 next（下一步） 按钮 :



步骤 4 : Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录，并点击 next（下一步）：



步骤 5 : 点击树形图标来选择你需要的安装模式 , 然后点击下一步 next（下一步）



步骤 6 :点击 Install（安装） 开始安装Node.js。你也可以点击 Back（返回）来修改先前的配置。 然后并点击 next（下一步）：



安装过程：



点击 Finish（完成）按钮退出安装向导。



检测PATH环境变量是否配置了Node.js，点击开始=》运行=》输入"cmd" => 输入命令"path"，输出如下结果：

PATH=C:\oraclexe\app\oracle\product\10.2.0\server\bin;C:\Windows\system32;

C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;

c:\python32\python;C:\MinGW\bin;C:\Program Files\GTK2-Runtime\lib;

C:\Program Files\MySQL\MySQL Server 5.5\bin;C:\Program Files\nodejs\;

C:\Users\rg\AppData\Roaming\npm

我们可以看到环境变量中已经包含了C:\Program Files\nodejs\

检查Node.js版本

node-version-test

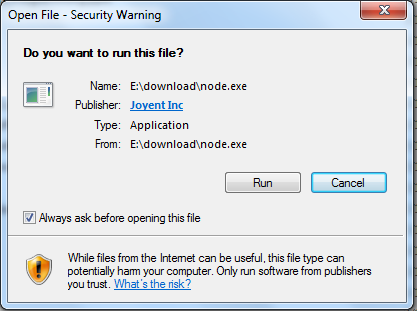
**2、Windows 二进制文件 (.exe)安装**

32 位安装包下载地址 : <http://nodejs.org/dist/v0.10.26/node.exe>

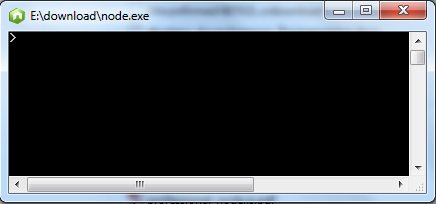
64 位安装包下载地址 : <http://nodejs.org/dist/v0.10.26/x64/node.exe>

**安装步骤**

步骤 1 : 双击下载的安装包 Node.exe ，将出现如下界面 :



点击 Run（运行）按钮将出现命令行窗口：



**版本测试**

进入 node.exe 所在的目录，如下所示：

node-version

如果你获得以上输出结果，说明你已经成功安装了Node.js。

**Ubuntu 上安装 Node.js**

**Node.js 源码安装**

以下部分我们将介绍在Ubuntu Linux下安装 Node.js 。 其他的Linux系统，如Centos等类似如下安装步骤。

在 Github 上获取 Node.js 源码：

$ sudo git clone https://github.com/nodejs/node.git

Cloning into 'node'...

修改目录权限：

$ sudo chmod -R 755 node

使用 **./configure** 创建编译文件，并按照：

$ cd node

$ sudo ./configure

$ sudo make

$ sudo make install

查看 node 版本：

$ node --version

v0.10.25

**Ubuntu apt-get命令安装**

命令格式如下：

sudo apt-get install nodejs

sudo apt-get install npm

**CentOS 下安装 Node.js**

1、下载源码，你需要在<https://nodejs.org/en/download/>下载最新的Nodejs版本，本文以v0.10.24为例:

cd /usr/local/src/

wget http://nodejs.org/dist/v0.10.24/node-v0.10.24.tar.gz

2、解压源码

tar zxvf node-v0.10.24.tar.gz

3、 编译安装

cd node-v0.10.24

./configure --prefix=/usr/local/node/0.10.24

make

make install

4、 配置NODE\_HOME，进入profile编辑环境变量

vim /etc/profile

设置nodejs环境变量，在 ***export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL*** 一行的上面添加如下内容:

#set for nodejs

export NODE\_HOME=/usr/local/node/0.10.24

export PATH=$NODE\_HOME/bin:$PATH

:wq保存并退出，编译/etc/profile 使配置生效

source /etc/profile

验证是否安装配置成功

node -v

输出 v0.10.24 表示配置成功

npm模块安装路径

/usr/local/node/0.10.24/lib/node\_modules/

**注：**Nodejs 官网提供了编译好的Linux二进制包，你也可以下载下来直接应用。

**Node.js 创建第一个应用**

如果我们使用PHP来编写后端的代码时，需要Apache 或者 Nginx 的HTTP 服务器，并配上 mod\_php5 模块和php-cgi。

从这个角度看，整个"接收 HTTP 请求并提供 Web 页面"的需求根本不需 要 PHP 来处理。

不过对 Node.js 来说，概念完全不一样了。使用 Node.js 时，我们不仅仅在实现一个应用，同时还实现了整个 HTTP 服务器。事实上，我们的 Web 应用以及对应的 Web 服务器基本上是一样的。

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

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

**创建 Node.js 应用**

**步骤一、引入 required 模块**

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

var http = require("http");

**步骤二、创建服务器**

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

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

var http = require('http');

http.createServer(function (request, response) {

// 发送 HTTP 头部

// HTTP 状态值: 200 : OK

// 内容类型: text/plain

response.writeHead(200, {'Content-Type': 'text/plain'});

// 发送响应数据 "Hello World"

response.end('Hello World\n');

}).listen(8888);

// 终端打印如下信息，因为是异步所以这句会执行

console.log('Server running at http://127.0.0.1:8888/');

以上代码我们完成了一个可以工作的 HTTP 服务器。

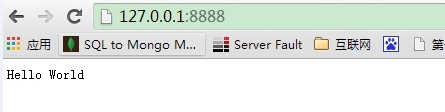
使用 **node** 命令执行以上的代码：

node server.js

Server running at http://127.0.0.1:8888/

cmdrun

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



**分析Node.js 的 HTTP 服务器：**

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

**NPM 使用介绍**

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

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

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 **"npm -v"**来测试是否成功安装。命令如下，出现版本提示表示安装成功:

$ npm -v

2.3.0

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

$ sudo npm install npm -g

/usr/local/bin/npm -> /usr/local/lib/node\_modules/npm/bin/npm-cli.js

npm@2.14.2 /usr/local/lib/node\_modules/npm

如果是 Window 系统使用以下命令即可：

npm install npm -g

*使用淘宝镜像的命令：*

cnpm install npm -g

**使用 npm 命令安装模块**

npm 安装 Node.js 模块语法格式如下：

$ npm install <Module Name>

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

$ npm install express

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

var express = require('express');

**全局安装与本地安装**

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

npm install express # 本地安装

npm install express -g # 全局安装

如果出现以下错误：

npm err! Error: connect ECONNREFUSED 127.0.0.1:8087

解决办法为：

$ npm config set proxy null

**本地安装**

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

**全局安装**

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

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

接下来我们使用全局方式安装 express

$ npm install express -g

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

express@4.13.3 node\_modules/express

├── escape-html@1.0.2

├── range-parser@1.0.2

├── merge-descriptors@1.0.0

├── array-flatten@1.1.1

├── cookie@0.1.3

├── utils-merge@1.0.0

├── parseurl@1.3.0

├── cookie-signature@1.0.6

├── methods@1.1.1

├── fresh@0.3.0

├── vary@1.0.1

├── path-to-regexp@0.1.7

├── content-type@1.0.1

├── etag@1.7.0

├── serve-static@1.10.0

├── content-disposition@0.5.0

├── depd@1.0.1

├── qs@4.0.0

├── finalhandler@0.4.0 (unpipe@1.0.0)

├── on-finished@2.3.0 (ee-first@1.1.1)

├── proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)

├── debug@2.2.0 (ms@0.7.1)

├── type-is@1.6.8 (media-typer@0.3.0, mime-types@2.1.6)

├── accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.6)

└── send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-errors@1.3.1)

**查看安装信息**

你可以使用以下命令来查看所有全局安装的模块：

$ npm list -g

├─┬ cnpm@4.3.2

│ ├── auto-correct@1.0.0

│ ├── bagpipe@0.3.5

│ ├── colors@1.1.2

│ ├─┬ commander@2.9.0

│ │ └── graceful-readlink@1.0.1

│ ├─┬ cross-spawn@0.2.9

│ │ └── lru-cache@2.7.3

……

如果要查看某个模块的版本号，可以使用命令如下：

$ npm list grunt

projectName@projectVersion /path/to/project/folder

└── grunt@0.4.1

**使用 package.json**

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

{

"name": "express",

"description": "Fast, unopinionated, minimalist web framework",

"version": "4.13.3",

"author": {

"name": "TJ Holowaychuk",

"email": "tj@vision-media.ca"

},

"contributors": [

{

"name": "Aaron Heckmann",

"email": "aaron.heckmann+github@gmail.com"

},

{

"name": "Ciaran Jessup",

"email": "ciaranj@gmail.com"

},

{

"name": "Douglas Christopher Wilson",

"email": "doug@somethingdoug.com"

},

{

"name": "Guillermo Rauch",

"email": "rauchg@gmail.com"

},

{

"name": "Jonathan Ong",

"email": "me@jongleberry.com"

},

{

"name": "Roman Shtylman",

"email": "shtylman+expressjs@gmail.com"

},

{

"name": "Young Jae Sim",

"email": "hanul@hanul.me"

}

],

"license": "MIT",

"repository": {

"type": "git",

"url": "git+https://github.com/strongloop/express.git"

},

"homepage": "http://expressjs.com/",

"keywords": [

"express",

"framework",

"sinatra",

"web",

"rest",

"restful",

"router",

"app",

"api"

],

"dependencies": {

"accepts": "~1.2.12",

"array-flatten": "1.1.1",

"content-disposition": "0.5.0",

"content-type": "~1.0.1",

"cookie": "0.1.3",

"cookie-signature": "1.0.6",

"debug": "~2.2.0",

"depd": "~1.0.1",

"escape-html": "1.0.2",

"etag": "~1.7.0",

"finalhandler": "0.4.0",

"fresh": "0.3.0",

"merge-descriptors": "1.0.0",

"methods": "~1.1.1",

"on-finished": "~2.3.0",

"parseurl": "~1.3.0",

"path-to-regexp": "0.1.7",

"proxy-addr": "~1.0.8",

"qs": "4.0.0",

"range-parser": "~1.0.2",

"send": "0.13.0",

"serve-static": "~1.10.0",

"type-is": "~1.6.6",

"utils-merge": "1.0.0",

"vary": "~1.0.1"

},

"devDependencies": {

"after": "0.8.1",

"ejs": "2.3.3",

"istanbul": "0.3.17",

"marked": "0.3.5",

"mocha": "2.2.5",

"should": "7.0.2",

"supertest": "1.0.1",

"body-parser": "~1.13.3",

"connect-redis": "~2.4.1",

"cookie-parser": "~1.3.5",

"cookie-session": "~1.2.0",

"express-session": "~1.11.3",

"jade": "~1.11.0",

"method-override": "~2.3.5",

"morgan": "~1.6.1",

"multiparty": "~4.1.2",

"vhost": "~3.0.1"

},

"engines": {

"node": ">= 0.10.0"

},

"files": [

"LICENSE",

"History.md",

"Readme.md",

"index.js",

"lib/"

],

"scripts": {

"test": "mocha --require test/support/env --reporter spec --bail --check-leaks test/ test/acceptance/",

"test-ci": "istanbul cover node\_modules/mocha/bin/\_mocha --report lcovonly -- --require test/support/env --reporter spec --check-leaks test/ test/acceptance/",

"test-cov": "istanbul cover node\_modules/mocha/bin/\_mocha -- --require test/support/env --reporter dot --check-leaks test/ test/acceptance/",

"test-tap": "mocha --require test/support/env --reporter tap --check-leaks test/ test/acceptance/"

},

"gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",

"bugs": {

"url": "https://github.com/strongloop/express/issues"

},

"\_id": "express@4.13.3",

"\_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",

"\_from": "express@\*",

"\_npmVersion": "1.4.28",

"\_npmUser": {

"name": "dougwilson",

"email": "doug@somethingdoug.com"

},

"maintainers": [

{

"name": "tjholowaychuk",

"email": "tj@vision-media.ca"

},

{

"name": "jongleberry",

"email": "jonathanrichardong@gmail.com"

},

{

"name": "dougwilson",

"email": "doug@somethingdoug.com"

},

{

"name": "rfeng",

"email": "enjoyjava@gmail.com"

},

{

"name": "aredridel",

"email": "aredridel@dinhe.net"

},

{

"name": "strongloop",

"email": "callback@strongloop.com"

},

{

"name": "defunctzombie",

"email": "shtylman@gmail.com"

}

],

"dist": {

"shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",

"tarball": "http://registry.npmjs.org/express/-/express-4.13.3.tgz"

},

"directories": {},

"\_resolved": "https://registry.npmjs.org/express/-/express-4.13.3.tgz",

"readme": "ERROR: No README data found!"

}

**Package.json 属性说明**

* **name** - 包名。
* **version** - 包的版本号。
* **description** - 包的描述。
* **homepage** - 包的官网 url 。
* **author** - 包的作者姓名。
* **contributors** - 包的其他贡献者姓名。
* **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node\_module 目录下。
* **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
* **main** - main 字段是一个模块ID，它是一个指向你程序的主要项目。就是说，如果你包的名字叫 express，然后用户安装它，然后require("express")。
* **keywords** - 关键字

**卸载模块**

我们可以使用以下命令来卸载 Node.js 模块。

$ npm uninstall express

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

$ npm ls

**更新模块**

我们可以使用以下命令更新模块：

$ npm update express

**搜索模块**

使用以下来搜索模块：

$ npm search express

**创建模块**

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

$ npm init

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields

and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and

save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (node\_modules) runoob # 模块名

version: (1.0.0)

description: Node.js 测试模块(www.runoob.com) # 描述

entry point: (index.js)

test command: make test

git repository: https://github.com/runoob/runoob.git # Github 地址

keywords:

author:

license: (ISC)

About to write to ……/node\_modules/package.json: # 生成地址

{

"name": "runoob",

"version": "1.0.0",

"description": "Node.js 测试模块(www.runoob.com)",

……

}

Is this ok? (yes) yes

以上的信息，你需要根据你自己的情况输入。在最后输入 "yes" 后会生成 package.json 文件。

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

$ npm adduser

Username: mcmohd

Password:

Email: (this IS public) mcmohd@gmail.com

接下来我们就用以下命令来发布模块：

$ npm publish

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

**版本号**

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

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

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

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

NPM支持的所有版本号范围指定方式可以查看[官方文档](https://npmjs.org/doc/files/package.json.html#dependencies)。

**NPM 常用命令**

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

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

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

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

**使用淘宝 NPM 镜像**

大家都知道国内直接使用 npm 的官方镜像是非常慢的，这里推荐使用淘宝 NPM 镜像。

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

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

$ npm install -g cnpm --registry=https://registry.npm.taobao.org

这样就可以使用 cnpm 命令来安装模块了：

$ cnpm install [name]

更多信息可以查阅：<http://npm.taobao.org/>。

**Node.js REPL(交互式解释器)**

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

Node 自带了交互式解释器，可以执行以下任务：

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

Node 的交互式解释器可以很好的调试 Javascript 代码。

开始学习 REPL

我们可以输入以下命令来启动 Node 的终端：

$ node

>

这时我们就可以在 > 后输入简单的表达式，并按下回车键来计算结果。

**简单的表达式运算**

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

$ node

> 1 +4

5

> 5 / 2

2.5

> 3 \* 6

18

> 4 - 1

3

> 1 + ( 2 \* 3 ) - 4

3

>

**使用变量**

你可以将数据存储在变量中，并在你需要的时候使用它。

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

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

$ node

> x = 10

10

> var y = 10

undefined

> x + y

20

> console.log("Hello World")

Hello World

undefined

> console.log("www.runoob.com")

www.runoob.com

undefined

**多行表达式**

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

$ node

> var x = 0

undefined

> do {

... x++;

... console.log("x: " + x);

... } while ( x < 5 );

x: 1

x: 2

x: 3

x: 4

x: 5

undefined

>

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

**下划线(\_)变量**

你可以使用下划线(\_)获取最后一次表达式的运算结果（类似python）：

$ node

> var x = 10

undefined

> var y = 20

undefined

> x + y

30

> var sum = \_

undefined

> console.log(sum)

30

undefined

>

**REPL 命令**

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

**停止 REPL**

前面我们已经提到按下两次 **ctrl + c** 键就能退出 REPL:

$ node

>

(^C again to quit)

>

**Node.js 回调函数**

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

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

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

**阻塞代码实例**

创建一个文件 input.txt ，内容如下：

菜鸟教程官网地址：www.runoob.com

创建 main.js 文件, 代码如下：

var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());

console.log("程序执行结束!");

以上代码执行结果如下：

$ node main.js

菜鸟教程官网地址：www.runoob.com

程序执行结束!

**非阻塞代码实例**

创建一个文件 input.txt ，内容如下：

菜鸟教程官网地址：www.runoob.com

创建 main.js 文件, 代码如下：

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err) return console.error(err);

console.log(data.toString());

});

console.log("程序执行结束!");

以上代码执行结果如下：

$ node main.js

程序执行结束!

菜鸟教程官网地址：www.runoob.com

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

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

**Node.js 事件循环**

Node.js 是单进程单线程应用程序，但是通过事件和回调支持并发，所以性能非常高。

Node.js 的每一个 API 都是异步的，并作为一个独立线程运行，使用异步函数调用，并处理并发。

Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

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

**事件驱动程序**

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

当这个请求完成，它被放回**处理队列**，当到达队列开头，这个结果被返回给用户。

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

在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。



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

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

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

以下程序绑定事件处理程序：

// 绑定事件及事件的处理程序

eventEmitter.on('eventName', eventHandler);

我们可以通过程序触发事件：

// 触发事件

eventEmitter.emit('eventName');

**实例**

创建 main.js 文件，代码如下所示：

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

// 创建事件处理程序

var connectHandler = function connected() {

console.log('连接成功。');

// 触发 data\_received 事件

eventEmitter.emit('data\_received');

}

// 绑定 connection 事件处理程序

eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data\_received 事件

eventEmitter.on('data\_received', function(){

console.log('数据接收成功。');

});

// 触发 connection 事件

eventEmitter.emit('connection');

console.log("程序执行完毕。");

接下来让我们执行以上代码：

$ node main.js

连接成功。

数据接收成功。

程序执行完毕。

**Node 应用程序是如何工作的？**

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

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

菜鸟教程官网地址：www.runoob.com

创建 main.js 文件，代码如下：

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err){

console.log(err.stack);

return;

}

console.log(data.toString());

});

console.log("程序执行完毕");

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

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

执行以上代码，执行结果如下：

程序执行完毕

菜鸟教程官网地址：www.runoob.com

接下来我们删除 input.txt 文件，执行结果如下所示：

程序执行完毕

Error: ENOENT, open 'input.txt'

因为文件 input.txt 不存在，所以输出了错误信息。

**Node.js EventEmitter**

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

Node.js里面的许多对象都会分发事件：一个net.Server对象会在每次有新连接时分发一个事件， 一个fs.readStream对象会在文件被打开的时候发出一个事件。 所有这些产生事件的对象都是 events.EventEmitter 的实例。

**EventEmitter 类**

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

你可以通过require("events");来访问该模块。

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

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

下面我们用一个简单的例子说明 EventEmitter 的用法：

//event.js 文件

var EventEmitter = require('events').EventEmitter;

var event = new EventEmitter();

event.on('some\_event', function() {

console.log('some\_event 事件触发');

});

setTimeout(function() {

event.emit('some\_event');

}, 1000);

执行结果如下：

运行这段代码，1 秒后控制台输出了 **'some\_event 事件触发'**。其原理是 event 对象注册了事件 some\_event 的一个监听器（通过on方法），然后我们通过 setTimeout 在 1000 毫秒以后向 event 对象发送事件 some\_event（通过emit方法），此时会调用some\_event 的监听器。

$ node event.js

some\_event 事件触发

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

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

让我们以下面的例子解释这个过程：

//event.js 文件

var events = require('events');

var emitter = new events.EventEmitter();

emitter.on('someEvent', function(arg1, arg2) {

console.log('listener1', arg1, arg2);

});

emitter.on('someEvent', function(arg1, arg2) {

console.log('listener2', arg1, arg2);

});

emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');

执行以上代码，运行的结果如下：

$ node event.js

listener1 arg1 参数 arg2 参数

listener2 arg1 参数 arg2 参数

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

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

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

**方法**

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

**类方法**

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **listenerCount(emitter, event)** 返回由指定EventEmitter对象创建的指定事件的监听器数量。 |

**事件**

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

**实例**

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

创建 main.js 文件，代码如下：

var events = require('events');

var eventEmitter = new events.EventEmitter();

// 监听器 #1

var listener1 = function listener1() {

console.log('监听器 listener1 执行。');

}

// 监听器 #2

var listener2 = function listener2() {

console.log('监听器 listener2 执行。');

}

// 绑定 connection 事件，处理函数为 listener1

eventEmitter.addListener('connection', listener1);

// 绑定 connection 事件，处理函数为 listener2

eventEmitter.on('connection', listener2);

var eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');

console.log(eventListeners + " 个监听器监听连接事件。");

// 处理 connection 事件

eventEmitter.emit('connection');

// 移除监绑定的 listener1 函数

eventEmitter.removeListener('connection', listener1);

console.log("listener1 不再受监听。");

// 触发连接事件

eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');

console.log(eventListeners + " 个监听器监听连接事件。");

console.log("程序执行完毕。");

以上代码，执行结果如下所示：

$ node main.js

2 个监听器监听连接事件。

监听器 listener1 执行。

监听器 listener2 执行。

listener1 不再受监听。

监听器 listener2 执行。

1 个监听器监听连接事件。

程序执行完毕。

**error 事件**

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

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

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

var events = require('events');

var emitter = new events.EventEmitter();

emitter.emit('error');

运行时会显示以下错误：

node.js:201

throw e; // process.nextTick error, or 'error' event on first tick

^

Error: Uncaught, unspecified 'error' event.

at EventEmitter.emit (events.js:50:15)

at Object.<anonymous> (/home/byvoid/error.js:5:9)

at Module.\_compile (module.js:441:26)

at Object..js (module.js:459:10)

at Module.load (module.js:348:31)

at Function.\_load (module.js:308:12)

at Array.0 (module.js:479:10)

at EventEmitter.\_tickCallback (node.js:192:40)

**继承 EventEmitter**

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、 http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。

为什么要这样做呢？原因有两点：

首先，具有某个实体功能的对象实现事件符合语义， 事件的监听和发射应该是一个对象的方法。

其次 JavaScript 的对象机制是基于原型的，支持 部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

**Node.js Buffer(缓冲区)**

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

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

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

**创建 Buffer 类**

Node Buffer 类可以通过多种方式来创建。

**方法 1**

创建长度为 10 字节的 Buffer 实例：

var buf = new Buffer(10);

**方法 2**

通过给定的数组创建 Buffer 实例：

var buf = new Buffer([10, 20, 30, 40, 50]);

**方法 3**

通过一个字符串来创建 Buffer 实例：

var buf = new Buffer("www.runoob.com", "utf-8");

utf-8 是默认的编码方式，此外它同样支持以下编码："ascii", "utf8", "utf16le", "ucs2", "base64" 和 "hex"。

**写入缓冲区**

**语法**

写入 Node 缓冲区的语法如下所示：

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

**参数**

参数描述如下：

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

**返回值**

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

**实例**

buf = new Buffer(256);

len = buf.write("www.runoob.com");

console.log("写入字节数 : "+ len);

执行以上代码，输出结果为：

$node main.js

写入字节数 : 14

**从缓冲区读取数据**

**语法**

读取 Node 缓冲区数据的语法如下所示：

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

**参数**

参数描述如下：

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

**返回值**

解码缓冲区数据并使用指定的编码返回字符串。

**实例**

buf = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {

buf[i] = i + 97;

}

console.log( buf.toString('ascii')); // 输出: abcdefghijklmnopqrstuvwxyz

console.log( buf.toString('ascii',0,5)); // 输出: abcde

console.log( buf.toString('utf8',0,5)); // 输出: abcde

console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde

执行以上代码，输出结果为：

$ node main.js

abcdefghijklmnopqrstuvwxyz

abcde

abcde

abcde

**将 Buffer 转换为 JSON 对象**

**语法**

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

buf.toJSON()

**返回值**

返回 JSON 对象。

**实例**

var buf = new Buffer('www.runoob.com');

var json = buf.toJSON();

console.log(json);

执行以上代码，输出结果为：

[ 119, 119, 119, 46, 114, 117, 110, 111, 111, 98, 46, 99, 111, 109 ]

**缓冲区合并**

**语法**

Node 缓冲区合并的语法如下所示：

Buffer.concat(list[, totalLength])

**参数**

参数描述如下：

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

**返回值**

返回一个多个成员合并的新 Buffer 对象。

**实例**

var buffer1 = new Buffer('菜鸟教程 ');

var buffer2 = new Buffer('www.runoob.com');

var buffer3 = Buffer.concat([buffer1,buffer2]);

console.log("buffer3 内容: " + buffer3.toString());

执行以上代码，输出结果为：

buffer3 内容: 菜鸟教程 www.runoob.com

**缓冲区比较**

**语法**

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

buf.compare(otherBuffer);

**参数**

参数描述如下：

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

**返回值**

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

**实例**

var buffer1 = new Buffer('ABC');

var buffer2 = new Buffer('ABCD');

var result = buffer1.compare(buffer2);

if(result < 0) {

console.log(buffer1 + " 在 " + buffer2 + "之前");

}else if(result == 0){

console.log(buffer1 + " 与 " + buffer2 + "相同");

}else {

console.log(buffer1 + " 在 " + buffer2 + "之后");

}

执行以上代码，输出结果为：

ABC在ABCD之前

**拷贝缓冲区**

**语法**

Node 缓冲区拷贝语法如下所示：

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

**参数**

参数描述如下：

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

**返回值**

没有返回值。

**实例**

var buffer1 = new Buffer('ABC');

// 拷贝一个缓冲区

var buffer2 = new Buffer(3);

buffer1.copy(buffer2);

console.log("buffer2 content: " + buffer2.toString());

执行以上代码，输出结果为：

buffer2 content: ABC

**缓冲区裁剪**

Node 缓冲区裁剪语法如下所示：

buf.slice([start[, end]])

**参数**

参数描述如下：

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

**返回值**

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

**实例**

var buffer1 = new Buffer('runoob');

// 剪切缓冲区

var buffer2 = buffer1.slice(0,2);

console.log("buffer2 content: " + buffer2.toString());

执行以上代码，输出结果为：

buffer2 content: ru

**缓冲区长度**

**语法**

Node 缓冲区长度计算语法如下所示：

buf.length;

**返回值**

返回 Buffer 对象所占据的内存长度。

**实例**

var buffer = new Buffer('www.runoob.com');

// 缓冲区长度

console.log("buffer length: " + buffer.length);

执行以上代码，输出结果为：

buffer length: 14

**方法参考手册**

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

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **new Buffer(size)**  分配一个新的 size 大小单位为8位字节的 buffer。 注意, size 必须小于 kMaxLength，否则，将会抛出异常 RangeError。 |
| 2 | **new Buffer(buffer)**  拷贝参数 buffer 的数据到 Buffer 实例。 |
| 3 | **new Buffer(str[, encoding])** 分配一个新的 buffer ，其中包含着传入的 str 字符串。 encoding 编码方式默认为 'utf8'。 |
| 4 | **buf.length** 返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容的大小。length 是 buffer 对象所分配的内存数，它不会随着这个 buffer 对象内容的改变而改变。 |
| 5 | **buf.write(string[, offset[, length]][, encoding])** 根据参数 offset 偏移量和指定的 encoding 编码方式，将参数 string 数据写入buffer。 offset 偏移量默认值是 0, encoding 编码方式默认是 utf8。 length 长度是将要写入的字符串的 bytes 大小。 返回 number 类型，表示写入了多少 8 位字节流。如果 buffer 没有足够的空间来放整个 string，它将只会只写入部分字符串。 length 默认是 buffer.length - offset。 这个方法不会出现写入部分字符。 |
| 6 | **buf.writeUIntLE(value, offset, byteLength[, noAssert])** 将 value 写入到 buffer 里， 它由 offset 和 byteLength 决定，最高支持 48 位无符号整数，小端对齐，例如：  var b = new Buffer(6);  b.writeUIntBE(0x1234567890ab, 0, 6);  // <Buffer 12 34 56 78 90 ab>  noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。 |
| 7 | **buf.writeUIntBE(value, offset, byteLength[, noAssert])** 将 value 写入到 buffer 里， 它由 offset 和 byteLength 决定，最高支持 48 位无符号整数，大端对齐。noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。 |
| 8 | **buf.writeIntLE(value, offset, byteLength[, noAssert])** 将value 写入到 buffer 里， 它由offset 和 byteLength 决定，最高支持48位有符号整数，小端对齐。noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。 |
| 9 | **buf.writeIntBE(value, offset, byteLength[, noAssert])** 将value 写入到 buffer 里， 它由offset 和 byteLength 决定，最高支持48位有符号整数，大端对齐。noAssert 值为 true 时，不再验证 value 和 offset 的有效性。 默认是 false。 |
| 10 | **buf.readUIntLE(offset, byteLength[, noAssert])** 支持读取 48 位以下的无符号数字，小端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。 |
| 11 | **buf.readUIntBE(offset, byteLength[, noAssert])** 支持读取 48 位以下的无符号数字，大端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。 |
| 12 | **buf.readIntLE(offset, byteLength[, noAssert])** 支持读取 48 位以下的有符号数字，小端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。 |
| 13 | **buf.readIntBE(offset, byteLength[, noAssert])** 支持读取 48 位以下的有符号数字，大端对齐。noAssert 值为 true 时， offset 不再验证是否超过 buffer 的长度，默认为 false。 |
| 14 | **buf.toString([encoding[, start[, end]]])** 根据 encoding 参数（默认是 'utf8'）返回一个解码过的 string 类型。还会根据传入的参数 start (默认是 0) 和 end (默认是 buffer.length)作为取值范围。 |
| 15 | **buf.toJSON()** 将 Buffer 实例转换为 JSON 对象。 |
| 16 | **buf[index]** 获取或设置指定的字节。返回值代表一个字节，所以返回值的合法范围是十六进制0x00到0xFF 或者十进制0至 255。 |
| 17 | **buf.equals(otherBuffer)** 比较两个缓冲区是否相等，如果是返回 true，否则返回 false。 |
| 18 | **buf.compare(otherBuffer)** 比较两个 Buffer 对象，返回一个数字，表示 buf 在 otherBuffer 之前，之后或相同。 |
| 19 | **buf.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]])** buffer 拷贝，源和目标可以相同。 targetStart 目标开始偏移和 sourceStart 源开始偏移默认都是 0。 sourceEnd 源结束位置偏移默认是源的长度 buffer.length 。 |
| 20 | **buf.slice([start[, end]])** 剪切 Buffer 对象，根据 start(默认是 0 ) 和 end (默认是 buffer.length ) 偏移和裁剪了索引。 负的索引是从 buffer 尾部开始计算的。 |
| 21 | **buf.readUInt8(offset[, noAssert])** 根据指定的偏移量，读取一个无符号 8 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 如果这样 offset 可能会超出buffer 的末尾。默认是 false。 |
| 22 | **buf.readUInt16LE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 字节序格式读取一个无符号 16 位整数。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。 |
| 23 | **buf.readUInt16BE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 字节序格式读取一个无符号 16 位整数，大端对齐。若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。 |
| 24 | **buf.readUInt32LE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个无符号 32 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。 |
| 25 | **buf.readUInt32BE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个无符号 32 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。 |
| 26 | **buf.readInt8(offset[, noAssert])** 根据指定的偏移量，读取一个有符号 8 位整数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。 |
| 27 | **buf.readInt16LE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 格式读取一个 有符号 16 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。 |
| 28 | **buf.readInt16BE(offset[, noAssert])** 根据指定的偏移量，使用特殊的 endian 格式读取一个 有符号 16 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默认是 false。 |
| 29 | **buf.readInt32LE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。 |
| 30 | **buf.readInt32BE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个有符号 32 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。 |
| 31 | **buf.readFloatLE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位双浮点数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer的末尾。默认是 false。 |
| 32 | **buf.readFloatBE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian 字节序格式读取一个 32 位双浮点数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer的末尾。默认是 false。 |
| 33 | **buf.readDoubleLE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian字节序格式读取一个 64 位双精度数，小端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。 |
| 34 | **buf.readDoubleBE(offset[, noAssert])** 根据指定的偏移量，使用指定的 endian字节序格式读取一个 64 位双精度数，大端对齐。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能会超出buffer 的末尾。默认是 false。 |
| 35 | **buf.writeUInt8(value, offset[, noAssert])** 根据传入的 offset 偏移量将 value 写入 buffer。注意：value 必须是一个合法的无符号 8 位整数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则不要使用。默认是 false。 |
| 36 | **buf.writeUInt16LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的无符号 16 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 37 | **buf.writeUInt16BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的无符号 16 位整数，大端对齐。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 38 | **buf.writeUInt32LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式(LITTLE-ENDIAN:小字节序)将 value 写入buffer。注意：value 必须是一个合法的无符号 32 位整数，小端对齐。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 39 | **buf.writeUInt32BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式(Big-Endian:大字节序)将 value 写入buffer。注意：value 必须是一个合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者offset可能会超出buffer的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 40 | **buf.writeInt8(value, offset[, noAssert])**<="" td=""> |
| 41 | **buf.writeInt16LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。 |
| 42 | **buf.writeInt16BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false 。 |
| 43 | **buf.writeInt32LE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 44 | **buf.writeInt32BE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 45 | **buf.writeFloatLE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer 。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 46 | **buf.writeFloatBE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer 。注意：当 value 不是一个 32 位浮点数类型的值时，结果将是不确定的。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value可能过大，或者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 47 | **buf.writeDoubleLE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个有效的 64 位double 类型的值。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成value被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 48 | **buf.writeDoubleBE(value, offset[, noAssert])** 根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 buffer。注意：value 必须是一个有效的 64 位double 类型的值。 若参数 noAssert 为 true 将不会验证 value 和 offset 偏移量参数。 这意味着 value 可能过大，或者 offset 可能会超出 buffer 的末尾从而造成value被丢弃。 除非你对这个参数非常有把握，否则尽量不要使用。默认是 false。 |
| 49 | **buf.fill(value[, offset][, end])** 使用指定的 value 来填充这个 buffer。如果没有指定 offset (默认是 0) 并且 end (默认是 buffer.length) ，将会填充整个buffer。 |