# 一、简介

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。

# 二、Node.js 安装配置

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

# Node.js 创建第一个应用

|  |
| --- |
| *Server.js* |
| *//1、创建第一个应用 //步骤一、引入 required 模块* **var *http***=require(**"http"**); *//步骤二、创建服务器* ***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(**"Hellow World\n"**);  } ).listen(8888);  *//终端打印如下信息* ***console***.log(**"Server runing at http://127.0.0.1:8888"**); |
| 使用 **node** 命令执行以上的代码：  IMG_256 |
| 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。  IMG_256 |

# 四、NPM 使用介绍

## 1、使用 npm 命令安装模块

使用 npm 命令安装常用的 Node.js web框架模块:

|  |
| --- |
| *$ npm install express* |
| 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,例如：

|  |
| --- |
| *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* |

### 2.1 本地安装

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

### **2.2 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。

**2.3 查看全局安装的模块**

|  |
| --- |
| 使用以下命令来查看所有全局安装的模块：  *D:\Program Files\nodejs>npm list -g C:\Users\lijua\AppData\Roaming\npm `-- express@4.17.1* |
| 如果要查看某个模块的版本号，可以使用命令如下：  *D:\Program Files\nodejs>npm list express nodejs@ D:\Program Files\nodejs `-- express@4.17.1* |

## **卸载、更新、搜索模块**

|  |
| --- |
| 卸载模块：*$ 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：*$ npm update express* |
| 搜索模块:*$ npm search express* |
|  |

## **创建模板**

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 initThis 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 fieldsand 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 |
| 接下来我们就用以下命令来发布模块：  $ npm publish |

## **使用 package.json**

package.json 位于模块的目录下，用于定义包的属性。可参看下 express 包的 package.json 文件，位于 node\_modules/express/package.json  
Package.json 属性说明：

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

## **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。
* 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/](http://npm.taobao.org/" \t "https://www.runoob.com/nodejs/_blank)。 |

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

Node 的交互式解释器可以很好的调试 Javascript 代码。

## 1、开始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。 |
| **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。 |
| **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：    **...** 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(\_)获取上一个表达式的运算结果：  > x + y  30  > var sum = \_ |

## 2、REPL 命令

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

## 3、停止 REPL

前面我们已经提到按下两次 **ctrl + c** 键就能退出 REPL。

# 六、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。

## 1、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，第二个是回调函数名称。  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 函数用于改变监听器的默认限制的数量。 |
| 7 | **listeners(event)** 返回指定事件的监听器数组。 |
| 8 | **emit(event, [arg1], [arg2], [...])** 按监听器的顺序执行执行每个监听器，如果事件有注册监听返回 true，否则返回 false。 |

## **2、类方法**

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

events.EventEmitter.listenerCount(emitter, eventName) //已废弃，不推荐

events.emitter.listenerCount(eventName) //推荐

## **3、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 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)

## **4、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。

# 七、Node.js 全局对象

## **1、Node.js 全局对象**

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

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

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

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

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

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

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

## **2、proces全局变量**

process 是一个全局变量，即 global 对象的属性。

它用于描述当前Node.js 进程状态的对象，提供了一个与操作系统的简单接口。

### 2.1 process最常用的成员方法

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

### 2.2退出状态码

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

### 2.3 Process 属性

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

### 2.4 **方法参考手册**

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

# 八、Node.js 文件系统

## 1、打开文件：flags - 文件打开的行为参数

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

## 获取文件信息：stats类中的方法

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

## 文件模块方法参考手册

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **fs.rename(oldPath, newPath, callback)** 异步 rename().回调函数没有参数，但可能抛出异常。 |
| 2 | **fs.ftruncate(fd, len, callback)** 异步 ftruncate().回调函数没有参数，但可能抛出异常。 |
| 3 | **fs.ftruncateSync(fd, len)** 同步 ftruncate() |
| 4 | **fs.truncate(path, len, callback)** 异步 truncate().回调函数没有参数，但可能抛出异常。 |
| 5 | **fs.truncateSync(path, len)** 同步 truncate() |
| 6 | **fs.chown(path, uid, gid, callback)** 异步 chown().回调函数没有参数，但可能抛出异常。 |
| 7 | **fs.chownSync(path, uid, gid)** 同步 chown() |
| 8 | **fs.fchown(fd, uid, gid, callback)** 异步 fchown().回调函数没有参数，但可能抛出异常。 |
| 9 | **fs.fchownSync(fd, uid, gid)** 同步 fchown() |
| 10 | **fs.lchown(path, uid, gid, callback)** 异步 lchown().回调函数没有参数，但可能抛出异常。 |
| 11 | **fs.lchownSync(path, uid, gid)** 同步 lchown() |
| 12 | **fs.chmod(path, mode, callback)** 异步 chmod().回调函数没有参数，但可能抛出异常。 |
| 13 | **fs.chmodSync(path, mode)** 同步 chmod(). |
| 14 | **fs.fchmod(fd, mode, callback)** 异步 fchmod().回调函数没有参数，但可能抛出异常。 |
| 15 | **fs.fchmodSync(fd, mode)** 同步 fchmod(). |
| 16 | **fs.lchmod(path, mode, callback)** 异步 lchmod().回调函数没有参数，但可能抛出异常。Only available on Mac OS X. |
| 17 | **fs.lchmodSync(path, mode)** 同步 lchmod(). |
| 18 | **fs.stat(path, callback)** 异步 stat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。 |
| 19 | **fs.lstat(path, callback)** 异步 lstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。 |
| 20 | **fs.fstat(fd, callback)** 异步 fstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。 |
| 21 | **fs.statSync(path)** 同步 stat(). 返回 fs.Stats 的实例。 |
| 22 | **fs.lstatSync(path)** 同步 lstat(). 返回 fs.Stats 的实例。 |
| 23 | **fs.fstatSync(fd)** 同步 fstat(). 返回 fs.Stats 的实例。 |
| 24 | **fs.link(srcpath, dstpath, callback)** 异步 link().回调函数没有参数，但可能抛出异常。 |
| 25 | **fs.linkSync(srcpath, dstpath)** 同步 link(). |
| 26 | **fs.symlink(srcpath, dstpath[, type], callback)** 异步 symlink().回调函数没有参数，但可能抛出异常。 type 参数可以设置为 'dir', 'file', 或 'junction' (默认为 'file') 。 |
| 27 | **fs.symlinkSync(srcpath, dstpath[, type])** 同步 symlink(). |
| 28 | **fs.readlink(path, callback)** 异步 readlink(). 回调函数有两个参数 err, linkString。 |
| 29 | **fs.realpath(path[, cache], callback)** 异步 realpath(). 回调函数有两个参数 err, resolvedPath。 |
| 30 | **fs.realpathSync(path[, cache])** 同步 realpath()。返回绝对路径。 |
| 31 | **fs.unlink(path, callback)** 异步 unlink().回调函数没有参数，但可能抛出异常。 |
| 32 | **fs.unlinkSync(path)** 同步 unlink(). |
| 33 | **fs.rmdir(path, callback)** 异步 rmdir().回调函数没有参数，但可能抛出异常。 |
| 34 | **fs.rmdirSync(path)** 同步 rmdir(). |
| 35 | **fs.mkdir(path[, mode], callback)** S异步 mkdir(2).回调函数没有参数，但可能抛出异常。 访问权限默认为 0777。 |
| 36 | **fs.mkdirSync(path[, mode])** 同步 mkdir(). |
| 37 | **fs.readdir(path, callback)** 异步 readdir(3). 读取目录的内容。 |
| 38 | **fs.readdirSync(path)** 同步 readdir().返回文件数组列表。 |
| 39 | **fs.close(fd, callback)** 异步 close().回调函数没有参数，但可能抛出异常。 |
| 40 | **fs.closeSync(fd)** 同步 close(). |
| 41 | **fs.open(path, flags[, mode], callback)** 异步打开文件。 |
| 42 | **fs.openSync(path, flags[, mode])** 同步 version of fs.open(). |
| 43 | **fs.utimes(path, atime, mtime, callback)** |
| 44 | **fs.utimesSync(path, atime, mtime)** 修改文件时间戳，文件通过指定的文件路径。 |
| 45 | **fs.futimes(fd, atime, mtime, callback)** |
| 46 | **fs.futimesSync(fd, atime, mtime)** 修改文件时间戳，通过文件描述符指定。 |
| 47 | **fs.fsync(fd, callback)** 异步 fsync.回调函数没有参数，但可能抛出异常。 |
| 48 | **fs.fsyncSync(fd)** 同步 fsync. |
| 49 | **fs.write(fd, buffer, offset, length[, position], callback)** 将缓冲区内容写入到通过文件描述符指定的文件。 |
| 50 | **fs.write(fd, data[, position[, encoding]], callback)** 通过文件描述符 fd 写入文件内容。 |
| 51 | **fs.writeSync(fd, buffer, offset, length[, position])** 同步版的 fs.write()。 |
| 52 | **fs.writeSync(fd, data[, position[, encoding]])** 同步版的 fs.write(). |
| 53 | **fs.read(fd, buffer, offset, length, position, callback)** 通过文件描述符 fd 读取文件内容。 |
| 54 | **fs.readSync(fd, buffer, offset, length, position)** 同步版的 fs.read. |
| 55 | **fs.readFile(filename[, options], callback)** 异步读取文件内容。 |
| 56 | **fs.readFileSync(filename[, options])**<br同步版的 fs.readfile.<="" td=""> |
| 57 | **fs.writeFile(filename, data[, options], callback)** 异步写入文件内容。 |
| 58 | **fs.writeFileSync(filename, data[, options])** 同步版的 fs.writeFile。 |
| 59 | **fs.appendFile(filename, data[, options], callback)** 异步追加文件内容。 |
| 60 | **fs.appendFileSync(filename, data[, options])** The 同步 version of fs.appendFile. |
| 61 | **fs.watchFile(filename[, options], listener)** 查看文件的修改。 |
| 62 | **fs.unwatchFile(filename[, listener])** 停止查看 filename 的修改。 |
| 63 | **fs.watch(filename[, options][, listener])** 查看 filename 的修改，filename 可以是文件或目录。返回 fs.FSWatcher 对象。 |
| 64 | **fs.exists(path, callback)** 检测给定的路径是否存在。 |
| 65 | **fs.existsSync(path)** 同步版的 fs.exists. |
| 66 | **fs.access(path[, mode], callback)** 测试指定路径用户权限。 |
| 67 | **fs.accessSync(path[, mode])** 同步版的 fs.access。 |
| 68 | **fs.createReadStream(path[, options])** 返回ReadStream 对象。 |
| 69 | **fs.createWriteStream(path[, options])** 返回 WriteStream 对象。 |
| 70 | **fs.symlink(srcpath, dstpath[, type], callback)** 异步 symlink().回调函数没有参数，但可能抛出异常。 |

# Node.js GET/POST请求

# Node.js Express 框架

## 请求和响应

**request** 和 **response** 对象的具体介绍：

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

1. req.app：当callback为外部文件时，用req.app访问express的实例
2. req.baseUrl：获取路由当前安装的URL路径
3. req.body / req.cookies：获得「请求主体」/ Cookies
4. req.fresh / req.stale：判断请求是否还「新鲜」
5. req.hostname / req.ip：获取主机名和IP地址
6. req.originalUrl：获取原始请求URL
7. req.params：获取路由的parameters
8. req.path：获取请求路径
9. req.protocol：获取协议类型
10. req.query：获取URL的查询参数串
11. req.route：获取当前匹配的路由
12. req.subdomains：获取子域名
13. req.accepts()：检查可接受的请求的文档类型
14. req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages：返回指定字符集的第一个可接受字符编码
15. req.get()：获取指定的HTTP请求头
16. req.is()：判断请求头Content-Type的MIME类型

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

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

# 十一、Node.js JXcore 打包

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

JXcore 是一个支持多线程的 Node.js 发行版本

这篇文章主要是要向大家介绍 JXcore 的打包功能。

## 1、JXcore 安装

**步骤：**

1. 下载 JXcore 安装包 [https://github.com/jxcore/jxcore-release](https://github.com/jxcore/jxcore-release" \t "https://www.runoob.com/nodejs/_blank)，
2. Linux/OSX 安装命令：

$ curl https://raw.githubusercontent.com/jxcore/jxcore/master/tools/jx\_install.sh | bash

如果权限不足，可以使用以下命令：

$ curl https://raw.githubusercontent.com/jxcore/jxcore/master/tools/jx\_install.sh | sudo bash

以上步骤如果操作正确，使用以下命令，会输出版本号信息：

$ jx --version

v0.10.32

## 2、包代码

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

drwxr-xr-x 2 root root 4096 Nov 13 12:42 images-rwxr-xr-x 1 root root 30457 Mar 6 12:19 index.htm-rwxr-xr-x 1 root root 30452 Mar 1 12:54 index.js

drwxr-xr-x 23 root root 4096 Jan 15 03:48 node\_modules

drwxr-xr-x 2 root root 4096 Mar 21 06:10 scripts

drwxr-xr-x 2 root root 4096 Feb 15 11:56 style

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

$ jx package index.js index

以上命令执行成功，会生成以下两个文件：

**index.jxp** 这是一个中间件文件，包含了需要编译的完整项目信息。

**index.jx** 这是一个完整包信息的二进制文件，可运行在客户端上。

## 3、载入 JX 文件

Node.js 的项目运行：

$ node index.js command\_line\_arguments

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

$ jx index.jx command\_line\_arguments