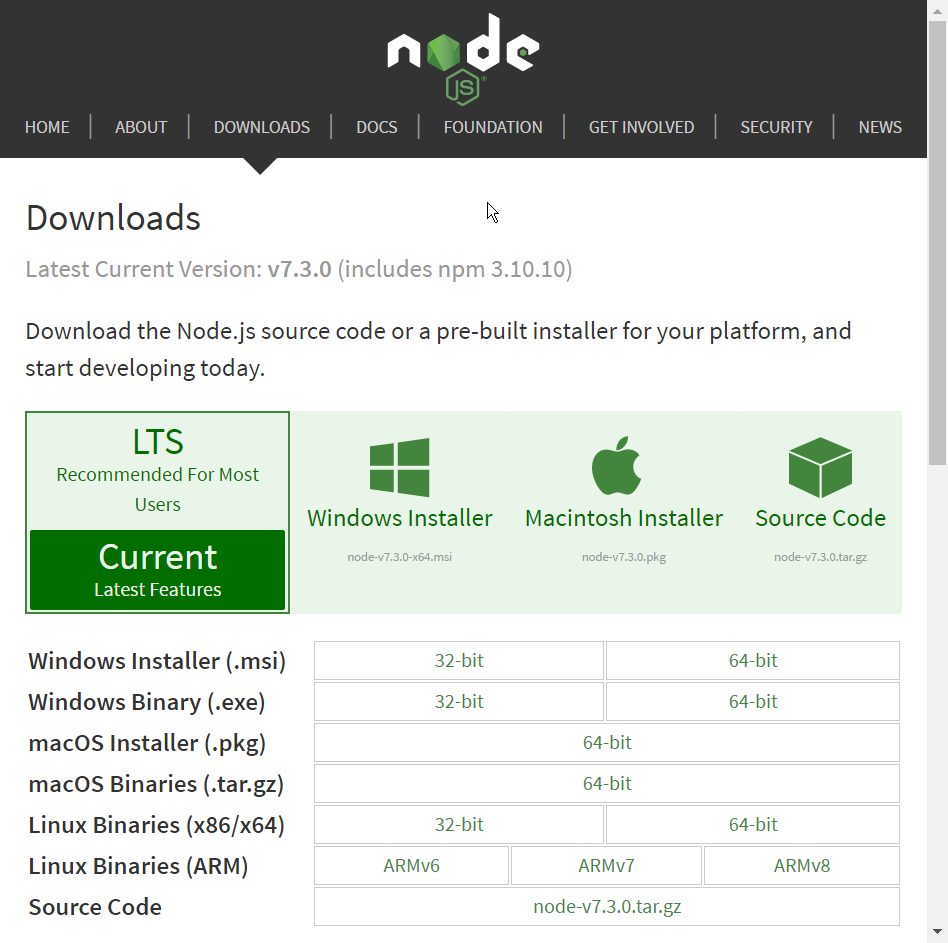
Node.js

简介:

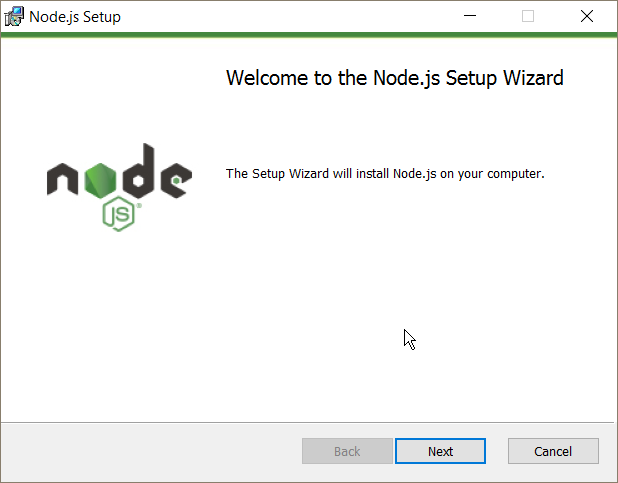
简单的来说, Node.js就是运行在服务端的JavaScript. Node.js是一个基于Chrome JavaScript运行时建立的一个平台.Node.js是一个事件驱动I/O服务端JavaScript环境, 基于Google的V8引擎, V8引擎执行JavaScript的速度非常快, 性能非常好.

# 安装配置

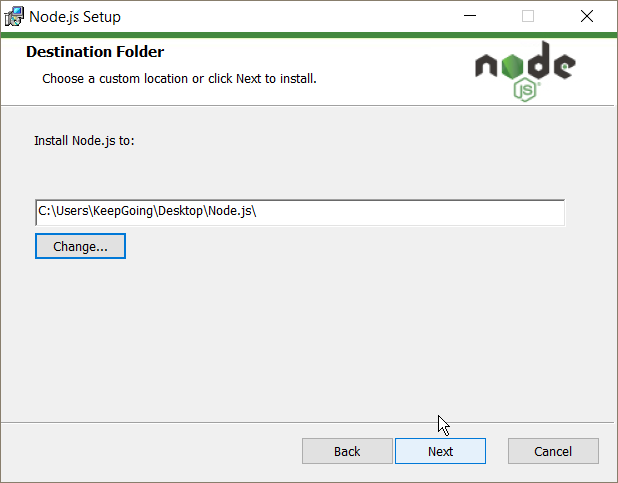
Download link: <https://nodejs.org/en/download/current/>

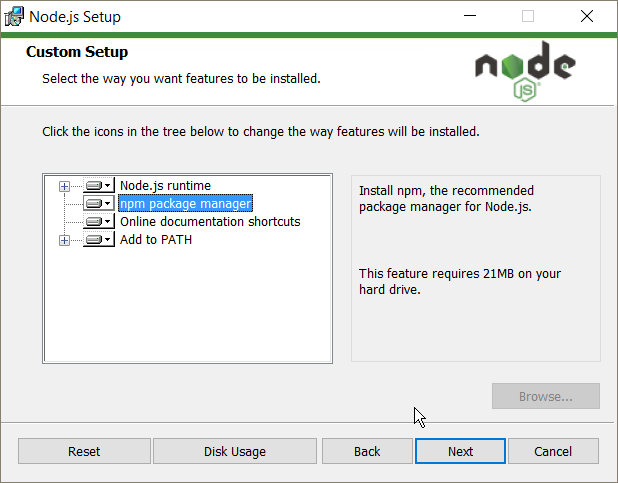


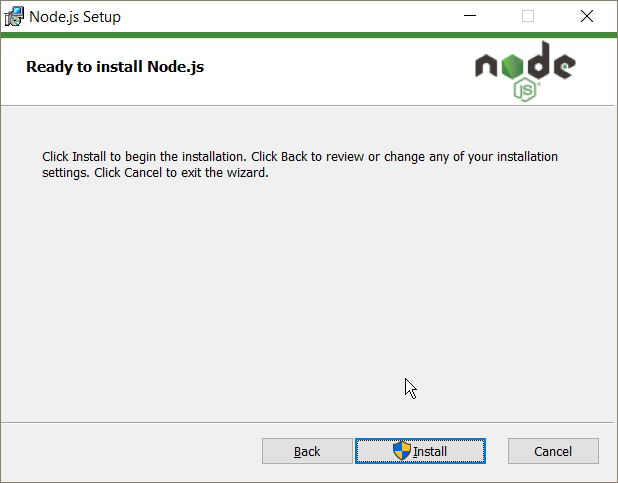
下载Windows Installer (.mis), 运行进行安装.



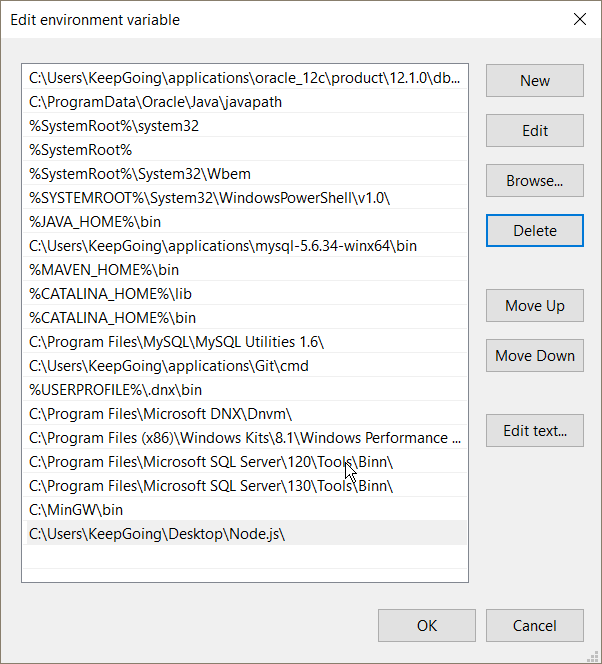




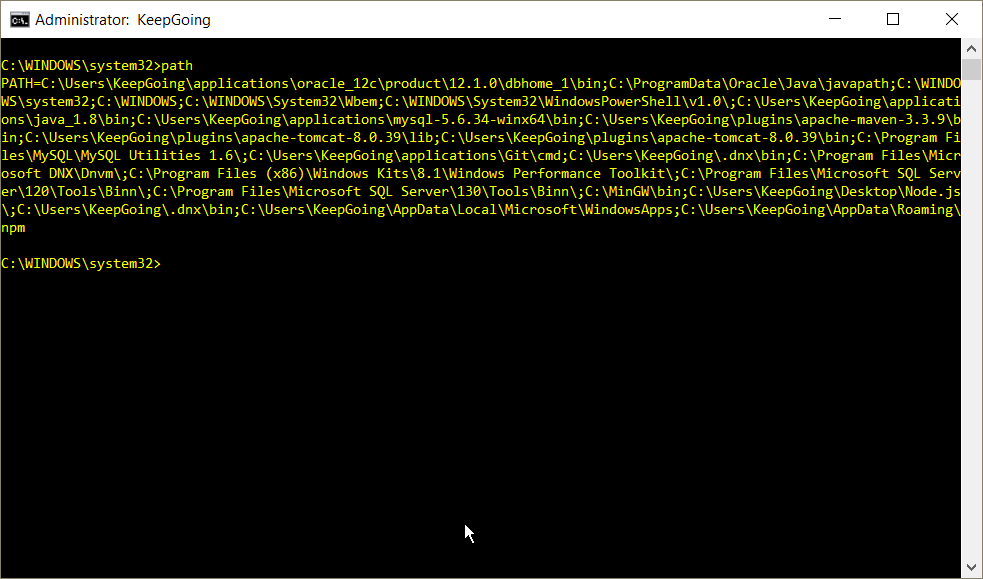




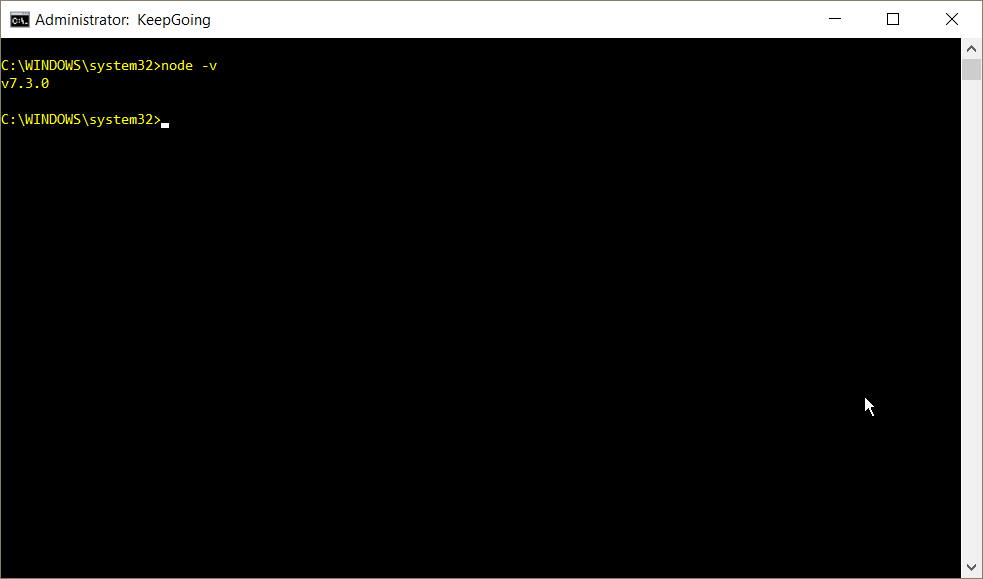
安装完成后会自动添加Node.js的PATH环境变量



或是在DOS中输入path查看



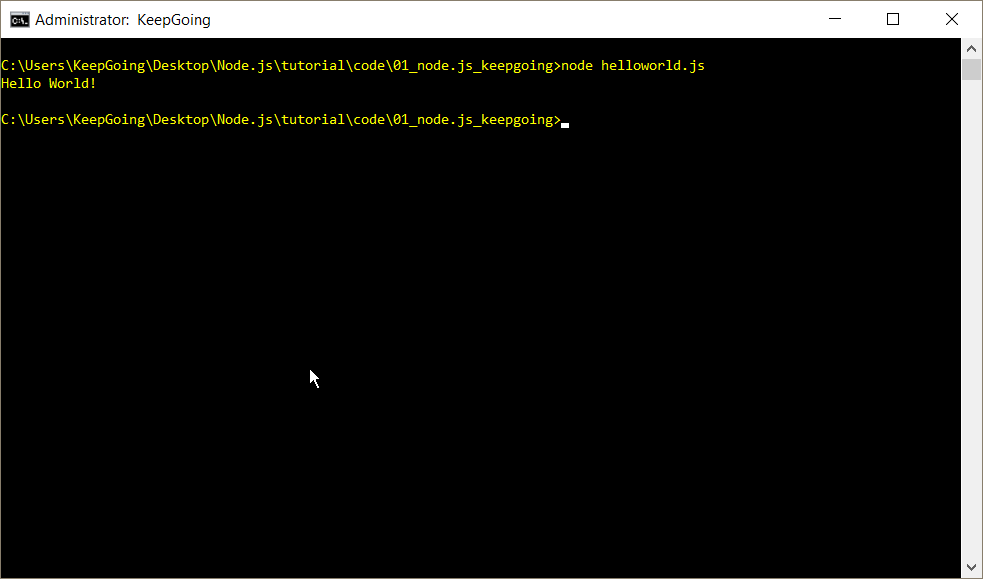
检查Node.js的版本



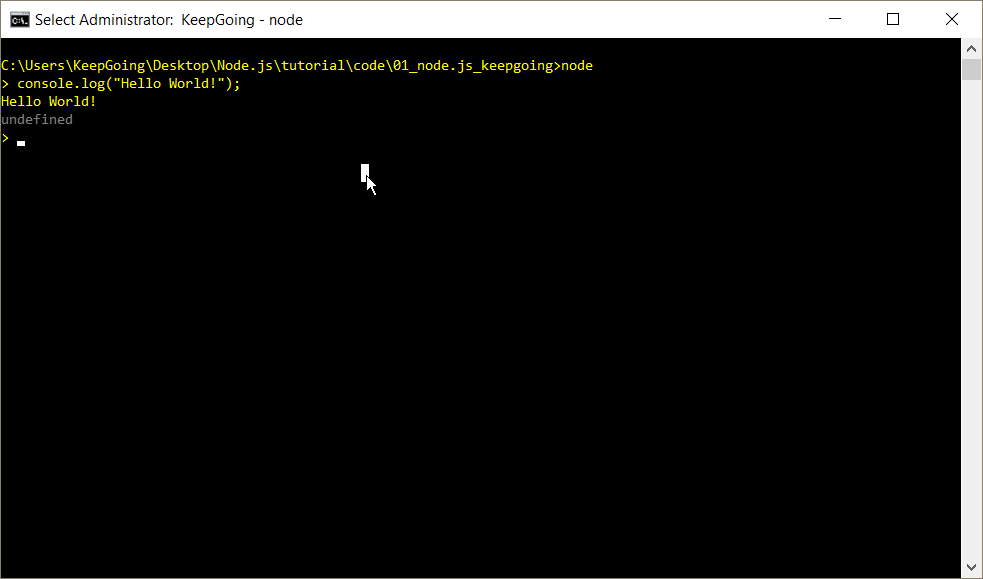
# helloworld

## 脚本模式

Console.log(“Hello World”);



## 交互模式



## Node.js创建第一个应用

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者终端发送HTTP请求, 服务器接收请求后返回响应数据.

server.js

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"

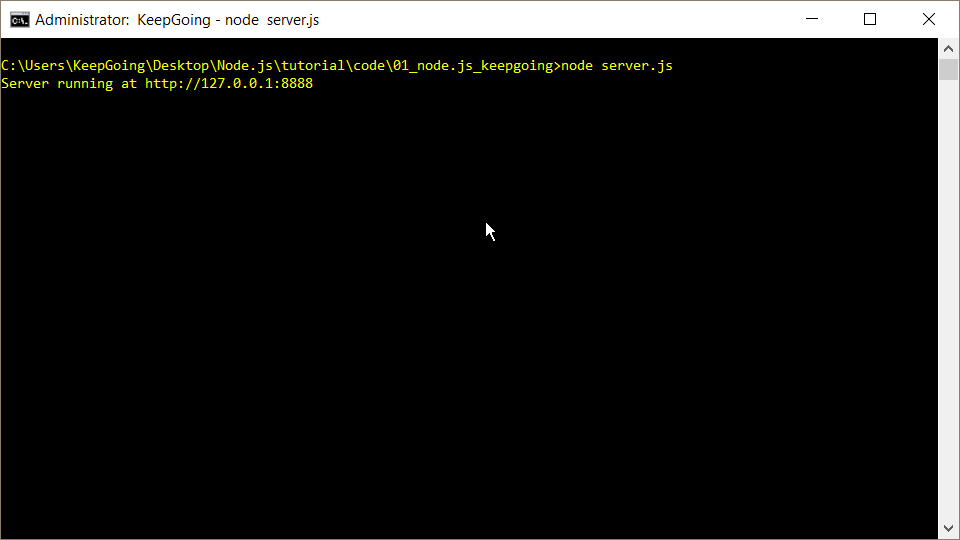
response.end('Hello World\n');

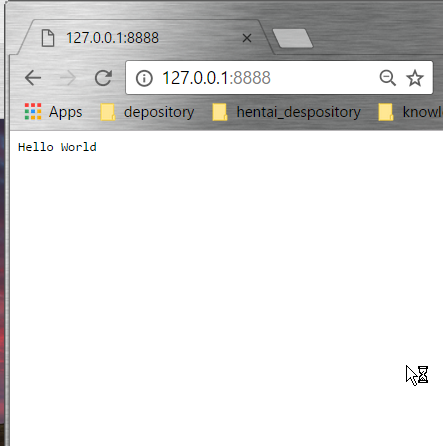
}).listen(8888);

// 终端打印如下信息

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

使用node命令执行以上的代码





分析Node.js的HTTP服务器

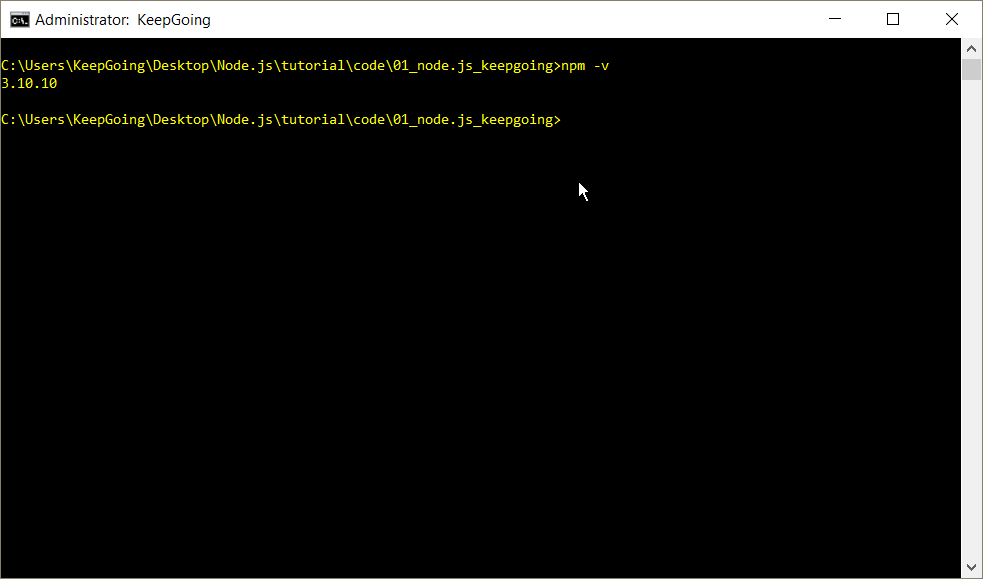
* 第一行请求(require)Node.js自带的http模块, 并且把它赋值给http变量.
* 接下来我们调用http模块提供的函数createServer. 这个函数会返回一个对象, 这个对象有一个叫做listen的方法, 这个方法有一个数值参数, 制定这个HTTP服务器监听的端口号.

# NPM

NPM是随同Node.js一起安装的包管理工具, 能解决Node.js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Node.js已经集成了NPM, 所以之前的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.



## 使用npm命令安装模块

npm install <Module Name>

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

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

* 将安装包放在当前目录的node\_modules下(运行npm命令时所在的目录), 如果没有node\_modules目录, 会自动生成.
* 可以通过require()来映入本地安装的包

全局安装

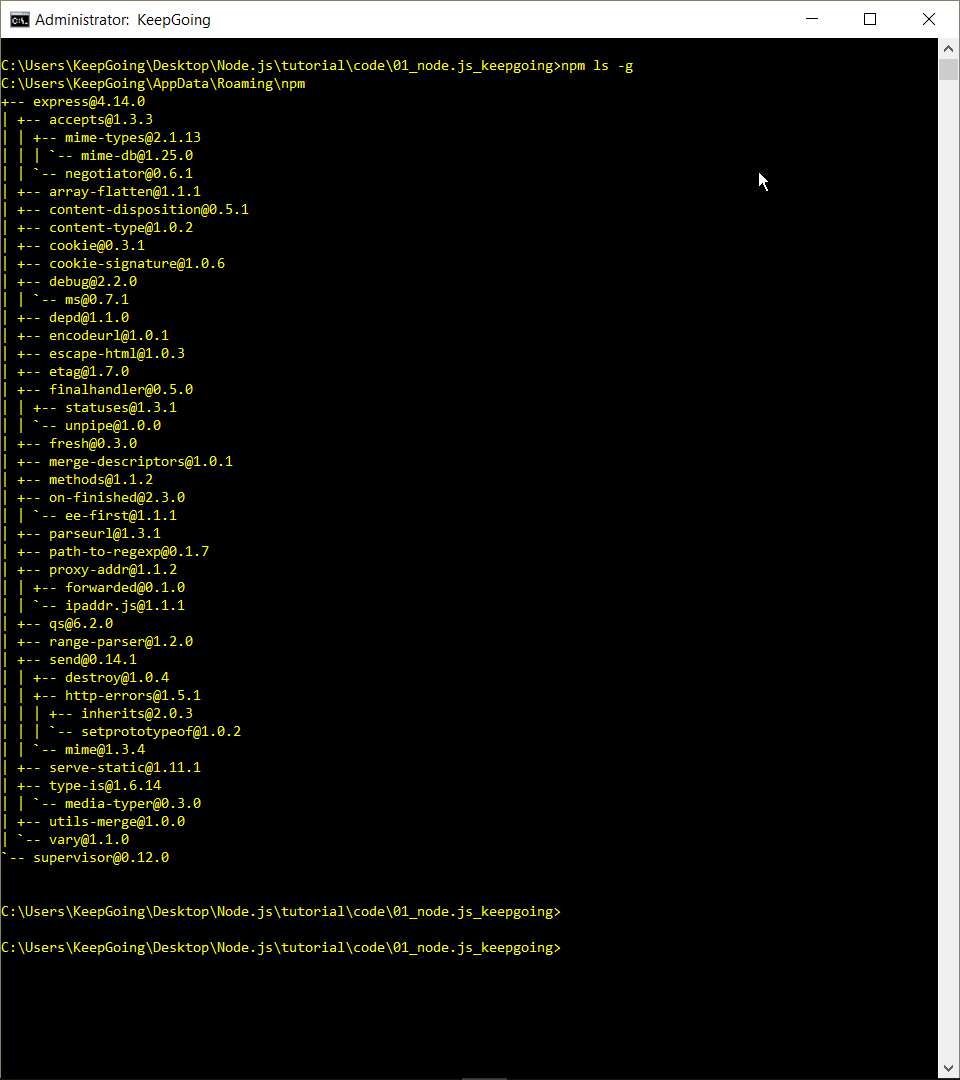
* 将安装包放在Node.js的安装目录.
* 可以在命令行直接使用.

如果你希望具备两者的功能, 则需要在两个地方安装他或使用”npm link”.

使用全局方式安装express

npm install express -g

使用”npm ls -g”查看所有全局安装的模块



## 使用package.json

package.json位于模块的目录下, 用于定义包的属性

package.json属性说明

* name: 包名.
* version: 包的版本号.
* description: 包的描述.
* homepage: 包的官方.
* author: 报的作者姓名.
* 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.
* main: main字段是一个模块ID, 它是一个指向你的程序的主要项目, 就是说, 如果你的包名叫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

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

使用以下命令搜索模块

npm search express

创建模块

创建模块, package.json文件是不可少的, 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, 你需要根据自己的情况输入. 在最后输入”yes”后会生成package.json文件.

接下在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: USERNAME

Password: PASSWORD

Email: (this IS public) Email

接下来我们就可以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, 你就可以和其他模块一样使用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, 在申明第三方依赖时, 除了可以依赖于一个固定的版本号外, 还可以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[官方文档](https://docs.npmjs.com/files/package.json#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里也有很多其他有用的字段.

除了可以在[npmjs.org/doc/](http://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.
* 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所在的目录使用npm install . -g 可先在本地安装当前命令含程序, 可用于发布前的本地测试.
* 使用npm update <pacakge>可以把当前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/>。

# REPL

Node.js REPL(Read Eval Print Loop:交互式编译器)表示一个电脑环境, 类似Window系统的终端或Unix/Linux shell, 我们可以在终端中输入命令, 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: 循环操作以上步骤知道用户两次按下Control-c按钮退出.

Node.js的交互式解释器可以很好的调试JavaScript代码.

我们可以输入以下命令来启动Node.js的终端:

node

>

这时我们就可以在>后面输入简单的表达式, 并按下回车键来计算结果.

## 简单的表达式运算

在Node.js REPL的命令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.js REPL支持输入多行表达式, 这就有点类似JavaScript. 接下再让我们执行一个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(…)符号是系统自动生成的, 你Enter换行后即可, Node.js会自动检查是否为连续的表达式.

下划线(\_)变量

可以使用下划线(\_)获取表达式的运算结果:

node

> var x = 10

undefined

> var y = 20

undefined

> x + y

30

> var sum = \_

undefined

> console.log(sum)

30

undefined

>

## REPL 命令

* Control+C: 退出当前终端.
* Control+C: 按下两次: 推出Node REPL.
* Control+D: 推出Node REPL.
* up/down: 查看输入的历史命令.
* table: 列出当前命令.
* .help: 列出使用命令.
* .break: 退出多行表达式.
* .clear: 退出多行表达是.
* .save filename: 保存当前的Node REPL会话到指定文件.
* .load filename: 载入当前Node REPL回话的文件内容.

# 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.js使用了大量的回调函数, Node.js所有API都支持回调函数.

例如, 我们可以另一边读文件, 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

Welcome to the league of Draven!

创建main01.js, 代码如下:

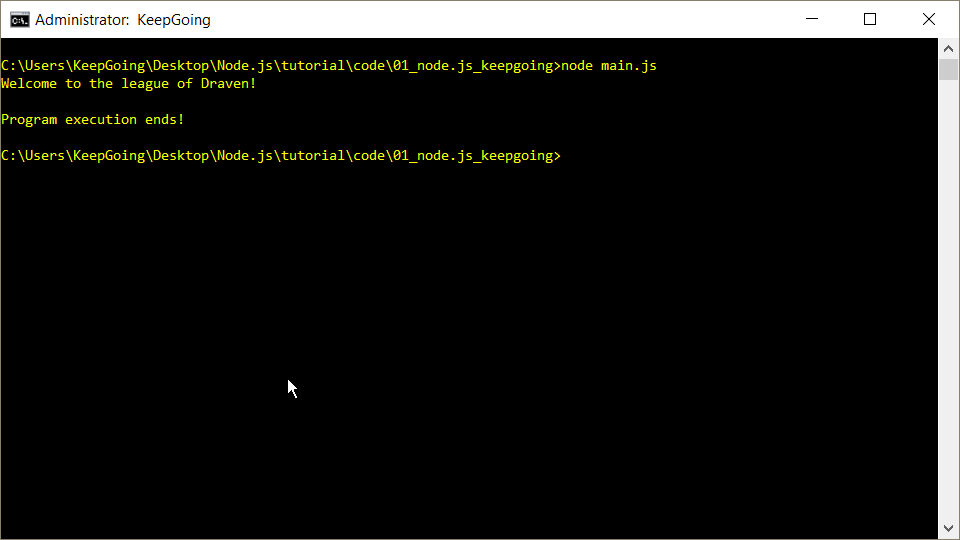
var fs = require("fs");

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

console.log(data.toString());

console.log("Program execution ends!");

代码运行结果如下



## 非阻塞代码实例

创建一个main02.js, 内容如下:

var fs = require("fs");

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

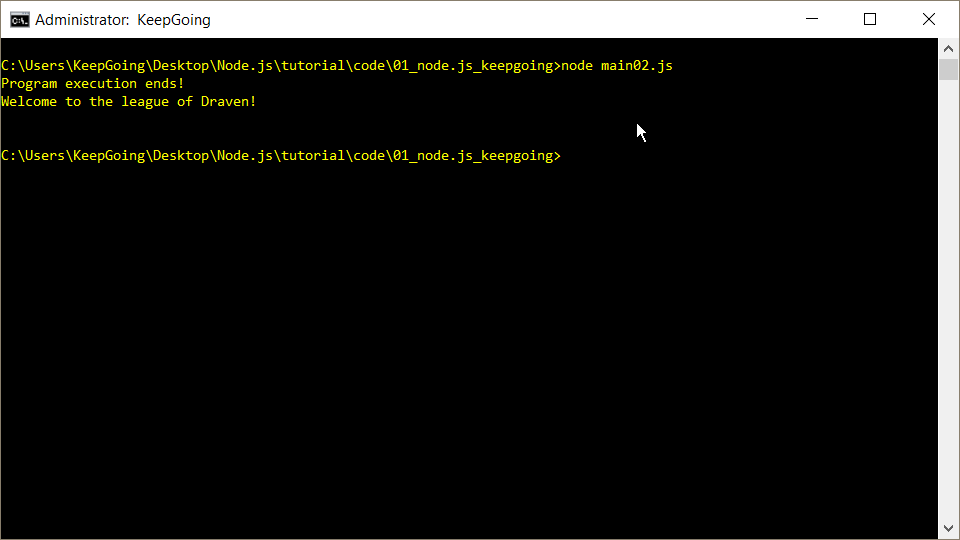
if (err) return console.error(err);

console.log(data.toString());

});

console.log("Program execution ends !");

代码运行结果如下



以上两个实例我们了解了阻塞与非阻塞调用的不同, 第一个实例的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.

# 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单线程类似进入了一个while(true)的时间循环, 直到没有事件观察者推出, 每个异步时间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, 就把它关闭后进行处理, 然后去服务下一个web请求.但这个请求完成, 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web server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I/O或者事件驱动I/O)

在事件驱动模型中, 会生成一个主循环来监听时间, 当检测到事件时触发回调函数.



整个事件就是这么实现的, 非常简洁, 有点类似于观察者模式, 事件相当于与一个主题(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();

// 绑定事件及事件的处理程序

eventEmitter.on('eventName', eventHandler);

// 触发事件

eventEmitter.emit('eventName');

创建event01.js

// 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(“Connection succeeded! ”);

// 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(“Data received successfully! ”);

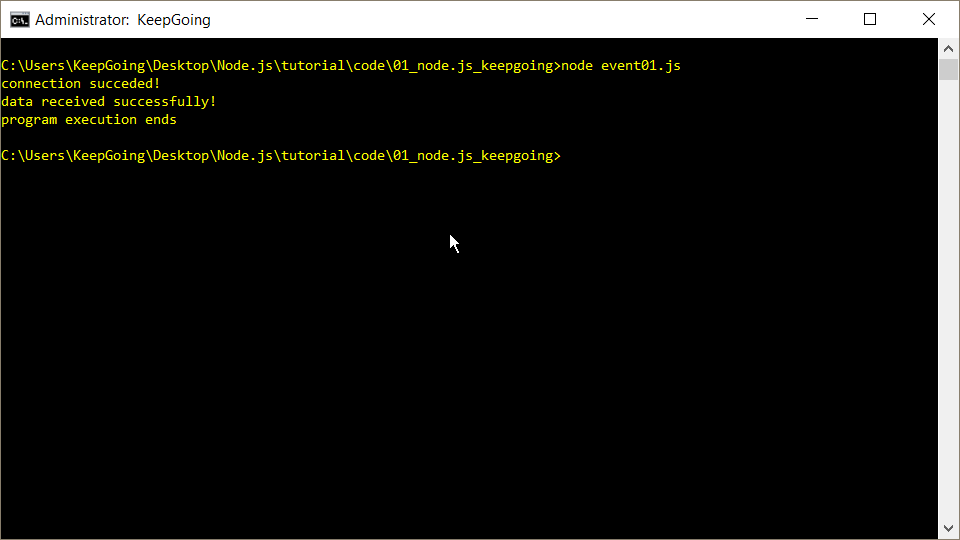
});

// 触发 connection 事件

eventEmitter.emit('connection');

console.log(“Program execution ends! ”);

执行代码



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, 回电函数接受错误对象组为第一个参数.

还是之前main02.js那个例子

var fs = require("fs");

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

if (err){

console.log(err.stack);

return;

}

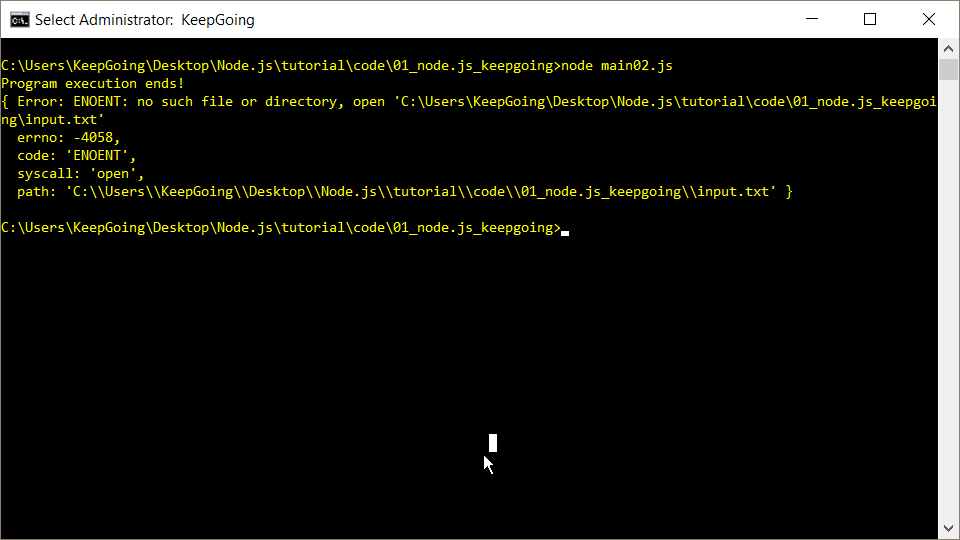
console.log(data.toString());

});

console.log("program execution ends!");

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. 如果没有发生错误, readFile()跳过err对象的输出, 文件内容就通过回调函数输出.

如果我们吧input.txt文件删除, 再次执行main02.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对象会在每次有新链接是奋发一个事件, 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.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, 就会触发”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事件会触发.

一个简单的例子说明EventEmitter的用法:

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

var event = new EventEmitter();

event.on("someEvents", function()

{

console.log("trigger someEvents");

});

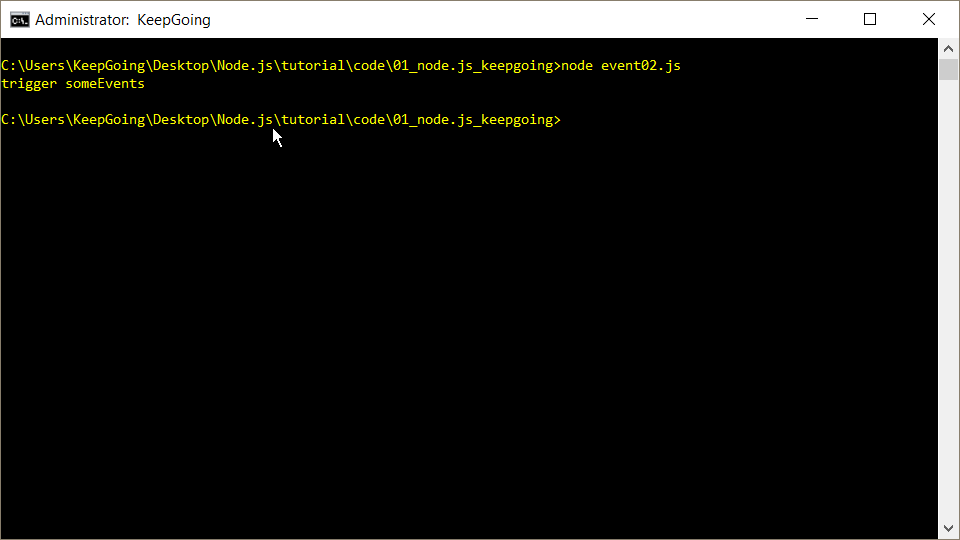
setTimeout(function()

{

event.emit("someEvents");

}, 1000);

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



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. 对于每个时间, 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.

var events = require("events");

var eventEmitter = new events.EventEmitter();

eventEmitter.on("someEvent", function(arg1, arg2)

{

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

});

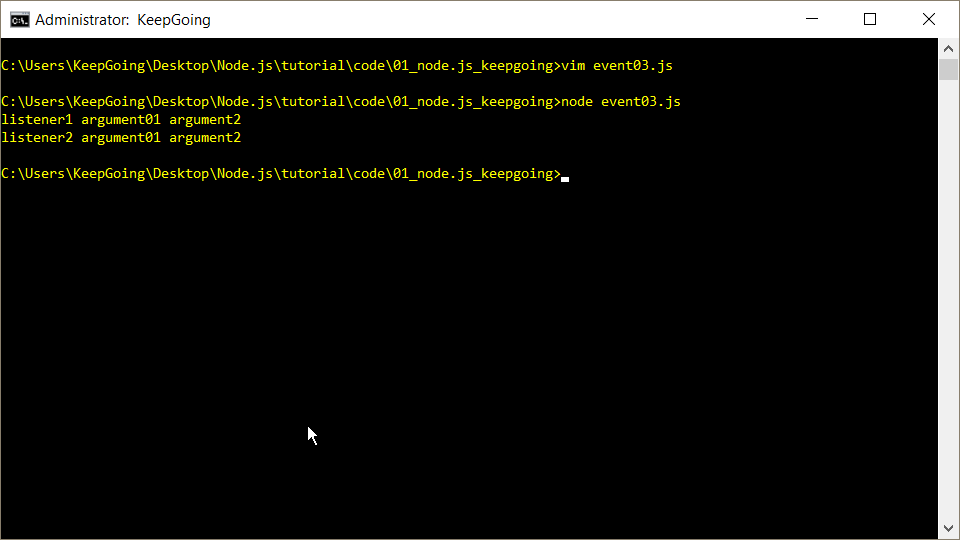
eventEmitter.on("someEvent", function(arg1, arg2)

{

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

});

eventEmitter.emit("someEvent", "argument01", "argument2");



以上的例子中, eventEmitter为时间someEvent注册了两个事件监听器, 然后出发了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函数用于绑定时间函数, emit属性用于触发一个事件.

## 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。  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。 |

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。");

运行结果

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。

# 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.

但是处理像TCP流或文件流时, 必须要使用到二进制数据, 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.js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类的事例中, 一个Buffer类似于一个整数数组, 但它对应于V8对内存之外的一块原始内存.

## 创建Buffer类

Node.js Buffer类可以通过多种方式来创建.

方法1

创建长度为10byte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("Here is KeepGoing!", "utf-8");

utf-8是默认编码

## 写入缓冲区

语法

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

参数描述

* 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UTF-8.

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

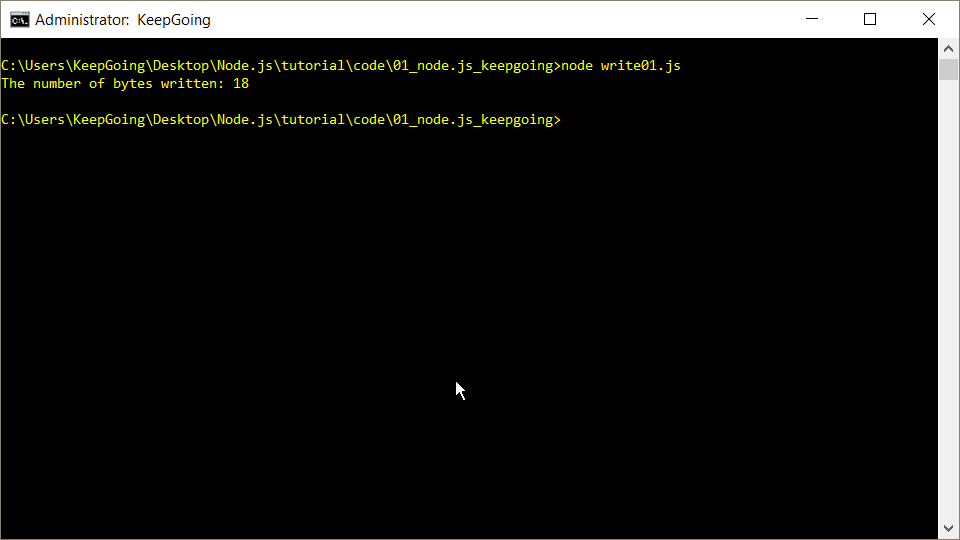
实例

buf = new Buffer(256);

len = buf.write("Here is KeepGoing!");

console.log("写入字节数 : "+ len);

执行以上代码， 输出结果为：



## 从缓冲区读取数据

语法

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

参数描述

* encoding： 使用的编码， 默认为UTF-8.
* start： 指定开始读取的索引位置， 默认为0.
* end： 结束为止， 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;

}

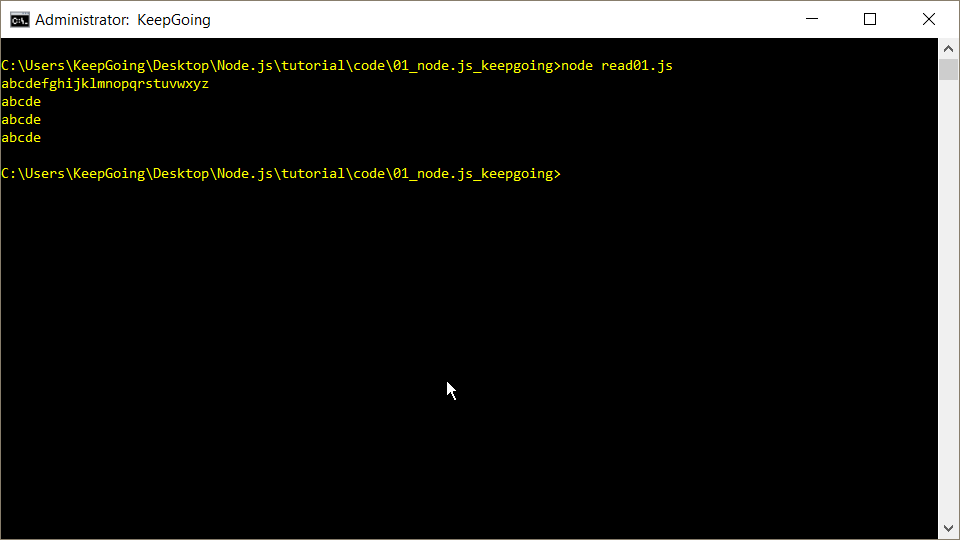
console.log( buf.toString('ascii')); // 输出: abcdefghijklmnopqrstuvwxyz

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：



## 将Buffer转换为JSON对象

语法

buffer.toJSON()

返回值

返回JSON对象

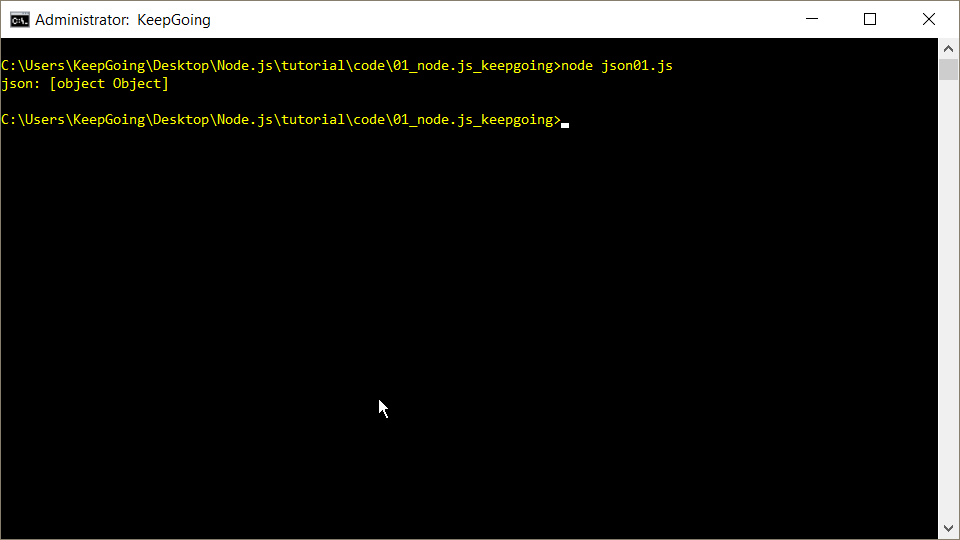
实例

var buf = new Buffer('Here is KeepGoing!');

var json = buf.toJSON(buf);

console.log(json);

执行以上代码， 输出结果为：



## 缓冲区合并

语法

Buffer.concat(list[, totalLength])

参数描述

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

返回值

返回一个或多个成员合并的新Buffer对象。

实例

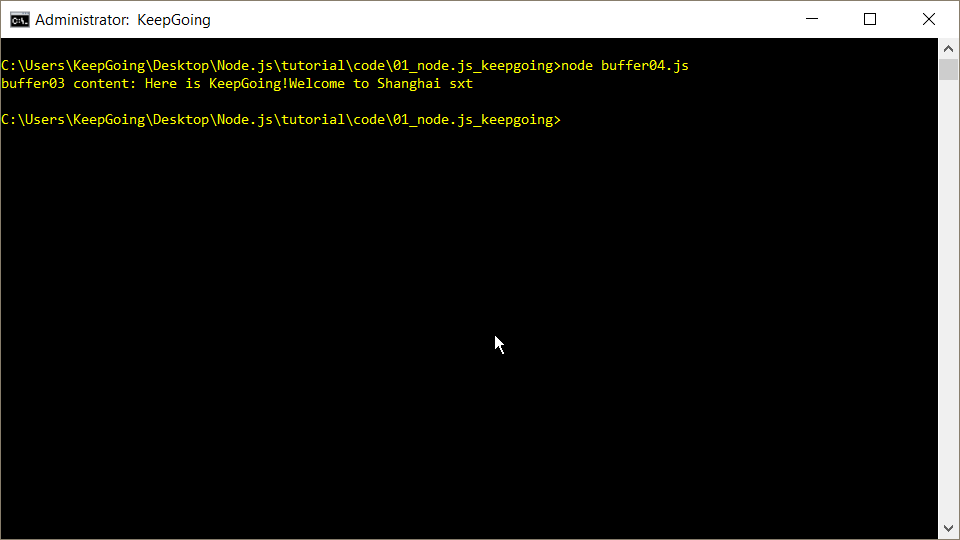
var buffer1 = new Buffer('Here is KeepGoing!');

var buffer2 = new Buffer('Welcome to Shanghai sxt');

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

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

执行以上代码， 输出结果为



## 缓冲区比较

语法

buf.compare(otherBuffer);

参数描述

* otherBuffer： 与buffer对象比较的另一个Buffer对象。

返回值

返回一个数字， 表示buffer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){

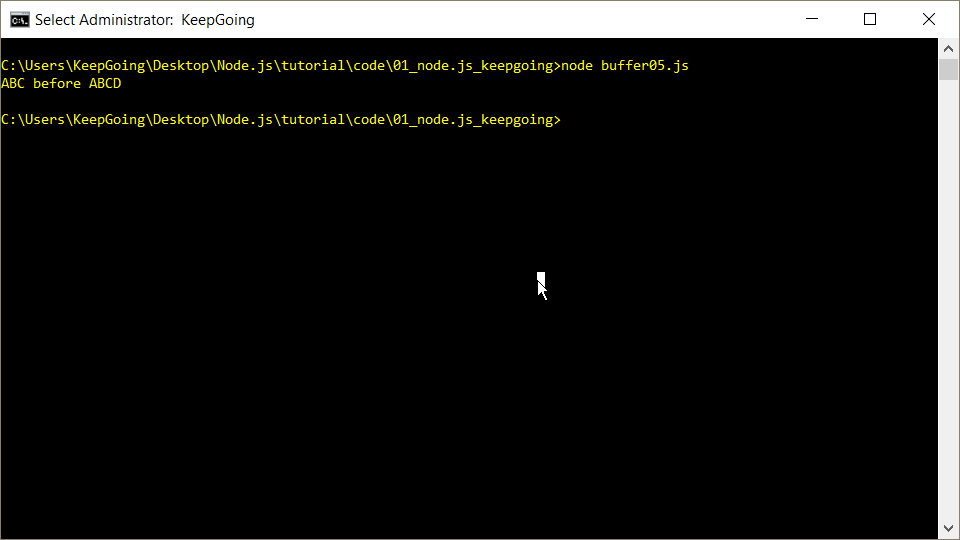
console.log(buffer1 + " 与 " + buffer2 + "相同");

}else {

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

}

执行以上代码， 输出结果为



## 拷贝缓冲区

语法

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

参数

* 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

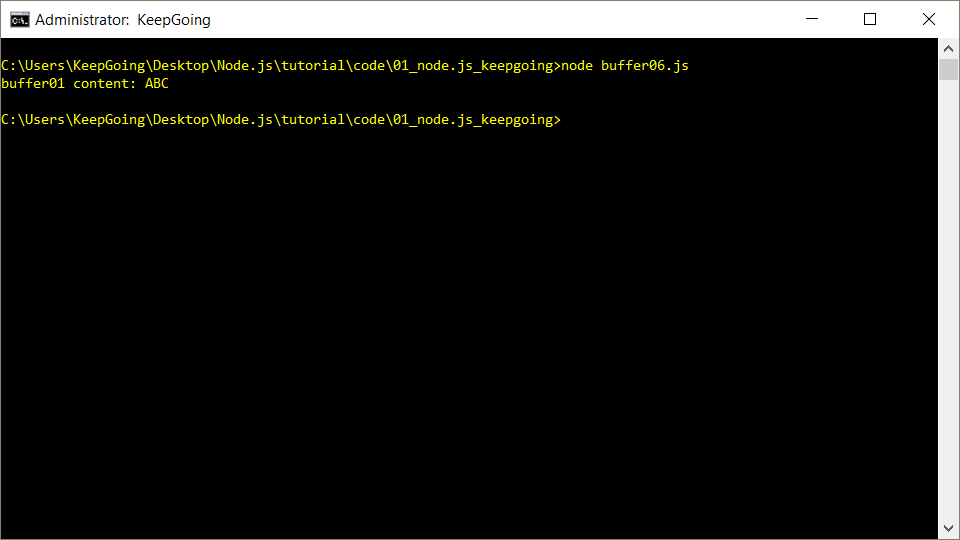
var buffer1 = new Buffer('ABC');

// 拷贝一个缓冲区

var buffer2 = new Buffer(3);

buffer1.copy(buffer2);

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



## 缓存区裁剪

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

参数描述

* start：
* end： 默认buffer.length

返回值

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

实例

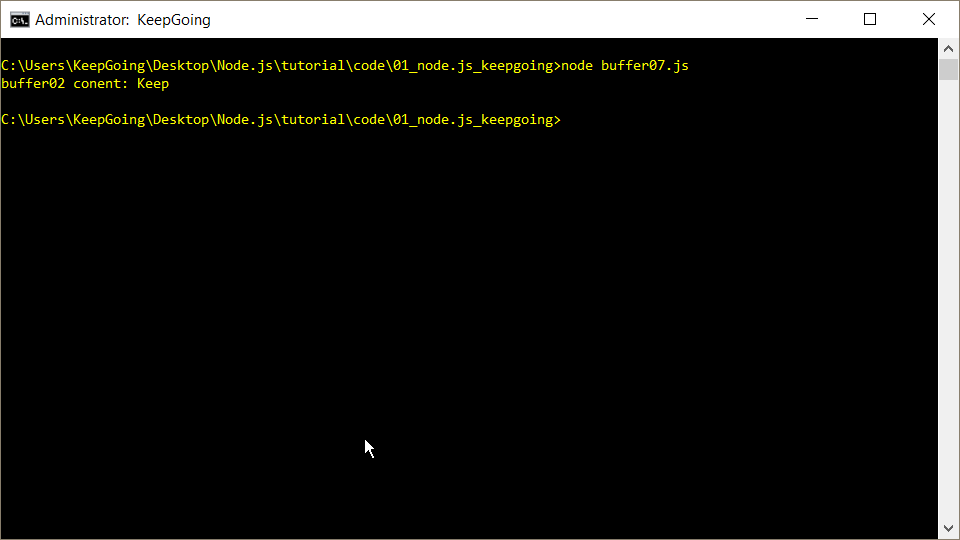
var buffer1 = new Buffer('runoob');

// 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：



## 缓冲区长度

语法

buf.length;

返回值

返回Buffer对象所占据的内存长度。

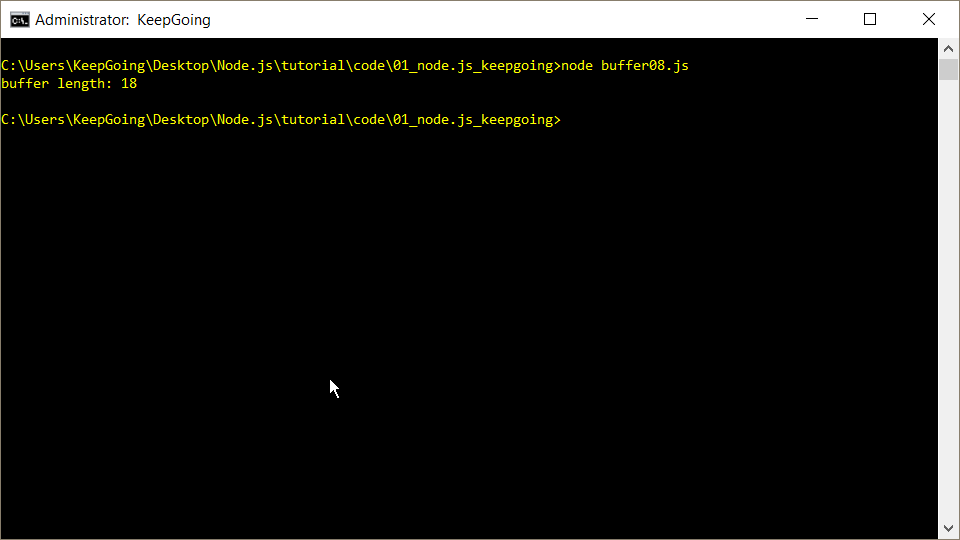
实例

var buffer = new Buffer('Here is KeepGoing!');

// 缓冲区长度

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

执行以上代码， 输出结果为：



## 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，支持 48 位计算，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，支持 48 位计算。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，支持 48 位计算。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，支持 48 位计算。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 位以下的数字。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 位以下的数字。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 位以下的数字。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 位以下的数字。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，读取一个有符号 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有符号 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有符号 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。 |
| 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有符号 32 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。 |
| 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有符号 32 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。 |
| 26 | **buf.readInt8(offset[, noAssert])** 根据指定的偏移量，读取一个 signed 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 signed 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。 |
| 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 signed 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。 |
| 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 signed 32 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。 |
| 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 signed 32 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。 |
| 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 位浮点数。 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。 |
| 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 位浮点数。 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。 |
| 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double。 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double。 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 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 必须是一个合法的有符号 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。 |
| 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 必须是一个合法的有符号 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。 |
| 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。 |
| 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。 |

# Stream

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

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

所有的Stream对象都是EventEmitter的实例。 常用的事件有：

* data： 当有数据可读时触发。
* end： 没有更多的数据可读时触发。
* error： 在接受和写入过程中发生错误时触发。
* finish： 所有数据已被写入到底层系统是触发。

## 从流中读数据

stream01.js

var fs = require("fs");

var data = '';

// 创建可读流

var readerStream = fs.createReadStream('input.txt');

// 设置编码为 utf8。

readerStream.setEncoding('UTF8');

// 处理流事件 --> data, end, and error

readerStream.on('data', function(chunk) {

data += chunk;

});

readerStream.on('end',function(){

console.log(data);

});

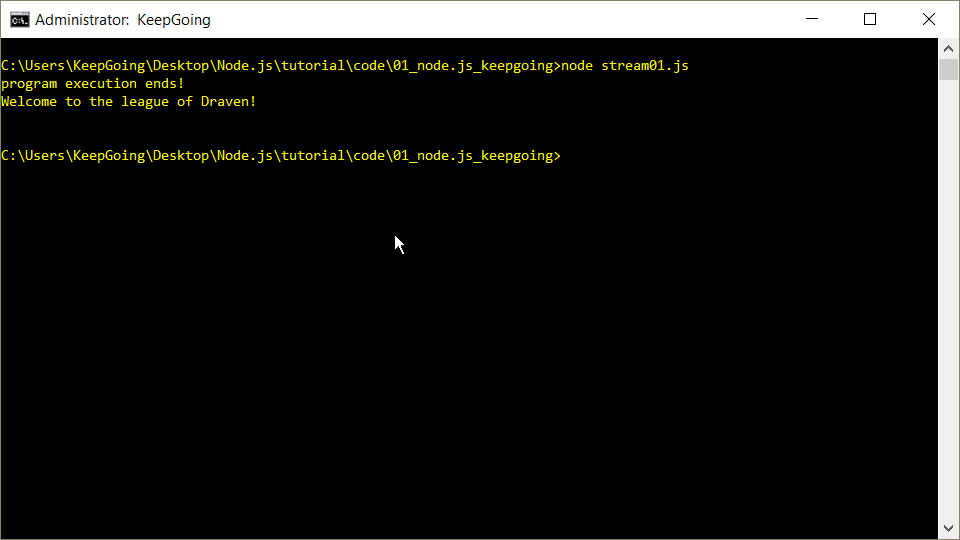
readerStream.on('error', function(err){

console.log(err.stack);

});

console.log("程序执行完毕");

代码执行结果如下：



## 写入流

var fs = require("fs");

var data = '菜鸟教程官网地址：www.runoob.com';

// 创建一个可以写入的流，写入到文件 output.txt 中

var writerStream = fs.createWriteStream('output.txt');

// 使用 utf8 编码写入数据

writerStream.write(data,'UTF8');

// 标记文件末尾

writerStream.end();

// 处理流事件 --> data, end, and error

writerStream.on('finish', function() {

console.log("写入完成。");

});

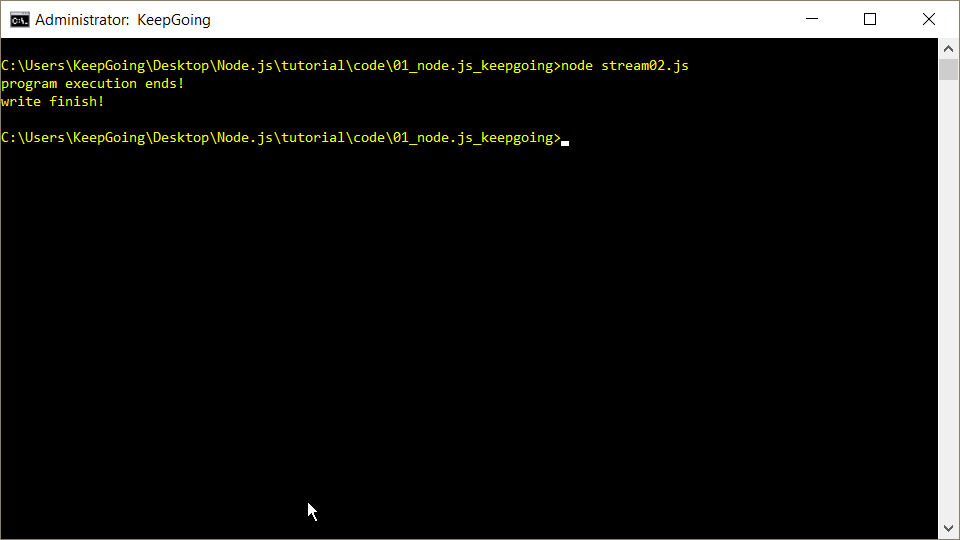
writerStream.on('error', function(err){

console.log(err.stack);

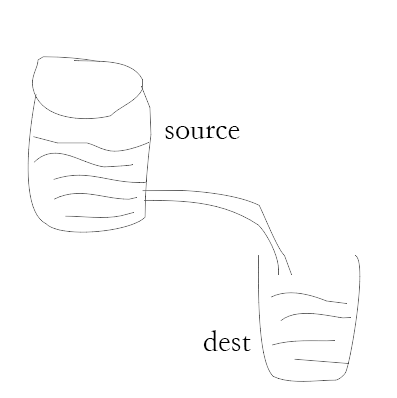
});

console.log("程序执行完毕");

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



## 管道流



如上图所示， 我们把文件比作装水的桶， 而水就是文件里的内容， 我们用一根管子（pipe）连接两个桶使得水从一个桶流入另一痛， 这样就慢慢的实现了大文件的复制过程。

var fs = require("fs");

// 创建一个可读流

var readerStream = fs.createReadStream('input.txt');

// 创建一个可写流

var writerStream = fs.createWriteStream('output.txt');

// 管道读写操作

// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中

readerStream.pipe(writerStream);

console.log("程序执行完毕");

## 链式流

链式流是通过连接输出流和另外一个流并创建多个对流操作链的机制。 链式流一般用于管道操作。

接下来我们就是用管道和链式流来压缩文件和解压文件。

var fs = require("fs");

var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.rar

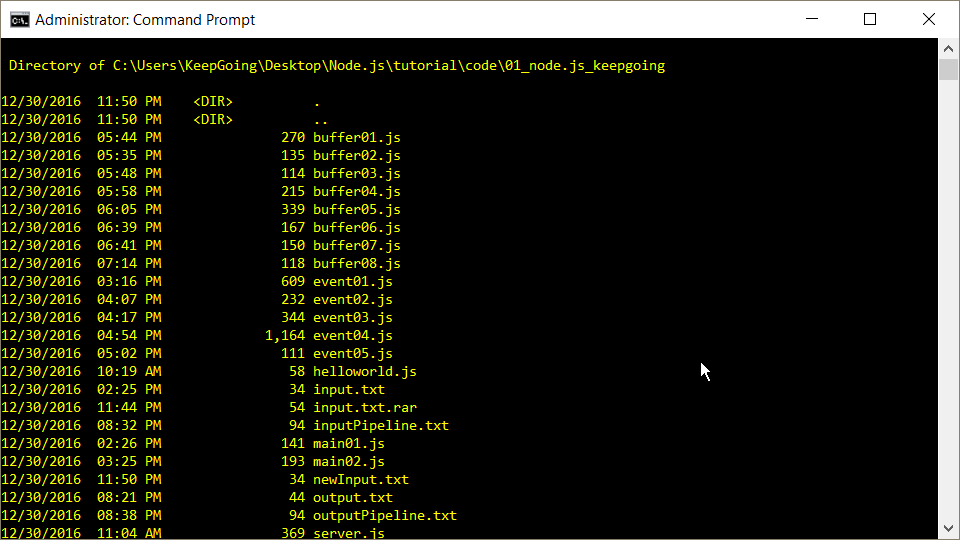
fs.createReadStream('input.txt')

.pipe(zlib.createGzip())

.pipe(fs.createWriteStream('input.txt.rar'));

console.log("文件压缩完成。");

代码执行结果如下



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

var fs = require("fs");

var zlib = require('zlib');

// 解压 input.txt.rar 文件为 newInput.txt

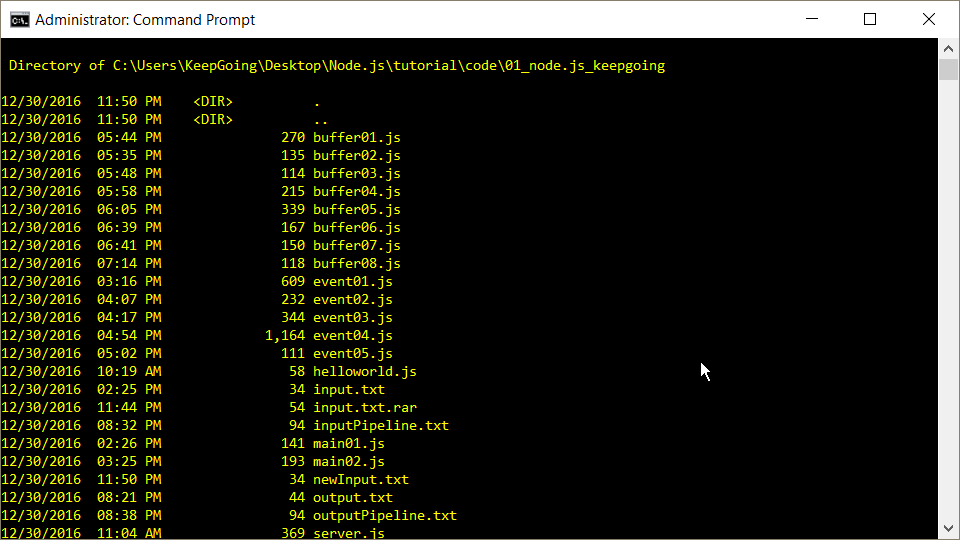
fs.createReadStream('input.txt.rar')

.pipe(zlib.createGunzip())

.pipe(fs.createWriteStream('newInput.txt'));

console.log("文件解压完成。");

代码执行结果如下：



执行完以上操作以后， 我们可以看到在当前目录生成了newInput.txt的解压缩文件。

# 模块系统

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

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

## 创建模块

var hello = require('./hello');

hello.world();

以上实例中， 代码require(“./hello”);引入了当前目录下的hello.js文件，(./为低昂前目录， 默认后缀为js)。 Node.js提供了exports和require两个对象， 其中exports是模块公开的口， require用于从外部获取一个模块的接口， 即所获取模块的exports对象。

hello.js

exports.world = function() {

console.log('Hello World');

}

在以上示例中， hello。js通过exports对象把world作为模块的访问接口， 在module01.js中通过require(“./hello”);加载这个模块， 然后就可以直接访问hello.js中exports对象的成员函数了。

有时候我们只是想把一个对象封装到模块中， 格式如下：

module.exports = function() {

// ...

}

例如

function Hello() {

var name;

this.setName = function(theName) {

name = theName;

};

this.sayHello = function() {

console.log('Hello ' + name);

};

};

module.exports = Hello;

然后就可以直接获取这个对象了：

var Hello = require('./hello');

hello = new Hello();

hello.setName('BYVoid');

hello.sayHello();

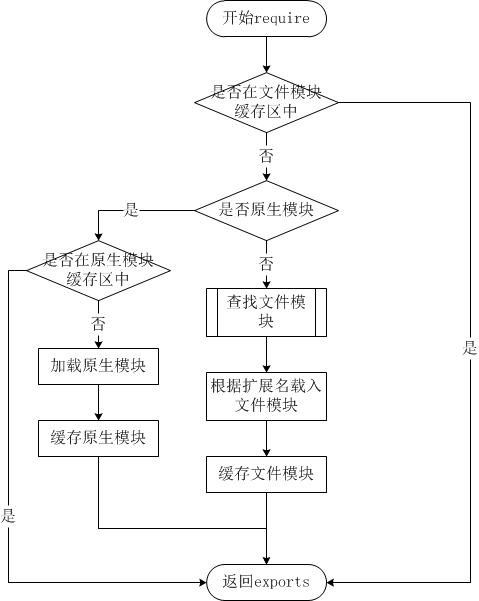
模块接口的唯一变化是使用module.exprts = Hello 代替了 exports.world = function(){}。 在外部引用该模块时， 其接口对象就是要输出的Hello对象本身， 而不是原先的exports

## 服务端的模块放在那里

Node.js中自带了一个叫做“http”的模块， 我们在我们的代码中请求他并把返回值赋给一个本地变量。这把我们的本地变量变成了一个拥有所有http模块所提供的公共方法的对象。

Node.js的require方法的文件查找策略如下：

由于Node.js中存在4类模块（原生模块和三种文件模块）， 尽管require方法及其简单， 但是内部的加载却是十分复杂的， 其加载优先级也各自不同。如下图所示：



从文件模块缓存中加载

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

从原生模块加载

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

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

从文件加载

当文件模块缓存中不存在， 而且不是原生模块的时候， Node.js会解require方法传入的参数， 并从文件系统中加载实际的文件。

require方法接收一下几种参数的传递：

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

# 函数

在JavaScript中， 一个函数可以作为另一个函数接收一个参数。 我们可以先定义一个函数， 然后传递， 也可以在传递参数的地方直接定义函数。

function say(word) {

console.log(word);

}

function execute(someFunction, value) {

someFunction(value);

}

execute(say, "Hello");

以上代码中， 我们把say函数作为execute函数的第一个参数进行行了传递。 这里返回的不是say的返回值， 而是say函数本身。 这样一来， say就变成了execute中的本地变量someFunction， execute可以通过调用someFunction()（带括号的形式）来使用say函数.

## 匿名函数

我们可以把一个函数作为变量传递， 但是我们不一定要绕过这个“先定义， 在传递”圈子， 我们可以直接在另一个函数的扩汉中定义和传递这个函数：

function execute(someFunction, value) {

someFunction(value);

}

execute(function(word){ console.log(word) }, "Hello");

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

用这种方式， 我们甚至不用给这个函数起名字， 这也是为什么它被叫做匿名函数。

# 路由

我们要为路由提供请求的URL和其他需要的GET及POST参数， 随后路由需要根据这些数据来执行相应的代码。因此， 我们需要查看HTTP请求， 从中提取出请求的URL以及GET/POST参数。 我们需要的所有数据都会包含在request对象中， 该对象作为onRequest()回调函数的第一个参数传递。 但是为了解析这数据， 我们需要额外的Node.js模块， 他们分别是url和querystirng模块。

url.parse(string).query

|

url.parse(string).pathname |

| |

| |

------ -------------------

http://localhost:8888/start?foo=bar&hello=world

--- -----

| |

| |

querystring(string)["foo"] |

|

querystring(string)["hello"]

我们给onRequest()函数加上一些逻辑， 用来找出浏览器的请求的URL路径：

var http = require("http");

var url = require("url");

function start() {

function onRequest(request, response) {

var pathname = url.parse(request.url).pathname;

console.log("Request for " + pathname + " received.");

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

response.write("Hello World");

response.end();

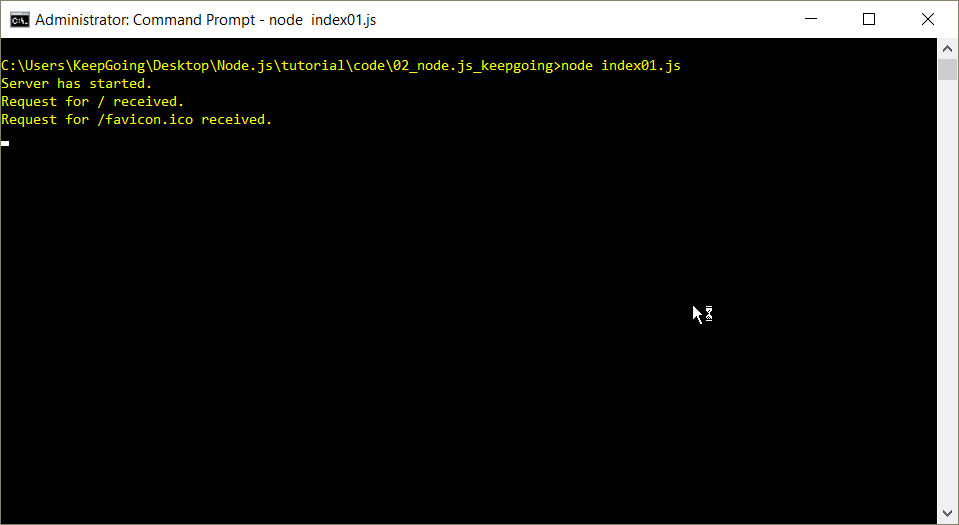
}

http.createServer(onRequest).listen(8888);

console.log("Server has started.");

}

exports.start = start;



我们的应用 现在可以通过请求的URL路径来区别不同的请求了， 这使我们得以使用路由来讲请求

以URL路径为基准映射到处理程序上。

## 编写路由

router.js

function route(pathname) {

console.log("About to route a request for " + pathname);

}

exports.route = route;

扩展服务器的start()函数， 以便将路由函数作为参数传递过去

var http = require("http");

var url = require("url");

function start(route) {

function onRequest(request, response) {

var pathname = url.parse(request.url).pathname;

console.log("Request for " + pathname + " received.");

route(pathname);

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

response.write("Hello World");

response.end();

}

http.createServer(onRequest).listen(8888);

console.log("Server has started.");

}

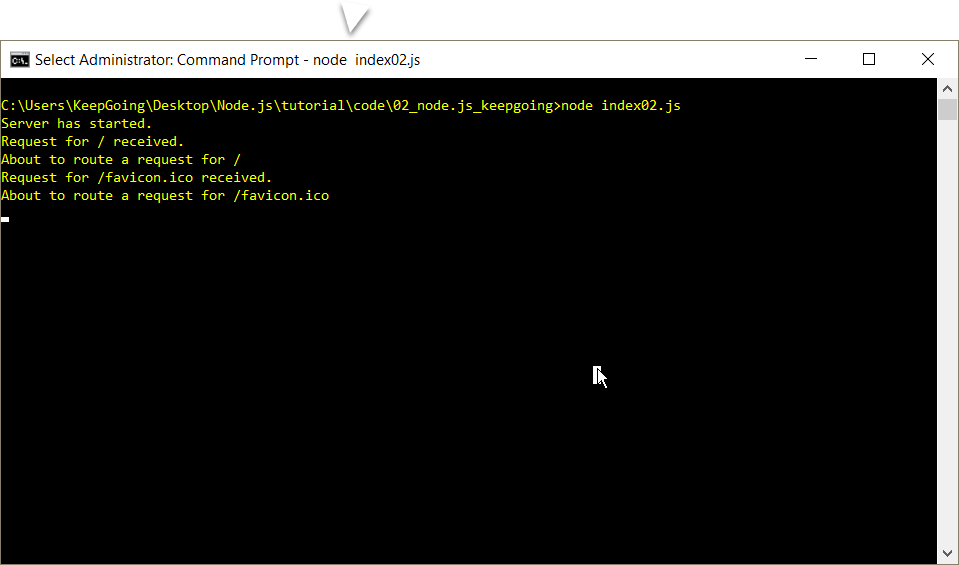
exports.start = start;

index.js

var server = require("./server");

var router = require("./router");

server.start(router.route);



# 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恶人和地方访问， 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。

## 全局对象与全局变量

global最根本的作用是作为全局变量的宿主。 安装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）。

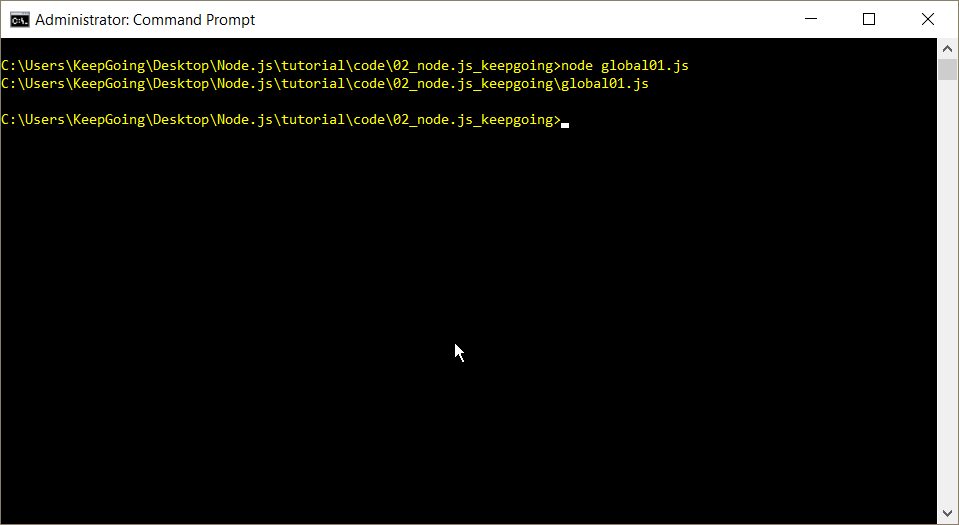
但你定义了一个全局变量时， 这个变量同时也会成为全局对象的属性， 需要注意的是， 在Node.js中你不可能在最外层定义变量， 因为所有用户代码都是属于当前模块的， 二模块本身不是最外层上下文。

注意：永远使用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。

\_\_filename

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

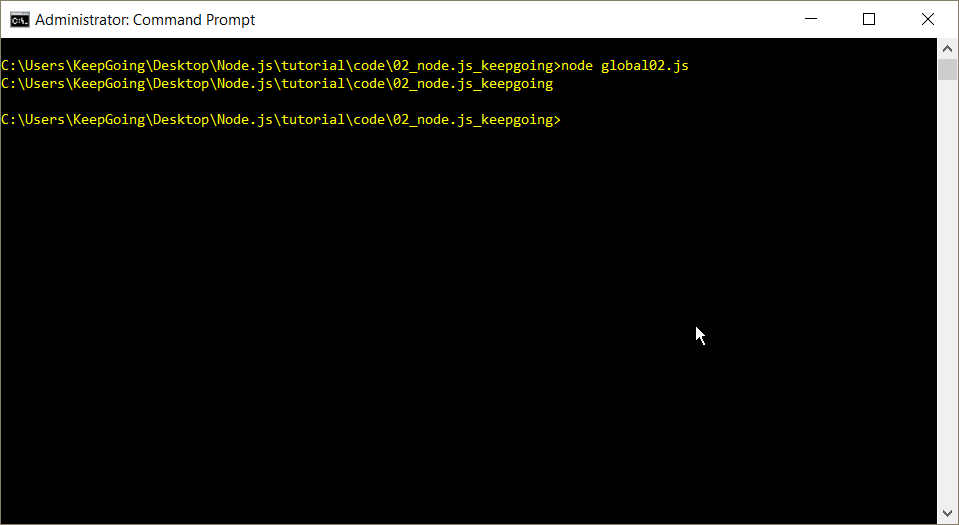
console.log( \_\_filename );



\_\_dirname

\_\_dirname表示当前执行脚本所在目录。

console.log( \_\_dirname );



setTimeout(cb, ms)

setTimeout(cb, ms)全局函数在指定的毫秒(ms)书后执行指定函数(cb)。setTimeout()只执行一次指定函数。返回一个代表定时器的句柄值。

function printHello(){

console.log( "Hello, World!");

}

// 两秒后执行以上函数

setTimeout(printHello, 2000);

clearTimeout(t)

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

function printHello(){

console.log( "Hello, World!");

}

// 两秒后执行以上函数

var t = setTimeout(printHello, 2000);

// 清除定时器

clearTimeout(t);

setInterval(cb, ms)

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

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

setInterval()反发挥不停的调用函数， 知道clearInterval()被调用或窗口被关闭。

function printHello(){

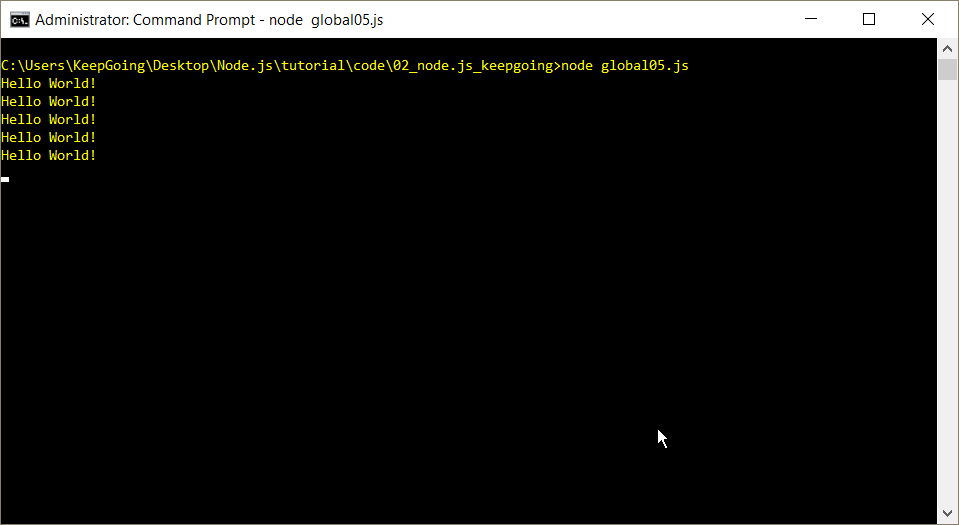
console.log( "Hello, World!");

}

// 两秒后执行以上函数

setInterval(printHello, 2000);

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



## console

console用于提供控制台标准输出， 他是有Internet Explorer的Jscript引擎提供的调试工具， 后来逐渐成为浏览器的事实标准。Node.js沿用了这个标准， 提供与习惯行为一致的console对象， 用于向标准输出流(stdout)或标准错误流(stderr)输出字符

console方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **console.log([data][, ...])** 向标准输出流打印字符并以换行符结束。该方法接收若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf() 命令的格式输出。 |
| 2 | **console.info([data][, ...])** 该命令的作用是返回信息性消息，这个命令与console.log差别并不大，除了在chrome中只会输出文字外，其余的会显示一个蓝色的惊叹号。 |
| 3 | **console.error([data][, ...])** 输出错误消息的。控制台在出现错误时会显示是红色的叉子。 |
| 4 | **console.warn([data][, ...])** 输出警告消息。控制台出现有黄色的惊叹号。 |
| 5 | **console.dir(obj[, options])** 用来对一个对象进行检查（inspect），并以易于阅读和打印的格式显示。 |
| 6 | **console.time(label)** 输出时间，表示计时开始。 |
| 7 | **console.timeEnd(label)** 结束时间，表示计时结束。 |
| 8 | **console.trace(message[, ...])** 当前执行的代码在堆栈中的调用路径，这个测试函数运行很有帮助，只要给想测试的函数里面加入 console.trace 就行了。 |
| 9 | **console.assert(value[, message][, ...])** 用于判断某个表达式或变量是否为真，接手两个参数，第一个参数是表达式，第二个参数是字符串。只有当第一个参数为false，才会输出第二个参数，否则不会有任何结果。 |

console.log() 向标准输出流打印字符并以换行符结束。

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

第一个参数是一个字符串， 如果没有参数， 只打印一个换行。

console.log('Hello world');

console.log('byvoid%diovyb');

console.log('byvoid%diovyb', 1991);

运行结果

Hello world

byvoid%diovyb

byvoid1991iovyb

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

console.trace();

实例

onsole.info("程序开始执行：");

var counter = 10;

console.log("计数: %d", counter);

console.time("获取数据");

//

// 执行一些代码

//

console.timeEnd('获取数据');

console.info("程序执行完毕。")

## process

process是一个全局变量， 即global对象的属性。

它用于描述当前Node.js进程状态的对象， 提供了一个与操作系统的简单接口。 通常在你写本地命令行程序的时候， 与它打交道。

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

实例

process.on('exit', function(code) {

// 以下代码永远不会执行

setTimeout(function() {

console.log("该代码不会执行");

}, 0);

console.log('退出码为:', code);

});

console.log("程序执行结束");

### 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。 |

### 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。 |

实例

// 输出到终端

process.stdout.write("Hello World!" + "\n");

// 通过参数读取

process.argv.forEach(function(val, index, array) {

console.log(index + ': ' + val);

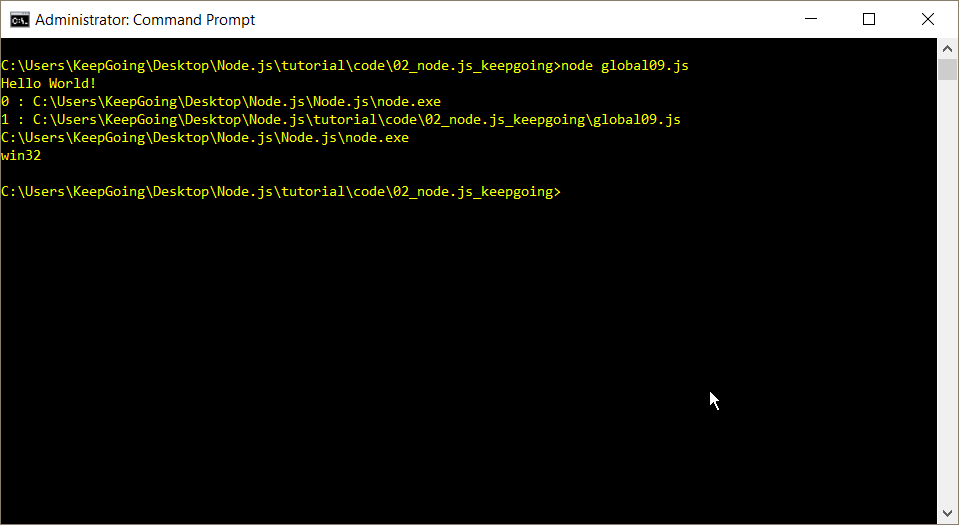
});

// 获取执行路局

console.log(process.execPath);

// 平台信息

console.log(process.platform);



### 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，所有你需要有 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，所有你需要有 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，调用回到函数。 |
| 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。 |

实例

// 输出当前目录

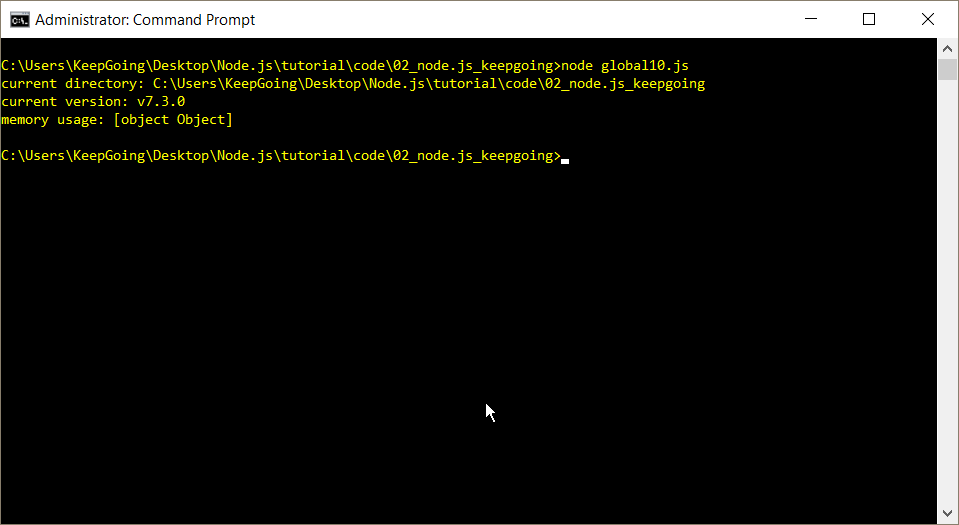
console.log('当前目录: ' + process.cwd());

// 输出当前版本

console.log('当前版本: ' + process.version);

// 输出内存使用情况

console.log(process.memoryUsage());



# 常用工具

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

## util.inherits

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

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

var util = require('util');

function Base() {

this.name = 'base';

this.base = 1991;

this.sayHello = function() {

console.log('Hello ' + this.name);

};

}

Base.prototype.showName = function() {

console.log(this.name);

};

function Sub() {

this.name = 'sub';

}

util.inherits(Sub, Base);

var objBase = new Base();

objBase.showName();

objBase.sayHello();

console.log(objBase);

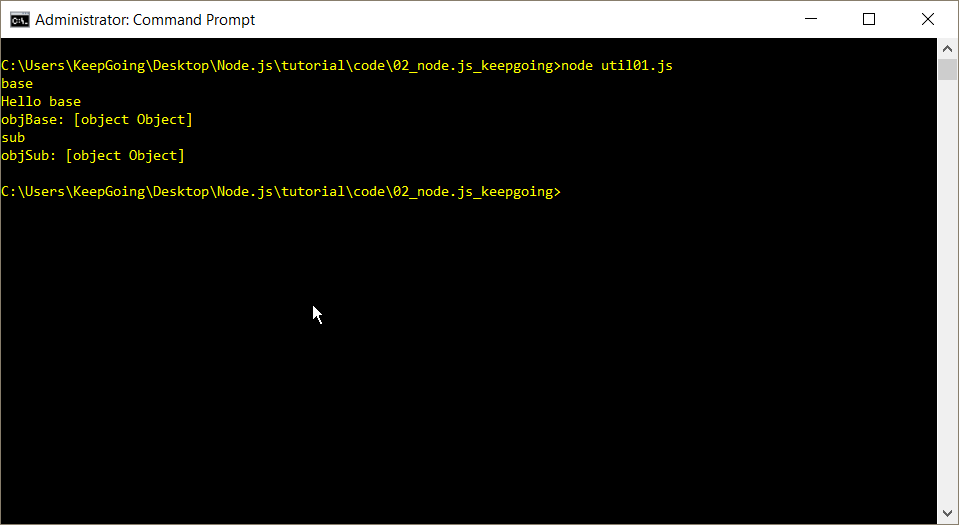
var objSub = new Sub();

objSub.showName();

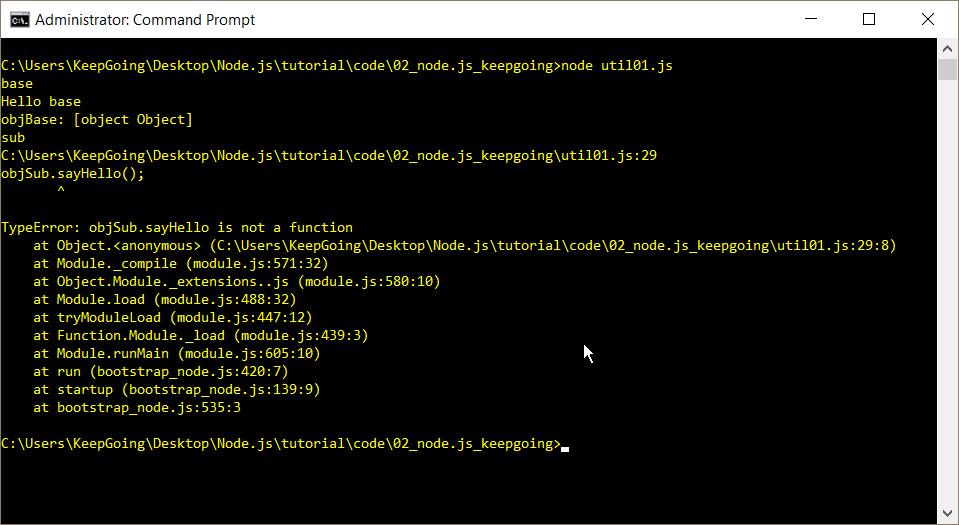
//objSub.sayHello();

console.log(objSub);

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



注意：Sub仅仅继承了Base在原型中定义的函数， 而构造函数内部创造的base属性和sayHello函数都没有被Sub继承。 同时， 在原型中定义的属性不会被console.log作为对象的属性输出。 如果我们去掉objSub.sayHello()这行的注释， 将会看到：



## util.inspect

util.inspect(object,[showHidden], [depth], [colors]);是一个将任意对象转换成字符串的方法， 通常用于调试和错误输出。 他至少接收一个参数object， 既要转换的对象。

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

depth表示最大递归层数， 如果对象很复杂， 你可以指定层数已控制输出信息的多少， 如果不指定depth， 默认会递归2层， 指定为nul表示将不限递归层数完成遍历对象。 如果color值为true， 输出格式将会以ANISI颜色编码， 通常用于在terminal显示更漂亮的效果。

util.inspect并不会简单地直接把对象转化为字符串， 即使该对象定义了toString方法也不会调用。

var util = require('util');

function Person() {

this.name = 'byvoid';

this.toString = function() {

return this.name;

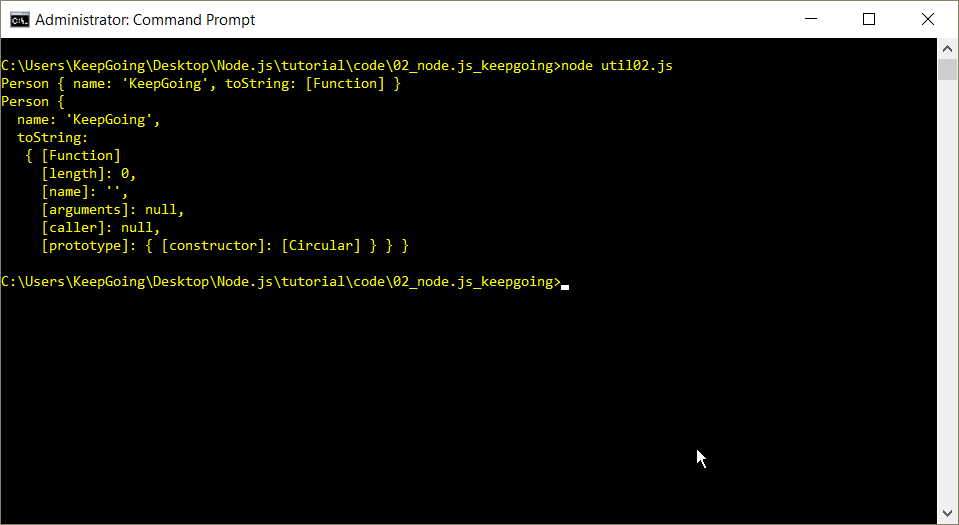
};

}

var obj = new Person();

console.log(util.inspect(obj));

console.log(util.inspect(obj, true));



## util.isArray(object)

如果给定的参数object是一个数组返回true， 否则返回false

var util = require('util');

util.isArray([])

// true

util.isArray(new Array)

// true

util.isArray({})

// false

## util.isREgExp(object)

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

var util = require('util');

util.isRegExp(/some regexp/)

// true

util.isRegExp(new RegExp('another regexp'))

// true

util.isRegExp({})

// false

## util.isDate(object)

如果给定的参数object是一个日期返回true， 否则返回false

var util = require('util');

util.isDate(new Date())

// true

util.isDate(Date())

// false (without 'new' returns a String)

util.isDate({})

// false

## util.isError(object)

如果给定的参数object是一个错误对象返回true， 否则返回false。

var util = require('util');

util.isError(new Error())

// true

util.isError(new TypeError())

// true

util.isError({ name: 'Error', message: 'an error occurred' })

// false

官方文档：  <http://nodejs.org/api/util.html>

# 文件系统

Node.js 提供了一组类似UNIX(POSIX)标准的文件操作API。 Node.js导入文件系统模块(fs)语法如下所示：

var fs = require("fs")

## 异步和同步

Node.js文件系统(fs模块)模块中的方法均有异步和同步版本， 例如读取文件内容的函数有异步的fs.readFile();和同步的fs.readFileSync();

异步的方法函数最后一个参数为回调函数， 回调函数的第一个参数包含了错误信息error。

建议大家使用异步方法， 比起同步， 异步方法性能更高， 速度更快， 而且没有阻塞。

实例

var fs = require("fs");

// 异步读取

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

if (err) {

return console.error(err);

}

console.log("异步读取: " + data.toString());

});

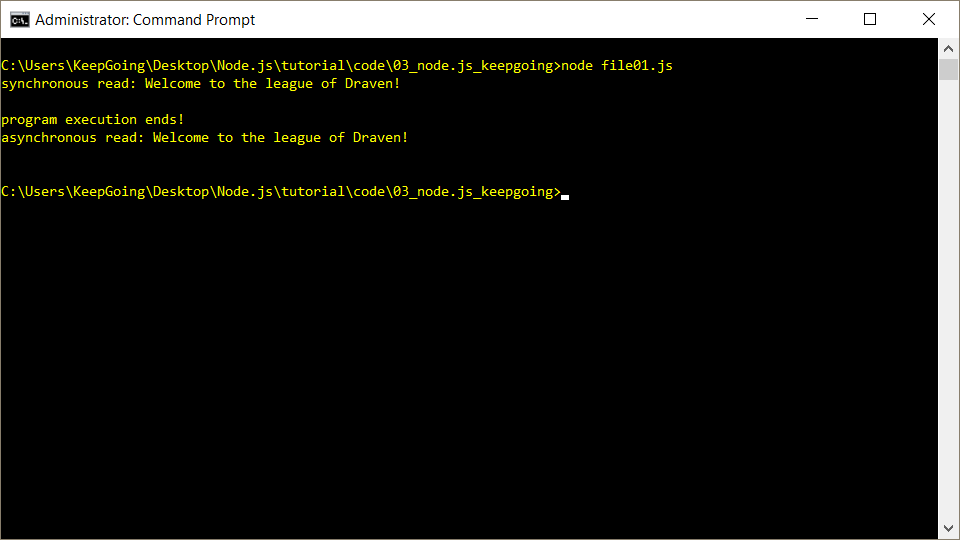
// 同步读取

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

console.log("同步读取: " + data.toString());

console.log("程序执行完毕。");

以上代码执行结果如下：



## 打开文件

语法

以下为在异步模式下打开文件的语法格式

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

参数

* path： 文件的路径。
* flags： 文件打开的行为。
* mode：　设置文件模式（权限），　文件创建默认权限为0666（可读， 可写）。
* callback： 回调函数， 带有两个参数如：callback(err, fs);

flags参数可以是以下值：

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

实例

var fs = require("fs");

// 异步打开文件

console.log("准备打开文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

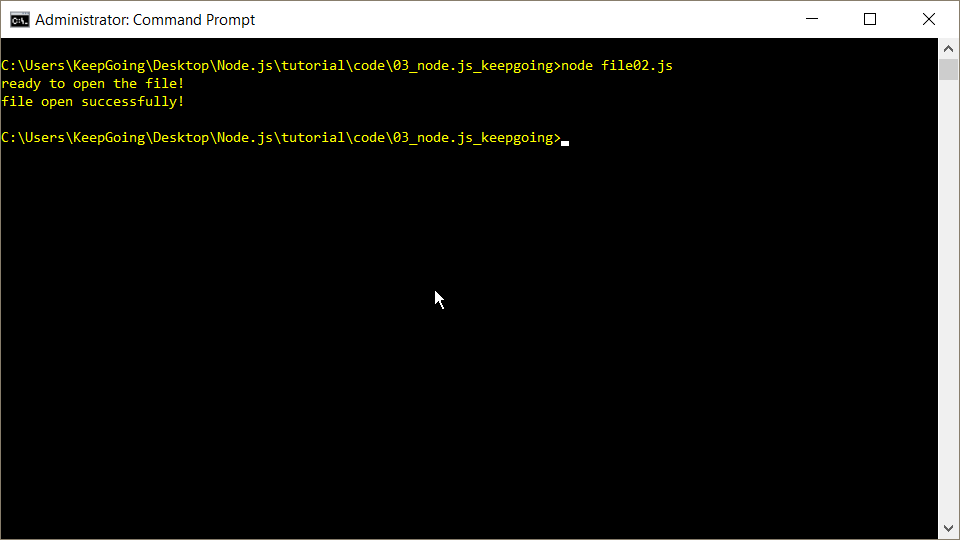
return console.error(err);

}

console.log("文件打开成功！");

});

以上代码执行结果为：



## 获取文件信息

语法

以下为通过异步模式获取文件信息的语法格式

fs.stat(path, callback)

参数：

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

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

var fs = require('fs');

fs.stat('/Users/liuht/code/itbilu/demo/fs.js', function (err, stats) {

console.log(stats.isFile()); //true

})

stats类中的方法有：

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

var fs = require("fs");

console.log("准备打开文件！");

fs.stat('input.txt', function (err, stats) {

if (err) {

return console.error(err);

}

console.log(stats);

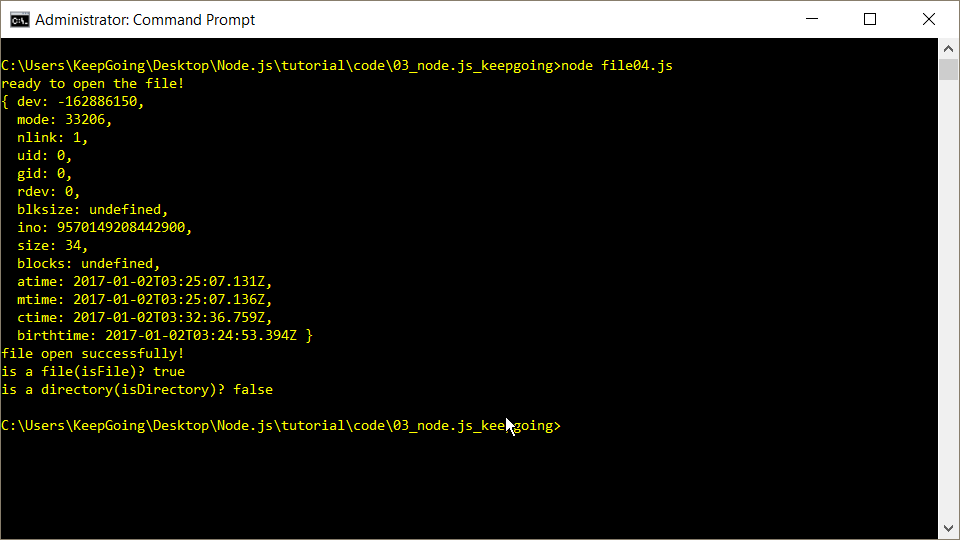
console.log("读取文件信息成功！");

// 检测文件类型

console.log("是否为文件(isFile) ? " + stats.isFile());

console.log("是否为目录(isDirectory) ? " + stats.isDirectory());

以上代码执行结果如下：



## 写入文件

语法

以下为异步模式下写入文件的语法格式：

fs.writeFile(filename, data[, options], callback)

如果文件存在， 该方法写入的内容会覆盖旧的文件内容。

参数

* path： 文件路径。
* data： 要写入文件的数据， 可以是String（字符串）或Buffer（流）对象。
* options： 该参数是一个对象， 包含{encoding， mode， flag}。 默认编码为UTF-8， 模式为0666， flag为“w”
* callback： 回调函数， 回调函数只包含错误信息参数（err）， 在写入失败时返回。

实例

var fs = require("fs");

console.log("准备写入文件");

fs.writeFile('input.txt', '我是通过写入的文件内容！', function(err) {

if (err) {

return console.error(err);

}

console.log("数据写入成功！");

console.log("--------我是分割线-------------")

console.log("读取写入的数据！");

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

if (err) {

return console.error(err);

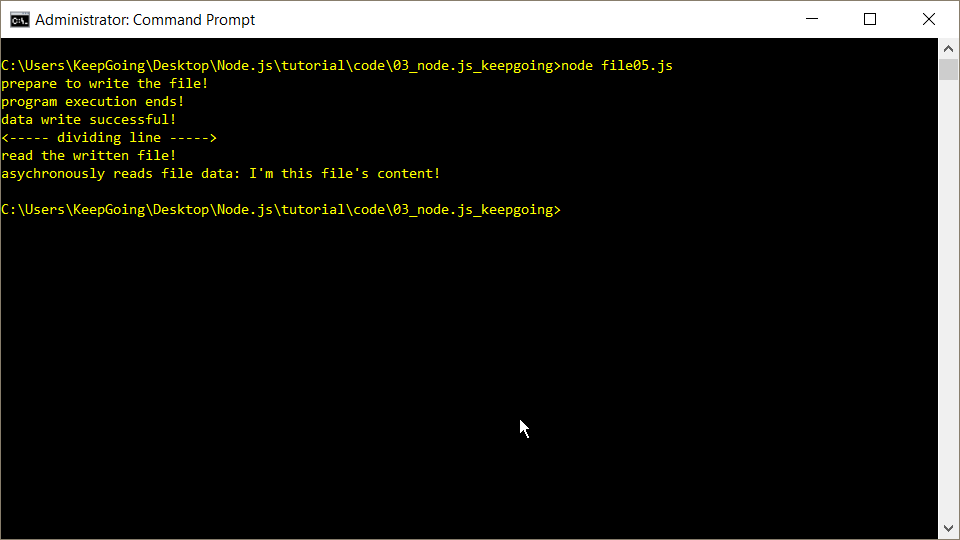
}

console.log("异步读取文件数据: " + data.toString());

});

});

以上代码执行结果如下：



## 读取文件

语法

以下为异步模式下读取文件的语法格式：

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

该方法使用了文件描述符来读取文件。

参数

* fd： 通过fs.open()方法返回的文件描述符。
* buffer： 数据写入的缓冲区。
* offset： 数据写入的写入偏移量。
* length： 要从文件中读取的字节数。
* position： 文件读取的起始位置， 如果position的值为null， 则会从当前文件指针的位置读取。
* callback： 回调函数， 有三个参数err， bytesRead， buffer， err为错误信息， bytesRead表示读取的字节数， buffer为缓冲区对象。

实例

var fs = require("fs");

var buf = new Buffer(1024);

console.log("准备打开已存在的文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

console.log("准备读取文件：");

fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){

if (err){

console.log(err);

}

console.log(bytes + " 字节被读取");

// 仅输出读取的字节

if(bytes > 0){

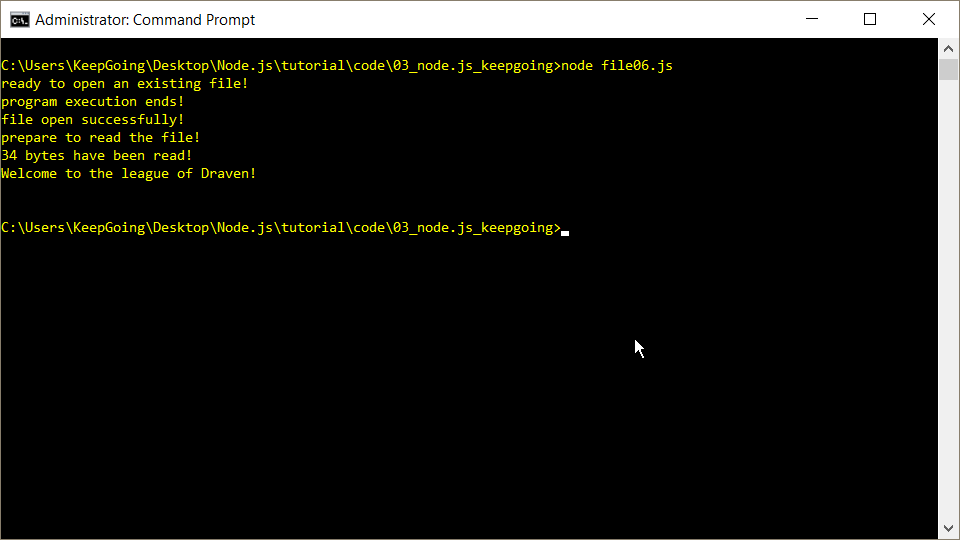
console.log(buf.slice(0, bytes).toString());

}

});

});

以下代码执行结果如下：



## 关闭文件

语法

以下为异步模式下关闭文件的语法格式

fs.close(fd, callback)

该方法使用了文件描述符来读取文件

参数

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

实例

var fs = require("fs");

var buf = new Buffer(1024);

console.log("准备打开文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

console.log("准备读取文件！");

fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){

if (err){

console.log(err);

}

// 仅输出读取的字节

if(bytes > 0){

console.log(buf.slice(0, bytes).toString());

}

// 关闭文件

fs.close(fd, function(err){

if (err){

console.log(err);

}

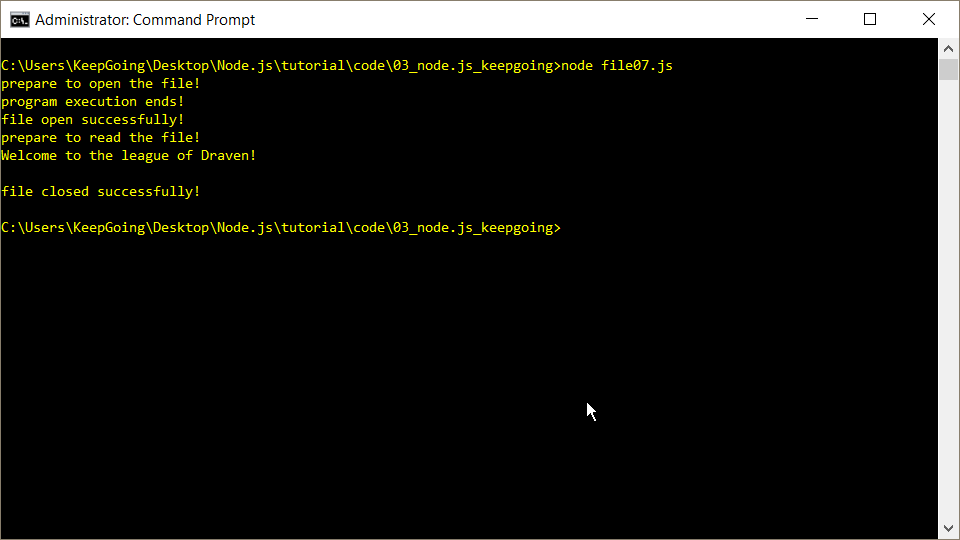
console.log("文件关闭成功");

});

});

});

以上代码执行结果如下：



## 截取文件

语法

以下为异步模式下截取文件的语法格式

fs.ftruncate(fd, len, callback)

该方法使用了文件描述符来读取文件

参数

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

实例

var fs = require("fs");

var buf = new Buffer(1024);

console.log("准备打开文件！");

fs.open('input.txt', 'r+', function(err, fd) {

if (err) {

return console.error(err);

}

console.log("文件打开成功！");

console.log("截取10字节后的文件内容。");

// 截取文件

fs.ftruncate(fd, 10, function(err){

if (err){

console.log(err);

}

console.log("文件截取成功。");

console.log("读取相同的文件");

fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){

if (err){

console.log(err);

}

// 仅输出读取的字节

if(bytes > 0){

console.log(buf.slice(0, bytes).toString());

}

// 关闭文件

fs.close(fd, function(err){

if (err){

console.log(err);

}

console.log("文件关闭成功！");

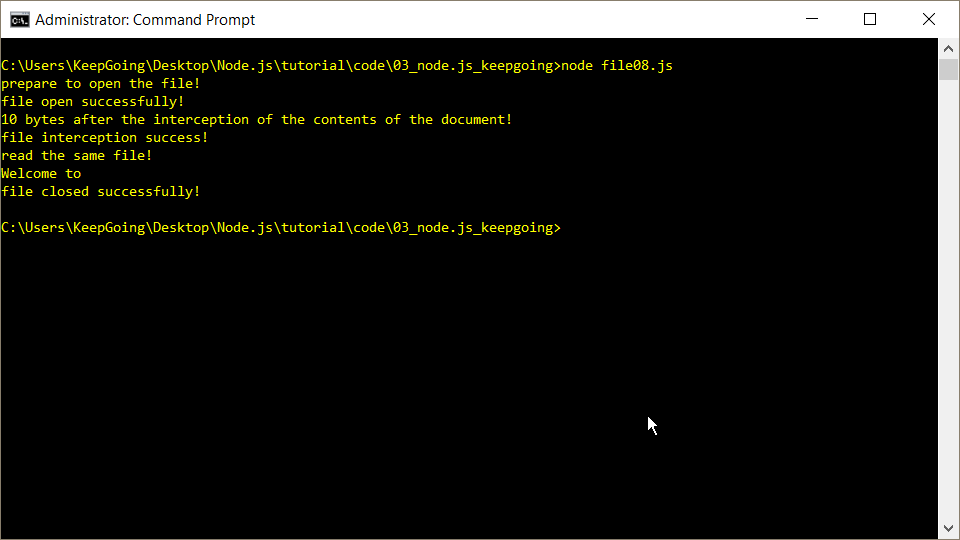
});

});

});

});

以上代码执行结果如下：



## 删除文件

语法

fs.unlink(path, callback)

参数

* path： 文件路径。
* callback： 回调函数， 没有参数

实例

var fs = require("fs");

console.log("准备删除文件！");

fs.unlink('input.txt', function(err) {

if (err) {

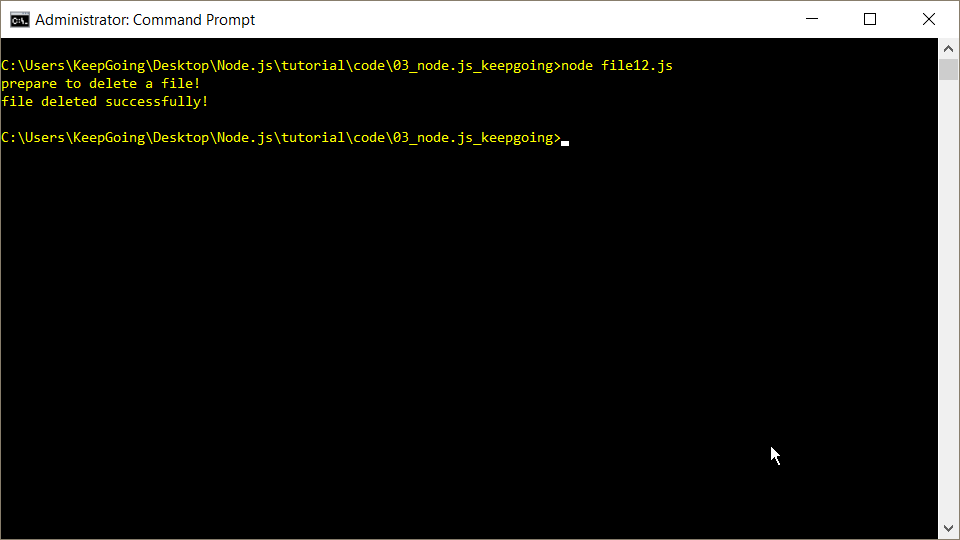
return console.error(err);

}

console.log("文件删除成功！");

});

以上代码执行结果如下



## 创建目录

语法

fs.mkdir(path[, mode], callback)

参数

* path： 文件路径。
* mode： 设置目录权限， 默认为0777.
* callback： 回调函数， 没有参数。

实例

var fs = require("fs");

console.log("创建目录 /tmp/test/");

fs.mkdir("/tmp/test/",function(err){

if (err) {

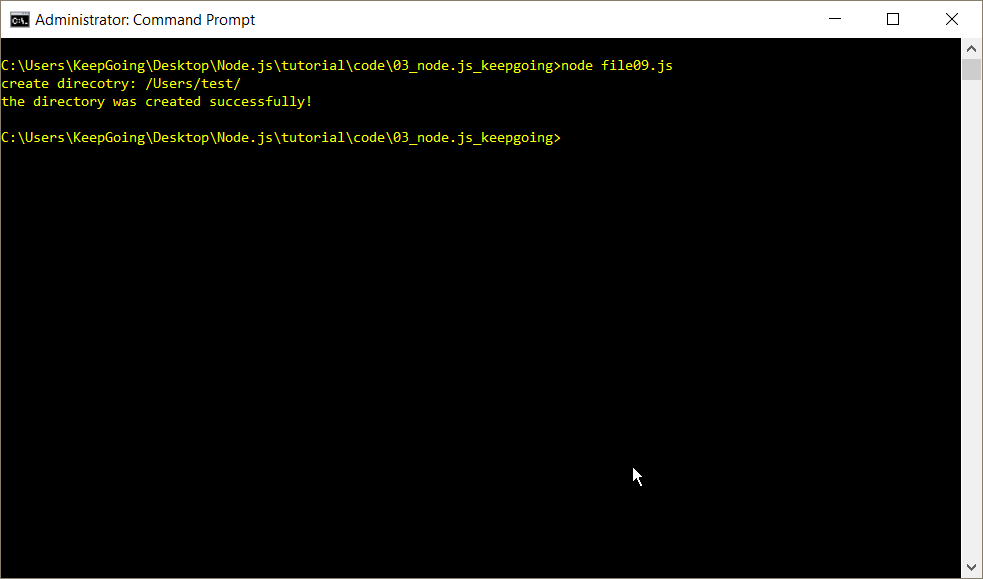
return console.error(err);

}

console.log("目录创建成功。");

});

以上代码执行结果如下：



## 读取目录

语法

fs.readdir(path, callback)

参数

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

实例

var fs = require("fs");

console.log("查看 /tmp 目录");

fs.readdir("/tmp/",function(err, files){

if (err) {

return console.error(err);

}

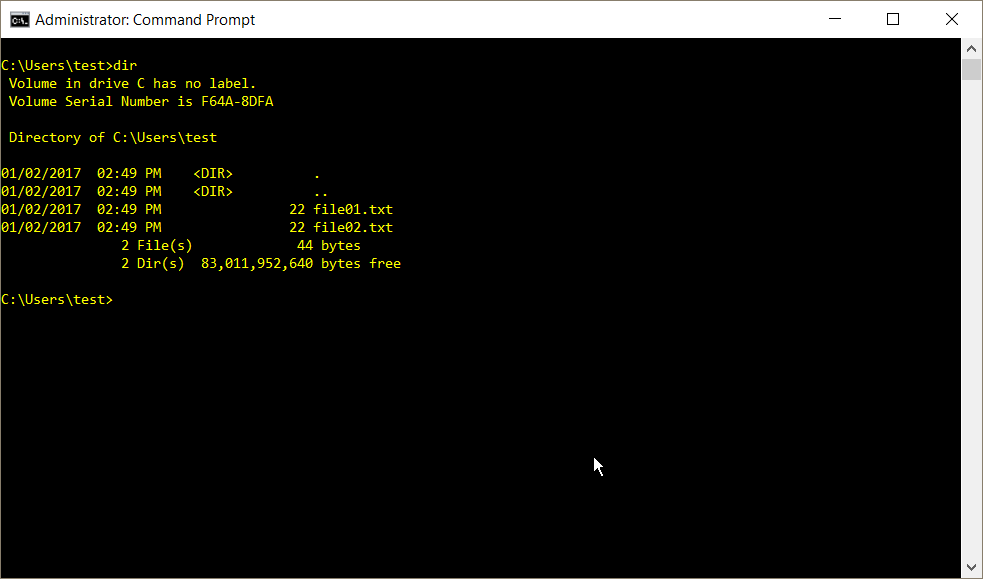
files.forEach( function (file){

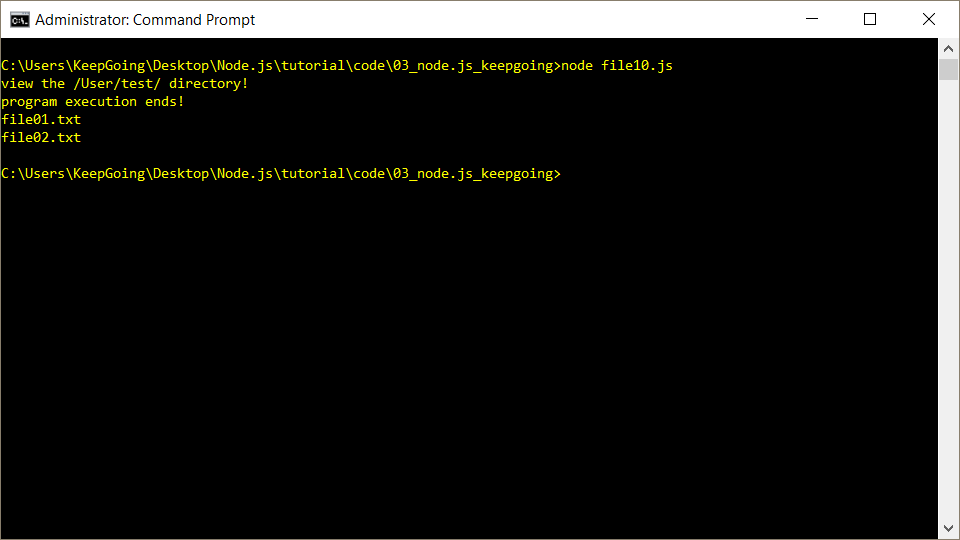
console.log( file );

});

});

以上代码执行结果如下





## 删除目录

语法

fs.rmdir(path, callback)

参数

* path： 文件路径。
* callback： 回调函数， 没有参数。

实例

var fs = require("fs");

console.log("准备删除目录 /tmp/test");

fs.rmdir("/tmp/test",function(err){

if (err) {

return console.error(err);

}

console.log("读取 /tmp 目录");

fs.readdir("/tmp/",function(err, files){

if (err) {

return console.error(err);

}

files.forEach( function (file){

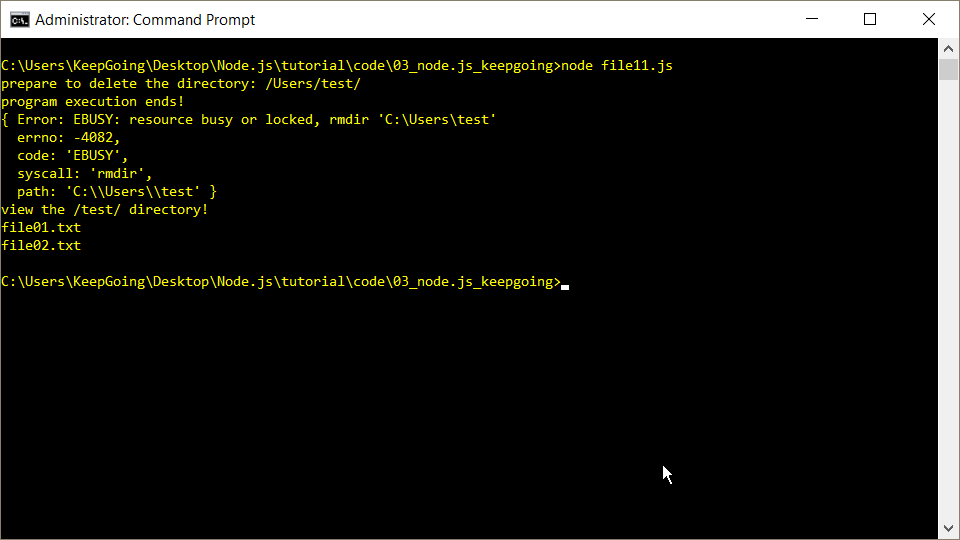
console.log( file );

});

});

});

以上代码运行结果为



## 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。 mode defaults to 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])**<="" 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。 |

official link: <https://nodejs.org/api/fs.html#fs_fs_rename_oldpath_newpath_callback>

# GET/POST请求

在很多场景中， 我们的服务器都需要跟用户的浏览器打交道， 如表单提交。

表单提交到服务器一般都是用GET/POST请求。

## 获取GET请求内容

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

Node.js中url模块的parse函数提供了这个功能。

var http = require('http');

var url = require('url');

var util = require('util');

http.createServer(function(req, res){

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

res.end(util.inspect(url.parse(req.url, true)));

}).listen(8888);

获取URL的参数

实例

var http = require('http');

var url = require('url');

var util = require('util');

http.createServer(function(req, res){

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

// 解析 url 参数

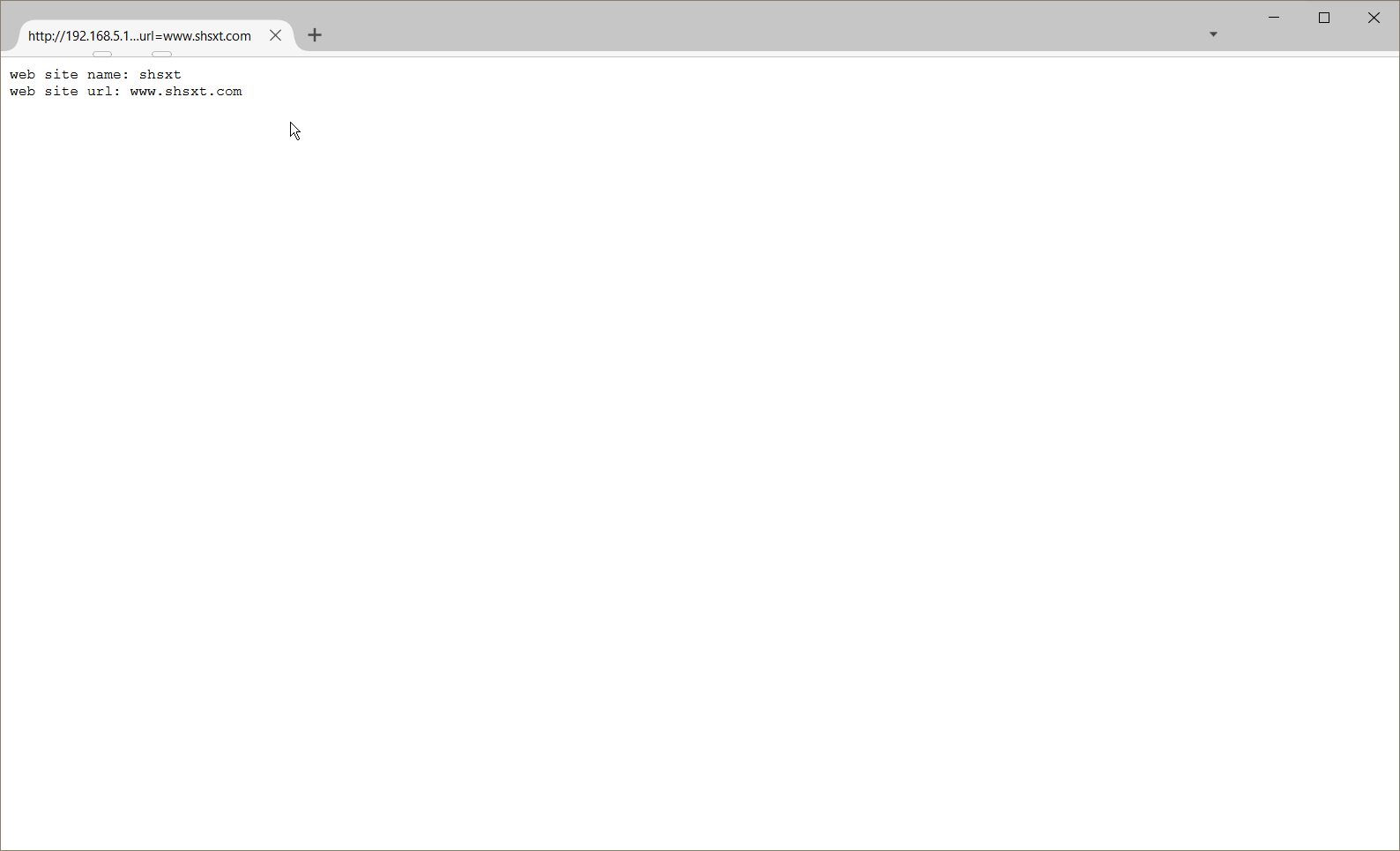
var params = url.parse(req.url, true).query;

res.write("网站名：" + params.name); res.write("\n");

res.write("网站 URL：" + params.url); res.end();

}).listen(8888);

在浏览器中访问: <http://localhost:8888/user?name=shsxt&url=www.shsxt.com>



## 获取POST请求内容

POST请求内的内容全部都在请求体中， http.ServerRequest并没有一个属性内容为请求体， 原因是等待请求体传输是一件耗时的工作。

比如上传文件， 而很多时候我们可能并必须要理会请求体中的内容， 恶意的POST请求回答大小号服务器的资源， 所有 Node.js默认是不会解析请求体的， 当你需要的时候， 需要手动来做。

var http = require('http');

var querystring = require('querystring');

http.createServer(function(req, res){

// 定义了一个post变量，用于暂存请求体的信息

var post = '';

// 通过req的data事件监听函数，每当接受到请求体的数据，就累加到post变量中

req.on('data', function(chunk){ post += chunk; });

// 在end事件触发后，通过querystring.parse将post解析为真正的POST请求格式，然后向客户端返回。

req.on('end', function(){

post = querystring.parse(post);

res.end(util.inspect(post)); });

}).listen(3000);

实例

var http = require('http');

var querystring = require('querystring');

var postHTML =

'<html><head><meta charset="utf-8"><title>KeepGoing</title></head>' +

'<body>' +

'<form method="post">' +

'网站名： <input name="name"><br>' +

'网站 URL： <input name="url"><br>' +

'<input type="submit">' +

'</form>' +

'</body></html>';

http.createServer(function (req, res) {

var body = "";

req.on('data', function (chunk) {

body += chunk;

});

req.on('end', function () {

// 解析参数

body = querystring.parse(body);

// 设置响应头部信息及编码

res.writeHead(200,

{'Content-Type': 'text/html; charset=utf8'});

if(body.name && body.url) {

// 输出提交的数据

res.write("网站名：" + body.name);

res.write("<br>");

res.write("网站 URL：" + body.url);

} else {

// 输出表单

res.write(postHTML);

}

res.end();

});

}).listen(3000);

访问： <http://localhost:8888>

# 工具模块

在Node.js模块库中有很多好用的模块， 一下是一种比较常用的模块

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

## OS模块

var os = require("os")

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **os.tmpdir()** 返回操作系统的默认临时文件夹。 |
| 2 | **os.endianness()** 返回 CPU 的字节序，可能的是 "BE" 或 "LE"。 |
| 3 | **os.hostname()** 返回操作系统的主机名。 |
| 4 | **os.type()** 返回操作系统名 |
| 5 | **os.platform()** 返回操作系统名 |
| 6 | **os.arch()** 返回操作系统 CPU 架构，可能的值有 "x64"、"arm" 和 "ia32"。 |
| 7 | **os.release()** 返回操作系统的发行版本。 |
| 8 | **os.uptime()** 返回操作系统运行的时间，以秒为单位。 |
| 9 | **os.loadavg()** 返回一个包含 1、5、15 分钟平均负载的数组。 |
| 10 | **os.totalmem()** 返回系统内存总量，单位为字节。 |
| 11 | **os.freemem()** 返回操作系统空闲内存量，单位是字节。 |
| 12 | **os.cpus()** 返回一个对象数组，包含所安装的每个 CPU/内核的信息：型号、速度（单位 MHz）、时间（一个包含 user、nice、sys、idle 和 irq 所使用 CPU/内核毫秒数的对象）。 |
| 13 | **os.networkInterfaces()** 获得网络接口列表。 |

属性

|  |  |
| --- | --- |
| **序号** | **属性 & 描述** |
| 1 | **os.EOL** 定义了操作系统的行尾符的常量。 |

实例

var os = require("os");

// CPU 的字节序

console.log('endianness : ' + os.endianness());

// 操作系统名

console.log('type : ' + os.type());

// 操作系统名

console.log('platform : ' + os.platform());

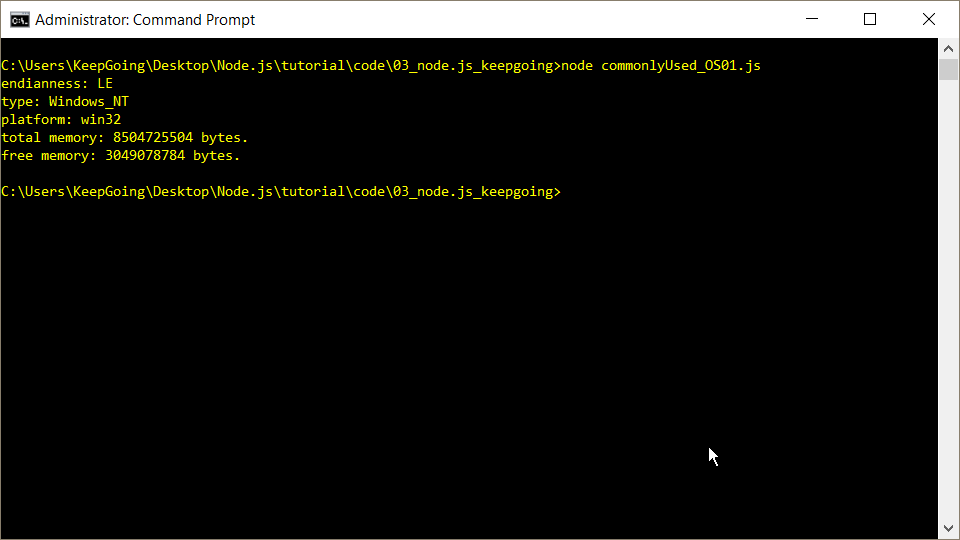
// 系统内存总量

console.log('total memory : ' + os.totalmem() + " bytes.");

// 操作系统空闲内存量

console.log('free memory : ' + os.freemem() + " bytes.");

代码执行结果如下



## Path模块

var path = require("path")

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **path.normalize(p)** 规范化路径，注意'..' 和 '.'。 |
| 2 | **path.join([path1][, path2][, ...])** 用于连接路径。该方法的主要用途在于，会正确使用当前系统的路径分隔符，Unix系统是"/"，Windows系统是"\"。 |
| 3 | **path.resolve([from ...], to)** 将 **to** 参数解析为绝对路径。 |
| 4 | **path.isAbsolute(path)** 判断参数 **path** 是否是绝对路径。 |
| 5 | **path.relative(from, to)** 用于将相对路径转为绝对路径。 |
| 6 | **path.dirname(p)** 返回路径中代表文件夹的部分，同 Unix 的dirname 命令类似。 |
| 7 | **path.basename(p[, ext])** 返回路径中的最后一部分。同 Unix 命令 bashname 类似。 |
| 8 | **path.extname(p)** 返回路径中文件的后缀名，即路径中最后一个'.'之后的部分。如果一个路径中并不包含'.'或该路径只包含一个'.' 且这个'.'为路径的第一个字符，则此命令返回空字符串。 |
| 9 | **path.parse(pathString)** 返回路径字符串的对象。 |
| 10 | **path.format(pathObject)** 从对象中返回路径字符串，和 path.parse 相反。 |

属性

|  |  |
| --- | --- |
| **序号** | **属性 & 描述** |
| 1 | **path.sep** 平台的文件路径分隔符，'\\' 或 '/'。 |
| 2 | **path.delimiter** 平台的分隔符, ; or ':'. |
| 3 | **path.posix** 提供上述 path 的方法，不过总是以 posix 兼容的方式交互。 |
| 4 | **path.win32** 提供上述 path 的方法，不过总是以 win32 兼容的方式交互。 |

实例

var path = require("path");

// 格式化路径

console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

// 连接路径

console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..'));

// 转换为绝对路径

console.log('resolve : ' + path.resolve('main.js'));

// 路径中文件的后缀名

console.log('ext name : ' + path.extname('main.js'));

以上代码执行结果如下

normalization : /test/test1/2slashes/1slash

joint path : /test/test1/2slashes/1slash

resolve : /web/com/1427176256\_27423/main.js

ext name : .js

## Net模块

var net = require("net")

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **net.createServer([options][, connectionListener])** 创建一个 TCP 服务器。参数 connectionListener 自动给 'connection' 事件创建监听器。 |
| 2 | **net.connect(options[, connectionListener])** 返回一个新的 'net.Socket'，并连接到指定的地址和端口。 当 socket 建立的时候，将会触发 'connect' 事件。 |
| 3 | **net.createConnection(options[, connectionListener])** 创建一个到端口 port 和 主机 host的 TCP 连接。 host 默认为 'localhost'。 |
| 4 | **net.connect(port[, host][, connectListener])** 创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。 |
| 5 | **net.createConnection(port[, host][, connectListener])** 创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。 |
| 6 | **net.connect(path[, connectListener])** 创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件上。返回 'net.Socket'。 |
| 7 | **net.createConnection(path[, connectListener])** 创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。 |
| 8 | **net.isIP(input)** 检测输入的是否为 IP 地址。 IPV4 返回 4， IPV6 返回 6，其他情况返回 0。 |
| 9 | **net.isIPv4(input)** 如果输入的地址为 IPV4， 返回 true，否则返回 false。 |
| 10 | **net.isIPv6(input)** 如果输入的地址为 IPV6， 返回 true，否则返回 false。 |

net.Server

net.Server通常用于创建一个TCP或本地服务器

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **server.listen(port[, host][, backlog][, callback])** 监听指定端口 port 和 主机 host ac连接。 默认情况下 host 接受任何 IPv4 地址(INADDR\_ANY)的直接连接。端口 port 为 0 时，则会分配一个随机端口。 |
| 2 | **server.listen(path[, callback])** 通过指定 path 的连接，启动一个本地 socket 服务器。 |
| 3 | **server.listen(handle[, callback])** 通过指定句柄连接。 |
| 4 | **server.listen(options[, callback])** options 的属性：端口 port, 主机 host, 和 backlog, 以及可选参数 callback 函数, 他们在一起调用server.listen(port, [host], [backlog], [callback])。还有，参数 path 可以用来指定 UNIX socket。 |
| 5 | **server.close([callback])** 服务器停止接收新的连接，保持现有连接。这是异步函数，当所有连接结束的时候服务器会关闭，并会触发 'close' 事件。 |
| 6 | **server.address()** 操作系统返回绑定的地址，协议族名和服务器端口。 |
| 7 | **server.unref()** 如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。 |
| 8 | **server.ref()** 与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。 |
| 9 | **server.getConnections(callback)** 异步获取服务器当前活跃连接的数量。当 socket 发送给子进程后才有效；回调函数有 2 个参数 err 和 count。 |

事件

|  |  |
| --- | --- |
| **序号** | **事件 & 描述** |
| 1 | **listening** 当服务器调用 server.listen 绑定后会触发。 |
| 2 | **connection** 当新连接创建后会被触发。socket 是 net.Socket实例。 |
| 3 | **close** 服务器关闭时会触发。注意，如果存在连接，这个事件不会被触发直到所有的连接关闭。 |
| 4 | **error** 发生错误时触发。'close' 事件将被下列事件直接调用。 |

net.Socket

net.Socket对象是TCP或UNIX Socked的抽象。 net.Socket实例实现了一个双工流接口， 他们可以在用户创建客户端(使用 connect())时使用， 或者由Node.js创建它们， 并通过connection服务器事件传递给用户。

事件

|  |  |
| --- | --- |
| **序号** | **事件 & 描述** |
| 1 | **lookup** 在解析域名后，但在连接前，触发这个事件。对 UNIX sokcet 不适用。 |
| 2 | **connect** 成功建立 socket 连接时触发。 |
| 3 | **data** 当接收到数据时触发。 |
| 4 | **end** 当 socket 另一端发送 FIN 包时，触发该事件。 |
| 5 | **timeout** 当 socket 空闲超时时触发，仅是表明 socket 已经空闲。用户必须手动关闭连接。 |
| 6 | **drain** 当写缓存为空得时候触发。可用来控制上传。 |
| 7 | **error** 错误发生时触发。 |
| 8 | **close** 当 socket 完全关闭时触发。参数 had\_error 是布尔值，它表示是否因为传输错误导致 socket 关闭。 |

属性

|  |  |
| --- | --- |
| **序号** | **属性 & 描述** |
| 1 | **socket.bufferSize** 该属性显示了要写入缓冲区的字节数。 |
| 2 | **socket.remoteAddress** 远程的 IP 地址字符串，例如：'74.125.127.100' or '2001:4860:a005::68'。 |
| 3 | **socket.remoteFamily** 远程IP协议族字符串，比如 'IPv4' or 'IPv6'。 |
| 4 | **socket.remotePort** 远程端口，数字表示，例如：80 or 21。 |
| 5 | **socket.localAddress** 网络连接绑定的本地接口 远程客户端正在连接的本地 IP 地址，字符串表示。例如，如果你在监听'0.0.0.0'而客户端连接在'192.168.1.1'，这个值就会是 '192.168.1.1'。 |
| 6 | **socket.localPort** 本地端口地址，数字表示。例如：80 or 21。 |
| 7 | **socket.bytesRead** 接收到得字节数。 |
| 8 | **socket.bytesWritten** 发送的字节数。 |

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **new net.Socket([options])** 构造一个新的 socket 对象。 |
| 2 | **socket.connect(port[, host][, connectListener])** 指定端口 port 和 主机 host，创建 socket 连接 。参数 host 默认为 localhost。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。 |
| 3 | **socket.connect(path[, connectListener])** 打开指定路径的 unix socket。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。 |
| 4 | **socket.setEncoding([encoding])** 设置编码 |
| 5 | **socket.write(data[, encoding][, callback])** 在 socket 上发送数据。第二个参数指定了字符串的编码，默认是 UTF8 编码。 |
| 6 | **socket.end([data][, encoding])** 半关闭 socket。例如，它发送一个 FIN 包。可能服务器仍在发送数据。 |
| 7 | **socket.destroy()** 确保没有 I/O 活动在这个套接字上。只有在错误发生情况下才需要。（处理错误等等）。 |
| 8 | **socket.pause()** 暂停读取数据。就是说，不会再触发 data 事件。对于控制上传非常有用。 |
| 9 | **socket.resume()** 调用 pause() 后想恢复读取数据。 |
| 10 | **socket.setTimeout(timeout[, callback])** socket 闲置时间超过 timeout 毫秒后 ，将 socket 设置为超时。 |
| 11 | **socket.setNoDelay([noDelay])** 禁用纳格（Nagle）算法。默认情况下 TCP 连接使用纳格算法，在发送前他们会缓冲数据。将 noDelay 设置为 true 将会在调用 socket.write() 时立即发送数据。noDelay 默认值为 true。 |
| 12 | **socket.setKeepAlive([enable][, initialDelay])** 禁用/启用长连接功能，并在发送第一个在闲置 socket 上的长连接 probe 之前，可选地设定初始延时。默认为 false。 设定 initialDelay （毫秒），来设定收到的最后一个数据包和第一个长连接probe之间的延时。将 initialDelay 设为0，将会保留默认（或者之前）的值。默认值为0. |
| 13 | **socket.address()** 操作系统返回绑定的地址，协议族名和服务器端口。返回的对象有 3 个属性，比如{ port: 12346, family: 'IPv4', address: '127.0.0.1' }。 |
| 14 | **socket.unref()** 如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。如果服务器已被 unref，则再次调用 unref 并不会产生影响。 |
| 15 | **socket.ref()** 与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。 |

实例

server.js

var net = require('net');

var server = net.createServer(function(connection) {

console.log('client connected');

connection.on('end', function() {

console.log('客户端关闭连接');

});

connection.write('Hello World!\r\n');

connection.pipe(connection);

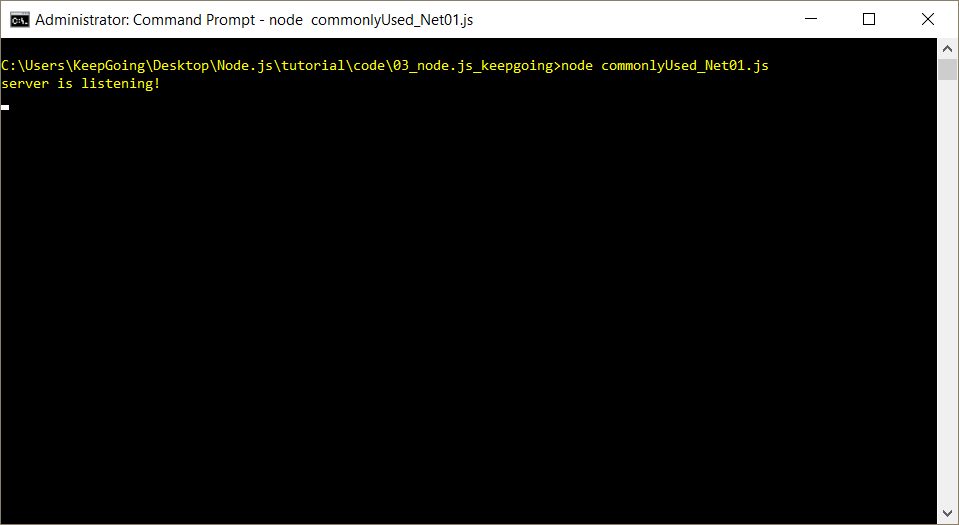
});

server.listen(8080, function() {

console.log('server is listening');

});

执行以上服务端代码



client.js

var net = require('net');

var client = net.connect({port: 8080}, function() {

console.log('连接到服务器！');

});

client.on('data', function(data) {

console.log(data.toString());

client.end();

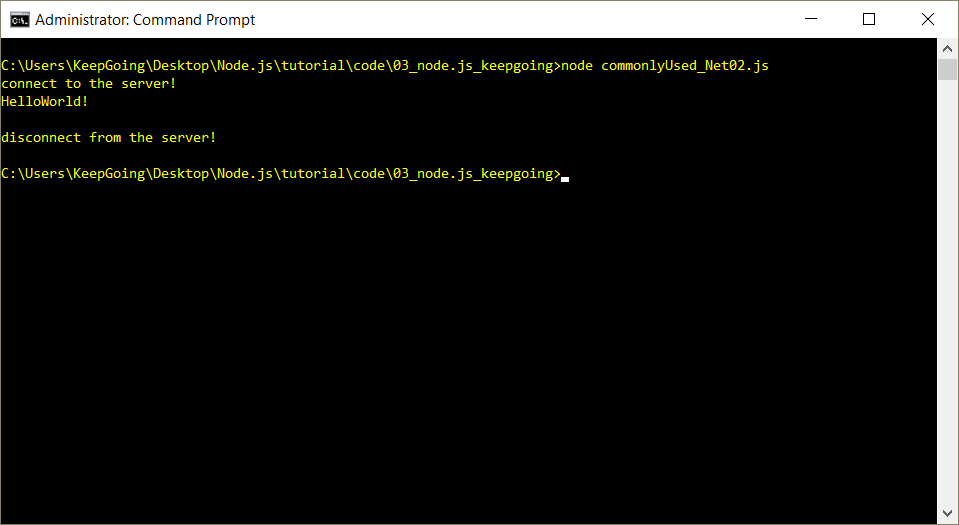
});

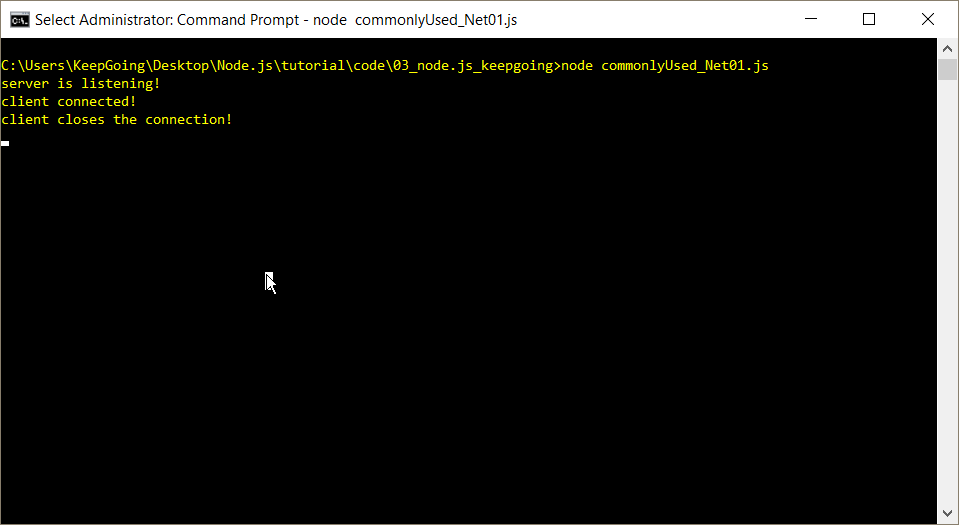
client.on('end', function() {

console.log('断开与服务器的连接');

});

执行以上客户端代码





## DNS模块

var dns = require("dns")

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **dns.lookup(hostname[, options], callback)** 将域名（比如 'runoob.com'）解析为第一条找到的记录 A （IPV4）或 AAAA(IPV6)。参数 options可以是一个对象或整数。如果没有提供 options，IP v4 和 v6 地址都可以。如果 options 是整数，则必须是 4 或 6。 |
| 2 | **dns.lookupService(address, port, callback)** 使用 getnameinfo 解析传入的地址和端口为域名和服务。 |
| 3 | **dns.resolve(hostname[, rrtype], callback)** 将一个域名（如 'runoob.com'）解析为一个 rrtype 指定记录类型的数组。 |
| 4 | **dns.resolve4(hostname, callback)** 和 dns.resolve() 类似, 仅能查询 IPv4 (A 记录）。 addresses IPv4 地址数组 (比如，['74.125.79.104', '74.125.79.105', '74.125.79.106']）。 |
| 5 | **dns.resolve6(hostname, callback)** 和 dns.resolve4() 类似， 仅能查询 IPv6( AAAA 查询） |
| 6 | **dns.resolveMx(hostname, callback)** 和 dns.resolve() 类似, 仅能查询邮件交换(MX 记录)。 |
| 7 | **dns.resolveTxt(hostname, callback)** 和 dns.resolve() 类似, 仅能进行文本查询 (TXT 记录）。 addresses 是 2-d 文本记录数组。(比如，[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]）。 每个子数组包含一条记录的 TXT 块。根据使用情况可以连接在一起，也可单独使用。 |
| 8 | **dns.resolveSrv(hostname, callback)** 和 dns.resolve() 类似, 仅能进行服务记录查询 (SRV 记录）。 addresses 是 hostname可用的 SRV 记录数组。 SRV 记录属性有优先级（priority），权重（weight）, 端口（port）, 和名字（name） (比如，[{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]）。 |
| 9 | **dns.resolveSoa(hostname, callback)** 和 dns.resolve() 类似, 仅能查询权威记录(SOA 记录）。 |
| 10 | **dns.resolveNs(hostname, callback)** 和 dns.resolve() 类似, 仅能进行域名服务器记录查询(NS 记录）。 addresses 是域名服务器记录数组（hostname 可以使用） (比如, ['ns1.example.com', 'ns2.example.com']）。 |
| 11 | **dns.resolveCname(hostname, callback)** 和 dns.resolve() 类似, 仅能进行别名记录查询 (CNAME记录)。addresses 是对 hostname 可用的别名记录数组 (比如，, ['bar.example.com']）。 |
| 12 | **dns.reverse(ip, callback)** 反向解析 IP 地址，指向该 IP 地址的域名数组。 |
| 13 | **dns.getServers()** 返回一个用于当前解析的 IP 地址数组的字符串。 |
| 14 | **dns.setServers(servers)** 指定一组 IP 地址作为解析服务器。 |

rrtypes

以下列出了dns.resolve()方法中有效的rrtypes值

* “A” IPV4地址， 默认。
* “AAAA” IPV6地址。
* “MX” 邮件交换记录。
* “TXT” text记录。
* “SRV” SRV记录。
* “PTR” 用来反向IP查找。
* “NS”域名服务器记录。
* “CNAME” 别名记录。
* “SOA” 授权记录的初始值。

错误码

每次DNS查询都可能返回一下错误码：

* dns.NODATA: 无数据响应。
* dns.FORMERR: 查询格式错误。
* dns.SERVFAIL: 常规失败。
* dns.NOTFOUND: 没有找到域名。
* dns.NOTIMP: 未实现请求的操作。
* dns.REFUSED: 拒绝查询。
* dns.BADQUERY: 查询格式错误。
* dns.BADNAME: 域名格式错误。
* dns.BADFAMILY: 地址协议不支持。
* dns.BADRESP: 回复格式错误。
* dns.CONNREFUSED: 无法连接到 DNS 服务器。
* dns.TIMEOUT: 连接 DNS 服务器超时。
* dns.EOF: 文件末端。
* dns.FILE: 读文件错误。
* dns.NOMEM: 内存溢出。
* dns.DESTRUCTION: 通道被摧毁。
* dns.BADSTR: 字符串格式错误。
* dns.BADFLAGS: 非法标识符。
* dns.NONAME: 所给主机不是数字。
* dns.BADHINTS: 非法HINTS标识符。
* dns.NOTINITIALIZED: c c-ares 库尚未初始化。
* dns.LOADIPHLPAPI: 加载 iphlpapi.dll 出错。
* dns.ADDRGETNETWORKPARAMS: 无法找到 GetNetworkParams 函数。
* dns.CANCELLED: 取消 DNS 查询。

实例

var dns = require('dns');

dns.lookup('www.github.com', function onLookup(err, address, family) {

console.log('ip 地址:', address);

dns.reverse(address, function (err, hostnames) {

if (err) {

console.log(err.stack);

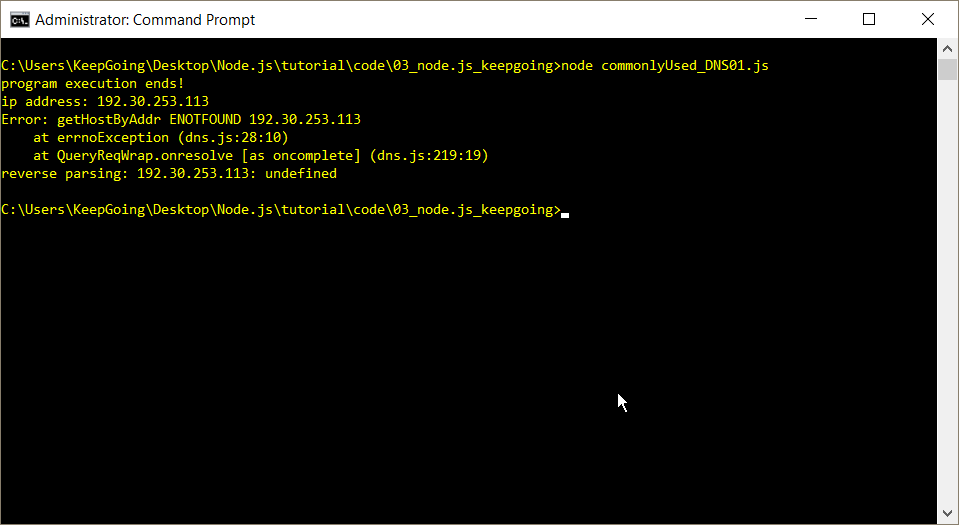
}

console.log('反向解析 ' + address + ': ' + JSON.stringify(hostnames));

});

});

执行以上代码， 结果如下所示



## Domain模块

Node.js Domain(域) 简化异步代码的异常处理， 可以捕捉处理try catch无法捕捉的异常。

var domain = require("domain")

domain模块， 把处理多个不同的I/O操作作为一个组， 注册事件和回调到domain， 当发生一个错误时间或抛出一个错误时， domain对象会被通知， 不会丢失上下文环境， 也不导致程序错误立即退出， 与process.on(“uncaughtException”)不同。Domain模块可以分为隐式绑定和显示绑定：

* 隐式绑定： 把在domain上下文中定义的变量， 自动绑定到domain对象。
* 显示绑定：把不是domain上下文中定义的变量， 以代码的方式绑定到domain对象。

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **domain.run(function)** 在域的上下文运行提供的函数，隐式的绑定了所有的事件分发器，计时器和底层请求。 |
| 2 | **domain.add(emitter)** 显式的增加事件 |
| 3 | **domain.remove(emitter)** 删除事件。 |
| 4 | **domain.bind(callback)** 返回的函数是一个对于所提供的回调函数的包装函数。当调用这个返回的函数被时，所有被抛出的错误都会被导向到这个域的 error 事件。 |
| 5 | **domain.intercept(callback)** 和 domain.bind(callback) 类似。除了捕捉被抛出的错误外，它还会拦截 Error 对象作为参数传递到这个函数。 |
| 6 | **domain.enter()** 进入一个异步调用的上下文，绑定到domain。 |
| 7 | **domain.exit()** 退出当前的domain，切换到不同的链的异步调用的上下文中。对应domain.enter()。 |
| 8 | **domain.dispose()** 释放一个domain对象，让node进程回收这部分资源。 |
| 9 | **domain.create()** 返回一个domain对象。 |

属性

|  |  |
| --- | --- |
| **序号** | **属性 & 描述** |
| 1 | **domain.members** 已加入domain对象的域定时器和事件发射器的数组。 |

实例

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

var domain = require("domain");

var emitter1 = new EventEmitter();

// 创建域

var domain1 = domain.create();

domain1.on('error', function(err){

console.log("domain1 处理这个错误 ("+err.message+")");

});

// 显式绑定

domain1.add(emitter1);

emitter1.on('error',function(err){

console.log("监听器处理此错误 ("+err.message+")");

});

emitter1.emit('error',new Error('通过监听器来处理'));

emitter1.removeAllListeners('error');

emitter1.emit('error',new Error('通过 domain1 处理'));

var domain2 = domain.create();

domain2.on('error', function(err){

console.log("domain2 处理这个错误 ("+err.message+")");

});

// 隐式绑定

domain2.run(function(){

var emitter2 = new EventEmitter();

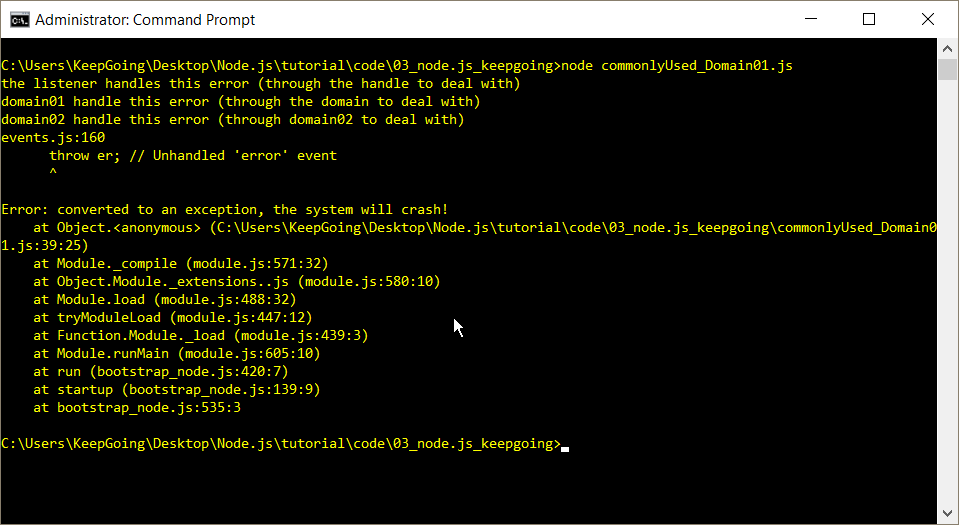
emitter2.emit('error',new Error('通过 domain2 处理'));

});

domain1.remove(emitter1);

emitter1.emit('error', new Error('转换为异常，系统将崩溃!'));

执行以上代码， 结果如下所示：



# Web模块

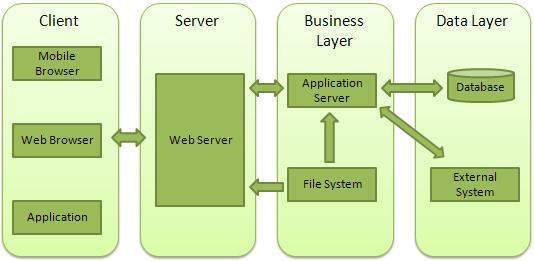
什么是Web服务器

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

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

目前最主流的三个Web服务器是Apache， Nginx， IIS。

Web应用架构



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

## 创建Web服务器

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

实例

server.js

var http = require('http');

var fs = require('fs');

var url = require('url');

// 创建服务器

http.createServer( function (request, response) {

// 解析请求，包括文件名

var pathname = url.parse(request.url).pathname;

// 输出请求的文件名

console.log("Request for " + pathname + " received.");

// 从文件系统中读取请求的文件内容

fs.readFile(pathname.substr(1), function (err, data) {

if (err) {

console.log(err);

// HTTP 状态码: 404 : NOT FOUND

// Content Type: text/plain

response.writeHead(404, {'Content-Type': 'text/html'});

}else{

// HTTP 状态码: 200 : OK

// Content Type: text/plain

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

// 响应文件内容

response.write(data.toString());

}

// 发送响应数据

response.end();

});

}).listen(8888);

// 控制台会输出以下信息

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

index.html

<html>

<head>

<title>Sample Page</title>

</head>

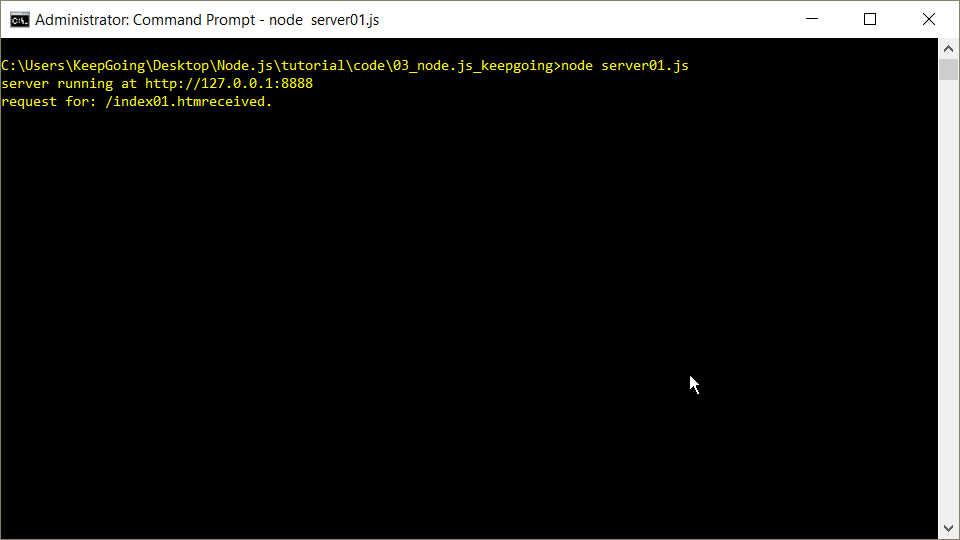
<body>

Hello World!

</body>

</html>

执行server.js文件， 并在浏览器中访问<http://127.0.0.1:8888/index.html>



## 创建Web客户端

Node.js创建Web客户端需要引入http模块

client.js

var http = require('http');

// 用于请求的选项

var options = {

host: 'localhost',

port: '8888',

path: '/index.htm'

};

// 处理响应的回调函数

var callback = function(response){

// 不断更新数据

var body = '';

response.on('data', function(data) {

body += data;

});

response.on('end', function() {

// 数据接收完成

console.log(body);

});

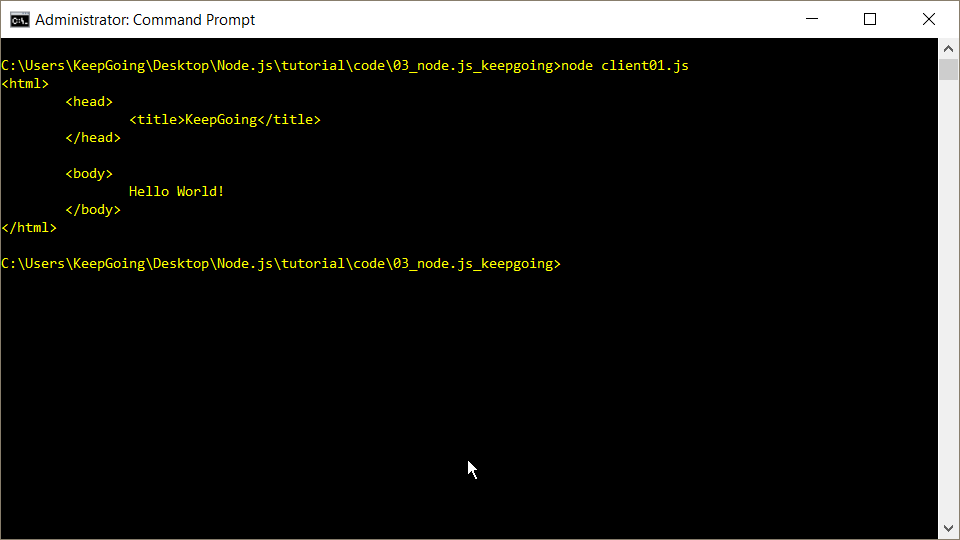
}

// 向服务端发送请求

var req = http.request(options, callback);

req.end();

执行以上代码， 运行结果为



# Express框架

简介

Express是一个简介而灵活的Node.js Web应用框架， 提供了一系列强大的特性帮助你创建各种Web应用， 和丰富的HTTP工具。

使用Express可以快速的搭建一个完整的功能网站。

Express框架核心特性：

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

## 安装Express

安装Express并将其保存到依赖列表中

npm install express –save

如需要使用express命令， 则需要如下安装

npm install express -g express-generator

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

* body-parser： Node.js中间件， 用于处理JSON， Raw， Text和URL编码的数据。
* cookie-parser： 这就是一个解析Cookie的工具。 通过request.cookies可以去的传过来的cookie， 并把它们转成对象。
* multer： Node.js中间件， 用于处理enctype = “multipart/form-data”（设置表单里的MIME编码）的表单数据。

npm install body-parser --save

npm install cookie-parser --save

npm install multer --save

Express框架实例

我们使用Express框架来输出“HelloWorld”

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

实例

var express = require('express');

var app = express();

app.get('/', function (req, res) {

res.send('Hello World');

})

var server = app.listen(8888, function () {

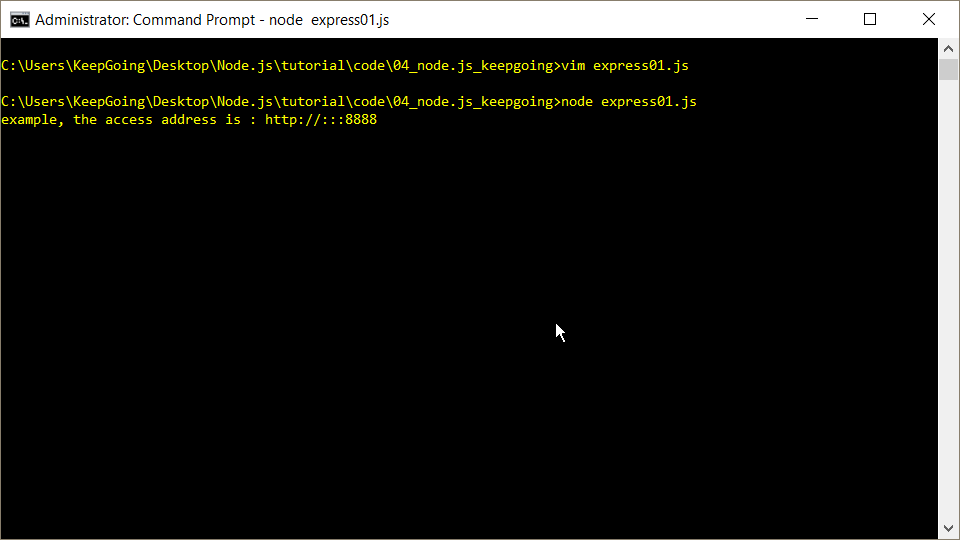
var host = server.address().address

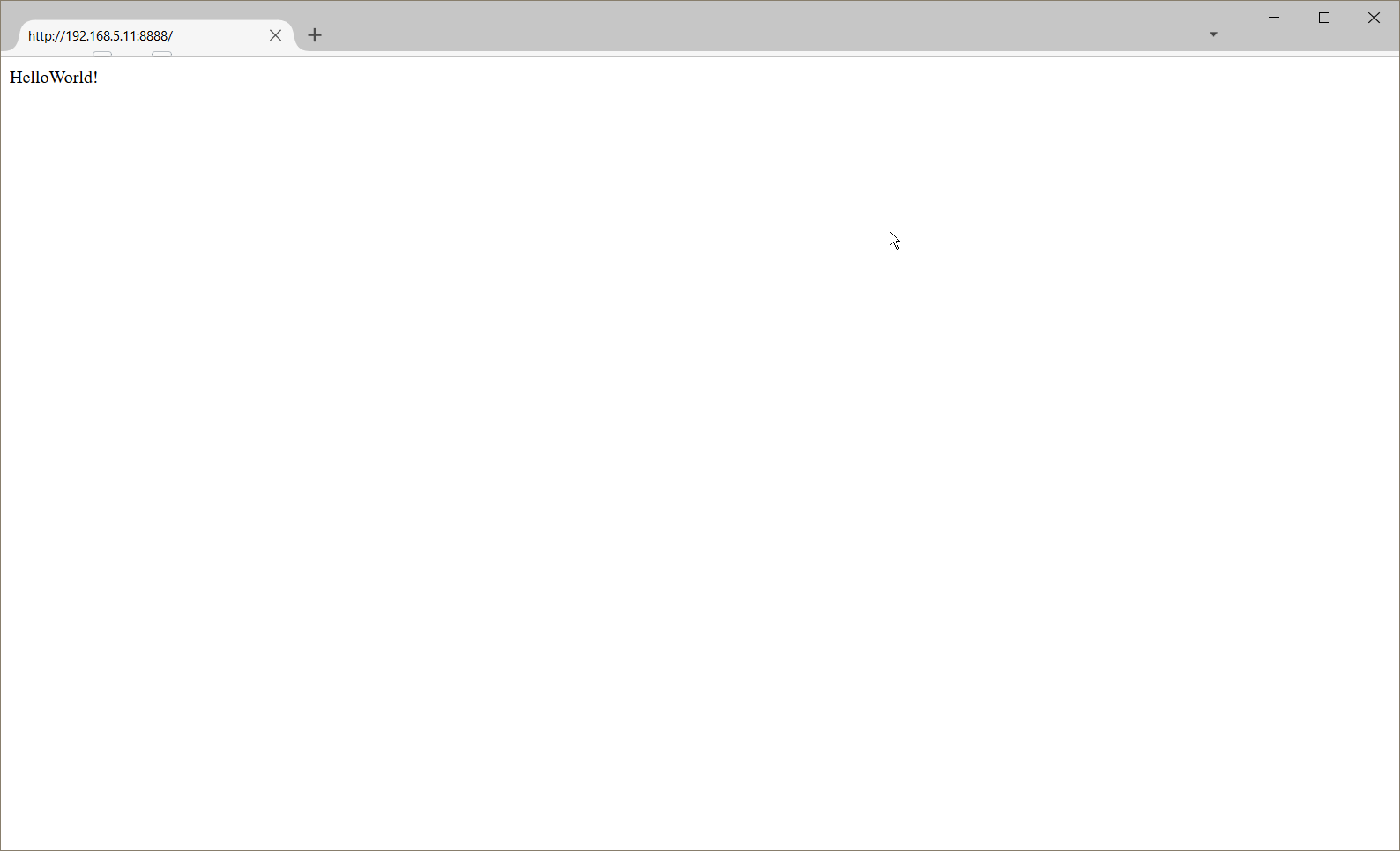
var port = server.address().port

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

})

执行以上代码， 结果为：





## 请求和响应

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

app.get('/', function (req, res) {

// --

})

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。 常见的属性有：

1. request.app: 当callback为外部文件时， 用request访问express的实例。
2. request.baseUrl： 获取路由当前安装的URL路径。
3. request.body / reques.cookies： 获得[请求主体] / Cookies.
4. request.fresh / request.stale: 判断请求是否还[新鲜]。
5. request.hostname / request.ip: 获取主机名和IP地址。
6. request.ooriginalUrl: 获取原始请求URL。
7. request.params: 获取路由的parameters。
8. request.path: 获取请求路径。
9. request.protocol: 获取协议类型。
10. request.query: 获取URL的查询参数串。
11. request.route: 获取当前匹配的路由。
12. request.subdomains: 获取子域名。
13. request.accpets(): 检查情的的Accept头的请求类型。
14. request.acceptesCharsets / request.acceptesEncodings / request.acceptsLanguages
15. request.get(): 获取指定的HTTP请求头。
16. request.is(): 判断请求头Content-Type的MIME类型。

Response对象： response对象表示HTTP响应， 即在接受到请求时向客户端发送的HTTP响应数据。 常见的属性有：

1. response.app: 同request.app一样。
2. response.append();追加指定HTTP头。
3. response.set(): 在response.append()后将重置之前设置的头。
4. response.cookie(name, value[option]):设置Cookie。
5. opition:domain/expires/httpOnly/maxAge/path/secure/signed
6. response.clearCookie(); 清除Cookie。
7. response.download();传送指定路径的文件。
8. response.get();返回指定的HTTP头。
9. response.json();传送JSON响应。
10. response.jsonp();传送JSON响应。
11. response.location();只设置相应的Location HTTP头， 不设置状态吗或者close response。
12. response.redirect();设置响应的Location HTTP头， 而且设置状态码302.
13. response.send();传送HTTP响应。
14. response.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。
15. response.set();设置HTTP头， 传入object可以一次设置多个头。
16. response.status();设置HTTP状态码。
17. response.type();设置Content-Type的MIME类型。

## 路由

路又决定了由谁（指定脚本）去响应客户端请求。在HTTP请求中， 我们可以通过路由提取出请求的URL以及GET/POST参数。接下来我们扩展Hello World， 添加一些功能来处理更多类型的HTTP请求。

实例

var express = require('express');

var app = express();

// 主页输出 "Hello World"

app.get('/', function (req, res) {

console.log("主页 GET 请求");

res.send('Hello GET');

})

// POST 请求

app.post('/', function (req, res) {

console.log("主页 POST 请求");

res.send('Hello POST');

})

// /del\_user 页面响应

app.get('/del\_user', function (req, res) {

console.log("/del\_user 响应 DELETE 请求");

res.send('删除页面');

})

// /list\_user 页面 GET 请求

app.get('/list\_user', function (req, res) {

console.log("/list\_user GET 请求");

res.send('用户列表页面');

})

// 对页面 abcd, abxcd, ab123cd, 等响应 GET 请求

app.get('/ab\*cd', function(req, res) {

console.log("/ab\*cd GET 请求");

res.send('正则匹配');

})

var server = app.listen(8888, function () {

var host = server.address().address

var port = server.address().port

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

})

## 静态文件

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

可以使用express.static中间件来设静态稳态路径。 例如， 如果你将图片， CSS， JavaScript文件放在public目录下，

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

public/images目录下放些图片

实例

var express = require('express');

var app = express();

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

app.get('/', function (req, res) {

res.send('Hello World');

})

var server = app.listen(8888, function () {

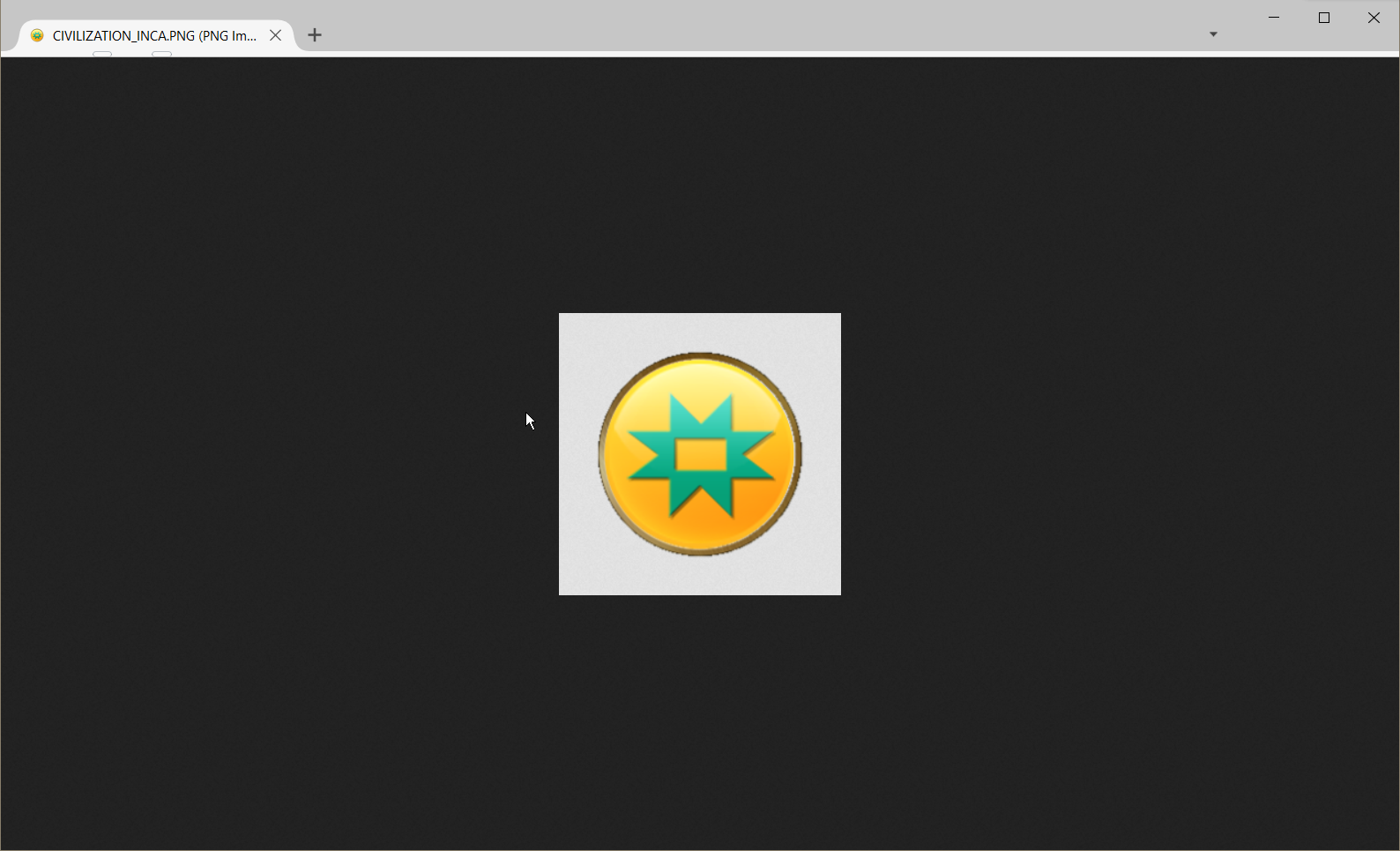
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 运行结果如下：



## GET方法

在表单中通过GET方法提交两个参数， 我们可以使用server.js文件内的process\_get路由器来处理输出

index.html

<html>

<body>

<form action="http://127.0.0.1:8081/process\_get" method="GET">

First Name: <input type="text" name="first\_name"> <br>

Last Name: <input type="text" name="last\_name">

<input type="submit" value="Submit">

</form>

</body>

</html>

server.js

var express = require('express');

var app = express();

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

app.get('/index.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.get('/process\_get', function (req, res) {

// 输出 JSON 格式

response = {

first\_name:req.query.first\_name,

last\_name:req.query.last\_name

};

console.log(response);

res.end(JSON.stringify(response));

})

var server = app.listen(8081, function () {

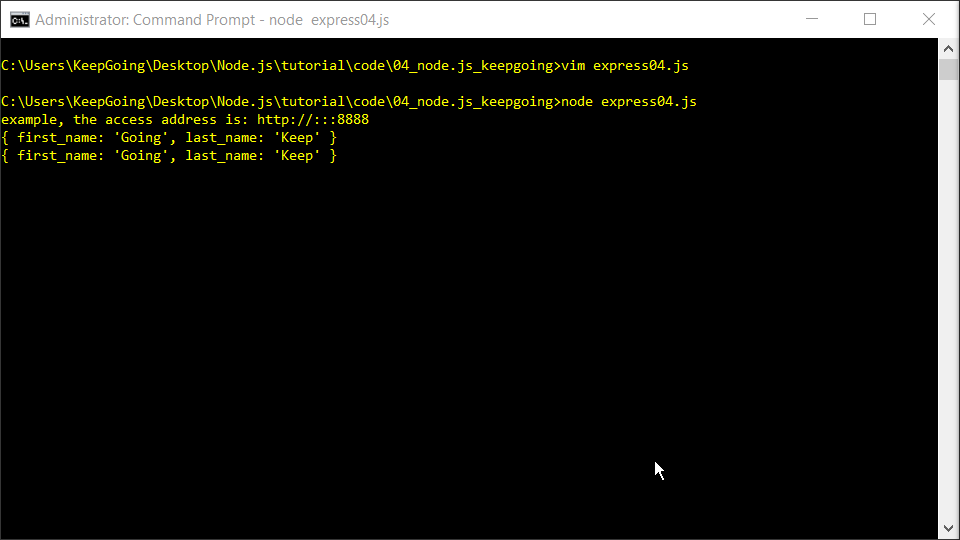
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 运行结果为



## POST方法

<html>

<body>

<form action="http://127.0.0.1:8081/process\_post" method="POST">

First Name: <input type="text" name="first\_name"> <br>

Last Name: <input type="text" name="last\_name">

<input type="submit" value="Submit">

</form>

</body>

</html>

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

// 创建 application/x-www-form-urlencoded 编码解析

var urlencodedParser = bodyParser.urlencoded({ extended: false })

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

app.get('/index.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.post('/process\_post', urlencodedParser, function (req, res) {

// 输出 JSON 格式

response = {

first\_name:req.body.first\_name,

last\_name:req.body.last\_name

};

console.log(response);

res.end(JSON.stringify(response));

})

var server = app.listen(8081, function () {

var host = server.address().address

var port = server.address().port

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

})

## 文件上传

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

<html>

<head>

<title>文件上传表单</title>

</head>

<body>

<h3>文件上传：</h3>

选择一个文件上传: <br />

<form action="/file\_upload" method="post" enctype="multipart/form-data">

<input type="file" name="image" size="50" />

<br />

<input type="submit" value="上传文件" />

</form>

</body>

</html>

var express = require('express');

var app = express();

var fs = require("fs");

var bodyParser = require('body-parser');

var multer = require('multer');

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

app.use(bodyParser.urlencoded({ extended: false }));

app.use(multer({ dest: '/tmp/'}).array('image'));

app.get('/index.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.post('/file\_upload', function (req, res) {

console.log(req.files[0]); // 上传的文件信息

var des\_file = \_\_dirname + "/" + req.files[0].originalname;

fs.readFile( req.files[0].path, function (err, data) {

fs.writeFile(des\_file, data, function (err) {

if( err ){

console.log( err );

}else{

response = {

message:'File uploaded successfully',

filename:req.files[0].originalname

};

}

console.log( response );

res.end( JSON.stringify( response ) );

});

});

})

var server = app.listen(8081, function () {

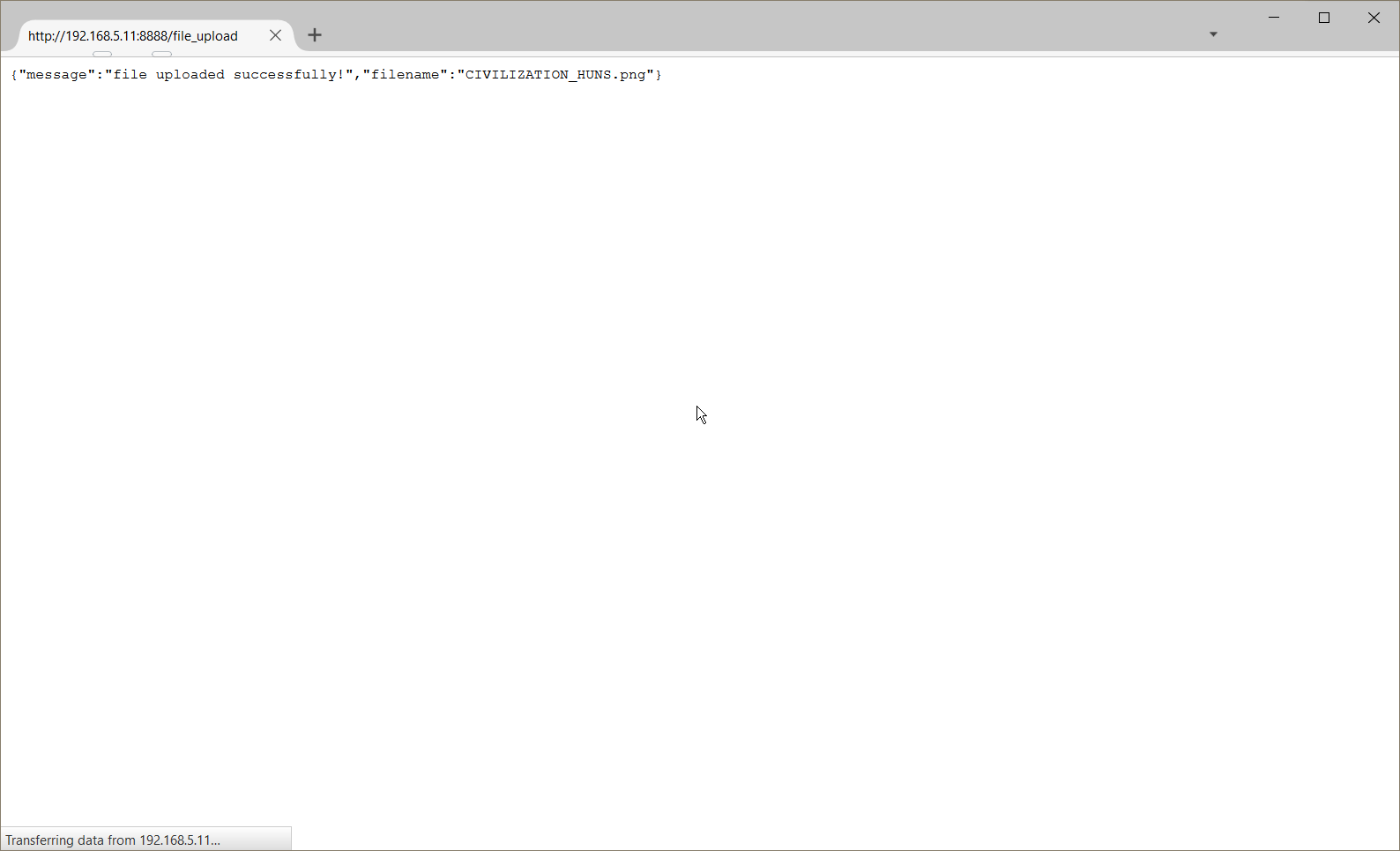
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 运行结果如下：



## Cookie管理

我们可以使用中间件向Node.js服务器发送cookie信息

var express = require('express')

var cookieParser = require('cookie-parser')

var app = express()

app.use(cookieParser())

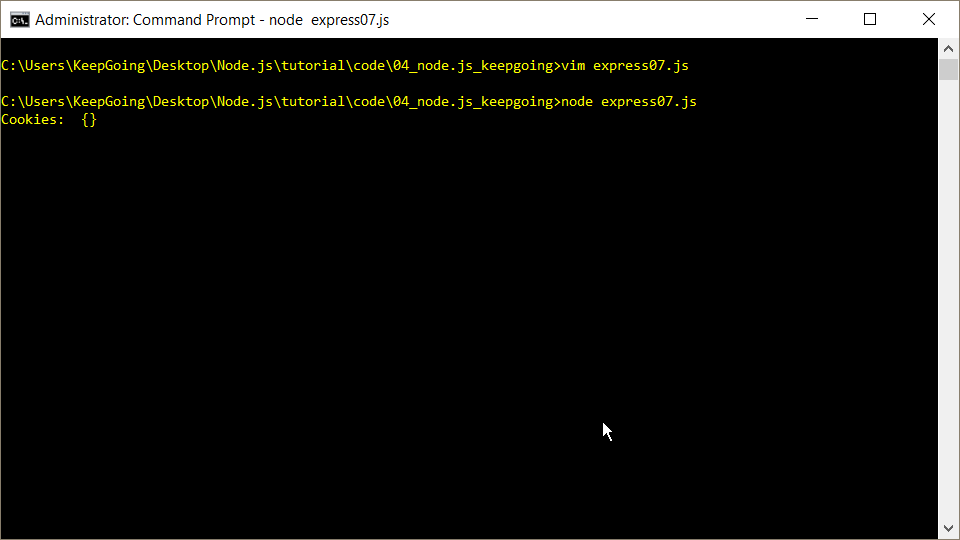
app.get('/', function(req, res) {

console.log("Cookies: ", req.cookies)

})

app.listen(8081)

运行以上代码， 执行结果如下。



# RESTful API

什么是REST？

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

表述性状态传递是一种架构约束条件和原则， 满足这些约束条件和原则的应用程序设计就是RESTful。 需要注意的是， REST是设计风格而不是标准， REST通常基于使用HTTP， URI， 和XML（标准通用标记语言下的一个子集以及HTML（标准通用标记语言下的一个应用）这些现有的广泛流行的协议和标准。 REST通常使用JSON数据格式。

## HTTP方法

以下为REST基本架构的四个方法：

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

## RESTful Web Services

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

基于REST架构的Web Services即是RESTful。

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

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

## 创建RESTful

创建一个JSON数据资源文件users.json

{

"user1" : {

"name" : "mahesh",

"password" : "password1",

"profession" : "teacher",

"id": 1

},

"user2" : {

"name" : "suresh",

"password" : "password2",

"profession" : "librarian",

"id": 2

},

"user3" : {

"name" : "ramesh",

"password" : "password3",

"profession" : "clerk",

"id": 3

}

}

基于以上的数据， 我们穿见一下RESTful API

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

### 获取用户列表：

创建RESTful API listUsers

var express = require('express');

var app = express();

var fs = require("fs");

app.get('/listUsers', function (req, res) {

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

console.log( data );

res.end( data );

});

})

var server = app.listen(8081, function () {

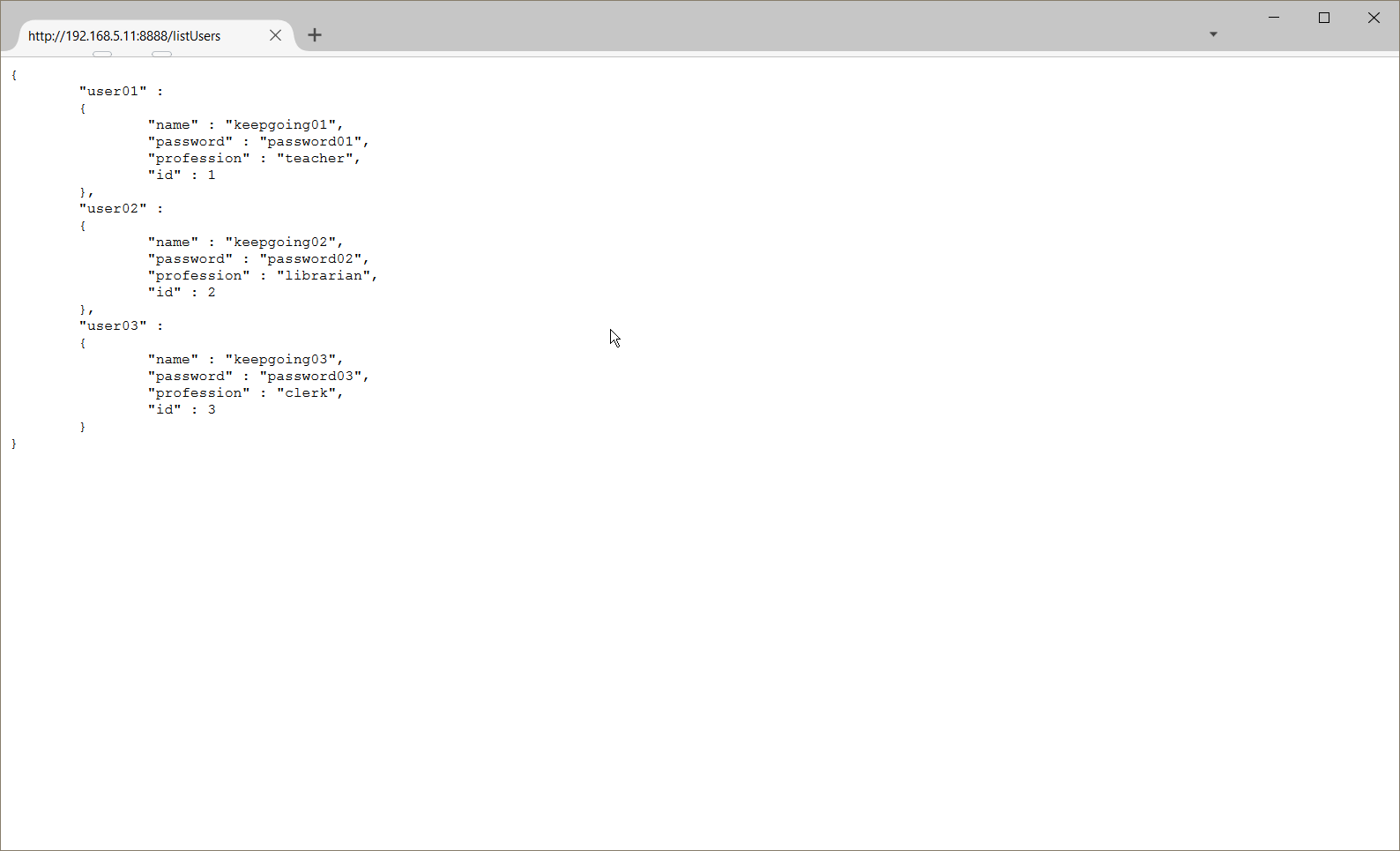
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 结果如下所示



### 添加用户

RESTful API addUser

var express = require('express');

var app = express();

var fs = require("fs");

//添加的新用户数据

var user = {

"user4" : {

"name" : "mohit",

"password" : "password4",

"profession" : "teacher",

"id": 4

}

}

app.get('/addUser', function (req, res) {

// 读取已存在的数据

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

data["user4"] = user["user4"];

console.log( data );

res.end( JSON.stringify(data));

});

})

var server = app.listen(8081, function () {

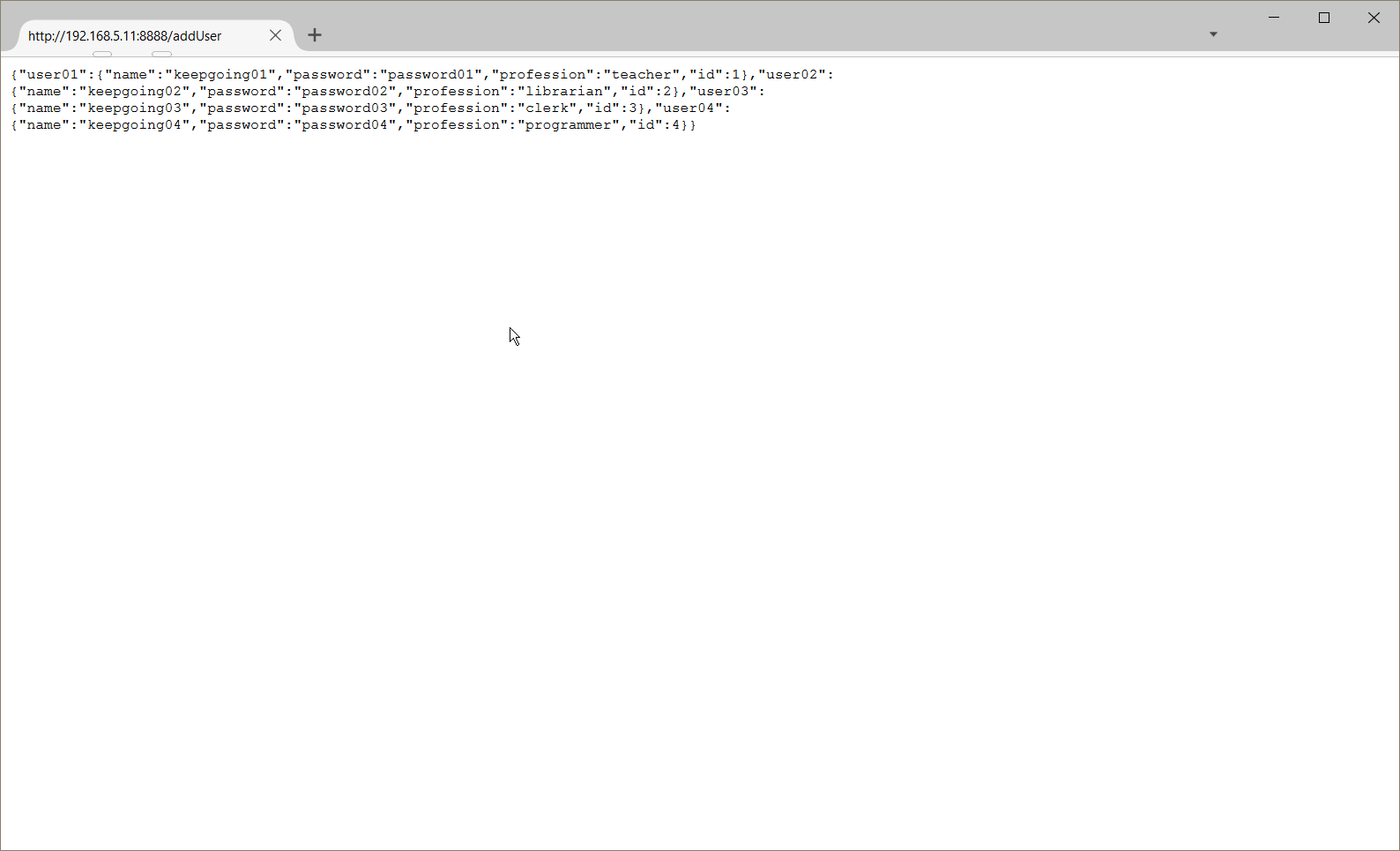
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 结果如下所示：



### 显示用户详情

RESTful API :id

var express = require('express');

var app = express();

var fs = require("fs");

app.get('/:id', function (req, res) {

// 首先我们读取已存在的用户

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

var user = data["user" + req.params.id]

console.log( user );

res.end( JSON.stringify(user));

});

})

var server = app.listen(8081, function () {

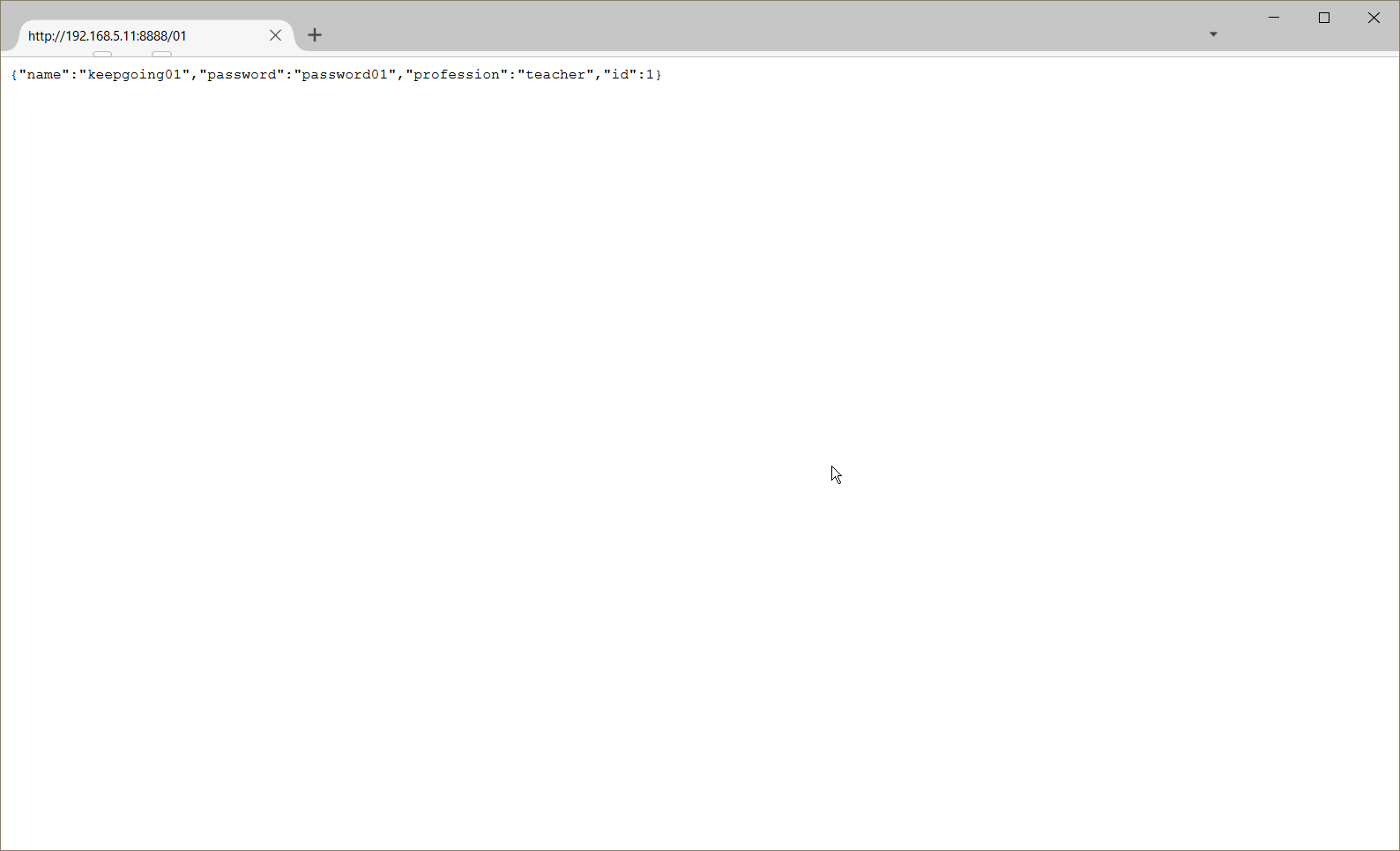
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 运行结果如下所示：



### 删除用户

RESTful API deleteUser

var express = require('express');

var app = express();

var fs = require("fs");

var id = 2;

app.get('/deleteUser', function (req, res) {

// First read existing users.

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

delete data["user" + 2];

console.log( data );

res.end( JSON.stringify(data));

});

})

var server = app.listen(8081, function () {

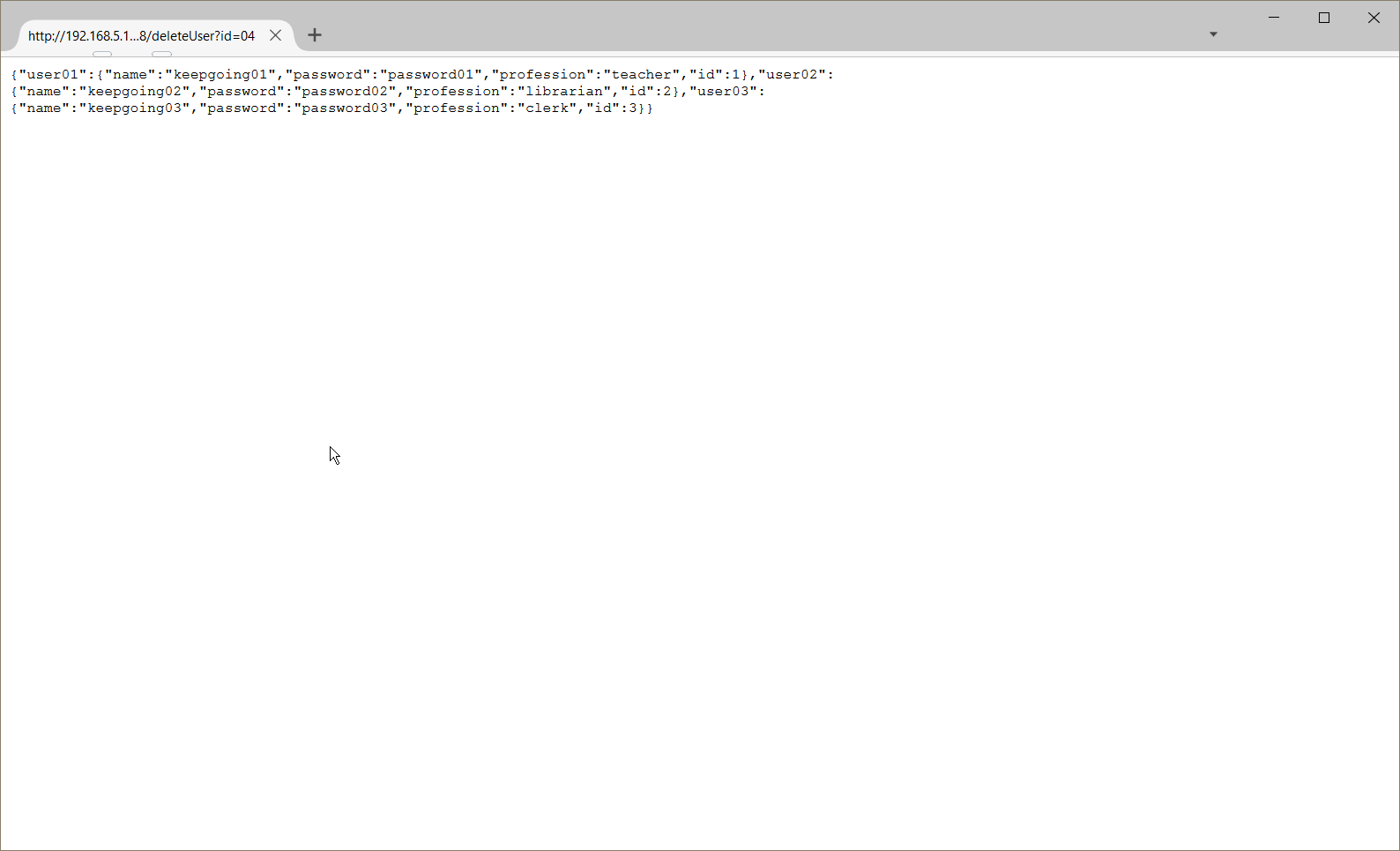
var host = server.address().address

var port = server.address().port

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

})

执行以上代码， 结果如下所示



# 多进程

我们都知道Node.js是以单线程的模式运行的， 但它使用的是事件驱动来处理并发， 这样有助于我们在多核CPU的系统上创建多个子进程， 从而提高性能。 每个子进程总是带有三个流对象： child.stdin, child.stdout和child.stderr。 他们可能会共享父进程的stdio流， 或者也可以是独立的。

Node.js提供了child\_process模块来创建子进程， 方法有：

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

## exec()方法。

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

语法：

child\_process.exec(command[, options], callback)

参数

command： 字符串， 将要运行的命令， 参数使用空格隔开。

options： 对象， 可以是：

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

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

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

实例

support.js

console.log("进程 " + process.argv[2] + " 执行。" );

master.js

const fs = require('fs');

const child\_process = require('child\_process');

for(var i=0; i<3; i++) {

var workerProcess = child\_process.exec('node support.js '+i,

function (error, stdout, stderr) {

if (error) {

console.log(error.stack);

console.log('Error code: '+error.code);

console.log('Signal received: '+error.signal);

}

console.log('stdout: ' + stdout);

console.log('stderr: ' + stderr);

});

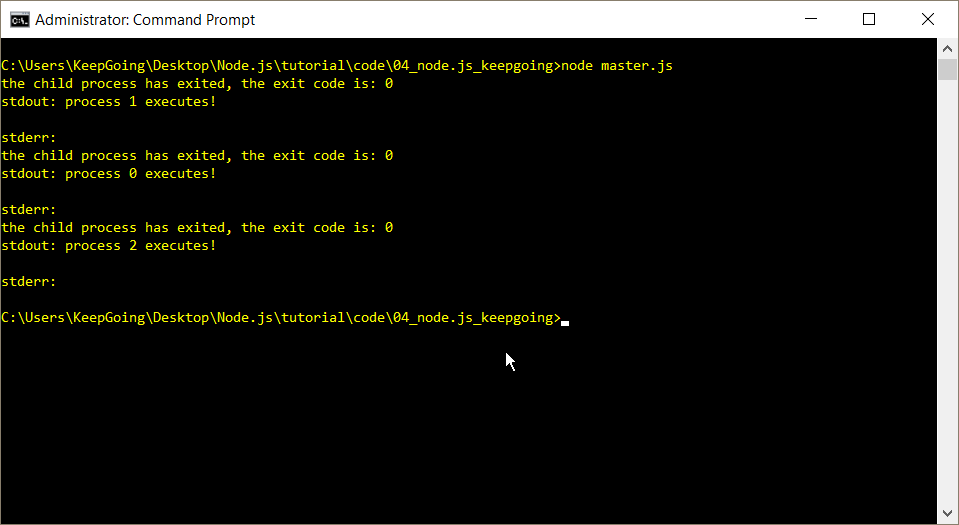
workerProcess.on('exit', function (code) {

console.log('子进程已退出，退出码 '+code);

});

}

执行以上代码， 输出结果为



## spawn()方法

child.process.spawn使用指定的命令行参数创建新进程。

语法

child\_process.spawn(command[, args][, options])

参数

command： 将要运行的命令。

args： Array字符串参数数组。

options Object

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

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

实例

support.js

console.log("进程 " + process.argv[2] + " 执行。" );

master.js

const fs = require('fs');

const child\_process = require('child\_process');

for(var i=0; i<3; i++) {

var workerProcess = child\_process.spawn('node', ['support.js', i]);

workerProcess.stdout.on('data', function (data) {

console.log('stdout: ' + data);

});

workerProcess.stderr.on('data', function (data) {

console.log('stderr: ' + data);

});

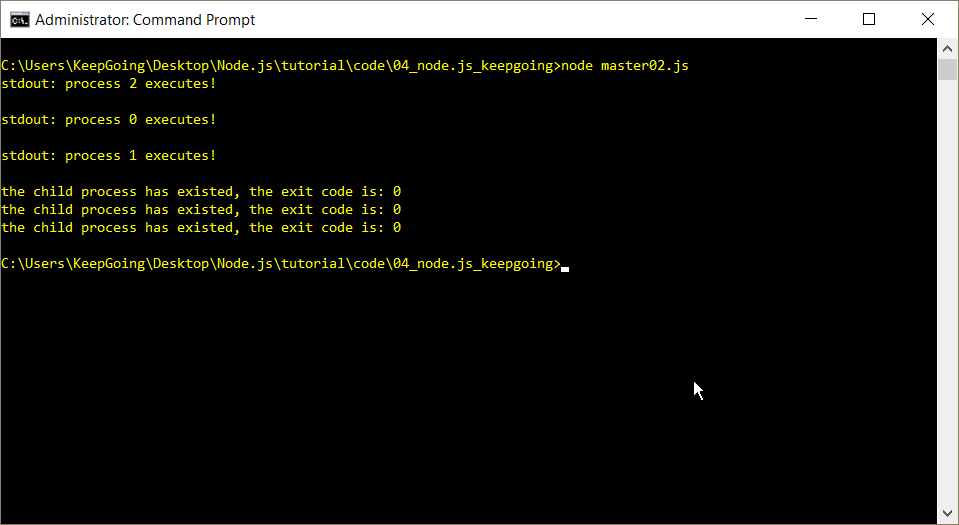
workerProcess.on('close', function (code) {

console.log('子进程已退出，退出码 '+code);

});

}

执行以上代码， 输出结果为



## fork方法

child\_process.fork是spawn()方法的特殊形式， 用于创建进程。

语法

child\_process.fork(modulePath[, args][, options])

参数

modulePath： String， 将要在子进程中运行的模块。

args： Array 字符串参数数组。

options： Object

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

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

实例

master.js

const fs = require('fs');

const child\_process = require('child\_process');

for(var i=0; i<3; i++) {

var worker\_process = child\_process.fork("support.js", [i]);

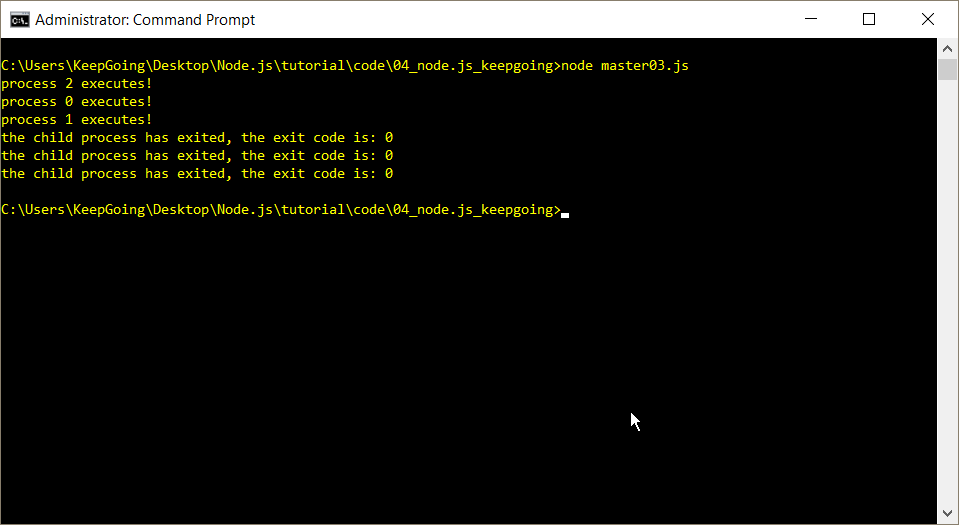
worker\_process.on('close', function (code) {

console.log('子进程已退出，退出码 ' + code);

});

}

执行以上代码， 输出结果为：



# JXcore打包

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

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

## JXcore安装

下载JXcore安装包， 并解压， 在解压的目录提供了jx二进制文件命令。

download link: <https://github.com/jxcore/jxcore-release>

如果安装配置成功， 使用以下命令， 会输出版本号信息：

jx --version

## 包代码

假如我们的Node.js项目中包含了以下几个文件， 其中index.js是主文件。

images, index.htm, index.js, node\_modules, scripts, style

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， 会生成一下两个文件：

* 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， 可以运行在客户端上。

## 载入JX文件

使用jx命令打包项目

jx index.jx 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

node index.js command\_line\_arguments