通过输入 **"npm -v"**来测试是否成功安装,返回NPM版本;

npm install npm –g 升级NPM

npm install 模块名 npm 安装 Node.js 模块语法 加-g表示全局安装

本地安装

1. 将安装包放在 ./node\_modules 下（运行 npm 命令时所在的目录），如果没有 node\_modules 目录，会在当前执行 npm 命令的目录下生成 node\_modules 目录。

2. 可以通过 require() 来引入本地安装的包。

全局安装

1. 将安装包放在 /usr/local 下或者你 node 的安装目录。

2. 可以直接在命令行里使用。

npm ls –g 查看所有全局安装的模块

Package.json 属性说明

name - 包名。

version - 包的版本号。

description - 包的描述。

homepage - 包的官网 url 。

author - 包的作者姓名。

contributors - 包的其他贡献者姓名。

dependencies - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node\_module 目录下。

repository - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。

main - main 字段是一个模块ID，它是一个指向你程序的主要项目。就是说，如果你包的名字叫 express，然后用户安装它，然后require("express")。

keywords - 关键字

npm uninstall express 来卸载 Node.js 模块

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

npm search 模块名 查找 npm update 模块名 更新

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

$ npm init

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

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

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

and exactly what they do.

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

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

Press ^C at any time to quit.

name: (node\_modules) runoob # 模块名

version: (1.0.0)

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

entry point: (index.js)

test command: make test

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

keywords:

author:

license: (ISC)

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

{

"name": "runoob",

"version": "1.0.0",

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

……

}

Is this ok? (yes) yes

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

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

$ npm adduser

Username: mcmohd

Password:

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

接下来我们就用以下命令来发布模块：

$ npm publish

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

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

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

使用npm help <command>可查看某条命令的详细帮助，例如npm help install。

在package.json所在目录下使用npm install . -g可先在本地安装当前命令行程序，可用于发布前的本地测试。

使用npm update <package>可以把当前目录下node\_modules子目录里边的对应模块更新至最新版本。

使用npm update <package> -g可以把全局安装的对应命令行程序更新至最新版。

使用npm cache clear可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。

使用npm unpublish <package>@<version>可以撤销发布自己发布过的某个版本代码。

Node.js 异步编程的直接体现就是回调。

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

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

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

创建 main.js 文件，代码如下所示：

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

// 创建事件处理程序

var connectHandler = function connected() {

console.log('连接成功。');

// 触发 data\_received 事件

eventEmitter.emit('data\_received');

}

// 绑定 connection 事件处理程序

eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data\_received 事件

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

console.log('数据接收成功。');

});

// 触发 connection 事件

eventEmitter.emit('connection');

console.log("程序执行完毕。");

$ node main.js

连接成功。

数据接收成功。

程序执行完毕。

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

创建一个 input.txt ,文件内容如下：

菜鸟教程官网地址：www.runoob.com

创建 main.js 文件，代码如下：

var fs = require("fs");

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

if (err){

console.log(err.stack);

return;

}

console.log(data.toString());

});

console.log("程序执行完毕");

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

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

执行以上代码，执行结果如下：

程序执行完毕

菜鸟教程官网地址：www.runoob.com

接下来我们删除 input.txt 文件，执行结果如下所示：

程序执行完毕

Error: ENOENT, open 'input.txt'

因为文件 input.txt 不存在，所以输出了错误信息。

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。

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

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

//event.js 文件

var events = require('events');

var emitter = new events.EventEmitter();

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

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

});

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

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

});

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

执行以上代码，运行的结果如下：

$ node event.js

listener1 arg1 参数 arg2 参数

listener2 arg1 参数 arg2 参数

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。

实例

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

创建 main.js 文件，代码如下：

var events = require('events');

var eventEmitter = new events.EventEmitter();

// 监听器 #1

var listener1 = function listener1() {

console.log('监听器 listener1 执行。');

}

// 监听器 #2

var listener2 = function listener2() {

console.log('监听器 listener2 执行。');

}

// 绑定 connection 事件，处理函数为 listener1

eventEmitter.addListener('connection', listener1);

// 绑定 connection 事件，处理函数为 listener2

eventEmitter.on('connection', listener2);

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

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

// 处理 connection 事件

eventEmitter.emit('connection');

// 移除监绑定的 listener1 函数

eventEmitter.removeListener('connection', listener1);

console.log("listener1 不再受监听。");

// 触发连接事件

eventEmitter.emit('connection');

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

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

console.log("程序执行完毕。");

以上代码，执行结果如下所示：

$ node main.js

2 个监听器监听连接事件。

监听器 listener1 执行。

监听器 listener2 执行。

listener1 不再受监听。

监听器 listener2 执行。

1 个监听器监听连接事件。

程序执行完毕。

error 事件

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

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

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

var events = require('events');

var emitter = new events.EventEmitter();

emitter.emit('error');

运行时会显示以下错误：

node.js:201

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

^

Error: Uncaught, unspecified 'error' event.

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

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

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

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

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

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

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

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

继承 EventEmitter

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

为什么要这样做呢？原因有两点：

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

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

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

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

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

**创建**。

方法 1

创建长度为 10 字节的 Buffer：

var buf = new Buffer(10);

方法 2

通过给定的数组创建 Buffer：var buf = new Buffer([10, 20, 30, 40, 50]);

方法 3

通过一个字符串来创建 Buffer：var buf = new Buffer("www.runoob.com", "utf-8");

**写入 Node 缓冲区**

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

参数

参数描述如下：

string - 写入缓冲区的字符串。

offset - 缓冲区开始写入的索引值，默认为 0 。

length - 写入的字节数，默认为 buffer.length

encoding - 使用的编码。默认为 'utf8' 。

返回值

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

实例

buf = new Buffer(256);

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

console.log("写入字节数 : "+ len);

执行以上代码，输出结果为：

$node main.js

写入字节数 : 14

**读取 Node 缓冲区数据**

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

参数

参数描述如下：

encoding - 使用的编码。默认为 'utf8' 。

start - 指定开始读取的索引位置，默认为 0。

end - 结束位置，默认为缓冲区的末尾。

返回值

解码缓冲区数据并使用指定的编码返回字符串。

实例

buf = new Buffer(26);

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

buf[i] = i + 97;

}

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

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

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

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

执行以上代码，输出结果为：

$ node main.js

abcdefghijklmnopqrstuvwxyz

abcde

abcde

abcde

**Node Buffer 转换为 JSON 对象**

buf.toJSON()

返回值

返回 JSON 对象。

实例

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

var json = buf.toJSON(buf);

console.log(json);

执行以上代码，输出结果为：

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

**缓冲区合并**

Buffer.concat(list[, totalLength])

参数

参数描述如下：

list - 用于合并的 Buffer 对象数组列表。

totalLength - 指定合并后Buffer对象的总长度。

返回值

返回一个多个成员合并的新 Buffer 对象。

实例

var buffer1 = new Buffer('菜鸟教程 ');

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

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

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

执行以上代码，输出结果为：

buffer3 内容: 菜鸟教程 [www.runoob.com](http://www.runoob.com)

**比较(**该方法在 Node.js v0.12.2 版本引入)

buf.compare(otherBuffer);

参数

参数描述如下：

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

返回值

返回一个数字，表示 buf 在 otherBuffer 之前，之后或相同。

实例

var buffer1 = new Buffer('ABC');

var buffer2 = new Buffer('ABCD');

var result = buffer1.compare(buffer2);

if(result < 0) {

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

}else if(result == 0){

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

}else {

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

}

执行以上代码，输出结果为：

ABC在ABCD之前

**拷贝**

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

参数

参数描述如下：

targetBuffer - 要拷贝的 Buffer 对象。

targetStart - 数字, 可选, 默认: 0

sourceStart - 数字, 可选, 默认: 0

sourceEnd - 数字, 可选, 默认: buffer.length

返回值

没有返回值。

实例

var buffer1 = new Buffer('ABC');

// 拷贝一个缓冲区

var buffer2 = new Buffer(3);

buffer1.copy(buffer2);

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

执行以上代码，输出结果为：

buffer2 content: ABC

**裁剪**

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

参数

参数描述如下：

start - 数字, 可选, 默认: 0

end - 数字, 可选, 默认: buffer.length

返回值

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

实例

var buffer1 = new Buffer('runoob');

// 剪切缓冲区

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

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

执行以上代码，输出结果为：

buffer2 content: ru

**长度**

buf.length;

返回值

返回 Buffer 对象所占据的内存长度。

实例

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

// 缓冲区长度

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

执行以上代码，输出结果为：

buffer length: 14

Node.js，Stream 有四种流类型：

Readable - 可读操作。

Writable - 可写操作。

Duplex - 可读可写操作.

Transform - 操作被写入数据，然后读出结果。

所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：

data - 当有数据可读时触发。

end - 没有更多的数据可读时触发。

error - 在接收和写入过程中发生错误时触发。

finish - 所有数据已被写入到底层系统时触发。

创建 input.txt 文件，内容如下：

菜鸟教程官网地址：[www.runoob.com](http://www.runoob.com)

创建 main.js 文件, 代码如下：

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");

以上代码执行结果如下：

程序执行完毕

菜鸟教程官网地址：[www.runoob.com](http://www.runoob.com)

写入流

创建 main.js 文件, 代码如下：

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 文件中。代码执行结果如下：

$ node main.js

程序执行完毕

写入完成。

查看 output.txt 文件的内容：

$ cat output.txt

菜鸟教程官网地址：[www.runoob.com](http://www.runoob.com)

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。

创建 main.js 文件, 代码如下：

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");

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

接下来我们就是用管道和链式来压缩和解压文件。

创建 compress.js 文件, 代码如下：

var fs = require("fs");

var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.gz

fs.createReadStream('input.txt')

.pipe(zlib.createGzip())

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

console.log("文件压缩完成。");

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

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

var fs = require("fs");

var zlib = require('zlib');

// 解压 input.txt.gz 文件为 input.txt

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

.pipe(zlib.createGunzip())

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

console.log("文件解压完成。");

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。

把一个对象封装到模块中，格式如下：

module.exports = function() {

// ...

}

例如:

//hello.js

function Hello() {

var name;

this.setName = function(thyName) {

name = thyName;

};

this.sayHello = function() {

console.log('Hello ' + name);

};

};

module.exports = Hello;

这样就可以直接获得这个对象了：

//main.js

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

hello = new Hello();

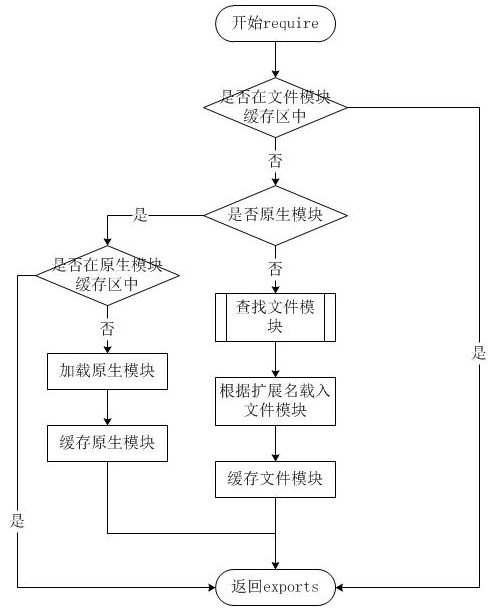
hello.setName('BYVoid');

hello.sayHello();

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

require方法接受以下几种参数的传递：

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



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

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函数的第一个变量进行了传递。这里返回的不是 say 的返回值，而是 say 本身！

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

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

可以直接在另一个函数的括号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 。

看看HTTP服务器两种写法

第一种

var http = require("http");

http.createServer(function(request, response) {

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

response.write("Hello World");

response.end();

}).listen(8888);

第二种

var http = require("http");

function onRequest(request, response) {

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

response.write("Hello World");

response.end();

}

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

路由

建立一个名为 router.js 的文件，添加以下内容：

function route(pathname) {

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

}

exports.route = route;

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

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;

会相应扩展index.js，使得路由函数可以被注入到服务器中：

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

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

server.start(router.route);

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

$ node index.js

Server has started.

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

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

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

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

**\_\_dirname** 表示当前执行脚本所在的目录。console.log( \_\_dirname );

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

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

function printHello(){

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

}

var t = setTimeout(printHello, 2000); // 两秒后执行以上函数

clearTimeout(t); // 清除定时器

**setInterval(cb, ms)** 全局函数在指定的毫秒(ms)数后执行指定函数(cb)。返回一个代表定时器的句柄值。可以使用 clearInterval(t) 函数来清除定时器。setInterval() 方法会不停地调用函数，直到 clearInterval() 被调用或窗口被关闭(按下 **ctrl + c** 按钮)。

以下为 console 对象的方法:

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **console.log([data][, ...])** 向标准输出流打印字符并以换行符结束。该方法接收若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf() 命令的格式输出。 |
| 2 | **console.info([data][, ...])** P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。 |

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

var util = require('util');

function Base() { //定义一个基础对象Base, 三个在构造函数内定义的属性和一个原型中定义的函数

this.name = 'base';

this.base = 1991;

this.sayHello = function() {

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

};

}

Base.prototype.showName = function() { //原型中定义的函数

console.log(this.name);

};

function Sub() {

this.name = 'sub';

}

util.inherits(Sub, Base); // sub 继承base 但仅继承原型中定义的函数

var objBase = new Base();

objBase.showName();

objBase.sayHello();

console.log(objBase);

var objSub = new Sub();

objSub.showName();

//objSub.sayHello();

console.log(objSub);

运行结果如下：

base

Hello base

{ name: 'base', base: 1991, sayHello: [Function] }

sub

{ name: 'sub' }

注意：Sub 仅仅继承了Base 在原型中定义的函数，而构造函数内部创造的 base 属 性和 sayHello 函数都没有被 Sub 继承。同时，在原型中定义的属性不会被console.log 作为对象的属性输出。

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

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

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

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

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));

运行结果是：

{ name: 'byvoid', toString: [Function] }

{ toString:

{ [Function]

[prototype]: { [constructor]: [Circular] },

[caller]: null,

[length]: 0,

[name]: '',

[arguments]: null },

name: 'byvoid' }

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

Node 导入文件系统模块(fs)语法如下所示：

var fs = require("fs")

创建 input.txt 文件，内容如下：

菜鸟教程官网地址：www.runoob.com

文件读取实例

创建 file.js 文件, 代码如下：

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：

$ node file.js

同步读取: 菜鸟教程官网地址：www.runoob.com

文件读取实例

程序执行完毕。

异步读取: 菜鸟教程官网地址：www.runoob.com

文件读取实例

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)。

**异步模式下打开文件的语法格式**：fs.open(path, flags[, mode], callback)

path - 文件的路径。

flags - 文件打开的行为。具体值详见下文。

mode - 设置文件模式(权限)，文件创建默认权限为 0666(可读，可写)。

callback - 回调函数，带有两个参数如：callback(err, fd)。

flags 参数可以是以下值：

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。

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

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！");

});

以上代码执行结果如下：

$ node file.js

准备打开文件！

文件打开成功！

**异步模式获取文件信息的语法格式：**fs.stat(path, callback)

path - 文件路径。

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

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

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：

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。

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

准备打开文件！

{ dev: 16777220,

mode: 33188,

nlink: 1,

uid: 501,

gid: 20,

rdev: 0,

blksize: 4096,

ino: 40333161,

size: 61,

blocks: 8,

atime: Mon Sep 07 2015 17:43:55 GMT+0800 (CST),

mtime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST),

ctime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST) }

读取文件信息成功！

是否为文件(isFile) ? true

是否为目录(isDirectory) ? false

**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。

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 utf8, 模式为 0666 ， flag 为 'w'

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

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

准备写入文件

数据写入成功！

--------我是分割线-------------

读取写入的数据！

异步读取文件数据: 我是通过写入的文件内容

**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。

fd - 通过 fs.open() 方法返回的文件描述符。

buffer - 数据写入的缓冲区。

offset - 缓冲区写入的写入偏移量。

length - 要从文件中读取的字节数。

position - 文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。

callback - 回调函数，有三个参数err, bytesRead, buffer，err 为错误信息， bytesRead 表示读取的字节数，buffer 为缓冲区对象。

input.txt 文件内容为：

菜鸟教程官网地址：www.runoob.com

接下来我们创建 file.js 文件，代码如下所示：

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());

}

});

});

以上代码执行结果如下：

$ node file.js

准备打开已存在的文件！

文件打开成功！

准备读取文件：

42 字节被读取

菜鸟教程官网地址：www.runoob.com

**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。

fd - 通过 fs.open() 方法返回的文件描述符。

callback - 回调函数，没有参数。

input.txt 文件内容为：

菜鸟教程官网地址：www.runoob.com

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

准备打开文件！

文件打开成功！

准备读取文件！

菜鸟教程官网地址：www.runoob.com

文件关闭成功

**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。

fd - 通过 fs.open() 方法返回的文件描述符。

len - 文件内容截取的长度。

callback - 回调函数，没有参数。

input.txt 文件内容为：

site:www.runoob.com

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

准备打开文件！

文件打开成功！

截取10字节后的文件内容。

文件截取成功。

读取相同的文件

site:www.r

文件关闭成功

**删除文件的语法格式：**fs.unlink(path, callback)

path - 文件路径。

callback - 回调函数，没有参数。

input.txt 文件内容为：

site:www.runoob.com

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

准备删除文件！

文件删除成功！

再去查看 input.txt 文件，发现已经不存在了。

**创建目录的语法格式：**fs.mkdir(path[, mode], callback)

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。

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

创建目录 /tmp/test/

目录创建成功。

**读取目录的语法格式：**fs.readdir(path, callback)

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。

接下来我们创建 file.js 文件，代码如下所示：

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：

$ node file.js

查看 /tmp 目录

input.out

output.out

test

test.txt

**删除目录的语法格式**：fs.rmdir(path, callback)

path - 文件路径。

callback - 回调函数，没有参数。

接下来我们创建 file.js 文件，代码如下所示：

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 );

});

});

});

以上代码执行结果如下：

$ node file.js

准备删除目录 /tmp/test

input.out

output.out

test

test.txt

读取 /tmp 目录

……

一个最基本的 HTTP 服务器架构(使用8081端口)，创建 server.js 文件，代码如下所示：

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(8081);

// 控制台会输出以下信息

console.log('Server running at http://127.0.0.1:8081/');

接下来我们在该目录下创建一个 index.htm 文件，代码如下：

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

执行 server.js 文件：

$ node server.js

Server running at http://127.0.0.1:8081/

执行 server.js 的控制台输出信息如下：

Server running at http://127.0.0.1:8081/

Request for /index.htm received. # 客户端请求信息

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

var http = require('http');

// 用于请求的选项

var options = {

host: 'localhost',

port: '8081',

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();

新开一个终端，执行 client.js 文件，输出结果如下：

$ node client.js

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

执行 server.js 的控制台输出信息如下：

Server running at http://127.0.0.1:8081/

Request for /index.htm received. # 客户端请求信息