# Nodejs

## nodejs学习1：安装与启动

本文最后一次更新于**2年前**，文章内容可能略有出入。若发现文章中有错误之处，可以留言评论告诉作者。

### 1、什么是nodejs

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js是一个可以快速构建网络服务及应用的平台。该平台的构建是基于Chrome's JavaScript runtime，也就是说，实际上它是对Google V8引擎（应用于Google Chrome浏览器)进行了封装。

### 2、为什么要学习nodejs

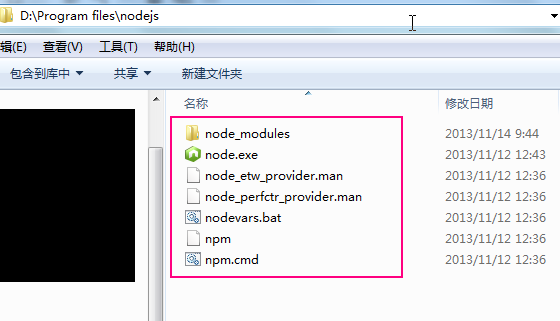
nodejs是一门新的思考方向，写法与js类似，免去了前端开发人员去额外学习PHP等语言的过程，也会相对程度的减少其他语言带来的一些难以预 计的问题。和前端的HTML5、CSS3一样，都会让前端开发变得更为主动、更加有趣、更有激情，仿佛前端开发的未来就在眼前了，只等你的伸手一触。

nodejs的执行过程是单线程异步的，与浏览器的js运行方式一致，回调是它的最主要特色，经常写ajax的应该会非常熟悉。异步回调有它的优点 也有它的缺点，暂且搁置不谈。 正因为它是异步回调的，所以它的执行过程是非阻塞的，经过测试，它的执行性能明显优于其他语言（在一门语言的初级阶段，这 是必然的，不然它就不会出现），即使它使用的是单核。

平常接触到的都是前端问题，总是让人觉得前端很低端（可能原因是它的入门比较低），在一般公司内，前端开发人员的地位都没有后端开发人员来的高，通 常前端开发和后端开发的人数比例是1:3以下，非常畸形，并且前端开发的进程是受到后端开发的限制和制约的。现如今，nodejs出现了（尽管它出生于 2009年），用js就可以写后端，多么振奋的一件事啊。前端人员再也不必忍受默默的鄙视了，终于有了扬眉吐气的时候了，成为全端开发人员指日可待。

### 3、如何安装

在官网下载对应操作平台的nodejs包，本学习笔记主要以windows为主。傻瓜化安装即可。安装完成的文件目录大致如下图：



### 4、如何启动

安装完成，打开cmd命令窗口（开始菜单->运行命令->输入“cmd”回车）。运行：

1. node console.log("Hello world!");
2. // Hello world!
3. // undefined

这些命令和在谷歌浏览器控制台操作几乎一致，只是在前面多了个node而已。

nodejs不但能运行控制台命令，也可以运行一个js文件。比如，在E盘根目录新建一个test.js文件，里面的内容为：

1. console.log("test");

然后在cmd命令窗口按两次ctrl+c退出刚才的命令过程，然后输入：

1. cd e:\
2. e:
3. node test.js
4. // test

上面4行代码的意思分别为：

1. 定位到E盘根目录
2. 切换到e盘该目录
3. 用nodejs执行test.js
4. 运行结果为控制台输出“test”

## Nodejs +express安装

相信对于很多关注javascript发展的同学来说,nodejs已经不是一个陌生的词眼,这里不想谈太多的nodejs的相关信息。只说一下，windows系统下简单nodejs环境配置

相信对于很多关注javascript发展的同学来说,nodejs已经不是一个陌生的词眼。有关nodejs的相关资料网上已经铺天盖地。由于它的高并发特性，造就了其特殊的应用地位。

国内目前关注最高，维护最好的一个关于nodejs的网站应该是http://www.cnodejs.org/ http://files.jb51.net/file_images/article/201301/201301080938342.png

这里不想谈太多的nodejs的相关信息。只说一下，windows系统下简单nodejs环境配置。

**第一步：下载安装文件**

下载地址：官网http://www.nodejs.org/download/ 

[](http://files.jb51.net/file_images/article/201301/201301080938344.jpg)

这里用的是 

**第二步：安装nodejs**

下载完成之后，双击 http://files.jb51.net/file_images/article/201301/201301080938346.jpgnode-v0.8.16-x86.msi，开始安装nodejs，默认是安装在C:\Program Files\nodejs下面

**第三步：安装相关环境**

打开C:\Program Files\nodejs目录你会发现里面自带了npm,直接用npm安装相环境既可

 进入node.js command prompt 命令窗口

进入nodejs 安装目录 C:\Program Files\nodejs

键入命令：cd C:\Program Files\nodejs 既可

**现在开始安装相关环境**

键入命令：npm express 回车等待安装express........

键入命令：npm jade 回车等待安装jade........

键入命令：npm mysql回车等待安装mysql........

........安装什么组件，取决于环境搭建需求

默认情况下上述组件都是安装在C:\Program Files\nodejs\node\_modules文件夹下 这也是nodejs相关组件的自动查找路径

**第四步：创建一个工程**

现在已经有express

键入:express myapp (myapp是随意起的工程名称)

你会发现多了一个 C:\Program Files\nodejs\myapp 目录

默认情况下：里会自动创建 [http://files.jb51.net/file_images/article/201301/201301080938347.jpg](http://files.jb51.net/file_images/article/201301/201301080938347.jpg)

这几个文件，不做解释，相信有过开发经验的同学都能一眼明了。

复制node\_modules到myapp下面

环境搭建到此完工，下面做一个demo测试！

在myapp下新建helloworld.js

复制代码 代码如下:

var http = require("http");   
http.createServer(function(request, response) {   
response.writeHead(200, {"Content-Type": "text/plain"});   
response.write("Hello World");   
response.end();   
}).listen(8888);   
console.log("nodejs start listen 8888 port!");

进入node.js command prompt 命令窗口,进入C:\Program Files\nodejs\myapp目录

键入node helloworld.js

http://files.jb51.net/file_images/article/201301/201301080938348.jpg

打开地址http://127.0.0.1:8888/

发现输出 :Hello World

## NODEJS原理篇

### 事件轮询机制

#### process.nextTick(callback)

Node.js是单线程的，基于事件循环，非阻塞 IO的。事件循环中使用一个事件队列，在每个**时间点**上，系统只会处理一个事件，即使电脑有多个CPU核心，也无法同时并行的处理多个事件。因此，node.js适合处理I／O型的应用，不适合那种CPU运算密集型的应用。在I／O型的应用中，给每一个输入输出定义一个回调函数，node.js会自动将其加入到事件轮询的处理队列里，当I／O操作完成后，这个回调函数会被触发，系统会继续处理其他的请求。

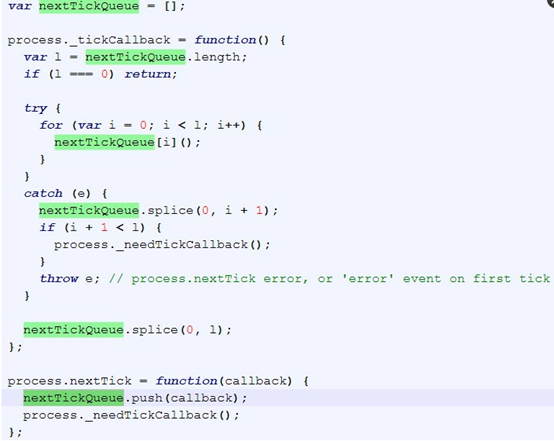
在这里用debuggable.com上的那个文章中的一段比喻来讲，非常容易理解。如下：

我们写的js代码就像是一个国王，而nodejs给国王提供了很多仆人。早上，一个仆人叫醒了国王，问他有什么需要。国王给他一份清单，上面列举了所有需要完成的任务，然后睡回笼觉去了。当国王回去睡觉之后，仆人才离开国王，拿着清单，给其它的仆人一个个布置任务。仆人们各自忙各自的去了，直到完成了自己的任务后，才回来把结果禀告给国王。国王一次只召见一个人，其它的人就在外面排着队等着。国王处理完这个结果后，可能给他布置一个新的任务，或者就直接让他走了，然后再召见下一个人。等所有的结果都处理完了，国王就继续睡觉去了。直接有新的仆人完成任务后过来找他。这就是国王的幸福生活。

**process.nextTick(callback)**

功能：在事件循环的下一次循环中调用 callback 回调函数。效果是将一个函数推迟到代码书写的下一个**同步方法执行完毕时**或**异步方法的事件回调函数开始执行时**；与setTimeout(fn, 0) 函数的功能类似，但它的效率高多了。

基于node.js的事件循环分析，每一次循环就是一次tick，每一次tick时，v8引擎从事件队列中取出所有事件依次进行处理，如果遇到nextTick事件，则将其加入到事件队尾，等待下一次tick到来时执行；造成的结果是，nextTick事件被延迟执行；以下是nextTick源码



从这几行代码中，我们可以看出很多信息：

1. nextTick的确是把某任务放在队列的最后（array.push)
2. nodejs在执行任务时，会一次性把队列中所有任务都拿出来，依次执行
3. 如果全部顺利完成，则删除刚才取出的所有任务，等待下一次执行
4. 如果中途出错，则删除已经完成的任务和出错的任务，等待下次执行
5. 如果第一个就出错，则throw erro

### Node是单线程的

### 回调函数实现非阻塞开发

#### Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

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

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

#### 阻塞代码实例

创建一个文件 input.txt ，内如如下：

菜鸟教程官网地址：www.runoob.com

创建 main.js 文件, 代码如下：

var fs = require("fs");

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

console.log(data.toString());

console.log("程序执行结束!");

以上代码执行结果如下：

$ node main.js

菜鸟教程官网地址：www.runoob.com

程序执行结束!

#### 非阻塞代码实例

创建一个文件 input.txt ，内如如下：

菜鸟教程官网地址：www.runoob.com

创建 main.js 文件, 代码如下：

var fs = require("fs");

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

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

console.log(data.toString());

});

console.log("程序执行结束!");

以上代码执行结果如下：

$ node main.js

程序执行结束!

菜鸟教程官网地址：www.runoob.com

以上两个实例我们了解了阻塞与非阻塞调用的不同。第一个实例在文件读取完后才执行完程序。 第二个实例我们呢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。

### 事件和回掉函数 – nodejs的基础

#### 内置事件模块

    events是node.js 最重要的模块，events模块只提供了一个对象events.EventEmitter，EventEmitter 的核心是事件发射与事件监听器。

    Node.js中大部分的模块，都继承自Event模块。

    与DOM树上事件不同，不存在事件冒泡、逐层捕获等行为。

    EventEmitter 支持若干个事件监听器。当事件发射时，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。

如何访问：  
require('events');

emitter.on(event, listener)

/\*  
    调用events模块，获取events.EventEmitter对象  
\*/  
var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
/\*  
    EventEmitter.on(event, listener) 为事件注册一个监听  
    参数1：event  字符串，事件名  
    参数2：回调函数  
\*/  
ee.on('some\_events', function(foo, bar) {  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
});  
console.log('第一轮');  
ee.emit('some\_events', 'Wilson', 'Zhong');  
console.log('第二轮');  
ee.emit('some\_events', 'Wilson', 'Z');  
EventEmitter.on(event, listener) 示例源码

**emitter.emit(event, [arg1], [arg2], [...])**

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
ee.on('some\_events', function(foo, bar) {           
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
});  
/\*  
    EventEmitter.emit(event, [arg1], [arg2], [...])   触发指定事件  
    参数1：event  字符串，事件名  
    参数2：可选参数，按顺序传入回调函数的参数  
    返回值：该事件是否有监听  
\*/  
var isSuccess = ee.emit('some\_events', 'Wilson', 'Zhong');  
ee.on('some\_events', function(foo, bar) {           
    console.log("第2个监听事件,参数foo=" + foo + ",bar="+bar );  
});  
ee.emit('some\_events', 'zhong', 'wei');  
var isSuccess2 = ee.emit('other\_events', 'Wilson', 'Zhong');  
console.log(isSuccess);  
console.log(isSuccess2);  
emitter.emit(event, [arg1], [arg2], [...]) 示例源码

示例进行了三次触发事件操作，其中some\_events注册了监听，调用时emit函数会返回一个true,而other\_events并没有注册监听，emit函数会返回一个false，表示该事件没有监听；当然也可以不用管这个返回值！

**emitter.once(event, listener)**

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
/\*  
    EventEmitter.once(event, listener)  为事件注册一次性监听，触发一次后移除监听  
    参数1：event  字符串，事件名  
    参数2：回调函数  
\*/  
ee.once('some\_events', function(foo, bar) {  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
});  
console.log('第一轮');  
ee.emit('some\_events', 'Wilson', 'Zhong');  
console.log('第二轮');  
var isSuccess =  ee.emit('some\_events', 'Wilson', 'Zhong');  
console.log(isSuccess);  
emitter.once(event, listener) 示例源码

    从上面示例代码执行结果可以看出，用emitter.once给some\_events注册一个监听后，分两轮调用emitter.emit触发，第二轮会返回false；这表示用emitter.once注册监听和用前面讲的emitter.on注册监听略有不同，

    emitter.once注册监听是一次性监听，当触发一次后，会移除该监听！当然，从名字上就看就比较明显了^\_^!

**emitter.removeListener(event, listener)**

 先来看一个失败的场景~~~

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
ee.on('some\_events', function(foo, bar) {  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
});  
/\*  
    看到API中removeListener移除方法时，以为应该是这样  
    但是结果^\_^!!!!!  
\*/  
ee.removeListener('some\_events', function(){  
    console.log('成功移除事件some\_events监听!');          
});  
console.log('第一轮');  
ee.emit('some\_events', 'Wilson', 'Zhong');  
emitter.removeListener(event, listener) 示例失败场景源码

    当我用emitter.on给some\_events注册了一个监听后，我用emiiter.removeListener移除some\_events的监听，随后再调用emitter.emit去触发，最后发现不是按我想像的在进行！为什么呢？

    我理所当然的认为emiiter.removeListener第二个参数是个回调函数，API还是要认真看清楚啊！！！

下面再看个成功的场景～～～

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
var listener = function(foo,bar)  
{  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
var listener2= function(foo,bar)  
{  
    console.log("第2个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
var listener3= function(foo,bar)  
{  
    console.log("第3个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
ee.on('some\_events', listener);  
ee.on('some\_events', listener2);  
ee.on('some\_events', listener3);  
/\*  
    EventEmitter.removeListener(event, listener)  移除指定事件的监听器  
    注意：该监听器必须是注册过的  
    PS：上一个例子之后以会失败，很大原因就是忽略了监听器，理所当然的认为传个事件名就OK了，所以就悲剧了!  
\*/  
ee.removeListener('some\_events', listener);  
ee.removeListener('some\_events', listener3);  
ee.emit('some\_events', 'Wilson', 'Zhong');  
emitter.removeListener(event, listener) 示例成功场景源码

    我用示例中写法，给some\_events添加了三个监听，又移除了第一个和第三个监听，最后再用emitter.emit触发some\_events，输出结果不难发现，用emitter.removeListener移除的第一个和第三个监听都没有再起作用，

    想当然是害人地，原来emitter.removeListener的第二个参数是要移除的监听，而非移除成功后的回调函数……^\_^!

**emitter.removeAllListeners([event])**

emitter.removeListener用过了，但一个事件可以有多个监听，需要全部移除时，一个个移除明显不是愉快的做法，不符合偷懒的天性！

让我们来体验一下emitter.removeAllListeners带来的便捷!

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
var listener = function(foo,bar)  
{  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
var listener2= function(foo,bar)  
{  
    console.log("第2个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
ee.on('some\_events', listener);  
ee.on('some\_events', listener2);  
ee.on('other\_events',function(foo,bar)  
{  
    console.log("其它监听事件,参数foo=" + foo + ",bar="+bar );  
});  
/\*  
    EventEmitter.removeAllListeners([event])   移除（批定事件）所有监听器  
    参数1：可选参数，event  字符串，事件名  
\*/  
ee.removeAllListeners('some\_events');  
ee.emit('some\_events', 'Wilson', 'Zhong');  
ee.emit('other\_events', 'Wilson', 'Zhong');  
emitter.removeAllListeners 传入事件名参数示例源码

    看看上面的执行结果，你会发现给some\_events注册了两个监听；给other\_events注册了一个监听；我调用emitter.removeAllListeners传了some\_events事件名；

    最后使用emitter.on函数触发some\_events和other\_events两个事件，最后发现some\_events注册的两个监听都不存在，而other\_events注册的监听还存在；

    这表示当 emitter.removeAllListeners传用事件名作为参数时，为移除传入事件名的所有监听，而不会影响其它事件监听！

emitter.removeAllListeners可以不传用事件名参数；直接执行

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
var listener = function(foo,bar)  
{  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
var listener2= function(foo,bar)  
{  
    console.log("第2个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
ee.on('some\_events', listener);  
ee.on('some\_events', listener2);  
ee.on('other\_events',function(foo,bar)  
{  
    console.log("其它监听事件,参数foo=" + foo + ",bar="+bar );  
});  
/\*  
    EventEmitter.removeAllListeners([event])   移除（批定事件）所有监听器  
    参数1：可选参数，event  字符串，事件名  
\*/  
ee.removeAllListeners();  
ee.emit('some\_events', 'Wilson', 'Zhong');  
ee.emit('other\_events', 'Wilson', 'Zhong');  
emitter.removeAllListeners 不传参数示例源码

    示例代码和传入参数时几乎一样，只是在调用emitter.removeAllListeners并没有传入指定事件名；

    运行结果会发现some\_events和other\_events所有监听都不存在了，它会移除所有监听！（比较暴力的方法一般要慎用~~）

**emitter.listeners(event)**

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
var listener = function(foo,bar)  
{  
    console.log("第1个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
var listener2= function(foo,bar)  
{  
    console.log("第2个监听事件,参数foo=" + foo + ",bar="+bar );  
}  
ee.on('some\_events', listener);  
ee.on('some\_events', listener2);  
ee.on('other\_events',function(foo,bar)  
{  
    console.log("其它监听事件,参数foo=" + foo + ",bar="+bar );  
});  
/\*  
    EventEmitter.listeners(event)   //返回指定事件的监听数组  
    参数1：event  字符串，事件名      
\*/  
var listenerEventsArr = ee.listeners('some\_events');  
console.log(listenerEventsArr.length)  
for (var i = listenerEventsArr.length - 1; i >= 0; i--) {  
    console.log(listenerEventsArr[i]);   
};  
emitter.listeners(event) 示例源码

    给some\_events注册两个监听，调用emitter.listeners函数，传入some\_events事件名，接收函数返回值；

    从结果可以看出，返回值接收到some\_events所有注册监听的集合！

**emitter.setMaxListeners(n)**

 一个事件可以添加多个监听是没错，但Nodejs默认最大值是多少呢？

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
/\*  
     给EventEmitter 添加11个监听  
\*/  
for (var i = 10; i >= 0; i--) {  
    ee.on('some\_events',function()  
    {  
        console.log('第'+ (i +1) +'个监听');  
    });  
};  
添加N个监听示例源码

上面示例中我用个循环给some\_events添加11个监听，执行代码，发现warning信息出现，并且提示的比较详细了，需要用emitter.setMaxListeners()去提升限值

var EventEmitter = require('events').EventEmitter;     
var ee = new EventEmitter();  
/\*  
    EventEmitter.setMaxListeners (n)   给EventEmitter设置最大监听  
    参数1： n 数字类型，最大监听数  
  
    超过10个监听时，不设置EventEmitter的最大监听数会提示：  
    (node) warning: possible EventEmitter memory leak detected. 11 listeners added.  
     Use emitter.setMaxListeners() to increase limit.  
    设计者认为侦听器太多，可能导致内存泄漏，所以存在这样一个警告  
\*/  
ee.setMaxListeners(15);  
/\*  
     给EventEmitter 添加11个监听  
\*/  
for (var i = 10; i >= 0; i--) {  
    ee.on('some\_events',function()  
    {  
        console.log('第'+ (i +1) +'个监听');  
    });  
};  
emitter.setMaxListeners 示例源码

    当我调用emitter.setMaxListeners传入15时，执行代码，warning信息不再出现；

    emitter.setMaxListeners的作用是给EventEmitter设置最大监听数，感觉一般是不需要设置这个值，10个还不够用的情况应该是比较少了！

    设计者认为侦听器太多会导致内存泄漏，所有就给出了一个警告！

其它...

 用的比较少的就不详细说了

EventEmitter.defaultMaxListeners

    EventEmitter.defaultMaxListeners功能与setMaxListeners类似，  
    给所有EventEmitter设置最大监听  
    setMaxListeners优先级大于defaultMaxListeners

EventEmitter.listenerCount(emitter, event)

    返回指定事件的监听数

 特殊的事件Error

    引用自Node.js开发指南：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 异常的时候通常会发射 error 事件。当 error 被发射时，EventEmitter 规定如果没有响 应的监听器，Node.js 会把它当作异常，退出程序并打印调用栈。我们一般要为会发射 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。

事件的继承

    以后归到util里再讲一下吧，有兴趣的可以自已看看 http://nodejs.org/api/util.html#util\_util\_inherits\_constructor\_superconstructor

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

Node.js里面的许多对象都会分发事件：

一个net.Server对象会在每次有新连接时分发一个事件，

一个fs.readStream对象会在文件被打开的时候发出一个事件。

所有这些产生事件的对象都是 events.EventEmitter 的实例。

事件概述：

  EventEmitter.on(event,listener) //注册一个时间，事件名为event，相应事件的回调函数为listener

     EventEmitter.emit(event,[arg1],[arg2],[…])//args为发射已注册的事件event，并传递若干个args参数

     EventEmitter.once(event,listener)//单词监听事件，时间最多只能触发一次，出发后该注册的event时间会解除监听

     EventEmitter.removeListrner(event,listener)//解除已监听了的事件

     EventEmitter.removeListrner(event,listener)

#### 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。

你可以通过require("events");来访问该模块。

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

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

下面我们用一个简单的例子说明 EventEmitter 的用法：

//event.js 文件

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

var event = new EventEmitter();

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

console.log('some\_event 事件触发');

});

setTimeout(function() {

event.emit('some\_event');

}, 1000);

执行结果如下：

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

$ node event.js

some\_event 事件触发

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

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

让我们以下面的例子解释这个过程：

//event.js 文件

var events = require('events');

var emitter = new events.EventEmitter();

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

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

});

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

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

});

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

执行以上代码，运行的结果如下：/p>

$ node event.js

listener1 arg1 参数 arg2 参数

listener2 arg1 参数 arg2 参数

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

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

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

方法

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | **addListener(event, listener)** 为指定事件添加一个监听器到监听器数组的尾部。 |
| 2 | **on(event, listener)** 为指定事件注册一个监听器，接受一个字符串 event 和一个回调函数。  server.on('connection', function (stream) {  console.log('someone connected!');  }); |
| 3 | **once(event, listener)** 为指定事件注册一个单次监听器，即 监听器最多只会触发一次，触发后立刻解除该监听器。  server.once('connection', function (stream) {  console.log('Ah, we have our first user!');  }); |
| 4 | **removeListener(event, listener)** 移除指定事件的某个监听器，监听器 必须是该事件已经注册过的监听器。  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 listner1 = function listner1() {

console.log('监听器 listner1 执行。');

}

// 监听器 #2

var listner2 = function listner2() {

console.log('监听器 listner2 执行。');

}

// 绑定 connection 事件，处理函数为 listner1

eventEmitter.addListener('connection', listner1);

// 绑定 connection 事件，处理函数为 listner2

eventEmitter.on('connection', listner2);

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

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

// 处理 connection 事件

eventEmitter.emit('connection');

// 移除监绑定的 listner1 函数

eventEmitter.removeListener('connection', listner1);

console.log("listner1 不再受监听。");

// 触发连接事件

eventEmitter.emit('connection');

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

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

监听器 listner1 执行。

监听器 listner2 执行。

listner1 不再受监听。

监听器 listner2 执行。

1 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 = newevents.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。

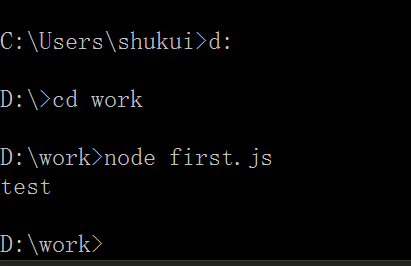
其次 JavaScript 的对象机制是基于原型的，支持 部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

## Nodejs基础

### Console.log

Console.log(‘123’);

### 执行一个外部文件



### Node.js 全局对象

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

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

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

#### 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。按照 ECMAScript 的定义，满足以下条 件的变量是全局变量：

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

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

**注意：** 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。

实例

创建文件 main.js ，代码如下所示：

// 输出全局变量 \_\_filename 的值

console.log( \_\_filename );

执行 main.js 文件，代码如下所示:

$ node main.js

/web/com/runoob/nodejs/main.js

#### \_\_dirname

**\_\_dirname** 表示当前执行脚本所在的目录。

实例

创建文件 main.js ，代码如下所示：

// 输出全局变量 \_\_dirname 的值

console.log( \_\_dirname );

执行 main.js 文件，代码如下所示:

$ node main.js

/web/com/runoob/nodejs

#### setTimeout(cb, ms)

**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。

实例

创建文件 main.js ，代码如下所示：

function printHello(){

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

}

// 两秒后执行以上函数

setTimeout(printHello, 2000);

执行 main.js 文件，代码如下所示:

$ node main.js

Hello, World!

#### clearTimeout(t)

**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。

实例

创建文件 main.js ，代码如下所示：

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);

执行 main.js 文件，代码如下所示:

$ node main.js

#### setInterval(cb, ms)

**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。

实例

创建文件 main.js ，代码如下所示：

function printHello(){

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

}

// 两秒后执行以上函数

setInterval(printHello, 2000);

执行 main.js 文件，代码如下所示:

$ node main.js

Hello, World! Hello, World! Hello, World! Hello, World! Hello, World! ……

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，它是由 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

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。 |

console.log()：向标准输出流打印字符并以换行符结束。

console.log 接受若干 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。

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);

运行结果为：

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();

运行结果为：

Trace:

at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)

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)

实例

创建文件 main.js ，代码如下所示：

console.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。")

执行 main.js 文件，代码如下所示:

$ node main.js

程序开始执行：

计数: 10

获取数据: 0ms

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，少不了要 和它打交道。下面将会介绍 process 对象的一些最常用的成员方法。

|  |  |
| --- | --- |
| **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

创建文件 main.js ，代码如下所示：

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");

执行 main.js 文件，代码如下所示:

$ node main.js

程序执行结束

退出码为: 0

退出状态码

退出状态码如下所示：

|  |  |
| --- | --- |
| **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

Process 提供了很多有用的属性，便于我们更好的控制系统的交互：

|  |  |
| --- | --- |
| **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

创建文件 main.js ，代码如下所示：

// 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);

执行 main.js 文件，代码如下所示:

$ node main.js

Hello World!

0: node

1: /web/www/node/main.js

/usr/local/node/0.10.36/bin/node

darwin

方法参考手册

Process 提供了很多有用的方法，便于我们更好的控制系统的交互：

|  |  |
| --- | --- |
| **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

创建文件 main.js ，代码如下所示：

// 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());

执行 main.js 文件，代码如下所示:

$ node main.js

当前目录: /web/com/runoob/nodejs

当前版本: v0.10.36

{ rss: 12541952, heapTotal: 4083456, heapUsed: 2157056 }

### 常用工具或者模块

#### util 内置的常用操作

##### util.inherits 继承的简单实现方式

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

JavaScript 的面向对象特性是基于原型的，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。

在这里我们只介绍util.inherits 的用法，示例如下：

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 和一个继承自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：

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。如果我们去掉 objSub.sayHello(); 这行的注释，将会看到：

node.js:201

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

^

TypeError: Object #&lt;Sub&gt; has no method 'sayHello'

at Object.&lt;anonymous&gt; (/home/byvoid/utilinherits.js:29:8)

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)

##### util.inspect – 调试用 输出对象

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 = newPerson();

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

##### **util.format(format,[...])格式化输出字符串**

#### Node.js os

Node.js os 模块提供了一些基本的系统操作函数。我们可以通过以下方式引入该模块：

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

创建 main.js 文件，代码如下所示：

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：

$ node main.js

endianness : LE

type : Linux

platform : linux

total memory : 25103400960 bytes.

free memory : 20676710400 bytes.

#### 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

创建 main.js 文件，代码如下所示：

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'));

代码执行结果如下：

$ node main.js

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

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

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

ext name : .js

#### NET – 服务器编程模块

Node.js Net 模块提供了一些用于底层的网络通信的小工具，包含了创建服务器/客户端的方法，我们可以通过以下方式引入该模块：

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 – 子命名空间

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 -子命名空间

net.Socket 对象是 TCP 或 UNIX Socket 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 创建它们，并通过 connection 服务器事件传递给用户。

事件

net.Socket 事件有：

|  |  |
| --- | --- |
| **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

net.Socket 提供了很多有用的属性，便于控制 socket 交互：

|  |  |
| --- | --- |
| **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

创建 server.js 文件，代码如下所示：

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：

$ node server.js

server is listening # 服务已创建并监听 8080 端口

新开一个窗口，创建 client.js 文件，代码如下所示：

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');

});

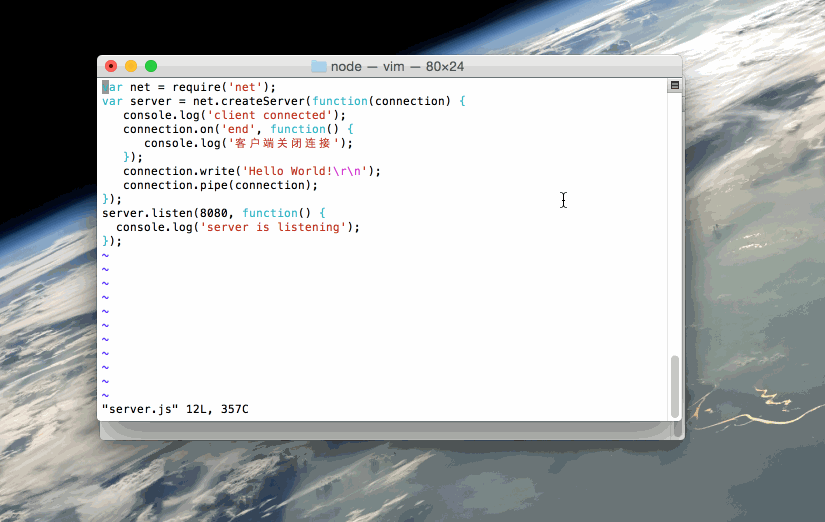
执行以上客户端的代码：

连接到服务器！

Hello World!

断开与服务器的连接

Gif 实例演示



#### DNS 模块

Node.js **DNS** 模块用于解析域名。引入 DNS 模块语法格式如下：

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 查询都可能返回以下错误码：

* 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

创建 main.js 文件，代码如下所示：

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:

address: 192.30.252.130

reverse for 192.30.252.130: ["github.com"]

#### Async

Async的内容分为三部分：

* 流程控制：简化十种常见流程的处理
* 集合处理：如何使用异步操作处理集合中的数据
* 工具类：几个常用的工具类

本文介绍其中最简单最常用的流程控制部分。   
由于nodejs是异步编程模型，有一些在同步编程中很容易做到的事情，现在却变得很麻烦。Async的流程控制就是为了简化这些操作。

##### series(tasks, [callback]) （多个函数依次执行，之间没有数据交换）

有多个异步函数需要依次调用，一个完成之后才能执行下一个。各函数之间没有数据的交换，仅仅需要保证其执行顺序。这时可使用series。   
纯js代码：

step1(function(err, v1) {

step2(function(err, v2) {

step3(function(err, v3) {

*// do somethig with the err or values v1/v2/v3*

}

}

});

从中可以看到这嵌套还是比较多深的，如果再多几步，会更深。在代码中忽略对了每一层err的处理，否则还都等加上 if(err) return callback(err)，那就更麻烦了。   
对于这种情况，使用async来处理，就是这样的：

var async = require(‘async’) async.series([step1, step2, step3],

function(err, values) *{*

*// do somethig with the err or values v1/v2/v3*

*}*);

可以看到代码简洁了很多，而且自动处理每个回调中的错误。当然，这里只给出来最最简单的例子，在实际中，我们常会在每个step中执行一些操作，这时可写成：

var async = require(‘async’) async.series([function(cb) {

step1(function(err, v1) {

// do something with v1

cb(err, v1);

}),

function(cb) {

step2(...)

},

function(cb) {

step3(...)

}],

function(err, values) {

// do somethig with the err or values v1/v2/v3

});

该函数的详细解释为：   
依次执行一个函数数组中的每个函数，每一个函数执行完成之后才能执行下一个函数。   
如果任何一个函数向它的回调函数中传了一个error，则后面的函数都不会被执行，并且将会立刻会将该error以及已经执行了的函数的结果，传给series中最后那个callback。   
当所有的函数执行完后（没有出错），则会把每个函数传给其回调函数的结果合并为一个数组，传给series最后的那个callback。   
还可以json的形式来提供tasks。每一个属性都会被当作函数来执行，并且结果也会以json形式传给series最后的那个callback。这种方式可读性更高一些。   
具体例子可参考：https://github.com/freewind/async\_demo/blob/master/series.js   
其代码中还包含了：   
如果中间某个函数出错，series函数如何处理   
如果某个函数传给回调的值为undefined, null, {}, []等，series如何处理   
另外还需要注意的是：多个series调用之间是不分先后的，因为series本身也是异步调用。

##### parallel(tasks, [callback]) （多个函数并行执行）

并行执行多个函数，每个函数都是立即执行，不需要等待其它函数先执行。传给最终callback的数组中的数据按照tasks中声明的顺序，而不是执行完成的顺序。   
如果某个函数出错，则立刻将err和已经执行完的函数的结果值传给parallel最终的callback。其它未执行完的函数的值不会传到最终数据，但要占个位置。   
同时支持json形式的tasks，其最终callback的结果也为json形式。   
示例代码：

async.parallel([function(cb) {

t.fire('a400', cb, 400)

},

function(cb) {

t.fire('a200', cb, 200)

},

function(cb) {

t.fire('a300', cb, 300)

}],

function(err, results) {

log(’1 err: ‘, err); // -> undefined

log(’1 results: ‘, results); // ->[ 'a400', 'a200', 'a300' ]

});

中途出错的示例：

async.parallel([function(cb) {

log('1: ', 'start');

t.fire('a400', cb, 400)

},

*// 该函数的值不会传给最终callback，但要占个位置*

function(cb) {

log('2: ', 'start');

t.err('e200', cb, 200)

},

function(cb) {

log('3: ', 'start');

t.fire('a100', cb, 100)

}],

function(err, results) {

log(’2 err: ‘, err); *// -> e200*

log(’2 results: ‘, results); *// -> [ , undefined, 'a100' ]*

});

以json形式传入tasks

async.parallel({

a: function(cb) {

t.fire(‘a400′, cb, 400)

},

b: function(cb) {

t.fire(‘c300′, cb, 300)

}

},

function(err, results) {

log(’3 err: ‘, err); *// -> undefined*

log(’3 results: ‘, results); *// -> { b: ‘c300′, a: ‘a400′ }*

});

更详细示例参见：https://github.com/freewind/async\_demo/blob/master/parallel.js

##### waterfall(tasks, [callback]) （多个函数依次执行，且前一个的输出为后一个的输入）

与seires相似，按顺序依次执行多个函数。不同之处，每一个函数产生的值，都将传给下一个函数。如果中途出错，后面的函数将不会被执行。错误信息以及之前产生的结果，将传给waterfall最终的callback。   
这个函数名为waterfall(瀑布)，可以想像瀑布从上到下，中途冲过一层层突起的石头。注意，该函数不支持json格式的tasks。

async.waterfall([function(cb) {

log('1: ', 'start');

cb(null, 3);

},

function(n, cb) {

log('2: ', n);

t.inc(n, cb);

},

function(n, cb) {

log('3: ', n);

t.fire(n \* n, cb);

}],

function(err, result) {

log(’1 err: ‘, err); // -> null

log(’1 result: ‘, result); // -> 16

});

更详细示例参见：https://github.com/freewind/async\_demo/blob/master/waterfall.js

##### auto(tasks, [callback]) （多个函数有依赖关系，有的并行执行，有的依次执行）

用来处理有依赖关系的多个任务的执行。比如某些任务之间彼此独立，可以并行执行；但某些任务依赖于其它某些任务，只能等那些任务完成后才能执行。   
虽然我们可以使用async.parallel和async.series结合起来实现该功能，但如果任务之间关系复杂，则代码会相当复杂，以后如果想添加一个新任务，也会很麻烦。这时使用async.auto，则会事半功倍。   
如果有任务中途出错，则会把该错误传给最终callback，所有任务（包括已经执行完的）产生的数据将被忽略。   
这里假设我要写一个程序，它要完成以下几件事：   
从某处取得数据   
在硬盘上建立一个新的目录   
将数据写入到目录下某文件   
发送邮件，将文件以附件形式发送给其它人。   
分析该任务，可以知道1与2可以并行执行，3需要等1和2完成，4要等3完成。

async.auto({

getData: function(callback) {

setTimeout(function() {

console.log(’1 : got data’);

callback();

},

300);

},

makeFolder: function(callback) {

setTimeout(function() {

console.log(’1 : made folder’);

callback();

},

200);

},

writeFile: ['getData', 'makeFolder',

function(callback) {

setTimeout(function() {

console.log('1: wrote file');

callback(null, 'myfile');

},

300);

}],

emailFiles: ['writeFile',

function(callback, results) {

log('1: emailed file: ', results.writeFile); *// -> myfile*

callback(null, results.writeFile);

}]

},

function(err, results) {

log(’1 : err: ‘, err); *// -> null*

log(’1 : results: ‘, results); *// -> { makeFolder: undefined,*

*// getData: undefined,*

*// writeFile: ‘myfile’,*

*// emailFiles: ‘myfile’ }*

});

更多详细示例参见：https://github.com/freewind/async\_demo/blob/master/auto.js

##### whilst(test, fn, callback)（用可于异步调用的while）

相当于while，但其中的异步调用将在完成后才会进行下一次循环。举例如下：

var count1 = 0;

async.whilst(function() {

return count1 < 3

},

function(cb) {

log(’1 count: ‘, count1);

count1++;

setTimeout(cb, 1000);

},

function(err) {

// 3s have passed

log(’1 err: ‘, err); // -> undefined

});

它相当于：

try {

whilst(test) {

fn();

}

callback();

} catch(err) {

callback(err);

}

该函数的功能比较简单，条件变量通常定义在外面，可供每个函数访问。在循环中，异步调用时产生的值实际上被丢弃了，因为最后那个callback只能传入错误信息。   
另外，第二个函数fn需要能接受一个函数cb，这个cb最终必须被执行，用于表示出错或正常结束。   
更详细示例参见：https://github.com/freewind/async\_demo/blob/master/whilst\_until.js

##### until(test, fn, callback) （与while相似，但判断条件相反）

var count4 = 0;

async.until(function() {

return count4 > 3

},

function(cb) {

log(’4 count: ‘, count4);

count4++;

setTimeout(cb, 200);

},

function(err) {

// 4s have passed

log(’4 err: ‘, err); // -> undefined

});

当第一个函数条件为false时，继续执行第二个函数，否则跳出。

##### queue （可设定worker数量的队列）

queue相当于一个加强版的parallel，主要是限制了worker数量，不再一次性全部执行。当worker数量不够用时，新加入的任务将会排队等候，直到有新的worker可用。   
该函数有多个点可供回调，如worker用完时、无等候任务时、全部执行完时等。   
定义一个queue，其worker数量为2，并在任务执行时，记录一下日志：

var q = async.queue(function(task, callback) {

log(‘worker is processing task: ‘, task.name);

task.run(callback);

}, 2);

worker数量将用完时，会调用saturated函数：

q.saturated = function() {

log(‘all workers to be used’);

}

当最后一个任务交给worker执行时，会调用empty函数

q.empty = function() {

log(‘no more tasks wating’);

}

当所有任务都执行完时，会调用drain函数

q.drain = function() {

console.log(‘all tasks have been processed’);

}

放入多个任务，可一次放一个，或一次放多个

q.push({

name: ’t1′,

run: function(cb) {

log(‘t1 is running, waiting tasks: ‘, q.length());

t.fire(‘t1′, cb, 400); *// 400ms后执行*

}

},

function(err) {

log(‘t1 executed’);

});

q.push([{

name: 't3',

run: function(cb) {

log('t3 is running, waiting tasks: ', q.length());

t.fire('t3', cb, 300); *// 300ms后执行*

}

},

{

name: 't4',

run: function(cb) {

log('t4 is running, waiting tasks: ', q.length());

t.fire('t4', cb, 500); *// 500ms后执行*

}

}],

function(err) {

log(‘t3 / 4 executed’);

});

更多详细示例参见：https://github.com/freewind/async\_demo/blob/master/queue.js

##### iterator(tasks) （将几个函数包装为iterator）

将一组函数包装成为一个iterator，可通过next()得到以下一个函数为起点的新的iterator。该函数通常由async在内部使用，但如果需要时，也可在我们的代码中使用它。

var iter = async.iterator([function() {

console.log('111')

},

function() {

console.log('222')

},

function() {

console.log('333')

}]);

console.log(iter());

console.log(iter.next());

直接调用()，会执行当前函数，并返回一个由下个函数为起点的新的iterator。调用next()，不会执行当前函数，直接返回由下个函数为起点的新iterator。   
对于同一个iterator，多次调用next()，不会影响自己。如果只剩下一个元素，调用next()会返回null。   
更详细示例参见：https://github.com/freewind/async\_demo/blob/master/iterator.js

##### apply(function, arguments..) （给函数预绑定参数）

apply是一个非常好用的函数，可以让我们给一个函数预绑定多个参数并生成一个可直接调用的新函数，简化代码。   
对于函数：

function(callback) { t.inc(3, callback); }

可以用apply改写为：

async.apply(t.inc, 3);

还可以给某些函数预设值，得到一个新函数：

var log = async.apply(console.log, ">");

log(‘hello’);

*// > hello*

更详细代码参见：https://github.com/freewind/async\_demo/blob/master/apply.js

##### nextTick(callback) （在nodejs与浏览器两边行为一致）

nextTick的作用与nodejs的nextTick一样，都是把某个函数调用放在队列的尾部。但在浏览器端，只能使用setTimeout(callback,0)，但这个方法有时候会让其它高优先级的任务插到前面去。   
所以提供了这个nextTick，让同样的代码在服务器端和浏览器端表现一致。

var calls = [];

async.nextTick(function() {

calls.push(‘two’);

});

calls.push(‘one’);

async.nextTick(function() {

console.log(calls); *// -> [ 'one', 'two' ]*

})

### 文件系统

Node.js 文件系统

Node.js 提供一组类似 UNIX（POSIX）标准的文件操作API。 Node 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)。

建议大家是用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞。

实例

<p>创建 input.txt 文件，内容如下：</p>

<pre>

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

接下来，让我们来具体了解下 Node.js 文件系统的方法。

#### 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

参数使用说明如下：

* **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：

|  |  |
| --- | --- |
| **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'，但是如果文件路径不存在，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+'， 但是如果文件路径不存在，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'， 但是如果文件路径不存在，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+'， 但是如果文件路径不存在，则文件读取追加失败。 |

实例

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！

#### 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

参数使用说明如下：

* **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：

|  |  |
| --- | --- |
| **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。 |

实例

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

#### 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

参数使用说明如下：

* **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。

实例

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

#### 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

参数使用说明如下：

* **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。

实例

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

#### 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

参数使用说明如下：

* **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

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

#### 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

参数使用说明如下：

* **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。

实例

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

语法

以下为删除文件的语法格式：

fs.unlink(path, callback)

参数

参数使用说明如下：

* **path** - 文件路径。
* **callback** - 回调函数，没有参数。

实例

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

语法

以下为创建目录的语法格式：

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

参数

参数使用说明如下：

* **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

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

语法

以下为读取目录的语法格式：

fs.readdir(path, callback)

参数

参数使用说明如下：

* **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

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

语法

以下为删除目录的语法格式：

fs.rmdir(path, callback)

参数

参数使用说明如下：

* **path** - 文件路径。
* **callback** - 回调函数，没有参数。

实例

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

……

#### 文件模块方法参考手册

以下为 Node.js 文件模块相同的方法列表：

|  |  |
| --- | --- |
| **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。 |

### Get post

Node.js 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使用GET/POST请求。

本章节我们将为大家介绍 Node.js GET/POST请求。

#### 获取GET请求内容 - url.parse

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，因此你可以手动解析后面的内容作为GET请求的参数。

node.js中url模块中的parse函数提供了这个功能。

var http = require('http');

var url = require('url');

var util = require('util');

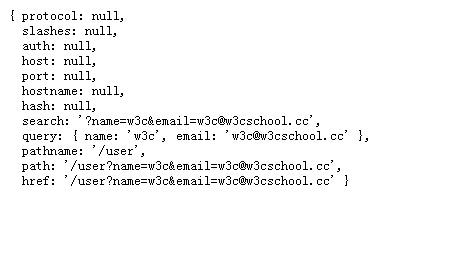
http.createServer(function(req, res){

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

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

}).listen(3000);

在浏览器中访问http://localhost:3000/user?name=w3c&email=w3c@w3cschool.cc 然后查看返回结果:



#### 获取POST请求内容

POST请求的内容全部的都在请求体中，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可能是一件耗时的工作。

比如上传文件，而很多时候我们可能并不需要理会请求体的内容，恶意的POST请求会大大消耗服务器的资源，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');

var util = require('util');

http.createServer(function(req, res){

var post = ''; //定义了一个post变量，用于暂存请求体的信息

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

post += chunk;

});

req.on('end', function(){ //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。

post = querystring.parse(post);

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

});

}).listen(3000);

### Node.js REPL(交互式解释器)

Node.js REPL(交互式解释器)

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

Node 自带了交互式解释器，可以执行以下任务：

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

Node 的交互式解释器可以很好的调试 Javascript 代码。

开始学习 REPL

我们可以输入以下命令来启动 Node 的终端：

$ node

>

这时我们就可以在 > 后输入简单的表达式，并按下回车键来计算结果。

#### 简单的表达式运算

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

$ node

> 1 +4

5

> 5 / 2

2.5

> 3 \* 6

18

> 4 - 1

3

> 1 + ( 2 \* 3 ) - 4

3

>

#### 使用变量

你可以将数据存储在变量中，并在你需要的使用它。

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 Workd

undefined

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

www.runoob.com

undefined

#### 多行表达式

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

$ node

> var x = 0

undefined

> do {

... x++;

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

... } while ( x < 5 );

x: 1

x: 2

x: 3

x: 4

x: 5

undefined

>

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

#### 下划线(\_)变量

你可以使用下划线(\_)获取表达式的运算结果：

$ node

> var x = 10

undefined

> var y = 20

undefined

> x + y

30

> var sum = \_

undefined

> console.log(sum)

30

undefined

>

#### REPL 命令

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

#### 停止 REPL

前面我们已经提到按下两次 **ctrl + c** 建就能退出 REPL:

$ node

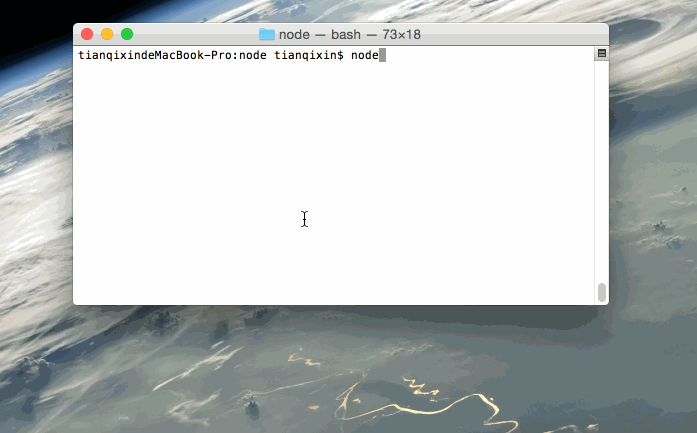
>

(^C again to quit)

>

#### Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：



### Buffer

#### 介绍

#### Buffer结构

Buffer是一个典型的Javascript和C++结合的模块，性能相关部分用C++实现，非性能相关部分用javascript实现。

Node在进程启动时Buffer就已经加装进入内存，并将其放入全局对象，因此无需require

Buffer对象：类似于数组，其元素是16进制的两位数。

**Buffer内存分配**

Buffer对象的内存分配不是在V8的堆内存中，在Node的C++层面实现内存的申请。

为了高效的使用申请来得内存，Node中采用slab分配机制，slab是一种动态内存管理机制，应用各种\*nix操作系统。slab有三种状态：

(1) full：完全分配状态

(2) partial：部分分配状态

(3) empty：没有被分配状态

JavaScript很擅长处理字符串，但是因为它最初的设计是用来处理HTML文档，因此它并不太擅长处理二进制数据。JavaScript没有byte类型，没有结构化的类型（structured types），甚至没有字节数组，只有数字和字符串。（原文：JavaScript doesn't have a byte type — it just has numbers — or structured types, or http://skylitecellars.com/ even byte arrays: It just has strings.）

因为Node基于JavaScript，它自然可以处理类似HTTP这样的文本协议，但是你也可以用它来跟数据库交互，处理图片或文件上传等，可以想象，如果仅仅用字符串来做这些事得有多困难。早些时候，Node通过将byte编码成文本字符来处理二进制数据，但这种方式后来被证明并不可行，既浪费资源，又缓慢，又不灵活，而且难以维护。

Node有一个二进制缓冲实现Buffer，这个伪类（pseudo-class）提供了一系列处理二进制数据的API，简化了那些需要处理二进制数据的任务。缓冲的长度由字节数据的长度决定，而且你可以随机的设置和获取缓冲内的字节数据。

注意：Buffer类有一个特殊的地方，缓冲内的字节数据所占用的内存不是分配在JavaScrp

It VM内存堆上的，也就是说这些对象不会被JavaScript的垃圾回收算法处理，取而代之的是一个不会被修改的永久内存地址，这也避免了因缓冲内容的内存复制所造成的CPU浪费。

#### 创建缓冲

你可以用一个UTF-8字符串创建缓冲，像这样：

复制代码 代码如下:

var buf = new Buffer(‘Hello World!');

也可以用指定编码的字符串创建缓冲：

复制代码 代码如下:

var buf = new Buffer('8b76fde713ce', 'base64');

可接受的字符编码和标识如下：

1.ascii——ASCI，仅适用于ASCII字符集。  
2.utf8——UTF-8，这种可变宽编码适用于Unicode字符集的任何字符，它已经成了Web世界的首选编码，也是Node的默认编码类型。  
3.base64——Base64，这种编码基于64个可打印ASCII字符来表示二进制数据，Base64通常用于在字符文档内嵌入可以被转化成字符串的二进制数据，在需要时又可以完整无损的转换回原来的二进制格式。

如果没有数据来初始化缓冲，可以用指定的容量大小来创建一个空缓冲：

复制代码 代码如下:

var buf = new Buffer(1024); // 创建一个1024字节的缓冲

#### 获取和设置缓冲数据

创建或接收一个缓冲对象后，你可能要查看或者修改它的内容，可以通过[]操作符来访问缓冲的某个字节：

复制代码 代码如下:

var buf = new Buffer('my buffer content');  
// 访问缓冲内第10个字节  
console.log(buf[10]); // -> 99

注意：当你（使用缓冲容量大小来）创建一个已初始化的缓冲时，一定要注意，缓冲的数据并没有被初始化成0，而是随机数据。

复制代码 代码如下:

var buf = new Buffer(1024);

console.log(buf[100]); // -> 5 (某个随机值)

你可以这样修改缓冲里任何位置的数据：

复制代码 代码如下:

buf[99] = 125; // 把第100个字节的值设置为125

注意：在某些情况下，一些缓冲操作并不会产生错误，比如：

1.缓冲内的字节最大值为255，如果某个字节被赋予大于256的数字，将会用256对其取模，然后将结果赋给这个字节。  
2.如果将缓冲的某个字节赋值为256，它的实际值将会是0（译者注：其实跟第一条重复，256%256=0）  
3.如果用浮点数给缓冲内某个字节赋值，比如100.7，实际值将会是浮点数的整数部分——100  
4.如果你尝试给一个超出缓冲容量的位置赋值，赋值操作将会失败，缓冲不做任何修改。

你可以用length属性获取缓冲的长度：

复制代码 代码如下:

var buf = new Buffer(100);

console.log(buf.length); // -> 100

还可以使用缓冲长度迭代缓冲的内容，来读取或设置每个字节：

复制代码 代码如下:

var buf = new Buffer(100);

for(var i = 0; i < buf.length; i++) {

    buf[i] = i;

}

上面代码新建了一个包含100个字节的缓冲，并从0到99设置了缓冲内每个字节。

#### 切分缓冲数据

一旦创建或者接收了一个缓冲，你可能需要提取缓冲数据的一部分，可以通过指定起始位置来切分现有的缓冲，从而创建另外一个较小的缓冲：

复制代码 代码如下:

var buffer = new Buffer("this is the content of my buffer");

var smallerBuffer = buffer.slice(8, 19);

console.log(smallerBuffer.toString()); // -> "the content"

注意，当切分一个缓冲的时候并没有新的内存被分配或复制，新的缓冲使用父缓冲的内存，它只是父缓冲某段数据（由起始位置指定）的引用。这段话含有几个意思。

首先，如果你的程序修改了父缓冲的内容，这些修改也会影响相关的子缓冲，因为父缓冲和子缓冲是不同的JavaScript对象，因此很容易忽略这个问题，并导致一些潜在的bug。

其次，当你用这种方式从父缓冲创建一个较小的子缓冲时，父缓冲对象在操作结束后依然会被保留，并不会被垃圾回收，如果不注意的话，很容易会造成内存泄露。

注意：如果你担心因此产生内存泄露问题，你可以使用copy方法来替代slice操作，下面将会介绍copy。

#### 复制缓冲数据

你可以像这样用copy将缓冲的一部分复制到另外一个缓冲：

复制代码 代码如下:

var buffer1 = new Buffer("this is the content of my buffer");

var buffer2 = new Buffer(11);

var targetStart = 0;

var sourceStart = 8;

var sourceEnd = 19;

buffer1.copy(buffer2, targetStart, sourceStart, sourceEnd);

console.log(buffer2.toString()); // -> "the content"

上面代码，复制源缓冲的第9到20个字节到目标缓冲的开始位置。

#### 解码缓冲数据

缓冲数据可以这样转换成一个UTF-8字符串：

复制代码 代码如下:

var str = buf.toString();

还可以通过指定编码类型来将缓冲数据解码成任何编码类型的数据。比如，你想把一个缓冲解码成base64字符串，可以这么做：

复制代码 代码如下:

var b64Str = buf.toString("base64");

使用toString函数，你还可以把一个UTF-8字符串转码成base64字符串：

复制代码 代码如下:

var utf8String = 'my string';

var buf = new Buffer(utf8String);

var base64String = buf.toString('base64')

**小结**

有时候，你不得不跟二进制数据打交道，但是原生JavaScript又没有明确的方式来做这件事，于是Node提供了Buffer类，封装了一些针对连续内存块的操作。你可以在两个缓冲之间切分或复制内存数据。

你也可以把一个缓冲转换成某种编码的字符串，或者反过来，把一个字符串转化成缓冲，来访问或处理每个bit。

#### Buffer tostring 方法

将buffer对象转换成指定的字符编码的字符串。

**语法：**

复制代码 代码如下:

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

**接收参数：**

encoding       转换成字符串后的字符编码，默认为 ‘utf8′

start                buffer 转换的起始位置，默认为 0

end                 buffer 转换的结束位置，默认为buffer长度

**例子：**

复制代码 代码如下:

var b = new Buffer(50);  
   
console.log(b);  
   
var c = b.toString('base64',0,10);  
   
console.log(c);

**源码：**

复制代码 代码如下:

// toString(encoding, start=0, end=buffer.length)  
Buffer.prototype.toString = function(encoding, start, end) {  
  var loweredCase = false;  
  start = start >>> 0;  
  end = util.isUndefined(end) ? this.length : end >>> 0;  
  if (!encoding) encoding = 'utf8';  
  if (start < 0) start = 0;  
  if (end > this.length) end = this.length;  
  if (end <= start) return '';  
  while (true) {  
    switch (encoding) {  
      case 'hex':  
        return this.hexSlice(start, end);  
      case 'utf8':  
      case 'utf-8':  
        return this.utf8Slice(start, end);  
      case 'ascii':  
        return this.asciiSlice(start, end);  
      case 'binary':  
        return this.binarySlice(start, end);  
      case 'base64':  
        return this.base64Slice(start, end);  
      case 'ucs2':  
      case 'ucs-2':  
      case 'utf16le':  
      case 'utf-16le':  
        return this.ucs2Slice(start, end);  
      default:  
        if (loweredCase)  
          throw new TypeError('Unknown encoding: ' + encoding);  
        encoding = (encoding + '').toLowerCase();  
        loweredCase = true;  
    }  
  }  
};

#### Buffer write

**方法说明：**

将string使用指定的encoding写入到buffer的offset处。

返回写入了多少个八进制字节。

如果Buffer没有足够的空间来适应整个string,那么将只有string的部分被写入。

**语法：**

复制代码 代码如下:

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

**接收参数：**

string                  String,被写入buffer的数据.  
offet                   number,可选,默认0.数据写入到buffer的位置.  
length                Number,可选,默认:buffer.length – offset,要写入数据的长度  
encoding           String,需要使用的编码格式,可选,默认为”utf8″.

**例子：**

复制代码 代码如下:

buf = new Buffer(256);  
   
len = buf.write('\u00bd + \u00bc = \u00be', 0);  
   
console.log(len + " bytes: " + buf.toString('utf8', 0, len));

**源码：**

复制代码 代码如下:

Buffer.prototype.write = function(string, offset, length, encoding) {  
  // allow write(string, encoding)  
  if (util.isString(offset) && util.isUndefined(length)) {  
    encoding = offset;  
    offset = 0;  
  // allow write(string, offset[, length], encoding)  
  } else if (isFinite(offset)) {  
    offset = ~~offset;  
    if (isFinite(length)) {  
      length = ~~length;  
    } else {  
      encoding = length;  
      length = undefined;  
    }  
  // XXX legacy write(string, encoding, offset, length) - remove in v0.13  
  } else {  
    if (!writeWarned) {  
      if (process.throwDeprecation)  
        throw new Error(writeMsg);  
      else if (process.traceDeprecation)  
        console.trace(writeMsg);  
      else  
        console.error(writeMsg);  
      writeWarned = true;  
    }  
    var swap = encoding;  
    encoding = offset;  
    offset = ~~length;  
    length = swap;  
  }  
  var remaining = this.length - offset;  
  if (util.isUndefined(length) || length > remaining)  
    length = remaining;  
  encoding = !!encoding ? (encoding + '').toLowerCase() : 'utf8';  
  if (string.length > 0 && (length < 0 || offset < 0))  
    throw new RangeError('attempt to write beyond buffer bounds');  
  var ret;  
  switch (encoding) {  
    case 'hex':  
      ret = this.hexWrite(string, offset, length);  
      break;  
    case 'utf8':  
    case 'utf-8':  
      ret = this.utf8Write(string, offset, length);  
      break;  
    case 'ascii':  
      ret = this.asciiWrite(string, offset, length);  
      break;  
    case 'binary':  
      ret = this.binaryWrite(string, offset, length);  
      break;  
    case 'base64':  
      // Warning: maxLength not taken into account in base64Write  
      ret = this.base64Write(string, offset, length);  
      break;  
    case 'ucs2':  
    case 'ucs-2':  
    case 'utf16le':  
    case 'utf-16le':  
      ret = this.ucs2Write(string, offset, length);  
      break;  
    default:  
      throw new TypeError('Unknown encoding: ' + encoding);  
  }  
  return ret;  
};

#### buffer.fill

**方法说明：**

将制定数据填充到buffer中。

**语法：**

复制代码 代码如下:

buffer.fill(value, [offset], [end])

**接收参数：**

value           将要填充的数据

offet           填充数据的开始位置，不指定默认为 0

end            填充数据的结束位置，不指定默认为 buffer 的 长度。

**例子：**

复制代码 代码如下:

//例子1中，不指定填充内容的起止位置  
var b = new Buffer(50);  
b.fill("h");  
   
//例子2中，指定了填充内容的起止位置  
var b = new Buffer(50);  
var len = b.length;  
b.fill("h" , len-1 , len);

#### buffer.Buffer.byteLength

**方法说明：**

获取字符串的字节长度。

这个函数与 String.prototype.length 不同点在于，后者返回的是字符串的字符数。

**语法：**

复制代码 代码如下:

Buffer.byteLength(string, [encoding])

**接收参数：**

string                              字符创  
encoding                        字符串编码，默认为 ‘utf8′

**例子：**

复制代码 代码如下:

str = '\u00bd + \u00bc = \u00be';  
console.log(str + ": " + str.length + " characters, " +  
  Buffer.byteLength(str, 'utf8') + " bytes");  
// ½ + ¼ = ¾: 9 characters, 12 bytes

**源码：**

复制代码 代码如下:

Buffer.byteLength = function(str, enc) {  
  var ret;  
  str = str + '';  
  switch (enc) {  
    case 'ascii':  
    case 'binary':  
    case 'raw':  
      ret = str.length;  
      break;  
    case 'ucs2':  
    case 'ucs-2':  
    case 'utf16le':  
    case 'utf-16le':  
      ret = str.length \* 2;  
      break;  
    case 'hex':  
      ret = str.length >>> 1;  
      break;  
    default:  
      ret = internal.byteLength(str, enc);  
  }  
  return ret;  
};

Buffer的转换

Buffer对象可以和字符串相互转换，支持的编码类型如下：

ASCII、UTF-8、UTF-16LE/UCS-2、Base64、Binary、Hex

#### **字符串转Buffer**

new Buffer(str, [encoding])，默认UTF-8

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

#### **Buffer转字符串**

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

**Buffer不支持的编码类型**

通过Buffer.isEncoding(encoding)判断是否支持

iconv-lite：纯JavaScript实现，更轻量，性能更好无需C++到javascript的转换

iconv：调用C++的libiconv库完成

#### Buffer的拼接

注意res.on('data', function(chunk) {})，其中的参数chunk是Buffer对象，直接用+拼接会自动转换为字符串，对于宽字节字符可能会导致乱码产生，解决方法

(1) 通过可读流中的setEncoding()方法，该方法可以让data事件传递不再是Buffer对象，而是编码后的字符串，其内部使用了StringEncoder模块。

(2) 将Buffer对象暂存到数组中，最后在组装成一个大Buffer让后编码转换为字符串输出

Buffer在文件I/O和网络I/O中广泛应用，其性能举足轻重，比普通字符串性能要高出很多。

Buffer的使用除了与字符串的转换有性能损耗外，在文件读取时候，有一个highWaterMark设置对性能影响至关重要。

a. highWaterMark设置对Buffer内存的分配和使用有一定影响

b. highWaterMark设置过小，可能导致系统调用次数过多

#### buffer介绍另一篇文章

**一，开篇分析**

所谓缓冲区Buffer，就是 "临时存贮区" 的意思，是暂时存放输入输出数据的一段内存。

JS语言自身只有字符串数据类型，没有二进制数据类型，因此NodeJS提供了一个与String对等的全局构造函数Buffer来提供对二进制数据的操作。除了可以读取文件得到Buffer的实例外，还能够直接构造，例如：

　var buffer = new Buffer([ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]) ;

Buffer与字符串类似，除了可以用.length属性得到字节长度外，还可以用[index]方式读取指定位置的字节，例如：

buffer[0] ; // 0x68;

Buffer与字符串能够互相转化，例如可以使用指定编码将二进制数据转化为字符串：

var str = buffer.toString("utf-8");  // hello

将字符串转换为指定编码下的二进制数据：

var buffer= new Buffer("hello", "utf-8") ; // <Buffer 68 65 6c 6c 6f>

一点儿区别：

Buffer与字符串有一个重要区别。字符串是只读的，并且对字符串的任何修改得到的都是一个新字符串，原字符串保持不变。

至于Buffer，更像是可以做指针操作的C语言数组。例如，可以用[index]方式直接修改某个位置的字节。

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

slice方法也不是返回一个新的Buffer，而更像是返回了指向原Buffer中间的某个位置的指针，如下所示。

[ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]  
    ^           ^  
    |           |  
   bin     bin.slice(2)  
因此对slice方法返回的Buffer的修改会作用于原Buffer，例如：

 var buffer= new Buffer([ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]) ;  
 var sub = bin.slice(2) ;  
 sub[0] = 0x65 ;  
 console.log(buffer) ; //  <Buffer 68 65 65 6c 6f>

如果想要拷贝一份Buffer，得首先创建一个新的Buffer，并通过.copy方法把原Buffer中的数据复制过去。

这个类似于申请一块新的内存，并把已有内存中的数据复制过去。以下是一个例子。

 var buffer= new Buffer([ 0x68, 0x65, 0x6c, 0x6c, 0x6f ]) ;  
 var dup = new Buffer(bin.length) ;  
 buffer.copy(dup) ;  
 dup[0] = 0x48 ;  
 console.log(buffer) ;  // <Buffer 68 65 6c 6c 6f>  
 console.log(dup) ;  // <Buffer 48 65 65 6c 6f>

总之，Buffer将JS的数据处理能力从字符串扩展到了任意二进制数据。

以上简单让大家了解一下什么是Buffer，下面具体说说如何使用和具体使用场景。

**二，聊聊Buffer**

JavaScript对字符串处理十分友好，无论是宽字节还是单字节字符串，都被认为是一个字符串。Node中需要处理网络协议、操作数据库、处理图片、文件上传等，还需要处理大量二进制数据，自带的字符串远不能满足这些要求，因此Buffer应运而生。

Buffer结构

Buffer是一个典型的Javascript和C++结合的模块，性能相关部分用C++实现，非性能相关部分用javascript实现。

Node在进程启动时Buffer就已经加装进入内存，并将其放入全局对象，因此无需require

Buffer对象：类似于数组，其元素是16进制的两位数。

Buffer内存分配

Buffer对象的内存分配不是在V8的堆内存中，在Node的C++层面实现内存的申请。

为了高效的使用申请来得内存，Node中采用slab分配机制，slab是一种动态内存管理机制，应用各种\*nix操作系统。slab有三种状态：

(1) full：完全分配状态

(2) partial：部分分配状态

(3) empty：没有被分配状态

**Buffer的转换**  
   
Buffer对象可以和字符串相互转换，支持的编码类型如下：

ASCII、UTF-8、UTF-16LE/UCS-2、Base64、Binary、Hex

**字符串转Buffer**

new Buffer(str, [encoding])，默认UTF-8  
buf.write(string, [offset], [length], [encoding])

**Buffer转字符串**

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

Buffer不支持的编码类型

通过Buffer.isEncoding(encoding)判断是否支持

iconv-lite：纯JavaScript实现，更轻量，性能更好无需C++到javascript的转换

iconv：调用C++的libiconv库完成

**Buffer的拼接**

注意 "res.on('data', function(chunk) {})"，其中的参数chunk是Buffer对象，直接用+拼接会自动转换为字符串，对于宽字节字符可能会导致乱码产生，

**解决方法：**

(1) 通过可读流中的setEncoding()方法，该方法可以让data事件传递不再是Buffer对象，而是编码后的字符串，其内部使用了StringEncoder模块。

(2) 将Buffer对象暂存到数组中，最后在组装成一个大Buffer让后编码转换为字符串输出。

Buffer在文件I/O和网络I/O中广泛应用，其性能举足轻重，比普通字符串性能要高出很多。

Buffer的使用除了与字符串的转换有性能损耗外，在文件读取时候，有一个highWaterMark设置对性能影响至关重要。

a，highWaterMark设置对Buffer内存的分配和使用有一定影响。

b， highWaterMark设置过小，可能导致系统调用次数过多。

什么时候该用buffer，什么时候不该用  ------ 纯粹的javascript支持unicode码而对二进制不是很支持，当解决TCP流或者文件流的时候，处理流是有必要的，我们保存非utf-8字符串，2进制等等其他格式的时候，我们就必须得使用 ”Buffer“ 。

**三，实例引入**

 var buf = new Buffer("this is text concat test !") ,str = "this is text concat test !" ;  
 console.time("buffer concat test !");  
 var list = [] ;  
 var len = 100000 \* buf.length ;  
 for(var i=0;i<100000;i++){  
     list.push(buf) ;  
     len += buf.length ;  
 }  
 var s1 = Buffer.concat(list, len).toString() ;  
 console.timeEnd("buffer concat test !") ;  
 console.time("string concat test !") ;  
 var list = [] ;  
 for (var i = 100000; i >= 0; i--) {  
   list.push(str) ;  
 }  
 var s2 = list.join("") ;  
 console.timeEnd("string concat test !") ;

以下是运行结果：

http://www.daxueit.com/upload/201506/01/2015010717121332.jpg

读取速度肯定string更快，buffer还需要toString()的操作。 所以我们在保存字符串的时候，该用string还是要用string，就算大字符串拼接string的速度也不会比buffer慢。

那什么时候我们又需要用buffer呢？没办法的时候，当我们保存非utf-8字符串，2进制等等其他格式的时候，我们就必须得使用了。

**四，总结一下**

（1），JavaScript适合处理Unicode编码数据，但对二进制数据的处理并不友好。  
（2），所以处理TCP流或文件系统时，对八位字节流的处理很有必要。  
（3），Node有几个用于处理，创建和消耗八位字节流的方法。  
（4），原始数据存放在一个Buffer实例中，一个Buffer类似一个整数数组，但是它的内存，分配在V8堆栈外。一个Buffer的大小是不能更改的。  
（5），处理的编码类型有：ascii,utf8,utf16le,ucs2（utf16le的别名）,base64,binary,hex。  
（6），Buffer为全局元素，直接new Buffer()就得到一个Buffer实例。

### Node.js Stream(流)

#### Stream就是事件对象的实例 – 非常重要

Stream 是一个抽象接口，Node 中有很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）。

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。

本教程会为大家介绍常用的流操作。

#### 从流中读取数据

创建 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");

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

#### 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。

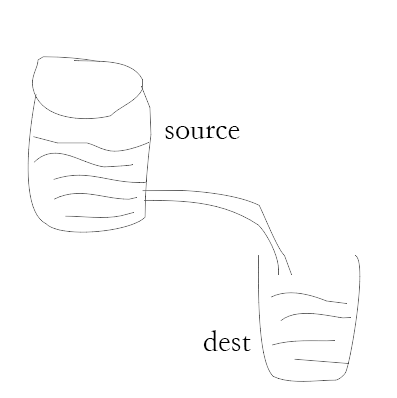
查看 output.txt 文件的内容：

$ cat output.txt

菜鸟教程官网地址：www.runoob.com

#### 管道流

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。



如上面的图片所示，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个桶，这样就慢慢的实现了大文件的复制过程。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

菜鸟教程官网地址：www.runoob.com

管道流操作实例

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");

代码执行结果如下：

$ node main.js

程序执行完毕

查看 output.txt 文件的内容：

$ cat output.txt

菜鸟教程官网地址：www.runoob.com

管道流操作实例

#### 链式流 – 就是链式访问+管道流

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很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。");

代码执行结果如下：

$ node compress.js

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。");

代码执行结果如下：

$ node decompress.js

文件解压完成。

### 路由编程

#### 请求地址基础知识- url querystring pathname – 重要

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。这一功能应当属于路由还是服务器（甚至作为一个模块自身的功能）确实值得探讨，但这里暂定其为我们的HTTP服务器的功能。

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。但是为了解析这些数据，我们需要额外的Node.JS模块，它们分别是url和querystring模块。

url.parse(string).query

|

url.parse(string).pathname |

| |

| |

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

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

--- -----

| |

| |

querystring(string)["foo"] |

|

querystring(string)["hello"]

### Nodejs内置服务器介绍

#### 搭建

我们先来实现一个简单的例子，hello world。

似乎每种语言教程的第一节都会讲这个，我们也不例外。

首先我们先创建一个项目目录，目录可自己定义，本案例的目录为 e:/nodetest/。

由于我们要搭建的是服务器，所以我把第一个文件命名为server.js。

在server.js里面输入以下代码：

var http = require("http");   
   
http.createServer(function(request, response) {   
   
    response.writeHead(200, {"Content-Type": "text/plain"});   
   
    response.write("Hello World");   
   
    response.end();   
   
}).listen(8888);

接着我们打开cmd。

用cd e:/nodetest/定位到项目目录下，然后执行 node server.js 命令，即可运行该文件；

然后打开浏览器访问 http://localhost:8888/，你会看到一个写着“Hello World”的网页；

其实这就是一个简单的可工作的服务器了，只是简单到什么都干不了，不过不要紧，跟着我一步一步来，我会教你怎么搭建一个完整可用的服务器。

下一节我们来分析一下这段代码的构成。

#### http模块介绍

紧接上一节，我们来分析一下代码：

第一行请求（require）Node.js自带的 http 模块，并且把它赋值给 http 变量。

接下来我们调用http模块提供的函数： createServer 。

这个函数会返回一个对象，这个对象有一个叫做 listen 的方法，这个方法有一个数值参数，指定这个HTTP服务器监听的端口号。

为了提高可读性，我们来改一下这段代码。

**原来的代码：**

var http = require("http");  
http.createServer(function(request, response) {   
 response.writeHead(200, {"Content-Type": "text/plain"});   
 response.write("Hello World");   
 response.end();   
}).listen(8888);

**可以改写成：**

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);

我们定义了一个onRequest()函数，并将它作为参数传给createServer，类似回调函数。

我们给某个方法传递了一个函数，这个方法在有相应事件发生时调用这个函数来进行回调，我们把这叫做基于事件驱动的回调。

接下来我们看一下onRequest() 的主体部分，当回调启动，我们的 onRequest() 函数被触发的时候，有两个参数被传入： request 和 response 。

request ： 收到的请求信息；

response ： 收到请求后做出的响应。

所以这段代码所执行的操作就是：

当收到请求时，

1、使用 response.writeHead() 函数发送一个HTTP状态200 和 HTTP头的内容类型（content-type）

2、使用 response.write() 函数在HTTP相应主体中发送文本“Hello World”。

3、调用 response.end() 完成响应。

### NODEJS路由

## Nodejs进阶

### 进程process

虽然node对操作系统做了很多抽象的工作，但是你还是可以直接和他交互，比如和系统中已经存在的进程进行交互，创建工作子进程。node是一个用于事件循环的线程，但是你可以在这个事件循环之外创建其他的进程（线程）参与工作。

#### 进程模块

　　process模块允许你获得或者修改当前node进程的设置，不想其他的模块，**process是一个全局进程(node主进程)**，你可以直接通过process变量直接访问它。

　　process实现了EventEmitter接口，exit方法会在当进程退出的时候执行。因为进程退出之后将不再执行事件循环，所有只有那些没有回调函数的代码才会被执行。在下面例子中，setTimeout里面的语句是没有办法执行到的。

[复制代码](javascript:void(0);)

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

　　setTimeout(function () {

　　　　console.log('This will not run');

　　}, 100);

　　console.log('Bye.');

});

[复制代码](javascript:void(0);)

　　在你接触node之后，你就会发现那些影响了主事件循环的异常会把整个node进程宕掉的。这会是相当严重的问题，所以process提供了另外一个有用的事件uncaughtException来解决这个问题，他会把异常抓取出来供你处理。

[复制代码](javascript:void(0);)

复制代码

process.on('uncaughtException', function (err) {

　　console.log('Caught exception: ' + err);

});

setTimeout(function () {

　　console.log('This will still run.');

}, 500);

// Intentionally cause an exception, but don't catch it.

nonexistentFunc();

console.log('This will not run.');

复制代码

[复制代码](javascript:void(0);)

 　　我们来看上面的例子，我们注册了uncaughtException事件来捕捉系统异常。执行到nonexistentFunc()时，因为该函数没有定义所以会抛出异常。因为javascript是解释性的语言，nonexistentFunc()方法上面的语句不会被影响到，他下面的语句不会被执行。所以他的执行结果如下：

Caught exception: ReferenceError: nonexistentFunc is not defined

This will still run.

　　我们再看一个例子。

[复制代码](javascript:void(0);)

复制代码

var http = require('http');

var server = http.createServer(function(req,res) {

　　res.writeHead(200, {});

　　res.end('response');

　　badLoggingCall('sent response');

　　console.log('sent response');

});

process.on('uncaughtException', function(e) {

　　console.log(e);

});

server.listen(8080);

复制代码

[复制代码](javascript:void(0);)

 　　在这里例子中我们创建了一个web服务器，当处理完请求之后，我们会执行badLoggingCall()方法。因为这个方法不存在，所以会有异常抛出。但是我们注册的uncaughtException事件会对异常做出处理，这样服务器不会受到影响得以继续运行。我们会在服务器端记录错误日志。

[ReferenceError: badLoggingCall is not defined]

#### 与当前进程交互

　　node提供了一些process的属性，如下：

process.version：包含当前node实例的版本号；

process.installPrefix：包含安装路径；

process.platform：列举node运行的操作系统的环境，只会显示内核相关的信息，如：linux2， darwin，而不是“Redhat ES3” ，“Windows 7”，“OSX 10.7”等；

process.uptime()：包含当前进程运行的时长（秒）；

process.getgid(), process.setgid()：获取或者设置group id；

process.getuid(), process.setuid()：获取或者设计user id；

process.pid：获取进程id；

process.title：设置进程名称；

process.execPath：当前node进程的执行路径，如：/usr/local/bin/node；

process.cwd()：当前工作目录；

process.memoryUsage()：node进程内存的使用情况，rss代表ram的使用情况，vsize代表总内存的使用大小，包括ram和swap；

process.heapTotal,process.heapUsed：分别代表v8引擎内存分配和正在使用的大小。

#### 事件循环和ticker

　　node中提供了process.nextTick()方法，允许你访问事件循环和延时那你的工作。他有点类似于setTimeout()，他会在下次tick的时候执行，而且每隔一段事件就会执行一次。我们这里有个例子：

[复制代码](javascript:void(0);)

复制代码

var http = require('http');

var s = http.createServer(function(req, res) {

　　res.writeHead(200, {});

　　res.end('foo');

　　console.log('http response');

　　process.nextTick(function(){console.log('tick')});

});

s.listen(8000);

复制代码

[复制代码](javascript:void(0);)

　　当请求来的时候，会记录日志‘http response’和‘tick’，当没有请求的时候，每隔一段事件会执行事件循环，会输出tick。

　　此外，nextTick创建的回调函数具有隔离性，他们之间不会相互影响。

[复制代码](javascript:void(0);)

复制代码

process.on('uncaughtException', function(e) {

　　console.log(e);

});

process.nextTick(function() {

　　console.log('tick');

});

process.nextTick(function() {

　　iAmAMistake();

　　console.log('tock');

});

process.nextTick(function() {

　　console.log('tick tock');

});

console.log('End of 1st loop');

复制代码

[复制代码](javascript:void(0);)

　　在这个例子中，首先输出‘End of 1st loop’，然后顺序的输出nextTick的回调函数，第一个会正常输出‘tick’，第二个是一个故意设置的异常会输出异常信息，不会输出‘tock’，因为nextTick回调函数的隔离性，第三个任然会输出‘tick tock’。结果如下：

End of 1st loop

tick

[ReferenceError: iAmAMistake is not defined]

tick tock

#### 子进程

node提供了child\_process模块，允许你为主进程创建子进程，这样你就可以使用更多的服务器资源，使用更多的cpu，这些概念在前面的章节有介绍。node提供了child\_process. spawn()和child\_process. exec()为你实现这一功能，下面我们就单独介绍。

**child\_process.exec( )**

　　我们来看exec的一个简单例子，他**创建了一个子进程，第一个参数是一个shell命令，第二个参数是回调函数**，处理返回结果。

[复制代码](javascript:void(0);)

复制代码

var cp = require('child\_process');

cp.exec('ls -l', function(e, stdout, stderr) {

　　if(!e) {

　　　　console.log(stdout);

　　　　console.log(stderr);

　　}

});

复制代码

[复制代码](javascript:void(0);)

exec()还可以传options的参数：

[复制代码](javascript:void(0);)

复制代码

var options = {

encoding: 'utf8',

timeout: 0,

maxBuffer: 200 \* 1024,

killSignal: 'SIGTERM',

setsid: false,

cwd: null,

env: null

};

var cp = require('child\_process');

cp.exec('ls -l', options, function(e, stdout, stderr) {

　　if(!e) {

　　　　console.log(stdout);

　　　　console.log(stderr);

　　}

});

复制代码

[复制代码](javascript:void(0);)

encoding：I/O流的编码格式；

timeout：进程超时时间；

killSignal：当时间或者缓冲区超限时终止进程的信号；

maxBuffer：stdout或stderr可增长的最大值；

setsid：决定在进程中是否创建一个新的会话；

cwd：进程的初始工作目录，为null时表示使用node的当前工作目录；

env：进程的环境变量。

**child\_process.spawn( )**

　　child\_process.spawn( )比child\_process.exec( )更加强大和灵活，例子如下：

[复制代码](javascript:void(0);)

复制代码

var cp = require('child\_process');

var cat = cp.spawn('cat');

cat.stdout.on('data', function(d) {

　　console.log(d.toString());

});

cat.on('exit', function() {

　　console.log('kthxbai');

});

cat.stdin.write('meow');

cat.stdin.end();

复制代码

[复制代码](javascript:void(0);)

### 多线程编程

#### TAGG 线程池

### 异常处理

#### NodeJS异常处理uncaughtException篇

王龑 — APRIL 08, 2015

很多 NodeJS 的开发者在抱怨异常处理太麻烦，我们会通过一些列博客梳理一下NodeJS中常见的异常处理的手段。   
和大多数编程语言一样，在 NodeJS 里可以通过throw抛出一个异常：

1. thrownewError('Catch me');

为了捕获这个异常需要把代码包在Try Catch中：

1. try{
2. thrownewError('Catch me');
3. }catch(e){
4. // error captured
5. }

然而，由于 NodeJS 的异步特性，上述代码只需稍加改造就会失效：

1. try{
2. process.nextTick(function my\_app(){
3. thrownewError('Catch me');
4. })
5. }catch(e){
6. // never called
7. }

在现实世界里，异常总是会产生在某个模块中。所谓模块就是能完成一个功能的单元，即使是一个简单的函数也可以被看做一个模块。随着项目代码行数增多，异步嵌套的复杂性加强，经常会有异常没捕获的情况发生。一个没有很强健壮性的 NodeJS 应用，会因为一个未捕获的异常就整个挂掉，导致服务不可用。要改变大家觉得NodeJS是脆弱的这个认识，需要开发者加深对这门语言异常处理机制的了解。

#### uncaughtException

uncaughtException 其实是 NodeJS 进程的一个事件。如果进程里产生了一个异常而没有被任何Try Catch捕获会触发这个事件。为了简化问题，我们还是先看看同步情况下的例子。

1. function external(){
2. thrownewError('Catch me');
3. }
4. functioninternal(){
5. external();
6. }
7. internal();//error will be thrown

在命令行里执行这个程序，脚本会在抛出异常的那一行中断。接下来，由于没有Try Catch，异常会一直冒泡直到事件循环为止，而NodeJS对异常的默认处理非常简单，处理的代码 类似 于：

1. function\_MyFatalException(err){
2. if(!process.emit('uncaughtException',err)){
3. console.error(err.stack);
4. process.emit('exit',1);
5. }
6. }

NodeJS对于未捕获异常的默认处理是： - 触发 uncaughtException 事件 - 如果 uncaughtException 没有被监听，那么 - 打印异常的堆栈信息 - 触发进程的 exit 事件

如果你正在用 NodeJS 开发服务器，那么你肯定不希望偶然的一个异常让整个服务器挂掉。那么是不是只要监听了 uncaughtException 就可以阻止服务器的进程退出呢？ 答案是可以，但是不要这么做！。看这个例子：

1. var express =require('express');
2. function external(cb){
3. process.nextTick(function(){
4. thrownewError();
5. cb.call(null,'sunny');
6. })
7. }
8. var app = express();
9. app.get('/weather',function(req, res){
10. external(function(data){
11. res.end('Weather of Beijing is '+ data);
12. })
13. })
14. app.listen(8018);
15. function noop(){}
16. process.on('uncaughtException', noop)

上面这个例子假设用户访问站点的时候可以看到当地的天气，我们用 apache2-utils来模拟请求

ab -n 1000 -c 20 <http://localhost:8018/weather>

糟糕！请求一直在等待，内存上涨。原因在于res.end 永远不会执行，现有的I/O处于等待的状态，已经开辟的资源不仅不会被释放，而且服务器还在不知疲倦地接受新的用户请求。

在 NodeJS 中处理异常是代价高昂的，而且一不小心就会导致内存泄露和让应用程序处于不稳定的状态。为了提高健壮性，我们可以用Cluster模式，由之而来的推荐做法是： - 针对发生异常的请求返回一个错误代码 - 出错的Worker不再接受新的请求 - 退出关闭Worker进程

### Nodejs和C++模块

## Nodejs创建一个web服务器

## 模块化开发 npm

### 概述

模块化的基本准则：

modules、require是最基本的方法，分别对应了创建模块和引用模块。

modules（模块）返回一个接口对象，可以是一个对象字面量、字符串、数组、函数，相对于模块来说它是创造出口的源头。

require（引用）引用一个接口对象，引用模块即将该模块缓存起来复用。

在某种意义上来说，模块就相当于jquery插件，require就相当于引用jquery插件，不同的是模块不会覆盖或重写、追加父级对象，不污染、不负重。

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

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

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

$ npm -v

2.3.0

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

$ sudo npm install npm -g

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

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

NPM (Node Package Manager, http://npmjs.org ) 是 NodeJS 的模块管理软件，除 NodeJS 内置的核心模块外，其他模块的安装、卸载等管理操作都要通过 NPM 来进行，我们自己写的模块，就要发布到NPM上来供其他人使用。

$ npm uninstall moudleName

全局安装的模块windows下默认在 C:\Users\Administrator\AppData\Roaming\npm\node\_modules

NodeJS 是一门年轻的语言，扩展模块并不太全，经常我们想用某个模块但是却找不到合适的。比如前两天我需要使用hmac和sha1来做签名，就没有找到一个比较好用的模块，这时候就需要我们自己来实现相应的功能了。自己写完之后，再把它打包成一个模块分享给大家来用，即方便了其他人，又能让自己有点小小的成就感，实在是一件一举多得的好事情。接下来，我就为大家介绍一下如何封装一个NodeJS模块并把它分享给其他人。

NPM (Node Package Manager, http://npmjs.org ) 是 NodeJS 的模块管理软件，除 NodeJS 内置的核心模块外，其他模块的安装、卸载等管理操作都要通过 NPM 来进行，我们自己写的模块，就要发布到NPM上来供其他人使用。

接下来，我们做一个非常简单的模块 “hello”，这个模块的功能只有一个：提供一个参数 “name” ，它在控制台输出 “Hello name”。在开始之前，我们首先要把node和npm装好，安装方法在其官方网站都有介绍，这里就不再多说了。

### 创建一个最简单的模块

首先，我们创建一个名为”hello”的目录，作为模块的主目录。进入该目录，开始我们的工作。

然后，写模块的核心代码，很简单，只有以下三行：

exports.Hello = function ( name ) {

console.log( "Hello " + name );

}

把它保存为 hello.js。

NodeJS每个扩展模块中都有一个package.json文件，用来描述模块的一些基本属性，比如模块名称、作者、版本号等等。关于package.json写法的详细说明，可以使用 “npm help json” 命令来查看。

初始化模块

我们可以在模块主目录下执行 npm init 来生成一个最基本的package.json。按照命令的提示依次输入信息即可。以下是在 hello 目录下执行 npm init 并填入相关信息后的结果：

$ npm init

Package name: (hello) //模块名字，npm init会自动取当前目录名作为默认名字，这里不需要改，直接确认即可

Description: A example for write a module //模块说明

Package version: (0.0.0) 0.0.1 //模块版本号，这个大家按自己习惯来定就可以

Project homepage: (none) //模块的主页，如果有的话可以填在这里，也可以不填

Project git repository: (none) //模块的git仓库，选填。npm的用户一般都使用github做为自己的git仓库

Author name: Elmer Zhang //模块作者名字

Author email: (none) freeboy6716@gmail.com //模块作者邮箱

Author url: (none) http://www.elmerzhang.com //模块作者URL

Main module/entry point: (none) hello.js //模块的入口文件，我们这里是hello.js

Test command: (none) //测试脚本，选填

What versions of node does it run on? (~v0.5.7) \* //依赖的node版本号，我们这个脚本可以运行在任何版本的node上，因此填 \*

About to write to /home/elmer/hello/package.json

// 以下是生成的package.json文件内容预览

{

"author": "Elmer Zhang <freeboy6716@gmail.com> (http://www.elmerzhang.com)",

"name": "hello",

"description": "A example for write a module",

"version": "0.0.1",

"repository": {

"url": ""

},

"main": "hello.js",

"engines": {

"node": "\*"

},

"dependencies": {},

"devDependencies": {}

}

Is this ok? (yes) //对以上内容确认无误后，就可以直接回车确认了

到此为止，我们这个模块就写完了。这时hello目录下应该有两个文件：hello.js和package.json。

安装模块到服务器中

我们可以返回到hello的上级目录，来测试安装一下这个模块：

$ npm install hello/

hello@0.0.1 ./node\_modules/hello

显示安装成功。简单的测试一下：

$ node

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

> Hello('world');

Hello world

正确输出了”Hello world”。

发布到npm管理中心

接下来我们把它发布到NPM上。

首先，我们需要有一个NPM帐号，可以使用npm adduser来注册一个：

$ npm adduser

Username: elmerzhang

Password:

Email: freeboy6716@gmail.com

简单三步，一个NPM用户注册成功。

最后回到 hello 根目录，执行一下npm publish，如果没有任何错误提示，那么就发布成功了。去 <http://search.npmjs.org/>上看一下吧，你的模块应该已经显示在”Latest Updates”一栏里了。

至此，一个NodeJS模块成功发布到NPM，以后就可以在任何能访问npm库的地方通过npm install来安装你的模块了。

#### NPM 使用介绍

#### 使用 npm 命令安装模块

npm 安装 Node.js 模块语法格式如下：

$ npm install <Module Name>

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

$ npm install express

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

var express = require('express');

#### 全局安装与本地安装

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

npm install express # 本地安装

npm install express -g # 全局安装

##### 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。
* 2. 可以直接在命令行里使用。
* 3. 不能通过 require() 来引入本地安装的包。

接下来我们使用全局方式安装 express

$ npm install express -g

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

express@4.13.3 node\_modules/express

├── escape-html@1.0.2

├── range-parser@1.0.2

├── merge-descriptors@1.0.0

├── array-flatten@1.1.1

├── cookie@0.1.3

├── utils-merge@1.0.0

├── parseurl@1.3.0

├── cookie-signature@1.0.6

├── methods@1.1.1

├── fresh@0.3.0

├── vary@1.0.1

├── path-to-regexp@0.1.7

├── content-type@1.0.1

├── etag@1.7.0

├── serve-static@1.10.0

├── content-disposition@0.5.0

├── depd@1.0.1

├── qs@4.0.0

├── finalhandler@0.4.0 (unpipe@1.0.0)

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

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

├── debug@2.2.0 (ms@0.7.1)

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

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

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

你可以使用以下命令来查看所有全局安装的模块：

$ npm ls -g

#### 使用 package.json

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

{

"name": "express",

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

"version": "4.13.3",

"author": {

"name": "TJ Holowaychuk",

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

},

"contributors": [

{

"name": "Aaron Heckmann",

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

},

{

"name": "Ciaran Jessup",

"email": "ciaranj@gmail.com"

},

{

"name": "Douglas Christopher Wilson",

"email": "doug@somethingdoug.com"

},

{

"name": "Guillermo Rauch",

"email": "rauchg@gmail.com"

},

{

"name": "Jonathan Ong",

"email": "me@jongleberry.com"

},

{

"name": "Roman Shtylman",

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

},

{

"name": "Young Jae Sim",

"email": "hanul@hanul.me"

}

],

"license": "MIT",

"repository": {

"type": "git",

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

},

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

"keywords": [

"express",

"framework",

"sinatra",

"web",

"rest",

"restful",

"router",

"app",

"api"

],

"dependencies": {

"accepts": "~1.2.12",

"array-flatten": "1.1.1",

"content-disposition": "0.5.0",

"content-type": "~1.0.1",

"cookie": "0.1.3",

"cookie-signature": "1.0.6",

"debug": "~2.2.0",

"depd": "~1.0.1",

"escape-html": "1.0.2",

"etag": "~1.7.0",

"finalhandler": "0.4.0",

"fresh": "0.3.0",

"merge-descriptors": "1.0.0",

"methods": "~1.1.1",

"on-finished": "~2.3.0",

"parseurl": "~1.3.0",

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

"proxy-addr": "~1.0.8",

"qs": "4.0.0",

"range-parser": "~1.0.2",

"send": "0.13.0",

"serve-static": "~1.10.0",

"type-is": "~1.6.6",

"utils-merge": "1.0.0",

"vary": "~1.0.1"

},

"devDependencies": {

"after": "0.8.1",

"ejs": "2.3.3",

"istanbul": "0.3.17",

"marked": "0.3.5",

"mocha": "2.2.5",

"should": "7.0.2",

"supertest": "1.0.1",

"body-parser": "~1.13.3",

"connect-redis": "~2.4.1",

"cookie-parser": "~1.3.5",

"cookie-session": "~1.2.0",

"express-session": "~1.11.3",

"jade": "~1.11.0",

"method-override": "~2.3.5",

"morgan": "~1.6.1",

"multiparty": "~4.1.2",

"vhost": "~3.0.1"

},

"engines": {

"node": ">= 0.10.0"

},

"files": [

"LICENSE",

"History.md",

"Readme.md",

"index.js",

"lib/"

],

"scripts": {

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

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

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

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

},

"gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",

"bugs": {

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

},

"\_id": "express@4.13.3",

"\_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",

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

"\_npmVersion": "1.4.28",

"\_npmUser": {

"name": "dougwilson",

"email": "doug@somethingdoug.com"

},

"maintainers": [

{

"name": "tjholowaychuk",

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

},

{

"name": "jongleberry",

"email": "jonathanrichardong@gmail.com"

},

{

"name": "dougwilson",

"email": "doug@somethingdoug.com"

},

{

"name": "rfeng",

"email": "enjoyjava@gmail.com"

},

{

"name": "aredridel",

"email": "aredridel@dinhe.net"

},

{

"name": "strongloop",

"email": "callback@strongloop.com"

},

{

"name": "defunctzombie",

"email": "shtylman@gmail.com"

}

],

"dist": {

"shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",

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

},

"directories": {},

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

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

}

##### Package.json 属性说明

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

#### 版本号

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

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

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

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

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

### NPM 常用命令

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

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

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

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

NPM常用的命令有

1) 安装Node模块

 $ npm install moduleNames

2) 查看当前目录下已安装的node包

 $ npm list

3) 查看node模块的package.json文件夹

$ npm view moduleNames

4) 查看包的依赖关系

$ npm view moduleName repository.url

5)  查看包所依赖的Node的版本

$ npm view moduleName engines

6) 更新node模块

$ npm update moduleName

7) 卸载node模块

#### 安装和卸载

安装： npm install moduleNames

卸载： npm uninstall moudleName

卸载模块

我们可以使用以下命令来卸载 Node.js 模块。

$ npm uninstall express

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

$ npm ls

#### 更新模块

我们可以使用以下命令更新模块：

$ npm update express

#### 搜索模块

使用以下来搜索模块：

$ npm search express

#### 创建模块

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

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。

### Nodejs中的模块分类

#### 核心模块

核心模块已经被编译到了 Node 中，我们在其源码中的 lib 目录可以找到这些核心模块。常见的核心模块：net、http、fs 模块等

#### 文件模块

文件模块通过一个相对或者绝对路径加载，例如我们上面看到的 circle.js

**编写一个最简单的模块**

我们知道每个模块对应一个js文件，这篇写一个最简单的模块hello.js， 然后在另一个js文件(main.js)中require自定义的模块。   
  
hello.js

复制代码 代码如下:

function hello(name) {   
console.log('hello, '+ name);   
}   
exports.hello = hello;

main.js

复制代码 代码如下:

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

约定：hello.js和main.js在同一个目录下，比如是node目录   
打开命令行，进入node目录，执行命令

复制代码 代码如下:

node main.js

可以看到命令行输出了：hello, snandy   
  
注意 ：   
main.js中require的参数不能是"hello" ，必须在前面加上"./"。

#### 包模块

自定义模块位于 node\_modules 目录中，我们通过 npm 安装的各种模块就放置在 node\_modules 目录中

##### **概述**

包是在模块基础上更深一步的抽象，它将某个独立的功能封装起来，用于发布、更新、依赖管理和版本控制。 Node.js 根据 CommonJS 规范实现了包机制，开发了npm来解决包的发布和获取需求。

Node.js 的包是一个目录，其中包含一个 JSON 格式的包说明文件 package.json。严格符   
合 CommonJS 规范的包应该具备以下特征：

package.json 必须在包的顶层目录下；

二进制文件应该在 bin 目录下；

JavaScript 代码应该在 lib 目录下；

文档应该在 doc 目录下；

单元测试应该在 test 目录下。

Node.js 对包的要求并没有这么严格，只要顶层目录下有 package.json，并符合一些规范   
即可。当然为了提高兼容性，建议在制作包的时候，严格遵守 CommonJS 规范。

作为文件夹的模块

模块与文件是一一对应的。文件不仅可以是 JavaScript 代码或二进制代码，还可以是一个文件夹。最简单的包，就是一个作为文件夹的模块。下面我们来看一个例子，建立一个叫做somepackage 的文件夹，在其中创建 index.js，内容如下：

//somepackage/index.jsexports.hello = function() { console.log('hello');};

然后在 somepackage 之外建立 getpackage.js，内容如下：

//getpackage.jsvar somePackage = require('./somePackage');somePackage.hello();

运行 node getpackage.js，[控制台](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=18&is_app=0&jk=bf7b6389d98fd54f&k=%BF%D8%D6%C6%CC%A8&k0=%BF%D8%D6%C6%CC%A8&kdi0=0&luki=5&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=4fd58fd989637bbf&ssp2=1&stid=0&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fweb%2Fjs%2F201502%2F85612%2Eshtml&urlid=0)将输出结果 hello 。

使用这种方法可以把文件夹封装为一个模块，即所谓的包。包通常是一些模块的集合，在模块的基础上提供了更高层的抽象，相当于提供了一些固定接口的函数库。通过定制package.json，我们可以创建更复杂、更完善、更符合规范的包用于发布。

##### **package.json**

在前面例子中的 somepackage 文件夹下，我们创建一个叫做 package.json 的文件，内容如   
下所示：

{ "main": "./lib/interface.js"}

然后将 index.js 重命名为 interface.js 并放入 lib 子[文件夹](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=18&is_app=0&jk=bf7b6389d98fd54f&k=%CE%C4%BC%FE%BC%D0&k0=%CE%C4%BC%FE%BC%D0&kdi0=0&luki=9&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=4fd58fd989637bbf&ssp2=1&stid=0&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fweb%2Fjs%2F201502%2F85612%2Eshtml&urlid=0)下。以同样的方式再次调用这个包，依然可以正常使用。

Node.js 在调用某个包时，会首先检查包中 package.json 文件的 main 字段，将其作为包的接口模块，如果 package.json 或 main 字段不存在，会尝试寻找 index.js 或 index.node 作为包的接口。

package.json 是 CommonJS 规定的用来描述包的文件，完全符合规范的 package.json 文件应该含有以下字段。

name：包的名称，必须是唯一的，由小写英文字母、数字和[下划线](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=18&is_app=0&jk=bf7b6389d98fd54f&k=%CF%C2%BB%AE%CF%DF&k0=%CF%C2%BB%AE%CF%DF&kdi0=0&luki=1&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=4fd58fd989637bbf&ssp2=1&stid=0&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fweb%2Fjs%2F201502%2F85612%2Eshtml&urlid=0)组成，不能包含空格。

description：包的简要说明。

version：符合语义化版本识别规范的版本字符串。

keywords：关键字数组，通常用于搜索。

maintainers：维护者数组，每个元素要包含 name、email（可选）、web（可选）字段。

contributors：贡献者数组，格式与maintainers相同。包的作者应该是贡献者数组的第一个元素。

bugs：提交bug的地址，可以是网址或者电子邮件地址。

licenses：许可证数组，每个元素要包含type（许可证的名称）和url（链接到许可证文本的地址）字段。

repositories：仓库托管地址数组，每个元素要包含type仓库的类型，如git ）、url（仓库的地址）和 path（相对于[仓库](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=18&is_app=0&jk=bf7b6389d98fd54f&k=%B2%D6%BF%E2&k0=%B2%D6%BF%E2&kdi0=0&luki=6&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=4fd58fd989637bbf&ssp2=1&stid=0&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fweb%2Fjs%2F201502%2F85612%2Eshtml&urlid=0)的路径，可选）字段。

dependencies：包的依赖，一个关联数组，由包名称和版本号组成。

下面是一个完全符合 CommonJS 规范的 package.json 示例：

{ "name": "mypackage", "description": "Sample package for CommonJS. ", "version": "0.7.0", "keywords": [ "package", "example" ], "maintainers": [ { "name": "Bill Smith", "email": "bills@example.com", } ], "contributors": [ { "name": "BYVoid", "web": "http://www.byvoid.com/" } ], "bugs": { "mail": "dev@example.com", "web": "http://www.example.com/bugs" }, "licenses": [ { "type": "GPLv2", "url": "http://www.example.org/licenses/gpl.html" } ], "repositories": [ { "type": "git", "url": "http://github.com/BYVoid/mypackage.git" } ], "dependencies": { "webkit": "1.2", "ssl": { "gnutls": ["1.0", "2.0"], "openssl": "0.9.8" } }}

##### ****Node.js 包管理器****

Node.js包管理器，即npm是 Node.js 官方提供的包管理工具，它已经成了 Node.js 包的   
标准发布平台，用于 Node.js 包的发布、传播、依赖控制。npm 提供了命令行[工具](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=18&is_app=0&jk=bf7b6389d98fd54f&k=%B9%A4%BE%DF&k0=%B9%A4%BE%DF&kdi0=0&luki=3&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=4fd58fd989637bbf&ssp2=1&stid=0&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fweb%2Fjs%2F201502%2F85612%2Eshtml&urlid=0)，使你可   
以方便地下载、安装、升级、删除包，也可以让你作为开发者发布并维护包。

1.获取一个包

使用 npm 安装包的命令格式为：

npm [install/i] [package name]

例如你要安装express，可以在命令行运行：

npm isntall express

或者

npm i express

说明:npm 在获取 express 的时候还将自动解析其依赖，并获取 express 依赖的 mime、 mkdirp、qs 和 connect。

2.本地模式和全局模式

npm在默认情况下会从http://npmjs.org搜索或下载包，将包安装到当前目录的node\_modules子目录下。

在使用 npm 安装包的时候，有两种模式： 本地模式和全局模式。

默认情况下我们使用 npm install命令就是采用本地模式，即把包安装到当前目录的node\_modules子目录下。 Node.js的 require 在加载模块时会尝试搜寻 node\_modules子目录，因此使用 npm 本地模式安装的包可以直接被引用。

npm 还有另一种不同的安装模式被成为全局模式，使用方法为：

npm [install/i] -g [package\_name]

为什么要使用全局模式呢？多数时候并不是因为许多程序都有可能用到它，为了减少多重副本而使用全局模式，而是因为本地模式不会注册 PATH环境变量。

举例说明，安装supervisor是为了在命令行中运行它，譬如直接运行 supervisor script.js，这时就需要在PATH环境变量中注册supervisor。 npm本地模式仅仅是把包安装到 node\_modules 子目录下，其中的 bin 目录没有包含在 PATH 环境变量中，不能直接在命令行中调用。而当我们使用全局模式安装时， npm 会将包安装到系统目录， 譬如/usr/local/lib/node\_modules/， 同时 package.json 文件中 bin 字段包含的文件会被链接到 /usr/local/bin/。 /usr/local/bin/ 是在PATH[环境变量](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=18&is_app=0&jk=bf7b6389d98fd54f&k=%BB%B7%BE%B3%B1%E4%C1%BF&k0=%BB%B7%BE%B3%B1%E4%C1%BF&kdi0=0&luki=7&n=10&p=baidu&q=smileking_cpr&rb=0&rs=1&seller_id=1&sid=4fd58fd989637bbf&ssp2=1&stid=0&t=tpclicked3_hc&td=1682280&tu=u1682280&u=http%3A%2F%2Fwww%2Eth7%2Ecn%2Fweb%2Fjs%2F201502%2F85612%2Eshtml&urlid=0)中默认定义的，因此就可以直接在命令行中运行 supervisor script.js命令了。

| 模式 | 可通过require使用 | 注册PATH |
| --- | --- | --- |
| 本地模式 | 是 | 否 |
| 全局模式 | 否 | 是 |

总而言之，当我们要把某个包作为工程运行时的一部分时，通过本地模式获取，如果要   
在命令行下使用，则使用全局模式安装。

提示:在 Linux/Mac 上使用 npm install -g 安装时有可能需要 root 权限，   
因为 /usr/local/lib/node\_modules/ 通常只有管理员才有权写入。

3.创建全局链接

npm 提供了一个有趣的命令 npm link， 它的功能是在本地包和全局包之间创建符号链接。我们说过使用全局模式安装的包不能直接通过 require 使用，但通过 npm link命令可以打破这一限制。举个例子，我们已经通过 npm install -g express 安装了 express，这时在工程的目录下运行命令：

npm link express ./node\_modules/express -> /usr/local/lib/node\_modules/express

我们可以在 node\_modules 子目录中发现一个指向安装到全局的包的符号链接。通过这   
种方法，我们就可以把全局包当本地包来使用了。

警告:npm link 命令不支持 Windows。

包的发布

在发布之前，首先需要让我们的包符合 npm 的规范， npm 有一套以 CommonJS 为基础包规范，但与 CommonJS并不完全一致，其主要差别在于必填字段的不同。通过使用 npm init 可以根据交互式问答产生一个符合标准的 package.json，创建一个名为novaline的目录，然后在这个目录中运行npm init:

npm initname: (novaline) novalineModuleversion: (1.0.0)description: "just for learning"entry point: (index.js)test command:git repository:keywords:author: novalinelicense: (ISC)About to write to F:/workspace/css\_pro/novaline/package.json:{ "name": "novalineModule", "version": "1.0.0", "description": "/"just for learning/"", "main": "index.js", "scripts": { "test": "echo /"Error: no test specified/" && exit 1" }, "author": "novaline", "license": "ISC"}

这样就在 novaline 目录中生成一个符合 npm 规范的 package.json 文件。创建一个   
index.js 作为包的接口，一个简单的包就制作完成了。

在发布前，我们还需要获得一个账号用于今后维护自己的包，使用 npm adduser 根据提示输入用户名、密码、邮箱，等待账号创建完成。完成后可以使用 npm whoami 测验是否已经取得了账号。

接下来，在 package.json 所在目录下运行 npm publish，稍等片刻就可以完成发布了。打开浏览器，访问 http://search.npmjs.org/ 就可以找到自己刚刚发布的包了。现在我们可以在世界的任意一台计算机上使用 npm install novalineModule 命令来安装它。

注意:如果出现如下错误

npm ERR! Windows\_NT 6.3.9600npm ERR! argv "C://Program Files//nodejs////node.exe" "C://Program Files//nodejs//node\_modules//npm//bin//npm-cli.js" "publish"npm ERR! node v0.12.0npm ERR! npm v2.5.1npm ERR! Invalid name: "novalineModule"npm ERR!npm ERR! If you need help, you may report this error at:npm ERR! <http://github.com/npm/npm/issues>npm ERR! Please include the following file with any support request:npm ERR! F:/workspace/css\_pro/novalineModule/npm-debug.log

说明novalineModule这个模块名已经被人使用了，需要在package.json里把name字段对应的模块名修改下，再发布。

如果你的包将来有更新，只需要在 package.json 文件中修改 version 字段，然后重新使用 npm publish 命令就行了。如果你对已发布的包不满意（比如我们发布的这个毫无意义的包），可以使用 npm unpublish 命令来取消发布。

##### 创建一个最简单的模块

首先，我们创建一个名为”hello”的目录，作为模块的主目录。进入该目录，开始我们的工作。

然后，写模块的核心代码，很简单，只有以下三行：

exports.Hello = function ( name ) {

console.log( "Hello " + name );

}

把它保存为 hello.js。

NodeJS每个扩展模块中都有一个package.json文件，用来描述模块的一些基本属性，比如模块名称、作者、版本号等等。关于package.json写法的详细说明，可以使用 “npm help json” 命令来查看。

##### 初始化模块

我们可以在模块主目录下执行 npm init 来生成一个最基本的package.json。按照命令的提示依次输入信息即可。以下是在 hello 目录下执行 npm init 并填入相关信息后的结果：

$ npm init

Package name: (hello) //模块名字，npm init会自动取当前目录名作为默认名字，这里不需要改，直接确认即可

Description: A example for write a module //模块说明

Package version: (0.0.0) 0.0.1 //模块版本号，这个大家按自己习惯来定就可以

Project homepage: (none) //模块的主页，如果有的话可以填在这里，也可以不填

Project git repository: (none) //模块的git仓库，选填。npm的用户一般都使用github做为自己的git仓库

Author name: Elmer Zhang //模块作者名字

Author email: (none) freeboy6716@gmail.com //模块作者邮箱

Author url: (none) http://www.elmerzhang.com //模块作者URL

Main module/entry point: (none) hello.js //模块的入口文件，我们这里是hello.js

Test command: (none) //测试脚本，选填

What versions of node does it run on? (~v0.5.7) \* //依赖的node版本号，我们这个脚本可以运行在任何版本的node上，因此填 \*

About to write to /home/elmer/hello/package.json

// 以下是生成的package.json文件内容预览

{

"author": "Elmer Zhang <freeboy6716@gmail.com> (http://www.elmerzhang.com)",

"name": "hello",

"description": "A example for write a module",

"version": "0.0.1",

"repository": {

"url": ""

},

"main": "hello.js",

"engines": {

"node": "\*"

},

"dependencies": {},

"devDependencies": {}

}

Is this ok? (yes) //对以上内容确认无误后，就可以直接回车确认了

到此为止，我们这个模块就写完了。这时hello目录下应该有两个文件：hello.js和package.json。

##### 安装模块到服务器中

我们可以返回到hello的上级目录，来测试安装一下这个模块：

$ npm install hello/

hello@0.0.1 ./node\_modules/hello

显示安装成功。简单的测试一下：

$ node

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

> Hello('world');

Hello world

正确输出了”Hello world”。

##### 发布到npm管理中心

接下来我们把它发布到NPM上。

首先，我们需要有一个NPM帐号，可以使用npm adduser来注册一个：

$ npm adduser

Username: elmerzhang

Password:

Email: freeboy6716@gmail.com

简单三步，一个NPM用户注册成功。

最后回到 hello 根目录，执行一下npm publish，如果没有任何错误提示，那么就发布成功了。去 <http://search.npmjs.org/>上看一下吧，你的模块应该已经显示在”Latest Updates”一栏里了。

至此，一个NodeJS模块成功发布到NPM，以后就可以在任何能访问npm库的地方通过npm install来安装你的模块了。

### 不同模块的加载顺序

核心模块总是优先被加载，假如有一个自定义模块 http，那么加载时仍然会加载核心模块 http 而非自定义模块 http。在加载自定义模块时，首先查找当前目录下的 node\_modules 目录，而后查找父目录下的 node\_modules 目录，以此类推，直到根目录。

require 加载的模块不是一个文件，而是一个目录时，这样的目录被叫做包（package）。包中存在一个名为 package.json 的文件（包描述文件），例如：

复制代码 代码如下:

{ "name" : "some-library",  
  "main" : "./lib/some-library.js" }

其中 main 指出需要被加载的模块。如果包中不存在 package.json 或者 package.json 中未指定 main 模块，那么 Node 将会尝试加载 index.js、index.node、index.json。

### 模块的编写- 返回值可以是字符串，函数，字面量，对象

nodejs中的模块化遵循CMD，但又有些不同。

#### 出口是一个或者多个字符串 数字等变量

1. // 新建一个计算圆周长、圆面积的模块
3. var pi = Math.PI;
5. // 计算周长
6. exports.perimeter = function (r) {
7. return 2 \* pi \* r;
8. }
10. // 计算面积
11. exports.area = function (r) {
12. return pi \* r \* r;
13. }

如上，定义了一个计算圆周长、圆面积的模块，因为每一个文件就是一个模块，就不需要写define函数了，直接写出口exports即可。exports出口可以是任何对象。如：

1. // module.js
2. // 模块出口是字面量
3. module.exports="I'm a module!";
5. // index.js
6. // 调用模块
7. var md = require("./module.js");
8. md;

#### 出口是字面量

1. // module.js
2. // 模块出口是对象
3. module.exports={
4. name:"yundanran",
5. love:"ming"
6. };
8. // index.js
9. // 调用模块
10. var md = require("./module.js");
11. md.name;
12. md.love;

#### 出口是对象

1. // module.js
2. function Animal(type){
3. this.type=type;
4. }
6. Animal.prototype.say=function(){
7. return "Animal type is "+this.type;
8. }
10. // 出口是一个类
11. module.exports=Animal;
13. // index.js
14. // 模块调用
15. var Md = require("./module.js");
16. var md = new Md("people");
17. md.say();

#### 出口是多个函数

1. // module.js
2. exports.fn1 = function () {
3. return "fn1";
4. }
6. exports.fn2 = function () {
7. return "fn2";
8. }
10. // index.js
11. var md = require("./module.js");
12. md.fn1();
13. md.fn2();

#### 模块调用 – require

#### 模块多次调用 – 会自动覆盖

无论调用多少次 require， 获得的模块都是同一个。我们在 getmodule.js 的基础上稍作修改：

//loadmodule.js

第一次调用

var a = require('./module');

setName('novaline');

第二次调用 – 自动覆盖第一次调用

var b = require('./module');

b.setName('nova');

a.sayHello();

运行结果：

nova

**说明**:变量a和b指向的是同一个实例，因此b.setName覆盖了a.setName。

### 模块的寻找路径

### 案例 - 输入原价，打折方式等得到最终价格

### 案例 – 输出html头部组件

### 改造案例—使用nodejs模块来做

## Node.js Express 模块

### Express 简介 - Express就是包模块

Express 是一个简洁而灵活的 node.js Web应用框架, 提供了一系列强大特性帮助你创建各种 Web 应用，和丰富的 HTTP 工具。

使用 Express 可以快速地搭建一个完整功能的网站。

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

#### 第一篇文章

安装 Express 并将其保存到依赖列表中：

$ npm install express --save

以上命令会将 Express 框架安装在当期目录的 **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req.cookies可以取到传过来的cookie，并把它们转成对象。
* **multer** - node.js 中间件，用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。

$ npm install body-parser --save

$ npm install cookie-parser --save

$ npm install multer --save

**安装Express**

Express是nodejs常用的一个框架。

a) 全局安装

**[plain]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. npm install express -gd

b) 安装在当前文件夹下

**[plain]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. npm install express

安装成功后，命令行会提示 npm info ok

-g代表安装到NODE\_PATH的lib里面，而-d代表把相依性套件也一起安装。如果沒有-g的话会安装目前所在的目录(会建立一个node\_modules的文件夹)。

**在项目中引用express包**

例：

**[javascript]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. **var** express = require('express');
2. **var** app = module.exports = express.createServer();

如果没有安装过express,那么首先需要在当前项目文件夹下安装一个express

命令行转到当前路径后，运行

**[plain]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. npm install express

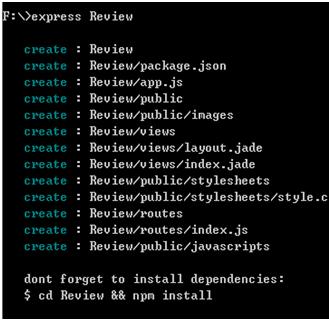
安装完成后，可以看到当前目录下多了一个【node\_modules】文件夹，下有一个【express】文件夹

注：项目中引用的包，都会被安装到【node\_modules】文件夹

**用express创建项目**

在命令行中输入【express 项目名称】，就可以在当前文件夹下创建一个新的项目

如图：



包括以下几个文件：



用此方法，只是创建了一个空的项目框架，和一个简单的实例程序，运行app.js可以查看（还需要在项目文件目录下，安装jade包，方法类似安装express）

#### 第二篇文章

Node.js是一个让JavaScript运行在服务端的针对Web的开发平台，因为它才使得JavaScript在服务端堪与PHP、Python、Perl、Ruby平起平坐，跻身于脚本语言世界的一等公民的行列。

Express作为Node.js的开发框架，目前也是最稳定，使用最广泛，而且Node.js官方推荐的唯一一个Web开发框架，毫无疑问首选开发框架就是它了。

那么接下来，构建一个node.js+express的开发环境将是一切的开始。

系统环境：windows8(32位)，windows7(32位)

1.安装node.js。



[**node.js v0.10.36 官方正式版**](http://www.cr173.com/soft/52260.html)评分:

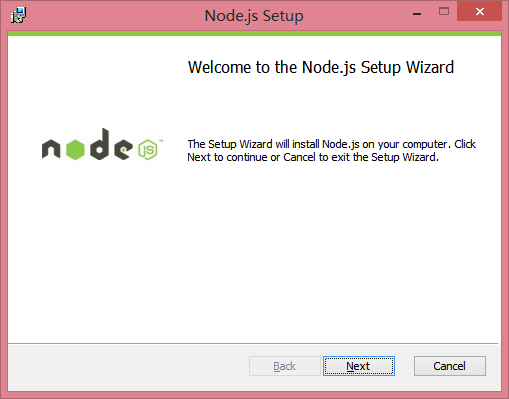
**10.0**

类别： 编程控件    大小：9.7M    语言： 中文   
[**查看详细信息 >>**](http://www.cr173.com/soft/52260.html)

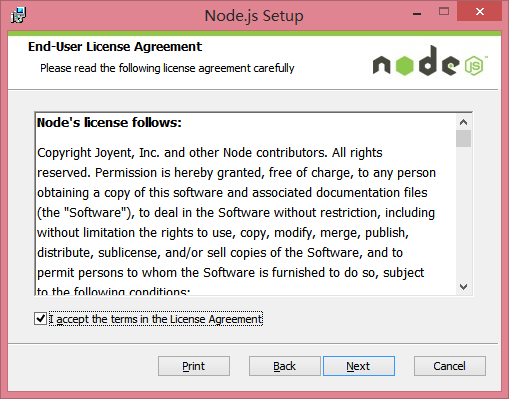
进入官网的下载地址：http://www.nodejs.org/download/。

选择Windows Installer或者选择Windows Installer (.msi) 32-bit，下载安装包。

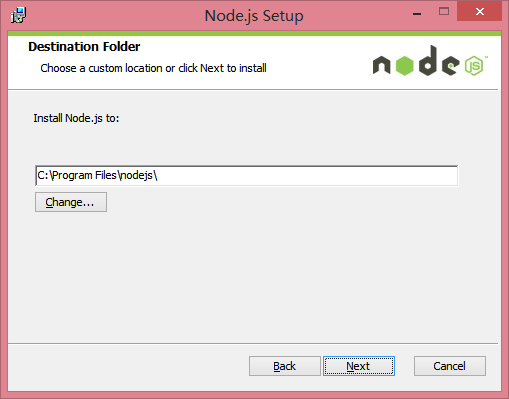
下载完成后，双击安装，点击Next（如下图）。



必然是为“I accept the terms in the License Agreement”打上勾（如下图）。



选择安装路径（如下图）。



查看并可修改将被安装的功能（如下图）。

**默认安装：**

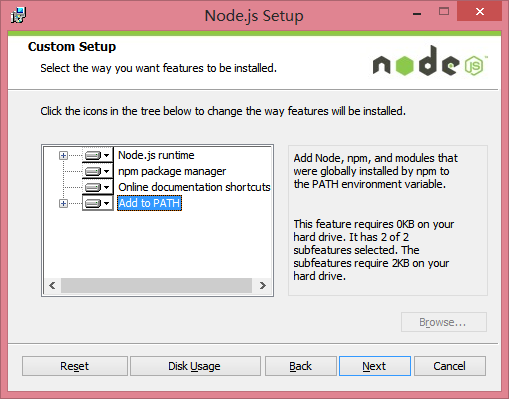
Node.js运行环境，必装，不解释。

npm包管理器，用来安装第三方的Package，Express就是。

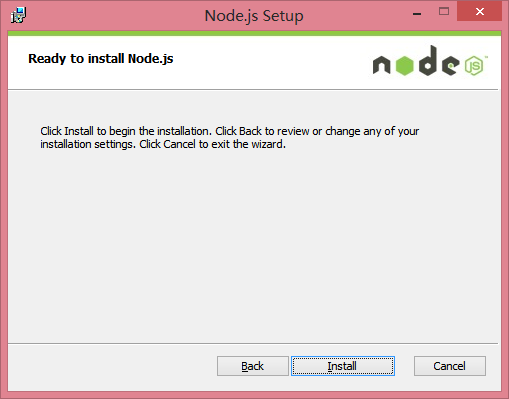
Online document shortcuts就是在安装中，会在[开始菜单](http://www.cr173.com/k/kscd/)的Node.js项下产生一个在线文档的链接。

Add to PATH就是在安装完后，将Node.js的安装目录添加到环境变量Path中，那么，在命令行中的任何路径下运行node命令。

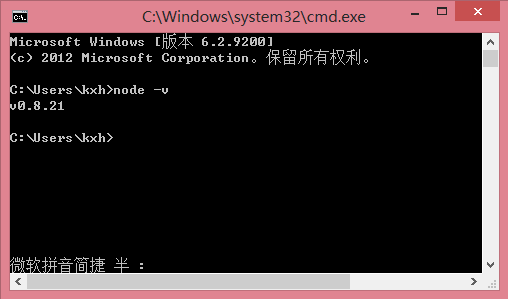
建议：请保持默认安装。



接下来，选择“Instal”，进行安装。



安装完成后，在命令行中输入node -v，看到版本号，则安装成功。



**2.安装Express。**

通过nmp包管理器进行安装，安装分为：

全局安装：自动安装到“C:\Users\[当前用户]\AppData\Roaming\npm”下，并且自动将路径添加到环境变量“Path”中。

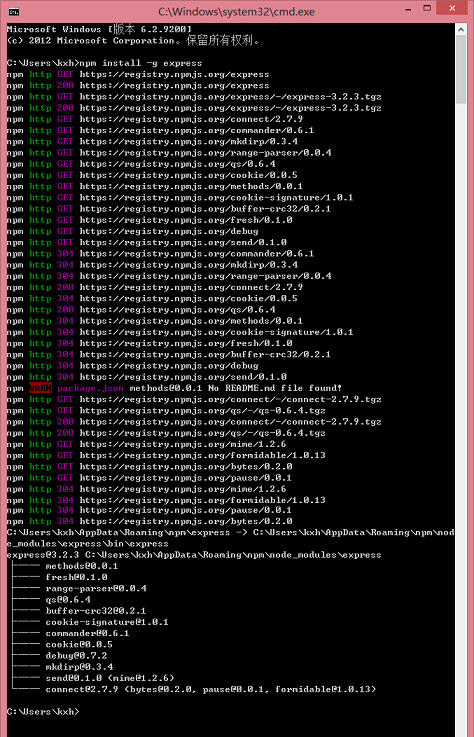
本地安装：安装到当前目录下，不会将路径添加到环境变量“Paht”中。

为了方便使用并且能在命令行中的任意路径下运行“express”命令，首选全局安装。

OK！现在就在命令行中输入“npm install -g express”。

注："-g"就是全局安装选项，没有这个就是本地安装。

稍等片刻后安装完成，然后就可以在命令行中输入“express --help”，查看帮助信息。



至此，环境安装完成。

### 常用命令

#### 安装Express

Express是nodejs常用的一个框架。

a) 全局安装

**[plain]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. npm install express -gd

b) 安装在当前文件夹下

**[plain]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. npm install express

安装成功后，命令行会提示 npm info ok

-g代表安装到NODE\_PATH的lib里面，而-d代表把相依性套件也一起安装。如果沒有-g的话会安装目前所在的目录(会建立一个node\_modules的文件夹)。

#### 在项目中引用express包

例：

**[javascript]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. **var** express = require('express');
2. **var** app = module.exports = express.createServer();

如果没有安装过express,那么首先需要在当前项目文件夹下安装一个express

命令行转到当前路径后，运行

**[plain]** [view plaincopy](http://blog.csdn.net/haidaochen/article/details/7257655)

1. npm install express

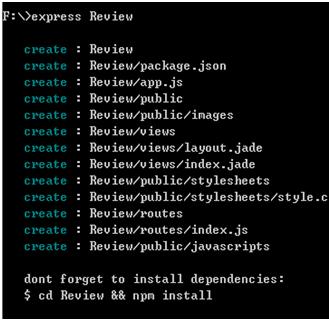
安装完成后，可以看到当前目录下多了一个【node\_modules】文件夹，下有一个【express】文件夹

注：项目中引用的包，都会被安装到【node\_modules】文件夹

#### 用express创建项目

在命令行中输入【express 项目名称】，就可以在当前文件夹下创建一个新的项目

如图：



包括以下几个文件：



用此方法，只是创建了一个空的项目框架，和一个简单的实例程序，运行app.js可以查看（还需要在项目文件目录下，安装jade包，方法类似安装express）

### Node和NPM的安装

够便捷了，不细说...有几点基础顺手提一下：

1. 安装命令中的 “-g” 表示全局(global)
2. express的版本不是通常的 “-v” 来查看，而是 “-V”
3. 安装express项目的命令如下  
   **express -e nodejs-product**

-e, --ejs add ejs engine support

-J, --jshtml add jshtml engine support (defaults to jade)

PS：模板引擎之类暂时不必care，不过俺当初学习搭建Node+express时用的是ejs，所以也就顺手一直用着了

### Node的小基友supervisor

　　每次修改代码后会自动重启。懒程序员就指望这种省事省力的工具活着了:)

　　安装：**npm install -g supervisor**

　　执行：**supervisor app.js**

### 另一个小基友forever

　　虚拟机一关node服务就关了，不过forever可以让node服务不停止，介绍如下，安装和执行不细说啦，我懒：

　　forever是一个简单的命令式nodejs的守护进程，能够启动，停止，重启App应用。forever完全基于命令行操作，在forever进程之下，创建node的子进程，通过monitor监控node子进程的运行情况，一旦文件更新，或者进程挂掉，forever会自动重启node服务器，确保应用正常运行。

### express项目目录

http://images.cnitblog.com/i/159097/201402/272113409312155.png

　　如上图就是一个express项目结构，简单过一下：

#### app.js

项目入口，反正express爱叫app.js没辙，你可以改成index.js或者main.js都成。相当于php项目中的 index.php、index.html

#### node\_modules： 存放项目的依赖库

#### package.json

* 项目依赖配置及开发者信息(这个要说就说多了，还是看文档好，俺就不误人子弟了。下期看看抽个小段单说Node模块)

#### public： 静态文件如 css,js,img (PS:俺其实习惯叫static)

#### routes： 路由文件(学习的重要攻克对象。尼玛业务好不好，路由是关键)

#### Views： 页面文件(Ejs或者jade的模板，默认是jade，俺这用Ejs，在初阶练手最重要，所以都可以试试)

     打开View 文件发现index.ejs比较不习惯，所以对app.js进行小改动：

1. “app.set('view engine', 'ejs');” 变成 “app.engine('.html', ejs.\_\_express);app.set('view engine', 'html');”
2. 上一行出现的ejs变量需要require ejs模块，增加代码“var ejs = require('ejs');”

     最终的app.js如下：



**代码小解：**

     因为针对的是初阶入门，俺们还是继续过一下express的使用与Node的方法哈：

#### Express基础知识点

##### require()

用于在当前模块中加载和使用其他模块；

此方法是模块的基础，使用中大概有路径的概念就行。PS：JS文件可以去掉".js"后缀

##### exports

表示模块的导出对象，用于导出模块的属性和公共方法。

在项目routes文件夹下有index.js和users.js（路由有细说），都使用到exports对象导出对象，如33行的routes.index和34行的user.list;

　　 PS：一个模块的代码只会在模块第一次被使用时执行，不会因require多次而被初始化多次。

##### express()

表示创建express应用程序

简单几行代码其实就可以创建一个应用，如下：

[复制代码](javascript:void(0);)

var express = require('express');

var app = express();

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

res.send('hello world');

console.log('hello world');

});

app.listen('8808');

[复制代码](javascript:void(0);)

##### app.listen()

就是在给定的主机和端口上监听请求

这个和node中http模块的http.createServer(function(){...}).listen()效果一致；

##### app.set 和 app.get

app.set(name, value)和app.get(name)就是你想的那样，set()为设置 name 的值设为 value，get()为获取设置项 name 的值。如俺app.js的图片16行中的一句“app.set('port', process.env.PORT || 3000)”就是设置项目的port，在下面使用http.createServer时就可以使用app.get('port')来获取，只是俺偷懒没用来着http://images.cnitblog.com/i/159097/201402/280809248922088.gif；

##### \_\_express

不用去care，其实就是ejs模块的一个公共属性，表示要渲染的文件扩展名。

##### app.engine()

     了解app.engine()方法之前先看看express应用的安装命令:“express -e nodejs-product”，其中的 -e 和 -J 我们一开始已经提到，表示ejs和jade模板。

     如果想把模板后缀改成“.html”时就会用到app.engine方法，来重新设置模板文件的扩展名，比如想用ejs模板引擎来处理“.html”后缀的文件：app.engine('.html', require('ejs').\_\_express);

     app.engine(ext, callback) 注册模板引擎的 callback 用来处理ext扩展名的文件。

　　 PS：\_\_express不用去care，其实就是ejs模块的一个公共属性，表示要渲染的文件扩展名。

##### app.use([path], function)

使用中间件 function,可选参数path默认为"/"。

使用 app.use() “定义的”中间件的顺序非常重要，它们将会顺序执行，use的先后顺序决定了中间件的优先级(经常有搞错顺序的时候);

     最后介绍个很有用的express API：

##### app.render(view, [options], callback)

渲染 view, callback 用来处理返回的渲染后的字符串。

**路由实战：**

     路径代码应该是项目中最本机的一部分了。express中创建一个或者一套新的handle非常简单，先看看express现有的，一会儿我们创建俩个实际的规则。

http://images.cnitblog.com/i/159097/201402/282058558455441.png

     变量 routes 和 user 都是刚才require的模块，他们各自exports了index方法和list方法；其中Response.render()表示渲染view，同时传进对应的数据，Response.send()为发送一个响应；在设置路由时index和list方法作为回调函数最终执行。

　　 流程大概了解啦，俺们也就实际搞一把，最easy的一种方式，简单俩步：

1. 第一种方式就是在当前的routes/index.js或者routes/test.js中加几行代码如下
2. exports.test = function(req, res){
3. res.send('test welcome.');

};

1. 在app.js文件设置路由那块加上app.get('/test', routes.test);

　　 第二种方式就是多了两步，先新建一个模块如test.js文件，输出然后exports对应的方法；在app.js中require这个模块，再加一行设置路由即完成了。

**快速炫起来，集成Bootstrap：**

     JS工程师使用Nodejs上手还是以快速搭建网站为主，所以才会介绍Express，那么为了让网站更快的体面起来，集成使用Bootstrap就是上佳选择，非常喜欢其响应式布局和整体系的脚手架。

     PS：因为Bootstrap的JS插件都依赖jQeury，所以jQuery也一并引入了。

　　 前文已经说到了，静态文件都放在public文件夹中，切文件夹内已经帮我们把类目都分好了，images、 javascripts、 stylesheets。

　　 分别引入bootstrap.min.css文件至stylesheets目录下；jquery-1.x.x.min.js和bootstrap.min.js放到javascripts文件夹下。

　　 然后俺们修改view/index.html把文件引入使用即可，下面放出俺在bootstrap demo的基础改的index.html，大家随意拿去使用和修改。

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　 如果样式有问题请检查下bootstrap的路径是否正确引入。

　　 启动项目之后觉得 高大上 很简单，有木有！！

### 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) {

// --

})

可以通过以下链接查看 **request** 和 **response** 对象的具体介绍：

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

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

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

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

### 路由

我们已经了解了 HTTP 请求的基本应用，而路由决定了由谁(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。

创建 express\_demo2.js 文件，代码如下所示：

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.delete('/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(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)

})

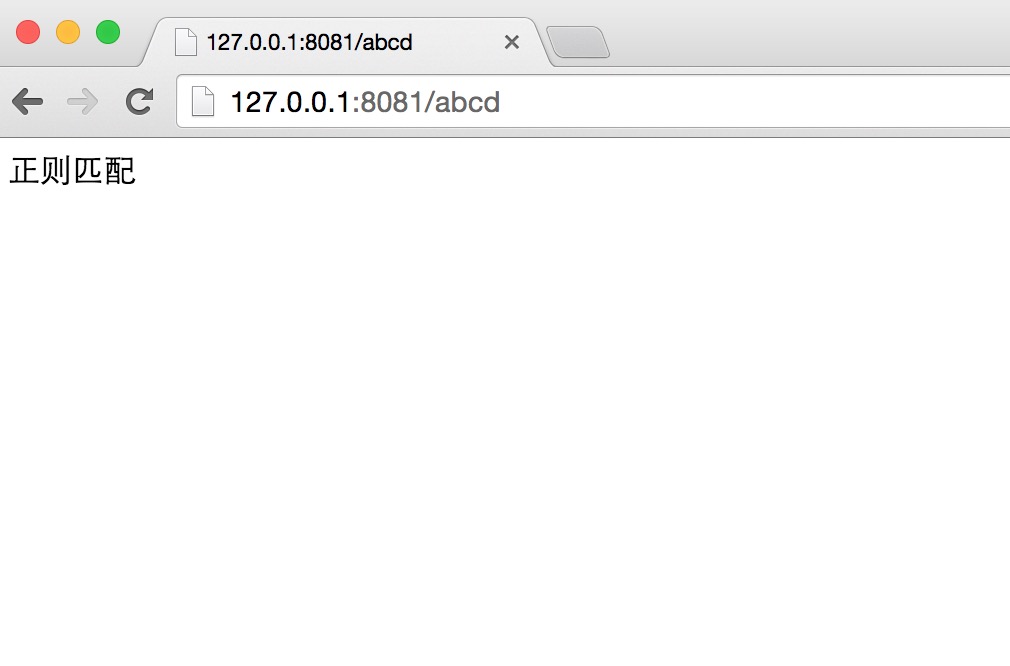
执行以上代码：

$ node express\_demo2.js

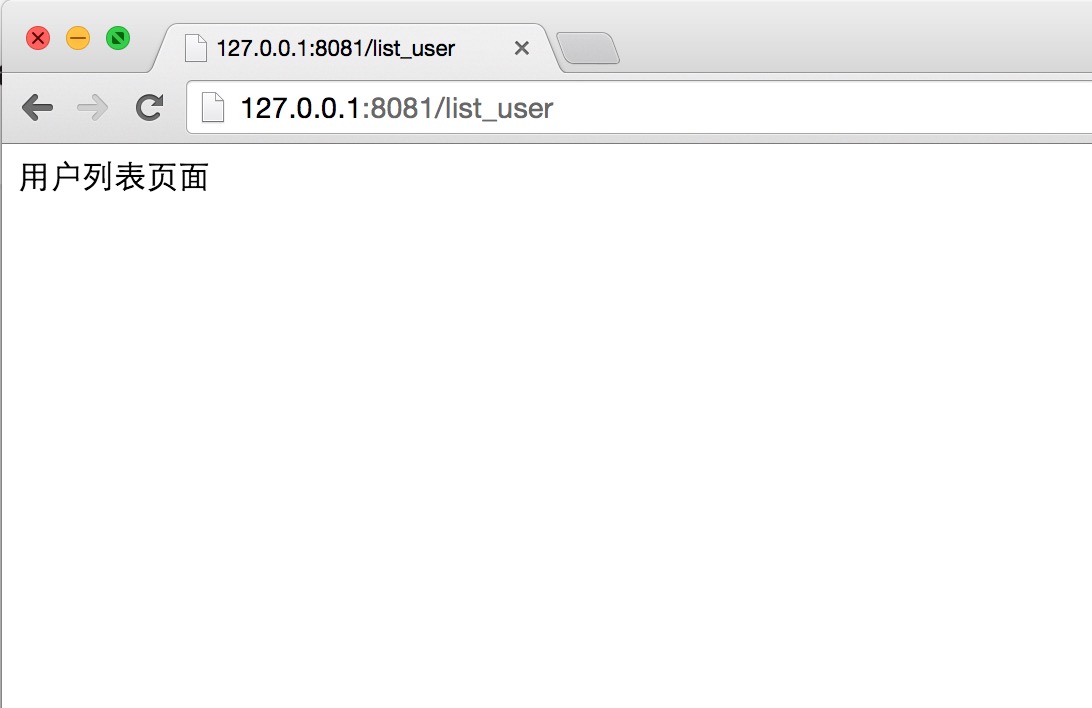
应用实例，访问地址为 http://0.0.0.0:8081

接下来你可以尝试访问 http://127.0.0.1:8081 不同的地址，查看效果。

在浏览器中访问 http://127.0.0.1:8081/list\_user，结果如下图所示：



在浏览器中访问 http://127.0.0.1:8081/abcd，结果如下图所示：



在浏览器中访问 http://127.0.0.1:8081/abcdefg，结果如下图所示：



### 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。

你可以使用**express.static**中间件来设置静态文件路径。例如，如果你将图片， CSS, JavaScript 文件放在 public 目录下，你可以这么写：

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

我们可以到 public/images 目录下放些图片,如下所示：

node\_modules

server.js

public/

public/images

public/images/logo.png

让我们再修改下 "Hello Word" 应用添加处理静态文件的功能。

创建 express\_demo3.js 文件，代码如下所示：

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(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)

})

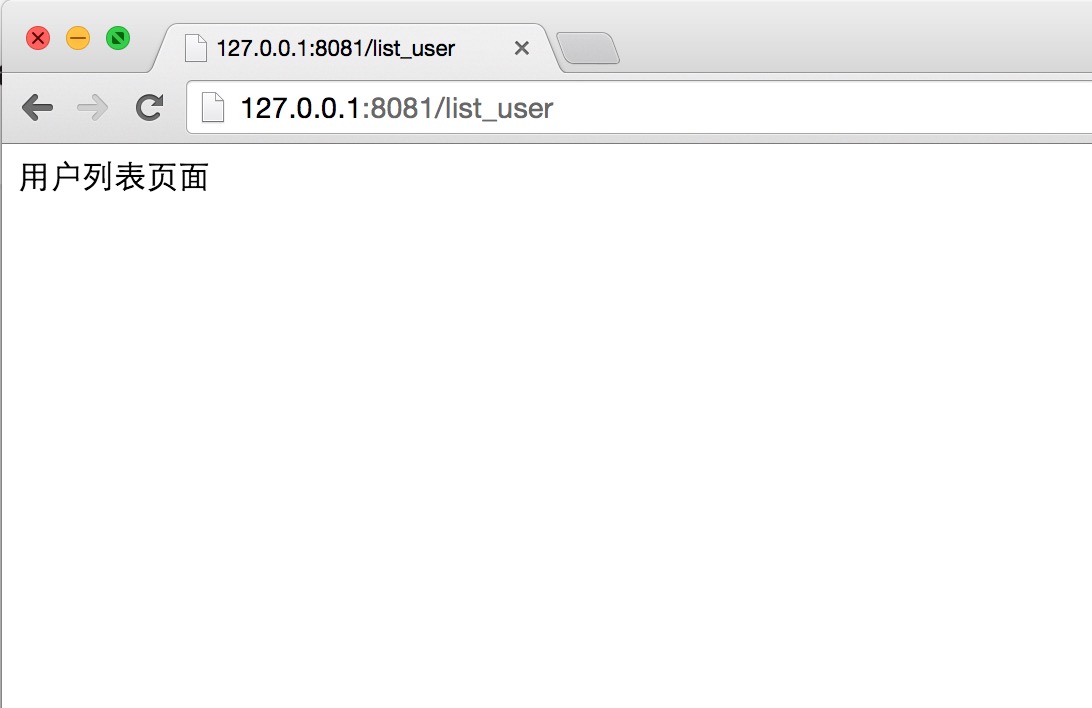
执行以上代码：

$ node express\_demo3.js

应用实例，访问地址为 http://0.0.0.0:8081

执行以上代码：

在浏览器中访问 http://127.0.0.1:8081/images/logo.png，结果如下图所示：



### GET 方法

以下实例演示了在表单中通过 GET 方法提交两个参数，我们可以使用 server.js 文件内的 **process\_get** 路由器来处理输入：

index.htm 文件代码如下：

<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 文件代码如下:

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)

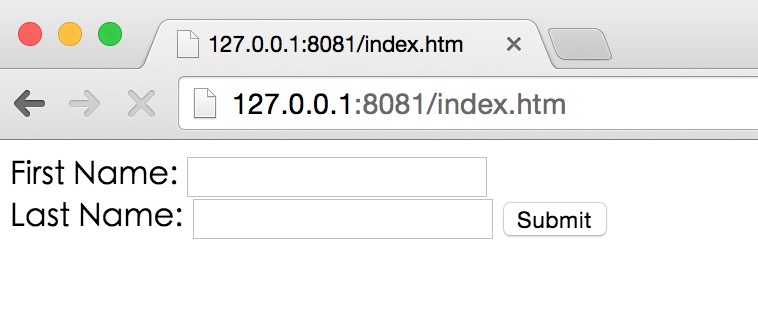
})

执行以上代码：

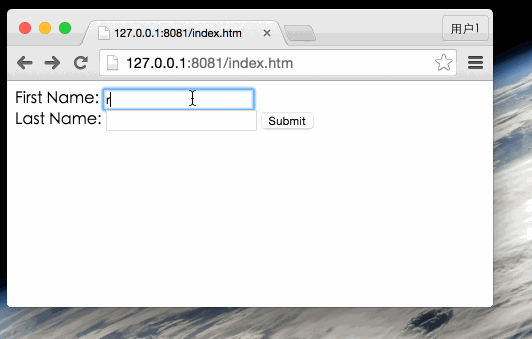
node server.js

应用实例，访问地址为 http://0.0.0.0:8081

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



### POST 方法

以下实例演示了在表单中通过 POST 方法提交两个参数，我们可以使用 server.js 文件内的 **process\_get** 路由器来处理输入：

index.htm 文件代码修改如下：

<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>

server.js 文件代码修改如下:

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)

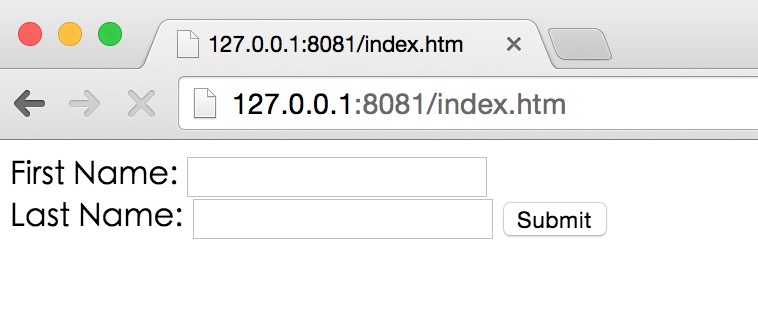
})

执行以上代码：

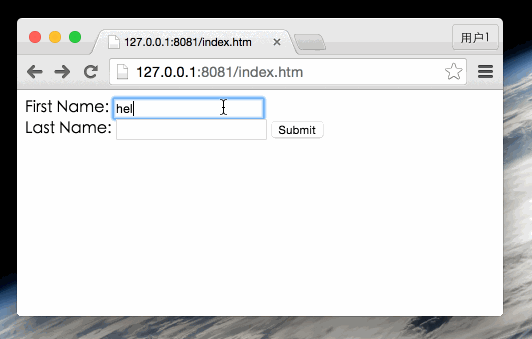
$ node express\_demo.js

应用实例，访问地址为 http://0.0.0.0:8081

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



### 文件上传

以下我们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。

index.htm 文件代码修改如下：

<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>

server.js 文件代码修改如下:

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：

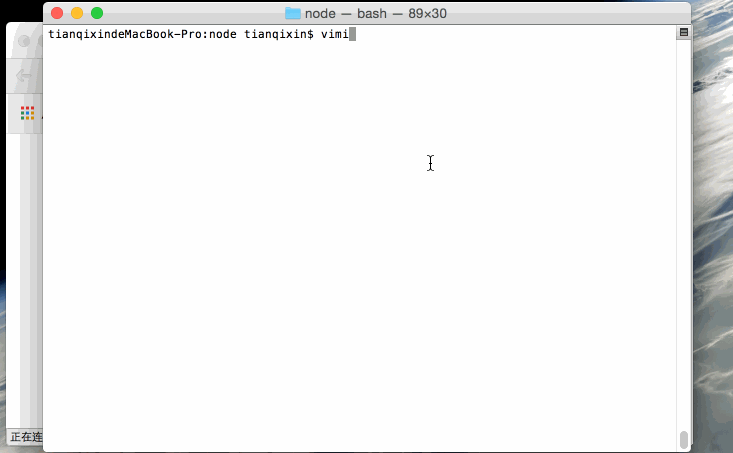
$ node express\_demo.js

应用实例，访问地址为 http://0.0.0.0:8081

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：



现在你可以向表单输入数据，并提交，如下演示：



### Cookie 管理

我们可以使用中间件向 Node.js 服务器发送 cookie 信息，以下代码输出了客户端发送的 cookie 信息：

// express\_cookie.js 文件

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)

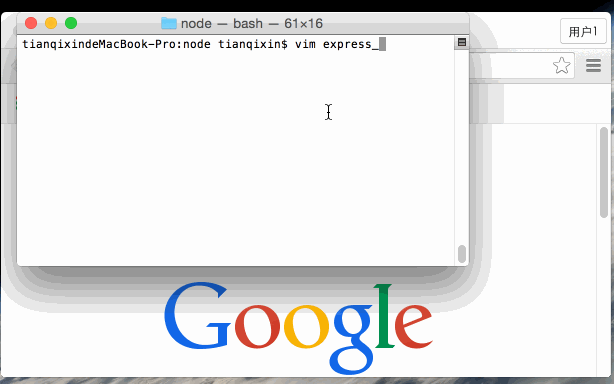
})

app.listen(8081)

执行以上代码：

$ node express\_demo.js

现在你可以访问 http://127.0.0.1:8081 并查看终端信息的输出，如下演示：



## 数据库

## Web service

## Rest

### Node.js RESTful API

### 什么是 REST？

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

表述性状态转移是一组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或设计就是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的应用程序，可使用开放的XML（标准通用标记语言下的一个子集）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，使用它们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

更多介绍，可以查看：[RESTful 架构详解](http://www.runoob.com/w3cnote/restful-architecture.html)

### 创建 RESTful

首先，创建一个 json 数据资源文件 users.json，内容如下：

{

"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

}

}

基于以上数据，我们创建以下 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：

**以下代码，我们创建了 RESTful API listUsers，用于读取用户的信息列表， server.js 文件代码如下所示：**

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)

})

**接下来执行以下命令：**

$ node server.js

应用实例，访问地址为 http://0.0.0.0:8081

**在浏览器中访问 http://127.0.0.1:8081/listUsers，结果如下所示：**

{

"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

}

}

### 添加用户

以下代码，我们创建了 RESTful API **addUser**， 用于添加新的用户数据，server.js 文件代码如下所示：

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)

})

接下来执行以下命令：

$ node server.js

应用实例，访问地址为 http://0.0.0.0:8081

在浏览器中访问 http://127.0.0.1:8081/addUsers，结果如下所示：

{ 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 },

user4:

{ name: 'mohit',

password: 'password4',

profession: 'teacher',

id: 4 }

}

### 显示用户详情

以下代码，我们创建了 RESTful API **:id（用户id）**， 用于读取指定用户的详细信息，server.js 文件代码如下所示：

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 = users["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)

})

接下来执行以下命令：

$ node server.js

应用实例，访问地址为 http://0.0.0.0:8081

在浏览器中访问 http://127.0.0.1:8081/2，结果如下所示：

{

"name":"suresh",

"password":"password2",

"profession":"librarian",

"id":2

}

### 删除用户

以下代码，我们创建了 RESTful API **deleteUser**， 用于删除指定用户的详细信息，以下实例中，用户 id 为 2，server.js 文件代码如下所示：

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)

})

接下来执行以下命令：

$ node server.js

应用实例，访问地址为 http://0.0.0.0:8081

在浏览器中访问 http://127.0.0.1:8081/deleteUser，结果如下所示：

{ user1:

{ name: 'mahesh',

password: 'password1',

profession: 'teacher',

id: 1 },

user3:

{ name: 'ramesh',

password: 'password3',

profession: 'clerk',

id: 3 }

}

## 半全栈开发

### MySql 来存储用户数据

### NoSql 存储历史天气数据

### Redis 作为 cache

## 全栈开发

## 大四喜

# nodejs学习2：Hello world

本文最后一次更新于**2年前**，文章内容可能略有出入。若发现文章中有错误之处，可以留言评论告诉作者。

## 1、经典的hello world

hello world几乎是每一门语言入门的经典例子。

在E盘根目录创建\_\_nodejs文件夹，然后在该文件夹新建hello world.js文件。内容为：

1. // 引用http模块
2. var http = require("http");
4. // 创建服务端
5. http.createServer(function (request, response) {
6. // 输出网页头信息，200状态
7. response.writeHead(200, {
8. // 网页MIME类型
9. "content-type": "text/html"
10. });
12. // 输出网页内容
13. response.write("<h1>Hello world!</h1>");
15. // 控制台输出
16. console.log("Hello world!");
18. // 结束输出
19. response.end();
20. })
22. // 监听2014端口
23. .listen(2014);

在命令窗口输入：

1. cd e:\\_\_nodejs
2. e:
3. node hello\_world.js

打开http://localhot:2014，页面上输出了Hello world。同时，在控制台输出了：

1. Hello world!
2. Hello world!

为什么会有两次呢？

* 第1次：请求http://localhost:2014
* 第2次：请求http://localhost:2014/favicon.ico

# nodejs学习3：url、get、post、querystring

本文最后一次更新于**2年前**，文章内容可能略有出入。若发现文章中有错误之处，可以留言评论告诉作者。

## 1、处理get请求

获取请求的get参数来响应不同的内容。在\_\_nodejs文件夹下创建get.js文件，内容为：

1. // 引用http模块
2. var http = require("http");
4. // 引用url模块
5. var url = require("url");
7. // 引用querystring模块
8. var querystring = require("querystring");
10. http.createServer(function (request, response) {
11. var objectUrl = url.parse(request.url);
12. var objectQuery = querystring.parse(objectUrl.query);
14. response.writeHead(200, {
15. "content-type": "text/html"
16. });
18. // 输出url的各项参数
19. response.write("<h1>objectUrl</h1>");
20. for (var i in objectUrl) {
21. if (typeof (objectUrl[i]) != "function") response.write(i + "=>" + objectUrl[i] + "<br>");
22. }
24. // 输出url中的query的各项参数
25. response.write("<h1>objectQuery</h1>");
26. for (var i in objectQuery) {
27. response.write(i + "=>" + objectQuery[i] + "<br>");
28. }
30. response.end();
31. }).listen(2014);

在命令窗口输入：

1. node get.js

在浏览器上打开http://localhost:2014/?a=1&b=2&c=3，输出了：

1. objectUrl：
2. protocol=>null
3. slashes=>null
4. auth=>null
5. host=>null
6. port=>null
7. hostname=>null
8. hash=>null
9. search=>?a=1&b=2&c=3
10. query=>a=1&b=2&c=3
11. pathname=>/
12. path=>/?a=1&b=2&c=3
13. href=>/?a=1&b=2&c=3
15. objectQuery：
16. a=>1
17. b=>2
18. c=>3

## 2、处理post请求

获取请求的post参数来响应不同的内容。在\_\_nodejs文件夹下创建post.js，内容为：

1. // 引用 http 模块
2. var http = require("http");
4. // 引用 querystring 模块
5. var querystring = require("querystring");
7. // 创建服务端
8. http.createServer(function (request, response) {
9. // 定义变量，用来处理post数据
10. var postData = "";
12. // 输出字符串
13. var responseString = "";
15. response.writeHead(200, {
16. "content-type": "text/html"
17. });
19. // 如果是get请求
20. if (request.method == "GET") {
21. responseString = '<!doctype html><html lang="en"><head><meta charset="UTF-8" /><title>Document</title></head><body>\
22. <form action="/" method="post">\
23. <input type="text" name="a" value="1" />\
24. <input type="text" name="b" value="2" />\
25. <input type="text" name="c" value="3" />\
26. <input type="submit" value="submit" />\
27. </form>\
28. </body></html>';
30. response.write(responseString);
32. response.end();
33. }
34. // 如果是post请求
35. else if (request.method == "POST") {
36. // 设置接收数据编码格式为 UTF-8
37. request.setEncoding("utf8");
39. // 因为nodejs在处理post数据的时候，会将数据分成小包来序列处理
40. // 所以必须监听每一个数据小包的结果
41. request.addListener("data", function (postDataChunk) {
42. postData += postDataChunk;
43. });
45. // 所有数据包接收完毕
46. request.addListener("end", function () {
47. // 解析post数据
48. var objectPostData = querystring.parse(postData);
50. for (var i in objectPostData) {
51. responseString += i + " => " + objectPostData[i] + "<br>";
52. }
54. response.write(responseString);
56. response.end();
57. });
58. }
59. }).listen(2014);

cmd命令窗口输入：

1. node post.js

浏览器打开http://localhost:2014/，输出了3个输入框和1个提交按钮，点击post提交之后，页面上输出：

1. a => 1
2. b => 2
3. c => 3

关于post上传，待到后续再说。

# nodejs学习4：filestream读写本地文件

本文最后一次更新于**1年前**，文章内容可能略有出入。若发现文章中有错误之处，可以留言评论告诉作者。

## 1、filestream API

* 读文件：fs.readFile（异步）
* 写文件：fs.writeFile（异步）
* 追加内容：fs.appendFile（异步）

核心例子为：

1. // 引用 fs（filestream） 模块
2. var fs = require("fs");
4. // 1、读文件
5. fs.readFile("text.txt", function (error, fileData) {
6. if (error) {
7. // 出现错误
8. }
9. // 操作fileData
10. });
12. // 2、写文件
13. fs.writeFiel("text.txt", "new fileData", function (error) {
14. if (error) {
15. // 出现错误
16. }
17. // 继续操作
18. });
20. // 3、追加内容
21. fs.appendFile("text.txt", "append fileData", function (error) {
22. if (error) {
23. // 出现错误
24. }
25. // 继续操作
26. });

## 2、集成例子

通过访问不同的querystring来操作不同的API：

* localhost:2014?read => 读文件
* localhost:2014?write => 写文件
* localhost:2014?append => 追加内容

示例如下：

1. // 引用 http 模块
2. var http = require("http");
4. // 引用 filestream 模块
5. var fs = require("fs");
7. // 引用 url 模块
8. var url = require("url")
10. // 引用 querystring 模块
11. var querystring = require("querystring")
13. http.createServer(function (request, response) {
14. var objQuery = querystring.parse(url.parse(request.url).query);
16. // 读取文件
17. if (objQuery.type == "read") {
18. // 为什么不是 fs.read
19. fs.readFile("./tmp/file.txt", function (error, fileData) {
20. if (error) {
21. write(response, "<h1>读取出现错误</h1>");
22. } else {
23. write(response, "<h1>读取内容为：</h1>" + fileData);
24. }
25. });
26. }
27. // 写入文件
28. else if (objQuery.type == "write") {
29. var writeString = "\n" + Date.now();
30. fs.writeFile("./tmp/file.txt", writeString, function (error) {
31. if (error) {
32. write(response, "<h1>写入出现错误</h1>");
33. } else {
34. write(response, "<h1>写入内容为：</h1>" + writeString);
35. }
36. });
37. }
38. // 追加内容
39. else if (objQuery.type == "append") {
40. var appendString = "\n" + Date.now();
41. fs.appendFile("./tmp/file.txt", appendString, function (error) {
42. if (error) {
43. write(response, "<h1>追加出现错误</h1>");
44. } else {
45. write(response, "<h1>追加内容为：</h1>" + appendString);
46. }
47. });
48. } else {
49. write(response, "<h1>请在网址上输入参数</h1>");
50. }
51. }).listen(2014);
53. function write(response, content) {
54. response.writeHead(200, {
55. "content-type": "text/html"
56. });
57. response.write(content);
58. response.end();
59. }

# nodejs学习5：简单模块化

本文最后一次更新于**3个月前**，文章内容可能略有出入。若发现文章中有错误之处，可以留言评论告诉作者。

## 1、单文件

继上一次写的根据querystring来操作不同的filestream API，现在来根据不同的pathname来操作不同的filestream API。如下：

1. // 引用 http 模块
2. var http = require("http");
4. // 引用 url 模块
5. var url = require("url");
7. // 引用 fs 模块
8. var fs = require("fs");
10. http.createServer(function (request, response) {
11. var pathname = url.parse(request.url).pathname;
12. var string = Date.now();
14. // http://localhost:2014/app1
15. if (/^\/app1\/?$/.test(pathname)) {
16. write(response, 200, "<h1>Hello app1!</h1>");
17. }
18. // http://localhost:2014/app1/read
19. else if (/^\/app1\/read\/?$/.test(pathname)) {
20. fs.readFile("../tmp/file.txt", function (error, data) {
21. if (error) {
22. write(response, 500, "<h1>Internet server error!</h1>");
23. } else {
24. write(response, 200, "<h1>读取内容为：</h1><pre>" + data + "</pre>");
25. }
26. });
27. }
28. // http://localhost:2014/app1/write
29. else if (/^\/app1\/write\/?$/.test(pathname)) {
30. var string = Date.now();
31. fs.writeFile("../tmp/file.txt", string, function (error) {
32. if (error) {
33. write(response, 500, "<h1>Internet server error!</h1>");
34. } else {
35. write(response, 200, "写入内容为：" + string + "");
36. }
37. });
38. }
39. // http://localhost:2014/app1/append
40. else if (/^\/app1\/append\/?$/.test(pathname)) {
41. var string = Date.now();
42. fs.appendFile("../tmp/file.txt", string, function (error) {
43. if (error) {
44. write(response, 500, "<h1>Internet server error!</h1>");
45. } else {
46. write(response, 200, "写入内容为：" + string + "");
47. }
48. });
49. }
50. // 404
51. else{
52. write(response, 404, "Page not found!");
53. }
54. });


58. function write(response, statusCode, content) {
59. response.writeHead(statusCode, {
60. "content-type": "text/html"
61. });
62. response.write(content);
63. response.end();
64. }

一个简单的web程序诞生了，有了最基本的查询、修改、追加的功能，并且配置了404。

## 2、模块化

我们可以把这个简单的web程序想象的更复杂、更庞大、功能更丰富，一个页面的js肯定是不符合管理的。就像一个function不会把所有的功能都包括进去，模块化从此而来。

模块化的基本准则：

modules、require是最基本的方法，分别对应了创建模块和引用模块。

modules（模块）返回一个接口对象，可以是一个对象字面量、字符串、数组、函数，相对于模块来说它是创造出口的源头。

require（引用）引用一个接口对象，引用模块即将该模块缓存起来复用。

在某种意义上来说，模块就相当于jquery插件，require就相当于引用jquery插件，不同的是模块不会覆盖或重写、追加父级对象，不污染、不负重。

完整例子：

创建一个新的文件夹app1，分别有index（主入口文件）、router（路由模块，用来指引请求）、config（配置模块，用于接收路由后的动作）。

index.js：

1. // 主入口
3. // 引用 http 模块
4. var http = require("http");
6. // 引用 url 模块
7. var url = require("url");
9. // 引用 fs 模块
10. var fs = require("fs");
12. // 引用 router 模块
13. var router = require("./router.js");
15. // 引用 config 模块
16. var config = require("./config.js");
18. http.createServer(function (request, response) {
19. // 根据请求获得事件句柄
20. var eventHandle = router.route(request);
22. // 执行配置事件输出响应
23. config[eventHandle](response);
24. }).listen(2014);

router.js：

1. // 路由
3. // 模块接口
4. exports.route = function (request) {
5. var url = require("url");
6. var pathname = url.parse(request.url).pathname;
7. var aReg = ['', 'read', 'write', 'append'];
8. var find = 0;
9. var reg;
10. var i;
11. var val;
13. for (i in aReg) {
14. val = aReg[i];
15. reg = new RegExp('^\\\/app1' + (val ? '\\\/' + val : val) + '\\\/?$');
16. if (reg.test(pathname)) {
17. return val ? val : "default";
18. }
19. }
21. return "404";
22. }

config.js：

1. // 配置
3. // 引用 fs 模块
4. var fs = require("fs");
6. // 模块接口
7. module.exports = {
8. "default": function (response) {
9. write(response, 200, "<h1>Hello app1!</h1>");
10. },
12. "read": function (response) {
13. fs.readFile("../tmp/file.txt", function (error, data) {
14. if (error) {
15. write(response, 500, "<h1>Internet server error!</h1>");
16. } else {
17. write(response, 200, "<h1>读取内容为：</h1><pre>" + data + "");
18. }
19. });
20. },
22. "write": function (response) {
23. var string = Date.now();
24. fs.writeFile("../tmp/file.txt", string, function (error) {
25. if (error) {
26. write(response, 500, "<h1>Internet server error!</h1>");
27. } else {
28. write(response, 200, "写入内容为：" + string + "");
29. }
30. });
31. },
33. "append": function (response) {
34. var string = Date.now();
35. fs.appendFile("../tmp/file.txt", string, function (error) {
36. if (error) {
37. write(response, 500, "Internet server error!");
38. } else {
39. write(response, 200, "追加内容为：" + string + "");
40. }
41. });
42. },
44. "404": function (response) {
45. write(response, 404, "Page not found!");
46. },
47. };
49. function write(response, statusCode, content) {
50. response.writeHead(statusCode, {
51. "content-type": "text/html"
52. });
53. response.write(content);
54. response.end();
55. }

# nodejs学习5：深入模块化

本文最后一次更新于**1年前**，文章内容可能略有出入。若发现文章中有错误之处，可以留言评论告诉作者。

## 1、什么是模块化

模块化是将一个整体系统按特定规则划分为各个独立的个体。比如计算机模块化为CPU、内存、电源、显示器等等。模块化是一个简化复杂的过程，同时在模块化后各个独立的个体之间的协调处理将变得更加的重要。用来协调处理各个模块之间的工作的部分称之为模块管理，监管和调配各个模块能够正常工作，并能处理意外错误。

随着模块的数量增加，如何处理模块显得非常的重要，要保证各个模块之间的工作不会过分耦合、不会导致模块之间互相抵触，还要保证好各个模块之间的依赖关系。

我们熟悉的jquery插件就相当于是jquery的模块化，这些模块是挂载在系统主枝干上的，也就是说命名了一个模块a，那么就不能再命名一个模块a，否则的话会出现模块覆盖的问题。

为了解决这种问题，一种随时用随时取的模块思想诞生了，即使是在其他语言中存在很久，如c#的useing等，以及常被人们津津乐道的class类。在js中，客户端语言和服务器语言最大的区别就是，使用源代码必须从服务器上下载下来，然后才能解释运行。因为种种网络原因，并不能保证模块能够正常的加载进来，如何加载模块又可以分成两类。

## 2、模块化分类

在js业界，模块的加载分为AMD、CMD两类，它们都有自己的适合场合，尤其是在服务端和客户端之间。

* AMD：Asynchronous Module Definition，异步模块规范。指在需要用到该模块的时候，才去加载模块，是一个异步过程，常用于客户端。因为这些模块都是远程文件，调用的时候花费时间不定。因此，在客户端浏览器JS中推荐使用该AMD。
* CMD：Common Module Definition，通用模块规范。指在初始化的时候就加载需要用到的模块，是一个同步过程，常用于服务端。因为这些模块都是本地文件，调用的时候花费时间非常少。因此，在服务端JS中推荐使用CMD。

AMD、CMD的使用场合没有硬性规定，各取所需，合适的就是最好的。

在模块规范上，就有着本质的区别，因此模块化的过程也是不同的。

* AMD：一个define是一个模块，同一个文件可以是多个模块。
* CMD：一个文件就是一个模块，多个文件集合是一个大模块。

如：

1. // AMD
2. // 定义 module1 ，依赖 depend1 和 depend2
3. define("module1", [" depend1", "depend2"], function (depend1, depend2) {
4. // depend1
5. // depend2
6. });
8. // 定义 module2 ，依赖 module1
9. define("module2", ["module1"], function (module1) {
10. // module1
11. });
13. // CMD
14. // 定义模块为文件名
15. define(function (require, exports, module) {
16. // require
17. // exports
18. // module
19. });

AMD和CMD的模块定义非常的类似，都使用了define函数。不同的是，AMD模块规范，在定义模块的时候就既定了依赖模块，而CMD模块并未这么做，模块之间的依赖关系有额外的配置文件决定。

在前端JS框架中，遵循AMD的有requireJS，遵循CMD的有seajs，各具特色。

## 3、单模块文件

nodejs中的模块化遵循CMD，但又有些不同。

1. // 新建一个计算圆周长、圆面积的模块
3. var pi = Math.PI;
5. // 计算周长
6. exports.perimeter = function (r) {
7. return 2 \* pi \* r;
8. }
10. // 计算面积
11. exports.area = function (r) {
12. return pi \* r \* r;
13. }

如上，定义了一个计算圆周长、圆面积的模块，因为每一个文件就是一个模块，就不需要写define函数了，直接写出口exports即可。exports出口可以是任何对象。如：

A、出口是字面量

1. // module.js
2. // 模块出口是字面量
3. module.exports="I'm a module!";
5. // index.js
6. // 调用模块
7. var md = require("./module.js");
8. md;

B、出口是对象

1. // module.js
2. // 模块出口是对象
3. module.exports={
4. name:"yundanran",
5. love:"ming"
6. };
8. // index.js
9. // 调用模块
10. var md = require("./module.js");
11. md.name;
12. md.love;

C、出口是类

1. // module.js
2. function Animal(type){
3. this.type=type;
4. }
6. Animal.prototype.say=function(){
7. return "Animal type is "+this.type;
8. }
10. // 出口是一个类
11. module.exports=Animal;
13. // index.js
14. // 模块调用
15. var Md = require("./module.js");
16. var md = new Md("people");
17. md.say();

D、出口是函数

1. // module.js
2. exports.fn1 = function () {
3. return "fn1";
4. }
6. exports.fn2 = function () {
7. return "fn2";
8. }
10. // index.js
11. var md = require("./module.js");
12. md.fn1();
13. md.fn2();

## 4、创建模块包

由多个模块文件组成的一个整体，称之为包（模块包、package），这些模块文件合作完成一个相对更大一点的功能，具有一些处理错误、异常的能力。在模块包（文件夹）中约定包含：

* 模块包配置文件（package.json）必须放在顶层目录。
* 二进制文件必须在bin文件夹下。
* JS文件必须在lib文件夹下。
* 文档必须在doc文件夹下。
* 单元测试必须在test文件夹下。

新建一个包module1，用来计算圆柱体的面积和体积。新建文件、文件夹如下：

* /node\_modules/module1/
* /node\_modules/module1/lib/
* /node\_modules/module1/doc/
* /node\_modules/module1/test/
* /node\_modules/module1/index.js
* /node\_modules/module1/lib/index.js
* /node\_modules/module1/lib/circle.js
* /node\_modules/module1/lib/rect.js

lib/circle.js：

1. // 计算圆周长、面积的模块
3. var pi = Math.PI;
4. module.exports = {
5. perimeter: function (r) {
6. return 2 \* pi \* r;
7. },
8. area: function (r) {
9. return pi \* r \* r;
10. }
11. };

lib/rect.js：

1. // 计算矩形面积模块
3. module.exports = function (w, h) {
4. return w \* h;
5. }

lib/index.js：

1. // 计算圆柱面积、体积
3. // 引用 circle 模块
4. var circle = require("./circle.js");
6. // 引用 rect 模块
7. var rect = require("./rect.js");
9. exports.area = function (r, h) {
10. return 2 \* circle.area(r) + circle.perimeter(r) \* h;
11. }
13. exports.volume = function (r, h) {
14. return circle.area(r) \* h;
15. }

index.js：

1. module.exports=require("./lib");
2. // 相当于 module.exports=require("./lib/index.js");

然后，cd到该模块文件夹，执行以下命令（关于nmp后续再说）：

1. npm init

命令板会要求输入模块包的各种信息，最后在该包文件夹下生成package.json文件（/node\_modules/module1/package.json）。package.json文件包含了模块包的各种信息，其中name值就是引用包需要用的名称。例package.json：

1. {
2. "name": "module1",
3. "version": "1.0.0",
4. "description": "",
5. "main": "index.js",
6. "directories": {
7. "doc": "doc",
8. "test": "test"
9. },
10. "scripts": {
11. "test": "test"
12. },
13. "author": "ydr.me",
14. "license": "BSD-2-Clause"
15. }

引用该模块包，在/app3/index.js：

1. // 引用系统的 http 模块
2. var http = require("http");
4. // 引用系统的 url 模块
5. var url = require("url");
7. // 引用系统的 querystring 模块
8. var qs = require("querystring");
10. // 引用自定义 module1 模块，名称即为 package.json 里的 name 值
11. var module1 = require("module1");
13. http.createServer(function (request, response) {
14. var query=qs.parse(url.parse(request.url).query);
15. var r = query.r;
16. var h = query.h;
18. if (r !== undefined && h !== undefined) {
19. write(response, "<h1>计算圆柱的面积和体积</h1>\
20. <p>半径 => " + r + "</p>\
21. <p>高 => " + r + "</p>\
22. <p>面积 => " + module1.area(r,h) + "</p>\
23. <p>体积 => " + module1.volume(r,h) + "</p>");
24. } else {
25. write(response, "<h1>计算圆柱的面积和体积</h1>\
26. <p>请输入圆柱的班级或高</p>");
27. }
29. }).listen(2014);
31. function write(response, text) {
32. response.writeHead(200, {
33. "content-type": "text/html"
34. });
35. response.write(text);
36. response.end();
37. }

打开localhost:2014即可看到效果。

## 5、模块路径

A、直接使用模块名称

在nodejs中，引用一个模块（包）名称，遵循以下规范。如在/app3/index.js中引用test模块包

1. var test = require("test");

系统依次搜寻以下路径，直到找到为止：

1. /app3/node\_modules/test.js
2. /node\_modules/test.js

如果是引用以下系统的核心模块也可以直接使用模块名称：

* http：提供HTTP服务器功能。
* url：解析URL。
* fs：即filestream，与文件系统交互。
* querystring：解析URL的查询字符串。
* child\_process：新建子进程。
* util：提供一系列实用小工具。
* path：处理文件路径。
* crypto：提供加密和解密功能，基本上是对OpenSSL的包装。

B、标明模块路径

* ./ => 当前路径
* ../ => 上层路径
* / => 根路径

如：

1. // 引用同级目录的test.js
2. var test = require("./test.js");
4. // 引用上级目录的test.js
5. var test = require("../test.js");
7. // 引用根级目录的test.js
8. var test = require("/test.js");