nodejs课件

|  |  |  |
| --- | --- | --- |
| 修改日期 | 备注 | 修改人 |
| 20190826 | 1~10章节 | fang |
| 20190828 | 10~15章节,阶段节点 | fang |

目录

[nodejs课件 1](#_Toc17666171)

[1. 参考资料 2](#_Toc17666172)

[2. 简单创建web服务, 监听自定义端口 2](#_Toc17666173)

[3. 数据库依赖 2](#_Toc17666174)

[4. npm介绍 3](#_Toc17666175)

[5. Node.js REPL(交互式解释器) 4](#_Toc17666176)

[6. 回调函数讲解 5](#_Toc17666177)

[7. 事件与事件对象 5](#_Toc17666178)

[7.1 原理 5](#_Toc17666179)

[7.2 事件与事件循环 6](#_Toc17666180)

[7.3 事件 事件对象 事件函数 6](#_Toc17666181)

[7.4 Node.js EventEmitter 7](#_Toc17666182)

[8. buffer // TODO 10](#_Toc17666183)

[9. 流 (stream) 12](#_Toc17666184)

[9.1什么是流? 12](#_Toc17666185)

[9.2 管道(pipe) 14](#_Toc17666186)

[10. 文件和目录(file) 15](#_Toc17666187)

[10.1 打开文件 15](#_Toc17666188)

[10.2 写入文件 16](#_Toc17666189)

[10.3 获取文件信息 stat 17](#_Toc17666190)

[10.4 读取文件(与open配套) 18](#_Toc17666191)

[10.5 关闭文件 19](#_Toc17666192)

[10.5 截取文件 20](#_Toc17666193)

[10.6 删除文件 20](#_Toc17666194)

[10.7 创建目录 20](#_Toc17666195)

[10.8 读取目录 21](#_Toc17666196)

[10.9 删除目录 22](#_Toc17666197)

[10. 10 文件模块方法参考手册 22](#_Toc17666198)

[11. http对象 22](#_Toc17666199)

[11.1 文件服务器 23](#_Toc17666200)

[12. crypto(提供通用的加密和哈希算法) 25](#_Toc17666201)

[13 Node.js模块系统 25](#_Toc17666202)

[14. 函数, 匿名函数与服务器 25](#_Toc17666203)

[15. get和post请求 25](#_Toc17666204)

[16. Node.js 路由 25](#_Toc17666205)

[17. node对session的操作 25](#_Toc17666206)

# 参考资料

// 教程地址(菜鸟教程)

https://www.runoob.com/nodejs/nodejs-tutorial.html

// 官网

https://nodejs.org/zh-cn/

中文文档

<http://nodejs.cn/api/>

// 目前版本 10.16.3 (lts)

// 接口文档

https://nodejs.org/api/

https://nodejs.org/dist/latest-v10.x/docs/api/

# 2. 简单创建web服务, 监听自定义端口

var http = require('http');

http.createServer(function (request, response) {

// 发送 HTTP 头部

// HTTP 状态值: 200 : OK

// 内容类型: text/plain

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

// 发送响应数据 "Hello World"

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

}).listen(8888);

// 终端打印如下信息

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

# 3. 数据库依赖

## 3.1 示例代码

安装mysql 模块: npm install mysql

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

const mysql = require("mysql");

const connection = mysql.createConnection({

host: "localhost", //主机地址

user: "root", //数据库用户名

password: "", //数据库用户密码

database: "fang" //数据库名

});

connection.connect(); //数据库连接

connection.query('select \* from Person', function (error, results, fields) {

if (error){

throw error;

} else{

//console.log('The solution is: ', results[0].solution);

console.info(results);

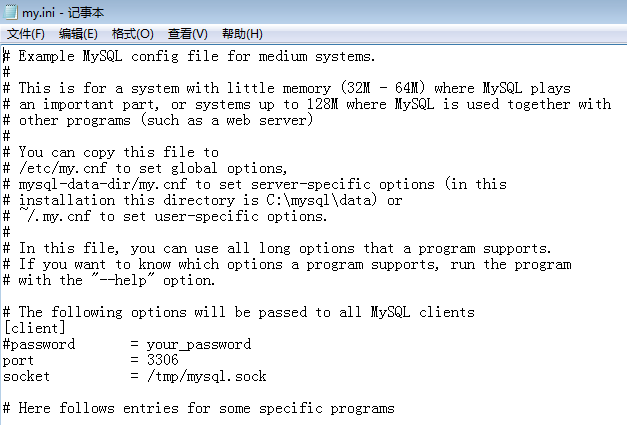
}

}); //检验连接成功

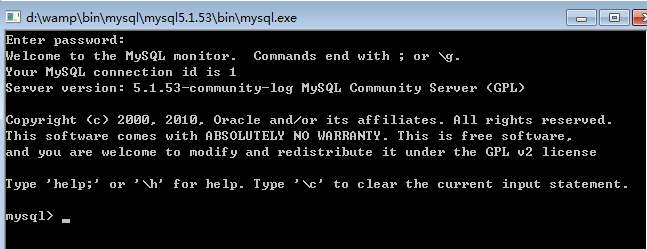
-----------------------

## 3.2 借用wampServer的数据库

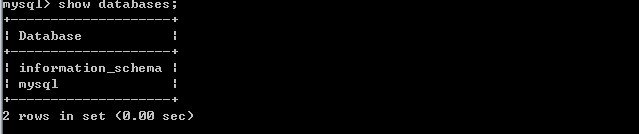
启动wampserver后, 点击mysql的init文件, 查看默认端口,默认是3306



打开mysql的控制台, 默认密码为空, 回车进入mysql



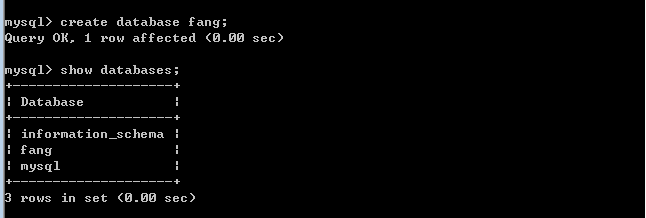
使用命令 查看mysql的所有数据库: show databases;



删除一个数据库的命令为 drop databases 数据库名;

可见, mysql自带了一些数据库, 通常我们不用. 还是自己建数据库

create database fang;



使用 use 命令, 进入相应的数据库 use fang;



使用 show tables; 查看该数据库内的所有表格(初始数据库无表格, 空) Empt set表示查询语句返回了空数据集

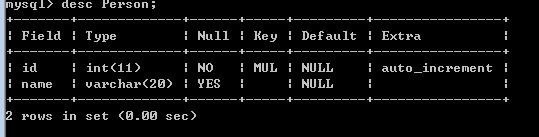


使用create table 命令, 创建一个名为Person的表

create table Person (id int auto\_increment, name varchar(20), key(id) );



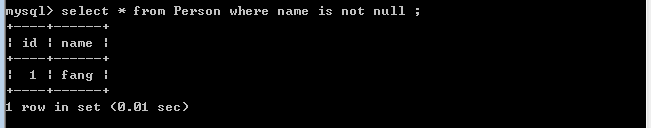
可以使用 desc Person; 查看该表的结构,是否正确.



使用 insert命令, 向这个表中添加一条记录



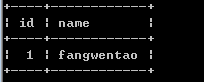
使用select 命令查询记录



使用update 命令修改表记录

update Person set name=’fangwentao’ where id=1;





// 作业1 - 编写一端插入一条记录的代码

# 4. npm介绍

查看版本

$ npm -v

2.3.0

升级或安装

$ sudo npm install npm -g

使用淘宝镜像的命令：

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

安装模块 以express为例

npm install express # 本地安装

npm install express -g # 全局安装

npm uninstall express # 卸载模块

npm ls # 查看所有模块

npm update express # 更新模块

npm search express #搜索模块

如果出现以下错误：

npm err! Error: connect ECONNREFUSED 127.0.0.1:8087

解决办法为：

$ npm config set proxy null

使用 package.json

package.json 位于模块的目录下，用于定义包的属性。

Package.json 属性说明

name - 包名。

version - 包的版本号。

description - 包的描述。

homepage - 包的官网 url 。

author - 包的作者姓名。

contributors - 包的其他贡献者姓名。

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

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

main - main 字段指定了程序的主入口文件，require('moduleName') 就会加载这个文件。这个字段的默认值是模块根目录下面的 index.js。

keywords - 关键字

创建模块 – TODO

# 5. Node.js REPL(交互式解释器)

简单理解为node的控制台, 使用方法这里不做展开

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。

# 6. 回调函数讲解

因为js是单线程的, 通过回调函数, 可以实现异步操作 ,后台不应被阻塞!

同步读取文件, 则最后的输出会等待读取文件结束, 这是一段阻塞代码

var fs = require("fs");

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

console.log(data.toString());

console.log("程序执行结束!");

非阻塞代码示例, 输出不会等待读取文件的结束, 什么时候读完, 什么时候用回调函数处理.

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!");

# 7. 事件与事件对象

## 7.1 原理

Node.js 是单进程单线程应用程序，但是因为 V8 引擎提供的异步执行回调接口，通过这些接口可以处理大量的并发，所以性能非常高。

Node.js 几乎每一个 API 都是支持回调函数的。

Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

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

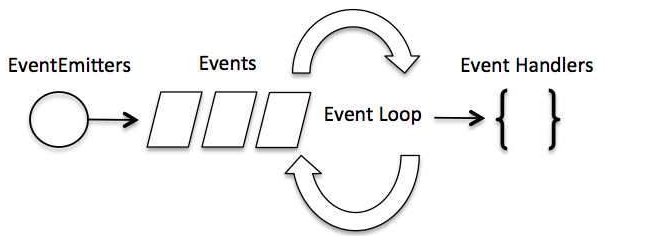
## 7.2 事件与事件循环

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

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

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

在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。



程序主循环用来监听事件, 直到碰到事件的对应回调函数, 将其移出事件队列

## 7.3 事件 事件对象 事件函数

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

// 绑定事件及事件的处理程序

eventEmitter.on('eventName', eventHandler);

// 触发事件 – 执行其事件响应函数

eventEmitter.emit('eventName');

说明: 这里的事件, 与浏览器事件不同, 事件类似于dojo的 topic 是一个自定义的标识(主题, 并不是由window之类预定好的事件名称)

实例:

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

// 创建事件处理程序

var connectHandler = function connected() {

console.log('连接成功。');

// 触发 data\_received 事件

eventEmitter.emit('data\_received');

}

// 绑定 connection 事件处理程序

eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data\_received 事件

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

console.log('数据接收成功。');

});

// 触发 connection 事件

eventEmitter.emit('connection');

console.log("程序执行完毕。");

**特别说明:**

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

// 作业2: 写一个计时器,每2秒的触发某事件, 这个事件的响应是, 打印当前时间的字符串,new Date().toLocalString() 打印5次以后, 触发另一个事件, 变成打印当前时间字符串的 毫秒数 new Date().getTime() , 又打印5次后, 恢复成打印时间字符串, 打印5次以后, 输出”停止输出”,

并关掉计时器.

## 7.4 Node.js EventEmitter

// 引入 events 模块

var events = require('events');

// 创建 eventEmitter 对象

var eventEmitter = new events.EventEmitter();

方法

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

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。 |

**error 事件**

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

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

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

var events = require('events');

var emitter = new events.EventEmitter();

emitter.emit('error');

运行时会显示以下错误：

node.js:201

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

^

Error: Uncaught, unspecified 'error' event.

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

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

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

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

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

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

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

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

**特别说明**

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。

例如 stream 等类, 都是EventEmitter 的实例

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， 事件的监听和发生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。

# 8. buffer // TODO

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

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

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

**Node.js 目前支持的字符编码包括：**

* **ascii** - 仅支持 7 位 ASCII 数据。如果设置去掉高位的话，这种编码是非常快的。
* **utf8** - 多字节编码的 Unicode 字符。许多网页和其他文档格式都使用 UTF-8 。
* **utf16le** - 2 或 4 个字节，小字节序编码的 Unicode 字符。支持代理对（U+10000 至 U+10FFFF）。
* **ucs2** - **utf16le** 的别名。
* **base64** - Base64 编码。
* **latin1** - 一种把 **Buffer** 编码成一字节编码的字符串的方式。
* **binary** - **latin1** 的别名。
* **hex** - 将每个字节编码为两个十六进制字符。

**创建 Buffer 类**

Buffer 提供了以下 API 来创建 Buffer 类：

* **Buffer.alloc(size[, fill[, encoding]])：** 返回一个指定大小的 Buffer 实例，如果没有设置 fill，则默认填满 0
* **Buffer.allocUnsafe(size)：** 返回一个指定大小的 Buffer 实例，但是它不会被初始化，所以它可能包含敏感的数据
* **Buffer.allocUnsafeSlow(size)**
* **Buffer.from(array)：** 返回一个被 array 的值初始化的新的 Buffer 实例（传入的 array 的元素只能是数字，不然就会自动被 0 覆盖）
* **Buffer.from(arrayBuffer[, byteOffset[, length]])：** 返回一个新建的与给定的 ArrayBuffer 共享同一内存的 Buffer。
* **Buffer.from(buffer)：** 复制传入的 Buffer 实例的数据，并返回一个新的 Buffer 实例
* **Buffer.from(string[, encoding])：** 返回一个被 string 的值初始化的新的 Buffer 实例

// 创建一个长度为 10、且用 0 填充的 Buffer。

const buf1 = Buffer.alloc(10);

// 创建一个长度为 10、且用 0x1 填充的 Buffer。

const buf2 = Buffer.alloc(10, 1);

// 创建一个长度为 10、且未初始化的 Buffer。

// 这个方法比调用 Buffer.alloc() 更快，

// 但返回的 Buffer 实例可能包含旧数据，

// 因此需要使用 fill() 或 write() 重写。

const buf3 = Buffer.allocUnsafe(10);

// 创建一个包含 [0x1, 0x2, 0x3] 的 Buffer。

const buf4 = Buffer.from([1, 2, 3]);

// 创建一个包含 UTF-8 字节 [0x74, 0xc3, 0xa9, 0x73, 0x74] 的 Buffer。

const buf5 = Buffer.from('tést');

// 创建一个包含 Latin-1 字节 [0x74, 0xe9, 0x73, 0x74] 的 Buffer。

const buf6 = Buffer.from('tést', 'latin1');

待续..................

# 9. 流 (stream)

// 作业3 :　把　tempfile.txt 的内容, 复制到 temp1. txt中

## 9.1什么是流?

什么是流？流是一种抽象的数据结构。想象水流，当在水管中流动时，就可以从某个地方（例如自来水厂）源源不断地到达另一个地方（比如你家的洗手池）。我们也可以把数据看成是数据流，比如你敲键盘的时候，就可以把每个字符依次连起来，看成字符流。这个流是从键盘输入到应用程序，实际上它还对应着一个名字：标准输入流（stdin）。

如果应用程序把字符一个一个输出到显示器上，这也可以看成是一个流，这个流也有名字：标准输出流（stdout）。流的特点是数据是有序的，而且必须依次读取，或者依次写入，不能像Array那样随机定位。

ps: 如果我们要打开一个文本文件, 我们可以用文件(file)的方式打开, 文本就会以字符串的方式处理, 但是, 如果我们打开的是一个图片, 音视频或者其他二进制文件, 我们就需要将其流的方式读取.

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

所以, request是可以调用on(“data”, callback) 的,从中解析出请求参数(例如post传参 )

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。

有些流用来读取数据，比如从文件读取数据时，可以打开一个文件流，然后从文件流中不断地读取数据。有些流用来写入数据，比如向文件写入数据时，只需要把数据不断地往文件流中写进去就可以了。

在Node.js中，流也是一个对象，我们只需要响应流的事件就可以了：data事件表示流的数据已经可以读取了，end事件表示这个流已经到末尾了，没有数据可以读取了，error事件表示出错了。

下面是一个从文件流读取文本内容的示例：

'use strict';

**var** fs = **require**('fs');

*// 打开一个流:*

**var** rs = fs.createReadStream('sample.txt', 'utf-8');

rs.on('data', **function** (chunk) {

console.log('DATA:')

console.log(chunk);

});

rs.on('end', **function** () {

console.log('END');

});

rs.on('error', **function** (err) {

console.log('ERROR: ' + err);

});

要注意，data事件可能会有多次，每次传递的chunk是流的一部分数据。

要以流的形式写入文件，只需要不断调用write()方法，最后以end()结束：

'use strict';

**var** fs = **require**('fs');

**var** ws1 = fs.createWriteStream('output1.txt', 'utf-8');

ws1.write('使用Stream写入文本数据...\n');

ws1.write('END.');

ws1.end();

**var** ws2 = fs.createWriteStream('output2.txt');

ws2.write(**new** Buffer('使用Stream写入二进制数据...\n', 'utf-8'));

ws2.write(**new** Buffer('END.', 'utf-8'));

ws2.end();

所有可以读取数据的流都继承自stream.Readable，所有可以写入的流都继承自stream.Writable。

讨论: 复制一张图片

## 9.2 管道(pipe)

就像可以把两个水管串成一个更长的水管一样，两个流也可以串起来。一个Readable流和一个Writable流串起来后，所有的数据自动从Readable流进入Writable流，这种操作叫pipe。

在Node.js中，Readable流有一个pipe()方法，就是用来干这件事的。

让我们用pipe()把一个文件流和另一个文件流串起来，这样源文件的所有数据就自动写入到目标文件里了，所以，这实际上是一个复制文件的程序：

'use strict';

**var** fs = **require**('fs');

**var** rs = fs.createReadStream('sample.txt');

**var** ws = fs.createWriteStream('copied.txt');

rs.pipe(ws);

默认情况下，当Readable流的数据读取完毕，end事件触发后，将自动关闭Writable流。如果我们不希望自动关闭Writable流，需要传入参数：

readable**.pipe**(writable, { end: false });

# 10. 文件和目录(file)

File System相关API

https://nodejs.org/dist/latest-v10.x/docs/api/fs.html

var fs = require("fs")

fs模块, 负责文件的操作.

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

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

建议大家使用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞。

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。");

## 10.1 打开文件

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

flags 参数可以是以下值：

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

实例: 创建一个文件, 并输入相应的内容

var newLine = "\n啊收到回复撒肯定封建时代泛海建设的看法和";

fs.appendFile('output.txt', newLine , function(err){

if (err) {

throw err;

}

console.log('ok2');

});

## 10.2 写入文件

写文件:

将数据写入文件是通过fs.writeFile()实现的：

'use strict';

**var** fs = **require**('fs');

**var** data = 'Hello, Node.js';

fs.writeFile('output.txt', data, **function** (err) {

**if** (err) {

console.log(err);

} **else** {

console.log('ok.');

}

});

writeFile()的参数依次为文件名、数据和回调函数。

**如果传入的数据是String，默认按UTF-8编码写入文本文件，如果传入的参数是Buffer，则写入的是二进制文件。**

回调函数由于只关心成功与否，因此只需要一个err参数。

和readFile()类似，writeFile()也有一个同步方法，叫writeFileSync()：

'use strict';

**var** fs = **require**('fs');

**var** data = 'Hello, Node.js';

fs.writeFileSync('output.txt', data);

## 10.3 获取文件信息 stat

fs.stat(path, callback)

主要用途是容错, 判断是文件还是路径

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: 写入文件并判断写入的数据

var fs = require("fs");

console.log("准备写入文件");

fs.writeFile('input.txt', '我是通 过fs.writeFile 写入文件的内容', 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());

});

});

## 10.4 读取文件(与open配套)

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

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。

var fs = require("fs");

var buf = new Buffer.alloc(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());

}

});

});

## 10.5 关闭文件

fs.close(fd, callback)

var fs = require("fs");

var buf = new Buffer.alloc(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");

});

});

});

## 10.5 截取文件

fs.ftruncate(fd, len, callback)

## 10.6 删除文件

fs.unlink(path, callback)

## 10.7 创建目录

参数使用说明如下：

* **path** - 文件路径。
* options 参数可以是：
  + **recursive**- 是否以递归的方式创建目录，默认为 false。
  + **mode** - 设置目录权限，默认为 0777。
* **callback** - 回调函数，没有参数。

var fs = require("fs");

// tmp 目录必须存在

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。");

});

// 递归参数recursive: true 参数，不管创建的目录 /tmp 和 /tmp/a 是否存在：

// fs.mkdir(path[, options], callback)

fs.mkdir('/tmp/a/apple', { recursive: true }, (err) => {

if (err) throw err;

});

## 10.8 读取目录

fs.readdir(path, callback)

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 );

});

});

## 10.9 删除目录

fs.rmdir(path, callback)

## 10. 10 文件模块方法参考手册

<https://www.runoob.com/nodejs/nodejs-fs.html>

# 11. http对象 url 和path

## 11.1 http和node

Node.js开发的目的就是为了用JavaScript编写Web服务器程序。因为JavaScript实际上已经统治了浏览器端的脚本，其优势就是有世界上数量最多的前端开发人员。如果已经掌握了JavaScript前端开发，再学习一下如何将JavaScript应用在后端开发，就是名副其实的全栈了。

然而首先, 我们得先了解一下http协议

<https://www.liaoxuefeng.com/wiki/1016959663602400/1017804782304672>

HTTP服务器

要开发HTTP服务器程序，从头处理TCP连接，解析HTTP是不现实的。这些工作实际上已经由Node.js自带的http模块完成了。应用程序并不直接和HTTP协议打交道，而是操作http模块提供的request和response对象。

request对象封装了HTTP请求，我们调用request对象的属性和方法就可以拿到所有HTTP请求的信息；

response对象封装了HTTP响应，我们操作response对象的方法，就可以把HTTP响应返回给浏览器。

使用node编写一个web服务器, 示例请参考本文档的第2章

'use strict';

// 导入http模块:

var http = require('http');

// 创建http server，并传入回调函数:

var server = http.createServer(function (request, response) {

// 回调函数接收request和response对象,

// 获得HTTP请求的method和url:

console.log(request.method + ': ' + request.url);

// 将HTTP响应200写入response, 同时设置Content-Type: text/html:

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

// 将HTTP响应的HTML内容写入response:

response.end('<h1>Hello world!</h1>');

});

// 让服务器监听8080端口:

server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');

## 11.2 文件服务器

让我们继续扩展一下上面的Web程序。我们可以设定一个目录，然后让Web程序变成一个文件服务器。要实现这一点，我们只需要解析request.url中的路径，然后在本地找到对应的文件，把文件内容发送出去就可以了。

解析URL需要用到Node.js提供的url模块，它使用起来非常简单，通过parse()将一个字符串解析为一个Url对象：

'use strict';

**var** url = **require**('url');

console.log(url.parse('http://user:pass@host.com:8080/path/to/file?query=string#hash'));

处理本地文件目录需要使用Node.js提供的path模块，它可以方便地构造目录：

'use strict';

var path = require('path');

// 解析当前目录:

var workDir = path.resolve('.'); // '/Users/michael'

// 组合完整的文件路径:当前目录+'pub'+'index.html':

var filePath = path.join(workDir, 'pub', 'index.html');

// '/Users/michael/pub/index.html'

使用path模块可以正确处理操作系统相关的文件路径。在Windows系统下，返回的路径类似于C:\Users\michael\static\index.html，这样，我们就不关心怎么拼接路径了。

综上所述, 完整的实现一个文件服务器,使用这个代码

输入服务器目录下的某个文件, 即可返回给浏览器

'use strict';

**var** fs = **require**('fs'),

url = **require**('url'),

path = **require**('path'),

http = **require**('http');

*// 从命令行参数获取root目录，默认是当前目录:*

**var** root = path.resolve(process.argv[2] || '.');

console.log('Static root dir: ' + root);

*// 创建服务器:*

**var** server = http.createServer(**function** (request, response) {

*// 获得URL的path，类似 '/css/bootstrap.css':*

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

*// 获得对应的本地文件路径，类似 '/srv/www/css/bootstrap.css':*

**var** filepath = path.join(root, pathname);

*// 获取文件状态:*

fs.stat(filepath, **function** (err, stats) {

**if** (!err && stats.isFile()) {

*// 没有出错并且文件存在:*

console.log('200 ' + request.url);

*// 发送200响应:*

response.writeHead(200);

*// 将文件流导向response:*

fs.createReadStream(filepath).pipe(response);

} **else** {

*// 出错了或者文件不存在:*

console.log('404 ' + request.url);

*// 发送404响应:*

response.writeHead(404);

response.end('404 Not Found');

}

});

});

server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');

讨论: 覆盖服务器的主页的写法

// 拦截根目录

if( pathname == "/" || pathname == "\\" ){

pathname = "/web/index.html"

}

讨论:

启动一个服务, 将磁盘根目录下的web文件夹的所有的文件的名字都返回给前端页面( 不需要路由, 不解析请求, 直接返回文件列表)

# 12. crypto(提供通用的加密和哈希算法)

略

# 13. Node.js模块系统 \*\*\*\*\*\*\*\*\*

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

Node.js 提供了 exports 和 require 两个对象，其中 exports 是模块公开的接口，require 用于从外部获取一个模块的接口，即所获取模块的 exports 对象。

# 14. 函数, 匿名函数与服务器

参考资料: https://www.runoob.com/nodejs/nodejs-function.html

# 15. get和post请求

## 15.1 获取 URL 的参数(get)

http://localhost:3000/user?name=菜鸟教程&url=www.runoob.com

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

我们可以使用 url.parse 方法来解析 URL 中的参数，代码如下：

## 实例

var http = require('http');

var url = require('url');

var util = require('util');

http.createServer(function(req, res){

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

// 第二个参数, 表示是否解析其属性值, 还是单纯只赋值给对象

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

res.write("网站名：" + params.name);

res.write("\n");

res.write("网站 URL：" + params.url);

res.end();

}).listen(3000);

## 15.2 获取Post 传递的参数

使用'querystring'模块解析一个post请求post = querystring.parse(post);

笔记:

var http = require('http');

var querystring = require('querystring');

var util = require('util');

http.createServer(function(req, res){

// 定义了一个post变量，用于暂存请求体的信息

var post = '';

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

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

post += chunk;

});

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

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

post = querystring.parse(post);

// util模块, util.inspect – 将对象转换成字符串

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

});

}).listen(3000);

作业4 :

使用get请求, 通过id参数, 查询数据库中的某位学生的信息

作业5 :

使用post请求, 修改数据库字段

# 16. Node.js 路由 \*

//TODO

# 17. node对session的操作

<https://www.jb51.net/article/147419.htm>

详细

<https://segmentfault.com/a/1190000010837077>