**Node.js重點整理**

**By Kiki Li**

**2017.6.27**

# Contents

[Contents 2](#_Toc486258461)

[CH 1 Node.js簡介和執行 5](#_Toc486258462)

[CH 1-1 什麼是Node.js 5](#_Toc486258463)

[CH 1-2 運行Node.js 5](#_Toc486258464)

[**Step 1： 開啟Node.js工具** 5](#_Toc486258465)

[**Step2： 版本查詢** 6](#_Toc486258466)

[**Step3： Hello World練習** 6](#_Toc486258467)

[CH 2 Node.js基本應用 7](#_Toc486258468)

[CH 2-1 NPM使用 7](#_Toc486258469)

[**Step 1：查詢npm版本** 7](#_Toc486258470)

[**Step 2：npm安裝 / 升級** 7](#_Toc486258471)

[**Step 3：npm 命令安裝模塊** 8](#_Toc486258472)

[**Step 4：本地安裝模塊** 9](#_Toc486258473)

[**Step 5：全局安裝模塊** 11](#_Toc486258474)

[**Step 6：移除模塊** 12](#_Toc486258475)

[**Step 7：更新模塊** 13](#_Toc486258476)

[**Step 8：搜尋模塊** 13](#_Toc486258477)

[CH 2-2 Node.js REPL(互動式解譯器) 13](#_Toc486258478)

[**Step 1：啟動終端** 14](#_Toc486258479)

[**Step 2：使用變量** 14](#_Toc486258480)

[**Step 3：多行運算式** 14](#_Toc486258481)

[**Step 4：底線(\_)變數** 15](#_Toc486258482)

[CH 2-3函數 15](#_Toc486258483)

[CH 2-4 Node.js组成 16](#_Toc486258484)

[**Step 1：引入 required 模組** 16](#_Toc486258485)

[**Step 2：創建伺服器** 16](#_Toc486258486)

[CH 3 Node.js進階應用I 18](#_Toc486258487)

[CH 3-1 Node.js 回呼函數 18](#_Toc486258488)

[**Step 1：阻塞代碼實例** 18](#_Toc486258489)

[**Step 2：非阻塞代碼實例** 18](#_Toc486258490)

[CH 3-2 Node.js事件迴圈 19](#_Toc486258491)

[CH 3-3 Node.js EventEmitter 20](#_Toc486258492)

[**Step 1：EventEmitter 的用法** 21](#_Toc486258493)

[**Step 2：EventEmitter 的若干個參數** 21](#_Toc486258494)

[**Step 3：EventEmitter監聽移除** 22](#_Toc486258495)

[**Step 4：error事件** 23](#_Toc486258496)

[**參考** 23](#_Toc486258497)

[CH 3-4 Node.js Buffer(緩衝區) 24](#_Toc486258498)

[**Step 1：write實例** 26](#_Toc486258499)

[**Step 2：toString實例** 26](#_Toc486258500)

[**Step 3：將 Buffer 轉換為 JSON** 26](#_Toc486258501)

[**Step 4：緩衝** 27](#_Toc486258502)

[**Step 5：緩衝比較** 27](#_Toc486258503)

[**Step 6：拷貝緩衝** 28](#_Toc486258504)

[**Step 7：緩衝區裁剪** 28](#_Toc486258505)

[**Step 8：緩衝長度** 28](#_Toc486258506)

[**參考** 28](#_Toc486258507)

[CH 4 Node.js進階應用II 33](#_Toc486258508)

[CH 4-1 Node.js Stream(流) 33](#_Toc486258509)

[**Step 1：從流中讀取數據** 33](#_Toc486258510)

[**Step 2：寫入流** 34](#_Toc486258511)

[**Step 3：管道流-文件複製** 35](#_Toc486258512)

[**Step 4：鏈式流-文件壓縮** 36](#_Toc486258513)

[**Step 5：鏈式流-解壓縮** 36](#_Toc486258514)

[CH 4-2 Node.js模組系統 36](#_Toc486258515)

[**Step 1：方法1** 36](#_Toc486258516)

[**Step 2：方法2** 37](#_Toc486258517)

[CH 4-3 Node.js 路由 38](#_Toc486258518)

[**Step 1：建立執行的項目** 38](#_Toc486258519)

[**Step 2：建立Http服務** 38](#_Toc486258520)

[**Step 3：注入服務及啟動** 38](#_Toc486258521)

[**Step 4：執行服務** 39](#_Toc486258522)

[CH 4-4 Node.js GET/POST請求 39](#_Toc486258523)

[**Step 1：GET請求** 39](#_Toc486258524)

[**Step 2：POST請求** 40](#_Toc486258525)

[CH 4-5 Node.js全域物件 41](#_Toc486258526)

[**Step 1：範例-取出完整路徑** 42](#_Toc486258527)

[**Step 2：範例-取出所在目錄** 42](#_Toc486258528)

[**Step 3：範例-間隔時間後執行** 42](#_Toc486258529)

[**Step 4：其他範例** 42](#_Toc486258530)

[**Step 5：console** 42](#_Toc486258531)

[**Step 6：process** 43](#_Toc486258532)

[**參考** 44](#_Toc486258533)

[CH 5 Node.js進階應用III 47](#_Toc486258534)

[CH 5-1 Node.js文件系统 47](#_Toc486258535)

[**Step 1：非同步和同步** 47](#_Toc486258536)

[**Step 2：打開文件** 47](#_Toc486258537)

[**Step 3：獲取檔資訊** 48](#_Toc486258538)

[**Step 4：寫入文件** 49](#_Toc486258539)

[**Step 5：讀取文件** 50](#_Toc486258540)

[**Step 6：關閉文件** 50](#_Toc486258541)

[**Step 7：截取文件** 50](#_Toc486258542)

[**Step 8：删除文件** 52](#_Toc486258543)

[**Step 9：創建目錄** 53](#_Toc486258544)

[**Step 10：讀取目錄** 53](#_Toc486258545)

[**Step 11：刪除目錄** 54](#_Toc486258546)

[**參考** 55](#_Toc486258547)

[CH 5-2 Node.js常用工具util 59](#_Toc486258548)

[**Step 1：util.inherits** 59](#_Toc486258549)

[**Step 2：util.inspect** 60](#_Toc486258550)

[**Step 3：util.isArray(object)** 60](#_Toc486258551)

[**Step 4：util.isRegExp(object)** 61](#_Toc486258552)

[**Step 5：util.isDate(object)** 61](#_Toc486258553)

[**Step 6：util.isError(object)** 61](#_Toc486258554)

[CH 5-3 Node.js工具模組 62](#_Toc486258555)

[**Step 1：OS 模組** 62](#_Toc486258556)

[**Step 2：Path模組** 63](#_Toc486258557)

[**Step 3：Net模組** 63](#_Toc486258558)

[**Step 4：DNS模組** 67](#_Toc486258559)

[**Step 5：Domain模組** 68](#_Toc486258560)

[CH 5-4 Node.js Web模組 69](#_Toc486258561)

[CH 5-5 Node.js Express 框架 71](#_Toc486258562)

[**Step 1：安裝 Express** 71](#_Toc486258563)

[**Step 2： Express框架範例** 72](#_Toc486258564)

[**Step 3： Express** **request 和 response** 72](#_Toc486258565)

[CH 5-6 Node.js RESTful API 75](#_Toc486258566)

[**Step 1： RESTful Web Services** 75](#_Toc486258567)

# CH 1 Node.js簡介和執行

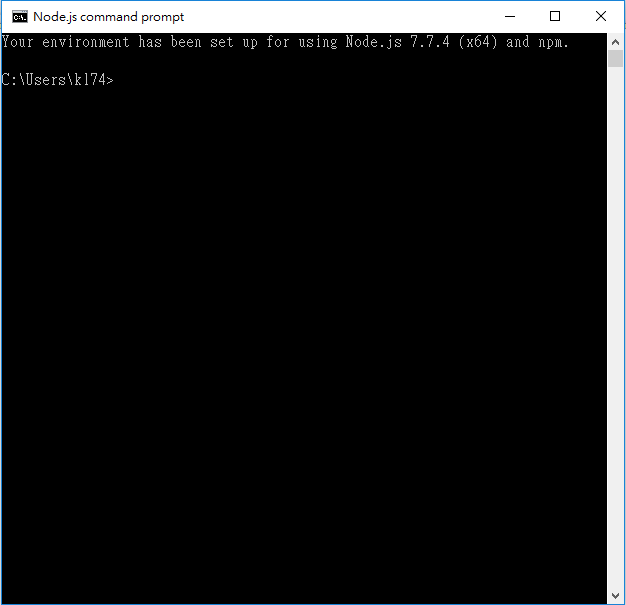
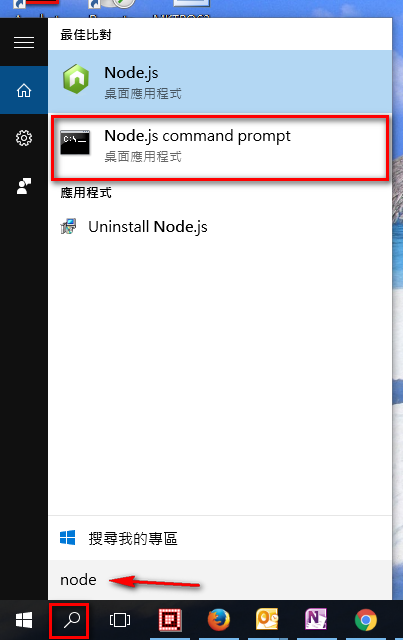
## CH 1-1 什麼是Node.js

* Node.js 是運行在服務端的 JavaScript
* 是基於Chrome JavaScript 運行時建立的一個平臺。
* 是一個事件驅動I/O服務端JavaScript環境，基於Google的V8引擎

## CH 1-2 運行Node.js

**Step 1： 開啟Node.js工具**

於開始功能表中，可透過查詢[node]快速開啟node.js工具



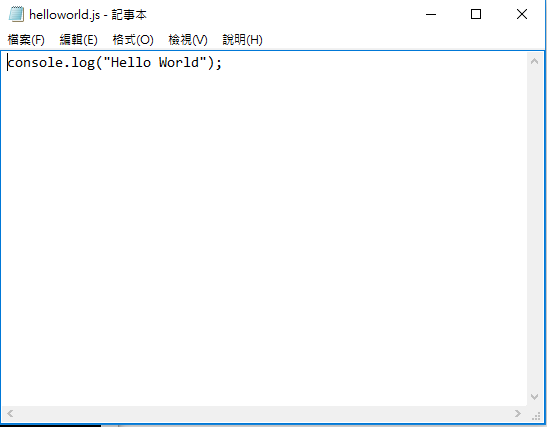
**Step2： 版本查詢**

輸入node –v



**Step3： Hello World練習**

於記事本寫入console.log("Hello World");



檔名存成helloworld.js



切入該目錄，並通過 node命令來執行：

node helloworld.js



就會在終端輸出 Hello World。

# CH 2 Node.js基本應用

## CH 2-1 NPM使用

NPM是隨同NodeJS一起安裝的包管理工具

* 允許用戶從NPM伺服器下載別人編寫的第三方包到本地使用。
* 允許用戶從NPM伺服器下載並安裝別人編寫的命令列程序到本地使用。
* 允許使用者將自己編寫的包或命令列程式上傳到NPM伺服器供別人使用。

**Step 1：查詢npm版本**

測試是否安裝成功:

npm –v



**Step 2：npm安裝 / 升級**

舊版本可透過命令升級:

npm install npm –g





**Step 3：npm 命令安裝模塊**

於指定目錄下，執行

npm install



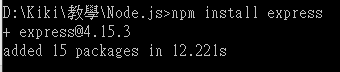
於指定路徑下找到[node\_modules]新的目錄



**Step 4：本地安裝模塊**

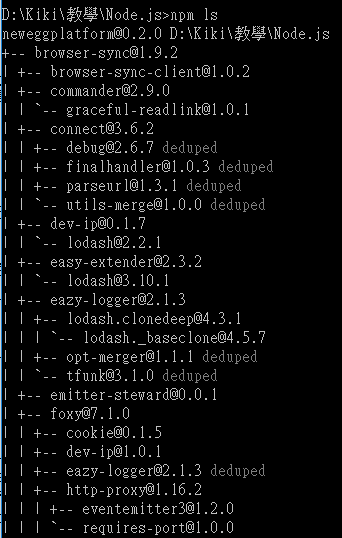
安裝常用的 Node.js web框架模組 express:

本地安裝npm install express ，本範例以: express安裝為例子，若安裝其他模塊則為npm install (名稱)以下範例均相同。

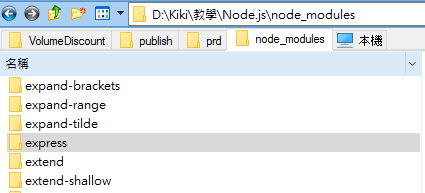


查看包是否存在:

npm ls 或 npm list



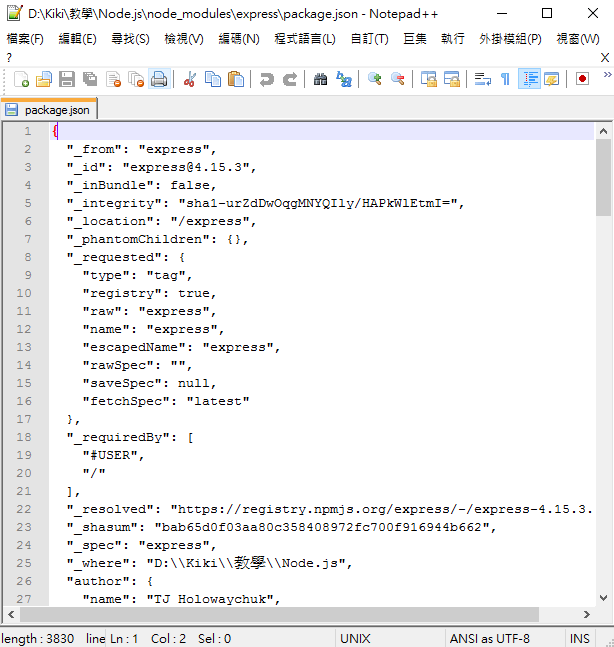
或查看express就放在node\_modules 目錄中，可以通過 require() 來引入本地安裝的包。



因此在代碼中只需要通過 require('express') 的方式就好，無需指定協力廠商包路徑。

|  |
| --- |
| var express = require('express'); |

package.json 位於模組的目錄下，用於定義包的屬性，node\_modules/express/package.json 內容：



name - 包名。

version - 包的版本號。

description - 包的描述。

homepage - 包的官網 url 。

author - 包的作者姓名。

contributors - 包的其他貢獻者姓名。

dependencies - 依賴包列表。如果依賴包沒有安裝，npm 會自動將依賴包安裝在 node\_module 目錄下。

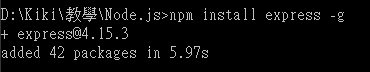
repository - 包代碼存放的地方的類型，可以是 git 或 svn，git 可在 Github 上。

main - main 欄位是一個模組ID，它是一個指向你程式的主要專案。就是說，如果你包的名字叫 express，然後用戶安裝它，然後require("express")。

**Step 5：全局安裝模塊**

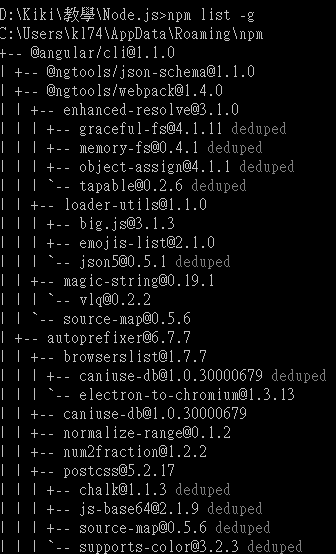
或全局安裝 npm install express –g

全局安裝的路徑為C:\Users\(名稱)\AppData\Roaming\npm\node\_modules

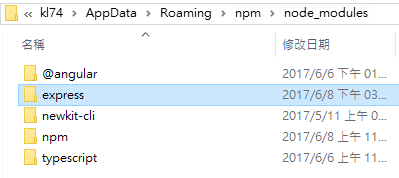


使用以下命令來查看所有全域安裝的模組：

npm list –g



全域安裝包放在 /usr/local 下或者你 node 的安裝目錄，可以直接在命令列裡使用。



**Step 6：移除模塊**

使用以下命令來卸載 Node.js的express項目

本地移除npm uninstall express



全局移除npm uninstall express -g



[express]該目錄被移除

**Step 7：更新模塊**

我們可以使用以下命令更新模組：

本地更新npm update express

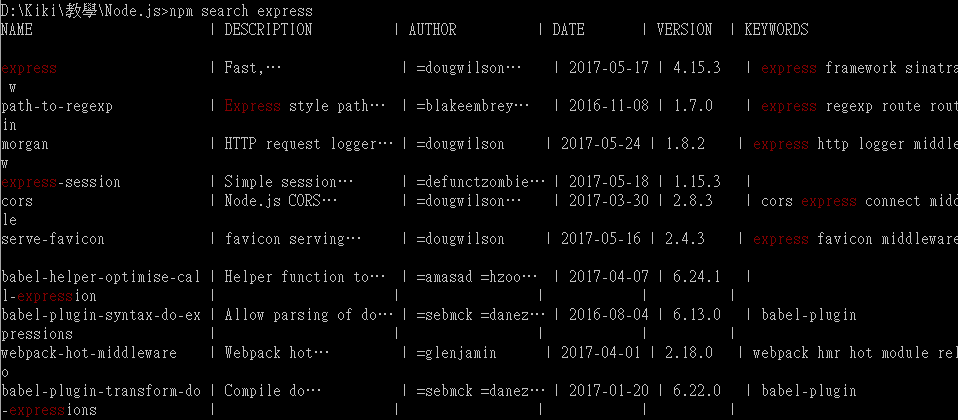
全局更新npm update express -g

**Step 8：搜尋模塊**

使用以下命令搜尋模組：

本地搜尋npm search express

全局搜尋npm search express -g



## CH 2-2 Node.js REPL(互動式解譯器)

類似 Window 系統的終端，可以在終端中輸入命令，並接收系統的回應

* 讀取 - 讀取用戶輸入，解析輸入了Javascript 資料結構並存儲在記憶體中。
* 執行 - 執行輸入的資料結構
* 列印 - 輸出結果
* 迴圈 - 迴圈操作以上步驟直到使用者兩次按下 ctrl-c 按鈕退出。
* REPL 命令:

ctrl + c - 退出當前終端。

ctrl + c 按下兩次 - 退出 Node REPL。

ctrl + d - 退出 Node REPL.

向上/向下 鍵 - 查看輸入的歷史命令

tab 鍵 - 列出當前命令

.help - 列出使用命令

.break - 退出多行運算式

.clear - 退出多行運算式

.save filename - 保存當前的 Node REPL 會話到指定檔案

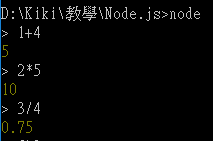
.load filename - 載入當前 Node REPL 會話的檔案內容。

**Step 1：啟動終端**

啟動 Node 的終端：

Node

輸入簡單的表達運算



**Step 2：使用變量**

輸入

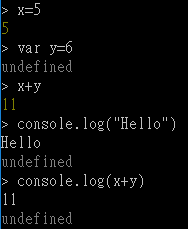
x=5

var y=6

x+y

console.log("Hello")

console.log(x+y)



**Step 3：多行運算式**

輸入> var x=5

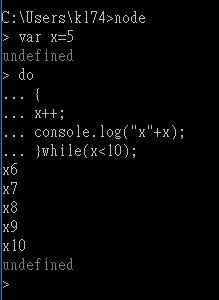
do {

x++;

console.log("x"+x);

}while(x<10);

他會顯示x<10的結果，印出6~10的資訊



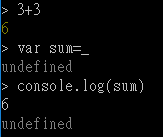
**Step 4：底線(\_)變數**

輸入以下，\_會取出上一次運算的結果存入變量

3+3

var sum=\_

console.log(sum)



## CH 2-3函數

Node.js中函數的使用與Javascript類似

創建檔案fun.js

|  |
| --- |
| function say(word) {  console.log(word);  }  function execute(someFunction, value) {  someFunction(value);  }  execute(say, "Hello"); //將say函數傳入execute實做 |
| //匿名函數寫法  function execute(someFunction, value) {  someFunction(value);  }  execute(function(word){ console.log(word) }, "Hello"); |

執行 node fun.js



## CH 2-4 Node.js组成

Node.js 應用是由哪幾部分組成:

* 引入 required 模組：我們可以使用 require 指令來載入 Node.js 模組。

**Step 1：引入 required 模組**

我們使用 require 指令來載入 http 模組，並將產生實體的賦值變數，實例如下:

|  |
| --- |
| var http = require("http"); |

**Step 2：創建伺服器**

* 創建伺服器：伺服器可以監聽用戶端的請求，類似於 Apache 、Nginx 等 HTTP 伺服器。
* 接收請求與回應請求：用戶端可以使用流覽器或終端發送 HTTP 請求，伺服器接收請求後返回回應資料

使用 http.createServer() 方法創建伺服器，並使用 listen 方法綁定 8888 埠，創建一個叫 server.js 的文件

|  |
| --- |
| var http = require("http");  http.createServer(function(request, response) {  //發送 HTTP 頭部  // HTTP 狀態值: 200 : OK  //內容類別型: text/plain  response.writeHead(200, {"Content-Type": "text/plain"});  //發送回應資料 "Hello World"  response.write("Hello World");  response.end();  }).listen(8888);  // 終端列印如下資訊  console.log("Server has started."); |

第一行請求（require）Node.js 自帶的 http 模組，並且把它賦值給 http 變數。

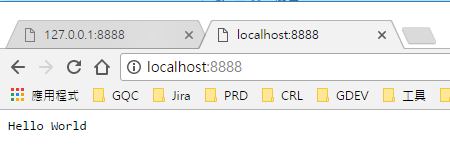
接下來我們調用 http 模組提供的函數： createServer 。這個函數會返回 一個物件，這個物件有一個叫做 listen 的方法，這個方法有一個數值參數， 指定這個 HTTP 伺服器監聽的埠號。

使用 node 命令執行以上的代碼：

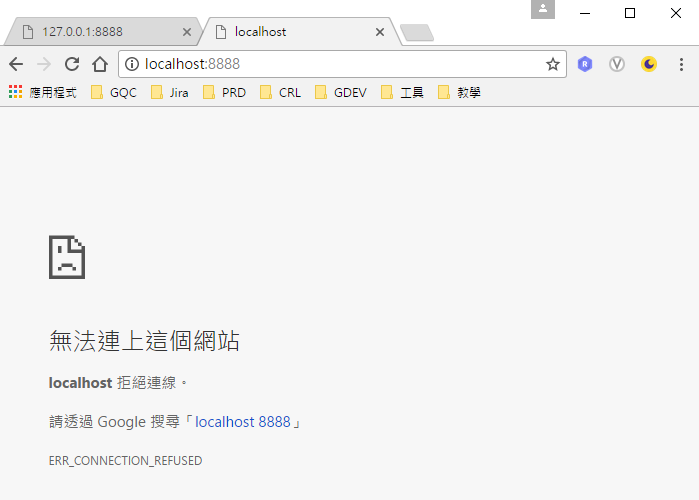
node server.js



開啟網頁<http://127.0.0.1:8888/> 或 <http://localhost:8888/>



當關閉Node.js工具時，網頁無法連線



# CH 3 Node.js進階應用I

## CH 3-1 Node.js 回呼函數

Node 使用了大量的回呼函數，Node 所有 API 都支持回呼函數。我們可以一邊讀取檔，一邊執行其他命令，在檔讀取完成後，我們將檔內容作為回呼函數的參數返回。

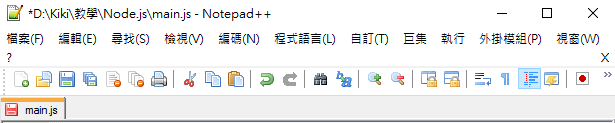
**Step 1：阻塞代碼實例**

此為同步處理，創建一個檔 input.txt



www.newegg.com.tw

創建 main.js 文件



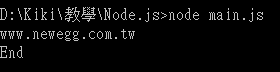
var fs = require("fs");

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

console.log(data.toString());

console.log("End");

於指定路徑下執行指令node main.js



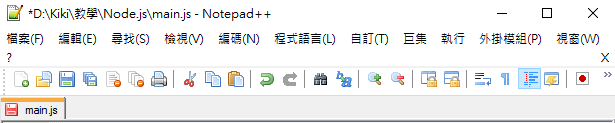
**Step 2：非阻塞代碼實例**

此為異步處理，創建一個檔 input.txt



www.newegg.com.tw

創建 main.js 文件



var fs = require("fs");

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

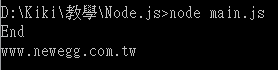
if (err) return console.error(err);

console.log(data.toString());

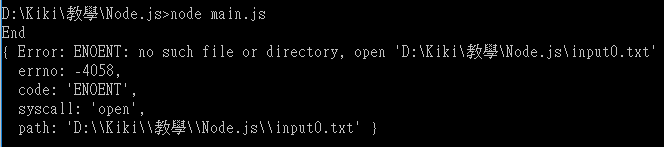
});

console.log("End");

於指定路徑下執行指令node main.js



若將檔名input.txt改為input0.txt不在在的檔案，會寫顯示錯誤訊息



第一個實例在檔案讀取完後才執行完程式。 第二個實例我們不需要等待檔案讀取完，這樣就可以在讀取檔案時同時執行接下來的代碼。

## CH 3-2 Node.js事件迴圈

* 是單進程單執行緒應用程式
* 的每一個 API 都是非同步的
* 所有的事件機制都是用觀察者模式實現。
* 單執行緒類似進入一個while(true)的事件迴圈，直到沒有事件觀察者退出

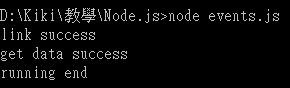
事件驅動程式：創建events.js檔

我們可以通過引入 events 模組，並通過產生實體 EventEmitter 類來綁定和監聽事件

|  |
| --- |
| // 引入 events 模組  var events = require('events');  // 創建 eventEmitter 對象  var eventEmitter = new events.EventEmitter();  // 創建事件處理程序  var connectHandler = function connected() {  console.log(' link success');    // 觸發 data\_received 事件  eventEmitter.emit('data\_received');  }  // 綁定 connection 事件處理常式  eventEmitter.on('connection', connectHandler);    // 使用匿名函數綁定 data\_received 事件  eventEmitter.on('data\_received', function(){  console.log(' get data success');  });  // 觸發 connection 事件  eventEmitter.emit('connection');  console.log("running end"); |

執行代碼

node events.js



## CH 3-3 Node.js EventEmitter

EventEmitter 的核心就是事件觸發與事件監聽器功能的封裝。

通過require("events");來訪問該模組。

|  |
| --- |
| // 引入 events 模組  var events = require('events');  // 創建 eventEmitter 對象  var eventEmitter = new events.EventEmitter();  //監聽事件新增方法1  eventEmitter.**addListener**('名稱', function…);  //監聽事件新增方法2  eventEmitter.on('名稱', function…);  …  //監聽數量統計  var eventListeners = require('events').EventEmitter.**listenerCount**(eventEmitter,'名稱');  //監聽事件觸發  eventEmitter.emit('名稱');  //監聽事件移除  eventEmitter.**removeListener**('名稱', 要移除的function名稱…); |

**Step 1：EventEmitter 的用法**

創建EventEmitter.js

|  |
| --- |
| var EventEmitter = require('events').EventEmitter;  var event = new EventEmitter();  event.on('some\_event', function() {  console.log('some\_event is running');  });  setTimeout(function() {  event.emit('some\_event');  }, 1000); //運行這段代碼，1 秒後控制台輸出了 'some\_event 事件觸發'。 |

於指定路徑下指令node EventEmitter.js，停留了一秒才顯示”some\_event is running”

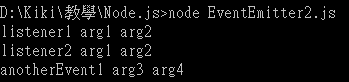


**Step 2：EventEmitter 的若干個參數**

EventEmitter 的每個事件由一個事件名和若干個參數組成，新增檔案EventEmitter2.js

|  |
| --- |
| var events = require('events');  var emitter = new events.EventEmitter();  emitter.on('someEvent', function(arg1, arg2) { //on 函數用於綁定事件函數  console.log('listener1', arg1, arg2);  });  emitter.on('someEvent', function(arg1, arg2) {  console.log('listener2', arg1, arg2);  });  emitter.on('anotherEvent', function(arg1, arg2) {  console.log('anotherEvent1', arg1, arg2);  });  emitter.emit('someEvent', 'arg1', 'arg2'); //emit 屬性用於觸發一個事件，觸發someEvent事件  emitter.emit('anotherEvent', 'arg3', 'arg4'); //觸發anotherEvent事件 |

執行node EventEmitter2.js

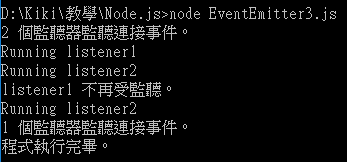


**Step 3：EventEmitter監聽移除**

新增檔案EventEmitter3.js

|  |
| --- |
| var events = require('events');  var eventEmitter = new events.EventEmitter();  // 監聽器 #1  var listener1 = function listener1() {  console.log('Running listener1');  }  // 監聽器 #2  var listener2 = function listener2() {  console.log('Running listener2 ');  }  // 綁定 connection 事件，處理函數為 listener1  eventEmitter.**addListener**('connection', listener1); //方法1  // 綁定 connection 事件，處理函數為 listener2  eventEmitter.on('connection', listener2); //方法2  var eventListeners = require('events').EventEmitter.**listenerCount**(eventEmitter,'connection');//監聽數量統計  console.log(eventListeners + " 個監聽器監聽連接事件。");  // 處理 connection 事件  eventEmitter.emit('connection');//第一次觸發  // 移除監綁定的 listener1 函數  eventEmitter.**removeListener**('connection', listener1); //刪除第一項監聽事件  console.log("listener1 不再受監聽。");  // 觸發連接事件  eventEmitter.emit('connection'); //第二次觸發  eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');//監聽數量統計  console.log(eventListeners + " 個監聽器監聽連接事件。");  console.log("程式執行完畢。"); |

執行node EventEmitter3.js



**Step 4：error事件**

當 error 被觸發時，EventEmitter 規定如果沒有響 應的監聽器，Node.js 會把它當作異常，退出程式並輸出錯誤資訊。

|  |
| --- |
| var events = require('events');  var emitter = new events.EventEmitter();  emitter.emit('error'); |

**參考**

**方法**

1.addListener(event, listener)

為指定事件添加一個監聽器到監聽器陣列的尾部。

2.on(event, listener)

為指定事件註冊一個監聽器，接受一個字串 event 和一個回呼函數。

server.on('connection', function (stream) {

console.log('someone connected!');

});

3.once(event, listener)

為指定事件註冊一個單次監聽器，即 監聽器最多只會觸發一次，觸發後立刻解除該監聽器。

server.once('connection', function (stream) {

console.log('Ah, we have our first user!');

});

4.removeListener(event, listener)

移除指定事件的某個監聽器，監聽器必須是該事件已經註冊過的監聽器。

它接受兩個參數，第一個是事件名稱，第二個是回呼函數名稱。

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監聽器， 如果指定事件，則移除指定事件的所有監聽器。

6.setMaxListeners(n)

預設情況下， EventEmitters 如果你添加的監聽器超過 10 個就會輸出警告資訊。 setMaxListeners 函數用於提高監聽器的預設限制的數量。

7.listeners(event)

返回指定事件的監聽器陣列。

8.emit(event, [arg1], [arg2], [...])

按參數的循序執行每個監聽器，如果事件有註冊監聽返回 true，否則返回 false。

**類方法**

1.listenerCount(emitter, event)

返回指定事件的監聽器數量。

**事件**

1.newListener

event - 字串，事件名稱

listener - 處理事件函數

該事件在添加新監聽器時被觸發。

2.removeListener.

event - 字串，事件名稱

listener - 處理事件函數

從指定監聽器陣列中刪除一個監聽器。需要注意的是，此操作將會改變處於被刪監聽器之後的那些監聽器的索引。

## CH 3-4 Node.js Buffer(緩衝區)

處理像TCP流或檔流時，必須使用到二進位資料。因此在 Node.js中，定義了一個 Buffer 類，該類用來創建一個專門存放二進位資料的緩存區，如果 buffer 空間不足， 則只會寫入部分字串。。

**創建方法:**

創建長度為 10

|  |
| --- |
| var buf = new Buffer(10); |

創建陣列

|  |
| --- |
| var buf = new Buffer([10, 20, 30, 40, 50]); |

創建字串

|  |
| --- |
| var buf = new Buffer("www.runoob.com", "utf-8"); |

**寫入緩衝區的語法**

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

參數

string - 寫入緩衝區的字串。

offset - 緩衝區開始寫入的索引值，預設為 0 。

length - 寫入的位元組數，預設為 buffer.length

encoding - 使用的編碼。默認為 'utf8' 。

從緩衝區讀取數據

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

參數

encoding - 使用的編碼。默認為 'utf8' 。

start - 指定開始讀取的索引位置，預設為 0。

end - 結束位置，默認為緩衝區的末尾。

緩衝區合併

|  |
| --- |
| Buffer.concat(list[, totalLength]) |

參數

list - 用於合併的 Buffer 物件陣列清單。

totalLength - 指定合併後Buffer對象的總長度。

緩衝區比較

|  |
| --- |
| buf.compare(otherBuffer); |

參數

otherBuffer - 與 buf 物件比較的另外一個 Buffer 物件。

拷貝緩衝區

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

參數

targetBuffer - 要拷貝的 Buffer 物件。

targetStart - 數字, 可選, 默認: 0

sourceStart - 數字, 可選, 默認: 0

sourceEnd - 數字, 可選, 默認: buffer.length

緩衝區裁剪

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

參數

start - 數字, 可選, 默認: 0

end - 數字, 可選, 默認: buffer.length

緩衝區長度

|  |
| --- |
| buf.length; |

**Step 1：write實例**

創建檔案brffer1.js

|  |
| --- |
| buf = new Buffer(256);  len = buf.write("www.newegg.com");  console.log("buf寫入長度 : "+ len); |

執行node brffer1.js

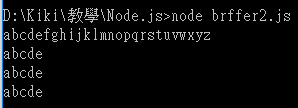


**Step 2：toString實例**

創建檔案brffer2.js

|  |
| --- |
| buf = new Buffer(26);  for (var i = 0 ; i < 26 ; i++) {  buf[i] = i + 97;  }  console.log( buf.toString('ascii')); // 輸出: abcdefghijklmnopqrstuvwxyz  console.log( buf.toString('ascii',0,5)); // 輸出: abcde  console.log( buf.toString('utf8',0,5)); // 輸出: abcde  console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 編碼, 並輸出: abcde |

執行node brffer2.js

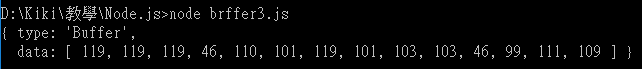


**Step 3：將 Buffer 轉換為 JSON**

創建檔案brffer3.js

|  |
| --- |
| var buf = new Buffer('www.newegg.com');  var json = buf.toJSON(buf);  console.log(json); |

執行node brffer3.js



**Step 4：緩衝**

創建檔案brffer4.js

|  |
| --- |
| var buffer1 = new Buffer('新蛋生活網');  var buffer2 = new Buffer('www.newegg.com.tw');  var buffer3 = Buffer.concat([buffer1,buffer2]);  console.log("buffer3 内容: " + buffer3.toString()); |

執行node brffer4.js

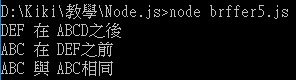


**Step 5：緩衝比較**

創建檔案brffer5.js

|  |
| --- |
| var buffer1 = new Buffer('DEF');  var buffer2 = new Buffer('ABCD');  var result = buffer1.compare(buffer2);  compareTest(result,buffer1,buffer2);  buffer1 = new Buffer('ABC');  buffer2 = new Buffer('DEF');  result = buffer1.compare(buffer2);  compareTest(result,buffer1,buffer2);  buffer1 = new Buffer('ABC');  buffer2 = new Buffer('ABC');  result = buffer1.compare(buffer2);  compareTest(result,buffer1,buffer2);  function compareTest(res,bf1,bf2) {  if(res < 0) {  console.log(bf1 + " 在 " + bf2 + "之前");  }else if(res == 0){  console.log(bf1 + " 與 " + bf2 + "相同");  }else {  console.log(bf1 + " 在 " + bf2 + "之後");  } |

執行node brffer5.js



**Step 6：拷貝緩衝**

創建檔案brffer6.js

|  |
| --- |
| var buffer1 = new Buffer('ABC');  // 拷貝一個緩衝區  var buffer2 = new Buffer(3);  buffer1.copy(buffer2);  console.log("buffer2 content: " + buffer2.toString()); |

執行node brffer6.js



**Step 7：緩衝區裁剪**

創建檔案brffer7.js

|  |
| --- |
| var buffer1 = new Buffer('ABCDEFG');  // 剪切缓冲区  var buffer2 = buffer1.slice(0,2);  console.log("buffer2 content: " + buffer2.toString()); |

執行node brffer7.js



**Step 8：緩衝長度**

創建檔案brffer8.js

|  |
| --- |
| var buffer = new Buffer('www.newegg.com');  // 緩衝區長度  console.log("buffer length: " + buffer.length); |

執行node brffer8.js



**參考**

1new Buffer(size)

分配一個新的 size 大小單位為8位元位元組的 buffer。 注意, size 必須小於 kMaxLength，否則，將會拋出異常 RangeError。

2.new Buffer(buffer)

拷貝參數 buffer 的資料到 Buffer 實例。

3.new Buffer(str[, encoding])

分配一個新的 buffer ，其中包含著傳入的 str 字串。 encoding 編碼方式預設為 'utf8'。

4.buf.length

返回這個 buffer 的 bytes 數。注意這未必是 buffer 裡面內容的大小。length 是 buffer 物件所分配的記憶體數，它不會隨著這個 buffer 物件內容的改變而改變。

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

根據參數 offset 偏移量和指定的 encoding 編碼方式，將參數 string 資料寫入buffer。 offset 偏移量預設值是 0, encoding 編碼方式預設是 utf8。 length 長度是將要寫入的字串的 bytes 大小。 返回 number 類型，表示寫入了多少 8 位元位元組流。如果 buffer 沒有足夠的空間來放整個 string，它將只會只寫入部分字串。 length 默認是 buffer.length - offset。 這個方法不會出現寫入部分字元。

6.buf.writeUIntLE(value, offset, byteLength[, noAssert])

將 value 寫入到 buffer 裡， 它由 offset 和 byteLength 決定，最高支持 48 位不帶正負號的整數，小端對齊，例如：

var b = new Buffer(6);

b.writeUIntBE(0x1234567890ab, 0, 6);

// <Buffer 12 34 56 78 90 ab>

noAssert 值為 true 時，不再驗證 value 和 offset 的有效性。 默認是 false。

7.buf.writeUIntBE(value, offset, byteLength[, noAssert])

將 value 寫入到 buffer 裡， 它由 offset 和 byteLength 決定，最高支持 48 位不帶正負號的整數，大端對齊。noAssert 值為 true 時，不再驗證 value 和 offset 的有效性。 默認是 false。

8.buf.writeIntLE(value, offset, byteLength[, noAssert])

將value 寫入到 buffer 裡， 它由offset 和 byteLength 決定，最高支援48位元有符號整數，小端對齊。noAssert 值為 true 時，不再驗證 value 和 offset 的有效性。 默認是 false。

9.buf.writeIntBE(value, offset, byteLength[, noAssert])

將value 寫入到 buffer 裡， 它由offset 和 byteLength 決定，最高支援48位元有符號整數，大端對齊。noAssert 值為 true 時，不再驗證 value 和 offset 的有效性。 默認是 false。

10.buf.readUIntLE(offset, byteLength[, noAssert])

支援讀取 48 位元以下的無符號數位，小端對齊。noAssert 值為 true 時， offset 不再驗證是否超過 buffer 的長度，默認為 false。

11.buf.readUIntBE(offset, byteLength[, noAssert])

支援讀取 48 位元以下的無符號數位，大端對齊。noAssert 值為 true 時， offset 不再驗證是否超過 buffer 的長度，默認為 false。

12.buf.readIntLE(offset, byteLength[, noAssert])

支援讀取 48 位元以下的有符號數位，小端對齊。noAssert 值為 true 時， offset 不再驗證是否超過 buffer 的長度，默認為 false。

13.buf.readIntBE(offset, byteLength[, noAssert])

支援讀取 48 位元以下的有符號數位，大端對齊。noAssert 值為 true 時， offset 不再驗證是否超過 buffer 的長度，默認為 false。

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

根據 encoding 參數（預設是 'utf8'）返回一個解碼過的 string 類型。還會根據傳入的參數 start (默認是 0) 和 end (默認是 buffer.length)作為取值範圍。

15.buf.toJSON()

將 Buffer 實例轉換為 JSON 物件。

16.buf[index]

獲取或設置指定的位元組。返回值代表一個位元組，所以返回值的合法範圍是十六進位0x00到0xFF 或者十進位0至 255。

17.buf.equals(otherBuffer)

比較兩個緩衝區是否相等，如果是返回 true，否則返回 false。

18.buf.compare(otherBuffer)

比較兩個 Buffer 物件，返回一個數位，表示 buf 在 otherBuffer 之前，之後或相同。

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

buffer 拷貝，源和目標可以相同。 targetStart 目標開始偏移和 sourceStart 源開始偏移默認都是 0。 sourceEnd 源結束位置偏移默認是源的長度 buffer.length 。

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

剪切 Buffer 物件，根據 start(預設是 0 ) 和 end (默認是 buffer.length ) 偏移和裁剪了索引。 負的索引是從 buffer 尾部開始計算的。

21.buf.readUInt8(offset[, noAssert])

根據指定的偏移量，讀取一個無符號 8 位元整數。若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 如果這樣 offset 可能會超出buffer 的末尾。默認是 false。

22.buf.readUInt16LE(offset[, noAssert])

根據指定的偏移量，使用特殊的 endian 位元組序格式讀取一個無符號 16 位元整數。若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出 buffer 的末尾。默認是 false。

23.buf.readUInt16BE(offset[, noAssert])

根據指定的偏移量，使用特殊的 endian 位元組序格式讀取一個無符號 16 位元整數，大端對齊。若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出 buffer 的末尾。默認是 false。

24.buf.readUInt32LE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian 位元組序格式讀取一個無符號 32 位元整數，小端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer 的末尾。默認是 false。

25.buf.readUInt32BE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian 位元組序格式讀取一個無符號 32 位元整數，大端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer 的末尾。默認是 false。

26.buf.readInt8(offset[, noAssert])

根據指定的偏移量，讀取一個有符號 8 位元整數。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出 buffer 的末尾。默認是 false。

27.buf.readInt16LE(offset[, noAssert])

根據指定的偏移量，使用特殊的 endian 格式讀取一個 有符號 16 位元整數，小端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出 buffer 的末尾。默認是 false。

28.buf.readInt16BE(offset[, noAssert])

根據指定的偏移量，使用特殊的 endian 格式讀取一個 有符號 16 位元整數，大端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出 buffer 的末尾。默認是 false。

29.buf.readInt32LE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian 位元組序格式讀取一個有符號 32 位元整數，小端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer 的末尾。默認是 false。

30.buf.readInt32BE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian 位元組序格式讀取一個有符號 32 位元整數，大端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer 的末尾。默認是 false。

31.buf.readFloatLE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian 位元組序格式讀取一個 32 位雙浮點數，小端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer的末尾。默認是 false。

32.buf.readFloatBE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian 位元組序格式讀取一個 32 位雙浮點數，大端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer的末尾。默認是 false。

33.buf.readDoubleLE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian位元組序格式讀取一個 64 位雙精度數，小端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer 的末尾。默認是 false。

34.buf.readDoubleBE(offset[, noAssert])

根據指定的偏移量，使用指定的 endian位元組序格式讀取一個 64 位雙精度數，大端對齊。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 offset 可能會超出buffer 的末尾。默認是 false。

35.buf.writeUInt8(value, offset[, noAssert])

根據傳入的 offset 偏移量將 value 寫入 buffer。注意：value 必須是一個合法的無符號 8 位元整數。 若參數 noAssert 為 true 將不會驗證 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則不要使用。默認是 false。

36.buf.writeUInt16LE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個合法的無符號 16 位元整數，小端對齊。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出buffer的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

37.buf.writeUInt16BE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個合法的無符號 16 位元整數，大端對齊。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出buffer的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

38.buf.writeUInt32LE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式(LITTLE-ENDIAN:小位元組序)將 value 寫入buffer。注意：value 必須是一個合法的無符號 32 位元整數，小端對齊。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著value 可能過大，或者offset可能會超出buffer的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

39.buf.writeUInt32BE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式(Big-Endian:大位元組序)將 value 寫入buffer。注意：value 必須是一個合法的有符號 32 位元整數。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者offset可能會超出buffer的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

40.buf.writeInt8(value, offset[, noAssert])

41.buf.writeInt16LE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個合法的 signed 16 位元整數。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false 。

42.buf.writeInt16BE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個合法的 signed 16 位元整數。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false 。

43.buf.writeInt32LE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個合法的 signed 32 位元整數。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

44.buf.writeInt32BE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個合法的 signed 32 位元整數。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

45.buf.writeFloatLE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer 。注意：當 value 不是一個 32 位浮點數類型的值時，結果將是不確定的。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

46.buf.writeFloatBE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer 。注意：當 value 不是一個 32 位浮點數類型的值時，結果將是不確定的。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value可能過大，或者 offset 可能會超出 buffer 的末尾從而造成 value 被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

47.buf.writeDoubleLE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個有效的 64 位double 類型的值。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成value被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

48.buf.writeDoubleBE(value, offset[, noAssert])

根據傳入的 offset 偏移量和指定的 endian 格式將 value 寫入 buffer。注意：value 必須是一個有效的 64 位double 類型的值。 若參數 noAssert 為 true 將不會驗證 value 和 offset 偏移量參數。 這意味著 value 可能過大，或者 offset 可能會超出 buffer 的末尾從而造成value被丟棄。 除非你對這個參數非常有把握，否則儘量不要使用。默認是 false。

49.buf.fill(value[, offset][, end])

使用指定的 value 來填充這個 buffer。如果沒有指定 offset (默認是 0) 並且 end (默認是 buffer.length) ，將會填充整個buffer。

# CH 4 Node.js進階應用II

## CH 4-1 Node.js Stream(流)

Stream 是一個抽象介面，Node 中有很多物件實現了這個介面。

Stream 有四種流類型：

Readable - 可讀操作。

Writable - 可寫操作。

Duplex - 可讀可寫操作.

Transform - 操作被写入数据，然后读出结果。

常用的事件有：

data - 當有數據可讀時觸發。

end - 沒有更多的資料可讀時觸發。

error - 在接收和寫入過程中發生錯誤時觸發。

finish - 所有資料已被寫入到底層系統時觸發。

**Step 1：從流中讀取數據**

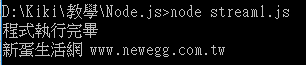
新增檔案input.txt

|  |
| --- |
| 新蛋生活網 www.newegg.com.tw |

新增檔案Stream1.js，說明如註解

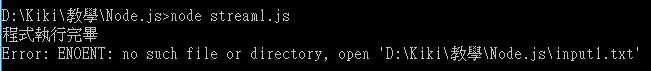
|  |
| --- |
| var fs = require("fs");  var data = '';  // 創建可讀流  var readerStream = fs.createReadStream('input.txt');  // 設置編碼為 utf8。  readerStream.setEncoding('UTF8');  // 處理流事件 --> data, end, and error  readerStream.on('data', function(chunk) {  data += chunk; //將讀取的資料寫入data  });  //當該事件處理完畢時執行此段  readerStream.on('end',function(){  console.log(data);  });  //出錯的時候執行此段代碼  readerStream.on('error', function(err){  console.log(err.stack);  });  console.log("程式執行完畢"); |

執行node Stream1.js



若將代碼中修改成一個不存在的txt檔，執行時會觸發error事件，顯示該錯誤訊息

|  |
| --- |
| var readerStream = fs.createReadStream('input1.txt'); |



**Step 2：寫入流**

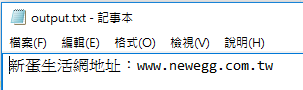
新增檔案Stream2.js

|  |
| --- |
| var fs = require("fs");  var data = '新蛋生活網地址：www.newegg.com.tw';  // 創建一個可以寫入的流，寫入到檔 output.txt 中  var writerStream = fs.createWriteStream('output.txt');  // 使用 utf8 編碼寫入資料  writerStream.write(data,'UTF8');  // 標記檔末尾  writerStream.end();  // 處理流事件 --> data, end, and error  writerStream.on('finish', function() {  console.log("寫入完成。");  });  writerStream.on('error', function(err){  console.log(err.stack);  });  console.log("程式執行完畢"); |

執行node Stream2.js



目錄中會多一個output.txt檔，開啟內容為 新蛋生活網地址：[www.newegg.com.tw](http://www.newegg.com.tw)'



**Step 3：管道流-文件複製**

新增檔案input.txt

|  |
| --- |
| 新蛋生活網 www.newegg.com.tw |

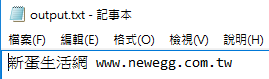
新增檔案Stream3.js

|  |
| --- |
| var fs = require("fs");  // 創建一個可讀流  var readerStream = fs.createReadStream('input.txt');  // 創建一個可寫流  var writerStream = fs.createWriteStream('output.txt');  // 管道讀寫操作  // 讀取 input.txt 檔內容，並將內容寫入到 output.txt 文件中  readerStream.pipe(writerStream);  console.log("程式執行完畢"); |

執行node Stream3.js



會將input.txt的內容寫入output.txt中。



**Step 4：鏈式流-文件壓縮**

文件壓處理，縮創建 compress.js

|  |
| --- |
| var fs = require("fs");  var zlib = require('zlib');  // 壓縮 input.txt 文件為 input.txt.gz  fs.createReadStream('input.txt')  .pipe(zlib.createGzip())  .pipe(fs.createWriteStream('input.txt.gz'));    console.log("檔案壓縮完成。"); |

執行node compress.js



目錄中多了一筆input.txt.gz壓縮檔



**Step 5：鏈式流-解壓縮**

文件解縮，創建decompress.js

|  |
| --- |
| var fs = require("fs");  var zlib = require('zlib');  // 解壓 input.txt.gz 文件為 input.txt  fs.createReadStream('input.txt.gz')  .pipe(zlib.createGunzip())  .pipe(fs.createWriteStream('input.txt'));    console.log("檔解壓完成。"); |

先移除目錄中的'input.txt檔，執行node compress.js，處理完後會將input.txt.gz解壓縮至input.txt檔



## CH 4-2 Node.js模組系統

通過 exports 設置模組的訪問介面，再利用require('./路徑') 載入這個模組

**Step 1：方法1**

新增檔案sum1.js

|  |
| --- |
| exports.sum = function(X,Y) {  console.log( X+' \* '+Y+ ' = '+ (X\*Y));  }  exports.add = function(X,Y) {  console.log( X+' + '+Y+ ' = '+ (X+Y));  } |

創建module1.js

|  |
| --- |
| var sum1= require('./sum1);  sum1.sum(4,5);  sum1.add(3,4); |

執行node module1.js



**Step 2：方法2**

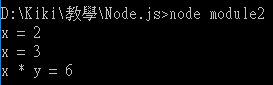
新增檔案sum2.js

|  |
| --- |
| function sum2() {  var num;  this.setNum = function(x,y) {  num = x \* y;  console.log('x = ' + x);  console.log('x = ' + y);  };  this.showNum = function() {  console.log('x \* y = ' + num);  };  };  // exports 設置模組的訪問介面，其介面物件就是要輸出的sum2本身，而不是原先的 exports  module.exports = sum2; |

創建module2.js

|  |
| --- |
| var sum2= require('./sum2); //引入sum2.js檔  sum2= new sum2 ();  sum2.setNum(2,3); //執行引入的方法  sum2.showNum(); |

執行node module2.js



## CH 4-3 Node.js 路由

查看HTTP請求，從中提取出請求的URL以及GET/POST參數

**Step 1：建立執行的項目**

創建router.js，此檔主要建立一個執行的項目，依傳入的不同路徑執行不同項目

|  |
| --- |
| function route(pathname) {  if(pathname=="/start")  { console.log(pathname+"功能執行~"); }  else if(pathname=="/get")  { console.log(pathname+"功能執行中~");}  else{console.log(pathname+"無此服務~");}  }  exports.route = route; |

**Step 2：建立Http服務**

創建routerServer.js，啟動http服務，並且傳遞資訊給route

|  |
| --- |
| var http = require("http");  var url = require("url");  function start(route) { //route是由外部傳入的function  function onRequest(request, response) {  var pathname = url.parse(request.url).pathname; //取出URL後面的名稱  if(pathname!="/favicon.ico") //排除/favicon.ico不明問題  {  console.log("Request 路徑 : " + pathname );  route(pathname);//執行router.js  response.writeHead(200, {"Content-Type": "text/plain"});  response.write("Server:"+pathname);//顯示於網頁  response.end();  }  }  http.createServer(onRequest).listen(8888);//啟動服務  console.log("Server has started.");  }  exports.start = start; |

**Step 3：注入服務及啟動**

創建routerIndex.js，routerServer和router兩者能能不會自動執行，需由此檔，來啟動服務，並且將路由注入服務中執行。

|  |
| --- |
| var server = require("./routerServer");  var router = require("./router");  server.start(router.route); |

**Step 4：執行服務**

執行node routerIndex.js

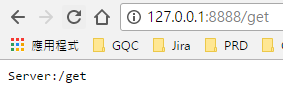


開啟網頁 <http://127.0.0.1:8888/start>





切換網頁 <http://127.0.0.1:8888/get>，會執行/get條件下的語法





切換網頁 <http://127.0.0.1:8888/abcd> ，找不到該條件事，顯示無此服務





## CH 4-4 Node.js GET/POST請求

服務器一般都使用 GET/POST 請求

**Step 1：GET請求**

GET請求直接被嵌入在路徑中，創建get01.js

|  |
| --- |
| var http = require('http');  var url = require('url');  var util = require('util');  http.createServer(function(req, res){  res.writeHead(200, {'Content-Type': 'text/plain; charset=utf-8'});    var params = url.parse(req.url, true).query; //取出query中的清單  res.write("網站名：" + params.name);  res.write("\n");  res.write("網站 URL：" + params.url);  res.write("\n");  //util.inspect是一個將任意物件轉換 為字串的方法 url.parse 方法來解析 URL 中的參數  res.end(util.inspect(url.parse(req.url, true)));  }).listen(3000); |

執行node get01.js，並開啟網頁

http://localhost:3000/user?name=新蛋生活網&url=www.newegg.com.tw



Node.js游標會停留在畫面上，網頁會抓取name、url顯示



**Step 2：POST請求**

創建pos01.js

|  |
| --- |
| var http = require('http');  var querystring = require('querystring');  var postHTML =  '<html><head><meta charset="utf-8"><title>練習</title></head>' +  '<body>' +  '<form method="post">' +  '網站名： <input name="name"><br>' +  '網站 URL： <input name="url"><br>' +  '<input type="submit">' +  '</form>' +  '</body></html>';  http.createServer(function (req, res) {  var body = "";// 定義了一個body變數，用於暫存請求體的資訊  req.on('data', function (chunk) {// 通過req的data事件監聽函數，每當接受到請求體的資料，就累加到body變數中  body += chunk;  });  req.on('end', function () {// 在end事件觸發後，通過querystring.parse將body解析為真正的POST請求格式，然後向用戶端返回。  // 解析參數  body = querystring.parse(body);  // 設置回應頭部資訊及編碼  res.writeHead(200, {'Content-Type': 'text/html; charset=utf8'});  if(body.name && body.url) { // 輸出提交的資料  res.write("網站名：" + body.name);  res.write("<br>");  res.write("網站 URL：" + body.url);  } else { // 輸出表單  res.write(postHTML);  }  res.end();  });  }).listen(3000); |

執行後開啟網頁<http://localhost:3000/>，初始值會顯示輸入框，填寫提交後顯示取得的值

## CH 4-5 Node.js全域物件

其所有屬性都可以在程式的任何地方訪問，即全域變數。

JavaScript 中，通常 window 是全域對象， 而 Node.js 中的全域物件是 globa

**Step 1：範例-取出完整路徑**

\_\_filename範例，新增檔案filename.js

|  |
| --- |
| console.log( \_\_filename ); |

輸出結果



**Step 2：範例-取出所在目錄**

\_\_dirname可取出當前所在目錄，新增檔案dirname.js

|  |
| --- |
| console.log( \_\_dirname ); |

輸出結果



**Step 3：範例-間隔時間後執行**

setTimeout(cb, ms) 全域函數在指定的時間 (毫秒ms)後執行指定函數(cb)

|  |
| --- |
| function printHello(){  console.log( "Hello, World!");  }  // 兩秒後執行以上函數  setTimeout(printHello, 2000); |

執行後間隔一段時間輸出



**Step 4：其他範例**

clearTimeout( t )清除定時器

|  |
| --- |
| var t = setTimeout(function (){ console.log( "Hello, World!");}, 2000);  clearTimeout(t);//函數還沒執行已被停止執行 |

setInterval(cb, ms)間隔一定時間(毫秒ms)後，執行指定函數(cb)，且會不停地被調用，直到被關閉(按下 **ctrl + c** 按鈕)或執行clearTimeout( t )

|  |
| --- |
| setInterval (function (){ console.log( "Hello, World!");}, 2000); |

**Step 5：console**

console它是由 Internet Explorer 的 JScript 引擎提供的調試工具，Node.js 沿用了這個標準，提供與習慣行為一致的 console 物件

方法:

console.log([data][, ...]) 以類似於C 語言 printf() 命令的格式輸出

console.info([data][, ...])返回資訊，與console.log差別並不大，chrome中會顯示藍色的驚嘆號

console.error([data][, ...]) 輸出錯誤消息的

console.warn([data][, ...]) 輸出警告消息。出現有黃色的驚嘆號。

console.dir(obj[, options]) 用來對一個物件進行檢查

console.time(label) 輸出時間，表示計時開始。

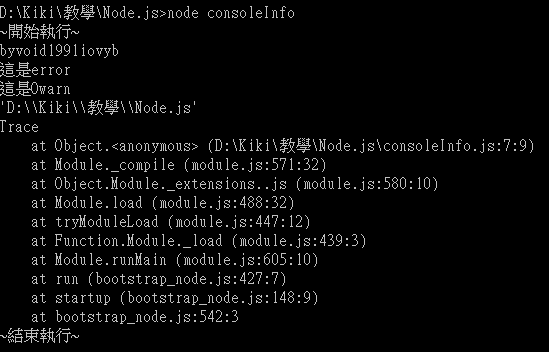
console.timeEnd(label) 結束時間，表示計時結束。

console.trace(message[, ...]) 執行的代碼在堆疊中調用路徑

console.assert(value[, message][, ...]) 用於判斷某個運算式或變數是否為真，第一個參數是運算式，第二個參數是字串

console範例:

|  |
| --- |
| console.info("~開始執行~");  console.log('byvoid%diovyb', 1991);  console.error("這是error");  console.warn("這是Owarn");  console.dir(\_\_dirname);//檢查\_\_dirname  console.trace();  console.info("~結束執行~") |



**Step 6：process**

它用於描述當前Node.js 進程狀態的物件，提供了一個與作業系統的簡單介面

方法:

exit 當進程準備退出時觸發。

beforeExit清空事件迴圈時觸發。

uncaughtException異常事件迴圈時觸發。

Signal當進程接收到信號時就觸發

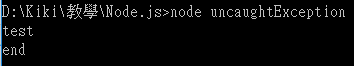
使用方式:

|  |
| --- |
| process.on('方法', function() {….}); |

範例:

|  |
| --- |
| var test=function() { console.log("end");}  process.on('exit', test);//當執行結束時被觸發  console.log("test"); |

執行結果



**參考**

其他物件參考如下

退出狀態碼

1.Uncaught Fatal Exception

有未捕獲異常，並且沒有被域或 uncaughtException 處理函數處理。

2.Unused

保留

3.Internal JavaScript Parse Error

JavaScript的源碼啟動 Node 進程時引起解析錯誤。非常罕見，僅會在開發 Node 時才會有。

4.Internal JavaScript Evaluation Failure

JavaScript 的源碼啟動 Node 進程，評估時返回函數失敗。非常罕見，僅會在開發 Node 時才會有。

5.Fatal Error

V8 裡致命的不可恢復的錯誤。通常會列印到 stderr ，內容為： FATAL ERROR

6.Non-function Internal Exception Handler

未捕獲異常，內部異常處理函數不知為何設置為on-function，並且不能被調用。

7.Internal Exception Handler Run-Time Failure

未捕獲的異常， 並且異常處理函數處理時自己拋出了異常。例如，如果 process.on('uncaughtException') 或 domain.on('error') 拋出了異常。

8.Unused

保留

9.Invalid Argument

可能是給了未知的參數，或者給的參數沒有值。

10.Internal JavaScript Run-Time Failure

JavaScript的源碼啟動 Node 進程時拋出錯誤，非常罕見，僅會在開發 Node 時才會有。

12.Invalid Debug Argument

設置了參數--debug 和/或 --debug-brk，但是選擇了錯誤埠。

128 Signal Exits

如果 Node 接收到致命信號，比如SIGKILL 或 SIGHUP，那麼退出代碼就是128 加信號代碼。這是標準的 Unix 做法，退出信號代碼放在高位。

Process 属性

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。

9version

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。

方法參考手冊

1.abort()

這將導致 node 觸發 abort 事件。會讓 node 退出並生成一個核心檔。

2.chdir(directory)

改變當前工作進程的目錄，如果操作失敗拋出異常。

3.cwd()

返回當前進程的工作目錄

4.exit([code])

使用指定的 code 結束進程。如果忽略，將會使用 code 0。

5.getgid()

獲取進程的群組標識（參見 getgid(2)）。獲取到得時群組的數字 id，而不是名字。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

6.setgid(id)

設置進程的群組標識（參見 setgid(2)）。可以接收數位 ID 或者群組名。如果指定了群組名，會阻塞等待解析為數字 ID 。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

7.getuid()

獲取進程的使用者標識(參見 getuid(2))。這是數字的使用者 id，不是用戶名。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

8.setuid(id)

設置進程的使用者標識（參見setuid(2)）。接收數位 ID或字串名字。果指定了群組名，會阻塞等待解析為數字 ID 。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

9.getgroups()

返回進程的群組 iD 陣列。POSIX 系統沒有保證一定有，但是 node.js 保證有。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

10.setgroups(groups)

設置進程的群組 ID。這是授權操作，所有你需要有 root 許可權，或者有 CAP\_SETGID 能力。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

11.initgroups(user, extra\_group)

讀取 /etc/group ，並初始化群組訪問列表，使用成員所在的所有群組。這是授權操作，所有你需要有 root 許可權，或者有 CAP\_SETGID 能力。

注意：這個函數僅在 POSIX 平臺上可用(例如，非Windows 和 Android)。

12.kill(pid[, signal])

發送信號給進程. pid 是進程id，並且 signal 是發送的信號的字串描述。信號名是字串，比如 'SIGINT' 或 'SIGHUP'。如果忽略，信號會是 'SIGTERM'。

13.memoryUsage()

返回一個物件，描述了 Node 進程所用的記憶體狀況，單位為位元組。

14.nextTick(callback)

一旦當前事件迴圈結束，調用回到函數。

15.umask([mask])

設置或讀取進程檔的遮罩。子進程從父進程繼承遮罩。如果mask 參數有效，返回舊的遮罩。否則，返回當前遮罩。

16.uptime()

返回 Node 已經運行的秒數。

17.hrtime()

返回當前進程的高分辨時間，形式為 [seconds, nanoseconds]陣列。它是相對於過去的任意事件。該值與日期無關，因此不受時鐘漂移的影響。主要用途是可以通過精確的時間間隔，來衡量程式的性能。

你可以將之前的結果傳遞給當前的 process.hrtime() ，會返回兩者間的時間差，用來基準和測量時間間隔。

# CH 5 Node.js進階應用III

## CH 5-1 Node.js文件系统

**Step 1：非同步和同步**

參考3-1非同步的 fs.readFile() 和同步的 fs.readFileSync()。

建議是用非同步方法，比起同步，非同步方法性能更高，速度更快，而且沒有阻塞。

**Step 2：打開文件**

非同步模式下打開檔

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

參數:

path - 檔的路徑。

flags - 文件打開的行為。具體值詳見下文。

r 以讀取模式打開檔。如果檔不存在拋出異常。

r+ 以讀寫模式打開檔。如果檔不存在拋出異常。

rs 以同步的方式讀取檔。

rs+ 以同步的方式讀取和寫入檔。

w 以寫入模式打開檔，如果檔不存在則創建。

wx 類似 'w'，但是如果檔路徑存在，則檔寫入失敗。

w+ 以讀寫模式打開檔，如果檔不存在則創建。

wx+ 類似 'w+'， 但是如果檔路徑存在，則檔讀寫失敗。

a 以追加模式打開檔，如果檔不存在則創建。

ax 類似 'a'， 但是如果檔路徑存在，則檔追加失敗。

a+ 以讀取追加模式打開檔，如果檔不存在則創建。

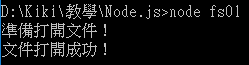
ax+ 類似 'a+'， 但是如果檔路徑存在，則檔讀取追加失敗。

mode - 設置檔模式(許可權)，檔創建默認許可權為 0666(可讀，可寫)。

callback - 回呼函數，帶有兩個參數如：callback(err, fd)。

範例，創建fs01.js

|  |
| --- |
| var fs = require("fs");  // 非同步打開檔  console.log("準備打開文件！");  fs.open('input.txt', 'r+', function(err, fd) {  if (err) {  return console.error(err);  }  console.log("文件打開成功！");  }); |



**Step 3：獲取檔資訊**

非同步模式獲取檔資訊

|  |
| --- |
| fs.stat(path, callback) |

參數

path - 檔路徑。

callback - 回呼函數，帶有兩個參數如：(err, stats), stats 是 fs.Stats 對象。

創建fs02.js

|  |
| --- |
| var fs = require('fs');  fs.stat('fs01.js', function (err, stats) {  if (err) {  return console.error(err);  }  console.log(stats);  console.log("是否為檔(isFile) ? " +stats.isFile()); //如果是文件返回true  }) |

stats.isFile() 如果是檔返回 true，否則返回 false。

stats.isDirectory() 如果是目錄返回 true，否則返回 false。

stats.isBlockDevice() 如果是塊設備返回 true，否則返回 false。

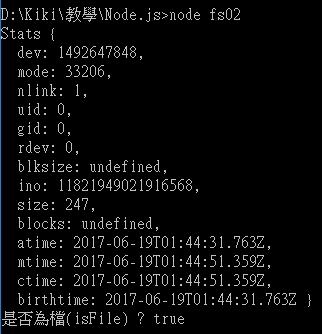
stats.isCharacterDevice() 如果是字元設備返回 true，否則返回 false。

stats.isSymbolicLink() 如果是軟連結返回 true，否則返回 false。

stats.isFIFO() 如果是FIFO，返回true，否則返回 false。FIFO是UNIX中的一種特殊類型的命令管道。

stats.isSocket() 如果是 Socket 返回 true，否則返回 false。

範例執行結果



**Step 4：寫入文件**

非同步模式下寫入檔

|  |
| --- |
| fs.writeFile(file, data[, options], callback) |

參數

file - 檔案名或檔描述符。

data - 要寫入檔的資料，可以是 String(字串) 或 Buffer(流) 對象。

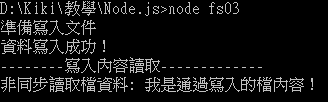
options - 該參數是一個物件，包含 {encoding, mode, flag}。預設編碼為 utf8, 模式為 0666 ， flag 為 'w'

callback - 回呼函數，回呼函數隻包含錯誤資訊參數(err)，在寫入失敗時返回。

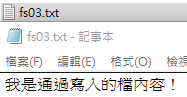
創建fs03.js

|  |
| --- |
| var fs = require("fs");  console.log("準備寫入文件");  fs.writeFile('fs03.txt', '我是通過寫入的檔內容！', function(err) {  if (err) {  return console.error(err);  }  console.log("資料寫入成功！");  console.log("--------寫入內容讀取-------------")  fs.readFile('fs03.txt', function (err, data) {  if (err) {  return console.error(err);  }  console.log("非同步讀取檔資料: " + data.toString());  });  }); |

執行結果



目錄中找到該檔案



**Step 5：讀取文件**

非同步模式讀取檔

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

參數

fd - 通過 fs.open() 方法返回的檔描述符。

buffer - 資料寫入的緩衝區。

offset - 緩衝區寫入的寫入偏移量。

length - 要從檔中讀取的位元組數。

position - 檔讀取的起始位置，如果 position 的值為 null，則會從當前文件指針的位置讀取。

callback - 回呼函數，有三個參數err, bytesRead, buffer，err 為錯誤資訊， bytesRead 表示讀取的位元組數，buffer 為緩衝區物件。

**Step 6：關閉文件**

非同步模式下關閉檔

|  |
| --- |
| fs.close(fd, callback) |

參數

fd - 通過 fs.open() 方法返回的檔描述符。

callback - 回呼函數，沒有參數。

**Step 7：截取文件**

异步模式下截取文件

|  |
| --- |
| fs.ftruncate(fd, len, callback) |

參數

fd - 通過 fs.open() 方法返回的檔描述符。

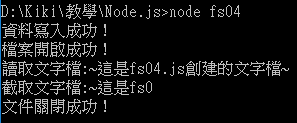
len - 檔內容截取的長度。

callback - 回呼函數，沒有參數。

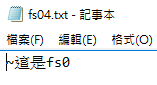
創建範例fs04.js

|  |
| --- |
| var fs = require("fs");  var buf = new Buffer(1024);//緩衝  //-----------------寫入文字檔-------------------  fs.writeFile('fs04.txt', '~這是fs04.js創建的文字檔~', function(err) {  if (err) {  return console.error(err);  }  console.log("資料寫入成功！");  //-----------------開啟文字檔-------------------  fs.open('fs04.txt', 'r+', function(err, fd) { //檔案開啟  if (err) {  return console.error(err);  }  console.log("檔案開啟成功！");  //-----------------讀取文字檔-------------------  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){ //bytes:字節數量  if (err){  console.log(err);  }  // 輸出的文字  if(bytes > 0){  console.log("讀取文字檔:"+buf.slice(0, bytes).toString());//輸出緩衝  }  });    //-----------------截取文字檔-------------------  fs.ftruncate(fd, 10, function(err){ //截取長度  if (err){  console.log(err);  }  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){  if (err){  console.log(err);  }  if(bytes > 0){  console.log("截取文字檔:"+buf.slice(0, bytes).toString());  }    //-----------------關閉文件-------------------  fs.close(fd, function(err){  if (err){  console.log(err);  }  console.log("文件關閉成功！");  });    });  });    });    }); |

執行結果



開啟時是以r+讀寫模式，檔案最後會寫入最終結果



**Step 8：删除文件**

刪除檔的語法

|  |
| --- |
| fs.unlink(path, callback) |

參數

path - 檔路徑。

callback - 回呼函數，沒有參數。

創建fs05.js

|  |
| --- |
| var fs = require("fs");  fs.unlink('fs04.txt', function(err) {  if (err) {  return console.error(err);  }  console.log("檔案刪除成功!");  }); |

執行後，該檔案已移除目錄，並顯示刪除成功訊息



**Step 9：創建目錄**

建目錄的語法

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

參數

path - 檔路徑。

mode - 設置目錄許可權，預設為 0777。

callback - 回呼函數，沒有參數。

**Step 10：讀取目錄**

讀取目錄的語法

|  |
| --- |
| fs.readdir(path, callback) |

參數

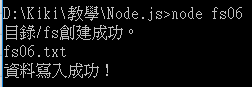
path - 檔路徑。

callback - 回呼函數，回呼函數帶有兩個參數err, files，err 為錯誤資訊，files 為 目錄下的檔陣列清單。

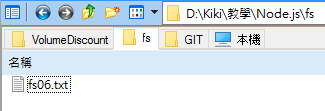
創建fs06.js範例

|  |
| --- |
| var fs = require("fs");  //--------創建目錄---------------  fs.mkdir("./fs/",function(err){  if (err) {  return console.error(err);  }  console.log("目錄/fs創建成功。");  });  //-----------------寫入文字檔-------------------  fs.writeFile('./fs/fs06.txt', '~這是fs06.js創建的文字檔~', function(err) {  if (err) {  return console.error(err);  }  console.log("資料寫入成功！");  });  //--------讀取目錄---------------  fs.readdir("./fs/",function(err, files){  if (err) {  return console.error(err);  }  files.forEach( function (file){  console.log( file );  });  }); |

執行後可找到一個新的目錄，及目錄中多一個檔案，處理完顯示目錄中有那些檔案。







**Step 11：刪除目錄**

|  |
| --- |
| fs.rmdir(path, callback) |

參數

path - 檔路徑。

callback - 回呼函數，沒有參數。

延申上一個範例，刪除目錄中檔案及目錄，創建sf07.js

|  |
| --- |
| var fs = require("fs");  //--------讀取目錄---------------  fs.readdir("./fs/",function(err, files){  if (err) {  return console.error(err);  }  //--------刪除檔案---------------  files.forEach( function (file){  fs.unlink("./fs/"+file, function(err) {  if (err) {  return console.error(err);  }  console.log("檔案刪除成功!");  });  });    // ------刪除目錄------  fs.rmdir("./fs/",function(err){  if (err) {  return console.error(err);  }  console.log("目錄刪除成功!");  });    }); |

執行結果



**參考**

1.fs.rename(oldPath, newPath, callback)

非同步 rename().回呼函數沒有參數，但可能拋出異常。

2.fs.ftruncate(fd, len, callback)

非同步 ftruncate().回呼函數沒有參數，但可能拋出異常。

3.fs.ftruncateSync(fd, len)

同步 ftruncate()

4.fs.truncate(path, len, callback)

非同步 truncate().回呼函數沒有參數，但可能拋出異常。

5.fs.truncateSync(path, len)

同步 truncate()

6.fs.chown(path, uid, gid, callback)

非同步 chown().回呼函數沒有參數，但可能拋出異常。

7.fs.chownSync(path, uid, gid)

同步 chown()

8.fs.fchown(fd, uid, gid, callback)

非同步 fchown().回呼函數沒有參數，但可能拋出異常。

9.fs.fchownSync(fd, uid, gid)

同步 fchown()

10.fs.lchown(path, uid, gid, callback)

非同步 lchown().回呼函數沒有參數，但可能拋出異常。

11.fs.lchownSync(path, uid, gid)

同步 lchown()

12.fs.chmod(path, mode, callback)

非同步 chmod().回呼函數沒有參數，但可能拋出異常。

13.fs.chmodSync(path, mode)

同步 chmod().

14.fs.fchmod(fd, mode, callback)

非同步 fchmod().回呼函數沒有參數，但可能拋出異常。

15.fs.fchmodSync(fd, mode)

同步 fchmod().

16.fs.lchmod(path, mode, callback)

非同步 lchmod().回呼函數沒有參數，但可能拋出異常。Only available on Mac OS X.

17.fs.lchmodSync(path, mode)

同步 lchmod().

18.fs.stat(path, callback)

非同步 stat(). 回呼函數有兩個參數 err, stats，stats 是 fs.Stats 對象。

19.fs.lstat(path, callback)

非同步 lstat(). 回呼函數有兩個參數 err, stats，stats 是 fs.Stats 對象。

20.fs.fstat(fd, callback)

非同步 fstat(). 回呼函數有兩個參數 err, stats，stats 是 fs.Stats 對象。

21.fs.statSync(path)

同步 stat(). 返回 fs.Stats 的實例。

22.fs.lstatSync(path)

同步 lstat(). 返回 fs.Stats 的實例。

23.fs.fstatSync(fd)

同步 fstat(). 返回 fs.Stats 的實例。

24.fs.link(srcpath, dstpath, callback)

非同步 link().回呼函數沒有參數，但可能拋出異常。

25.fs.linkSync(srcpath, dstpath)

同步 link().

26.fs.symlink(srcpath, dstpath[, type], callback)

非同步 symlink().回呼函數沒有參數，但可能拋出異常。 type 參數可以設置為 'dir', 'file', 或 'junction' (默認為 'file') 。

27.fs.symlinkSync(srcpath, dstpath[, type])

同步 symlink().

28.fs.readlink(path, callback)

非同步 readlink(). 回呼函數有兩個參數 err, linkString。

29.fs.realpath(path[, cache], callback)

非同步 realpath(). 回呼函數有兩個參數 err, resolvedPath。

30.fs.realpathSync(path[, cache])

同步 realpath()。返回絕對路徑。

31.fs.unlink(path, callback)

非同步 unlink().回呼函數沒有參數，但可能拋出異常。

32.fs.unlinkSync(path)

同步 unlink().

33.fs.rmdir(path, callback)

非同步 rmdir().回呼函數沒有參數，但可能拋出異常。

34.fs.rmdirSync(path)

同步 rmdir().

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

S非同步 mkdir(2).回呼函數沒有參數，但可能拋出異常。 mode defaults to 0777.

36.fs.mkdirSync(path[, mode])

同步 mkdir().

37.fs.readdir(path, callback)

非同步 readdir(3). 讀取目錄的內容。

38.fs.readdirSync(path)

同步 readdir().返回檔陣列清單。

39.fs.close(fd, callback)

非同步 close().回呼函數沒有參數，但可能拋出異常。

40.fs.closeSync(fd)

同步 close().

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

非同步打開檔。

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)

修改檔時間戳記，檔通過指定的檔路徑。

45.fs.futimes(fd, atime, mtime, callback)

46.fs.futimesSync(fd, atime, mtime)

修改檔時間戳記，通過檔描述符指定。

47.fs.fsync(fd, callback)

非同步 fsync.回呼函數沒有參數，但可能拋出異常。

48.fs.fsyncSync(fd)

同步 fsync.

49.fs.write(fd, buffer, offset, length[, position], callback)

將緩衝區內容寫入到通過檔描述符指定的檔。

50.fs.write(fd, data[, position[, encoding]], callback)

通過檔描述符 fd 寫入檔內容。

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)

通過檔描述符 fd 讀取檔內容。

54.fs.readSync(fd, buffer, offset, length, position)

同步版的 fs.read.

55.fs.readFile(filename[, options], callback)

非同步讀取檔內容。

56.fs.readFileSync(filename[, options])

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

非同步寫入檔內容。

58.fs.writeFileSync(filename, data[, options])

同步版的 fs.writeFile。

59.fs.appendFile(filename, data[, options], callback)

非同步追加檔內容。

60.fs.appendFileSync(filename, data[, options])

The 同步 version of fs.appendFile.

61.fs.watchFile(filename[, options], listener)

查看檔的修改。

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 可以是檔或目錄。返回 fs.FSWatcher 對象。

64.fs.exists(path, callback)

檢測給定的路徑是否存在。

65.fs.existsSync(path)

同步版的 fs.exists.

66.fs.access(path[, mode], callback)

測試指定路徑使用者許可權。

67.fs.accessSync(path[, mode])

同步版的 fs.access。

68.fs.createReadStream(path[, options])

返回ReadStream 對象。

69.fs.createWriteStream(path[, options])

返回 WriteStream 對象。

70.fs.symlink(srcpath, dstpath[, type], callback)

非同步 symlink().回呼函數沒有參數，但可能拋出異常。

## CH 5-2 Node.js常用工具util

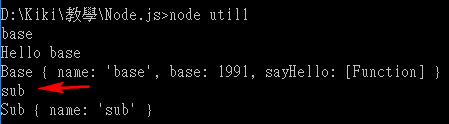
util 是一個Node.js 核心模組，提供常用函數的集合

**Step 1：util.inherits**

util.inherits是一個實現物件間原型繼承的函數

|  |
| --- |
| var util = require('util');  function Base() {  this.name = 'base';  this.base = 1991;  this.sayHello = function() {  console.log('Hello ' + this.name);  };  }  Base.prototype.showName = function() { //此為原型事件宣告  console.log(this.name);  };  function Sub() {  this.name = 'sub';  }  **util.inherits(Sub, Base);** //繼承原型事件  var objBase = new Base();  objBase.showName(); //本身的原型事件  objBase.sayHello(); //本身的非原型事件  console.log(objBase);  var objSub = new Sub();  objSub.showName(); //繼承Base的原型事件，非原型事件不可使用  console.log(objSub); |

執行結果，圖中sub是透過Base的原型事件執行的結果。

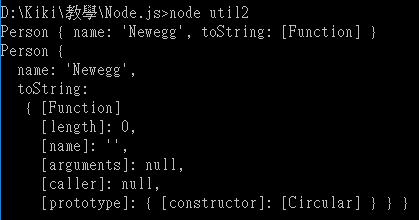


**Step 2：util.inspect**

util.inspect是一個將任意物件轉換 為字串的方法

|  |
| --- |
| var util = require('util');  function Person() {  this.name = 'Newegg';  this.toString = function() {  return this.name;  };  }  var obj = new Person();  console.log(util.inspect(obj));  console.log(util.inspect(obj, true)); |

執行結果

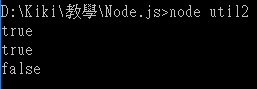


**Step 3：util.isArray(object)**

util.isArray(object) 是一個陣列返回true，否則返回false。

|  |
| --- |
| var util = require('util');  console.log(util.isArray([1,2,3,4,5,6]));  // true  console.log(util.isArray(new Array));  // true  console.log(util.isArray("1,2,3,4,5,6"));  // false |

執行結果



**Step 4：util.isRegExp(object)**

util.isRegExp(object)是一個規正表達式返回true，否則返回false。

**Step 5：util.isDate(object)**

util.isDate(object)是一個日期返回true，否則返回false。

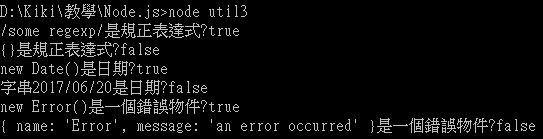
**Step 6：util.isError(object)**

util.isError(object)是一個錯誤物件返回true，否則返回false。

範例

|  |
| --- |
| var util = require('util');  console.log("/some regexp/是規正表達式?"+util.isRegExp(/some regexp/));  // true  console.log("{}是規正表達式?"+util.isRegExp({}));  // false    console.log("new Date()是日期?"+util.isDate(new Date()));  // true  console.log("字串2017/06/20是日期?"+util.isDate("2017/06/20"));  // false    console.log("new Error()是一個錯誤物件?"+util.isError(new Error()));  // true  console.log("{ name: 'Error', message: 'an error occurred' }是一個錯誤物件?"+util.isError({ name: 'Error', message: 'an error occurred' }));  // false |

執行結果



## CH 5-3 Node.js工具模組

**Step 1：OS 模組**

|  |
| --- |
| var os = require("os") |

方法

1.os.tmpdir() 返回作業系統的預設暫存檔案夾。

2.os.endianness() 返回 CPU 的位元組序，可能的是 "BE" 或 "LE"。

3.os.hostname() 返回作業系統的主機名稱。

4.os.type() 返回作業系統名

5.os.platform() 返回作業系統名

6.os.arch() 返回作業系統 CPU 架構，可能的值有 "x64"、"arm" 和 "ia32"。

7.os.release() 返回作業系統的發行版本本。

8.os.uptime() 返回作業系統運行的時間，以秒為單位。

9.os.loadavg() 返回一個包含 1、5、15 分鐘平均負載的陣列。

10.os.totalmem() 返回系統記憶體總量，單位為位元組。

11.os.freemem() 返回作業系統空閒記憶體量，單位是位元組。

12.os.cpus() 返回一個物件陣列，包含所安裝的每個 CPU/內核的資訊：型號、速度（單位 MHz）、時間（一個包含 user、nice、sys、idle 和 irq 所使用 CPU/內核毫秒數的物件）。

13.os.networkInterfaces() 獲得網路介面清單。

屬性

1.os.EOL 定義了作業系統的行尾符的常量。

範例，創建os1.js

|  |
| --- |
| var os = require("os");  console.log("目前系統版本為:"+os.release());  console.log("記憶體容量為:"+ os.totalmem() +"位元組");  console.log("系統運行時間為:"+os.uptime()); |

執行結果



**Step 2：Path模組**

用於處理檔路徑的小工具

|  |
| --- |
| var path = require("path") |

方法

1.path.normalize(p) 規範化路徑，注意'..' 和 '.'。

2.path.join([path1][, path2][, ...]) 用於連接路徑。該方法的主要用途在於，會正確使用當前系統的路徑分隔符號，Unix系統是"/"，Windows系統是"\"。

3.path.resolve([from ...], to) 將 to 參數解析為絕對路徑。

4.path.isAbsolute(path) 判斷參數 path 是否是絕對路徑。

5.path.relative(from, to) 用於將相對路徑轉為絕對路徑。

6.path.dirname(p) 返回路徑中代表資料夾的部分，同 Unix 的dirname 命令類似。

7.path.basename(p[, ext]) 返回路徑中的最後一部分。同 Unix 命令 bashname 類似。

8.path.extname(p) 返回路徑中檔的尾碼名，即路徑中最後一個'.'之後的部分。如果一個路徑中並不包含'.'或該路徑只包含一個'.' 且這個'.'為路徑的第一個字元，則此命令返回空字串。

9.path.parse(pathString) 返回路徑字串的物件。

10.path.format(pathObject) 從物件中返回路徑字串，和 path.parse 相反。

屬性

1.path.sep 平臺的檔路徑分隔符號，'\\' 或 '/'。

2.path.delimiter 平臺的分隔符號, ; or ':'.

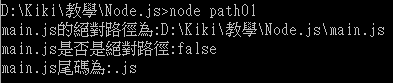
3.path.posix 提供上述 path 的方法，不過總是以 posix 相容的方式交互。

4.path.win32 提供上述 path 的方法，不過總是以 win32 相容的方式交互。

範例

|  |
| --- |
| var path = require("path");  console.log('main.js的絕對路徑為:'+path.resolve('main.js'));  console.log('main.js是否是絕對路徑:'+path.isAbsolute('main.js'));  console.log('main.js尾碼為:'+path.extname('main.js')); |

執行結果



**Step 3：Net模組**

用於底層的網路通信。提供了服務端和用戶端的的操作。

|  |
| --- |
| var net = require("net") |

方法

1.net.createServer([options][, connectionListener])

創建一個 TCP 伺服器。參數 connectionListener 自動給 'connection' 事件創建監聽器。

2.net.connect(options[, connectionListener])

返回一個新的 'net.Socket'，並連接到指定的位址和埠。當 socket 建立的時候，將會觸發 'connect' 事件。

3.net.createConnection(options[, connectionListener])

創建一個到埠 port 和 主機 host的 TCP 連接。 host 默認為 'localhost'。

4.net.connect(port[, host][, connectListener])

創建一個埠為 port 和主機為 host的 TCP 連接 。host 默認為 'localhost'。參數 connectListener 將會作為監聽器添加到 'connect' 事件。返回 'net.Socket'。

5.net.createConnection(port[, host][, connectListener])

創建一個埠為 port 和主機為 host的 TCP 連接 。host 默認為 'localhost'。參數 connectListener 將會作為監聽器添加到 'connect' 事件。返回 'net.Socket'。

6.net.connect(path[, connectListener])

創建連接到 path 的 unix socket 。參數 connectListener 將會作為監聽器添加到 'connect' 事件上。返回 'net.Socket'。

7.net.createConnection(path[, connectListener])

創建連接到 path 的 unix socket 。參數 connectListener 將會作為監聽器添加到 'connect' 事件。返回 'net.Socket'。

8.net.isIP(input)

檢測輸入的是否為 IP 位址。 IPV4 返回 4， IPV6 返回 6，其他情況返回 0。

9.net.isIPv4(input)

如果輸入的位址為 IPV4， 返回 true，否則返回 false。

10.net.isIPv6(input)

如果輸入的位址為 IPV6， 返回 true，否則返回 false。

net.Server

1.server.listen(port[, host][, backlog][, callback])

監聽指定埠 port 和 主機 host ac連接。 預設情況下 host 接受任何 IPv4 位址(INADDR\_ANY)的直接連接。埠 port 為 0 時，則會分配一個隨機埠。

2.server.listen(path[, callback]) 通過指定 path 的連接，啟動一個本地 socket 伺服器。

3.server.listen(handle[, callback]) 通過指定控制碼連接。

4.server.listen(options[, callback])

options 的屬性：埠 port, 主機 host, 和 backlog, 以及可選參數 callback 函數, 他們在一起調用server.listen(port, [host], [backlog], [callback])。還有，參數 path 可以用來指定 UNIX socket。

5.server.close([callback])

伺服器停止接收新的連接，保持現有連接。這是非同步函數，當所有連接結束的時候伺服器會關閉，並會觸發 'close' 事件。

6.server.address() 作業系統返回綁定的位址，協定族名和伺服器埠。

7.server.unref() 如果這是事件系統中唯一一個活動的伺服器，調用 unref 將允許程式退出。

8.server.ref()

與 unref 相反，如果這是唯一的伺服器，在之前被 unref 了的伺服器上調用 ref 將不會讓程式退出（預設行為）。如果伺服器已經被 ref，則再次調用 ref 並不會產生影響。

9.server.getConnections(callback)

非同步獲取伺服器當前活躍連接的數量。當 socket 發送給子進程後才有效；回呼函數有 2 個參數 err 和 count。

事件

1.listening 當伺服器調用 server.listen 綁定後會觸發。

2.connection 當新連接創建後會被觸發。socket 是 net.Socket實例。

3.close 伺服器關閉時會觸發。注意，如果存在連接，這個事件不會被觸發直到所有的連接關閉。

4.error 發生錯誤時觸發。'close' 事件將被下列事件直接調用。

net.Socket事件: net.Socket 實例實現了一個雙工流介面

1.lookup 在解析功能變數名稱後，但在連接前，觸發這個事件。對 UNIX sokcet 不適用。

2.connect 成功建立 socket 連接時觸發。

3.data 當接收到資料時觸發。

4.end 當 socket 另一端發送 FIN 包時，觸發該事件。

5.timeout 當 socket 空閒超時時觸發，僅是表明 socket 已經空閒。用戶必須手動關閉連接。

6.drain 當寫緩存為空得時候觸發。可用來控制上傳。

7.error 錯誤發生時觸發。

8.close 當 socket 完全關閉時觸發。參數 had\_error 是布林值，它表示是否因為傳輸錯誤導致 socket 關閉。

net.Socket屬性

1.socket.bufferSize 該屬性顯示了要寫入緩衝區的位元組數。

2.socket.remoteAddress 遠端的 IP 位址字串，例如：'74.125.127.100' or '2001:4860:a005::68'。

3.socket.remoteFamily 遠端IP協定族字串，比如 'IPv4' or 'IPv6'。

4.socket.remotePort 遠端埠，數字表示，例如：80 or 21。

5.socket.localAddress

網路連接綁定的本地介面 遠端客戶端正在連接的本地 IP 位址，字串表示。例如，如果你在監聽'0.0.0.0'而用戶端連接在'192.168.1.1'，這個值就會是 '192.168.1.1'。

6.socket.localPort 本地埠位址，數字表示。例如：80 or 21。

7.socket.bytesRead 接收到得位元組數。

8.socket.bytesWritten 發送的位元組數。

net.Socket方法

1.new net.Socket([options]) 構造一個新的 socket 物件。

2.socket.connect(port[, host][, connectListener])

指定埠 port 和 主機 host，創建 socket 連接 。參數 host 默認為 localhost。通常情況不需要使用 net.createConnection 打開 socket。只有你實現了自己的 socket 時才會用到。

3.socket.connect(path[, connectListener])

打開指定路徑的 unix socket。通常情況不需要使用 net.createConnection 打開 socket。只有你實現了自己的 socket 時才會用到。

4.socket.setEncoding([encoding]) 設置編碼

5.socket.write(data[, encoding][, callback]) 在 socket 上發送資料。第二個參數指定了字串的編碼，預設是 UTF8 編碼。

6.socket.end([data][, encoding]) 半關閉 socket。例如，它發送一個 FIN 包。可能伺服器仍在發送資料。

7.socket.destroy() 確保沒有 I/O 活動在這個通訊端上。只有在錯誤發生情況下才需要。（處理錯誤等等）。

8.socket.pause() 暫停讀取數據。就是說，不會再觸發 data 事件。對於控制上傳非常有用。

9.socket.resume() 調用 pause() 後想恢復讀取資料。

10.socket.setTimeout(timeout[, callback]) socket 閒置時間超過 timeout 毫秒後 ，將 socket 設置為超時。

11.socket.setNoDelay([noDelay])

禁用納格（Nagle）演算法。預設情況下 TCP 連接使用納格演算法，在發送前他們會緩衝資料。將 noDelay 設置為 true 將會在調用 socket.write() 時立即發送資料。noDelay 預設值為 true。

12.socket.setKeepAlive([enable][, initialDelay])

禁用/啟用長連接功能，並在發送第一個在閒置 socket 上的長連接 probe 之前，可選地設定初始延時。默認為 false。 設定 initialDelay （毫秒），來設定收到的最後一個資料包和第一個長連接probe之間的延時。將 initialDelay 設為0，將會保留默認（或者之前）的值。預設值為0.

13.socket.address()

作業系統返回綁定的位址，協定族名和伺服器埠。返回的物件有 3 個屬性，比如{ port: 12346, family: 'IPv4', address: '127.0.0.1' }。

14.socket.unref()

如果這是事件系統中唯一一個活動的伺服器，調用 unref 將允許程式退出。如果伺服器已被 unref，則再次調用 unref 並不會產生影響。

15.socket.ref()

與 unref 相反，如果這是唯一的伺服器，在之前被 unref 了的伺服器上調用 ref 將不會讓程式退出（預設行為）。如果伺服器已經被 ref，則再次調用 ref 並不會產生影響。

範例

|  |
| --- |
| var net = require('net');  var server = net.createServer(function(connection) {  console.log('client connected');  connection.on('end', function() {  console.log('用戶端關閉連接');  });  connection.write('Hello World!\r\n');  connection.pipe(connection);  });  server.listen(8080, function() {  console.log('server is listening');  }); |

執行結果，網頁開啟時顯示client connected，關閉網頁顯示用戶端關閉連接



**Step 4：DNS模組**

用於解析功能變數名稱。

|  |
| --- |
| var dns = require("dns") |

方法

1.dns.lookup(hostname[, options], callback)

將功能變數名稱（比如 'runoob.com'）解析為第一條找到的記錄 A （IPV4）或 AAAA(IPV6)。參數 options可以是一個物件或整數。如果沒有提供 options，IP v4 和 v6 位址都可以。如果 options 是整數，則必須是 4 或 6。

2.dns.lookupService(address, port, callback) 使用 getnameinfo 解析傳入的位址和埠為功能變數名稱和服務。

3.dns.resolve(hostname[, rrtype], callback)

將一個功能變數名稱（如 'runoob.com'）解析為一個 rrtype 指定記錄類型的陣列。

4.dns.resolve4(hostname, callback)

和 dns.resolve() 類似, 僅能查詢 IPv4 (A 記錄）。 addresses IPv4 位址陣列 (比如，['74.125.79.104', '74.125.79.105', '74.125.79.106']）。

5.dns.resolve6(hostname, callback) 和 dns.resolve4() 類似， 僅能查詢 IPv6( AAAA 查詢）

6.dns.resolveMx(hostname, callback) 和 dns.resolve() 類似, 僅能查詢郵件交換(MX 記錄)。

7.dns.resolveTxt(hostname, callback)

和 dns.resolve() 類似, 僅能進行文字查詢 (TXT 記錄）。 addresses 是 2-d 文本記錄陣列。(比如，[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]）。 每個子陣列包含一條記錄的 TXT 塊。根據使用情況可以連接在一起，也可單獨使用。

8.dns.resolveSrv(hostname, callback)

和 dns.resolve() 類似, 僅能進行服務記錄查詢 (SRV 記錄）。 addresses 是 hostname可用的 SRV 記錄陣列。 SRV 記錄屬性有優先順序（priority），權重（weight）, 埠（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() 類似, 僅能查詢權威記錄(SOA 記錄）。

10.dns.resolveNs(hostname, callback)

和 dns.resolve() 類似, 僅能進行功能變數名稱伺服器記錄查詢(NS 記錄）。 addresses 是功能變數名稱伺服器記錄陣列（hostname 可以使用） (比如, ['ns1.example.com', 'ns2.example.com']）。

11.dns.resolveCname(hostname, callback)

和 dns.resolve() 類似, 僅能進行別名記錄查詢 (CNAME記錄)。addresses 是對 hostname 可用的別名記錄陣列 (比如，, ['bar.example.com']）。

12.dns.reverse(ip, callback) 反向解析 IP 地址，指向該 IP 位址的功能變數名稱陣列。

13.dns.getServers() 返回一個用於當前解析的 IP 位址陣列的字串。

14.dns.setServers(servers) 指定一組 IP 地址作為解析伺服器。

範例

|  |
| --- |
| var dns = require('dns');  dns.lookup('www.newegg.com', function onLookup(err, address, family) {  console.log('ip 地址:', address);  }); |

執行結果



**Step 5：Domain模組**

簡化非同步代碼的異常處理，當發生一個錯誤事件或拋出一個錯誤時，domain物件會被通知

|  |
| --- |
| var domain = require("domain") |

顯式綁定: 把不是在domain上下文中定義的變數，以代碼的方式綁定到domain物件

隱式綁定: 把在domain上下文中定義的變數，自動綁定到domain物件

方法

1.domain.run(function)

在域的上下文運行提供的函數，隱式的綁定了所有的事件分發器，計時器和底層請求。

2.domain.add(emitter) 顯式的增加事件

3.domain.remove(emitter) 刪除事件。

4.domain.bind(callback)

返回的函數是一個對於所提供的回呼函數的包裝函數。當調用這個返回的函數被時，所有被拋出的錯誤都會被導向到這個域的 error 事件。

5.domain.intercept(callback)

和 domain.bind(callback) 類似。除了捕捉被拋出的錯誤外，它還會攔截 Error 對象作為參數傳遞到這個函數。

6.domain.enter() 進入一個非同步調用的上下文，綁定到domain。

7.domain.exit()

退出當前的domain，切換到不同的鏈的非同步調用的上下文中。對應domain.enter()。

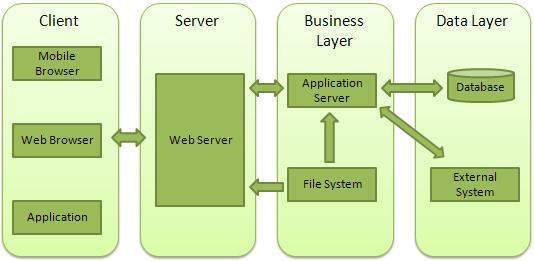
8.domain.dispose() 釋放一個domain物件，讓node進程回收這部分資源。

9.domain.create() 返回一個domain物件。

屬性

1.domain.members 已加入domain對象的域計時器和事件發射器的陣列。

## CH 5-4 Node.js Web模組



Client - 用戶端，一般指流覽器，流覽器可以通過 HTTP 協定向伺服器請求資料。

Server - 服務端，一般指 Web 伺服器，可以接收用戶端請求，並向用戶端發送回應資料。

Business - 業務層， 通過 Web 伺服器處理應用程式，如與資料庫交互，邏輯運算，調用外部程式等。

Data - 資料層，一般由資料庫組成。

http 模組主要用於搭建 HTTP 服務端和用戶端

|  |
| --- |
| var http = require('http'); |

以下範例主要利用http和url服務，並開啟相對應的html及伺服器

創建web01.js

|  |
| --- |
| var http = require('http');  var fs = require('fs');  var url = require('url');  // 創建伺服器  http.createServer( function (request, response) {  // 解析請求，包括檔案名  var pathname = url.parse(request.url).pathname;  if (pathname=='/'){ pathname="/index.html"; } //預設網頁  // 輸出請求的檔案名  console.log("Request for " + pathname + " received.");    // 從檔案系統中讀取請求的檔內容  fs.readFile(pathname.substr(1), function (err, data) {  if (err) {  console.log(err);  response.writeHead(404, {'Content-Type': 'text/html'}); //404網頁連線失敗，取得text/html格式  response.write('<html><head><title>404</title><meta http-equiv="Content-Type" content="text/html; charset=utf-8"></head><body>404網頁不存在!</body></html>');  }else{  response.writeHead(200, {'Content-Type': 'text/html'}); //200連線成功  // 回應檔內容  response.write(data.toString());  }  // 發送回應資料  response.end();  });  }).listen(8888);  // 控制台會輸出以下資訊  console.log('Server running at http://127.0.0.1:8888/'); |

創建檔案page01.html

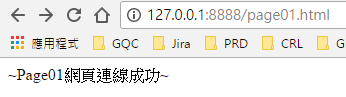
|  |
| --- |
| <html>  <head><title>Sample Page</title></head>  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  <body>  ~Page01網頁連線成功~  </body>  </html> |

創建檔案index.html

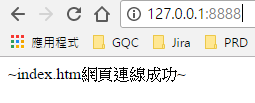
|  |
| --- |
| <html>  <head><title>Sample Page</title></head>  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  <body>  ~index.html網頁連線成功~  </body>  </html> |

開啟網頁<http://127.0.0.1:8888/page01.html>

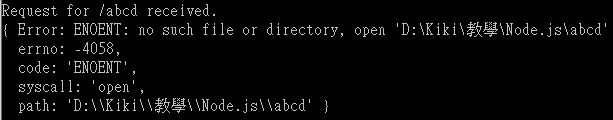




開啟網頁http://127.0.0.1:8888



開啟網頁http://127.0.0.1:8888/abcd





## CH 5-5 Node.js Express 框架

用 Express 可以快速地搭建一個完整功能的網站

**Step 1：安裝 Express**

|  |
| --- |
| npm install express --save |

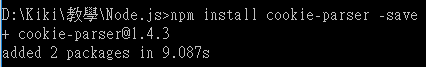




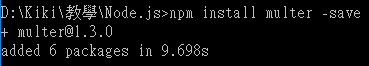
npm install body-parser –save 用於處理 JSON, Raw, Text 和 URL 編碼的資料



npm install cookie-parser –save 這就是一個解析Cookie的工具



npm install multer –save 中介軟體，用於處理 enctype="multipart/form-data"的表單數據。

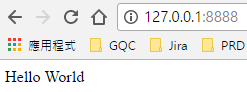


**Step 2： Express框架範例**

創建express01.js

|  |
| --- |
| var express = require('express');  var app = express();    app.get('/', function (req, res) {  res.send('Hello World');  })    var server = app.listen(8888, function () {    var host = server.address().address  var port = server.address().port    console.log("應用實例，訪問位址為 http://localhost:", host, port)    }) |

開啟網頁http://localhost:8888/



**Step 3： Express** **request 和 response**

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

Request 對象

req.app：當callback為外部文件時，用req.app訪問express的實例

req.baseUrl：獲取路由當前安裝的URL路徑

req.body / req.cookies：獲得「請求主體」/ Cookies

req.fresh / req.stale：判斷請求是否還「新鮮」

req.hostname / req.ip：獲取主機名稱和IP地址

req.originalUrl：獲取原始請求URL

req.params：獲取路由的parameters

req.path：獲取請求路徑

req.protocol：獲取協議類型

req.query：獲取URL的查詢參數串

req.route：獲取當前匹配的路由

req.subdomains：獲取子功能變數名稱

req.accepts()：檢查可接受的請求的文檔類型

req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages：返回指定字元集的第一個可接受字元編碼

req.get()：獲取指定的HTTP請求頭

req.is()：判斷請求頭Content-Type的MIME類型

Response 對象

res.app：同req.app一樣

res.append()：追加指定HTTP頭

res.set()在res.append()後將重置之前設置的頭

res.cookie(name，value [，option])：設置Cookie

opition: domain / expires / httpOnly / maxAge / path / secure / signed

res.clearCookie()：清除Cookie

res.download()：傳送指定路徑的檔

res.get()：返回指定的HTTP頭

res.json()：傳送JSON回應

res.jsonp()：傳送JSONP回應

res.location()：只設置回應的Location HTTP頭，不設置狀態碼或者close response

res.redirect()：設置回應的Location HTTP頭，並且設置狀態碼302

res.send()：傳送HTTP回應

res.sendFile(path [，options] [，fn])：傳送指定路徑的檔 -會自動根據檔extension設定Content-Type

res.set()：設置HTTP頭，傳入object可以一次設置多個頭

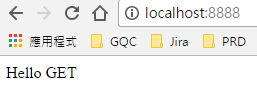
res.status()：設置HTTP狀態碼

res.type()：設置Content-Type的MIME類型

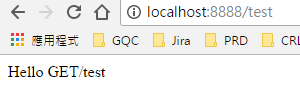
創建express02.js

|  |
| --- |
| var express = require('express');  var app = express();    // 主頁輸出 "Hello World"  app.get('/', function (req, res) {  console.log("主頁 GET 請求");  res.send('Hello GET');  })      // POST 請求  app.post('/', function (req, res) {  console.log("主頁 POST 請求");  res.send('Hello POST');  })  // 主頁輸出 "Hello GET/test"  app.get('/test', function (req, res) {  console.log("主頁 GET/test' 請求");  res.send('Hello GET/test');  })  var server = app.listen(8888, function () {    var host = server.address().address  var port = server.address().port    console.log("應用實例，訪問位址為 http://%s:%s", host, port)    }) |

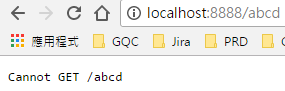
開啟網頁http://localhost:8888/



開啟網頁<http://localhost:8888/test>



開啟網頁<http://localhost:8888/abcd>



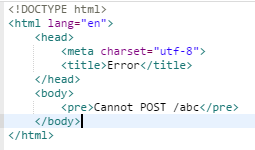
使用工具(Post Man)開啟http://localhost:8888/





使用工具開啟http://localhost:8888/abc





## CH 5-6 Node.js RESTful API

REST通常基於使用HTTP，URI，和XML以及HTML這些現有的廣泛流行的協議和標準。REST 通常使用 JSON 資料格式。滿足這些約束條件和原則的應用程式或設計就是RESTful

HTTP 方法

GET - 用於獲取資料。

PUT - 用於更新或添加資料。

DELETE - 用於刪除資料。

POST - 用於添加資料。

**Step 1： RESTful Web Services**

創建RESTful01.json

|  |
| --- |
| {  "user1" : {  "name" : "mahesh",  "password" : "password1",  "profession" : "teacher",  "id": 1  },  "user2" : {  "name" : "suresh",  "password" : "password2",  "profession" : "librarian",  "id": 2  },  "user3" : {  "name" : "ramesh",  "password" : "password3",  "profession" : "clerk",  "id": 3  }  } |

創建RESTful01.js以下範例處理顯全部、查詢、新增、刪除jason的內容

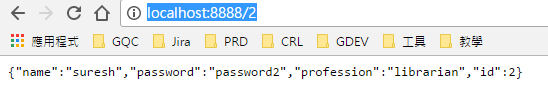
|  |
| --- |
| var express = require('express');  var app = express();  var fs = require("fs");  //--------------Show all-----------------------  app.get('/listUsers', function (req, res) {  fs.readFile( \_\_dirname + "/" + "RESTful01.json", 'utf8', function (err, data) {  console.log( data );  res.end( data );  });  })  //--------------Search ID-----------------------  app.get('/:id', function (req, res) {  // 首先我们读取已存在的用户  fs.readFile( \_\_dirname + "/" + "RESTful01.json", 'utf8', function (err, data) {  data = JSON.parse( data );  var user = data["user" + req.params.id]  console.log( user );  res.end( JSON.stringify(user));  });  })  //--------------addUser-----------------------  app.get('/addUser/:id', function (req, res) {  var user = {  "user4" : {  "name" : "mohit",  "password" : "password4" ,  "profession" : "teacher",  "id": 4  }  }  // 读取已存在的数据  fs.readFile( \_\_dirname + "/" + "RESTful01.json", 'utf8', function (err, data) {  data = JSON.parse( data );  data["user4"] = user["user4"];  console.log( data );  res.end( JSON.stringify(data));  });  })  //--------------Del User-----------------------  app.get('/deleteUser/:id', function (req, res) {  // First read existing users.  fs.readFile( \_\_dirname + "/" + "RESTful01.json", 'utf8', function (err, data) {  data = JSON.parse( data );  delete data["user" + req.params.id];    console.log( data );  res.end( JSON.stringify(data));  });  })  var server = app.listen(8888, function () {  var host = server.address().address  var port = server.address().port  console.log("URL: http://%s:%s", host, port)  }) |

執行結果

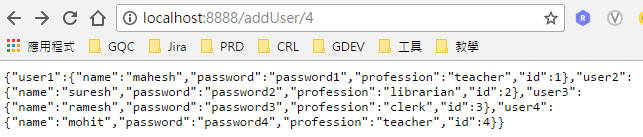
開啟http://localhost:8888/listUsers



開啟http://localhost:8888/2 會將id=2的值查詢出來



開啟http://localhost:8888/addUser/4 此範例會寫入一筆新的值 ( 範例寫入的值是固定的 )



開啟http://localhost:8888/deleteUser/1 此範例會刪除編號內的值再顯示

