

Node.js Taiwan 社群強力推薦

實戰
程式設計



The Node.js *Little Book*



中文版

適用最新版 Node.js 0.6.7

本書由 Node.js Taiwan 社群程式設計師編寫

最快樂的 Node.js 入門學習指南



ContPub Press

Node.js Taiwan



The Little Node.js Book

clonn, lyhcode

2012 年 03 月 11 日

目錄

Node.js Wiki Book	1
授權	1
關於作者	1
致謝	2
最新版本	2
 1 Node.js 簡介	 3
 2 JavaScript 與 NodeJS	 7
2.1 Event Loop	7
2.2 Scope 與 Closure	8
2.3 Callback	9
2.4 CPS (Continuation-Passing Style)	12
2.5 函數返回函數與 Currying	13
2.6 流程控制	14
 3 Node.js 安裝與設定	 21
3.1 Linux base	21

3.2	Windows	22
4	Node.js 基礎	25
4.1	node.js http 伺服器建立	25
4.2	node.js http 路徑建立	27
4.3	node.js 檔案讀取	30
4.4	node.js http 靜態檔案輸出	32
4.5	node.js http GET 資料擷取	34
4.6	本章結語	36
5	NPM 套件管理工具	37
5.1	安裝 NPM	37
5.2	使用 NPM 安裝套件	41
5.3	套件的更新及維護	44
5.4	使用 package.json	45
6	Express 介紹	47

Node.js Wiki Book

本書為 The Little Node.js Book 正體中文版。

授權

The Little Node.js Book 採用創用 CC 姓名標示 -非商業性授權。您不必為本書付費。

The Little Node.js Book book is licensed under the Attribution-NonCommercial 3.0 Unported license. You should not have paid for this book.

基本上您可以複製、散佈、修改或播放本書，但請務必尊重原作者的著作權，勿將本書用於商業用途。

您可以在以下網址取得授權條款全文。

<http://creativecommons.org/licenses/by-nc/3.0/legalcode>

關於作者

- Caesar Chi (clonn)
- Fillano Feng (fillano)
- Kyle Lin (lyhcode)

致謝

最新版本

本書最新的原始碼（中文版）網址如下：

<http://github.com/nodejs-tw/the-little-nodejs-book>

Node.js 簡介

Node.js 是一個高效能、易擴充的網站應用程式開發框架 (Web Application Framework)。它誕生的原因，是為了讓開發者能夠更容易開發高延展性的網路服務，不需要經過太多複雜的調校、效能調整及程式修改，就能滿足網路服務在不同發展階段對效能的要求。

Ryan Dahl 是 NodeJS 的催生者，目前任職於 Joyent (主機託管服務公司)。他開發 NodeJS 的目的，就是希望能解決 Apache 在連線數量過高時，緩衝區 (buffer) 和系統資源會很快被耗盡的問題，希望能建立一個新的開發框架以解決這個問題。因此嘗試使用效能十分優秀的 V8 JavaScript Engine，讓網站開發人員熟悉的 JavaScript 程式語言，也能應用於後端服務程式的開發，並且具有出色的執行效能。

JavaScript 是功能強大的物件導向程式語言，但是在 JavaScript 的官方規格中，主要是定義網頁 (以瀏覽器為基礎) 應用程式需要的應用程式介面 (API)，對 JavaScript 程式的應用範圍有所侷限。為使 JavaScript 能夠在更多用途發展，CommonJS 規範一組標準函式庫 (standard library)，使 JavaScript 的應用範圍能夠和 Ruby、Python 及 Java 等語言同樣豐富。撰寫 NodeJS 的 JavaScript 程式碼，符合 CommonJS 規範，可以使用 CommonJS API 為基礎開發程式，並且在不同的 CommonJS 兼容 (compliant) JavaScript 執行環境中，程式碼具有可攜性。

瀏覽器的 JavaScript 與實現 CommonJS 規範的 NodeJS 有何不同呢？瀏覽器的 JavaScript 提供 XMLHttpRequest，讓程式可以和網頁伺服器建立資料傳輸連線，但這通常只能適用於網站開發的需求，因為我們只能用 XMLHttpRequest 與網頁伺服器通訊，卻無法利用它建立其他類型如 Telnet / FTP / NTP 的伺服器通訊。如果我們想開發網路服務程式，例如 SMTP 電子郵件伺服器，就必須使用 Sockets 建立 TCP (某些服務則用 UDP) 監聽及連線，其他程式語言如 PHP、Java、Python、Perl 及 Ruby 等，在標準開發環境中皆有提供 Sockets API，而瀏覽器的 JavaScript 基於安全及貼近網站設計需求的考量下，並未將 Sockets 列入標準函式庫之中。CommonJS 的規範填補這種基礎函式庫功能的空缺，遵循 CommonJS 規範的 NodeJS 可以直接使用 Sockets API 建立各種網路服務程式。

JavaScript 語言本身支援 Lambda 的特性，因此一個匿名函式 (anonymous function) 可以被儲存成一個變數，並當作參數傳遞給另一個函式。

```
var proc1 = function(op, x) { return op(x);  
}  
  
var op1 = function(x) { return x+1; } var op2 = function(x) { return x*x;  
}  
  
proc1(op1, 3); // result is 4 proc1(op2, 5); // result is 25
```

另一個 JavaScript 開發者必須掌握的語言特性稱為 Closure。

NodeJS 符合 CommonJS 的規範，使得 Callback 方式易於實現，也能夠讓更多同好基於 JavaScript 開發符合 NodeJS 的外掛模組 (Module)。

回想以前要寫一個能夠同時容納上百人的上線的網路服務，需要花費多大的苦工，可能 10 人多就需要經過一次程式調整，而 NodeJS 就是為了解決這個困境，NodeJS 因此誕生，它是一種利用 V8 Javascript 編譯器，所開發的產品，利用 V8 編譯器的高效能，與 Javascript 的程式開發特性所產生的網路程式。

開發人員所編寫出來的 Javascript 腳本程式，怎麼可能會比其他語言寫出來的網路程式還要快上許多呢？以前的網路程式原理是將使用者每次的連線 (connection) 都開啟一個執行緒 (thread)，當連線爆增的時候將會快速耗盡系統效能，並且容易產生阻塞 (block) 的發生。

NodeJS 對於資源的調校有所不同，當程式接收到一筆連線 (connection)，會通知作業系統透過 `epoll`, `kqueue`, `/dev/poll`, 或 `select` 將連線保留，並且放入 `heap` 中配置，先讓連線進入休眠 (Sleep) 狀態，當系統通知時才會觸發連線的 `callback`。這種處理連線方式只會佔用掉記憶體，並不會使用到 CPU 資源。另外因為採用 Javascript 語言的特性，每個 `request` 都會有一個 `callback`，如此可以避免發生 Block 的狀況發生。

基於 Callback 特性，目前 NodeJS 大多應用於 Comet(long pulling) Request Server，或者是高連線數量的網路服務上，目前也有許多公司將 NodeJS 設為內部核心網路服務之一。在 NodeJS 也提供了外掛管理 (Node package management)，讓愛好 NodeJS 輕易開發更多有趣的服務、外掛，並且提供到 `npm` 讓全世界使用者快速安裝使用。

本書最後執行測試版本為 `node.js v0.6.8`，相關 API 文件可查詢 '`http://nodejs.org` <<http://nodejs.org>>' 本書所有範例均可於 `linux`, `windows` 上執行，如遇到任何問題歡迎至 '`http://nodejs.tw` <<http://node.js.tw>>'，詢問對於 `node.js` 相關問題。

JavaScript 與 NodeJS

其實使用 JavaScript 在網頁端與伺服器端的差距並不大，但是為了使 NodeJS 可以發揮他最強大的能力，有一些知識還是必要的，所以還是針對這些主題介紹一下。其中 Event Loop、Scope 以及 Callback 其實是比較需要了解的基本知識，cps、currying、flow control 是更進階的技巧與應用。

2.1 Event Loop

可能很多人在寫 Javascript 時，並不知道他是怎麼被執行的。這個時候可以參考一下 jQuery 作者 John Resig 一篇好文章，介紹事件及 timer 怎麼在瀏覽器中執行：How JavaScript Timers Work。通常在網頁中，所有的 Javascript 執行完畢後（這部份全部都在 global scope 跑，除非執行函數），接下來就是如 John Resig 解釋的這樣，所有的事件處理函數，以及 timer 執行的函數，會排在一個 queue 結構中，利用一個無窮迴圈，不斷從 queue 中取出函數來執行。這個就是 event loop。

（除了 John Resig 的那篇文章，Nicholas C. Zakas 的 “Professional Javascript for Web Developer 2nd edition” 有一個試閱本：<http://yuiblog.com/assets/pdf/zakas-projs-2ed-ch18.pdf>，598 頁剛好也有簡短的說明）

所以在 Javascript 中，雖然有非同步，但是他並不是使用執行緒。所有

的事件或是非同步執行的函數，都是在同一個執行緒中，利用 `event loop` 的方式在執行。至於一些比較慢的動作例如 I/O、網頁 `render`, `reflow` 等，實際動作會在其他執行緒跑，等到有結果時才利用事件來觸發處理函數來處理。這樣的模型有幾個好處：沒有執行緒的額外成本，所以反應速度很快不會有任何程式同時用到同一個變數，不必考慮 `lock`，也不會產生 `dead lock` 所以程式撰寫很簡單但是也有一些潛在問題：任一個函數執行時間較長，都會讓其他函數更慢執行（因為一個跑完才會跑另一個）在多核心硬體普遍的現在，無法用單一的應用程式 `instance` 發揮所有的硬體能力用 NodeJS 撰寫伺服器程式，碰到的也是一樣的狀況。要讓系統發揮 `event loop` 的效能，就要盡量利用事件的方式來組織程式架構。另外，對於一些有可能較為耗時的動作，可以考慮使用 `process.nextTick` 函數來讓他以非同步的方式執行，避免在同一個函數中執行太久，擋住所有函數的執行。

如果想要測試 `event loop` 怎樣在「瀏覽器」中運行，可以在函數中呼叫 `alert()`，這樣會讓所有 Javascript 的執行停下來，尤其會干擾所有使用 `timer` 的函數執行。有一個簡單的例子，這是一個會依照設定的時間間隔嚴格執行動作的動畫，如果時間過了就會跳過要執行的動作。點按圖片以後，人物會快速旋轉，但是在旋轉執行完畢前按下「`delay`」按鈕，讓 `alert` 訊息等久一點，接下來的動畫就完全不會出現了。

2.2 Scope 與 Closure

要快速理解 JavaScript 的 `Scope`（變數作用範圍）原理，只要記住他是 `Lexical Scope` 就差不多了。簡單地說，變數作用範圍是依照程式定義時（或者叫做程式文本?）的上下文決定，而不是執行時的上下文決定。

為了維護程式執行時所依賴的變數，即使執行時程式運行在原本的 `scope` 之外，他的變數作用範圍仍然維持不變。這時程式依賴的自由變數（定義時不是 `local` 的，而是在上一層 `scope` 定義的變數）一樣可以使用，就好像被關閉起來，所以叫做 `Closure`。用程式看比較好懂：

```
function outer(arg1) {  
  //arg1 及 free_variable1 對 inner 函數來說，都是自由變數  
  var free_variable1 = 3;  
  return function inner(arg2) {
```

```

    var local_variable1 = 2; //arg2 及 local_variable1 對 inner 函數來說，都是本地變數
    return arg1 + arg2 + free_variable + local_variable1;
  };
}

```

var a = outter(1); //變數 a 就是 outter 函數執行後返回的 inner 函數 var b = a(4); //執行 inner 函數，執行時上下文已經在 outter 函數之外，但是仍然能正常執行，而且可以使用定義在 outter 函數裡面的 arg1 及 free_variable 變數 console.log(b); //結果 10

在 Javascript 中，scope 最主要的單位是函數（另外有 global 及 eval），所以有可能製造出 closure 的狀況，通常在形式上都是有巢狀的函數定義，而且內側的函數使用到定義在外側函數裡面的變數。

Closure 有可能會造成記憶體洩漏，主要是因為被參考的變數無法被垃圾收集機制處理，造成佔用的資源無法釋放，所以使用上必須考慮清楚，不要造成意外的記憶體洩漏。（在上面的例子中，如果 a 一直未執行，使用到的記憶體就不會被釋放）

跟透過函數的參數把變數傳給函數比較起來，Javascript Engine 會比較難對 Closure 進行最佳化。如果有效能上的考量，這一點也需要注意。

2.3 Callback

要介紹 Callback 之前，要先提到 JavaScript 的特色。

JavaScript 是一種函數式語言（functional language），所有 Javascript 語言內的函數，都是高階函數（higher order function，這是數學名詞，計算機用語好像是 first class function，意指函數使用沒有任何限制，與其他物件一樣）。也就是說，函數可以作為函數的參數傳給函數，也可以當作函數的返回值。這個特性，讓 Javascript 的函數，使用上非常有彈性，而且功能強大。

callback 在形式上，其實就是把函數傳給函數，然後在適當的時機呼叫傳入的函數。Javascript 使用的事件系統，通常就是使用這種形式。NodeJS 中，有一個物件叫做 EventEmitter，這是 NodeJS 事件處理的核心物件，所有會使用事件處理的函數，都會「繼承」這個物件。（這裡說的繼承，實作上應該像是 mixin）他的使用很簡單：可以使用物件.on(事件名稱, callback

函數) 或是物件.addListener(事件名稱, callback 函數) 把你想要處理事件的函數傳入在物件中, 可以使用物件.emit(事件名稱, 參數...) 呼叫傳入的 callback 函數這是 Observer Pattern 的簡單實作, 而且跟在網頁中使用 DOM 的 addEventListener 使用上很類似, 也很容易上手。不過 NodeJS 是大量使用非同步方式執行的應用, 所以程式邏輯幾乎都是寫在 callback 函數中, 當邏輯比較複雜時, 大量的 callback 會讓程式看起來很複雜, 也比較難單元測試。舉例來說:

```
var p_client = new Db('integration_tests_20', new Server("127.0.0.1", 27017,
{}), {'pk':CustomPKFactory}); p_client.open(function(err, p_client) {

    p_client.dropDatabase(function(err, done) {

        p_client.createCollection('test_custom_key', function(err, collection) {

            collection.insert({'a':1}, function(err, docs) {

                collection.find({'_id':new ObjectId("aaaaaaaaaaaa")}, function(err, cursor) {

                    cursor.toArray(function(err, items) {
                        test.assertEquals(1, items.length); p_client.close();
                    });
                });
            });
        });
    });
});
```

這是在網路上看到的一段操作 mongodb 的程式碼, 為了循序操作, 所以必須在一個 callback 裡面呼叫下一個動作要使用的函數, 這個函數裡面還是會使用 callback, 最後就形成一個非常深的巢狀。

這樣的程式碼, 會比較難進行單元測試。有一個簡單的解決方式, 是盡量不要使用匿名函數來當作 callback 或是 event handler。透過這樣的方式,

就可以對各個 handler 做單元測試了。例如：

```
var http = require('http'); var tools = {  
  cookieParser: function(request, response) { if(request.headers['Cookie'])  
    { //do parsing } }  
}; var server = http.createServer(function(request, response) {  
  this.emit('init', request, response); //...  
}); server.on('init', tools.cookieParser); server.listen(8080, '127.0.0.1');
```

更進一步，可以把 tools 改成外部 module，例如叫做 tools.js：

```
module.exports = { cookieParser: function(request, response) {  
  if(request.headers['Cookie']) { //do parsing } }  
};
```

然後把程式改成：

```
var http = require('http');  
var server = http.createServer(function(request, response) { this.emit('init', re-  
  quest, response); //...  
}); server.on('init', require('./ tools').cookieParser); server.listen(8080,  
'127.0.0.1');
```

這樣就可以單元測試 cookieParser 了。例如使用 nodeunit 時，可以這樣寫：

```
var testCase = require('nodeunit').testCase; module.exports = testCase({  
  "setUp": function(cb) { this.request = { headers: { Cookie:  
    'name1:val1; name2:val2' } }; this.response = {}; this.result  
    = { name1:'val1',name2:'val2'};  
    cb();  
  }, "tearDown": function(cb) {  
    cb();
```



```
    }, "normal_case": function(test) {  
  
        test.expect(1);          var      obj      =      require('./  
tools').cookieParser(this.request,      this.response);  
        test.deepEqual(obj, this.result); test.done();  
  
    }  
  
});
```

善於利用模組，可以讓程式更好維護與測試。

2.4 CPS (Continuation-Passing Style)

cps 是 callback 使用上的特例，形式上就是在函數最後呼叫 callback，這樣就好像把函數執行後把結果交給 callback 繼續運行，所以稱作 continuation-passing style。利用 cps，可以在非同步執行的情況下，透過傳給 callback 的這個 cps callback 來獲知 callback 執行完畢，或是取得執行結果。例如：

```
<html> <body> <div id="panel" style="visibility:hidden"></ div>  
</ body> </ html> <script> var request = new XMLHttpRequest(); re-  
quest.open('GET', 'test749.txt? timestamp='+new Date().getTime(), true); re-  
quest.addEventListener('readystatechange', function(next){  
  
    return function() { if(this.readyState===4&&this.status===200) {  
        next(this.responseText);//<== 傳入的 cps callback 在動作完成時執  
        行並取得結果進一步處理 } };  
  
})(function(str){//<== 這個匿名函數就是 cps callback  
    document.getElementById('panel').innerHTML=str;                docu-  
ment.getElementById('panel').style.visibility = 'visible';  
  
}), false); request.send(); </script>
```

進一步的應用，也可以參考 2-6 流程控制。

2.5 函數返回函數與 Currying

前面的 cps 範例裡面，使用了函數返回函數，這是為了把 cps callback 傳遞給 `onreadystatechange` 事件處理函數的方法。（因為這個事件處理函數並沒有設計好會傳送/接收這樣的參數）實際會執行的事件處理函數其實是內層返回的那個函數，之外包覆的這個函數，主要是為了利用 Closure，把 next 傳給內層的事件處理函數。這個方法更常使用的地方，是為了解決一些 scope 問題。例如：

```
<script> var accu=0,count=10; for(var i=0; i<count; i++) {  
    setTimeout(  
        function(){ count--; accu+=i; if(count<=0)  
            console.log(accu)  
        }  
        , 50)  
    } </script>
```

最後得出的結果會是 100，而不是想像中的 45，這是因為等到 `setTimeout` 指定的函數執行時，變數 `i` 已經變成 10 而離開迴圈了。要解決這個問題，就需要透過 Closure 來保存變數 `i`：

```
<script> var accu=0,count=10; for(var i=0; i<count; i++) {  
    setTimeout(  
        function(i) { return function(){ count--;  
            accu+=i; if(count<=0)  
                console.log(accu)  
            };  
        }(i)  
        , 50)  
    }
```

```
} //淺藍色底色的部份，是跟上面例子不一樣的地方 </script>
```

函數返回函數的另外一個用途，是可以暫緩函數執行。例如：

```
function add(m, n) { return m+n;

    } var a = add(20, 10); console.log(a);
```

add 這個函數，必須同時輸入兩個參數，才有辦法執行。如果我希望這個函數可以先給它一個參數，等一些處理過後再給一個參數，然後得到結果，就必須用函數返回函數的方式做修改：

```
function add(m) {

    return function(n) { return m+n;

    };

} var wait_another_arg = add(20); //先給一個參數 var a = function(arr) {

    var ret=0; for(var i=0;i<arr.length;i++) ret+=arr[i]; return ret;

}([1,2,3,4]); //計算一下另一個參數 var b = wait_another_arg(a); //然後再繼續執行 console.log(b);
```

像這樣利用函數返回函數，使得原本接受多個參數的函數，可以一次接受一個參數，直到參數接收完成才執行得到結果的方式，有一個學名就叫做...Currying

綜合以上許多奇技淫巧，就可以透過用函數來處理函數的方式，調整程式流程。接下來看看...

2.6 流程控制

(以 sync 方式使用 async 函數、避開巢狀 callback 循序呼叫 async callback 等奇技淫巧)

建議參考：

- <http://howtonode.org/control-flow>
- <http://howtonode.org/control-flow-part-ii>

- <http://howtonode.org/control-flow-part-iii>
- <http://blog.mixu.net/2011/02/02/essential-node-js-patterns-and-snippets>

這幾篇都是非常經典的 NodeJS/Javascript 流程控制好文章（阿，mixu 是在介紹一些 pattern 時提到這方面的主題）。不過我還是用幾個簡單的程式介紹一下做法跟概念：

並發與等待

下面的程式參考了 mixu 文章中的做法：

```
var wait = function(callbacks, done) { console.log('wait start'); var counter = callbacks.length; var results = []; var next = function(result) { //接收函數執行結果，並判斷是否結束執行 results.push(result); if(--counter == 0) { done(results); //如果結束執行，就把所有執行結果傳給指定的 callback 處理 } }; for(var i = 0; i < callbacks.length; i++) { //依次呼叫所有要執行的函數 callbacks[i](next); } console.log('wait end'); }

wait( [ function(next){ setTimeout(function(){ console.log('done a'); var result = 500; next(result) }, 500); }, function(next){ setTimeout(function(){ console.log('done b'); var result = 1000; next(result) }, 1000); }, function(next){ setTimeout(function(){ console.log('done c'); var result = 1500; next(1500) }, 1500); } ], function(results){ var ret = 0, i=0; for(; i<results.length; i++) { ret += results[i]; } console.log('done all. result: '+ret); } );
```

執行結果：wait start wait end done a done b done c done all. result: 3000

可以看出來，其實 wait 並不是真的等到所有函數執行完才結束執行，而是在所有傳給他的函數執行完畢後（不論同步、非同步），才執行處理結果的函數（也就是 done()）

不過這樣的寫法，還不夠實用，因為沒辦法實際讓函數可以等待執行完畢，又能當作事件處理函數來實際使用。上面參考到的 Tim Caswell 的文

章，裡面有一種解法，不過還需要額外包裝（在他的例子中）NodeJS 核心的 fs 物件，把一些函數（例如 readFile）用 Currying 處理。類似像這樣：

```
var fs = require('fs'); var readFile = function(path) {  
  return function(callback, errback) {  
    fs.readFile(path, function(err, data) {  
      if(err) { errback();  
    } else { callback(data);  
    }  
  });  
};  
}
```

其他部份可以參考 Tim Caswell 的文章，他的 Do.parallel 跟上面的 wait 差不多意思，這裡只提示一下他沒說到的地方。

另外一種做法是去修飾一下 callback，當他作為事件處理函數執行後，再用 cps 的方式取得結果：

```
<script> function Wait(fns, done) {  
  var count = 0; var results = []; this.getCallback = function(index) {  
    count++; return (function(waitback) {  
      return function() { var i=0,args=[];  
        for(;i<arguments.length;i++) {  
          args.push(arguments[i]);  
        } args.push(waitback); fns[index].apply(this,  
          args);  
      };  
    })(function(result) { results.push(result); if(--count == 0) {
```

```

        done(results);

    }

});

}

} var a = new Wait(

[ function(waitback){ console.log('done a'); var result = 500; wait-
back(result) }, function(waitback){ console.log('done b'); var result =
1000; waitback(result) }, function(waitback){ console.log('done c'); var
result = 1500; waitback(result) } ], function(results){ var ret = 0, i=0;
for( i<results.length; i++) { ret += results[i]; } console.log('done all.
result: '+ret); }

); var callbacks = [a.getCallback(0),a.getCallback(1),a.getCallback(0),a.getCallback(2)];
//一次取出要使用的 callbacks, 避免結果提早送出 setTimeout(callbacks[0],
500); setTimeout(callbacks[1], 1000); setTimeout(callbacks[2], 1500); setTime-
out(callbacks[3], 2000); //當所有取出的 callbacks 執行完畢, 就呼叫 done() 來
處理結果 </script>

```

執行結果：

```
done a done b done a done c done all. result: 3500
```

上面只是一些小實驗，更成熟的作品是 Tim Caswell 的 step: <https://github.com/creationix/step>

如果希望真正使用同步的方式寫非同步，則需要使用 Promise.js 這一類的 library 來轉換非同步函數，不過他結構比較複雜 XD（見仁見智，不過有些人認為 Promise 有點過頭了）：<http://blogs.msdn.com/b/rbuckton/archive/2011/08/15/promise-js-2-0-promise-framework-for-javascript.aspx>

如果不透過其他 Library 做轉換，又能直接用同步方式執行非同步函數，大概就要使用一些需要額外 compile 原始程式碼的方法了。例如 Bruno Jounier 的 streamline.js: <https://github.com/Sage/streamlinejs>

循序執行

循序執行可以協助把非常深的巢狀 `callback` 結構攤平，例如用這樣的簡單模組來做（`serial.js`）：

```
module.exports = function(funs) { var c = 0; if(!isArrayOfFunctions(funs)) {  
    throw('Argument type was not matched. Should be array of func-  
    tions.');
```

```
    } return function() {  
        var  args  = Array.prototype.slice.call(arguments,  0);  if(!  
        (c>=funs.length)) {  
            c++; return funs[c-1].apply(this, args);  
        }  
    };  
}
```

```
function isArrayOfFunctions(f) { if(typeof f !== 'object') return false; if(!f.length)  
    return false; if(!f.concat) return false; if(!f.splice) return false; var i = 0; for(;  
    i<f.length; i++) {  
        if(typeof f[i] !== 'function') return false;  
    } return true;  
}
```

簡單的測試範例（`testSerial.js`），使用 `fs` 模組，確定某個 `path` 是檔案，然後讀取印出檔案內容。這樣會用到兩層的 `callback`，所以測試中有使用 `serial` 的版本與 `nested callbacks` 的版本做對照：

```
var serial = require('./serial'), fs = require('fs'), path = './dclient.js', cb = serial([ func-  
    tion(err, data) {  
        if(!err) {  
            if(data.isFile) { fs.readFile(path, cb);
```

```
    }  
  } else { console.log(err);  
  }  
}, function(err, data) {  
  if(!err) { console.log('[flattened by searial:]'); console.log(data.toString('utf8'));  
  } else { console.log(err);  
  }  
}  
]); fs.stat(path, cb);  
fs.stat(path, function(err, data) { //第一層 callback if(!err) {  
  if(data.isFile) {  
    fs.readFile(path, function(err, data) { //第二層 callback if(!err) {  
      console.log('[nested callbacks:]'); console.log(data.toString('utf8'));  
    } else { console.log(err);  
    }  
  }  
});  
  } else { console.log(err);  
  }  
}  
});
```

關鍵在於，這些 callback 的執行是有順序性的，所以利用 `serial` 返回的一個函數 `cb` 來取代這些 callback，然後在 `cb` 中控制每次會循序呼叫的函數，

就可以把巢狀的 callback 攤平成循序的 function 陣列（就是傳給 serial 函數的參數）。

測試中的 ./dclient.js 是一個簡單的 dnode 測試程式，放在跟 testSerial.js 同一個目錄：

```
var dnode = require('dnode');

dnode.connect(8000, 'localhost', function(remote) {

  remote.restart(function(str) { console.log(str); process.exit();

  });

});
```

執行測試程式後，出現結果：

```
[flattened by searial:] var dnode = require('dnode');

dnode.connect(8000, 'localhost', function(remote) {

  remote.restart(function(str) { console.log(str); process.exit();

  });

});

[nested callbacks:] var dnode = require('dnode');

dnode.connect(8000, 'localhost', function(remote) {

  remote.restart(function(str) { console.log(str); process.exit();

  });

});
```

對照起來看，兩種寫法的結果其實是一樣的，但是利用 serial.js，巢狀的 callback 結構就會消失。

不過這樣也只限於順序單純的狀況，如果函數執行的順序比較複雜（不只是一直線），還是需要用功能更完整的流程控制模組比較好，例如 <https://github.com/caolan/async>。

Node.js 安裝與設定

本篇將講解如何在各個不同 OS 建立 NodeJS 環境，目前 NodeJS 0.4.8 版本環境架設方式需依賴 Linux 指令才可編譯完成，當然在不同作業系統中也已經有 NodeJS package，可以直接使用指令快速架設。以下各不同作業系統解說如何安裝 NodeJS。

3.1 Linux base

Ubuntu Linux 使用 apt-get 安裝 Node.js

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:chris-lea/node.js-devel
sudo apt-get update
sudo apt-get nodejs
```

Linux 很適合作為 NodeJS 的伺服器作業系統及開發環境。安裝前，請先確認以下套件已正確安裝。

- curl (wget) 用來下載檔案的工具
- git 先進的版本控制工具

- g++ GNU C++ 軟體編譯工具
- make GNU 軟體專案建置工具

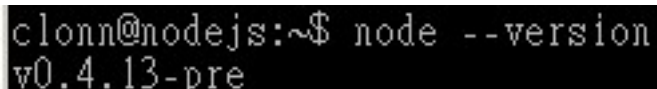
安裝指令如下，如設有權限問題，請在指令前面加上 `sudo`

```
git clone https://github.com/joyent/node.git
cd node
git checkout v0.6.7
./configure
make
sudo make install
```

接著測試 nodeJS 是否正常執行

```
node --version
```

出現版本訊息即表示安裝成功。



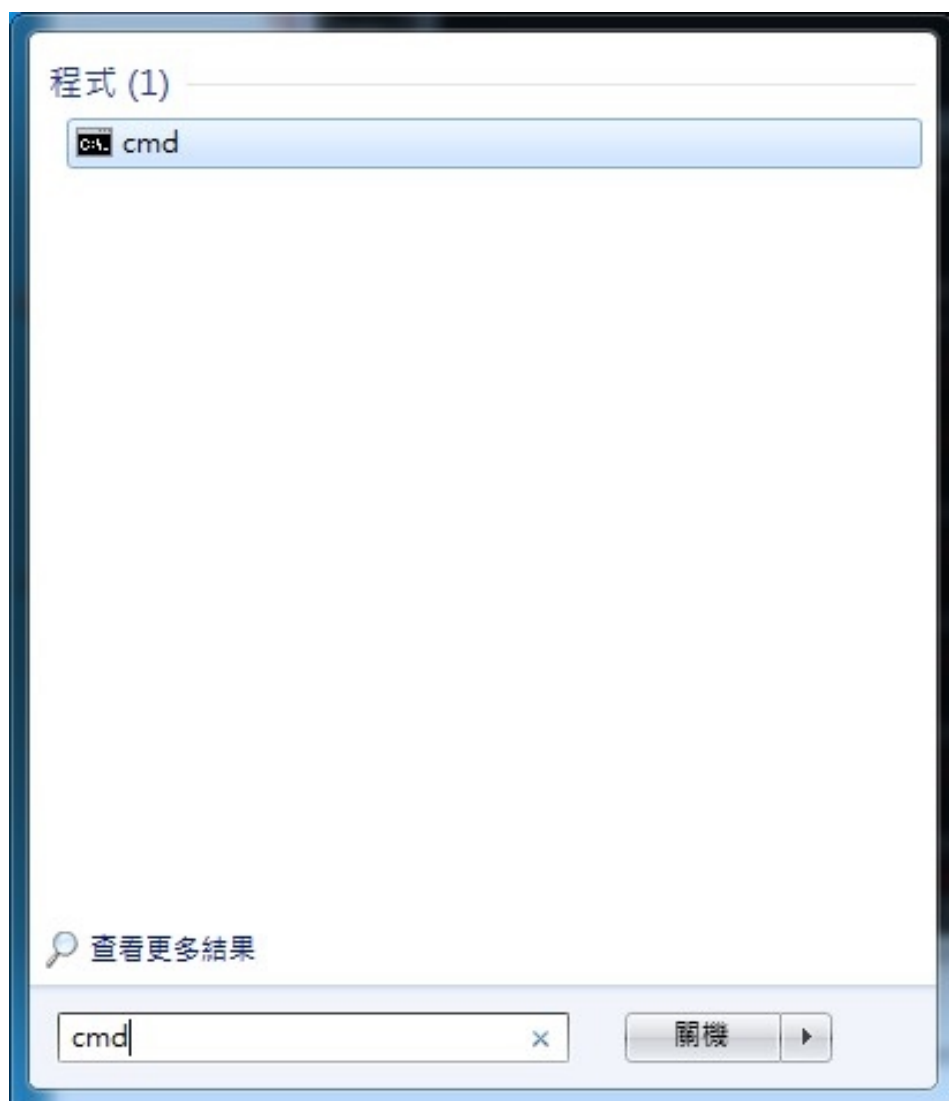
```
clonn@nodejs:~$ node --version
v0.4.13-pre
```

3.2 Windows

nodeJS 在 v0.6.0 版本之後開始正式支援 windows native，直接使用 `node.exe` 就可以執行程式，支援性完全與 linux 相同，更棒的部份就是不需經過編譯，經過下載之後，簡單設定完成，立即開發 node 程式。

下載 `node.js` 安裝檔案 <<http://nodejs.org/#download>>

如此完成 windows native `node.exe` 安裝，接著可以進入 command line 執行測試。在 command line 輸入“node”，會出現 node.js 版本訊息畫面，表示安裝完成。



```
C:\Users\Caesar>node  
>
```

Node.js 基礎

前篇文章已經由介紹、安裝至設定都有完整介紹，nodeJS 內部除了 javascript 常用的函式 (function)、物件 (object) 之外，也有許多不同的自訂物件，nodeJS 預設建立這些物件為核心物件，是為了要讓開發流程更為，這些資料在官方文件已經具有許多具體說明。接下來將會介紹在開發 nodeJS 程式時常見的物件特性與使用方法。

4.1 node.js http 伺服器建立

在 'node.js 官方網站 <<http://nodejs.org>>' 裡面有舉一個最簡單的 HTTP 伺服器建立，一開始初步就是建立一個伺服器平台，讓 node.js 可以與瀏覽器互相行為。每種語言一開始的程式建立都是以 Hello world 開始，最初也從 Hello world 帶各位進入 node.js 的世界。

輸入以下程式碼，儲存檔案為 node_basic_http_hello_world.js

```
var server,
    ip    = "127.0.0.1",
    port  = 1337,
    http  = require('http');

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

```
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World\n');
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

程式碼解講，一開始需要有幾個基本的變數。

- ip: 機器本身的 ip 位置，因為使用本地端，因此設定為 127.0.0.1
- port: 需要開通的埠號，通常設定為 http port 80，因範例不希望與基本 port 相衝，隨意設定為 1337

在 node.js 的程式中，有許多預設的模組可以使用，因此需要使用 `require` 方法將模組引入，在這邊我們需要使用 `http` 這個模組，因此將 `http` 載入。`Http` 模組裡面內建有許多方法可以使用，這邊採用 `createServer` 創建一個基本的 `http` 伺服器，再將 `http` 伺服器給予一個 `server` 變數。裡面的回呼函式 (call back function) 可以載入 `http` 伺服器的資料與回應方法 (`request`, `response`)。在程式裡面就可以看到我們直接回應給瀏覽器端所需的 `Header`，回應內容。

```
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World\n');
```

`Http` 伺服器需要設定 `port`, `ip`，在最後需要設定 `Http` 監聽，需要使用到 `listen` 事件，監聽所有 `Http` 伺服器行為。

```
http.listen(port, ip);
```

所有事情都完成之後，需要確認伺服器正確執行因此使用 `console`，在 `javascript` 裡就有這個原生物件，`console` 所印出的資料都會顯示於 `node.js` 伺服器頁面，這邊印出的資料並不會傳送到使用者頁面上，之後許多除壞 (`debug`) 都會用到 `console` 物件。

```
console.log("Server running at http://" + ip + ":" + port);
```

4.2 node.js http 路徑建立

前面已經介紹如何建立一個簡單的 http 伺服器，接下來本章節將會介紹如何處理伺服器路徑 (route) 問題。在 http 的協定下所有從瀏覽器發出的要求 (request) 都需要經過處理，路徑上的建立也是如此。

路徑就是指伺服器 ip 位置，或者是網域名稱之後，對於伺服器給予的要求。修改剛才的 hello world 檔案，修改如下。

```
server = http.createServer(function (req, res) {  
  console.log(req.url);  
  res.writeHead(200, {'Content-Type': 'text/plain'});  
  res.end('hello world\n');  
});
```

重新啟動 node.js 程式後，在瀏覽器端測試一下路徑行為，結果如下圖，



當在瀏覽器輸入 `http://127.0.0.1:1337/test`，在伺服器端會收到兩個要求，一個是我們輸入的 `/test` 要求，另外一個則是 `/favicon.ico`。`/test` 的路徑要求，http 伺服器本身需要經過程式設定才有辦法回應給瀏覽器端所需要的回應，在伺服器中所有的路徑要求都是需要被解析才有辦法取得資料。從

上面解說可以了解到在 `node.js` 當中所有的路徑都需要經過設定，未經過設定的路由會讓瀏覽器無法取得任何資料導致錯誤頁面的發生，底下將會解說如何設定路由，同時避免發生錯誤情形。先前 `node.js` 程式需要增加一些修改，才能讓使用者透過瀏覽器，在不同路徑時有不同的結果。根據剛才的程式做如下的修改，

```
var server,
    ip    = "127.0.0.1",
    port  = 1337,
    http  = require('http'),
    url   = require('url'),
    path;

server = http.createServer(function (req, res) {
  path = url.parse(req.url);

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

  switch (path.pathname) {
    case "/index":
      res.end('I am index.\n');
      break;
    case "/test":
      res.end('this is test page.\n');
      break;
    default:
      res.end('default page.\n');
      break;
  }
});

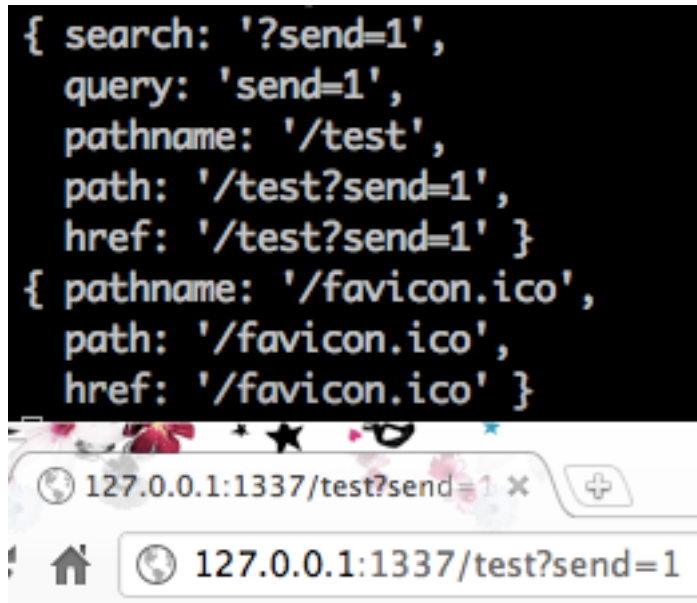
server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

程式做了片段的修改，首先載入 `url` 模組，另外增加一個 `path` 變數。`url` 模組就跟如同他的命名一般，專門處理 `url` 字串處理，裡面提供了許多方法來解決路徑上的問題。因為從瀏覽器發出的要求路徑可能會帶有多種需求，或者 `GET` 參數組合等。因此我們需要將路徑單純化，取用路徑部分的資料即可，例如使用者可能會送出 `http://127.0.0.1:1337/test?send=1`，如果直接

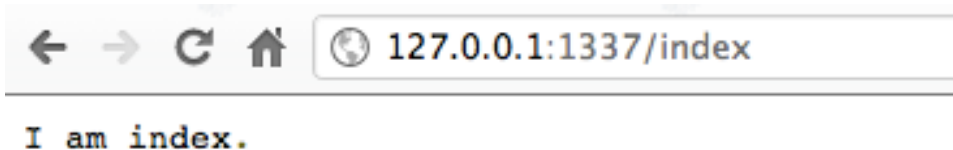
信任 `req.url` 就會收到結果為 `/test?send=1`，所以需要透過 `url` 模組的方法將路徑資料過濾。

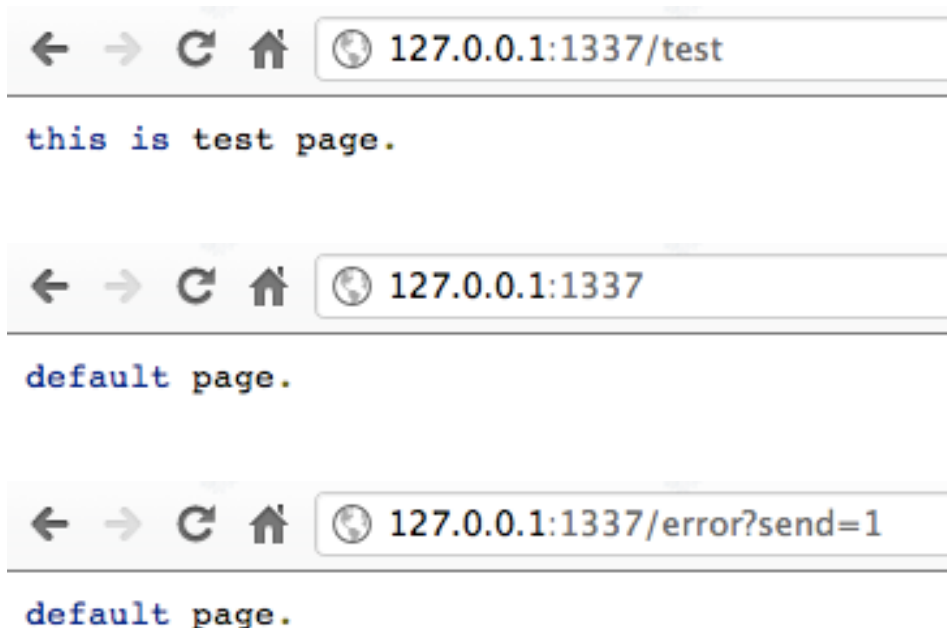
在這邊使用 `url.parse` 的方法，裡面帶入網址格式資料，會回傳路徑資料。為了後需方便使用，將回傳的資料設定到 `path` 變數當中。在回傳的路徑資料，裡面包含資訊，如下圖，



這邊只需要使用單純的路徑要求，直接取用 `path.pathname`，就可以達到我們的目的。

最後要做路徑的判別，在不同的路徑可以指定不同的輸出，在範例中有三個可能結果，第一個從瀏覽器輸入 `/index` 就會顯示 `index` 結果，`/test` 就會呈現出 `test` 頁面，最後如果都不符合預期的輸入會直接顯示 `default` 的頁面，最後的預防可以讓瀏覽器不會出現非預期結果，讓程式的可靠性提昇，底下為測試結果。





4.3 node.js 檔案讀取

前面已經介紹如何使用路由 (route) 做出不同的回應，實際應用只有在瀏覽器只有輸出幾個文字資料總是不夠的，在本章節中將介紹如何使用檔案讀取，輸出檔案資料，讓使用者在前端瀏覽器也可以讀取到完整的 html, css, javascript 檔案輸出。

檔案管理最重要的部分就是 'File system <<http://nodejs.org/docs/latest/api/fs.html>>' 這個模組，此模組可以針對檔案做管理、監控、讀取等行為，裡面有許多預設的方法，底下是檔案輸出的基本範例，底下會有兩個檔案，第一個是靜態 html 檔案，

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-7" />
  <title>node.js index html file</title>
</head>
<body>
```

```

    <h1>node.js index html file</h1>
  </body>
</html>

```

另一個為 node.js 程式，

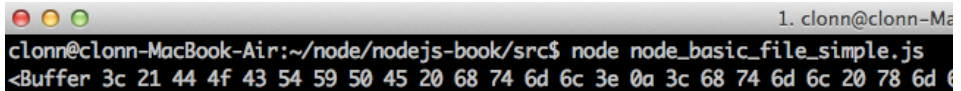
```

var fs = require("fs"),
    filename = "static/index.html",
    encode = "utf8";

fs.readFile(filename, encode, function(err, file) {
  console.log(file);
});

```

一開始直接載入 **file system** 模組，載入名稱為 **fs**。讀取檔案主要使用的方法為 **readFile**，裡面以三個參數 **路徑 (file path)**，**編碼方式 (encoding)**，**回應函式 (callback)**，路徑必須要設定為靜態 html 所在位置，才能指定到正確的檔案。靜態檔案的編碼方式也必須正確，這邊使用靜態檔案的編碼為 **utf8**，如果編碼設定錯誤，node.js 讀取出來檔案結果會使用 **byte raw** 格式輸出，如果 **錯誤編碼格式**，會導致輸出資料為 **byte raw**



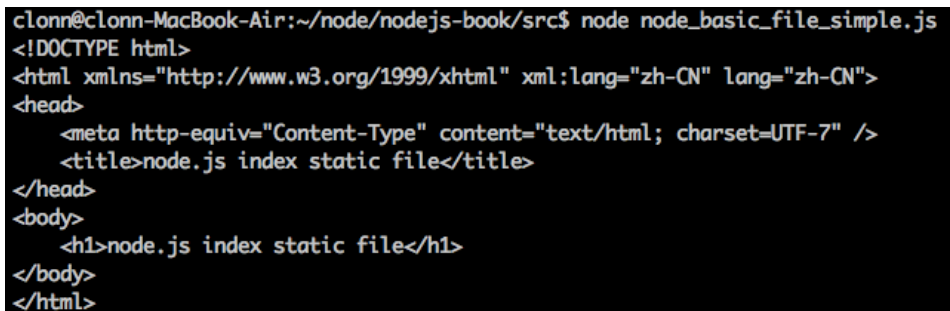
```

1. clonn@clonn-Ma
clonn@clonn-MacBook-Air:~/node/nodejs-book/src$ node node_basic_file_simple.js
<Buffer 3c 21 44 4f 43 54 59 50 45 20 68 74 6d 6c 3e 0a 3c 68 74 6d 6c 20 78 6d 6

```

回應函式 中裡面會使用兩個變數，**error** 為錯誤資訊，如果讀取的檔案不存在，或者發生錯誤，**error** 數值會是 **true**，如果成功讀取資料 **error** 將會是 **false**。**content** 則是檔案內容，資料讀取後將會把資料全數丟到 **content** 這個變數當中。

最後程式的輸出結果畫面如下，



```

clonn@clonn-MacBook-Air:~/node/nodejs-book/src$ node node_basic_file_simple.js
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-7" />
  <title>node.js index static file</title>
</head>
<body>
  <h1>node.js index static file</h1>
</body>
</html>

```

4.4 node.js http 靜態檔案輸出

前面已經了解如何讀取本地端檔案，接下來將配合 http 伺服器路由，讓每個路由都能夠輸出相對應的靜態 html 檔案。首先新增加幾個靜態 html 檔案，

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-7" />
  <title>node.js index html file</title>
</head>
<body>
  <h1>node.js index html file</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-7" />
  <title>node.js test html file</title>
</head>
<body>
  <h1>node.js test html file</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-7" />
  <title>node.js static html file</title>
</head>
<body>
  <h1>node.js static html file</h1>
</body>
</html>
```

準備一個包含基本路由功能的 http 伺服器

```
var server,
    ip    = "127.0.0.1",
    port  = 1337,
    http  = require('http'),
    url   = require('url');

server = http.createServer(function (req, res) {
    var path = url.parse(req.url);
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

加入 `file system` 模組，使用 `readFile` 的功能，將這一段程式放置於 `createServer` 的回應函式中。

```
fs.readFile(filePath, encode, function(err, file) {
});
```

`readFile` 的回應函式裡面加入頁面輸出，讓瀏覽器可以正確讀到檔案，在這邊我們設定讀取的檔案為 `html` 靜態檔案，所以 `Content-type` 設定為 `text/html`。讀取到檔案的內容，將會正確輸出成 `html` 靜態檔案。

```
fs.readFile(filePath, encode, function(err, file) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(file);
    res.end();
});
```

到這邊為止基本的程式內容都已經完成，剩下一些細節的調整。首先路徑上必須做調整，目前的靜態檔案全部都放置於 `static 資料夾` 底下，設定一個變數來記住資料夾位置。

接著將瀏覽器發出要求路徑與資料夾組合，讀取正確 `html` 靜態檔案。使用者有可能會輸入錯誤路徑，所以在讀取檔案的時候要加入錯誤處理，同時回應 `404` 伺服器無法正確回應的 `http header` 格式。

加入這些細節的修改，一個基本的 `http` 靜態 `html` 輸出伺服器就完成了，完整程式碼如下，

```
var server,
    ip    = "127.0.0.1",
    port  = 1337,
    http  = require('http'),
    fs    = require("fs"),
    folderPath = "static",
    url   = require('url'),
    path,
    filePath,
    encode = "utf8";

server = http.createServer(function (req, res) {
    path = url.parse(req.url);
    filePath = folderPath + path.pathname;

    fs.readFile(filePath, encode, function(err, file) {
        if (err) {
            res.writeHead(404, {'Content-Type': 'text/plain'});
            res.end();
            return;
        }

        res.writeHead(200, {'Content-Type': 'text/application'});
        res.write(file);
        res.end();
    });
});

server.listen(port, ip);

console.log("Server running at http://" + ip + ":" + port);
```

4.5 node.js http GET 資料擷取

http 伺服器中，除了路由之外另一個最常使用的方法就是擷取 GET 資料。本單元將會介紹如何透過基本 http 伺服器擷取瀏覽器傳來的要求，擷取 GET 資料。

在 http 協定中，GET 參數都是藉由 URL 從瀏覽器發出要求送至伺服器

端，基本的傳送網址格式可能如下，

```
http://127.0.0.1/test?send=1&test=2
```

上面這段網址，裡面的 GET 參數就是 send 而這個變數的數值就為 1，如果想要在 http 伺服器取得 GET 資料，需要在瀏覽器給予的要求 (request) 做處理，

首先需要載入 **query string** 這個模組，這個模組主要是用來將字串資料過濾後，轉換成 **javascript** 物件 ******。

```
qs = require('querystring');
```

接著在第一階段，利用 **url** 模組過濾瀏覽器發出的 URL 資料後，將回應的物件裡面的 **query** 這個變數，是一個字串值，資料過濾後如下，

```
send=1&test=2
```

透過 **query string**，使用 **parse** 這個方法將資料轉換成 **javascript** 物件，就表示 GET 的資料已經被伺服器端正式擷取下來，

```
path = url.parse(req.url);  
parameter = qs.parse(path.query);
```

整個 **node.js** http GET 參數完整擷取程式碼如下，

```
var server,  
    ip    = "127.0.0.1",  
    port  = 1337,  
    http  = require('http'),  
    qs    = require('querystring'),  
    url   = require('url');  
  
server = http.createServer(function (req, res) {  
    var path = url.parse(req.url),  
        parameter = qs.parse(path.query);  
  
    console.dir(parameter);  
  
    res.writeHead(200, {'Content-Type': 'text/plain'});  
    res.write('Browser test GET parameter\n');  
    res.end();  
});
```



```
server.listen(port, ip);
```

```
console.log("Server running at http://" + ip + ":" + port);
```

程式運作之後，由瀏覽器輸入要求網址之後，node.js 伺服器端回應資料為，

```
Server running at http://127.0.0.1:1337  
{ send: '1', test: '1' }
```

4.6 本章結語

前面所解說的部份，一大部分主要是處理 http 伺服器基本問題，雖然在某些部分有牽扯到 http 伺服器基本運作原理，主要還是希望可以藉由這些基本範例練習 node.js，練習回應函式與語法串接的特點，習慣編寫 javascript 風格程式。當然直接這樣開發 node.js 是非常辛苦的，接下來在模組實戰開發的部份將會介紹特定的模組，一步一步帶領各位從無到有進行 node.js 應用程式開發。

NPM 套件管理工具

npm 全名為 Node Package Manager，是 Node.js 的套件（package）管理工具，類似 Perl 的 ppm 或 PHP 的 PEAR 等。安裝 npm 後，使用 `npm install module_name` 指令即可安裝新套件，維護管理套件的工作會更加輕鬆。

npm 可以讓 Node.js 的開發者，直接利用、擴充線上的套件庫（packages registry），加速軟體專案的開發。npm 提供很友善的搜尋功能，可以快速找到、安裝需要的套件，當這些套件發行新版本時，npm 也可以協助開發者自動更新這些套件。

npm 不僅可用於安裝新的套件，它也支援搜尋、列出已安裝模組及更新的功能。

5.1 安裝 NPM

Node.js 在 0.6.3 版本開始內建 npm，讀者安裝的版本若是此版本或更新的版本，就可以略過以下安裝說明。

若要檢查 npm 是否正確安裝，可以使用以下的指令：

```
npm -v
```

執行結果說明

若 npm 正確安裝，執行 `npm -v` 將會看到類似 1.1.0-2 的版本訊息。

若讀者安裝的 Node.js 版本比較舊，或是有興趣嘗試自己動手安裝 npm 工具，則可以參考以下的說明。

安裝於 Windows 系統

Node.js for Windows 於 0.6.2 版開始內建 npm，使用 nodejs.org 官方提供的安裝程式，不需要進一步的設定，就可以立即使用 npm 指令，對於 Windows 的開發者來說，大幅降低環境設定的問題與門檻。

除了使用 Node.js 內建的 npm，讀者也可以從 npm 官方提供的以下網址：

<http://npmjs.org/dist/>

這是由 npm 提供的 Fancy Windows Install 版本，請下載壓縮檔（例如：`npm-1.1.0-3.zip`），並將壓縮檔內容解壓縮至 Node.js 的安裝路徑（例如：`C:\Program Files\nodejs`）。

解壓縮後，在 Node.js 的安裝路徑下，應該有以下的檔案及資料夾。

- `npm.cmd`（檔案）
- `node_modules`（資料夾）

安裝於 Linux 系統

Ubuntu Linux 的使用者，可以加入 [NPM Unofficial PPA](#) 這個 repository，即可使用 `apt-get` 完成 npm 安裝。

Ubuntu Linux 使用 apt-get 安裝 npm

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:gijs-kay-lee/npm
sudo apt-get update
sudo apt-get npm
```

npm 官方提供的安裝程式 `install.sh`，可以適用於大多數的 Linux 系統。使用這個安裝程式，請先確認：

1. 系統已安裝 `curl` 工具（請使用 `curl --version` 查看版本訊息）
2. 已安裝 Node.js 並且 `PATH` 正確設置
3. Node.js 的版本必須大於 0.4.x

以下為 npm 提供的安裝指令：

```
curl http://npmjs.org/install.sh | sh
```

安裝成功會看到如下訊息：

install.sh 安裝成功的訊息

```
npm@1.0.105 /home/USERNAME/local/node/lib/node_modules/npm
It worked
```

安裝於 Mac OS X

建議採用與 Node.js 相同的方式，進行 npm 的安裝。例如使用 MacPorts 安裝 Node.js，就同樣使用 MacPorts 安裝 npm，這樣對日後的維護才會更方便容易。

使用 MacPorts 安裝 npm 是本書比較建議的方式，它可以讓 npm 的安裝、移除及更新工作自動化，將會幫助開發者節省寶貴時間。

安裝 MacPorts 的提示

在 MacPorts 網站，可以取得 OS X 系統版本對應的安裝程式（例如 10.6 或 10.7）。

<http://www.macports.org/>

安裝過程會詢問系統管理者密碼，使用預設的選項完成安裝即可。安裝 MacPorts 之後，在終端機執行 `port -v` 將會看到 MacPorts 的版本訊息。

安裝 npm 之前，先更新 MacPorts 的套件清單，以確保安裝的 npm 是最新版本。

```
sudo port -d selfupdate
```

接著安裝 npm。

```
sudo port install npm
```

若讀者的 Node.js 並非使用 MacPorts 安裝，則不建議使用 MacPorts 安裝 npm，因為 MacPorts 會自動檢查並安裝相依套件，而 npm 相依 nodejs，所以 MacPorts 也會一併將 nodejs 套件安裝，造成先前讀者使用其它方式安裝的 nodejs 被覆蓋。

讀者可以先使用 MacPorts 安裝 curl (`sudo port install curl`)，再參考 Linux 的 `install.sh` 安裝方式，即可使用 npm 官方提供的安裝程式。

NPM 安裝後測試

npm 是指令列工具 (command-line tool)，使用時請先打開系統的文字終端機工具。

測試 npm 安裝與設定是否正確，請輸入指令如下：

```
npm -v
```

或是：

```
npm --version
```

如果 npm 已經正確安裝設定，就會顯示版本訊息：

執行結果（範例）

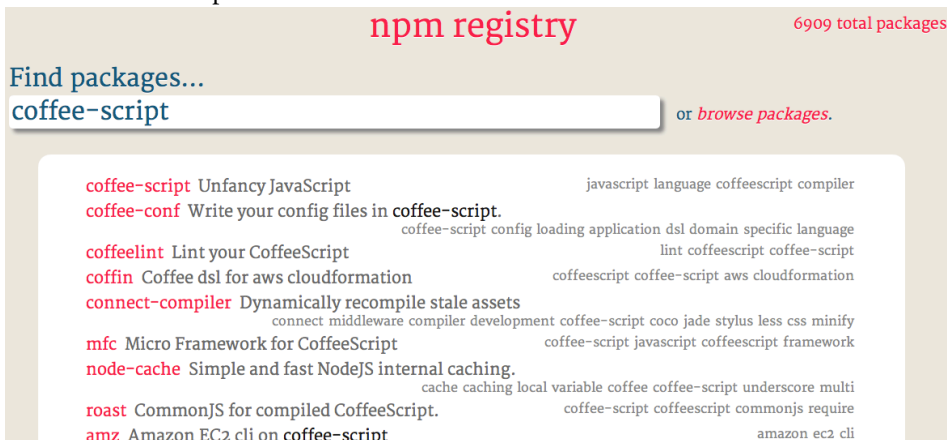
1.1.0-2

5.2 使用 NPM 安裝套件

npm 目前擁有超過 6000 種套件（packages），可以在 [npm registry](https://www.npmjs.com/) 使用關鍵字搜尋套件。

<http://search.npmjs.org/>

舉例來說，在關鍵字欄位輸入「coffee-script」，下方的清單就會自動列出包含 coffee-script 關鍵字的套件。



接著我們回到終端機模式的操作，npm 的指令工具本身就可以完成套件搜尋的任務。

例如，以下的指令同樣可以找出 coffee-script 相關套件。

```
npm search coffee-script
```

以下是搜尋結果的參考畫面：

```

終端機 — bash — 131x21
npm http GET https://registry.npmjs.org/-/all/since?stale-update_after&startkey=1328322837000
npm http 200 https://registry.npmjs.org/-/all/since?stale-update_after&startkey=1328322837000
NAME DESCRIPTION AUTHOR DATE KEYWORDS
amz Amazon EC2 cli on coffee-script -seled 2012-01-31 16:39 amazon ec2 cli
coffee-conf Write your config files in coffee-script. -MSNexploder 2011-11-20 21:43 coffee-script c
coffee-script Unfancy JavaScript -jashkenas 2012-01-13 23:07 javascript lang
coffee-toaster Minimalist dependency management system for coffee-script. -nybras 2011-10-29 12:25
coffeeapp CoffeeApp wrapper (handling coffee-script) for CouchApp (http://couchapp.org/). -andrzejliwa 2011-01-28 23:16
coffeescript Lint your CoffeeScript -clutchski 2012-01-26 05:56 lint coffeescri
coffeescript-notify CoffeeScript notify tool for coffee based on the coffeescript-growl tool -bdryanovski 2011-09-17 14:34 coffe
coffin Coffee DSL for AWS CloudFormation -chrisfjones 2012-01-04 20:12 coffeescript co
connect-compiler Dynamically recompile stale assets -dsc 2011-12-26 15:43 connect middlew
cupcake Quick CoffeeScript Web App Template Generator -jackhq 2012-02-04 01:52 javascript expr
gesundheits Concise SQL generation in coffee-script -gmcdr 2012-02-01 06:07
iced-coffee-script IcedCoffeeScript -maxtaco 2012-02-04 01:09 javascript lang
mfc Micro Framework for CoffeeScript -tadeuzagallo 2012-01-17 17:34 coffee-script j
node-cache Simple and fast NodeJS internal caching. -tcs-de 2011-10-20 14:52 cache caching l
roast CommonJS for compiled CoffeeScript. -chrislloyd (prehistoric) coffee-script c
stitcher coffee-script js less css eco commonJS stitcher. -sunliutao 2012-01-17 09:02
tamed-coffee-script Unfancy JavaScript -maxtaco 2011-12-12 22:01 javascript lang
tiers Web framework built on coffee-script. -terryvesper 2011-03-14 22:38 tiers web frame

```

找到需要的套件後（例如 `express`），即可使用以下指令安裝：

```
npm install coffee-script
```

值得注意的一點是，使用 `npm install` 會將指定的套件，安裝在工作目錄（Working Directory）的 `node_modules` 資料夾下。

以 Windows 為例，如果執行 `npm install` 的目錄位於：

```
C:\project1
```

那麼 `npm` 將會自動建立一個 `node_modules` 的子目錄（如果不存在）。

```
C:\project1\node_modules
```

並且將下載的套件，放置於這個子目錄，例如：

```
C:\project1\node_modules\coffee-script
```

這個設計讓專案可以個別管理相依的套件，並且可以在專案佈署或發行時，將這些套件（位於 `node_modules`）一併打包，方便其它專案的使用者不必再重新下載套件。

這個 `npm install` 的預設安裝模式為 **local**（本地），只會變更當前專案的資料夾，不會影響系統。

另一種安裝模式稱為 **global**（全域），這種模式會將套件安裝到系統資料夾，也就是 `npm` 安裝路徑的 `node_modules` 資料夾，例如：

```
C:\Program Files\nodejs\node_modules
```

是否要使用全域安裝，可以依照套件是否提供**新指令**來判斷，舉例來說，`express` 套件提供 `express` 這個指令，而 `coffee-script` 則提供 `coffee` 指令。

在 `local` 安裝模式中，這些指令的程式檔案，會被安裝到 `node_modules` 的 `.bin` 這個隱藏資料夾下。除非將 `.bin` 的路徑加入 `PATH` 環境變數，否則要執行這些指令將會相當不便。

為了方便指令的執行，我們可以在 `npm install` 加上 `-g` 或 `--global` 參數，啟用 `global` 安裝模式。例如：

```
npm install -g coffee-script
npm install -g express
```

使用 `global` 安裝模式，需要注意執行權限的問題，若權限不足，可能會出現類似以下的錯誤訊息：

```
npm ERR! Error: EACCES, permission denied '...'
npm ERR!
npm ERR! Please try running this command again as root/Administrator.
```

要獲得足夠得執行權限，請參考以下說明：

- Windows 7 或 2008 以上，在「命令提示字元」的捷徑按右鍵，選擇「以系統管理員身分執行」，執行 `npm` 指令時就會具有 `Administrator` 身分。
- Mac OS X 或 Linux 系統，可以使用 `sudo` 指令，例如：

```
sudo npm install -g express
```
- Linux 系統可以使用 `root` 權限登入，或是以「`sudo su -`」切換成 `root` 身分。（使用 `root` 權限操作系統相當危險，因此並不建議使用這種方式。）

若加上 `-g` 參數，使用 `npm install -g coffee-script` 完成安裝後，就可以在終端機執行 `coffee` 指令。例如：

```
coffee -v
```


執行結果（範例）

```
CoffeeScript version 1.2.0
```

5.3 套件的更新及維護

除了前一節說明的 `search` 及 `install` 用法，`npm` 還提供其他許多指令（commands）。

使用 `npm help` 可以查詢可用的指令。

```
npm help
```

執行結果（部分）

```
where <command> is one of:
```

```
adduser, apihelp, author, bin, bugs, c, cache, completion,
config, deprecate, docs, edit, explore, faq, find, get,
help, help-search, home, i, info, init, install, la, link,
list, ll, ln, login, ls, outdated, owner, pack, prefix,
prune, publish, r, rb, rebuild, remove, restart, rm, root,
run-script, s, se, search, set, show, star, start, stop,
submodule, tag, test, un, uninstall, unlink, unpublish,
unstar, up, update, version, view, whoami
```

使用 `npm help command` 可以查詢指令的詳細用法。例如：

```
npm help list
```

接下來，本節要介紹開發過程常用的 `npm` 指令。

使用 `list` 可以列出已安裝套件：

```
npm list
```

執行結果（範例）

```
└-- coffee-script@1.2.0
└--┬ express@2.5.6
    ├── connect@1.8.5
    │ └-- formidable@1.0.8
    ├── mime@1.2.4
    ├── mkdirp@0.0.7
    └-- qs@0.4.1
```

檢視某個套件的詳細資訊，例如：

```
npm show express
```

升級所有套件（如果該套件已發佈更新版本）：

```
npm update
```

升級指定的套件：

```
npm update express
```

移除指定的套件：

```
npm uninstall express
```

5.4 使用 package.json

對於正式的 Node.js 專案，可以建立一個命名為 `package.json` 的設定檔（純文字格式），檔案內容參考範例如下：

package.json (範例)

```
{
  "name": "application-name"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
    "express": "2.5.5"
    , "coffee-script": "latest"
    , "mongoose": ">= 2.5.3"
  }
}
```

其中 name 與 version 依照專案的需求設置。

需要注意的是 dependencies 的設定，它用於指定專案相依的套件名稱及版本：

- "express": "2.5.5"

//代表此專案相依版本 2.5.5 的 express 套件

- "coffee-script": "latest"

//使用最新版的 coffee-script 套件（每次更新都會檢查新版）

- "mongoose": ">= 2.5.3"

//使用版本大於 2.5.3 的 mongoose 套件

假設某個套件的新版可能造成專案無法正常運作，就必須指定套件的版本，避免專案的程式碼來不及更新以相容新版套件。通常在開發初期的專案，需要盡可能維持新套件的相容性（以取得套件的更新或修正），可以用「>=」設定最低相容的版本，或是使用「latest」設定永遠保持最新套件。

Express 介紹

在前面的 node.js 基礎當中介紹許多許多開設 http 的使用方法及介紹，以及許多基本的 node.js 基本應用。

接下來要介紹一個套件稱為 express [Express] (<http://expressjs.com/>)，這個套件主要幫忙解決許多 node.js http server 所需要的基本服務，讓開發 http service 變得更為容易，不需要像之前需要透過層層模組 (module) 才有辦法開始編寫自己的程式。

這個套件是由 TJ Holowaychuk 製作而成的套件，裡面包含基本的路由處理 (route)，http 資料處理 (GET/POST/PUT)，另外還與樣板套件 (js html template engine) 搭配，同時也可以處理許多複雜化的問題。

=Express 安裝 =

安裝方式十分簡單，只要透過之前介紹的 NPM 就可以使用簡單的指令安裝，指令如下，

這邊建議需要將此套件安裝成為全域模組，方便日後使用。

=Express 基本操作 =

express 的使用也十分簡單，先來建立一個基本的 hello world，

```
var app = require('express').createServer(),
```

```
port = 1337;

app.listen(port);

app.get('/', function(req, res){
  res.send('hello world');
});

console.log('start express server\n');
```

可以從上面的程式碼發現，基本操作與 node.js http 的建立方式沒有太大差異，主要差在當我們設定路由時，可以直接透過 `app.get` 方式設定回應與接受方式。

==Express 路由處理 ==

Express 對於 http 服務上有許多包裝，讓開發者使用及設定上更為方便，例如有幾個路由設定，那我們就統一藉由 `app.get` 來處理，

```
// ... Create http server

app.get('/', function(req, res){
  res.send('hello world');
});

app.get('/test', function(req, res){
  res.send('test render');
});

app.get('/user/', function(req, res){
  res.send('user page');
});
```

如上面的程式碼所表示，`app.get` 可以帶入兩個參數，第一個是路徑名稱設定，第二個為回應函式 (call back function)，回應函式裡面就如同之前的 `createServer` 方法，裡面包含 `request`，`response` 兩個物件可供使用。使用者就可以透過瀏覽器，輸入不同的 url 切換到不同的頁面，顯示不同的結果。

路由設定上也有基本的配對方式，讓使用者從瀏覽器輸入的網址可以是一個變數，只要符合型態就可以有對應的頁面產出，例如，

```
// ... Create http server
```

```
app.get('/user/:id', function(req, res){
  res.send('user: ' + req.params.id);
});

app.get('/:number', function(req, res){
  res.send('number: ' + req.params.number);
});
```

裡面使用到:number，從網址輸入之後就可以直接使用 req.params.number 取得所輸入的資料，變成 url 參數使用，當然前面也是可以加上路徑的設定，/user/:id，在瀏覽器上路徑必須符合/user/xxx，透過 req.params.id 就可以取到 xxx 這個字串值。

另外，express 參數處理也提供了路由參數配對處理，也可以透過正規表示法作為參數設定，

```
var app = require('express').createServer(),
    port = 1337;

app.listen(port);

app.get(/^\/ip?(?:\/(\d{2,3}))(?:\.(\d{2,3}))?(?:\.(\d{2,3}))?(?:\.(\d{2,3}))?$/),
  res.send(req.params);
});
```

上面程式碼，可以發現後面路由設定的型態是正規表示法，裡面設定格式為/ip 之後，必須要加上 ip 型態才會符合資料格式，同時取得 ip 資料已經由正規表示法將資料做分群，因此可以取得 ip 的四個數字。

此程式執行之後，可以透過瀏覽器測試，輸入網址為 localhost:3000/ip/255.255.100.10，可以從頁面獲得資料，

此章節全部範例程式碼如下，

```
/**
 * @overview
 *
 * @author Caesar Chi
 * @blog clonn.blogspot.com
 * @version 2012/02/26
 */
```

```
// create server.
var app = require('express').createServer(),
    port = 1337;

app.listen(port);

// normal style
app.get('/', function(req, res){
    res.send('hello world');
});

app.get('/test', function(req, res){
    res.send('test render');
});

// parameter style
app.get('/user/:id', function(req, res){
    res.send('user: ' + req.params.id);
});

app.get('/:number', function(req, res){
    res.send('number: ' + req.params.number);
});

// REGX style
app.get(/^\/ip?(?:\/(\d{2,3}))(?:\.(\d{2,3}))(?:\.(\d{2,3}))?)?/, function(req, res){
    res.send(req.params);
});

app.get('*', function(req, res){
    res.send('Page not found!', 404);
});

console.log('start express server\n');

==Express middleware==
```

Express 裡面有一個十分好用的應用概念稱為 `middleware`，可以透過 `middleware` 做出複雜的效果，同時上面也有介紹 `next` 方法參數傳遞，就是靠 `middleware` 的概念來傳遞參數，讓開發者可以明確的控制程式邏輯。

```
// .. create http server

app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.session());
```

上面都是一種 `middleware` 的使用方式，透過 `app.use` 方式裡面載入函式執行方法，回應函式會包含三個基本參數，`response`，`request`，`next`，其中 `next` 表示下一個 `middleware` 執行函式，同時會自動將預設三個參數繼續帶往下個函式執行，底下有個實驗，

上面的片段程式執行後，開啟瀏覽器，連結上 `localhost:1337` 會發現伺服器回應結果順序如下，

```
first middle ware
second middle ware
execute middle ware
end middleware function
```

從上面的結果可以得知，剛才設定的 `middleware` 都生效了，在 `app.use` 設定的 `middleware` 是所有 `url` 皆會執行方法，如果有指定特定方法，就可以使用 `app.get` 的 `middleware` 設定，在 `app.get` 函式的第二個參數，就可以帶入函式，或者是匿名函式，只要函式裡面最後會接受 `request`，`response`，`next` 這三個參數，同時也有正確指定 `next` 函式的執行時機，最後都會執行到最後一個方法，當然開發者也可以評估程式邏輯要執行到哪一個階段，讓邏輯可以更為分明。

==Express 路由應用 ==

在實際開發上可能會遇到需要使用參數等方式，混和變數一起使用，`express` 裡面提供了一個很棒的處理方法 `app.all` 這個方式，可以先採用基本路由配對，再將設定為每個不同的處理方式，開發者可以透過這個方式簡化自己的程式邏輯，

```
/**
 * @overview
 *
 * @author
 * @version 2012/02/26
 */
```



```
// create server.
var app = require('express').createServer(),
    port = 1337,
    users = [
      {name: 'Clonn'},
      {name: 'Chi'}
    ];

app.listen(port);

app.all('/user/:id/:op?', function(req, res, next){
  req.user = users[req.params.id];
  if (req.user) {
    next();
  } else {
    next(new Error('cannot find user ' + req.params.id));
  }
});

app.get('/user/:id', function(req, res){
  res.send('viewing ' + req.user.name);
});

app.get('/user/:id/edit', function(req, res){
  res.send('editing ' + req.user.name);
});

app.get('/user/:id/delete', function(req, res){
  res.send('deleting ' + req.user.name);
});

app.get('*', function(req, res){
  res.send('Page not found!', 404);
});

console.log('start express server\n');
```

內部宣告一組預設的使用者分別給予名稱設定，藉由 `app.all` 這個方法，可以先將路由雛形建立，再接下來設定 `app.get` 的路徑格式，只要符合格式就會分配進入對應的方法中，像上面的程式當中，如果使用者輸入路徑為/

`user/0`，除了執行 `app.all` 程式之後，執行 `next` 方法就會對應到路徑設定為 `/user/:id` 的這個方法當中。如果使用者輸入路徑為 `/user/0/edit`，就會執行到 `/user/:id/edit` 的對應方法。