Node.js Taiwan 社群協作中文電子書

Dca

Published with GitBook

Table of Contents

關於本書

這是一本關於 Node.js 技術的開放源碼電子書，我們使用 GitHub 維護電子書內容，並交由 GitBook 系統自動線上發佈。本書提供 PDF、EPUB、MOBI 及 HTML 等格式，您除了可以在網站檢視本書所有內容，也可以將電子書下載至閱讀器保存。

本書的線上閱讀網址，與 GitHub 資料同步更新。

http://book.nodejs.tw/

如果您想要取得本書的其他格式，可以從 GitBook 的電子書專頁下載最新版本。

https://www.gitbook.com/book/dca/nodejs-tw-wiki-book

本書適合 Node.js 初學者至進階開發者，也歡迎您在學習時一起參與本書內容撰寫。

編寫語法與規範

markdown 版本待補充

根目錄結構

- old -> 原先版本的內容
- book -> 改版至 gitbook 後的各章節內容

授權

作者

本書由 Node.js Taiwan 社群成員協作，以下名單依照字母排序。

- Caesar Chi (clonn)
- Fillano Feng (fillano)
- Kevin Shu (Kevin)
- lyhcode http://about.me/lyhcode

Node.js Taiwan 是一個自由開放的技術學習社群，我們歡迎您加入一起學習、研究及分享。

下載電子書

線上閱讀本書。

http://book.nodejs.tw/

PDF格式，適合一般電腦及7吋以上平板電腦閱讀

http://contpub.org/download/nodejs-wiki-book.pdf

EPUB格式，適合 iPad、iPhone 行動裝置閱讀

http://contpub.org/download/nodejs-wiki-book.epub

MOBI格式，適合 Kindle 電子書閱讀器

http://contpub.org/download/nodejs-wiki-book.mobi

原始碼

本書最新的原始碼（中文版）網址如下：

http://github.com/nodejs-tw/nodejs-wiki-book

精選文章收錄流程

精選文章的用意是鼓勵作者在自己的網誌發表 Node.js 教學，再由 Node.js Taiwan 社群挑選系列文章列入電子書的精選文集。

1. 將文章標題及連結貼到「精選文章」分類
2. 社群工作小組以 E-Mail 通知作者文章列入精選，並邀請將內文授權給 Node.js Taiwan 電子書分享
3. 作者同意後，由工作小組負責整理圖文，發佈至電子書
4. 以 E-Mail 寄出感謝函通知原作者文章已收錄，依作者意願調整文章內容
5. 於 Node.js Taiwan 首頁及粉絲專頁推薦作者的文章

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 Node.js 的催生者，目前任職於 Joyent 主機託管服務公司。他開發 Node.js 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，讓網站開發人員使用熟悉的 JavaScript 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，並且能在不同的 CommonJS 兼容 (compliant) JavaScript 執行環境中，使程式碼具有可攜性。

瀏覽器的 JavaScript 與實現 CommonJS 規範的 Node.js 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。而 CommonJS 的規範就填補了這種基礎函式庫功能的空缺，遵循 CommonJS 規範的 Node.js 可以直接使用 Sockets API 建立各種網路服務程式，也能夠讓更多同好基於 JavaScript 開發符合 Node.js 的外掛模組 (Module)。

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)。

Node.js 對於資源的調配有所不同，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。

基於 callback 特性，目前 Node.js 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 Node.js 設為內部核心網路服務之一。在 Node.js 也提供了外掛管理 (Node package management)，讓愛好 Node.js 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 v0.12.0，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://nodejs.tw>`，詢問對於 Node.js 相關問題。

Node.js 安裝與設定

本篇將講解如何在各個不同作業系統建立 Node.js 環境，目前 `Node.js v0.12.0` 版本環境架設方式相對來說都比以往單純許多。以下就各作業系統解說如何安裝 Node.js 。

Ubuntu Linux

使用 **nvm** 安裝 **(推薦)**

```
git clone git://github.com/creationix/nvm.git ~/.nvm
echo ". ~/.nvm/nvm.sh" >> ~/.bashrc
nvm install v0.12.0
nvm alias default v0.12.0
```

以上可參考： https://github.com/creationix/nvm/

透過系統套件管理安裝

不推薦此方式，如果有需要可參考官方教學：

https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

OSX

使用 **nvm** 安裝 **(推薦)**

```
git clone git://github.com/creationix/nvm.git ~/.nvm
echo ". ~/.nvm/nvm.sh" >> ~/.bashrc
nvm install v0.12.0
nvm alias default v0.12.0
```

以上可參考： https://github.com/creationix/nvm/

使用 **brew** 安裝

```
brew install node
```

Windows

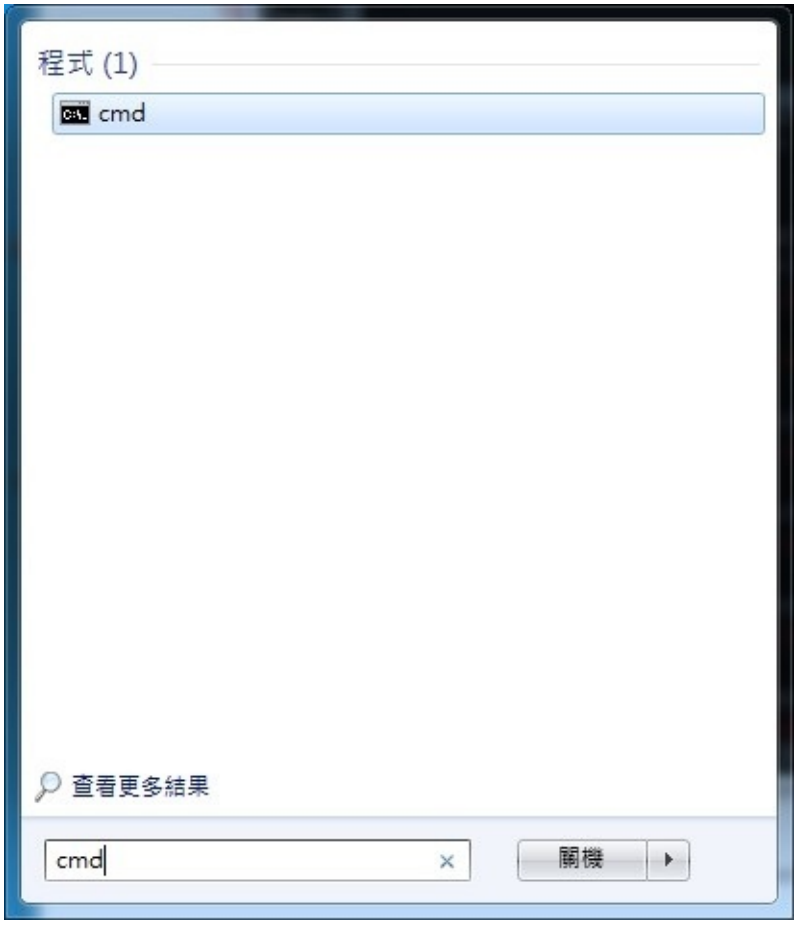
nodeJS 在 v0.6.0 版本之後開始正式支援 windows native，直接使用 node.exe 就可以執行程式，支援性完全與 linux 相同，更棒的部份就是不需經過編譯，經過下載之後，簡單設定完成，立即開發 node 程式。

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 輸指令如下

```
node -v
```

接著出現node.js 版本訊息畫面，表示安裝完成。



Node.js 基礎

前篇文章已經由介紹、安裝至設定都有完整介紹，Node.js 基礎 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，Node.js 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 Node.js 程式時常見的物件特性與使用方法。

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: 需要開通的阜號，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，之後許多除錯(debug) 都會用到 `console` 物件。

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

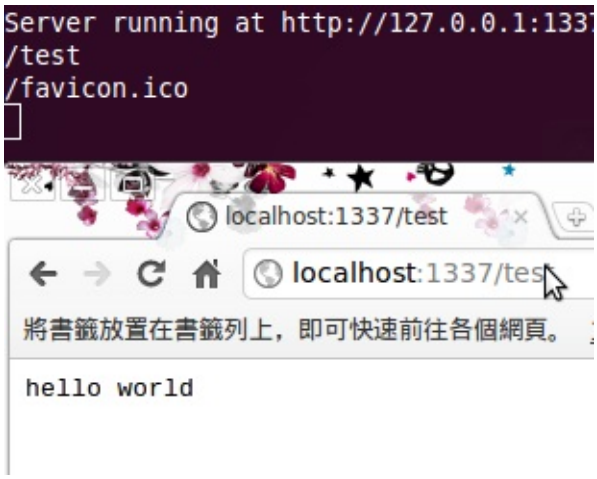
Node.js http 路徑建立

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，

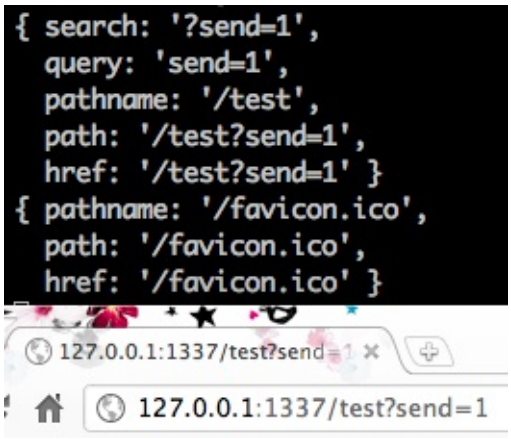


當在瀏覽器輸入 http://127.0.0.1:1337/test ，在伺服器端會收到兩個要求，一個是我們輸入的 /test 要求，另外一個則是 /favicon.ico。 /test 的路徑要求，http 伺服器本身需要經過程式設定才有辦法回應給瀏覽器端所需要的回應，在伺服器中所有的路徑要求都是需要被解析才有辦法取得資料。 從上面解說可以了解到在 Node.js 當中所有的路徑都需要經過設定，未經過設定的路由會讓瀏覽器無法取得任何資料導致錯誤頁面的發生，底下將會解說如何設定路由，同時避免發生錯誤情形。 先前 Node.js 程式需要增加一些修改，才能讓使用者透過瀏覽器，在不同路徑時有不同的結果。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');

server = http.createServer(function (req, res) {
  console.log(req.url);
  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。

Node.js 檔案讀取

前面已經介紹如何使用路由（rount）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 `./static/index.html` ，

./static/index.html

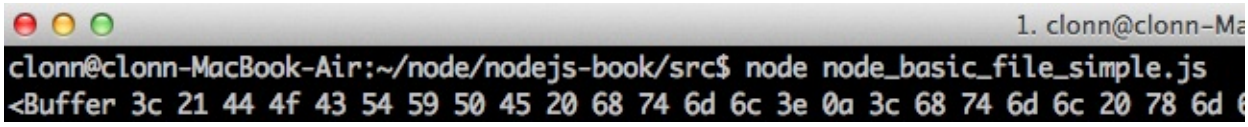
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="zh-TW">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Node.js index html file</title>
</head>
<body>
    <h1>Node.js index html file</h1>
</body>
</html>
```

另一個為 Node.js 程式，

node_basic_file_simple.js

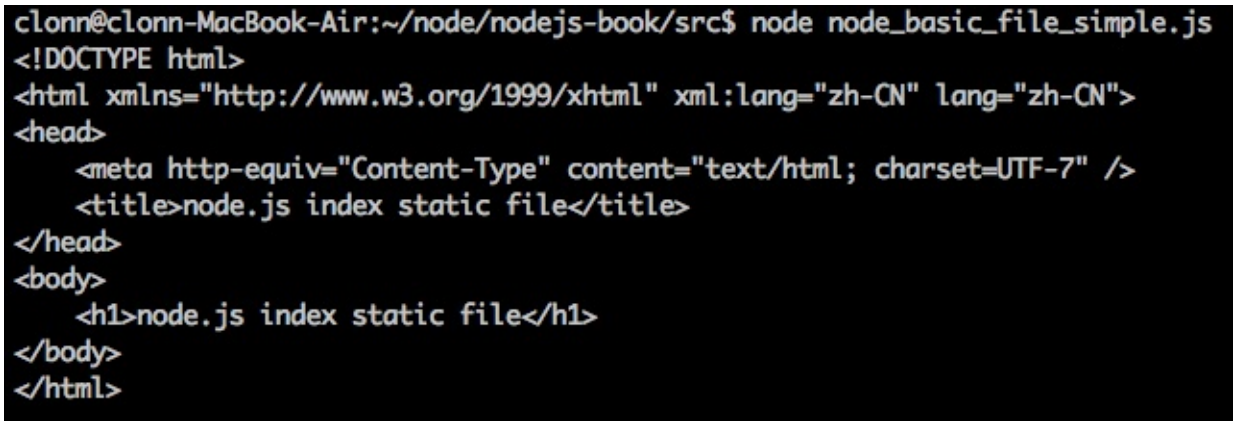
```
var fs = require("fs"),
    filename = "static/index.html",
    encode = "utf8";

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**



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，



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，

./static/index.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="zh-TW">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>node.js index html file</title>
</head>
<body>
    <h1>node.js index html file</h1>
</body>
</html>
```

./static/test.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="zh-TW">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>node.js test html file</title>
</head>
<body>
    <h1>node.js test html file</h1>
</body>
</html>
```

./static/static.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-TW" lang="zh-TW">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <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) {
  // something
});
```

`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，

./src/node_basic_file_http_static.js

```
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);
```

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。

```
var 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，

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

整個 Node.js http GET 參數完整擷取程式碼如下，

./src/node_basic_http_get.js

```
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' }
```

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 套件管理工具

npm 全名為 **N**ode **P**ackage **M**anager， 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。

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
```

.. topic:: 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 Unoffcial PPA <https://launchpad.net/~gias-kay-lee/+archive/npm>`_ 這個

repository，\ 即可使用 apt-get 完成 npm 安裝。

.. topic:: Ubuntu Linux 使用 apt-get 安裝 npm

```
::

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

.. topic:: 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。

.. topic:: 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：

.. topic:: 執行結果（範例）

```
::

    1.1.0-2
```

使用 NPM 安裝套件

npm 目前擁有超過 6000 種套件（packages），\ 可以在 `npm registry <http://search.npmjs.org/>`_ 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。

.. image:: ../images/zh-tw/node_npm_registry.png

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：

.. image:: ../images/zh-tw/node_npm_search.png

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，\ 需要注意執行權限與搜尋路徑的問題，\ 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
```

.. topic:: 執行結果（範例）

```
::

    CoffeeScript version 1.2.0
```

若未將 Node.js 套件安裝路徑加入環境變數 NODE_PATH，在引入時會回報錯誤。

.. topic:: 報錯範例

```
::

    module.js:340
        throw err;
              ^
    Error: Cannot find module 'express'
        at Function.Module._resolveFilename (module.js:338:15)
        at Function.Module._load (module.js:280:25)
        at Module.require (module.js:362:17)
        at require (module.js:378:17)
        at Object.<anonymous> (/home/clifflu/test/node.js/httpd/express.js:3:15)
        at Module._compile (module.js:449:26)
        at Object.Module._extensions..js (module.js:467:10)
        at Module.load (module.js:356:32)
        at Function.Module._load (module.js:312:12)
        at Module.runMain (module.js:492:10)
```

.. topic:: 使用 ubuntu PPA 安裝 Node.js 的設定範例

```
::
    echo 'NODE_PATH="/usr/lib/node_modules"' | sudo tee -a /etc/environment
```

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
```

.. topic:: 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
```

.. topic:: 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
```

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：

.. topic:: 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

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，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，

.. code-block::

```
npm install -g express
```

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，

.. code-block:: javascript

```
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，

.. code-block:: javascript

```
// ... 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，

.. code-block:: javascript

```
// ... 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，

.. code-block:: javascript

```
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})))?/, function(req, res){
    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，

.. code-block::

```
[
    "255",
    "255",
    "100",
    "10"
```

```
]
```

此章節全部範例程式碼如下，

.. literalinclude:: ../src/node_express_basic.js :language: javascript

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。

.. code-block:: javascript

```
// .. 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，

.. literalinclude:: ../src/node_express_middle_simple.js :language: javascript

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，

.. literalinclude:: ../src/node_express_basic_app.js :language: javascript

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。

Express GET 應用範例

我們準備一個使用GET方法傳送資料的表單。

.. literalinclude:: ../src/view/express_get_example_form.html :language: javascript

這個表單沒有什麼特別的地方，我們只需要看第9行，form使用的method是GET，然後action是"http://localhost:3000/Signup"，等一下我們要來撰寫/Signup這個URL Path的處理程式。

處理 *Signup* 行為

我們知道所謂的GET方法，會透過URL來把表單的值給帶過去，以上面的表單來說，到時候URL會以這樣的形式傳遞

::

```
http://localhost:3000/Signup?username=xxx&email=xxx
```

所以要能處理這樣的資料，必須有以下功能:

- 解析URL
- 辨別動作是Signup
- 解析出username和email

一旦能取得username和email的值，程式就能加以應用了。

處理 Signup 的程式碼雛形，

.. code-block:: javascript

```
// load module
var url  = require('url');

urlData = url.parse(req.url,true);
action = urlData.pathname;
res.writeHead(200, {"Content-Type":"text/html; charset=utf-8"});

if (action === "/Signup") {
   user = urlData.query;
   res.end("<h1>" + user.username + "歡迎您的加入</h1><p>我們已經將會員啟用信寄至" + user.email + "</p>");
}
```

首先需要加載 url module，它是用來協助我們解析URL的模組，接著使用 url.parse 方法，第一個傳入url 字串變數，也就是 req.url。另外第二個參數的用意是，設為ture則引進 querystring模組來協助處理，預設是false。它影響到的是 urlData.query，設為true會傳回物件，不然就只是一般的字串。url.parse 會將字串內容整理成一個物件，我們把它指定給 urlData。

action 變數作為記錄pathname，這是我們稍後要來判斷目前網頁的動作是什麼。接著先將 html 表頭資訊 (Header)準備好，再來判斷路徑邏輯，如果是 */Signup* 這個動作，就把urlData.query裡的資料指定給user，然後輸出user.username和 user.email，把使用者從表單註冊的資料顯示於頁面中。

最後進行程式測試，啟動 node.js 主程式之後，開啟瀏覽器就會看到表單，填寫完畢按下送出，就可以看到結果了。

完整 node.js 程式碼如下，

.. literalinclude:: ../src/node_express_get_form.js :language: javascript

Express POST 應用範例

一開始準備基本的 html 表單，傳送內容以 POST 方式， form 的 action 屬性設定為 POST，其餘 html 內容與前一個範例應用相同，

.. literalinclude:: ../src/view/express_post_example_form.html :language: javascript

node.js 的程式處理邏輯與前面 GET 範例類似，部分程式碼如下，

.. code-block:: javascript

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

if (action === "/Signup") {
    formData = '';
    req.on("data", function (data) {

        formData += data;

    });

    req.on("end", function () {
        user = qs.parse(formData);
        res.end("<h1>" + user.username + "歡迎您的加入</h1><p>我們已經將會員啟用信寄至" + user.email + "</p>");
    });
}
```

主要加入了'querystring' 這個moduel，方便我們等一下解析由表單POST回來的資料，另外加入一個formData的變數，用來搜集待等一下表單回傳的資料。前面的GET 範例，我們只從req 拿出url的資料，這次要在利用 req 身上的事件處理。

JavaScript在訂閱事件時使用addEventListener，而node.js使用的則是on。這邊加上了監聽 *data* 的事件，會在瀏覽器傳送資料到 Web Server時被執行，參數是它所接收到的資料，型態是字串。

接著再增加 *end* 的事件，當瀏覽器的請求事件結束時，它就會動作。

由於瀏覽器使用POST在上傳資料時，會將資料一塊塊地上傳，因為我們在監聽data事件時，透過formData 變數將它累加起來< 不過由於我們上傳的資料很少，一次就結束，不過如果日後需要傳的是資料比較大的檔案，這個累加動作就很重要。

當資料傳完，就進到end事件中，會用到 qs.parse來解析formData。formData的內容是字串，內容是：

::

```
username=wordsmith&email=wordsmith%40some.where
```

而qs.parse可以幫我們把這個querystring轉成物件的格式，也就是：

::

```
{username=wordsmith&email=wordsmith%40some.where}
```

一旦轉成物件並指定給user之後，其他的事情就和GET方法時操作的一樣，寫response的表頭，將內容回傳，並將user.username和user.email代入到內容中。

修改完成後，接著執行 node.js 程式，啟動 web server ，開啟瀏覽器進入表單測試看看，POST 的方式能否順利運作。

完整程式碼如下，

.. literalinclude:: ../src/node_express_post_form.js :language: javascript

Express AJAX 應用範例

在Node.js要使用Ajax傳送資料，並且與之互動，在接受資料的部份沒有太大的差別，client端不是用GET就是用POST來傳資料，重點在處理完後，用JSON格式回傳。當然Ajax不見得只傳JSON格式，有時是回傳一段HTML碼，不過後者對伺服器來說，基本上就和前兩篇沒有差別了。所以我們還是以回傳JSON做為這一回的主題。

這一回其實大多數的工作都會落在前端Ajax上面，前端要負責發送與接收資料，並在接收資料後，撤掉原先發送資料的表單，並將取得的資料，改成HTML格式之後，放上頁面。

首先先準備 HTML 靜態頁面資料，

.. literalinclude:: ../src/view/express_ajax_example_form.html :language: javascript

HTML 頁面上準備了一個表單，用來傳送註冊資料。接著直接引用了 Google CDN 來載入 jQuery，用來幫我們處理 Ajax 的工作，這次要傳送和接收的工作，很大的變動都在 HTML 頁面上的 JavaScript當中。我們要做的事有(相關 jQuery 處理這邊不多做贅述，指提起主要功能解說)：

- 用jQUery取得submit按鈕，綁定它的click動作
- 取得表單username和email的值，存放在user這個物件中
- 用jQuery的$.post方法，將user的資料傳到Server
- 一旦成功取得資料後，透過greet這個function，組成回報給user的訊息
  - 清空原本給使用者填資料的表單
  - 將Server回傳的username、email和id這3個資料，組成回應的訊息
  - 將訊息放到原本表單的位置

經過以上的處理後，一個Ajax的表單的基本功能已經完成。

接著進行 node.js 主要程式的編輯，部分程式碼如下，

.. code-block:: javascript

```
var fs   = require("fs"),
    qs   = require('querystring');

if (action === "/Signup") {
    formData = '';
    req.on("data", function (data) {

        formData += data;

    });

    req.on("end", function () {
        var msg;

        user = qs.parse(formData);
        user.id = "123456";
        msg = JSON.stringify(user);
        res.writeHead(200, {"Content-Type":"application/json; charset=utf-8","Content-Length":msg.length});
        res.end(msg);
    });
}
```

這裡的程式和前面 POST 範例，基本上大同小異，差別在：

- 幫user的資料加上id，隨意存放一些文字進去，讓Server回傳的資料多於Client端傳上來的，不然會覺得Server都沒做事。
- 增加了msg這個變數，存放將user物件JSON文字化的結果。JSON.stringify這個轉換函式是V8引擎所提供的，如果你好奇的話。
- 大重點來了，我們要告訴Client端，這次回傳的資料格式是JSON，所在Content-type和Content-Length要提供給Client。

Server很輕鬆就完成任務了，最後進行程式測試，啟動 node.js 主程式之後，開啟瀏覽器就會看到表單，填寫完畢按下送出，就可以看到結果了。

最後 node.js 本篇範例程式碼如下，

.. literalinclude:: ../src/node_express_ajax_form.js :language: javascript

原始資料提供

- [Node.JS初學者筆記(1)-用GET傳送資料] (http://ithelp.ithome.com.tw/question/10087402)
- [Node.JS初學者筆記(2)-用POST傳送資料] (http://ithelp.ithome.com.tw/question/10087489)
- [Node.JS初學者筆記(3)-用Ajax傳送資料] (http://ithelp.ithome.com.tw/question/10087627)

用 **Express** 和 **MongoDB** 寫一個 **todo list**

用 Express 和 MongoDB 寫一個 todo list

練習一種語言或是 framework 最快的入門方式就是寫一個 todo list 了. 他包含了基本的 C.R.U.D. (新增, 讀取, 更新, 刪除). 這篇文章將用 node.js 裡最通用的 framework Express 架構 application 和 MongoDB 來儲存資料.

原始檔

Live Demo http://dreamerslab.com/blog/tw/write-a-todo-list-with-express-and-mongodb/

功能

無需登入, 用 *cookie* 來辨別每一問使用者 可以新增, 讀取, 更新, 刪除待辦事項(todo item)

安裝

開發環境 開始之前請確定你已經安裝了 node.js, Express 和 MongoDB, 如果沒有可以參考下列文章.

```
* `How to setup a node.js development environment on Mac OSX Lion <http://dreamerslab.com/blog/tw/how-to-setup-a-node-j
* `How to setup a node.js development environment on Ubuntu 11.04 <http://dreamerslab.com/blog/tw/how-to-setup-a-node-j
* `How to setup a node.js development environment on Windows <http://dreamerslab.com/blog/tw/how-to-setup-a-node-js-dev
```

node.js 套件

參考文件： `npm basic commands<http://dreamerslab.com/blog/en/npm-basic-commands/>`

- 安裝 Express

::

```
$ npm install express@2.5.11 -g
```

這個練習裡我們用 Mongoose 這個 ORM. 為何會需要一個必須定義 schema 的 ORM 來操作一個 schema-less 的資料庫呢? 原因是在一般的網站資料結構的關聯, 驗證都是必須處理的問題. Mongoose 在這方面可以幫你省去很多功夫. 我們會在後面才看如何安裝.

步驟

用 Express 的 command line 工具幫我們生成一個 project 雛形 預設的 template engine 是 jade, 在這裡我們改用比較平易近人的 ejs.

::

```
$ express todo -t ejs

create : todo
create : todo/package.json
create : todo/app.js
create : todo/public
create : todo/public/javascripts
create : todo/public/images
create : todo/public/stylesheets
create : todo/public/stylesheets/style.css
create : todo/routes
create : todo/routes/index.js
create : todo/views
create : todo/views/layout.ejs
create : todo/views/index.ejs
```

在專案根目錄增加 .gitignore 檔案

::

```
.DS_Store
node_modules
*.sock
```

將 connect 以及 mongoose 加入 dependencies，編輯 package.json

::

```
{
  "name"         : "todo",
  "version"      : "0.0.1",
  "private"      : true,
  "dependencies" : {
    "connect"  : "1.8.7",
    "express"  : "2.5.11",
    "ejs"      : "0.8.3",
    "mongoose" : "3.2.0"
  }
}
```

安裝 dependencies

::

```
$ cd todo && npm install -l
```

Hello world 開啟 express server 然後打開瀏覽器瀏覽 127.0.0.1:3000 就會看到歡迎頁面.

::

```
$ node app.js
```

Project 檔案結構

::

```
todo
|-- node_modules
|   |-- ejs
|   |-- express
```

```
|   `-- mongoose
|
|-- public
|   |-- images
|   |-- javascripts
|   `-- stylesheets
|       |-- style.css
|
|-- routes
|   `-- index.js
|
|-- views
|   |-- index.ejs
|   `-- layout.ejs
|
|-- .gitignore
|
|-- app.js
|
`-- package.json
```

- node_modules - 包含所有 project 相關套件.
- public - 包含所有靜態檔案.
- routes - 所有動作包含商業邏輯.
- views - 包含 action views, partials 還有 layouts.
- app.js - 包含設定, middlewares, 和 routes 的分配.
- package.json - 相關套件的設定檔.

MongoDB 以及 Mongoose 設定

在 Ubuntu 上 MongoDB 開機後便會自動開啟. 在 Mac 上你需要手動輸入下面的指令.

::

```
$ mongod --dbpath /usr/local/db
```

在根目錄下新增一個檔案叫做 db.js 來設定 MongoDB 和定義 schema.

.. code-block:: js

```
var mongoose = require( 'mongoose' );
var Schema   = mongoose.Schema;

var Todo = new Schema({
    user_id    : String,
    content    : String,
    updated_at : Date
});

mongoose.model( 'Todo', Todo );

mongoose.connect( 'mongodb://localhost/express-todo' );
```

在 app.js 裡 require.

::

```
require( './db' );
```

將 require routes 移動到 db config 之後.

.. code-block:: js

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

var app = module.exports = express.createServer();

// 設定 mongoose
require( './db' );

// 設定 middleware
// 移除 methodOverride, 新增 favicon, logger 並將 static middleware 往上移
app.configure( function (){
  app.set( 'views', __dirname + '/views' );
  app.set( 'view engine', 'ejs' );
  app.use( express.favicon());
  app.use( express.static( __dirname + '/public' ));
  app.use( express.logger());
  app.use( express.bodyParser());
  app.use( app.router );
});

app.configure( 'development', function (){
  app.use( express.errorHandler({ dumpExceptions : true, showStack : true }));
});

app.configure( 'production', function (){
  app.use( express.errorHandler());
});

// Routes
var routes = require( './routes' );

app.get( '/', routes.index );

app.listen( 3000, function (){
  console.log( 'Express server listening on port %d in %s mode', app.address().port, app.settings.env );
});
```

修改 project title "routes/index.js"

.. code-block:: js

```
exports.index = function ( req, res ){
  res.render( 'index', { title : 'Express Todo Example' });
};
```

修改 **index view**

我們需要一個 text input 來新增待辦事項. 在這裡我們用 POST form 來傳送資料. views/index.ejs

::

```
<h1><%= title %></h1>
<form action="/create" method="post" accept-charset="utf-8">
  <input type="text" name="content" />
</form>
```

新增待辦事項以及存檔，routes/index.js，首先先 require mongoose 和 Todo model.

.. code-block:: js

```
var mongoose = require( 'mongoose' );
var Todo     = mongoose.model( 'Todo' );
```

新增成功後將頁面導回首頁.

.. code-block:: js

```
exports.create = function ( req, res ){
  new Todo({
    content    : req.body.content,
    updated_at : Date.now()
  }).save( function( err, todo, count ){
    res.redirect( '/' );
  });
};
```

將這個新增的動作加到 routes 裡.

app.js

.. code-block:: js

```
// 新增下列語法到 routes
app.post( '/create', routes.create );
```

顯示待辦事項 routes/index.js

.. code-block:: js

```
// 查詢資料庫來取得所有待辦是事項.
exports.index = function ( req, res ){
  Todo.find( function ( err, todos, count ){
    res.render( 'index', {
        title : 'Express Todo Example',
        todos : todos
    });
  });
};
```

views/index.ejs

.. code-block:: js

```
// 在最下面跑回圈來秀出所有待辦事項.
<% todos.forEach( function( todo ){ %>
  <p><%= todo.content %></p>
<% }); %>
```

刪除待辦事項 在每一個待辦事項的旁邊加一個刪除的連結. routes/index.js

.. code-block:: js

```
// 根据待辦事項的 id 来移除他
exports.destroy = function ( req, res ){
  Todo.findById( req.params.id, function ( err, todo ){
    todo.remove( function ( err, todo ){
      res.redirect( '/' );
    });
  });
};
```

views/index.ejs

::

```
// 在迴圈裡加一個刪除連結
<% todos.forEach( function ( todo ){ %>
  <p>
    <span>
      <%= todo.content %>
    </code>
    <span>
      <a href="/destroy/<%= todo._id %>" title="Delete this todo item">Delete</a>
    </code>
  </p>
<% }); %>
```

將這個刪除的動作加到 routes 裡. app.js

.. code-block:: js

```
// 新增下列語法到 routes
app.get( '/destroy/:id', routes.destroy );
```

編輯待辦事項 當滑鼠點擊待辦事項時將他轉成一個 text input. routes/index.js

.. code-block:: js

```
exports.edit = function ( req, res ){
  Todo.find( function ( err, todos ){
    res.render( 'edit', {
        title   : 'Express Todo Example',
        todos   : todos,
        current : req.params.id
    });
  });
};
```

Edit view 基本上和 index view 差不多, 唯一的不同是在選取的那個待辦事項變成 text input. views/edit.ejs

::

```
<h1><%= title %></h1>
<form action="/create" method="post" accept-charset="utf-8">
  <input type="text" name="content" />
</form>

<% todos.forEach( function ( todo ){ %>
  <p>
    <span>
      <% if( todo._id == current ){ %>
      <form action="/update/<%= todo._id %>" method="post" accept-charset="utf-8">
        <input type="text" name="content" value="<%= todo.content %>" />
      </form>
      <% }else{ %>
        <a href="/edit/<%= todo._id %>" title="Update this todo item"><%= todo.content %></a>
      <% } %>
    </code>
    <span>
      <a href="/destroy/<%= todo._id %>" title="Delete this todo item">Delete</a>
    </code>
  </p>
<% }); %>
```

將待辦事項包在一個 link 裡, link 可以連到 edit 動作. views/index.ejs

::

```
<h1><%= title %></h1>
<form action="/create" method="post" accept-charset="utf-8">
  <input type="text" name="content" />
</form>

<% todos.forEach( function ( todo ){ %>
  <p>
    <span>
      <a href="/edit/<%= todo._id %>" title="Update this todo item"><%= todo.content %></a>
    </code>
    <span>
      <a href="/destroy/<%= todo._id %>" title="Delete this todo item">Delete</a>
    </code>
  </p>
<% }); %>
```

將這個編輯的動作加到 routes 裡. app.js

::

```
// 新增下列語法到 routes
app.get( '/edit/:id', routes.edit );
```

更新待辦事項 新增一個 update 動作來更新待辦事項. routes/index.js

.. code-block:: js

```
// 結束後重新導回首頁
exports.update = function ( req, res ){
  Todo.findById( req.params.id, function ( err, todo ){
    todo.content    = req.body.content;
    todo.updated_at = Date.now();
    todo.save( function ( err, todo, count ){
      res.redirect( '/' );
    });
  });
};
```

將這個更新的動作加到 routes 裡. app.js

::

```
// 新增下列語法到 routes
app.post( '/update/:id', routes.update );
```

排序 現在待辦事項是最早產生的排最前面, 我們要將他改為最晚產生的放最前面. routes/index.js

.. code-block:: js

```
exports.index = function ( req, res ){
  Todo.
    find().
    sort( '-updated_at' ).
    exec( function ( err, todos ){
      res.render( 'index', {
          title : 'Express Todo Example',
          todos : todos
      });
    });
};

exports.edit = function ( req, res ){
  Todo.
    find().
    sort( '-updated_at' ).
```

```
    exec( function ( err, todos ){
      res.render( 'edit', {
          title   : 'Express Todo Example',
          todos   : todos,
          current : req.params.id
      });
    });
};
```

多重使用者 現在所有使用者看到的都是同一份資料. 意思就是說每一個人的 todo list 都長得一樣, 資料都有可能被其他人修改. 我們可以用 cookie 來記錄使用者資訊讓每個人有自己的 todo list. Express 已經有內建的 cookie, 只要在 app.js 新增一個 middleware 就好. 另外我們也會需要新增一個依據 cookie 來抓取當下的使用者的 middleware. app.js

.. code-block:: js

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

var app = module.exports = express.createServer();

// 設定 mongoose
require( './db' );

// 將 routes 移到 middlewares 設定上面
var routes = require( './routes' );

// 設定 middleware
// 移除 methodOverride, 新增 favicon, logger 並將 static middleware 往上移
app.configure( function (){
  app.set( 'views', __dirname + '/views' );
  app.set( 'view engine', 'ejs' );
  app.use( express.favicon());
  app.use( express.static( __dirname + '/public' ));
  app.use( express.logger());
  app.use( express.cookieParser());
  app.use( express.bodyParser());
  app.use( routes.current_user );
  app.use( app.router );
});

app.configure( 'development', function (){
  app.use( express.errorHandler({ dumpExceptions : true, showStack : true }));
});

app.configure( 'production', function (){
  app.use( express.errorHandler());
});

// Routes
app.get( '/', routes.index );
app.post( '/create', routes.create );
app.get( '/destroy/:id', routes.destroy );
app.get( '/edit/:id', routes.edit );
app.post( '/update/:id', routes.update );

app.listen( 3000, function (){
  console.log( 'Express server listening on port %d in %s mode', app.address().port, app.settings.env );
});
```

routes/index.js

.. code-block:: js

```
var mongoose = require( 'mongoose' );
var Todo     = mongoose.model( 'Todo' );
var utils    = require( 'connect' ).utils;

exports.index = function ( req, res, next ){
  Todo.
    find({ user_id : req.cookies.user_id }).
    sort( '-updated_at' ).
    exec( function ( err, todos, count ){
      if( err ) return next( err );
```

```
      res.render( 'index', {
          title : 'Express Todo Example',
          todos : todos
      });
    });
};

exports.create = function ( req, res, next ){
  new Todo({
      user_id    : req.cookies.user_id,
      content    : req.body.content,
      updated_at : Date.now()
  }).save( function ( err, todo, count ){
    if( err ) return next( err );

    res.redirect( '/' );
  });
};

exports.destroy = function ( req, res, next ){
  Todo.findById( req.params.id, function ( err, todo ){
    if( todo.user_id !== req.cookies.user_id ){
      return utils.forbidden( res );
    }

    todo.remove( function ( err, todo ){
      if( err ) return next( err );

      res.redirect( '/' );
    });
  });
};

exports.edit = function( req, res, next ){
  Todo.
    find({ user_id : req.cookies.user_id }).
    sort( '-updated_at' ).
    exec( function ( err, todos ){
      if( err ) return next( err );

      res.render( 'edit', {
        title   : 'Express Todo Example',
        todos   : todos,
        current : req.params.id
      });
    });
};

exports.update = function( req, res, next ){
  Todo.findById( req.params.id, function ( err, todo ){
    if( todo.user_id !== req.cookies.user_id ){
      return utils.forbidden( res );
    }

    todo.content    = req.body.content;
    todo.updated_at = Date.now();
    todo.save( function ( err, todo, count ){
      if( err ) return next( err );

      res.redirect( '/' );
    });
  });
};

// ** 注意!! express 會將 cookie key 轉成小寫 **
exports.current_user = function ( req, res, next ){
  if( !req.cookies.user_id ){
    res.cookie( 'user_id', utils.uid( 32 ));
  }

  next();
};
```

Error handling

要處理錯誤我們需要新增 next 參數到每個 action 裡. 一旦錯誤發生遍將他傳給下一個 middleware 去處理. routes/index.js

.. code-block:: js

```
... function ( req, res, next ){
  // ...
};

...( function( err, todo, count ){
  if( err ) return next( err );

  // ...
});
```

Run application

::

```
$ node app.js
```

到此為止我們已經完成了大部分的功能了. 原始碼裡有多加了一點 css 讓他看起來更美觀. 趕快開啟你的 server 來玩玩看吧 :)

附錄

Node.js 與 JavaScript

附錄 Node.js 與 JavaScript

JavaScript 基本型態

JavaScript 有以下幾種基本型態。

- Boolean
- Number
- String
- null
- undefined

變數宣告的方式，就是使用 var，結尾使用『;』，如果需要連續宣告變數，可以使用 『,』 做為連結符號。

::

```
// 宣告 x 為 123, 數字型態
var x=123;

// 宣告 a 為456, b 為 'abc' 字串型態
var a=456,
    b='abc';
```

布林值

布林，就只有兩種數值, true, false

::

```
var a=true,
    b=false;
```

數字型別

Number 數字型別，可以分為整數，浮點數兩種，

::

```
var a=123,
    b=123.456;
```

字串型別

字串，可以是一個字，或者是一連串的字，可以使用 " 或 "" 做為字串的值。 (盡量使用雙引號來表達字串，因為在node裡不會把單引號框住的文字當作字串解讀)

::

```
var a="a",
    a='abc';
```

運算子

基本介紹就是 +, -, *, / 邏輯運算就是 && (and), || (or), ^ (xor), 比較式就是 >, <, !=, !==, ==, ===, >=, <=

判斷式

這邊突然離題，加入判斷式來插花，判斷就是 if，整個架構就是，

::

```
if (判斷a) {
    // 判斷a 成立的話，執行此區域指令
} else if (判斷b) {
    // 判斷a 不成立，但是 判斷b 成立，執行此區域指令
} else {
    // 其餘的事情在這邊處理
}
```

整體架構就如上面描述，非 a 即 b的狀態，會掉進去任何一個區域裡面。整體的判斷能夠成立，只要判斷轉型成 Boolean 之後為 true，就會成立。大家可以這樣子測試，

```
Boolean(判斷);
```

應用

會突然講 if 判斷式，因為，前面有提到 Number, String 兩種型態，但是如果我們測試一下，新增一個 test.js

::

```
var a=123,
    b='123';

if (a == b) {
    console.log('ok');
}
```

編輯 test.js 完成之後，執行底下指令

::

```
node test.js
// print: ok
```

輸出結果為 ok。

這個結果是有點迥異， a 為 Number, b 為 String 型態，兩者相比較，應該是為 false 才對，到底發生什麼事情？ 這其中原因是，在判斷式中使用了 == ， JavaScript 編譯器，會自動去轉換變數型態，再進行比對，因此 a == b 就會成立，如果不希望

轉型產生，就必須要使用 === 做為判斷。

:: if (a === b) { console.log('ok); } else { console.log('not ok'); } // print: not ok

轉型

如果今天需要將字串，轉換成 Number 的時候，可以使用 parseInt, parseFloat 的方法來進行轉換，

::

```
var a='123';
console.log(typeof parseInt(a, 10));
```

使用 typeof 方法取得資料經過轉換後的結果，會取得，

::

```
number
```

要注意的是，記得 parseInt 後面要加上進位符號，以免造成遺憾，在這邊使用的是 10 進位。

Null & undefined 型態差異

空無是一種很奇妙的狀態，在 JavaScript 裡面，null, undefined 是一種奇妙的東西。今天來探討什麼是 null ， 什麼是 undefined.

null

變數要經過宣告，賦予 null ，才會形成 null 型態。

::

```
var a=null;
```

null 在 JavaScript 中表示一個空值。

undefined

從字面上就表示目前未定義，只要一個變數在初始的時候未給予任何值的時候，就會產生 undefined

::

```
var a;

console.log(a);

// print : undefined
```

這個時候 a 就是屬於 undefined 的狀態。另外一種狀況就是當 Object 被刪除的時候。

::

```
var a = {};
delete a;
console.log(a);

//print: undefined.
```

Object 在之後會介紹，先記住有這個東西。而使用 delete 的時候，就可以讓這個 Object 被刪除，就會得到結果為 undefined.

兩者比較

null, undefined 在本質上差異並不大，不過實質上兩者並不同，如果硬是要比較，建議使用 === 來做為判斷標準，避免 null, undefined 這兩者被強制轉型。

::

```
var a=null,
    b;

if (a === b) {
    console.log('same');
} else {
    console.log('different');
}

//print: different
```

從 typeof 也可以看到兩者本質上的差異，

::

```
typeof null;
//print: 'object'

typeof undefined;
//print: 'undefined'
```

null 本質上是屬於 object, 而 undefined 本質上屬於 undefined ， 意味著在 undefined 的狀態下，都是屬於未定義。

如果用判斷式來決定，會發現另外一種狀態

::

```
Boolean(null);
// false

Boolean(undefined);
// false
```

可以觀察到，如果一個變數值為 null, undefined 的狀態下，都是屬於 false。

這樣說明應該幫助到大家了解，其實要判斷一個物件、屬性是否存在，只需要使用 if

::

```
var a;
```

```
if (!a) {
    console.log('a is not existed');
}

//print: a is not existed
```

a 為 undefined 由判斷式來決定，是屬於 False 的狀態。

JavaScript Array

陣列也是屬於 JavaScript 的原生物件之一，在實際開發會有許多時候需要使用 Array 的方法，先來介紹一下陣列要怎麼宣告。

陣列宣告

宣告方式,

.. code-block:: js

```
var a=['a', 'b', 'c'];

var a=new Array('a', 'b', 'c');
```

以上這兩種方式都可以宣告成陣列，接著我們將 a 這個變數印出來看一下,

.. code-block:: js

```
console.log(a);
//print: [0, 1, 2]
```

Array 的排列指標從 0 開始，像上面的例子來說， a 的指標就有三個，0, 1, 2，如果要印出特定的某個陣列數值，使用方法,

.. code-block:: js

```
console.log(a[1]);
//print: b
```

如果要判斷一個變數是不是 Array 最簡單的方式就是直接使用 Array 的原生方法,

.. code-block:: js

```
var a=['a', 'b', 'c'];

console.log(Array.isArray(a));
//print: true

var b='a';
console.log(Array.isArray(b));
//print: false
```

如果要取得陣列變數的長度可以直接使用,

.. code-block:: js

```
console.log(a.length);
```

length 為一個常數，型態為 Number，會列出目前陣列的長度。

pop, shift

以前面所宣告的陣列為範例，

.. code-block:: js

```
var a=['a', 'b', 'c'];
```

使用 pop 可以從最後面取出陣列的最後一個值。

.. code-block:: js

```
console.log(a.pop());
//print: c

console.log(a.length);
//print: 2
```

同時也可以注意到，使用 pop 這個方法之後，陣列的長度內容也會被輸出。另外一個跟 pop 很像的方式就是 shift，

.. code-block:: js

```
console.log(a.shift());
//print: a

console.log(a.length);
//print: 1
```

shift 跟 pop 最大的差異，就是從最前面將數值取出，同時也會讓呼叫的陣列少一個數組。

slice

前面提到 pop, shift 就不得不說一下 slice，使用方式，

.. code-block:: js

```
console.log(a.slice(1,3));
//print: 'b', 'c'
```

第一個參數為起始指標，第二個參數為結束指標，會將這個陣列進行切割，變成一個新的陣列型態。 如果需要給予新的變數，就可以這樣子做，完整的範例。

.. code-block:: js

```
var a=['a', 'b', 'c'];

var b=a.slice(1,3);

console.log(b);
```

```
//print: 'b', 'c'
```

concat

concat 這個方法，可以將兩個 Array 組合起來，

.. code-block:: js

```
var a=['a'];

var b=['b', 'c'];

console.log(a.concat(b));
//print: 'a', 'b', 'c'
```

concat 會將陣列組合，之後變成全新的數組，如果以例子來說，a 陣列希望變成 ['a', 'b', 'c']，可以重新將數值分配給 a，範例來說

.. code-block:: js

```
a = a.concat(b);
```

Iterator

陣列資料，必須要有 Iterator，將資料巡迴一次，通常是使用迴圈的方式，

.. code-block:: js

```
var a=['a', 'b', 'c'];

for(var i=0; i < a.length; i++) {
    console.log(a[i]);
}

//print: a
//       b
//       c
```

事實上可以用更簡單的方式進行，

.. code-block:: js

```
var a=['a', 'b', 'c'];

a.forEach(function (val, idx) {
    console.log(val, idx);
});

/*
print:
a, 0
b, 1
c, 2
*/
```

在 Array 裡面可以使用 foreach 的方式進行 iterator， 裡面給予的 function (匿名函式)，第一個變數為 Array 的 Value, 第二個變數為 Array 的指標。

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。

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操作，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。

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：

.. code-block:: js

```
function outter(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_variable1 + 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_variable1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。

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：

.. code-block:: js

```
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：

.. code-block:: js

```
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：

.. code-block:: js

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

然後把程式改成：

.. code-block:: js

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

var server = http.createServer(function(request, response) {
 this.emit('init', request, 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：

.. code-block:: js

```
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。

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：

.. code-block:: html

```
<html>
<body>
<div id="panel" style="visibility:hidden"></div>
</body>
</html>
<script>
var request = new XMLHttpRequest();
request.open('GET', 'test749.txt?timestamp='+new Date().getTime(), true);
request.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;
 document.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。

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：

.. code-block:: js

```
<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：

.. code-block:: js

```
<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：

.. code-block:: js

```
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：

.. code-block:: js

```
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...

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：

.. code-block:: js

```
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：

.. code-block:: js

```
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：

.. code-block:: js

```
<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;
 waitback(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);
setTimeout(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

如果想不透過其他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 Jouhier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）：

.. code-block:: js

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

.. code-block:: js

```
var serial = require('./serial'),
    fs = require('fs'),
    path = './dclient.js',
    cb = serial([
    function(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：

.. code-block:: js

```
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:]

.. code-block:: js

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

dnode.connect(8000, 'localhost',  function(remote) {
    remote.restart(function(str) {
        console.log(str);
        process.exit();
    });
});
```

[nested callbacks:]

.. code-block:: js

```
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 。